NOTES

SUBJECT: DESIGN AND ANALYSIS OF ALGORITHM

SUBJECT CODE: ECS-502

BRANCH: CSE/IT

SEM: V

SESSION: 2014-15

Evaluation Scheme

Subject Code	Name of Subject	Periods			Evaluation Scheme				Subject Total	Credit
		L	T	P	CT	TA	TOTAL	ESC		
ECS-502	Design and Analysis of algorithm	3	1	0	30	20	50	100	150	4

Mr. B.N.Panday, Asst Prof/CSE
Mr. Kapil Tomar, Asst Prof/IT
Mrs. R. Shobana Lakshmi, Asst Prof/IT
AKGEC Gzb

CONTENTS

1.7 Shell sort

UNIT-II ADVANCE DATA STRUCTURE

2.1 Red Black Tree

- **2.1.1Properties of red-black trees**
- **2.1.2 Representation of a Red-Black Tree**
- **2.1.3 Lemma**

2.2 Rotations in Red-Black Tree

2.3 Insertion in Red Black tree

- **2.3.1 Analysis**

2.4 Deletion in Red Black Tree

- **2.4.1 Analysis**

2.5 B-Trees

- **2.5.1 Definition of B-trees**
- **2.5.2 The height of a B-tree**
- **2.5.3 Theorem**
- **2.5.4 Basic operations on B-trees**
  - **2.5.4.1 Searching a B-tree**
  - **2.5.4.2 Creating an empty B-tree**
  - **2.5.4.3 Inserting a key into a B-tree**
  - **2.5.4.4 Splitting a node in a B-tree**
  - **2.5.4.5 Inserting a key into a B-tree in a single pass down the tree**
  - **2.5.4.6 Deleting a key from a B-tree**

2.6 Binomial Heaps

- **2.6.1 Binomial trees and binomial heaps**
- **2.6.2 Binomial trees**
- **2.6.3 Lemma**
- **2.6.4 Representing binomial heaps**
- **2.6.5 Operations on binomial heaps**
  - **2.6.5.1 Creating a new binomial heap**
  - **2.6.5.2 Finding the minimum key**
  - **2.6.5.3 Uniting two binomial heaps**
  - **2.6.5.4 Inserting a node**
  - **2.6.5.5 Extracting the node with minimum key**
  - **2.6.5.6 Decreasing a key**
  - **2.6.5.7 Deleting a key**

UNIT-III Divide and Conquer, Greedy Methods

3.1 Divide and conquer

- **3.1.1. Strassen's algorithm for matrix multiplication**
  - **3.1.1.1 Determining the submatrix products**
- **3.1.2 Finding the convex hull**
  - **3.1.2 .1 Graham's Scan**
  - **3.1.2.2 Jatvis March**
- **3.1.3 Binary Search**

3.2 Greedy Methods

- **3.2.1 Elements of the greedy strategy**
- **3.2.2 Greedy-choice property**
- **3.2.3 Optimal substructure**
- **3.2.4 An activity-selection problem**
  - **3.2.4.1 A recursive greedy algorithm**
  - **3.2.4.2 An iterative greedy algorithm**
- **3.2.5 Task scheduling problem**
- **3.2.6 The Fractional Knapsack Problem**
- **3.2..7 Minimum spanning tree**
  - **3.2.7.1 Growing a minimum spanning tree**
  - **3.7.1.2 Kruskal's algorithm**
  - **3.7.1.3 Prim's Algorithm**
- **3.2.8 Single source shortest path**
  - **3.2.8.1 Relaxation**
  - **3.2.8.2 The Bellman-Ford algorithm**
  - **3.2.8.3 Dijkstra's algorithm**

UNIT-IV Dynamic Programming, Backtracking And Branch And Bound

4.1 0/1 knapsack Problem

4.2 Matrix Chain Multiplication

- **4.2.1 Counting the No. of Parenthesization**
- **4.2.2 Apply Dynamic Programming**
- **4.2.3 Constructing Optimal Solution**

4.3 Longest Common Sequence

- **4.3.1 Characterizing a longest Common sequence**
- **4.3.2 Recursive Solution**
- **4.3.3 Computing length of an LCS**
- **4.3.4 Constructing an LCS**

4.4 All Pair Shortest path

- **4.4.1.Floyd & Warshall Algorithm**

4.4.1.1 Structure of shortest path

4.4.1.2 Recursive Solution to all pair shortest problem

4.4.1.3 Computing the shortest path weights bottom up

4.5 Resource Allocation Problem

4.6 Backtracking

4.6.1 N-Queen Problem

4.7 Graph Coloring

4.8 Hamiltonian Cycle

4.9 Sum of Subset Problem

4.10 Branch & Bound

4.10.1 Least Cost search

4.10.2 Bounding

4.10.3 Travelling Sales Man

UNIT-V Selected Topics

5.1 Fast Fourier transform

5.1.1 Complex roots of unity

5.1.2 The FFT

5.1.3 Complexity

5.2 String matching

5.2.1 The naive string-matching algorithm

5.2.2 The Rabin-Karp algorithm

5.2.3 The Knuth-Morris-Pratt algorithm

5.3 Theory of NP completeness

5.3.1 The partition problem

5.3.2 The Sum of Subset Problem

5.3.3 The Satisfiability Problem

5.3.4 The Minimal Spanning Tree Problem

5.3.5 The Traveling Salesperson Problem

5.4 Approximation algorithms

5.4.1 The vertex-cover problem

5.5 Randomized algorithms

5.5.1 A randomized version of quicksort

SYLLABUS

L T P 3 1 0

Pre-requisites: Data Structure and C programming.

COURSE: B.Tech. **SEMESTER:** V **SECTION:** I

SUBJECT CODE & NAME: ECS-502 DESIGN AND ANALYSIS OF ALGORITHM

Objectives: To study the concepts of design and analysis of algorithm.

Unit-I

Introduction: Algorithms, Analyzing algorithms, Complexity of algorithms, growth of functions, Performance measurements, Sorting and order Statistics -Shell sort, Quick sort, Merge sort, Heap sort, Comparison of sorting algorithms, Sorting in linear time.

Unit -II

Advanced Data Structures*:* Red-Black trees, B – trees, Binomial Heaps, fibonacci heaps.

Unit - III

Divide and Conquer with examples such as Sorting, Matrix Multiplication, Convex hull and Searching. Greedy methods with examples such as Optimal Reliability Allocation, Knapsack, Minimum Spanning trees – Prim's and Kruskal's algorithms, Single source shortest paths - Dijkstra's and Bellman Ford algorithms.

Unit - IV

Dynamic programming with examples such as Kanpsack, All pair shortest paths – Warshal's and Floyd's algorithms, Resource allocation problem. Backtracking, Branch and Bound with examples such as Travelling Salesman Problem, Graph Coloring, n-Queen Problem, Hamiltonian Cycles and Sum of subsets.

Unit -V

Selected Topics: Algebraic Computation, Fast Fourier Transform, String Matching, Theory of NP-completeness, Approximation algorithms and Randomized algorithms.

References:

1. Thomas H. Coreman, Charles E. Leiserson and Ronald L. Rivest, "Introduction to Algorithms", Printice Hall of India.

2. RCT Lee, SS Tseng, RC Chang and YT Tsai, "Introduction to the Design and Analysis of Algorithms", Mc Graw Hill, 2005.

3. E. Horowitz & S Sahni, "Fundamentals of Computer Algorithms",

4. Berman, Paul," Algorithms", Cengage Learning.

5. Aho, Hopcraft, Ullman, "The Design and Analysis of Computer Algorithms" Pearson Education, 2008.

DESIGN AND ANALYSIS OF ALGORITHM

NOTES

The Course:

Purpose: a rigorous introduction to the design and analysis of algorithms

- Not a lab or programming course
- Not a math course, either

Textbook: *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein

- The "Big White Book"
- Second edition: now "Smaller Green Book"
- An excellent reference you should own

NOTES (UNIT-I) Introduction

1.1 Algorithm

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

For example, given the input sequence {31, 41, 59, 26, 41, 58), a sorting algorithm returns as output the sequence {26, 31, 41, 41, 58, 59}. Such an input sequence is called an instance of the sorting problem. ,

Instance : An instance of a problem consists of the input needed to compute a solution to the problem.

An algorithm is said to be correct if, for every input instance, it halts with the correct output.

There are *two aspects* of algorithmic performance:

• Time
  - Instructions take time.
  - How fast does the algorithm perform?
  - What affects its runtime?

• Space
  - Data structures take space

- What kind of data structures can be used?
- How does choice of data structure affect the runtime?

1.1.1 Analysis of Algorithms

Analysis is performed with respect to a computational model

- We will usually use a generic uniprocessor random-access machine (RAM)
  - All memory equally expensive to access
  - No concurrent operations
  - All reasonable instructions take unit time
    - Except, of course, function calls
  - Constant word size

Unless we are explicitly manipulating bits

Input Size:

- Time and space complexity
  - This is generally a function of the input size
    - E.g., sorting, multiplication
  - How we characterize input size depends:
    - Sorting: number of input items
    - Multiplication: total number of bits
    - Graph algorithms: number of nodes & edges
    - Etc

Running Time:

- Number of primitive steps that are executed
  - Except for time of executing a function call most statements roughly require the same amount of time
    - y = m * x + b
    - c = 5 / 9 * (t - 32)
    - z = f(x) + g(y)
- We can be more exact if need be

Analysis:

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee
- Average case
  - Provides the expected running time
  - Very useful, but treat with care: what is “average”?
    - Random (equally likely) inputs
    - Real-life inputs

1.1.2 Analyzing algorithm

Example: Insertion Sort:

```
InsertionSort(A, n) {
for i = 2 to n {
        key = A[i]
        j = i - 1;
        while (j > 0) and (A[j] > key) {
                A[j+1] = A[j]
                j = j - 1
                }
        A[j+1] = key
        }

}
```

Analysis

Statement	Effort
InsertionSort(A, n) {	
for i = 2 to n {	$c_1 n$
key = A[i]	$c_2(n-1)$
j = i - 1;	$c_3(n-1)$
while (j > 0) and (A[j] > key) {	$c_4 T$
A[j+1] = A[j]	$c_5(T-(n-1))$
j = j - 1	$c_6(T-(n-1))$
}	0
A[j+1] = key	$c_7(n-1)$
}	0
}	

$T = t_2 + t_3 + \ldots + t_n$ where t_i is number of while expression evaluations for the i^{th} for loop iteration

$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$
$= c_8 T + c_9 n + c_{10}$

Best case -- inner loop body never executed

$t_i = 1$ ➔

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

$= an-b = \Theta(n)$

T(n) is a linear function

Worst case -- inner loop body executed for all previous elements

$t_i = i$ ➔

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - (c_2 + c_4 + c_5 + c_8).
\end{aligned}
$$

$= an^2 + bn - c = O(n^2)$

T(n) is a quadratic function

Average case: The "average case" is often roughly as bad as the worst case.

1.1.3 Designing algorithms

The divide and conquer approach

The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

1.2 Merge sort

The ***merge sort*** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure MERGE (A, p, q, r) where A is an array and p, q, and r are indices into the array such that $p \leq q < r$. The procedure assumes that the subarrays A (p…. q) and A (q+1…… r) are in sorted order. It ***merges*** them to form a single sorted subarray that replaces the current subarray A (p …….r). Our MERGE procedure takes time $\Theta(n)$, where n = r- p+1 is the total number of elements being merged.

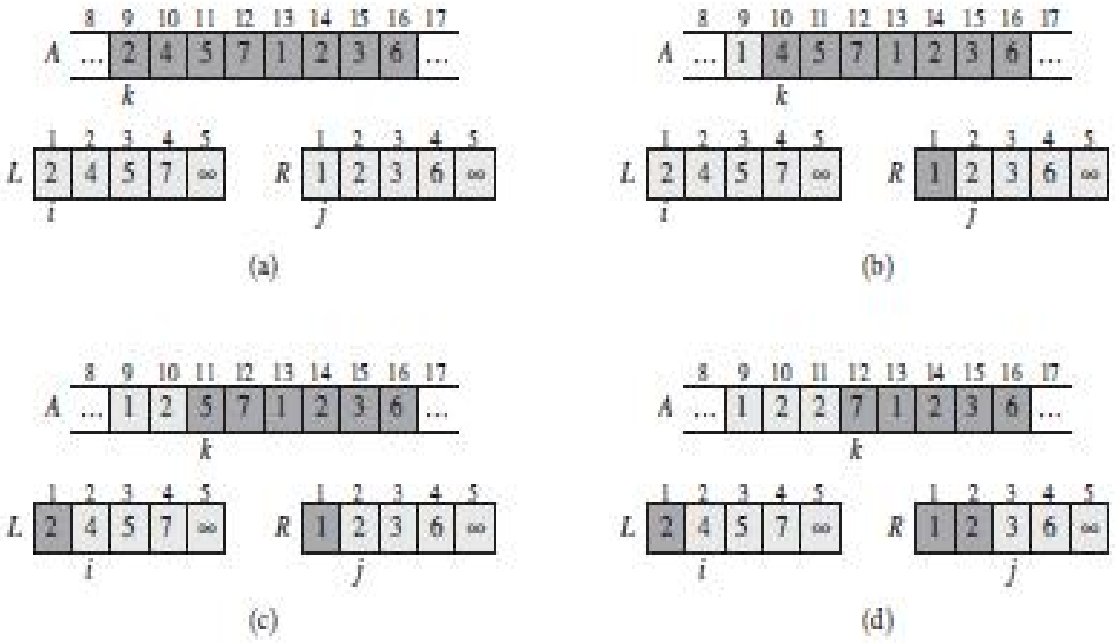
MERGE(*A*, *p*, *q*, *r*)

1 $n1 \leftarrow q - p + 1$

2 $n2 \leftarrow r - q$

3 create arrays $L[1 \ldots\ldots n1 + 1]$ and $R[1 \ldots\ldots\ldots n2 + 1]$
4 for $i \leftarrow 1$ to $n1$
5 do $L[i] \leftarrow A[p + i - 1]$
6 for $j \leftarrow 1$ to $n2$
7 do $R[j] \leftarrow A[q + j]$
8 $L[n1 + 1] \leftarrow \infty$
9 $R[n2 + 1] \leftarrow \infty$
10 $i \leftarrow 1$
11 $j \leftarrow 1$
12 for $k \leftarrow p$ to r
13 do if $L[i] \leq R[j]$
14 then $A[k] \leftarrow L[i]$
15 $i \leftarrow i + 1$
16 else $A[k] \leftarrow R[j]$
17 $j \leftarrow j + 1$

Example:



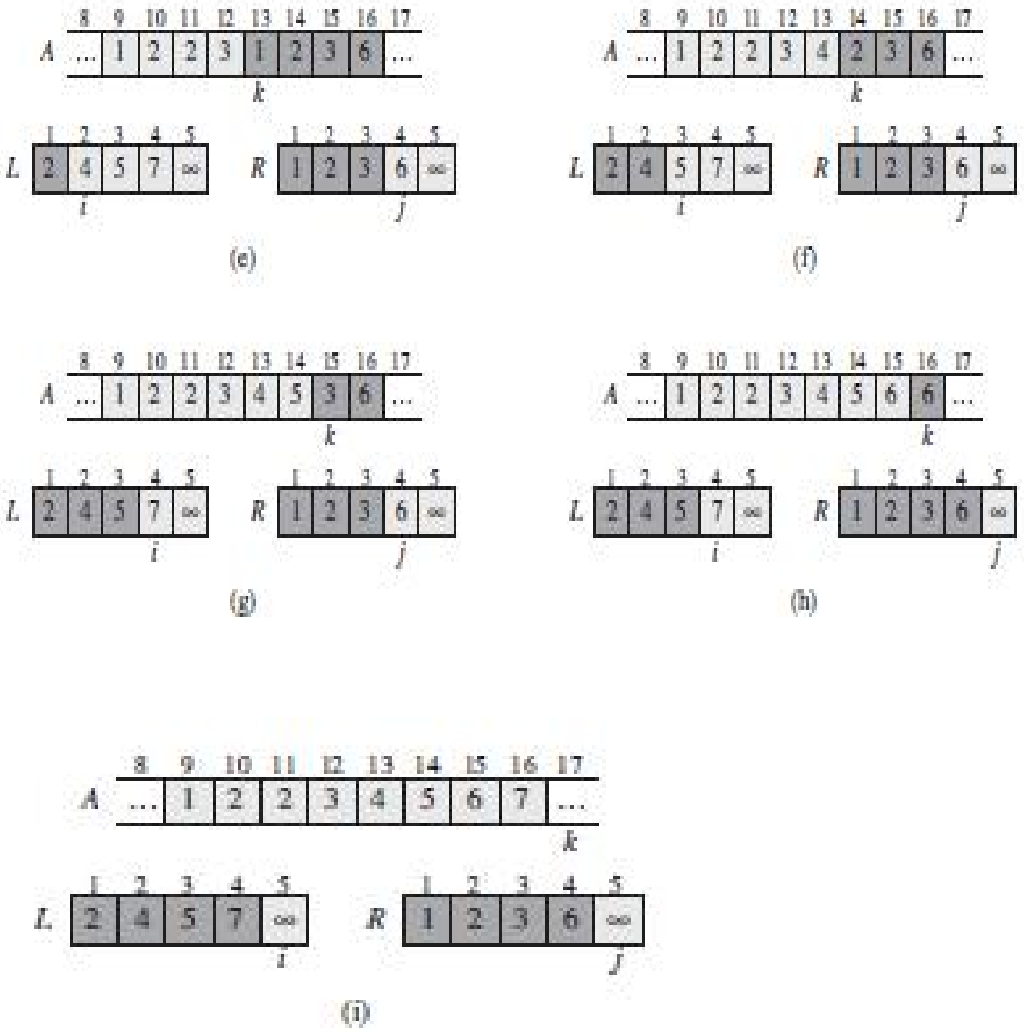


Fig: The Merge procedure applies on given array and sort and combines the solution in recursive iteration.

MERGE-SORT(*A*, *p*, *r*)

1 if $p < r$

2 then $q \leftarrow (p + r)/2$

3 MERGE-SORT(A, p, q)

4 MERGE-SORT(A, $q + 1$, r)

5 MERGE(A, p, q, r)

1.2.1 Analysis of Merge-sort

When we have n > 1 elements, we break down the running time as follows.

Divide: The divide step just computes the middle of the subarray, which takes constant time. Thus, D(n)=Θ(1).

Conquer: We recursively solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.

Combine: We have already noted that the MERGE procedure on an n-element subarray takes time Θ(n) and so C(n)=Θ(n).

The recurrence for the worst-case running time T(n) of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

The solution for above recurrence is Θ (n log n).

1.3 Growth of functions

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N=(0, 1, 2……). Such notations are convenient for describing the worst-case running-time function T(n), which usually is defined only on integer input sizes.

1.3.1 Asymptotic notations

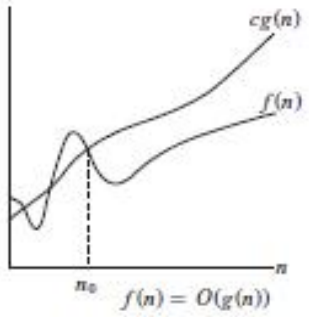
1.3.1.1 Upper Bound Notation or O-notation

We say InsertionSort's run time is $O(n^2)$. Properly we should say run time is *in* $O(n^2)$. Read O as "Big-O" (you'll also hear it as "order")

In general a function f(n) is O(g(n)) if there exist positive constants c and n_0 such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$

Formally

$O(g(n)) = \{ f(n): \exists$ positive constants c and n_0 such that $f(n) \le c \cdot g(n) \ \forall \ n \ge n_0$



Example. 1. functions in **$O(n^2)$**

$n^2/1000$, $n^{1.9}$, n^2, n^2+n, $1000\,n^2+50n$

2. Show $2n^2 = O(n^3)$

$0 \le h(n) \le cg(n)$ Definition of O(g(n))

$0 \le 2n^2 \le cn^3$ Substitute

$0/n^3 \le 2n^2/n^3 \le cn^3/n^3$ Divide by n^3

Determine c

$0 \le 2/n \le c$ $2/n = 0$

2/n maximum when n=1

$0 \le 2/1 \le c = 2$ Satisfied by c=2

Determine n_0

$0 \le 2/ n_0 \le 2$

$0 \le 2/2 \le n_0$

$0 \le 1 \le n_0 = 1$ Satisfied by n_0=1

$0 \le 2n^2 \le 2n^3$ $\forall$ $n \ge n_0$=1

If $f(n) \le cg(n)$, $c > 0$, $\forall$ $n \ge n_0$ then $f(n) \in O(g(n))$

3. 1000n2 + 50n = O(n2)

$0 \le h(n) \le cg(n)$

$0 \le 1000n2 + 50n \le cn2$ Substitute

$0/n2 \le 1000\ n2/\ n2 + 50n/\ n2 \le c\ n2/\ n2$ Divide by n2

$0 \le 1000 + 50/n \le c$ Note that $50/n \rightarrow 0$ as $n \rightarrow \infty$

Greatest when n = 1

$0 \le 1000 + 50/1 = 1050 \le c = 1050$ Satisfied by c=1050

$0 \le 1000 + 50/n0 \le 1050$ Find n0 $\forall n \ge n0$

$-1000 \le 50/\ n0 \le 50$

$-20 \le 1/\ n0 \le 1$

$-20\ n0 \le 1 \le n0 = 1$ Satisfied by n0=1

$0 \le 1000\ n2 + 50n \le 1050\ n2$ $\forall n \ge n0$=1, c=1050

If $f(n) \le cg(n)$, $c > 0$, $\forall n \ge n0$ then $f(n) \in O(g(n))$

Big O Fact

A polynomial of degree k is $O(n^k)$

Proof:

Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0$

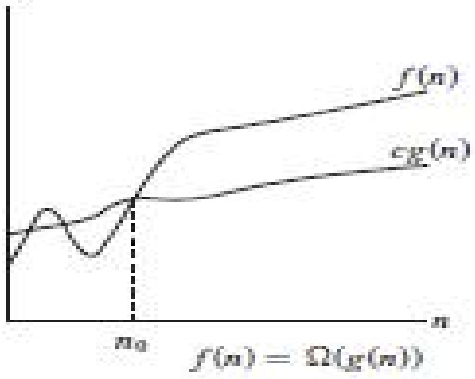
Let $a_i = | b_i |$

$f(n) \le a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$

$$\le \quad n^k \sum a_i \frac{n^i}{n^k} \quad \le \quad n^k \sum a_i \quad \le \quad cn^k$$

1.3.1.2 Lower Bound Notation or Ω-notation

We say Insertion Sort's run time is Ω(n). In general a function f(n) is Ω(g(n)) if ∃ positive constants *c* and n_0 such that $0 \le c \cdot g(n) \le f(n) \ \forall \ n \ge n_0$



Proof: Suppose run time is an + b. Assume a and b are positive (what if b is negative?).

an ≤ an + b

Example. 1. Functions in $\Omega(n^2)$

$n^2/ 1000$, $n^{1.999}$, n^2+n, $1000\ n^2+50n$

2. n3 = $\Omega(n^2)$ with c=1 and n_0=1

$0 \le cg(n) \le h(n)$

$0 \le 1*1^2 = 1 \le 1 = 13$

$0 \le cg(n) \le h(n)$

$0 \le c\ n^2 \le n^3$

$0/ n^2 \le c\ n^2/ n^2 \le n^3/ n^2$

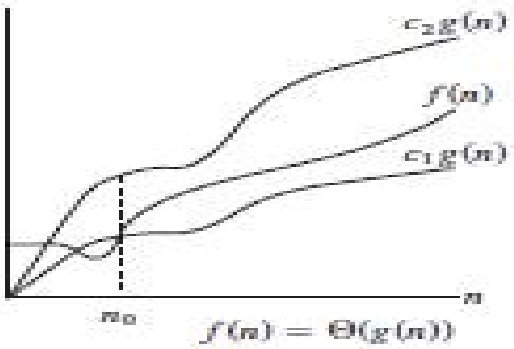
$0 \le c \le n$

$0 \le 1 \le 1$ with c=1 and n_0=1 since n increases.

1.3.1.3 Asymptotic Tight Bound or -notation

A function f(n) is Θ(g(n)) if ∃ positive constants c_1, c_2, and n_0 such that

$c_1\ g(n) \le f(n) \le c_2\ g(n) \ \forall \ n \ge n_0$



Theorem: f(n) is Θ(g(n)) iff f(n) is both O(g(n)) and Ω(g(n))

Example . 1. Show that $½n^2 - 3n \in \Theta(n^2)$

Determine ∃ c1, c2, n0 *positive constants* such that:
c1 n2 ≤ ½ n2 - 3n ≤ c2 n2
c1 ≤ ½ - 3/n ≤ c2 Divide by n2
O: Determine c2 = ½
½ - 3/n ≤ c2
as n → ∞, 3/n → 0
½ - 3/n = ½
therefore ½ ≤ c2 or c2 = ½ ½ - 3/n maximum for as n → ∞
Ω: Determine c1 = 1/14
0 < c1 ≤ ½ - 3/n ½ - 3/n > 0 minimum for n0 = 7
0 < c1 = ½ - 3/7 = 7/14 - 6/14 = 1/14
n0: Determine n0 = 7
c1 ≤ ½ - 3/ n0 ≤ c2
1/14 ≤ ½ - 3/ n0 ≤ ½
½n2-3n = Θ(n2)
c1=1/14, c2=1/2 and n
Θ: ½n2 - 3n ∈Θ(n2) when c1 = 1/14, c2 = ½ and n0 = 7

1.3.1.4 Other Asymptotic Notations

o-notation: A function f(n) is o(g(n)) if ∃ positive constants c and n_0 such that f(n) < c g(n) ∀ n ≥ n_0

ω-notation: This notation is A function f(n) is ω(g(n)) if ∃ positive constants c and n_0 such that c g(n) < f(n) ∀ n ≥ n_0

Intuitively,

- o() is like <
- ω() is like >
- Θ() is like =
- Ω() is like ≥
- O() is like ≤
-

1.4 Recurrences

Recurrences

A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, the worst-case running time T (n) of the MERGE-SORT procedure by the recurrence

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & n > 1 \end{cases}$$

Whose solution is to be T (n)=O(n lg n).

1.4.1 Solving Recurrences: following methods are used to solve recurrences and find its asymptotic performance

1. Substitution method
2. Recursion Tree
3. Master method
4. Iteration Method

1.4.1.1 The substitution method

This method is called "making a good guess method" and solving recurrences comprises two steps:

1. Guess the form of the solution.

2. Use mathematical induction to find the constants and show that the solution

We can use the substitution method to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

We guess that the solution is T(n)=O(n lg n). The substitution method requires us to prove $T(n) \leq c\, n \lg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) = c(\lfloor n/2 \rfloor) \lg (\lfloor n/2 \rfloor)$. Substituting into the recurrence yields

$T(n) \leq 2\, c(\lfloor n/2 \rfloor) \lg (\lfloor n/2 \rfloor) + n$

$\leq cn \lg (n/2) + n$

$= cn \lg n \_ cn \lg 2 + n$

$\leq cn \lg n \_ cn + n$

$\leq cn \lg n$

where the last step holds as long as $c \geq 1$.

1.4.1.2 Changing variables:

Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one you have seen before. As an example, consider the recurrence

$$T(n) = 2T\left(\lfloor \sqrt{n} \rfloor\right) + \lg n$$

which looks difficult. We can simplify this recurrence, though, with a change of variables. For convenience Renaming m= lg n yields

$$T(2^m) = T(2^{m/2}) + m$$

We can now rename $S(m) = T(2^m)$ to produce the new recurrence

$S(m) = S(m/2) + m$

The solution is to be S (m)= O (m lg m)

After replacing m= log n the solution becomes

$T(n) = O(\lg n \lg \lg n)$

1.4.1.3 The recursion-tree method: Substitution method is not coming up with good guess. Drawing out a recursion tree provides a method to devise a good guess. In a ***recursion tree***, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

For example, let us see how a recursion tree would provide a good guess for the recurrence

$$T(n)= 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$$

we create a recursion tree for the recurrence $T(n)= 3T(n/4) + cn^2$, having written out the implied constant coefficient $c > 0$.

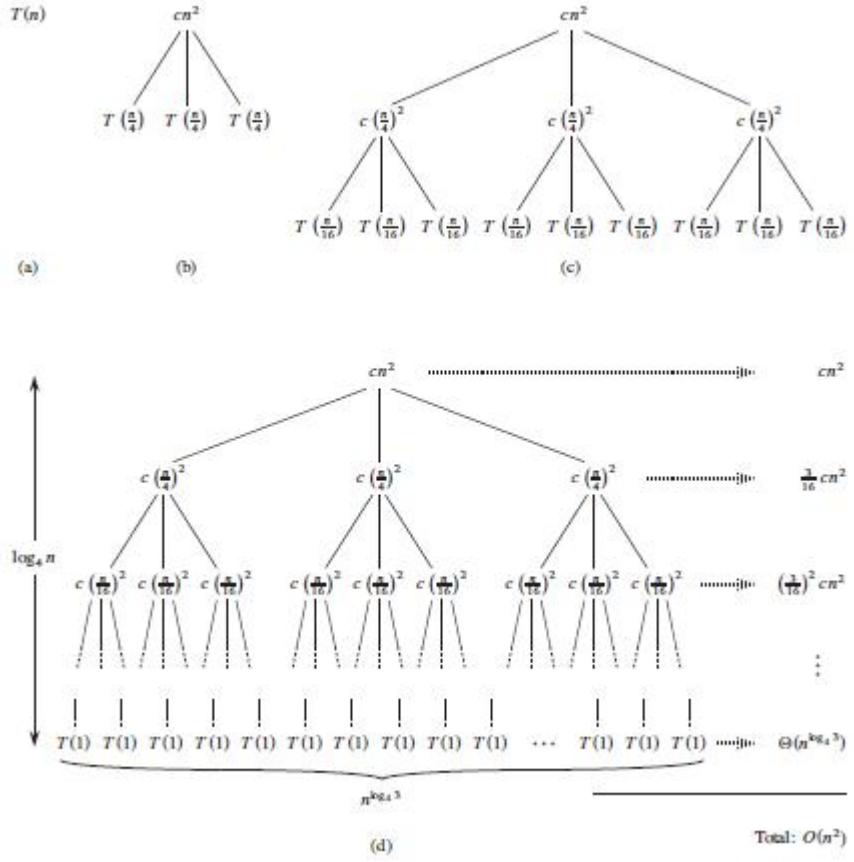


Fig: Constructing a recursion tree for the recurrence $T(n)= 3T(n/4) + cn^2$. Part **(a)** shows T (n), which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n$ +1 levels).

The above fig shows how we derive the recursion tree for $T(n)= 3T(n/4) + cn^2$ .For convenience, we assume that n is an exact power of 4 so that all subproblem sizes are integers. Part (a) of the figure shows T(n), which we expand in part (b) into an equivalent tree representing the recurrence. The cn^2 term at the root represents the cost at the top level of recursion, and the three subtrees of the root represent the costs incurred by the subproblems of size n=4. Part (c) shows

this process carried one step further by expanding each node with cost T(n/4) from part (b). The cost for each of the three children of the root is $c(n/4^2)$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence.

Because subproblem sizes decrease by a factor of 4 each time the subproblem size for a node at depth i is $(n/4^i)$. Thus for subproblem size at last level $n=1$ $n/4^i=1$ then $i = \log_4 n$ and the tree has $\log_4 n+1$ levels. The cost of each node is derived by generalized term at depth i where i= 0, 1, 2…….$\log_4 n-1$ by $c\,(n/4^i)^2$. the total cost over all nodes at depth i, for i= 0, 1, 2…….$\log_4 n-1$ is $3^i\,(n/4^i)^2 = (3/16)^i\ cn^2$. The last level i.e. at depth $\log_4 n$ cost is $3^{\log 4n} = n^{\log_4 3}$ nodes each contributing cost T(1), for a total cost $n^{\log_4 3}\,T(1)$ which is $\Theta(n^{\log_4 3})$ since we assume T(1) is constant.

Now we add up the costs over all levels to determine the cost for the entire tree:

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)}\,cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13}\,cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)\,.
\end{aligned}
$$

Thus, we have derived a guess of $T(n)= O(n^2)$. Now we can use the substitution method to verify that our guess was correct, that is, $T(n)= O(n^2)$ is an upper bound for the recurrence $T(n)= 3T(n/4) + \Theta(n^2)$. We want to show that $T(n)\leq dn^2$ for some constant $d > 0$. Using the same constant $c > 0$ as before, we have

$$
\begin{aligned}
T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\
&\leq 3d\,\lfloor n/4 \rfloor^2 + cn^2 \\
&\leq 3d(n/4)^2 + cn^2 \\
&= \frac{3}{16}\,dn^2 + cn^2 \\
&\leq dn^2\,,
\end{aligned}
$$

Where the last step holds as long as $d = (16/13)c$.

1.4.1.4 The master method

For divide and conquer algorithm**,** An algorithm that divides the problem of size n into a subproblems, each of size n/b Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function f(n), then, the Master Theorem gives us a cookbook for the algorithm's running time in the form of recurrence given below

$$T(n) = a\,T(n/b) + f(n)$$

Where $a\geq 1$and $b>1$ are constants and f (n) is an asymptotically positive function.

1.4.1.4.1 The master theorem:

The master method depends on the following theorem.

Let a≥1and b>1 be constants, let f (n) be a function, and let T(n) be defined on the nonnegative integers by the recurrence

$$T(n) = a\,T(n/b) + f(n)$$

where we interpret n/b to mean either floor (n/b) or ceil (n/b). Then T(n) has the following asymptotic bounds:

1. If $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$ for some constant ϵ>0 then T(n)= $\Theta\left(n^{\log_b a}\right)$.
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$ then T(n)= $\Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ for some constant ϵ>0 and if a T (n/b) ≤c f(n)(regularity function) for some constant c<1 and all sufficiently large n then T(n)= Θ(f (n)).

In each of the three cases, we compare the function f (n) with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determines the solution to the recurrence. In case 1the function $n^{\log_b a}$ is larger, solution is T(n)= $\Theta\left(n^{\log_b a}\right)$. In case 3 function f(n) is larger, solution is T(n)= Θ(f (n)). In case 2 both functions has same value, solution is T(n)= $\Theta(n^{\log_b a} \log n)$.

In the first case, not only must f(n) be smaller than $n^{\log_b a}$, it must be *polynomially* smaller. In the third case, not only must f (n) be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the "regularity" condition that a T (n/b) ≤c f(n). An addition to that all three cases do not cover all possibilities. Some function might be lies in between case 1 and 2 and some other lies in between case 2 and 3 because the comparison is not polynomial larger or smaller and in case 3 the regularity condition fails.

Example. 1. . The given recurrence is

$$T(n) = 9T(n/3) + n$$

Sol: a=9, b=3, f(n) = n

$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$

Since $f(n) = O(n^{\log_3 9 - \varepsilon})$, where ε=1, case 1 applies:

$T(n) = \Theta\left(n^{\log_b a}\right)$ when $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$

Thus the solution is $T(n) = \Theta(n^2)$

2. T(n)= T(2n/3)+1

in which a = 1, b= 3/2, f(n)= 1, and $n^{\log_b a} = n^{\log 3/2} = n^0 = 1$. Case 2

applies, since f (n)= Θ ($n^{\log_b a}$)= Θ(1) and thus the solution to the recurrence is T(n) =Θ(lg n)

3. The master method does not apply to the recurrence:

T(n)= 2T(n/2) +n log n

Sol: a=2, b=2, f(n)= n log n , $n^{\log_b a}=n^{\log 2}=n$

Since f(n) is larger than $n^{\log_b a}$ you mistakenly apply case 3 but the f(n) is larger but not polynomialy larger. The ratio f(n)/ $n^{\log_b a}$= log n is asymptotically less than n^{ϵ} for any positive constant ϵ. Consequently, the recurrence falls into the gap between case 2 and case 3.

1.4.1.5 Iteration method

Another option is the "iteration method" which expand the recurrence by using iterative equations, work some algebra to express as a summation and finally evaluate the summation. It's a iterative procedure which recursively used and prorogate to a final single value.

For example

Floor and ceilings are a pain to deal with. If n is assumed to be a power of 2 ($2^k = n$), this will simplify the recurrence to

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

The iteration method turns the recurrence into a summation. Let's see how it works. Let's expand the recurrence:

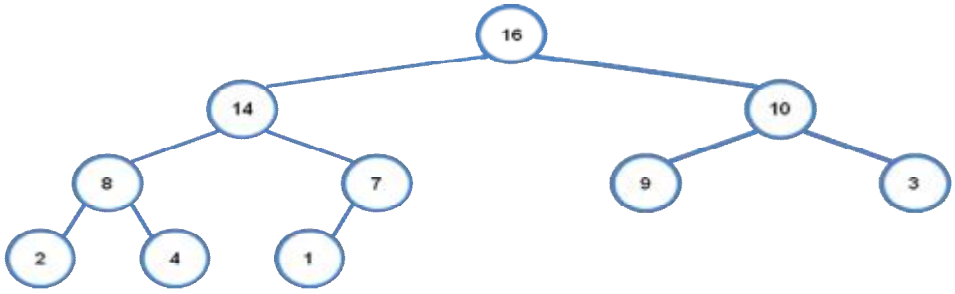
$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + n + n \\ &= 4(2T(n/8) + n/4) + n + n \\ &= 8T(n/8) + n + n + n \\ &= 8(2T(n/16) + n/8) + n + n + n \\ &= 16T(n/16) + n + n + n + n \end{aligned}$$

If n is a power of 2 then let $n = 2^k$ or $k = \log n$.

$$\begin{aligned} T(n) &= 2^k T(n/(2^k)) + \underbrace{(n + n + n + \cdots + n)}_{k \text{ times}} \\ &= 2^k T(n/(2^k)) + kn \\ &= 2^{(\log n)} T(n/(2^{(\log n)})) + (\log n)n \\ &= 2^{(\log n)} T(n/n) + (\log n)n \\ &= nT(1) + n \log n = n + n \log n \end{aligned}$$

1.5 Heap sort

A heap can be seen as a nearly binary tree:



An array A that represents a heap is an object with two attributes: A.length, which (as usual) gives the number of elements in the array, and A.heap-size, which represents how many elements in the heap are stored within array A. That is, although A (1……..A.length) may

contain numbers, only the elements in A(1…….A.heap-size), where 0≤ A.heap-size≤ A.length, are valid elements of the heap. The root of the tree is A(1), and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

```
Parent(i)
return floor (i/2)

Left(i)
return 2i

Right(i)
return 2i+1
```

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap. In a max-heap, the max-heap property is that for every node i other than the root,

$$A[parent(i)] \geq A[i]$$

That is the value of a node is lesser or equal to parent.

A min-heap is organized in the opposite way; the min-heap property is that for every node i other than the root,

$$A[parent(i)] \leq A[i]$$

For the heapsort algorithm, we use max-heaps.

1.5.1 Maintaining the heap property:

1.5.1.1 MAX-HEAPIFY(A, i)

```
{
        l = Left(i); r = Right(i);
        if (l <= A.heap-size && A[l] > A[i])
                largest = l;
        else
                largest = i;
        if (r <= A.heap-size && A[r] > A[largest])
                largest = r;
        if (largest != i)
                exchange A[i] with A[largest];
                MAX-HEAPIFY(A, largest);
}
```

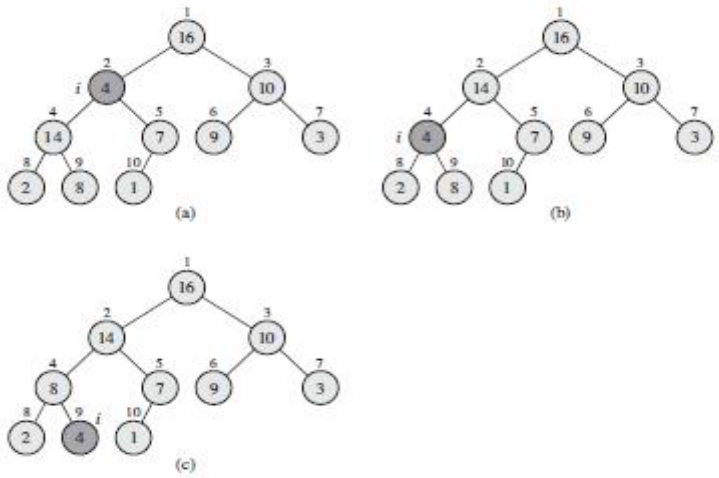


Figure: The action of MAX-HEAPIFY(*A*, 2), where *heap-size*[*A*] = 10. *(a)* The initial configuration, with *A*[2] at node *i* = 2 violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in *(b)* by exchanging *A*[2] with *A*[4], which destroys the max-heap property for node 4. The recursive call MAXHEAPIFY(*A*, 4) now has *i* = 4. After swapping *A*[4] with *A*[9], as shown in *(c)*, node 4 is fixed up, and the recursive call MAX-HEAPIFY(*A*, 9) yields no further change to the data structure.

