

차례

기본사항들	1
1 장 소개	3
알고리즘 / 4	
본 교재 주제들의 개요 / 5	
2 장 C++와 C	9
예제 : 유클리드(Euclid) 알고리즘 / 10	
데이터 형(Types of Data) / 12	
입/출력 / 13	
결론 / 15	
연습문제 / 17	
3 장 기초적인 자료구조	19
배열(Arrays) / 19	
연결 리스트(linked list) / 22	
기억장치 할당(Storage Allocation) / 27	
스택 삽입(Pushdown Stacks) / 30	
스택들의 연결 리스트 구현 / 34	
큐(Queues) / 36	
추상적이고 구체적인 자료 형 / 38	
연습문제 / 41	
4 장 트리	43
용어 풀이(Glossary) / 44	
성질(properties) / 47	
이진 트리의 표현 / 49	
포리스트의 표현(Representing Forests) / 52	
트리 운행(Traversing Trees) / 54	
연습문제 / 60	

5 장 재귀 61

- 재발생(recurrence) / 62
- 분할-정복(Divide-and-Conquer) / 64
- 재귀적으로 트리를 운행 / 70
- 재귀를 제거 / 72
- 전망(Perspective) / 76
- 연습문제 / 77

6 장 알고리즘 분석 79

- 구조(Framework) / 80
- 알고리즘 분류 / 81
- 계산 복잡도(Computational Complexity) / 84
- 평균 경우 분석 / 87
- 근사치와 점진적 결과 / 87
- 기본적인 재발생(basic recurrences) / 89
- 통찰 / 92
- 연습문제 / 94

7 장 알고리즘 구현 95

- 알고리즘 선택 / 96
- 실험적 분석 / 97
- 프로그램 최적화 / 99
- 알고리즘과 시스템 / 101
- 연습문제 / 103

정렬 알고리즘 105

8 장 기초적인 정렬 방법들 107

- 게임의 규칙 / 108
- 선택 정렬(Selection Sort) / 111
- 삽입 정렬(Insertion Sort) / 113
- 버블 정렬(Bubble Sort) / 116
- 기초적인 정렬들의 활용도 특징 / 116
- 큰 레코드로된 파일을 정렬 / 120
- 셸 정렬(Shellsort) / 124

분배 계수기(Distribution Counting) / 128	
연습문제 / 131	
9 장 퀵 정렬	133
기본 알고리즘 / 134	
퀵 정렬의 활용도 특징 / 139	
재귀를 제거(Removing Recursion) / 141	
적은 부분 파일들 / 144	
세 가지중의 중위수를 분할(Median-of-Three Partitioning) / 145	
선택 / 146	
연습문제 / 150	
10 장 기수 정렬	151
비트(bits) / 152	
기수 교환 정렬(Radix Exchange Sort) / 153	
일직선상의 기수 정렬(Straight Radix Sort) / 158	
기수 정렬의 활용도 특성 / 160	
선형 정렬(Linear Sort) / 161	
연습문제 / 164	
11 장 우선순위	165
기초적인 구현들 / 167	
힙프 자료구조 / 169	
힙프에서의 알고리즘 / 170	
힙프 정렬(Heapsort) / 175	
간접 힙프(Indirect Heaps) / 181	
고급 구현 / 183	
연습문제 / 184	
12 장 병합 정렬	185
병합 / 186	
병합 정렬 / 188	
리스트 병합 정렬 / 189	
상향식 병합 정렬 / 191	
활용도 특징 / 195	
최적화된 구현 / 197	

재 방문된 재귀 / 198

연습 문제 / 200

13 장 외부 정렬 201

정렬-병합 / 202

균형화된 다중 방법 병합(Balanced Multiway Merging) / 203

대체 선택(Replacement Selection) / 205

실제적인 고려들 / 208

다중 단계 병합(Polyphase Merging) / 210

더 쉬운 방법 / 212

연습 문제 / 214

검색 알고리즘 215

14 장 기초적인 검색 방법들 217

순차적 검색(Sequential Searching) / 219

이진 검색(Binary Search) / 223

이진 트리 검색(Binary Tree Search) / 228

삭제(Deletion) / 235

간접 이진 검색 트리들 / 237

연습 문제 / 239

15 장 균형 트리 241

하향식 2-3-4 트리들 / 242

적색-흑색(Red-Black) 트리 / 246

다른 알고리즘 / 256

연습문제 / 257

16 장 해싱 259

해시 함수(Hash Functions) / 260

분리된 연쇄(Separate Chaining) / 262

선형 조사(Linear Probing) / 265

이중 해싱(Double Hashing) / 268

통찰 / 271

연습문제 / 273

17 장 기수 검색	275
디지털 검색 트리(Digital Search Trees) /	276
기수 검색 트라이(Radix Search Tries) /	279
다중 방법 기수 검색(Multiway Radix Searching) /	282
패트리시아(Patricia) /	284
연습문제 /	289
18 장 외부 검색	291
색인 순차 접근(Indexed Sequential Access) /	292
B-트리(B-trees) /	294
확장 가능 해싱(Extendible Hashing) /	299
가상 기억장치(Virtual Memory) /	305
연습문제 /	306
스트링 처리	307
19 장 스트링 검색	309
간단한 역사 /	310
주먹구구식 알고리즘 /	311
Knuth-Morris-Pratt 알고리즘 /	313
Boyer-Moore 알고리즘 /	319
Rabin-Karp 알고리즘 /	323
다중 검색(Multiple Searches) /	325
연습문제 /	326
20 장 패턴 일치	327
패턴을 기술 /	328
패턴 일치 기계 /	329
기계를 표현 /	333
기계를 모의 실험 /	334
연습문제 /	339
21 장 파싱	341
문맥 자유 문법들(Context-Free Grammars) /	342
하향식 파싱(Top-Down Parsing) /	345

상향식 파싱 /	348
컴파일러(Compilers) /	349
컴파일러-컴파일러 /	353
연습문제 /	355

22 장 파일 압축 357

반복문자-길이의 코드화(Run-Length Encoding) /	358
가변문자-길이 코드화(Variable-Length Encoding) /	361
Huffman 코드의 구성(Building the Huffman Code) /	363
구현(Implementation) /	366
연습문제 /	370

23 장 암호학 371

게임 규칙(Rules of the Game) /	372
간단한 방법들(Simple Methods) /	374
암호 작성/해독기들(Encryption/Description Machines) /	376
공개키 암호 시스템들(Public-key Cryptosystems) /	377
연습문제 /	381

기하학적 알고리즘 383

24 장 기초적인 기하학적 방법 385

점, 선 및 다각형(Points, Lines, and Polygons) /	386
선분 교차 (Line Segment Intersection) /	388
단순 폐경로 (Simple Closed Path) /	390
다각형 안에 포함됨(Inclusion in a Polygon) /	392
전망 (Perspective) /	394
연습문제 /	396

25 장 볼록 외곽 찾기 397

게임 규칙(Rules of the Game) /	399
패키지 포장(Package-Wrapping) /	400
Graham 스캔 (The Graham Scan) /	403
내부제거(Interior Elimination) /	408
성능 문제(performance Issues) /	409
연습문제 /	412

26 장	구간 검색	413
	기본 방법(Elementary Methods) /	415
	격자법(Grid Method) /	417
	이차원 트리(Two-Dimensional Tree) /	421
	다차원 구간 검색(Multidimensional Range Searching) /	426
	연습문제 /	428
27 장	기하학적 교차	429
	수평-수직선들(Horizontal and Vertical Line) /	430
	구현(Implementation) /	434
	일반적 선분 교차(General Line Intersection) /	437
	연습문제 /	441
28 장	가장 인접한 점 문제들	443
	가장 인접한 쌍 문제(Closest-Pair Problem) /	444
	Voronoi 다이어그램(Voronoi Diagram) /	450
	연습문제 /	455
그래프 알고리즘		457
29 장	기본 그래프 알고리즘들	459
	용어풀이(Glossary) /	460
	표현(Representation) /	463
	깊이-우선 검색(Depth-First Search) /	468
	비재귀적 깊이-우선 검색(Nonrecursive Depth-First Search) /	474
	너비-우선 검색(Breadth-First Search) /	477
	미로(Mazes) /	480
	전망(Perspective) /	481
	연습문제 /	483
30 장	연결	485
	연결된-요소들(Connected Components) /	486
	이중연결성(Biconnectivity) /	487
	찾기-합하기 알고리즘들(Union-Find Algorithm) /	490
	연습문제 /	500

31 장	가중치 그래프	501
	최소 스패닝 트리(Minimum Spanning Tree) /	503
	우선순위-우선 검색(Priority-First Search) /	504
	Kruskal 방법(Kruskal's Method) /	509
	최단 경로(Shortest Path) /	512
	기하학적 문제(Geometric Problems) /	519
	연습문제 /	521
32 장	방향성 그래프	523
	깊이-우선 검색(Depth-First Search) /	524
	추이적 폐포(Transitive Closure) /	526
	모든 최단 경로들(All Shortest Paths) /	529
	위상 정렬(Topological sorting) /	532
	강하게 연결된-요소(Strongly Connected Components) /	535
	연습문제 /	538
33 장	네트워크 흐름	539
	네트워크 흐름 문제(The Network Flow Problems) /	540
	Ford-Fulkerson 방법(Ford-Fulkerson Method) /	542
	네트워크 검색(Network Searching) /	544
	연습문제 /	549
34 장	매칭	551
	양분 그래프(Bipartite Graphs) /	553
	안정적 결혼 문제(Stable Marriage Problem) /	556
	고수준 알고리즘(Advanced Algorithms) /	561
	연습문제 /	562
수학적 알고리즘들		563
35 장	난수	565
	응용 분야들(Applications) /	566
	선형 조화 방법(Linear Congruential Method) /	567
	부가적 조화 방법(Additive Congruential Method) /	571
	무작위성 시험(Testing Randomness) /	574

	구현 노트(Implementation Notes) / 576	
	연습문제 / 578	
36 장	산술연산	579
	다항식 연산(Polynomial Arithmetic) / 580	
	다항식 평가법 및 보간법(Polynomial Evaluation and Interpolation) / 583	
	다항식 곱(Polynomial Multiplication) / 587	
	큰 정수 값을 가지는 경우의 산술연산 (Arithmetic Operations with Large Integers) / 590	
	행렬 산술 연산(Matrix Arithmetic) / 592	
	연습문제 / 595	
37 장	Gauss 소거법	597
	단순 예제(A Simple Example) / 597	
	방법 개요(Outline of the Method) / 600	
	변형과 확장(Variations and Extensions) / 604	
	연습문제 / 607	
38 장	곡선 맞춤	609
	다항식 보간(Polynomial Interpolation) / 610	
	스플라인 보간(Spline Interpolation) / 611	
	최소 제곱 방법(Method of Least Squares) / 616	
	연습문제 / 620	
39 장	적분	621
	심볼릭 적분(Symbolic Integration) / 622	
	간단한 구상법(Simple Quadrature Methods) / 623	
	합성법(Compound Methods) / 627	
	적응력있는 구상법(Adaptive Quadrature) / 628	
	연습문제 / 631	
고수준 토픽들	633
40 장	병렬 알고리즘	635
	일반적 방식(General Approaches) / 636	
	완전-셔플(Perfect Shuffles) / 637	

	시스톨릭 배열(Systolic Arrays) / 643	
	전망(Perspective) / 647	
	연습문제 / 649	
41 장	빠른 Fourier 변환	651
	평가, 곱셈, 보간(Evaluate, Multiply, Interpolate) / 651	
	복소수 단위근들(Complex Roots of Unity) / 653	
	단위근들에서의 계산(Evaluation at the Roots of Unity) / 654	
	단위근들에서의 보간(Interpolation at the Roots of Unity) / 657	
	구현(Implementation) / 659	
	연습문제 / 662	
42 장	동적 프로그래밍	663
	배낭 문제(Knapsack Problem) / 664	
	행렬 연속곱셈(Matrix Chain Product) / 667	
	최적 검색 트리들(Optimal Binary Search Tree) / 671	
	시간 및 공간 요구 조건(Time and Space Requirements) / 673	
	연습문제 / 675	
43 장	선형 프로그래밍	677
	선형 프로그래밍(Linear Programming) / 678	
	기하학적 해석(Geometric Interpretation) / 679	
	심플렉스 기법(Simplex Method) / 683	
	구현(Implementation) / 688	
	연습문제 / 692	
44 장	완전 검색	693
	그래프에서의 완전-검색(Exhaustive Search in Graphs) / 694	
	역추적(Backtracking) / 697	
	이탈(Digression): 순열 생성 (Permutation Generation) / 701	
	근사치 알고리즘(Approximation Algorithm) / 703	
	연습문제 / 706	
45 장	NP-완전 문제들	707
	확정적 및 비확정적 다항식-시간 알고리즘 / 708	
	NP-완전성(NP-Completeness) / 710	
	쿠크(Cook)의 정리 / 713	

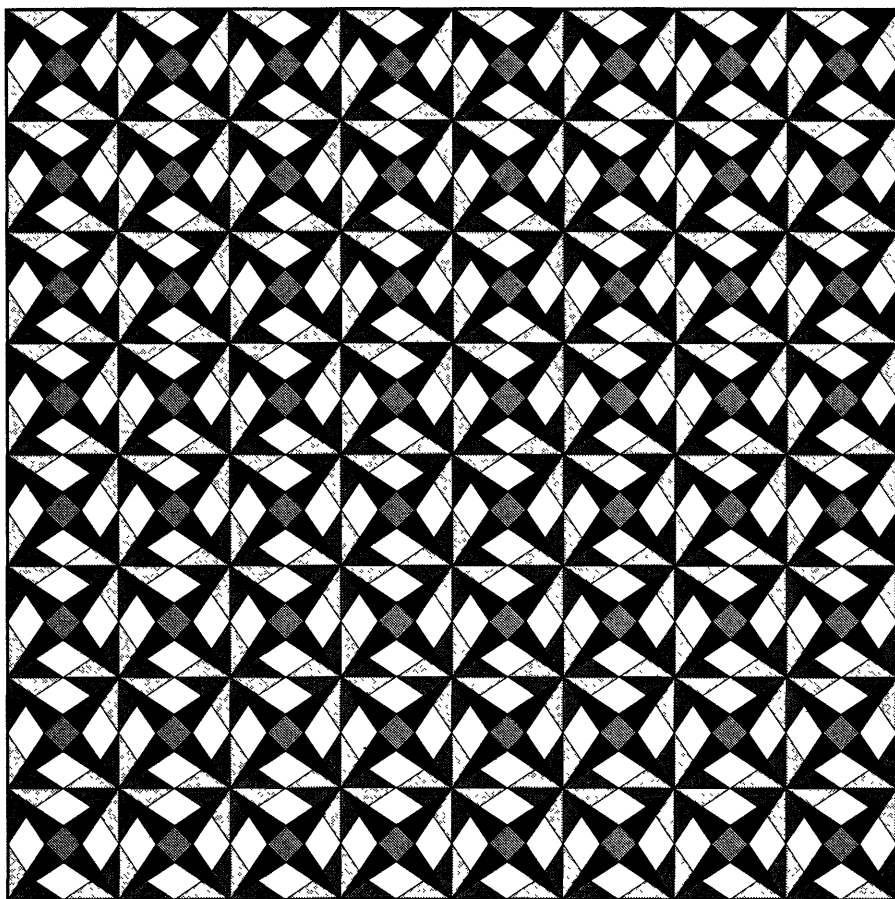
일부 NP-완전 문제들 / 713

연습문제 / 716

용어풀이 717

찾아보기 733

기본사항들



빈 면

1 장

소개

본 교재의 목적은 중요하고 유용한 알고리즘 즉, 컴퓨터 구현에 적합하도록 문제를 해결하는 방법을 광범위하게 공부하는데 그 주된 목적이 있다. 공부하는데 흥미가 있고 반드시 알아야만 하는 기본적인 알고리즘에 항상 중점을 두면서 여러 가지 종류의 응용에 대해서 다루고자 한다. 다루어야 할 알고리즘과 영역이 많으므로해서, 많은 문제들을 깊이 있게 다루지는 못했으나 알고리즘의 특징을 이해하는데 중점을 두려고 노력을 했다. 간단히 말해서, 우리의 목표는 오늘날 컴퓨터에서 이용되는 많은 종류의 중요한 알고리즘을 배워서 그것을 충분히 이용하고 평가하는데 있다.

알고리즘을 잘 이해하기 위해서는, 그것을 컴퓨터에 실행시켜 보아야 할 것이다. 따라서, 본 교재에 제시된 프로그램을 이해함에 있어서 추천된 전략은 그것들을 구현하고 실행시키고, 실제 문제에 적용시키도록 해 보는데 있다. 이같은 알고리즘의 구현을 위해서 C++ 프로그래밍 언어를 이용할 것이다. 그러나 언어는 상대적으로 적은 부분만 이용되기 때문에, 여기에 이용된 프로그램들이 여러 가지 프로그래밍 언어로서 쉽게 변형시켜 이용될 수 있다.

본 교재를 읽는 여러분은 적어도 한 개 이상의 프로그래밍 언어에 대해서 지식이 있어야만 할 것이다. 또한 배열, 스택, 큐, 트리와 같은 기본적인 자료구조 비록 이런 문제들이 3장과 4장에서 어느 정도 복습이 되지만 이것들에 관한 기초적인 지식은 본 교재를 습득하는데 도움이 될 것이다. 기계 구조, 프로그래밍 언어와 다른 기본적인 컴퓨터 개념들의 지식이 또한 요구된다.(이런 문제들이 적절하게 본 교재에서 제시하지만 그러나 특별한 문제를 해결하는 문맥 내에 그런 것들이 존재하게 된다) 우리가 처리하고자 하는 몇 가지 응용들은 기초적인 수학 지식도 요구된다. 선형대수, 기하학과 이산 수학을 포함한 몇 가지 매우 기본적인 문제들이 이용되나, 이같은 지식이 사전에 반드시 필요한 것은 아니다.

알고리즘

컴퓨터 프로그램을 기술함에 있어서, 알고리즘은 이전에 제안된 문제를 해결하는 방법을 일반적으로 구현하는 것이다. 이 방법은 특별한 컴퓨터와는 별개로, 많은 종류의 컴퓨터에서 동등하게 이용된다. 어떤 경우에서든지, 문제를 해결하는 방법을 배우기 위해서는 컴퓨터 프로그램 그 자체가 아니고 방법론인 것이다. 알고리즘(algorithm)이란 용어는 컴퓨터 프로그램으로 구현에 적합한 문제 해결 방법을 기술하기 위해서 전산학에서 이용된다. 알고리즘은 전산학의 기본으로 대부분의 경우는 아니지만 많은 분야에서 연구의 주된 부분이다.

대부분 관심이 되는 알고리즘은 계산에서 수반된 자료를 구성하는 복잡한 방법들이 포함된다. 이같은 목적으로 생성된 목적물을 자료구조(data structures)라 부르고 또한 그것들은 전산학에서 중심적인 요소가 된다. 이와 같이 자료구조와 알고리즘은 서로 협조가 이루어지는 것이다. 본 교재에서는 알고리즘의 부산물이나 마지막 생산물과 알고리즘을 이해하기 위해서 연구해야 할 필요로서 자료구조가 존재하는 견해를 취하고 있다. 간단한 알고리즘은 복잡한 자료구조를 야기시키고 역으로 복잡한 알고리즘은 간단한 자료구조를 이용할 수가 있다. 본 교재에서는 많은 종류의 자료구조에 대한 특징들을 공부하게 될 것이다.

매우 큰 컴퓨터 프로그램이 개발될 때, 해결되어야 할 문제를 이해하고 정의하고 복잡도를 처리하고 쉽게 구현을 시키기 위해서 보다 적은 형태의 부분 타스크로 분해하도록 하는데 많은 노력이 필요케 된다. 분해 후에 대부분의 알고리즘은 구현하기가 단순하게 된다. 그러나 대부분의 경우에서, 몇몇 알고리즘은 선택이 중요한 경우가 있다. 그 이유는 대부분의 시스템 자원들이 이같은 알고리즘을 수행하는데 소비가 되기 때문이다. 본 교재에서는 많은 영역의 큰 프로그램에 기본이 되는 여러 가지 기본적인 알고리즘에 대해서 공부하고자 한다.

컴퓨터 시스템에서 프로그램들의 공유는 상당히 널리 이용되므로 중요한 컴퓨터 이용자에게는 본 교재의 알고리즘에서 많은 부분들이 이용되는 반면에 그것들의 다소 적은 부분만이 구현하는데 필요케 된다. 그러나 기본적인 알고리즘의 단순한 버전으로 구현함은 그것들을 더 잘 이해하는데 도움을 주고, 고차원의 버전에 더 효과적으로 이용할 수 있도록 도움을 준다. 또한, 많은 컴퓨터에서 소프트웨어를 공유함에 대한 메카니즘은 특별한 작업을 효과적으로 수행하는 표준 프로그램을 작성하기가 어렵으므로 기초적인 알고리즘을 다시 구현해야 할 때가 가끔 발생된다.

컴퓨터 프로그램은 가끔 과도할 정도로 최적화 된다. 알고리즘이 매우 큰 작업에 대해 이용되지 않거나 여러 번 이용되지 않으면 구현이 가장 효율적이라는 가능성에는 별로 가치가

없다. 그 이외의 경우, 상대적으로 간단한 구현으로 충족된다. 즉, 그것들이 수행되도록 하는 어떤 자신을 가질 수가 있고 특별한 몇초동안에 수행됨을 의미하는 가장 좋은 가능한 버전보다 아마도 5내지 10배 정도로 느리게 수행될 것이다. 대조적으로, 첫 번째 위치에서 알고리즘의 적절한 선택은 백배나 천배 혹은 그 이상의 요소로서 어려움을 주게 되며, 처리 시간에서 수 분, 수 시간 혹은 그 이상으로 번역된다. 본 교재에서는 가장 좋은 알고리즘의 가장 단순하고 합리적인 구현에 대해 집중 논한다.

가끔 여러 가지 다른 알고리즘(또는 구현)은 같은 문제에 대해 해결하게 된다. 어떤 특정한 작업에 대한 가장 좋은 알고리즘의 선택은 가끔 복잡한 수학적 분석이 수반되면서 매우 복잡한 처리가 존재케 된다. 이러 분야를 연구하는 전산학의 가치가 알고리즘 분석(analysis of algorithms)이다. 공부하고자 할 알고리즘의 대부분은 분석을 통해서 매우 좋은 수행성을 지님을 볼 것이다. 반면에 다른 것들은 경험을 통해서만 잘 처리가 되어짐을 알 수가 있다. 우리는 비교적인 활용도 문제에 대해서 안주하지는 않을 것이다. 우리의 목표는 중요한 일들에 대해서 몇 가지 합당한 알고리즘을 배우는 것이다. 그러나 어떤 자원이 소모되는가 하는 몇 가지 개념을 지니지 않고는 알고리즘을 이용할 수가 없다. 그래서 알고리즘이 어떻게 수행되는 가를 알아야만 할 것이다.

본 교재 주제들의 개요

아래의 문제들에 대한 일반적인 방침을 지시 뿐만 아니라 몇 가지 특별한 주제들의 주된 부분들을 본 교재에서 간단히 기술한다. 주제들의 이같은 범주는 가능한 많은 기본적인 알고리즘을 다루게 될 것이다. 다루어질 분야의 몇몇은 널리 응용되는 기본적인 알고리즘을 깊이 있게 배워야 할 정도로 전산학에서 핵심이 된다. 다른 분야들은 전산학과 관련된 분야들 즉, 수치해석, O.R., 컴파일러 구성론과 알고리즘의 이론분야내에서 연구해야 하는 상위 분야이다.

본 교재 내용에서 기본적인 것들(Fundamentals)은 다음 장에서 이용되는 도구들과 방법들이다. 배열, 연결 리스트, 스택, 큐와 트리들이 포함된 기본적인 자료구조에 대한 소개와 더불어서 C++의 간단한 설명이 포함된다. 재귀(recursion)의 실제적인 이용이 논의되고, 알고리즘을 구현하고 분석하는데 기본적인 접근 방법이 포함된다.

파일들을 순서적으로 재배열하는 정렬(Sorting) 방법은 기본적으로 중요한 것들로서 어느 정도 깊이 있게 다루게 된다. 다양한 방법들이 개발되고, 기술되고 비교가 된다. 여러 가지

관련된 문제들에 대한 알고리즘들은 우선 순위 큐, 선택과 병합등이 수반되어서 취급된다. 이 같은 몇 가지 알고리즘은 본 교재의 뒤에 존재하는 다른 알고리즘의 기본으로 이용된다.

파일들에서 어떤 것들을 찾는 검색(Searching) 방법 역시 기본적으로 중요하다. 여기서는 이진 검색 트리, 균형 트리, 해싱, 디지털 검색 트리와 트라이(tries) 그리고 상당히 큰 파일에 대해 적절한 방법들을 수반한 트리를 이용한 검색과 디지털 키 변형에 대한 기초적이고 상위의 방법들에 대해서 논한다. 이같은 방법들 사이에 관계가 논의되고 정렬방법들과의 유사성이 지적된다.

스트링 처리(String Processing) 알고리즘은 연속된 문자들을 처리하는 방법들의 범주가 포함된다. 스트링 검색은 파싱(parsing)으로된 패턴 일치(pattern matching)로 이끌어진다. 파일 추약 기법과 암호화 또한 고려된다. 다시, 상위 주제들에 대한 소개가 중요하다고 판단된 몇 가지 기초적인 문제들에 대해서 주어진다.

기하학(Geometric) 알고리즘은 점들과 선들(그리고 다른 간단한 기하학적 물체)이 수반된 문제들을 해결하는 방법들의 집합이다. 점들 집합의 볼록 면(convex hull)을 찾고, 기하학적 물체들 사이에 교차점을 찾고, 가장 근접된 점 문제를 해결하고 다차원 검색에 대한 알고리즘들이 고려된다. 이같은 방법들 대부분은 더 기초적인 정렬과 검색 방법들이 요구된다.

그래프(Graph) 알고리즘은 여러 가지 어렵고 중요한 문제들에 대해 유용하다. 그래프에서 검색에 대한 일반적인 방법이 개발되고 최소 경로, 최소 스패닝 트리, 망 흐름(network flow)과 일치가 포함된 기본적인 연결성 문제에 적용된다. 이같은 알고리즘의 통일된 취급은 모두가 같은 프로시저에 기본을 둔 것이고 이같은 프로시저는 앞에서 개발된 기초적인 자료구조에 의존케 된다.

수학적(Mathematical) 알고리즘은 산술과 수치해석에서 기본적인 알고리즘이 포함된다. 많은 내용에서 제거되는 여러 가지 수학적 문제들 즉, 무작위 발생기, 방정식 해, 데이터 적합성과 적분을 해결하기 위한 알고리즘 뿐만 아니라 정수, 다항식과 행렬로된 산술식에 대한 방법을 공부케 된다. 중요성은 수학적 기초에 있는 것이 아니고 방법의 알고리즘적인 면에 있다.

고급(Advanced) 주제는 연구의 여러 가지 다른 분야에 대해 본 교재의 주제와 관련되어서 논의케 된다. 특수 목적의 하드웨어, 동적 프로그래밍, 선형 프로그래밍, 철저한(exhaustive) 검색과 NP-완전성은 본 교재에서 직면한 기초적인 문제들에서 제시된 흥미로운 고급 연구 분야에 대해서 여러분들에게 몇 가지 평가를 제공시키기 위해서 기초적인 견해에서 조사가 된다.

알고리즘의 공부는 그것이 풍부한 전통을 지닌(몇몇 알고리즘은 수천년동안 알려져왔다) 새로운 영역이므로(거의 모든 알고리즘들이 25년 이내의 것들이다) 흥미롭다. 새로운 발견이 계속적으로 이루어지고 있고 어떤 알고리즘은 완전히 이해도 못하고 있다. 본 교재에서는 거대하고 간단하고 쉬운 알고리즘 뿐만 아니라 미묘하고 복잡하고 어려운 알고리즘을 다루게 된다. 우리의 도전은 이전의 것을 이해하고 많은 잠재적인 응용들의 내용에서 나중에 평가를 하게 된다. 그렇게 함으로써, 도래될 계산적 도전에서 잘 서비스되도록 하는 “알고리즘적 사고”의 방법을 개발하고 여러 가지 유용한 도구들을 조사케 된다.

빈 면

2 장

C++와 C

본 교재에서 사용되는 프로그래밍 언어는 C++이다. 모든 언어들은 좋은 점과 나쁜 점을 지니고 있으므로 해서 어느 교재에서든지 특별한 언어의 선택은 장단점을 지닌다. 그러나 많은 현재 프로그래밍 언어는 비슷하고 상대적으로 적은 언어 구성을 이용하고 C++의 특성에 기반을 둔 구현 판단을 피하므로서 우리는 쉽게 다른 언어로 번역할 수 있는 프로그램들을 개발할 수 있다. 우리의 목표는 가능한 간단하고 직접적인 형태인 알고리즘을 제시하고자 하며, C++가 이같은 것을 수행할 수 있도록 한다.

알고리즘은 가끔 상상적인 언어로서 교과서나 연구 보고서에서 기술된다. 즉, 불행히도, 가끔 상세한 내용은 생략되고 유용한 구현과는 거리가 먼 것을 여러분들에게 남겨준다. 본 교재에서는 알고리즘을 이해하고 그것의 유용성을 확실히 하기 위한 가장 좋은 방법으로 실제적인 구현으로된 경험을 통한 견해를 택한다. 현재 언어들은 충분히 표현이 가능하므로 실제적인 구현은 상상한 것과 같이 간결하고 세련되게 될 것이다. 여러분들은 국지적인 C++ 프로그래밍 환경과 관련된 것에 고무될 것이다. 그 이유는 본 교재에서의 구현은 실행되고, 실험되고, 변경되도록 된 프로그램을 처리하는 것이다.

본 교재에서의 C++를 이용하는 장점은 그것이 널리 이용되고 여러 가지 구현에서 필요로 하는 기본적인 특징을 지니고 있는 것이다. 반면 단점은 널리 이용되는 현재 언어로서 유용하지 못한 특징을 지니고 프로그램에서 진정한 언어 의존도를 알고서 취급해야 한다. 몇몇 프로그램들은 상위의 언어 특징을 지니므로 해서 간단화 되어지나 이것은 우리가 생각했던 것보다 사실적이지 못하다. 적절할 때, 그런 프로그램들의 논의에서 적절한 언어 문제를 다루게 될 것이다. 특히, C와 조화되는 C++의 주된 특징중에 하나의 장점을 취하게 된다. 즉, 대

부분 코드는 C로서 쉽게 인식이 되나 중요한 C++ 특징들은 많은 구현에서 중요한 역할을 하게 된다.

C++언어의 간결한 기술은 Stroustrup 책인 The C++ Programming Language(2 판)을 참조하기 바란다. 이장의 목적은 그 책에 있는 정보를 반복하는 것이 아니고 간단한(그러나 고전적인 것) 알고리즘 구현에 이용되는 형태와 언어의 기본적인 특징을 제시하기 위함이다. 입/출력을 제외하고는 이장의 C++코드 역시 C 코드이다. 즉, 다른 C++ 특징들은 다음 몇 장에서 보게 될 더 복잡한 데이터와 프로그램 구조를 고려할 때 제 역할을 하게 된다.

예제 : 유클리드(Euclid) 알고리즘

우선 먼저, 고전적인 문제를 해결하기 위해서 C++ 프로그램을 고려해보자. 즉, “주어진 분수를 가장 낮은 항으로 줄여보자”에서 $4/6$, $200/300$ 또는 $178468/267702$ 가 아니고 $2/3$ 으로 기술하도록 한다. 이같은 문제를 해결하기 위해서는 분모와 분자의 최대 공약수(gcd: greater common divisor)를 찾는 것으로 그것들 둘 다를 나누는 가장 큰 정수이다. 분수 부분은 분모와 분자 둘 다 최대 공약수에 의해서 나누므로써 가장 적은 항으로 줄여지게 된다. 최대 공약수를 찾기 위한 효과적인 방법은 2천년 전에 고대 그리스 사람에 의해서 발견되었다. 이것이 유클리드의 유명한 논문인 Elements에서 제시되었기 때문에 유클리드 알고리즘이라 부른다.

유클리드 방법은 u 가 v 보다 큰 경우, u 와 v 의 최대 공약수는 v 와 $u - v$ 의 최대 공약수와 같다는 사실에 근거로 하고 있다. 이같은 내용을 C++로 구현하면 다음과 같다.

```
#include <iostream.h>
int gcd(int u, int v)
{
    int t;
    while ( u > 0 )
    {
        if ( u < v ) { t = u; u = v; v = t; }
        u = u - v;
    }
    return v;
}
```

```

main()
{
    int x, y;
    while ( cin >> x && cin >> y )
        if ( x > 0 && y > 0 ) count << x << ' ' << y << ' '
                                << gcd(x, y) << '\n';
}

```

먼저, 이 코드에서 금지되는 언어의 속성을 고려하자. C++는 프로그램의 주된 특징들로서 쉬운 인식을 허용하는 엄격한 고급 구문을 지니고 있다. 프로그램은 함수의 리스트로 구성이 되고 그 중 하나는 프로그램 몸체인 main 이름을 지니고 있다. 함수는 return문에서 값을 되돌려 준다. C++는 입출력에 대한 “스트림 라이브러리(stream library)”가 포함된다. include문은 라이브러리에 참조를 할 수 있게 한다. 연산자 <<는 출력 스트림 cout에 놓는 것을 의미하고 똑같이 >>는 입력 스트림 cin에서 가져오는 것을 의미한다. 이같은 연산자들은 스트림들과 같이 인쇄되는 것들의 형들과 일치가 된다. 이 경우 두 개 10진 정수가 읽혀지고, 그것의 최대 공약수에 따라서 인쇄가 된다.(\n인 “개행” 문자에 의해) cin >> x의 값은 더 이상의 입력이 없을 때 0이 된다.

위의 프로그램 몸체는 단순한 것으로 입력에서 수들의 쌍을 읽어서 그것들 둘 다가 양수이면 그것과 최대 공약수를 출력한다.(gcd는 u 나 v 가 음수이거나 0으로 부를 때 무엇이 발생되는가?) gcd함수는 유클리드 함수 그 자체를 구현한 것이다. 즉, 먼저 필요한 경우 그것들을 교환시키므로서 $u \geq v$ 임을 확인하고, u 를 $u - v$ 로 대체케 된다. 변수 u 와 v 의 최대 공약수는 프로시저에 제시된 원래 값들의 최대 공약수와 항상 같다. 결국에는 u 가 0이고 v 는 u 와 v 의 원래(그리고 모든 중간값) 값들의 최대 공약수와 같을 때 처리는 종료된다.

위의 예는 여러분이 몇몇 C++ 프로그래밍 시스템에서 익숙한 완전한 C++ 프로그램으로 기술된 것이다. 관심이 되는 알고리즘은 서브루틴(gcd)으로서 구현되고, 주 프로그램은 서브루틴을 처리하는 “장치”이다. 이같은 구조가 전형적인 것이고 완전한 예제가 본 교재의 알고리즘들이 몇몇 표본 입력 값들에 대해서 구현되고 수행될 때 이해하기 가장 좋은 점이라는 것을 강조하기 위해서 여기에 포함시켰다. 사용 가능한 디버깅 환경의 질에 따라서, 여러분들은 프로그램들을 더 많이 구성시킬 수 있다. 예를 들면, while 반복에서 u 와 v 에 의해서 취해진 중간값은 위의 프로그램에서 관심사항이 된다.

이전 절에서의 주제는 알고리즘이 아니라 언어이지만, 고전적인 유클리드 알고리즘을 정당화해야 한다. 즉, 위의 구현은 한 번 $u > v$ 이면, 수가 v 보다 적을때까지 u 에서부터 v 의 배수를 뺀다는 사실을 알므로써 개선된다. 그러나 이 수는 u 를 v 로 나눈후에 남아있는 나머지와 정확히 같다. 이것은 나머지(modulo) 연산자(%)로 계산이 되어진 것이 무엇인가를 나타낸다. 즉, u 와 v 의 최대 공약수는 v 와 $u \% v$ 의 최대 공약수와 같다. 예를 들면, 461952와 116298의 최대 공약수는 다음 순서에 나타난바와 같이 18이다.

461952, 116298, 113058, 3240, 2898, 342, 162, 18

이같은 순서에서의 각 항목은 이전의 두수를 나눈뒤에 남아있는 나머지이다. 즉, 순서는 18을 162로 나누기 때문에 종료가 되므로 18은 모든 수들의 최대 공약수이다. 여러분은 %연산자를 이용하기 위해서, 매우 큰 수와 매우 작은 수의 최대 공약수를 찾을 때 훨씬 더 효과적인 변형이 어떤 것인지를 나타내기 위해서 위의 구현을 변형시키기를 바랄 것이다. 이 알고리즘은 항상 상대적으로 적은 단계로 이루어진다.

데이터 형(Types of Data)

본 교재에서의 대부분 알고리즘들은 간단한 데이터 형으로 처리가 된다. 즉, 정수 형, 실수 형, 문자 형 또는 문자들의 스트링등이다. C++에서 대부분 중요한 특징중의 하나는 이같은 기초적인 블록을 생성하는 것에서부터 더 복잡한 데이터 형을 생성하는데 대한 준비이다. 이같은 많은 예제들을 본 교재의 나중에 보게 될 것이다. 그러나, 예제들을 간단히 하게 하고 그것들의 데이터 속성이라기 보다는 알고리즘의 동적인면에 초점을 맞추기 위해서 그런 편리성을 과도하게 이용하는 것은 피해야한다. 일반성의 결여없이 이같은 것들을 수행하려고 할 것이다. 정말로, C++와 같은 상위의 능력이 제공되는바로 그 유용성으로 간단한 데이터 형으로 수행하는 “장난감”에서 C++ class에 대한 중요한 연산을 구현하는 일하는 말(workhorse)로 알고리즘을 변형시키기가 쉽다. 기본적인 방법이 사용자 정의 형으로서 잘 기술이 될 때, 그렇게 된다. 예를 들면, 24장에서 28장에있는 기하학적 방법들은 점, 선 다각형 등등에 대한 형에 기본을 둔 것이고 11장의 우선 순위 방법과 14장에서 18장의 검색 방법은 C++ class구조를 이용한 특별한 자료구조와 관련이 된 연산의 집합으로 잘 표현한 것이다. 3장에서 이 문제를 다시보게 되고, 본 교재에서 훨씬 더 많은 예제들을 보게 될 것이다.

가끔 데이터의 적절한 저급 표현이 효용성에 열쇠가 되는 경우가 있다. 개념적으로, 프로그램이 처리되는 방법은 수들이 나타나는 방법이나 문자들이 팩화(두 수를 꼬집어내기 위해서)시키는 방법에 의존되는 것이 아니나 이같은 개념의 추적을 통해서 효용성에서 지불되는 비용은 가끔은 높다. 과거에 프로그래머들은 표현에서 몇 가지 제약이 없는 어셈블리 언어나 기계어로 이동시키는 철저한 단계를 취하므로써 이같은 상황에 대응했다. 다행히도, 현재 고급언어들은 그런 극단적인 것들로 가지 않고 감각적인 표현을 생성하기 위한 메카니즘을 제공해준다. 물론, 그런 메카니즘은 반드시 기계 의존적인 것이므로 그것의 설명이 필요할때를 제외하고는 상세하게 그것들을 고려하지 않을 것이다. 이 문제들은 10장, 17장과 22장에 상세히 기술된다. 여기서 데이터의 이진 표현을 기본으로 한 알고리즘들이 고려가 되었다.

또한 문자 형과 문자 스트링에 대한 연산을 수행하는 알고리즘을 고려할 때 기계 의존적인 표현 문제를 처리하는 것을 피하고자 한다. 가끔은 정수 i 에 의해서 표현된 알파벳의 i 번째 문자를 간단한 코드로 이용해서, 대문자 A에서 Z로서만 처리하는 예제들을 간단히 만든다. 문자형과 문자 스트링의 표현은 그런 데이터를 처리하기 위해서 알고리즘을 구현하기 전에 그것을 완전히 이해해야 하는 프로그래머, 프로그래밍 언어와 기계사이에 인터페이스의 기본적인 부분이다. 단순화된 표현을 기본으로 한 본 교재에서 주어진 방법들은 그때 쉽게 채택된다.

가능하면 정수형을 이용케 된다. 실수형을 처리하는 프로그램들은 수치해석 영역에서 수들이 떨어지도록 만들어야 한다. 전형적으로, 그것들의 활용도는 표현의 수학적 특성과 연결되어 있다. 몇 가지 수학적 알고리즘이 논의될 37장, 38장, 39장, 41장과 43장에서 이 문제를 논하게 된다. 정상적으로는 실수형 표현과 관계되므로해서 비효율성과 부정확성을 피하기 위해 실수형이 더 적절하다고 보일지라도 정수형을 쓰게 될 것이다.

입/출력

의미있는 기계 의존도의 또 다른 영역은 입력과 출력으로 언급되어진 프로그램과 그것의 데이터사이에 상호작용이다. 운영체제에서, 이같은 용어는 컴퓨터와 자기 테이프나 디스크와 같은 물리적 매개체사이에 데이터의 전송을 의미한다. 이 문제에 대해서는 13장과 18장에서 언급이 되어진다. 가끔 위의 gcd와 같은 알고리즘의 구현에서부터 결과를 유추하고 데이터를 얻으려고 하는 체계적인 방법을 단순히 모색케 된다.

“입력(reading)”과 “출력(writing)”이 불려질 때, 표준적인 C++ 특징을 이용하나 가능한 유용한 특별한 포매팅 기법의 몇 가지로서 야기시킨다. 다시말해, 우리의 목표는 프로그램을 간결하고 쉽게 변형이 가능하도록 하는 것이다. 즉, 여러분이 프로그램을 변경시키려고 하는 방법은 프로그래머와 같이 그것의 인터페이스로 장식을 하는 것이다. 만약 존재하는 경우, 몇 가지 안되는 현재의 C++나 다른 프로그래밍 환경은 외부 장치를 언급하기 위해서 실제로 `cin`이나 `cout`을 취한다. 대신, 그것들은 정상적으로 “논리 장치”나 “스트림”으로 언급된다. 이와 같이 한 프로그램의 출력은 실제로 읽거나 쓰기 없이도 다른 것에 입력으로 이용된다. 구현에서 입출력을 합리화시키는 경향은 그것들을 그런 환경에서 훨씬 더 유용하게 만든다.

실제로, 대부분 현재 프로그래밍 환경에서 그것은 본 교재를 통한 그림들에서 이용된 것들과 같이 그림으로서의 표현을 이용하는 것이 오히려 쉽고 적절하다. 끝맺는 말에 기술된 대로, 이같은 그림들은 의미있게 장식된 인터페이스로서 프로그램 그 자체에 의해서 실제로 생성된다.

논의된 대부분의 방법은 보다 더 큰 응용 시스템내에서 이용하고자 하는 의미가 있으므로 해서, 그것들이 데이터를 얻고자 하는 더 적절한 방법은 파라미터를 통한 것이다. 이것은 위의 gcd 프로시저에 대해 이용된 방법이다. 또한, 본 교재의 이후 장들에서 여러 가지 구현들은 이전 장에서의 프로그램을 이용하는 것이다. 다시, 알고리즘 그 자체로 관심을 돌리는 것을 피하기 위해서, 일반적인 유틸리티 프로그램들로서 이용한 구현을 “패케지(package)”화하도록 한다. 확실히, 우리가 공부한 대부분의 구현들은 그런 유틸리티에 대한 시작점으로서 아주 적절하나 여기서 우리가 무시하는 수 많은 시스템과 응용에 의존되는 가의 의문은 그런 패케지를 만드는데 만족할 만큼 나타나 있다.

가끔, 과도한 파라미터 전달을 피하기 위해서 “광의(global)” 데이터로서 수행되는 프로그램을 기술케 된다. 예를 들면, gcd 함수는 파라미터 `u` 와 `v` 로서 성가스럽게 하기 보다는 오히려 `x` 와 `y`에 대해 직접적으로 수행을 한다. 이것은 gcd 가 그것의 두 입력들에 의해서 잘 정의가 되었기 때문에 이 경우에는 공정하지 않으나, 여러 알고리즘이같은 데이터에 대해 수행을 할 때나 많은 양의 데이터가 전달될 때, 불필요한 데이터의 이동을 피하기 위해서 그리고 알고리즘을 표현하는데 경제적으로 하기 위해서 광의 변수를 이용케 된다. 다른 한편, C++는 인터페이스를 명확하게 하기 위한 방법으로서 그것에 관련된 자료 구조와 알고리즘에 대해 이상적인 언어이고 대응되는 C나 파스칼(Pascal) 프로그램들에서 보다 C++ 구현에서 광의 변수를 훨씬 더 적게 이용하는 경향이 있다.

결론

위의 프로그램과 유사한 많은 예제들은 *The C++ Programming Language* 교재를 참고하기 바란다. 여러분들은 매뉴얼을 보고 몇 가지 간단한 프로그램을 구현하고 테스트해 본 후에 C++의 기본적인 특징들에 대해서 마음 편안히 이해할 수 있도록 매뉴얼을 읽는 것이 좋다.

본 교재에서 주어진 C++ 프로그램들은 완전한 구현의 예제로서 그리고 실제 프로그램들에 대한 시작점으로서 알고리즘의 간결한 기술을 나타내고자 한다. 위에 언급된 것처럼, 여러분들은 C++로 제시된 알고리즘을 읽는데 별로 어려움이 없어야 하고 다른 언어에도 지식을 가지고 있어야 알고리즘을 다른 언어로 구현을 할 수가 있게 된다. 예를 들면, 다음의 프로그램은 유클리드 알고리즘을 파스칼로 표현한 것이다.

```
program euclid (input, output);
var x, y: integer;
function gcd (u, v: integer): integer;
    var t: integer;
    begin
    repeat
        if u < v then
            begin t := u; u = v; v := t end;
        u := u - v
    until u = 0;
    gcd := v
    end;
begin
while not eof do
    begin
    readln (x, y);
    if (x > 0) and (y > 0) then writeln(x, y, gcd(x, y))
    end;
end.
```

이같은 알고리즘에 대해서, 비록 C++와 파스칼 두 언어의 구현에서 훨씬 더 간결한 구현을 만들기가 어렵지 않지만, 의도한대로 C++와 파스칼 문장들 사이에 거의 1 대 1 대응이 된다. C++구현은 이 경우에서 단지 입출력 경우에만 C와 다르다. 즉, 물론 대부분의 프로그램들이 C에서 유용하지 않은 C++ 구조를 이용하지만 그렇게 할 수 있도록 하는 것이 당연할 때 양립성이 유지케 된다.

연습 문제

1. 교재에 기술된 것과 같이 유클리드 알고리즘의 고전적인 버전을 구현해 보라.
2. u 와 v 가 반드시 양수가 아닐 때 $u \% v$ 에 대한 것을 C++시스템이 계산할 때 어떤 값을 가지는지 체크해 보아라.
3. `struct fraction {int numerator; int denominator;}`를 이용해서 주어진 분수를 가장 하위 항으로 줄이는 프로시저를 구현하라.
4. 빈칸으로 종료되는 한 번에 한 문자(디지트)씩 10진수를 읽고 그 수의 값을 되돌려주는 함수 `int convert()`를 기술하라.
5. 한 수의 이진수 값을 인쇄하는 함수 `binary(int x)`를 기술하라.
6. gcd가 처음 부름 `gcd(12345, 56789)`로 야기될 때 u 와 v 가 얻어진 모든 값들은 무엇인가?
7. 이전 연습문제의 부름에 대해서 얼마나 많은 C++문장이 실행이 되는지 정확하게 기술하라.
8. 세 가지 정수 u , v 와 w 의 최대 공약수를 계산하는 프로그램을 기술하라.
9. 최대 공약수가 1인 C++ 시스템의 정수형들로서 표현되는 수들중 제일 큰 쌍을 구하라.
10. 유클리드 알고리즘을 FORTRAN이나 BASIC으로 구현하라.

빅면

3 장

기초적인 자료구조

본 장에서는 컴퓨터 프로그램에 의해서 처리되는 데이터 구성의 기본적인 방법에 대해서 논하고자 한다. 많은 응용에 있어서, 적절한 자료 구조의 선택은 구현에서 수반되는 유일한 판단이다. 즉, 선택이 한 번 이루어지면, 매우 간단한 알고리즘이 필요케 된다. 같은 데이터에 대해, 몇 가지 자료구조는 다른 것들에 비해 다소 적은 공간을 요구하고, 데이터에 대한 같은 연산에 대해, 몇몇 자료구조는 다른 것에 비해 다소 적은 효율을 지닌 알고리즘으로 이루어진다. 이같은 방법은 알고리즘과 자료구조의 선택이 상당히 뒤엎히고 적절한 선택으로 시간과 공간을 절약시키는 방법을 계속적으로 모색함으로서 본 교재를 통해서 가끔 재발생된다.

자료구조는 수동적인 내용이 아니고 또한 그곳에 수행이 되어지는 연산을 고려해야 한다. (그리고 이같은 연산에 대해 이용된 알고리즘) 이같은 개념은 이장의 끝에 설명될 추상적 데이터 형(`abstract data type`)의 개념을 공식화한다. 그러나 주된 관심은 구체적인 구현에 있고 특별한 표현과 처리에 초점을 맞추게 된다.

본 장에서 배열, 연결 리스트, 스택, 큐와 다른 간단한 변형들을 처리케 된다. 이같은 것은 널리 사용되는 고전적인 자료구조이다. 트리(4장 참조)를 따라서, 본 교재에서 고려된 모든 알고리즘에 대해 기본을 형성케 된다. 본 장에서는 이같은 구조들을 처리하는 방법과 기본적인 표현을 고려하고, 그것들의 이용에서 어떤 특별한 예제들을 통해 처리하고 기억장치 관리와 같은 관련된 문제에 대해 논한다.

배열(Arrays)

아마도 가장 기본적인 자료구조는 배열로서 C++와 대부분 다른 프로그래밍 언어들에서 제일 처음에 정의된다. 배열은 연속적으로 저장이 되고 인덱스에 의해서 접근이 가능한 데이터

항목들의 고정된 수이다. 배열 a 의 i 번째 요소를 $a[i]$ 로서 언급케 된다. 그것을 언급하기 전에 배열 위치 $a[i]$ 에서 어떤 의미있는 것을 저장시키는 것은 바로 프로그래머의 책임이다. 즉, 배열을 무시하는 것은 가장 범하기 쉬운 프로그래밍 실수 중의 하나이다.

배열을 이용한 간단한 예제 즉, 1000 이하의 모든 소수를 인쇄하는 프로그램이 다음과 같이 주어져 있다. 기원전 3세기경에 이용된 이 방법을 “Eratosthenes의 추출(sieve)”이라 부른다.

```
const int N=1000;
main()
{
    int i, j, a[N+1];
    for (a[1] = 0, i = 2; i <= N; i++) a[i] = 1;
    for ( i = 2; i <= N/2; i++)
        for ( j = 2; j <= N/i; j++)
            a[i*j] = 0;
    for ( i = 1; i <= N; i++)
        if ( a[i] ) cout << i << ' ';
    cout << '\n';
}
```

이 프로그램은 가장 간단한 구성 요소들의 형인 부울리언(0-1) 값들로 구성된다. 프로그램의 목적은 i 가 소수일 경우 $a[i]$ 에 1을 세트시키고, i 가 소수가 아닌 경우에는 $a[i]$ 에 0을 세트시킨다. 다른 수의 배수인 어떤 수가 소수가 아닌 경우, 각 i 에 대해서 i 의 각 배수에 대응되는 배열 요소들에 0으로 세트시키므로 수행을 할 수가 있다. 그때 소수들을 인쇄하면서 배열을 통해서 단지 한 번 수행된다. 먼저 배열은 어떤 수들도 소수가 아닌 것으로 표현하기 위해서 초기화시킨다. 즉, 알고리즘은 소수가 아닌 것으로 알려진 인덱스들에 대해 대응되는 배열 요소들에 0으로 세트시킨다. 프로그램은 j 를 수반한 for 루프 이전에 $a[i]$ 를 테스트함으로써 더 효율적으로 만들게 된다. 그 이유는 i 가 소수가 아닌 경우, 그것의 배수에 대응되는 모든 배열의 요소들은 이미 표시가 되어졌기 때문이다. 정수보다는 오히려 비트들의 배열을 명확하게 이용함으로써 훨씬 더 공간을 효과적인 것으로 만든다.

Eratosthenes의 추출은 배열의 어떤 항목이 효과적으로 접근되는 사실을 추출하는 알고리즘의 전형적인 것이다. 또한 알고리즘은 배열의 항목을 순차적으로 하나씩 접근이 가능하다. 많은 응용에서 순차적인 순서가 매우 중요하다. 즉, 다른 응용에서 순차적인 순서는 다른 것

들만큼 좋기 때문에 이용된다. 그러나 배열들의 주된 특징은 만약 인덱스를 알고 있을 경우, 어떤 항목도 상수시간내에 접근이 가능케 된다.

배열의 크기는 이전에 알고 있다. 즉, N 의 다른 값에 대해서 위의 프로그램을 수행하기 위해서, 상수 N 을 변경시키는 것이 필요하고 그때 컴파일 되고 실행된다. 몇몇 프로그래밍 환경에서, 실행 시간에 배열의 크기를 선언하는 것이 가능하다.(예를 들면, N 의 값에 있어서 사용자 형을 지니고 그때 사용자가 형화하는 어떤 값들 만큼 큰 배열을 선언하므로써 기억 장소를 낭비시킴이 없이 N 보다 적은 소수들로서 응답된다) C++에서, 기억 장소 할당 메커니즘의 적절한 이용을 통해서 이같은 효과를 성취시키는 것이 가능하나 그것의 크기가 고정되고 그것들이 이용되기 전에 알고 있어야만 하는 배열의 속성이 여전히 있다.

배열들은 모든 컴퓨터들에서의 기억장치와 직접적인 대응이 된다는 점에서 기본적인 자료 구조이다. 기계어에서 기억 장소의 단어 내용을 검색하기 위해서 번지를 제공한다. 이와 같이, 배열 인덱스에 대응되는 기억 장소 번지와 함께 배열로서 전체 컴퓨터 기억 장소를 생각케 된다. 대부분 컴퓨터 언어 프로세서는 직접적으로 기억 장소를 접근하는 오히려 효율적인 기계어 프로그램으로 배열을 수반하는 프로그램으로 번역케 된다.

정보를 구조화시키는 또 다른 친숙한 방법은 열과 행들로 구성된 수들의 표를 이용케 된다. 과목에서 학생들의 성적에 관한 표는 각 학생들에 대해 한 행을, 그리고 각 과제에 대해 각 열을 이용케 된다. 컴퓨터상에 그런 표는 두 개 인덱스 즉, 행과 한 개 열에 한 개씩으로 된 2차원 배열로서 표현된다. 그런 구조에 대한 여러 가지 알고리즘은 간단하다. 예를 들면, 한 과제에 대한 평균 성적을 계산하기 위해서 열에 있는 모든 요소들을 합하고 그리고 행들의 수로 나누면 된다. 과목에 대한 특정한 학생의 평균 성적을 계산하기 위해서는 행에 있는 모든 요소들의 값들을 더해서 열의 수로 나누면 된다. 2차원 배열은 이같은 형태의 응용에서 널리 이용된다. 컴퓨터상에서 가끔은 편리하고 2차원 이상에 이용하는 것이 오히려 간단하다. 즉, 선생님들은 수년 동안의 학생 성적표를 유지하기 위해서 제 3의 인덱스를 이용케 된다.

또한 배열은 항목들의 인덱스화된 리스트에 대한 수학적 용어로서 벡터에 직접적으로 대응된다. 비슷하게, 2차원 배열은 행렬에 대응된다. 이같은 수학적 내용을 처리하는 알고리즘을 36장과 37장에서 공부하도록 하자.

연결 리스트(linked list)

고려될 두 번째 기초적인 자료구조는 C++에서는 아니지만 몇몇 프로그래밍 언어(특히 Lisp에서)에서 기본적인 것으로 정의가 되는 것이 연결 리스트이다. 그러나 C++는 연결 리스트를 이용하기 쉽게하도록 하는 기본적인 연산을 제공한다.

배열에 대한 연결 리스트의 기본적인 장점은 연결 리스트가 처리되는 동안에 크기가 늘어나 줄게 된다. 특히, 최대 크기는 미리 알 필요가 없게 된다. 실제적인 응용에서, 어떤 순간에서 그것의 상대적인 크기에 대해 특별한 주의 없이도 여러 가지 자료구조가 같은 공간을 공유하도록 하게 된다.

연결 리스트의 두 번째 장점은 항목들을 효율적으로 재배열하도록 하는데 있어서 유연성을 제공하는 것이다. 이같은 유연성은 리스트에서 어떤 임의의 항목에 대한 직접적인 접근의 비용으로서 얻어진다. 이것은 그곳에 수행되는 기본적인 연산들의 몇몇과 연결 리스트의 기본적인 속성 몇 가지를 조사한 후에 아래에서 더 명백하게 될 것이다.

연결 리스트는 배열에서와 같이 순차적으로 구성된 항목들의 집합이다. 배열에서는 순차적인 구조가 은연중으로 제공되는 반면(배열에서 위치에 의해서) 연결 리스트에서는 각 항목이 다음 노드에 “연결”을 포함한 “노드”의 일부분인 명백한 배열을 이용케 된다. 그림 3.1은 문자들에 의해서 표현된 항목들과, 원으로서 노드들과 노드를 연결하는 선들로 연결들로서된 연결 리스트를 제시하고 있다. 지금은 노드와 연결의 의미를 단순히 말하기 때문에 리스트가 컴퓨터 내에서 어떻게 표현이 되는가는 아래에 자세하게 기술된다.

그림 3.1의 간단한 표현은 고려해야 할 두 가지 상세한 내용을 나타낸 것이다. 첫째, 모든 노드는 연결을 지니므로해서 리스트의 마지막 노드에서 연결은 어떤 특별한 “다음” 노드를 기술해야만 한다. 이같은 목적으로 *z*라 부르는 “가상(dummy)” 노드를 설정케 된다. 리스트의 마지막 노드는 *z*를 가리키고 *z*는 자신을 가리킨다. 부가해서, 리스트의 다른 쪽 끝에 정상적으로 가상 노드를 가진다. 가상 노드의 주된 목적은 특히 리스트의 첫 번째와 마지막 노드를 수반으로 하는것들을 연결하는 것으로서 어떤 처리를 더 편리하게 한다. 다른 규약들은 아래에서 기술한다. 그림 3.2는 수반된 이같은 가상 노드들로된 리스트 구조를 나타낸 것이다.

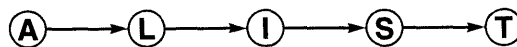


그림 3.1 연결 리스트

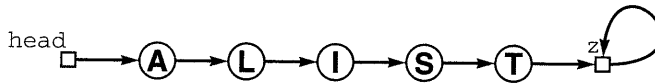


그림 3.2 가상 노드로된 연결 리스트

지금 이같은 명백한 순서들의 표현은 어떤 연산들을 배열에 대해 가능하도록한 것보다 훨씬 더 효과적으로 수행하도록 허용케 된다. 예를 들면, 리스트의 끝에서 제일 처음으로 T를 이동시킨다고 가정해 보자. 배열에서 처음에 새로운 항목에 대한 여지를 주기 위해서는 모든 항목들을 이동시켜야만 한다. 연결 리스트에서는 그림 3.3에 제시된 것과 같이 세 가지 연결만을 변경시키면 된다. 그림 3.3에 제시된 두 개 변형은 비록 그것들의 그림이 다르게 그려졌더라도 의미는 똑같다. 즉, T는 A를 가리키고 있는 노드이고, S는 z를 head는 T를 가리키는 노드들이다. 비록 리스트가 매우 길다고 할 지라도, 단지 세 가지 연결들만을 변경시킴으로써 이같은 구조를 변경시킬 수가 있다.

더 중요한 것은 항목을 연결 리스트에 “삽입”시키는 경우(길이가 한 개 증가되는 것이다) 이같은 연산은 배열에서 매우 불편한 것이다. 그림 3.4는 S를 가리키는 노드에 X를 놓고 그때 I가 새로운 노드를 가리키는 것으로 하는 삽입 예제가 어떻게 수행이 되는가를 보여준 것이다. 리스트가 얼마나 길다고 할지라도, 단지 두 개 연결만이 이같은 연산에 대한 변경에 필요케 된다.

비슷하게 연결 리스트에서 한 항목을 “삭제” 시키는 경우에 대해 보자.(길이는 한 개 줄어든다) 예를 들면, 그림 3.4의 세 번째 리스트는 X를 넘어서 S를 가리키도록 노드 I를 만들므로서 단순히 두 번째 리스트에서 X를 삭제하는 방법을 나타낸 것이다. 여기서, X가 포함된 노드는 여전히 존재하고,(사실 X 노드는 여전히 S를 가리킨다) 어떤 방법으로 처리가 되어

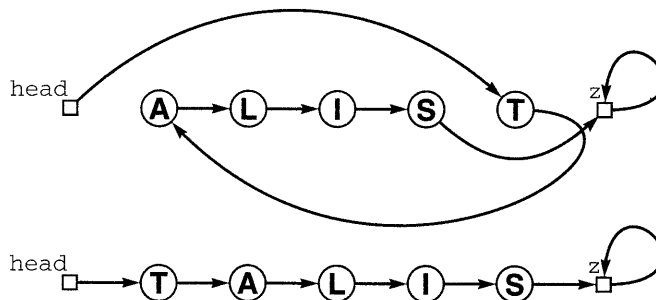


그림 3.3 연결 리스트를 재배열

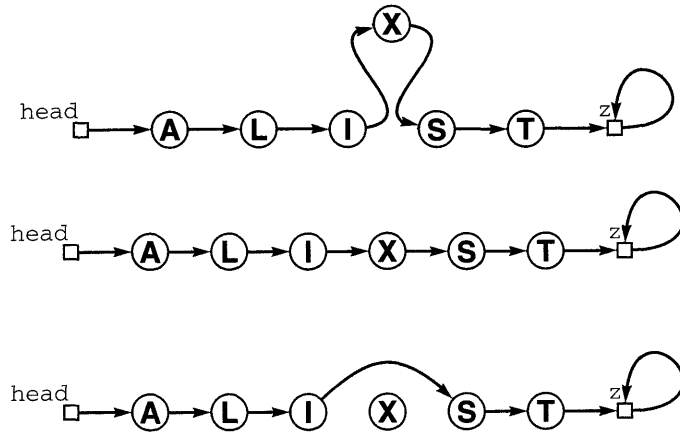


그림 3.4 연결 리스트에서 삭제와 삽입

진다. X가 더 이상 이같은 리스트의 부분이 아니라는 점 그리고 head에서부터 다음의 연결들에 의해 접근이 되어질수가 없다. 이같은 문제는 다음에서 논하기로 하자.

다른 한편, 연결 리스트가 적절하지 못하는 것에 대한 다른 연산이 있다. 이같은 것중 가장 명백한 것은 “ k 번째 항목”을 찾는 것이다.(인덱스가 주어진 항목을 찾는 것이다) 즉, 배열에서 이같은 것은 $a[k]$ 를 단순히 접근함으로써 수행이 되나 리스트에서는 k 개의 연결을 통해서 수행된다. 연결 리스트에서 불편한 또 다른 연산은 “주어진 항목 이전의 항목을 찾는 것”이다. 만약 우리가 가지고 있는 모든 것이 예제 리스트에서 T에 연결이 되면, 그때 S에 연결을 찾는 유일한 방법은 head에서 시작하고 t를 가리키는 노드를 찾기 위해서 리스트를 찾게 된다. 사실, 이같은 연산은 연결 리스트에서 주어진 노드를 삭제하는 경우에 필요케 된다. 즉, 어느 연결이 변경되는 노드인가를 어떻게 찾을 수 있는가? 많은 응용에서, “다음 노드를 삭제”하기 위해서 삭제 연산을 다시 설계하므로써 이같은 문제를 피하게 된다. 비슷한 문제가 리스트에서 삽입 연산으로 “주어진 노드뒤에 주어진 항목을 삽입” 시키므로써 삽입을 피할 수가 있게 된다.

기본적인 연결 리스트를 어떻게 C++에서 구현이 되는가를 제시하기 위해서, 리스트 노드들의 양식을 상세히 기술하고 다음과 같이 빈 리스트를 생성함으로써 시작된다.

```
struct node
{ int key; struct node *next; };
struct node *head, *z;
head = new node; z = new node;
head->next = z; z->next = z;
```

struct 선언은 리스트들이 nodes들로 구성되고, 각 노드는 리스트에서 next 노드의 포인터와 정수를 포함하는 것으로 설명된다. key는 여기서 간단히 정수라고만 하고, 어떤 형이 될 수도 있고 next 포인터는 리스트의 키가 된다. *는 변수들 head와 z가 노드들의 포인터로 선언되는 것이다. 노드들은 내장 함수(built-in function) new가 불러질때만 실제로 생성된다. 이것은 프로그래머로 하여금 리스트가 증가되는 것과 같이 노드들에 대한 기억 장소를 “할당”시키는 일을 감소시키는 의도가 있는 복잡한 메카니즘이 숨어 있다. 이같은 메카니즘을 아래에서 좀더 상세하게 논하고자 한다. “화살표” 기호는 구조를 통해서 포인터를 따라가기 위해서 C++에서 이용된다. 그 연결에 의해서 지시된 노드에 참조를 지시하기 위해서 이같은 기호에 의해서 다음에 나오는 연결에 참조를 기술케 된다. 이와 같이, 위의 코드는 head와 z에 의해서 언급된 두 개 새로운 노드들을 생성하고 z를 가리키기 위해서 그것들을 다들 세트시킨다.

키 값 v로서 새로운 노드를 주어진 노드 t 다음에 있는 점에 연결 리스트로 삽입시키기 위해서는 노드를 생성하고($x = \text{new node}$) 키값을 넣고($x \rightarrow \text{key} = v$) 그리고 t의 연결을 그곳에 복사케($x \rightarrow \text{next} = t \rightarrow \text{next}$)하고 t의 연결을 새로운 노드로 가리키도록 한다. ($t \rightarrow \text{next} = x$)

연결 리스트에서 주어진 노드 t 다음에 있는 노드를 추출하기 위해서는 그 노드에 포인터를 얻고($x = t \rightarrow \text{next}$), 리스트에서 그것을 끄집어내기 위해서 그 포인터를 t에 복사하고($t \rightarrow \text{next} = x \rightarrow \text{next}$), 그리고 리스트가 비어 있지 않는 한($\text{if } (x \neq z) \text{ delete } x$) 내장 함수 프로시저 delete를 이용해서 기억 장소 할당 시스템에 되돌려지게 된다.

여러분에게 그림 3.4에 대한 이같은 C++ 구현을 체크하도록 한다. head노드는 리스트 제일 앞에서 삽입에 대한 특별한 테스트를 지니는 것을 피하므로 가능케 되고, z 노드는 비어 있는 리스트에 대한 시험을 위해서 편리한 방법을 제공케 된다. 14장에서 z에 대한 또 다른 이용을 보게 될 것이다.

다음 장에서 연결 리스트에 관한 이것들과 다른 기본적인 연산들의 응용에 대한 많은 예를 보게 될 것이다. 연산은 단지 몇 가지 문장들만 포함이 되므로서, 데이터 형을 통해서라기 보다는 오히려 리스트에서 직접적으로 처리케 된다. 예제로서, Eratosthenes의 추출 정신으로서 소위 “요세퍼스(Josephus) 문제”를 해결하는 프로그램을 다음에 고려케 된다. N 사람들은 원에서 그 자신들을 재 배열하고 원 주위의 M 번째 사람을 죽이고 각 사람들이 원 밖으로 떨어지도록하는 정렬을 하므로서 대량학살을 자행한다고 상상해 보자. 문제는 어느 사람이

마지막에 죽게 되는지를 찾는 것이다.(아마도 끝에서 심장의 변화를 가진 사람일 것이다) 또는 더 일반적으로 사람이 실행해야 할 순서를 찾기 위한 것이 문제이다. 예를 들면, $N = 9$ 이고 $M = 5$ 인 경우, 사람들은 5 1 7 4 3 6 9 2 8의 순서로 죽게 된다. 다음 프로그램은 N 과 M 을 읽어서 이같은 순서를 인쇄케 하는 것이다.

```
struct node
{ int key; struct node *next; };
main( )
{
    int i, N, M;
    struct node *t, *x;
    cin >> N >> M;
    t = new node; t->key = 1; x = t;
    for ( i = 2; i <= N; i++)
    {
        t->next = new node;
        t = t->next; t->key = i;
    }
    t->next = x;
    while ( t != t->next )
    {
        for ( i = 1; i < M; i++) t = t->next;
        cout << t->next->key << ' ';
        x = t->next; t->next = x->next;
        delete x;
    }
    cout << t->key << '\n';
}
```

프로그램은 실행 순서를 직접적으로 모의 실험하기 위해서 “환상형(circular)” 연결 리스트를 이용한다. 첫째, 리스트는 1에서 N 에 이르는 키로서 생성된다. 즉, 변수 x 는 그것이 생성된 대로 리스트의 처음에 유지가 되고, 리스트에서 마지막에 있는 포인터는 x 로 세트된다. 그때, 프로그램은 $M - 1$ 개 항목을 통해서 계수가 되고, 단지 한 개가 남을 때까지(그후 자신을 가리킨다) 다음을 삭제시키면서 리스트를 통해서 처리된다. 실행에 대응되는 삭제에 대한 delete의 호출을 주시하라. 이것은 위에 언급된 new의 반대이다.

가끔 환상형 리스트는 리스트의 시작(과 끝)을 표시하기 위해서 그리고 비어있는 리스트의 경우를 처리하는데 도움을 주기 위해서 한 개의 가상 노드로서 가상 노드 head나 z를 지니도록하는 대안으로서 이용된다.

이전의 항목 한 개, 이후에 있는 항목 한 개등 각 노드에 대해 두 개의 연결을 유지하도록 하는 이중 연결 리스트(doubly linked list)를 이용하므로써 “주어진 항목 이전에 항목을 찾는” 연산을 수행하도록한다. 이같은 특별한 능력을 제공하는 비용은 기본 연산당 연결 처리의 수가 배가되므로써 특별한 호출없이 정상적으로 이용이 되지 않는다. 그러나 위에 언급된 대로, 노드가 삭제되고 노드에 연결만이 유용한 경우에(아마도 몇몇 다른 자료구조의 일부분 일 것이다) 이중 연결이 불리워지게 된다.

기억장치 할당(Storage Allocation)

C++의 포인터는 위에 제시된것과 같이 리스트를 구현하는 편리한 방법이나 다른 방법들이 있다. 본 절에서는 연결 리스트를 구현하기 위해서 배열을 어떻게 이용하고 C++프로그램에서 이것이 리스트들의 실제적인 표현에 어떻게 관련이 되는지를 논하고자 한다. 위에 언급한 대로, 배열은 컴퓨터 기억장소의 오히려 직접적인 표현이므로써 자료구조가 배열로서 어떻게 구현이 되는가의 분석은 컴퓨터의 낮은 레벨에서 어떻게 표현되는가의 통찰이 주어진다. 특히, 여러 가지 리스트들이 동시에 어떻게 표현이 되는가를 보는것에 관심이 집중된다.

연결 리스트의 직접 배열 표현에서, 연결 대신에 인덱스를 이용케 된다. 처리하는 한 방법은 위에 있는 것들과 같으나 next 필드에 대한 포인터라기 보다는 오히려 정수형(배열 인덱스에 대한것)을 이용하여서 레코드의 배열을 정의하는 것이다. 더 편리한 방법은 “병렬 배열(parallel array)”를 이용하는 것이다. 즉, 배열 key에서의 항목들과 배열 next에서 연결들을 유지케 된다. 이와 같이, `key[next[head]]`는 리스트상에 첫 번째 항목과 관련된 정보를 언급하고, 두 번째는 `key[next[next[head]]]`를 언급케 하는등등이다. 병렬 배열을 이용한 장점은 구조가 데이터의 “제일 위에” 설정이 되는 것이다. 즉, 배열 key는 데이터를 포함하고 모든 구조에서 단지 데이터는 병렬 배열 next내에 있다. 예를 들면, 다른 리스트는 같은 데이터 배열을 이용해서 생성되고 다른 병렬 “연결” 배열이나 더 많은 데이터는 더 많은 병렬 배열들로서 부가된다.

코드의 다음 라인은 병렬 key와 next배열로서 표현된 연결 리스트에서 “삽입 후” 연산을 구현시킬수가 있다.

```
key[x] = v; next[x] = next[t]; next[t] = x++;
```

“포인터” x는 배열에서 다음에 사용되지 않은 위치를 파악하고 기억장소 할당 함수 new상에 부름이 필요하지 않게 된다. 코드를 추출하기 위해서 $next[t] = next[next[t]]$ 로 기술을 하나 배열 위치는 $next[t]$ 에 의해서 “지시된대로” 배열 위치는 잃어버리게 된다. 아래에서 그런 잃어버린 공간이 어떻게 다시 요구되는지를 고려해 보자.

그림 3.5는 예제 리스트가 병렬 배열에서 어떻게 표현이 되는지와 이같은 표현을 이용하고 있는 그림적인 표현에 어떻게 관련이 되는지를 나타내고 있다. key와 next배열은 S L A I T가 초기에 비어있는 리스트에 삽입되는 경우, head 뒤에 S, L과 A가 삽입되고 L뒤에 I, S뒤에 T와 같이 되도록 왼쪽에 제시된다. 위치 0은 head이고 위치 1은 z가 된다.(이것들은 리스트가 초기화 될 때 세트된다) $next[0]$ 이 4이므로, 리스트에서 첫 번째 항목은 $key[4]$ (A)이다. 그리고 $next[4]$ 가 3이므로 리스트에서 두 번째 항목은 $key[3]$ (L) 등이다. 왼쪽에서 두 번째 그림은 next배열에 대한 인덱스들이 선으로 대체된다. 즉, $next[0]$ 에 “4”를 놓으므로써, 노드 0에서 노드 4 아래로 진행하는 선을 그릴수가 있다. 세 번째 그림에서, 하나씩 리스트 요소들을 배열하기 위해서 연결들을 해결하게 된다. 그때 보통 그림으로 표현한 것이 오른쪽에 존재케 된다.

문제의 급소는 내장 프로시저 new와 delete가 구현되는 방법을 고려한 것이다. 노드들과 연결들에 대한 공간만이 우리가 이용하는 배열들인 것으로 가정하자. 이같은 가정은 시스템이 고정된 자료구조로서 자료 구조를 증가시키고 줄이고(자료구조 그 자체)하는 능력을 제공하는 상황에 있게 된다. 예를 들면, A를 포함한 노드는 그림 3.5의 예제에서 삭제가 되고 그때 처리가 된다. 이것은 연결들을 재 배열하는 것으로 노드는 리스트상에서 더 이상 끌어들이지 않으나 그 노드에서 차지하고 있는 공간으로 무엇을 처리할 수 있는가? 그리고 new가 불려지고 더 많은 공간이 필요할 때 한 노드에 대한 공간을 어떻게 찾을 수 있는가? 숙고해보면 해가 명백함을 볼 수가 있다. 즉, 연결리스트는 비어있는 공간의 상태를 파악하는데 이용된다! 이같은 리스트를 “자유 리스트(free list)”로 언급케 된다. 그때, 리스트에서 노드를 삭제할 때, 그것을 비어있는 리스트에 삽입시키므로써 처리되고 new 노드를 필요로할 때, 비어있는 리스트에서 삭제시키므로써 얻어진다. 이같은 메카니즘은 같은 배열을 차지하기 위해서 여러 가지 다른 리스트들을 허용케 된다.

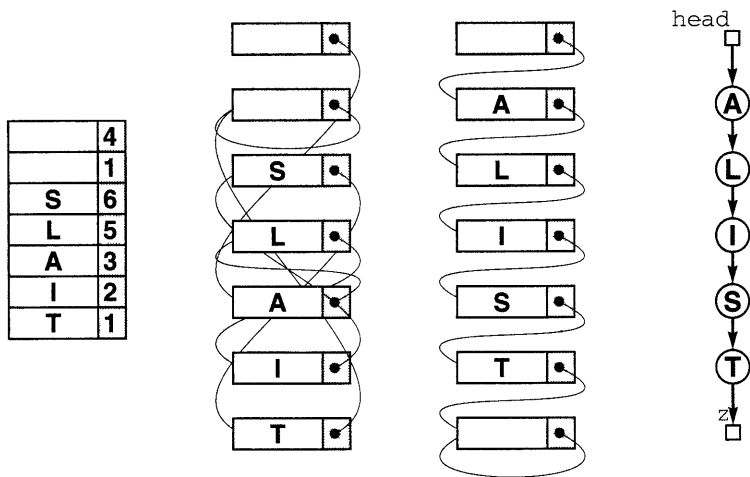


그림 3.5 연결 리스트의 배열 구현

두 개 리스트(그러나 비어있지 않은 리스트)로서 된 간단한 예제가 그림 3.6에 제시되어있다. 두 개 리스트 헤더 노드 $hd1 = 0$ 과 $hd2 = 6$ 이 존재하나 양 리스트는 같은 z 를 공유케 된다. class구조를 이용한 전형적인 C++ 구현은 각 리스트에 시작과 끝 노드를 지닌다. 여기서, $next[0]$ 은 4이므로, 첫 번째 리스트에 첫 번째 항목은 $key[4]$ (O)이다. $next[6]$ 은 7이므로해서, 두 번째 리스트에 첫 번째 항목은 $key[7]$ (T) 등이다. 그림 3.6에서의 다른 그림은 그림 3.5에서 처럼 노드들을 해결하고 간단한 그림적 표현으로 변경시키므로써 $next$ 값들을 선들로서 대체되는 결과를 나타낸다. 이같은 기법은 위에 기술된 것과 같이 비어있는 리스트가 되는 것중의 하나인 같은 배열에서 여러 가지 리스트들을 유지시키는데 이용된다.

기억 장소 관리가 C++에서 처럼 시스템에서 제공될 때, 이같은 방법으로 무시해야할 이유가 없다. 위의 기술은 기억장소 관리가 시스템에서 어떻게 수행이 되는가를 나타내는 것이다.(만약 여러분 시스템에서 기억장소 관리를 수행하지 못하면, 위의 기술은 구현에 대한 시작점을 제고케 된다) 시스템에서 직면한 실제적인 문제는 모든 노드들이 반드시 같은 크기를 필요치 않은 것처럼 훨씬 더 복잡하다. 또한, 몇몇 시스템은 어떤 연결에 의해서 참조되지 않은 어떤 노드를 자동적으로 제거 시키기 위한 “쓰레기 수집(garbage-collection)” 알고리즘을 이용해서 명백히 노드들을 삭제시킬 필요에 대한 것을 경감시킨다. 수 많은 좋은 기억장소 관리 알고리즘들은 이같은 두 가지 상황을 처리하기 위해서 개발된다.

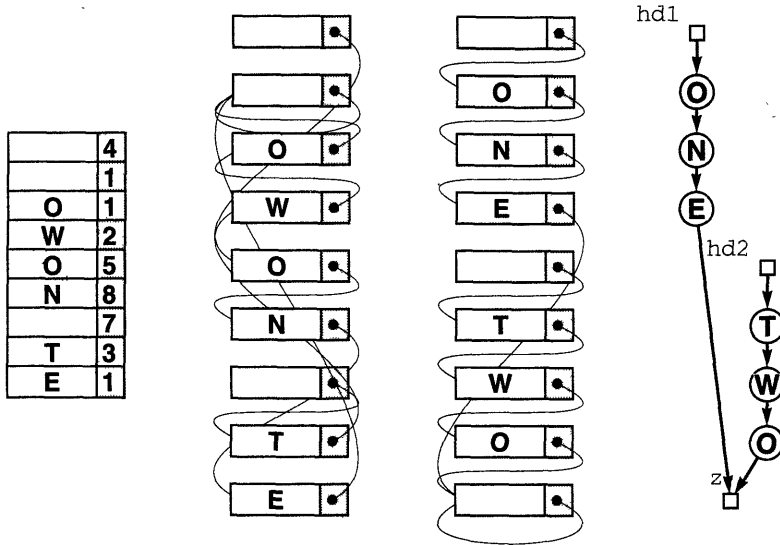


그림 3.6 같은 공간을 공유하는 두 개 리스트들

스택 삽입(Pushdown Stacks)

임의적으로 항목들을 삽입, 삭제 혹은 접근하기 위하여 데이터를 구조화하는데 중점을 두자. 실제로 많은 응용에서 자료구조가 어떻게 접근되는가에 여러 가지(오히려 엄중한) 제약을 고려하기에 충분하다고 판명되었다. 이같은 제약은 다음 두 가지 방법에서 이득이 있다. 첫째, 상세한 것으로 관련된 자료구조를 이용해서 프로그램에 대한 필요성을 경감시킬 수가 있다.(예를 들면, 항목들의 연결들이나 인덱스들의 상태를 파악하는 것이다) 둘째, 적은 연산만을 제공하므로써 훨씬 간단하고 더 유연한 구현을 허용케 된다.

가장 중요한 제약을 둔 접근 자료구조는 스택 삽입(pushdown stack)이다. 단지 두 개의 연산이 포함된다. 즉, 하나는 항목을 스택에 삽입하는 것이고(상단에 삽입) 다른 하나는 항목을 삭제시키는 것이다.(상단에서 제거) 스택은 박스 “내”에서 아주 바쁜 관리자들의 일처럼 연산을 한다. 즉, 스택에 일거리를 파일형태로 일을 쌓아놓고, 관리자들이같은 작업을 수행할 때마다 제일 위에 있는 일을 꺼내서 처리하는 것과 같다. 이것은 어느 순간에 스택의 하단에 무엇인가를 고정시켜 놓은 것과 같은 의미이다. 그러나 훌륭한 관리자는 주기적으로 스택을 비어있도록 하게 될 것이다. 가끔 컴퓨터 프로그램은 이같은 방법으로 자연스럽게 구성하고 다른 일을 수행하는 동안에 몇몇 일은 지연을 시키고, 이와 같이 해서 스택을 삽입시키는 것은 많은 알고리즘에 대해서 기본적인 자료구조로서 나타난다.

스택상에 유지된 항목들의 형은 유연성을 제공하기 위해서 이름 `itemType`로서 설명되지 않고 남겨둔다. 예를 들면, 문장 `typedef int itemType;`는 스택으로 하여금 정수형으로 유지되도록 하는데 이용된다. 이같은 것에 대한 C++의 대안은 이 장의 끝에서 논의가 된다.

위의 코드는 여기서 고려해야 할 여러 가지 C++ 구조를 제시한 것이다. 구현은 C++의 “사용자 정의 형”인 `class`이다. 이같은 정의로서, `Stack`은 다른 내장 형종의 `int`, `char`이나 다른 어떤 것으로서 같은 상태를 지니다. 구현은 두 가지 부분으로 구분된다. 즉, 데이터가 구성되어지는 방법을 설명하는 `private`부분과 데이터에 허용된 연산을 정의하는 `public`부분이다. 스택을 이용한 프로그램들은 `public`부분을 단지 참조하는데 필요하고 스택이 어떻게 구현되는가에 관심을 둘 필요는 없다. 함수 `Stack`는 그것이 먼저 정의가 될 때 스택을 초기화시키고 생성하는 “구성자”이고 함수 `~Stack`는 더 이상 필요가 없을 때 스택을 삭제하는 “파괴자”이다. 짧은 함수를 부르는데 적합한 키워드 `inline`은 함수 부름의 불필요한 연산을 제거시키는 것을 의미한다.

다음과 같이 본 장에서 스택 응용에 대한 많은 예제를 보기로 하자. 즉, 입문적인 예제로서, 산술식을 계산하는데 있어서 스택을 이용함을 보게 될 것이다. 다음과 같은 정수의 덧셈과 곱셈을 포함한 간단한 산술식의 값을 찾겠다고 가정해 보자.

$$5 * ((9 + 8) * (4 * 6)) + 7$$

계산은 어떤 중간적인 결과를 보관시키는 것이 필요케 된다. 예를 들면, 먼저 $9 + 8$ 을 계산하고 그후에 17은 잠시동안 어느곳에서간 가령 $4 * 6$ 이 계산될때까지 보관이 되어져야만 한다. 스택은 그런 계산에서 중간적인 결과를 보관시키는 이상적인 메카니즘으로 판명된다.

우선 먼저, 식을 재 배열을 하고 각 연산자는 그것의 두 값들 뒤에(사이가 아니고) 나타내게 된다. 위의 예는 다음 식에 대응된다.

$$5\ 9\ 8\ +\ 4\ 6\ *\ * 7\ +\ *$$

이것을 역 폴리쉬(reverse Polish) 기법(이것이 Polish 논리학자에 의해서 소개되었으므로) 또는 후위 표기(postfix)라고 부른다. 산술식을 기술하는 통례적인 방법을 중위 표기(infix)라 부른다. 후위 표기의 흥미로운 속성은 괄호가 요구되지 않는다는 것이다. 즉 중위 표기에서, $5 * ((9 + 8) * (4 * 6)) + 7$ 과 $((5 * 9) + 8) * ((4 * 6) + 7)$ 을 구별하는 것이 필요하다. 후위 표기의 더 흥미로운 속성은 스택에 중간적인 결과를 보관시키면서 계산을 수행하는 간단한 방법을 제공하는

것이다. 다음 프로그램은 후위 표기식으로 읽고, “스택에 오퍼랜드를 삽입시키기 위한” 명령어로서 각 오퍼랜드를 해독하고 “스택에서 두 개 오퍼랜드를 삭제시키고 연산을 수행하고 결과를 삽입시키기” 위한 명령어로서 각 연산자를 해독하는 것이다.

```
char c; Stack acc(50); int x;
while ( cin.get(c) )
{
    x = 0;
    while ( x == ' ' ) cin.get(c);
    if ( c == '+' ) x = acc.pop( ) + acc.pop( );
    if ( c == '*' ) x = acc.pop( ) * acc.pop( );
    while ( c >= '0' && c <= '9' )
        { x = 10 * x + ( c - '0' ); cin.get(c); }
    acc.push(x);
}
cout << acc.pop( ) << '\n';
```

스택 삽입 save는 다른 프로그램 변수들에 따라서 선언되고 정의된다. 그리고 save에 대한 push와 pop연산은 입력 스트림 cin에 대한 get과 같은 방법으로 상세하게 된다. 이같은 프로그램은 정수형의 곱셈과 덧셈을 수반으로 한 어떤 후위 표기식을 읽고서 식의 값을 인쇄케 된다. 빈칸은 무시가 되고 while 반복은 문자 형에서 계산에 대한 숫자 형으로 정수를 변형시키게 된다. 그렇지 않으면, 프로그램의 연산은 간단하다. C++에서, 두 개 pop() 연산이 수행되는 순서가 설명이 되어지지 않으므로해서, 다소 더 복잡한 코드는 뺄셈과 나눗셈과 같은 비 교환 법칙에 대해 필요케 된다.

다음 프로그램은 합법적으로 완전히 괄호화된 중위 표기식을 후위 표기식으로 변경 시킨 것이다.

```
char c; Stack save(50);
while ( cin.get(c) )
{
    if ( c == ')' ) cout.put(dave.pop());
    if ( c == '+' ) save.push(c);
    if ( c == '*' ) save.push(c)
    while ( c >= '0' && c <= '9' )
```

```

        { cout.put(c); cin.get(c); }
        if ( c != '(' ) cout << ' ';
    }
    cout << '\n';

```

연산자는 스택에 삽입되고 인수들은 단순히 지나가게 된다. 이와 같이 인수들은 중위 표기와 같은 순서로서 후위 표기식에서 나타난다. 각각 오른쪽 괄호는 마지막 연산자에 대한 양쪽 인수들이 출력으로 이루어짐을 나타내고, 연산자 그 자체는 삭제되거나 출력된다. 정확히 두 오퍼랜드에 대해 연산자들을 이용하기 때문에, 왼쪽 괄호는 중위 표기식에서는 필요가 없다는 것은 놀라만한 사실이다.(그리고 이같은 프로그램은 그것들을 넘어간다) 간단히 말해서, 이같은 프로그램은 입력에서 오류들에 대해 체크를 하지 않고 연산자, 괄호와 오퍼랜드 사이에 공간을 요구한다.

“중간값을 보관하는” 보기는 기본적이고 스택 삽입은 가끔 일어난다. 많은 기계들은 그것들이 함수 부름 메카니즘을 자연스럽게 구현하기 때문에 하드웨어에서 기본적인 스택 연산을 구현한다. 즉, 스택상에 정보를 삽입시키는 프로시저에 대한 입구에 현재 환경을 보관하고, 스택에서 삭제하는 정보를 이용함으로써 출구에 환경을 다시 보관케 된다. 몇몇 계산기와 몇몇 계산 언어들은 명백히 스택 연산에 계산 방법을 기초로 한 것이다. 즉, 모든 연산은 스택에서 그것의 인수를 삭제하고 결과를 스택에 되돌려 준다. 5장에서 보듯이, 명확하게 이용되지 않을 때 조차 스택은 내재적으로 가끔 일어나게 된다.

스택들의 연결 리스트 구현

그림 3.7은 많은 수의 스택 연산이 존재할 때 조차 적은 양의 스택이 충분히 전형적인 경우를 나타낸 것이다. 만약 이것이 그 경우로 확신이 된다면, 그때 위에 주어진 배열 표현은 적절하다. 그렇지 않으면, 연결 리스트가 스택으로 하여금 좋은 방향으로 증가되고 축소된다. 그리고 많은 그런 자료구조들중의 하나인 경우에 특히 유용하다. 연결 리스트를 이용한 기본적인 스택 연산을 구현하는 것은 인터페이스를 정의함으로써 시작된다.

```

class Stack
{
    public :

```

```

    Stack(int max);
    ~Stack( );
    void push(itemType v);
    itemType pop( );
    int empty( );
private ;
    struct node
    { itemType key; struct node *next; };
    struct node *head, *z;
};

```

C++에서 그런 인터페이스는 두 가지 목적으로 서비스된다. 즉, 스택을 이용하기 위해서, 어떤 연산이 제공되는가를 배우기 위한 인터페이스의 public 부분을 보면되고, 스택 루틴을 구현하기 위해서는 구현과 관련된 기본적인 자료구조들을 배우기 위해서 private 부분을 보면 된다. 스택 프로시저의 구현은 분리된 파일로서 포함이 될지라도 class선언과 분리가 되어야한다. 인터페이스와 구현을 분리시키는 능력과 그러므로해서 다른 구현으로 쉬운 실험은 이 장 끝에 상세히 설명할 C++의 바로 중요한 특징인 것이다.

다음은 스택을 선언할 때 생성되는 “구성자”가 필요하고 더 이상 필요가 없을 때(범위를 벗어날때), 삭제하는 “파괴자”가 필요케 된다.

```

Stack::Stack(int max)
{
    head = new node; z = new node;
    head->next = z; z->next = z;
}
Stack::~Stack()
{
    struct node *t = head;
    while ( t != z )
        { head = t; t = t->next; delete head; }
}

```

마지막으로, 스택 삽입 연산의 실제적인 구현을 보자.

```

void Stack::push(itemType v)
{
    struct node *t = new node;
    t->key = v; t->next = head->next;
    head->next = t;
}
itemType Stack::pop()
{
    itemType x;
    struct node *t = head->next;
    head->next = t->next; x = t->key;
    delete t; return x;
}
int Stack::empty()
{ return head->next == z; }

```

연결 리스트와 스택 삽입 양쪽의 이해를 증진시키기 위해서 이같은 코드를 조심스럽게 공부해야만 한다.

큐(Queues)

또 다른 기본적인 제한된 접근 자료구조는 큐(queue)이다. 단지 두 가지 기본적인 연산이 수반된다. 즉, 제일 앞에 항목을 큐에 삽입시키는 것과 끝에서 항목을 제거시키는 것이다. 아마도 박스 “내”에서 바쁜 관리자의 작업이 먼저 도착한 작업을 먼저 처리되는 큐와 같이 처리된다. 스택상에서, 어떤 것은 끝에 묻혀 있으나 큐에서 모든 것들은 받아들여진 순서대로 처리된다.

그림 3.8은 다음 순서로서 표현된 연속적인 get과 put을 통해서 예제 큐가 어떻게 처리되는가를 나타내고 있다.

A * S A * M * P * L E * Q * * * U * E U * * E *

여기서 리스트상에서의 각 문자는 “삽입(put)”이고 *는 “삭제(get)”이다.

비록 스택은 재귀와의 기본적인 관계 때문에 큐보다 더 가끔 이용이 되지만(5장 참조), 큐는 자연스런 자료 구조인 알고리즘으로 보게 될 것이다. 20장에서 스택과 큐의 조합인

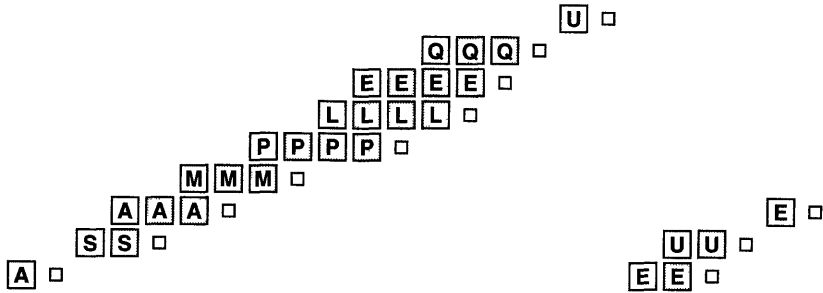


그림 3.8 큐의 동적 특성

deque(또는 “double-ended queue”)를 보게 되고 4장과 30장에서는 트리와 그래프의 불법적인 이용을 허용하도록 하기 위한 메카니즘으로서 큐의 응용이 포함된 기본적인 예제들을 논하게 된다. 스택은 가끔 “마지막에 들어온 것이 먼저 출력되는(LIFO)” 방법이고 큐는 “먼저 들어온 것이 먼저 출력되는(FIFO)” 기법이다.

큐 연산의 연결 리스트 구현은 간단하므로 연습문제로 남겨둔다. 스택과 같이, 배열은 역시 put, get 과 empty 함수의 다음 구현에서 처럼 최대 크기를 추정할 수 있는 경우에 또한 이용된다.(스택 삽입의 배열 구현에 대한 위에 주어진 코드와 비슷하기 때문에 여기서는 인터페이스, 구성자와 파괴자 함수를 생략케 된다)

```
void Queue::put(itemType v)
{
    queue[tail++] = v;
    if ( tail > size ) tail = 0;
}
itemType Queue::get()
{
    itemType t = queue[head++];
    if ( head > size ) head = 0;
    return t;
}
int Queue::empty()
{ return head == tail; }
```

여기에는 세 가지 class변수가 있다. 즉, 큐의 크기와 두 개 인덱스로 하나는 큐의 제일 앞에(head), 다른 하나는 끝에(tail)있다. 큐의 내용은 배열의 끝에 도달될 때 다시 0으로하

고, head와 tail사이 배열에서의 모든 구성요소이다. 만약 head와 tail이 동등하면, 그때 큐는 비어 있는 것으로 정의가 된다. 그러나 put으로 인해 그것들이 똑같아지면, 그때 큐가 꽉찬 것으로 간주가 된다.(비록 위의 코드에서는 이같은 내용을 체크하지는 않았지만) 이같은 것은 배열의 크기가 큐에서 적합한 구성요소의 최대 수보다 한 개 큰 것으로 요구가 된다. 즉, 꽉찬 큐는 한 개 비어 있는 배열의 위치를 가르킨다.

추상적이고 구체적인 자료 형 (Abstract and Concrete Data Types)

상세한 구현에 의해서라기 보다는 오히려 수행된 연산에 의해서 알고리즘과 자료구조를 기술할 때가 가끔은 편리할때가 있음을 위에서 보았다. 이같은 방법으로 자료구조를 정의할 때, 그것을 추상적 자료형(abstract data type)라고 부른다. 생각은 어떤 특정한 구현에서 자료구조가 무엇을 수행하는가의 “개념”을 분리시키는 것이다.

추상적 자료형의 특징을 정의함은 기본적인 연산에 대한 함수와 프로시저 부름을 통한 것은 제외하고 어떤 것 내부를 언급하는 곳에서 연산을 하는 자료구조와 알고리즘 정의의 외부에 아무것도 없는 것이다. 추상적 자료형의 발전에 대한 주된 동기는 큰 프로그램을 구성함에 대한 메카니즘으로서 존재케 된다. 추상적 자료형은(잠재적으로 복잡한) 알고리즘과 관련된 자료구조와 알고리즘과 자료구조를 이용한(잠재적으로 큰) 프로그램들 사이에 인터페이스의 크기와 복잡도를 제한하는 방법을 제공한다. 이것은 큰 프로그램을 이해하기 쉽게하고 기본적인 알고리즘을 증진시키거나 변경시키는 것이 더 편리하다.

스택과 큐는 추상적 자료형의 고전적인 예제이다. 즉, 대부분 프로그램들은 연결과 인덱스들의 상세한 것은, 아니지만 몇 가지 잘 정의된 기본적인 연산에만 관심을 필요로한다.

배열과 연결 리스트는 차례로 선형 리스트(linear list)라 부르는 기본적인 추상적 자료형의 정교한 내용으로 간주가 된다. 그것들의 각각은 순차적으로 순서화된 항목들의 기본적으로 중요한 구조에 삽입, 삭제와 접근과 같은 연산을 수행케 된다. 이같은 연산들은 알고리즘을 기술하기에 충분하고 선형 리스트 추상화는 알고리즘 생성의 첫 번째 단계에서 유용하게 된다. 그러나 본는 바와 같이, 어느 연산들이 이용되는가를 정의하는 것은 바로 프로그래머의 관심이다. 그 이유는 다른 구현들이 아주 다른 활용도 특징을 지니기 때문이다. 예를 들면 Eratosthenes의 추출에 대해서 배열 대신에 연결 리스트를 이용함은 알고리즘의 효율성이 어

면 배열 위치에서 다른 것으로 빠르게 얻어질 수 있는 것에 의존되기 때문에 비용이 들며, 요세퍼스(Josephus) 문제에 대해서 연결 리스트 대신에 배열을 이용하는 것은 알고리즘의 효율성이 삭제된 요소들의 없어짐에 의존되기 때문에 비용이 든다.

더 많은 연산들을 효율적으로 유지시키도록 하기 위해서, 훨씬 더 복잡한 알고리즘과 자료구조들을 요구하는 선형 리스트에 그 자체들을 제안케 된다. 두 가지 중요한 연산은 키의 증가되는 순서로서 항목을 정렬하는 것과(8-13장의 주제) 특별한 키로서 항목을 찾는 것(14-18장의 주제)이다.

한 추상적 자료 형은 다른 것을 정의하는데 이용된다. 즉, 스택과 큐를 정의하기 위해서 연결 리스트와 배열을 이용케 된다. 연결 리스트를 생성하기 위해서 C++에서 제공된 “포인터”와 “레코드” 추상화를 이용하고 배열을 생성하기 위해서 C++에서 제공된 “배열” 추상화를 이용케 된다. 부가적으로 배열로서 연결 리스트를 생성할 수 있다는 것을 위에서 볼 수가 있고 배열은 연결 리스트로서 기쁨 생성이 되어진다는 것을 36장에서 볼 수가 있다. 추상적 자료 형의 실제적인 힘은 컴퓨터에서 제공된 기계어 명령어에서 프로그래밍 언어에 제공된 여러 가지 능력에 이르기 까지 그리고 본 교재에서 논의된 것과 같이 알고리즘에서 제공된 정렬, 검색과 다른 고급 능력에 이르기까지와 응용에서 제시된 추상화의 더 높은 레벨들에 이르기까지 추상화의 다른 레벨상에 큰 시스템을 구성시키는데 편리하도록 하는 것이다.

본 교재에서 관련된 자료구조와 오히려 상당히 밀접하게 된 상대적으로 적은 프로그램들을 처리케 된다. 알고리즘과 자료구조사이의 인터페이스에서 추상화에 대해서 이야기 하는 것이 가능한 반면에, 추상화의 더 높은 레벨들(응용에 더 근접된 것)에 초점을 맞추는 것이 실제적으로 더 적절하다. 즉, 추상화의 개념은 가장 효율적인 해를 찾는데서 특별한 문제에 이르기까지 주위를 돌리지는 못한다. 여기서 활용도가 문제가 된다는 견해를 취한다! 이같은 생각으로 개발된 프로그램은 큰 시스템에 대한 추상화의 고급 레벨을 개발하는데 어떤 자신을 가지고 이용케 된다.

이 장에서 스택과 큐 연산의 구현은 그곳들에서 수행되는 알고리즘과 자료구조를 함께 묶는 것인 구체적 자료 형(concrete data types)의 예제들이다. 본 교재에서 이같은 그림을 가끔 이용케 된다. 그 이유는 응용에 이용을 위해서 널리 사용되도록 하는 코드를 동시에 개발하는 반면에 기본적인 알고리즘을 기술하는데 편리한 방법이기 때문이다. C++는 진정한 추상적 자료 형을 구현하기 위해서 “class 계층구조”와 “가상적 함수”를 이용하므로써 방법을 제공케 된다. 여기서 함수들로(데이터 표현이 아니고)만 구성된 인터페이스가 제공이 된다.

여기서의 초점은 진정한 추상적 자료 형을 이용할 때 유지하기가 어려운 활용도 특징들의 인지에 관한 것이다.

위에 언급한 바와 같이, 실제적 자료구조는 드물게 정수들과 연결들로서 간단히 구성되는 것이다. 노드들은 가끔 많은 정보들이 포함이 되고 여러개 독립된 자료구조에 속하게 된다. 예를 들면, 개인 데이터 파일은 고용인에 대한 이름, 번지와 다른 정보형태로 된 레코드를 포함한다. 그리고 각 레코드는 특정한 고용인에 대해 검색을 하기 위한 한 개의 자료구조와 통계적 질의에 대한 답변을 하기 위한 것 등의 또 다른 자료 구조에 속하는 것이 필요케 된다. C++는 간단한 알고리즘을 복잡한 구조상에 처리되도록 확장시키는데 쉬운 방법을 제공하는 template 기법이라 부르는 일반적인 메카니즘을 지닌다. typedef를 이용하기 보다는 오히려, template <class itemType> 코드는 본 장에서 class 정의가 어떤 형으로 처리되는 구조들의 정의로 변경시키기 전에 놓여진다. 예를 들면, float들의 스택을 생성하기 위해서 이용이 되어진 Stack<float> acc()와 같이 선언이 되도록 한다. 본 교재에서, 항상 정수 형으로 처리가 되나 typedef들이나 template들은 적절하게 응용에서 쉽게 이용이 됨을 이해함으로써 이 장에서 처럼 형들을 설명되지 않은 채로 남겨둔다.

연습문제

1. i 와 j 의 최대 공약수가 1인 경우에 $a[i][j]$ 를 1로 세트하고 그렇지 않으면 0으로 세트 시킴으로써 부울리언 값들의 2차원 배열로 채우는 프로그램을 기술하시오.
2. 리스트의 앞에 t 에 의해서 가르키는 노드 다음으로 노드들을 이동시키는 연결 리스트에 대해서 루틴 `movenexttofront(struct node *t)`를 구현케 된다.(그림 3.3은 t 가 리스트에서 다음번에서 마지막까지의 노드를 가르키는 특별한 경우에 대한 예제이다)
3. t 와 u 에 의해서 가르키는 노드들 뒤에 노드들의 위치를 변경시키는 연결 리스트에 대한 루틴 `exchange(struct node *t, struct node *u)`를 구현하라.
4. 연결 리스트 대신에 배열을 이용하여 요세퍼스(Josephus)문제를 해결하기 위한 프로그램을 기술하라.
5. 이중 연결 리스트에서 삽입과 삭제에 대한 프로시저를 기술하라.
6. 병렬 배열을 이용해서 삽입 스택의 연결 리스트 구현에 대한 프로시저를 기술하라.
7. 순서 $E A S * Y * * Q U E * * * S T * * * I * O N * *$ 에서 각 연산뒤에 스택의 내용은 무엇인가? 여기서 문자는 “삽입”을 의미하고 *는 “삭제”를 의미한다.
8. 순서 $E A S * Y * * Q U E * * * S T * * * I * O N * *$ 에서 각 연산뒤에 큐의 내용은 무엇인가? 여기서 문자는 “삽입”을 의미하고 *는 “삭제”를 의미한다.
9. 초기에 비어있는 리스트에서 그림 3.5를 생성한 `deletenext`와 `insertafter`에 대한 연속적인 부름은 무엇인가?
10. 연결 리스트를 이용하여 큐에 대한 기본적인 연산을 구현하시오.

빈 면

4 장

트리

3장에서 논의된 구조는 본질적으로 1차원 배열에 관한 것이다. 즉, 한 항목 다음에 다른 항목이 따른다. 본 장에서는 많은 중요한 알고리즘의 심장부인 트리(trees)라 부르는 2차원 연결 구조를 고려케 된다. 트리의 완전한 논의가 본 교재의 전체에 걸쳐서 기술된다. 그 이유는 전산학 이외에서도 트리가 많이 이용되고 있으며 수학에서도 광범위하게 이용되기 때문이다. 본 교재에서는 트리들에 대한 논의로 가득 차 있다고 말할 수가 있다. 본 장에서는 트리에 관련된 기본적인 정의와 용어를 설명하고 컴퓨터 내에서 트리를 표현하는 방법에 대해서 보게 될 것이다. 다음 장들에서는 이것은 기본적인 자료구조상에 처리되는 많은 알고리즘을 보게 된다.

트리는 매일의 생활에서 가끔 보게 되며 여러분들은 기본적인 개념에 확실히 친숙해야만 한다. 예를 들면, 많은 사람들은 족보 트리로서 조상들과/혹은 후손들의 상태를 파악케 된다. 우리가 볼 수 있듯이, 많은 용어들이 이같은 용법에서 이끌어진다. 또 다른 예는 스포츠 토너먼트 구조에서 볼 수도 있다. 즉, 11장에서 볼 수 있는 이같은 용법은 Lewis Carroll에 의해서 연구가 된 것이다. 세 번째 예제는 큰 회사의 조직 표에서 볼 수가 있다. 이같은 용법은 많은 전산학 응용에서 보게 된 “계층적 분해”의 암시적인 것이다. 4번째 예제로는 영어 문장을 성분 부분으로 분할한 “파서(parse) 트리”이다. 이것은 21장에 더 많이 논의 할 것으로 컴퓨터 언어의 처리와 궁극적으로 관련된 것이다. 다른 예제들은 교재를 통해서 논의가 된다.

용어 풀이(Glossary)

트리를 추상적인 물체로서 정의를 하고 기본적으로 관련된 용어들 대부분을 소개함으로써 트리에 대한 논의를 시작해 보자. 트리를 정의하는데는 수많은 동등한 방법들이 있고 이같은 동등성을 함축하는 많은 수학적 성질을 지닌다. 이것들에 대한 자세한 내용은 다음절에서 보기로 하자.

트리(tree)는 어떤 요구 조건을 만족하는 정점들(vertices)과 선분들(edges)의 비어 있지 않은 집합을 의미한다. 정점은 이름을 지니고 다른 관련된 정보를 나타내는 간단한 물체(또한 노드(node)로 언급)이다. 반면에 선분은 두 정점들 사이의 연결이다. 트리에서 경로(path)는 연속적인 정점들이 트리에서 선분들에 의해서 연결된 명확한 정점들의 리스트이다. 트리에서 한 노드는 근(root)으로서 지시가 된다. 즉, 트리의 정의되는 성질은 근과 트리에서 다른 노드들 사이에는 정확히 한 경로를 지닌다. 만약 근과 어떤 노드 사이에 경로가 한 개 이상인 경우나, 근과 어떤 노드 사이에 경로가 없으면, 그때 트리가 아니고 그래프가(29장 참조) 된다. 그림 4.1은 트리의 예제를 나타낸 것이다.

정의에서 선분상에 “방향”은 나타내지 않았지만, 선분들은 근에서부터 멀리 떨어지는 모든 것으로(그림 4.1에서 밑으로 가는 것) 혹은 응용에 따라서 근 노드로 향하고 있는 것(그림 4.1에서 위로가는 것)으로 간주된다. 트리를 제일 위의 근에서 시작해서 그리는 것이 보통이고 x 가 y 에서 근에 이르는 경로상에 존재하면(즉, y 가 x 밑에 존재하고 근을 통해서 전달되지 않는 경로에 의해서 x 에 연결이 되는 경우이다) 노드 y 는 노드 x 밑에 존재케 된다.(그리고 x 는 y 위에 존재케 된다) 각 노드(근을 제외)는 자신의 상위에 부모(parent)라 부르는 노드가 정확히 한 개 있다. 즉, 어떤 노드 바로 아래에 있는 노드를 자식(children)이라고 부른다. 가끔 족보에서 보듯이 어떤 노드의 “할아버지(grandparent)”나 “형제(sibling)”를 언급케 된다. 그림 4.1에서 P 는 R 의 손자이고, 형제가 세 개나 된다.

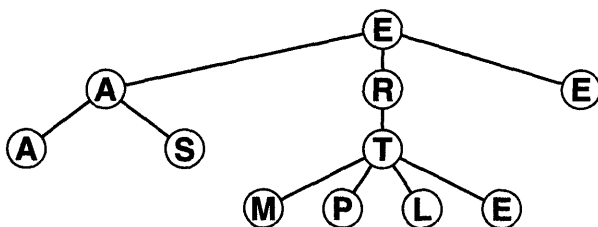


그림 4.1 예제 트리

자식이 없는 노드를 잎(leaves) 혹은 터미널 노드(terminal node)라고 부른다. 나중에 용어를 통일시키기 위해서, 적어도 한 개 자식을 지닌 노드를 논 터미널(nonterminal) 노드라고 부른다. 터미널 노드들은 논 터미널 노드들과는 다르다. 예를 들면, 그것들은 이름이나 관련된 정보가 없다. 특히 그런 상황에서, 논 터미널 노드를 내부(internal) 노드로 터미널 노드를 외부(external) 노드로 언급케 된다.

어떤 노드는 그 노드를 구성하는 부분 트리(subtree)의 근이고, 그 노드 아래에는 노드들이 존재케 된다. 그림 4.1에 제시된 트리에서 한 개로 구성된 부분 트리가 7개이고 세 개 노드로 구성된 부분 트리가 3개이고, 5개 노드로 구성된 부분 트리가 1개 그리고 6개 노드로 구성된 부분 트리가 1개로 되어져 있다. 트리의 집합을 포리스트(forest)라고 부른다. 예를 들면, 근을 제거하고 그림 4.1의 트리에서 근과 연결된 선분들을 제거시키면, 근 노드가 A, R과 E로 된 세 개의 트리로 구성하는 포리스트로 된다.

가끔 각 노드의 자식들이 순서화 되어지는 방법은 의미가 있는 경우도 있고 그렇지 않은 경우도 있다. 순서화된(ordered) 트리는 모든 노드에서 자식들의 순서가 의미를 지닌 트리이다. 물론 자식들은 트리를 그릴 때 어떤 순서로서 놓여지고 순서화되지 않은 트리를 그리기 위한 방법은 많이 존재케 된다. 아래에서 보듯이, 이같은 차이는 컴퓨터에서 트리를 표현할 때 의미가 있다. 그 이유로는 순서화된 트리를 표현하는 방법에서 유연성이 적기 때문이다. 트리의 형태는 불려지는 응용에서 항상 명백하다.

트리에서 노드는 그 자체가 레벨(levels)들로 구분된다. 즉, 한 노드의 레벨은 특정 노드에서 근(자신을 불 포함)에 이르는 경로상에 있는 노드의 수이다. 예를 들면, 그림 4.1에서 R은 레벨 1이고, S는 레벨 2이다. 트리의 높이(height)는 트리에서 모든 노드들 사이의 최대 레벨이다.(또는 어떤 노드에서 근에 이르는 최대 거리) 트리의 경로 길이(path length)는 트리에서 모든 노드들의 레벨의 합이다.(또는 각 노드에서 근에 이르는 경로 길이의 합) 그림 4.1에서 트리의 높이는 3이고 경로 길이는 21이다. 만약 내부 노드들이 외부 노드와 구별이 되면, 내부 경로 길이와 외부 경로 길이에 대해 생각을 할 수가 있다.

만약 각 노드가 특별한 순서로서 나타나는 특별한 자식들의 수를 지녀야만 한다면, 그때 다중 방법(multiway) 트리를 지닌다. 그런 트리에서 자식이 없는 특별한 외부 노드를 정의하기에 적합하다.(그리고 항상 이름이나 관련된 정보를 지니지 않고 있다) 그때, 외부 노드들은 자식이 특별한 수를 지니지 않는 노드를 위해 “가상(dummy)” 노드로서 취급된다.

특히, 다중 방법 트리의 가장 간단한 형태가 이진 트리(binary tree)이다. 이진 트리는 노드들의 두 가지 형태로 구성된 순서화 트리이다. 즉, 자식이 없는 외부 노드들과 정확히 두 개

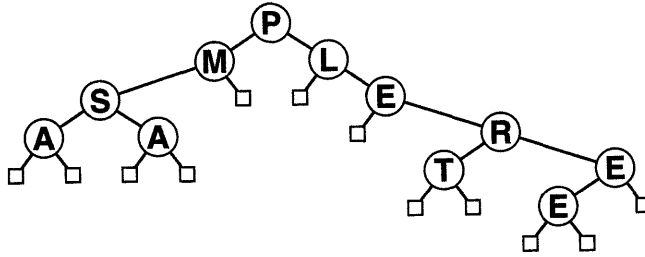


그림 4.2 예제 이진 트리

자식을 가진 내부 노드들이다. 이진 트리의 예제는 그림 4.2에 제시가 되어 있다. 각 내부 노드들의 두 개 자식은 순서화가 되어서므로, 각각을 내부 노드들의 왼쪽 자식(left child)과 오른쪽 자식(right child)이라고 언급을 한다. 즉, 모든 내부 노드는 비록 그것들의 둘 다나 혹은 한쪽이 외부 노드이라고 할지라도 왼쪽과 오른쪽 자식 둘 다를 지녀야만 한다.

이진 트리의 목적은 내부 노드들을 구조화시키는 것이다. 즉, 외부 노드들은 장소 보유자로서만 제공을 한다. 이진 트리에 대한 가장 공통적으로 이용된 표현은 각 외부 노드를 설명하기 때문에 정의에서 포함케 된다. 이진 트리는 내부 노드가 없고 외부 노드가 한 개인 “비어 있는” 것이다.

전(full) 이진 트리는 내부 노드들이 마지막의 것은 제외하고 모든 레벨에서 완전히 채워져 있는 트리이다. 정(complete) 이진 트리는 제일 밑에 있는 내부 노드들이 그 레벨상에 외부 노드들의 왼쪽에 나타나는 전 이진 트리이다. 그림 4.3은 정 이진 트리의 예이다. 보는 바와 같이, 이진 트리들은 컴퓨터 응용에서 광범위하게 이용되고, 이진 트리가 꼭 찬 경우(또는 거의 꼭 찬 경우) 활용도가 가장 좋게 된다. 11장에서 정 이진 트리를 기본으로 한 중요한 자료 구조에 대해서 보게 될 것이다.

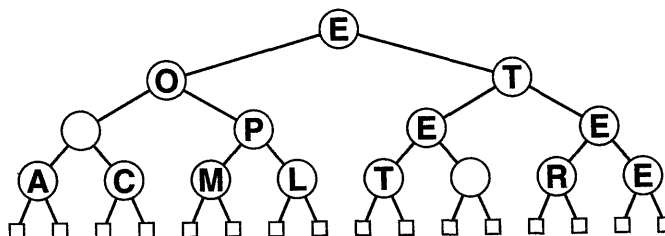


그림 4.3 정 이진 트리

여러분은 모든 이진 트리가 트리는 되지만 이진 트리는 되지 않는 점을 주시해야만 한다. 모든 노드들이 0, 1 혹은 2개의 자식을 지니는 순서화된 트리로서만 고려를 할 때조차, 1개의 자식을 지닌 노드들은 이진 트리에서 왼쪽이거나 오른쪽 중 한곳에 존재하기 때문에 각각 그런 트리는 많은 이진 트리들에 대응이 된다.

트리들은 다음 장에서 보게 될 재귀(recursion)와 밀접한 관계가 있다. 사실, 트리를 정의하는 가장 간단한 방법은 다음과 같다. 즉, “트리는 한 개 노드이거나 트리들의 집합에 연결된 근 노드이고” 그리고 “이진 트리는 외부 노드이거나 혹은 왼쪽 이진 트리와 오른쪽 이진 트리에 연결된 근(내부) 노드이다.”

성질(properties)

표현을 고려하기 전에, 수많은 트리의 중요한 성질들을 고려함으로써 수학적 특징을 계속해 된다. 고려해야 할 광대한 양의 성질들이 존재해 된다. 즉, 목적은 본 교재에서 고려해야 할 알고리즘에 특별히 관련된 것들을 고려하는 것이다.

성질 4.1 트리에서 어떤 두 노드를 연결하는데는 정확히 한 경로가 있다.

어떤 두 노드는 최소 공통 선조(least common ancestor)를 지닌다. 즉, 양쪽 노드에서 근에 이르는 경로상의 노드이다. 그러나 같은 성질을 지닌 자식들은 없다. 예를 들면, 그림 4.3의 트리에서 O는 C와 L의 최소 공통 선조이다. 근이 최소 공통 선조이거나 노드들의 둘 다가 근의 자식들중 하나에 근으로 된 부분 트리에 존재하기 때문에 최소 공통 선조는 항상 존재해 된다. 후자의 경우, 그 노드는 최소 공통 선조이거나 혹은 노드들의 양쪽이 그것의 자식들중의 하나로 근화된 부분 트리에서 존재해 된다. 노드들의 각각에서 최소 공통 선조에 이르는 경로가 존재한다. 즉, 이같은 두 경로를 함께 이어 맞추는 것은 두 노드를 연결하는 경로로 주어진다. □

성질 1의 가장 중요한 의미는 어떤 노드도 근으로 될 수가 있다는 것이다. 즉, 트리에서 각 노드는 트리에서 모든 다른 노드로서 된 그 노드를 연결하는 정확히 한 개의 경로가 존재해 된다. 기술적으로, 근이 인식되어지는 정의는 근화된 트리(rooted tree) 혹은 방향을 지닌 트리(oriented tree)에 관계된다. 즉, 근이 인식되어지지 않는 트리를 자유 트리(free tree)라 부른다. 여러분은 이같은 구별을 하는 것에 대해서는 관심을 둘 필요는 없다.

성질 4.2 N 개 노드들로 된 트리는 $N-1$ 개의 선분들을 지닌다.

이 성질은 근을 제외하고 각 노드가 유일한 부모를 지니고 모든 선분은 한 노드가 그것의 부모에 연결이 된다는 사실에서 직접적으로 따른다. 또한 이같은 사실은 재귀적 정의에 의해서 귀납적으로 증명된다. \square

다음 두 가지 성질은 이진 트리에 관련이 된 것이다. 위에 언급된 대로, 이같은 구조들은 본 교재를 통해서 가끔 발생된다. 그래서 그것들의 특징들에 주의를 두는 것은 가치가 있다. 이것은 우리가 직면할 여러 가지 알고리즘의 활용도 특징을 이해하는 토대가 된다.

성질 4.3 N 개의 내부 노드로 된 이진 트리는 $N+1$ 개의 외부 노드를 지닌다.

이 성질은 귀납법으로 증명된다. 내부 노드가 없는 이진 트리는 외부 노드가 한 개이므로 $N = 0$ 에 대한 성질은 유지가 된다. $N > 0$ 에 대해서 보면, 근은 내부 노드이므로, N 개 내부 노드를 가진 이진 트리는 0과 $N - 1$ 개사이의 어떤 k 에 대해서 왼쪽 부분 트리는 k 개의 내부 노드를 지니고 오른쪽 부분 트리는 $N - 1 - k$ 개의 내부 노드를 지닌다. 귀납적 가설에 의해서, 왼쪽 부분 트리는 $k + 1$ 개의 외부 노드를 지니고 오른쪽 부분 트리는 $N - k$ 개의 외부 노드를 가지며, 이들의 전체는 $N + 1$ 개의 노드를 지닌다. \square

성질 4.4 N 개의 내부 노드들로 된 어떤 이진 트리의 외부 경로 길이는 내부 경로 길이보다 $2N$ 배 크다.

이 성질 역시 귀납법으로 증명이 되지만 대안의 증명으로도 유익하다. 어떤 이진 트리가 다음 과정에 의해서 구성된다. 즉, 한 개 외부 노드로 구성된 이진 트리로부터 시작을 해 보자. 그때 다음을 N 번 반복케 된다. 한 개의 외부 노드를 선택해서 그것을 자식으로 두 개의 외부 노드로된 새로운 한 내부 노드에 의해 대체가 된다. 만약 선택된 외부 노드가 레벨 k 에서 존재하면, 내부 경로 길이는 k 씩 증가가 되나 외부 경로 길이는 $k + 2$ 씩 증가를 한다.(레벨 k 에서 한 외부 노드는 제거가 되나, 레벨 $k + 1$ 의 두 개는 첨가가 된다) 과정은 내부와 외부 경로 길이 둘 다가 0인 트리로서 시작해서 N 단계의 각각에 대해, 외부 경로 길이는 내부 경로 길이보다 2씩 증가가 된다. \square

마지막으로 “가장 좋은” 형태의 이진 트리 즉, 전 이진 트리의 간단한 성질을 고려해야 한다. 이같은 트리들은 그것의 높이가 낮은 것으로서 근에서 어떤 노드에 이르는데 작업이 그리 많이 요구가 되지 않기 때문에 관심이 집중이 되고 있다.

성질 4.5 N 개 이진 트리인 전 이진 트리의 높이는 약 $\log_2 N$ 이다.

그림 4.3을 보면, 높이가 n 인 경우에 $N+1$ 개의 외부 노드가 있으므로 다음을 얻을 수가 있다.

$$2^{n-1} < N + 1 \leq 2^n$$

이것은 성질에 나타난 것을 의미한다.(실제로, 높이는 거의 정수에 가깝도록 반올림한 $\log_2 N$ 과 정확히 같으나, 6장에서 논의 된 것과 같이 아주 상세하게 나타내는 것을 삼가할 것이다) □

트리의 더 많은 수학적 성질들은 다음 장에서 필요할 때 논의가 될 것이다. 여기서는 컴퓨터에서 트리를 표현하고 그것을 효율적인 방법으로 나타내는 실제적인 문제에 대해 집중을 할 것이다.

이진 트리의 표현

이진 트리의 가장 많이 쓰는 표현은 노드당 두 개의 연결을 이용한 레코드들의 이용이다. 정상적으로, 표현을 위해서 선택된 순서는 본 교재에서 트리가 그려진 방법에 대응이 되도록 나타내기 위해서 연결 이름들 l 과 r (“왼쪽”과 “오른쪽”의 축약)를 이용케 된다. 어떤 응용에서는 레코드의 두 가지 형태 즉, 하나는 내부 노드이고 다른 하나는 외부 노드에 대한 것을 지니는 것이 적절할 수가 있다. 다른 응용에서는 노드에는 단지 한 개의 형태를 지니고 어떤 다른 목적을 위해 외부 노드에서 연결을 이용하기 위한 것이 적절할 수도 있다.

이진 트리를 구성하고 이용하는 예제로서 산술식을 처리하는 이전 장의 예제를 계속해 보자. 그림 4.4에 제시된 것과 같이 산술식과 트리들 사이에 근본적인 대응이 있다.

인수에 대한 수들보다는 한 개 문자 인식자(identifier)를 이용케 된다. 이같은 것에 대한 이유는 아래에서 분명하게 제시가 된다. 식에 대한 파서 트리(parse tree)는 간단한 재귀 규칙에 의해서 정의가 된다. 즉, “근에 연산자를 놓고, 왼쪽에는 첫번째 오퍼랜드에 대응되는 식

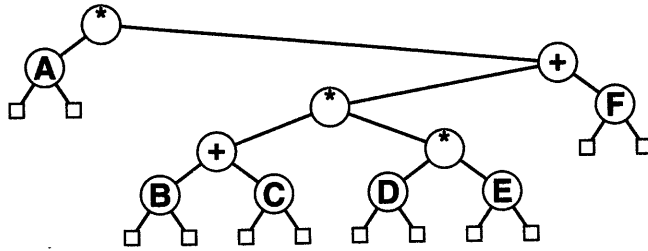


그림 4.4 $(B+C)*(D+E))+F$ 에 대한 파서 트리

에 대한 트리를 오른쪽에는 두 번째 오퍼랜드에 대응되는 식에 대한 트리를 놓는다.“ 그림 4.4는 $A B C + D E * * F + *$ (후위 표기식)에 대한 파서 트리이다. 중위 표기식과 후위 표기식과 같이, 산술식을 표현하는 데는 두 가지 방법이 있고 파서 트리는 세 번째이다.

연산자는 정확히 두 개의 오퍼랜드를 가지므로 해서, 이진 트리는 이같은 종류의 식에 대해 적절하다. 더 복잡한 식은 트리의 다른 형태를 요구케 된다. 이 문제를 21장에서 더 자세히 보기로 하자. 여기에서의 목적은 단순히 산술식의 트리 표현을 구성하는 것이다.

다음 코드는 후위 표기 입력 표현에서 산술식에 대한 파서 트리를 설정하는 것이다. 이것은 스택을 이용해서 후위 표기식을 계산하는 이전 장에 주어진 프로그램의 간단한 변형이다. 스택에 중간 결과 값을 누적시키 보다는 오히려, 다음 구현에서와 같이 식 트리를 보관케 된다.

```
struct node
{ char info; struct node *l, *r; };
struct node *x, *z;
char c; Stack stack(50);
z = new node; z->l = z; z->r = z;
while (cin.get(c))
{
    while ( c == ' ') cin.get(c);
    x = new node;
    x->info =c; x->l =z; x->r =z;
    if ( c == '+' || c == '*')
        { x->r = stack.pop(); x->l = stack.pop(); }
    stack.push(x);
}
```

3장의 스택 삽입 형 typedef를 적절히 이용하므로 해서, 포인터들은 정수라기 보다는 오히려 스택에 놓여지게 된다. 모든 노드는 한 문자를 지니고 다른 노드들에 두 개의 연결을 가진다. 새로운 비어있지 않은 문자를 만날 때마다, 한 노드는 표준 C++ new 기억 장소 할당 구조를 이용해서 그것에 대해 설정케 된다. 만약 그것이 연산자이면, 그것의 오퍼랜드에 대한 부분 트리는 후위 표기 계산에서와 같이 스택의 제일 위에 존재케 된다. 만약 그것이 오퍼랜드인 경우, 그때 그것의 연결들은 비어 있다. 비어 있는 연결들을 이용하기 보다는 오히려, 연결이 자기 자신을 가르키는 가상 노드 z를 이용한다. 이것은 트리상에 어떤 연산을 더 편리하게 만든다.(14장 참조) 그림 4.5는 그림 4.4의 트리 구성에 있어서 중간 단계를 나타낸다.

이같은 오히려 간단한 프로그램은 지수와 같은 한 개의 인수인 연산자를 포함한 더 복잡한 식을 처리하도록 변경을 할 수가 있다. 그러나 메카니즘은 매우 일반적이다. 즉, 예를 들면, C++ 프로그램을 분석하고 컴파일 하기 위해서 같은 메카니즘이 이용된다. 파서 트리가 한번 생성되면, 식 계산이나 식을 계산하기 위한 컴퓨터 프로그램을 생성하는 것과 같은 여러 가지 목적으로 이용된다. 21장에서 파서 트리를 생성하는 일반적인 절차를 논하게 될 것이다. 아래에서 트리 그 자체가 식을 어떻게 이용되는지 나타내고 있다. 그러나 이장의 목적은 트리 구성에 대한 메카니즘에 더 관심을 둔다.

연결 리스트로서, 이진 자료 구조를 구현하기 위한 포인터와 레코드보다는 오히려 병렬 배열을 이용한 대안의 방법이 있다. 이전처럼, 이같은 것은 노드들의 수를 이미 알고 있을 때 특히 유용하다. 또한 이전처럼, 노드들이 어떤 다른 목적에 대해 배열을 차지할 필요가 있는 특별한 경우에는 이같은 대안을 부르게 된다.

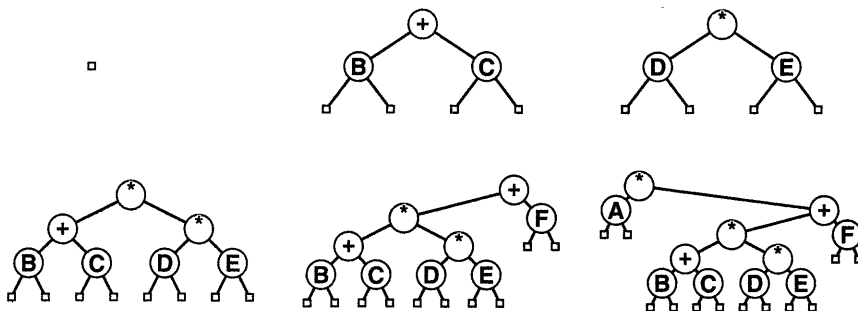


그림 4.5 $A B C + D E * * F + *$ 에 대한 파서 트리 생성

위에서 이용된 이진 트리에 대한 두 가지 연결 표현은 트리를 따라 밑으로 내려온 것(down)은 허용을 하지만 트리의 위로(up) 올라가는 방법은 제공하지는 않는다. 이 상황은 이중 연결 리스트 대 단일 연결 리스트와 비슷하다. 즉, 이동을 더 자유롭게 하기 위해서 각 노드에 다른 연결을 첨가할 수가 있으나 더 복잡한 구현으로 인해 비용이 상승된다. 여러 가지 다른 옵션들은 트리 주위로 이동함을 용이하게 하기 위해서 고급 자료구조에서 이용이 가능하나, 본 교재의 알고리즘에서는 일반적으로 두 개 연결 표현으로도 충분하다.

위의 프로그램에서 외부 노드 대신에 “가상” 노드를 이용한다. 연결 리스트로서, 이것은 대부분의 상황에서 편리하다고 판명이 되었으나 항상 적절한 것은 아니다. 그리고 공통적으로 이용하는 두 가지 다른 해결책이 있다. 그 하나의 옵션은 외부 노드 즉, 연결이 없는 한 노드에 대한 노드의 다른 형을 이용하는 것이다. 또 다른 옵션은 어떤 방법으로(트리에서 그것들과 다른 연결을 구별하기 위해서) 연결들을 표시케 하는 것이다. 그때 그것들은 트리의 어느 곳인가를 가르키게 하면 된다. 즉, 한 방법으로 아래에 제시가 되어져 있다. 이 문제에 대해서 14장과 17장에서 다시 보도록 하자.

포리스트의 표현(Representing Forests)

이진 트리는 각 내부 노드 아래에 두 개의 연결을 지닌다. 그래서 그것들에 의해 위에서 이용된 표현은 즉각적인 것이다. 그러나 각 노드가 아래 노드들에 연결의 수를 요구함에 있어서, 일반적인 트리나 포리스트들에 대해서 우리는 무엇을 할 수가 있는가? 이같은 문제를 해결하기 위한 것에는 상대적으로 간단한 두 가지 방법이 있다고 판명되었다.

첫째, 많은 응용에서 트리의 밑으로 가지 말고 단지 위로 올라가도록 하는 것이다. 그런 경우에 각 노드는, 자신의 부모에 대해 단지 한 개의 연결을 가지면 된다. 그림 4.6은 그림 4.1의 트리에 대한 표현을 나타낸 것이다. 즉, 배열 a 는 각 레코드와 관련된 정보를 포함하고 배열 dad 는 부모 연결들을 포함한다. 이와 같이 $a[i]$ 의 부모와 관련된 정보는 $a[dad[i]]$

k	1	2	3	4	5	6	7	8	9	10	11
$a[k]$	(A)	(S)	(A)	(M)	(P)	(L)	(E)	(T)	(R)	(E)	(E)
$dad[k]$	3	3	10	8	8	8	8	9	10	10	10

그림 4.6 트리의 부모 연결 표현

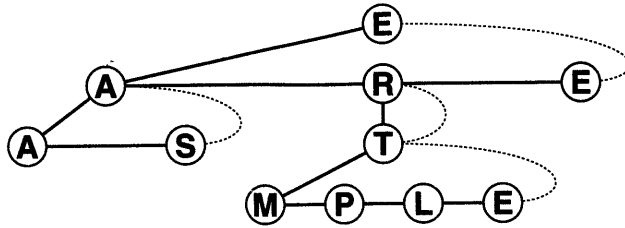


그림 4.7 트리의 가장 왼쪽 자식, 오른쪽 형제의 표현

내에 있다는 등이다. 편리하게 근은 자신을 가르키도록 만든다. 이것은 트리를 위로 올라가도록 하는 것이 적절할 경우에 채택하도록 하는 오히려 간결한 표현이다. 이같은 표현의 이용에 대한 예를 22장과 30장에서 볼 것이다.

하향식(top-down) 처리로서 포리스트를 표현하기 위해서, 어떤 노드에 대한 특별한 수를 미리 할당시킴이 없이 각 노드의 자식을 처리하는 방법이 필요하다. 그러나 이것은 연결 리스트가 제거되도록 설계된 제약의 형태이다. 명백히 각 노드의 자식에 대한 연결 리스트를 이용케 된다. 그때 각각 노드는 두 개 연결을 포함한다. 하나는 형제와 연결된 연결 리스트에 다른 하나는 자식에 연결된 연결 리스트에 대한 것이다. 그림 4.7은 그림 4.1의 트리에 대한 이같은 표현을 나타낸 것이다. 각 리스트를 종료하기 위해 가상 노드를 이용하기보다는 오히려 마지막 노드를 부모로 되돌아가도록 하는 것이다. 즉, 이것은 트리에서 아래로 내려가는 것 뿐만 아니라 위로 올라가도록 하는 방법을 제공해 준다.(이같은 연결들은 그것들과 “형제” 연결을 구별하도록 표시하는 것이다. 대안으로 부모의 이름을 보관하거나 표시하므로 해서 노드의 자식을 통해서 조사가 되므로해서 조사는 부모를 다시 방문할 때 멈추어지게 된다)

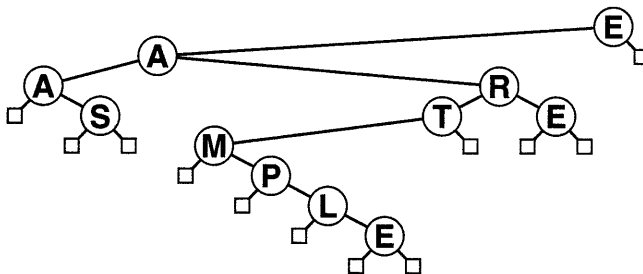


그림 4.8 트리의 이진 트리 표현

그러나 이같은 표현에서 각 노드는 정확히 두 개의 연결을 지닌다.(하나는 오른쪽에 있는 자신의 형제에, 다른 하나는 가장 왼쪽에 있는 자신의 자식에 대한 것이다) 그때 이같은 자료 구조와 이진 트리사이에 차이가 있는 지에 대해 생각을 해야만 한다. 그 답은 그림 4.8에 제시된 것과 같이 없다.(그림 4.1에서 트리의 이진 트리 표현) 즉, 어떤 포리스트에서 각 노드의 왼쪽 연결은 그것의 가장 왼쪽 자식을 가르키도록 하고, 각 노드의 오른쪽 연결은 그것의 형제를 가르키도록 한다.(이같은 사실은 초보자에게는 놀라울 만한 것이다)

이와 같이 알고리즘 설계에서 편리성을 제기할 때마다 이용된다. 밑에서 위로 처리를 할 때, 부모 연결 표현으로 인해 포리스트로 하여금 거의 어떤 다른 종류의 트리보다 처리하기가 더 쉽게 하도록 하고 위에서 밑으로 처리할 때는 본질적으로 이진 트리와 똑같다.

트리 운행(Traversing Trees)

한 번 트리가 구성되면, 알아야 할 필요가 있는 첫 번째 것은 그것을 운행하는 방법이다. 즉, 체계적으로 어떻게 모든 노드를 방문하는가이다. 이같은 연산은 정의에 의하면 선형 리스트에 대해 간단하다. 그러나 트리에 대해서는 처리하는데 여러 가지 다른 방법들이 있다. 방법들은 노드를 방문하는 순서에 따라서 주로 다르다. 볼 수 있듯이, 다른 노드 순서는 다른 응용에 대해서 적절하다.

여기서 이진 트리를 운행하는 곳에 중점을 두기로 하자. 포리스트와 이진 트리사이의 차이가 없으므로 방법들은 포리스트에서도 유용하다. 그러나 방법이 어떻게 직접적으로 포리스트에 적용되는 가는 나중에 언급하게 된다.

고려할 첫번째 방법은 프리 오더(preorder) 운행이다. 예를 들면, 전위 표기로서 그림 4.4의 트리에 의해 표현된 식을 기술하기 위해서 이용될 수가 있다. 이 방법은 간단한 재귀 규칙에 의해서 정의가 된다. 즉, “근을 방문하고, 그 후 왼쪽 부분 트리를 방문하고 그리고 오른쪽 부분 트리를 방문하는 것이다.” 이같은 방법의 가장 간단한 구현인 재귀적 방법은 다음에 제시된 스택을 기반으로 한 구현과 밀접하게 관련이 되어진 것으로 다음 장에서 보기로 하자.

```

traverse(struct node *t)
{
    stack.push(t);
    while(!stack.empty())

```

```

{
    t=stack.pop(); visit(t);
    if(t->r != z) stack.push(t->r);
    if(t->l != z) stack.push(t->l);
}
}

```

규칙에 따라서, 근을 먼저 방문하므로써 “부분 트리를 방문”케 된다. 그때 양쪽 부분 트리를 한 번에 방문할 수가 없기 때문에, 스택에 오른쪽 부분 트리를 보관하고 왼쪽 부분 트리를 방문케 된다. 왼쪽 부분 트리가 방문이 되어졌을 때, 오른쪽 부분 트리는 스택의 상단에 놓여지게 된다. 즉, 그 후 오른쪽 부분 트리가 방문된다. 그림 4.9는 프로그램에 의해서 그림 4.2의 이진 트리에 적용될 때 연산 과정을 나타낸 것이다. 즉, 노드들이 방문되어진 순서는 P M S A A L E R T E E이다.

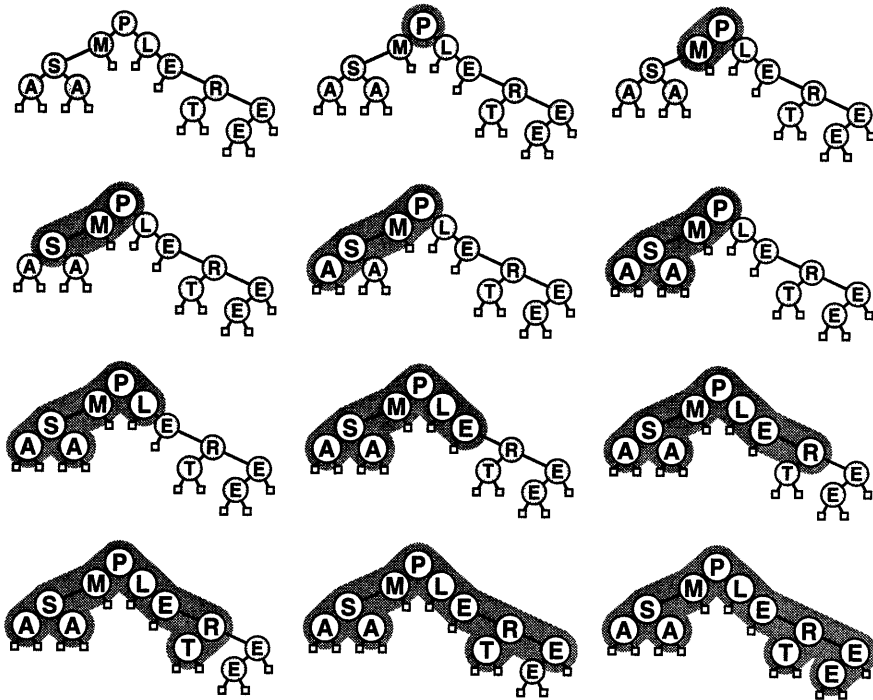


그림 4.9 프리 오더(preorder) 운행

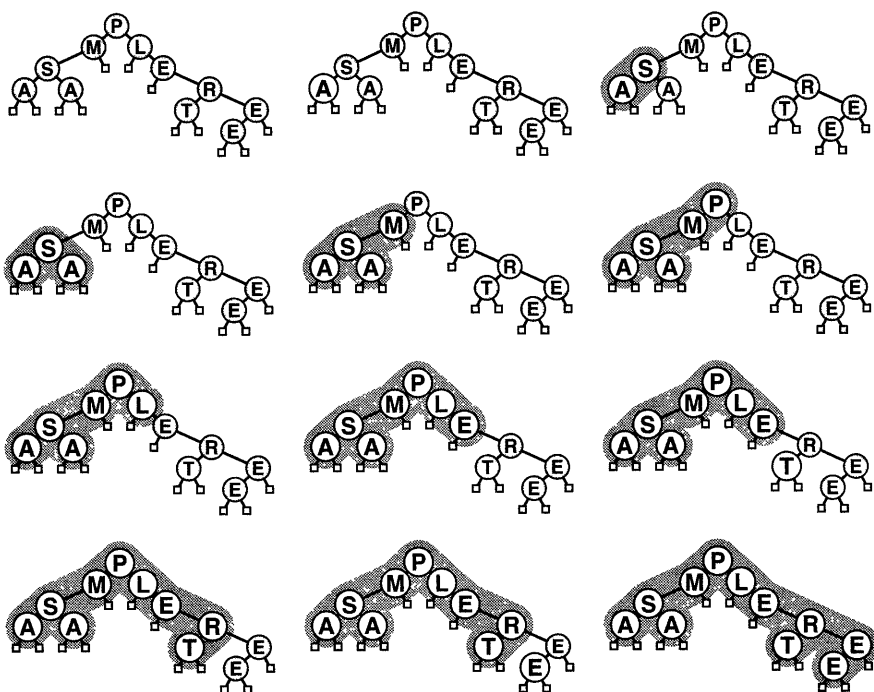


그림 4.10 인 오더(Inorder) 운행

이 프로그램이 프리 오더로서 트리의 노드들을 실제로 방문되는지를 입증하기 위해서, 부분 트리들이 프리 오더로 방문이 되어지고 부분 트리를 방문하기 이전에 스택의 내용이 이후 스택의 내용과 같은지를 귀납적 가설로서 귀납법을 이용케 된다.

둘째, 인 오더(inorder) 운행을 고려하자. 예를 들면, 중위 표기로 파서 트리에 대응되는 산술식을 기술하기 위해 이용된 것이다.(괄호를 오른쪽으로 하도록 하는 몇 가지 특정한 작업과 더불어) 프리 오더와 비슷한 방법으로 인 오더는 “왼쪽 부분 트리를 방문하고 그 후 근을 방문하고 나서 오른쪽 부분 트리를 방문하는” 재귀적 규칙으로서 정의된다. 또한 이것을 가끔은 대칭적 순서(symmetric order)로 부른다. 인 오더에 대한 스택을 기반으로 한 프로그램의 구현은 위의 프로그램과 같다. 즉, 이것은 다음 장의 주된 주제가 되기 때문에 여기서는 생략케 된다. 그림 4.10은 그림 4.2의 트리에 대한 노드들이 어떻게 인 오더로 방문되어지는지를 나타낸 것이다. 즉, 방문된 노드들의 순서는 A S A M P L E T R E E이다. 이같은 운행 방법은 아마도 가장 널리 이용되는 것이다. 예를 들면, 이것은 14장과 15장의 응용에서 중심적인 역할을 한다.

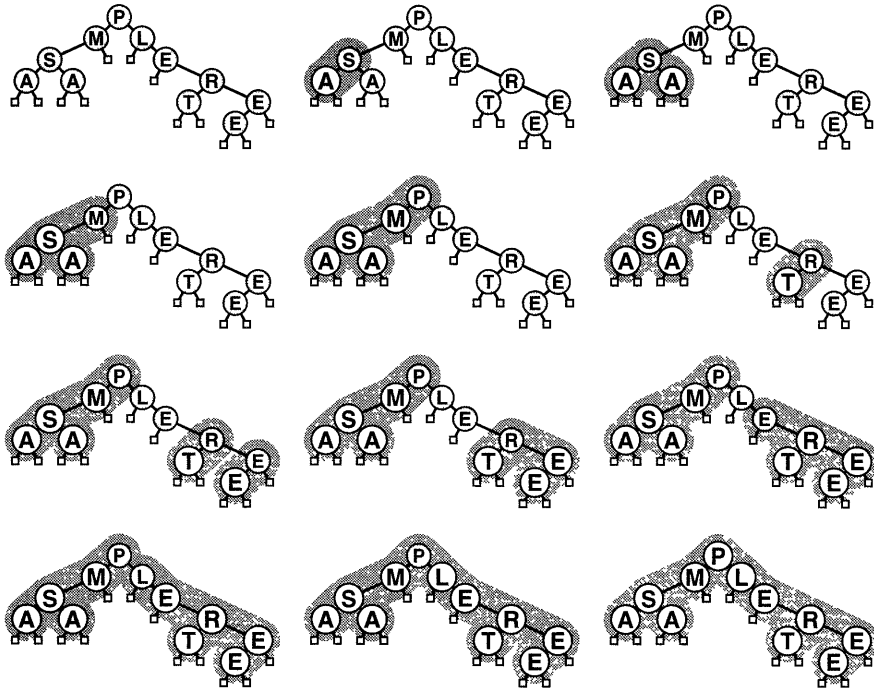


그림 4.11 포스트 오더(Postorder) 운행

포스트 오더(postorder)라 부르는 재귀 운행의 세 번째 형은 물론 재귀 규칙 “왼쪽 부분 트리를 방문한 후 오른쪽 부분 트리 그리고 근을 방문하는” 것으로 정의가 된다. 그림 4.11은 그림 4.2의 트리에 있는 노드들이 포스트 오더로 어떻게 방문되어지는지를 나타낸 것이다. 즉, 방문되어진 순서는 A A S M T E E R E L P이다. 포스트 오더로서 그림 4.4의 식 트리를 방문하면 식은 $A B C + D E * * F + *$ 로 된다. 포스트 오더에 대한 스택을 기반으로 한 프로그램의 구현은 다른 두 방법보다 훨씬 더 복잡하다. 그 이유는 왼쪽 부분 트리가 방문되는 동안에 보관되어야 할 근과 오른쪽 부분 트리에 대해서 배열을 하고 그리고 오른쪽 부분 트리가 방문되어지는 동안에 보관될 근에 대해서 배열을 해야 한다. 이같은 구현의 상세한 내용은 연습 문제로 남겨 둔다.

생각해야 할 4번째 운행 방법은 전혀 재귀적인 것이 아니다. 즉, 페이지에 나타난 대로 노드들을 위에서 밑으로, 왼쪽에서 오른쪽으로 읽어 내려가면서 단순히 방문을 하면 된다. 각 레벨상에 있는 모든 노드들이 한꺼번에 순서적으로 나타나기 때문에 이것을 레벨 오더(level-order) 운행이라고 부른다. 그림 4.12는 그림 4.2의 트리에 노드들이 어떻게 레벨 오더로서 방문이 되는지를 나타낸 것이다.

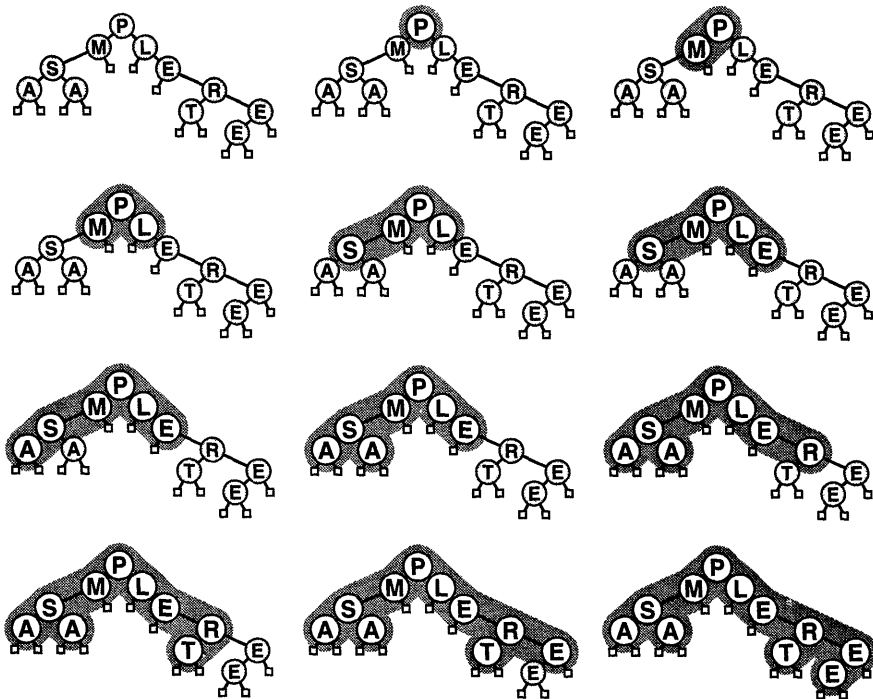


그림 4.12 레벨 오더(level order) 운행

레벨 오더 운행은 스택 대신에 큐로서 위의 프리 오더에 대한 프로그램을 이용해서 만들어진다.

```

traverse(struct node *t)
{
    queue.put(t);
    while ( ! queue.empty() )
    {
        t = queue.get(); visit(t);
        if ( t->l != z ) queue.put(t->l);
        if ( t->r != z ) queue.put(t->r);
    }
}

```

다른 한편, 이 프로그램은 실제적으로는 위의 프로그램과 같다. 주된 차이는 다른 것들이 LIFO 구조를 이용하는데 반해 FIFO구조를 이용한 것이다. 한편, 이같은 프로그램들은 근본적

으로 다른 방법으로 트리를 처리케 된다. 이같은 프로그램들은 스택과 큐사이에 본질적인 차이를 나타내기 때문에 조심스럽게 취급해야 한다. 이 문제는 30장에서 다시 보기로 하자.

프리 오더, 포스트 오더와 레벨 오더는 포리스트에 대해서도 잘 정의가 된다. 정의 일관성을 유지시키기 위해서, 상상적인 근으로된 트리를 포리스트로 생각해 보자. 그때 프리 오더 규칙은 “근을 방문한 후, 부분 트리의 각각을 방문케 된다”. 포스트 오더의 규칙은 “부분 트리의 각각을 방문한 후, 근을 방문케 된다”. 레벨 오더의 규칙은 이진 트리에서 인 오더와 같으나 레벨 순서는 같지 않다. 스택과 큐를 이용한 직접적인 구현은 이진 트리에 대해서 위에 주어진 프로그램을 일반화시키는데 있어서 간단하다.

연습 문제

1. 그림 4.3의 트리가 프리 오더, 포스트 오더, 인 오더와 레벨 오더로 방문할 때 노드들이 방문되어지는 순서를 나열하라.
2. n 개 노드로된 정(complete) 4-way 트리의 높이는 어떻게 되는가?
3. 식 $(A + B) * C + (D + E)$ 에 대한 파서 트리를 그려라.
4. 그림 4.2의 트리를 이진 트리로서 표현된 포리스트로서 고려하시오. 그같은 표현을 그려라.
5. 그림 4.9에 제시된 프리 오더 운행 동안에 노드가 방문될 때마다 큐의 내용은 무엇인가?
6. 그림 4.9에 제시된 레벨 오더 운행 동안에 노드가 방문될 때마다 큐의 내용은 무엇인가?
7. 프리 오더 운행로서 스택이 레벨 오더 운행에서 큐보다 더 많은 공간을 이용하는 트리의 예제를 주어라.
8. 프리 오더 운행로서 스택이 레벨 오더 운행에서 큐보다 더 적은 공간을 이용하는 트리의 예제를 주어라.
9. 이진 트리의 포스트 오더 운행에 대한 스택을 기반으로한 구현은 무엇인가?
10. 이진 트리로서 표현된 포리스트의 레벨 오더 운행을 구현하는 프로그램을 기술하시오.

5 장

재귀

재귀는 수학과 전산학에서 기본적인 개념이다. 간단한 정의로서 재귀 프로그램은 자기 자신을 부르는 것이다.(그리고 재귀 함수는 자기 자신에 의해서 정의된 것이다) 그러나 재귀 프로그램은 항상 자기 자신을 부를 수 있는 것이 아니고 혹은 결코 멈출 수 없는 것이다.(그리고 재귀 함수는 항상 그 자체에 의해서 정의 될 수가 없고 혹은 정의는 순회적으로 이루어진다) 또 다른 본질적인 성분은 프로그램이 자기 자신을 부르는 것을 멈추도록 할 수 있을 때는 종료 조건(termination condition)이 있어야만 한다. 모든 실제적인 계산은 재귀구조로서 표현을 한다.

본 장의 주된 목적은 실제적인 도구로서 재귀를 조사하는 것이다. 첫째, 간단한 수학적 재발생(recurrence)과 간단한 재귀 프로그램 사이에 관계를 나타내면서 재귀가 실제적인 것이 아니라는 몇 가지 예를 보여줄 것이다. 다음으로, 본 교재의 나중 여러 절에서 근본적인 문제들을 해결하는데 이용이 되는 형의 “분할-정복(divide-and-conquer)” 재귀 프로그램의 모형 예를 보여줄 것이다. 마지막으로, 어떤 재귀 프로그램에서 어떻게 재귀가 제거되는가를 논하고, 비재귀 스택을 기반으로한 알고리즘을 얻기 위해서 간단한 재귀 트리에서 재귀를 제거시키는 상세한 예를 제시할 것이다.

보는 바와 같이, 많은 흥미로운 알고리즘들은 재귀 프로그램으로서 간단히 표현을 하고 많은 알고리즘 설계자들이 방법들을 재귀적으로 표현하게 된다. 그러나 똑같은 흥미로운 알고리즘이 (반드시)비 재귀 구현으로 상세하게 나열되는 것은 숨겨져 있게 된다. 즉, 본 장에서는 이같은 알고리즘을 찾기 위한 기법을 논하고자 한다.

재발생(recurrence)

함수의 재귀 정의들은 수학에서 아주 보편화 된 것이다. 즉, 정수형 인수를 포함한 가장 간단한 형을 재발생 관계(recurrence relations)라고 부른다. 아마도 그런 함수와 가장 친숙한 것은 팩토리얼(factorial) 함수로서 다음 공식으로 정의된다.

$$N! = N (N - 1) !$$

여기서,

$$N \geq 1 \text{ 이고 } 0! = 1 \text{ 이다.}$$

이것은 다음의 간단한 재귀 알고리즘으로 직접 대응된다.

```
int factorial(int N)
{
    if ( N == 0 ) return 1;
    return N * factorial(N-1);
}
```

한편으로, 이 프로그램은 재귀 프로그램의 기본 특징을 제시한 것이다. 즉, 자신을 부르고 (인수의 적은 값으로), 결과를 직접 계산할 때에 종료 조건을 지닌다. 다른 한편으로는 이 프로그램이 미화된 for 반복이상 아무것도 없다는 사실을 감추는 것은 없다. 그래서 재귀의 힘을 나타내는 예제로서는 힘들다. 또한 이것은 방정식이 아니고 프로그램임을 기억하는 것이 중요하다. 예를 들면, 위의 방정식이나 프로그램은 음수 N 에 대해서 “처리”하는 것이 아니나 이같은 통찰의 음수의 효과는 아마도 방정식보다 프로그램에서 더 현저한 것이다. 부름 factorial(-1)은 무한히 반복되는 재귀 반복이 된다. 즉, 더 복잡한 재귀 프로그램에서 더 미묘한 형태로서 나타나는 것이 사실 공통적으로 발생하는 오류이다.

두 번째로 잘 알려진 재발생 관계는 피보나치 수열(Fibonacci numbers)이다.

$$F_N = F_{N-1} + F_{N-2}$$

여기서,

$$N \geq 2 \text{ 이고 } F_0 = F_1 = 1 \text{ 이다.}$$

이같은 정의는 다음 순서를 나타낸다.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610,

다시 재발생은 다음의 간단한 재귀 프로그램에 대응된다.

```
int fibonacci(int N)
{
    if ( N <= 1 ) return 1;
    return fibonacci(N-1) + fibonacci(N-2);
}
```

이것은 재귀의 능력면에서 납득이 덜된 예이다. 즉, 재귀은 맹목적으로 이용되지 않고 비효율성으로서 납득이 되는 예제이다. 여기서 문제는 재귀 부름이 F_{N-1} 을 계산하기 위해서, F_{N-2} (그리고 F_{N-3})를 이용하였을 때 F_{N-1} 과 F_{N-2} 는 독립적으로 계산을 나타낸다. F_N 를 계산하기 위해서, 요구된 위의 프로시저 fibonacci에 정확히 부른 수를 이해하는 것은 실제로는 쉽다. 즉, 단지 한 번 부름이 필요할 때, $N = 0$ 이거나 $N = 1$ 이 아니더라도 F_N 를 계산하는데 필요한 부름의 수는 F_{N-1} 을 계산하는데 필요한 부름의 수와 F_{N-2} 를 계산하는데 필요로 하는 부름의 수를 더한 것이다. 이것은 피보나찌 수열을 정확히 정의하는 재발생 관계에 적합하다. 즉, F_N 를 계산하기 위해 fibonacci상에 부름의 수는 정확히 F_N 이다. $\phi = 1.61803...$ 는 “귀중한 비율(golden ratio)”인 F_N 은 약 ϕ^N 이라고 잘 알려져 있다. 위의 프로그램은 피보나찌 수열을 계산하기 위해서 지수 시간 알고리즘이다.

대조적으로 다음과 같이 선형 시간 내에서 F_N 를 계산하는 것은 매우 쉽다.

```
const int max=25;
int fibonacci(int N)
{
    int i, F[max];
    F[0] = 1; F[1] = 1;
    for (i = 2; i <= max; i++)
        F[i] = F[i-1] + F[i-2];
    return F[N];
}
```

이 프로그램은 배열 크기 max를 이용하여 처음에 max 피보나찌 수열을 계산하는 것이다. (수는 지수로 증가하므로 해서, max는 적게 된다)

사실, 이전 결과를 저장시키는 배열을 이용한 이같은 방법은 재발생 관계를 계산하는 선택 방법이다. 그 이유는 오히려 복잡한 방정식으로 하여금 단일적이고 효율적인 방법으로 처리 하도록 허용한 것이기 때문이다. 재발생 관계는 재귀 프로그램의 활용도 특징을 결정하고자 할 때 가끔 발생되고 본 교재에서 여러 가지 예제를 보게 될 것이다. 예를 들면, 9장에서는 다음 방정식을 볼 수가 있다.

$$C_N = N - 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}), \quad \text{for } N \geq 1 \text{ with } C_0 = 1.$$

C_N 의 값은 위의 프로그램에서 처럼 배열을 이용해서 오히려 쉽게 계산된다. 9장에서 이같은 공식이 수학적으로 어떻게 처리가 되는지 논하고 알고리즘 분석에서 가끔 발생하는 여러 가지 다른 재발생은 6장에서 논하게 된다.

이와 같이 재귀 프로그램과 재귀적으로 정의된 함수사이의 관계는 실제적인 것보다 훨씬 더 철학적인 것이다. 엄격히 말하면, 위에서 지적한 문제는 재귀 그 자체의 개념과 관계된 것이 아니고 구현과 관계가 된 것이다. 즉, (매우 영리한) 컴파일러는 팩토리얼 함수가 실제상에 반복으로서 구현되고 피보나찌 함수는 배열에서 미리 계산된 값들 모두를 저장 시키므로 써 더 잘 처리를 함을 보게 된다. 아래에서 재귀 프로그램을 구현하는 메카니즘에 대해 더 상세히 보게 될 것이다.

분할-정복(Divide-and-Conquer)

본 교재에서 고려된 대부분의 재귀 프로그램은 두 개의 재귀 부름을 이용하며 각 재귀 부름은 입력의 약 반을 처리한다. 이것을 알고리즘 설계에서는 소위 “분할-정복”이라 한다. 분할-정복 프로그램들은 위의 팩토리얼 프로그램에서 두 개의 재귀 부름을 지니고 있기 때문에 반복들을 감소시킬 수가 없다. 즉, 입력은 겹침이 없이 나누어지기 때문에 위의 피보나찌 수열에 대한 프로그램에서와 같이 과도하게 다시 계산할 필요가 없다.

예제로서, 자에 인치별로 표시하는 것을 보기로 하자. 그림 5.1에 제시된 것과 같이 1/2” 점에 표시를 하고, 더 적은 1/4” 점에 표시를 하고, 더 적은 1/8” 간격에 표시를 하는 등의 순서이다. 이같은 일을 수행하는데는 많은 방법들이 있고 이것은 간단한 분할-정복 계산의 모형이다.



그림 5.1 자

만약 요구된 해결책이 $1/2^n$ "이면, 0과 2^n 사이에 모든 점에서 표시를 하기 위해서 다시 측정을 해야만 한다. 위치 x 에서 높이를 h 단위로 표시하기 위해서 임의적으로 프로시저 $\text{mark}(x, h)$ 를 지닌다고 가정해 보자. 중간점 표시는 n 단위의 높이이고 왼쪽과 오른쪽 중간의 표시는 $n - 1$ 단위의 높이인 것등이다. 다음의 "분할-정복" 재귀 프로그램은 이같은 목적을 이루기 위한 간단한 방법이다.

```
rule(int l, int r, int h)
{
    int m = (l+r) / 2;
    if ( h > 0 )
    {
        mark(m, h);
        rule(l, m, h-1);
        rule(m, r, h-1);
    }
}
```

예를 들면, 부름 $\text{call}(0, 64, 6)$ 은 적절한 척도를 지니것으로 그림 5.1과 같다. 여기에 숨겨진 개념은 다음과 같다. 즉, 간격들 사이에 표시를 하고 먼저 중앙에 있는 긴 표시를 만드는 것이다. 이것은 간격을 두 개 똑같은 크기로 나누는 것이다. 같은 절차를 이용해서 각각의 반으로된 부분에 (더 적은)표시를 하면 된다.

재귀 프로그램의 종료 조건에 특별한 주의를 나타내는 것은 신중해야 한다. 그렇지 않으면, 종료가 안된다. 위의 프로그램에서, rule 은 생성된 표시의 길이가 0일 때 종료된다.(자기 자신을 부르지는 않았다) 그림 5.2는 프로시저 부름의 리스트를 주면서 상세하게 처리되는 과정을 나타낸 것이고, 부름 $\text{rule}(0, 8, 3)$ 의 결과를 표시한 것이다. 중간을 표시하고 왼쪽 반에 대해 rule 을 부르고 그후 왼쪽 반에 대해 같은 처리를 수행하는 등이 길이의 표시가 0일 때까지 수행을 한다. 결국 rule 로 되돌아가서 같은 방법으로 오른쪽 반을 수행한다.

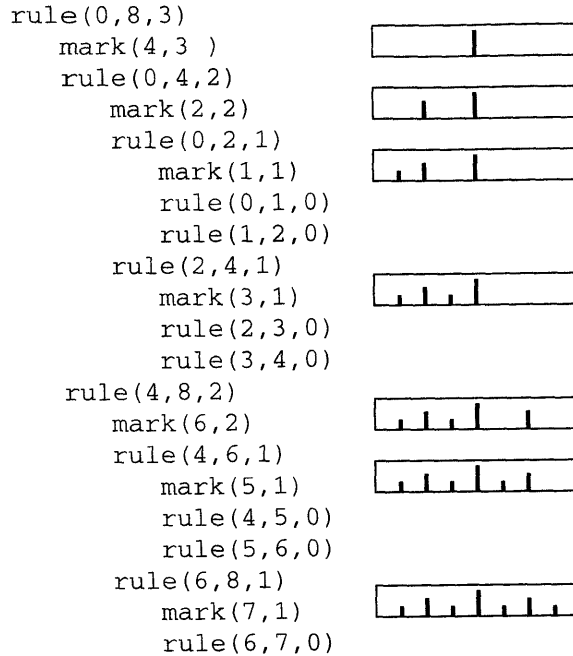


그림 5.2 자 그리기

이같은 문제에 대해, 표시가 그려진 순서는 실제로 의미가 없다. 두 개 재귀 부름 사이에 mark 부름을 놓는 것이다. 어느 경우이든지 예제에서 점들은 그림 5.3에 제시된 것과 같이 왼쪽에서 오른쪽의 순서로 단순히 그려지게 된다.

이같은 두 개 프로시저에 의해서 그려진 표시의 집합은 같으나 순서는 아주 다르다. 이같은 차이는 그림 5.4에 제시된 트리 그림으로서 설명된다. 이같은 그림은 부름에서 이용된 파라미터들로 표시된 rule에 각 부름에 대한 한 노드를 지닌다. 각 노드의 자식은 그같이 야기된 것에 대해 그들의 파라미터를 따라서 rule에 대한(재귀적) 부름에 대응된다. 이같은 트리는 어떤 프로시저의 집합에 대한 동적인 특징들을 제시하도록 그려진 것이다. 지금, 그림 5.2는 이같은 트리를 프리 오더로 운행하게 된다.(여기서 한 노드를 “방문”함은 지시된 부름에 mark가 되도록 하는 것이다) 즉, 그림 5.3은 인 오더로 트리를 운행하는 것에 대응된다.

일반적으로 분할-정복 알고리즘은 입력을 두 개 부분으로 분리시키는 작업 또는 입력의 두 개 별개의 “해결된” 부분들을 처리하여 결과를 병합시키는 것 또는 입력의 반이 처리되어진 후에 어떤 것을 도울 수 있도록 하는 작업이 수반된다. 즉, 두 개의 재귀적 부름 이전 혹은 이후 또는 그 사이의 코드를 나타낸 것이다. 9장, 12장, 27장, 28장과 41장에서 그런 알고

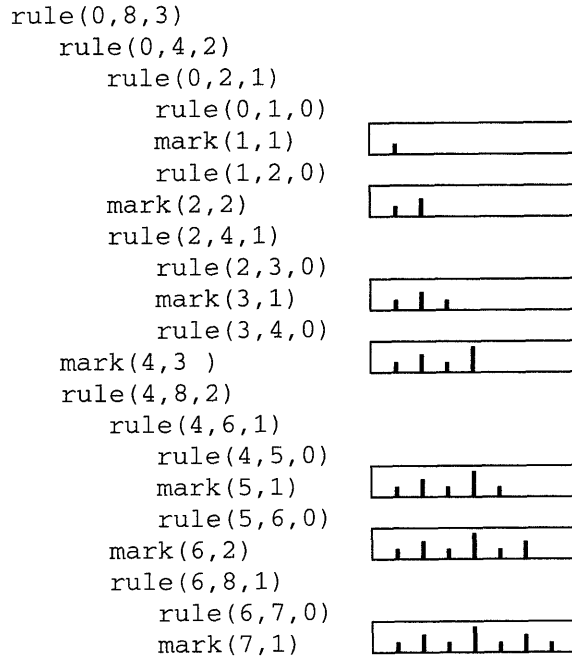


그림 5.3 자 그리기(인 오더 버전)

리즘의 많은 예제를 보게 될 것이다. 또한 분할-정복 정체를 완전히 따르도록 하는 것이 가능하지 않은 그런 알고리즘을 보게 될 것이다. 아마도 입력은 똑같은 크기로 또는 두 개 부분 이상으로 구분이 되지 않거나 혹은 부분들 사이에 몇 가지 겹치는 부분이 존재케 된다.

또한 이같은 작업에 대해 비 재귀적 알고리즘을 생성하는 것이 쉽다. 가장 간단한 방법은 그림 5.3에서 처럼 표시를 순서대로 단순히 그리기 위한 것이나 직접적인 반복문인 `for (i = 1; i < N; i++) mark (i, height(i));`로 이루어진다. 이것을 위해 필요한 함수 `height(i)`는 계산하기 어려운 것이 아님이 판명되었다. 즉, 그것은 `i`의 이진 표현에서 마지

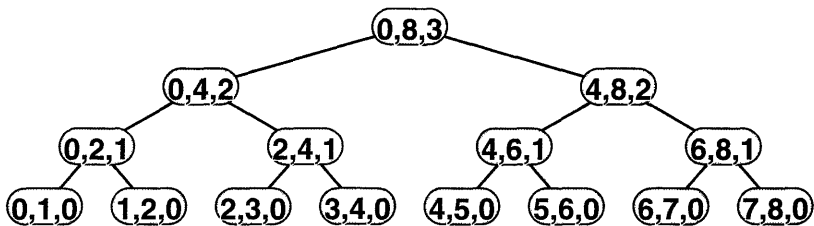


그림 5.4 자를 그리기 위한 재귀적 부름 트리

막에 위치한 0 비트들의 수가 된다. 이 함수를 C++로 구현하는 것은 연습 문제로 남겨 두자. 이같은 방법을 또 다른 문제로서 아래에 자세히 기술한 “재귀 문제를 제거”를 통해서 재귀적 버전으로 부터 직접 나타내는 것은 실제로 가능하다.

어떤 재귀적 구현에 대응이 되지 않은 또 다른 비 재귀적 알고리즘은 다음 프로그램에서와 같이 먼저 가장 짧은 곳에 표시를 하고 그 다음은 다음으로 짧은 곳에 표시하는 순으로 그리게 된다.

```
rule (int l, int r, int h)
{
    int i, j, t;
    for ( i =1, j = 1; i <= h; i++, j += j)
        for ( t = 0; t <= ( l + r ) / j; t++)
            mark(l+j+t*(j+j), i);
}
```

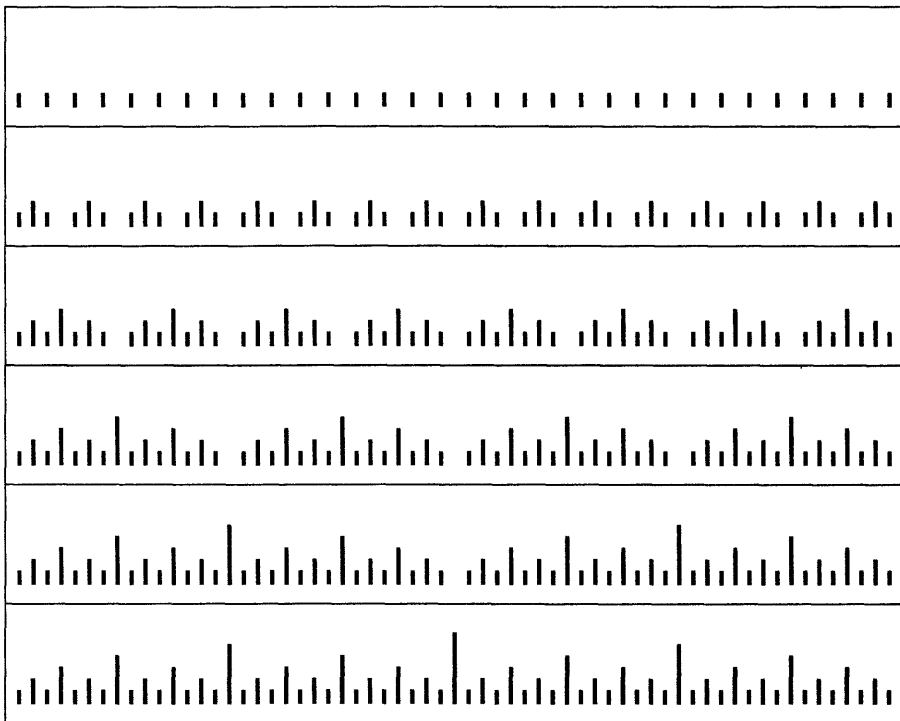


그림 5.5 자를 비 재귀적으로 그리는 것

그림 5.5는 이같은 프로그램이 어떻게 표시를 그리는 가를 나타낸 것이다. 이같은 과정은 레벨 오더로서(밑에서 위로) 그림 5.4의 트리를 운행하는 것이나 재귀적인 것은 아니다.

이것은 먼저 사소한 부분 프로그램들을 해결하고 그후에 전체 문제가 해결 될 때까지 다소 더 큰 부분 프로그램을 해결하기 위한 이같은 해를 결합시키므로써 문제를 해결하는 알고리즘 설계의 일반적인 방법에 대응된다. 이같은 방법을 “결합과 정복(combine and conquer)”이라고 부른다. 어떤 재귀 프로그램과 똑같은 비 재귀적 구현을 얻는 것이 항상 가능한 반면에 이 방법으로 계산을 재 배열하는 것은 항상 가능하지 않는다. 즉, 많은 재귀 프로그램들은 특별한 순서로서 해결된다. 이것은 분할-정복의 하향식(top-down) 방법과 대조를 이룬 상향식(bottom-up) 방법이다. 이것에 대해 여러 가지 예제를 보게 될 것이다. 즉, 가장 중요한 것이 12장이다. 방법의 일반화는 42장에서 논의케 된다.

자를 그리는 예제를 자세히 보기로 하자. 매우 간단한 것에서 매우 복잡한 것에 이르는 경계를 가로 지를수 있는 것을 이야기 한다는 것은 쉽지 않으므로 재귀적으로 간단한 예제를 상세하게 보는 구실이 된다. 그림 5.6은 간단한 재귀 기술이 오히려 복잡한 계산으로 되어진 계산으로 어떻게 이끌어지는지를 제시한 2차원 패턴을 나타낸 것이다. 오른쪽 패턴은 한 개로만 나타난 경우에 훨씬 더 신비로운 것인 반면에 왼쪽의 패턴은 재귀적 구조로 쉽게 인식케 된다. 왼쪽의 패턴을 생성하는 프로그램은 실제로는 rule에 대해 일반화만 시킨 것이다.

```
star(int x, int y, int r)
{
    if ( r > 0 )
    {
        star(x-r, y+r, r/2);
        star(x+r, y+r, r/2);
        star(x-r, y-r, r/2);
        star(x+r, y-r, r/2);
        box(x, y, r);
    }
}
```

이용된 그림 원칙은 (x, y) 를 중심으로한 크기 $2r$ 의 사각형으로 그려진 간단한 프로그램이다. 이와 같이 그림 5.6의 왼쪽에 있는 패턴은 재귀 프로그램으로 일반화시키기가 간단한 것이다. 즉, 오른쪽에 제시된 패턴의 외각선을 그리는 재귀적 방법을 찾으려고 할 것이다. 왼

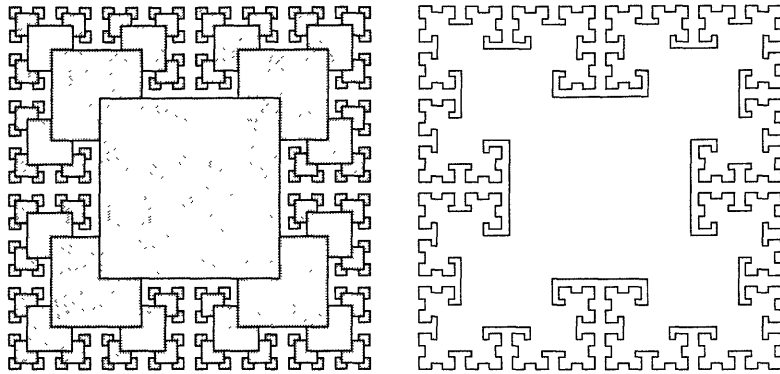


그림 5.6 박스들로(왼쪽) 그리고 외곽선만(오른쪽)으로 그려진 프렉탈(fractal) 별

쪽의 패턴 또한 그림 5.5에서 표시된 것과 같이 상향식으로 일반화시키기가 쉽다. 즉, 제일 먼저 가장 작은 사각형을, 그 후 두 번째 작은 것등의 순이다. 또한 외곽선을 그리는 비 재귀적 방법을 찾으려고 할 것이다.

그림 5.6과 같이 재귀적으로 정의된 기하학적 패턴을 프렉탈(fractal)이라고 부른다. 만약 더 복잡한 그림 원칙이 이용되고 더 복잡한 재귀가 야기 되면(특히 실수와 복소수 평면에서 재귀적으로 정의된 함수를 포함), 현저한 다양성과 복잡성의 패턴이 개발된다.

재귀적으로 트리를 운행

4장에 지시 한대로, 트리의 노드들을 운행하는 가장 간단한 방법은 재귀적으로 구현하는 것이다. 예를 들면, 다음 프로그램은 인 오더로서 이진 트리의 노드들을 방문케 된다.

```

traverse(struct node *t)
{
    if ( t != z )
    {
        traverse(t->l);
        visit(t);
        traverse(t->r);
    }
}

```

구현은 인 오더의 정의와 같다. 즉, “만약 빈 트리가 아니면, 먼저 왼쪽 부분 트리를 방문하고 그후 근을 그리고 오른쪽 부분 트리를 방문하는 것이다.” 명백히, 프리 오더는 두 개 재귀 부름 이전에 방문에 대한 부름을 놓으므로 써 구현되고, 포스트 오더는 두 개 재귀적 부름 뒤에 방문에 대한 부름을 놓으므로 써 구현된다.

트리 운행의 이같은 재귀적 구현은 트리들이 재귀적으로 정의된 구조들이고, 프리 오더, 포스트 오더와 인 오더는 재귀적으로 정의된 처리들인 두 가지 이유 때문에 스택을 기반으로 한 구현보다 더 자연스럽다. 대조적으로, 레벨 오더 운행에 대한 재귀적 프로시저를 구현하는데는 편리한 방법이 없음을 알아야 한다. 트리보다 훨씬 더 복잡한 그래프의 운행 알고리즘을 고려할 때 29장과 30장에서 이 문제를 다시 다루게 된다.

위의 재귀 프로그램에 대한 간단한 변형과 visit의 적절한 구현은 간단한 방법으로 트리의 여러 가지 성질들을 계산하는 프로그램으로 나타낸다. 예를 들면, 다음 프로그램은 본 교재에서 이진 트리 노드들을 놓기 위한 좌표들을 어떻게 계산하는 것인가를 나타낸 것이다. 가령 노드들에 대한 레코드가 페이지에 있는 노드의 x 와 y 좌표들에 대해 두 개의 정수형 필드를 포함한다고 하자.(척도와 번역의 상세함을 피하기 위해서, 이것들은 상대적인 좌표들로 가정된다. 즉, 만약 트리가 N 개의 노드를 지니고 높이가 h 이면, x 좌표는 1에서 N 으로 왼쪽에서 오른쪽으로 처리를 하고 y 좌표는 1에서 h 로 하향식 처리를 한다) 다음 프로그램은 각 노드에 대한 적절한 값으로서 이같은 필드들을 채운다.

```
visit(struct node *t)
{ t->x = ++x; t->y = y; }
traverse(struct node *t)
{
    y++;
    if(t != z)
    {
        traverse(t->l);
        visit(t);
        traverse(t->r)
    }
    y--;
}
```


프로그램은 두 개의 광의 변수 x 와 y 를 이용하고 둘 다는 초기에 0으로 되어졌다고 가정하자. 변수 x 는 인 오더로서 방문되어진 노드의 수를 파악하고, 변수 y 는 트리의 높이를 유지케 된다. `traverse`가 트리에서 밑으로 갈때마다 1씩 증가가 되고, 트리에서 위로 올라 갈때마다 1씩 감소케 된다.

비슷한 방법으로 트리의 경로 길이를 계산하고, 트리를 그리기 위해서 다른 방법으로 구현하거나 혹은 식 트리로써 표현된 식을 계산하기 위한 것등의 재귀 프로그램들을 구현케 된다.

재귀를 제거

트리 운행에 대한 위의(재귀적)것에 대한 구현과 4장에서(비재귀적) 제시된 것의 구현 사이에 관계는 무엇인가? 확실히 이같은 두 개 프로그램은 어떤 트리가 주어진 경우, 그것들은 `visit`에 대한 똑같은 연속된 부름과 밀접한 관련이 있다. 본 절에서는, 비재귀적 구현을 얻도록 하기 위해서 위에 주어진 프리 오더 운행 프로그램에서 재귀를 “기계적으로” 제거시키므로써 생성된다.

이것은 재귀 프로그램을 기계어로 번역을 하는 작업이 주어질 때 컴퓨터가 직면한 일과 같다. 여기서의 목적은 번역 기법에 있는 것이 아니고(컴파일러에 의해서 직면한 프로그램들에 몇 가지 통찰을 얻으므로써) 오히려 알고리즘의 재귀와 비재귀 구현 사이의 관계를 공부하는 것이다. 이같은 주제는 본 교재를 통해서 다시 발생된다.

우선 먼저, 위에 제시된 것과 같이 프리 오더 운행의 재귀적 구현으로 시작해 보자.

```
traverse(struct node *t)
{
    if(t != z)
    {
        visit(t);
        traverse(t->l);
        traverse(t->r);
    }
}
```

먼저, 두 번째 재귀 부름은 그것을 따르는 코드가 없기 때문에 쉽게 제거가 된다. 두 번째 부름이 실행 될 때, `traverse`가 불리진다.(인수 `t->r`과 함께) 그후, 부름이 완료된 후,

traverse 역시 끝이 난다. 그러나 같은 연속된 사건은 다음과 같이 재귀 부름보다는 오히려 goto로서 구현을 하게 된다.

```
traverse(struct node *t)
{
    l: if (t == z) goto x;
        visit(t);
        traverse(t->l);
        t = t->r;
        goto l;
    x: ;
}
```

이것은 많은 컴파일러에서 구현되는 재귀 끝의 제거(end-recursion removal)라고 부른다. 재귀 프로그램들은 위의 factorial과 fibonacci에서 제기된 문제들과 같이 가공적이고 비효율적인 것이 발생이 되기 때문에 이같은 능력 없이도 시스템상에 생존 능력이 덜하다. 9장에서는 중요하고 실제적인 예제를 보게 될 것이다.

다른 재귀 부름을 제거하는 것은 더 많은 작업을 요구한다. 일반적으로, 대부분의 컴파일러는 어떤 프로시저에 대한 같은 연속된 행위를 통해 처리되는 코드를 나타낸다. 즉, “스택에 국지 변수들과 다음 명령의 번지를 삽입하고, 프로시저에 파라미터의 값을 세트시키고, 프로시저 제일 처음으로 가게 한다.” 프로시저가 완료된 후, “스택에서 국지 변수의 값과 되돌려지는 번지를 삭제하고, 변수를 다시 세트시키고 그리고 되돌아가야 할 번지로 가게 한다.” 물론, 실제 컴파일러에서 직면한 일반적인 상황 보다 더 복잡할 수도 있다. 이같은 상황에서, 다음과 같이 위의 프로그램에서 두 번째 재귀 부름을 제거시킬 수가 있다.

```
traverse(struct node *t)
{
    l: if ( t == z ) goto s;
        visit(t);
        stack.push(t); t = t->l; goto l;
    r: t = t->r; goto l;
    s: if (stack.empty() ) goto x;
        t = stack.pop(); goto r;
    x: ;
}
```

국지 변수는 t 한 개 이므로 스택에 삽입하고 goto를 시작점으로 옮긴다. 고정되어진 단지 한 개의 되돌아가는 번지 r 이 있으므로 그것은 스택에 삽입시키지 않는다. 프로시저의 끝에, 스택에서 t 를 세트시키고 되돌아가는 번지 t 로 가게 한다. 스택이 비어 있으면, 첫 번째 부름에서 traverse로 되돌린다.

지금, 재귀가 제거되었으나 오히려 불투명한 프로그램을 나타내는 혼란스러운 goto들로만 구성된다. 그러나 이것들 역시 코드의 훨씬 더 구조화된 부분으로 나타내도록 하기 위해서 “기계적”으로 제거된다. 첫째, 라벨 r 과 두 번째 goto x 사이에 코드의 부분은 goto들로서 둘러싸여져 있다. 이것은 라벨 r 과 관련된 goto를 제거하므로써 간단히 제거된다. 다음으로, 스택에서 삭제될 때, t 를 $t \rightarrow r$ 로 세트시킨다. 즉, 그 값을 스택에 삽입시킨다. 라벨 x 와 첫 번째 got x 사이에 코드는 while 반복으로 이루어진다.

```
traverse(struct node *t)
{
    l: while ( t != z )
        {
            visit(t);
            stack.push(t->r); t = t->l;
        }
    if ( stack.empty() ) goto x;
    t = stack.pop(); goto l;
    x: ;
}
```

goto가 없는 프로그램으로 되도록 특정한 스택 삽입(운행하고자 하는 초기 인수 t 의)을 부가함으로써 while 반복으로 변형시킨 또 다른 반복을 가진다.

```
traverse(struct node *t)
{
    stack.push(t);
    while ( !stack.empty() )
    {
        t = stack.pop();
        while ( !stack.empty() )
        {
```

```

        visit(t);
        stack.push(t->r);
        t = t->l;
    }
}
}

```

이같은 버전은 “표준” 비 재귀 트리 운행 방법이다. 실제로, 이 프로그램의 반복 내에 반복 구조는 간단히 시킬 수가 있다.

```

traverse(struct node *t)
{
    stack.push(t);
    while( !stack.empty() )
    {
        t = stack.pop();
        if ( t != z )
        {
            visit(t);
            stack.push(t->r);
            stack.push(t->l);
        }
    }
}

```

이 프로그램은 엄격히 보면 원래의 재귀 프리 오더 알고리즘과 같으나 두 개 프로그램은 진정으로 아주 다르다. 주된 차이는 이 프로그램은 어떤 프로그램 환경에서도 처리가 되어질 수 있는데 반해 재귀 구현은 명백히 재귀를 유지하는 것을 요구한다. 그런 환경에서 조차, 이 같은 스택을 기반으로 한 방법은 오히려 더 효율적이다.

마지막으로 이 프로그램은 비어 있는 부분 트리 들을 스택에 놓고 재귀 프로시저의 첫 번째 행동으로서 부분 트리가 비어있는지를 테스트하기 위해서 원래 구현에서 판단의 직접적인 결과를 주시해야 한다. 재귀 구현은 $t \rightarrow l$ 과 $t \rightarrow r$ 을 테스트함으로써 비어있지 않은 부분 트리에 대해서만 재귀 부름을 만들 수 있다. 위의 프로그램에서 이같은 변화를 반영시키므로서 4장의 스택을 기반으로 한 프리 오더 운행 알고리즘으로 된다.

```

traverse(struct node *t)
{
    stack.push(t);
    while (!stack.empty())
    {
        t = stack.pop(); visit(t);
        if (t->r != z) stack.push(t->r);
        if (t->l != z) stack.push(t->l);
    }
}

```

어떤 재귀 알고리즘은 재귀를 제거시키기 위해서 위의 것처럼 처리된다. 정말로 이것은 컴파일러의 주된 작업인 것이다. 여기에 기술된 것과 같이 “수작업”으로 재귀 제거는 가끔은 효율적인 비 재귀 구현과 계산의 본질을 더 잘 이해하도록 하게 한다.

전망(Perspective)

그렇게 간단한 논의로서 재귀와 같은 기본적인 주제에 대해서 정당화시키는 것은 확실히 불가능하다. 재귀 프로그램의 대부분 좋은 예제들은 본 교재를 통해서 나타난다. 즉, 분할-정복 알고리즘은 널리 이용되는 문제들에 대해서 고안이 된 것이다. 많은 응용에 대해서, 간단하고 직접적인 재귀 구현 이상을 넘어갈 이유는 없다. 다른 것들에 대해서는 이 장에서 기술된 것과 같이 재귀 제거의 결과를 고려하거나 직접적으로 다른 비 재귀 구현을 이끌어 낸다.

재귀는 초기에 이론적인 연구의 중심으로 한 계산의 본질로 된다. 재귀 함수와 프로그램은 해결 못한 프로그램에서 컴퓨터로 해결할 수 있는 문제들로 분리시키도록 하는 수학적 연구에서 중심적인 역할을 한다.

44장에서는 가능한 많은 해결책이 조사되어야만 하는 어려운 문제를 푸는데 있어서 재귀 프로그램(그리고 다른 기법들)을 이용함을 보게 된다. 재귀 프로그래밍은 가능성의 집합을 통해서 복잡한 검색을 구성하는 아주 효과적인 수단이 된다.

연습 문제

1. 근이 페이지의 중앙에 나타내고 왼쪽 부분 트리의 근은 페이지의 왼쪽 반 중앙에 위치케 하도록 하는 이진 트리를 그리는 재귀 프로그램을 기술하시요.
2. 이진 트리의 외부 경로 길이를 계산하는 재귀 프로그램을 기술하라.
3. 이진 트리로서 표현된 트리의 외부 경로 길이를 계산하는 재귀 프로그램을 기술하라.
4. 교재에 주어진 재귀적인 트리를 그리는 프로시저는 그림 4.2의 이진 트리에 적용될 때 생성되는 좌표는?
5. 비 재귀 구현을 얻기 위해서 교재에서 주어진 fibonacci 프로그램에서 재귀를 기계적으로 제거하라.
6. 비 재귀 구현을 얻기 위해서 재귀 인 오더 트리 운행 알고리즘에서 재귀를 기계적으로 제거하라.
7. 비 재귀 구현을 얻기 위해서 재귀 포스트 오더 트리 운행 알고리즘에서 재귀를 기계적으로 제거하라.
8. 정수 좌표로만 이용해서 점들을 그리므로써 두 점 (x_1, y_1) 과 (x_2, y_2) 를 연결하는 선 세그먼트 근사치를 그리는 재귀적 “분할-정복” 프로그램을 기술하라.(힌트: 먼저 중앙과 가까운 점을 그려라)
9. 요세퍼스(Josephus)문제(3장 참조)를 해결하는 재귀 프로그램을 기술하라.
10. 유클리드(Euclid) 알고리즘(2장 참조)의 재귀 구현을 기술하라.

빈 면

6 장

알고리즘 분석

많은 문제에 대해서 많은 다른 알고리즘들이 사용 가능하다. 가장 좋은 구현의 선택을 어떻게 하는가? 이것은 전산학에서 잘 발전된 분야이다. 근본적인 알고리즘의 활용도를 기술하는 연구 결과를 요청하는 경우가 가끔 있다. 그러나, 알고리즘 비교에서는 정말로 도전적이고 어떤 일반적인 안내가 유용하다.

항상 해결해야 할 문제들은 정상적으로 N 이라 부르는 당연한 “크기”(전형적으로 처리해야 할 양)를 지닌다. N 의 함수로서 이용된 자원들(대부분 걸린 시간의 양) 기술하고자 한다. 프로그램이 “전형적인” 입력 데이터에 걸리는 시간의 양인 평균 경우(average case)와 프로그램이 최악의 가능한 입력 구성상 걸리는 시간의 양인 최악의 경우(worst case)에 관심을 둔다.

본 교재에서 몇몇 알고리즘들은 정확한 수학 공식들이 평균적이고 최악의 경우 처리 시간에 대해 알려진 점에 대해 매우 잘 이해가 된다. 그런 공식들은 근본적인 수학적 양들로서 수행 시간을 찾기 위해 프로그램을 조심스럽게 연구하고 그후 수반되는 양들의 수학적 분석을 통해 만들어진다. 다른 한편, 본 교재에서 다른 알고리즘의 활용도 성질은 전혀 이해가 되지 않는다. 즉, 그것들의 분석은 아마도 미 해결된 수학적 의문으로 이끌어지거나 혹은 아마도 알려진 구현이 상세한 분석에 대해서는 너무 복잡해서 합리적이지 못하거나 혹은(대부분의 경우) 아마도 만나게 될 입력의 형태가 적절히 이루어지지 못하는 것이다. 많은 알고리즘들은 이같은 극단적인 것들 사이의 어딘가에 존재케 된다. 즉, 몇 가지 사실들은 그것의 활용도에 대해 알고 있어야 하나 진정으로 완전히 분석되지는 않는다.

여러 가지 중요한 요인들은 주어진 프로그래머의 영향권 밖에 항상 존재하므로 써 이같은 분석이 된다. 첫째, C++ 프로그램들은 주어진 컴퓨터에 대해서 기계어 코드로 번역되고, 심지어 한 개 C++ 명령문이 실행하는데 얼마나 오래 걸리는가를 이해하는 것이다.(특히 자원들

이 공유되어진 환경하에서, 같은 프로그램이 다양한 활용도 특징을 지닌다 할지라도) 둘째, 많은 프로그램들은 그들의 입력 데이터에 상당히 민감하고, 활용도는 입력에 따라서 상당히 변동이 있다. 평균의 경우는 프로그램이 이용되어진 실제 데이터의 표현이 아닌 수학적 허구이고, 최악의 경우는 실제로 결코 발생이 되지 않는 색다른 구성이다. 세 번째, 관심이 되는 많은 프로그램들은 잘 이해가 되지 않고 특별한 수학적 결과가 이용되지 않는다. 마지막으로 프로그램은 전혀 비교가 되지 않는 경우가 가끔 있다. 즉, 어떤 것은 어떤 특정한 종류의 입력에 훨씬 더 효과가 있게 수행을 하고 다른 것은 다른 환경하에서 효율적으로 수행된다.

그럼에도 불구하고 위의 설명은 특정한 프로그램이 얼마나 오래 걸리는 가를 예측을 하거나 혹은 한 프로그램이 특별한 상황에서 다른 것보다 낮게 수행됨을 아는 것이 가능하다. 알고리즘의 활용도에 관해서 가능한 많은 정보를 찾으려고 하는 것이 바로 알고리즘 분석가의 일이다. 그리고 특정 응용에 대해서 알고리즘을 선택함에 있어서, 그런 정보를 적용하는 것은 바로 프로그래머의 일이다. 본 장에서는 분석가의 이상적인 세계에 오히려 관심을 집중하고 다음에서는 구현의 실제적인 고려 사항을 논하게 된다.

구조(Framework)

알고리즘 분석에서 첫 번째 단계는 알고리즘의 입력으로서, 이용된 데이터를 특성화시키고 어떤 형태의 분석이 적절한지를 결정하는 것이다. 개념적으로, 가능한 입력들의 발생 확률이 어떤 주어진 분포에 대해서 알고리즘의 수행 가능한 시간에 대응되는 분포로 된다. 어떤 간단하지 않은 알고리즘에 대해서 이같은 개념은 성취할 수가 없는 것이므로 입력이 무엇이든지 간에 수행 시간이 “상한(upper bound)”보다 항상 적다는 것을 입증하려고 하는 활용도 통계를 설정하고 “무작위” 입력에 대해서는 평균 수행 시간에 중점을 둔다.

알고리즘 분석에서의 두 번째 단계는 분석과 구현을 구분하기 위해서 알고리즘을 기반으로 한 추상적 연산을 인식하는 것이다. 예를 들면, 특정한 컴퓨터가 한 코드 형태인 `if (a[i] < v) ...`에 대해서 특정한 컴파일러가 어떤 기계어 코드를 생성하는가를 실행하기 위해 걸리는 시간이 몇 마이크로(micro) 초인지의 결정과 정렬 알고리즘이 얼마나 많은 비교를 하는지의 연구는 분리가 된다. 이같은 요소들 모두가 특정한 컴퓨터에 프로그램의 실제 실행 시간을 결정하는데 요구되는 것들이다. 이같은 분리는 특정한 컴퓨터나 특별한 구현과는 적어도 다소 독립적인 알고리즘의 비교를 하도록 하는 것이다.

수반된 추상적 연산들의 수는 원칙적으로 큰 반면에, 고려할 알고리즘의 활용도는 단지 몇 가지의 양에 의존하는 경우가 많다. 일반적으로 어떤 특정한 프로그램에 대한 관련된 양을 인식하는데는 오히려 간단하다. 즉, 그렇게 하는 한 방법은 같은 표본 처리에 대해서 명령의 빈도수를 주기 위해서 “윤곽을 그리는(profiling)” 옵션을 이용케 된다. 본 교재에서는 각 프로그램에 대한 가장 중요한 그런 양에 집중을 할 것이다.

알고리즘 분석에서 세 번째 단계는 근본적인 양들의 각각에 대한 평균의 경우와 최악의 경우 값들을 찾는 목표로서 수학적 분석 그 자체로 진행을 한다. 프로그램의 수행 시간에 관한 상한을 찾는 것은 그리 어렵지 않다. 즉, 도전은 최악의 입력이 만나는 경우 실제로 성취되어지는 가장 좋은 상한을 찾는 것이다. 이것은 최악의 경우를 나타낸다. 평균의 경우는 오히려 복잡한 수학적 분석을 요구한다. 그런 분석은 근본적인 양들, 각 양과 관련된 시간과 얻어진 전체 수행 시간에 대해서 식들에서 성공적으로 수행된다.

원칙적으로 알고리즘의 활용도는 아주 상세한 레벨에 이르기까지 분석될 수가 있고, 컴퓨터의 활용도에 관해 불확실성으로서 혹은 추상적인 양의 몇 가지 성질들을 결정하는 어려움에 의해서만 제한된다. 그러나 완전하고 상세한 분석을 수행하기 위한 가치가 없는 것이므로 해서 상세함을 감추기 위해서 추정에 항상 관심을 둔다.(실제로 매우 거칠게 보이는 추정은 오히려 정확함으로 판명이 된다) 그런 거친 추정은 이전 프로그램이 “시간의 90%는 코드의 10%로서 소비된다는” 말을 통해서 얻어지는 것이 가끔은 아주 쉽다.(이것은 “90%”의 많은 다른 값들에 대해서 과거에 인용된 것이다)

알고리즘 분석은 요구된 상세한 레벨에 답이 나올 때까지 분석을 하고, 추정하고 그리고 분석에서 정제하는 재발생 과정이다. 실제로 다음장에 논의된 것과 같이 처리는 구현에서 개선을 포함하고, 그런 개선은 분석에서 가끔 제시가 된다.

마음 속에 이같은 단서로서 운용하는 방법이 필요할 때, 충분한 분석이 중요한 프로그램에 대해서 수행되는 지식으로서 분류, 안정의 목적으로 프로그램의 수행시간에 대해 거친 추정을 찾는 것이다.

알고리즘 분류

위에 언급한 대로, 대부분의 알고리즘은 파라미터 N 으로 수행 시간이 가장 의미 있게 영향을 주는 처리되어질 데이터 항목들의 수를 지닌다. 파라미터 N 은 다항식의 차수이고 정렬

되거나 검색될 파일의 크기 혹은 그래프에서 노드의 수등이어야 한다. 본 교재에서 모든 알고리즘은 다음 함수 중의 하나에 비례하는 수행 시간을 지낸다.

- 1 프로그램의 대부분 명령들은 한 번 혹은 기껏해야 단지 몇 번 실행된다. 만약 프로그램의 모든 명령들이 이같은 성질을 지니면, 수행 시간은 상수(constant)라고 한다. 이것은 알고리즘 설계에서 얻으려고 애쓰는 명확한 상황이다.
- $\log N$ 프로그램의 수행 시간이 대수(logarithmic)이면, 프로그램은 N 이 증가되므로 서 다소 느리게 얻어진다. 이같은 실행 시간은 큰 문제를 더 적은 문제로 변형시키고 크기를 어떤 상수로서 나누므로써 큰 문제를 해결하는 프로그램들에서 공통적으로 발생된다. 관심의 범주에 대해서, 실행 시간은 “큰” 상수보다 적은 것으로서 고려된다. 대수의 기저(base)는 상수를 변경하는 것이나, 대부분 그렇지 않다. 즉, N 가 천이고, 기저가 10인 경우에 $\log N$ 는 3이고 기저가 2일 때는 10이다. 그리고 N 가 백만이면, $\log N$ 는 2배가 된다. N 가 두 배로 될 때, $\log N$ 는 상수로 증가를 하나 $\log N$ 는 N 가 N^2 으로 증가될때까지 2배로 되지 않는다.
- N 프로그램의 실행 시간이 선형일 때, 적은 양의 처리가 각 입력 요소에 수행을 한다. N 이 백만일 때, 실행시간도 같다. N 이 두배로 증가할때마다, 실행시간도 그렇게 된다. N 개 입력을 처리하는(또는 N 출력을 생성하는) 알고리즘에 대해 최적의 상황이다.
- $N \log N$ 이같은 실행시간은 더 적은 부분 문제로 쪼개고 그것을 독립적으로 해결을 하고, 해를 결합하므로써 문제를 해결하는 알고리즘에서 야기된다. 더 좋은 형용사적 결여로서(linearithmic?), 그런 알고리즘의 실행 시간을 “ $N \log N$ ”이라고 말을 한다. N 이 백만일 때, $N \log N$ 은 아마도 2백만이 된다. N 이 두배로 되면 실행 시간은 두배이상으로 된다.(그러나 훨씬 더 많은 것은 아니다)
- N^2 알고리즘의 실행시간이 2차원(quadratic)일 때, 상대적으로 적은 문제들에서만 이용할 때 유용하다. 2차원의 실행은 모든 데이터 항목들을(아마도 2번의 반복이 내장) 처리하는 알고리즘에서 전형적으로 발생된다. N 이 천일 때, 실행시간은 백만이고 N 이 두배로 될 때 마다 실행시간은 4배가 된다.
- N^3 비슷하게, 데이터 항목의 3승을 처리하는(아마도 반복이 3번) 알고리즘은 입체(cubic) 실행시간을 지니고 적은 문제들에서만 이용이 가능하다. N 이 백일 때 실행시간은 백만이고 N 이 두배로 증가되면 실행시간은 8배로 증가된다.

2^N 지수(exponential) 실행시간으로 된 적은 알고리즘들은 비록 그런 알고리즘이 문제들에 대한 “주먹구구식(brute-force)” 해로서 자연히 발생이 되지만 실제 이용에 대해 적절하다. N 이 20일 때 실행 시간은 백만이다. N 이 두배로 증가할 때, 실행시간은 평방제곱이 된다.

특정한 프로그램의 실행시간은 이같은 항들(앞에 표시된 항)중 하나와 더 적은 항들을 더 하므로써 상수배로 증가된다. 상수 계수와 수반된 항들의 값은 분석의 결과에 따르고 구현에 의존케 된다. 앞 항의 계수는 내부 반복에서 명령의 수와 처리를 하는 것이다. 즉, 알고리즘 설계의 어떤 레벨에서 그런 명령들의 수를 제한하는 것은 신중해야 한다. 큰 N 에 대해서 앞 항들의 효과가 지배적이고 적은 N 에 대해서나 공학 알고리즘에 대해서는 더 많은 항들이 공헌되고 알고리즘의 비교는 더 어렵게 된다. 대부분의 경우에 대해서, 더 상세한 분석이나 실험적 연구로서 효율성이 매우 중요한 곳의 경우에서 수행을 한다는 내재적인 이해로서 “선형”, “ $N \log N$ ”, “입체”등으로서 프로그램의 실행시간을 단순히 언급케 된다.

몇 가지 다른 함수들이 제기된다. 예를 들면, N 에서 실행시간이 입체를 지닌 N^2 입력으로 된 알고리즘은 $N^{3/2}$ 알고리즘으로서 분류가 된다. 또한 몇몇 알고리즘은 $N \log^2 N$ 에 비례하는 실행시간인 부분 프로그램 분해의 두 가지 단계를 지닌다. 이같은 함수 둘 다는 큰 N 에 대해서 N^2 보다는 $N \log N$ 에 훨씬 더 가까운 것으로 된다.

“log”함수에 대해서 살펴보도록 하자. 위에 언급한 대로, 대수의 기저는 상수 요소로서만 어떤 것을 변경한다. 기저가 무엇인가는 관계없이 상수 요소내에 대해서만 해석적 결과를 처리하므로써, “ $\log N$ ”등으로 언급케 된다. 다른 한편, 개념은 어떤 특별한 기저가 이용될 때 더 명확하게 설명된다. 수학에서, 자연 대수(natural logarithm)(기저 $e = 2.718281828\dots$)는 가끔 발생이 되어져서 특별한 축약이 공통적으로 이용된다. 즉, $\log_e N \equiv \ln N$ 이다. 전산학에서 이진 대수(기저 2)는 가끔 이용되며 축약 $\log_2 N \equiv \lg N$ 가 공통적으로 이용된다. 예를 들면, 가장 가까운 정수값으로 반올림되는 $\lg N$ 은 이진수로 N 을 표현하는데 요구되는 비트의 수이다.

그림 6.1은 이같은 함수들의 몇 가지 상대적 크기를 나타낸다. 즉, $\lg N$, $\lg^2 N$, \sqrt{N} , $N \lg N$, $N \lg^2 N$, $N^{3/2}$, N^2 의 근사값은 여러 가지 N 에 대해서 주어진다. 2차 함수는 특히 큰 N 에 대해서 명확히 지배를 하고, 더 적은 함수들 가운데 차이는 적은 N 에 대해서 기대한 대로 존재치 않는다. 예를 들면, $N^{3/2}$ 는 매우 큰 N 에 대해 $N \lg^2 N$ 보다 큰 것이 되나 실제로 발생되는 적은 값에 대해서는 그렇지 않다. 이같은 표는 모든 N 에 대해 함수들의 리터럴 비

$\lg N$	$\lg^2 N$	\sqrt{N}	N	$N \lg N$	$N \lg^2 N$	$N^{3/2}$	N^2
3	9	3	10	30	90	30	100
6	36	10	100	600	3,600	1,000	10,000
9	81	31	1,000	9,000	81,000	31,000	1,000,000
13	169	100	10,000	130,000	1,690,000	1,000,000	100,000,000
16	256	316	100,000	1,600,000	25,600,000	31,600,000	ten billion
19	361	1,000	1,000,000	19,000,000	361,000,000	one billion	one trillion

그림 6.1 함수들의 상대적인 값들의 근사치

교를 할 의양은 없다. 특별한 알고리즘에 관련된 수들, 표들과 그래프들은 그것을 수행케 된다. 그러나 이것은 실제적인 첫 번째 생각나게 하는 것은 아니다.

계산 복잡도(Computational Complexity)

알고리즘의 활용도를 연구하는 한 방법은 입력들의 수에(또는 몇 가지 다른 변수) 대해 실행 시간(또는 다른 척도)의 함수적 의존도를 결정하기 위해서 상수 요소는 무시하고 최악의 활용도를 연구하는 것이다. 이같은 방법은 프로그램들의 실행시간에 관해서 수학적 문장들을 입증하도록 허용하기 때문에 매력적이다. 예를 들면, 병합 정렬(Mergesort; 11장 참조)의 실행시간은 $N \log N$ 에 비례한다고 말할 수가 있다.

과정에서 첫 번째 단계는 어떤 특정한 구현에서 알고리즘의 분석을 분리시키는 동시에 그 반면 수학적으로 “비례”라는 개념을 이용하는 것이다. 개념은 분석에서 상수를 무시하는 것이다. 대부분의 경우 알고리즘의 실행시간이 N 에 비례하거나 혹은 $\log N$ 에 비례하는지 알기를 원하면, 알고리즘이 마이크로 컴퓨터 혹은 수퍼 컴퓨터에 수행하는지에는 관계가 없고, 내부의 반복이 단지 몇 가지 명령들로서 구현이 되는지 혹은 많은 명령으로 나쁘게 구현이 되는지에 대해서도 관계가 없다. 수학적 견지에서, 이같은 두 요소들은 똑같은 것이다.

이같은 개념을 정확하게 표현하는 수학적 가공품을 다음에 정의된 것과 같이 O -기법 또는 “큰- O 기법”이라고 부른다.

기법. $g(N)$ 가 모든 $N > N_0$ 에 대해 $c_0 f(N)$ 보다 적은 상수 c_0 과 N_0 이 존재하는 경우에 함수 $g(N)$ 는 $O(f(N))$ 이라고 말한다.

이것은 “비례”의 개념을 나타내는 것이고, 특정한 기계 특징에 대해 상세한 내용을 고려함에서 분석가들로 하여금 자유롭게 만들어 준다. 더구나, 알고리즘의 실행시간이 $O(f(N))$ 인 문장은 알고리즘의 입력과는 무관하다. 입력이나 구현에 대한 것이 아니고 알고리즘 연구에 관심이 있으므로 해서, O -기법은 입력들과 구현의 상세함 둘다와 별개인 실행시간에 상한을 나타내는 유용한 방법이다.

O -기법은 활용도에 의해 알고리즘을 분류하도록 분석가들에게 도움을 주고, 중요한 문제들에 대해 “가장 좋은” 알고리즘들에 대한 검색에서 알고리즘 설계자를 인도하는데 상당히 유용한 것이다. 계산 기하학의 연구 목적은 어떤 함수 f 에 대해 실행시간이 $O(f(N))$ 임과 어떤 “더 적은” 함수 $g(N)$ ($\lim_{N \rightarrow \infty} g(N)/f(N) = 0$ 로 된 함수)에 대한 $O(g(N))$ 의 실행시간으로 된 알고리즘은 없음을 나타낸다. 최악의 경우 실행시간에 대한 “상한”과 “하한” 둘 다를 제공하려고 노력을 한다. 상한을 증명함은 문자의 빈도수를 계산하고 분석하는 문제이다.(다음의 장들에서 많은 예를 보게 된다) 그리고 하한을 증명함은 기계 모델을 조심스럽게 구성하고 문제를 해결(이것에 대해서는 거의 취급하지 않는다)하기 위해서 어떤 알고리즘에 의해서 기본적인 연산들이 수행이 되는지를 결정하는 어려운 문제이다. 계산적 연구가 알고리즘의 상한이 그것의 하한과 일치됨을 나타낼 때, 기본적으로 더 빠른 알고리즘을 설계하려고 하는 것은 효과가 없는 것과 구현에 중점을 두어 시작함으로써 몇가지 확신을 지닌다. 이같은 견지가 최근 알고리즘 설계자들에게 많은 도움을 준 것으로 입증된다.

그러나, 적어도 4가지 이유로서 O -기법을 이용하여 표현된 결과를 해독하는 것에 극히 주의해야 한다. 첫째, 그것은 “상한”이고 의문시 되는 양은 훨씬 적다. 둘째, 최악의 경우를 야기시키는 입력은 실제로 발생되지 않는 것과 같다. 셋째, 상수 ω 은 알 수가 없고 적게 존재될 필요도 없다. 네 번째, 상수 N_0 은 알 수가 없고 적게 존재될 필요도 없다. 차례로 이것들 각각을 보기로 하자.

알고리즘의 실행 시간이 $O(f(N))$ 인 문장은 알고리즘이 그렇게 오래 걸리는 것을 의미하지는 않았다. 즉, 분석가는 결코 더 오랫동안 걸리지 않는 것을 입증할 수가 있을때만 말할 수가 있다. 실제로 실행시간은 항상 더 적게 된다. 더 좋은 기법은 실행시간이 $O(f(N))$ 인 것에 대한 몇몇 입력이 존재하는 것과 최악의 경우 입력을 구성하기가 오히려 어렵다는 것에 대해 많은 알고리즘이 존재함을 알고 있는 상황을 대처하도록 개발된 것이다.

비록 최악의 경우 입력을 알고 있는 경우라 할지라도, 실제적으로 나열된 입력들은 훨씬 더 낮은 수행시간으로 된다. 예를 들면, 아마도 가장 널리 이용된 정렬 알고리즘인 Quicksort

N	$\frac{1}{4}N \lg N$	$\frac{1}{2}N \lg^2 N$	$N \lg^2 N$	$N^{3/2}$
10	22	45	90	30
100	900	1,800	3,600	1,000
1,000	20,250	40,500	81,000	31,000
10,000	422,500	845,000	1,690,000	1,000,000
100,000	6,400,000	12,800,000	25,600,000	31,600,000
1,000,000	90,250,000	180,500,000	361,000,000	1,000,000,000

그림 6.2 함수들을 비교함에 있어서 상수 요소들의 중요성

는 실행시간이 $O(N^2)$ 이나 실제로 나열된 입력들에 대한 실행시간은 $N \log N$ 에 비례하도록 배열하는 것이 가능하다.

O -기법에 내포된 상수 c_0 과 N_0 은 실제로 중요한 구현 내용에서 가끔은 감추어져 있다. 명백히, 알고리즘의 실행시간이 $O(f(N))$ 이라고 말하기 위함은 N 이 N_0 보다 더 적은 것으로 발생하는 경우는 실행시간에 관해서 할 이야기가 없고 c_0 은 나쁜 최악의 경우를 피하기 위해서 설계된 “불필요한” 많은 양을 감추고 있는 것이다. $\log N$ 백단위를 이용한 한 개에 대해 것에 N^2 나노 초(nano seconds)를 이용한 알고리즘을 택하게 된다. 그러나 O -기법의 기초로서 이같은 선택은 하지는 않는다. 그림 6.2는 범위가 $0 \leq n \leq 1,000,000$ 에서 상수들의 더 실제적인 값들로서 두 가지 전형적인 함수들에 대한 상황을 나타낸 것이다. 점진적으로는 그것이 가장 큰 것이므로 서, 4개에서 가장 큰 것인 것으로 오해될 수가 있는 $N^{3/2}$ 함수는 실제로 적은 N 에 대해 가장 적은 값들이 가운데 존재케되고 N 이 수천 배로 잘 처리될 때까지 $N \lg^2 N$ 보다 적다. 실행시간이 이같은 함수들에 의존되는 프로그램들은 상수 요소와 구현의 상세함에 주의를 기울이지 않고서는 비교가 될 수 없다.

예를 들면, 실행시간이 $O(N)$ 로 된 것을 지지하면서 실행시간이 $O(N^2)$ 인 알고리즘을 이용하여 이전에 두 번은 확고하게 생각을 해야하나 O -기법으로 표현된 계산적 결과를 그 어느 것도 맹목적으로 따르지 않는다. 본 교재에서 고려된 형태의 알고리즘 실제 구현에 대해서, 복잡도 증명은 너무 일반적인 것이고 O -기법은 너무 부정확해서 도움이 되질 않는다. 계산적 복잡도는 그것의 성질들에 관해 더 자세하게 나타내도록 하는 알고리즘 분석을 정리하는 진보적인 과정에서 바로 첫 번째 단계를 고려하는 것이다. 본 교재에서는 실제적인 구현에 더 가깝도록한 나중의 단계에 집중을 할 것이다.

평균 경우 분석

알고리즘의 활용도를 공부하는 또 다른 방법은 평균의 경우를 조사하는 것이다. 간단한 상황에서 알고리즘에 대한 입력들을 정확하게 특성화할 수가 있다. 예를 들면, 정렬 알고리즘은 N 무작위 정수형의 배열에 대해 처리를 하거나 혹은 기하학 알고리즘은 0과 1사이 좌표의 평면에서 N 무작위 점들의 집합을 처리케 된다. 그때, 각 명령이 실행되는 평균 횟수를 계산하고 명령에 대해 요구된 시간을 각 명령의 빈도수로 곱하고 그것 모두를 함께 더함으로써 프로그램의 평균 실행시간을 계산한다. 그러나, 여기서 차례로 설명할 이같은 방법에는 적어도 세 가지 어려운 점이 존재케 된다.

첫째, 어떤 컴퓨터상에서 각 명령에 대해 요구되는 시간의 정확한 양을 결정하는 것은 오히려 어렵다. 최악으로, 이것은 변경하기가 쉽고 한 컴퓨터에 대해 많은 상세한 분석은 또 다른 컴퓨터상에 같은 알고리즘의 실행시간에 전혀 관계가 없을 수도 있다. 이것은 계산 복잡도 연구가 피하도록 설계된 문제들의 형태이다.

둘째, 평균의 경우 분석 그 자체는 복잡하고 상세한 인수들을 요구하는 어려운 수학적 도전이다. 그 본질에 의하면, 상한을 증명하는데 수반되는 수학들은 그것을 자세하게 나타낼 필요가 없기 때문에 덜 복잡하다. 많은 알고리즘의 평균 경우 활용도는 미지로 되어 있다.

가장 심각한 세 번째, 평균 경우 분석에서 입력 모델은 실제적으로 나열된 입력들을 정확하게 특징지을수가 없고 혹은 자연스러운 입력 모델이 전혀 없다는 것이다. 영어 텍스트를 처리하는 프로그램에 대한 입력을 어떻게 특성화하는가? 다른 한편, 정렬 알고리즘에 대해 “무작위적으로 순서화된 파일” 혹은 기하학 알고리즘에 대한 “무작위 점 집합”과 같은 입력 모델들의 이용에 대해서 논의가 별로 없고, 그런 모델에 대해 실제 응용에서 처리되는 프로그램들의 활용도를 정확히 예언하는 수학적 결과로 이끌어내는 것이 가능하다는 것이다. 그런 결과들의 유추는 정상적으로 이 교재의 범위를 벗어나지만, 몇 가지 예제들(9장 참조)을 보게 되고 적절하게 관련된 결과를 인용케 된다.

근사치와 점진적 결과 (Approximate and Asymptotic Results)

물론 수학적 분석의 결과는 정확하지 않으나 정확한 기술적 의미로서는 근사치이다. 즉, 결과는 연속적으로 감소되어지는 항들로 구성된 식이다. 프로그램의 내부 반복에 관심을 두

는 것과 같이, 수학 식의 제일 앞에 존재하는 항(가장 큰 항)에도 관심을 둔다. O -기법은 원래 개발되고 적절히 이용되어진 것은 바로 응용의 이같은 형태이고 수학적 결과에 좋은 근사치를 주는 간결한 문장들이 되도록 허용하는 것이다.

예를 들면, 특정한 알고리즘은 평균적으로(말하자면) $N \lg N$ 번 반복되는 내부 반복과 N 번 반복되는 외부 부분 그리고 정확히 한 번 실행되는 몇 가지 초기화 코드를 지닌 것으로 결정된다고 가정하자.(몇몇 수학적 분석뒤에) 내부 루프의 각 반복은 a_0 마이크로 초(micro seconds)를 요구하고 외부 부분은 a_1 마이크로 초를, 초기화 부분은 a_2 마이크로 초를 요구함을 결정(구현을 조심스럽고 자세히 본 뒤에)한다고 가정해 보자. 그때 프로그램의 평균 실행 시간(마이크로 초)은 다음과 같다.

$$a_0 N \lg N + a_1 N + a_2$$

그러나 실행시간이 다음과 같은 것도 사실이다.

$$a_0 N \lg N + O(N)$$

($O(N)$ 의 정의에서 이것을 체크하도록 해야 한다) 근사치 답에 관심을 지니고 있다면, 큰 N 에 대해서 a_1 이나 a_2 의 값을 찾을 필요가 없기 때문에 의미가 있다. 더 중요한 것은 분석하기가 어려운 정확한 실행시간에서 다른 항들이 존재케 된다. 즉, O -기법은 그런 항으로 귀찮게 함이 없이 큰 N 에 대해 근사치 답을 얻도록 하는 방법으로 제공케 된다.

기술적으로, O -기법의 정의는 상수 a_0 의 크기에 관해 무엇인가에 아무것도 말할 수 없기 때문에 적은 항들은 이같은 방법에서 무시가 된다는 것은 확신이 없다. 즉, 그것은 매우 크다. 그러나(항상 귀찮게 하지는 않지만) N 에 비례할 때 적은 상수들에 경계를 놓는 것은 그런 경우에서 방법들이 존재케 된다. 그래서 잘 기술된 제일 앞의 항일 때 O -기법에 의해서 표현된 양들을 무시함으로 공정하게 된다. 이같은 것을 수행할 때, 드물게는 그렇게 수행을 하지만 절대적으로 필요한 경우, 그런 증명을 수행하는 지식에서 안전하게 된다.

사실, 함수 $f(N)$ 는 다른 함수 $g(N)$ 에 점진적으로 비교되어진 가장 큰 경우, 본 교재에서는 $f(N)+O(g(N))$ 를 의미하기 위해서 " $f(N)$ 에 관한" 용어를 이용케 된다. 수학적 편견으로 잃어버린 것은 수학적 상세함에서 보다 알고리즘의 활용도에 더 관심을 두기 때문에 명확하게 얻을 수가 있다. 그런 경우 큰 N 에 대해서(모든 N 에 대한 경우는 아닌 경우), 의문시 되는

양은 $f(N)$ 에 오히려 근접된다. 예를 들면, 비록 양이 $N(N-1)/2$ 인 것을 안다고 할지라도, “약” $N^2/2$ 로서 언급된다. 이것은 훨씬 빠르게 이해가 되고 예를 들면, $N = 1000$ 에 대한 일 퍼센트의 $1/10$ 로서만 된다. 우리의 목표는 알고리즘의 활용도를 기술할 때 정확하고 간결한 것으로 존재시키는 것이다.

기본적인 재발생(basic recurrences)

다음 장에서 볼 수 있듯이, 상당히 많은 알고리즘은 원래 문제를 해결하기 위해서 부분 문제들에 대한 해를 이용해서 큰 문제를 더 작은 문제로 재귀적으로 나누는 원칙이 기본이다. 그런 알고리즘의 실행시간은 분해의 비용과 부 프로그램의 수와 크기에 의해서 결정된다. 본 절에서는 그런 알고리즘을 분석하는 기본적인 방법을 보고 공부하게 될 많은 알고리즘들의 분석에서 제기되는 몇 가지 표준 공식에 대한 해를 구하게 된다. 본 절에서 공식의 수학적 성질을 이해하는 것은 교재를 통한 알고리즘의 활용도 성질들에 통찰을 주게 된다.

재귀 프로그램의 본질은 크기 N 의 입력에 대한 실행시간은 더 작은 입력들에 대해서 그것의 실행시간에 의존케 된다. 즉, 이것은 이전 장의 앞에서 나열된 재발생 관계(recurrence relations)로 되돌린다. 그런 공식은 대응되는 알고리즘들의 활용도를 자세히 설명하는 것이다. 실행시간을 유추하기 위해서 재발생을 해결케 된다. 특별한 알고리즘에 관련된 더 정밀한 인수들은 알고리즘에 도달될 때 나타난다. 즉, 여기서는 알고리즘이 아니고 공식에 중점을 둘 것이다.

【공식 1】 이같은 재발생은 한 항목을 제거하기 위해서 입력을 통해서 반복되는 재귀 프로그램에 대해 발생된다.

$$C_N = C_{N-1} + N$$

여기서,

$C_1 = 1$ 로된 $N \geq 2$ 에 대한 것이다.

【해답】 C_N 은 약 $N^2/2$ 이다. 그런 재발생을 해결하기 위해서, 다음과 같이 자신에 적용함으로써 볼 수가 있다.

$$\begin{aligned}
C_N &= C_{N-1} + N \\
&= C_{N-2} + (N-1) + N \\
&= C_{N-3} + (N-2) + (N-1) + N \\
&\vdots \\
&= C_1 + 2 + \cdots + (N-2) + (N-1) + N \\
&= 1 + 2 + \cdots + (N-2) + (N-1) + N \\
&= \frac{N(N+1)}{2}
\end{aligned}$$

합 $1 + 2 + \dots + (N-2) + (N-1) + N$ 을 계산하는 것은 간단하다. 즉, 위에 주어진 결과는 같은 합을 그러나 역순으로서 항단위로 더하므로 설정이 된다. 값을 두 번 찾는 이같은 결과는 N 항으로 구성이 되고 각각은 $N+1$ 에 합해진다.

【공식 2】 이같은 재발생은 한 단계로서 입력을 반으로 나누는 재귀 프로그램에 대해 발생된다.

$$C_N = C_{N/2} + 1$$

여기서,

$C_1 = 0$ 인 $N \geq 2$ 에 대한 것이다.

【해답】 C_N 은 약 $\lg N$ 이다. 기술된 대로, 이같은 방정식은 N 이 짝수가 아닐 때는 의미가 없다. $N/2$ 를 정수형 나눗셈이라고 가정하자. 즉, 여기서 $N = 2^n$ 라고 가정을 하면, 재발생은 항상 잘 정의가 된다. ($n = \lg N$ 이다) 그러나 재발생은 첫 번째 재발생보다 훨씬 더 쉽게 나타낸다.

$$\begin{aligned}
C_{2^n} &= C_{2^{n-1}} + 1 \\
&= C_{2^{n-2}} + 1 + 1 \\
&= C_{2^{n-3}} + 3 \\
&\vdots \\
&= C_{2^0} + n \\
&= n
\end{aligned}$$

일반적인 N 에 대한 자세한 해는 N 의 이진 표현 성질에 의존되나, C_N 는 모든 N 에 대해 약 $\lg N$ 인 것으로 판명된다.

【공식 3】 이같은 재발생은 입력을 반으로 나누는 재귀 프로그램에 발생이 되나 아마도 입력에서 모든 항목을 조사해야 한다.

$$C_N = C_{N/2} + N$$

여기서,

$C_1 = 0$ 인 $N \geq 2$ 에 대한 것이다.

【해답】 C_N 은 약 $2N$ 이다. 이것은 $N + N/2 + N/4 + N/8 + \dots$ 합이 된다.(위에 언급된 것과 같이, N 이 2의 멱승일 때 자세하게 정의된다) 만약 순서가 무한이면, 정확히 $2N$ 을 계산하는 간단한 기하학적 순서들이다. 일반적인 N 에 대해, 자세한 해는 N 의 이진 표현을 수반한다.

【공식 4】 이같은 재발생은 그것이 두 개의 반으로 분리된 후, 동안 혹은 전에 입력을 통해 선형적으로 전달이 되도록 하는 재귀 프로그램에 대해 야기된다.

$$C_N = 2C_{N/2} + N$$

여기서,

$C_1 = 0$ 인 $N \geq 2$ 에 대한 것이다.

【해답】 C_N 은 약 $N \lg N$ 이다. 이것은 표준적인 분할-정복 알고리즘의 모형이므로 가장 널리 인용되는 해이다.

$$C_{2^n} = 2C_{2^{n-1}} + 2^n$$

$$\frac{C_{2^n}}{2^n} = \frac{C_{2^{n-1}}}{2^{n-1}} + 1$$

$$= \frac{C_{2^{n-2}}}{2^{n-2}} + 1 + 1$$

\vdots

$$= n$$

해는 공식 2에서 처럼 매우 많이 개발된다. 그러나 재발생을 투명하게 하기 위해서 두 번째 단계에서 $2'$ 로써 재발생의 양쪽을 나누는 부가적인 속임이 있다.

【공식 5】 이같은 재발생은 5장에서의 자를 그리는 프로그램에서와 같이 한 단계에서 입력을 반으로 분리시켜 두 개로하는 재귀 프로그램에서 발생된다.

$$C_N = 2C_{N/2} + 1$$

여기서,

$C_1 = 0$ 인 $N \geq 2$ 에 대한 것이다.

【해답】 C_N 은 약 $2N$ 이다. 이것은 공식 4와 같은 방법이다.

다른 초기 조건이나 부가적인 항에서 다른 것을 수반하면서 이같은 공식의 사소한 편차는 비록 비슷하게 보이는 몇 가지 재발생을 풀기에는 오히려 어려운 것이지만 같은 해 기법을 이용해서 처리된다.(수학적 엄정함으로 그런 방정식을 처리하는 여러 가지 고급의 일반적인 기법이다) 다음장들에서 몇 가지 더 복잡한 재발생을 보게 될것이나 그것이 발생될 때까지 그것의 해의 논의를 미루게 된다.

