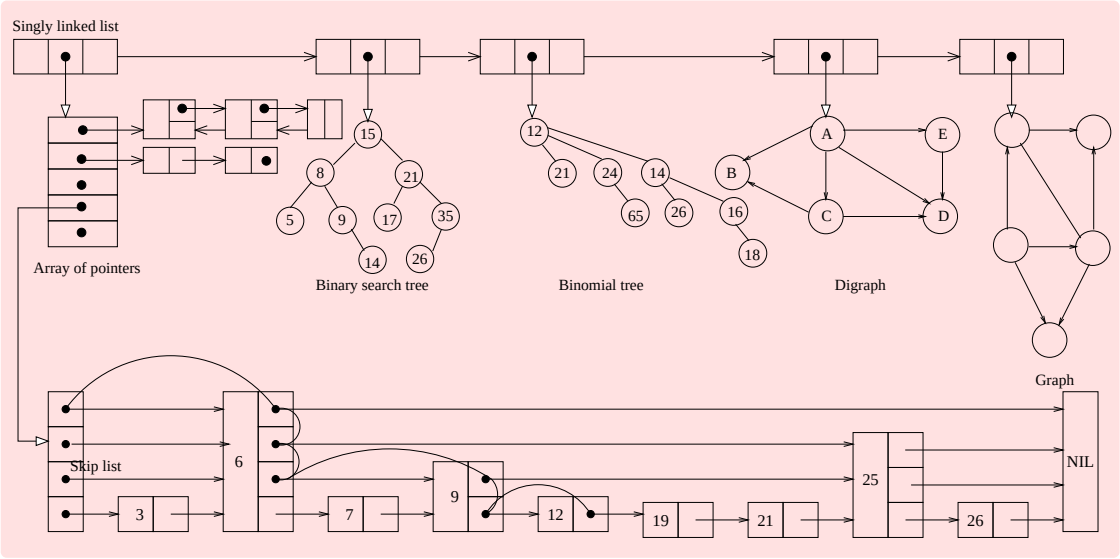
Electronic Lecture Notes

DATA STRUCTURES AND ALGORITHMS



Y. NARAHARI
Computer Science and Automation
Indian Institute of Science
Bangalore - 560 012

August 2000

Preface

As a subject, *Data Structures and Algorithms* has always fascinated me and it was a pleasure teaching this course to the Master's students at the Indian Institute of Science for six times in the last seven years. While teaching this course, I had prepared some notes, designed some programming assignments, and compiled what I thought were interesting problems. In April-95, Professor Srikant and I offered a course on Data Structures to the ITI engineers and followed it up with a more intensive AICTE-sponsored course on Object Oriented Programming and Data Structures in November-95. On both these occasions, I had prepared some lecture notes and course material. There was then a desire to put together a complete set of notes and I even toyed with the idea of writing the ideal book on DSA for Indian university students (later abandoned). Finally, encouraged by Professor Mrutyunjaya, past Chairman, CCE, Professor Pandyan, current Chairman-CCE, and Prof. K.R. Ramakrishnan (QIP co-ordinator, CCE), and supported by some CCE funding, I embarked on the project of preparing a Web-enabled Lecture Notes.

At the outset, let me say this does not pretend to be a textbook on the subject, though it does have some ingredients like examples, problems, proofs, programming assignments, etc. The lecture notes offers an adequate exposure at theoretical and practical level to important data structures and algorithms. It is safe to say the level of contents will lie somewhere between an undergraduate course in Data Structures and a graduate course in Algorithms. Since I have taught these topics to M.E. students with a non-CS background, I believe the lecture notes is at that level. By implication, this lecture notes will be suitable for second year or third year B.E./B. Tech students of Computer Science and for second year M.C.A. students. It is also useful to working software professionals and serious programmers to gain a sound understanding of commonly used data structures and algorithm design techniques. Familiarity with C programming is assumed from all readers.

OUTLINE

The Lecture Notes is organized into eleven chapters. Besides the subject matter, each chapter includes a list of problems and a list of programming projects. Also, each chapter concludes with a list of references for further reading and exploration of the subject.

Most of the material (including some figures, examples, and problems) is either sourced or adapted from classical textbooks on the subject. However about 10 percent of the material has been presented in a way different from any of the available sources. The primary sources include the following six textbooks.