1.5.1.2 Complexity

We can describe the running time of MAX-HEAPIFY by the recurrence

$$T(n) \leq 2T(n/3) + \Theta(1)$$

The solution to this recurrence, by case 2 of the master theorem is T(n)= O(lg n)

1.5.2 Building a heap:

We can build a heap in a bottom-up manner by running **Heapify()** on successive subarrays. For array of length *n*, all elements in range A[$\lfloor n/2 \rfloor + 1$ .. n] are heaps.

1.5.2.1 BUILD-MAX-HEAP(A)

1. A.heap-size = A.length
2. for i = $\lfloor$length[A]/2$\rfloor$ downto 1
3. MAX-HEAPIFY(A, i)

The time required by MAX-HEAPIFY when called on a node of height h is O(h), the running time of above algorithm is O(n).

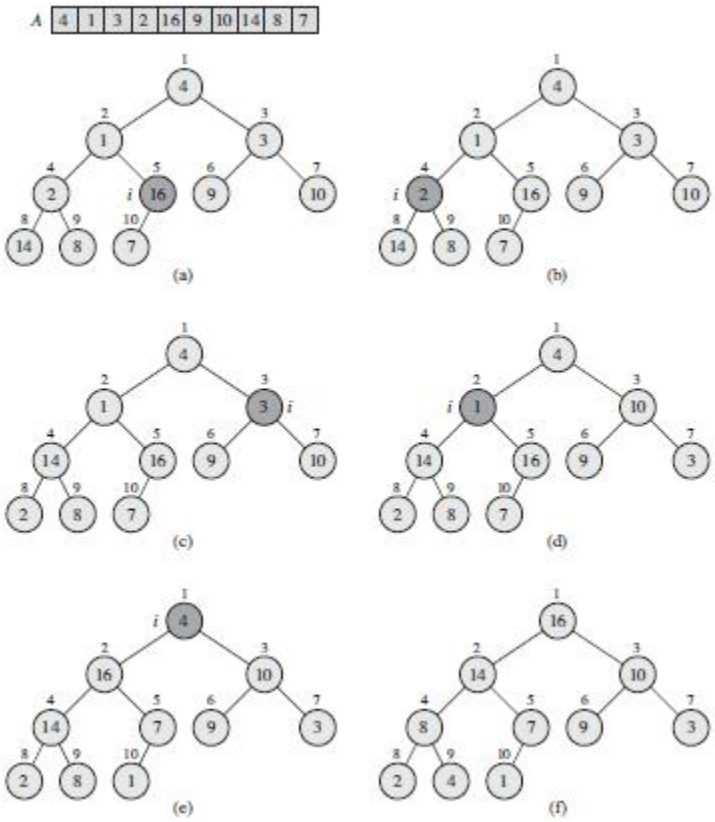


Figure: The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. *(a)* A 10-element input array *A* and the binary tree it represents. The figure shows that the loop index *i* refers to node 5 before the call MAX-HEAPIFY(*A, i*). *(b)* The data structure that results. The loop index *i* for the next iteration refers to node 4. *(c)-(e)* Subsequent iterations of the *for* loop in BUILD-MAXHEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. *(f)* The max-heap after BUILD-MAX-HEAP finishes.

1.5.3 The heapsort Algorithm

The heapsort algorithm starts to build max heap by using procedure BUILD-MAX-HEAP and then picks the root element which has the higher value. Then remove root value from the tree and built again it max heap. This process performs up to last value and sorted array is formed.

1.5.3.1 HEAPSORT(A)

1. BUILD-MAX-HEAP(A)
2. for i = length(A) downto 2
3. Exchange (A[1] with A[i]);
4. A.heap-size= A.heap-size -1;
5. MAX-HEAPIFY (A, 1);

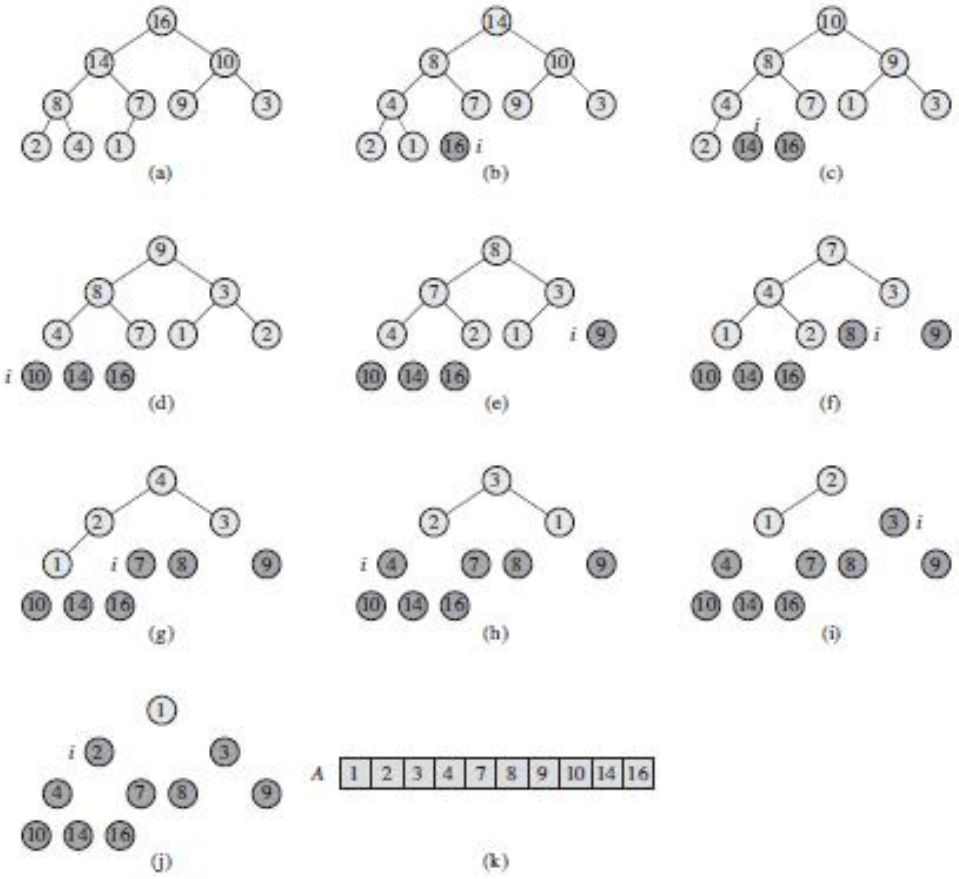


Figure: The operation of HEAPSORT. *(a)* The max-heap data structure just after it has been built by BUILD-MAX-HEAP. *(b)-(j)* The max-heap just after each call of MAXHEAPIFY in line 5. The value of *i* at that time is shown. Only lightly shaded nodes remain in the heap. *(k)* The resulting sorted array *A*.

1.6 Quick sort

Quicksort is also a divide-and-conquer algorithm. An unsorted array A taken in which p and r is the lower bound and upper bound of the elements respectivally.

Divide: The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]. Invariant: All elements in A[p..q] are less than all elements in A[q+1..r].

Conquer: The subarrays are recursively sorted by calls to quicksort.

Combine: Unlike merge sort, no combining step: two subarrays form an already-sorted array.

The following procedure implements quicksort:

QUICKSORT(A, p, r)

1. if (p < r)
2. q = PARTITION(A, p, r)

3. QUICKSORT(A, p, q)
4. QUICKSORT(A, q+1, r)

To sort an entire array A, the initial call is QUICKSORT (A, 1, A.length).

Partitioning the array:

The key to the algorithm is the PARTITION procedure, which rearranges the subarray A(p….r) in place.

PARTITION(*A*, *p*, *r*)

1 $x \leftarrow A[r]$

2 $i \leftarrow p - 1$

3 for $j \leftarrow p$ to $r - 1$

4 do if $A[j] \leq x$

5 then $i \leftarrow i + 1$

6 exchange $A[i] \leftrightarrow A[j]$

7 exchange $A[i + 1] \leftrightarrow A[r]$

8 return $i + 1$

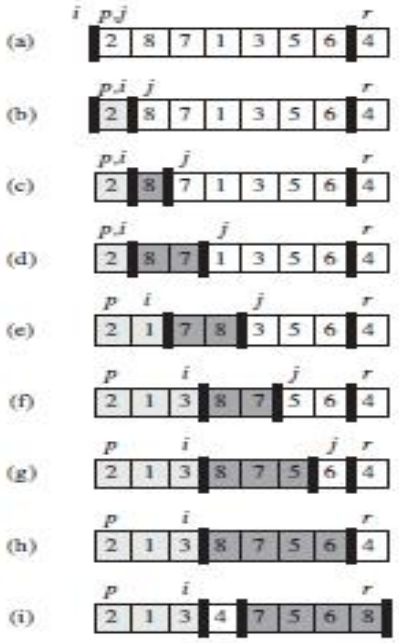


Figure: The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than *x*. Heavily shaded elements are in the second partition with values greater than *x*. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. *(a)* The initial array and variable settings. None of the elements have been placed in either of the first two partitions. *(b)* The value 2 is "swapped with itself" and put in the partition of smaller values. *(c)-(d)* The values 8 and 7 are added to the partition of larger values. *(e)* The values 1 and 8 are swapped, and the smaller partition Grows. *(f)* The values 3 and 8 are swapped, and the smaller partition grows. *(g)-(h)* The

larger partition grows to include 5 and 6 and the loop terminates. *(i)* In lines 7-8, the pivot element is swapped so that it lies between the two par titions.

1.6.1 Performance of Quicksort:

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning.

1.6.1.1 Worst Case Partitioning:

The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n - 1 elements and one with 0 elements. The recurrence for the running time is

$T(n)=T(n-1)+T(0) + \Theta(1)= \Theta(n^2)$ i.e $T(n) = \Theta(n^2)$

1.6.1.2 Best case partitioning:

in the best case it partition the array into equal sub arrays. The recurrence for balanced portioning is

$T(n) = 2T(n/2) + \Theta(n)= \Theta(n \lg n)$ i.e. $T(n) = \Theta(n \lg n)$

1.7 Shell sort

Shellsort, also known as Shell sort or Shell's method, is an in-place comparison sort. It can be seen as either a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort).The method starts by sorting elements far apart from each other and progressively reducing the gap between them.

```
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]

# Start with the largest gap and work down to a gap of 1
foreach (gap in gaps)
{
   # Do a gapped insertion sort for this gap size.
   # The first gap elements a[0..gap-1] are already in gapped order
   # keep adding one more element until the entire array is gap sorted
   for (i = gap; i < n; i += 1)
   {
      # add a[i] to the elements that have been gap sorted
      # save a[i] in temp and make a hole at position i
      temp = a[i]
      # shift earlier gap-sorted elements up until the correct location for a[i] is found
      for (j = i; j >= gap and a[j - gap] > temp; j -= gap)
```

```
        {
            a[j] = a[j - gap]
        }
        # put temp (the original a[i]) in its correct location
        a[j] = temp
    }

}
```

An example run of Shellsort with gaps 5, 3 and 1 is shown below

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
input data:	62	83	18	53	07	17	95	86	47	69	25	28
after 5-sorting:	17	28	18	47	07	25	83	86	53	69	62	95
after 3-sorting:	17	07	18	47	28	25	69	62	53	83	86	95
after 1-sorting:	07	17	18	25	28	47	53	62	69	83	86	95

The first pass, 5-sorting, performs insertion sort on separate subarrays (a_1, a_6, a_{11}), (a_2, a_7, a_{12}), (a_3, a_8), (a_4, a_9), (a_5, a_{10}). For instance, it changes the subarray (a_1, a_6, a_{11}) from (62, 17, 25) to (17, 25, 62). The next pass, 3-sorting, performs insertion sort on the subarrays (a_1, a_4, a_7, a_{10}), (a_2, a_5, a_8, a_{11}), (a_3, a_6, a_9, a_{12}). The last pass, 1-sorting, is an ordinary insertion sort of the entire array (a_1,..., a_{12}).

The complexity of this algorithm is O ($n^{1.25}$).

1.8 Sorting in linear time

The algorithms that can sort n numbers in O(n lg n) time is Merge sort and heap sort and achieve this upper bound in the worst case; Quick sort achieves it on average. These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms comparison sorts. All the sorting algorithms introduced thus far are comparison sorts.

We shall prove that any comparison sort must make $\Theta(n \lg n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heap sort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

Here three sorting algorithms-counting sort, radix sort, and bucket sort-that run in linear time. These algorithms use operations other than comparisons to determine the sorted order. Consequently, the $\Theta(n \lg n)$ lower bound does not apply to them

1.8.1 Counting sort

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When k = O(n), the sort runs in ,Θ (n) time.

1.8.1.1 The algorithm:

1. Input: A[1..n], where A[j] ∈ {1, 2, 3, …, k}
2. Output: B[1..n], sorted (notice: not sorting in place)
3. Also: Array C[1..k] for auxiliary storage

COUNTING-SORT(A, B, k)

1 for $i \leftarrow 0$ to k
2 do $C[i] \leftarrow 0$
3 for $j \leftarrow 1$ to $length[A]$
4 do $C[A[j]] \leftarrow C[A[j]] + 1$
5 //$C[i]$ now contains the number of elements equal to i.
6 for $i \leftarrow 1$ to k
7 do $C[i] \leftarrow C[i] + C[i - 1]$
8 //$C[i]$ now contains the number of elements less than or equal to i.
9 for $j \leftarrow length[A]$ downto 1
10 do $B[C[A[j]]] \leftarrow A[j]$
11 $C[A[j]] \leftarrow C[A[j]] - 1$

1.8.1.2 Running time of Counting Sort

The for loop of lines 1-2 takes time $\Theta(k)$, the for loop of lines 3-4 takes time $\Theta(n)$, the for loop of lines 6-7 takes time $\Theta(k)$, and the for loop of lines 9-11 takes time $\Theta(n)$. Thus, the overall time is $\Theta(k+n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

Example:

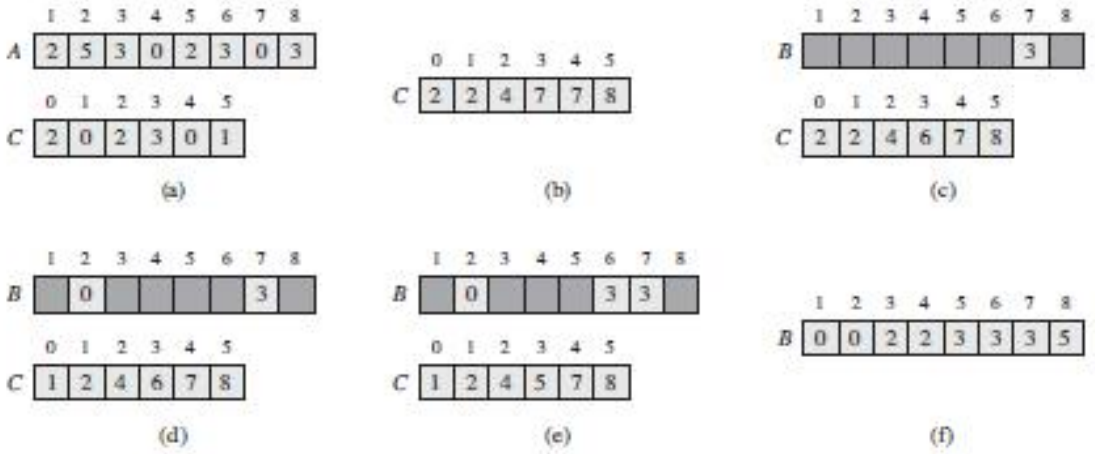


Figure: The operation of COUNTING-SORT on an input array A[1,….., 8], where each element of A is a nonnegative integer no larger than k = 5. *(a)* The array A and the auxiliary array C after line 4. *(b)* The array C after line 7. *(c)-(e)* The output array B and the auxiliary array C after one,

two, and three iterations of the loop in lines 9-11, respectively. Only the lightly shaded elements of array *B* have been filled in. *(f)* The final sorted output array *B*.

1.8.2 Radix Sort

Radix sort solves the problem of card sorting—by sorting on the *least significant* digit first. The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. The process continues until the cards have been sorted on all d digits. Only d passes through the deck are required to sort. The algorithm is

RadixSort(A, d)

for i=1 to d

StableSort(A) on digit i

Given *n* *d*-digit numbers in which each digit can take on up to *k* possible values, RADIXSORT correctly sorts these numbers in $\Theta(d(n + k))$ time.

Example:

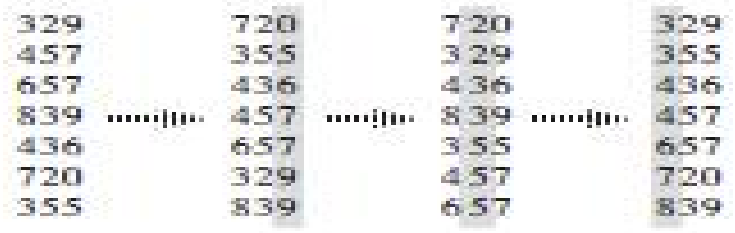


Figure: The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

1.8.3 Bucket sort

Assumption: input is *n* reals from [0, 1)

Basic idea: Create *n* linked lists (*buckets*) to divide interval [0,1) into subintervals of size 1/*n*. Add each input element to appropriate bucket and sort buckets with insertion sort. Uniform input distribution → O(1) bucket size. Therefore the expected total time is O(n). These ideas will return when we study *hash tables.*

BUCKET-SORT(*A*)

1 *n* ← *length*[*A*]

2 for *i* ← 1 to *n*

3 do insert *A*[*i*] into list B[⌊n A[i]⌋]

4 for *i* ← 0 to *n* - 1

5 do sort list *B*[*i*] with insertion sort

6 concatenate the lists *B*[0], *B*[1], . . ., *B*[*n* - 1] together in order

Example:

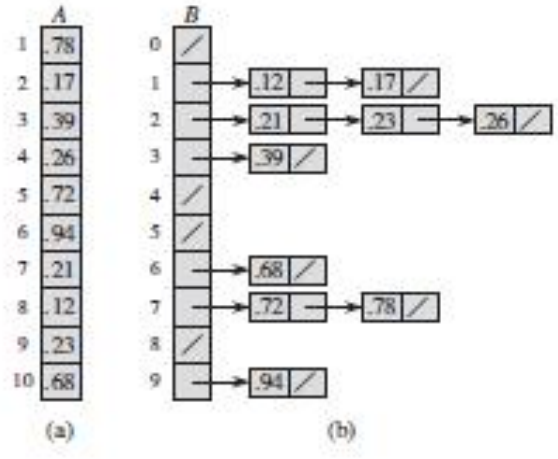


Figure: The operation of BUCKET-SORT for n= 10. **(a)** The input array A(1……..10). **(b)** The array B(0…… 9) of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval [i/10,.i + 1/10). The sorted output consists of a concatenation in order of the lists B[0], B[1]…….B[9]

To analyze the running time, observe that all lines except line 5 take O (n) time in the worst case.

NOTES (UNIT-II) Advance Data Structure

2.1 Red Black Tree

2.1.1Properties of red-black trees

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

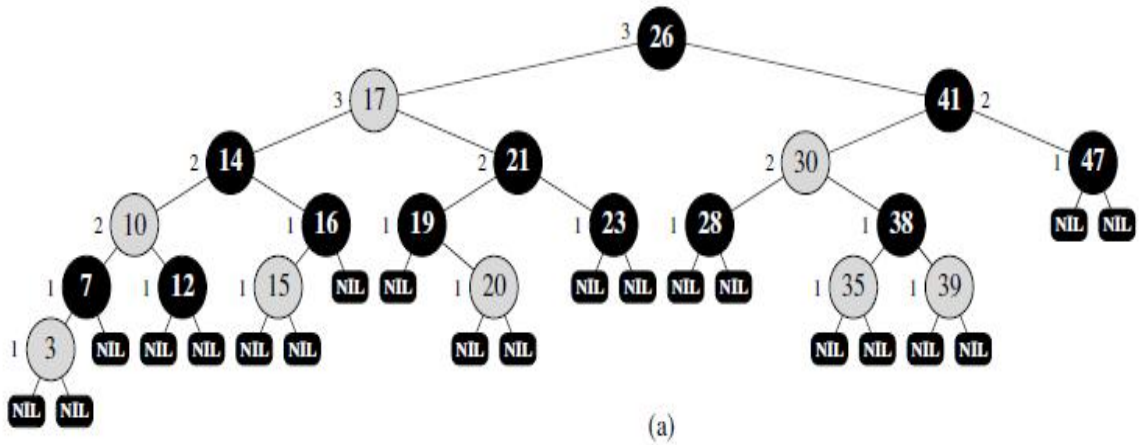
Each node of the tree now contains the attributes color, key, left, right, and p. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following **red-black properties**:

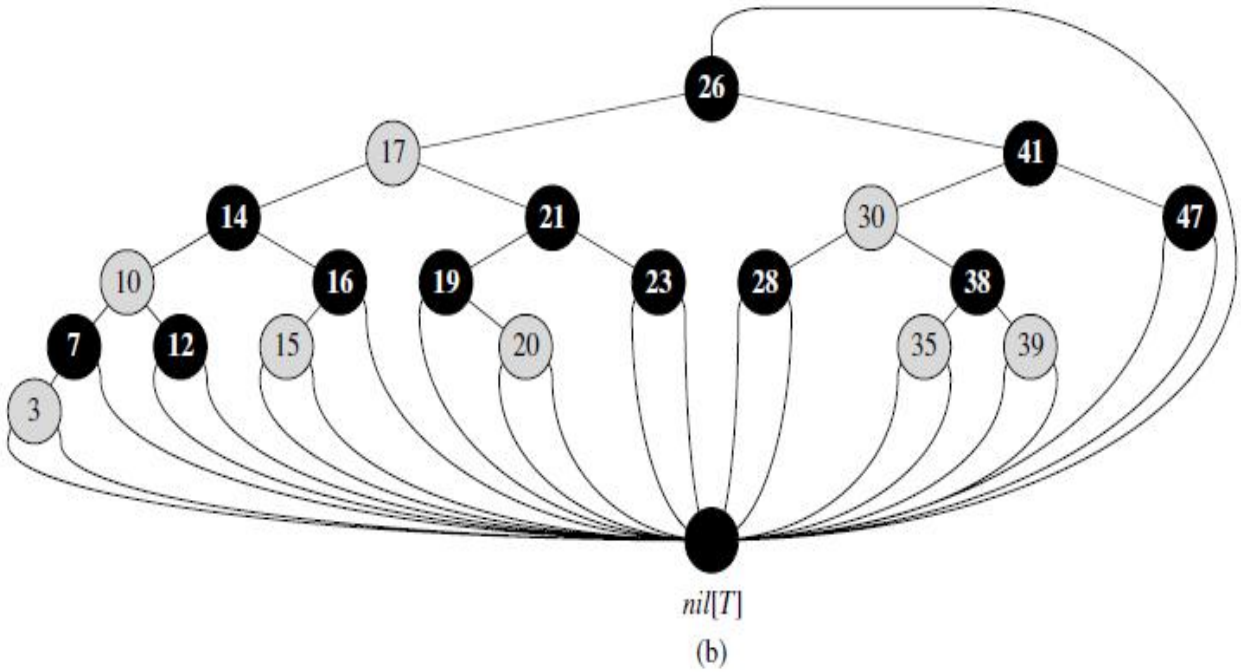
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the **black-height** of the node, denoted bh(x). By property 5, the notion of black-height is well defined, since all descending simple paths from the node have the same number of black nodes. We define the black-height of a red-black tree to be the black-height of its root.

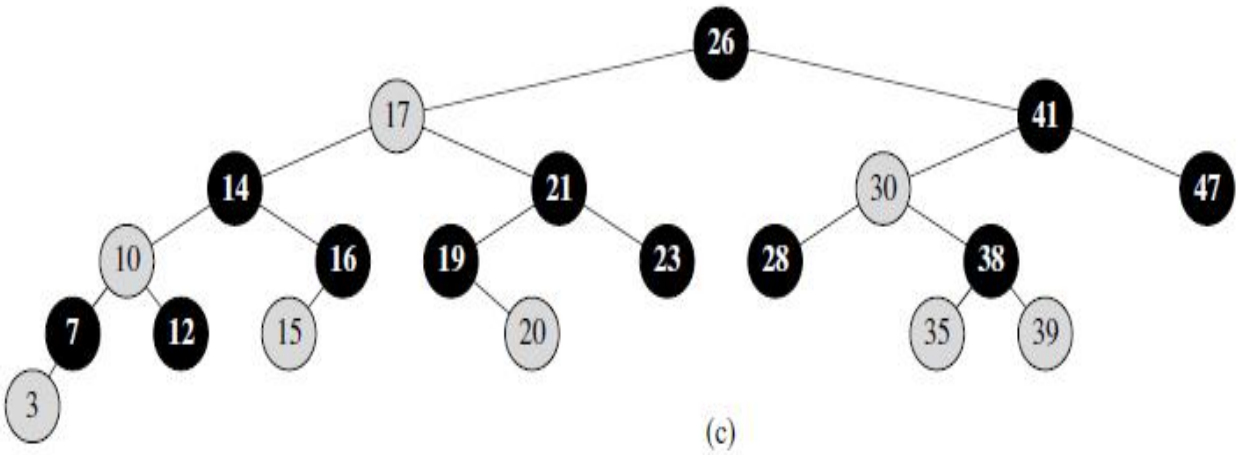
2.1.2 Representation of a Red-Black Tree



(a)



(b)



(c)

Figure: A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is both red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. (a) Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL's have black-height 0. (b) The same red-black tree but with each NIL replaced by the single sentinel nil[T], which is always black, and with black-heights omitted. The root's parent is also the sentinel. (c) The same red-black tree but with leaves and the root's parent omitted entirely.

The following lemma shows why red-black trees make good search trees.

2.1.3 Lemma 1

A red-black tree with *n* internal nodes has height at most 2 lg(*n* + 1).

Proof We start by showing that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes. We prove this claim by induction on the height of x. If the height of x is 0, then x must be a leaf ($nil[T]$), and the subtree rooted at x indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes, which proves the claim.

To complete the proof of the lemma, let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least $h/2$; thus,

$n \geq 2^{h/2} - 1$.

Moving the 1 to the left-hand side and taking logarithms on both sides yields $\lg(n + 1) \geq h/2$,

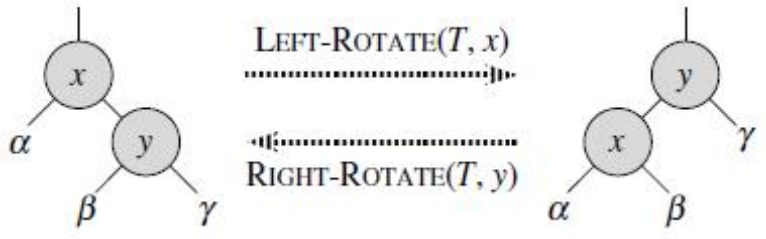
or $h \leq 2 \lg(n + 1)$.

2.2 Rotations in Red-Black Tree

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red black tree with n keys, take O(lg n) time. Because they modify the tree, the result may violate the red-black properties. To restore red black tree properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

We change the pointer structure through rotation, which is a local operation in a search tree that preserves the binary-search-tree property. There are two kinds of rotations: left rotations and right rotations. When we do a left rotation on a node x, we assume that its right child y is not nil[T]; x may be any node in the tree whose right child is not nil[T]. The left rotation “pivots” around the link from x to y. It makes y the new root of the sub tree, with x as y’s left child and y’s left child as x’s right child.

The pseudocode for LEFT-ROTATE assumes that right[x] /= nil[T] and that the root’s parent is nil[T].

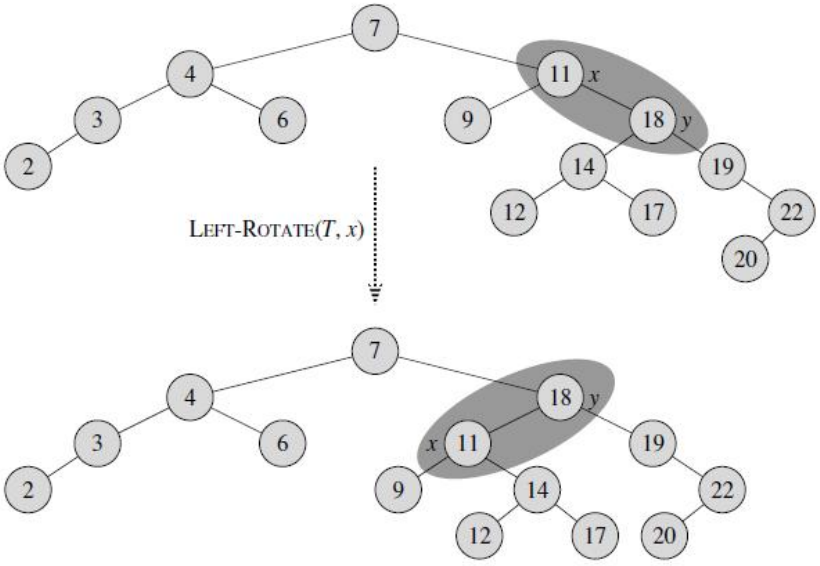


The rotation operations on a binary search tree. The operation LEFT-ROTATE (T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a

constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation RIGHT-ROTATE(T, y). The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede key[x], which precedes the keys in β, which precede key[y], which precedes the keys in γ .

```
LEFT-ROTATE(T, x)
 1  y ← right[x]            ▷ Set y.
 2  right[x] ← left[y]      ▷ Turn y's left subtree into x's right subtree.
 3  if left[y] ≠ nil[T]
 4     then p[left[y]] ← x
 5  p[y] ← p[x]             ▷ Link x's parent to y.
 6  if p[x] = nil[T]
 7     then root[T] ← y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10             else right[p[x]] ← y
11  left[y] ← x             ▷ Put x on y's left.
12  p[x] ← y
```

The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in O(1) time. Only pointers are changed by a rotation; all other fields in a node remain the same.



An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

2.3 Insertion in Red Black tree

```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6             then x ← left[x]
 7             else  x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z)
```

```
RB-INSERT-FIXUP(T, z)
 1  while color[p[z]] = RED
 2      do if p[z] = left[p[p[z]]]
 3            then y ← right[p[p[z]]]
 4                 if color[y] = RED
 5                   then color[p[z]] ← BLACK                    ▷ Case 1
 6                        color[y] ← BLACK                       ▷ Case 1
 7                        color[p[p[z]]] ← RED                   ▷ Case 1
 8                        z ← p[p[z]]                            ▷ Case 1
 9                   else if z = right[p[z]]
10                          then z ← p[z]                        ▷ Case 2
11                               LEFT-ROTATE(T, z)               ▷ Case 2
12                        color[p[z]] ← BLACK                    ▷ Case 3
13                        color[p[p[z]]] ← RED                   ▷ Case 3
14                        RIGHT-ROTATE(T, p[p[z]])               ▷ Case 3
15            else (same as then clause
                        with "right" and "left" exchanged)
16  color[root[T]] ← BLACK
```

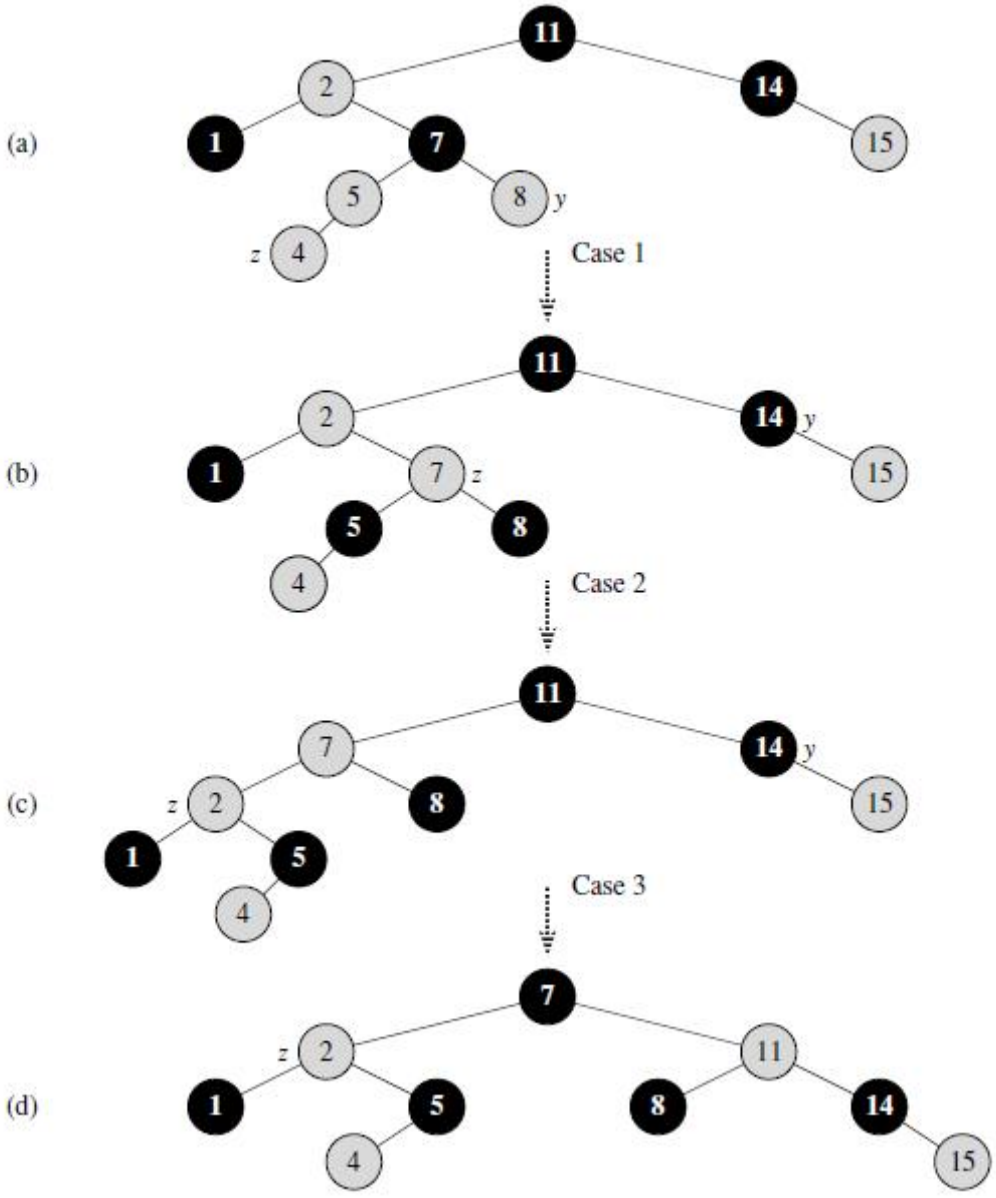
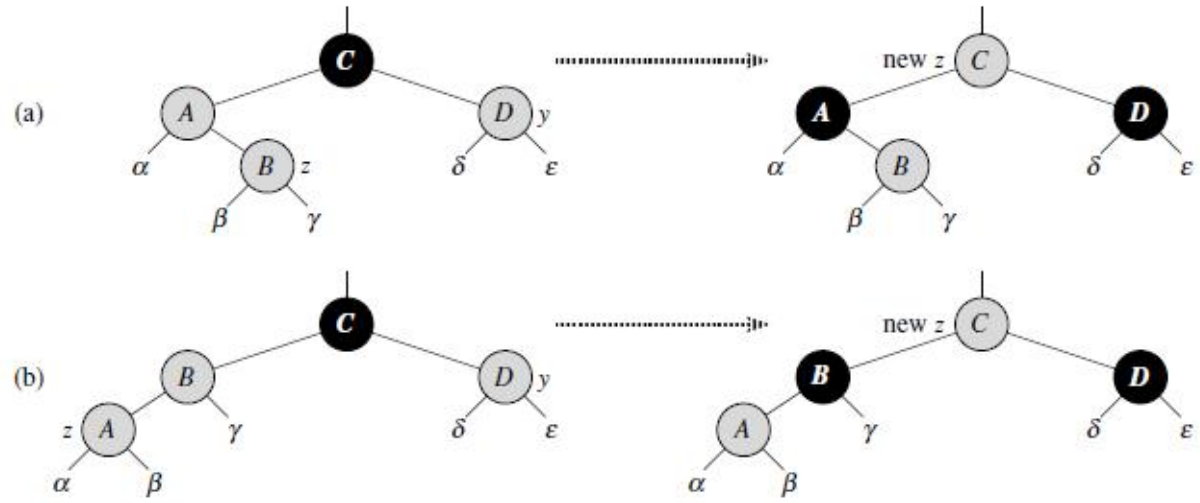


Figure: The operation of RB-INSERT-FIXUP. (a) A node z after insertion. Since z and its parent p[z] are both red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code can be applied. Nodes are recolored and the pointer z is moved up the tree, resulting in the tree shown in (b). Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of p[z], case 2 can be applied. A left rotation is performed, and the tree that results is shown in (c). Now z is the left child of its parent, and case 3 can be applied. A right rotation yields the tree in (d), which is a legal red-black tree.

Case 1: z's uncle y is red

we can color both p[z] and y black, thereby fixing the problem of z and p[z] both being red, and color p[p[z]] red,

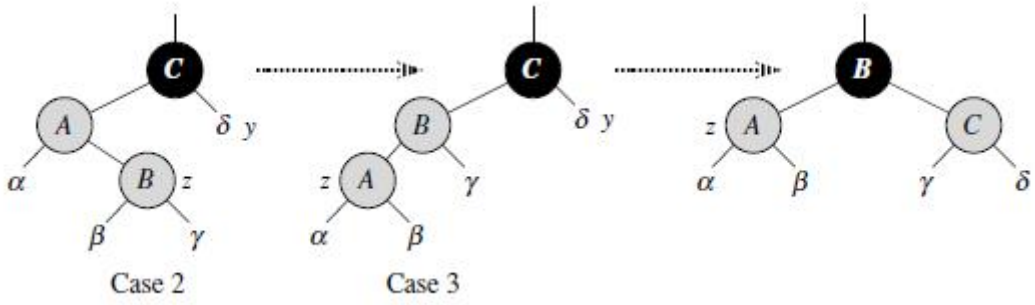


Case 2: z's uncle y is black and z is a right child

In cases 2 and 3, the color of z's uncle y is black. The two cases are distinguished by whether z is a right or left child of p[z]. In case 2, node z is a right child of its parent. We immediately use a left rotation to transform the situation into case 3 , in which node z is a left child. Because both z and p[z] are red, the rotation affects neither the black-height of nodes nor property 5.

Case 3: z's uncle y is black and z is a left child

In case 3, we execute some color changes and a right rotation, which preserve property 5, and then, since we no longer have two red nodes in a row, we are done.



2.3.1 Analysis

What is the running time of RB-INSERT? Since the height of a red-black tree on n nodes is O(lg n, RB-INSERT take O(lg n) time. In RB-INSERTFIXUP, the while loop repeats only if case 1 is executed, and then the pointer z moves two levels up the tree. The total number of times the

while loop can be executed is therefore O(lg n). Thus, RB-INSERT takes a total of O(lg n) time. Interestingly, it never performs more than two rotations, since the while loop terminatesif case 2 or case 3 is executed.

2.4 Deletion in Red Black Tree

Like the other basic operations on an n-node red-black tree, deletion of a node takes time O(lgn). Deleting a node from a red-black tree is only slightly more complicated than inserting a node.