통찰

본 교재의 많은 알고리즘은 상세한 수학적 분석과 활용도 연구는 너무 복잡해서 여기서 논의하지 못하는 것으로 된다. 정말로, 논의될 많은 알고리즘을 추천할 수 있다는 것은 바로 그런 연구들의 기본에 있다.

모든 알고리즘은 그런 심한 정밀성에 가치는 없다. 설계의 과정에서 상세함 없이도 설계 과정을 인도하는 근사치 활용도 지시자로서 처리하는 것을 택한다. 설계가 더 정교하게 되므로 서, 분석과 더 복잡한 수학적 도구들을 적용할 필요가 있다. 가끔, 설계 처리는 어떤 특정한 응용에서라기 보다는 오히려 “이론적” 알고리즘으로 이끄는 상세하고 복잡한 연구로 된다. 복잡한 연구에서 거친 분석들은 즉각적으로 효율적인 실제 알고리즘으로 번역된다는 가정은 일반적인 잘못이다. 다른 한편, 계산 복잡도는 중요하고 새로운 방법들은 기본으로한 설계에서 출발을 제시하는데 강력한 도구가 된다.

그것이 어떻게 수행되는가의 몇 가지 지시 없이 알고리즘을 이용할 수는 없다. 즉, 이 장에 기술된 방법들은 널리 이용되는 알고리즘에 대한 활용도의 몇 가지 지시를 제공하는데 도움을 주고 이것들을 다음장에서 보게 된다. 다음 장에서는 알고리즘을 선택할 때, 처리될 다른 중요한 요소에 대해 논의케 된다.

연습 문제

1. 한 알고리즘의 실행시간이 $O(N \log N)$ 이고, 다른 알고리즘의 실행시간이 $O(N^3)$ 이라고 가정하자. 알고리즘들의 상대적 활용도에 관해 이것은 무엇이라고 말하는가?
2. 한 알고리즘의 실행시간이 항상 약 $N \log N$ 이고, 다른 알고리즘의 실행시간이 $O(N^3)$ 이라고 가정하자. 알고리즘들의 상대적 활용도에 관해 이것은 무엇이라고 말하는가?
3. 한 알고리즘의 실행시간이 항상 약 $N \log N$ 이고, 다른 알고리즘의 실행시간이 항상 약 N^3 이라고 가정하자. 알고리즘들의 상대적 활용도에 관해 이것은 무엇이라고 말하는가?
4. $O(1)$ 와 $O(2)$ 의 차이를 설명하시오.

5. 다음 재발생을 푸시오.

$$C_N = C_{N/2} + N^2$$

여기서,

$C_1 = 0$ 인 $N \geq 2$ 에 대한 것이고 N 은 2의 멍승이다.

6. N 의 어떤 값이 $10N \lg N > 2N^2$ 에 대한 것인가?
7. 5 장에 논의대로, 공식 2에서 C_N 의 정확한 값을 계산하는 프로그램을 쓰시오. 그리고 결과를 $\lg N$ 과 비교하여라.
8. 공식 2에 대한 상세한 해가 $\lg N + O(1)$ 임을 증명하라.
9. $\log_2 N$ 보다 적은 가장 큰 정수를 계산하는 재귀 프로그램을 기술하시오.(힌트: $N > 1$ 에 대해, $N/2$ 에 대한 이같은 함수의 값은 N 에 대한 것보다 한 개 크다)
10. 이전 연습 문제에서 문제에 대한 반복 프로그램을 기술하라. 그리고 C++ 라이브러리 서브루틴을 이용해서 계산을 하는 프로그램을 기술하라. 만약 자신의 컴퓨터에서 가능한 경우, 이같은 세 가지 프로그램들의 활용도를 비교하라.

7 장

알고리즘 구현

1 장에서 언급 한대로, 본 교재의 초점은 알고리즘 그 자체에 있는 것이다. 즉, 알고리즘을 논의할 때, 알고리즘 활용도가 어떤 큰 일의 성공적인 수행에 있어서 중요한 요인인 경우처럼 취급케 된다. 이같은 관점은 그런 상황이 각 알고리즘에 대해서 발생하는 이유와 어떤 문제를 해결하기 위한 효율적인 방법을 찾는 데 조심스런 주의는 더 효율적인 알고리즘으로 이끌수 있다는 두 가지 이유로서 정당화된다. 물론 이같은 좁은 초점은 컴퓨터로 복잡한 문제를 해결을 할 때 고려해야하는 매우 현실적인 요소들이 존재하기 때문에 오히려 비현실적이다. 본 장에서는 실제적인 응용에서 유용하게 기술되는 이상적인 알고리즘을 만드는데 관련된 주제에 대해서 논하고자 한다.

결국 알고리즘의 성질들은 동전의 한쪽 면에서만 존재케 된다. 즉, 컴퓨터는 문제 그 자체를 잘 이해를 하는 경우에만 문제를 효과적으로 해결하는데 이용된다. 응용에서 성질들의 조심스런 고려는 본 교재의 범위를 벗어난 것이다. 여기서의 의도는 그것들의 이용에 관해서 더 지적인 판단을 하도록 하는 기본적인 알고리즘에 관한 충분한 정보를 제공하는 것이다. 고려되는 대부분의 알고리즘은 여러 가지 응용에서 유용하다고 입증되었다. 여러 가지 문제들을 해결하는데 가능한 알고리즘의 범주는 여러 가지 응용들의 필요성 범주를 소유하는 것이다.(한 예를 선택하면) 거기에는 “가장 좋은” 검색 알고리즘이 없으나, 항공 예약 시스템의 응용에 대해 적절한 방법이 있고, 다른 방법으로는 코드를 없애는 프로그램의 내부적 반복 이용에 아주 유용한 것이 있다.

어떤 궁극적인 구현에 대한 것 없이도 방법들을 개발하는 이론적인 알고리즘 설계자와 잘 이해되는 문제들을 해결하는데 여러 가지 방법을 마구 이용하는 응용 시스템 프로그램들의 마음에서 가능하다는 것은 제외하고 알고리즘들은 거의 비어있지는 않게 된다. 적당한 알고

리즘 설계로 인하여 어떤 사고를 구현함에서 잠재적인 설계 판단의 요인으로 이용하고 적절한 응용 프로그래밍은 어떤 사고를 이용된 기본적인 방법들의 활용도 성질로 이용케 된다.

알고리즘 선택

다음의 장에서 보게 될과 같이, 간단한 “주먹구구식” (그러나 아마도 비효율적인) 해에서부터 복잡하고 “잘 조정된”(그리고 심지어 최적인 것) 해답에 이르는 범주로서 다른 활용도 특징으로 된 모든 것들과 각 프로그램을 해결하는데 이용되는 여러 가지 알고리즘이 존재케 된다.(일반적으로, 더 효율적인 알고리즘이 존재하면, 더 복잡한 구현이 존재한다는 것은 사실이 아니다. 그 이유는 몇 가지 가장 좋은 알고리즘이 오히려 간결하고 좋으나 본 논의의 목적에 대해서는 이같은 규칙이 유지된다고 가정했기 때문이다) 위에 언급된 대로, 문제의 필요성을 분석함이 없이 어떤 문제에 대해 어떤 알고리즘이 이용되는 지를 결정할 수는 없다. 수행될 프로그램은 얼마나 존재하는가? 이용되어진 컴퓨터 시스템의 일반적인 특징은 무엇인가? 알고리즘은 큰 응용의 적은 부분인가?

구현에서 첫 번째 구현은 주어진 문제를 해결하는 가장 간단한 알고리즘을 먼저 구현하는 것이다. 만약 나열된 특정한 문제 예시가 쉽게 판명되는 경우, 그때 간단한 알고리즘은 문제를 해결하고 그 어느 것도 더 이상 수행되어질 필요는 없다. 만약 더 복잡한 알고리즘을 요청한 경우, 그때 간단한 구현은 같은 경우에 대해 정확한 체크와 활용도 특징을 평가하는데 기준선으로 제공케 된다.

만약 알고리즘이 너무 크지 않은 경우에 대해 여러 번만 수행된다면, 프로그래머로 하여금 복잡한 구현을 개발하는데 의미 있는 양의 부가적인 시간이 걸리는 것보다 컴퓨터로 하여금 다소 덜 효율적인 알고리즘을 실행하는데 적은 양의 부가적인 시간을 소비하는 것을 택하게 된다. 물론, 원래 계획했던 것 이상으로 프로그램을 이용해서 끝을 내는 위험이 존재케 되므로해서, 항상 더 좋은 알고리즘을 구현하고 시작시킬 준비를 해야 한다.

만약 알고리즘이 큰 시스템의 일부분으로서 구현된다면, “주먹구구식” 구현은 신뢰할 수 있는 방법으로서 요구된 가능성을 제공케 되고 활용도는 나중에 더 좋은 알고리즘으로 대체함으로써 제어된 방법으로 상향 조정된다. 물론, 나중에 상향 조정하기 어려운 그런 방법으로서 알고리즘을 구현함으로써 옵션을 강제로 근접시키기 위한 것이 아니고 대체적으로 시스템의 활용도를 연구할 때 활용도 장애가 어느 알고리즘에서 존재하는 가를 조심스럽게 보게 된다. 또한, 큰 시스템에서 시스템의 설계 요구는 어느 알고리즘이 좋은가에서 시작해서부터

지시되는 경우가 있다. 예를 들면, 아마도 시스템 중심 자료구조는 연결 리스트나 트리의 특별한 형태이므로 그같은 특정한 구조를 기반으로한 알고리즘이 선택된다. 다른 한편, 결국에는 본 교재에서 논의된 것들과 같이 몇 가지 기본적인 알고리즘의 활용도에 전체 시스템의 활용도가 의존이 되기 때문에 그런 시스템 중심의 판단을 하려고 할 때 이용되어진 알고리즘들에 주의를 기울여야 한다.

만약 알고리즘이 매우 큰 문제들상에서 몇 번만 수행된다면, 의미 있는 출력을 생성하고 그것이 얼마나 오래 지속 되는지의 몇 가지 추정을 나타내게 된다는 자신을 갖게 된다. 간단한 구현은 출력을 체크하는 도구의 개발을 포함해서 오랜 수행 동안 설정하는데 아주 유용하게 된다.

알고리즘을 선택하는데 가장 공통적인 잘못은 활용도 특징을 무시하는 것이다. 더 빠른 알고리즘은 가끔은 더 복잡하고 구현자는 부가된 복잡성을 처리하기 위한 것을 피하는 더 늦은 알고리즘을 받아 드리게 된다. 그러나 빠른 알고리즘은 훨씬 더 복잡한 것이 아니고 조금 부가된 복잡성을 지닌 것을 처리하는 것은 느린 알고리즘을 처리하는 것을 피하기 위해 지불한 적은 비용이다. 놀라만한 수의 컴퓨터 시스템 이용자들은 다소 더 복잡한 $N \log N$ 알고리즘이 쪼개진 시간내에 수행될 수있도록 할 때 간단한 2차 알고리즘을 끝내기 위해서 기다리는 부수적인 시간이 없어진다.

알고리즘을 선택하는데 두 번째 가장 공통적인 잘못은 활용도 특징에 너무 많은 주의를 기울리는 것이다. $N \log N$ 알고리즘은 같은 문제에 대해 2차 알고리즘보다 다소 더 복잡한 것일 뿐이다. 그러나 더 좋은 $N \log N$ 알고리즘은 복잡도에서 부가적인 증가로 야기시키게 된다.(그리고 N 의 매우 큰 값 등에 대해서만 실제적으로 더 빠르다) 또한 많은 프로그램들은 실제적으로 단지 몇 번만을 수행케 된다. 즉, 최적 알고리즘을 구현하고 디버그하는데 요구되는 시간은 다소 더 적게 수행되는 것을 단순히 요구하는 시간 이상으로 요구된다.

실험적 분석

6장에서 언급 한대로, 수학적 분석은 주어진 알고리즘이 주어진 상황에서 수행하도록 얼마나 잘 기대를 할 수 있는가에 매우 적은 주의를 하게 된다. 그런 경우에서 “전형적인” 입력에 대한 활용도를 감시하고 알고리즘을 조심스럽게 구현하는 실험적인 분석에 의존할 필요가 있게 된다. 사실, 이것은 정당성을 체크하기 위해서 완전한 수학적 결과가 이용되어질 때 초차 수행이 된다는 것이다.

같은 문제를 해결하기 위해서 두 개 알고리즘이 주어진 경우, 방법에는 신비스러운 것이 존재하지 않는다. 즉, 어느 것이 더 오래 걸리는 가를 보기 위해서 그것들 둘 다를 수행 시켜 보자! 이것은 너무 명백해서 언급하지는 않는다. 그러나 알고리즘의 비교적 연구에서 아마도 가장 공통적인 생략이 된다. 한 알고리즘이 다른 것보다 10배 이상으로 빠르다는 사실은 한 알고리즘을 끝내기 위해서 3초를 기다리는 사람과 다른 알고리즘을 끝내기 위해서 30초를 기다리는 사람의 주목을 피하려고 하는 것은 아니다. 그러나 수학적 분석에서 적은 상수의 불필요한 요소로서 보기는 매우 쉽다.

그러나 특히 다른 기계, 컴파일러 혹은 시스템이 동반되는 경우나 혹은 잘 기술이 안된 입력들로서된 매우 큰 프로그램들이 비교를 하는 경우에 구현을 비교를 할 때 실수를 하기가 또한 쉽다. 정말로, 알고리즘에 대한 수학적 분석의 개발로 이끌어지는 요인은 아마도 어느 활용도가 조심스런 분석을 통해서 잘 이해가 되는가는 “기준점”에 의존케 되는 경향이 있다.

실험적으로 프로그램을 비교함에 있어서 주된 위험은 한 구현이 다른 것보다 더 “최적”인 것이다. 제안된 새로운 알고리즘의 창시자는 구현의 모든 면에 주의를 기울려야 하는 것이지 고전적인 경쟁 알고리즘을 상세하게 구현하는 것은 아니다. 알고리즘을 비교하는데 실험적인 연구의 정확성에 대해서는 같은 양의 주의가 구현에도 이루어져야 하는 것이다. 다행히도, 이것은 다음의 경우에 가끔 발생된다. 즉, 많은 훌륭한 알고리즘은 같은 문제에 대해 다른 알고리즘에 대한 상대적으로 적은 양의 변형에서 이끌어지고 비교적인 연구는 실제로로 정당하게 된다.

중요하고 특별한 경우는 알고리즘이 그 자체의 다른 버전 것과 비교를 할 때 혹은 다른 구현 방법들이 비교가 되어질 때에 발생된다. 특정한 변형이나 구현 개념의 효력을 체크하는 훌륭한 방법은 몇 가지 “전형적인” 입력에 둘 다의 버전을 수행하는 것으로 그때 더 빠른 것에 더 많은 주의를 기울려야 한다. 다시, 이것은 너무 명백해서 언급을 하지 않는 것처럼 보이나, 알고리즘 설계에서 수반이 된 놀라만한 수의 연구자들은 그것의 설계를 결코 구현을 하지 않으므로 해서 사용자로 하여금 조심하게 한다!

위와 6장의 앞에서 기술한 대로, 여기의 견해는 설계, 구현, 수학적 분석과 실험적 분석들 모두가 좋은 알고리즘의 좋은 구현을 개발함에 중요한 방법들에 공헌하는데 있다는 것이다. 프로그램들의 성질들에 관해서 정보를 얻을 수 있는 어느 도구를 이용하는가 그리고 그때 그 정보를 기반으로 새로운 프로그램을 변경하고 개발하는 것이다. 다른 한편, 활용도 개선의 희망으로서 적은 변화의 많은 수들을 만들도록 하는데 항상 정당화되는 것은 아니다. 다음으로, 이 문제에 대해서 더 자세히 보도록 하자.

프로그램 최적화

더 빠른 또 다른 버전을 생성하기 위해서 프로그램에 변화를 증가시키도록 하는 일반적인 과정을 프로그램 최적화라고 부른다. 이것은 “가장 좋은” 구현을 보기 위한 것이 아니기 때문에 틀린 이름이다. 즉, 프로그램을 최적화 시킬 수는 없으나 개선하기를 희망하게 된다. 정상적으로 프로그램 최적화는 컴파일된 코드의 활용도를 개선하기 위한 컴파일 과정의 일부 분으로서 적용된 자동 기법을 언급한다. 여기서, 용어를 알고리즘 기술 개선으로 언급해 된다. 물론 과정은 또한 이용된 기계와 프로그래밍 환경에 오히려 의존케 되므로 특별한 기법이 아니라 여기서 일반적인 문제만을 고려케 된다.

이같은 활동의 형태는 프로그램이 여러 번 이용된 경우 혹은 큰 입력에 대해서만 그리고 구현을 개선하는 노력이 더 좋은 활용도로서 보상을 한다고 입증을 한다면 정당화된다. 알고리즘의 활용도를 개선하는 가장 좋은 방법은 더 좋은 구현으로 프로그램을 변형시키는 점진적인 과정을 통한 것이다. 5장에서의 재귀 제거 예제는 비록 활용도 개선이 그 경우에서의 목표는 아니지만 그런 과정의 예제이다.

알고리즘을 구현하는 첫 번째 단계는 가장 간단한 형태로서 알고리즘의 작업 버전을 개발하는 것이다. 이것은 정제와 개선에 대한 기준선이고 위에 언급한 대로, 가끔은 필요한 모든 것이다. 어떤 유용한 수학적 결과는 구현에 대해서 체크가 되어져야만 한다. 예를 들면, 분석은 실행시간이 $O(\log N)$ 이라고 말할 수 있는 것으로 보이나 실제 실행시간은 초단위로서 처리가 된다면, 그때 구현이나 분석에서 어떤 것인가가 잘못된 것이다. 그리고 둘 다는 더 조심스럽게 연구를 해야 한다.

다음 단계는 “내부 반복”을 인식하는 것이고 수반된 명령들의 수를 최소화하는 것이다. 아마도 내부 반복을 찾는 가장 쉬운 방법은 프로그램을 수행하고 어느 명령이 대부분 수행이 되는 가를 체크하는 것이다. 정상적으로 이것은 프로그램이 어디서 개선되어야 하는 가의 좋은 예시이다. 내부 반복에서 모든 명령은 재조사가 되어져야만 한다. 그것이 진정으로 필요한가? 같은 일을 수행하는데 더 좋은 방법이 있는가? 예를 들면, 그것은 항상 내부 반복에서 프로시저 부름을 제거하도록 해야 한다. 이것을 수행하는 다른 많은 “자동적인” 기법들 존재한다. 그리고 이것은 대부분이 표준 컴파일러를 구현하는 것이다. 궁극적으로 가장 좋은 활용도는 내부 반복을 기계나 어셈블리 언어로 이동시키므로써 해결된다. 그러나 이것은 항상 마지막 수단이다.

모든 “개선”은 실제적으로 활용도 이득으로 기인되는 것이 아니므로 각 단계에서 인식된 보관의 정도를 체크하는 것이 아주 중요하다. 더구나, 구현이 더욱 더 세련되면, 코드의 상세함에 그런 조심스러운 주의가 정당화되는지를 재 조사 하는 것이 현명하다. 과거에는 컴퓨터 시간이 비싸므로 해서 컴퓨터 사이클을 보관시키는데 프로그래머의 시간을 쓰는 것은 항상 정당화되어졌으나 최근에는 그런 경향이 변화가 되었다.

예를 들면, 5장에서 논의된 프리 오더 트리 운행 알고리즘을 고려해 보자. 실제적으로, 재귀 제거는 내부 반복에만 초점을 맞추기 때문에 이같은 알고리즘을 “최적화”시키는 첫 번째 단계이다. 주어진 비 재귀 버전은 내부 반복이 더 길고 두개 대신에 4개의(비 재귀적인) 프로시저 부름들(삽입, 삭제, 꼭참과 비었는 것)을 포함하기 때문에 많은 시스템에서(이것을 테스트해야만 할 것이다) 재귀 버전 보다 더 느리게 되는 것이다. 만약 스택 프로시저 부름은 스택을 직접적으로 접근하는 코드로서 대체가 된다면(항상 배열 구조를 이용해서), 이같은 프로그램은 재귀 버전보다 훨씬 더 빠르게 된다.(삽입 연산 중의 하나는 알고리즘에서 불필요하므로 표준 반복내의 반복 프로그램은 아마도 최적화된 버전에 대한 기본일 것이다) 그때, 내부 반복은 스택 포인터를 증가시키고, 스택 배열에서 포인터($t \rightarrow r$)를 분류하고, t 를 포인터($t \rightarrow r$)로 다시 세트시키고 그리고 z 와 비교를 하는 것이 수반된다. 많은 기계에서, 이것은 비록 전형적인 컴파일러가 2배나 그 이상으로 생성이 된다고 하지만 4가지 기계어 명령으로 구현된다. 이같은 프로그램은 아마도 너무 많은 작업이 없이도 간단하고 재귀적인 구현에서 보다 4배내지 5배정도로 빠르게 된다.

명백히, 여기서 논의된 문제들은 아주 시스템과 기계 의존적이다. 오히려 운영체제와 프로그래밍 환경의 상세한 지식 없이도 프로그램의 속도를 높이는 시도에 관계를 할 수가 없다. 프로그램의 최적화 버전은 변경하기가 오히려 위험하고 어렵게 되고 새로운 컴파일러나 새로운 운영체제(새로운 컴퓨터는 언급하지 않고)는 조심스럽게 최적화된 구현을 파괴시키게 된다. 특별한 주의나 변화가 가끔 일어나고 개선이 기대되어지는 곳에서 C++와 같은 진보적인 언어를 이용할 때 보증이 된다.

다른 한편, 높은 레벨에서 내부 반복에 주의를 기울이고 알고리즘에서 불필요한 것을 최소화시키기 위해서 구현에서 효율성에 초점을 두게 된다. 본 교재의 프로그램은 어떤 특별한 프로그래밍 환경하에 대해 직접적인 방법으로 더 좋은 개선을 하도록 코드화되고 수정된다.

알고리즘의 구현은 프로그램을 개발하고, 디버그하고 그리고 그것의 성질들을 배우고 그후 요구된 수준의 활용도에 도달이 될 때까지 구현을 정제하는 사이클 과정이다. 6장에서 논의

한 것과 같이, 수학적 분석은 항상 과정에서 도움을 준다. 첫째, 어느 알고리즘이 조심스런 구현에서 잘 수행이 되도록 후보자가 되는 지를 약속하는 것을 암시하고 둘째, 구현이 기대한대로 수행된 것을 검증하도록 도움을 주는 것이다. 어떤 경우에는 이같은 과정이 새로운 알고리즘이나 이전의 것에서 부수적인 개선을 제시하는 문제들에 관한 사실들로 된다.

알고리즘과 시스템

본 교재에서 알고리즘 구현은 넓은 범주의 여러 가지 큰 프로그램들, 운영체제와 응용 시스템들에서 볼 수가 있다. 우리의 의도는 알고리즘을 기술하고 주어진 구현으로서 실험을 통해 동적인 성질들에 초점을 맞추도록 하는 것이다. 어떤 응용에서 구현은 주어진 대로 아주 유용한 것이나 다른 응용에 대해서는 더 많은 작업이 요구된다.

첫째, 2장에서 언급한 대로 본 교재의 프로그램들은 C++와 다른 프로그래밍 환경에서 이용되는 더 고급화된 능력들의 잇점을 이용하기보다는 오히려 C++의 기본적인 특징만을 이용케 된다. 목적은 시스템 프로그래밍이나 프로그래밍 언어의 고급 특징들이 아니고 알고리즘을 공부하는 것에 있다. 알고리즘의 본질적인 특징은 거의 전체적인 언어에서 간단하고 직접적인 구현을 통해서 가장 좋은 것이 된다.

이용하는 프로그래밍의 형태는 간단한 변수 명과 적은 비교로서 다소 간결하지만 결과적으로 제어 구조가 나타난다. 실제 응용에서 이같은 프로그램을 이용하는 여러분은 특별한 응용에서 그것을 채용하므로써 신선하게 만든다. 더 “방어적인” 프로그래밍 형태는 실제 시스템을 구성하는데 있어 정당화된다. 즉, 프로그램들은 구현되어야 하고 그 결과 그것들은 쉽게 변경되고 다른 프로그래머에 의해서 빠르게 읽혀지고 이해가 되어져야 하고 시스템의 다른 부분들과 잘 인터페이스가 되어야 한다.

특히, 응용에 대해 요구되는 자료구조는 비록 고려되는 알고리즘들이 더 복잡한 자료 구조에 대해 적절하지만 본 교재에서 이용된 것들보다 오히려 더 많은 정보를 포함한다. 예를 들면, 응용이 전형적으로 큰 레코드들의 부분인 긴 문자 스트링으로 고려되기를 요구하는 반면에 정수나 짧은 문자 스트링을 포함한 파일을 통해서 검색한다고 말할 수 있다. 그러나 양쪽의 경우에서 이용되는 기본적인 방법들은 같다. 그런 경우에서 각 알고리즘의 현저한 특징과 그것들이 여러 가지 응용의 요구사항에 어떻게 관련되는지를 논하게 된다.

특정 알고리즘의 활용도를 개선하는데 관심을 둔 위의 대부분의 비교는 큰 시스템에서 활용도를 개선하는데 적용된다. 그러나 이같은 큰 척도로써, 시스템의 활용도를 개선하는 기법

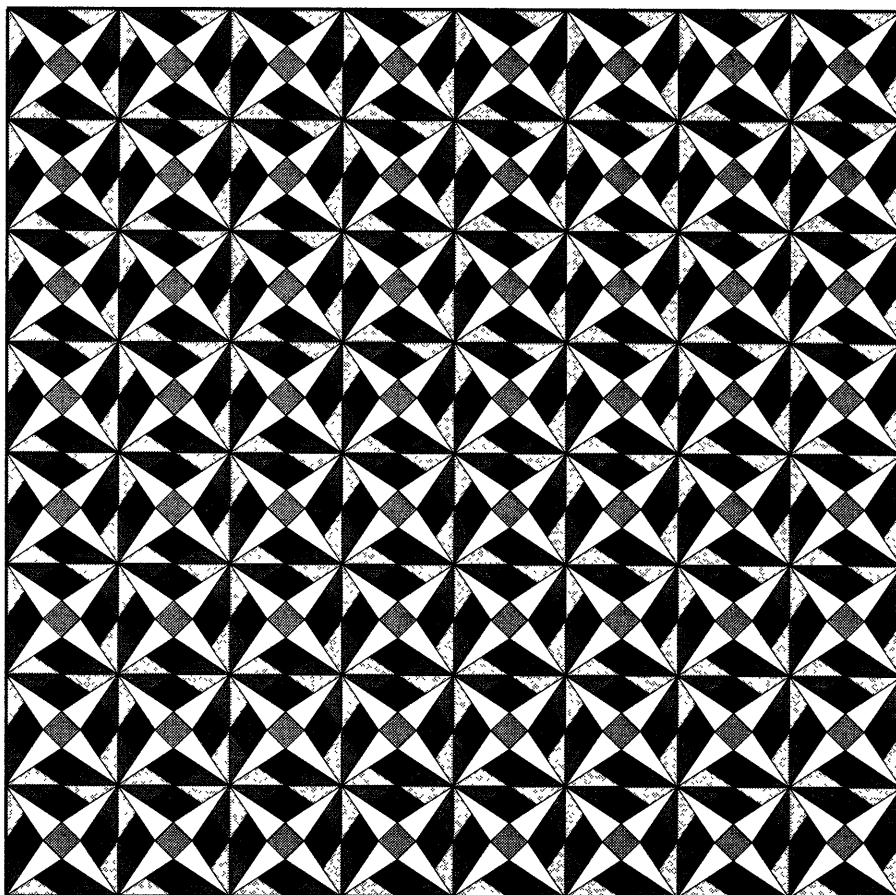
은 다른 것을 구현하는 모듈러로 한 알고리즘을 구현하는 모듈러를 대체시키는 것이다. 큰 시스템을 설정하는 기본적인 원칙은 그런 변화가 가능하게 하는 것이다. 전형적으로 시스템이 개발되어온 것과 같이, 더 자세한 지식은 특별한 모듈러에 대해 특별한 요구에 관해 얻어진다. 더 특별한 지식은 이같은 필요를 만족시키는 이용에 대해 가장 좋은 알고리즘을 더 조심스럽게 선택한 것이 가능하도록 하는 것이다. 그때, 위에 기술한 대로 그 알고리즘의 활용도를 개선하는데 중점을 둔다. 대다수의 시스템 코드는 몇 번만(또는 전혀 한 번도 아닌) 실행을 하는 경우는 확실하다. 즉, 시스템 생성의 주된 관심은 응집된 전체를 생성하는 것이다. 다른 한편, 시스템을 이용할 때, 그것의 많은 자원들은 본 교재에서 논의된 형태의 기본적인 문제를 해결하는데 전념을 하므로써, 시스템 생성자는 논의될 기본적인 알고리즘을 인식할 수 있도록 하는 것이 적절하다.

연습 문제

1. 100,000을 계산하는데 얼마나 오래 걸리는가? 프로그램 `j = 0; for (i = 1; i < 100000; i++) j ++;`는 여러분 자신의 환경하에서 얼마나 오래 걸리는가를 측정하고 측정을 테스트하기 위한 프로그램을 실행시켜라.
2. repeat와 while을 이용하여 위의 문제에 답하시오.
3. 적은 값으로 처리하므로 써, $N = 1,000,000$ 을 처리하기 위해서 3장에서 Eratosthenes의 추출 구현은 얼마나 오래 걸리는가를 측정하라.(만약 충분한 기억 장소가 제공된다면)
4. 10초 이내로 계산이 되도록 가장 큰 소수를 찾기 위해서 3장에서 구현한 Eratosthenes의 추출을 “최적화”하라.
5. 5 장에서(스택 연산에 대한 프로시저의 부름으로서) 프리 오더 트리 운행 알고리즘에서 재귀를 제거는 프로그램을 더 느리게 만든다는 테스트의 주장을 테스트하시오.
6. 5 장에서(그리고 스택 연산 inline을 구현시키고) 프리 오더 트리 운행 알고리즘에서 재귀를 제거는 프로그램을 더 빠르게 만든다는 테스트의 주장을 테스트하시오.
7. 5장에서 재귀 프리 오더 트리 운행 알고리즘에 대해 여러분의 국지 프로그래밍 환경에서 C++컴파일어로서 생성된 어셈블리 언어 프로그램을 조사하라.
8. 여러분 프로그래밍 환경에서 더 효율적인 것이 스택 삽입의 연결 리스트인지 혹은 배열 구현인지를 테스트하는 실험을 설계하시오.
9. 5장에서 주어진 자를 그리는 비 재귀 방법이나 재귀 방법중 어느 것이 더 효율적인가?
10. 프리 오더로 $2^n - 1$ 의 정(complete) 이진 트리를 운행할 때 5장에서 주어진 비 재귀 구현에 의해 얼마나 많은 스택 삽입이 이용되는지를 설명하라?

빈 면

정렬 알고리즘



빈 면

8 장

기초적인 정렬 방법들

정렬 알고리즘 영역의 첫 번째 것은 어떤 특별한 구조로 된 적은 파일들에 대해 몇 가지 “기본적인” 방법들에 대해 보는 것이다. 이같은 간단한 정렬 알고리즘을 어느 정도 상세하게 공부하는데는 여러 가지 이유가 있다. 첫째, 정렬 알고리즘에 대한 용어와 기본적인 메카니즘을 배우는데 상대적으로 별 어려움이 없는 방법이 제공되기 때문이다. 그래서 더 복잡한 알고리즘을 공부하는데 적절한 배경을 얻을 수가 있다. 둘째로, 정렬의 아주 많은 응용들에서 더 강력하고 일반적인 방법들에서 보다 이같이 간단한 방법들을 이용하는 것이 더 낫을 수가 있다. 마지막으로 간단한 방법들중의 어떤 것은 더 좋은 범용 방법들로 확장되거나 더 강력한 방법들의 효율성을 개선 시키는데 이용된다.

위에 언급한 대로, 상대적으로 간단한 알고리즘은 선택 방법에 따라 여러 가지 정렬 알고리즘이 존재 한다. 가끔 정렬 프로그램은 단지 한 번만 이용된다.(혹은 단지 몇 번만) 만약 정렬되어질 항목의 수가 너무 크지가 않다면,(5백개 이하의 요소들로 구성) 복잡한 방법을 구현하고 디버그하는 것보다 간단한 방법을 처리하는 것이 훨씬 더 효율적이다. 기초적인 방법은 적은 파일들에 대해 항상 적절하다.(50개 이하의 요소들) 매우 큰 수의 그런 파일들이 정렬 되어지지 않는다면, 복잡한 알고리즘은 적은 파일에 대해 정당화되는 것이 아니다. 상대적으로 정렬하기 쉬운 다른 형태의 파일은 똑같은 키들로 된 큰 수들이 포함된다. 간단한 방법들은 범용 방법들보다 훨씬 더 구조화된 파일들에서 수행을 할 수가 있다.

규칙으로서, 논의할 기초적인 방법들은 N 개의 무작위로 배열된 항목들을 정렬하기 위해서는 약 N^2 의 단계가 걸린다. 만약 N 이 충분히 적으면, 이것은 문제가 안되고, 만약 항목들이 무작위적으로 배열이 안되면, 방법들중 몇 가지는 더 복잡한 것들보다 훨씬 빠르게 된다. 그

러나 이같은 방법들은 많은 응용에 대한 선택중 실제적인 정렬 방법인 셸 정렬(Shellsort)의 예외로서 크고 무작위적으로 배열된 파일들에 이용될 수 없다는 것에 중점을 두어야만 한다.

게임의 규칙

어떤 특별한 알고리즘을 고려하기 전에, 정렬 알고리즘에 대한 몇 가지 일반적인 용어와 기본적인 가정들을 논하는 것이 유용하다. 키들이 포함된 레코드의 파일들을 정렬하는 방법을 고려해 보기로 하자. 레코드의 일부분인(가끔은 적은 부분) 이같은 키들을 정렬시키는데 이용된다. 정렬 방법의 목적은 레코드를 재배열하고 그것의 키들을 어떤 잘 정의된 순서 규칙에 따라 순서화 된다.(항상 수치이거나 알파벳 순서)

만약 정렬된 파일이 기억 장소에 적합하면,(혹은 내용에서 C++배열에 적합한 경우) 그때 정렬 알고리즘을 내부(internal) 정렬이라고 부른다. 테이프나 디스크에서 파일을 정렬시키는 것을 외부(external) 정렬이라고 부른다. 두 가지의 주된 차이는 어떤 레코드는 내부 정렬로 쉽게 접근이 가능한 반면, 외부 정렬은 레코드들을 순차적으로 적어도 큰 블록 단위로서 접근이 되어져야만 한다. 13장에서 외부 정렬에 대해서 보기로 하자. 그러나 고려할 대부분의 알고리즘은 내부 정렬들이다.

평상시와 같이, 관심을 둔 주된 활용도 파라미터는 정렬 알고리즘의 실행 시간에 있다. 더 고급화 된 방법들이 N 개 항목들을 정렬하는데 $N \log N$ 시간에 비례하는 반면에 본 장에서 보게 될 첫 번째 4가지 방법은 N 개 항목들을 정렬하는데 N^2 시간에 비례를 한다.(정렬 알고리즘이 키들 사이에 $N \log N$ 비교 이하로서 이용하는 것이 없다는 것을 제시한다) 간단한 방법들을 조사한 후, 실행시간 $N^{3/2}$ 나 그 이하로서 처리되는 더 고급화된 방법을 보게 되고 N 에 비례하는 전체 시간을 얻기 위해서 키들의 디지털 성질을 이용하는 방법들을 보게 된다.

정렬 알고리즘에서 이용된 특별한 기억 장소의 양은 고려될 두 번째 중요한 요소가 된다. 기본적으로 방법들은 세 가지 형태로 구분된다. 즉, 장소내에 정렬을 하고 적은 스택이나 표들에 대해 아마도 예외 없이 특정한 기억 장소를 이용하지 않는 것들, 연결 리스트 표현을 이용한 것과 리스트 포인터에 대해 기억 장소의 N 개 특별한 단어들을 이용하기 위한 것들 그리고 정렬될 배열의 또 다른 복사를 유지시키기 위해 충분히 특별한 기억 장소를 필요로 한다.

가끔은 실제로 중요한 정렬 알고리즘의 특징들은 안전성(stability)이다. 정렬 알고리즘은 파일에서 똑같은 키들의 상대적인 순서를 유지하는 경우에 안전(stable)하다고 한다. 예를 들면, 만약 알파벳으로 된 class 리스트가 등급에 의해서 정렬이 되면, 안정된 방법은 같은 등급을 지닌 학생들이 여전히 같은 알파벳 순서로서 된 리스트를 생성하는 것이나, 비 안정된 방법은 원래 알파벳 순서의 흔적 없이 된 리스트를 생성하는 것이다. 대부분의 간단한 방법들은 안전성이나 대부분 잘 알려진 복잡한 알고리즘들은 그렇지 않다. 만약 안정성이 중요한 경우, 적은 인덱스를 정렬하기 전에 각 키에 부가시키거나 어떤 다른 방법으로 정렬키를 길이화함으로써 강요된다. 안정성을 인정하는 것은 쉽다. 즉, 불신으로 된 불안정성의 불편한 효과에 반응하게 된다. 실제로 몇 안되는 방법들은 의미 있는 특별한 시간이나 공간을 사용함이 없이도 안정성을 유지케 된다.

다음 프로그램은 이용된 일반적인 규약을 제시하고자 하는 의도가 있다. N 개 수를 읽고 그것을 정렬하기 위해서 서브루틴을 부르는 주 프로그램으로 구성된다. 이 예에서, 루틴은 읽혀진 단지 세 가지 수들을 정렬하기만 한다. 즉, 어떤 정렬 프로그램은 sort3에 대해 대체가 되어지고 이같은 “장치” 프로그램에 의해서 불려진다.

```
inline void swap(itemType a[], int i, int j)
{ itemType t = a[i]; a[i] = a[j]; a[j] = t; }
sort3(itemType a[], int N)
{
    if ( a[1] > a[2] ) swap(a, 1, 2);
    if ( a[1] > a[3] ) swap(a, 1, 3);
    if ( a[2] > a[3] ) swap(a, 2, 3);
}
const int maxN = 100;
main()
{
    int N, i; itemType v, a[maxN+1];
    N = 0; while (cin >> v) a[++N] = v;
    a[0] = 0;
    sort3(a, N);
    for( i = 1; i <= N; i++) cout << a[i] << ' ';
    cout << '\n';
}
```

3장에서 itemType과 같이 정렬된 항목들의 형태는 미 기술된채로 남겨두고서 알고리즘 문제에 초점을 맞추고, 정수형의 배열은 숫자적인 순서로서, 문자형의 배열은 알파벳 순서로 단순히 정렬하는 것으로 알고리즘을 간주한다. C++에서 큰 키들이나 레코드들을 수반한 실제적인 응용에서 이용을 위한 그런 알고리즘을 채택하도록 템플레이트나 typedef를 이용하는 것은 간단하다.

정상적으로 정렬 프로그램들은 두 가지 방법들 중의 하나로서 레코드를 접근하게 된다. 즉, 키들을 비교를 위해서 접근시키거나 혹은 전체 레코드를 이동시키기 위해서 접근시키는 것중의 하나이다. 공부할 대부분의 알고리즘은 임의의 레코드상에서 이같은 두 가지 연산에 의해서 고쳐지게 된다. 만약 정렬될 레코드들이 큰 경우, “간접 정렬”을 수행하므로써 그것들을 썬는 것을 피하는 것이 현명하다. 즉, 레코드 그 자체는 반드시 재배열 될 필요는 없으나 오히려 포인터들(또는 인덱스들)의 배열은 재배열이 되어져서 첫 번째 포인터는 가장 적은 레코드를 지칭하게 하는등이다. 키들은(만약 키들이 큰 경우) 레코드로서 혹은(만약 키들이 적은 경우) 포인터로서 유지시키면 된다. 필요한 경우, 레코드들은 본 장의 나중에 기술하는 대로 정렬된 뒤에 재 배열을 해야한다.

프로시저 swap는 “교환” 연산을 구현하고, 교환은 많은 정렬 프로그램에서 기본이 되고 가끔은 내부 반복안에서 이루어지기 때문에 inline이 된다. 실제로 프로그램에서 파일에 대해 훨씬 더 제약적인 접근을 이용하게 된다. “두 개 레코드를 비교하고 가장 적은 키를 먼저 놓는 것이 필요한 경우 그것들을 교환”하는 형태의 명령에는 세 가지가 있다. 그런 명령들에 제약되는 프로그램들은 그것들이 하드웨어 구현에 대해 적합하기 때문에 관심이 된다. 이 문제에 대해서 자세한 내용은 40장에서 보도록 하자.

C++의 어떤 기법들은 응용에서 유용하게 고려되는 여러 가지 알고리즘들을 만드는 장점을 추출하는 몇 가지 표식이 여전히 주어지는 반면에 정렬들이 어떻게 “패케지화” 되는지의 일반적인 문제에 대해 다루는 것을 피하고자 한다. 예를 들면, 알고리즘들을 더 많은 형태의 키들에 대해서 유용하도록 하기 위해 템플레이트나 typedef를 이용하므로써 이미 문제를 다루었다. 또 다른 예제는 다음과 같다. C++에서 파라미터로서 정렬 루틴에 배열을 전달시키는 것이 합당하나 어떤 다른 언어에서는 그렇지 않게 된다. 대신 광의 배열상에서 프로그램이 수행이 될 수 있는가? 정렬을 포함한 어떤 배열상에서 수행될 연산을 확장시키는 class의 일부분으로 정렬 루틴이 되는가? 어떤 운영체제는 위에서 제시된 하나와 것과 입력과 출력 사이에 “필터(filter)”로서 서비스되는 간단한 프로그램들을 함께 놓는 것을 쉽도록 한다. 다

른 한편, class들과 필터들과 같은 메카니즘은 진정으로 많은 응용에서 요구되는 것이 아니고 응용 코드에 둘러쌓인 정렬 코드의 적은 부분이 가장 좋은 것이다. 명백히 이같은 비교들은 비록 정렬 알고리즘의 연구가 대부분 중요한 문제들을 나타내지만 조사될 다른 알고리즘의 대부분에 적용된다.

또한 비록 응용에서 그렇게 수행함에 신중하지만, 많은 “오류 체크(error-checking)” 코드를 포함하지는 않는다. 예를 들면, 장치 루틴은 N 이 $\max N$ 을 초과 하지 않음을 체크하게 된다.(sort3은 $N=3$ 인가를 체크한다) 또 다른 유용한 체크는 장치로 하여금 sort3을 부른 뒤에 배열이 정렬될것이 확실한가이다. 이것은 정렬 프로그램이 처리되는 것을 보증하는 것은 아니다.(왜?) 그러나 그것은 노출된 오류들에 도움을 준다.

몇몇 프로그램들은 몇 가지 다른 광의 변수들을 이용하게 된다. 명백하지 않은 선언들은 프로그램 코드로서 포함된다. 또한 알고리즘의 몇 가지에 의해서, 특별한 키들이 이용되도록 유지시키기 위해서 $a[0]$ (그리고 때때로 $a[N+1]$)을 보관한다. 예제들에 대해서 숫자라기 보다는 오히려 알파벳에서 문자들을 가끔 이용케 된다. 즉, 이같은 것들은 정수형과 문자형의 사이에 C++의 표준 형 변환 함수를 이용하는 명백한 방법으로 처리된다.

선택 정렬(Selection Sort)

가장 간단한 정렬 알고리즘중의 하나는 다음과 같이 처리된다. 즉, 먼저 배열에서 가장 적은 요소를 찾아서 그것을 첫 번째 위치에 있는 요소와 교환을 한다. 그후 두 번째로 가장 적은 요소를 찾고, 두 번째 위치에 있는 요소와 교환을 하고 그리고 전체 배열이 정렬될 때까지 이같은 방법으로 계속된다. 그림 8.1에 제시된 것과 같이 반복해서 남아 있는 가장 적은 요소를 “선택”함으로서 처리가 되기 때문에 이같은 방법을 선택 정렬이라고 부른다. 첫 번째 과정은 왼쪽에 A보다 더 적은 것이 배열에서 없기 때문에 효과가 없다. 두 번째 과정에서는 두 번째 A가 나머지 요소들 가운데 가장 적은 것이므로 두 번째 위치의 S와 교환한다. 그리고 첫 번째 E는 세 번째 과정에서 세 번째 위치의 O와 교환하고, 그리고 두 번째 E는 4번째 과정으로서 4번째 위치의 값 R과 교환하는 등이다.

다음 프로그램은 이같은 과정의 구현이다. 1에서 $N-1$ 에 이르는 각 i 에 대해, 그것은 $a[1], \dots, a[N]$ 에서 최소 요소와 $a[i]$ 를 교환케 된다.

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
	S	O	R	T	I	N	G	E	X	A	M	P	L	E
		O	R	T	I	N	G	E	X	S	M	P	L	E
			R	T	I	N	G	O	X	S	M	P	L	E
				T	I	N	G	O	X	S	M	P	L	R
					I	N	T	O	X	S	M	P	L	R
						N	T	O	X	S	M	P	L	R
							T	O	X	S	M	P	N	R
								O	X	S	T	P	N	R
									X	S	T	P	O	R
										S	T	P	X	R
											T	S	X	R
												S	X	T
													X	T
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

그림 8.1 선택 정렬

```

void selection(itemType a[], int N)
{
    int i, j, min;
    for ( i = 1; i < N; i ++)
    {
        min = i;
        for (j = i + 1; j <= N; j++)
            if (a[j] < a[min] ) min = j;
        swap(a, min, i);
    }
}

```

인덱스 i 는 파일의 왼쪽에서 오른쪽으로 처리함으로 써, 인덱스의 왼쪽에 있는 요소들은 배열에서 마지막 위치내에 존재하므로,(그리고 다시는 이것을 처리하지 않는다) 배열이 인덱스가 오른쪽 끝에 도달이 될 때 완전히 정렬된다.

이것은 정렬 방법들 가운데 가장 간단한 것이고 적은 파일에서 매우 잘 처리가 된다. “내부 반복”은 더 간단하게 할 수 없는 비교 $a[j] < a[\text{min}]$ 이다.(여기에 j 를 증가시키고 그것이 N 을 초과하지 않는 것인지를 체크하는 것이 필요한 코드를 더한 것이다) 아래에서는 이같은 명령들이 실행되는 것을 여러번 논의를 할 것이다.

더구나, 선택 정렬은 실제로 아주 중요한 응용이다. 각 항목은 실제로 기껏해야 한 번만 이동되기 때문에 선택 정렬은 매우 큰 레코드와 적은 키들로된 파일을 정렬하는 선택 방법이 된다. 이것은 아래 자세히 기술된다.

삽입 정렬(Insertion Sort)

알고리즘은 선택 정렬과 같이 간단하나 아마도 더 적응성이 있는 것이 삽입 정렬이다. 이것은 사람들이 브리지 핸드(bridge hands)를 정렬하는데 가끔 이용된다. 즉, 한 번에 한 개씩 요소를 고려해서 각각을 이미 고려된 것들 가운데 적절한 위치에 삽입시킨다.(그것들은 정렬된 형태로 유지된다) 고려된 요소는 그림 8.2에 제시된 것과 같이 더 큰 요소를 한자리 이동시키고 그 요소를 비어있는 위치에 삽입시키므로써 이루어진다. 두 번째 위치에서 S 는 A 보다 더 큰 것이므로 이동이 되지 않는다. 세 번째 위치에서 O 가 만날 때, 그것은 S 와 교환을 해서 정렬된 순서 $A O S$ 로 되는 등등으로 이루어진다.

이같은 과정은 다음 프로그램으로 구현된다. 2에서 N에 이르는 각 i에 대해, 요소들 $a[1], \dots, a[i]$ 에서 $a[i]$ 를 $a[1], \dots, a[i-1]$ 에서 요소들의 정렬된 리스트 가운데 해당 위치에 놓으므로써 정렬된다.

```
void insertion(itemType a[], int N)
{
    int i, j; itemType v;
    for ( i = 2; i <= N; i++)
    {
        v = a[i]; j = i;
        while ( a[j-1] > v )
            { a[j] = a[j-1]; j--; }
        a[j] = v;
    }
}
```

선택 정렬에서 처럼, 인덱스 i의 왼쪽에 있는 요소들은 정렬동안에 정렬된 순서로서 존재를 하나 나중에 더 적은 요소들에 대한 여지를 남겨 주기 위해서, 이동이 되는 것과 같이 그것들이 마지막 위치에는 존재를 하지 않게 된다. 그러나 배열은 인덱스가 오른쪽 끝에 도달될 때 완전히 정렬된다.

고려해야할 중요한 내용이 하나 더 있다. 즉, 프로시저 insertion은 대부분의 입력들에 대해서 처리가 되지 않는 것이다! while은 v가 배열에서 가장 적은 요소일 때 배열의 왼쪽 끝을 지나 처리된다. 이것을 고정시키기 위해서, 적어도 배열에서 가장 적은 요소와 같은 정도로 적은 것으로 만들므로써, $a[0]$ 에 “표지(sentinel)” 키를 놓게 된다. 표지는 내부 반복내에서 거의 항상 성공적인 테스트(이 경우는 $j > 1$)를 포함함을 피하기 위해서 이같은 상황에서 공통적으로 이용된다.

같은 이유로서 표지를 이용하는 경우가 불편한 경우가 있다면,(예를 들면, 아마도 가장 적은 키를 쉽게 정의의 하지 못한다) 그때 테스트 $while\ j>1 \ \&\&\ a[j-1] > v$ 가 이용된다. $j=1$ 은 단지 드물게 발생되기 때문에 매력이 없고 그래서 내부 반복내에서 그것에 대해 왜 가끔 테스트를 하게 되는가? j 가 1과 같을 때, 논리식이 C++에서 계산이 되는 방법 때문에 위의 테스트는 $a[j-1]$ 을 접근하지 못한다. 어떤 다른 언어들에서 배열밖에 존재하는 것은 그런 경우 접근이 된다. C++에서 이같은 상황을 처리하는 또 다른 방법은 while 반복의 외부에

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
	S	O												
	O	S	R											
		R	S	T										
				T	I									
	I	O	R	S	T	N								
		N	O	R	S	T	G							
	G	I	N	O	R	S	T	E						
	E	G	I	N	O	R	S	T	X					
									X	A				
	A	E	G	I	N	O	R	S	T	X	M			
					M	N	O	R	S	T	X	P		
								P	R	S	T	X	L	
					L	M	N	O	P	R	S	T	X	E
			E	G	I	L	M	N	O	P	R	S	T	X
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

그림 8.2 삽입정렬

goto나 break를 이용하는 것이다.(어떤 프로그래머는 반복 종결을 확실히 하도록 하기 위해 반복내에 행동을 수행함으로써 goto 명령을 피하도록 어떤 길이로 가도록 하는 것이다. 이 경우에서, 그런 해는 거의 정당화되질 않는다. 그 이유는 프로그램을 더 명확하게 하지 못하고 드문 사건에 대해 보호하기 위해 반복을 통해서 매번 부가적인 내용이 첨가된다)

버블 정렬(Bubble Sort)

가끔 입문적 형태인 계층에서 배우는 기초적인 정렬 알고리즘이 버블 정렬이다. 즉, 파일을 통해 유지하고, 필요한 경우 인접된 요소들을 교환시키고 어떤 과정에서 교환이 없을 때 파일은 정렬이 된다. 이 방법의 구현은 아래와 같다.

```
void bubble(itemType a[], int N)
{
    int i, j;
    for ( i = N; i >= 1; i--)
        for ( j = 2; j <= i; j++)
            if ( a[j-1] > a[j] ) swap( a, j-1, j);
}
```

이같은 것이 처리되도록 확신시키기 위해서는 순간의 반영을 취한다. 그렇게 수행하도록 하기 위해서, 최대 요소가 첫 번째 과정동안 만나게 될 때 마다, 배열 오른쪽 끝의 위치에 도달될 때까지 요소들의 각각을 그것의 오른쪽과 교환한다. 두 번째 과정에서, 두 번째로 큰 요소가 위치에 삽입되는 등등이다. 이와 같이 버블 정렬은 각 요소를 위치에 존재시키도록 하는데 훨씬 더 많은 노력이 필요하지만 선택 정렬의 한 형태로서 처리된다.

기초적인 정렬들의 활용도 특징

선택 정렬, 삽입 정렬과 버블 정렬의 처리 특징에서 직접적인 예시는 그림 8.3, 8.4와 8.5에 주어져 있다. 이같은 다이어그램들은 외부 반복이 $N/4$, $N/2$ 와 $3N/4$ 번 반복된 후에(입력으로서 정수 1에서 N 에 이르는 무작위 순열로서 시작한 것이다) 알고리즘의 각각에 대해 배열 a 의 내용을 제시한 것이다. 다이어그램에서, 사각형은 $a[i] = j$ 에 대한 위치 (i, j) 에 놓

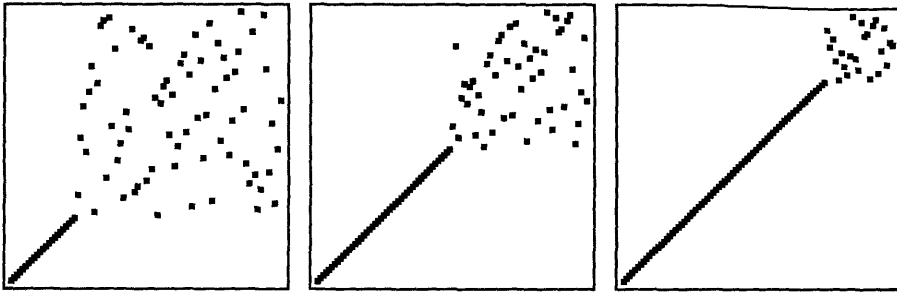


그림 8.3 무작위 순열의 선택 정렬

여진 것이다. 이와 같이 비 순서화된 배열은 사각형의 무작위 디스플레이이다. 즉, 정렬된 배열에서, 각 사각형은 그것의 왼쪽에 나타난 것 위에 표시된다. 다이어그램의 명확성에 대해, 정렬될 때 주 대각선을 따라 일렬로 되어진 사각형인 순열들(정수 1에서 N 에 이르는 것의 재 배열)을 제시한다. 다이어그램들은 다른 방법들이 이같은 목표를 통해서 어떻게 진전되는가를 제시한다.

그림 8.3은 마지막 위치에 요소들을 놓으므로써 선택 정렬이 어떻게 왼쪽에서 오른쪽으로 이동되는가를 제시한다. 이같은 다이어그램에서 명백하지 않은 것은 선택 정렬이 배열의 “비 정렬된” 부분에서 최소 요소를 찾으려고 하는데 대부분의 시간을 소비한다는 사실이다.

그림 8.4는 새로이 만난 요소들을 위치에 삽입시키므로써, 삽입 정렬이 어떻게 왼쪽에서 오른쪽으로 이동되는가를 제시한다. 배열의 왼쪽 부분은 계속해서 변경된다.

그림 8.5는 선택 정렬과 버블 정렬 사이에 동질성을 제시한 것이다. 버블 정렬은 각 단계에서 남아 있는 요소 가운데 최대치를 “선택”한다. 그러나 배열의 “비 순서화된” 부분에 어떤 순서를 알려주기 위한 노력이 소비된다.

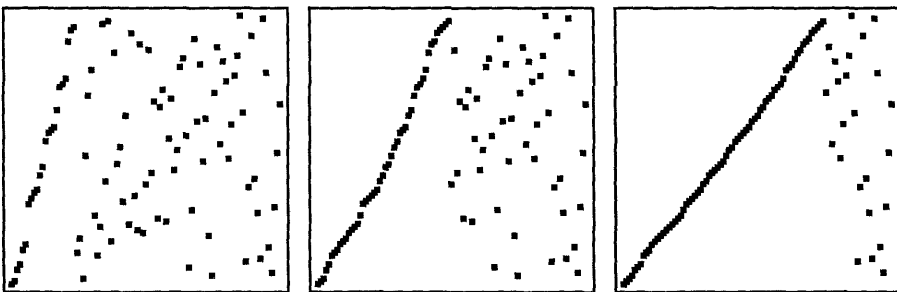


그림 8.4 무작위 순열의 삽입 정렬

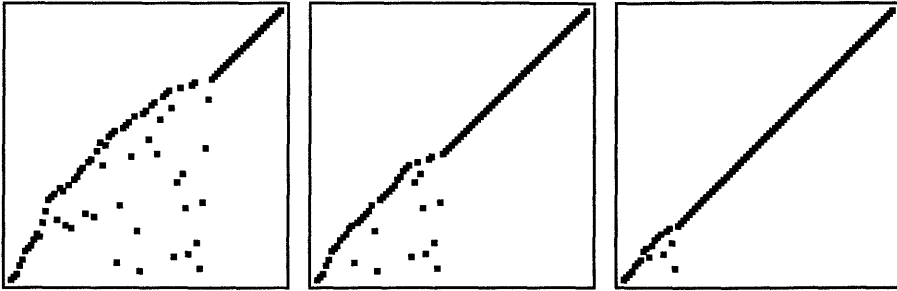


그림 8.5 무작위 순열의 버블 정렬

모든 다른 방법들은 최악의 경우와 평균적인 경우 둘 다에서 2차원적이고 특별한 기억 장소를 필요로 하지 않는다. 이와 같이 그것들 사이에 비교는 입력의 특별한 특징이나 내부 반복의 길이에 의존한다.

성질 8.1 선택 정렬은 약 $N^2/2$ 번 비교와 N 번 교환된다.

이같은 성질은 그림 8.1을 조사함으로써 쉽게 보게 되고, 이것은 문자가 각 비교에 대응되는 N 행 N 열로 된 표이다. 그러나 이것은 대각선 위에 있는 요소들의 약 반이 된다. 대각선에 있는 $N - 1$ 개 요소들(마지막은 아니고) 각각은 교환에 대응된다. 더 자세히 보면 다음과 같다. 1에서 $N - 1$ 에 이르는 각 i 에 대해, 한 번의 교환과 $N - i$ 번의 비교가 있으므로 해서 전체가 $N - 1$ 번의 교환과 $(N - 1) + (N - 2) + \dots + 2 + 1 = N(N - 1)/2$ 번의 비교가 된다. 이같은 관찰은 입력 데이터가 무엇이든지간에 유지된다. 즉, 입력에 의존되는 선택 정렬의 유일한 부분은 min이 갱신되는 빈도수이다. 최악의 경우, 이것 또한 2차적이 것이나 그러나 평균의 경우에는 이같은 양이 $O(N \log N)$ 으로에서만 존재하므로 해서, 입력에 아주 덜 영향을 받는것으로 선택 정렬의 실행 시간을 기대케 된다. □

성질 8.2 삽입 정렬은 평균의 경우 약 $N^2/4$ 번 비교와 $N^2/8$ 번 교환이 되고, 최악의 경우는 평균적인 경우의 두배가 된다.

위에서 구현한 대로, 비교 수와 “반 교환”(이동)의 수는 같다. 논의된대로, 이같은 양은 알고리즘 연산의 상세함을 제공해 주는 N 행 N 열 다이어그램인 그림 8.2에서와 같이 시각화하는 것이 쉽다. 여기서, 대각선 아래의 요소들은 최악의 경우에서 그것들 모두를 계수화하는 것이다. 무작위 입력에 대해, 각 요소은 평균의 경우 약 반을 뒤로 옮기는 것으로 되고 그래서 대각선 아래의 요소들 반은 계수가 된다.(이같은 인수들을 더 자세하게 하는 것은 어렵지 않다) □

성질 8.3 버블 정렬은 평균의 경우와 최악의 경우 약 $N^2/2$ 번 비교와 $N^2/2$ 번 교환된다.

최악의 경우에서(역순으로된 파일에서), i 번째 버블 정렬 과정은 $N - i$ 번 비교와 교환을 요구하므로 서, 증명은 선택 정렬에 대한 것과 같다. 그러나 버블 정렬의 실행시간은 입력에 의존된다. 예를 들면, 파일이 이미 순서적으로 배열된 경우,(삽입 정렬 또한 이경우에서 빠르다) 단지 한 번의 과정이 요구된다. 비록 이같은 분석이 오히려 더 어렵다 할지라도 나타난바와 같이 평균 경우의 활용도가 최악의 경우보다 낫다는 것은 의미가 없다. □

성질 8.4 삽입 정렬은 “거의 정렬된” 파일들에 대해 선형적이다.

비록 “거의 정렬된” 파일의 개념이 오히려 자세하지 않지만, 삽입 정렬은 실제로 가끔 발생이 되는 비 무작위 파일의 몇 가지 형태에 대해 잘 처리된다. 범용 정렬들은 그런 응용에 대해서 공통적으로 잘못 이해된다. 실제로, 삽입 정렬은 파일에서 존재한 순서의 장점을 이용케 된다.

예를 들어서 이미 정렬이 된 파일상에서 삽입 정렬의 처리를 고려해 보자. 각 요소는 파일에서 적절한 위치에 존재하도록 결정되고 전체 실행시간이 선형적으로 된다. 버블 정렬에서도 같으나 선택 정렬은 2차원이다. 비록 파일이 완전히 정렬이 되어있지 않더라도, 삽입 정렬은 그것의 실행시간이 파일에 존재하는 순서상에 아주 많이 의존되기 때문에 삽입 정렬은 아주 유용하다. 실행시간은 도치(inversions)의 수에 의존케 된다. 그 이유는 각 요소가 그것의 큰 왼쪽에 대한 요소들의 수를 계산하는 것이다. 이것은 삽입 정렬동안 파일에 삽입될 때 이동 되어지는 요소들의 거리이다. 그 속에 어떤 순서를 지니는 파일은 임의적으로 끊어 모으는 것 보다 그 속에 있는 몇 안되는 도치를 지는 것이다.

가령 더 큰 정렬된 파일을 생성하기 위해서, 정렬된 파일에 몇몇 요소들을 부가시킨다고 하자. 그렇게 하는 한 방법은 파일의 끝에 새로운 요소를 첨가시키고, 그 후에 정렬 알고리즘을 부른다. 명백히, 도치의 수는 그런 파일내에서는 적다. 즉, 장소를 벗어난 일정한 요소들의 수만으로 된 파일은 도치의 선형적인 수만을 지닌다. 또 다른 예로서 각 요소가 그것의 마지막 위치에서 어떤 일정한 거리에서만 존재한다는 것이다. 파일은 어떤 고급 정렬 방법들의 초기 단계에서 생성된다. 즉, 어떤 위치에서 삽입 정렬로 전환하는 것이 더 가치가 있다.

그런 파일에 대해, 삽입 정렬은 다음 몇 장에서 기술된 훨씬 더 복잡한 방법들 보다 성능이 더 우수하다. □

방법들을 더 많이 비교하기 위해서, 레코드와 키들의 크기에 차례로 의존하는 요인인 비교와 교환의 비용을 분석할 필요가 있다. 예를 들면, 만약 위의 구현에서 처럼 레코드들이 한개 워드로된 키들이면, 교환(두개 배열 접근들)은 비교보다 약 두 배로 더 비싸게 된다. 그런 상황에서, 선택과 삽입 정렬의 실행시간은 상당히 비교적이나 버블 정렬은 두 배정도 느리다. (사실, 버블 정렬은 거의 어떤 환경하에서 삽입 정렬정도로 두 배가 느리는 것이다!) 그러나 만약 레코드들이 키들에 대한 비교에서 큰 경우, 그때 선택 정렬이 가장 좋다.

성질 8.5 선택 정렬은 큰 레코드와 적은 키들로 된 파일들에 대해 선형적이다.

비교의 비용이 1시간 단위이고, 교환의 비용이 M 시간 단위라고 가정하자.(예를 들면, 이것은 M -워드 레코드들과 1-워드 키들로 된 경우이어야만 한다) 그때 선택 정렬은 크기 NM 의 파일을 정렬하기 위해서, 비교에 대해 약 N^2 시간이 그리고 교환에 대해 NM 시간이 걸린다. 만약 $N = O(M)$ 이면, 이것은 데이터의 양내에서 선형이다. □

큰 레코드로된 파일을 정렬

알고리즘을 파일에 대해 간접적으로 처리하고(인덱스의 배열을 이용하고) 그 후에 재배열을 수행하도록 함으로 해서, 어떤 정렬 방법은 단지 N 개의 완전한 레코드들의 “교환”만을 이용하기 위해서 어떤 것을 배열하는 것이 실제적으로 가능하다.(그리고 바라는 바이다)

만약 배열 $a[1], \dots, a[N]$ 이 큰 레코드들로 구성되면, 비교에 대해서만 원래 배열을 접근하는데 “인덱스 배열” $p[1], \dots, p[N]$ 을 처리하는 것이 낫다. 만약 처음에 $p[i] = i$ 로 정의를 하면, 그때 위의 알고리즘들은(그리고 다음 장에 나타나는 알고리즘들 모두) 비교에서 $a[i]$ 를 이용할 때 $a[i]$ 보다는 $a[p[i]]$ 를 언급하기 위한 것이고 데이터 이동을 할 때 a 보다는 p 를 선택하기 위해서만 변경을 필요로 한다. 이것은 인덱스 배열을 “정렬”하는 알고리즘을 생성하고 그 결과 $p[1]$ 은 a 내에서 가장 적은 요소이고 $p[2]$ 는 a 내에서 두 번째로 적은 요소 등등으로 되고 큰 레코드주위를 과도하게 이동시키는 것은 이같은 방법으로 처리되도록 변경되어야만 한다.

```
void insertion(itemType a[], int p[], int N)
{
    int i, j; itemType v;
```

```

for ( i = 0; i <= N; i++ ) p[i] = in;
for ( i = 2; i <= N; i++ )
{
    v = p[i]; j = i;
    while ( a[p[j-1]] > a[v] )
        { p[j] = p[j-1]; j-- }
    p[j] = v;
}
}

```

이 프로그램에서, 배열 *a*는 두 개 레코드의 키들을 비교하는데만 접근된다. 이와 같이, 큰 레코드의 적은 필드만을 접근하는 비교를 변경시키거나 비교를 더 복잡한 프로시저로 만들도록 함으로써 매우 큰 레코드로 된 파일들을 처리하는데 쉽게 변경된다. 그림 8.6은 이같은 과정이 배열 요소들로 하여금 정렬된 리스트를 정의하기 위해서 접근이 되는 순서를 기술하는 순열을 생성하는 방법을 제시한다. 많은 응용에 대해, 이것은 충분하다.(데이터는 전혀 이동이 될 필요가 없다) 예를 들면, 정렬 그 자체로서 인덱스 배열을 통해 그것을 간접적으로 언급하므로써 단순히 정렬된 순서로서 데이터를 출력케 된다.

그러나 그림 8.6의 밑에서 처럼 만약 데이터가 실제로 재배열이 되는 경우는 무엇인가? 만약 배열에서 또 다른 복사를 위한 충분한 공간의 기억장소가 있다면, 이것은 간단하나 파일의 또 다른 복사에 대한 충분한 공간이 있지 않을 때는 더 정상적인 상황은 어떤 것인가?

Before Sort

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

After Sort

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

After Permute

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A	E	E	G	I	L	M	N	O	P	R	S	T	X
p[k]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

그림 8.6 “정렬된” 배열을 재 배열

예제에서, 첫 번째 A는 $p[1] = 1$ 로서 적절한 위치에 존재케 된다. 그래서 어떤 것도 그것과 수행될 필요가 없다. 수행할 첫 번째 것은 다음번째로 가장 적은 키(인덱스 $p[2]$ 로 된 것)로 된 레코드를 파일에서 두 번째 위치에 삽입시키는 것이다. 그것을 수행하기 전에, 그 위치내에 있는 것 말하자면 t내에 존재하는 레코드를 보관할 필요가 있다. 지금 이동후에, 위치 $p[2]$ 에 있는 파일내의 “홀(hole)”이 있는 것으로 고려된다. 그러나 $p[p[2]]$ 에 위치한 레코드는 결국에는 꼭 채워지게 된다는 것을 알게 된다. 이 방법을 계속하면, 결국에는 t내에 유지되고 있는 두 번째 위치에서 원래의 항목을 필요로 하는 곳인 점으로 도달되게 된다. 예제에서 이같은 과정은 다음 연속된 할당문으로 된다. 즉, $t=a[2]; a[2]=a[11]; a[11]=a[13]; a[13]=a[2]; a[2]=t$;이다. 이같은 할당문은 키 A, P 와 S로 된 레코드를 $p[2]=2, p[11]=11$ 그리고 $p[13]=13$ 으로 세팅시키므로써 표시된 파일에서 적절한 위치에 놓게 된다.($p[i]=i$ 로 된 어떤 요소는 그 위치 내에 있고 다시는 그것을 다루지 않게 된다) 지금 과정은 그 위치내에 없는 다음 요소에 대해서 다시 처리가 되는 등이고 이것은 궁극적으로 다음 코드에서 처럼 각 레코드를 단지 한 번만 이동시키므로써 전체 파일을 재배열한다.

```
void institu(itemType a[], int p[], int N)
{
    int i, j, k; itemType t;
    for ( i = 1; i <= N; i++)
        if ( p[i] != i )
        {
            t = a[i]; k = i;
            do
            {
                j = k; a[j] = a[p[j]];
                k = p[j]; p[j] = j;
            }
            while ( k != i );
            a[j] = t;
        }
}
```

특별한 응용에 대한 이같은 기법의 생존력은 정렬되어질 파일에서 레코드들과 키들의 상대적인 크기에 의존케 된다. 인덱스 배열에 대해 요구되는 특별한 공간과 간접 비교에 대해 요구되는 특별한 시간 때문에 확실히 적은 레코드들로 구성된 파일에 대해 그런 문제는 지니지 않게 된다. 그러나 큰 레코드들로 구성된 파일에 대해, 간접 정렬을 이용하도록 하는 것이 항상 바람직하고 많은 응용에서 데이터를 이동시키는데 전혀 필요가 없게 된다. 물론, 매우 큰 레코드로 된 파일에 대해, 선택 정렬은 위에 논의 된것과 같이 이용하는 방법이다.

주어진 간접 수단에 대한 “인덱스 배열”은 배열을 지원하는 어떤 프로그래밍 환경에서 처리된다. C++에서, 배열 요소의 기계 번지를 이용하므로써 같은 원칙을 기반으로 한 구현이 되도록 하는 것이 편리하다. 즉, 3장에서 간단히 논의된 “진정한 포인터”이다. 예를 들면, 다음 코드는 포인터의 배열 p 이용해서 삽입 정렬을 구현한 것이다.

```
void insertion(itemType a[], itemType *p[], int N)
{
    int i, j; itemType *v;
    for ( i = 0; i <= N; i++ ) p[i] = &a[i];
    for ( i = 2; i <= N; i++ )
    {
        v = p[i]; j = i;
        while ( *p[j-1] > *v )
            { p[j] = p[j-1]; j--; }
        p[j] = v;
    }
}
```

포인터와 배열 사이의 강한 관계는 C++가 C에서 온것과 같이 가장 명확한 특징중의 하나이다. 일반적으로, 포인터로 구현한 프로그램들이 더 효율적이나 이해하기가 힘들다.(비록 이같은 특별한 응용에 대해, 차이가 많지는 않다) 관심있는 여러분은 위에 주어진 포인터 정렬에 대응하기 위해 필요한 institu 프로그램을 구현한다.

본 교재에서는 구현으로서 데이터를 직접 접근하고 포인터나 인덱스 배열을 보증될 때 과도한 데이터 이동을 피하기 위해 이용된다는 사실에서 안전하게 된다. 이같은 간접적인 접근 방법의 이용성 때문에, 본 장에서 말하고자하는 결론과 정수형 파일을 정렬하기 위하여 방법들을 비교할 때 따르는 것들은 더 일반적인 상황에 적용된다.

셸 정렬(Shellsort)

삽입 정렬은 인접된 요소들만을 교환하기 때문에 느리다. 예를 들면, 만약 가장 적은 요소들이 배열의 끝에서 존재될 때, 그것이 속한 곳을 얻기 위해서 N 단계가 필요케 된다. 셸 정렬은 멀리 떨어진 요소들과 교환을 허용하므로써 속도를 얻는 삽입 정렬의 간단한 확장이다.

개념은 모든 h 번째 요소(어느곳에서 시작해서)를 취함은 정렬된 파일을 생성하다는 속성을 주도록 파일을 재배열하는 것이다. 그런 파일을 h -정렬된 것이라고 말한다. 또 다른 방법으로서, h -정렬된 파일을 함께 사이에 끼워넣으면서 h 개 독립적으로 정렬된 파일들이다. h 의 어떤 큰 값에 대해서 h -정렬을 함으로써, 배열에서 요소들을 긴 거리로 이동하고 h 의 더 적은 값에 대해 h -정렬에 대해 더 쉽도록 하는 것이다. 1로서 끝나는 h 의 어떤 연속된 값들에 대해 그런 프로시저를 이용하는 것은 정렬된 파일을 생성하는 것이다. 즉, 이것이 셸 정렬이다.

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A													L	
	E													S
A	E	O	R	T	I	N	G	E	X	A	M	P	L	S
A				E				P				T		
	E				I				L				X	
		A				N				O				S
			G				M				R			
A	E	A	G	E	I	N	M	P	L	O	R	T	X	S
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

그림 8.7 셸 정렬

그림 8.7은 증가치가, 13, 4, 1로 된 예제 파일에서 셸 정렬의 처리를 나타낸 것이다. 첫 번째 과정에서 위치 1에 있는 A는 14번째 있는 L과 비교를 하고, 2번째 있는 S는 15번째 있는 E와 비교를 한다.(그리고 교환된다) 두 번째 과정에서, 위치 1, 5, 9와 13에 있는 A, T, E, P는 그 위치에서 A, E, P, T로 되게 재 배열을 하고 그후 위치 2, 6, 10과 14는 정렬된다. 마지막 과정은 삽입 정렬이 되나 멀리 이동될 요소들은 없다.

셸 정렬을 구현하는 한 방법은 각 h 에 대해 h 부분 파일의 각각에서 독립적으로 삽입 정렬을 이용한다.(표지는 사용된 h 의 가장 큰 값에 대해서 그것들의 h 가 되기 때문에 이용되지 않는다) 그러나 그것보다는 더 쉬운 것으로 판명된다. 즉, 만약 삽입 정렬에서 “1”로 발생하는 모든 것을 “ h ”로(그리고 “2”를 “ $h + 1$ ”로) 대체한다면, 다음과 같은 결과 프로그램이 h -정렬 파일이 되고 간단한 셸 정렬 구현이 된다.

```
void shellsort(itemType a[], int N)
{
    int in, j, h; itemType v;
    for ( h = 1; h <= N/9; h = 3*h+1 );
    for ( ; h > 0; h /= 3 )
        for ( i = h+1; i <= N; i += 1)
```

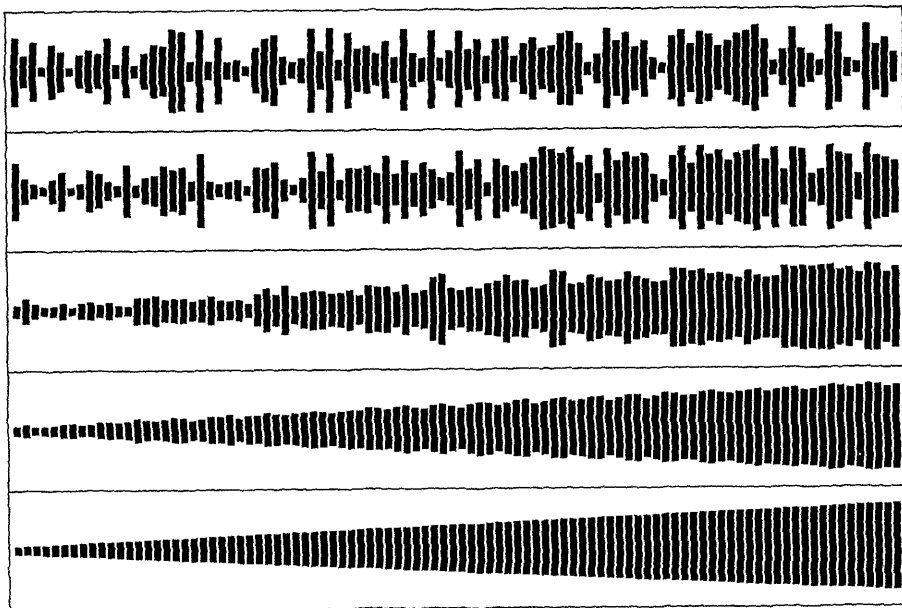


그림 8.8 무작위 순열의 셸 정렬

```

{
    v = a[i]; j = i;
    while ( j > h && a[j-h] > v )
        { a[j] = a[j-h]; j -= h; }
    a[j] = v;
}
}

```

이 프로그램은 증가 순서, 1093, 364, 121, 40, 13, 4, 1을 이용한다. 다른 증가 순서는 실제로 이것 뿐만 아니라 그것들에 관해서 수행을 하나 몇 가지 주의는 아래에 논의되는 것과 같이 고려된다. 그림 8.8은 각 h -정렬후에 배열 a 의 내용을 디스플레이하므로써 무작위 수열상에 처리하는 프로그램을 나타낸다.

이 프로그램에서 증가 순서는 이용하기가 쉽고 효율적인 정렬로 된다. 많은 다른 증가 순서는 더 효율적인 정렬로 되나, 상대적으로 큰 N 에 대해서 조차 20%이상으로 위의 프로그램을 줄이는 것은 어렵다.(훨씬 더 좋은 증가 순서의 가능성은 여전히 존재하나 아주 실제적이다) 다른 한편, 어떤 증가치는 나쁘게 된다. 예를 들면,, 64, 32, 16, 8, 4, 2, 1은 홀수 위

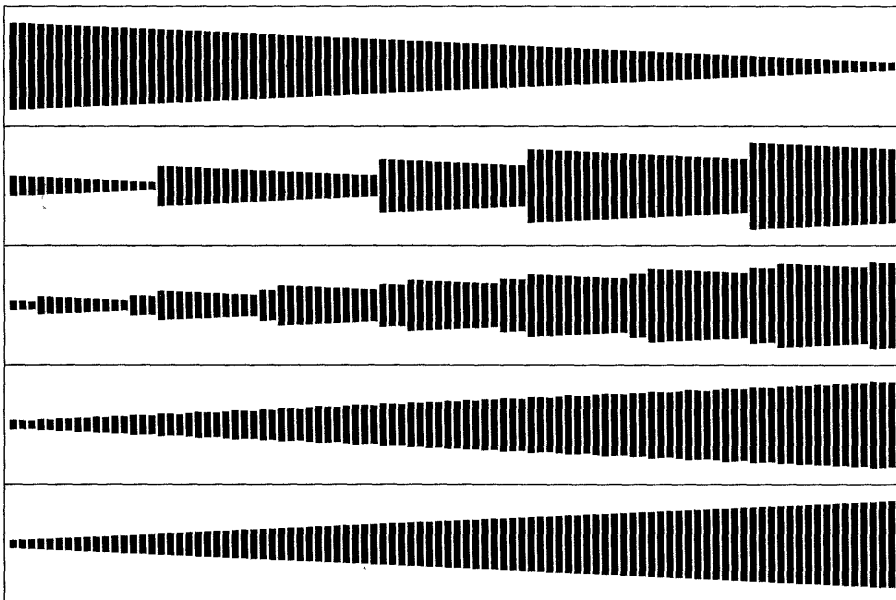


그림 8.9 역 순으로된 수열의 셀 정렬

치에 있는 요소들이 끝까지 짝수 위치에 있는 요소들과 비교를 하지 못하기 때문에, 나쁜 활용도로 된다. 비슷하게, 셸 정렬은 $h = N$ 에서 시작하므로써 구현된다.(같은 순서를 보증하도록 하기 위해서 초기화하는 대신에 항상 위의 것과 같이 이용된다) 이같은 것은 나쁜 순서가 어떤 N 에 대해 간단히 하는 것을 가상적으로 보증케 된다.

셸 정렬 효율성의 기술은 어느 누구도 알고리즘을 분석할 수가 없기 때문에 상세하지가 않다. 이것은 다른 증가 순서를 계산할 뿐만 아니고 해석적으로 다른 방법과 셸 정렬을 비교할 수 있는 것은 어렵다. 셸 정렬에 대한 실행시간의 함수적 형태조차 알려지지 않았으므로, 두 개의 연결은 $N(\log N)^2$ 와 $N^{1.25}$ 이다. 실행시간은 이미 순서적으로 된 파일에 대해 선형이나 역순으로된 파일에 대해 2차적인 삽입 정렬에서 특히 대조적으로 파일의 초기 순서에 상당히 감각적이 아니다. 그림 8.9는 그런 파일상에 셸 정렬의 처리를 나타낸다.

성질 8.6 셸 정렬은 $N^{3/2}$ 번 비교이상을 수행하지 않는다.(증가치 1, 4, 13, 40, 121,에 대해)

이 성질의 증명은 본 교재의 범위를 벗어난 것이나 여러분은 그것의 어려움을 인식하고 셸 정렬이 느리게 처리되는 파일을 구성하려고 함으로써 실제로 잘 처리가 되는 것 또한 확인을 해야한다. 위에 언급한대로, 셸 정렬은 2차원적인 비교의 수가 요구되는 어떤 나쁜 증가치 순서가 존재하나 $N^{3/2}$ 경계는 위에서 이용된 것을 포함한 넓은 여러 가지 순서들에 대해 유지되도록 제시된다. 훨씬 좋은 최악의 경우는 어떤 특별한 순서에 대해 알려져 있다. □

처리상에서 셸 정렬의 다른 견해를 나타내는 그림 8.10은 그림 8.3, 8.4와 8.5와 비교된다. 이같은 그림은 각 h -정렬뒤에(마지막을 제외하고 정렬이 완료된것) 배열의 내용을 나타낸것이다. 그림 8.3, 8.4와 8.5에서의 세 가지 다이어그램들은 제시된 알고리즘에 의해서 의미있는 처리의 양을 나타낸 것이다. 대조적으로 그림 8.10에서의 다이어그램들 각각은 한 개의 h -정렬 과정만을 나타낸다.

상당히 큰 파일들에 대해서(말하자면, 5000개의 요소들이하) 실행시간을 받아들이고 처리를 더 쉽게 하는 코드의 매우 적은 양만을 요구하기 때문에 많은 정렬 응용에서 셸 정렬은 선택의 방법이다. 다음 몇 개의 장들에서 더 효율적이나 그것들은 큰 N 을 제외하고 아마도 두 배정도로 빠른 것이고, 그것들은 의미가 있게 더 복잡한 방법들을 보게 된다. 간단히 보면, 만약 정렬 문제를 지니면, 위의 프로그램을 이용하고 그것과 더 복잡한 방법을 대체하는데 요구되는 특별한 노력이 가치가 있는지에 대해서 결정하게 된다.

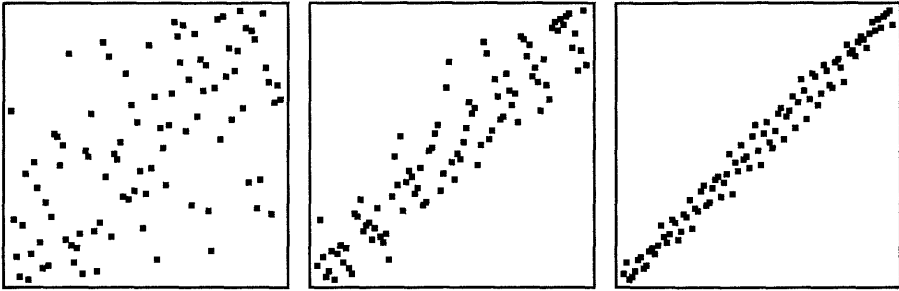


그림 8.10 무작위 순열의 셀 정렬

분배 계수기(Distribution Counting)

간단한 정렬 알고리즘인 매우 특별한 상황은 다음과 같다. 즉, “키들이 1에서 N 까지 서로 가른 정수형으로 된 N 개 레코드들의 파일을 정렬하는 것이다.” 이 문제는 문 `for (i = 1; i <= N; i++) b[a[i]] = a[i]`로 된 일시적인 배열 b 를 이용해서 해결한다.(또는 위에서 본바와 같이, 비록 더 복잡하지만, 외부의 배열 없이도 이같은 문제를 해결하는 것이 가능하다)

더 현실적인 문제는 다음과 같다. 즉, “키들이 1에서 $M - 1$ 사이에 정수형인 N 레코드들의 파일을 정렬하라.” 만약 M 이 너무 크지 않으면, 분산 계수기라 부르는 알고리즘은 이같은 문제를 해결하는데 이용된다. 개념은 각 값으로 된 키들의 수를 세고 그리고 다음 코드에서와 같이 파일을 통해서 두 번째 과정상의 위치로 레코드를 이동시키기 위해서 계수를 이용한다.

```
for (j = 0; j < M; j++) count[j] = 0;
for (i = 1; i <= N; i++) count[a[i]]++;
for (j = 1; j < M; j++) count[j] += count[j-1];
for (i = N; i >= 1; i--) b[count[a[i]]-1] = a[i];
for (i = 1; i <= N; i++) a[i] = b[i];
```

이같은 코드가 어떻게 처리되는가를 보기 위해서, 그림 8.11의 제일 위의 열에서 정수형의 예제 파일을 고려해 보자. 첫 번째 `for` 반복은 0으로 초기화시키는 것이고, 두 번째 것은 6개의 A가 있고 4개의 B가 있는 등등으로 인해서 `count[1]=6`, `count[2]=4`, `count[3]=1` 그리고 `count[4]=4`로 세트시킨다. 그때 세 번째 `for` 반복은 `count[1]=6`, `count[2]=10`,

A	B	B	A	C	A	D	A	B	B	A	D	D	A
					A								
					A								D
					A							D	D
				A	A							D	D
				A	A				B			D	D
				A	A			B	B			D	D
			A	A	A			B	B			D	D
			A	A	A			B	B		D	D	D
		A	A	A	A			B	B	C	D	D	D
	A	A	A	A	A			B	B	C	D	D	D
	A	A	A	A	A		B	B	B	C	D	D	D
	A	A	A	A	A	B	B	B	B	C	D	D	D
A	A	A	A	A	A	B	B	B	B	C	D	D	D

그림 8.11 분배 계수기

count[3]=11 그리고 count[4]=15를 생성하기 위해서 이같은 수들을 더한다. 즉, A와 같거나 적은 키는 6개가 있고 B와 같거나 적은 키들은 10개가 있는 등으로 된다.

여기서, 이것들은 그림에 제시된것과 같이 배열을 정렬하기 위한 번지들로서 이용된다. 원래 입력 배열 a는 제일 위의 라인에 제시된다. 즉, 그림의 나머지는 채워질 임시적인 배열을 나타낸 것이다. 예를 들면, 파일의 끝에 있는 A에 도달될 때, count[1]은 A와 같거나 적은 키들이 6개가 있으므로해서 그것을 위치 6에 삽입을 한다. a와 같거나 적은 키들이 몇 개 인가는 알 수 있으므로 count[1]이 결정된다. 그때 파일에서 그 다음부터 마지막 위치에 이르는 D는 위치 14로 놓게 되고 count[4]는 감소가 되는 등등으로 된다. 내부 반복은 N에서 1로 진행이 되므로 해서 정렬이 안정적이다.(여러분은 이것을 체크하기 바란다)

이같은 방법은 당연한 것으로 여기는 파일의 형태에 대해서 매우 잘 처리된다. 더구나, 10장에서 보게될 훨씬 더 강력한 방법을 생성하도록 확장된다.

연습 문제

1. 4개 레코드에 대한 “비교-교환” 처리의 순서를 제시하라.
2. 세 가지 기초적인 방법들(선택 정렬, 삽입 정렬 혹은 버블 정렬)중의 어느것이 이미 정렬된 파일에 대해서 빠르게 처리되는가?
3. 세 가지 기초적인 방법들 중 어느것이 역 순으로 되어진 파일에 대해서 빠르게 처리되는가?
4. 선택 정렬은 세 가지 기초적인 방법들중의 가장 빠른 것인가(정수형 정렬에 대해)에 대한 가설을 테스트하라. 그 후에 삽입 정렬 그 뒤에 버블정렬에 대해서 해 보아라.
5. 삽입 정렬에 대한 표지 키를 이용하는데 불편한 것중의 좋은 이유는 무엇인가?(셸 정렬의 구현에서 나타나는 것은 제외하고)
6. 셸 정렬에 의해서 7-정렬은 얼마나 많은 비교가 이루어지는가? 그 후에 3-정렬로서 키들이 E A S Y Q U E S T I O N에 대해서는 어떠한가?
7. 8, 4, 2, 1은 셸 정렬 증가 순서를 끝내기 위해 좋은 방법이 아닌 이유를 제시 하도록 하는 예를 제시하라.
8. 선택 정렬은 안전성이 있는가? 삽입 정렬과 버블 정렬은 어떠한가?
9. 구성요소들이 두 개의 값들중의 하나(x 혹은 y)로 만 되는 파일을 정렬하는데 분배 계수 기의 전문화된 버전은 무엇인가?
10. 셸 정렬에 대한 다른 증가 순서로서 실험을 하라. 즉, 1000개 요소들의 무작위 파일에 대해 주어진 것 보다 빠르게 처리되는 것을 찾아라.

빈 면

9 장

퀵 정렬

본 장에서는 다른 방법들보다 아마도 가장 많이 이용하는 정렬 알고리즘인 퀵 정렬에 대해서 알아 보고자 한다. 기본 알고리즘은 1960년에 C. A. R. Hoare에 의해서 개발되었고 그 이후에 많은 사람들에 의해서 연구가 되어져 왔다. 퀵 정렬은 구현하기가 어렵지 않고 범용적인 정렬이고,(여러 가지 상황에서도 잘 처리가 된다) 많은 상황에서 어떤 다른 정렬 알고리즘보다도 더 적은 자원을 이용하기 때문에 대중적으로 이용되었다.

퀵 정렬 알고리즘의 바람직한 특징은 그 장소내에서 이루어지고,(단지 적은 양의 표지 스택을 이용) n 개의 항목을 정렬하는데 평균의 경우에는 약 $N \log N$ 연산이 요구되고 상당히 적은 내부 반복을 지닌다. 알고리즘의 단점은 재귀적이고,(재귀가 적용이 안되면 구현이 복잡하다) 최악의 경우에 약 N^2 연산을 요구하고 미묘한 성질을 지닌 것이다. 즉, 구현에서 간단한 잘못이 무시되는 경향이 있고 어떤 파일에 대해서는 나쁘게 수행되는 경우가 있다.

퀵 정렬의 활용도에 대해 매우 잘 이해를 할 수가 있다. 그것은 철저한 수학적 분석에 영향을 받고 매우 자세한 문장은 활용도 문제에 대해서 이루어진다. 분석은 광대한 실험적 경험에 의해서 검증되고 알고리즘은 실제적인 정렬 응용의 다양성에 있어서 선택의 방법을 제공하는 점에 새롭게 된다. 이것은 퀵 정렬을 효율적으로 구현하는 방법에서 다른 알고리즘보다 다소 더 조심스럽게 보도록하는 것이 가치가 있다. 비슷한 구현 기법은 다른 알고리즘에 대해서 적절하다. 즉, 그것의 활용도를 아주 잘 이해하므로해서 퀵 정렬을 자신을 가지고 이용을 할 수가 있다.

퀵 정렬을 증진시키는 방법을 개발하려고 하는 것을 시도 해보자. 즉, 빠른 정렬 알고리즘은 전산학의 좋은 쥐덫이다. Hoare가 알고리즘을 처음 공포한 순간 직후 바로, 문헌에서 “개

선된” 버전들이 나타나기 시작하였다. 많은 개념들이 시도되고 분석이 되었으나 그러나 그것은 잘못 생각하기가 쉬운 것이다. 그 이유는 알고리즘은 그렇게 잘 균형이 잡혀있으므로 해서 프로그램 한쪽 부분에서 개선점들의 효과가 프로그램 다른쪽 부분의 효과에 의한 상쇄되어지는 이상의 것이기 때문이다. 부수적으로 퀵 정렬은 개선하는 세 가지 변형을 보도록 하자.

퀵 정렬의 조심스럽고 조화로운 버전은 어떤 다른 정렬 방법보다 대부분 컴퓨터상에서 더 빠르게 처리되는 것이다. 그러나, 어떤 알고리즘을 조화롭게 하는 것은 어떤 입력들에 대해서는 바라지 않고 기대하지 않은 효과로 나타내면서 더 미묘하게 만드는 것에 주의를 해야한다. 그런 효과들에 자유롭게 보여주는 것으로 된 버전들이 개발되면, 이것은 중요한 정렬 응용에 대해서나 혹은 라이브러리 정렬 유틸리티에 대해 이용하는 프로그램이 된다. 그러나 만약 퀵 정렬 구현이 단점을 지니지 않는다면, 셸 정렬이 더 적은 구현의 노력으로 아주 잘 수행이 되는 더 안전한 방법이 된다.

기본 알고리즘

퀵 정렬은 정렬에 대한 “분할-정복” 방법이다. 이것은 파일을 두 개 부분으로 나누고 그 부분들을 독립적으로 정렬을 하는 것이다. 보는 바와 같이, 분할 된것들의 정확한 위치는 파일에 의존하므로 해서, 다음의 재귀 구조를 지닌다.

```
void quicksort(itemType a[], int l, int r)
{
    int i;
    if ( r > l )
    {
        i = partition ( a, l, r );
        quicksort( a, l, i-1 );
        quicksort( a, i+1, r );
    }
}
```

파라미터 l 과 r 은 정렬이 되어질 원래 파일내에서 부분 파일의 경계를 정하는 것이다. 즉, 부름 `quicksort(a, 7, N)`는 전체 파일을 정렬하는 것이다.

방법의 실마리는 다음 세 가지 조건이 되도록 배열을 재배열하여야 하는 `partition` 프로시저 이다.

- (1) 요소 $a[i]$ 는 어떤 i 에 대해, 배열에서 마지막 장소내에 있어야 한다.
- (2) $a[1], \dots, a[i-1]$ 내에 요소들은 어느 것도 $a[i]$ 보다 큰 것이 없어야 한다.
- (3) $a[i+1], \dots, a[r]$ 내에 요소들은 어느 것도 $a[i]$ 보다 적은 것이 없어야 한다.

이것은 다음의 일반적인 방법을 통해서 간단하고 쉽게 구현된다. 첫 째, 마지막 위치의 요소가 존재하도록 $a[r]$ 을 임의적으로 선택한다. 다음으로, $a[r]$ 보다 큰 한 요소를 찾을 때까지 배열의 왼쪽 끝에서부터 조사를 해 나간다. 그리고 $a[r]$ 보다 적은 한 요소를 찾을 때까지 배열의 오른쪽 끝에서부터 조사를 한다. 조사가 멈추어지는 곳의 두 요소들은 마지막 분할된 배열에서 그 자리를 벗어난 것이므로 서 그것들을 교환한다.(실제로, 아래 기술된 여러 가지 이유로서, 비록 이것이 어떤 불필요한 교환이 수반이 되는 것 처럼 보이지만 $a[r]$ 과 같은 요소들에 대해 조사를 또한 멈추기 위해서는 가장 좋은 것이다) 이같은 방법을 계속하므로써 왼쪽 포인터의 왼쪽에 대한 모든 배열의 요소들은 $a[r]$ 보다 적고 오른쪽 포인터의 오른쪽에 대한 모든 배열의 요소들은 $a[r]$ 보다 크다. 조사 포인터가 서로 엇갈려 지날 때, 분할 과정은 거의 완성된다. 즉, 남아있는 모든 것은 $a[r]$ 을 오른쪽 부분 파일의 가장 왼쪽에 있는 요소와 교환하는 것이다.(왼쪽 포인터에 의해서 지시된 요소)

그림 9.1은 키들의 예제 파일이 이같은 방법으로 어떻게 분할되는 가를 나타내고 있다. 가장 오른쪽의 요소 E는 분할 요소로서 선택된다. 첫째, 왼쪽에서 조사가 S에 멈추고 오른쪽의 조사는 A에서 멈추게 되고,(표의 두 번째 라인에 제시된 것과 같이) 그리고 그 후에 두 값을 교환한다. 다음으로 왼쪽 조사는 O에서 멈추고, 오른쪽에서의 조사는 E에서 멈추므로,

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S									A	M	P	L	E
A	A	O						E	X	S	M	P	L	E
A	A	E	R	T	I	N	G	O	X	S	M	P	L	E
A	A	E	E	T	I	N	G	O	X	S	M	P	L	R

그림 9.1 분할

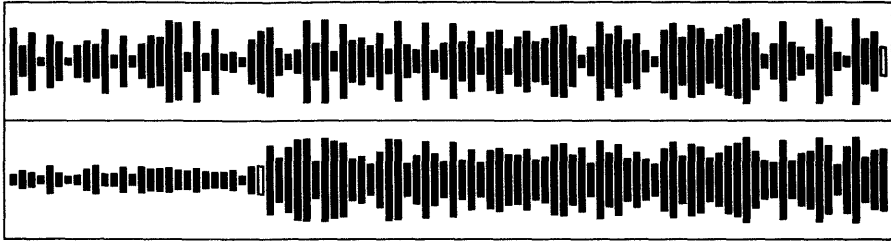


그림 9.2 더 큰 파일을 분할

(표의 세 번째 라인에 제시된것과 같이) 그 후에 두 값은 교환된다. 다음으로 포인터는 서로 엇갈려 지나간다. 왼쪽에서의 조사는 R에 멈추고 오른쪽의 조사는 E에 멈추게 된다. 이같은 점에서 적절한 이동은 그림 9.1의 마지막 라인에 제시된 파일을 분할 시키는 것이 남아 있지만 E를 R과의 오른쪽에 있는 것을 교환하는 것이다.

분할 과정은 어떤 키가 교환 동안에 그것들과 같은 큰 수(지금까지 조사되지 않은 것)를 지나서 이동되므로 안정성이 없다.

그림 9.2는 더 큰 파일을 분할시키는 결과이다. 즉, 왼쪽에 있는 적은 요소들과 오른쪽에 있는 큰 요소들로서, 분할된 파일은 무작위 파일보다 그 속에서 상당히 더 “순서화”되어져 있다. 정렬은 분할된 요소의 어느 측면(재귀적으로)상에서 두 개 부분 파일을 정렬함으로써 끝나게 된다. 다음 프로그램은 방법의 완전한 구현을 나타낸다.

```
void quicksort(itemType a[], int l, int r)
{
    int i, j; itemType v;
    if ( r > l )
    {
        v = a[r]; i = l - 1; j = r;
        for ( ; ; )
        {
            while ( a[++i] < v ) ;
            while ( a[--j] > v ) ;
            if ( i >= j ) break;
            swap( a, i, j );
        }
        swap( a, i, r );
    }
}
```

```

        quicksort( a, l, i-1);
        quicksort( a, i+1, r);
    }
}

```

이같은 구현에서, 변수 v 는 “분할 요소” $a[r]$ 의 현재 값을 가지고 있고 i 와 j 는 각각이 왼쪽과 오른쪽 조사 포인터들이다. 분할 반복은 포인터가 엇갈려 지날 때 break명령으로 빠져나오는 무한 반복으로 구현된다. 이 방법은 break 능력이 왜 유용한가의 전형적인 예이다. 즉, 여러분은 break를 이용하지 않고 분할을 구현하는 방법을 고려하는 것이 즐거운 것이다.

삽입 정렬에서와 같이, 표지 키는 분할 요소가 파일에서 가장 작은 요소인 경우에서 조사를 멈추게 할 필요가 있다. 이같은 구현에서 분할요소가 파일에서 가장 큰 요소일 때 표지는 조사를 멈출 필요는 없다. 그 이유는 분할 요소 그자체가 조사를 멈추게 하는 파일의 오른쪽 끝에 존재케 된다. 양쪽 표지 키들을 제거하는데 쉬운 방법을 볼 수가 있다.

퀵 정렬의 “내부 반복”은 고정된 값에 반대하여 배열을 비교함과 포인터를 증가하는 것이 단순히 포함된다. 이같은 것은 퀵 정렬이 얼마나 빠르게 하는가이다. 즉, 더 간단한 내부 반복을 만드는 것은 어렵다. 내부 반복에 공간의 테스트만을 첨가함은 활용도상에서 현저한 효과를 지니므로 해서 표지들의 이득적인 효과 또한 여기서 중요시한다.

지금 두 개 부분파일은 정렬을 끝내면서 재귀적으로 정렬된다. 그림 9.3은 이같은 재귀적인 부름을 통해서 추적된다. 각 라인은 분할 요소를 이용해서(다이아그램에서 그림자 표시된 부분) 디스플레이된 부분 파일을 분할 시키는 결과로 된다. 만약 프로그램에서 처음 테스트가 $r > 1$ 이기 보다는 오히려 $r \geq 1$ 인 경우, 그때 모든 요소는 분할 요소로서 이용되므로써 그 장소에(결국에는) 놓여지게 된다. 즉, 주어진 구현에서 크기 1의 파일은 그림 9.3에서와 같이 분할 될 필요가 없다. 이같은 개선된 내용의 일반화는 아래에 더 자세히 기술이 되어져 있다.

위 프로그램의 가장 혼란스런 특징은 간단한 파일 상에서 매우 비 효율적인 것이다. 예를 들면, 만약 이미 정렬되어진 파일을 부르는 경우, 분할은 감소되고 프로그램은 각 부름에 대해 한 요소만을 처리함으로써 자신을 N 번 부르게 된다. 이것은 요구된 시간이 약 $N^2/2$ 일 뿐만 아니라 제귀를 처리하는데 요구되는 공간은 받아들일 수 없는 약 N (아래 참조)이 됨을 의미한다. 다행히도, 이같은 최악의 경우는 프로그램의 실제적 응용에서 발생되지 않는 것을 보증하기는 상대적으로 쉬운 방법들있다.

A	A	E	E	T	I	N	G	O	X	S	M	P	L	R
A	A	E												
A	A													
				L	I	N	G	O	P	M	R	X	T	S
				L	I	G	M	O	P	N				
				G	I	L								
					I	L								
								N	P	O				
									O	P				
												S	T	X
													T	X

그림 9.3 퀵 정렬에서의 부분 파일

동등한 키들이 파일내에 존재할 때, 두 개의 세밀한 구분은 명백하다. 첫째, 양쪽 포인터들이 분할 요소와 같은 키들상에서 멈춤을 가지는지 혹은 한 개 포인터가 멈추고 다른 포인터는 그것들에 대해 조사를 하는지 혹은 양쪽 포인터가 그것들에 대해 조사를 하는지에 대한 의문이 있다. 이같은 의문은 실제적으로 수학적면에서 상세하게 연구가 되고 결과는 양쪽 포인터들이 멈추도록 하는 것이 제일 좋은 것으로 나타난다. 이것은 많은 같은 키들의 존재에서 분할을 균형되도록하는 경향이 있다. 둘째, 똑같은 키들의 존재에서 가로지르는 포인터를 적절히 처리하는 의문이 있다. 실제로 위의 프로그램은 $j < i$ 일 때 조사를 종료하고 첫 번째 재귀 부름에 대해 $\text{quicksort}(a, l, j)$ 를 이용해서 다소 개선된다. 이것은 $j=i$ 일 때

반복이 한 번이상 이루어지도록 함으로써 분할로서 두 개 요소들을 위치에 넣기 때문에 개선된다.(이같은 경우는 예를 들면, 위의 예에서 R이 E인 경우에 발생된다) 주어진대로 프로그램이 a[r]에서 분할 키와 동등한 키로써 레코드에 남겨주기 때문에 이같은 변화를 만들 가치가 있고 그것의 가장 오른쪽 키가 가장 적은 것이기 때문에 부름 quicksort(a, i+1, r)에서 첫 번째 분할을 감소시키도록 한다. 위에 주어진 분할의 구현은 이해하기가 다소 쉽다. 그래서 똑같은 키들이 존재할 때 이같은 변화가 일어나는 것을 이해함으로써 이같은 논의를 아래에서와 같이 남겨두기로 하자.

퀵 정렬의 활용도 특징

퀵 정렬에서 발생하는 가장 좋은 것은 각 분할 단계에서 정확히 반으로 파일을 나누는 것이다. 이것은 퀵 정렬에 의해서 이용된 비교의 수를 분할-정복 재발생을 만든다.

$$C_N = 2 C_{N/2} + N$$

$2C_{N/2}$ 는 두 개 부분 파일들을 정렬하는 비용을 나타낸다. 여기서, N 은 포인터나 다른 것을 분할하는 것을 이용해서 각 요소를 조사하는 비용이다. 6장에서, 이같은 재발생은 다음 해를 지낸다.

$$C_N \approx N \lg N$$

비록 이같은 것이 항상 잘 처리되지는 않지만, 분할은 평균의 경우 중앙에서 이루어진다. 각 분할 위치의 상세한 확률을 고려하는 것은 재발생을 더 복잡하게 하고 풀기가 더 어렵게 하나 마지막 결과는 유사하다.

성질 9.1 퀵 정렬은 평균의 경우 약 $2N \ln N$ 비교가 된다.

N 요소의 무작위 순열에 대한 퀵 정렬에 의해 이용된 비교의 수에 대한 상세한 재발생 공식은 다음과 같다.

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k}).$$

여기서

$C_1 = C_0 = 0$ 로 된 $N \leq 2$ 에 대한 것이다.

$N + 1$ 항은 다른 것들의 각각과 분할 요소를 비교하는 비용을 나타낸다.(포인터가 가로지르는 두 개와 별개의 것) 나머지는 크기 $k - 1$ 과 $N - k$ 의 무작위 파일들로서 남겨진 뒤에 각 요소 k 가 확률 $1/k$ 로서 된 분할 요소인 것에서 생성된다.

비록 그것이 복잡하게 보일지라도, 이같은 재발생은 실제로 세단계로서 해결하기 쉽다. 첫째, $C_0 + C_1 + C_2 + \dots + C_{N-1}$ 은 $C_{N-1} + C_{N-2} + \dots + C_0$ 과 같으므로 해서 다음과 같다.

$$C_N = N + 1 + \frac{2}{N} \sum_{1 \leq k \leq N} C_{k-1}.$$

둘째로 N 을 양쪽에 곱하고 $N - 1$ 에 대한 같은 공식을 빼므로해서 다음과 같이 합이 제거된다.

$$N C_N + (N - 1) C_{N-1} = N (N + 1) - (N - 1) N + 2 C_{N-1}$$

이것은 재발생으로 간단하게 된다.

$$N C_0 = (N + 1) C_{N-1} + 2N$$

세 번째로, $N(N + 1)$ 을 양쪽을 나누는 것은 다음과 같은 재발생이 주어진다.

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1} = \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} = \dots = \frac{C_2}{3} + \sum_{3 \leq k \leq N} \frac{2}{k+1}.$$

이것의 정확한 답은 다음 적분에 의해 쉽게 근사치화되는 합과 거의 같다.

$$\frac{C_N}{N+1} \approx 2 \sum_{1 \leq k \leq N} \frac{1}{k} \approx \int_1^N \frac{1}{x} dx = 2 \ln N,$$

그리고 이것은 제시된 결과이다. $2N \ln N \approx 1.38 N \lg N$ 이므로, 평균 비교의 수는 제일 좋은 경우보다 약 38% 높다. \square

이와 같이, 위의 구현은 무작위 파일에 대해 매우 잘 수행된다. 그리고 이것은 많은 응용에 대해 매우 합당한 범용 정렬이다. 그러나, 만약 정렬이 상당히 여러번 이용되면 혹은 매우 큰 파일을 정렬하는데 이용된다면, 그때 나쁜 경우가 발생할 것에 훨씬 적게 만들고, 평균 실행시간이 20%정도로 감소시키고 표지 키에 대한 필요성을 쉽게 제거시키는 것이 아래에 논의된 여러 가지 개선점을 구현하는 것이 가치가 있다.

재귀를 제거(Removing Recursion)

5장에서 처럼, 명확한 스택 삽입을 이용해서 퀵 정렬 프로그램에서 재귀를 제거 할 수가 있다. 그리고 정렬되어진 부분 파일의 양식에서 “수행될 작업”을 포함한 것으로 간주된다. 처리할 부분 파일을 필요할 때, 스택에서 제거를 한다. 분할을 할 때, 스택에 삽입되어서 처리될 두 개 부분 파일을 생성한다. 이것은 다음의 비재귀 구현으로 나타난다.

```
void quicksort(itemType a[], int l, int r)
{
    int i; Stack<int> sf(50);
    for (;;)
    {
        while ( r > l )
        {
            i = partition ( a, l, r);
            if ( i-l > r-i )
                { sf.push(l); sf.push(i-1); l = i+1; }
            else
                { sf.push(i+1); sf.push(r); r = i-1; }
        }
        if ( sf.empty()) break;
        r = sf.pop(); l = sf.pop();
    }
}
```

이 프로그램은 위의 기술된 내용에서 두 가지 중요한 방면에서 다르다. 첫째, 두 개 부분 파일은 어떤 임의의 순서로서 스택에 삽입되지 않으나 그것들의 크기가 체크 된후에 두 개

중의 더 큰 것이 먼저 스택에 삽입된다. 둘째, 두 개 부분 파일중의 더 작은 것은 전혀 스택에 삽입되지를 않는다. 즉, 파라미터들의 값을 단순히 다시 세트된다. 이것이 5장에서 논의된 “끝의 재귀 제거(end-recursion-removal)” 기법이라고 한다. 퀵 정렬에 대해서, 끝의 재귀 제거와 두 개 부분 파일들중의 더 적은 것을 처리하는 방법의 조화는 스택의 제일 꼭대기뒤에 각 엔트리는 이전 엔트리의 크기에 반이하로 된 부분 파일을 나타내야만 하기 때문에 약 $\lg N$ 엔트리들에 대해서만 스택의 필요성에 여지를 포함함을 확신시키도록 먼저 판명된다.

이것은 N 만큼 큰 것인(예를 들면, 파일이 이미 정렬이 될 때) 재귀 구현에서 최악의 경우에 스택의 크기에 상당히 대조적으로 이루어진다. 이것은 미묘한 것이나, 퀵 정렬의 재귀적인 구현에서 진정한 어려움이 있다. 즉, 거기에는 항상 중요시되는 스택이 있고 큰 파일상에 변

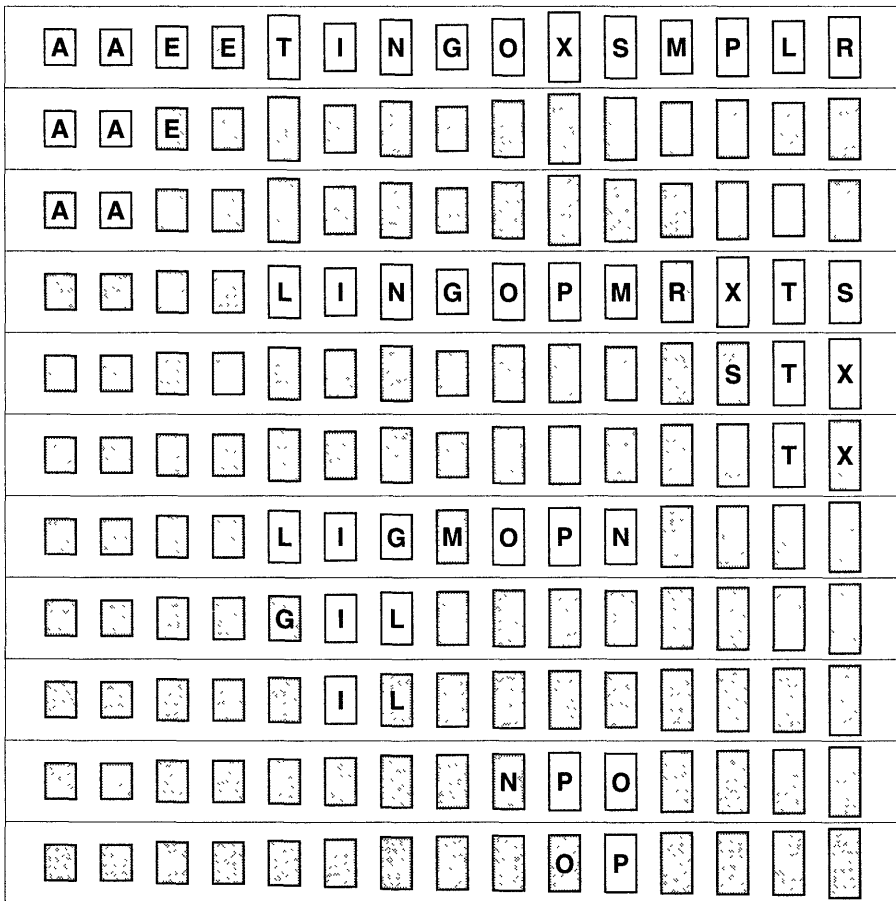


그림 9.4 퀵 정렬에서 부분파일(비 재귀적)

질된 경우는 기억장소의 부족으로 인해 프로그램이 비정상적으로 끝나거나 라이브러리 정렬 루틴에 대해 명백히 바라지 못한 경우로 된다. 아래에서 나쁜 경우는 상당히 바람직하지 않음을 나타내는 방법을 보게되나, 재귀 구현에서 이같은 문제를 회피하는 것은 끝의 재귀 제거없이 어렵다.(어느 부분 파일이 처리되는가의 순서를 바꾸는 것조차 도움이 되지 않는다) 다른 한편, 어떤 C++ 컴파일러는 자동적으로 끝의 재귀를 제거하고 어떤 기계는 재귀에 대한 직접적인 하드웨어 도움을 준다. 위의 프로그램은 그런 환경하에서 재귀 구현보다 실제적으로는 더 느리다.

위의 프로그램에서 명확한 스택의 간단한 이용은 직접적인 재귀 구현보다 훨씬 더 효율적이고 더 안전하게 하고 제거되어야 할 불필요한 동작들이 여전히 존재한다. 만약 양쪽 부분 파일이 단지 한 개의 요소를 지니면, $r = 1$ 로된 엔트리는 즉각적으로 빼내서 무시하는 것으로 스택상에 삽입된다. 스택상에 그런 파일이 삽입이 안되도록 하기 위해서, 프로그램을 변경하는 것은 간단하다. 이같은 변화는 아래 기술된 다음 개선이 포함될 때 훨씬 더 효과적이다. 이것은 더 적은 부분 파일을 같은 방법으로 무시되는 것이 포함되고 그래서 양쪽 부분 파일들이 무시되어질 필요가 있는 변화는 훨씬 크다.

물론 비 재귀적 방법은 어떤 파일에 대한 재귀 방법으로서 같은 부분 파일을 처리한다. 즉, 그것들은 다른 순서로서 수행이 된다. 그림 9.4는 우리가 제시한 예제에 대한 분할을 나타낸 것이다. 즉, 첫 번째 세 가지 분할은 같으나 그때 그것은 왼쪽에 존재하므로써 재귀 방법은 R의 오른쪽 부분 파일을 분할하는 등이다.

만약 그림 9.3과 9.4를 “합치고” 각 분할 요소를 그것의 두 개 부분 파일에서 이용된 분할 요소에 연결을 하면, 그림 9.5에 제시된 분할 과정의 정적인 표현을 얻게 된다. 이같은 이진 트리에서, 각 부분 파일은 그것의 분할 요소(또는 크기 1인 경우에는 그자체가)로 표현되고 각 노드의 부분 트리는 분할 후에 부분 파일을 나타내는 트리들이다. 트리에서 사각형의 외

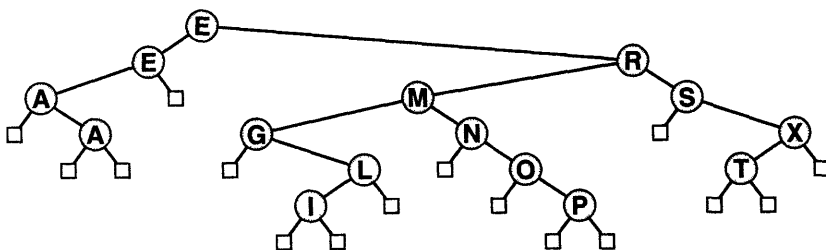


그림 9.5 쿼 정렬에서 분할 과정의 트리 다이어그램

부 노드들은 빈 부분 파일을 나타낸다.(명확히, 두 번째 A, I, P와 T는 두 개 빈 부분 파일을 지나는 것으로 제시된다. 즉, 위에 논의 된대로, 알고리즘의 편차는 빈 부분 파일을 다르게 처리하는 것이다) 퀵 정렬의 재귀 구현은 프리 오더로 이 트리의 노드들을 방문하는 것에 대응된다. 비 재귀적인 구현은 “더 적은 부분 트리를 먼저 방문하는” 규칙에 대응된다. 14장에서, 퀵 정렬과 근본적인 검색 방법사이의 직접적인 관계가 어떻게 되는가를 보게 될 것이다.

적은 부분 파일들

퀵 정렬에 대한 두 번째 개선점은 재귀 프로그램이 많은 적은 부분 파일들에 대해 그 자체를 부르는 것으로 적은 부분 파일이 만날 때 가능한 좋은 방법을 이용해야하는 것에서 제기된다. 이것을 수행하는 한 가지 명백한 방법은 삽입 정렬상에(정렬되어질 부분 파일을 정의하는 파라미터를 받아드리도록 변경하는 것) 부름에 대한 “if ($r > 1$)”에서 재귀 루틴의 시작에 있는 테스트를 변경시키는 것이다. 즉, “if ($r-1 \leq M$) insertion(l, r)”이다. 여기서 M 은 구현에서 정확한 값에 의존하는 파라미터이다. M 에 대해 선택된 값은 제일 좋은 값일 필요는 없다. 즉, 알고리즘은 약 5에서 약 25에 이르는 범주에서 M 에 대해 거의 같게 처리된다. 실행 시간이 줄어드는 것은 대부분 응용에 대해서 20%의 순서에서 이루어진다.

또한 다소 훨씬 효율적인 적은 파일들을 처리하는 다소 쉬운 방법은 “if ($r-1 > M$)”에 대한 처음에 테스트를 변경시키는 것이다. 즉, 분할 동안에 적은 부분 파일을 단순히 무시하는 것이다. 비 재귀 구현에서, 스택상에 M 이하의 어떤 파일을 삽입시키지 않고도 수행된다. 분할 후에, 왼쪽에 있는 것은 거의 분할된 파일이다. 그러나 이전 장에서 언급한 것 처럼 삽입 정렬은 그런 파일에 대한 선택의 방법이다. 즉, 삽입 정렬은 마치 그것들이 직접적으로 사용된 것처럼 얻을 수 있는 적은 파일의 수집에 대해서와 같이 그런 파일에 대한 것을 처리한다. 이같은 방법은 삽입 정렬이 퀵 정렬을 전혀 작동하지 않은 것으로 되는 오류를 지니는 것조차 항상 정렬되어지는 것이기 때문에 주의를 가지고 이용을 해야한다.

그림 9.6은 크고 무작위적으로 순서화된 배열에서 이같은 처리의 견해를 준다. 이같은 다이어그램들은 각 분할이 독립적으로 조정되어진 두 개 독립적인 부분 문제들로 부분 배열을 나누는 방법을 그림적으로 기술한 것이다. 이같은 그림에서 부분 배열은 무작위적으로 배열된 점들의 사각형으로서 제시가 되었다. 즉, 분할 과정은 대각선으로 끝이나는 한 요소(분할 요소)로 된 그런 사각형을 두 개 더 적은 사각형으로 나눈다. 분할에서 수반되지 않은 요소

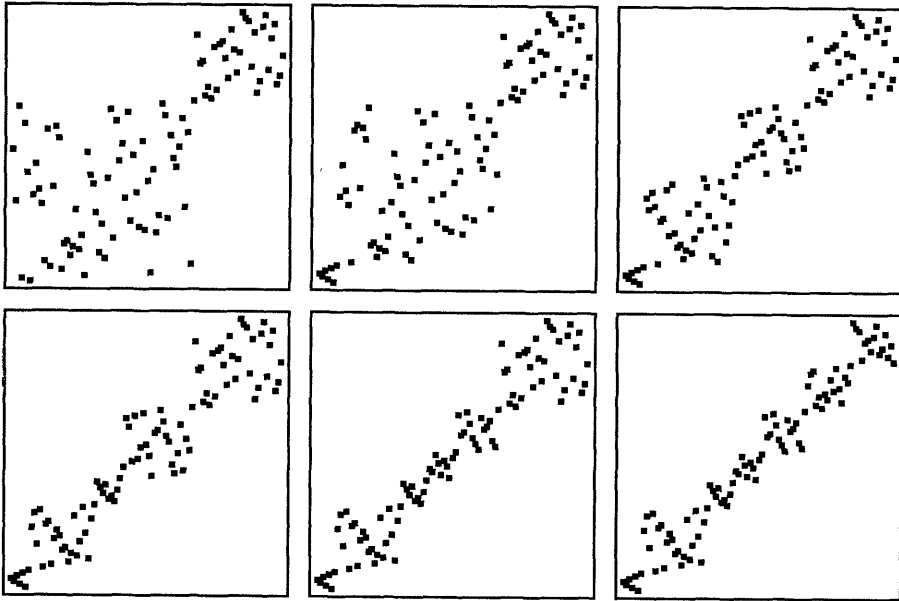


그림 9.6 퀵 정렬(재귀, 적은 부분파일들은 무시된다)

들은 삽입 정렬에 의해서 쉽게 처리되는 배열은 남겨두고서 대각선에 아주 근접되는 것으로 끝이 난다. 위에 지시한 대로, 퀵 정렬의 비 재귀 구현에 대해 대응되는 다이어그램은 비슷하지만 그러나 분할은 다른 순서로서 행해진다.

세 가지중의 중위수를 분할(Median-of-Three Partitioning)

퀵 정렬의 세 번째 개선점은 더 좋은 분할 요소를 이용하는 것이다. 여기에는 여러 가지 가능성이 있다. 최악의 경우를 피하는 가장 안전한 선택은 분할 요소에 대한 배열에서 무작위 요소인 것이다. 그때 최악의 경우는 적은 확률로서 발생된다. 이것은 입력의 배열에 관계 없이 거의 항상 좋은 활용도를 이루는 무작위를 이용하는 “확률론적 알고리즘”의 간단한 예제이다. 특히 무작위는 입력에서 어떤 편차가 의심되는 경우에, 알고리즘 설계에서 유용한 도구가 된다. 퀵 정렬에서, 이같은 목적에 대해 완전한 무작위 발생기에서 삽입되는 것을 없애는 것이다. 즉, 임의의 수로 수행된다.(35장 참조)

더 유용한 개선점은 파일에서 세 가지 요소를 취해서 분할요소에 대한 세 가지 중의 중위수값을 이용하는 것이다. 만약 선택된 세 가지 요소가 배열의 왼쪽, 중간 그리고 오른쪽에서 존재하는 경우, 그때 표지의 이용은 다음과 같이 피하게 된다. 즉, 세 가지 요소를 정렬하고

(이전 장에서 세 가지를 교환하는 방법을 이용하므로써), 그때 중간에 있는 것과 $a[r-1]$ 을 교환하고 그리고 $a[l+1], \dots, a[r-2]$ 상에 분할 알고리즘을 처리하는 것이다. 이같은 개선점을 세 가지 중의 중위수를 분할하는 방법이라고 부른다.

세 가지 중의 중위수 방법은 세 가지 방법으로 퀵 정렬을 도와준다. 첫째, 어떤 실제적인 정렬에서 발생하는 것보다 훨씬 다른 최악의 경우로 만든다. 정렬이 N^2 시간으로 처리되도록 하기 위해서, 조사된 세 가지 요소중의 두 가지만을 분할의 대부분을 통해 계속적으로 발생이 된다. 둘째, 이같은 함수는 분할전에 조사된 세 가지 요소들에 의해서 처리되므로 분할에 대한 표지 키에 대한 필요성을 제거시키는 것이다. 세 번째로, 약 5%정도 알고리즘의 전체 평균 실행시간을 실제적으로 감소시키는 것이다.

적은 부분 파일에 대해 중단된 세 가지중의 중위수 방법의 비 재귀적 구현에 대한 조합은 순수한 재귀적 구현에 대해 퀵 정렬의 실행시간을 25%에서 30%로 증진시키는 것이다. 더 알고리즘적인 개선이 가능하지만, (예를 들면, 5개나 그이상의 요소들중의 중위수가 이용된다) 그러나 얻어진 시간의 양은 최저이다. 더 의미있는 시간의 절약은 어셈블리나 기계어에서 내부 반복(또는 전체 프로그램)을 코딩하므로써(더 적은 노력으로) 인식된다. 중요한 정렬 응용에 전문가들에 대해서 가능한 것은 제외하고 어느 경로도 추천되지 않는다.

선택

완전한 정렬이 반드시 필요하지 않은 정렬과 관련된 한 응용은 수들 집합의 중위수를 찾는 연산이다. 이것은 통계학이나 여러 가지 다른 자료 처리 응용에서 공통적인 응용이다. 처리되는 한 방법은 수들을 정렬하고 중위수에서 보게 되나 퀵 정렬 분할 처리를 이용해서 더 잘 수행된다.

중위수를 찾는 연산은 선택(selection) 연산의 특별한 경우이다. 즉, 수들의 집합가운데 k 번째 적은 것을 찾는 것이다. 알고리즘은 특별한 항목을 조사하지 않고도 k 번째 적은 값이고 더 적은 $k - 1$ 개 요소가 인식이 되고 더 큰 $N - k$ 개의 요소들을 인식하는 것이므로 해서, 대부분의 선택 알고리즘은 상당한 정도의 특별한 계산이 없이도 파일의 k 개 가장 적은 요소들 모두를 되돌려준다.

선택은 지수나 다른 데이터의 처리에서 많은 응용을 지닌다. 중위수와 파일을 더 적은 집합으로 나누는 다른 순서 통계의 이용에서 매우 공통적인 것이다. 큰 파일의 적은 부분만이 더 많은 처리에 대해 보관이 된다. 그런 경우에서, 선택되는 프로그램 즉, 말하자면 파일의 요소들중에서 제일 위의 10개는 완전한 정렬보다 더 적절할 것이다.

선택에 직접적으로 적합한 알고리즘을 이미 보았다. 만약 k 가 매우 적으면, 그때 선택 정렬은 Nk 의 시간을 요구하면서 잘 처리된다. 즉, 먼저 가장 적은 요소를 찾고 그후 나머지 요소들에서 가장 적은 것을 찾으므로써 두 번째로 적은 요소를 찾는 등등으로 된다. 다소 큰 k 에 대해, $N \log k$ 에 비례하는 시간에 처리되는 방법을 11장에서 보게 된다. k 의 모든 값들에 대한 평균의 경우 선형 시간으로 처리되는 흥미로운 방법은 퀵 분할에서 이용되는 분할 프로시저에서 공식화 된다. 퀵 정렬의 분할 방법은 배열 $a[1], \dots, a[N]$ 을 재배열하고 $a[1], \dots, a[i-1]$ 이 $a[i]$ 와 같거나 적은 것이고 $a[i+1], \dots, a[N]$ 은 $a[i]$ 보다 같거나 큰 것인 정수형 i 를 돌려준다. 그렇지 않으면, 만약 $k < i$ 이면, 그때 왼쪽 부분 파일에서 k 번째 가장 적은 요소를 찾을 필요가 있고, 만약 $k > i$ 이면, 그때 오른쪽 부분 파일에서 $(k-i)$ 번째 적은 요소를 찾을 필요가 있다. 이것을 배열 $a[1], \dots, a[r]$ 에서 k 번째 가장 적은 요소를 찾는 것에 적용하는 것으로 조정하는 것은 다음 프로그램과 같다.

```
void select(itemType a[], int l, int r, int k)
{
    int i;
    if ( r > l )
    {
        i = partition(a, l, r);
        if ( i > l+k-1 ) select(a, l, i-1, k);
        if ( i < l+k-1 ) select(a, i+1, r, k-i);
    }
}
```

이같은 프로시저는 배열을 재 배열을 해서 $a[1], \dots, a[k-1]$ 은 $a[k]$ 와 같거나 적은 것이고 $a[k+1], \dots, a[r]$ 은 $a[k]$ 와 같거나 크다.

예를 들면, 부름 $\text{select}(1, N, (N+1)/2)$ 는 중위수 값에서 배열을 분할한다. 정렬 예제에서 키들에 대해, 이 프로그램은 그림 9.7에 제시된대로 중위수를 찾기 위해서 세 가지 재귀 부름만을 이용한다. 파일은 재 배열이 되어서 중위수의 왼쪽에는 중위수보다 적은 요소들이 그리고 오른쪽에는 더 큰 값들이 존재케 되나, (똑같은 값을 지닌 요소는 어느쪽에 존재해도 된다) 그것은 완전히 정렬된 것이 아니다.

`select` 프로시저가 그자체에서 부름으로 항상 끝나므로써, 재귀 부름에 대해서 시간이 발생될 때 단순히 파라미터를 다시 세트하고 처음의 위치로 되돌린다. (스택은 재귀를 제거하기 위해서 필요치 않는다) 또한 다음 구현에서와 같이 k 를 포함한 단순한 계산을 제거한다.

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	A	E	E	T	I	N	G	O	X	S	M	P	L	R
				L	I	N	G	O	P	M	R	X	T	S
				L	I	G	M	O	P	N				

그림 9.7 중위수를 찾기 위한 분할

```

void select(itemType a[], int N, int k)
{
    int i, j, l, r; itemType v;
    l = 1; r = N;
    while ( r > l )
    {
        v = a[r]; i = l - 1; j = r;
        for ( ; ; )
        {
            while ( a[++i] < v ) ;
            while ( a[--j] > v ) ;
            if ( i >= j ) break;
            swap( a, i, j );
        }
        swap( a, i, r );
        if ( i >= k ) r = i - 1;
        if ( i <= k ) l = i + 1;
    }
}

```

퀵 정렬에 대한 똑같은 분할 프로시저를 이용하고 퀵 정렬로써와 같이 많은 똑같은 키들이 기대되는 경우에 조금 변경이 된다.

그림 9.8은 더 큰(무작위) 파일상에서 선택 처리를 제시한 것이다. 퀵 정렬로써, 매우 큰 파일에서 각 분할은 배열을 반으로 쪼개는 것이므로써 전체 처리는 약 $N + N/2 + N/4 + N/8 + \dots = 2N$ 번 비교를 요구한다. 퀵 정렬과 같이, 이같은 논의는 진실과는 그리 멀어지는 것은 아니다.

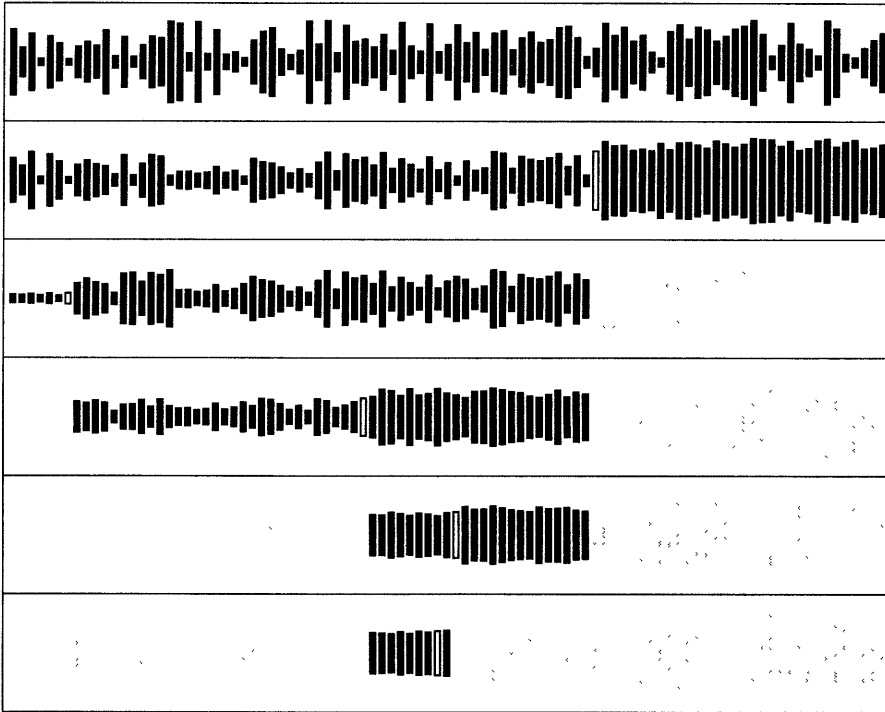


그림 9.8 중위수를 구함

성질 9.2 퀵 정렬을 기본으로 한 선택은 평균의 경우에 선형시간이다.

퀵 정렬에 대해 위에 주어진 것은 비교에서 평균의 수가 k 의 어떤 허용된 값에 대해 선형인 약 $2N + 2k \ln(N/k) + 2(N - k) \ln(N/(N - k))$ 인 결과로 나타내는 것과 비슷하거나 그러나 그것보다 훨씬 더 복잡한 분석이 된다. $k = N/2$ (중위수를 포함)에 대해, 이것은 약 $(2 + 2 \ln 2)N$ 번 비교로 계산된다. \square

최악의 경우는 퀵 정렬에서와 같다. 즉, 이미 정렬된 파일에서 가장 적은 요소를 찾는 이 같은 방법을 이용함은 실행시간이 2차원적인 것이다. 임의적으로나 무작위의 분할 요소를 이용하나 그러나 어떤 주의에 관심을 기울여야 한다. 예를 들면, 만약 가장 적은 요소가 발견되면, 거의 중간에서 파일을 분리하는 것을 원치 않을 것이다. 그것은 이 같은 퀵 정렬을 기반으로 한 선택 프로시저를 변경시키는 것이 가능하므로해서 그것의 실행시간은 선형적이다. 이론적으로는 중요하지만 이 같은 변형들은 상당히 복잡하고 전혀 실질적인 것이 안된다.

연습 문제

1. M 보다 적은 것으로 된 부분 파일에 대해 삽입 정렬에서 잘라내는 것으로 된 재귀 퀵 정렬을 구현하고 1000개 요소들의 무작위 파일에서 가장 빠르게 처리되는 M 의 값을 실험적으로 결정하라.
2. 비재귀 구현에 대한 이전 문제를 푸시오.
3. 세 가지중의 중위수의 개선점을 통합함으로써 이전의 문제를 해결하라.
4. N 개 똑같은 요소들의 파일을 정렬하는데 퀵 정렬은 얼마나 걸리는가?
5. 가장 큰 요소가 이동이 될 때 퀵 정렬의 실행동안 시간의 최대 수는 어떻게 되는가?
6. 교재에서 제시된 두 가지 방법을 이용해서 파일 A B A B A B A를 어떻게 분할하는가를 보여라.
7. 퀵 정렬은 키 E A S Y Q U E S T I O N을 정렬하기 위해서 얼마나 많은 비교가 이용되는가?
8. 삽입 정렬이 퀵 정렬내에서부터 직접적으로 분리워진 경우에 얼마나 많은 “표지”적인 키들이 필요한가?
9. 퀵 정렬의 비재귀 구현에 대해 스택 대신에 큐를 이용하는 것이 합당한가? 그런 경우의 이유와 그렇지 않은 경우의 이유는?
10. 중위수와 같은 키들로 된 모든 요소들은 왼쪽에 더 적은 요소들로 그리고 오른쪽에 더 큰 요소들로 존재하기 위해서 파일을 재배열하는 프로그램을 기술하시오.

10 장

기수 정렬

많은 정렬 응용의 파일에 대해 레코드들의 순서를 정의하는데 이용되는 “키”들은 매우 복잡하다.(예를 들면, 전화 번호부나 도서관 목록에서 이용되는 순서화된 기능을 고려해 보자) 이것 때문에, 두 개 키를 “비교”하고 두 개 레코드를 “교환”하는 기본적인 연산에 의해서 정렬 방법을 정의하는 것이 합당하다. 연구되어진 대부분의 방법은 두 가지 기본적인 연산에 의해서 기술된다. 그러나 많은 응용에 대해, 키들이 어떤 제약된 범주에서 수들로 간주한다는 사실의 잇점을 취하는 것이 가능하다. 이같은 수들에 대해 디지털 성질의 잇점을 취하는 정렬 방법을 기수 정렬이라고 한다. 이같은 방법들은 키들을 비교하는 것이 아니라 키들의 부분들을 비교하는 것이다.

기수 정렬 알고리즘은 키들을 M (기수)의 다른 값에 대해 기저- M 진법으로서 표현된 수들로서 취급한다. 그리고 수들 개개의 디지털로서 처리한다. 예를 들면, 세 자리의 수들로 인 쇄되어진 카드 뭉치를 정렬해야만 하는 사무원을 고려해 보자. 그가 처리하는 한 가지 합당한 방법은 10개의 뭉치를 만드는 것이다. 즉, 100이하의 수들에 대해 한 개 뭉치, 100과 199 사이에 존재하는 수들에 대해 한 개 뭉치등등으로 구성하고, 뭉치들로 된 카드를 놓고 그리고 고 단지 몇 개의 카드들로만 존재한 경우에는 어떤 더 쉬운 방법을 이용하거나 혹은 다음 숫자에서도 앞의 같은 방법을 이용해서 개별적으로 처리를 한다. 이것은 $M = 10$ 으로 된 기수 정렬의 간단한 예제이다. 본 장에서는 이 방법과 다른 방법들에 대해서 자세히 보도록 하자. 물론, 대부분의 컴퓨터에서, $M = 10$ 이기 보다는 $M = 2$ (또는 다른 2의 멍승)인 것으로 처리하는 것이 편리하다.

디지털 컴퓨터 내부에 표현된 것을 이진수로서 취급하므로써, 많은 정렬 응용들은 이진수인 키들을 처리하는 기수 정렬의 이용을 실행 가능하게 하도록 다시 만들게 된다. 다행히도,

C++는 간단하고 효율적인 방법으로서 그런 연산을 구현하는 것이 가능한 저급 연산자를 제공한다. 많은 다른 언어(예를 들면, 파스칼)들은 수의 이진 표현에 의존하는 프로그램을 기술하는 것이 어렵게 된다.

비트(bits)

이진수가 주어진 경우(표현된 키), 기수 정렬에 대해 필요한 기본적인 연산은 수에서 연속적인 비트 집합으로 추출하는 것이다. 가령 우리가 알고 있는 키들이 0에서 100사이에는 정수형을 처리한다고 가정하자. 이것들은 10개 비트의 이진수로서 표현된다고 하자. 기계어에서, 비트들은 비트 연산인 “and”와 이동을 이용함으로써 이진 수에서 추출한다. 예를 들면, 10개 비트 수 중 앞의 두 개 비트는 8개 비트 위치를 오른쪽으로 이동시키므로써 추출되고 그 뒤에 마스크 값 “0000000011”을 비트 연산자 “and”와 수행하므로써 이루어진다. C++에서는 연산을 비트 처리 연산자인 >>와 &로서 직접적으로 구현한다. 예를 들면, 10개 비트로 된 수 x 의 앞의 두 개 비트는 $(x >> 8) \& 03$ 에 의해서 주어진다. 일반적으로 “모든 것을 0으로 하나 x 의 j 의 가장 오른쪽에 있는 비트들”은 $x \& \sim(\sim 0 < j)$ 로서 구현이 된다. 그 이유는 $\sim(\sim 0 < j)$ 는 가장 오른쪽의 j 비트에는 1로 그 이외에서는 0으로 마스크(mask)를 하기 때문이다.

이점에 대해 정렬 알고리즘의 구현은 비교 연산자 <, == 와 >가 정수형, 부동 소숫점이나 스트링과 같은 전형적인 키들에 대해 이용이 가능한 내재적인 가정으로서 정렬되어진 항목들의 키들 형(itemType)으로 기술하지 않고 남겨두었다. 기수 정렬 알고리즘에 대해, 비교 연산자를 이용하지는 않지만 그러나 C++는 정수형 키들에 대한 다음의 정의에서와 같이 이용하는 연산자를 허용케 된다.

```
class bitskey
{
private:
    int x;
private:
    bitskey & operator=(int i)
        { x = i; return *this; }
    nline unsigned bits(int k, int j)
```

```

        { return ( x>> k) & ~(~0 << j); }
};
typedef bitskey itemType;

```

이것은 itemType 키들에 대해 두 개 연산을 이용하도록 하는 것이다. 즉, 정수형 값과 bits에서 할당이다. 할당 코드는 표준적인 C++ 관용구이다. bits 연산자는 x의 오른쪽에서 k 비트를 나타내는 j 비트를 되돌려주도록 하기 위해 위에서 기술된 비트 처리 명령을 이용하는 것이다. 예를 들면, t가 형 itemType인 경우, 그때 문 t = 1000은 t의 데이터 필드에 단순히 1000(이진수로 1111101000)을 할당하는 것이고 그때 문 t.bits(2, 4)는 2(이진수로 10, t의 오른쪽에서 5번째와 6번째 비트)이다. 기수 정렬 알고리즘을 이용하기 위해서, 정렬된 키들이 비트 스트링으로서 논리적으로 나타내기 위해 bits 연산자를 지닌다. 키들로서 길이를 변화시키는 비트 연산자를 허용하는 방법 혹은 키에서 비트들의 최대 수와 같은 다른 것들이 그 형에 포함된다. 평상시와 같이, 알고리즘의 본질적인 성질에 초점을 맞추기 위해서, 상세한 것은 생략한다.

이같은 기본적인 도구로 무장함으로써, 키들의 비트들을 조사하는 순서가 다르게 되는 기수 정렬의 두 가지 형태를 고려하게 된다. 키들이 짧지가 않으므로 그것의 비트를 추출할 가치가 있다. 만약 키들이 짧으면, 그때 8장의 분배 계수 방법이 이용된다. 이 방법은 계수를 위해 크기 M의 표지 표와 레코드를 재 배열하는데 크기 N의 다른 보조 표를 이용해서 선형 시간내에 0에서 M - 1사이에 존재하는 정수형으로 알려진 N 키들을 정렬할 수가 있다. 이와 같이 만약 크기 2^b 의 표를 제공하는 경우, 그때 b-비트 키들은 선형시간으로 쉽게 정렬된다. 기수 정렬은 가능하지 않지만 키들이 충분히 긴 경우(가령 b = 32)로 될 수가 있다.

기수 교환 정렬(radix exchange sort)이라 부르는 기수 정렬의 첫 번째 기본적인 방법은 왼쪽에서 오른쪽으로 키들의 비트들을 조사하고, 퀵 정렬에 대해 비슷한 방법으로서 레코드를 처리한다. 일직선상의 기수 정렬(straight radix sort)이라 부르는 두 번째 기본적인 방법은 오른쪽에서 왼쪽으로 키들의 비트들을 조사하고, 여러 가지 합당한 환경에서 선형 시간으로 처리되도록 만든다.

기수 교환 정렬(Radix Exchange Sort)

파일의 레코드를 재배열하므로써 0 비트로 시작되는 키들 모두는 1 비트로 시작하는 키들 모든 것 전에 오게 된다고 가정하자. 이것은 퀵 정렬과 같이 재귀 정렬 방법으로 정의된다.

즉, 만약 두 개 부분 파일이 개별적으로 정렬되면, 그때 전체 파일은 정렬된다. 파일을 재배열하기 위해서, 1 비트로 시작하는 키를 찾기 위해서 왼쪽에서 조사를 하고, 0 비트로 시작되는 키를 찾기 위해서는 오른쪽에서 조사를 해서 그것을 교환하고, 포인터가 서로 엇갈릴때 까지 계속된다.

```
void radixexchange(itemType a[], int l, int r, int b)
{
    int i, j; itemType t;
    if ( r > l && b >= 0 )
    {
        i = l; j = r;
        while ( j != i )
        {
            while ( !a[i].bits(b, 1) && i < j ) i++;
            while ( a[j].bits(b, 1) && j > i ) j--;
            swap(a, i, j);
        }
        if ( !a[r].bits(b, 1) ) j++;
        radixexchange(a, l, j-1, b-1);
        radixexchange(a, j, r, b-1);
    }
}
```

부름 radixexchange(1, N, 30)은 만약 $a[1], \dots, a[N]$ 이 2^{32} 보다 적은 양의 정수인 경우 배열을 정렬하는 것이다.(그래서 그것은 31-비트 이진수로서 표현된다) 변수 b는 30(가장 왼쪽)에서 0(가장 오른쪽)으로 내려가는 범주에서 조사되는 비트들의 상태를 파악한다. 이용되는 실제 비트들의 수는 응용에서 간단한 방법으로 워드당 비트의 수와 정수형과 음수의 기계 표현에 의존한다.

이같은 구현은 9장에서의 퀵 정렬의 재귀 구현과 아주 비슷하다. 본질적으로, 기수 교환 정렬에서의 분할은 파일에서 어떤 수대신에 분할의 요소로서 수 2^b 를 사용하는 것을 제외하고 퀵 정렬의 분할과 같은 것이다. 2^b 가 파일내에 존재하지 않으면, 요소는 분할동안에 마지막 위치에 존재할 수는 없다. 또한 단지 한 비트가 조사하므로해서, 포인터 조사를 멈추기 위해서 표지에 의존할 수는 없다. 그러므로 해서, 테스트 ($i < j$)는 조사되는 반복에 포함된

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	E	O	L	M	I	N	G	E	A	X	T	P	R	S
A	E	A	E	G	I	N	M	L	O					
A	A	E	E	G										
A	A													
A	A													
		E	E	G										
		E	E											
						I	N	M	L	O				
						L	M	N	O					
						L	M							
								N	O					
										S	T	P	R	X
										S	R	P	T	
										P	R	S		
											R	S		

그림 10.1 기수 교환 정렬에서의 부분 파일

다. 이것은 비록 자신과 $a[i]$ 의 “교환”을 하는 경우 영향을 미치지 않지만, 퀵 정렬의 구현에서와 같이 break로서 피하는 경우, $(i == j)$ 에 대해 특별한 교환이 된다. 분할은 i 와 같은 j 그리고 b 번째 위치에서 1 비트를 지나는 $a[i]$ 의 오른쪽에 대한 모든 요소들 그리고 위치 b 에서 0을 지나는 $a[i]$ 의 왼쪽에 대한 모든 요소들로서 멈추게 한다. 위의 구현은 이 같은 경우를 처리하는 분할 반복후에 특별한 테스트를 지닌다.

그림 10.1은 키들의 예제 파일이 어떻게 분할되고 이 방법에 의해서 어떻게 정렬이 되는지를 나타낸 것이다. 비록 분할 방법의 연산이 키들의 이진 표현 없이는 완전히 불투명하지만, 퀵 정렬에 대한 그림 9.2와 비교된다. 그림 10.2는 키들의 이진 표현에 의해서 분할을 나타낸다. 수 i 의 이진 표현으로서 표현된 알파벳에서 i 번째 문자로 된 간단한 5개 비트 코드가 이용된다. 이것은 더 많은 비트를 이용하는(7개나 8개) 것과 더 많은 문자를 표현하는(대/소문자, 숫자, 특수 기호) 실제 문자 코드의 간단한 버전이다. 이같은 5개 비트 문자 코드에 대해 그림 10.1에서 키들을 변형시키는 것은 표를 압축시키는 것으로서 부분 파일 분할은 라인당 하나씩이라기 보다는 오히려 “병렬”로서 제시가 되고 그후에 열과 행을 바꾸므로 그림 10.2에서 키들 앞에 존재하는 비트들이 어떻게 분할을 제어하는 가를 제시한다. 이같은 그림에서, 크기 1의 부분 파일들이 만날 때 처럼 분할 과정을 제외하고 오른쪽에 있는 다음 그림에서 회색의 “1”인 부분 파일에 의해 따르는 흰색 “0” 부분 파일에 의한 각 분할이 지시된다.

이 예제에서 나타내지 않은 기수 정렬에 대한 한 가지 잠재적인 문제는 감소되는 분할이(이용된 비트에 대해 같은 값을 지닌 모든 키들로 된 분할) 가끔 발생된다. 이같은 상황은 적은 수들이(많은 앞의 0의 값을 지니것들로 된) 정렬되어질 때, 실제 파일에서 공통적으로 일어난다. 또한 문자에 대해서도 발생된다. 예를 들면, 32-비트 키들이 표준 8-비트 코드에서 각각을 암호화시키고, 그때 그것들을 함께 놓으므로서 4개 문자로 구성된다. 그때 감소되는 분할은 각 문자 위치의 시작에서 발생된다. 그 이유는 소문자들 모두는 대부분 문자 코드에서 같은 비트들로서 시작되기 때문이다. 많은 다른 비슷한 효과는 암호화된 데이터를 정렬할 때 관심 되는 것이다.

그림 10.2에서 한 번 키가 그것의 왼쪽 비트들에 의해서 모든 다른 키들과 구별이 되기만 하면, 더 이상 비트들은 조사가 되지 않는다. 이것은 다른 것에서는 단점이 되고 어떤 상황에서는 명백한 장점이 된다. 키들이 진정으로 무작위 비트들일 때, 각 키는 키들에서 비트의 수보다 훨씬 적은 것인 약 $\lg N$ 비트후에 다른 것들과 다를 수가 있다. 이것은 무작위 상황에서, 각 분할에서 부분 파일을 반으로 나누기 때문이다. 예를 들면, 1000개 레코드로 된 파일

A 00001	A 00001	A 00001	A 00001	A 00001	A 00001
S 10011	E 00101	E 00101	A 00001	A 00001	A 00001
O 01111	O 01111	A 00001	E 00101	E 00101	E 00101
R 10010	L 01100	E 00101	E 00101	E 00101	E 00101
T 10100	M 01101	G 00111	G 00111	G 00111	
I 01001	I 01001	I 01001	I 01001		
N 01110	N 01110	N 01110	N 01110	L 01100	L 01100
G 00111	G 00111	M 01101	M 01101	M 01101	M 01101
E 00101	E 00101	L 01100	L 01100	N 01110	N 01110
X 11000	A 00001	O 01111	O 01111	O 01111	O 01111
A 00001	X 11000	S 10011	S 10011	P 10000	
M 01101	T 10100	T 10100	R 10010	R 10010	R 10010
P 10000	P 10000	P 10000	P 10000	S 10011	S 10011
L 01100	R 10010	R 10010	T 10100		
E 00101	S 10011	X 11000			

그림 10.2 기수 교환 정렬("왼쪽에서 오른쪽" 기수 정렬)

을 정렬함은 각 키(키들이 존재한다 할지라도 가령 32-비트 키들)에서 약 10개나 7개 비트를 조사하는 것만 수반된다. 다른 한편, 똑같은 키들의 모든 비트들이 조사되는 것을 주시해야 한다. 기수 정렬은 많은 똑같은 키들이 포함되는 파일상에서는 잘 처리 되지 않는다. 기수 교환 정렬은 정렬되어진 키들이 진정으로 무작위 비트들로 구성이 되는 경우에 퀵 정렬보다 실제적으로 다소 빠르게 되나 퀵 정렬은 더 적은 무작위 상황에서도 적합하다.

기수 교환 정렬에 대해 분할 처리를 나타내는 트리인 그림 10.3은 그림 9.5와 비교된다. 이 같은 이진 트리에서, 내부 노드는 분할 점으로 나타내고, 외부 노드는 파일에서 키들로서 모두가 크기 1의 부분 파일로 끝이 난다. 17장에서 이같은 트리가 기수 교환 정렬과 기본적인 검색 방법사이의 직접적인 관계를 어떻게 나타내는 가를 보게 될 것이다.

위에 주어진 기본적인 재귀 구현은 퀵 정렬에서와 같이 재귀를 제거하고 적은 부분 파일을 다르게 취급하므로써 개선된다.

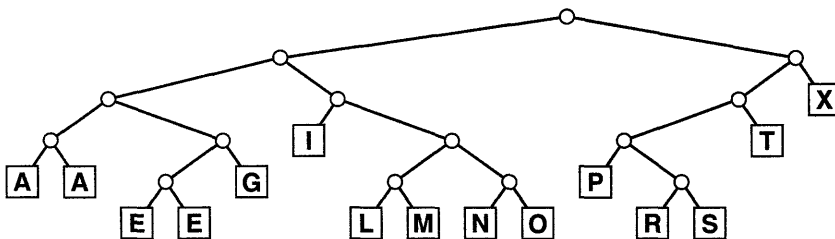


그림 10.3 기수 교환 정렬에서 분할 처리의 트리 다이어그램

일직선상의 기수 정렬(Straight Radix Sort)

또 다른 기수 정렬은 오른쪽에서 왼쪽으로 비트들을 조사하는 것이다. 이것은 옛날의 컴퓨터-카드-정렬기 기계에서 이용된 것이다. 즉, 카드 뭉치는 오른쪽에서 왼쪽으로 각 열에 대해 한 번씩 기계를 80번 수행을 하는 것이다. 그림 10.4는 오른쪽에서 왼쪽으로 비트단위로 기수 정렬이 예제 키들의 파일상에 어떻게 수행되는 가를 나타낸 것이다. 그림 10.4에서 i 번째 열은 키들의 마지막 i 비트상에서 정렬되고 i 번째 비트에서 0으로 된 모든 키들을 추출하므로써 $(i - 1)$ 번째 열에서 이끌어지고 그때 i 번째 비트에서 1로된 모든 키들이 된다.

방법이 처리되는 것을 확인하기는 쉽지 않다. 사실, 한 비트 분할 처리가 안정성이 있지 않으면 그것은 전혀 처리가 되지 않는다. 한 번 안정성이 중요한 것으로 인식되면, 방법의 사소한 증명을 찾을 수 있다. 즉, i 번째 비트 1로서 된 것들(안정성 방법으로) 이전에 i 번째 비트 0으로 된 키들을 놓은 후에, 어떤 두 개 키는 파일에서 적절한 순서(지금까지 조사된 비트들의 기본으로)로서 나타남을 볼수가 있다. 이같은 이유는 분할이 적절한 순서로 놓는 경우에 i 번째 비트가 서로 다르기 때문이거나 혹은 안정성 때문에 적절한 순서로서 되어지는 경우에서 i 번째 비트들이 같기 때문이다. 예를 들면, 안정성 수단의 요구사항은 기수 교환 정렬에서 이용된 분할 방법이 오른쪽에서 왼쪽 정렬에 대해 이용되지 않는 것이다.

A 00001	R 10010	T 10100	X 11000	P 10000	A 00001
S 10011	T 10100	X 11000	P 10000	A 00001	A 00001
O 01111	N 01110	P 10000	A 00001	A 00001	E 00101
R 10010	X 11000	L 01100	I 01001	R 10010	R 10010
T 10100	P 10000	A 00001	A 00001	S 10011	G 00111
I 01001	L 01100	I 01001	R 10010	T 10100	I 01001
N 01110	A 00001	E 00101	S 10011	E 00101	L 01100
G 00111	S 10011	A 00001	T 10100	E 00101	M 01101
E 00101	O 01111	M 01101	L 01100	G 00111	N 01110
X 11000	I 01001	E 00101	E 00101	X 11000	O 01111
A 00001	G 00111	R 10010	M 01101	I 01001	P 10000
M 01101	E 00101	N 01101	E 00101	L 01100	R 10010
P 10000	A 00001	S 10011	N 01101	M 01101	S 10011
L 01100	M 01101	O 01101	O 01101	N 01101	T 10100
E 00101	E 00101	G 00111	G 00111	O 01101	X 11000

그림 10.4 일직선상의 기수정렬("오른쪽에서 왼쪽" 기수정렬)

분할은 단지 두 개 값으로 된 파일을 정렬하는 것과 같고 8장에서 분배 계수 정렬은 이것들에 대해 전적으로 적절하다. 만약 분배 계수 프로그램에서 $M = 2$ 이고 $a[i]$ 를 $\text{bits}(a[i], k, 1)$ 로서 대체한다고 가정하면, 그 프로그램은 오른쪽에서 비트 k 위치에서 배열 a 의 요소를 정렬하고 결과를 임시 배열인 b 에 놓는 방법이다. 그러나 $M = 2$ 를 이용하는 이유는 없다. 즉, 실제로 M 계수의 표가 필요하다는 인식에서 M 은 가능한 크게 만들게 된다. 이것은 $M = 2^m$ 인 정렬 동안에 한 번에 m 비트를 이용하는 것에 대응된다. 이와 같이 일직선상의 기수 정렬은 가장 오른쪽 비트상에 $a[1], \dots, a[N]$ 을 정렬함에 다음 구현에서처럼 분배 계수 정렬의 일반화된 것 이상 아니다.

```
void straightradix(itemType a[], itemType b[], int N)
{
    int i, j, pass, count[M-1];
    for ( pass = 0; pass < w/m; pass++ )
    {
        for ( j = 0; j < M; j++ ) count[j] = 0;
        for ( i = 1; i <= N; i++ )
            count[a[i].bits(pass*m, m)]++;
        for ( j = 1; j < M; j++ )
            count[j] += count[j-1];
        for ( i = N; i >= 1; i-- )
            b[count[a[i].bits(pass*m, m)]--] = a[i];
        for ( i = 1; i <= N; i++ ) a[i] = b[i];
    }
}
```

이같은 구현은 정렬되어질 배열 뿐만 아니라 입력 파라미터로서 임시적인 배열을 전달한다고 가정하자. 비록 어떤 프로그래밍 환경에서 m 과 M 사이에 차이점을 말할 수 없다고 경고하지만 $M = 2^m$ 은 변수명에서 유지가 된다.

위의 프로시저는 w 가 m 의 배수인 경우에만 적절히 처리된다. 정상적으로, 이것은 기수 정렬에 대한 특별히 엄격한 가정은 아니다. 즉, 똑같은 크기 조각들의 정수 수로 정렬이 된 키들을 나누는 것에 단순히 대응된다. $m == w$ 일 때, 분배 계수 정렬을 지닌다. 즉, $m == 1$ 일 때, 일직선상의 기수정렬을 지니고 오른쪽에서 왼쪽으로 비트단위로 된 기수 정렬이 위의 예제에서 기술된다.

위의 구현은 각 분배 계산 과정동안에 a에서 b에 이르는 파일을 이동하고 단순한 반복으로서 a로 되돌아 간다. 이같은 “배열 복사” 반복은 분배 계수 코드의 두 개 복사 즉, 하나는 a에서 b로 정렬하는 것이고 다른 하나는 b에서 a로 정렬하는 것으로써, 요구되는 경우를 제거한다.

기수 정렬의 활용도 특성

b 비트 키들로서 N 개 레코드를 정렬하는 기본적인 두 개의 기수 정렬에 대한 실행시간은 본질적으로 Nb 이다. 한편으로는 이같은 실행시간을 수들이 모두 다른 경우에 b 는 적어도 $\log N$ 이므로서, $N \log N$ 과 본질적으로 같은 것으로 간주한다. 다른 한편으로는 두 개 방법이 Nb 연산보다 적은 것을 이용케 된다. 즉, 키들 사이에 차이가 발견되면 멈출 수가 있기 때문에 왼쪽에서 오른쪽의 방법과 단지 한 번에 많은 비트를 수행하므로써 오른쪽에서 왼쪽으로의 방법이 있다.

성질 10.1 기수 교환 정렬은 평균의 경우 약 $N \lg N$ 비트이다.

만약 파일의 크기가 2의 멍승이고 비트들이 무작위로 존재하면, 앞의 비트 반은 0으로 그리고 반은 1로 기대된다. 그래서 재발생 $C_N = 2C_{N/2} + N$ 은 9장에서 퀵 정렬에 대한 것과 같이 활용도를 기술한 것이다. 다시, 상황의 이같은 기술은 분할이 평균의 경우(그리고 키들에서 비트들의 수가 유한이기 때문에)에서만 중앙에서 존재하므로 정확하지 않다. 그러나, 이같은 모델에서 분할은 퀵 정렬에서 보다 중앙에 존재하므로써 나타난 결과는 사실이 된다.(이 범주를 벗어난 상세한 분석은 이것을 증명하는데 요구된다) \square

성질 10.2 두 개의 기수 정렬은 N 개 b -비트 키들을 정렬하기 위해서 Nb 보다 더 적게 된다.

다시 말해서, 기수 정렬은 취해진 시간이 입력의 비트 수에 비례를 한다는 의미에서 선형이다. 이것은 프로그램의 시험에서부터 직접적으로 이끌어진다. 즉, 어느 비트도 한 번 이상 조사 되지는 않는다. \square

큰 무작위 파일에 대해, 기수 교환 정렬은 그림 9.6에서 처럼 퀵 정렬과 같이 처리된다. 그러나 일직선상의 기수 정렬은 오히려 다르게 처리된다. 그림 10.5는 5개 비트 키들의 무작위

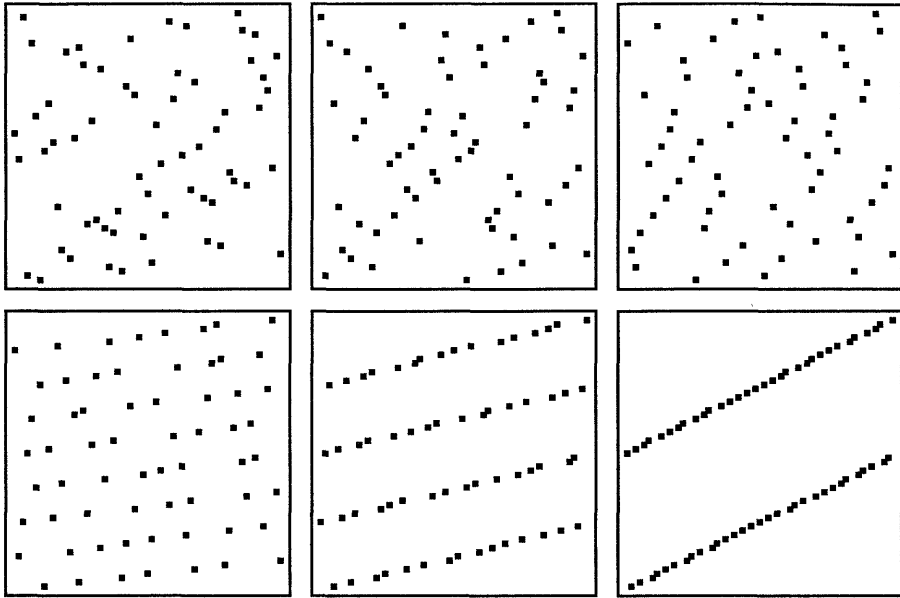


그림 10.5 일직선상의 기수 정렬

파일에 일직선상의 기수 정렬의 단계를 나타낸다. 정렬 동안에 파일의 진보적인 구조는 이같은 다이어그램에서 명확하게 나타낸다. 예를 들면, 세 번째 단계 뒤에서(왼쪽 밑), 파일은 4개 서로 혼합된 정렬 부분 파일들로 구성된다. 즉, 00으로 시작된 키들(밑의 자국), 01로 시작되는 키들 등으로 된다.

성질 10.3 일직선상의 기수 정렬은 2^m 계수기에 대해 특별한 공간을 이용해서(그리고 파일을 재배열하는 버퍼) b/m 과정에 b -비트 키들로 된 N 개 레코드를 정렬할 수가 있다.

이 증명은 구현에서 간단히 이루어진다. 특히, 너무 많은 특별한 기억장소를 이용하지 않고 $m = b/4$ 를 취하는 경우, 선형 정렬을 얻을 수가 있다! 이같은 성질의 실제적인 분기는 다음에 더 자세히 설명한다. □

선형 정렬(Linear Sort)

이전 절에서 주어진 일직선상의 기수 정렬 구현은 파일을 통해서 b/m 과정이 된다. m 을 크게 하므로해서, 사용가능한 기억 장소가 $M = 2^m$ 을 지니는 한 더 효율적인 정렬 방법을 얻

을 수가 있다. 합당한 선택은 m 을 약 $1/4$ 워드 크기($b/4$)로 하는 것이다. 그래서 기수 정렬은 4가지 배분 계수 과정을 지닌다. 키들은 기저- M 수들로서 취급을 하고 각 키의 각(기저- M) 디지트는 조사가 되나 거기에는 키당 단지 4개의 디지트만을 지닌다.(이것은 많은 컴퓨터의 건축적 구조에 직접적으로 대응된다. 즉, 전형적인 구조는 32-비트 워드를 지니고 각각은 4개의 8-비트 바이트로 구성된다. bits 프로시저는 컴퓨터에 효율적으로 수행이 되는 경우의 워드에서부터 특별한 바이트를 추출하는 것으로 마무리 된다) 지금 각 배분 계수 과정은 선형이고 그것들중 단지 4가지만이 지니므로, 전체 정렬은 선형적이고 확실히 가장 좋은 활용도인 것이다.

사실, 단지 두 개의 배분 계수 과정들로서 얻을 수 있다.(비록 이같은 시기에 왼쪽에서 오른쪽을 말하는 것은 어렵지만 몇 가지 노력으로 인해서 이 방법을 이해하는 것이 필요하다) 파일은 b -비트 키들 앞의 $b/2$ 비트들로만 이용되는 경우에 거의 정렬이 된다는 사실의 장점으로 수행한다. 퀵 정렬에서와 같이, 정렬은 전체 파일에 삽입 정렬을 이용하므로써 효율적으로 완전하게 될 수가 있다. 이 방법은 위의 구현에 사소한 변형인 것이다. 즉, 키들 앞의 반쪽을 이용해서 오른쪽에서 왼쪽 정렬을 수행하는 것은 $pass=0$ 이기 보다는 오히려 $pass=b/(2*m)$ 에서의 외부 반복을 단순히 시작케 된다. 그때 편리한 삽입 정렬은 결과로된 거의 순서화된 파일상에서 이용된다. 파일을 앞에 존재하는 비트들상에 정렬된다고 확신하도록 하기 위해서, 그림 10.2의 첫 번째 몇 가지 안되는 열들을 조사케 된다. 예를 들면, 이미 정렬된 첫 번째 세 가지 비트상에서 처리되는 삽입 정렬은 단지 6번의 교환만을 요구한다.

두 개 배분 계수 과정을 이용함(약 $1/4$ 워드 크기인 m)과 그때 작업을 마치기 위해서 삽입 정렬을 이용하는 것은 키들이 무작위 비트들인 큰 파일에 대해 보았던 다른 것들의 어떤 것도 보다 빠르게 처리하는 정렬 방법이다. 그것의 주된 단점은 정렬되어진 배열로서 같은 크기의 특별한 배열을 요구하는 것이다. 연결 리스트 기법을 이용해서 특별한 배열을 제거하는 것이 가능하지만 그러나 N 에 비례하는 특별한 공간(연결에 대해)이 여전히 요구된다.

선형 정렬는 많은 응용에 대해 명백히 바람직하나 그러나 보이는 것과 같이 만병 통치 약은 아닌 이유에는 여러 가지가 있다. 첫째, 그것의 효율성은 순서적으로 무작위 비트들인 키들에 의존을 하는 것이다. 만약 이같은 조건이 만족되지 않으면, 심각히 절하된 활용도와 비슷하다. 둘째, 정렬된 배열의 크기에 비례되는 특별한 공간이 요구된다. 셋째로, 프로그램의 “내부 반복”은 비록 그것이 선형이고 아주 큰 파일(특별한 배열이 실제로서 믿을 수 있는 점에서)에 대해서는 제외하고 기대한 것과 같이 퀵 정렬보다 훨씬 빠르지는 않는다.

기수 정렬은 그것의 생존성이 많은 다양한 응용에 적합하기 때문에 아주 널리 이용되는 퀵 정렬과 같은 “범용” 알고리즘과 대조적으로 키들의 특별한 성질들에 의존하게 때문에 “특수 목적” 접근 방법으로서 특성화 된다. 퀵 정렬과 기수 정렬사이의 선택은 키, 레코드와 파일 크기와 같은 응용의 특징 뿐만 아니라 개개 비트들의 이용과 접근의 효율성에 관계되는 프로그래밍과 기계 환경의 특징에 의존된다. 적절한 응용에 대해, 기수 정렬은 퀵 정렬보다 훨씬 더 빠르게 즉, 아마도 두배 정도로 빠르지만 그러나 공간이 문제가 되거나 혹은 키들이 유동 크기이거나 혹은 반드시 무작위일 필요가 없는 경우에 문제가 될 소지는 없다.

연습 문제

1. 파일 001, 011, 101, 110, 000, 001, 010, 111, 110, 010에 대한 퀵 정렬로서 이용된 수와 기수 정렬로 이용한 교환의 수를 비교하라.
2. 퀵 정렬에 대해서와 같이 기수 교환 정렬에서의 재귀를 제거하는 것이 중요하지 않은 이유는?
3. 모든 키들상에서 동일한 앞의 비트들을 넘여가기 위한 기수 교환 정렬을 변경하시오. 어느 상황에서 이것이 가치가 있는가?
4. 다음 사실이 맞는가 틀린가: 일직선상의 기수 정렬의 실행시간은 입력 파일에서 키들의 순서에 의존하지 않는다. 답을 설명하시오.
5. 기수 교환 정렬과 일직선상의 기수정렬에서 어느 방법이 모두 같은 키들의 파일에 대해 빠른가?
6. 다음 사실이 맞는가 틀린가: 일직선상의 기수 정렬과 기수 교환 정렬 둘다가 파일에서 모든 키들의 비트를 조사한다. 답을 설명하시오.
7. 특별한 기억장소 요구사항은 별개로 하고, 키들 앞의 비트들에서 일직선상의 기수 정렬을 수행하는 방법의 주된 단점이 무엇이고 그 후에 삽입 정렬로서 단점을 제거 할 수가 있는가?
8. N 개의 b -비트 키들의 4개 과정 일직선상의 기수 정렬을 수행하는데 얼마나 많은 기억장소를 요구하는지를 설명하시오.
9. 어떤 형태의 입력 파일에서 기수 교환 정렬이 대부분 느리게 처리되는가?(매우 큰 N 에 대해)
10. 10,000개의 32-비트 키들의 무작위 파일에 대해 기수 교환 정렬과 일직선상의 기수 정렬을 실험적으로 비교하라.

11 장

우선순위

많은 응용에서, 키들로 된 레코드들이 순서적으로 처리되나, 반드시 완전히 정렬되어진 것은 아니고 반드시 한 번에 모든 것을 처리할 필요는 없다. 가끔 레코드의 집합을 수집해서 가장 큰 것을 처리하고 그리고 아마도 더 많은 레코드를 수집해서 다음으로 큰 것을 처리하는 등등이다. 그런 환경에서 적절한 데이터 구조는 새로운 요소를 삽입시키고 가장 큰 요소를 삭제 시키는 연산을 지원하는 것이다. 큐(가장 오래된 것을 삭제)와 스택(가장 최근의 것)과 대조적인 자료구조를 우선순위 큐(priority queue)라고 부른다. 사실, 이같은 자료구조가 적절한 우선순위 할당을 이용해서 우선순위로 구현되므로 해서, 우선순위 큐는 스택과 큐(그리고 다른 간단한 자료 구조)의 일반화로서 간주된다.

우선순위 큐의 응용은 모의 실험 시스템,(키들이 순서적으로 처리되는 “매 사건들”에 대응되는 곳) 컴퓨터 시스템에서 작업 스케줄링(키들은 어느 사용자가 먼저 처리되는가를 나타내는 “우선순위”에 대응되는 곳) 그리고 수치 계산(키들이 계산적인 오류를 지닌 것이므로 해서 가장 큰 것이 먼저 처리되는 곳)을 포함한다.

본 교재의 나중에, 더 고급화된 알고리즘에 대한 기본적인 설정 블록으로서 우선순위 큐를 어떻게 이용하는가를 볼 것이다. 22장에서는 이장에 있는 루틴을 이용해서 파일 축약 알고리즘을 개발하고, 31장과 33장에서는 우선 순위 큐가 여러 가지 기본적인 그래프-검색 알고리즘에 대한 기본적인 것으로서 어떻게 서비스 되는 가를 나타낸다. 이것들은 알고리즘 설계에서 기본적인 도구로서 우선순위 큐에 의해 중요한 역할을 하는 몇 가지 예제들이다.

위에 언급된 것들과 같은 응용에 대해 그것들을 효과적으로 이용하고 유지하기 위해서 우선 순위 큐상에 수행되는데 필요한 여러 가지 연산이 있기 때문에, 우선순위 큐를 어떻게 처

리하는가에 관해서 다소 더 상세한 것이 필요하다. 정말로, 우선 순위 큐가 그렇게 유용한가의 주된 이유는 키들로서 레코드 집합을 효율적으로 수행하는 여러 가지 다른 연산들의 다양성을 허용함에 유연성이 있는 것이다. 수치 키(우선순위들)들로 된 레코드를 포함하고, 다음 연산들의 몇 가지를 지원하는 자료구조를 설정하고 유지시키기를 원한다.

N 개 주어진 항목들에서 우선순위를 구성(Construct)

새로운 항목을 삽입(Insert)

가장 큰 항목을 제거(Remove)

가장 큰 항목과 새로운 항목(새로운 항목이 더 큰 경우가 아니면)을 대체(Replace)

항목의 우선순위를 변경(Change)

임의로 설정된 항목을 삭제>Delete)

두 개의 우선순위 큐를 하나의 더 큰 것으로 결합(Join)

(만약 레코드들이 똑 같은 키를 지니면, 가장 큰 키값으로 된 “레코드를 의미하기 위해서” 가장 큰 것을 취한다)

대체 연산은 제거뒤에 삽입을 하는 것과 같다.(삽입/삭제는 우선순위 큐를 일시적으로 한 요소씩 증가시키도록 하는 차이가 있다) 즉, 삽입뒤에 따르는 제거와는 아주 다름을 주시해야한다. 이것은 우선 순위 큐의 어떤 구현이 대체 연산을 아주 효과적으로 수행하기 때문에 분리된 능력으로서 포함된다. 비슷하게, 변경 연산은 삽입뒤에 따르는 삭제로서 구현되고, 구성 연산은 삽입 연산의 반복된 이용으로서 구현된다. 그러나 이같은 연산들은 자료구조의 어떤 선택에 대해 고급화된 자료구조를 직접적으로 구현한다. 결합 연산은 효율적인 구현에 대해 고급화된 자료 구조를 요구한다. 즉, 처음 5개 연산의 효율적인 구현을 허용하기 위해서 힙(heap)라 부르는 “고전적인” 자료 구조에 집중을 할 수가 있다.

위에 기술된 우선 순위 큐는 3장에서 기술된 추상화된 자료구조(abstract data structure)의 훌륭한 예제이다. 즉, 어떤 특별한 구현에서 데이터가 어떻게 구성되고 처리 되는 가와는 별개로 그것에 수행하는 연산들에 의해서 매우 잘 정의된다.

우선 순위 큐의 다른 구현은 비용 교환으로 수행된 여러 가지 연산에 대해 다른 활용도 특징이 수반된다. 정말로, 활용도 차이는 추상화된 자료 구조 개념에서 제기되는 유일한 차이이다. 첫째, 우선순위 큐들을 구현하는데 몇 가지 기초적인 자료구조를 논하므로써 잇점을 나타낸다. 다음으로, 더 고급화된 자료 구조를 조사하고 여러 가지 연산이 이같은 자료구조를

효율적으로 이용하여서 어떻게 구현되는가를 나타내는 것이다. 그리고 이같은 구현들에서 당연히 따르는 중요한 정렬 알고리즘을 보게 된다.

기초적인 구현들

우선 순위 큐를 구성하는 한 방법은 키 값들에 주의를 기울이지 않고도 배열 $a[1], \dots, a[N]$ 에서 항목을 단순히 유지하으로써 비순서화된 리스트로서 존재하는 것이다.(평상시와 같이, 그것이 필요한 경우 표지 값에 대해 $a[0]$ 와 가능하면 $a[N+1]$ 를 유지한다) 배열 a 와 크기 N 는 우선 순위 함수에 의해서만 언급하고 부르는 루틴에서 “숨겨진”것으로 된다고 가정하자. 비순서화된 리스트를 구현하는 배열로서, 우선 순위 큐 class는 다음과 같이 쉽게 구현된다.

```
class PQ
{
private:
    itemType *a;
    int N;
public:
    PQ(int max)
        { a = new itemType[max]; N = 0; }
    ~PQ()
        { delete a; }
    void insert(itemType v)
        { a[++N] = v; }
    itemType remove()
        {
            int j, max = 1;
            for ( j = 2; j <= N; j++)
                if ( a[j] > a[max] ) max = j;
            swap( a, max, N);
            return a[N--];
        }
};
```


삽입을 위해서, N 를 증가시키고 새로운 항목을 $a[N]$ 에 상수 시간 연산으로 삽입시킨다. 그러나 제거는 선형시간이 걸리는(배열에서 모든 요소들이 조사되어야하기때문) 가장 큰 키로된 요소를 찾기 위해서 배열을 통해서 조사되는 것 그리고 가장 큰 키로서 된 요소와 $a[N]$ 를 교환하고 N 를 감소시키는 것을 요구한다. 대체의 구현은 아주 비슷하므로 생략된다.

변경 연산을 구현하기 위해서는($a[k]$ 에서 항목의 우선순위를 변경), 단순히 새로운 값을 저장하고 그리고 $a[k]$ 에서 항목을 삭제하기 위해서는 제거의 마지막 라인에서 처럼 그것과 $a[N]$ 를 교환하고 N 를 감소한다. 특별한 데이터 항목을 언급하는 그런 연산은 참조가 자료구조에서 그것의 위치에 각 항목에 대해 유지되는 곳에 “포인터”나 “간접적인” 구현으로서 만 이해를 한다. 그런 구현은 이 장의 끝에 주어진다.

이용된 또 다른 기본적인 구조는 순서화된 리스트로서 다시 배열 $a[1], \dots, a[N]$ 를 이용하나 그것들의 키들이 증가되는 순서로서 항목을 유지시키도록 하는 것이다. 제거는 단순히 $a[N]$ 를 되돌리고 N 를 감소시키는 것(상수 시간 연산)을 수반한다. 그러나 삽입은 선형시간이 걸리는 배열의 오른쪽 한 위치에서 더 큰 요소들을 이동하는 것이 수반된다. 그리고 구성은 정렬을 포함한다.

어떤 우선 순위 큐 알고리즘은 정렬되어질 모든 항목들을 포함한 우선 순위 큐를 설정하기 위해서 삽입을 반복적으로 이용하고 그후에 역 순으로 항목을 받기 위해서 우선 순위 큐를 비우도록 제거를 반복적으로 이용하므로써 정렬 알고리즘으로 바뀌게 된다. 이같은 방법으로 비 순서화된 리스트로서 표현된 우선 순위 큐를 이용함은 선택 정렬에 대응된다. 즉, 순서화 된 리스트는 삽입 정렬에 대응된다.

연결 리스트는 또한 비순서화된 리스트에 대해 혹은 위에 주어진 배열 구현에서 보다 오히려 순서화된 리스트에 대해 이용된다. 이것은 삽입, 제거나 대체에 대해서 근본적인 활용도 특징을 변경시키지 않으나 상수시간으로서 삭제와 결합을 행하는 것이 가능토록 한다. 이같은 구현은 3장에서 주어진 기본적인 리스트 연산과 유사하고 검색 문제(주어진 키로서 레코드를 찾는 것)에 대한 비슷한 방법들의 구현이 14장에서 주어지기 때문에 여기서는 생략을 한다.

평상시와 같이, 많은 실제적인 상황에서 더 복잡한 방법을 수행하기 때문에 이같은 간단한 구현을 명심하는 것이 현명하다. 예를 들면, 비순서화된 리스트 구현은 순서화된 리스트가 삽입된 항목들이 우선 순위 큐에서 가장 큰 요소에 근접된 경향이 있는 경우에 적절한 반면에 단지 몇 가지 “가장 큰 요소를 제거”하는 연산은 삽입과 반대되는 것으로 수행되는 응용에서 적절하다.

히프 자료구조

우선 순위 큐 연산을 지원하는 자료구조는 각 키가 두 개 다른 특별한 위치에 있는 키들보다 더 큰 것으로 된 방법으로서 배열에서 레코드들을 저장하는 것이 수반된다. 차례로, 이 같은 키들의 각각은 두 개 이상의 키들보다 더 큰 것등등이다. 이같은 순서는 그림 11.1에서 처럼 더 적은 것으로 알려진 두 개의 키들에 각 키에서부터 라인들로 된 2차원 트리 구조로서 배열을 그리는 경우를 보는 것은 매우 쉽다.

4장을 회고해보면, 이같은 구조를 “정(complete) 이진 트리”라고 부른다. 즉, 한 노드를 놓고(근이라 부르는 것) 아래로 내려가면서 왼쪽에서 오른쪽으로 처리를 하고 N 노드들이 놓여질때까지 이전 레벨상에 각 노드아래의 두 노드를 연결시키므로써 구성된다. 각 노드아래의 두 노드를 자식(children)이라 부른다. 각 노드 위의 노드는 부모(parent)라고 부른다. 지금 히프 조건을 만족시키는 트리에서 키들을 원한다. 즉, 각 노드에서의 키는 자식(만약 존재하는 경우)에 있는 키들보다 큰 경우(또는 같거나)이다. 이것은 가장 큰 키들이 근에 존재하는 것을 의미한다.

그림 11.1에서 숫자화 된 것처럼 단순히 근을 위치 1에 놓고, 그것의 자식을 위치 2 와 3에 놓고, 다음 레벨에 있는 노드들을 위치 4, 5, 6과 7에 놓는등으로 인해서 배열내에서 정 이진 트리를 순서적으로 표현한다. 예를 들면, 위의 트리에 대한 배열 표현은 그림 11.2에 제시가 되어져 있다.

이같은 당연한 표현은 한 노드에서 부모와 자식에 이르는 것을 얻기가 쉽기 때문에 유용하다. 위치 j 에 있는 노드의 부모는 위치 $j/2$ 에 있고(j 가 홀수이면 가장 가까운 정수), 거꾸로 위치 j 에서 노드의 두 자식은 위치 $2j$ 와 $2j + 1$ 에 존재한다. 이것은 트리가 연결 표현으로서 구현된 것 보다.(각 요소는 포인터를 부모와 자식을 포함하도록 하는 것) 훨씬 쉽게 트

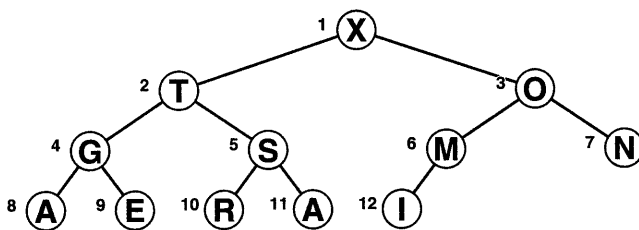


그림 11.1 히프의 정 이진 트리 표현

k	1	2	3	4	5	6	7	8	9	10	11	12
a[k]	X	T	O	G	S	M	N	A	E	R	A	I

그림 11.2 힙의 배열 표현

리를 운행할 수 있다. 배열로서 표현된 정 이진 트리의 엄격한 구조는 자료구조로서 그것들의 유용성을 제한하나 효율적인 우선 순위 큐 알고리즘의 구현을 허용하도록 하는 충분한 유연성을 제공한다. 힙은 모든 노드들이 힙 조건을 만족하는 배열로 표현된 정 이진 트리이다. 특히, 가장 큰 키는 배열에서 항상 첫 번째 위치에 존재케 된다.

모든 알고리즘들은 힙의 근에서 제일 밑에까지 이르는 경로(path)를 따라서 처리된다.(부모에서 자식으로 혹은 자식에서 부모로 이동하므로써) N 개 노드를 지닌 힙에서, 모든 경로들은 $\lg N$ 노드를 지님을 보는 것은 쉽다.(제일 밑에는 약 $N/2$ 개의 노드가 있고, 제일 밑에서 바로 위의 레벨에 있는 노드는 약 $N/4$ 개이고 그 바로 위의 노드들은 약 $N/8$ 개로 구성되는 등이다. 각 “세대”는 다음 세대의 노드들보다 약 $1/2$ 개정도를 지니고 이것은 기껏해야 $\lg N$ 세대를 지님을 의미한다) 이와 같이 모든 우선 순위 큐 연산들은(결합은 제외) 힙를 이용해서 대수 시간으로서 수행된다.

힙에서의 알고리즘

힙에서의 우선 순위 큐 알고리즘은 힙 조건을 위반하는 간단한 구조적 변경을 먼저 해주고 그후에 힙 조건이 어느곳에서든지 만족되도록 변경시켜서 운행하도록 처리한다. 밑에서 위로 가는, 다른 것은 위에서 아래로 가는 힙를 통해서 몇몇 알고리즘은 운행한다. 알고리즘 모두에서, 정수 N 으로서 유지된 힙의 현재 크기는 어떤 최대 크기 배열 a 에 저장된 한 개 워드의 정수형 키들이 레코드임을 가정하자. 위 처럼, 배열과 크기는 우선 순위 큐 루틴에 서로 다른 것으로 가정된다. 즉, 데이터는 서브루틴 부름을 통해서만 사용자와 우선 순위 큐사이에 전달된다.

힙를 생성하기 위해서는 먼저 삽입 연산을 구현할 필요가 있다. 이같은 연산은 힙의 크기를 단지 한 개 증가시키므로 해서, N 는 증가된다. 그때 삽입될 레코드는 $a[N]$ 로 놓으나 이것은 힙 성질을 위배케 된다. 만약 힙 성질이 위배되면(새로운 노드가 부모보다 큰 경우), 위배된 것은 새로운 노드와 부모를 교환하므로써 고쳐 나간다. 차례로 이것은 조건을 위

배케 하고 같은 방법으로 고쳐나가면 된다. 예를 들면, P가 위의 힙에 삽입된 경우, M의 왼쪽 자식으로서 a[N]에 먼저 저장된다. 그리고 M가 그것보다 크므로써, M와 교환을 하고, 그것이 O보다 크므로써 O와 교환을 하고나서 X보다 적을때 처리는 종료된다. 이같은 결과가 그림 11.3에 제시가 되어져 있다.

이같은 방법의 코드는 간단하다. 즉, 다음 구현은 새로운 항목을 N에 놓은 후에 힙 조건을 위배된 것을 고치기 위해서 upheap(N)를 이용한다.

```
void PQ :: upheap(int k)
{
    itemType v;
    v = a[k]; a[0] = itemMax;
    while ( a[k/2] <= v )
        { a[k] = a[k/2]; k = k/2; }
    a[k] = v;
}

void PQ :: insert(itemType v)
{ a[++N] = v; upheap(N); }
```

만약 k/2를 이 프로그램의 어느 곳에서든지 k-1로 대체를 하면, 본질적으로 삽입 정렬의 한 단계를 지닌다.(순서화된 리스트로서 우선 순위 큐를 구현) 여기서, 대신에 N에서 근에 이르는 경로를 따라서 새로운 키를 “삽입”하는 것이다. 삽입 정렬로서, v가 항상 교환에서 수반되기 때문에 반복내에서 완전한 교환을 수행하는 것이 불필요하다. 표지 키는 v가 힙에 있는 모든 키들보다 더 큰 경우에 대해 반복을 멈추도록 하기 위해서 a[0]에 놓아야만 한다. 아래에서 a[0]의 다른 이용을 보게 된다.

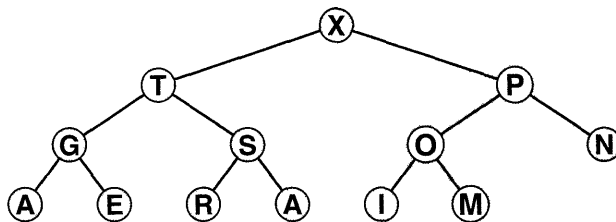


그림 11.3 새로운 요소(P)를 힙에 삽입

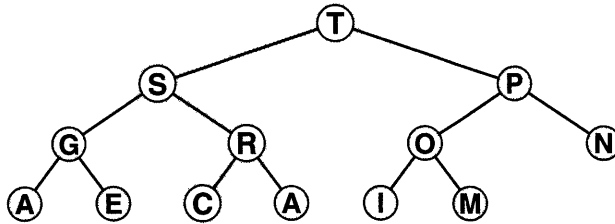


그림 11.4 힙에서 가장 큰 키를 대체(C로).

대체 연산은 새로운 키를 근에 놓고, 힙 조건을 재 저장시키기 위해서 제일 위에서 제일 아래로 힙을 이동시킨다. 예를 들면, 만약 위의 힙에서 X가 C로 대체가 되면, 첫 번째 단계는 근에 C를 저장하는 것이다. 이것은 힙 조건을 위배한 것이나, 위배는 C와 T를 교환함으로써 고쳐지게 된다. 이것은 두 자식중의 더 큰 것인 것(여기서는 S)과 C를 교환함으로써 고쳐지게 된다. 이같은 과정은 힙 조건이 C에 의해서 차지한 노드에 더 이상 위배된 것이 없을때까지 계속된다. 예제에서, C는 그림 11.4에서 나타난 힙을 남겨두면서 힙의 제일 밑에 이르는 모든 방법들로 만들게 한다.

“가장 큰 것을 제거”하는 연산은 같은 과정을 거친다. 힙은 연산후에 한 요소가 더 적은 것이므로써, 마지막 위치에 저장된 요소에 대해 장소가 없는 것을 남겨두고서 N를 감소하는 것이 필요하다. 그러나 가장 큰 요소($a[1]$ 에 존재)는 제거가 되어지고 그래서 제거 연산은 $a[N]$ 에 존재하는 요소를 이용해서 대체로 이끌어진다. 그림 11.5에 제시된 힙은 T를 M와 대체시키고, M보다 더 적은 양쪽 자식으로 된 노드에 도달될 때까지 아래로 이동을 하고 두 자식중의 더 큰 것을 찾으므로써 그림 11.4의 힙에서 T를 제거시키는 결과가 된다.

이같은 연산들 둘다의 구현은 근을 제외하고 어느 곳에서도 힙 조건을 만족하는 힙을 고쳐나가는 과정 주위에 중점을 둔 것이다. 만약 근에 있는 키가 너무 적으면, 도달된 노드들

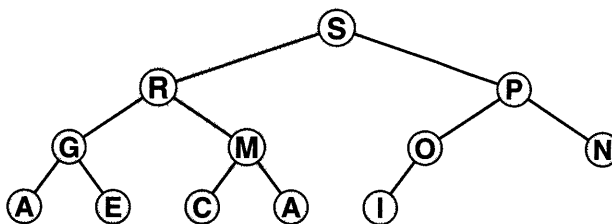


그림 11.5 힙에서 가장 큰 요소 제거

의 어떤 곳에서도 힙의 성질을 위배하지 않고도 힙을 따라 내려가게 된다. 같은 연산은 어떤 위치에서의 값이 더 낮게 된 후에 힙을 고쳐나가는데 이용된다. 그것은 다음과 같이 구현된다.

```
void PQ :: downheap(int k)
{
    int j; itemType v;
    v = a[k];
    while ( k <= N/2 )
    {
        j = k + k;
        if ( j < N && a[j] < a[j+1] ) j++;
        if ( v >= a[j] ) break;
        a[k] = a[j]; k = j;
    }
    a[k] = v;
}
```

이같은 프로시저는 힙의 밑으로 내려가면서 k 위치에 있는 노드와 필요한 경우 그것의 두 개 자식중의 더 큰 것과 교환하고, k에 있는 노드가 양쪽 자식들보다 더 크거나 혹은 제일 밑에 도달이 될 때 멈추도록 하는 것이다.(k에 있는 노드가 단지 한 개 자식을 가지도록 하는 것은 가능하다. 이 경우는 적절하게 취급이 된다!) 위에 처럼, 완전한 교환은 v가 교환에서 항상 포함이 되기 때문에 필요치않다. 이 프로그램에서 내부 반복은 진정으로 두 가지 명확한 출구를 지니는 반복의 예제이다. 즉, 하나는 힙의 제일 밑에 도달된 경우(위의 첫 번째 예제로서)이고, 다른 하나는 힙의 내부 어느곳에서든지 힙의 조건을 만족하는 경우이다. 이것은 break 명령을 이용하기 위해 부르는 상황의 전형적인 것이다.

제거 연산은 이같은 프로시저의 직접적인 응용이다.

```
itemType PQ::remove()
{
    itemType v = a[1];
    a[1] = a[N--];
    downheap(1);
    return v;
}
```

되돌려지는 값은 $a[1]$ 에 세트되고, $a[N]$ 에서의 요소는 $a[1]$ 에 놓고 그리고 힙의 크기는 한 개 감소된다. 이때 어느 곳에서든지 힙의 조건에 만족되도록 하기 위해서 `downheap`에 대한 부름만을 남겨두게 된다. 대체 연산은 다소 더 복잡한 것이다.

```
itemType PQ :: replace(itemType v)
{
    a[0] = v;
    downheap(0);
    return a[0];
}
```

이 코드는 임의적인 방법으로서 $a[0]$ 를 이용한다. 즉, 그것의 자식은 0(자기 자신)과 1이다 그래서 만약 v 가 힙에서 가장 큰 요소보다 큰 경우, 힙은 더 이상 수행이 안된다. 그렇지 않으면, v 를 힙에 삽입시키고 $a[1]$ 을 되돌려준다.

힙에서 임의의 요소에 대한 추출(`extract`) 연산과 변경 연산은 위의 방법들의 간단한 조합을 이용함으로써 구현된다. 예를 들면, k 위치에 있는 요소의 우선 순위가 제거되면, `upheap(k)`가 불려지고, 그것이 낮으면, `downheap(k)`이 처리된다.

성질 11.1 모든 기본적인 연산들 즉, 삽입, 제거, 대체, (`downheap`, `upheap`) 삭제와 변경은 N 개 요소들의 힙에서 수행할 때 $2 \lg N$ 번 비교보다 더 적게 된다.

이같은 모든 연산들은 크기 N 힙에 대해 단지 $\lg N$ 요소들만을 포함한 힙의 근과 제일 밑에 있는 것사이에 경로를 따라서 이동되는 것이 수반된다. 2의 인자는 `downheap`에서 오고, 이것은 내부 반복에서 두 번의 비교를 한다. 다른 연산은 단지 $\lg N$ 번 비교만을 요구한다. \square

결합 연산은 이같은 리스트에 포함되지 않음을 주시해야 한다. 이같은 연산을 효율적으로 수행함은 훨씬 더 복잡한 자료 구조를 요구한다. 다른 한편으로 많은 응용에서 이같은 연산은 다른 것들보다 훨씬 더 적은 빈도를 요구한다.

히프 정렬(Heapsort)

우아하고 효율적인 정렬 방법은 위에서 논의된 히프들에 대한 기본적인 연산들에서 정의가 된다. 히프 정렬이라 부르는 이같은 방법은 특별한 기억장소를 이용하지 않고 입력이 무엇이든지 간에 약 $N \log N$ 단계로서 N 개 요소들을 정렬하도록 하는 것이다. 불행히도, 내부 반복은 퀵 정렬의 내부 정렬보다 한 비트 더 긴 것이고 평균적으로 퀵 정렬보다 약 두 배로 느리다.

개념은 정렬되어질 요소들을 포함한 히프를 단순히 생성을 하고, 그것들 모두를 순서적으로 제거하는 것이다. 정렬하는 한 방법은 다음 코드의 첫 번째 두 개 라인에서와 같이 처음에 비어있는 히프로 요소들을 하나씩 삽입시키는 것이고(실제적으로는 `construct(a, N)`를

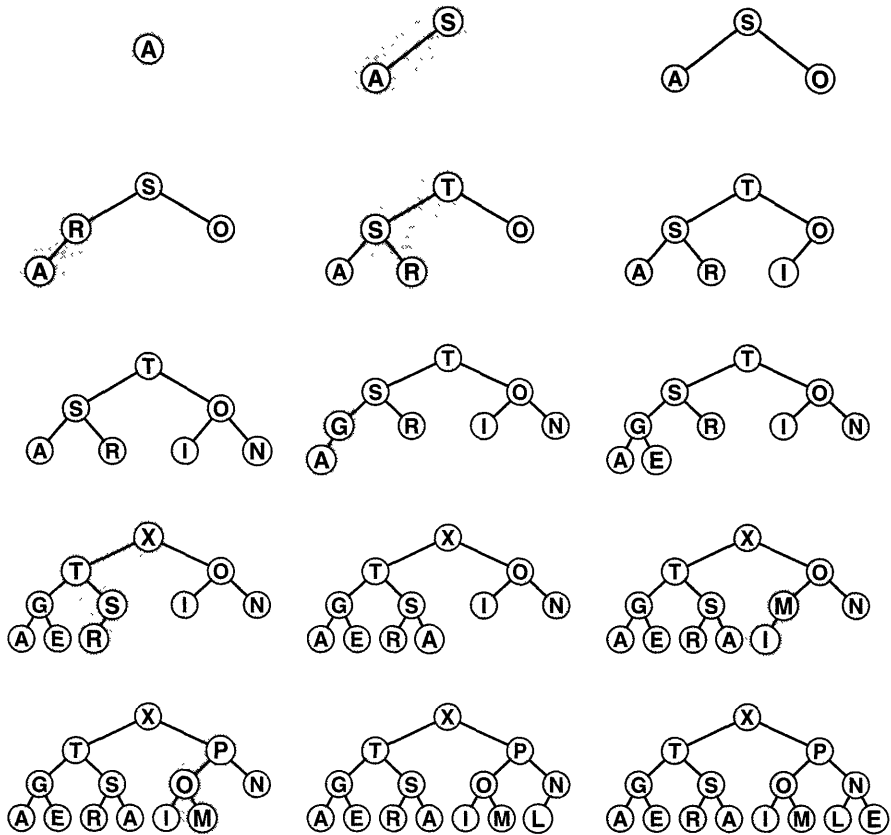


그림 11.6 하향식 히프 구조

구현), 줄어드는 힙에 의해서 비어있는 장소로 제거된 요소를 놓으므로써 N번 제거 연산을 수행한다.

```
void heapsort(itemType a[], int N)
{
    int i; PQ heap(N);
    for ( i = 1; i <= N; i++) heap.insert(a[i]);
    for ( i = N; i >= 1; i--) a[i] = heap.remove();
}
```

우선 순위 큐 프로시저는 명확한 목적에 대해서만 이용된다. 즉, 정렬의 실제적인 구현에서, 불필요한 프로시저 부름을 피하기 위해서 프로시저로부터 코드를 단순히 이용케 된다. 더

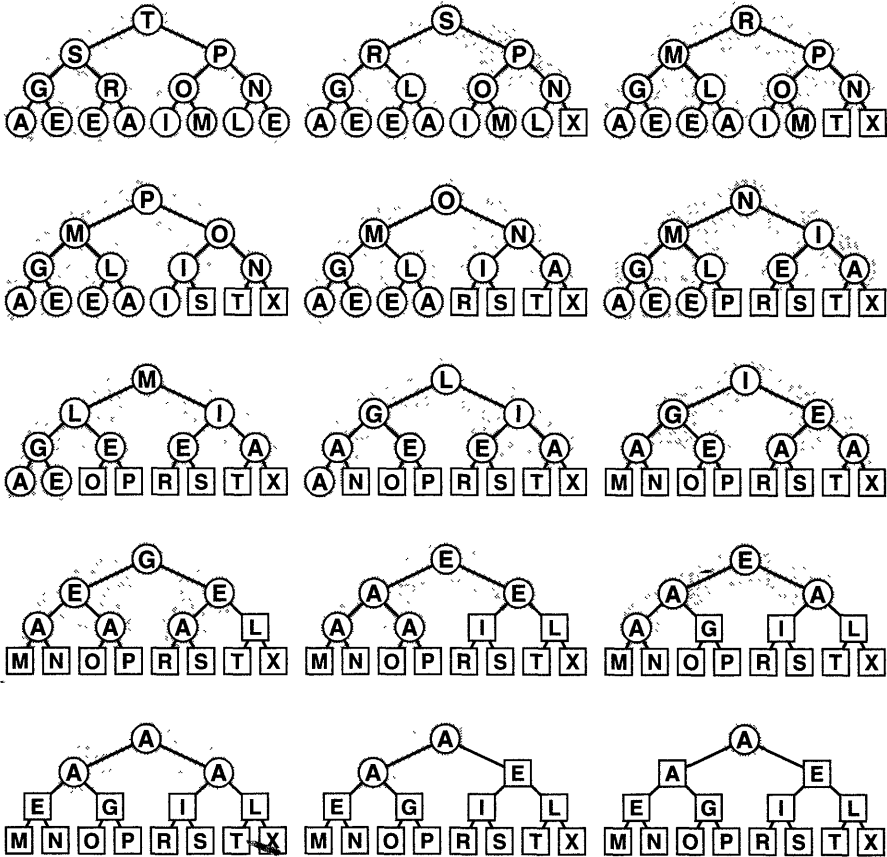


그림 11.7 힙에서 정렬

중요한 것은 계산이 배열로 되어져서 정렬이 heapsort로 하여금 배열에 직접적인 접근을하도록 하고, $a[1], \dots, a[k-1]$ 에 존재하는 우선 순위 큐를 남겨둠으로써 그 장소내에서 이루어진다.(힙에 대한 특별한 기억장소를 이용함이 없이도)

그림 11.6은 키 A S O R T I N G E X A M P L E가 처음에 비어있는 힙으로 그 같은 순서로 삽입이 되도록 힙 구성을 나타내는 것이고, 그림 11.7은 이같은 키들이 X를 제거시키고 그 후에 T를 제거시키는 등 등에 의해서 정렬되는 방법을 제시한다.

그림 11.8에 제시된 것과 같이 밑에서 위로 적은 부분 힙을 만들면서 그것을 통해 뒤로 가도록 함으로써 힙을 생성하는 것이 실제적으로 좋다. 이같은 방법은 배열에서 모든 위치를 적은 부분 힙의 근으로 간주하고, downheap은 큰 힙에 대한 것으로서 그런 부분 힙에 대해서도 똑같이 처리를 한다는 사실의 잇점을 취한다. 힙을 통해 뒤로 처리하기 위해서, 모든 노드는 근에 대해 것만을 제외하고 힙 순서화된 부분 힙의 근이 된다. 즉, downheap은 작업을 완료한다. 조사는 크기 1의 부분 힙은 그냥 넘어가도록 하기 때문에 배열을 통해서 거의 반 뒤에서 시작된다.

제거 연산은 첫 번째와 마지막 번째 요소를 교환하고 N를 감소하고, downheap(1)을 부르므로써 구현이 됨을 이미 알 수가 있다. 이것은 힙 정렬의 다음 구현으로 된다.

```
void heapsort(itemType a[], int N)
{
    int k;
    for ( k = N/2; k >= 1; k-- )
        downheap(a, N, k);
    while ( N > 1 )
        { swap( a, 1, N ); downheap( a, --N, 1 ); }
}
```

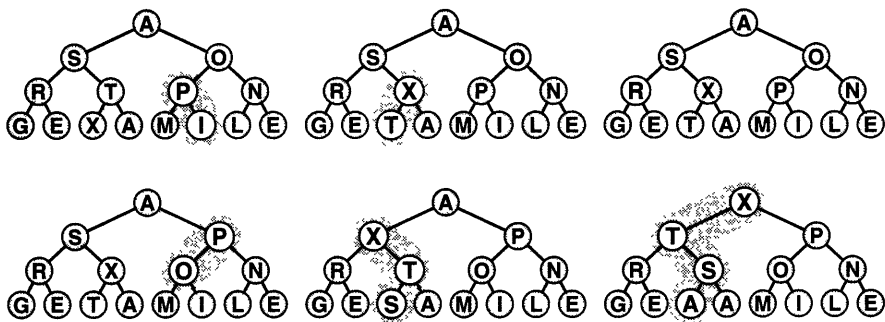


그림 11.8 상향식 힙 구성

여기서 표현을 숨기는 것의 어떤 개념은 포기하고, downheap은 첫 번째 두 개 인수로서 배열과 힙의 크기를 취하도록 변경시킨다고 가정을 하자. 첫 번째 for반복은 선형시간내에 배열을 힙 순서로 하는 구성자를 구현하는데 이용된다. 그때 while반복은 가장 큰 요소와 마지막에 있는 요소를 교환하고 그리고 이전처럼 힙을 고쳐나간다. 이 프로그램에서 반복들이 어렵게 수행이 되는 것처럼 보이지만 그것들은 같은 근본적인 프로시저로서 설정이 되는 것을 주시함은 흥미로운 것이다.

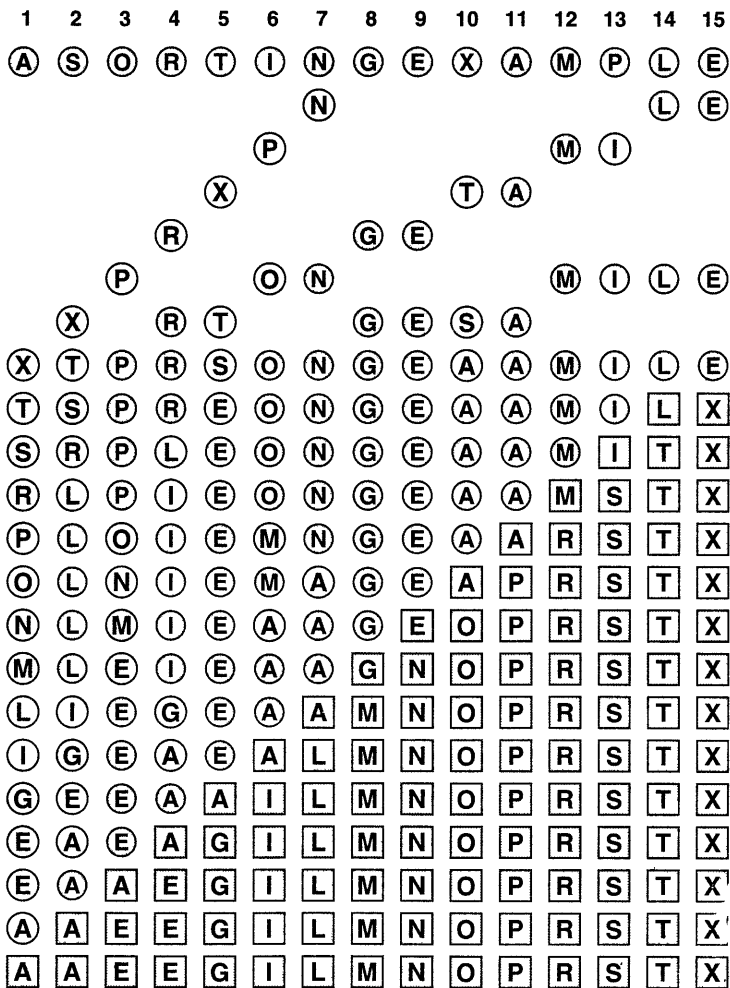


그림 11.9 힙 정렬 데이터 이동

그림 11.9는 downheap으로 힙 조건이 어느곳에서든지 만족하도록 한 후에 정렬 예제에 대해서 downheap에 의해 수행이 되는 각 힙의 내용을 제시함으로써 힙 정렬에서 데이터 이동을 나타낸 것이다.

성질 11.2 상향식 힙 구성은 선형시간이다.

이같은 성질에 대한 이유는 처리된 대부분의 힙들이 적다는 것이다. 예를 들면, 127개 요소의 힙을 생성하기 위해서, 방법은 크기 1의 64개 힙들, 크기 3의 32개 힙들, 크기 7의 16개 힙들, 크기 15의 8개 힙들, 크기 31의 4개 힙들, 크기 63의 2개 힙들 그리고 크기 127의 1개 힙에 대해서 downheap을 부르므로 해서 $64 \cdot 0 + 32 \cdot 1 + 16 \cdot 2 + 8 \cdot 3 + 4 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 120$ “개선”은(많은 비교와 같이 2배) 최악의 경우에 요구된다. $N = 2^n$ 에 대해, 비교의 수에 대한 상한(upper bound)은 다음과 같고

$$\sum_{1 \leq k \leq n} (k-1)2^{n-k} = 2^n - n - 1 < N,$$

그리고 비슷한 증명이 N 가 2의 멱승이 아닐 때 유지된다. \square

이같은 성질은 정렬에 대해 시간이 $N \log N$ 에 여전히 지배를 받기 때문에 힙 정렬에 대해서는 특별히 중요하지는 않지만 그러나 선형 시간 construct가 선형 시간 알고리즘으로 되는 다른 우선 순위 큐에 대해서는 중요하다. N 개 연속적인 삽입으로 힙을 구성하는 것은 최악의 경우 $N \log N$ 단계를 요구한다는 것을 주시해야한다.(비록 평균의 경우는 선형 시간이라 할지라도)

성질 11.3 힙 정렬은 N 개 요소를 정렬하는데 $2N \lg N$ 번 비교 보다 적게 이용된다.

다소 높은 경계 즉, 말하자면, $3N \lg N$ 는 성질 11.1에서 나타낸 것이다. 여기서 주어진 경계는 성질 11.2에서 기본으로한 더 조심스런 계수에서 따른다. \square

위에 언급한 대로, 성질 11.3은 힙 정렬이 실제적인 관심이 된다는 주된 이유이다. 즉, N 개 요소를 정렬하는데 요구되는 단계의 수는 입력에 관계없이 $N \log N$ 에 비례된다. 다른 방법과는 달리, 힙 정렬을 더 느리게 하는 “최악의 경우” 입력은 없다.

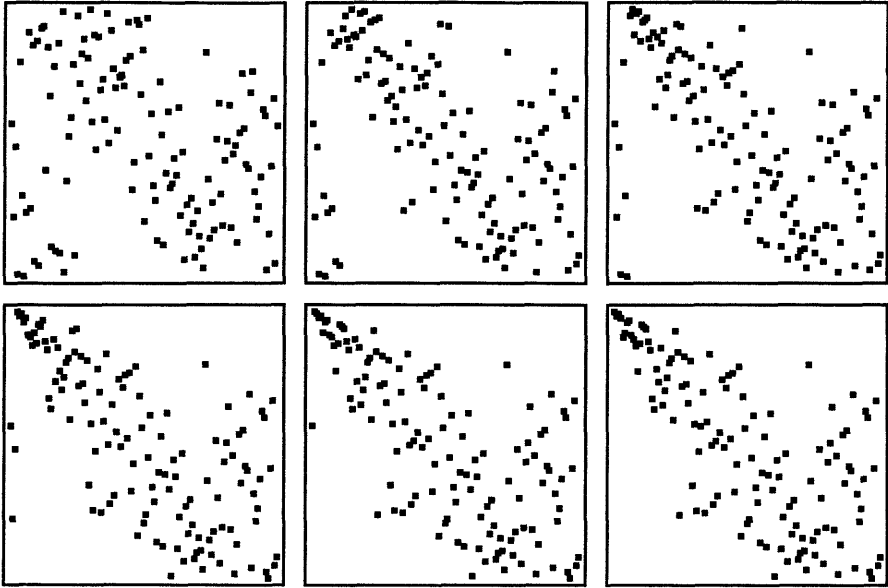


그림 11.10 무작위 순열을 힙 정렬. 구성 단계

그림 11.10과 11.11은 더 큰 무작위적으로 순서화된 파일에서 힙 정렬에 대한 처리를 나타낸 것이다. 그림 11.10에서는 큰 요소가 파일의 시작점으로 이동이 되기 때문에 정렬이외의 어느 것도 존재됨을 보게 된다. 그러나 그림 11.11은 파일로서 유지되어진 이같은 구조는 큰 요소를 집어내므로써 정렬되는 것을 나타내게 된다.

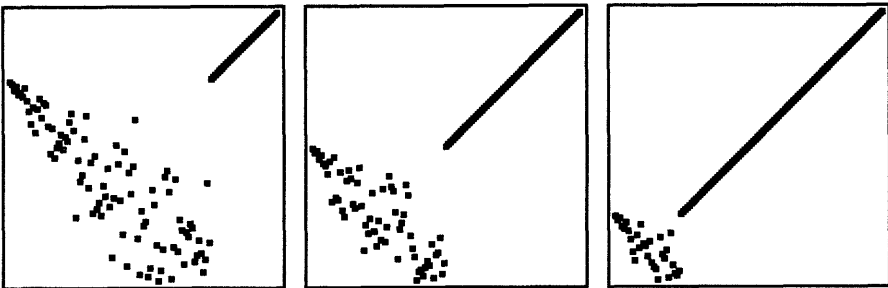


그림 11.11 무작위 순열을 힙 정렬. 정렬 단계

간접 힙(Indirect Heaps)

우선 순위 큐의 많은 응용에 대해, 레코드를 주위로 전혀 이동되기를 원치 않는다. 대신에, 값을 되돌리는 것이 아니라 레코드중의 어느것이 가장 큰 것인가 하는 등을 말해주는 우선 순위 큐를 원한다. 이것은 8장에서 기술한 “포인터 정렬” 개념 혹은 “간접 정렬”와 비슷하다. 이 방법은 이용하기가 상당히 편리하기 때문에, 여기서 더 자세하게 조사하는 것이 가치가 있다.

한 방법은 8장에서 제시된 것과 같이 어떤 것을 설정해서 배열 a 에서 키들을 배열하는 대신에 우선 순위 큐 루틴을 키들로 간접적으로 언급하도록 인덱스나 포인터들의 배열을 처리하게 한다. 더구나, 변경과 삭제 연산을 구현하기 위해서 각 요소의 힙 위치 상태를 파악하는 것이 필요하다. 이것은 8장에서 논의된 것과 비슷한 방법으로 위의 코드 변형으로서 이루어진다.

본 교재에서 나중 특히 22장과 31장에서 이용될 우선 순위 큐와 거의 똑같지만 다소 더 일반적인 방법을 채택케 된다. 삽입과 변경을 위해 다른 인수를 포함하는 인터페이스를 일반화 한다. 즉, 우선 순위 큐로 삽입될 키와 관계가 되는 것인 1과 $size$ 사이의 정수이다. 특히, 제거 연산은 키 그 자체가 아니라 큐에서 가장 적은 키와 관계된 정수를 되돌려 주는 것이다. 예를 들면, 만약 키들이 배열에서 정렬되면,(또는 그것들이 배열에서 저장된 레코드들의 일부분이면) 배열 인덱스는 이같은 목적을 위해서 이용된다. 그때 키와 어떤 다른 관련된 정보는 배열에서 검색된다.

이같은 것이 어떻게 구현되는 가를 제시하기 위해서, 이 장의 제일 앞에 주어진 “비순서화된 배열” 구현의 변경된 버전을 고려하게 된다. 관련된 정보를 파악하기 위해서 배열 $info$ 를 첨가하고 큐에서 키들의 위치를 파악하기 위해서 배열 p 를 첨가 한다. 이같은 배열들은 큐에서 어떤 키 v 와 관련된 것인 모든 정수 x 에 대해 $p[info[x]] = info[p[x]] = x$ 와 $a[p[x]] = v$ 를 만족케 한다.

```
class PQ
{
    private :
        int *a, *p, *info;
        int N;
    public :
```

```

PQ(int size)
{ a = new itemType[size];
  p = new itemType[size];
  info = new int[size]; N = 0; }
~PQ()
{ delete a; delete p; delete info; }
void insert(int x, itemType v)
{ a[++N] = v; p[x] = N; info[N] = x; }
void change(int x, itemType v)
{ a[p[x]] = v; }
int remove() // remove smallest
{
  int j, min = 1;
  for ( j = 2; j <= N; j++ )
    if ( a[j] < a[min] ) min = j;
  swap(a, min, N); swap(info, min, N);
  p[info[min]] = min;
  return info[N--];
}
int empty()
{ return ( N <= 0 ); }
};

```

이같은 구현은 더 적은 키가 더 높은 우선순위를 지닌다. 그리고 이것은 힙 정렬에 대해 이용된 것의 반대 방법으로 응용에서는 적어도 공통적인 것이다. 이같은 구현을 판단하는 열쇠는 변경 연산이다. 보통과 같이, 오류 체크를 하지 않고, (예를 들면) x가 적절한 범주내에 존재하고 그리고 사용자가 꽉찬 큐에 어떤 것을 삽입시키려고 하지 않거나 비어있는 큐에서 그것을 제거하지 않은 경우라고 가정을 하자. 그런 체크에 대한 C++ 코드를 첨가하는 것은 간단하다.

만약 우선 순위 큐가 매우 크고 그리고/혹은 상당수의 제거 연산이 수행이 되면, 배열 a는 대수 요소에 의해 모든 연산의 실행시간이 경계되도록 하기 위해서 힙에서 유지가 된다. 위에 주어진 힙 루틴은 또한 요구대로 info와 p배열을 유지하도록 하기 위해서 간단한 방법으로서 변경된다. 즉, 비교들은 a를 직접적으로 ~~연급을~~ 연급을 하나, 이동은 위의 구현에서 처럼 info에 직접적으로 그리고 p에 간접적으로 반영된다. 이것은 인터페이스를 변경하지

않고도 모든 것을 수행을 하므로서 그것들이 기술된바와 같이 히프를 이용해서 효율적으로 구현을 하다고 가정하면 우선 순위 큐를 본 교재에서 class로서 정의한다.

어떤 응용에서, 다른 구현들은 수행되어지는 우선 순위 큐 연산의 혼합과 키들의 본질에 따라서 적절하게 된다. 예를 들면, 큐가 오히려 적으면,(가령 20개 요소나 그런 경우) 위의 구현은 아주 잘 처리가 된다. 인터페이스에서 적은 변경 또한 적절하다. 예를 들면, 관련된 정보에 대한 인덱스가 아니라 큐에서 가장 큰 우선 순위 키 값을 되돌려 주는 함수를 원한다. 위의 class에 그런 프로시저를 첨가하는 것은 쉬우나 모든 연산들에 대한 대수 활용도가 인정되는 class를 개발하는 것은 더 도전적이다.

고급 구현

만약 결합 연산이 효율적으로 수행되면, 그때 지금까지 수행된 구현은 충분치 못하고 더 고급화된 기법들이 필요케 된다. 비록 그런 방법들의 상세한 내용에 대한 여분이 없으면, 설 계상에서 몇 가지를 고려해야만한다.

“효율적으로”에 의하면, 결합은 다른 연산들과 똑같이 수행된다. 이것은 두 개 큰 히프들이 큰 배열에 그것들중 적어도 하나에 있는 모든 요소들을 이동시키므로써 만이 결합되기 때문에 이용되어진 히프에 대한 연결이 없는 표현을 배제케 된다. 연결된 표현을 이용하는 알고리즘으로 변경하는 것은 쉽다. 사실, 그렇게 수행하는데는 여러 가지 이유가 있다.(예를 들면, 크고 연속적인 배열을 지니는 것이 불편하다) 직접적인 연결 표현에서, 연결은 부모와 양쪽 자식들을 가르키는 각 노드에서 유지되는 것이다.

히프 조건 그 자체는 너무 강해서 결합 연산의 구현을 효율적으로 허용하지 못하는 것이다. 고급 자료 구조가 결합에 대해 필요한 유연성을 얻기 위해서 히프나 혹은 균형 조건을 약화시키는 이같은 문제를 해결하기 위해서 설계가 된다. 이같은 구조들은 대수 시간내에 모든 연산들이 완료가 되도록 한다.

연습 문제

1. 초기에 비어있는 힙에서 다음 연산들이 수행될 때 결과가 되는 힙을 그려라.
삽입(1), 삽입(5), 삽입(2), 삽입(6), 대체(4), 삽입(8), 제거, 삽입(7), 삽입(3)
2. 역으로 정렬된 순서 파일은 힙인가?
3. 비어있는 힙에서 시작해서 삽입은 키 E A S Y Q U E S T I O N에 대해 연속적으로 부를 때 결과적으로 나타나는 힙은 무엇인가?
4. 크기 32의 힙에서 3번째 가장 큰 키가 차지하는 위치는 어느 것인가? 크기 32의 힙에서 3번째로 적은 키가 차지하지 못하는 키는 어느 것인가?
5. downheap에서 $j < N$ 테스트를 피하기 위해서 표지를 왜 이용하지 않는가?
6. 우선 순위 큐의 특별한 경우로서 정상적인 큐와 스택의 기능들을 어떻게 얻는지를 설명하라.
7. 힙에서 “가장 큰 것을 제거”하는 연산동안에 이동이 되어지는 키들의 최소의 수는 무엇인가? 최소의 것이 성취되는 것에 대한 크기 15의 힙을 그려라.
8. 힙에서 위치 d에 있는 요소를 삭제하는 프로그램을 기술하라.
9. 10000개의 무작위 키들로서 힙을 생성하므로써 상향식 힙 구조와 하향식 힙 구조를 실험적으로 비교하라.
10. 키 E A S Y Q U E S T I O N(i 에 대응되는 순서에서 i 번째 문자로서)는 처음에 비어있는 힙으로 삽입시킨 후에 p와 info 배열의 내용은 어떻게 되는가?

12 장

병합 정렬

9장의 파일에서 k 번째로 적은 요소를 찾기 위해서 선택 연산에 대해 공부하였다. 선택은 파일을 두 개 부분으로 구분 즉, k 개의 적은 요소들과 $N - k$ 개의 큰 요소들로 구성된 것이다. 본 장에서는 하나의 더 큰 정렬된 파일을 얻기 위해서 두 개 정렬된 파일을 결합시키는 연산에 대해 보충적인 과정인 병합을 보게 된다. 병합은 간단한 재귀 정렬 알고리즘에 대한 기본이다.

선택과 병합에서 선택은 파일을 두 개의 독립된 파일로 분리시키고, 병합은 두 개 독립된 파일을 하나의 파일로 결합시키는 의미에서 보충적인 연산이다. 또한 이같은 연산들의 관계는 정렬 방법을 생성하도록 “분할-정복”에 적용을 시키도록한다. 파일은 두 개 부분이 정렬될 때 전체 파일이 정렬되도록 재배열을 하거나 혹은 정렬되도록 두 개 부분으로 나누고 그리고 정렬된 전체 파일을 만들도록 결합시키는 것이다. 첫 번째 예에서 무엇이 발생하는가는 두 개 재귀 부름에 의해서 선택 프로시저로 구성되는 퀵 정렬에서 병합 과정에 따르는 기본적인 두 개 재귀 부름이 구성된다는 점에서 병합 정렬은 퀵 정렬의 보충으로 된다.

히프 정렬과 같이 병합 정렬은 최악의 경우 조차 $N \log N$ 시간내에 N 개 요소들의 파일을 정렬하는 잇점을 지닌다. 병합 정렬의 주된 단점은 이같은 단점을 극복하려는 노력없이도 N 에 비례하는 특별한 공간이 요구된다. 내부 반복의 길이는 퀵 정렬과 병합 정렬의 것 사이에 어디엔가 존재하고 그래서 병합 정렬은 속도가 본질적인 문제로 되는 경우, 특히 공간이 이용가능한 경우 후보자가 된다. 더구나, 병합 정렬은 주로 데이터를 순차적인 방법으로 접근을 하게 하도록(하나씩) 구현되고, 이것은 때때로 명확한 장점이 된다. 예를 들면, 병합 정렬은 연결 리스트에 대한 선택 방법이고, 그곳은 순차 접근이 사용가능한 일종의 접근이다. 비슷하

게, 13장에서 볼수 있듯이, 병합은 비록 그 내용에서 이용된 방법이 병합 정렬에서 이용된 것과 다소 다르게 되지만 순차 접근 장치에서 정렬에 대한 기본이다.

병합

많은 자료 처리 환경에서, 큰(정렬된) 데이터 파일은 새로운 엔트리가 정기적으로 첨가되는 곳에서 유지된다. 전형적으로 새로운 엔트리의 수는(훨씬 더 큰) 주된 파일에 첨가되어서 “일괄 처리”로 되고, 전체 것은 다시 저장된다. 이같은 상황은 병합에 대해서 맞추어 놓은 것이다. 즉, 더 좋은 방법은 새로운 엔트리의(적은) 일괄처리를 정렬하는 것이고, 그것을 큰 주된 파일에 병합하는 것이다. 병합은 그것의 연구를 가치있게 하는 다른 많은 비슷한 응용을 지닌다. 또한 병합을 기준으로 한 정렬 방법을 조사케 된다.

이 장에서는 두 가지 방법(two-way) 병합에 대한 프로그램에 집중을 할 것이다. 즉, 두 개 정렬된 입력 파일을 하나의 정렬된 출력 파일로 만들도록 결합하는 프로그램이다. 다음 장에서는 두 개 이상의 파일이 수반되는 다중 방법(multiway) 병합에 대해서 상세하게 보게 될 것이다.(다중 방법 병합의 가장 중요한 응용은 장의 주제인 외부 정렬(external sorting)이다)

우선적으로, 세 번째 배열 $c[1], \dots, c[M+N]$ 로 병합이 되도록 하는 두 개의 정렬된 정수형 배열 $a[1], \dots, a[N]$ 와 $b[1], \dots, b[N]$ 를 지닌다고 가정하자. 다음은 a 와 b 에서 남아있는 가장 적은 요소를 c 에 계속적으로 선택하도록 하는 명백한 방법의 직접적인 구현이다.

```
i = 1; j = 1;
a[M+1] = itemMAX; b[N+1] = itemMAX;
for( k = 1; k <= M+N; k++ )
    c[k] = ( a[i] < b[j] ) ? a[i++] : b[j++];
```

구현은 다른 모든 키들보다 더 큰 값으로 된 표지 키에 대해 a 와 b 배열에 여지를 주므로써 간단히 할 수가 있다. $a(b)$ 배열이 소모될 때, 반복은 단순히 $b(a)$ 배열의 나머지를 c 배열로 이동시키는 것으로 명확히 $M + N$ 번 비교를 이용한다. 만약 $a[M+1]$ 과 $b[N+1]$ 은 표지 키로서 유용하지 않는 경우, 그때 i 가 항상 M 보다 적고, j 는 N 보다 적다는 테스트가 추가된다. 이같은 어려움 주위에 또 다른 방법은 병합 정렬의 구현에 대해 아래에서 사용된다.

병합 파일의 크기에 비례하는 특별한 공간을 이용하는 대신에, 한 입력에 대해 $c[1], \dots, c[M]$ 를, 다른 입력에 대해 $c[M+1], \dots, c[M+N]$ 를 이용하는 장소내에 방법을 지니도록 하는 것이 바람직하다. 첫째로, 이것은 쉽게 행해 질 수 없다는 것과 그런 방법이 존재한다는 것을 확인 시키는 것은 어렵다. 그러나 장소내의 정렬은 상당히 많은 주의가 연습되지 않으면, 더 효율적으로 되기는 매우 복잡하다. 이같은 문제는 아래에서 보도록 하자.