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.

2. Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996. Indian Edition published by Prentice Hall of India, 1998.

3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts, 1990.

4. D. E. Knuth. *Fundamental Algorithms (The Art of Computer Programming: Volume 1)*. Second Edition, Narosa Publishing House, New Delhi, 1985.

5. R. L. Kruse. *Data Structures and Program Design in C*. Prentice Hall of India, New Delhi, 1994.

6. A. Weiss. *Data Structures and Algorithms in C++*. Addison-Wesley, Reading, Massachusetts, 1994.

Two topics that have been covered implicitly rather than in the form of independent chapters are: Algorithm Analysis Techniques (such as recurrence relations) and Algorithm Design Techniques (such as greedy strategy, dynamic programming, divide and conquer, backtracking, local search, etc.). Two topics which have not been covered adequately are: Memory management Techniques and Garbage Collection, and Secondary Storage Algorithms. Pointers to excellent sources for these topics are provided at appropriate places.

ACKNOWLEDGEMENTS

I should first acknowledge six generations of students at IISc who went through the course and gave valuable inputs. Some of them even solved and latexed the solutions of many problems. The following students have enthusiastically and uncomplainingly supported me as teaching assistants. They certainly deserve a special mention:

Jan-Apr 1992	R.Venugopal, Somyabrata Bhattacharya
Aug-Dec 1993	R. Venugopal, G. Phanendra Babu
Aug-Dec 1994	S.R. Prakash, Rajalakshmi Iyer, N.S. Narayana Swamy, L.M. Khan
Aug-Dec 1995	S.R. Prakash, N. Gokulmuthu, V.S. Anil Kumar, G. Suthindran, K.S. Raghunath
Aug-Dec 1998	Manimaran, Ashes Ganguly, Arun, Rileen Sinha, Dhiman Ghosh
Aug-Dec 1999	M. Bharat Kumar, R. Sai Anand, K. Sriram, Chintan Amrit

My special thanks to Professor N. Viswanadham for his encouragement. I shall like to thank Dr. Ashok Subramanian for clarifying many technical subtleties in the subject at various points. Many thanks to Professors V.V.S. Sarma, U.R. Prasad, V. Rajaraman, D.K. Subramanian, Y.N. Srikant, Priti Shankar, S.V. Rangaswamy, M. Narasimha Murty, C.E. Veni Madhavan, and Vijay Chandru for their encouragement. Thanks also to Dr. B. Shekar for his interest. Special thanks to Professor Kruse for sending me all the material he could on this subject.

More than 200 problems have been listed as exercises in the individual chapters of this lecture notes. Many of these problems have been freely borrowed or adapted from various sources, including the textbooks listed above and the question papers set by colleagues. I would like to acknowledge the help received in this respect.

The Latexing of this document was done near flawlessly by Renugopal first and then by Mrs Mary. The figures were done with good care by Amit Garde, Arghya Mukherjee, Mrs Mary, and Chandra Sekhar. My thanks to all of them. Numerous students at CSA have gone through drafts at various points and provided valuable feedback.

Behind any of my efforts of this kind, there are two personalities whose blessings form the inspirational force. The first is my divine Mother, who is no more but whose powerful personality continues to be a divine driving force. The second is my revered Father who is the light of my life. He continues to guide me like a beacon. Of course, I simply cannot forget the love and affection of Padmasri and Naganand and all members of my extended family, which is like a distributed yet tightly coupled enterprise.

There are bound to be numerous typographical/logical/grammatical/stylistic errors in this the first draft. I urge the readers to unearth as many as possible and to intimate to me on the email (hari@csa.iisc.ernet.in). Any suggestions/comments/criticism on any aspect of this lecture notes are most welcome.