The procedure RB-DELETE is a minor modification of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotations to restore the red black properties.

```
RB-DELETE(T, z)
 1  if left[z] = nil[T] or right[z] = nil[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          copy y's satellite data into z
16  if color[y] = BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ root[T] and color[x] = BLACK
 2      do if x = left[p[x]]
 3          then w ← right[p[x]]
 4               if color[w] = RED
 5                 then color[w] ← BLACK                          ▷ Case 1
 6                      color[p[x]] ← RED                         ▷ Case 1
 7                      LEFT-ROTATE(T, p[x])                      ▷ Case 1
 8                      w ← right[p[x]]                           ▷ Case 1
 9               if color[left[w]] = BLACK and color[right[w]] = BLACK
10                 then color[w] ← RED                            ▷ Case 2
11                      x ← p[x]                                  ▷ Case 2
12                 else if color[right[w]] = BLACK
13                        then color[left[w]] ← BLACK             ▷ Case 3
14                             color[w] ← RED                     ▷ Case 3
15                             RIGHT-ROTATE(T, w)                 ▷ Case 3
16                             w ← right[p[x]]                    ▷ Case 3
17                      color[w] ← color[p[x]]                    ▷ Case 4
18                      color[p[x]] ← BLACK                       ▷ Case 4
19                      color[right[w]] ← BLACK                   ▷ Case 4
20                      LEFT-ROTATE(T, p[x])                      ▷ Case 4
21                      x ← root[T]                               ▷ Case 4
22          else (same as then clause with "right" and "left" exchanged)
23  color[x] ← BLACK
```

A call to RBDELETE-FIXUP is made if y is black. If y is red, the red-black properties still hold when y is spliced out, for the following reasons:

• no black-heights in the tree have changed,

• no red nodes have been made adjacent, and

• since y could not have been the root if it was red, the root remains black

We can now examine how the procedure RB-DELETE-FIXUP restores the red black properties to the search tree.

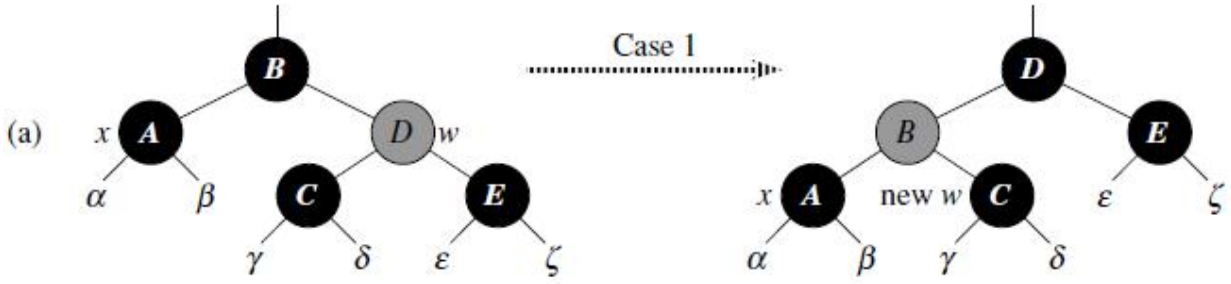
If the spliced-out node y in RB-DELETE is black, three problems may arise. First, if y had been the root and a red child of y becomes the new root, we have violated property 2. Second, if both x and p[y] (which is now also p[x]) were red, then we have violated property 4. Third, y's removal causes any path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree. We can correct this problem by saying that node x has an "extra" black. That is, if we add 1 to the count of black nodes on any path that contains x, then under this interpretation, property 5 holds. When we splice out the black node y, we "push" its blackness onto its child. The problem is that now node x is neither red nor black, thereby violating property 1. Instead, node x is either "doubly black" or "red-and-black," and it contributes either 2 or 1, respectively, to the count of black nodes on paths containing x. The color attribute of x will still be either RED (if x is red-and-black) or BLACK

(if x is doubly black). In other words, the extra black on a node is reflected in x's pointing to the node rather than in the color attribute.

Case 1: x's sibling w is red

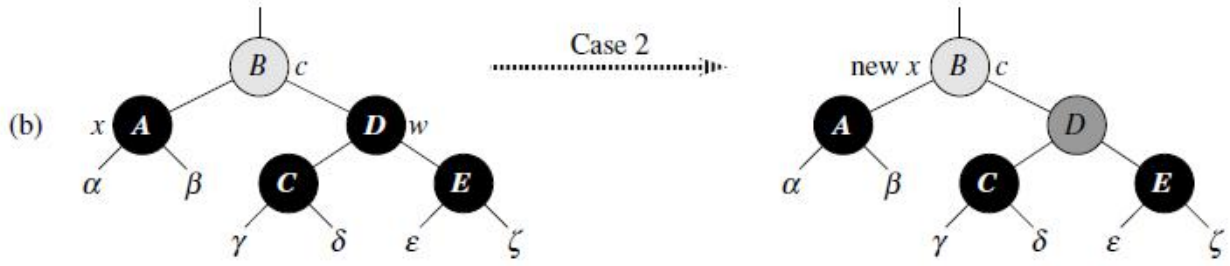
Case 1 (a) occurs when node *w*, the sibling of node *x*, is red. Since *w* must have black children, we can switch the colors of *w* and *p*[*x*] and then perform a left-rotation on *p*[*x*] without violating any of the red-black properties. The new sibling of *x*, which is one of *w*'s children prior to the rotation, is now black, and thus we have converted case 1 into case 2,3, or 4.

Cases 2, 3, and 4 occur when node *w* is black; they are distinguished by the colors of *w*'s children.



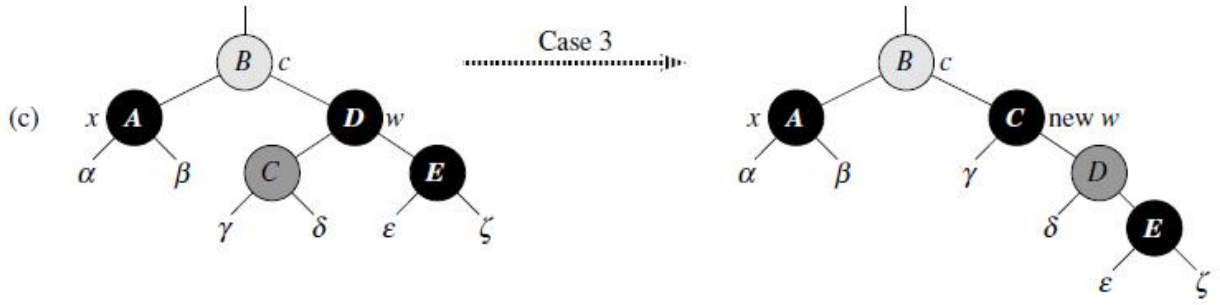
Case 2: x's sibling w is black, and both of w's children are black

In case (b), both of w's children are black. Since w is also black, we take one black off both x and w, leaving x with only one black and leaving w red. To compensate for removing one black from x and w, we would like to add an extra black to p[x], which was originally either red or black. We do so by repeating the while loop with p[x] as the new node x. Observe that if we enter case 2 through case 1, the new node x is red-and-black, since the original p[x] was red. Hence, the value c of the color attribute of the new node x is RED, and the loop terminates when it tests the loop condition. The new node x is then colored (singly) black.



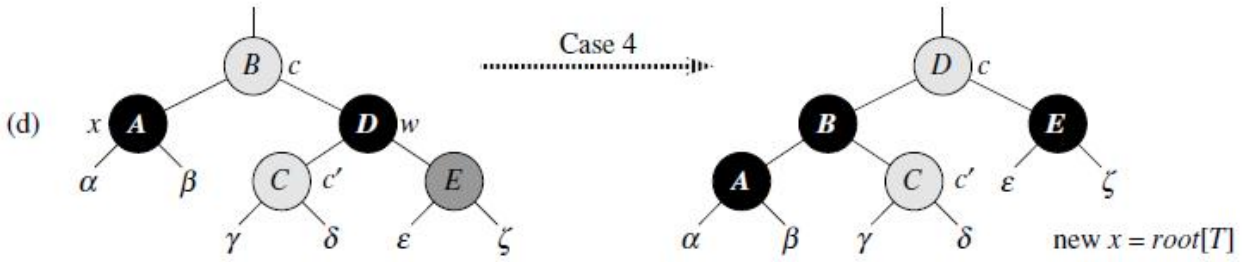
Case 3: x's sibling w is black, w's left child is red, and w's right child is black

Case 3 (c) occurs when w is black, its left child is red, and its right child is black. We can switch the colors of w and its left child left[w] and then perform a right rotation on w without violating any of the red-black properties. The new sibling w of x is now a black node with a red right child, and thus we have transformed case 3 into case 4.



Case 4: x's sibling w is black, and w's right child is red

Case 4 (d) occurs when node *x*'s sibling *w* is black and *w*'s right child is red. By making some color changes and performing a left rotation on *p*[*x*], we can remove the extra black on *x*, making it singly black, without violating any of the red-black properties. Setting *x* to be the root causes the **while** loop to terminate when it tests the loop condition.



2.4.1 Analysis

What is the running time of RB-DELETE? Since the height of a red-black tree of n nodes is O(lg n), the total cost of the procedure without the call to RB-DELETEFIXUP takes O(lg n) time. Within RB-DELETE-FIXUP, cases 1, 3, and 4 each terminate after performing a constant number of color changes and at most three rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves up the tree at most O(lg n) times and no rotations are performed. Thus, the procedure RB-DELETE-FIXUP takes O(lg n) time and performs at most three rotations, and the overall time for RB-DELETE is therefore also O(lg n).

2.5 B-Trees

B-trees are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage devices. B-trees are similar to red-black trees, but they are better at minimizing disk I/O operations. Many database systems use B-trees, or variants of B-trees, to store information.

B-trees differ from red-black trees in that B-tree nodes may have many children, from a handful to thousands. That is, the "branching factor" of a B-tree can be quite large, although it is usually determined by characteristics of the disk unit used. Btrees are similar to red-black trees in that every n-node B-tree has height O(lg n), although the height of a B-tree can be considerably less than that of a red-black tree because its branching factor can be much larger. Therefore, B-trees can also be used to implement many dynamic-set operations in time O(lg n).

2.5.1 Definition of B-trees

A common variant on a B-tree, known as a B+ tree, stores all the satellite information in the leaves and stores only keys and child pointers in the internal nodes, thus maximizing the branching factor of the internal nodes.

A B-tree T is a rooted tree (whose root is root[T]) having the following properties:

1. Every node x has the following fields:

a. n[x], the number of keys currently stored in node x,

b. the n[x] keys themselves, stored in nondecreasing order, so that key1[x] ≤key2[x] ≤ · · · ≤ keyn[x][x],

c. leaf [x], a boolean value that is TRUE if x is a leaf and FALSE if x is aninternal node.

2. Each internal node x also contains $n[x]+1$ pointers $c1[x], c2[x], \ldots, cn[x]+1[x]$ to its children. Leaf nodes have no children, so their ci fields are undefined.

3. The keys keyi [x] separate the ranges of keys stored in each subtree: if ki is any key stored in the subtree with root ci [x], then

$k1 \le key1[x] \le k2 \le key2[x] \le \cdots \le keyn[x][x] \le kn[x]+1$.

4. All leaves have the same depth, which is the tree's height h.

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \ge 2$ called the minimum degree of the B-tree:

a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

b. Every node can contain at most $2t - 1$ keys. Therefore, an internal node can have at most 2t children. We say that a node is full if it contains exactly $2t - 1$ keys.

2.5.2 The height of a B-tree

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree. We now analyze the worst-case height of a B-tree.

2.5.3 Theorem 18.1

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2} .$$

Proof If a B-tree has height h, the root contains at least one key and all other nodes contain at least $t - 1$ keys. Thus, there are at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth h there are at least $2t^{h-1}$ nodes. Figure 18.4 illustrates such a tree for $h = 3$. Thus, the number n of keys satisfies the inequality

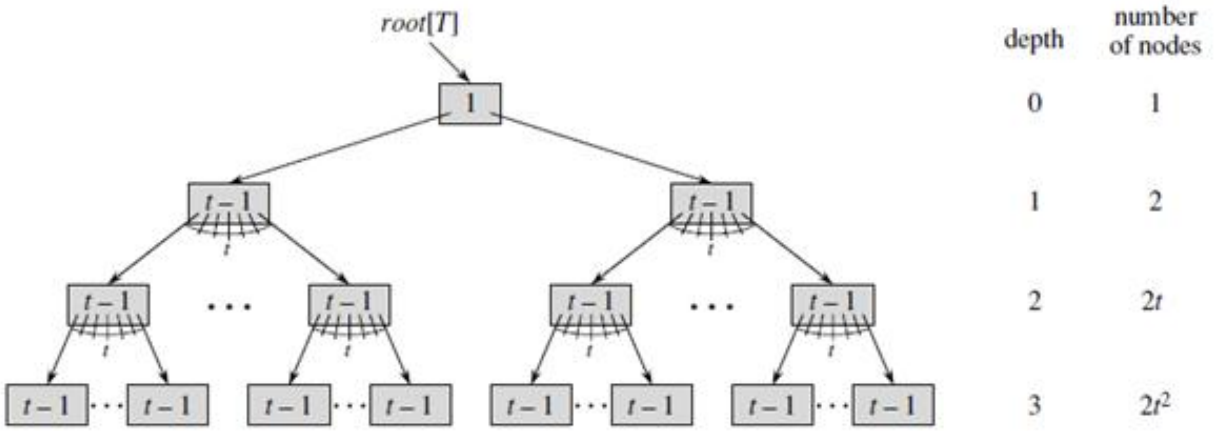


Figure:: A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $n[x]$.

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 . \end{aligned}$$

By simple algebra, we get $t^h \leq (n + 1)/2$. Taking base-t logarithms of both sides proves the theorem.

Here we see the power of B-trees, as compared to red-black trees. Although the height of the tree grows as $O(\lg n)$ in both cases (recall that t is a constant), for B-trees the base of the logarithm can be many times larger. Thus, B-trees save a factor of about $\lg t$ over red-black trees in the number of nodes examined for most tree operations. Since examining an arbitrary node in a tree usually requires a disk access, the number of disk accesses is substantially reduced.

2.5.4 Basic operations on B-trees

1. Searching a B-tree
2. Creating an empty B-tree
3. Inserting a key into a B-tree
4. Inserting a key into a B-tree in a single pass down the tree
5. Deleting a key from a B-tree

2.5.4.1 Searching a B-tree

```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] and k > key_i[x]
3        do i ← i + 1
4  if i ≤ n[x] and k = key_i[x]
5     then return (x, i)
6  if leaf[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9          return B-TREE-SEARCH(c_i[x], k)
```

As in the TREE-SEARCH procedure for binary search trees, the nodes encountered during the recursion form a path downward from the root of the tree. The number of disk pages accessed by B-TREE-SEARCH is therefore $O(h) = O(\log_t n)$, where h is the height of the B-tree and n is the number of keys in the B-tree. Since $n[x] < 2t$, the time taken by the **while** loop within each node is $O(t)$, and the total CPU time is $O(th) = O(t \log_t n)$.

2.5.4.2 Creating an empty B-tree

To build a B-tree T, we first use B-TREE-CREATE to create an empty root node and then call B-TREE-INSERT to add new keys. Both of these procedures use an auxiliary procedure ALLOCATE-NODE, which allocates one disk page to be used as a new node in O(1) time. We can assume that a node created by ALLOCATENODE requires no DISK-READ, since there is as yet no useful information stored on the disk for that node.

```
B-TREE-CREATE(T)
1  x ← ALLOCATE-NODE()
2  leaf[x] ← TRUE
3  n[x] ← 0
4  DISK-WRITE(x)
5  root[T] ← x
```

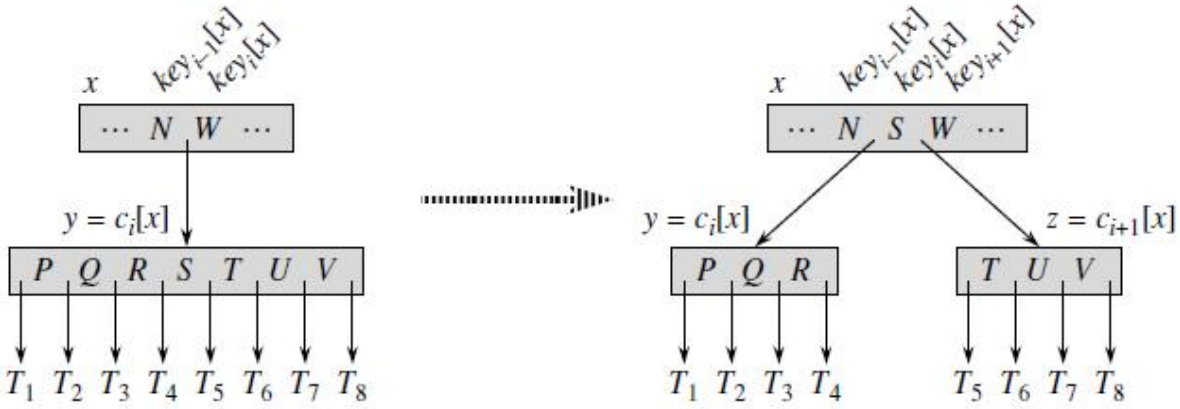
B-TREE-CREATE requires O(1) disk operations and O(1) CPU time.

2.5.4.3 Inserting a key into a B-tree

Inserting a key into a B-tree is significantly more complicated than inserting a key into a binary search tree. As with binary search trees, we search for the leaf position at which to insert the new key. With a B-tree, however, we cannot simply create a new leaf node and insert it, as the resulting tree would fail to be a valid B-tree. Instead, we insert the new key into an existing leaf node. Since we cannot insert a key into a leaf node that is full, we introduce an operation that splits a full node y (having 2t − 1 keys) around its median key keyt [y] into two nodes having t – 1 keys each. The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's parent is also full, it must be split before the new key can be inserted, and thus this need to split full nodes can propagate all the way up the tree.

2.5.4.4 Splitting a node in a B-tree

The procedure B-TREE-SPLIT-CHILD takes as input a nonfull internal node x (assumed to be in main memory), an index i , and a node y (also assumed to be in main memory) such that y = ci [x] is a full child of x. The procedure then splits this child in two and adjusts x so that it has an additional child. (To split a full root, we will first make the root a child of a new empty root node, so that we can use B-TREE-SPLIT-CHILD. The tree thus grows in height by one; splitting is the only means by which the tree grows.)



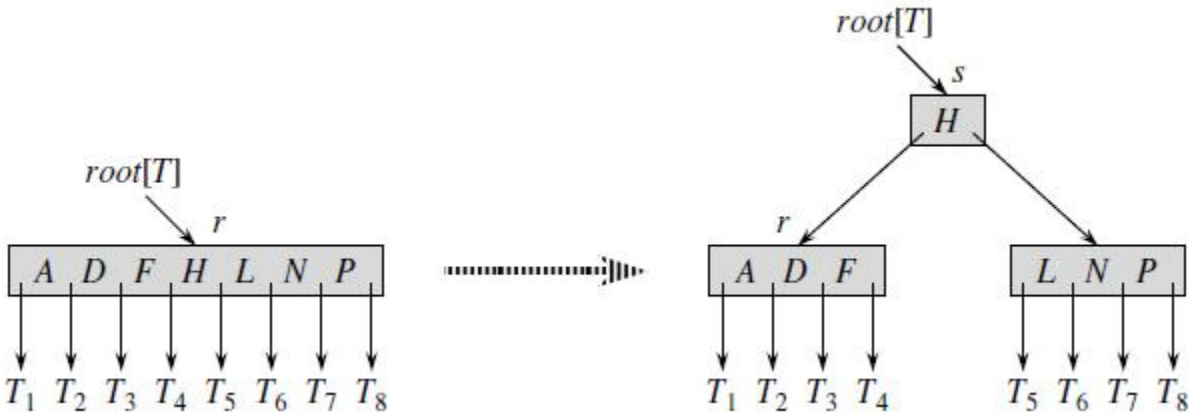
Splitting a node with t = 4. Node y is split into two nodes, y and z, and the median key S of y is moved up into y's parent.

```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE()
 2  leaf[z] ← leaf[y]
 3  n[z] ← t − 1
 4  for j ← 1 to t − 1
 5       do key_j[z] ← key_{j+t}[y]
 6  if not leaf[y]
 7     then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11       do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14       do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

2.5.4.5 Inserting a key into a B-tree in a single pass down the tree

We insert a key k into a B-tree T of height h in a single pass down the tree, requiring O(h) disk accesses. The CPU time required is O(th) = O(t logt n). The B-TREE-INSERT procedure uses B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.

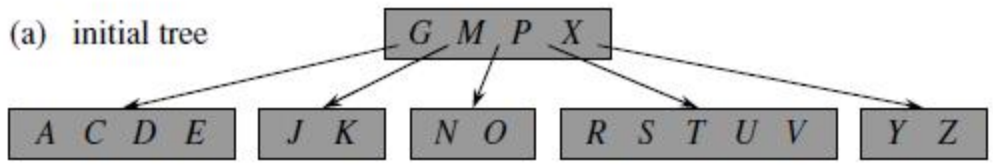
```
B-TREE-INSERT(T, k)
 1  r ← root[T]
 2  if n[r] = 2t − 1
 3     then s ← ALLOCATE-NODE()
 4          root[T] ← s
 5          leaf[s] ← FALSE
 6          n[s] ← 0
 7          c1[s] ← r
 8          B-TREE-SPLIT-CHILD(s, 1, r)
 9          B-TREE-INSERT-NONFULL(s, k)
10     else B-TREE-INSERT-NONFULL(r, k)
```

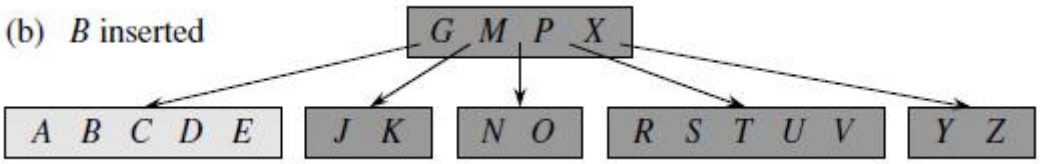


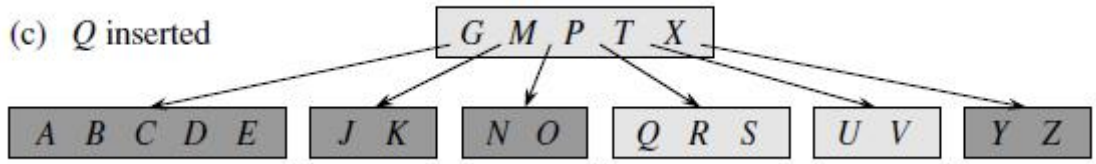
Splitting the root with $t = 4$. Root node r is split in two, and a new root node s is created. The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

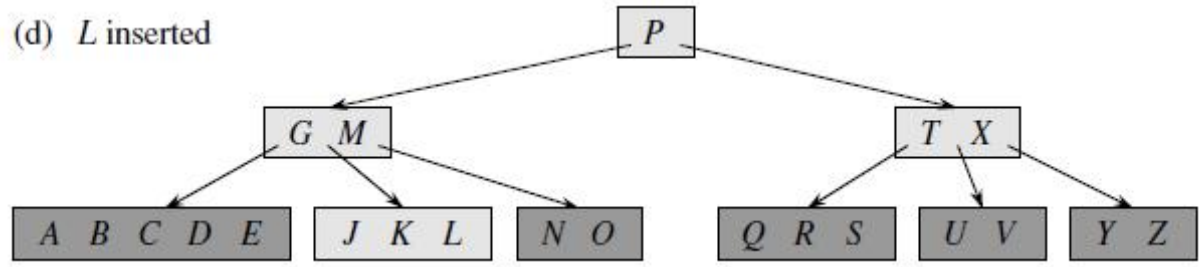
```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 and k < key_i[x]
 4             do key_{i+1}[x] ← key_i[x]
 5                i ← i − 1
 6          key_{i+1}[x] ← k
 7          n[x] ← n[x] + 1
 8          DISK-WRITE(x)
 9     else while i ≥ 1 and k < key_i[x]
10             do i ← i − 1
11          i ← i + 1
12          DISK-READ(c_i[x])
13          if n[c_i[x]] = 2t − 1
14            then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                 if k > key_i[x]
16                   then i ← i + 1
17          B-TREE-INSERT-NONFULL(c_i[x], k)
```

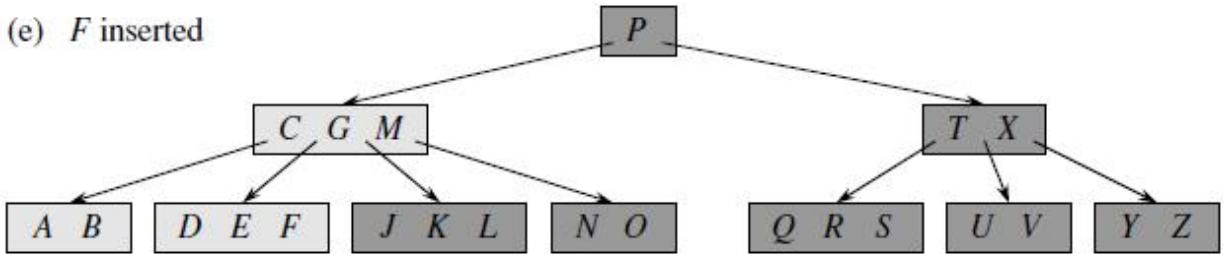
The number of disk accesses performed by B-TREE-INSERT is *O(h)* for a Btree of height *h*, since only *O(1)* DISK-READ and DISK-WRITE operations are performed between calls to B-TREE-INSERT-NONFULL. The total CPU time used is $O(th) = O(t \log_t n)$. Since B-TREE-INSERT-NONFULL is tail-recursive, it can be alternatively implemented as a **while** loop, demonstrating that the number of pages that need to be in main memory at any time is *O(1)*.







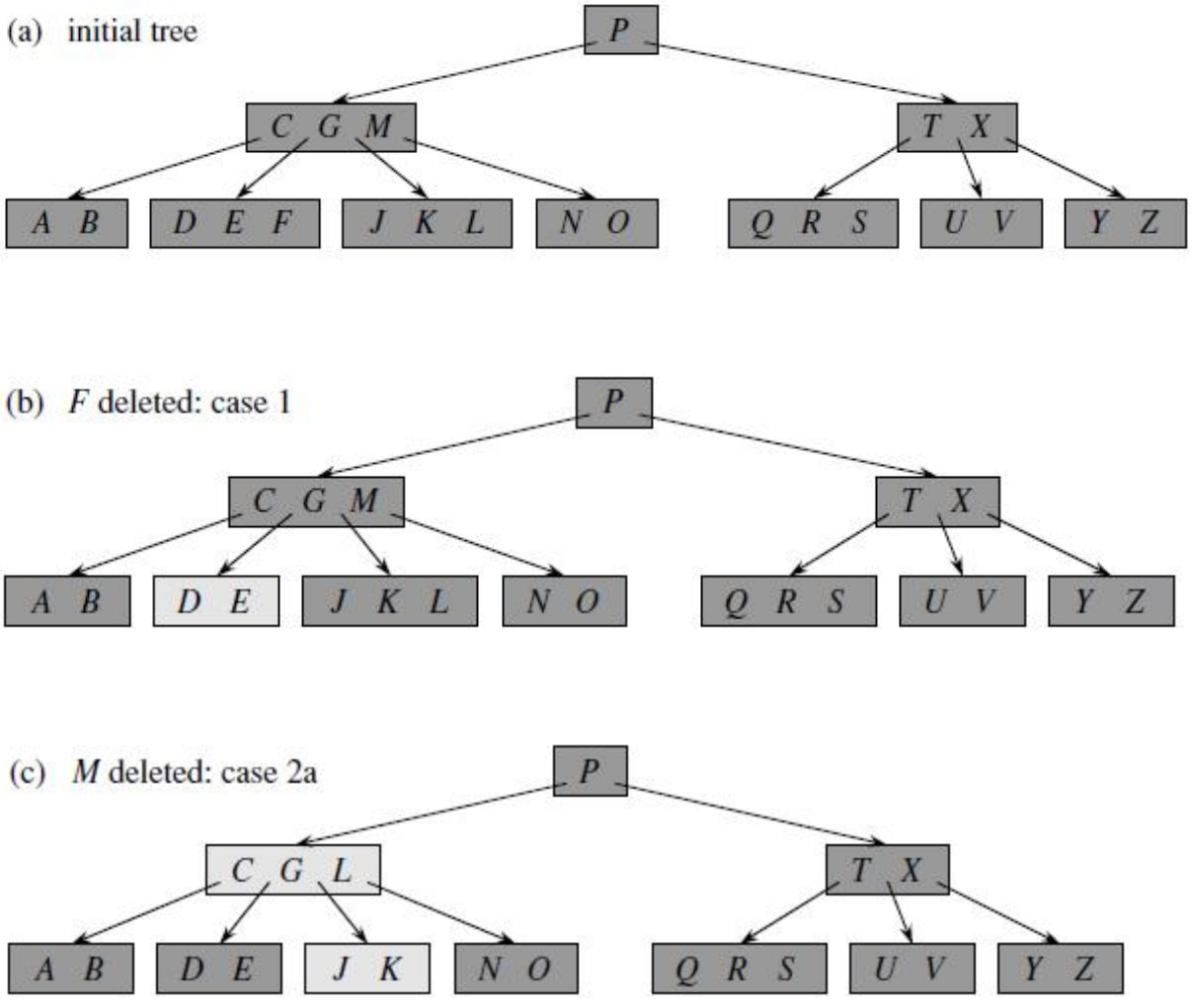


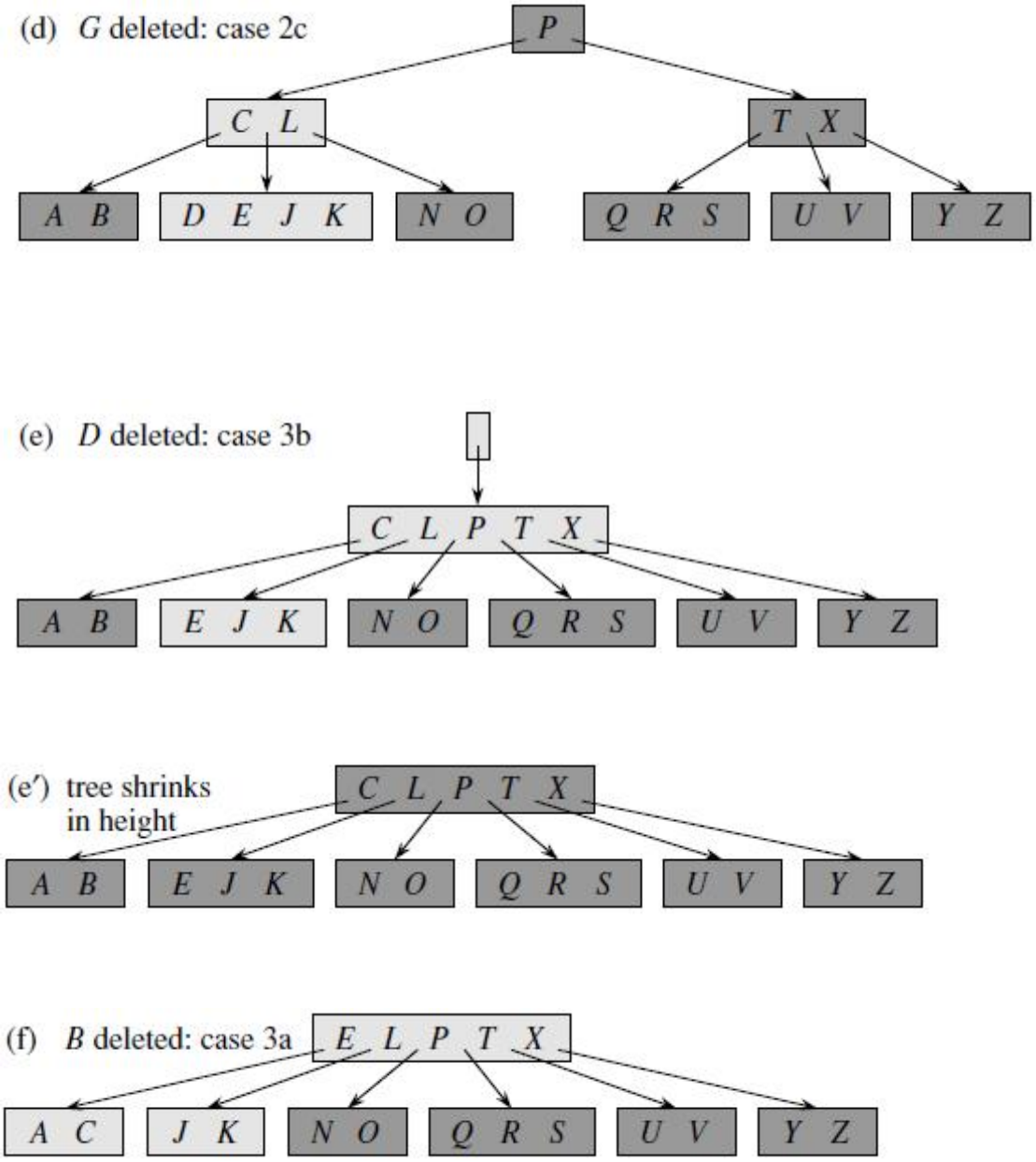


Inserting keys into a B-tree. The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded. **(a)** The initial tree for this example. **(b)** The result of inserting *B* into the initial tree; this is a simple insertion into a leaf node. **(c)** The result of inserting *Q* into the previous tree. The node *RSTUV* is split into two nodes containing *RS* and *UV*, the key *T* is moved up to the root, and *Q* is inserted in the leftmost of the two halves (the *RS* node). **(d)** The result of inserting *L* into the previous tree. The root is split right away, since it is full, and the B-tree grows in height by one. Then *L* is inserted into the leaf containing *J K*. **(e)** The result of inserting *F* into the previous tree. The node *ABCDE* is split before *F* is inserted into the rightmost of the two halves (the *DE* node).

2.5.4.6 Deleting a key from a B-tree

Deletion from a B-tree is analogous to insertion but a little more complicated, because a key may be deleted from any node—not just a leaf—and deletion from an internal node requires that the node's children be rearranged. As in insertion, we must guard against deletion producing a tree whose structure violates the B-tree properties. Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number $t - 1$ of keys, though it is not allowed to have more than the maximum number $2t - 1$ of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.





Deleting keys from a B-tree. The minimum degree for this B-tree is t = 3, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of F. This is case 1: simple deletion from a leaf. **(c)** Deletion of M. This is case 2a: the predecessor L of M is moved up to take M's position. **(d)** Deletion of G. This is case 2c: G is pushed down to make node DEGJ K, and then G is deleted from this leaf (case 1). **(e)** Deletion of D. This is case 3b: the recursion can't descend to node CL because it has only 2 keys, so P is pushed down and merged with CL and T X to form CLPT X; then, D is deleted from a leaf (case 1). **(e_)** After (d), the root is deleted and the tree shrinks in height by one. **(f)** Deletion of B. This is case 3a: C is moved to fill B's position and E is moved to fill C's position.

In above figures we have illustrates the various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x.

2. If the key k is in node x and x is an internal node, do the following.

a. If the child y that precedes k in node x has at least t keys, then find the

predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)

b. Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)

c. Otherwise, if both y and z have only t - 1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t$ - 1 keys. Then, free z and recursively delete k from y.

3. If the key k is not present in internal node x, determine the root $ci[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $ci[x]$ has only t - 1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of x.

a. If $ci[x]$ has only t - 1 keys but has an immediate sibling with at least t keys, give $ci[x]$ an extra key by moving a key from x down into $ci[x]$, moving a key from $ci[x]$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $ci[x]$.

b. If $ci[x]$ and both of $ci[x]$'s immediate siblings have t - 1 keys, merge $ci[x]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Since most of the keys in a B-tree are in the leaves, we may expect that in practice, deletion operations are most often used to delete keys from leaves. The B-TREE-DELETE procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

Although this procedure seems complicated, it involves only $O(h)$ disk operations for a B-tree of height h, since only $O(1)$ calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure. The CPU time required is $O(th) = O(t \log_t n)$.

2.6 Binomial Heaps

Binomial Heaps and Fibonacci Heaps data structures known as ***mergeable heaps***, which support the following five operations

MAKE-HEAP() creates and returns a new heap containing no elements.

INSERT(H, x) inserts node x, whose key field has already been filled in, into heap H.

MINIMUM(H) returns a pointer to the node in heap H whose key is minimum.

EXTRACT-MIN(H) deletes the node from heap H whose key is minimum, returning a pointer to the node.

UNION(H1, H2) creates and returns a new heap that contains all the nodes of heaps H1 and H2. Heaps H1 and H2 are "destroyed" by this operation.

In addition, these data structures also support the following two operations.

DECREASE-KEY(H, x, k) assigns to node x within heap H the new key value k, which is assumed to be no greater than its current key value.1

DELETE(H, x) deletes node x from heap H.

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Running times for operations on three implementations of mergeable heaps. The number of items in the heap(s) at the time of an operation is denoted by n.

2.6.1 Binomial trees and binomial heaps

A binomial heap is a collection of binomial trees, so this section starts by defining binomial trees and proving some key properties. We then define binomial heaps and show how they can be represented.

2.6.2 Binomial trees

The binomial tree Bk is an ordered tree defined recursively. As shown in below Figure (a), the binomial tree B0 consists of a single node. The binomial tree Bk consists of two binomial trees Bk−1 that are linked together: the root of one is the leftmost child of the root of the other. Figure (b) shows the binomial trees B0 through B4.

Some properties of binomial trees are given by the following lemma.

2.6.3 Lemma (Properties of binomial trees)

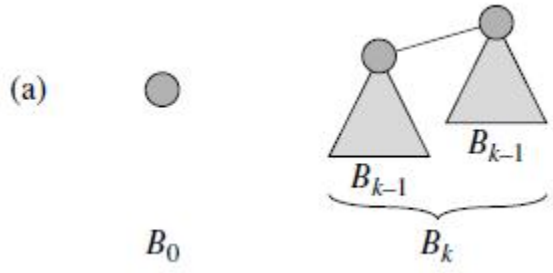
For the binomial tree B_k,

1. there are 2^k nodes,
2. the height of the tree is k,
3. there are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$, and
4. the root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, child i is the root of a subtree B_i.

Proof The proof is by induction on k. For each property, the basis is the binomial tree B_0. Verifying that each property holds for B_0 is trivial.

For the inductive step, we assume that the lemma holds for B_{k-1}.

1. Binomial tree B_k consists of two copies of B_{k-1}, and so B_k has $2^{k-1}+2^{k-1} = 2^k$ nodes.
2. Because of the way in which the two copies of B_{k-1} are linked to form B_k, the maximum depth of a node in B_k is one greater than the maximum depth in B_{k-1}. By the inductive hypothesis, this maximum depth is $(k-1)+1 = k$.
3. Let $D(k, i)$ be the number of nodes at depth i of binomial tree B_k. Since B_k is composed of two copies of B_{k-1} linked together, a node at depth i in B_{k-1} appears in B_k once at depth i and once at depth $i+1$. In other words, the number of nodes at depth i in B_k is the number of nodes at depth i in B_{k-1} plus



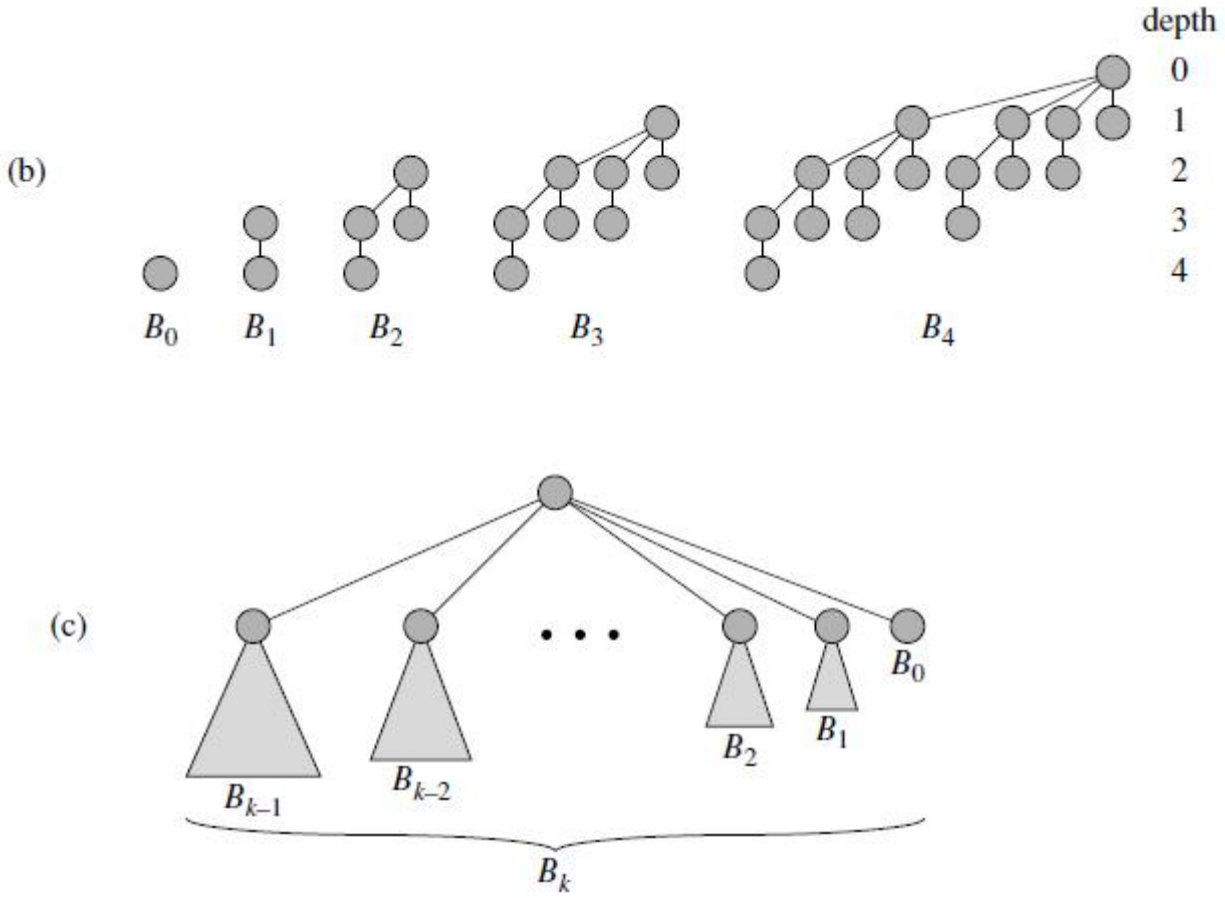


Figure: (a) The recursive definition of the binomial tree *Bk*. Triangles represent rooted subtrees. **(b)** The binomial trees *B*0 through *B*4. Node depths in *B*4 are shown. **(c)** Another way of looking at the binomial tree *Bk*.

the number of nodes at depth $i - 1$ in B_{k-1}. Thus,

$$\begin{aligned} D(k,i) &= D(k-1,i) + D(k-1,i-1) \quad \text{(by the inductive hypothesis)} \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} \\ &= \binom{k}{i} . \end{aligned}$$

4. The only node with greater degree in B_k than in B_{k-1} is the root, which has one more child than in B_{k-1}. Since the root of B_{k-1} has degree $k-1$, the root of B_k has degree k. Now, by the inductive hypothesis, and as Figure 19.2(c) shows, from left to right, the children of the root of B_{k-1} are roots of $B_{k-2}, B_{k-3}, \ldots, B_0$. When B_{k-1} is linked to B_{k-1}, therefore, the children of the resulting root are roots of $B_{k-1}, B_{k-2}, \ldots, B_0$.

A **binomial heap** H is a set of binomial trees that satisfies the following **binomial heap properties**.

1. Each binomial tree in H obeys the **min-heap property**: the key of a node is greater than or equal to the key of its parent. We say that each such tree is **min-heap-ordered**.

2. For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

The first property tells us that the root of a min-heap-ordered tree contains the smallest key in the tree.

The second property implies that an n-node binomial heap H consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees. To see why, observe that the binary representation of n has $\lfloor \lg n \rfloor + 1$ bits, say $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \ldots, b_0 \rangle$, so that $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$. By property 1 of Lemma , therefore, binomial tree B_i appears in H if and only if bit $b_i = 1$. Thus, binomial heap H contains at most $\lfloor \lg n \rfloor + 1$ binomial trees.

Figure (a) shows a binomial heap H with 13 nodes. The binary representation of 13 is $\langle 1101 \rangle$, and H consists of min-heap-ordered binomial trees B_3, B_2, and B_0, having 8, 4, and 1 nodes respectively, for a total of 13 nodes.

2.6.4 Representing binomial heaps

As shown in Figure, each binomial tree within a binomial heap is stored in the left child, right-sibling representation. Each node has a *key* field and any other satellite information required by the application. In addition, each node x contains pointers $p[x]$ to its parent, *child*$[x]$ to its leftmost child, and *sibling*$[x]$ to the sibling of x immediately to its right. If node x is a root, then $p[x]$ = NIL. If node x has no children, then *child*$[x]$ = NIL, and if x is the rightmost child of its parent, then *sibling*$[x]$ = NIL. Each node x also contains the field *degree*$[x]$, which is the number of children of x.

As Figure also shows, the roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the ***root list***. The degrees of the roots strictly increase as we traverse the root list. By the second binomial-heap property, in an n-node binomial heap the degrees of the roots are a subset of {0, 1, ..., $\lfloor \lg n \rfloor$ }. The *sibling* field has a different meaning

for roots than for nonroots. If x is a root, then $sibling[x]$ points to the next root in the root list. (As usual, $sibling[x]$ = NIL if x is the last root in the root list.)

A given binomial heap H is accessed by the field $head[H]$, which is simply a pointer to the first root in the root list of H . If binomial heap H has no elements, then $head[H]$ = NIL.

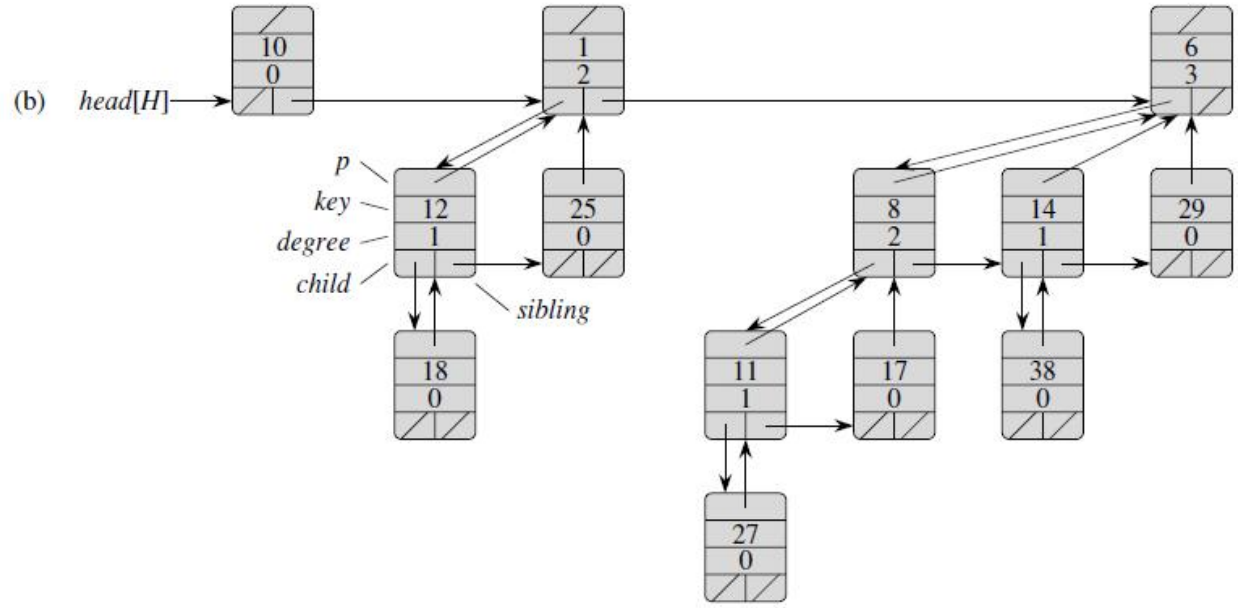


Figure: A binomial heap H with n = 13 nodes. **(a)** The heap consists of binomial trees $B0$, $B2$, and $B3$, which have 1, 4, and 8 nodes respectively, totaling n = 13 nodes. Since each binomial tree is min-heap-ordered, the key of any node is no less than the key of its parent. Also shown is the root list, which is a linked list of roots in order of increasing degree. **(b)** A more detailed representation of binomial heap H. Each binomial tree is stored in the left-child, right-sibling representation, and each node stores its degree.

2.6.5 Operations on binomial heaps

Creating a new binomial heap

1. Finding the minimum key
2. Uniting two binomial heaps
3. Inserting a node
4. Extracting the node with minimum key
5. Decreasing a key

2.6.5.1 Creating a new binomial heap

To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates and returns an object H, where head[H] = NIL. The running time is Θ (1).

2.6.5.2 Finding the minimum key

The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H. This implementation assumes that there are no keys with value ∞.