특별한 공간이 실제적인 구현에 대해 요구되는 것으로서, 연결 리스트 구현을 고려하게 된다. 사실, 이 방법은 연결 리스트에 매우 적합한 것이다. 우리가 이용하는 모든 편리성을 제시하는 구현은 아래에 주어진다. 즉, 실제 병합에 대한 코드가 위의 코드에서와 같이 간단하다.

```
struct node
{ itemType key; struct node *next; }
struct node *z;
struct node *merge(struct node *a, struct node *b)
{
    struct node *c;
    d = z;
    do
        if ( a->key <= b->key )
            { c->next = a; c = a; a = a->next; }
        else
            { c->next = b; c = b; b = b->next; }
    while ( c != z );
    c = z->next; z->next = z;
    return c;
}
```

이 프로그램은 a에 의해서 지적되는 리스트와 보조 포인터 c의 도움으로서 b에 의해서 지적되는 리스트와 병합된다.

이 장에서는 병합 정렬 알고리즘을 표현하는 경제성에 대해 3장에서 class List를 이용하기 보다는 오히려 직접적으로 리스트상에 연결들을 처리한다. 리스트들은 3장에서 처럼 가상의 “꼬리” 노드를 지나다고 하자. 즉, 모든 리스트들은 가상 노드 z로서 끝이 나고, 이것은 정상적으로 자기 자신을 가르키고 그리고 또한 $z->key == \text{itemMAX}$ 로서 표지의 역할을 한다. 병합동안에, z는 새로이 병합된 리스트의 시작에 유지되도록 이용되고,(즉, next 필드가

리스트의 시작을 가르키는 가상의 헤더 노드로서 이용된다) c는 새로이 병합된 리스트의 끝을 나타낸다.(연결 필드는 새로운 요소가 리스트에 추가되도록 변경이 되어지는 노드) 병합 리스트가 생성된 후에, 첫 번째 노드에 대한 포인터는 $z \rightarrow next$ 로부터 검색이 되고 z는 그 자체를 지적하도록 다시 세트된다.

병합에서 키 비교는 동등성을 포함하고 그래서 병합은 b리스트가 a리스트를 따르도록 고려되는 경우 안정적이다. 병합에서 이같은 안정성은 병합을 이용하는 정렬 프로그램에서 안정성을 이루도록 하는데 이용되는 방법을 아래에서 보게 된다.

병합 정렬

한 번 병합 프로시저를 지니면, 재귀 정렬 프로시저에 대해 기본적으로 이용하는 것은 어렵지 않다. 주어진 파일을 정렬하기 위해서, 그것을 반으로 쪼개고, 두 개의 반쪽으로 된 것을 정렬하고(재귀적으로) 그리고 그때 두 개의 반쪽으로된 것을 함께 병합한다. 이 과정의 구현은 배열 $a[1], \dots, a[r]$ (보조 배열 $b[1], \dots, b[r]$ 을 이용해서)을 정렬하는 것이다.

```
void mergesort(itemType a[], int l, int r)
{
    int i, j, k, m;
    if ( r > l )
    {
        m = ( r + l ) / 2;
        mergesort(a, l, m);
        mergesort(a, m+1, r);
        for ( i = m+1; i > l; i-- ) b[i-1] = a[i-1];
        for ( j = m; j < r; j++ ) b[r+m-j] = a[j+1];
        for ( k = 1; k <= r; k++ )
            a[k] = ( b[i] < b[j] ) ? b[i++] : b[j--];
    }
}
```

이 프로그램은 두 번째 배열을 첫 번째 배열뒤의 위치로 복사하므로써 그러나 열 순으로 표지를 이용함이 없이 병합이 수행되도록 하는 것이다. 이와 같이 각 배열은 다른 것에 대해 "표지"로서 이용된다. 즉, 가장 큰 요소(한 배열이나 다른 것내에 것)는 다른 배열이 병합에

대해 소모된 후에 적절히 이동을 시켜서 유지를 한다. 이 프로그램의 “내부 반복”은 오히려 짧고,(b로 이동하고, b로 다시 이동하고 i나 j를 증가시키고, k를 증가시키고 테스트한다) 비록 이것들이 거꾸로 표지로 되돌아 가지만 코드의 두 개 복사(a에서 b로 병합하는 하나와 b를 a로 병합시키는 다른 하나)를 지니므로써 훨씬 더 짧게한다.

예제 키들의 파일은 그림 12.1에서와 같이 처리된다. 각 라인은 병합에 대한 부름의 결과를 나타낸다. 첫째, A 와 S를 병합해서 A S를 얻고, O와 R을 병합해서 O R를 얻고 그리고 이것을 A S와 병합을 해서 A O R S를 얻게 된다. 나중에 I T와 G N를 병합해서 G I N T를 얻고, 이것을 A O R S와 병합해서 A G I N O R S T등으로 된다. 이와 같은 방법은 정렬된 적은 파일을 더 큰 파일로 재귀적으로 생성하는 것이다.

리스트 병합 정렬

이 과정은 연결 리스트 표현이 고려되는 충분한 데이터 이동을 수반한다. 다음 프로그램은 입력으로서 비 정렬된 리스트에 대한 포인터를 취하고 그 값을 리스트의 정렬된 버전에 대한 포인터로 되돌려주는 함수의 직접적인 재귀적 구현이다. 프로그램은 리스트의 노드들을 재배열함으로써 수행한다. 즉, 임시적인 노드들이나 리스트들은 할당될 필요가 없다.(그것은 재귀 프로그램에 대한 파라미터로서 리스트 길이를 전달하는 것이 편리하다. 즉, 대안으로 리스트로서 저장되거나 혹은 프로그램이 그것의 길이를 찾기 위해서 리스트를 조사할 수가 있다)

```
struct node *mergesort(struct node *c)
{
    struct node *a, *b;
    if ( c->next != z )
    {
        a = c; b = c->next->next->next;
        while ( b != z )
            { c = c->next; b = b->next->next; }
        b = c->next; c->next = z;
        return merge(mergesort(a), mergesort(b));
    }
    return c;
}
```

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S													
		O	R											
A	O	R	S											
				I	T									
						G	N							
				G	I	N	T							
A	G	I	N	O	R	S	T							
								E	X					
										A	M			
								A	E	M	X			
												L	P	
												E	L	P
								A	E	E	L	M	P	X
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

그림 12.1 재귀 병합 정렬

이 프로그램은 c에 의해서 지시된 리스트를 a와 b로서 지적된 두 개의 반쪽으로 분리시키고 그리고 마지막 결과를 얻기 위해서 병합을 이용해서 정렬된다. 다시, 이같은 프로그램은 모든 리스트들이 z로서 끝이나는 규약으로 된다. 즉, 입력 리스트는 z로서 끝나고(그리고 그러므로해서 b 리스트를 수행한다) 그리고 명확한 명령 $c \rightarrow next = z$ 는 a리스트의 끝에 z를 놓으면 된다. 이 프로그램은 비록 그것이 오히려 복잡한 알고리즘이라 할지라도 재귀 공식으로 이해하는 것이 아주 쉽다.

상향식 병합 정렬

5장에서 논의대로, 모든 재귀 프로그램은 비록 동등하지만 다른 순서로서 계산을 수행하는 비 재귀 아나로그(analog)를 지닌다. 병합 정렬은 많은 그런 계산을 특성화하는 “결합과 정복”의 모형이다. 그리고 그것의 비 재귀적 구현을 상세히 연구할 가치가 있다.

병합 정렬의 가장 간단한 비 재귀 버전은 다른 순서로서 다른 파일의 집합을 처리한다. 즉, 첫째 크기 2의 정렬된 부분 파일을 얻기 위해서 1개씩 병합을 수행하면서 리스트를 통해 조사를 하고, 크기 4의 정렬된 부분 파일을 얻기 위해서 2개씩 병합을 수행하면서 리스트를 통해 조사를 하고 그리고 크기 8의 정렬된 부분 파일을 얻기 위해서 4개씩 병합을 수행하는 등으로 전체 리스트가 정렬될 때까지 이루어진다.

그림 12.2는 이같은 방법이 예제 파일에 대해 그러나 다른 순서로서 그림 12.1에 제시된 것과 같은 병합으로 본질적으로 어떻게 수행이 되는 가를 나타내고 있다.(그것의 크기가 2의 멍승에 가까우므로 해서) 일반적으로 $\log N$ 는 각 과정에서 정렬된 부분 파일들의 크기를 두 배로 증가시키므로해서 N 개 요소들의 파일을 정렬하기 위해 요구된다.

이같은 “상향식” 방법에 의해 만들어진 실제적인 병합은 위의 재귀 구현에 의해서 수행된 병합과 같은 것이 아니라는 것을 주시하는 것이 중요하다. 그림 12.3에 제시된 95개 요소들의 정렬을 고려해보자. 마지막 병합은 재귀 정렬에서 47개와 48개를 병합하는데 반해서 64개와 31개를 병합시키는 것이다. 그러나 이것을 재배열하는 것이 가능하므로해서, 비록 그렇게 해야할 특별한 이유가 없지만 두 개 방법으로 해서 만들어진 병합의 순서는 같다.

이같은 상향식 방법을 연결 리스트를 이용한 상세한 구현은 아래와 같다.

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S													
		O	R											
				I	T									
						G	N							
								E	X					
										A	M			
												L	P	
														E
A	O	R	S											
				G	I	N	T							
								A	E	M	X			
												E	L	P
A	G	I	N	O	R	S	T							
								A	E	E	L	M	P	X
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

그림 12.2 재귀 병합 정렬

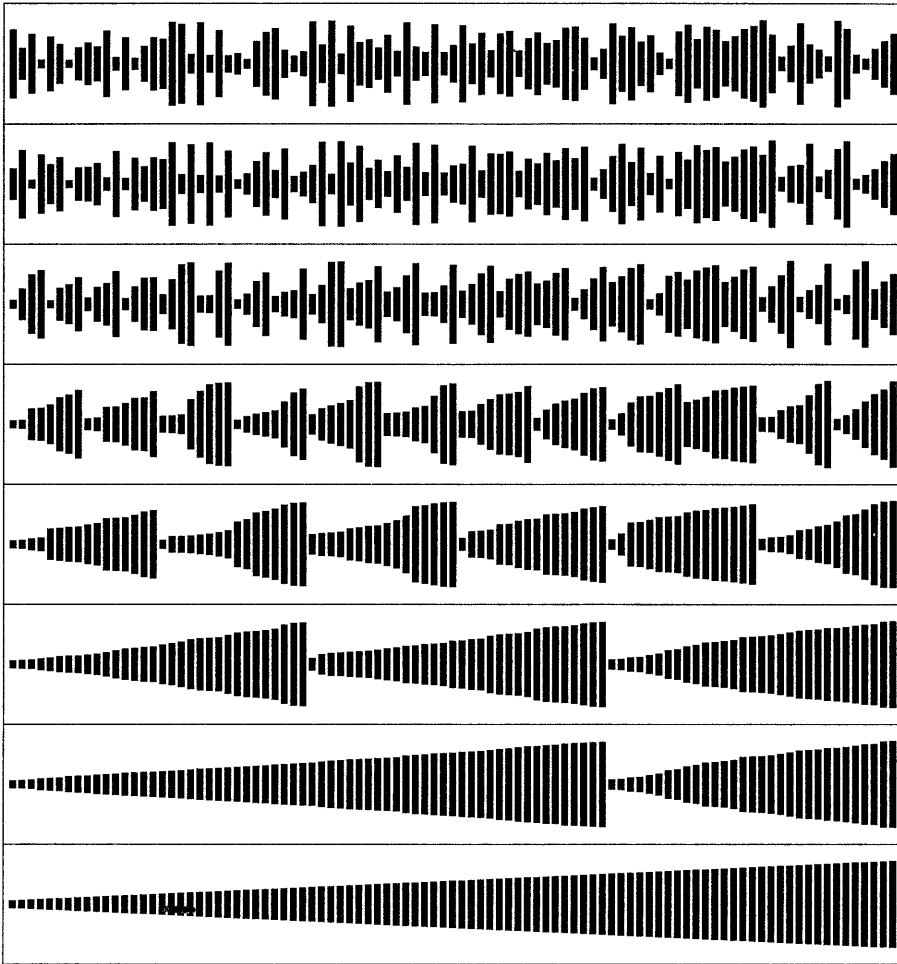


그림 12.3 무작위 순열을 병합 정렬

```

struct node *mergesort(struct node *c)
{
    int i, N;
    struct node *a, *b, *head *todo, *t;
    head = new node;
    head->next = c; a = z;
    for ( N = 1; a != head->next; N = N+N )
    {
        todo = head->next; c = head;
        while ( todo != z )

```

```

    {
        t = todo; a = t;
        for ( i = 1; i < N; i++ ) t = t->next;
        b = t->next; t->next = z; t = b;
        for ( i = 1; i < N; i++ ) t = t->next;
        todo = t->next; t->next = z;
        c->next = merge(a, b);
        for ( i = 1; i < N+N; i++ ) c = c->next;
    }
}
return head->next;
}

```

이 프로그램은 어느 연결 필드가 정렬되어진 연결 리스트를 지시하는가에 대해 “리스트-헤더” 노드(head에 의해 지시된 것)를 이용한다. 외부(for) 반복문의 각 반복은 이전의 과정에서와 같은 정도의 길이가 2배로 정렬된 부분 파일들로 이루어진 연결 리스트를 나타내면서 파일을 통해서 전달된다. 이것은 두 개 포인터 즉, 하나는 지금껏 보지 못한 리스트의 일부분이고(todo), 다른 하나는 부분 파일이 이미 병합되어진(c) 것에 대한 리스트 일부분의 끝을 나타내는 것으로 수행된다. 내부 반복(while)은 todo에 의해 지시된 노드에서 시작해서 결과 리스트인 c로 연결이 되는 길이 $N+N$ 의 부분 파일을 나타내면서 길이 N 의 두 개 부분 파일을 병합하는 것이다.

실제적인 병합은 a에서 병합되어질 첫 번째 부분 파일에 연결을 보관시키고, 그때 N 개 노드들을 그냥 넘기고, (임시적인 연결 t를 이용해서) z를 a 리스트의 끝에 연결을 시키고, 그때 b에 의해서 지적된 N 개 노드들의 다른 리스트를 얻기 위해서 같은 것을 수행하고, (방문된 마지막 노드의 연결로서 todo를 갱신) 그때 merge를 부르므로써 수행된다. (그때 c는 방금 병합된 리스트의 끝을 단순히 쫓아 내려가면서 갱신된다. 이것은 merge로 하여금 각 리스트 노드에서 시작과 끝 양쪽에 포인터를 되돌려 주게 하는 것 또는 여러개 포인터를 유지시키는 것과 같이 이용가능한 여러 가지 대안보다는 다소 더 간단한(그러나 다소 효율성이 적은 것) 방법이다.

상향식 병합 정렬은 배열 구현에 대해 이용하는 흥미로운 방법이다. 즉, 이것은 연습문제로 남겨둔다.

활용도 특징

병합 정렬은 안정적인 방법으로 구현 될 수 있는 오히려 간단하고 “최적인” 정렬 방법이 기 때문에 중요하다. 이같은 사실들은 증명하기가 상대적으로 쉽다.

성질 12.1 병합 정렬은 N 개 요소들의 어떤 파일을 정렬하는데 약 $N \lg N$ 번 비교를 요구한다.

위의 구현에서, 각 M 과 N 병합은 $M + N$ 번의 비교를 한다.(이것은 표지가 어떻게 이용 되는가에 따라서 하나나 두 개로서 다양하게 된다) 지금 상향식 병합 정렬에 대해, $\lg N$ 과 정들은 각각이 N 번의 비교를 요구하면서 이용된다. 재귀 버전에 대해, 비교의 수는 표준 “분할-정복” 재발생인 $M_1 = 0$ 인 $M_N = 2 M_{N/2} + N$ 에 의해서 기술이 된다. 6장에서 이것은 $M_N \approx N \lg N$ 임을 알았다. 이같은 인수들은 N 가 2의 멍승인 경우에 진이 된다. 일반적인 N 에 대해서 그것들이 똑같이 유지되는 가를 나타내는 것은 연습문제로 남겨둔다. 더구나, 평균의 경우에서 또한 유지된다. \square

성질 12.2 병합 정렬은 N 에 비례하는 특별한 공간을 이용한다.

이것은 비록 단계들이 문제의 영향을 감소시키지만 구현에서 명확하게 된다. 물론 만약 정렬되어진 “파일”이 연결 리스트를 이용하면, “특별한 공간”(연결에 대해)이 다른 목적으로 존재되기 때문에 문제가 제기 되지 않는다.

배열에 대해, 먼저 두 개 배열(연습문제 2번 참조)중의 더 적은 것에 대한 특별한 공간을 이용해서 M 와 N 으로된 병합을 수행하는 것은 간단함을 주시해야한다. 이것은 병합 정렬에 대한 공간 요구를 반으로 줄이는 것이다. 비록 이것이 실제로서 가치가 있지는 않지만 그 장소에 병합을 하고 더 잘 처리하는 것은 가능하다. \square

성질 12.3 병합 정렬은 안정적인다.

모든 구현에서 실제적으로는 병합동안에서만 키들이 이동되므로 해서, 병합 그자체가 안정적인가를 단순히 검증하는 것이 필요하다. 그러나 이것은 보기에는 단순하다. 즉, 똑같은 키들의 상대적인 위치는 병합 과정에서 분배되지 않는다. \square

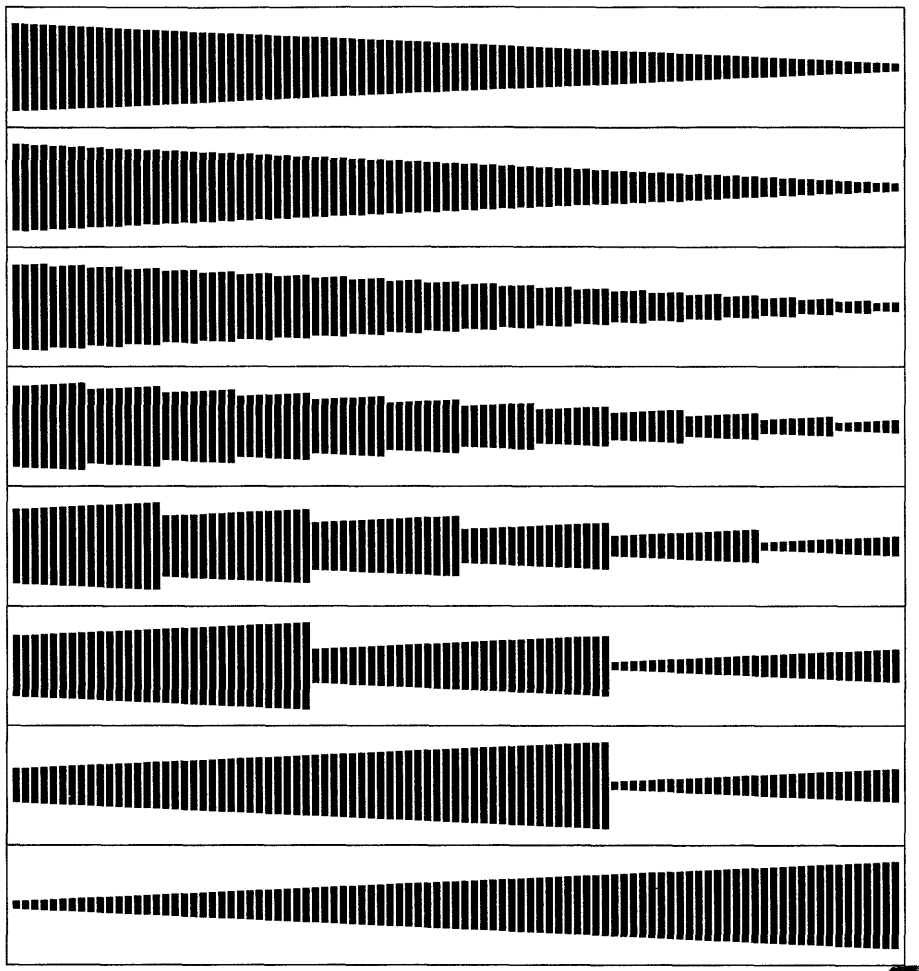


그림 12.4 역 순서로 된 순열을 병합 정렬

성질 12.4 병합 정렬은 입력의 초기 순서에 영향을 덜 받는다.

구현에서, 입력은 어느 요소들이 병합동안에 처리가 되는가 하는 순서로 처리가 되고 그래서 이같은 문장은 문자적으로 사실이다.(if 문이 어떻게 컴파일되고 수행이 되는지에 따라서 여러 가지 변화에 대한 것은 제외하고) 소비된 첫 번째 파일에 대한 명확한 테스트를 수반으로 한 병합의 다른 구현은 그렇게 많이는 아니지만 입력에 따라서 몇 가지 큰 변화로 된다. 요구되는 과정들의 수는 그것의 내용이 아니라 파일의 크기에만 명확히 의존하고 그리고 각 과정은 약 N 번의 비교를 확실히 요구한다.(실제적으로 아래에서 설명하는 것과 같이 평균적으로 $N - O(1)$ 이 된다) 그러나 최악의 경우는 평균의 경우에서와 같다. □

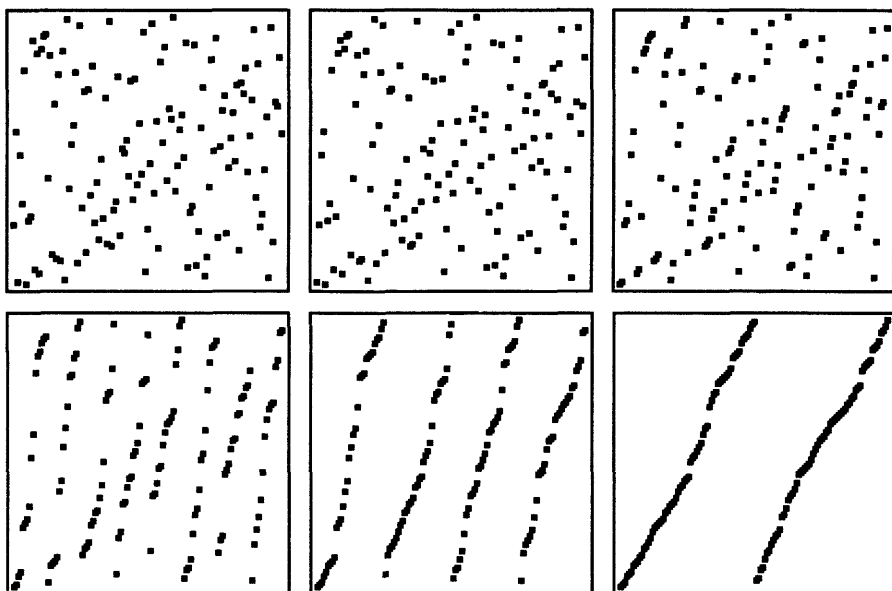


그림 12.5 무작위 순열의 병합 정렬

그림 12.4는 초기에 역순으로 되어진 파일상에서 처리되는 상향식 병합 정렬을 제시한 것이다. 같은 연산을 수행하는 셸 정렬인 그림 8.9의 그림과 비교하는 것은 흥미로운 것이다.

그림 12.5는 앞의 장들에서 비슷한 견해로서 비교를 함으로써 무작위 순열을 처리함에 있어서 병합 정렬의 또 다른 견해를 제시한 것이다. 특히, 그림 12.5는 그림 10.5와 놀라정도로 똑같다. 즉, 어떤 의미에서는 병합 정렬이 일직선상의 기수 정렬의 “변형”이다.

최적화된 구현

표지의 논의에서 배열을 기본으로 한 병합 정렬의 내부 반복에 대한 주의를 이미 보았다. 그리고 내부 반복에서 배열 경계 테스트가 배열중 하나의 순서를 거꾸로 함으로써 피하게 된다. 이것은 위의 구현에서 주된 비효율성에 대해 주의를 해야한다. 즉, a에서 b로 이동이다. 10장에서 일직선상의 기수 정렬에 대해서 본 것과 같이, 이같은 이동은 코드의 두 개 복사를 지니므로써 피하게 된다. 즉, 하나는 a에서 b로 병합하는 곳이고 다른 하나는 b에서 a로 병합하는 곳이다.

이같은 두 가지 개선점을 조합적으로 성취하기 위해서, 어떤 것을 변경하는 것이 필요하게 되고 그 결과 merge는 증가되는 순서나 감소되는 순서 어느 하나로서 배열을 출력시킨다.

비 재귀 버전에서, 증가되는 순서와 감소되는 순서 출력사이를 번갈아 가면서 성취된다. 반면, 재귀 버전에서 4가지 루틴을 지닌다. 즉, 감소되거나 증가되는 순서의 결과로서 $a(b)$ 에서 $b(a)$ 로 병합시키기 위한 것이다. 이것들의 어느 것은 병합 정렬의 내부 반복을 비교, 저장, 두 개 포인터 증가, i 또는 j 그리고 k 그리고 포인터 테스트로 줄여준다. 이것은 퀵 정렬의 비교, 증가와 테스트 그리고(부분적으로) 교환과 상당히 경쟁적이고 그리고 퀵 정렬의 내부 반복은 병합 정렬의 약 38%인 $2 \ln N \approx 1.38 \lg N$ 번 실행된다.

재 방문된 재귀

퀵 정렬에서와 같이 이 장의 프로그램들은 분할-정복 알고리즘의 전형적인 구현이다. 다음 장들에서 비슷한 구조를 지닌 여러 가지 알고리즘을 보게 될 것이다. 그래서 이같은 구현들의 기본적인 특징들 몇 가지를 더 상세하게 보는 것은 가치가 있다.

퀵 정렬은 실제적으로는 “분할-정복” 알고리즘이다. 즉, 재귀 구현에서 대부분의 작업은 재귀 부름 전에 이루어진다. 다른 한편, 재귀 병합 정렬은 분할-정복의 정신을 더 지닌 것이다. 즉, 첫째, 파일을 두 개 부분으로 나누고, 각 부분을 개별적으로 처리를 한다. 병합 정렬이 실제로 처리를 하는 것에 대한 첫 번째 문제는 적은 것이고 마지막에 가장 큰 부분 파일을 처리하는 것이다. 퀵 정렬은 가장 큰 부분 파일상에 실제적인 처리가 시작되고 그리고 적은 것으로서 마치게 된다.

이같은 차이는 두 가지 방법의 비 재귀적인 구현에서 상세하게 된다. 퀵 정렬은 데이터의 존 방법으로서 나누는 큰 부분 문제들을 보관시키기 때문에 스택을 유지시킨다. 병합 정렬은 파일을 나누는 방법이 데이터와는 별개이므로해서 부분 문제들을 처리하는 순서는 더 간단한 프로그램을 주기 위해서 재배열로서 단순한 비재귀 버전이다.

그 자체를 상세하게 하는 또 다른 실제적인 차이는 병합 정렬이 안정적이나, (만약 적절하게 구현된 경우) 퀵 정렬은 그렇지 못하다. (많은 특별한 난제를 처리함이 없으므로) 병합 정렬에 대해, 부분 파일들이 안정적으로 정렬이 되어진 경우, 그때 쉽게 배열이 되는 안정적인 방법으로서 병합이 수행되는 것만이 확실할 필요가 있다. 그러나 퀵 정렬에 대해, 안정적인 방법으로 분할을 수행하는데 쉬운 방법은 그자체를 제시하는 것이 아니므로 해서, 안정성의 가능성은 비록 재귀가 처리되기 전이라도 이미 처리가 된 것이다.

마지막으로 주의 할 점은 다음과 같다. 퀵 정렬이나 어떤 다른 재귀 프로그램과 같이, 병합 정렬은 다른 방법으로 적은 부분 파일들을 취급하므로써 개선된다. 프로그램의 재귀 버전

에서, 이것은 삽입 정렬로서 적은 부분 파일들을 처리하거나 혹은 그 후에 개끗이 된 과정을 수행하므로써 퀵 정렬에 대한 것처럼 정확히 구현된다. 비 재귀 버전에서 적은 정렬로된 부분 파일들은 삽입이나 선택 정렬의 적당한 변형을 이용해서 초기 과정에서 생성된다. 병합 정렬에 대해 제시된 또 다른 개념은 파일에서 첫 번째로 두 개 정렬된 처리들을 병합하는, (그러나 그것들이 얼마나 오래 일어나는 가) 그리고 그때 다음 두 개의 처리등으로 하는 상향식 방법을 이용해서 파일에서 “자연스런” 순서를 이용한다. 보는 바와 같이 매력적인 것은 어떤 악화되어지는 경우들에 대한 것은(이미 정렬되어지는 파일과 같은 것) 제외하고 성취된 축약을 상쇄시키는 것 이상인 내부 반복에서 떨어지는 처리를 인식하는 비용 때문에 논의된 표준 방법을 지탱할 수가 없다.

연습 문제

1. M 개 요소 이하로 된 부분 파일들에 대해 삽입 정렬에 대한 축약적으로된 재귀 병합 정렬을 구현하라. 즉, 1000개 요소들의 무작위 파일상에서 가장 빠르게 처리하는 M 의 값을 실험에 의해서 결정하라.
2. 연결 리스트와 $N = 1000$ 에 대한 재귀적인 것과 비 재귀적인 병합 정렬을 실험적으로 비교하라.
3. $N/2$ 이하인 크기의 보조 배열을 이용해서 N 정수형의 배열에 대해 재귀적 병합 정렬을 구현하라.
4. 다음 사실이 맞는가 틀리는 가: 병합 정렬의 실행시간은 입력 파일에서 키들의 값에 의존하지 않는다. 답을 설명하라.
5. 병합 정렬이 이용하는 가장 적은 순의 단계는 무엇인가?(상수 요소내로)
6. 연결 리스트 대신에 두 개 배열을 이용한 상향식 비 재귀적인 병합 정렬을 구현하라.
7. 재귀 병합 정렬이 키 E A S Y Q U E S T I O N을 정렬하는데 이용될 때 수행되는 병합을 제시하라. *
8. 비 재귀 병합 정렬이 키 E A S Y Q U E S T I O N을 정렬하는데 이용될 때 각 반복에서 연결 리스트의 내용을 제시하라.
9. 2가지 방법 병합 보다는 오히려 3가지 방법을 수행하는 개념을 이용해서 배열을 이용한 재귀 병합 정렬을 수행하도록 하라.
10. 크기 1000의 무작위 파일에 대해 파일에서 “자연스런” 순서의 잇점을 지닌 개념이 결과로 되지 않은 텍스트에서 요청을 실험적으로 테스트하라.

13 장

외부 정렬

많은 중요한 정렬 응용은 매우 큰 파일들을 처리해야 하므로 해서, 그 크기가 너무커서 컴퓨터의 주 기억장소에 적합하지 않을정도이다. 그런 응용에 대한 적절한 방법은 중앙 처리 장치에 대한 외부적으로 처리하는 큰 양을 지니므로해서 외부 방법이라고 부른다.(지금까지 살펴본 내부 방법들과 반대되는 개념)

외부 알고리즘을 지금까지 본 것과 아주 다른 두 가지 중요한 요소가 있다. 첫째, 항목을 접근하는 비용은 어떤 장부나 계산 비용보다 더 큰 정도의 순서이다. 둘째, 이같은 높은 비용 이상으로 이용된 외부 기억장치에 따라서 접근에는 심각한 제약이 있다. 예를 들면, 자기 테이프상에 항목들은 순차적인 방법으로서만 접근된다.

외부 기억장치 종류와 비용들의 많은 다양성은 현재 기술에 매우 의존적인 외부 정렬 방법들로 만든다. 이같은 방법들은 복잡하고 그리고 많은 파라미터들이 그것의 활용도에 영향을 준다. 즉, 기술적으로 간단한 변경 때문에 좋은 방법으로 평가되지 않고 사용되지 않는다는 것은 외부 정렬에서 명백히 가능하다. 이같은 이유에 대해, 본 장에서는 특별한 구현을 개발하기 보다는 오히려 일반적인 방법들에 중점을 둔다.

간단히 외부 정렬에 대해, 문제의 “시스템” 면은 확실히 “알고리즘” 면과 같이 중요하다. 양쪽 영역은 효과적인 외부 정렬이 개발되어지는 경우에 조심스럽게 고려가 되어야 한다. 외부 정렬에서의 주된 비용은 입력과 출력이다. 어떤 사람이 매우 큰 파일을 정렬하기 위해서 효율적인 프로그램을 구현하려고 하는 좋은 연습은 먼저 큰 파일을 복사하기 위해서 효율적인 프로그램을 구현하는 것이고, 그때(만약 그것이 매우 쉬운 경우) 큰 파일에서 요소들의 순서를 거꾸로 하도록 효율적인 프로그램을 구현하는 것이다. 이같은 문제들을 효율적으로

풀기 위해서 제기되는 시스템 문제들은 외부 정렬에서 제기 되는 것들과 비슷하다. 어떤 비사소한 방법으로서 큰 외부 파일을 순열화 시키는 것이 비록 키들 비교는 없는 것이지만 그것을 정렬하는 것과 같은 정도로 어렵다. 외부 정렬에서, 외부 기억장치와 주 기억장치 사이에 데이터의 각 조각이 이동되는 수를 제한하고, 그런 변형이 이용가능한 하드웨어에 의해 효율적으로 수행이 되는 것이 관심사이다.

외부 정렬 방법은 과거의 천공 카드와 종이 테이프에 대해, 현재는 자기 테이프와 디스크에 대해 그리고 앞으로의 기술은 버블 기억장소와 비디오 디스크에 대해 적절하게 개발된다. 여러 가지 장치들 가운데 본질적인 차이는 이용가능한 기억장소의 상대적인 크기와 속도이고 데이터 접근 제약점들의 형태이다. 이같은 장치들이 널리 이용되도록 남겨두고서 많은 외부 기억 시스템을 특징화하는 접근의 두 가지 기본적으로 다른 모드들을 제시하는 것이기 때문에 자기 테이프와 디스크상에 정렬에 대한 기본적인 방법들에 중점을 둔다. 가끔 현대 컴퓨터 시스템은 여러 가지 더 싸고, 느리고 더 큰 기억장치의 “기억 장치 계층구조”를 지닌다. 고려된 대다수의 알고리즘은 그런 환경을 잘 처리하도록 하는데 적합하나 주 기억장치와 디스크나 테이프로 구성된 “두 단계” 기억장소로서 배타적으로 처리할 것이다.

정렬-병합

대부분의 외부 정렬 방법은 다음과 같은 방법을 이용한다. 즉, 정렬되어진 파일을 통해 첫 번째 과정을 하고, 그것을 내부 기억 장소의 크기에 대한 블록으로 쪼개고 그리고 이같은 블록들을 정렬한다. 그때 파일을 통해 여러 가지 과정을 거쳐서 전체 파일이 정렬될 때까지 연속적으로 더 큰 정렬된 블록들을 생성하면서 정렬된 블록들을 병합시킨다. 데이터는 이같은 방법이 대부분 외부 장치에 적합한 성질을 지닌 순차적인 방법으로서 접근된다. 외부 정렬에 대한 알고리즘들은 파일을 통해 과정의 수를 줄이고 그리고 가능한 복사의 비용에 근접토록 하는 한 개의 비용으로 줄이려고 한다.

대부분의 외부 정렬 방법의 비용은 입력과 출력에 대한 것이므로 해서, 파일에서 각 워드가 읽혀지고 쓰여진 수를 계산하므로써 정렬-병합 비용의 척도로 얻는다.(데이터에 대한 과정의 수들) 많은 응용에 대해, 고려되는 방법은 10개나 그 이하 그런 과정들의 순서에 포함된다. 이것은 단지 한 과정을 제거하는 방법들에 관심을 두는 것을 의미한다. 또한 전체 외부 정렬의 실행 시간은 위에서 제시된 “역으로 된 파일 복사” 연습과 같이 어떤 것의 실행시간에서 쉽게 추정된다.

균형화된 다중 방법 병합(Balanced Multiway Merging)

우선 적은 예제에 대해 가장 간단한 정렬-병합 프로시저의 여러 가지 단계를 통해서 추적 을 해보자. 가령 입력 테이프 상에 키 A S O R T I N G A N D M E R I N G E X A M P L E로 된 레코드를 지닌다고 하자. 이것들은 정렬되어져서 출력 테이프에 놓여지게 된 다. “테이프”를 이용하는 것은 단순히 레코드들을 순차적으로 읽혀 들이도록 하는 제약성을 의미한다. 즉, 두 번째 레코드는 첫 번째가 읽혀질 때까지 읽혀지지 않는다는 등등이다. 컴퓨터 기억장치에 세 개 레코드에 대한 공간이 충분히 있고 이용가능한 테이프는 많이 있다고 하자.

첫 번째 단계는 파일에서 세 가지 레코드를 한 번에 읽고 그것들을 세 개 레코드 블록을 만들고 그리고 정렬된 블록을 출력시키는 것이다. 이와 같이 먼저 A S O를 읽고, A O S를 출력시키고 다음으로 R T I를 읽고 블록 I R T를 출력시키는 등으로 이루어진다. 지금, 이 같은 블록들에 대한 것을 함께 병합 시키도록 하기 위해서 그림 13.1에서 제시된 구조로서 정렬 과정후에 끝이 나는 세 가지 테이프를 이용한다.

지금 크기 3인 정렬된 블록들을 병합할 준비가 되어있다. 먼저 각 입력 테이프에 대해 레 코드 단위로 잘라 읽고(기억장소에는 충분한 공간이 있어야 한다) 그리고 가장 적은 키로 된 레코드를 출력시킨다. 그때 지금 막 출력된 레코드와 같은 테이프에서 다음 레코드를 읽고, 다시 기억장소내에 가장 적은 레코드를 출력시킨다. 입력에서 세 개 워드로된 블록의 끝이 만날 때, 그 테이프는 다른 두 테이프들이 처리되고 9개의 레코드가 출력될 때까지 무시된다. 그리고 처리는 각 테이프상에 두 번째 세 개 워드 블록을 9개의 워드 블록으로 병합 시키기 위해서 되풀이 된다.(다음 병합을 위해서 준비가 된 다른 테이프상에 출력되는 것) 이같은 방법을 계속함으로써, 그림 13.2에 제시된 것과 같이 구성된 세개 긴 블록을 얻을 수가 있다.

Tape 1	A	O	S	■	D	M	N	■	A	E	X	■
Tape 2	I	R	T	■	E	G	R	■	L	M	P	■
Tape 3	A	G	N	■	G	I	N	■	E			
Tape 4												
Tape 5												
Tape 6												

그림 13.1 균형화된 세 가지 방법 병합: 첫 번째 과정의 결과

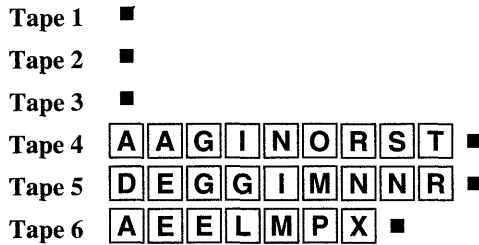


그림 13.2 균형화된 세 가지 방법 병합: 두 번째 과정의 결과

여기서 하나 이상의 세 가지 방법 병합은 정렬을 마치게 된다. 만약 각 테이프에 크기 9의 많은 블록을 지닌 더 긴 파일을 지니면, 그때 테이프 1, 2 와 3에서 크기 27의 블록으로된 두 번째 과정을 마치게 되고 그 후에 세 번째 과정은 테이프 4, 5와 6에 크기 81의 블록을 생성하는 등으로 된다. 임의의 큰 파일을 정렬하기 위해서는 6개의 테이프가 필요하다. 즉, 각 세 가지 방법 병합의 입력에 3개와 출력에 3개이다.(실제적으로는 단지 4개의 테이프로서 얻을 수가 있다. 즉, 출력은 단지 한 테이프상에서만 놓고 그리고 그 테이프에서 블록들은 병합 과정사이에 세 가지 입력 테이프에 분배를 하게 된다)

이같은 방법을 균형화된 다중 방법 병합이라고 한다. 즉, 그것은 외부 정렬에 대한 합당한 알고리즘이고 외부 정렬의 구현에 대한 좋은 시작점이다. 아래의 더 복잡한 알고리즘은 어느 정도의 빠름으로 그러나 훨씬 많지는 않은 정도로 정렬을 하게 된다.(그러나 실행시간이 외부 정렬에서 보통의 경우는 아니지만 시간으로 측정될 때, 실행 시간의 적은 양의 퍼센트 감소는 아주 의미가 있다)

가령 N 개 워드를 정렬에 의해서 처리하도록 하고, 크기 M 의 내부 기억 장소를 지닌다고 하자. 그때 “정렬”과정은 N/M 정렬된 블록을 생성한다.(이같은 추정은 한 개 워드 레코드로 가정한다. 즉, 더 큰 레코드에 대해, 정렬된 블록의 수는 레코드 크기에 의해서 곱해지는 것으로 계산된다) 만약 각 연속적인 과정에서 P -방법 병합이 수행되는 경우, 부수적인 과정들의 수는 약 $\log_P(N/M)$ 이다. 이것은 각 과정이 P 요소로서 정렬된 블록들의 수를 감소시키기 때문이다.

비록 적은 예제들이 알고리즘의 상세한 내용을 이해하는데 도움을 주지만, 외부 정렬을 처리할 때 매우 큰 파일로서 생각하는 것이 현명하다. 예를 들면, 위의 공식은 기억장소가 백만 워드인 컴퓨터에서 200백만 워드 파일을 정렬하기 위해서 4 가지 방법 병합을 이용하는 것은 전체 약 5개의 과정을 요구한다. 실행시간의 매우 거친 측정은 위에서 제시된 역으로된 파일 복사 구현에 대해 실행시간을 5배로 증가시키는 것이다.

대체 선택(Replacement Selection)

상세한 구현은 우선순위 큐를 이용해서 거대하고 효율적인 방법으로서 개발된다. 첫째, 우선순위 큐는 다중 방법 병합을 구현하는 당연한 방법을 제공한다. 더 중요한 것은 그것들이 내부 기억 장소에 적합하게 된 것 보다 훨씬 더 길게 정렬된 블록을 생성한다는 그런 방법으로서 초기의 정렬 과정에 대한 우선순위 큐를 이용한다.

P -방법 병합을 수행하는데 필요한 기본적인 연산은 병합되어진 p 블럭들의 각각에서 아직껏 출력되지 않은 가장 적은 요소들중의 가장 적은 것을 되풀이 해서 출력시키는 것이다. 가장 적은 요소는 그것이 오게되는 블럭에서 다음 요소들로서 대체된다. 크기 P 의 우선 순위 큐상에 대체 연산은 정확히 필요로 하는 것이다.(실제적으로 11장에 기술된 것과 같이 우선 순위 큐 루틴의 “간접적인” 버전은 이같은 응용에 더 적절하다) 특별히 P -방법 병합을 수행함은 11장에서 $PQ :: insert$ 프로시저를 이용해서 P 입력들 각각에서 가장 적은 요소들로 된 크기의 우선 순위를 채우므로써 시작된다.(적절히 변경되고 그 결과 가장 큰 것이라기 보다는 오히려 가장 적은 요소는 힙의 제일 위에 있다) 그때 11장에서 $PQ :: replace$ 프로시저를 이용해서(같은 방법으로 변경된 것) 가장 적은 요소를 출력하고 우선 순위에서 그것의 블럭에서 다음 요소로 대체를 시킨다.

병합 과정에서 크기 3의 힙을 이용해서 A O S와 I R T 그리고 A G N(위의 예제에서 첫 번째 병합)을 병합시키는 과정은 그림 13.3에 제시가 되어져 있다. 이같은 힙에서 “키들”은 각 노드에서 가장 적은(첫번째) 키이다. 명확히 해서, 힙의 노드들에서 전체 블록을 보게 된다. 물론, 실제적인 구현은 포인터의 간접적인 힙을 블록으로 하게 한다. 첫째, A는

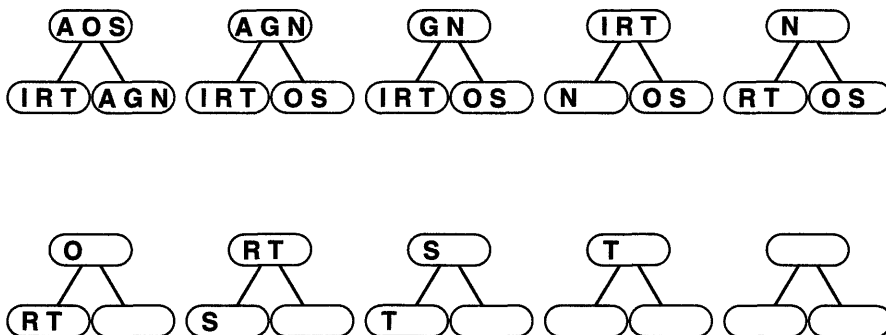


그림 13.3 크기 3인 힙으로 병합에 대한 대체 선택

출력이 되고 그래서 O(그 블록에서 다음 키)는 근의 “키”가 된다. 이것은 힙 조건을 위해 하므로 해서 A, G와 N이 포함이 된 노드들과 교환된다. 그때 A는 출력이 되고 그 블록에서 다음 키 G로 대체가 된다. 이같은 방법을 계속해서, 정렬된 파일을 생성한다.(첫번째 힙 위치에서 나타나고 출력되는 순서로 키들을 보기 위해서 그림 13.3에서 트리의 근 노드에서 가장 적은 키를 읽게 된다) 블록이 소모될 때, 표지는 모든 다른 키들보다 더 큰 것으로 고려되고 힙으로서 놓여진다. 힙은 모든 표지들을 수정할 때, 병합이 완료된다. 우선 순위 큐를 이용한 이같은 방법은 가끔 대체 선택으로 일컫는다.

이와 같이 P-방법 병합을 수행하기 위해서, $\log P$ 단계로서 출력이 되도록 각 요소를 찾기 위해서 크기 P의 우선 순위 큐상에 대체 선택을 이용하게 된다. 이같은 활용도 차이는 주먹구구식 구현이 P 단계로서 출력되도록 각 요소를 찾고 P가 정상적으로는 매우 적어서 이같은 비용이 실제로 요소를 출력하는 비용으로서 줄여주는 것이므로 특별한 실제적인 관련성이 없는 것이다. 대체 선택의 실제적인 중요성은 정렬-병합 과정의 첫 번째 부분에서 이용되는 방법이다. 즉, 병합 과정에 대한 기초를 제공하는 초기의 정렬된 블록을 형성하는 것이다.

개념은 위의 우선 순위 큐상에 가장 적은 요소를 항상 출력하므로 해서, 그것을 하나의 부가적인 조건으로 입력에서 다음 요소와 대체시키므로 큰 우선 순위 큐를 통해서(비 순서화된) 입력으로 전달된다. 즉, 만약 새로운 요소가 출력된 마지막 요소보다 더 적으면, 그때 그것은 현재 정렬된 블록의 일부분이 되지 못하므로써 다음 블록의 구성원으로 표시하고 그리고 현재 블록에서 모든 요소들보다 큰 것으로 취급한다. 표시된 요소는 우선 순위 큐의 제일 위로 만들므로써, 이전 블록은 끝나고 새로운 블록이 시작된다. 다시, 이것은 11장에서의 $PQ :: \text{insert}$ 와 $PQ :: \text{replace}$ 로 적절히 변경해서 쉽게 구현되므로써, 가장 적은 요소는 힙의 제일 위에 존재하고 $PQ :: \text{replace}$ 는 표시가 안된 요소들보다 항상 큰 것으로써 표시된 요소들을 취급하도록 변경하는 것이다.

예제 파일에서 대체 선택의 값을 명확히 나타내는 것이다. 단지 3개의 레코드만을 유지하도록 하는 내부 기억장소로서, 그림 13.4에 제시된 것과 같이 크기 5, 3, 6, 4, 5와 2의 정렬된 블록을 생성한다. 이전처럼, 키들이 힙에서 처음 위치에 차지하는 순서는 그것들이 출력하는 순서이다. 그림자로 표시된 것은 힙에서 키들이 어느 다른 블록에 속하는 것을 나타낸다. 즉, 근에 있는 요소와 같은 방법으로 표시된 요소는 현재 정렬된 블록에 속하고 그리고 다른 것들은 다음 정렬된 블록에 속한다. 힙 조건(첫번째 키는 두 번째와 세 번째 것보다

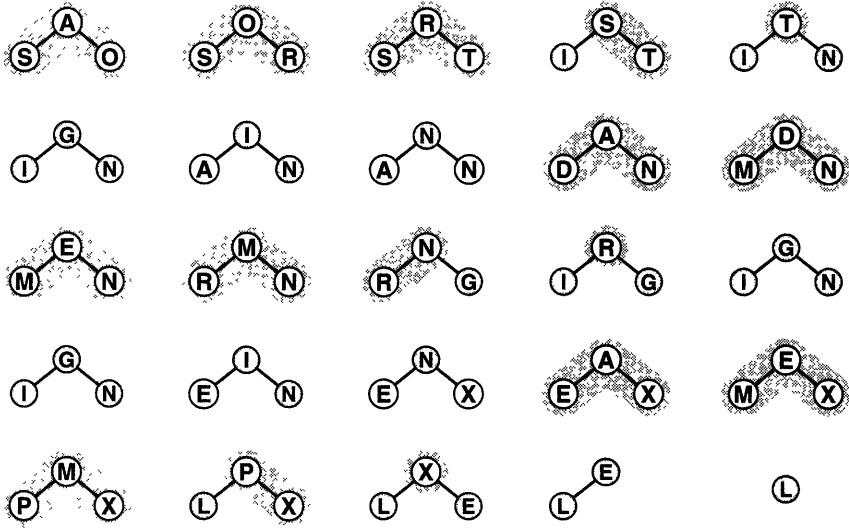


그림 13.4 초기 처리를 생성하는 대체 선택

적다)은 현재 정렬된 블록에서 새로운 요소들보다 큰 것으로 고려된 다음 정렬된 블록에서의 요소들로서 유지된다. 첫 번째 처리는 이같은 키들이 근에서 더 큰 키로서 도착되고,(그래서 그것들은 첫 번째 처리에서 포함이 안되고) 두 번째 처리는 히프에서 A N D로 끝나는 등 이기 때문에 히프에서 I N G로서 끝난다.

성질 13.1 무작위 키에 대해, 대체 선택에서 생성된 처리는 이용된 히프의 크기보다 약 2배정도이다.

이같은 성질의 증명은 오히려 복잡한 분석을 요구하나 그러나 실험적으로 검증하기는 쉽다. □

이것의 실제적인 효과는 한 개 병합 과정을 보관하는 것이다. 즉, 내부 기억장소의 크기에 관해 정렬된 처리로서 시작하고, 내부 기억장치의 크기에 약 2배인 처리를 나타내기 위해서 병합과정이라기 보다는 오히려, 크기 M 의 우선 순위 큐로서 대체 선택을 이용해서 내부 기억장소의 크기를 약 2배로 하는 처리로서 오른쪽으로 시작된다. 만약 키들에서 어떤 순서가 존재하면, 처리는 훨씬 더 길어진다. 예를 들면, 키들이 파일에서 그것 전에 M 이상의 더 큰 키들이 존재하지 않으면, 파일은 대체 선택 과정에 의해서 완전히 정렬되고 병합은 필요하지 않게 된다! 이 방법을 이용한 가장 중요한 실제적인 이유이다.

요약적으로 대체 선택 기법은 균형화된 다중 방법 병합의 “정렬”과 “병합” 단계 양쪽에 대해 이용된다.

성질 13.2 N 개 레코드의 파일은 약 $1 + \log_P(N/2M)$ 과정들에서 $P + 1$ 개 테이프와 M 개 레코드를 유지하는 내부 기억장소를 이용해서 정렬된다.

위에 논의한대로, 먼저 약 $2M$ (무작위 상황에서)나 그 이상(파일이 부분적으로 순서화되면)의 크기인 초기 처리를 생성하기 위해서 크기 M 의 우선 순위 큐로서 대체 선택을 이용한다. 그리고 그때 약 $\log_P(N/2M)$ (또는 그이하의) 병합 과정들에 관해 크기 P 의 우선 순위 큐로서 대체 선택을 이용한다. \square

실제적인 고려들

위에 나타난 정렬 방법을 구현하는 것을 마치기 위해서는 실제로 프로세서와 외부 장치들 사이에 데이터를 전송하는 입-출력 함수들을 구현하는 것이 필요하다. 이같은 함수들은 외부 정렬에 대해 좋은 활용도에 대한 명백한 열쇠이고 그리고 어떤 시스템들(알고리즘에 반대되는 것) 문제들의 조심스런 고려를 명백히 요구한다.(시스템 레벨에서 컴퓨터와 관심이 없는 여러분은 다음 몇 안되는 절들을 스쳐지나가게 된다)

구현에서의 주된 목표는 가능한 읽고, 쓰고 계산하는 것을 겹치게 하는 것이다. 대부분의 큰 컴퓨터 시스템은 이같은 겹침을 가능하도록 하는 대규모 입/출력 장치들을 조정하는 독립적인 처리 단위를 지닌다. 외부 정렬 방법에 의해서 성취된 효율성은 이용이 가능한 그런 장치들의 수에 의존을 한다.

각 파일을 읽고 쓰는 것에 대해, 이중 버퍼링(double-buffering)이라 부르는 표준 시스템 프로그래밍 기법은 계산과 입/출력을 최대화 시키는데 이용된다. 개념은 두 개 “버퍼”를 유지하는 것이다. 그 하나는 주 프로세서에 의해 이용을 하고, 다른 하나는 입/출력 장치에 이용하는 것이다.(또는 입/출력 장치를 조정하는 프로세서에) 프로세서가 버퍼를 이용해서 끝날 때, 그것은 입력 장치가 자신의 버퍼를 채울 때까지 기다리고 그리고 버퍼의 역할을 변경시킨다. 즉, 입력장치는 프로세서에 의해서 이미 이용된 데이터로 버퍼를 다시 채우는 반면에 프로세서는 막 채워진 버퍼에서 새로운 데이터를 이용한다. 같은 기법이 거꾸로된 프로세서와 장치들의 역할로 출력에서도 작동된다. 항상 입/출력 시간은 처리 시간보다 훨씬 크고 그

래서 이중 버퍼의 효과는 전적으로 계산 시간을 겹치게 한다. 즉, 이와 같이 버퍼들은 가능한 크게 존재한다.

이중 버퍼링의 어려움점은 실제로 이용가능한 기억 공간의 약 반만을 이용하는 것이다. 이것은 P 가 적지 않을 때 P -방법 병합의 경우에서 처럼 많은 버퍼가 수반이 되는 경우에 비효율적으로 된다. 이같은 문제는 병합 과정 동안에 단지 한 개의 특별한 버퍼(P 가 아니고)의 이용을 요구하는 예측(forecasting)이라 부르는 기법을 이용하여 처리하는 것이다. 그리고 이것은 어느 버퍼인가를 결정하는 것은 쉽다. 즉, 비어있는 다음 입력 버퍼는 마지막 항목이 가장 적은 것이라는 것이다. 예를 들면, A O S와 I R T 그리고 A G N을 병합할 때, 세 번째 버퍼가 먼저 비어있어야 하고 그리고 병합뒤에 첫 번째 것이 비어있게 된다. 다중 방법 병합에 대한 입력으로 겹쳐서 처리되는 간단한 방법은 이같은 규칙에 따라서 입력 장치에 의해서 채워지는 한 가지 특별한 버퍼를 유지한다. 프로세서가 비어있는 버퍼를 만날 때, 입력 버퍼가 채워질때까지(그것이 이미 채워지지 않은 경우) 기다리고 그 버퍼를 이용해서 시작하는 것을 변경시키고 그리고 예측 규칙에 따라서 방금 비어있는 버퍼를 채우기 시작하도록 입력 장치에 지시하는 것이다.

다중 방법 병합의 구현에서 행해진 가장 중요한 판단은 병합의 “순서”인 P 값의 선택이다. 테이프 정렬에 대해, 단지 순차적 접근만이 허용될 때, 이같은 선택은 쉽다. 즉, P 는 다중 방법 병합이 P 개 입력 테이프를 이용하고 한 개의 출력을 이용하므로 이용가능한 테이프 장치의 수보다 하나 적게 된다. 명백히 거기에는 적어도 두 개 입력 테이프가 존재하므로 해서 세개 테이프보다 적은 것으로된 테이프 정렬을 수행한다는 것을 이해하지는 못한다.

디스크 정렬에 대해, 임의의 위치로 접근이 허용되나 순차적인 접근 보다 다소 더 비싼 경우에, P 를 이용가능한 디스크의 수보다 하나 적은 것으로 선택하는 것이 합당하다. 예를 들면, 만약 두 개 다른 입력 파일이같은 디스크상에 존재하는 경우에 수반되는 비 순차적인 접근의 높은 비용을 피하는 것이 합당하다. 또 다른 대안으로 공통적으로 이용된 것은 P 를 충분히 크게 해서 정렬이 두 개 병합 과정에서 완료되도록 하는 것이다. 즉, 한 번의 과정에서 정렬을 수행하는 것은 항상 비합리적이다. 그러나 두 번의 과정 정렬은 합리적인 적은 P 로써 수행된다. 대체 선택은 약 $N/2M$ 처리들을 생성하고 그리고 각 병합 과정은 P 에 의해서 처리의 수를 나누므로해서, P 가 $P^2 > N/2M$ 인 가장 적은 정수로 되도록 선택하는 것을 의미한다. 1백만개 워드 기억장소를 지닌 컴퓨터에 200백만개 워드 파일을 정렬하는 예제에 대해, $P = 11$ 은 두 개 과정 정렬을 확실화하는 안전한 선택임을 의미한다. (P 의 올바른 값은 정렬 과정이 완료된후에 정확히 계산되는 것을 의미한다) P 의 가장 적은 합당한 값과 P 의 가

장 높은 합당한 값의 이같은 두 가지 대안들 사이에 가장 좋은 선택은 많은 시스템 파라미터 상에 매우 의존적이다. 즉, 양쪽 대안은(그리고 사이에 몇몇은) 고려된다.

다중 단계 병합(Polyphase Merging)

테이프 정렬에 대한 균형화된 다중 방법 병합의 한 문제는 테이프 장치의 과도한 수나 과도한 복사중의 하나를 요구하는 것이다. P -방법 병합에 대해, $2P$ 개 테이프를 이용하거나,(입력에 대해 P 개, 출력에 대해 P 개), 혹은 병합 과정 사이에 한 개 출력 테이프에서 P 개 입력 테이프에 이르는 파일의 거의 모든 것을 복사하는 것이다. 그리고 이것은 약 $2\log P(N/2M)$ 로 되는 과정의 수를 효과적으로 두배로 하게 된다. 여러 가지 명확한 테이프 정렬 알고리즘들은 적은 정렬된 블록들을 함께 병합이 되도록 하는 방법을 변경함으로써 이같은 복사의 모든 것을 가상적으로 제거한다. 이같은 방법의 가장 탁월한 것을 다중 단계 병합이라고 부른다.

다중 단계 병합에 있는 기본적인 개념은 이용가능한 테이프 장치들 가운데서(하나는 빈 것으로 남겨두고) 대체 선택에 의해서 생성된 정렬된 블록들을 분배하는 것이고 출력 테이프 중의 하나를 지시하고 그리고 입력 테이프의 역할을 바꾸는 “비어있을 때까지 병합(merge-until-empty)” 기법을 적용한다.

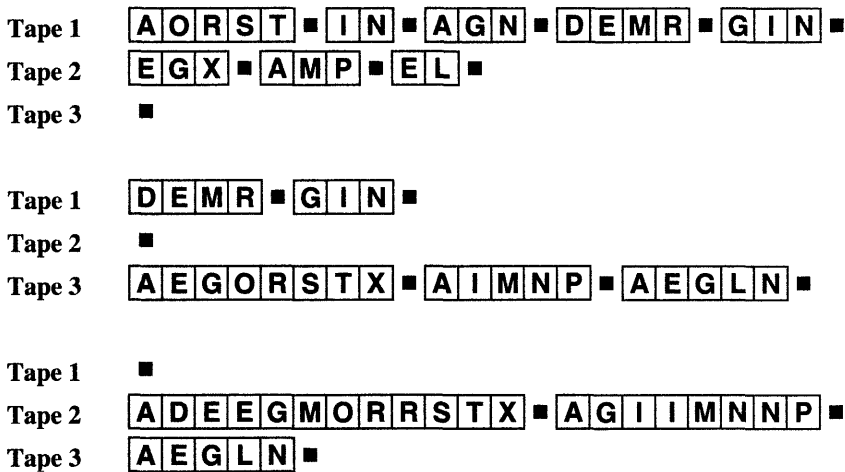


그림 13.5 세 개 테이프로 된 다중 단계 병합의 초기 단계

예를 들면, 지금 세 개 테이프가 있다고 하고 그림 13.5의 제일 위에 제시된 테이프상에 정렬된 블록들의 초기 구조로서 시작된다고 가정하자.(이것은 단지 두 개 레코드만을 유지하는 내부 기억장치로서 예제 파일에 대해 대체 선택을 적용하는 것에서 온다) 테이프 3은 처음에 비어있고 첫 번째 병합에 대해 출력 테이프가 된다. 테이프 1과 2에서 테이프 3으로 세 개로된 두 가지 방법 병합후에, 두 번째 테이프는 그림 13.5에 제시된 것과 같다. 그리고 테이프 1과 3에서 테이프 2에 이르는 두 개의 두 가지 방법 병합후에 첫 번째 테이프는 그림 13.5의 밑에 제시된것과 같이 비어있게 된다. 정렬은 두 개 더 많은 단계로서 끝난다. 첫째, 테이프 2와 3에서 테이프 1에 이르는 두 가지 방법 병합은 테이프 2에 한 개 파일을 남겨두고 테이프 1에 한 개를 남겨둔다. 그리고 테이프 1과 2에서 테이프 3에 이르는 두 가지 방법 병합은 테이프 3상에 전체 정렬된 파일이 남게 된다.

이같은 비어 있을때까지의 병합 기법은 임의의 테이프 수에 대해 처리되도록 확장이 된 것이다. 그림 13.6은 6개 테이프가 497개의 초기 처리를 정렬하기 위해 어떻게 이용되는 가를 나타낸 것이다. 만약 그림 13.6에 지시된 것과 같이 테이프 2에는 출력 테이프, 테이프 1은 61개의 초기 처리를 그리고 테이프 3에는 120개의 초기 처리를 가지는 등의 것으로 시작되면, 5가지 방법 “비어 있을 때까지 병합”을 수행한 후에 그림 13.6의 두 번째 열에서와 같이 테이프 1은 비어있고 테이프는 61개(길이) 처리가 그리고 테이프 3에는 59개의 처리를 지닌다. 여기에서 테이프 2를 다시감고 그리고 그것을 입력 테이프, 테이프 1을 다시 감고 그리고 그것을 출력 테이프, 테이프 3으로 이용케 된다. 이 방법을 계속하면, 결국에는 전체 정렬된 파일이 테이프 1에 저장된다. 병합은 모든 데이터가 수반이 되지않으나 그러나 직접인 복사 없이도 되는 많은 과정으로 분할된다.

다중 단계 병합을 구현하는데 주된 어려움은 초기 처리를 어떻게 배분하느냐에 있다. 뒤로 처리하므로써 위의 표를 생성하는 방법을 보는 것은 어렵지 않다. 즉, 각 열에서 제일 큰 수

Tape 1	61	0	31	15	7	3	1	0	1
Tape 2	0	61	30	14	6	2	0	1	0
Tape 3	120	59	28	12	4	0	2	1	0
Tape 4	116	55	24	8	0	4	2	1	0
Tape 5	108	47	16	0	8	4	2	1	0
Tape 6	92	31	0	16	8	4	2	1	0

그림 13.6 6개 테이프 다중 단계 병합에 대한 처리 분배

를 취해서 그것을 0으로 만들고 그리고 그것에 이전 열을 얻기 위해서 다른 수들의 각각을 더하는 것이다. 이것은 현재 열을 주는 이전 열에 대해 제일 높은 순서 병합을 정의하는 것에 대응된다. 이같은 기법은 어떤 테이프의 수에 대해 처리가 된다.(적어도 세 개) 즉, 제거되는 수들은 많은 흥미로운 성질을 지닌 “일반화된 피보나찌 수열”이다. 물론, 초기 처리의 수는 미리 알수가 없으므로 그리고 일반화된 피보나찌 수를 정확하게 모르기 때문이다. 이와 같이 “가상” 처리의 수는 표에서 필요로 하는 것을 초기 처리의 수를 더하는 것이다.

다중 단계 병합의 분석은 복잡하고 흥미롭고 그리고 놀라만한 결과를 나타낸다. 예를 들면, 테이프들 가운데 가상 처리를 분배하는 것에 대한 가장 좋은 방법은 특별한 단계와 필요로 하는 것 보다 더 많은 가상 처리를 이용하는 것을 포함케 된다. 이것에 대한 이유는 어떤 처리가 다른 것들보다 더 가끔 병합에서 이용된다.

많은 다른 요소들은 대부분의 효율적인 테이프 정렬 방법을 구현하는데 고려된다. 전혀 고려가 되지 않은 주된 요소는 테이프를 다시 감는 것이다. 이같은 주제는 광범위하게 공부해야 하는 것이고 그리고 많은 놀라운 방법들이 정의된다. 그러나 위의 언급한대로, 단순히 균형된 다중 방법 병합에 대해 성취되어지는 보관은 아주 제한적인 것이다. 비록 다중 단계 병합이라 할지라도, 적은 P 에 대해서만 균형화된 병합보다 더 좋은 경우이고 그리고 부수적인 것이 아니다. $P > 8$ 에 대해, 균형화된 병합은 다중 단계 병합보다 빠르고 그리고 더 적은 P 에 대해 다중 단계의 효과는 두 개 테이프를 보관하기 위해 기본적으로 된다.(두개 특별한 테이프로 된 균형화된 병합은 더 빠르다)

더 쉬운 방법

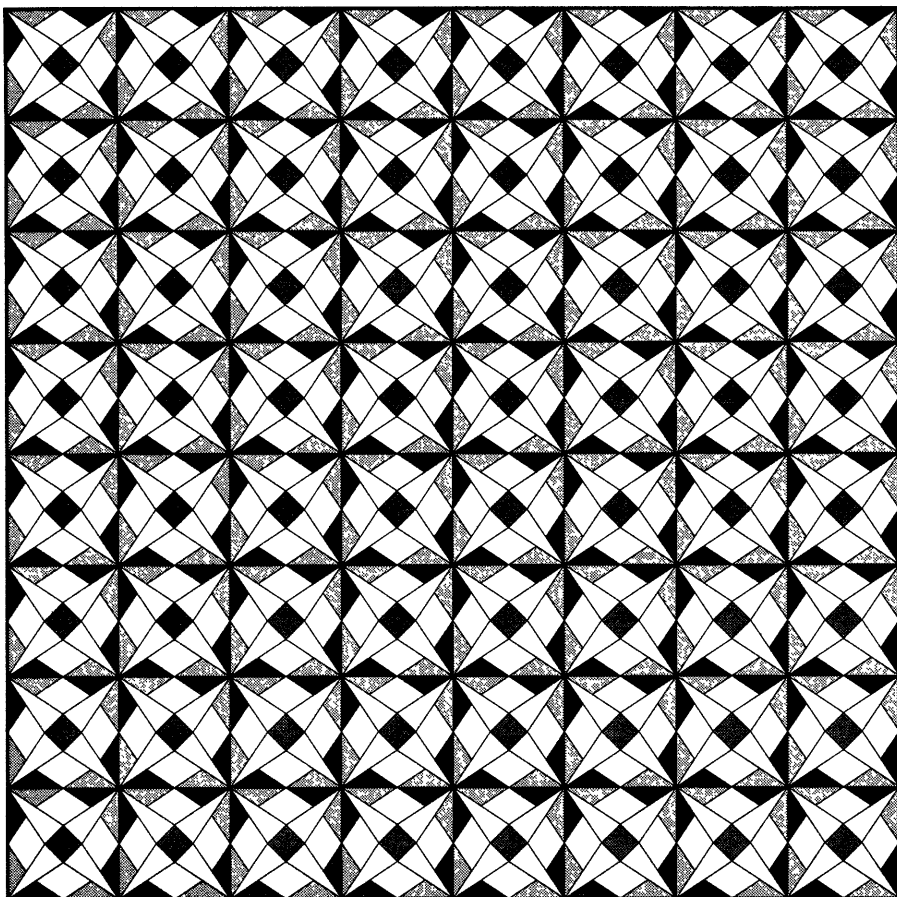
많은 현대 컴퓨터 시스템은 매우 큰 파일들을 정렬하는 방법을 구현함에 있어 간과하지 않는 큰 가상 기억장치 능력을 제공한다. 좋은 가상 기억 장치 시스템에서, 프로그래머는 데이터가 필요할 때 외부에서 내부 기억장소에 전송하는 것은 시스템의 책임을 남겨두고서 데이터의 많은 양을 나타낸다. 이같은 방법은 많은 프로그램이 상대적으로 적은 “참조의 국지성(locality of reference)”을 지닌다는 사실에 의존된다. 즉, 기억 장소에 각 참조는 다른 최근의 참조된 영역에 상대적으로 근접된 기억장소의 영역이다. 이것은 외부에서 내부 기억장소로의 전송이 일어나는 것을 의미한다. 적은 참조의 국지성으로 된 내부 정렬 방법은 기억 장소 시스템에서 매우 잘 처리가 된다.(예를 들면, 퀵 정렬은 두 개의 “국지성”을 지닌다. 즉,

대부분의 참조는 2개의 분할된 포인터중 하나에 근접된 것이다) 그러나 의미있는 보관에 대해 시스템 프로그래머가 체크하라. 즉, 어느 경우에서든 참조의 국지성을 지니지 않은 기수 정렬과 같은 방법은 가상 기억장치 시스템에서 그리고 비록 이용이 가능한 가상 기억 장치 시스템이 얼마나 잘 구현이 되는가에 따라서 쿼 정렬이 문제가 되는 것이다. 다른 한편으로 디스크 파일을 정렬하는 단순한 내부 정렬 방법의 전략은 좋은 가상 기억장치 환경에서 심각한 고려를 해야 한다.

연습 문제

1. 외부 선택을 어떻게 수행하는지를 설명하라. 즉, N 개 요소 파일에서 k 번째 가장 큰 요소를 찾는 것이다. 여기서 N 는 파일에 대해 너무 커서 주 기억장소에는 적합하지 않다.
2. 대체 선택 알고리즘을 구현하고 생성된 처리가 내부 기억장소의 크기를 약 두배정도로 요구하는 것을 테스트 하기 위해서 그것을 이용하라.
3. 대체 선택이 $M < N$ 인 크기 M 의 우선 순위 큐를 이용해서 N 개 레코드의 파일에서 초기 처리를 생성할 때 발생하는 최악의 경우는 무엇인가?
4. 만약 다른 기억장치(주 기억장치는 제외)가 이용하는데 유용하지 않으면 디스크의 내용을 어떻게 정렬하는가?
5. 단지 한 개의 테이프(그리고 주 기억장치)가 이용하는데 유용하면 디스크의 내용은 어떻게 정렬하는가?
6. 31개의 초기 처리에 대해 4개 테이프와 6개의 테이프의 다중 방법 병합을 같은 수의 테이프로 된 다중 단계 병합과 비교하여라.
7. 처음에 26, 15, 22과 28개로가 포함이 된 4개 테이프로 시작될 때 5개 테이프 다중 단계 병합은 얼마나 많은 단계를 이용하는가?
8. 4개 테이프 다중 단계 병합에서 31개의 초기 처리는 각각 한 개의 레코드 길이를 지닌다고 하자.(처음에 0, 13, 11, 7로 분배) 3가지 방법 병합에서 수반되는 파일들의 각각에서 얼마나 많은 레코드들이 존재하는가?
9. 가상 기억 장치 환경에서 매우 큰 파일 상에 처리되도록 쿼 정렬 구현에서 어떻게 적은 파일들이 처리가 되는가?
10. 외부 우선 순위 큐를 어떻게 구성하는가?(피상적으로 우선 순위 큐에서 요소들의 수가 주 기억장소에 넣기에는 너무 크기될 때 11장의 insert와 remove연산을 제공하는 방법을 설계하라)

검색 알고리즘



빈 면

14 장

기초적인 검색 방법들

많은 계산적인 일의 기본적인 연산은 검색(searching)이다. 즉, 이전에 저장된 많은 정보의 양에서 어떤 특정한 부분의 정보를 검색하는 것이다. 정상적으로 정보는 레코드라고 하는 것으로 나누어지고 각 레코드는 검색에서 이용되는 키를 지닌다. 검색의 목표는 주어진 검색 키와 일치하는 키들로 된 모든 레코드를 찾는 것이다. 검색의 목적은 처리를 위해 레코드내에(단순히 키는 아니지만) 정보를 접근하는 것이다.

검색의 응용들은 널리 사용되고 많은 다른 연산들을 포함한다. 예를 들면, 은행은 고객의 잔고에 대한 모든 상태를 파악할 필요가 있고, 여러 가지 거래의 유형을 체크하기 위해서 고객의 잔고를 통해 검색한다. 항공 예약 시스템은 어떤 방법에서는 비슷한 의미를 지니나 그러나 많은 데이터들의 수명이 오히려 짧다.

검색에 대한 자료 구조를 기술하는데 가끔 이용되는 두 가지 공통적인 용어에는 사전(dictionaries)과 기호 표(symbol tables)가 있다. 예를 들면, 영어 사전에서, “키”들은 워드이고 “레코드”는 정의, 발음과 다른 정보를 포함한 워드들과 관련이 있는 엔트리들이다. 영어 사전에서 검색에 대한 시스템을 어떻게 구현하는 가에 대해 생각하므로써 검색 방법을 배우고 평가할 준비가 되어있다. 기호 표는 프로그램에 대한 사전이다. 즉, “키”들은 프로그램에서 이용된 기호 이름이고, “레코드”들은 명명된 사물을 기술하는 정보를 포함한다.

검색에서(정렬에서와 같이) 많이 이용되고 가끔 이용되는 프로그램을 지닌다. 그래서 상세하게 여러 가지 방법으로 공부하는 것은 가치가 있다. 정렬에서와 같이, 적은 표에 대해 매우 유용한 것이고, 더 고급화된 방법으로 추출된 기본적인 기법을 제시된 것을 보므로써 시작해보자. 키 비교로서 검색이 되거나 혹은 키 값에 의해서 인덱스화 되는 배열에서의 레코드

를 저장하는 방법을 보고, 키 값에 의해서 정의된 구조를 만드는 기본적인 방법을 보게 될 것이다.

우선 순위 큐로서 검색 알고리즘은 특별한 구현에서 분리된 여러 가지 유전자 연산들을 구현하는 패키지(package)에 속하는 것으로 간주한다. 그 결과 대안의 구현으로 쉽게 대체가 된다. 관심이 있는 연산은 다음과 같다.

자료 구조를 초기화(Initialize)

주어진 키를 지닌 레코드를 검색(Searching)

새로운 레코드를 삽입(Insert)

특별한 레코드를 삭제>Delete)

두 개 사전을 더 큰 하나의 것으로 만들도록 결합(Join)

사전을 정렬(Sort) 해서 정렬된 순서로 모든 레코드를 출력

우선 순위 큐로서, 이같은 연산들 몇 가지를 결합하는 것이 가끔 편리하다. 예를 들면, 검색과 삽입 연산은 똑같은 키들로 된 레코드들이 자료 구조내에 유지되지 못하는 상황에서는 효율성에 대해서 수반된다. 많은 방법에서, 키들이 자료 구조에서 생성되지 않으면, 그때 검색 프로시저의 내부 상태는 주어진 키로서 새로운 레코드를 삽입하는데 필요한 정보를 상세하게 나타낸다.

똑같은 키로 된 레코드는 응용에 따라서 여러 가지 방법중의 하나로서 처리된다. 첫째, 명확한 키들로 된 레코드들만이 주된 검색 자료 구조에 포함되는 것으로 주장한다. 예를 들면, 이같은 자료 구조에서 각 “레코드”는 그 키를 지닌 모든 레코드들의 리스트에 연결을 포함한다. 이것은 주어진 검색 키로 된 모든 레코드들이 검색으로 되돌려지기 때문에 어떤 응용에서도 편리하다. 두 번째 가능성은 주된 검색 자료 구조에서 똑같은 키들로 된 레코드는 남겨두고, 검색에 대한 주어진 키로서 어떤 레코드를 되돌려 주는 것이다. 이것은 한 번에 한 레코드를 처리하는 응용에 대해서는 더욱 간단하다. 그리고 여기서 처리되어진 똑같은 키로된 레코드의 순서는 그리 중요하지 않는 것이다. 주어진 키로서 모든 레코드들 혹은 다른 레코드를 검색하는 메카니즘은 여전히 제공되기 때문에 알고리즘 설계에서 불편하다. 세 번째 가능성은 각 레코드가 유일한 식별자(키와는 별개로)를 지니다고 가정하고, 검색은 키로 주어진 인식자로 된 레코드를 찾는 것을 요구하는 것이다. 네 번째 가능성은 주어진 키로서, 각 레코드에 대한 특별한 함수를 부르는 검색 프로그램에 대해 배열을 하는 것이다. 혹은 어떤

더 복잡한 메카니즘이 필요하다. 검색 알고리즘을 기술할 때, 이용되는 메카니즘이 어떤 것인가를 상세히 기술함이 없이도 공식적으로 똑같은 키들로 된 레코드들이 어떻게 찾아지는가를 언급한다. 예제들은 일반적으로 똑같은 키들을 포함한다.

위에 나열된 기본적인 연산들 각각은 중요한 응용을 지니고, 기본적인 구조의 아주 많은 양의 연산이 여러 가지 조합에서 효율적인 이용으로 되는 것으로 제시된다. 이것과 다음 몇 장에서 삭제와 정렬에 대한 적절한 몇 가지 주식과 더불어 기본적인 함수들인 삽입과 검색(그리고 물론 초기화)의 구현에 집중을 할 것이다. 우선 순위 큐로서 다른 연산들과의 조합에서 효율적으로 결합하는 구현은 여기의 범위를 정상적으로는 벗어난 것이다.

순차적 검색(Sequential Searching)

검색에 대한 가장 간단한 방법은 배열에서 레코드를 단순히 저장하는 것이다. 새로운 레코드가 삽입될 때, 배열의 끝에 삽입시킨다. 즉, 검색이 수행될 때, 배열을 순차적으로 본다. 다음 코드는 간단한 구조를 이용한 기본적인 함수들의 구현을 나타내고, 검색 방법들을 구현하는데 이용되는 몇 가지 규약을 제시한 것이다.

```
class Dict
{
    private :
        struct node
        { itemType key; infoType info; };
        struct node *a;
        int N;
    public :
        Dict(int max)
        { a = new node[max]; N = 0; }
        ~Dict()
        { delete a; }
        infoType search(itemType v);
        void insert(itemType v, infoType info);
};

infoType Dict::search(itemType v) //Sequential
{
```

```

    int x = N+1;
    a[0].key = v; a[0].info = infoNIL;
    while ( v != a[--x].key);
    return a[x].info;
}

void Dict::insert(itemType v, infoType info)
{ a[++N].key = v; a[N].info = info; }

```

이것은 키들이 “관련된 정보”(info)를 다시 부르고 저장하는데 이용되는 사전적인 자료 형태를 유지하는 구현이다. 정렬과 같이, 많은 응용에서 더 복잡한 레코드들과 키들을 취급하도록 프로그램을 확장시키는 것이 필요하다. 그러나 알고리즘에서 기본적인 변화를 제공하지는 않는다. 예를 들면, itemType에 char*로 세팅하고, strcmp를 수행하기 위해서 != 연산자를 과부하(overload)시키는 것은 키로서 정수형 대신에 문자 스트링을 이용하는 패키지로 변화시키는 것이다. 혹은 info는 더 복잡한 레코드 구조에 대한 포인터이다. 이와 같이, 필드는 똑같은 키들로 된 레코드들 사이를 구별하는데 이용하는 레코드의 유일한 인식자로서 서비스 된다. 여기서, 검색은 찾은 키를 지나는 첫 번째 레코드에서(그런 레코드가 없는 경우는 infoNIL) info 필드를 되돌리는 것이다.

표지 레코드가 이용된다. 즉, 그것의 키 필드는 검색이 항상 종결되도록 하기 위해 찾은 값으로 초기화시키고 그러므로해서 내부 반복으로 하여금 단지 하나의 완료 테스트로서 코드화 되도록 하는 것이다. 표지 레코드의 info필드는 infoNIL로 세트가 되어서 다른 레코드가 주어진 키 값을 지니지 않을 때 값이 되돌려지게 된다. 이것은 여러 가지 정렬 알고리즘의 코딩을 간단히 하는 가장 큰 혹은 가장 적은 키 값을 포함하는 표지 레코드의 이용과 비슷하다.

성질 14.1 순차적 검색(배열 구현)은 비 성공적인 검색에서 (항상) $N + 1$ 번의 비교를, 성공적인 검색(평균의 경우)에 대해서는 약 $N/2$ 번의 비교를 한다.

비성공적인 검색에 대한 성질은 코드에서 직접적으로 이루어진다. 즉, 각 레코드는 어떤 특별한 키로 된 레코드가 없다는 것을 결정하도록 조사 되어야만 한다. 성공적인 검색에 대해, 만약 각 레코드가 찾아진 것과 똑같은 경우, 그때 평균 비교의 수는 $(1 + 2 + \dots + N) / N = (N + 1) / 2$ 이다. 정확히 비 성공적인 비용의 반이다. □

순차적인 검색은 레코드에 대해 연결 리스트 표현으로 된 당연한 방법으로서 채택된다.

```
class Dict
{
private:
    struct node
    { itemType key; infoType info;
      struct node *next;
      node(itemType k, infoType i, struct node *n)
        { key = k; info = i; next = n; };
    };
    struct node *head, *z;
public:
    Dict(int max)
    {
        z = new node(itemMAX, infoNIL, 0);
        head = new node(0, 0, z);
    }
    ~Dict();
    infoType search(itemType v);
    void insert(itemType v, infoType info);
};
```

연결 리스트로서 평상시와 같이, 가상 헤더 노드 head와 마지막 노드 z는 코드의 간단화를 제공한다. 그것들이 생성된 것과 같이 node들의 필드에서 채워지는 것이 더 편리하도록 하기 위해서 구성자를 이용한 형식으로 이동된다. 즉, 검색은 앞 장에서 본 것 보다 더 독창적인 작업이 요구된다.

연결 리스트를 이용하는 한 가지 이유는 정렬된 리스트를 유지하는 것을 쉽다.(아래 참조) 이것은 검색을 더 효율적으로 하는 것이다. 즉, 리스트가 정렬되어지므로 해서, 각 검색은 검색 키보다 더 적은 것이 아닌 키인 레코드를 찾을때 종료된다.

```
infoType Dict::search(itemType v) // Sorted list
{
    struct node *t = head;
    while ( v > t->key ) t = t->next;
    return ( v == t->key ) ? t->info : z->info;
}
```

정렬된 순서는 그것에 대한 비 성공적인 검색이 종료되어지는 곳에 새로운 레코드를 삽입 하므로써 유지하는 것이 쉽다.

```
void Dict::insert(itemType v, infoType info)
{
    struct node *x, *t = head;
    while ( v > t->next->key ) t = t->next;
    x = new node(v, info, t->next);
    t->next = x;
}
```

이것은 삽입, 검색과 초기화를 유지하면서 위의 배열 구현에서 처럼, 같은 추상적 자료 형태의 또 다른 구현으로서 코드화된다. 적절할 때 다른 기능들을 추가하면서, 이같은 방법으로 검색 알고리즘을 코드화가 계속된다. 그렇지 않으면, 구현은 시간과 공간 요구사항에서만 다른것으로서 응용에서 상호 교환적으로 이용된다. 예를 들면, 이같은 연결 리스트 구현에 sort 함수를 추가하는 것은 간단하다. 그러나 8장 - 12장에서의 어떤 방법들은 위의 배열 구현에 대한 sort를 추가 하기 위해서 코드화 되어진다.

성질 14.2 순차적 검색(정렬된 리스트 구현)은 성공적인 경우와 비 성공적인 경우 둘 다에 대해(평균적인 경우) 약 $N/2$ 번 비교이다.

성공적인 검색에 대해, 상황은 이전것과 같다. 비 성공적인 검색에 대해, 만약 검색이 끝 노드 z 혹은 리스트에서 요소들의 각각에 의해서 종료되는 것과 비슷하게 되는 것으로 가정 되면(“무작위” 검색 모델의 수에 대한 경우인 것), 그때 평균 비교의 수는 크기 $N + 1$ 혹은 $(N + 2)/2$ 의 표에서 성공적인 검색에서와 같다. □

또한 배열 구현에 관한 활용도 특징으로서의 순차적인 검색에서 “비 순서화된 리스트” 구현을 개발하는 것은 쉽다. 예를 들면, 만약 검색들이 상대적으로 적은 경우, 그때 상수시간 삽입의 장점을 지닌다.

만약 어떤 것이 여러 가지 레코드에 대해 상대적인 접근의 빈도를 알고 있으면, 그때 부수적인 절약은 레코드를 순서화하는 것으로 단순히 인식된다. “최적화” 배열은 시작에서 가장 많이 접근된 레코드를 놓고, 두 번째로 가장 많이 접근된 레코드는 두 번째 위치로하는 등으로 된다. 이같은 기법은 특히 적은 양의 레코드들에 가끔 접근이 되는 경우 매우 효과적이다.

만약 정보가 접근 빈도수에 대해 이용가능하면, 그때 최적 배열에 대한 근사치는 “스스로 구성된” 검색으로 이루어진다. 즉, 레코드가 접근될 때마다 리스트의 제일 앞으로 이동된다. 이같은 방법은 연결 리스트 구현이 이용될 때 더 편리하게 구현된다. 물론 실행시간은 레코드 접근 분배에 의존이 되므로써 일반적으로 방법이 어떻게 수행되는가를 예측하기는 어렵다. 그러나 많은 레코드들에 많은 접근들이 함께 근사화될 때 의 공통적인 상황에 적합하다.

이진 검색(Binary Search)

만약 레코드 집합이 큰 경우, 전체 검색 시간은 “분할-정복”을 적용한 검색 프로시저를 이용해서 줄어든다. 즉, 레코드 집합을 두 개 부분으로 나누고, 찾아진 키들의 속하는 곳이 두 개 부분중 어느곳인가를 결정하면 그 부분에 집중을 하게 된다. 레코드 집합을 나누는 합리적인 방법은 레코드들이 정렬되도록 하고, 작업이 되어지는 배열의 일부분을 제거하기 위해서 정렬된 배열로 인덱스를 이용하게 된다.

```
infoType Dict::search(itemType v) // Binary Search
{
    int l = 1; int r = N; int x;
    while ( r >= l )
    {
        x = ( l + r ) / 2;
        if ( v == a[x].key ) return a[x].info;
        if ( v < a[x].key ) r = x - 1; else l = x + 1;
    }
    return infoNIL;
}
```

주어진 키 v 가 표에 있는지를 찾기 위해서, 먼저 표의 중간 위치에 있는 요소를 비교한다. 만약 v 가 더 적으면, 표의 첫 번째 반에 존재를 하고, 만약 v 가 더 크면 표의 두 번째 반에 존재한다. 그리고 이 방법을 재귀적으로 적용한다. 단지 한 번의 재귀 부름이 야기 되므로써, 방법을 반복적으로 표현하는 것은 더 간단하다.

퀵 정렬과 기수 교환 정렬에서와 같이, 이 방법은 현재 처리가 되어지는 부분 파일들의 경계를 위해서 포인터 l 과 r 을 이용한다. 만약 이같은 부분 파일이 비어있으면, 검색은 비성공

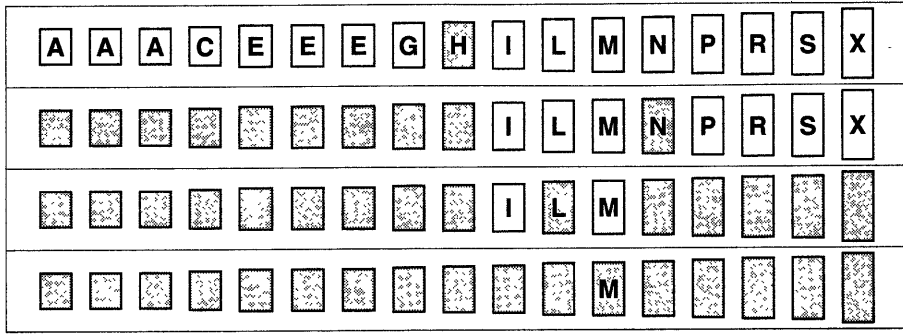


그림 14.1 이진 검색

적인 것이다. 그렇지 않으면, 변수 x 는 간격의 중간점을 가르키게 되고 거기에는 세 가지 가능성이 있다. 즉, 검색 값 v 가 $a[x]$ 에 저장된 레코드의 키 값과 같거나 보다 적거나 혹은 큰 경우에 따라서 주어진 키로 된 레코드를 찾거나 혹은 왼쪽 포인터가 $x+1$ 로 변경이 되거나 또는 오른쪽 포인터가 $x-1$ 로 변경되는 것이다.

그림 14.1은 키 A S E A R C H I N G E X A M P L E를 삽입시켜 생성된 표에서 M에 대한 검색을 할 때, 이 방법에 의해 조사된 부분 파일을 나타낸 것이다. 간격의 크기는 각 단계에서 적어도 반으로 되므로서 단지 4번의 비교만으로 검색된다. 그림 14.2는 95개 레코드로 된 더 큰 예제를 나타내고, 여기서는 검색을 위해서 단지 7번의 비교만을 요구된다.

성질 14.3 이진 검색은 성공적인 경우나 비 성공적인 경우 어느 것에 대해서도 결코 $\lg N + 1$ 번의 비교이상이 되지 않는다.

이것은 부분 파일의 크기가 각 단계에서 적어도 반으로 되어진다는 사실에서 기인된다. 즉, 비교의 수에 대한 상한은 재발생 $C_1 = 1$ 로 된 $C_N = C_{N/2} + 1$ 를 만족한다. \square

새로운 레코드를 삽입할때 요구되는 시간은 이진 검색이 높다는 것을 주시하는 것이 중요하다. 즉, 배열은 정렬된 대로 유지되고 그래서 어떤 레코드들은 새로운 레코드에 대한 공간을 주기 위해서 이동을 해야만 한다. 만약 새로운 레코드가 표에 존재하는 어떤 레코드보다 더 적은 키들을 지니면, 그때 모든 엔트리는 한 위치 이동된다. 무작위 삽입은 평균적으로 $N/2$ 개 레코드들의 이동을 요구한다. 이와 같이 이같은 방법은 많은 삽입이 수반되는 응용에 대해 이용되질 않는다. 아마도 쉘 정렬나 퀵 정렬을 이용해서 표들이 시간 계산이전에 “생성”되는 상황에 적합하다. 그리고 그때 더 큰 수의(더 효율적으로) 검색에 대해 이용된다.

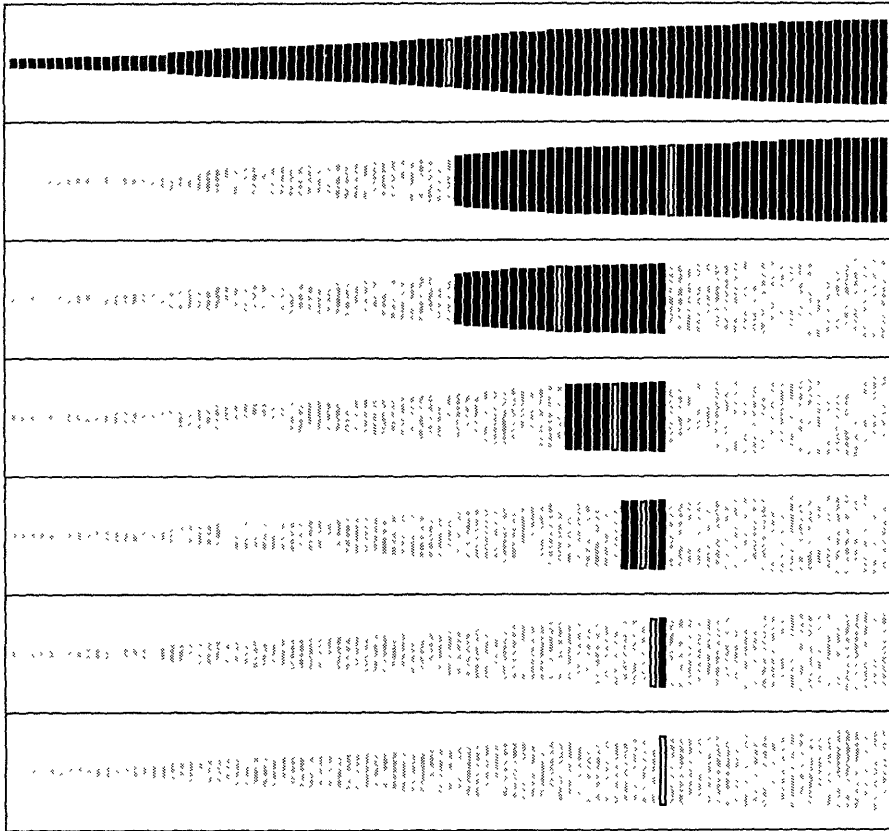


그림 14.2 더 큰 파일에서 이진 검색

현재 여러번 존재하는 키 v 와 관련된 info에 대한 성공적인 검색은 키 v 로 된 연속적인 레코드들의 블록내에서 어디에선가 끝이 난다. 만약 응용이 그런 모든 레코드들에 접근을 요구하면, 검색이 종료되는 점에서 양쪽 방향을 주사함으로써 발견된다. 비슷한 메카니즘은 특별한 간격내에 존재하는 키들로 된 모든 레코드들을 찾는 더 일반적인 문제를 해결하는데 이용된다.

이진 검색 알고리즘에 의한 비교의 연속성은 미리 결정된다. 즉, 이용된 특정한 순서는 찾은 키들의 값과 N 값에 의존을 한다. 비교 구조는 이진 트리 구조에 의해서 단순히 기술한 것이다. 그림 14.3은 키들의 예제 집합에 대한 비교 구조를 나타낸 것이다. 예를 들면, 키 M 로 된 레코드를 찾음에 있어서 먼저 H 와 비교한다. 이때 M 가 큰 경우는 다음번의 N 와 비교를 하고(그렇지않으면, C 와 비교를 한다) 그리고 L 과 비교를 한다. 검색은 4번 비교 끝에

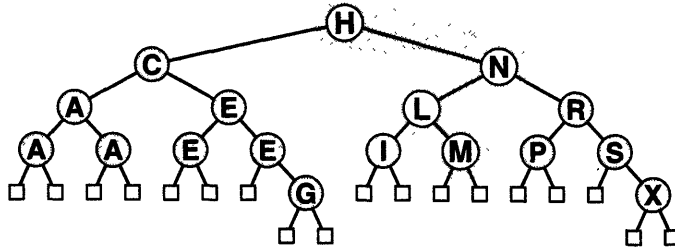


그림 14.3 이진 검색에 대한 비교 트리

성공적으로 끝난다. 아래에서 검색에 대한 명확하게 구성된 이진 트리 구조를 이용하는 알고리즘을 보게 된다.

이진 검색에서 한 가지 개선된 가능성은 찾은 키가 현재 관심이 있는 간격내에 존재하는 곳에서 더 자세하게 추측하도록 하는 것이다.(오히려 각 단계에서 중간 값을 이용하는 것보다) 예를 들면 전화 번호부에서 숫자를 보는 방법과 같다. 만약 찾을 이름이 B로 시작되면, 전화번호부의 앞쪽 근처를 보게 되나 만약 이름이 Y로 시작을 하면, 끝에서 찾게 된다. 보간 검색(interpolation search)이라 부르는 이같은 방법은 위의 프로그램에 대한 단순한 변형을 요구한다. 위의 프로그램에서, 검색에 대한 새로운 장소(간격의 중앙점)는 다음 식으로 된 문장 $x = (l+r) / 2$ 로서 계산된다.

$$x = l + \frac{1}{2}(r - l).$$

간격의 중앙은 간격의 크기 반을 왼쪽 끝점에 더하므로서 계산된다. 보간 검색은 키들이 이용 가능한 값들을 기본으로 한 곳의 추정에 의해 이같은 공식에서 1/2로 대체된다. 즉, 1/2는 v 가 $a[l].key$ 와 $a[r].key$ 사이에 간격의 중앙에 위치를 하면 적절하지만 그러나 $x =$

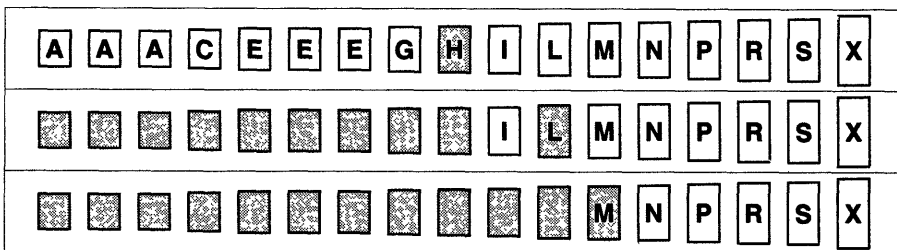


그림 14.4 보간 검색

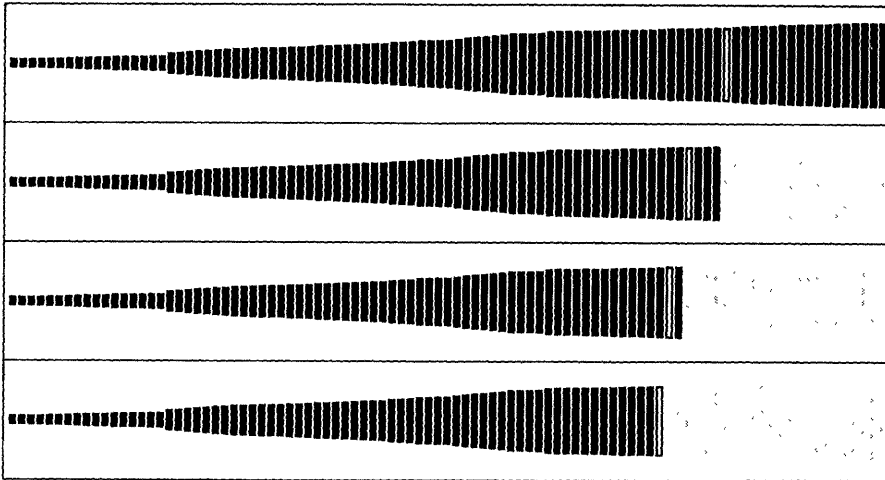


그림 14.5 더 큰 파일에서 보간 검색

$1 + (v - a[l].key) * (r - l) / (a[r].key - a[l].key)$ 는 더 좋은 추측이다.(키 값은 수치적이고 분배가 적절히 되었다고 가정하자)

예제에서 가령 알파벨의 i 번째 문자가 숫자 i 로 표현된다고 하자. 그때 M 에 대한 검색에서 조사된 첫 번째 표 위치는 $1 + (13 - 1) * (17 - 1) / (24 - 1) = 9.3...$ 이므로 9가 된다. 검색은 그림 14.4에 제시된 것과 같이 단지 3번의 단계로서 완료된다. 다른 검색 키는 훨씬 더 효율적으로 찾아진다. 즉, 예를 들면 첫 번째와 마지막번째 요소는 첫 번째 단계에서 찾아진다. 그림 14.5는 그림 14.2에서 95개의 요소들로된 파일상에서 보간 검색을 한 것을 나타낸다. 즉, 이진 검색에서 7개를 요구하는 것에 대해 이것은 단지 4번의 비교로 된다.

성질 14.4 보간 검색은 무작위 키들의 파일에서 성공적인 것과 비 성공적인 검색에 대해 $\lg \lg N + 1$ 번 비교보다 적게 된다.

증명은 본 교재의 범위를 벗어난 것이다. 이같은 기능은 실제적인 목적에 대해 상수로서 아주 서서히 증가된다. 즉, 만약 N 가 10억개이면, $\lg \lg N < 5$ 가 된다. 이와 같이, 어떤 레코드는 이진 검색에 대한 부수적인 개선점으로 단지 몇 가지 접근들(평균적인 경우)로서 찾아지게 된다. □

그러나 보간 검색은 키들의 간격이 오히려 잘 분배되어져 있는 가정하에 주로 의존된다. 즉, 실제상에 공통적으로 발생되는 나쁜 형태의 분배 키들에 의해서 나쁘게 되는 것이다. 또

한 방법은 어떤 계산을 요구한다. 즉, 적은 N 에 대해 일직선상에서 이진 검색의 $\lg N$ 비용은 충분히 $\lg \lg N$ 에 근접되므로서 보간의 비용이 가치가 있지 않게 된다. 다른 한편으로 보간 검색은 비교들이 특별히 비싼 곳의 응용에 대해 혹은 매우 큰 접근 비용들이 수반되는 곳의 외부 방법에 대한 큰 파일에 대해서 고려된다.

이진 트리 검색(Binary Tree Search)

이진 트리 검색은 전산학에서 가장 기본적인 알고리즘중의 하나로서 간주되는 간단하고 효율적인 동적 검색 방법이다. 이것은 아주 간단하기 때문에 여기서는 “기초적인” 방법으로 분류가 되나 사실 그것은 많은 상황에서 선택의 방법이다.

4장에서 어느 정도 트리에 관해 살펴보았다. 용어를 복습하도록 해 보자. 즉, 트리를 정의하는 성질은 모든 노드들이 부모라고 부르는 단지 한 개의 다른 노드에 의해서 지칭된다. 이진 트리를 정의하는 성질은 각 노드가 왼쪽과 오른쪽 연결을 지닌다. 검색에 대해, 각각 노드는 키 값으로 된 레코드를 지닌다. 이진 탐색 트리에서, 더 적은 키들로 된 모든 레코드는 왼쪽 부분 트리에 있고, 오른쪽 부분 트리에 있는 모든 레코드는 더 큰(또는 같은) 키 값을 지닌다. 연속적으로 새로운 노드를 삽입하므로 해서 생성된 이진 검색 트리들은 이같은 정의의 성질을 만족하는 것은 간단하다. 이진 검색 트리의 예제는 그림 14.6에 제시된다. 즉, 보통 때와 같이 비어있는 부분 트리는 적은 사각형 노드들로서 나타낸다.

이진 검색과 같은 검색 프로시저는 이같은 구조에 대해 그 자체를 제시한다. 주어진 키 v 로서 레코드를 찾기 위해서는 먼저 근에 대해 비교를 수행하고, 만약 그것이 더 적으면 왼쪽 부분 트리로 가고, 똑같으면 멈추고, 더 크면 오른쪽 부분 트리로 간다. 이같은 방법을 재귀적으로 적용을 하자. 각 단계에서, 현재 부분 트리와 트리의 일부분은 키 v 로 된 레코드

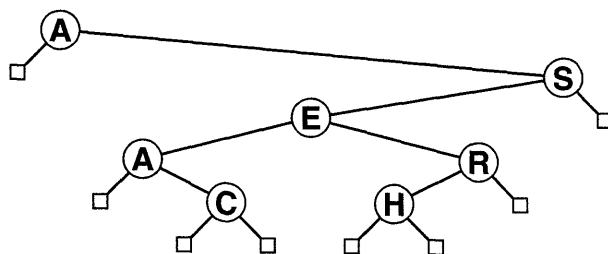


그림 14.6 이진 검색 트리

를 포함하지 않는 것이고, 이진 검색에서 간격의 크기는 줄어드는 것으로서 “현재 부분 트리”는 항상 더 적게 된다. 프로시저는 키 v로 된 레코드가 찾아지거나 혹은 그런 레코드가 없을 경우 “현재 부분 트리”가 비어있을 때에 멈추게 된다. “이진”, “검색”과 “트리”라는 단어는 이점에서는 다소 과장되게 이용되고 여러분은 이장의 앞에서 주어진 이진 검색 함수와 여기서 기술된 이진 검색 트리스이에 차이를 이해해야만 한다. 이진 검색에서는 배열에서 검색하는 함수에 의해 연속적인 비교를 위해 이진 트리를 이용한다. 즉, 여기서는 연결된 레코드의 자료구조를 실제적으로 구성하고, 검색을 위해 그것을 이용한다.

```
class Dict
{
private:
    struct node
    { itemType key; infoType info;
      struct node *l, *r;
      node(itemType k, infoType i,
          struct ndoe *ll, struct node *rr)
        { key = k; info = i; l = ll; r = rr; };
    };
    struct node *head, *z;
public:
    Dict(int max)
        { z = new node(0, infoNIL, 0, 0);
          head = new node(itemMIN, 0, 0, z); }
    ~Dict();
    infoType search(itemType v);
    void insert(itemType v, infoType info);
};

infoType Dict::search(itemType v)
{
    struct node *x = head->r;
    z->key = v;
    while ( v != x->key )
        x = ( v < x->key ) ? x->l : x->r;
    return x->info;
}
```

오른쪽 연결이 트리의 실제적인 근 노드를 지칭하고 어떤 키는 모든 다른 키 값들보다 더 적은 것인 트리 헤더 노드 head를 이용하는 것이 편리하다. head에 대한 왼쪽 연결은 이용되지 않는다. head에 대한 필요는 삽입을 논의 할 때 아래에서 더 자세하게 기술한다. 만약 노드가 왼쪽(오른쪽) 부분 트리를 지니지 않으면, 그때 그것의 왼쪽(오른쪽) 연결은 “끝” 노드인 z 를 가르키도록 한다. 순차적 검색에서와 같이, 비 성공적인 검색을 멈추기 위해서는 z 에 찾은 값을 놓는다. 이와 같이, x 에 의해서 지칭된 “현재 부분 트리”는 결코 비어있지 않으므로 모든 검색들이 “성공적”으로 된다. $z \rightarrow \text{info}$ 를 infoNIL 으로 초기화하는 것은 이용 규약에 따라서 비성공적인 검색의 지시자로 되돌려지게 된다. 이장에서의 프로그램은 z 의 연결을 결코 접근 하는 것은 아니나 나중에 보게될 더 고급화된 프로그램에 대해 z 자체를 가르키기 위해서 z 의 연결을 초기화하는 것이 가끔은 편리하다.

그림 14.6에서는 위의 제시된 것과 같이, 외부 노드에서 끝나는 모든 비성공적인 검색들로서 상상적인 외부 노드들을 가르키는 것으로 z 를 지칭하는 연결을 고려하는 것이 편리하다. 키들을 포함한 보통의 노드들을 내부 노드라고 부른다. 즉, 외부 노드들을 소개함으로써, 비록 구현에서 모든 외부 노드들이 한 개의 노드 z 에 의해서 표현이 되지만 모든 내부 노드는 트리에서 두 개의 다른 노드들을 가르키는 것이라고 말할 수있다. 그림 14.7은 이같은 연결들과 가상 노드들을 명확히 제시하는 것이다.

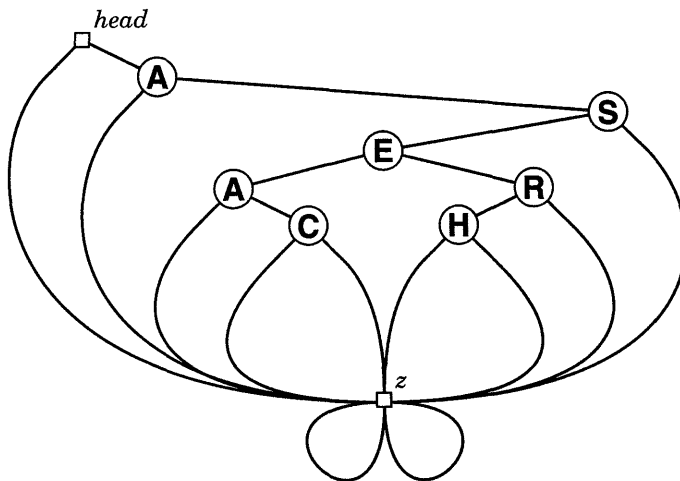


그림 14.7 이진 검색 트리(가상 노드들로 된 것)

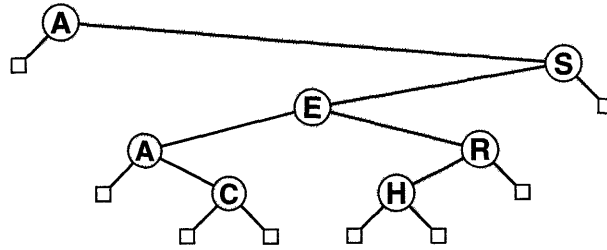


그림 14.8 이진 검색 트리에서(I에 대해) 검색

그림 14.8은 search를 이용해서 I가 예제 트리에서 찾아지는 것을 나타낸다. 첫째, 근에서 키인 A를 비교하는 것이다. I가 더 크므로 해서, A를 포함한 노드의 오른쪽 자식의 키인 S에 대해서 다음번에 비교를 한다. 이같은 방법을 계속하므로 해서, I는 그 노드의 왼쪽인 E에 대해 다음에 비교를 하고 그리고 R 그뒤에 H를 비교한다. H를 포함한 노드에서 연결들은 z에 대한 포인터이므로써 검색이 종료된다. 즉, I는 z에서 자신과 비교를 하고 검색은 비성공적인 것으로 된다.

노드를 트리에 삽입하기 위해서는, 그것에 대한 비성공적인 검색을 수행하고 검색이 종료 되는 점인 z의 위치에 그것을 첨가시키게 한다. 삽입을 수행하기 위해서, 다음 코드는 트리의 아래로 진행하는 것과 같이 x의 부모인 p의 상태를 파악 하는 것이다. 트리의 밑에 도달 될 때($x == z$), p는 그것의 연결이 삽입된 새로운 노드를 가르키는 것으로 변경되는 노드를 가르킨다.

```
void Dict::insert(itemType v, infoType info)
{
    struct node *p, *x;
    p = head; x = head->r;
    while ( x != z )
        { p = x; x = ( v < x->key ) ? x->l : x->r; }
    x = new node(v, info, z, z);
    if ( v < p->key ) p->l = x; else p->r = x;
}
```

이같은 구현에서, 트리에서 이미 어떤 키와 동등한 키인 새로운 노드가 삽입이 될 때, 트리에서 노드의 오른쪽에 삽입될 것이다. 이것은 똑같은 키로 된 노드들은 z를 만날때까지 search가 종료되는 점에서 검색을 단순히 계속함으로써 찾아진다.

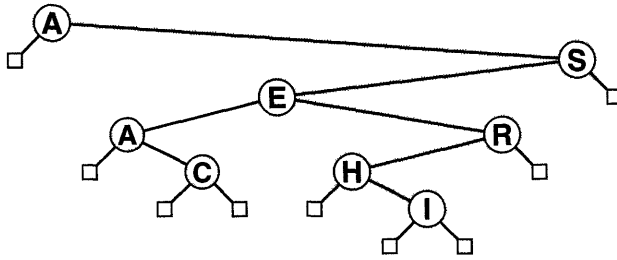


그림 14.9 이진 검색 트리에서 (I의)삽입

그림 14.9에서의 트리는 키 A S E A R C H I가 처음에 비어있는 트리에 삽입이 될 때의 결과를 나타낸 것이다. 그림 14.10은 N G E X A M P L E가 추가 될 때 예제의 완성을 나타낸 것이다. 여러분은 이같은 트리에서 똑같은 키들의 위치에 특별한 주의를 기울려야만 한다. 예를 들면, 비록 세 개 A가 트리를 통해서 퍼져나간다 할찌라도 그것들 “사이”에 키는 없다.

sort 함수는 이진 검색 트리가 이용될 때 거의 자유롭게 된다. 그 이유는 이진 검색 트리 로 하여금 바른 방법으로서 보는 경우 정렬된 파일을 나타내기 때문이다. 그림에서, 키들은 왼쪽에서 오른쪽으로 읽을 경우(그것들의 높이와 연결을 무시하면서) 순서적으로 나타난다. 프로그램은 작업할 연결들만을 지나나, 정렬 방법은 이진 검색 트리의 정의되는 성질에서 직접적으로 이루어진다. 이것은 퀵 정렬에서 요소를 분할하는 것과 비슷한 역할을 하는 트리의 근 노드로서 퀵 정렬과 현저하게 비슷한 정렬 방법을 정의한다.(왼쪽에 있는 키들은 더 큰 것이 없고 오른쪽에 있는 키들은 더 적은 것이 없다) 특별히, 5장의 기초적인 재귀인 인 오 더 트리 운행으로 작업을 수행하는 것이다. 이 경우에서, 만약 x 는 z 즉, 인수 $x \rightarrow l$ 로 된 인수 자신을 부르는 것이 아니면, 그때 `visit(x)`를 부르고, 인수 $x \rightarrow r$ 로 된 자신을 부르는 5장의 기본적 루틴인 `inorder(struct node *x)`가 있는 `Dict::inorder(head->r)`를 단순히 부르는 `class` 연산 `Dict::traverse()`를 첨가하는 것이다. 예를 들면, 그때 x 의 키 필드를 출력시키기 위해서 `Dict::visit(struct node *x)`를 구현하는 경우, `traverse`에 부름은 정렬된 순서로서 전체 트리를 출력하는 것이다. 혹은 27장에서 보는 바와 같이, 더 복잡한 `visit`는 더 복잡한 알고리즘으로 된다.

이진 검색 트리상 알고리즘의 실행시간은 트리의 그림자들에 아주 의존적이다. 가장 좋은 경우, 트리는 근과 각 외부 노드사이에 약 $\lg N$ 노드들로서 그림 14.3과 같이 그림자로 되어

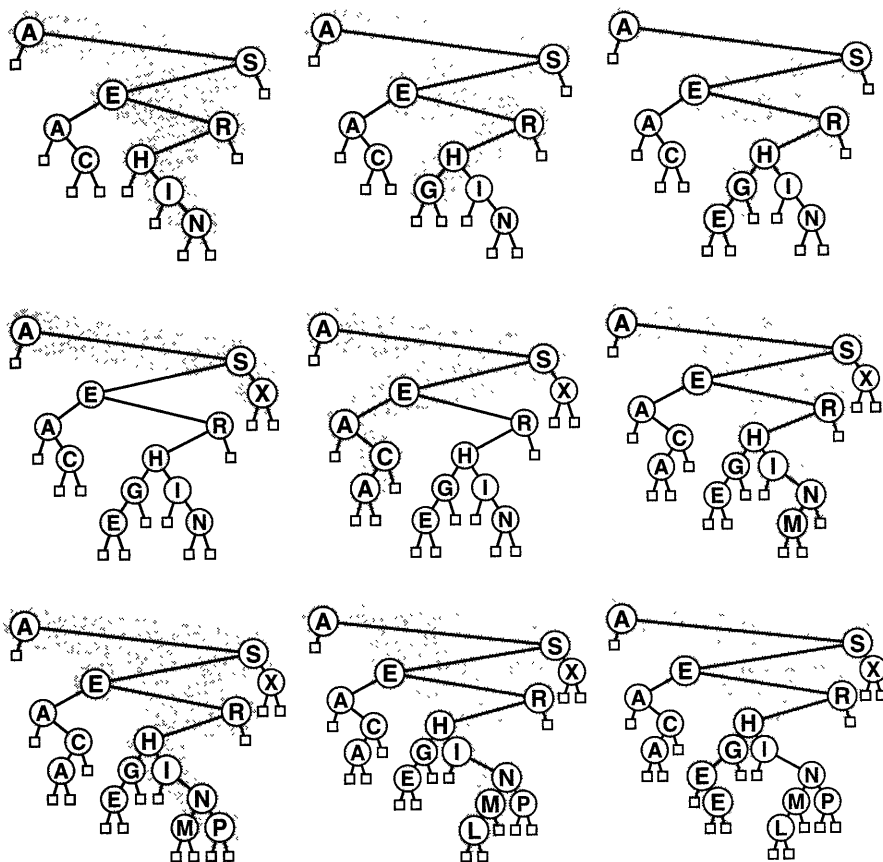


그림 14.10 이진 검색 트리 생성

졌다. 삽입된 첫 번째 요소가 트리의 근이 되기 때문에 평균적으로 대수의 검색 시간을 나타낸다. 즉, 만약 N 키들이 무작위로서 삽입되면, 그때 이같은 요소는 키들을 반으로 나누는 것이고(평균적으로), 대수 검색 시간을 나타낸다.(부분 트리에 같은 인수를 이용해서) 정말로, 그것들이 똑같은 키들에 대한 것이 아니면, 이진 검색에 대한 비교 구조를 기술하는 위에 주어진 트리가 생성된다. 이것은 모든 검색에 대한 대수 실행 시간으로서 가장 좋은 경우의 알고리즘이 된다. 실제적으로 무작위로 된 상황에서, 근은 어떤 키로서 존재하는 것과 똑같으므로 해서 그런 완전히 균형된 트리는 극단적으로 드물다. 그러나 무작위 키들이 삽입되는 경우, 트리들은 균형적인 것으로 판명된다.

성질 14.5 이진 검색 트리에서 검색이나 삽입은 N 개 무작위 키들로 생성된 트리에서 평균적으로 약 $2 \ln N$ 비교를 한다.

트리에서 각 노드에 대해, 그 노드에 대한 성공적인 검색에 대해 이용되는 비교의 수는 근에 대한 거리이다. 모든 노드들에 대한 이같은 거리들의 합은 트리의 내부 경로 길이(internal path length)라고 부른다. 내부 경로 길이를 N 으로 나누면, 성공적인 검색에 대한 평균 비교의 수를 얻는다. 그러나 만약 C_N 가 N 노드의 이진 검색 트리에서 평균 내부 경로 길이로 나타내면, $C_1 = 1$ 인 다음과 같은 재발생을 얻는다.

$$C_N = N - 1 + \frac{1}{N} \sum_{1 \leq k \leq N} (C_{k-1} + C_{N-k})$$

($N - 1$ 은 트리에서 다른 $N - 1$ 노드들의 각각 경로 길이에 대해 근을 1로 하는 사실을 고려한 것이다. 즉, 식의 나머지는 근에서 키들이(삽입된 첫 번째) 크기 $k - 1$ 과 $N - k$ 의 무작위 부분 트리를 남겨두고서 k 번째 가장 큰 것과 똑같다) 그러나 이것은 퀵 정렬에 대해 9장에서 해결된 것과 같은 재 발생이고, 제시된 결과를 나타내기 위해서 같은 방법으로서 쉽게 해결된다. 비 성공적인 검색에 대한 인수는 비록 다소 더 복잡하지만 비슷하다. \square

그림 14.11은 95개의 요소들의 무작위 순열에서 생성된 큰 이진 검색 트리를 나타낸다. 그것은 짧은 경로와 어떤 긴 경로를 지나는 반면에, 아주 잘 균형화된 것으로서 특성화된다. 즉, 어떤 검색은 12번 이하의 비교를 하고, 트리에서 어떤 키를 찾기 위해서 비교의 “평균” 수는 이진 검색에 대해 5.74와 비교되는 것으로 7.00이다.(무작위의 비성공적인 검색에 대한 평균 비교의 수는 성공적인 경우보다 하나 더 많다) 더구나 새로운 키는 같은 비용으로서 삽

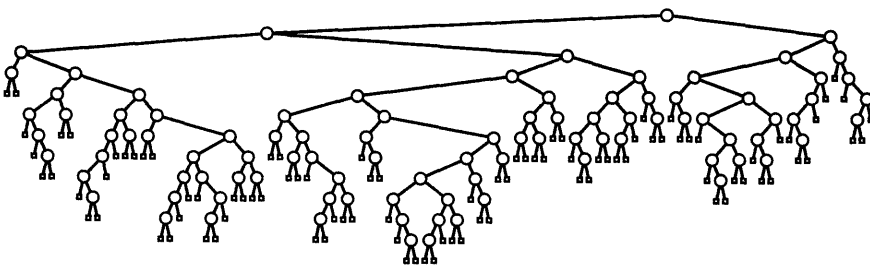


그림 14.11 큰 이진 검색 트리

입이 되고 이진 검색에 이용되지 않은 유연성을 제공한다. 그러나 만약 키들이 무작위적으로 순서화 되어있지 않으면, 알고리즘은 나쁘게 수행된다.

성질 14.6 최악의 경우에서, N 개 키로된 이진 검색 트리의 검색은 N 번의 비교를 한다.

예를 들면, 키들이 순서적으로 삽입될 때(또는 역순으로), 이진 트리 검색 방법은 이 장의 시작에서 본 순차적인 검색 방법보다 좋지 않다. 더구나, 같은 최악의 경우로 되어진 트리의 많은 다른 형태들로 존재한다.(예를 들면, 키들 A Z B Y C X ...이 처음에 비어있는 트리로 그같은 순서로서 삽입될 때 형성된 트리를 고려하자) 다음 장에서 이같은 최악의 경우를 제거하고, 모든 트리를 가장 좋은 경우의 트리와 같은 것으로 보이도록 하는 기법을 조사케 된다. □

삭제(Deletion)

이진 트리 구조를 이용한 기본적인 *search*, *insert*와 *sort* 함수들에 대해 위에 주어진 구현들은 아주 간단하다. 그러나 이진 트리는 검색 알고리즘에서 재발생 주제의 좋은 예제를 제공한다. 즉, 삭제 함수는 구현하기가 아주 애매하다.

그림 14.12에서 왼쪽에 제시된 트리를 고려하자. 노드가 자식이 없는 경우 즉, L이나 P(그것의 부모에서 적절한 연결을 비어있도록 함으로써 제거가 된다)와 같은 경우 혹은 A, H 혹은 R과 같은(자식에서 연결을 적절한 부모 연결로 이동하는 것) 자식이 단지 한 개를 지나는 경우 또는 그것들의 두 자식중의 하나가 N과 같이(부모를 대체하기 위해 그 노드를 이용) 자식이 없는 경우에 노드 삭제가 쉽다. 그러나 E와 같은 트리에서 더 높은 곳에 있는 노드에 대해서는 어떤가?

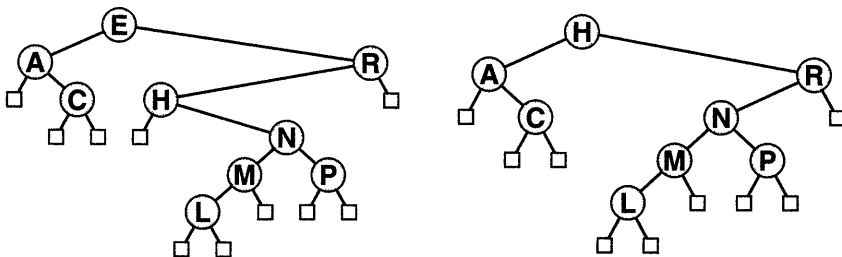


그림 14.12 이진 검색 트리에서(E의) 삭제

그림 14.12는 E를 삭제하는 한 방법이다. 즉, 그것을 다음 제일 높은 키(이경우는 H)로 대체를 한다. 이같은 노드는 기껏해야 한 노드를 지니고,(그것과 삭제된 노드사이에 노드가 없기 때문에 그것의 왼쪽 연결은 비어있어야만 한다) 쉽게 제거된다. 그림 14.12에서 왼쪽에 있는 트리에서 E를 제거하기 위해서, R의 왼쪽 연결을 H의 오른쪽 연결(N)에 대해 가르키도록 하고, E를 포함한 노드에서 H를 포함한 노드에 이르기까지 연결들을 복사하고, head->r을 H로 한다. 이것은 그림에서 오른쪽에 있는 트리를 나타낸다.

이같은 모든 경우들을 처리하는 코드는 검색과 삽입에 대해 간단한 루틴보다는 오히려 더 복잡하나, 다음 장에서 수행되는 더 복잡한 트리 처리에 대해 준비를 하기 위해 보는 것도 가치가 있다. 다음 프로시저는 키 v로 된 트리에서 만난 첫 번째 노드를 삭제한다.(대안으로서, 삭제될 노드를 인식하기 위해서 info를 이용한다) 변수 p는 트리에서 x의 부모의 상태를 파악하고, 변수 c는 삭제되어진 노드의 후계 노드를 찾는데 이용된다. 삭제후, x는 p의 자식이 된다.

```
void Dict::remove(itemType v)
{
    struct node *c, *p, *x, *t;
    z->key = v;
    p = head; x = head->r;
    while ( v != x->key )
        { p = x; x = ( v < x->key ) ? x->l : x->r; }
    t = x;
    if ( t->r == z ) x = x->l;
    else if ( t->r->l == z ) { x = x->r; x->l = t->l; }
    else
    {
        c = x->r; while ( c->l->l != z ) c = c->l;
        x = c->l; c->l = x->r;
        x->l = t->l; x->r = t->r;
    }
    delete t;
    if ( v < p->key ) p->l = x; else p->r = x;
}
```

첫째 프로그램은 트리에서 t의 위치를 얻기 위해서, 정상적인 방법으로 트리를 검색한다. (실제적으로, 이같은 검색의 주된 목적은 p로 세트되고 그래서 또 다른 노드는 t가 지나간

후에 연결된다) 다음으로 프로그램은 다음 세 가지 경우를 체크한다. 만약 t 가 오른쪽 자식을 가지지 않은 경우, 삭제후에 p 의 자식이 t 의 왼쪽 자식이 된다.(그림 14.12에서 C, L, M, P와 R에 대한 경우이다) 만약 t 가 왼쪽 자식이 없는 오른쪽 자식을 가지면, 그때 그 오른쪽 자식은 t 에서 복사된 왼쪽 연결로서 삭제후에 p 의 자식이 된다.(그림 14.12에서 A와 N의 경우에 대한 것이다) 그렇지 않으면, x 는 t 의 오른쪽에 대한 부분 트리에서 가장 적은 키로 된 노드로 한다. 즉, 그 노드의 오른쪽 연결은 부모의 왼쪽 연결에 대해 복사가 되고, 연결의 양쪽은 t 에서 세트된다.(그림 14.12에서 H와 E에 대한 경우이다) 경우의 수를 적게 하기 위해서, 비록 왼쪽을 보므로써 어떤 경우에서든지 쉽게 될지라도 이같은 코드는 오른쪽을 보므로써 항상 삭제된다.(예를 들면, 그림 14.12에서 H를 삭제하기 위함)

접근은 체계적이지 못하고 오히려 많은 방법들을 지닌다. 예를 들면, 뒤에 있는 것 대신에 삭제되기 전에 키를 즉각적으로 이용하지 않는 이유는 무엇인가? 여러 가지 비슷한 변형이 제시가 되어지지만, 차이는 비록 위의 알고리즘이 매우 큰 수의 무작위 삭제-삽입 쌍들에 영향을 받는 경우 다소 균형이 잡히지 않는 트리(평균 높이는 \sqrt{N} 에 비례하는 것)로 남겨두는 경향이 있는 것을 제시하지만 실제적인 응용에서는 주시되지 않는다.

삭제에 대해 의미있는 더 복잡한 구현을 요구하는 것이 바로 실제적이고 전형적인 검색 알고리즘이다. 즉, 키들 자신은 구조에 대해 완전한 것이고, 키의 제거는 복잡한 수리상태를 포함한다. 적절한 한 가지 대안은 가끔 노드들이 자료 구조에서 남겨지나, 검색 목적에 대해 “삭제된”것으로서 표시되어지는 소위 *lazy* 삭제이다. 위의 코드에서, 이것은 검색을 중단하기 전에 그런 노드에 대해 하나 더 체크를 첨가하므로써 구현된다. “삭제된” 노드들의 많은 수는 시간이나 공간의 과도한 소비로 되지 않으나, 많은 응용에 대해 문제가 되지 않은 것으로 판명된다. 대안으로 “삭제된” 노드들을 남겨두므로써 전체 자료 구조를 주기적으로 다시 생성할 수가 있다.

간접 이진 검색 트리들

11장의 히프에서 본것과 같이, 많은 응용에 대해 레코드를 찾는 것을 단순히 도와주고, 그것들 주위로 이동하지 않도록 하는 검색 구조를 원한다. 예를 들면, 키들로 된 레코드들의 배열을 지니고 어떤 키와 일치하는 레코드의 배열로 인덱스를 주는 search루틴을 원한다. 혹은 검색 구조에서 주어진 인덱스로서 레코드를 제거하기를 바라나 어떤 다른 이용에 대해서는 배열이 여전히 유지된다.

그런 상황에서 이진 검색 트리를 채택하기 위해서, 단순히 노드들의 info 필드를 배열 인덱스로 하는 것이다. 그때 검색 루틴으로 하여금 직접적으로 레코드에서 키들을 접근시키도록 하는 key 필드를 제거하는 것이다. 즉, $\text{if } (v < a[x \rightarrow \text{info}]) \dots$ 와 같은 명령을 통해서이다. 그러나, 키를 복사하도록 하고, 주어진 것과 같이 위의 코드를 이용하는 것이 좋다. 이것은 키들의 특별한 복사를 이용하는 것이나, (배열에서 하나, 트리에서 하나) 이것은 한 개 이상의 배열에 대해서 혹은 27장에서 보논바와 같이같은 배열에서 한 개 이상의 키 필드에 대해서 이용되도록 하는 같은 함수를 허용하는 것이다. (이것을 이루기 위해서는 다른 방법들이 있다. 예를 들면, 프로시저는 레코드에서 키들을 추출하는 각 트리와 관련된 것이다)

이진 검색 트리에 대한 “간접적인 수단”을 이루는 또 다른 직접적인 방법은 전적으로 연결된 구현으로서 처리하고, 3장에서 논의된 것과 같이 직접적인 배열 표현을 이용하는 것이다. 모든 연결들은 key 필드와 l과 r 인덱스 필드를 포함한 배열 $a[0], \dots, a[N+1]$ 으로 인덱스로 되게 한다. 그때 $x \rightarrow \text{key}$ 와 $x = x \rightarrow l$ 과 같은 연결 참조는 $a[k].\text{key}$ 와 $x = a[x].l$ 과 같은 배열 참조로 된다. new에 대한 부름은 이용되지 않는다. 그 이유는 트리는 레코드 배열내에 존재를 하기 때문이다. 즉, 가상 노드들은 $\text{head} = 0$ 와 $z = 1$ 로 세트시키므로 해서 할당된다. 그리고 구성자는 레코드 배열에서 다음번 비어있는 공간에 대한 포인터를 단순히 증가시키고, 필드를 채운다.

레코드의 큰 배열을 검색하는데 있어서 도움을 주는 이진 검색 트리를 구현하는 방법은 이전 절에서와 같이 키를 복사하는 특별한 경비를 피하고 new에 의해서 내포된 기억장치 할당 메카니즘의 불필요한 동작을 제거하기 때문에 많은 응용에서 선택된다. 그것의 단점은 사용되지 않은 연결로 인해서 레코드 배열에서 공간을 소비하는 것이다.

세 번째 대안은 3장의 연결 리스트로 수행한 것과 같이 병렬 배열을 이용한다. 이같은 구현은 키들, 왼쪽 연결과 오른쪽 연결에 대해 각각 하나씩 된 세 가지 배열을 이용한 것을 제외하고는 이전 절에서 기술된 것과 같다. 이것의 장점은 유연성이다. 특별한 배열(각 노드와 관련된 특별한 정보)들은 전혀 트리 처리 코드를 변경시키지 않고도 쉽게 추가가 되고 검색 루틴이 노드에 대한 인덱스를 줄 때, 그것은 모든 배열들을 즉각적으로 접근하는 방법이 주어진다.

연습 문제

1. 정렬된 배열에서 레코드들을 유지시키면서 성공적인 경우와 비성공적인 경우 둘다에 대해 평균적으로 약 $N/2$ 단계로 되는 순차적인 검색 알고리즘을 구현하라.
2. 키 E A S Y Q U E S T I O N으로 된 레코드들이 스스로 구성된 검색 휴리스틱 (heuristic)을 이용해서 search와 insert로된 초기에 비어있는 표에 넣는 후에 키들의 순서는 어떻게 되는가?
3. 이진 검색의 재귀 구현은 무엇인가?
4. 가령 $1 \leq i \leq N$ 에 대해 $a[i] == 2*i$ 이라 하자. $2k-1$ 에 대한 비성공적인 검색동안에 보간 검색에 의해서 얼마나 많은 표 위치를 조사하는가?
5. 키 E A S Y Q U E S T I O N으로된 처음에 비어있는 트리 레코드로 삽입하는 것에서 결과적으로 되는 이진 검색 트리를 그려라.
6. 이진 트리의 높이를 계산하는 재귀 프로그램을 기술하십시오. 즉, 근에서 외부 노드에 이르는 가장 긴 거리이다.
7. 이진 트리에서 접근되는 검색 키들이 얼마나 되는 가에 대해 미리 추정된 시간을 가진다고 하자. 키들은 접근에서 비슷한 빈도로서 증가되거나 혹은 감소되어지는 순서로서 트리에 삽입이 되는가? 그러면 그 이유는?
8. 트리에서 똑같은 키들을 함께 유지하도록 하는 이진 트리 검색을 변형하십시오.(만약 트리에서 어떤 다른 노드들이 주어진 노드와 같은 키를 지니면, 그것의 부모나 혹은 자식들 중의 하나는 똑같은 키를 지닌다)
9. 순서적인 이진 검색 트리에서 키들을 인쇄하기 위한 비 재귀 프로그램을 기술하십시오.
10. 키 E A S Y Q U E S T I O N으로된 처음에 비어있는 트리 레코드로 삽입이 되고 그 다음에 Q를 삭제하는 것으로 되는 이진 검색 트리를 그려라.

빈 면

15 장

균형 트리

이전장에서 이진 트리 알고리즘은 널리 사용되는 응용에 대해 매우 잘 처리가 되나, 그것들은 나쁜 최악의 경우의 문제를 지닌다. 더욱이 퀵 정렬과 같이, 나쁜 최악의 경우는 알고리즘의 사용자가 그것에 대해 지켜보지 않았을 경우에 실제로 발생이 되는 것은 놀라만한 사실이다. 이미 순서적으로 되어진 파일들, 역 순으로 된 파일들, 크고 적은 키들로 번갈아 된 파일들 혹은 간단한 구조를 지닌 어떤 큰 세크먼트로 된 파일들은 이진 트리 검색 알고리즘으로 하여금 매우 나쁘게 수행되도록 한다.

퀵 정렬과 더불어서, 상황을 개선하는 유일한 의지는 무작위적으로 다시 정렬하는 것이다. 즉, 무작위 분할 요소를 선택함으로써, 최악의 경우에서 보관되어야할 확률의 법칙에 의존한다. 다행히도, 이진 트리 검색에 대해, 훨씬 더 잘 수행된다. 즉, 이같은 최악의 경우가 발생되지 않도록 하는 일반적인 기법이 존재한다. 균형(balancing)이라 부르는 이같은 기법은 여러 가지 다른 “균형 트리” 알고리즘에 대한 기초로서 이용된다. 그런 알고리즘에서 자세히 보도록 하고, 이용된 다른 방법들의 몇 가지와 어떻게 관련되는 가를 간단히 논의를 한다.

아래에서 명백히 된것과 같이, 균형 트리 알고리즘을 구현하는 것은 “수행 된 것보다 더 쉽게 말하는” 경우가 확실하다. 가끔, 알고리즘에 숨겨진 일반적인 개념은 쉽게 기술되나, 구현은 특별하고 체계적인 경우들의 문제이다. 이 장에서 개발된 프로그램은 중요한 검색 방법 일 뿐만 아니라 또한 알고리즘을 구현하기 위해서 “고급” 기술과 “저급” C++ 프로그램 사이에 관계를 잘 나타낸다.

하향식 2-3-4 트리들

이진 검색 트리들에서 최악의 경우를 제거하기 위해서는 이용된 자료구조로 부터 어떤 유연성을 필요로 한다. 이같은 유연성을 얻기 위해서, 트리에서 노드들이 한 개이상 키를 유지한다고 가정하자. 특별히, 각각이 두 개와 세 개 키로 지니는 것인 3-노드(3-node)들과 4-노드(4-node)들을 허용하는 것이다. 3-노드는 그것을 나타내는데 3개의 연결을 지닌다. 즉, 그것의 양쪽 키들보다 더 작은 키들로 된 모든 레코드에 대해 한 개, 두 개 키들사이에서 키들로서 된 모든 레코드에 대해 한 개 그리고 그것의 키들 양쪽 보다 더 큰 키들로 된 모든 레코드에 대해 한 개이다. 비슷하게, 4-노드는 4개의 연결을 지닌다. 즉, 키들에 의해 정의되는 간격 각각에 대해 한 개다.(표준 이진 검색 트리에서 노드들을 2-노드라고 부른다. 즉, 한 개 키와 두 개의 연결을 지닌다) 아래에서 이같은 확장된 노드들에서 기본적인 연산을 정의하고 구현하는데는 몇 가지 효율적인 방법들이 있다. 트리를 형성하기 위해 그것들이 어떻게 함께 놓여지는 가를 보고, 그것들을 편리하게 처리될 수 있다고 가정하자.

예를 들면, 그림 15.1은 키 A S E A R C H I N를 포함한 2-3-4 트리를 나타낸 것이다. 그런 트리에서 어떻게 검색되는 가를 보는 것은 쉽다. 예를 들면, 그림 15.1의 트리에서 G를 검색하기 위해서, G는 E와 R사이에 존재하기 때문에 근에서부터 중간 연결을 따라가서 그뒤에 H, I와 N를 포함한 노드들로 부터 왼쪽 연결로 비성공적인 종료로 한다.

2-3-4트리에 새로운 노드를 삽입시키기 위해서, 이전처럼 비성공적인 검색을 수행하고, 노드를 고정시키는 것이다. 성공적으로 종료되는 노드가 2-노드인 경우에 무엇을 해야하는지를 보는 것은 쉽다. 즉, 3-노드로 변경시키는 것이다. 예를 들면, X는 S를 포함한 노드에 추가하므로써(그리고 다른 연결) 그림 15.1의 트리에 첨가된다. 비슷하게, 3-노드는 4-노드로 쉽게 변경된다. 그러나 4-노드에 새로운 노드를 삽입시킬 필요가 있는 경우에 무엇을 해야 하는가? 예를 들면, 그림 15.1의 트리로 G를 삽입하는 경우에 이것은 어떻게 수행되는가? 한 가지 가능성은 H, I와 N를 포함한 4-노드의 새로운 가장 왼쪽의 자식으로서 고정시키는 것이

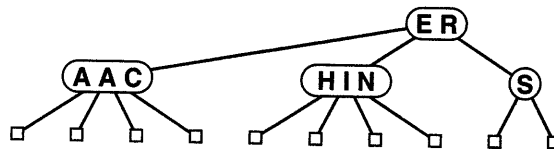


그림 15.1 2-3-4 트리들

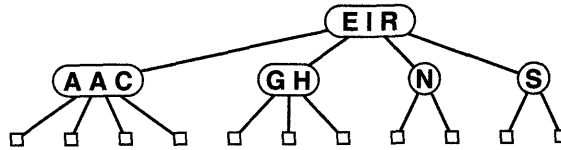


그림 15.2 2-3-4 트리에(G의) 삽입

나 그러나 더 좋은 해는 그림 15.2에 제시되어있다. 즉, 먼저 4-노드를 두 개의 2-노드로 분리시키고 그것의 부모에 이르기까지 키들중의 하나를 전달한다. 첫째 H, I와 N를 포함한 4-노드는 두 개의 2-노드로 분리시키고(하나는 H를 포함하고, 다른하나는 N를 포함한다) “중간 키” I는 4-노드로 변화시키면서 E와 R이 포함된 3-노드에 이르기까지 전달된다. 그때 H를 포함한 2-노드에서 G에 대한 여분이 존재한다.

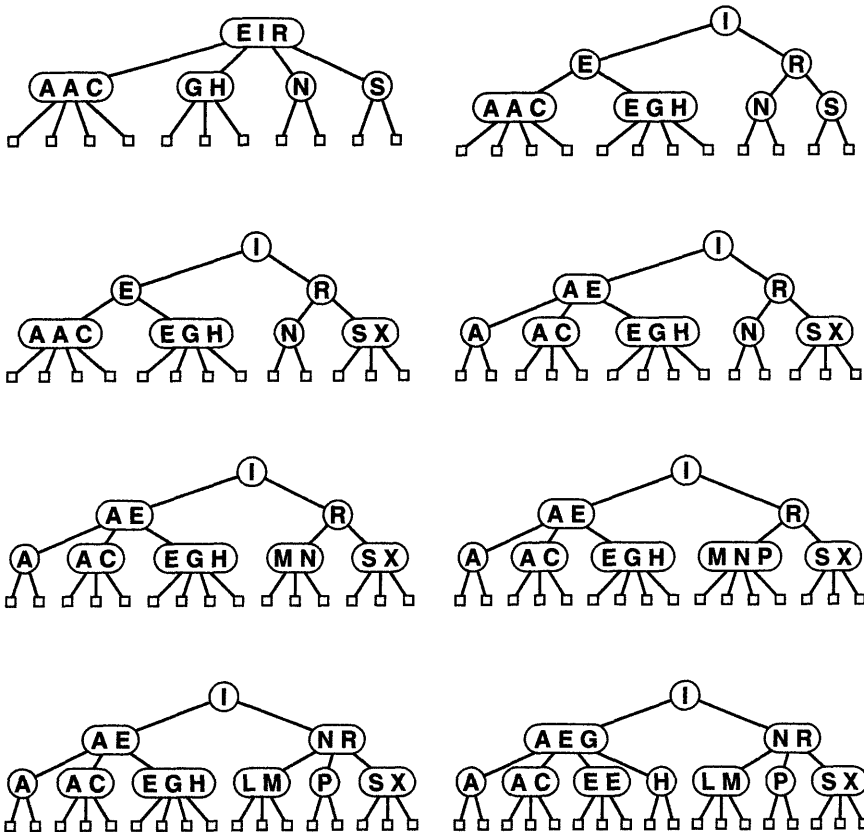


그림 15.3 2-3-4 트리를 생성

그러나 그것의 부모가 또한 4-노드인 경우에 4-노드로 분리시키는 경우는 어떻게 되는가? 한 가지 방법은 부모를 분리시키는 것이나, 조부모 또한 4-노드이므로 그것의 부모가 될 수가 있는 등이다. 즉, 트리를 따라 위로 올라가는 방법 모두로 노드들을 분리시키는 것을 유지한다. 더 쉬운 방법은 어떤 노드의 부모가 트리 아래로 진행되는 방법에서 어떤 4-노드를 분리시키므로써 4-노드가 되지는 않는다. 그림 15.3은 키 A S E A R C H I N G E X A M P L E의 완전한 집합에 대해 2-3-4 트리의 구성을 완료한 것이다. 첫 번째 라인에서 근 노드는 두 번째 E의 삽입동안에 분리되고, 다른 분리는 두 번째 A와 L 그리고 세 번째 E가 삽입될 때 발생된다.

위의 예제는 새로운 노드들을 검색하고 트리밑으로 가는 방법에서 4-노드를 분리시키므로써 2-3-4 트리들로 쉽게 삽입된다. 특별히, 그림 15.4에 제시된 것과 같이, 두 개 2-노드들에 연결된 3-노드로 변형을 시키고, 3-노드를 만날때마다 4-노드에 연결을 시키고 그것을 두 개의 2-노드와 연결된 4-노드로 변형을 한다.

이같은 “분리” 연산은 키들 뿐만 아니라 포인터들이 이동되는 방법 때문에 처리된다. 두 개의 2-노드들은 4-노드로서 포인터의 같은 수(4개)를 지니므로써 분리는 분리 노드 아래에 어떤 것을 변경시킴이 없이도 수행된다. 그리고 3-노드는 다른 키를 추가시켜서만 4-노드로 변경되지 않는다. 즉, 다른 포인터가 또한 필요하다.(이 경우에서, 특별한 포인터가 분리에서 제공된다) 중요한 점은 이같은 변형이 순수하게 “국지적”인 것이다. 즉, 트리의 어느부분도 조사되거나 그림 15.4에 제시된 것 이외에 변형이 없다. 변형의 각각은 4-노드들에서 트리의 부모에 이르는 키들 중의 하나를 전달하고 그것에 따라서 연결을 재 구성한다.

변형은 트리에서 각 노드를 통해 전달되므로써 4-노드가 아닌 노드를 생성하도록 하는 것이기 때문에 4-노드가 되는 부모에 관해서 걱정할 필요는 없다. 특히, 트리의 밑에서 나오게

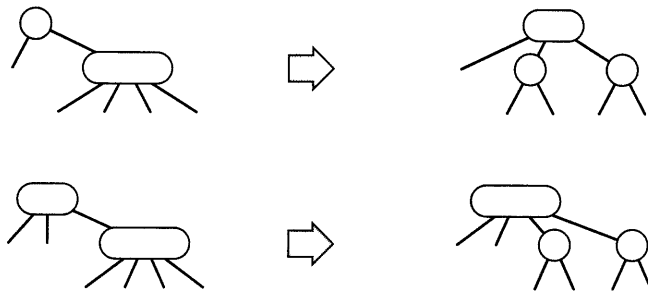


그림 15.4 4-노드를 분리

될 때, 4-노드상에는 없고, 2-노드를 3-노드에 혹은 3-노드를 4-노드로 변형시키므로써 새로운 노드를 직접적으로 삽입할 수가 있다. 실제적으로 삽입되어질 새로운 키를 전달하는 상상적인 4-노드의 분리로서 제일 밑에서 삽입을 취급하는 것이 편리하다.

한 가지 상세한 내용은 다음과 같다. 트리의 근이 4-노드가 될 때마다, 위의 예제에서 분리된 첫 번째 노드에 대해 수행한 것과 같이 세 개의 2-노드로 분리시킨다. 이것은 근의 부모에 관해서 걱정할 필요가 없기 때문에 분리를 수행하는 다음 삽입까지 기다리는 제안보다 더 간단한 것으로 판명된다. 근을 분리하는 것(그리고 단지 이같은 연산)은 트리를 한 레벨 “더 높게” 만드는 것이다.

위에서 스케치한 알고리즘은 2-3-4 트리에서 검색과 삽입을 수행하는 방법이다. 즉, 4-노드들은 위에서 아래로 수행하는 방법과 분리가 되므로 트리들을 하향식 2-3-4 트리라고 부른다. 흥미로운 것은 비록 균형에 관해서 전혀 걱정은 없지만 결과적인 트리는 완전히 균형적으로 된 것이다.

성질 15.1 N -노드 2-3-4 트리들에서 검색은 $\lg N + 1$ 노드들 이상으로 방문되지 않는다.

근에서 모든 외부 노드에 이르는 거리는 같다. 즉, 수행하는 변형은 근을 분리할때를 제외하고 어떤 노드에서 근에 이르는 거리상에 효과가 없다. 이 경우에서 모든 노드들에서 근에 이르는 거리는 하나씩 증가된다. 만약 노드들 모두가 2-노드이면, 제시된 결과는 트리가 전이진 트리와 같게 된다. 즉, 거기에 3-노드와 4-노드를 지니면, 높이는 더 낮게 된다. □

성질 15.2 N -노드 2-3-4 트리로의 삽입은 최악의 경우 $\lg N + 1$ 노드들이 분리되는 것보다 더 적게 되고, 평균의 경우에는 분리된 노드보다 적은 것이 된다.

발생되는 최악의 것은 삽입점에 대한 경로에서 모든 노드들이 4-노드들이고 그것들 모두는 분리된다. 그러나 N 개의 요소로 부터 무작위 순열으로 생성된 트리에서, 최악의 경우에서는 발생이 되지 않을 뿐만 아니라 몇 개 안되는 분할은 많은 4-노드들이 없기 때문에 평균의 경우에서 요구된다. 그림 15.5는 95개의 요소들의 무작위 순열에서 생성된 트리를 나타낸 것이다. 즉, 거기에는 9개의 4-노드들이 있고 그것들 중의 하나는 제일 밑의 레벨에 있지 않는다. 2-3-4 트리들의 평균적인 활용도에서 해석적인 결과는 지금까지 전문가들에게 배제가 되었으나, 실험적인 연구는 몇안되는 분리에서 수행되는 것을 계속적으로 보여준다. □

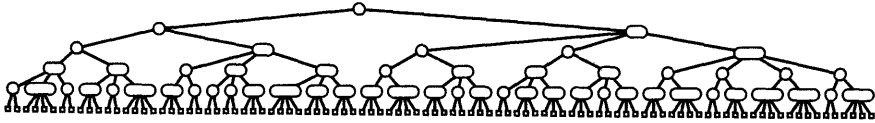


그림 15.5 큰 2-3-4 트리

위에 주어진 기술은 좋은 최악의 경우 활용도를 지닌 2-3-4 트리를 이용해서 검색에 대한 알고리즘을 정의하는 것이 충분하다. 그러나 실제적인 구현에 대해서는 단지 반정도의 방법이 있다. 2-, 3-와 4-노드를 나타내는 명확한 자료 형태들상에 실제로 변형을 수행하는 알고리즘을 기술하는 것이 가능한 반면에, 수행될 필요가 있는 대부분의 것은 이같은 직접적인 구현에서는 매우 불편한 것이다.(두 개 노드 변형중의 더 간단한 것을 구현함으로써 확신을 할 수 있다) 더구나, 더 복잡한 노드 구조들을 처리하는데 발생이 되는 불필요한 연산은 표준 이진 트리 검색보다 더 느린 알고리즘으로 수행이 된 것이다. 균형화의 주된 목적은 나쁜 최악의 경우에 대한 “보충”을 제공하는 것이나 알고리즘의 수행때마다 그 보충에 대한 불필요한 연산의 비용을 지불하는 것은 불행한 것이다. 다행히도 아래에 제시된 것과 같이, 표준적인 이진 트리 검색에 의해 발생되는 비용을 넘어서 아주 적은 양의 불필요한 연산들의 유일한 방법으로서 수행되도록 변형을 허용한다.

적색-흑색(Red-Black) 트리

현저하게도 노드당 단지 한 비트를 이용해서 표준 이진 트리(2-노드들로만) 2-3-4 트리를 표현 하는 것이 가능하다. 개념은 “적색” 연결들에 의해서 적은 이진 트리들 경계로서 3-노드들과 4-노드들을 표현하는 것이다. 표현은 더 간단하다. 즉, 그림 15.6에 제시된 것과 같이 4-노드들은 적색 연결에 의해서 연결된 2개의 2-노드들로서 표현되고 3-노드들은 적색 연결에 의해서(적색 연결은 굵은 선으로 그려진다) 연결된 두 개의 2-노드들로서 표현된다. (어느 중심에서나 3-노드에 대해서는 합당하다)

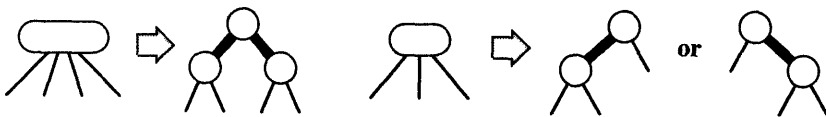


그림 15.6 3-노드들과 4-노드들의 적색-흑색 표현

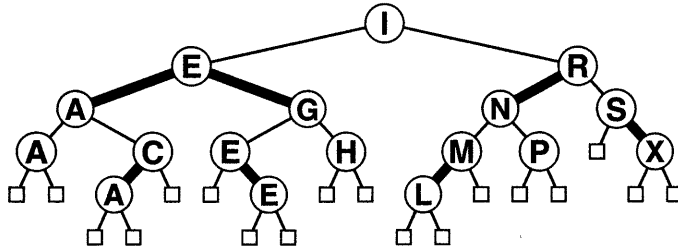


그림 15.7 적색-흑색 트리

그림 15.7은 그림 15.3에서의 마지막 트리를 나타내는 한 방법이다. 만약 그것들이 연결하는 노드들을 함께 합치고 적색 연결을 제거시키는 경우, 결과는 그림 15.3에서와 같이 2-3-4 트리가 된다. 노드당 특별한 비트는 그 노드를 가르키는 연결의 색깔을 저장시키는데 이용된다. 즉, 이같은 방법에서 표현된 2-3-4 트리를 적색-흑색 트리로서 언급한다.

각 3-노드의 “경사(slant)”는 아래 기술이 된 알고리즘의 동적인 것에 의해서 결정된다. 각 2-3-4 트리에 대응되는 많은 적색-흑색 트리가 있다. 3-노드들 모두를 같은 방법으로 경사를 지게 하는 규칙은 가능하나 그렇게 해야할 이유가 없다.

이같은 트리들은 그것들이 정의되는 방법으로 되는 많은 성질들을 지닌다. 예를 들면, 근에서 외부 노드에 이르는 어떤 경로를 따라서 행에서 두 개의 적색 연결들이 결코 존재하지 않는 것이다. 한 개 경로가(흑색-적색으로 번갈아 가면서) 다른 것들(모두 흑색)인 것의 두배 정도인 것은 가능하나 모든 경로 길이는 $\log N$ 에 비례된다.

그림 15.7의 놀라만한 특징은 이중 키들의 위치이다. 어떤 균형 트리 알고리즘은 그 노드의 양쪽 측면에 나타내도록 하기 위해 주어진 노드에 대해 똑같은 키들로서 레코드들에게 허용한다. 이것은 표준 이진 검색 트리에 대해서와 같이 검색 프로시저를 계속하트서 주어진 키로 된 모든 노드들을 찾을 수가 없다. 대신에, 14장의 treeprint 프로시저와 같은 프로시저는 이용되어야만 하거나 혹은 이중 키들은 14장에서 논의된다.

적색-흑색 트리들의 매우 훌륭한 성질은 표준 이진 트리 검색에 대해 search 프로시저가 변형없이 작동하는 것이다.(이전 절에서 논의된 이중 키의 문제는 제외하고) 노드를 지칭하는 연결이 적색이면 1이고, 흑색이면 0인 각 노드에 한 비트의 필드 b 를 첨가시키므로써 연결 색깔을 구현한다. 즉, search 프로시저는 그 필드를 결코 단순히 조사하는 것이 아니다. 이와 같이, “불필요한 연산(overhead)”은 근본적인 검색 프로시저에 의해서 취해진 시간에 균형화된 메카니즘을 첨가시키는 것은 없다. 각 키는 단지 한 번 삽입되나 전형적인 응용에서

여러번 검색되므로써, 결과는 상대적으로 적은 비용으로(균형화에 대한 작업이 검색동안에 이루어지지 않기 때문) 검색 시간을 개선할 수가 있다.(트리들이 균형화가 되었기 때문이다)

더구나 삽입에 대한 불필요한 연산은 매우 적다. 즉, 4-노드들을 볼때만 어떤 것들을 다르게 수행을 하고 그것을 항상 위로 쪼개기 때문에 트리에서 많은 4-노드들을 지니지 않는다. 내부 반복은 삽입 프로시저의 다음 구현에서 제시된 것과 같이 단지 한 개의 특별한 테스트를 필요로 한다:(만약 한 노드가 두 개의 적색 자식을 지니면, 4-노드의 일부분이 된다)

```
void Dict::insert(itemType v, infoType info)
{
    x = head; p = head; g = head;
    while ( x != z )
    {
        gg = g; g = p; p = x;
        x = ( v < x->key ) ? x->l : x->r;
        if ( x->l->b && x->r->b ) split (v);
    }
    x = new node ( v, info, 1, z, z );
    if ( v < p->key ) p->l = x; else p->r = x;
    split(v); head->r->b = black;
}
```

명확하게 하기 위해, 이것과 연속적인 코드에서 상수들 red = 1 이고 black = 0으로 이용한다. 즉, 간단히 red를 언급함이 없이 0이 아닌것에 대한 테스트로서 흑색이 아님을 테스트한다. 이 프로그램에서, x는 이전 처럼 트리의 아래로 이동을 하고 gg, g와 p는 트리에서 x의 조조부모, 조부모와 부모를 가르키도록 한다. 이같은 모든 연결들이 필요한 이유를 보기

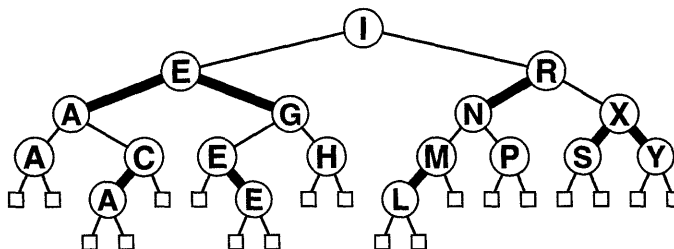


그림 15.8 적색-흑색 트리로(Y를 삽입)

위해서, 그림 15.7에서 트리에 Y를 첨가시키는 것을 고려해 보자. S와 X가 포함된 3-노드의 오른쪽에서 외부 노드에 도달할 때, gg는 R, g는 S이고 p는 X가 된다. 지금 Y는 그림 15.8에 제시된 트리의 결과로 S, X와 Y를 포함한 4-노드로 되도록 첨가된다.

R의 오른쪽 연결이 S가 아니고 X를 지칭하는 것으로 변경되기 때문에 R(gg)에 대한 포인터가 필요하다. 이것을 어떻게 나타내는 가를 정확히 보기 위해서, split 프로시저의 연산을 볼 필요가 있다. 수행할 두 개의 변형에 대한 적색-흑색 표현을 고려해 보자. 즉, 만약 4-노드에 연결된 2-노드를 지니면, 그때 그것을 두 개의 2-노드들에 연결된 3-노드로 변경시킨다. 새로운 노드가 밑에 추가가 되는 경우, 상상의 4-노드에 중간 노드로 된다.(즉, 비록 이것은 명백히 테스트되지는 않지만 z를 red로 간주를 한다)

4-노드에 연결이 된 2-노드를 만날 때 요구되는 변형은 쉽고, 같은 변형이 그림 15.9에 제시된 것과 같이 “오른쪽” 방법으로 4-노드에 연결된 3-노드를 지나는 경우로 처리된다. 이와 같이 split는 x를 적색인 것으로 하고 x의 자식을 흑색인 것으로 하므로써 시작된다.

이것은 그림 15.10에 제시된 것과 같이 4-노드에 연결된 3-노드를 만나는 경우에 발생하는 두 가지 다른 상황이 남겨진다.(실제적으로, 이같은 두 개의 비치는 상들은 다른 중심에 대해 3-노드에 대해 발생되므로써 4가지 상황이 존재한다) 이같은 경우에서, 4-노드를 분리하는 것은 행으로 두 개의 적색 연결들과 정정이 되어야하는 불법적인 상황이 남겨진다. 이것은 코드에서 쉽게 테스트를 한다. x를 적색으로만 표시를 하고, x의 부모 P가 또한 적색이면, 더 이상의 행동을 취한다. 상황은 적색 연결들에 의해서 연결된 세 가지 노드들을 지니기 때문에 너무 나쁘지 않다. 즉, 수행할 필요가 있는 모든 것은 트리를 변형시키는 것이고 그래서 적색 연결들은 같은 노드에서 아래로 지적된다.

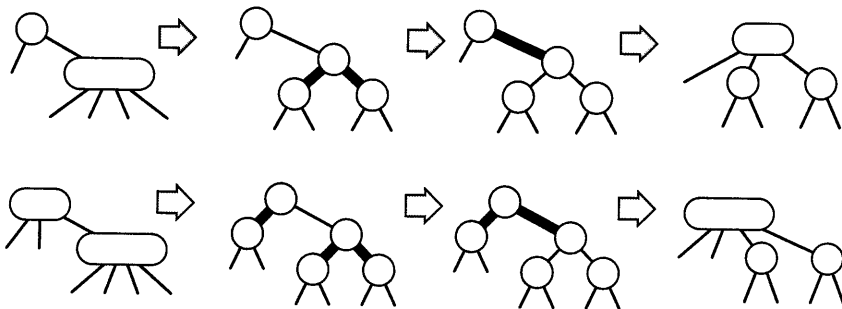


그림 15.9 색깔 플립으로서 4-노드들을 분리

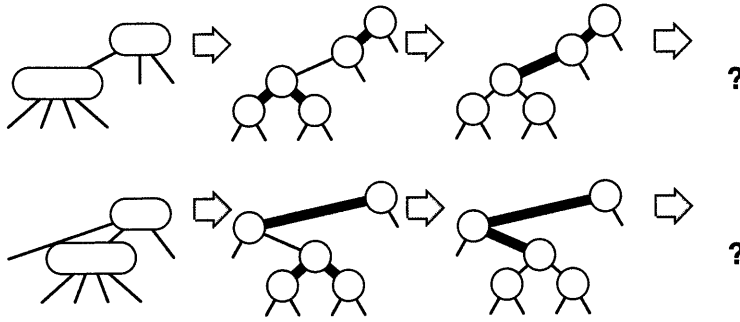


그림 15.10 색깔 플립으로서 4-노드들을 분리: 회전이 필요

다행히도 요구된 효과를 이루는 간단한 연산이 존재한다. 적색 연결이같은 방법으로 중심 되는 두 개중의 더 쉬운 것인 그림 15.10의 첫 번째(제일 위의) 경우로서 시작된다. 문제는 3-노드가 틀린 방법으로 중심을 이루는 것이다. 따라서, 3-노드의 중심을 변경하도록 트리를 재구성하고, 이 경우는 x의 색깔 플립과 그것의 자식은 그림 15.9의 두 번째 경우처럼 충분히 줄인다. 3-노드를 다시 중심으로 하는 트리를 재구성하는 것은 그림 15.11에 제시된 것과 같이 세 개의 연결들을 변경시키는 것이 포함된다. 즉, 그림 15.11은 그림 15.8과 회전되어질 N과 R을 포함한 3-노드와 같은 것이다. R의 왼쪽 연결은 P를 지칭하도록 변경을 하고, N의 오른쪽 연결은 R을 지칭하도록 변경을 하고 그리고 I의 오른쪽 연결은 N를 지칭하도록 하기 위해서 변경된다. 또한 두 개 노드들의 색깔들이 바뀌어지는 것을 조심해야한다.

이같은 단일 회전(single rotation) 연산은 어떤 이진 검색 트리상에 정의되고,(색깔을 포함한 연산들을 무시하는 경우) 여러 가지 균형 트리 알고리즘에 대한 기초가 된다. 이것은 검색 트리의 본질적인 성격을 유지하고, 단지 세 가지 연결로 변경되는 것을 포함한 국지적 변경이 되기 때문이다. 그러나 단일 회전은 트리의 균형을 반드시 개선하는 것이 아니라는 것

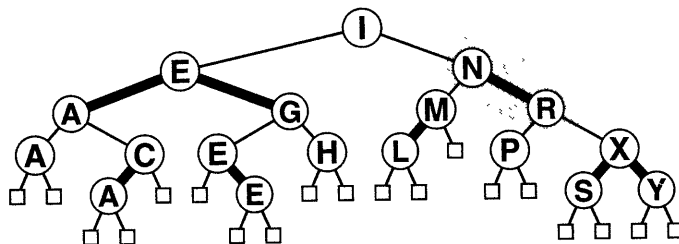


그림 15.11 그림 15.8에서 3-노드를 회전

을 주시하는 것이 중요하다. 그림 15.11에서, 회전은 근에 한 단계 더 가까운 N의 왼쪽으로 모든 노드들을 가져가나, R의 오른쪽에 있는 노드들은 한 단계 더 낮게된다. 즉, 이같은 경우에서 회전은 트리를 더 짧게하지는 않지만 더 적은 균형화된 것으로 한다. 하향식 2-3-4트리들은 균형을 개선하는 것인 단일 회전을 인식하는데는 편리한 방법으로서 간단히 보게 된다.

단일 회전을 수행하는 것은 트리의 구조 즉, 주의를 가지고 수행을 하는 것을 변경시키는 것이다. 14장에서 삭제 알고리즘을 고려할 때 보는 바와 같이, 코드는 왼쪽-오른쪽 대칭으로서 비슷한 경우의 수를 지니기 때문에 필요이상으로 더 복잡하다. 예를 들면, 그림 15.8에서 연결들 y , c 와 gc 는 I , R 과 N 를 각각 지칭한다고 가정하자. 그때 그림 15.11에 대한 변형은 연결 변경 $c \rightarrow l = gc \rightarrow r$; $gc \rightarrow r = c$; $y \rightarrow r = gc$ 로서 효과를 낸다. 거기에는 세 가지 다른 경우가 있다. 즉, 3-노드는 다른 방법을 중심으로 하거나 혹은 y 의 왼쪽 측면상에 존재하는 것이다.(어느 방법을 중심으로) 이같은 4가지 다른 경우를 처리하는 편리한 방법은 노드 y 의 손자(gc)와 관련된 자식(c)을 “다시 발견하기” 위해서 검색 키 v 를 이용한다.(검색이 그것의 제일 밑에 있는 노드에 이르면 단지 3-노드를 재 중심으로 하게 된다) 이것은 c 와 gc 에 대응되는 두 개의 연결들 뿐만 아니라 그것들이 오른쪽이거나 왼쪽 연결인지를 검색 동안에 기억하는 대안 보다 다소 더 간단한 코드로 된다. 부모가 y 인 v 에 대한 검색 경로를 따라서 3-노드를 재 중심으로하는 것에 대한 다음 함수를 볼 수가 있다.

```
struct node *rotate(itemType v, struct node *y)
{
    struct node *c, *gc;
    c = ( v < y->key ) ? y->l : y->r;
    if ( v < c->key )
        { gc = c->l; c->l = gc->r; gc->r = c; }
    else
        { gc = c->r; c->r = gc->l; gc->l = c; }
    if ( v < y->key ) y->l = gc; else y->r = gc;
    return gc;
}
```

만약 y 가 근을 지칭하면, c 는 y 의 오른쪽 연결이고 gc 는 c 의 왼쪽 연결이 되는 경우, 이것은 그림 15.8에서 그림 15.11에 있는 트리를 생성하기 위해서 필요한 연결 변형을 정확히 만든다. 여러분은 다른 경우들을 체크하기를 바랄 것이다. 이같은 기능은 3-노드의 제일 위로 연결을 되돌리나, 색깔 변경 그 자체는 수행이 되지 않는다.

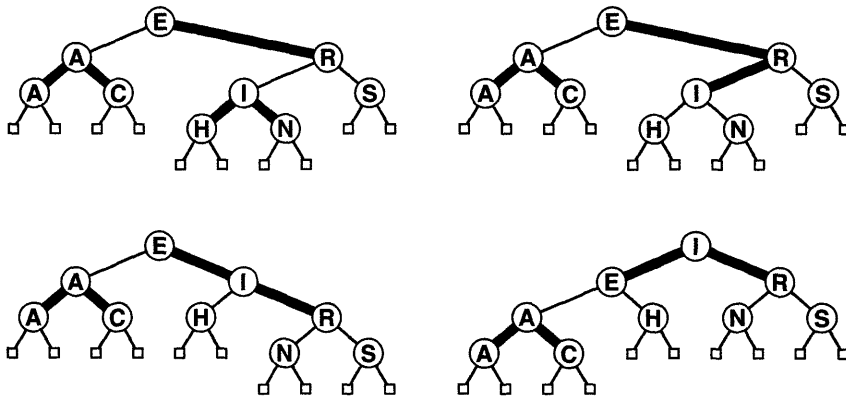


그림 15.12 적색-흑색 트리에서 노드를 분리

이와 같이, split(그림 15.10 참조)에 대한 세 번째 경우를 처리하기 위해서, x 를 흑색으로 한다. 이같은 것은 g 와 p 로서 지정한 두 개의 노드들로 구성된 3-노드를 다시 중심으로 하고, 3-노드가 올바른 방법으로 중심될 때 두 번째의 경우와 같다.

마지막으로, 두 개의 적색 연결들이 다른 방향으로 중심될 때(그림 15.10참조)의 경우를 처리하기 위해서, 단순히 p 를 $rotate(v, g)$ 에 세트 시킨다. 이같은 노드들은 같은 색깔을 지니므로 해서 색깔 변경은 필요가 없고 그리고 세 번째 경우로 줄어들게 된다. 이것과 세 번째 경우에 대한 회전을 비교하는 것을 이중 회전(double rotation)이라고 부른다.

그림 15.12는 G 가 첨가 될 때 예제에서 발생하는 split를 제시한다. 첫째, H , I 와 N 를 포함한 4-노드를 분리시키기 위해서 색깔 플립이 존재한다. 다음으로 이중 회전이 필요하다. 즉, I 와 R 사이에 가장자리 주위에 첫 번째 부분과 E 와 I 사이의 가장자리에 두 번째 부분이다. 이같은 변형뒤에, G 는 그림 15.13에 제시된 것과 같이 H 의 왼쪽에 삽입된다. 만약 근이 4-노드인 경우,(그림 15.13에서 첫 번째 트리로 삽입) 그때 split 프로시저는 근을 적색으로 만든다. 즉, 이것은 그것 위에 있는 가상 노드를 따라서 3-노드로 변형시키는 것에 대응된다. 이것을 수행하는 이유는 없다. 그래서 문장은 근을 흑색으로 유지시키기 위해 삽입의 끝에 포함된다.

이것은 split에 의해서 수행된 연산의 기술을 완료한다. x 와 자식들의 색깔을 변경하고 필요한 경우 이중 회전에서 밑의 부분을 수행하고, 다음과 같이 필요한 경우 단일 회전을 수행한다.

```

void split(itemType v)
{
    x->b = red; x->l->b = black; x->r->b = black;
    if ( p->b )
    {
        g->b = red;
        if ( v < g->key != v < p->key ) p = rotate(v, g);
        x = rotate(v, gg);
        x->b = black;
    }
}

```

이같은 프로시저는 회전 뒤에 색깔을 고정시키고, 검색은 모든 연결 변경들로 인해서 잃어버리지 않도록 트리에서 충분한 높이로 x를 다시 시작한다.

split와 rotate 코드는 insert에서 변수 gg 등등을 이용해서 분리된 프로시저로서 포함된다. 어떤 언어에서는 insert내에서 이같은 프로시저를 내부에 존재하도록 함으로써 처리가 된다. 즉, C++에서 여러 가지 덜 매력적인 대안들이 그것들을 광의적으로 하므로서 friend들에서 Dict에 이르는 것으로 선언하는 범주에 이르기까지 이용이 가능하다.

적색-흑색 트리들에 대한 class 선언은 초기화시키기 위해 node 구성자에서 인수를 포함하면서 node에서 이진 플래그 v 필드의 부가로 보통의 이진 검색에 대해서는 이전 장에서 주어진 것과 같다. 만약 공간이 자유롭게 이용이 가능하면, b를 정수로서 선언하나 그러나 아마도 정수형 key의 부호 비트인 것 혹은 infor에 의해서 언급된 레코드에서 어디에서나 단지 한 비트를 이용하도록 하는 것이 정상적이다. 그때 Dict 구성자에서 가상 노드들은 다음과 같이 조심스럽게 초기화된다.

```

Dict(int max)
{
    z = new node ( 0, infoNIL, black, 0, 0);
    z->l = z; z->r = z;
    head = new node(itemMIN, 0, black, 0, z);
}

```

z의 연결들은 z를 가르키도록 한다. 평상시와 같이, 반드시 요구되지는 않지만 부호에서 그런 세팅들은 코딩을 간단하게 한다. 예를 들면, 이같은 세팅은 insert에서 while 반복에서 break를 이용하는 것을 피하도록 한다.

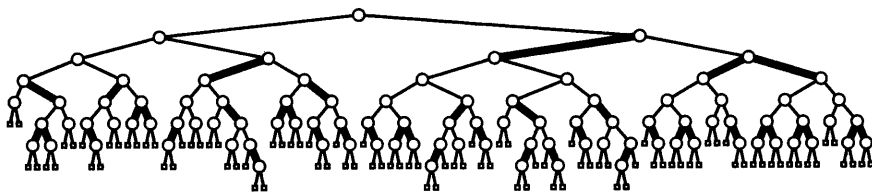


그림 15.14 큰 적색-흑색 트리

이같은 알고리즘에서 평균의 경우 분석은 아직 처리를 해야하나, 부분적인 분석들과 모의 실험에서 결과를 야기시킬 수가 있다. 그림 15.14는 이용된 더 큰 예제에서 생성된 트리를 제시한다. 즉, 트리에서 무작위 키들을 검색 하는 동안에 방문되어지는 노드들의 평균 수는 약 5.81로 14장에서 같은 키들로 생성된 트리에 대해서는 7.00과 비교되고, 완전히 균형 트리에 대해서는 가장 좋은 경우인 5.74와 비교된다. □

그러나 적색-흑색 트리의 실제적인 의미는 최악의 활용도이고, 이같은 활용도가 매우 적은 비용으로서 이루어진다는 사실이다. 그림 15.15는 1에서 95에 이르는 수들이 초기에 비어있는 트리로 순서적으로 삽입되는 경우에 생성된 트리를 제시한 것이다. 즉, 이같은 트리는 매우 잘 균형화된 것이다. 노드당 검색 비용은 균형 트리가 기초적인 알고리즘에 의해서 구성되는 경우처럼 낮은 경우이고 삽입은 단지 한 개의 특별한 비트의 테스트와 가끔 split를 수반한다.

성질 15.4 N 개 노드로된 적색-흑색 트리의 검색은 $2 \lg N + 2$ 번의 비교보다 적은 것이고 삽입은 비교와 같은 회전의 1/4보다 적은 것이다.

2-3-4 트리에서 4-노드에 연결되어 3-노드에 대응되는 “분리”는 적색-흑색 트리에 대응되는데서 회전을 요구하므로 해서, 이같은 성질은 성질 15.2에서 따른다. 최악의 경우는 삽입점에 경로가 3-노드와 4-노드를 번갈아 가면서 구성될 때 일어난다. □

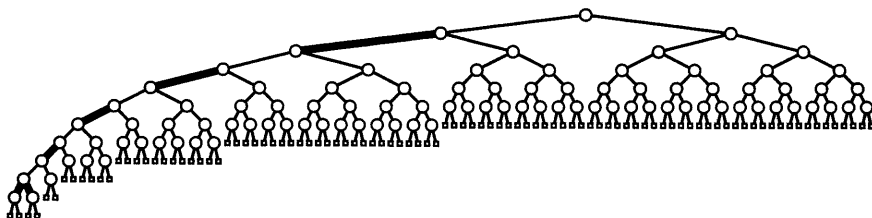


그림 15.15 퇴보되는 경우에 대한 적색-흑색 트리

요약하면, 이같은 방법을 이용해서, 5십만개의 레코드 파일에서 키는 약 20개의 다른 키들에 비교하므로써 찾게 된다. 나쁜 경우에서, 기교는 두 배정도가 필요하나, 그이상은 아니다. 더구나 매우 적은 불필요한 연산이 각 비교에서 관련이 되고, 매우 빠른 검색이 보증된다.

다른 알고리즘

이전 절에서 주어진 적색-흑색 구조를 이용하여 “하향식 2-3-4 트리”는 균형 이진 트리를 구현하는데 제안된 여러 가지 비슷한 방법들중의 하나이다. 위에서 보듯이, 트리를 균형있게 하기 위해서는 실제로 “회전” 연산들이 된다. 즉, 회전을 할 때 결정하기 쉽도록 하기 위해서는 트리의 특별한 견해를 보는 것이다. 트리의 다른 견해는 여기에서 언급이 된 몇 가지 다른 알고리즘들로 된다.

균형 트리에 대해 가장 오래되고 잘 알려진 자료구조는 AVL 트리이다. 이 트리는 각 노드의 두 개 부분 트리들의 높이가 기껏해야 하나만 다른 성질을 지닌다. 만약 이같은 조건이 삽입으로 인해 위배가 되면, 그것은 회전을 이용해서 원래대로 만든다. 그러나 이것은 특별한 반복이 필요하다. 즉, 기본적인 알고리즘으로 삽입이 되어지는 값에 대해 검색을 하는 것이고, 회전을 이용해서 노드의 높이를 조정하면서 막 방문된 경로를 따라서 트리의 위로 처리해 나간다. 또한 각 노드가 형제들의 높이 보다 하나 적거나, 같거나 혹은 하나 큰 것인 높이를 지님을 알 필요가 있다. 이것은 적색-흑색 구조를 이용해서 노드당 단지 한 비트로서 얻어진 방법이지만 간단한 방법으로서 해독이 되는 경우에는 두 비트를 요구한다.

두 번째 잘 알려진 균형 트리 구조는 2-3 트리로서, 2-노드와 3-노드가 허용된다. AVL 트리에서와 같이 회전을 포함한 “특별한 반복”을 이용해서 insert를 구현하는 것이 가능하나, 편리한 하향식 버전은 충분한 유연성이 없다. 그러나 실제로 상향식 2-3-4 트리를 이용하는 것이 낫다. 그리고 트리의 제일 밑을 검색하고, 그곳에 삽입하도록 하고,(밑의 노드가 4-노드인 경우) 4-노드들을 분리시키고 그림 15.10의 경우들을 처리하는데 수반되는 회전이 존재하는 곳인 부모로서 2-노드와 3-노드를 만날 때 까지 중간 노드를 부모 노드에 삽입시켜서 검색 경로를 거꾸로해서 위로 이동을 한다. 구현은 위에 주어진 하향식 방법보다 더 복잡한 것으로 된다.

18장에서 균형 트리의 가장 중요한 트리의 형태인 B-트리라 부르는 2-3-4 트리의 확장을 공부하게 된다. 이것은 큰 M에 대해 노드당 M 키들까지 허용을 하고, 매우 큰 파일들을 포함한 검색 응용에 대해서 널리 이용된다.

연습문제

1. 키 E A S Y Q U E S T I O N이 초기에 비어있는 트리에 삽입(그 순서로서)을 할 때 생성된 하향식 2-3-4 트리를 그려라.
2. 이전 질문에서 트리의 적색-흑색 표현을 그려라.
3. Z가 이장에 대한 예제 트리로 삽입될 때(Y 뒤에) split와 rotate로서 어떤 연결이 변경되는지를 정확히 기술하시오.
4. A에서 Z에 이르는 문자들이 순서적으로 삽입될 때 결과와 일반적으로 키들이 적은 것에서 큰 순으로 되어진 트리로 삽입이 될 때 발생하는 것을 기술하는 적색-흑색 트리를 그려라.
5. 이중 회전에 대해 얼마나 많은 트리 연결들이 실제로 변경되고, 주어진 구현에서 실제로 얼마나 많이 변경되는가?
6. 두 개의 무작위로 32-노드 적색-흑색 트리를 생성하고, 그것들(손으로나 프로그램으로) 그리고 그리고 같은 키들로 생성된 비 균형 이진 검색 트리와 비교를 하라.
7. 10개의 무작위 1000-노드 적색-흑색 트리들을 생성하라. 트리를 구성하기 위해서 요구되는 회전의 수와 근에서 외부 노드에 이르는 그것들 사이에 평균 거리를 계산하라. 결과에 대해 논하시오.
8. “색깔”에 대해 노드당 한 비트로서, 2-, 3-과 4-노드들을 표현한다. 만약 색깔에 대해 노드당 두 개의 비트로서 이용되면 얼마나 많이 다른 형태의 노드들이 표현되는가?
9. 회전은 3-노드가 “비균형적으로”된 방법으로서 4-노드로 만들 때 적색-흑색 트리에서 요구가 된다. 두 개의 적색 연결들로서 연결(완전히 균형화되거나 그렇지 않은 경우)이 된 어떤 세 가지 노드들로서 표현되도록 4-노드들을 허용함으로써 회전을 제거시키지 못하는 이유는?
10. 그림 15.11에 제시된 적색-흑색 트리를 구성하는 삽입의 순서를 주어라.

빈 면

16 장

해싱

이전 장에서 비교를 기본으로 한 트리 구조에서 검색하는 완전히 다른 방법이 바로 해싱(hashing)에 의해 제공된다. 즉, 키들을 산술적인 표 번지(table address)로 변형하여 수행함으로써 표에서 레코드들을 직접적으로 참조하는 방법이다. 만약 키들이 1에서 N 에 이르는 서로 다른 정수값이면, 그때 키 값에 대한 즉각적인 접근에 대한 준비로 표 위치 i 에서 키 i 로 된 레코드를 저장할 수 있다. 해싱은 키 값에 관해 그런 특별한 지식이 없을 때의 전형적인 검색 응용에 대한 방법의 일반화이다.

해싱을 이용한 검색에 있어서, 첫 번째 단계는 검색 키를 표 번지로서 변형시키는 해시 함수(hash function)를 계산하는 것이다. 개념적으로, 다른 키들은 다른 번지에 대응되나, 해시 함수는 완전하지 않고 두 개나 그 이상의 다른 키들이같은 표 번지에 대해 것을 논한다. 해싱 검색에 대한 두 번째 부분은 그런 키들을 처리하는 충돌 해결책(collision-resolution)이다. 충돌 해결책중의 하나는 연결 리스트이고, 수 많은 검색 키들은 미리 예측을 할 수가 없으므로 동적인 상황에서 적절하다. 조사될 다른 두 가지 충돌 해결책은 고정된 배열내에 저장된 레코드상에서 가장 빠른 검색 시간으로 된다.

해싱은 시간-공간 교환(trade-off)의 좋은 예제이다. 만약 기억장소에 제약이 없으면, 그때 키를 기억 장소 번지로 단순히 이용하므로써 기억장소 접근만으로 검색을 하게 된다. 만약 시간 제약이 없다면, 순차적인 검색 방법을 이용해서 기억장소의 최소의 양만 얻게 된다. 해싱은 이같은 두 개의 극단적인 것들 사이에 균형을 나타내도록 기억공간과 시간 둘다의 합리적인 양을 이용하는 방법을 제공한다. 이용가능한 기억공간의 효율적인 이용과 기억장치에 빠른 접근은 어떤 해싱 방법에서도 주된 관심사이다.

해싱은 기금껏 상당한 정도로 연구가 되어온 여러 가지 알고리즘들에서 널리 이용된다는 의미에서 전산학의 “고전적인”문제이다. 상당히 넓은 정도의 응용에 대해 해싱의 유틸리티를 유지하는 것이 상당히 실험적이고 해석적인 증거가 존재한다.

해시 함수(Hash Functions)

첫 번째 문제는 키들을 표 번지로 변형하는 해시 함수의 계산이다. 이것은 33장에서 보게 될 난수 발생기와 비슷한 성질을 지닌 산술 계산이다. 필요로 하는 것은 키들을(항상 정수형 혹은 적은 문자 스트링) 범위 $[0, M - 1]$ 에서 정수형으로 변형시키는 함수이다. 여기서 M 는 이용가능한 기억공간의 양에 적합한 레코드의 수이다. 이상적인 해시 함수는 계산이 쉽고, “무작위” 함수를 근사치로 하는 것이다. 즉, 각 입력에 대해 모든 출력은 어떤 의미에서는 똑 같은 것들이다.

이용되는 방법들이 산술적이므로서, 첫 번째 단계는 키들이 산술연산을 수행하는 숫자들로 변형시키는 것이다. 적은 키들에 대해, 숫자로서 키들의 이진 표현을 이용하도록 허용하는 경우, 어떤 프로그래밍 환경하에서는 전혀 작업이 수반되지 않는다.(10장의 논의 참조) 더 긴 키에 대해, 문자 스트링에서 비트들을 제거시키는 것과 기계어로 함께 묶는 것을 보게 된다. 그러나, 어떤 길이의 키들을 처리하는 유일한 방법은 아래에 제시가 되어있다.

첫째, 키들에 직접적으로 대응되는 큰 정수를 지닌다고 하자. 아마도 해싱에 대해 가장 공통적으로 이용되는 방법은 소수로서 M 을 선택하는 것이고, 어떤 키 k 에 대해, $h(k) = k \bmod M$ 를 계산하는 것이다. 이것은 많은 환경에서 계산하기가 쉽고, 키 값들을 잘 전달하는 간단한 방법이다.

예를 들면, 표 크기가 101이라 하고, 4개의 문자 키 A K E Y에 대한 인덱스를 계산한다고 가정하자. 만약 키가 10장에서 이용한 5 비트 코드로 해독되면,(여기서 알파벳의 i 번째 문자는 숫자 i 의 이진 표현으로 표현된다) 그때 10진수 44217은 다음의 이진수로 간주

00001010110010111001

를 한다. 여기서 $44217 \equiv 80 \pmod{101}$ 이므로 키 A K E Y는 위치 80에 “대해 논한다”. 거기에는 많은 가능성이 있는 키들과 상대적으로 적은 표 위치가 존재하므로서 많은 다른 키들은 같은 위치를 논한다.(예를 들면, 키 B A R H는 위에서 이용된 코드에서 해시 번지 80을 지닌다)

해시 표 크기 M 가 왜 소수이어야만 하는가? 이같은 질문에 대한 답은 mod 함수의 산술적인 성질에 의존한다. 본질적으로, 키들을 한 문자에 한 숫자인 기저-32인 수로서 취급을 한다. 예제 A K E Y는 알파벳에서 A가 첫 번째 문자이고, K는 11번째 문자들이므로 다음과 같이 기술이 되는 숫자 44217에 대응된다.

$$1 \cdot 32^3 + 11 \cdot 32^2 + 5 \cdot 32^1 + 25 \cdot 32^0$$

여기서 잘못된 선택으로 $M = 32$ 라고 가정하자. $k \bmod 32$ 의 값은 32의 곱셈을 추가함으로서 영향을 받지 않기 때문에, 어떤 키의 해시 함수는 단순히 마지막 문자의 값이 된다! 확실히 좋은 해시 함수는 키들의 모든 문자들을 고려하는 것이다. 그렇게 수행하는 가장 간단한 방법은 M 를 소수로 하는 것이다.

그런 가장 전형적인 상황으로 키들이 숫자가 아니고, 필요한 만큼 짧은 것이나 오히려 영숫자 스트링(가능하면 아주 긴것)일 때 이다. V E R Y L O N G K E Y와 같은 것에 대해 해시 함수를 어떻게 계산하는가? 코드에서 이것은 다음의 55 비트 스트링에 대응이 되고,

1011000101100101100101100011110111000111010110010111001

또는 숫자로

$$22 \cdot 32^{10} + 5 \cdot 32^9 + 18 \cdot 32^8 + 25 \cdot 32^7 + 12 \cdot 32^6 + 15 \cdot 32^5 + 14 \cdot 32^4 + 7 \cdot 32^3 + 11 \cdot 32^2 + 5 \cdot 32^1 + 25$$

이것은 너무 커서 대부분의 컴퓨터에서 정상적인 산술 함수에 대해 표현을 할 수가 없다. (그리고 훨씬 더 긴 키들을 처리를 할 수 있다) 그런 상황에서, 키들을 한 조각씩 변형시키므로써 위에서의 한 개와 같은 해쉬 함수를 계산할 수가 있다. mod 함수와 호너(Horner)의 방법이라고 부르는(36장 참조) 간단한 계산 기법에서의 산술적인 성질의 잇점을 취한다. 이같은 방법은 키들에 대응이 되는 숫자를 기술하는 다른 방법을 기본으로 한다. 즉, 예제에서 다음 식을 기술할 수가 있다.

$$((((((((22 \cdot 32 + 5)32 + 18)32 + 25)32 + 12)32 + 15)32 + 14)32 + 7)32 + 11)32 + 5)32 + 25$$

이것은 해시 함수를 계산하는 직접적인 산술 방법이다. 이장에서의 구현은 키가 정수형이 아니고 스트링을 이용한다. itemType는 할당과 불평등 테스트 연산을 지원하는 형인

stringkey이다.(그리고 물론 hash) 이것은 비록 다른 장에서 일치된 것에 대해, 예제의 키로서 한 개 문자 스트링을 이용하지만 해싱을 기술하는 가장 당연한 상황이다. 이것은 키로 된 비트 스트링으로 10장과 17장의 만나는 상황과 비슷하다. 즉, C++는 키들상에 수행되는 것이 어느 연산인가를 상세하게 나타내도록 한다. 해싱에 대해 키의 모든 형은 해시 함수를 필요로 한다. 스트링에 대해 호너의 방법을 기본으로 한 해시 함수는 디지트로써 문자를 취급하는 단순한 문제이다.

```
unsigned stringkey::hash(int M)
{
    int h; char *t = v;
    for( h = 0; *t; t++ )
        h = ( 64*h + *t ) % M;
    return h;
}
```

여기서 h 는 계산된 해시 값이고, 상수 64는 엄격히 말해서 알파벳 크기에 관련된 구현의 존적인 상수이다. 이같은 상수의 상세한 값은 실제적으로는 특별히 중요하지 않다. 이같은 방법의 단점은 키의 각 문자에 대해 몇 가지 산술적인 연산을 요구한다. 이것은 더 큰 조각으로 키를 처리함으로 개선된다. % 없이도, 이같은 코드는 위의 방정식에서 처럼 키에 대응이 되는 수를 계산하나 계산은 긴 키에 대해 과잉 넘침이 발생된다. 그런 %가 존재하는 것으로서, 나머지(modulus) 연산의 덧셈과 곱셈 성질 때문에 해시 함수를 상세하게 계산을 하고 과잉 넘침(overflow)은 %가 항상 M 보다 적은 값으로 되기 때문에 피하게 된다. $M = 101$ 로서 V E R Y L O N G K E Y에 대해 이같은 프로그램에 의해 계산이 된 해시 번지는 97이다.

분리된 연쇄(Separate Chaining)

위의 해시 함수는 키들을 표 번지로 바꾸는 것이다. 즉, 두 개 키들이같은 번지로 설정될 경우에 처리되는 방법을 결정할 필요가 있다. 가장 간단한 방법은 그 번지에 어느 키들이 설정되는 지에 대해 레코드의 연결 리스트를 생성하는 것은 간단하다. 같은 표 위치에 설정되는 키들이 연결 리스트로서 유지가 되므로 순서적으로 된다. 이것은 14장에서 논의된 기본적

인 리스트 검색 방법의 일반화로 된다. 여기서 논의된 단일 리스트 헤더 노드인 head로서 단일 리스트를 유지하기 보다는 오히려, 다음과 같이 초기화해서 M 리스트 헤더 노드들로서 M 리스트를 유지한다.

```
Dict::Dict(int sz)
{
    M = sz;
    z = new node; z->next = z; z->info = infoNIL;
    heads = new node*[M];
    for ( int i = 0; i < M; i++)
        { head[i] = new node; heads[i]->next = z; }
```

14장에서는 리스트 검색과 삽입 프로시저가 이용되고 변경이 되므로써 해시 함수는 단순히 head에 대한 참조를 heads[hash(v)]로 대체하므로써 리스트의 가운데를 선택한다.

예를 들면, 만약 표본 예제들이 그림 16.1에 해시 함수를 이용해서 처음에 비어있는 표로 연속적 삽입을 할 경우, 그때 그림 16.2에 제시된 리스트의 집합으로 된다. 이같은 방법을 분리된 연쇄(separate chaining)라고 부르며, 그 이유는 충돌되는 레코드들이 분리된 연결 리스트로서 함께 “연결”된다. 리스트들은 정렬된 순서로 유지가 되나, 정렬된 리스트를 유지하는 것은 리스트들이 아주 짧기 때문에 기초적인 순차 검색에 대한 응용만큼 중요하지 않다. 명백히 검색에 요구되는 시간의 양은 리스트 길이에 의존한다.(그리고 키들의 상대적 위치들)

“비 성공적인 검색”에 대해(표에 있지 않는 키로서 레코드에 대한 검색), 해시 함수는 M 리스트의 각각이 검색되는 것과 같고 순차적 리스트 검색에서와 같이 검색된 리스트는 단지 반 정도만 운행함으로서(평균적으로) 충분히 어떤 것으로 변경하게 된다. 이 예제에서 비 성공적인 검색에 대해 조사된 리스트의 평균 길이(z 는 계산하지 않고)는 $(0+4+2+2+0+4+0+2+2+1+0)/11 \approx 1.55$ 이다. 정렬된 순서로 리스트를 유지함으로써 반으로 줄일수가 있다. “성공적인 검색”에 대해(표에서 레코드중의 하나를 검색), 각 레코드는 동등 하게 찾아진다고 하

key:	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
hash:	1	8	5	1	7	3	8	9	3	7	5	2	1	2	5	1	5

그림 16.1 해시 함수($M = 11$)

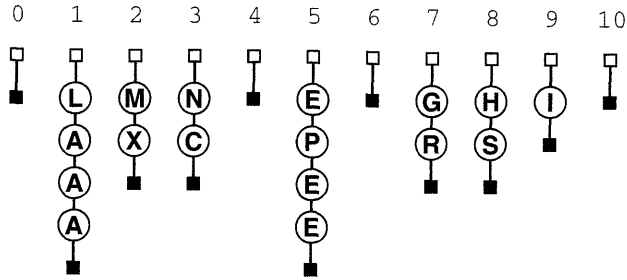


그림 16.2 분리된 인쇄

자. 즉, 키들중 7개는 조사된 첫 번째 리스트 항목에 존재하고, 6개는 조사된 두 번째 항목에 존재하는 등으로 되므로서 평균은 $(7 \cdot 1 + 6 \cdot 2 + 2 \cdot 3 + 2 \cdot 4) / 17 \approx 1.94$ 이다.(이같은 계산은 똑같은 키들의 유일한 식별자로서 혹은 어떤 다른 메카니즘으로서 구별이 되고, 검색 루틴은 각각 개별적인 키에 대해 검색을 하는데 적절하게 변경된다)

성질 16.1 분리된 연쇄는 M 연결에 대한 특별한 공간을 제외하고는 M 의 요소로서 순차적인 검색(평균적으로)에 대한 비교의 수를 줄일 수가 있다.

만약 N 즉, 표에서 키들의 수가 M 보다 훨씬 큰 경우, 그때 리스트 평균 길이의 근사치는 N/M 이고, 이것은 M 해시 값 각각이 해시 함수 설계에서 “동등하게 똑같다”. 14장에서 처럼, 비성공적인 검색은 어떤 리스트의 끝으로 가고, 성공적인 검색은 어떤 리스트를 반정도 밑으로 가게 된다. □

위에 주어진 구현은 실제 키를 포함한 리스트 헤더에 연결 해시 표를 이용하는 것이다. M 리스트 헤더 노드를 유지하는 또 다른 대안은 그것을 제거하고, heads를 리스트에서 첫 번째 키로하는 연결 표로 만드는 것이다. 이것은 알고리즘에서 어떤 복잡한 것으로 된다. 예를 들면, 새로운 레코드를 리스트 시작에 첨가시키는 것은 리스트에서 새로운 레코드를 어느곳에서든지 추가시키는 것과는 다른 연산이 된다. 그 이유는 레코드의 필드가 아니고 연결 표에서 엔트리를 변경시키는 것이 수반되기 때문이다. 비록 대안들이 어떤 상황에서도 적은 공간을 이용하지만, M 는 리스트 헤더 노드들을 이용한 특별한 편리성으로 아마도 공정하게되는 N 에 비례함으로서 항상 충분히 적은 것이다.

분리된 연쇄 구현에서, M 는 연속적인 기억공간에서 큰 영역을 이용하지 않도록 하기 위해서 상대적으로 적은 것을 선택된다. 그러나 아마도 M 를 선택하는 것이 가장 좋고 그래서

리스트들은 그것들에 대해 가장 효율적인 방법을 순차적인 검색으로 하기에 충분히 적게 된다. 즉, “하이브리드(hybrid)” 방법(연결 리스트대신에 이진 트리를 이용한 것과 같이)은 아마도 문제가 될 수도 없다. 경험으로, M 는 표에서 존재하는 것으로 기대된 키들 수의 약 1/10이 되도록 선택을 한 결과 리스트는 약 10개 키들 각각을 포함하는 것으로 된다. 분리된 연쇄 장점중의 하나는 이같은 판단이 중요하지 않다는 것이다. 즉, 만약 더 많은 키들이 기대한 것 이상으로 도착이 되면, 그때 검색은 더 길게 되어지나 표에서 키들이 적으면, 아마도 특별한 공간이 조금은 이용이 된다. 만약 기억공간이 실제로 중요한 자원이면, 제공이 되는 것만큼의 M 로서 선택을 하는 것은 활용도에서 M 개선의 요소로서 된다.

선형 조사(Linear Probing)

만약 해시 표에 놓여질 요소의 수가 미리 추정되고 그리고 연속적인 충분한 기억 공간이 부족한 방으로서 모든 키들을 유지하기 위해 이용 가능하면, 그때 해시 표에서 어떤 연결을 이용하는 것은 가치가 없을 것이다. 충돌 해결로서 도움이 되는 표에서 비어있는 장소에 의존을 하면서, 표의 크기 $M > N$ 인 N 개의 레코드를 저장하는데 여러 가지 방법들이 제안되었다. 그런 방법들을 열린 번지(open-addressing) 해싱 방법이라고 부른다.

가장 간단한 열린 번지 방법은 선형 조사이다. 즉, 충돌이 발생할 때(이미 존재한 표에 어떤 값을 놓으려고 할 때와 그런 키들이 검색 키와 같지가 않을 때), 그때 표에서 다음 위치를 조사(probe)한다. 즉, 검색 키하고 그곳의 레코드에 있는 키를 비교한다. 조사에서 3가지 가능한 결과를 볼 수가 있다. 만약 키가 일치되면, 그때 검색은 성공적인 것으로 종료하고, 거기에 레코드가 없다면 그때 검색은 비성공적이 된다. 그렇지 않으면, 검색 키나 비어있는 표 위치가 발견될 때까지 계속해서 다음 위치를 조사하게 된다. 만약 검색 키를 포함한 레코드가 비성공적인 검색 다음에 삽입되면, 그때 검색을 종료시키는 비어있는 표 공간으로 단순히 삽입시키면 된다. 이같은 방법은 다음과 같이 쉽게 구현된다.

```
class Dict
{
private:
    struct node
    { itemType key; infoType info;
      node() { key = " "; info = infoNIL; }
```

```
key: A S E A R C H I N G E X A M P L E
hash: 1 0 5 1 18 3 8 9 14 7 5 5 1 13 16 12 5
```

그림 16.3 해시 함수($M = 19$)

```
};
struct node *a;
int M;
public:
    Dict(int sz)
    { M = sz; a = new node[M]; }
    int search(itemType v);
    void insert(itemType v, infoType info)
    {
        int x = v.hash(M);
        while ( a[x].info != infoNIL ) x = ( x+1 ) % M;
        a[x].key = v; a[x].info = info;
    }
};
```

선형조사는 표에서 비어있는 점을 나타내기 위해 특별한 키를 요구한다. 즉, 이같은 프로그램은 그 목적에 대해 한 개 비어있는 것으로 이용을 한다. 계산 $x = (x + 1) \% M$ 가 다음 위치를 조사하는 것에 대응된다.(표의 끝에 도달될 때 시작으로 다시가게 한다) 이같은 프로그램은 표가 표의 전체를 다 채워는지에 대해 체크하지 않는다.(이 경우에 무엇이 발생하는가?) search의 구현은 insert와 비슷하다. 즉, 단순히 조건 “($v \neq a[x].key$)”를 while 반복에 추가하고 레코드를 저장하는 다음 라인을 `return a[x].info`로 대체를 한다.

$M = 19$ 인 키들의 예제 집합에 대해, 그림 16.3에 제시된 해시 값을 얻을 수가 있다. 만약 이같은 키들은 처음에 비어있는 표로 주어진 순서대로 삽입을 하면, 그림 16.4와 같이 된다. 이중 키들은 초기 조사 위치와 다음에 비어있는 표 위치 사이에 나타나, 연속적인 것은 필요하지는 않는다.

선형 조사에 대한 표 크기는 $M > N$ 이므로 분리된 연쇄에 대한 것 보다 큰 것이나, 연결이 이용되지 않기 때문에 기억공간의 전체 양은 덜 이용된다. 이 예제에 대한 성공적인 검색에 대해서 조사된 항목의 평균 수는 $33/17 \approx 1.94$ 이다.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A																		
S																		
															E			
	A	A																
																	R	
				C														
								H										
									I									
														N				
							G											
					E	E												
					E	E	G	H	I	X								
	A	A	C	A														
														M				
																P		
													L					
					E	E	G	H	I	X	E							

그림 16.4 선형 조사

성질 16.2 선형 조사는 $2/3$ 이하가 꽉 찬 해시 표에 대해 평균적으로 5개이하의 조사를 한다.

해시 표 $\alpha = N/M$ 의 “부하 요소(load factor)”에 의해서 요구된 조사의 평균수에 대한 정확한 공식은 비 성공적인 검색에 대해 $1/2 + 1/2(1 - \alpha)^2$ 이고 성공적인 검색에 대해 $1/2 + 1/2(1 - \alpha)$ 이다. 이와 같이, 만약 $\alpha = 2/3$ 을 취하면, 평균적으로 비 성공적인 검색은 5번의 조사, 평균적으로 성공적인 검색은 2번의 조사를 얻는다. 비 성공적인 검색은 항상 두 개의 것보다 훨씬 비싸다. 즉, 성공적인 검색은 표가 약 90%정도 꽉 찰때까지 5번이하의 조사를 한다. 표가 꽉차지므로해서, (α 가 1에 접근이 되므로서) 이같은 숫자들은 매우 크게 된다. 즉, 이것은 아래에서 더 많이 요구되는 것과 같이 실제적으로 발생하는 것을 허용하지 않는다. □

이중 해싱(Double Hashing)

key **A S E A R C H I N G E X A M P L E**
 hash 1: 1 0 5 1 18 3 8 9 14 7 5 5 1 13 16 12 5
 hash 2: 7 3 3 7 6 5 8 7 2 1 3 8 7 3 8 4 3

그림 16.6 이중 해시 함수(M = 19)

나 특별히 긴 키들에 대해, 두 번째 해시 함수를 계산하는 비용이 본질적으로 클러스터를 제거하기 위해서 몇 가지 조사를 보관시키는데만 검색의 비용을 두배로 증가시킨다. 특히, 훨씬 더 간단한 두 번째 해시 함수는 $h_2 = 8 - (k \bmod 8)$ 과 같은 것으로 충분하다. 이같은 함수는 k 의 마지막 세 개 비트들로서만 이용된다. 비록 현저한다고 할지라도, 효과는 실제적으로 의미가 있는 것과 같지는 않지만 큰 표에 대해서 몇 개 더 이용하는 것이 적절하게 된다.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A																		
S																		
				E														
	A						A											
																	R	
			C															
							A								H			
								I										
													N					
							G											
				E		A			E									
				E							X							
	A						A						A					
S		C		M							X			H				
		P		E							X			H				
											L							
				E		A			E			N			E			

그림 16.7 이중 해싱

姓名	张德胜	性别	男	出生日期	1985-03-15	身份证号	310101198503150001	学历	本科	专业	计算机科学与技术	工作单位	上海某某科技有限公司	职位	软件工程师	联系电话	13800138000	电子邮箱	zhangdes@163.com	备注	无不良记录
姓名	李小明	性别	男	出生日期	1990-07-22	身份证号	310102199007220002	学历	硕士	专业	金融学	工作单位	中国工商银行上海分行	职位	客户经理	联系电话	13900139000	电子邮箱	liming@icbc.com.cn	备注	无不良记录
姓名	王小红	性别	女	出生日期	1988-11-05	身份证号	310103198811050003	学历	本科	专业	会计学	工作单位	上海某某会计师事务所	职位	审计师	联系电话	13700137000	电子邮箱	wanghr@shcpa.com	备注	无不良记录
姓名	赵国强	性别	男	出生日期	1975-05-18	身份证号	310104197505180004	学历	本科	专业	法学	工作单位	上海市司法局	职位	律师	联系电话	13600136000	电子邮箱	zhaogq@sh.gov.cn	备注	无不良记录
姓名	孙丽娟	性别	女	出生日期	1992-09-10	身份证号	310105199209100005	学历	本科	专业	英语语言文学	工作单位	上海某某大学	职位	教师	联系电话	13500135000	电子邮箱	sunlj@shu.edu.cn	备注	无不良记录
姓名	周大伟	性别	男	出生日期	1980-12-01	身份证号	310106198012010006	学历	本科	专业	工商管理	工作单位	上海某某集团	职位	销售经理	联系电话	13400134000	电子邮箱	zhoudw@shgroup.com	备注	无不良记录
姓名	吴小芳	性别	女	出生日期	1983-04-25	身份证号	310107198304250007	学历	本科	专业	新闻学	工作单位	上海某某报社	职位	编辑	联系电话	13300133000	电子邮箱	wuxf@shnews.com	备注	无不良记录
姓名	郑浩然	性别	男	出生日期	1978-08-12	身份证号	310108197808120008	学历	本科	专业	物理学	工作单位	上海某某大学	职位	教授	联系电话	13200132000	电子邮箱	zhph@shu.edu.cn	备注	无不良记录
姓名	陈美玲	性别	女	出生日期	1986-06-03	身份证号	310109198606030009	学历	本科	专业	艺术设计	工作单位	上海某某设计公司	职位	设计师	联系电话	13100131000	电子邮箱	chenml@shdesign.com	备注	无不良记录
姓名	林志强	性别	男	出生日期	1972-10-28	身份证号	310110197210280010	学历	本科	专业	历史学	工作单位	上海某某大学	职位	副教授	联系电话	13000130000	电子邮箱	linzq@shu.edu.cn	备注	无不良记录

그림 16.8 더 큰 표에서 이중 해싱

예제 키들에 대해, 이같은 함수는 그림 16.6에서 제시된 해시 값을 나타낸다. 그림 16.7은 예제 키들을 이같은 값들로 된 이중 해싱을 이용해서 처음에 비어있는 표에 연속적으로 삽입을 함으로써 생성되는 표를 나타낸 것이다.

성공적인 검색에 대해 조사된 항목들의 평균 수는 이같은 예제에 대해 선형 조사에서 보다 다소 큰 것이다. 즉, $35/17 \approx 2.05$ 이다. 그러나 희박한(sparse) 표에서는 그림 16.8에 제시된 것과 같이 훨씬 적은 클러스터가 있다. 이같은 예제에 대해, 선형 조사(그림 16.5)에 대한 것과 같은 정도보다 두배인 클러스터가 존재하거나 혹은 동일한 평균 클러스터는 길이가 약 반정도로 된다.

성질 16.3 이중 해싱은 평균적으로 선형 조사보다 더 적은 조사를 한다.

“독립적인” 이중 해싱 함수로된 이중 해싱에 대해 만들어진 조사의 평균 수에 대한 실제적인 공식은 비 성공적인 검색에 대해 $1/(1-\alpha)$ 이고 성공적인 검색에 대해 $-\ln(1-\alpha)/\alpha$ 이다.(이 같은 공식은 상당한 정도의 수학적 분석이고, 큰 α 에 대해 검증은 하지 않았던 것이다) 위에 추천된 더 간단하고 계산하기 쉬운 두 번째 해시 함수는 이같은 것을 아주 잘 처리를 하는 것이 아니다. 그러나 M 에 가능한 근접된 값의 범주로 만들도록 하는데 이용된 충분한 비트들이 존재를 하면 오히려 더 근접된다. 실제로, 더 적은 표가 주어진 응용에 대해 선형조사로서 이중 해싱과 같은 검색 시간을 얻을 수가 있다. 즉, 평균 조사의 수는 표가 80%이하로 채워진 경우에 비 성공적인 검색에 대한 것이고, 표가 90%이하로 꽉 찬경우는 성공적인 검색에 대해 5이하이다. □

열린 번지 방법은 예측이 불가능한 삽입과 삭제의 수가 처리가 되어질 때의 동적인 상황에서는 불편하다. 첫째, 표는 얼마만큼 크게 하는가? 몇 가지 추정으로서 어떻게 많은 삽입될 수 있는가로 구성되나 활용도는 표들이 팍차도록 하기 위해 시작에서와 같이 가공할 정

도로 줄어든다. 이같은 문제에 공통적인 해는 빈도수가 (아주)적은 것을 기본으로 하여 더 큰 표로 모든 것을 다시 해싱하는 것이다. 둘째로, 삭제에 대해 주의점이 있다. 즉, 레코드는 선형 조사나 이중 해싱으로서 생성된 표에서 단순히 제거가 되어질 수가 없다. 이유는 표에서 나중의 삽입이 그 레코드를 지나가기 때문이고 이같은 레코드에 대한 검색들이 삭제된 레코드에 의해 남겨진 구멍에 존재를 하게 된다. 이같은 문제를 풀기 위한 방법은 검색에 대한 자리 유지자로서 서비스되는 또 다른 특별한 키를 지니는 것이나, 삽입에 대해 비어있는 위치로 인식되고 추천될 수가 있다. 표 크기나 삭제 어느 것도 분리된 연쇄로서 특별한 문제가 아니다.

통찰

위에서 논의된 방법들이 완전히 분석되고 어느 정도 상세하게 활용도를 비교하는 것이 가능하다. 위에 주어진 공식은 D. E. Knuth가 지은 정렬과 검색이라는 그의 책에서 기술된 상세한 분석에서 요약된다. 공식은 α 가 1에 근접하므로써 나쁜 활용도가 어떻게 열린 번지에 대해 감소가 되는지를 나타내고 있다. 큰 M 와 N 에 대해 약 90%정도의 확찬 표에서 선형 조사는 비성공적인 검색에 대해 약 50번의 조사를 요구하지만 이중 해싱에서는 10번의 조사를 한다. 그러나 실제로는 90%가 확 찬 것으로 얻도록 해시 표를 두는 것은 결코 없다. 적은 부하 요소에 대해, 단지 몇번의 조사만이 요구된다. 만약 적은 부하 요소가 배열이 되지 않는 경우, 해싱은 이용되지 않는다.

분리된 연쇄에 대해서 선형 조사와 이중 해싱을 비교하는 것은 더 많은 기억 공간이 열린 번지화 방법(여기에는 연결들이 없기 때문이다)들에서 이용이 가능하기 때문에 더 복잡하다. α 의 값은 키들과 연결들의 상대적인 크기를 기본으로 이것을 고려하도록 변경되어야 한다. 이것은 활용도를 기본으로 이중 해싱에 대해 분리된 연쇄를 선택하도록 정상으로 정당화시키는 것은 아님을 의미한다.

특별한 응용에 대한 가장 좋은 해싱 방법의 선택은 매우 어렵다. 그러나 가장 좋은 방법은 주어진 상황에서 드물게 필요로 하고 그리고 여러 가지 방법들이 기억 장치 자원을 심각하게 긴장시키지 않는한 비슷한 활용도를 지닌다. 일반적으로 가장 좋은 행동의 과정은 처리되어진 레코드의 수를 미리 알 수가 없을 때(그리고 좋은 기억 할당자는 이용이 가능하고), 가 공할 정도의 검색 시간을 줄이기 위한 간단한 분리된 연쇄 방법을 이용하는 것이고, 어느 크기가 예측되어진 시간을 지니는 키들의 집합을 검색하기 위해서 이중 해싱을 이용하는 것이다.

많은 다른 해싱 방법들은 어떤 특별한 상황에서 응용을 지닌 것을 개발한다. 비록 상세한 것은 볼 수가 없지만, 특별히 채택된 해싱 방법들의 본질을 제시하기 위해 간단히 두 가지 예를 보기로 하자. 이것들과 다른 방법들은 Knuth와 Gonnet의 책에 완전히 기술히 되어져 있다.

순서화 된 해싱(ordered hashing)이라 부르는 첫 번째는 열린 번지 표 내에서 순서를 추출한다. 표준 선형 조사에서, 검색 키와 같은 키로된 레코드나 비어있는 표 위치를 찾을 때 검색이 완료된다. 즉, 순서화된 해싱에서, 검색 키와 같은 것이나 큰 것인 키로된 레코드를 찾을 때 검색을 완료한다.(표는 이같은 작업을 수행하기 위해서 구성된 것이다) 이같은 방법은 성공적인 검색에 대해 그것을 근사치시키는 것으로 비 성공적인 검색에 대한 시간을 줄일 수 있는 것으로 판명된다.(이것은 분리된 연쇄에서 발생하는 같은 종류의 개선점이다) 이같은 방법은 비 성공적인 검색이 가끔 이용되는 응용에서 유용하다. 예를 들면, 텍스트 처리 시스템은 대부분의 단어들에 대해 잘 처리하는 단어들을 하이픈으로 연결을 하나 색다른 경우에 대해서는 그렇지 않는 알고리즘을 지닌다. 상황은 비 성공적으로 된 대부분의 검색에 대해 특별한 방법으로서 처리가 되어진 단어의 상대적으로 적은 예외적인 사전에서 모든 단어들을 찾아봄으로써 처리된다.

비슷하게 성공적 검색을 더 효율적으로 하도록 하기 위해서, 비 성공적인 검색 동안에 어떤 레코드를 주위로 이동시키는 방법들이 존재한다. 사실 R. P. Brent는 사전과 같은 매우 큰 표에서 가끔 성공적인 검색을 수반하는 응용에 대해 매우 유용한 방법을 주므로써 성공적인 검색에 대한 평균시간이 상수인 경우에 대한 방법을 개발하였다.

이것들은 해싱에 대해 제안된 많은 수의 알고리즘적인 개선점의 두 가지 예제만을 나타낸 것이다. 이같은 많은 개선점들은 흥미롭고 중요한 응용을 지닌다. 그러나 보통의 주의로 해서, 분리된 연쇄와 이중 해싱이 간단하고, 효율적이고 대부분의 응용에 대해 잘 받아 질 수가 있기 때문에 심각한 검색 응용에 대해 전문가를 제외하고는 고급 방법들의 성숙한 이용함을 제기한다.

해싱은 다소 더 간단하고 공간이 충분히 큰 표에 대해 이용 가능하면 매우 빠른(상수) 검색 시간을 제공을 하기 때문에, 많은 응용에 대해 이전의 두 개의 장들의 이진 구조를 선택한다. 이진 트리 구조는 동적이고(삽입의 수에 대한 이미 알고있는 정보는 필요가 없다) 최악의 경우 활용도를 제공할 수가 있고(모든 것은 가장 좋은 해싱 방법에서 조차 같은 장소로 해싱 되는 것이다) 그리고 더 넓은 범주의 연산을 지원하는(대부분 중요한 것으로 sort 함수) 잇점들을 지니고 있다. 이같은 요소들이 중요하지 않을 때, 해싱은 확실히 선택의 검색 방법이다.

연습문제

1. 좋은 난수 발생기의 이용으로서 해시 함수를 어떻게 구현 하는가를 기술하라. 해시 함수를 이용해서 난수 발생기를 구현하는 것을 이해하는가?
2. 비순서화 리스트로 된 분리된 연쇄를 이용해서 처음에 비어있는 표로 N 개의 키들을 삽입시키기 위해 최악의 경우 시간은 얼마나 걸리는가? 정렬된 리스트에 대해서도 같은 질문에 답하시요.
3. 키 E A S Y Q U E S T I O N이 선형 조사를 이용해서 크기 13의 처음에 비어있는 표로 그 순서대로 삽입을 시킬 때 결과가 되는 해시 표의 내용은 무엇인가? (k 번째 알파벳에 대해 해시 함수로서 $h_1(k) = k \bmod 13$ 을 이용하라.)
4. 키 E A S Y Q U E S T I O N이 이중 해싱을 이용해서 크기 13의 처음에 비어있는 표로 그 순서대로 삽입을 시킬 때 결과가 되는 해시 표의 내용은 무엇인가?(이전 질문에서 $h_1(k)$ 과 두 번째 해시 함수에 대해 $h_2(k) = 1 + (k \bmod 11)$ 을 이용하라.)
5. 이중 해싱이 N 개의 똑같은 키들로 구성되는 표를 생성하기 위해서 이용될 때 얼마나 많은 조사들이 수반되는가?
6. 많은 똑같은 키들이 현재 존재하는 응용에 대해 어느 해싱 방법이 이용되는가?
7. 해시 표에서 삽입되는 항목의 수를 미리 알고 있다고 가정하자. 분리된 연쇄는 이중 해싱을 어떤 조건하에서 선택을 하게 되는가?
8. 프로그래머는 자신의 이중 해싱에서 오류를 지니므로 해서 해시 함수들 가운데 하나는 같은 값(0이 아니고)으로 항상 되돌린다고 가정하자. 각 상황에서 어떤 일이 발생되는가 (첫 번째 것이 잘못될 때와 두번째 것이 잘못될 때)
9. 키 값들이 상대적으로 적은 범주로 되어진 것을 미리 아는 경우에 어떤 해시 함수가 이용되는가?

10. 선형조사로서 생성된 해시 표에서 삭제에 대한 다음 알고리즘에 대해 논하시오. 비어있는 위치를 찾기 위해서 삭제가 되어진(필요한 경우 끝에서 처음으로) 요소에서 오른쪽으로 조사를 하고 같은 해시 값으로 된 요소를 찾기 위해서 왼쪽을 조사한다. 그때 표 위치를 비어놓고서 삭제된 요소와 그 요소를 대체하라.

17 장

기수 검색

여러 가지 검색 방법들은 각 단계의 키들사이에서 비교를 사용하는 대신에 한 번에 한 비트씩 검색 키를 조사하므로써 처리된다. 기수 검색(radix searching) 방법이라고 부르는 이같은 방법은 해싱에서 이용된 키들의 변형된 버전에 반대되는 것으로서, 키들 그 자체의 비트들로서 처리된다. 기수 정렬 방법에서와 같이(10장 참조), 이같은 방법들은 검색 키들의 비트를 쉽게 접근하고 검색 키들의 값들이 잘 분배가 되어질 때 유용하다.

기수 검색 방법의 주된 장점은 균형 이진 트리의 복잡성없이도 합당한 최악의 경우 활용도를 제공하는 것이다. 즉, 유동적인 길이의 키들을 처리하는 쉬운 방법을 제공한다. 검색 구조내에서 키들중의 일부분을 저장하므로써 공간에서 몇 가지 절약을 허용한다. 그리고 이진 검색 트리와 해싱 둘다는 경쟁적인 데이터에 매우 빠른 접근을 제공한다. 단점은 편향된 데이터로서 나쁜 활용도를 지닌 트리도 감소시키고(그리고 문자들로 이루어진 데이터가 편향된다), 몇 가지 방법들은 공간에서 비 효율적인 이용을 한다. 또한, 기수 정렬로서 이같은 방법들은 컴퓨터 구조의 특별한 특징의 잇점으로 설계된다. 즉, 키들이 디지털 성질을 지니므로 해서, 어떤 고급언어에서 효율적인 구현을 하도록 하는 것은 어렵거나 불가능하다.

매우 긴 키들이 수반되는 곳인 검색 응용에 대해 아주 유용하고 중요한 방법을 축적시키면서 이전의 것에서 내재된 문제를 정정하는 각각의 것에서 연속적인 방법들을 조사케 된다. 부가적으로, 같은 원칙을 기반으로 한 “상수 시간” 검색인 10장의 “선형 시간 정렬”과 유사한 것을 보게 된다.

디지털 검색 트리(Digital Search Trees)

가장 간단한 기수 검색 방법은 디지털 트리 검색이다. 즉, 알고리즘은 키들사이의 비교 결과에 대한 것이 아니고 키들의 비트로 트리에서 분기되는 것을 제외하고는 이진 트리 검색과 아주 같다. 첫 번째 레벨에서 제일 앞에 있는 비트가 이용되고, 두 번째 레벨에서는 앞에서 두 번째 비트가 이용되는 등등으로서 외부 노드가 만날때까지 수행된다. 이것에 대한 코드는 이진 트리 검색에 대한 코드와 같다. 유일한 차이는 키들이 기수 정렬에서 이용된 형태인 `bitskey`이고, 키 비교대신에 개개 비트들을 접근하기 위해서는 `bits` 함수를 이용한다.(10장에서 보면, `v.bits(k, j)`는 `v`의 오른쪽에서 부터 `k`가 나타나는 `j` 비트를 나타낸다. 즉, 오른쪽으로 `k` 비트를 이동시키므로써 기계어로 효율적으로 구현된다. 그때 가장 오른쪽에 있는 `j` 비트들 거의 모두는 0으로 세트된다)

```
infoType Dict::search(itemType v) // Digital tree
{
    struct node *x = head;
    int b = itemType::maxb;
    z->key = v;
    while ( v != x->key )
        x = ( v.bits(b--, 1) ) ? x->r : x->l;
    return x->info;
}
```

이같은 프로그램에 대한 자료구조는 기초적인 이진 검색 트리의 것과 같다. 상수 `maxb`는 정렬된 키들에서 비트의 수이다. 프로그램은 각 키에서 첫 번째 비트가(오른쪽에서 `(max+1)` 번째)는 0이므로,(아마도 키는 `maxb`의 두 번째 인수로 된 `bits`를 이용한 결과이다) 검색은 0으로된 트리 헤더 노드에 대한 연결과 검색 트리를 가르키는 왼쪽 연결인 `head`에서 시작을 한다고 가정하자. 이와 같이 프로그램에 대한 초기화 프로시저는 `head->r = z` 대신에 `head->l = z`로 시작하는 것을 제외하고는 이진 트리 검색과 같다.

10장에서 동일한 키는 기수 정렬에서 없는 것으로 간주하였다. 동일한 것은 특별한 알고리즘에서는 아니지만 나중에 조사하게 될 기수 검색에서는 사실이다. 이와 같이 본 장에서는 자료구조에서 나타난 모든 키들이 서로 다르다고 한 것이다. 필요한 경우, 연결 리스트는 키들이 그 값을 지니는 레코드 각각의 키 값에 대해 유지된다. 이전 장에서 처럼, 알파벳에서의

A	0 0 0 0 1
S	1 0 0 1 1
E	0 0 1 0 1
R	1 0 0 1 0
C	0 0 0 1 1
H	0 1 0 0 0
I	0 1 0 0 1
N	0 1 1 1 0
G	0 0 1 1 1
X	1 1 0 0 0
M	0 1 1 0 1
P	1 0 0 0 0
L	0 1 1 0 0

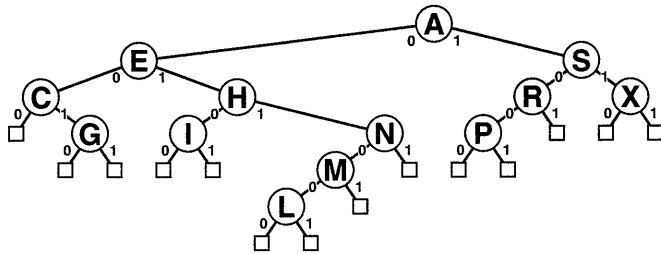


그림 17.1 디지털 검색 트리

i 번째의 문자는 i 의 5개 비트로 주어진 표현으로서 표현된다. 본 장에서 이용된 표본 키들은 그림 17.1에 주어져 있다. bits들과 일치하도록 하기 위해서, 비트들을 오른쪽에서 왼쪽으로 0에서 4에 이르는 수로서 고려를 한다. 이와 같이 비트 0은 A의 유일한 0이 아닌 비트이고, 비트 4는 P의 유일한 0이 아닌 비트이다.

디지털 검색 트리에 대한 삽입 프로시저는 이진 검색 트리들에 대해 대응되는 프로시저에서 직접적으로 나오는 것이다.

```
void Dict::insert(itemType v, infoType info)
{
    struct node *p, *x = head;
    int b = itemType::maxb;
    while (x != z)
    {
        p = x;
        x = (v.bits(b--, 1)) ? x->r : x->l;
    }
    x = new node;
    x->key = v; x->info = info; x->l = z; x->r = z;
    if (v.bits(b+1, 1)) p->r = x; else p->l = x;
}
```

표본 키들을 처음에 비어있는 트리로 삽입할 때, 이 프로그램에서 생성된 트리가 그림 17.1에 제시되어 있다. 그림 17.2는 새로운 키 $Z = 11010$ 이 그림 17.1의 트리에 첨가될 때 발

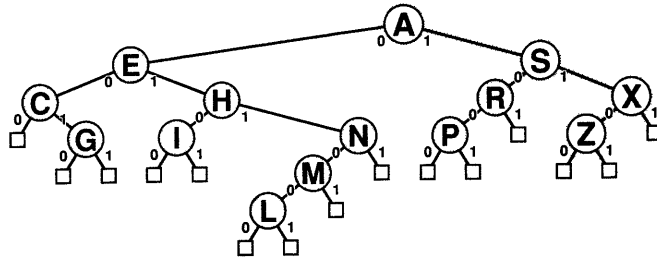


그림 17.2 디지털 검색 트리로 삽입(Z)

생되는 것이다. Z의 앞 두비트가 1이기 때문에 오른쪽으로 두 번가고, 그 뒤에 왼쪽으로 간다. 여기서, Z가 삽입된 X의 왼쪽에서 외부 노드를 만나게 된다.

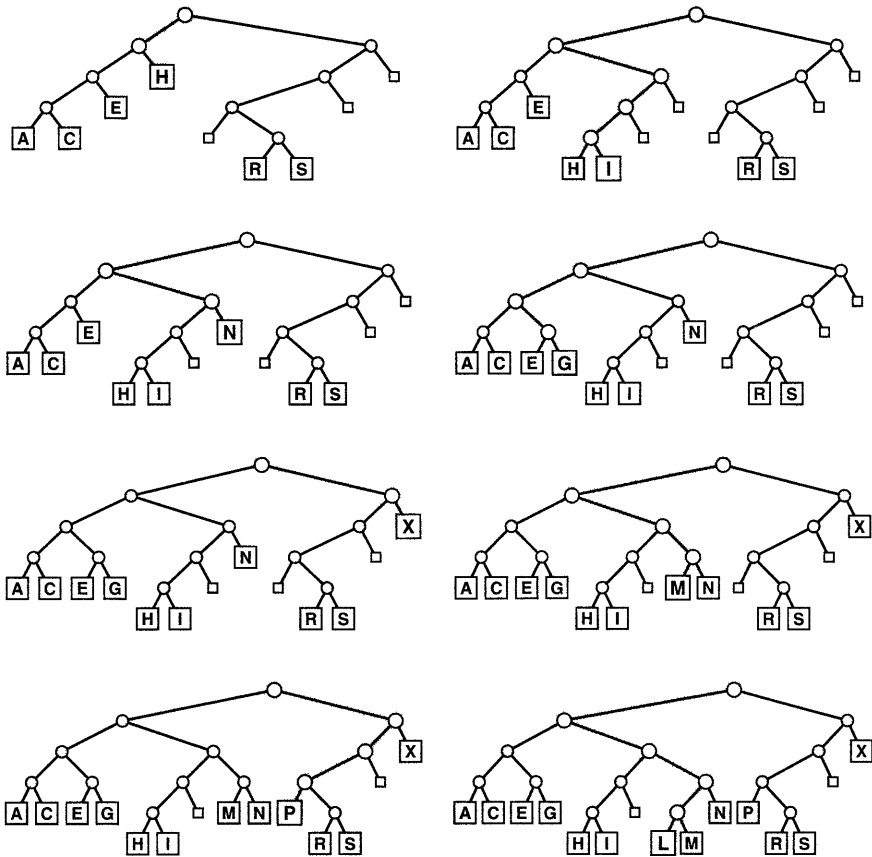
디지털 검색으로 생성된 트리의 최악의 경우는 키들의 수가 크고, 키들이 길지 않은 경우에 이진 검색 트리에서 보다 훨씬 더 좋다. 디지털 검색 트리에서 가장 긴 경로의 길이는 트리에서 어떤 두 개 키들 사이 앞에 존재하는 비트들에서 가장 길게 일치하는 길이이고, 이것은 많은 응용에 대해 상대적으로 적은 것이다.(예를 들면, 키들이 난수 비트들로 구성되는 경우이다)

성질 17.1 디지털 검색 트리에서 검색이나 삽입은 평균적으로 약 $\lg N$ 번 비교이고, N 개의 무작위 b -비트 키들에서 생성된 트리에서는 최악의 경우에 b 번의 비교이다.