Y. NARAHARI

Electronic Enterprises Laboratory
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

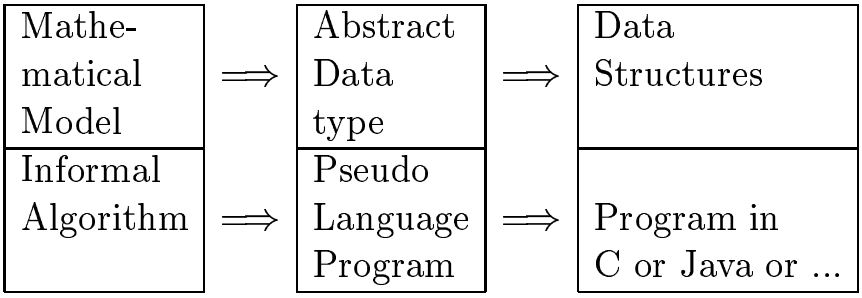
Contents

List of Figures

Chapter 1

Introduction

"Data Structures and Algorithms" is one of the classic, core topics of Computer Science. Data structures and algorithms are central to the development of good quality computer programs. Their role is brought out clearly in the following diagram (Aho, Hopcroft, and Ullman (1983)).



The Problem Solving Process in Computer Science

1.1 Some Definitions

We provide below informal definitions of a few important, common notions that we will frequently use in this lecture notes.

DEF. **Algorithm**. A finite sequence of instructions, each of which has a clear meaning and can be executed with a finite amount of effort in

finite time.

- whatever the input values, an algorithm will definitely terminate after executing a finite number of instructions.

DEF. Data Type. Data type of a variable is the set of values that the variable may assume.

Basic data types in C :

- **int char float double**

Basic data types in Pascal:

- **integer real char boolean**

DEF. Abstract Data Type (ADT): An ADT is a set of elements with a collection of well defined operations.

  - The operations can take as operands not only instances of the ADT but other types of operands or instances of other ADTs.
  - Similarly results need not be instances of the ADT
  - At least one operand or the result is of the ADT type in question.

Object-oriented languages such as C++ and Java provide explicit support for expressing ADTs by means of *classes*.

Examples of ADTs include list, stack, queue, set, tree, graph, etc.

DEF. Data Structures: An implementation of an ADT is a translation into statements of a programming language,

  - the declarations that define a variable to be of that ADT type
  - the operations defined on the ADT (using procedures of the programming language)

An ADT implementation chooses a *data structure* to represent the ADT. Each data structure is built up from the basic data types of the underlying programming language using the available data structuring facilities, such as arrays, records (structures in C), pointers, files, sets, etc.

Example: A "Queue" is an ADT which can be defined as a sequence of elements with operations such as null(Q), empty(Q), enqueue(x, Q), and dequeue(Q). This can be implemented using data structures such as

- array
- singly linked list
- doubly linked list
- circular array

1.1.1 Four Fundamental Data Structures

The following four data structures are used ubiquitously in the description of algorithms and serve as basic building blocks for realizing more complex data structures.

- Sequences (also called as lists)
- Dictionaries
- Priority Queues
- Graphs

Dictionaries and priority queues can be classified under a broader category called *dynamic sets*. Also, binary and general trees are very popular building blocks for implementing dictionaries and priority queues.

1.2 Complexity of Algorithms

It is very convenient to classify algorithms based on the relative amount of time or relative amount of space they require and specify the growth of

time /space requirements as a function of the input size. Thus, we have the notions of:

- **Time Complexity**: Running time of the program as a function of the size of input
- **Space Complexity**: Amount of computer memory required during the program execution, as a function of the input size

1.2.1 Big Oh Notation

- A convenient way of describing the growth rate of a function and hence the time complexity of an algorithm.

Let n be the size of the input and $f(n)$, $g(n)$ be positive functions of n.

DEF. Big Oh. $f(n)$ is $O(g(n))$ if and only if there exists a real, positive constant C and a positive integer n_0 such that

$$f(n) \leq Cg(n) \quad \forall \quad n \geq n_0$$

- Note that $O(g(n))$ is a class of functions.
- The "Oh" notation specifies asymptotic upper bounds
- $O(1)$ refers to constant time. $O(n)$ indicates linear time; $O(n^k)$ (k fixed) refers to polynomial time; $O(\log n)$ is called logarithmic time; $O(2^n)$ refers to exponential time, etc.