```
BINOMIAL-HEAP-MINIMUM(H)
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x ≠ NIL
5        do if key[x] < min
6              then min ← key[x]
7                   y ← x
8           x ← sibling[x]
9  return y
```

Since a binomial heap is min-heap-ordered, the minimum key must reside in a root node. The BINOMIAL-HEAP-MINIMUM procedure checks all roots, which number at most $\lfloor \lg n \rfloor + 1$, saving the current minimum in min and a pointer to the current minimum in y. When called on the binomial heap of Figure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with key 1.

Because there are at most $\lfloor \lg n \rfloor + 1$ roots to check, the running time of BINOMIAL-HEAP-MINIMUM is $O(\lg n)$.

2.6.5.3 Uniting two binomial heaps

The operation of uniting two binomial heaps is used as a subroutine by most of the remaining operations. The BINOMIAL-HEAP-UNION procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the B_{k-1} tree rooted at node y to the B_{k-1} tree rooted at node z; that is, it makes z the parent of y. Node z thus becomes the root of a B_k tree.

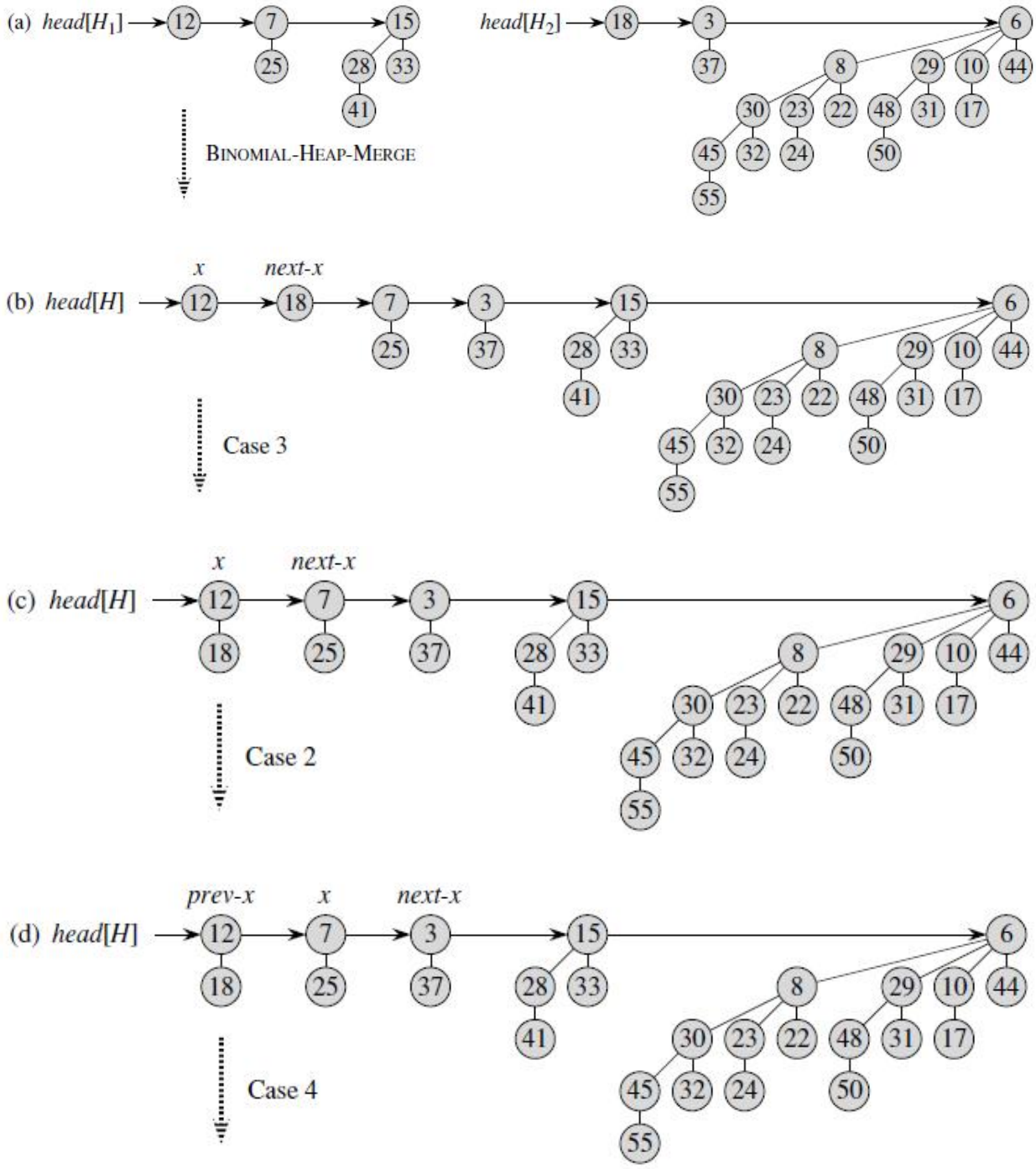
```
BINOMIAL-LINK(y, z)
1  p[y] ← z
2  sibling[y] ← child[z]
3  child[z] ← y
4  degree[z] ← degree[z] + 1
```

The BINOMIAL-LINK procedure makes node y the new head of the linked list of node z's children in O(1) time. It works because the left-child, right-sibling representation of each binomial tree matches the ordering property of the tree: in a B_k tree, the leftmost child of the root is the root of a B_{k-1} tree.

The following procedure unites binomial heaps H1 and H2, returning the resulting heap. It destroys the representations of H1 and H2 in the process. Besides BINOMIAL-LINK, the procedure uses an auxiliary procedure BINOMIALHEAP- MERGE that merges the root lists of H1 and H2 into a single linked list that is sorted by degree into monotonically increasing order. The BINOMIAL-HEAPMERGE procedure, whose pseudocode, is similar to the MERGE procedure.

```
BINOMIAL-HEAP-UNION(H1, H2)
 1  H ← MAKE-BINOMIAL-HEAP()
 2  head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
 3  free the objects H1 and H2 but not the lists they point to
 4  if head[H] = NIL
 5     then return H
 6  prev-x ← NIL
 7  x ← head[H]
 8  next-x ← sibling[x]
 9  while next-x ≠ NIL
10        do if (degree[x] ≠ degree[next-x]) or
                 (sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
11              then prev-x ← x                                  ▷ Cases 1 and 2
12                   x ← next-x                                  ▷ Cases 1 and 2
13              else  if key[x] ≤ key[next-x]
14                       then sibling[x] ← sibling[next-x]       ▷ Case 3
15                            BINOMIAL-LINK(next-x, x)           ▷ Case 3
16                       else  if prev-x = NIL                   ▷ Case 4
17                                then head[H] ← next-x          ▷ Case 4
18                                else  sibling[prev-x] ← next-x ▷ Case 4
19                            BINOMIAL-LINK(x, next-x)           ▷ Case 4
20                            x ← next-x                         ▷ Case 4
21           next-x ← sibling[x]
22  return H
```

Below Figure shows an example of BINOMIAL-HEAP-UNION in which all four cases given in the pseudocode occur.


(a) head[H1]
12
7
15
25
28
33
41
BINOMIAL-HEAP-MERGE
head[H2]
18
3
6
37
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55
(b) head[H]
x
next-x
12
18
7
3
15
6
25
37
28
33
41
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55
Case 3
(c) head[H]
x
next-x
12
7
3
15
6
18
25
37
28
33
41
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55
Case 2
(d) head[H]
prev-x
x
next-x
12
7
3
15
6
18
25
37
28
33
41
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55
Case 4

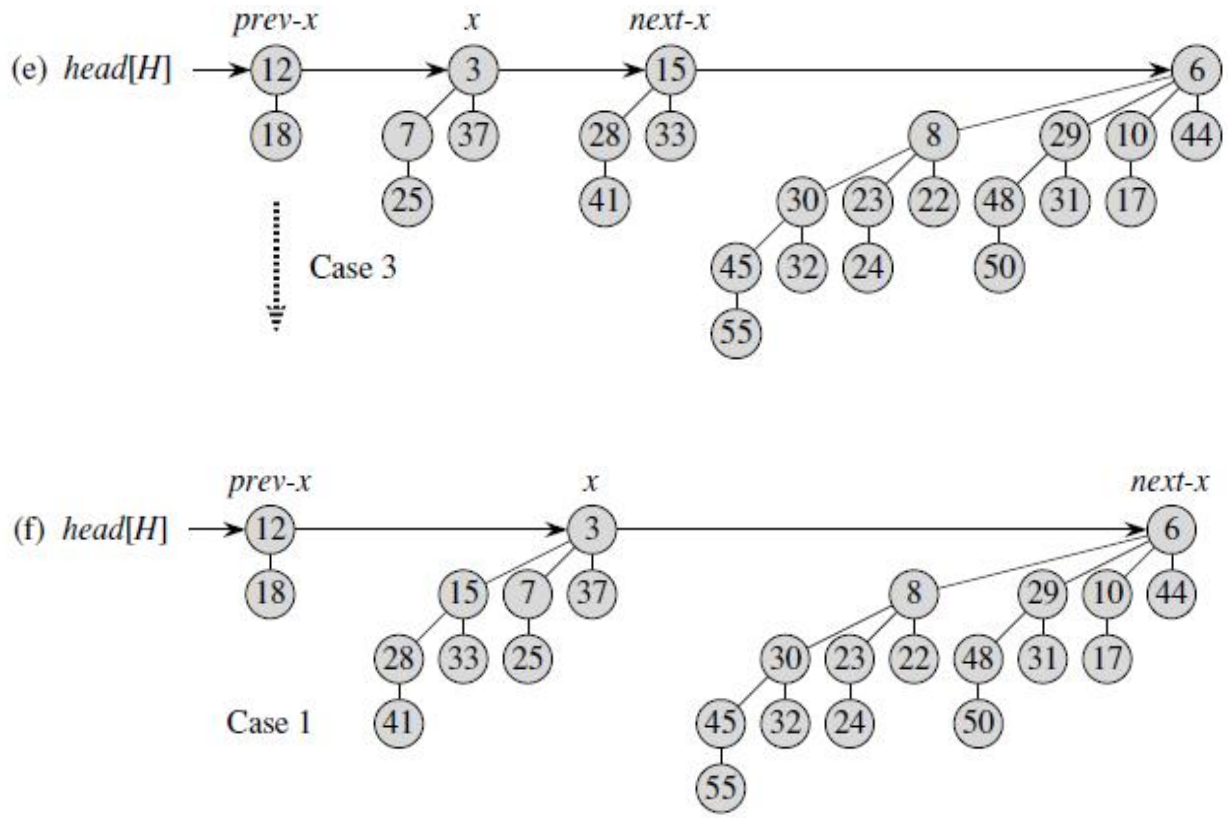


Figure: The execution of BINOMIAL-HEAP-UNION. **(a)** Binomial heaps *H*1 and *H*2. **(b)** Binomial heap *H* is the output of BINOMIAL-HEAP-MERGE*(H*1*, H*2*)*. Initially, *x* is the first root on the root list of *H*. Because both *x* and *next-x* have degree 0 and *key*[*x*] < *key*[*next-x*], case 3 applies. **(c)** After the link occurs, *x* is the first of three roots with the same degree, so case 2 applies. **(d)** After all the pointers move down one position in the root list, case 4 applies, since *x* is the first of two roots of equal degree. **(e)** After the link occurs, case 3 applies. **(f)** After another link, case 1 applies, because *x* has degree 3 and *next-x* has degree 4. This iteration of the **while** loop is the last, because after the pointers move down one position in the root list, *next-x* = NIL.

Case 1, shown in Figure (a), occurs when ***degree*[*x*] =! *degree*[*next-x*],** that is, when *x* is the root of a *Bk*-tree and *next-x* is the root of a *Bl*-tree for some $l > k$.

To handle this case. We don't link x and next-x, so we simply march the pointers one position farther down the list.

Case 2, shown in Figure (b), occurs when x is the first of three roots of equal degree, that is, when **degree[x] = degree[next-x] = degree[sibling[next-x]] .**

We handle this case in the same manner as case 1: we just march the pointers one position farther down the list.

Cases 3 and 4 occur when x is the first of two roots of equal degree, that is, when

degree[x] = degree[next-x] _= degree[sibling[next-x]] .

These cases may occur in any iteration, but one of them always occurs immediately following case 2. In cases 3 and 4, we link x and next-x. The two cases are distinguished by whether x or next-x has the smaller key, which determines the node that will be the root after the two are linked.

In case 3, shown in Figure (c), key[x] ≤ key[next-x], so next-x is linked to x. removes next-x from the root list, and makes next-x the leftmost child of x.

In case 4, shown in Figure (d), next-x has the smaller key, so x is linked to next-x. Remove x from the root list; there are two cases depending on whether x is the first root on the list or is not. Then makes x the leftmost child of next-x, and updates x for the next iteration.

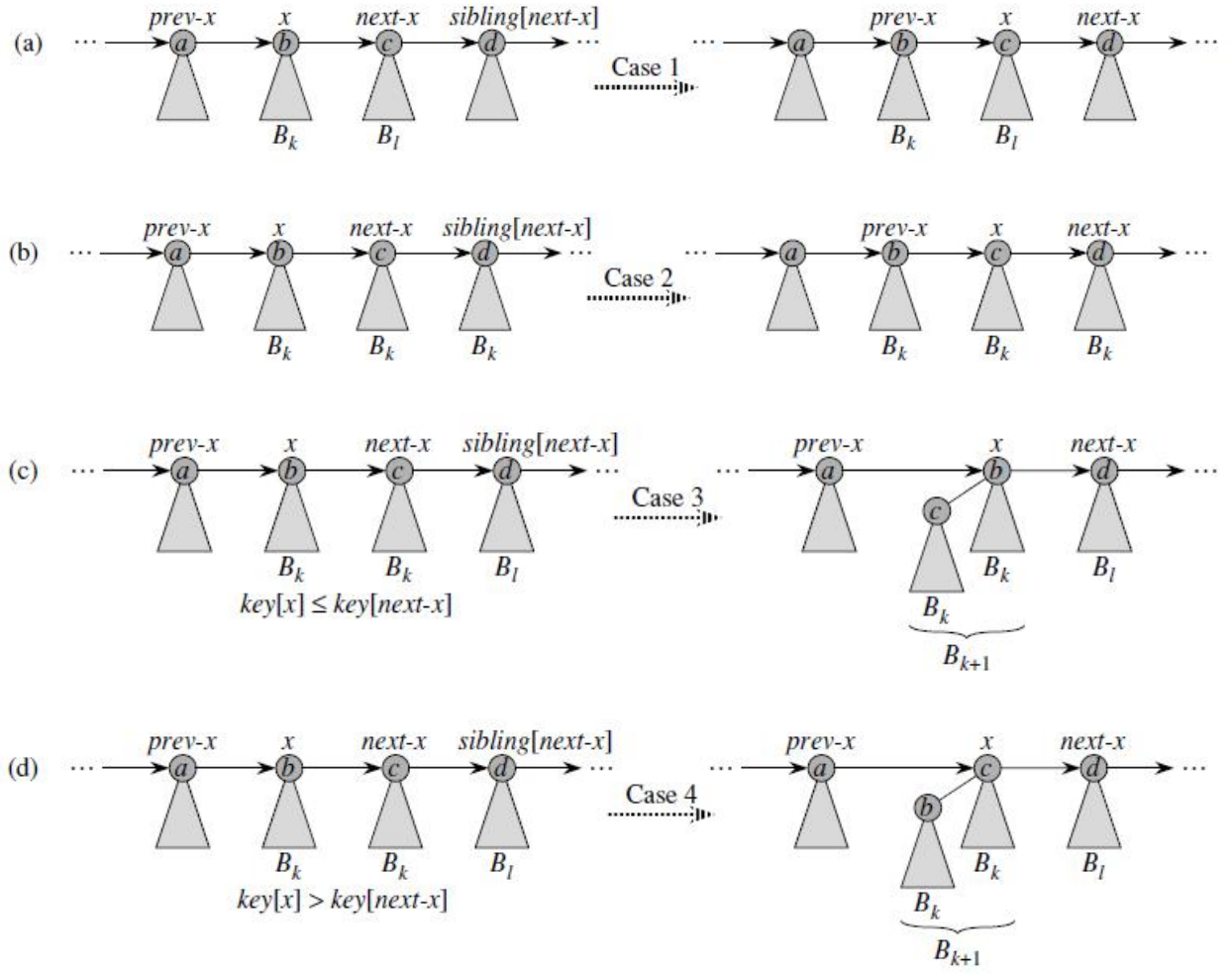


Figure The four cases that occur in BINOMIAL-HEAP-UNION. Labels a, b, c, and d serve only to identify the roots involved; they do not indicate the degrees or keys of these roots. In each case, x is the root of a B_k-tree and $l > k$. (a) Case 1: $degree[x] \neq degree[next\text{-}x]$. The pointers move one position farther down the root list. (b) Case 2: $degree[x] = degree[next\text{-}x] = degree[sibling[next\text{-}x]]$. Again, the pointers move one position farther down the list, and the next iteration executes either case 3 or case 4. (c) Case 3: $degree[x] = degree[next\text{-}x] \neq degree[sibling[next\text{-}x]]$ and $key[x] \leq key[next\text{-}x]$. We remove $next\text{-}x$ from the root list and link it to x, creating a B_{k+1}-tree. (d) Case 4: $degree[x] = degree[next\text{-}x] \neq degree[sibling[next\text{-}x]]$ and $key[next\text{-}x] \leq key[x]$. We remove x from the root list and link it to $next\text{-}x$, again creating a B_{k+1}-tree.

2.6.5.4 Inserting a node

The following procedure inserts node x into binomial heap H, assuming that x has already been allocated and $key[x]$ has already been filled in.

```
BINOMIAL-HEAP-INSERT(H, x)
1  H' ← MAKE-BINOMIAL-HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL-HEAP-UNION(H, H')
```

The procedure simply makes a one-node binomial heap H_ in O(1) time and unites it with the n-node binomial heap H in O(lg n) time. The call to BINOMIAL-HEAPUNION takes care of freeing the temporary binomial heap H'.

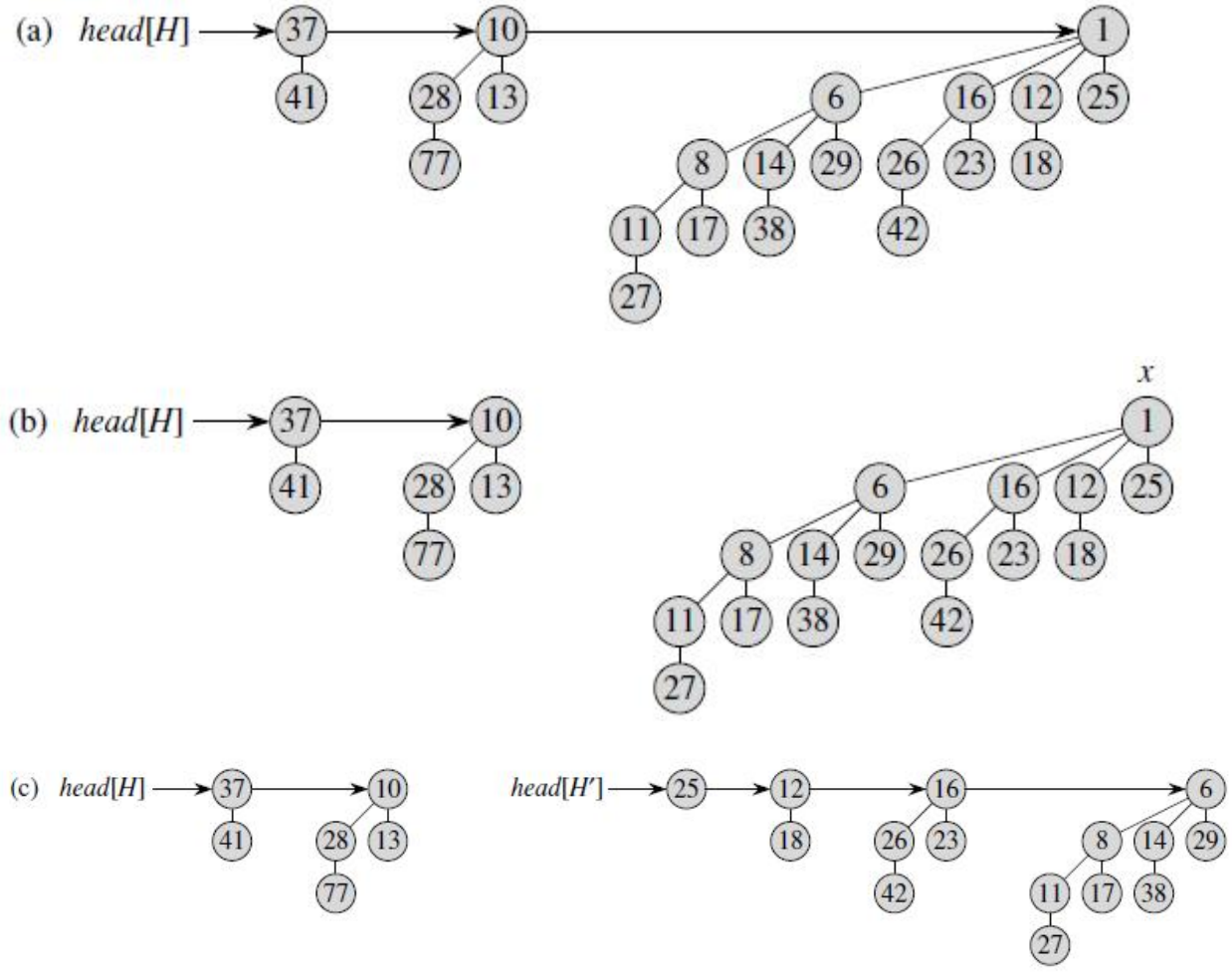
2.6.5.5 Extracting the node with minimum key

The following procedure extracts the node with the minimum key from binomial heap H and returns a pointer to the extracted node.

BINOMIAL-HEAP-EXTRACT-MIN(H)

1 find the root x with the minimum key in the root list of H,
and remove x from the root list of H
2 $H' \leftarrow$ MAKE-BINOMIAL-HEAP()
3 reverse the order of the linked list of x's children,
and set $head[H']$ to point to the head of the resulting list
4 $H \leftarrow$ BINOMIAL-HEAP-UNION(H, H')
5 **return** x

BINOMIAL-HEAPEXTRACT-MIN runs in O(lg n) time.



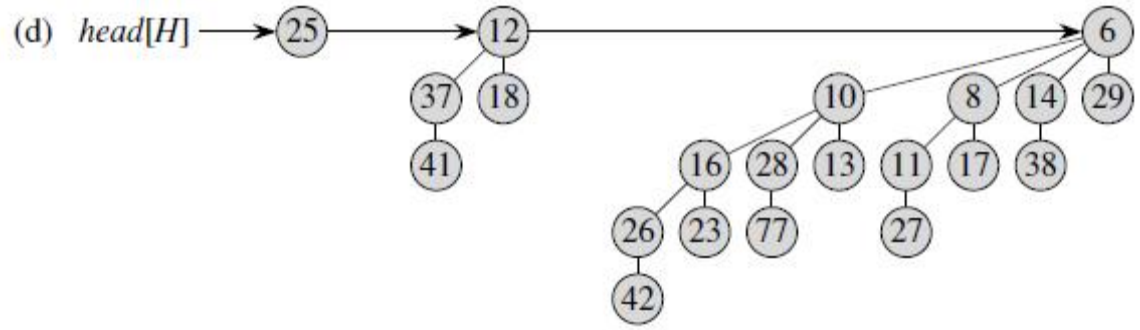


Figure: The action of BINOMIAL-HEAP-EXTRACT-MIN. *(a)* A binomial heap *H*. *(b)* The root *x* with minimum key is removed from the root list of *H* . *(c)* The linked list of *x*'s children is reversed, giving another binomial heap *H'*. *(d)* The result of uniting *H* and *H'*.

2.6.5.6 Decreasing a key

The following procedure decreases the key of a node *x* in a binomial heap *H* to a new value *k*. It signals an error if *k* is greater than *x*'s current key.

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
 1  if k > key[x]
 2     then error "new key is greater than current key"
 3  key[x] ← k
 4  y ← x
 5  z ← p[y]
 6  while z ≠ NIL and key[y] < key[z]
 7        do exchange key[y] ↔ key[z]
 8           ▷ If y and z have satellite fields, exchange them, too.
 9           y ← z
10           z ← p[y]
```

The BINOMIAL-HEAP-DECREASE-KEY procedure takes O(lg n) time.

2.6.5.7 Deleting a key

It is easy to delete a node *x*'s key and satellite information from binomial heap *H* in $O(\lg n)$ time. The following implementation assumes that no node currently in the binomial heap has a key of $-\infty$.

```
BINOMIAL-HEAP-DELETE(H, x)
1  BINOMIAL-HEAP-DECREASE-KEY(H, x, −∞)
2  BINOMIAL-HEAP-EXTRACT-MIN(H)
```

The BINOMIAL-HEAP-DELETE procedure makes node x have the unique minimum key in the entire binomial heap by giving it a key of $-\infty$. Deals with the situation in which $-\infty$ cannot appear as a key, even temporarily.) It then bubbles this key and the associated satellite information up to a root by calling BINOMIAL-HEAP-DECREASE-KEY. This root is then removed from H by a call of BINOMIAL-HEAP-EXTRACT-MIN. The BINOMIAL-HEAP-DELETE procedure takes O(lg n) time.

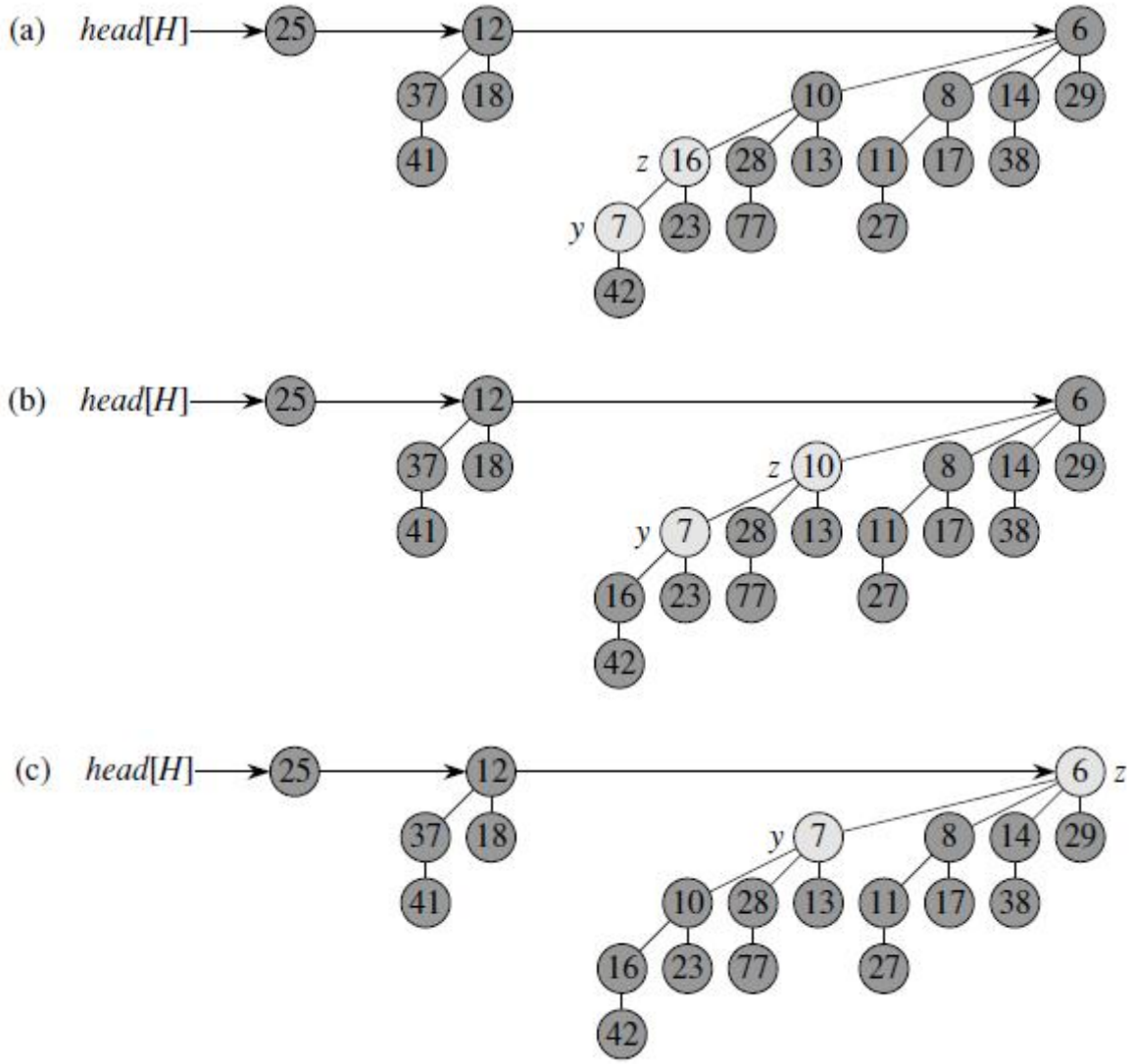


Figure: The action of BINOMIAL-HEAP-DECREASE-KEY. **(a)** The situation just before line 6 of the first iteration of the **while** loop. Node *y* has had its key decreased to 7, which is less than

the key of *y*'s parent *z*. **(b)** The keys of the two nodes are exchanged, and the situation just before line 6 of the second iteration is shown. Pointers *y* and *z* have moved up one level in the tree, but min-heap order is still violated. **(c)** After another exchange and moving pointers *y* and *z* up one more level, we find that min-heap order is satisfied, so the **while** loop terminates.

2.7 Fibonacci Heaps

2.7.1 Structure of Fibonacci heaps

Like a binomial heap, a Fibonacci heap is a collection of min-heap-ordered trees. The trees in a Fibonacci heap are not constrained to be binomial trees, however. Figure below shows an example of a Fibonacci heap.

Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but unordered. As Figure (b) shows, each node x contains a pointer p[x] to its parent and a pointer child[x] to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the child list of x. Each child y in a child list has pointers left[y] and right[y] that point to y's left and right siblings, respectively. If node y is an only child, then left[y] = right[y] = y. The order in which siblings appear in a child list is arbitrary.