경로는 키들에서 비트의 수보다 어떤 긴 것이 없는 것은 명백하다. 예를 들면, 디지털 검색 트리는 8개의 문자 키들로 생성된 디지털 검색 트리, 말하자면 문자당 6개 비트는 거기에 수십만개의 키들이 존재한다고 할지라도 경로는 48개 이하로 되어진다. 디지털 검색 트리가 평균적으로 거의 완전하게 균형화된 결과는 비록 무작위 키의 “다음번” 비트가 1 비트와 같은 정도로 0 비트가 존재하는 것으로 키들의 반 정도는 어떤 노드에서 어느 측면에 존재한다는 직관적인 개념이 확실하지만 그것은 본 교재의 범위를 벗어난 수학적 분석을 요구한다. 그림 17.3은 95개의 무작위 7-비트 키들에서 만들어진 디지털 검색 트리를 나타낸 것이다. 즉, 이같은 트리는 아주 잘 균형이 잡힌 것이다. □

이와 같이 디지털 검색 트리는 비트 추출이 키 비교를 하는 것 만큼 쉬운 것인 경우 표준 이진 검색 트리에 대한 매력적인 대안을 제공한다.(그리고 이것은 기계 의존으로 고려된다)

평상시와 같이 비 성공적인 검색에 대해, 키를 포함하지 않는 경우, 검색을 종료하는 외부 노드를 대체시키므로써 찾아진 키를 삽입할 수가 있다. 이것은 그림 17.5의 첫 번째 트라이에 제시된 것과 같이 H가 그림 17.4의 트라이로 삽입되는 경우이다. 만약 검색을 종료하는 외부 노드가 키를 포함하는 경우, 그때 찾아진 키를 지닌 내부 노드와 그 아래에 있는 외부



노드에서 검색을 종료하는 키에 의해서 대체된다. 불행히도 만약 이같은 키들이 더 많은 비트 위치로서 동의되면, 트리에서 키가 없는 것에 대응되는 어떤 외부 노드를 첨가시키는 것이 필요하다.(혹은 또 다른 방법에서, 자식으로서 비어있는 외부 노드인 어떤 내부 노드들) 이것은 그림 17.5의 두 번째 트라이에 제시된 것과 같이 I가 삽입이 될 때 발생된다. 그림 17.5의 나머지는 키들 N G X M P L이 추가되는 것으로서 예제를 완료할 수가 있다.

이 방법을 C++로 구현하는 것은 노드들의 두 가지 형태가 내부 노드의 연결에 의해서 지칭되도록 유지하는 필요성 때문에 엄격한 것을 요구한다. 이것은 저급 구현이 고급 구현에서 보다 더 쉬운 알고리즘의 예제이다. 이 문제를 피하기 위한 개선점을 아래에서 볼 수가 있기 때문에, 이것에 대한 코드는 생략한다.

이진 기수 검색 트라이의 왼쪽 부분 트리는 앞에 존재하는 비트에 대해 0을 지닌 모든 키들을 지닌다. 즉, 오른쪽 부분 트리는 앞의 비트들에 대해 1을 지닌 모든 키들을 지닌다. 이것은 기수 정렬에 대한 즉각적인 대응으로 된다. 즉, 이진 트라이 검색은 기수 교환 정렬에서와 같이 정확히 같은 방법으로서 파일을 분할을 한다.(키들이 다소 다름을 주시한 뒤에, 기수 교환 정렬에 대한 분할 다이어그램인 그림 10.1과 위의 트라이를 비교하라) 이같은 대응은 이진 트리 검색과 퀵 정렬이 유사하다.

성질 17.2 기수 검색 트라이에서 검색이나 삽입은 평균적인 검색에 대해 약 $\lg N$ 비트 비교를 하고, N 개 무작위 b -비트 키들에서부터 생성된 트리의 최악의 경우는 b 비트 비교한다.

위의 것처럼, 비록 조사된 각 비트가 1 비트와 같은 정도로 0 비트가 존재하는 것이므로서, 키들의 반 정도는 어떤 트라이 노드의 각 측면에 존재한다는 직관적인 개념이 확실하지만, 최악의 경우 결과는 알고리즘에서 직접적으로 나오고 평균의 경우 결과는 본 교재의 범주를 벗어난 수학적 분석을 요구한다. □

기수 트라이의 괴로운 특징과 검색 트리의 다른 형태들에서 그것들을 구별하는 것은 공통적으로 비트의 많은 수들로 키들에 대해 요구되는 “한 방향” 분기이다. 예를 들면, 마지막 비트에서만 다른 키들은 얼마나 많은 키들이 트리내에 있는가에 관계없이 키 길이와 같은 길이를 지닌 경로를 요구한다. 내부 노드들의 수는 키들의 수보다 다소 크다.

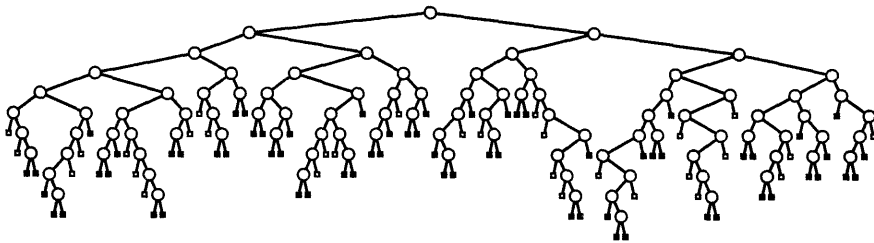


그림 17.6 큰 기수 검색 트라이

성질 17.3 N 무작위 b -비트 키들에서 생성된 기수 검색 트라이는 평균적인 경우 약 $N/\ln 2 \approx 1.44 N$ 을 지닌다.

다시 이같은 결과의 증명은 비록 그것이 실험적으로 쉽게 검증되지만 본 교재의 범주를 벗어난 것이다. 그림 17.6은 131개 노드를 지닌 95개의 무작위 10-비트 키들에서 생성된 트라이이다. □

트라이의 높이는 키들에 대해 비트의 수에 의해서 여전히 제약을 받지만 그러나 해독된 분자 데이터에서 야기되는 것과 같이 아마도 어떤 유일성을 지니는 매우 긴 키들(말하자면 1000 비트나 그이상)로된 레코드를 처리하는 가능성을 고려한 것이다. 트리에서 경로를 간단히 하는 한 방법은 노드당 두 개 연결이상을 이용한다.(비록 이것이 너무 많은 노드들을 이용하는 “공간” 문제를 악화시키지만) 또 다른 방법은 한 방향 분기를 포함한 경로를 한 개의 연결들로 “만드는 것”이다. 다음 두 절에서 이 방법들에 대해 논하기로 하자.

다중 방법 기수 검색(Multiway Radix Searching)

기수 정렬에 대해, 한 번에 한 비트이상을 고려함으로써 속도면에서 상당한 개선점을 얻게 된다. 기수 검색에 대해서도 같은 것이다. 즉, 한 번에 m 개 비트를 조사함으로써, 2^m 의 요인에 의해서 검색 속도를 높일 수가 있다. 그러나 기수 정렬에서 필요한 것 이상으로 개념을 적용할 때는 더 조심스러울 필요가 있다. 문제는 한 번에 M 비트를 고려한다는 것은 $M = 2^m$ 연결들로 된 트리 노드들을 이용하는 것에 대응되는 것이다. 그리고 이것은 사용되지 않은 연결들에 대해서는 상당한 양의 공간이 소비된다.

예를 들면, 만약 $M = 4$ 인 경우, 그림 17.7에 제시된 트라이는 예제 키들에 대해 형성된다. 이같은 트라이를 검색하기 위해서, 키들에서 한 번에 두 비트씩 고려한다. 즉, 만약 첫 번째

두 비트가 00이면, 첫 번째 노드에서 왼쪽 연결을 취하고, 01이면 두 번째 연결을 취하고 그리고 11이면 오른쪽 연결을 취한다. 그때 세 번째와 네 번째 등등에 따라서 다음 레벨로 분기한다. 예를 들면, 그림 17.7의 트라이에서 $T = 1\ 0100$ 에 대해 검색을 위해서, 근에서 세 번째 연결을 취하고, 그때 외부 노드에 접근을 하기 위해서 근의 세 번째 자식에서 세 번째 연결을 취한다. T 를 삽입하기 위해서, 그 노드는 T 를 포함한 새로운 노드에 의해서 대체된다. (그리고 4개의 외부 연결들)

사용되지 않은 외부 연결들의 많은 수 때문에 이같은 트리에서 어떤 소비된 공간이 존재함을 주시해야 한다. M 가 크면 클수록, 이같은 효과는 더 악화된다. 즉, 사용된 연결의 수는 무작위 키들에 대해 약 $MN/\ln M$ 이다. 다른 한편, 이것은 매우 효율적인 검색 방법이다. 즉, 수행시간은 약 $\log_m N$ 이다. 합당한 타협안은 다중 방법 트라이의 시간 효율성과 제일 위에서는 M 의 큰 값으로(말하자면 첫 두 개 레벨) 그리고 제일 밑에서는 M 의 작은 값(또는 어떤 기초적인 방법)으로서 된 “하이브리드” 방법을 이용해서 다른 방법들의 공간 효율성 사이에 부딪치는 것이다. 그러나 그런 방법들의 효율적인 구현은 여러개의 노드 형태 때문에 아주 복잡하다.

예를 들면, 두 개 레벨 32-방법 트리는 키들을 1024개의 범주로 나누고, 각각은 트리에서 두단계씩 밑으로 가면서 접근된다. 이것은 범주당(단지) 몇 개의 키들로서 존재하기 때문에 키들의 수천개 파일들에 대해 아주 유용하다. 다른 한편으로 더 적은 M 는 수백개의 키들의 파일에 대해 적절하다. 그 이유는 그렇지 않으면 대부분의 범주는 비어있고, 너무 많은 공간이 소비되고, 더 큰 M 는 수 백만의 키들로 된 파일에 대해 적절하다. 그 이유는 그렇지 않으면 대부분의 범주는 너무 많은 키들을 지니고, 너무 많은 시간이 소비된다.

예를 들어서 “하이브리드” 검색 방법은 전화번호부에서의 이름과 같은 것에 대해 우리 인간들이 검색하는 방법과 아주 유사한 것이다. 첫 번째 단계는 순차적인 검색에 의해 따르는 어떤 두 가지 방법의 판단으로 따르는 다중 방법 판단이다. 물론 컴퓨터는 다중 방법 검색에서 인간보다 다소 더 낫은 것으로 존재하므로서 두단계 레벨이 적절하다. 또한 26방법 분기

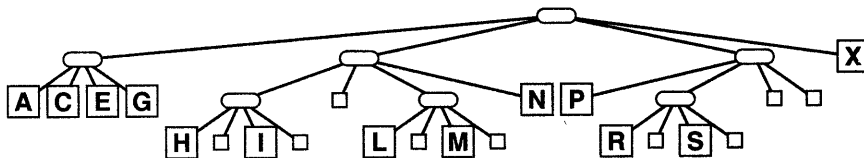


그림 17.7 4가지 방법 기수 트라이

(훨씬 더 많은 레벨로서)는 단순히 문자들로서 구성되는(예를 들면, 사전에서) 키들의 고려를 위한 아주 합당한 대안이다.

다음 장에서는 임의의 파일 크기에 대해 다중 방법 기수 검색의 잇점을 취하는 구조를 채택케 되는 체계적인 방법을 보게 된다.

패트리시아(Patricia)

위에 제시된 기수 트라이 검색은 두 가지 나쁜 단점이 있다. 즉, “한 방향 분기”는 트리에서 특별한 노드의 생성으로 되고 코드를 다소 복잡하게 하는 트리에서의 두 가지 다른 형태의 노드들이 있다.(특히 삽입 노드) D. R. Morrison는 패트리시아(Patricia: “Practical Algorithm To Retrieve Information Coded In Alphanumeric”)로 명명된 방법에서 이같은 문제들 둘다를 피하는 방법을 발견했다. 아래 주어진 알고리즘은 Morrison에 의해서 제시된 것과 같은 형태로서 된 것은 아니다. 그 이유는 19장에서 보게 될 형태의 “스트링 검색” 응용에 관심이 있기 때문이다. 현재 내용에서 패트리시아는 단지 N 개의 노드들로 된 트리에서 N 개의 임의적으로 긴 키들에 대해 검색을 허용하나, 검색당 단지 하나의 완전한 키 비교를 요구한다.

한 방향 분기는 간단한 장치로서 피하게 된다. 즉, 각 노드는 그 노드에서 취해진 것이 어느 경로인지를 결정하기 위해서 테스트되어진 비트의 인덱스를 포함한다. 외부 노드들은 트리에서 위로 향하게 가르키는 연결들로 된 외부 노드들에 대한 연결, 거꾸로는 한 개의 키와 두 개의 연결을 지닌 트리에서 정상적인 형태에 대한 연결로 대체를 시키므로써 피하게 된다. 패트리시아가 어떻게 작동되는가를 보기 위해서, 전형적인 트리에서 어떻게 처리가 되는지를 먼저 보고 그리고 트리는 첫 번째 위치에서 어떻게 구성되는가를 조사하는 것이다. 그림 17.8에 제시된 패트리시아 트리는 예제 키들이 연속적으로 삽입될 때 구성된다.

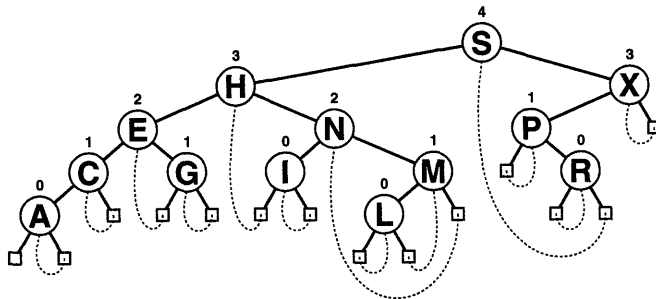


그림 17.8 패트리시아 트리

이같은 트리에서 검색을 하기 위해서, 검색 키에서 조사가 되어진 것이 어느 비트인가를 말해주기 위해, 각 노드에서 비트 인덱스를 이용하여 근에서 시작에서 트리의 밑으로 처리된다. 즉, 그 비트가 1이면 오른쪽으로 가고 0이면 왼쪽으로 간다. 노드들에서 키들은 트리의 밑으로 가는 방향으로서 전혀 조사가 되지 않는다. 결국 상위의 연결이 만나게 된다. 즉, 각 상위 연결은 그 연결을 취하는 검색을 야기시키는 비트를 지닌 트리에서 유일한 키를 가르키게 된다. 예를 들면, S는 비트 패턴 10*11과 일치하는 트리에서의 유일한 키이다. 이와 같이 만난 첫 번째 상위 연결에 의해서 지적되는 노드의 키가 검색 키와 같은 경우에 검색은 성공적으로 된다. 그렇지 않으면, 비 성공적인 경우가 된다. 트라이에 대해, 모든 검색들은 외부 노드에서 종료되고, 한 개의 딱찬 키 비교가 검색이 성공인지 아닌지를 결정하기 위해서 수행된다. 패트리시아에 대해서는 모든 검색이 상위의 연결에서 종료되고, 한 개의 딱찬 키 비교가 검색이 성공적인지를 결정하기 위해서 수행된다. 더구나 그 노드에서(정의에 의해) 비트 인덱스들은 트리를 따라 아래로 진행하는 것과 같이 감소가 되기 때문에 연결이 위를 가르키는 것인지를 테스트하는 것은 쉽다. 이것은 패트리시아에 대한 다음의 검색 코드를 나타내는 것이고, 기수 트리나 트라이 검색에 대한 코드와 같이 간단한 것이다.

```
infoType Dict::search(itemType v) // Patricia tree
{
    struct node *p, *x;
    p = head; x = head->l;
    while ( p->b > x->b )
    {
        p = x;
        x = ( bits(v, x->b, 1) ) ? x->r : x->l;
    }
    if ( v != x->key ) return infoNIL;
    return x->info;
}
```

이같은 함수는 키 v로 된 레코드를 포함하는 유일한 노드를 찾는 것이고 그때 검색이 정말로 성공한지를 테스트한다. 이와 같이 위의 트리에서 Z = 11010에 대한 것을 검색하기 위해서, 오른쪽으로 가고나서 X의 오른쪽 연결에서 위로 간다. 거기의 키는 Z가 아니므로 검색은 비 성공적이다.

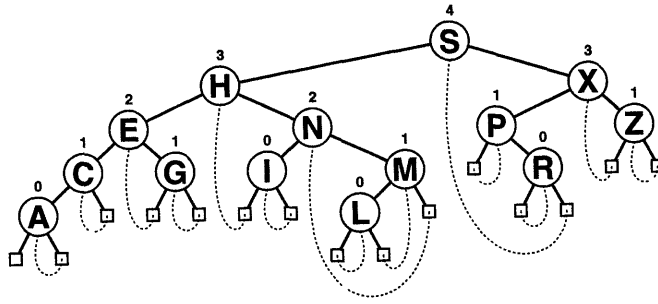


그림 17.9 패트리시아 트리로 '외부' 삽입

그림 17.9는 $Z = 11010$ 을 그림 17.8의 패트리시아 트리로 삽입하는 결과를 제시한 것이다. 위에 기술한 것과 같이, Z 에 대한 검색은 $X=11000$ 를 포함한 노드에서 끝이 난다. 트리의 정의된 성질에 의해서, X 는 검색이 그 노드에서 종료되는 트리의 유일한 키이다. 만약 Z 가 삽입되면, 그런 노드가 두 가지 있고 그래서 X 를 포함한 노드를 따르는 상위의 연결은 X 와 Z 가 서로 다른 가장 왼쪽의 점에 대응되는 비트 인덱스로서 그리고 두 개의 상위 연결로서 Z 를 포함한 새로운 노드를 가르키도록 한다. 즉, 하나는 X 를 가르키고 다른 하나는 Z 를 가르킨다. 이것은 X 를 포함한 외부 노드와 비트 인덱스를 포함하여서 제거가 되어진 한 방향 분기로서 기수 트라이 삽입에서 자식으로서 X 와 Z 로 된 새로운 내부 노드로 대체가 되어지는 것에 대응된다.

$T = 10100$ 을 삽입하는 것은 그림 17.10에 제시된 것과 같이 더 복잡한 경우를 나타낸다. T 에 대한 검색은 P 가 패턴 $10*0*$ 로 된 트리에서 유일한 키를 나타내면서 $P = 10000$ 에서 끝나게 된다. 지금 T 와 P 는 검색동안에 넘어간 위치인 비트 2에서 서로 다르다. 비트 인덱스는 트리를 따라 아래로 진행하는 것과 같이 감소가 되어진 요구사항은 T 가 자신의 비트 2에

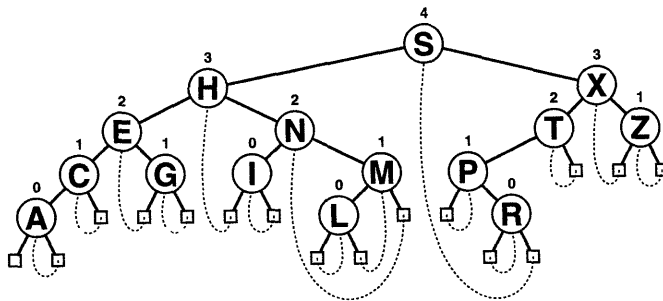


그림 17.10 패트리시아 트리로 내부적 삽입

대응되는 상위의 포인터로서 X와 P사이에 삽입되는 것을 나타낸다. 비트 2는 T의 삽입전에 넘어간 사실로 인해서 P와 R은 같은 비트-2 값을 지닌 것을 내포함을 주시해야 한다.

이같은 예제들은 패트리시아의 삽입에서 제기되는 단지 두 가지 경우를 나타낸 것이다. 다음 구현은 상세한 것을 나타낸다.

```
void Dict::insert(itemType v, infoType info)
{
    struct ndoe *p, *t, *x;
    int i = maxb;
    p = head; t = head->l;
    while ( p->b > t->b )
        { p = t; t = (bits(v, t->b, 1)) ? t->r : t->l; }
    if ( v == t->key ) return;
    while ( bits(t->key, i, 1) == bits(v, i, 1) ) i--;
    { p = x; x = (bits(v, x->b, 1)) ? x->r : x->l; }
    t = new node;
    t->key = v; t->info = info; t->b = i;
    t->l = (bits(v, t->b, 1)) ? x : t;
    t->r = (bits(v, t->b, 1)) ? t : x;
    if (bits(v, p->b, 1)) p->r = t; else p->l = t;
}
```

(이같은 코드는 head가 0의 키 필드, maxb의 비트 인덱스와 양쪽의 자신의 포인터로서 초기화 된다) 첫째, v와 구별이 되는 키를 찾기 위해서 검색을 수행한다. 조건 $x \rightarrow b > p \rightarrow b$ 는 각각이 그림 17.10과 17.9에 제시된 상황을 특성화한 것이다. 그때 그 점에서 트리를 따라 아래로 가면서 서로 다른 가장 왼쪽의 비트 위치를 결정하는 것이고 그 점에서 v를 포함한 새로운 노드를 삽입하는 것이다.

패트리시아는 본질적으로 기수 검색 방법이다. 즉, 검색 키들을 구별하는 비트들을 인식하는 것으로 관리를 하고 똑같은 자료구조에서 어떤 검색 키에서부터 유일한 키에 이르는 것으로 된 자료구조로(잉여의 노드없이) 생성하는 것이다. 명백히, 패트리시아에서 이용된 같은 기법은 한 방향 분기를 제거하기 위한 이진 기수 트라이 검색에서 이용되어지나 다중-노드-형태 문제를 악화시키는 것이다. 그림 17.11은 그림 17.6의 트라이를 생성하기 위해 이용되는 같은 키에 대해 패트리시아 트리를 제시한 것이다. 즉, 이같은 트리는 44%이하의 노드들로만 지닐 뿐 아니라 아주 잘 균형을 이룬 것이다.

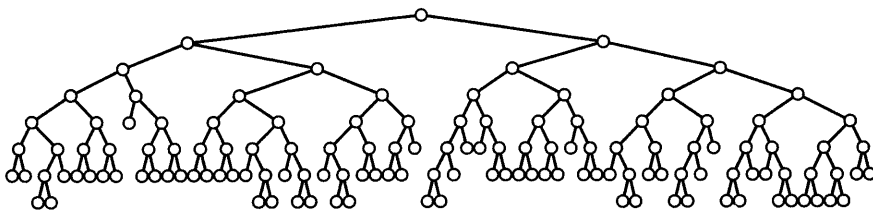


그림 17.11 큰 패트리시아 트리

표준 이진 트리 검색과는 달리, 기수 방법은 키들이 삽입이 되어지는 순서에 민감하지 않다. 즉, 키들 그자체의 구조에서만 의존된다. 패트리시아에 대해 상위 연결의 위치는 삽입 순서에 의존하는 것이 아니나, 트리 구조는 다른 방법에서와 같이 키들에서 비트들에만 의존을 한다. 이와 같이, 비록 패트리시아가 001, 0001, 00001, 000001 등과 같은 키들의 집합으로 된 문제를 지닌다고 할찌라도 정상적인 키 집합에 대해, 트리는 상대적으로 잘 균형을 잡힌 것이므로서 비록 매우 긴 키들에 대해, 비트 검사들의 수는 트리에서 N 개 노드들이 존재할 때 $\lg N$ 에 비례를 한다.

성질 17.4 N 개 무작위 b -비트 키들에서 생성이 된 패트리시아 트라이는 N 개 노드를 지니고 평균적인 검색에 대해 $\lg N$ 비트 비교를 한다.

이 장의 다른 방법들에 대한 것과 같이, 평균적인 경우의 분석은 오히려 어렵다. 즉, 패트리시아는 표준 트라이에서 수행하는 것보다 평균적으로 한 개 적은 비교를 수반한다. \square

기수 트라이 검색의 가장 유용한 특징은 다양한 길이의 키들로 그것이 효율적으로 수행되는 것이다. 다른 검색 방법의 모든 것에서, 키들의 길이는 어떤 방법으로서 검색 프로시저를 “생성하는” 것이고 그결과 실행시간은 키들의 수 뿐만 아니라 길이에 의존을 한다. 이용 가능한 특별한 절약은 이용된 비트의 방법에 의존한다. 예를 들면, 데이터의 8-비트 “바이트들”을 효율적으로 접근할 수 있는 컴퓨터와 수백개의 1000-비트 키들 가운데 검색을 한다고 가정하자. 그때 패트리시아는 검색에 대해 검색 키의 약 9나 10 바이트들과 한 개의 125-바이트와 동등한 비교를 접근하는 것이 요구되고 반면에 해싱은 해시 함수와 몇 개의 동등한 비교를 더한 것을 계산하는 검색 키의 모든 125 바이트를 접근하는 것이 요구가 되고 비교를 기반으로 한 방법은 여러 가지 긴 비교를 요구한다. 이같은 효과는 패트리시아로 하여금(또는 제거된 한 방향 분기로된 기수 트라이 검색) 매우 긴 키들이 수반이 될 때 선택의 검색 방법이다.

연습문제

1. 키 E A S Y Q U E S T I O N이 그같은 순서로서 처음에 비어있는 트리로 삽입될 때 결과적으로 되는 디지털 검색 트리를 그려라.
2. 1000개 노드 디지털 검색 트리를 생성하고 그것의 높이와 같은 키로 생성된 적색-흑색 트리(15장 참조)와 표준 이진 검색 트리에 대해서 각 레벨에서 노드의 수를 비교하라.
3. 특별히 나쁜 균형 디지털 검색 트라이를 만드는 12개 키들의 집합을 찾아라.
4. 키들 E A S Y Q U E S T I O N이 그같은 순서로서 처음에 비어있는 트리로 삽입을 할 때 결과적으로 되는 기수 검색 트라이를 그려라.
5. 26-방법 다중 방법 기수 검색 트라이로된 문제는 알파벳의 어떤 문자들이 매우 가끔 이용되는 것이다. 이같은 문제를 고정 시키는 방법을 제안하시오.
6. 다중 방법 기수 검색 트리에서 요인을 어떻게 삭제하는가를 기술하시오.
7. 키 E A S Y Q U E S T I O N이 그같은 순서로서 처음에 비어있는 트리로 삽입을 할 때 결과적으로 되는 패트리시아 트리를 그려라.
8. 특별히 나쁜 균형 패트리시아 트리로 만드는 12 개 키들의 집합을 찾아라.
9. 주어진 검색 키로서 똑같은 초기의 t 비트들을 지나는 패트리시아 트리에서 모든 키들을 인쇄하는 프로그램을 기술하시오.
10. 정렬된 순서로서 인쇄가 되는 프로그램을 기술하는 것이 어느 기수 방법이 합당한가를 제시한 것인가? 방법들 중의 어느것이 이같은 연산에 대해 수정이 가능하지 않는가?

빈 면

18 장

외부 검색

매우 큰 파일로부터 항목들을 접근하는데 적절한 검색 알고리즘은 실제로 중요한 것들이다. 검색은 큰 데이터 파일에서의 기본적인 연산이고 많은 컴퓨터 설치장소에서 이용되는 자원들의 매우 의미가 있는 조각을 소비하는 것이다.

디스크 검색은 가장 실제적인 관심이 되기 때문에 큰 디스크 파일의 검색에 대한 방법을 주로 다루는 것이다. 테이프와 같은 순차적인 장치로서, 검색은 느린 방법으로 되보시키는 것이다. 즉, 항목에 대해 테이프를 검색하기 위해서, 테이프를 설치하고 항목이 발견될때까지 계속적으로 읽는 것보다 훨씬 좋을 것은 없다. 현저하게, 다루어야 할 방법은 단지 두 개나 세 개의 디스크 접근들로서만 1억 단어정도의 큰 디스크에서 항목을 찾을 수가 있다.

외부 정렬과 같이, 복잡한 입/출력 하드웨어의 “시스템” 면은 외부 검색 방법의 활용도에 서 주된 요인이 되나 상세히 공부할 필요는 없다. 그러나 외부 방법들이 내부 방법들과 아주 다른 정렬과는 달리, 외부 검색 방법은 내부 방법들의 논리적인 확장이다.

검색은 디스크 장치에 대한 기본적인 연산이다. 파일들은 가능한 효율적으로 정보를 접근하도록 하므로써 특별한 장치의 특성에 대한 잊점을 취하도록 구성되었다. 정렬에서와 같이, 기본적인 방법들의 주된 특징들을 설명하기 위해서 “디스크” 장치들에 대해 오히려 간단하고 자세하지 않은 모델로서 처리된다. 특별한 응용에 대한 가장 좋은 외부 검색 방법을 결정하는 것은 극도로 복잡하고, 하드웨어(그리고 시스템 소프트웨어)에 매우 의존적이므로서 본 교재의 범주를 벗어난 것이다. 그러나 이용이 될 일반적인 방법들을 제시한다.

많은 응용에 대해, 매우 큰 파일들 내부에 정보의 적은 비트들을 변경하고, 추가하고, 삭제하거나 혹은(가장 중요한 것은) 빠른 접근에 대해서 가끔은 좋다. 본 장에서는 이진 검색과

순차적인 검색에 대한 이진 검색 트리와 해싱을 제공하는 간단한 방법들의 잇점을 제공하는 그런 동적인 상황에 대해 몇 가지 방법들을 조사하게 된다.

컴퓨터를 이용하여 처리되는 정보의 매우 큰 집합을 데이터 베이스(data base)라고 부른다. 대부분의 내용은 데이터 베이스를 이용하고 생성하고 유지하는 방법들로 이루어진다. 그러나, 많은 데이터 베이스는 매우 높은 비활동성을 지닌다. 즉, 매우 큰 데이터 베이스가 특별한 검색 방법을 생성한다면, 또 다른 곳에 다시 생성하도록 하는데는 비용이 비싼 것이다. 이같은 이유로서, 더 오래되고 정적인 방법이 널리 이용되고 비록 새로운 것 즉 동적인 방법들이 새로운 데이터 베이스에 대해 이용을 시작을 하지만 그 상태로 남아있게 하는 것이다.

데이터 베이스 응용 시스템은 한 개의 키에 대해 기본적인 항목에 대해 간단한 검색보다 훨씬 더 복잡한 연산을 제공한다. 검색들은 한 개이상 키가 포함된 범주를 기본으로 한 것이고, 큰 수의 레코드들을 되돌리도록 한다. 다음 장들에서, 이같은 형태의 어떤 검색 요청에 대해 적절한 알고리즘의 몇 가지 예제를 보게 되나 일반적인 검색 요청은 그것들이 범주에 있는 가를 보기 위해서 각 레코드를 테스트하면서 전체 데이터 베이스에 대해 순차적인 검색을 수행하는 것이 전형적인 것이다.

논의될 방법들은 모든 파일이 유일한 인식자를 지니고 파일의 목적이 그 인식자를 기본으로 한 효율적인 접근, 삽입과 삭제를 지원하는 것인 매우 큰 파일 시스템의 구현에서 실제로 중요하다. 모델은 디스크 하드웨어로서 효율적으로 접근이 가능한 것인 연속적인 정보의 블록들인 페이지들(pages)로 나누므로써 디스크 기억장소를 고려한다. 각 페이지는 많은 레코드를 지닌다. 즉, 우리의 일은 단지 몇 개의 페이지를 읽으므로써 어떤 레코드가 접근할 수가 있는가 하는 방법으로 레코드를 페이지내에 구성하는 것이다. 페이지를 읽는데 요구되는 입/출력 시간은 그 페이지를 포함한 어떤 계산을 수행하는데 요구되는 처리 시간이 된다. 위에 언급한 것과 같이, 이같은 모델은 많은 방법으로 상당히 간단하게 했으나 이용된 몇 가지 기본적인 방법들을 고려하는 것을 허용하기 위해서 실제적인 외부 기억장치의 특징들이 충분히 유지된다.

색인 순차 접근(Indexed Sequential Access)

순차적 디스크 검색은 14장에서 고려된 기초적인 순차적 검색 방법들의 자연스런 확장이다. 즉, 레코드들은 키들의 순서로 증가시키고, 검색은 검색 키보다 크거나 같은 키를 포함한 것이 찾아질때까지 레코드를 하나씩 읽어나가므로써 간단히 수행된다. 예를 들면, 만약 검색



그림 18.1 순차 접근

키들이 EXTERNALSEARCHINGEXAMPLE에서 나타내는 것이고 각각 4개 레코드가 세 개 페이지를 유지하는 디스크를 지니면, 그때 그림 18.1에 제시된 구조를 지닌다.(외부 정렬과 같이, 알고리즘을 이해하는 상당히 적은 예제들을 고려하고, 활용도를 평가하는 매우 큰 예제들에 대해 생각을 하는 것이다) 명백히, 순수한 순차적 검색은 그림 18.1에서 W에 대한 검색이 모든 페이지를 읽기를 요구하기 때문에 매력이 없다.

광대하게 검색의 속도를 증가시키기 위해서, 각 디스크에 대해 그림 18.2에서 처럼 어느 키가 어느 페이지에 속하는 가의 “인덱스”를 유지한다. 각 디스크의 첫 번째 페이지는 그것의 인덱스이다. 즉, 적은 문자들은 완전한 레코드가 아니고 키 값만이 저장되는 것을 나타내고, 적은 수는 페이지 인덱스이다.(0은 디스크에서 첫 번째 페이지를 의미하고, 1은 다음 페이지를 의미하는 등이다) 인덱스에서, 각 페이지 수는 이전 페이지상에 마지막 키의 값들 아래에 나타난 것이다.(빈칸은 다른 모든 것보다 적은 표지 키이고 “+”는 “다음 디스크상에 보는 것”을 의미한다) 예를 들면, 디스크 2에 대한 인덱스는 첫 번째 페이지가 E와 I사이에 총괄적인 키들로된 레코드를 포함한다. 그리고 두 번째 페이지는 I와 N사이에 총괄적인 된 키들의 레코드를 포함한다. “데이터” 페이지에 있는 레코드보다 인덱스 페이지에 있는 더 많은 키들과 페이지 인덱스들을 맞게 하는 것은 가능하다. 사실 전체 디스크에 대한 인덱스는 단지 몇 개의 페이지만을 요구한다.

검색을 더 신속히 처리하기 위해서, 이같은 인덱스들은 어느 키가 어느 디스크에 존재하는 가를 알려주는 “원장(master) 인덱스”로서 대체가 된다. 예를 들면, 원장 인덱스에서 디스크 1은 E보다 적거나 같은 키들을 포함하고 디스크 2는 N보다 적거나 같은 키들을 포함하고(그러나 E보다 적지 않은 것) 그리고 디스크 3은 X보다 적거나 같은 키를 포함한다.(그러나

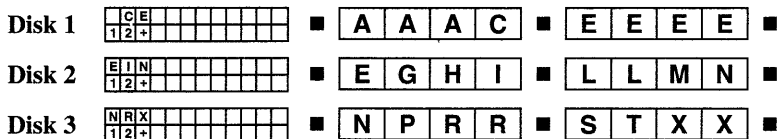


그림 18.2 색인 순차 접근

E보다 적지 않은 것과 말한다. 원장 인덱스는 그것이 기억공간에 유지가 되기에 충분히 적은 것이므로 해서 대부분의 레코드들은 단지 두 개 페이지만을 접근함으로써 찾아진다. 그 중 하나는 디스크상의 인덱스에 대한 것이고, 다른 하나는 레코드를 포함한 페이지에 대한 것이다. 예를 들면, W에 대한 검색은 디스크 3에서 인덱스 페이지를 먼저 읽는 것으로 다음에는 W를 포함하는 유일한 것인 디스크 3에서 두 번째 데이터 페이지를 읽는 것을 수반한다. 즉, 인덱스에서 키 값을 측면 지원하는 인덱스와 두 개 페이지를 더한 것이다. 만약 이중 키가 파일에서 존재하지 않으면, 특별한 페이지 접근은 피하게 된다. 다른 한편으로 만약 파일에서 많은 똑같은 키들이 존재하면, 여러 가지 페이지 접근들이 불리워진다.(똑같은 키들로된 레코드는 여러 가지 페이지를 채운다)

순차적인 키 구조와 인덱스 접근을 결합시키면, 이같은 구조를 색인 순차 접근(indexed sequential access)이라고 부른다. 그것은 데이터 베이스에 대한 변화가 빈번하게 만들어지는 응용에 대한 선택의 방법이다. 색인 순차 접근을 이용하는 단점은 매우 유동적이지 못하다. 예를 들면, 위 구조에 B를 추가하는 것은 인덱스에 대한 새로운 값들과 키들의 대분에 대해 새로운 위치로서 전체 데이터 베이스를 다시 생성도록한다.

성질 18.1 색인 순차 파일에서 검색은 디스크 접근의 상수 수들로서 요구가 되나 삽입은 전체 파일을 재 배열하는 것을 수반한다.

실제적으로 여기에 수반되는 “상수”는 디스크의 수와 레코드, 인덱스와 페이지들의 상대적인 크기에 의존한다. 예를 들면, 한 개 단어 키의 많은 파일은 접근에서 상수의 수로서 검색을 허용하는 방법과 같이 단지 한 개 디스크에 저장될 수가 없다. 혹은 다른 극단적인 미묘한 예제를 취하기 위해서, 각각이 단지 한 개 레코드를 유지하는 많은 수의 매우 적은 디스크들 또한 검색하기가 어렵다. □

B-트리(B-trees)

동적 상황에서 검색을 하는 좋은 방법은 균형 트리를 이용하는 것이다.(상대적으로 비싼) 디스크 접근의 수를 감소시키기 위해서, 노드당 키들의 많은 수를 허용하는 것이 합당하므로서 노드들은 많은 분기 요인을 지닌다. 그런 트리들을 외부 검색에 대해 다중 방법 균형 트리의 이용을 고려한 첫번째로 R. Bayer와 E. McCreight에 의해서 B-tree라고 명명을 하였

다.(많은 사람들은 Bayer와 McCreight에 의해서 제시된 알고리즘에 의해 생성된 정확한 자료구조를 기술하기 위해서 “B-트리”라는 말을 쓰고 있다.)

2-3-4 트리에 대해 이용되는(15장 참조) 하향식 알고리즘은 노드당 더 많은 키들을 처리할 준비가 되도록 확장을 한 것이다. 즉, 노드당 1에서 $M-1$ 개 키들에 이르기까지 어디선간 존재한다고 가정하자.(그리고 노드당 2에서 M 개의 연결들에 이르기까지 어디선간 존재한다) 검색은 2-3-4 트리와 유사한 방법으로 처리된다. 즉, 한 노드에서 다음으로 이동하기 위해서, 먼저 현재 노드에서 검색 키에 대한 적절한 간격을 찾고, 다음 노드를 얻기 위해서 대응되는 연결을 통해서 빠져나간다. 이 방법을 외부 노드에 도달될 때까지 계속하고 그때 새로운 키에 도달된 마지막 내부 노드로 삽입을 한다. 하향식 2-3-4 트리로서, 트리의 아래로 진행하는 방법상에서 “확산” 것으로 존재하는 노드들을 “분리”하는 것이 필요하다. M -노드에 부착된 k -노드를 보는 순간에, 두 개의 $(M/2)$ -노드들에 부착이 된 $(k + 1)$ -노드로서 대체를 한다.(분리에 대해서, M 는 짝수라고 가정을 하자) 이것은 제일 밑에 도달이 될 때 새로운 노드를 삽입하는 방이 존재케 된다.

$M = 4$ 에 대해 구성된 B-트리와 예제 키들은 그림 18.3에 제시가 되어있다. 이 트리는 13개 노드를 지니고 각각은 디스크 페이지에 대응된다. 각 노드는 레코드 뿐만 아니라 연결들을 포함한다. 비록 그것이 우리들에서 친숙한 2-3-4트리로서 남겨 지지만 $M = 4$ 라는 선택은 이 점을 강조하기 위함을 의미한다. 즉, 초기에는 페이지당 4개 레코드가 적합을 하고 그 중 단지 3개만이 연결에 대한 방으로 남겨두는 것이 적합하다. 이용된 실제적인 공간의 양은 레코드와 연결들의 상대적인 크기에 의존을 한다. 레코드와 연결들의 이같은 하이브리드를 피하기 위한 방법은 아래에서 보게 된다.

기억 공간에 색인 순차 검색에 대해 원장 인덱스를 유지하는 것과 같이, 기억 공간에서 B-트리의 근 노드를 유지하는 것이 합당하다. 그림 18.3의 B-트리에 대해서, E와 같거나 적은 키들로된 레코드를 포함한 부분 트리의 근은 디스크 1의 페이지 0에 있고, N와 같거나 적은(그러나 E보다 적지 않는 것) 키들로된 부분 트리의 근은 디스크 1의 페이지 1에 존재하

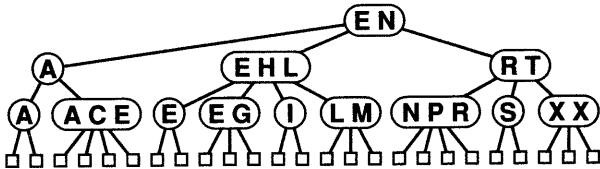


그림 18.3 B-트리

증명은 본 책의 범주를 벗어난 것이나, 소비된 공간의 양은 모든 노드들이 약 반정도 꽉찰 때인 최악의 경우에서 약 N 에 이르도록 범주가 이루어진다. □

위의 예제의 노드들에서 연결에 대한 방을 절약하기 위한 필요 때문에 $M = 4$ 를 선택하도록 강요한다. 그러나 대부분의 노드들에서는 연결을 이용하지 않은 것으로 끝난다. 그 이유는 B-트리에서 대부분의 노드들이 외부이고, 대부분 연결들이 비어있기 때문이다. 더구나 M 의 훨씬 더 큰 값은 색인 순차 접근에서와 같이 내부 노드들에서 키들만(꼭찬 레코드는 아니고) 저장할 하는 경우에 트리의 더 높은 레벨에서 이용된다. 예제에서 이같은 관찰의 잇점을 보기위해서, 페이지상 7개의 키와 8개의 연결들까지 적합하도록 하므로서 내부 노드에 대해서는 $M = 8$ 를, 제일 밑의 레벨 노드들에 대해서는 $M = 5$ 를 이용한다.(연결에 대한 공간은 제일 밑에서 유지가 될 필요가 없기 때문에 $M = 4$ 이다) 제일 밑의 노드는 추가가 될 때 분리된다.(두개 레코드들에서 하나 그리고 세 개 레코드들에서 하나로) 즉, 분리는 위의 트리가 $M = 8$ 에 대해 정상적인 B-트리로서 처리되기 때문에(레코드가 아니고 저장된 키상에서) 공간이 존재하는 위 노드로 중간의 레코드 키를 “삽입”하므로서 분리된다. 이것은 그림 18.5에 제시된 트리로 나타낸다.

전형적인 응용에 대한 효과는 트리의 분기 요인이 큰 것으로 되어진 키 크기와 레코드 크기의 비율에 의해서 증가가 되기 때문에 훨씬 더 극적이다. 또한 이같은 형태의 구조로서, “인덱스”(키들과 연결들이 포함이된) 색인 순차 검색에서와 같이 실제 레코드에서 분리된다. 그림 18.6은 그림 18.5에 트리가 어떻게 저장이 되는 가를 나타낸 것이다. 즉, 큰 노드는 비록 대부분의 응용에서 위에서 처럼 기억 공간에 유지가 되어진 것이지만 디스크 1의 페이지 0-상에 존재한다.(그림 18.5의 트리는 그림 18.3의 트리보다 한 개 적은 노드이기 때문에 그것에 대한 공간이 있다) 디스크 상에 노드 배치에 관한 위의 다른 비고들 또한 여기에 적용된다.

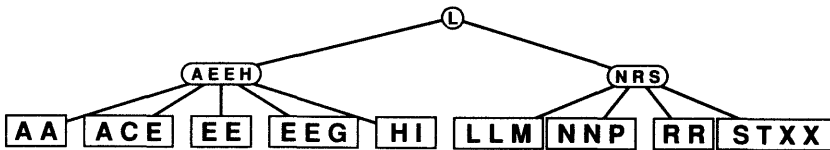


그림 18.5 외부 노드들에서만 레코드로된 B-트리

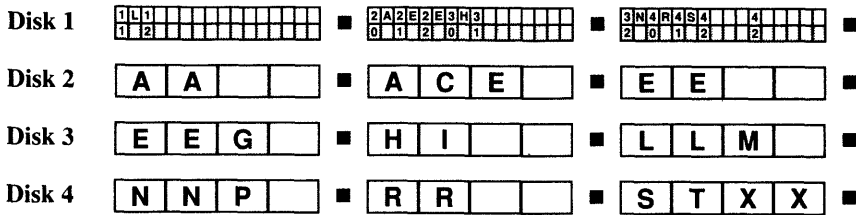


그림 18.6 외부노드에서만 레코드를 지닌 B-트리 접근

지금 M 의 두 개 값을 지닌다. 그 중 하나는 트리의 분기 요인(M_I)을 결정하는 내부 노드들에 대한 것이고, 다른 하나는 페이지들에 레코드 할당을 결정하는 제일 밑 레벨 노드들에 대한 것이다. (M_B) 디스크의 접근을 최소화하기 위해서 어떤 특별한 계산 경비로서 M_I 와 M_B 둘다 가능한 크게 만들기를 원한다. 다른 한편으로 대부분의 트리 노드들이 비어있고 공간이 소비되기 때문에 M_I 를 거대하게 만들기를 원치 않는다. 그리고 제일 밑 레벨 노드들의 순차적 검색으로 줄여주기 때문에 M_B 를 거대하게 만들기를 원치 않는다. 항상 M_I 와 M_B 둘다 페이지 크기에 관련시키는 것이 가장 좋다. M_B 에 대한 명백한 선택은 페이지(1을 더함)에 적합한 레코드의 수이다. 즉, 검색의 목적은 찾을 레코드를 포함한 페이지를 찾는 것이다. 만약 M_I 가 4개 페이지에 대해 두 개 적합한 키들의 수로서 취해지면, 그때 B-트리는 매우 큰 파일에 대해서 조차($M_I = 2048$ 로 된 세 개 레벨 트리는 1024^3 이나 1억개이상의 엔트리까지 처리가 된다) 단지 세 개의 레벨 깊이로 존재를 한다. 그러나 트리상에 모든 연산에 대해 접근이 되는 트리의 근 노드가 기억 공간에 유지를 하므로서 단지 두 개의 디스크 접근만이 파일에서 어떤 요소를 찾는 것을 요구한다.

15장의 끝에서 간단히 언급이 된 것과 같이, 더 복잡한 “하향식” 삽입 방법은 공통적으로 B-트리에서 이용된다. (비록 상향식과 하향식 방법들 사이의 차이가 세 개 레벨 트리에서는 별로 중요하지 않지만) 기술적으로 여기에 기술된 트리는 그것들과 문헌에서 공통적으로 논의된 것들을 구별하기 위해서 “상향식” B-트리로서 언급된다. 많은 다른 방법들이 기술되고 그들 중 몇몇은 외부 검색에 대해 아주 중요하다. 예를 들면 노드가 확장할 때, 분리(그리고 결과적으로 받은 비어있는 노드)는 노드 내용들의 어떤 것은 그것의 “형제” 노드로(만약 그것이 확장하지 않은 경우) 보내므로써 예견할 수가 있다. 이것은 노드들내에서 공간 이용을 더 좋게 하는 것이고 대규모 디스크 검색 응용에서 주된 관심사이다.

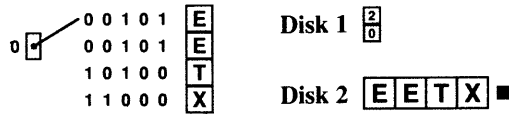


그림 18.7 확장 가능 해싱. 첫 번째 페이지

확장 가능 해싱(Extendible Hashing)

외부 검색에 적용을 위해서 디지털 검색 알고리즘을 확장시키는 B-트리의 대안은 1978년에 R.fagin, J. Nievergelt, N. Pippenger와 R. Strong에 의해서 개발되었다. 확장 가능 해싱이라 부르는 이같은 방법은 전형적인 응용에서 각 검색에 대해 두 개의 디스크 접근을 하는 반면에 동시에 효율적인 삽입이 허용된다. B-트리로서 레코드들은 그것들이 채워질 때 두 개의 조각으로 분리를 시키는 페이지에 저장된다. 즉, 색인 순차 접근에서와 같이, 검색 키와 일치하는 레코드를 포함한 페이지를 발견하도록 접근을 하는 인덱스를 유지한다. 확장 가능 해싱은 검색 키들의 디지털 성질들을 이용해서 이같은 접근들을 결합한다.

확장 가능 해싱이 어떻게 처리되는가를 보기 위해서, 4개 레코드의 용량으로서 페이지를 이용한 EXTERNAL SEARCHING EXAMPLE에서 키들의 연속적인 삽입을 어떻게 처리하는가를 고려하게 된다. 레코드를 유지하는 페이지에 대한 포인터인 단지 한 개 엔트리로된 “인덱스”로서 시작을 한다. 첫 번째 4개 레코드는 그림 18.7에 제시된 구조를 남겨두므로써 페이지에 적합하다.

디스크 1의 디렉토리는 모든 레코드들이 디스크 2의 페이지 0에 존재한다고 할 수 있고, 그들의 키들의 정렬된 순서로서 유지된다. 또한 알파벳의 i 번째 문자에 대해 i 의 5 비트 이진 표현의 표준적인 해독을 이용해서 키들의 이진 값을 주게 된다. 지금 페이지는 꽉차있고, 키 $R = 10010$ 를 추가하도록 분리를 시켜야 한다. 그 방법은 간단하다. 즉, 한 페이지에서 0으로

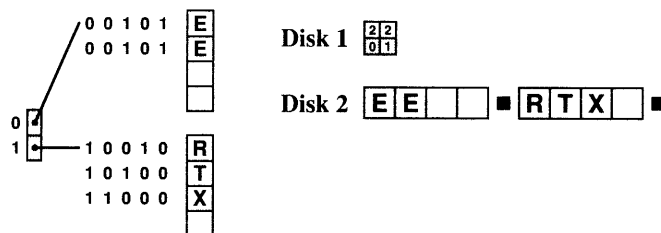


그림 18.8 확장 가능한 해싱: 디렉토리 분리

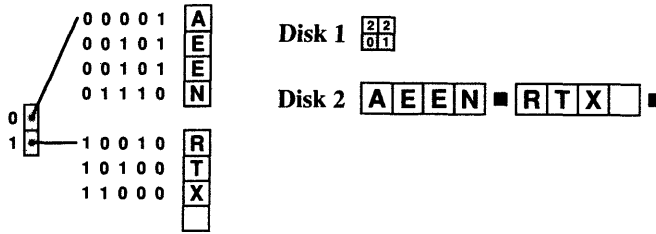


그림 18.9 확장 가능한 해싱 첫 번째 페이지는 다시 확장

시작을 하는 키들로 된 레코드와 다른 페이지에 1로 시작하는 키들로 된 레코드를 놓게 된다. 이것은 그림 18.8에서 제시된 구조를 남겨두면서 디스크 2의 페이지 0에서 키들의 반을 새로운 페이지로 이동을 시키고 디렉토리의 크기를 두배로 하는 것이 필요하다.

지금 $N = 01110$ 과 $A = 00001$ 이 추가가 되나 이것은 그림 18.9에 제시된 것과 같이 첫 번째 페이지를 다시 채운다. 또 다른 분리는 $L = 01100$ 이 추가 되기 전에 필요로 한다. 그리고 첫 번째 페이지를 두 개 조각 즉 하나는 00으로 시작되는 키들에 대한 것과 다른 하나는 01로 시작되는 키들에 대한 것으로 분리시키므로써 첫 번째 분리에 대한 것과 같은 방법으로 처리된다. 명백하지 않은 것은 디렉토리로서 수행하는 것이 무엇인가이다. 하나의 대안은 각 페이지에 한 포인터로서 또 다른 엔트리를 단순히 추가 하는 것이다. 이것은 색인 순차 검색 (비록 기수 버전)으로 본질적으로 줄여주기 때문에 매력이 없다. 즉, 디렉토리는 검색 동안에 적절한 페이지를 발견하기 위해 순차적으로 조사된다. 대안으로서 그림 18.10에 제시된 구조를 주면서 다시 디렉토리의 크기를 두배로 한다. 새로운 페이지(디스크 2상에 페이지 2)는 01로(A, E와 E) 시작하는 키들을 포함하고, 1로(R, T와 X) 시작하는 키들을 포함한 페이지

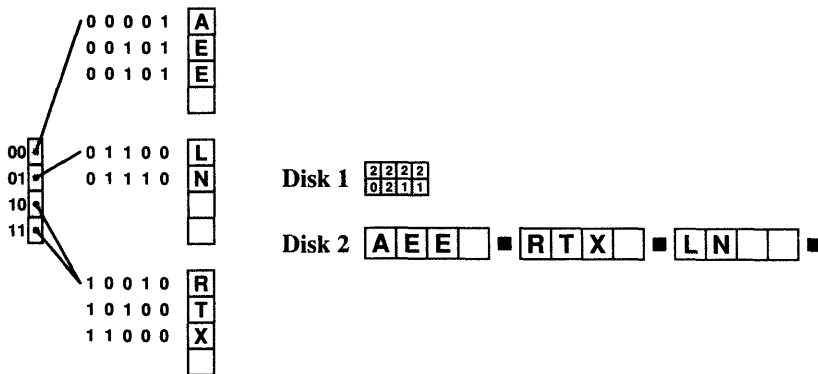


그림 18.10 확장가능한 해싱: 두 번째 분리

는 비록 그것을 지칭하는데 두 개의 포인터 즉, 하나는 10으로 시작이 되는 키들이 거기에 저장된 것을 나타내기 위한 것이고 다른 하나는 11로 시작되는 키들이 거기에 저장되는 것을 나타내는 것이지만 영향을 주지 못한다. 지금 레코드를 포함한 페이지의 번지를 포함한 디렉토리 엔트리를 직접적으로 접근하는 키의 첫 번째 두 비트를 이용해서 어떤 레코드를 접근할 수가 있다.

정렬된 순서로서 페이지내에 레코드를 유지하는 것은 주먹구구식의 간단화와 같은 것처럼 보인다. 그러나 페이지 장치에서 디스크 입/출력을 수행하고, 처리시간은 페이지를 입력하거나 출력하는 시간에 비교를 하는 기본적인 가정들을 회고할 수가 있다. 이와 같이, 키들의 정렬된 순서로서 레코드를 유지하는 것은 실제적인 경비가 아니다. 즉, 페이지에 레코드를 추가시키기 위해서, 페이지를 기억장소로 읽어들이고, 그것을 변경하고 그리고 거꾸로 인쇄하도록 해야만한다. 정렬된 순서를 유지하는데 요구되는 특별한 시간은 페이지가 크지 않을 때인 전형적인 경우에서 현저하지 않다는 것이다.

조금 더 계속해 보면, 또 다른 분리가 $A = 0001$ 을 추가할 필요가 있기전에 $S = 10011$ 과 $E = 00101$ 을 추가 할 수가 있다. 이같은 분리는 그림 18.11에 제시된 구조를 나타내면서 디렉토리를 두배로 하는 것을 요구한다. 디렉토리를 두배로 하는 처리는 간단하다. 즉, 이전 디렉토리를 읽고 이전 것의 각 엔트리를 두 번 인쇄하므로써 새로운 것으로 만든다. 이것은 분리에 의해서 생성된 새로운 페이지에 포인터에 대한 공간을 만든다.

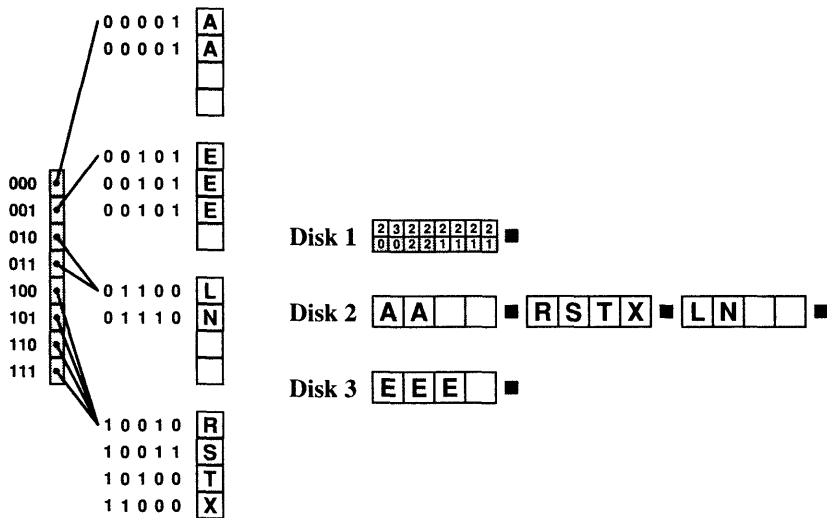


그림 18.11 확장 가능한 해싱: 세 번째 분리

일반적으로 확장 가능 해싱으로서 생성된 구조는 2^d 단어들의(각각 d -비트 패턴에 대해 하나) 디렉토리와 특별한 비트 패턴(d 비트와 같거나 적은 것)으로 시작되는 키들로 된 모든 레코드를 포함하는 잎(leaf) 페이지의 집합으로 구성된다. 검색은 디렉토리로 인덱스에 대한 키 앞의 d 비트들을 이용해서 남기고 잎 페이지에 대한 포인터를 포함한다. 그때 참조된 잎 페이지는 적절한 레코드에 대해 접근되고 검색(어떤 방법을 이용해서)된다. 잎 페이지는 한 디렉토리 엔트리이상으로 된다. 즉, 상세하게 만약 잎 페이지가 특별한 k 비트로(그림에서 그림자가 없는 것들) 시작되는 키들로 된 모든 레코드들을 포함하면, 그때 그것을 가르키는 2^{d-k} 개 디렉토리를 지닌다. 그림 18.11에서 $d = 3$ 이고 디스크 2의 페이지 1은 1 비트로서 시작하는 키들로 된 모든 레코드들을 포함하는 것으로서 그것을 지칭하는데는 4개의 디렉토리 엔트리가 있다.

지금까지 예제에서, 각 페이지 분리는 디렉토리 분리를 요구하는 것이나 정상적인 상황에서 디렉토리는 드물게 분리된다. 이것이 알고리즘의 본질적인 것이다. 즉, 디렉토리에서 특별한 포인터가 동적인 성장으로 제공하는 구조를 허용한다. 예를 들면, R이 그림 18.11에서 구조로 삽입될 때, 디스크 2의 페이지 1은 1로 시작을 하는 5개 키를 제공하는 것으로 분리된

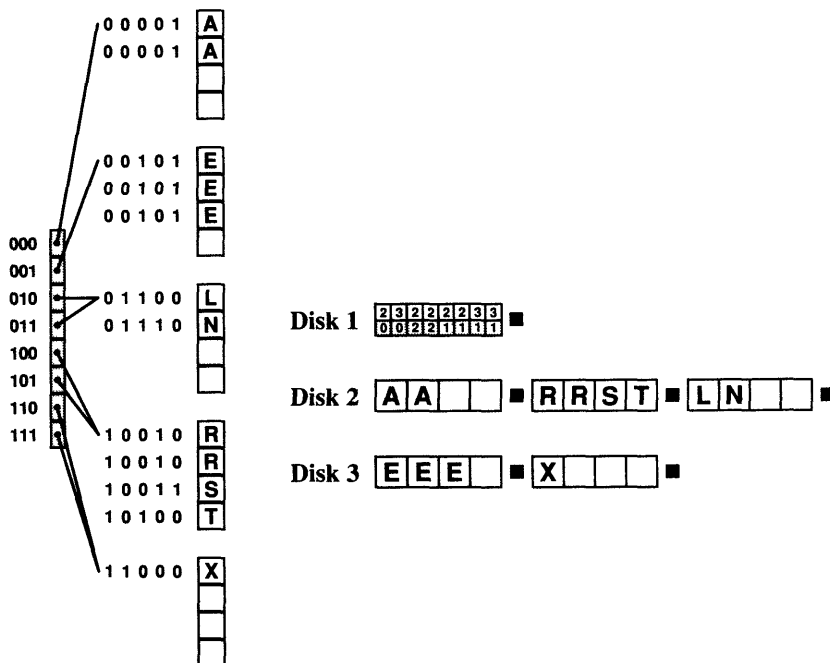


그림 18.12 확장 가능한 해싱: 4번째 분리

다. 그러나 디렉토리는 그림 18.12가 제시한 것과 같이 증가될 필요는 없다. 디렉토리에 대한 유일한 변경은 마지막 두 개의 포인터가 디스크 3의 페이지 1을 지칭하는 것으로 변경이 되고 새로운 페이지는 11로 시작하는(X) 자료 구조에서 모든 키들을 제공하도록 분리에서 생성하는 것이다.

디렉토리는 페이지에 대해 유일한 포인터이다. 이것들은 레코드들이나 키들보다 더 적은 것으로 더 많은 디렉토리 엔트리들이 각 페이지 상에 적합하다. 예제에 대해 비록 이같은 비율이 실제적으로 훨씬 높지만 페이지상에 레코드와 같은 수의 디렉토리 엔트리들이 두배로 된다. 디렉토리가 한 페이지 이상으로 펼쳐질 때, 같은 인덱싱 방법을 이용해서 어느 디렉토리 페이지가 어디에 존재하는 가를 말해주는 기억공간에서의 “근 노드”를 유지한다. 예를 들면, 만약 디렉토리가 두 개 페이지로 되면, 근 노드는 0으로 시작되는 키들로서 모든 레코드들에 대한 디렉토리가 디스크 1의 페이지 0에 존재하고, 1로 시작하는 모든 키들에 대한 디렉토리가 디스크 1의 페이지 1에 존재하는 것을 나타낸다. 예제에 대해 C, H, I, N, G와 E를 삽입시킨 후에 분리가 된다. 계속적으로 X, A, M, P와 L를 삽입한 후에 그림 18.13에서 제시된 디스크 기억 공간 구조를 얻는다.(명백히, 비록 실제적으로는 다른 페이지로서 하이브리드 이 되어지지만 디렉토리에 대해 디스크 1을 유지할 수가 있고 각 디스크의 페이지 0가 유지가 되어야 하거나 어떤 다른 방법이 이용된다)

이와 같이 확장 가능 해싱 구조로 삽입은 검색 키들을 포함한 앞 페이지가 접근이 되어진 후 세 가지 연산들중의 하나를 수반할 수가 있다. 만약 앞 페이지에 방이 존재하면, 새로운 레코드는 단순히 삽입을 한다. 그렇지 않으면, 앞 페이지는 두 개로 분리된다.(레코드 반을 새로운 페이지로 이동된다) 만약 디렉토리가 그 앞 페이지를 지칭하는 한 개 엔트리 이상을 지니면, 그때 디렉토리 엔트리는 페이지가 존재하는 것과 같이 분리가 된다. 만약 그렇지 않으면, 디렉토리의 크기는 두배로 된다.

지금까지 기술한 것과 같이 이같은 알고리즘은 나쁜 입력 키 분배에 매우 민감한 것이다. 즉, d 값은 앞 페이지에 맞도록 충분히 적은 집합으로 키들을 분리시키는 것을 요구하는 비트의 가장 큰 수이다. 이와 같이 만약 키들의 큰 수가 앞 비트들의 큰 수에서 동의가 된다면, 디렉토리는 받아들일 수 없는 정도로 큰 것이 된다. 실제 대규모 응용에 대해, 이같은 문제는 앞에 존재하는 비트들을 (유사-)무작위로 만든다. 레코드를 검색하기 위해서, 디렉토리를 접근하는데 이용하는 비트의 순서를 얻기 위해서 키를 해시한다. 즉, 디렉토리는 같은 키로 된 레코드에 대해 검색을 할 것이 어느 페이지인지를 알려준다. 해시의 관점에서, 알고리즘은 해

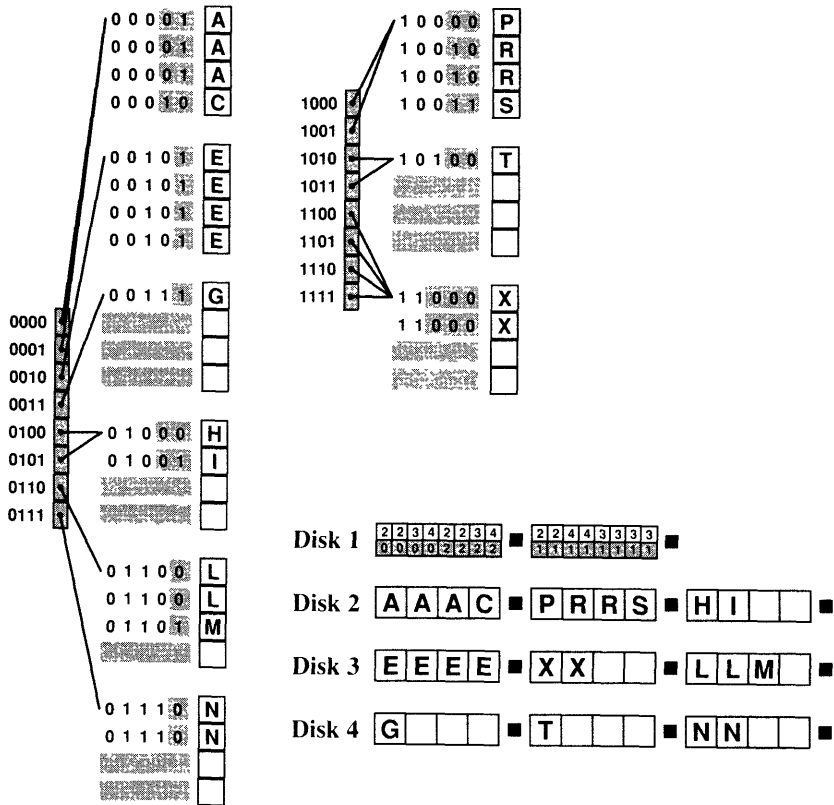


그림 18.13 확장 가능한 해싱 접근

시 값 충돌을 조심스럽게 취급하면서 노드들을 분리시키는 것으로 간주한다. 여기서 이것이 “확장 가능 해싱”이다. 이같은 방법은 색인 순차 검색과 B-트리에 대한 매우 매력적인 대안을 제시한다. 그 이유는 각 검색에 대해 정확히 두 개의 디스크 접근들을(색인 순차 접근과 같은 것) 이용하기 때문이고 반면에 매우 많은 공간을 소비함이 없이도 효율적인 삽입(B-트리와 같은 것)에 대한 능력을 여전히 유지시킨다.

성질 18.4 M 개 레코드를 지닌 페이지로서, 확장 가능 해싱은 N 개 레코드들의 파일에 대해 약 $1.44(N/M)$ 페이지를 요구한다. 디렉토리는 약 $N^{1+1/M}/M$ 엔트리들을 지닌다.

이같은 분석은 이전 장에서 언급된 트라이 분석의 복잡한 확장이다. M 가 클 때, 소비된 공간의 양은 B-트리와 같으나, 적은 M 에 대한 디렉토리는 너무 큰 것을 얻을 수가 있다. □

해상과 더불어서, 특별한 단계들은 똑같은 키들의 큰 수가 현재 존재하는 경우에 취해진다. 그것들은 디렉토리를 인위적으로 크게 만드는 것이다. 그리고 알고리즘은 한 개의 잎 페이지에서 적합한 것이상 똑같은 키들이 존재하는 경우 전적으로 쪼개진다.(이것은 예제에서 5개 E가 지니 것으로서 발생된다) 만약 많은 똑같은 키들이 현재 존재하면(예를 들면), 자료 구조에서 서로 다른 키들로 가정하고 포인터를 잎 페이지들에서 똑같은 키들을 포함하는 레코드들의 연결 리스트에 놓을 수 있다. 수반된 복잡성을 보기 위해서, 마지막 E가 그림 18.13에 있는 구조로 삽입시키기 위해 존재된 경우에 발생이 되는 것을 고려해 보자.

처리하는 덜 재앙적인 상황은 한 개 새로운 키의 삽입은 한 번 이상으로 분리를 시키기 위해 디렉토리를 야기시킨다. 이것은 한 개 이상의 비트가 너무 가득찬 페이지에 키들을 구별하는데 충분하지가 않다. 예를 들면, 만약 $D = 00100$ 의 값을 지닌 두 개 키들이 그림 18.12의 확장 가능 해상 구조로 삽입이 되는 경우, 그때 두 개 디렉토리 분리는 5개 비트가 D와 E를 구별하기 위해서 필요하다.(4번째 비트는 도움이 않된다) 이것은 구현에 대한 대처가 간단하나, 간과해서는 안된다.

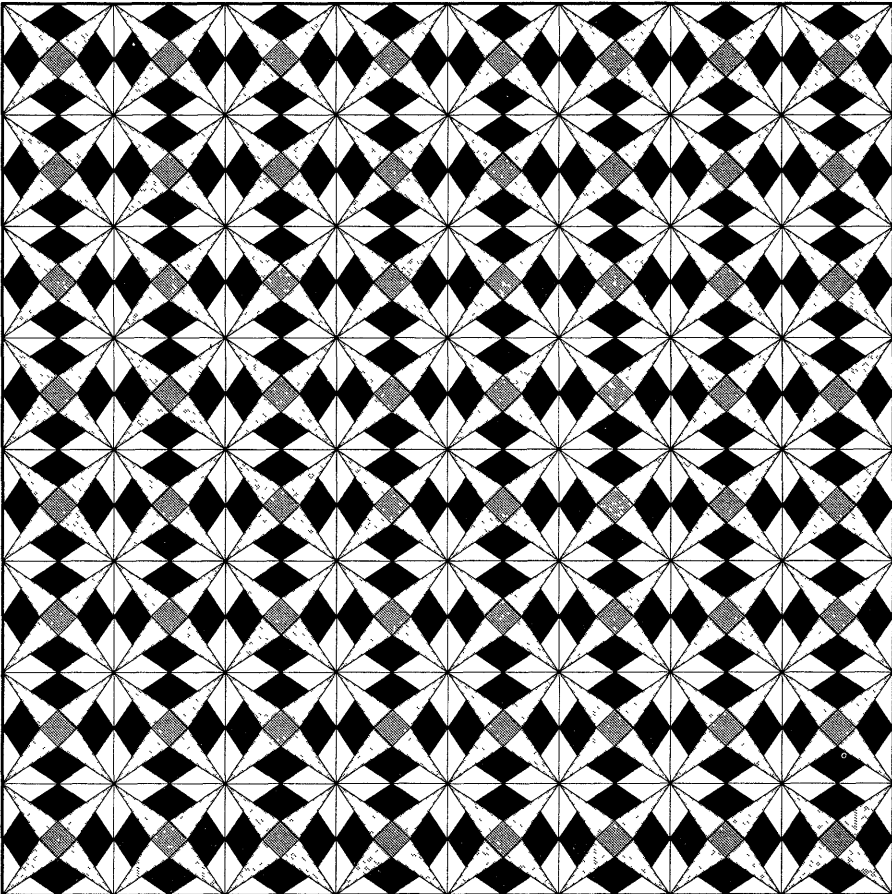
가상 기억장치(Virtual Memory)

외부 정렬에 대한 13장의 끝에서 논의된 “더 쉬운 방법”은 검색 문제에 직접적으로 그리고 사소하게 적용이 된다. 가상 기억장치는 실제적으로는 범용 외부 검색 방법이상 아무것도 없다. 즉, 번지(키)가 주어지면, 그 번지와 관련된 정보를 되돌려 준다. 그러나 가상 기억장치의 직접적인 이용은 쉬운 검색 응용으로서 추천이 되지 않는다. 13장에서 언급한 대로, 가상 기억장치는 대부분의 접근들이 상대적으로 이전 접근들에 근접이 되는 것이다. 정렬 알고리즘은 이것에 적합 한 것이나, 검색의 매우 본질적인 것은 요청들이 데이터 베이스의 임의의 부분에서 정보에 대한 것이다.

연습문제

1. 키 E A S Y Q U E S T I O N을 $M = 5$ 인 처음에 비어있는 트리 그같은 순서로서 삽입이 될 때 결과로 되는 B-트리의 내용은 무엇인가?
2. 키 E A S Y Q U E S T I O N을 $M = 6$ 인 처음에 비어있는 트리 그같은 순서로서 삽입이 될 때 결과로 되는 B-트리의 내용은 무엇인가? 모든 레코드들이 외부 노드들에서 유지가 되어지는 방법들의 변화를 이용하라.
3. 16개의 똑같은 키들이 $M = 5$ 인 초기에 비어있는 트리 삽입을 할 때 생성이 되는 B-트리를 그려라.
4. 데이터 베이스에서 한 페이지가 파괴가 된다고 가정하자. 텍스트에서 기술된 B-트리 구조의 각각에 대해 이같은 사건들을 어떻게 처리하는 가를 기술하시오.
5. 키 E A S Y Q U E S T I O N을 4개 레코드의 페이지 능력으로서 처음에 비어있는 표로 그같은 순서로서 삽입이 될 때 결과로 되는 확장 가능한 해싱의 내용은 무엇인가? (텍스트의 예제를 따르면, 해시가 되는 것이 아니나, i 번째의 문자에 대한 키로서 i 의 5 비트 이진 표현을 이용한다)
6. 3개 레코드의 페이지 능력으로서 초기에 비어있는 표에서 크기 16에 이르기까지 확장 가능한 해싱을 나타내도록 하는 가능한 몇안되는 서로 다른 키들의 순서를 나타내시오.
7. 확장 가능한 해싱 표에서 항목을 제거하는 방법의 개요를 기술하시오.
8. 데이터에 대한 병행성 접근에 대해 “하향식” B-트리보다 좋은 것이 “상향식” B-트리인 이유는 무엇인가?(예를 들면, 두 개 프로그램들이 동시에 새로운 노드를 삽입하려고 한다고 가정하자.)
9. 확장 가능한 해싱 방법을 이용해서 내부 검색에 대한 search와 insert를 구현하시오.
10. 이전 예제의 프로그램에서 내부 검색 응용에 대해 이중 해싱과 기수 트라이 검색을 어떻게 비교하는가를 논하시오.

스트링 처리



빈 면

19 장

스트링 검색

처리되어지는 데이터는 조그만하게 인식되어지는 조각들로 된 독립적인 데이터로서 논리적으로 분해가 되어지지 않는다. 데이터의 이같은 형태는 스트링으로서 기술되어질 수있다는 사실로서만 특성화된다. 즉, 선형적인 문자들의 순서(전형적으로 매우 긴)이다. 물론, 그것들이 C++로서의 기본적인 자료 구조로서 3장과 16장의 예제에 대해 이전의 스트링을 만난다.

스트링은 명백히 워드 처리 시스템에서 중심이 되는 것이고, 이것은 텍스트 처리에 대한 다양한 능력을 제공한다. 그런 시스템들은 텍스트 스트링(text string)을 처리하고, 이것은 연속된 문자, 숫자 그리고 특수 문자들로서 정의된다. 이같은 사물들은 아주 길고(예를 들면, 본 교재에서는 백만자 이상의 문자를 포함한다) 그리고 효율적인 알고리즘은 그것을 처리하는데 중심적인 역할을 한다.

스트링의 또 다른 형태는 단순히 0과 1의 순서로 되어진 이진 스트링(binary string)이다. 이것은 단순히 텍스트 스트링의 특별한 형태라는 의미이나 다른 알고리즘들에 적절하기 때 문과 또한 이진 스트링은 많은 응용에서 자연적으로 발생되기 때문에 명확하게 할 가치가 있다. 예를 들면, 몇 가지 컴퓨터 그래픽 시스템은 이진 스트링으로서 그림들을 나타낸다.(본 교재에서는 그런 시스템들을 인쇄한다. 즉, 현재 페이지는 수백만 비트들로서 구성된 이진 스트링으로서 한 번에 표현된다)

한 의미로서, 텍스트 스트링은 그것들이 큰 알파벳에서 문자들로 구성되기 때문에 이진 스트링과는 아주 다른 사물들이다. 다른 의미로서, 각 텍스트 문자가(말하자면) 8개의 이진 비트들로서 표현되고, 이진 스트링은 문자를 8개 비트 조각으로 취급을 하므로써 텍스트 스트링으로 간주를 하기 때문에 스트링의 두 가지 형태는 똑같다. 스트링을 형성하기 위해 어느

문자가 취급되어지는 가에서 알파벳의 크기는 스트링 처리 알고리즘의 설계에서 중요한 요인이 된다.

스트링상에 근본적인 처리는 패턴 일치(pattern matching)이다. 즉, 길이 N 의 텍스트 스트링과 길이 M 의 패턴이 주어진 경우, 텍스트내에 패턴의 발생을 찾는 것이다.(연속된 0-1 값들이나 스트링의 어떤 다른 특별한 형태를 언급을 할 때라도 항목 “텍스트”를 이용한다) 이 같은 문제에 대한 대부분의 알고리즘은 그것들이 텍스트를 통해 순차적으로 조사되고, 다음의 일치점을 찾기 위해서 일치되는 곳의 시작되는 점 뒤 바로 그 점에서 다시 시작된다.

패턴 일치 문제는 키들과 같이 패턴으로된 검색 문제를 특성화하는데 있다. 그러나 공부한 검색 알고리즘은 패턴이 길고 그리고 알려지지 않은 방법에서 텍스트로 “일직선화 시키는” 것이기 때문에 직접적으로 적용되지를 않는다. 그것은 흥미로운 문제이다. 즉, 여러 가지 매우 다른(그리고 놀라만한) 알고리즘들은 광범위한 유용하고 실제적인 방법들을 제공할 뿐만 아니라 몇 가지 기본적인 알고리즘 설계 기법을 제공하는 것으로 최근에 발견되었다.

간단한 역사

조사되어질 알고리즘들은 흥미로운 역사를 지니므로서, 여러 가지 방법들을 통찰하는데 도움을 주기 위해서 여기서 요약하도록 한다.

널리 이용되는 스트링 처리에 대한 명백한 주먹구구식의 알고리즘이 존재한다. 이것은 최악의 경우 실행시간이 MN 에 비례를 하는 반면에, 많은 응용에서 제기되는 스트링들은 항상 실행시간이 $M + N$ 에 비례를 한다. 더구나, 그것은 대부분의 컴퓨터 시스템에서 좋은 구조적인 특징으로 적합하도록 하므로서 최적화된 버전은 좋은 알고리즘으로 단축되도록 하는 것이 어려운 “표준적”인 것을 제공한다.

1970년에 S. A. Cook는 알고리즘이 최악의 경우 $M + N$ 시간에 비례를하는 패턴 일치 문제를 해결하는 추상적 기계의 특별한 형태에 관해 이론적인 결과를 증명하였다. D. E. Knuth와 V. R. Pratt는 자신의 정리를 증명하기 위해서 이용된 Cook 구성을 통해서 따르고,(실용적인 의도가 전혀 없는 것) 상대적으로 단순하고 실제적인 알고리즘으로 그것을 다듬도록 할 수 있는 알고리즘을 얻었다. 이것은 실제적 응용으로된 이론적인 결과의 예제를 만족시키고 드문 것으로 보인다. 그러나 그것은 J. H. Morris가 텍스트 편집기를 구현할 때 직면한 괴로운 실제적인 문제에 대한 해결책과 같은 알고리즘을 발견한 것이다.(그는 텍스트 스트링에서

뒷받침을 하는 것을 원치 않는다) 그러나 같은 알고리즘은 두 가지 그런 다른 방법들에서 제기된다는 사실은 문제에 대한 근본적인 해로서 신뢰할 수 있도록 한다.

Kunth, Morris와 Pratt는 1976년 까지 그들의 알고리즘을 내놓으려고 하지 않았고, 그 동안에 R. S. Boyer와 J. S. Moore(그리고 R. W. Gosper 혼자)는 그것이 텍스트 스트링에서 문자들의 한 부분만을 조사하기 때문에 많은 응용에서 훨씬 더 빠른 알고리즘을 발견하였다. 많은 텍스트 편집기는 스트링 검색들에 대한 응답의 시간으로서 현저한 감소를 이루기 위해서 이같은 알고리즘을 이용한다.

Knuth-Morris-Pratt와 Boyer-Moore 알고리즘 둘다는 이해하기가 어려운 패턴상에 어떤 복잡한 선처리를 요구하고, 이용의 정도를 제한한다.(사실, 미지의 시스템 프로그래머는 Morris의 알고리즘이 이해하기에는 너무 어렵고 그것을 주먹 구구식 구현으로서 대체된다는 이야기이다)

1980년에 R. M. Karp와 M. O. Rabin는 문제가 제시되어진 것과 같은 정도로 표준적인 검색 문제와는 다른 것이 아니고 항상 실행시간이 $M+N$ 에 비례를하는 주먹 구구식의 알고리즘과 같은 정도로 간단한 알고리즘을 제안하였다. 더구나 그것들의 알고리즘은 2차원 패턴과 텍스트로 쉽게 확장되고, 그것은 그림 처리에 대해 다른 것들보다 더 유용하게 만드는 것이다.

이같은 이야기는 “더 좋은 알고리즘”에 대한 검색이 여전히 가끔 정당화가 되어짐을 제시한다. 정말로 이같은 문제에 대한 것조차 수평적인 것에서 여전히 더 많은 개발이 존재하는 것은 의심을 해 볼만하다.

주먹구구식 알고리즘

즉각적으로 떠오르는 패턴에 대한 명백한 방법은 패턴과 일치되는 텍스트에서 각각 가능한 위치에 대해 그것이 실제 일치하는지를 체크하는 것이다. 다음 프로그램은 텍스트 스트링 a에서 패턴 스트링 p의 첫 번째 발생에 대해 이같은 방법으로서 검색을 한다.

```
int brutesearch(char *p, char *a)
{
    int i, j, M = strlen(p), N = strlen(a);
    for ( i = 0; j = 0; j < M && i < N; i++, j++ )
        if ( a[i] != p[j] ) { i -= j - 1; j = -1; }
    if ( j == M ) return i-M; else return i;
}
```


프로그램은 한 포인터(i)를 텍스트로 유지를 하고, 다른 포인터(j)를 패턴으로 유지한다. 그것들이 문자들에 일치되는 것으로 지적이 되는 한, 양쪽 포인터는 증가된다. 만약 i와 j가 문자들과 일치되지 않으면, 그때 j는 패턴의 시작을 가르키도록 다시 세트를 하고, i는 텍스트에 대해 일치되는 것에 대해 패턴을 오른쪽으로 한 위치 이동시키는것에 대응되도록 다시 세트를 한다. 특히, if 문장은 j를 -1로 세트를 하면, 그때 for 반복문의 연속적인 반복은 첫 번째 패턴 문자가 일치되는 텍스트 문자를 만날때까지 i를 증가시킨다.

만약 패턴의 끝에 도달되면(j == M), 그때 a[i-M]에서 시작되는 일치가 있다. 그렇지 않으면, 텍스트 끝에 도달 되기전(i == M)에 도달되면, 그때 일치되는 것은 없다. 즉, 패턴은 텍스트에는 나타나지 않고, “표지” 값으로서 N가 되돌려준다.

텍스트 편집 응용에서, 이 프로그램의 내부 반복은 좀처럼 반복되어지지 않고 실행시간은 조사된 텍스트 문자의 수에 거의 비례를 한다. 예를 들면, 다음 텍스트 스트링에서 패턴 STING를 찾는다고 가정을 하자.

A STRING SEARCHING EXAMPLE CONSISTING OF

그때 문장 j++는 실제 일치가 되기전에 단지 4번 실행된다.(각 S에 대해 한 번 그러나 첫 번째 ST에 대해 2번)

다른 한편, 주먹구구식 검색은 어떤 패턴에 대해서는 매우 느리다. 예를 들면, 텍스트가 이진수(두개의 문자)인 경우는 그림처리와 시스템 프로그래밍 응용에서 발생하는 것과 같다.

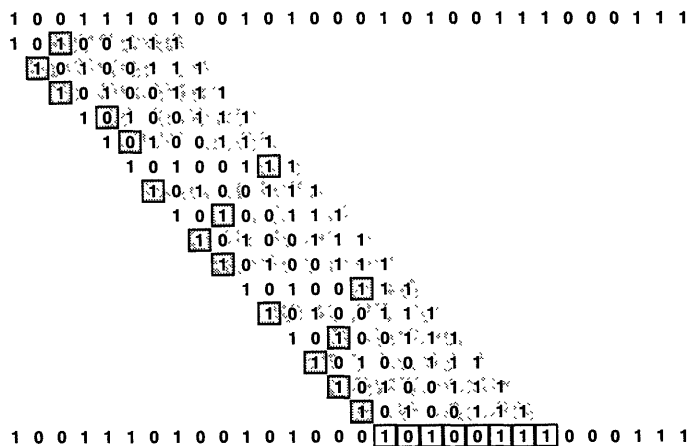


그림 19.1 이진 텍스트에서 주먹구구식 스트링 검색

그림 19.1은 알고리즘이 긴 이진 텍스트에서 패턴 10100111에 대해 검색을 하는데 이용될 때 무엇이 발생되는가를 나타낸다. 각 라인(일치된 것인 마지막은 제외하고)은 패턴을 따르는 하나의 불일치와 일치되는 0개나 그 이상의 문자들로 구성된다. 이것들은 패턴을 찾으려고 노력을 할 때 발생하는 “거짓 시작(false start)”이다. 즉, 알고리즘 설계에서 명백한 목적은 숫자들과 이것들의 길이를 제한시키는 것이다. 이같은 예제에 대해, 패턴에서 약 두 개의 문자들은 비록 상황이 최악의 경우라 할지라도 각 텍스트 위치에 대해서 평균적으로 조사된다.

성질 19.1 주먹구구식의 스트링 검색은 약 NM 문자 비교를 요구한다.

최악의 경우는 패턴과 텍스트 둘다가 1으로 따르는 모두가 0일때이다. 그때 $N - M + 1$ 개 가능한 일치 위치의 각각에 대해, 패턴에서의 모든 문자들은 전체 비용이 $M(N - M + 1)$ 인 텍스트에 대해 체크한다. 정상적으로 M 는 N 에 비례된 매우 적은 것이므로해서, 전체는 NM 된다. □

그런 퇴보된 스트링들은 영어(또는 C++) 텍스트에서 아마도 비슷하지 않으나 그러나 그것들은 이진 텍스트들이 처리될 때 발생이 잘 되고 그래서 더 좋은 알고리즘을 찾을 수가 있다.

Knuth-Morris-Pratt 알고리즘

Knuth, Morris와 Pratt에 의해 개발된 알고리즘뒤에 숨겨진 개념은 다음과 같다. 즉, 일치되지 않은 것이 감지될 때, “거짓 시작”은 이미 알고 있는 문자들로서(그것들이 패턴내에 존재하므로서) 구성된다. 어디선가 이같은 모든 알려진 문자들에 대해, i 포인터를 보충하는 대신에 이같은 정보의 잊점을 이용할 수가 있다.

이것의 간단한 예제에 대해, 패턴에서의 첫 번째 문자는 패턴에서 다시 나타나지 않는다고 가정하자.(말하자면 패턴은 10000000이다) 그때 텍스트에서 어떤 위치에 거짓 시작 j 문자의 길이를 지닌다. 일치되지 않는 것을 감지했을 때, j 문자들이 일치된다는 사실에 의해서 텍스트 포인터 i 를 “보충”할 필요는 없다. 그 이유는 텍스트에서 이전 $j-1$ 문자들중 어느것도 패턴에서 첫 번째 문자와 일치할 수가 있기 때문이다. 이같은 변화는 위의 프로그램에서 $i -= j-1$ 를 $i++$ 로 대체시키므로써 구현된다. 이같은 변화의 실제적인 효과는 그런 특별한 패턴이 특별히 발생하는 것 같지는 않지만 그러나 개념은 생각해 볼 가치가 있고, Knuth-Morris-

j	next[j]	
1	0	10100111 10100111
2	0	10100111 10100111
3	1	10100111 10100111
4	2	10100111 10100111
5	0	10100111 10100111
6	1	10100111 10100111
7	1	10100111 10100111

그림 19.2 Knuth-Morris-Pratt 검색에 대한 다시 시작되는 위치

-Pratt 알고리즘은 그것의 일반화이기 때문에 제한된다. 놀랍게도, 어떤 것을 배열하는 것은 항상 가능하므로 그 결과 i 포인터는 결코 감소가 되지 않는다.

이전 절에서 기술된 것과 같이 불 일치를 감지함에 패턴을 지나 완전히 넘어가는 것은 패턴이 불일치의 시점에서 그 자체와 일치되어 있을 때 작동을 하지 않는다. 예를 들면, 1010100111에서 10100111을 검색할 때 먼저 5번째 문자에서 불일치되는 것을 감지하나 그러나 검색을 계속하기 위해서 세 번째 문자에 대해 보충을 하는 것이 더 낫다. 그 이유는 그렇지 않으면, 일치가 되지 않기 때문이다. 그러나 그림 19.2에서 제시된 바와 같이 단지 패턴에서만 의존이 되기 때문에 정확히 무엇을 해야하는 지를 미리 이해를 해야한다.

배열 $next[M]$ 는 불 일치를 감지 했을 때 어떻게 보충을 할 것인가를 결정하는데 이용된다. 패턴의 두 번째 문자에 대한 복사의 첫 번째 문자로서 시작해서 그리고 모든 겹치는 문자들이 일치될때(또는 존재하지 않을 때) 멈추므로서 왼쪽에서 오른쪽으로 그 자체에 대해 패턴의 첫 번째 j 문자의 복사로 돌아가게 하는 것을 상상하라. 이같은 겹친 문자들은 불 일치 $p[j]$ 에서 감지될때에 패턴이 일치될 가능성이 있는 다음번째 위치를 정의한다. 패턴에서($next[j]$) 보충하기 위한 거리는 겹치는 문자들의 수와 정확히 같다. 특별히 $j > 0$ 에 대해, $next[j]$ 의 값은 패턴의 첫 번째 k 문자들이 패턴의 첫 번째 j 문자들의 마지막 k 문자들과 일치를 하는 것에 대한 최대치 $k < j$ 이다. 곧 보게 될것과 같이, $next[0]$ 를 -1로 정의하는 것이 편리하다.

이같은 $next$ 배열은 위에서 논의 한 대로 텍스트 포인터 i 의 “보충”을 제한하는(사실 보는바와 같이 제거가 된다) 방법을 주는 것이다. i 와 j 가 불일치되는 문자들을 지적할때,(텍스

트 스트링에서 위치 $i-j+1$ 에서 시작되는 패턴 일치에 대한 테스트) 그때 패턴 일치에 대한 다음번 가능한 위치는 위치 $i - \text{next}[j]$ 에서 시작되는 것이다. 그러나 next 표의 정의에서, 그 위치에서 첫 번째 $\text{next}[j]$ 문자들은 패턴의 첫 번째 $\text{next}[j]$ 문자들과 일치를 한다. 그래서 i 포인터를 보충할 필요는 없다. 즉, 다음 프로그램에서와 같이 단순히 i 포인터를 변경시키지 않도록 하고 j 포인터를 $\text{next}[j]$ 로 세트를 한다.

```
int kmpsearch(char *p, char *a)
{
    int i, j, M = strlen(p), N = strlen(a);
    initnext(p);
    for (i = 0, j = 0; j < M && i < N; i++, j++)
        while ((j >= 0) && (a[i] != p[j])) j = next[j];
    if (j == M) return i-M; else return i;
}
```

$j = 0$ 이고 $a[i]$ 가 $p[0]$ 와 일치를 않을 때, 겹치는 것은 없다. 그래서 i 를 증가시키고, j 를 패턴의 시작에서 세트시키도록 유지하는 것을 원한다. 이것은 $\text{next}[0]$ 를 -1 로 정의를 함으로서 되고 결과는 while 반복에서 j 가 -1 로 세트된다. 그때 i 는 증가되고 j 는 for 반복이 되는 것과 같이 0 으로 세트된다. 기능적으로 이 프로그램은 bruteforce와 같으나 스스로 반복적인 패턴에 대해 보다 빠르게 수행되는 것이다.

이것은 next 표를 계산하기 위해서 남아있게 된다. 이것에 대한 프로그램은 간단하다. 즉, 위에서와 같은 프로그램이 기본적인 그 자체에 대해서 패턴은 일치를 한다.

```
initnext(char *p)
{
    int i, j, M = strlen(p);
    next[0] = -1;
    for (i = 0, j = -1; i < M; i++, j++, next[i] = j)
        while ((j >= 0) && (p[i] != p[j])) j = next[j];
}
```

i 와 j 가 증가 되어진 후에, 패턴의 첫 번째 j 문자들은 위치 $p[i-j-1], \dots, p[i-1]$ 에서 패턴의 첫 번째 i 문자에서 마지막 j 문자와 일치된다. 그리고 이것은 이같은

속성을 지닌 가장 큰 j 이다. 그 이유는 그렇지 않으면, 그 자체와 패턴의 “가능한 일치”는 없어진다. 이와 같이 j 는 `next[i]`에 할당된 값이다.

이같은 알고리즘을 보는 흥미로운 방법은 패턴을 고정된 것으로 고려하므로서 `next` 표가 프로그램에 “선으로 연결(wired-in)” 된다. 예를 들면, 다음 프로그램은 고려된 패턴에 대해 위의 프로그램과 정확히 일치된다. 그러나 그것은 훨씬 더 효율적인 것이다.

```
int kmpsearch(char *a)
{
    int i = -1;
sm: i++;
s0: if ( a[i] != '1' ) goto sm; i++;
s1: if ( a[i] != '0' ) goto sm; i++;
s2: if ( a[i] != '1' ) goto sm; i++;
s3: if ( a[i] != '0' ) goto sm; i++;
s4: if ( a[i] != '0' ) goto sm; i++;
s5: if ( a[i] != '1' ) goto sm; i++;
s6: if ( a[i] != '1' ) goto sm; i++;
s7: if ( a[i] != '1' ) goto sm; i++;
    return i-8;
}
```

이 프로그램에서 `goto` 라벨은 `next` 표에 대응된다. 사실, `next` 표를 계산하는 위의 `initnext` 프로그램은 이같은 프로그램을 출력하기 위해서 쉽게 변경된다. 매번 i 가 증가되므로 해서 $i == N$ 인지를 체크하는 것을 피하기 위해서, 패턴 그 자체는 $a[N], \dots, a[N+M-1]$ 에서 표지 값으로 텍스트의 끝에서 저장된다고 가정하자. (이같은 개선은 또한 표준적인 구현에서 이용되어진다) 이것은 “스트링 검색 컴파일러”의 간단한 예제이다. 즉, 패턴이 주어진 경우, 임의적으로 긴 텍스트 스트링에서 그 패턴에 대해 주사(scan)하기 위해서 매우 효율적인 프로그램을 생성할 수가 있다. 다음 두 개의 장에서 이같은 개념의 일반성에 대해 보기로 하자.

위의 프로그램은 스트링 검색 문제를 해결하기 위한 몇 가지 기본적인 연산들을 이용하게 된다. 이것은 유한 상태 기계(finite-state machine)라 부르는 매우 단순한 기계에 의해서 쉽게 기술된다. 그림 19.3은 위의 프로그램에 대한 유한 상태 기계를 제시한 것이다.

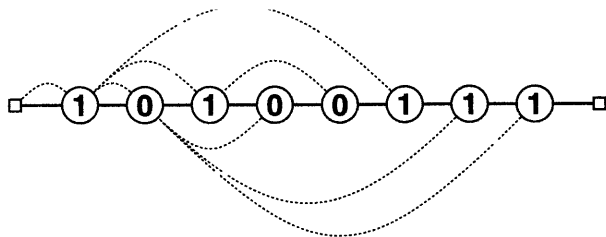


그림 19.3 Knuth-Morris-Pratt 알고리즘에 대한 유한 상태 기계

기계는 상태들(state: 원에 숫자로 표시)과 전이(transition: 선으로 표시)로 구성된다. 각 상태는 두 개의 전이들을 지닌다. 즉, 일치된 전이(굵은 선, 오른쪽으로 가는 것)과 불 일치된 전이(점선, 왼쪽으로 가는 것)이다. 상태들은 기계가 어디서 명령을 실행하는 지를 나타낸다. 즉, 전이들은 goto 명령들이다. 상태에서 라벨이 “x”일 때, 기계는 단지 한 개의 명령을 실행한다. 만약 현재 문자가 x이면, 그때 그것을 지나서 주사를 하고 그리고 일치된 전이를 취한다. 그렇지 않으면, 불 일치 전이를 취한다. 문자를 “지난 것을 주사”하는 것은 “현재 문자”로서 스트링에서 다음 문자를 취하는 것을 의미한다. 즉, 기계는 그것들과 일치하는 것으로서 문자를 지나간 것을 주사한다. 이것에는 두 가지 예외가 있다. 즉, 첫 번째 상태는 항상 일치 전이이고 다음 문자에 대해 주사를 하고(본질적으로 이것은 패턴에서 첫 번째 문자의 첫 번째 연속 발생에 대해 주사하는 것에 대응된다) 그리고 마지막 상태는 일치가 되어진 것을 나타내는 “중지” 상태이다. 다음장에서, 훨씬 더 강력한 패턴 일치 알고리즘을 개발하도록 도와주기 위해서 비슷한(그러나 더 강력한) 기계를 어떻게 이용하는 가를 보게 될 것이다.

현명한 여러분은 이 알고리즘에서 여전히 개선의 여지가 있음을 알게 될 것이다. 그 이유는 불 일치를 야기시키는 문자는 고려하지 않기 때문이다. 예를 들면, 텍스트가 1011로 시작하고, 예제 패턴 10100111에 대해 검색을 한다고 가정하자. 101이 일치된 후에, 4번째 문자에서 불 일치가 됨을 알게 된다. 이 점에서 next 표는 텍스트의 4번째 문자에 대해 패턴의 두 번째 문자를 조사한다고 말할 수가 있다. 그 이유는 101 일치의 기본에서, 패턴의 첫 번째 문자는 텍스트의 세 번째 문자와 일치선으로 된다.(그러나 그것들 둘다가 1임을 알기 때문에 이것들을 비교하지 않아도 된다) 그러나 여기서 일치하는 것은 없다. 즉, 불 일치로부터, 텍스트에서 다음 문자는 패턴에서 요구한 것과 같이 0이 아님을 알게 된다. 이것을 보기 위한 다른 방법은 “선으로 연결된” 다음 표로서 프로그램의 버전을 보는 것이다. 즉, 라벨 4에서 $a[i]$ 가 0이 아닌 경우 2로 가나, 라벨 2에서 $a[i]$ 가 0이 아니면 1로 가게 된다. 왜 직접적으로 1로 가지 못하는가? 그림 19.4는 예제에 대해 유한 상태 기계의 개선된 버전을 제시한다.

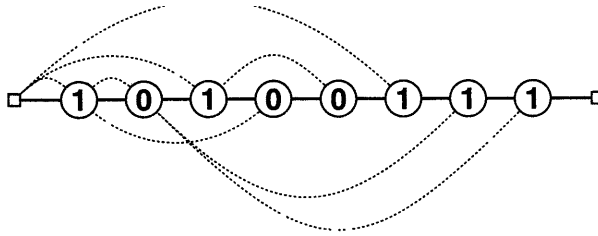


그림 19.4 Knuth-Morris-Pratt 유한 상태 기계(계산된 것)

다행히도 이같은 변화를 알고리즘에 넣는 것은 쉽다. 다음으로 `initnext` 프로그램에서 문장 `next[i] = j`를 대체하기만 하면 된다.

```
next[i] = ( p[i] == p[j] ) ? next[j] : j
```

왼쪽에서 오른쪽으로 처리되므로, 필요로 하는 next의 값은 이미 계산이 되어지므로 해서 그것을 이용하기만 하면 된다.

성질 19.2 Knuth-Morris-Pratt 스트링 검색은 $M + N$ 문자 비교 이상을 결코 이용하지 못한다.

이같은 성질은 그림 19.5에 제시되고 코드에서 또한 명백하다. 즉, 각 i 에 대해 기껏해야 한 번 next 표에서 그것을 다시 세트시키거나 혹은 j 를 증가시키는 것중의 하나이다. \square

그림 19.5는 이같은 방법이 이진 예제에 대해 주먹구구식 방법보다 훨씬 더 적은 비교를 이용함이 확실하다. 그러나 Knuth-Morris-Pratt 알고리즘은 많은 실제적인 응용에서 주먹구구식 방법보다 의미적으로 더 빠르다는 것은 아니다. 그 이유는 단지 몇 가지 응용만이 고도의 스스로 반복되는 텍스트에서 고도의 스스로 반복되는 패턴에 대해 검색하는 것을 수반한

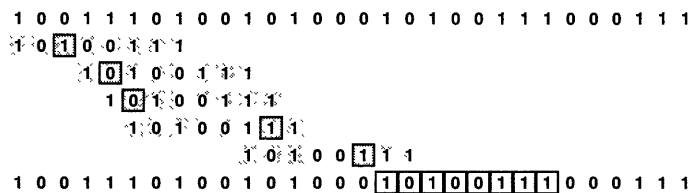


그림 19.5 이진 텍스트에서 Knuth-Morris-Pratt 스트링 검색

다. 그러나 방법은 중요한 실제적인 장점을 지닌다. 즉, 그것은 입력을 통해서 순차적으로 처리되고 입력에서 결코 “보충”되지 않는다. 이것은 어떤 외부 장치에서 읽어들이는 더 큰 파일에 이용하는 것이 편리하도록 하는 것이다.(알고리즘은 이같은 상황에서 어떤 복잡한 버퍼링을 요구한다)

Boyer-Moore 알고리즘

만약 “보충하는 것”이 어렵지 않다면, 의미적으로 더 빠른 스트링 검색 방법은 텍스트에 대해 그것과 일치되도록 할 때 오른쪽에서 왼쪽으로 패턴을 조사함으로써 개발된다. 표본 패턴 10100111에 대해 검색을 할 때, 만약 5번째 상에는 없지만 8번째, 7번째와 6번째 문자상에서 일치되는 가를 찾는 경우, 그때 패턴을 오른쪽에서 7번째 위치로 옮기고 그리고 다음에 15번째 문자를 체크한다. 그 이유는 부분적인 일치가 111을 찾기 때문이다. 그리고 이것은 패턴의 어디인가에 나타나게 된다. 물론 끝에 있는 패턴은 일반적으로 어디에서간 나타나므로서 위에서 처럼 next 표가 필요하다.

패턴 10110101에 대한 next 표의 오른쪽에서 왼쪽 버전은 그림 19.6에 제시되어있다. 즉, 이경우에서 next[j]는 오른쪽에서 왼쪽의 주사에서 불일치가 패턴에서 오른쪽에서 j번째 문자상에 발생이 되는 것으로 주어지면, 패턴은 오른쪽으로 이동되는 문자 위치들의 수이다. 이것은 패턴의 마지막 문자로서 일직선에 놓여진 복사의 다음에서 마지막에 이르는 문자로서 시작을 해서 모든 겹치는 문자들이 일치될 때(또한 불일치를 야기시키는 문자를 고려한

j	next[j]	
2	4	<pre> 10110101 10110101 </pre>
3	7	<pre> 10110101 10110101 </pre>
4	2	<pre> 10110101 10110101 </pre>
5	5	<pre> 10110101 10110101 </pre>
6	5	<pre> 10110101 10110101 </pre>
7	5	<pre> 10110101 10110101 </pre>
8	5	<pre> 10110101 10110101 </pre>

그림 19.6 Boyer-Moore 검색에 대한 다시 시작되는 위치

다) 멈추게 하므로서 왼쪽에서 오른쪽으로 그 자체의 마지막 j 문자들에 대한 패턴의 복사를 하게 하므로서 이전과 같이 찾게 된다. 예를 들면, `next[2]`는 7이 된다. 그 이유는 만약 마지막에 두 개의 문자가 일치하고 오른쪽에서 왼쪽으로 주사에서 불 일치되면, 그때 001은 텍스트에서 만나게 된다. 즉, 이것은 1이 패턴에서 첫 번째 문자로 일치선으로 이루는 경우를 제외하고 패턴에서 나타나지 않는다. 그래서 오른쪽으로 7개 위치를 옮긴다.

이것은 Knuth-Morris-Pratt 방법의 위의 구현과 아주 비슷한 프로그램이다. 많은 경우에서 더 좋은 것인 오른쪽에서 왼쪽으로의 패턴 주사로서 문자들을 넘기는 아주 다른 방법들이 존재하기 때문에 이것을 더 자세하게 조사할 필요는 없다.

개념은 패턴 뿐만 아니라 텍스트에서 불일치를 야기시키는 문자의 기본에서 다음에 무엇을 해야하는 지를 결정하는 것이다. 선 처리(preprocessing) 단계는 텍스트에서 발생하는 각각 가능한 문자에 대해 그 문자가 불일치를 야기 시키는 경우에 무엇을 할 것인지를 결정하는 것이다. 이것의 가장 간단한 인식은 아주 유용한 프로그램으로 직접적으로 이끌어지게 된다.

그림 19.7은 첫 번째 표본 텍스트상에서 이 방법을 제시한 것이다. 패턴과 일치하도록 오른쪽에서 왼쪽으로 처리를 하므로서, 텍스트에서 R(5번째 문자)에 대해 패턴에서 먼저 G를 체크하는 것이다. 이것들이 일치되지 않을 뿐만 아니라 R은 패턴에서 어느곳에서도 존재하지 않으므로 해서 R을 지나는 모든 방법들로 그것을 옮긴다. 다음 비교는 R뒤인 5번째 문자에 대해 패턴에서 G의 것이다.(SEARCHING에서 S) 여기서 그것의 S가 텍스트에서 S와 일치할 때까지 오른쪽으로 패턴을 옮긴다. 그때 패턴에서 G는 패턴에서 나타나지 않은 SEARCHING에서의 C에 대한 비교를 하는 것이므로 해서 패턴은 오른쪽으로 5개 더 떨어진 위치로 옮긴다. 3번더 5개 문자들을 넘긴후에, CONSISTING에서 T에 도달이 되고 이점은 그것의 T가 텍스트에서 T와 일치되고 그리고 완전한 일치를 찾게 된다. 이 방법은 텍스트에서 단지 7개의 문자들을 조사하는 비용으로서 일치되는 위치로 가져가게 된다.(그리고 일치를 검증하기 위해서 5개 더)!

```

A  STRING SEARCHING EXAMPLE CONSISTING OF ...
STIN[G]
   STIN[G]
      STIN[G]
         STIN[G]
            STIN[G]
               STIN[G]
                  STIN[G]
                     STIN[G]
                        STIN[G]
                           STIN[G]
                              STIN[G]
                                 STIN[G]
A  STRING SEARCHING EXAMPLE CONSI[STING] OF ...

```

그림 19.7 불일치된 문자 휴리스틱을 이용한 Boyer-Moore 스트링 검색

이같은 “불 일치 문자” 알고리즘을 구현하기는 아주 쉽다. 알파벳에서 각 문자에 대해 텍스트에서 그 문자들이 발생하는 경우에 얼마 만큼 넘어가도록 하는지와 스트링 검색동안에 불 일치를 야기시키는 것을 말해주는 배열 skip를 초기시키기 위해서 주먹구구식으로 오른쪽에서 왼쪽 패턴 주사를 단순히 개선하는 것이다. 텍스트에서 발생이 될 가능성이 있는 각 문자에 대해서 skip에서의 엔트리이어야만 한다. 즉, 간단하게 하기 위해 char를 인수로서 취하고, 알파벳의 i 번째 글자에 대해서는 i 를 그리고 빈칸에 대해서는 0을 되돌려주는 index함수를 지닌다고 가정하자. 또한 패턴에는 없는 문자에 대해 M에 skip 배열을 초기화하고, 0에서 M-1에 이르는 j 에 대해 skip[index(p[j])]를 M-j-1로 세트시키는 서브루틴 initskip()를 가정하자. 그때 구현은 간단하다.

```
int mischsearch(char *p, char *a)
{
    int i, j, t, M = strlen(p), N = strlen(a);
    initskip(p);
    for ( i = M-1, j = M-1; j > 0; i--, j-- )
        while ( a[i] != p[j] )
        {
            t = skip[index(a[i])];
            i += ( M-j > t ) ? M-j : t;
            if ( i >= N ) return N;
            j = M-1;
        }
    return i;
}
```

만약 skip 표 모두가 0인 경우,(그것은 결코 존재 하지 않는다) 이것은 주먹구구식 방법의 오른쪽에서 왼쪽의 버전에 대응된다. 그 이유로서 문 $i += M-j$ 는 i 으로 하여금 텍스트 스트링에서 다음번 위치로 다시 세트시키는 것이기 때문이다. 그때 $j = M - 1$ 은 오른쪽에서 왼쪽으로의 문자단위의 일치에 대해 준비를 하도록 패턴 포인터를 다시 세트시킨다. 방금 논의한대로, skip표는 보증된 텍스트 즉, 한 번에 M개의 문자들로서 멀리 가로지르므로서 이동이 되는 패턴으로 이끌어진다.(패턴에 제시되지 않은 텍스트 문자들은 만날 때) 패턴 STING에 대해, G에 대한 skip 엔트리는 0, N에 대한 엔트리는 1, I에 대한 엔트리는 2, T에 대한 엔트리는 3, S에 대한 엔트리는 4 그리고 모든 다른 문자들에 대한 엔트리는 5가 된다. 예를

들면, S 가 오른쪽에서 왼쪽으로의 검색동안에 만나면, i 포인터는 4로서 증가가 되므로서 패턴의 끝은 S 의 오른쪽에서 네번째 위치에 조정된다.(그리고 계속적으로 패턴에서 S 가 텍스트에서 S 와 일치선상에 놓이게 된다) 만약 패턴에서 한개 S 이상이 존재하면, 이같은 계산에 대해 가장 오른쪽에 있는 것을 이용하도록 한다. 여기서 skip 배열은 왼쪽에서 오른쪽으로의 주사를 통해 생성된다.

Boyer와 Moore는 요청이 되어진 두 번의 넘어간 중에 더 큰 것을 고르므로서 오른쪽에서 왼쪽으로의 패턴 주사에 대한 개요에 대해서 두 가지 방법을 결합하는 것으로 제시된다.

성질 19.3 Boyer-Moore 스트링 검색은 $M + N$ 문자 비교이상을 결코 이용을 하지 않고, 알파벳이 적지 않고 그리고 패턴이 길지 않은 경우 약 N/M 단계를 이용한다.

알고리즘은 Kunth-Morris-Pratt 방법과 같은 방법으로서 최악의 경우 선형적이다.(두개의 Boyer-Moore 휴리스틱중의 하나만을 수행하는 위의 주어진 구현은 선형이 아니다) “평균의 경우” 결과 N/M 은 여러 가지 무작위 스트링 모델에 대해서 증명이 되었으나 그러나 이같은 경향들은 비 현실적이다. 그래서 상세한 것들은 그냥 넘어가자. 많은 실제적인 상황에서, 단지 몇 가지 알파벳 문자들이 패턴의 어느 곳에서도 나타나지 않는다는 것은 사실이고 그래서 각 비교는 그냥 넘어간 M 개 문자들로 이끌어지고 그리고 이것은 지시된 결과로 주어진다. □

불 일치된 문자 알고리즘은 이진 스트링에 대해 많이 도움이 되지 않는다. 그 이유는 불 일치를 야기시키는 문자들에 대해 단지 두 가지 가능성이 존재하기 때문이다.(그리고 이것들은 패턴에서 존재하는 것들이다) 그러나 비트들은 위에서 처럼 정확히 이용된 “문자”들로 만들도록 함께 그룹화할 수가 있다. 만약 한 번에 b 비트를 취하면, 그때 2^b 엔트리들로 된 skip 표가 필요하다. b 의 값은 충분히 적게 선택되어져서 이같은 표는 너무 크지가 않으나 그러나 충분히 큰 것으로서 텍스트의 대부분의 b -비트 절들은 패턴에서 존재하지 않는 것이다. 특별히, 패턴에서 $M - b + 1$ 개의 다른 b -비트 절들이 존재를 하고($M - b + 1$ 을 통해서 1에서 시작한 각 비트 위치에서 시작되는 것) 그래서 의미적으로 2^b 보다 적은 것인 $M - b + 1$ 을 원한다. 예를 들면, b 를 약 $\lg(4M)$ 으로 하도록 하면, 그때 skip 표는 M 엔트리로 채워진 것이 3/4이상 존재한다. 또한 b 는 $M/2$ 이하로 존재케 된다. 그 이유는 그렇지 않으면, 두 개의 b -비트 텍스트 절들 사이에서 분리된 경우에 전적으로 패턴을 놓치게 된다.

Rabin-Karp 알고리즘

위에서 조사하지 않은 스트링 검색에 대한 주먹구구식 방법은 텍스트의 각각 가능한 M -문자 절을 표준적인 해시 표에서 키로서 취급을 하므로써 큰 기억장치를 추출하도록 한다. 그러나 문제는 단지 한 개 키를 찾도록 하기 위해 설정되기 때문에 전체 해시 표를 유지시키는 것이 필요하지 않다. 즉, 수행될 필요가 있는 모든 것들은 텍스트의 가능한 M -문자 절들의 각각에 대해 해시 함수를 계산하는 것이고, 그것은 패턴의 해시 함수와 같은 경우인지를 체크하는 것이다. 이같은 방법의 문제는 그것들이 패턴과 일치하는 지를 단순히 체크함으로써 텍스트에서 M 개 문자들에 대한 해시 함수를 계산하는 것은 어려운 것이 된다. Rabin과 Karp는 16장에서 이용된 해시 함수에 대한 이같은 어려움에 대해 쉬운 방법을 찾은 것이다. 즉, $h(k) = k \bmod q$ 이다. 여기서 q (표 크기)는 큰 소수이다. 이 경우에서, 어느것도 해시 표에 저장되지 않으므로써 q 는 매우 큰 것으로 취급된다.

방법은 위치 $i-1$ 에 대한 그것의 값들이 주어진 텍스트에서 위치 i 에 대한 해시 함수를 계산하는 것을 기본으로 하고 그리고 수학적인 공식에서 직접적으로 따른다. 그것들을 컴퓨터 워드로서(정수로서 취급이 되어지는) 함께 팩(pack)화 시키므로써 M 개 문자들을 숫자들로써 번역을 한다고 하자. 이것은 기저가 d 인 진법의 숫자로서 문자들을 기술하는 것에 대응된다. 여기서 d 는 가능한 문자들의 수이다. 이와 같이 $a[i] \dots a[i + M - 1]$ 에 대응되는 숫자는 다음과 같다

$$x = a[i]d^{M-1} + a[i+1]d^{M-2} + \dots + a[i+M-1]$$

그리고 $h(x) = x \bmod q$ 의 값을 안다고 가정하자. 그러나 텍스트에서 오른쪽으로의 한자리 이동은 단순히 x 를 다음과 같이 대체하므로써 대응된다.

$$(x - a[i]d^{M-1})d + a[i+M]$$

mod 연산자의 기본적인 성질은 각 산술 연산 뒤에 q 로써 나누어진 나머지를 취하는 경우 (적은 것으로 취급이 되어진 숫자들을 유지하기 위해서), 그때 산술 연산의 모든 것들을 수행하는 경우 즉, q 로 나누어진 나머지를 취하는 것과 같은 값을 얻을 수가 있다.

이것은 아래에서 구현된 매우 간단한 패턴-일치 알고리즘으로 된다. 프로그램은 위에서 처럼 같은 index 함수로 가정이 되나 그러나 $d = 32$ 는 효율성을 위해 이용된다.(곱셈은 이동으로 구현이 된다)

```

const int q = 33554393;
const int d = 32;
int rksearch(char *p, char *a)
{
    int i, dM = 1, h1 = 0, h2 = 0;
    int M = strlen(p), N = strlen(a);
    for ( i = 1; i < M; i++ ) dM = ( d*dM ) % q;
    for ( i = 0; i < M; i++ )
    {
        h1 = ( h1 * d + index(p[i]) ) % q;
        h2 = ( h2 * d + index(a[i]) ) % q;
    }
    for ( i = 0; h1 != h2; i++ )
    {
        h2 = ( h2 + d * q - index(a[i]) *dM ) % q;
        h2 = ( h2 * d + index(a[i+M]) ) % q;
        if ( i > N-M ) return N;
    }
    return i;
}

```

먼저 프로그램은 패턴에 대한 해시 값 $h1$ 과 텍스트의 첫 번째 M 문자들에 대한 해시 값 $h2$ 를 계산한다.(그것은 또한 변수 dM 에서 $d^{M-1} \bmod q$ 의 값을 계산한다) 그때 그것은 각 i 에 대해 위치 i 에서 시작하고, 각 새로운 해시 값을 $h1$ 과 비교하는 M 개 문자들에 대해 해시 함수를 계산하기 위해서는 위의 기법을 이용하므로써 텍스트 스트링을 통해서 처리된다. 소수 q 는 가능한 큰 것으로 선택을 하나 그러나 충분히 적은 것으로서 $(d+1) * q$ 는 과잉넘침(overflow)을 발생하지 않는다.(특별한 $d*q$ 는 $\%$ 연산자가 이전대로 수행되도록 하기 위해서 모든 것들이 양수로 남아 있도록 $h2$ 계산동안에 첨가된다)

성질 19.4 Rabin-Karp 패턴 일치는 극단적으로 선형이다.

이같은 알고리즘은 명백히 $N + M$ 의 시간에 비례하나 그러나 패턴과 같은 해시 값을 지니는 텍스트에서의 위치를 진정으로 발견하는 것에 주의를 해야한다. 확실히 하기 위해서, 패턴과 더불어서 그 텍스트의 직접적인 비교를 수행하는 것이다. 그러나, $\%$ 계산과 실제적인 해시 표를 유지할 필요가 없는 사실에 의해서 이루어진 q 의 매우 큰 값의 이용은 충돌이 발

생되어지는 것이 아닌 것으로 그것을 극단적으로 만드는 것이다. 이론적으로 이같은 알고리즘은 최악의 경우 여전히 $O(NM)$ 이 걸리나 실제적으로는 약 $N + M$ 단계에 의존한다. □

다중 검색(Multiple Searches)

지금까지 논의된 알고리즘들은 모두 특별한 스트링 검색 문제쪽에서 이루어졌다. 즉, 주어진 텍스트 스트링에서 주어진 패턴의 발생빈도를 찾는 것이다. 만약 같은 텍스트 스트링이 많은 패턴 검색의 목적물로서 존재를 하는 경우, 그때 연속적인 검색들이 효율적으로 이루어지도록 하기 위해서 스트링상에 어떤 처리를 수행할 가치가 있다.

만약 큰 수의 검색들이 존재를 하면, 스트링 검색 문제는 이전 절에서 본 일반적인 검색 문제의 특별한 경우로서 간주를 하게 된다. 텍스트 스트링을 단순히 N 개 겹치는 “키들”로서 즉, i 위치에서 시작하는 텍스트 스트링 전체인 $a[i], \dots, a[N]$ 로서 정의되는 i 번째 키로서 취급한다. 물론, 키들 그 자체는 아니나 그것들에 대한 포인터로서 처리를 한다. 즉, 키들 i 와 j 를 비교할 필요가 있을 때, 텍스트 스트링에서 위치 i 와 j 에서 시작하는 문자단위의 비교를 수행한다.(만약 끝에서 모든 다른 문자들보다 더 큰 “표지” 문자를 이용한 경우, 키들 중의 하나는 항상 다른 것들보다 더 크게 된다) 그때 이전 절에서의 해싱, 이진 트리와 다른 알고리즘들이 직접적으로 이용된다. 첫째, 전체 구조는 텍스트 스트링에서 생성되고 그리고 효율적인 검색은 특별한 패턴에 대해 수행된다.

많은 상세한 것들이 이같은 방법으로서 스트링 검색으로 검색 알고리즘을 적용하므로서 수행된다. 즉, 의도는 어떤 스트링 검색 응용에 대해 옵션으로서 이것들을 지적하도록 하는 것이다. 다른 방법들은 다른 상황에서 적절하게 된다. 예를 들면, 검색들이같은 길이의 패턴에 대해 존재를 하는 경우, Rabin-Karp 방법에서와 같이 한 번의 주사로서 생성된 해시 표는 평균적으로 일정한 검색 시간을 나타낸다. 다른 한편으로 만약 패턴들이 가변 길이를 지니면, 그때 트리 구조의 방법들중의 하나는 적절하다.(패트리시아는 그런 응용에 대해 특별히 적합한 것이다)

문제에서 다른 상황들은 그것을 더 어렵게 하고 그리고 다음 두 장에서 논의할 가공할 정도의 다른 방법들로 나타낸다.

연습문제

1. 오른쪽에서 왼쪽으로 패턴을 주사하는 주먹구구식 패턴 일치 알고리즘을 구현하라.
2. 패턴 AAAAAAAAAA에 대한 Knuth-Morris-Pratt 알고리즘의 next 표는 무엇인가?
3. 패턴 ABRACADABRA에 대한 Knuth-Morris-Pratt 알고리즘의 next 표는 무엇인가?
4. 패턴 ABRACADABRA에 대해 검색할 수 있는 유한 상태 기계를 그려라.
5. 50개 연속적인 빈칸들의 스트링에 대해 어떻게 텍스트를 검색하는가?
6. 패턴 ABRACADABRA에 대해 오른쪽-왼쪽 주사에 대한 오른쪽에서 왼쪽으로의 skip 표는 무엇인가?
7. 단지 불 일치 휴리스틱으로서 오른쪽에서 왼쪽으로의 주사가 나쁘게 수행이 되어지는 예제를 만들어라.
8. 중간 문자가 “와일드 카드”(어떤 텍스트 문자도 그것과 일치 될 수 있는 것)인 부가적인 조건으로된 주어진 패턴에 대해 검색을 하기 위해서 어떻게 Rabin-Karp 알고리즘을 변경시키는 가?
9. 2차원 텍스트에서 패턴에 대한 검색을 위해서 Rabin-Karp 알고리즘의 버전을 구현하라. 패턴과 텍스트 둘다는 문자들의 직각인 것으로 가정을 한다.
10. 무작위의 1000-비트 텍스트 스트링을 생성하는 프로그램을 기술하고, $k = 5, 10, 15$ 에 대해 스트링에서 어느 곳에서도 존재하는 마지막 k 비트들의 모든 발생빈도를 찾아라.(다른 방법들은 k 의 다른 값들에 대해 적절하다)

찾고자하는 패턴에 대해 다소 적은 완전한 정보로서 스트링 검색을 수행하도록 하는 것이 가끔은 바람직하다. 예를 들면, 텍스트 편집기의 사용자들은 패턴의 단지 일부분만을 기술하거나 혹은 몇 가지 다른 단어들과 일치하는 패턴을 설명하기 위해서 혹은 어떤 특별한 문자들의 발생되어지는 빈도수는 무시되도록 기술하기를 바라는 경우를 보면 된다. 본 장에서는 이같은 형태의 패턴 일치(pattern matching)가 어떻게 효율적으로 수행되는 가를 고려한 것이다.

이전 장에서의 알고리즘들은 패턴의 완전한 기술에 대해 오히려 기본적인 의존을 지니므로 다른 방법들을 고려해야만 한다. 고려해야 할 기본적인 메카니즘들은 최악의 경우 MN^2 시간에 비례를 하고 전형적인 응용에서 매우 빠른 N -문자 텍스트 스트링에서 복잡한 M -문자 패턴들과 일치하는 매우 강력한 스트링 검색 기법을 가능토록 하는 것이다.

첫째, 패턴을 기술하는 방법을 찾아야만 한다. 즉, 엄정한 방법으로서 위에서 제시된 부분적인 스트링 검색 문제들의 한 종류를 기술하는데 이용되는 “언어”이다. 이같은 언어는 이전 절에서 이용된 연산인 “텍스트 스트링의 i 번째 문자가 패턴의 j 번째 문자와 일치하는가를 단순히 체크”하는 것 보다 더 강력한 기본적인 연산들을 수반한다. 본 장에서는 텍스트 스트링에서 패턴을 검색하는 기계의 상상적인 형태에 의해서 기본적으로 세 가지 연산을 고려할 수가 있다. 패턴 일치 알고리즘은 기계의 이같은 형태의 연산을 모의 실험하는 방법이다. 다음 장에서는 알고리즘이 실제로 검색을 수행하는 사용자들 자신의 스트링 검색 일을 기계 구조에 적용하도록 기술하는 패턴 기술자로서 어떻게 번역하는 가를 보게 된다.

보는 바와 같이, 패턴 일치 문제에 대해 개발된 해법은 전산학에서 기본적인 과정들에 관련된 것이다. 예를 들면, 주어진 패턴 기술자에 의해서 내재된 스트링 검색 일을 수행하기 위

해, 프로그램에서 이용되는 방법은 주어진 C++ 프로그램에서 내재된 계산적인 일을 수행하도록 C++ 시스템에서 이용되는 방법과 유사하다.

패턴을 기술

다음의 세 가지 기본적인 연산으로 함께 결합된 기호들로 구성된 패턴 기술자를 고려해 보자.

- (i) 결합(Concatenation) 이것은 이전 장에서 이용된 연산이다. 만약 두 개의 문자들이 패턴에서 인접된 경우, 같은 두 개 문자들이 텍스트에서 인접된 경우에만 일치가 된다. 예를 들면, AB는 A다음에 B가 되는 것을 의미한다.
- (ii) 혹은(Or) 이것은 패턴에서 대안을 설명하도록 하는 연산이다. 만약 문자들 사이에 *or*를 지니면, 그때 문자들중의 어느 것도 텍스트에서 발생되지만 하면 일치되는 것이다. 이 같은 연산을 기호 +를 이용함으로써 정의되고 임의적인 방법으로서 결합과 함께 그것을 결합하기 위해서 괄호를 이용한다. 예를 들면, A+B는 “A 혹은 B”를 의미한다. 그리고 C(AC+B)D는 “CACD 혹은 CBD”를 의미하고 그리고 (A+C)((B+C)D)는 “ABD 혹은 CBD 또는 ACD 또는 CCD”를 의미한다.
- (iii) 폐쇄(Closure) 이 연산은 임의적으로 반복이 되어지도록 패턴의 일부분을 허용하는 것이다. 만약 기호의 폐쇄를 지니면, 기호가 어떤 수만큼(0을 포함해서) 발생하는 경우에 일치된다. 폐쇄는 반복되어지는 문자나 괄호로 이루어진 그룹뒤에 *를 놓으므로서 정의된다. 예를 들면, AB*는 A다음에 어떤 수만큼의 B가 반복이 되어져 나오는 스트링으로 구성되고 반면에 (AB)*는 A와 B가 번갈아 가면서 발생하는 스트링이다.

이같은 세 가지 연산을 이용해서 생성된 기호들의 스트링을 정규 식(regular expression)이라고 부른다. 각 정규 식은 많은 특별한 텍스트 패턴을 기술한다. 우리의 목적은 주어진 정규 식으로 기술이 된 패턴의 어떠한 것도 주어진 텍스트 스트링에서 발생하는 지를 결정하는 알고리즘을 개발하는 것이다.

정규 식 패턴-일치 알고리즘을 개발하는데 있어, 기본적인 원칙을 제시하기 위해서는 결합, 혹은(or) 그리고 폐쇄에 집중을 할 것이다. 여러 가지 부가적인 것들이 실제 시스템에서 편리하도록 공통적으로 만들어 진다. 예를 들면, -A는 “A를 제외한 어떤 문자와 일치”함을 의

미한다. 이같은 *not* 연산이 A를 제외한 모든 문자들을 수반으로 한 *or*와 같은 것이나, 이용하기는 훨씬 쉽다. 비슷하게, “?”는 “어떤 문자와 일치”됨을 의미한다. 다시 이것은 큰 *or*보다 훨씬 더 조밀한 것이다. 큰 패턴의 기술을 쉽게하는 부가적인 기호들의 다른 예제들은 라인의 시작이나 끝 혹은 어떤 문자나 어떤 숫자등과 일치하는 기호들이다.

이같은 연산들은 현저하게 기술적이다. 예를 들면, 패턴 기술자 $?(ie + ei)?$ 는 그것들 내에 *ie*나 *ei*를 지니는 모든 단어들과 일치한다.(그리고 그것은 철자가 틀릴 수도 있다) 즉, $(1 + 01) * (0 + 1)$ 는 두 개 연속적인 0을 지니지 않는 0의 것과 1의 것의 모든 스트링들을 기술한다. 명백히 같은 스트링을 기술하는 많은 다른 패턴 기술자가 존재한다. 즉, 효율적인 알고리즘들을 기술하고자 하는 것과 같이 간결한 패턴 기술자를 설명하도록 한다.

패턴 일치 알고리즘은 주먹구구식 왼쪽에서 오른쪽으로 스트링 검색하는 방법의 일반화로써 간주된다.(19장에서 보게된 첫 번째 방법) 패턴 기술과 일치하는 그 위치에서 시작하는 부분 스트링이 존재하는 지를 각 위치에서 테스트하면서, 알고리즘은 왼쪽에서 오른쪽으로 텍스트 스트링을 주사함으로서 패턴 기술과 일치하는 텍스트 스트링에서 가장 왼쪽의 부분 스트링을 찾는 것이다.

패턴 일치 기계

Knuth-Morris-Pratt 알고리즘은 텍스트를 조사하는 검색 패턴에서 구성된 유한 상태 기계로서 간주하는 것을 회상하자. 정규 식 패턴 일치에 대해 이용되는 방법은 이것의 일반화이다.

Knuth-Morris-Pratt 알고리즘에 대한 유한 상태 기계는 텍스트 스트링에서 문자를 보고 그리고 일치하는 것이 있으면 한 상태로 혹은 일치 한 것이 없으면 다른 상태로 변경시키므로써 상태에서 상태로 변경을 하는 것이다. 어떤 점에서의 불 일치는 패턴의 그 위치에서 시작되는 텍스트에서 발생되지 않는 것을 의미한다. 알고리즘 그 자체는 기계의 모의 실험으로서 간주한다. 모의 실험을 쉽도록 하는 기계의 특성은 확정적(deterministic)인 것이다. 즉, 각 상태 전이는 다음 입력 문자에 의해서 완전히 결정을 하는 것이다.

정규 식을 처리하기 위해서, 더 강력한 추상적인 기계를 고려하는 것이 필요하다. *or* 연산 때문에, 기계는 패턴이 단지 한 개의 문자를 조사하므로써 주어진 점에서 발생이 되는지 아닌지를 결정할 수가 없다. 사실, 폐쇄 때문에 많은 문자들이 불 일치가 발견되기 이전에 어떻게 해서 조사가 되어질 필요가 있는지를 결정할 것도 없는 것이다. 이같은 문제들을 극복하기 위한 가장 자연스런 방법은 비 확정적(nondeterminism)의 능력으로 된 기계를 부여하는

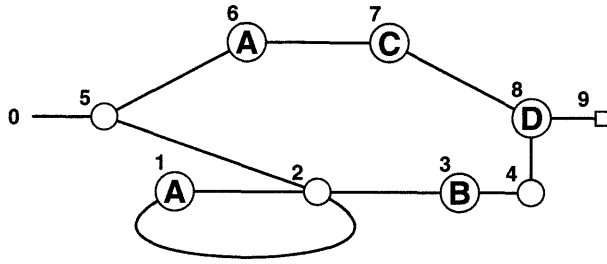


그림 20.1 (A*B+AC)D에 대한 비 확정적 패턴 인식 기계

것이다. 즉, 패턴과 일치되도록 하는데 한 가지 방법이상과 직면을 할 때, 기계는 올바른 것으로 “추측”이 된다! 이같은 연산은 허용하기 불가능한 것처럼 보이나 그런 기계의 행동을 모의 실험하기 위한 프로그램을 기술하는 것은 쉽다.

그림 20.1은 텍스트 스트링에서 패턴 기술자 (A * B + AC)D에 대해 검색을 하는데 이용되는 비 확정적 유한 상태 기계를 제시한 것이다.(상태들은 숫자로 되어져 있다) 이전 장의 확정적 기계와는 달리, 기계는 문자로 표시된 상태에서 텍스트 스트링의 그 문자와 일치하므로서(그리고 지나온 것을 주사하는 것) 그 상태에 의해 “지적이 되는” 상태에 이르기까지 계속된다. 기계를 비확정적으로 만드는 것은 라벨이 존재하지 않을 뿐 아니라 두 개 다른 연속적인 상태들을 “지적” 할 수가 있는 어떤 상태들(빈(null) 상태들이라 부름)이 존재하는 것이다.(그림에서 상태 4와 같은 몇 가지 비어있는 상태들은 기계의 연산에 영향을 주지 않고 단지 보는 바와 같이 기계를 구성하는 프로그램의 구현을 용이하게 하는 하나의 출구로서의 “연산이 없는(no-op)” 상태들이다. 상태 9는 기계를 멈추게 하는 출구 없이 비어있는 상태이다) 그런 상태에 존재할 때, 기계는 입력에 관계없이(어떤 것을 지나온 것을 주사함이 없이도) 연속적인 상태들중의 어느곳에 든지 갈 수가 있다. 이전 장에서 처럼 “불 일치” 전이가 없음을 주시해야한다. 즉, 기계는 일치가 되도록 하는 연속적인 전이를 추측해야할 방법이 없는 경우에 일치되도록 하지는 못한다.

기계는 유일한 초기(initial) 상태(왼쪽에 부착된 라인으로 지시된 것)와 유일한 마지막(final) 상태(오른쪽의 작은 사각형)를 지닌다. 초기 상태내에서 시작을 할 때, 기계는 문자들을 읽고, 그것의 규칙에 따라서 상태를 변경하고 “마지막 상태”로 끝이 나게하는 패턴에 의해서 기술된 어떤 스트링을 “인식”하도록 하는 것이다. 기계는 비 확정적인 능력을 지니기 때문에, 해결로 이끌어지는 연속적인 상태 변경을 추측할 수가 있다.(그러나 표준 컴퓨터상에 기계를 모의 실험을 하려고 할 때, 모든 가능성들을 추구해야만 한다) 예를 들면, 패턴 기술자(A * B + AC)D가 다음 텍스트 스트링에서 발생이 되는지를 결정하기 위해서,

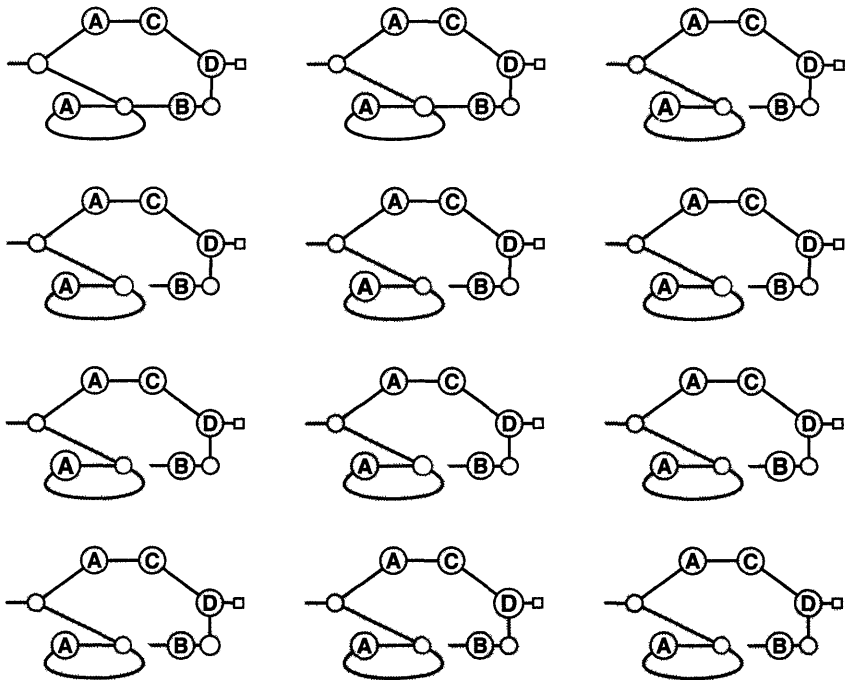


그림 20.2 AAABD를 인식

CDAABCAAABDDACDAAC

기계는 첫 번째나 두 번째 문자상에서 출발을 할 때 즉각적으로 실패를 보고한다. 그것은 다음 두 개 문자들상에 실패 보고를 하기 위해서 어떤 것을 처리한다. 그리고 5번째나 6번째 문자들상에 실패를 즉각적으로 보고를 한다. 그리고 7번째 문자상에서 시작이 되는 경우 AAABD를 인식하기 위해서 그림 20.2에서 제시된 연속적인 상태 전이들을 추측할 수가 있다.

식의 부분들에 대해 부분적인 기계들을 생성하고, 두 개의 부분적인 기계들이 세 가지 연산들의 각각에 대해 더 큰 것으로 구성하는 방법을 정의함으로써 주어진 정규 식에 대한 기계를 구성을 할 수가 있다. 즉, 결합, *or*와 폐쇄이다.



그림 20.3 한 문자를 인식하기 위해 두 개 상태 기계

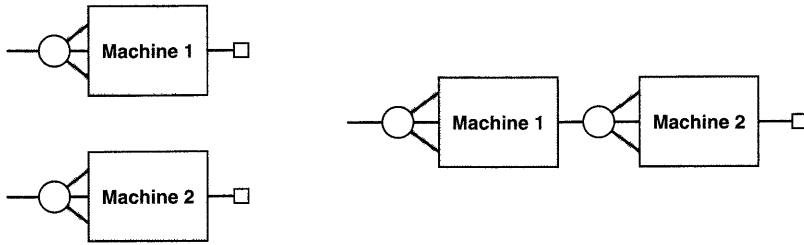


그림 20.4 상태 기계 구성: 결합

특별한 문자를 인식하는 사소한 기계로서 시작을 하자. 그림 20.3에 제시된 것과 같이 초기 상태(또한 문자를 인식하는 것)와 마지막 상태인 두 개의 상태 기계로서 이것을 기술하는 것이 편리하다.

지금 개별적인 식들에 대한 기계들에서 두 개의 식을 결합하는 기계를 설정하기 위해서, 그림 20.4에 제시된 것과 같이 두 번째의 초기 상태와 첫 번째 초기 상태를 단순히 병합하는 것이다.

비슷하게 *or* 연산에 대한 기계는 두 개의 초기 상태를 가르키는 새로이 비어있는 상태를 첨가함과 하나의 마지막 상태가 다른 것을 지적하도록 하므로서 생성된다. 그리고 이것은 그림 20.5에 제시된 것과 같이 결합된 기계의 마지막 상태가 되는 것이다.

마지막으로 폐쇄 연산에 대한 기계는 마지막 상태를 처음 상태로 하고, 그림 20.6에 제시된 것과 같이 그것을 옛날의 초기 상태와 새로운 마지막 상태로 되돌리도록 하면서 생성하는 것이다.

기계는 이같은 규칙을 연속적으로 적용을 함으로서 어떤 정규 식에 대응된다. 위의 예제 기계에 대한 상태는 기계가 왼쪽에서 오른쪽으로 패턴을 조사하므로서 생성되는 것과 같이 생성의 순서로서 숫자화가 된다. 그래서 위의 규칙들에서 기계의 구성은 쉽게 추적된다. 정

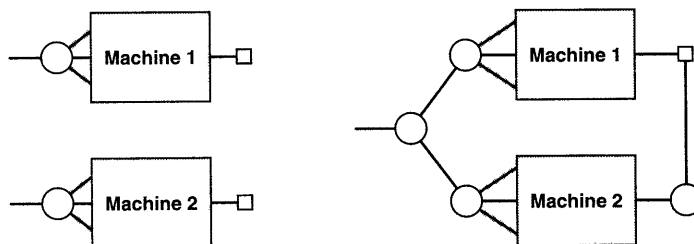


그림 20.5 상태 기계 구성: *or*



그림 20.6 상태 기계 구성 : 폐쇄

규 식에서 각 문자에 대한 두 가지 상태 기계를 지니고, 각 +와 *는 한 상태로 하여금 생성 되도록 하는 것(결합은 하나를 삭제되도록 한다)을 주시해야 한다. 그래서 상태의 번호는 정규 식에서 문자들의 수보다 2배 적은 것이다.

기계를 표현

비 확정적인 기계는 위에서 제시된 단지 세 가지 구성 규칙을 이용해서 모든 것이 구성되고, 그것들을 직접적인 방법으로서 처리하기 위해서 그것들의 간단한 구조의 장점을 이용한다. 예를 들면, 단지 두 개의 선들이 어떤 상태에서 나오게 한다. 사실 거기에는 단지 두 개의 상태 형을 지닌다. 즉, 입력 알파벳에서 한 문자로서 표시된 것(한 개 선만이 나오는 것)과 표시가 않된(비어있는) 상태(두 개나 그 이하의 선들이 나오는 것)들이다. 이것은 기계들이 노드당 몇 개의 정보 조각들로서 만 표현이 되는 것을 의미한다. 단지 번호로서만 상태들을 접근하기 때문에, 기계에 대한 대부분의 적절한 구조는 배열 표현이다. 기계를 접근하고 표현하기 위해서 state에 의해 인덱스된 세 가지 병렬 배열 ch, next1과 next2를 이용한다. 각 상태는 단지 두 개의 의미있는 정보의 조각을 이용하기 때문에 이같은 공간의 양에서 2/3로서 얻는 것이 가능하나 그러나 명확성을 위해 이같은 개선을 하고 그리고 또한 패턴 기술자는 특별히 길지가 않기 때문이다.

위의 기계는 그림 20.7에서와 같이 표현된다. state에 의해서 인덱스된 엔트리들은 양식의 비확정적인 기계에 대한 교훈 즉, “만약 여러분이 state내에 존재하고, ch[state]를 보는 경우, 그때 문자를 주사하고 상태 next1[state](또는 next2[state])로 가게 된다”로서 해석된다. 상태 9는 이 예제에서의 마지막 상태이고, 상태 0는 next 엔트리들이 실제 초기 상태의 수인 유사(pseudo) 초기 상태이다. 한 개의 연속자로 된 비어있는 상태들(둘다 next엔트리들이 같다)과 마지막 상태(둘다는 next 엔트리들이 0이다)로서 비어있는 상태에 대해 이용함을 주시해야한다.

	0	1	2	3	4	5	6	7	8	9
ch		A		B			A	C	D	
next1	5	2	3	4	8	6	7	8	9	0
next2	5	2	1	4	8	2	7	8	9	0

그림 20.7 그림 20.1의 기계에 대한 배열 표현

정규 식 패턴 기술자에서 기계를 어떻게 생성을 하고 그리고 배열로서 그런 기계들이 어떻게 표현되는 지를 보게 된다. 그러나 정규 식에서 대응되는 비 확정적 기계 표현에 이르는 번역을 수행하는 프로그램을 기술하는 것은 아주 다른 문제이다. 사실, 주어진 정규 식이 합법적인 경우인가를 결정하기 위한 프로그램을 기술하는 것조차 미창출된 것에 대한 도전이다. 다음 장에서 파싱(parsing)이라 부르는 이같은 연산을 더 자세하게 보게 될 것이다. 이같은 번역이 수행된다고 가정하자. 그러면 관심은 정규 식 패턴 기술에 대응되는 특별한 비확정적 기계를 나타내는 ch, next1과 next2 배열들을 이용가능토록 한다.

기계를 모의 실험

일반적인 정규 식 패턴 일치 알고리즘의 개발에서 마지막 단계는 비 확정적인 패턴 일치 기계의 연산을 어디선가 모의 실험하는 프로그램을 기술하는 것이다. 올바른 답을 “추측”할 수 있는 프로그램을 기술하는 개념은 우스운 것이다. 그러나 이 경우에서 시스템적인 방법으로 모든 가능한 일치들의 상태를 파악 하도록 하므로서 결국에는 정확한 것을 만나게 된다.

한 가지 가능성은 비 확정적인 기계를 흉내내는 재귀의 프로그램을 개발하는 것이다.(그러나 올바른 것을 추측하기 보다는 오히려 모든 가능성을 시도하는 것이다) 이같은 방법을 사용하는 대신에, 디큐(deque)라 부르는 오히려 특정한 자료 구조에서 고려중인 상태를 유지함으로써 방법의 기본적인 연산 원칙을 나타내는 비 재귀적인 구현을 보게 된다.

개념은 기계들이 현재 입력 문자를 “보므로서” 만날 가능성이 있는 모든 상태들을 파악하는 것이다. 이같은 상태들의 각각은 차례로 처리된다. 즉, 비어있는 상태는 두 개의 상태(또는 더 적은 상태들로), 현재 입력과 일치하지 않은 문자들은 제거시키는 상태들과 현재 입력과 일치하는 문자들은 기계가 다음의 입력 문자로서 볼 때 이용을 위해 새로운 상태로 이끌어지는 것에 대한 상태들로 이끌어진다. 이와 같이, 비 확정적인 기계는 텍스트에서 특별한 점에서 존재할 가능성이 있는 모든 상태들의 리스트를 유지하는 것을 원한다. 문제는 이같은 리스트에 대해 적절한 자료 구조를 설계하는 것이다.

비어있는 상태를 처리하는 것은 스택(stack)을 요구하는 것으로 보인다. 그 이유는 순환 제거에서와 같이 수행이 되어진 두 개의 것들 중의 하나를 본질적으로 연기를 시키는 것이기 때문이다.(그래서 새로운 상태는 그것이 무한정으로 연기가 되지 않도록 현재 리스트의 시작에 놓게 된다) 이같은 두 개의 자료 구조 사이에서 선택을 하기 보다는 오히려 둘다를 이용한다! 디큐(deque: “double-ended queues”)는 스택과 큐의 장점들로 결합한 것이다. 즉, 디큐는 항목들이 어느 한쪽에서 삽입되는 리스트이다.(실제적으로 끝이 아니고 시작에서 항상 항목들을 제거시키기 때문에 “출력 제한 디큐”를 이용한다. 즉, “덱(deck)의 제일 마지막에서 처리”되는 것이다)

기계의 중요한 성질은 비어있는 상태들로 구성되는 “반복”이 없는 것이다. 그 이유는 그렇지 않으면, 영원히 반복이 되어지는 것인 비확정적으로서 결정되기 때문이다. 이것은 어느 순간에 디큐상에 상태의 수가 패턴 기술자에서 문자의 수보다 적은 것으로 됨을 의미하는 것이다.

위에 주어진 프로그램은 위에서 기술한대로 비 확정적 패턴 일치 기계의 행동을 모의 실험하도록 디큐를 이용하도록 한다. 입력에서 특별한 문자를 조사하는 반면에, 비 확정적인 기계는 여러 가지 가능한 상태들 중의 어느 하나인 것이다. 즉, 프로그램은 3장에서 프로시저 push, put와 pop를 이용해서 디큐에서 이것들의 상태를 파악 하는 것이다. 배열 표현이나 (3장에서 큐의 구현과 같은 것) 연결 표현(3장에서 스택 구현에서와 같이)이 이용되며 그것들의 구현은 생략했다.

프로그램의 주된 반복은 디큐에서 상태를 제거하고 그리고 요구된 행동을 수행하는 것이다. 만약 문자가 일치되면, 입력은 요구된 문자에 대해 체크를 한다. 즉, 만약 그것이 발견되면, 상태 전이는 디큐의 끝에서 새로운 상태를 놓으므로서 영향을 받는다.(그 결과 현재 문자를 포함한 모든 상태들은 다음 상태를 포함한 것들 전에 처리를 해야한다) 만약 상태가 비어있으면, 모의 실험되어지는 두 가지 가능한 상태는 디큐의 시작에 놓게 된다. 현재 입력 문자를 포함하는 상태들은 디큐에서 scan=-1인 표시자로서 다음이 포함된 것들에서 분리되어서 유지된다. 즉, scan이 만날 때, 입력 스트링으로 포인터를 미리 앞서게 한다. 반복은 입력의 끝에 도달될 때 종료되고,(일치가 일어나지 않을 때) 상태 0에 도달이 되거나(합법적인 일치) 혹은 단지 한 개의 항목인 scan 표시자는 디큐상에 남겨진다.(일치가 없을때) 이것은 다음 구현으로서 직접적으로 이끌어진다.

```
const int scan = -1;
int match(char *a)
```



```

{
    int n1, n2; Deque dq(100);
    int j = 0, N = strlen(a), state = next1[0];
    dq.put(scan);
    while(state)
    {
        if ( state == scan ) { j++; dq.put(scan); }
        else if ( ch[state] == a[j] )
            dq.put(next1[state]);
        else if ( ch[state] == ' ' )
        {
            n1 = next1[state]; n2 = next2[state];
            dq.push(n1); if ( n1 != n2 ) dq.push(n2);
        }
        if ( dq.empty() || j == N ) return 0;
        state = dq.pop();
    }
    return j;
}

```

이같은 함수는 위에서 기술한대로 ch, next1과 next2에서 패턴을 표현하는 비 확정적인 기계를 이용해서 일치하는 텍스트 스트링 a를 인수(포인터에)로서 취급을 한다. 패턴에서 일치된(일치되지 않으면 0으로) a의 가장 짧은 초기의 부분 스트링의 길이를 되돌려 준다. 편리성을 위해, 텍스트 스트링 a의 마지막 문자는 패턴을 나타내는 ch 배열에서 차지 못한 유일한 표지 문자로서 존재한다고 가정하자.

그림 20.8은 표본 기계가 텍스트 스트링 AAABD로서 처리를 할 때, 제거가 될 때마다 디큐의 내용을 나타낸 것이다. 이같은 다이어그램은 3장에서의 큐에 대해 이용하는 것과 같이 배열 표현으로 가정을 한 것이다. 즉, + 기호는 scan을 나타낸다. scan표시자가 디큐의 앞에 도착할 때마다.(다이어그램에서 제일 밑에), j 포인터는 텍스트에서 다음 문자로 증가된다. 이와 같이 텍스트에서 첫 번째 문자(첫번째 A)를 주사하는 반면에 상태 5로서 시작된다. 첫 번째 상태 5는 상태 2와 6으로 이끌어지고 그때 상태 2는 상태 1과 3으로 이끌어지고, 이것들 모두는 같은 문자를 주사할 필요가 있고 그리고 디큐의 시작 상에 존재케 된다. 그때 상태 1은 상태 2로 이끌어지나 디큐의 끝에 존재케 된다.(다음 입력 문자에 대해) 상태 3은 B를 주사하는 동안에 다른 상태로 이끌어지므로써 A가 주사되는 동안에는 그것은 무시된다.

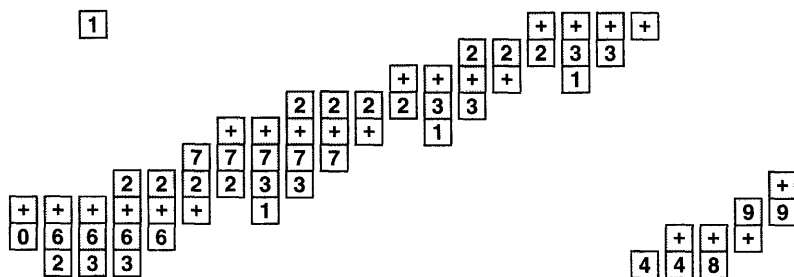


그림 20.8 AAABD의 인식 동안의 디큐 내용

“scan” 표지인 것은 디큐의 앞에 도달이 될 때, 기계는 A를 주사한 후에 상태 2나 상태 7로 존재케 된다. 그때 디큐의 앞에 scan이 도달되는 두 번째 순간 상태 2는 AA를 주사한 후에 유일한 가능성을 지닌 것을 찾기 위해서 두 번째 A를 “보는 것인” 반면에, 프로그램은 상태 2, 1, 2와 7을 시도한다. 지금 세 번째 A를 보면서, 유일한 가능성은 상태 2, 1과 3이다.(AC 가능성은 지금 배제가 된 것이다) 이같은 세 가지 상태는 AAAB를 주사한 후에 결국 상태 4로 이끌어지도록 하는 것이다. 계속적으로해서, 프로그램은 상태 8로 가서 D를 주사하고 마지막 상태에서 끝난다. 일치는 찾아지게 되나 더 중요한 것은 텍스트 스트링과 일치되는 모든 전이가 고려되는 것이다.

성질 20.1 N 개 문자들의 텍스트 스트링에서 패턴을 찾기 위해서 M -상태 기계의 연산을 모의 실험하는 것은 최악의 경우 NM 보다 적은 상태 전이들로서 수행된다.

match의 실행시간은 일치되는 패턴에 주로 의존적인 것이다. 그러나 N 개 입력 문자들의 각각에 대해, 기껏해야 기계의 M 개 상태를 처리하는 것으로 보이고 그래서 최악의 경우 실행시간은 MN 에 비례를 한다.(텍스트에서 각 시작 위치에 대해) 불행히도, 이것은 위에서 구현한 것과 같은 match에 대해 사실이 아니다. 그 이유는 디큐상에 상태를 놓을 때, 프로그램은 그것이 이미 거기에 존재하는 지를 체크할 수가 없다. 그래서 디큐는 같은 상태의 이중 복사를 포함한다. 이것은 실제 응용에서 많은 효과를 지니지 않는다. 그러나 체크되지 않고 남겨두는 경우 간단한 병리적인 경우로서 마구 퍼지면서 처리된다. 예를 들면, 이같은 문제는 패턴 $(A^*A)^*B$ 가 N 개의 A다음에 N 개의 B로 되는 스트링에 대해서 일치가 될 때 2^{N-1} 상태로 된 디큐로 된다. 이같은 것을 피하기 위해서, match에 의해서 이용된 디큐 루틴은 디큐상에 이중 상태들을 놓는 것을 피하기 위한 방법으로 구현된다.(그리고 기껏해야 M 개

상태들이 각 입력 문자에 대해 처리가 되어지는 것을 보증한다) 이것은 디큐상에 존재하는 것을 나타내는 state에 의해서 인덱스되는 배열을 유지하도록 함으로서 수행된다. 이같은 변화에 대해, 텍스트 스트링의 어떤 부분이 패턴에 의해서 기술이 되는 가를 결정하는 것에 대해서 그때 전체 합계는 $O(MN^2)$ 이다. \square

거의 모든 비 확정적인 기계는 40장에 더 자세히 설명이 된 것과 같이 매우 효율적으로 모의 실험이 되는 것은 아니나, 이 응용에서 간단한 가설적인 패턴 일치 기계의 이용은 아주 어려운 문제에 대한 아주 합리적인 알고리즘으로 된다. 그러나 알고리즘을 완성하기 위해서, 임의의 정규식을 위의 코드로서 해석하는 것에 대한 “기계”로 번역을 하는 프로그램이 필요하다. 다음장에서, 컴파일러와 파싱 기법의 더 일반적인 논의의 내용에서 그런 프로그램의 구현을 보게된다.

연습문제

1. 이진 스트링에서 4개나 몇안되는 연속적인 1의 것의 모든 발생을 인식하는 것에 대한 정규식은 무엇인가?
2. 패턴 기술자 $(A+B)^*C$ 에 대한 비 확정적인 패턴 일치 기계를 그려라.
3. 이전 연습문제에서 여러분의 기계가 ABBAC를 인식하도록 하는 상태 변이는 무엇인가?
4. *not* 함수를 처리하기 위해서 비 확정적인 기계를 어떻게 변경하는가를 설명하십시오.
5. “don’t-care” 문자를 처리하기 위해서 비 확정적인 기계를 어떻게 변경하는가를 설명하십시오.
6. M 개 *or* 연산자와 폐쇄 연산자로 정규식을 기술할 수 있는 것은 얼마나 많은 패턴들이 존재하는가?
7. *not* 함수와 “don’t-care” 문자들로서 정규식을 처리하도록 *match*를 변경하십시오.
8. *match*의 실행시간이 가능한 크게 되는 것에 대한 길이 M 의 패턴 기술자와 길이 N 의 텍스트 스트링을 어떻게 구성하는가를 보여라.
9. 성질 20.1의 증명에서 기술이 된 문제를 피하기 위한 *match*의 버전을 구현하라.
10. 텍스트 스트링 ACD로된 텍스트에서 예제 기계를 모의 실험토록 *match*가 이용될 때 상태가 제거될 때 마다의 디큐의 내용을 보여라.

빈 면

21 장

파싱

여러 가지 기본적인 알고리즘들은 합법적인 컴퓨터 프로그램을 인식하고, 그것들을 더 낮은 처리에 대해 적합한 형식으로 분해하도록 개발을 한 것이다. 파싱(parsing)이라 부르는 이 같은 연산은 전산학을 넘어선 응용을 지닌다. 그 이유는 일반적으로 언어 구조의 연구와 직접적인 관련이 된 것이기 때문이다. 예를 들면, 파싱은 자연(인간) 언어를 “이해”하려고 하는 시스템에서 그리고 한 언어를 다른 언어로 번역을 하는 시스템에서 중요한 역할을 하는 것이다. 특별한 관심의 경우는 C++와 같은 “고급(high-level)” 컴퓨터 언어에서(인간이 사용하기 적합한 것) “저급(low-level)” 어셈블리나 기계어(기계 실행에 적합한 것)에 이르도록 번역을 하는 것이다. 그런 번역을 수행하는 것을 컴파일러(compiler)라고 부른다. 실제로 4장에서 산술식을 표현하는 트리를 생성할때에 파싱 방법에 대해서 이미 보았다.

두 개 일반적인 방법들은 파싱에 대해 이용이 된다. 즉, 하향식 방법은 전체 한 프로그램에서 합법적인 프로그램의 부분들을 먼저 찾고 그리고 일부분의 일부분을 찾아나가는 등으로 이루어진다. 이것은 조각들이 입력을 직접적으로 일치시킬 수 있을 만큼 충분히 적을때까지 계속된다. 상향식 방법은 입력의 조각들을 함께 놓고서 한 개의 합법적인 프로그램이 구성이 될 때까지 더 큰 조각들로서 만드는 방법이다. 일반적으로 하향식 방법은 재귀적이고 상향식 방법은 반복적이다. 즉, 하향식 방법들은 구현하기가 쉬운 것으로 생각이 되고 상향식 방법은 더 효율적인 것으로 간주된다. 4장의 방법은 상향식이다. 즉, 본 장에서는 하향식에 대해서 상세하게 살펴볼 것이다.

파서와 컴파일러 구성에서 수반되는 문제들의 완전한 취급은 본 교재의 범주를 벗어난 것이다. 그러나 이전 장의 패턴 일치 알고리즘을 완성하기 위해 간단한 “컴파일러”를 생성하므로, 수반된 몇 가지 기본적인 개념들중의 몇 가지를 고려할 수가 있다. 그때 정규식을 이전

장의 match 프로시저에 의해 이용되는 패턴 일치 기계로 번역하는 프로그램을 만들도록 파서를 변경하는 것이다.

본 장에서의 의도는 파싱과 컴파일러의 기본적인 원칙들에 대한 몇 가지 개념을 주는 동시에 유용한 패턴 일치 알고리즘을 개발하는 것이다. 확실히 어느정도의 깊이가 있는 레벨에서 수반되는 문제들은 취급을 할 수가 없다. 여러분은 미묘한 어려움으로 인해서 비슷한 문제에 대해서 같은 방법을 적용하므로서 제기되고 컴파일러 구성은 심각한 응용에 대해 유용한 여러 가지 고급화된 방법들로서의 아주 잘 개발된 영역임을 주시해야 한다.

문맥 자유 문법들(Context-Free Grammars)

주어진 언어로서 쓰여진 프로그램이 합법적인가를 결정하는 프로그램을 기술하기 이전에 정확히 어떤 것이 합법적인 프로그램으로 구성하는가의 기술이 필요하다. 이같은 기술을 문법(grammar)이라고 부른다. 즉, 용어를 설명하기 위해서 영어로서 언어를 생각해 보고, 이전 문장에서 “프로그램”에 대해 “문장”을 읽는다.(첫번째 발생하는 것은 제외하고) 프로그래밍 언어는 문맥 자유 문법들이라고 부르는 문법의 특별한 형태로 기술된다. 예를 들면, 모든 합법적인 정규식의 집합을 정의하는 문맥 자유 문법은 아래와 같다.

$$\begin{aligned}\langle expression \rangle &::= \langle term \rangle \mid \langle term \rangle + \langle expression \rangle \\ \langle term \rangle &::= \langle factor \rangle \mid \langle factor \rangle \langle term \rangle \\ \langle factor \rangle &::= (\langle expression \rangle) \mid v \mid (\langle expression \rangle)^* \mid v^*\end{aligned}$$

이같은 문법은 $(1+01)^*(0+1)$ 나 혹은 $(A*B+AC)D$ 와 같은 이전장에서 이용된 것들과 같은 정규식을 기술한다. 문법에서 각 라인들을 생성 규칙(production)이나 대체 규칙(replacement rule)이라고 부른다. 생성 규칙은 기술되어진 언어에서 이용되는 기호들인 터미널(terminal) 기호들 (,), + 와 *,(특별한 기호인 “v”는 어떤 문자나 디지트를 나타낸다) 문법의 내부에 존재하는 논 터미널(nonterminal) 기호 $\langle expression \rangle$, $\langle term \rangle$ 와 $\langle factor \rangle$ 그리고 생성 규칙의 의미를 기술하는데 이용되는 metasymbols $::=$ 와 \mid 로서 구성이 된다. “is a”라고 읽는 $::=$ 기호는 오른쪽에 의해서 생성 규칙의 왼쪽을 정의한다. 그리고 “or”로서 읽는 \mid 기호는 대안의 선택들을 나타낸다. 비록 이같은 간결한 기호 표시로서 표현을 하지만 여러 가지 생성 규칙들은 문법의 직관적인 기술에 대해 간단한 방법으로 대응된다. 예를 들면, 예제 문법에서

이같은 유추 과정의 결과를 기술하는 한 가지 자연스러운 방법이 파서 트리(parse tree)이다. 즉, 파스되는 스트링의 완전한 문법적인 구조이다. 예를 들면, 그림 21.1에서의 파서 트리는 스트링 $(A*B+AC)D$ 에 대해 위의 문법에서 기술된 언어내에 존재하는 것을 나타낸다. 이같은 파서 트리는 가끔 문장들을 주어, 동사 목적어등으로 나누는 영어에 이용된다.

```

graph TD
    E1[expression] --> T1[term]
    E1 --> F1[factor]
    T1 --> F2[factor]
    F2 --> LP["("]
    F2 --> E2[expression]
    F2 --> RP[")"]
    E2 --> T2[term]
    E2 --> P1["+"]
    E2 --> E3[expression]
    T2 --> F3[factor]
    F3 --> A1[A]
    F3 --> M1["*"]
    T2 --> T3[term]
    T3 --> F4[factor]
    F4 --> B[B]
    E3 --> T4[term]
    T4 --> F5[factor]
    F5 --> A2[A]
    T4 --> T5[term]
    T5 --> F6[factor]
    F6 --> C[C]
    F1 --> T6[term]
    T6 --> F7[factor]
    F7 --> D[D]
  
```

21장 파싱 343

문맥 자유 문법의 또 다른 예제는 The C++ Programming Language라는 책에서 볼 수가 있다. 즉, 이것은 합법적인 C++ 프로그램들을 기술한다. 합법적인 식들을 이용하고 인식하기 위해서 이절에서 고려되는 원칙들은 C++ 프로그램들을 실행하고 컴파일하는 복잡한 작업으로 직접적으로 적용된다. 예를 들면, 다음의 문법은 C++의 매우 적은 부분 집합들 즉, 덧셈과 곱셈을 포함한 산술식을 기술한 것이다.

$$\begin{aligned} \langle expression \rangle &::= \langle term \rangle \mid \langle term \rangle + \langle expression \rangle \\ \langle term \rangle &::= \langle factor \rangle \mid \langle factor \rangle * \langle term \rangle \\ \langle factor \rangle &::= (\langle expression \rangle) \mid v \end{aligned}$$

이같은 규칙들은 4장에서 인정이 된 것의 공식적인 방법으로 기술을 한 것이다. 즉, 그것들은 “합법적”인 산술식으로 구성되는 것으로 설명되는 규칙들이다. 다시 v 는 어떤 문자에 대한 특별한 기호이나 이 문법에서 문자들은 수치 값으로서 변수들을 표현하는 것이다. 이같은 문법에 대한 합법적인 스트링의 예제들은 $A+(B*C)$ 와 $A*(((B*C)*(D*E))+F)$ 이다. 후자의 경우는 4장에서 이미 파서 트리를 보았다. 그러나 그 트리는 위의 문법에 대응되지 않는다. 예를 들면, 괄호들이 명확히 매제 된 것은 아니다.

어떤 것을 정의하므로써, 몇몇 스트링들은 산술식과 정규식들다에서 완전히 합법적이다. 예를 들면, $A*(B+C)$ 는 “B와 C를 더하고 결과를 A와 곱하거나” 혹은 “B나 C 둘 중의 하나가 A의 어떤 수를 뒤따른 것을 취하게 되는 것”을 의미한다. 이것은 스트링이 형성이 되는지 아닌지를 체크하는 한 가지 사실은 명백하나, 그것이 의미하는 것을 이해하는 것은 아주 다른 것이다. 그것이 어떤 문법에 의해서 기술이 되는지 아닌지를 체크하기 위해서는 스트링을 어떻게 파스하는가를 본 후에 이 문제를 보도록 하자.

각 정규식은 그 자체가 문맥 자유 문법의 예제이다. 즉, 정규식에 의해서 기술이 될 수 있는 어떤 언어는 또한 문맥 자유 문법에 의해서 기술된다. 이것의 역은 틀린 경우이다. 예를 들면, 괄호가 “균형”을 이룬다는 개념은 정규식으로서 만들수가 없다. 문법들의 다른 형들은 문맥 자유 문법들이 할 수 없는 언어들을 기술할 수가 있는 것이다. 예를 들면, 문맥 의존형(context-sensitive) 문법들은 생성 규칙의 왼쪽 편은 단일 논 터미널이 될 필요가 없는 것을 제외하고는 위의 것들과 같은 것들이다. 언의 계층들과 그것을 기술하는 문법의 계층구조사이의 차이는 매우 조심스럽게 처리가 되어지는 것이고 전산학의 중심이 되는 아름다운 이론을 형성하는 것이다.

하향식 파싱(Top-Down Parsing)

한 가지 파싱 방법은 문법에 의해서 기술되는 것과 같은 언어들에서 스트링을 인식하기 위해 재귀를 이용하는 것이다. 단순히 놓으므로써, 문법은 그것을 프로그램에 직접적으로 돌리는 언어의 완전한 기술이다!

각 생성 규칙은 왼쪽에 있는 논 터미널 이름의 프로시저에 대응된다. 입력의 오른쪽에 있는 논 터미널은(가능한 재귀인) 프로시저 부름에 대응된다. 그리고 터미널은 입력 스트링을 주사하는 것에 대응된다. 예를 들면, 다음 프로시저는 정규식 문법에 대한 하향식 파서의 일 부분이다.

```
expression()
{
    term();
    if ( p[j] == '+' )
        { j++; expression(); }
}
```

스트링 p는 현재 조사가 되는 문자를 가르키는 인덱스 j로서 파서된 정규식을 포함한다. 주어진 정규식 p를 파서하기 위해서, j를 0으로 세트하고 expression을 부른다. 만약 이것은 j가 M으로 세트되어지는 결과가 되면, 그때 정규식은 문법에서 기술이 되는 언어내에 존재를 한다. 만약 그렇지 않으면, 여러 가지 오류 조건들이 어떻게 처리가 되는 지를 아래에서 보게 된다. expression이 수행하는 첫 번째 것은 다소 더 복잡한 구현으로 되는 term을 부르는 것이다.

```
term()
{
    factor();
    if ( ( p[j] == '(' ) || letter(p[j]) ) trem();
}
```

문법에서의 직접적인 번역은 단순히 term으로 하여금 factor를 부르고 나서 term을 부른다. 이것은 term으로부터 나오는 방법이 없기 때문에 작동되지를 않는다. 즉, 이같은 프로그램이 불려지는 경우에 무한적인 반복으로 된다.(그런 반복들은 많은 시스템에서 특별히 불

쾌한 효과를 지니다) 위의 구현은 term이 불러지는 것을 결정하기 위해서 입력을 먼저 체크함으로서 이것을 피하게 된다. term이 수행하는 제일 먼저 것은 입력에서 불일치를 감지할 수 있는 프로시저중의 하나인 factor를 부르는 것이다. 문법에서 factor가 불러질 때, 현재 입력 문자는 “(”이거나 혹은 입력 문자(v로서 표현)중의 하나이다. 무엇을 할 것인가를 결정하기 위해서 j를 증가시키지 않고 다음 문자를 체크하는 과정을 미리보기(lookahead)라고 부른다. 어떤 문법에 대해, 이것은 반드시 필요한 것은 아니다. 그리고 다른 것들에 대해, 훨씬 더 많은 미리보기를 요구한다.

지금 factor의 구현은 문법에서 직접적으로 따른다. 만약 조사되어진 입력 문자가 “(”나 입력 문자가 아닌 경우, 프로시저 error은 오류 조건을 처리하기 위해서 부른다.

```
factor()
{
    if ( p[j] == '(' )
    {
        j++; expression();
        if ( p[j] == ')' ) j++; else error();
    }
    else if ( letter(p[j]) ) j++; else error();
    if ( p[j] == '*' ) j++;
}
```

)”가 빠졌을 때 또 다른 오류 조건이 발생된다.

expression, term과 factor 함수들은 명백히 재귀적이다. 사실 그것들은 서로 뒤엎겨서 각 함수가 그것을 이용하기 전에 선언되도록 하기 위해서 그것들을 나열하는 방법을 없게 된다.(이것은 어떤 프로그래밍 언어에 대한 문제를 제시한다)

주어진 스트링에 대한 파서 트리는 파싱 동안에 재귀 부름 구조를 주는 것이다. 그림 21.2는 p가 (A*B+AC)D를 포함하고 그리고 expression이 j=1로서 부를때에 위의 세 가지 프로시저의 연산을 통해서 추적을 한 것이다. + 기호는 제외하고, 모든 “주사”는 factor에서 이루어진다. 읽기 가능성에 대해, 괄호에 대한 것은 제외하고 프로시저 factor를 주사하는 문자들은 factor 부름과 같은 라인 상에 놓여지게 된다.

여러분은 그림 21.1에 있는 트리와 문법에 대해 이같은 과정을 관련시킨 것을 알 수가 있다. 이같은 과정은 비록 미리보기 기법이 본질적으로 문법을 변경시키도록 장착이 되었기 때

```

expression
  term
    factor
      (
        expression
          term
            factor      A      *
            term
              factor    B
            +
          expression
            term
              factor    A
              term
                factor   C
            )
        term
          factor    D
      )

```

그림 21.2 (A*B+AC)D를 파싱

문에 대응이 정확하지는 않지만 프리 오더로서 트리를 운행하는 것에 대응된다. 트리의 제일 위에서 시작을 해서 밑으로 처리를 해 나가므로서, “하향식” 이름의 원조가 생겨나게 된 것이다. 또한 그런 파서를 재귀적으로 파서 트리를 아래로 이동시키기 때문에 재귀적-하강 파서(*recursive-descent parser*)라고 부른다.

하향식 방법은 모든 가능한 문맥 자유 문법에 대해서 처리가 되지 않는다. 예를 들면, 생성 규칙 $\langle expression \rangle ::= v \mid \langle expression \rangle + \langle term \rangle$ 으로 만약 위에서 처럼 C++로 기계적인 번역을 따르면, 바라지 않는 결과를 얻게 된다.

```

badexpression();
{
  if ( letter(p[j]) ) j++; else
  {
    badexpression();
    if ( p[j] == '+' ) { j++; term(); }
    else error();
  }
}

```

이같은 프로시저가 비문자인(예제에서 $j = 1$ 에 대한 것) $p[j]$ 로 부르면, 무한 재귀 반복으로 되게 된다. 그런 반복을 피하기 위해서 재귀적-하강 파서의 구현에서 주된 어려움이다. term에 대해, 그런 반복을 피하기 위해서 미리보기를 이용한다. 이같은 경우 문제를 회피하기 위한 적절한 방법은 문법을 가령 $\langle term \rangle + \langle expression \rangle$ 으로 변경을 시키는 것이다. 그 자체에 대한 대체 규칙의 오른쪽편상에 있는 첫 번째 항목으로서 는 터미널의 재 발생을 왼쪽 재귀(left recursion)이라고 부른다. 실제로 왼쪽 재귀는 간접적으로 발생이 되기 때문에 더 미묘하다. 예를 들면, 생성 규칙 $\langle expression \rangle ::= \langle term \rangle$ 과 $\langle term \rangle ::= v \mid \langle expression \rangle + \langle term \rangle$ 로 된 것이다. 재귀적-하강 파서는 그런 문법에 대해 작동을 하지 않는다. 즉, 그것들은 왼쪽 재귀 없이도 동등한 문법으로 변형을 할 수가 있거나 혹은 어떤 다른 파싱 방법이 이용될 수가 있다. 일반적으로 그것들을 인식하는 문법들과 파서들 사이에 연관성에 대해 친숙한 것이고, 상당히 널리 연구를 하고 그리고 파싱 기법의 선택은 파서된 문법의 특징들에 의해서 가끔 지시를 받는다.

상향식 파싱

비록 위의 프로그램에서 여러 가지 재귀적 부름이 존재하지만, 재귀를 체계적으로 제거시키는 것은 교훈적인 연습이 된다. 5장에서 회고해 보면 재귀를 구현하기 위해서 C++ 시스템이 무엇을 하는 가를 훑내내면서, 각 프로시저 부름은 스택 삽입으로 그리고 각 프로시저 되돌림은 스택 삭제로서 대체된다. 또한 이것을 수행하는 한 가지 이유는 재귀적으로 되어진 많은 부름이 진정으로 재귀가 아니라는 것을 회고할 것이다. 프로시저 부름이 프로시저 마지막 행동일 때, 단순한 goto가 이용된다. 이것은 expression과 term을 병합되고, 한 개의 실제적인 재귀 부름으로된 한 개의 프로시저를 나타내도록 하기 위해서 factor와 결합시키는 간단한 반복으로 바꾸는 것이다.(factor내에 expression에 대한 부름)

이같은 견해는 정규식이 합법적인가를 체크하기 위한 아주 간단한 방법으로 직접적으로 된다. 모든 프로시저 부름이 한 번 제거되면, 각 터미널 기호는 그것이 만나는 것과 같이 간단히 \주사된다. 수행된 유일한 실제 처리는 왼쪽 괄호와 일치하는 오른쪽 괄호가 있는지, 각 “+”는 한 문자나 “(” 중의 하나가 뒤따르는 것인지 그리고 각 “*”는 한 문자나 “)” 중의 하나가 뒤따르는 지를 체크하는 것이다. 즉, 정규식이 합법적인가를 체크하는 것은 균형적으로 이루어진 괄호에 대해 체크하는 것과 본질적으로 동일하다. 이것은 단순히 초기에 0으로된 계수기를 구현하면 된다 그리고 계수기(counter)는 왼쪽 괄호를 만날 때 증가가 하고 오른쪽

괄호를 만날 때 감소하게 된다. 만약 계수기가 식의 끝에서 0으로 되고 그리고 식에서 “+”와 “*” 기호는 위의 언급한 요구사항들을 만족시킨 경우에 식은 합법적으로 된다.

물론, 입력 스트링이 합법적인가를 단순히 체크하기 보다는 파싱에 대한 것이 더 많다. 즉, 주된 목적은 다른 처리에 대해 파서 트리를 생성하는 것이다.(비록 그것이 하향식 파서에서처럼 내재적인 방법내에 존재한다고 할지라도) 기술된 괄호를 체크하는 것과 같은 본질적인 구조로 된 프로그램들로서 이것을 수행하는 것은 가능하다고 판단된다. 이같은 방법으로 처리하는 파서의 한 형태를 소위 이동-감소 파서(shift-reduce parser)이라고 부른다. 개념은 터미널과 논 터미널 기호를 지니는 스택 삽입을 유지하는 것이다. 파서에서 각 단계는 스택상에 단순히 다음 문자를 삽입시키는 이동 단계이거나 혹은 스택상의 제일 상위의 문자는 문법에서 어떤 생성규칙의 오른쪽과 일치되는 것이고, 그 생성 규칙의 왼쪽 편에 논 터미널로서 “감소” 되는(대체된다) 감소 단계 중의 하나이다.(이동-감소 파서를 생성하는데 주된 어려움은 언제 이동하고 언제 감소하는 가를 결정하는 것이다. 이것은 문법에 따라서 복잡한 판단이 이루어진다) 결국에는 모든 입력 문자들이 스택으로 이동되고 결국에 스택은 하나의 논 터미널 기호로 감소된다. 먼저 식을 후위 표기식으로 바꾸므로써 중위 표기식에서 파서를 생성하는 것에 대한 3장과 4장에서의 프로그램들은 그런 파서의 간단한 예제를 나타낸 것이다.

상향식 파싱은 일반적으로 실제적인 프로그래밍 언어에 대한 선택의 방법이고, 프로그래밍 언어를 설명하는데 필요한 형태의 큰 문법에 대한 파서를 개발하는데 광범위한 문헌들이 존재를 한다. 간단한 기술로 인해서 포함이 된 문제들의 겉면만을 보게 된다.

컴파일러(Compilers)

컴파일러는 한 언어를 다른 언어로 번역하는 프로그램으로 간주를 한다. 예를 들면, C++ 컴파일러는 C++ 언어의 프로그램을 어떤 특별한 컴퓨터의 기계어로 번역을 한다. 정규식 패턴 일치 예제로 계속을 하므로써 이것을 수행하기 위한 한 방법을 제시한다. 그러나 지금 정규식의 언어를 패턴 일치 기계에 대한 언어 즉, 이전 장의 match 프로그램의 ch, next1과 next2 배열들로서 번역 하도록 하자.

번역 과정은 본질적으로 “1대 1”이다. 즉, 패턴에서 각 문자에 대해(괄호는 예외로서), 패턴 일치 기계에 대한 상태를 생성하기를 원한다.(배열들의 각각에서 한 엔트리) 여기의 기교는 next1과 next2 배열들에서 채워질 필요가 있는 정보를 파악하는 것이다. 그렇게 수행을 하기 위해서, 재귀-하강 파서에서 프로시저 각각을 패턴 일치 기계를 생성하는 함수로 바꾸는

것이다. 각 함수는 `ch`, `next1`과 `next2` 배열들의 끝으로 필요한대로 새로운 상태를 추가하고, 생성된 기계의 초기 상태의 인덱스를 되돌려 준다.(마지막 상태는 항상 배열에서 마지막 엔트리이다) 예를 들면, `<expression>` 생성 규칙에 대해 아래에서 주어진 함수는 패턴 일치 기계에 대한 *or* 상태들을 생성하는 것이다.

```
int expression()
{
    int t1, t2, r;
    t1 = term(); r = t1;
    if ( p[j] == '+' )
    {
        j++; state++;
        t2 = state; r = t2; state++;
        setstate(t2, ' ', expression(), t1);
        setstate(t2-1, ' ', state, state);
    }
    return r;
}
```

이같은 함수는 각각이 두 번째와 세 번째 그리고 네 번째 인수로서 주어진 값들에 대한 첫 번째 인수로서 인덱스된 `ch`, `next1`과 `next2` 배열 엔트리로 세트를 시키는 프로시저 `setstate`를 이용한다. 인덱스 `state`는 생성된 기계에서 “현재” 상태를 파악한다. 즉, 새로운 상태가 생성이 될 때마다, `state`는 증가된다. 이와같이 특별한 부름에 대응되는 기계에 대한 상태 인덱스들은 입구상에 `state`의 값과 출구상에 `state`의 값사이의 범주로 된다.(출구전에 `state`를 증가시키므로서 마지막 상태를 실제적으로는 “생성”하지 않는다. 그 이유는 보는 바와 같이 나중의 초기 상태들과 마지막 상태를 “병합”시키는 것을 쉽도록 하는 것이다)

이같은 규약으로서, 위의 프로그램은 이전 장에서 다이어그램화한 것과 같은 *or* 연산으로 된 두개의 기계를 구성하는 규칙을 구현하는 것이다. 첫째, 식의 첫 번째 부분에 대한 기계를 생성하는 것이다.(재귀적으로) 이때 두 개 새로이 비어있는 상태들이 첨가되고, 식의 두 번째 부분이 생성이 된다. 첫 번째 비어있는 상태(인덱스 `t2-1`로 된것)는 요구한 대로 식의 두 번째 부분에 대한 기계에 대해서 마지막 상태를 그냥 넘어가도록 하기 위해서 “연산이 없는(*no-op*)” 상태로 만드는 식의 첫 번째 부분의 기계중의 마지막 상태이다. 두 번째 비어있는

상태(인덱스 t2로서)는 초기의 상태이고 그래서 그것의 인덱스는 expression에 대한 되돌려지는 값이고, 그것의 next1과 next2 엔트리는 두 개 식들 가운데 초기의 상태들을 지적하도록 하는 것이다. 조심스럽게도 이것들은 기대했던 것에 대한 반대의 순서로서 구성이 됨을 주시해야한다. 그 이유는 연산이 없는 상태에 대한 state의 값은 expression에 대한 재귀 부름이 만들어질 때까지 알려진 것이 아니다.

<term>에 대한 함수는 <factor>에 대한 기계를 먼저 생성하고 필요한 경우, 그 기계의 마지막 상태와 또 다른 <term>에 대한 초기의 상태를 병합하는 것이다. 이것은 state가 factor에 대한 부름의 마지막 인덱스이기 때문에 말한 것보다 더 쉽게 이루어진다.

```
int term()
{
    int t, r;
    r = factor();
    if ( ( p[j] == '(' ) || letter( p[j] ) ) t = term();
    return r;
}
```

term에 대한 부름에 의해서 되돌려진 초기 상태 인덱스를 단순히 무시를 한다. 즉, C++는 그것을 어느곳엔가 놓도록 하는 것을 요구한다. 그래서 그것을 임시적인 변수 t에 놓는다.

<factor>에 대한 함수는 그것의 세 가지 경우를 처리하기 위해서 비슷한 기법을 이용한다. 즉, 이전 절에서 폐쇄 다이어그램에 따라서 괄호는 expression에서 재귀적인 부름에 대한 부름, v는 새로운 상태의 단순한 결합에 대한 부름이고 *는 expression에 있는 것들과 비슷한 연산들에 대한 부름이다.

```
int factor()
{
    int t1, t2, r;
    t1 = state;
    if ( p[j] == '(' )
    {
        j++; t2 = expression();
        if ( p[j] == ')' ) j++; else error();
    }
}
```



```

else if ( letter( p[j] ) )
{
    setstate( state, p[j], state+1, state+1);
    t2 = state; j++; state++;
}
else error();
if ( p[j] != '*' ) r = t2; else
{
    setstate( state, ' ', state+1, t2 );
    r = state; next1[t1-1] = state;
    j++; state++;
}
return r;
}

```

그림 21.3은 이전 장의 예제인 패턴 $(A*B+AC)D$ 에 대해 상태들이 어떻게 설정되는가를 나타낸 것이다. 첫째, 상태 1은 A에 대해서 구성을 한다. 상태 2는 폐쇄 오퍼랜드에 대해 구성을 하고, 상태 3은 B에 대해 부착된 것이다. 다음으로 “+”를 만나고 상태 4와 5는 expression에 의해서 설정되나 그것들의 영역들은 expression에 대한 재귀적 부름뒤에 까지 채워지지 않고 결국에는 상태 6과 7의 구성으로 되게 된다. 마지막으로 D의 결합은 상태 8로서 처리가 되고 마지막 상태로서 상태 9는 남겨둔다.

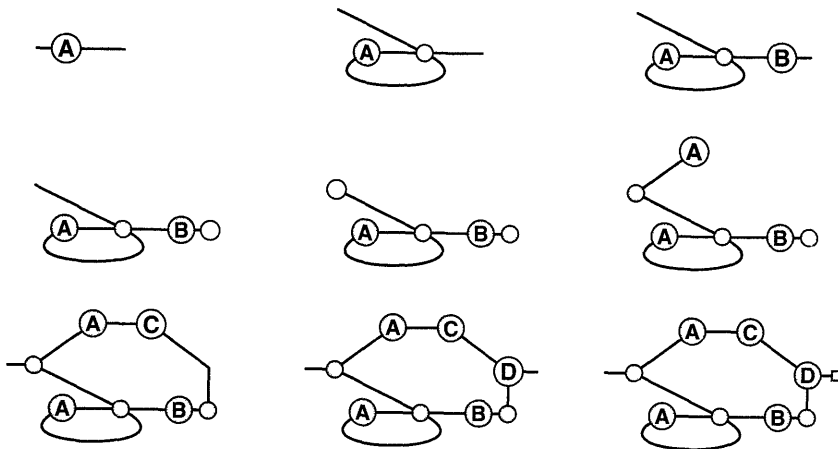


그림 21.3 $(A*B+AC)D$ 에 대한 패턴 일치 기계를 설정

일반적인 정규식 패턴 일치 알고리즘의 개발에서 마지막 단계는 match 프로시저와 함께 이같은 프로시저들을 놓는 것이다.

```
void matchall(char *a)
{
    j = 0; state = 1;
    next1[0] = expression();
    setstate(0, ' ', next1[0], next1[0] );
    setstate(state, ' ', 0, 0 );
    while ( *a ) cout << match(a++) << ' ';
    cout << '\n';
}
```

이 프로그램은 텍스트 스트링 a에서 각 문자 위치에 대해 패턴 p와 일치되는 그 위치에서 시작되는 가장 짧은 부분 스트링의 길이이다.(일치되지 않으면 0)

컴파일러-컴파일러

이것과 이전 장에서 개발된 일반적인 정규식 패턴 일치에 대한 프로그램은 효율적이고 아주 유용하다. 몇 가지 부가적인 즐거움들로서된("don't-care" 문자들 등과 같은 것을 처리) 이같은 프로그램의 버전은 많은 컴퓨터 시스템상에 가장 많이 이용되는 유틸리티들 가운데서 존재하는 것이다.

더 철학적인 관점에서 이같은 알고리즘에 영향을 주는 것은 흥미로운 것이다.(어떤 것은 혼동을 주지만) 이 장에서는 문맥 자유 문법을 이용해서 정규식의 공식적인 기술을 기반으로 한 정규식의 구조를 푸는 것에 대한 파서들을 고려한다. 다른 방법으로서, 특별한 "패턴"을 설명하기 위해서 문맥 자유 문법을 이용한다. 즉, 합법적으로 균형화된 괄호들로 된 연속적인 문자들이다. 그때 파서는 패턴이 입력에서 발생하는 가를 보기 위해서 체크를 한다.(그러나 전체 입력 스트링을 다루는 경우에만 합법적으로 일치하는가를 고려한다) 이와 같이 입력 스트링이 어떤 문맥 자유 문법에 의해서 정의된 스트링의 집합내에서 존재하는 파서들과 입력 스트링이 어떤 정규식에 의해서 정의된 스트링 집합내에 존재하는 패턴 일치자들은 본질적으로 같은 함수를 수행하는 것이다! 주된 차이는 문맥 자유 문법이 훨씬 더 넓은 스트링의 계층을 기술할 수가 있다. 예를 들면, 정규식은 모든 정규식들의 집합을 기술할 수는 없다.

프로그램에서 또 다른 차이는 문맥 자유 문법이 파서 “로 설정”이 되는 반면에 match 프로시저는 “표에서 유추된” 것이다. 즉, 한 번 그것들이 적절한 형식으로서 번역이 되므로서 같은 프로그램은 모든 정규식에 대해 처리를 한다. 같은 방법으로서 표에서 유추되는 파서들을 설정하는 것은 가능하다고 판명이 되므로서 같은 프로그램은 문맥 자유 문법에 의해서 기술되는 모든 언어들을 파서하는데 이용된다. 파서 발생기(parser generator)는 문법을 입력으로 취급을 하고 그 문법에서 기술되는 언어에 대한 파서를 출력으로서 생성하는 프로그램이다. 이것은 한 단계 더 실행하는 것이다. 즉, 입력과 출력 언어 둘다에 의해서 표에 유추되는 컴파일러를 설정할 수가 있다. 컴파일러-컴파일러는 두 개 문법들(그리고 그것들 사이에 관계들의 상세한 내용)을 입력으로 취하고 출력으로서 한 언어에서 다른 언어에 이르는 스트림들을 번역하는 컴파일러를 생성하는 것이다.

파서 발생기와 컴파일러-컴파일러는 많은 계산 환경하에서 일반적인 이용에 대해 이용이 가능하고, 상대적으로 적은 노력의 양으로서 효율적이고 신뢰있는 파서들과 컴파일러들로 번역을 하는 컴파일러를 생성하는데 이용되는 아주 유용한 도구들이다. 다른 한편으로는, 여기서 고려되는 형태의 하향식 재귀적-하강 파서들은 많은 응용에서 제기되는 간단한 문법에 대해서 아주 서비스적인 것이다. 이와같이 고려된 많은 알고리즘으로서 많은 구현의 노력들이 정당화되지 않는 응용에 대해서 적절한 것과 대규모 응용에 대해서 의미가 있는 정도의 활용도 개선점들로 이끌어지는 여러 가지 고급화된 방법들에서 간단한 방법들을 지닌다. 위에 나타낸 바와 같이, 이같은 광범위한 연구 영역의 표면을 단지 펼치게 된다.

연습문제

1. 불완전한 $(A+B)*BC$ 와 같은 정규식에서 오류를 재귀적-하강 파서가 어떻게 찾는가?
2. 정규식 $((A+B)+(C+D))*$ 에 대한 파서 트리는 어떻게 되는가?
3. 지수승, 나눗셈과 나머지(modulus) 연산자를 포함하는 산술식 문법을 확장하라.
4. 단지 두 개 연속적인 1로서 된 모든 스트링들을 기술하는 문맥 자유 문법은 무엇인가?
5. 결합, *or*와 폐쇄 연산의 수와 괄호의 수에 의해서 정규식을 인식하기 위해서 재귀적-하강 파서에 의해서 얼마나 많은 프로시저 부름이 이용되는가?
6. 패턴 $((A+B)+(C+D))*$ 에 대한 패턴 일치 기계를 설정하는 것에서 결과적으로 되는 `ch`, `next1`과 `next2` 배열들은 무엇인가?
7. “not” 함수와 “don’t-care” 문자들을 처리하기 위해서 정규식 문법을 어떻게 변경하는가?
8. 이전 답에 대한 해답에서 개선된 문법을 기본으로 하여 일반적인 정규식 패턴 일치를 설정하라.
9. 재귀적-하강 컴파일러에서 재귀를 제거하고, 가능한 많은 결과로 되어지는 코드를 간단히 하라. 비 재귀적인 것과 재귀적인 방법들의 실행시간을 비교하라.
10. 텍스트에서 문법에 의해서 기술된 간단한 산술식에 대한 컴파일러를 기술하라. 세 가지 연산을 할 수 있는 기계에 대한 “명령들”의 리스트를 생성하는 것이다. 즉, 변수의 값을 스택에 삽입(push), 스택의 제일 두 개 값을 덧셈, 그것들을 스택에서 제거, 그곳에 그 결과를 놓고 그리고 같은 방법으로 스택에 있는 제일 위 두 값을 곱셈을 한다.

빈 면

지금까지 공부해온 대부분의 알고리즘들은 먼저 수행시간을 짧게 하고 나서 가능하면 수행 공간도 적게 사용하도록 설계되었다. 이 장에서는 먼저 수행공간을, 그리고 가능하다면 수행 시간도 적게 사용하는 알고리즘들을 공부하기로 한다. 이러한 수행공간 절약을 위한 알고리즘들은 통신 시스템의 필요 정보량을 최소화하기 위해 개발된 정보 이론(information theory)으로부터 나온 “코딩”방식들로서 원래는 공간절약이 아니라 시간절약을 위해 만들어진 것이다.

일반적으로, 대부분의 컴퓨터 파일들은 상당량의 잉여 정보를 갖고 있다. 우리가 연구하려는 방법들은 대부분의 파일들이 상대적으로 적은 “정보량(information content)”을 가진다는 사실을 이용하여 시간을 절약한다. 파일 압축 기법들은 다른 파일들에 비해 특정 문자들이 아주 빈번히 나타나는 텍스트 파일, 동질성을 갖는 영역들을 많이 가질 수 있는 그림들의 코드화를 위한 “래스터(raster)”파일, 그리고 반복 패턴들을 많이 갖는 소리나, 다른 아날로그 신호들의 디지털화를 위한 파일들에서 자주 사용된다.

우선, 파일 압축 문제에 대해 유용한 기본 알고리즘을 살펴보고 나서 진보된 방법을 살펴보기로 한다. 이 방법들에 의해 저장될 수 있는 저장 공간의 양은 해당 파일의 특성에 따라 다르다. 통상적으로 텍스트 파일은 20%에서 50%이고, 이진(binary) 파일의 경우 50%에서 90%까지의 공간절약이 가능하다. 일부 파일의 경우, 예를 들면 랜덤(random)비트 들로 구성된 파일 같으면, 거의 공간절약이 되지 않는다. 사실상, 어떤 범용 압축 방법들은, 일부 파일들의 경우, 크기를 더욱 크게 만들 수 있음을 주목하라.

한편, 파일 압축 기법들은 이전에 비해 그 중요성이 감소됐다고 말할 수도 있다. 왜냐하면 컴퓨터 저장 장치들의 가격이 급속히 떨어졌고, 아주 큰 저장 장치들의 사용이 일상화되어 가고 있기 때문이다. 그러나 또 다른 한 편으로는, 파일 압축 기법들은 이제까지보다 더 중요

하다고 말할 수도 있다. 왜냐하면 아주 많은 저장 장치가 사용되어지고 있으므로 저장 공간이 보다 더 커지기 때문이다. 또한, 압축 기법들은 아주 빠른 액세스(access)가 가능하고 기본적으로 값이 비싼 저장 장치에서 사용될 때 아주 적합하다.

반복문자-길이의 코드화(Run-Length Encoding)

파일에서 가장 단순한 잉여 정보 형태는 길게 반복된 문자열이다. 예로서, 다음 문자열을 보자:

```
AAAABBBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD
```

이 문자열은 각 반복 문자열을 해당 문자와 그 발생 빈도로 대치함으로써 보다 간결하게 코드화 할 수 있다. 이 문자열은 4개의 A, 3개의 B, 2개의 A, 5개의 B 등의 순서로 나타남을 알 수 있다. 문자열을 이런 방식으로 압축하는 것은 반복 문자-길이 코드화(run-length encoding)라 한다. 반복문자열의 길이가 길수록 보다 많은 공간의 절약이 이루어진다. 응용 분야의 특성에 따라 작업을 할 수 있는 여러 가지 방법이 있을 수 있다.(반복문자열이 상대적으로 긴가? 얼마나 많은 비트들이 문자들의 코드화를 위해 사용되는가?) 우선 한 특정 방식에 대해 살펴본 후, 다른 방식들에 대해 설명하기로 한다.

만약 문자열이 단순히 글자(letter)들로만 구성되어 있음을 안다면, 각 글자 앞에 그 글자의 발생 빈도를 삽입함으로써 코드화할 수 있다. 이같이 할 때 문자열은 다음과 같이 코드화된다:

```
4A3BAA5B8CDABCB3A4B3CD
```

여기서 “4A”는 4개의 A를 의미한다. 글자 1개나 2개의 경우는 코드화할 가치가 없음에 주목하라. 왜냐하면, 코드화는 2개의 문자를 필요로하기 때문이다.

이진 파일들의 경우, 상당량 공간 절약을 위해 위 방법을 잘 다듬어서 만든 버전(version)이 통상적으로 사용된다. 이 방식은 반복 문자열들이 0과 1을 교대로 발생시키는 점을 이용하여 단순히 문자 반복 길이들을 저장하는 것이다. 이 방식은 짧은 문자 반복이 거의 없는 경우를 가정한다.(문자 반복의 길이가, 이진수로 표현할 때, 필요한 비트들의 수보다 많은 경우에만 비트들을 절약할 수 있다) 그러나, 어떤 문자 반복 길이 코드화 방식도 대부분의 반복 문자열들이 길지 않는 한 잘 동작하지 않는다.

반복문자-길이 코드화는 원래 파일과 그 코드화 된 버전간의 표현의 분리를 필요로 한다. 따라서 이 방식은 모든 종류의 파일들에 적용-가능하지는 않다. 이런 분리 방식은 상당히 불편할 수 있다. 예로서, 상기 제시된 문자 파일압축 방식은 숫자들을 갖는 파일 문자열들의 경우에는 동작할 수 없을 것이다. 만약 다른 문자들을 숫자를 코드화하기 위해 이용한다면, 이를 위해 사용된 문자들을 갖는 문자열들의 경우에 또한 동작 불가능하게 된다. 한 문자들의 집합에서 만들어진 임의의 문자열을 코드화하는 한 가지 방법을 설명하기 위해서, 단지 26개의 알파벳(그리고 공백 문자)만을 가지고 문자열을 구성한다고 가정하자.

[illegible]

22장 파일 압축 359

어떻게 임의의 글자(letter)들이 숫자(digit)들을 나타내고 또한 다른 글자들이 코드화될 문자(character)열의 부분들을 나타내도록 만들 수 있을까? 한 가지 해결책은 소위 확장문자(escape character)라고 불리는 텍스트에서는 거의 사용되지 않는 문자를 사용하는 것이다. 이 문자가 나타날 때마다 그 다음의 두 글자는(숫자, 문자) 쌍을 형성함을 알 수 있다. 물론, 알파벳의 i 번째 글자를 가짐으로써 표현되는 숫자들은 알파벳 문자의 개수를 나타낸다. 이같이, 예제의 문자열은 확장문자로서 Q를 가지고 표현될 때 다음과 같다.

QDABBBAAQEBQHCDABCBAAAQDBCCCD

확장문자, 숫자, 그리고 반복문자의 조합을 확장순서(escape sequence)라고 한다. 각 반복문자의 코드화에 적어도 세 문자가 필요하므로, 네 문자 미만의 반복문자는 코드화할 필요가 없음에 주목하라.

그러나 만약 확장문자 자체가 입력파일에 있으면 어떻게 할까? 이런 가능성을 단순히 배제 할 수는 없다. 왜냐하면 어느 특정 문자가 발생하지 않는다고 확신할 수 없기 때문이다. (예로서, 어떤 사람은 이미 코드화된 문자열의 코드화를 또 다시 시도할 수도 있다) 이 문제에 대한 한 가지 해결책은 확장문자의 표현을 위해 0의 숫자를 갖는 확장문자를 사용하는 것이다. 이같이 하면, 우리의 예제에서, 공백문자는 0을 나타낼 수 있고 확장순서 “Q<공백문자>”는 입력에 Q문자가 있음을 나타낼 수 있다. Q를 가지는 파일들만이 이 압축방법에서 더 길이가 길게 되는 경우가 발생함을 주목하라. 이미 압축된 한 파일을 또다시 압축한다면, 그 파일은 적어도 사용된 확장문자 수만큼 크기가 증가한다.

매우 긴 반복문자열은 다수의 확장문자를 가지고 코드화할 수 있다. 예를 들면, 51개의 A의 반복은 이전 방법에 따르면 QZAQYA로서 부호화될 수 있다. 만약 매우 많은 긴 반복문자열이 있을 경우에는, 그 반복 횟수를 코드화하기 위해 한 문자이상을 보존하여 사용하는 것이 좋을 수 있다. 실제로, 압축 및 확장 프로그램 모두를 어느 정도는 오류에 민감하게 만드는 것이 좋다. 이는 압축파일에 소량의 잉여 정보를 포함시킴으로서 가능한데, 그러면 확장프로그램은 압축과 확장시에 파일에 생기는 사고에 의한 사소한 변화 같은 문제들을 교정하여 원상복귀 시킬 수 있다. 예로서, 위 글자 “q”의 압축 버전에 “줄 종료”문자들을 삽입함으로써, 확장 프로그램은 오류 발생시 이 문자들을 가지고 재 동기화를 시도할 수 있다.

반복문자-길이 코드화 방식은 텍스트 파일들의 경우 그다지 효과적이지 않다. 왜냐하면, 반복될 확률이 많은 유일한 문자는 공백(blank)문자이고, 또한 반복된 공백문자들을 코드화하

기 위한 보다 단순한 방법들이 있기 때문이다.(이 방식은 과거 많은 공백문자들을 가질 필요가 없었던 텍스트 파일들의 압축에 아주 유용했다) 현대의 시스템들에서는 반복된 공백문자열들이 발생하지 않는다: 줄의 시작부에 있는 반복되는 공백문자들은 “탭(tab)”으로서 코드화 되고 줄의 종료부의 공백문자들은 “줄 종료”문자에 의해 표현된다. 위에 구현된 것 같은 반복문자-길이 코드화 방식은(그러나 모든 표현가능문자들을 처리할 수 있도록 수정된) 이 장을 구성하는 텍스트 파일의 경우, 약 4%의 절약을 보였다. 이 절약은 모두 문자 “q”의 경우에서 발생했다.

가변문자-길이 코드화(Variable-Length Encoding)

이 장에서는 텍스트 파일들의 경우 상당량의 공간을 절약해 주는 파일압축 기법을 알아보기로 한다. 주된 개념은 텍스트 파일들이 보통 저장되는 방식을 사용하지 않고, 즉, 각 문자당 7 또는 8비트를 사용하는 대신, 자주 나오는 문자들의 경우 자주 나오지 않는 문자들에 비해 더 작은 비트 수를 할당하여 표현하는 것이다.

어떻게 코드가 생성되는지를 알아보기에 앞서, 간단한 예를 가지고, 어떻게 코드가 사용되는지를 먼저 보는 것이 더 좋을 수가 있다. 문자열 “ABRACADABRA”를 코드화 하려고 한다 하자. 이 문자열을 알파벳의(공백문자는 0으로 표현) i 번째 글자를 표현하는 다섯 비트로 구성된 이진 표현으로 코드화하면 다음과 같은 형태가 된다:

0000100010100100000100011000010010000001000101001000001

이 메시지를 해독(decode)하려면, 단순히 한 번에 다섯 비트씩을 읽어서 위에 정의된 이진 코드에 따라 변환하는 것이다. 이 표준 코드에 따르면 D는 한 번만 발생하는데 다섯 번 발생하는 A와 같은 수의 비트를 필요로 한다. 가변문자-길이 코드화 방식에서는 자주 발생하는 문자들의 표현시 필요한 비트 수를 최소화함으로써 메시지의 전체 사용 비트 수를 줄이는 것이다.

가장 자주 사용되는 글자에 가장 작은 비트 수를, 예를 들면, A는 0, B는 1, R는 01, C는 10, 그리고 D는 11을 할당할 수 있다. 그러면, ABRACADABRA는 0 1 01 0 10 0 11 0 1 01 0 으로 코드화된다. 이 경우 단지 15비트만이 사용된다.(앞서의 경우, 55비트가 사용됐다) 그러나, 이 방식은 코드화가 아니다. 왜냐하면 문자들을 구분하기 위해 공백문자들을 사용했기

때문이다. 공백문자들을 없애면 문자열 010101001101010인데, 이는 RRRARBRRRA 또는 여러 다른 문자열로서 해독될 수 있다. 아직도 15비트와 10개의 공백 구분문자(delimiter)들을 사용함이 표준코드보다는 더 간략하다. 이의 주된 이유는 메시지에 나타나지 않는 글자들을 코드화하기 위해서 어떤 비트들도 사용하지 않기 때문이다. 공정한 비교를 위해서, 코드 자체에 있는 비트 수를 세어 볼 필요가 있다. 왜냐하면 메시지는 비트들 없이는 코드화될 수 없고, 코드 또한 메시지에 의존하여 만들어지기 때문이다.(다른 메시지들은 또 다른 문자 발생빈도를 가질 것이다) 이 문제에 대해서는 나중에 다루기로 하고 당분간은 얼마나 메시지를 간략하게 만들 수 있는 가에 대해서만 생각해보기로 한다. 첫째, 구분문자들은 어떤 문자 코드가 다른 문자 코드의 접두어(prefix)인 경우에는 필요치 않다. 예를 들면, 만약 A를 11, B는 00, C는 010, D는 10, 그리고 R은 011로 코드화했다면, 25비트문자열을 해독하는 방법은 유일하게 한 가지 뿐이다.

1100011110101110110001111

코드를 표현하는 한 가지 쉬운 방법은 트라이(trie) 구조를 이용하는 것이다.(17장을 보라) 사실상, M 개의 외부 노드(node)를 갖는 트라이 구조는 M 개의 다른 문자를 갖 메시지를 코드화하는데 사용될 수 있다. 예로서, 그림 22.2는 ABRACADABRA를 코드화하는데 이용할 수 있는 두 가지 형태를 보여주고 있다. 각 문자에 대한 코드는 그 문자의 루트(root)로부터의 경로(path)에 의해 결정된다.(트라이 좌측 방향 진행은 0으로, 우측방향진행은 1로서) 이같이, 좌측의 트라이는 상기 코드(1100011110101110110001111)에 해당하고, 우측 트라이는 문자열 01101001111011100110100을 산출하는 코드에 해당한다.

후자의 경우, 두 비트가 더 짧다. 트라이 표현은 어떤 문자 코드도 다른 문자 코드의 접두어가되지 않음을 보장한다. 그래서, 트라이에서의 문자열은 유일한 해독이 가능하다. 루트에서 시작하여 메시지의 비트들을 트라이구조를 따라 하향 진행 해보라. 한 외부노드가 마주칠 때마다, 그 노드에 있는 문자가 출력되고 루트에서부터 재시작하게 된다.

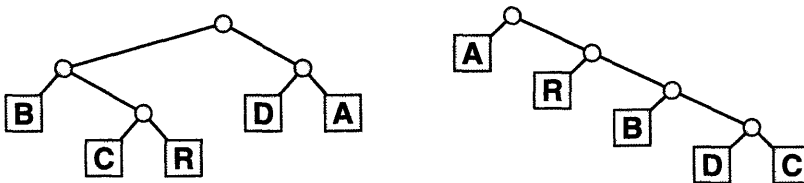


그림 22.2 A, B, C, D 및 R에 대한 두 트라이 코드화

그러면, 어느 트라이구조가 가장 사용하기 좋은가? 임의의 주어진 메시지에 대해 최초 비트 문자열 길이를 산출하는 트라이 구조를 찾을 수 있는 한 좋은 방법이 있음이 알려져있다. 이 방식은 D. Huffman에 의해 1952년에 발견되어졌으므로 *Huffman* 코드화 방식이라 불린다.(여기서 살펴볼 구현방식은 일부 진보한 알고리즘 기법을 사용한다)

Huffman 코드의 구성(Building the Huffman Code)

Huffman 코드를 구성하는 첫 번째 절차는 코드화될 메시지내의 각 문자의 발생빈도를 세는 것이다. 아래 프로그램은 문자열 *a*가 한 메시지에서 몇 번 발생하는가를 계산하여 *count*[26] 배열에 기록한다.(이 프로그램은 공백문자들은 *count*[0]에 그리고 알파벳의 *i* 번째 문자들은 *count*[*i*]에 그 발생빈도를 기록하기 위해, 19장에서 설명한 *index* 프로시저를 사용한다)

```
for (i=0; i<=26; i++) count[i]=0;
for (i=0; i<M; i++) count[index(a[i])]++;
```

예로서, 문자열 “A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS”를 코드화 하려 한다 하자. 그림 22.3에 보여지는 것과 같은 계수도표가 만들어진다; 여기에는 7개의 공백문자, 3개의 A, 3개의 B 등이 있다.

다음번 절차는 빈도에 따라 밑에서부터 코드 트라이를 만드는 것이다. 트라이 형성시, 트라이를 노드 내에 빈도가 저장된 이진 트리로서 간주할 수 있다. 이 만들어진 트라이를, 위 그림에 있는 것과 같은, 부호화를 위한 트라이로 간주한다. 첫째로, 각 빈도값이 0이 아닌 경우에 대해 그림 22.4의 첫 행의 좌측에 있는 것 같은 트리노드를 형성한다.(노드들이 나타나는 순서는 아래 설명된 알고리즘의 동적 구성에 의해 결정된다. 그러나, 이 문제는 지금 우리

	A	B	C	D	E	F	G	I	L	M	N	O	P	R	S	T	U	
k	0	1	2	3	4	5	6	7	9	12	13	14	15	16	18	19	20	21
count[k]	11	3	3	1	2	5	1	2	6	2	4	5	3	1	2	4	3	2

그림 22.3 A SIMPLE STRING TO BE EN... 에 대한 빈도값

가 말하고자 하는 바와 별상관이 없다) 둘째로, 가장 빈도가 작은 두 노드를 찾아서 이 둘을 자식노드로 갖는 한 새로운 노드를 만들고, 이 노드에 두 자식노드들이 가진 빈도값의 합을 부여한다. 그림 22.4의 첫 줄의 우측에 이것이 설명되어있다.(만약 최소 빈도값을 갖는 노드가 둘 이상이라면, 어느 것을 먼저 선택하든 상관이 없다) 그리고 나서, 포리스트(forest)에서 최소 빈도를 갖는 두 노드들을 다시 찾아서, 그림 22.4 두 번째 행의 좌측에서 보여지는 것처럼, 새 노드를(이전과 동일한 방식으로) 만든다. 이렇게 계속하면, 서브트리들이 점점 크게되는 동시에 포리스트 내의 트리수는 매 스텝마다 하나씩 감소한다.(두개가 제거되고, 한 개가 추가된다) 궁극적으로, 모든 노드들이 단일 트리를 형성하게 된다.

아주 낮은 빈도를 갖는 노드들은 트리상에서 거의 하위부에 있게되고 아주 높은 빈도를 갖는 노드들은 트리의 루트에 인접하여 위치하게 된다. 이 트리에서 외부(사각형)노드를 표시하는 숫자는 빈도수이고, 각 내부(원형)노드내의 숫자는 두 자식노드의 합이다. 이제, 하부 노드들의 빈도값을 해당 글자 값으로 바꾸고, 트리를 좌측은 코드 비트 0, 우측은 코드 비트 1인 코드화된 트라이로 간주하여 Huffman 코드가 유도되었다. 명백히, 높은 빈도를 갖는 글자 값은 트리의 루트에 인접하게 되고, 보다 작은 비트수로 코드화되므로 이 코드는 좋은 코드이다. 그러나 왜 최적코드인가?

성질 22.1 코드화된 메시지의 길이는 Huffman 빈도 트리에서 가중치가 부여된 외부경로의 길이와 동일하다.

한 트리의 “가중치가 부여된 외부 경로 길이”는 모든 외부노드의 가중치(부여된 빈도값)과 루트로부터의 거리의 곱이다. 명백히, 이것은 메시지 길이를 계산하는 한 방법이다: 모든 문자들에 대해, 발생 개수와 발생문자당의 비트수의 곱을 더한 것과 동일하다. □

성질 22.2 외부노드들에서 동일 빈도를 갖는 어떤 트리도 Huffman 트리보다 더 낮은 가중치 값을 갖는 외부 경로길이를 갖지 않는다.

Huffman 트리를 만들 때 사용한 것과 동일한 과정으로 한 트리를 재구성할 수 있다. 그러나, 매 스텝마다 최소가중치를 갖는 두 노드를 선택할 필요는 없다. 두 개의 최소 가중치를 갖는 노드를 먼저 선택하는 것보다 좋은 방식이 없음은 귀납법적으로 증명할 수 있다. □

한 노드가 선택할 때마다, 동일한 가중치를 갖는 여러 노드가 있을 수 있다. Huffman 방식은 동일한 값을 갖는 노드중 어떤 것을 먼저 선택할지를 규정하지는 않는다. 선택방식의

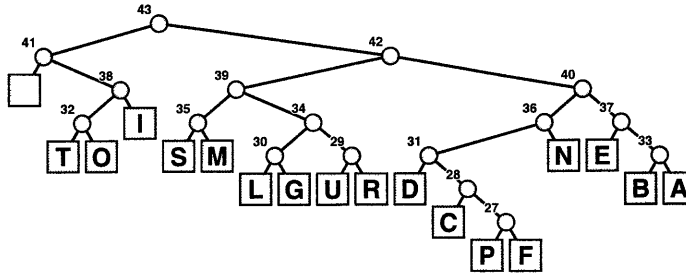


그림 22.5 A SIMPLE STRING TO...에 대한 Huffman 코딩 트라이

차이에 따라 다른 코드들이 나올 수 있다. 그러나 이러한 모든 코드들은 모두 같은 수의 비트들을 가지고 메시지를 코드화한다. 예로서, 앞서 다른 예제에 대한 또 하나의 트라이가 그림 22.5에 있다. A에 대한 코드는 1111, B에 대해서는 1110, C에 대해서는 110010 등. 이 트리는 그림 22.4에 구성된 트리와는 구조적으로 다르다. 독자는 이들이 동일한 가중치를 갖는 외부경로 길이를 가짐을 입증하기를 원할 수도 있다.(위 트리상의 각 노드에서 작은 숫자값은 아래 구현조사시 참조를 위한 인덱스(index)이다)

이제까지 Huffman 코드화를 계산하는 방법을 우리가 공부해온 알고리즘적 동작에 입각하여 설명했다. 평상시와 마찬가지로, 이와 같은 실제 구현에 대한 설명 스텝은 아주 유익했다. 이제는 구현에 대한 자세한 사항들을 살펴보기로 한다.

구현(Implementation)

빈도값들을 갖는 트리의 형성 과정은 순서 없이 나열된 원소값들의 집합에서 최소값을 제거하는 일반적인 스텝을 포함한다. 그래서, 빈도값에 대한 우선 순위 큐(priority queue)를 만들고 유지하기 위해 11장에서 PQ클래스를 이용하기로 한다. 이 우선 순위 큐를 사용하여 위에서 설명한 트리를 작성하는 코드는 다음과 같다:

```
for(i=0; i<=26; i++)
    if(count[i]) pq.insert(count[i], i);
for( ; !pq.empty(); i++)
{
    t1 = pq.remove(); t2 = pq.remove();
    dad[i] = 0; dad[t1] = i; dad[t2] = -i;
```

```

count[i] = count[t1] + count[t2];
if(!pq.empty()) pq.insert(count[i], i);
}

```

먼저 우선 순위 큐에 모든 0이 아닌 값들을 삽입한다. 그리고 그 중에서 가장 최소값을 갖는 원소들을 두 개 끄집어내어 더하고 그 결과를 다시 큐에 집어 넣는다. 이 스텝을 큐에 아무것도 남지 않을 때까지 계속한다. 각 스텝에서, 매번 새로운 값이 만들어지고 우선 순위 큐의 원소 수는 하나씩 감소한다. 이 프로세스는 만들어진 트리상의 내부노드 각각에 대해 한 개씩, 즉 모두 $N-1$ 개의, 새로운 값을 만들어 낸다. 인덱스값 i 는 count배열을 계속하여 참조한다. 그래서, 첫 번째 내부노드는 인덱스값 27을 갖는다. i 를 1씩 증가시킴으로써 새로운 내부노드가 만들어진다. 트리자체는 “부모”와의 “연결(link)”들의 배열로서 표현된다. $dad[t]$ 는 가중치가 $count[t]$ 내의 가중치를 갖는 노드의 부모에 대한 인덱스이다. $dad[t]$ 의 음/양 값은 노드가 그 부모의 좌측 자식인지 우측 자식인지를 나타낸다. 그림 22.6은 그림 22.5의 트리에 대해 이러한 데이터구조의 전체 내용을 보여준다. 예를 들면, $count[40]=21$, $dad[40]=-42$, 그리고 $count[42]=37$ 을 갖고 있다. 여기서, 가중치 21인 노드는 인덱스 0140이며, 인덱스 4201이고 가중치가 37인 부모의 우측 자식이다.

트라이 구조는 코드 그 자체를 설명하기에 충분하다. 이 점을 확실히 하기 위해, 코드를 어떻게 명백히 구성하는지를 살펴보자. 그림 22.7은 우리 예제에 대한 전체 코드인데 2차원 배열로서 표현됐다. 정수 $code[k]$ 의 이진표현인 최우측 $len[k]$ 비트들은 k 번째 글자에 대한 코드이다. 예를 들면, I는 9번째 글자이고 코드로서 011을 갖는다. 따라서 $code[9]=3$ 이고 $len[9]=3$ 인데, 이들은 코드가 숫자 3의 이진표현을 위한 최우측의 3비트들(즉, 011)임을 나

k	0	1	2	3	4	5	6	7	9	12	13	14	15	16	18	19	20	21
count[k]	11	3	3	1	2	5	1	2	6	2	4	5	3	1	2	4	3	2
dad[k]	41	-33	33	28	31	37	-27	-30	-38	30	-35	-36	-32	27	-29	35	32	29

k	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
count[k]	2	3	4	4	5	6	6	8	8	10	11	12	16	21	23	37	60
dad[k]	-28	-31	-34	34	36	38	-37	-39	39	40	-40	-41	42	-42	43	-43	0

그림 22.6 그림 22.5의 Huffman 트라이의 부모와의 연결 표현

		A	B	C	D	E	F	G	I	L	M	N	O	P	R	S	T	U
k	0	1	2	3	4	5	6	7	9	12	13	14	15	16	18	19	20	21
code[k]	0	31	30	50	24	14	103	21	3	20	9	13	5	102	23	8	4	22
len[k]	2	5	5	6	5	4	7	5	3	5	4	4	4	7	5	4	4	5
	00	11111	11110	110010	11000	1110	1100111	10101	011	10100	1001	1101	0101	1100110	10111	1000	0100	10110

그림 22.7 A SIMPLE STRING TO BE...에 대한 Huffman 코드

타내고 있다. 아래 프로그램은 코드의 트라이표현(그림 22.6에 있는 것같은 dad배열)을 이러한 표현식으로 변환해준다.

```
for (k = 0; k <= 26; k++)
{
    i = 0; x = 0; j = 1;
    if (count[k])
        for (t=dad[k]; t ; t = dad[t], j+=j, i++)
            if (t<0) { x+=j; t=-t }
    code[k] = x; len[k] = i;
}
```

code와 len 배열들은 dad배열을 이용하여 트리를 거슬러 올라가면서 작업하여 쉽게 계산된다.

메시지는, 자신을 구성하는 각 문자에 대해, 이런 방식으로 트라이를 운행함으로써 코드화될 수 있다. 또 다른 방법으로서, 메시지 코드화를 위해 code와 len배열을 직접 사용할 수도 있다.

```
for (j = 0; j<M; j++)
    for (i = len[index(a[j])]; i>0; i--)
        cout << ((code[index(a[j])]) >> i-1) & 1);
```

우리의 샘플 메시지는 단지 236비트로서 코드화되는데, 300비트를 사용하는 직접 코드화 방식에 비해 21%의 절약이 가져다준다.

```
111110010000111001110011010100111000100001001011101110110101000100
0101001111011100011101101110010010111000111011000001011010000111101
101010011111001001011110101110011111101000011011011010011111011101
01110001011100111001111001101001000
```

이제, 앞서 언급한 것처럼, 트리는 메시지에 따라 메시지를 해독하기 위해 저장되거나 보내져야한다. 운 좋게도, 이렇게 하는데 실제적으로 아무런 어려움이 없다. code 배열을 저장하는 것만이 필요하다. 왜냐하면, 그 배열에 있는 엔트리를 빈 트리에 삽입함으로써 생기는 기수 검색(radix search) 트라이는 부호해독 트라이이기 때문이다.

이같이, 위에서 언급한 저장 장소의 절약은 아주 정확하지는 않다. 왜냐하면 메시지는 트라이 없이는 해독될 수 없고, 또한 메시지에 따라 트라이(즉 code 배열)를 저장하는데 드는 비용을 고려해야하기 때문이다. 그러므로, Huffman 코드화는 메시지를 절약하는 것이 그 소요비용보다 중요한 경우인 파일들, 또는 많은 메시지에 대해 코드 트라이가 미리 계산되고 사용될 수 있는 경우의 파일들에만 유용하다. 예를 들면, 영어에서 문자의 발생빈도에 근거한 트라이는 텍스트 문서에 사용될 수 있다. 이 경우 C++프로그램의 문자의 발생빈도에 근거한 트라이는 프로그램을 코드화하는데 사용될 수 있다.(예로서, “;”는 그 같은 트라이의 거의 윗 부분에 있게된다) Huffman 코드화 알고리즘을 이 장의 텍스트에 적용했을 때, 약 23%의 절약을 가져왔다.

전과 마찬가지로, 진짜로 랜덤한 파일의 경우에는, 이 코드화 방법도 잘 동작하지 않을 것이다. 왜냐하면, 각 문자가 거의 동일한 수 만큼 발생하며 완전히 균형을 이룬 코드 트리가 만들어지고 코드에서 각 글자 표현시 같은 수의 비트를 사용하게 되기 때문이다.

연습문제

1. Q를 확장문자로 사용하여, 본문에서 설명한 고정된 알파벳에 대해 반복문자-길이 코드화에 대한 압축 및 확장 프로시저들을 구현하라.
2. 본문에서 설명한 방법에 의해 압축된 파일의 어디엔 가에 “QQ”가 발생할 수 있을까? 또한 “QQQ”가 발생할 수 있을까 ?
3. 본문에서 설명한 이진 파일 코드화에 대한 압축 및 확장 프로시저들을 구현하라.
4. 본문에 있는 글자 “q”는 5 비트로 구성된 일련의 문자들로서 처리될 수 있다. 문자에 근거한 반복문자열-길이 코드화 방법을 사용하기 위해 이렇게 처리할 경우 찬반 양론을 논하라.
5. 본문에서 사용된 방법을 문자열 “ABRACADABRA”에 대한 Huffman 코딩 트리를 만드는데 사용했을 때의 형성과정을 보여라. 코드화된 메시지는 얼마나 많은 비트들을 필요로 하는가?
6. 이진 파일에 대한 Huffman 코드는 무엇인가? N 문자 3원치(three-valued) 파일에 대해 Huffman 코드에서 사용될 수 있는 최대 비트수를 보여주는 예를 들라.
7. 코드화될 모든 문자들의 발생 빈도가 다르다고 하자. 이 경우, Huffman 코딩 트리는 유일한가?
8. Huffman 코딩은 2 비트 문자들을(4 방향 트리를 사용하여) 코드화 할 수 있도록 쉽게 확장 가능하다. 이렇게 했을 경우의 주된 장점들과 단점들을 설명하라.
9. Huffman 코드로 코드화된 문자열을 5 비트 문자들로 나눈 후 거기에 다시 Huffman 코드화를 시도하면 어떤 결과가 나올까 ?
10. code와 len배열이 주어졌을 경우, Huffman 코드화된 문자열을 해독하는 프로시저를 구현하라.

23 장

암호학

전 장에서 저장 공간의 절약을 위해 문자열을 코드화하는 방법들을 살펴보았다. 물론 문자열들을 코드화하는 또 한 가지의 중요한 이유는 이들의 비밀 보장을 하는 것이다.

암호학(Cryptology)-비밀 통신을 위한 시스템들의 연구-는 두 상호보완적인 분야로 구성된다: 하나는 비밀 통신의 설계를 위한 암호 작성법(Cryptography)이고 또 다른 하나는 비밀 통신 시스템들의 해결 방법들의 연구인 암호해독법(Cryptanalysis)이다. 암호학은 주로 군사 및 정치적 용도의 통신 시스템들에 주로 응용되어 왔으나 이 학문의 적용을 필요로 하는 다른 응용 분야들이 많이 생기고 있다. 이러한 두 중요 분야는(각 사용자가 자신을 파일들을 비밀로 유지하기를 원하는) 컴퓨터 파일 시스템 분야와(아주 많은 돈을 다루는) 전자식 자금 이전(funds transfer) 시스템 분야이다. 컴퓨터 사용자는 자신의 컴퓨터 파일들을 자신의 금고에 보관된 서류들처럼 가능한 한 비밀로서 보관하고 싶어하며 은행은 전자식 자금 이전이 마치 무장된 수송 차량에 의한 자금 이전처럼 보안이 되기를 바란다.

군사 분야를 제외한 모든 암호 작성자들은 좋은 사람들이고 암호 해독자들을 나쁜 사람들이라고 가정한다: 목표는 우리의 컴퓨터 파일들과 은행 계좌들을 범죄자들로부터 보호하는 것이다. 만약 이러한 견해가 다소 비우호적이라면, 암호 작성자들이 비우호적인 나쁜 사람(암호 해독자)들의 존재를 가정하고 있음을 주목해야 한다. 물론, 좋은 사람들은 암호 해독에 관한 지식을 가져야만 한다. 왜냐하면 한 시스템이 안전할 수 있는 가장 좋은 방법은 자신이 시스템의 비밀해독 방법을 아는 것이기 때문이다.(또는 암호 해독을 함에 의해 전쟁들이 끝나고 많은 생명들이 살게 된 여러 문서화된 사례들이 있다)

암호학은 컴퓨터 과학 및 알고리즘 분야와 특히 이제까지 공부한 산술 및 문자열 처리 알고리즘과 많은 밀접한 관련이 있다. 실제로 암호 기술(과학?)은 컴퓨터 및 이제 막 이해되어지기 시작한 컴퓨터 과학과 밀접한 관계가 있다. 알고리즘들의 경우처럼, 암호 시스템(Cryptosystem)들은 컴퓨터보다 훨씬 오래 전부터 존재해 왔다. 암호 시스템의 설계와 알고리즘의 설계는 그 근원이 같고 같은 사람들이 이 두 가지 모두에 관심을 가져왔다.

암호학의 어느 부분이 컴퓨터가 생김으로 인해 가장 많이 영향을 받았는지는 확실치 않다. 암호 작성자들은 전보다 아주 강력한 암호화 기계들을 사용할 수 있으나 또한 그림으로써 실수를 할 수 있는 여지가 더 많아졌다. 암호 해독자는 암호를 해독하는데 있어서 전보다 훨씬 더 강력한 도구들을 가지게 되었다. 그러나 이러한 암호의 해독은 더욱 어려워지게 됐다. 암호 해독은 컴퓨터 같은 계산 자원들 상에 커다란 영향을 준다: 암호 해독은 초기 컴퓨터 응용 분야 중의 하나일 뿐만 아니라 현재 수퍼 컴퓨터들에 대한 한 주된 응용 분야로서 존재한다.

최근에 컴퓨터가 널리 사용됨으로써 위에서 언급한 것처럼 암호학이 필요한 새롭고 중요한 응용 분야들이 나타나게 되었다. 이런 분야들에 적합한 암호 작성법들이 최근에 개발되고 있으며 이러한 방법들은 암호학과 45장에서 간략하게 살펴볼 컴퓨터 과학 이론의 한 중요한 분야간의 근본적인 관계를 발견하게 해 준다.

이 장에서는, 암호 작성 알고리즘들의 일부 기본적인 특성들을 살펴보기로 한다. 물론 여기서는 자세한 구현 사항들을 설명하지 않기로 한다: 암호학은 분명히 이 분야의 전문가들이 깊이 다뤄야 할 분야이다. 한 단순한 암호 작성 알고리즘을 이용하여 정보들을 암호화하여 사람들이 이들을 알 수 없게 만드는 것은 어렵지 않지만, 비전문가가 만든 암호 작성법을 사용하는 것은 위험한 일이다.