1.2.2 Examples

- Let f(n) = n^2 + n + 5. Then
  - f(n) is O(n^2)
  - f(n) is O(n^3)
  - f(n) is not O(n)
- Let f(n) = 3^n

  - f(n) is $O(4^n)$
  - f(n) is not $O(2^n)$

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then
  - $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$

1.2.3 An Example: Complexity of Mergesort

Mergesort is a divide and conquer algorithm, as outlined below. Note that the function *mergesort* calls itself *recursively*. Let us try to determine the time complexity of this algorithm.

```
list mergesort (list L, int n);
  {
    if (n = = 1)
      return (L)
    else {
    Split L into two halves L1 and L2 ;
    return (merge (mergesort (L1, n/2), (mergesort (L2, n/2))
    }
  }
```

Let T(n) be the running time of Mergesort on an input list of size n. Then,

$$\begin{aligned} T(n) &\leq C_1 \quad (\text{if } n = 1) \quad (C_1 \text{ is a constant}) \\ &\leq \underbrace{2\,T\left(\frac{n}{2}\right)}_{\text{two recursive calls}} + \underbrace{C_2 n}_{\text{cost of merging}} \quad (\text{if } n > 1) \end{aligned}$$

If $n = 2^k$ for some k, it can be shown that

$$T(n) \leq 2^k T(1) + C_2 k 2^k$$

That is, $T(n)$ is $O(n \log n)$.

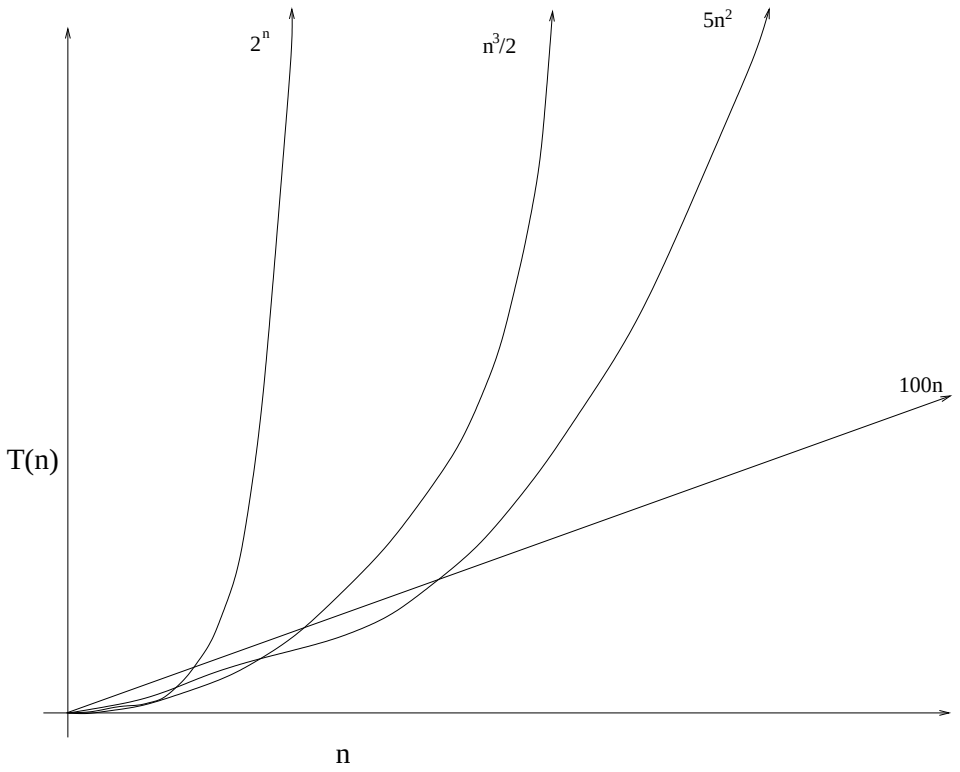


Figure 1.1: Growth rates of some functions

T(n)	Maximum Problem Size that can be solved in		
	100 Time Units	1000 Time Units	10000 Time Units
100 n	1	10	100
$5\ n^2$	5	14	45
$n^3/2$	7	12	27
2^n	8	10	13

Table 1.1: Growth rate of functions

1.2.4 Role of the Constant

The constant C that appears in the definition of the asymptotic upper bounds is very important. It depends on the algorithm, machine, compiler, etc. It is to be noted that the big "Oh" notation gives only asymptotic complexity. As such, a polynomial time algorithm with a large value of the constant may turn out to be much less efficient than an exponential time algorithm (with a small constant) for the range of interest of the input values. See Figure 1.1 and also Table 1.1.