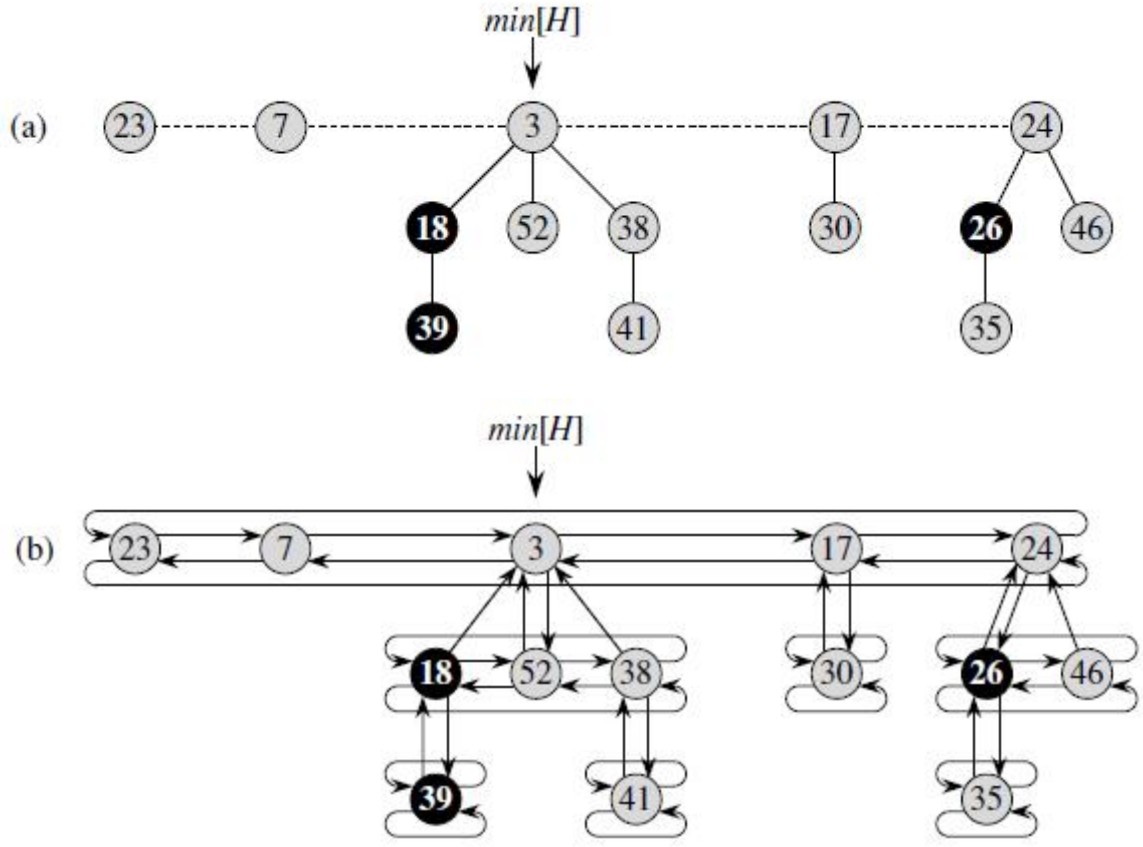


Figure: (a) A Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. The three marked nodes are blackened. The potential of this particular Fibonacci heap is 5+2·3 = 11. **(b)** A more complete representation showing pointers *p* (up arrows), *child* (down arrows), and *left* and *right* (sideways arrows). These details are omitted in the remaining figures in this chapter, since all the information shown here can be determined from what appears in part (a).

2.7.2 Potential function

As mentioned, we shall use the potential method to analyze the performance of Fibonacci heap operations. For a given Fibonacci heap *H*, we indicate by *t (H)* the number of trees in the root list of *H* and by *m(H)* the number of marked nodes in *H*. The potential of Fibonacci heap *H* is then defined by

$$\Phi(H) = t(H) + 2\,m(H)\ .$$

For example, the potential of the Fibonacci heap shown in Figure 20.1 is 5+2 · 3 = 11. The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. We shall assume that a unit of potential can pay for a constant amount of work, where the constant is sufficiently large to cover the cost of any of the specific constant-time pieces of work that we might encounter.

2.7.3 Mergeable-heap operations

we describe and analyze the mergeable-heap operations as implemented for Fibonacci heaps. If only these operations—MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN, and UNION—are to be supported, each Fibonacci heap is simply a collection of "unordered" binomial trees. An unordered binomial tree is like a binomial tree, and it, too, is defined recursively. The unordered binomial tree U_0 consists of a single node, and an unordered binomial tree U_k consists

of two unordered binomial trees U_{k-1} for which the root of one is made into any child of the root of the other.

2.7.4 Creating a new Fibonacci heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H, where $n[H] = 0$ and $min[H] =$ NIL; there are no trees in H. Because $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is $\Phi(H) = 0$. The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost.

2.7.5 Inserting a node

The following procedure inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that key[x] has already been filled in.

```
FIB-HEAP-INSERT(H, x)
 1  degree[x] ← 0
 2  p[x] ← NIL
 3  child[x] ← NIL
 4  left[x] ← x
 5  right[x] ← x
 6  mark[x] ← FALSE
 7  concatenate the root list containing x with root list H
 8  if min[H] = NIL or key[x] < key[min[H]]
 9     then min[H] ← x
10  n[H] ← n[H] + 1
```

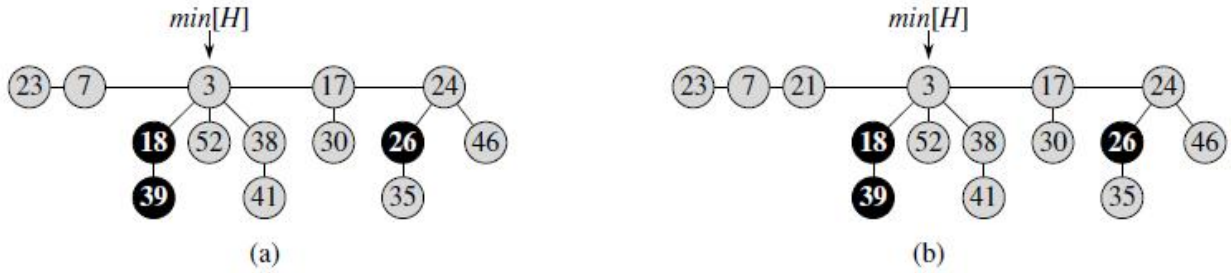


Figure: Inserting a node into a Fibonacci heap. **(a)** A Fibonacci heap *H*. **(b)** Fibonacci heap *H* after the node with key 21 has been inserted. The node becomes its own min-heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

Unlike the BINOMIAL-HEAP-INSERT procedure, FIB-HEAP-INSERT makes no attempt to consolidate the trees within the Fibonacci heap. If k consecutive FIBHEAP-INSERT operations occur, then k single-node trees are added to the root list.

To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1 .$$

Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$.

2.7.6 Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer min[H], so we can find the minimum node in O(1) actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its O(1) actual cost.

2.7.7 Uniting two Fibonacci heaps

The following procedure unites Fibonacci heaps H_1 and H_2, destroying H_1 and H_2 in the process. It simply concatenates the root lists of H_1 and H_2 and then determines the new minimum node.

```
FIB-HEAP-UNION(H1, H2)
1  H ← MAKE-FIB-HEAP()
2  min[H] ← min[H1]
3  concatenate the root list of H2 with the root list of H
4  if (min[H1] = NIL) or (min[H2] ≠ NIL and key[min[H2]] < key[min[H1]])
5     then min[H] ← min[H2]
6  n[H] ← n[H1] + n[H2]
7  free the objects H1 and H2
8  return H
```

The change in potential is

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\ &= 0 ,\end{aligned}$$

because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized cost of FIB-HEAP-UNION is therefore equal to its $O(1)$ actual cost.

2.7.8 Extracting the minimum node

The process of extracting the minimum node is the most complicated of the operations presented in this section. It is also where the delayed work of consolidating trees in the root list finally occurs. The following pseudocode extracts the minimum node. The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged. It also uses the auxiliary procedure CONSOLIDATE, which will be presented shortly.

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z ← min[H]
 2  if z ≠ NIL
 3     then for each child x of z
 4             do add x to the root list of H
 5                p[x] ← NIL
 6          remove z from the root list of H
 7          if z = right[z]
 8            then min[H] ← NIL
 9            else min[H] ← right[z]
10                 CONSOLIDATE(H)
11          n[H] ← n[H] − 1
12  return z
```

The next step, in which we reduce the number of trees in the Fibonacci heap, is ***consolidating*** the root list of H; this is performed by the call CONSOLIDATE(H). Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct *degree* value.

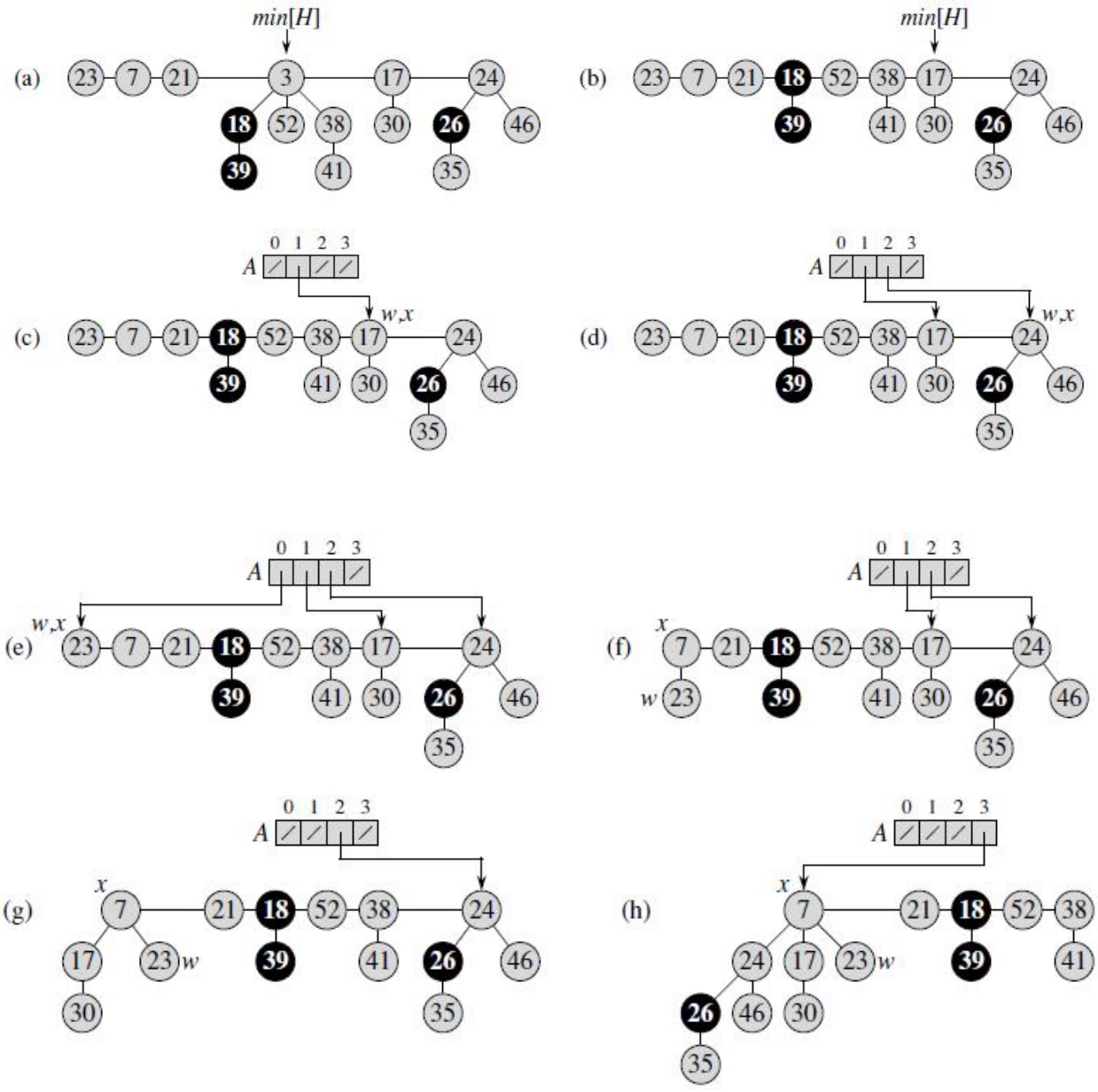
1. Find two roots x and y in the root list with the same degree, where $key[x] \leq key[y]$.
2. ***Link*** y to x: remove y from the root list, and make y a child of x. This operation is performed by the FIB-HEAP-LINK procedure. The field $degree[x]$ is incremented, and the mark on y, if any, is cleared.

The procedure CONSOLIDATE uses an auxiliary array $A[0..D(n[H])]$; if $A[i] = y$, then y is currently a root with $degree[y] = i$.

```
CONSOLIDATE(H)
 1  for i ← 0 to D(n[H])
 2       do A[i] ← NIL
 3  for each node w in the root list of H
 4       do x ← w
 5          d ← degree[x]
 6          while A[d] ≠ NIL
 7               do y ← A[d]        ▷ Another node with the same degree as x.
 8                  if key[x] > key[y]
 9                    then exchange x ↔ y
10                  FIB-HEAP-LINK(H, y, x)
11                  A[d] ← NIL
12                  d ← d + 1
13          A[d] ← x
14  min[H] ← NIL
15  for i ← 0 to D(n[H])
16       do if A[i] ≠ NIL
17             then add A[i] to the root list of H
18                  if min[H] = NIL or key[A[i]] < key[min[H]]
19                    then min[H] ← A[i]
```

FIB-HEAP-LINK(H, y, x)

1 remove y from the root list of H
2 make y a child of x, incrementing $degree[x]$
3 $mark[y] \leftarrow$ FALSE



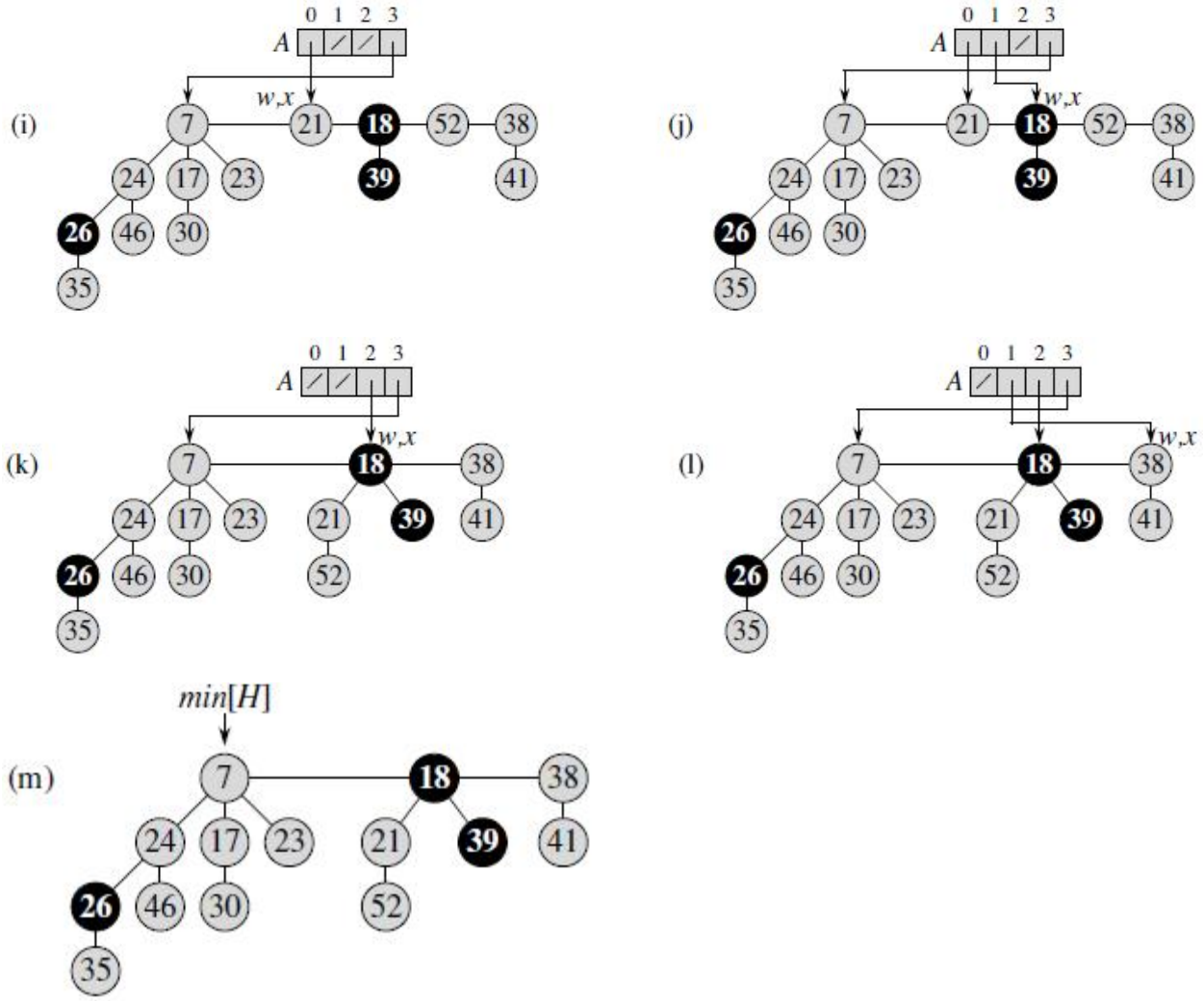


Figure: The action of FIB-HEAP-EXTRACT-MIN. **(a)** A Fibonacci heap H. **(b)** The situation after the minimum node z is removed from the root list and its children are added to the root list. **(c)–(e)** The array A and the trees after each of the first three iterations of the **for** loop of lines 3–13 of the procedure CONSOLIDATE. The root list is processed by starting at the node pointed to by $min[H]$ and following *right* pointers. Each part shows the values of w and x at the end of an iteration. **(f)–(h)** The next iteration of the **for** loop, with the values of w and x shown at the end of each iteration of the **while** loop of lines 6–12. Part (f) shows the situation after the first time through the **while** loop. The node with key 23 has been linked to the node with key 7, which is now pointed to by x. In part (g), the node with key 17 has been linked to the node with key 7, which is still pointed to by x. In part (h), the node with key 24 has been linked to the node with key 7. Since no node was previously pointed to by $A[3]$, at the end of the **for** loop iteration, $A[3]$ is set to point to the root of the resulting tree. **(i)–(l)** The situation after each of the next four iterations of the **for** loop. **(m)** Fibonacci heap H after reconstruction of the root list from the array A and determination of the new $min[H]$ pointer.

The potential before extracting the minimum node is $t(H) + 2m(H)$, and the potential afterward is at most $(D(n) + 1) + 2m(H)$, since at most $D(n) + 1$ roots remain and no nodes become marked during the operation. The amortized cost is thus at most

$$\begin{aligned} & O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ &= O(D(n)) + O(t(H)) - t(H) \\ &= O(D(n)) , \end{aligned}$$

since we can scale up the units of potential to dominate the constant hidden in $O(t(H))$. Intuitively, the cost of performing each link is paid for by the reduction in potential due to the link's reducing the number of roots by one. We shall see in Section 20.4 that $D(n) = O(\lg n)$, so that the amortized cost of extracting the minimum node is $O(\lg n)$.

2.7.9 Decreasing a key and deleting a node

In this section, we show how to decrease the key of a node in a Fibonacci heap in O(1) amortized time and how to delete any node from an n-node Fibonacci heap in O(D(n)) amortized time. These operations do not preserve the property that all trees in the Fibonacci heap are unordered binomial trees. They are close enough, however, that we can bound the maximum degree D(n) by O(lg n). Proving this bound, which we shall do in Section 20.4, will imply that FIB-HEAP-EXTRACTMIN and FIB-HEAP-DELETE run in O(lg n) amortized time.

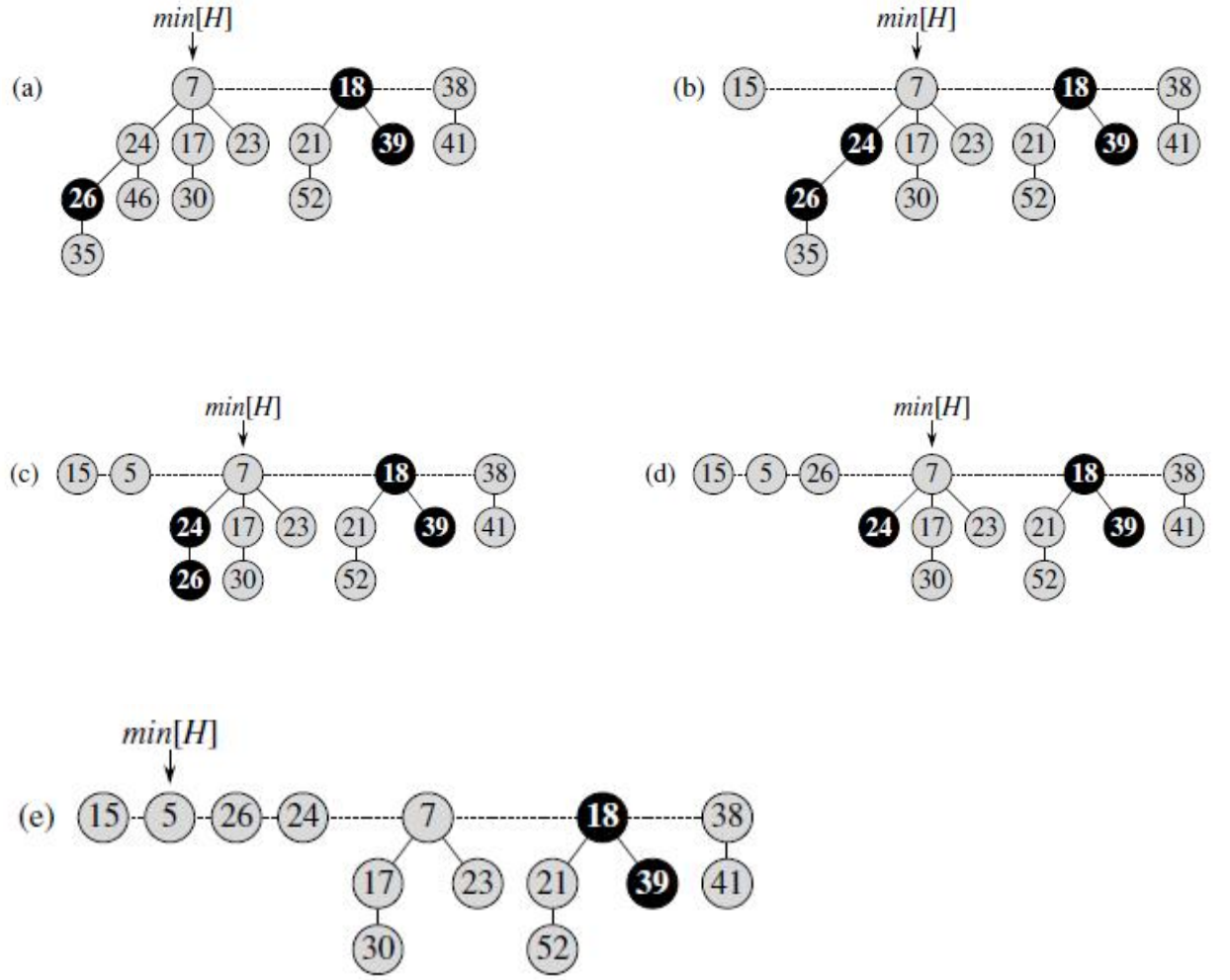
2.7.10 Decreasing a key

In the following pseudocode for the operation FIB-HEAP-DECREASE-KEY, we assume as before that removing a node from a linked list does not change any of the structural fields in the removed node.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > key[x]
2     then error "new key is greater than current key"
3  key[x] ← k
4  y ← p[x]
5  if y ≠ NIL and key[x] < key[y]
6     then CUT(H, x, y)
7          CASCADING-CUT(H, y)
8  if key[x] < key[min[H]]
9     then min[H] ← x
```

```
CUT(H, x, y)
1  remove x from the child list of y, decrementing degree[y]
2  add x to the root list of H
3  p[x] ← NIL
4  mark[x] ← FALSE
```

```
CASCADING-CUT(H, y)
1  z ← p[y]
2  if z ≠ NIL
3     then if mark[y] = FALSE
4             then mark[y] ← TRUE
5             else CUT(H, y, z)
6                  CASCADING-CUT(H, z)
```



Two calls of FIB-HEAP-DECREASE-KEY. **(a)** The initial Fibonacci heap. **(b)** The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had previously been unmarked, becomes marked. **(c)–(e)** The node with key 35 has its key decreased to 5. In part (c), the node, now with key 5, becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (d). Another cascading cut occurs, since the node with key 24 is marked as well. This node is cut from its parent and made an unmarked root in part (e). The cascading cuts stop at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading cuts would stop, since it is unmarked.) The result of the FIB-HEAP-DECREASE-KEY operation is shown in part (e), with min[H] pointing to the new minimum node.

We shall now show that the amortized cost of FIB-HEAP-DECREASE-KEY is only O(1). We start by determining its actual cost. The FIB-HEAP-DECREASEKEY procedure takes O(1) time, plus the time to perform the cascading cuts. Suppose that CASCADING-CUT is recursively called c times from a given invocation of FIB-HEAP-DECREASE-KEY. Each call

of CASCADING-CUT takes O(1) time exclusive of recursive calls. Thus, the actual cost of FIB-HEAP-DECREASE-KEY, including all recursive calls, is O(c).

We next compute the change in potential. Let H denote the Fibonacci heap just prior to the FIB-HEAP-DECREASE-KEY operation. Each recursive call of CASCADING-CUT, except for the last one, cuts a marked node and clears the mark bit. Afterward, there are t (H)+c trees (the original t (H) trees, c−1 trees produced by cascading cuts, and the tree rooted at x) and at most m(H)−c+2 marked nodes (c−1 were unmarked by cascading cuts and the last call of CASCADING-CUT may have marked a node). The change in potential is therefore at most

$$((t(H)+c)+2(m(H)-c+2))-(t(H)+2m(H))=4-c\ .$$

Thus, the amortized cost of FIB-HEAP-DECREASE-KEY is at most

$$O(c)+4-c=O(1)\ ,$$

since we can scale up the units of potential to dominate the constant hidden in O(c).

You can now see why the potential function was defined to include a term that is twice the number of marked nodes. When a marked node y is cut by a cascading cut, its mark bit is cleared, so the potential is reduced by 2. One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node y becoming a root.

2.7.11 Deleting a node

It is easy to delete a node from an n-node Fibonacci heap in O(D(n)) amortized time, as is done by the following pseudocode. We assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, −∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE is analogous to BINOMIAL-HEAP-DELETE. It makes x become the minimum node in the Fibonacci heap by giving it a uniquely small key of $-\infty$. Node x is then removed from the Fibonacci heap by the FIB-HEAPEXTRACT-MIN procedure. The amortized time of FIB-HEAP-DELETE is the sum of the O(1) amortized time of FIB-HEAP-DECREASE-KEY and the O(D(n)) amortized time of FIB-HEAP-EXTRACT-MIN. Since we shall see above that D(n) = O(lg n), the amortized time of FIB-HEAP-DELETE is O (lg n).

NOTES (UNIT-III) Divide and Conquer, Greedy Methods

3.1 Divide and Conquer

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

3.1.1 Strassen's algorithm for matrix multiplication

Strassen's algorithm can be viewed as an application of a familiar design technique: divide and conquer. Suppose we wish to compute the product $C = AB$, where each of A, B, and C are $n \times n$ matrices. Assuming that n is an exact power of 2, we divide each of A, B, and C into four $n/2 \times n/2$ matrices, rewriting the equation $C = AB$ as follows:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} . \qquad (1)$$

The above equation corresponds to the four equations

r=ae+bg, s= af+bh, t=ce+dg, u=cf+dh (2)

Each of these four equations specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their $n/2 \times n/2$ products. Using these equations to define a straightforward divide and-conquer strategy, we derive the following recurrence for the time $T(n)$ to multiply two $n \times n$ matrices:

$T(n)= 8T(n/2)+\Theta(n^2)$

$= \Theta(n^{\log 8})$

$= \Theta(n^3)$

Strassen discovered a different recursive approach that requires only 7 recursive multiplications of $n/2 \times n/2$ matrices and $\Theta(n2)$ scalar additions and subtractions, yielding the recurrence.

$T(n)= 7T(n/2)+\Theta(n^2)$

$= \Theta(n^{\log 7})$

$= \Theta(n^{2.81})$

Strassen's method has four steps:

1. Divide the input matrices A and B into $n/2 \times n/2$ submatrices,
2. Using $\Theta(n2)$ scalar additions and subtractions, compute 14 matrices $A1$, $B1$, $A2$, $B2$, . . . , $A7$, $B7$, each of which is $n/2 \times n/2$.
3. Recursively compute the seven matrix products $Pi = Ai\ Bi$ for $i = 1, 2, \ldots, 7$.
4. Compute the desired submatrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the Pi matrices, using only $\Theta(n2)$ scalar additions and subtractions.

3.1.1.1 Determining the submatrix products

It is not clear exactly how Strassen discovered the submatrix products that are the key to making his algorithm work.

Let us guess that each matrix product *Pi* can be written in the form

$$\begin{aligned} P_i &= A_i B_i \\ &= (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d) \cdot (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h) , \end{aligned}$$

where the coefficients αij, βij are all drawn from the set {-1, 0, 1}.

We can rewrite the equation (2) as

$$\begin{aligned} r &= ae + bg \\ &= \begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} \\ &= \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array} . \end{aligned}$$

The last expression uses an abbreviated notation in which "+" represents +1, "·" represents 0, and "-" represents -1. Using these notations the all expressions of equation (2) becomes

$$\begin{aligned} s &= af + bh \\ &= \begin{pmatrix} \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} , \end{aligned}$$

$$\begin{aligned} t &= ce + dg \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \end{pmatrix} , \end{aligned}$$

$$\begin{aligned} u &= cf + dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix} . \end{aligned}$$

The submatrics s can be computed as s=P1+P2 where P1 and P2 computed as follows:

$$
\begin{aligned}
P_1 &= A_1 B_1 \\
&= a \cdot (f - h) \\
&= af - ah \\
&= \begin{pmatrix} \cdot & + & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.
\end{aligned}
\qquad
\begin{aligned}
P_2 &= A_2 B_2 \\
&= (a + b) \cdot h \\
&= ah + bh \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.
\end{aligned}
$$

The matrix t can be computed in a similar manner as $t = P3 + P4$, where

$$
\begin{aligned}
P_3 &= A_3 B_3 \\
&= (c + d) \cdot e \\
&= ce + de \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}
\end{aligned}
\qquad
\begin{aligned}
P_4 &= A_4 B_4 \\
&= d \cdot (g - e) \\
&= dg - de \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & + & \cdot \end{pmatrix}.
\end{aligned}
$$

Then P5 is calculated as

$P5 = A5B5 = (a + d) \cdot (e + h)$

$= ae + ah + de + dh$

$$
= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}.
$$

We can use $P4$ and $P2$ to cancel them, but two other inessential terms then appear:

$$
\begin{aligned}
P_5 + P_4 - P_2 &= ae + dh + dg - bh \\
&= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \end{pmatrix}.
\end{aligned}
$$

By adding an additional product

$$
\begin{aligned}
P_6 &= A_6 B_6 \\
&= (b - d) \cdot (g + h) \\
&= bg + bh - dg - dh \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & - & - \end{pmatrix},
\end{aligned}
$$

however, we obtain

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 \\
&= ae + bg \\
&= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.
\end{aligned}
$$

We can obtain u in a similar manner from $P5$ by using $P1$ and $P3$ to move the inessential termsof $P5$ in a different direction:

$$
\begin{aligned}
P_5 + P_1 - P_3 &= ae + af - ce + dh \\
&= \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.
\end{aligned}
$$

By subtracting an additional product

$$
\begin{aligned}
P_7 &= A_7 B_7 \\
&= (a - c) \cdot (e + f) \\
&= ae + af - ce - cf \\
&= \begin{pmatrix} + & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & - & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},
\end{aligned}
$$

we now obtain

$$
\begin{aligned}
u &= P_5 + P_1 - P_3 - P_7 \\
&= cf + dh \\
&= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.
\end{aligned}
$$

The 7 submatrix products $P1$, $P2$, . . . , $P7$ can thus be used to compute the product $C = AB$, which completes the description of Strassen's method.

3.1.2 Finding the convex hull

The ***convex hull*** of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. We denote the convex hull of Q by CH(Q).

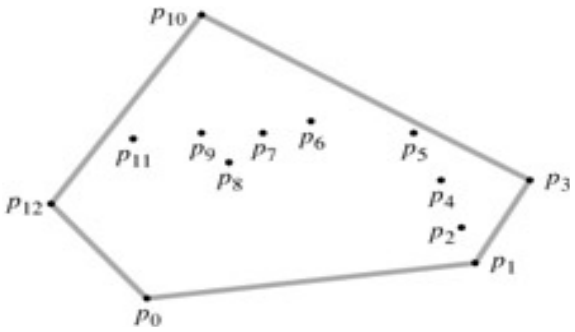


Figure : A set of points $Q = \{p0, p1, ..., p12\}$ with its convex hull CH(Q) in gray.

Here two algorithm Graham's scan and Jarvis march that computes convex hull of a set of n points in counterclockwise order. Several methods that compute convex hulls include the following.

1. In the ***incremental method***, can be implemented to take a total of $O(n \lg n)$ time.
2. In the ***divide-and-conquer method***, runs in $O(n \lg n)$ time.
3. The ***prune-and-search method*** is similar to the worst-case linear-time median algorithm runs in only $O(n \lg h)$ time.

3.1.2.1 Graham's scan

Graham's scan solves the convex-hull problem by maintaining a stack S of candidate points. Each point of the input set Q is pushed once onto the stack, and the points that are not vertices of CH(Q) are eventually popped from the stack. When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary.

GRAHAM-SCAN(*Q*)

1 let $p0$ be the point in Q with the minimum y-coordinate, or the leftmost such point in case of a tie

2 let $\_p1, p2, ..., pm\_$ be the remaining points in Q, sorted by polar angle in counterclockwise order around $p0$ (if more than one point has the same angle, remove all but the one that is farthest from $p0$)

3 PUSH($p0$, S)

4 PUSH($p1$, S)

5 PUSH($p2$, S)

6 **for** $i \leftarrow 3$ **to** m

7 **do while** the angle formed by points NEXT-TO-TOP(S), TOP(S), and pi makes a nonleft turn

8 **do** POP(S)

9 PUSH(pi, S)

10 **return S**

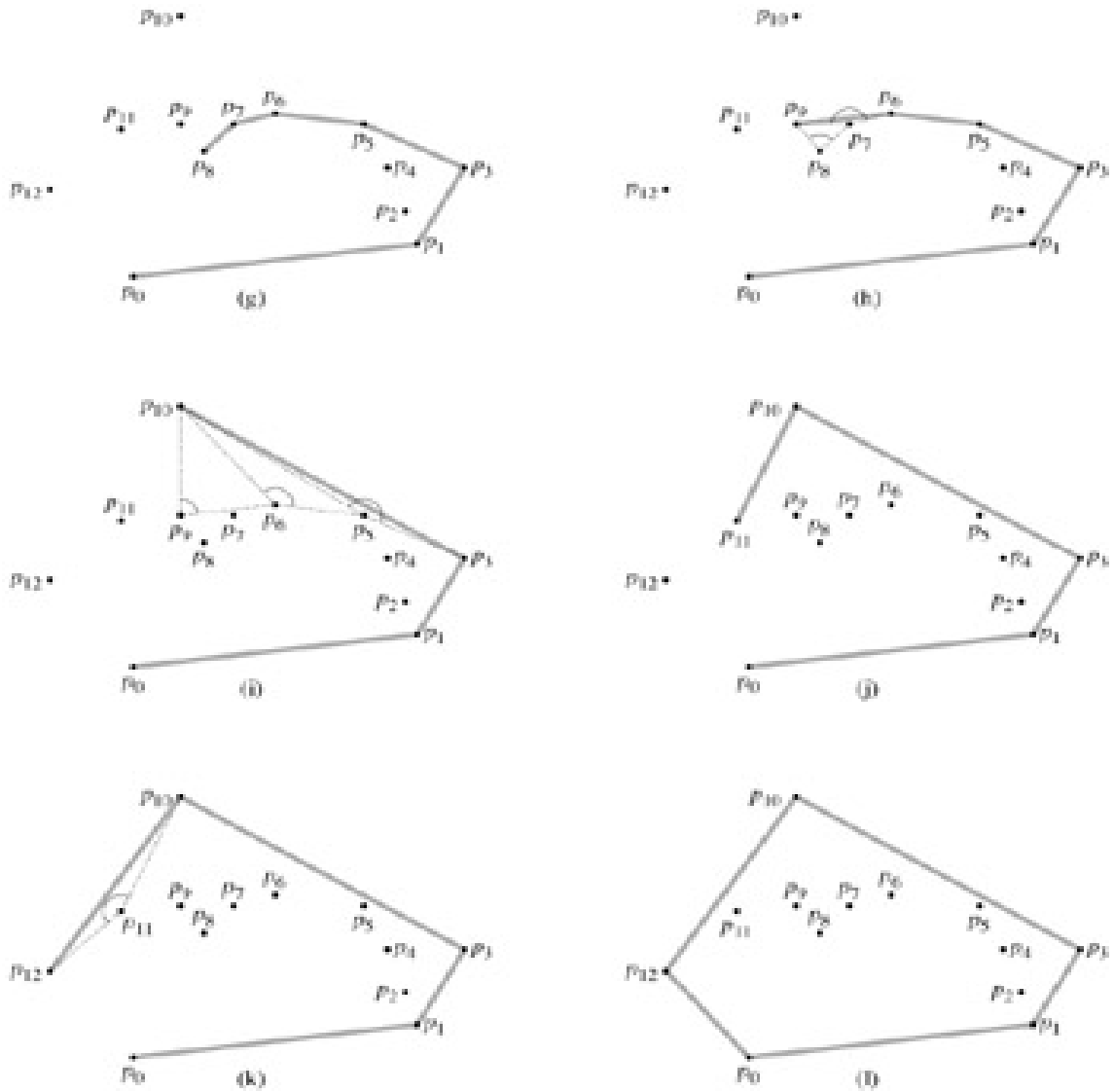


Figure : The execution of GRAHAM-SCAN on the set Q of Figure 1. The current convex hull contained in stack S is shown in gray at each step. *(a)* The sequence $\{p1, p2, ..., p12\}$ of points numbered in order of increasing polar angle relative to $p0$, and the initial stack S containing $p0$, $p1$, and $p2$. *(b)-(k)* Stack S after each iteration of the *for* loop of lines 6-9. Dashed lines show nonleft turns, which cause points to be popped from the stack. In part (h), for example, the right turn at angle $\angle p7p8p9$ causes $p8$ to be popped, and then the right turn at angle $\angle p6p7p9$ causes $p7$ to be popped. *(l)* The convex hull returned by the procedure.

Complexity

The complexity of Graham's Scan is O(nlogn) because the line 2 use sorting of points by their polar angle.

3.1.2.2 Jarvis's march

Jarvis's march computes the convex hull of a set Q of points by a technique known as package wrapping (or gift wrapping). The algorithm runs in time O(nh), where h is the number of vertices of CH(Q). When h is o(lg n), Jarvis's march is asymptotically faster than Graham's scan.

Jarvis's march builds a sequence H = {p0, p1, ..., ph-1} of the vertices of CH(Q). We start with p0. As Figure shows, the next convex hull vertex p1 has the smallest polar angle with respect to p0. (In case of ties, we choose the point farthest from p0.) Similarly, p2 has the smallest polar angle with respect to p1, and so on. When we reach the highest vertex, say pk (breaking ties by choosing the farthest such vertex), we have constructed, as Figure shows, the right chain of CH(Q). To construct the left chain, we start at pk and choose pk+1 as the point with the smallest polar angle with respect to pk, but from the negative x-axis. We continue on, forming the left chain by taking polar angles from the negative x-axis, until we come back to our original vertex p0.

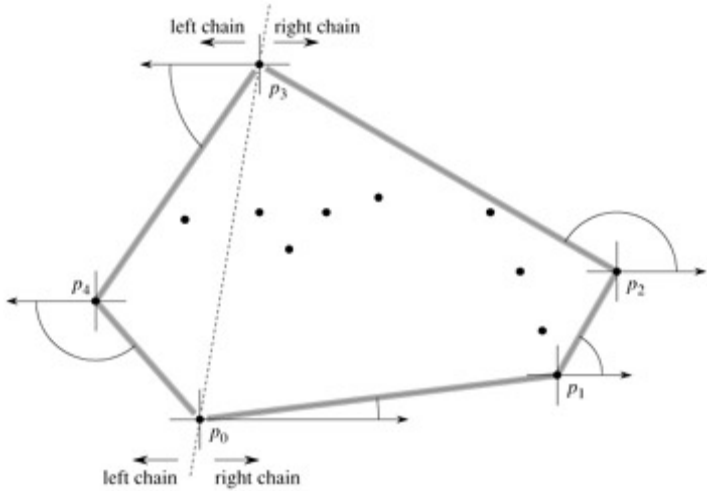


Figure: The operation of Jarvis's march. The first vertex chosen is the lowest point p0. The next vertex, p1, has the smallest polar angle of any point with respect to p0. Then, p2 has the smallest polar angle with respect to p1. The right chain goes as high as the highest point p3. Then, the left chain is constructed by finding smallest polar angles with respect to the negative x-axis.

complexity

If implemented properly, Jarvis's march has a running time of $O(nh)$.