게임 규칙(Rules of the Game)

두 사람간의 안전한 통신을 위한 수단을 제공하는 요소들을 통칭하여 암호 시스템(Cryptosystem)이라 한다. 전형적인 암호 시스템의 표준 구조를 그림 23.1에서 볼 수 있다. 송신자는 한(plaintext라고 불려지는) 원문 메시지를 암호작성 알고리즘(암호화 기법)과 암호 해독 키(key) 파라미터들을 이용하여 비밀 전송에 적합한 암호문(ciphertext)으로 변환시켜서 수신자에게 보낸다. 이 메시지를 읽으려면 수신자는 송신측의 암호 작성 알고리즘과 매

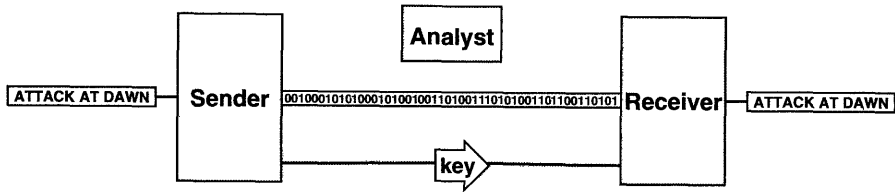


그림 23.1 전형적 암호 시스템

치(match)되는 암호해독 알고리즘(암호해독 기법)과 동일한 키 파라미터들을 가져야 한다. 그러면 암호문을 다시 원문-즉 메시지-로 환원시킬 수가 있다. 우리는 보통 암호문이 보안성이 없는, 그래서 암호 해독자가 액세스 할 수 있는, 통신선을 통해 전송된다고 가정한다. 또한 통상적으로 암호화 기법과 암호해독 기법 모두를 암호 해독자가 안다고 가정한다: 암호 해독자의 목표는 키 파라미터들 없이도 암호문을 해독하여 원문을 찾아내는 것이다. 이 암호 시스템은 송신자와 수신자가 서로 같은 키값을 갖게 해주기 위한 별도의 중요한 통신 방법에 의존함에 주목하라. 일반적으로, 키 파라미터들이 많을 수록 암호 시스템은 더욱 안전해지나 사용하기에는 더 불편해진다. 이 상황은 많은 기존의 보안 시스템들의 경우와 유사하다: 숫자-맞추기 금고(combination safe)는 그 자물쇠에 더 많은 숫자들을 이용할수록 더욱 안전해지나 그 조합 번호는 기억하기가 더 어려워진다. 알아야 할 것은 어떤 보안 시스템도 키를 갖고 있는 사람이 가질 수 있는 만큼의 보안성만 가진다는 것이다.

암호 시스템에서 비용 문제가 중요한 역할을 함을 기억함이 중요하다. 간단한 암호화 장치와 해독 장치가 필요한 이유가 있다.(왜냐하면 이러한 장치들은 많이 필요하며 이들이 복잡하면 비용이 많이 들기 때문이다) 또한, 분배되어야 할 키 정보의 양을 줄여야 할 이유도 있다.(왜냐하면 아주 안전하나 비싼 통신 방법이 사용 되어 하기 때문이다) 암호작성 알고리즘의 구현 비용과 키 정보의 분배 비용간의 차액이 암호 해독자가 암호 시스템을 깨뜨리기 위해 기꺼이 지불할 수 있는 돈의 양이다. 대부분의 분야에서, 암호 작성자의 목표는 암호 해독자가 지불하려는 것보다 아주 많은 메시지들의 판독 비용을 지불해야 하는 성질을 갖는 값싼 시스템을 개발하는 것이다. 일부 응용 분야들의 경우에는, 암호 해독자가 어떤 노력을 하든지 간에 메시지들을 전혀 판독할 수 없음을 보장하는 안전성을 증명할 수 있는(provable secure) 암호 시스템이 요구된다.(암호학의 일부 응용 분야들에서의 아주 큰 위험성은 암호 해독을 위해 엄청난 돈이 사용되는 것을 뜻한다) 알고리즘의 설계시에 최선의 알고리즘을 선택하기 위해 설계 비용을 계산해 봐야 한다; 암호학에서는, 암호 작성 비용이 설계 과정에서의 중심 역할을 한다.

간단한 방법들(Simple Methods)

암호화를 위한 가장 간단하고(가장 오래된) 방법들 중의 하나가 Caesar 암호화 기법이다: 만약 원문의 한 글자가 알파벳에서 N 번째 글자라면, 이 글자를 알파벳 내의 $(N + K)$ 번째 글자로 대체한다; 여기서 k 는 한 고정 상수이다.(Caesar는 $k = 3$ 을 사용했다) 아래 표는 $k = 1$ 일 때 어떻게 한 메시지가 이 방법으로 암호화되는지를 보여준다:

원문: ATTACK AT DAWN
암호문: BUUBDLABUAEBXO

이 방법은 쉽게 해독될 수 있다. 왜냐하면 암호 해독자는 단지 k 값을 찾으면 되기 때문이다: 26개의 글자 각각에 대해 시도함으로써 메시지를 판독할 수가 있다.

훨씬 더 좋은 방법은 치환(substitution)을 정의하는 한 일반적인 도표를 사용하는 것이다: 원문의 각 글자에 대해, 이 도표는 암호문에 치환되어 들어갈 글자를 알려준다. 예로서, 만약 이 도표가 알파벳에 대해

ABCDEFGHIJKLMNOPQRSTUVWXYZ
THE QUICKBROWNFXJMPDVRLAZYG

의 관계가 있다면, 메시지는 다음과 같이 암호화된다:

원문: ATTACK AT DAWN
암호문: HVVH OTHVTQHAF

이 방식은 단순한 Caesar 암호 방식보다 훨씬 더 강력하다. 왜냐하면 암호 해독자는 메시지 판독을 위해 아주 많은 개수의 도표들 (약 $27! > 10^{28}$) 을 가지고 시도해야만 하기 때문이다. 그러나 이같이 단순한 치환에 근거한 암호들은 해당 언어 고유의 글자의 발생 빈도에 따라 쉽게 판독되기 때문이다. 예로서, 글자 E는 영문에서는 가장 많이 발생하기 때문에, 암호 해독자는 암호문에서 가장 많이 발생하는 문자를 찾아 E로 대체한 후 해독을 시작할 수 있다. 물론 이것이 바른 선택이 아닐 수는 있지만 아무것도 모르고 26글자에 대해 모두 시도하는 것보다는 훨씬 낫다. 다이그램(digram)-두 글자의 조합-들을 고려하면 암호 해독자가 해독을 하기에 더욱 좋은 상황이 된다: (QJ 같은) 특정 다이그램들은 영문에서는 드물게 나타

나나 다른 (ER같은) 다이어그램들은 아주 자주 나타난다. 글자들과 글 조합들의 발생 빈도들을 조사함으로써, 암호 해독자는 단순한 치환으로 만들어진 암호를 아주 쉽게 해독할 수 있다.

이런 유형의 해독을 위한 공격을 보다 어렵게 만드는 한 가지 방법은 도표를 여러개 사용하는 것이다. 이의 한 간단한 예는 Caesar 암호의 한 확장 형태인 Vigenere 암호이다. 이 방식에서는 몇 개의 글자로 구성된 키가 각 글자에 대한 k 의 값을 정하기 위해 반복적으로 사용된다. 매 스텝마다 키 글자의 인덱스가 원문의 글자의 인덱스에 더해져서 암호문의 글자 인덱스를 만든다. 우리가 샘플로 사용한 원문은 키로서 ABC가 있을 때 다음과 같이 암호화된다:

키: ABCABCABCABCAB
원문: ATTACK AT DAWN
암호문: BVWBENACWAFDXP

예를 들어, 암호문의 마지막 글자는 알파벳의 16번째 글자인 P이다. 왜냐하면 원문의 이에 상응하는 글자는 N (14번째 글자)이고 이때의 키 글자는 B(2번째 글자) 이기 때문이다.

Vigenere 암호는(오프셋(offset)을 사용하지 않고) 원문의 각 글자에 대해 다른 도표들을 사용함으로써 더욱 복잡하게 만들어질 수 있다. 또한 키가 길수록, 만들어진 암호의 해독이 어려워짐이 명백하다. 만약 키가 원문만큼 길면 보통 일회용 패드(one-time pad)라고 불리는 Vernam 암호가 된다. 이것이 유일하게 알려질 안전성을 증명할 수 있는 암호 시스템으로써 Washington과 Moscow간의 직통선(hotline) 및 다른 아주 중요한 분야들에 사용되어지는 것으로 알려져 있다. 이 방식에서는 각 키 글자가 단지 한번만 사용되므로, 암호 해독자가 메시지의 각 글자에 대해 모든 가능한 키를 시도해 보는 것만큼 어려운, 거의 해독을 바랄 수 없는 상황이다. 그러나 각 키 글자를 단지 한번만 사용함은 확실히 심각한 키 분배 문제를 야기시킨다. 따라서 일회용-패드는 아주 이따금씩 전송돼야 하는 상대적으로 짧은 메시지들에 대해서만 유용하다.

만약 메시지와 키가 이진수로 코드화되어 있다면, 각 위치별(position-by-position)로 암호화하기 위한 보다 일반적인 방법은 XOR(exclusive-or) 함수를 이용하는 것이다: 원문의 암호화를 위해, 원문을 키와 비트별로 XOR한 것이다. 이 암호문은 같은 키를 가지고 XOR하면 원문이 나오게 된다. 암호문과 원문을 XOR하면 키가 뒀에 주목하라. 이 결과는 처음에는 의외인 듯이 보인다. 그러나 실제로 많은 암호 작성 시스템들은 암호 해독자가 원문을 갖고 있으면 키를 찾을 수 있는 그러한 속성을 가진다.

암호 작성/해독기들(Encryption/Description Machines)

많은 암호 작성 분야들(예로서 군용 통신을 위한 음성 시스템들)은 많은 양의 데이터 전송을 수반하며 따라서 일회용-패드 방식의 사용이 가능치 않다. 우리가 필요로 하는 것은 분배될 소수의 글자로 구성된 실제 키에서 많은 글자를 갖는 모조-키(pseudo-key)를 만들어 내는 일회용-패드의 유사품 같은 것이다.

이러한 경우들에서의 일반적인 구성은 다음과 같다: 암호 작성기에 전송자가 어떤 암호 변수들(실제키)을 넣으면, 길다란 키 비트-스트림(bit-stream)-즉, 모조키-가 만들어진다. 이 모조키의 비트들과 원문을 XOR하면 암호문이 만들어진다. 암호 작성기와 유사한 장비와 동일한 암호 변수들을 갖고 수신자는 동일한 모조키를 만든 다음, 이를 암호문과 XOR하여 원문을 만들어 낸다.

이런 점에서 키의 생성은 해싱 및 난수생성과 아주 유사하므로, 16장과 35장에서 논의된 방법들은 키 생성을 위해 사용될 수 있다. 실제로 35장에서 논의된 일부 메카니즘들은 처음에는 여기서 언급한 것 같은 암호 작성/해독기들에서 사용하기 위해 개발되었다. 그러나 키 생성 장치들은 난수 발생 장치보다 다소 복잡해져야 한다. 왜냐하면 단순한 기계들을 공격하는 여러 방법들이 있기 때문이다. 문제는 암호 해독자가 원문의 일부(예로서 음성 시스템에서의 침묵 같은)를 얻을 수가 있고 따라서 키의 일부를 쉽게 알 수 있을 수가 있다. 만약 암호 해독자가 키 생성 장치를 잘 안다면, 그 알려진 키의 일부만 갖고도 키(암호 변수들) 전체를 알아낼 수가 있다. 그렇게 되면 키 생성 장치의 동작이 밝혀지고 결국 모든 키들이 찾아질 수가 있게 된다.

암호 작성자들은 이런 문제들을 피할 수 있는 여러 가지 방법들을 알고 있다. 한 가지 방법은 기계 구조 자체의 일부를 한 암호 변수(즉 키의 일부)가 되게 만드는 것이다. 보통 암호 변수들을 제외한 모든 기계 구조를 암호 해독자가 안다고 가정한다. 그러나 암호 변수중의 일부가 기계를 구성하는데 이용되었다면, 이 암호 변수값을 찾기는 어려울 것이다. 암호 해독자를 당혹스럽게 만들기 위해 보통 이용되는 또 한가지 방법은 두 다른 기계가 한 복잡한 키 스트림을 만들기 위해(즉 서로 서로를 작동시키기 위해) 결합되는 결합 암호(product cipher) 방법이다. 또 다른 방법으로 비선형 치환(nonlinear substitution)법이 있다; 이 방법에서는 원문과 암호문간의 치환이 비트 단위가 아니라 큰 블록 단위로 행해진다. 이러한 복잡한 방법들이 갖는 일반적인 문제는 이들이 암호 작성자조차도 이해하기가 어려울 정도로

복잡하고 그래서 또한 어떤 선택된 암호 변수들의 경우에는 암호화가 더 나쁘게 되어질 가능성이 항상 있다는 것이다.

공개키 암호 시스템들(Public-key Cryptosystems)

전자식 자금 이전이나 컴퓨터 우편 같은 상업용 응용 분야들의 경우에는, 키 분배 문제가 기존 암호화의 응용 분야들에서 보다 아주 심각하다. 자주 변경되어야 하는 긴 키들을 모든 사람들에게 보안성과 비용상의 효율성을 모두 유지하면서 제공하는 것은 어렵다. 따라서 이러한 시스템을 개발해서는 안된다. 그러나 최근에 키 분배 문제를 완전히 해결한 방법들이 개발되었다. 공개키(public-key) 암호 시스템들이라 불리는 이러한 시스템들은 머지않아 널리 이용될 것으로 보인다. 이러한 시스템들중 가장 널리 알려진 한 시스템은 우리가 공부해 온 일부 산술 알고리즘들에 그 기반을 두고 있다. 이런 이유로 여기서는 어떻게 이 시스템이 동작하는지를 자세히 살펴보기로 한다.

공개키 암호 시스템의 기본 개념은(전화번호부 같이) 암호키들에 대한 공개 책자를 이용하는 것이다. 모든 사람들의(P로 표기되는) 암호들은 공개된다: 예를 들어, 한 사람의 키가 전화 번호부의 전화번호 바로 옆에 기재되어 있을 수도 있다. 모든 사람들은 또한 암호해독을 위한 해독키를 가진다. 한 메시지 M 을 전송하기 위해, 송신자는 수신자의 공개키를 찾아서, 이 키를 사용하여 M 을 암호화한 후 전송한다. 이 암호화된 메시지 (암호문)을 $C = P(M)$ 으로 나타내기로 한다. 수신자는 개인용 해독키를 갖고 수신된 암호문을 해독한다. 이 시스템이 동작하려면, 적어도 다음 조건들이 만족되어야 한다:

- (i) 모든 메시지 M 에 대해 $S(P(M)) = M$ 이어야 한다.
- (ii) 모든 (S, P) 쌍들이 달라야 한다.
- (iii) P 에서 S 를 유도하는 것이 M 을 판독해 내는 것만큼 어려워야 한다..
- (iv) S 나 P 모두 계산하기가 쉬워야 한다.

여기서 (i)은 근본적인 암호학적 성질이고, (ii)와 (iii)은 보안성을 제공한다. 그리고 (iv)는 이 시스템의 사용을 가능케 한다.

이 일반적인 방법은 W. Diffie와 M. Hellman 에 의해 1976년에 개략적으로 설명이 되어졌다. 그러나 이들은 이러한 모든 조건들을 만족시키는 방법을 찾지 못했다. 얼마 뒤 R.

Rivest, A. Shamir와 L. Adleman에 의해 그러한 방법이 만들어졌는데 이들 이름의 첫자를 따서 이 방법을 RSA 공개키 암호화 시스템이라 한다. RSA는 매우 큰 정수들에 대해 수행되는 산술 알고리즘들에 그 근거를 두고 있다. 암호키 P 는 정수쌍 (N, p) 이고 해독키 S 는 정수쌍 (N, s) 이다. 여기서 S 는 비밀이다.(즉, 공개되지 않는다) 이들 정수들은 매우 커야 한다. (보통 N 은 200자리 숫자이고 P 와 S 는 100자리 숫자이다) 암호화 방법과 그 해독 방법은 아주 간단하다: 먼저 메시지를 N 보다 작은 숫자들로 나눈다.(예로서, 메시지의 한 문자에 해당하는 이진 문자열을 한번에 $\ln N$ 비트들 만큼씩 택함으로써) 그러면 이 숫자들은 각기 먹승(raised to a power) modulo N 이 된다: 한 조각의 메시지 M 을 암호화하려면 $C = P(M) = M^p \bmod N$ 을 수행하고, 암호문 C 를 해독하려면 $M = S(C) = C^s \bmod N$ 을 수행한다. 어떻게 이 계산들을 하는지는 36장에서 배우기로 한다. 200자리 숫자들을 계산하는 것은 성가신 일인 반면, 우리가 단지 N 으로 나눔으로써 생긴 나머지만을 필요로 한다는 사실은, 비록 M^p 과 C^s 이 엄청나게 큰 숫자들임에도 불구하고 그 숫자들을 커지지 않게 할 수 있음을 뜻한다.

성질 23.1 RSA 암호 시스템에서 메시지는 선형 시간 내에 암호화될 수 있다.

긴 메시지들의 경우, 키들로서 사용되는 숫자들의 길이는 상수로 간주될 수 있다. 유사하게, 한 숫자의 먹승은 상수 시간 내에 행해진다. 왜냐하면 숫자들은 한 상수 길이보다 더 길 수가 없기 때문이다. 이 설명은, 긴 숫자들을 가지고 연산하는 것과 관련된, 많은 구현시의 고려 사항들에 대해 언급하지 않고 있다; 이들 연산 동작들에 따른 비용은 실제로는 이 방법이 널리 응용되는 것을 막는 인자가 되고 있다. □

이렇게, 위의 조건 (iv)가 만족되고 조건(ii)는 쉽게 시행될 수 있다. 또한 암호 변수들 N , p 와 s 가 조건 (i)과 (iii)을 만족시킬 수 있도록 선택되어야 한다. 이 조건들에 대한 확신을 갖기 위해서는 이 책의 범위를 넘는 수 이론(number theory)을 설명해야 한다. 여기서는 이 이론의 주개념들의 개요만 설명하기로 한다. 우선, 세계의 큰(약 100자리) 무작위 소수(prime number)를 만들어야 한다: 이중 가장 큰 숫자는 s , 나머지 두 숫자는 x 와 y 로 부르기로 한다. 그리고 나서, x 와 y 의 곱이 되도록 N 을 선택하고 p 는 $ps \bmod (x-1)(y-1) = 1$ 이 되도록 선택된다. 이렇게 선택된 N , p 와 s 를 가지고, 모든 메시지 M 에 대해 $M^{ps} \bmod N = M$ 임을 증명함이 가능하다.

예를 들어, 우리의 표준 코드화 방식에서, 메시지 ATTACK AT DAWN은 A는 알파벳에서 첫번째(01)글자이고, T는 스무 번째(20) 글자 등등이므로 아래와 같은 28자리 숫자에 해당한다:

0120200103110001200004012314.

간단한 설명을 위해(원래 필요한 100자리가 아니라) $x = 47$, $y = 79$ 와 $s = 97$ 의 2자리 소수들을 가지고 예를 들기로 한다. 이 값들은(x 와 y 의 곱인) $N = 3713$, 그리고(97 이 곱해지고 3588 로 나누어지면 나머지가 1 이 되는 유일한 정수로서) $p = 37$ 이 되게 한다. 이제, 메시지를 코드화 하기 위해, 메시지를 4자리 숫자 단위로 나눈 후 p 제곱(modulo N)을 한다. 이렇게 함으로써 아래와 같은 코드화된 버전이 만들어진다:

1404293235360001328422802235.

즉, $1020^{37} \equiv 1404$, $2001^{37} \equiv 2932$, $0311^{37} \equiv 3536 \pmod{3713}$ 등등이다. 해독 과정은 암호화 과정과 같으나 p 대신 s 를 사용한다. 이같이 하면 $1404^{97} \equiv 0120$, $2932^{97} \equiv 2001 \pmod{3713}$ 등등이 되므로, 원래 메시지가 다시 만들어진다.

이 계산에서 가장 중요한 부분은 성질 23.1에서 논의한 것 처럼 메시지의 코드화이다. 그러나 만약 키 변수들을 계산할 수 없다면 어떤 암호 시스템도 있을 수 없다. 비록 이 암호 시스템이 복잡한 수 이론과 큰 숫자들을 조작하는 상대적으로 복잡한 프로그램들을 포함하고 있지만, 키들을 계산하는 시간은 키 길이의 제곱보다 작을 확률이 높다.(키 계산 시간은 키를 나타내는 숫자의 크기에는 비례하지 않는다)

성질 23.2 RSA 암호 시스템을 위한 키들은 과도한 계산 없이 만들어질 수 있다.

또 다시 이 책의 범위를 벗어나는 수 이론에 근거한 방법들이 여기서 필요하다; 각 큰 소수는 먼저 큰 난수(random number)를 만들고 이 수에서부터 한 소수를 발견할 때까지 이 수 다음에 연속하는 수들을 조사함으로써 구해질 수 있음이 밝혀져 있다. 한 간단한 방법은 한 난수에 대해 그 수가 소수가 아님을 증명할 수 있는 $1/2$ 의 확률을 가지고 소수 시험을 한다.(소수가 아닌 수가 검출 안 될 확률은 이 방법을 20번 적용할 경우 백만 분의 일보다 작고 30번 적용할 경우 십억 분의 일보다 작다) 마지막으로 행해야 할 일은 p 를 계산하는 것이다: Euclid 알고리즘의 한 변형 알고리즘으로 이를 계산 할 수 있음이 밝혀져 있다. \square

암호 해독 키 s (그리고 N 의 인자인 x 와 y)는 비밀로 유지돼야 하며 이 방법의 성공은 암호 해독자가 N 과 p 를 가지고 s 값을 찾을 수 없음에 의존한다. 우리가 앞서 언급한 간단한 예제의 경우, $3713 = 47 * 79$ 임을 찾는 것을 쉽다. 그러나 만약 N 이 200자리 숫자라면, 사람들이 이 값의 두 인자를 찾을 가능성은 희박하다. 즉, 어느 누구도 증명은 할 수 없었지만 p (그리고 N)을 알더라도 s 를 계산해 내기는 어려운 듯하다. 명백히, s 로부터 p 를 찾으려면 x 와 y 를 알아야 하며 또한 x 와 y 를 계산하려면 N 을 인수분해함이 필요하다. 그러나 N 을 인수분해하는 것은 아주 어렵다고 생각된다: 알려진 최상의 인수분해 알고리즘들을 갖고도 현재의 기술 능력 하에서는 200자리 숫자를 인수분해 하려면 수백만 년의 시간이 걸릴 수 있다.

RSA 시스템의 한 가지 매력은 N , p 그리고 s 들에 관련한 복잡한 계산들이 시스템의 각 사용자에게 대해 단지 한번만 행해진다는 것이다. 반면에, 아주 빈번히 일어나는 암호화 및 해독 연산들은 단지 메시지를 나누는 동작과 간단한 지수화(exponentiation) 프로시저를 적용하는 동작만을 가진다. 이러한 계산의 단순성은 공개키 암호시스템들이 제공하는 모든 편리한 특징들과 결합되어 시스템을 비밀(보안) 통신을 위한(특히 컴퓨터 시스템과 네트워크들에 대해) 아주 매력있는 시스템으로 만든다.

RSA 방법은 다음과 같은 단점들이 있다: 지수화 프로시저는 암호학적 표준 방법들에 비해 그 비용이 많이 든다. 그리고 더욱 나쁜 것은 이 방법으로 암호화된 메시지들의 판독에 오랜 시간이 걸릴 수가 있다는 것이다. 이것은 많은 암호 시스템들의 경우에도 그렇다: 암호화 방법은 확신을 가지고 이용되기에 앞서 심각한 암호해독을 위한 공격들에도 잘 견딜 수 있어야 한다.

공개키 암호 시스템들의 구현을 위해 여러 가지 다른 방법들이 제시되어져 왔다. 이 중 가장 흥미있는 일부 방법들은 일반적으로 확실치는 않지만 매우 어려운 것으로 여겨지는, 45장에서 논의될, 한 중요한 문제들의 부류와 관련이 있다. 이러한 암호시스템들은 한번의 성공적인 공격이 일부 잘 알려진(RSA 방법에서의 인수분해 같은) 어려운 미해결 문제들을 어떻게 해결해야 하는지에 대한 통찰력을 제공할 수 있는 재미있는 성질을 가지고 있다. 암호학과 컴퓨터 과학 분야의 기본 연구 과제들 간의 이러한 연관성은, 공개키 암호 작성법이 널리 이용될 가능성과 함께, 암호학을 활기 있는 연구 분야로 만들어 주고 있다.

연습문제

1. 다음 메시지는 Vigenere 암호 기법을 가지고(A 앞에 한 개의 블랭크(blank) 문자를 갖는 27글자 알파벳 상의) 필요하다면 반복될 수 있는 패턴 CAB를 키로 이용하여 암호화된 것이다:

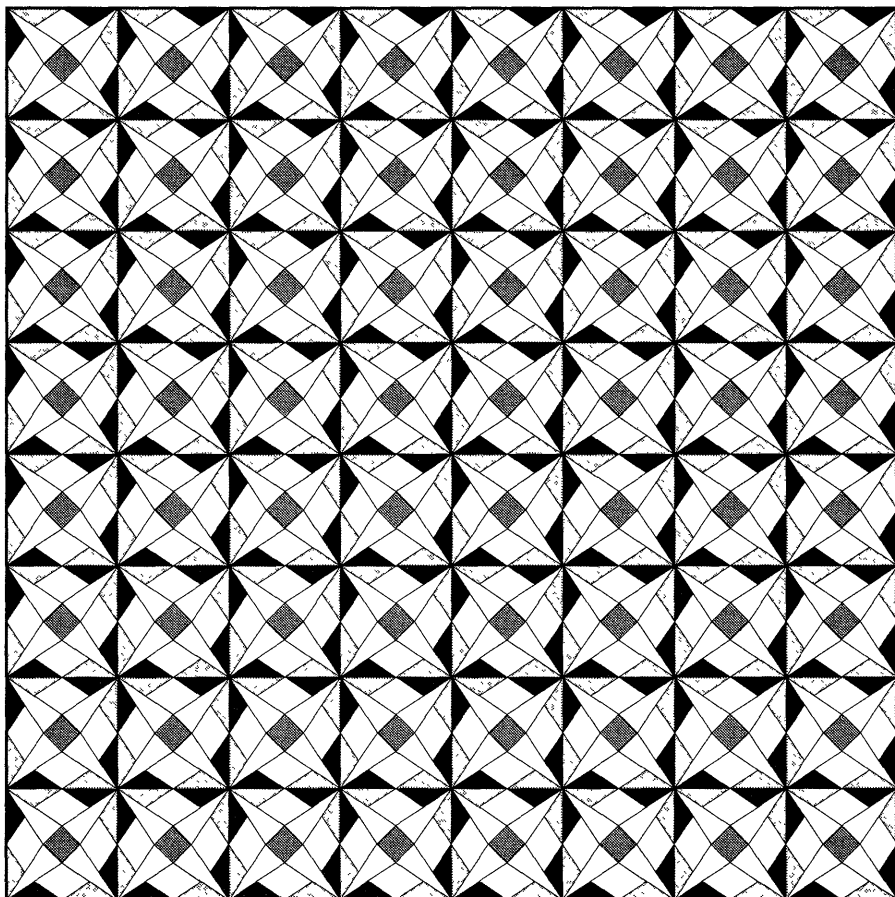
DOBHBUAASXFZWJQQ.

이 메시지를 해독하라.

2. 도표 치환 기법을 이용하여 암호화된 메시지들을 해독하려면 어떤 도표를 사용해야 하는가?
3. 두 문자 키를 갖는 한 Vigenere 암호기법이 상대적으로 긴 한 메시지의 암호화를 위해 사용됐다고 하자. 홀수 자리에 있는 각 문자의 발생 빈도는 대략 짝수 자리에 있는 각 문자의 발생 빈도와 같다는 가정 하에 키를 추리해 내는 프로그램을 작성하라.
4. 한 버전의 이진 메시지와 35장의 선형 조화 난수 발생기(linear congruential random number generator)로 부터 나온 한 이진 스트림 간에 XOR 연산을 사용하는 암호화 프로시저 및 해독 프로시저를 작성하라.
5. 문제 4의 암호화 기법을 깨뜨리는 프로그램을 작성하라. 단 메시지의 첫 10문자는 블랭크 문자라고 가정하라.
6. 한 원문을 키와 비트 단위로 AND 시켜 암호화할 수 있는가? 답에 대해 그 이유를 설명하라.
7. 공개키를 이용하여 암호를 작성하면 한 메시지를 여러 다른 사용자들에게 보내는 것이 편리한가? 예 또는 아니오로 답하고 그 이유를 설명하라.
8. 공개키 암호 작성을 위한 RSA 방법에서 $P(S(M))$ 의 결과는?
9. RSA 코드화는 M^n 의 계산을 수반할 수가 있다. 여기서 M 은 k 정수들의 배열로서 표현되는 k 자리 숫자일 수 있다. 이 계산을 위해 필요한 연산 동작의 수는?

빈 면

기하학적 알고리즘



빈 면

기초적인 기하학적 방법

컴퓨터는 본질적으로 기하학적인 많은 문제들을 해결하기 위해 널리 이용되고 있다. 점, 선 그리고 다각형 같은 기하학적 객체들은 다양한 중요 응용 분야의 기초가 되며 흥미있는 문제들과 알고리즘들을 유발시킨다.

기하학적 알고리즘들은 건물과 자동차에서 VLSI회로(very large-scale integrated circuit)들에 이르는 물리적 객체를 모델링하는 설계 및 분석 시스템들에 있어 중요한 역할을 한다. 물리적 객체를 다루는 설계자는 컴퓨터의 표현 방식으로는 지원하기 어려운 기하학적 감각을 가지고 있다. 많은 다른 응용 분야들은 기하학적 데이터 처리를 직접하고 있다. 예를 들어, 똑같은 인구로 선거구를 나누는(그리고 대부분의 다른 당의 당원들을 한 지역으로 배치 하것 같은 여러 기준들을 만족시키는) 정치적인 선거구 개편은 복잡한 기하학적 알고리즘이다. 이러한 기하학의 또 다른 응용 분야로서 자연스럽게 기하학적으로 표현될 수 있는 수학과 통계학에서의 많은 문제들이 있다.

지금까지 공부한 대부분의 알고리즘들은 기본적으로 대부분의 프로그래밍 환경에서 자연스럽게 표현되고 처리되는 문자와 숫자들을 수반한다. 그리고 실제로 요구되는 기본 연산들은 대부분의 컴퓨터 시스템에서 하드웨어로 구현되었다. 우리는 기하학적 문제들의 경우에는 상황이 이와 다를 것을 알게 될 것이다: 점과 선들에 대한 가장 기본적인 연산 동작들조차도 계산적으로 쉽지 않을 수 있다.

기하학적 문제는 쉽게 가시화 된다. 그러나 이 점이 단점이 될 수 있다. 사람이 한 장의 문서를 보고 즉시 해결할 수 있는 많은 기하학적 문제(예: 한 다각형 안에 점이 있는가?)들도 컴퓨터 프로그램으로 구현하기는 상당히 어렵다. 많은 응용 분야들의 더 복잡한 문제들의

경우에는, 컴퓨터로 구현하기 적합한 해결책과 사람에게 적합한 해결책이 아주 다른 당연하다.

기하학적 알고리즘들은 고대 기하학의 구조적인 성질과 다양하고 유용한 응용 분야들로 인해 오랜 역사가 있는 것으로 생각될 수 있으나, 실제로 이 분야의 많은 업적들은 아주 최근에 이루어 졌다. 하지만 고대 수학자의 업적은 때때로 현대 컴퓨터를 위한 알고리즘들의 개발에 유용하게 쓰인다. 기하학적 알고리즘 분야는 그 역사적 배경과 새로운 중요 알고리즘들의 지속적인 개발과 이러한 알고리즘들을 필요로 하는 중요하고 광범위한 많은 응용 분야들을 볼 때, 관심을 가져볼 만한 분야이다.

점, 선 및 다각형(Points, Lines, and Polygons)

우리가 배울 대부분의 프로그램은, 비록 몇 개는 삼차원 이상의 공간에 대한 것이지만, 이차원 공간에서 정의된 단순한 기하학적 객체들 상에서 동작한다. 기본 객체는 한 쌍의 정수-Cartesian(데카르트) 시스템에서의 점의 좌표-로 표시되는 점이다. 선은 직선 선분에 의해 연결된 한 쌍의 점이다. 다각형은 점들의 리스트이다: 연속되는 점들은 선분들에 의해 연결되고 첫 점이 끝점에 연결되어 폐도형(closed figure)을 형성한다고 가정한다.

이같은 기하학적 객체들을 다루기 위해, 이들의 표현 방법을 정하는 것이 필요하다. 여기서는 다각형의 표현을 위해 연결-리스트나 다른 표현 방식을(적합하다면) 사용할 수도 있지만, 배열을 사용하기로 한다. 본문에 있는 대부분의 프로그램은 다음과 같은 단순한 표현을 사용한다:

```
struct point {int x, y; char c;};  
struct line {struct point p1, p2;};  
struct point polygon[Nmax];
```

점들이 정수 좌표들로 표시 되도록 제약됨을 주목하라. 물론 부동 소수점 표현이 사용될 수도 있지만 정수 좌표를 사용하는 것이 좀더 간단하고 효과적인 알고리즘들을 가져다준다. 이 정수 좌표의 사용은 생각보다는 심각한 제약이 아니다. 2장에서 언급한 것처럼, 가능하면 정수를 사용하여 작업을 함으로써 많은 경우 상당한 시간 절약이 가능하다. 왜냐하면 전형적으로 정수 계산은 부동 소수점 계산보다 훨씬 더 효과적이기 때문이다. 그러므로, 특별한 계산 없이 정수 계산만으로 원하는 것을 얻어낼 수 있다면, 정수 좌표를 사용할 것이다.

보다 복잡한 기하학적 객체들도 이러한 기본적인 요소들로서 표현될 수 있다. 예를 들어, 다각형들은 점들의 배열들로서 표현될 수 있다. 선의 배열을 다각형의 표현에 사용하면(비록 이 표현이 어떤 알고리즘에서는 자연스러운 표현일지라도) 다각형의 각 점이 두 번씩 포함되는 결과를 가져올 수 있다는 점에 주목하라. 또한 어떤 응용 분야에서는 각 점과 선에 관련하여 잉여 정보를 추가함이 유용하다; 이 추가는 레코드들에 정보 필드를 추가함으로 가능해진다.

몇 개의 기하학적 알고리즘들의 동작을 설명하기 위해 그림 24.1에 있는 것과 같은 점들의 집합들을 사용하기로 한다. 왼편의 16개의 점은 예제들의 설명시의 참조를 위해 각각 한 글자로 명명되어 있고 그림 24.2에 있는 것과 같은 정수 좌표들을 갖는다.(점들의 이름은 그 점들의 가정된 입력 순서에 따라 부여된다) 프로그램들은 점들을 그 이름을 가지고 참조하지는 않는다; 점들은 단순히 배열에 저장되고 인덱스에 의해 참조된다. 어떤 프로그램들에서는 배열 상에 나타나는 점들의 순서가 매우 중요할 수 있다: 정말로, 일부 기하학적 알고리즘들은 한 특정 순서로 점들을 “정렬”하는 것을 목적으로 한다. 그림 24.1의 오른편 그림에는 0과 1000사이의 정수 좌표값에서 무작위로 생성된 128개의 점이 있다.

한 전형적인 프로그램은 점의 배열 p 를 유지하고, N 쌍의 정수를 읽어 들이며, 첫 번째 쌍을 $p[1]$ 의 x 와 y 좌표에 할당하고, 두 번째 쌍을 $p[2]$ 에 할당하는 식으로 동작한다. p 가 한 다각형을 나타낼 때는, 때때로 “표지값(sentinel value)” $p[0]=p[N]$ 과 $p[N+1]=p[1]$ 을 유지하는 것이 편리하다.

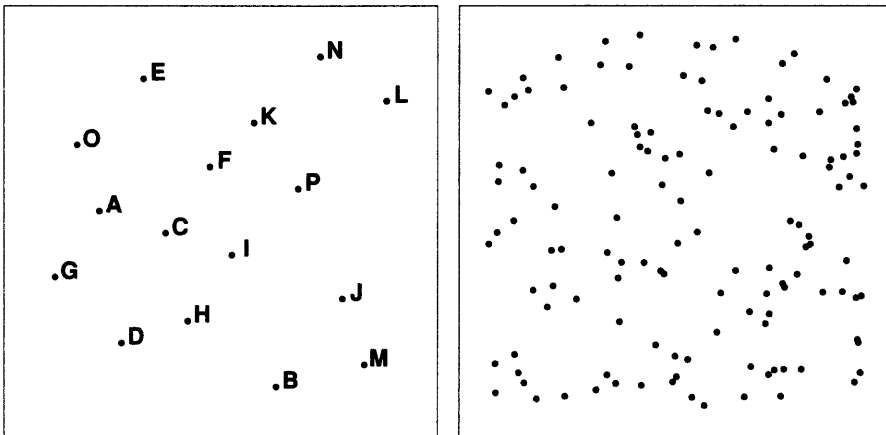


그림 24.1 기하학적 알고리즘들을 위한 샘플 점 집합들

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
x	3	11	6	4	5	8	1	7	9	14	10	16	15	13	3	12
y	9	1	8	3	15	11	6	4	7	5	13	14	2	16	12	10

그림 24.2 작은 샘플 집합 내의 점들의 좌표값들(그림 24.1 참조)

선분 교차(Line Segment Intersection)

첫 번째 기본적인 기하학적 문제로서, 주어진 두 선분이 서로 교차하는지 아닌지를 결정하는 문제를 다루기로 한다. 그림 24.3은 일어날 수 있는 일부 상황들을 보여 준다. 첫 번째 경우에는, 선분들이 교차한다. 두 번째 경우에는, 한 선분의 끝점이 다른 선분 상에 접하고 있다: 우리는 선분들이 “닫혀 있다”고 가정하여,(끝점은 선분들의 일부분이다) 선분들이 교차한다고 간주한다; 이같이 선분들은 공통 교차 끝점을 갖는다. 그림 24.3의 마지막 두 가지 경우에는, 선분들이 교차하지 않는다. 그러나 선분에 의해 정의되는 직선의 교차점을 고려하면 다르다. 네 번째 경우에는 교차점이 한 선분상에 있게 되지만 세 번째 경우에는 그렇지 않다. 또한 직선들은 평행을 이룰 수도 있다.(종종 발생하는 특별한 경우는 두 선분들 중의 하나 또는 모두가 하나의 점일 경우이다)

이 문제를 해결하는 한 간단한 방법은 선분들에 의해 정의된 선들의 교차점을 찾아서 이 교차점이 선분의 양끝 점의 사이에 있게 되는지를 조사하는 것이다. 또 한 가지 쉬운 방법은 우리가 나중에 유용함을 알게 될 한 도구에 기반을 둔 것인데, 이에 대해 좀 더 자세히 살펴보기로 한다. 주어진 세 점을 가지고, 첫 번째 점에서 두 번째 점을 거쳐 세 번째 점으로 갈 때, 시계방향으로 가야 하는지 반시계 방향으로 가야 하는지를 알기를 원한다. 예를 들어, 그림 24.1의 점 A, B, C의 경우에는, 그 답이 ‘예’이지만, 점 A, B, D에 대한 답은 ‘아니오’이다. 이 함수는 아래와 같은 선들의 방정식들로부터 쉽게 계산된다:

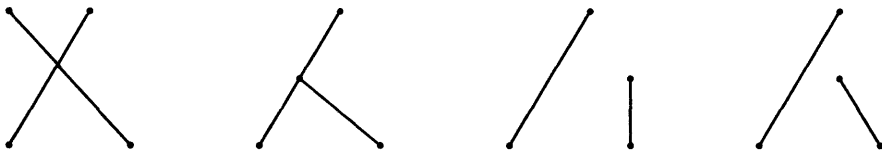


그림 24.3 선분들의 교차 여부의 시험: 4가지 경우.

```

int ccw(struct point p0,
        struct point p1,
        struct point p2)
{
    int dx1, dx2, dy1, dy2;
    dx1 = p1.x - p0.x; dy1 = p1.y - p0.y;
    dx2 = p2.x - p0.x; dy2 = p2.y - p0.y;
    if (dx1*dy2 > dy1*dx2) return +1;
    if (dx1*dy2 < dy1*dx2) return -1;
    if ((dx1*dx2 < 0) || (dy1*dy2 < 0)) return -1;
    if ((dx1*dx1+dy1*dy1) < (dx2*dx2+dy2*dy2))
        return +1;

    return 0;
}

```

프로그램이 어떻게 동작하는지 이해하기 위해, 먼저 정수 $dx1$, $dx2$, $dy1$, $dy2$ 모두가 양수라고 하자. 그러면, $p0$ 와 $p1$ 을 연결하는 직선의 기울기는 $dy1/dx1$ 이 되고 $p0$ 와 $p2$ 를 연결하는 직선의 기울기는 $dy2/dx2$ 가 됨에 주목하라. 만약 두 번째 직선의 기울기가 첫 번째 직선의 기울기보다 크다면, $p0$ 에서 $p1$, $p2$ 로 이동하는데 반시계 방향(왼쪽)으로 이동이 필요하다; 만약 작다면, 시계 방향(오른쪽)으로 이동이 필요하다. 프로그램에서 기울기를 비교하는 것은 약간 불편하다. 왜냐하면 직선들이 수직($dx1$ 이나 $dx2$ 는 0이 될 수도 있다)이 될 수 있기 때문이다: 이를 피하기 위해 $dx1*dy2$ 를 곱한다. 이 테스트가 올바르게 되기 위해 기울기가 양수일 필요는 없음이 알려져 있다-이의 확인 문제는 연습 문제로 남겨 둔다.

그러나, 위의 설명에서 빠진 중요한 한 가지는 기울기들이 모두 같은 경우(세 점이 동일 선 상에 있다)들이 무시되었다는 것이다. 이러한 경우들에 있어서, 사람들은 ccw 함수를 정의하는 다양한 방법을 생각 할 수 있다. 우리가 선택한 방법은 이 함수의 리턴 값을 세 가지로 만드는 것이다: 즉, 0이나 0이 아닌 값을 리턴 하는 대신, 1과 -1, 그리고 $p2$ 가 $p0$ 과 $p1$ 사이의 선분 상에 접했을 경우에는 0을 리턴 하는 것이다. 만약 점들이 동일 선 상에 있고, $p0$ 가 $p2$ 와 $p1$ 사이에 있다면 ccw 의 리턴 값은 -1이 되고, $p2$ 가 $p0$ 와 $p1$ 사이에 있으면, ccw 는 0이 된다; 그리고, $p1$ 이 $p0$ 와 $p2$ 사이에 있으면 ccw 는 1이 된다. 이러한 규칙은 이 장과 다음 장에 있는 ccw 를 사용하는 함수들의 코딩을 간단하게 해준다.

이 설명을 가지고 즉시 $intersect$ 함수를 구현할 수 있다. 만약 각 선의 양 끝점이 서로 다른 “쪽”에 있다면(즉 다른 ccw 값을 가지면), 그 선들은 반드시 교차한다.

```

int intersect (struct line l1, struct line l2)
{
    return ((ccw (l1.p1, l1.p2, l2.p1)
               *ccw (l1.p1, l1.p2, l2.p2) ) <= 0)
           && ((ccw (l2.p1, l2.p2, l1.p1)
               *ccw (l2.p1, l2.p2, l1.p2) ) <= 0);
}

```

이 해결책은 이런 간단한 문제에 대해 상당량의 계산을 수행하는 듯하다. 독자는 좀더 간단한 해결 방법을 찾기 위해 노력해야 한다. 단 그 해결 방법이 모든 경우에 확실히 동작해야만 함을 명심하라. 예를 들어, 4개의 점이 모두 동선 상에 있다면(점들이 일치하는 경우를 빼고도) 6가지의 다른 경우들이 있다. 이 중 4가지의 경우가 교차하는 경우이다. 이와 같은 특별한 경우들이 기하학적 알고리즘들의 취약점이다: 이를 피할 수는 없지만, ccw 같은 기본 함수들에 대한 영향을 줄일 수는 있다.

만약 많은 직선들이 있다면 상황은 더 복잡해진다. 27장에서는, N 개의 직선의 집합에서 어느 두 직선이 교차하는지의 여부를 알려주는 매우 복잡한 알고리즘을 보게 될 것이다.

단순 폐경로(Simple Closed Path)

점들의 집합들을 다루는 문제들을 잘 이해하기 위해서, N 개의 점들로 구성된 집합의 모든 점들을 지나며 시작점으로 다시 돌아오는, 그러나 자신을 교차하지 않는, 경로를 생각해 보자. 이러한 경로를 단순 폐경로(simple closed path)라고 부른다. 이를 실제로 응용한 예를 생각해 보면, 점들은 집들을 나타낼 수 있고 경로는 우편 배달부가 각각의 집에 이르는, 자신의 경로를 가로지르지 않고 가는, 길로 표현될 수 있다. 또는 우리는 플로터 장치를 사용해서 점들을 그리기 위한 한 합리적인 방법을 원할 수도 있다. 단지 점들을 연결하는 폐경로만을 찾는 이 문제는 기본적인 문제이다. 외판원 문제(traveling salesman problem)라 불리는 최적의 이같은 경로를 찾는 문제는 훨씬 더 어렵다. 따라서 이 문제는 이 책의 마지막 몇 장들에서 좀더 자세히 살펴보기로 한다. 다음 장에서는, 이와 관련된 좀 더 쉬운 문제인 주어진 N 개의 점들의 집합을 에워싸는 최단 경로를 찾는 문제를 다루기로 한다. 31장에서는, 한 점들의 집합을 연결하기 위한 최적의 방법을 어떻게 찾는지에 대해 배울 것이다.

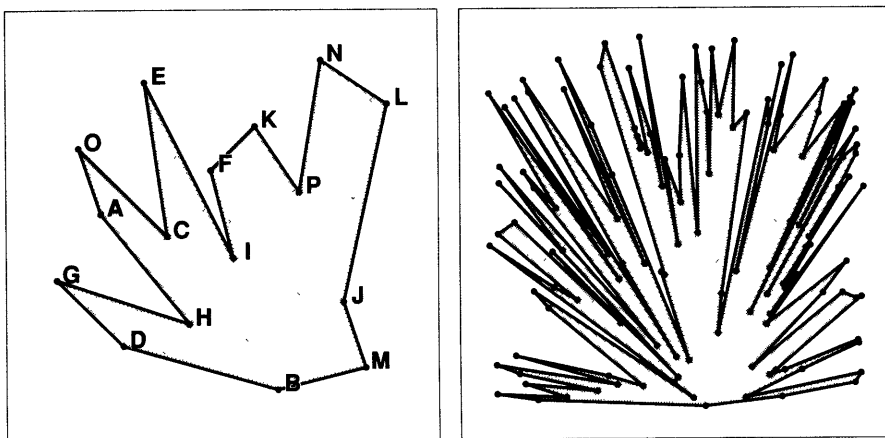


그림 24.4 단순 폐경로

기초적인 문제를 해결하기 위한 한 간단한 방법은 다음과 같다. “닻(anchor)”의 역할을 할 한 점을 점들 중에서 선택한다. 그리고 나서, 각 점에서 닻까지 가는 선을 긋고 다시 양의 수평 방향으로 진행할 때 만들어지는 각도를 계산한다.(이것은 닻으로 지정된 점을 기점으로 하는 각 점의 극좌표의 일부이다) 다음에는, 각도에 따라 점들을 정렬한다. 마지막으로, 인접한 점들을 연결한다. 결과로서, 그림 24.4에서 보여지는 것처럼, 그림 24.1의 점들을 연결한 단순 폐경로가 만들어진다. 왼쪽 그림에서는 B가 닻으로 이용된다: 만약 점들이

B M J L N P K F I E C O A H G D B

의 순으로 방문된다면, 한 단순 폐다각형이 그려질 것이다.

만약 dx 와 dy 가 닻으로 지정된 점에서 어느 다른 점까지 가는, x 축과 y 축 상에서의 거리들이라면, 이 알고리즘에 필요한 각은 $\tan^{-1}dy/dx$ 이다. 비록 \tan^{-1} 는 C++에서(그리고 일부 다른 프로그래밍 환경에서 내장함수이나, 속도가 느리고 다시 계산을 해야하는 적어도 두 개의 귀찮은 조건(dx 가 0인지 그리고 어떤 사분면에 점이 있는지)을 갖게 된다. 이 알고리즘 내에서 각도는 단지 정렬을 위해서만 사용되기 때문에, 계산하기가 훨씬 더 쉬운, 그러나 \tan^{-1} 와 동일한 “순서” 성질들을 갖는, 함수를 사용하는 것이 좋다.(그래서 정렬되어질 때, 동일한 결과를 얻어진다) 이러한 함수에 대한 한 좋은 후보는 $dy/(dy + dx)$ 이다. 예외 조건의 시험이 필요하긴 하지만 간단하다. 다음 프로그램은 0에서 360사이의 숫자를 리턴한다. 물론 이 숫

자는 p1과 p2, 그리고 수평선에 의해 만들어진 각도는 아니지만 그 각도와 동일한 “순서”성 질들을 가진다.

```
float theta(struct point p1, struct point p2)
{
    int dx, dy, ax, ay;
    float t;
    dx = p2.x - p1.x; ax = abs(dx);
    dy = p2.y - p1.y; ay = abs(dy);
    t = (ax + ay == 0) ? 0 : (float) dy/(ax+ay);
    if (dx < 0) t = 2-t; else if (dy < 0) t = 4+t;
    return t*90.0;
}
```

어떤 프로그래밍 환경에서는 표준 삼각 함수 대신 이와 같은 프로그램을 사용하는 것이 가치가 없을지도 모른다. 그러나 다른 환경에서는 상당한 절약을 가져올 수 있다.(어떤 경우에는 theta가 정수 값을 갖도록 변경하여 부동소수점의 사용을 완전히 없애는 것이 더 좋을 수도 있다)

다각형 안에 포함됨(Inclusion in a Polygon)

다음 번 문제는 한 점과 점들의 배열로서 표현된 한 다각형을 가지고 그 점이 다각형 안에 있는지 또는 밖에 있는지를 결정하는 문제이다. 이 문제를 해결하는 방법은 다음과 같다: 주어진 점으로부터 임의의 방향으로 긴 직선 선분을 긋고(이 점에서 그은 선분이 다각형 밖으로 나가도록 길게) 이 선분이 가로질러서 다각형과 만나는 수를 센다. 만약 홀수라면 주어진 점은 분명히 다각형 안에 위치하고, 만약 짝수라면 다각형 밖에 위치한다. 이 방법은 다각형 밖에 있는 끝점에서 다각형 안으로 들어오면서 발생하는 사건들을 추적해보면 쉽게 알 수 있다: 첫 번째 선분 다음에는 다각형 안에 있게 되고, 두 번째 선분 다음에는 다각형 밖에 있게 되고…. 이렇게 짝수 번을 행하면, 결국에는 다각형 밖에 있게 된다.

그러나 상황이 그렇게 간단하지만은 않다. 왜냐하면 주어진 다각형을 형성하는 바로 그 정점들 상에서 교차가 발생할 수 있기 때문이다. 그림 24.5는 처리되어야 할 몇 가지 상황들을 보여준다. 첫 번째 경우는 단순히 외부에 있는 경우이고, 두 번째 경우는 단순히 내부에 있는

경우이다. 세 번째 경우는 시험을 위해 그은 직선이(다각형의 두 정점을 거쳐) 다각형을 빠져나가는 경우이며, 네 번째 경우는 시험을 위해 그은 직선이 다각형을 벗어나기 전에 다각형의 한 선분과 일치하는 경우이다. 시험을 위해 그은 직선이 다각형의 한 정점과 교차하는 경우들에서는 다각형과 한 번 교차가 일어난 것으로서 계산되어야 하며, 또 다른 경우들에서는 교점이 없거나 두 번 교차한 것으로 계산되어야 할 것이다. 독자들은 더 진도를 나가기에 앞서, 이같은 경우들을 구별하는 간단한 시험 방법을 찾기 위해 노력해 보자.

다각형의 정점이 시험용 직선들과 마주치는 경우들을 다루려면 시험용 직선이 다각형과 교차하는 수만 세어서는 안된다. 필수적으로, 다각형 주변을 살펴서 시험용 직선의 한쪽에서 다른 쪽으로 갈 때는 언제나 교차 수를 하나씩 증가시킨다. 이를 프로그램으로 구현하는 한 방법은 시험용 직선과 마주치는 점들을 아래 프로그램에서처럼 무시해버리는 것이다:

```
int inside (struct point t, struct point p[], int N) .
{
    int i, count = 0, j = 0;
    struct line lt, lp;
    p[0] = p[N]; p[N+1] = p[1];
    lt.p1 = t; lt.p2 = t; lt.p2.x = INT_MAX;
    for (i = 1; i <= N; i++)
    {
        lp.p1 = p[i]; lp.p2 = p[i];
        if (!intersect(lp,lt))
        {
            lp.p2 = p[j]; j = i;
            if (intersect(lp,lt)) count++;
        }
    }
    return count & 1;
}
```

이 프로그램은 계산을 쉽게 하기 위해 시험용으로 수평 직선(그림 24.5를 45도 회전시킨 것으로 생각하면 된다)을 사용한다. 변수 j 는 시험용 직선 위에 있지 않다고 알려진 다각형상의 마지막 점에 대한 인덱스이다. 이 프로그램은 가장 작은 y 좌표를 가지는 모든 점들 중에서 가장 작은 x 좌표를 가지는 점을 $p[1]$ 이라고 가정한다. 그래서 만약 $p[1]$ 이 시험용 직

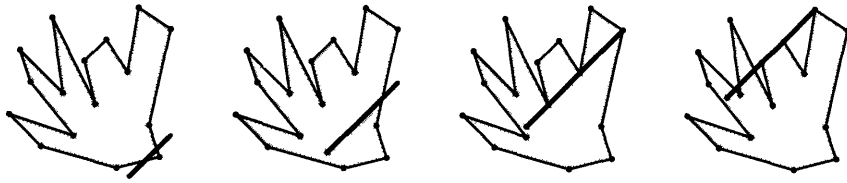


그림 24.5 다각형 내의 점을 찾는 알고리즘에 의해 다뤄지는 경우들

선 위에 있다면, $p[0]$ 는 직선 상에 있을 수 없다. 이 다각형은 N 개의 다른 p 배열들로서 표현될 수 있다. 그리고 때로는 $p[1]$ 에 대해 한 표준 규칙을 정함이 편리하다.(예를 들면, 앞에서 제시된, 한 단순 폐다각형을 계산하기 위한, 프로시저의 경우 $[p1]$ 을 닛으로 사용하면 좋다) 만약 시험용 직선 상에 있지 않은 다각형 상의 다음 점(j 번째 점으로서) 시험용 직선과 같은 쪽에 있으면, 교차 횟수를 증가시킬 필요가 없다; 만약 아니면 교차 횟수를 증가시켜야 한다. 독자들은 이 알고리즘이 그림 24.5에 있는 경우들에 대해 올바르게 동작하는지를 확인해 보기 바란다.

만약 다각형이 많은 응용프로그램의 경우처럼 단지 3개 또는 4개의 면을 가지고 있다면, 이처럼 복잡한 프로그램은 필요하지 않다: ccw 함수 호출에 기반을 둔 한 단순한 프로시저 만으로도 충분할 수 있다. 다음 장에서 논의할 또 하나의 특수한 경우는 볼록 다각형(convex polygon)으로서 어떤 시험용 직선도 다각형과 두 번 이상 교차할 수 없는 성질을 가지고 있다. 이 경우 이진 검색 같은 프로시저가, 한 점이 안에 있는지 밖에 있는지의 여부를 $O(\log N)$ 스텝 내에 결정하기 위해, 사용될 수 있다.

전망 (Perspective)

앞에 주어진 몇 가지 예제를 들을 보고서, 컴퓨터를 사용하여 특정 기하학적 문제를 해결하는데 있어서의 어려움을 과소평가하기 쉽다. 그러나 아직도 우리가 전혀 다루지 않은 많은 기초적인 기하학적 계산들이 많이 있다. 예를 들면, 한 다각형의 면적을 계산하는 프로그램은 아주 흥미 있는 과제이다. 또한 우리가 다뤘은 문제들은 다음 장들에서 보다 어려운 문제들을 풀 때 유용하게 쓰일 약간의 기본적인 도구들을 제공한다. 그러나 고려될 문제들과 알고리즘들의 범위가 너무 크므로 근본적인 문제들을 해결 해주고 이제까지 살펴본 알고리즘과 연관되는 몇 가지 선택된 예제들만 다루기로 한다.

우리가 연구할 일부 알고리즘들은 주어진 점들의 집합으로부터 기하학적 구조를 형성하는 것을 포함한다. “단순 폐다각형”은 이에 대한 가장 기초적인 예이다. 우리는 이와 같은 구조들에 대한 적절한 표현방식을 결정할 필요가 있으며, 그 구조들을 형성하는 알고리즘을 개발해야 하며, 그리고 특정 응용 분야들에서 이들을 이용하는 방법을 연구해야 한다. 보통 이런 연구 사항들은 서로 연관되어 있다. 예를 들어, 이 장에서 ‘inside’ 프로시저에 사용된 알고리즘은 근본적으로 단순 폐다각형이(무순서로 있는 직선들의 집합으로 보다는) 순서대로 나열된 점들의 집합으로 표현됨에 의존한다.

보통 C++의 데이터 추상화 기능은 응용 분야들에 대한 표현 방식들 중에서 편리한 방법을 선택할 수 있도록 해준다. 게다가, 기하학적 응용 분야들은 전형적으로 C++에서 제공하는 계층적 클래스 구조를 이용하여 모든 다른 유형의 객체들과 이 객체들에 대해 구현 돼야 하는 연산 동작들을 올바르게 형성한다. 정말로 C++ 문들은 이런 방식의 이점들을 보여주기 위해 기하학적 객체들을 종종 사용한다. 우리는 더 자세하게 이런 문제점들을 다루지는 않겠다. 왜냐하면 알고리즘적인 관점에서 보면, 이에 따르는 연산들은 너무 기초적이거나(예를 들어 한 형상을 그리거나 회전시킨다) 너무 어려운(예를 들어 두 다각형의 교점을 계산한다) 경향을 보이기 때문이다. 또한 점들, 선들 그리고 다각형들의 동적인 집합에서의 검색, 교차, 그리고 다른 연산 동작들을 지원하는 패키지는 이 책의 범위를 벗어난다. 그러나 일부 가장 중요한 지원 연산 동작들이 어떻게 효율적으로 구현될 수 있는지를 고려해 보고자 한다.

공부할 많은 알고리즘들은 기하학적 검색을 수반한다: 우리는 주어진 집합 내의 어떤 점들이 한 주어진 점과 가까운지, 어떤 점이 주어진 사각형 내에 있는지, 또는 어떤 점들이 서로 가장 가까이 있는지 등을 알고자 한다. 이러한 검색 문제들에 적합한 많은 알고리즘들은 14장에서 17장까지에서 공부한 검색 알고리즘들과 밀접한 관련이 있다. 이들이같은 종류들임은 아주 명백하다.

상대적 특성이 정확히 언급될 수 있을 정도로 분석된 기하학적 알고리즘은 거의 없다. 우리가 이미 본 것처럼, 한 기하학적 알고리즘의 수행시간은 많은 것들에 따라 좌우된다. 점들의 분포, 점들이 입력 상에 나타나는 순서, 그리고 삼각함수의 사용여부 등은 그 기하학적 알고리즘의 수행시간에 모두 상당한 영향을 끼칠 수 있다. 그러나 보통 이런 상황에서, 특정 응용 분야들의 경우 좋은 알고리즘들을 제시해 주는 경험적 증거가 있다. 또한 많은 알고리즘들은 복잡도 연구(complexity study)를 통해 유도되며 최악의 경우를 고려하여 설계된다.

연습문제

1. 두 점이 같으며 나머지 한 점은 다를 때의 세 가지 경우들에 대한 ccw 값과 세 점이 동일할 경우의 ccw 값을 구하라.
2. 분할함이 없이 두 선분이 평행한지 아니지를 결정하는 빠른 알고리즘을 구하라.
3. 분할함이 없이 네 선분이 평행한지 아니지를 결정하는 빠른 알고리즘을 구하라.
4. 주어진 선들의 배열이 단순 폐다각형을 구성하는지의 여부를 보려면 어떻게 시험해야 하나?
5. 그림 24.1의 A, C 와 D를 본문에서 설명한 닳으로 이용하였을 때 만들어지는 단순 폐다각형들을 그려라.
6. 단순 폐다각형의 계산을 위해 임의의 한 점을 본문에서 설명한 “닳”으로서 이용한다고 하자. 이 임의로 선택된 점을 갖고 동작하기 위해 만족 되어야 하는 조건들은?
7. 동일한 선분의 복사본 두 개를 가지고 intersect 함수가 호출됐을 때의 리턴값은?
8. inside함수는 다각형의 한 정점을 내부에서 호출하는가 외부에서 호출하는가?
9. N개의 정점을 갖는 다각형 상에서 inside 함수가 실행되었을 때, count가 가질 수 있는 최대값은? 답이 맞음을 예를 들어 설명하라.
10. 주어진 한 점이 주어진 한 사변형의 내부에 있는지 아니지를 결정하는 효과적인 프로그램을 작성하라.

25 장

볼록 외곽 찾기

처리해야 할 많은 점들이 있을 때, 종종 이들 점들의 집합의 경계에 대해 관심을 갖게 된다. 평면상에 찍혀진 점들의 집합에 대한 다이어그램에서 가장 자리에 있는 점들과 점 집합 내부에 있는 점들을 구분하는데는 별로 어려움이 없다. 이러한 구분이 점 집합들의 한 근본적인 성질이다; 이장에서는 자연적인 경계를 이루는 점들을 구분해 주는 알고리즘들을 살펴봄으로서 어떻게 이 구분이 특징지어질 수 있는지를 보기로 한다.

점 집합의 자연적인 경계를 설명하는 수학적 방법은 볼록면체(convexity)라 불리는 한 기하학적 성질에 의존한다. 이 성질은 독자가 전에 본적이 있는 단순한 개념이다: 볼록 다각형은 그 다각형내의 어느 두 점을 연결하는 선도 그 다각형 내에 위치해야만 하는 성질을 가진다. 예로서 앞 장에서 계산한 단순 폐다각형은 분명히 볼록 다각형이 아니다; 한 편 모든 삼각형이나 사각형은 볼록 다각형이다.

한 점 집합의 자연적인 경계를 수학적으로는 볼록 외곽(convex hull)이라 한다. 평면상의 한 점 집합의 볼록 외곽은 점들을 모두 포함하는 최소의 볼록 다각형으로써 정의된다. 또한 이와 동등하게, 볼록 외곽은 점들을 에워싸는 최단 경로이다. 볼록 외곽의 증명하기 쉬운 한 명확한 성질은 외곽을 정의하는 볼록 다각형의 정점들은 점 집합에 있는 점들이다. N 개의 주어진 점들이 있을 때, 이 중 일부는 다른 모든 점들을 포함하는 볼록 다각형을 형성한다. 문제는 이러한 점들을 찾는 것이다. 볼록 외곽을 찾기 위해 많은 알고리즘들이 개발되어졌다; 이 장에서는 이 중 몇 개의 중요한 알고리즘들을 살펴보기로 한다.

그림 25.1은 그림 24.1의 점 집합들과 이에 대한 볼록 외곽들을 보여준다. 점의 갯수가 작은 집합에서는 외곽 상에 8개의 점이 있고 점의 갯수가 큰 집합에서는 외곽 상에 15개의 점

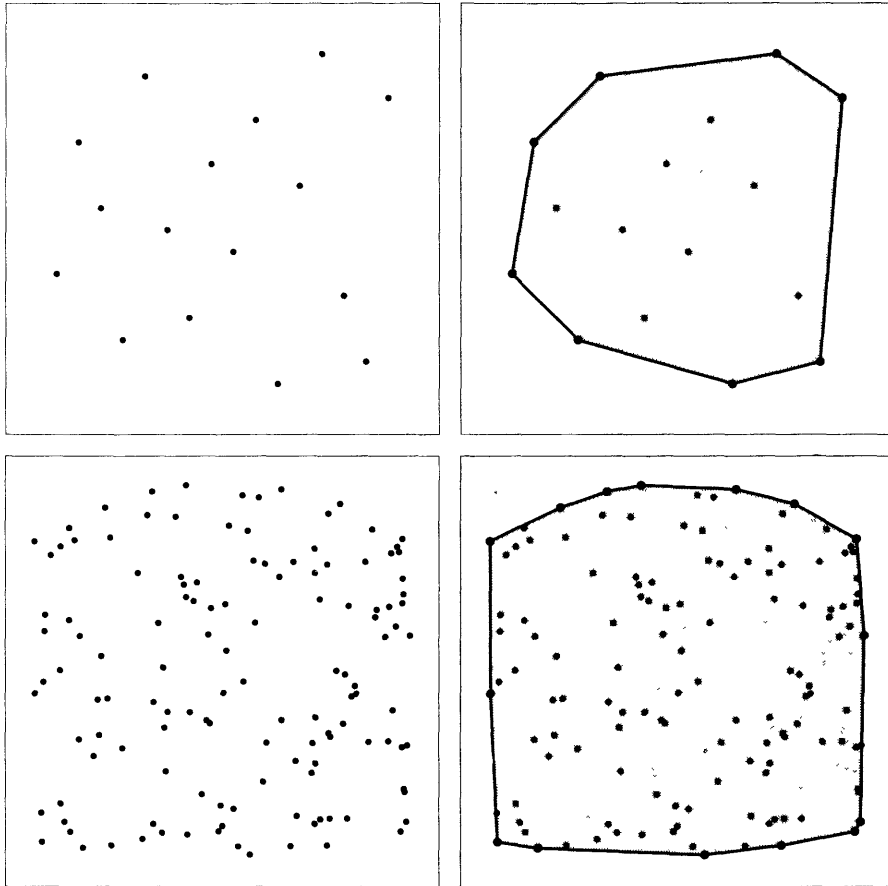


그림 25.1 그림 241의 점 집합들에 대한 볼록외각들

이 있다. 일반적으로, 볼록외곽은 점 세 개로도 구성될 수 있고(만약 이 세 점으로 생성된 외곽이 다른 모든 점들을 포함한다면) 모든 점으로 구성될 수도 있다.(만약 모든 점들이 볼록 외곽을 형성한다면) 무작위적 점들의 집합의 볼록 외곽 상에 있는 점들의 수는 이 두 극단의 경우의 사이에 있게 된다. 일부 알고리즘들은 볼록 외곽 상에 많은 점들이 있을 때 잘 동작하나 나머지 알고리즘들은 외곽 상에 점들이 많지 않을 때에 더 잘 동작한다.

볼록 외곽의 한 근본적인 성질은 외곽 밖의 한 선이 외곽 쪽으로 이동될 때는, 반드시 외곽의 정점들 중의 하나와 만난다는 것이다.(이것은 외곽을 정의하는 또 다른 방법이다: 외곽은 무한대로부터 어느 일정 각도로서 안으로 이동해 오는 한 선과 만날 수 있는 점 집합의 한 부분집합이다) 특히 이 규칙을 수평선과 수직선에 적용함으로써 외곽 상에 있는 소수의

점을 찾는 것은 쉽다: 최소 x, y 좌표와 최대 x, y 좌표를 갖는 점들은 모두 볼록 외곽 상에 있다. 이 사실은 우리가 살펴보고자 하는 알고리즘들의 시작점으로서 이용된다.

게임 규칙(Rules of the Game)

볼록 외곽을 찾는 알고리즘에 대한 입력은 물론 점들의 배열이다; 이를 위해 앞에서 정의한 point 형태(type)를 이용할 수 있다. 출력은 한 다각형인데, 이 다각형은 배열에 있는 순서대로 점들을 추적하면 다각형의 외곽을 추적한 것이 되는 성질을 갖는 점들의 배열로서 나타내어진다. 잘 생각해 보면, 이것은 볼록 외곽 계산에 대해 또 하나의 순서 조건(ordering condition)을 필요로 하는 듯이 보인다.(왜 외곽상의 점들을 임의의 순서로 리턴하지 않는가?) 그러나 순서에 따라 출력함이 확실히 더 좋다. 무순서적으로 계산하는 것이 더 쉬운 것은 아님이 알려져 있다. 우리가 살펴보고자 하는 모든 알고리즘의 경우 계산을 적절한 장소에서 하는 것이 편리하다: 원래의 점 집합을 위해 사용된 배열은 출력의 저장을 위해 또한 사용된다. 알고리즘들은 원래 배열에 있던 점들을 단순히 재배열하여 볼록 외곽을 구성하는 점들이 배열 내의 첫 M 개의 엔트리에 순서대로 있게 한다.

위 설명에서 볼 때, 볼록 외곽의 계산은 정렬과 아주 밀접한 관계가 있다. 실제로, 볼록 외곽 알고리즘은 다음과 같은 방법으로 정렬을 위해 사용될 수 있다. N 개의 정렬될 숫자들을 고정된 반지름을 갖는(적절히 정규화된) 각들로서 간주하여(극 좌표 상에서의) 점들로 변환한다. 이 점 집합의 볼록 외곽은 모든 점들을 포함하는 한 개의 N -각형이다. 출력 순서는 이 다각형상에 점들이 나타나는 순서에 따라야 하므로, 볼록 외곽은 원래 값들로부터 정렬된 순서를 찾는데 이용될 수 있다.(입력은 순서 없이 되었음을 기억하라) 이 설명은 볼록외곽의 계산이, 예를 들자면 숫자들을 다각형상의 점들로 변화하는데 필요한 삼각함수들의 비용 또한 고려해야 하기 때문에, 정렬보다 결코 쉽지 않다는 것에 대한 공식적인 증명은 아니다.(삼각함수 연산을 수반하는) 볼록 외곽 알고리즘들과(키들간의 비교를 수반하는) 정렬 알고리즘들과의 비교는 어떻게 보면 사과와 귤을 비교하는 것과 같다. 비록 그렇다 하더라도 어떤 볼록 외곽 알고리즘도 정렬의 경우와 마찬가지로 약 $N \log N$ 연산동작을 필요로 한다.(비록 연산동작들이 아주 다르겠지만) 한 점들의 집합에 대한 볼록 외곽을 찾는 것을 일종의 이차원 정렬로서 간주함이 도움이 될 때가 많다. 왜냐하면 볼록 외곽을 찾는 알고리즘들의 연구 시 정렬 알고리즘들의 경우와 유사한 경우가 빈번히 발생하기 때문이다.

실제로 우리가 공부할 알고리즘들은 블록 외곽을 찾는 것 또한 정렬보다 결코 어렵지 않음을 보여준다: 최악의 경우 $N \log N$ 에 비례한 시간 내에 수행되는 여러 알고리즘들이 있다. 많은 알고리즘들은 실제 점 집합들 상에서 더 짧은 시간이 걸리는 경향이 있다. 왜냐하면 이들의 수행시간은 어떻게 점들이 분포되어 있는가와 외곽 상에 있는 점들의 수에 좌우되기 때문이다.

모든 기하학적 알고리즘들의 경우와 마찬가지로, 입력에서 나타날 수 있는 퇴행적 경우들에 주목해야 한다. 예를 들어, 동일 선분 상에 모든 점들이 있는 집합의 블록 외곽은 무엇인가? 응용 분야에 따라서, 블록 외곽은 모든 점들일 수도 있고 단지 양극에 있는 두 점일 수도 있고, 또는 두 양극 점을 포함하는 어느 점 집합일 수도 있다. 비록 이 예는 극단적인 듯 하나 세 점 이상의 점들이 한 점 집합의 외곽을 정의하는 선분들 중의 하나에 있는 것은 생소한 경우가 아니다. 아래에서 배울 알고리즘들에서는 블록 외곽 선분상의 점들이 반드시 포함돼야 한다고 주장하지 않을 것이다. 왜냐하면 그럴 경우에는 일반적으로 더 많은 일들이 수반되기 때문이다.(그러나 필요하다면 언제 포함되어질 수 있는지는 알려줄 것이다) 우리는 또한 외곽 선분상의 점들이 생략 되어 한다고도 주장하지 않을 것이다. 왜냐하면 이 경우는 필요하다면 나중에도 조사될 수 있기 때문이다.

패키지 포장(Package-Wrapping)

인간이 한 점 집합의 블록 외곽을 그리는 방법에 필적하는 가장 자연스런 블록 외곽 알고리즘은 점 집합을 포장하는 한 체계적인 방법이다. 블록 외곽 상에 있는 한 점에서 시작하여, (최소 y 좌표 값을 갖는 점이라고 하자) 수평선 상의 양의 방향으로 광선을 비추고 다른 한 점이 찾아질 때까지 상향 이동한다; 이렇게 하여 찾아진 점은 반드시 외곽 상에 있어야 한다. 이 찾아진 점에서 다시 시작하여 또 다른 점이 찾아질 때까지 광선을 상향이동 한다. 이런 식으로 패키지가 완전히 포장될 때까지(시작점이 다시 찾아질 때 까지) 계속한다. 그림 25.2는 우리의 샘플점 집합에 대해 이런 식으로 외곽이 찾아지는 과정을 보여준다. 점 B는 최소 y 좌표 값을 가지며 시작점이 된다. M은 광선을 상향 이동할 때 찾아지는 첫 번째 점이고 그 다음에 찾아지는 점은 L 등이다.

실제로는 모든 각도를 다 조사할 필요는 없다; 우리는 다음번 점을 찾기 위해 한 표준 최소값-찾기(find-the-minimum)계산을 수행한다. 외곽 상에 포함될 각 점을 찾기 위해 외곽 상에 아직 포함 안된 각 점을 조사할 필요가 있다. 이처럼, 이 방법은 선택 정렬(selection

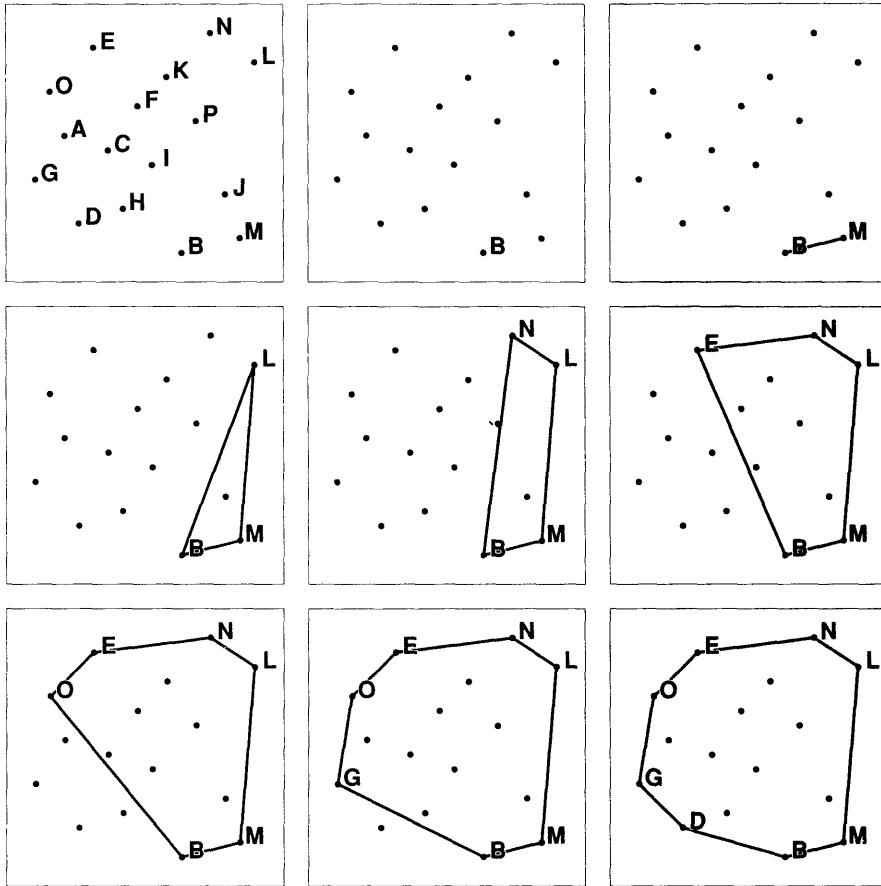


그림 25.2 패키지 포장

sorting)과 아주 유사하다. 우리는 최소값을 찾기 위해 한 주먹구구식 방법을 사용하여 아직 선택되지 않은 점들 중에서 최적의 점을 계속하여 선택한다. 이에 관련된 실제 데이터 이동을 그림 25.3에서 볼 수 있다. 도표에서 M번째 줄은 M번째 점이 외곽에 더해진 후의 상황을 보여준다.

아래 프로그램은 24장의 시작부에서 설명한 것처럼 표현된 N 개의 점들에 대한 배열 p 의 볼록 외곽을 찾는다. 이 프로그램 구현의 기본은 전 장에서 개발된 함수 θ 이다. 이 함수는 두 점 p_1 과 p_2 를 인수로 받아들여서 p_1 과 수평선 그리고 p_2 와 수평선 사이의 각을 리턴하는(실제로는 동일 순서 성질들을 갖고 더욱 쉽게 계산된 수를 리턴한다) 함수로 생각할 수 있다. 이 외의 모든 구현은 위에서 논의된 것을 따른다. 최소값-찾기 계산을 위해서는

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
B	A	C	D	E	F	G	H	I	J	K	L	M	N	O	P
B	M	C	D	E	F	G	H	I	J	K	L	A	N	O	P
B	M	L	D	E	F	G	H	I	J	K	C	A	N	O	P
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P
B	M	L	N	E	O	G	H	I	J	K	C	A	D	F	P
B	M	L	N	E	O	G	H	I	J	K	C	A	D	F	P
B	M	L	N	E	O	G	D	I	J	K	C	A	H	F	P

그림 25.3 패키지-포장시의 데이터 이동

한 표지값(sentinel)이 필요하다: 보통은 $p[0]$ 가 이용될 수 있도록 배열하나 이 경우에는 $p[N+1]$ 이 이용되도록 배열을 하는 것이 더 편리하다.

```

int wrap(point p[], int N)
{
    int i, min, M;
    float th, v;
    for (min = 0, i = 1; i < N; i++)
        if (p[i].y < p[min].y) min = i;
    p[N] = p[min]; th = 0.0;
    for (M = 0; M < N; M++)
    {
        swap(p, M, min);
        min = N; v = th; th = 360.0;
        for (i = M+1; i <= N; i++)
            if (theta(p[M], p[i]) > v)
                if (theta(p[M], p[i]) < th)
                    { min = i; th = theta(p[M], p[min]); }
        if (min == N) return M;
    }
}

```

우선, 최소 y 좌표 값을 갖는 점을 찾아서 아래에 설명된 것처럼 루프를 정지시키기 위해 $p[N+1]$ 로 복사한다. 변수 M 은 이제까지 외곽 상에 포함된 점들의 수를 가지며 v 는 현재의 탐색(sweep)각(수평선을 기준으로 한 $p[N+1]$ 과 $p[M]$ 사이의 각)이다. for 루프는 찾아진 마지막 점을 M 번째 점과 바꿈으로써 외곽 상에 넣고 이 전장에서 본 θ 함수를 사용하여 수평선을 기준으로 한 이 점과 외곽 상에 아직 포함 안된 각 점들 사이의 각을 계산하고, v 보다 큰 각들 중에서 최소 각을 찾는다. 루프는 첫 번째 점(즉 $p[N+1]$ 로 복사된 점)을 다시 만날 때 정지한다.

이 프로그램은 볼록 외곽 선분 상에 있는 점들을 리턴할 수도 있고 안 할 수도 있다. 이러한 상황은 알고리즘의 실행동안 두 점 이상이 $p[M]$ 과 동일한 θ 값을 가질 때 일어난다; 위의 구현 프로그램은 이러한 점들 중에서, 비록 $p[M]$ 에 보다 인접한 다른 점들이 있더라도, 처음 만나는 점을 리턴한다. 볼록 외곽 선분 상에 있는 점들을 찾는 것이 중요한 경우에는 θ 함수를 인수로서 주어진 점들 간의 거리를 고려하고 이 두 점들이같은 각을 가질 때 더 가까이 인접한 점에게 더 작은 수를 할당하도록, 바꿈으로써 가능해 진다.

패키지-포장 알고리즘의 주된 단점은 최악의 경우-모든 점이 볼록 외곽 상에 있을 경우-그 수행 시간(선택 정렬의 경우처럼) N^2 에 비례한다는 점이다. 한편 이 방법은 삼차원 이상으로 일반화될 수 있는 매력이 있다. k -차원 공간에서의 점 집합의 볼록 외곽은 점들을 모두 포함하는 최소의 다중체(poly tope)이다. 여기서 볼록 다중체는 내부의 두 점을 연결하는 선은 반드시 내부에 있어야 하는 성질로서 정의된다. 예로서 삼차원에서 점들의 집합의 볼록 외곽은 각 면이 평평한 볼록 삼차원 객체이다. 이 볼록 외곽은 외곽이 찾아질 때까지 한 평면을 탐색(sweeping)하고, 그 평면의 면들을 싸고(folding), 그 외곽의 경계상의 또 다른 선들에서 이 작업을 다시 시작(anchoring)하고 하여 패키지가 완전히 포장 될 때까지 계속한다. (많은 기하학적 알고리즘들처럼 이 일반화된 알고리즘의 구현하는 것 보다는 설명만 하는 것이 더 쉽다!)

Graham 스캔(The Graham Scan)

우리가 검토할 다음 방법은 1972년 R.L. Graham에 의해 고안된 것으로서 대부분의 계산이 정렬을 위한 것이라는 점이 흥미롭다: 이 알고리즘은 정렬을 위한 계산과 정렬 뒤에 따르는 사소한 계산들을 수행한다. 이 알고리즘은 전 장에 있는 방법을 사용하여 점들을 가지고

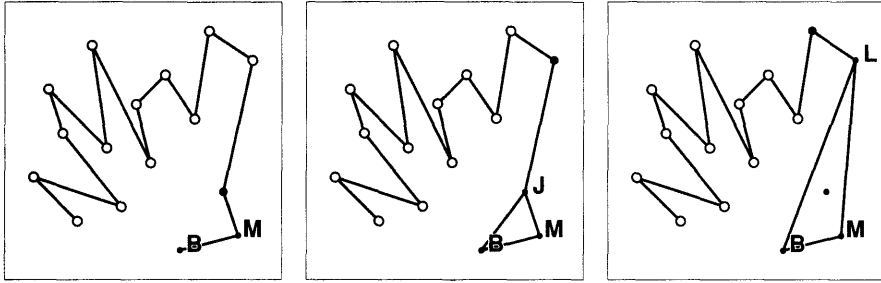


그림 25.4 Graham 스캔의 시작

한 단순 폐다각형을 형성함으로써 시작한다: $p[1]$ 을 닳점(anchor point)으로 하고 각 점을 연결하는 선분에 의해 만들어진 수평선을 기준으로 한 각(angle)에 해당하는 θ 함수의 값들을 키로서 사용하여 점들을 정렬한 다음, $p[1], p[2], \dots, p[N], p[1]$ 까지를 추적(trace)하면 한 폐다각형이 만들어진다. 우리의 예제 점 집합에서는, 전 장에서 본 단순 폐다각형이 얻어진다. $p[N], p[1]$ 그리고 $p[2]$ 는 외곽상의 연속적인 점들임에 주목하라; 정렬을 함으로써, 우리는 필연적으로 패키지-포장 프로시저를(양방향으로) 한번 수행하게 된다.

볼록외곽 계산은 외곽 상에 있을 수 있는 각 점을 찾고 외곽 상에 위치할 수 없는 이전에 찾아진 점을 제거하는 것이다. 예로서, B M J L N P K F I E C O A H G D의 순서로 있는 점들을 생각해 보자; 첫 일부 스텝들이 그림 25.4에 있다. 처음에 우리는 정렬에 의해 B와 M이 외곽에 있음을 안다. J를 만나게 되면 첫 세 점에 대한 잠정적 외곽(trial hull)상에 J를 포함시킨다. L을 만나게 되면 J가 외곽 상에 있을 수 없음을 알게 된다.(왜냐하면 J는 삼각형 BML 내에 위치하기 때문이다)

일반적으로, 어떤 점들을 제거해야 할 지를 조사하는 것은 어렵지 않다. 각 점이 추가된 후, 우리는 현재까지 추적해 온 점들을 근거로 하여 볼 때 볼록 외곽 상의 점들일 수도 있는 이제까지 추적한 많은 점들을 제거했다고 가정한다. 우리는 추적을 하는 동안 각 외곽 정점에서 좌회전을 기대한다. 만약 새로운 한 점이 우회전하게 되다면 바로 전에 추가된 점은 제거되어야만 한다. 왜냐하면 이(제거될) 점을 포함하는 볼록 다각형이 존재하기 때문이다. 특히 한 점의 제거여부는 전 장에 있는 ccw프로시저를 이용하여 다음과 같이 조사된다. $p[1], \dots, p[m]$ 이 $p[1], \dots, p[i-1]$ 의 검사를 근거로 하여 구해진 일부 외곽점들이라고 하자. 새로운 한 점 $p[i]$ 를 검사할 때 만약 $ccw(p[m], p[m-1], p[i])$ 가 음수가 아니면 $p[m]$ 을 제거한다. 만약 음수이면 $p[m]$ 은 제거하지 않고 그냥 외곽 상에 놔둔다.

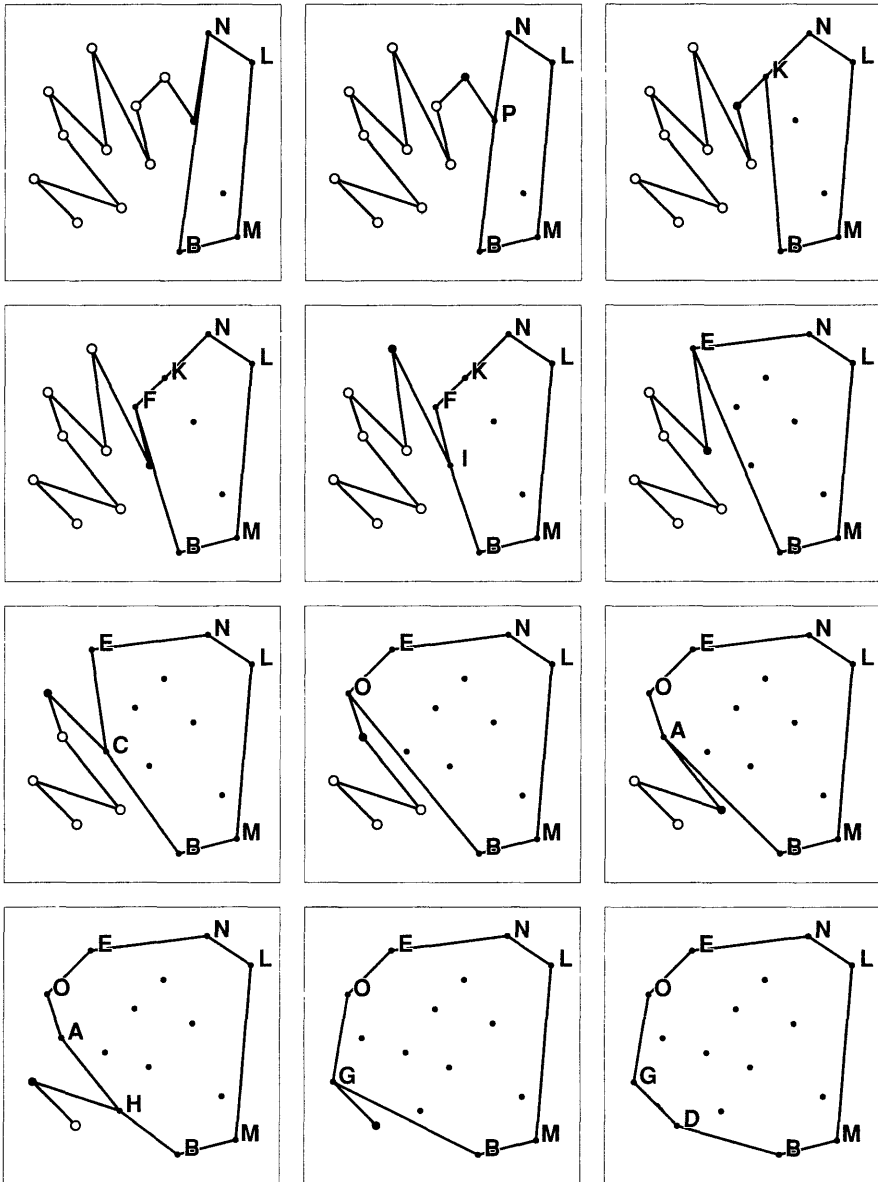


그림 25.5 Graham 스캔의 완료과정

그림 25.5는 우리의 샘플 점 집합에 대한 이 알고리즘의 완료 과정을 보여주고 있다. 새로운 점들이 마주칠 때마다의 상황이 다이어그램으로 나타나 있다: 이 각각의 새로운 점들은 이제까지 형성된 외곽 상에 추가되며 이전에 고려됐던(0개 이상의) 점들을 제거하기 위한 증거로서 이용된다. L, N 그리고 P가 외곽 상에 추가된 후, K가 고려될 때, P는 제거된다.(왜냐하면 NPK가 우회전이므로) 그리고 나서 F와 I가 추가되고 KFE와 NKE가 우회전이므로 F와 K가 제거된다. 이 처럼 두 점 이상이 이 과정에서 제거될 수 있다. 이런 식으로 계속하여, 알고리즘은 다시 B로 돌아오게 된다.

초기의 정렬은 각 점이 차례로 외곽 가능점(possible hull point)으로서 고려되도록 보장해 준다. 왜냐하면 먼저 고려될 모든 점들이 나중에 고려될 점들보다 작은 θ 값을 갖기 때문이다. 이 제거과정에서 살아 남은 각 선은 이제까지 고려된 모든 점들이 그 선과 같은 쪽에 있는 성질을 가진다. 그래서, 정렬에 의해 당연히 외곽 상에 있는 $p[N]$ 으로 다시 돌아올 때, 모든 점들에 대한 완전한 블록 외곽을 갖게 된다.

패키지-포장법의 경우처럼, 외곽 선분 상의 점들은(비록 동일 직선상의 점들에서 일어날 수 있는 두 가지 다른 경우가 있지만) 포함될 수도 안될 수도 있다. 첫째로, 만약 $p[1]$ 과 동일 직선 상에 두 점이 있다면, 위에서처럼 θ 를 이용한 정렬을 통해서 이들을 동일 선상에서의 순서대로 얻을 수 있을 수도 있고 얻을수 없을 수도 있다. 이 경우, 순서대로 있지 않은 점들은 스캔시에 제거된다. 둘째로, 잠정적 외곽(trial hull)을 따라 동일 직선 상에 있는(그리고 제거되지 않는) 점들이 있을 수 있다.

일단 기본적 방법이 이해되면, 그 구현은 -비록 주의를 기울여야할 많은 세부 사항들이 있지만- 간단하다. 먼저, 최소 y 값을 가진 모든 점 중에서 최대 x 값을 가진 점이 $p[1]$ 과 교환된다. 그리고 나서, 점들을 재배열하기 위해 두 점의 θ 값들과 $p[1]$ 을 비교하는 비교 연산동작을 갖는 클래스로 구현된 point를 가지고 shellsort를 한다.(어떤 비교에 근거한 정렬 루틴도 사용 가능하다) 정렬 후에는, $p[3]$ 가 외곽상에 있지 않을 경우의 표지값(sentinel)으로서 동작하도록 $p[N]$ 이 $p[0]$ 로 복사된다. 마지막으로, 위에서 설명한 스캔이 수행된다. 다음은 점 집합 $p[1], \dots, p[N]$ 의 블록 외곽을 찾는 프로그램이다:

```
int grahamscan(point p[], int N)
{
    int i, min, M;
    for(min = 1, i = 2; i <= N; i++)
```

```

    if (p[i].y < p[min].y) min = i;
for(i = 1; i <= N ; i++)
    if (p[i].y == p[min].y)
        if (p[i].x > p[min].x) min = i;
swap(p, 1, min);
shellsort(p, N);
p[0] = p[N];
for (M = 3, i = 4; i <= N ; i++)
{
    while (ccw(p[M], p[M-1], p[i]) >= 0) M--;
    M++; swap(p, i, M);
}
return M;
}

```

루프는, 위에 설명된 것처럼, 부분 외곽 $p[1], \dots, p[m]$ 을 유지한다. 새로운 i 값이 고려 될 때마다, 이 부분 외곽에서 점들을 제거하기 위해 필요하다면 M 이 감소되고 $p[i]$ 가 임시

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	M	J	L	N	P	K	F	I	E	C	O	A	H	G	D
B	M	L	J												
B	M	L	N	J											
B	M	L	N	P	J										
B	M	L	N	K	J	P									
B	M	L	N	K	F	P	J								
B	M	L	N	K	F	I	J	P							
B	M	L	N	E	F	I	J	P	K						
B	M	L	N	E	C	I	J	P	K	F					
B	M	L	N	E	O	I	J	P	K	F	C				
B	M	L	N	E	O	A	J	P	K	F	C	I			
B	M	L	N	E	O	A	H	P	K	F	C	I	J		
B	M	L	N	E	O	G	H	P	K	F	C	I	J	A	
B	M	L	N	E	O	G	D	P	K	F	C	I	J	A	H

그림 25.6 Graham 스캔시의 데이터 이동

로 부분외곽에 추가되도록 하기 위해 $p[i]$ 를 $p[M+1]$ 과 바꾼다. 그림 25.6은 이 예제에서 매번 새로운 한 점이 고려될 경우의 배열 p 의 내용을 보여준다.

독자는 24장에서 설명한 표준형인 최소 y 좌표값을 갖는 모든 점들 중에서 최소 x 좌표값을 갖는 점을 찾기 위해 왜 \min 계산이 필요한지는 알고 싶을 수도 있다. 위에서 논의된 것처럼, 또 하나의 미묘한 사항은 동일선 상의 점들이 동일한 θ 값들을 갖게되나 선상에 나타나는 순서대로 정렬되지 않을 수도 있다는 사실에 따른 영향을 고려해야 한다는 점이다.

이 방법이 연구할만한 관심거리가 되는 한 가지 이유는, 이 방법이 역추적(backtracking)기법(44장에서 다루어지는, 어떤 것을 시도하고 잘되지 않으면 다른 것을 시도하는 알고리즘 설계 기법)의 한 단순한 형태이기 때문이다.

내부제거(Interior Elimination)

거의 모든 볼록 외곽 방법들은 대부분의 점들을 빨리 찾아주는 한 간단한 기법에 의해 상당히 성능이 개선될 수 있다. 일반적인 개념은 단순하다: 외곽 상에 있는 것으로 알려진 네 개의 점을 선택하고 이 네 점에 의해 만들어진 사변형 내의 모든 점을 제거하라. 이렇게 하면 Graham 스캔법이나 패키지-포장법에 의해 고려되어야 할 점들이 상당히 줄어든다.

외곽 상에 있는 것으로 알려진 네 점들은 입력되는 점들에 대해 알려진 정보를 통해 육안으로 선택한다. 일반적으로, 점들을 입력 점들의 분포에 따라 선택함이 최선이다. 예를 들어, 만약 특정 범위내의 모든 x 와 y 값들이같은 확률을 가지면,(즉 사각형 분포이면) 구석에서부터 안쪽으로 스캔해 감으로써 선택된 네 점들은(두 좌표값의 합과 차가 최소이고 최대인 네 점들을 찾는다) 거의 모든 점들을 제거시켜 주는 것으로 알려져 있다. 그림 25.7은 이 기법이 우리의 두 예제 점 집합들에 적용됐을 때, 외곽 상에 있지 않은 대부분의 점들을 제거해 주는 것을 보여준다.

내부제거법의 구현시, 무작위적 점들의 집합들에 대한 내부 루프는 한 주어진 점이 사변형 내에 있는지의 여부에 대한 검사를 위한 루프이다. 이 검사는 x 와 y 축에 평행한 선분을 갖는 사각형을 이용하면 다소 빨리 수행될 수 있다. 위에서 설명한 사변형에 적합한 최대 사각형은 그 사변형을 정의하는 네 점들의 좌표에서 쉽게 찾아진다. 이 사각형을 사용하면 내부에 있는 점들이 좀 덜 제거될 것이다. 그러나 이 방법을 사용하여 얻어지는 빠른 실행속도는 이 손실을 감당하고도 남음이 있다.

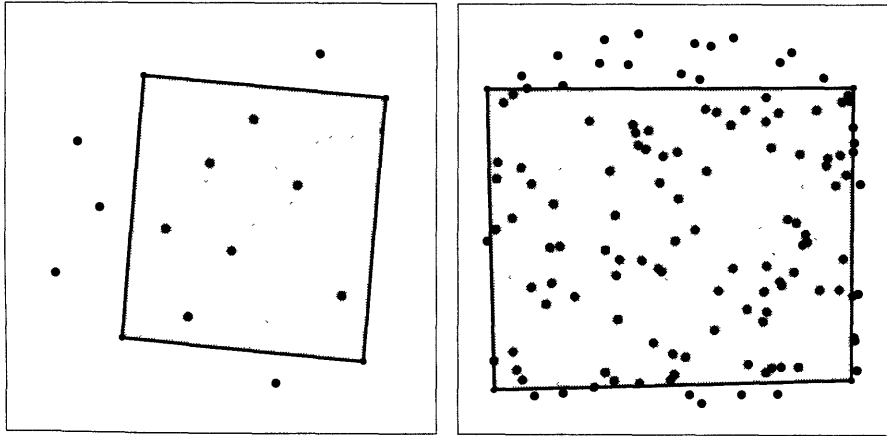


그림 25.7 내부 제거

성능 문제(performance Issues)

전장에서 언급한 것처럼, 기하학적 알고리즘들은 우리가 공부해온 일부 분야들의 알고리즘들보다 입력(그리고 출력)을 특정 짓기가 더 어렵기 때문에 분석하기가 다소 어렵다. 무작위적 점 집합들이라고 말하는 것이 이해가 가지 않을 경우가 종종 있다: 예를 들어, N 이 커짐에 따라 사각형 분포에 따라 그려진 점들의 볼록 외곽은 점들의 분포를 정의하는 사각형과 아주 가까워 보인다. 우리가 살펴본 알고리즘들은 점 집합 분포의 서로 다른 속성들에 의존하므로 실제로는 비교할 수가 없다. 왜냐하면 그들을 분석적으로 비교하려면 점 집합들의 거의 이해되지 않은 성질들 간의 매우 복잡한 상호 작용의 이해가 필요하기 때문이다. 이와는 달리, 우리는 특정 분야에 대한 알고리즘의 선택을 돕는 알고리즘들의 성능에 대한 사항들을 언급할 수가 있다.

성질 25.1 정렬 후의 Graham 스캔은 선형시간 내에 처리된다.

이 성질이 사실인지를 확실히 하려면 잠시 생각할 필요가 있다. 왜냐하면 프로그램에 중첩 루프(loop-within-a-loop)가 있기 때문이다. 그러나 어떤 점도 한번 이상 제거되지 않음을 쉽게 알 수 있다. 따라서 중첩 루프 내의 프로그램은 N 번 보다 적게 반복 수행된다. 이 방법을 이용하여 볼록 외곽을 찾는데 걸리는 총 시간은 $O(N \log N)$ 이다. 그러나 이 방법의 “내부 루프”는 8장에서 12장에 있는 기법들을 이용하여 효율적으로 될 수 있는 정렬(sorting)이다. □

성질 25.2 만약 외곽 상에 M 개의 정점이 있으면, 패키지-포장 기법은 약 MN 스텝을 필요로 한다.

우선 최소값을 찾기 위해 $N-1$ 개의 각들을 계산해야 한다. 그리고, 그 다음으로 작은 값을 찾기 위해 $N-2$ 개의 각들을 계산하고, 그 다음에는 $N-3$ 개 등등을 계산한다. 그래서 각도 계산의 총 횟수는 $(N-1) + (N-2) + \dots + (N-M+1)$ 이 된다. 이 값은 정확히 $MN - M(M-1)/2$ 와 같다. 이 방법을 Graham 스캔 방법과 분석적으로 비교하려면 N 의 관점에서의 M 에 대한 한 공식이 필요로하는데 이는 통계적 기하학(stochastic geometry)에서의 한 어려운 문제이다. 원형 분포(그리고 일부 다른 분포들)의 경우에는 M 이 $O(N^{1/3})$ 이 되는 것이 답이고, N 의 값이 크지 않은 경우 $N^{1/3}$ 은 사각형 분포의 경우의 기대값인 $\log N$ 에 비교될 수 있다.

그러므로 이 방법은 Graham 스캔 방법과 상당히 견주어 볼 만하다. 물론 최악의 경우는 N^2 임을 항상 염두에 두어야 한다. \square

성질 25.3 내부제거법은 평균적으로 선형이다.

이 방법을 완전히 수학적으로 분석하려면 위의 경우보다 더 복잡한 통계적 기하학이 필요하다. 그러나 일반적인 결과는 직감적으로 얻어진 결과와 같다: 거의 모든 점들이 사변형 내에 있게 되어 제거되고 남아 있는 점들의 수는 $O(\sqrt{N})$ 이다. 이것은 앞에서 설명한 것과 같이 사각형이 이용되어도 마찬가지이다. 이에 의해 볼록 외곽 알고리즘의 평균 수행 시간이 N 에 비례하게 된다. 왜냐하면 대부분의 점들이 단지 한번(제거될 때) 검사되기 때문이다. 평균적 시간을 고려할 때는 나중에 어떤 방법이 이용되는지는 별로 문제가 안된다. 왜냐하면 남아 있는 점들이 아주 극소수이기 때문이다. 그러나 최악의 경우(모든 점이 볼록 외곽 상에 있을 때)를 막기 위해, Graham 스캔 방법을 이용하는 것이 신중한 선택이다. 이렇게 하면 실제로 알고리즘이 거의 선형시간 내에 수행되어지며 $N \log N$ 에 비례하는 시간내에 수행되는 것이 보장된다. \square

성질 25.3의 평균적인 경우의 결과는 사각형 내에서 무작위적으로 분포한 점들에 대해서만 성립한다. 최악의 경우에는 이 내부 제거법에 의해 어떤 점도 제거되지 않는다. 그러나 다른 분포들이나 미지의 성질들을 갖는 점 집합들의 경우에도 이 방법은 권장할 만 하다. 왜냐하면 그 비용이 적고(몇 개의 간단한 시험들과 점들에 대한 선형 스캔이 있다) 절약될 가능성이 높기 때문이다.(대부분의 점들은 쉽게 제거될 수 있다) 또한 이 방법은 쉽게 고차원으로 확장이 된다.

내부제거법을 재귀적으로 구현할 수가 있다: 극점들을 찾고 위에서 정의된 것과 같은 사변형의 내부에 있는 점들을 제거한 후 나머지 점들을 동일한 방법을 이용해서 독립적으로 해결될 수 있도록 분할한다. 이 재귀적 기법은 12장에서 논의한 선택을 위한 빠른 정렬(quick sort)같은 select 프로시저와 유사하다. 이 프로시저처럼, 이 재귀적 방법도 N^2 최악 수행시간이 걸리는 단점이 있다. 예로서, 만약 원래의 점들 모두가 볼록 외곽상에 있다면, 어떤 점도 이 재귀적 스텝동안 제거되어지지 않을 것이다. select처럼, 평균적인 수행시간은(비록 증명하기는 쉽지 않지만) 선형이다. 그러나 아주 많은 점들이 첫 번째 스텝에서 제거되므로 실제 응용분야에서는 재귀적 분할 방식을 채택하여 수고할 필요는 거의 없다.

연습문제

1. 한 점 집합에 대한 블록 외곽이 삼각형임을 미리 알았다고 하자. 이 삼각형을 찾기 위한 한 쉬운 알고리즘을 작성하라. 블록 외곽이 사변형일 경우에 대한 알고리즘 또한 작성하라.
2. 한 점이 주어진 한 블록 다각형 내에 있는지를 확인해 주는 효율적인 방법을 제시하라.
3. 문제 2의 방법을 이용하여 삽입 정렬(insertion sort)같은 블록 외곽 알고리즘을 구현하라.
4. Graham 스캔을 외곽 상에 있는 한 점에서 시작하는 것이 정말로 필요한가? 답을 제시하고 그 이유를 설명하라.
5. 패키지-포장법을 외곽 상에 있는 한 점에서 시작하는 것이 정말로 필요한가? 답을 제시하고 그 이유를 설명하라.
6. 블록 외곽을 찾는 Graham 스캔을 특히 비효율적을 만드는 한 점들의 집합을 그려라.
7. Graham 스캔법으로 어떤 단순 다각형의 정점들을 형성하는 점들의 블록 외곽을 찾을 수 있는가? 있다면 그 이유를 설명하고, 없다면 그 예를 들어 설명하라.
8. 만약 입력이 한 원 내에 무작위적으로(무작위적 극 좌표들을 사용하여) 분포되어 있다고 가정한다면, 내부제거법에서 사용할 수 있는 네 개의 점들은 무엇인가?
9. 각 점의 x 와 y 값이 0과 1000 사이에서 있을 같은 확률을 갖는 아주 많은 점들의 집합에 대한 Graham 스캔 방법과 패키지-포장법을 경험적으로 비교하라.
10. 내부 제거법을 구현하고 이 방법이 각 점의 x 와 y 값이 0과 1000 사이에 있을 같은 확률을 갖는 점 집합에 적용됐을 때 50개의 점이 남아 있기를 기대하려면 점 집합은 얼마나 큰 N 값을(얼마나 많은 점들을) 가져야 하는지를 경험적으로 찾아라.

26 장

구간 검색

평면상에 점들의 한 집합이 주어졌을 때, 이 점들 중 어느 점들이 특정 영역 내에 있는지를 물어 보는 것은 당연하다. “Princeton에서 50 마일 이내의 모든 도시들을 나열하라”는 질문은 만약 미국의 모든 도시에 해당하는 점들의 집합이 있다면 당연히 물어 볼 수 있는 그런 형태의 질문이다. 기하학적인 도형의 형태가 사각형에 국한되어 있을 때, 문제는 쉽게 비기하학적 문제들로 확장된다. 예로서, “6만불과 10만불 사이의 소득을 갖는 나이 21세에서 25까지의 모든 사람을 나열하라”는 질문은 사람의 이름, 나이, 소득 등을 갖는 데이터 파일에서 어떤 “점들이” 나이-소득 평면상에서 특정 사각형 안에 들어가는지를 묻는 것이다.

이 문제는 삼차원 이상으로도 확장 가능하다. 만약 태양으로부터 50광년 내의 모든 별들을 나열하기를 원한다면, 이 문제는 삼차원 문제가 되고 앞 절의 젊은 부자들의 경우 키가 크고 또한 여성인 부자들을 나열하기를 원하면, 이 문제는 사차원 문제가 된다. 사실상, 이와 같은 문제들의 차원은 매우 높을 수 있다.

일반적으로, 임의의 정렬된(ordered) 집합으로부터의 값들을 갖는, 애트리뷰트(attribute)들로 구성된 레코드(record)들의 집합이 있다고 가정하자.(이러한 것을 종종 데이터베이스(database)라고도 한다. 물론 이 데이터베이스란 용어에 대해, 보다 구체적이고 완벽한 정의들이 되어져 있다) 한 데이터베이스에서 지정된 애트리뷰트들의 집합에서 지정된 구간의 제약 조건들을 만족시키는 모든 레코드들을 찾는 것은 실제 응용 분야에서는 어렵고도 중요한 문제이다. 이 장에서는, 레코드들이 점들이고 애트리뷰트들이 그 점들의 좌표(coordinate)들이인 이차원의 기하학적 문제들을 집중적으로 다루고 이에 맞는 일반화 방법들에 대해 논하기로 한다.

우리가 살펴보고자 하는 방법들은 일차원에서 단일 키(key)들을 찾는 경우들에서 보아 온 방법들을 일반화한 것이다. 우리는 많은 질의(query)들이 동일한 점들의 집합에서 행해진다

고 가정한다. 그래서 이 문제는 두 부분이 필요하다. 하나는 선 처리(preprocessing) 알고리즘인데, 이 알고리즘은 주어진 점들을 구간 검색(range searching)이 효율적일 수 있는 구조로 만들어 주며, 또 하나는 구간 검색 알고리즘인데, 이 알고리즘은 임의의 주어진(다차원) 구간 내에 있는 점들을 찾기 위해 선 처리에서 만들어진 구조를 이용한다. 이렇게 분리되어 있기 때문에, 여러 다른 방법들의 비교가 어렵다. 왜냐 하면 각 방법의 총 소요 비용이 연관된 점들의 분포 뿐만이 아니라 질의들의 갯수와 성질에도 좌우되기 때문이다.

일차원에서의 구간 검색 문제는 특정 구조 내에 있는 모든 점들을 찾는 것이다. 이 검색은 모든 점들을 선 처리하여 정렬시킨 다음, 구조 내에 있는 모든 점들을 찾기 위해 그 구간의 양 끝 점들에 대해 이진 검색(binary searching)을 함으로서 수행된다. 또 한 가지 방법은 이진 검색 트리를 구성한 후에, 이 트리를 가지고 재귀적 운행(recursive traversal)을 수행하여 해당 구간 내의 점들을 찾아 주고 그 구간 밖의 부분들은 무시해 버리는 것이다. 이를 위해 필요한 프로그램은 단순한 재귀적 운행 프로그램이다.(4장과 14장을 보라) 만약 구간의 좌측 끝점이 루트(root)에서의 좌측 접이면, 좌측 서브트리(subtree)를 (재귀적으로)검색한다. 만약 끝점이 오른쪽일 경우에는 이와 유사한 방법으로 검색한다. 이 때, 점들이 해당 구간 내에 있는지를 알아보기 위해서 만나는 각 노드들을 조사한다.

```
int range(int v1, int v2)
{ return ranger(head->r, v1, v2); }
int ranger(struct node *t, int v1, int v2)
{
    int tx1, tx2, count = 0;
    if (t == z) return 0;
    tx1 = (t->key >= v1);
    tx2 = (t->key <= v2);
    if (tx1) count += ranger(t->l, v1, v2);
    if (tx1 && tx2) count++;
    if (tx2) count += ranger(t->r, v1, v2);
    return count;
}
```

경우에 따라서는, 이 연산들은 14장의 이진 검색 트리 사전의 구현에 추가되면 유용할 수도 있다. 그림 26.1은 이 프로그램이 한 샘플(sample) 트리 상에서 수행되었을 때 찾아진 점들을 보여준다. 리턴된(찾아진) 점들은 트리에 연결될 필요는 없다.

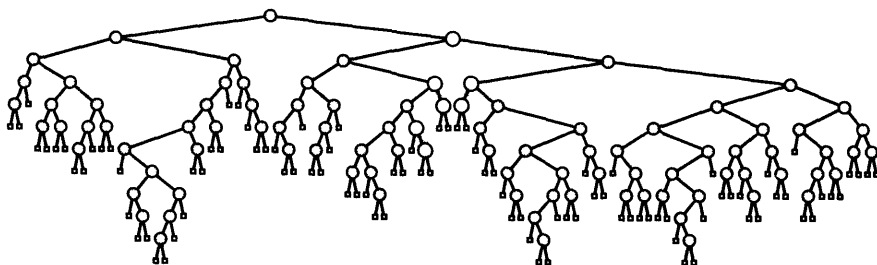


그림 26.1 이진 검색 트리를 이용한 일차원적 구간 검색

성질 26.1 일차원 구간 검색은 선처리는 $O(N \log N)$ 으로 행해지고 구간 검색은 $O(R + \log N)$ 으로 행해진다. 여기서 R 은 실제 구간 안에 있는 점들의 수이다.

이 성질은 검색 구조들의(14장, 15장을 보라) 기본 성질들로부터 바로 얻어진다. 바람직하다면, 균형 트리(balanced tree)가 이용될 수도 있다. \square

이 장의 목적은 다차원 구간 검색에 대해서도 이와 동일한 수행 시간을 얻으려는 것일 것이다. 파라미터 R 은 아주 중요할 수 있다: 구간 질의들을 위한 편의 기능(facility)이 주어진 경우, 사용자는 쉽게 모든 또는 거의 모든 점들을 필요로 할 수도 있는 질의들을 쉽게 만들 수 있다. 이런 유형의 질의는 많은 응용 분야에서 발생될 수 있음을 쉽게 예측할 수 있다. 그러나 모든 질의들이 이런 유형을 가진다면 복잡한 알고리즘들은 필요가 없다. 살펴볼 알고리즘들은 많은 수의 점들을 찾아 주지 않아도 되는(그렇게 예측되는) 질의들에 대해 효율적일 수 있는 알고리즘들이다.

기본 방법(Elementary Methods)

이차원에서의 구간은 한 평면에서의 영역이다. 단순성을 위해, x 좌표 점들은 주어진 x 구간 내에 있고 y 좌표 점들은 주어진 y 구간 내에 있는 모든 점들을 찾는 문제를 생각해 보자: 즉, 한 주어진 사각형 내에 있는 모든 점을 찾는 문제이다. 그래서, rect형(type)을 4개의 정수들-즉 수평선과 수직선의 끝점들-의 레코드라고 가정할 것이다. 수행되어야 할 기본 연산은 한 점이 주어진 사각형 내에 있는지의 여부를 시험하는 것이므로, 이를 확실하게 조사하여 만약 p 가 r 내에 있으면 0이 아닌 값을 리턴해 주는 함수 `insidirect(struct point p, struct rect r)` 또한 있다고 가정한다. 여기서 목표는 가능하면 `insidirect`를 거의 호출(call)하지 않고 주어진 사각형 내에 있는 모든 점들을 찾는 것이다.

이 문제 해결을 위한 가장 단순한 방법은 순차 검색(sequential search)이다; 모든 점들을 스캔하고, 이 각 점들이 지정 구간 내에 있는지의 여부를 시험한다.(각 점에 대해 insidertest를 호출함으로써) 실제로 이 방법은 많은 데이터베이스 분야에서 사용되는데, 그 이유는 이 방법이 모든 점들을 한번 스캔하고 많은 동일점들에 대한 질의들을 한꺼번에 시험하는 “묶음(batching)”구간 질의 방식에 의해 쉽게 향상될 수 있기 때문이다. 아주 큰 데이터베이스에서, 데이터가 외부 기억장치에 있고 그 데이터들을 읽는 시간이 비용을 좌우하는 중요한 이유인 경우에는 많이 질의들을 모아서 외부 기억장치에 있는 데이터 파일에 대해 한꺼번에 검색을 하는 이 방법이 아주 합리적인 것일 수 있다. 만약 이런 묶음처리가 잘 될 수 없거나 데이터베이스가 작은 경우에는 더 좋은 방법들이 있다.

그러나, 기하학적 문제의 경우에는, 순차 검색은 그림 26.2에 보여지는 것처럼 너무 작업량이 많은 듯 하다. 검색할 사각형에는 아주 소수의 점들만 있는 듯 하다. 그런데 이 소수의 점들을 찾기 위해 모든 점들을 다 검색 해야만 할 필요가 있는가? 순차 검색에 대한 한 단순한 개선책은 검색될 한 개 이상의 차원들에 대해 알려진 일차원 방법들을 곧 바로 적용하는 것이다. 이의 수행을 위한 한 가지 방법은 x 좌표 값이 사각형에 의해 규정된 x 의 구간 내에 있는 모든 점들을 찾고 이 점들이 주어진 사각형 안에 있는지를 알기 위해 이 점들의 y 좌표 값들을 조사하는 것이다. 이렇게 하면 사각형 안에 있지 않은 점들은 x 좌표 값이 x 구간을 벗어나기 때문에 절대로 조사되지 않는다. 이 기법을 투영(projection)기법이라 한다; 물론 이 방식을 y 구간에 대해 적용시켜도 된다. 한 예로서, 위에 설명한 것처럼, x 구간의 경우

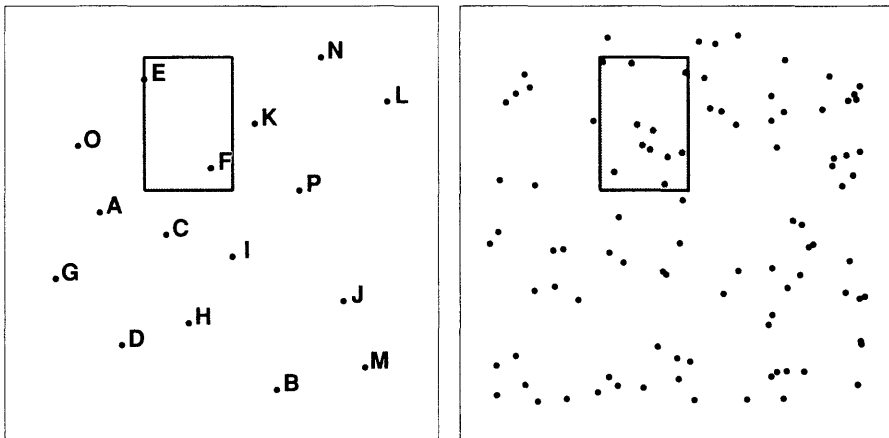


그림 26.2 이차원 구간 조사

E C H F 와 I가 조사될 수도 있다. 그리고 y 구간의 경우라면, O E F K P N과 L이 조사될 수도 있다. 이 찾아진 점들(E와 F)의 집합은 바로 이 두 투영에 나타나 있는 점들임에 주목하라.

만약 점들이 사각형의 영역 내에서 균등하게 분포되어 있다면, 조사될 점들의 평균 수를 계산하는 것이 아주 쉬울 것이다. 한 주어진 사각형 내에서 찾아질 점들의 비율은 전체 영역에 대한 그 사각형의 비율이다; x 축에 대한 투영의 경우 조사 되어 할 것으로 예측되는 점들의 비율은 그 영역의 폭(너비)에 대한 그 사각형의 폭의 비율이다. 예로서, 16×16 의 영역에서 4×6 의 사각형을 사용한다는 것은 사각형에서 $3/32$ 의 비율만큼의 점들을 찾을 수 있을 것으로 기대됨-즉 x 축 투영에서 $1/4$, y 축 투영에서 $3/8$ 만큼-을 의미한다. 이같은 상황하에서는 두 사각형의 차원 중 더 작은 축에 대해 투영을 시도하는 것이 최고이다. 또 한 편으로는, 이러한 투영 기법이 실패할 수 있는 경우를 만들어 내는 것 또한 쉽다. 예로서, 만약 점들의 집합이 “L”자형이고 검색은 단지 “L”자의 구석 부분에 있는 점들만을 포함하는 구간에서만 행해진다면, 어느 축에서 투영을 해도 단지 점들의 반만큼만 제거된다.

처음 보기에는, 투영 기법은 x 구간 내의 점들과 y 구간 내의 점들을 어떠한 “교차(intersect)”시킴으로써 개선될 수도 있을 듯 싶다. 모든 점이 x 범위 내에 있거나 모든 점이 y 범위 내에 있는 최악의 경우에 대한 조사없이 이런 시도들을 하는 것은 우리로 하여금 앞으로 공부할 보다 복잡한 방법들을 살펴보게 한다.

격자법(Grid Method)

평면에 있는 점들간의 인접 관계들을 유지하기 위한, 한 간단하고 효과적인 기법은 검색될 영역을 조그만 정사각형으로 나누고 이 각 정사각형 내의 점들에 대해 짧은 리스트들을 유지하는 한 격자를 인위적으로 형성하는 것이다.(한 예로서, 이 기법은 고고학에서 사용된다) 그리고 나서, 주어진 사각형 내에 있는 점들이 찾아지면, 그 사각형을 교집합 되는 정사각형에 해당하는 리스트들만 검색하면 된다. 우리 예제에서는 그림 26.3에서 보여지는 바와 같이 단지 E, C, F와 K만이 검사된다.

남아 있는 중요 결정 사항은 격자의 크기를 정하는 것이다: 만약 너무 크면 각 정사각형 격자가 너무 많은 점들을 포함할 것이고, 만약 너무 작으면 검색할 격자 수가 너무 많게 될 것이다.(이중지 대부분은 점들이 없는 빈 격자 일수도 있다) 이 양 극간의 균형을 유지할 수 있는 한 가지 방법은 격자의 수가 총 점들의 수에 대해 한 상수 비율을 갖도록 격자의 크기

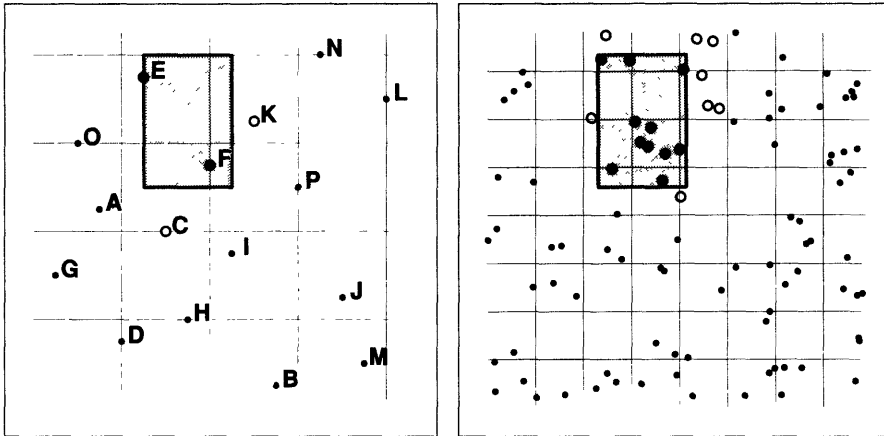


그림 26-3 구간 검색을 위한 격자법

를 선택하는 것이다. 그러면 각 정사각형 격자 내의 점들의 수는 어떤 작은 상수 치에 가깝게 될 것이다. 우리의 작은 샘플 점 집합의 예에서, 16개의 점에 대해 4×4의 격자를 사용하면 이는 각 격자가 평균적으로 1개의 점을 가질 것임을 의미한다.

아래에 격자 법을 이용하여 이차원 공간에서 구간 검색을 지원하는 한 클래스의 간단한 구현(프로그램)이 있다. 변수 maxG와 size는 격자의 해상도(resolution)(즉, 정사각형 격자의 수와 크기)를 제어하는데 사용된다. 여기서는 정수 좌표계를 사용하고 size는 정사각형 격자의 폭으로 가정한다. maxG×maxG개의 정사각형 격자가 있으므로 점들에 대한 좌표계의 구간은 0에서 size×maxG이다. 어느 격자에 어느 점이 속하는지를 찾기 위해 좌표계는 size로 나누어진다:

```
class Range
{
private:
    struct node
    { struct point p; struct node *next; };
    struct node *grid[maxG][maxG];
    struct node *z;
public:
    Range();
    void insert(struct point p);
    int search(rect range);
};
```

```

};
Range::Range()
{
    int i, j;
    z = new node;
    for(i = 0; i <= maxG; i++)
        for(j = 0; j <= maxG; j++)
            grid[i][j] = z;
}
void Range::insert(struct point p)
{
    struct node *t = new node;
    t->p = p; t->next = grid[p.x/size][p.y/size];
    grid[p.x/size][p.y/size] = t;
}

```

이 프로그램은 표준형 연결-리스트 표현 방식을 사용하는데, grid 배열 내에 모조(dummy) 테일(tail)노드 z와 리스트 헤더들을 가진다. 리스트들은 무순서(unordered)이며 3장에서 본 바와 같이 앞에서부터 삽입이 행해진다.

변수 size와 maxG를 적절히 설정함은 점들의 갯수, 사용 가능 기억장치의 양, 그리고 좌표 값들의 구간 등에 달려 있다. 대략, 만약 N 개의 점이 있고, 격자당 M 개의 점이 있기를 원하면 약 N/M 개의 격자를 필요로 하게 되어 maxG는 $\sqrt{N/M}$ 에 가장 가까운 정수로서 선택되어야 하고 size는 최대 좌표 점을 $\sqrt{N/M}$ 으로 나눈 값에 근사한 값이어야 한다. 이는 결과적으로, 약 N/M 개의 정사각형 격자가 있게 한다. 이러한 추정치들은 작은 값을 가진 파라미터들의 경우에는 정확하지가 않다. 그러나 대부분의 경우에는 유용하며 특수한 경우의 응용 분야들에 대해서도 이와 유사한 추정치들이 쉽게 얻어질 수 있다. 사용되는 값들은 아주 정확히 계산될 필요는 없다-보통 격자당 점들의 갯수가 최대로 두 배까지 밖에 안되는 반면에 size와의 곱셈과 나눗셈이 아주 효율적이 되기 때문에 size 값으로 2의 제곱을 이용한다.(위의 주어진 예제의 경우에는 1에서 2로)

위의 예제에서는 보통 $M = 1$ 을 사용한다. 만약 공간이 크다면, 더 큰 값이 적합할 수도 있다. 그러나 더 작은 값은 아주 특수한 경우들을 제외하고는 유용하지 않다.

이제 구간 검색에 대한 대부분의 작업은 아래와 같이 grid배열을 인덱싱함으로써 다음과 같이 간단히 처리된다:

```

int Range::search(struct rect range) // Grid method
{
    struct node *t;
    int i, j, count = 0;
    for (i = range.x1/size; i <= range.x2/size; i++)
        for (j = range.y1/size; j <= range.y2/size; j++)
            for (t = grid[i][j]; t != z; t = t->next)
                if (insidirect(t->p, range)) count++;
    return count;
}

```

이 프로그램은 단순히 구간 내의 점들의 갯수를 전역변수 count를 이용하여 계수한다. 이 프로그램을 출력하게 하거나 또는 구간 내의 모든 점들을 리턴하게 수정하는 것은 아주 간단하다.

성질 26.2 구간 검색을 위한 격자법은 구간 내의 점들의 갯수에 평균적으로는 선형적(linear)이고, 최악의 경우에는 점들의 총 수에 선형적이다.

이 말은 파라미터들의 선택에 따라 다르다는 것으로서 각 격자 내의 점들의 예상 갯수는 위에 설명한 것처럼 상수이다. 만약 검색 사각형 내의 점들의 갯수가 R 이면, 검사될 격자들의 수는 R 에 비례할 것이다. 검색 사각형 내에 완전하게 들어가 있지 않은, 검사될 격자의 수는 분명히 한 작은 상수 $\times R$ 에 선형적이다. 큰 R 값의 경우, 검색 사각형 내에 있지 않은 검사될 격자들의 수는 아주 적게 될 것이다. 모든 그런 점들은 검색 사각형의 선분과 교차하는 격자 내에 있으며, 그와 같은 격자들의 수는(큰 R 의 경우) \sqrt{R} 에 비례한다. 이 말은 격자가 아주 작은 경우나(사각형 내에 너무 많은 빈 격자가 있을 수 있다), 아주 큰 경우,(검색 사각형의 경계 상의 격자들 내에 너무 많은 점들이 있을 수 있다) 또는 만약 검색 사각형이 격자보다 더 작은 경우(이런 경우, 그 사각형은 그 안에 점들이 거의 없는 아주 많은 격자들과 교차할 수 있다)에는 맞지 않는다. \square

격자법은 만약 점들이 가정된 구간 내에 잘 분산되어 있다면 잘 동작한다. 그러나 만약 점들이 군데군데 몰려 있으면 잘 동작 할 수 없다.(예를 들면, 모든 점들이 한 격자 내에 있을 수 있다. 이 경우는 모든 격자법의 사용이 아무 의미가 없음을 뜻한다) 다음에 우리가 조사

할 방법은 즉시 점들의 집합에 순응하여 비 균등한 방법으로 공간을 다시 나눔으로서 이러한 최악의 경우가 잘 발생하지 않도록 하는 방법이다.

이차원 트리(Two-Dimensional Tree)

이차원(2D) 트리들은 동적이며 이진 트리와 아주 유사한 융통성있는(adaptable) 데이터 구조이면서도 구간 검색이나 다른 문제들에 사용하기에 편리한 방법으로 기하학적 공간을 분할한다. 이 방법의 개념은 x 와 y 좌표의 점들을 정확히 한번씩 번갈아 가면서 키값으로 사용하여 노드내에 점들을 갖는 이진 검색 트리들을 형성하는 것이다.

보통의 이진 검색 트리에서와 똑같은 알고리즘이 2D 트리에 점들을 삽입하기 위해 사용되나, 루트에서만은 y 좌표를 사용한다.(만약 삽입될 점이 루트에 있는 점보다 더 작은 좌표값을 가지면, 좌측으로 가고, 아니면 우측으로 간다) 그리고 나서, 다음 번 계층에서는 x 좌표를 또 그 다음 번에는 y 좌표를 사용하는 식으로, 한 외부(external) 노드와 만날 때까지 계속한다. 그림 26.4는 우리의 작은 샘플 점 집합에 상응하는 2D 트리를 보여 주고 있다.

이 기법의 중요성은 이 기법이 평면을 간단히 분할해 주는 것과 같은 역할을 한다는 것이다: 루트에 있는 점 아래의 모든 점들은 좌측 서브트리로 가고, 위의 모든 점들은 우측 서브트리로 간다. 그리고 나서 루트에 있는 점 위에 있으면서 우측 서브트리에 있는 점의 좌측에 있는 모든 점들은 루트의 우측 서브트리의 좌측 서브트리로 같다. 이런 방식으로 하여 계속 진행이 된다.

그림 26.5과 26.6은 어떻게 평면이 분할되어져서 그림 26.4에 있는 트리와 일치되는지를 보여준다. 먼저 A의 y 좌표에서 수평선이 그어지고, 첫 노드가 삽입된다. 그리고 나서, B는 A 밑에 있으므로, 트리에서 A의 좌측으로 간다. 다시 A밑의 반쪽 평면에 B의 x 좌표에서 수직

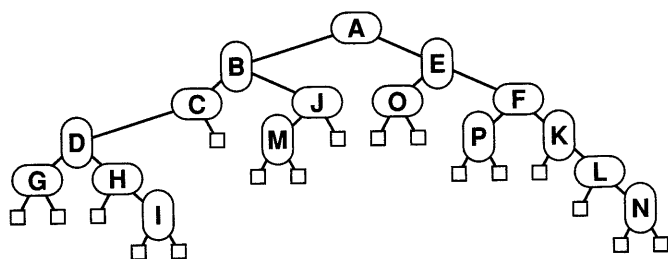


그림 26.4 2차원 트리

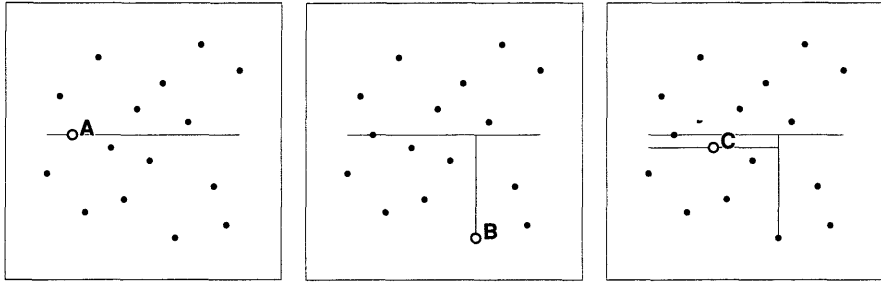


그림 26.5 2D 트리를 이용한 평면 분할: 초기 스텝들

선이 그어진다.(그림 26.5의 2번째 다이어그램) 계속하여 C는 A밑에 있으므로 루트에서 좌측으로 가는데, 또한 C는 B의 좌측에 있으므로 B에서 좌측으로 간다. 그리고 A밑의 평면과 B의 좌측부를 C의 y좌표에서 수평선을 그어 분할한다.(그림 26.5의 3번째 다이어그램) 이런 식으로 계속하여 분할이 진행된다.

모든 트리의 외부 노드는 평면에서의 한 사각형에 해당한다. 각 평면상의 영역(region)은 트리에서의 외부 노드에 해당한다; 각 점은 그 점에서 행해진 분할을 정의하는 수평선 또는 수직선 상에 있다.

2D트리를 형성하는 프로그램 코드는 보통 이진 트리의 검색/삽입 알고리즘을 각 트리 계층에서 x와 y좌표를 번갈아 가며 수행되도록 수정하여 간단히 구현된다:

```
class Range
{
private:
    struct node
    { struct point p; struct node *l, *r; };
    struct node *z, *head;
    struct point dummy;
    int searchr(struct node *t,
                struct rect range, int d);
public:
    Range();
    void insert(struct point p);
    int search(rect range);
};

void Range::insert(struct point p) // 2D tree
```

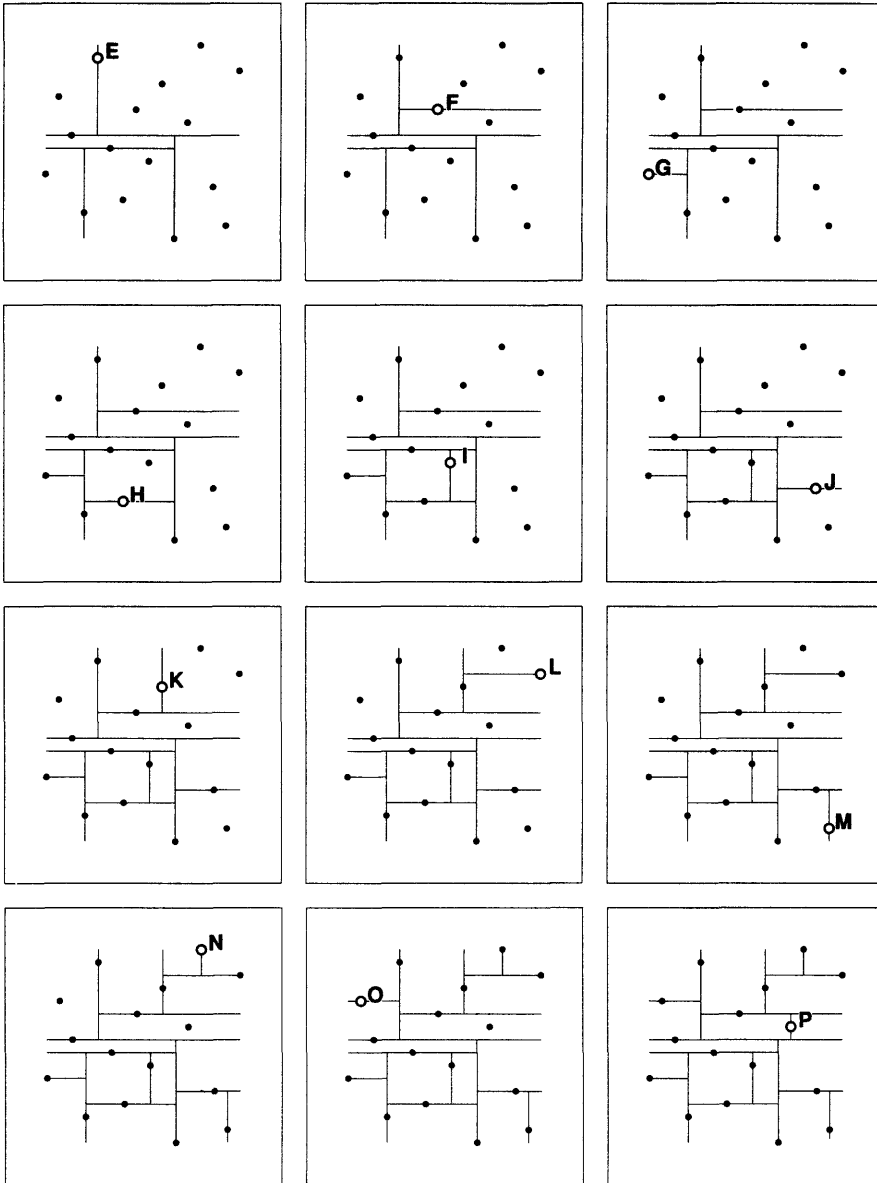


그림 26.6 2D 트리를 이용한 평면 분할: 계속


```

{
    struct node *f, *t; int d, td;
    for (d = 0, t = head; t != z; d != d)
    {
        td = d ? (p.x < t->p.x) : (p.y < t->p.y);
        f = t; t = td ? t->l : t->r;
    }
    t = new node; t->p = p; t->l = z; t->r = z;
    if (td) f->l = t; else f->r = t;
}

```

이 클래스에 있는 public 인터페이스는 앞에서 설명한 격자법의 경우와 동일하나, 이의 구현은 이진 트리들을 사용했다.

성질 26.3 N 개의 무작위적 점들로부터 2D트리를 형성하려면 평균적으로 $2N \ln N$ 번의 비교가 필요하다.

정말로, 무작위로 분포된 점들에 대해, 2D트리들은 이진 트리와 동일한 성능 특성을 가진다. 두 좌표계는 모두 무작위 키(key)값들로서 작용한다. □

2D트리를 사용하여 구간 검색을 하려면, 먼저 N 개의 점들을 초기의 빈 트리에 삽입해야 한다. 이 초기화 프로그램 코드는 트리를 통해 이동하는(travel) 프로그램 코드에 대한 시작 조건들과 조심스럽게 조정되어야 한다. 그렇지 않으면, 트리가 y 좌표 값들을 갖는 경우에 x 좌표 값들을 찾는 알고리즘에서 또는 이와 반대의 경우에 성가신 버그(bug)가 발생할 것이다.

그리고 나서, 구간 검색을 위해, 각 노드에 있는 점들을 그 노드의 평면을 분할하는데 사용되는 차원의 구간에 대해 시험한다. 예로서, 루트의 좌측으로 그리고 노드 E에서 우측으로 가 보자. 왜냐하면 검색 사각형은 전적으로 A의 위와 E의 우측에 있기 때문이다. 그런 후, 노드 F에서는, 양쪽의 서브트리 모두로 가 봐야 한다. 왜냐하면 F는 그 사각형에 의해 정의된 x 구간 내에 있기 때문이다.(이 말은 F가 그 사각형 안에 있다는 말과 같지 않음을 주목하라) 이제 P와 K의 좌측 서브 트리들이 조사되는데, 이것은 검색 사각형과 일부 겹치는(overlap) 평면 영역들을 조사하는 것에 해당한다.(그림 26.7과 26.8을 보라)

이 과정은 이 장의 시작 부에서 본 D range 프로시저를 단순하게 일반화하여 쉽게 구현할 수 있다:

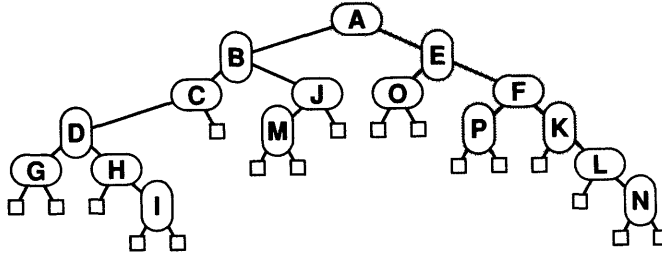


그림 26.7 2D트리에서의 구간 검색

```
int Range::search(struct rect range) // 2D tree
{ return searchr(head->r, range, 1); }
int Range::searchr(struct node *t, struct rect range, int d)
{
    int t1, t2, tx1, tx2, ty1, ty2, coun = 0;
    if (t == z) return 0;
    tx1 = range.x1 < t->p.x; tx2 = t->p.x <= range.x2;
    ty1 = range.y1 < t->p.y; ty2 = t->p.y <= range.y2;
    t1 = d ? tx1 : ty1; t2 = d ? tx2 : ty2;
    if (t1) count += searchr(t->l, range, !d);
    if (insidirect(t->p, range)) count++;
    if (t2) count += searchr(t->r, range, !d);
    return count;
}
```

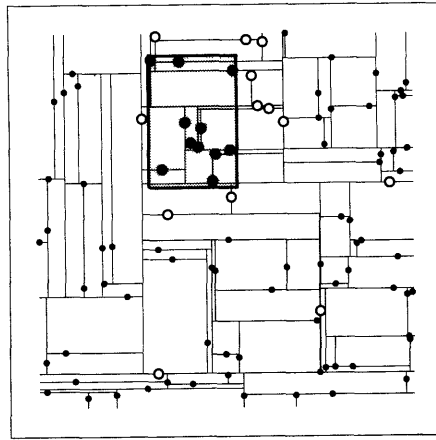
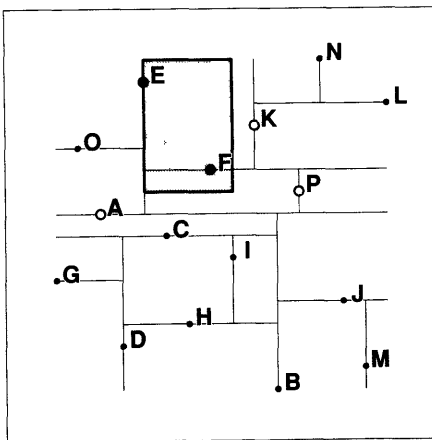


그림 26.8 2D 트리에서의 구간 검색(평면 세부분할)

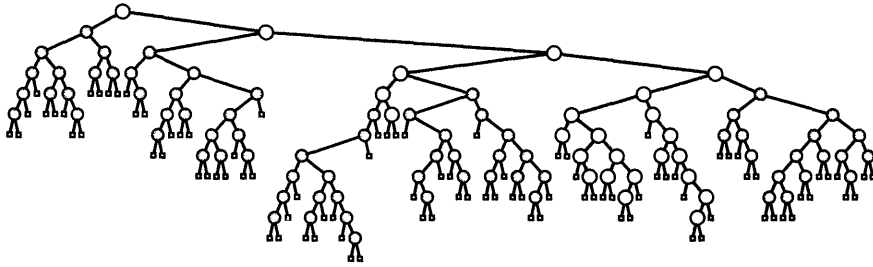


그림 26.9 큰 2D 트리에서의 구간 검색

이 프로시저는 선의 분할이 사각형을 자르는(cut) 경우에만 양쪽 서브트리 모두로 가는데, 이런 경우는 상대적으로 작은 사각형들에서는 잘 발생하지 않는다. 그림 26.8은 우리의 두 예제들에서 검색된 평면의 세부분할들과 점들을 보여주고 있다.

성질 26.4 2D트리에서의 구간 검색은 N 개의 점을 가진 영역내에서 적당한 구간 내의 R 개의 점을 검색하기 위해 $R + \log N$ 정도의 스텝을 사용하는 듯 하다.

이 방법은 아직 더 분석되어야 하고, 언급된 성질 또한 경험적 증거에 근거한 추측(conjecture)이다. 물론 성능(그리고 분석)은, 항상 그렇지만, 사용된 구간의 형태에 따라 상당히 좌우된다. 그러나 이 방법은 격자법과 비교해 볼 수 있을 정도의 경쟁력(competition)이 있고 점 집합의 무작위성에 다소 덜 좌우된다. 그림 26.9는 우리의 큰 예제에 대한 2D 트리를 보여주고 있다. □

다차원 구간 검색(Multidimensional Range Searching)

격자법과 2D트리들은 이차원 보다 더 일반화 될 수 있다. 위 알고리즘들에 아주 직접적이고도 단순한 확장을 하면 즉시 다차원에서 동작하는 구간 검색 방법들을 만들 수 있다. 그러나 다차원 공간의 성질상 약간의 주의가 요구되고 그 알고리즘들의 성능 특성들이 특정 분야의 경우에는 예측하기 어려운 수도 있다.

k -차원 검색을 위한 격자법의 구현을 위해서는, grid를 k -차원 배열로 바꾸고 차원당 한 개씩의 인덱스를 사용한다. 주된 문제는 size에 대해 적당한 값을 찾는 것이다. 이 문제는 k 값이 클 경우에는 아주 명백해진다: 10-차원 검색을 위해 어떤 유형의 격자를 사용해야 하

는가? 문제는 차원당 단지 3번의 나눗셈을 한다 하더라도, 3개의 정사각형 격자가 필요하다는 것이다. 물론 합리적인 N 값의 경우 이들 중 대부분은 비어 있을 것이다.

2D를 k D트리로 일반화하는 것 또한 간단하다: 트리를 타고 내려가며 진행하는 동안(우리가 2차원에서 x 와 y 좌표를 번갈아 가며 했듯이) k 차원의 경우도 동일한 방법을 시도한다. 전과 마찬가지로, 무작위적 상황에서는, 결과로서 생긴 트리는 이진 검색 트리와 동일한 특성을 지닌다. 또한 트리와 단순한 기하학적 과정간의 자연적인 상용관계가 있다. 삼차원에서는, 각 노드에서의 분기(branching)는 한 평면에서 해당 삼차원 영역을 자르는(절단하는) 것에 해당한다; 일반적으로, $(k-1)$ 차원의 하이퍼평면(hyperplane)으로 k -차원의 영역을 자를 수 있다.

만약 k 가 아주 크다면, k D 트리들에 아주 심각할 정도의 불균형이 생길 확률이 있다. 왜냐하면 실제적인 점 집합들은 매우 큰 차원들의 경우에 무작위성을 보여줄 정도로 클 수는 없기 때문이다. 통상적으로, 한 서브트리 내의 모든 점들은 여러 차원들에 걸쳐서 동일한 값을 가질 것이다. 이 결과로서 트리들에 여러 개의 단방향(one-way) 분기들이 생긴다. 이 문제를 완화시키는 한 가지 방법은, 여러 차원들에 대해 계속하여 돌아가며 순환 수행하는 대신에, 점 집합을 최적의 방법으로 나누는 차원을 사용하는 것이다. 또한 이 기법은 2D트리들에도 적용될 수 있다. 그 대신 기법은 차원을 구분해 줄 수 있는 그러한 정보를 각 노드에 추가해야만 한다. 그러나 이 기법은 불균형 문제를 경감해 준다. 특히 고차원 트리들의 경우에는 더욱 그렇다.

요약하면, 다차원 문제들을 처리할 수 있도록 구간 검색을 위한 프로그램들을 어떻게 일반화하는지를 아는 것은 쉽다. 이와 같은 과정은 큰 응용 분야의 경우에는 가볍게 다루어져서는 안된다. 레코드당 많은 애트리뷰트들을 갖는 대용량 데이터베이스는 정말로 매우 처리하기 어려운 것일 수 있다. 그리고 특정 분야에 대한 효율적인 구간 검색 방법을 개발하려면 데이터베이스의 특성을 아주 잘 이해하는 것이 필요하다. 이 문제는 아직도 활발히 연구되고 있는 아주 중요한 문제이다.

연습문제

1. 본문의 1D range 프로그램에 대한 비재귀적(nonrecursive) 버전을 작성하라.
2. 한 지정된 구간 내에 있지 않은 이진 트리에 있는 모든 점들을 프린트하는 프로그램을 작성하라.
3. 격자법에서 정사각형 격자들과 검색 사각형의 차원들의 함수들로서 검색될 정사각형 격자의 최대 수 및 최소 수를 구하라.
4. 연결-리스트를 이용하여 빈 정사각형 격자의 검색을 피하기 위한 방법을 논하라: 각 정사각형 격자는 동일 행에 있는 다음 번 비어 있지 않은 정사각형 격자와 그리고 동일 열에 있는 다음 번 비어 있지 않은 정사각형 격자와 연결되어 있을 수 있다. 이와 같은 방식의 사용시 이용될 정사각형 격자의 size는 어떤 영향을 미치는가?
5. 이 장의 샘플 점들에 대해 2D트리를 형성한다고 할 때, 그 때의 트리와 평면의 분할 결과를 그려라.
6. 두 개의 자식 노드를 가지는 어떤 노드가 없는 최악의 2D트리를 야기시키는 점들의 집합을 구하라. 결과로서 나타나는 평면의 분할 결과는?
7. 한 주어진 원 내에 있는 모든 점들을 찾아 주기 위해(리턴 시키기 위해), 이 장에서 소개한 각각의 방법들을 어떻게 수정해야 할지를 설명하라.
8. 동일 영역 내의 모든 검색 사각형들 중에서, 어떤 형태가 이 장에서 소개한 각각의 방법들이 최악의 수행을 하게 만들까?
9. 큰 그룹들 내에서 점들이 군집(cluster)을 이루어 멀리 떨어져 있을 때, 어떤 방법이 구간 검색이 좋을까?
10. 초기의 빈 트리에 점 (3,1,5), (4,8,3), (8,3,9), (6,2,7), (1,6,3), (1,3,5), (6,4,2)들이 삽입될 때, 그 결과로서 나타나는 3D트리를 그려라.

기하학적 데이터를 갖는 응용 분야에서 빈번히 발생하는 한 가지 자연적인 문제는 “ N 개의 객체의 집합에서, 임의의 두 객체가 서로 교차하는가”하는 문제이다. 이 객체들은 선, 사각형, 원, 다각형, 또는 다른 형태의 기하학적 객체들일 수 있다. 우리는 물리적 객체들이 동시에 같은 곳에 위치할 수 없음을 쉽게 인지할 수가 있다. 그러나 컴퓨터 프로그램들은 이러한 인지를 위해 상당한 시간과 노력을 필요로 한다. 예로서 IC나 PCB들을 설계하고 처리하는 시스템에서는, 어떤 두 선이 교차하면 단락(short circuit) 현상이 생김을 아는 것이 중요하다. 수치적으로 제어되는 절삭 도구에 의해 만들어질 레이아웃(layout)을 설계하는 산업용 시스템에서는 레이아웃의 어느 두 부분도 교차하지 않음을 아는 것이 중요하다. 컴퓨터 그래픽스 분야에서는, 객체 집합 내의 어느 객체가 특정 지점에서 볼 때 잘 보이지 않는가를 찾아내는 문제는 객체들을 가시 평면(viewing plane)상으로 투영했을 때 나온 결과물(투영물)들에 대한 한 기하학적 교차 문제로서 공식화되어(formulate) 나타날 수 있다. 물리적 객체들이 비록 없는 경우라도, 문제를 수학적으로 공식화했을 때 자연스럽게 기하학적 교차 문제가 되는 많은 예들이 있다. 한 특별히 중요한 이러한 경우의 예를 보려면 43장을 보라.

교차 문제에 대한 확실한 해결책은 각 쌍의 객체들이 교차하는지의 여부를 조사하는 것이다. 약 $N^2/2$ 쌍의 객체들이 있으므로, 이 교차 여부를 조사하는 알고리즘의 수행 시간은 N^2 에 비례한다. 이것은 다른 인자들이 처리될 객체들의 수를 제한시켜 주므로 문제가 되지 않을 수가 있다. 그러나, 많은 응용 분야들의 경우, 수십만 또는 수백만 개의 객체들을 다루는 일이 발생한다. 이런 경우 주먹구구식 N^2 알고리즘은 확실히 적합하지 않다. 이 장에서는, N 개의 객체의 집합 내의 임의의 두 객체가 교차하는지의 여부를 $N \log N$ 에 비례하는 시간

내에 알아내는, 한 일반적인 방법을 살펴보기로 한다. 이 방법은 M. Shamos와 D. Hoey가 1976년 발표한 독창적인 논문에 실린 알고리즘들에 근거한 것이다.

먼저, 수평 또는 수직인 선들의 집합에서 교차하는 두 선들의 갯수를 세는 알고리즘을 보기로 한다. 이 문제는 어떤 점에서는(수평선 및 수직선들로 구성되어 상대적으로 간단하므로) 쉽고, 또 어떤 점에서는(모든 교차하는 선들의 쌍을 세는 것이 단지 교차하는 쌍들의 존재 여부를 찾는 것보다는 더 어렵기 때문에) 어렵다. 전 장에서와 마찬가지로, 프로그램 코드를 좀 더 간단히 하기 위해 갯수 세기(counting) 문제만(모든 교차점들에 대한 정보 대신) 생각하기로 한다—모든 교차점들에 대한 정보를 보여주도록 프로그램을 확장하는 것은 간단하다. 구현 방법으로서 이진 검색 트리 및 전 장의 중복적 재귀 프로그램의 구간-검색 프로그램이 적용된다.

다음으로는, N 개의 선분들의 집합에서 임의의 두 선분의 교차 여부를 찾는 문제를 살펴보기로 한다; 이 경우에는 선분들에 대한 어떤 제약 조건도 없다. 수직-수평선들의 경우에 사용한 것과 동일한 전략이 적용될 수 있다. 사실상, 많은 다른 형태의 기하학적 객체들의 교차 여부를 찾는 데 있어서 동일한 기본 개념이 이용된다. 그러나 선분들과 다른 객체들의 경우에는, 모든 교차하는 객체 쌍들을 알려주도록 확장할 때, 수직-수평선들의 경우 보다 다소 복잡하다.

수평-수직선들(Horizontal and Vertical Line)

우선, 모든 선들이 수평선이거나 수직선이라고 가정하자: 각 선분을 정의하는 두 점은 그림 27.1에 있는 두 다이어그램에서 보여지는 바와 같이 같은 x 좌표를 갖거나 같은 y 좌표를 가진다.(이를 Manhattan 기하학이라고도 한다. 왜냐하면 Broadway와는 달리 Manhattan가의 지도는 대개 수평선과 수직선들로 구성되어 있기 때문이다) 선분이 수평선이나 수직선이 되도록 제한함은 확실히 심각한 제약 조건이다. 그러나 이것은 간단한 “장난감”같은 문제가 아니다.

실제로 이 제약 조건은 특정 분야에 부가될 수 있다: 예로서, 대용량 집적 회로(VLSI)들은 보통 이 제약 조건하에서 설계된다. 오른쪽 그림의 선분들은, 많은 분야들에서 그렇듯이, 상대적으로 짧다.

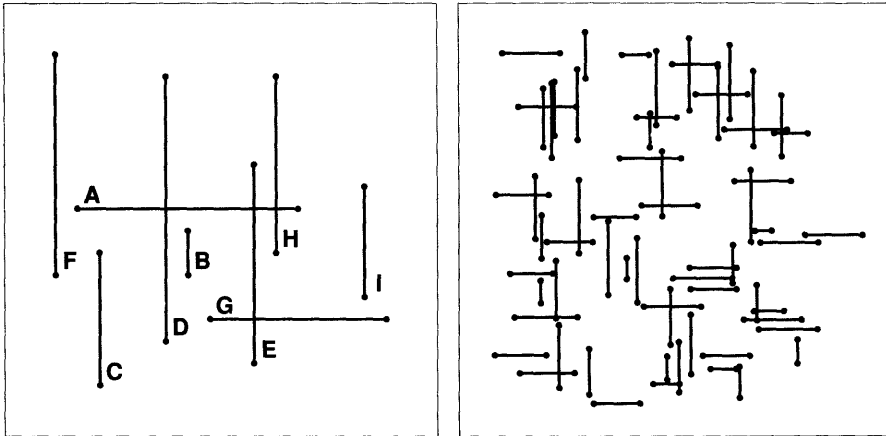


그림 27.1 두 선분 교차 문제들(Manhattan)

이러한 수평선 및 수직선들의 집합들에서 교차하는 선분들을 찾는 알고리즘은 일반적으로 밑에서부터 위로 진행하는 수평 주사선(scan line)이 있다고 여긴다. 이 주사선 상에 투영될 때, 수직선들은 점이고 수평선들은 구간(interval)들이 된다: 주사선이 밑에서 위로 진행함에 따라, (수직선들을 나타내는) 점들이 나타났다가 사라졌다가하며 수평선들은 주기적으로 마주치게 된다. 주사선 상에서 한 수직선을 나타내는 한 점과 한 수평선을 나타내는 한 구간이 만나면, 이 두 선분이 교차한 것이다. 한 점과 만난다는 것은 수직선이 주사선을 교차하고 수평선이 그 주사선 상에 있어서 결국 그 수평선과 수직선이 교차해야 함을 의미한다. 이런 식으로, 교차선들의 쌍을 찾는 이차원적 문제는 전장에서 본 일차원 구간-검색 문제로 축약된다.

물론, 실제로는 수평선들을 선분들의 집합 상에서 주사(scan)할 필요는 없다; 단지 선분들의 끝점들과 마주칠 때만 처리하면 되므로, 이 선분들을 자신들의 y 좌표에 따라 정렬하고, 이 정렬 순서에 따라 선분들을 처리하면 된다. 만약 한 수직선의 하부 끝점과 마주치면, 이 선의 x 좌표를 이진 검색 트리에 추가한다.(여기서는 이 트리를 x -트리라 부른다); 만약 이 수직선의 상부 끝점과 마주치게 되면, 이 선분을 트리에서 제거한다. 그리고 만약 한 수평선과 만나면, 이 선분의 두 개의 x 좌표 값을 이용하여 구간-검색을 한다. 앞으로 알게 되겠지만, 선분의 끝점들이 동일한 좌표 값을 가질 때 조심스럽게 처리해야 한다.

그림 27.2는 그림 27.1의 왼쪽 예제에서 교차선들을 찾기 위해 주사해 나가는 처음 몇 스텝들을 보여준다. 주사는 최소 y 좌표 값을 갖는 점(C의 하부 끝점)에서부터 시작하여, 다음에는 E, 그 다음에는 D와 만난다. 이 나머지 과정은 그림 27.3에 있다: 다음에 만나는 선분은

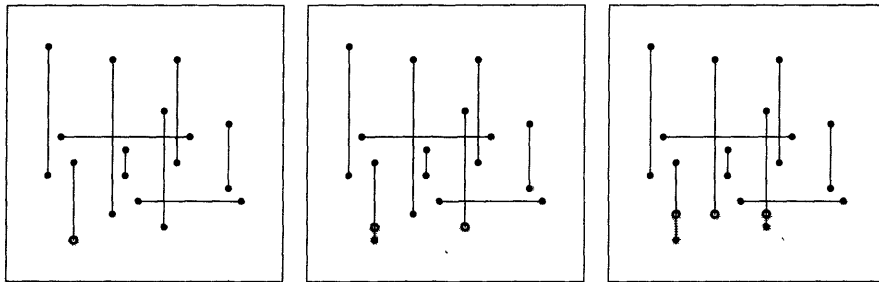


그림 27.2 교차선들을 찾기 위한 주사: 초기 스텝들.

수평선 G인데, 이 선과 C, D 그리고 E(즉, 주사선들을 교차하는 수직선들)와의 교차 여부가 조사된다.

이 주사 방식의 구현을 위해서, 우리는 단지 선분들의 끝점들의 y 좌표 값만을 정렬하면 된다. 우리 예제의 경우, 이렇게 하면 리스트

C E D G I B F C H B A I E D H F

가 얻어진다. 이 리스트에서 각 수직선은 두 번씩 나타나고 각 수평선은 한 번씩 나타난다. 선분 교차 알고리즘을 위한 목적의 경우, 이 정렬된 리스트는 일련의 “삽입”(수직선들의 하부 끝점들과 만날 때), “삭제”(수직선들의 상부 끝점들과 만날 때), 그리고 “구간”(수평선들의 끝점들에 대한) 명령들로서 간주될 수 있다. 모든 이러한 명령들은 키값으로서 x 좌표를 사용하여 14장과 26장에 있는 표준형 이진 트리 루틴들을 호출하는 것과 같다.

그림 27.4는 주사시의 x -트리 형성 과정을 보여준다. 이 트리의 각 노드는 한 수직선에 해당한다 - 이 트리에서 사용된 키는 x -좌표 값이다. E는 C의 오른쪽에 있으므로, E는 C의 서브트리이다 등등. 그림 27.4의 첫 번째 줄의 트리들은 그림 27.2에 해당하고 나머지는 그림 27.3에 해당한다.

한 수평선과 만나면, 이 선을 이용하여 트리에서 구간 검색을 한다: 이 수평선에 의해 표시되는 구간 내의 모든 수직선들은 각각이 하나의 교차에 해당한다. 우리 예제에서는, E와 G가 교차함이 발견되고, I, B 그리고 F가 삽입된다. 그리고 나서 C가 삭제되고, H가 삽입되고, B가 삭제된다. 그 다음에 A와 만나게 되고, 이 A와 D, E 및 H가 교차함이 발견된다. 그리고 다음으로, I, E, D, H와 F의 상부 끝점들이 삭제되어 다시 빈(empty) 트리가 된다.

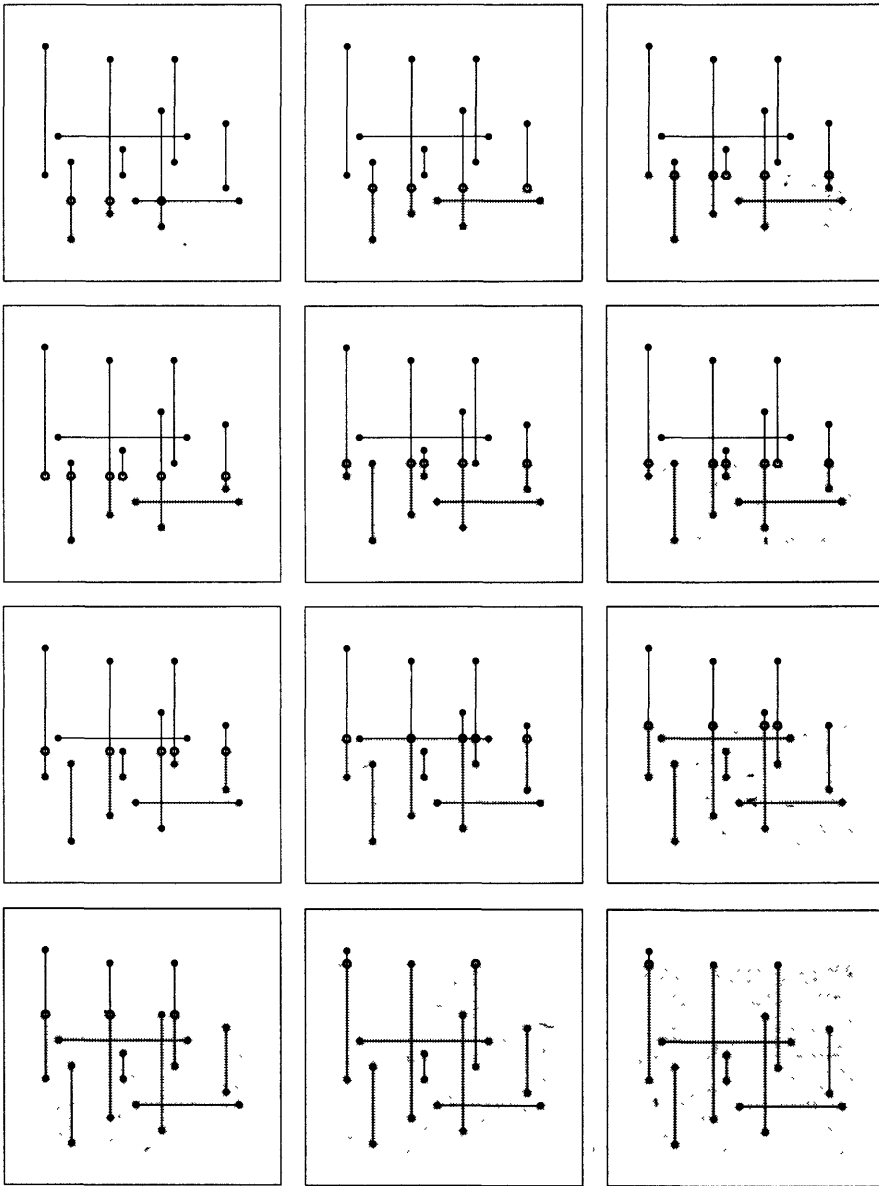


그림 27.3 교차선들을 찾기 위한 주사: 완료 과정.

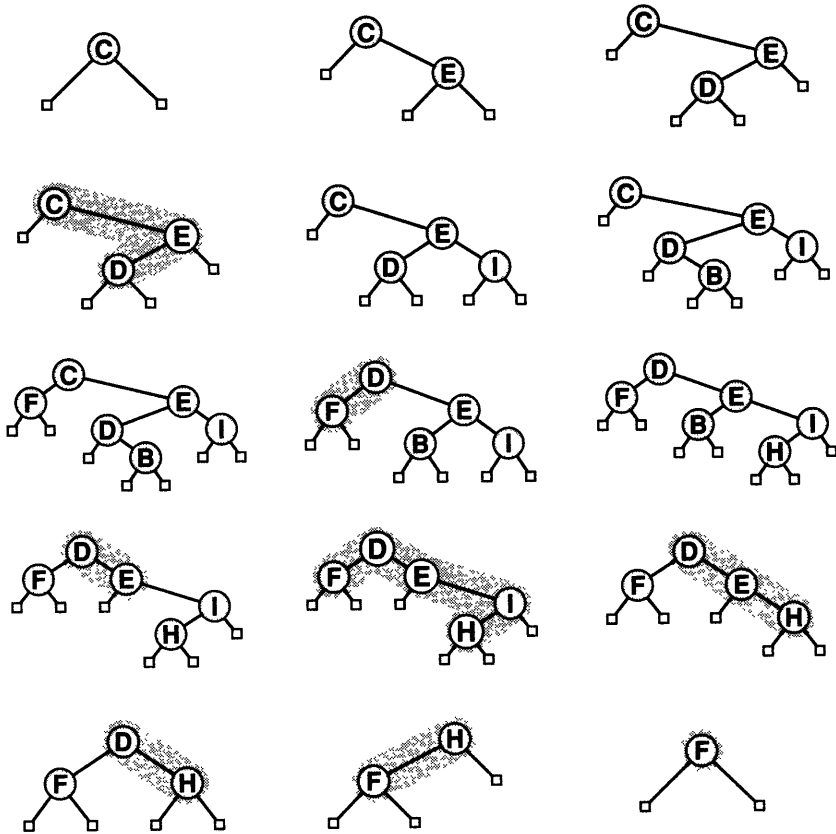


그림 27.4 주사시의 데이터 구조. x -트리의 형성.

구현(Implementation)

구현의 첫 번째 스텝은 선분들의 끝점들을 그들의 y 좌표 값으로 정렬하는 것이다. 그러나 이진 트리들은 수평 주사선에 대해 수직선들의 상태를 유지하는데 사용되므로, 초기의 y 좌표 정렬시 사용되는 것이 좋다! 특별히, 14장의 이진 트리 사전(dictionary) 클래스에 있는 두 이진 트리 $Xtree$ 와 $Ytree$ 를 사용할 것이다. y 트리는 한 번에 하나씩 순서에 따라 처리될 선분들의 끝점들을 가진다; x 트리는 현재의 수평 주사선을 교차하는 선분들을 가진다.

아래 프로그램은 먼저 표준 입력(standard input)으로부터의 선분들을 규정하는, 4 숫자로 구성된, 그룹들을 읽어 들이고 수직선들의 y 좌표들과 수평선들의 y 좌표를 삽입함으로써 y 트리를 만든다. 그리고 주사(scan)는 $Ytree$ 를 중위 운행(inorder traversal)하는 것이다:

```

Dict Xtree(Nmax), Ytree(Nmax);
struct line lines[Nmax];
int count = 0;
int intersections()
{
    int x1, y1, x2, y2, N;
    for(N = 1; cin >> x1 >> y1 >> x2 >> y2; N++)
    {
        lines[N].p1.x = x1; lines[N].p1.y = y1;
        lines[N].p2.x = x2; lines[N].p2.y = y2;
        Ytree.insert(y1, N);
        if (y2 != y1) Ytree.insert(y2, N);
    }
    Ytree.traverse();
    return count;
}

```

우리의 선분들의 집합 예제에서, 그림 27.5에 있는 트리가 형성된다. 알고리즘이 필요로 하는 “y에 따른 정렬”은 y값의 오름차순으로 노드 각각에 대해 visit 프로시저를 호출하는(14장을 보라) traverse 함수에 의해 영향을 받는다.(x좌표 값들에 대한 또 다른 이진 트리를 사용하여) 교차 선들을 찾는 작업들 모두는 visit 프로시저에서 행해진다.

이들 작업들은 아래에 규정되어 있다. 각 노드가 “방문되는” 시점에서 프로그램 코드를 합치면 알고리즘의 설명이 더 간단해진다:

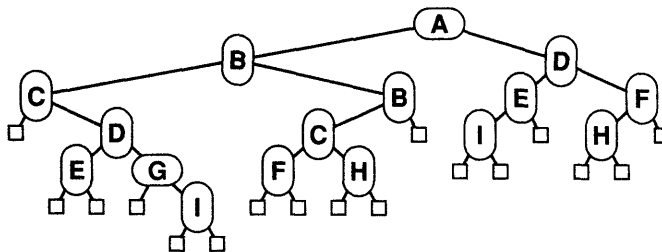


그림 27.5 y-트리를 이용한 주사를 위한 정렬

```

void Dict::Visit(itemType v, infoType info)
{
    int t, x1, x2, y1, y2;
    x1 = lines[info].p1.x; y1 = lines[info].p1.y;
    x2 = lines[info].p2.x; y2 = lines[info].p2.y;
    if (x2 < x1) { t = x2; x2 = x1; x1 = t; }
    if (y2 < y1) { t = y2; y2 = y1; y1 = t; }
    if (v == y1)
        Xtree.insert(x1, info);
    if (v == y2)
    {
        Xtree.remove(x1, info);
        count += Xtree.range(x1, x2);
    }
}

```

먼저, 해당 선의 끝점의 좌표들이 노드의 info필드(field)에 의해 인덱스 되어지는 lines 배열로부터 끄집어내어 진다. 그리고 나서 이 노드가 선분의 상부 끝점 또는 하부 끝점에 해당되는지를 알기 위해 이 노드의 key필드를 끝점의 좌표들과 비교한다. 만약 이 노드가 하부 끝점이면 x 트리에 삽입되고, 상부 끝점이면 x 트리에서 삭제되고 구간 검색이 수행된다. 위 구현은 이 설명과 약간 다르다. 왜냐하면 수평선들이 실제로는 x 트리에 삽입되고, 바로 삭제되며, 한 점 구간에 대한 구간 검색이 수직선들에 대해 수행되기 때문이다. 이렇게 함으로써 이 프로그램 코드는 “교차”하는 것으로 간주되는, 수직선들끼리 겹치는(overlapping), 경우를 올바르게 처리할 수 있게 된다.

x 와 y 좌표들 상에서 동작하는 재귀적 프로시저들이 혼합적으로 적용되는 이런 방식은 기하학적 알고리즘들에 있어서 아주 중요하다. 이에 대한 또 한 가지 예는 전 장의 2D트리 알고리즘이다. 다음 장에서 또 한 가지 예를 볼 수가 있다.

성질 27.1 N 개의 수평-수직선들 간의 모든 교차는 $N \log N + I$ 에 비례한 시간 내에 찾아질 수 있다. 여기서 I 는 교차의 갯수이다.

트리 조작 연산들은 평균적으로 $\log N$ 에 비례한 시간이 걸린다.(만약 균형 트리(balanced tree)가 사용된다면, 최악의 경우 $\log N$ 이 보장될 수 있다) 그러나 구간 검색에 걸리는 시간 또한 총 교차 수에 비례한다. 일반적으로, 교차의 갯수는 매우 많을 수 있다. 예로서, 만약

$N/2$ 개의 수평선들과 $N/2$ 개의 수직선들이 그물망 형태로 배열된다면, 교차의 갯수는 N^2 에 비례한다. □

구간 검색의 경우처럼, 만약 교차의 갯수가 많다는 것을 미리 알면, 한 주먹구구식 접근 방식이 이용되어야 한다. 응용 분야들은 아주 많은 선분들이 있음에도 교차하는 선분들은 아주 일부분인, “건초 더미에서 바늘을 찾는” 것과 같은, 상황들을 가지고 있다.

일반적 선분 교차(General Line Intersection)

선분들이 임의의 기울기를 가지면 상황은 그림 27.6에 예시된 것처럼 더욱 복잡해진다. 첫째로, 다양한 선분 방향(line orientation)들로 인하여 특정 선분들의 교차 여부의 시험이 필요하다—단순한 구간 구간 시험으로는 불가능하다. 둘째로, 이진 트리에 대한 선분들간의 순서 관계가 전보다 더욱 복잡하다. 왜냐하면 순서는 현재 고려 중인 y 의 구간에 좌우되기 때문이다. 셋째로, 어떤 교차들이 발생하면 새로운 “관심거리가 되는”(이전에 우리가 선분들의 끝점들에서 얻은 y 값들의 집합과는 다를 수 있는) y 값들이 추가된다.

이러한 문제들은 위에서 주어진 것과 동일한 기본 구조를 갖는 한 알고리즘으로 처리할 수 있음이 밝혀졌다. 설명을 간단히 하기 위해, N 개의 선분의 집합에 교차하는 한 선분 쌍이 존재하는지의 여부를 찾는 알고리즘을 살펴보고, 이 알고리즘을 어떻게 확장하면 모든 교차 선분 쌍들을 찾을 수 있게되는지에 대해 논의하기로 한다.

전과 마찬가지로, 먼저 평면을 어떤 끝점도 나타낼 수 없는 띠(strip)들로 나누기 위해 y 값에 따라 정렬한다. 그리고 나서 이 정렬된 점들의 리스트를 가지고 작업을 수행하여, 만약 한 선분의 하부 끝점과 만나면 그 선분을 이진 검색 트리에 넣고, 만약 상부 끝점과 만나면 그 선분을 트리에서 삭제한다. 이진 검색 트리는 두 인접한 y 값들 사이에 있는 수평 “띠”에서 나타나는 선분들의 순서를 제공한다. 예로서, 그림 27.6에 있는 D의 하부 끝점과 B의 상부 끝점간의 띠에서는, 선분들은 F B D H G의 순으로 나타나야 한다. 현재 고려 중인 수평 띠 내에는 어떤 교차도 없다고 가정한다: 우리의 목표는 이 트리 구조를 유지하고 첫 번째 교차 선분 쌍을 찾기 위해 이를 이용하는 것이다.

트리를 형성하기 위해서, 단순히 선분 끝점들의 x 좌표들을 키로서 이용할 수는 없다.(이렇게 할 경우, 위 예제를 예로 들면, B와 D가 잘못된 순서로 트리에 넣어 질 수 있다) 그 대신, 우리는 보다 일반적인 순서 관계를 이용하기로 한다: 선분 x 는, 만약 x 의 양 끝점들이 y

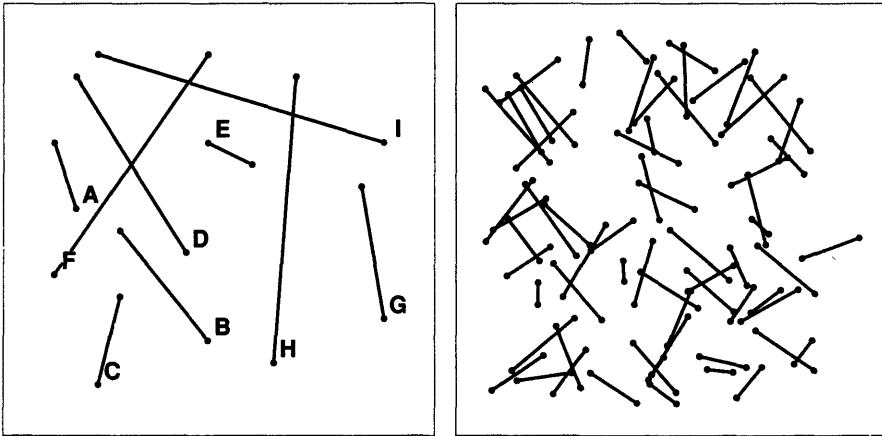


그림 27.6 두 개의 일반적 선분 교차 문제들

와 비교해 아주 오른쪽 멀리에 있거나 만약 y 가 x 와 비교해 x 의 아주 왼쪽 멀리에 있다면, 선분 y 의 오른쪽에 있는 것으로 정의된다. 따라서, 위 다이어그램에서, B는 A의 오른쪽에 있고 B는 C의 오른쪽에 있다.(왜냐하면 C는 B의 왼쪽에 있으므로) 만약 x 가 y 의 왼쪽에도 또한 오른쪽에도 있지 않으면, 이 둘은 교차해야만 한다. 이 일반화된 “선분 비교” 연산은 24장의 ccw 프로시저를 이용하여 구현 가능하다. 비교가 필요할 때마다 이 함수를 사용하는 것을 제외하면, 표준 이진 검색(바람직하다면, 구현 트리들에 대한) 트리 프로시저들이 사용될 수 있다. 그림 27.7은 우리의 예제에 대해 선분 C와 만나는 시간과 선분 D와 만나는 시간 사이의 트리 조작을 보여준다. 트리 조작 동안 행해지는 각 “비교”는 실제로는 선분-교차 시험이다: 만약 이진 검색 트리 프로시저가 오른쪽으로 가야 할지 왼쪽으로 가야 할지를 결정할 수 없다면, 이 때의 시험되는 두 선분은 교차한다.

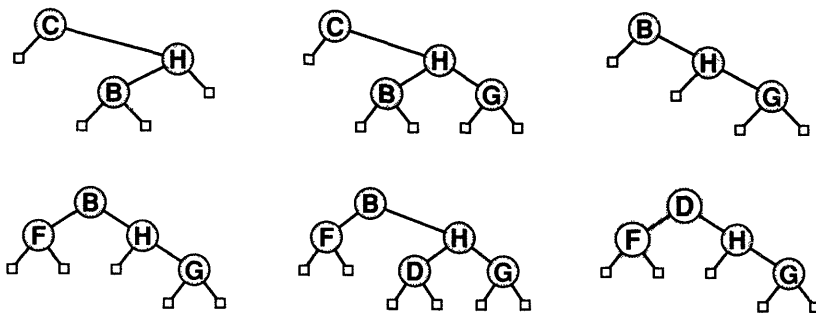


그림 27.7 일반적 문제에 대한 데이터 구조(x -트리)

그러나 이것이 전부는 아니다. 왜냐하면 이 일반화된 비교 연산은 추이적(transitive)이 아니기 때문이다. 위의 예제에서, F는 B의 왼쪽(왜냐하면 B는 F의 오른쪽이므로) 그리고 B는 D의 왼쪽에 있으나, F는 D의 왼쪽에 있지 않다. 이를 주목해 봄은 중요하다. 왜냐하면 이진 트리 삭제 프로시저는 비교 연산이 추이적이라고 가정하기 때문이다: B가 트리에서 삭제되면, 그림 27.7에서 보여지는 것과 같은 트리가 F와 D에 대한 어떤 명백한 비교도 없이 형성된다. 우리의 교차-시험 알고리즘이 올바르게 동작하려면, 트리 구조가 변경될 때마다 비교가 올바른지를 명백히 시험해야 한다. 특히, 노드 x 의 왼쪽 링크가 노드 y 를 가리키게 만들 때마다, x 에 해당하는 선분이, 앞의 정의에 따라, y 에 해당하는 선분의 왼쪽에 있는지를 명백히 시험한다. 물론 오른쪽 링크의 경우도 이와 유사하게 시험한다. 물론, 이 비교를 하면 결국, 우리 예제에서 행해진 것과 같이, 교차를 찾을 수가 있다.

요약하자면, N 개의 선분들의 집합에서 한 선분 쌍의 교차를 시험하기 위해, 위 프로그램을 사용했다. 그러나 range에 대한 호출을 제거하고 앞에서 설명한 일반화된 비교를 사용하기 위해 이진 트리 루틴들을 확장했다. 만약 교차가 없다면, 널(null)-트리를 가지고 시작하여 어떤 비교 불가능한 선분들 없이 널-트리로 끝난다. 만약 교차가 있으면, 교차하는 두 선분은 주사 과정 동안 어느 시점에선가 서로 비교되어지고 서로 교차함이 찾아질 것이다.

그러나 일단 한 선분 쌍의 교차가 찾아지면, 다른 교차 선분 쌍들을 계속해서 찾을 수는 없다. 왜냐하면 교차하는 두 선분들은 교차 바로 직후 바로 뒤에 그 순서가 서로 바뀌어야 하기 때문이다. 이 문제를 처리하는 한 가지 방법은 y 값에 따른 정렬에 대해 이진 트리 대신 우선 순위 큐를 사용하는 것일 수도 있다: 선분들을 그들의 끝점들의 y 좌표 값들에 따라 우선 순위 큐에 넣어지고 이 우선 순위 큐에서 최소 y 좌표를 끄집어내어 삽입 및 삭제가 수행된다. 교차가 찾아지면, 교차점을 각 선분의 하부 끝점으로 사용하여 각 선분에 대한 새로운 엔트리들이 우선 순위 큐에 추가된다.

모든 교차들은 찾는 또 한 가지 방법은, 만약 교차가 아주 많이 발생하지 않으면 적절한 방법으로서, 한 선분 쌍의 교차가 찾아질 때마다 교차 선분 중 하나를 제거하는 것이다. 그러면 주사가 완료된 후, 모든 교차 쌍들은 이렇게 제거된 선분들 중의 하나를 가질 것이다. 모든 교차들을 세기 위해서는 주먹구구식 방법이 사용될 수 있다.

성질 27.2 N 개의 선분들 간의 모든 교차들은 $(N + I)\log N$ 에 비례한 시간 내에 찾아질 수 있다. 여기서 I 는 교차의 갯수이다.

이 성질은 위 설명에서 바로 증명된다. \square

위 프로시저의 한 흥미로운 특징은 일반화된 비교를 사용함으로써 보다 일반적인 기하학적 형태들의 집단간에 일어나는 교차들을 시험할 수 있게 된다는 것이다. 예를 들어, 만약 선분들이 수직-수평선인 두 사각형을 “만약 사각형 x 의 오른쪽 선분이 사각형 y 의 왼쪽 선분의 왼쪽에 있으면 x 는 y 의 왼쪽에 있다”라는 사소한 규칙에 따라 비교하는 한 프로시저를 구현한다면, 이같은 사각형들의 집합에서의 교차를 위 방법을 사용하여 시험할 수 있다. 원의 경우에는, 중심점의 x 좌표들을 이용하여 순서대로 배열하여 교차 여부를 시험할 수 있다.(예로서, 원들의 중심점들간의 거리와 반지름들의 합을 비교한다) 만약 이 비교 프로시저를 위 방법에 적용하면, 원들의 집합들에 대해 교차 여부를 시험하는 알고리즘이 만들어진다. 이러한 경우들에서 모든 교차를 찾는 문제는 -비록 바로 전 절에서 언급된 주먹구구식 방법이 거의 교차가 없는 경우에는 잘 동작하지만- 아주 많이 복잡해진다. 많은 응용 분야들에 적합한 또 한 가지 방법은 복잡한 객체들을 선들의 집합으로 간주하고 선분-교차 프로시저를 이용하는 것이다.

연습문제

1. 두 삼각형이 교차하는지를 어떻게 알 수 있나? 사각형의 경우는? $N > 4$ 인 정 n -각형의 경우는?
2. 수평-수직 선분 교차 알고리즘에서, 교차가 전혀 없는 선분들의 집합에서 최악의 경우 얼마나 많은 선분 쌍들이 교차 여부가 시험되나?
3. 수평-수직 선분 교차 프로시저가 임의의 기울기를 갖는 선분들의 집합에 대해 사용될 때 어떤 일이 일어나나?
4. 0과 1000사이의 무작위 정수 좌표 값들을 이용하여 그리고 수직선과 수평선의 구분을 위해 한 무작위 비트를 사용하여, 만들어진 N 개의 무작위적 수평-수직 선분들 중에서 교차 선분 쌍들의 갯수를 찾는 프로그램을 작성하라.
5. 한 주어진 다각형이 단순 다각형(즉, 자신을 교차하지 않는 다각형)인지를 시험하는 방법은?
6. 한 다각형이 다른 다각형 내에 완전하게 포함되는지를 시험하는 방법은?
7. 두 선분간의 최소 간격이 이 선분들 중의 최대 길이 보다 크다는 사실을 알 때 일반적인 선분-교차 문제를 어떻게 해결할 지를 설명하라.
8. 선분 교차 알고리즘이 그림 27.6을 90도 회전시켜 만들어진 선분들의 교차들을 찾을 때 존재하는 이진 트리 구조들은?
9. 원(circle)들과 본문에서 설명한 Manhattan 사각형들에 대한 비교 프로시저들은 추이적인가?
10. N 개의 무작위 선분들에서 교차 선분 쌍들의 갯수를 찾는 프로그램을 작성하라. 단, 각 선분은 0과 1000사이의 무작위 정수 좌표 값들을 갖고 만들어졌다.

빈 면

가장 인접한 점 문제들

평면상의 점들을 포함하는 기하학적 문제들은 보통 점들간의 거리들을 은연중에 또는 명시적으로 다룬다. 예로서 많은 응용 분야들에서 일어나는 한 자연적인 문제는 한 새로이 주어진 점과 가장 인접한 점을 주어진 점들의 집합 내에서 찾는 가장 인접한 이웃(nearest-neighbor) 문제이다. 이 문제에서는 한 주어진 점에서 집합 내의 각 점까지의 거리들을 계산해야 할 듯 싶다. 그러나 훨씬 더 좋은 해결책들이 있을 수 있다. 이 장에서는 몇 개의 다른 거리 문제들과 프로토타입(prototype) 알고리즘 및 이러한 종류의 다양한 문제들을 푸는데 효과적으로 이용될 수 있는 Voronoi 다이어그램이라 불리는 한 근본적인 기하학적 구조를 살펴보기로 한다. 한 단순한 문제에 대한 프로토타입 구현을 주의 깊게 살펴봄으로써 가장 인접한 점 문제들을 풀기 위한 일반적인 방법을 설명하기로 한다.

이 장에서 다루어질 일부 문제들은 26장의 구간 검색 문제들과 유사하며 26장에서 다룬 격자 및 2D 트리 방법들은 가장 인접한 이웃 문제나 다른 문제들을 푸는데 적합하다. 그러나 이 방법들의 근본적인 단점은 이들이 점 집합의 무작위성에 근거하고 있다는 점이다: 이들은 최악의 경우시의 성능이 나쁘다. 이 장에서의 우리의 목표는 어떤 입력 형태이든지 관계없이 많은 문제들에 대해 좋은 성능을 보장하는 한 일반적인 방식을 조사하는 것이다. 일부 방법들은 너무 복잡하여 이들의 구현을 완전하게 살펴보기가 어렵다. 또한 이 방법들은 상당한 오버헤드를 수반하므로 이들보다 간단한 방법들을 이용하면 점 집합이 크지 않거나 점들이 잘 분포되어 있을 경우에는 더 좋을 수가 있다. 그러나 최악의 경우에도 좋은 성능을 보여주는 방법들을 살펴봄으로써(비록 더 간단한 방법들이 특정 경우들에 있어서 더 좋을 수 있을 지라도) 알아야 할 근본적인 점 집합들의 성질들을 살펴 볼 수가 있다.

우리가 살펴 볼 일반적인 방식은 두 좌표 방향으로 처리를 함께 하는 중복 재귀 프로시저들의 또 다른 사용 예이다. 이미 배운 이런 유형의 두 가지(kD 트리와 선분 교차) 방법들은 이진 검색 트리에 근거한다; 여기서 사용할 방법은 병합정렬(mergesort)에 근거한 “결합-정복(combine and conquer)” 방법이다.

가장 인접한 쌍 문제(Closest-Pair Problem)

가장 인접한 쌍(closest-pair) 문제는 점들의 집합에서 가장 인접해 있는 두 점을 찾는 것이다. 이 문제는 가장 인접한 이웃 문제와 연관이 있다; 비록 이 문제는 널리 응용되지는 않지만, 다른 문제들에 대해서도 적합한 일반적인 재귀적 구조를 갖는 알고리즘에 의해 해결될 수 있으므로, 가장 인접한 쌍 문제의 프로토타입으로 이용될 수 있다.

가장 인접한 거리를 찾기 위해서는 점들의 쌍들 간의 거리를 모두 조사할 필요가 있다: 이 말은 N 개의 점들이 있을 때 N^2 에 비례하는 수행 시간이 걸림을 의미할 수도 있다. 그러나 정렬을 사용하여 최악의 경우 약 $N \log N$ 개의 거리만을 조사하고(평균적으로는 훨씬 더 작다) 그래서 $N \log N$ 에 비례하는 최악의 수행 시간이 걸리 수 있게 만들 수 있다.(평균적으로는 훨씬 더 좋다) 이 절에서는, 이 알고리즘을 자세히 살펴보기로 한다.

사용할 알고리즘은 단순한 “분할-정복” 전략에 근거한 것이다. 방법은 점들을 한 좌표 상에서(예로서, x 좌표 상에서) 정렬시키고 이 정렬된 점들을 반으로 나눈다. 전체 점들의 집합에서 가장 인접한 쌍은 집합의 한 분할부 내에 있거나 각 분할부에 한 점씩 위치할 것이다. 물론 관심거리가 되는 경우는 가장 인접한 쌍의 두 점이 다른 분할부에 하나씩 있을 때이다: 각 분할부에 있는 가장 인접한 쌍은 재귀적 호출을 이용하여 쉽게 찾을 수 있다. 그러나 어떻게 가장 효율적으로 찾을 수 있는가?

우리가 찾고자 하는 것은 점 집합 내의 가장 인접한 쌍이므로, 분할선(dividing line)에서 거리 \min 내에 있는 점들을 조사할 필요가 있다. 여기서 \min 은 두 분할부에서 찾아진 가장 인접한 쌍들 간의 거리들 중에서의 최소 값이다. 그러나 최악의 경우 이것만으로는 해결이 안된다. 왜냐하면 분할선에 아주 가까운 많은 점의 쌍들이 있기 때문이다; 예를 들어, 각 분할부 내의 점들은 분할선 바로 옆에 줄지어(lined up) 있을 수 있다.

이러한 경우들의 처리를 위해, 점들을 y 좌표에 따라 정렬할 필요가 있다. 이렇게 하면 거리 계산의 횟수를 다음과 같이 하여 줄일 수 있다: y 의 오름차순으로 점들을 처리하는데, 각 점이 분할선에서 \min 내에 있는 모든 점을 포함하는 수직 띠 내에 있는지를 조사한다. 각각

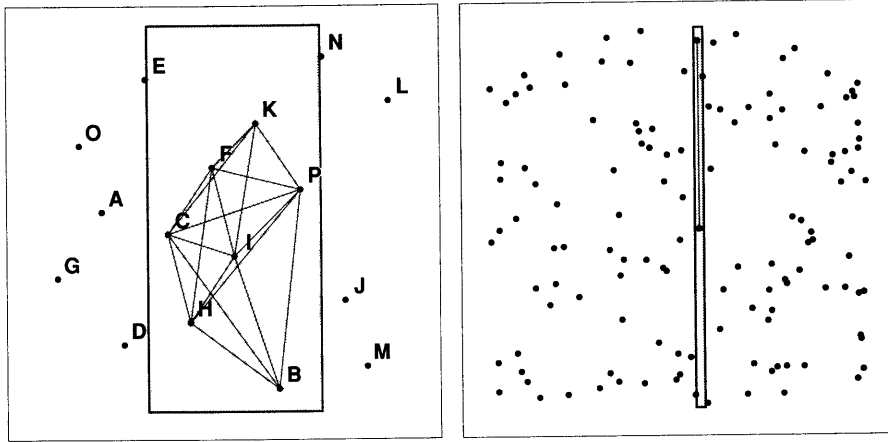


그림 28.1 가장 인접한 쌍을 찾기 위한 분할-정복 방식

의 이런 점과 그 점의 y 좌표 값보다 작은 y 좌표 값을 갖는 그러나 기껏해야 \min 만큼 작은 수직띠내의 또 한점과의 거리를 계산한다. 각 분할부에 있는 점들에 대한 모든 쌍들간의 거리가 적어도 \min 만큼 된다는 사실은 단지 몇개의 점들만이 조사되어질 확률이 큼을 의미한다.

그림 28.1의 좌측에 있는 점들의 갯수가 작은 집합에서는 F의 바로 우측에 수직 분할선을 (마음속으로) 그으면, 이 선의 좌측과 우측 모두에 8개의 점들이 있게 된다. 좌측에서의 가장 인접한 쌍은 AC(또는 AO)이고, 우측에서의 가장 인접한 쌍은 JM이다. 만약 점들이 y 좌표 값에 따라 정렬되어 있으면, 분할선에 의해 분할된 가장 인접한 쌍은 전체 점 집합에서 가장 인접한 쌍들인 HI, CI, FK와 마지막으로 EK를 조사함으로써 찾아진다. 우측의 점들이 많은 집합들에서는, 분할선에 걸쳐 있는 가장 인접한 쌍을 포함하는 띠(band)가 더 좁아진다.

비록 이 알고리즘이 간단히 설명되었지만, 이를 효율적으로 구현하려면 약간의 주의가 필요하다: 예를 들어, 우리의 재귀적 서브루틴 내에서 y 값에 따라서 점들을 정렬하면 비용이 많이 든다. 우리는 수행 시간이 반복(recurrence) 공식 $C_N = 2C_{N/2} + N$ 으로 설명되는 알고리즘 들을 보아왔다. 여기서 C_N 은 $N \log N$ 에 비례한다. 만약 y 값에 따라서 완전 정렬을 시키면, 반복공식은 $C_N = 2C_{N/2} + N \log N$ 이 될 수도 있다. 이 경우 C_N 은 $N \log^2 N$ 이 된다.(6장을 보라) 이를 피하려면, y 값에 따른 정렬을 해서는 안된다.

이 문제에 대한 해결책은 단순하지만 난해하다. 12장에서 배운 병합 정렬(mergesort) 방법은 정렬될 원소들을 위에서 점들을 분할한 것과 똑같은 방법으로 분할하는 것에 그 기반을 두고 있다. 문제가 분할됐으므로 해결할 문제는 두 개가 됐으나 이들을 풀기 위해서는 동일

한 일반적 방법을 이용한다. 따라서 이 두 문제를 동시에 해결하는 것이 더 낫다! 특히, 우리는 y 값에 따라 정렬도 시키고 가장 인접한 쌍도 찾는 한 재귀적 루틴을 작성할 것이다. 이 루틴은 점 집합을 반으로 분할하고, 이 두 분할된 부분을 y 값에 따라 정렬하기 위해 자신을 재귀적으로 호출하며 각 분할된 부분에서 가장 인접한 쌍을 찾은 다음, 정렬을 완료시키기 위해 병합을 하고, 가장 인접한 쌍을 찾는 계산을 종료하기 위해 위에서 언급한 절차를 적용한다. 이런 식으로, 정렬을 위한 데이터 이동과 가장 인접한 쌍의 계산을 위해 필요한 데이터 이동을 한꺼번에 함으로서, y 값에 따라 정렬을 다시 한 번 더하는 것을 피할 수 있다.

y 값 정렬에서, 반으로 분할함은 어떤 식으로든지 행해질 수 있다. 그러나 가장 인접한 쌍의 계산에서는, 한 분할된 반쪽에 있는 점들 모두가 다른 반쪽에 있는 점들 보다 작은 x 좌표 값을 가져야만 한다. 이것은 분할에 앞서 x 값에 따라 정렬을 함으로써 쉽게 해결된다. 사실 상, x 값에 따른 정렬시에는 y 값에 따른 정렬을 위한 루틴을 사용하면 된다! 일단 이러한 일반적인 구현 계획이 이해되면, 구현 프로그램을 이해하는 것은 어렵지가 않다.

앞서 언급한 것처럼, 구현 시에 12장에 있는 재귀적인 정렬 및 병합 프로시저들을 사용한다. 첫 번째 스텝은 키들 대신 점들을 갖도록 리스트 구조들을 수정하고, merge 프로시저를 어떻게 비교할지를 결정 하기 위한 한 전역변수 pass를 조사하도록 수정해야 한다. 만약 pass가 1이면, 두 점의 x 좌표 값을 비교해야 하고, pass가 0이면 두 점의 y 좌표 값을 비교해야 한다. 이의 구현은 간단하다:

```
int comp(struct node *t)
{ return (pass == 1) ? t->p.x : t->p.y; }
struct node *merge(struct node *a, struct node *b)
{
    struct node *c;
    c = z;
    do
        if (comp(a) < comp(b))
            { c->next = a; c = a; a = a->next; }
        else
            { c->next = b; c = b; b = b->next; }
    while (c != z);
    c = z->next; z->next = z;
    return c;
}
```

모든 리스트들의 끝에 나타나는 모조 노드 z 는 인위적으로 아주 높은 x, y 좌표 값들을 갖는 한 표지(sentinel)점을 포함하도록 초기화된다.

거리 계산을 위해, 인수로서 주어진 두 점들간의 거리가 전역변수 min 보다 작은지를 조사하는 또 하나의 간단한 프로시저를 사용한다. 만약 작다면, 그 거리 값을 min 에다 넣고 전역변수 $cp1$ 과 $cp2$ 에다 이 두 점을 저장한다.

```
check(struct point p1, struct point p2)
{
    float dist;
    if ((p1.y != z->p.y) && (p2.y != z->p.y))
    {
        dist = sqrt((p1.x - p2.x)*(p1.x - p2.x)
                    + (p1.y - p2.y)*(p1.y - p2.y));
        if (dist < min)
            { min = dist; cp1 = p1; cp2 = p2; }
    }
}
```

이같이, 전역변수 min 은 항상 $cp1$ 과 $cp2$ 상의 거리-이제까지 찾아진 것 중에서 가장 인접한 점들의 거리-를 가진다.

다음 스텝은 12장의 재귀적 프로시저 `sort`를 수정하여 `pass`가 2일 때 가장 인접한 점 계산을 하도록 만드는 것이다. 이렇게 하여 만들어진 `sort`는 다음과 같다:

```
struct node *sort(struct node *c, int N)
{
    int i;
    struct node *a, *b;
    float middle;
    struct point p1, p2, p3, p4;
    if (c->next == z) return c;
    a = c;
    for (i = 2; i <= N/2; i++) c = c->next;
    b = c->next; c->next = z;
    if (pass == 2) middle = b->p.x;
    c = merge(sort(a, N/2), sort(b, N-(N/2)));
}
```



```

    if (pass == 2)
    {
        p1 = z->p; p2 = z->p; p3 = z->p; p4 = z->p;
        for (a = c; a != z; a = a->next)
            if (fabs(a->p.x - middle) < min)
            {
                check(a->p, p1);
                check(a->p, p2);
                check(a->p, p3);
                check(a->p, p4);
                p1 = p2; p2 = p3; p3 = p4; p4 = a->p;
            }
    }
    return c;
}

```

만약 pass가 1이면, 12장의 재귀적 병합 정렬 루틴과 똑같다: 여기서는 x 좌표 값에 따라 정렬된 점들을 갖는 연결-리스트가 리턴된다.(왜냐하면 merge 함수가 위에서 설명된 것처럼 첫 번째 pass상에서 x 좌표 값들을 비교하도록 수정됐기 때문이다) 이 구현의 기묘함은 pass가 2일 때에 있다. 이 경우, y 값에 따른 정렬 뿐만 아니라(왜냐하면 merge 함수가 두 번째 pass상에서 y 좌표 값들을 비교하도록 수정 됐기 때문에) 가장 인접한 점 계산도 수행 완료한다. 재귀적 호출에 앞서 점들은 x 값에 따라 정렬된다: 이 정렬된 순서는 점들의 집합을 분할하고 분할선의 x 좌표를 찾을 때 이용된다. 재귀적 호출 후에는, 점들이 y 값에 따라 정렬되고 각 분할부에 있는 모든 점들의 쌍의 거리가 min보다 크을 알게 된다. y 값에 따른 정렬은 분할선 부근에 있는 점들을 스캔하기 위해 이용된다; min값은 시험될 점들의 갯수를 제한하는데 이용된다. 분할선에서 min거리 내에 있는 각 점은 이전에 분할선에서 min거리 내에 있었던 4개의 점 각각과 비교 조사(check)된다. 이 조사는 한 분할부상에서 min보다 더 가까운 점들의 쌍을 찾는 것을 보장한다.

왜(두 점, 세 점, 다섯 점이 아닌) 이전의 네 점을 조사하는가? 이것은 여러분이 증명하기를 원하는 재미있는 기하학적 사실이다: 같은 분할부에 있는 점들은 적어도 min 만큼 떨어져 있음을 알기 때문에 물로 네 개의 점보다 많은 점들을 조사해도 상관이 없다. 그리고 네 점이면 충분함을 보이는 것은 어렵지 않다.

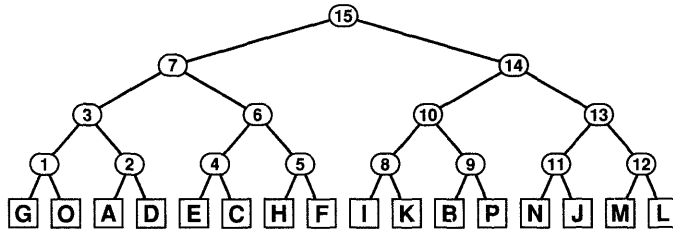


그림 28.2 가장 인접한 쌍 계산을 위한 재귀적 호출 트리

아래 프로그램 코드는 가장 인접한 쌍의 계산을 위해 sort를 두 번 호출한다. 먼저 x 에 대한 정렬($\text{pass}=1$ 일 때)을 하고 그리고 y 에 대해 정렬한 후 가장 인접한 쌍을 찾는다. ($\text{pass}=2$ 일 때):

```
z = new node; z->p.x = max;
          z->p.y = max; z->next = z;
h = new node; h->next = readlist();
min = max;
pass = 1; h->next = sort(h->next, N);
pass = 2; h->next = sort(h->next, N);
```

위와 같은 함수의 호출 후, 가장 인접한 쌍은 두 전역변수 cp1 과 cp2 에서 찾아지는데, 이 cp1 과 cp2 는 “최소 값을 찾는” 프로시저인 check에 의해 관리된다.

그림 28.2는 점들의 갯수가 작은 집합에 대한 이 알고리즘의 동작을 나타내는 재귀 호출 트리를 보여준다. 이 트리의(타원으로 표시된) 내부 노드는 좌측과 우측의 서브 트리들에 있는 점들을 분할하는 수직선을 나타낸다. 이 노드들은 알고리즘에서 수직선들이 시도된 순서에 따라 번호가 붙여진다. 이 번호부여 방식은, 분할 선을 포함한 계산이 프로그램에서의 재귀적 호출들 후에 행해지므로 트리의 후위순행(postorder traversal)에 해당한다. 그리고 이 번호부여 방식은 재귀적 병합-정렬시(12장을 보라) 병합의 수행 순서를 보는 또 다른 방법이다.

이같이, 먼저 G와 O사이의 선에 대해 시도되고, GO는 이제까지의 쌍 중에서 가장 인접한 쌍으로서 보존된다. 다음으로 A와 D에 대해 시도되는데 A와 D는 너무 멀리 있어서 min을 변경시킬 수 없다. 그리고나서 O와 A간의 선이 시도되고 GD, GA 및 OA 모두가 계속하여 더 인접한 쌍들이 된다. 이 예제에서는, 분할선에서 마지막으로 조사된 쌍인 FK까지는 더 이상 인접한 쌍들이 발견되지 않는다.

독자는 우리가 위에서 설명한 순수한 분할-정복 알고리즘을 구현하지 않았음을 알 수 있을 것이다. 실제로 우리는 두 분할부에서 각기 가장 인접한 쌍을 계산한 후 그 중 더 인접한 쌍을 선택하지 않았다. 그 대신, 재귀적 계산 동안 단지 \min 을 위한 한 전역변수를 사용하여 두 개의 가장 인접한 쌍 중에서 더 인접한 쌍을 택한다. 더 인접한 쌍이 찾아질 때마다, 재귀적 계산에서 어디에 있는지에 관계없이, 현 분할선 주변의 더 좁은 수직 띠를 고려할 수가 있다.

그림 28.3은 이 과정을 자세히 보여준다. 이 다이어그램들에서, x 좌표는 이 과정의 x 편향을 강조하고 병합-정렬(12장의 보라)과의 유사성을 보이기 위해 확대되었다. 우리는 가장 좌측의 네 점 G O A D에 대해 y -정렬을 함으로써 시작하여, GO를 정렬하고 AD를 정렬한 다음 병합한다. 이 병합 후 y -정렬이 끝난다. 그리고 분할선을 가로지르는 가장 인접한 쌍 AO를 찾는다. 그 다음에는 E C H F 등등에 대해 같은 방법으로 진행한다.

성질 28.1 N 개 점의 집합에서 가장 인접한 쌍은 $O(N \log N)$ 스텝 내에 찾아질 수 있다.

본질적으로, 이 계산은 두 병합-정렬(하나는 x 좌표, 또 하나는 y 좌표에 대한 정렬)을 위한 시간과 분할선을 따라 조사하는 시간을 합한 시간 내에 행해진다. 이 시간 비용은 또한 반복 공식 $Tn = 2T_{N/2} + N$ 에 따른다. \square

여기서 사용된 가장 인접한 쌍 문제에 대한 일반적 방식은 다른 기하학적 문제들을 푸는 데도 사용될 수 있다. 예를 들어, 관심거리가 되는 또 하나의 문제는 모든 가장 인접한 이웃들(all-nearest-neighbors)의 문제이다: 각 점에 대해, 우리는 그 점과 가장 가까이 있는 점을 찾기를 원한다. 이 문제는 위의 프로그램에다가 각 점에 대해 분할선을 따라 자기 쪽에 있는 가장 인접한 점 보다 더 인접한 점이 다른 쪽에 있는지를 찾는 처리를 추가하면 해결될 수 있다. 이 계산을 위해 y -정렬이 유용하게 사용될 수 있다.

Voronoi 다이어그램(Voronoi Diagram)

한 점 집합에서 주어진 한 점에(다른 점들보다) 더 가까운 모든 점들의 집합은 주어진 점에 대한 Voronoi 다각형이라 불리는 흥미로운 기하학적 구조이다. 한 점 집합에 대한 모든 Voronoi 다각형들의 합집합을 Voronoi 다이어그램이라고 부른다. 이 다이어그램이 가장 인접한 점 계산에서의 최종 결론이다: 우리는 점들간의 거리를 포함한 대부분의 문제들이 Voronoi

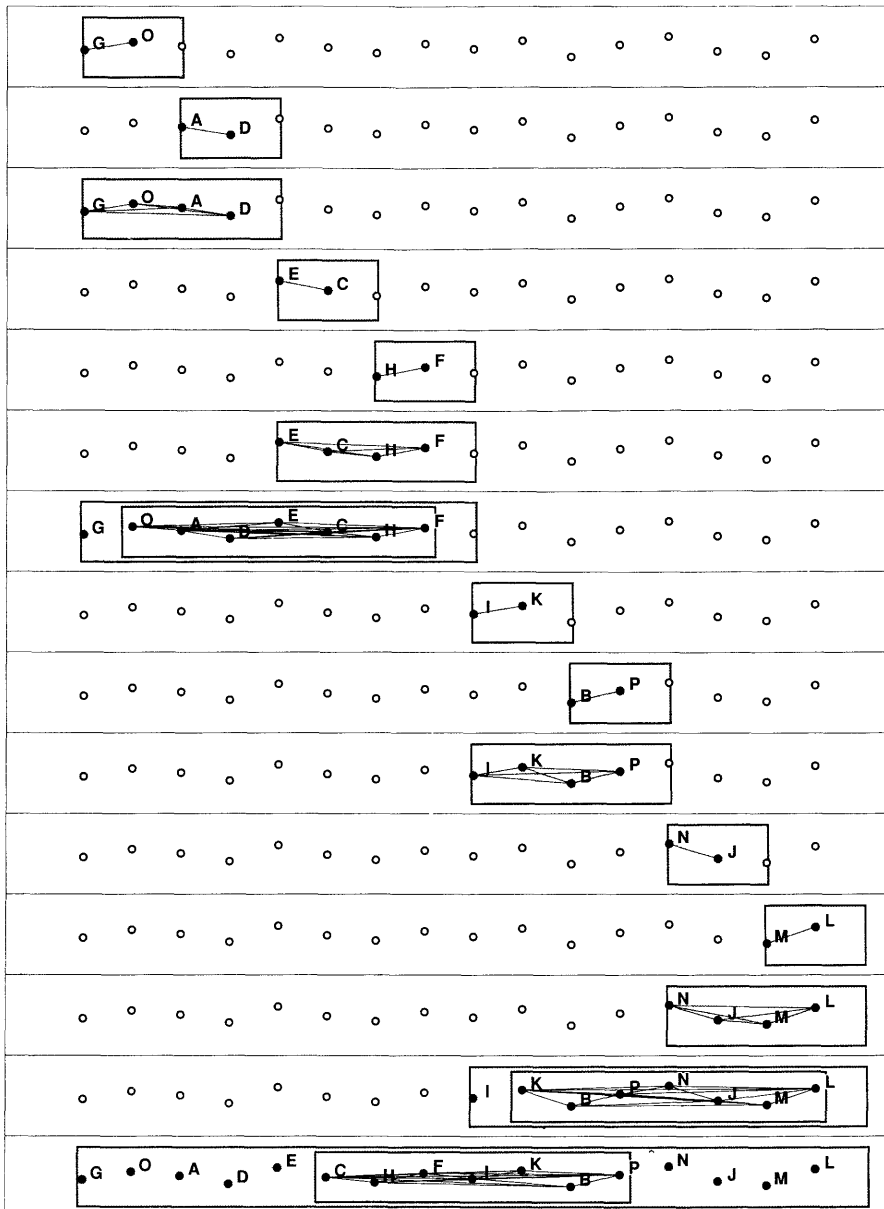


그림 28.3 가장 인접한 쌍 계산 (x 좌표는 확대되어 있음)

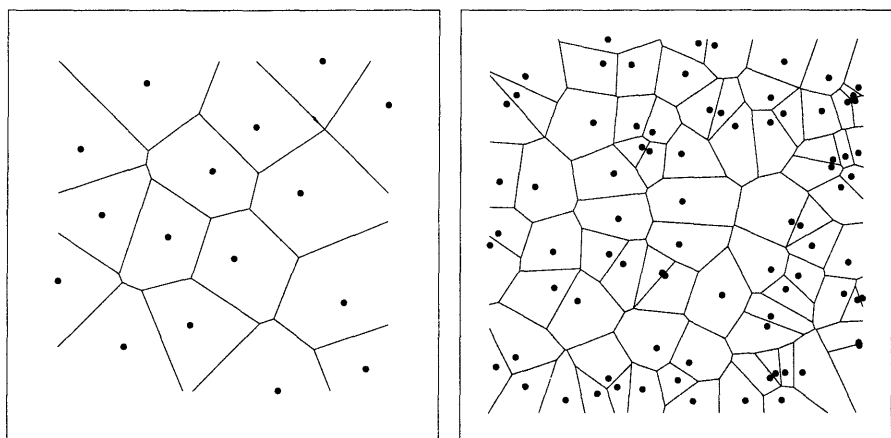


그림 28.4 Voronoi 다이어그램

다이어그램에 근거한 자연스럽고 재미있는 해들을 가짐을 알 수 있을 것이다. 우리의 샘플 점 집합들에 대한 Voronoi 다이어그램들이 그림 28.4에 있다.

한 점에 대한 Voronoi 다각형은 그 점을 가장 인접한 점들과 연결해 주는 조각들의 수직 이등분선들로 이루어진다. 이 다각형에 대한 실제 정의를 다시 써 보면 다음과 같다: Voronoi 다각형은 점 집합에서 한 주어진 점과(다른 어느 점들을 보다) 더 인접한 모든 점들의 집합의 주변(테두리)으로써 정의된다. Voronoi 다각형상의 각 선분은 주어진 한 점을 자신에 “가장 인접한” 점들의 하나와 분리시킨다.

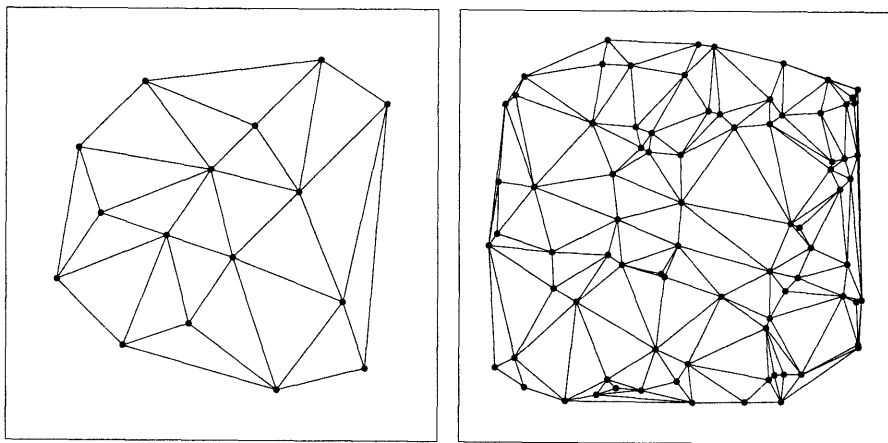


그림 28.5 Delaunay 삼각형

그림 28.5에 있는 Voronoi 다이어그램의 쌍대(dual)는 이러한 관계를 명확히 해준다: 이 쌍대에서는, 각 점과 그 점에 가장 인접한 모든 점들 각각의 사이에 한 선분이 그려진다. 이렇게 그려진 도형을 또한 Delaunay 삼각형이라고 불려진다. 점 x 와 y 는, 만약 이들의 Voronoi 다각형들이 한 공통 선분을 가진다면, Voronoi의 쌍대(dual)에서 연결되어 있다.

Voronoi 다이어그램과 Delaunay 삼각형은 가장 인접한 점 문제들에 대해 효율적인 알고리즘들을 가져다주는 많은 성질들을 가지고 있다. 이 알고리즘들을 효율적으로 만들어 주는 성질은 Voronoi 다이어그램과 그 쌍대에 있는 선분들의 수는 한 작은 상수의 N 배에 비례한다는 것이다. 예로서, 가장 인접한 쌍을 연결하는 선분은 반드시 쌍대에 있어야 하므로, 앞 절의 문제는 쌍대를 계산하고 이 쌍대에 있는 선들 중에서 최소 길이를 갖는 선분을 찾음으로써 해결될 수 있다. 이와 유사하게, 각 점을 가장 인접한 이웃과 연결해 주는 선분은 반드시 쌍대에 있어야 하므로, 모든 가장 인접한 이웃 문제는 바로 쌍대를 찾는 문제가 된다. 점 집합의 볼록 외곽(convex hull)은 쌍대의 일부이므로, Voronoi의 쌍대는 또 하나의 볼록 외곽 알고리즘이다. 우선 Voronoi 쌍대를 먼저 찾음으로서 효율적으로 해결되는 문제에 대한 또 하나의 예는 31장에서 살펴보기로 한다.

Voronoi 다이어그램을 정의하는 성질은 이 다이어그램이 가장 인접한 이웃 문제를 푸는데 이용될 수 있음을 알려준다: 한 점 집합에서 주어진 한 점과 가장 인접한 이웃을 찾기 위해서는, 그 주어진 점이 어느 Voronoi 다각형 내에 있는지만 찾으면 된다. 이들에 대한 검색을 보다 효율적으로 할 수 있게 하기 위해, Voronoi 다각형들을 2D트리 같은 구조로 구성하는 것이 가능하다.

Voronoi 다이어그램은 위에 있는 가장 인접한 점 알고리즘과 똑같은 일반적 구조를 갖는 알고리즘을 사용하여 계산될 수 있다. 먼저 점들을 그들의 x 좌표 값에 따라 정렬하고, 이 정렬된 점들을 반으로 분할되어 각 분할부에 있는 점 집합에 대한 Voronoi 다이어그램을 찾기 위해 재귀적 호출을 두 번 수행한다. 이와 동시에, 점들은 y 좌표 값에 따라 정렬된다; 마지막으로, 분할부에 대한 두 개의 Voronoi 다이어그램들이 병합된다. 전과 마찬가지로, (pass가 2일 때) 행해지는 이 병합은 점들이 재귀적 호출에 앞서 x 값에 따라 정렬되고, y 값에 따라 정렬되고, 그리고 재귀적 호출 후에 두 분할부에 대한 Voronoi 다이어그램들이 만들어진다는 사실을 이용할 수 있다. 그럼에도 불구하고 병합은 아주 복잡한 작업이며 이의 완전한 구현은 이 책의 구간을 벗어난다.

Voronoi 다이어그램은 가장 인접한 점 문제들에 대한 자연스런 구조이다. 한 문제의 특성을 Voronoi 다이어그램이나 그 쌍대의 관점에서 이해하는 것은 분명히 가치 있는 것이다. 그

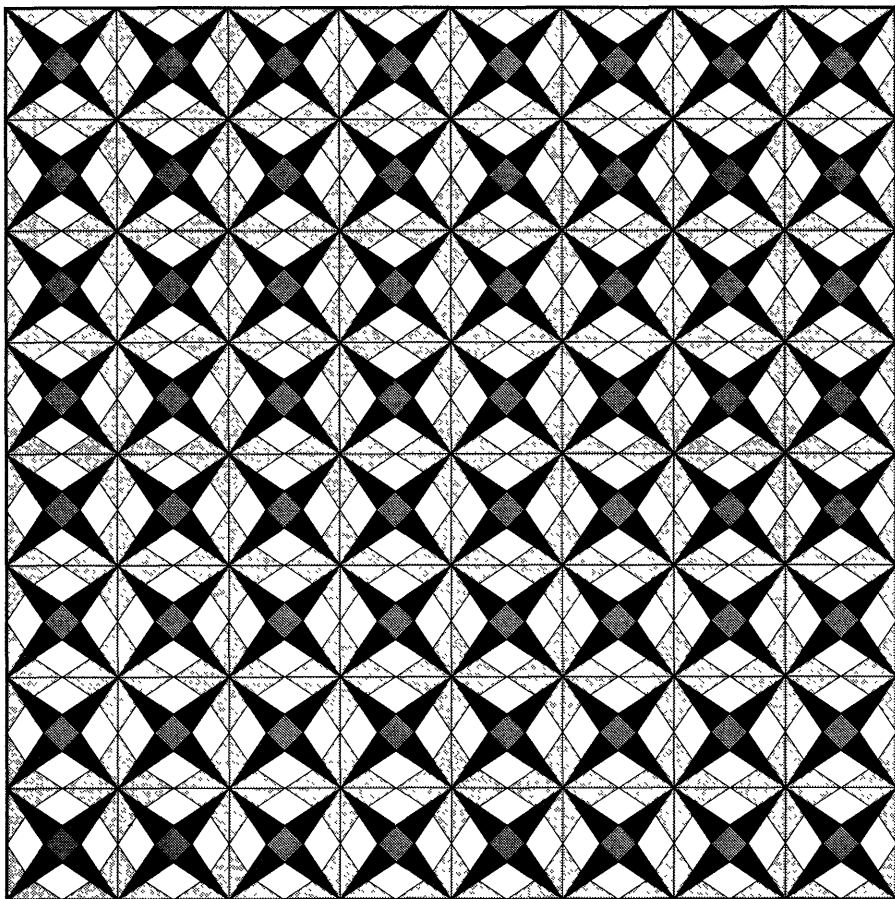
러나 많은 실제 문제들의 경우, 이 장에서 설명한 일반적인 개요에 근거하여 직접 구현해도 좋을 수 있다. 이 개요는 Voronoi 다이어그램을 계산하기에 충분하므로 Voronoi 다이어그램에 근거한 알고리즘들에 대해서도 충분하며 또한 우리가 가장 인접한 쌍 문제에서 본 것 같은 더 간단하고 효율적인 프로그램의 작성을 가능케 해줄 수도 있다.

연습문제

1. 가장 인접한 이웃 문제를 풀기 위한 프로그램을, 처음에는 격자(grid)법을 사용하여 작성하고 그 다음에는 2D 트리들을 사용하여 작성하라.
2. 가장 인접한 쌍 프로시저가 동일 수평선상에 등 간격(equal space)으로 있는 점들의 집합에 대해 사용될 때 어떤 일들이 일어나는지 설명하라.
3. 가장 인접한 쌍 프로시저가 동일 수직선 상에 등 간격으로 있는 점들의 집합에 대해 사용될 때 어떤 일들이 일어나는지 설명하라.
4. $2N$ 개의 점들의 집합에서 반은 양수 x 좌표 값을 갖고 나머지 반은 음수 x 좌표 값을 가질 때 각 반쪽에 한 점씩 있는 가장 인접한 쌍을 찾는 알고리즘을 작성하라.
5. 본문의 프로그램이 본문에 있는 우리의 예제에서 A 를 제거한 점 집합 상에서 수행될 때 $cp1$ 과 $cp2$ 에 할당되는 연속적인 점들의 쌍들을 기록하라.
6. 한 많은 무작위 점들의 집합에 대해, 본문에서 주어진 구현 프로그램의 성능과 순수 재귀적 구현 프로그램의 성능을 비교함으로써, \min 이 전역변수가 될때의 효율성을 시험하라.
7. 선분들의 집합에서 가장 인접한 쌍을 찾는 알고리즘을 작성하라.
8. 본문에 있는 샘플 점 집합의 점 $A B C D E F$ 에 대해 Voronoi 다이어그램과 그 쌍대를 그려라.
9. Voronoi 다이어그램을 계산하는(N^2 에 비례한 시간을 필요로할 수도 있는) 한 “주먹구구식” 방법을 찾아라.
10. 점들의 집합에 대한 볼록 외곽을 찾기 위해, 본문에 있는 가장 인접한 쌍의 구현과 동일한 재귀적 구조를 사용하는 프로그램을 작성하라.

빈 면

그래프 알고리즘



빈 면

29 장

기본 그래프 알고리즘들

아주 많은 문제들이 객체들과 객체들의 연결들의 관점에서 자연스럽게 공식화되어진다. 예로서, 미동부의 항공로 지도가 주어졌을 때, 우리는 다음과 같은 질문들에 관심이 있을 수 있다: “Providence에서 Princeton까지 가는 가장 빠른 길은 어디인가?” 또는, 시간보다는 비용에 더 관심이 있어서 가장 값싸게 갈 수 있는 길을 찾을 수도 있다. 이러한 질문들에 답하기 위해서는, 객체들(도시들)간의 상호 연결(항공 경로)에 대한 정보가 필요하다.

전기 회로들은 객체들간의 상호 연결이 중요한 역할을 하는 또 하나의 명백한 예이다. 트랜지스터, 레지스터, 그리고 축전기들 같은 회로 구성 원소들은 아주 복잡하게 연결되어 있다. 이같은 회로들은 “모두 연결되어 있는가?” 같은 단순한 질문들에서부터 “만약 이 회로가 만들어지면, 제대로 동작할 것인가?” 같은 복잡한 질문들에 이르기까지 답할 수 있도록 컴퓨터 내에서 표현되고 처리 될 수 있어야 한다. 여기서, 첫 번째 질문에 대한 답은 단지 상호 연결(선들)의 성질에 따라 결정된다. 반면에 두 번째 질문에 대한 답은 선들과 이 선들에 의해 연결된 객체들 모두에 대한 상세한 정보를 필요로 한다.

세 번째 예는 “작업 스케줄링(job scheduling)”으로서, 이 경우 객체들은(말하자면, 한 생산 공정에서의) 수행 되어 할 TASK(task)들이며 이들간의 연결은 어떤 작업이 먼저 수행되어져야 하는가를 알려준다. 이같은 경우, 우리는 “언제 각 TASK가 수행 되어야 하는가?” 같은 질문들에 대한 답에 관심을 가질 수가 있다.

그래프는 이와 같은 상황들을 정확히 모델링 해주는 수학적 객체이다. 이 장에서는, 그래프의 몇몇 기본적인 성질들을 살펴보고, 다음 장들에서는 위에서 제기된 유형의 질문들에 답하기 위한 다양한 알고리즘들을 배우기로 한다.

실제로, 우리는 전 장들에서 이미 그래프들을 보아 왔다. 링크된 데이터 구조는 실제로 그래프의 표현이며, 그래프 처리를 위해 앞으로 배울 일부 알고리즘들은 트리와 다른 구조의 처리를 위한(이미 배운) 알고리즘들과 유사하다. 예로서, 19장과 20장의 유한-상태 기계(finite-state machine)들은 그래프 구조로서 표현 될 수 있다.

그래프 이론은 조합 수학(combinatorial mathematics) 분야의 한 큰 분야로서 수 백년 동안 활발하게 연구되어 왔고 그래프들의 많은 중요하고 유용한 성질들이 증명되어 왔다. 그러나, 아직도 많은 어려운 문제들이 해결되어야만 한다. 여기서는 단지 그래프들에 대해 알려진 것들의 아주 일부만을, 그러나 근본적인 알고리즘들을 이해하기에는 충분한 내용을 다루기로 한다.

이제 까지 공부해 온 많은 분야들 처럼, 그래프도 최근에야 알고리즘적인 관점에서 연구되기 시작했다. 비록 일부 근본적인 알고리즘들은 아주 오래 됐지만, 많은 관심사가 되는 문제들은 최근 십 년 이내에 알려진 것이다. 심지어는 아주 단순한 그래프 알고리즘들도 흥미로운 컴퓨터 프로그램 거리가 될 수 있다. 여기서, 우리가 다룰 복잡한 알고리즘들은 알려진 알고리즘들 중에서 가장 세련되고 흥미로운 것 중에 속한다.

용어풀이(Glossary)

아주 많은 용어들이 그래프들과 연관되어 있다. 이 용어들의 대부분은 아주 단순한 정의를 가지고 있으며, 비록 이들 중 일부는 나중에 사용될 것이지만, 편의상 여기서 함께 설명하기로 한다.

그래프는 정점(vertex)들과 선분(edge)들의 집합이다. 정점은 이름과 다른 성질들을 가질 수 있는 단순한 객체이다; 선분은 두 정점을 연결해 준다. 그래프는 정점은 점으로 선분은 정점들을 연결해 주는 선으로 표현하여 그려질 수 있다. 그러나, 그래프는 그 표현 방법과는 무관하게 정의됨을 명심해야 한다. 예로서, 그림 29.1의 두 그림은 동일한 그래프를 나타내고 있다. 이 그래프는 정점 A B C D E F G H I J K L M들의 집합과 두 정점들 AG AB AC LM JM JL JK ED FD HI FE AF GE를 연결해 주는 선분들의 집합으로써 정의된다.

앞의 항공로 예제 같은 일부 응용분야들의 경우에는, 그림 29.1에서처럼 정점들을 재배열하는 것은 납득이 되지 않는다. 앞의 전기회로 같은 분야들의 경우에는, 어느 특정 기하학적 위치와는 상관없이 선분들과 정점들에 대해서만 관심을 갖고 집중하는 것이 좋다. 그리고 19장과 20장에서 다룬 유한-상태 기계들 같은 또 다른 분야들의 경우에는, 노드들에 대한 어느 특정한 기하학적 위치도 내재되어 있지 않다. 그래프 알고리즘들과 기하학적 알고리즘들간의

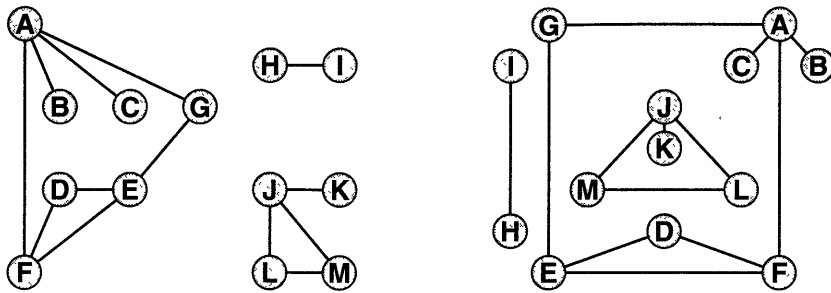


그림 29.1 동일한 그래프에 대한 두 가지 표현

관계는 31장에서 더 자세히 논의된다. 당분간은, 선분들과 노드들의 단순한 집합들을 처리하는 “순수한(pure)” 그래프 알고리즘들만 집중적으로 다루기로 한다.