1.2.5 Worst Case, Average Case, and Amortized Complexity

- **Worst case Running Time**: The behavior of the algorithm with respect to the worst possible case of the input instance. The worst-case running time of an algorithm is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer. There is no need to make an educated guess about the running time.
- **Average case Running Time**: The expected behavior when the input is randomly drawn from a given distribution. The average-case running time of an algorithm is an estimate of the running time for an "average" input. Computation of average-case running time entails knowing all possible input sequences, the probability distribution of occurrence of these sequences, and the running times for the individual sequences. Often it is assumed that all inputs of a given size are equally likely.
- **Amortized Running Time** Here the time required to perform a sequence of (related) operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a simple operation might be expensive. Amortized analysis guarantees the average performance of each operation in the worst case.

1. For example, consider the problem of finding the minimum element in a list of elements.

   Worst case = O(n)

   Average case = O(n)

2. Quick sort

   Worst case = $O(n^2)$

   Average case = O(n log n)

3. Merge Sort, Heap Sort

   Worst case = O(n log n)

   Average case = O(n log n)

4. Bubble sort

   Worst case = $O(n^2)$

   Average case = $O(n^2)$

5. Binary Search Tree: Search for an element

   Worst case = O(n)

   Average case = O(log n)

1.2.6 Big Omega and Big Theta Notations

The Ω notation specifies asymptotic lower bounds.

DEF. Big Omega. $f(n)$ is said to be $\Omega(g(n))$ if $\exists$ a positive real constant C and a positive integer n_0 such that

$$f(n) \geq Cg(n) \quad \forall \quad n \geq n_0$$

An Alternative Definition : $f(n)$ is said to be $\Omega(g(n))$ iff $\exists$ a positive real constant C such that

$$f(n) \geq Cg(n) \text{ for infinitely many values of } n.$$

The Θ notation describes asymptotic tight bounds.

DEF. Big Theta. $f(n)$ is $\Theta(g(n))$ iff $\exists$ positive real constants C_1 and C_2 and a positive integer n_0, such that

$$C_1 g(n) \leq f(n) \leq C_2 g(n) \quad \forall\ n \geq n_0$$

1.2.7 An Example:

Let $f(n) = 2n^2 + 4n + 10$. $f(n)$ is $O(n^2)$. For,

$f(n) \leq 3n^2 \ \ \forall \ \ n \geq 6$

Thus, C = 3 and $n_0 = 6$

Also,

$f(n) \leq 4n^2 \ \ \forall \ \ n \geq 4$

Thus, $C = 4$ and $n_0 = 4$

$f(n)$ is $O(n^3)$

In fact, if $f(n)$ is $O(n^k)$ for some k, it is $O(n^h)$ for $h > k$

$f(n)$ is not $O(n)$.

Suppose $\exists$ a constant C such that

$$2n^2 + 4n + 10 \leq Cn \ \ \forall n \geq n_0$$

This can be easily seen to lead to a contradiction. Thus, we have that:

$f(n)$ is $\Omega(n^2)$ and $f(n)$ is $\Theta(n^2)$

1.3 To Probe Further

The following books provide an excellent treatment of the topics discussed in this chapter. The readers should also study two other key topics: (1) Recursion; (2) Recurrence Relations, from these sources.

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

2. Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996. Indian Edition published by Prentice Hall of India, 1998.

3. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.

4. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.

5. R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-wesley, Reading, 1990. Indian Edition published by Addison-Wesley Longman, 1998.

1.4 Problems

1. Assuming that n is a power of 2, express the output of the following program in terms of n.

```
int mystery (int n)
{ int x=2, count=0;
  while (x < n) { x *=2; count++}
  return count;
}
```

2. Show that the following statements are true:

   (a) $\frac{n(n-1)}{2}$ is $O(n^2)$

   (b) $\max(n^3, 10n^2)$ is $O(n^3)$

   (c) $\sum_{i=1}^{n} i^k$ is $O(n^{k+1})$ and $\Omega(n^{k+1})$, for integer k

   (d) If $p(x)$ is any k^{th} degree polynomial with a positive leading coefficient, then $p(n)$ is $O(n^k)$ and $\Omega(n^k)$.