3.1.3.Binary search

A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

Algorithm:

```
int binary_search(int A[], int key, int imin, int imax)
{
 // test if array is empty
 if (imax < imin)
   // set is empty, so return value showing not found
   return KEY_NOT_FOUND;
 else
   {
    // calculate midpoint to cut set in half
    int imid = midpoint(imin, imax);

    // three-way comparison
    if (A[imid] > key)
      // key is in lower subset
      return binary_search(A, key, imin, imid-1);
    else if (A[imid] < key)
      // key is in upper subset
      return binary_search(A, key, imid+1, imax);
    else
      // key has been found
      return imid;
   }
}
```

It is invoked with initial `imin` and `imax` values of `0` and `N-1` for a zero based array of length N. The number type "int" shown in the code has an influence on how the midpoint calculation can be implemented correctly. With unlimited numbers, the midpoint can be calculated as `"(imin + imax) / 2"`. In practical programming, however, the calculation is often performed with numbers of a limited range, and then the intermediate result `"(imin + imax)"` might overflow. With limited numbers, the midpoint can be calculated correctly as `"imin + ((imax - imin) / 2)"`.

Example: The list to be searched: L = 1 3 4 6 8 9 11. The value to be found: X = 4.

Compare X to 6. X is smaller. Repeat with L = 1 3 4.

Compare X to 3. X is bigger. Repeat with L = 4.

Compare X to 4. They are equal. We're done, we found X.

3.2 Greedy Methods

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. The greedy method is quite powerful and works well for a wide range of problems.

3.2.1 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. For each decision point in the algorithm, the choice that seems best at the moment is chosen. This heuristic strategy does not always produce an optimal solution. We went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
4. Show that all but one of the subproblems induced by having made the greedy choice are empty.
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

More generally, we design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

3.2.2 Greedy-choice property

The first key ingredient is the greedy-choice property: a globally optimal solution can be arrived at by making a locally optimal (greedy) choice. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems.

3.2.3 Optimal substructure

A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms.

3.2.4 An activity-selection problem

Suppose we have a set $S = \{a1, a2, ..., an\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity ai has a

start time si and a finish time fi, where $0 \le si < fi < \infty$. Activities ai and aj are compatible if the intervals [si, fi) and [sj fj) do not overlap (i.e., ai and aj are compatible if $si \ge fj$ or $sj \ge fi$). The activity-selection problem is to select a maximum-size subset of mutually compatible activities.

Let us assume that the activities are sorted in monotonically increasing order of finish time:

$$f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_{n+1} .$$

3.2.4.1 A recursive greedy algorithm

A recursive greedy solution as the procedure RECURSIVE-ACTIVITY-SELECTOR is defined. It takes the start and finish times of the activities, represented as arrays *s* and *f*, as well as the starting indices *i* and *j* of the subproblem *Si,j* it is to solve. It returns a maximum-size set of mutually compatible activities in *Si,j*. We assume that the *n* input activities are ordered by monotonically increasing finish time. The initial call is RECURSIVE-ACTIVITY-SELECTOR(*s, f*, 0, *n* + 1).

RECURSIVE-ACTIVITY-SELECTOR(*s, f, i, j*)

1 $m \leftarrow i + 1$

2 while $m < j$ and $sm < fi$ // Find the first activity in *Sij*.

3 do $m \leftarrow m + 1$

4 if $m < j$

5 then return $\{am\}$U RECURSIVE-ACTIVITY-SELECTOR(*s, f, m, j*)

6 else return Ø

For example, consider the following set S of activities, which we have sorted in monotonically increasing order of finish time:

i 1 2 3 4 5 6 7 8 9 10 11

si 1 3 0 5 3 5 6 8 8 2 12

fi 4 5 6 7 8 9 10 11 12 13 14

the implementation is given by figure below

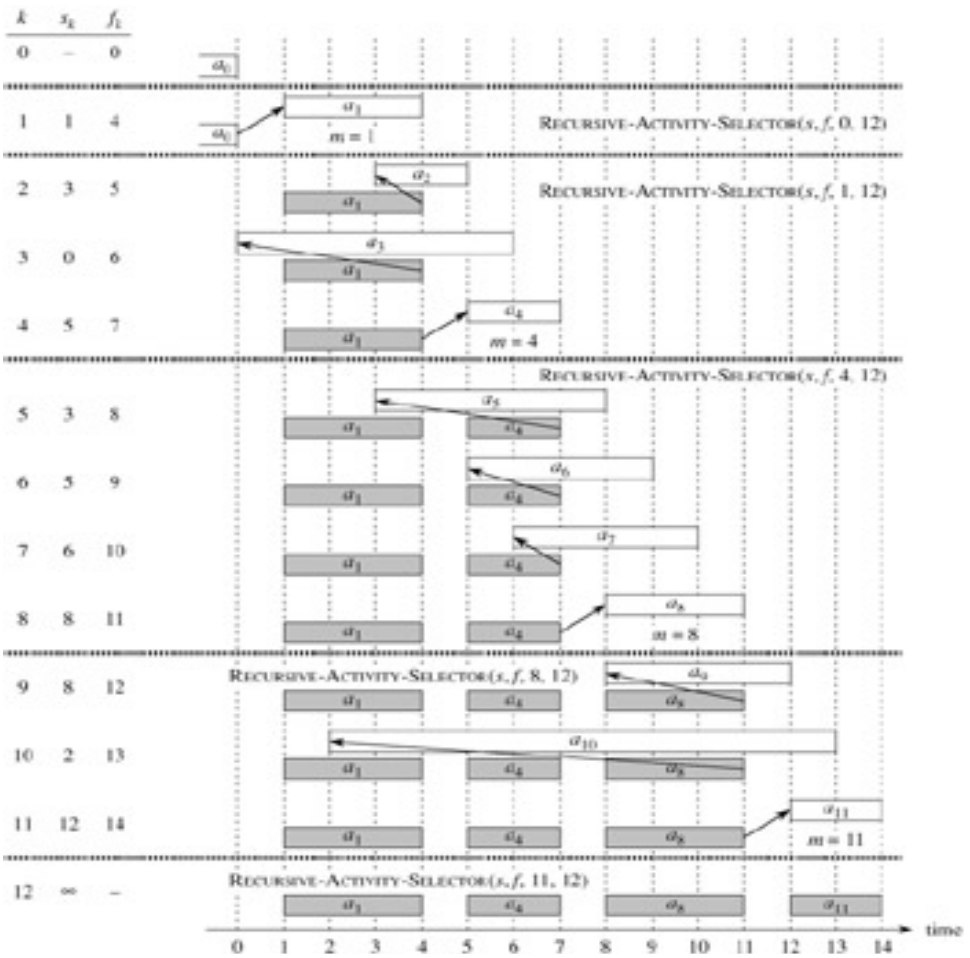


Figure: The operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities given earlier. Activities considered in each recursive call appear between horizontal lines. The fictitious activity $a0$ finishes at time 0, and in the initial call, RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 12), activity $a1$ is selected. In each recursive call, the activities that have already been selected are shaded, and the activity shown in white is being considered. If the starting time of an activity occurs before the finish time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call, RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 12), returns Ø. The resulting set of selected activities is $\{a1, a4, a8, a11\}$.

Complexity

Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n + 1) is $\Theta(n)$.

1.2.4.2 An iterative greedy algorithm

The RECURSIVEACTIVITY-SELECTOR works for any subproblem *Sij*, but we have seen that we need to consider only subproblems for which $j = n + 1$, i.e., subproblems that consist of the last activities to finish.

The procedure GREEDY-ACTIVITY-SELECTOR is an iterative version of the procedure RECURSIVE-ACTIVITY-SELECTOR. It also assumes that the input activities are ordered by monotonically increasing finish time. It collects selected activities into a set *A* and returns this set when it is done.

GREEDY-ACTIVITY-SELECTOR(*s, f*)

1 $n \leftarrow length[s]$

2 $A \leftarrow \{a1\}$

3 $i \leftarrow 1$

4 **for** m ← 2 to *n*

5 **do if** $sm \geq fi$

6 **then** A ← $A \ U \ \{am\}$

7 $i \leftarrow m$

8 **return** *A*

The set A returned by the call GREEDY-ACTIVITY-SELECTOR(s, f) is precisely the set returned by the call RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, n + 1). Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in Θ(n) time, assuming that the activities were already sorted initially by their finish times.

3.2.5 Task scheduling problem

A unit-time task is a job, such as a program to be run on a computer, that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a schedule for S is a permutation of S specifying the order in which these tasks are to be performed. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2, and so on. The problem of scheduling unit-time tasks with deadlines and penalties for a single processor has the following inputs:

a set S = {a1, a2, ..., an} of n unit-time tasks;

a set of n integer deadlines d1, d2, ..., dn, such that each di satisfies $1 \leq di \leq$ nand task ai is supposed to finish by time di; and

a set of n nonnegative weights or penalties w1, w2, ..., wn, such that we incur a penalty of wi if task ai is not finished by time di and we incur no penalty if a task finishes by its deadline.

We are asked to find a schedule for S that minimizes the total penalty incurred for missed deadlines.

Example:

The figure below gives an example of a problem of scheduling unit-time tasks with deadlines and penalties for a single processor. In this example, the greedy algorithm selects tasks *a*1, *a*2, *a*3, and *a*4, then rejects *a*5 and *a*6, and finally accepts *a*7. The final optimal schedule is

S= {*a*2, *a*4, *a*1, *a*3, *a*7, *a*5, *a*6},

which has a total penalty incurred of $w5 + w6 = 50$.

	Task						
ai	**1**	**2**	**3**	**4**	**5**	**6**	**7**
di	4	2	4	3	1	4	6
wi	70	60	50	40	30	20	10

Figure: The problem with deadlines and penalties.

3.2.6 The Fractional Knapsack Problem

A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

The 0-1 knapsack problem:

1. The thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
2. Carrying at most W pounds, maximize value
    - Note: assume v_i, w_i, and W are all integers
    - "0-1" b/c each item must be taken or left in entirety
3. The knapsack problem solved by Dynamic programming.
4.

The fractional knapsack problem:

1. Thief can take fractions of items
2. Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust
3. The problem will be solved by using greedy algorithm.

There are n items in a store. For i =1,2, . . . , n, item i has weight $w_i > 0$ and worth $v_i > 0$. Thief can carry a maximum weight of W pounds in a knapsack. In this version of a problem the items can be broken into smaller piece, so the thief may decide to carry only a fraction x_i of object i, where $0 \le x_i \le 1$. Item i contributes $x_i w_i$ to the total weight in the knapsack, and $x_i v_i$ to the value of the load.

In Symbol, the fraction knapsack problem can be stated as follows. maximize $^{n}\mathbf{S}_{i=1}\, x_i v_i$ subject to constraint $^{n}\mathbf{S}_{i=1}\, x_i w_i \le W$

It is clear that an optimal solution must fill the knapsack exactly, for otherwise we could add a fraction of one of the remaining objects and increase the value of the load. Thus in an optimal solution $^{n}\mathbf{S}_{i=1}\, x_i w_i = W$.

Greedy-fractional-knapsack (*w, v, W*)

```
FOR i =1 to n
   do x[i] =0
weight = 0
while weight < W
   do i = best remaining item
      IF weight + w[i] ≤ W
         then x[i] = 1
            weight = weight + w[i]
         else
            x[i] = (w - weight) / w[i]
            weight = W
return x
```

Analysis

If the items are already sorted into decreasing order of v_i / w_i, then the while-loop takes a time in *O(n)*; Therefore, the total time including the sort is in *O(n log n)*.

Example. Consider 5 items along their respective weights and values

I = <I1, I2, I3, I4, I5>

w =<5, 10, 20, 30, 40>

v = <30, 20, 100, 90, 160>

The capacity of knapsack W = 60.

The maximum profit will found by using fractional knapsack approach.

Initially

Item	**Weight(w_i)**	**Value(v_i)**
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value/weight ratio $p_i=v_i/w_i$

Item	Weight (w_i)	Value(v_i)	$p_i=v_i/w_i$
I_1	5	30	6
I_2	10	20	2
I_3	20	100	5
I_4	30	90	3
I_5	40	160	4

Now the arrange the value of p_i in decreasing order

Item	Weight (w_i)	Value(v_i)	$p_i=v_i/w_i$
I_1	5	30	6
I_3	20	100	5
I_5	40	160	4
I_4	30	90	3
I_2	10	20	2

Now fill the knapsack according to its capacity. Item I_1 is filled firstly and then item I_3 filled. Now the filled weight is 5+20=25. The remaining capacity is 60-25=35. Then fill the fraction of item I_5 in the knapsack. The amount of I_5 filled in knapsack is 35.

Now we calculate the maximum profit as given below

Maximum profit = 30+100+35*4=130+140=270

So the knapsack grabs the 270 profit.

3.2..7 Minimum spanning tree

Given a connected, undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components.

3.2.7.1 Growing a minimum spanning tree:

Assume that we have a connected, undirected graph $G = (V, E)$ with a weight function $w : E \rightarrow \mathbf{R}$, and we wish to find a minimum spanning tree for G. This greedy strategy is captured by the

following "generic" algorithm, which grows the minimum spanning tree one edge at a time. The algorithm manages a set of edges A, maintaining the following loop invariant:

• Prior to each iteration, A is a subset of some minimum spanning tree.

At each step, we determine an edge (u, v) that can be added to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree. We call such an edge a ***safe edge*** for A, since it can be safely added to A while maintaining the invariant.

GENERIC-MST(*G*, *w*)

1 $A \leftarrow \emptyset$

2 **while** A does not form a spanning tree

3 **do** find an edge (u, v) that is safe for A

4 $A \leftarrow A \cup \{(u, v)\}$

5 **return** A

3.7.1.3 Kruskal's algorithm

Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* (a minimum spanning tree for each connected component).

Description:

- create a forest F (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph
- while S is nonempty and F is not yet spanning
- remove an edge with minimum weight from S
- if that edge connects two different trees, then add it to the forest, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

MST-KRUSKAL(*G*, *w*)

1 $A \leftarrow \emptyset$

2 **for** each vertex $v \_ V[G]$

3 **do** MAKE-SET(v)

4 sort the edges of E into nondecreasing order by weight w

5 **for** each edge $(u, v) \_ E$, taken in nondecreasing order by weight

6 **do if** FIND-SET(u) $\neq$ FIND-SET(v)

7 **then** $A \leftarrow A \_ \{(u, v)\}$

8 UNION(u, v)

9 **return** A

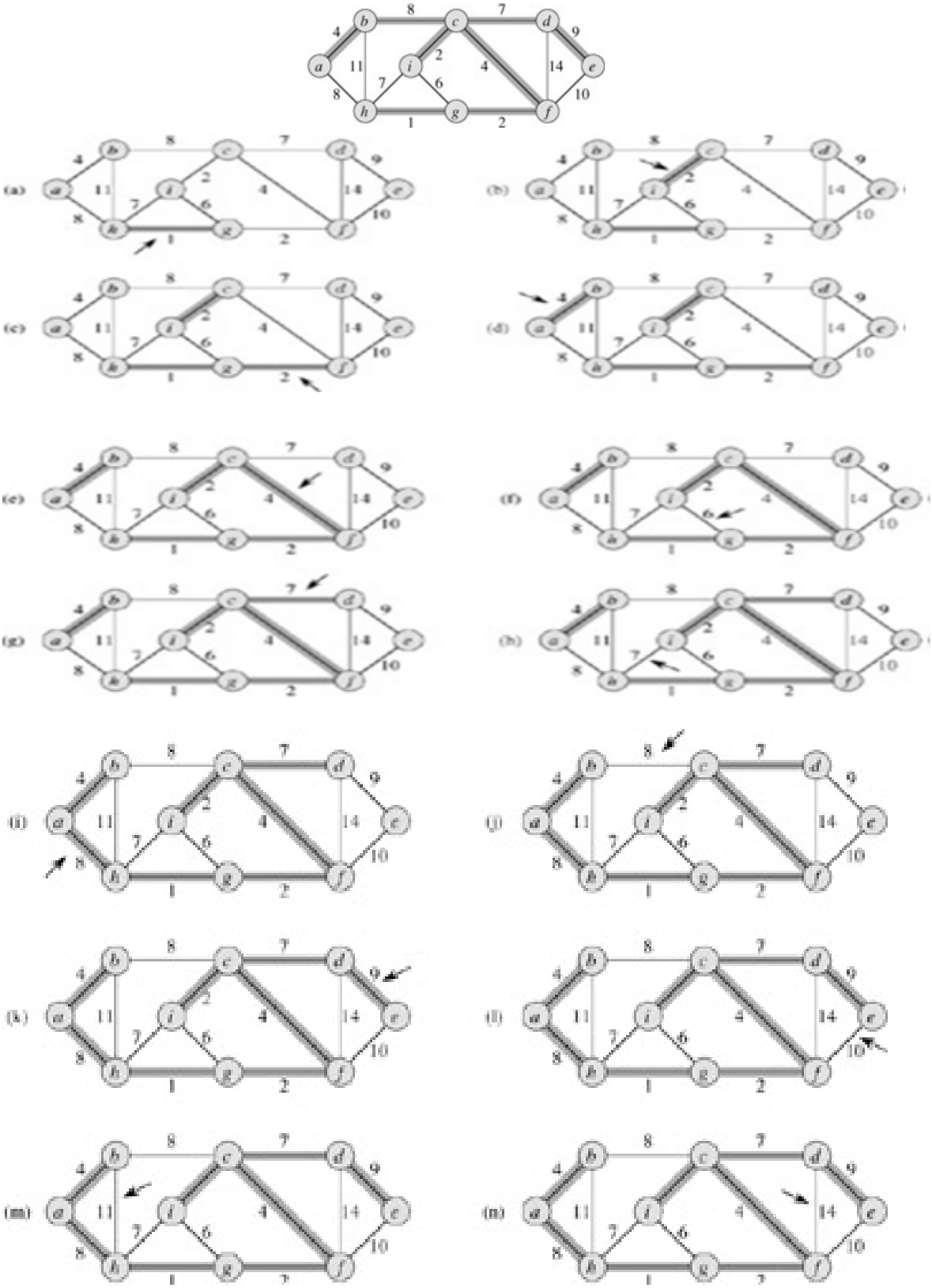


Figure: The execution of Kruskal's algorithm on the graph. Shaded edges belong to the forest A being grown. The edges are considered by the algorithm in sorted order by weight. An arrow

points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

Running time

- E is at most V^2 and $\log V^2 = 2 \log V$ is $O(\log V)$.
- Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain $V \leq E+1$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time; this allows the step "remove an edge with minimum weight from S" to operate in constant time. Next, we use a disjoint-set data structure (Union & Find) to keep track of which vertices are in which components. We need to perform O(E) operations, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform O(E) operations in $O(E \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$.

3.7.1.3 Prim's Algorithm

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Prim's algorithm is a special case of the generic minimum-spanning tree algorithm. Prim's algorithm has the property that the edges in the set A always form a single tree. the tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. At each step, a light edge is added to the tree A that connects A to this rule adds only edges that are safe for A; therefore, when the algorithm terminates, the edges in A form a minimum spanning tree.an isolated vertex of $GA = (V, A)$.

MST-PRIM(G, w, r)

1 **for** each $u \in V[G]$
2 **do** $key[u] \leftarrow \infty$
3 $\pi[u] \leftarrow$ NIL
4 $key[r] \leftarrow 0$
5 $Q \leftarrow V[G]$
6 **while** $Q \neq \emptyset$
7 **do** $u \leftarrow$ EXTRACT-MIN(Q)
8 **for** each $v \in Adj[u]$
9 **do if** $v \in Q$ and $w(u, v) < key[v]$
10 **then** $\pi[v] \leftarrow u$
11 $key[v] \leftarrow w(u, v)$

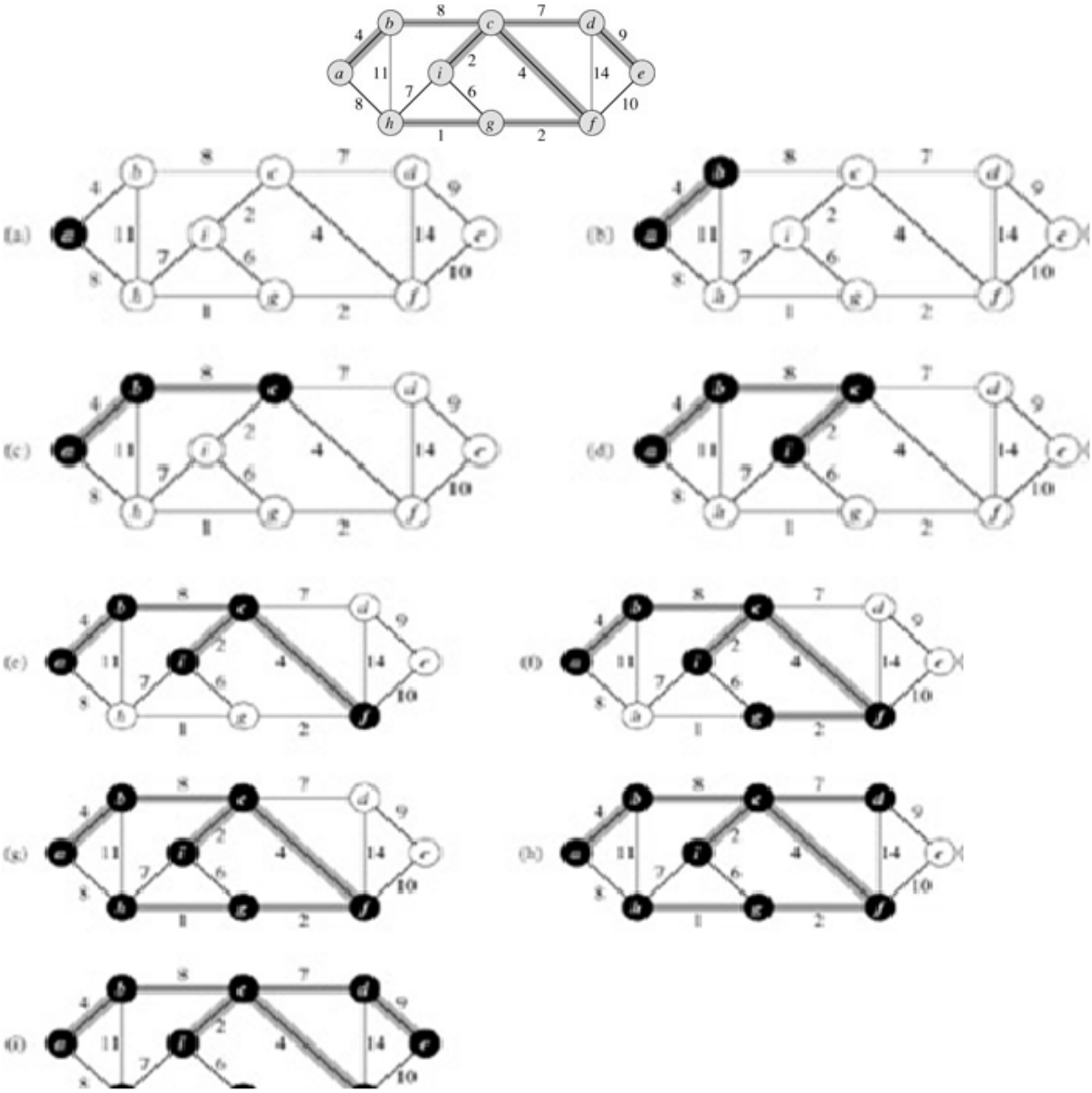


Figure: The execution of Prim's algorithm on the graph from Figure. The root vertex is *a*. Shaded edges are in the tree being grown, and the vertices in the tree are shown in black. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. In the second step, for example, the algorithm has a choice of adding either edge (*b, c*) or edge (*a, h*) to the tree since both are light edges crossing the cut.

Complexity:

By using min priority queue the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

3.2.8 Single source shortest path

In a ***shortest-paths problem***, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow \mathbf{R}$ mapping edges to real-valued-weights. The ***weight*** of path $p = \{v0, v1, ..., vk\}$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

We define the **shortest-path weight** from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v , \\ \infty & \text{otherwise} . \end{cases}$$

Many other problems can be solved by the algorithm for the single-source problem, including the following variants.

Single-destination shortest-paths problem

Single-pair shortest-path problem

All-pairs shortest-paths problem

In some instances of the single-source shortest-paths problem, there may be edges whose weights are negative.

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbf{R}$, and assume that G contains no negative-weight cycles reachable from the source vertex $s \in V$, so that shortest paths are well defined. A **shortest-paths tree** rooted at s is a directed subgraph $G' = (V', E')$, where $V' \in V$ and $E' \in E$, such that

1. V' is the set of vertices reachable from s in G,
2. G' forms a rooted tree with root s, and
3. For all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G.

3.2.8.1 Relaxation

We initialize the shortest-path estimates and predecessors by the following $\Theta(V)$-time procedure.

INITIALIZE-SINGLE-SOURCE(*G*, *s*)

1 **for** each vertex $v \in V[G]$

2 **do** $d[v] \leftarrow \infty$

3 $\pi[v] \leftarrow$ NIL

4 $d[s] = 0$

After initialization, $\pi[v] =$ NIL for all $v \in V$, $d[s] = 0$, and $d[v] = \infty$ for $v \in V - \{s\}$.

The process of ***relaxing***] an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $d[v]$ and $\pi[v]$.

RELAX (*u*, *v*, *w*)

1 **if** $d[v] > d[u] + w(u, v)$

2 **then** $d[v] \leftarrow d[u] + w(u, v)$

3 $\pi[v] \leftarrow u$

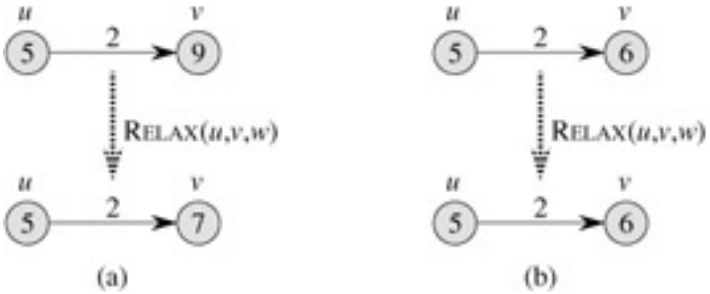


Figure: Relaxation of an edge (*u, v*) with weight *w* (*u, v*) = 2. The shortest-path estimate of each vertex is shown within the vertex. (a) Because $d[v] > d[u] + w(u, v)$ prior to relaxation, the value of $d[v]$ decreases. (b) Here, $d[v] \leq d[u] + w(u, v)$ before the relaxation step, and so $d[v]$ is unchanged by relaxation.

3.2.8.2 The Bellman-Ford algorithm

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the generalcase in which edge weights may be negative. Given a weighted, directed graph G = (V, E) with source s and weight function w : E → R, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

BELLMAN-FORD(*G*, *w*, *s*)

1 INITIALIZE-SINGLE-SOURCE(*G*, *s*)

2 for $i \leftarrow 1$ to $|V[G]| - 1$

3 do for each edge (u, v) _ $E[G]$

4 do RELAX(*u*, *v*, *w*)

5 for each edge (u, v) _ $E[G]$

6 do if $d[v] > d[u] + w(u, v)$

7 then return FALSE

8 return TRUE

Complexity:

The Bellman-Ford algorithm runs in time $O(V E)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2-4 takes $\Theta(E)$ time, and the **for** loop of lines 5-7 takes $O(E)$ time.

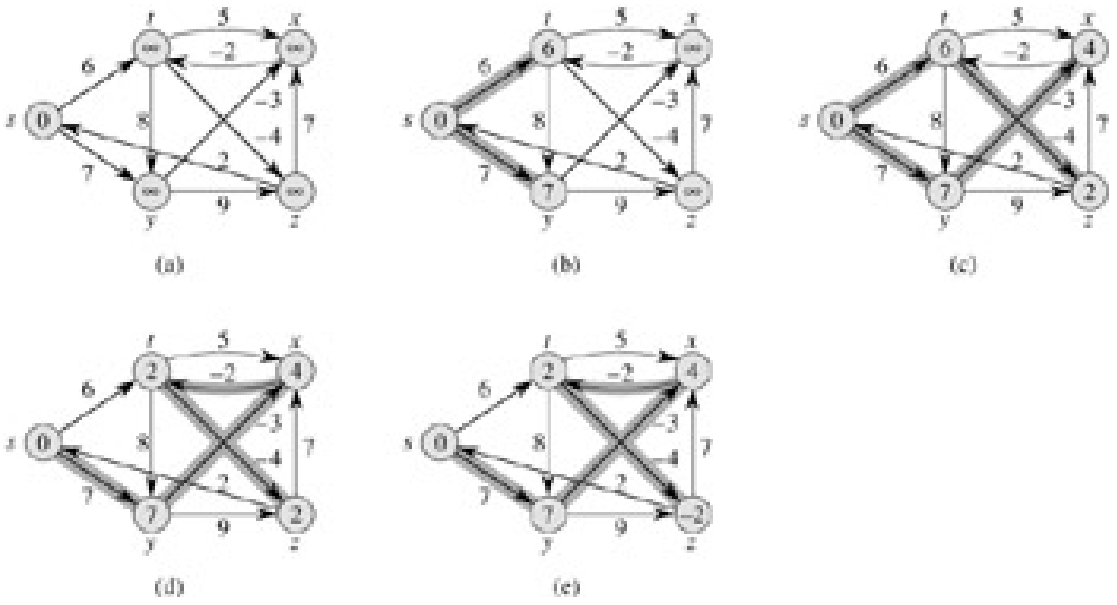


Figure: The execution of the Bellman-Ford algorithm. The source is vertex *s*. The *d* values are shown within the vertices, and shaded edges indicate predecessor values: if edge (*u, v*) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order (*t, x*), (*t, y*), (*t, z*), (*x, t*), (*y, x*), (*y, z*), (*z, x*), (*z, s*), (*s, t*), (*s, y*). (a) The situation just before the first pass over the edges. (b)-(e) The situation after each successive pass over the edges. The *d* and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.

3.2.8.3 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

DIJKSTRA(*G*, *w*, *s*)

1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S \leftarrow \emptyset$
3 $Q \leftarrow V[G]$
4 while $Q \neq \emptyset$
5 do $u \leftarrow$ EXTRACT-MIN(Q)
6 $S \leftarrow S \cup \{u\}$
7 for each vertex $v \in Adj[u]$
8 do RELAX(u, v, w)

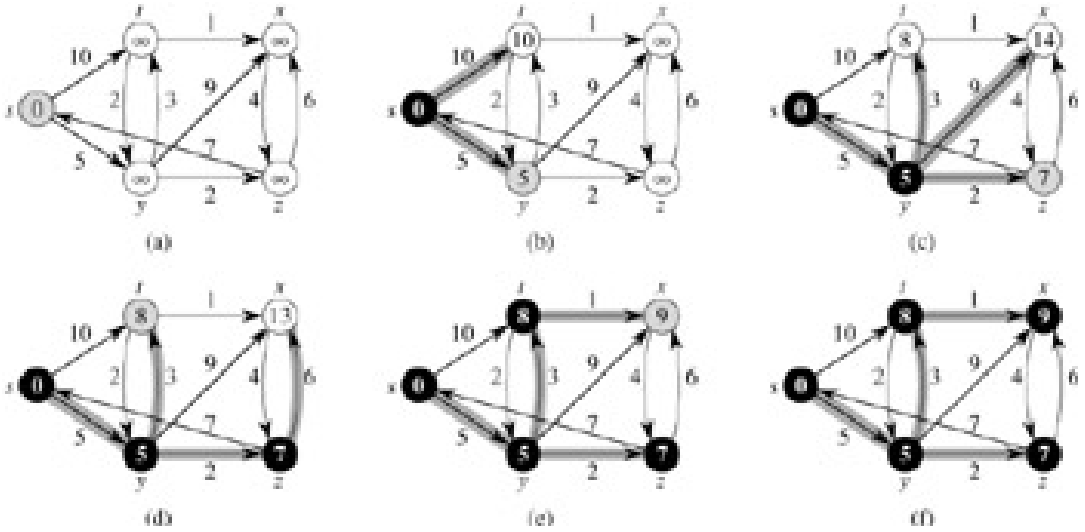


Figure: The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set *S*, and white vertices are in the min-priority queue $Q = V - S$. (a) The situation just before the first iteration of the while loop of lines 4-8. The shaded vertex has the minimum *d* value and is chosen as vertex *u* in line 5. (b)-(f) The situation after each successive iteration of the while loop. The shaded vertex in each part is chosen as vertex *u* in line 5 of the next iteration. The *d* and π values shown in part (f) are the final values.

Complexity

The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and extract minimum from Q is simply a linear search through all vertices in Q. In this case, the running time is $O(|E| + |V|^2) = O(|V|^2)$.

NOTES (UNIT-IV) Dynamic Programming, Backtracking, Branch and Bound

Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.

Example 1: [Knapsack] The solution to the knapsack problem (Section 4.3) may be viewed as the result of a sequence of decisions. We have to decide the values of Xi, $1 \leq i \leq n$. First we may make a decision on x 1 ,then on x2, then on X_J etc. An optimal sequence of decisions will maximize the objective function E p;x;. (It will also satisfy the constraints $\sum w_i x_i \leq M$ and $0 \leq x_i \leq 1$.)

Example 2: [Optimal Merge Patterns] An optimal merge pattern tells us which pair of files should be merged at each step. As a decision sequence, the problem calls for us to decide which pair of files should be merged first; which pair second; which pair third, etc. An optimal sequence of decisions is a least cost sequence. D

Example 3 [Shortest Path] One way to find a shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, which the fourth; etc. until vertex j is reached. An optimal sequence of decisions is one which results in a path of least length.

4.1 0/l-KNAPSACK :

- A solution to the knapsack problem may be obtained by making a sequence of decisions on the variables *xi, x2, ... ,Xn.*
- A decision on variable *x;* involves deciding which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x; are made in the order *x n, x n* -1 , ••• , *x* 1•
- Following a decision on *Xn*we may be in one of two possible states: the capacity remaining in the knapsack is *M* and no profit has accrued or the capacity remaining is *M* - *Wn*and a profit of *Pn*has accrued.
- It is clear that the remaining decisions *Xn-1,* ••• *,xi* must be optimal with respect to the problem state resulting from the decision on *Xn.* Otherwise, *Xn,* ••• , *x* 1 will not be optimal. Hence, the principle of optimality holds.
- Let$f_j(X)$ be the value of an optimal solution to KNAP(l,j, X). Since the principle of optimality holds, we obtain

$$f_n(M) = \max\{f_{n-1}(M), f_{n-1}(M - w_n) + p_n\}$$

For arbitrary *f;(X), i* >0, the above equation generalizes to

$$f_i(X) = \max\{f_{i-1}(X), f_{i-1}(X - w_i) + p_i\}$$

```
line  procedure DKP(p, w, n, M)
 1      S^0 ← {(0, 0)}
 2      for i ← 1 to n - 1 do
 3        S^i_1 ← {(P1, W1) | (P1 - p_i, W1 - w_i) ∈ S^(i-1) and W1 ≤ M}
 4        S^i ← MERGE_PURGE(S^(i-1), S^i_1)
 5      repeat
 6      (PX, WX) ← last tuple in S^(n-1)
 7      (PY, WY) ← (P1 + p_n, W1 + w_n) where W1 is the largest W in
          any tuple in S^(n-1) such that W + w_n ≤ M
        //trace back for x_n, x_(n-1), ..., x_1//
 8      if PX > PY then x_n ← 0
 9                 else x_n ← 1
10      endif
11      trace back for x_(n-1), ..., x_1
12    end DKP
```

Informal knapsack algorithm

```
line  procedure DKNAP(p, w, n, M, m)
        real p(n), w(n), P(m), W(m), pp, ww, M
        integer F(0:n), l, h, u, i, j, p, next
 1      F(0) ← 1; P(1) ← W(1) ←0   //S^0//
 2      l ← h ← 1   //start and end of S^0//
 3      F(1) ← next ← 2   //next free spot in P and W//
 4      for i ← 1 to n − 1 do   //generate S^i//
 5        k ← l
 6        u ← largest k, l ≤ k ≤ h, such that W(k) + w_i ≤ M
 7        for j ← l to u do   //generate S_1^i and merge//
 8          (pp, ww) ← (P(j) + p_i, W(j) + w_i)   //next element in S_1^i//
 9          while k ≤ h and W(k) ≤ ww do   //merge in from S^(i-1)//
10            P(next) ← P(k); W(next) ← W(k)
11            next ← next + 1; k ← k + 1
12          repeat
13          if k ≤ h and W(k) = ww then pp ← max(pp, P(k))
14                                         k ← k + 1
15          endif
16          if pp > P(next − 1) then (P(next), W(next)) ← (pp, ww)
17                                   next ← next + 1
18          endif
19          while k ≤ h and P(k) ≤ P(next − 1) do   //purge//
20            k ← k + 1
21          repeat
22        repeat
          //merge in remaining terms from S^(i-1)//
23        while k ≤ h do
24          (P(next), W(next)) ← (P(k), W(k))
25          next ← next + 1; k ← k + 1
26        repeat
          //initialize for S^(i+1)//
27        l ← h + 1; h ← next − 1; F(i + 1) ← next
28      repeat
29      call PARTS
30    end DKNAP
```

Algorithm 5.7 Algorithm for 0/1 knapsack problem

4.2 Matrix-chain multiplication

- Matrix-chain multiplication is an example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication .We are given a sequence (chain) <A1, A2,…..An> of n matrices to be multiplied, and we wish to compute the product:
  A1A2 …… An
- We can evaluate the above expression using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together.

- Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully Parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is <A1,A2,A3,A4> then we can fully parenthesize the product A1A2A3A4 in five distinct ways:

  (A1(A2(A3A4))) ,
  (A1((A2A3)A4)) ,
  ((A1A2)(A3A4)) ,
  ((A1(A2A3))A4) ,
  (((A1A2)A3)A4).

- How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode, which generalizes the SQUARE-MATRIX-MULTIPLY procedure.

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

- We can multiply two matrices A and B only if they are ***compatible*** the number of columns of A must equal the number of rows of B.
- If A is a p x q matrix and B is a q x r matrix, the resulting matrix C is a p x r matrix.
- To illustrate the different costs incurred by different parenthesizations of a matrixproduct, consider the problem of a chain <A1; A2; A3> of three matrices.
- Suppose that the dimensions of the matrices are 10 x 100, 100 x 5, and 5 x 50, respectively. If we multiply according to the parenthesization ((A1A2)A3), we perform 10 .100 . 5 = 5000 scalar multiplications to compute the 10 x 5 matrix product A1A2, plus another 10 .5. 50 D 2500 scalar multiplications to multiply this matrix by A3, for a total of 7500 scalar multiplications.
- According to the parenthesization (A1(A2A3)), we perform 100 . 5 . 50 = 25,000 scalar multiplications to compute the 100 x 50 matrix product A2A3, plus another 10 .100 .50 =50,000 scalar multiplications to multiply A1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.
- We state the ***matrix-chain multiplication problem*** as follows: given a chain<A1,A2,….An>of n matrices, where for i =1,2,….n, matrix Ai has dimension pi_1 x pi , fully parenthesize the product A1A2…. An in a way that minimizes the

number of scalar multiplications.