그래프의 정점 x 에서 y 까지의 경로는 연속적인 정점들이 선분들에 의해 연결되어 있는 정점들의 리스트이다. 예로서, BAFEG 그림 29.1에 있는 B에서 G로 가는 경로이다. 한 그래프에 각 노드에서 모든 다른 노드로 가는 경로가 있으면, 그 그래프는 “연결되어 있다.(connected)”고 한다. 만약 정점들이 물리적 객체들이고 선분들이 이 객체들을 연결하는 선들이라면, 연결된 그래프는 그 그래프 내의 한 정점을 들어올리면 모두 한꺼번에 들어올려지게 된다. 연결되어 있지 않은 그래프는 연결된-요소(connected component)들로 구성된다; 예로서, 그림 29.1의 그래프는 세 개의 연결된-요소들을 가진다. 단순 경로(simple path)는 그 경로 상의 어떤 정점도 반복되지 않는 경로이다.(예로서, BAFEGAC는 단순 경로가 아니다) 사이클(cycle)은 첫 정점과 마지막 정점이같은 점만 제외하면 단순 경로가 되는 경로이다:(즉 한 점에서 다시 그 점으로 되돌아오는 경로) 경로 AFEGA는 사이클이다.

사이클이 없는 그래프를 트리(tree)라 한다.(4장을 보라) 연결 안된 트리들의 그룹(group)을 포리스트(forest)라 한다. 한 그래프의 스패닝 트리(spanning tree)는 모든 정점을 갖지만 선분은 트리를 구성할 수 있을 만큼만 갖는 서브그래프이다. 예로서, 선분 AB AC AF FD DE EG는 그림 29.1의 그래프의 가장 큰 요소에 대한 한 스패닝 트리이다. 그림 29.2는 더 큰 그래프와 그 그래프에 대한 한 스패닝 트리를 보여준다.

만약 트리에 선분 하나를 더하면, 사이클이 형성되어야 한다.(왜냐하면 이 선분이 연결하는 두 정점들간에는 이미 한 경로가 설정되어 있기 때문이다) 또한, 4장에서 본 바와 마찬가지로, V 개의 정점들에 대한 트리는 $V - 1$ 개의 선분을 갖는다. 만약 V 개의 정점을 갖는 그래프가 $V - 1$ 개 보다 작은 선분을 가지면, 이 그래프는 연결된 그래프가 될 수 없다. 만

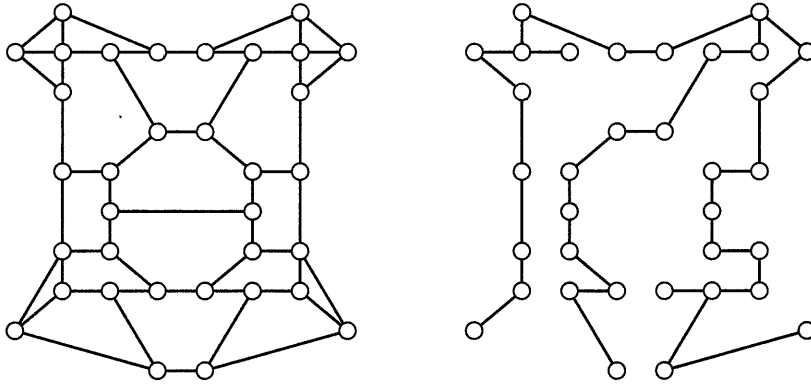


그림 29.2 한 큰 그래프와 그 그래프에 대한 한 스패닝 트리

약 $V - 1$ 개 보다 많은 선분이 있으면, 이 그래프에는 사이클이 있게 된다.(그러나 정확히 $V - 1$ 개의 선분이 있더라도, 이 그래프가 반드시 트리가 되는 것은 아니다)

한 그래프에 있는 정점들의 수는 V 로 표시하고, 선분들의 수는 E 로 표시한다. E 의 개수는 0 과 $\frac{1}{2}V(V - 1)$ 사이의 한 값임에 주목하라. 모든 선분이 존재하는 그래프를 완전 그래프(complete graph)라 한다. 상대적으로 선분의 수가 아주 작은(예로서 $V \log V$ 보다 작은) 그래프는 스파아스(sparse)하다고 한다. 거의 모든 선분들이 있으면 덴스(dense)하다고 한다.

이 두 개(정점과 선분)의 파라미터에 대해 그래프 기하학이 근본적으로 의존함은 이제까지 공부해 온 많은 알고리즘들보다 그래프 알고리즘들의 비교 연구를 좀더 복잡하게 만든다. 예로서, 한 알고리즘의 수행 시에 약 V^2 스텝이 걸릴 수 있는 반면에, 또 다른 알고리즘을 동일한 문제에 대해 수행할 경우 $(E + V)\log E$ 스텝이 걸리 수가 있다. 두 번째 알고리즘은 스파아스 그래프들의 경우 좋을 수 있으나, 덴스 그래프들의 경우에는 첫 번째 알고리즘이 바람직하다.

지금까지 정의된 그래프들은 무방향 그래프(undirected graph)들이라고 불리는 가장 단순한 형태의 그래프이다. 우리는 또한 노드(정점)들과 선분들에 많은 정보가 결합되어진 보다 복잡한 형태의 그래프들을 다룰 것이다. 가중치(weighted) 그래프에서는, 정수(가중치)들이 각 선분들에 할당되어 거리나 비용들을 나타낸다. 방향성(directed) 그래프에서는, 선분들이 “한 방향”으로 간다: 선분은 x 에서 y 로 갈 수는 있으나 y 에서 x 로 갈 수는 없다. 방향성 가중치 그래프는 종종 네트워크(network)라 불린다. 앞으로 보게 되겠지만, 가중치 그래프와 방향성 그래프들이 갖는 이러한 추가된 정보는 단순한 무방향 그래프들에 비해 이 그래프들의 처리를 좀더 어렵게 만든다.

표현(Representation)

컴퓨터 프로그램으로 그래프들을 처리하려면, 이 그래프들이 컴퓨터에서 어떻게 표현되어야 할 지를 먼저 결정할 필요가 있다. 여기서는 통상적으로 이용되는 두 가지 표현 방식을 살펴보기로 한다. 이 둘 중에서 어느 것을 선택하는가는(비록 수행될 연산들의 성질 또한 중요한 역할을 하지만) 주로 그래프가 텐스한가 스파아스한가에 따라 좌우된다.

그래프를 표현의 첫 번째 단계는 정점 명들을 1에서 V 사이의 정수에 매핑(mapping)하는 것이다. 이렇게 하는 주된 이유는 각 정점에 해당하는 정보를 배열 인덱싱을 이용하여 빨리 액세스 할 수 있도록 하기 위함이다. 어떤 규격화된 검색 알고리즘도 이를 위해 사용될 수 있다: 예로서, 어느 한 정점 명에 해당하는 정수를 찾기 위해 검색될 수 있는 해시 테이블(hash table)이나 이진 트리를 유지함으로써, 정점 명들을 1에서 V 사이의 정수에 매핑시킬 수 있다. 이미 이런 기법들을 공부해왔으므로, 정점 명들을 1에서 V 사이의 정수로 변환시켜 주는 함수 `index`와 정수들을 정점 명들로 변환시켜 주는 함수 `name`이 있다고 가정한다. 알고리즘들을 쉽게 이해 할 수 있도록 하기 위해, 정점 명은 알파벳 상의 한 문자로 표현하는데, i 번째 알파벳은 정수 i 에 해당한다. 따라서 `name`과 `index` 함수들은 우리의 예제들의 경우에 있어서 구현이 쉬우나, 이들을 사용함으로써 알고리즘들이 14-17장에 있는 기법들을 사용하여, 한 글자가 아닌 실제의 정점 명들을 포함하는 그래프들을 처리할 수 있도록 확장하기가 쉬워진다.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	0	0	1	1	0	0	0	0	0	0
B	1	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	1	0	0	0	0	0	0
F	1	0	0	1	1	1	0	0	0	0	0	0	0
G	1	0	0	0	1	0	1	0	0	0	0	0	0
H	0	0	0	0	0	0	0	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	1	1	0	0
L	0	0	0	0	0	0	0	0	0	1	0	1	1
M	0	0	0	0	0	0	0	0	0	1	0	1	1

그림 29.3 인접-행렬 표현

그래프들에 대한 가장 단순한 표현 방식은 소위 인접-행렬(adjacency-matrix)이라 부르는 방식이다. 이 방식에서는 만약 정점 x 에서 y 로 가는 선분이 있으면 1, 아니면 0으로 채워지는 V 행- V 열의 배열을 유지한다. 그림 29.1에 대한 인접-행렬이 그림 29.3에 있다.

각 선분은 실제로는 두 비트에 의해 표현됨에 주목하라: x 와 y 를 연결하는 한 선분은 $a[x][y]$ 와 $a[y][x]$ 두 곳에서 참(1)으로 표시된다. 이러한 대칭적인 행렬의 절반만 저장 이용함으로써 저장 공간을 절약할 수 있으나, 이러한 방식은 C++의 경우에는 이용하기가 다소 불편하며 알고리즘들도 행렬 전부를 이용하면 약간 더 간단해진다. 또한 각 정점에서 그 자신으로 가는 선분이 있다고 가정하는 것이 통상적으로 편리하다. 따라서, 1에서 V 까지의 값을 갖는 x 에 대해 $a[x][x]$ 의 값은 1이 된다.(일부 경우에 있어서는, 대각선상의 모든 원소들을 0으로 만드는 것이 더 편리할 수 있다: 우리는 필요하다면 언제든지 이 방식을 사용하기로 한다)

그래프는 노드들의 집합과 이들을 연결해 주는 선분들의 집합으로서 정의된다. 그래프를 입력으로 받아들이기 위해서는, 노드들의 집합과 선분들의 집합을 읽어 들이기 위한 포맷을 설정할 필요가 있다. 한 가지 가능한 방법은 입력 포맷으로서 인접-행렬 자체를 사용하는 것이다. 그러나, 앞으로 알게 되겠지만, 이 방법은 스파이스 그래프들의 경우에는 적합치 않다. 따라서 이 방법 대신 보다 직접적인 포맷을 이용하겠다: 먼저 정점 명들을 읽어 들인 다음, 이어서(선분들을 정의하는) 정점 쌍들을 읽어 들인다. 앞서 언급한 것처럼, 한 가지 쉬운 처리 방법은 정점 명들을 해시 테이블이나 이진 검색 트리에 넣고 각 정점 명에 인접-행렬 같은 정점-인덱스된 배열들을 액세스 할 때 사용하기 위해 정수 값을 할당하는 것이다. 읽어 들여진 i 번째 정점에 정수 값 i 가 할당 될 수 있다. 프로그램들의 단순성을 위해, 먼저 V (정점의 개수)와 E (선분의 개수)를 읽어들이고 나서 정점들과 선분들을 읽어 들인다. 이와는 다르게, 입력은 정점들과 선분들을 구별해 주는 구분 문자를 이용하여 배열될 수가 있고, 프로그램이 입력된 정점들과 선분들로부터 V 와 E 를 계산할 수도 있다.(우리 예제들의 경우에는, 정점 명들로서 알파벳의 첫 V 문자만을 사용하므로 V 와 E 의 입력이 더 간단해진다. 이렇게 할 경우, 알파벳의 첫 V 개 문자들 중의 두 문자씩으로 구성된 E 개의 문자 쌍을 만들어 사용하면 된다) 선분들이 나타나는 순서는 중요하지 않다. 왜냐하면 선분들이 어떤 순서로 배열되더라도 동일한 그래프를 나타내며, 결과적으로 아래 프로그램에 의해 계산된 것과 같은 동일한 인접-행렬이 만들어진다:

```

int V, E;
int a[maxV][maxV];
void adjmatrix()
{
    int j, x, y;
    cin >> V >> E;
    for (x = 1; x <= V; x++)
        for (y = 1; y <= V; y++) a[x][y] = 0;
    for (x = 1; x <= V; x++) a[x][x] = 1;
    for (j = 1; j <= V; j++)
    {
        cin >> v1 >> v2;
        x = index(v1); y = index(v2);
        a[x][y] = 1; a[y][x] = 1;
    }
}

```

변수 $v1$ 과 $v2$ 의 유형(type)과 `index` 함수의 프로그램 코드는 이 프로그램에서 생략되어 있다. 이들은 요구되는 그래프의 입력 표현 방식에 따라, 간단히 추가될 수 있다. 우리 예제들의 경우, $v1$ 과 $v2$ 는 `char`형의 변수 일수 있고 `index`는 `c-'A'+1`이거나 이와 유사한 값을 리턴 해주는 간단한 함수일 수 있다.

그래프의 표현 방식을 그래프에 적용될 기본 함수들로 구성된 인터페이스로 숨겨 주도록 C++클래스를 만들 수 있다. 이런 방식은 사전이나 우선 순위 큐 같이 널리 사용되는 구조들에서 널리 이용되어 왔으나, 그래프 알고리즘들의 경우에는 이용하지 않고자 한다. 왜냐하면, 전역변수들을 사용하여 구현된 프로그램들이 더 간결하고, 응용 분야에 따른 구현상의 득실(tradeoff)들이 좋은 클래스들의 구현에 필요한 듯 하기 때문이다. 그래프 알고리즘들에 대한 우리의 목적은 표현 방식들의 차이점들을 설명하기 위함이지 이들을 숨기기 위함이 아니다. 이를 배운 독자는 적절한 표현을 택할 수 있고, 이 택해진 표현을 특정 응용 분야에 결합할 수 있도록 해주는 C++의 데이터 추상화 도구들을 사용할 수 있다.

인접-행렬 표현은 처리될 그래프들이 덴스한 경우에만 사용하기 좋다: 행렬은 V^2 개의 저장 공간을 위한 비트들과 이들의 초기화를 위한 V^2 번의 스텝들이 요구된다. 만약 선분들의 수(행렬 내에 1인 비트들의 수)가 V^2 에 비례하면, 이 조건들은 받아들일 만한 것이다. 왜냐하면 선분들을 읽어 들이기 위해 어쨌든 약 V^2 번의 스텝들이 필요하기 때문이다. 그러나 만약 그

래프가 스पा아스하면, 행렬의 초기화를 위한 시간이 알고리즘 수행 시간의 지배적 요소 일수 있다. 이 또한 실행을 위해 V^2 보다 많은 스텝들을 필요로 하는 일부 알고리즘들의 경우에는 최선의 표현일 수가 있다.

이제 텐스하지 않은 그래프들에 대해 보다 적합한 표현을 살펴보자. 인접-구조(adjacency-structure) 표현의 경우에는, 각 정점에 연결된 모든 정점들이 그 정점에 대한 인접-리스트 상에 기록되어 있다. 이것은, 우리의 샘플 그래프에 대해 인접 구조를 만들어 주는 아래 프로그램에서 보여지는 바와 같이, 연결-리스트(linked list)들을 이용하여 쉽게 행해질 수 있다. 연결-리스트들은 평상시처럼 구성되는데, 리스트 맨 끝에 리스트 자신을 가리키는 인공 노드(artificial node) z 가 있다. 리스트들의 시작을 가리키는 인공 노드들은 정점-인덱스된 배열 adj 내에 있게 된다. x 와 y 를 연결하는 한 선분을 이 그래프 표현에 추가하기 위해, x 를 y 의 인접-리스트에 더하고 y 를 x 의 인접-리스트에 더한다:

```
struct node
{ int v; struct node *next; };
int V, E;
struct node *adj[maxV], *z;
void adjlist()
{
    int j, x, y; struct node *t;
    cin >> V >> E;
    z = new node; z->next = z;
    for (j = 1; j <= V; j++) adj[j] = z;
    for (j = 1; j <= E; j++)
    {
        cin >> v1 >> v2;
        x = index(v1); y = index(v2);
        t = new node;
        t->v = x; t->next = adj[y]; adj[y] = t;
        t = new node;
        t->v = y; t->next = adj[x]; adj[x] = t;
    }
}
```

인접-리스트 표현은 스पा아스 그래프들에게 더 좋다. 왜냐하면 필요한 저장 공간이, 인접-행렬 표현시 필요한 $O(V^2)$ 과는 대조적으로, $O(V + E)$ 이기 때문이다.

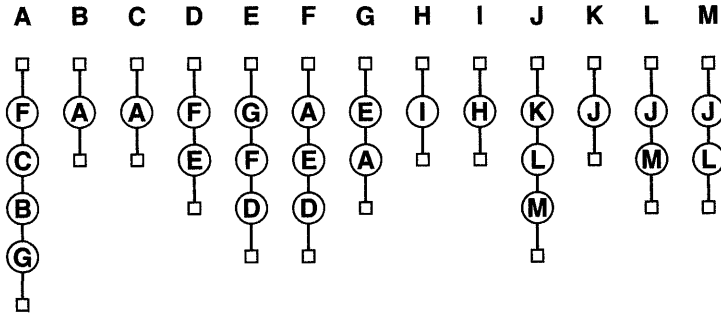


그림 29.4 한 인접 구조 표현

만약 선분들이 AG AB AC LM JM JL JK ED FD HI FE AF GE의 순으로 나타나면, 위 프로그램은 그림 29.4에 있는 것과 같은 인접-리스트 구조를 만들 것이다. 각 선분이 두 번씩 표현됨을 다시 한번 주목하라: x 와 y 를 연결하는 선분은 y 의 인접-리스트 상에 x 를 포함하는 한 노드로서 그리고 x 의 인접-리스트 상에 y 를 포함하는 한 노드로서 표현된다. 이들 모두를 포함함은 중요하다. 만약 그렇지 않으면 “어떤 노드들이 노드 x 와 직접 연결되어 있는가?” 같은 질문들이 효율적으로 답해질 수 없을 수가 있다.

이 표현의 경우, 입력시에 선분들이 나타나는 순서는 아주 중요하다: 이 순서는(사용된 리스트 삽입 방법과 함께) 정점들이 인접-리스트들 상에 나타나는 순서를 결정해 준다. 따라서, 동일한 그래프도 인접-리스트 구조에서는 많은 다른 형태들로서 표현될 수 있다. 단지 선분들의 순서만을 검사해서는 어떤 형태의 인접-리스트들이 만들어질지 예측하기가 어렵다. 왜냐하면 각 선분은 두 인접-리스트 상에 삽입되기 때문이다.

인접-리스트 상에서 선분이 나타나는 순서는 알고리즘에 의해 선분이 처리되는 순서에 영향을 준다. 다시 말해서, 인접-리스트 구조는 우리가 살펴보고 있는 여러 알고리즘들이 어떻게 그래프를 “보는가(see)”를 결정한다. 알고리즘은 선분들이 인접-리스트들 상에서 어떤 순서로 있는 간에 정답을 산출해야 하지만, 다른 입력 선분 순서에 대해 아주 다른 계산 순서들로서 그 답에 이를 수 있다. 그리고 두 개 이상의 “정답”이 있다면, 입력 순서에 따라(순서가 다르면) 다른 답이 나올 수가 있다.

인접-리스트 표현으로 해결할 수 없는 몇개의 간단한 연산들이 있다. 예를 들어, 한 정점 x 와 이에 연결된 모든 선분들의 삭제를 원할 수가 있다. 이를 위해, 인접-리스트에서 노드들을 삭제하는 것은 충분치 않다: 인접-리스트 상의 각 노드는, x 에 해당하는 노드의 삭제를 위해 검색 되어 하는 인접-리스트를 갖는, 다른 노드를 지정한다. 이 문제는 한 특정 선분에

해당하는 두 리스트 노드를 함께 링크시키고 인접-리스트들을 이중 연결시킴으로써 해결될 수 있다. 이렇게하면 한 선분이 제거 되어야 할 경우, 그 선분에 해당하는 두 리스트 노드들이 빨리 삭제 될 수 있다. 물론, 이 추가된 링크들은 처리하기가 성가시다. 따라서 이 링크들은 삭제 같은 연산 동작이 필요 하지 않는한 추가되지 말아야 한다.

이같은 고려 사항들은 또한, 할당된 레코드들로서 표현된 정점들과 정점들 대신 정점들에 대한 링크들을 갖고 있는 선분 리스트들을 가지고 그래프를 정확하게 모델링하는 데이터 구조 같은, "직접적인" 표현을 왜 사용하지 않는가 하는 불만을 갖게 만든다. 정점들을 직접 액세스하지 않으면 간단한 연산 동작들도 문제가 된다. 예로서, 이런 방식으로 표현된 한 그래프에 한 선분을 추가하려면, 해당 정점들을 찾기 위해 어떤 식으로든 그래프를 검색해야만 한다.

방향성 그래프들과 가중치 그래프들도 유사한 구조로서 표현된다. 방향성 그래프들의 경우, 각 선분이 한 번씩만 표현되는 것만 빼고는 모든 표현이 같다: x 에서 y 로 가는 한 선분은 인접-행렬에서는 $a[x][y]=1$ 로서 표현되고, 인접-리스트 구조에서는 x 의 인접-리스트 상에 y 가 나타나게 함으로써 표현된다. 따라서, 무방향 그래프는 한 선분에 의해 연결된 각 정점쌍 사이에 각기 다른 방향으로 가는 두 개의 방향성 선분이 있는 방향성 그래프로 간주할 수 있다. 가중치 그래프들의 경우, 인접-행렬 표현에서는 이진 값들 대신 가중치들을 행렬에 넣는 것만 제외하면 모든 표현이 같고(선분이 없는 경우에는 가중치가 없다): 인접-리스트 표현에서는 선분 가중치에 대한 한 필드를 인접 구조상에 추가하면 된다.

복잡한 알고리즘들에서는, 보다 복잡한 객체들을 모델링하거나 어떤 정보들을 저장하는 것을 가능케 하기 위해, 그래프의 정점이나 노드에 다른 정보를 결합시키는 것이 종종 필요하다. 각 정점에 결합된 이 추가 정보는 정점 번호에 의해 인덱스된 보조 배열을 사용하거나 인접 구조 표현에서 adj 를 레코드들의 배열로 만듦으로써 결합될 수 있다. 각 선분에 결합된 추가 정보는 인접-리스트 노드들 내에 놓여지거나,(또는 인접-행렬 표현에서는 레코드들의 배열 a 내에) 선분 번호에 의해 인덱스된 보조 배열들 내에 놓여질 수 있다.(선분들에게 번호를 부여함이 필요하다)

깊이-우선 검색(Depth-First Search)

이 장의 앞 부분에서, 그래프를 다룰 때 발생하는 여러 가지 질문들을 보았다. 그래프가 연결되어 있는가? 만약 연결되어 있지 않으면, 어떤 것들이 연결된-요소들인가? 그래프에 사

이클이 있는가? 이러한 많은 문제들은 깊이-우선 검색(depth-first search)이라고 불리는 기법을 이용하여 쉽게 해결될 수 있다. 깊이-우선 검색은 체계적으로 그래프내의 모든 노드들을 체계적으로 “방문”하고 모든 선분들을 조사하는 자연스런 방법이다. 우리는 앞으로 다룰 여러 장들에서 이 방법의 한 일반화된 형태를 약간씩 변형시킴으로써 다양한 형태의 그래프 문제들을 해결함을 볼 것이다.

지금부터는, 그래프의 모든 부분을 조직적으로 시험해 보는 기법들을 집중적으로 다루기로 한다. 정점들이 방문되는 순서를 기록하기 위해 배열 `val[V]`를 사용하기로 한다. 이 배열의 각 엔트리는 어떤 정점도 아직 방문되지 않았음을 알리기 위해 `unseen`으로 초기화된다. 우리의 목표는 그래프의 모든 정점들을 체계적으로 방문하여(id가 1, 2, ..., V인 정점들에 대해) id번째의 정점이 방문되면 `val` 배열의 해당 엔트리에 그 id를 넣는 것이다. 아래 프로그램은 인수로서 주어진 한 정점과 동일한 연결된-요소 내에 있는 모든 정점들을 방문하는 프로시저인 `visit`를 사용한다.

```
void search()
{
    int k;
    for (k = 1; k <= V; k++) val[k] = unseen;
    for (k = 1; k <=V; k++)
        if (val[k] == unseen) visit(k);
}
```

첫 번째 루프는 `val` 배열을 초기화시킨다. 그리고 나서 첫 번째 정점에 대해 `visit` 프로시저가 호출된다. 이 결과로서 이 정점에 연결된 모든 정점들에 해당하는 `val` 배열의 엔트리 값들이 `unseen`에서 해당 id값으로 변경된다. 이런 식으로, `search` 프로시저는 `val` 배열을 스캔하여 아직 `unseen` 상태로 있는 정점을 찾아 그 정점에 대해 `visit` 프로시저를 호출하는 동작을 모든 정점들이 방문되어질 때까지 계속한다. 이 방법은 그래프의 표현방식이나 `visit`의 구현방식과는 무관함을 주목하라.

먼저, 인접-리스트 표현을 이용한 `visit`의 한 재귀적 구현 방식을 살펴보기로 한다: 한 정점을 방문하기 위해, 모든 그 정점에 연결된 선분들을 조사하여 그 선분들이 아직 `unseen` 상태의 정점들과 연결되어 있는지를 알아 봐야 한다; 만약 그렇다면, 그 정점들을 방문해야 한다.

```

void visit(int k) // DFS, adjacency lists
{
    struct node *t;
    val[k] = ++id;
    for (t = adj[k]; t != z; t = t->next)
        if (val[t->v] == unseen) visit(t->v);
}

```

그림 29.5는 우리의 샘플 그래프 내의 한 큰 요소에 대한 깊이-우선 검색 동작을 추적하여 어떻게 그 요소 내의 모든 선분이 visit(1) 호출의 결과로서 접촉되어지는지를 보여주고 있다.(그림 29.4에 있는 인접-리스트들이 만들어진 후에) 실제로, 각 선분은 두 번씩 “접촉”된

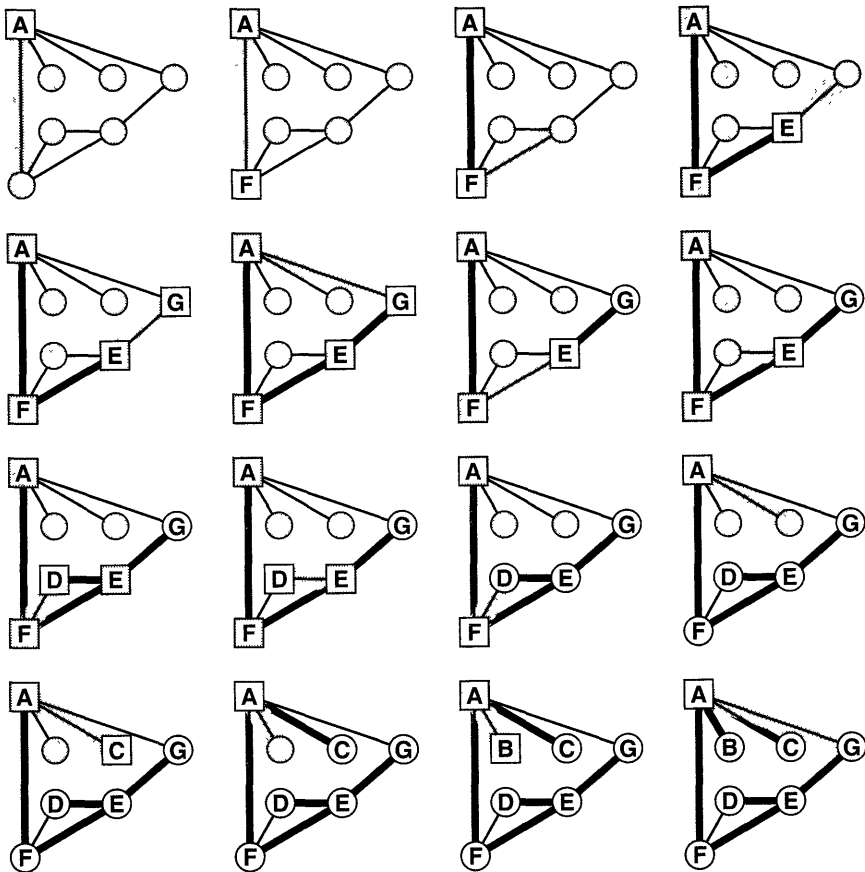


그림 29.5 큰 그래프 요소에 대한 깊이-우선 검색(재귀적)

다. 왜냐하면 각 선분은 그 선분이 연결된 정점의 두 인접-리스트 상에서 표현되기 때문이다. 그림 29.5에는 운행된 각 선분에 대해 하나씩의 다이어그램이 있다.(링크 t 가 한 인접-리스트 상의 한 노드를 가리키도록 설정(set)될 때마다) 각 다이어그램에서, “현재 처리 중인(current)” 선분은 음영이 지게 표시하고(shaded) 그 선분을 포함하는 인접-리스트를 갖는 노드는 정사각형 내에 명명된다. 그리고, 한 노드가 처음 방문될 때마다,(즉, visit에 대한 한 새로운 호출이 있을 때마다) 그 노드로 가는 선분은 굵은 점선 선으로 표시한다. 접촉이 아직 안된 노드들은 음영이 진 그러나 명명은 안된, 상태로 있게 표시하며 방문이 완료된 노드들은 명명된 정사각형을 음영 지게 함으로써 표시한다.

운행된 첫 번째 선분은 AF-즉 첫 인접-리스트 상의 첫 노드 F로 가는 선분-이다. 그리고 나서 노드 F에 대해 visit가 호출되고 선분 FA가 운행되어 진다. 왜냐하면 A는 F의 인접-리스트에 있는 첫 번째 노드이기 때문이다. 그러나 노드 A는 이 시점에서는 unseen상태에 있지 않으므로 F의 인접-리스트 상의 두 번째 엔트리에 있는 노드 E로 가는 선분 FE가 선택된다. 그리고 나서 EG가 운행되어지고, 다음에는 GE가 운행되어진다.(왜냐하면 G와 E는 각각의 인접-리스트 상에서 첫 번째에 위치하기 때문이다) 그 다음에는 GA가 운행된다; 이렇게 하여 G에 대한 방문이 완료된다.

알고리즘은 계속하여 E를 방문하고 EF를 운행하고, ED를 운행한다. 그 다음에 D를 방문하여 DE와 DF를 운행하지만, 이 둘 모두 새 노드로 가게 만들지는 않는다. 왜냐하면 D는 E의 인접-리스트 상에서 마지막 노드이기 때문이다. 이렇게하여 E의 방문은 완료되고, F의 방문 또한 FD를 운행함으로써 완료된다. 그리하여 최종적으로 A로 되돌아와서 AC, CA, AB, BA, 그리고 AG를 운행한다.

깊이-우선 검색을 추적해 가는 또 하나의 방법은 그림 29.6에 있는 것처럼 visit 프로시저의 수행 기간동안 재귀적 호출들에 의해 표시된 그래프를 다시 그리는 것이다. 그러면 각 연결된-요소들은 트리로 그려지게 되는데, 이를 그 요소에 대한 깊이-우선 검색 트리라고 한다. 이 트리를 전위 운행(preorder traversal)하면 검색시 처음 마주쳤던 그래프 상의 정점들이 나타난다; 이 트리를 후위 운행(postorder traversal)하면 정점들에 대한 visit가 완료되는 순서대로 정점들이 나타난다. 이 깊이-우선 검색 트리들의 포리스트가 그래프를 그리는 또 한 가지의 방법임을 주목함이 중요하다: 여기서, 검색 알고리즘은 그래프의 모든 정점과 선분을 검사한다.

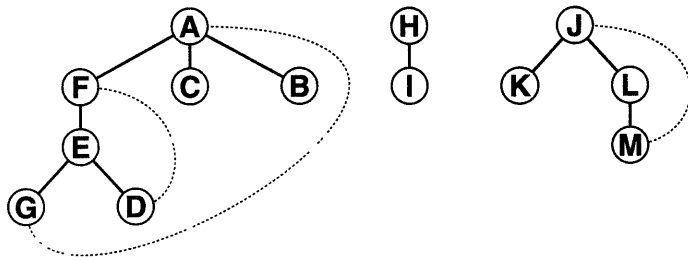


그림 29.6 깊이-우선 검색 포리스트

그림 29.6의 실선은 하부 정점이 상부 정점의 선분 리스트 상에 있음이 알고리즘에 의해 찾아졌고 아직 방문되지 않았음을 나타낸다. 점선은 이미 방문된 정점으로 가는 선분을 나타낸다; 그래서, visit 프로시저 내의 if 조건 검사가 실패하게 되고 그 선분은 재귀적 호출에 의해 “추적”되지 않게 된다. 지금 이 설명은 각 선분이 처음 마주칠 때에 해당하는 설명이다; 또한 visit 내의 if 조건 검사는 그림 29.5에서 본 것 처럼 선분들과 두 번 마주칠 때 이들을 추적하는 것을 방지해 준다.

무방향 그래프들에 대한, 이러한 깊이-우선 검색 트리들의 한 중요한 성질은 점선들이 항상 한 노드에서 트리상의 한 조상 노드(동일 트리상에 있고 루트로 가는 경로 상에서 상부에 위치한 노드)로 간다는 것이다. 알고리즘 실행 중의 어느 시점에서든지 정점들은, 방문이 끝난 정점들, 방문이 일부 끝난 정점들, 그리고 아직 방문이 안된 정점들의, 세 가지 클래스로 구분된다. 방문의 정의에 따르면, 첫 번째 클래스에서는 어떤 정점을 가리키는 선분도 만날 수 없을 것이고, 세 번째 클래스에서는, 만약 한 정점에 대한 선분과 만나게 되면, 재귀적 호출이 행해질 것이다.(그래서 그 선분은 깊이-우선 검색 트리에서 실선이 될 것이다) 이제 남은 정점들은 두 번째 클래스에 속하는 것들이다. 이들은 현 정점에서 동일 트리 내의 루트로 가는 경로 상에 있는 바로 그 정점들이다. 그리고 이들 중의 한 정점에 대한 선분은 깊이-우선 검색 트리에서의 점선에 해당한다.

성질 29.1 인접-리스트로 표현된 한 그래프의 깊이-우선 검색은 $V + E$ 에 비례하는 시간을 필요로 한다.

우리는 V 개의 val 배열 값을 세트시킨다.(따라서 V 항) 또한 각 선분을 두 번씩 검사한다.(따라서 E 항) 어떤 경우에는, $E < V$ 인 극도로 스파아스한 그래프를 만날 수도 있다. 그러나 만약 독립된-정점들(isolated vertices)이 허용되지 않는다면,(예로서, 이 정점들은 선 처

리(preprocessing) 단계에서 제거 될 수 있다) 깊이-우선 검색의 수행 시간은 선분의 수에 선형적으로 간주될 수 있다. □

동일한 기본 방법이 인접-행렬들로 표시된 그래프들에 대해, 아래 visit 프로저를 사용하여, 적용될 수 있다:

```
void visit(int k) // DFS, adjacency matrix
{
    int t;
    val[k] = ++id;
    for(t = 1; t <= V; t++)
        if (a[k][t] != 0)
            if (val[t] == unseen) visit(t);
}
```

한 인접-리스트를 통해 운행하는 것(traveling)은 인접-행렬의 한 행을(선분들에 해당하는) 1인 값들을 찾기 위해 스캔하는 것에 해당한다. 전과 마찬가지로, 아직 미 방문된 한 정점에 대한 선분은 재귀적 호출을 통해 “추적”된다. 이제는 각 정점에 연결된 선분들이 다른 순서로 검사된다. 따라서 그림 29.7에 있는 것처럼, 또 하나의 다른 깊이-우선 검색 포리스트가 만들어진다. 이것은 깊이-우선 검색 포리스트가, 그 특정 구조가 검색 알고리즘과 사용된 내부 구조에 따라 좌우되는, 그래프의 한 다른 표현에 지나지 않는다는 점을 보여주고 있다.

성질 29.2 인접-행렬로 표현된 그래프의 깊이-우선 검색은 V^2 에 비례하는 시간을 필요로 한다.

이 사실의 증명은 아주 간단하다: 인접-행렬의 모든 비트가 검사되니까 그렇다. □

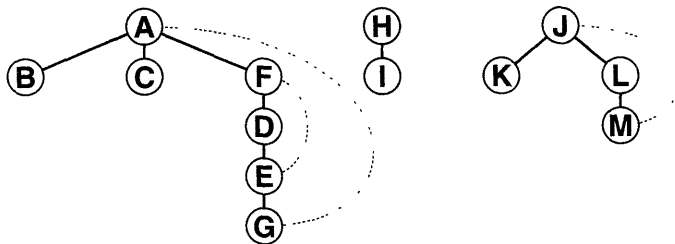


그림 29.7 깊이-우선 검색 포리스트(그래프의 행렬 표현)

깊이-우선 검색은 즉시 일부 기본적인 그래프-처리 문제들을 해결해 준다. 예를 들면, 이 프로시저는 연결된-요소들을 차례로 찾는 것에 그 기반을 두고 있다: 연결된 요소들의 수는 프로그램의 마지막 줄에 있는 visit가 호출된 수와 같다. 또한 한 그래프가 사이클을 갖는지의 시험도 위 프로그램을 조금 수정하면 가능하다. 그래프는 unseen 상태에 있지 않은 노드가 visit에서 찾아지기만 하면 사이클을 가진다. 즉, 이미 방문된 한 정점을 가리키는 선분과 만나게 되면, 사이클이 있는 것이다. 궁극적으로, 깊이-우선 검색 트리들의 점선들은 모두 사이클을 형성한다.

비재귀적 깊이-우선 검색(Nonrecursive Depth-First Search)

그래프의 깊이-우선 검색은 트리 운행의 한 일반화된 형태이다. 트리상에서 실행될 때, 이 검색은 트리 운행과 정확히 같다; 그래프에서, 이 검색은 그래프를 스패(span)하는 트리운행에 해당하는데, 이 때의 트리는 검색이 진행되면서 찾아진다.

깊이-우선 검색에서의 재귀(recursion)는, 5장의 트리 운행에서의 재귀를 제거한 것과 같은 방법으로, 스택을 사용하여 제거될 수 있다. 트리의 경우, 재귀의 제거로 또 하나의(오히려 간단한) 구현이 만들어짐을 알았으며 이와 관련된 한 비재귀적(레벨-순서) 운행 알고리즘을 찾았다. 그래프의 경우, 우리는 이와 유사한, 궁극적으로(31장에서 볼) 한 범용(general-purpose) 그래프-운행 알고리즘을 가져다 주는, 진행 과정을 보게 될 것이다.

5장에서 얻은 경험에 따라, 바로 스택-기반 구현을 하기로 한다:

```
Stack stack(maxV);
void visit(int k) // non-recursive DFS, adj lists
{
    struct node *t;
    stack.push(k);
    while (!stack.empty())
    {
        k = stack.pop(); val[k] = ++id;
        for (t = adj[k]; t != z; t = t->next)
            if (val[t->v] == unseen)
                { stack.push(t->v); val[t->v] = -1; }
    }
}
```

접촉은 됐지만 아직 미 방문된 정점들은 스택에 보관된다. 한 정점을 방문하기 위해, 우리는 그 정점에 대한 선분들을 따라 운행하고 아직 미 방문되고 스택상에 없는 정점은 스택에 집어넣는다. 재귀적 구현의 경우에는, “부분 방문된” 정점들에 대한 기록이 재귀적 프로시저 내의 지역변수(local variable) t 내에 있었다. 이를 인접-리스트들 내에(t 에 해당하는) 포인터들을 유지시킴으로써 직접 구현할 수도 있다. 그러나 그 대신, 스택 상의 정점들을 표시할 수 있도록 단순히 val 배열 엔트리들의 의미를 확장시킨다: unseen과 동일한 val 엔트리 값을 갖는 정점들은(전과 마찬가지로) 아직 만난적이 없는 것들이고, 음수 val 엔트리 값을 갖는 정점들은 스택상에 있는 것들이고, 1에서 V 사이의 val 엔트리 값들을 갖는 정점들은 이미 방문된 것들이다.(이 정점들의 인접-리스트들 상의 모든 선분들은 스택에 넣어진다)

그림 29.8은 우리의 샘플 그래프의 첫 네 노드들이 방문되어질 때의 스택-기반 깊이-우선 검색 프로시저의 동작을 추적한 것이다. 이 그림의 각 다이어그램은 한 노드의 방문에 해당한다: 방문된 노드는 정사각형으로 그려지고, 그 노드의 인접-리스트 상에 있는 모든 선분들은 음영으로서 표현된다. 전과 마찬가지로, 아직 만나지 않은 노드들은 이름이 없이 음영이진 형태로 나타내진다. 방문이 완료된 노드들은 이름이 부여되며 음영이 없어지고, 각 노드는 자신을 스택에 넣어지게 만든 노드와 굵은 검정색 선분에 의해 연결된다. 스택상의 노드는 정사각형으로 표시되어 있다.

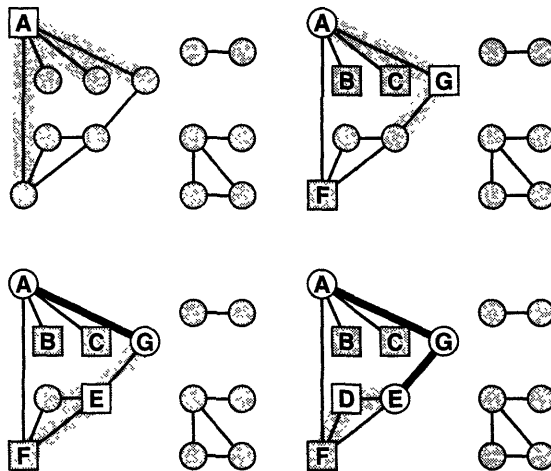


그림 29.8 스택-기반 검색의 시작

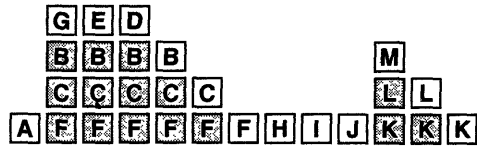


그림 29.9 스택-기반 검색 동안의 스택 내용

제일 먼저 방문된 노드는 A이다: 선분 AF, AB, AC와 AG가 운행된다. 그리고 F, B, C와 G가 스택에 넣어진다. 그리고 나서 G가 끄집어내진다.(G가 A의 인접-리스트 상의 마지막 노드였음에 주목하라) 그래서, 선분 GA와 GE가 운행되고, 그 결과로서 E가 스택에 넣어진다.(A는 스택에 다시 넣어지지 않는다) 그리하여 EG, EF와 ED가 운행되고 D가 스택에 넣어진다.... 그림 29.9는 검색 동안의 스택의 내용을 보여준다. 그림 29.10은 검색의 완료 과정을 보여준다.

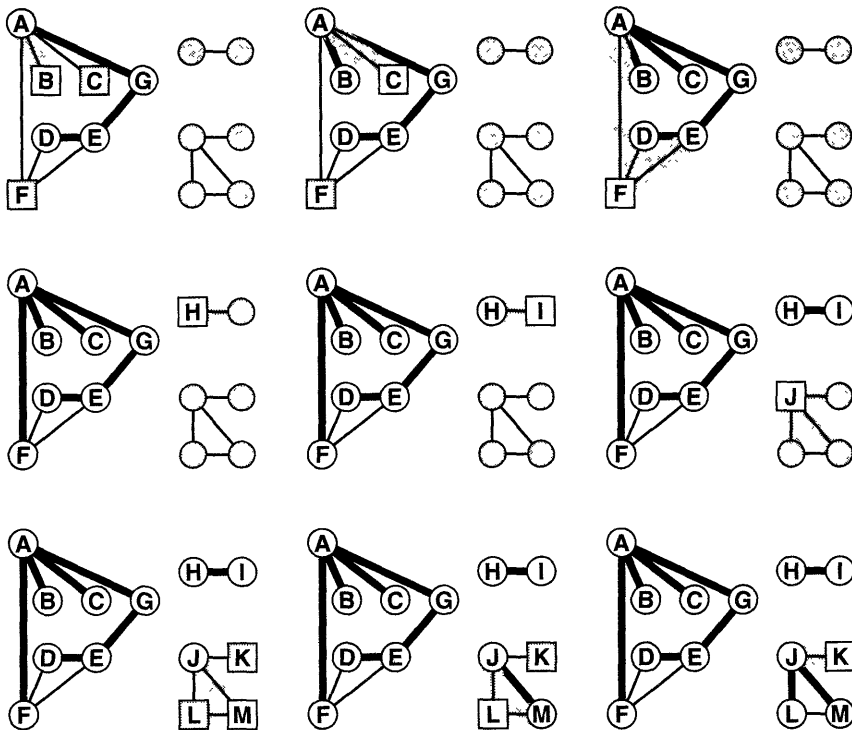


그림 29.10 스택-기반 검색의 완료과정

위 프로그램이 재귀적 구현과 같은 순서로 선분들과 노드들을 방문하지 않음을 명백히 알 수가 있다. 스택에 선분들을 역순으로 넣고 스택상에 이미 있는 한 정점과 다시 만날 경우 처리를 다르게 하므로 그럴 수가 있다. 먼저, 만약 각 노드에 대한 인접-리스트 상에 있는 선분들을 그들이 리스트 상에 있는 순서와 역으로 스택에 집어넣는다면, 재귀적 구현시와 똑 같은 순서로 해당 노드들을 끄집어내고 방문할 수도 있다.(이것은 5장의 트리 운행의 비재귀적 구현에서 오른쪽의 서브 트리를 왼쪽 것 보다 먼저 스택에 넣는 것과 같은 경우이다) 보다 근본적인 차이는 스택-기반 방법이 기술적으로는 전혀 “깊이-우선 검색”이 아니라는 점이다: 왜냐하면 이 방법은 가장 최근에 스택에 넣어진 노드를 방문하지, 재귀적 깊이-우선 검색의 경우처럼 가장 최근에 마주친 노드를 방문하지 않기 때문이다. 이 문제는 스택에 현재 있는 노드들을 다시 만나게 될 때 스택의 맨 위에 있도록 이동시킴으로써 해결할 수가 있다. 그러나 이 연산 동작은 스택보다 더 강력한 데이터 구조를 필요로 한다. 이의 구현을 위한 한 가지 간단한 방법은 31장에서 다루기로 한다.

너비-우선 검색(Breadth-First Search)

트리 운행의 경우와 마찬가지로(4장을 보라), 정점들을 저장하기 위한 데이터 구조로서 스택이 아니라 큐를 사용할 수도 있다. 이렇게 함으로써 너비-우선 검색(breadth-first search)이라 불리는 두 번째의 고전적 그래프-운행 알고리즘이 만들어진다. 너비-우선 검색의 구현을 위해, 위의 스택-기반 검색 프로그램의 스택 연산 동작을 큐 연산 동작으로 바꾼다:

```
Queue queue(maxV);
void visit(int k) // BFS, adjacency lists
{
    struct node *t;
    queue.put(k);
    while (!queue.empty())
    {
        k = queue.get(); val[k] = ++id;
        for(t = adj[k]; t != z; t = t->next)
            if (val[t->v] == unseen)
                { queue.put(t->v); val[t->v] = -1; }
    }
}
```

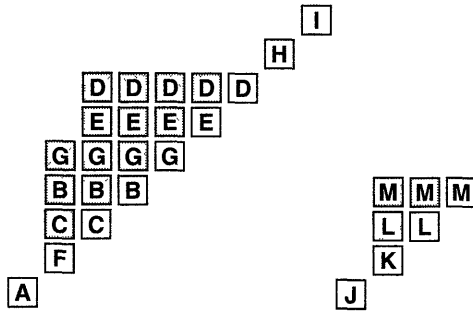


그림 29.11 너비-우선 검색 중의 큐 내용

이렇게 데이터 구조를 바꾸면 노드들의 방문 순서가 영향을 받는다. 우리의 작은 샘플 그래프의 경우, 선분들은 AF AC AB AG FA FE FD CA BA GE GA DF DE EG EF ED HI IH JK JL JM KJ LJ LM MJ ML의 순으로 방문된다. 운행 중의 큐 내용을 그림 29.11에서 볼 수 있다.

깊이-우선 검색에서처럼, 그림 29.12에 있는 것 같은, 각 노드로 처음으로 가게 해주는 선분들로부터 포리스트를 정의할 수 있다. 너비-우선 검색은 이 포리스트에서 레벨(level)순서에 따라서 트리들을 운행하는 것에 해당한다.

깊이-우선 검색과 너비-우선 검색의 두 알고리즘 모두에서, 정점들은 이미 데이터 구조에서 꼬집어내진 트리(즉 방문된) 정점들, 트리 정점들에 인접해 있지만 아직 미 방문된 fringe 정점들, 그리고 아직 전혀 만난적이 없는 unseen 정점들의 세 가지 유형들로 구분된다. 만약 각 정점이 자신을 데이터 구조에 추가되도록 하는 선분과 연결돼 있다면(그림 29.8과 그림 29.10에서의 굵은 검정색 선분들), 이러한 선분들은 트리를 형성한다.

그래프의 한 연결된-요소를 체계적으로 검색하기 위해(visit 프로시저를 구현하기 위해), fringe상의 한 정점에서 시작하여(이 때 모든 다른 정점들은 unseen상태에 있다) 모든 정

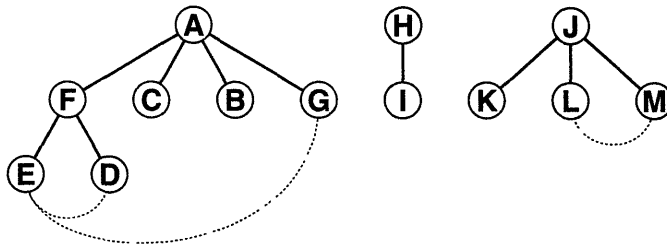


그림 29.12 너비-우선 검색 트리

점들이 방문 될 때까지 다음 스텝을 수행한다: “fringe에 있는 한 정점(x 라 하자)을 트리로 옮기고 x 에 인접한 unseen 정점들을 fringe에 넣는다.” 그래프 운행 방법들은 어떤 정점이 fringe에서 트리로 옮겨가야 하는지를 결정하는 방법들에 있어 차이가 있다. 깊이-우선 검색에서는 가장 최근에 마주친 fringe상의 정점을 선택하는데, 이는 fringe상의 정점의 선택을 위해 스택을 사용하는 것에 해당한다. 너비-우선 검색에서는 가장 최근에 마주친 fringe상의 정점을 선택하는데, 이는 fringe상의 정점의 선택을 위해 큐를 사용하는 것에 해당한다. 31장에서 fringe에 대해 우선 순위 큐(priority queue)를 사용할 경우의 영향에 대해 배우게 될 것이다.

깊이-우선 검색과 너비-우선 검색 간의 차이점은 큰 그래프를 고려할 때 아주 명백해진다. 그림 29.13은 큰 그래프에서의 깊이-우선 검색 동작과정을 삼분의 일 수행했을 경우와 삼분의 이 수행 했을 경우를 보여준다; 그림 29.14는 너비-우선 검색 동작 과정을 보여준다. 이들 다이어그램에서, 트리 정점들과 선분들은 검정색으로, unseen 정점들은 음영 지게, 그리고 fringe 정점들은 흰색으로 표시했다.

이 두 경우, 검색은 좌측 하단의 노드에서 시작한다. 깊이-우선 검색은 그래프를 따라 수직으로 진행하며 경로들이 분기되는(branch off) 모든 점들은 스택상에 저장된다; 너비-우선 검색은 그래프를 따라 “수평으로 진행”하며 큐를 사용하여 방문된 장소들의 맨처음(frontier)을 기억한다. 깊이-우선 검색은 시작점에서부터 멀리 있는 그래프 상의 새로운 정점들을 찾아다니는데, 길이 없을 경우에만 근접한 정점들을 택한다; 너비-우선 검색은 시작점에서 가까운 영역을 완전히 검색하는데, 모든 근접한 점들이 검색된 다음에야 멀리 있는 것들로 이동해 간

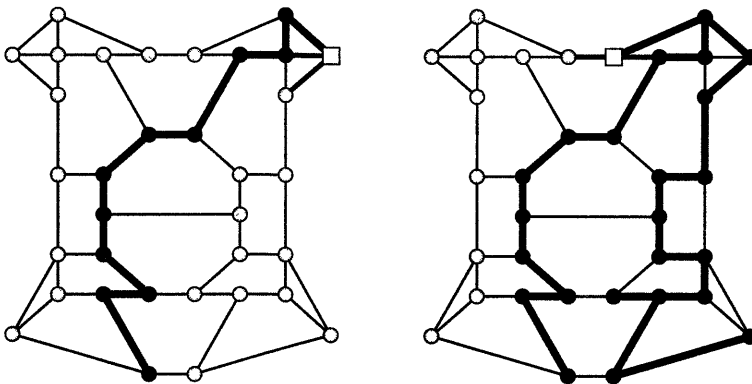


그림 29.13 큰 그래프에서의 깊이-우선 검색

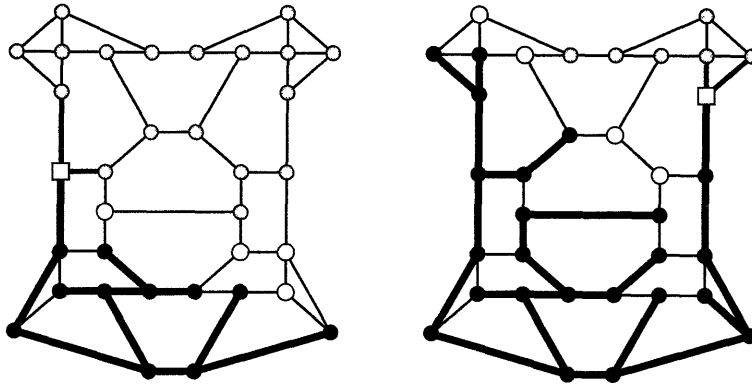


그림 29.14 큰 그래프에서의 너비-우선 검색

다. 대체로 노드들이 방문되는 순서는 선분들이 입력 상에 나타나는 순서와 정점들이 인접-리스트들 상에 나타나는 순서에 따라 이 선분 입력 순서가 받는 영향력에 의해 좌우된다.

이러한 동작상의 차이점들 외에도, 이들 방법들의 구현에 있어서 근본적인 차이점들이 있게 하는 이유들도 관심거리가 된다. 깊이-우선 검색은 재귀적으로 아주 간단히 표현된다.(왜냐하면 그 기초가 되는 데이터 구조는 스택이기 때문이다) 그리고 너비-우선 검색의 경우는 간단한 비재귀적 구현이 허용된다.(왜냐하면 그 기초가 되는 데이터 구조가 큐이기 때문이다) 31장에서는 그래프 알고리즘들에 실제로 내재하는 데이터 구조는 우선 순위 큐이고, 이는 흥미있는 성질들과 알고리즘들이 아주 많음을 의미함을 보게 될 것이다.

미로(Mazes)

그래프의 모든 정점과 선분을 검사하는 체계적인 방법은 잘 알려진 오랜 역사가 있다: 깊이-우선 검색은 미로들의 운행 방법으로써 수백 년 전에 처음으로 공식적으로 언급되었다. 예를 들어, 그림 29.15의 왼쪽 다이어그램은 한 유명한 미로이며, 오른쪽 그래프는 미로에 선택해야 할 둘 이상의 경로가 있는 지점에 정점을 위치시키고 이들을 경로에 따라 연결함으로써 형성된다. 이 미로는 울타리(벽)를 따라 만들어진 경로들로 구성된 초기 영국의 정원 미로들 보다 더 복잡하다. 이 미로들에서는, 모든 벽들이 바깥 쪽 벽들과 연결되어 있다. 그래서 안으로 들어간 사람들 중 영리한 사람들은 단지 그들의 오른손을 벽에 닿게 함으로써 나가는 길을 찾을 수가 있다.(실험실용 쥐가 이 방법을 배웠다는 보고가 있다) 독립적인 내

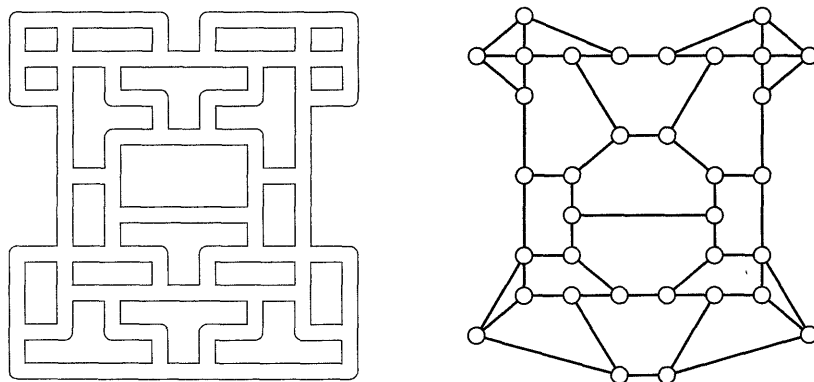


그림 29.15 한 미로와 그 관련 그래프

부 벽들이 있는 경우에는, 미로를 빠져 나오기 위해 보다 복잡한 전략이 필요하다. 결과적으로, 깊이-우선 검색이 필요하게 된다.

깊이-우선 검색을 이용하여 미로의 한 곳에서 다른 곳으로 가기 위해, 시작점에 해당하는 그래프 상의 정점에서 visit를 수행한다. visit가 재귀적 호출을 통해 선분을 따라 갈 때마다, 우리는 미로에서의 해당 경로를 따라 걷는다. 미로를 걸어다니는 비결은 우리가 각 정점으로 들어가기 위해 사용했던 경로를 따라, 정점에 대한 visit가 끝날 때마다, 왔던 길로 되돌아가야 한다. 이렇게 하면 우리는 깊이-우선 검색 트리에서 한 스텝 위의 정점으로 되돌아가게 되어 다음 선분을 따라 갈 수 있게 된다.(이 과정은 그래프에 대한 깊이-우선 검색의 운영을 정확히 모방한다) 깊이-우선 검색은 사람이 미로에서 어떤 것을 찾는데 적합하다. 왜냐하면 “다음 번 찾을 곳”은 항상 옆에 있기 때문이다; 너비-우선 검색은 모든 방향으로 흩어져서 어떤 것을 찾는 한 집단의 사람들과 아주 비슷하다.

전망(Perspective)

다음 장들에서는 무방향 및 방향성 그래프들의 연결 성질들을 결정해 주는데 주목적을 둔 다양한 그래프 알고리즘들을 살펴 볼 것이다. 이들 알고리즘들은 그래프들을 처리하기 위한 근본적인, 그러나 주제에 대한 시작에 불과하다. 이 책의 범위를 넘는 많은 흥미롭고 유용한 알고리즘들이 개발되어져 왔고, 좋은 알고리즘들이 아직 찾아지지 않은 많은 흥미로운 문제들이 연구되고 있다.

여기서 설명하기에는 너무 복잡한, 아주 효율적인 알고리즘들이 일부 개발되어 있다. 예로서, 그래프가 선분들의 교차 없이 평면상에 그려질 수 있는지의 여부를 효율적으로 결정해주는 것이 가능하다. 이 문제는 평면화(planarity) 문제라고 불리는데 이 문제의 해결을 위한 어떤 효율적인 알고리즘도 1974년 R.E. Tarjan이 이 문제를 선형 시간 내에 해결해 주는 깊이-우선 검색을 사용하는 한 교묘한(그러나 아주 복잡한) 알고리즘을 개발해 내기 전까지는 알려진 바가 없었다.

아주 자연스럽게 발생하고 그래서 설명하기 쉬운 일부 그래프 문제들도 해결하기에는 아주 어려울 수 있다. 이들의 해결을 위한 어떤 좋은 알고리즘들도 알려진 바 없다. 예로서, 한 가중치 그래프에서 각 정점을 방문하는 최소-비용 여행 경로를 찾는 어떤 효율적 알고리즘도 알려진 바가 없다. 외판원(traveling salesman) 문제로 불리는 이 문제는 45장에서 보다 자세히 논의 될 한 부류의 어려운 문제들에 속한다. 대부분의 전문가들은 어떤 효율적인 알고리즘도 이러한 문제들에 대해 존재하지 않는다고 믿고 있다.

다른 그래프 문제들은, 아직 발견되지 않았더라도, 효율적인 알고리즘들을 갖는 것이 당연하다. 이런 문제의 한 예가, 두 그래프가 정점 명들을 개명함으로써 같아질 수 있는지를 결정하는, 그래프 동형(graph isomorphism) 문제이다. 많은 특별한 형태의 그래프들의 경우, 이 문제에 대한 효율적인 알고리즘들이 알려져 있다. 그러나 일반적인 경우는 아직 해결되지 않은 상태이다.

요약하면, 그래프를 다루는 아주 폭 넓은 영역의 문제들과 알고리즘들이 있다. 나타나는 모든 문제들의 해결을 기대 할 수 없음은 자명하다. 심지어는 아주 단순해 보이는 문제들도 전문가들을 당황스럽게 만든다. 그러나, 많은(상대적으로) 쉬운 문제들도 아주 빈번히 발생하며 우리가 공부할 그래프 알고리즘들은 아주 다양한 응용 분야들에 걸쳐 잘 동작한다.

연습문제

1. 한 정점의 고립(어떤 다른 정점과도 연결 안된 상태) 여부를 빠르게 결정하기 위해 가장 적합한 무방향 그래프 표현은 ?
2. 깊이-우선 검색이 이진 검색 트리 상에서 이용되고, 각 노드에서 오른쪽 선분이 왼쪽 선분보다 먼저 탐색된다고 하자. 어떤 순서로 노드들이 방문되나?
3. V 개의 노드와 E 개의 선분을 가지는 무방향 그래프를 인접-행렬로 표현하는데 필요한 저장 비트들의 수는? 인접-리스트로 표현하는데 필요한 저장 비트들의 수는?
4. 두 선분이 교차함이 없이 종이 위에 그려질 수 없는 그래프 하나를 그려라.
5. 인접-리스트들로서 표현된 그래프에서 한 선분을 제거하는 프로그램을 작성하라.
6. 정점 인덱스의 정렬된 순서로서 인접-리스트들을 유지하는 한 버전의 `adjlist` 프로그램을 작성하고 이 방식의 장점을 기술하라.
7. `search` 프로시저가 정점들을 역순으로(V 에서부터 1까지) 스캔할 때 본문의 예제에 대해 결과로서 나타나는 깊이-우선 검색 포리스트들을(인접-행렬 표현과 인접-리스트 표현의 두 가지 표현의 경우에 대해) 그려라.
8. 정점의 수 V , 선분의 수 E , 그리고 연결된-요소의 수인 C 의 관점에서, 무방향 그래프를 깊이-우선 검색할 경우 얼마나 많이 `visit` 프로시저가 호출되나?
9. 샘플 그래프의 선분들이 그림 29.4에 있는 구조를 만들기 위해 사용된 순서의 역순으로 읽어 들여질 때 산출되는 인접-리스트들을 구하라.
10. 9번의 인접-리스트들에 대해 재귀적 루틴이 사용됐을 경우 본문의 샘플 그래프에 대한 깊이-우선 검색 포리스트를 그려라.

빈 면

전 장의 기본적인 깊이-우선 검색 프로시저는 한 주어진 그래프의 연결된-요소들을 찾는다; 이 장에서는 이와 관련된 알고리즘들과 다른 그래프-연결 성질들에 대한 문제들을 다루기로 한다.

연결에 대한 정보를 얻기 위해 몇 개의 깊이-우선 검색의 직접적인 응용 분야들을 본 후에, 연결선의 한 일반적인 이중연결(biconnectivity)에 대해 알아보기로 한다. 우리는 한 그래프의 한 정점에서 다른 정점으로 가는 한 방법보다 더 많은 방법들이 있는지를 알아보는데 관심이 있다. 한 그래프가 각 정점 쌍들을 연결하는 적어도 두개의 다른 경로가 있기만 하더라도, 그 그래프는 이중 연결된 것이다. 따라서 한 정점과 이에 연결된 모든 선분들이 제거되더라도, 그 그래프는 아직 연결되어 있다. 어떤 응용 분야에서 그래프가 연결되어 있다는 것이 중요하다면, 그 그래프가 연결된 상태로 남아 있는 것 또한 중요할지도 모른다. 이러한 문제에 대한 해결책은 전 장의 유향 알고리즘들보다 더 복잡한 알고리즘들이다. 그러나 이들 역시 깊이-우선 검색에 기초를 두고 있다.

자주 발생하는 연결성 문제의 한 특정한 경우는 선분들이 두개의 특정 정점이 동일한 연결된-요소에 속하는 지에 대한 질문들에 대한 응답으로서 선분들이 그래프에 하나씩 추가되는 동적인 상황을 수반한다. 이 문제는 잘 연구되어진 문제인데, 이와 관련된 두 개의 “고전적인” 알고리즘을 자세히 살펴보겠다. 이 방법들은 간단하고 널리 이용될 뿐만 아니라, 또한 단순한 알고리즘들도 분석하기가 얼마나 어려운지를 보여준다. 이 문제는 종종 “찾기-합하기” 문제로 불려지는데, 이 명칭은 원소들의 집합들에 대한 단순한 연산을 하는데 이 알고리즘들을 적용시킨 데서부터 비롯됐다.

연결된-요소들(Connected Components)

전 장의 어떤 그래프-유행 방법도 한 그래프의 연결된-요소들을 찾는 데 이용될 수 있다. 왜냐하면 이들은 모두 다른 연결된-요소로 가기 전에 한 연결된-요소 내의 모든 노드들을 방문하는 전략에 근거하여 만들어졌기 때문이다. 연결된-요소들을 출력하는 한 쉬운 방법은 재귀적 깊이-우선 검색 프로그램들 중의 하나를 수정하여 visit 프로시저가 방문 중인 정점을 출력하도록 만들고(즉, 끝내고 빠져 나오기 바로 전에 name(k)를 출력함으로써), 한 새로운 연결된-요소가 시작되어진다는 것의 표시를 search내에 있는 visit를(비 재귀적으로) 호출하기 바로 전에 출력한다. 이 기법은 (그림 29.1의) 우리 샘플 그래프 상에서 깊이-우선 검색 (29장의 search와 visit의 인접-리스트 버전)을 수행할 때 다음과 같은 출력 결과가 나올 수 있다:

```
G D E F C B A
I H
K M L J
```

visit의 인접-행렬 버전, 스택에 근거한 깊이-우선 검색, 그리고 너비-우선 검색 같은 다른 종류의 알고리즘들도 동일한 연결된-요소들을 계산해 낼 수 있다. 그러나 정점들이 다른 순서로서 출력될 것이다.

연결된-요소들에 대해 좀더 복잡한 처리를 하도록 확장하는 것은 간단하다. 예로서, val[k]=id 문 뒤에 inval[id]=k를 삽입함으로써, val 배열의 “역”이 얻어진다. 이 경우, id번째의 엔트리는 방문된 id번째 정점의 인덱스이다. 동일한 연결된-요소에 있는 정점들은 이 배열 상에서 연속하여(contiguous)있게 되고, 매번 visit가 search에서 호출될 때마다 id값에 의해 주어진 각 새로운 연결된-요소의 인덱스가 된다. 이 값들은 inval에서 구분 문자(delimiter)로서 저장되거나 이용될 수 있다.(예로서, 각 연결된-요소의 첫 엔트리는 음수 값이 될 수 있다)

k	1	2	3	4	5	6	7	8	9	10	11	12	13
name[k]	A	B	C	D	E	F	G	H	I	J	K	L	M
val[k]	1	7	6	5	3	2	4	8	9	10	11	12	13
inval[k]	-1	6	5	7	4	3	2	-8	9	-10	11	12	13

그림 30.1 연결된-요소들에 대한 데이터 구조들

그림 30.1은 search의 인접-리스트 버전이 이 방법으로 수정될 경우 우리의 예제에 대해 이 배열들이 갖는 값들을 보여준다. 통상적으로, 이같은 기법들은 그래프를 더 복잡한 알고리즘들에 의해 나중에 처리될 수 있도록 하기위해 연결된-요소들로 나눌때 사용할 만한 가치가 있다. 이 경우, 더 복잡한 알고리즘들은 연결되지 않은 요소들을 처리해야 하는 부담감을 덜 수가 있다.

이중연결성(Biconnectivity)

종종 그래프의 점들간에 둘 이상의 루우트(route)들이 있도록 설계함이 좋은 경우가 종종 있다. 이렇게 하는 경우, 연결점들(정점들)에서 발생할 수 있는 고장(failure)을 처리할 수가 있다. 예로서, 비록 New York에 눈이 오더라도, New York대신 Philadelphia를 거쳐서 비행기를 타고 Providence에서 Princeton까지 갈 수가 있다. 또한 한 IC회로의 주요 통신 선들은 이중연결된 경우가 많다; 그래서 그 회로의 한 요소가 고장나더라도 그 회로의 나머지 요소들은 계속 동작할 수가 있다.(비록 실제적인 것은 아니지만 개념을 자연스럽게 설명할 수 있는) 또 다른 응용 분야로서 적군이 아군의 철도 수송로들을 차단하기 위해 적어도 두 개의 역에 폭탄 투하를 하는 전시 상황을 상상해 볼 수 있다.

한 연결된 그래프에서의 연결점(articulation point)은, 만약 없어진다면, 그 그래프를 둘 이상의 조각들로 나뉘지게 만들 수 있는 정점이다. 연결점들이 없는 그래프들을 이중연결됐다고 한다. 이중연결된 그래프에서는, 두 개의 다른 경로가 각 정점 쌍들마다 있다. 이중연결이 안된 그래프는 두 다른 경로를 통해 상호 액세스가 가능한 노드들의 집합인 이중연결된 요소들로 나누어질 수 있다.

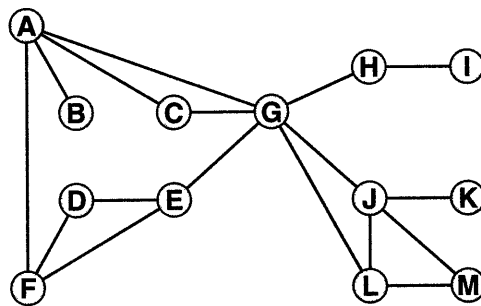


그림 30.2 이중연결 되지않은 그래프

그림 30.2는 연결된 그러나 이중연결은 되지 않은 그래프를 보여준다.(이 그래프는 전 장의 그래프에 선분 GC, GH, JG,와 LG를 추가함으로써 얻어진다. 우리의 예제들에서는, 이 네 개의 선분이 순서대로 입력의 맨끝에 추가되었다고 가정한다. 그러면 인접-리스트들은 그림 29.4에 이들 선분들로 인해 생긴 여덟 개의 새로운 엔트리들을 더한 것과 유사한 형태가 될 것이다) 이 그래프의 연결점들은 A(왜냐하면 B를 그래프의 나머지 부분과 연결해 주므로), H,(왜냐하면 I를 그래프의 나머지 부분과 연결해 주므로) 그리고 G(왜냐하면 G가 제거되면 그래프가 세 부분으로 나누어지므로)이다. 그리고 이 그래프에는 여섯 개의 이중연결된 요소들이 있다: {A C G D E F}, {G J L M}, 그리고 노드 B, H, I와 K이다.

연결점들을 찾는 것은 깊이-우선 검색을 간단히 확장하면 가능해진다. 이를 보기 위해, 그림 30.3에 있는 것과 같은 이 그래프에 대한 깊이-우선 검색 트리를 생각해 보자. 노드 E의 제거는 그래프를 절단시키지는 않는다. 왜냐하면 G와 D모두가 이들로부터(트리에서의 E의 부모인)F로 가는 대체 연결선(alternate connection)들을 만들어 주는 점선 링크들을 갖고 있기 때문이다. 이와는 달리, G의 제거는 그래프를 절단시킨다. 왜냐하면 L이나 H에서(G의 부모인) E로 가는 대체 연결선이 없기 때문이다.

정점 x 는, 모든 자식 정점 y 밑에 있는 어떤 노드가 x 보다 위에 있는 어떤 노드와(점선 링크를 통해)연결되어서 x 에서 y 로 가는 대체 연결선 있게 되면, 연결점이 아니다. 이 조사는 깊이-우선 검색 트리의 루트에 대해서는 잘 되지 않는다. 왜냐하면 루트보다 더 상위의 노드는 없기 때문이다. 루트는 둘 이상의 자식 노드를 가지면 연결점이다. 왜냐하면 그 루트의 자식 노드를 연결하는 유일한 경로는 루트를 거쳐가기 때문이다. 이러한 조사는 노드-방문 프로시저를 검색 동안 보여진 트리상의 최 상위 점(노드)을 리턴 하는 함수로 변경함으로써 다음과 같이 깊이-우선 검색에 쉽게 추가된다:

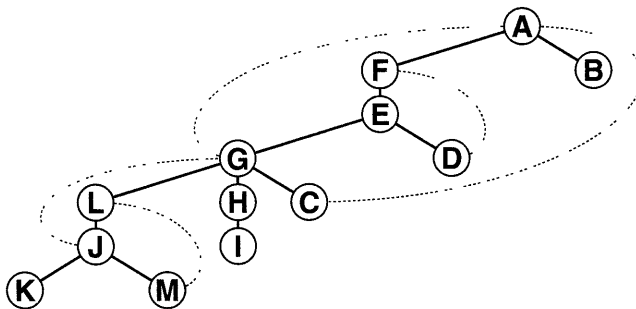


그림 30.3 이중연결에 대한 깊이-우선 검색

```

int visit(int k) // DFS to find articulation points
{
    struct node *t;
    int m, min;
    val[k] = ++id;
    min = id;
    for (t = adj[k]; t != z; t = t->next)
        if (val[t->v] == unseen)
        {
            m = visit(t->v);
            if (m < min) min=m;
            if (m >= val[k]) cout << name(k);
        }
    else if (val[t->v] < min) min = val[t->v];
    return min;
}

```

이 프로시저는 정점 k 의 임의의 후손으로부터 한 점선 링크를 통해 도달할 수 있는 트리상의 최 상위 점을 찾고, k 가 연결점인지 아닌지를 알기 위해 이 찾아진 정보를 이용한다. 보통 이 계산은 한 자식으로부터 도달할 수 있는 최소 값(정점)이 트리에서 더 상위에 있는지 아닌지에 대한 조사를 수반한다. 그러나, k 가 깊이-우선 검색 트리의 루트인지 아닌지를(또는, 이와 동등하게, 이것이 k 를 포함하는 연결된-요소에 대한 visit의 첫 호출인지 아닌지를) 결정하는 또 하나의 조사를 해야 한다. 왜냐하면 두 경우 모두에서 동일한 재귀적 프로그램을 이용하고 있기 때문이다. 이 조사는 재귀적 프로시저 visit의 밖에서 적절히 수행되므로 위 프로그램 코드에는 나타나지 않는다.

성질 30.1 한 그래프의 이중연결된 요소들은 선형 시간 내에 찾아질 수 있다.

비록 위 프로그램은 단순히 연결점들을 출력하지만, 연결된-요소들에서 행해졌던 것처럼, 연결점들과 이중연결된 요소들에 대한 부가적인 처리를 할 수 있도록 쉽게 확장될 수 있다. 위 프로그램은 깊이-우선 검색 프로시저이므로, 수행 시간은 $V+E$ 에 비례한다.(인접-행렬에 근거한 유사한 프로그램은 $O(V^2)$ 스텝 내에 수행될 수 있다.) \square

이 프로시저는, 위에서 언급한(이중연결이 신뢰도 향상을 위해 이용된) 응용 분야들 외에도, 큰 그래프들을 관리 가능한 크기의 조각들로 분할하는 경우에도 이용될 수 있다. 많은 응용 분야의 경우, 매우 큰 그래프에서 한 번에 하나씩 연결된-요소들을 처리하도록 되어짐이 명확하다; 그래프에서 한 번에 하나씩 이중연결된 요소를 처리하도록 되어짐은 다소 불명확하지만 가끔 유용할 수는 있다.

찾기-합하기 알고리즘들(Union-Find Algorithm)

어떤 분야에서는, 그래프의 한 정점 x 가 다른 정점 y 와 연결되어 있는지만을 알기를 원할 수가 있다; 이들을 연결하는 실제 경로와는 관계가 없을 수도 있다. 최근 이 문제가 조심스럽게 연구되어져 왔다; 개발된 효율적인 알고리즘들은 관심거리가 되는 이유는 이들이 집합들(객체들의 모임)의 처리에 이용될 수 있기 때문이다. 그래프들은 자연스럽게 객체들의 집합들과 일치한다: 정점들은 객체들을 나타내고 선분들은 “~로서 같은 집합 내에 있다”는 의미를 가진다. 따라서, 전 장의 샘플 그래프는 집합들 $\{A B C D E F G\}$, $\{H I\}$, 그리고 $\{J K L M\}$ 에 해당한다. 이들 각 집합을 일컫는 또 하나의 용어는 동치류(equivalence class)이다. 한 선분의 추가는 정점들에 의해 표현된 동치류들이 연결되어 결합됨에 해당한다. 우리는 “ x 와 y 는 동치인가?” 또는 “ x 는 y 와 동일한 집합에 속하는가?” 같은 기본적인 질문에 관심이 있다. 이 질문은 명백히 “정점 x 는 정점 y 와 연결되어 있는가?” 하는 기본적인 그래프 관련 질문에 해당한다.

선분들의 집합을 가지고, 우리는 해당 그래프를 인접-리스트로 표현하고 깊이-우선 검색을 사용하여 각 정점에 그 정점의 연결된-요소에 대한 인덱스를 할당할 수 있다. 따라서 “ x 는 y 와 연결되어 있는가?”와 같은 형태의 질문들에 대해 두 번의 배열 액세스와 한 번의 비교만으로 답할 수 있다. 여기서 살펴보려는 방법들의 또 한 가지의 문젯거리는 이들이 동적(dynamic)이라는 것이다: 이들은 입력된 새로운 선분들을 받아들일 수 있고 입력된 정보를 이용하여 질문들에 대해 올바르게 답할 수도 있다. 집합 문제의 관점에서, 한 새로운 선분의 추가는 합하기(union)연산이라 부르고 질문들은 찾기(find)연산이라 부른다.

우리의 목표는 두 정점 x 와 y 가 같은 집합(그래프의 경우, 같은 연결된-요소) 내에 있는지를 조사하고 만약 같은 집합 내에 없으면, 같은 집합 내에 있게 하는(그래프에서는, 이들간에 선분을 그려 넣는) 함수를 작성하는 것이다. 인접-리스트 방식이나 그래프의 다른 표현 방식을 만들어 사용하는 대신, 특별히 “찾기”와 “합하기”연산을 지원하게끔 고안된 한 내부

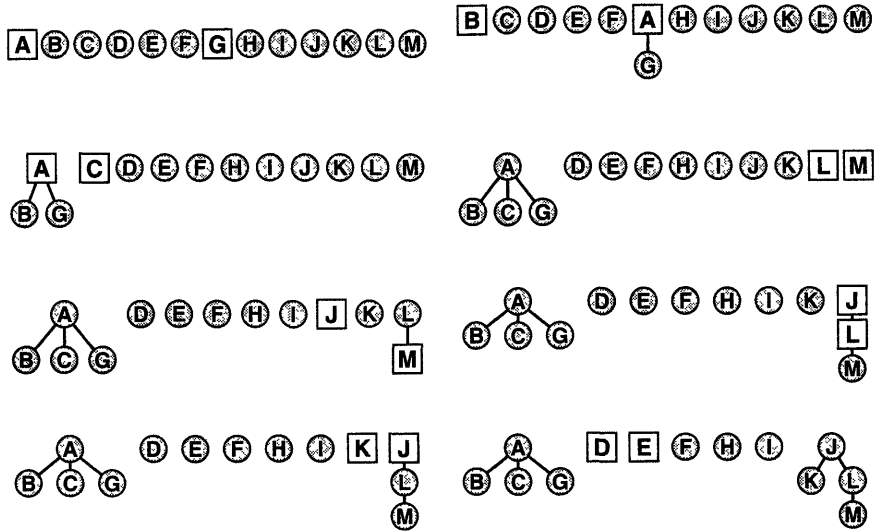


그림 30.4 찾기-합하기의 초기 스텝들

구조를 사용함으로써 효율을 높일 것이다. 이 내부 구조는 트리들로 구성된 한 개의 포리스트인데, 각 트리는 한 개의 연결된-요소를 나타낸다. 우리는 두 정점이 같은 집합 내에 있는지를 알 수 있어야 하고 두 트리를 하나로 결합시킬 수 있어야 한다. 이 두 연산 동작을 효율적으로 구현할 수 있음이 알려져 있다.

이 알고리즘이 어떻게 동작하는지를 설명하기 위해, 그림 30.1에 있는 샘플 그래프의 선분들이 AG AB AC LM JM JL JK ED FD HI FE AF GE GC GH JG LG의 순서로 처리될 때 형성되는 포리스트를 보기로 한다. 그림 30.4은 첫 일곱 스텝을 보여준다. 처음에는, 모든 노드들이 다 별도의 트리로서 존재한다. 그 다음에 선분 AG가 추가되어 A가 루트인 두 개의 노드를 갖는 트리가 만들어진다.(이 루트는 임의로 선택된다-따라서 G를 루트로 할 수도 있다) 선분 AB와 AC가 같은 방식으로 추가되어 트리에 노드 B와 C가 더해진다. 그 다음에는, LM, JM, JL 그리고 JK는 약간 다른 구조를 갖는 J, K, L, M을 포함하는 트리를 형성한다.(LM과 JM이 L과 J를 같은 연결된-요소에 있게 하므로 JL은 기여하는 바가 없다)

그림 30.5는 완료 과정을 보여준다. 선분 ED, FD와 HI가 두 트리를 더 만들어 준다. 따라서 네 개의 트리를 갖는 포리스트가 된다. 이 포리스트는 이 시점까지 처리된 선분들이 네 개의 연결된-요소를 갖는 그래프를 나타냄을 보여준다. 이를 집합적으로 표현하면 네 개의 집합 {A B C G}, {J K L M}, {D E F}와 {H I}이다. 이제 FE는 구조에 어떤 기여도 하지 않는다. 왜냐하면 F와 E가 같은 요소 내에 있기 때문이다. 그러나 AF는 처음 두 트리를 연

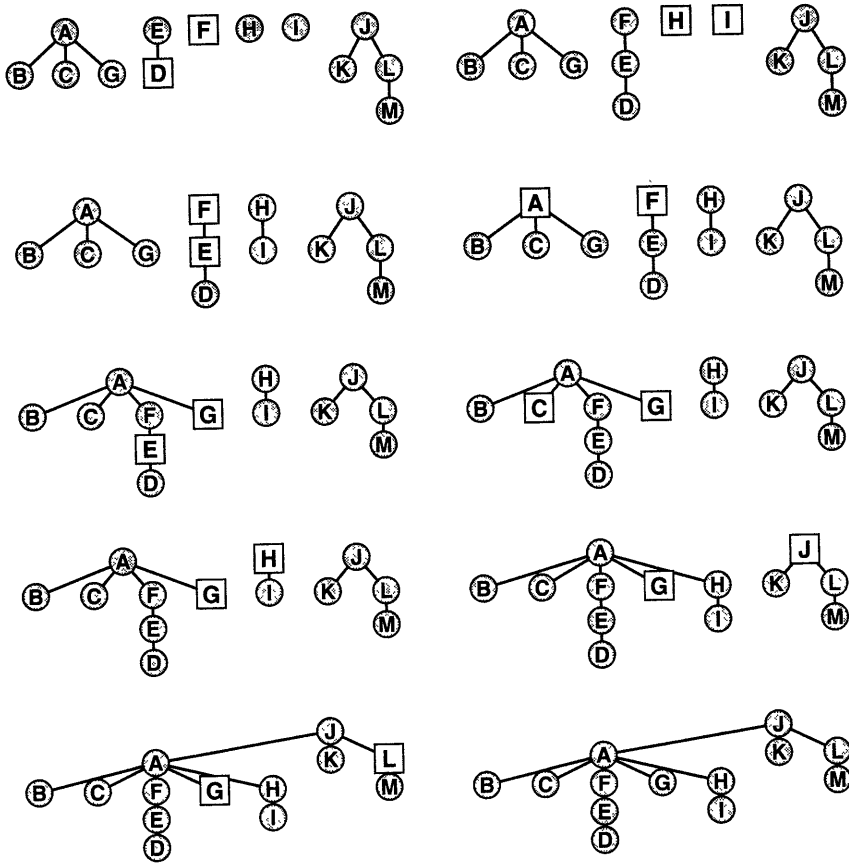


그림 30.5 찾기-합하기의 완료과정

결시킨다; 그리고 GE와 GC는 기여를 하지 않고, GH와 JG는 모든 트리들이 결합되어 한 개의 트리가 되게 한다.

깊이-우선 검색 트리와는 다르게, 이 찾기-합치기 트리들과 주어진 선분들을 갖는 해당 그래프와의 유일한 연관 관계는 같은 방식으로 정점들을 집합들로 분할시킨다는 것이다. 예로서, 트리에 있는 노드들을 연결하는 경로들과 그래프에 있는 노드들을 연결하는 경로들은 상응 관계가 전혀 없다. 이러한 포리스트들을 유지하기 위해 어떤 데이터 구조가 선택되어야 하는가? 우리는 트리의 위쪽으로만 가지, 아래쪽으로는 가지 않는다. 그러므로 “부모와의 링크”를 표현해 주는 방식(4장을 보라)이 적절하다. 특히 각 정점에 대해 그 부모에 대한 인덱스(트리의 루트는 인덱스가 0임)를 갖는 dad배열을 아래의 클래스 선언에서 명시된 것처럼 유지해야 한다:

```

class EQ
{
    private:
        int *dad;
    public:
        EQ(int size);
        int find(int x, int y, int doit);
};

```

객체 형성 함수(constructor) EQ는 dad배열을 위한 공간을 할당하고 모든 배열 값은 초기에는 0이 된다.(이 프로그램 코드는 여기서 생략되었다) “합하기”와 “찾기”의 두 연산 동작을 구현하는데 find프로시저를 사용한다. 만약 세 번째 인수가 0이면 “찾기”만 수행하고, 0이 아니면, “합하기”도 수행한다. 한 정점 y의 부모를 찾으려면, y=dad[j]만 수행하고, j가 속하는 트리의 루트를 찾으려면 이 연산 동작을 dad[j]=0이 될 때까지 반복한다. 이런 식으로 하여, 아래와 같은 find 프로시저가 간단히 구현된다:

```

int EQ::find(int x, int y, int doit)
{
    int i = x, j = y;
    while (dad[i] > 0) i = dad[i];
    while (dad[j] > 0) j = dad[j];
    if (doit && (i != j)) dad[j] = i;
    return (i != j);
}

```

함수 find는 주어진 두 정점이같은 요소 내에 있으면 0을 리턴 한다. 만약 같은 요소에 있지 않고 doit 플래그(flag)가 1이면, 이 두 정점은 같은 요소 내에 놓여진다. 방법은 간단하다. dad배열을 사용하여 각 정점을 포함하는 트리의 루트로 간다. 그리고 나서 루트가 같은지를 확인한다. 만약 다르면, dad[j]=i를 함으로써 두 트리를 합친다.

그림 30.6은 이 과정 동안의 데이터 구조의 내용을 보여준다. 평상시와 마찬가지로, 정점 명들과 1부터 N사이의 정수 값을 대응시켜 주는 index와 name함수가 있다고 가정한다: 각 도표의 엔트리는 해당 dad배열 엔트리의 이름이다. 예로서, EQ클래스의 한 객체를 만들기 위한 선언을(선언문 EQ(eqmax)를 가지고) 한 후, x로 명명된 한 정점과 y로 명명된 한 정

	A	B	C	D	E	F	G	H	I	J	K	L	M
AG							A						
AB		A					A						
AC		A	A				A						
LM		A	A				A						L
JM		A	A				A					J	L
JL		A	A				A					J	L
JK		A	A				A				J	J	L
ED		A	A	E			A				J	J	L
FD		A	A	E	F		A				J	J	L
HI		A	A	E	F		A		H		J	J	L
FE		A	A	E	F		A		H		J	J	L
AF		A	A	E	F	A	A		H		J	J	L
GE		A	A	E	F	A	A		H		J	J	L
GC		A	A	E	F	A	A		H		J	J	L
GH		A	A	E	F	A	A	A	H		J	J	L
JG	J	A	A	E	F	A	A	A	H		J	J	L
LG	J	A	A	E	F	A	A	A	H		J	J	L

그림 30.6 찾기-합하기 데이터 구조

점이 같은 요소 내에 있는지를(그 두 정점간에 선분의 추가 없이) 함수 `eq.find(index(x), index(y), 0)`를 호출함으로써 조사할 수 있다.

이 클래스는 그래프 연결 문제 뿐만 아니라 동치류들이 중요한 역할을 하는 그런 분야에서 사용되어 질 수 있도록 작성 되어 있다. 유일한 필요조건은 집합 원소들이 1에서 V 까지의 인덱스로서 사용될 수 있는 정수명을 가져야 한다는 것이다. 앞서 언급한 것처럼, 아주 앞에서 설명한 다양한 형태의 사전(dictionary) 구현 방식들이 이러한 경우에 사용될 수 있다.

위에서 설명한 알고리즘은 최악 성능(worst-case performance)이 나쁘다. 왜냐하면 형성된 트리들이 퇴행적(degerate)일 수 있기 때문이다. 예를 들면, 선분들을 AB BC CD DE EF FG GH HI IJ...YZ의 순으로 택하면, Y를 가리키는 Z, X를 가리키는 Y등으로 나타나는 긴 고리가 생긴다. 이런 유형의 구조는 그 구조의 형성시에 V^2 에 비례하는 시간이 걸리며 평균적인 동치 조사(average equivalence test)시에는 V 에 비례하는 시간이 걸린다.

이 어려움을 극복하기 위해 여러 가지 방법들이 제안되어 졌다. 한 가지 자연스러운 방법은 두 트리를 합할 때에 임의적으로 `dad[j]=i`로 하는 대신에 “올바른” 처리를 시도해 보는 것이다. `i`에서 루트를 가리키는 한 트리와 `j`에서 루트를 가리키는 한 트리가 합해질 때, 한 노드는 루트가 되고 다른 한 노드는(그리고 그 모든 후손들은) 그 트리의 한 레벨 밑에 위치해야 한다. 대부분의 노드들과 루트와의 거리를 최소화하기 위해, 더 많은 후손들을 가지는 노드를 루트로 택함은 이해가 간다. 가중 균형(weight balancing)으로 불리는 이 방식은 각 루트 노드에 대한 `dad`배열 내에 각 트리의 크기(루트 밑의 후손들의 개수)를 유지함으로써 쉽게 구현된다. 이 경우 크기를 양수가 아닌 값으로 표시함으로써 `find` 프로시저에서 트리를 위로 찾아올라 갈 때 루트 노드의 인식을 가능케 해준다.

이상적으로는, 모든 노드가 그 트리의 루트를 직접 가리키게 하면 좋다. 그러나, 어떤 전략이 사용되더라도 이러한 이상을 달성하려면 적어도 합해질 두 트리 중 한 트리에 있는 모든 노드들을 검사함이 필요하며, 보통 `find` 프로시저가 검사하는, 루트로 가는 경로에 있는, (상대적으로) 아주 소수인 노드들과 비교하여 볼 때 너무 많다. 그러나 우리가 검사하는 모든 노드들이 루트를 가리키게 함으로써 이 이상에 접근할 수 있다! 이 방식은 처음 보기에는 무리인 듯 싶다. 그러나 아주 만들기가 쉽고, 이 트리들의 구조에 변경하면 안되는 어떤 것도 없다: 만약 트리들이 알고리즘의 효율을 높이기 위해 수정 되어 한다면 수정을 해야만 한다. 경로 축약(*path compression*)이라 불리는 이 방법은 루트가 찾아진 후, 각 트리를 다시한번 지나가며 도중에 만나는 각 정점에 해당하는 `dad`배열의 엔트리를 루트를 가리키도록 변경함으로써 쉽게 구현된다.

가중 균형과 경로 축약을 함께 결합시켜 사용하면 알고리즘들이 아주 빠르게 수행되는 것이 보장된다. 다음의 구현 프로그램은 이를 위해 추가된 코드의 비용은 퇴행적 경우들의 경우 의지 비용에 비해 아주 작다.

```
int EQ::find(int x, int y, int doit)
{
    int t, i=x, j=y;
    while (dad[i] > 0) i = dad[i];
    while (dad[j] > 0) j = dad[j];
    while (dad[x] > 0)
        { t = x; x = dad[x]; dad[t] = i; }
    while (dad[y] > 0)
```



```

    { t = y; y = dad[y]; dad[t] = j; }
if (doit && (i != j))
    if (dad[j] < dad[i])
        { dad[j] += dad[i] - 1; dad[i] = j; }
    else
        { dad[i] += dad[j] - 1; dad[j] = i; }
return (i != j);
}

```

그림 30.7은 이 방법을 우리의 예제 데이터에 적용시켰을 때의 첫 여덟 스텝을 보여주며 그림 30.8은 완료 과정을 보여준다. 최종 결과로서 나온 트리의 평균 경로 길이는 $31/13 \approx 2.38$ 로서 그림 30.5의 $38/13 \approx 2.92$ 에 비교해 볼 때 짧다. 처음 다섯 개의 선분들이 추가되었을 경우에는, 그 결과 트리가 그림 30.4같이 된다; 그러나 마지막 세 선분의 경우에는, 가중 구현 규칙으로 인해 J, K, L 그리고 M을 포함하는 “평평한(flat)”트리가 만들어진다. 이 예제에서의 포리스트들은 아주 평평해서 합하기 연산 동작에 관련된 모든 정점들이 루트에 있거나 바로 밑에 있다-경로 축약(더 트리를 평평하게 해 줄 수 있는)은 여기서는 사용하지 않았다. 예로서, 마지막 합하기 연산이 GJ에 대한 것이 아니라 FJ에 대한 것이었다면, 마지막에는 F도 A의 자식이 될 수가 있었을 것이다.

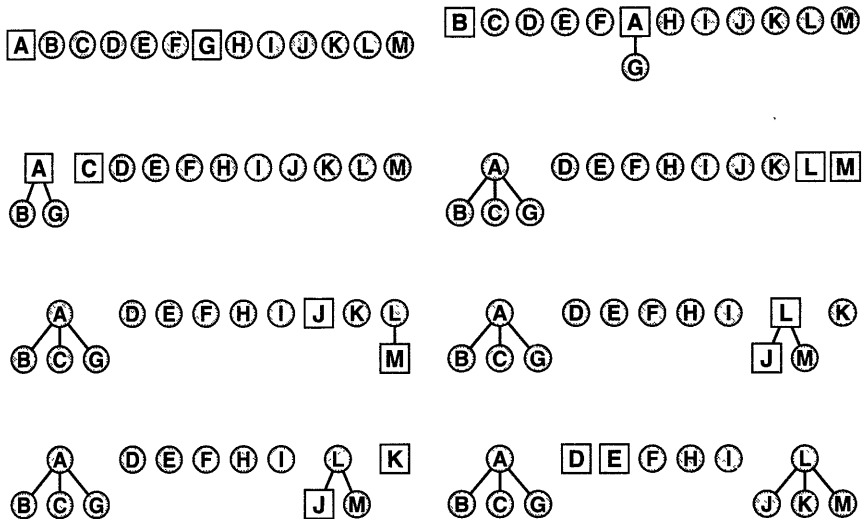


그림 30.7 찾기-합하기의 초기 스텝들(가중 균형과 경로 축약을 결합 사용했을 경우)

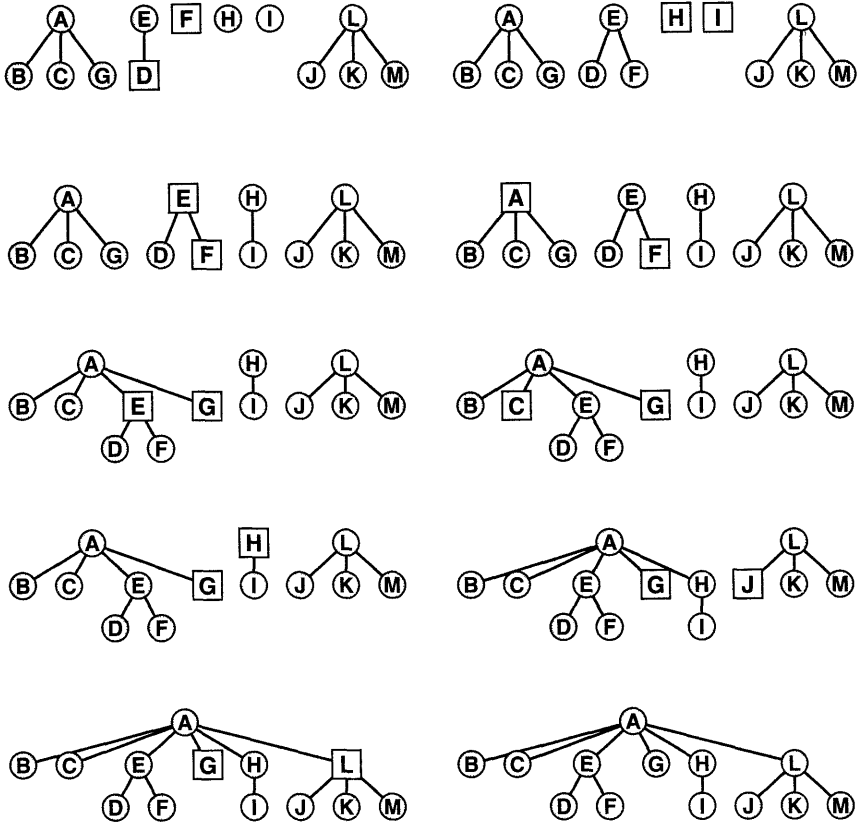


그림 30.8 찾기-합하기의 완료 과정(가중 균형과 경로 축약을 결합 사용했을 경우)

그림 30.9는 이 포리스트가 형성되어 가는 과정의 dad 배열 내용을 보여준다. 이 도표의 명확성을 위해, 각 양수 엔트리 i 는 알파벳(부모명)의 i 번째 문자로 대체했고, 각 음수 엔트리는 양수 값(트리의 가중치)으로 만들기 위해 보수를 취했다.

퇴행적 구조를 피하기 위해 여러 다른 기법들이 개발되어 왔다. 예로서, 경로 축약은 트리를 또한번 통과하며 검사함을 필요로 한다. 반줄임(halving)이라 불리는 또 하나의 기법은 각 노드로 하여금 트리에서 조부모 노드를 가리키게 만든다. 또한 분할(splitting)기법은 반줄임 기법과 같으나, 검색 경로 상의 노드들에 대해 한 노드씩 건너뛰어서 작업을 수행한다. 이들 각각은 가중균형(weight balancing) 또는 높이균형(height balancing)과 결합되어 사용될 수 있다. 높이 균형의 경우, 트리를 어떻게 합할지를 결정할 때 트리의 높이를 사용한다.

그러면 이러한 방법들 중에서 어떻게 하나를 선택할까? 그리고 산출된 트리는 얼마나 “평평”할까? 이 문제에 대한 분석은 아주 어렵다. 왜냐하면 성능은 파라미터 V 와 E 에 의해서만

		A	B	C	D	E	F	G	H	I	J	K	L	M
AG	1							A						
AB	2	A						A						
AC	3	A	A					A						
LM	3	A	A					A					1	L
JM	3	A	A					A			L		2	L
JL	3	A	A					A			L		2	L
JK	3	A	A					A			L	L	3	L
ED	3	A	A	E		1		A			L	L	3	L
FD	3	A	A	E		2	E	A			L	L	3	L
HI	3	A	A	E		2	E	A	1	H	L	L	3	L
FE	3	A	A	E		2	E	A	1	H	L	L	3	L
AF	6	A	A	E	A	E	A		1	H	L	L	3	L
GE	6	A	A	E	A	E	A		1	H	L	L	3	L
GC	6	A	A	E	A	E	A		1	H	L	L	3	L
GH	8	A	A	E	A	E	A	A	H	L	L		3	L
JG	12	A	A	E	A	E	A	A	H	L	L	A	L	
LG	12	A	A	E	A	E	A	A	H	L	L	A	L	

그림 30.9 찾기-합하기 데이터 구조(가중 균형과 경로 축약을 결합 사용했을 경우)

좌우되는 것이 아니라 “찾기”연산의 횟수와(더욱 나쁜 것은) “찾기”연산과 “합하기”연산이 나타나는 순서에도 좌우되기 때문이다. 정렬의 경우(이 경우 실제 파일들이 거의 무작위 적으로 나타난다)와는 달리, 그래프들을 어떻게 모델링해야 할지를 아는 것과 실제로 나타날 수 있는 패턴들을 요구하는 것은 어렵다. 이런 이유로, 보통 최악의 경우에도 잘 동작하는 알고리즘들이 찾기-합하기와(다른 그래프 알고리즘들)의 경우에 선호된다.(비록 지나치게 보수적인 방식이지만)

비록 최악의 경우만 살펴봤으나 찾기-합하기 알고리즘들의 분석은 아주 복잡 미묘하다. 이 점은 그 분석 결과들의 성질이 복잡함을 보아도 알 수가 있다. 그럼에도 불구하고 이 결과들은 실제의 경우 이 알고리즘들이 어떻게 수행될지를 알려 준다.

성질 30.2 만약 가중 균형이나 높이 균형 방식이 축약, 빈줄임 또는 분할 방식과 결합되어 사용되면, E 개의 선분을 사용하여 한 구조를 형성하는데 필요한 연산 동작의 총 수는 거의(그러나 아주 많이는 아닌) 선형적이다.

정확히, 필요한 연산 동작의 수는 $E\alpha(E)$ 에 비례한다. 여기서 $\alpha(E)$ 는, $\lg E$ 에 또 \lg 를 취하고 또 \lg 를 취하고 하는 식으로 16번을 반복해도 그 결과가 1보다 큰 경우가 아니면, $\alpha(E)$ 는 4보다 작게 ($\alpha(E) < 4$) 천천히 증가하는 한 함수이다. 따라서, 각 “찾기”와 “합하기” 동작을 수행하는데 걸리는 시간은 상수라고 가정해도 아주 안전하다. 이 결과는 R. E. Tarjan에 의해 밝혀졌는데, 그는 나중에 이 문제에 대한 어떤 알고리즘도(한 특정한 그러나 일반적인 클래스에서) $E\alpha(E)$ 보다 더 좋게 수행 될 수 없음을 보였다-함수 $E\alpha(E)$ 는 이 문제에 국한된 것이다. \square

찾기-합하기 알고리즘들의 한 중요한 응용 분야는 V 개의 정점과 E 개의 선분을 갖는 한 그래프가 V (그리고 거의 선형 시간)에 비례하여 공간적으로 연결되는지를 알려 주는 것이다. 이것은 어떤 경우들에 있어서는 깊이-우선 검색에 비해 선분들이 저장될 필요가 전혀 없는 장점이 있다. 그러므로, 수천 개의 정점들과 4백만 개의 선분을 갖는 그래프의 연결 여부를 선분들을 한 번에 빨리 조사하여 알 수 있다.

연습문제

1. 우리의 샘플 그래프에 GJ를 제거하고 IK를 추가함으로써 만들어진 그래프의 연결점 (articulation point)들과 이중연결된-요소(biconnected component)들을 찾아라.
2. 문제 1에서 설명한 그래프에 대한 깊이-우선 검색 트리를 그려라.
3. V 개의 정점으로 이중연결된 그래프를 만드는데 필요한 최소 선분 수는?
4. 한 그래프의 이중연결된 요소들을 출력하는 프로그램을 작성하라.
5. 본문의 예제에 대해 형성된 찾기-합하기 포리스트를 그려라.
단, find의 $a[j]=i$ 가 $a[i]=j$ 로 바뀌어 졌다는 가정 하에 그려라.
6. 5번 문제를 경로 축약을 사용했다는 가정 하에 풀어라.
7. 선분 AB BC CD DE EF ... YZ들에 대해 형성된 찾기-합하기 포리스트를 그려라. 처음에는 경로 축약 없이 가중 균형만 이용했을 경우에 대해 그리고, 그 다음에는 가중 균형 없이 경로 축약만 이용했을 경우에 대해 그려라.
8. 7번 문제를 경로 축약과 가중 균형 모두를 이용했다는 가정 하에 풀어라.
9. 본문에서 설명한 찾기-합하기의 변형들을 구현하고, 이들의 성능을 두 인수 값을 1과 100사이의 난수에서 취하는 “합하기” 연산을 1000번 수행하여 비교하라.
10. 1과 V 사이에서 무작위 적으로 정점의 쌍들을 취함으로써 연결하여, V 개의 정점에 대해 무작위적으로 연결된 그래프를 프로그램을 작성하라. 한 연결된 그래프를 만들기 위해 얼마나 많은 선분들이 필요한지를 V 의 함수로서 측정하라.

31 장

가중치 그래프

우리는 종종 가중치나 비용이 각 선분들 상에 부여되어 있는 그래프들을 사용하여 실제적인 문제점들을 모델링 하기를 원한다. 선분이 비행 경로를 나타내는 항공로 지도에서, 이러한 가중치들은 거리나 비행기 요금 등을 나타낼 수 있다. 선분이 전선을 나타내는 전기회로에서는, 전선의 길이나 가격이 자연히 가중치로 사용된다. 작업 스케줄링 표에서, 가중치는 타스크 수행 시간이나 타스크들의 수행 대기 시간 또는 비용들을 나타낼 수 있다.

이런 경우, 어떻게 하면 비용을 최소화할 수 있는가하는 문제들이 자연히 발생한다. 이 장에서는, 이와 같은 문제들에 대한 두 개의 알고리즘을 자세히 살펴보겠다. 하나는 모든 점들을 연결하기 위한 최소 비용 방법을 찾는 것이고 또 하나는 두 개의 점간을 연결하는 최소 비용 경로를 찾는 것이다. 첫째는, 전기회로와 같은 것들을 그래프로 표현할 때 아주 유용한 최소 스패닝 트리(minimum spanning tree) 문제이고 둘째는, 항공로 지도와 같은 것들을 그래프로 표현할 때 아주 유용한 최단 경로(shortest path) 문제이다. 이러한 문제들은 가중치 그래프(weighted graph)에서 발생하는 다양한 문제들을 대표한다.

알고리즘들은 그래프를 통한 검색을 수반한다. 그리고 우리는 가끔 직감적으로 가중치를 거리로 생각한다: 우리는 “ x 에서 가장 가까운 정점” 등과 같이 말한다. 사실, 이러한 선입견은 최단 경로 문제라는 명칭에서 비롯된다. 그럼에도 불구하고, 가중치들이 전혀 거리에 비례할 필요가 없다는 것을 아는 매우 중요하다. 가중치는 시간이나 비용 또는 어떤 완전히 다른 어떤 것을 나타낼 수 있다. 가중치가 실제로 거리를 나타내는데 사용되었을 때, 이에 적절한 다른 알고리즘들이 있을 것이다. 이러한 문제점들은 이 장의 끝에서 보다 자세히 논의된다.

그림 31.1은 가중치 무방향 그래프의 한 샘플이다. 가중치 그래프들을 어떻게 표현해야 하는지는 명백하다: 인접-행렬 표현 시에는 행렬이 부울값들이 아니라 선분의 가중치들을 가지

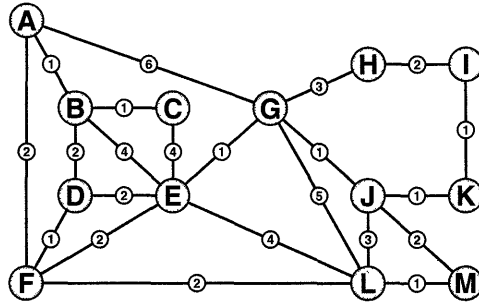


그림 31.1 가중치 무방향 그래프

고, 인접-구조 표현 시에는 선분을 나타내는 한 필드가 각 리스트 요소에 더해진다. 우리는 모든 가중치들이 양수라고 가정한다. 어떤 알고리즘들은 음수 가중치들을 처리할 수 있도록 되어 있다. 이 경우 알고리즘이 상당히 복잡해진다. 다른 경우들에 있어서도 음수 가중치는 문제의 성질을 근본적으로 변화시키고 그래서(여기서 살펴볼 것들보다) 좀더 정교한 알고리즘을 요구한다. 발생할 수 있는 난점들의 한 예로서, 한 사이클을 만드는 선분들의 가중치의 합이 음수인 경우를 생각해 보자: 무한하게 짧은(음수의) 한 경로가 생길 수 있다.

몇몇 “고전적” 알고리즘들이 최소 스패닝 트리와 최단 경로 문제에 대해 개발되어 왔다. 이런 방법들은 가장 잘 알려진 것들 중에 속하며 이 책에서 가장 많이 이용된 알고리즘들이다. 이전에 본 바와 같이, 옛 알고리즘들을 공부할 때, 고전적인 방법들은 일반적인 방식(approach)을 제공한다. 그러나, 최신의 데이터 구조는 간결하고 능률적인 구현 방법들을 제공한다. 이 장에서는, 29장의 일반화된 그래프 운행 방법들이 스파아스(sparse) 그래프들에 대해 최소 스패닝 트리와 최단 경로 문제를 풀기 위해, 어떻게 우선순위 큐를 사용하는지를 알아 볼 것이다; 또한 이것과 덴스(dense) 그래프들에 대한 고전적인 방법들과의 관계를 알아 볼 것이다. 그리고, 전적으로 다른 방식을 사용하는 최소 스패닝 트리 문제에 대한 한 방법을 보기로 한다.

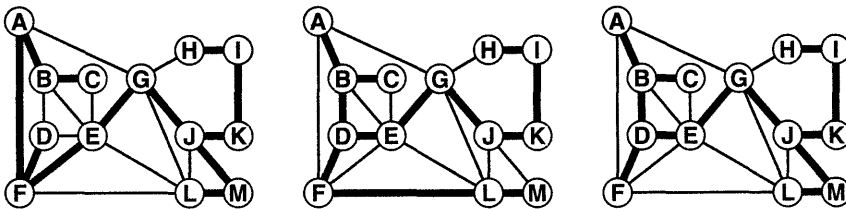


그림 31.2 최소 스패닝 트리들

최소 스패닝 트리(Minimum Spanning Tree)

가중치 그래프의 최소 스패닝 트리는 모든 정점을 연결하는 선분들의 가중치들의 합이 모든 정점을 연결하는 선분들의 다른 집합들에서의 가중치의 합 만큼 작은 선분들의 집합이다. 최소 스패닝 트리는 유일하지는 않다: 그림 31.2는 앞의 샘플 그래프에 대한 세 개의 최소 스패닝 트리를 보여준다. 스패닝 트리의 정의에서 “선분들의 집합”이 한 스패닝 트리를 형성해야만 함을 증명하기는 쉽다: 만약 사이클이 있다면, 그 사이클에 있는 어떤 선분을 삭제해서 모든 정점들을 연결하는 보다 작은 가중치를 갖는 선분의 집합을 만들 수 있다.

많은 그래프 운행 프로시저들이 그래프에 대한 스패닝 트리를 계산해서 만드는 것을 29장에서 보았다. 어떻게 가중치 그래프를 처리해야 만들어진 트리의 가중치의 총 합이 최소가 되게 할 수 있을까? 그렇게 만들기 위한 여러 가지 방법들이 있다. 이들 모두는 다음의 최소 스패닝 트리에 대한 일반적인 성질에 근거한다.

성질 31.1 한 그래프의 정점들이 두 개의 집합으로 분할되어 주었졌을 때, 최소 스패닝 트리는 한 집합 내의 한 정점과 또 하나의 집합 내의 한 정점을 연결하는 선분들 중에서 최소치를 갖는 선분을 포함한다.

예를 들어, 집합 $\{A, B, C, D\}$ 와 $\{E, F, G, H, I, J, K, L, M\}$ 으로 샘플 그래프의 정점들을 나누면 DF가 어느 최소 스패닝 트리에 틀림없이 있게 된다. 이 성질은 모순 명제(contradiction)로서 쉽게 증명된다. 두 개의 집합을 연결하는 가장 짧은 선분이 s 라고 하자, 그리고 s 는 최소 스패닝 트리안에 없다고 가정한다. 그리고나서, 최소 스패닝 트리라고 여겨지는 트리에 s 를 더하여 형성된 그래프를 고려해 보자. 이 그래프는 사이클을 갖는다: 이 사이클 내의 s 를 제외한 어느 선분도 두 집합을 연결해야만 한다. 이 선분을 삭제하고 s 를 추가하면 좀더 짧은 스패닝 트리가 생긴다. 그런데 이것은 s 가 최소 스패닝 트리안에 없다는 가정에 모순된다. \square

이같이, 임의의 정점에서 시작하여, 다음에는 항상 이미 선택된 정점과 가장 가까운 정점을 선택함으로써 최소 스패닝 트리를 만들 수 있다. 다시 말해서, 트리 상에 이미 있는 정점들을 아직 트리상에 없는 정점들과 연결해 주는 그러한 선분들 중에서 최소 가중치를 가진 선분을 찾는다. 그런 후, 해당 선분과 정점을 트리에 추가한다.(가중치가 같은 선분이 있을

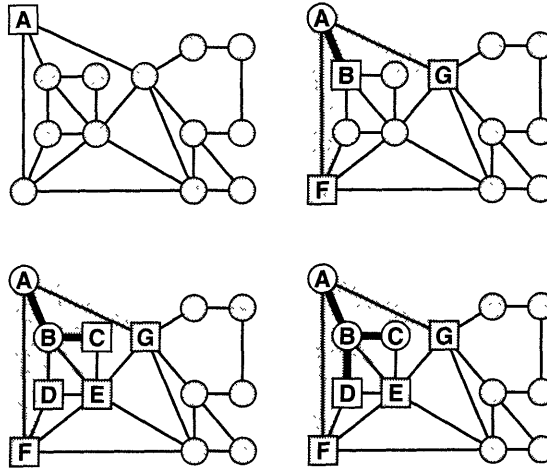


그림 31.3 최소 스패닝 트리를 형성하기 위한 초기 스텝들

경우, 그 중 하나를 택하면 된다) 성질 31.1은 추가된 각 선분이 최소 스패닝 트리의 일부임을 보증한다.

그림 31.3은 이런 전략이 예제 그래프의 노드 A부터 시작하여 적용됐을 때의 첫 네 스텝을 보여준다. 최소 가중치를 갖고 연결된 A에 가장 가까운 정점은 B이다. 그래서 AB는 최소 스패닝 트리안에 있다 A, B와 연결된 모든 선분에서, 선분 BC가 최소 가중치를 가지므로 트리에 추가되고 정점 C가 다음에 연결된다. 이제 A, B, C에 가장 인접한 정점은 D이다. 그래서, BD가 트리에 추가된다. 이런 과정의 완료 결과가 그림 31.5에 있다.

어떻게 이런 전략을 실제로 구현하는가? 지금까지 독자는 29장에서 깊이-우선 검색과 너비-우선 검색 전략들을 특징짓는 트리, 그리고 fringe 상의 정점들 및 미방문된(unvisited) 정점들의 기초 구조를 확실히 인식해 왔다.(스택이나 큐대신) fringe 정점들을 포함하는 우선순위 큐를 사용하여 동일한 방법이 작동됨이 알려져 있다.

우선순위-우선 검색(Priority-First Search)

29장을 다시 보면, 그래프 검색은 정점들을 세 개의 집합으로 나누는 관점에서 설명될 수 있다: 트리 정점들의 집합-여기에 있는 모든 선분들은 이미 검사되었다; fringe 정점들의 집합-이 정점들은 처리를 위해 데이터 구조상에 위치해 있다; 그리고 unseen 정점들의 집합-이 정점들은 처리시 전혀 접촉이 없었다. 우리가 사용하는 기본적인 그래프 검색 방법은

“fringe상의 한 정점(x 라 하자)을 트리로 옮기고, x 에 인접한 unseen정점을 fringe에 넣는다.” fringe에서 어느 정점을 택할 것인가를 결정하기 위한, 우선순위 큐를 사용하는 일반적인 정책을 지칭하여 우선순위-우선 검색이라고 부르기로 한다. 이 방식은 아주 융통성이 크다. 앞으로 보게 되겠지만, 단지 우선순위의 선택에서만 여러 고전적 알고리즘들(깊이-우선 검색 및 너비-우선 검색을 포함한)과 다르다.

최소 스패닝 트리를 계산하기 위해서는, fringe상의 각 정점의 우선순위는 그 정점을 트리에 연결하는 최소 선분 길이를 가진 정점에 부여되어야 한다. 그림 31.4는 그림 31.3과 31.5에서 설명된 트리 형성 과정 동안의 우선순위 큐의 내용을 보여준다. 명확성을 위해, 큐안의 항목들은 정렬된 순서로 보여진다. 이러한 우선순위 큐에 대한 “정렬 리스트” 구현 방식은 작은 그래프에서는 유용할 수 있다. 그러나 큰 그래프에서 모든 연산이 $O(\log N)$ 스텝 내에 완료될 수 있도록 하기 위해서는 힙(heap)을 사용해야 한다.(11장을 보라)

우선, 인접-리스트 표현 방식의 스파이스 그래프를 고려해 보자. 위에서 언급한 것처럼, edge 레코드에 가중치 필드를 추가한다.(그리고, 가중치를 읽을 수 있도록 입력 코드를 수정한다) 그리고나서, fringe에 대해 우선순위 큐를 사용하여, 아래와 같이 구현한다:

```
PQ pq(maxV);
visit (int k) // PFS, adjacency lists
{
    struct node *t;
    if (pq.update(k, -unseen) != 0) dad[k] = 0;
    while (!pq.empty())
    {
        id++; k = pq.remove(); val[k] = -val[k];
        if (val[k] == -unseen) val[k] = 0;
        for (t = adj[k]; t != z; t = t->next)
            if (val[t->v] < 0)
                if (pq.update(t->v, priority))
                {
                    val[t->v] = -(priority);
                    dad[t->v] = k;
                }
    }
}
```

																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																															</
--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

그림 31.4 최소 스패닝 트리 형성 동안의 우선순위 큐의 내용

최소 스패닝 트리를 계산하기 위해, 두 번의 priority를 t->w로 대체해야 한다. 11장에서 설명한 것 같은 우선순위 큐 클래스 사전을 사용한다: update 함수는 쉽게 구현되어 추가되었는데, 이 함수의 목적은 주어진 정점이 적어도 주어진 우선순위를 갖고 큐에 나타나도록 하는 것이다: 만약 그 정점이 큐에 없으면, insert가 행해지고, 만약 그 정점이 큐에 있으나 더 큰 우선순위 값을 가지고 있으면, 우선순위를 바꾸기 위해 change가 사용된다. 만약 어떤 변화가(삽입이나 우선순위 변화 같은) 생기면, update는 0이 아닌 값을 리턴 한다.

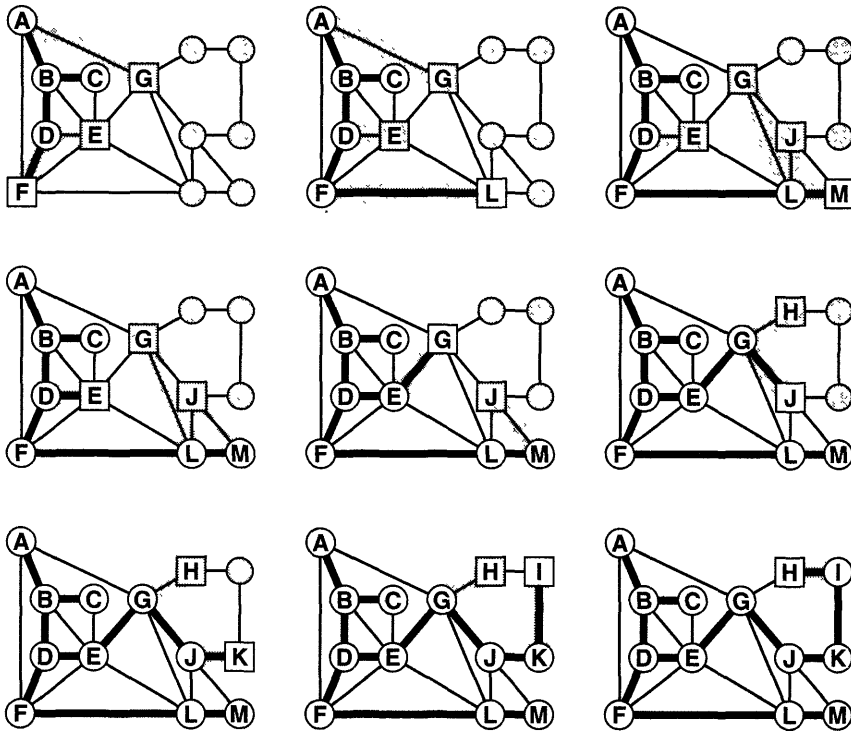


그림 31.5 최소 스패닝 트리 완성 과정

이것은 프로그램이 val과 dad 배열을 현 상태로 유지하게끔 해준다. val 배열 그 자체는 “간접적”으로 우선순위 큐를 가질 수도 있었다; 위 프로그램에서는 명확성을 위해 우선순위 큐를 따로 떼어냈다.

데이터 구조를 우선순위 큐로 바꾼 것과 두 가지 예외 사항을 제외하면, 이 프로그램은 깊이-우선 검색과 너비-우선 검색에서 사용했던 것과 사실상 같은 프로그램이다. 첫 번째 예외 사항은, 이미 fringe상에 있는 한 선분과 만날 때의 처리가 필요하다: 깊이-우선 검색과 너비-우선 검색의 경우에는 이같은 선분들은 무시되어졌다. 하지만, 위 프로그램에서는 새로운 선분이 우선순위를 더 낮게 만드는지 아닌지의 조사가 필요하다. 이렇게 함으로써, 항상 트리에서 가장 가까운 fringe에 있는 정점을 다음에 방문하게 된다. 두 번째 예외 사항은, 이 프로그램은 우선순위-우선 검색 트리에서 각 노드의 부모를 저장하는 dad배열을 가짐으로써 트리를 놓치지 않고 명백히 유지한다.(즉, 노드를 fringe에서 트리로 이동하게 만든 노드의 이름을) 또한, 트리상의 각 노드 k에 대해, val[k]는 k와 dad[k] 간의 선분의 가중치이다. fringe상의 정점은 전처럼 음수 val값들로 표시된다: 표지-정점 unseen은 큰 음수 값을 취하게 된다.(그 이유는 아래에서 명백해 질 것이다)

그림 31.5는 우리 예제에 대한 최소 스패닝 트리의 완성 과정을 보여준다. 평상시와 마찬가지로, fringe상의 정점들은 정사각형 내에 이름을 부여하고, 트리 정점들은 원형 내에 이름을 부여하며, 미방문된 정점들은 이름 없는 원형으로 표시된다. 트리 선분들은 두꺼운 검은선으로 나타내고, fringe상에서 각 정점을 연결하는 최소로 짧은 선분은 음영(그늘)이 지게 표시한다. 그림 31.4와 그림 31.5에서 사용된 트리의 형성 과정을 추적해 오도록 독자에게 권하

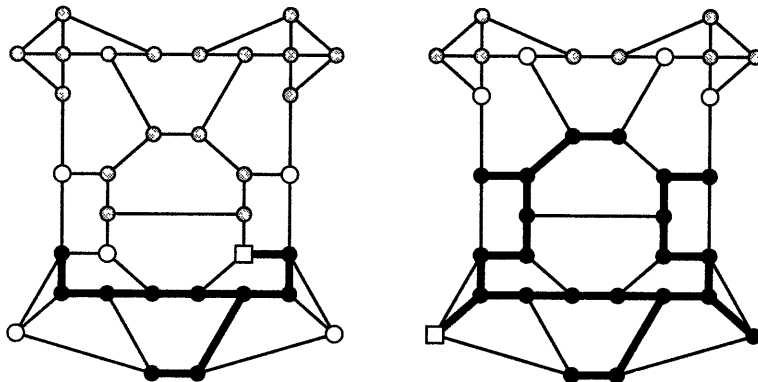


그림 31.6 큰 최소 스패닝 트리의 형성

고 싶다. 특히, 정점 G가 몇 스텝을 거쳐 fringe상에 있다가 어떻게 트리에 추가되는지에 주목하라. 초기에는, G에서 트리까지의 거리는 6이다.(선분 GA 때문에) L이 트리에 추가된 후, GL의 거리는 5로 줄고, 그 다음에 E가 추가되어, 마지막에는 거리가 1이된다. 그리고나서 G가 J에 앞서 트리에 추가된다. 그림 31.6은 선분의 길이를 가중치로서 사용한 큰 “미로” 그래프에 대한 최소 스패닝 트리를 보여준다.

성질 31.2 스파아스 그래프 상에서 우선순위-우선 검색은 $O((E + V) \log V)$ 스텝 내에 최소 스패닝 트리를 계산한다.

성질 31.1을 적용하자: 문제가 되는 노드들의 두 집합은 방문된 노드들의 집합과 미방문된 노드들의 집합이다. 매 스텝에서, 한 방문된 노드에서 한 fringe 노드까지 가는 최소 선분을 선택한다.(방문된 노드에서 unseen 노드까지 가는 선분들은 없다) 이제, 성질 31.1에 의해, 선택된 모든 선분은 최소 스패닝 트리 상에 있다. 우선순위 큐는 단지 정점들 만을 포함한다; 만약 힙을 가지고 구현하면(11장을 보라), 각 연산 동작은 $O(\log V)$ 스텝을 필요로 한다. 각 정점은 삽입을 야기 시키고, 각 선분은 우선순위 변경 연산을 야기 시킨다. \square

이 방법이 적절히 선택된 우선순위를 가지고 최단 경로 문제를 푸는 것을 아래에서 볼 것이다. 또한, 우선순위 큐를 어떻게 구현하면 텐스 그래프에 적당한 V^2 알고리즘이 만들어지는지를 볼 것이다. 이것은 1957년경에 만들어진 오래된 알고리즘과 같다: 최소 스패닝 트리 알고리즘은 일반적으로 R. Prim의 업적이고 최단 경로 알고리즘은 일반적으로 E. Dijkstra의 업적이다. 일관성을 위해, 텐스 그래프에 대한 이러한 각각의 해결책들을 Prim 알고리즘 그리고 Dijkstra의 알고리즘이라고 부르기로 한다. 또한 위의 스파아스 그래프에 대한 방법을 “우선순위-우선 검색” 방법 이라고 부르기로 한다.

우선순위-우선 검색은 너비-우선 검색과 깊이-우선 검색의 일반적 형태이다. 왜냐하면 이들 방법들을 우선순위를 적절히 설정하여 유도할 수 있기 때문이다. 알고리즘 수행 중에 id는 1에서 V 까지 증가하므로 정점들 각각에 유일한 우선순위 값을 부여할 수 있음을 상기하라. 만약 listpfs의 두 개의 priority를 $V-id$ 로 바꾸면, 깊이-우선 검색이 된다. 왜냐하면, 새로 만나는 노드들이 최고의 우선순위를 갖기 때문이다. 만약 priority 대신 id를 사용하면, 너비-우선 검색이 된다. 왜냐하면 오래된 노드들이 최고의 우선순위를 갖기 때문이다. 이러한 우선순위 할당은 우선순위 큐를 스택이나 큐같이 동작하게 만든다.

Kruskal 방법(Kruskal's Method)

최소 스패닝 트리를 찾는 완전히 다른 한 가지 방법은 단순히 매 스텝에서 사이클을 형성하지 않은 최소 선분을 사용하여 한 번에 한 선분씩 추가하는 것이다. 다시 말해서, 이 알고리즘은 N 개의 트리 포리스트를 가지고 시작한다: N 스텝 동안, 이 알고리즘은 한 개의 트리가 만들어질 때까지(가능하면 최단 선분을 사용하여) 두 트리를 연결한다. 이 알고리즘은 1956년경에 J. Kruskal에 의해 만들어졌다.

그림 31.7과 그림 31.8은 우리 예제 그래프 상에서의 이 알고리즘의 동작을 보여준다. 선택된 첫 번째 선분들은 그래프에서 최단 길이(1)를 가진 선분들이다. 그리고 길이 2인 선분에 대해 시도한다; 특히 FE는 고려는 되었지만, 포함되지는 않았음에 주목하라. 그 이유는 FE가 트리에 이미 있는 선분들과 사이클을 형성하기 때문이다.(우선순위-우선 검색과는 대조적으로) 연결이 끊어진 상태의 요소들도 점차 트리에 포함되는데 여기서, 트리는 한 번에 “한 선분”씩 추가되어 커진다.

Kruskal 알고리즘의 구현은 이미 배운 프로그램들을 가지고 할 수 있다. 첫째로, 한 번에 하나씩 가중치의 오름차순으로 선분을 고려할 필요가 있다. 한 가지 방법은 단순히 이들을 정렬시키는 것이다. 그러나 우선순위 큐를 사용하는 것이 더 좋다는 것이 입증되었다. 그 주된 이유는 모든 선분들을 검사할 필요가 없기 때문이다. 문제는 좀 더 세부적으로 아래에서 논

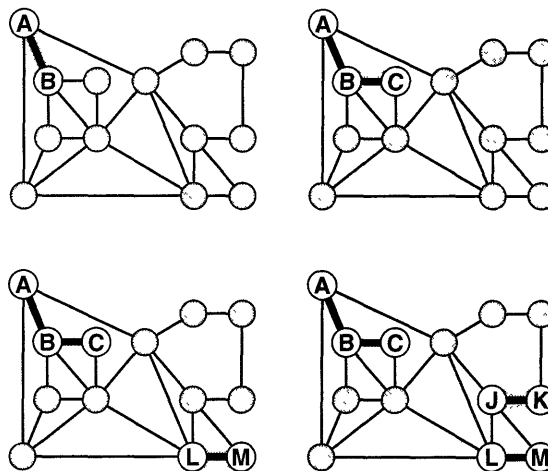


그림 31.7 Kruskal 알고리즘의 초기 스텝들

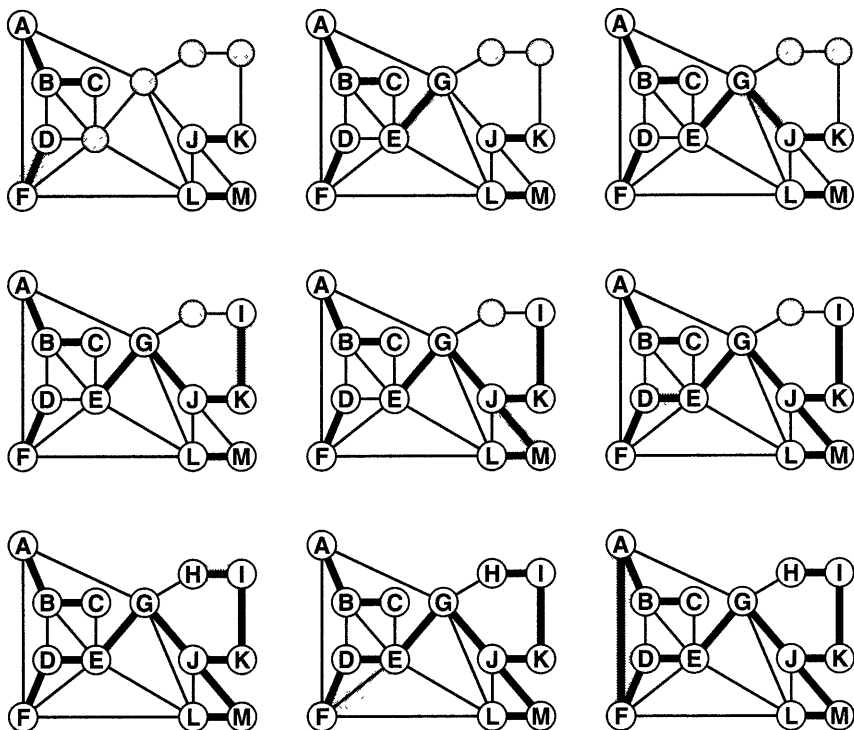


그림 31.8 Kruskal 알고리즘의 완료

의된다. 둘째로, 만약 한 주어진 선분이 이제까지 선택된 선분들에 더해질 때 사이클이 형성되는지의 여부를 시험할 필요가 있다. 전장에서 논의된(정점) 찾기-합하기 구조는 바로 이를 위해 설계된 것이다.

그래프에 대한 적절한 데이터 구조는 각 선분에 대해 한 엔트리를 갖는 단순한 배열 e 이다. 이 구조는 깊이-우선 검색이나 더 단순한 프로시저와 그래프의 인접-리스트나 인접-행렬 표현을 가지고 쉽게 만들어질 수 있다. 그러나, 아래 프로그램에서는 이 배열을 직접 입력 받은 값으로 채운다. 11장에 있는 우선순위 큐 사전 클래스 PQ 는 배열 e 내의 가중치 필드 값들을 우선순위 값으로 가지고 사용됐고 30장의 `find` 프로시저에 근거한 한 동격의 클래스 EQ 는 사이클 여부의 시험에 사용된다. 아래 프로그램은 스패닝 트리에 있는 각 선분에 대해 `edgefound` 프로시저를 호출한다; 약간 더 수정 작업을 하면, `dad` 배열이나 다른 표현 방식들에 대해서도 연산 가능해질 수 있다.

```

void kruskal()
{
    int i, m, V, E;
    struct edge
        { char v1, v2; int w; };
    struct edge e[maxE];
    PQ pq(maxE); EQ eq(maxE);
    cin >> V >> E;
    for (i = 1; i <= E; i++)
        cin >> e[i].v1 >> e[i].v2 >> e[i].w;
    for (i = 1; i <= E; i++)
        pq.insert (i, e[i].w);
    for (i = 0; i < V-1; )
    {
        if (pq.empty()) break; else m = pq.remove();
        if (eq.find(index(e[m].v1), index(e[m].v2), 1))
            { cout << e[m].v1 << e[m].v2 << ' '; i++; };
    }
}

```

이 과정이 종료되는 두 가지 방법이 있음을 주목하라. 만약 $V - 1$ 개의 선분들을 발견하면 트리가 만들어지고 종료된다. 만약 우선순위 큐가 먼저 비게되면, 모든 선분이 검사됐으나 스패닝 트리가 찾아지지 못한 채로 종료된다; 그래프가 연결되어 있지 않은 경우, 이런 일이 발생한다. 이 프로그램의 실행 시간은 우선순위 큐에 있는 선분들을 처리하는데 소모한 시간에 의해 좌우된다.

성질 31.3 Kruskal 알고리즘은 $O(E \log E)$ 스텝내에 그래프의 최소 스패닝 트리를 계산한다.

이 알고리즘의 정확성은 또한 성질 31.1으로 부터 알 수 있다. 문제의 두(정점들의) 집합들은 트리에서 선택된 선분에 연결된 정점들의 집합과 아직 연결되지 않은 정점들의 집합이다. 더해진 선분은 이 두 집합들에 있는 선분들 간의 최단 선분이다. 최악의 경우는 그래프가 연결 안된 상태인데, 이 경우 모든 선분들이 검사되어야 한다. 연결된 그래프에서조차도, 최악의 경우는 마찬가지이다. 왜냐하면 그래프가 두 개의 정점들의 집단(cluster)으로 이루어져 있고 각 집단 내의 정점들은 모두 매우 짧은 선분들로 연결되어 있으나, 이 두 집단을 연결하

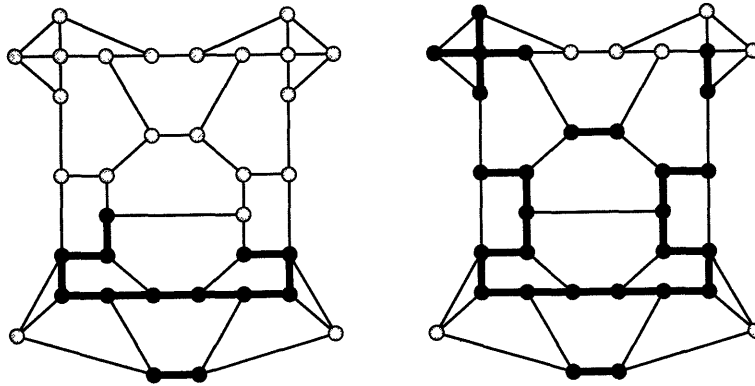


그림 31.9 Kruskal 알고리즘을 이용한 큰 최소 스패닝 트리 찾기

는 선분만 매우 길 수가 있기 때문이다. 이런 경우, 그래프에서 가장 긴 선분이 최소 스패닝 트리에 있을 때, 이 선분은 우선순위 큐의 가장 끝에 있을 것이다. 통상적인 그래프들의 경우, 그래프 내의 최장 선분이 검사되기 한참 전에 스패닝 트리가 잘 완료될 것을 기대해 볼 수도 있다.(스패닝 트리는 단지 $V - 1$ 개의 선분을 갖는다) 그러나, 초기에 우선순위 큐를 구성하는데 걸리는 시간은 항상 E 에 비례한다.(성질 11.2를 보라) \square

그림 31.9는 Kruskal 알고리즘으로 큰 최소 스패닝 트리를 형성하는 것을 보여준다. 이 다 이어그램은 어떻게 알고리즘이 모든 짧은 선분들을 먼저 고르는지를 확실하게 보여준다. 이 방법은 긴(대각선의) 선분들은 나중에 추가할 것이다.

우선순위 큐를 사용하는 것보다, 처음에 가중치로 선분을 간단하게 정렬할 수도 있다. 그리고 나서, 그들을 정렬된 순서대로 처리한다. 또한, 사이클 조사를 찾기-합하기 방법보다 훨씬 더 간단한 방법을 가지고, $E \log E$ 에 비례하는 시간 내에 행할 수 있다. 따라서 항상 $E \log E$ 스텝이 걸리는 최소 스패닝 트리 알고리즘이 구해진다. 이것이 Kruskal이 제안한 방법이다. 그러나, 우리는 위에 있는 우선순위 큐들과 찾기-합하기 구조들을 이용하여 만들어진 보다 현대화된 알고리즘 버전을 “Kruskal 알고리즘”이라 부르기로 한다.

최단 경로(Shortest Path)

최단 경로(Shortest Path) 문제는 한 가중치 그래프에서 주어진 두 정점 x 와 y 를 연결하는 경로 상의 모든 선분들의 가중치의 합이 최소인 성질을 갖는 경로를 찾는 것이다.

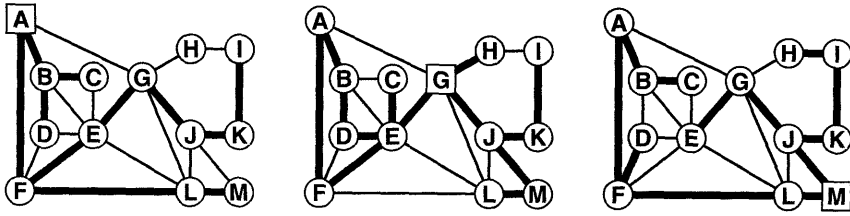


그림 31.10 세 개의 최단 경로 스패닝 트리

만약 가중치가 모두 1이더라도, 문제는 아직 재미가 있다: 이 문제는 x 와 y 를 연결하는 선분들의 최소 개수를 갖는 경로를 찾는 것이다. 게다가, 우리는 이미 그 문제를 푸는 방법인 너비-우선 검색 알고리즘에 대해 배웠다. x 의 너비-우선 검색 시작은 하나의 선분으로 x 에 연결될 수 있는 모든 정점을 우선 방문하고, 그리고 나서 두 개의 선분으로 x 에 연결될 수 있는 모든 정점을 방문하고, 이런식으로 계속하여, k 개의 선분으로 x 에 연결될 수 있는 모든 정점을 $k + 1$ 개의 선분으로 x 에 연결될 수 있는 정점들에 앞서 방문함을, 귀납법으로 쉽게 증명할 수 있다. 이같이 하여 한 정점 y 에 도달했을 때, 이것이 x 에서부터의 최단 경로인 것이다. 왜냐하면 y 까지 도달하는 더 이상 짧은 경로가 없기 때문이다.

일반적으로, x 에서 y 까지의 경로는 모든 정점들을 포함할 수 있다. 그래서 우리는 보통 주어진 한 정점 x 를 그래프 내의 모든 다른 정점들과 연결하는 최단 경로를 찾는 문제를 고려한다. 이 문제 또한 앞서 설명한 우선순위-우선 그래프-유행 알고리즘으로 쉽게 풀 수 있다.

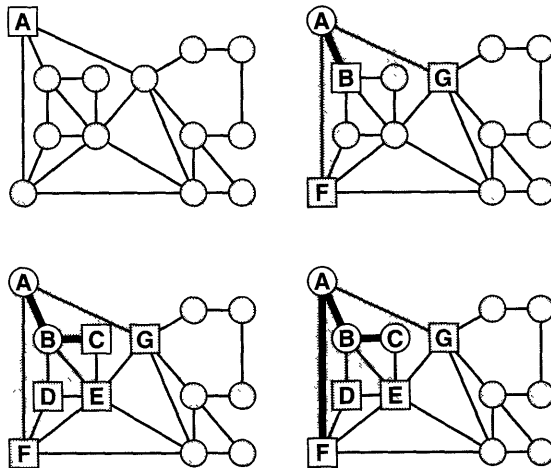


그림 31.11 최단 경로 트리 형성을 위한 초기 스텝들

만약 그래프 내의 한 정점 x 에서 모든 다른 정점으로 가는 최단 경로를 그린다면, 거기에는 분명히 사이클이 생기지 않으며, 스패닝 트리가 형성된다. 각기 다른 정점에서 이같은 일을 시작하면, 각기 다른 스패닝 트리가 만들어진다. 예를 들어, 그림 31.10은 우리가 이제까지 사용한 예제 그래프에서 정점 A, G와 M을 가지고 각각 만들어진 세 개의 최단 경로 스패닝 트리를 보여준다.

이 문제에 대한 우선순위-우선 검색의 해결 방법은 최소 스패닝 트리에 대한 해결 방법과 사실상 동일하다: 한 정점 x 에 대한 트리는 매 스텝마다 fringe상의 x 에 가장 가까운 정점을 트리에 추가하여 만들어진다.(tree에 가장 가까운 정점을 더하기에 앞서) fringe 상의 어느 정점이 x 에 가장 가까운지를 알기 위해, val배열이 사용된다: 각 트리의 정점 k 에 대해, $val[k]$ 는 k 에서 x 로 가는 최단 경로의 거리이다.(물론 이 최단 경로는 트리상의 노드들로 구성된다) k 가 트리에 더해질 때, k 의 인접 리스트를 찾아봄으로써 fringe를 변경한다. 이 리스트 상의 각 노드 t 에 대해, $t \rightarrow v$ 로 부터 k 를 경유하는 x 까지의 최단거리는 $val[k] + t$

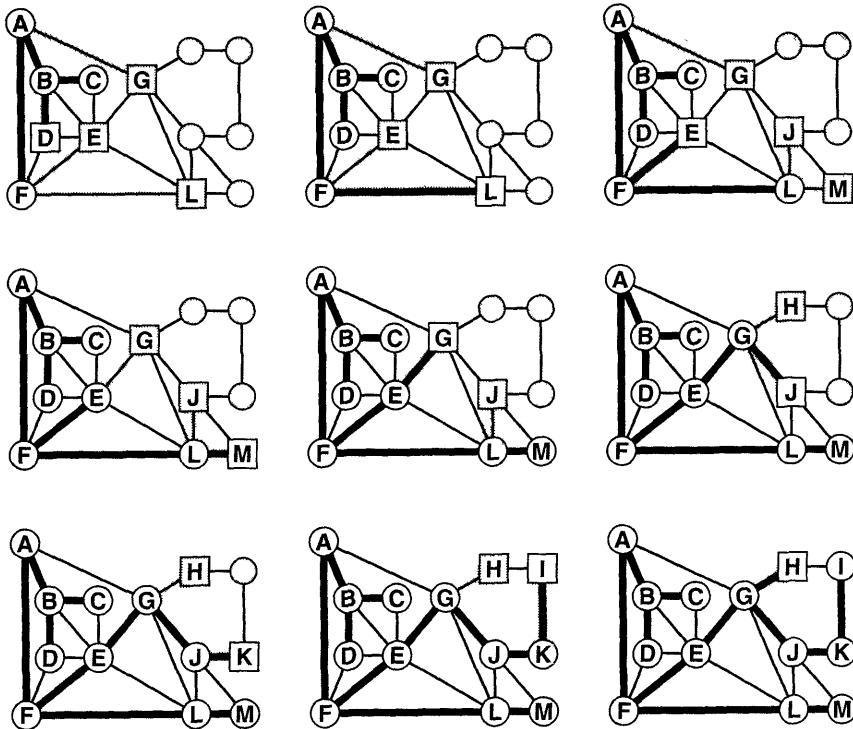


그림 31.12 최단 경로 트리 형성의 완성 과정

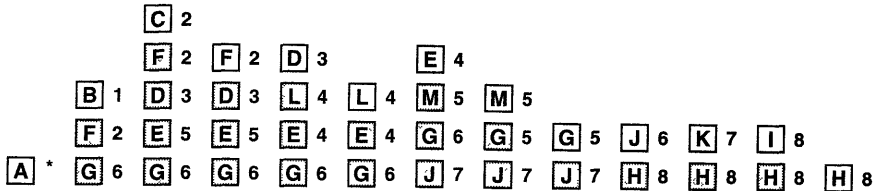


그림 31.13 최단 경로 트리 구성 과정 동안의 우선순위 큐의 내용

->v이다. 이같이, 우선순위-우선 그래프-운행 프로그램의 priority에 이 값을 사용하여 알고리즘을 간단히 구현한다.

그림 31.11은 우리의 예제에서 정점 A에서 최단 스패닝 트리를 형성할 때의 처음 네 단계를 보여준다. 우선, A에서 가장 가까운 정점인 B를 방문한다. 그리고 C와 F 모두가 A로부터의 거리가 2이므로 이들을 다음 번에 방문한다.(우선순위 큐는 어떤 순서로 이들을 리턴 하는데, 여기서는 C 그리고 나서 F가 리턴된 것으로 하자) 이 완성 과정은 그림 31.12에 있고, 이 검색 과정 동안의 우선순위 큐의 내용 변화는 그림 31.13에 있다.

다음으로, A까지의 거리 3의 경로를 얻기 위해, D가 F나 B에 연결될 수 있다.(B가 F 전에 트리에 넣어졌으므로, 이 알고리즘은 D를 B에 연결한다; 이같이 F가 트리에 넣어졌을 때 D는 이미 fringe 상에 있었으므로 F는 A로 가는 더 짧은 경로를 제공하지는 않는다) 그리고 나서, L E M G J K I 및 H가 A에서부터의 그들의 최단 거리의 오름차순으로 트리에 더해진다. 예를 들어, H는 A에서 가장 먼 노드이다: 경로 AFEGH는 총 가중치가 8이다. H로 가는 그리고 A로부터 모든 다른 노드로 가는, 더 이상의 짧은 경로는 없다.

그림 31.14는 우리 예제에 대한 dad와 val 배열의 최종 값들을 보여준다. A에서 H까지의 최단 경로의 총 가중치는 8이고(H에 대한 엔트리인 val[8]에서 찾아진다) 그 경로는 A에서, F, E, G를 거쳐 H로 가는 경로이다.(H에서 시작하여 dad 배열을 역으로 거슬러 추적하면 찾아진다) 이 프로그램은, 우리가 listpfs에서 했던 것과 마찬가지로, val 엔트리의 루트 값이 0이 되게 해야함에 주목하라.

k	1	2	3	4	5	6	7	8	9	10	11	12	13
name (dad[k])	A	B	B	F	A	E	G	K	G	I	F	L	
val[k]	0	1	2	3	4	2	5	8	8	6	7	4	5

그림 31.14 최단 경로 스패닝 트리 표현

성질 31.4 스파이스 그래프에서 우선순위-우선 검색은 $O((E+V)\log V)$ 스텝 내에 최단 경로 트리를 계산한다.

이 알고리즘의 정확성은 성질 31.1의 경우와 매우 유사한 방법으로 증명될 수 있다. 12장에서와 마찬가지로 우선순위 큐의 구현시 힙(heap)들을 이용함으로써, 언급된 시간 경계치 내에 우선순위-우선 검색이 되어지는 것을 항상 보증할 수 있다.(어떤 우선순위 규칙들이 사용되더라도) □

아래에서, 우리는 어떻게 우선순위 큐를 다르게 구현하여 텐스 그래프에 적합한 V^2 알고리즘을 만드느지를 볼 것이다. 최단 경로 문제의 경우, 1959년 경에 만들어진-일반적으로 E. Dijkstra의 업적으로 간주되는 한 방법으로 축약된다.

그림 31.15는 한 큰 최단 경로 트리를 보여준다. 전처럼, 선분들의 길이를 이 그래프에서의 가중치로 사용했다. 그래서, 이 해결책은 맨 밑의 좌측 노드에서 모든 다른 노드로 가는 최소 길이의 경로를 찾는 것에 해당한다. 나중에, 이같은 그래프들에 적합할 수 있는 한 구현법에 대해 논의하기로 한다. 그러나, 이런 그래프에서 조차도, 다른 값들을 가중치로서 사용하는 것이 적합할 수도 있다: 예로서, 그래프가 하나의 미로 그래프라면(29장을 보라), 한 선분의 가중치는 미로 그 자체에 대한 거리를 나타내는 것이지, 그래프 상에 그려진(표시된) 지름길들에 대한 거리를 나타내는 것이 아니다.

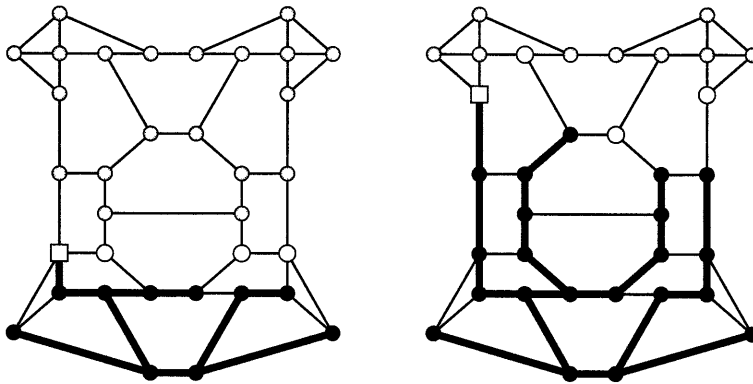


그림 31.15 한 큰 최단 경로 트리의 형성

덴스 그래프에서의 최소 스패닝 트리와 최단 경로 (Minimum Spanning Tree and Shortest Paths in Dense Graphs)

인접-행렬로 나타내어진 그래프의 경우, 우선순위-우선 그래프 운행 알고리즘으로 V^2 의 실행 시간을 얻기 위해서는, 우선순위 큐에 대해 무순서 배열 표현 방식을 사용하는 것이 최고이다. 이것은 우선순위들의 변경을 위한 루프와 최소치를 찾기 위한 루프를 결합함으로써 가능하다: 매번 fringe에서 한 정점을 꺼낼 때마다, 모든 정점들을 조사하고, 필요하다면 그들의 우선순위를 변경하고, 그리고 찾아진 최소치를 보관하고 있어야 한다. 이렇게 함으로써, 덴스 그래프들의 경우 우선순위-우선 검색시(그리고 최소 스패닝 트리 및 최단 경로 문제들의 경우에) 선형 알고리즘이 제공된다.

특히, 여기서는 val 배열에서 우선순위 큐를 유지한다.(위에서 논의한 바와 같이, listpfs에서도 똑같이 행해질 수 있다) 그러나, 힙을 사용하는 대신 우선순위 큐에 대한 동작(연산)들을 직접 구현한다. 전과 같이, val 엔트리의 부호(음수, 양수)는 해당 정점이 트리상에 있는지 우선순위 큐상에 있는지를 알려준다. 모든 정점들은 우선순위 큐에서 시작하고 표지 우선순위 값 unseen을 갖는다. 한 정점의 우선순위를 바꾸려면, 그 정점에 해당하는 val 엔트리에 새로운 우선순위 값을 할당하면 된다. 최고 우선순위를 가진 정점을 제거하기 위해서는, val 배열을 스캔하여 최대 음수 값(0에 가까운)을 갖는 정점을 찾아서, 그 값의 보수 값으로 해당 val 엔트리를 채운다. 이제까지 우리가 사용해 온 listpfs 프로그램을 이런 식으로 바꾸면, 다음과 같은 간결한 프로그램이 만들어진다:

```
void search() // PFS, adjacency matrix
{
    int k, t, min = 0;
    for (k = 1; k <= V; k++)
        { val[k] = unseen; dad[k] = 0; }
    val[0] = unseen-1;
    for (k = 1; k != 0; k = min, min = 0)
        {
            val[k] = -val[k];
            if (val[k] == -unseen) val[k] = 0;
            for (t = 1; t <= V; t++)
                if (val[t] < 0)
                    {
```

```

        if (a[k][t] && (val[t] < -priority))
            { val[t] = -priority; dad[t] = k; }
        if (val[t] > val[min]) min = t;
    }
}
}

```

-unseen보다 큰 값을 갖는 정점은 최소치를 찾기 위한 한 표지 정점으로서 사용되는데, 이 값의 음수 표현이 반드시 가능해야 한다.

만약 인접-행렬에 가중치들을 저장하고 이 프로그램에서 priority로서 $a[k][t]$ 를 사용하면, 최소 스패닝 트리를 찾는 Prim 알고리즘이 된다; 만약 priority로서 $val[k] + a[k][t]$ 를 사용하면, 최단 경로를 찾는 Dijkstra 알고리즘이 된다. 전과 같이, 이제까지 검색된 정점의 수를 나타내는 id를 유지하기 위한 프로그램 코드를 포함시키고, 우선순위에 대해 $v-id$ 를 사용하면, 깊이-우선 검색 프로그램이 얻어진다. 이 프로그램은 우리가 이제까지 스패닝 그래프들에 대해 행해 온 우선순위-우선 검색 프로그램과 비교해 볼 때, 단지 그래프 표현 방식(인접-리스트 대신 인접-행렬)과 우선순위 큐의 구현 방식(간접적 힙 대신에 무순위 배열)을 사용했다는 점만이 다르다.

성질 31.5 최소 스패닝 트리와 최단 경로 문제는 텐스 그래프에서 선형 시간 내에 해결될 수 있다.

최악 실행 시간이 V^2 에 비례함은 위 프로그램을 살펴보면 즉각적으로 알 수 있다. 각 정점이 방문될 때마다. 인접-행렬의 한 행에서 V 개의 엔트리들을 통과하며 모든 인접 선분들을 검사하고 우선순위 큐의 값들을 변경하고 다음번 최소치를 찾는다. 따라서, 수행 시간은 E 가 V^2 에 비례할 때 선형 시간이 걸린다. □

이제까지 아주 다른 수행 성능 특성들을 갖는 최소 스패닝 트리 문제에 대한 세 개의 프로그램들에 대해(제일 먼저, 우선순위-우선 검색 방법, 그리고 Kruskal 알고리즘 그리고 마지막으로 Prim 알고리즘에 대해) 논의했다. 어떤 그래프의 경우에는, Prim 알고리즘이 셋 중에서 제일 빠른 듯 싶고, 또 다른 그래프의 경우에는 Kruskal 알고리즘이, 또 다른 그래프의 경우에는, 우선순위-우선 검색 알고리즘이 제일 빠른 듯 싶다. 앞서 설명한 것처럼, 우선순위-우선 검색 방법은 최악의 경우에 $(E + V)\log V$ 인 반면, Prim의 경우는 V^2 이고 Kruskal의

경우는 $E \log E$ 이다. 그러나, 최악의 경우에 근거하여 알고리즘을 선택함은 현명한 방법이 아니다. 왜냐하면 최악의 경우는 실제로는 잘 발생하지 않기 때문이다. 사실상, 우선순위-우선 검색과 Kruskal 방법은 모두 실제로 존재하는 그래프들에 대해 E 에 비례하는 수행 시간을 갖는다. 그 첫 번째 이유는 대부분의 선분들이 $\log V$ 스텝이 걸리는 우선순위 큐의 조정을 필요로 하지 않기 때문이고, 그 두 번째 이유는 최소 스패닝 트리에서 가장 긴 선분은 아마도 별로 많지 않은 선분들이 우선순위 큐에서 제거되게 할 정도로 짧기 때문이다. 스파아스 그래프들에 대해서는, 우선순위-우선 검색이 가장 빠를 듯하다. 왜냐하면, 이 방법은 조그만 우선순위 큐를 가질 확률이 높기 때문이다. 물론, Prim 방법은 텐스 그래프들에 대해 약 E 에 비례하는 수행 시간을 갖는다.(그러나 이 방법은 스파아스 그래프들에 대해서는 사용되서는 안된다)

기하학적 문제(Geometric Problems)

평면에 N 개의 점이 주어지고 그 점들을 모두 연결하는 최단 선분들의 집합을 찾으려 한다 하자. 이 문제는 Euclid의 최소 스패닝 트리 문제라고 불리어지는데, 앞에서 설명한 그래프 알고리즘을 이용하여 풀 수 있다. 그러나 기하학은 보다 더 효율적인 알고리즘들의 개발이 가능하도록 충분한 여분의 구조를 제공한다.

앞서 설명한 알고리즘을 이용하여 Euclid 문제를 푸는 방법은 N 개의 정점들과 $N(N-1)/2$ 개의 선분들을 갖는 완전 그래프(complete graph)를 만드는 것이다; 한 선분이 두 정점을 연결하는데, 이 두 정점간의 거리가 가중치가 된다. 최소 스패닝 트리는, Prim 알고리즘을 사용할 경우 N^2 에 비례하는 시간 내에 찾아질 수 있다.

더 좋은 알고리즘을 만들 수 있음이 증명되어있다. 그 요점은 기하학적 구조가 완전 그래프의 대부분의 선분들을 문제와 상관없이 만든다는데 있다. 따라서, 최소 스패닝 트리 형성의 시작전에 그들을 제거할 수 있다. 사실상, 최소 스패닝 트리는 Voronoi 다이어그램의 쌍대(dual)를 구성하는 선분들만을 취함으로써 유도된 한 그래프의 부분집합임이 증명되었다.(28장을 보라) 우리는 이 그래프가 N 에 비례하는 많은 수의 선분들을 가지고 있고, Kruskal 알고리즘과 우선순위-우선 검색 방법 모두가 이와 같은 스파아스 그래프들에 대해 잘 동작함을 알고 있다. 그러면, 원리상으로는, ($N \log N$ 에 비례하는 시간이 걸리는) Voronoi의 쌍대를 얻을 수 있고, $N \log N$ 에 비례하는 시간이 걸리는 Euclid의 최소 스패닝 트리를 얻기 위해 Kruskal 알고리즘이나 우선순위-우선 검색 방법을 수행시킬 수 있다. 그러나, Voronoi 쌍대

를 얻기 위한 프로그램을 작성하는 것은 아주 전문적인 프로그래머라 해도 아주 어렵다. 따라서, 이 방식은 구간 검색을 위해 26장에서 사용된 격자법에서의 경우처럼 실행 불가능한 것으로서 인정되는 듯 싶다.

또 다른 한 가지 방법은, 무작위 점 집합들의 경우 사용될 수 있는 것인데, 구간 검색을 위해 26장에서 사용된 격자법에서의 경우처럼, 그래프 내의 선분 수를 제한하기 위해 점들의 분포를 이용하는 것이다. 만약 우리가 평면을 정사각형들로 분할하고(각 정사각형이 약 $\log N/2$ 개의 점이 있을 높은 확률을 갖도록) 정사각형 내의 각 점이 그 이웃하는 정사각형들 내의 점들과 연결되는 선분들만을 그래프 내에 포함되도록 하면,(비록 보장은 못하지만) 최소 스패닝 트리에 있는 모든 선분들을 얻을 수 있는 확률이 높다. 이 말은 kruskal 알고리즘이나 우선순위-우선 검색 방법이 그 수행 작업을 일찍 끝낼 수 있음을 의미한다.

앞 절들에서 제기된 문제에 의해 나타난 그래프와 기하학적 알고리즘들 간의 관계를 다시 생각해 봄직하다. 많은 문제들이 기하학적이거나 그래프 적인 문제들로 귀착되어 나타남은 사실이다. 만약 객체들의 실제적인 물리적 위치가 주된 특성이라면, 24~28장에서 다룬 기하학적 알고리즘들이 적당할 것이다. 그러나 만약 객체들의 상호 연결이 근본적인 중요성을 지니면, 이 장에서 다룬 그래프 알고리즘들의 사용이 더 바람직하다. Euclid의 최소 스패닝 트리 문제는 이 두 방식의 인터페이스(접합부)에 있는 듯(입력은 기하학과 연관되고 출력은 상호 연결과 연관되어 있다)하며, 이에 대한 그리고 이에 관련된 문제들에 대한 단순하고 쉬운 방법들을 개발하는 것은 어려운 목표로서 남아 있다.

기하학적 알고리즘들과 그래프 알고리즘들이 상호 연관이 있게 되는 또 하나의 분야는, 정점들이 평면상의 점들이고 선분들은 그 정점들을 연결하는 선들인 그래프에서 정점 x 에서 y 로가는, 최단 경로를 찾는 문제이다. 이제까지 사용해 온 미로 그래프가 그러한 종류의 그래프로서 간주 될 수 있다. 이 문제에 대한 해결책은 간단하다: 먼저, 우선순위-우선 검색을 이용하고, 만나게 되는 각 fringe 정점의 우선순위를 x 에서 그 fringe 정점까지의 거리로서 설정하고(주어진 알고리즘에서와 마찬가지로) 각 fringe 정점에서 y 까지의 거리를 Euclid 거리로서 설정한다. 그런 후, y 가 트리에 더해질 때 알고리즘을 끝낸다. 이 방법은 x 에서 y 로가는 최단 경로를 항상 y 로 향해 진행함으로써 매우 빨리 찾는다. 반면에 표준(형) 그래프 알고리즘은 y 에 대해 “검색”을 시도해야 한다. 한 큰 미로 그래프의 한 쪽 구석에서 다른 쪽 구석으로 감은 \sqrt{V} 에 비례하는 많은 노드들을 검사함을 필요로 할 수 있다. 반면에 표준 알고리즘의 경우에는 거의 모든 노드들이 검사돼야 한다.

연습문제

1. 이 장의 시작부의 예제 그래프에 대해 또 하나의 최소 스패닝 트리를 구하라.
2. 한 연결된 그래프의 최소 스패닝 포리스트를 찾기 위한 알고리즘을 작성하라.(각 정점은 임의의 한 선분에 의해 접속(연결)되어야 하지만, 그 결과 그래프는 연결될 필요는 없다)
3. 최소 스패닝 트리 문제에 대한 우선순위-우선 처리 방식의 해결책이 $(E + V)\log V$ 에 비례하는 시간을 필요로 하는, V 개의 정점들과 E 개의 선분들을 가진, 그래프가 있는가? 그 예를 들거나 답이라고 생각하는 바를 서술하라.
4. 우선순위 큐를 일반 그래프-운행 구현 시의 정렬된-리스트로서 유지한다고 하자. 최악 수행 시간은 어느 정도일까? 상수(일정) 비율 내에 있을까? 만약 있다면, 언제 이 방법이 적절할까?
5. 다음의 “그리디(greedy)” 방법이 왜 최단 경로 혹은 최소 스패닝 트리 문제 모두에 동작하지 않는가를 보여주는 예를 설명하라. 이 방법의 각 스텝에서는, 방금 방문했던 것에 가장 인접한 미방문된 정점이 방문된다.
6. 이 장의 예제 그래프에서 다른 노드들에 대해 최단 경로 트리 들을 형성하라.
7. 아주 큰 그래프에서 최소 스패닝 트리를 어떻게 찾을 것인지를 설명하라.(주기억장치에 한 번에 넣을 수 없게 크다)
8. V 개의 정점을 갖는 무작위로 연결된 그래프를 생성하는 프로그램을 작성하라. 그리고 나서 임의의 한 정점에서의 최소 스패닝 트리와 최단 경로 트리를 찾아라. 1에서 V 까지의 무작위 수를 가중치로 사용하라. V 값들이 다를 경우, 트리들의 가중치들은 어떻게 비교하나?
9. V 개의 정점을 갖는 무작위적 완전 그래프들을 단순히 1과 V 사이의 무작위적 숫자(난수)들을 인접-행렬 내에 채움으로써 생성하는 프로그램을 작성하라. 또한 이 방법, Prim 방법, 그리고 Kruskal 방법 중 어느 것이 더 빨리 최소 스패닝 트리를 찾는지를 알기 위해 $V = 10, 25, 100$ 인 경우에 대해 경험에 따라 시험하라.

10. 왜 다음의 Euclid 최소 스패닝트리를 찾는 방법이 동작되지 않는가를 보여주는 예를 설명하라. 이 방법에서는, “ x 좌표계 상의 점들을 정렬한 다음, 첫 반에 해당하는 부분의 점들에 대해 최소 스패닝 트리들을 찾고 나머지 반에 해당하는 최소 스패닝 트리들을 찾는다. 그리고 나서, 이 두 부분을 연결하는 최소 선분을 찾는다.”

방향성 그래프는 노드들을 연결하는 선분들이 모두 한 방향인 그래프이다; 이 부가된 구조는 다양한 성질들을 결정하는 것을 더 어렵게 만든다. 이같은 그래프들의 처리는 많은 편도들이 있는 도시를 여행하는 것이나 왕복 항공 노선이 거의 없는 나라에서 여행하는 것과 유사하다. 이런 경우들에 있어서, 한 지점에서 다른 지점으로 가는 것은 정말로 하나의 난제이다.

종종, 선분의 방향은 모델링된 응용 분야에서 일종의 선행 관계를 나타낸다. 예로서, 방향성 그래프는 제조 작업 라인의 모델로서 사용될 수도 있다: 노드들은 수행되어야 할 작업들에 해당하고, 만약 노드 x 에 해당하는 작업이 노드 y 에 해당하는 작업보다 먼저 수행되어야 한다면, 노드 x 에서 y 로 가는 선분이 존재한다. 어떻게 이런 작업들의 수행 순서를 결정하여 작업들간의 선행(precedence) 관계들을 위배함이 없게 하는가?

이 장에서는, 방향성 그래프에 대한 깊이-우선 검색,(연결성 정보를 요약하는) 추이적 폐포(transitive closure) 계산을 위한 알고리즘들, 그리고 위상학적 정렬(topological sorting) 및 선행 관계들과 관련이 있는 강하게 연결된-요소(strongly connected component)들에 대한 알고리즘들을 다룬다.

29장에서 언급한 것처럼, 방향성 그래프에 대한 표현 방식은 무방향 그래프에 대한 표현 방식을 단순히 확장한 것(실제로는 제약을 가한 것)이다. 인접-리스트 표현에서는, 각 선분은 단지 한번에 나타난다: x 에서 y 까지의 선분은 x 에 해당하는 연결 리스트에서 y 를 갖는 한 리스트 노드로서 표현된다. 인접-행렬 표현에서 만약 x 에서 y 로 가는 선분이 있다면, 행 x 와 열 y 에 해당하는 엔트리가 1로 표시된 V 행- V 열 정방 행렬을 유지할 필요가 있다.

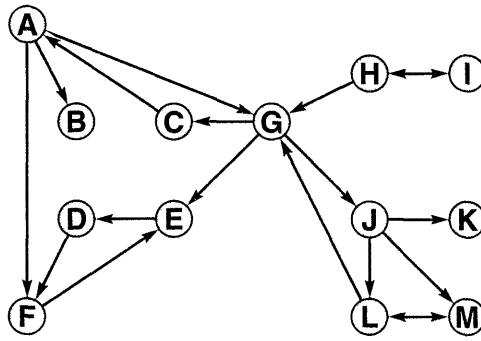


그림 32.1 한 방향성 그래프

우리가 지금까지 고려해 온 무방향 그래프와 유사한 한 방향성 그래프가 그림 32.1에 있다. 이 그래프는 선분 AG AB CA LM JM JL JK ED DF HI FE AF GE GC HG GJ LG IH ML 로 구성된다. 선분들을 구분할 때 나타나는 정점들의 순서가 방향성 그래프에서는 중요하다: 표기 AG는 A에서 G로 가는 선분을 나타내며 G에서 A로 가는 선분을 나타내지는 않는다. 물론 두 노드간에 양방향으로 하나씩 두 개의 선분을 갖는 것은 가능하다.(그림 32.1의 경우, HI와 IH 그리고 LM과 ML의 두 개의 양방향 선분들이 있다)

무방향 그래프와 이 그래프에 있는 각 선분에 대해 두개의 서로 반대 방향의 선분을 갖는 방향성 그래프의 표현간에는 어떤 의미의 차이도 없다.(즉, 동일한 의미를 지닌다) 이처럼, 이 장의 일부 알고리즘들은 전 장들에서 본 알고리즘들의 일반화된 형태로서 간주될 수 있다.

깊이-우선 검색(Depth-First Search)

29장의 깊이-우선 검색 알고리즘은 주어진 방향성 그래프들에 대해 올바르게 동작한다. 사실 상, 이 알고리즘의 동작은 무방향 그래프들에 대한 경우보다 약간 더 간단하다. 왜냐하면, 그래프의 두 노드간에(서로 양방향인) 두 선분이 명백히 존재하지 않는 한, 두 선분에 대해 고려할 필요가 없기 때문이다.

그러나 검색 트리는 다소 복잡한 구조를 갖는다. 예를 들어, 그림 32.2는 이 장의 샘플 그래프(그림 32.1)에 대해 29장의 있는 재귀적 알고리즘을 설명해 주는 깊이-우선 검색 구조를 보여준다. 전과 마찬가지로, 이것은 다시 그려진 예제 그래프의 한 버전이다: 실선 선분들은 재귀적 호출들을 통해 정점들을 방문하는데 실제로 사용된 선분들이고, 점선 선분들은 그 선

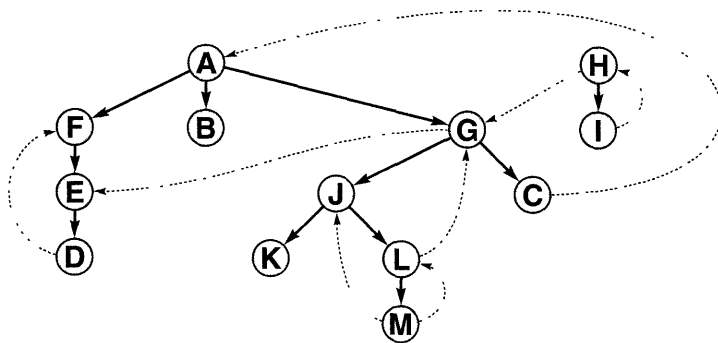


그림 32.2 한 방향성 그래프에 대한 깊이-우선 검색 포리스트

분이 고려될 때 이미 방문된 정점들을 가리키기(나타내기)위한 선분들이다. 노드들은 A F E D B G J K L M C H I의 순으로 방문된다.

선분들의 방향은 방향성 그래프들에 대한 깊이-우선 검색 포리스트들을 무방향 그래프에 대한 깊이-우선 검색 포리스트들과 아주 다르게 만든다. 예를 들어, 원래의 그래프가 연결되어 있었을 지라도, 점선 선분에 의해 정의된 깊이-우선 검색 구조에서는 연결 되지 않을 수가 있다: 왜냐하면 이 구조는 포리스트이지 트리가 아니기 때문이다.

무방향 그래프들의 경우, 트리에서 한 조상 정점과 연결된 한 정점을 나타내는 단 한 종류의 점선 선분이 있었다. 방향성 그래프들의 경우에는, 세 가지 종류의 점선 선분들이 있다: 첫째는 한 정점에서 트리상의 한 조상 정점을 가리키는 *up* 선분이고, 둘째는 한 정점에서 트리상의 한 후손 정점을 가리키는 *down* 선분이며, 마지막으로 한 정점에서 트리상의 조상 정점도, 후손 정점도 아닌 또 하나의 정점을 가리키는 *cross* 선분이 있다.

무방향 그래프에서 처럼, 방향성 그래프의 결합 성질은 흥미롭다. 우리는 “정점 x 에서 y 로 가는 방향성 경로(directed path)(즉, 지정된 방향에 있는 선분들만을 따라가는 경로)가 있는가?”, “정점 x 에서 한 방향성 경로를 따라가면 어떤 정점들에 도달하는가?” 그리고 “정점 x 에서 y 로 가는 방향성 경로와 정점 y 에서 x 로 가는 방향성 경로가 있는가?” 같은 질문들에 대해 답하고 싶다. 무방향 그래프들의 경우와 똑같이, 비록 여러 종류의 점선 선분들이 알고리즘을 약간 복잡하게는 만들겠지만, 기본적인 깊이-우선 검색 알고리즘을 적절히 수정함으로써 위와 같은 질문들에 대해 답 할 수 있다.

추이적 폐포(Transitive Closure)

무방향 그래프에서, 단순 연결(simple connectivity)은 한 주어진 정점에서 그래프의 선분들을 따라 운행함으로써 도달할 수 있는 정점들을 알려준다: 즉 이 정점들 모두는 동일한 연결된 요소(component)내에 있다. 이와 유사하게, 방향성 그래프들의 경우에는, 한 주어진 정점에서 지정된 방향으로 그래프 상의 선분들을 운행함으로써 도달할 수 있는 정점들의 집합에 대해 관심을 갖게 된다. 방향성 그래프들에 대한 문제들은 단순 연결의 경우보다 좀 더 복잡해진다.

29장의 깊이-우선 검색의 재귀적 프로시저인 `visit`가 시작 노드에서 도달할 수 있는 모든 노드들을 방문함을 증명하는 것은 쉽다. 따라서, 이 프로시저에 방문하는 모든 노드들을 출력하는 코드를 삽입(즉, `cout<<name(k)`를 삽입)한다면, 시작 노드에서 도달할 수 있는 모든 노드들이 출력 될 것이다. 그러나, 깊이-우선 검색 포리스트 내의 각 트리는 그 트리의 루트에서 도달할 수 있는 모든 노드들을 반드시 포함하지는 않는다는 사실을 주의 깊게 주목하라: 여기 예제에서는, 그래프 상의 모든 노드들이 H에서 부터는 도달 가능하나 I에서 부터는 그렇지 않다. 각 노드에서 부터 방문 될 수 있는 모든 노드들을 얻으려면, `visit` 프로시저를 V 번 호출하면 된다.(즉, 각 노드당 한번씩):

```
for ( k = 1; k <= V; k++)
{
    id = 0;
    for (j = 1; j <= V; j++) val[j] = 0;
    visit(k);
    cout << "\n";
}
```

이 프로그램은 그림 32.1의 방향성 그래프에 대해 아래와 같은 출력을 한다. 전과 같이, 각 줄의 정점 이름들의 순서는 특정 그래프 표현 방식과 사용된 검색 프로시저에 따른 산물이다. 각 줄에 있는 노드들의 집합은 그래프 자체의 구조적 성질이다: 각 줄은 그 줄의 첫 번째 노드에서 방향성 경로를 경유하여 도달할 수 있는 그러한 노드들을 표시하고 있다.

A	F	E	D	B	G	J	K	L	M	C		
B												
C	A	F	E	D	B	G	J	K	L	M		
D	F	E										
E	D	F										
F	E	D										
G	J	K	L	M	C	A	F	E	D	B		
H	G	J	K	L	M	C	A	F	E	D	B	I
I	H	G	J	K	L	M	C	A	F	E	D	B
J	K	L	G	C	A	F	E	D	B	M		
K												
L	G	J	K	M	C	A	F	E	D	B		
M	L	G	J	K	C	A	F	E	D	B		

무방향 그래프들의 경우, 이러한 계산은 한 연결된-요소(component)내의 노드들에 해당하는 각 줄이 그 요소 내의 모든 노드들을 나열하는 그러한 성질을 갖는 한 도표를 만들어 낼 수도 있을 것이다. 위의 도표도 그와 유사한 성질을 지닌다: 특정 줄들은 동일한(노드들의) 집합을 나열하고 있다.(노드 나열 순서를 무시할 때) 아래에서는, 이러한 성질을 설명해 주는 연결성의 일반화에 대해 알아보기로 한다.

평소와 같이, 단지 도표만 출력 하는 것보다는 좀 더 다른 처리를 실행할 수 있도록 프로그램 코드를 추가할 수도 있다. 우리가 더 추가하려 하는 한 가지 연산은, 만약 x 에서 y 로 가는 어떤 길이 있다면, x 에서 y 로 가는 선분을 직접 추가하는 것이다. 이런 성질을 갖는 모든 선분들을 추가함으로써 만들어진 그래프를 그 그래프의 추이적 폐포(transitive closure)라 rh 부른다. 보통은 많은 선분들이 추가되며 추이적 폐포는 텐스(dense)할 확률이 높으므로, 인접-행렬 표현 방식이 바람직하다. 이것은 무방향 그래프에서 연결된-요소들에 대한 경우와 유사하다; 한번 이런 계산을 수행해 놓으면, “ x 에서 y 로 가는 방법이 있는가?” 같은 질문들에 대해 쉽게 답할 수 있다.

성질 32.1 깊이-우선 검색은 방향성 그래프의 추이적 폐포를 스파이스 그래프에 대해서는 $O(V(E + V))$ 스텝들 내에 그리고 텐스 그래프에 대해서는 $O(V^3)$ 스텝들 내에, 계산하는데 이용될 수 있다.

이 성질은 29장의 기본 성질에서 바로 얻어진다: 그래프 상의 V 개의 정점들 각각에 대해 깊이-우선 검색 프로시저를 수행한다. 너비-우선 검색의 경우에도 동일한 결과가 나온다: 앞서 언급한 것처럼, 노드들이 방문되는 순서는 이 문제와 특별한 관련이 없다. □

인접-행렬로서 표현된 한 그래프의 추이적 폐포를 계산하는 아주 간단한 비재귀적 프로그램이 있다:

```
for (y = 1; y <= V; y++)
  for (x = 1; x <= V; x++)
    if (a[x][y])
      for (j = 1; j <= V; j++)
        if (a[y][j]) a[x][j] = 1;
```

S. Warshall은 이 방법을 1962년에 고안해 냈다. “만약 노드 x에서 y로 가는 길이 있고, 노드 y에서 j로 가는 길이 있다면, 노드 x에서 j로 가는 길이 있다”라는 단순한 관측을 사용하여, 이 방법의 묘법은 이 관측된 사실을 좀 더 강하게 만드는 것이다. 그래서, 행렬을 통한 한 번에 계산이 행해지게 하는 것이다: 즉, “만약 y보다 작은 인덱스 값들을 가진 노드들만을 사용하여 노드 x에서 y로 가는 길이 있고, 노드 y에서 j로 가는 길이 있다면, y+1보다 작은 인덱스 값들을 가지는 노드들만을 이용하여 노드 x에서 j로 가는 길이 있다”. 위 프로그램은 이의 직접적인 구현이다.

Warshall 방법은 한 그래프에 대한 인접-행렬을 그 추이적 폐포에 대한 인접-행렬로 변환한다. 이 알고리즘을 따라가는 한 가지 방법은 이 알고리즘이 한번에 행렬의 한 행 전부의 값들을 채운다고 간주하는 것이다. y번째 열의 처리는 y열에서 1인 각 행을 y열 그 자신과 y행의 값을 “or” 한 값으로 대체시키는 것이다. 그림 32.3은 우리의 샘플 그래프에 대한 초기 행렬과 첫 두 열과 셋째 열의 반을 처리한 후의 상태를 보여준다: 여기까지는 단지 C 행만이

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	0	0	0	1	1	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	0	1	0	0	0	0	0	0	0
E	0	0	0	1	1	0	0	0	0	0	0	0	0
F	0	0	0	0	1	1	0	0	0	0	0	0	0
G	0	0	1	0	1	0	1	0	0	1	0	0	0
H	0	0	0	0	0	0	1	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	0	0	0	0	0	0	1	0	0	0	0	1	1
M	0	0	0	0	0	0	0	0	0	0	0	1	1

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	0	0	0	1	1	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	0	0	1	1	0	0	0	0	0	0
D	0	0	0	1	0	1	0	0	0	0	0	0	0
E	0	0	0	1	1	0	0	0	0	0	0	0	0
F	0	0	0	0	1	1	0	0	0	0	0	0	0
G	0	0	1	0	1	0	1	0	0	1	0	0	0
H	0	0	0	0	0	0	1	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	0	0	0	0	0	0	1	0	0	0	0	1	1
M	0	0	0	0	0	0	0	0	0	0	0	1	1

그림 32.3 warshall 알고리즘의 초기 단계들

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	1	1	1	1	0	0	1	1	1	1
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	1	1	1	1	0	0	1	1	1	1
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	0	0	0	0	0	0	0
F	0	0	0	1	1	1	0	0	0	0	0	0	0
G	1	1	1	1	1	1	1	0	0	1	1	1	1
H	1	1	1	1	1	1	1	1	1	1	1	1	1
I	1	1	1	1	1	1	1	1	1	1	1	1	1
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	1	1	1	1	1	1	1	0	0	1	1	1	1
M	0	0	0	0	0	0	0	0	0	0	0	1	1

그림 32.4 Warshall 알고리즘의 최종 단계들

영향을 받는다. 그림 32.4는 마지막 두 열 반이 남은 경우의 행렬의 상태와 최종 처리 결과(추이적 폐포)를 보여준다.

성질 32.2 Warshall 알고리즘은 $O(V^3)$ 스텝 내에 추이적 폐포를 찾는다.

만약 행렬이 초기에 1로 채워져 있다면, 이 성질은 프로그램 내부의 3개의 중첩 루프(nested loop)로 인해 그냥 봐도 명백해진다. 또한, 스파이스 그래프라 하더라도 이 경우로 축약 될 수 있다. 예를 들어, 만약 첫번째 노드가 모든 다른 노드에 연결되어 있다면, 행렬은 y 가 2가 되기 전에 1로 모두 채워질 것이다. □

매우 큰 그래프들의 경우에는, 이 계산을 체계적으로 구성하여 비트들에 대한 연산이 한 번에 한 개의 컴퓨터 워드(word)단위로서 행해질 수 있다. 이렇게 하면 많은 경우에, 아주 큰 시간 절약이 된다.

모든 최단 경로들(All Shortest Paths)

한 가중치 없는 방향성이나 무방향 그래프의 추이적 폐포는 모든 정점 쌍 x, y 에 대해 “ x 에서 y 로 가는 경로가 있는가?”라는 질문에 답한다. 가중치 있는 방향성이나 무방향 그래프들의 경우에는, 모든 정점 쌍들에 대해 x 에서 y 로 가는 최단 경로를 찾게 해주는 도표를 만드는 것이 좋을 것이다. 이것은 모든 쌍에 대한 최단 경로 문제(all-pairs shortest-path

problem)이다. 예로서, 그림 32.5에 있는 가중치 그래프에서, 우리는 도표를 액세스 함으로써 M에서 K까지의 최단 경로는 길이 8을 갖고 J에서 F까지의 최단 경로는 길이 12를 갖는다는 등의 정보를 알기를 원한다.

위에서와 마찬가지로, 전 장에 있는 최단 경로 알고리즘은 시작 정점에서 모든 다른 정점들로 가는 최단 경로를 찾는다. 그러므로 단지 프로시저를 각 정점에서 시작하여 V 번 수행함이 필요하다. 따라서, 알고리즘은 $O((E + V)V \log V)$ 스텝 내에 수행되게 된다. 그러나 Warshall의 방법같은 것을 사용하는 것 또한 가능한데, 이는 일반적으로 R. W. Floyd의 업적으로 간주된다:

```
for (y = 1; y <= V; y++)
  for (x = 1; x <= V; x++)
    if (a[x][y])
      for (j = 1; j <= V; j++)
        if (a[y][j] > 0)
          if (!a[x][j] || (a[x][y] + a[y][j] < a[x][j]))
            a[x][j] = a[x][y] + a[y][j];
```

이 알고리즘의 구조는 정확하게 Warshall의 방법과 같다. 경로들을 계속 따라가기 위해 “or”를 사용하는 대신에, 선분이 새로운 짧은 경로에 속하는지를 알기위해 각 선분에 대해 약간의 계산을 더 할 필요가 있다: “ $y + 1$ 보다 작은 인덱스 값들을 갖는 노드들만 사용하여 노드 x 에서 y 로 가는 최단거리는 y 보다 작은 인덱스 값들을 갖는 노드들만을 사용하여 노드 x 에서 j 로 가는 최단거리이거나, 만약 더 짧다면, 노드 x 에서 y 로 그리고 y 에서 j 로 가는 거

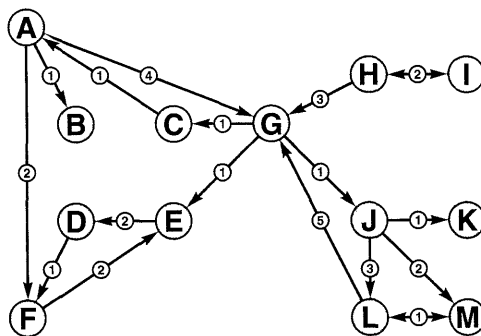


그림 32.5 한 가중치 방향성 그래프

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	0	0	0	2	4	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	1	0	0	0	0	0	0	0
E	0	0	0	2	0	0	0	0	0	0	0	0	0
F	0	0	0	0	2	0	0	0	0	0	0	0	0
G	0	0	1	0	1	0	0	0	0	1	0	0	0
H	0	0	0	0	0	0	3	0	1	0	0	0	0
I	0	0	0	0	0	0	0	1	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	3	2	
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	5	0	0	0	0	0	1
M	0	0	0	0	0	0	0	0	0	0	0	1	0

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	0	0	0	2	4	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	2	0	0	0	3	5	0	0	0	0	0	0
D	0	0	0	0	0	1	0	0	0	0	0	0	0
E	0	0	0	2	0	0	0	0	0	0	0	0	0
F	0	0	0	0	2	0	0	0	0	0	0	0	0
G	0	0	1	0	1	0	0	0	0	1	0	0	0
H	0	0	0	0	0	0	3	0	1	0	0	0	0
I	0	0	0	0	0	0	0	1	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	3	2	
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	5	0	0	0	0	0	1
M	0	0	0	0	0	0	0	0	0	0	0	1	0

그림 32.6 Floyd 알고리즘의 초기 단계들

리를 더한 것이 최단거리이다.” 다른 때와 마찬가지로, 행렬의 엔트리 값이 0임은 해당 선분이 없음을 뜻한다; 따라서 이 프로그램은 모든 값에 대한 비교를 없앴으로써(무한히 큰 가중치를 갖는 선분에 해당하는 한 표지값(sentinel value)을 사용하여) 다소 간단해 질 수 있다.

성질 32.3 Floyd의 알고리즘은 $O(V^3)$ 스텝 내에 모든 쌍에 대한 최단 경로 문제를 해결한다.

이 성질은 성질 32.1에서와 같은 방법으로 추론하여 나온다. □

그림 32.6과 32.7은 우리의 예제에 대한 Floyd 알고리즘의 성능을 더 자세히 보여 주는데, 그림 32.3과 32.4와의 비교를 위해 같은 방식으로 배열을 했다. 이 행렬들에서 0 엔트리들은

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	6	1	5	6	4	2	4	0	0	5	6	8	7
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	2	6	7	5	3	5	0	0	6	7	9	8
D	0	0	0	5	3	1	0	0	0	0	0	0	0
E	0	0	0	2	5	3	0	0	0	0	0	0	0
F	0	0	0	4	2	5	0	0	0	0	0	0	0
G	2	3	1	3	1	4	6	0	0	1	2	4	3
H	5	6	4	6	4	7	3	2	1	4	5	7	6
I	6	7	5	7	5	8	4	1	2	5	6	8	7
J	0	0	0	0	0	0	0	0	0	1	3	2	
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	7	8	6	8	6	9	5	0	0	6	7	9	1
M	0	0	0	0	0	0	0	0	0	0	0	1	0

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	6	1	5	6	4	2	4	0	0	5	6	8	7
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	2	6	7	5	3	5	0	0	6	7	9	8
D	0	0	0	5	3	1	0	0	0	0	0	0	0
E	0	0	0	2	5	3	0	0	0	0	0	0	0
F	0	0	0	4	2	5	0	0	0	0	0	0	0
G	2	3	1	3	1	4	6	0	0	1	2	4	3
H	5	6	4	6	4	7	3	2	1	4	5	7	6
I	6	7	5	7	5	8	4	1	2	5	6	8	7
J	10	11	9	11	9	12	8	0	0	9	1	3	2
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	7	8	6	8	6	9	5	0	0	6	7	9	1
M	8	9	7	9	7	10	6	0	0	7	8	1	2

그림 32.7 Floyd 알고리즘의 최종 단계들

두 인덱스 정점간에 경로가 없음을 보여 주는데, 이는 Warshall 알고리즘의 경우와 동일하다. Warshall 알고리즘에서는 0이 아닌 엔트리들은 두 정점들 간에 경로가 있음을 나타내나, Floyd 알고리즘의 경우에는, 그 값들은 현재까지 발견된 최단 경로의 길이이다. 실제적인 최단 경로는 또한 전 장들에서 본 dad 배열의 한 행렬 버전을 사용하여 계산될 수 있다: 행이 x 이고 열이 j 인 엔트리를 x 에서 j 로 가는 최단 경로 상의 바로 앞에 있는 정점의 이름(위 프로그램 코드의 내부 루프에서의 정점 y)으로 채워라.

위상 정렬(Topological sorting)

순환 그래프들은 방향성 그래프들에 연관된 많은 분야에서 발생한다. 그러나, 만약 그림 32.1의 그래프가 제조 작업 라인을 모델링한 것이라면, 이 그래프는 단순히 작업 A는 작업 G 전에 수행 되어야 하고, 작업 G는 작업 C 전에 수행 되어야 하며, 작업 C는 작업 A 전에 수행 되어야 함을 의미할 수도 있다. 그러나 이와 같은 경우 일관성이 없다: 이러한 그리고 많은 다른 응용 분야들의 경우에, 방향성 사이클(모든 선분들의 방향이같은 사이클)들이 없는 방향성 그래프들이 선호된다. 이와 같은 그래프들을 방향성 비순환 그래프(directed acyclic graph)들이라 하고, 약어로 dags로 표기한다. Dags는 선분 방향이 고려 안될 경우 많은 사이클들을 가질 수 있다; 이 dags를 정의하는 성질은 지정된 방향에 있는 선분들이 사이클을 결코 형성하지 않는다는 것이다. 그림 32.8은 그림 32.1의 방향성 그래프와 유사한 한 dag를 보여주고 있다; 물론 사이클을 제거하기 위해 일부 선분들이 제거되거나 그 방향이 변경되었다. 이 그래프에 대한 선분 리스트는 30장에 있는 연결된 그래프의 경우와 동일하다; 그러나, 선분이 지정될 때 주어지는 정점들의 순서에는 차이가 있다.

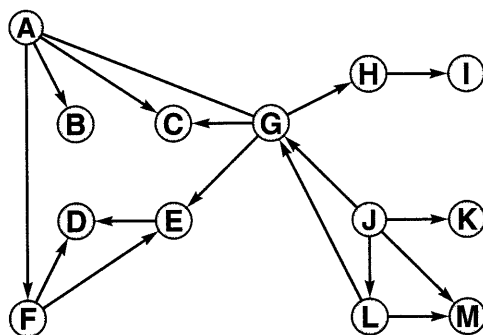


그림 32.8 한 방향성 비순환 그래프

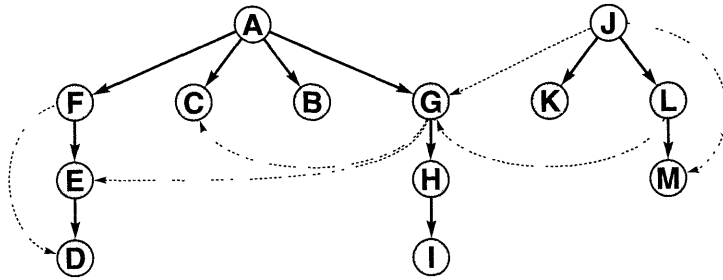


그림 32.9 한 dag에 대한 깊이-우선 검색

Dags는 일반 방향성 그래프들과는 아주 다르다: 어떤 의미에서, 이들은 어느 정도는 트리이고 또 어느 정도는 그래프이다. 이들의 처리시, 우리는 이들의 특수한 구조를 확실히 이용할 수 있다. 한 임의의 정점에서 보면, 한 dag는 트리 같이 보인다; 다시 말해서, 한 dag에 대한 깊이-우선 검색 포리스트는 up선분들을 갖지 않는다. 그림 32.9는 그림 32.8의 dag에 대한 깊이-우선 검색 포리스트를 보여준다.

Dags에 대한 한 기본 동작은 어느 정점도 그 정점을 가리키는 정점에 앞서 처리되지 않는 순서로서 그래프 상의 정점들을 처리하는 것이다. 예를 들어, 위 그래프 상의 노드들은 다음과 같은 순서로 처리 될 수 있다:

J K L M A G H I F E D B C

만약 이런 정점 순서 상에서 선분들을 다시 그리면, 선분들은 모두 왼쪽에서 오른쪽으로 진행할 수도 있다. 앞서 언급한 것처럼, 이에 대한 명백한, 예를 들면, 제조 작업 과정들을 나타내는 그래프들의 경우와 같은, 응용 분야가 있다. 왜냐하면, 이 방식은 그래프에 표시된 제약 조건들 내에서 작업을 처리할 수 있는 한 특정 방법을 알려 주기 때문이다. 이러한 동작 방식을 위상 정렬(topological sorting)이라 한다. 왜냐하면 그래프 상의 정점들에 대해 순서를 갖기 때문이다.

일반적으로, 위상 정렬에 의해 만들어진 정점들의 순서는 유일하지가 않다. 예로서, 아래 순서는 우리의 예제에 대한 한 올바른 위상 정렬이다. 그리고 많은 다른 위상 정렬들이 있다:

A J G F K L E M B H C I D

위에서 언급한 제조 작업 분야에서도, 한 작업이 다른 한 작업과 직접적이든 간접적이든 간에 관계가 없는 경우, 이러한 상황이 발생한다. 물론 이런 경우에는 그들 작업이 어떤 순서로 수행되든지 문제가 없다.

때로는 그래프의 선분들을 다른 식으로 해석하는 것이 좋은 경우도 있다: 말하자면, 정점 x 에서 y 로 가는 한 선분은 정점 x 가 정점 y 에 “의존한다.(depend on)”는 의미이다. 예로서, 정점들이 프로그래밍 언어 설명서(도는 책)에 정의된 용어들을 나타낼 수도 있다; 만약 정점 x 의 정의시 y 를 이용한다면, x 에서 y 로 가는 선분이 있다. 이런 경우, 모든 용어가 다른 용어의 정의시 사용되기 전에 정의되어지는 성질을 가지고서 용어들의 순서를 정하는 것이 좋을 것이다. 이것은 모든 정점들을 한 선상에 놓고 이들간의 선분들은 모두 오른쪽에서 왼쪽으로 가도록 하는 것에 해당한다. 우리의 샘플(예제)그래프에 대한 역(reverse)위상 순서는 다음과 같다:

D E F C B I H G A K M L J

여기에서 그 차이는 중요한 것이 아니다: 한 그래프에 대해 역위상 정렬을 수행하는 것은 그 그래프의 모든 선분들의 방향을 바꿈으로써 얻어진 그래프에 대해 위상 정렬을 수행함과 동일하다.

그러나 이미 29장에서 역위상 정렬을 위한 알고리즘인 표준형 재귀적 깊이-우선 검색 프로시저를 배웠다!(입력 그래프가 dfs로 하여금 역위상 순서로서 정점들을 출력하게끔 할 때: 예로서, 프로시저 맨 끝에 `cout<<name(k)`를 삽입함으로써) 이것이 동작함을 한 간단한 귀납법적 설명으로 증명할 수 있다: 각 정점이 가리키는 모든 정점들의 이름들을 출력한 후에 그 정점의 이름을 출력하라. `visit`가 이런 식으로 변경되고 우리의 예제 그래프에 대해 수행됐을 때, `visit`는 위에 주어진 역 위상 순서대로 정점들을 출력한다. 이 재귀적 프로시저의 종료시, 정점의 이름을 출력하는 것은 이 프로그램을 시작할 때 스택(stack)에 그 정점의 이름을 넣고, 종료시 그 이름을 빼내어 출력하는 것과 같다. 이 경우 스택을 만들어서 사용할 이유가 없다. 왜냐하면 재귀를 위한 메커니즘이 자동으로 스택을 제공하기 때문이다; 그러나, 아래에서 다룰 더 어려운 문제의 경우에는 스택이 필요하다.

강하게 연결된-요소(Strongly Connected Components)

만약 한 그래프가 한 방향성 사이클을 가진다면, (만약 우리가 한 노드에서 지정된 방향에 있는 선분을 따라 갈 때 그 노드로 다시 오게 된다면) 이 그래프는 dag가 아니므로 위상 정렬이 될 수 없다: 사이클 상의 어느 정점이 먼저 출력 되더라도, 그 정점은 자기를 가리키는 그러나 아직 출력 안된 또 다른 정점을 가질 것이다. 사이클 상에 있는 노드들은 그들 사이클 상의 다른 노드로 갈 수 있고 또 이의 역의 경우도 성립된다는 의미에서 볼 때 상호 액세스가 가능하다고 할 수 있다. 이와는 달리, 비록 한 그래프가 연결 그래프라 할지라도, 방향성 그래프인 경우 어떤 한 노드에서 어떤 다른 노드에든지 도달할 수 있을 확률은 거의 없다. 사실상, 이 노드들은 한 요소(component)내의 모든 노드들은 상호 액세스가 가능하나 한 요소 내의 한 노드에서 다른 요소 내의 한 노드로 오고 가고(상호 액세스)할 방법이 없는 성질을 갖는 강하게 연결된-요소(strongly connected component)들이라고 불리는 노드들의 집합들로 분할된다. 그림 32.1의 방향성 그래프에서 강하게 연결된-요소들은 두 단독 노드 B와 K, 노드 쌍 H I, 세 노드 한 조인 D E F, 그리고 여섯 노드 A C G J L M으로 구성된 한 큰 요소이다. 예로서, 정점 A는 정점 F와 다른 요소 내에 있다. 왜냐하면 A에서 F로 가는 경로는 있으나, F에서 A로 가는 경로는 없기 때문이다.

한 방향성 그래프의 강하게 연결된-요소들은, 여러분이 예측 할 수 있듯이, 깊이-우선 검색 외 한 변형을 가지고 찾을 수 있다. 여기서 다룰 방법은 1972년 R. E. Tarjan이 만든 것이다. 이 방법은 깊이-우선 검색에 근거하여 만들어 졌으므로, $V + E$ 에 비례한 시간 내에 수행된다. 그러나, 이 방법은 실제로 아주 교묘한 방법이다; 이 방법은 단지 우리의 기본 visit 프로시저를 약간만 수정하면 만들어진다. 그러나, Tarjan이 이 방법을 제안하기 전에는, 이 문제에 대해 많은 사람들이 연구해 왔었음에도 불구하고 어떤 선형 시간 알고리즘도 알려진 바가 없었다.

한 그래프의 강하게 연결된-요소들을 찾기 위해 사용되는 깊이-우선 검색의 수정 버전은 30장에서 공부한 이중연결된-요소들을 찾기 위한 프로그램들과 아주 유사하다. 아래 주어진 재귀적 visit함수는 정점 k의 어떤 후손 정점으로부터 up링크를 경유하여 도달할 수 있는 최상위 정점을 찾기 위해 동일한 min계산을 사용한다. 그러나 강하게 연결된-요소들을 찾기 위한 경우에는 min의 값이 약간 다른 방법으로 이용된다.


```

Stack stack(maxV);
int visit (int k) // DFS to find strong components
{
    struct node *t; int m, min;
    val[k] = ++id; min = id;
    stack.push(k);
    for (t = adj[k]; t != z; t = t->next)
    {
        m = (!val[t->v]) ? visit(t->v) : val[t->v];
        if (m < min) min = m;
    }
    if (min == val[k])
    {
        do
        {
            m = stack.pop(); cout << m;
            val[m] = V+1;
        }
        while (m != k);
        cout << "\n";
    }
    return min;
}

```

이 프로그램은 visit를 수행 시작 할 때 스택에 정점들의 이름을 집어넣는다. 그리고나서, 각 강하게 연결된-요소의 마지막 정점을 방문하고 나올 때, 그들을 꼬집어내어 출력한다. 이 계산의 요점은 min과 val[k]가 같은지를 시험하는 것이다: 만약 같으면,(이미 출력된 것들을 제외한) 시작 이후 마주친 모든 정점들은 동일한 강하게 연결된-요소 k에 속한다. 다른 때와 마찬가지로, 이 프로그램은 단순히 요소들을 나타내 주는 것 외에도 더 정교하고 세련된 처리를 할 수 있도록 쉽게 수정 가능하다.

성질 32.4 그래프의 강하게 연결된-요소들은 선형 시간 내에 찾아짐을 알 수 있다.

위의 알고리즘이 강하게 연결된-요소들을 계산하는것에 대한 완전하고 정밀한 증명은 이 책의 범위를 벗어난다. 그러므로 이 증명의 주된 개념들만 대략 설명하기로 한다. 이 방법은

두 가지 관측에 근거하여 만들어 졌다. 첫째는 “한 정점에 대한 visit 호출의 종료 시점이 되면, 동일한 강하게 연결된-요소내에 있는 더 이상의 정점들과 마주치지 않을 것이다”라는 것이다. 왜냐하면, 그 정점으로부터 도달할 수 있는 모든 정점들이(우리가 위상 정렬에서 주목해 본 바와 같이) 다 처리되었기 때문이다. 둘째는, “트리 내에서의 up 링크들은 한 정점에서 다른 정점까지의 두 번째 경로를 제공하며 강하게 연결된-요소들을 함께 결합시킨다”는 것이다. 연결점들을 찾는 30장에 있는 알고리즘의 경우와 같이, 각 노드의 모든 후손들로부터 한 up 링크를 경유하여 도달할 수 있는 최고의 조상을 계속 추적하여 알고 있어야 한다. 만약 한 정점 x 가 깊이-우선 검색 트리에서 어떤 후손들이나 up 링크들을 갖고 있지 않거나, 자신을 가리키는 한 up 링크를 갖는 깊이-우선 검색 트리 내의 한 후손을 갖고 있지만 그 트리 내에서 그 윗 부분을 가리키는 up 링크들을 갖는 어떤 후손들도 갖고 있지 않다면, 그 정점과 그 모든 후손들은(이들과 동일한 성질을 만족시키는 정점들과 그 후손들을 제외한) 한 강하게 연결된-요소를 형성한다. 따라서, 그림 32.2의 깊이-우선 검색 트리 에서, 노드 B와 K는 첫 번째 조건을 만족시키고,(그래서 이들은 강하게 연결된-요소들 그 자체임을 나타낸다) 그리고 F E D를 나타내는 노드 F, H I를 나타내는 노드 H 및 A G J L M C 를 나타내는 노드 A는 두 번째 조건을 충족시킨다. A로서 표현되는 요소를 구성하는 정점들은 B K F와 그들의 후손들을 제거함으로써 찾아진다.(이들은 이 전에 찾아진 요소들에서 나타난다) 이와 동일한 성질을 만족시키지 않는 x 의 모든 후손 y 는 트리에서 y 보다 더 높은 곳을 가리키는 한 개의 up 링크를 가진다. 트리를 홀어 내려가면 x 에서 y 로 가는 한 경로가 있다; 그리고 y 에서 x 로 가는 한 경로는 y 에서 부터 계속 내려가서 y 를 지나서 바로 도달하는 up 링크를 갖는 정점까지 감으로써 찾아지고, x 에 이를 때까지 동일한 과정을 반복한다. 또 하나의 중요한 점은 한번 정점이 처리되면, 이 정점에 높은 val값이 부여되므로, 그 정점으로가는 cross링크들은 무시된다는 것이다. □

이 프로그램은 상대적으로 어려운 문제에 대해 아주 간단한 해결책을 제공한다. 방향성 그래프들의 검색에 관련한 미묘한 문제들이 있음은 확실하다, 그러나(이 경우에는) 조심하여 만들어진 재귀적 프로그램으로서 이 미묘한 문제들을 처리할 수 있다.

연습문제

1. 그림 32.8의 dag의 추이적 폐포에 대한 인접-행렬을 구하라.
2. 인접-행렬로 표현된 무방향 그래프에 대해 추이적 폐포 알고리즘을 실행시키면 어떤 결과가 나올까?
3. 한 주어진 방향성 그래프(인접-리스트로 표현된)의 추이적 폐포에 있는 선분들의 수를 결정하는 프로그램을 작성하라.
4. Warshall 알고리즘이 이 책에 설명된 깊이-우선 검색 기법을 사용함으로써 유도된 추이적 폐포 알고리즘과 어떻게 비교될 수 있는지를 논하라. 단, 인접-행렬 형태를 갖는 visit 프로시저를 이용하고, 알고리즘을 비재귀적으로 만든 다음 비교하라.
5. 제안된 방법은 인접-행렬 표현을 사용하나 dfs는 미방문된 정점들을 찾을 때 정점들을 역순으로(즉, V 에서 1로) 스캔할 때, 그림 32.8에 주어진 dag에 대해 산출된 위상 순서(topological order)를 구하라.
6. 31장의 최단 경로 알고리즘은 방향성 그래프들에 대해서도 동작하는가? 만약 그렇다면 그 이유를 설명하고, 아니라면 동작이 안되는 경우의 예를 들라.
7. 주어진 한 방향성 그래프가 dag인지 아닌지를 결정 해주는 프로그램을 작성하라.
8. 한 dag상에 얼마나 많은 강하게 연결된-요소들이 있는가? 또한 크기가 V 인 방향성 사이클을 갖는 그래프의 경우에는 어떠한가?
9. 29장과 30 장에 있는 프로그램들을 사용하여 V 개의 정점들을 갖는 아주 큰 무작위적 방향성 그래프를 만들라. 이처럼 큰 그래프들이 얼마나 많은 강하게 연결된-요소들을 갖는 경향이 있는지에 대해 논하라.
10. 30장에 있는 find와 기능적으로 유사한, 그러나 입력 선분들에 의해 설명되는(알 수 있는) 방향성 그래프의 강하게 연결된-요소들을 유지시켜 주는 프로그램을 작성하라.(이 문제는 쉽지가 않다; 따라서 find 만큼 효율적인 프로그램을 작성할 수 없을 것임은 확실하다.)

33 장

네트워크 흐름

가중치 방향성 그래프들은 상호 연결된 네트워크를 통한 물류 흐름을 포함한 여러 유형의 응용 분야에 유용한 모델들이다. 복잡하게 상호 연결되고 접합부에 흐름의 방향을 조절하는 스위치를 가진 다양한 크기의 기름 파이프 네트워크를 생각해 보자. 더 나아가서 네트워크의 연결된 모든 파이프들에 대해 하나의 근원지(예, 기름 유전)와 하나의 목적지(예, 큰 정제기)를 갖는다고 하자. 출발지에서 목적지까지 흐르는 오일의 양을 극대화시킬 수 있는 스위치 설정(setting)은 무엇인가? 접합부에서의 물질의 흐름을 포함한 복잡한 상호작용은 네트워크 흐름 문제를 풀기 어려운 문제로 만든다.

이런 일반적으로 동일한 형태(setup)는 고속도로 상의 교통의 흐름이나, 공장들에서의 물질의 흐름 등을 설명하는데 사용될 수 있다. 이런 문제의 많은 다른 형태들이 적용될 수 있는 많은 실제 해당 상황들이 연구되고 있다. 이런 문제들에 대해 효율적인 알고리즘을 찾기 위한 확실하고 강한 동기가 있다.

이런 유형의 문제는 컴퓨터 과학 분야와 과학적 경영 분석(operations research) 분야간의 경계(interface)에 있다. 과학적 경영 분석 분야들은 일반적으로 의사 결정의 용도를 위한 복잡한 시스템들의 수학적 모델링(최적의 경우가 물론 선호된다)과 관계가 있다. 네트워크 흐름은 과학적 경영 분석 문제의 한 전형적인 예이다. 42-45장에서 일부 다른 문제들도 간략하게 언급하기로 한다.

43장에서는, 과학적 경영 분석 모델들로부터 전형적으로 생성되는 복잡한 수학 방정식을 해결하기 위한 일반적 방식인 선형 프로그래밍(linear programming)을 공부할 것이다. 네트워크 흐름 문제와 같은 특정 문제에 대해 보다 좋은 알고리즘이 있음직하다. 사실상, 우리는

지금까지 조사해 왔던 그래프 알고리즘들이 네트워크 흐름 문제들에 대한 고전적인 해결책과 밀접하게 연관되어 있는 것을 볼 것이다. 그리고 우리가 개발해 온 알고리즘 도구들을 사용하여 이 문제를 해결하기 위한 프로그램을 개발하는 것이 오히려 쉽다는 것을 알게 될 것이다. 이 문제는 아직도 활발하게 연구되고 있다: 우리가 이제까지 본 많은 문제들과는 다르게, 이 문제에 대한 “최고”의 해결책은 아직도 발견되지 않았고 많은 새로운 알고리즘들이 아직도 개발되고 있다.

네트워크 흐름 문제(The Network Flow Problems)

그림 33.1에 있는 것처럼 작은 기름 파이프들의 네트워크이 이상적으로 그려진 그림을 생각해 보자. 파이프들은 그들의 크기에 비례하여 용량이 고정되어 있고 기름은 단지 아래쪽으로만 흐를 수 있다. 더군다나, 각 접합부에 있는 스위치들은 각 방향으로 흐르는 기름의 양을 조절한다. 어떻게 스위치들이 배치되어 있다 하더라도, 시스템은 최상부의 기름 흐름의 양과 최하부의 기름 흐름의 양이 같을 때 평형 상태(equilibrium state)에 도달한다. 여기서는 흐름의 양과 파이프 용량을 정수 단위로 측정한다.(예, 초당 갤런(gallon))

스위치의 배치 방식이 총 최대치 흐름에 실제로 영향을 줄 수 있는지를 즉시 알 수 있는지는 명확하지 않다: 그림 33.1은 영향을 줄 수 있음을 보여주고 있다. 첫째로, 그림 왼쪽 도면에서처럼, 스위치를 조절하는 파이프 AB가 열리고 그 파이프와 파이프 BD, 그리고 파이프 BF에 기름이 거의 채워지고 있다고 가정하자. 다음에 파이프 AC가 열리고 스위치 C는 파이프 CD를 닫고 파이프 CE를 연다.(아마, 스위치 D의 조작자(operator)는 스위치 C의 조작자

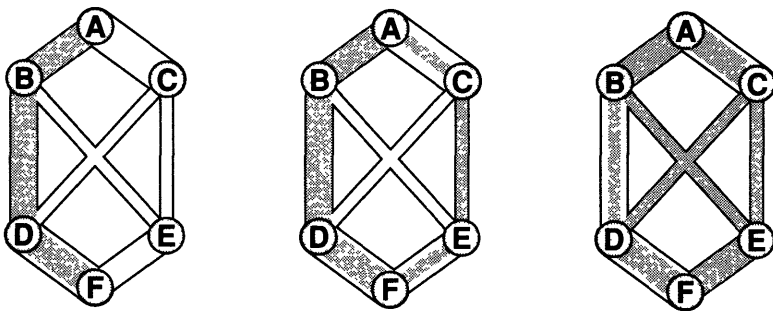


그림 33.1 한 단순한 네트워크에서의 최대 흐름

에게 B로부터의 흐름의 부하 때문에 C의 조작자가 D를 더 이상 처리 못함을 알렸을 것이다) 흐름의 결과는 그림의 중간 도면에서 보여진다. 파이프 BD 와 CE가 가득 채워진다.

이제 흐름은 파이프 DF를 채우기 위해 경로 ACDF를 통해 충분히 보냄으로써 조금씩 증가될 수 있다. 그러나, 세 번째 도면에서 보여주는 것처럼 좀 더 나은 해결책이 있다. BE를 채우도록 흐름의 방향을 바꾸기 위해, 스위치 B의 현 설정을 변경함으로써, 파이프 DF에 충분한 용량을 주게 되고 따라서 스위치 C는 파이프 CD를 완전히 열 수 있게 된다. 네트워크로 들어오고 나가는 모든 흐름의 양은 적절한 스위치의 배치를 찾음으로써 증가된다.

우리가 하고자 하는 것은 임의의 네트워크에 대해서 “적절한” 스위치의 배치를 발견할 수 있는 알고리즘을 개발하는 것이다. 이에 더하여, 어떤 스위치 배치도 더 많은 흐름을 줄 수 없음이 보장되기를 원한다.

이런 상황은 방향성 그래프에 의해 명백히 모델화될 수 있으며, 우리가 이제까지 공부해 온 프로그램들을 적용할 수 있음도 알려져 있다. 네트워크를 두 부류의 다른 정점들로 구성된 가중치 방향성 그래프로 정의하자: 안으로 들어오는 입력 선분이 없는 것을 근원지, 밖으로 나가는 출력 선분이 없는 것을 종착지라 한다. 선분 상의 가중치는, 0이 아니라고 가정할 때, 선분 용량(capacity)이라 불린다. 그리고, 흐름은 각 선분이 용량보다 작거나 같은 가중치들의 또 하나의 집합으로 정의된다. 각 정점으로 들어가는 흐름은 그 정점으로부터 나가는 흐름과 같다. 그 흐름의 값은 근원지로부터의 흐름(또는 종착지로 가는 흐름이다.) 네트워크 흐름 문제는 주어진 네트워크에서 흐름의 최대치를 찾는 것이다.

네트워크는 우리가 전 장에서 그래프에 대해 사용해 왔던 인접-행렬이나 인접-리스트 표현 방식으로 명확하게 묘사될 수 있다. 하나의 가중치 대신에 각 선분에 가중치 size와 가중치 flow가 부여되어 있다. 이들 인접-행렬로 표현 시에는 두 개의 행렬로서, 인접-리스트 노드로 표현 시에는 두 개의 필드로써, 또는 두 가지 표현 방식 중의 어느 경우에서는 한 레코드내의 두 개의 필드로서 나타내어 질 수 있다. 비록 네트워크가 방향성 그래프일지라도, 우리가 조사할 알고리즘들은 잘못된(반대) 방향으로 선분을 운행할 필요가 있다. 그래서, 우리는 여기서 무방향 그래프 표현을 사용한다: 만일 크기가 s 이고 흐름이 f 인, x 에서 y 로 가는 선분이 있다면, 또한 크기가 $-s$ 이고 흐름이 $-f$ 인, y 에서 x 로 가는 선분을 유지한다. 인접-리스트 표현에서는, 각 선분을 나타내는 두 개의 리스트 노드를 연결하는 링크들을 유지함이 필요하다. 그럼으로써, 우리가 흐름을 한 방향으로 바꿀 때, 다른 방향의 흐름 또한 변경할 수 있다.

Ford-Fulkerson 방법(Ford-Fulkerson Method)

네트워크 흐름 문제에 대한 한 고전적인 접근 방식은 1962년 L.R. Ford와 D.R. Fulkerson에 의해 개발됐다. 이들은 한 타당한 흐름을(물론, 최대의 흐름을 제외하고) 개선하기 위한 방법을 제시했다. 흐름 0에서 시작하여, 이 방법을 반복하여 적용한다. 방법이 적용될 수 있는 한, 이 방법은 흐름을 증가시킨다. 만일 더 이상 적용될 수 없다면, 최대치 흐름이 찾아진 것이다. 사실, 그림 33.1에 있는 흐름은 이 방법을 사용해 만들어진 것이다, 그림 33.2에 있는 그래프 표현의 관점에서 우리는 이 흐름을 재조사하기로 한다.

단순성을 위해, 화살표를 생략한다. 왜냐하면 이들 모두가 아래를 가리키기 때문이다. 우리가 고려하고 있는 방법들은 방향성 선분만 있는 그래프에 제한되는 것은 아니다. 그러나 파이프 내에서의 액체 흐름의 관점에서, 네트워크 흐름을 이해하는데 있어서 좋은 직관력을 제공하므로 방향성 선분만 있는 그래프를 사용하기로 한다.

근원지에서 종착지까지의 네트워크를 통하는 한 직접적인 경로를 생각해 보자. 분명히 흐름은 경로 상의 사용안된 선분의 용량만큼이라도 경로상의 모든 선분상의 흐름을 증가시킴으로서 증가될 수 있다. 이 규칙은 그림 33.2의 왼쪽 다이어그램에서는 경로 ABDF를 따라 적용됐고, 중앙 다이어그램에서는 경로 ACEF를 따라 적용됐다.

앞에서 언급한 것처럼, 그 후에 경로 ACDF를 따라 그 규칙을 적용할 수 있고, 그럼으로써 네트워크를 통한 모든 방향성 경로들이 적어도 하나의 용량으로 채워진 선분을 갖는 상황을 만들어 낸다. 그러나 흐름을 증가시키는 다른 방법이 있다: 우리는 역방향(잘못된 방향)을 가리키는 선분들을 가지는 네트워크를 통한 경로들을 고려해 볼 수 있다. 흐름은 근원지부터 종착지까지의 선분 상의 흐름을 증가시키는 그런 경로를 따라 흐름이 증가되고, 종착지부터 근원지까지의 선분 상에서는 같은 양만큼 흐름이 줄어든다. 우리의 예제에서는, 네트워크를

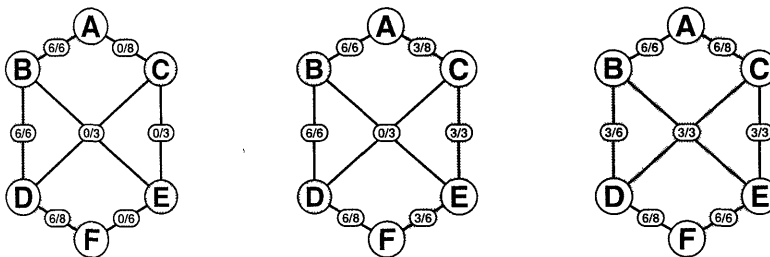


그림 33.2 한 네트워크 내에서의 최대 흐름 찾기

통한 흐름은 그림 33.2의 세 번째 그림과 같이 경로 ACDBEF를 따라가면 3 만큼 증가될 수 있다. 위에서 설명한 것처럼, 이것은 AC와 CD를 통해 흐름이 3 단위 만큼 추가된 것에 해당되며, 그리고나서 스위치 B에서 는, 3 단위의 흐름을 BD에서 BE와 EF로 보낸다: 물론 DF에서는 어떤 흐름의 손실도 없다. 왜냐하면, BD로부터 와서 사용된 3 단위의 흐름은 이제는 CD로부터 오기 때문이다.

용어를 단순화하기 위해, 근원지로부터 종착지까지의 특별한 경로를 통해 흐르는 선분을 순방향 선분(forward edge)이라고 하고, 종착지에서부터 근원지까지의 흐름의 선분을 역방향 선분(backward edge)이라고 한다. 흐름의 증가 가능 양은 순방향 선분들에서 미사용된 용량들의 최소치에 국한되고 역방향 선분 흐름들의 최소치에 의해 국한됨을 주목하라. 다시 말해서, 새로운 흐름에서, 경로를 따라 순방향 선분 중의 적어도 하나는 꼭차게 되거나 또는 그 경로상의 역방향 선분 중의 하나는 적어도 비게 된다. 더 나아가서, 흐름은 꼭 찬 순방향 선분이나 빈 역방향 선분을 포함하는 경로 상에서는 증가될 수 없다.

위의 절에서는 꼭 찬 순방향 선분이나 빈 역방향 선분이 없는 경로가 있다면, 네트워크 상에서 흐름을 증가시키는 방법을 알려주었다. Ford-Fulkerson 방법의 핵심은 그러한 경로가 발견되지 않을 경우 그 흐름이 최대치라는 것을 보여줌에 있다.

성질 33.1 한 네트워크에서 근원지로부터 종착지까지의 모든 경로가 꼭찬 순방향 선분이나 빈 역방향 선분을 가지면, 그 흐름은 최대이다.

이 사실의 증명을 위해, 먼저 그래프를 살펴보고 모든 경로 상의 첫 꼭 찬 순방향 선분이나 빈 역방향 선분을 찾아보자. 이러한 선분들의 집합은 그래프를 두 부분으로 절단(cut)한다.(우리의 예제에서, 선분 AB, CD 그리고 CE가 이같은 절단부를 구성한다) 네트워크가 두 부분으로 절단된 경우, 그 절단부를 “횡단하는” 흐름을 측정할 수 있다: 즉 근원지에서 종착지로 가는 선분들의 흐름의 총량을 측정할 수 있다. 일반적으로, 선분들은 절단부를 횡단하여 양방향으로 갈 수 있다: 그 절단부를 횡단하는 흐름을 얻기 위해, 다른 방향으로 가는 선분들 상의 흐름의 총량은 감소되어야 한다. 우리 예제에서는 절단부는 12의 값을 가지는데, 이 값은 네트워크에 대한 전체 흐름의 양과 같다. 절단부의 흐름이 전체 흐름과 같을 때는 언제나, 그 흐름이 최대일 뿐만 아니라 절단부는 최소라는 것을(즉 모든 다른 절단이 최소한 “횡단흐름(crossflow)”만큼 크다.) 알 수 있다. 이를 “최대흐름-최소절단 정리(maxflow-mincut theorem)”라고 한다. 이 흐름은 더 이상 커질 수 없고(그렇지 않으면, 절단부의 흐름 또한 커질 수 있

다) 더 이상 작은 절단부의 흐름 또한 존재하지 않는다.(그렇지 않으면, 흐름은 더욱 작아져야만 할 수도 있다) 이 증명의 자세한 부분은 생략한다. □

네트워크 검색(Network Searching)

위에 기술한 Ford-Fulkerson 방법은 다음과 같이 요약될 수 있다: “흐름 0인 어느 곳에서든지 시작하여 꼭 찬 순방향 선분이나 빈 역방향 선분이 없는 근원지에서 종착지까지 가는 경로를 따라 흐름을 증가시킨다. 이 과정은 네트워크에 그와 같은 경로들이 없을 때까지 계속된다.” 그러나 이것은 일상적인 의미의 알고리즘은 아니다. 왜냐하면 경로를 찾는 방법이 지정되지 않았고 따라서 어떤 경로도 사용될 수 있기 때문이다. 예를 들어, 어느 사람은 경로가 길수록 네트워크가 더 채워질 것이라는 직감에 그 방법의 기초를 둘 수 있다. 하지만 그림 33.3에서 보여진 예는 몇 가지 주의해야 할 사항을 보여준다.

이 네트워크에서, 만약 선택된 첫 번째 경로가 ABCD라면, 그 흐름은 단지 하나 증가한다. 그리고 선택된 두 번째 경로가 ACBD이면, 또다시 흐름이 하나 증가하여 바깥쪽 선분들 상의 흐름이 하나씩 증가함을 제외하고는, 상황이 초기 상황으로 가게 된다. 이런 두 경로를 선택한 알고리즘은(예를 들어 긴 경로들을 찾는 알고리즘) 이 방식(전략)으로 계속할 것이다. 그리하여 최대 흐름이 찾아지기에 앞서, 1000번 이상의 반복이 필요하게 된다. 만약 바깥쪽 선분들 상의 숫자가 10억이라면, 20억 번의 반복이 필요할 것이다. 이것은 명백히 바람직하지 않다. 왜냐하면 경로 ABD 및 ACD는 단지 두 스텝에 최대 흐름을 주기 때문이다. 알고리즘이 유용하려면, 실행 시간이 용량의 크기에 좌우되는것을 피해야 한다. 다행히도, 이 문제는 쉽게 제거된다.

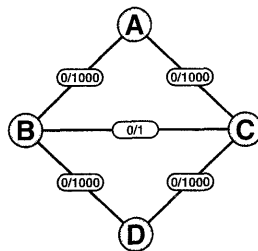


그림 33.3 많은 반복(iteration)을 필요로 할 수 있는 네트워크.

성질 33.2 만약 근원지에서 종착지까지의 최소의 이용 가능 경로가 Ford-Fulkerson 방법에서 이용되면, V 개의 정점들과 E 개의 선분을 갖는 네트워크에서 최대 흐름이 찾아가지기 전까지 사용된 경로들의 수는 반드시 VE 보다 작아야 한다.

이 사실은 1972년에 Edmonds와 Karp에 의해 증명되었다. 이 증명의 자세한 사항은 이 책의 범위를 벗어난다. □

다시 말해서, 한 가지 좋은 방안은 경로를 찾기 위해 적당하게 수정된 버전의 너비-우선 검색을 사용하는 것이다. 성질 33.2에서 주어진 경계 값은 최악의 경우의 경계 값이다: 전형적인 네트워크는 아주 작은 스텝 수를 필요로 하는 경우가 많다.

31장의 우선순위(priority) 그래프-유행 방법으로 Edmonds와 Karp에 의해 제안된 다른 방법을 구현할 수 있다: 가장 큰 양으로 흐름을 증가시키는 네트워크를 통한 경로를 찾는 것이다. 이것은 간단히(그 값이 적당히 설정된) 우선순위(priority)에 대한 변수를 31장의 인접-리스트 또는 인접-행렬 “우선순위 우선 검색 방법(priority-first search)” 방법들 중의 하나를 이용하여 구해질 수가 있다. 행렬 표현에 대해 아래 프로그램은 우선순위를 계산하며, 리스트 표현에 대한 코드는 이와 유사하다:

```
priority = -flow[k][t];
if (size[k][t] > 0) priority += size[k][t];
if (priority > val[k]) priority = val[k];
```

그리고 우리는 최고 우선순위 값을 가진 노드를 택하기를 원하기 때문에, 이러한 프로그램들 내에서 우선순위 큐 메커니즘을 최소값 대신 최대값을 리턴 하도록 변경하거나 그들을 있는 그대로 쓰되 변수 priority를 임의의 큰 정수에 따라 보수를 취해야 한다.(그리고, 이 과정은 그 값이 지워졌을 때 역순으로 되야한다) 또한, 우리는 근원지와 종착지를 인수로써 취하기 위해, 우선순위-우선 검색 프로시저를 수정해야 한다. 그런 후, 각 근원지에서 검색을 시작하고 종착지로 가는 경로가 발견됐을 때 검색을 멈춘다.(경로 발견 시는 1을, 아니면 0을 리턴 한다) 경로가 없으면, 부분적인 검색 트리는 네트워크에 대한 최소절단(mincut)을 정의한다; 그렇지 않으면, 그 흐름이 개선될 수 있을 것이다. 마지막으로, 근원지에 대한 val은 검색 시작 전에 maxint로 설정 되어 한다. 이는 근원지에서 얻어질 수 있는 어떤 흐름의 양도 나타내기 위함이다.(비록 이것이 근원지로부터 직접 나오는 모든 파이프들의 총 용량 합계에 의해 즉시 제약되어지기는 하지만)

앞 절에서 기술된 우선순위-우선 검색 구현을 가지고, 최대흐름을 찾는 것은 아주 간단해진다. 다음 프로그램은 네트워크에 대해 인접-행렬표현 방식을 사용함을 가정하여 작성한 것이다.

```
for( ; ; )
{
    if (!search(1,V)) break;
    y = V; x = dad[v];
    while (x!=0)
    {
        flow[x][y] = flow[x][y]+val[v];
        flow[y][x] = -flow[x][y];
        y = x; x = dad[y];
    }
}
```

search 루틴이 흐름을 최대로 늘리는 경로를 찾을 수 있는 한, 우리는 그 경로를 통하여 거슬러 추적하고(search에 의해 형성된 dad 배열을 이용하여) 지적된 것처럼 흐름을 증가시킨다. 만약 V 가 몇 번의 search에 대한 호출 후에도 unseen으로 남아 있다면 최소절단이 발견되고 알고리즘은 끝나게 된다.