3. Which function grows faster?

   (a) $n^{\log n}$; $(\log n)^n$

(b) $\log n^k$; $(\log n)^k$

(c) $n^{\log\log\log n}$; $(\log n)!$

(d) n^n; $n!$.

4. If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ where f_1 and f_2 are positive functions of n, show that the function $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$.

5. If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ where f_1 and f_2 are positive functions of n, state whether each statement below is true or false. If the statement is true(false), give a proof(counter-example).

   (a) The function $|f_1(n) - f_2(n)|$ is $O(\min(g_1(n), g_2(n)))$.

   (b) The function $|f_1(n) - f_2(n)|$ is $O(\max(g_1(n), g_2(n)))$.

6. Prove or disprove: If $f(n)$ is a positive function of n, then $f(n)$ is $O(f(\frac{n}{2}))$.

7. The running times of an algorithm A and a competing algorithm A' are described by the recurrences

$$T(n) = 3T(\frac{n}{2}) + n; \quad T'(n) = aT'(\frac{n}{4}) + n$$

   respectively. Assuming $T(1) = T'(1) = 1$, and $n = 4^k$ for some positive integer k, determine the values of a for which A' is asymptotically faster than A.

8. Solve the following recurrences, where $T(1) = 1$ and $T(n)$ for $n \geq 2$ satisfies:

   (a) $T(n) = 3T(\frac{n}{2}) + n$

   (b) $T(n) = 3T(\frac{n}{2}) + n^2$

   (c) $T(n) = 3T(\frac{n}{2}) + n^3$

   (d) $T(n) = 4T(\frac{n}{3}) + n$

   (e) $T(n) = 4T(\frac{n}{3}) + n^2$

   (f) $T(n) = 4T(\frac{n}{3}) + n^3$

   (g) $T(n) = T(\frac{n}{2}) + 1$

   (h) $T(n) = 2T(\frac{n}{2}) + \log n$

   (i) $T(n) = 2T(\frac{n}{2}) + n^2$

   (j) $T(n) = 2T(n-1) + 1$

   (k) $T(n) = 2T(n-1) + n$

9. Show that the function $T(n)$ defined by $T(1) = 1$ and

$$T(n) = T(n-1) + \frac{1}{n}$$

   for $n \geq 2$ has the complexity $O(\log n)$.

10. Prove or disprove: f(3**n) is O(f(2**n))

11. Solve the following recurrence relation:

$$T(n) \leq cn + \frac{2}{n-1} \sum_{i=1}^{n-1} T(i)$$

where c is a constant, $n \geq 2$, and $T(2)$ is known to be a constant c_1.

Chapter 2

Lists

A *list*, also called a *sequence*, is a container that stores elements in a certain linear order, which is imposed by the operations performed. The basic operations supported are retrieving, inserting, and removing an element given its position. Special types of lists include *stacks* and *queues*, where insertions and deletions can be done only at the head or the tail of the sequence. The basic realization of sequences is by means of arrays and linked lists.

2.1 List Abstract Data Type

A list is a sequence of zero or more elements of a given type

$$a_1, a_2, \ldots, a_n \quad (n \geq 0)$$

- n : length of the list
- a_1 : first element of the list
- a_n : last element of the list
- $n = 0$: empty list
- elements can be linearly ordered according to their position in the list

We say a_i precedes a_{i+1}, a_{i+1} follows a_i, and a_i is at position i

Let us assume the following:

L	:	list of objects of type element type
x	:	an object of this type
p	:	of type position
END(L)	:	a function that returns the position following the last position in the list L

Define the following operations:

1. Insert (x, p, L)
   - Insert x at position p in list L
   - If $p = END(L)$, insert x at the end
   - If L does not have position p, result is undefined
2. Locate (x, L)
   - returns position of x on L
   - returns END(L) if x does not appear
3. Retrieve (p, L)
   - returns element at position p on L
   - undefined if p does not exist or $p = END(L)$
4. Delete (p, L)
   - delete element at position p in L
   - undefined if $p = END(L)$ or does not exist
5. Next (p, L)
   - returns the position immediately following position p
6. Prev (p, L)
   - returns the position previous to p
7. Makenull (L)
   - causes L to become an empty list and returns position $END(L)$

8. First (L)
   - returns the first position on L
9. Printlist (L)
   - print the elements of L in order of occurrence

2.1.1 A Program with List ADT

A program given below is independent of which data structure is used to implement the list ADT. This is an example of the notion of *encapsulation* in object oriented programming.