4.2.1 Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming let us convince ourselves that exhaustively checking all possible parenthesizations does not yield an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of matrices by P.n/. When n D 1, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When n _ 2, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and .k C 1/st matrices for any k = 1,2,….n _ 1. Thus, we obtain the recurrence

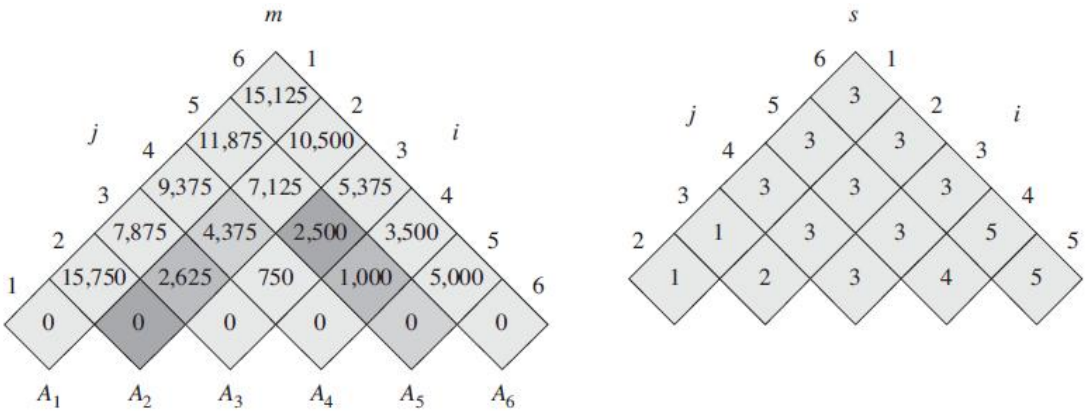
$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

4.2.2 Applying dynamic programming

We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                  // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```



The m and s tables computed by MATRIX-CHAIN-ORDER for n D 6 and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is m[1,6]= 15,125. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing scalar multiplications to multiply the 6 matrices is m[1,6]= 15,125. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing.

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\
&= 7125\,.
\end{aligned}
$$

4.2.3 Constructing optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The following recursive procedure prints an optimal parenthesization of <Ai, Ai+1, .. Aj i>, given the s table computed by MATRIX-CHAINORDER and the indices i and j . The initial call PRINT-OPTIMAL-PARENS S(1,…n)prints an optimal parenthesization of <A1;A2; …;An>.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

4.3 Longest common subsequence

- Biological applications often need to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called ***bases***, where the possible bases are adenine, guanine, cytosine, and thymine. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set {A; C; G; T}
- For example, the DNA of one organism may be S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, and the DNA of another organism may be S2 D=GTCGTTCGGAATGCCGTTGCTCTGTAAA.
- One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.
- For example, we can say that two DNA strands are similar if one is a substring of the other .In our example, neither S1 nor S2 is a substring of the other. Alternatively ,we could say that two strands are similar if the number of changes needed to turn one into the other is small.Yet another way to measure the similarity of strands S1 and S2 is by finding a third strand S3in which the bases in S3 appear in each of S1 and S2; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S3 we can find, the more similar S1 and S2 are. In our example, the longest strand S3 is GTCGTCGGAAGCCGGCCGAA.
- A subsequence of a given sequence is just the given sequence with zero or more elements left out. Formally, given a sequence X =<x1; x2; : : : ;xm>, another sequence Z = <1,2, …….k> is a ***subsequence*** of X if there exists a strictly increasing sequence <i_1; i_2,…i_K>of indices of X such that for all j =1;2,…..k, we have x_{ij} =j . For example, Z = <B,C, D,B> is a subsequence of X =<A, B,C, B,D,A,B> with corresponding index sequence <2; 3; 5; 7>
- In the ***longest-common-subsequence problem***, we are given two sequences X =<x1; x2; : : : ; xm> and Y = <y_1; y_2,…… y_n> and wish to find a maximum length common subsequence of X and Y . This section shows how to efficiently solve the LCS problem using dynamic programming.

4.3.1 Characterizing a longest common subsequence

In a brute-force approach to solving the LCS problem, we would enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y , keeping track of the longest subsequence we find. Each subsequence of X corresponds to a subset of the indices

{1,2,…m} of X. Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

4.3.2 A recursive solution

- we should examine either one or two sub problems when finding an LCS of X = <x_1; x_2;….x_m> and Y = <y_1; y_2,…..y_n>. If xm = yn, we must find an LCS of X_{m_1} and Y_{n_1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y . If x_m not equal to y_n, then we must solve two sub problems.

- Finding an LCS of X_{m_1} and Y and finding an LCS of X and Y_{n_1}. Whichever of these two LCSs is longer is an LCS of X and Y . Because these cases exhaust all possibilities, we know that one of the optimal sub problem solutions must appear within an LCS of X and Y . Because these cases exhaust all possibilities, we know that one of the optimal subproblem solutions must appear within an LCS of X and Y .

- We can readily see the overlapping-sub problems property in the LCS problem. To find an LCS of X and Y , we may need to find the LCSs of X and Y_{n_1} and of X_{m_1} and Y . But each of these sub problems has the sub problem of finding an LCS of X_{m_1} and Y_{n_1}. Many other sub problems share sub problems.
  As in the matrix-chain multiplication problem, our recursive solution to the. The optimal substructure of the LCS problem gives the recursive formula

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

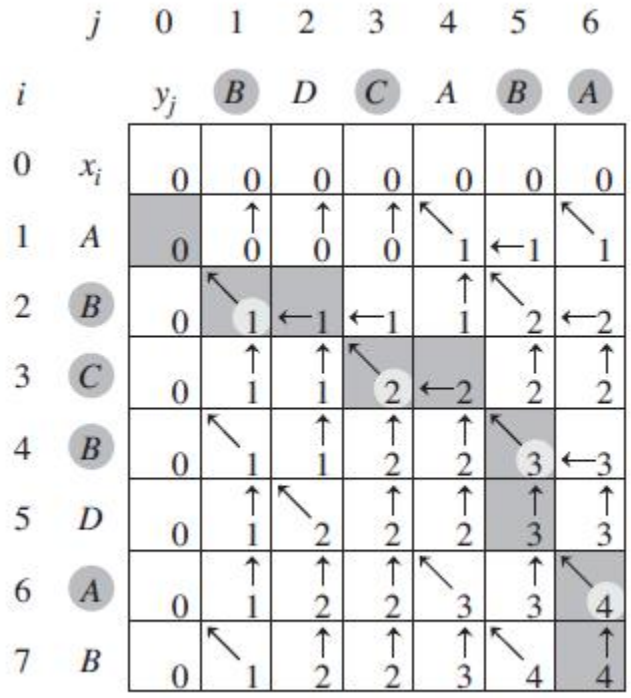
4.3.3 Computing the length of an LCS

Algorithm to compute the length of an LCS of two sequences is written here. Procedure LCS-LENGTH takes two sequences X = <x1; x2; …… xm> and Y =<y1;y2; ….yn> as inputs. It stores the c[i; j] values in a table c[0 ,..m; 0,…n], and it computes the entries in ***row-major*** order. The procedure also maintains the table b[1…m; 1… n] to help us construct an optimal solution. Intuitively,b[i; j]points to the table entry corresponding to the optimal subproblem solution chosen when computing c[i; j]. The procedure returns the b and c tables; c[m; n] contains the length of an LCS of X and Y .

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

4.3.4 Constructing an LCS

The b table returned by LCS-LENGTH enables us to quickly construct an LCS of X = <x_1, x_2,…. x_m> and Y =<y_1, y_2,…y_n>. We simply begin at b[m; n] and trace through the table by following the arrows. Whenever we encounter a "↖" in entry b[i; j], it implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH.



i		j	0	1	2	3	4	5	6
		y_j		B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

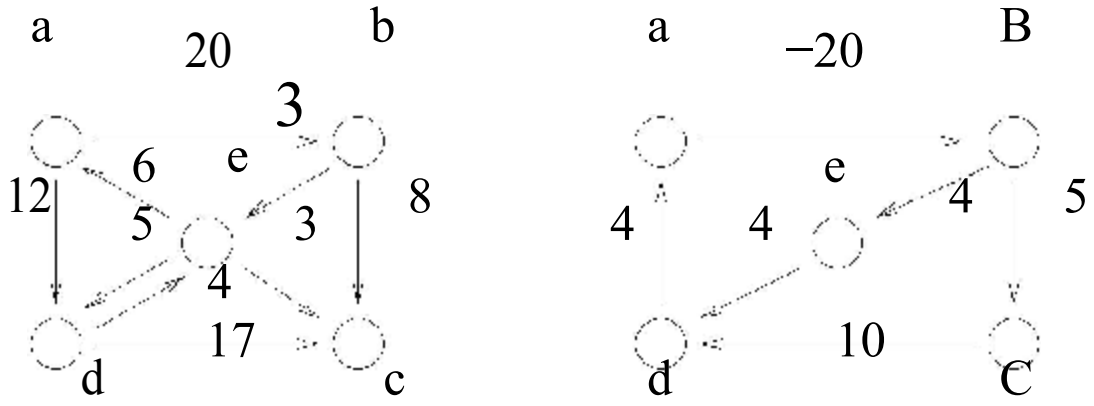
found. With this method, we encounter the elements of this LCS in reverse order. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS(b,X,X:*length*,Y:*length*).

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

The procedure takes time O(m + n), since it decrements at least one of i and j in each recursive call.

4.4 The All-Pairs Shortest Paths Problem

Given a weighted digraph G= (V, E) with a weight function W : E → R , where R is the set of real numbers, determine the length of the shortest path(i.e.,distance) between all pairs of vertices in . Here weassume that there are no cycle with zero or negativecost.



without negative cost cycle with negative cost cycle

Solution 1: Assume no negative*edges*.

Run Dijkstra's algorithm, times, once with each vertex as source.

O (n^3 log n) O(n^3) with more sophisticated data structures.

Solution 2: Assume no negative*cycles*.

Dynamic programming solution, based on a natural decomposition of the problem.

O(n^4) O(n^3log n)using " repeated squaring".

4.4.1 THE FLOYD-WARSHALL ALGORITHM

we shall use a different dynamic-programming formulation to solve the all-pairs shortest-paths problem on a directed graph G =(.V,E). The resulting algorithm, known as the ***Floyd-Warshall algorithm***, runs in ,O(V^3)time. As before, negative-weight edges may be present, but we assume that there are no negative-weight cycles. We follow the dynamic-programming process to develop the algorithm. After studying the resulting algorithm, we present a similar method for finding the transitive closure of a directed graph.

4.4.1.1 The structure of a shortest path

- In the Floyd-Warshall algorithm, we characterize the structure of a shortest path differently . The Floyd-Warshall algorithm considers the intermediate vertices of a

shortest path, where an ***intermediate*** vertex of a simple path $p = \langle v_1, v_2 \ldots v_l \rangle$ is any vertex of p other than v_1 or v_1, that is, any vertex in the set $\{v_2; v_3; \ldots . v_{1\_1}\}$.

- The Floyd-Warshall algorithm relies on the following observation. Under ourassumption that the vertices of G are $V = \{v_1, v_2, \ldots . v_n\}$
- let us consider a subset$\{1,2,\ldots . k\}$of vertices for some k. For any pair of vertices i; $j \in V$, consider allpaths from i to j whose intermediate vertices are all drawn from $\{1,2,\ldots k\}$, and
- let$p$ be a minimum-weight path from among them. (Path p is simple.)
- The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i

  to$j$ with all intermediate vertices in the set $\{1,2,\ldots .. k \_ 1\}$. The relationship depends on whether or not k is an intermediate vertex of path p.
- If k is not an intermediate vertex of path p, then all intermediate vertices ofpath p are in the set $\{1,2\ldots\ldots . k \_ 1\}$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $\{1,2\ldots k _ 1\}$is also ashortest path from i to j with all intermediate vertices in the set $\{1,2,\ldots . k\}$
- If k is an intermediate vertex of path p, then we decompose p into $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$,
- $p_1$is a shortest path from i to $k$with all intermediate vertices in the set $\{1,2,\ldots .. k\}$. In fact, we can make aslightly stronger statement. Because vertex k is not an intermediate vertex of pathp_1, all intermediate vertices of p_1 are in the set $\{1,2,\ldots\ldots k \_ 1\}$. There all intermediate vertices in $\{1, 2\ldots\ldots k _ 1\}$all intermediate vertices in $\{1, 2,\ldots . k \_ 1\}$

4.4.1.2 Recursive Solution to All Pair Shortest Problem

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$. When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. Following the above discussion, we define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases}$$

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

4.4.1.3 Computing the shortest path weights bottom up

We can use the following ottom-up procedure to compute the values $d_{ij}^{(k)}$ in order of increasing values of k. Its input is an n x n matrix.

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

The running time of algorithm is $\Theta(n^3)$.

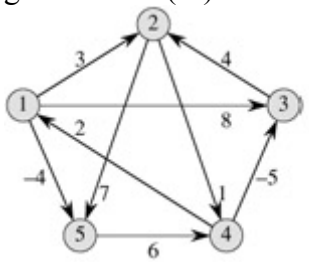


Figure1: A graph

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure2: The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 1

4.5 Resource-Allocation Problems

Resource-allocation problems, in which limited resources must be allocated among several activities, are often solved by dynamic programming.

To use linear programming to do resource allocation, three assumptions must be made:

- **Assumption 1** : The amount of a resource assigned to an activity may be any non negative number.

Assumption 2 : The benefit obtained from each activity is proportional to the amount of the resource assigned to the activity

- **Assumption 3**: The benefit obtained from more than one activity is the sum of the benefits obtained from the individual activities.

- Even if assumptions 1 and 2 do not hold, dynamic programming can be used to solve resource-allocation problems efficiently when assumption 3 is valid and when the amount of the resource allocated to each activity is a member of a finite set.

Generalized Resource Allocation Problem

- The problem of determining the allocation of resources that maximizes total benefit subject to the limited resource availability may be written as

$$\max \sum_{t=1}^{t=T} r_t(x_t)$$

$$\text{s.t. } \sum_{t=1}^{t=T} gt(xt) \le w$$

where x_t must be a member of {0,1,2,…}.

- To solve this by dynamic programming, define $f_t(d)$ to be the maximum benefit that can be obtained from activities $t, t+1, \ldots, T$ if d unites of the resource may be allocated to activities $t, t+1, \ldots, T$.
- We may generalize the recursions to this situation by writing
  $f_{T+1}(d) = 0$ for all d

$$f_t(d) = \max_{x_t} \{r_t(x_t) + f_{t+1}[d - g_t(x_t)]\}$$

where x_t must be a non-negative integer satisfying $g_t(x_t) \le d$.

4.6 BACKTRACKING

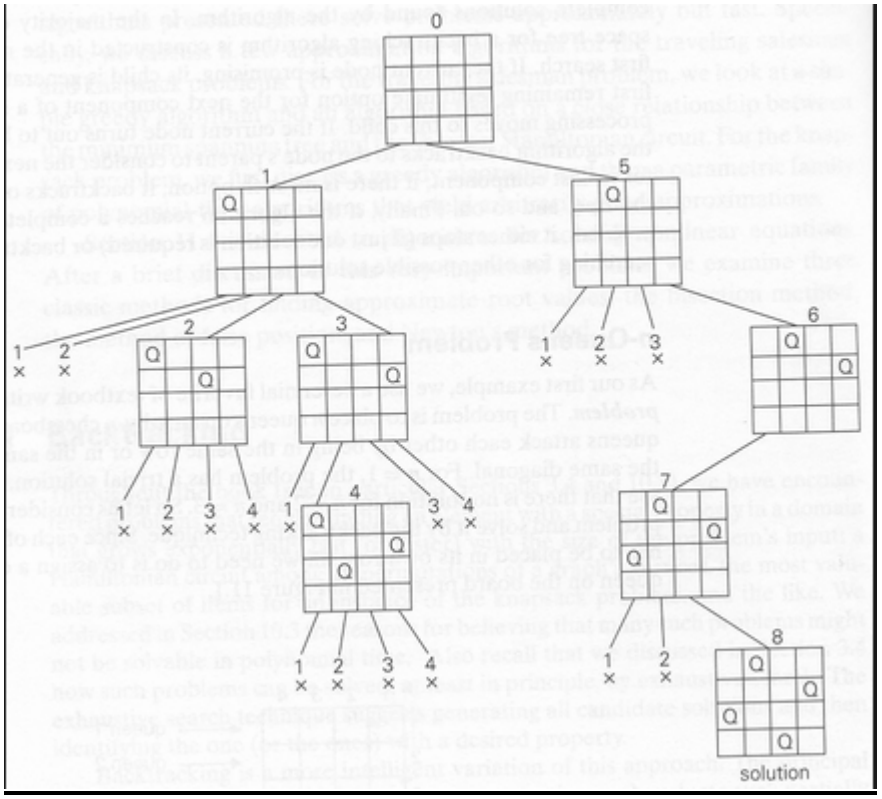
For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path.If you make the correct set of choices, you end up at the solution. On the other hand, if you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different path. Algorithms that use this approach are called **backtracking** algorithms.

4.6.1 N-QUEEN PROBLEM

- N-Queen problem is to place n-queen in a such manner on an n x n chessboard that no two queens attack each other by being in the same row, column or diagonal.
- It can be see that for n=1, the problem has a trivial solution , and no solution exist for n=2 and n=3. So first we will consider the 4- queen problem and then generalize it to n – queens problem.
- Since we have to place 4 queen such as q1,q2,q3,q4 on a chessboard such that o two queens attack each other. In such a conditions each queen must be placed o a different row, i.e., we place queen “i” on row “i”.
- Now place queen q1 in the first acceptable position (1,1).
- Next we place queen q2 i such a way that they do not attack each other. We find that if we place q2 in column 1 and 2 then the dead end is encountered. Thus the acceptable position for q2 is column 3 i.e. (2,3)but then no position is left for placing queen q3 safely.
- So we backtrack and one step and place the queen in (2,4), the next best possible solution. Q3 is placed in (3,2).
- Later this position also lead to dead end so we backtrack till ‘q1’ and place it in (1,2), q2 in (2,4) and q3 in (3,1) and q4 in (4,1).

- Thus the solution is <2,4,1,3> one of the feasible solution for 4-queen problem. the other solution is< 3,1,4,2>.

Place (k,i) returns a Boolean value that is true if k_H queen ca be placed in column i.It tests both whether I is distinct from all previous values $X_1,X_2,\ldots..,X_{k-1}$ ad whether there is no other queen on the same diagonal.



The implicit tree for 4-queen problem for solution <2,4,1,3> which is clear from the above diagram

ALGORITHM

```
Place (k, i)
{
for j := 1 to k-1 do
if (( x[ j ] = // in the same column
or (Abs( x [ j ] - i) =Abs ( j – k ))) // or in the same diagonal
then return false;
return true;
}
NQueens( k, n) //Prints all Solution to the n-queens problem
{
for i := 1 to n do
{
if Place (k, i) then
{
```

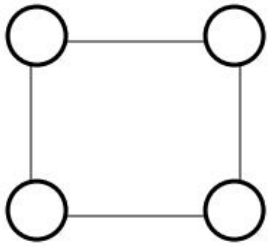
```
x[k] := i;
if ( k = n) then write ( x [1 : n]
else NQueens ( k+1, n);}}}
```

4.7 GRAPH COLORING

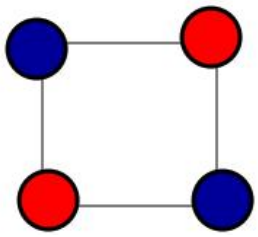
Assign colors to the vertices of a graph so that no adjacent vertices share the same color.

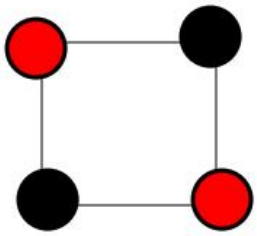
- Vertices i, j are adjacent if there is an edge from vertex i to vertex j

Example



No.of Colors used: 2



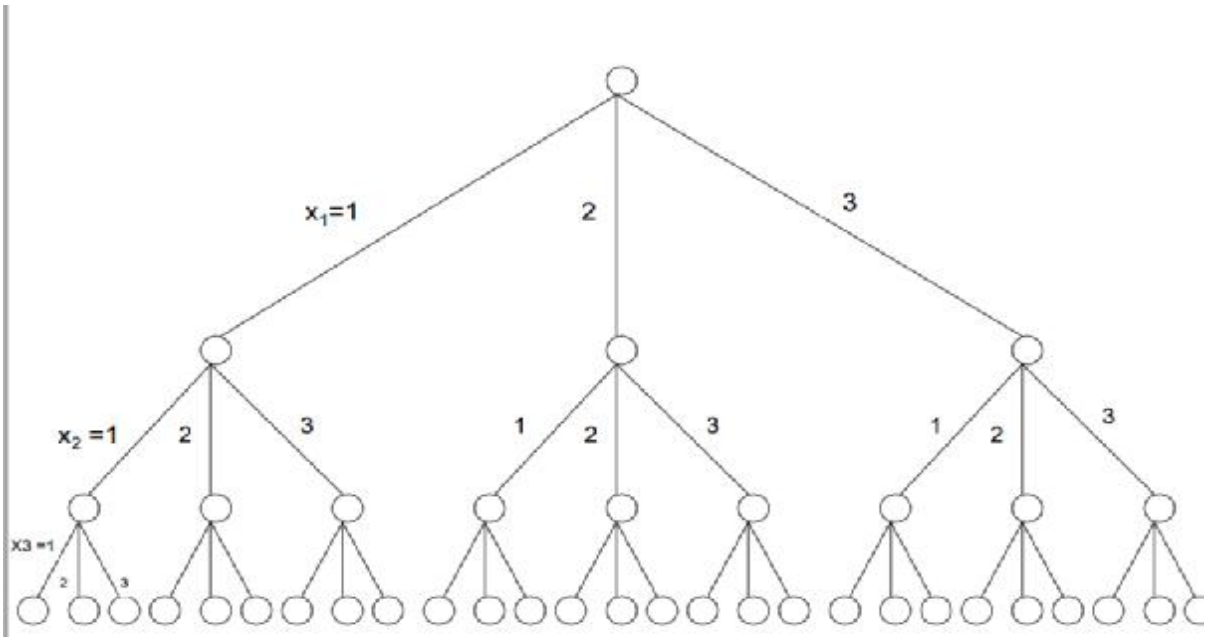


m-colorings problem

Find all ways to color a graph with at most m colors.

Problem formulation:-

- Represent the grapth with adjacency matrix G[1:n,1:n].
- The colors are represented by integer numbers 1,2,….m.
- Solution is represented by n- tuple (x1,….xn), where xi is thecolor of node i.



Solution space tree for mColorigng when n=3 and m=3

Algorithm :- finding all m- colorings of a graph. Function mColoring is begun by first assigning the graph to its adjacency matrix, setting the array x[ ] to zero, and then invoking the statement mColoring(1);

Algorithm

```
mColoring( k )
// k is the index of the next vertex to color.
{
repeat
{ // Generate all legal assignments for x[k]
NextValue( k ); // Assign to x[k] a legal color
if ( x[k]=0 ) then return; // No new color possible
if ( k=n ) then // At most m colors have been used to color the n vertices
write( x[1:n] );
else mColoring( k+1);
} until ( false );
}
```

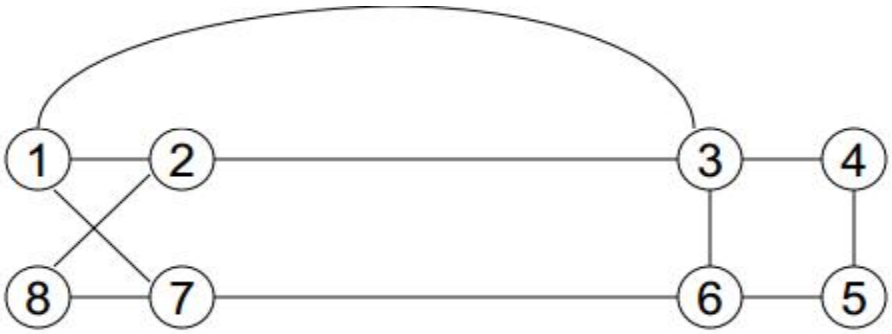
GENERATING A NEXT COLOR

```
Algorithm NextValue( k )
// x[1],…..x[ k-1 ] have been assigned integer values in the range [ 1 ,m ].
// A value for x[ k ] is determined in the range [ 0,m ]
{
repeat
{
```

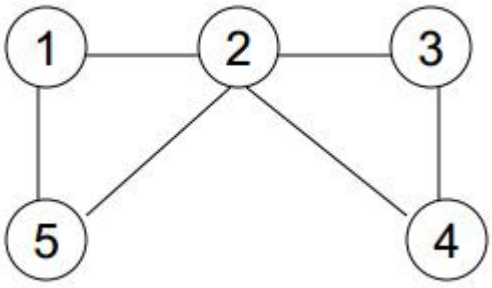
```
x[k] = ( x[k] +1) mod ( m+1 ); // Next highest color.
if ( x[k]=0 ) then return; // All colors have been used.
for j = 1 to n do
{
if (( G[ k,j ] ≠ 0 ) and ( x[k] = x[j] )) then
break;
}
if( j = n+1 ) then return; // Color found
} until ( false );
}
```

4.8 Hamiltonian cycle

A Hamiltonian cycle is a round-trip path along n edges of connected undirected graph G that visits every vertex once and returns to its starting position.



The above graph contains the Hamiltonian cycle
1,2,8,7,6,5,4,3,1



The above graph does not contain Hamiltonian cycles.

Find all possible Hamiltonian cycles

Problem formulation:-

- Represent the grapth with adjacency matrix G[1:n,1:n].
- Solution is represented by n- tuple (x1,….xn), where xi represents the ith visited vertex of the cycle.

- Start by setting x[2: n] to zero and x[1]=1, and then executing Hamiltonian(2);

Algorithm

```
Hamiltonian( k )
{
repeat
{ // Generate values for x[k]
NextValue( k ); // Assign a legal next value to x[ k ]
if ( x[k]=0 ) then return; // No new value possible
if ( k=n ) then write( x[1:n] );
else Hamiltonian( k+1);
} until ( false);
}
```

Algorithm

```
NextValue( k )
{
repeat
{
x[k] = ( x[k] +1) mod ( m+1 ); // Next vertex.
if ( x[k]=0 ) then return;
if( G[ x[ k-1], x[ k ] ] ≠ 0 )
{
for j:= 1 to k-1 do if ( x[ j ] = x [k ] ) then break;
if( j = k ) then // if true, then the vertex is distinct
if( ( k < n ) or ( ( k=n ) and G [ x[n], x[1]] ≠ 0 )
then return;
}
} until ( false );
}
```

4.9 Sum of Subset Problem

In the subset problem we have to find a subset s' of the given set S=<s_1,s_2,s_3,……s_n> where the elements of the set S are n positive integers in such manner that s' Є S and sum of the elements of subset 's'' is equal to some positive integer 'X.

The subset problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that represents that no decision is yet taken on any input. We assume that the elements of the given set are arranged in an increasing order:

$$S_1<S_2<S_3…..<S_n$$

The left child of the root node indicate that we have to include 'S_1 ' from the set 'S and the right child of the root node indicates that we have to exclude 'S1'.Each node stores the sum of the

partial solution elements. If at any stage the number equals to 'X' then search is successful and terminates.

The dead end in the tree occurs only when either of the two inequalities exists:

- The sum of s' is too large i.e.,

$s' + S_i + 1 > X$

- The sum of s' is too small i.e.,

$s' + \sum_{j=i+1}^{n} S_j < X$

4.10 BRANCH AND BOUND

- The design technique known as ***branch and bound*** is very similar to backtracking in that it searches a tree model of the solution space and is applicable to a wide variety of discrete combinatorial problems.
- Each node in the combinatorial tree generated in the last Unit defines a *problem state*. All paths from the root to other nodes define the ***state space***of the problem.
- ***Solution states***are those problem states's'for which the path from the root to *'s'* defines a tuple in the solution space. The leaf nodes in the combinatorial tree are the solution states.
- ***Answer states*** are those solution states's'for which the path from the root to *'s'*defines a tuple that is a member of the set of solutions (i.e., it satisfies the implicit constraints) of the problem.
- The tree organization of the solution space is referred to as the ***state space tree.***
- A node which has been generated and all of whose children have not yet been generated is called a***live node.***
- The ***live node*** whose children are currently being generated is called the *E*-node (node being expanded).
- A ***dead node***is a generated node, which is not to be expanded further or all of whose children have been generated.
- ***Bounding functions*** are used to kill live nodes without generating all their children.
- Depth first node generation with bounding function is called backtracking. State generation methods in which the *E*-node remains the *E*-node until it is dead lead to***branch-and-bound method.***
- The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node.
- In branch-and-bound terminology breadth first search(BFS)- like state space search will be called FIFO (First In First Output) search as the list of live nodes is a first -in-first -out list(or queue).
- A ***D-search*** (depth search) state space search will be called LIFO (Last In First Out) search, as the list of live nodes is a list-in-first-out list (or stack).
- Bounding functions are used to help avoid the generation of sub trees that do not contain an answer node.
- The branch-and-bound algorithms search a tree model of the solution space to get the solution. However, this type of algorithms is oriented more toward optimization. An algorithm of this type specifies a real -valued cost function for each of the nodes that appear in the search tree.
- Usually, the goal here is to find a configuration for which the cost function is minimized. The branch-and-bound algorithms are rarely simple. They tend to be quite complicated in

many cases.

- Let us see how a FIFO branch-and-bound algorithm would search the state space tree for the 4-queens problem.

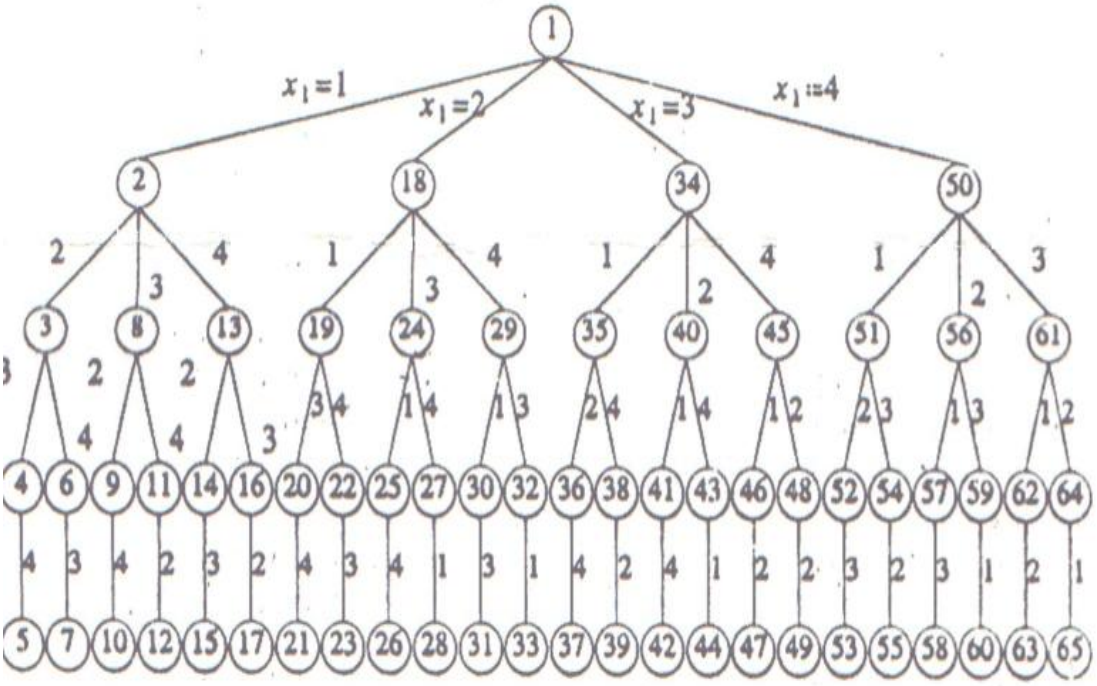


Fig A. Tree organization of 4-queen solution space. Nodes are numbered as in depth first search

- Initially, there is only one live node, node1. This represents the case in which no queen has been placed on the chessboard. This node becomes the E-node.
- It is expanded and its children, nodes2, 18, 34 and 50 are generated.
- These nodes represent a chessboard with queen1 in row 1and columns 1, 2, 3, and 4 respectively.
- The only live nodes 2, 18, 34, and 50.If the nodes are generated in this order, then the next E-node are node 2.
- It is expanded and the nodes 3, 8, and 13 are generated. Node 3 is immediately killed using the bounding function. Nodes 8 and 13 are added to the queue of live nodes.
- Node 18 becomes the next *E*-node. Nodes 19, 24, and 29 are generated. Nodes 19 and 24 are killed as a result of the bounding functions. Node 29 is added to the queue of live nodes.
- Now the *E*-node is node 34.Fig B shows the portion of the tree of Fig A. that is generated by a FIFO branch-and-bound search. Nodes that are killed as a result of the bounding functions are a "B" under them.
- Numbers inside the nodes correspond to the numbers in Fig A. Numbers outside the nodes give the order in which the nodes are generated by FIFO branch-and-bound.
- At the time the answer node, node 31, is reached, the only live nodes remaining are nodes 38 and 54.

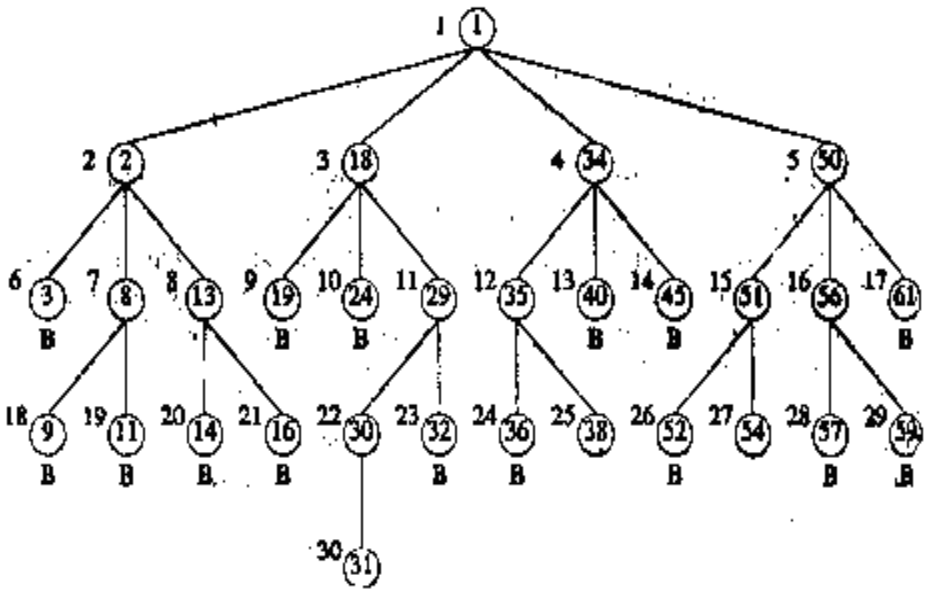


Fig B.Portion of 4-Queen state space tree generated by FIFO branch and bound

3.2.8. Least Cost (LC) Search:

- In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid and in a sense blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.
- Thus, in Example 8.1, when node 30 is generated, it should have become obvious to the search algorithm that this node will lead to answer node in one move. However, the rigid FIFO rule first requires the expansion of all live nodes generated before node 30 was expanded.
- The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}(.)$ for live nodes. The next *E*-node is selected on the basis of this ranking function.
- If in the 4-queens example we use a ranking function that assigns node 30 a better rank than all other live nodes, then node 30 will become *E*-node, following node 29.The remaining live nodes will never become *E*-nodes as the expansion of node 30 results in the generation of an answer node (node 31).
- The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node. For any node *x,* this cost could be (1) The number of nodes on the sub-tree *x* that need to be generated before any answer node is generated or, more simply,(2) The number of levels the nearest answer node (in the sub-tree *x*) is from *x.*
- Using cost measure (2), the cost of the root of the tree of Figure 8.1 is 4 (node 31 is four levels from node 1).The costs of nodes 18 and 34,29 and 35,and 30 and 38 are respectively 3, 2, and 1.The costs of all remaining nodes on levels 2, 3, and 4 are respectively greater than 3, 2, and 1.
- Using these costs as a basis to select the next E-node, the E-nodes are nodes 1, 18, 29,

and 30 (in that order).The only other nodes to get generated are nodes 2, 34, 50, 19, 24, 32, and 31.

- The difficulty of using the ideal cost function is that computing the cost of a node usually involves a search of the sub-tree x for an answer node. Hence, by the time the cost of a node is determined, that sub-tree has been searched and there is no need to explore x again. For this reason, search algorithms usually rank nodes only based on an estimate $\hat{g}(.)$ of their cost.
- Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. node x is assigned a rank using a function (.) such that $\hat{c}(x) = f(h(x)) + \hat{g}(x)$, where h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.
- A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$, to select the next e-node would always choose for its next e-node a live node with least (.).Hence, such a strategy is called an LC-search (least cost search).
- Cost function c (.) is defined as, if x is an answer node, then c(x) is the cost (level, computational difficulty, etc.) of reaching x from the root of the state space tree. If x is not an answer node, then c(x) =infinity, providing the sub-tree x contains no answer node; otherwise c(x) is equals the cost of a minimum cost answer node in the sub-tree x.
- It should be easy to see that $\hat{c}(.)$ with $f(h(x)) = h(x)$ is an approximation to c (.). From now on (x) is referred to as the cost of x.

4.10.2 Bounding:

- A branch -and-bound searches the state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node.

- We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC.
- A cost function $\hat{c}(.)$ such that $\hat{c}(x) <= c(x)$ is used to provide lower bounds on solutions obtainable from any node x. If upper is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}(x) > upper$ may be killed as all answer nodes reachable from x have cost $c(x) >= \hat{c}(x) > upper$. The starting value for upper can be set to infinity.

- Clearly, so long as the initial value for upper is no less than the cost of a minimum-cost answer node, the above rule to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node .Each time a new answer is found ,the value of upper can be updated.
- As an example optimization problem, consider the problem of job scheduling with deadlines. We generalize this problem to allow jobs with different processing times. We are given n jobs and one processor. Each job i has associated with it a three tuple
- (P_i d_i t_i).job i requires t_i units of processing time .if its processing is not completed by the deadline d_i, and then a penalty P_i is incurred.
- The objective is to select a subset j of the n jobs such that all jobs in j can be completed by their deadlines. Hence, a penalty can be incurred only on those jobs not in j. The subset j should be such that the penalty incurred is minimum among all possible subsets j.

such a j is optimal.

- Consider the following instances: n=4,(p_1, d_1, t_1)=(5,1,1),(p_2, d_2, t_2)=(10,3,2),(p_3, d_3, t_3)=(6,2,1),and(p_4, d_4, t_4)=(3,1,1).The solution space for this instances consists of all possible subsets of the job index set{1,2,3,4}. The solution space can be organized into a tree by means of either of the two formulations used for the sum of subsets problem.