우리가 봐온 바와 같이, 이 알고리즘은 우선 경로 ABDF, 그 다음은 ACEF, 그리고는 ACDBEF를 따라 흐름을 증가시킨다. 이 방법은 세 번째 경로로서 ACDF를 선택하지 않았다. 왜냐하면 이 경로는 더 긴 경로를 통한 세 단위의 증가가 아닌, 한 단위만의 흐름증가를 가져오기 때문이다. 성질 33.2의 너비-우선 “최단경로-우선” 방법은 이 선택을 할 수도 있음을 주목하라.

비록 알고리즘은 구현하기 쉽고 실제 네트워크에서 잘 동작할 듯은 하나, 이의 분석은 아주 복잡하다. 우선, search는 최악의 경우에 V^2 스텝을 필요로 한다; 대안으로, 각 반복(iteration) 당 $(E + V)\log V$ 에 비례하는 시간 so 에 실행되는 인접-리스트를 사용한다. 비록 이 알고리즘은 종착지에 도착하면 정지하기 때문에, 실행시 다소 빠른 경향이 있을 수는 있다. 그러나 얼마나 많은 반복이 필요한가?

성질 33.3 최대 흐름을 증가시키는 근원지에서 종착지까지의 경로가 Ford-Fulkerson 방법에서 사용된다면, 네트워크에서 최대흐름이 발견되기 전에 사용된 경로들의 수는 1

$+ \log_{M/M-1} f^*$ 보다 작다. f^* 는 흐름의 비용이고 M 은 네트워크의 한 절단부(cut)에 있는 선분들의 최대수이다.

Edmonds와 Karp에 의해 증명된 이 사실은 이 책의 범위를 넘어선다. 이것은 확실히 계산하기에 복잡하다. 그러나 실제 네트워크들에서는 그럴 가능성이 별로 없을 것 같다. □

우리는 이러한 성질을, 실제 네트워크 상에서 얼마나 알고리즘이 오래 수행되는지를 보이기 위해서가 아니라, 오히려 이 분석의 복잡성을 보이기 위해 언급했다. 실제로, 이 문제는 아주 널리 연구되어 왔고 개선된 최악시의 경계값(bounds)을 갖는 복잡한 알고리즘들이 개발되어 왔다. 그러나 위에 구현된 Edmonds-Karp 알고리즘은 실제 응용 분야에서 발생하는 네트워크들에서 동작하기는 어려울 듯 싶다. 그림 33.4는 보다 큰 네트워크 상에서 동작하는 알고리즘을 보여준다.

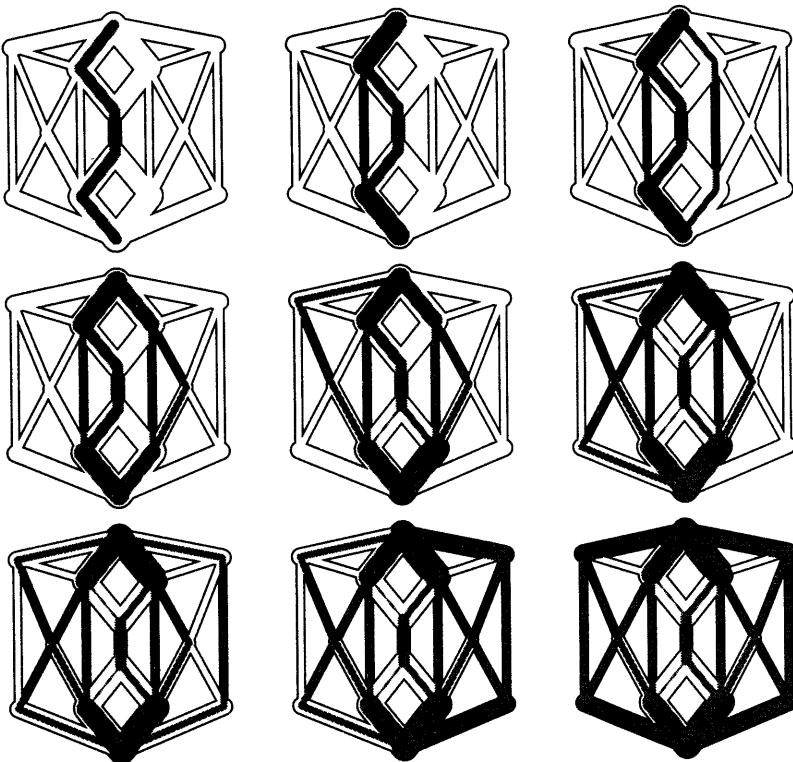


그림 33.4 보다 큰 네트워크에서의 최대흐름 찾기

네트워크 흐름의 문제점은 여러 가지 방법들로 확장될 수 있다. 그리고, 많은 변형된 방법들이 좀 더 자세히 연구되고 있다. 왜냐하면 이들은 실제 응용 분야에서 매우 중요하기 때문이다. 예를 들어, 다종류 물류 흐름(multicommodity) 문제는 다중 근원지와 종착지 그리고 여러 유형의 물류들을 네트워크에 도입한 것이다. 이것은 문제를 더욱 어렵게 만들고, 여기서 살펴본 것들 보다 더 고수준의 알고리즘을 요구한다. 예로써, 일반적인 경우, 최대흐름-최소절단 정리에 대한 어떤 유사 정리도 성립한다고 알려진 바가 없다. 네트워크 흐름 문제에 대한 또 다른 확장은 정점 상의 용량 제약 조건(capacity constraints)을 부가하는 것(이들 용량의 처리는 인공적인 선분들을 도입함으로써 쉽게 처리된다), 무방향 선분들을 허용하는 것(이 또한 단 방향 선분들의 쌍을 만듦으로써 쉽게 처리된다) 그리고 선분 상의 흐름들에 대해 하한값(lower bound)들을 부여하는 것이다.(이것은 그렇게 쉽게 처리되지 않는다) 만약 파이프들이 용량 뿐만 아니라 그 사용비용 또한 부가되어 있다는 실제적인 가정을 하면, 문제는 최소비용-흐름(min-cost-flow) 문제가 되는데, 이는 과학적 경영 분석 분야에서 아주 어려운 문제 중의 하나이다.

연습문제

1. 네트워크의 종착지가 없어졌을 때 한 트리가 형성되는 경우의 네트워크 흐름 문제를 해결하는 알고리즘을 구하라.
2. 가중치가 각각 3인 B에서 C 그리고 E에서 D의 두 선분을 더함으로써 얻어진 네트워크에서 최대흐름을 찾으려 할 때, 성질 33.3에서 언급한 알고리즘에 의해 어떤 경로들이 추적되나?
3. 이 책에서 논의된 예제에서 search에 대한 각 호출시 계산된 우선순위 검색 트리를 그려라.
4. 이 책에서 논의된 예제에서 search에 대한 각 호출 후의 flow 행렬의 내용을 구하라.
5. 참 또는 거짓: 네트워크에서 모든 선분을 조사하지 않고 최대흐름을 찾는 알고리즘은 없다.
6. 네트워크가 방향성 사이클을 가질 때, Ford-Fulkerson 방법의 사용시 어떤 일이 일어나는가?
7. 모든 용량이 $O(1)$ 인 경우에, 한 단순화된 버전의 Edmonds-Karp 경계를 구하라.
8. 왜 깊이-우선순위 검색이 네트워크 흐름 문제에 대해 적합하지 않는지를 보여주는 반증을 구하라.
9. 인접-리스트 표현 방식을 가진 우선순위-우선 검색을 사용하여, 네트워크 흐름 문제에 대한 해결책을 너비 우선순위 검색으로 구현하라.
10. V 개의 노드와 약 $10V$ 개의 선분을 가진 무작위(random) 네트워크에서 최대흐름을 찾는 프로그램을 작성하라. $V = 25, 50, 100$ 의 경우 얼마 만큼의 반복이 필요한가?

빈 면

이 장에서는 그래프 구조에서 물체들을 짝지을 때와 불일치하는 선호도의 관계들에 따른 문제점을 설명하기로 한다.

예를 들면, 미국에서 졸업반 의대생들을 전문의 수련자로 배치하기 위해 아주 복잡한 시스템이 설치되었다. 각 학생들은 자신의 선호도 순으로 병원 목록을 작성하고, 각 병원들도 그들의 선호도 순으로 학생 목록을 작성하였다. 문제는 학생과 병원이 각각 작성한 목록들에 따라 공정하게 학생들을 분배하는 것이다. 성적이 좋은 학생들은 여러 병원에 의해 선호되고, 인기 있는 병원들의 자리는 몇몇 학생들에 의해 선호되기 때문에, 복잡한 알고리즘이 요구된다. 각 병원이 작성한 선호도-목록에 따라서 그대로 각 병원의 자리가 학생들로 채워질 수 없으며, 또한 각 학생들도 자신이 선호하는 순으로 작성한 목록대로 자리를 배정 받을 수는 없다. 이런 일은 자주 발생한다. 사실 알고리즘을 이용하여 선호도에 따라 학생과 병원을 매치한 후, 마지막에는 나머지 매치되지 않은 학생과 병원의 처리에 고심할 것이다.

이 예제는 폭넓게 연구되어온 그래프들에 대한 어려운 원천적 문제의 한 특별한 경우이다. 주어진 한 그래프에서, 매칭은 두 번 이상 정점(vertex)이 나타나지 않는 선분(edge)들의 부분집합이다. 즉, 매칭에서 선분들 중 하나에 연결된 각 정점은 그 선분의 다른 한 정점과 쌍을 이룬다. 그러나 몇몇 정점들은 매치되어 있지 않을 수 있다. 비록 가능한 많은 정점들에 대해 매칭이 되더라도, 매치안된 어떤 선분들도 매치안된 정점들과는 연결되어서는 안된다는 점에서, 선분들을 선택하는 여러 다른 방법들에 따라 매치되지 않고 남은 정점들의 수는 다를 수 있다.

특히 흥미 있는 것은 최대 매칭(maximum matching)이다. 이 매칭은 가능한 많은 선분들을 포함하거나, 또는 정점들의 수를 최소로 한다. 우리가 바라는 최상의 경우는($2V$ 개의 정

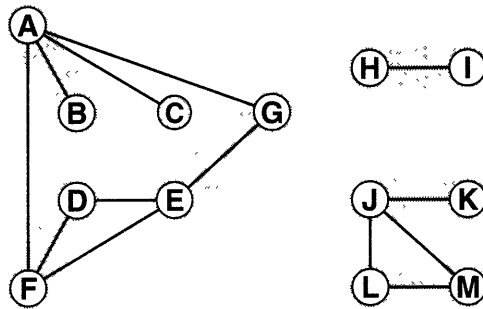


그림 34.1 최대 매칭

점들을 가진 그래프에서, 최상의 매칭은 V 개의 선분을 가진다(각 정점이 한 번씩만 나타나는 선분들의 집합을 갖는 것이다. 그러나, 이것이 항상 가능한 것은 아니다.

그림 34.1은 우리의 샘플 예제에 대한 최대 매칭(음영이 있는 선분들)을 보여주는 그래프이다. 13개의 정점들을 가지고, 여섯 선분들을 가진 매칭보다 더 좋게 할 수는 없다. 그러나 매칭들을 찾기 위한 간단한 알고리즘들은 이 예제의 경우 오히려 어려움이 있을 것이다. 예를 들어, 한 방법에서는 깊이-우선 검색(그림 29.7 참조)에 나타난 것처럼, 매칭을 위해 가능한 선분들을 선택할 수도 있다. 그림 34.1의 경우, 이 방법은 방금 언급한 것처럼 AF EG HI JK LM의 다섯 선분들을 찾아 주는데, 이것은 최대 매칭이 아니다. 또한 주어진 그래프에서 얼마나 많은 수의 선분들이 최대 매칭안에 있을지를 아는 것도 쉬운 일이 아니다. 예를 들어, A부터 F까지의 단지 여섯 개의 정점과 그 정점들을 연결하는 선분들로 구성된 서브그래프에서는 세-선분(three-edge) 매칭이 없음에 주목하라. 큰 그래프 상에서 많은 매칭을 찾는 것은 종종 매우 쉬운 반면에,(예로서, 29장의 “미로(maze)” 그래프에서 최대 매칭을 찾는 것은 어려운 일이 아니다) 위 경우와 같이 상반 된 예제들에서 보는 것처럼, 임의의 그래프에 대해 최대 매칭을 찾는 알고리즘을 개발한다는 것은 참으로 어려운 일이다.

위에서 서술한 의대생 매칭 문제의 경우, 학생들과 병원들은 그래프의 노드(node)에 해당하고 그들의 선호도는 선분에 해당한다. 만약 그들의 선호도에 대해(1부터 10까지의) 가중치를 주면, 가중치 매칭 문제(weighted matching problem)가 된다: 즉, 주어진 가중치 그래프에서, 선택된 선분들의 가중치들의 합이 최대가 되도록 하면서 어떤 정점도 한번만 나타나는 선분들의 집합을 찾아라. 아래에서는, 선호도의 순서에 비중을 두나 선호도 값에는 무관한 또 하나의 방법을 고려하고자 한다.

매칭 문제는 그 직관성과 넓은 응용성으로 인해 수학자들의 관심을 아주 많이 끌고 있다. 매칭시의 해결책은, 일반적인 경우, 이 책의 범위를 아주 벗어나는 복잡하면서도 기묘한 조합 수학(combinatorial mathematics)을 포함한다. 여기서 우리의 의도는 독자들이 약간의 재미있고 특별한 경우들을 평가해 볼 수 있게 해줌과 동시에 이들에 대해 유용한 알고리즘들을 개발해보도록 유도하려는 것이다.

양분 그래프(Bipartite Graphs)

앞서 언급한 예제인 의대생들을 전문의 수련을 위한 병원으로 매칭하는 것은 많은 다른 매칭 응용 분야들 중 확실히 대표적인 예이다. 예를 들어, 남녀의 매칭, 원하는 일자리와 직업 응모자와의 매칭, 강의와 강의 시간의 매칭, 또는 의회 의원들의 해당 위원회로의 매칭 등이 있을 수 있다. 이런 경우 나타나는 그래프들을-모든 선분들이 노드의 두 집합에 연결돼 있는 그래프로 정의된-양분 그래프라 한다. 즉, 노드들은 두 집합으로 나뉘어져 있고 어떤 선분들도 같은 집합에 있는 두 노드와 연결되어 있지 않다.(솔직히 직업 응모자를 또 다른 응모자와, 또한 한 할당된 위원을 또 다른 위원회로 할당하는 것은 원하지 않는다) 양분 그래프의 예는 그림 34.2에 있다. 독자는 이 그래프에서 최대 매칭을 찾는 것에 관심이 있을 것이다.

양분 그래프들에 대한 인접 행렬 표현에서는, 하나의 집합에 대해서는 단지 행들만을, 그리고 또 다른 한 집합에 대해서는 단지 열들만을 포함하므로써 명백한 절약을 할 수 있다. 인접-리스트(list) 표현에서는, 정점들에 대한 이름들을 현명하게 명명함으로써 그 정점이 어느 집합에 속하는지는 쉽게 알 수 있게 하는 것을 제외하고는, 특별한 절약이나 이점이 없다.

우리는 예제들에서, 한 집합에서는 노드들을 글자로 표시하고 또 다른 집합에서는 숫자로 표시했다. 양분 그래프에서 최대 매칭 문제는 “어떤 두 쌍도 같은 문자 또는 숫자를 갖지 않

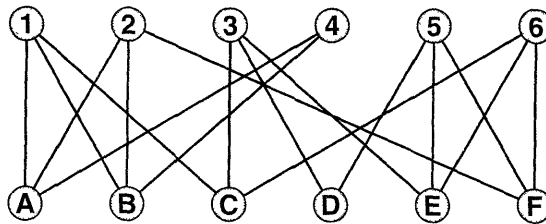


그림 34.2 양분 그래프

는 성질을 갖는 문자-숫자 쌍들의 집합에서 최대의 부분 집합을 찾아라”라는 표현으로 설명될 수 있다. 그림 34.2에서 양분 그래프에 대한 최대 매칭을 찾는 것은, E5 A2 A1 C1 B4 C3 D3 B2 A4 D5 E3 B1들 같은 쌍들에 대해 문제를 푸는 것과 같다.

양분 그래프의 매칭 문제에 대한 직접적인 해결책을 찾는 것은 재미있는 시도이다. 이 문제는 언뜻 보기에는 쉬워 보인다. 그러나 난해함이 금방 눈에 띈 것이다. 확실히 모든 가능성을 시도해 보기에는 너무나 많은 조합들이 있다: 문제에 대한 해결책은 정점들을 매칭시키기 위해 단지 소수의 가능성들만을 가지고 시도하여 해결해야 한다는 것이다.

여기서 우리가 시도하는 해결책은 간접적인 것이다: 한 특정 경우의 매칭 문제를 풀기 위해, 먼저 네트워크 흐름 문제의 한 경우를 구성하고, 전 장에 나온 알고리즘을 사용한다. 그리고 나서 매칭 문제를 풀기 위해 네트워크 흐름 문제에 대한 해결책을 사용한다. 다시 말하면, 네트워크 흐름 문제로 매칭 문제를 축약(reduce) 시키는 것이다. 축약이란 시스템 프로그래머가 라이브러리 서브루틴을 사용하는 것과 다소 비슷한 알고리즘 설계의 한 방법이다. 이것은 고수준의 조합형 알고리즘들의 이론에서 근본적으로 중요하다.(40장을 보라) 당분간, 축약은 양분 매칭 문제에 대한 효율적인 해결책을 제공해 줄 것이다.

형성 방식은 직접적이다. 한 경우의 양분 매칭을 가지고, 양분 그래프 내의 한 집합의 모든 원소들을 가리키는 선분들을 갖는 근원(source) 정점을 생성함으로써 한 경우의 네트워크

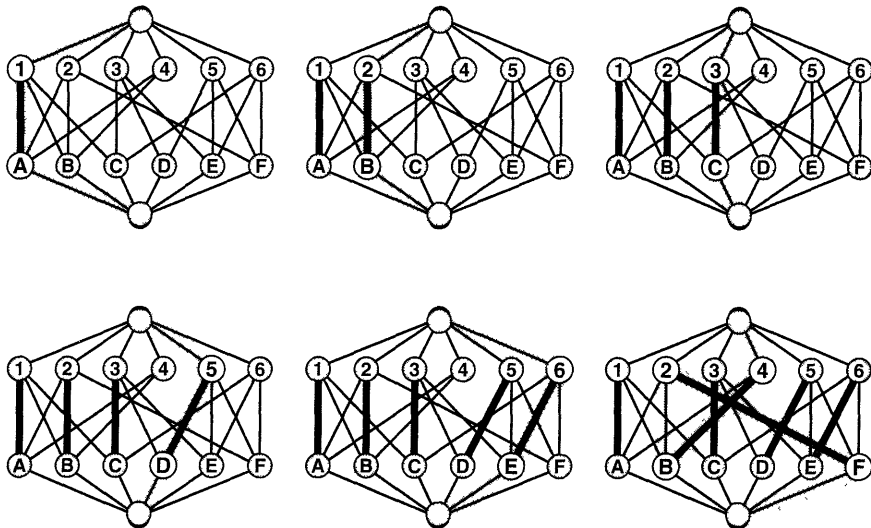


그림 34.3 양분 그래프에서 최대 매칭을 찾기 위한 네트워크 흐름의 사용

흐름을 형성한다. 그리고나서, 양분 그래프내의 모든 선분들이 한 집합에서 또 다른 한 집합을 가리키게 한다. 그다음, 또 다른 집합의 모든 구성 원소들이 가리키는 종착(sink) 정점을 추가한다. 이 결과로 만들어진 그래프의 모든 선분들은 용량 1을 갖는다.

그림 34.3은 그림 34.2의 양분 그래프로부터 한 네트워크 흐름 문제를 형성하고 전 장의 네트워크 흐름 알고리즘을 사용했을 때 어떤 일이 발생하는가를 보여준다. 그래프의 양분 성질, 흐름의 방향, 그리고 용량이 일이라는 사실은 네트워크의 각 경로가 매칭에서의 한 선분에 해당하게 함을 주목하라: 예제에서, 처음 네 스텝(step)에서 발견된 경로들은 A1 B2 C3 D5에 해당한다. 네트워크 흐름 알고리즘이 “search”를 호출할 때마다, 흐름이 1씩 증가되는 경로가 찾아지거나 종료된다.

다섯 번째 스텝에서, 네트워크상의 모든 순방향 경로가 모두 채워졌다. 그러면, 알고리즘은 역방향 선분들을 사용해야만 한다. 이 스텝에서 찾아진 경로가 4B2F이다. 전 장에서 서술한 것처럼, 이 경로는 명백히 네트워크 흐름을 증가시킨다. 현재의 문맥에서는, 이 경로를 현재 상태에서(선분이 하나 더 있는) 부분 매칭을 만들기 위한 명령들의 집합으로써 생각할 수 있다. 이 형성은 경로를 순서에 따라 추적하여 자연스럽게 행해졌다: “4B”는 매칭에 B4를 추가하는 것을, “B2”는 B2를 제거하는 것을, 그리고 “2F”는 F2를 추가하는 것을 의미한다. 이같이, 이 경로가 처리된 후, A1 B4 C3 D5 E6 F2 매칭이 생긴다. 이와 동등하게, 네트워크 흐름은 이 노드들을 연결하는 선분들의 모든 파이프(pipe)들에 의해 주어진다. 알고리즘은 F6을 매칭함으로써 끝난다. 즉 모든 파이프들은 그 근원지를 떠나서 종착지로 모두 들어가므로써 최대 매칭을 갖게된다.

매칭이 최대 흐름 알고리즘에 의한 용량으로 채워진 바로 그 선분들이라는 것의 증명은 간단하다. 첫째로 네트워크 흐름은 항상 정당한 매칭이다. 왜냐하면 각 정점은 들어오거나(incoming) 혹은 나가는(outgoing) 용량 일의 선분 하나만을 갖고, 최대 1의 흐름만 각 정점을 통과할 수 있기 때문이다. 이 말은 각 정점이 매칭에 단 한 번만 포함됨을 뜻한다. 두 번째로, 어떤 매칭도 여러 선분들을 갖지 않는다. 왜냐하면, 그러한 매칭은 최대 흐름 알고리즘에 의해 생성되는 것보다 더 좋은 흐름이 될 것이기 때문이다.

이같이, 양분 그래프에 대한 최대 매칭을 계산하기 위해, 그래프를 단순한 형태로 포맷(format)하여 전 장의 네트워크 흐름 알고리즘에 입력시키기 쉽게 만들었다. 물론, 이런 경우 네트워크 흐름 알고리즘에 주어진 그래프들은, 알고리즘이 처리해야할 일반적인 그래프들보다 훨씬 더 단순하므로, 이 경우 알고리즘이 더 효율적임이 증명되었다.

성질 34.1 양분 그래프에서 최대 매칭은 만약 그래프가 텐스(dense) 그래프라면 $O(V^3)$ 스텝내에, 스파스(sparse) 그래프라면 $O(V(E + V)\log V)$ 스텝내에 찾아진다.

앞서 설명한 형성 방식은 search에 대한 각 호출이 매칭에 하나의 선분을 추가함을 확실하게 한다. 그래서, 알고리즘의 실행중, search 호출이 최대 $V/2$ 번 발생한다. 이같이, 걸린 시간은 31장에서 논의됐던 것처럼, 한 번 검색시 걸리는 시간보다 V 배 큰 시간이 걸린다. □

안정적 결혼 문제(Stable Marriage Problem)

이장의 첫 부분에 주어진 의대생과 병원을 포함한 예제는 명백히 그 참여자들에 의해 심각하게 받아들여졌을 것이다. 그러나, 매칭을 위해 우리가 조사하려는 방법은 다소 묘한 상황에서 아마도 더 잘 이해될 수 있을 것이다. 서로의 선호도를 표현한 N 명의 남자와 N 명의 여자가 있다고 하자.(각 남자는 N 명의 여자 각각에 대해 자신이 느끼는 바를 정확히 말해야 한다) 문제는 모든 사람들의 선호도를 존중해서 N 쌍의 결혼의 집합을 찾고자 하는 것이다.

어떻게 선호도들이 표현되어야 하나? 첫 번째 방법은 각각 1부터 10까지 점수를 매기는 것이다. 남녀 양측은 상대측의 특징인들에 대해 절대 점수를 부여한다. 이 방법은 풀기가 상대적으로 어려운 가중치 매칭 문제와 동일하게 만든다. 게다가, 절대 점수를 사용하는 그 자체가 부정확성을 야기시킬 수 있다. 왜냐하면 각 사람마다 점수의 기준이 다를 수 있기 때문이다.(한 여자의 10은 다른 여자의 7이 될 수 있다) 선호도의 설명을 위한 보다 자연스러운 방법은 각 개인으로 하여금 이성 모두에 대한 선호도 목록을 갖게 하는 것이다. 그림 34.4는 각 다섯 명의 남녀 집합 사이에서 나타나는 선호도 목록을 보여준다. 여기서 우리는 해싱

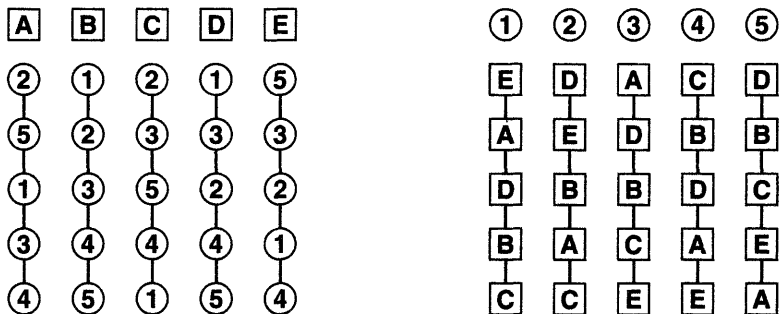


그림 34.4 안정적 결혼 문제에 대한 선호도-목록

(hashing) 혹은 다른 방법이, 실제 이름 대신 여자에 대해서는 한 숫자를 남자에 대해서는 한 글자를 부여하기 위해 사용됐다고 가정한다.

이러한 선호도에서는 종종 불일치가 생긴다: 예를 들면, A와 C 모두가 일 순위로 2를 택하고, 어느 누구도 일 순위로 4를 택하지 않았다.(그러나 누군가가 그녀와 짝이 되어 한다) 문제는 그들의 선호도를 존중해 주는 방법으로 가능한 한 많은 남녀를 짝지어 주고, 그리하여 거대한 축제로서 N 쌍의 결혼식을 올리는 것이다. 해결책을 만들 때, 선호도에서 일 순위가 아닌 사람과 짝을 맺는 사람들은 실망을 할 것이고, 가능하면 선호도가 높은 사람과 짝지어지기를 원할 것이다. 결혼 쌍들의 집합에서 결혼하지 않은 두 사람이 자신의 배우자보다는 서로를 선호한다면 그 집합은 불안정(unstable)하다고 한다. 예를 들어, A1 B3 C2 D4 E5의 경우, A는 1보다는 2를 선호하고 2는 C보다 A를 선호하기 때문에 불안정이다. 따라서, 그들은 그들의 선호도에 따라 행동할 것이고, 그 결과 A는 1을 떠나 2에게, 2는 C를 떠나 A에게 갈 수 있다.(1과 C는 서로 결합하는 것 외에는 선택의 여지가 거의 없다)

안정적 구조를 발견한다는 것은 표면상 어려운 문제처럼 보인다. 왜냐하면 너무 배열의 가능성이 많기 때문이다. 위의 예제에서 A2와 C1이 새로이 매치가 된 후에도 독자들이 불안정한 쌍을 찾을 수 있는 것처럼, 구조가 안정적인지 아닌지를 아는거 또한 간단치 않다. 일반적으로, 주어진 선호도-목록들의 집합에는 많은 안정적인 배열이 있다. 거기서 우리는 단지 하나만 찾으면 된다.(모든 안정적 배열을 찾는 것은 훨씬 더 어려운 문제이다)

안정적 구조를 찾는 한 가지 방법은 불안정한 쌍을 한 번에 하나씩 제거하는 것이다. 그러나, 이 처리는 안정성을 결정하는데 시간이 많이 걸릴 뿐만 아니라 반드시 종료된다는 보장도 없다. 예를 들어, 위 예에서 A3와 C1이 매치된 후, B와 2는 불안정한 쌍이 되고 구조는 A3 B2 C1 D4 E5가 된다. 이 경우, B와 1은 불안정한 쌍이 되고 구조는 A3 B1 C2 D4 E5가 된다. 마지막으로, A와 1은 원래 배치로 되돌아가서 불안정한 배치가 된다. 하나씩 불안정한 쌍을 제거하는 이 알고리즘은 이런 형태의 루프(loop)를 갖게 된다.

이 대신, 다소 이상적인 “실 생활”에 발생하는 것에 근거한 방법을 사용해 체계적으로 안정된 쌍을 찾아가는 알고리즘을 알아보자. 이 방식은 각 남자를 구혼자가 되게 하여 신부를 찾는 것이다. 명백히 남자는 그의 목록의 첫 번째 여자에게 청혼할 것이다. 만약 그녀가 이미 자신이 선호하는 남자와 약혼했다면, 그는 목록의 다음 번 여자에게 청혼할 것이다. 이 행위는 목록에 있는 여자 중 그를 좋아하는 여자나 약혼자보다 그를 더 좋아하는 여자를 찾을 때까지 시도한다. 만약 여자가 약혼하지 않았다면 그는, 그 여자와 약혼하게 되고, 다음번 남

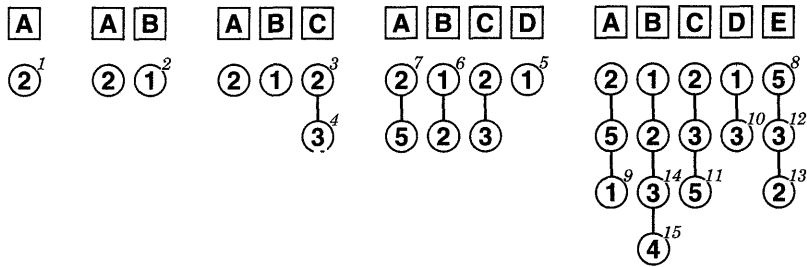


그림 34.5 안정적 결혼 문제의 해결

자가 구혼자가 된다. 만약 여자가 약혼했었다면, 그녀는 약혼을 깨고(그녀가 선호하는) 그 구혼자를 선택한다. 따라서 그녀의 이전 약혼자는 약혼하지 않은 상태가 되어 다시 구혼자 목록에 들어간다. 결국, 그는 새로운 약혼녀를 발견하게 되나 이로 인해 다른 약혼들이 깨질 수 있다. 이 방법은 구혼자가 아직 약혼하지 않은 여자를 찾을 때 까지 계속된다; 물론 필요하다면, 약혼을 깨뜨릴 수 있다.

이 방법은 19세기 소설 속에서나 있을 법한 모델이다 그러나, 이 방법이 안정적인 집합을 만드는 것을 보여 주려면 조심스러운 검토가 필요하다. 그림 34.5는 우리 예제의 초기 처리 단계에 대한 사건 순서를 보여 준다. 먼저, A는 2에게 청혼하여(그의 첫 번째 선택자인) 이루어 졌고, B가 1에게 청혼하여 이루어 졌다. 그리고 세 번째 다이어그램에 있는 것처럼, C는 2에게 청혼했으나 거절당한 후 3에게 청혼하여 이루어 졌다.

각 다이어그램은 새로운 남자들이 약혼녀를 찾기 위한 구혼자로 등장할 때의 사건 순서이다. 각 선(line)은 해당 남자에 대해 “사용된” 선호도-목록을 알려주며 각 연결(link)은 여자에게 청혼하기 위해 남자가 언제 그 연결을 사용했는지를 알려주는 값을 가지고 레이블(label)되어 있다. 이 정보는 D와 E가 구혼자가 되었을 때 청혼의 순서를 추적하는데 유용하다: D가 1에게 청혼했을 때 1은 B보다 D를 선호하므로 첫 번째 파혼이 있었음을 알 수 있다. 그때 B가 구혼자가 되고 2에게 청혼을 한다. 그러나 2는 A보다 B를 선호하므로 두 번째 파혼이 발생했음을 알 수 있다. 그 다음 A가 구혼자가 되고 5에게 청혼을 하게 되고 안정된 상태에 있게 된다. 그러나, 이 안정성은 단지 임시적일 뿐이다. 독자는 E가 구혼자가 되었을 때 만들어진 청혼의 순서를 통한 추적을 원할 것이다. 결과를 보면, 여덟 번의 청혼 후에도 안정화되지 못했음을 알 수 있다. 이 과정에서 E는 두 번 구혼자가 되었음에 주목하라.

구현의 첫 스텝은 선호도 목록을 위해 사용될 자료 구조를 설계하는 것이다. 이들은 모두 추상적 자료 구조인 단순한 선형-리스트이다. 그러나, 3장과 다른 곳의 예제에서 배운 것처럼

럼, 표현의 적절한 선택은 성능에 직접적인 영향을 줄 수 있다. 이 경우 또한 남녀에 대해 다른 구조들을 사용함이 적당하다. 왜냐하면 그들의 선호도-목록 사용 방식이 다르기 때문이다.

남자들은 단순히 그들의 선호도-목록을 순서대로 찾아 나간다. 그래서 선형-리스트 실행이 사용된다. 선호도-목록은 모두 길이가 같기 때문에 이차원 배열로써 간단히 구현할 수 있다. 예를 들어, `prefer[m][w]`는 m 번째 남자의 선호도-목록의 w 번째 여자이다. 게다가, 우리는 각 남자가 자신의 목록 상에서 얼마나 진행했는지를 추적할 필요가 있다. 이것은 0(zero)으로 초기화된 일차원 배열인 `next`와 남자 m 의 선호도-목록 상의 다음번 여자에 대한 인덱스 `next[m]+1`을 가지고 처리할 수 있다. 이 여자의 id는 `prefer[m][next[m] + 1]`에서 찾아진다.

각 여자의 경우에는, 여자의 약혼자를(`fiancee[w]`는 여자 w 와 약혼자 남자이다) 추적할 필요가 있다. 그리고 “남자 s 가 `fiancee[w]`를 선호하는가?”라는 질문에 대한 답을 알 필요가 있다. 이것은 s 혹은 `fiancee[w]`를 발견할 때까지 순차적으로 선호도-목록을 찾아야 한다. 그러나 이 방법은, 그들이 둘 다 끝 쪽에 가깝다면, 오히려 비효율적일 수 있다. 바람직한 것은 선호도-목록의 “역”을 취하는 것이다. `rank[w][s]`는 여자 w 의 선호도-목록의 남자 s 에 대한 인덱스이다. 위 예제의 경우, A 는 1의 선호도-목록 상의 두 번째이기 때문에 `rank[1][1]`은 2이다. 또한 D 는 5의 선호도 목록 상의 네 번째이기 때문에 `rank[5][4]`는 1이다.

구혼자 s 의 안정성은 `rank[w][s]`가 `rank[w][fiancee[w]]`보다 작은지 아닌지를 검사함으로써 매우 빨리 결정할 수 있다. 이 배열은 선호도-목록으로부터 직접 쉽게 형성된다. 첫 구혼자로서 “표지값” `man 0(zero)`을 사용하고, 모든 여자들의 선호도-목록 제일 끝에 그를 위치시킨다.

이 방법으로 초기화된 자료 구조를 갖는, 위 설명한 것에 대한, 구현은 다음과 같다.

```
for (m = 1; m <= N; m++)
{
    for(s = m; s != 0; )
    {
        next[s]++; w = prefer[s][next[s]];
        if (rank[w][s] < rank[w][fiancee[w]])
        { t = fiancee[w]; fiancee[w] = s; s = t; }
    }
}
```


이 구현에서 각 반복(iteration)은 약혼 안한 남자로 시작해서 약혼 안한 여자로 끝난다. 안쪽 루프는 모든 남자의 목록이 모든 여자를 포함하고 이 루프의 각 반복은 남자 목록의 인덱스를 증가시키므로 종료되어야 한다. 그리고 이같이 하여 약혼 안한 여자들과 남자 목록이 끝나기 전에 만나게 된다. 이 알고리즘에 의해 만들어진 약혼의 집합은 안정적이다. 왜냐하면 한 남자가 약혼녀보다 더 선호하는 여자는 그 여자가 더 선호하는 남자와 약혼했기 때문이다.

성질 34.2 안정적 결혼 문제는 선형 시간 내에 풀 수 있다.

방금 언급한 것처럼, 각 반복문은 남자의 선호도-목록을(인덱스를) 증가시킨다. 최악의 경우, 모든 목록의 엔트리들(entries)이 검사된다.(그러나 두 번 검사된 엔트리는 없다) 사실, 알고리즘은 목록들을 작성하는 시간보다 훨씬 적은 시간을 소모할지도 모른다. 왜냐하면 안정적 구조는 모든 목록들의 검사가 완료되기 한참 전에 찾아 질 수도 있기 때문이다. 이런 종류의 문제는 흥미로운 분석 문제들을 야기시킨다. □

이 알고리즘에는 내재된 몇 개의 명백한 바이어스(bias)가 있다. 첫째, 남자들은 목록의 순서대로 여자들을 찾는다. 반면에, 여자들은 “원하는 남자”를 기다린다. 이런 바이어스는 선호도-목록들이 입력되는 순서를 바꿈으로써 교정되어야 한다.(실제보다는 좀 더 쉬운 방법으로) 이것은 안정 구조 1E 2D 3A 4C 5B를 형성하는데, 여기서 모든 여자는 5번을 제외하고는 첫 번째 선택을 얻게 된다. 일반적으로, 많은 안정적인 구조가 있다: 이것은 그녀 자신의 목록에서 더 나은 선택을 할 수 없다는 의미에서, 여자들에 대한 “최적”임을 보여준다.(물론 예제에서 첫 번째 안정적인 구조는 남자들에 대해 최적이다) 알고리즘에서 또 하나의 바이어스는 남자가 구혼자가 되는 순서이다:

청혼하는 첫 번째 남자가 되는 것이 좋은가?(따라서 잠시만이라도 첫 번째 선호하는 여자와 약혼하는) 아니면 마지막 남자가 되는 것이 좋은가?(따라서 파혼의 모욕을 되도록 줄이든지) 사실 이 문제에 대한 답은 전혀 바이어스가 없다는 것이다. 남자들이 구혼자가 되는 순서는 문제될 것이 없다. 각 남자들이 청혼을 하고 각 여자들이 자기의 목록에 따라 응답하는 한, 동일한 안정적 구조가 나올 것이다.

고수준 알고리즘(Advanced Algorithms)

우리가 살펴볼 두 개의 특별한 경우는 매칭 문제가 얼마나 복잡한지를 어느 정도 보여주고 있다. 이들 알고리즘들은 여러 실제 응용 분야에서 유용하다. 하지만, 지적인 바와 같이, 대부분 다른 분야에서는 보다 일반적인 문제의 해결책을 필요로 한다.

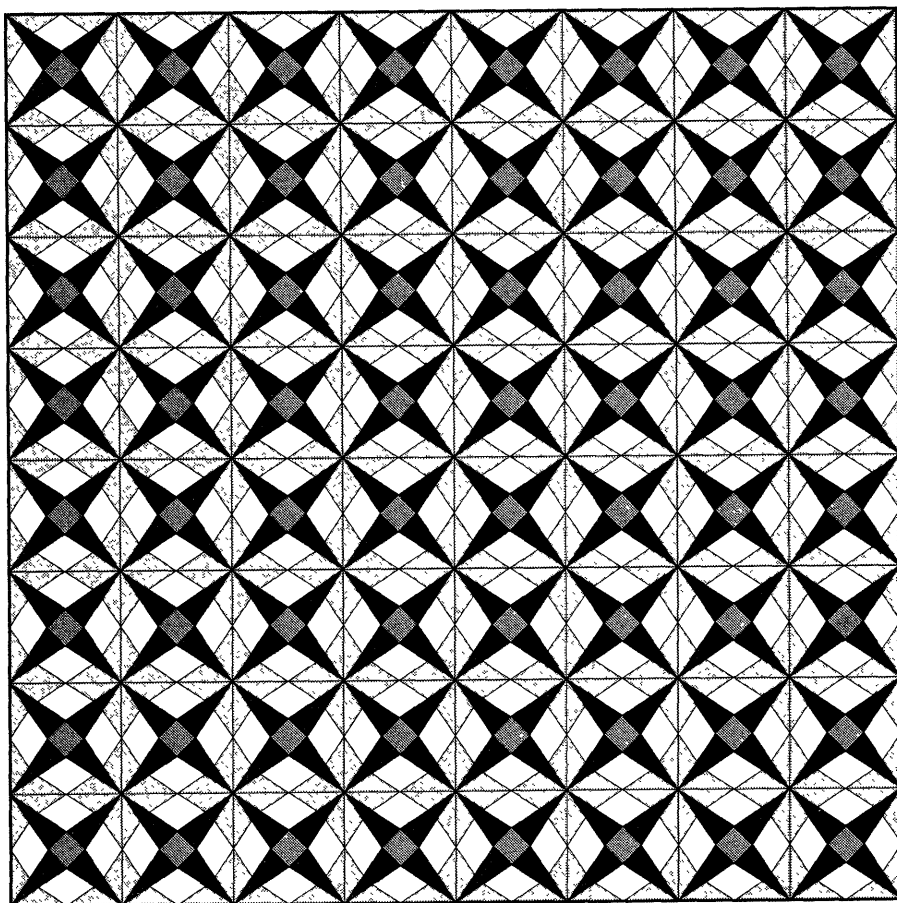
보다 일반적인 문제 중에는 자세히 연구해 볼 것들이 있다: 예를 들면, 일반적인(반드시 양분이 되어질 필요는 없는) 그래프에 대한 최대 매칭 문제; 선분들에 가중치가 있고 통합 최대 가중치를 가진 매칭이 찾아지는, 양분 그래프들에 대한 가중치 매칭; 그리고 일반적인 그래프에 대한 가중치 매칭이 있다.

양분 그래프들에 대한 가중치 매칭과 이와 유사한 일반화된 문제들은 일반화된 네트워크 흐름 문제에 대한 알고리즘들이 알려진 정도 만큼(에 따라) 처리 될 수 있다. 그러나, 일반적 그래프는 아주 다르다.(안정적 결혼 문제는, 그 문제를 정제함으로써, 일반적 그래프들에 대한 가중치 매칭 문제를 피하는 방식으로 특징지어질 수 있다) 일반적 그래프들에서의 매칭에 대해 시도해온 많은 기술들을 다루면 이 책 전체를 채울 수도 있다: 이 문제는 그래프 이론에서 가장 광범위하게 연구해 볼 문제 중의 하나이다.

연습문제

1. 그림 34.2에 있는 양분 그래프에 대해 다섯 개의 선분을 갖는 모든 매칭을 찾아라.
2. 50개의 정점과 100개의 선분을 갖는 무작위 양분 그래프들에 대해 본문에 있는 알고리즘을 사용하여 최대 매칭을 찾아라. 이 매칭들에는 얼마나 많은 선분들이 있는가?
3. 세-선분(three-edges) 매칭을 갖는, 6개의 노드와 8개의 선분으로 구성된 양분 그래프를 형성하라. 만약 없다면, 없음을 증명하라.
4. 양분 그래프의 정점들이 직업과 사람을 나타내고 각 사람은 두 가지 직업에 할당되어야 한다고 가정하자. 네트워크 흐름에 대한 축약은 이 문제에 대한 알고리즘을 제공해 주는가? 증명하라.
5. 33장의 네트워크 흐름 프로그램을 양분 매칭에서 발생하는 0-1 네트워크들의 특수한 구조의 장점을 이용하여 변경하라.
6. 결혼 문제에 대한 할당이 안정적인지 아닌지를 찾는 효율적인 프로그램을 작성하라.
7. 두 남자가 안정적 결혼 문제에서 그들의 마지막 선택자를 택하는 것이 가능한가? 증명하라.
8. 안정적 결혼 문제에서, 모든 사람이 그들의 두 번째 선택자를 택하는 $N = 4$ 인 선호도-목록들의 집합을 구성하라. 아니면 그러한 집합이 존재하지 않음을 증명하라.
9. 남녀 모두의 선호도-목록이같은 경우, 안정적 결혼 문제에 대한 안정적 구조를 보여라. (단, 오름차순으로)
10. 선호도-목록들에 대해 무작위로 순서를 바꾸면서(permutation), $N = 50$ 인 경우, 안정적 결혼 문제를 실행시켜라. 알고리즘의 실행 동안 얼마나 많은 청혼이 있겠는가?

수학적 알고리즘들



빈 면

다음 알고리즘들은 컴퓨터가 난수를 생성하는 방법들이다. 이 교재의 여기저기에서 난수를 접해 왔지만, 우선 난수가 무엇인지에 대한 보다 정확한 개념을 알아보기로 한다.

종종 대화시, 사람들은 “임의(arbitrary)”를 뜻할 때, 무작위(random)란 용어를 사용한다. 임의의 수가 무어나고 묻는다면, 사람들은 보통 “언어진 수가 어떤 수든지 간에 상관이 없는 수”라고 말한다: 거의 모든 수가 그럴 것이다. 이와는 대조적으로, 난수(random number)는 정확하게 정의된 수학적 개념이다. 한 난수는 한 임의의 수를 필요로 하는 사람을 만족시켜 주나, 그 반대의 경우는 그렇지 않다.

“모든 수는 동일한 발생 확률을 갖는다”는 말이 타당성이 있으려면, 수를 어느 제한된 유한 영역에서 사용되도록 제한해야 한다. 우리는 일정 범위 내에서만의 정수인 난수를, 그리고 일정 범위 내에서 어느 정도의 정밀도(precision)만을 갖는 실수인 난수를 얻을 수 있다.

거의 대부분의 경우, 단지 난수 한 개가 아니라, 연속적인 난수가 필요하다.(그렇지 않다면, 임의의 수만으로도 일을 처리할 수도 있다) 따라서, 수학이 생겨났다: 연속적 난수들에 관한 성질들에 대한 많은 사실들의 증명이 가능하다. 예를 들면, 작은 숫자 영역에서 난수가 많이 발생하면, 각 숫자가 거의 동일한 횟수만큼 나타나리라는 것을 예측할 수 있다. 난수 순서(random sequence)는 많은 자연적인 현상들을 모델화했는데, 이에 대한 많은 성질들이 알려져 있다. 이제까지와 마찬가지로 일관성을 위해, 난수 순서를 난수라고 부르기로 한다.

컴퓨터 상에서 진짜 난수를 만들 수는 없다. 프로그램이 일단 작성되면, 이 프로그램들이 만들어 내는 숫자들의 추론이 가능하다. 우리가 할 수 있는 최선의 방법은, 난수는 아니나 난수와 동일한 성질들을 많이 가진 수들을 연속적으로 만드는, 프로그램을 작성하는 것이다. 그

런 수들은 실제로는 난수는 아니나 난수에 대한 근사치로서 유용할 수 있다.(부동 소수점 숫자가 실수에 대한 근사치로서 유용한 것처럼) 그래서 이같은 수들의 연속을 의사난수(pseudo random number)라고 한다.(가끔은 구분을 조금 더 세밀하게 하는 것이 더욱 편리할 때가 있다: 어떤 경우에는 난수들의 일부 성질들이 아주 중요한 반면 또 어떤 경우에는 별 관련이 없다. 이런 경우에는, 필요 성질들만 갖고 나머지는 별로 없게 만든 준난수(quasi-random number)를 만들 수 있다. 일부 응용분야에서는 준난수가 의사난수보다 더 선호될 수가 있다)

수의 긴 연속에서 “각 수가 동일 발생 확률을 갖는다”는 성질에 준함은 충분치 않음을 쉽게 알 수 있다. 예를 들면, 1부터 100까지의 범위 내의 각 수가 단 한 번씩 1, 2, ..., 10의 순서로 나타났다고 하자. 그러나 이 순서는 난수 순서에 대한 근사치로서 유용할 수는 없다. 사실, 1부터 100까지의 범위 내의 길이가 100인 난수 순서에서, 몇몇 숫자는 한 번 이상 나타나거나 또는 한 번도 나타나지 않을 것이다. 만약 일련의 의사난수들이 이런식으로 발생하지 않는다면, 난수 생성기에 이상이 있다고 봐야 한다. 이같은 특정 관측들에 근거한 많은 복잡한 방법들이, 한 긴 순서의 의사난수가 난수가 갖는 어떤 성질을 갖고 있는지를 시험하기 위해, 난수 생성기들을 위해 고안됐다. 우리가 공부할 난수 생성기들은 이러한 시험들을 잘 통과한 것이다. 이 장에서는 가장 중요한 시험들 중의 하나인 X^2 (chi-square) 시험에 대해 자세히 살펴 볼 것이다.

우리는 각 값들의 발생 확률이 거의 같은 균등한(uniform) 난수에 대해서만 언급해 왔고 앞으로도 그럴 것이다. 또한 몇몇 값들이 다른 값들보다 발생 확률이 큰 다른 분포에 따르는 난수들을 다룰 것이다. 불균등한(non-uniform) 분포를 갖는 의사난수들은 대개 균등하게 분포된 수들에 대해 어떤 연산을 수행함으로써 얻어진다. 이 책에 있는 대부분의 응용 분야들은 균등한 난수(uniform random number)들을 사용한다. 앞으로 알게 되겠지만, 만든 수들이 난수들의 “모든” 성질들을 갖는다고 확신하기는 어렵다; 이 문제는 다른 유형의 분포들이 있을 때 더욱 심각해진다.

응용 분야들(Applications)

이 책에서 난수들이 유용하게 쓰이는 많은 분야들을 보아 왔다. 그 중 일부를 여기서 요약해 본다. 한 확실한 응용 분야는 암호화(cryptography)인데, 여기서의 주된 목적은 메시지를 코드화하여 허가 받은 수신인만 읽을 수 있고 그외의 어느 누구도 읽지 못하게 하는 것이다. 23장에서 본 것처럼, 이것이 하는 일은 메시지를 난수처럼 보이게 하는 것이다.(코드화 할 때

허가 받는 수신인이 암호해독에 사용하는 것과 똑같은) 의사난수 순서를 이용하여 난수처럼 보이게 하는 것이다.

난수가 폭넓게 사용되는 또 다른 분야는 모의실험(simulation)이다. 한 전형적인 모의실험은 실세계의 어느 측면을 모델화하는 큰 프로그램이다: 난수들은 당연히 이런 프로그램들에 대한 입력으로 사용된다. 비록 진짜 난수들이 필요한 것은 아니나, 모의실험들은 전형적으로 입력으로서 많은 임의의 수를 필요로 하고, 이들은 난수 생성기에 의해 쉽게 만들어 진다.

방대한 양의 데이터 분석이 필요할 때, 무작위로 표본 추출(sampling)된 적은 양의 데이터 만의 처리로도 충분할 경우가 종종 있다. 이러한 응용 분야들은 아주 많다. 그 중에서도 가장 두드러진 것은 국가 정치에 대한 여론 조사이다.

고려 중에 있는 모든 인자들이 같아 보일 때, 선택을 해야 하는 경우가 있다. 70년대의 국가 발행 복권이나 학생들에게 기숙사 방을 배정할 때 학교에서 사용하는 메커니즘들은, 의사 결정을 위해 난수들을 사용한 예이다. 이런 식으로, 결정의 책임은 “숙명적”이거나 혹은 컴퓨터에 의해 좌우된다.

독자들은 프로그램들에 무작위적의 또는 임의의 입력 값을 주기 위해 난수를 모의실험에 널리 사용함을 알았을 것이다. 또 다른 사용의 예는 표본 추출을 위해 또는 의사결정의 지원을 위해 난수가 사용된 알고리즘들이다. 이에 대한 주된 예제는 Quicksort(9장)와 Rabin-Karp 문자열 검색 방법(19장)이다.

선형 조화 방법(Linear Congruential Method)

난수 생성을 위한 가장 널리 알려진 방법은 선형 조화 방법(linear congruential method)으로서 1951년 D. Lehmer에 의해 소개된 이후 지금까지 사용되고 있다. 만약 시드(seed)가 임의의 값을 가졌다면, 아래 프로그램은 이 방법을 사용해 N개의 난수들로 배열을 채울 것이다.

```
a[0] = seed;
for(i = 1; i <= N; i++)
    a[i] = (a[i - 1] * b + 1) % m;
```

즉, 새로 얻어진 난수는, 이 전 값을 상수 b와 곱하고 거기에 1을 더한 다음 상수 m으로 나눈, 나머지이다. 그 결과 값은 0과 m - 1 사이의 정수이다. % 연산자는 컴퓨터에서 사용하기가 좋다. 왜냐하면 구현하기가 매우 쉽기 때문이다: 만일 산술 연산에서 오버플로우

(overflow)를 무시한다면, 대부분의 컴퓨터 하드웨어가 오버플로우된 비트를 버릴 것이고, 그래서 % 연산을 컴퓨터 워드 상에서 표현될 수 있는 최대 정수 값보다 1 많은 m 을 가지고, 효과적으로 수행한다. 다시 말하자면, 사용된 숫자들은 무작위가 아니다; 이 프로그램은 우리가 어느 다른 과정에서 무작위로 나타나기를 바라는 그런 수들을 만들어 낸다.

보기에는 단순하나, 선형 조합 난수 생성기는 상당히 자세하고 복잡한 수학적인 분석-연구 과제가 되어왔다. 이 분석 작업은 상수들인 시드 b 와 m 을 선택하는데 지침이 된다. 일부 “상식적” 원리들이 적용되나, 이런 경우 상식적 원리는 좋은 난수를 만들기에는 충분치 않다. 우선, m 값은 커야 한다: 앞서 말한 것처럼, 컴퓨터의 워드 크기이거나 필요치 않으면 그보다 작을 수도 있다. 2^{20} 또는 10^{20} 이 보통 적당하다. 두 번째, b 를 너무 크게 또는 너무 작게 하지 말아야 한다. 적당한 값은 m 보다 자리수가 하나 작은 값이다. 세 번째, b 는 그 구성 숫자가 특정 패턴(pattern)이 없는 임의의 상수여야 한다. 물론 b 가 $\dots x21$, 짝수 개의 x 를 가지고 끝나는 경우는 예외로 한다. 이 마지막 요구는 좀 특별하나, 수학적 분석에 의해 발견안된 오동작들의 발생 가능성을 막아준다.

위에 기술한 규칙들은 D.E. Knuth에 의해 개발되었는데, 그의 책에서 더 자세히 다루고 있다. Knuth는 이런 선택들이 선형 조합 방법으로 하여금 몇몇 복잡한 통계 시험들을 통과하는 좋은 난수들을 생성해 줄을 보여주고 있다. 잠재적인 가장 심각한 문제는 생성기가 사이클내에서 동작함으로써 숫자들을 생성할 시점보다 훨씬 더 빨리 생성하는 것이다. 예를 들어, $seed = 0, b = 19, m = 381$ 인 경우는 별로 무작위가 아닌 0부터 380 사이의 숫자들의 순서인 0, 1, 20, 0, 1, 20, ...을 생성한다. 불행히도, 이런 문제점들을 찾아내기는 쉽지 않으며, 그래서 사람들은 Knuth가 제안한 지침들을 따르도록 권장된다. 이같이 하여 그가 발견한 많은 어려운 트랩(trap)들을 피할 수 있다.

어떤 값도 별 영향없이 난수 생성기를 시작하는데 이용될 수 있다.(물론, 다른 초기 값을 사용하면 다른 난수 순서가 만들어 진다) 종종, 위 프로그램에서와 같이, 모든 난수 문서를 저장할 필요는 없다. 오히려, 우리는 어떤 값으로 초기화되고 $(a * b + 1) \% m$ 의 연산 결과에 따라 변경되는 전역변수(global variable) a 를 유지한다.

C++(그리고 다른 많은 프로그래밍 언어들)에서, 실제 동작하는 프로그램을 구현하려면 오버플로우를 무시할 수 없기 때문에 한 스텝을 더 고려해야 한다: 이 오버플로우는 예기치 않은 결과를 가져오는 오류의 조건이다. 컴퓨터가 32비트 워드를 갖고 $m = 100000000, b = 31415821$ 이고, 초기 값으로 $a = 1234567$ 이라 가정하자. 이 모든 값들은 표현될 수 있는 최

대 정수 값보다 작으나, 첫 번째 $a * b + 1$ 연산에서 오버플로우가 발생한다. 오버플로우를 야기시키는 곱셈부는 이 계산과는 관계가 없다—우리는 단지 마지막 여덟 자리 숫자에 관심이 있다. 오버플로우를 피하는 한 가지 조작 기법은 곱셈을 부분 부분 나누어 하는 것이다. $p = 10^4 p_1 + p_0$ 이고 $q = 10^4 q_1 + q_0$ 일 때 p 와 q 의 곱셈은 다음과 같다:

$$\begin{aligned} pq &= (10^4 p_1 + p_0)(10^4 q_1 + q_0) \\ &= 10^8 p_1 q_1 + 10^4 (p_1 q_0 + p_0 q_1) + p_0 q_0 \end{aligned}$$

우리는 결과로서 마지막 여덟자리 숫자만 원하므로, 첫째 항과 둘째 항의 처음 네 숫자를 무시할 수 있다. 이 결과로서 나온 프로그램은 다음과 같다:

```
# include <iostream.h>
const int m = 100000000;
const int m1 = 10000;
int mult(int p, int q)
{
    int p1, p0, q1, q0;
    p1 = p / m1; p0 = p % m1;
    q1 = q / m1; q0 = q % m1;
    return ((p0 * q1 + p1 * q0) % m1) * m1 + p0 * q0 % m;
}
class Random
{
private:
    int a;
public:
    Random(int seed)
        { a = seed; }
    int next()
        { const int b = 31415821;
          a = (mult(a, b) + 1) % m;
          return a;
        }
};
main()
```

```

{
    int i, N; Random x(1234567);
    cin >> N;
    for(i = 1; i <= N; i++)
        cout << x.next() << ' ';
    cout << "\n";
}

```

이 프로그램에서 mult 함수는, m 이 표현될 수 있는 최대 정수 값의 $1/2$ 보다 작은 한, 오버플로우 없이 $p \cdot q \% m$ 을 계산한다. 이 기법은 다른 m_1 값에 대해 $m = m_1 \cdot m_1$ 으로서 적용될 수 있다.

입력 $N = 10$ 이고 $a = 1234567$ 일 때, 이 프로그램은 다음 10개의 수를 출력한다; 35884508, 80001069, 63512650, 43635651, 1034472, 87181513, 6917174, 209855, 67115956, 59939877. 이 수들은 다소 난수적이 아니다: 예로서, 이 수들의 마지막 자리 수는 0부터 9까지의 숫자에서 순환한다. 이 공식에서 이런 현상이 발생함을 증명하는 것은 쉽다. 일반적으로, 오른쪽 자리의 수들이 무작위가 아니라는 사실은 이런 유형의 난수 생성기의 사용시 공통적이고 심각한 오류의 근원이다. 응용분야에서는 전형적으로 범위 $[0, r)$ 의 난수를 원하는 데, 여기서 r 은 응용분야에 따른 정수이고 위 프로그램의 M 처럼 난수 생성기에 쉽게 구현될 수 없다.

다음은 작은 범위 내에서 난수들을 생성하는 잘못된 프로그램의 한 예이다.

```

int badnext(int r)
{ const int b = 31415821;
  a = (mult(a, b) + 1) % m;
  return a % r;
}

```

여기서 사용된 숫자들은 오른쪽에 있는 비난수(non-random) 숫자들 뿐이다. 그래서, 결과적으로 나타난 난수의 순서는 원하는 성질들을 거의 갖고 있지 않다. 이 문제는 왼쪽 자리의 수를 대신 사용하여 쉽게 고쳐질 수 있다. 우리는 $(a \cdot r) / m$ 을 계산함으로써 0과 $r-1$ 사이의 수를 얻기를 원하나, 아래에 구현된 것처럼, 오버플로우가 방지되어야 한다.

```

int next(inr r)
{ const int b = 31415821;
  a = (mult(a, b) + 1) % m;
  return ((a / m1) * r) / m1;
}

```

이 프로그램으로, $r = 2$ 를 취해 이진 스트림(stream)을 얻거나, $r = 10$ 을 취해 얻거나, 또는 $r = N$ 을 가지고 퀵 정렬(Quicksort)의 분할 요소를 얻을 수 있는 등의, 여러 가지가 행해질 수 있다.

일반적으로 이용되는 또 하나의 기술은 위 수들을 소수점 아래자리 숫자로 간주함으로써 0과 1 사이의 실수인 난수를 생성하는 것이다. 이것은 정수 a 보다는 실수 값 a/m 을 리턴함으로써 쉽게 구현될 수 있다. 그러면 사용자는 이 값에 r 을 곱해서 범위 $[0, r)$ 내의 정수를 구하고, 가장 가까운 정수로 자리내림(truncation)을 한다. 또는 정확히 요구되는 0과 1 사이의 실수인 난수도 얻을 수 있다.

부가적 조화 방법(Additive Congruential Method)

난수들을 생성하기 위한 또 한 가지의 방법은 초기 암호화 기계(encryption machine)에서 사용한 선형-피드백-쉬프트-레지스터(linear feedback shift register)를 기본으로 한 것이다. 방법은 모두 0은 아닌 어떤 임의의 패턴으로 채워진 레지스터를 가지고 시작하여, 한 번에 한 스텝씩 오른쪽으로 쉬프트한다. 그래서 그 레지스터의 내용에 따라 정해진 비트를 가지고 왼쪽부터 빈자리를 채운다.

그림 35.1은 오른쪽 마지막 두 비트들을 “exclusive or”해서 얻어진 새로운 비트를 갖는, 간단한 4-비트 선형-피드백-쉬프트-레지스터를 보여준다. 예를 들어, 만약 레지스터가 1111 패턴으로 초기화되었다면, 첫 스텝 후 0111이 될 것이다: 111 비트들이 하나씩 오른쪽으로 쉬프트되고 최좌측 비트가 0이 된다. 왜냐하면 최우측의 두 비트가 같기 때문이다. 계속하면, 레지스터의 값이 0011, 0001, 1000, 0100, 0010, 1001, 1100, 0110, 1011, 0101, 1010, 1101, 1110, 1111이 된다. 일단 초기 값으로 되돌아오면, 과정의 순환이 계속된다.

0이 아닌 모든 가능한 비트 패턴이 발생함에 주목하라: 시작 값은 매번 15스텝 후에 반복된다. 그러나, 우리가 피드백을 위해 사용된 비트들인 “탭(tap)”의 위치들을 0과 1에서(오른편에서부터 볼 때) 0과 2로 바꾼다면, 1111, 0111, 0011, 1001, 1100, 1110, 1111이 나오는데 이

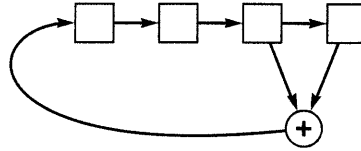


그림 35.1 4 비트 선형 피드백 쉬프트 레지스터

것은 완전한 사이클(full cycle)은 아니다. 우리는 항상 완전한 사이클을 얻는 것이 보장되기를 원한다.

일반적으로, n -비트 선형-피드백-쉬프트-레지스터에 대해, 배열의 사이클 길이가 $2^n - 1$ 이 되게 배열하는 것이 가능하다. 따라서, 큰 n 의 경우, 이같은 레지스터들은 좋은 난수 생성기를 만들어 준다. 전형적으로, $n = 31$ 이나 $n = 63$ 을 사용한다. 선형 조화 방법에서 처럼, 이런 레지스터들의 수학적 성질들은 광범위하게 연구되어왔다. 예를 들면, 다양한 크기의 레지스터들에 대해 모든 비트 패턴을 생성할 수 있도록 “탭” 위치를 선택하는 방법이 많이 알려져 있다. 예를 들어, $n = 31$ 일 때, 탭 위치로서 0과 4, 7, 8, 14, 19, 25, 26 또는 29 중의 하나를 선택하면 가능할 수 있다.

레지스터의 연속적인 내용은 난수 순서로는 유용하지 못하다. 왜냐하면, 한 비트만 빼고는 모든 비트가 각 연속되는 비트 쌍에 중복되기 때문이다. 오히려, 이 장치는 우리의 예제 1000100110110111에서 레지스터의 최좌측에서부터 일련의 난수 비트들을 만드는 것으로써 간주될 수 있다. 23장에서 설명한 것처럼, 이 장치는 작은 암호값(key)들로부터 긴 비트 순서들(long bit sequences)을 생성할 수 있기 때문에, 암호학에서는 유용하다.

또 하나 재미있는 사실은 같은, 재귀(recursion) 공식에 따라, 한 번에 한 비트보다는 오히려 한 번에 한 워드씩 계산될 수 있다는 것이다. 우리의 예제에서, 만일 두 개의 연속적인 워드를 비트별로 “exclusive or” 했다면, 우리는 리스트에서 세 자리 뒤에 나타나는 워드를 얻을 수 있다. 이것은 우리로 하여금 범용 컴퓨터 상에서 쉽게 난수 생성기를 구현하게 한다. 비트 b 와 c 를 탭(tab)으로 하여 피드백 레지스터를 사용하는 것은 재귀적 방식 $a[k] = (a[k-b] + a[k-c]) \% m$ 의 사용에 상응한다. 쉬프트-레지스터 모델과 일관성을 유지하기 위해, 이 재귀식의 “+”는 비트별 “exclusive or”이어야 한다. 그러나, 일반 정수 덧셈이 이용되더라도 더 좋은 난수들이 만들어질 수 있음이 알려져 있다. 이를 부가적 조화 방법이라고 한다.

부가적 조화 생성기에서는, 수들의 가장 오른쪽 비트들은 해당 선형-피드백-쉬프트-레지스터에 있는 비트들 같이 동작한다. 그래서, 이 방법이 반복을 시작하기 전의 스텝의 수는 적어도 사이클 길이만큼 크다. 이 사실을 제외하면, 이러한 생성기들에 의해 만들어진 수들에 대해 알려진 특정 결과는 거의 없다: 이들을 지원하는 증거는 주로 경험적인 증거 뿐이다.(이들은 통계적 시험들 통과한다)

부가적 조화 생성기를 구현하기 위해, 항상 c 개의 가장 최근에 만들어진 숫자를 갖는 크기 c 의 도표는 유지할 필요가 있다. 계산은 아래 구현된 것처럼, 도표에서의 두 다른 수의 합으로 도표에 있는 수들 중의 하나를 대체하는 것이다.

```
class Random
{
private:
    int a[55], j;
public:
    Random(int seed);
    int next(int r)
    {
        j = (j + 1) % 55;
        a[j] = (a[(j + 23) % 55] + a[(j + 54) % 55]) % m;
        return ((a[j] / m1) * r) / m1;
    }
};
```

전역변수 a 는 전체도표와 포인터(j)로 대체된다. Knuth는 부가적 조화 방법에서 순환(recurrence)을 위해 $b = 31$ 과 $c = 55$ 를 택할 것을 권한다. 그래서, 가장 최근에 발생된 55개의 수를 계속 추적해 볼 수가 있다. 이를 위한 적절한 자료 구조는 큐(3 장을 보라)이다. 그러나 큐는 고정된 크기를 가지므로 순환 포인터를 이용하여 인덱스되는 고정 크기의 배열만 사용한다. 많은 양의 “전역 상태(global state)”는 몇몇 분야에서는 이 생성기의 단점이 되지만, 아주 긴 사이클을 가지는 장점이 있기도 하다.(모듈로 계수 m 은 크지는 않을지라도, 적어도 $2^{55} - 1$ 은 되어야 한다)

도표는 너무 크지도 작지도 않은 수들로 초기화되어야 한다. 이 수들을 얻을 수 있는 한 가지 쉬운 방법은 간단한 선형-조화-생성기(linear congruential generator)를 사용하는 것이다:

```

Random::Random(int seed)
{ const int b = 31415821;
  a[0] = seed;
  for (j = 1; j <= 54; j++)
    a[j] = (mult(a[j-1], b) + 1) % m;
}

```

이 프로그램은 0과 $r-1$ 사이의 정수 난수를 리턴한다. 앞에서 언급한 것처럼, 이 프로그램은 0과 1 사이의 실수 난수 ($a[j]/m$)을 리턴할 수 있도록 변경될 수 있다.

무작위성 시험(Testing Randomness)

종종 순서가 무작위가 아닌 경우를 찾을 수 있다. 그러나 순서가 무작위임을 확실히 하는 것은 실로 어려운 일이다. 앞서 언급한 것처럼, 어떤 컴퓨터에 의해 만들어진 순서는 사실 무작위는 아니다. 그러나 많은 난수 성질들을 갖는 숫자의 연속은 얻을 수 있다. 불행히도, 난수들의 어떤 성질들이 특정 응용분야에 중요한가를 정확히 설명하는 것은 불가능하다. 더군다나 난수 생성기에 대해, 퇴행적인 경우들이 생기지 않도록 하기 위해, 어떤 시험을 하는 것은 항상 좋다. 난수 생성기들은 아주 좋을 수가 있다. 그러나, 나쁘면, 아주 나쁘다.

한 순서가 실제 난수 순서가 가지는 다양한 성질들을 가지고 있는지 아닌지를 결정해주는 많은 시험 방법들이 개발되어 왔다. 대부분의 이런 시험 방법들은 수학에 그 기초를 두고 있는데 이들 방식의 자세한 내용은 이 책의 범위를 넘어간다. 그러나, 한 통계적 시험인 χ^2 (chi-square) 시험은 성질상 아주 근본적이고, 구현이 아주 쉬우며, 여러 응용 분야에서 유용하므로, 이에 대해서 좀 더 자세히 알아보기로 한다.

χ^2 시험의 개념은 산출된 값들이 고르게 퍼져 있는지 아닌지를 검사하는 것이다. 만약 r 보다 작은 N 개의 양수를 생성하였다면, 각 값에 대해 N/r 값을 기대할 것이다. 그러나 그리고 이것이 문제의 본질이다—모든 값의 발생 빈도가 정확히 같지는 않아도 된다: 그것은 무작위일 수는 없다! 난수 순서 뿐만 아니라 수들의 순서가 분산되었는지 어떤지를 계산하는 것은 어렵지 않다: 각 값의 기대 빈도에 따라 비율 조절(scale)된 제곱값을 더한 다음, 그 순서의 크기를 빼라. 이 결과치인 “ χ^2 통계치”는 수학적으로

$$\chi^2 = \frac{\sum_{0 \leq i < r} (f_i - N/r)^2}{N/r}$$

로 표현될 수 있으며, 다음 프로그램에서 처럼 아주 쉽게 계산된다:

```
float chisquare(int N, int r)
{
    int i, t, f[rmax]; Random x(1234567);
    for(i = 0; i < r; i++) f[i] = 0;
    for(i = 0; i < N; i++) f[x.next(r)]++;
    for(i = 0, t = 0; i < r; i++) t += f[i] * f[i];
    return (float) ((r*t / N) - N);
}
```

만약 χ^2 통계가 r 에 가깝다면, 수들은 무작위이다; 만약 멀다면, 무작위가 아니다. “가깝다”와 “멀다”라는 용어는 좀더 자세히 정의될 수 있다: 난수 순서의 성질들과 통계를 연관시키는 법을 정확히 보여주는 도표들이 있다. 우리가 하고 있는 단순한 시험의 경우, 그 통계는 $2\sqrt{r/r}$ 내에 있어야 한다. 이것은 N 이 $10r$ 보다 크면 타당하다: 확실히 하기 위해서는, 시험은 여러 번 행해져야 한다. 왜냐하면 10번 중 한 번 정도는 잘못될 수 있기 때문이다.

이 시험은 아주 구현이 단순하므로 모든 난수 생성기에 첨가되어 어떤 예기치 못한 것들이 심각한 문제들을 일으키지 못하도록 해야한다. 이제까지 언급한 “좋은” 생성기는 이 시험을 통과한 것들이며, “나쁜” 생성기는 통과하지 못한 것들이다. 위의 생성기를 사용하여 100

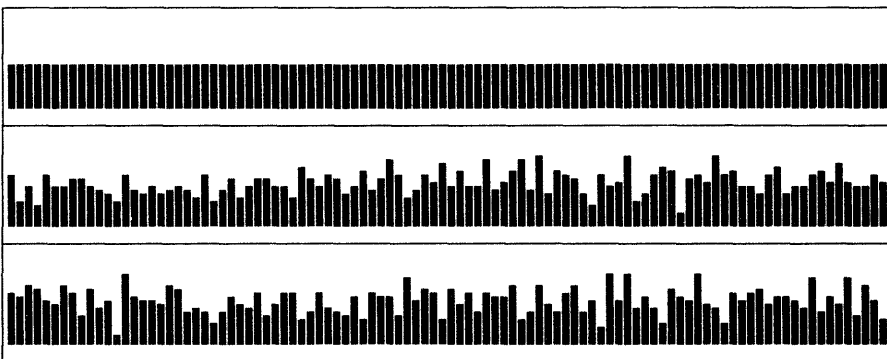


그림 35.2 세 생성기에 대한 빈도들: 우측 비트들, 좌측 비트들, 나쁜 승수

보다 작은 수를 수천 개 만들기 위해, 선형 조화 방법으로는 100.8의 x^2 통계를, 부가적 조화 방법으로는 105.4의 x^2 통계를 얻는다. 둘 다 20/100내에서 잘 동작한다. 그러나 선형 조화 생성기에서 오른쪽 비트들을 사용한 “나쁜” 생성기의 경우, 통계는 0이다.(왜?) 그리고 나쁜 승수(101011)를 사용한 선형 조화 방식의 경우, 통계는 77.8이었다. 이 값은 범위를 상당히 벗어난 값이다. 그림 35.2는 앞에 언급한 선형 조화 방법의 세 가지 버전들에 대해 100보다 작은 값들 각각의 발생 빈도를 보여준다. 왼쪽 비트들을(최상단 차트) 사용하는 것은 명백히 나쁜 것이 보이나, 중간과 끝 부분의 빈도 차트들간의 차이점을 알기는 쉽지 않다. x^2 통계는 나쁜 승수를 구별해내는 방법을 제공한다.

구현 노트(Implementation Notes)

많은 유용한 도구들이 다양한 응용분야들에 유용한 난수 생성기를 만들기 위해 추가되곤 한다. 생성기를 초기화되고, 반복적으로 호출되며, 매번 다른 난수를 리턴하는 함수를 만들면 아주 좋을 것이다. 또 다른 가능성은 한 번 난수 생성기를 호출하면 특정 계산에 필요한 모든 난수들을 배열에 채워주도록 만드는 것이다. 어느 경우에는, 연속적인 호출시 생성기가 동일 순서를 산출함(초기 디버깅(debugging)이나 동일 입력을 가지고 프로그램을 비교하기 위해) 그리고 임의의 순서를(차후의 디버깅을 위해) 산출함이 바람직하다. 이러한 도구들은 모두 난수 생성기에 의한 호출들간에 보존된 “상태”들을 조작한다. 이것은 어떤 프로그래밍 환경에서는 매우 불편할 수도 있으나, 위의 C++ 프로그램은 그와 같은 도구들을 추가된 클래스(class) 함수들로서 쉽게 구현할 수 있다. 부가적 생성기는 상대적으로 상당히 큰 상태(최근 생성된 워드들의 배열)를 갖는 단점이 있으나, 각 사용자가 그 생성기를 초기화할 필요가 없을 정도로 긴 사이클을 갖는 장점이 있다.

난수 생성기의 이상 동작으로부터 보호하기 위한 한 보수적인 방법은 두 개의 생성기를 결합하는 것이다.(생성기에 대한 도표를 초기화하기 위해 선형 조화 생성기를 사용하는 것이 그 한 예이다) 결합 생성기 구현을 위한 한 쉬운 방법은 첫 번째 생성기로 도표를 채우고, 두 번째 생성기로(출력을 위해 그리고 첫 번째 생성기로부터의 새로운 수를 저장하기 위해) 무작위적으로 도표 위치에서 수들을 끄집어내는 것이다.

난수 생성기를 사용하는 프로그램을 디버깅할 때는, 수를 리턴할 때 항상 0이나 1을 리턴하는 것처럼, 처음에는 쉽고 단순한 생성기를 사용하는 것이 좋다.

대개, 난수 생성기는 고장나기가 쉬우므로 잘 다루어야 한다. 특정 난수 생성기가 많은 통계적 시험없이 '좋다'고 확신하기는 곤란하다. 좋은 생성기의 선택 및 사용을 위한 식별법은 수학적 분석과 많은 사람들의 경험에 근거하여야 한다: 확신을 기하기 위해, 난수로 보이는 수들이 진짜 난수인지를 시험해 보는 것이다; 만약 무엇인가 잘못됐다면, 난수 생성기에 문제가 있는 것이다.

연습문제

1. 무작위로(문자들의 집합인) 4-문자 워드(word)들은 생성하는 프로그램을 작성하라. 프로그램이 얼마나 많은 단어들을 반복이 발생하기 전까지 생성하는지를 측정하라.
2. 두 개의 주사위를 던지고 그 합에 의해 난수들을 생성하는 것을 어떻게 모의 실험 할 것인가, 여기서 주사위 상의 숫자들은 정상 주사위 수가 아니라 1, 2, 3, 5, 8 그리고 13이라고 가정한다.
3. 그림 35.1과 같은 선형-피드백-쉬프트-레지스터에 의해 산출된 패턴들의 순서를 보여라. 단, 처음과 마지막 비트들을 탭 위치로 갖는다. 초기 패턴은 1111이다.
4. 왜 “or”나 “and” 함수는(“exclusive or”가 아니라) 선형-피드백-쉬프트-레지스터들에서 동작하지 않는가?
5. 무작위 이차원 이미지(image)를 만드는 프로그램을 작성하라.(예: 무작위 비트들을 생성하라; 1이 발생하면 “*”를, 0이면 “ ”를 출력하라. 다른 예: 이차원 데카르트(cartesian) 시스템의 좌표값으로 난수들을 이용한다. 지정된 지점에 “*”를 출력한다.
6. 1000보다 작은 1000개의 양수들을 생성하기 위해, 부가적 조화 난수 생성기를 사용하라. 이 수들이 난수인지 아닌지를 결정하기 위한 시험 방식을 설계하고 적용하라.
7. 1000보다 작은 1000개의 양수들을 생성하기 위해, 당신 마음대로 선택한 파라미터(parameter)들을 갖고 선형 조화 난수기를 사용하라. 이 수들이 난수인지 아닌지를 결정하기 위한 시험방법을 설계하고 적용하라.
8. 부가적 조화 생성기는 왜 $b = 3$ 과 $c = 6$ 인 예제에 사용하면 좋지 않은가?
9. 항상 같은 값을 리턴(return)하는 퇴행적(degenerate) 생성기에 대한 X^2 통계 값은 얼마인가?
10. 컴퓨터 워드 크기보다 큰 m 으로 어떻게 난수를 생성할 수 있는지를 설명하라.

36 장

산술연산

비록 상황이 변하더라도, 많은 컴퓨터 시스템들이 존재하는 이유는 산술 계산을 정확하고 빠르게 하는 능력을 가지고 있기 때문이다. 컴퓨터는 정수 그리고 실수를 표현하는데 사용되는 부동소수점 표기법을 이용하여 산술연산을 수행하는 기능을 내장하고 있다. 예를 들어 C++는 정수 형태나 부동소수 형태의 수를 허용한다. 물론, 정수나 부동소수 형태로서 정의되는 모든 일반 연산도 다 허용한다. 산술 연산을 위해 사용되는 실제 방법은 컴퓨터 구조의 한 부분이므로 여기서는 고려하지 않기로 한다.(비록 예로서, 32비트의 두 정수를 곱하는 빠르고 새로운 알고리즘은 사실 매우 중요한 일이지만) 그 대신, 보다 복잡한 수학적 객체(object)들 상에서 연산들이 수행되어야 할 때 사용되는 몇몇 알고리즘들에 대하여 논하기로 한다.

이장에서는 다항식, 긴 정수(long integer), 그리고 행렬의 덧셈이나 곱셈을 하는데 사용되는 알고리즘들을 C++로 구현하는 것에 대해 살펴볼 것이다. 이런 것들에 대한 기본적인 알고리즘들은 친숙하고 또한 간단하기는 하지만, 어떻게 특정 상황에서 이러한 기본적인 데이터 구조체들을 적용하는지 생각해 볼만한 가치가 있다. 여기서는 알고리즘들의 응용분야에 대해 중점을 두지 않으려고 한다. 대신, 기초적인 산술 문제들의 계산적 복잡도를 고찰하고자 한다. 예를 들어, 다항식, 긴 정수, 그리고 행렬을 위한 C++클래스들의 환경에서, 알고리즘들을 구현하는 것이 자연스럽기는 하지만, 알고리즘들의 본질적 특징을 집중 고찰해 보기 위해 구현은 하지 않기로 한다. 한편, 여기서 언급되는 방법들이 도움이 되는 실제적인 응용분야들은 우리가 지금까지 공부해 왔던 다른 분야들에 비해 볼 때 다소 적다. 그 반면에, 근본적인 수학적 객체들에 대한 기본적인 산술 연산들을 수행하기 위한 알고리즘들을 공부하는 것에 그 이상의 이유는 필요하지 않다.

산술연산은 어떻게 알고리즘들을 적절히 응용하여 기본적인 방법보다 아주 더 효과 있는 세련된 방법을 만들어 내는가에 대한 훌륭한 예를 제공한다. 이러한 연구는 이러한 본질적인 계산적 문제의 특징 뿐만 아니라 그 역사적인 배경으로 인해 더 흥미롭다. 덧셈, 곱셈, 나눗셈 같은 기본적인 산술연산을 위한 알고리즘들은 아주 오랜 역사를 가지고 있다. 아라비아인 al-Krowarizmi의 연구까지 그 기원이 거슬러 올라갈 뿐만 아니라 고대 그리스와 바빌론까지 거슬러 올라간다.

우리는 다항식의 덧셈과 행렬의 곱셈이 분할-정복(divide-and-conquer) 패러다임(paradigm)이 강력함을 보여주는 고전적인 예임을 알게 될 것이다. 불행히도, (41장에서 언급되는 중요한 방법을 제외한) 결과 알고리즘들은 실제로 응용되기 어렵다. 기초적인 데이터 구조체를 현명하게 사용하는 기본적인 방법들은 지나치게 큰 문제를 제외한 문제들의 경우에는 최고로 좋다.

다항식 연산(Polynomial Arithmetic)

두 개의 다항식을 더하는 프로그램을 작성한다고 하자. 아마도 다음과 같은 연산을 수행하기를 원할 것이다.

$$(1 + 2x - 3x^3) + (2 - x) = 3 + x - 3x^3$$

일반적으로 프로그램이 $r(x) = p(x) + q(x)$ 를 계산할 수 있기를 원한다하자. 여기서, p 와 q 는 N 개의 계수를 갖는 다항식이다. 배열을 사용하면 이러한 계산은 아주 쉽다. 우리는 $p[j] \equiv$ (즉, $p[j]$ 를 p_j 라 할 때) 다항식 $p(x) = p_0 + p_1x + \dots + p_{N-1}x^{N-1}$ 을 배열 $p[N]$ 으로 나타낸다. 그러면, 이 다항식의 덧셈은 다음과 같은 단 한 줄의 프로그램으로 표현된다.

```
for (i = 0; i < N; i++) r[i] = p[i]+q[i];
```

C++에서는 그렇듯이(3장 참고), 먼저 얼마나 큰 N 값을 얻을 수 있는지를 결정해야 한다. 왜냐하면, p , q , r 배열의 크기가 예상 최대 크기로 설정되어야 하기 때문이다.

이 프로그램은, 덧셈이 다항식의 배열형태로 표현될 경우, 아주 간단한 문제라는 것을 보여준다. 다른 경우의 연산도 또한 쉽게 할 수 있다. 예를 들어, 다음 프로그램부분은 다항식 곱셈을 간단히 구현한 것이다.

```

for (i = 0; i < 2*N-1; i++) r[i] = 0;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        r[i+j] += p[i]*q[j];

```

r은 곱셈을 위한 많은 계수들의 배수만큼 선언되어야만 한다. p의 N개의 계수들은 각각 q의 N개의 계수 각각과 곱해진다. 그래서, 이 알고리즘의 실행 시간은 정확히 계수 수의 제곱이다.

3장에서 본 바와 같이, 계수를 가지는 배열형태로 다항식을 표현하는 것의 한 가지 장점은 어떤 계수든지 바로 참조(reference)할 수 있다는 것이다. 한 가지 단점은 필요이상의 저장공간이 필요하다는 것이다. 예를 들어, 위 프로그램은, 비록 입력 다항식이 단지 네 개의 계수를 그리고 출력 다항식이 단지 세 개의 계수를 가지고 있지만, 곱셈 연산을 하는데 합리적이지 않을 수 있다.

$$(1 + x^{10000})(1 + 2x^{10000}) = 1 + 3x^{10000} + 2x^{20000}$$

이와 다른 방법으로, 다항식을 연결 리스트(linked list)를 사용하여 나타내어, 덧셈을 할 수 있다:

```

struct node *add(struct node *p, struct node *q)
{
    struct node *t;
    t = z; z->c = 0;
    while ((p != z) && (q != z))
    {
        t->next = new node;
        t = t->next; t->c = p->c + q->c;
        p = p->next; q = q->next;
    }
    t->next = z; t = z->next; z->next = z;
    return t;
}

```

입력 다항식은 계수당 하나의 원소를 가지는 연결-리스트로 표현된다. 출력 다항식은 add 프로시저(procedure)에 의해 만들어진다. 이 연결-리스트의 조작 방법은 3, 8, 14, 29장과 이 책 여러 곳에서 나타나는 예제 프로그램과 유사하다.

보는 바와 같이, 프로그램은 C++의 동적 배열의 결핍을 교묘히 처리하는 것을 제외하고는, (계수당 링크를 위한 공간을 부여하여) 배열 표현방식에 비해 실질적인 개선이 없다. 그러나, 위의 예가 시사하는 바와 같이, 많은 계수가 0일 거라는 가능성의 장점을 이용할 수 있다. 우리는 리스트 노드내에 나타나는 항의 차수를 포함시킴으로서 리스트 노드가 다항식의 0이 아닌 항만을 나타내게 할 수 있다. 그래서 각각의 리스트 노드는 cx^j 를 나타내기 위해 c 와 j 값을 가진다. 노드를 생성하는 함수와 리스트에 노드를 추가하는 함수를 다음과 같이 분리하면 편하다.

```
struct node
{ int c; int j; struct node *next; };
struct node *insert(struct node *t, int c, int j)
{
    t->next = new node;
    t = t->next; t->c = c; t->j = j;
    return t;
}
```

insert함수는 새로운 노드를 생성하고, 거기에 특정한 필드(Field)를 만들고, 노드 t 다음의 리스트에 그 노드를 연결한다. 보다 정돈된 방법으로 다항식의 처리를 가능하게 하려면, 리스트 노드는 항의 차수의 오름차순 형태로 될 수 있다.

add함수는 더 재미있다. 왜냐하면, 차수가 매치하는 항에 대해서만 덧셈을 수행해야 하고 계수가 0인 어떤 항도 출력되지 않도록 해야 하기 때문이다.

```
struct node *add(struct node *p, struct node *q)
{
    struct node *t;
    t = z; z->c = 0; z->j = maxN;
    while ((p != z) || (q != z))
    {
        if ((p->j == q->j) && ((p->c + q->c) != 0))
```

```

    {
        t = insert(t, p->c+q->c, p->j);
        p = p->next; q = q->next;
    }
    else if (p->j < q->j)
        { t = insert(t, p->c, p->j); p = p->next; }
    else if (q->j < p->j)
        { t = insert(t, q->c, q->j); q = q->next; }
    }
    t->next = z; t = z->next; z->next = z;
    return t;
}

```

이러한 개선은 많은 0의 계수를 갖는 “스파아스” 다항식을 처리하기 위해 할만한 가치가 있다. 왜냐하면 다항식을 처리하기 위한 저장 공간과 처리 시간이 다항식의 차수에 비례하는 것이 아니고 계수의 수에 비례하기 때문이다. 이와 비슷한 절약이 다항식의 곱셈 같은, 다른 연산들에서도 가능하다. 하지만, 다항식은 많은 연산들이 수행되고 난 후에는 두드러지게 덜 스파아스 해지기 때문에 이에 대한 주의가 필요하다. 0의 계수를 가진 항이 약간 있거나 다항식의 차수가 높지 않을 때에는 배열 표현이 더 효과적이다. 설명을 쉽게 하기 위해, 이 표현법을 아래 주어진 다항식에 대한 많은 알고리즘들을 설명하는데 사용한다고 가정한다.

다항식은 단순히 하나 뿐만 아니라 여러 개의 변수를 포함할 수 있다. 예를 들어, 다음과 같은 다항식을 처리해야할 필요가 있을 지 모른다.

$$1 + wx^2 + y^6z + w^{25}x^{50}y^{99}z^{38} + x^{1000}z^{1000}$$

이와 같은 경우들은 연결 리스트를 이용해 표현하면 확실히 좋다. 이의 대안(다차원 배열)은 너무 많은 공간을 소모할 수도 있다. 이같은 다항식들의 처리를 위해(예제에서와 같은) add 프로그램을 확장하는 것은 어렵지 않다.

다항식 평가법 및 보간법 (Polynomial Evaluation and Interpolation)

한 점에서 다항식의 값을 어떻게 계산하는지 살펴보자. 예를 들어, 다항식을 한 x 에서 평가하기

$$p(x) = x^4 + 3x^3 - 6x^2 + 2x + 1$$

위해, x^4 을 계산하고 그 다음에 $3x^3$ 을 계산하여 더하는 방식으로 연산할 수가 있다. 이 방법은 x 의 승수들의 재 계산을 필요로 한다; 다른 방법으로는, x 의 승수들을 그 계산 값이 나올 때마다 저장해 두는 방법이 있지만, 이 방법은 더 많은 저장 공간을 필요로 한다.

이러한 재 계산과 잉여 공간의 사용을 피하기 위해, Horner법칙이라 알려진 방법을 사용한다. 이 법칙은 덧셈과 곱셈을 적절히 번갈아 가며 수행함으로써, 차수- N 의 다항식의 경우, $N - 1$ 번의 곱셈과 N 번의 덧셈만으로 평가할 수 있다. 아래의 괄호식

$$p(x) = x(x(x(x + 3) - 6) + 2) + 1$$

은 명백히 다음과 같은 계산 순서를 만들어 낸다.

```
y = p[N];
for (i = N-1; i >= 0; i--) y = x*y + p[i];
```

16장에서 긴 키(long key)값들에 대한 해시(Hash)함수를 계산시 이미 이 방법을 사용한 바 있다.

많은 서로 다른 점에서 주어진 다항식을 평가하는 문제는 더 복잡하다. 알고리즘들은 얼마나 많은 평가가 행해져야 되는지와 그 평가들이 동시에 이루어 질 수 있는지 아닌지에 따라 그 적합 여부가 결정된다. 만약 아주 많은 평가가 행해져야 하면, 다음 번 평가비용을 약간 절약하기 위해 값들을 미리 계산(precomputing) 해 두는 것이 좋을 수 있다. Horner방법은, N 개의 서로 다른 점에서 차수- N 을 가지는 다항식의 평가를 위해, N^2 번의 곱셈 계산을 해야함에 주목하라. $N(\log N)^2$ 스텝내에 문제를 풀 수 있도록 더 세련된 방법들이 설계되어 왔다. 41장에서는 관심사가 되는 N 개의 점들의 집합에 대해 단지 $N \log N$ 번의 곱셈을 사용하는 한 방법을 살펴볼 것이다.

만약 주어진 다항식이 단지 하나의 항을 가지고 있다면, 이 다항식 평가 문제는 x^N 을 계산하는 지수화(exponentiation)문제로 축약된다. 이 경우, Horner법칙은 $N - 1$ 번의 곱셈을 요하는 사소한 알고리즘이 된다. 좀더 잘 만들 수 있는지를 알아보기 위해, x^{32} 를 계산하는 다음 순서를 살펴보자.

$$x, x^2, x^4, x^8, x^{16}, x^{32}.$$

각 항은 바로 앞 항의 제곱이므로, 31번의 연산이 필요한 것이 아니라 단지 5번의 곱셈만 이 필요하다.

“연속 제곱(successive-squaring)” 방법은 계산의 결과가 저장된다면 일반적인 N 으로 쉽게 확장이 될 수 있다. 예를 들어, x^{55} 는 위의 값에서 곱셈을 네 번 더 함으로써 구할 수 있다.

$$x^{55} = x^{32} x^{16} x^4 x^2 x^1.$$

일반적으로 N 의 이진표현은 계산된 값들 중에서 어떤 값을 사용할지를 결정할 때 사용될 수 있다.(예에서, $55 = (110111)_2$ 이기 때문에, x^8 을 제외한 모두가 사용된다) 연속 제곱이 계산되고 N 의 비트들은 동일 루프 내에서 조사된다. Horner방법과 같이, 단지 하나의 “누산기(accumulator)”를 사용하여 이를 구현하는 두 가지 방법이 있다. 한 알고리즘은 누산기 안의 1로 부터 시작하여 N 의 이진 표현을 왼쪽에서 오른쪽으로 스캐닝(scanning)하는 것을 수반한다. 각 단계에서 누산기를 제곱하고, 이진 표현된 N 에 1이 있으면 x 를 곱한다. 다음 일련의 값들은 이 방법으로 $N = 55$ 에 대해 계산한 것이다.

$$1, 1, x, x^2, x^3, x^6, x^{12}, x^{13}, x^{26}, x^{27}, x^{54}, x^{55}$$

또 하나 아주 잘 알려진, 이와 유사하게 동작하는, 알고리즘이 있는데, 이 알고리즘은 N 을 오른쪽에서 왼쪽으로 스캔한다. 이 문제는 하나의 기본 프로그래밍 연습문제이다. 이 문제는 아주 큰 수를 계산하는데 별 실제적 가치가 없어 보이지만, 이 방법이 23장의 공개-키(public-key) 암호 시스템을 구현할 때 한 역할을 담당함을, 아래에서 큰 정수의 경우에 대해 논할 때, 살펴볼 것이다.

N 개의 점에서 차수 N 인 다항식을 동시에 계산하는 문제의 “역(inverse)”은 다항식 보간법(polynomial interpolation) 문제이다: 이 문제에서는 주어진 점 x_1, x_2, \dots, x_N 과 이들에 상응하는 값이 y_1, y_2, \dots, y_N 인 집합에서, 아래에 있는 차수 $N - 1$ 의 유일한 다항식을 찾는다.

$$p(x_1) = y_1, p(x_2) = y_2, \dots, p(x_N) = y_N.$$

다항식 보간법 문제는 주어진 점들과 값들의 집합에서 다항식을 찾기 위한 것이다. 다항식 계산법 문제는 주어진 다항식과 점들에서 값들을 찾기 위한 것이다.(주어진 다항식과 값들에서 점들을 찾는 문제는 루트 찾기(root-finding)문제이다)

보간법 문제의 전통적인 해결책은 차수 $N - 1$ 을 갖는 다항식이 N 개의 점들에 의해서 완전히 찾아짐을 증명할 때 종종 사용되는 Lagrange 보간법에 의해 주어진다.

$$p(x) = \sum_{1 \leq j \leq N} y_j \prod_{\substack{1 \leq i \leq N \\ i \neq j}} \frac{x - x_i}{x_j - x_i}.$$

이 공식은 보기에는 어려워 보이지만, 실제로는 아주 간단하다. 예를 들어, $p(1) = 3$, $p(2) = 7$, 그리고 $p(3) = 13$ 인 차수 2인 다항식이 다음과 같이 주어졌다고 하자.

$$p(x) = 3 \frac{x-2}{1-2} \frac{x-3}{1-3} + 7 \frac{x-1}{2-1} \frac{x-3}{2-3} + 13 \frac{x-1}{3-1} \frac{x-2}{3-2}$$

이 식은 다음과 같이 단순화된다.

$$x^2 + x + 1.$$

x_1, x_2, \dots, x_N 의 원소를 갖는 x 로부터 공식이 만들어지고 따라서 x 가 1이 될 때 $1 \leq r \leq N$ 에 대해 $p(x_k) = y_k$ 이다. 왜냐하면 $j = k$ 가 아닌 한 그 곱은 0이 되기 때문이다. 예에서, $x = 1$ 일 때 마지막 두 항은 0이고, $x = 2$ 일 때 처음과 마지막 항은 0이며, $x = 3$ 일 때 처음 두 항은 0이다.

Lagrange 공식에 의해 설명되는 형태인 다항식을 표준 계수 표현법(standard coefficient representation)으로 바꾸는 것은 전혀 간단하지 않다. 그 합에 N 개의 항이 있고, 각각이 N 개의 인자를 갖는 곱으로 이루어져 있기 때문에, 최소한 N^2 개의 연산이 필요한 것처럼 보인다. 실제로 이차식을 얻으려면 약간의 지혜가 필요하다. 왜냐하면 그 인자들은 단순히 숫자들도 있지만 차수가 N 인 다항식도 있기 때문이다. 반면에, 각 항은 전과 아주 유사하다. 독자 중에는 이 이차식 알고리즘을 얻기 위해 어떻게 이를 이용하는지에 대해 흥미를 느끼는 이들도 있을 것이다. 이 문제는 독자가 수학적 공식에 표현된 계산을 수행하는 한 효율적인 프로그램의 작성이 간단하지 않음을 보여준다.

다항식의 계산과 마찬가지로, $N(\log N)^2$ 스텝내에 문제를 해결할 수 있는 좀 더 세련된 방법이 있다. 그리고, 41장에서 특정한 N 개의 점의 집합에서 단지 $N \log N$ 의 곱셈 계산만을 사용하는 한 방법을 볼 수 있을 것이다.

다항식 곱(Polynomial Multiplication)

우리가 살펴본 첫 번째 정교한 알고리즘은 주어진 두 개의 다항식 $p(x)$ 과 $q(x)$ 에서 두 다항식의 곱인 $p(x)q(x)$ 을 계산하는 다항식 곱셈문제이다. 이장의 처음에서 언급했듯이, 차수 $N - 1$ 의 다항식은 N 개의 항(상수를 포함해서)을 가질 수 있고 그 곱은 차수 $2N - 2$ 를 가지며 $2N - 1$ 항을 가진다. 예로서, 다음 다항식의 곱을 보자.

$$(1 + x + 3x^2 - 4x^3)(1 + 2x - 5x^2 - 3x^3) = (1 + 3x - 6x^3 - 26x^4 + 11x^5 + 12x^6)$$

이장의 처음에서 주어진, 이 문제에 대한 고지식한(naive) 알고리즘은 차수 $N - 1$ 을 가지는 다항식에 대해 N^2 번의 곱셈을 필요로 한다. $p(x)$ 의 N 항들 각각은 $q(x)$ 의 N 항들의 각각과 곱해진다.

이 알고리즘을 개선하려면, “분할-정복”방법을 사용해야한다. 다항식을 둘로 나누는 한 가지 방법은 계수를 반으로 나누는 것이다. N 개의 계수를 갖는 차수 $N - 1$ 의 다항식을 $N/2$ 개의 계수를 가지는 두 개의 다항식으로 나눌 수 있다.(단 N 이 짝수라고 가정한다) N 을 둘로 나눈 것 중 아래쪽의 계수를 하나의 다항식에, 나머지 위쪽의 계수들을 다른 하나의 다항식에 사용한다. $p(x) = p_0 + p_1x + \dots + p_{N-1}x^{N-1}$ 의 다항식에 대해 다음을 정의하자.

$$\begin{aligned} p_l(x) &= p_0 + p_1x + \dots + p_{N/2-1}x^{N/2-1} \\ p_h(x) &= p_{N/2} + p_{N/2+1}x + \dots + p_{N-1}x^{N/2-1} \end{aligned}$$

그리고, $q(x)$ 도 같은 방법으로 나눈다.

$$\begin{aligned} p(x) &= p_l(x) + x^{N/2} p_h(x), \\ q(x) &= q_l(x) + x^{N/2} q_h(x). \end{aligned}$$

이제, 이 작은 다항식들로부터, 다항식의 곱을 다음과 같이 구할 수 있다.

$$p(x)q(x) = p_l(x)q_l(x) + (p_l(x)q_h(x) + q_l(x)p_h(x))x^{N/2} + p_h(x)q_h(x)x^N$$

(오버플로우를 피하기 위해, 35장의 방식과 같은 분할 방식을 사용했다)

이 방식에서의 요점은 이 곱들을 계산하기 위해(위 공식에서 있을 법한 4번의 곱셈이 아니다) 단지 세 번의 곱셈만이 필요하다는 것이다. 왜냐하면 만약 우리가 $r_l(x) = p_l(x)q_l(x)$,

$r_h(x) = p_h(x)q_h(x)$ 와 $r_m(x) = (p_l(x) + p_h(x))(q_l(x) + q_h(x))$ 을 계산한다고 하면, $p(x)q(x)$ 의 결과는 다음과 같이 얻어지기 때문이다.

$$p(x)q(x) = r_l(x) + (r_m(x) - r_l(x) - r_h(x))x^{N/2} + r_h(x)x^N.$$

이같이 줄어든 곱셈 연산은, 이 작은 예에서는 아마도 명확히 알 수는 없을 것이다. 이 방법은 다항식의 덧셈이 선형 알고리즘을 필요로 하다는데에 기초한다. 그리고, 주먹구구식의 (brute-force) 다항식 곱셈은 이차식이다. 그래서 한 번의 곱셈을 줄이기 위해, 약간의 덧셈을 더 수행할 가치가 있다. 여기서 이 방법에 대해 좀 더 자세히 살펴보자.

$p(x) = 1 + x + 3x^2 - 4x^3$ 이고 $q(x) = 1 + 2x - 5x^2 - 3x^3$ 인 위의 예에서, 다음을 얻는다.

$$r_l(x) = (1 + x)(1 + 2x) = 1 + 3x + 2x^2,$$

$$r_h(x) = (3 - 4x)(-5 - 3x) = -15 + 11x + 12x^2$$

$$r_m(x) = (4 - 3x)(-4 - x) = -16 + 8x + 3x^2.$$

이같이, $r_m(x) - r_l(x) - r_h(x) = -2 - 6x - 11x^2$ 이고 그 곱은 위의 공식에 따라 세 항을 합하여 구할 수 있다.

$$\begin{aligned} p(x)q(x) &= (1 + 3x + 2x^2) + (-2 - 6x - 11x^2)x^2 + (-15 + 11x + 12x^2)x^4 \\ &= 1 + 3x - 6x^3 - 26x^4 + 11x^5 + 12x^6 \end{aligned}$$

이 분할-정복 방식은 $N/2$ 크기로 나누어진 세 개의 작은 문제를 풀어서, 크기 N 인 다항식의 곱셈 문제를 해결한다. 물론 이 문제를 여러 개의 조각으로 나누고 그 결과치들을 더하기 위해 한 다항식 덧셈 방법을 사용했다.(만일 $N = 1$ 이면, 그 곱은 바로 상수 계수의 스칼라 (scalar) 곱이다) 따라서, 이 프로시저는 쉽게 재귀적 호출 프로그램으로 표현될 수 있다.

```
float *mult(float p[], float q[], int N)
{
    float pl[N/2], ql[N/2], ph[N/2], qh[N/2],
        t1[N/2], t2[N/2];
    float r[2*N-2], rl[N], rm[N], rh[N];
    int i, N2;
    if (N == 1)
```

```

    { r[0] = p[0]*q[0]; return (float *) r; }
for (i = 0; i < N/2; i++)
    { pl[i] = p[i]; ql[i] = q[i]; }
for (i = N/2; i < N; i++)
    { ph[i-N/2] = p[i]; qh[i-N/2] = q[i]; }
for (i = 0; i < N/2; i++) t1[i] = pl[i]+ph[i];
for (i = 0; i < N/2; i++) t2[i] = ql[i]+qh[i];
rm = mult(t1, t2, N/2);
rl = mult(pl, ql, N/2);
rh = mult(ph, qh, N/2);
for (i = 0; i < N-1; i++) r[i] = rl[i];
r[N-1] = 0;
for (i = 0; i < N-1; i++) r[N+i] = rh[i];
for (i = 0; i < N-1; i++)
    r[N/2+i] += rm[i] - (rl[i]+rh[i]);
return (float *) r;
}

```

비록 위 프로그램 코드가 이 방법의 간결한 표현이라 해도, 불행히도 이것은 배열을 동적으로 정의 할 수 없기 때문에, 올바른 C++프로그램이 아니다. 이 프로그램은 분명하게 메모리가 할당되고 반환되는 임시(Scratch)배열이나 연결-리스트로서 다항식을 나타냄으로서 C++에서 처리될 수 있다. 이 문제를 독자에게 연습문제로 남겨둔다. 위 프로그램은 일반적인 N 에 대한 세부적인 처리를 하여 쉽게 답을 얻을 수 있지만, N 이 2의 제곱이라고 가정한다. 주된 문제는 재귀호출이 올바르게 완료 되어야 한다는 것과 다항식이, N 이 홀수일 때, 올바르게 나뉘지도록 해야 한다는 것이다.

왜 이 분할-정복 방법이 진보된 형태일까? 이의 답을 찾으려면, 6장의 기본적인 것보다는 약간 복잡하고 세련된 한 기본적 순환 공식(recurrence formula)을 해결해야 할 필요가 있다.

성질 36.1 차수 N 인 두 개의 다항식은 약 $N^{1.58}$ 번의 곱셈을 함으로써 곱해질 수 있다.

재귀호출 프로그램에서, 크기 N 인 두 다항식을 곱하는데 필요한 정수 곱셈의 횟수는 크기 $N/2$ 인 세 쌍의 다항식들을 곱하기 위한 곱셈의 횟수와 동일함이 명백하다.(단지 데이터 이동을 위한 $r_h(x)x^N$ 의 계산의 경우에는 곱셈이 필요 없음을 주목하라) 만약 M_N 이 크기 N 의 다항식들을 곱하는데 필요한 곱셈의 횟수라면, 우리는 다음과 같은 것을 얻을 수 있다.

$M_1 = 1$ 이고 $N \geq 2$ 인 경우에 대해, $M_N = 3M_{N/2}$.

이같이 하여, $M(2) = 3$, $M(4) = 9$, $M(8) = 27$ 등등이 된다. 6장에서와 같이, 만약 $N = 2^n$ 을 취한다면, 해를 찾기 위해 이를 반복 적용할 수 있다.

$$M(2^n) = 3M(2^{n-1}) = 3^2M(2^{n-2}) = 3^3M(2^{n-3}) = \cdots = 3^nM(1) = 3^n$$

만일 $N = 2^n$ 이면, $3^n = 2^{(\lg 3)n} = 2^{n \lg 3} = N^{\lg 3}$ 이다. 비록 이 해법은 $N = 2^n$ 일 때만 정확하지만, 일반적으로

$$M_N \approx N^{\lg 3} \approx N^{1.58}$$

이다. 따라서, 고지식한 N^2 방법에 비해 상당한 시간절약이 된다. \square

만약 간단한 분할-정복 방법을 통해 네 개의 곱셈 모두를 사용했다면, 기초적 방법과 똑같은 성능을 보였을 수 있다. 왜냐하면, 순환공식은, 그 해로서 $M(2^n) = 4^n = N^2$ 을 갖는, $M(N) = 4M(N/2)$ 이었을 수 있기 때문이다.

이 방법은 분할-정복 방법을 잘 보여준다. 하지만, 이 방법은 실제로는 거의 쓰이지 않는다. 왜냐하면 41장에서 나올 보다 개선된 분할-정복 방법들이 있기 때문이다. 현재의 이 방법은 원래 문제를 단지 두 작은 문제들로 나눔으로서 거의 처리시간의 추가 없이 해결책을 얻는다. 이것은 결국 요구되는 곱셈의 횟수에 대해 우리의 표준 $M_N = 2M_{N/2} + N$ 분할-정복의 반복(recurrence)을 야기시키고, 그 결과로서 M_N 이 약 $N \lg N$ 인 해를 산출한다.

큰 정수 값을 가지는 경우의 산술연산 (Arithmetic Operations with Large Integers)

큰 정수는 계수들에 제한이 있는 다항식과 같이 취급된다. 예를 들어, 다음과 같은 28자리의 정수

$$0120200103110001200004012314$$

는 아래의 다항식에 해당한다.

$$x^{26} + 2x^{25} + 2x^{23} + x^{20} + 3x^{18} + x^{17} + x^{16} + x^{12} + 2x^{11} + 4x^6 + x^4 + 2x^3 + 3x^2 + x + 4.$$

즉, 이 수는 $x = 10$ 일 때의 다항식 값이다. 역으로, 10보다 작은 양의 계수를 갖는 임의의 차수 $N - 1$ 의 다항식은 정확하게 N 자리 정수에 해당한다.

이처럼, 큰 정수를 다루기 위해서 다항식 연산을 사용할 수 있다. 즉, 정수들을 배열로 표시하고, 마치 그 배열들이 다항식을 나타내는 것처럼 우리가 바로 위에서 설명한 다항식 조작 루틴을 사용한다. 예를 들어, 두 개의 100자리 정수를 곱하기 위해서는, 200자리 결과를 계산하기 위해 위에서 언급된 알고리즘을 사용할 수 있다. 이 방법의 결점은 결과 값의 계수가 10보다 작을 확률이 별로 없다는 것이다. 이것은 금방 교정될 수 있다. $i=0$ 에서 시작해서, $p[i]/10$ 을 $p[i+1]$ 에 더한 후, $p[i]$ 을 $p[i]\%10$ 으로 대체하고 i 를 증가시킨다. 이 과정을 0의 계수가 없을 때까지 반복하면 된다.

10보다 큰 기수(radix)가 사용될 수도 있다. 예를 들어 위의 28자리 이상의 수는 다음 다항식에 해당한다.

$$120x^6 + 2001x^5 + 311x^4 + x^3 + 2000x^2 + 401x + 2314$$

$x = 10000$ 에서 이 다항식을 계산하면 결과가 정수 값으로 나온다. 이것은 정수를 메모리를 조금 가지고 표현할 수 있게 하지만, 어떤 중간 연산을 하는 동안 계수에서 오버플로우가 일어날 가능성을 만들어 낸다. 얼마나 큰 기수가 사용될 수 있는가의 수학적 문제는 조심스럽게 해결해야 한다. 그러나, 실제로는 작은 기수를 사용하는 보수적인 방법을 채택하더라도 거의 문제가 없다.

RSA 암호시스템(Cryptosystem)의 경우(23장), 큰 정수를 곱하는 것 뿐만 아니라, 지수계산과 나눗셈도 필요하다. 특히 M , p , N 이 모두 큰 정수일 때, $M^p \bmod N$ 을 계산하는 것도 필요하다. 이것은 수행하기에 간단한 계산이 아니다. 하지만, 대충 그 방법을 기술해 볼 수 있다. 먼저 지수부는 위에서와 같이 연속 곱셈으로 계산할 수 있다. 그러므로 M_1 , M_2 , 그리고 N 이 모두 큰 정수일 때, $M_1M_2 \bmod N$ 을 계산하는 것만으로 충분하다. 모듈로 계수(modulus) 연산을 수행하는 핵심은 마주치는 최대의 큰 정수보다 작은 모든 $10'$ 에 대해 $10' \bmod N$ 을 계산하는 것이다. 그러면, 어떤 특정 모듈로 계수는 이러한 값들의 선형 조합일 것이다. 보다 큰 기수는 필요 계산 양을 줄여 줄 것이다. 수학적 용어로, 이 방법은 아래의 계산에 해당한다:

$$0120200103110001200004012314 \bmod N$$

이 연산은 $x = 10000$ 일 때,

$$120(x^6 \bmod N) + 2001(x^5 \bmod N) + 311(x^4 \bmod N) \\ + (x^3 \bmod N) + 2000(x^2 \bmod N) + 401(x \bmod N) + 2314$$

를 계산하는 것에 해당한다. $10000^i \bmod N$ 의 값들은 미리 계산해서 도표에 저장하거나, 필요하다면 19장에서 살펴본 Rabin-Karp 문자열 검색 알고리즘에서처럼, 증분식으로(incrementally) 계산할 수 있다.

행렬 산술 연산(Matrix Arithmetic)

행렬에 대한 기본적인 연산 동작들의 구현은 위에서 다항식에 대해 언급했던 바와 유사하다. 예를 들어, 행렬의 합은 다항식에서와 같이 항-대-항(term-by-term) 연산이기 때문에 별로 어렵지 않다.

행렬 곱셈 또한 간단하다. 만약 r 이 p 와 q 의 곱이라면, 원소 $r[i][j]$ 는 p 의 i 번째 행과 q 의 j 번째 열의 내적(dot product)이다. 내적은 간단하게 아래와 같은 N 개의 항-대-항 곱셈 값들의 합이다.

$$p_{i1}q_{1j} + p_{i2}q_{2j} + \cdots + p_{i(N-1)}q_{(N-1)j},$$

이는 다음과 같은 프로그램으로 표현될 수 있다.

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0, r[i][j] = 0; k < N; k++)
            r[i][j] += p[i][k]*q[k][j];
```

위의 프로그램을 검사해보기 위해, 다음 예제를 사용해 볼 수 있다.

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} 8 & 3 & 0 \\ 3 & 10 & 2 \\ 0 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 17 & 25 & -18 \\ 11 & 9 & -10 \\ -14 & -13 & 26 \end{pmatrix}$$

결과 행렬의 N^2 개의 원소들 각각은 N 번의 곱셈을 통해서 계산된다. 그래서, 약 N^3 번의 연산이 두 개의 N 행- N 열의 정방 행렬을 곱하는데에 필요하게 된다.

다항식에서와 같이, 스파아스 행렬들은(많은 0의 원소들을 갖는 행렬들) 연결 리스트 표현법을 이용함으로써 보다 효율적으로 처리할 수 있다. 이차원 배열 구조를 변경 없이 유지하기 위해서는, 각 0이 아닌 행렬 원소가 한 개의 값과 두 개의 링크(link)를 갖는 리스트 노드에 의해 표현 되어야 하는데 여기서 한 링크는 동일 행의 0이 아닌 원소를 가리키고, 또 다른 한 링크는 동일 열의 0이 아닌 원소를 가리킨다. 이같이 표현된 스파아스 행렬 덧셈의 구현은 스파아스 다항식의 경우와 유사하다. 하지만, 각 노드가 두 개의 리스트 상에 나타나야 하므로 좀 더 복잡하다.

산술 문제에 대한 분할-정복 기법의 가장 유명한 응용 분야는 행렬 곱셈을 위한 Strassen 방법이다. 여기서 자세히 살펴보지는 않겠지만, 다항식 곱셈의 개념과 상당히 유사하므로 방법만 간단히 기술하겠다.

두 개의 N 행- N 열의 행렬 곱셈을 위한 가장 단순한 N^3 의 스칼라 곱셈들을 필요로 한다. 왜냐하면 곱행렬 내의 N^2 개의 요소들 각각이 N 개의 곱셈에 의해 얻어지기 때문이다. Strassen 방법은 문제의 크기를 반으로 나눈다. 이것은 각각의 행렬을 4분의 1씩(즉, 각각이 $N/2$ -by- $N/2$ 이다) 나누는 것과 같다. 남은 문제는 2행-2열의 행렬을 곱하는 것이다. 단지 곱셈의 수를 줄이기 위해, 다항식 곱셈 문제의 항을 조합하는 것을 통해 네 개를 셋으로 줄였던 것과 같이, Strassen은 행렬 곱셈 문제를 위해 항들을 조합하여 2행-2열의 행렬 곱셈의 수를 8에서 7로 줄이는 법을 발견할 수 있었다. 재배열과 필요한 항들은 아주 복잡하다.

성질 36.2 두 개의 N 행- N 열의 행렬은 약 $N^{2.81}$ 곱셈으로 계산될 수 있다.

위에서 논의한 것에서 보면, Strassen 방법을 사용하는 행렬 곱셈을 위해 요구되는 곱셈들의 수는 $M(N) \approx N^{\lg 7} \approx N^{2.81}$ 을 해로서 갖는 $M(N) = 7M(N/2)$ 분할-정복 반복(recurrence)에 의해 정의된다. \square

1968년에 이 결과가 처음 소개되었을 당시에는, 행렬 곱셈을 위해서는 N^3 의 곱셈이 필수적일 것이라고 생각되어져 왔기 때문에, 꽤 놀라운 결과였다. 그래서 이 문제는 최근까지 집중적으로 연구되고, Strassen 방법보다 조금 더 나은 방법이 발견되었다. 행렬의 곱을 위한 최고의 알고리즘은 아직도 발견되지 않은 상태이고, 이것은 컴퓨터 과학(computer science)분야에서 가장 눈에 띄게 연구되고 있는 분야중의 하나이다.

우리가 이제까지 단순히 곱셈의 횟수만을 고려해 왔다는 것을 주목함이 중요하다. 실제 응용분야를 위해서는, 알고리즘을 선택하기 전에, 항을 조합하기 위한 부수적인 덧셈, 뺄셈과 재귀호출에 따른 비용들을 먼저 고찰해봐야 한다. 이러한 비용들은 전적으로 특정 구현방식이나 사용된 컴퓨터에 따라 다르다. 하지만 이러한 오버헤드(overhead)가, 작은 행렬에서, Strassen 알고리즘을 표준 알고리즘보다 더 비효율적으로 만든다. 큰 행렬에서조차도, 입력된 데이터 항목들의 수의 관점에서는, Strassen 방법은 단지 $N^{1.5}$ 에서 $N^{1.41}$ 로의 개선만을 보여준다. 이러한 개선은 아주 큰 N 의 경우를 제외하고는 되어지기 어렵다. 예를 들어, Strassen 방법이 표준 방법보다 곱셈의 수에서 4분의 1(1/4)만큼만 사용하게끔 되려면, N 값은 적어도 일 백만 보다는 커야 할 것이다.(비록 곱셈당 오버헤드가 표준 방식의 경우 4배가 되기는 하지만) 이처럼 Strassen 알고리즘은 실제적이라기 보다는 이론적인 면에서 가치 있는 공헌을 했다.

연습문제

1. 다항식은 $r_0(x - r_1)(x - r_2) \cdots (x - r_N)$ 의 형태로 표현될 수 있다. 이 방법으로 표현된 두 개의 다항식을 어떻게 곱할 수 있을까?
2. 연결-리스트 표현을 사용하여 계수가 0인 항들이 없는 노드들에 대해 스파이스 다항식을 계산하는 C++프로그램을 작성하라.
3. 스파이스 행렬의 i 번째 행과 j 번째 열의 원소 값을 v 로 세트(set) 하는 C++함수를 작성하라. 행렬은 계수가 0인 엔트리들이 있는 노드가 없는 연결 리스트로 표현된다고 가정하라.
4. 알려진 루트 r_1, r_2, \dots, r_N 을 가지고 다항식을 계산하는 방법을 기술하라. 그리고 Horner 방법과 비교하라.
5. Horner방법을 이용하여 다항식을 계산하는 프로그램을 작성하라. 다항식은 연결-리스트들로서 표현된다. 스파이스 다항식에 대해서 효율적으로 동작하도록 작성하라.
6. Lagrange 보간법을 수행하는 N^2 프로그램을 작성하라.
7. x^{55} 은 아홉 번 미만의 횟수의 곱셈으로 계산될 수 있는가? 만약 그렇다면, 그 방법에 대해 말하고, 아니면 왜 아닌지에 대해 답하라.
8. 이 책에 나타난 분할-정복 방법인 mult가 $1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8$ 을 제공하는데 사용될 때 수행되는 다항식 곱셈들을 나열하라.
9. mult 방법은, 만약 입력의 모든 계수들이 0이라면 0을 리턴 함으로서 스파이스 다항식에서 더 효율적으로 행해질 수 있다. 이 프로그램에서 얼마나 많은 수의 곱셈이 $1 + x^N$ 를 제공하는데에 사용될까?
10. 연결-리스트 표현을 사용하는 한 버전의 mult 함수를 구현하고,(연결-리스트 표현을 사용하는) 주먹구구식 방법보다는 이 버전이 더 빨리 동작할 수 있는 경우의 N 의 값을 경험적으로 찾아라.

빈 면

37 장

Gauss 소거법

가장 근본적인 과학 계산 원리 중의 하나가 연립 방정식(systems of simultaneous equation) 시스템의 해법이다. 이 방정식 시스템을 풀기 위한 기본적인 알고리즘으로 Gauss 소거법(Gaussian elimination)이 있는데, 이 알고리즘은 간단하며 만들어진 이래 150년 동안 거의 변경이 없었다. 이 알고리즘은 특히 지난 20년간 잘 이해되어 졌고, 그래서 정확한 결과 값을 효율적으로 산출할 것이라는 확신을 갖고 사용될 수 있다.

Gauss 소거법은 대부분 컴퓨터들의 설치시에 공급되어 사용될 수 있는 알고리즘의 한 예이다; 실제로, 여러 컴퓨터 언어들의, 특히 APL과 Basic의 경우, 기본 함수(primitive)이다. 이 알고리즘은 이해와 구현이 쉽다. 그러나 특별한 경우, 한 표준화된 서브루틴(subroutine)으로 사용하는 것보다는 수정하여 사용하는 것이 바람직하다. 또한, 이 방식은 오늘날 사용되는 가장 중요한 수치 해석 방법들 중의 하나로서 연구될 만한 가치가 있다.

이제 까지와 마찬가지로, 여기에서도 단지 기본 원리에 중점을 두어 설명할 것이다. 기본 방법을 이해하는 데는 선형 대수에 익숙하지 않아도 된다. 우리는 여기서 간단한 응용 분야들을 위한 라이브러리(library) 서브루틴 보다 더 쉬울 수 있는 한 루틴을 C++로 구현한다. 또한 발생 가능한 어려움들을 보여주는 예제들은 살펴볼 것이다. 크고 중요한 응용 분야의 경우는 기본 수학에 대한 어느 정도의 이해와 아주 숙달되고 세련된 구현이 요구된다.

단순 예제(A Simple Example)

다음과 같은 세 개의 방정식과 세 개의 변수 x , y , z 가 있다고 가정하자:

$$x + 3y - 4z = 8,$$

$$x + y - 2z = 2,$$

$$-x - 2y + 5z = -1.$$

우리의 목표는 이 방정식들을 모두 만족시키는 변수값들을 계산하는 것이다. 경우에 따라서는, 이런 문제에 대한 해가 없을 수도 있고(예를 들면, 두 개의 식이 $x + y = 1$, $x + y = 2$ 처럼 모순되는 경우) 많은 해가 있을 수도 있다.(예를 들어, 두 개의 식이 같거나 식들보다 변수가 더 많을 경우) 앞으로는 방정식의 개수와 변수의 개수가 같다고 가정한다. 그리고, 유일한 해가 있을때 그 해를 찾는 알고리즘을 살펴볼 것이다.

방정식들이 세 점(변수)이상을 다룰 수 있도록 확장하기 위해, 다음과 같이 변수명을 첨자를 붙여서 만들기로 한다:

$$x_1 + 3x_2 - 4x_3 = 8,$$

$$x_1 + x_2 - 2x_3 = 2,$$

$$-x_1 - 2x_2 + 5x_3 = -1.$$

변수들의 반복적 사용을 피하기 위해, 연립 방정식을 행렬을 이용하여 표현하면 편리하다. 위 식을 행렬로 표현하면 다음과 같다.

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 2 \\ -1 \end{pmatrix}$$

이런 방정식들에 대해 해의 변경없이 될 수 있는 몇 가지 연산 동작들이 있다:

- 방정식 교환: 방정식들이 쓰여진 순서는 확실히 해에 아무런 영향을 주지 않는다. 행렬 표현 식에서, 이 동작은 행렬(그리고 오른쪽 벡터)에서 행을 바꾸는 것과 동일하다.
- 변수명 교환: 이것은 행렬의 열을 교환하는 것에 해당한다.(만약 열 i 와 j 가 교환되었다면, 변수 x_i 와 x_j 도 반드시 교환되어야 한다)
- 방정식에 상수 값 곱하기: 이것은 행렬의 한 행(그리고 오른쪽 벡터의 해당 원소)에 한 상수를 곱하는 것에 해당한다.
- 방정식 치환: 두 방정식을 더한 후 그 중 한 방정식과 치환시키는 것.

이러한 연산 동작들이, 특히 마지막 동작의 경우, 방정식들의 해에 어떤 영향도 주지 않을
을 알려면, 잠시 생각을 해봐야 한다. 예로서, 위 연립 방정식의 첫째식과 둘째식의 차를 구
하여 둘째식과 치환하면 다음과 같은 행렬식이 된다:

$$\begin{pmatrix} 1 & 3 & -4 \\ 0 & 2 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 6 \\ -1 \end{pmatrix}$$

두 번째식의 x_1 이 제거됐음에 주목하라. 이와 유사하게, 첫째식과 셋째식을 더하여 셋째식
과 치환하면 x_1 이 제거된다:

$$\begin{pmatrix} 1 & 3 & -4 \\ 0 & 2 & -2 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 6 \\ 7 \end{pmatrix}$$

이제 첫째식을 제외한 모든 식에서 변수 x_1 이 제거되었다. 이런 방법을 체계적으로 수행함
으로써, 원래의 식을 풀기 쉬운 형태로 변경할 수 있다. 이 예제의 경우, 앞서 행한 두 연산
동작을 결합한 한 단계만이 더 남아 있다: 둘째식과 셋째식을 두 배수 한 값의 차를 구하여
셋째식과 치환한다. 이렇게 함으로써, 주 대각선(main diagonal) 아래의 원소들을 모두 0으로
만든다. 특히, 이런 형태의 방정식은 풀기가 쉽다. 이렇게 해서 얻어진 연립 방정식들은 다음
과 같다:

$$\begin{aligned} x_1 + 3x_2 - 4x_3 &= 8, \\ 2x_2 - 2x_3 &= 6, \\ -4x_3 &= -8. \end{aligned}$$

이제 셋째식은 즉시 풀 수 있다: $x_3 = 2$ 이다. 이 값을 두 번째식에 대입하면 x_2 가 구해진다:

$$\begin{aligned} 2x_2 - 4 &= 6, \\ x_2 &= 5. \end{aligned}$$

이와 유사하게, 이 두 값을 첫째식에 대입하면 x_1 이 구해진다:

$$\begin{aligned} x_1 + 15 - 8 &= 8, \\ x_1 &= 1, \end{aligned}$$

이렇게 하여 연립 방정식의 해가 구해졌다.

이 예제는 Gauss 소거법의 두 기본적 단계를 설명해 주고 있다. 첫째는, 순방향-소거(forward-elimination)단계이다; 이 단계에서는, 원래의 방정식에서 변수들이 체계적으로 제거되어 주 대각선 밑이 모두 0이 된다. 이 과정은 종종 삼각 행렬화(triangularization) 과정이라 불린다. 둘째는 역방향-치환(backward-substitution) 단계로서, 이 단계에서는 첫 단계에서 구해진 삼각 행렬을 사용하여 변수들의 값을 계산한다.

방법 개요(Outline of the Method)

일반적으로, 우리는 N 개의 방정식과 N 개의 미지수를 갖는 방정식 시스템을 풀기를 원한다:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1N}x_N &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \cdots + a_{NN}x_N &= b_N. \end{aligned}$$

이를 행렬식으로 표현하면 다음과 같다:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix}$$

또는 이 식을 간단히 $Ax = b$ 로 표시한다. 여기서, A 는 행렬을 나타내고, x 는 변수 벡터를 나타내며, b 는 오른쪽 벡터를 나타낸다. A 의 행들은 b 의 원소들과 함께 조작되므로, b 를 A 의 $(N + 1)$ 번째 열로 간주하여 N 행- $(N + 1)$ 열의 배열을 사용하여 함께 저장하면 편리하다.

이제 순방향-소거 단계(forward-elimination phase)는 다음과 같이 요약될 수 있다: 첫째식을 제외한 모든 다른 식들에서, 첫째식에 적당한 값을 곱해서 각각의 다른 식들과 더함으로써, 첫 번째 변수를 제거하고, 그 다음 둘째식에 적당한 값을 곱해서 세째식 이하의 모든 식들과 각각 더하여, 첫 번째 두 식을 제외한 모든 식에서 두 번째 변수를 제거한다. 이런 식으로 계속하여 변수들을 제거한다. j 번째 방정식($i + 1$ 과 N 사이의 j)에서 i 번째 변수를 제거하기 위해서는, i 번째 방정식에 a_{ji}/a_{ii} 를 곱하고, j 번째식에서 그것을 빼 주면 된다. 이 과정이 아래의 한 부분 코드(partial code)에 보다 간결하게 나타나 있다:

```

for (i = 1; i <= N; i++)
    for (j = i+1; j <= N; j++)
        for (k = N+1; k >= i; k--)
            a[j][k] -= a[i][k]*a[j][i]/a[i][i];

```

이 부분 코드는 세 개의 중첩 루프로 구성되어 있고, 총 수행 시간은 필연적으로 N^3 에 비례한다. 세 번째 루프는 $a[j][i]$ 의 값이 동일 행의 다른 원소들의 값을 조정하는데 사용되기 전에 파괴(변경)되는 것을 막기 위해서 역 방향으로 수행된다.

위 부분 코드는 올바르게 동작하기에는 너무 단순화되어 있다: $a[i][i]$ 가 영(0)이 되면, 영(0)으로-나누기(division-by-zero)가 발생할 수도 있다. 하지만, 이것은 쉽게 고쳐질 수 있다. 왜냐하면 외부 루프에서 $a[i][i]$ 를 영(0)이 아닌 수로 만들기 위해서 i 번째 행을 다른 행($i+1$ 과 N 사이의)과 교환할 수 있기 때문이다. 만약 이런 행들을 찾을 수 없다면, 이 행렬은 singlar(singular)이다: 이 경우에는 유일한 해가 없다.(이 경우 프로그램이 이를 보고하게 만들거나, 궁극적으로 영으로-나누기가 발생하게 하여 이를 알게 할 수 있다) 우리는 i 번째 열에 영(0)이 아닌 원소를 갖는 i 번째 행 밑의 행을 찾아서 i 번째 행과 바꾸기 위해, 위 부분 코드에 코드를 더 추가해야 한다. $a[i][i]$ 는 궁극적으로, i 번째 열에서 주 대각선 아래의 영이 아닌 원소들을 제거하기 위해 사용되는데, 이 $a[i][i]$ 를 피보트(pivot)라 한다.

실제로, 단지 i 번째 열에서 영(0)이 아닌 값을 갖는 행을 찾는 것 외에도 권장할 만한 수행 동작이 있다.($i+1$ 에서 N 번째)행 중에서 그 행의 i 번째 열의 값의 절대값이 최대인 행을 택하는 것이 최고로 좋다. 그 이유는 만약 행을 축척(scale)하는데 이용된 피보트 값이 아주 작으면 심각한 계산상의 오차가 발생할 수 있기 때문이다. 만약 $a[i][i]$ 가 아주 작으면, j 번째식에서($i+1$ 과 N 사이의) i 번째 변수를 제거하기 위해 사용된 축척 인자(scaling factor) $a[j][i]/a[i][i]$ 가 매우 크게 될 것이다. 실제로, 이 인자가 너무 커서 실제 계수 $a[j][k]$ 를 “반올림 오차(round-off-error)”에 의해 문제가 될 정도로 작게 만들 수 있다.

간단히 말해서, 크기가 너무 많이 차이나는 숫자들은 실수를 나타내기 위해 사용되는 부동소수점 체계에서는 정확하게 더해지거나 빼질 수 없다; 그리고, 작은 피보트 값을 사용하면 이같은 연산이 행해질 확률이 아주 높아진다. $i+1$ 행부터 N 행까지에서 i 번째 열에 있는 최대 값을 사용하면 축척 인자가 항상 1보다 작게 됨을 보증할 수 있고 이런 유형의 오차를 방지할 수가 있다. 혹자는 큰 원소 값을 얻기 위해 i 번째 열 외의 것들에 대해 심각하게 고려할 수도 있다. 그러나 그렇게 하지 않고도 정확한 값들이 얻어짐이 밝혀져 있다.

가우스 소거법의 순방향-소거 단계를 위한 아래 프로그램 코드는 이 과정을 간단히 구현한 것이다. 1부터 N사이의 각 i에 대해,(i번째 뒤의 행에서) 가장 큰 값을 갖는 i번째 열을 찾아 내려간다. 이 원소를 갖고 있는 행은 i번째 행과 교환되고, 그리하여 i+1번째에서 N번째까지의 방정식에서 i번째 변수가 제거된다:

```
eliminate()
{
    int i, j, k, max;
    float t;
    for (i=1; i<=N; i++)
    {
        max=i;
        for (j=i+1; j<=N; j++)
            if (abs(a[j][i])>abs(a[max][i])) max=j;
        for (k=i; k<=N+1; k++)
        {
            t=a[i][k];
            a[i][k]=a[max][k];
            a[max][k]=t;
        }
        for (j = i+1; j <= N; j++)
            for (k = N+1; k >= i; k--)
                a[j][k] -= a[i][k]*a[j][i]/a[i][i];
    }
}
```

어떤 알고리즘들에서는, i번째를 제외한 모든 방정식에서 i번째 변수를 제거하기 위해 (i+1번째부터 N번째까지가 아니라) 피보트 $a[i][i]$ 를 사용함이 필요하다. 이런 과정을 완전-피보팅(full pivoting)이라 한다; 순방향-소거 시에는 단지 이 작업의 일부만이 수행되므로, 그 과정을 부분-피보팅(partial pivoting)이라 한다. 43장에서 완전 피보팅 알고리즘을 다룰 것이다.

순방향-소거 단계가 완료된 후, 배열 a의 대각선 아래의 배열 값들은 모두 0이 되고 역방향-치환 단계가 시작될 수가 있게 된다.

```
substitute()
{
```

```

int j, k;
float t;
for (j = N; j >= 1; j--)
{
    t=0.0;
    for (k = j+1; k <= N; k++) t += a[j][k]*x[k];
    x[j] = (a[j][N+1]-t)/a[j][j];
}
}

```

eliminate의 호출 뒤에 substitute를 호출함으로서 N -원소를 갖는 배열 x 에서의 해가 계산된다. 영으로-나누기 연산 동작은 싱글러 행렬들에 대해 아직도 발생할 수 있다. 이러한 발생은 라이브러리 루틴을 이용하면 명백히 조사될 수도 있을 것이다. 실제로, 대부분의 라이브러리 루틴들은, 아래에서 더 논의되겠지만, 훨씬 더 광범위한 검사를 수행한다.

성질 37.1 N 개의 미지수를 갖는 N 개의 방정식으로 구성된 연립방정식은 약 $N^3/3$ 번의 곱셈과 덧셈을 사용하여 풀 수 있다.

substitute의 수행 시간은 $O(N^2)$ 이다. 그래서, 대부분의 작업은 eliminate에서 행해진다. 이 루틴을 자세히 살펴보면, 각 i 값에 대해, k 루프는 $N - i + 2$ 번, 그리고 j 루프는 $N - i$ 번 수행되는 것을 알 수 있다; 이것은 내부루프가 $\sum_{1 \leq i < N} (N - i + 2)(N - i) = N^3/3 + O(N^2)$ 번 수행됨을 뜻한다. $-a[j][i]/a[i][i]$ 의 값은 k 루프 밖에서 계산될 수 있다. 그래서, 내부루프는 한번의 곱셈과 한번의 덧셈으로 구성된다. \square

순방향-소거가 대각선 밑을 모두 영(0)으로 만든 뒤에 행해지는 또 한 가지의 방법은, 같은 방법을 이용하여 대각선 위를 모두 영(0)으로 만드는 것이다: 먼저, $a[N][N]$ 을 제외한 마지막 열의 모든 값을 $a[N][N]$ 에 적절한 상수 값을 곱해서 나온 값을 더함으로써 영(0)으로 만든다. 똑같은 방법을 마지막 열 바로 옆의 열에 적용한다. 이런 작업을 옆의 열들에 대해 계속 반복한다. 즉, 각 열의 “다른” 부분에 대해 역순으로 “부분-피보팅”을 수행한다. Gauss-Jordan 축약법(reduction)이라 불리는 이 과정이 끝나면, 대각선상의 원소들만이 영(0)이 아닌 값으로 남게 된다. 그리하여 아주 단순하게 해를 구하게 된다. 그러나, 이 과정에서 사용된 산술 연산의 횟수는 역방향-치환의 경우보다 훨씬 더 많다.

Gauss 소거법에서의 주된 관심사는 계산 오차(computational error)이다. 앞서 언급한 것처럼, 계수들의 크기가 아주 다른 경우들에 대해서 주의해야 한다. 부분-피보팅을 위해 열에 있는 가장 큰 원소를 사용하면 피보팅 과정에서 큰 계수 값들이 임의로 생성되지 않음이 보장된다. 그러나, 심각한 오차를 항상 피해 갈 수는 없다. 예로서, 두 다른 방정식이 서로 아주 비슷한 계수 값들을 가질 때, 아주 작은 계수 값들이 나타날 수 있다. 그러나 실제로는 이러한 문제들이 부정확한 해가 되게 만드는지를 미리 알 수가 있다. 각 행렬에는 계산된 답의 정확도를 측정하는데 사용될 수 있는 조건 번호(condition number)라 불리는 숫자가 부여되어 있다. Gauss 소거법에 대한 좋은 라이브러리 서브루틴은 해 뿐만 아니라 행렬의 조건 번호도 계산할 것이다; 그래서 그 해의 정확도를 알 수가 있다. 이와 관련된 문제점들을 여기서 다루는 것은 이 책의 범위를 벗어난다.

가능한 최대의 피보트 값을 사용하여 부분-피보팅을 하는 Gauss 소거법은 매우 작은 계산 오차 내에서 결과값들을 산출하는 것을 “보장한다”. 아주 나쁜 조건을 갖는 행렬들의 경우를 제외한(이 나쁜 조건의 행렬은 해법상의 문제라기보다는 방정식상에 문제가 있음에 대한 표시일 수 있다) 행렬들에 대해 계산된 답이 아주 정확함을 보여주는, 아주 조심스럽게 연구하여 나온, 수학적 결과가 있다. Gauss 소거 알고리즘은 상당히 세밀한 이론적 연구의 주제가 되어 왔고 아주 넓은 응용성을 갖는 계산 프로시저로서 추천될 만 하다.

변형과 확장(Variations and Extensions)

앞에서 설명한 방법은 대부분의 N^2 개의 원소가 영(0)이 아닌 N 행- N 열의 행렬에 적합하다. 다른 문제들에서 본 것처럼, 대부분의 원소가 영(0)인 스파이스 행렬들의 경우에는 특수한 기법들을 사용함이 좋다. 이런 상황은 연립 방정식의 각 식이 단지 몇 개의 항만을 가지는 경우에 해당한다.

만약 영(0)이 아닌 원소들이 어떤 특별한 구조를 갖고 있지 않다면, 36장에서 논의한 각 영(0)이 아닌 원소에 대해 한 노드씩이 부여되고 행과 열 모두에 의해 링크된 연결-리스트 표현이 적합하다. 이 표현에 대해 영(0)이 아닌 원소들을 만들고 제거할 필요성이 있음에 기인하여 추가된 복잡성은 있지만) 한 표준적인 방법이 구현 될 수 있다. 이기법은 만약 행렬 전체가 메모리 내에 있을 수 있다면 별로 가치가 없는 듯 하다. 왜냐하면 이 기법은 일반적인 방법보다 아주 많이 복잡하기 때문이다. 또한, 스파이스 행렬들은 Gauss 소거 과정 동안 스파이스 한 것이 상당히 없어지기 때문이다.

어떤 행렬들은 단지 소수의 영(0)이 아닌 원소들을 가질 뿐만 아니라 아주 단순한 구조를 가질 수도 있다. 따라서 이런 경우에는 연결-리스트의 사용이 불필요하다. 이런 경우의 가장 일반적인 예는 “띠(band)”행렬이다. 이 행렬에서는, 영(0)이 아닌 원소들은 대각선 근처에 모두 몰려 있다. 이러한 경우에는, Gauss 소거 알고리즘들의 내부루프들은 아주 작은 횟수만큼 반복 수행되므로 총 수행 시간(그리고 필요로 되는 저장 장소의 양)이 N^3 이 아니라 N 에 비례한다.

띠 행렬의 한 재미있고 특별한 경우는 “삼선 대각선(tridiagonal)” 행렬인데, 이 행렬에서는 주 대각선과 그 바로 위 선과 그 바로 밑 선상의 원소들만이 영(0)이 아니다. 예로서, $N = 5$ 인 경우의 삼선 대각선의 일반적인 형태는 다음과 같다:

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{pmatrix}$$

이같은 행렬들의 경우에는, 순방향-소거와 역방향-치환 프로그램 코드 각각이 한 개의 for 루프 문장으로 줄어든다:

```
for (i=1; i<N; i++)
{
    a[i+1][N+1] -= a[i][N+1]*a[i+1][i]/a[i][i];
    a[i+1][i+1] -= a[i][i+1]*a[i+1][i]/a[i][i];
}
for (j = N; j >= 1; j--)
    x[j] = (a[j][N+1] - a[j][j+1]*x[j+1])/a[j][j];
```

순방향-소거의 경우에는, 단지 $j=i+1$ 과 $k=i+1$ 만 포함될 필요가 있다. 왜냐하면 $a[i][k]$ 는 $i+1$ 보다 큰 k 에 대해서는 영(0)이기 때문이다. ($k=i$ 의 경우는 생략될 수 있다. 왜냐하면, 이 경우 다시 검사되지 않는 배열 원소들은 영(0)이 되기 때문이다-이와 같은 변경은 단순한 Gauss 소거법의 경우에도 가능하다)

성질 37.2 삼선 대각선형의 연립방정식은 선형 시간 내에 풀 수 있다.

물론, 크기 N^2 의 이차원 배열은 삼선 대각선형의 행렬에는 사용되지 않는다. 위 프로그램에서 필요한 저장 장소는 한 행렬 대신 네 개의 배열(세 개의 대각선 각각에 대해 한 배열씩, 그리고 $(N + 1)$ 번째 열에 대해 한 배열)을 가짐으로서 N 에 선형이 될 수가 있다. 반드시 가장 큰 원소 상에서 피보팅을 할 필요는 없으므로, 영으로-나누기나 계산 오차들의 누적에 대한 어떤 대비나 보호가 없음에 주목하라. 그러나 번번하게 발생하는 일부 삼선 대각선형의 행렬들의 경우에는, 이런 문제들이 관심거리가 될 이유가 없음이 증명될 수 있다. \square

Gauss-Jordan 축약법은 한 행렬을 한 번에 역으로 치환시키는 완전-피보팅을 가지고 구현될 수 있다. A^{-1} 로 표시되는 행렬 A 의 역은 연립 방정식 $Ax = b$ 가 행렬 곱 $x = A^{-1}b$ 를 수행함으로써 바로 풀리게 하는 성질을 가진다. 그래도 역시 b 가 주어졌을 때 x 를 계산하는데는 N^3 의 연산 동작이 요구된다. 그러나, 행렬을 선처리(preprocess)하여 부분 행렬들로 “분할”할 수가 있다. 이 분할은 해당 방정식이(b 가 주어졌을 때) N^2 에 비례하는 시간 내에 풀리게 해줌으로써 Gauss 소거를 사용할 때에 비해 N 의 비율에 해당하는 시간 절약을 가져온다. 대략적으로, 이 방법은 순방향-소거 단계 시에($N + 1$)번째의 열에 대해 수행된 연산들을 기억하므로, 새로운($N + 1$)번째 열에 대한 순방향-소거의 결과가 효율적으로 계산될 수 있다. 역방향-치환은 전과 마찬가지로 수행된다.

선형 방정식 시스템들을 푸는 것은 행렬 곱셈과 계산적으로 동치임이 알려져 있다. 따라서, N 개의 변수와 N 개의 식을 갖는 연립 방정식을 $N^{2.81}$ 에 비례하는 시간 내에 풀 수 있는 알고리즘들이 있다. 행렬 곱셈의 경우와 마찬가지로, 이같은 방법의 사용은 아주 큰 연립 방정식이 정기적으로 처리되어야 하지 않는 한 별 가치가 없다. 입력 개수의 관점에서, Gauss 소거법의 실질적인 수행 시간은 $N^{3/2}$ 이다. 실제적인 경우, 이 수행 시간을 개선하기는 쉽지 않다.

연습문제

1. Gauss 소거법이 $x + y + z = 6$, $2x + y + 3z = 12$, 그리고 $3x + y + 3z = 14$ 로 구성된 연립 방정식을 풀기 위해 사용될 때, 순방향-소거(eliminate)에 의해 산출되는 행렬을 구하라:
2. 순방향-소거를 위한 세번 중첩된 for 루프를 갖는 구현 방식이 실패하는(비록 해는 있지만), 세 개의 미지수와 세 개의 방정식을 갖는 시스템을 찾아라.
3. 단지 $3N$ 개의 영(0)이 아닌 원소를 갖는 N 행- N 열의 행렬에서의 Gauss 소거시에 필요한 저장 장소는 얼마나 되나?
4. 한 행이 모두 영(0)인 행렬에 대해 eliminate가 사용되면 어떤 일이 발생하는지를 설명하라.
5. 한 열이 모두 영(0)인 행렬에 대해 eliminate와 substitute를 이어서 사용하면 어떤 일이 발생하는지를 설명하라.
6. Gauss-Jordan 축약에서는 대략 얼마나 많은 산술 연산이 사용(수행)되는가?
7. 만약 행렬의 열들을 상호 교환하면, 해당 연립 방정식에 어떤 영향이 있을까?
8. eliminate를 사용하여 어떻게 방정식들이 서로 상치되는(contradictory) 경우를 조사할 수 있을까? 방정식들이 서로 같은 경우에는 어떻게 조사할 수 있을까?
9. N 개의 미지수와 N 개의 방정식을 갖는 시스템에서, $M < N$ 인 경우, Gauss 소거법이 어떻게 이용될 수 있을까? 만약 $M > N$ 이면, 어떠할까?
10. 단지 두자리 유효숫자(significant digit)만을 갖고(단, 모든 숫자들은 한자리 정수 x, y, z 에 대해 $x, y \times 10^z$ 의 형태를 가져야만 한다) 숫자들이 표현될 수 있는 가상의 아주 원시적인 컴퓨터를 이용하여, 최대 값을 갖는 원소에 대해 피보팅을 할 필요성을 보여주는 예를 설명하라

빈 면

곡선 또는 데이터 맞춤(curve or data fitting)이란 용어는 주어진 점들의 집합에서 관측된 값들을 매치(match)시키는 함수를 찾는 일반적인 문제의 설명시에 사용된다. 특히, 이 점들의 집합

$$x_1, x_2, \dots, x_N$$

과 이에 상응하는 값들의 집합인

$$y_1, y_2, \dots, y_N$$

이 주어졌을 때,

$$f(x_1) = y_1, f(x_2) = y_2, \dots, f(x_N) = y_N$$

이고 $f(x)$ 가 다른 데이터 점들에서 “타당한(reasonable)” 값을 가정할 때,(아마도 특정 유형의) 한 함수를 찾는 것이 목표이다. x 들과 y 들이 어떤 알려지지 않은 함수와 연관이 있을 때, 그 함수를 찾는 것이 목표일 수 있다. 그러나, 일반적으로, “타당한”이라는 말의 정의는 응용 분야에 따라 다르다. “타당치 않은” 함수들을 쉽게 식별하는 것이 쉬움을 종종 보게 될 것이다.

곡선 맞춤은 실험 데이터 분석에 관한 분야 뿐만 아니라 다른 많은 분야에서도 사용된다. 예로써, 아주 많은 플롯(plot)되어야 할 점들을 저장하는 오버헤드 없이 “보기에 좋은” 곡선들을 만들어 내기 위해, 컴퓨터 그래픽스 분야에서 사용될 수 있다. 곡선 맞춤의 한 응용 분야는 한 임의의 점에서 한 알려진 함수값의 계산을 위한 빠른 알고리즘을 만드는 것이다: 즉, 정확한 값들로 구성된 작은 도표를 가지고, 다른 값들을 찾기 위해 곡선 맞춤을 하는 것이다.

이 문제 해결을 위한 접근 방법으로서 두 가지 주된 방법이 사용된다. 첫 번째는 보간법(interpolation)이다: 이 방식으로, 주어진 값들을 주어진 점들에 정확히 매치시켜주는 매끄러운(smooth) 함수가 찾아진다. 두 번째 방법은 최소 제곱(least-squares) 데이터 맞춤 방법인데, 이 방법은 주어진 값들이 정확하지 않을 수 있으나 이들을 가능한 한 점들과 잘 매치시키는 함수를 찾는 것이다.

다항식 보간(Polynomial Interpolation)

이미 데이터 맞춤 문제를 풀기 위한 방법으로서, “만약 f 가 차수 $N - 1$ 의 다항식이면, 다항식 보간법 문제이다”라는 한 가지 방법을 36장에서 알았다. 비록 f 에 대한 특정 지식이 없더라도, $f(x)$ 를 주어진 점들과 값들에 대한 차수 $N - 1$ 의 다항식 보간법이 되게 함으로써 데이터 맞춤 문제를 풀 수 있다. 이 푸는 방법은 36장에서 개략적으로 설명된 방법들을 사용하면 가능할 것이다. 그러나 데이터 맞춤 문제에 대해 다항식 보간법을 사용하지 않는 많은 이유가 있다. 그 한 가지 이유는 계산량이 상당하다는 것이다.(고수준의 방법들의 경우에는 $N(\log N)^2$ 이다. 그러나 기본 방법들은 이차식을 필요로 한다) 예로서, 차수가 100인 다항식 계산은 100개의 점들로 형성된 곡선을 보간하는데에는 너무도 과다한 듯 싶다.

다항식 보간의 주된 단점은 차수가 높은 다항식은 맞춤이 되어지는 함수에 잘 맞지 않는 예기치 않는 성질들을 가질 수 있는 복잡한 함수들이라는 것이다. 고전적인 수학에서 나온 한 결과(Weierstrass 근사치 정리)를 보면 한 합리적인 함수를 충분히 높은 차수의 다항식에 근접하게 만드는 것이 가능하다. 불행히도, 매우 높은 차수의 다항식은 아주 심하게 변동하는 경향이 있다. 비록 대부분의 함수들이 보간법에 의해 거의 모든 폐구간(closed interval) 상에서 아주 근사치에 가깝게 만들어질 수 있다 하더라도, 그러한 근사치가 아주 엉망으로 얻어지는 경우들이 항상 발생할 수 있음이 알려져 있다. 게다가, 이 이론은 데이터 값들이 어느 모르는 함수로부터 나온 정확한 값들이라는 가정하에 만들어진 것이다. 그러나, 주어진 데이터 값들은 단지 근사치인 경우가 자주있다. 만약 y 값들이 미지의 낮은 차수의 다항식에서 나온 값들이라면, 우리는 다항식 보간에서의 고차수 항들의 계수가 영(0)이기를 기대할 수 있다. 그러나 그렇게 되지 않는 경우가 대부분이다; 그 대신, 다항식 보간시 정확한 맞춤(fit)을 얻기 위해 그 고차수 항들을 이용할 것이다. 이런 것들이 다항식 보간을 많은 곡선 맞춤 응용 분야에 적합하지 않게 만든다.

스플라인 보간(Spline Interpolation)

낮은 차수의 다항식들은 분석하기 쉬운 단순 곡선들이고, 그래서 곡선 맞춤에 널리 이용되고 있다. 방법은 한 다항식이 모든 점들을 처리하게 하지 않고, 그 대신 여러 다항식을 사용하여 인접 점들을 연결하고, 이들을 매끄럽게 연결하는 것이다. 이런 방식의 한 세련되고 특수한 경우(이 경우 또한, 상대적으로 간단한 계산을 포함한다)를 스플라인 보간(spline interpolation)이라고 한다.

한 스플라인은 도안자가 미학적으로 좋은 곡선들을 그리는데 사용하는 기계적 장치이다: 이 도안자는 그가 그리는 그림상에 노트(knot)라 불리는 점들의 집합으로 고정시키고, 그 점들 주변에서 유연한 플라스틱이나 목재(즉, 스플라인)들을 구부려서 이들이 곡선을 만들도록 한다. 스플라인 보간법은 이 과정과 수학적으로 같으므로 결과적으로 동일한 곡선을 만들어 낸다. 그림 38.1은 열 개의 노트를 갖는 한 스플라인을 보여준다.

두 인접 노트간의 스플라인이 갖는 것으로 가정된 형상은 삼차 다항식임을 기초 기계역학을 이용하여 보여줄 수 있다. 우리의 데이터 맞춤 문제로 해석하면 이는 아래와 같은 $N - 1$ 개의 다른 삼차 다항식들로 표현된 곡선을 고려해야 함을 의미한다:

$$s_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i, \quad i = 1, 2, \dots, N - 1,$$

여기서 $s_i(x)$ 는 구간 $[x_i, x_{i+1}]$ 에서 사용될 삼차 다항식으로서 정의된다.

스플라인은 네개의 일차원 배열(즉, 4-by-($N - 1$) 2차원 배열)로서 명백히 표현될 수 있다. 한 스플라인의 생성은 주어진 x 개의 점들과 y 개의 점들로부터 필요한 a, b, c, d 계수들을 계산하는 것으로 구성된다. 이 스플라인에 대한 물리적 제약 조건들은 계수 값들의 산출을 위해 풀어야 할 연립 방정식들에 해당한다.

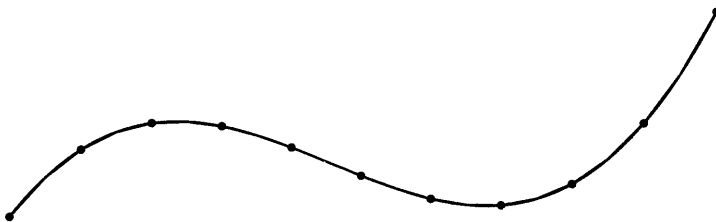


그림 38.1 열 개의 노트를 갖는 한 스플라인.

예를 들어, $i = 1, 2, \dots, N - 1$ 의 경우, 명백히 $s_i(x_i) = y_i$ 이고 $s_i(x_{i+1}) = y_{i+1}$ 이어야 한다. 왜냐하면 스플라인은 노트들을 거쳐가야만 하기 때문이다. 이 뿐만 아니라, 스플라인은 어떤 예리한 구부러진 부분 없이 노트들 주변에서 매끄럽게 곡선을 형성해야 한다. 수학적으로, 이 말은 스플라인 다항식의 일차 미분계수들은 노트들에서 같아야 함($i = 2, 3, \dots, N - 1$ 의 경우, $s'_{i-1}(x_i) = s'_i(x_i)$)을 의미한다. 실제로, 다항식들의 이차 미분 계수들 또한 노트들에서 같아야 함이 밝혀졌다. 이러한 조건들은 $4(N - 1)$ 개의 미지의 계수들이 있을 때 총 $4N - 6$ 개의 방정식을 갖게 한다. 스플라인의 양끝 점들을 설명하려면 조건이 2개 더 규정 되어야 할 필요가 있다.

여러 옵션(option)들이 가능하다; 여기서는 소위 “자연적(natural)”스플라인이라 불리는 것을 사용하는데, 이들은 $s'_1(x_1) = 0$ 과 $s''_{N-1}(x_N) = 0$ 으로부터 유도됐다.

이들 유도 조건들은 $4N - 4$ 개의 미지수가 있을 때, $4N - 4$ 개의 방정식을 갖는 시스템이 생기게 한다. 이 시스템은 가우스 소거법(Gaussian elimination)을 사용하여 그 스플라인을 나타내는 모든 계수들을 계산함으로써 풀 수 있다.

그러나 이와 동일한 스플라인을 좀 더 효율적으로 계산할 수 있다. 왜냐하면 실제로는 $N - 2$ 개의 미지수만이 있기 때문이다: 대부분의 스플라인 조건들은 잉여(redundant)조건들이다. 예를 들어, p_i 가 x_i 에서의 스플라인의 2차 미분 계수 값이고, 그래서, $i = 2, \dots, N - 1$ 이고 $p_1 = p_N = 0$ 인 경우 $s''_{i-1}(x_i) = s''_i(x_i) = p_i$ 라고 하자. 만약 p_1, \dots, p_N 의 값들을 안다면, a, b, c, d 계수들 모두가 스플라인 세그먼트(segment)들에 대해 계산 될 수 있다. 왜냐하면 각 세그먼트에 대해 4개의 미지수와 4개의 방정식을 가지기 때문이다: $i = 1, 2, \dots, N - 1$ 의 경우, 아래 식들을 가져야만 한다.

$$\begin{aligned} s_i(x_i) &= y_i \\ s_i(x_{i+1}) &= y_{i+1} \\ s''_i(x_i) &= p_i \\ s''_i(x_{i+1}) &= p_{i+1} \end{aligned}$$

x 와 y 값은 주어진다; 스플라인을 완전히 결정하기 위해, 단지 p_2, \dots, p_{N-1} 의 값들만 구하면 된다. 이를 위해, 첫 미분 계수들이 매치해야만 한다는 조건을 사용한다: 이 $N - 2$ 개의 조건들은 $N - 2$ 개의 미지수에, 즉 p_i 의 이차 미분값들을 얻기 위해 필요한 $N - 2$ 개의 방정식을 제공한다.

p 의 2차 미분 값들의 관점에서 a, b, c, d 계수를 설명하려면, 이들의 표현 식을 각각의 세그먼트에 대해 위에 기술된 4개의 방정식으로 치환해야 하는데, 이 경우 불필요하게 복잡한 표현 식들이 생길 수 있다. 따라서, 그 대신에 스플라인 세그먼트들에 대한 방정식들을 거의 미지의 계수가 없는 한 정규형(canonical form)으로 표시하는 것이 편리하다. 만약 변수들을 $t = (x - x_i)/(x_{i+1} - x_i)$ 로 바꾸면, 스플라인은 다음과 같이 표현될 수 있을 것이다:

$$s_i(t) = ty_{i+1} + (1 - t)y_i + (x_{i+1} - x_i)^2((t^3 - t)p_{i+1} - ((1 - t)^3 - (1 - t))p_i)/6.$$

이제 각 스플라인은 구간 $[0, 1]$ 상에서 정의된다. 이 방정식은 보기보다는 덜 어렵다. 왜냐하면 우리의 관심사는 주로 끝점인 0과 1이고, 이들 끝점들에서 t 나 $(1 - t)$ 는 0이기 때문이다. 이 표현 식은 스플라인이 보간하고 연속임을 조사하는 것을 사소한 일이 되게 만든다. 왜냐하면 $i = 2, \dots, N - 1$ 에 대해 $s_{i-1}(1) = s_i(0) = y_i$ 이기 때문이다. 이것은 2차 미분 계수가 ($s'_i(1) = s'_{i+1}(0) = p_{i+1}$ 이기 때문에) 연속임을 증명하는 것보다는 약간 더 어렵다. 이들은 끝점들에서 필요한 조건들을 만족시키는 삼차 다항식들이다. 따라서 이들은 위에 기술한 스플라인 세그먼트들과 동일하다. 만약 t 를 대치하고 x^3 등의 계수들을 찾을 수 있다면, 마치 우리가 전 절에서 설명된 방법을 사용한 것처럼, x, y, p 들의 관점에서 a, b, c , 와 d 에 대해 동일한 표현식을 얻을 수도 있을 것이다. 그러나, 그렇게 해야 할 이유가 전혀 없다. 왜냐하면, 우리는 이 스플라인 세그먼트들이 끝점 조건들을 만족시킴을 확인했고 t 를 계산하고 위 공식을 사용함으로써(우리가 p 들만 안다면) 그 구간 내의 임의의 한 점에서 각각을 평가할 수 있기 때문이다.

p 들에 대해 풀려면, 끝점들에서 스플라인 세그먼트들의 일차 미분 계수들을 같게 만들어야 한다. 위 방정식의(x 에 대한) 일차 미분 계수는 다음과 같다.

$$s'_i(t) = z_i + (x_{i+1} - x_i)((3t^2 - 1)p_{i+1} + (3(1 - t)^2 - 1)p_i)/6$$

여기서 $z_i = (y_{i+1} - y_i)/(x_{i+1} - x_i)$ 이다. 이제, $i = 2, \dots, N - 1$ 의 경우, $s'_{i-1}(1) = s'_i(0)$ 로 함으로써 $N - 2$ 개의 방정식을 가진 시스템이 만들어진다.

$$(x_i - x_{i-1})p_{i-1} + 2(x_{i+1} - x_{i-1})p_i + (x_{i+1} - x_i)p_{i+1} = 6(z_i - z_{i-1})$$

이 방정식 시스템은 37장에서 본 가우스 소거법의 한 퇴행적(degenerate)버전을 갖고 쉽게 풀 수 있는 삼선대각선(tridiagonal) 형태이다. 만약 $u_i = x_{i+1} - x_i$, $d_i = 2(x_{i+1} - x_{i-1})$, $w_i = 6(z_i - z_{i-1})$ 이라면, $N = 7$ 인 경우, 아래의 방정식들이 생긴다.

$$\begin{pmatrix} d_2 & u_2 & 0 & 0 & 0 \\ u_2 & d_3 & u_3 & 0 & 0 \\ 0 & u_3 & d_4 & u_4 & 0 \\ 0 & 0 & u_4 & d_5 & u_5 \\ 0 & 0 & 0 & u_5 & d_6 \end{pmatrix} \begin{pmatrix} p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{pmatrix} = \begin{pmatrix} w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{pmatrix}$$

실제로, 이 방정식들은 대칭형 삼선대각선 시스템으로서, 중앙 대각선 바로 밑의 대각선이 중앙 대각선 바로 위와 같다. 이 방정식 시스템의 경우에는 정확한 해를 구하기 위해 최대 원소에 대한 피보팅(pivoting)이 불필요함이 증명되어 있다.

위 삼차식 스플라인 계산을 위해, 위 절에서 설명한 방법은 매우 쉽게 C++로 작성될 수 있다:

```

makespline(float x[], float y[], int N)
{
    for (i = 2; i < N; i++) d[i] = 2*(x[i+1]-x[i-1]);
    for (i = 1; i < N; i++) u[i] = x[i+1]-x[i];
    for (i = 2; i < N; i++)
        w[i] = 6.0*((y[i+1]-y[i])/u[i] - (y[i]-y[i-1])/u[i-1]);
    p[1] = 0.0; p[N] = 0.0;
    for (i = 2; i < N-1; i++)
    {
        w[i+1] = w[i+1] - w[i]*u[i]/d[i];
        d[i+1] = d[i+1] - u[i]*u[i]/d[i];
    }
    for (i = N-1; i>1; i--)
        p[i] = (w[i]-u[i]*p[i+1])/d[i];
}

```

행렬 d 와 u 는 37장의 프로그램을 사용하여 푼 삼선 대각선 행렬을 표현한 것이다. 그 프로그램에서 $a[i][i]$ 가 있는 곳에 $d[i]$ 를 사용했고 $a[i+1][i]$ 나 $a[i][i+1]$ 이 있는 곳에 $u[i]$ 를 사용했다. 그리고 $a[i][N+1]$ 이 있는 곳에 $z[i]$ 는 사용됐다.

성질 38.1 N 개의 점에 대한 삼차식 스플라인은 선형 시간(linear time)내에 형성될 수 있다.

이 사실은 프로그램에서 볼 때 자명하다. 이것은 단순히 그 데이터를 통한 한 연속된 선형 처리 과정이다. \square

삼차식 스플라인 형성의 한 예로서, 한 스플라인을 아래 6개의 점들에 맞추는 경우를 고려해 보자:

$$(1.0, 2.0), (2.0, 1.5), (4.0, 1.25), (5.0, 1.2), (8.0, 1.125), (10.0, 1.1)$$

(이들은 함수 $1+1/x$ 에서부터 얻어진다) 이 스플라인 파라미터들은 다음 방정식 시스템을 풀으로써 찾아진다.

$$\begin{pmatrix} 6 & 2 & 0 & 0 \\ 2 & 6 & 1 & 0 \\ 0 & 1 & 8 & 3 \\ 0 & 0 & 3 & 10 \end{pmatrix} \begin{pmatrix} p_2 \\ p_3 \\ p_4 \\ p_5 \end{pmatrix} = \begin{pmatrix} 2.250 \\ .450 \\ .150 \\ .075 \end{pmatrix}$$

결과치는 $p_2 = 0.39541$, $p_3 = -0.06123$, $p_4 = 0.02658$, $p_5 = -0.00047$ 이다.

영역 $[x_1, x_N]$ 내의 한 임의값 x 에 대한 스플라인을 계산하기 위해서는, 단지 x 를 포함하는 구간 $[x_i, x_{i+1}]$ 을 찾고 나서 t 를 계산하고 $s_i(x)$ 를 얻기 위해 상기 공식을 사용하면 된다.(이 공식은 p_i 와 p_{i+1} 에 대해 계산된 값들을 사용한다)

```
float f(float x)
{ return x*x*x-x; }
float eval(float v)
{
    float t; int i=1;
    while (v>x[i+1]) i++;
    t = (v-x[i])/u[i];
    return t*y[i+1]+(1-t)*y[i]+
        u[i]*u[i]*(f(t)*p[i+1]+f(1-t)*p[i])/6.0;
}
```


이 프로그램은 v 가 $x[1]$ 과 $x[N]$ 사이에 있지 않을 경우의 오류 조건(error condition)에 대한 조사는 하지 않는다. 만약 스플라인 세그먼트들의 수가 크다면(즉, N 이 크면), v 를 포함하는 구간을 찾기 위해 14장에 있는 것과 같은 좀 더 효율적인 검색 기법이 사용될 수도 있다.

다항식들을 “매끄럽게” 결합함으로써 곡선 맞춤을 하는 방식의 많은 변형된 방법들이 있다: 스플라인들의 계산은 아주 잘 연구 개발된 분야이다. 다른 형태의 스플라인들은 그 스플라인들이 각 데이터 점을 정확히 지나가야만 한다는 조건을 완화시키는 변경 뿐만 아니라 다른 형태의 매끄러움에 대한 기준들 또한 가진다. 스플라인들은, 계산적으로는 그들이 연결되어야 하는 방법에 대한 제약 조건(constraint)들로부터 유도된 선형 방정식들의 시스템을 해결함으로써, 각각의 스플라인 조각들에 대한 계수들을 결정하는데 있어 동일한 스텝들을 거친다.

최소 제곱 방법(Method of Least Squares)

데이터 값들은 정확하지 않은 반면에, 그 데이터 값들의 맞춤을 수행해야 하는 함수 형태에 대한 개념이 있어야 하는 경우가 종종 발생한다. 이 함수는 어쩌면, 파라미터들

$$f(x) = f(c_1, c_2, \dots, c_M, x)$$

에 의해 좌우될 수 있다. 이 경우 곡선 맞춤 프로시저는 주어진 점들에서 관측된 값들과 “최고로”매치 하는 파라미터들을 선택하는 것이다. 만약 함수가 한 다항식(그 파라미터들이 계수인)이고 그 값들이 정확하다면, 이는 보간법일 수도 있다. 그러나 지금 우리는 보다 일반적인 함수들과 정확하지 않은 데이터의 경우를 고려 중에 있다. 간략히 설명하기 위해, 보다 단순한 함수들의 선형 조합으로서 설명되는 미지의 파라미터들을 계수들로써 갖는, 아래와 같은 함수들에 대한 맞춤 문제만 집중적으로 다루기로 한다:

$$f(x) = c_1 f_1(x) + c_2 f_2(x) + \dots + c_M f_M(x)$$

이 공식은 우리가 관심을 갖는 대부분의 함수들을 포함한다. 이 경우에 대해 연구한 후, 보다 일반적인 함수들에 대해 생각해 보기로 한다.

함수가 얼마나 잘 맞춤을 하는가를 측정하는 한 가지 공통적인 방법은 최소 제곱 기준

(least-square's criterion)에 의한 방법이다. 여기서, 오차(error)는 각 관측 점들에서의 오차들의 제곱을 더함으로써 계산한다:

$$E = \sum_{1 \leq j \leq N} (f(x_j) - y_j)^2.$$

이 방법은 매우 자연스런 추정법이다: 제곱이 행해진 것은 부호(sign)가 서로 다른 오차 값들이 상쇄되어짐을 피하기 위한 것이다. 명백히, E 를 최소화해 주는 파라미터들을 찾아서 선택함이 가장 바람직하다. 이러한 선택 방식이 효율적임이 확인되었다: 이것이 소위 말하는 최소 제곱 방법이다.

이 방법은 정의로부터 정확히 얻어진다. 유도를 단순화하기 위해, $M = 2$, $N = 3$ 인 경우를 생각해보기로 한다. 그러나, 일반적 방법은 그대로 따른다. 세 개의 점 x_1, x_2, x_3 과 이 각각에 해당하는 $f(x) = c_1 f_1(x) + c_2 f_2(x)$ 의 형태를 갖는 함수로 맞추어야 할 값인 y_1, y_2, y_3 가 있다 하자. 여기서 할 일은 최소 제곱 오차를 최소화해주는 계수 c_1, c_2 를 찾아서 선택하는 것이다.

$$\begin{aligned} E = & (c_1 f_1(x_1) + c_2 f_2(x_1) - y_1)^2 \\ & + (c_1 f_1(x_2) + c_2 f_2(x_2) - y_2)^2 \\ & + (c_1 f_1(x_3) + c_2 f_2(x_3) - y_3)^2 \end{aligned}$$

오차를 최소화하는 c_1 과 c_2 를 찾기 위해, 미분 계수 dE/dc_1 과 dE/dc_2 를 영(0)으로 설정(set)할 필요가 있다. c_1 의 경우는 다음과 같다:

$$\begin{aligned} dE/dc_1 = & 2(c_1 f_1(x_1) + c_2 f_2(x_1) - y_1)f_1(x_1) \\ & + 2(c_1 f_1(x_2) + c_2 f_2(x_2) - y_2)f_1(x_2) \\ & + 2(c_1 f_1(x_3) + c_2 f_2(x_3) - y_3)f_1(x_3) \end{aligned}$$

미분계수를 영(0)으로 만듦으로써 변수 c_1 과 c_2 가 만족시켜야 하는 한 방정식이 남게 된다. ($f_1(x_1)$ 등은 모두 알려진 값들을 가진 “상수”들이다) :

$$\begin{aligned} c_1(f_1(x_1)f_1(x_1) + f_1(x_2)f_1(x_2) + f_1(x_3)f_1(x_3)) + c_2(f_2(x_1)f_1(x_1) + f_2(x_2)f_1(x_2) + f_2(x_3)f_1(x_3)) = \\ y_1 f_1(x_1) + y_2 f_1(x_2) + y_3 f_1(x_3) \end{aligned}$$

미분계수 dE/dc_2 를 영(0)으로 설정할 때도, 이와 유사한 방정식이 얻어진다. 이러한 보기에 복잡한 방정식들은 벡터(vector) 표현과 “내적(dot product)” 연산을 사용하면 아주 단순화될 수 있다. $x = (x_1, x_2, x_3)$ 이고 $y = (y_1, y_2, y_3)$ 로서 정의하면, x 와 y 의 내적은 아래와 같이 정의된 실수이다.

$$\mathbf{x} \cdot \mathbf{y} = x_1y_1 + x_2y_2 + x_3y_3.$$

이제, 벡터를 $f_1 = (f_1(x_1), f_1(x_2), f_1(x_3))$ 이고 $f_2 = (f_2(x_1), f_2(x_2), f_2(x_3))$ 로 정의하면, 계수 c_1 과 c_2 에 대한 방정식들은 아래와 같이 아주 간단히 표현될 수 있다:

$$\begin{aligned} c_1 \mathbf{f}_1 \cdot \mathbf{f}_1 + c_2 \mathbf{f}_1 \cdot \mathbf{f}_2 &= \mathbf{y} \cdot \mathbf{f}_1, \\ c_1 \mathbf{f}_2 \cdot \mathbf{f}_1 + c_2 \mathbf{f}_2 \cdot \mathbf{f}_2 &= \mathbf{y} \cdot \mathbf{f}_2. \end{aligned}$$

가우스 소거법을 사용하여 원하는 계수값을 찾음으로서 이들은 해결된다. 예로서, 다음의 데이터 점들이 $c_1 + c_2/x$ 의 형태의 함수에 의해 맞추되어야 한다고 하자.(이 데이터 점들은 $1+1/x$ 에 대한 정확한 값들로부터 약간 변경된 값들이다):

$$(1.0, 2.05) (2.0, 1.53) (4.0, 1.26) (5.0, 1.21) (8.0, 1.13) (10.0, 1.1)$$

이 경우, f_1 은 상수 $f_1 = (1.0, 1.0, 1.0, 1.0, 1.0, 1.0)$ 이고 $f_2 = (1.0, 0.5, 0.25, 0.2, 0.125, 0.1)$ 이다. 그래서 다음과 같은 방정식 시스템을 풀어야 한다.

$$\begin{pmatrix} 6.000 & 2.175 \\ 2.175 & 1.378 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 8.280 \\ 3.623 \end{pmatrix}$$

결과로서 얻어진 $c_1 = 0.998$ 이고 $c_2 = 1.054$ 이다.(예상했던 것처럼 둘 다 1에 가깝다) 위에 개략적으로 설명된 방법은 계수가 둘보다 많은 경우로도 쉽게 일반화된다. 점에 대한 최소 제곱 오차를 최소화 해주는 아래 함수

$$f(x) = c_1 f_1(x) + c_2 f_2(x) + \cdots + c_M f_M(x)$$

와 관측 벡터들

$$\begin{aligned} \mathbf{x} &= (x_1, x_2, \cdots, x_N), \\ \mathbf{y} &= (y_1, y_2, \cdots, y_N). \end{aligned}$$

에서 상수 c_1, c_2, \dots, c_M 을 찾기 위해서는, 먼저 함수 원소(component) 벡터들을 다음과 같이 계산하고

$$\begin{aligned} \mathbf{f}_1 &= (f_1(x_1), f_1(x_2), \dots, f_1(x_N)), \\ \mathbf{f}_2 &= (f_2(x_1), f_2(x_2), \dots, f_2(x_N)), \\ &\vdots \\ \mathbf{f}_M &= (f_M(x_1), f_M(x_2), \dots, f_M(x_N)), \end{aligned}$$

그리고 나서,

$$\begin{aligned} a_{ij} &= \mathbf{f}_i \cdot \mathbf{f}_j, \\ b_j &= \mathbf{f}_j \cdot \mathbf{y}. \end{aligned}$$

를 갖는 M 행- M 열의 방정식들 $A\mathbf{c} = \mathbf{b}$ 의 M 행- M 열 선형 방정식 시스템 $A\mathbf{c} = \mathbf{b}$ 를 만든다. 이 연립 방정식들에 대한 해는 필요한 계수들을 산출한다.

이 방법은 \mathbf{y} 를 $(M + 1)$ 번째 벡터로서 간주하고, \mathbf{f} 벡터들에 대한 이차원 배열을 만듦으로써 쉽게 구현될 수 있다. 이와 같은 배열은, 위 설명에 따라서, 다음과 같은 코드를 작성함으로써 만들어진다:

```
for (i = 1; i <= M; i++)
    for (j = 1; j <= M+1; j++)
    {
        t = 0.0;
        for (k = 1; k <= N; k++)
            t += f[i][k] * f[j][k];
        a[i][j] = t;
    }
```

그리고 해당 연립 방정식 시스템은 37장에 있는 가우스 소거법을 이용하면 해결된다.

최소 제곱 방법은 비선형 함수들(예로서 $f(x) = c_1 e^{-c_2 x} \sin c_3 x$)을 처리할 수 있도록 확장될 수 있다. 그리고 이 확장된 방법은 종종 그런 종류의 응용 분야에서 사용된다. 기본 개념은 동일하다: 문제는 미분 계수들의 계산이 용이하지 않다는 점이다. 사용되는 방법은 반복법(iterative method)이다: 계수들에 대한 어떤 추정치(estimate)를 구한 다음, 이들을 최소 제곱 방법에서 사용하여 미분 계수를 구한다. 이같이 하여 계수들에 대한 좀 더 나은 추정치를 얻을 수 있다. 오늘날 아주 널리 사용되고 있는 이 기본적인 방법은 1820년대에 Gauss에 의해 개략적으로 설명되었다.

연습문제

1. 점 1, 2, 3, 4, 5에서 사차 보간 다항식을 가지고 함수 $\lg x$ 의 근사치를 구하라. 점 1.5, 2.5, 3.5, 4.5에서 오차들의 제곱의 합을 계산함으로써 그 맞춤의 적합도를 평가하라.
2. 함수 $\sin x$ 에 대해 위 문제를 반복하라. 함수와 그 근사치를 컴퓨터 상에서 플롯(plot)하라.
3. 다항식 보간법 대신 삼차 스플라인을 이용하여 앞의 문제들을 해결하라.
4. 1과 10사이에 있는 N 에 대해 2^N 개의 노트들을 갖는 삼차 스플라인을 가지고 함수 $\lg x$ 의 근사치를 구하라. 동일 구간에서 노트들의 위치를 바꿔가면서 반복 시행하여 좀 더 잘 맞는 값들을 구하라.
5. 만약 함수들 중의 하나가 임의의 i 에 대해 $f_i(x) = 0$ 이었다면, 최소 제곱 데이터 맞춤에서 어떤 일이 일어날까?
6. 최소 제곱 곡선 맞춤을 사용하여, 퀵정렬(quicksort)이 한 무작위 파일 상에서 수행될 때 실행되는 명령어들의 총수를 설명하기 위해, $aN \ln N + bN$ 형태의 최고의 공식을 갖게 해주는 a 와 b 값을 찾아라.
7. 관측치인 $f(1) = 0, f(4) = 13, f(8) = 41$ 에 근접하기 위해 함수 $f(x) = ax \log x + bx + c$ 를 사용할 때, 최소 제곱 오차를 최소화 해주는 a, b, c 값을 구하라.
8. 가우스 소거법의 소거 단계를 제외하면, N 개의 관측치에 근거하여 M 개의 계수를 찾기 위해 최소 제곱 방법을 이용했을 때 발생하는 곱셈의 횟수는 얼마인가?
9. 어떤 상황에서 최소 제곱 곡선 맞춤법에서 생기는 행렬이 역행렬이 없는 singlar(singular) 행렬이 될까?
10. 만약 두 개의 다른 관측치가 동일점에 대해 있다면 최소 제곱 방식이 동작할 수 있는가?

적분 계산은 컴퓨터 상에서 처리되는 함수들에 대해 종종 수행되는 기본적이고 분석적인 연산이다. 우리는 “곡선아래의 영역”을, 효율적으로 그리고 적당한 정확도를 가지고, 찾기를 원한다. 이 장에서는 이 기초적인 수치 문제의 해결을 위한 많은 전통적인 알고리즘들을 살펴보기로 한다.

먼저, 함수의 명백한 표현이 가능할 때의 상태를 간단히 언급할 것이다. 이같은 경우에, 한 함수의 표현을 적분을 위한 유사한 표현으로 변형하는 심볼릭 적분(symbolic integration)을 하는 것이 가능하다. 이것은 처리중인 함수가 적분이 분석적으로 가능한 제약된 계층의 함수들이거나 그와 같이 표현된 함수들을 처리하는 시스템들이 있는 경우에 적합하다.

또 하나의 극단적인 경우, 함수는 도표에 의해 정의 될 수 있다. 그래서, 단지 몇 개 점에서의 함수값만을 알 수 있다. 이같은 경우에는, 함수가 어떻게 점들간에서 동작하는지에 근거하여, 적분에 대한 근사값만을 구할 수 있다. 적분의 정확도는 거의 전적으로 그에 대한 가정들이 얼마나 정확한지에 달려있다.

가장 일반적인 상황이 이러한 양극단 사이에 있다. 적분될 함수는 특정 점에서의 값이 계산될 수 있는 방법으로 나타내진다. 적분의 정확도는 계산을 위해 선택된 점들 사이에서 함수가 어떻게 동작하는지에 대한 가정들에 의해 좌우된다. 목적은 함수의 적분에 대한 근사값을 지나친 함수 계산 없이 구하는 것이다. 이 계산은 종종 수치 해석자들에 의해 구상법(quadrature)이라 불려진다.

이장에서는, 몇 개의 기초적인 구상법을 살펴본다. - 우리의 목표는 기본적인 수치적 방법 같은 계산법들에 대한 경험을 쌓는 것이다. 많은 응용 프로그램들은 실제로 우리가 처음 공

부하는 기본적인 방법들을 적절히 응용함으로써 혜택을 받을 수 있다. 그러나 보다 진보된 형태의 문제들을 풀기 위한, 특히 미분방정식들의 수치적 해를 얻기 위한, 방법들은 실제적으로 아주 중요하다.

심볼릭 적분(Symbolic Integration)

만약 함수에 대한 충분한 정보가 있다면, 수치들을 가지고 일하기보다는 함수의 표현식을 조작하는 방법을 사용해 볼만한 가치가 있다. 목적은 한 함수의 표현식을 부정적분(indefinite integral)을 손으로 해결하는 것과 거의 같은 방법으로 적분의 표현식으로 변형하기 위한 것이다.

이의 간단한 한 예는 다항식들의 적분이다. 36장에서, 다항식에 대한 한 표현식에서 동작하는 프로그램을 이용하고 입력의 표현식으로부터 답에 대한 표현식을 산출하는, 다항식의 합과 곱을 “심볼릭”으로 계산하기 위한, 방법들을 살펴보았다. 다항식의 적분 그리고 미분 연산 또한 이같은 방법으로 행해질 수 있다. 만약 다항식

$$p(x) = p_0 + p_1x + p_2x^2 + \cdots + p_{N-1}x^{N-1}$$

이 배열 p 에 계수들의 값을 유지함으로써 간단히 표현될 수 있다면, 적분은 다음과 같이 쉽게 계산된다:

$$\text{for } (i = N; i > 0; i--) \text{ p}[i] = \text{p}[i-1]/i; \text{ p}[0] = 0;$$

이 프로그램은 다항식의 각 항에 대해, $i > 0$ 인 경우에 잘 알려진 심볼릭 적분 법칙인 $\int_0^x t^{i-1} dt = x^i/i$ 를 적용한다. 다항식보다 더 광범위한 계층의 함수들은 심볼릭한 법칙들을 더 많이 추가함으로써 처리될 수 있다. 다음의 부분 적분(integration by parts)

$$\int u \, dv = uv - \int v \, du,$$

같은 복합적인 법칙들을 추가함으로써, 처리할 수 있는 함수들의 집합을 크게 확장할 수 있다. 부분적분은 미분능력을 필요로 한다. 심볼릭 미분(symbolic differentiation)은 심볼릭 적분보다는 다소 쉬울 것이다. 왜냐하면 기본 법칙들에 대한 한 타당한 집합과 복합적 연쇄법칙(chain rule)만 갖고도 대부분의 일반 함수들을 충족시키기에 충분하기 때문이다.

특정 함수에 적용될 수 있는 많은 법칙들은 심볼릭 적분을 어렵게 만든다. 실제로, 어떤 함수의 적분값을 리턴하거나 답이 기본적인 함수로 표현될 수 없음을 알려주는 알고리즘이 이러한 작업에 대해 있다는 것이 최근에 알려졌다. 이러한 알고리즘을 완전한 일반성을 갖도록 설명하는 것은 이 책의 범위를 벗어난다. 그러나, 처리될 함수가 아주 제한된 계층에 있다면 심볼릭 적분은 강력한 도구가 될 수 있다.

물론, 심볼릭 기법들은 “아주 많은 적분들의 경우 심볼릭하게 계산될 수 없다”는 근본적인 한계를 가진다. 다음에는, 실제 적분값들에 대한 근사치 계산을 위해 개발된 몇 가지 기법들을 살펴볼 것이다.

간단한 구상법(Simple Quadrature Methods)

아마도 한 적분 근사치를 구하는 가장 명백한 방법은 사각형법(rectangle method)일 것이다. 적분 계산은 한 곡선 아래의 영역을 계산하는 것과 같고, 그림 39.1의 다이어그램에서와 같이, 그 곡선 아래에 거의 꼭 맞는 작은 사각형 영역을 더함으로써 곡선 아래의 영역을 측정해 볼 수 있다.

보다 정확히 하기 위해, 우리가 $\int_a^b f(x)dx$ 를 계산한다고 하자. 그리고 이 적분의 수행구간 $[a, b]$ 가 점 x_1, x_2, \dots, x_{N+1} 에 의해 구간 경계가지어진, N 개의 부분으로 나뉘어져 있다고 하자. 그러면 i 번째 ($1 \leq i \leq N$) 사각형의 폭이 $x_{i+1} - x_i$ 인 N 개의 사각형을 갖게 된다. i 번째 사각형의 높이는 $f(x_i)$ 나 $f(x_{i+1})$ 를 이용하여 구할 수 있지만, 위의 다이어그램에서와 같이, 구간의 중간점 $f((x_i + x_{i+1})/2)$ 에서의 f 의 값을 사용하면 더 정확할 수 있다. 이렇게하여, 다음과 같은 구상법 공식이 만들어 지게 된다:

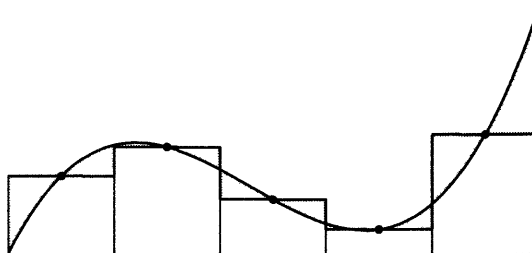


그림 39.1 사각형법

$$r = \sum_{1 \leq i \leq N} (x_{i+1} - x_i) f\left(\frac{x_i + x_{i+1}}{2}\right).$$

이 공식은 $a = x_1$ 와 $b = x_{N+1}$ 의 구간에서 $f(x)$ 의 적분 값을 측정한다. 모든 구간이 동일한 크기를 갖는(보통의) 경우, 즉 $x_{i+1} - x_i = w$ 인 경우에는, $x_{i+1} + x_i = (2i + 1)w$ 이다. 그러므로 적분에 대한 근사치 r 은 쉽게 계산된다.

```
double intrect(double a, double b, int N)
{
    int i; double r = 0; double w = (b-a)/N;
    for (i = 1; i <= N; i++) r+= w*f(a-w/2+i*w)
    return r;
}
```

물론, N 이 커짐에 따라 답은 더 정확해 진다. 그림 39.2는 그림 39.1에 있는 함수에 대해 더 작은 구간 크기를 사용했을 경우의 결과를 보여준다.

아래의 예는 $\int_1^2 dx/x$ (우리가 $\ln 2 = 0.6931471805599$ 로 아는)에 대해, $N = 10, 100, 1000$ 일 경우, 이 함수가 `intrect(1.0, 2.0, N)`을 호출함으로써 산출되는 양적인 측정치들의 예이다.

10	0.6928353604100
100	0.6931440556283
1000	0.6931471493100

N 값이 1000일 때 십진자리수로 대략 7까지는 정확한 답이 얻어진다. 보다 정교한 구상법은 일의 양을 줄이면서도 더 정확한 결과를 얻을 수 있게 해준다.

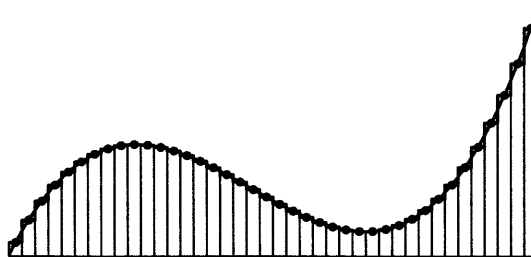


그림 39.2 보다 작은 구간 크기를 가질 때의 사각형법

특정 방법들에 대해 측정치 오차들이 있다는 것은 더 정확한 방법이 있을 수 있음을 암시한다. 각 구간의 중앙에서 Tayler 급수(series)로 $f(x)$ 를 전개하고, 적분하고, 그리고 모든 구간에 대한 결과치를 더함으로써, 사각형법에서 만들어진 오차에 대해 분석적으로 표현해 보자. 이 계산의 자세한 것들까지는 다루지 않겠다. 그러나 단지 다음 식만큼은 언급하겠다.

$$\int_a^b f(x)dx = r + w^3 e_3 + w^5 e_5 + \dots$$

여기서 w 는 구간 폭 $((b - a)/N)$ 이고 e_3 는 구간의 중앙점에서의 f 의 삼차 미분계수 값이다. 등등..

이것은 통상적으로 아주 좋은 근사치 방법이다. 왜냐하면 대개의 “타당한” 함수들은 고차의 미분 계수들을 조금 가지기 때문이다.(비록 항상 그렇지는 않더라도) 예로서, 만약 $w = 0.01$ 이 되게 하면(이 값은 위 예제에서 $N = 200$ 에 해당하는다), 이 공식은 위 프로시저에 의해 계산된 적분값이 약 6자리수까지는 정확하다는 것을 말해준다.

적분 근사치를 구하는 또 하나의 방법은 곡선 아래의 영역을 그림 39.3과 같이 사다리꼴(trapezoid)로 나누는 것이다. 사다리꼴 영역은 밑변과 윗변의 합에 높이를 곱한 값의 절반 $(1/2)$ 임을 상기하라. 사다리꼴법에서 부터 나온 구상법은 다음과 같다:

$$t = \sum_{1 \leq i \leq N} (x_{i+1} - x_i) \frac{f(x_i) + f(x_{i+1})}{2}.$$

다음은 모든 구간들이 같은 폭을 갖는 일반적인 경우에 사다리꼴법을 구현하는 프로시저이다.

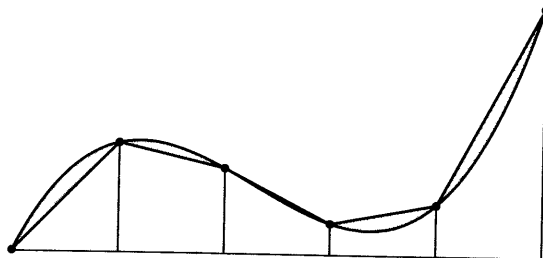


그림 39.3 사다리꼴법

```
double inttrap(double a, double b, int N)
{
    int i; double t=0; double w=(b-a)/N;
    for (i=1; i<=N; i++)
        t += w*(f(a+(i-1)*w)+f(a+i*w))/2;
    return t;
}
```

이 방법에서의 오차는 사각형법의 경우와 유사한 방법으로 유도될 수 있다. 구해진 공식은 다음과 같다.

$$\int_a^b f(x)dx = t-2w^3e_3 - 4w^4e_5 + \dots$$

이같이, 사각형법은 사다리꼴법보다 두배 정확하다. 이것은 예를 통해 검증된다. 이 프로 시저는 $\int_1^2 dx/x$ 에 대해 다음과 같은 측정치를 산출한다.

10	0.6937714031754
100	0.6931534304818
1000	0.6931472430599

사각형법이 사다리꼴법보다 더 정확하다는 것은 처음에는 놀라울 수도 있다. 그러나, 사다리꼴은 완전히 곡선아래에 있거나 완전히 곡선 위에 있게 되는 경향이 있는 반면, 사각형은 부분적으로 그렇게 되는 경향이 있음을 기억하라. 그림 39.4는 보다 작은 구간 크기를 가질 때의 사다리꼴법을 보여준다. 이것은 곡선과 정확히 맞는 듯하다. 그러나, 실제로는 그림 39.2가 곡선 아래 영역에 대해 더 좋은 측정치를 구해준다.

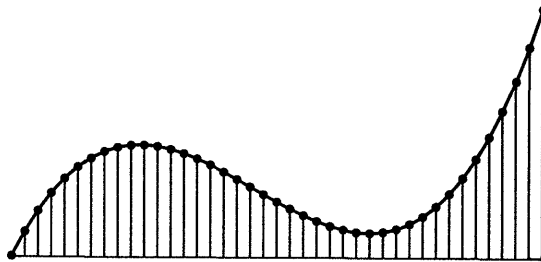


그림 39.4 보다 작은 구간 크기를 가질 때의 사다리꼴법

또 하나의 완전히 “타당한” 방법은 스플라인 구상법이다. 스플라인 보간법이 전장에서 설명한 방법을 사용하여 수행되고 그리고 나서 위에서 설명한 간단한 심볼릭 다항식 적분기법을 구획단위로 적용하여 적분을 계산한다. 이 방법은 아래에서 알게되겠지만 실제로 사각형법과 사다리꼴법에 아주 밀접하게 연관되어 있다.

합성법(Compound Methods)

사각형법과 사다리꼴법의 오차에 대한 위의 공식들을 살펴봄으로써, Simpson방법 (Simpson's method)이라 불리는 보다 더 정확한 방법이 찾아진다. 이 방법은 두 방법을 결합하여 오차의 주된 항을 제거하는 것이다. 사각형법 공식에 2를 곱하고, 사다리꼴법에 대한 공식을 더한 뒤, 3으로 나누면 다음과 같은 공식이 만들어진다.

$$\int_a^b f(x)dx = \frac{1}{3} (2r + t - 2w^5e_5 + \dots)$$

여기서 w^3 항이 제거되어 족음에 주목하라. 그러므로 이 공식은, 우리가 구상법 공식들을 동일한 방법으로 결합함으로써, w^5 범위 내에서 정확성을 유지하는 방법을 얻을 수 있음을 알려준다:

$$s = \sum_{1 \leq i \leq N} \frac{x_{i+1} - x_i}{6} \left(f(x_i) + 4f \frac{x_i + x_{i+1}}{2} + f(x_{i+1}) \right)$$

만약 Simpson의 법칙에 구간크기 0.01을 사용하면, 적분은 약 10자리 수의 정확도로 계산될 수 있다. 이것 또한 예제를 통해 검증될 수 있다. Simpson 방법의 구현은 다른 방법에 비해 약간 더 복잡하다.(여기서도 구간들이 동일한 폭을 가지는 경우를 생각해 본다):

```
double intsimp(double a, double b, int N)
{
    int i; double s = 0; double w = (b-a)/N;
    for (i = 1; i <= N; i++)
        s += w*(f(a+(i-1)*w) +
                4*f(a-w/2+i*w) +
                f(a+i*w))/6;
    return s;
}
```

이 프로그램은 내부 루프에서 세 개의 “함수 계산”을 필요하다. 그러나, 앞의 두 방법보다 훨씬 더 정확한 값을 산출한다.

10	0.6931473746651
100	0.6931471805795
1000	0.6931471805599

유사한 오차들을 갖는 간단한 방법들을 조합함으로써 보다 정확한 값을 얻는 더 복잡한 구상법이 고안되어 왔다. 가장 잘 알려진 방법은 Romberg적분법인데, 이 방법은 두 “방법들”에 대한 두 개의 다른 부구간(subinterval)들의 집합을 이용한다.

Simpson방법은 데이터를 구획별 이차 함수로 보간한 후 적분하는 것과 같다. 언급한 네 개의 방법 모두가 보간법의 형태로 변경될 수 있음은 주목할만 하다: 사각형법은 상수로 보간된다.(영(0)차 다항식); 사다리꼴법은 선으로(일차 다항식); Simpson의 법칙은 이차 다항식; 그리고 스플라인 구상법은 삼차 다항식으로 보간된다.

적응력있는 구상법(Adaptive Quadrature)

이제까지 다뤄 왔던 방법들의 주된 결점은 오차가 사용된 부구간(subinterval) 크기 뿐만 아니라 적분되는 함수의 고차수 미분계수들의 값에 좌우된다는 것이다. 이것은 이러한 방법들이 특정 함수들에서는 전혀 동작하지 않을 수 있다는 것을 의미한다. 하지만, 큰 고차수 미분계수들을 갖는 함수는 거의 없다. 미분계수들이 큰 경우에는 작은 구간들을, 작은 경우에는 큰 구간들을 사용함이 합리적이다. 이를 체계적인 방법으로 적용하는 방법을 적응력있는 구상 루틴(adaptive quadrature routine)이라고 한다.

적응력있는 구상법에서의 일반적인 방법은 각 부구간에 대해 두 개의 다른 구상법을 이용하고 결과를 비교하여, 그 차이가 너무 크면 구간을 또 나누어 세분화하는 것이다. 물론, 모두 좋지 않은 두 개의 방법이 사용된다면 결과가 좋지 못한 값에 가까워지므로 주의를 해야만 한다. 이를 피하기 위한 한 방법은 결과를 항상 과대평가하고 또 한 방법은 결과를 항상 과소평가 하는 것이다. 또 다른 방법은 한 방법이 또 다른 하나보다 더 정확하도록 만드는 것이다. 후자의 방법은 다음에 설명한다.

구간을 재귀적으로 세분화할 때 심각한 오버헤드가 수반된다. 그래서, 구간들을 추정하기 위해 다음과 같은 좋은 구현방법을 사용할 필요가 있다.

```
double adapt(double a, double b)
{
    double x=intsimp(a, b, 10);
    if (fabs(x-intsimp(a, b, 5))>tolerance)
        return adapt(a, (a+b)/2)+adapt((a+b)/2, b);
    return x;
}
```

적분을 위한 두 근사치가 Simpson방법에서 유도됐다. 그러나 어떤 사람은 다른 사람보다 두배 더 많은 세부분할을 사용한다. 본질적으로는 이것은 조사 대상구간에 대해 Simpson방법의 정확도를 조사하고, 만약 좋지 않으면, 더 세부분할을 하는 것에 해당한다.

얼마나 많은 일을 하기를 원하는가 그리고 그 결과로서 어떤 정확도가 나오든 간에 그것을 수궁해야 하는 이제까지 논의해 왔던 다른 방법들과는 다르게, 적응력있는 구상법에서는 미리 정해진 정확도를 얻기 위해 해야할 일이 얼마나 많은 간에 그 일을 한다. 이것은 허용 오차(tolerance)가 아주 조심스럽게 선택되어야함을 의미한다. 그래서 이 루틴은 불가능한 높은 허용오차를 얻기 위해 무한 루프를 반복하면 안된다. 필요한 스텝 수는 적분될 함수에 따라 아주 많이 좌우된다. 아주 변위가 심한 함수는 아주 많은 스텝들을 필요로 한다. 그러나 그런 함수는 “고정 구간”방법들을 이용하면 아주 부정확한 결과를 산출한다. 그림 39.5는 그림 39.1에서부터 그림 39.4까지의 함수에 사다리꼴법에 근거한 적응력있는 구상법이 사용됐을 때 계산된 점을 보여준다. 함수가 바르고 매끄러운 수록 구간들이 더 커지고, 함수가 보다 급히 기울수록 구간들이 더 작아짐에 주목하라.

우리의 예제 같은 매끄러운 함수는 합당한 스텝 수로서 처리될 수 있다. 다음 표는 다양한 t 값에 대해 생성되는 값과 $\int_1^2 dx/x$ 를 계산하기 위해 위의 루틴에 의해 요구되는 재귀호출의 수를 보여준다.

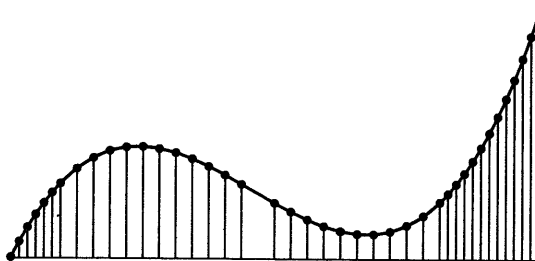


그림 39.5 적응력있는 구상법

0.00001000000	0.6931473746651	1
0.00000010000	0.6931471829695	5
0.00000000100	0.6931471806413	13
0.00000000001	0.6931471805623	33

위 프로그램은 여러 가지 방법으로 개선될 수 있다. 먼저, `intsimp(a, b, 10)`을 호출하기 위한 함수 값들은 `intsimp(a, b, 5)`와 공유될 수 있다. 다음으로, 만약 허용오차가 현재 구간의 크기와 전 구간의 크기의 비에 의해 축적(scale)된다면, 허용오차 한계와 답의 정확도와의 관계는 보다 가깝게 연관될 수 있다. 또한, Simpson방법(하지만 Simpson방법은 재귀적 법칙의 기본이다)보다 더 나은 구상법을 사용함으로써 보다 나은 루틴이 개발될 수 있다. 복잡한 적응력있는 구상법 루틴은 어느 다른 방법에 의해 처리될 수 없는 문제들에 대해 아주 정확한 결과를 제공할 수도 있다. 그러나 처리될 함수들의 유형에 주의를 기울이는 것이 필요하다.

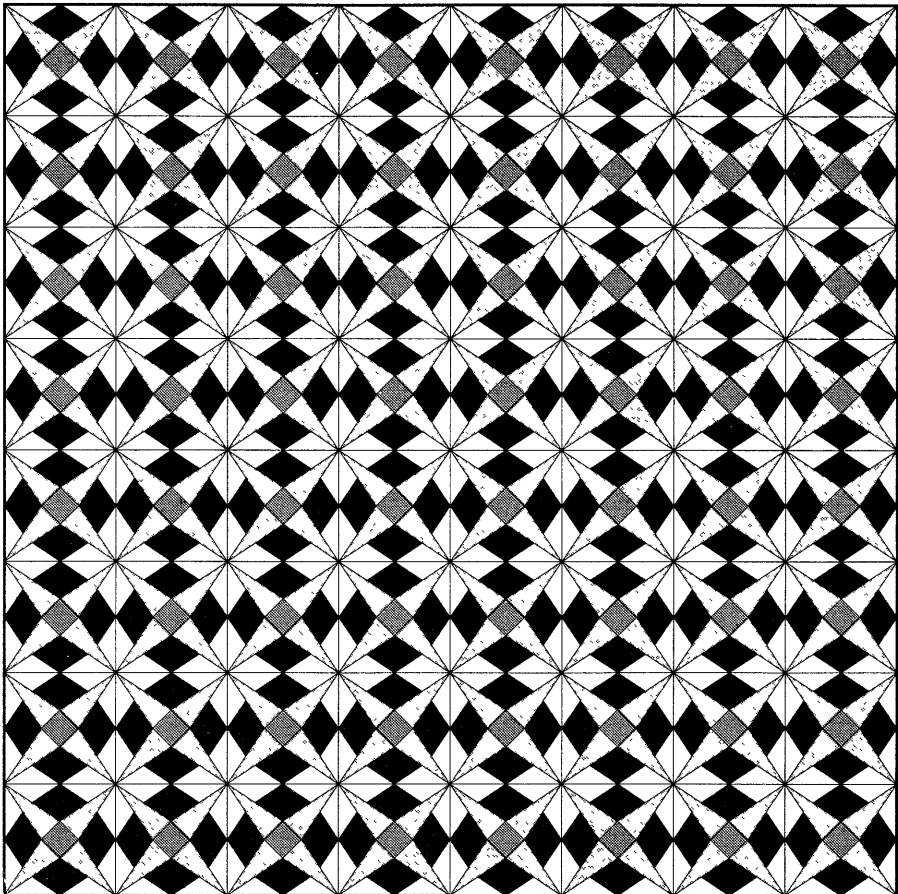
이같이 분할-정복 알고리즘 설계 패러다임은 수치적 문제에 유용하다. 사실, 이런 형태의 적응력있는 방법들은, 고차원에서의 적분과 미분 방정식들의 수치적 해법 같은, 고수준의 수치적 문제들에 대한 해결 기법으로서 아주 중요하다.

연습문제

1. x 와 $\ln x$ 에서 다항식을 심볼릭 적분(그리고 심볼릭 미분)하는 프로그램을 작성하라. 부분 적분에 기초하여 재귀적인 구현을 하라.
2. 어떤 구상법이 다음 함수들 $f(x) = 5x$, $f(x) = (3 - x)(4 + x)$, $f(x) = \sin(x)$ 에 대한 적분값을 가장 정확하게 산출할 수 있겠는가.
3. 네 개의 기본적인 구상법(사각형, 사다리꼴, Simpson, 스플라인)을 사용하여, 구간 $[1, 10]$ 에서 $y = 1/x$ 에 대한 각각의 적분값을 보여라.
4. 위 문제에서 함수가 $y = \sin x$ 일 때의 답을 구하라.
5. 적응력있는 구상법이 구간 $[-1, 2]$ 에서 함수 $y = 1/x$ 를 적분하는데 이용될 경우, 어떤 일이 일어날 지에 대해 논하라.
6. 위 문제를 기본적인 적응력 있는 구상법들일때의 경우에 대해 답하라.
7. 구간이 $[1, 10]$ 이고 허용 한계가 .1일 때, 함수 $y = 1/x$ 에서 적분값을 계산하는데 적응력 있는 구상법을 사용할 때의 계산점들을 구하라.
8. 위 문제에서 주어진 적분에 Simpson 방법에 기초한 적응력있는 구상법을 사용할 때와 사각형법에 기초한 적응력있는 구상법을 사용할 때의 정확도를 비교하라.
9. 위 문제를 함수가 $y = \sin x$ 때의 경우에서 답하라.
10. 적응력있는 구상법이 다른 방법보다 아주 대단히 정확한 결과를 보여 줄 수 있는 함수의 한 특정 예를 제시하라.

빈 면

고수준 토픽들



빈 면

우리가 공부하는 대부분의 병렬(parallel) 알고리즘들은 응용도가 아주 높다. 우리가 보아 온 대부분의 방법들은 10년 또는 그 이상 된 것들이고 많은 컴퓨터 하드웨어와 소프트웨어의 급격한 변화에도 잘 적응해 왔다. 새 하드웨어 제품들과 새 소프트웨어 기능들은 특정 알고리즘들에 대해 심각한 영향을 줄 수 있다. 그러나 옛 기계에서 좋은 알고리즘은 대개 새 기계에서도 좋은 알고리즘이다.

이의 한 가지 이유는 기존 컴퓨터의 기본적인 설계들이 상당 기간동안 거의 바뀌지 않았다는 것이다. 거의 모든 컴퓨터 시스템의 설계는 근대 컴퓨터 역사의 초기에 수학자 J. von Neumann이 만들어 낸 하나의 동일한 기본 원리를 따른다. von Neumann의 계산 모델(model of computation)에서는, 명령어들과 데이터가 한 메모리에 저장되고 한 개의 프로세서가 이 메모리에서 명령어들을 끄집어내어(물론 이 명령어들은 데이터를 가지고 동작한다) 한 번에 하나씩 실행시킨다. 컴퓨터들이 더 싸고, 더 빠르고, 더 작고, 더 좋은 성능을 갖도록 만들기 위해 여러 정교한 메커니즘들이 개발되어 왔다. 그러나 대개의 컴퓨터 시스템들은 von Neumann 구조의 변형될 형태들이다.

그러나 최근 컴퓨터 부품 가격의 획기적인 하락은 아주 많은 수의 명령어들이 동시에 실행될 수 있거나 여러개의 특수용 기계(컴퓨터)들이 한 문제를 같이 해결하도록 명령어들이 서로 “연결되어(wired-in)” 있거나 또는 많은 아주 작은 컴퓨터들이 동일한 문제를 풀기위해 서로 협력할 수 있는, 아주 다른 형태의 컴퓨터들을 만들려는 생각을 하게 했다. 요약하면, 한 번에 한 명령어를 실행하는 기계보다는 동시에 많은 연산을 할 수 있는 기계를 만드는 것을 생각해 볼 수 있다. 이 장에서는, 이제까지 연구되고 있는 문제들과 알고리즘들에 대해

이같은 생각들이 줄 수 있는 잠재적 효과를 고려해 보고자 한다. 특히, 병렬 알고리즘(parallel algorithm)들의 개발에 이용하기 좋은 기계 구조에 대한 두 가지 접근 방식인, 완전-셔플(perfect shuffle) 방식과 시스톨릭 배열(systolic array) 방식을 다룰 것이다.

일반적 방식(General Approaches)

특정한 기본 알고리즘들은 항상 더 크고 더 빠른 컴퓨터들 상에서 수행돼야만 좋은 아주 큰 문제들을 풀기 위해 빈번히 사용된다. 이의 한 결과로서 최근의 기술을 이용한 슈퍼 컴퓨터들이 계속 만들어 졌다. 이 슈퍼 컴퓨터들은 기본적인 von Neumann 개념을 버렸으나 범용성 있게 설계되어 모든 프로그램들에게 좋도록 만들어 졌다. 이제까지 우리가 공부해 온 유형의 문제들을 이러한 기계에서 사용하는 일반적인 방식은 기존 기계들에서 가장 좋았던 알고리즘들을 가지고 시작하여, 이들을 새 기계의 장점들에 맞도록 하는 것이다. 이 방식은 옛 알고리즘과 구조를 새 기계에서 지속시키는데 있어 확실히 고무적이다.

상당한 계산 능력을 가진 마이크로프로세서들의 값이 아주 싸지고 있다. 또 하나 아주 명백한 방식은 큰 문제를 풀기 위해 많은 프로세서들을 함께 사용하는 것이다. 어떤 알고리즘들은 이런 방법으로 분산되어 잘 수행될 수 있는 적응력이 있다. 그러나 또 다른 어떤 알고리즘들은 그렇지 못하다.

값싸고 강력한 기능을 가진 프로세서들의 개발은 또한 새 프로세서들을 설계하고 만드는 데 이용되는 범용 도구들의 개발을 가져왔다. 이런 도구들은 또한 특정 문제들을 위한 특수용 기계들의 개발을 증가 시켰다. 만약 어떤 중요한 알고리즘을 실행시키기에 적합한 기계도 없다면, 우리는 그런 기계를 설계하고 만들 수 있다. VLSI 칩에 적합한 많은 문제들을 풀 수 있는 적절한 기계들이 설계되고 만들어질 수 있다.

이러한 모든 방식들의 한 공통된 맥락은 병렬 실행(parallelism)이다; 많은 일들이 한 순간에 발생하도록 함으로 시간을 절약하려는 것이다. 이런 일은 잘 정돈되어 행해지지 않으면, 혼돈 상태를 야기시킬 수 있다. 아래에서는, 일부 특정 계층의 문제들에 대해 상당한 병렬 실행을 가능케 하는 기법들을 보여주는 두 가지 예를 다룰 것이다. 이 개념은 한 프로그램이 수행될 수 있는 M 개의 프로세서를 갖는 것이다. 따라서, 잘 동작되면, 프로그램을 전보다 M 배나 빠르게 수행시킬 수가 있을 것이다.

같은 문제를 풀기 위해 M 개의 프로세서가 함께 동작함에 관련하여 생각되는 몇 개의 문제점들이 있다. 가장 중요한 것은 이들이 어떤 식으로든 통신을 해야 한다는 것이다: 이들을

상호연결하는 선들이 있어야 하고 그 선들을 통해 데이터가 오고 갈 수 있어야 한다. 더욱이, 허용되는 상호연결의 형태에도 물리적 제약이 있다. 예를 들어, “프로세서”가 상호연결을 위해 32핀을 갖는 IC 칩이라고 하자. 비록 1000대의 그와 같은 프로세서들이 있다 해도, 최대 32대만을 연결할 수 있다. 프로세서들의 연결방법의 선택은 병렬 계산에 있어서 아주 근본적인 중요성이 있다. 게다가, 이 결정이 미리 되어 함을 기억함이 중요하다: 프로그램은 해결해야 할 문제에 따라 그 수행 방법을 바꿀 수 있지만, 기계는 일반적으로 그 부속품들의 연결 상태를 다시 바꿀 수가 없다.

병렬 계산을 고정된 상호연결 패턴을 갖는 독립된 프로세서의 관점에서 보는 일반적 견해는 앞서 기술한 세 가지 영역(방식)들 각각에 적용된다: 슈퍼 컴퓨터는 특정 프로세서들과(그 구조에 필수적이고, 그 성능에 여러 양상으로 영향을 주는) 상호연결 패턴을 갖는다. 상호연결된 마이크로프로세서들은 상대적으로 단순연결된 소수의 프로세서를 가지며, VLSI 칩들은 아주 복잡하게 상호연결된 아주 많은 프로세서들(회로소자들)을 갖고 있다.

많은 다른 병렬 계산에 대한 견해들이 von Neumann 이래로(값싼 프로세서들이 나온 이래로 새로운 관심거리로서) 계속 연구되어 왔다. 이에 관련한 모든 문제점(issue)들을 다루는 것은 이 책의 범위를 넘어선다. 그대신, 몇 개의 잘 알려진 문제들을 위해 제안된 두 특정 기계를 살펴보기로 한다. 이 기계들은 알고리즘 설계에 대해 기계 구조가 주는 영향을 또는 그 반대 경우의 영향을 보여준다. 상호협력 동작이 여기에 있다: 분명히 새로운 컴퓨터는 어디에 사용될 것인지에 대한 목적 없이는 설계되지 않을 것이고, 사람들은 가장 중요한 기본적인 알고리즘들을 실행하려 할 때 현존하는 것 중에서 최고로 좋은 컴퓨터를 사용하고 싶어할 것이다.

완전-셔플(Perfect Shuffles)

프로그램 대신에 기계로 알고리즘을 구현할 때 생기는 일부 문제점을 설명하기 위해, 하드웨어 구현에 적합한 한 병합(merging) 방법을 살펴보겠다. 앞으로 보게 될 것처럼, 동일한 일반적인 방법이, 병합 문제 뿐만 아니라 여러 다른 문제들을 해결하기 위해 M 개의 프로세서들의 병렬 연산을 성취시켜 주는 근본적인 상호연결 패턴을 갖는, “알고리즘 기계(하드웨어로 구현된 알고리즘)”의 설계 시에 적용된다.

앞서 언급한 것처럼, 한 문제를 풀기 위한 프로그램의 작성과 한 기계의 설계간의 근본적인 차이는, 프로그램은 풀고 있는 문제의 어느 한 순간에 따라 융통성 있게 연산을 수행할 수

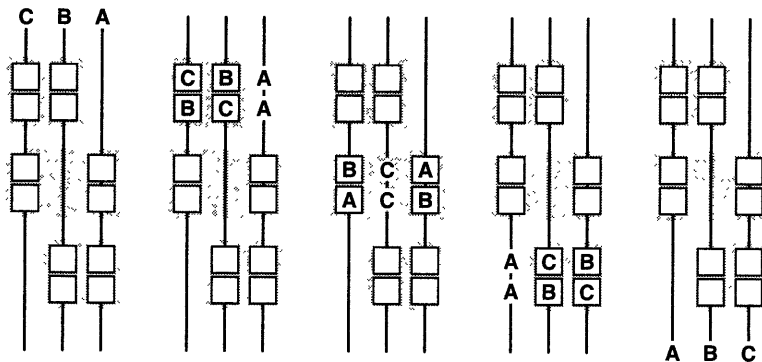


그림 40.1 세 개의 입력을 정렬하여 출력하는 정렬 기계

있는 반면에, 기계는 미리 “연결되어” 있으므로 항상 일정한 순서로 연산을 수행한다. 이 차이점을 알기 위해, 8장에서 공부한 첫 번째 정렬 알고리즘인 sort3를 살펴보자. 이 프로그램은, 어떤 세 숫자가 데이터 상에 있든 간에, 항상 동일한 순서로 세 개의 기본적인 “비교-교환(compare-and-exchange)” 연산을 수행한다. 지금까지 배워 온 어떤 다른 정렬 알고리즘도 이런 성질을 갖고 있지 않다; 이들 변수는 이전 비교들의 결과에 따라 일련의 비교들을 수행하므로 하드웨어로 구현할 때 심각한 문제점들이 있다.

특히, 만약 두 개의 입력 선과 두 개의 출력 선을 갖는 한 하드웨어 부품이 두 입력을 비교하고(필요하다면) 출력시 이 둘을 교환한다면, 이런 하드웨어 부품 세 개를 함께 사용함으로써 그림 40.1에 보이는 것과 같은 세 개의 입력과 세 개의 출력을 갖는 정렬 기계(sorting machine)를 만들 수 있다.

여기서, 정렬 결과를 출력하기 위해 첫 번째 상자(부품)는 C와 B, 두 번째 상자는 B와 A, 그리고 세 번째 상자는 C와 B를 교환한다. 이 기계는 어떤 세 입력들의 조합이든지 간에 정렬하여 출력한다.(바로 sort3가 했던 것처럼)

물론, 이 방식에 근거하여 실제로 정렬 기계를 만들기에 앞서, 많은 세부 사항들에 대한 연구 작업을 해야 한다. 한 예로서, 입력들의 코드화 방법은 아직 결정되지 않았다. 한 가지 방법은 위 다이어그램 상의 각 선을, 그 선 내부에 한 선당 한 비트씩을 전달하는 데이터 전송에 충분한 수의 선들을 갖는, 하나의 “버스 (bus)”로서 간주하는 것이다. 또 한 가지 방법은 비교-교환 소자들이 그들의 입력들을 한 번에 한 비트씩(최대 유효 비트(most significant bit) 먼저) 선을 통해 읽게 하는 것이다. 또 하나의 미 결정된 사항은 타이밍(timing)이다: 어떤 비교-교환 소자도 입력이 준비되기 전에 동작을 수행하지 않도록 해주는 메커니즘이 있

어야 한다. 물론 여기서 이와 같은 회로 설계의 문제 점들을 더 깊게 설명할 수는 없다. 그 대신, 보다 큰 문제들을 풀기 위해 비교-교환 소자같은 간단한 프로세서들의 상호연결에 관련한 고수준의 문제들을 집중하여 다루기로 한다.

우선, 두 개의 정렬된 파일들을 병합하는 알고리즘을 생각해 보자. 이 알고리즘은 병합될 특정 숫자들과는 무관하게 일련의 “비교-교환” 연산을 한다. 따라서 하드웨어로 구현하기에 적합하다. 그림 40.2는 병합되어 한 파일이 될, 여덟 개씩의 키(key)값을 갖는 두 개의 정렬된 파일에 대한 “비교-교환” 연산을 보여준다.

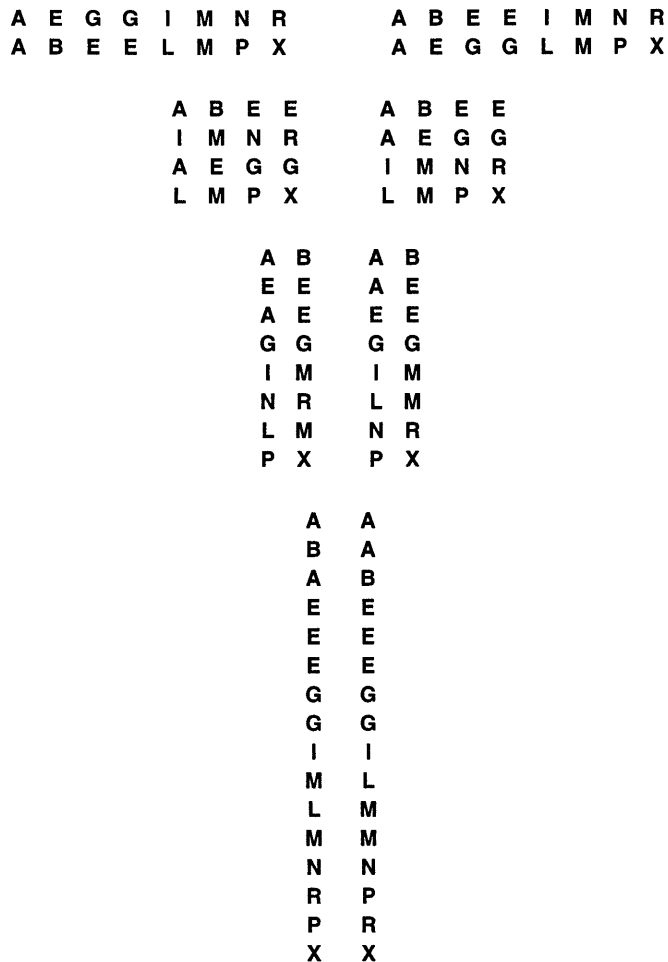


그림 40.2 분할 끼워넣기 병합

먼저, 한 파일을 다른 파일 밑에 추가하고, 이(수직으로) 인접한 두 파일을 비교하여 크기가 작은 파일 밑에 큰 파일이 위치하도록(필요하다면) 교환을 한다. 그리고 나서는, 각 줄을 반으로 분할(split)하여 그 분할된 반 줄들을 끼워넣기(interleave)한다. 그리고 나서는, 두 번째 줄과 세 번째 줄에 있는 숫자들에 대해 동일한 비교-교환 연산을 한다.(나머지 두 줄의 비교는 벌써 이전에 정렬되었기 때문에 불필요하다) 이렇게 하여 도표의 행과 열들이 정렬된다. 이 사실은 이 방법의 한 근본적인 성질이다: 비록 이의 증명은 생각보다는 어렵지만 독자는 이것이 사실인지를 조사해 볼 수도 있다.

이 성질은 일반적으로 동일한 연산에 의해 보존되어짐이 밝혀져 있다: 각 줄을 반으로 분할하고 그 분할된 것들을 끼워넣고, 각기 다른 줄에 있었으나 지금은 수직으로 인접한 값들 간에 비교-교환을 수행한다. 매 스텝마다 행의 수가 두 배가되고, 열의 수는 반으로 줄어드나, 아직도 행과 열들은 정렬된 상태로 있다. 처음에는, 16열 1행이었고, 그 다음에는 8열 2행, 4열 4행, 2열 8행, 마지막으로 1열-16행이 되어 정렬이 끝나게 된다.

성질 40.1 N 개의 원소를 갖는 두 정렬된 파일들의 병합은 약 $\log N$ 스텝 내에 행해질 수 있다.

만약 $N = 2^n$ 이면, 이 방법은 정확히 N 스텝이 걸릴 것임이 명백하다. 스텝 각각은 $N/2$ 회보다 적은 비교를 필요로 한다. 이 방법이 정렬을 수행함을 증명하려면, 열들이 정렬된 채로 남아 있음을 증명할 필요가 있다: 이 증명은 연습문제로서 남겨 둔다. 줄들의 크기가 다른 경우에도 모조 키값들을(dummy key) 추가함으로써 처리할 수 있다. □

앞에서 설명한 “각 줄을 반으로 분할하고 그 분할된 것들을 끼워넣어라”는 기본적인 동작을 종이에 그려서 보여주는 것은 쉽다. 그러나, 이를 어떻게 기계 부품들의 연결로(wiring)으로 바꿀까? 이 질문에 대한 한 놀랍고도 세련된 답이, 도표들을 다른 방법으로 작성함으로써, 나올 수가 있다. 도표들을 이차원으로 작성하는 대신에, 열-우선 순서로써 정리된 일차원 리스트를 작성한다: 먼저 첫 열에 원소들을 넣고, 다음에 둘째 열에 원소들을 넣고, 하는 식으로 계속하여 작성한다. 비교-교환은 단지 수직으로 인접한 값들 간에만 행해지므로, 이러한 리스트 작성은 각 단계가 한 그룹의 비교-교환 상자들을 포함하고 있음을 의미한다. 이 상자들은 값들을 비교-교환 상자들 안으로 가져오는 데 필요한 “분할-끼워넣기” 동작에 따라서 함께 연결된다.

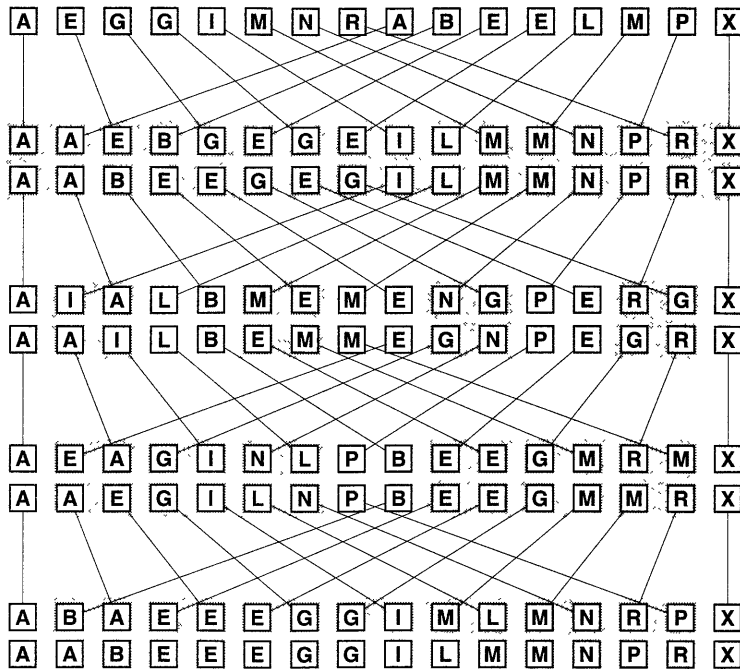


그림 40.3 완전-셔플 형태의 홀수-짝수 병합

이 결과가 그림 40.3에 있는데, 이는 도표들이 열-우선 순서로 작성된 것을 제외하고는 정확하게 위 도표들의 사용에 대한 설명과 일치한다. 독자는 이 다이어그램과 위에 주어진 도표들이 일치함을 조사해 보고 싶을 것이다. 비교-교환 상자들이 그려지고, 분할-끼워넣기 동작 시 어떻게 원소들이 이동하는지를 보여주는 선들이 그려진다: 놀랍게도, 이 표현의 경우, 각 “분할 끼워넣기” 동작은 동일한 상호연결 패턴으로 변환된다. 이 패턴을 완전-셔플(perfect shuffle)이라 부른다. 왜냐하면 선들이 두 부분으로 분할된 카드들이 각 부분에서 한 장씩 끄집어 내어져서 섞이는 것과 똑같은 방법으로 선들이 끼워넣어지기 때문이다.

이 방법은 1968년에 이 방법의 고안자인 K. E. Batchier에 의해 홀수-짝수(odd-even) 병합으로 명명됐다. 이 방법의 핵심적인 특징은 각 단계에서의 모든 비교-교환 동작들이 병렬로 행해질 수 있다는 점이다. 성질 40.1에서 언급된 것처럼, 병렬 처리됨은 매우 중요하다. 왜냐하면 N 개의 원소들을 갖는 두 파일들이 $\log N$ 병렬 스텝내에(도표의 행들의 수는 매 스텝마다 반이 된다) $N \log N$ 보다 적은 수의 비교-교환 상자들을 사용하면서 병합될 수 있기 때문이다. 앞의 설명에 따르면, 이것은 아주 간단히 나온 결과처럼 보일 수도 있다; 실제로는, 이러한 기계를 찾는 문제는 연구자들을 오랫동안 어렵게 만들었다.

Batcher는 또한 바이토닉(bitonic) 병합이라 불리는, 홀수-짝수 병합과 아주 밀접한 관계가 있는 그러나 이해하기는 더 어려운, 알고리즘을 만들었는데, 이 알고리즘은 그림 40.4에 있는 것 같은 더 단순한 기계를 만들 수 있게 해 준다. 이 방법은 위에서와 마찬가지로 도표들에 대한 “분할-끼워넣기” 동작으로서 설명된다. 그러나 한 가지 차이점은 두 번째 파일을 가지고 역 정렬 순서로서 시작하고, 항상 같은 줄에 있었던 수직으로 인접한 값들을 가지고 비교-교환을 한다는 점이다. 여기서 이 방법이 동작함을 증명하지는 않겠다: 우리의 관심사는 이 방법이 홀수-짝수 병합의 나쁜 특징-첫 단계의 비교-교환 상자들이 다음 단계들에 있는 상자들로부터 한 자리 쉬프트(shift)하는 것을 제거하는데 있다. 그림 40.4에서 보여지는 것처럼, 바이토닉 병합의 각 단계는 같은 수의 비교기(상자)를 같은 위치에 갖고 있다.

바이토닉 병합은 상호연결 뿐만 아니라 비교-교환 상자들의 위치에 있어서도 규칙성이 있다. 홀수-짝수 병합보다는 많은 비교-교환 상자가 있게 된다. 그러나 이것은 별 문제가 되지 않는다.

왜냐하면 홀수-짝수 병합의 경우와 같은 수의 병렬 스텝을 갖기 때문이다. 바이토닉 병합 방법은 단지 N 개의 비교-교환 상자를 사용하여 병합을 할 수 있다는데 그 중요성이 있다.

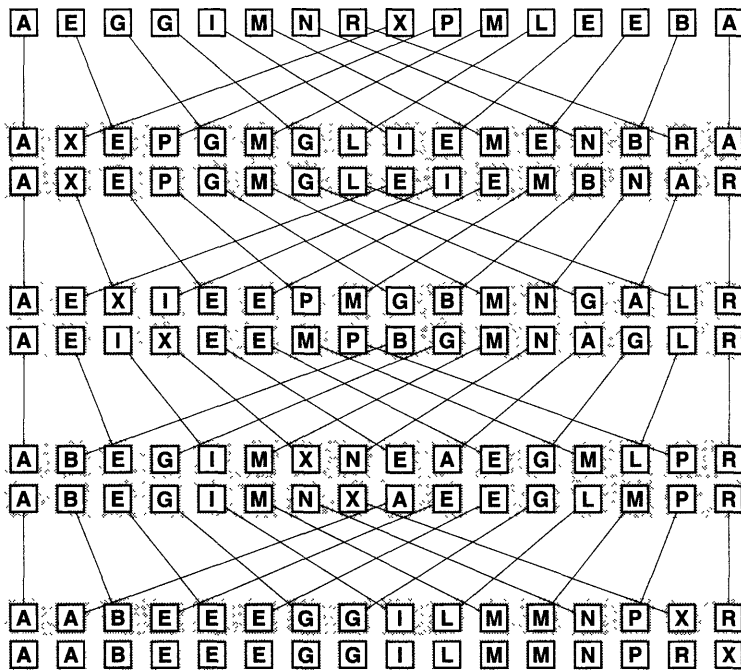


그림 40.4 완전-서플 형태의 바이토닉 병합

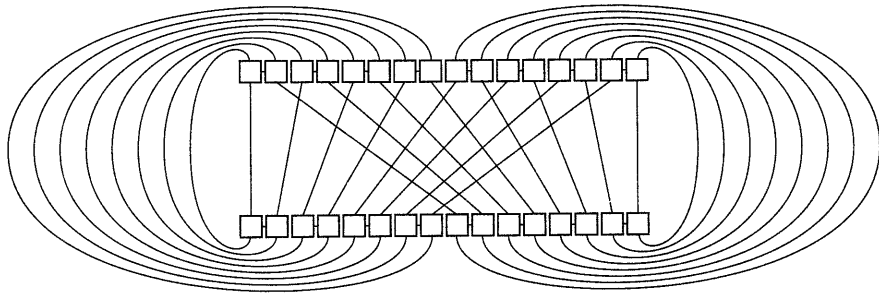


그림 40.5 완전-서플 기계

핵심 개념은 단순히 도표의 행들을 묶어서 한 쌍의 행을 만들고, 그리고 나서 그림 40.5에 있는 것 같은 순환하는(cycling)기계를 만드는 것이다. 이같은 기계는 그림의 각 단계마다 한 번씩 $\log N$ 비교-교환 서플 “사이클”을 수행할 수 있다.

이것은 아주 “이상적인” 병렬 수행 성능이 아님에 주목하라: 각각 N 개의 요소들을 갖는 두 개의 파일을 한 개의 프로세서를 갖고 N 에 비례하는 많은 스텝 내에 병합할 수 있으므로, N 개의 프로세서를 사용할 경우 일정 상수 번의 스텝 내의 병합을 할 수 있다고 기대 할 수도 있다. 그러나, 이 경우에 이런 이상적인 결과를 얻는 것이 불가능하다는 것과 이 기계가 병합 시 비교-교환 상자를 사용하여 최대 가능한 성능을 발휘함이 증명되어져 있다.

완전-서플 상호연결 패턴은 여러 다른 문제들에서도 적합하다. 예를 들어, 만약 한 2^n 행- 2^n 열의 정방 행렬이 행-우선 순서로 유지된다면, n 개의 완전-서플들은 행렬을 전치(transpose)할 것이다.(즉, 열-우선 순위로 변환된다) 더 중요한 예제들 중에는(41장에서 배울) 빠른 Fourier 변환,(위에서 설명한 방법들 중 하나를 재귀적으로 적용함으로써 만들어 질 수 있는) 정렬, 그리고 다항식 계산 등을 비롯한 많은 문제들이 있다. 이러한 문제들 각각은 그림 40.5에서 보여주는 것과 동일한 상호연결을 갖는, 그러나 다른(다소 더 복잡한) 프로세서들을 가지는 순환하는 완전-서플 기계를 사용하여 해결될 수 있다. 일부 연구자들은 “범용” 병렬 컴퓨터들에 해해서도 완전-서플 상호연결을 이용해야 한다고 말하고 있다.

시스톨릭 배열(Systolic Arrays)

완전-서플이 한 가지 문제는 상호연결 시 사용된 선들이 길다는 것이다. 게다가, 많은 선들이 서로 교차한다: N 개의 선을 가진 서플은 N^2 에 비례하는 많은 수의 교차를 갖는다. 이

두 성질들이 완전-서플 기계가 실제로 만들어질 때 어려움을 주는 것으로 나타났다: 긴 선들은 시간의 지연을 가져오고 선들의 교차는 상호연결을 비싸고 불편하게 한다.

이 두 가지 문제점들을 피하는 한 가지 자연스런 방법은 프로세서들을 물리적으로 인접한 프로세서들 하고만 연결되도록 하는 것이다. 앞서서와 마찬가지로, 프로세서들은 동기적으로 (synchronously) 동작한다: 매 스텝에서, 각 프로세서는 그 이웃들로 부터의 입력을 읽고, 계산을 하고, 그 결과를 이웃들에게 출력한다. 이러한 방법이 반드시 제약을 받는 방식이 아님이 밝혀졌다. 실제로 H. T. Kung은 1978년에 이와 같은 프로세서들의 배열을 시스톨릭 배열 (Systolic array)이라 불렀는데, (왜냐하면, 그들 내부에서의 데이터가 흘러가는 방법이 심장 박동을 연상시키기 때문이다) 이 배열은 일부 근본적인 문제들에 대해 아주 효율적인 프로세서들의 사용을 가져다준다.

한 전형적인 응용분야로서, 행렬 벡터 곱셈에서 시스톨릭 배열을 사용하는 경우를 생각해 보자. 예로서, 다음의 행렬 계산을 고려하기로 한다:

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \\ 2 \end{pmatrix} = \begin{pmatrix} 8 \\ 2 \\ -1 \end{pmatrix}$$

이 계산은 그림 40.6에 있는 것 같은 세 개의 입력 선과 두 개의 출력 선을 갖는 일련의 단순한 프로세서들 상에서 수행될 것이다. 다섯 개의 프로세서가 사용되는 이유는 아래에 설명된 것처럼 입력들을 나타냄과 출력들의 읽음을 시간에 맞도록 하기 위함이다.

각 스텝 동안, 각 프로세서는 좌측에서 한 입력을 읽고, 상부에서 하나를 읽고, 그리고 우측에서 나머지 하나를 읽는다. 특히, 우측 출력은 좌측 입력에 무엇이 있던 간에 받아서 출력하고, 좌측 출력은 좌측과 상부 입력을 곱한 값에 우측 입력을 더해서 얻어진 결과이다. 프로세서들의 한 중요한 특징은 이들이 항상 입력들을 동적으로 변환시켜 출력들로 보내는 작업을 수행한다는 것이다; 이들은 계산된 값들을 전혀 “기억”할 필요가 없다. (이것은 완전-서플

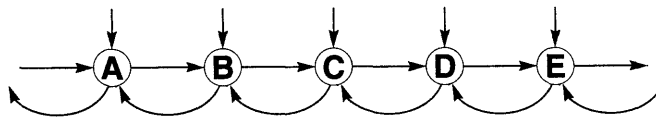


그림 40.6 시스톨릭 배열

기계의 프로세서들의 경우에도 마찬가지이다) 이것이 바로 하드웨어 설계시 부가되는 한 기본적인 규칙이다. 왜냐하면 “메모리(기억)” 능력의 추가는(상대적으로) 아주 값비싼 것이기 때문이다.

앞 절에서 시스톨릭 기계를 위한 “프로그램(동작 설명)”이 주어졌다; 이 설명을 끝내기 위해, 어떻게 입력 값들이 기계에 입력되는지를 정확히 설명할 필요가 있다. 모든 입력 값들이 한 번에 주어지고 모든 수행 결과 값들이 얼마 뒤에 출력되는 완전-서플 기계와 비교하여 볼 때, 이 입력 타이밍은 시스톨릭 기계의 필수적인 특징이다.

일반적인 방법은, 프로세서들의 상부 입력들을 행렬에 넣고 주 대각선을 기준으로 투영하고 45도 회전시킨다. 그리고 프로세서 A의 좌측 입력으로서 입력 벡터를 넣는다. 이 입력 값들은 물론 프로세서 A의 우측 프로세서로 전달된다. 중간 결과 값들은 시스톨릭 배열상에서 우측에서 좌측으로 전달되고, 궁극적으로, 프로세서 A의 좌측 출력에 최종 결과 값이 출력되어 나타난다. 그림 40.7은 앞의 예제에 대한 특정 타이밍을 보여준다.

입력 벡터는 프로세서 A의 좌측 입력 단에, 스텝 1, 3, 5에서 주어지고, 이 각 스텝의 다음 스텝에서, 우측에 있는 다음 번 프로세서로 전달된다. 입력 행렬은 스텝 3에서 시작하여 프로세서들의 상부 입력 단들로 입력되는데, 입력이 회전되어졌으므로 다음 번 스텝들부터는 행렬의(우측에서 좌측으로) 대각선에 있는 값들이 입력된다. 출력 벡터는 프로세서 A의 좌측 출력 단으로 스텝 6, 8, 그리고 10에서 출력된다.(다이아그램에서, 이 출력들은 A의 좌측에 프로세서가 있는 것으로 가정하면 이 가상 프로세서의 우측 입력으로서 나타난다. 이들을 모은 것이 답이 된다)

실제 계산 과정은 시스톨릭 배열에서 우측에서 좌측으로 이동하는 우측 입력들(좌측 출력들)을 따라감으로서 추적될 수 있다. 모든 계산 과정들은 스텝 3까지는 결과치로서 0을 산출한다; 스텝 3이후에서는 프로세서 C가 그 좌측 입력으로 1 그리고 상부 입력으로 1을 가진다. 그래서 결과로서 1이 계산된다. 이 계산 결과는 스텝 4에서 프로세서 B의 우측 입력으로 보내진다. 스텝 4에서, 프로세서 B는 모든 세 개의 입력 값을 갖고서, 결과치 16을 산출한다. 이 역시 스텝 5에서 프로세서 A로 전달된다. 그 사이에, 프로세서 D는 스텝 5에서 프로세서 C가 사용하는 값 1을 계산하여 전달한다. 그리고 스텝 5에서, 프로세서 A는 8을 결과치로서 계산해 내는데, 이것이 스텝 6의 첫 출력값이다. 프로세서 C는 스텝 6에서 B가 사용할 값을 계산하고, E는 스텝 6에서 D가 사용할 값으로서 -1을 계산해낸다. 두 번째 출력 값의 계산은 스텝 6에서 B에 의해 완료되어 스텝 8에서 프로세서 A에 의해 출력된다. 세 번째 출력값의 계산은 스텝 7에서 C에 의해 완료되어 B를 거쳐서 스텝 10에서 A에 의해 출력된다.

646 고수준 토픽들

처리 과정이 위에서 상세히 설명되었으므로 이 방법이 어느 정도의 상위 레벨에서는 좀 더 잘 이해된다. 그림 40.7의 중간 부들의 사각형 내의 숫자들은 입력 행렬 내의 값들 즉, 복사 값들이다.(물론 상부 입력 단으로의 입력을 위해 회전되고 투영된 행렬 값들이다) 만약 이 행렬을 받아들이는 프로세서들의 우측 출력에 해당하는 값들을 조사해 본다면, 세 개의 입력 벡터값들이 행렬의 각 행들에 대한 곱셈을 위해 정확히 필요한 시간과 위치에서 나타남을 알 수가 있다. 그리고 프로세서들의 좌측 출력 값들도 각 곱셈에 대한(입력 벡터와 각 행렬의 행의) 중간 결과값들을 보여준다. 예로서, 입력 벡터와 행렬의 중간 행의 곱셈은 각 부분연산값이 $1 * 1 = 1$, $1 + 1 * 5 = 6$, $6 + (-2) * 2 = 2$ 이다. 시스틀릭 배열은 타이밍을 잘 관리하여, 각 행렬 요소가 해당 입력 벡터엔트리와 또한 이 요소가 입력된 프로세서에서의 부분계산 값과 정확히 만나게 해줌으로써, 이 요소를 부분 계산에 결합시킨다.

이 방법은 확실한 방법으로 N 행- N 열의 행렬과 N 행-1열의 벡터를 $2N - 1$ 개의 프로세서를 사용하여 $4N - 2$ 스텝 내에 처리할 수 있도록 확장 가능하다. 결과적으로, 모든 프로세서가 매 스텝마다 유용한 작업을 수행하는 이상적인 상황이 되는 것이다; 이차식(quadratic) 알고리즘은 선형 개수의 프로세서들을 사용함으로써, 한 선형(linear) 알고리즘으로 변환된다.

지금까지는 병렬 계산의 한 일반적인 방법에 대해서만 설명했다. 시스틀릭 배열 기계를 만들기 전에, 논리 설계 시의 많은 세부 사항들에 대한 작업이 필요하다. 앞 예제에서 여러분은 시스틀릭 배열이 단순하면서도 아주 좋다는 것을 알 수가 있었을 것이다. 왼쪽 끝단의 출력 벡터가 마치 마술처럼 나타난다! 그러나, 각 개개의 프로세서는 위의 설명에서와 같이 아주 단순한 계산만을 수행한다: 이러한 마술 같은 효과는 상호연결기법과 입력들을 타이밍을 맞춰서 입력 단들에 넣어 주는 기법에 의해 얻어진 것이다.

완전-셔플 기계의 경우와 마찬가지로, 시스틀릭 배열들은 문자열 매칭 및 행렬 곱셈 같은 것들을 포함한 많은 매우 다른 유형의 문제들에 대해서도 사용될 수 있다. 또한, 일부 연구자들은 이 상호연결 패턴을 “범용” 병렬 처리 기법들에 대해서도 사용할 것은 제안하기도 한다.

전망(Perspective)

완전-셔플과 시스틀릭 기계들을 공부하면서 하드웨어 설계가 알고리즘 설계에 상당한 영향을 줄 수 있었다. 명백히, 하드웨어 설계는 새로운 알고리즘들이 만들어 질 수 있는 여건을 제공하고 알고리즘 설계자들에게 신선한 문제거리들을 가져다준다.

이 문제는 앞으로 연구해 볼만한 한 흥미있고 많은 결과들이 나올 수 있는 분야이다. 그러나 이에 관련하여 몇 가지 사항을 진지하게 언급하고 결론을 내리고자 한다. 첫째로, 위에서 개략적으로 설명한 병렬 계산에 대한 일반적인 방법들을 좋은 성능을 갖는 실제 알고리즘 기계로 만들기 위한 많은 공학적인 노력이 요구된다. 많은 분야들의 경우, 요구되는 자원들에 대한 비용 문제는 그냥 필요하다는것으로써 정당화 되지 않을 것이다; 기존 알고리즘들을 기존의 값싼 마이크로프로세서에서 동작시킴으로써 만들어지는 단순한 “알고리즘 기계”들도 아주 잘 동작할 수 있을 것이다. 예로서, 한 문제를 풀기 위해 이 문제를 여러 대의 마이크로프로세서들 상에 각기 부여하여 수행시킨다면, 기존 알고리즘을 사용하고, 상호연결 없이도 이 상적인 병렬 처리 성능이 얻어질 수 있다. 만약 정렬 되어 할 파일이 N 개 있고 이들의 정렬을 위해 이용 가능한 프로세서가 N 개 있다면, 왜 모든 N 개 파일의 정렬을 위해 N 개의 프로세서를 함께 사용하는 대신에 각 파일의 정렬을 위해 한 개씩의 프로세서를 할당하여 사용하지 않는가?

여러 병렬 처리 전략들이 알고리즘의 성능에 어떤 영향을 주는지를 평가하는 것은 아주 어렵다. 6장과 7장에서 논의된 모든 문제점들은 기계 그 자체를 변수로서 고려함과 동시에 고려 되어야 한다. 기계들을 비교하기 위한 가장 쉽고도(아마도) 가장 많이 이용되는 방법이 아주 크게 잘못 된 것일 수 있다: 두 개의 기계 상에서 한 동일한 알고리즘을 수행시키는 방식은, 만약 해당 알고리즘이 한 기계의 구조와는 아주 밀접히 연관되어 있고, 다른 한 기계와는 그렇지 않다면, 잘못 실험된 방식이다. 구조가 조화될 때가 그렇지 않을 때 보다 훨씬 좋을 수밖에 없다. 기계들을 공정하게 비교하려면, 해결 되어 할 문제에 초점을 맞추고 그 문제에 대해 각 기계 상에서 최적인 알고리즘을 찾아야 한다.

이 장에서 논의 된 것들 같은 기법들은 현재 매우 특별한 시간 이나 공간 조건을 갖는 분야에서서만 그 타당성을 인정받을 수 있다. 다양한 병렬 계산 방법들과 다양한 알고리즘들에 대한 그들의 영향(효과)들을 공부할 때, 아주 광범위하고 다양한 알고리즘들에 대해 성능 향상을 가져다주는 범용 병렬 컴퓨터들의 개발을 기대할 수 있다.

연습문제

1. 퀵정렬(quicksort)에서 병렬개념을 이용하기 위한 두 가지 가능한 방법을 개략적으로 설명하라.
2. “분할-끼워넣기” 방법이 열들이 정렬 되도록 하는 성질을 보존함을 증명하라.
3. Batcher의 바이토닉 방법을 사용하여 파일들을 병합시키는 C++ 프로그램을 작성하라.
4. Batcher의 바이토닉 방법을 사용하되, 어떤 서플들도 실제로 행해짐이 없이 파일을 병합시키는 C++ 프로그램을 작성하라.
5. 얼마나 많은 서플들을 가져야만 크기가 2^n 인 배열내의 모든 원소들을 원래 자리로 다시 들어가게끔 할 수 있는가?
6. 다음 문제에서 시스틀릭 행렬 벡터의 승수(multiplier)의 동작을 설명하기위해, 그림 40.7에서와 같은 도표를 그려라.

$$\begin{pmatrix} 2 & 1 & 4 \\ 3 & 0 & 1 \\ 1 & -1 & 3 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 3 \\ 8 \\ -1 \end{pmatrix}$$

7. N 행- N 열의 행렬과 N 행-1열의 벡터의 곱셈을 수행하는 시스틀릭 기계의 동작을 모의실험(simulate)하는 C++프로그램을 작성하라.
8. 한 행렬의 전치행렬(transposed matrix)을 얻기 위한 시스틀릭 배열의 사용 방법을 보여라.
9. M 행- N 열의 행렬과 N 행-1열의 벡터의 곱셈을 수행하는 시스틀릭 기계의 경우, 이를 위해 얼마나 많은 프로세서들과 얼마나 많은 스텝들이 요구되어지나?
10. 계산된 값들을 기억할 수 있는 능력이 있는 프로세서들이 있다 하자. 이들을 이용하여, 행렬-벡터 곱셈을 위한 한 단순한 병렬 기법을 만들라.

빈 면

빠른 Fourier 변환

가장 널리 이용되는 알고리즘들 중의 하나는 빠른 Fourier 변환(fast Fourier transform)이다. 이 변환은 많은 기본적인 수학 계산을 수행하는 한 효율적인 방법이다. Fourier 변환은 수학적 분석시 아주 중요하며 방대한 양의 연구들에 대한 주제거리이다. 이 계산에 대한 효율적인 알고리즘의 출현은 컴퓨터 계산 역사에 있어서 한 획기적인 사건이었다.

Fourier 변환의 응용 분야는 무수히 많다. 이 변환은 현재 널리 사용되는 신호처리 분야의 많은 근본적인 조작 기법들의 기본이 된다. 더구나, 앞으로 보게 되겠지만, 이 변환은 평범한 산술 문제들에 대한 알고리즘들의 효율을 향상시킬 수 있는 효율적인 방법을 제공해 준다. Fourier 변환에 대한 수학적 근거에 대해 개략적으로 설명하거나 많은 응용 분야들을 소개하는 것은 이 책에서 할 사항이 아니다. 여기서는, 이 변환을 위한 한 근본적인 알고리즘의 특징을 이제까지 공부해 온 일부 알고리즘의 관점에서 살펴보기로 한다.

특히, 이 알고리즘이 36장에서 배운 문제인 다항식 곱셈이 걸리는 시간을 어떻게 줄여 주는지를 알아보기로 한다. 어떻게 Fourier 변환이 다항식들의 곱셈에 이용될 수 있는지를 보이기 위해 아주 일부의 복소수 분석(complex analysis)의 기본 사항들만이 필요하며, 내재된 수학의 완전한 이해 없이도 빠른 Fourier 변환을 평가하는 것이 가능하다. 이 알고리즘은, 이제까지 본 다른 중요한 알고리즘들과 비슷한 방법으로, 분할-정복 기법을 적용한다.

평가, 곱셈, 보간(Evaluate, Multiply, Interpolate)

다항식 곱셈에 대해 개선된 방법을 얻기 위한 일반적인 전략은 차수 $N-1$ 의 다항식은 N 개의 다른 점에 있는 해당 값들에 의해 완전히 결정(completely determined)된다는 사실을 이

용하는 것이다. 차수 $N - 1$ 의 다항식 두 개를 곱하면, 차수 $2N - 2$ 인 다항식이 구해진다; 만약 $2N - 1$ 개의 점에서 그 다항식 값들을 찾을 수 있다면, 이 다항식은 완전히 결정된다. 그러나 곱해질 두 다항식을 해당 점에서 계산하고 그 해당 숫자들을 곱함으로써 임의의 점에서의 결과값을 얻을 수 있다.

이렇게하여, 차수 $N - 1$ 인 두 다항식의 곱셈을 위한 아래의 일반적인 규칙이 생겨났다:

- $2N - 1$ 개의 다른 점에서 입력된 두 다항식을 평가한다.
- 각 점에서 얻어진 두 값을 곱한다.
- 각 주어진 점들에서 주어진 값을 갖는 유일한 결과 다항식을 얻기 위해 보간 한다.

예로서, $p(x) = 1+x+x^2$ 이고 $q(x) = 2-x+x^2$ 일 때, $r(x) = p(x)q(x)$ 를 계산하기 위해 다섯 개의 점들에서 $p(x)$ 와 $q(x)$ 를 평가한다. 예로서 이 점들이 $-2, -1, 0, 1, 2$ 라 할 때, 다음과 같은 값들이 얻어진다:

$$\begin{aligned} [p(-2), p(-1), p(0), p(1), p(2)] &= [3, 1, 1, 3, 7], \\ [q(-2), q(-1), q(0), q(1), q(2)] &= [8, 4, 2, 2, 4]. \end{aligned}$$

이들을 항-대-항으로 곱하면 곱다항식에 대해 다음 값들이 구해진다:

$$[r(-2), r(-1), r(0), r(1), r(2)] = [24, 4, 2, 6, 28].$$

이들의 계수들은 보간을 함으로써 얻어진다. Lagrange공식에 의해,

$$\begin{aligned} r(x) &= 24 \frac{x+1}{-2+1} \frac{x-0}{-2-0} \frac{x-1}{-2-1} \frac{x-2}{-2-2} \\ &\quad + 4 \frac{x+2}{-1+2} \frac{x-0}{-1-0} \frac{x-1}{-1-1} \frac{x-2}{-1-2} \\ &\quad + 2 \frac{x+2}{0+2} \frac{x+1}{0+1} \frac{x-1}{0-1} \frac{x-2}{0-2} \\ &\quad + 6 \frac{x+2}{1+2} \frac{x+1}{1+1} \frac{x-0}{1-0} \frac{x-2}{1-2} \\ &\quad + 28 \frac{x+2}{2+2} \frac{x+1}{2+1} \frac{x-0}{2-0} \frac{x-1}{2-1} \end{aligned}$$

가 얻어지는데, 이들을 단순화하여 나온 결과는 다음과 같다:

$$r(x) = 2 + x + 2x^2 + x^4.$$

이제까지 설명한 것처럼, 이 방법은 다항식 곱셈에 대해 그렇게 매력 있는 알고리즘은 아닙니다. 왜냐하면 위 두 가지 평가(Horner방법의 반복 적용)와 보간(Lagrange공식)에 대한(이제까지 우리가 아는 바로는) 최고로 좋은 알고리즘도 N^2 연산을 필요로 하기 때문이다. 그러나, 이 방법은 어느 $2N - 1$ 개의 점에 대해서든지 동작하고, 평가와 보간이 어떤 점들의 집합의 경우 다른 점들의 집합들의 경우보다 더 쉬움을 기대해 볼 수도 있기 때문에, 좀더 나은 알고리즘을 찾을 수 있는 가능성이 있다.

복소수 단위근들(Complex Roots of Unity)

다항식 평가 및 보간에 사용할 가장 편리한 점들은 복소수, 실제로는, 복소수 단위근들(complex roots of unity)이라 불리는 복소수들의 한 특정 집합임이 알려져 있다.

복소수 분석에 대한 몇 가지 사실들을 잠시 복습해 봄이 필요하다. $i = \sqrt{-1}$ 은 허수(imaginary number)이다: 비록 $\sqrt{-1}$ 이 실수로서는 의미가 없지만 이를 i 로 명명하고 주고, 이 이름을 가지고 대수적인 조작을 하고, i^2 이 나타나면 -1 로 바꾸어 줌이 편리하다. 복소수는 실수부와 허수부로 구성된다; 보통 $a+bi$ 로 표기하는데, 여기서 a 와 b 는 실수이다. 복소수들을 곱하려면, 평상시의 곱셈 규칙을 적용하되 i^2 은 -1 로 바꾸어 준다. 예로서

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$$

이다. 곱셈 시에 종종 실수부나 허수부가 삭제 될 수도 있다. 예로서

$$(1 - i)(1 - i) = -2i,$$

$$(1 + i)^4 = -4,$$

$$(1 + i)^8 = 16.$$

등의 경우이다. 여기서 마지막 방정식을 $16 = \sqrt{2}^8$ 으로 나누면,

$$\left(\frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}\right)^8 = 1.$$

이 얻어진다. 일반적으로 곱셈이 계속되면 1이 되는 많은 복소수들이 있다. 이들을 일컬어 복소수 단위근들이라 한다. 실제로, 각 N 에 대해, $z^N = 1$ 인 복소수가 정확히 N 개 있음이 알려져 있다. 이들 중의 하나인 w_N 으로 명명된 복소수를 주된(principal) N 번째 단위근이라 한다; 나머지들은 w_N 을 k 승 ($k = 0, 1, 2, \dots, N - 1$ 에 대해)함으로써 얻어진다. 예로서, 8번째 단위근들을 다음과 같이 나열할 수 있다:

$$w_8^0, w_8^1, w_8^2, w_8^3, w_8^4, w_8^5, w_8^6, w_8^7.$$

첫 번째 단위근 w_N^0 은 1이고 두 번째 w_N^1 은 주된 단위근(principal root)이다. 또한 N 이 짝수인 경우, 근 $w_N^{N/2}$ 은 -1이다.(왜냐하면 $(w_N^{N/2})^2 = 1$ 이므로) 근들의 정확한 값들은 당분간 중요치 않다. 여기서는 단지 N 번째 단위근의 N 승 값이 1이어야만 한다는 기본 사실에서 쉽게 유도 되는 단순한 성질들만을 사용할 것이다.

단위근들에서의 계산(Evaluation at the Roots of Unity)

구현의 핵심은 N 번째 단위근들에서 차수 $N - 1$ 인 다항식을 평가하는 한 프로시저이다. 즉, 이 프로시저는 다항식을 정의하는 N 개의 계수들을 모든 N 번째 단위근들에서 다항식을 평가하여 나온 N 개의 값들로 변환시킨다.

이것은 우리가 정말로 원했던 것과 일치하지 않는 듯 하다. 왜냐하면 다항식 곱셈 프로시저의 첫 스텝에서, $2N - 1$ 개의 점에서 차수 $N - 1$ 인 다항식들을 평가할 필요가 있기 때문이다. 실제로, 차수 $N - 1$ 인 다항식은(상위 $N - 1$ 개의 계수 값이 0인) 차수 $2N - 2$ 인 다항식으로 생각 될 수 있으므로, 아무 문제가 없다.

N 개의 점에서 함께 차수 $N - 1$ 인 다항식을 평가하기 위해 사용할 알고리즘은 단순한 분할-정복 전략에 근거하여 동작하는 알고리즘이다. 다항식들을 중앙에서(4장의 곱셈 알고리즘에서와 같이) 나누지 않고, 홀수 번째 항들과 짝수 번째 항들의 두 부분으로 나눈다. 이러한 분할은 계수의 개수의 반만 가지는 다항식들로서 쉽게 표현된다. 예로서, $N = 8$ 인 경우, 항들을 재배열하면 다음과 같다:

$$\begin{aligned}
p(x) &= p_0 + p_1x + p_2x^2 + p_3x^3 + p_4x^4 + p_5x^5 + p_6x^6 + p_7x^7 \\
&= (p_0 + p_2x^2 + p_4x^4 + p_6x^6) + x(p_1 + p_3x^2 + p_5x^4 + p_7x^6) \\
&= p_e(x^2) + xp_o(x^2)
\end{aligned}$$

N 번째 단위근들은 한 단위근을 제공하면 또 하나의 단위근이 만들어지기 때문에, 이러한 분할에 편리하다. 이보다 더 좋은 점은 N 이 짝수일 때 한 개의 N 번째 단위근을 제공하면, 한 $\frac{1}{2}N$ 번째 단위근($\frac{1}{2}N$ 번째 곱해질 때 1이되는 수)이 되는 것이다. 이것이 분할-정복 방법이 동작하는 데 필요한 바로 그 성질들이다. N 개의 점들에 대해 N 개의 계수들을 갖는 한 다항식을 평가하기 위해, 이 다항식은 $\frac{1}{2}N$ 개의 계수들을 갖는 두 개의 다항식으로 분할된다. 이 분할된 두 개의 다항식은 단지 $\frac{1}{2}N$ 개의 점($\frac{1}{2}N$ 번째 단위근)들에 대해서만 평가 될 필요가 있다.

이를 좀 더 확실히 알기 위해, 아래의 여덟 번째 단위근들에 대한 차수가 7인 다항식 $p(x)$ 를 평가 해보자:

$$W_8 : w_8^0, w_8^1, w_8^2, w_8^3, w_8^4, w_8^5, w_8^6, w_8^7,$$

$w_4^8 = -1$ 이므로, 이 단위근들은 다음과 같다:

$$W_8 : w_8^0, w_8^1, w_8^2, w_8^3, -w_8^0, -w_8^1, -w_8^2, -w_8^3,$$

이 들의 각각을 제공하면, 아래와 같이 네 번째 단위근들의 집합 $\{W_4\}$ 가 두 번 나타난 형태가 된다:

$$W_8^2 : w_4^0, w_4^1, w_4^2, w_4^3, w_4^0, w_4^1, w_4^2, w_4^3,$$

이제 방정식

$$p(x) = p_e(x^2) + xp_o(x^2)$$

은 이러한 순서들에서 어떻게 여덟 번째 단위근들에서 $p(x)$ 를 평가하는지를 바로 보여주고 있다. 먼저, $p_e(x)$ 와 $p_o(x)$ 를 네 번째 단위근들에서 평가한다. 그리고 위 방정식의 x 를 여덟 개의 단위근들 각각으로 대치시킨다. 이를 위해서는 적절한 p_e 값을 여덟 번째 단위근을 적절한 p_o 값과 곱한값에 더해야 한다:

$$p(w_8^0) = p_e(w_4^0) + w_8^0 p_o(w_4^0),$$

$$p(w_8^1) = p_e(w_4^1) + w_8^1 p_o(w_4^1),$$

$$p(w_8^2) = p_e(w_4^2) + w_8^2 p_o(w_4^2),$$

$$p(w_8^3) = p_e(w_4^3) + w_8^3 p_o(w_4^3),$$

$$p(w_8^4) = p_e(w_4^0) - w_8^0 p_o(w_4^0),$$

$$p(w_8^5) = p_e(w_4^1) - w_8^1 p_o(w_4^1),$$

$$p(w_8^6) = p_e(w_4^2) - w_8^2 p_o(w_4^2),$$

$$p(w_8^7) = p_e(w_4^3) - w_8^3 p_o(w_4^3),$$

일반적으로, N 번째 단위근들에 대해 $p(x)$ 를 평가하려면, $p_e(x)$ 와 $p_o(x)$ 를 $\frac{1}{2}N$ 번째 단위근들에 대해서 재귀적으로 평가한 뒤에 위에서와 같이 N 번 곱셈을 수행한다. 이 방식은 N 이 우수일 때만 동작하므로 지금부터는 N 을 2의 제곱으로 가정한다. 그래서 재귀동작을 통하여 N 값은 항상 짝수로 있게 된다. $N = 2$ 일 때 재귀 수행이 끝나게 되고 $P_0 + P_1x$ 가 1과 -1에서 평가 된다. 이 결과로서, $P_0 + P_1$ 과 $P_0 - P_1$ 이 나온다.

성질 41.1 차수 $N - 1$ 인 다항식은 N 번째 단위근들에서 약 $N \lg N$ 번의 곱셈을 함으로써 평가될 수 있다.

사용된 곱셈의 회수는 기본적인 “분할-정복” 반복공식(recurrence) $M(N) = 2M(N/2) + N$ 을 만족시키는데, 그 해로서 $M(N) = N \lg N$ (6장의 공식 4)를 갖는다. 이 결과는 보간법에 대한 단순한 N^2 방법에 비해 볼 때, 상당한 향상을 가져왔다. 그러나, 이 방식은 단위근들에 대해서만 동작한다. □

이 성질은 기존의 N 개의 계수들로서 표현된 한 다항식을 그 다항식의 단위근들을 사용한 표현 방식으로 변형하는 방법을 가져다 준다. 이러한 다항식 표현 방식의 변환이 Fourier 변환이다. 지금까지 여기서 기술한 효율적인 재귀적 계산 프로시저를 “빠른” Fourier 변환(FFT)이라고 부른다.(이와 동일한 기법들을 다항식보다 더 일반적인 형태의 함수들에도 적용할 수 있다. 더 정확히 말해서, 우리는 지금 “이산형(discrete)” Fourier 변환을 하고 있다)

단위근들에서의 보간(Interpolation at the Roots of Unity)

이제 우리는 점들의 한 특정한 집합에서 다항식들을 평가하는 빠른 방법을 가지고 있으므로 이런 점들에서 다항식들을 보간하는 빠른 방법만 있으면, 빠른 다항식 곱셈법을 만들 수가 있다. 놀랍게도, 단위근들에 대해서는, 점들의 한 특정한 집합상에서 평가 프로그램 (evaluation program)을 수행시키면 보간이 행해진다! 이것은 Fourier변환의 근본적인 “역 (inversion)” 성질의 한 특정한 경우인데, 이로부터 많은 수학적 결과들이 유도 될 수 있다.

우리의 예제에서 $N = 8$ 일 경우, 보간문제는

$$r(w_8^0) = s_0, \quad r(w_8^1) = s_1, \quad r(w_8^2) = s_2, \quad r(w_8^3) = s_3,$$

$$r(w_8^4) = s_4, \quad r(w_8^5) = s_5, \quad r(w_8^6) = s_6, \quad r(w_8^7) = s_7$$

인 다항식

$$r(x) = r_0 + r_1x + r_2x^2 + r_3x^3 + r_4x^4 + r_5x^5 + r_6x^6 + r_7x^7$$

을 찾는 것이다.

점들이 복소 단위근들일때는, 보간 문제가 평가 문제의 “역”임이 분명하다. 만약

$$s(x) = s_0 + s_1x + s_2x^2 + s_3x^3 + s_4x^4 + s_5x^5 + s_6x^6 + s_7x^7$$

라고 하면, 계수

$$r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7$$

은 다항식 $s(x)$ 를 아래와 같은 단위근들의 “역”들에서 평가함으로써 구할 수 있다:

$$W_8^{-1} : w_8^0, w_8^{-1}, w_8^{-2}, w_8^{-3}, w_8^{-4}, w_8^{-5}, w_8^{-6}, w_8^{-7},$$

이들을 다음과 같이 다른 순서로 재배열하면, “평가” 프로그램 루틴을 사용하여 보간을 할 수 있다:

$$W_8^{-1} : w_8^0, w_8^7, w_8^6, w_8^5, w_8^4, w_8^3, w_8^2, w_8^1,$$

이의 증명을 위해 유한 합(sum)들에 대한 기본적인 조작이 필요하다: 이런 조작에 익숙하지 않으면 이 절을 생략해도 좋다. N 번째 단위근의 t 번째 근의 역에서 $s(x)$ 를 평가하면 다음과 같다:

$$\begin{aligned}
 s(w_N^{-t}) &= \sum_{0 \leq j < N} s_j(w_N^{-t})' \\
 &= \sum_{0 \leq j < N} r(w_N^j)(w_N^{-t})' \\
 &= \sum_{0 \leq j < N} \sum_{0 \leq i < N} r_i(w_N^j)'(w_N^{-t})' \\
 &= \sum_{0 \leq j < N} \sum_{0 \leq i < N} r_i w_N^{j(i-t)} \\
 &= \sum_{0 \leq i < N} r_i \sum_{0 \leq j < N} w_N^{j(i-t)} = Nr_t.
 \end{aligned}$$

마지막 항에서는, 거의 모든 것이 없어진다. 왜냐하면 $i = t$ 이면 합(\sum)이 N 이 되기 때문이다. 만약 $i \neq t$ 이면,

$$\sum_{0 \leq j < N} w_N^{j(i-t)} = \frac{w_N^{(i-t)N} - 1}{w_N^{(i-t)} - 1} = 0$$

가 된다. N 의 비율조정 인자(scaling factor)가 생김에 주목하라. 이것이 이산형 Fourier 변환에 대한 “역 정리(inversion theorem)”인데, 이 정리는, 한 가지 방법이 다항식을 계수들로서 표현되는 형태와 복소수 단위근들에서의 값들로서 표현되는 형태의, 두 가지 형태로 변환해 줄을 보여준다.

정리 41.2 차수 $N - 1$ 의 다항식은 약 $N \lg N$ 의 곱셈을 함으로써 N 번째 단위근들에서 보간 될 수 있다.

위의 수식 전개는 복잡해 보인다. 그러나, 그 결과들은 상당히 적용하기 쉽다: N 번째 단위 근들에 대해 한 다항식을 보간하려면, 보간 값들을 다항식 계수으로써 이용하여 “평가(evaluation)”를 위해 사용한 것과 동일한 프로시저를 수행 한 다음, 그 결과값들을 재배열하고 비율 조정(scale)한다. \square

구현(Implementation)

이제 우리는 두 다항식을 약 $N \lg N$ 동작을 하여 곱하는 한 “분할-정복” 알고리즘을 만들기 위한 모든 사항들을 알고 있다. 이의 구현을 위한 일반적인 방법은:

- $(2N - 1)$ 번째 단위근들에서 입력 다항식들을 평가한다.
- 각 점에서 얻어진 두 개의 값을 곱한다.
- $(2N - 1)$ 번째 단위근들에서 계산된 숫자들로서 정의된 다항식을 계산함으로써 얻어진 결과를 찾기위해 보간을 수행한다.

이 설명은 N 번째 단위근들에서 $N - 1$ 차 다항식을 평가하는 프로시저(eval)를 사용하는 프로그램으로 직접 변환 될 수 있다. 이 알고리즘에서의 모든 연산은 복소수 연산이다. C++에서는, 이 복소수 연산의 처리를 위해 일반 연산자들을 중첩(overloading)시켜서 복소수 처리를 위한 사용자가 정의한 연산자 형태로 바꿀 수 있다. 이렇게 함으로써, 알고리즘이 명확해진다. 다음 프로그램은 complex 연산자 형태가 있다는 가정 하에 구현된 것이다:

```
eval (p, outN, 0);
eval (q, outN, 0);
for (i = 0; i <= outN; i++) r[i] = p[i]*q[i];
eval (r, outN, 0);
for (i = 1; i <= N; i++)
    { t = r[i]; r[i] = r[outN+1-i]; r[outN+1-i] = t; }
for (i = 0; i <= outN; i++) r[i] = r[i]/(outN+1);
```

이 프로그램은 전역변수 outN은 $2N-1$ 이고, p, q는 0에서 $2N - 1$ 까지의 인덱스에 의해 지정되는 복소수들을 갖는 배열들이다. 곱해질 두 다항식 p와 q는 차수가 $N - 1$ 이고, 이들 배열들에 있는 다른 계수들은 초기에 영(0)으로 설정된다. 프로시저 eval은 다항식이 단위근들에서 평가될 때 첫 번째 인수로서 주어진 다항식의 계수들을 구해진 값들로 대체한다. 두 번째 인수는 다항식의 차수(계수와 단위근들의 개수보다 하나 작은)를 나타내며 세 번째 인수는 아래에서 설명된다. 위 프로그램 코드는 p와 q의 곱을 계산하여 그 결과를 r에다 넣는다.

이제 eval을 구현해야 할 차례이다. 전부터 보아온 것 처럼, 배열들과 관련하여 수행되는 재귀적 프로그램들을 작성하는것은 아주 까다롭다. 이 알고리즘의 경우, 값들의 저장 관리시에 발생하는 통상적인 문제들을 저장장소를 재치있게 재사용함으로써 피할 수가 있다. 여기

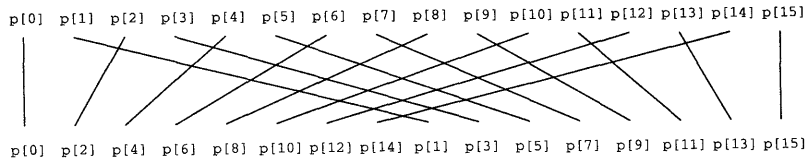


그림 41.1 FFT를 위한 완전 셔플

서 취하려는 방법은 $N + 1$ 개의 계수들의 연속적인(contiguous) 배열을 입력으로 취하고 그 연속적인 배열에다가 $N + 1$ 개의 값을 리턴시켜주는 재귀적 프로시저를 만드는 것이다. 그러나 이 재귀적 스텝은 두 불연속적인 배열들(홀수번째 및 짝수번째 계수들)의 처리를 수반한다. 돌이켜보면, 전 장의 완전 셔플이 여기에서 필요함을 알 수가 있다. 이 입력에 대해, 그림 41.1과 같이 $N = 15$ 의 경우, 완전 셔플을 수행함으로써 한 연속적 부분 배열에서 홀수번째의 계수들을 얻을 수 있고, 한 연속적 부분 배열에서는 짝수번째의 계수들을 얻을 수 있다.

물론, 단위근들의 실제값들이 구현시 필요하다.

$$w'_N = \cos\left(\frac{2\pi j}{N+1}\right) + i \sin\left(\frac{2\pi j}{N+1}\right);$$

임은 잘 알려져 있다. 이 값들은 기존의 삼각 함수들을 사용하여 쉽게 계산된다. 아래 프로그램의 경우, 배열 w 에 (out $N+1$)번째 단위근들이 있다고 가정한다.

이 가정 하에서 구현된 FFT의 구현 프로그램은 다음과 같다:

```
eval (complex p[], int N, int k)
{
    int i, j;
    if (N == 1)
    {
        p0 = p[k]; p1 = p[k+1];
        p[k] = p0+p1; p[k+1] = p0-p1;
    }
    else
    {
        for (i = 0; i <= N/2; i++)
        {
            j = k+2*i;
```

```

        t[i] = p[j]; t[i+1+N/2] = p[j+1];
    }
    for (i = 0; i <= N; i++) p[k+i] = t[i];
    eval (p, N/2, k);
    eval (p, N/2, (k+1+N)/2);
    j = (outN+1)/(N+1);
    for (i = 0; i <= N/2; i++)
    {
        p0 = w[i*j]*p[k+(N/2)+1+i];
        t[i] = p[k+i]+p0;
        t[i+(N/2)+1] = p[k+i]-p0;
    }
    for (i = 0; i <= N; i++) p[k+i] = t[i];
}
}

```

이 프로그램은 차수 N 인 다항식을 앞에서 요약 설명된 재귀적 방법을 사용하여 부분 배열 $p[k], \dots, [k+N]$ 에 위치시킨다. 이 프로그램의 단순성을 위해 $N+1$ 이 2의 승수라고 가정한다.(비록 이 가정을 제거하는 것이 어렵지는 않지만) 만약 $N = 1$ 이면, 1과 -1에서 평가를 위한 계산이 수행된다. 그렇지 않으면, 서플을 먼저하고 두개의 반으로 나뉘어진 부분을 변환하기 위해 자신을 재귀적으로 호출한 다음 그 결과를 앞에서 설명한 것처럼 결합한다. 필요한 단위근들을 얻기 위해, 프로그램은 배열 w 로부터 변수 i 에 의해 결정된 한 구간을 선택한다. 예로서, 만약 $outN$ 이 15이면, 네 번째 단위근들로서 $w[0], w[4], w[8]$ 과 $w[12]$ 가 찾아진다. 이 방식을 이용하면 매번 사용시 마다 단위근들을 재 계산할 필요가 없게 된다.

성질 41.3 차수 N 인 두 개의 다항식은 $2N \lg N + O(N)$ 만큼의 복소수 곱셈을 수행함으로써 곱해질 수 있다.

이 장의 맨 앞에서 언급한 것처럼, FFT의 응용 범위는 여기서 언급된 것 보다 훨씬 더 크다; 그리고 FFT 알고리즘은 아주 다양한 분야에서 많이 사용되고 또한 연구되고 있다. 그럼에도 불구하고, 고수준의 응용분야들에서의 동작의 기본 원리들도 여기서 설명된 다항식 곱셈의 경우와 같다. FFT는 “분할-정복”알고리즘의 설계 패러다임이 실질적인 계산상의 절감을 가져올 수 있는 분야 중의 한 예이다.

연습문제

1. 근 p_0, p_1, \dots, p_{N-1} 과 q_0, q_1, \dots, q_{N-1} 을 갖는 두 다항식 $p(x)$ 와 $q(x)$ 를 곱해 주는 단순한 평가-곱셈-보간 알고리즘을 어떻게 향상시킬 수 있는가?
2. 차수 N 의 다항식이 N^2 동작보다 적게 수행하여 “평가”될 수 있는 N 개의 실수의 집합을 찾아라.
3. 차수 N 의 다항식이 N^2 동작보다 적게 수행하여 “보간”될 수 있는 N 개의 실수의 집합을 찾아라.
4. $M > N$ 일 때 w_N^M 의 값은?
5. FFT를 이용하여 스파이스 다항식을 곱셈할 만한 가치가 있는가?
6. FFT의 구현은, 36장의 다항식 곱셈 프로시저에서 mult를 세 번 호출한 것과 마찬가지로, eval을 세 번 호출한다. 왜 FFT구현 방식이 더 효율적인가?
7. 네 번의 정수 곱셈 동작보다 적게 수행하여 두 복소수를 곱하는 방법을 구하라.
8. 만약 완전 셔플로서 저장 장소 관리의 문제를 없애지 않았다면, FFT에 의해 사용되는 저장 장소는 얼마나 많을까?
9. 왜 완전 셔플같은 기법이 36장의 다항식 곱셈 프로시저에 있는 동적으로 선언된 배열들로 인한 문제들을 피하는데는 사용될 수 없을까?
10. 차수 N 의 다항식(반드시 2의 승수일 필요가 없는)을 차수 M 인 다항식과 곱해 주는 효율적인 프로그램을 작성하라.

42 장

동적 프로그래밍

“큰 문제를 해결하려면 그 문제를 독립적으로 해결 될 수 있는 작은 문제들로 나누라”는 분할-정복 원리는 우리가 배워 온 많은 알고리즘들의 설계를 인도해 왔다. 동적 프로그래밍(dynamic programming)에서는 이 원리가 전적으로 적용된다: 어떤 작은 문제들을 해결해야 하는지를 정확히 모르면, 그들 모두를 해결한 다음 그 답들을 다음에 큰 문제들을 해결하는데 이용할 수 있도록 저장해 둔다. 이 방식은 과학적 경영 분석(operations research)분야의 경우 널리 이용된다. 여기서, “프로그래밍”은 문제의 제약조건(constraints)들을 공식화하여 이 방식이 적용 가능하도록 해주는 과정을 뜻한다. 이러한 프로그래밍은 여기서 더 자세히 다룰 수 없는 기교들이므로 몇 가지 예만 들기로 한다.(우리가 관심을 갖는 “프로그래밍”은 해결책들을 찾기 위한 C++ 프로그램들의 작성을 뜻한다)

우리는 동적 프로그래밍의 테두리 안에 속할 수 있는 몇개의 알고리즘들을 이미 봤다. 예로서, 한 그래프의 추이적 폐포(transitive closure)를 찾는 Warshall 알고리즘과 가중치 그래프에서 모든 최단 경로들을 찾는 Floyd 알고리즘(둘 다 32장에 있다)은 둘 다 정점들을 하나씩 하나씩 처리해 가며 작업을 하는데, 현재 처리 중인 정점에 대한 한 큰 문제에 속하는 작은 부분 문제들은 이 전에 이미 처리한 모든 정점들에 대한 해결책들을 이용하여 해결한다.

동적 프로그램의 적용시 발생할 수 있는 두 가지 어려운 점은 다음과 같다. 첫째, 큰 문제의 해결책을 구하기 위해 작은 문제들에 대한 해결책들을 결합시키는 것이 항상 가능하지는 않다는 점이다. 둘째, 해결해야 할 작은 문제들의 수가 너무 많은 경우이다. 어느 누구도 어떤 문제들이 동적 프로그래밍으로 효율적으로 해결될 수 있는지를 정확하게 규정하지 못했다; 동적 프로그램을 적용할 수 없어 보이는 많은 “어려운” 문제들이 있으며 또한 동적 프

그램을 적용하면 기존의 표준형 알고리즘들의 경우보다 효율적이지 못한 많은 “쉬운” 문제들도 있다.

이 장에서는 동적 프로그래밍이 아주 효과적일 수 있는 문제들에 대한 몇 가지 예를 들기로 한다. 이러한 문제들은 어떤 일을 해결하기 위한 “최선책”을 찾는 일을 수반하며, 한 작은 부분 문제를 해결하기 위한 최선책을 찾는 데에 따른 어떤 결정도 그 부분 문제가 한 큰 문제의 일부가 될 때 잘 된 결정이 되게 하는 일반적 성질을 갖는다.

배낭 문제(Knapsack Problem)

한 금고털이 도둑이 금고 안에 크기와 그 값이 다른 N 개 유형의 물건들이 있음을 알았다. 그러나 그 도둑은 물건을 가지고 가는데 이용할 용량이 M 인 한 개의 작은 배낭밖에 가지고 있지 않다. 배낭 문제(knapsack problem)는 가지고 가는 물건들의 총 값어치가 최대가 되도록 물건들의 조합을 선택하는 것이다.

예로서, 배낭의 용량이 17이고 금고는, 그림 42.1에 있는 것과 같은, 각기 크기와 값이 다른 물건들을 가지고 있다 하자.(여기서도 예제에 있는 물건들을 한 자리 문자로 명명하고 프로그램에서는 이들 물건들의 인덱싱을 위해 정수를 사용한다) 이때 도둑이 5개의 A를 택하면 총 값어치가 20이 되고, 한 개의 D와 한 개의 E를 택하면 총 값어치가 24가 된다. 물론 다른 물건들의 조합을 택할 수도 있다. 그러면 어떤 조합을 택해야 그 총 값어치가 최대가 될 수 있을까?

배낭 문제에 대한 해결책이 중요한 많은 상업적인 문제들이 있다. 예로서, 선적 회사는 선적될 물건들을 갖고 있는 트럭이나 수송기에 선적하기 위한 최선책을 알기를 원할 수도 있

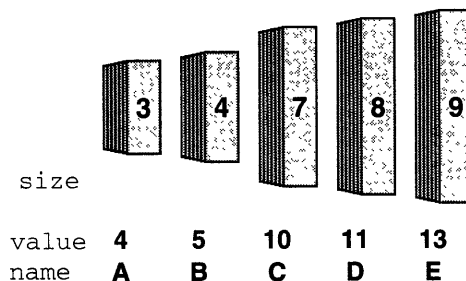


그림 42.1 배낭 문제의 한 예

다. 이와 같은 분야들에는, 많은 변형된 문제들이 또한 있을 수 있다: 예를 들면, 각 물건들의 숫자가 한정되어 있을 수가 있다. 이같이 변형된 많은 문제들은 또한 위에서 언급한 기본 문제를 풀기 위해 살펴보려고 하는 것과 동일한 방식으로 처리 될 수 있다.

배낭 문제에 대한 동적 프로그래밍 해결 방법에서는, 용량 M 까지의 모든 배낭 크기들에 대해 최선의 조합을 계산한다. 이러한 계산은 아래 프로그램에서처럼 적절한 순서로 일들을 처리함으로써 매우 효율적으로 수행될 수 있음이 밝혀졌다:

```
for (j = 1; j <= N; j++)
{
    for (i = 1; i <= M; i++)
        if (i >= size[j])
            if (cost[i] < cost[i - size[j]] + val[j])
            {
                cost[i] = cost[i - size[j]] + val[j];
                best[i] = j;
            }
}
```

이 프로그램에서, $cost[i]$ 는 용량이 i 인 배낭을 갖고 얻을 수 있는 최상의 값이고 $best[i]$ 는 이 최상의 값을 얻기 위해 배낭에 추가 될 수 있는 마지막 물건이다.(아래 기술된 것처럼, 이 $best[i]$ 는 배낭의 내용을 복구하는데 이용된다)

먼저 A형의 물건들만이 택해질 때의 모든 배낭 크기들에 대한 최고치를 구하고 그 다음에는 A형과 B형의 물건들만이 택해질 때의 모든 배낭 크기들에 대한 최고치를 구한다. 이런 식으로 계속하여 모든 가능한 조합에 대해 최고치를 구한다. 결국, $cost[i]$ 에 대해 간단한 계산을 함으로써 해가 얻어진다. 물건 j 가 배낭에 놓여지게 됐다고 하자: 이 경우 얻어질 수 있는 최고치는(물건 j 에 대한) $val[j] +$ (배낭의 나머지 부분을 채우는) $cost[i-size[j]]$ 일 것이다. 만약 이 값이 j 없이 얻어질 수 있는 최고치 보다 크면, $cost[i]$ 와 $best[i]$ 를 수정하고, 아니면 그대로 둔다. 이 방식으로 문제를 풀 수 있음을 단순한 귀납법적 증명을 통해 보일 수 있다.

그림 42.2는 우리의 예제에 대한 계산 과정을 추적한 것이다. 첫($j=1$) 두 줄은 A형 물건들만으로써 얻어질 수 있는 최고치를($cost$ 와 $best$ 배열의 내용을) 보여주고, 그 다음($j=2$) 두 줄은 A형과 B형의 물건들만을 가지고 얻어질 수 있는 최고치를 보여준다. 이 경우, 배낭크

k	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1															
cost[k]	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
best[k]	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2															
cost[k]	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
best[k]	A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3															
cost[k]	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
best[k]	A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4															
cost[k]	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
best[k]	A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=5															
cost[k]	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
best[k]	A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

그림 42.2 배낭 문제의 해

기 17을 갖고 얻어질 수 있는 최고치는 24이다. 이 계산 과정에서, “A형, B형과 C형의 물건들만 갖고 크기 16인 배낭을 채울 때 얻어지는 최고치는 22이다” 처럼 많은 작은 부분 문제들이 해결된다.

최적의 배낭 내용은 best 배열을 이용하여 계산할 수 있다. 정의에 따라, best[M]이 추가되고 배낭의 남은 내용물은 크기가 $M - \text{size}[\text{best}[M]]$ 인 배낭의 내용물과 같게 된다. 따라서, $[M - \text{size}[\text{best}[M]]]$ 이 포함된다 등등... 예로서 $\text{best}[17] = C$ 이면, 나머지 배낭 크기는 10이 된다; 이때 C를 하나 더 선택하면, 나머지 배낭 크기는 3이 되고, A를 하나 더 선택하게 되면 배낭이 다 차게 된다.

성질 42.1 배낭 문제에 대한 동적 프로그래밍 해결 방법은 NM에 비례한 시간이 걸린다.

이 성질은 프로그램 코드에서 쉽게 알 수가 있다. □

이처럼 배낭 문제는 용량 M 이 크지 않으면 쉽게 해결된다. 그러나, 배낭의 용량이 크면, 수행 시간이 용납하기가 어렵게 크다. 게다가, 간과할 수 없는 중요한 것은 이 방법이 M 과

물건 크기와 그 값어치들이 (예로서) 정수 값이 아니라 실수 값이라면 전혀 동작하지 않는다는 점이다. 이 점은 사소한 것이 아니고 근본적으로 어려운 것이다. 이에 대한 어떤 좋은 해결책도 알려진 바가 없다. 45장에서 보게 되겠지만, 많은 사람들이 이에 대한 좋은 해결책이 없다고 믿고 있다. 이 문제를 검토해 보기 위해, 혹자는 값어치가 모두 1이고, j 번째 물건의 크기는 \sqrt{j} 이고 M 은 $N/2$ 인 경우에 대해 해를 구해 보고자 할 수도 있을 것이다. 그러나, 용량 M 과 물건의 크기와 값어치가 모두 정수라면, 최적 결정은 한번 내려지면 변경될 필요가 없다는 근본적인 원리를 따를 수 있다. 만약 첫 j 개의 물건들을 임의의 크기의 배낭에 넣는 최선책을 안다면, 다음에 넣어질 물건들이 무엇이든지에 관계없이, 이들 문제들을 재검사할 필요가 없다. 이 일반적 원리가 적용 가능한 경우에는, 언제나 동적 프로그래밍의 적용이 가능하다. 이 경우에는 정수 값들이 정확한 “최적” 결정을 내리게 해주므로 동적 프로그래밍의 적용이 가능하다.

이 알고리즘에서는, 만약 M 이 크지 않다면 이전의 최적 결정들에 대해 적은 양의 정보만이 저장될 필요가 있다. 이런 식으로, 다른 종류의 동적 프로그래밍 응용 분야들은 아주 다른 요구 사항들을 가지고 있다; 아래에서 다른 예제들을 살펴보기로 한다.

행렬 연속곱셈(Matrix Chain Product)

동적 프로그래밍의 한 고전적 응용 분야는 일련의 크기가 다른 행렬들을 곱하는데 필요한 계산의 양을 최소화하는 문제에서 찾아진다. 이같은 방법들은 행렬 조작을 많이 갖는 분야에서는 반드시 고려되어야 할 사항이다.

다음과 같은 여섯 개의 행렬이 서로 곱해져야 한다고 하자:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{12} \\ c_{13} \end{pmatrix} (d_{11} \ d_{12}) \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix}$$

물론, 곱셈이 행해지려면 한 행렬의 열의 수가 다음 행렬의 행의 수와 같아야 한다. 그러나 수반되는 곱셈의 총 수는 행렬들이 곱해지는 순서에 따라 좌우된다. 예로서, 우리는 곱셈을 좌에서 우로 진행시킬 수 있다: A와 B를 곱하면 24번의 스칼라 곱셈을 한 후에 4행-3열의 행렬이 구해진다. 이 것을 C와 곱하면 12번의 스칼라 곱셈을 한 후 4행-1열의 행렬이 구

해진다. 이 결과를 또 D와 곱하면 8번의 스칼라 곱셈을 한 후에 4행-2열의 행렬이 구해진다. 이런 식으로 계속하여, 총 84번의 스칼라 곱셈을 해야 그 결과로서 4행-3열의 행렬이 얻어진다. 그러나 곱셈을 우에서 좌로 진행시키면, 동일한 결과를 단지 69번의 스칼라 곱셈을 하여 얻을 수 있다.

물론 많은 다른 행렬 순서 상에서 곱셈을 수행할 수 있다. 곱셈의 순서는 괄호를 이용하여 표현될 수 있다: 예로서, 좌에서 우로의 진행 순서는 (((((AB)C)D)E)F)이고 우에서 좌로의 진행 순서는 (A(B(C(D(EF))))))이다. 어떤 식으로 괄호를 해도 그 괄호가 규칙에 맞으면 올바른 결과를 생성할 것이다. 그러나 어떤 괄호식이 가장 작은 횟수의 스칼라 곱셈을 하게 하는가?

곱해질 행렬들이 큰 경우에는 상당한 스칼라 곱셈 횟수를 줄일 수 있다: 예로서, 위 예제의 행렬 B, C, F 각각이 3행이나 3열이 아니라 300행이나 300열을 가진다면, 좌에서 우로의 진행 순서는 6024번의 스칼라 곱셈을 필요로 하는 반면, 우에서 좌로의 진행 순서는 274,200번의 스칼라 곱셈을 필요로 한다.(이 계산들에서, 표준형 행렬 곱셈이 이용된다고 가정한다) Strassen 방법이나 다른 방법을 이용하면, 큰 행렬의 경우, 어느 정도는 곱셈 횟수를 줄일 수 있다. 그러나 곱셈의 진행 순서는 동일하게 적용된다. 따라서, p행-q열의 행렬을 q행-r열의 행렬과 곱하면 p행-r열의 행렬이 만들어진다. 이 행렬의 각 원소는 q번의 곱셈을 통해 만들어졌으므로 총 pqr번의 스칼라 곱셈이 행해질 것이다.

일반화하여, 아래와 같은 N 개의 행렬들을 곱한다고 하자:

$$M_1 M_2 M_3 \cdots M_N$$

여기서 M_i 는 $1 \leq i \leq N$ 인 i 에 대해, r_i 행과 r_{i+1} 열을 가져야 하는 제약 조건을 만족시킨다. 우리의 목표는 총 스칼라 곱셈 수를 최소화 해주는 행렬 곱셈 순서를 찾는 것이다. 모든 가능한 순서들을 시험해 봄은 확실히 비현실적이다.(이 순서들의 개수는 “ N 개의 변수를 괄호에 넣는 방법의 수는 약 $4^{N-1}/N\sqrt{\pi N}$ 이다”라는, Catalan수라고 불리는, 잘 연구된 조합 함수이다) 그러나 좋은 해결책을 찾기 위해 좀더 노력해 볼만한 가치는 충분히 있다. 왜냐하면 N 은 일반적으로 행해져야 하는 곱셈의 횟수에 비해 아주 작기 때문이다.

전과 마찬가지로, 이 문제에 대한 동적 프로그래밍 해결책은 “밑에서 위로” 진행된다; 즉, 재 계산을 피하기 위해 작은 부분 문제들에 대해 얻어진 답들을 저장한다. 먼저, M_1 과 M_2 , 와 M_3, \dots, M_{N-1} 과 M_N 곱하는데는 단지 한 가지 방법만이 있다; 이들의 곱셈 비용을 기록해 놓는다. 그 다음에는, 연속된 세 행렬을 곱하기 위한 최선책을, 이제까지 계산된 모든 정보를

이용하여, 계산한다. 예로서 $M_1M_2M_3$ 을 곱하는 최선책은, 우선 M_1M_2 의 비용을 저장한 정보에서 찾고 나서 거기에다가 M_3 을 곱할 때 생기는 비용을 더한다. 이 비용과 M_2M_3 의 곱셈 후 M_1 을 곱할 때의 비용을 비교하여 작은 비용을 갖는 것을 저장한다. 이런 절차를 모든 연속되는 세 행렬 곱셈에 대해 행한다. 같은 방법으로, 연속되는 네 행렬 곱셈에 대한 최선책을 찾는다. 이렇게 계속하면, 결국에는 모든 행렬들을 곱할 때의 최선책이 찾아진다.

이를 프로그램으로 작성하면 다음과 같다:

```
for (i = 1; i <= N; i++)
  for (j = i+1; j <= N; j++) cost[i][j] = INT_MAX;
for (i = 1; i <= N; i++) cost[i][i] = 0;
for (j = 1; j < N; j++)
  for (i = 1; i <= N-j; i++)
    for (k = i+1; k <= i+j; k++)
      {
        t = cost[i][k-1] + cost[k][i+j] + r[i]*r[k]*r[i+j+1];
        if (t < cost[i][i+j])
          { cost[i][i+j] = t; best[i][i+j] = k; }
      }
```

$1 \leq j \leq N-1$ 에 대해, 행렬 $M_iM_{i+1}\cdots M_{i+j}$ 를 계산하는 최소 비용을 $1 \leq i \leq N-j$ 와 i 와 $i+j$ 사이의 모든 k 에 대해, $M_iM_{i+1}\cdots M_{k-1}$ 과 $M_kM_{k+1}\cdots M_{i+j}$ 의 계산 비용을 구하고, 거기에 이 두 결과를 곱하는 비용을 더함으로써 찾는다. 우리는 항상 한 그룹을 두 개의 작은 그룹으로 나누므로, 이 두 그룹 각각에 대한 최소비용은 이미 저장된 결과에서(재계산하지 않고) 그냥 얻을 수 있다. 특히, $\text{cost}[1][r]$ 은 $M_1M_{1+1}\cdots M_r$ 을 계산할 때의 최소비용이다. 위 프로그램의 첫 번째 그룹의 비용은 $\text{cost}[i][k-1]$ 이고 두 번째 그룹의 비용은 $\text{cost}[k][i+j]$ 이다. 마지막 곱셈비용은 쉽게 구해진다: $M_iM_{i+1}\cdots M_{k-1}$ 은 r_i 행- r_{k-1} 열의 행렬이고 $M_kM_{k+1}\cdots M_{i+j}$ 는 r_k 행- r_{i+j+1} 열의 행렬이므로, 이 둘을 곱하는 비용은 $r_i r_k r_{i+j+1}$ 이다. 이런식으로, 위 프로그램은, $1 \leq i \leq N-j$ 에 대해, j 를 1에서 $N-1$ 까지 증가시켜 가면서 $\text{cost}[i][i+j]$ 를 계산한다. $j = N-1$ (그리고 $i = 1$)에 이르면, 우리가 바라던 $M_1M_2\cdots M_N$ 계산의 최소비용이 찾아진다.

우리는 실제 곱셈 순서가 정해졌을 때, 나중에 곱셈 순서를 알기 위해, 별도의 배열 best 에 있는 결정들을 계속 유지시킬 필요가 있다. 아래 프로그램은 위 프로그램에서 계산된 cost 와 best 배열에서 최적의 괄호 식을 구해 내는 과정을 구현한 것이다:

```

order (int i, int j)
{
    if (i == j) cout << name(i); else
    {
        cout << "(";
        order (i, best[i][j]-1); order (best[i][j], j);
        cout << ")";
    }
}

```

그림 42.3은 위에 주어진 예제 문제에 대한 이 프로그램들의 진행을 추적하는데 이용될 수 있는 도표이다. 이 도표는 행렬들의 리스트에서 각 부분서(subsequence)에 대한 총 비용과 최적의 “최종” 곱셈을 보여준다. 예로서, 행 A와 열 F의 엔트리는 A에서 F까지의 행렬들을 곱하는데 36번의 스칼라 곱셈이 필요하며 이 결과는 A에서 C를 최적의 방법으로 곱하고 D에서 F를 최적의 방법으로 곱한 다음 이 결과로서 나온 두 행렬을 곱함으로써 얻어진다는 것을 알려준다. 단지 D만이 실제로 best 배열에 있다. 완전한 최적 분할들이 다이어그램에 나타나 있다. 최적의 방법으로 어떻게 A에서 C까지를 곱하는지를 찾기 위해, 행 A와 열 C를 보면 된다. 예로서, 계산된 괄호식은 ((A(BC))((DE)F))인데, 이것은 앞에서 언급된 것처럼, 단지 36번의 스칼라 곱셈만을 필요로 한다. B, C, 와 F의 행 또는 열이 3에서 300으로 바뀌었을 때의 경우에도, 같은 괄호식이 최적의 경우이며 총 2412번의 스칼라 곱셈이 요구된다.

	B	C	D	E	F
A	24 [A] [B]	14 [A] [BC]	22 [ABC] [D]	26 [ABC] [DE]	36 [ABC] [DEF]
B		6 [B] [C]	10 [BC] [D]	14 [BC] [DE]	22 [BC] [DEF]
C			6 [C] [D]	10 [C] [DE]	19 [C] [DEF]
D				4 [D] [E]	10 [DE] [F]
E					12 [E] [F]

그림 42.3 행렬 연속곱셈의 해

성질 42.2 동적 프로그래밍은 행렬 연속곱셈 문제를 N^3 에 비례하는 시간 내에 그리고 N^2 에 비례하는 공간에서 처리한다.

이 성질 또한 프로그램 코드를 봄으로써 바로 알 수가 있다. 특히, 필요 공간의 요구 조건은 배낭 문제에서 우리가 사용한 공간 보다 실제적으로 아주 크다. 그러나 최적의 경우를 찾기 위한 시간과 공간의 요구 조건들은 절약된 곱셈의 수에 비교하면 무시될 정도의 것이다. □

최적 검색 트리들(Optimal Binary Search Tree)

많은 검색의 응용 분야에서, 검색 키(key)들의 발생 빈도가 아주 크게 다를 수 있음이 알려져있다. 예로서, 영문 교재에서 단어의 철자를 조사하는 프로그램은 “dynamic”이나 “programming” 같은 단어들 보다 훨씬 더 “and”나 “the” 같은 단어들과 마주치는 듯 싶다. 유사하게, C++컴파일러는 “goto”나 “main”같은 키워드(keyword)들 보다는 “if”나 “for” 같은 키워드들을 더 자주 보게 된다. 만약 이진-트리 검색이 사용된다면, 트리의 상부에 가장 빈번하게 찾아지는 키들이 있는 것이 좋을 것이다. 키들을 어떻게 트리에 배열해야 검색 비용이 최소화 될 수 있는지를 알기 위해 동적 프로그래밍 알고리즘이 이용될 수 있다.

그림 42.4에 있는 이진 검색 트리의 각 노드는 액세스 빈도에 비례하는 정수 값으로 레이블되어 있다. 즉, 이 트리에서 발생한 총 18번의 검색 중에서, 4번은 A, 2번은 B, 1번은 C 등으로 발생할 것으로 기대된다. A에 대한 4번의 각각의 검색은 두 노드 액세스(루트에서부터 시작하여)를 필요로 하고, B에 대한 2번의 각각의 검색은 세 노드 액세스를 필요로 하고, 등등... 우리는 단순히 각 노드의 발생 빈도와 루트에서부터의 거리를 곱하고 이들을 더함으로써 트리의 “비용”의 추정치를 계산할 수 있다. 이것이 트리의 가중치 내부-경로-길이(weighted internal path length of the tree)이다. 그림 42.4의 트리에서의 가중치 내부-경로-길이는 $4 * 2 + 2 * 3 + 1 * 1 + 3 * 3 + 5 * 4 + 2 * 2 + 1 * 3 = 51$ 이다. 우리는 주어진 키와 빈도를 갖고 있는 모든 트리들 중에서 최소 내부-경로-길이를 갖는 이진 검색 트리를 찾고자 한다.

이 문제는 Huffman 코드화를 배울 때 보았던(22장에서) 가중치 외부-경로-길이를 최소화 하는 문제와 유사하다. 그러나 Huffman 코드에서는, 키들의 순서를 유지할 필요가 없다; 이진 검색 트리에서는 루트의 좌측에 있는 모든 노드들은 작은 키들을 가져야 하는 등의 성질이 보존되어야 한다. 이 요구사항으로 인해, 이 문제는 앞에서 다룬 행렬 연속 곱셈 문제와 아주 유사해진다: 거의 같은 프로그램이 이용될 수 있다.

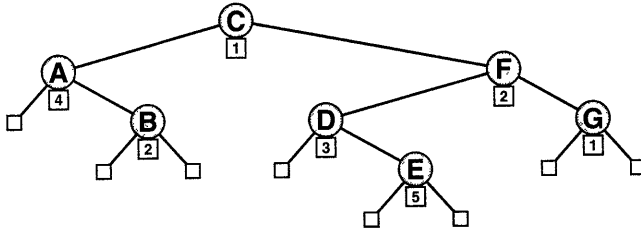


그림 42.4 발생빈도 값을 갖는 이진 검색 트리

특히, $K_1 < K_2 < \dots < K_N$ 인 검색키의 집합과 해당 빈도 r_0, r_1, \dots, r_N 이 주어진다고 가정한다. 여기서 r_i 는 키 K_i 의 참조(reference)에 대한 기대빈도이다. 우리는 모든 키들에 대해 이들 빈도들과 루트로부터의 거리 해당 노드를 액세스 하는 비용을 곱한 값들의 합을 최소화 해주는 이진 검색 트리를 찾고자 한다.

이 문제에 대한 동적 프로그래밍 방식은, 1에서 $N - 1$ 까지의 각각의 j 에 대해, 아래 프로그램에서처럼 $i \leq i \leq N - j$ 에 대해 $K_i, K_{i+1}, \dots, K_{i+j}$ 를 포함하는 한 서브트리를 형성하는 최선책을 찾는 것이다:

```

for (i = 1; i <= N; i++)
    for (j = i+1; j <= N+1; j++) cost[i][j] = INT_MAX;
for (i = 1; i <= N; i++) cost[i][i] = f[i];
for (i = 1; i <= N+1; i++) cost[i][i-1] = 0;
for (j = 1; j <= N-1; i++)
    for (i = 1; i <= N-j; i++)
    {
        for (k = i; k <= i+j; k++)
        {
            t = cost[i][k-1] + cost[k+1][i+j];
            if (t < cost[i][i+j])
                { cost[i][i+j] = t; best[i][i+j] = k; }
        }
        for (k = i; k <= i+j; cost[i][i+j] += f[k++]);
    }

```

각 j 에 대해, 각 노드가 루트가 되도록 하고 서브트리들을 계산하는 최선책을 찾기 위해 미리 계산된 값들을 사용하여 계산을 한다. 우리는, i 와 $i + j$ 사이의 각 k 에 대해, 루트에 K_k 를 갖는 $K_i, K_{i+1}, \dots, K_{i+j}$ 를 포함하는 최적의 트리를 찾고자 한다. 이 트리는 좌측 서브트리

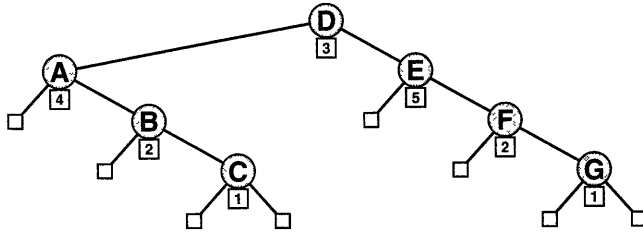


그림 42.5 최적 이진 검색 트리

로서 $K_i, K_{i+1}, \dots, K_{k-1}$ 에 대한 최적트리를, 우측 서브트리로서 $K_{k+1}, K_{k+2}, \dots, K_{i+j}$ 에 대한 최적 트리를 사용함으로써 형성된다. 이 트리의 내부-경로-길이는 두 서브트리들의 내부 경로 길이의 합에다 모든 노드들에 대한 빈도의 합을 더한 것이다.(왜냐하면 이 새 트리상의 각 노드는 루트에서 한 단계 밑에 있기 때문이다)

모든 빈도들의 합이 비용에 더해지므로 최소비용을 찾을 때 이 값이 필요가 없음에 주목하라. 또한, 한 노드가 단지 한 개의 자식을 가질 경우를 처리하기 위해, $\text{cost}[i][i-1]=0$ 을 가져야만 한다.(행렬 연속 곱셈 문제에는 이와 비슷한 경우가 없다)

전과 마찬가지로, 한 짧은 재귀적 프로그램이 프로그램에 의해 계산된 best 배열로부터 실제의 트리를 만들어 내는데 필요하다. 그림 42.4의 트리에 대해 구해진 최적 트리가 그림 42.5에 있다. 이 트리의 가중치 내부-경로-길이는 41이다.

성질 42.3 한 최적 이진-검색 트리를 찾는 동적 프로그래밍 방법은 N^3 에 비례하는 시간이 걸리고 N^2 에 비례하는 공간을 필요로 한다.

이 알고리즘은 크기 N^2 인 행렬을 가지고 동작하며 각 행렬의 엔트리에서 N 에 비례하는 시간을 소모한다. 실제로 이 경우에는, 한 트리의 루트에 대한 최적의 위치는 그 트리보다 약간 작은 트리의 루트에 대한 최적의 위치와 별로 멀리 있지 않으므로 위 프로그램에서 k 에 대해 i 에서 $i+j$ 사이의 모든 값들을 고려할 필요가 없다는 사실을 이용하여, 시간 요구 조건을 N^2 으로 줄일 수 있다. □

시간 및 공간 요구 조건(Time and Space Requirements)

위 예제는 동적 프로그래밍 응용분야들은 작은 부분 문제들에 대한 정보의 양에 따라 시간 및 공간 요구조건들이 크게 다를 수 있음을 알려주고 있다. 최단-경로 알고리즘의 경우에

는, 더 이상의 추가공간이 필요가 없다; 배낭 문제에서는, 배낭의 크기에 비례하여 공간이 필요하다; 다른 문제들의 경우에는, N^2 공간이 필요하다. 이 각각의 문제에 대해, 필요시간은 필요공간 보다 N 배 만큼 크다.

동적 프로그래밍의 가능한 응용범위는 여기 예제들에서 다루어진 것 보다 훨씬 더 넓다. 동적 프로그래밍의 관점에서 보면, 분할-정복 재귀문제들은 작은 경우들에 대한 최소 양의 정보가 계산되고 저장되어야 하는 한 특수한 경우로서 간주할 수 있고, (44장에서 다룰) 완전-검색(exhaustive search)은 작은 경우들에 대한 최대 양의 정보가 계산되고 저장 되어야 하는 한 특수한 경우로서 간주 할 수 있다. 동적 프로그래밍은 이러한 범주에 속하는 문제들을 해결하기 위해 많은 다른 모습으로 나타나는 아주 자연스런 설계 기법이다.

연습문제

1. 배낭 문제에 대해 주어진 예제에서, 물건들이 크기에 따라 정렬된다. 만약 물건들이 임의의 순서로 나타난다면, 알고리즘이 정상적으로 동작할까?
2. 배낭 문제에서 물건의 유형에 따라 물건의 개수가 한정되어 있다 하자. 이 제약 조건이 물건의 개수들을 가지는 한 배열에 의해 정의 될 때, 이 조건하에서 동작할 수 있도록 배낭 프로그램을 수정하라.
3. 만약 물건의 값(어치)들 중의 하나가 음수 값이면, 배낭 프로그램은 무슨 일을 하겠는가?
4. 참 또는 거짓: 만약 한 행렬 연속이 1행- k 열의 행렬과 k 행-1열의 행렬의 곱셈을 포함한다면, 이 곱셈이 마지막에 위치할 경우 최적의 해가 있다. 택해진 답에 대한 이유를 설명하라.
5. 한 연속된 행렬들을 곱하기 위한 두 번째의 최선책을 찾는 프로그램을 작성하라.
6. 본문의 예제에 대해 최적의 이진 검색 트리를 그려라. 단, 모든 빈도를 1씩 증가시켜라.
7. 최적이진 검색 트리를 형성해 주는 프로그램을 작성하라.
8. 임의의 키들과 빈도들의 집합에 대해 최적의 이진 검색 트리를 구했다 하자. 그리고 한 빈도를 1증가시켰다 하자. 이 경우 새로운 최적의 트리를 구하기 위한 프로그램을 작성하라.
9. 왜 배낭 문제를 행렬 연속이나 최적이진 검색 트리 문제들과 같은 방법으로(1에서 M 사이의 k 에 대해, 크기 k 인 배낭에 대해 얻을 수 있는 최대치와 트리 $M-k$ 의 배낭에 대해 얻을 수 있는 최대치의 합을 최소화함으로써) 풀지 않는가?
10. i 에 j 까지 가는 최단 경로로 배열 `path`를 채우는 프로시저 `paths(int i, int j)`를 포함하도록 최단-경로 문제에 대한 프로그램을 확장하라. 이 프로시저는 32장에 주어진 한 프로그램의 한 수정 버전에 의해 만들어진 보조 데이터 구조를 사용하여 호출될 때마다 경로의 길이에 비례하는 시간이 걸려야 한다.

빈 면

43 장

선형 프로그래밍

많은 실제적 문제들은 많은 변화하는 문제들간의 복잡한 상호작용을 수반한다. 이에 대한 한 예가 33장에서 논의된 네트워크 흐름(network flow)문제이다: 여러 파이프들에서의 흐름은 네트워크에 대한 물리적 법칙에 따라야 한다. 또 다른 한 예는 제조공정에서 시간제한, 우선순위 등등에 따라 여러 TASK(작업)들을 스케줄링하는 문제이다. 수반된 상호작용들을 정확히 설명해주고 문제를 보다 단순한 수학문제로 만들어주는 수학적 공식을 개발하는 것이 가능하다. 주어진 실제 문제의 해를 의미하는 수학방정식들의 집합의 해를 유도하는 과정을 수학적 프로그래밍이라 부른다. 42장에서처럼 여기서도 “프로그래밍”이란 용어는 변수들을 선택하고 방정식들을 설정하고 그래서 그 방정식들에 대한 해가 그 문제에 대한 해에 해당하는 과정을 뜻한다. 이 장에서는, 수학적 프로그래밍들의 한 근원적인 변형인 선형 프로그래밍(linear programming)과 이 선형 프로그램들을 푸는 한 효율적인 알고리즘인 심플렉스 기법(simplex method)을 살펴보기로 한다.

선형 프로그래밍과 심플렉스 기법은 근본적으로 중요하다. 왜냐하면 아주 다양한 종류의 중요한 문제들이 선형 프로그램들로 쉽게 공식화되고 심플렉스 기법에 의해 효율적인 해가 얻어질 수 있기 때문이다. 일부 특정 문제들에 대해서는 더 좋은 알고리즘들이 있음이 알려져 있으나, 선형 프로그램으로 문제를 먼저 공식화하고 심플렉스 기법을 사용하여 해를 구하는 과정처럼 널리 응용되는 문제-해결 기법들은 거의 없다. 심플렉스 기법을 위한 라이브러리 루틴은 복잡한 문제들을 해결하기 위한 필수불가결한 도구일수 있다.

선형 프로그래밍에서의 연구는 광범위하게 되어왔고 이에 관련된 모든 문제점들의 완전한 이해는 이 책의 범위를 넘어서는 수학적 지식을 필요로 한다. 반면에, 일부 기본 개념들은 이

해하기 쉽고 실제로 심플렉스 알고리즘은, 앞으로 보게 되겠지만, 구현하기가 그리 어렵지 않다. 41장의 빠른 Fourier 변환에서 처럼, 우리의 의도는 완전하고 실제적인 구현을 하고자 함이 아니라, 알고리즘의 일부 기본성질들과 우리가 배워온 다른 알고리즘들과의 관계를 배우고자 하는 것이다.

선형 프로그래밍(Linear Programming)

수학 프로그램들은 수학방정식(제약조건)들의 집합에 관련된 변수들의 집합과 제약조건들에 따라 최대화되어야 할 변수들을 포함하는 목적함수(objective function)를 가지고 있다. 만약 포함된 모든 방정식들이 단순히 변수들의 선형조합들이라면, 이는 우리가 다루고자하는 선형 프로그래밍이라는 특수한 경우이다.

다음 선형 프로그램은 33장의 네트워크 흐름 문제에 해당된다.

아래 제약조건들을 만족시키면서 $x_{AB} + x_{AD}$ 를 극대화하라:

$$x_{AB} \leq 6 \quad x_{CD} \leq 3$$

$$x_{AC} \leq 8 \quad x_{CE} \leq 3$$

$$x_{BD} \leq 6 \quad x_{DF} \leq 8$$

$$x_{BE} \leq 3 \quad x_{EF} \leq 6$$

$$x_{BD} + x_{BE} = x_{AB}$$

$$x_{CD} + x_{CE} = x_{AC}$$

$$x_{BD} + x_{CD} = x_{DF}$$

$$x_{BE} + x_{CE} = x_{EF}$$

$$x_{AB}, x_{AC}, x_{BD}, x_{BE}, x_{CD}, x_{CE}, x_{DF}, x_{EF} \geq 0.$$

선형 프로그램에는 각각의 파이프내의 흐름에 해당하는 변수가 하나씩 있다. 이들 변수들은 파이프들에 대한 용량 제약 조건들에 해당하는 부등식들과 매 접합부에서의 흐름제약 조건들에 해당하는 등식들의 두 가지 유형의 방정식들이 있다. 예로서, 부등식 $x_{AB} \leq 8$ 은 파이프 AB는 용량이 8임을 말해주고 방정식 $x_{AB} + x_{AC} = x_{AB}$ 는 유출흐름(outflow)은 유입흐름(inflow)과 접합부 B에서 같아야 함을 말해준다. 모든 등식들은 함께 묵시적 제약조건 $x_{AB} +$

$x_{AC} = x_{DF} + x_{EF}$ 을 만드는데, 이 제약조건은 전체 네트워크에 대해 유입흐름은 유출흐름과 같아야함을 말해준다. 물론 모든 흐름은 양수 값이어야 한다는 것이다.

위 선형 프로그램은 네트워크 흐름 문제를 수학적으로 공식화한 것이다. 이 특정 수학적 문제의 해는 네트워크 흐름 문제의 특정 경우의 해이다. 이 예의 요지는 선형 프로그래밍이 이 특정 문제에 대해 더 좋은 알고리즘을 제공할 것이라는 것이 아니라, 선형 프로그래밍은 다양한 문제들에게 적용될 수 있는 아주 일반적인 기법이라는 것이다. 예로서, 만약 네트워크 흐름 문제를 일반화하여 용량들 뿐만 아니라 비용들도 포함시킨다 하더라도, (비록 문제를 직접 풀기에는 아주 어려워질지는 몰라도) 선형 프로그래밍의 공식화는 그다지 어려워 보이지는 않는다.

선형 프로그램들은 표현력이 풍부할 뿐만 아니라, 실제 발생하는 많은 문제들에 대해 아주 효과적인 것으로서 증명된 알고리즘(심플렉스 알고리즘) 또한 존재한다. (네트워크 흐름 같은) 일부 문제들에 대해서는 선형 프로그래밍/심플렉스 보다 더 잘 수행할 수 있고 그 문제들에 대해 지향적인 알고리즘이 있을 수 있다. 그러나(네트워크 흐름 문제의 다양한 확장형태를 포함한) 다른 문제들의 경우에는, 더 좋은 알고리즘이 알려진 바 없다. 더 좋은 알고리즘이 있더라도, 그 알고리즘은 어쩌면 아주 복잡하거나 구현하기 어려울 수 있다. 반면에 선형 프로그램을 개발하는 프로시저와 만들어진 선형 프로그램을 심플렉스 라이브러리 루틴을 가지고 푸는 것은 종종 아주 간단하다. 이러한 범용성은 아주 매력적이어서 이 방법이 널리 사용되게 됐다. 이 방법에 너무 의존할 때의 위험성은 이 방법이 일부 간단한(예로서 이 책에서 다루어진 많은) 문제들에 대해 비효율적인 해를 산출할 수 있다는 것이다.

기하학적 해석 (Geometric Interpretation)

선형 프로그램들은 기하학적으로 표현될 수 있다. 아래의 선형 프로그램은 가시화하기가 쉽다. 왜냐하면 단지 두 변수만이 있기 때문이다:

아래 제약조건들을 만족시키면서 $x_1 + x_2$ 를 극대화하라

$$-x_1 + x_2 \leq 5,$$

$$x_1 + 4x_2 \leq 45,$$

$$2x_1 + x_2 \leq 27,$$

$$3x_1 - 4x_2 \leq 24,$$

$$x_1, x_2 \geq 0$$

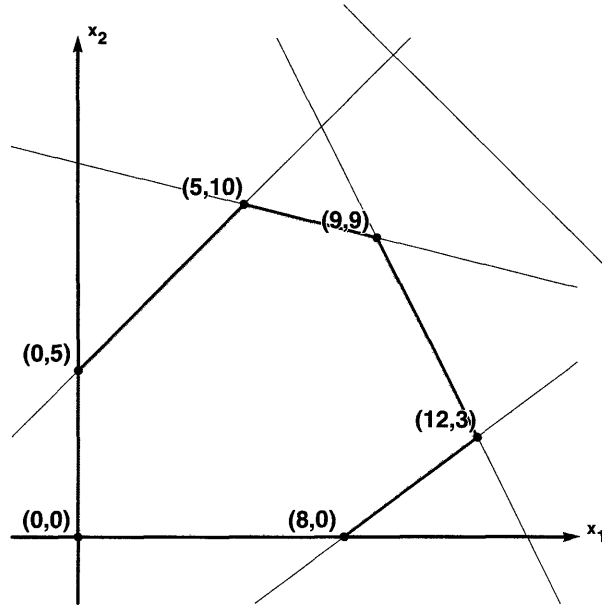


그림 43.1 이차원 심플렉스

이 선형 프로그램은 그림 43.1에 그려진 기하학적 표현에 해당한다. 각 부등식은 선형 프로그램에 대한 해가 있어야하는 반평면(halfplane)을 정의한다. 예로서 $x_1 \geq 0$ 은 해가 x_2 축의 오른쪽에 있어야 한다는 것을 뜻하며 $-x_1+x_2 \leq 5$ 는 해가 $((0, 5)$ 와 $(5, 10)$ 을 통과하는 $-x_1+x_2 = 5$ 를 나타내는 선의 오른쪽 아래부분에 있어야 함을 뜻한다. 선형 프로그램의 해는 모든 이러한 제약조건들을 만족시켜야 한다. 그러므로, 모든 이러한 반평면들의 교집합으로써 정의되는 영역(그림의 음영진 부분)은 모든 가능한 해들의 집합이다. 선형 프로그램을 찾기 위해서는, 목적함수를 극대화시키는 이 영역내의 점을 찾아야한다.

반평면들의 교집합으로서 정의되는 영역은 항상 볼록(convex)하다.(이 사실은 이미 우리가 25장의 볼록 외곽의 정의들 중의 하나에서 배운 것이다) 이 심플렉스라 불리는 볼록 영역은 알고리즘이 목적함수를 극대화해주는 선형 프로그램에 대해 해를 찾기 위한 토대를 형성한다.

알고리즘에 의해 이용되는 이 심플렉스의 한 근본적인 성질은 목적함수가 심플렉스의 정점들 중의 한 점에서 극대화된다는 것이다; 그러므로 이 정점들에 대해서만 검사하면 된다. (내부의 점들에 대해서는 검사할 필요가 없다) 왜 이렇게 되는지를 알기 위해, 그림 43.1의 오른쪽 위 부분에 있는 음영진 선에 대해 생각해보자, 이 선은 목적 함수에 해당한다. 목적함수는 해당 기울기(이 경우-1)를 갖는 선과 미지의 점(unknown position)을 정의하는 것으로

간주될 수 있다. 우리는 이 선이 무한에서부터 안으로 이동해가면서 심플렉스와 마주치는 점에 관심이 있다. 이 점이 선형 프로그램에 대한 해이다. 이점은 심플렉스상에 있으므로 모든 부등식들을 만족시키며 또한 어떤 더 큰 값을 갖는 점들과는 마주치지 않으므로 목적함수를 극대화시킨다. 우리 예제의 경우에는, 목적함수를 나타내는 선이 심플렉스와 점 (9, 9)에서 만나는데, 이점은 목적함수를 극대화시켜 18이 되게 한다.

다른 목적함수들은 다른 기울기를 가진 선들로서 나타난다. 그러나 극대치는 항상 심플렉스상의 정점들 중의 하나에 있다. 우리가 아래에서 살펴볼 알고리즘은 극소치를 찾기 위해 정점에서 정점으로 이동해 가는 한 체계적인 방법이다. 이차원의 경우에는 어떻게 할 지에 대한 선택의 여지가 별로 없다. 그러나, 앞으로 알게 되겠지만, 심플렉스는 많은 변수들이 관련될 때에는 아주 복잡한 객체가 된다.

또한 그림 43.1에서 비선형 함수들을 갖는 수학적 프로그램은 왜 처리하기가 아주 어려운지를 음미해 볼 수가 있다. 예로서, 만약 목적함수가 비선형이면, 이 함수는 심플렉스의(정점이 아니라) 선분중의 하나와 마주치는 곡선일수 있다. 만약 부등식들이 또한 비선형이라면, 이 심플렉스에 해당하는 아주 복잡한 기하학적 형태가 생길 수 있다.

기하학적으로 생각해보면 다양한 비정상적 상황들이 발생함이 확실하다. 예를 들어, 위 예제의 선형 프로그램에 부등식 $x_1 \leq 13$ 을 추가한다고 하자. 이 경우 반평면들의 교집합은 공집합이 됨을 그림 43.1에서 확실히 알 수 있다. 이같은 선형 프로그램을 실행 불가능하다. (infeasible)고 한다: 목적함수의 극대화는 차치하고서도, 부등식들을 만족시키는 점조차 없다. 이 와는 달리, 부등식 $x_1 \leq 13$ 는 잉여(redundant)식이다: 심플렉스의 전 부분이 이 반평면내에 포함되므로 이 부등식은 심플렉스 내에 표현되지 않는다. 이러한 잉여 부등식들은 해에 전혀 영향을 미치지 않는다. 그러나 이 식들은 해를 찾기 위한 검색 시에는 다루어져야 한다.

보다 심각한 한 가지 문제를 심플렉스가 개(비한정적)영역(open region)일수 있다는 점이다. 이 경우, 해가 잘 정의가 되지 않을 수 있다. 위 예제의 경우, 두 번째와 세 번째 부등식이 제거되면 이러한 경우가 될 수 있다. 일부 목적함수들의 경우에는 비록 심플렉스가 개영역이더라도, 해가 잘 정의 될 수 가 있다. 그러나, 이 해를 찾는 알고리즘은 이 개영역을 피해가기 위해 상당한 어려움을 겪을 수 있다.

이러한 문제들은 두 변수들과 몇 개의 부등식이 있는 경우에는 아주 쉽게 보이지만, 많은 변수들과 부등식들을 갖는 일반적인 문제에 대해서는 아주 명백치 않게 된다는 점이 강조되어야 한다.

기하학적으로 볼 때도 많은 변수들이 있는 경우에는 이를 알 수가 있다. 삼차원의 경우 심플렉스는 방정식들이 부등식들을 등식들로 바꿈에 의해 구해지는 평면들에 의해 정의된 반평면들을 교집합(intersecting)함으로서 만들어진 볼록한 삼차원 물체이다. 예를 들어, 만약 위 선형 프로그램에 부등식 $x_3 \leq 4$ 와 $x_3 \geq 0$ 을 추가하면, 심플렉스는 그림 43.2에 있는 삼차원 물체가 된다.

예제를 더 삼차원적으로 하기 위해, 목적함수를 $x_1+x_2+x_3$ 으로 바꾸기로 한다. 이 함수는 $x_1 = x_2 = x_3$ 인 선에 수직인 한 평면을 정의한다. 만약 이 선을 따라 한 평면을 무한대에서부터 안으로 이동시키면, 심플렉스의 점(9, 9, 4)과 마주치게 되는데 이것이 해이다.(또한, 그림 43.2에서는, 심플렉스상의 정점(0, 0, 0)에서 해까지 가는 정점들의 경로를 볼 수 있다)

n 차원에서는 n 차원 심플렉스를 정의하기 위해 $(n - 1)$ -차원 하이퍼평면들로서 정의된 반평면들을 교집합하고 심플렉스상의 해에 해당하는 점을 찾기(교차하기)위해 $(n - 1)$ 차원 하이퍼평면을 무한대에서부터 안으로 끌고 온다. 위에 언급된 것처럼, 우리는 이해를 돕기 위해 이차원과 삼차원의 경우들로 단순화했다. 그러나 볼록성, 하이퍼들의 교차 같은 위의 사실들의 증명은 이 책의 범위를 다소 넘는 선형대수(linear algebra)를 포함한 도구를 수반한다. 아직, 기하학적 직감은 가치가 있다. 왜냐하면, 이 직감은 고차원 문제들을 풀기 위해 실제로 사용되는 기본방법의 근본적 특성들에 대한 이해를 돕기 때문이다.

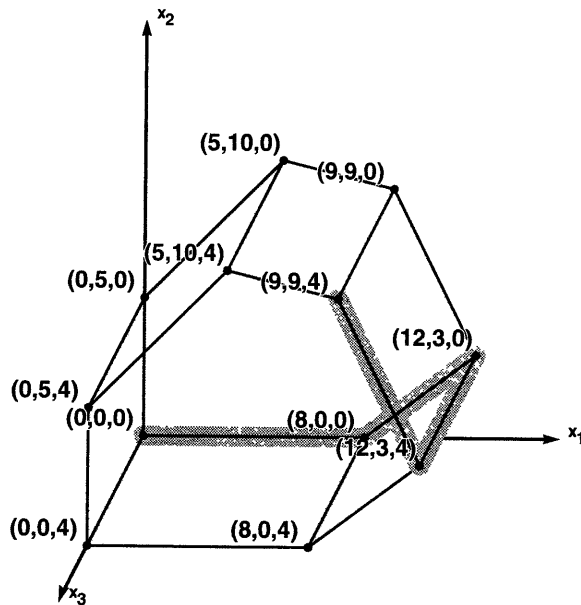


그림 43.2 삼차원 심플렉스

심플렉스 기법(Simplex Method)

심플렉스 기법은 Gauss소거법에서도 사용되는 근본적 방법인 피보팅(pivoting)을 사용하여 선형 프로그램들을 푸는 일반적 방식을 설명하기 위해 일반적으로 이용되는 이름이다. 피보팅이 해를 찾기 위해 심플렉스상의 점에서 점으로 이동하는 기하학적 동작과 자연스럽게 일치함이 밝혀져 있다. 일반적으로 사용되는 여러 알고리즘들은 심플렉스 정점들이 검색되는 순서에 관련하여 필수적인 세부사항들에 있어서 차이점이 있다. 즉, 이 문제에 대해 잘 알려진 “알고리즘”은, 여러 가지 방법들중의 한 방법으로 정제될 수 있는 한 포괄적 방법으로서 보다 정확히 설명될 수 있다. 우리는 전에도 Gauss소거 (37장)이나 Ford-Fulkerson 알고리즘(33장)에서 이런 상황에 직면한 적이 있다.

먼저, 선형 프로그램들은 많은 다른 형태를 가질 수 있음은 확실하다. 예로서, 네트워크 흐름문제에 대한 위의 선형 프로그램은 등식들과 부등식들의 한 혼합형이다. 그러나, 위의 기하학적 예제들은 부등식들 만을 사용한다. 모든 선형 프로그램들은 모든 방정식들이, 변수가 음수가 아님을 나타내주는 각 변수에 대한 부등식을 제외하고는, 등식인 한 표준형으로 나타하도록 하여 다소나마 가능성들의 수를 줄이는 것이 좋다. 이런 표준형의 사용은 상당한 제약인 듯 싶다. 그러나 실제로는 일반적인 선형 프로그램들을 한 표준형으로 변환하는 것은 어렵지가 않다. 아래 선형 프로그램은 그림 43.2에 있는 삼차원 예제에 대한 표준형이다.

아래 제약조건들을 만족시키면서 $-x_1 + x_2 + x_3$ 을 극대화하라:

$$-x_1 + x_2 + y_1 = 5$$

$$x_1 + 4x_2 + y_2 = 45$$

$$2x_1 + x_2 + y_3 = 27$$

$$3x_1 - 4x_2 + y_4 = 24$$

$$x_3 + y_5 = 4$$

$$x_1, x_2, x_3, y_1, y_2, y_3, y_4, y_5 \geq 0.$$

한 변수보다 많은 변수를 갖는 각 부등식은 하나의 새 변수를 추가함으로써 등식으로 변환된다. y 로 나타나는 변수들은 슬랙(slack)변수들이라고 한다. 왜냐하면 이 변수들은 부등식 일 때 생기는 슬랙을 없애주기 때문이다. 단지 한 변수만을 갖는 부등식은 단순히 그 변수형을 바꿈으로서 표준형의 음수가 아닌 조건으로 변환될 수 있다. 예로서 $x_3 \leq -1$ 같은 조건은 x_3 이 나타날 때마다 $-1-x_3'$ 으로 대치함으로써 처리할 수 있다.

이렇게 공식화하면 선형프로그래밍과 연립방정식이 대등하게 된다. 따라서, 모두 양수여야 하는 M 개의 미지의 변수를 갖는 N 개의 방정식이 생기게 된다. 이 경우, 각 방정식당 하나씩 모두 N 개의 슬랙변수가 있음에 주목하라.(왜냐하면 N 개의 부등식이 있었기 때문에) 우리는 $M > N$ 으로 가정한다; $M > N$ 은 방정식들에 대해 많은 해들이 있음을 뜻한다. 따라서, 문제는 목적함수를 극대화해주는 한 해를 찾는 것이다.

우리 예제의 경우, 방정식들에 대한 해가 단순하다: $x_1 = x_2 = x_3 = 0$ 을 취한다. 그리고 등식을 만족시키기 위해 적정 값들을 슬랙 변수들에 할당한다. 이 방식은 옳다. 왜냐하면 $(0,0,0)$ 이 이 예제의 심플렉스상의 한 점이 되기 때문이다. 이 방식은 심플렉스 기법을 설명하는 일반적인 경우일 수는 없다. 그러나, 우리는 당분간 이 방식이 적용될 수 있는 선형 프로그래밍의 경우만을 다루기로 한다. 아직 아주 많은 형태의 선형 프로그램들이 있다: 예로서, 만약 표준형 선형 프로그램에 있는 부등식들의 오른쪽에 있는 모든 숫자들이 양수이고 모든 슬랙 변수들이 양의 계수들을 가진다면(우리의 예제처럼) 모든 원래의 변수들이 0이 되는 해가 반드시 있다. 일반적인 경우는 나중에 다루기로 한다.

$M - N$ 개의 변수들이 영인 해가 주어졌을 때, 피보팅을 사용하여 같은 성질을 갖는 다른 해를 찾을 수 있음이 알려져 있다. 이 피보팅은 근본적으로 Gauss소거시 사용되는 것과 같은 연산 동작이다: 한 원소 $a[p][q]$ 가 방정식들에 의해 정의된 계수들의 행렬에서 선택된다. 그리고 p 번째 행에 한 적당한 스칼라 값을 곱하고, 이 값을 모든 행에 더하여 q 번째 열에서 q 행의 (1인) 엔트리를 제외한 모든 엔트리가 영이 되도록 만든다.

어떻게 피보팅이 선형 프로그램들을 풀 수 있도록 해 주는지를 보기 위해, 위에 주어진 선형 프로그램을 나타내는 행렬을 살펴보자:

$$\begin{pmatrix} -1.00 & -1.00 & -1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ -1.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 5.00 \\ 1.00 & 4.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 45.00 \\ 2.00 & 1.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 27.00 \\ 3.00 & -4.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 24.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

이 $(N + 1) - (M + 1)$ 행의 행렬은 표준형 선형 프로그램의 계수들을 포함한다. 여기서 $(M + 1)$ 번째 열은(Gauss 소거에서처럼) 방정식들의 오른쪽에 있는 숫자들을 가지며 0번째 행은 목적 함수들의 계수들의 부호를 바꾼 값들을 가진다. 0번째 행의 중요성은 아래에서 논의된다; 당분간은 이 행을 다른 모든 행들과 마찬가지로 다룰 것이다.

우리 예제에 대해, 모든 계산을 두 자리 십진수에서 수행하기로 한다. 이렇게 함은, Gauss 소거의 경우에서 처럼, 중요한 계산적 정확성과 누적 오차같은 문제점들을 명백히 무시하는 것이다.

한 해에 해당하는 변수들은 기저(basis)변수라고 불리며 해를 얻기 위해 영으로된 변수들은 기저가 아닌 변수이다. 행렬에서, 기저변수들에 해당하는 열들은 정확히 한 개의 1만 갖고 모두 다른 엔트리들은 영이 된다. 반면에 기저가 아닌 변수들은 영이 아닌 엔트리가 하나보다 많은 열들에 해당한다.

이제 위 행렬을 $p = 4$ 와 $q = 1$ 에 대해 피보팅 하려 한다 하자. 즉, 네 번째 행에 적당값을 곱한 다음 이를 다른 행들 각각에 더하여 4행의 1인 엔트리를 제외한 첫 열의 모든 엔트리를 영으로 만든다. 그러면 다음과 같은 행렬이 만들어진다:

$$\begin{pmatrix} 0.00 & -2.33 & -1.00 & 0.00 & 0.00 & 0.00 & 0.33 & 0.00 & 8.00 \\ 0.00 & -0.33 & 0.00 & 1.00 & 0.00 & 0.00 & 0.33 & 0.00 & 13.00 \\ 0.00 & 5.33 & 0.00 & 0.00 & 1.00 & 0.00 & -0.33 & 0.00 & 37.00 \\ 0.00 & 3.67 & 0.00 & 0.00 & 0.00 & 1.00 & -0.67 & 0.00 & 11.00 \\ 1.00 & -1.33 & 0.00 & 0.00 & 0.00 & 0.00 & 0.33 & 0.00 & 8.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

이 연산동작은 기저에서 7열이 제거되게 하고 그 기저에 1열이 추가되게 한다. 정확히 한 기저열(basis column)이 제거된다. 왜냐하면 행 p 에 1을 가진 기저열은 정확히 하나있기 때문이다.

정의에 따라, 모든 기저가 아닌 변수들을 영으로 만들고 기저에 주어진 단순해를 사용하여 선형 프로그램에 대한 해를 얻을 수 있다. 위 행렬에 해당하는 해에서, x_2 와 x_3 는 영이다. 왜냐하면 이 둘은 기저가 아닌 변수들이기 때문이다. 그리고 $x_1 = 9$ 이다. 따라서 행렬은 심플렉스상의 점 $(8,0,0)$ 에 해당한다.(슬랙 변수들의 값에 대해서는 관심을 갖지 않는다) 행렬의 우측 상부 (행 0, 열 $M + 1$)는 이 점에서의 목적함수 값을 포함한다. 이것은 앞으로 곧 알게 되겠지만 의도적인 것이다.

$p = 3$ 과 $q = 2$ 에 대해 피보팅을 한다고 하자.

$$\begin{pmatrix} 0.00 & 0.00 & -1.00 & 0.00 & 0.00 & 0.64 & -0.09 & 0.00 & 15.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.09 & 0.27 & 0.00 & 14.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & -1.45 & 0.64 & 0.00 & 21.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.27 & -0.18 & 0.00 & 3.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.36 & 0.09 & 0.00 & 12.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

이 연산동작은 기저에서 6열이 제거되게 하고 그 기저에 2열이 추가되게 한다. 기저가 아닌 변수들을 영으로 만들고 기저 변수들에 대해 풀면, 이 행렬은 심플렉스상의 점(12, 3, 0)에 해당함을 알 수 있다. 이 경우, 목적함수는 값 15를 갖게 된다. 목적함수의 값이 증가했음에 주목하라. 이것 또한 곧 알게 되겠지만 의도적인 것이다.

피보팅을 위해 사용할 p 와 q 값을 어떻게 결정할까? 이것이 영번째 행이 추가된 이유이다. 각 기저가 아닌 변수에 대해, 영번째 행은 해당 변수 값이(부호를 역으로 하여) 0에서 1로 변할 때 목적함수가 증가될 수 있는 양을 포함한다.(부호를 역으로 취함으로써 표준피보팅 동작 시 어떤 변경 없이 행0을 유지할 수 있다) q 열을 이용한 피보팅은 해당 변수의 값을 0에서 어떤 양수 값으로 바꾸는 것에 해당한다. 따라서 우리가 행 0에 있는 음수 엔트리 열을 이용하면 목적함수가 증가할 것임을 확신할 수 있다.

어느 행의 양수 엔트리 열 상에서의 피보팅은 목적함수를 증가시킬 수 있었을 것이다. 그러나, 우리는 또한 이 피보팅이 심플렉스상의 한 점에 해당하는 한 행렬을 산출하도록 해야 한다. 여기서의 주된 관심사는 $M + 1$ 열의 엔트리중의 하나는 음수가 될 수도 있다는 것이다. 이문제는, q 열에 있는(영번째 행은 제외한) 양수원소들 중에서, 같은 행에 있는($M + 1$)번째 원소로 나누어질 때 최소치를 만들어주는 원소를 찾음으로써 피할 수 있다. 만약 우리가 이 원소와 피보트를 포함하는 행의 인덱스로서 p 를 택하면, 확실히 목적함수가 증가되고 $M + 1$ 열에 있는 엔트리들 중의 어떤 것도 음수가 될 수 없음이 확실해진다; 이렇게 함으로써, 결과행렬이 심플렉스상의 한 점에 해당하도록 함이 보장된다.

피보트 행을 찾기 위한 이 절차상에는, 두 가지의 잠재적 어려움이 있다. 첫째는, 만약 q 번째 열에 양수 엔트리들이 없다면 어떻게 할 것인가? 이런 경우는 일관성이 없는 경우이다: 영번째 행의 음수 엔트리는 목적함수가 증가될 수 있음을 말해주지만, 증가시킬 방법이 없다. 이런 경우는 심플렉스가 비제한적인 경우에만 발생함이 알려져 있다. 따라서 이 경우 알고리즘은 종료되면서 이 문제를 알릴 수 있다. 보다 미묘한 어려운 문제는(q 열에 양수 엔트리를 갖는) 어느 행의 ($M + 1$)번째 엔트리가 영인 퇴행적 경우(degenerate case)에 발생한다. 이 경우 이 행이 선택되나, 목적함수는 영만큼 증가될 것이다.(즉, 전혀 증가되지 않는다) 이것은 이 자체로는 전혀 문제가 아니다: 문제는 이런 행들이 두 개 있을 때 일어난다. 이 두 행 중에서 하나를 선택하기 위한 특정방식은 자연적으로 사이클링(cycling)-목적함수를 전혀 증가시키지 않는 무한 번의 피보팅-을 유발시킨다. 우리는 이제까지 우리 예제에서의 설명을 명확히 하기 위해 사이클링 같은 어려운 문제점들은 다루지 않았다. 그러나, 이러한 퇴행적 경

우들이 실제로는 아주 발생할 확률이 높음을 강조하고 싶다. 선형 프로그래밍의 일반성은 일반적인 문제의 퇴행적 경우들이 특정문제들의 해에서 일어날 것임을 의미한다.

사이클링을 피하는데 이용할 수 있는 여러 방법들이 있다. 한 가지 방법은 타이(tie)들을 무작위적으로 선택하여 피하는 것이다. 이 방법은 사이클링이 거의 발생하지 않도록(그러나 수학적으로는 가능하다) 해 준다. 또 하나의 사이클링 제거방법이 아래에 설명되어 있다.

우리 예제에서, $q = 3$ (왜냐하면 영번째 행 세 번째 열의 값 -1 때문에)과 $p = 5$ (왜냐하면 1은 세 번째 열 유일한 양수 값이므로)를 갖고 피보팅을 다시 할 수 있다. 이렇게 하여 다음 행렬이 만들어진다:

$$\begin{pmatrix} 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.64 & -0.09 & 1.00 & 19.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.09 & 0.27 & 0.00 & 14.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & -1.45 & 0.64 & 0.00 & 21.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.27 & -0.18 & 0.00 & 3.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.36 & 0.09 & 0.00 & 12.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

이 행렬은 심플렉스상의 점 (12,3,4)에 해당하는데, 이 경우의 목적함수 값은 19이다.

일반적으로, 영번째 행에 여러 개의 음수 엔트리들이 있을 수 있고 이들 중에서 하나를 선택하는 여러 가지의 다른 전략들이 제안되어져 왔다. 우리는 이제까지 가장 널리 이용되는 방법들 중의 하나인 최대-증분법(greatest-increment method)을 이용해 왔다: 항상 영번째 행에서 최소값(절대치로는 최대값)을 갖는 열을 택한다. 이 방법이 반드시 목적함수를 최대 로 증가시키지는 않는다. 왜냐하면 선택된 행 p 에 따라 비율조정(scaling)이 행해져야하기 때문이다. 만약 이 열-선택 정책이, 타이의 발생 시 기저상의 최소 인덱스의 열을 제거하는 행을 이용한, 행-선택 정책과 결합되면 사이클링이 일어날 수 없게 된다.(이 사이클링 제거정책은 R.G. Bland에 의한 것이다) 열 선택을 위한 또 하나의 가능한 방법은 목적함수가 각 열에 대해 증가할 수 있는 양을 계산하여 최대치를 가져다주는 행을 선택하는 것이다. 이 방법은 최하 강하법(steepest-descent method)이다. 또 하나의 방법은 이용가능한 열중에서 무작위적으로 선택하는 것이다.

마지막으로, $p = 2$ 와 $q = 7$ 에서 한번 더 피보팅을 하면, 해가 얻어진다:

$$\begin{pmatrix} 0.00 & 0.00 & 0.00 & 0.00 & 0.14 & 0.43 & 0.00 & 1.00 & 22.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & -0.43 & 0.71 & 0.00 & 0.00 & 5.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.57 & -2.29 & 1.00 & 0.00 & 33.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.29 & -0.14 & 0.00 & 0.00 & 9.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & -0.14 & 0.57 & 0.00 & 0.00 & 9.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

이 행렬은 심플렉스상의 점 (9,9,4)에 해당하는데, 이 행렬을 목적함수가 22가 되도록 극대화시킨다. 이제는 영번째 행의 모든 엔트리들은 음수가 아니므로 어느 피보트도 목적함수를 감소시킬 뿐이다.

위 예제는 선형 프로그램들을 풀기 위한 심플렉스기법을 개략적으로 보여준다. 요약하자면, 만약 심플렉스상의 한 점에 해당하는 계수들의 행렬을 갖고 시작하면, 심플렉스 상의 인접점들로 이동시키는 일련의 피보트 스텝들을 수행하여 극대치에 이를 때까지 목적함수를 항상 증가시킨다.

아직 언급하지 않은 한 근본적 사실은 이 프로시저의 정확한 동작을 위해 아주 중요하다: 만약 어떤 한 피보트도 목적함수를 향상시킬 수 없는 점("국부적(local)극대치")에 이르면, 우리는 "전역적(global)"극대치에 도달한 것이다. 이것이 바로 심플렉스 알고리즘의 기본이다. 앞에서 언급한 것처럼, 이의(그리고 기하학적 해석시 아주 명확해 보이는 많은 다른 사실들의) 증명은 일반적으로 이 책의 범위를 아주 넘어선다. 그러나 일반적인 경우의 심플렉스 알고리즘은 위에서 보아온 단순한 문제의 경우와 근본적으로 동일한 방법으로 동작한다.

구현(Implementation)

위에서 설명한 경우에 대한 심플렉스기법의 구현은 아주 단순하다. 먼저, 필수적인 피보팅 프로시저는 37장의 Gauss소거법의 구현과 유사한 프로그램 코드를 이용한다:

```
pivot(int p, int q)
{
    int j, k;
    for (j = 0; j <= N; j++)
        for (k = M+1; k >= 1; k--)
            if (j != p && k != q)
                a[j][k] = a[j][k] - a[p][k] * a[j][q] / a[p][q];
    for (j = 0; j <= N; j++)
```

```

        if (j != p) a[j][q] = 0;
    for (k = 1; k <= M+1; k++)
        if (k != q) a[p][k] = a[p][k]/a[p][q];
    a[p][q] = 1;
}

```

이 프로그램은, 앞에서 설명한 것처럼, 행 p 에 적정 값을 곱해서 얻어진 결과를 q 열의 행 q 의 1을 제외한 모든 열 값을 영으로 만들기 위해, (필요하다면) 각 행에 더한다.

37장의 경우와 마찬가지로, $a[p][q]$ 값의 사용이 끝나기 전에 이 값을 변경하지 않도록 조심할 필요가 있다. Gauss소거에서는, 순방향-소거시에는 행렬에서 p 아래의 행들만을 처리했고, 역방향 치환시에는 Gauss-Jordan의 방법을 사용하여 p 위의 행들만을 처리했다. N 개의 미지수와 N 개의 선형방정식으로 구성된 시스템은 i 를 1에서 N 까지 증가시켜 가며 $\text{pivot}(i, i)$ 를 호출하고 가 i 가 다시 1이 될 때까지 $\text{pivot}(i, i)$ 를 호출하여 풀 수 있다.

그러면, 심플렉스 알고리즘은 단순히 위에서 설명한 p 와 q 의 값들을 찾고 pivot 을 호출하고 최적치가 구해질 때까지 또는 심플렉스가 비 한정적임이 알려질 때까지 이 과정을 반복한다.

```

for( ; ; )
{
    for (q = 0; (q<=M+1) && (a[0][q]>=0); q++) ;
    for (p = 0; (p<=N+1) && (a[p][q]<=0); p++) ;
    if (q>M || p>N) break;
    for ( i = p+1; i <= N; i++)
        if (a[i][q] > 0)
            if (a[i][M+1]/a[i][q] < a[p][M+1]/a[p][q])
                p = i;
    pivot(p, q);
}

```

만약 프로그램이 $q=M+1$ 로서 종료되면, 최적해가 찾아진다: 목적함수에 대해 구해진 값은 $a[0][M+1]$ 에 있고 변수들에 대한 값들은 기저로부터 얻어질 수 있다. 만약 프로그램이 $P=N+1$ 에서 종료되면, 비한정적 상황이 감지된다.

이 프로그램에서는 사이클의 제거 문제는 고려되지 않았다. Bland의 방법을 구현하기 위해서는 기저에서 제거될 수 있는 열을 추적함이 필요하다.(만약 행 p 를 이용하여 피보팅이 행

해진다면) 이 추적은 각 피보팅 후에 $\text{outb}[p]$ 를 q 로 만듦으로써 쉽게 행해진다. 그리고 만약 비율시험(ratio test)에서 등식이 성립하고 $\text{outb}[p] < \text{outb}[q]$ 이면, p 의 계산을 위한 루프 또한 p 를 i 로 만들도록 수정될 수 있다. 또 다른 방법으로서, 한 원소가 무작위 정수 x 를 발생시키고 각 배열 참조(reference) $a[p][q]$ (또는 $a[i][q]$)를 $a[(p+x)\%(N+1)][q]$ (또는 $a[(i+x)\%(N+1)][q]$)로 대치시킴으로서 무작위적으로 선택될 수 있다. 이것은 전과 같이 q 를 통해 검색하는 효과를 갖는다; 그러나 처음부터 하는 대신 무작위로 선택된 점에서부터 검색을 시작한다. 이런 종류의 기법은 피보팅을 하기 위한(영번째 행에 음수엔트리를 가지는) 열을 무작위적으로 선택하는데 이용될 수 있다.

위의 프로그램과 예제는 심플렉스 알고리즘에 내재한 원리를, 실제 응용분야들에서 일어나는 복잡한 사항들은 피하여, 설명해주는 한 단순한 경우를 다루고 있다. 주요 생략 사항은 프로그램은 행렬이 항등행렬(identity matrix)로 치환될 수 있는 행들과 열들의 집합인 실행 가능한 기저(feasible basis)를 갖는 것을 필요로 한다는 것이다. 이 프로그램은 목적함수 내에서 나타나는 $M - N$ 개의 변수들이 영이 되는 해가 있고 슬랙변수들을 포함하는 N 행- N 열의 부분행렬은 그 부분행렬을 항등행렬로 만듦으로써 “풀릴 수”있다는 가정하에 시작한다. 우리가 언급했던(모든 양수변수들에 대해 부등식을 갖는) 특정 유형의 선형 프로그램의 경우에는 이렇게 하기가 쉽다. 그러나 일반적으로는 심플렉스상의 어느 점을 찾을 필요가 있다. 일단 한 해가 찾아지면, 행렬을 필요한 형태로 바꿔주는(그 점을 근원점으로 매핑해주는) 적절한 변환들을 행할 수 있다. 그러나 처음부터 해가 존재하는지의 여부조차 모를 수가 있다.

사실상 해의 존재여부를 탐색하는 것은 최적의 해(최적의 해가 있을 경우)를 찾는 것만큼 어렵다는 사실이 알려져 있다. 따라서 해의 존재유무를 탐색할 때 일반적으로 사용되는 기법이 심플렉스 기법이라는 것은 놀랄만한 일이 아니다. 특별히, 인공변수들 s_1, s_2, \dots, s_N 의 집합을 추가하고 변수 s_i 를 i 번째 방정식에 추가시킨다. 이 추가는 행렬에 항등행렬로 채워진 N 개의 열들을 추가함으로써 쉽게 행해진다. 이렇게 하면, 바로 이 새로운 선형 프로그램에 대한 실행가능한 기저가 생긴다. 그리고 나서 위 알고리즘을 목적함수 $-s_1 - s_2 - \dots - s_N$ 을 갖고 수행시킨다. 만약 원래의 선형 프로그램에 해가 있다면, 이 목적함수는 영에서 극대화된다. 만약 도달한 극대치가 영이 아니면, 원래의 선형 프로그램은 실행 불가능한(즉, 해를 구할 수 없는) 것이다. 만약 극대치가 영이면, 정상 상황으로서, s_1, s_2, \dots, s_n 모두가 기저가 아닌 변수가 된다. 따라서, 원래의 선형 프로그램에 대해 실행 가능한(즉, 해가 있는) 기저를 계산한 것이다. 퇴행 경우들에 있어서는, 일부 인공변수들이 기저에 남아 있을 수 있다. 그러므로(비용의 변동 없이) 이들을 제거하기 위해 피보팅을 더 수행함이 필요하다.

요약하면, 일반적인 선형 프로그램들을 풀기 위해서는 이단계의 과정이 보통 이용된다. 먼저, 원래 문제에 대한 심플렉스상의 한 점을 얻기 위해 인공변수들 s 를 포함시켜 선형 프로그램을 푼다. 그리고나서, 이 s 변수들을 제거하고 이 시점에서 계속 진행하여 해에 이르기 위해 원래의 목적 함수를 다시 도입한다.

심플렉스 기법의 실행시간을 분석하는 것은 아주 복잡하여 이 분석결과로서 알려진 것들이 거의 없다. 어느 누구든 “최선의” 피보트 선택전략이 무엇인지를 알지 못한다. 왜냐하면 어떤 결과들도 어느 타당한 부류의 문제들에 대해 얼마나 많은 피보팅 스텝이 요구되는지를 알려주지 못하기 때문이다. 심플렉스 기법의 수행시간이(변수들의 수의 지수함수인) 매우 큰 예제들을 인위적으로 만드는 것이 가능하다. 그러나, 실제적인 경우, 이 알고리즘을 사용해온 사람들은 만장일치로 실제 문제들을 풀 때 이 알고리즘의 효율성이 좋음을 말하고 있다.

우리가 살펴본 단순화된 버전의 심플렉스 알고리즘은 아주 유용하다, 그러나 이 알고리즘은 다양한 종류의 중요하고 실제적인 문제들을 푸는데 사용될 수 있는 도구들의 집합을 제공하는, 한 일반적이고 멋진 수학적 구조물의 한 부분일 뿐이다.

연습문제

1. 부등식 $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_1 + 2x_2 \leq 20$ 과 $x_1 + x_2 + x_3 \leq 10$ 에 의해 정의된 심플렉스를 그려라.
2. 선택된 피보트 열이 $a[0][q]$ 가 음수인 최대의 q 일 때 본문의 예제에 대해 산출된 행렬들의 순서를 구하라.
3. 목적함수가 $x_1 + 5x_2 + x_3$ 일 때, 본문의 예제에 대해 산출된 행렬들의 순서를 구하라.
4. 심플렉스 알고리즘이 값이 모두 영인 한 열을 갖는 행렬 상에서 수행되면 어떤 일이 발생하는지를 설명하라.
5. 입력행렬의 행들이 서로 바뀌어지는 경우에도 심플렉스 알고리즘은 동일한 스텝 수를 이용하는가?
6. 42장의 배낭 문제의 예제에 대한 선형 프로그래밍 공식을 만들라.
7. “조건 $x_1, \dots, x_M \leq 1$ 이고 $x_1, \dots, x_M \geq 0$ 을 만족시키면서 $x_1 + \dots + x_M$ 을 극대화하라”는 선형 프로그램을 푸는데 필요한 피보트 스텝의 수는 얼마인가?
8. 심플렉스 알고리즘이 적어도 $N/2$ 피보트들을 필요로 하는 두 변수 상에서, N 개의 부등식을 가지는 선형 프로그램을 구성하라.
9. 최대-증분법과 최하 강하법간의 차이를 설명해주는 삼차원 선형 프로그래밍 문제를 찾아 설명하라.
10. 본문의 구현 프로그램을 최적해인 점의 좌표를 출력하도록 수정하라.

어떤 문제들은 답을 구하기 위해 많은 답의 가능성들에 대한 검색을 필요로 하며 효율적인 알고리즘들에 의해 그 해결책을 얻기가 쉽지 않을 때가 있다. 이 장에서는 이런 종류의 문제들이 갖는 일부 특성과 이들을 해결하는데 유용한 기법들에 대해 공부하기로 한다.

우선 효율적인 알고리즘이 무엇인가에 대해 어느 정도 생각을 정리해 보아야 한다. 이제까지 다루어 왔던 대부분의 분야에서는 알고리즘이 선형이거나 또는 그 수행 시간이 $N \log N$ 이나 $N^{3/2}$ 같은 것에 비례해야만 효율적인 것으로 생각해 왔다. 일반적으로 이차(quadratic) 알고리즘은 나쁘고 삼차(cubic) 알고리즘은 아주 나쁜 것으로 간주해 왔다. 그러나 일부 컴퓨터 과학자들은 이 장과 다음 장에서 고려할 문제들에 대해 삼차 알고리즘을 찾더라도 정말로 좋게 여길 수도 있다. 실제로 N^{50} 알고리즘조차도(이론적으로는) 아주 좋은 알고리즘일 수 있다. 왜냐하면 그런 문제들은 지수(exponential)시간이 소요되는 것으로 알려져 있기 때문이다.

한 알고리즘이 2^N 에 비례하는 시간을 소모한다고 하자. 만약 오늘날 사용가능한 가장 빠른 슈퍼컴퓨터보다 1000배나 더 빨리 동작하는 컴퓨터가 있다면, 아마도 $N = 50$ 인 경우, 한 시간 정도면 문제를 해결할 수 있을 것이다.(알고리즘이 아주 단순하다는 최선의 가정 하에서) 그러나 $N = 51$ 의 경우에는 두 시간이 걸려야 해결할 수 있고, $N = 59$ 의 경우에는 일년이란 시간을 필요로 한다. 설혹 일백만배나 빠른 새로운 컴퓨터가 개발되더라도, $N = 100$ 의 경우에는 일년이 걸려도 해결할 수가 없을 것이다. 현실적으로, N 은 25에서 30정도로 고정돼야만 된다. 이럴 경우, 더욱 효율적인 알고리즘은 아마도 $N = 100$ 인 경우에 대해서 보다 실질적인 시간과 경비를 절약하며 문제를 해결할 수 있는 알고리즘일 것이다.

이런 유형의 문제중 가장 잘 알려진 것이 “외판원(traveling salesmen)” 문제로서, “ N 개의 도시가 있을 때, 이 모두를 방문하여 연결하는 최단 경로를 찾아라”는 것이다. 단 어떤 도시

도 두 번 이상 방문해서는 안된다. 이 문제는 아주 많은 응용분야에서 자연적으로 발생했으므로 아주 오랫동안 연구되어져 왔다. 이 장에서는 이 문제를 일부 기본적인 기법들의 시험을 위한 예로서 사용한다. 많은 진보된 방법들이 이 문제에 대해 개발되어져 왔으나, 아직도 $N = 1000$ 인 경우에 대해서는 그 해결 방안에 대해서 생각조차 하지 못하고 있다.

외판원 문제는 아주 어려운 문제이다. 왜냐하면 아주 많은 가능한 경로(tour)들의 길이를 조사하지 않을 방법이 없어 보이기 때문이다. 모든 경로를 조사하는 것이 완전-검색이다: 우선, 어떻게 이 조사가 행해지는지 살펴보자. 그리고 나서, 조사-결정 과정에서 가능한 한 빨리 옳지 않은 결정들을 발견함으로써 조사되어야 할 가능성들의 개수를 현격히 줄일 수 있도록 하는 방법을 찾아보자.

앞서 언급한 바와 같이, 아주 긴 경로를 갖는 외판원 문제를 해결하는 것은 실제적으로는 현재 알려진 것 중 가장 좋은 기법을 사용하더라도 불가능하다. 다음 장에서 알게 되겠지만, 이러한 문제들이 실제로 발생했을 때 어떻게 할 수 있을까? 외판원은 어떤 해결책이든 찾아야한다. 물론 너무 해결하기 어려운 문제나 상태가 발생하는 것을 간과할 수는 없다. 이 장의 말미에서는, 완전 검색을 필요로 하는 실제적 문제들을 다루도록 만들어진 몇개의 방법들의 예를 보기로 한다. 다음 장에서는, 많은 이와 같은 문제들에 대해 왜 효율적인 알고리즘들이 찾아질 수 없는지에 대한 그리고 그 이유들을 자세히 알아보기로 한다.

그래프에서의 완전-검색(Exhaustive Search in Graphs)

만약 외판원이 특정 도시 간만 여행한다면(예로서, 비행기를 타고 간다면), 문제를 그래프로서 직접적으로 모델화 할 수 있다: 주어진 가중치(방향성 일수도 있는) 그래프를 가지고, 모든 노드들을 연결하는 최소 단순 사이클을 찾으려 할 수도 있다.

이것은 더욱 간단히 보이는 또 한 가지의 문제를 생각하게 한다: 주어진 무방향 그래프를 가지고, 단순 사이클을 갖도록 모든 노드들을 연결하는 방법이 있을까 하는 것이다. 즉, 임의의 노드에서 시작하여 모든 다른 노드들을 방문해서 시작노드로 돌아올 수 있을까 하는 것이다. 물론 그래프상의 모든 노드는 정확히 한 번만 방문해야 한다. 이 문제는 Hamilton 사이클(Hamilton cycle) 문제로서 널리 알려져 있다. 다음 장에서, 아주 엄격한 기술적 의미에서 볼 때, 이 문제가 외판원 문제와 계산적으로 동치(computationally equivalent)임을 알 수 있을 것이다.

29장과 31장에서 보았듯이, 그래프상의 모든 노드들을 체계적으로 방문하는 많은 방법들이 있다. 이런 알고리즘들의 경우에는, 계산방식을 조정 배열하여 각 노드를 단지 한 번씩만 방문하도록 하는 것이 가능했다. Hamilton 사이클 문제에서는, 이러한 해결책이 있는지 명백하지 않다. 아마도 각 노드를 여러번 방문해야 될 듯 싶다. 또한 다른 문제 등의 경우에는, 트리를 형성할 수도 있다; 검색 시 “한 경로의 막다른 끝(dead end)에 이르면, 트리의 다른 부분에 대해 작업을 계속 진행해야 한다. 이런 유형의 문제의 경우에는, 트리가 특정 구조(사이클)를 가져야만 한다: 만약 검색과정중 구성되는 트리가 사이클을 형성할 수 없음이 확인되면, 해당 상태로 복귀하여 그 부분을 재형성해야 한다.

이에 관련한 문제점들에 대한 예로서, 그림 44.1에 있는 Hamilton 사이클 문제를 보자. 이 그래프에서 깊이-우선검색(depth-first search)은 A B C E F D G H I K J L M의 순으로 노드들을 방문한다.(인접-행렬이나 정렬된 인접-행렬 리스트 표현을 가정했을 때) 이것은 단순 사이클이 아니므로, Hamilton 사이클을 찾으려면 다른 노드-방문 방법을 시도해야만 한다. 우리는 visit 프로시저를 다음과 같이 조금 수정함으로써, 모든 가능성을 체계적으로 시도해 볼 수 있다:

```
visit( int k )
{
    int t;
    val[k] = ++id;
    for(t = 1; t <= V; t++)
        if (a[k][t])
            if (val[t] == 0) visit(t);
    id--; val[k] = 0;
}
```

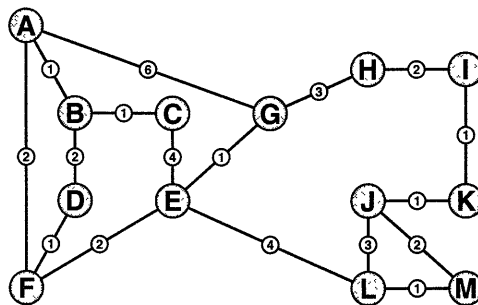


그림 44.1 외판원 문제의 한 예

이 프로시저는 자신이 거쳐간 모든 노드를 영이 아닌 val값으로 마크(mark)하는 대신에 “방문 뒤에 그 자취를 지움”으로써 id와 val배열을 자신이 그 값들을 발견했을 때와 똑같이 남겨 둔다. 유일하게 마크된 노드들은 방문이 완료되지 않은 노드들인데, 이들은 그래프에서 초기노드에서 현재 방문중인 노드까지의 길이가 id인 단순경로에 해당한다. 한 노드를 방문하기 위해서는 모든 마크되지 않은 인접-노드들을 방문하면 된다.(마크된 노드들은 단순 경로에 해당되지 않을 수 있다) 재귀적(recursive) 프로시저는 초기 노드에서 시작하는 그래프에서의 모든 단순 경로들을 조사한다.

그림 44.2는 그림 44.1의 예제 그래프에서 위 프로시저에 의해 경로가 조사된 순서를 보여준다. 트리에서의 각 노드는 visit를 한번 호출(call)한 것에 해당한다: 이처럼 각 노드의 자식노드들(descendents)은 호출시 마크 안된 인접-노드들 이다. 트리의 한 노드에서 루트까지의 각 경로는 그래프의 단순 경로에 해당한다. 따라서, 첫 번째 경로는 A B C E F D이다. 이 시점에서 D에 인접한 모든 정점들은(0이 아닌 val값으로) 마크된다. 그러므로 D에 대한 visit는 D의 마크를 해제하고(unmark) 복귀(return)한다. 그러면 F에 대한 visit는 F를 마크 해제하고 복귀한다. 또한 E에 대한 visit는 G에 대해 시도하고 G는 H에 대해 시도하고 하는 식으로 시도가 계속 되어, 궁극적으로는, 경로 A B C E G H I K J L M이 찾아진다. 깊이-우선검색에서는 노드들이 방문된 후 마크되나, 완전-검색에서는 노드들이 여러 번 방문됨에 유의하라. 노드를 “마크 해제함”은 완전-검색이 깊이-우선검색과 근본적으로 다를 것을 알려준다. 비록 프로그램 코드는 유사해 보일지라도, 여러분들은 이 차이점을 분명히 이해해야 한다.

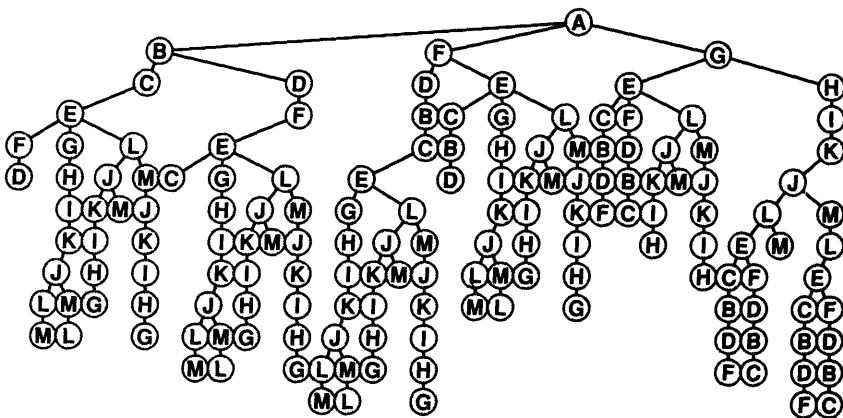


그림 44.2 완전 검색

앞서 말한 바와 같이, id 는 현재 조사되고 있는 경로의 길이이고, $val[k]$ 는 그 경로 상의 노드 k 의 위치이다. 따라서 우리는 위의 `visit` 프로시저를 $val[k]=v$ 일 때 k 에서 l 로 가는 선분(edge)이 있는지를 조사하게 만듦으로써, Hamilton 사이클의 존재 유무를 조사하게 할 수 있다. 상기 예제의 경우에는,(트리의 경우에는 2개가 있었지만) 양방향으로 운행하는 단지 한 개의 Hamilton 사이클이 존재한다.(길이 v 인 3개의 또다른 경로가 있다) 위 프로그램은 val 배열에서 한 경로의 길이를 계속 추적하고 찾아진 Hamilton 사이클의 길이 중 최소치를 찾음으로서 외판원 문제를 해결할 수 있다.

역추적(Backtracking)

앞서 설명한 완전-검색 프로시저에 의해 걸리는 시간은 `visit`의 호출 횟수에 비례한다. 이 횟수는 완전-검색 트리에서의 노드 수이다. 그래프가 큰 경우에는, 시간이 매우 오래 걸릴 것이다. 예를 들어, 그래프가 각 노드가 모든 다른 노드로 연결된 완전그래프(complete graph)라면, $V!$ 개의 단순 사이클이 있는데 이들 각각은 노드들의 각 배열에 해당한다.(이 경우는 아래에서 자세히 알아보기로 한다) 그림 44.1의 그래프에서조차도, 육안으로는 Hamilton 사이클을 찾기가 쉽지 않다. 그러므로, 컴퓨터를 사용하는 보다 효과적인 방법을 찾아야 할 것이다.

다음으로, 시도 횟수를 현저히 줄일 수 있는 기법들을 알아보자. 이들 기법들 모두에는, 재귀적 호출들이 행해지지 않아도 되는 특정 노드들을 찾아내기 위해 `visit`에 대한 시험이 추가된다. 이것은 완전-검색 트리에서 불필요한 부분을 제거(pruning)하는 것에 해당한다: 즉, 특정 트리의 가지와 그에 연결된 모든 노드를 없애는 것이다.

한 중요한 제거 기법은 대칭 부분을 제거하는 것이다. 이 기법의 가치는 상기 예제에서 볼 때, 각 사이클이 양방향으로 운행되어 따라서 두번 찾아진다는 사실에 의해 입증된다. 이 경우, 우리는 세 개의 특정노드가 특정순서로 나타나게함으로써 각 사이클을 한 번씩만 찾는 것을 보증할 수 있다. 예로서, 우리가 세 노드 C 가 노드 A 뒤에, 그리고 노드 B 앞에 나타나게 한다면, 노드 C 가 이미 그 경로 상에 있지 않는 한, 노드 B 에 대해 `visit`를 호출할 필요가 없다. 이런방식에 의해 현저히 줄어든 트리를 그림 44.3에서 볼 수 있다.

이런 기법이 항상 적용 가능한 것은 아니다. 예로서, 모든 정점들을 연결하는(반드시 사이클이 아닐 수도 있는) 한 경로를 찾으려 한다 하자. 이 경우, 위 기법은 사용될 수 없다. 왜냐하면 경로가 사이클이 될지 아닐지를 미리 알 수가 없기 때문이다.

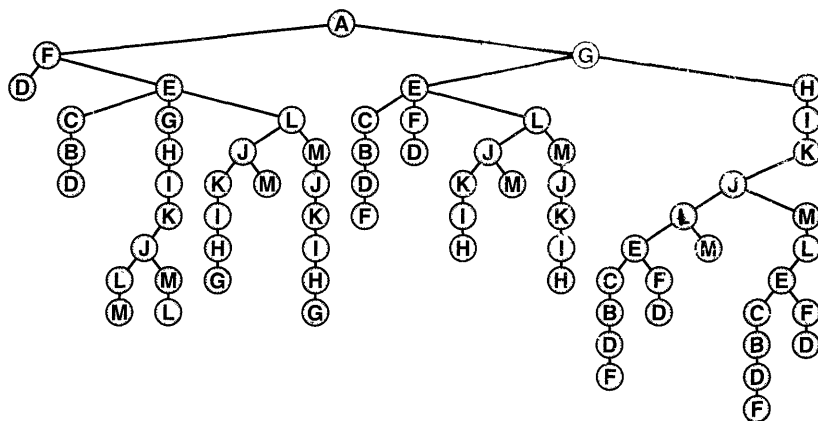


그림 44.3 B앞에 그리고 C앞에 A를 갖는 사이클의 검색

한 노드에서 트리의 검색을 포기할 때마다, 그 노드밑의 모든 서브트리(subtree)에 대한 검색도 하지 않는다. 이 방식의 사용은, 매우 큰 트리들의 경우, 아주 실제적인 절약 효과가 있다. 실제로, 이러한 절약은 매우 중요하므로 visit내에서 가능하면 이를 많이 행함으로서 재귀적 호출의 수행을 피해야 한다. 주어진 예제의 경우, 수행 방식에 여러 가지가 있을 수 있다: 하나는 일부 경로들의 경우, 마크안된 노드들은 연결이 되어 있지 않도록 분할하면, 어떤 사이클도 찾아지지 않는다는 점에 주목하는 것이다. 예로서, 그림 44.1에서는 ABE로 시작하는 어느 단순 경로도 있을 수 없다. 왜냐하면, 이 경로는 B, C, D를 그래프의 나머지 부분과 분리시켜 놓기 때문이다. 이 사실을 찾는데 드는 깊이-우선검색의 비용을 보면, visit에 대한 25번의 재귀적 호출이 절약된다.(그림 44.3을 보라)

그림 44.4는 그림 44.3의 트리에 이 규칙을 적용했을 때 생기는 검색 트리이다. 또 한 번 트리가 아주 조그마하게 줄어들었다: 원래 완전-검색트리(그림 44.2)의 153개 노드에서 단지 19개의 노드만이 남아있다. 이 장난감 같은 문제에서 얻어진 절약 효과는 더 복잡하고 큰 실제 문제에서 얻어질 수 있는 결과에 대한 지표일 뿐이다. 트리에서 가지 절삭을 많이 하면 할수록, 실질적 절약이 많아진다; 명백히 절삭되어야 할 것을 하지 않으면 상당한 낭비가 생길 것이다.

상기 설명된 모든 가능한 해결책을 체계적으로 발생시켜서 문제를 해결해 나가는 일반적인 프로시저를 역추적(backtracking)이라고 한다. 한 문제에 대한 부분해들은 연속적으로 적용하여 한 완전한 해를 산출할 수 있는 경우에는 언제나, 위 프로그램들 같은 재귀적 구현 방식이 적합하다. 앞서서와 마찬가지로, 이 과정은 노드들이 부분 해에 상응하는 완전-검색

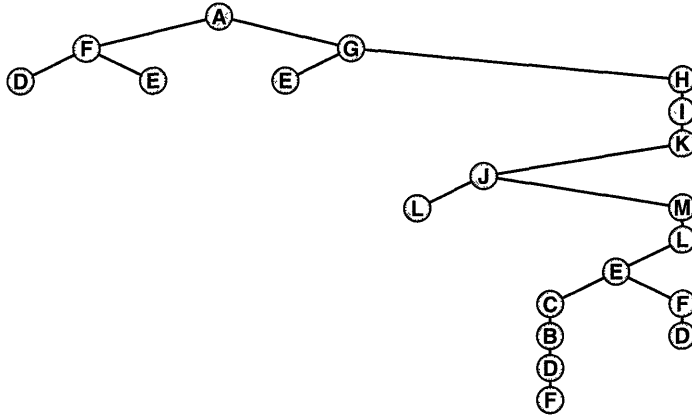


그림 44.4 그래프가 분리되었을 때 절삭이 있는 사이클에 대한 검색

트리로서 설명될 수 있다. 트리에서 하향진행은 보다 완전한 해로의 접근을 의미한다; 트리에서의 상향진행은 이전에 만들어진 부분로 다시 하향 진행하여 갈 수 있는 지점까지의 역추적에 해당한다.

또 다른 예로서, 42장의 배낭 문제를 생각해 보자. 그때 언급한 바와 같이, 이 문제는 값들이 정수가 아닌 경우에는 상당히 풀기가 어렵다. 이 문제에 대한 부분 해들은 명백히 “배낭”에 대한 일부 선택 항목들이고, 역추적은 어떤 다른 조합을 찾기 위해 한 항목을 제거하는 것에 해당한다. 검색 트리의 대칭부를 제거하는 것은 이 문제의 경우 아주 효과적이다. 왜냐하면 물체들이 배낭에 놓여지는 순서는 그 비용에 영향을 주지 않기 때문이다.

최적 경로를 찾을 때(외판원 문제에서), 검색이 성공 못할 것으로 판명되자 마자, 검색을 종료시키는 또하나의 중요한 제거(pruning)기법이 가능하다. 그래프에서 비용이 x 인 경로가 찾아졌다 하자. 이 시점에서, 이제까지 찾아온 경로중 그 비용이 x 보다 큰 경로에 대한 검색을 계속함은 무의미하다. 이의 중단은 만약 현재까지의 부분 경로가 이제까지 찾아진 최적 총 경로의 비용보다 더 크다면, visit에 대한 재귀적 호출을 중단함으로써 간단히 구현할 수 있다. 이런 방식을 이용하는 것은 최소 비용 경로를 찾는 데 문제를 일으키지 않는다.

제거 기법은 만약 어느 저비용 경로가 검색중 일찍 찾아졌을 때, 더욱 효과적이다; 이렇게 되게 하는 한 방법은 현 노드에 인접한 노드들을 비용증가의 순으로 방문 하는 것이다. 사실상, 이보다 더 좋은 방법도 있을 수 있다: 우리는 종종 주어진 부분 경로에서 시작하여, 모든 총 경로들에 대한 경계값(bound)을 계산할 수 있다. 예제에서, 마크 안된 노드들에 대한 최소 스패닝 트리(spanning tree)의 비용을 추가함으로써, 마크된 노드들로 구성된, 부분 경로

로 시작하는, 완전경로의 비용에 대해 보다 개선된 경계값을 얻을 수 있다.(경로의 나머지 부분은 마크안된 노드들에 대한 스페닝 트리이다; 그 비용은 확실히 그 노드들의 최소 스페닝 트리의 비용보다 작을 수 없을 것이다)

검사할 필요가 있는 완전해(full solution)들의 수를 제한하기 위해 부분해들에 대한 경계값들을 계산하는 일반적인 기법들을 종종 분기-한계(branch-and-bound)기법이라고 한다. 물론 이 기법은 비용이 경로 상에 부가되어 있을 때면 언제나(그리고 비용을 최소화하려 할 때) 적용 가능하다. 보통 이런 문제들에 있어서의 목적은 해결책(해)을 구하기 위한 방대한 수의 검색가능성을 줄이는 것이다-잘못 선택된 경로로 진행하는 것을 피하기 위해, 이전에 설명된 것 같은, 상당수의 경험적(heuristic) 규칙들을 적용하는 것이 정상적이다.

역추적과 분기-한계 기법은 일반적인 문제의 해결을 위한 기법으로서 아주 널리 사용된다. 예로서, 이들은 장기나 체커(checker)게임 같은 것을 수행하는 많은 프로그램들의 기본이 된다. 이 경우, 부분해는 게임보드상의 모든 부분중 이동 가능한 일부 위치들일 것이고, 완전-검색 트리에서 한 노드의 자식노드들은 일부 올바른 이동의 결과에 따른 위치일 것이다. 이 상적으로는, 프로그램이 모든 가능성에 대해 완전히 검색을 시도 한 후, 상대방이 어떻게 응수하더라도 이길 수 있도록 이동함이 최고이나, 그렇게 하기에는 너무도 많은 가능성이 존재한다. 그래서 역추적 검색은 통상적으로 아주 복잡한 제거규칙들과 함께 수행되어 단지 관심사가 되는 부분들만 조사한다. 완전 검색 기법들은 인공지능 분야에서도 사용된다.

다음 장에서는, 이러한 기법들을 사용하여 처리될 수 있는-이제까지 우리가 공부해온 것과 유사한-몇몇 다른 문제들을 공부할 것이다. 특정 문제를 푸는 데에는 검색을 제약할 수 있는 복잡한 기준들의 개발이 필요할 수 있다. 우리는 지금까지 외판원 문제에서 시도되어온 많은 기법들 중의 아주 일부 예만을 살펴봤다. 이들과 복잡도가 비슷한 다른 중요한 문제들 또한 개발되어 왔다.

그 기준들이 얼마나 복잡하든지 간에, 역추적 알고리즘의 수행 시간이 지수시간(exponential time)이 걸린다는 것은 일반적으로 알려진 사실이다. 대강 말해서, 만약 검색 트리내의 각 노드가 평균적으로, a 개의 자식노드를 가지고 해결 경로(solution path)의 길이가 N 이라면, 그 트리에서의 노드 수는 a^N 에 비례할 것으로 기대된다. 다른 역추적 규칙들도 a 값, 즉 각 노드에서 시도할 선택의 수를 줄이는 것에 해당한다. 물론 이에 대해 좀더 생각해 볼 필요가 있다. 왜냐하면 a 의 감소는 해결할 문제의 크기를 증가시키기 때문이다. 예를 들어, 1.1^N 에 비례하여 동작하는 알고리즘은 2^N 의 비례하여 동작하는 알고리즘보다 8배나 더 커질 수 있기

때문이다. 한편, 앞서 언급한 것처럼, 매우 큰 문제에 대해서는 양자 모두 좋은 결과를 기대할 수 없다.

이탈(Digression): 순열 생성(Permutation Generation)

N 개의 다른 항목을 배열할 수 있는 모든 가능한 방법을 찾는 프로그램을 작성하는 것은 아주 재미있는 문제이다. 이 순열 생성 문제에 대한 한 간단한 프로그램은 앞서 주어진 그래프에 대한 완전 검색 프로그램으로부터 직접 유도될 수 있다. 앞서 언급한 것처럼, 이 프로그램이 완전-그래프 상에서 수행된다면, 모든 가능한 순서로 그 그래프의 모든 정점에 대해 방문을 시도해야만 한다. 검색 경로상에 나타난 순서대로 모든 정점들을 명명(labeling)하여 유지하고, 아래 프로그램에서처럼 길이 v 인 경로가 생길 때마다 모든 정점명을 출력함으로써 모든 순열을 얻는다:

```
visit(int k)
{
    int t;
    val[k] = ++id;
    if (id == V) writeperm();
    for(t = 1; t <= V; t++)
        if (val[t] == 0) visit(t);
    id--; val[k] = 0;
}
```

이 프로그램은 위 프로시저에서 인접 행렬에 대한 모든 참조 부분을 제거함으로써 만들어졌다. 왜냐하면 완전 그래프에서는 모든 노드간에 선분이 존재하기 때문이다. 프로시저 writeperm은 단지 val배열의 해당 엔트리를 기록한다. 이 기록은 $id=v$ 일 때마다 행해진다.(실제로 이 프로그램은 $id=v$ 일 때의 for 루프문을 생략하면 좀 더 개선될 수 있다. 왜냐하면 이 시점에는 모든 val배열의 엔트리 값이 영이 아님을 알기 때문이다) 정수 1에서 N 까지의 모든 순열을 출력하기 위해, $id=-1$ 로 그리고 val 배열은 영으로 초기화한 후, visit(0)를 호출하여 이 프로시저를 시작시킨다. 이 경우는 완전 그래프 상에 모조(dummy)노드를 추가하고 노드 0에서 시작하는 그래프 내의 모든 경로에 대해 조사를 하는 것에 해당한다. $N=4$ 인 경우, 이와 같은 방법으로 프로시저를 실행하면, 다음과 같은 출력을 얻는다:(여기서는 2열로 출력됐다)

1 2 3 4	2 3 1 4
1 2 4 3	2 4 1 3
1 3 2 4	3 2 1 4
1 4 2 3	4 2 1 3
1 3 4 2	3 4 1 2
1 4 3 2	4 3 1 2
2 1 3 4	2 3 4 1
2 1 4 3	2 4 3 1
3 1 2 4	3 2 4 1
4 1 2 3	4 2 3 1
3 1 4 2	3 4 2 1
4 1 3 2	4 3 2 1

말할 필요도 없이, 완전 그래프에서 경로 생성 프로시저는 겨우 이행될 정도이다. 그러나, 직접적인 검사를 통해, 프로시저가 첫 위치에 1을 갖는 $(N - 1)!$ 개의 순열을 먼저 생성함으로써 정수 1에서 N 까지에 대한 총 $N!$ 개의 순열을 생성하고(물론 2에서 N 까지 재귀적으로 수행하여), 그리고 나서 2번째 위치에 1을 갖는 $(N - 1)!$ 개의 순열을 생성하는 것과 같은 작업을 반복한다는 것을 알 수 있다.

$N = 16$ 의 경우에는, 이 프로그램을 사용할 생각조차 하기 어렵다. 왜냐 하면 $16! > 2^{50}$ 이기 때문이다. 아직도 이 문제는 연구할 만한 중요성이 있다. 왜냐하면 한 집합에서 원소들을 재배열할 때 생기는 문제를 해결하는 것이 역추적 프로그램의 기본을 이룰 수 있기 때문이다.

예로서 Euclid의 외판원 문제를 생각해 보자: 주어진 평면상의 N 개 지점의 집합에서, 이들 모두를 연결하는 최소의 여행 경로를 찾아라. 지점들의 각 배열이 하나의 여행 경로이므로, 위 프로그램은 문제에 대한 해결책을 완전-검색을 함으로써 찾을 수 있는데, 이는 각 여행 경로에 대한 비용을 계산하고 이중 최소 비용을 갖는 경로를 찾도록 수정함으로써 가능하다. 그런 후에, 이 Euclid문제에 국한한 다양한 역추적 경험적 기법 뿐만 아니라, 앞서 사용되어진 것과 동일한 분기-한계기법이 적용될 수 있다.(예로서, 최적의 여행 경로는 자신을 교차해서는 안된다는 것을 증명하기는 쉽다. 따라서 검색 도중 교차가 발생하는 경우 모든 부분 경로들에 대한 검색을 중단할 수 있다) 여러 경험적 검색 기법(search heuristics)이 있는데, 이들은 많은 순열 방법들에 해당한다. 이러한 기법들은 많은 일을 덜어 주는 반면, 다

큰 많은 할 일을 남겨 준다. Euclid의 외판원 문제에 대한 정확한 해결책을 찾는 것이 항상 쉬운 것은 아니다. $N = 16$ 이하인 경우에서조차도 쉽지 않을 수가 있다.

순열 생성이 관심을 끄는 또하나의 이유는 다른 조합체들을 생성하는데 연관된 많은 프로시저들이 있기 때문이다. 어떤 경우에는, 생성된 물체의 수가 순열의 경우처럼 그렇게 많지 않을 수 있다. 실제로 이런 경우의 프로시저들은 큰 N 값에 대해서도 아주 유용할 수 있다. N 개의 항목의 집합에서 크기 k 인 부분집합을 선택해 내는 모든 방법을 찾아내는 프로시저가 그 한 예이다. 큰 N 값과 작은 k 값에 대해, 이 일을 처리하는 방법들의 수는 대략 N^k 에 비례한다. 이와 같은 프로시저는 배낭문제를 풀기 위한 역추적 프로그램의 기본으로서 사용될 수도 있다.

근사치 알고리즘(Approximation Algorithm)

최단 여행 경로를 찾는 것은 일반적으로 아주 많은 연산 시간을 필요로 하므로, 최단거리에 거의 준하는 경로를 찾는, 보다 쉬운 방법이 있는지를 생각해 볼만하다. 만약 절대적으로 최단 가능 경로를 찾아야 된다는 제약만 완화시키면, 앞서 설명한 기법들을 가지고 할 수 있는 것보다 훨씬 많은 문제들을 다룰 수 있을 것이다.

예로서, 최적 경로보다 최대 두배정도 긴 경로를 찾는 것은 상대적으로 훨씬 쉬운 것이다. 이 방법은 최소 스패닝 트리를 찾는 방법에 근거한다: 이 방법은 경로 길이에 대한(앞서 언

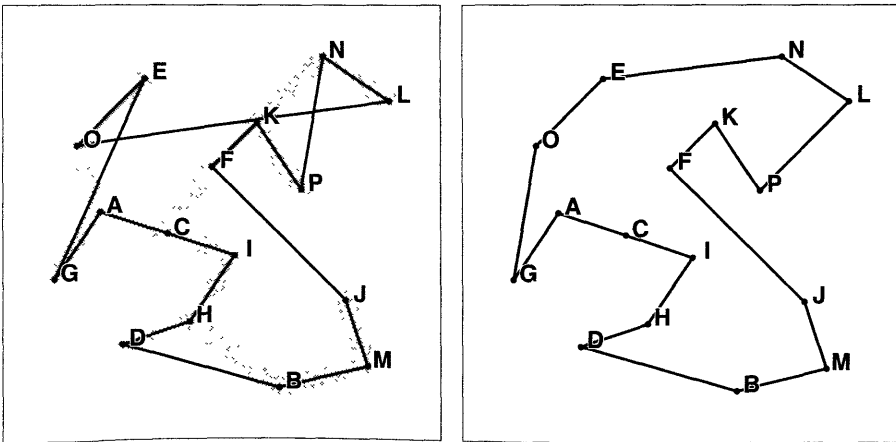


그림 44.5 단순 Euclid의 외판원 경로

급한 것처럼) 최소 경계치(lower bound) 뿐만 아니라 아래 DP 설명된 바와 같이 최대 경계치(upper bound) 또한 알게 해준다. 주어진 최소 스패닝 트리를 가지고, 다음 프로시저를 사용하여 최소 스패닝 트리의 노드들을 방문하여 경로를 찾아라: 노드 x 를 처리하기 위해, x 를 방문하고 나서 x 의 모든 자식노드들을 방문하라. 이 방문 프로시저를 재귀적으로 적용하여 x 의 모든 자식노드들을 방문한 후 노드 x 로 되돌아와서 종료한다. 이 경로는 스패닝 트리상의 모든 선분을 두 번 지나간다. 그러므로, 경로 비용은 트리 비용의 두배가 된다. 이것은 단순 경로가 아니다. 왜냐하면 노드들이 여러번 방문될 수 있기 때문이다. 그러나 이 경로 방식은 각 노드가 처음 나타나는 것을 제외한 모든 경우를 제거하면 단순 경로도 변환될 수 있다. 한 노드 발생의 제거는 그 노드를 지나가는 지름길을 찾는 것에 해당한다: 이러한 제거가 경로 비용의 증가를 가져오지는 않는다. 이같이 하여, 최소 스패닝 트리의 경로 비용의 두배보다는 적은 비용을 갖는 단순 경로를 찾을 수 있다.

예를 들면, 그림 44.5의 좌측 다이어그램은(31장에서 설명되어지고 계산된) 해당 단순 경로와 함께 샘플 지점들의 집합에 대한 최소 스패닝 트리를 보여준다. 이 경로는 분명히 최적 경로는 아니다. 왜냐하면 그 자신이 경로를 교차하기 때문이다. 아주 많은 임의로 선택된 지점들의 경우, 이런 방식으로 만들어진 경로는, 비록 결론을 내릴 수 있는 어떤 분석도 하지는 않았지만, 최적에 가까울 수도 있을 듯 싶다.

시도되어온 또 하나의 방식은 주어진 경로를 개선하기 위한 기법들을 개발하고 이들을 반복적으로 적용하면 더 짧은 경로가 얻어질 수 있다는 희망하에 행해지는 방식이다. 예로서, 그래프의 거리가 평면상의 지점간의 거리인 경우, Euclid의 외판원 문제에서 자신을 교차하는 경로는 그 운행을 다음과 같이 제거함으로써 개선될 수 있다. 만약 직선 AB와 CD를 없애고 AD와 CB를 추가함으로써 얻어진다. 이 프로시저를 반복적용하면, 주어진 임의의 경로에 대해 더 이상 교차가 없는 경로를 찾을 수 있을 것이다. 예를 들면,(그림 44.5 좌측에 있는) 최소 스패닝 트리로부터 산출된 경로에 적용된 프로시저는 그림 44.5의 우측에서처럼 더 단축된 경로를 산출한다.

사실상, Euclid의 외판원 문제에 대해 근사치를 얻는 가장 효과적인 방법은 S. Lin에 의해 만들어 졌는데, 프로시저를 일반화하여 현 경로 상에서 셋 또는 네 개의 선분들을 바꿔치기함으로써 경로들을 개선하는 것이다. 이 프로시저를 더 이상 개선이 없을 때까지 연속적으로 초기의 무작위로 선택된 경로 상에 적용함으로써, 아주 좋은 결과가 얻어졌다. 보통 사람들은 최적치에 가까운 경로에서 시작하여 개선을 시도함이 더 좋다고 생각할 수 있다. 그러나 Lin의 연구는 이것이 꼭 그렇지는 않다는 것을 보여주고 있다.

위에 기술된 외판원 문제에 대한 근사치를 얻기 위한 여러 접근 방식들은 완전 검색을 위해 사용될 수 있는 기법들의 형태들에 대한 지침들일 뿐이다. 위의 간략한 설명으로 이제까지 만들어진 많은 훌륭한 해결 방식들을 올바르게 다뤘다고 할 수는 없다. 이런 형태의 알고리즘들의 공식화와 분석은 컴퓨터 과학 분야에서 아직도 아주 활발한 연구 분야이다.

혹자는 왜 외판원 문제가 그리고 간접적으로 언급한 문제들이 완전 검색을 필요로 하는가에 대해 의문을 가질 수도 있다. 우리가 최소 스패닝 트리를 찾을 수 있는 것처럼, 쉽고 빠르게 최소 경로를 찾을 수 있는 좋은 알고리즘이 있을 수 있지 않겠는가? 다음 장에서는 왜 많은 컴퓨터 과학자들이 그런 알고리즘은 없고 따라서 왜 이 절에서 언급한 근사치 알고리즘들이 더 연구되어야 하는지에 대한 이유에 대해 공부할 것이다.

연습문제

1. N^5 스텝이 걸리는 알고리즘과 2^N 스텝이 걸리는 알고리즘 중 어느 것을 선택하는 것이 좋을까?
2. 29장의 “미로” 그래프는 Hamilton 사이클을 갖는가?
3. 정점A 대신 정점B에서 시작하는 샘플 그래프 상에서 Hamilton 사이클을 찾을 때 그림 44.4에 해당하는 트리를 그려라.
4. 모든 노드가 정확히 다른 두 노드들과 연결돼 있는 그래프에서, 한 Hamilton 사이클을 찾는데 걸리는 완전-검색 시간은 얼마나 길까? 모든 노드가 정확히 세 개의 다른 노드들과 연결돼 있을 경우에 대해서도 검색 시간을 찾아보라.
5. 수열생성 프로시저에 의해(V 의 함수로서) 얼마나 많은 수의 $visi$ 호출이 행해지는가?
6. 주어진 프로그램에서 재귀적이 아닌 순열생성 프로시저를 유도하라.
7. 두 주어진 인접-행렬이 동일한 그래프를 나타내는지(정점 이름이 다른 것만 제외하고)를 결정하는 프로그램을 작성하라.
8. 물건들의 크기가 실수일 때, 42장의 배낭문제를 푸는 프로그램을 작성하라.
9. 평면에서 교차하는 선분이 없이 N 개의 정점의 집합을 갖는 스패닝트리의 수를 세는 프로그램을 작성하라.
10. 16개의 샘플 지점에 대해 Euclid의 외판원 문제를 풀라.

45 장

NP-완전 문제들

이 책에서 공부해온 알고리즘들은 일반적으로 실제적인 문제들을 해결하는데 사용되므로 합리적인 양의 자원들을 소모한다. 대부분 알고리즘들이 실제로 유용함은 분명하나, 많은 문제들 각각의 경우에 알맞은 알고리즘을 선택하기에는 너무 많은 알고리즘들이 있다. 안타깝게도, 전 장에서 언급한 바와 같이, 실제로으로는 이러한 효율적인 해결책들을 받아들일 수 없는 많은 문제점들이 발생한다. 게다가 더욱 나쁜 것은, 많은 그런 종류의 문제들에 대해 효율적인 알고리즘이 존재하는지의 여부조차 모르고 있다는 것이다.

이런 상황은 많은 분야의 문제에 대해 어떤 효율적인 알고리즘도 발견하지 못한 프로그래머나 알고리즘 디자이너 뿐만 아니라, 왜 이런 문제들이 어려운가에 대한 어떤 이유도 찾을 수 없었던 이론가에게도 극도의 당혹감을 가져다 주었다. 아주 많은 연구가 이 분야에서 행해져 왔고, 새로운 문제들이 기술적인 의미에서 옛날의 문제들만큼 해결이 어려운 것으로 분류될 수 있게 해주는 개발을 가져왔다. 이런 연구의 많은 이 책의 본서의 범위를 벗어나기는 하지만, 이 중 특정 이론은 공부하기에 그리 어렵지 않다. 효율적인 알고리즘인지를 알 수 없는 그런 종류의 문제들에 대해 평가를 내려야 할 경우, 이들 이론이 유용하게 활용될 것이다.

종종 어려움과 쉬움의 차이는 사소할 수 있다. 예를 들면, 우리는 31장에서는, 문제에 대한 효율적 알고리즘을 연구했었다. “주어진 가중치가 부여된 그래프에서, 정점 x 에서 y 로 가는 최단 경로를 찾아라” 그러나 만약 정점 x 에서 y 로 가는 사이클이 없는 최장 경로를 찾도록 요구한다면, 어느 누구도 모든 경로를 조사하는 방법보다 더 좋은 해결책을 알 수가 없을 것이다. 단지 “예-아니오”의 답만을 요구하는 유사한 문제들의 경우에는 더욱 그렇다.:

Easy : x 에서 y 로 가는 가중치 $\leq M$ 인 경로가 있는가?

Hard(?) : x 에서 y 로 가는 가중치 $\geq M$ 인 경로가 있는가?

깊이-우선검색은 선형시간(linear time) 내에 첫 번째 질문에 대한 해결책을 알려준다. 그러나 두 번째 질문에 관련하여 알려진 모든 알고리즘들은 지수-시간(exponential time)이 걸릴 수 있다.

“지수-시간이 걸릴 수 있다”라고 하는 것보다는 좀 더 정확히 설명할 수도 있으나, 현재는 그런 것은 크게 문제되지 않는다. 일반적으로, 입력크기가 N 인 경우, 지수-시간 알고리즘은 적어도 2^N 에 비례하는 시간이 걸린다고 생각함이 좋다.(의논해야 할 결과들의 실체는, 만약 2가 1보다 큰 임의의 수 a 로 대체되더라도, 변화가 없다) 이 말은, 예를 들자면, 지수-시간 알고리즘은 크기가 아주 큰(100이상인) 문제에 대해서는 동작을 보장할 수 없다. 왜냐하면 어느 누구도 알고리즘이 2^{100} 스텝의 수행을 하는 동안 기다릴 수 없기 때문이다. 지수적 증가는 기술적 진보를 미미하게 만든다: 슈퍼컴퓨터가 주판보다 일백억 배가 빠르다 하더라도, 어느 누구도 2^{100} 스텝은 수행해야 하는 문제의 해결에 도달하기 어려울 것이다.

확정적 및 비확정적 다항식-시간 알고리즘

(Deterministic and Nondeterministic Polynomial-Time Algorithms)

이제까지 공부해온 효율적인 알고리즘들과 각 가능성을 모두 검사하는 “지수-시간” 알고리즘들 간의 큰 성능 차이는 한 단순한 공식적 모델을 가지고 이들 간의 인터페이스를 연구하는 것을 가능하게 했다. 이 모델에서, 한 알고리즘의 효율은, “합당한” 코드와 알고리즘을 사용하여, 입력을 코드화하는데 사용되는 비트들의 수에 대한 함수이다.(“합당한”이라는 의미의 정확한 정의는 컴퓨터에 대한 코드화에 관련된 모든 일반적인 방법들을 포함한다: 합당치 않은 코딩 방식의 한 가지 예는 단항(unary)이다. 여기서 숫자 M 을 나타내는데 사용된 M 비트들은 $\log M$ 에 비례해야한다. 우리의 관심사는 단지 어떤 다항식에 비례하는 시간 내에 수행이 보장되는 입력 비트 수에 대한 알고리즘들을 알아내는 것이다. 이러한 알고리즘에 대해서, 해결할 수 있는 문제는 P 에 속한다고 말한다. 여기서, P 는 다항식-시간 내에 확정적(deterministic) 알고리즘이 해결할 수 있는 모든 문제들의 집합이다.

“확정성”이란 어느 시점에서 알고리즘이 무엇을 하든 간에 다음번 할 수 있는 일이 유일한 경우를 의미한다. 이 개념은 프로그램들이 실제 컴퓨터들 상에서 수행되는 방법을 포함한

다. 다항식이란 말은 여기에 전혀 명시되어지지 않았고, 이 정의는 명백히 이제까지 우리가 공부해온 표준 알고리즘들을 망라함을 주목하라. 정렬(sorting)은 P 에 속한다. 왜냐하면, 예를 들어, 삽입정렬(insertion sort)은 N^2 에 비례하는 시간 내에 수행되기 때문이다. ($N \log N$ 삽입정렬 알고리즘들이 있음은 이 경우와는 무관하다) 또한 한 알고리즘에 의해 걸리는 시간은 사용된 컴퓨터에 따라 다르다. 그러나, 다른 컴퓨터를 사용하는 것은 단지 다항식의 인자만을 변화시키므로 수행시간에 약간의 차이만을 가져온다. 그래서, 이 또한 여기서 말하고자 하는 것과 별 관련이 없다.

물론, 논의중인 이론적 결과는, 우리가 작성하고 있는 모든 수행 문장들이 증명될 수 있는, 완전하게 규정된 연산모델에 근거하여야 한다. 여기서 우리의 의도는 중심사상의 일부를 알아보자는 것이지, 엄격한 정의나 정리들을 만들자는 것이 아니다. 독자들은 명백한 논리적 오류들이 이론 그 자체보다는 설명의 비공식적인 성질에 기인한다고 확신할 수 있다.

컴퓨터의 성능을 증가시키는 한 가지 “비합리적인” 방법은 “비확정성(nondeterminism)”의 힘을 빌리는 것이다: 한 알고리즘은, 여러 가지 선택사항 중의 하나를 선택해야 할 경우에, 올바른 선택을 할 수 있는 능력을 가진다. 아래에서 논의하기 위해, 비확정적 상태기에 대한 알고리즘을 한 문제에 대한 해결책을 “추측”하는 것으로서 간주하고, 그것이 올바른지를 검증 해볼 수 있다. 20장에서 어떻게 비확정성이 알고리즘 설계를 위한 도구로서 유용한지를 배웠다; 여기서는 이들 문제들을 분류하기 위한 논리적 장치로 사용하기로 한다. 우리는 NP를 다항식-시간 내에 비확정성 알고리즘에 의해 해결될 수 있는 모든 문제들의 집합으로 정의한다.

명백히, P 에 속하는 모든 문제는 또한 NP에 속한다. 그러나 NP에 속하는 많은 다른 문제들이 있는 듯하다: 한 문제가 NP에 속함을 보이기 위해서는, 주어진(추측으로 얻어진) 해결책이 맞다는 것을 조사하기 위한, 다항식-시간 알고리즘을 찾을 필요가 있다. 예를 들면, 최장경로 문제의 “예-아니오” 버전은 NP에 속한다. NP에 속하는 또 한 가지 예는 만족성(satisfiability)문제이다. 주어진 다음형태의 공식이 있다고 하자.

$$(x_1 + x_3 + x_5) * (x_1 + \bar{x}_2 + x_4) * (\bar{x}_3 + x_4 + x_5) * (x_2 + \bar{x}_3 + x_5)$$

여기서, x_i 들은 부울변수(참 또는 거짓)를 나타내고, “+”는 or, “*”는 and, 그리고 x' 는 not을 의미한다. 만족성 문제는 공식을 참으로 만드는(즉, 만족시키는) 변수들에 대한 진리(truth)값들의 할당이 있느냐 없느냐를 결정하는 것이다. 이 문제가 이론 분야에서 아주 중요한 역할을 함을 아래에서 알 수 있다.

비확정성은 중요하게 여겨도 좋은 강력한 연산이다. 왜 어려운 문제들을 쉽게 해주는 이러한 이상적인 도구를 사용하기를 주저하는가? 그 이유는 비확정성이 강력해 보이는 만큼 어느 누구도 특정문제를 해결하는데 그것이 도움이 되었다는 것을 증명할 수 없기 때문이다. 다시 말해서, 어느 누구도 NP에 속한다고 증명된 어느 문제가 P에는 속하지 않는다는 것을 찾을 수 없었다는 것이다.(또는 심지어 그런 것이 존재한다는 것조차도) 우리는 현재 $P = NP$ 가 성립하는지 알지 못한다. 이러한 상황은 참으로 당혹스러운 것이다. 왜냐하면, 많은 중요한 실제적 문제들은 NP에 속하나,(이들 문제는 비확정적 상태기에서 쉽게 해결될 수 있다) P에 속하는지 아닌지를 잘 알 수가 없기 때문이다.(우리는 어느 확정적 상태에서 이들에 대한 효율적인 알고리즘들이 있는지를 알지 못하고 있다) 만약 문제가 P에 속하지 않는다는 것을 증명할 수 있다면, 그 문제에 대한 효율적 해결을 위한 검색을 하지 않을 것이다. 그러나, 이런 증명을 할 수 없는 경우에는, 어떤 효율적인 알고리즘이 발견되지 않았을 가능성이 있다. 실제로, 우리가 현재 아는 바로는, NP에 속하는 모든 문제에 대해 어떤 효율적인 알고리즘이 있을 수 있다. 이 말은 많은 효율적인 알고리즘들이 아직 발견되지 않았다는 의미이다. 거의 모든 사람이 $P = NP$ 라고 믿고 있으나, 한편으로는 이것이 사실이 아님을 증명하기 위해 상당한 노력이 투자되어 왔다. 그러나, 이 문제는 컴퓨터 과학분야에서는 아직 미해결 연구과제로 남아있다.

NP-완전성(NP-Completeness)

여기서는 NP에는 속하는 것으로 알려진, 그러나 P에 속하는지 아닌지는 알지 못하는 일련의 문제들을 살펴보기로 한다. 즉, 이들 문제들은 비확정적 상태기에서는 풀기가 쉽다. 그러나 상당한 노력에도 불구하고, 아무도 이들 문제중 어느 하나에 대해서도 기존 상태기에서 유용한 알고리즘을(또는 그런 알고리즘이 없음) 찾아낼 수 없었다는 점이다. 이들 문제들은 $P \neq NP$ 라는 확실한 증거를 보여주는 부가적인 성질들을 가진다: 만약 이들 중 어느 하나라도 확정적 상태기에서 다항식-시간내에 해결될 수 있다면, NP내에 속하는 모든 문제들도 다항식 시간 내에 풀 수가 있다.(즉, $P = NP$ 이다) 즉, 연구자들이 모든 이러한 문제에 대한 효율적인 알고리즘을 찾지 못한 것은 $P = NP$ 임의 증명을 못한 것으로 간주할 수 있다. 이런 문제들을 NP-완전 문제라고 한다. 많은 관심거리가 되는 실질적 알고리즘들이 NP-완전 문제들인 것으로 확인됐다.

NP-완전 문제임을 증명하는데 사용되는 주된 도구는 다항식 축약(polynomial reduction)이다. NP에 속하는 새 문제들을 풀기 위한 어떤 알고리즘도 다음 절차에 따라 일부 알려진 NP-완전 문제를 푸는데 이용될 수 있음을 보여준다: NP-완전 문제로 알려진 문제를 새 문제로 변환하고, 주어진 알고리즘을 이용하여 그 문제를 해결하고, 그 해결책을 다시 NP-완전 문제의 해결책으로 변환한다. 이와 유사한 예를 본 적이 있다. 거기서는 양분매칭(bipartite matching) 문제를 네트워크-흐름문제로 축약시켰다. “다항식-시간내에 축약될 수 있는 (polynomially reducible)”이라는 말은 모든 변환이 다항식 시간 내에 행해질 수 있음을 의미한다. 이같이, 새로운 문제에 대한 다항식-시간 알고리즘이 존재함은 NP-완전 문제에 대해 다항식시간 알고리즘이 존재할 수도 있음을 뜻한다. 그리고, 이 말은(정의에 따르면) NP내의 모든 문제에 대해 다항식-시간 알고리즘이 존재할 수 있음도 뜻한다.

축약의 개념은 알고리즘을 분류하는데 유용한 메커니즘을 제공한다. 예로서, NP에 속하는 한 문제가 NP-완전임을 증명하기 위해, 단지 알려진 NP-완전 문제가 그 문제로 다항식 시간내의 축약이 가능함을 보여주기만 하면 된다: 즉, 새로운 문제에 대한 다항식시간 알고리즘은 NP-완전 문제의 해결에 이용될 수 있다는 말이다. NP-완전 문제는, 다시 NP에 속한 모든 문제들을 푸는데 이용될 수 있다. 축약의 예로서, 44장에서 다루었던 아래 두 문제를 다시 보자.

- 외판원 문제: 주어진 두 도시간의 거리들의 집합을 가지고, 거리가 M 이하인 모든 도시들을 경유하는 경로를 찾아라.
- Hamilton 사이클: 주어진 그래프에서, 모든 정점들을 포함하는 단순 사이클을 찾아라.

만약, Hamilton 사이클 문제가 NP-완전 문제임을 아는 상태에서, 외판원 문제 또한 NP-완전 문제인지 아닌지를 알고자 한다 하자. 외판원 문제의 해결을 위한 어떤 알고리즘도, 축약을 통하여, Hamilton 사이클 문제를 해결하는데 이용될 수 있다: 주어진 Hamilton 사이클 문제의 한 경우(그래프)를 가지고 외판원 문제의 한 경우(도시들의 한 집합, 두 도시들 간의 거리)를 다음과 같이 구성하라: 외판원문제의 도시들에 대해서는, 그래프의 정점들의 집합을 사용하라; 각 두 도시간의 거리에 대해서는, 만약 그래프의 해당 정점간에 선분이 있으면 1을 쓰고, 없으면 2를 써라. 그리고 나서, 외판원 문제에 대한 알고리즘이 그래프상의 정점의 수가 N 이하인 거리를 갖는 경로를 찾게 하라. 이 경로는 정확히 Hamilton 사이클에 상응해야만 한다. 외판원 문제에 대한 효율적인 알고리즘은 또한 Hamilton 사이클에 대한 효율적

알고리즘 일수가 있다. 즉, Hamilton 사이클 문제는 외판원문제로 축약된다. 그래서, Hamilton 사이클이 NP-완전 문제임은 외판원 문제가 NP-완전 문제임을 뜻한다.

Hamilton 사이클 문제의 외판원 문제로의 축약은 상대적으로 단순하다. 왜냐하면, 두 문제들이 유사하기 때문이다. 실제로, 다항식-시간 내의 축약들은 아주 복잡하고 상이하게 보이는 문제들 간에도 행해질 수 있다. 예를 들면, 만족성 문제를 Hamilton 사이클 문제로 축약하는 것이 가능하다. 자세한 설명대신 증명의 개요에 대해서만 설명하기로 한다.

Hamilton 사이클 문제에 대한 다항식-시간 해결책이 있다면, 다항식-시간 내의 축약을 통해 만족성 문제에 대해서도 다항식-시간 해결책을 얻으려 할 것이다. 증명은 주어진 만족성 문제의 한 예(부울 공식)를 가지고, 한 그래프가 Hamilton 사이클을 가짐을 아는 것이 우리에게 부울 공식이 만족스러운지를 알려주는 성질을 갖는 Hamilton 사이클 문제의 한 예(그래프)를 형성하는 자세한 방법으로 구성된다. 그래프는 단지 두 방법 중(변수들의 참 또는 거짓에 해당하는)의 하나에서 한 단순 경로에 의해 운행되는(변수들에 상응하는) 작은 원소들로 구성된다. 이들 작은 원소들은 함께 결합되어 구문(clause)들로 규정되는데, 이를 위해, 그 구문들의 참 또는 거짓에 해당하는 단순경로들에 의해 운행 될 수 있는 복잡한 서브그래프(subgraph)들을 사용한다. 간단한 설명으로 이과정의 전체 구성을 설명하기는 너무도 어렵다. 이런 형태의 자세한 증명을 하는 것 또한 하나의 도전이라고 할 수 있다.(이 분야의 전문가들조차도 축약을 위해 만들어진 특별한 “장치”를 적용하는 편이다) 여기서 우리가 말하고자 하는 관점은 다항식-시간 축약이 아주 상이해 보이는 문제들 간에도 적용될 수 있다는 것이다.

외판원 문제에 대해 다항식-시간 알고리즘을 가질 수 있다면, Hamilton 사이클 문제에 대해서도 다항식-시간 알고리즘을 가질 수 있을 것이다. 이는 또한 만족성 문제에 대해서도 다항식-시간 알고리즘을 갖는 것이 가능할 수 있음을 뜻한다. NP-완전 문제로서 증명된 각각의 문제는 또하나의 앞으로 생길 NP-완전 문제를 증명하기 위한 토대를 제공한다. 그 증명은 위에 설명한 Hamilton 사이클을 외판원 문제로 축약하는 것처럼 간단할 수도 있고, 만족성 문제를 Hamilton 사이클 문제로 축약하는 것처럼 복잡하고 어렵거나, 또는 그 중간일수도 있다. 말 그대로, 아주 다양한 영역의 수천개의 문제들이 한 문제를 또 다른 문제로 (위에 언급한 방식으로) 변환시킴으로서 NP-완전 문제들이 증명되었다.

쿠크(Cook)의 정리

축약 방식은 한 문제의 NP-완전 다른 문제의 NP-완전성을 의미함을 이용한 것이다. 그러나 이 방식이 사용될 수 없는 한 가지 경우가 있다: 어떻게 첫 번째 문제가 NP-완전 문제임을 알 수 있을까? 1971년 S. A. Cook 가 이를 찾아냈다. Cook은 만족성 문제가 비다항식-완전성 문제임을 직접 증명했다: 만족성 문제에 대한 다항식-시간 알고리즘이 있다면, NP내의 모든 문제들은 다항식-시간내에 해결될 수 있다.

이 증명은 매우 복잡하나 일반적인 증명방법은 대략 설명할 수 있다. 우선, NP내의 임의의 문제를 해결할 수 있는 기계의 수학적 정의를 만들어 낸다. 이 기계는 입력을 읽고, 특정 연산을 하고, 출력을 하는 Turing 기계(Turing machine)로 알려진 컴퓨터의 한 단순화된 모델이다. Turing 기계는 한 컴퓨터가 할 수 있는 연산을 동일한 양의 시간(다항식-인자 내에서)을 이용하여 수행할 수 있고, 또한 수학적으로 이 과정을 간략하게 설명할 수 있는 장점이 있다. 또한 Turing 기계는 비확정성의 능력을 이용하여 NP내의 어떠한 문제도 해결할 수 있다. 증명의 다음 스텝은 만족성 문제에 나타나는 것 같은 논리적 공식들의 관점에서, 어떻게 명령어들이 수행되는가를 포함한 기계의 각 특징을 설명하는 것이다. 이같이하여, NP내의 모든 문제와(비확정적 Turing 기계 상에서 프로그램으로써 설명될 수 있는) 만족성 문제의 어떤 경우(그 프로그램의 논리적 공식으로의 변경)간에 대응이 성립된다. 이제 만족성 문제에 대한 해결은 근본적으로 주어진 입력에 대해 주어진 프로그램을 수행하는 기계의 모의실험에 해당한다. 그러므로, 주어진 문제의 한 경우(예)에 대한 해결책이 산출된다. 이 증명에 대한 자세한 설명은 이 책의 범위를 넘어간다. 다행히도, 이러한 증명은 단지 한 번만 필요하다: NP-완전 문제의 증명은 축약방식을 사용하여 하는 것이 쉽다.

일부 NP-완전 문제들

앞서 언급한 것처럼, 실제로 수천의 다양한 문제들이 NP-완전 문제들로서 알려졌다. 이 절에서는, 연구되어온 다양한 문제들을 예시하기 위해 일부를 나열한다. 물론, 만족성, 외판원, Hamilton 사이클, 그리고 최장 경로 문제들을 먼저 나열한다. 아래 나열된 문제들은 그 외의 대표적인 문제들이다.

- 분할(PARTITION): 주어진 정수의 집합을 가지고, 그 합이 동일한 두 개의 부분집합으로 분할될 수 있는가?

- 정수 선형 프로그래밍(INTEGER LINEAR PROGRAMMING): 주어진 선형 프로그램에서, 정수해가 존재하는가?
- 다중프로세서 스케줄링(MULTIPROCESSOR SCHEDULING): 두 동일한 프로세서 상에서 수행될 수행시간이 가변인
- 타스크들의 집합과 한 데드라인(deadline)이 주어졌을 때, 타스크들이 모든 데드라인을 만족시키도록 배열될 수 있는가?
- 정점커버(VERTEX COVER): 주어진 그래프와 정수 N 을 가지고, 모든 선분들을 접촉(touch)하는 N 개 정점보다 작은 수의 집합이 있는가?

이러한 많은 연관된 문제들은 중요한 실제 응용분야들이 있다. 그리고, 이들의 해결을 위해 언젠가는 좋은 알고리즘들이 찾아져야 할 강한 동기가 있다. 이들 어느 문제들에 대해서도 어떤 좋은 알고리즘이 찾아질 수 없었다는 사실은 $P \neq NP$ 인 강력한 증거가 된다. 또한 대부분의 연구자들은 확실히 그렇다고 믿고 있다.(이와는 달리, 어느 누구도 이들 문제들이 P 에 속하지 않는다는 것을 증명할 수 없었다는 사실은 $P \neq NP$ 가 아닐 수도 있다는 것을 추론하게도 한다) $P = NP$ 이든 아니든 간에, 현재 우리가 아는 사실은 NP-완전 문제의 효율적인 해결을 보장하는 어떤 알고리즘도 없다는 것이다.

전 장에서 지적한 바와 같이, 여러 기법들이 이런 상황을 다루기 위해 개발되어 왔다. 왜냐하면 이런 다양한 문제들에 대해, 실제로 어떤 종류의 해결책이든 찾아져야 하기 때문이다. 한 가지 방법은 문제를 변형하여 최적치에 가까운 값을 찾아주는 “근사치(approximation)” 알고리즘을 찾는 것이다:(안타깝게도 이 방식은 종종 NP-완전 문제가 된다) 또 하나의 방법은 “평균시간”성능에 의존하여 어떤 경우에서의 해결책을 찾는 알고리즘을 만드는 것이다. 그러나, 이 알고리즘은 모든 경우에 대해서는 잘 동작할 수 없다. 즉, 모든 경우에 대해 잘 동작하는 것을 보장하는 알고리즘을 찾는 것은 가능하지 않더라도, 실제상황에서 일어나는 거의 모든 경우에 잘 동작하는 알고리즘은 찾을 수 있다는 것이다. 세 번째 방식은 효율적인 지수-시간 알고리즘을 전 장에서 설명한 역추적 방식과 함께 사용하는 것이다. 마지막으로, 이론상으로는 언급 안된 다항식-시간과 지수-시간간의 상당한 격차가 있다. $N^{\log N}$ 또는 $2^{\sqrt{N}}$ 에 비례한 시간으로 수행하는 알고리즘은 어떠한가?

이 책에서 공부해온 모든 분야는 NP-완전 문제와 연관되어 있다: 정렬 및 검색, 문자열 처리, 기하학, 그래프 처리 같은 수치응용분야에는 많은 NP-완전 문제들이 있다. NP-완전성

이론의 가장 중요한 실제적 공헌은 이것이 다양한 분야에 있는 한 새로운 문제가 “쉬우냐” 또는 “어려우냐”를 찾을 수 있는 메커니즘을 제공했다는 것이다. 만약 한 새로운 문제를 풀어주는 효율적인 알고리즘을 찾을 수 있다면, 더 이상 어려움이 있을 수 없다. 만약 아니라면, 문제가 NP-완전 문제임의 증명은 적어도 우리에게 효율적 알고리즘을 만드는 것이 정말로 근사한 업적이라는 것을 알려준다.(그리고 또다른 방식도 시도해 볼 만하게 된다) 이 책에서 공부한 20여개의 효율적인 알고리즘들은 우리가 Euclid 이후 효율적 알고리즘들을 아주 많이 연구해 왔다는 증거이다. 그러나, 비다항식-완전성 이론은 아직 우리가 공부할 것이 아주 많음을 알려주고 있다.

연습문제

1. 주어진 가중치 그래프에서 노드 x 에서 y 로 가는 최장 단순 경로를 찾는 프로그램을 작성하라.
2. 만약 $P \neq NP$ 라면, $N \log N$ 의 평균시간 내에 한 NP-완전 문제를 푸는 알고리즘이 있을 수 있는가?
3. 분할(partition) 문제를 풀기 위한 비확정적 다항식-시간 알고리즘을 구하라.
4. 그래프상의 외판원 문제에서 즉시 Euclid의 외판원 문제로 다항식-시간 축약을 할 수 있는가? 또는 이의 역은 가능한가?
5. 1.1^N 에 비례하는 시간에 외판원 문제를 풀 수 있는 프로그램의 중요성은 무엇인가?
6. 본문에 주어진 논리공식은 만족(satisfiable)스러운가 ?
7. 완전-병렬인 “알고리즘 기계” 중의 하나가 만약 $P \neq NP$ 라면, 다항식-시간 내에 NP-완전 문제를 해결하는데 사용될 수 있는가?
8. “ 2^N 의 정확한 값을 계산하라”는 문제가 어떻게 $P - NP$ 분류기법에 해당될 수 있는가?
9. 방향성 그래프에서 Hamilton 사이클을 찾는 문제가 NP-완전 문제임을 증명하라. 무방향성그래프의 경우, Hamilton 사이클 문제는 NP-완전 문제임을 이용하라.
10. 두 문제가 NP-완전 문제임을 안다고 가정하자. 만약 $P \neq NP$ 라면, 이것이 한 문제에서 또다른 한 문제로의 다항식-시간 축약이 있음을 뜻하는가?

Glossary(용어풀이)

- Abstract data type(ADT) - 추상적 데이터 형
Access - 액세스, 접근
Accumulator - 누산기
Adapt - 적응(하다), 순응(하다)
Adaptable - 적응할 수 있는, 융통성 있는
Adaptive - 적응하는, 적응적, 순응하는, 순응적
Adjacency matrix - 인접행렬
Alphabet - 알파벳(벨)
Analog - 아날로그
Approach - (접근)방식
Approximation - 근사, 근사치
Arithmetic - 산술연산
Array - 배열
Articulation point - 연결(접합)점
Ascending (or increasing) order - 오름차순
Asymptotic results - 점진적(근사) 결과
Attribute - 애트리뷰트(DB 에서는 이를 '속성' 이라고도 함)

Backtracking - 역추적
Backward path - 역방향 경로
Balanced Tree - 균형 트리
Balanced multiway merging - 균형 다중 방법 트리
Base - 기저
Batch - 묶음, 일괄
Batch processing - 일괄 처리
Bias - (전기,전자)바이어스, (일반)편견, 선입견
Biconnected - 이중연결된

Biconnectivity - 이중연결성
Binary Search - 이진 검색
Binary tree - 이진 트리
Bipartite - 양분
Bit - 비트
Bitmap - 비트맵
Bottom-up - 상향식
Bound - 경계값
Blank - 공백, 공간(공간 문자), 빈칸
Branch-and-bound (technique) - 분기-한계(기법)
Breath-first search - 너비-우선 탐색
Bridge hands - 브리지 핸드
Brute-force - 주먹구구식(의)
Bubble sort - 버블 정렬
Built-in function - 내장 함수
Bug - 버그
Byte - 바이트

Call/Return - 호출/리턴, 복귀, 되돌림
Canonical form - 정규형
Capacity - 용량
Cartesian - 데카르트의, 카티잔
Catalog - 목록, 카타로그
Chain rule (for differentiation) - (미분에서의) 연쇄법칙
Character - 문자
Circular - 환상형
Circumvent - 앞지르다, 우회하다, 함정에 빠뜨리다, 포위하다
Class - (프로그램의 경우)클래스, (일반적으로)계층, 계급
Classical - 고전적(의), 정통(의), 모범(의)
Closed interval - 폐구간
Closed path - 폐경로
Closest-points - 가장 인접한-점들
Closest-pair - 가장 인접한-쌍
Closure - 폐쇄, 폐포
Cluster - 집단, 클러스터

Code - 코드, 부호
 Coefficient - 계수
 Column-major - 열-우선
 Collision - 충돌
 Collision-resolution - 충돌 해결책
 Complete binary tree - 정(완전) 이진 트리
 Complete graph - 완전 그래프
 Complexity - 복잡도
 Complex number - (수학)복소수
 Complex roots of unity - 복소수 단위근들
 Compound Method - 합성법
 Computation - 계산
 Compression - 압축
 Computational complexity - 계산 복잡도
 Conjecture - 추측
 Concave - 오목(한)
 Concatenation - 결합
 Concrete - 구체적인
 Connectivity - 연결성
 Constant - 상수
 Constraint - 제약조건
 Context - 문맥, 전후관계, 상황, 배경
 Context-free - 문맥-자유
 Context-free grammar - 문맥-자유 문법
 Context-sensitive - 문맥의존형
 Convex Hull - 볼록 외곽
 Coordinate - 좌표(계의, 식의)
 Correspondence - 대응(관계), 일치, 상응, 해당
 Corresponding - 상응하는, 해당하는, 일치하는
 Counter - 계수기
 Curve fitting - 곡선 맞춤
 Cycle - 사이클, 순환
 Cryptography, Cryptology - 암호학
 Cryptosystem - 암호 시스템
 Cubic - 삼(3)차식(의), 입체

Cut - 절단(부), 삭감

Cutoff - 절삭

Data - 데이터

Database - 데이터베이스

Deadline - 데드라인

Debugging - 디버깅

Decision-making - 의사결정

Deck - 덱

Decode, Decrypt - 해독(하다)

Degenerate - 퇴보(퇴행)적 또는 퇴보하다, 타락하다

Degree - 차수

Delimiter - 구분문자

Dense - 텐스, 조밀

Depth-first search - 깊이-우선 탐색

Deque - 디큐

Derivation - (수학의)미분, (일반적으로)유도

Derivative - 미분계수

Descending order - 내림차순

Deterministic - 확정적(인)

Device routine - 장치 루틴

Diagram - 다이어그램

Digital search tree - 디지털 검색 트리

Directed graph - 방향성 그래프

Directory - 디렉토리

Distinguish - 구별(된다)

Distribute counting - 분배 계수기

Divide-and-conquer - 분할-정복

Dot product - 내적

Double buffering - 이중 버퍼링

Double hashing - 이중 해싱

Double rotation - 이중 회전

Doubly linked list - 이중 연결 리스트

Dummy node - 가상(모조) 노드

Dynamic programming - 동적 프로그래밍

Edge - 선분
 Equilibrium state - 평형상태
 Encode - 코드화 하다.
 Encoding - 코드화
 Encryption - 암호(화)
 End-of-line - 줄 종료
 End-recursion removal - 재귀 끝 제거
 Entry - 엔트리
 Error - 오차, 오류
 Error-checking - 오류 체크
 Escape character - 확장 문자
 Escape sequence - 확장 순서
 Estimate - 추정치, 평가치
 Exhaustive - 철저한, 완전한
 Exhaustive search - 완전 검색
 Expand - (수학)전개하다, (일반)확장하다
 Exponential time - 지수 시간
 Exponentiation - 지수화
 Extensible hashing - 확장 가능한 해싱
 External node - 외부 노드
 External search - 외부 검색
 External sort - 외부 정렬
 External storage device - 외부(또는 보조)기억장치
 Extract - 추출
 Euclid - 유클리드(Euclid)
 Evaluation - 평가, 계산

 Facility - 편의기능, 편의도구
 Factor - 계수, 요소, 인자, 인수
 Factorial - 팩토리얼
 Fake, False - 거짓
 Feedback - 피드백, 귀환
 Fetch - 인출
 Fibonacci number - 피보나찌 수열
 Field - 필드, 영역

Filter - 필터
File - 파일
Flip - 플립
Flow - 플로우, 흐름
Format - 포맷, 초기화
Formatting - 포매팅
Forward path - 순방향 경로
Forecasting - 예측
Forest - 포리스트
Fourier series - Fourier 급수
Fourier transform - Fourier 변환
Fractal - 프렉탈
Fraction - 분수
Free list - 자유 리스트
Free tree - 자유 트리
Full binary tree - 전 이진 트리

Gallon - 갤런
Gaussian elimination - 가우스 소거법
Garbage collection - 쓰레기 수집
GCD - 최대 공약수
Geometric - 기하(학)의, 기하학적 도형의
Geometry - 기하학, 기하학적 도형 배열
Global - 광의적
Global state - 전체상태
Global variable - 전역변수
Glossary - 용어풀이
Graph - 그래프
Graphics - 그래픽스
Greedy method - 그리디 방법
Grid(Method)- 격자(법)

Hash - 해시
Hashing - 해싱
Hash function - 해시 함수

Head - 헤드
 Header - 헤더
 Heap - 힙
 Heapsort - 힙 정렬
 Height - 높이
 High-level - 고급
 Horner's method - Horner의 방법
 Hybrid - 하이브리드, 혼합
 Hyperplane - 하이퍼 평면

 Identifier - 인식자
 Implicit - 내재된
 Implement - 구현하다
 Incremental - 증분식(의)
 Indefinite integral - 부정적분
 Index - 인덱스, 색인
 Index sequential access - 색인 순차 접근
 Indirect heap - 간접 힙
 Integration by parts - 부분적분
 Interface - 인터페이스
 Infix - 중위 표기식
 Information theory - 정보이론
 Index - 색인, 인덱스
 Insertion sort - 삽입 정렬
 Interface - 인터페이스, 경계(부분)
 Interleave - 삽입
 Internal search - 내부 검색
 Internal sort - 내부 정렬
 Internal path length - 내부 경로 길이
 Interpolation - 보간법
 Interpolation search - 보간 검색
 Intersect - 교차하다, 교집합하다
 Interval - 구간
 Inversion - 도치
 Item - 항목

Iterative method - 반복법

Job scheduling - 작업 스케줄링

Key - 암호, 열쇠, 주된, (DB, 파일처리분야에서는)키

Knapsack problem - 배낭 문제

Knot - 노트

Label - 이름, 명, 명명하다

Labelling - 명명(하는)

Leaf - 잎

Least common ancestor - 최소 공통 선조

Least-square - 최소제곱

Left child - 왼쪽 자식

Left recursion - 왼쪽 재귀

Legal - 정당한, 법률(상)의, 합법의

Letter - 글자, 문자

Level - 레벨, 계층

Level order - 레벨 오더

Linked list - 연결 리스트

Linear congruential method - 선형 조화 방법

Linear list - 선형 리스트

Linear sort - 선형 정렬

Linear time - 선형 시간

Linear probe - 선형 조사

Linear programming - 선형 프로그래밍

Link - 링크, 연결

Linked-list - 연결-리스트

List - 리스트

List node - 리스트 노드

Literal - 리터럴

Load - 로드, 부하

Load factor - 부하요소

Local - 국지적

Locality of reference - 참조의 국지성

Logarithm - 대수

Lookahead - 미리보기

Loop - 루프, 반복

Low-level - 저급

Lowerbound - 하한

Mark - 마크

Mask - 마스크

Match - 매치(하다, 시키다), 필적하다, 조화를 이루다, 일치하다

Matching - 매칭, 부합, 일치

Maximum flow, maxflow

Mechanism - 메카니즘

Memory - 메모리, 기억장치

Merging - 병합

Merging-until-empty - 비어있을때까지 병합

Micro - 마이크로

Minimum cut, mincut - 최소절단

Modulo - 모듈로, 나머지

Modulus - 모듈로 계수

Modeling - 모델링

Multiple - 다중

Multiway - 다중방법

Naive - 고지식한, 미경험적, 순진한

Nanosecond - 나노초

Natural language - 자연언어

Natural logarithm - 자연 대수

Network - 네트워크

Network flow - 네트워크 플로우(흐름)

Node - 노드

Nonlinear programming - 비선형 프로그래밍

Nondeterministic - 비확정적(인)

Nonmatching - 불일치

Nonterminal - 논터미널

Nonterminal node - 논터미널 노드

Non-uniform - 불균등한

Notation - 주해, 기법

Note - 노트, 메모

NP-complete - NP-완전

Numeral - 수치(적), 숫자

Null - 비어있는, 빈

Object - 객체, 목적, 목적물

Open addression - 열린 번지

Operations research(OR) - 과학적 경영 분석

Operator - 조작자(운전자), 연산자

Option - 옵션

Order - 정렬, 순서

Ordered tree - 순서화된 트리

Ordered hashing - 순서화된 해싱

Overflow - 오버플로우, 과잉넘침

Overhead - 오버헤드, 불필요한 연산

Overlap - 일부 겹치다

Overload - 과부하

Pack - 팩

Package - 패키지

Packaging-wrapping - 패키지(꾸러미) 포장

Paradigm - 패러다임, 범례

Paragraph - 절, 단락

Parallel array - 병렬 배열

Parameter - 파라미터, 매개변수

Parent - 부모

Partial/Complete(Full) solution - 부분/완전 해

Parsing - 파싱

Parse tree - 파서 트리

Parser generator - 파서 발생기

Pass - 통로, 통과(하다)

Path - 경로

Path length - 경로 길이

Patricia - 패트리시아
 Pattern - 패턴
 Pattern descriptor - 패턴 기술자
 Pipe - 파이프
 Pivoting - 피보팅(가우스 소거법의 경우)
 Placement - 배치
 Plot - 플롯(하다)
 Plotting - 플로팅
 Plotter - 플로터
 Polygon - 다각형
 Polynormal time - 다항식 시간
 Polyphase merging - 다중 단계 병합
 Pointer - 포인터
 Popup - (스택의) 삭제
 Postfix - 후위 표기식
 Precedence relationship - 선행 관계
 Prefix - 접두어, 접두사, 전위 표기식
 Preorder - 프리오더
 Preprocessing - 선처리
 Print - 프린트하다, 인쇄하다
 Printing - 프린팅
 Priority - 우선순위
 Priority-first search - 우선순위-우선 탐색
 Priority queue - 우선순위 큐
 Production - 생성규칙
 Projection - 투영
 Pruning - 제거
 Procedure - 프로시저, 절차
 Process - 프로세스, 과정
 Property - 성질
 Pseudo - 유사
 Pseudo-random number - 의사(유사) 난수
 Public-key - 공용-키
 Pushdown - (스택의) 삽입, 푸쉬다운

Quadratic - 이(2)차식(의)

Quadrature (method) - 구상(법)

Quasi-random number - 준난수

Query - 질의

Queue - 큐, 대기 행렬

Quicksort - 퀵 정렬

Range - 범위, 영역

Raster - 래스터

Radix exchange sort - 기수 교환 정렬

Radix search - 기수 탐색

Radix search tries - 기수 검색 트라이

Random - 랜덤, 임의, 무작위

Random number - 난수

Recomputation - 재계산

Record - (컴퓨터)레코드, (일반)기록(하다)

Rectangle method - 사각형법

Recurrence - 반복, 재현, 순환, 재발생

Recursive - 재귀적, 재귀

Recursive-descent parser - 재귀적-하강 파서

Red-black tree - 적색-흑색 트리

Reduce - 축약하다

Reduction - 축약, (Compiler의 경우)환원

Redundant - 잉여(의)

Reflexive - 반사적

Register - 레지스터

Regular expression - 정규식

Reorient - 재교육(재배열) 시키다, 순응시키다

Replacement rule - 대체 규칙

Resident - 전문의 수련자

Resolution - 해상도

Retrieve - 탐색

Return - 리턴, 복귀

Reverse polish - 역 폴리쉬

Right child - 오른쪽 자식

Root - (수학) 근, 루트, 근원, 뿌리
Root-finding - 루트찾기
Routine - 루틴
Row-major - 행 우선
Routine - 루틴 , 일과, 일상적인
Run-length - 반복문자-길이

Sample - 표본, 샘플
Sampling - 표본 추출, 샘플링
Scalar - 스칼라
Scan - 스캔, 주사
Scheduling - 스케줄링
Scratch pad - 임시패드
Search - 검색
Seed - 시드
Segment - 세그먼트
Selection - 선택
Selection sort - 선택 정렬
Sentinel - 표지
Seperate chaining - 분리된 연쇄
Sequential - 순차(적)
Sequential access - 순차 접근
Set - 집합 , 설정(하다, 시키다)
Setting - 설정, 설치, 세팅
Setup - 조직, 편제, 기구, 배치
Shaded - 음영이 있는, 그늘진
Shell sort - 셸 정렬
Shift - 쉬프트, 이동
Shift-reduce parser - 이동-감소(쉬프트-리듀스) 파서
Shuffle - 혼합
Sibling - 형제
Sign - 부호, 신호, 서명날인하다
Simple cycle - 단순 사이클
Simple path - 단순 경로
Simulation - 모의실험

Single rotation - 단일 회전
 Singular matrix - 단독 행렬
 Sink - 종착(지)
 Solution - 해결책, 해
 Sophisticated - 세련된, 복잡한
 Sorting - 정렬
 Source - 근원(지), 원천, 원시
 Spanning tree - 스패닝 트리
 Space - 공백, 공간(공간문자)
 Sparse - 스파아스, 희소
 Spline - 스플라인
 Split - 분할(하다)
 Square - 제곱, 자승
 Stable - 안전성
 Stack - 스택
 State - 상태
 Step - 스텝, 단계
 Straight radix sort - 일직선상의 기수 정렬
 Stream - 스트림
 Stream library - 스트림 라이브러리
 String - 문자열, 스트링
 Strongly connected components - 강하게 연결된 요소
 Subinterval - 부구간
 Subroutine - 부분 루틴
 Subset - 부분 집합
 Subtask - 부분 타스크
 Subtree - 부분(서브) 트리
 Successive-squaring - 연속제곱
 Suffix - 접미어, 접미사
 Symbolic differentiation - 심볼릭 미분
 Symbol - (일반적) 기호
 Symbol table - 기호 표
 Symbolic integration - 심볼릭 적분
 Symmetric - 대칭적
 Symmetric order - 대칭적 순서

Systolic array - 시스톨릭 배열

Table - 도표, 테이블, 표

Table address - 표 번지

Tail - 테일

Tap - 탭

Task - 타스크

Taylor series - Taylor 급수

Term - (수학에서는) 항, 용어

Terminal - 터미널

Terminal node - 터미널 노드

Term-by-term - 항-대-항

Text - 텍스트

Text editor - 텍스트 편집기

Top-down - 하향식

Topology - 위상

Transitive - 추이적

Transition - 전이

Trapzoid method - 사다리꼴법

Trie - 트라이

Traverse - 운행하다

Traversal - 운행

Traveling salesman - 외판원

Turing machine - Turing 기계

Trade-off - 교환

Tree - 트리

Tridiagonal - 3선 대각선(의)

Truth - 진리, 참

Tree node - 트리 노드

- Child node - 자식 노드

- Parent node - 부모 노드

- Descendents - 후손 노드

- Ascendents - 조상 노드

Touch - 터치, 접촉

Type - 형, 유형, 형태

Unary - 단항(의)

Underflow - 언더플로우

Undirected graph - 무방향 그래프

Uniform - 균등한

Uniform-find - 찾기-합치기

Upperbound - 상한

User-defined - 사용자-정의의

Vector - 벡터

Vertex - 정점

Version - 버전

Virtual memory - 가상 기억장치

Weakly connected component - 약하게 연결된 요소

Weight - 가중치

Weighted graph - 가중치 그래프

wired-in - (선으로)연결된

Word - 워드

찾아보기

2-3-4 트리	242	Gauss-Jordan 축약법(reduction)	603
2-노드	242	Gauss 소거법(Gaussian elimination)	597
2차원(quadratic)	82	Graham 스캔(The Graham Scan)	403
2차원 배열	21		
3-노드	242	Hamilton 사이클(Hamilton cycle) 문제	694
4-노드	242	Hamilton 사이클	711
		Hoare	132
AVL 트리	256	Horner법칙	584
		Huffman 코드	363
Bland의 방법	689	Huffman 코드화	671
Boyer-Moore 알고리즘	319	Huffman 트리	365
B-tree	294	h-정렬	124
C++	3, 9	inline	110
C	14		
Caesar 암호화 기법	374	Knuth-Morris-Pratt 알고리즘	313
Catalan수	668	Kruskal 방법(Kruskal's Method)	509
Class 계층구조	39		
		Lagrange 보간법	586
Dags	532	lazy 삭제	237
Delaunay 삼각형	453		
Dijkstra의 알고리즘	508	Manhattan 기하학	430
deque	37		
don't-care	353	NP-완전 문제	707, 710
downheap	174	NP-완전성(NP-Completeness)	710
Edmonds-Karp 알고리즘	547	O-기법	84
Eratosthenes의 추출	20		
Euclid의 외판원 문제	704	partition	133
		plaintext	372
FFT	660	Prim 알고리즘	508
Floyd 알고리즘	531, 663	P-방법 병합	205
Ford-Fulkerson 방법	542, 544		

Quicksort	567	강하게 연결된-요소	523, 535
		개선	99
Rabin-Karp 문자열 검색	592	객체 형성 함수(constructor)	493
Rabin-Karp 문자열 검색 방법	567	거짓 시작(false start)	313
Rabin-Karp 알고리즘	323	검색(searching)	217
Romberg 적분법	628	검색 휴리스틱	239
RSA 공개키 암호화 시스템	378	격자법(Grid Method)	417
		결합(Concatenation)	328
Simpson 방법(Simpson's method)	627	결합과 정복	69
Strassen 방법	593	결합 암호(product cipher)	376
		경로(path)	44, 170
Taylor 급수(series)	625	경로 길이	45
Turing 기계(Turing machine)	713	경로 축약	497
template 기법	40	경사(slant)	247
		경험적(heuristic) 규칙	700
upheap	174	경험적 검색 기법(search heuristics)	702
		계산 복잡도	84
Vernam 암호	375	계산 오차(computational error)	604
Vigenere 암호	375	계수기(counter)	348
von Neumann 구조	635	고급(high-level)	341
Voronoi 다각형	450	고급언어	13
Voronoi 다이어그램	443, 450	곡선 또는 데이터 맞춤	609
Voronoi의 쌍대(dual)	453	곱셈(Multiply)	651
		공개키(public-key)	377
Warshall 알고리즘	532, 663	과부하	220
Weierstrass 근사치 정리	610	광의(global)	14
		교환	110, 151
가변문자-길이 코드화	361	구간 검색(range searching)	414
가상(dummy)	22	구상법(quadrature)	621
가상 기억장치	305	구성자	32
가상적 함수	39	구체적 자료 형	39
가시 평면(viewing plane)	429	국지성	212
가장 인접한 쌍 문제	444	귀납법	48
가장 인접한 이웃 문제	443	균등한(uniform) 난수	566
가중균형(weight balancing)	497	균형(balancing)	241
가중치(weighted) 그래프	462	균형 트리	241
가중치 그래프	501	균형화된 다중 방법 병합	203
가중치 내부-경부-길이	671	그래프	44
가중치 매칭 문제	552	그래프 동형(graph isomorphism) 문제	482
간접 정렬	110, 123	근(root)	44
간접 히프	181	근사치	87

근사치 알고리즘	703	다이아그램	116
근화된 트리	47	다중 검색	325
기계어	13	다중 단계 병합	210
기수 검색 트라이	281, 369	다중 방법(multiway)	45
기수 검색	275	다중 방법(multiway) 병합	186
기수 교환 정렬	153	다중 방법 기수 검색	282
기수 정렬	151	다차원 구간 검색	426
기억 장소	21	다항식 곱(Polynomial Multiplication)	587
기억 장치 계층구조	202	다항식 보간법(polynomial interpolation)	585
기저(base)	82	다항식-시간내에 축약될 수 있는	711
기저(basis)변수	685	다항식 연산(Polynomial Arithmetic)	580
기저열(basis column)	685	다항식 축약(polynomial reduction)	711
기하학	3	다항식 평가법 및 보간법	583
기하학적 교차	429	단순 경로(simple path)	461
기호 표	217	단순 연결(simple connectivity)	526
깊이-우선 검색	468, 524	단순 폐경로(Simple Closed Path)	390
꼬리	187	단일 연결 리스트	52
끝의 재귀 제거	141	단일 회전	250
		대수(logarithmic)	82
나노	86	대체 규칙(replacement rule)	342
나머지(modula)	12	대체 연결선(alternate connection)	488
난수(random number)	565	대체 연산	166
난수 발생기	260	대칭적 순서	56
난수 생성기	566	데이터 베이스	292
난수 순서(random sequence)	565	데이터 형	12
내부(internal) 노드	45	덱(deck)	335
내부(internal) 정렬	108	덴스(dense)	462
내부 경로 길이	234	도치(inversions)	119
내부 노드	230	독립된-정점들(isolated vertices)	472
내부제거법	408	동적 검색	228
내장 함수	25	동적 프로그래밍(dynamic programming)	663
내장 함수 프로시저	25	동치류(equivalence class)	490
내적(dot product)	592	두 가지 방법(two-way) 병합	186
너비-우선 검색(Breadth-First Search)	477	디버그	107
네트워크(network)	462	디지털 검색	278
네트워크 흐름 문제	540, 677	디지털 검색 트리	276
노드	44	디지털 트리 검색	276
노트(knot)	611	디큐(deque)	334
논 터미널(nonterminal)	45, 342		
높이(height)	45	레벨(levels)	45
높이균형(height balancing)	497	레벨 오더	57

레코드	217	보간(Interpolate)	651
루트 찾기(root-finding)문제	585	보간 검색	226
		보조 포인터	187
마스크	152	보조 표	153
마이크로(micro)	80	복소수 단위근들	
마지막(final)	330	(Complex Roots of Unity)	653
만족성(satisfiability)문제	709	볼록면체(convexity)	397
매칭	551	볼록 외곽(convex hull)	397
메카니즘	13	부가적 조화 방법	
모든 쌍에 대한 최단 경로 문제	529	(Additive Congruential Method)	571
모의실험(simulation)	26, 567	부동 소숫점	152
모의 실험 시스템	165	부모(parent)	44
모조-키(pseudo-key)	376	부분 적분(integration by parts)	622
목적함수(objective function)	678	부분 트리	45
무방향 그래프(undirected graph)	462	부분-피보팅(partial pivoting)	602
무작위(random)	565	부울리언	41
무작위성 시험(Testing Randomness)	574	부정적분(indefinite integral)	622
문맥 의존형(context-sensitive)	344	부하 요소(load factor)	267
문맥 자유 문법	342	분기-한계(branch-and-bound)기법	700
문법(grammar)	342	분리된 연쇄	263
문자 형	12	분배 계수기	128
미로(Mazes)	480	분배 계수 정렬	159
미리보기(lookahead)	346	분할(splitting)기법	497
		분할 요소	136
바이토닉(bitonic) 병합	642	분할-정복	61
반복 문자-길이 코드화		불균등한(non-uniform) 분포	566
(run-length encoding)	358	불필요한 연산(overhead)	247
반줄임(halving)	497	브리지 핸드	113
방향성(directed) 그래프	462, 523	블록	108, 292
방향성 비순환 그래프	532	비교	151
배낭 문제(Knapsack Problem)	664	비교-교환(compare-and-exchange)	638
배열	19	비 교환 법칙	33
버블 기억장소	202	비디오 디스크	202
버블 정렬	116	비선형 치환(nonlinear substitution)	376
변경 연산	168	비어있을 때까지 병합	210
병렬(parallel) 알고리즘	635	비재귀적 깊이-우선 검색	
병렬 배열	27	(Nonrecursive Depth-First Search)	474
병렬 실행(parallelism)	636	비트	152
병합(merging) 방법	637	비트 연산자	152
병합	188	비 확정적(nondeterminism)	329, 709
병합 정렬(mergesort)	84, 185, 445	빈(null)	330

사각형법(rectangle method)	623	스트링 검색 문제	316
사다리꼴법	625	스트링 검색 컴파일러	316
사이클링 제거정책	687	스파아스(sparse)	462
삼선 대각선(tridiagonal) 행렬	605	스플라인 보간(spline interpolation)	611
상수(constant)	82	시드(seed)	567
상태	317	시스톨릭 배열(Systolic Arrays)	643
상한(upper bound)	80	식별자	218
상향식(bottom-up)	69	실수 형	12
상향식 파싱	348	실행 가능한 기저(feasible basis)	690
색인 순차 접근	292, 294	심볼릭 적분(Symbolic integration)	621, 622
생산 규칙(production)	342	심플렉스 기법(Simplex Method)	677, 683
선분(edge)	44, 460	쓰레기 수집	29
선분 교차(Line Segment Intersection)	388		
선 처리(preprocessing)	320	아나로그(analog)	191
선 처리(preprocessing) 알고리즘	414	안전	109
선택(selection)	145	안전성	109
선택 정렬(selection sorting)	111, 400	안정적 결혼 문제 (Stable Marriage Problem)	556
선형대수	3	알고리즘	3
선형 리스트	38	알고리즘 분석	5
선형 정렬	161	암호문(ciphertext)	372
선형 조화 방법 (Linear Congruential Method)	567	암호 시스템(Cryptosystem)	372
선형-조화-생성기 (linear congruential generator)	573	암호 작성법(Cryptography)	371
선형 프로그래밍	677, 678	암호학(Cryptology)	371
선형-피드백-쉬프트-레지스터	571	암호해독법(Cryptanalysis)	371
세 가지 중의 중위수	144	암호해독 알고리즘	373
수치 계산	165	암호해독 키	372
수치해석	13	암호화(cryptography)	566
순방향-소거	689	양분 그래프(Bipartite Graphs)	553
순방향-소거 단계 (forward-elimination phase)	600	양분매칭(bipartite matching) 문제	711
순서화된(ordered) 트리	45	어셈블리 언어	13
순열 생성(Permutation Generation)	701	엔트리	217
순차적 검색	219	역방향 치환	689
순차적 디스크 검색	292	역추적(Backtracking)	697
셀 정렬	108, 124	역 폴리쉬	32
스택	19	연결된-요소(connected component)	461
스택 삽입	30	연결된-요소들(Connected Components)	486
스트림 라이브러리	11	연결-리스트(linked list)	19, 22, 466
스트링	12, 309	연결점(articulation point)	487
		열	21
		열린 번지(open-addressing)	265

예측(forecasting)	209	이진 트리	45
오류 체크	111	이진 트리 검색	228
완전-검색(exhaustive search)	674, 693	이탈(Digression)	701
완전 그래프(complete graph)	462	인덱스	24
완전-셔플(Perfect Shuffles)	637	인덱스 배열	120
완전-피보팅(full pivoting)	602	인식자(identifer)	49
완전해(full solution)	700	인 오더	56
외부(external) 노드	45	인접-구조(adjacency-structure)	466
외부(external) 정렬	108	인접-리스트	466
외부 경로 길이	77	인접-행렬(adjacency-matrix)	464
외부 기억장치	201	인터페이스	34
외부 노드	230	일괄 처리	186
외부 정렬	201	일반적 선분 교차	437
외판원 문제	390, 693, 711	일직선상의 기수 정렬	153
왼쪽 재귀(left recursion)	348	일회용 패드(one-time pad)	375
요세퍼스(Josephus) 문제	25	입력(reading)	14
우선순위(priority) 그래프-운행 방법	545	입력 스트림	33
우선순위	165	입체(cubic)	82
우선순위-우선 검색		잎(leaves)	45
(Priority-First Search)	504		
우선 순위 큐(priority queue)	165, 479	자기 테이프	201
원문 메시지	372	자료구조	4
원장(master)	293	자식(children)	44
위상 정렬(Topological sorting)	532	자연 대수	83
위상학적 정렬(topological sorting)	523	자유 리스트	28
유전자	218	자유 트리	47
유클리드(Euclid)	10	작업 스케줄링(job scheduling)	165, 459
유한 상태 기계	316	장치	109
윤곽을 그리는(profiling)	81	재귀(recursion)	47, 61
의사난수(pseudo random number)	566	재귀 끝의 제거	73
이동-감소 파서(shift-reduce parser)	349	재귀적 운행(recursive traversal)	414
이산 수학	3	재귀적-하강 파서	
이중 버퍼링	208	(recursive-descent parser)	347
이중연결(biconnectivity)	485	재귀 함수	61
이중 연결 리스트	27	재발생	61
이중 해싱	268	재발생 관계	62
이중 회전	252	저급(low-level)	341
이진 검색(binary searching)	223, 414	적색	246
이진 대수	83	적색-흑색(Red-Black) 트리	246
이진 스트링	309	적응력있는 구상법	
이진 트라이 검색	281	(Adaptive Quadrature)	628

전(full) 이진 트리	46	최소 스패닝 트리	
전산학	4	(Minimum Spanning Tree)	501, 503
전위 순회(preorder traversal)	471	최소 제곱 방법	
전이	317	(Method of Least Squares)	616
정(complete) 이진 트리	46, 169	최악의 경우	79
정규 식(regular expression)	328	추상적 데이터 형	19
정렬	107	추상적 자료형	38
정렬 기계(sorting machine)	638	추상화	38
정렬 알고리즘	107	추이적(transitive)	439
정렬 프로그램	107	추이적 폐포(Transitive Closure)	523, 526
정밀도(precision)	565	추출(extract) 연산	174
정보량	357	축척 인자(scaling factor)	601
정수 형	12	출력(writing)	14
정점(vertex)	44, 460	충돌 해결책	259
정제	99		
제거(pruning)기법	699	컴파일러-컴파일러	353
제거 연산	173	컴파일러(compiler)	341
조사(probe)	265	쿠크(Cook)의 정리	713
종료 조건	61	퀵 정렬	132
주먹구구식(brute-force)	83	큐(queue)	19, 36
주사(scan)	316	큰-O 기법	84
준난수(quasi-random number)	566	클러스팅	268
중앙 처리 장치	201	키	217
중위 순회(inorder traversal)	434	키 분배 문제	377
중위 표기	32	키워드	32
지닌 트리	47		
지수(exponential)	83	터미널(terminal)	342
지수화(exponentiation)문제	584	터미널 노드	45
		테이프	203
참조의 국지성	212	텍스트 스트링	309
찾기-합하기 알고리즘들	490	템플레이트	110
초기(initial)	330	통계적 기하학(stochastic geometry)	410
최단 경로(Shortest Path)	512	투영(projection)기법	416
최대 경계치(upper bound)	704	트라이(trie) 구조	362
최대 공약수	10	트라이	280
최대 매칭(maximum matching)	551	트리(trees)	19, 43
최대-중분법	687	트리 순회	54
최대흐름-최소절단 정리	543		
최소값-찾기(find-the-minimum)계산	400	파괴자	32
최소 경계치(lower bound)	704	파라미터	108
최소 공통 선조	47	파서 트리(parse tree)	43, 343

파스칼(Pascal)	14	하한	85
파싱(parsing)	334, 341	하향식(top-down)	53
패케지(package)	14	하향식 2-3-4 트리	245
패키지 포장(Package-Wrapping)	400	하향식 파싱	345
패턴 일치	310, 327	할아버지(grandparent)	44
패트리시아	284	합성법(Compound Methods)	627
패트리시아 트라이	288	항등행렬(identity matrix)	690
패트리시아 트리	287	해시 테이블(hash table)	463
팩토리얼	62	해시 함수	259
팩화	13	해싱	259
페이지	292	행	21
평가(Evaluate)	651	행렬 연속곱셈(Matrix Chain Product)	667
평균 경우	79	형제(sibling)	44
평면화(planarity) 문제	482	호너(Horner)	261
평형 상태(equilibrium state)	540	혹(Or)	328
폐도형(closed figure)	386	홀(hole)	122
폐쇄(Closure)	328	홀수-짝수(odd-even) 병합	641
포리스트(forest)	45, 461	화살표	25
포스트 오더	57	확률론적 알고리즘	144
포인터	28	확장 가능 해싱	299
표본 추출(sampling)	567	확장순서(escape sequence)	360
표지(sentinel)	114	확정적(deterministic)	329
표지 레코드	220	확정적(deterministic) 알고리즘	708
프랙탈(fractal)	70	확정적 및 비확정적 다항식-시간 알고리즘	708
프로그래밍	3	환상형(circular)	26
프로그램 최적화	99	환상형 리스트	27
프리 오더	54	횡단흐름(crossflow)	543
피보나찌 수열	62	후위 순회(postorder traversal)	449, 471
피보트(pivot)	601	후위 표기	32
피보팅	685	희박한(sparse)	270
필터(filter)	110	히프(heap)	166
하이브리드(hybrid)	265	히프 정렬	175
하이퍼평면(hyperplane)	427	히프 조건	172