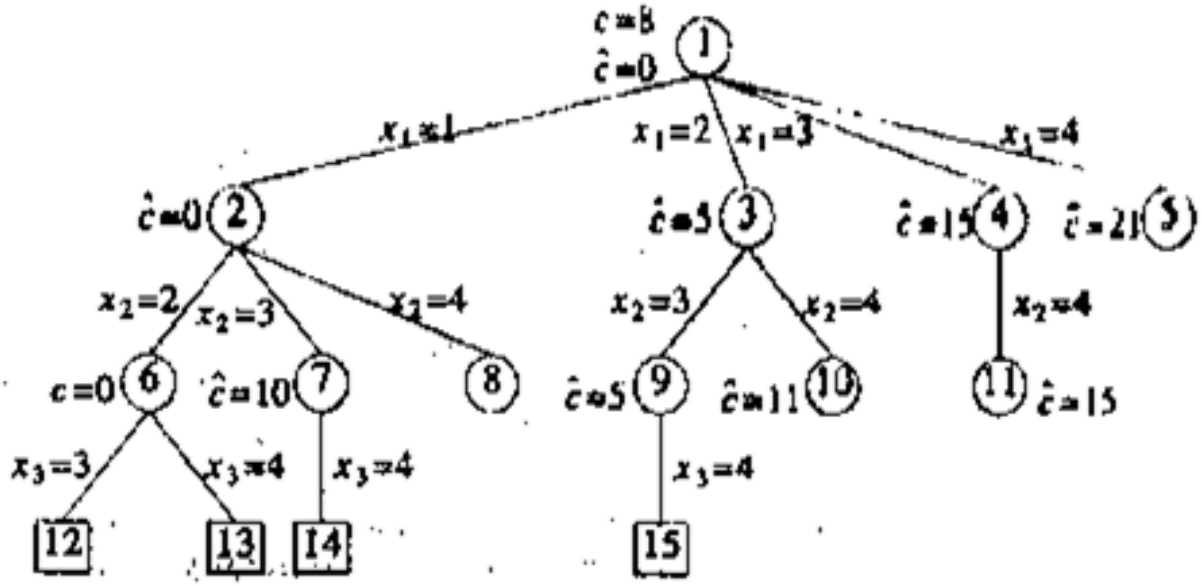


FIGURE A: State space tree corresponding to variable tuple size formulation

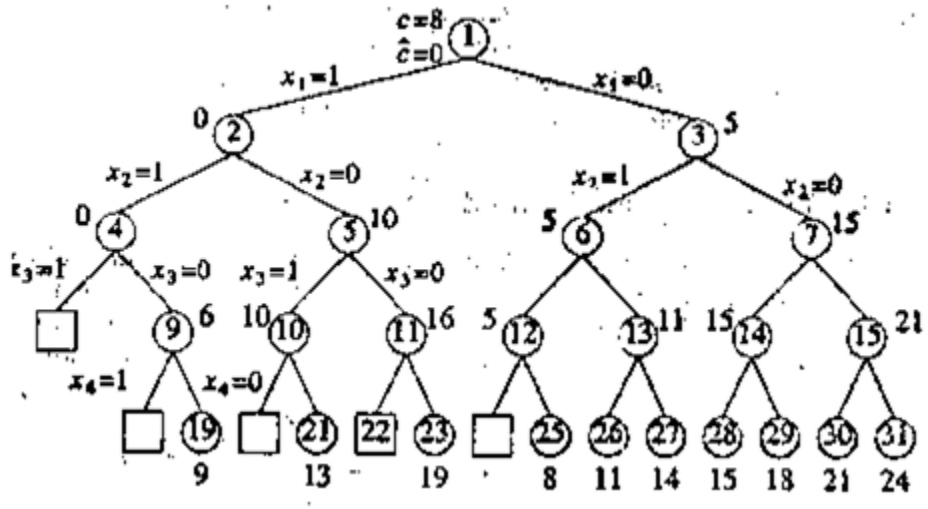


Figure B State Space tree corresponding to fixed tuple size formulation

Figure A corresponds to the variable tuple size formulations while figure B corresponds to the fixed tuple size formulation. In both figures square nodes represent infeasible subsets. In figure A all non-square nodes are answer nodes. Node 9 represents an optimal solution and is the only minimum-cost answer node .For this node j= {2,3} and the penalty (cost) is 8. In figure B only non-square leaf nodes are answer nodes. Node 25 represents the optimal solution and is also a

minimum-cost answer node. This node corresponds to j={2,3} and a penalty of 8. The costs of the answer nodes of figure B are given below the nodes.

4.10.3 TRAVELLING SALESMAN PROBLEM

INTRODUCTION

TSP includes a sales person who has to visit a number of cities during a tour and the condition is to visit all the cities exactly once and return back to the same city where the person started.

Steps:

Let G=(V,E) be a direct graph defining an instance for TSP.

- This graph is represented by a cost matrix where
  C_{ij}=the cost of the edge, If there is a path between vertex i and vertex j
  C_{ij}=∞ , if there is no path.

- Convert cost matrix in to reduced matrix i.e., every row and column should contain atleast one zero entry.
- Cost of reduced matrix is the sum of elements that are subtracted from rows and columns of cost matrix to make it reduced.
- Make the state space tree for reduced matrix.
- To find the next E node, find least cost valued node y calculation the reduced cost matrix with every node.
- If <i,j> edge is to be included , then three conditions to accomplish this task:
  I. Change all entries in row i and column j of A to ∞.
  II. Set A[j,i] =∞.
  III. Reduce all rows and columns in resulting matrix except for rows and columns containing ∞.
- Calculate the cost of matrix where
  Cost=L+cost(i,j)+r
  Where L= cost of original reduced cost matrix
  R=new reduced cost matrix.
- Repeat the above steps for all the nodes until all the nodes are generated and we get a path.

NOTES (UNIT-V) Selected Topics

5.1 Fast Fourier transform

5.1.1 Complex roots of unity

A ***complex*** n***th root of unity*** is a complex number ω such that

$$\omega^n=1 :$$

There are exactly n complex nth roots of unity: $e^{2\Pi ik/n}$ for k= 0, 1,……, n- 1.

To interpret this formula, we use the definition of the exponential of a complex number:

$$e^{iu}= \cos(u)+i \sin(u)$$

The value

$$\omega_n = e^{2\Pi i/n}$$

is the ***principal*** n***th root of unity,*** all other complex nth roots of unity are powers of ωn.

The n complex nth roots of unity,

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$$

5.1.2 The FFT

By using a method known as the ***fast Fourier transform (FFT)***, which takes advantage of the special properties of the complex roots of unity, we can compute DFTn (a) in time ,Θ(n lg n) as opposed to the $\Theta(n^2)$ time of the straightforward method. We assume throughout that n is an exact power of 2.

The FFT method employs a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients of A.x/ separately to define the two new polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ of degree-bound n/2:

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1},$$
$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}.$$

$A^{[0]}$ contains all the even-indexed coefficients of A and $A^{[1]}$ contains all the odd-indexed coefficients. It follows that

$$A(x) =A^{[0]}(x^2)+ A^{[1]}(x^2)$$

so that the problem of evaluating A(x) a $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to

1. evaluating the degree-bound n/2 polynomials $A^{[0]}$ (x) and $A^{[1]}$ (x) at the points $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$.
2. combining the results according to equation.

   The above computations are being done by recursive FFT algorithm

```
RECURSIVE-FFT(a)
 1  n = a.length                    // n is a power of 2
 2  if n == 1
 3      return a
 4  ω_n = e^(2πi/n)
 5  ω = 1
 6  a^[0] = (a_0, a_2, ..., a_{n-2})
 7  a^[1] = (a_1, a_3, ..., a_{n-1})
 8  y^[0] = RECURSIVE-FFT(a^[0])
 9  y^[1] = RECURSIVE-FFT(a^[1])
10  for k = 0 to n/2 - 1
11      y_k = y_k^[0] + ω y_k^[1]
12      y_{k+(n/2)} = y_k^[0] - ω y_k^[1]
13      ω = ω ω_n
14  return y                        // y is assumed to be a column vector
```

The RECURSIVE-FFT procedure works as follows. Lines 2–3 represent the basis of the recursion; the DFT of one element is the element itself, since in this case

$$\begin{aligned} y_0 &= a_0 \omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 . \end{aligned}$$

Lines 6–7 define the coefficient vectors for the polynomials $A^{[0]}$ and $A^{10]}$. Lines 4, 5, and 13 guarantee that ω is updated properly so that whenever lines 11–12 are executed, we have $\omega = \omega_n^k$. Lines 8–9 perform the recursive $DFT_{n/2}$ computations, setting, for k=0, 1,…….., n/2-1.

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k) , \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k) , \end{aligned}$$

or, since $\omega_{n/2}^k = \omega_n^{2k}$ by the cancellation lemma,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_n^{2k}) , \\ y_k^{[1]} &= A^{[1]}(\omega_n^{2k}) . \end{aligned}$$

Lines 11–12 combine the results of the recursive $DFT_{n/2}$ calculations. For y0, y_1, ……$y_{n/2-1}$, line 11 yields

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \end{aligned}$$

For $y_{n/2}$, $y_{n/2+1}$, ……., yn_1, letting k=0, 1,…….., n/2-1., line 12 yields

$$
\begin{aligned}
y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\
&= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \\
&= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\
&= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\
&= A(\omega_n^{k+(n/2)})
\end{aligned}
$$

Thus, the vector y returned by RECURSIVE-FFT is indeed the DFT of the input vector a.

5.1.3 Complexity

To determine the running time of procedure RECURSIVE-FFT, we note that exclusive of the recursive calls, each invocation takes time Θ (n), where n is the length of the input vector. The recurrence for the running time is therefore

$T(n) = 2T(n/2) + \Theta(n)$

$= \Theta (n \lg n)$

Thus, we can evaluate a polynomial of degree-bound n at the complex nth roots of unity in time Θ (n lg n) using the fast Fourier transform.

5.2 String matching

We formalize the ***string-matching problem*** as follows. We assume that the text is an array $T[1\ldots\ldots\ldots n]$ of length n and that the pattern is an array $P[1\ldots\ldots m]$ of length $m \le n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. For example, we may have Σ = {0, 1} or Σ = {a, b, . . . , z}. The character arrays P and T are often called ***strings*** of characters.

We say that pattern P occurs with shift s in text T (or, equivalently, that pattern P occurs beginning at position s + 1 in text T) if $0 \le s \le n - m$ and T [s + 1 _ s + m] = P[1 _ m] (that is, if T [s + j] = P[j], for $1 \le j \le m$). If P occurs with shift s in T , then we call s a valid shift; otherwise, we call s an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T . Figure illustrates these definitions.

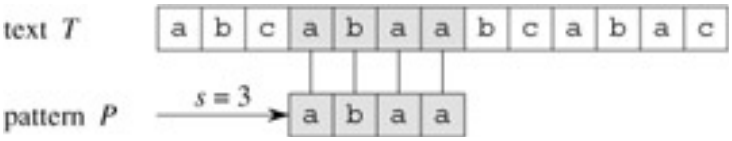


Figure: The string-matching problem. The goal is to find all occurrences of the pattern P = abaa in the text T = abcabaabcabac. The pattern occurs only once in the text, at shift s = 3. The shift s = 3 is said to be a valid shift. Each character of the pattern is connected by a vertical line to the matching character in the text, and all matched characters are shown shaded.

An analysis of all string matching algorithm is given by following Table

Algorithm	**Preprocessing time**	**Matching time**
Naive	0	$O((n - m + 1)m)$
Rabin-Karp	$\Theta(m)$	$O((n - m + 1)m)$
Finite automaton	$O(m \lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

5.2.1 The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1\ldots\ldots m] = T[s + 1\ldots\ldots..s + m]$ for each of the $n - m + 1$ possible values of s.

NAIVE-STRING-MATCHER(*T*, *P*)

1 $n \leftarrow length[T]$

2 $m \leftarrow length[P]$

3 **for** $s \leftarrow 0$ **to** $n - m$

4 **do if** $P[1 \_ m] = T[s + 1 \_ s + m]$

5 **then** print "Pattern occurs with shift" s

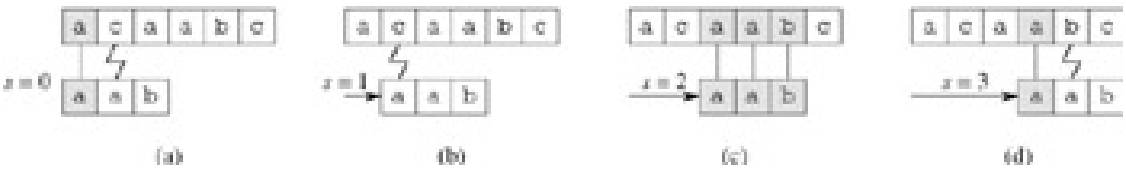


Figure: The operation of the naive string matcher for the pattern P = aab and the text T = acaabc. We can imagine the pattern P as a "template" that we slide next to the text. (a)-(d) The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. One occurrence of the pattern is found, at shift s = 2, shown in part (c).

Complexity

Procedure NAIVE-STRING-MATCHER takes time $O((n - m + 1)m)$, and this bound is tight in the worst case.

5.2.2 The Rabin-Karp algorithm

Rabin and Karp have proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n - m + 1)m)$. Based on certain assumptions, however, its average-case running time is better.

Given a pattern $P[1\ldots\ldots\ldots m]$, we let p denote its corresponding decimal value. In a similar manner, given a text $T[1\ldots\ldots\ldots n]$, we let ts denote the decimal value of the length-m substring $T[s + 1 \_ s + m]$, for $s = 0, 1, \ldots, n - m$. Certainly, $ts = p$ if and only if $T[s + 1 \_ s + m] = P[1\ldots\ldots m]$; thus, s is a valid shift if and only if $ts = p$. If we could compute p in time $\Theta(m)$ and all the ts values in a total of $\Theta(n - m + 1)$ time,[1] then we could determine all valid shifts s in time $\Theta(m) + \Theta(n - m + 1) = \Theta(n)$ by comparing p with each of the ts's.

We can compute p in time $\Theta(m)$ using Horner's rule:

$p = P[m] + 10\,(P[m - 1] + 10(P[m - 2] + \cdots + 10(P[2] + 10P[1])\,))$.

The value $t0$ can be similarly computed from $T\,[1\ldots.\ m]$ in time $\Theta(m)$.

To compute the remaining values $t1, t2, \ldots, tn\text{-}m$ in time $\Theta(n - m)$, it suffices to observe that $ts{+}1$ can be computed from ts in constant time, since

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]\ .$$

For example, if $m = 5$ and $ts = 31415$, then we wish to remove the high-order digit $T\,[s + 1] = 3$ and bring in the new low-order digit (suppose it is $T\,[s + 5 + 1] = 2$) to obtain

$$ts{+}1 = 10(31415 - 10000 \cdot 3) + 2$$
$$= 14152.$$

Subtracting $10^{m\text{-}1}\ T[s + 1]$ removes the high-order digit from ts, multiplying the result by 10 shifts the number left one position, and adding $T\,[s + m + 1]$ brings in the appropriate low order digit. The only difficulty with this procedure is that p and ts may be too large to work with conveniently.

With a d-ary alphabet $\{0, 1, \ldots, d - 1\}$, we choose q so that dq fits within a computer word and adjust the recurrence to work modulo q, so that it becomes

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q\ ,$$

where $h = d^{m\text{-}1} \pmod q$ is the value of the digit "1" in the high-order position of an m-digit text window.

The following algorithm implemented the above idea and inputs are text T, pattern P, the radix d and the prime q.

RABIN-KARP-MATCHER(*T*, *P*, *d*, *q*)

1 $n \leftarrow length[T]$

2 $m \leftarrow length[P]$

3 $h \leftarrow dm\text{-}1 \bmod q$

4 $p \leftarrow 0$

5 $t0 \leftarrow 0$

6 **for** $i \leftarrow 1$ **to** m ▹ Preprocessing.

7 **do** $p \leftarrow (dp + P[i]) \bmod q$

8 $t0 \leftarrow (dt0 + T[i]) \bmod q$

9 **for** $s \leftarrow 0$ **to** $n - m$ ▹ Matching.

10 **do if** $p = ts$

11 **then if** $P[1\ \_\ m] = T\,[s + 1\ \_\ s + m]$

12 **then** print "Pattern occurs with shift" s

13 **if** $s < n - m$
14 **then** $ts{+}1 \leftarrow (d(ts - T[s + 1]h) + T[s + m + 1]) \bmod q$

The following figure shows the implementation of algorithm

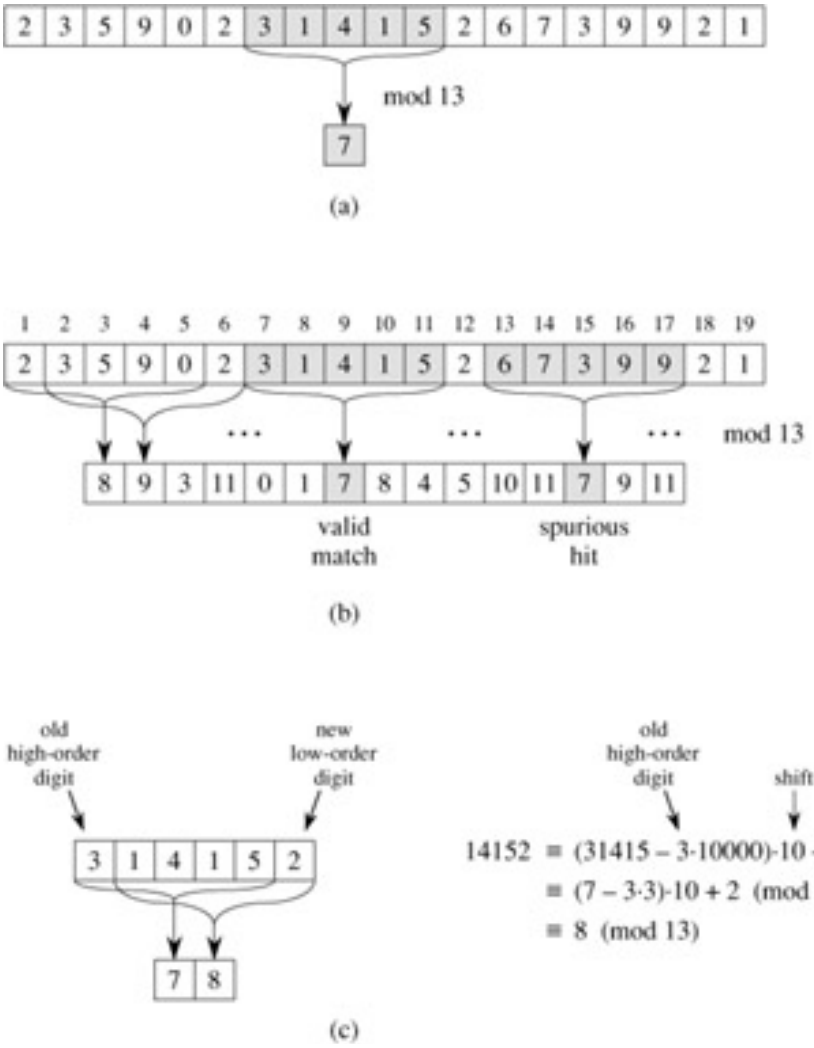


Figure: The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. *(a)* A text string. A window of length 5 is shown shaded. The numerical value of the shaded number is computed modulo 13, yielding the value 7. *(b)* The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern $P = 31415$, we look for windows whose value modulo 13 is 7, since $31415 \equiv 7 \pmod{13}$. Two such windows are found, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit. *(c)* Computing the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low order digit 2 gives us the new value 14152. All computations are performed modulo 13, however, so the value for the first window is 7, and the value computed for the new window is 8.

5.2.3 The Knuth-Morris-Pratt algorithm

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt using just an auxiliary function $\pi[1\ldots\ldots m]$ precomputed from the pattern in time $\Theta(m)$. The array π allows the transition function δ to be computed efficiently. For any state $q = 0, 1, \ldots\ldots\ldots, m$ and any character $a \in \Sigma$, the value $\pi[q]$ contains the information that is independent of a and is needed to compute $\delta(q, a)$. Since the array π has only m entries, whereas δ has $\Theta(m\ |\Sigma|)$ entries, we save a factor of $|\Sigma|$ in the preprocessing time by computing π rather than δ.

The prefix function for a pattern

The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself and shown by following figure.

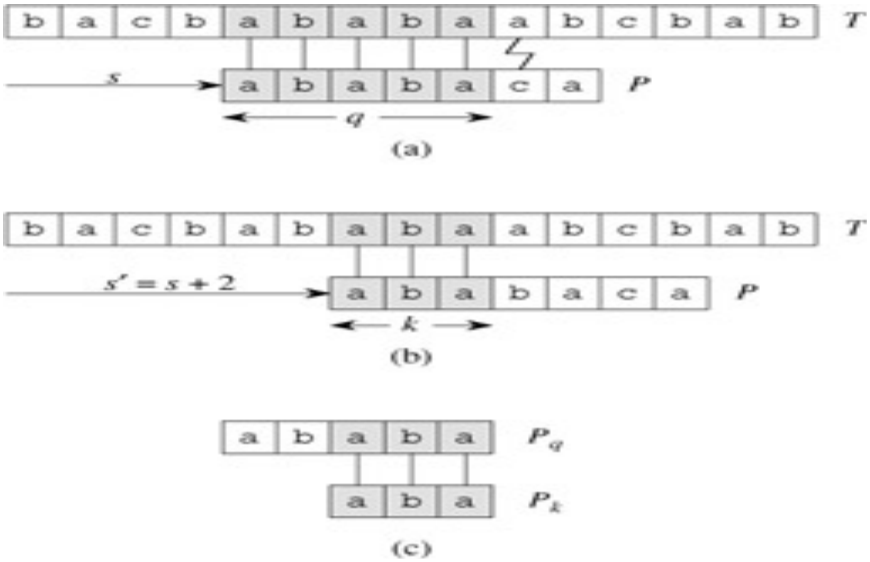


Figure: The prefix function π. *(a)* The pattern P = ababaca is aligned with a text T so that the first $q = 5$ characters match. Matching characters, shown shaded, are connected by vertical lines. *(b)* Using only our knowledge of the 5 matched characters, we can deduce that a shift of $s + 1$ is invalid, but that a shift of $s' = s + 2$ is consistent with everything we know about the text and therefore is potentially valid. *(c)* The useful information for such deductions can be precomputed by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a proper suffix of $P5$ is $P3$. This information is precomputed and represented in the array π, so that $\pi[5] = 3$. Given that q characters have matched successfully at shift s, the next potentially valid shift is at $s' = s + (q - \pi[q])$.

We formalize the precomputation required as follows. Given a pattern $P[1\ldots m]$, the ***prefix function*** for the pattern P is the function $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m - 1\}$ such that

$$\pi[q] = \max\ \{k : k < q \text{ and } Pk \supset Pq\}.$$

That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of Pq.

The Knuth-Morris-Pratt matching algorithm is given in pseudocode below as the procedure KMP-MATCHER which calls auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π.

KMP-MATCHER(*T*, *P*)

```
1 n ← length[T]
2 m ← length[P]
3 π ← COMPUTE-PREFIX-FUNCTION(P)
4 q ← 0 //Number of characters matched.
5 for i ← 1 to n //Scan the text from left to right.
6 do while q > 0 and P[q + 1] ≠ T[i]
7 do q ← π[q] //Next character does not match.
8 if P[q + 1] = T[i]
9 then q ← q + 1 //Next character matches.
10 if q = m //Is all of P matched?
11 then print "Pattern occurs with shift" i - m
12 q ← π[q] //Look for the next match.
```

COMPUTE-PREFIX-FUNCTION(*P*)

```
1 m ← length[P]
2 π[1] ← 0
3 k ← 0
4 for q ← 2 to m
5 do while k > 0 and P[k + 1] ≠ P[q]
6 do k ← π[k]
7 if P[k + 1] = P[q]
8 then k ← k + 1
9 π[q] ← k
10 return π
```

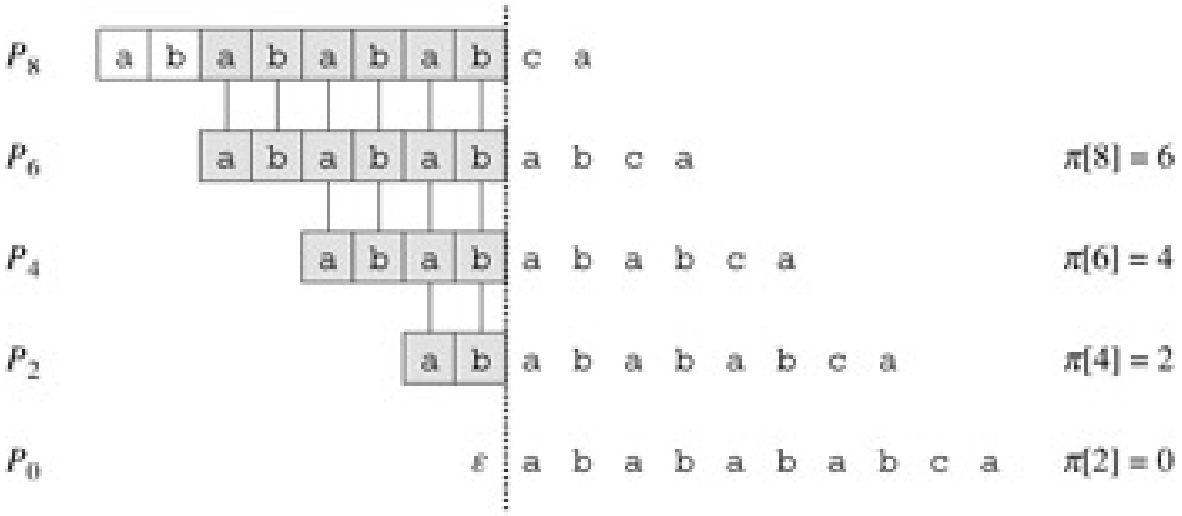
Running-time analysis

The running time of COMPUTE-PREFIX-FUNCTION is $\Theta(m)$, Compared to FINITE-AUTOMATON-MATCHER, by using π rather than δ, we have reduced the time for preprocessing the pattern from $O(m\ |\Sigma|)$ to $\Theta(m)$, while keeping the actual matching time bounded by $\Theta(n)$.

The figure given below show the implementation of KMP algorithm.

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)



(b)

Figure: An example for the pattern P = abababacba and q = 8. *(a)* The π function for the given pattern. Since π[8] = 6, π[6] = 4, π[4] = 2, and π[2] = 0, by iterating π we obtain π*[8] = {6, 4, 2, 0}. *(b)* We slide the template containing the pattern P to the right and note when some prefix Pk of P matches up with some proper suffix of $P8$; this happens for k = 6, 4, 2, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after $P8$. Successive rows show all the shifts of P that cause some prefix Pk of P to match some suffix of $P8$. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k : k < q$ and $Pk \supset Pq\} = \{6, 4,2, 0\}$. The lemma claims that $\pi^*[q] = \{k : k < q$ and $Pk \supset Pq\}$ for all q.

5.3 Theory of NP completeness

The concept of NP-completeness is perhaps the most difficult one in the field of design and analysis of algorithms. It is impossible to present this idea formally. We shall instead present an informal discussion of these concepts.

Let us first define some problems.

5.3.1 The partition problem: We are given a set S of numbers and we are asked to determine whether S can be partitioned into two subsets S_1 and S_2 such that that the sum of elements in S_1 is equal to the sum of elements of S_2.

For example, let $S = \{13, 2, 17, 20, 8\}$. The answer to this problem instance is "yes" because we can partition S into $S_1 = \{13, 17\}$ and $S_2 = \{2, 20, 8\}$.

5.3.2 The Sum of Subset Problem: We are given a set *S* of numbers and a constant *c* and we are asked to determine whether there exists a subset *S'* of *S* such that the sum of *S'* is equal to *c*.

For example, let *S* = {12, 9, 33, 42, 7, 10, 5} and *c* = 24. The answer of this problem instance is "yes" as there exists *S'* = {9, 10, 5} and the sum of the elements in *S'* is equal to 24. If *c* is 6, the answer will be "no".

5.3.3 The Satisfiability Problem: We are given a Boolean formula *X* and we are asked whether there exists an assignment of true or false to the variables in *X* which makes *X* true.

For example, let *X* be $(-x_1 \vee x_2 \vee -x_3) \wedge (x_1 \vee -x_2) \wedge (x_2 \vee x_3)$. Then the following assignment will make *X* true and the answer will be "yes".

$$x_1 \leftarrow F, x_2 \leftarrow F, x_3 \leftarrow T$$

If *X* is $-x_1 \wedge x_1$, there will be no assignment which can make *X* true and the answer will be "no".

5.3.4 The Minimal Spanning Tree Problem: Given a graph *G*, find a spanning tree *T* of *G* with the minimum length.

5.3.5 The Traveling Salesperson Problem: Given a graph *G* = (*V*,*E*), find a cycle of edges of this graph such that all of the vertices in the graph is visited exactly once with the minimum total length.

For example, consider Figure below. There are two cycles satisfying our condition. They are $C_1 = a \to b \to e \to d \to c \to f \to a$ and $C2 = a \to c \to b \to e \to d \to f \to a$. C_1 is shorter and is the solution of this problem instance.

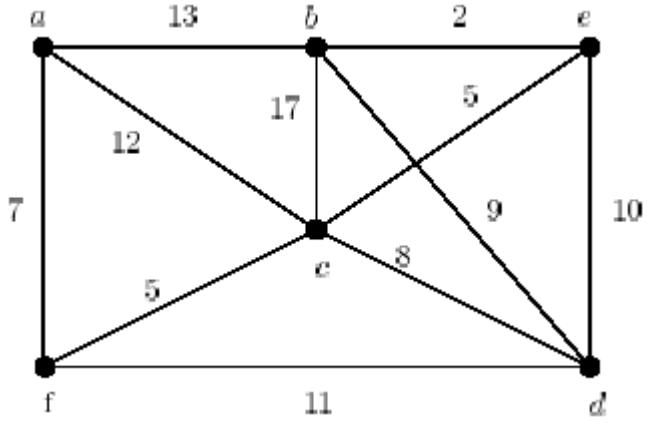


Figure : A Graph

For the partition problem, the sum of subset problem and the satisfiability problem, their solutions are either "yes" or "no". They are called *decision problems*. The minimal spanning tree problem and the traveling salesperson problem are called *optimization problems*.

For an optimization problem, there is always a decision problem corresponding to it. For instance, consider the minimal spanning tree problem, we can define a decision version of it as follows: Given a graph G, determine whether there exists a spanning tree of G whose total length is less than a given constant c. This decision version of the minimal spanning tree can be solved after the minimal spanning tree problem, which is an optimization problem, is solved. Suppose the total length of the minimal spanning tree is a. If $a < c$, the answer is "yes"; otherwise, its answer is "no". The decision version of this minimal spanning tree problem is called the *minimal spanning tree decision problem*. Similarly, we can define the longest common subsequence decision problem as follows: Given two sequences, determine whether there exists a common subsequence of them whose length is greater than a given constant c. We again call this decision problem the *longest common subsequence decision problem*. The decision version problem will be solved as soon as the optimization problem is solved.

In general, optimization problems are more difficult than decision problems. To investigate whether an optimization problem is difficult to solve, we merely have to see whether its decision version is difficult or not. If the decision version is difficult already, the optimization version must be difficult.

Before discussing NP-complete problems, note that there is a term called *NP problem*. We cannot formally define NP problems here as it is too complicated to do so. The reader may just remember the following: (1) NP problems are all decision problems. (2) Nearly all of the decision problems are NP problems. Among the NP problems, there are many problems which have polynomial algorithms. They are called *P problems*. For instance, the minimal spanning tree decision problem and the longest common subsequence decision problem are all P problems. There are also a large set of problems which, up to now, have no polynomial algorithms.

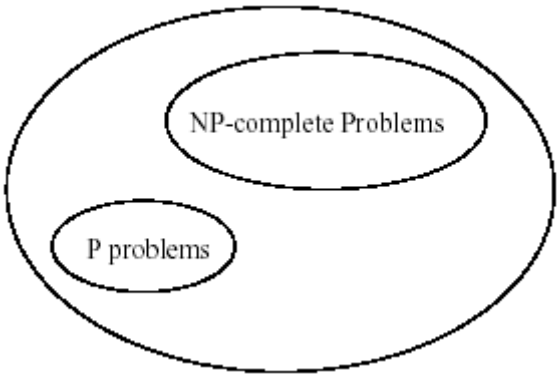


Figure : NP Problems

NP-complete problems constitute a subset of NP problems, as shown in Figure above. Precise and formal definition of NP-complete problems cannot be given in this book. But some important properties of NP-complete problems can be stated as follows:

(1) Up to now, no NP-complete problem has any worst case polynomial algorithm.

(2) If any NP-complete problem can be solved in polynomial time in worst case, all NP problems, including all NP-complete problems, can be solved in polynomial time in worst

case.

(3) Whether a problem is NP-complete or not has to be formally proved and there are thousands of problems proved to be NP-complete problems.

(4) If the decision version of an optimization problem is NP-complete, this optimization problem is called *NP-hard*.

Base upon the above facts, we can conclude that all NP-complete and NP-hard problems must be difficult problems. Not only they do not have polynomial algorithms at present, it is quite unlikely that they can have polynomial algorithms in the future because of the second property stated above.

The satisfiability problem is a famous NP-complete problem. The traveling salesperson problem is an NP-hard problem. Many other problems, such as the chromatic number problem, vertex covering problem, bin packing problem, 0/1 knapsack problem and the art museum problem are all NP-hard. In the future, we will often claim that a certain problem is NP-complete without giving a formal proof. Once a problem is said to be NP-complete, it means it is quite unlikely that a polynomial algorithm can be designed for it. In fact, the reader should never even try to find a polynomial algorithm for it. But the reader must understand that we cannot say that there exists no polynomial algorithms for NP-complete problems. We are merely saying that the chance of having such algorithms is very small.

It should be noted here that NP-completeness refers to worst cases. Thus, it is still possible to find an algorithm for an NP-complete problem which has polynomial time-complexity in average cases. It is our experience that this is also quite difficult as the analysis of average cases is usually quite difficult to begin with. Is is also possible to design some algorithms which perform rather well although we cannot have an average case analysis of them.

Should we give up hope when we have proved that a problem is NP-hard? No, we should not. Whenever we have proved a problem to be NP-complete, we should try to design an approximation algorithm for it.

5.4 Approximation algorithms

An optimization problem algorithm has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) .$$

We also call an algorithm that achieves an approximation ratio of $\rho(n)$ a ***$\rho(n)$ approximation algorithm*** The definitions of approximation ratio and of $\rho(n)$-approximation algorithm apply for both minimization and maximization problems. For a maximization problem, $0 < C \le C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution. Similarly, for a minimization problem, $0 < C^* \le C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of an

optimal solution. The approximation ratio of an approximation algorithm is never less than 1, since C/C^* 003C; 1 implies $C^*/C > 1$. Therefore, a 1-approximation algorithm produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

An ***approximation scheme*** for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1 + \epsilon)$-approximation algorithm. We say that an approximation scheme is a ***polynomial-time approximation scheme*** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

We say that an approximation scheme is a ***fully polynomial-time approximation scheme*** if it is an approximation scheme and its running time is polynomial both in $1/\epsilon$ and in the size n of the input instance. For example, the scheme might have a running time of $O((1/\epsilon)^2 n^3)$.

5.4.1 The vertex-cover problem

A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V \epsilon V$ such that if (u, v) is an edge of G, then either $u \epsilon V'$ or $v \epsilon V'$ (or both). The size of a vertex cover is the number of vertices in it.

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***. This problem is the optimization version of an NP-complete decision problem.

APPROX-VERTEX-COVER(*G*">

1 $C \leftarrow \emptyset$

2 $E' \leftarrow E[G]$

3 **while** $E' \neq \emptyset$

4 **do** let (u, v) be an arbitrary edge of E'

5 $C \leftarrow C$ _ $\{u, v\}$

6 remove from E' every edge incident on either u or v

7 **return** C

The figure given below shown the implementation OF Vertex-cover problem

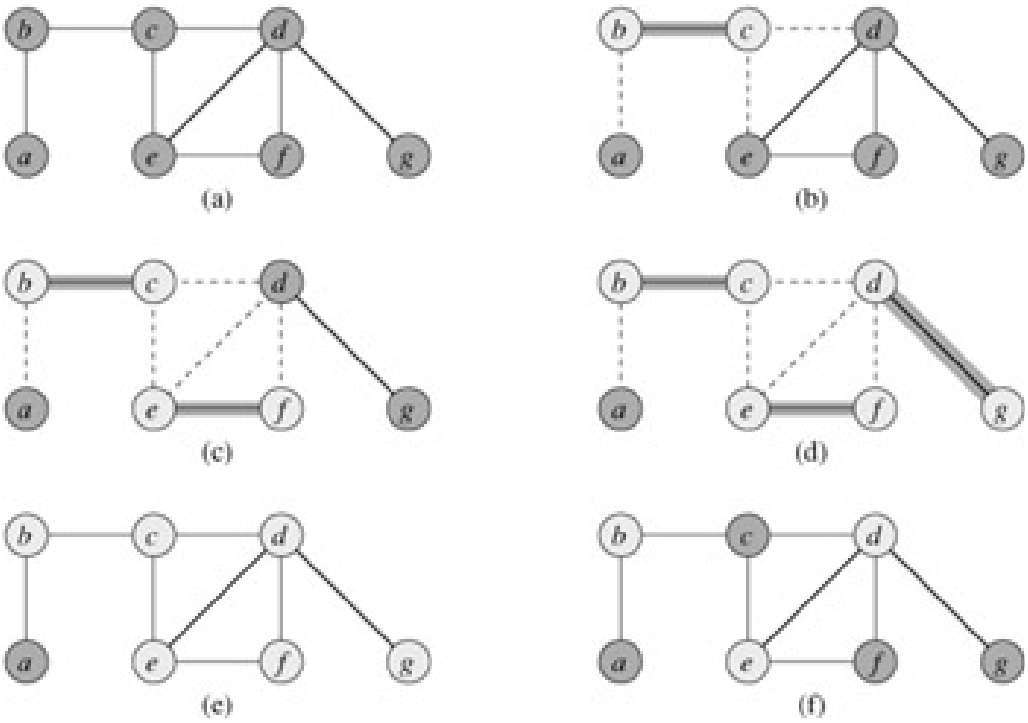


Figure: The operation of APPROX-VERTEX-COVER. *(a)* The input graph *G*, which has 7 vertices and 8 edges. *(b)* The edge (*b, c*), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices *b* and *c*, shown lightly shaded, are added to the set *C* containing the vertex cover being created. Edges (*a, b*), (*c, e*), and (*c, d*), shown dashed, are removed since they are now covered by some vertex in *C*. *(c)* Edge (*e, f*) is chosen; vertices *e* and *f* are added to *C*. *(d)* Edge (*d, g*) is chosen; vertices *d* and *g* are added to *C*. *(e)* The set *C*, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices *b, c, d, e, f, g*. *(f)* The optimal vertex cover for this problem contains only three vertices: *b, d*, and *e*.

5.5 Randomized algorithms

A **randomized algorithm** is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables.

One has to distinguish between algorithms that use the random input to reduce the expected running time or memory usage, but always terminate with a correct result (**Las Vegas algorithms**) in a bounded amount of time, and **probabilistic algorithms**, which, depending on the random input, have a chance of producing an incorrect result (**Monte Carlo algorithms**) or fail to produce a result either by signaling a failure or failing to terminate.

5.5.1 A randomized version of quicksort

Instead of always using $A[r]$ as the pivot, we will use a randomly chosen element from the subarray $A[p \_ r]$. We do so by exchanging element $A[r]$ with an element chosen at random from $A[p \_ r]$. This modification, in which we randomly sample the range $p,...,r$, ensures that the pivot element $x = A[r]$ is equally likely to be any of the $r - p + 1$ elements in the subarray. Because the pivot element is randomly chosen, we expect the split of the input array to be reasonably well balanced on average.

The changes to PARTITION and QUICKSORT are small. In the new partition procedure, we simply implement the swap before actually partitioning:

RANDOMIZED-PARTITION(*A*, *p*, *r*)

1 $i \leftarrow$ RANDOM(p, r)

2 exchange $A[r] \leftrightarrow A[i]$

3 **return** PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(*A*, *p*, *r*)

1 **if** $p < r$

2 **then** $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

3 RANDOMIZED-QUICKSORT(A, p, q - 1)

4 RANDOMIZED-QUICKSORT(A, q + 1, r)

Analysis of Randomized Quicksort

We are discussing here worst and best case complexity of randomized quicksort algorithm

Worst case complexity

Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size n. We have the recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n),$$

where the parameter q ranges from 0 to n - 1 because the procedure PARTITION produces two subproblems with total size n - 1. We guess that $T(n) \le cn2$ for some constant c. Substituting this guess into recurrence. We obtain

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n - q - 1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n - q - 1)^2) + \Theta(n). \end{aligned}$$

The expression $q2 +(n-q-1)2$ achieves a maximum over the parameter's range $0 \le q \le n$ - 1 at either endpoint, as can be seen since the second derivative of the expression with respect to q is positive. Continuing with the recurrence we obtain

$T(n) \leq cn^2 - c(2n - 1) + \Theta(n)$

$\leq cn^2$

since we can pick the constant c large enough so that the $c(2n - 1)$ term dominates the $\Theta(n)$term. Thus, $T(n) = O(n^2)$.

Expected running time

If, in each level of recursion, the split induced by RANDOMIZED-PARTITION puts any constant fraction of the elements on one side of the partition, then the recursion tree has depth $\Theta(\lg n)$, and $O(n)$ work is performed at each level. Even if we add new levels with the most unbalanced split possible between these levels, the total time remains O (n lg n).