Purpose:

To eliminate all duplicates from a list

Given:

1. Elements of list L are of element type
2. function same (x, y), where x and y are of element type, returns **true** if x and y are same, **false** otherwise
3. p and q are of type position

   p : current position in L

   q : moves ahead to find equal elements

Pseudocode in C:

```
{
   p = first (L) ;
     while (p! = end(L)) {
      q = next(p, L) ;
      while (q! = end(L)) {
        if (same (retrieve (p,L), retrieve (q, L))))
```

```
        delete (q,L);
      else
        q = next (q, L) ;
    }
   p = next (p, L) ;
   }
}
```

2.2 Implementation of Lists

Many implementations are possible. Popular ones include:

1. Arrays
2. Pointers (singly linked, doubly linked, etc.)
3. Cursors (arrays with integer pointers)

2.2.1 Array Implementation of Lists

- Here, elements of list are stored in the (contiguous) cells of an array.
- List is a structure with two members.
  member 1 : an array of elements
  member 2 : **last** — indicates position of the last element of the list

Position is of type integer and has the range 0 to maxlength–1

```
# define maxlength 1000
typedef int   elementtype; /* elements are integers */
typedef struct  list–tag {
          elementtype  elements [maxlength];
            int   last;
   } list–type;
```

end(L)

```
int end (list–type *ℓp)
{
  return (ℓp → last + 1)
}
```

Insert (x, p,L)

```
void insert (elementtype x ; int p ; list–type *ℓp) ;
    {
     int v;  /* running position */
    if (ℓp → last >= maxlength–1)
        error ("list is full")
    elseif  ((p < 0) || (p > ℓp → last + 1))
        error (position does not exist)
        else
        for (q = ℓp → last ; q <= p, q−−)
          ℓp → elements [q + 1] = ℓp → elements [q] ;
        ℓp → last = ℓp → last + 1 ;
        ℓp → elements [p] = x
    }
```

Delete (p, L)

```
void  delete (int p ; list–type *ℓp)
    {
        int q ;    /* running position */
        if ((p > ℓp → last) || (p < 0))
          error ("position does not exist")
        else    /* shift elements */ {
          ℓp → last −− ;
          for (q = p ; q <= ℓp → last; q ++)
            ℓp → elements [q] = ℓp → elements [q+1]
        }
    }
```

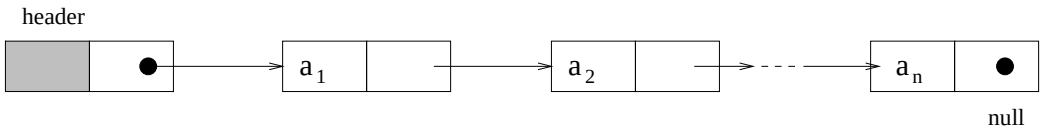


Figure 2.1: A singly linked list

Locate (x, L)

```
int  locate (element type *x ; list–type *ℓp)
     {
       int q ;
       for (q = 0 ; q <= ℓp → last ; q++)
         if (ℓp → elements [q] = = x]
           return (q) ;
       return (ℓp → last + 1) /* if not found */
     }
```

2.2.2 Pointer Implementation of Lists

- In the array implementation,
  1. we are constrained to use contiguous space in the memory
  2. Insertion, deletion entail shifting the elements
- Pointers overcome the above limitations at the cost of extra space for pointers.
- Singly Linked List Implementation

  A list $a_1, a_2, \ldots, a_n$ is organized as shown in Figure 2.1
- Let us follow a convention that position i is a pointer to the cell holding the pointer to the cell containing a_i, (for $i = 1, 2, \ldots, n$). Thus,
  - Position 1 is a pointer to the header

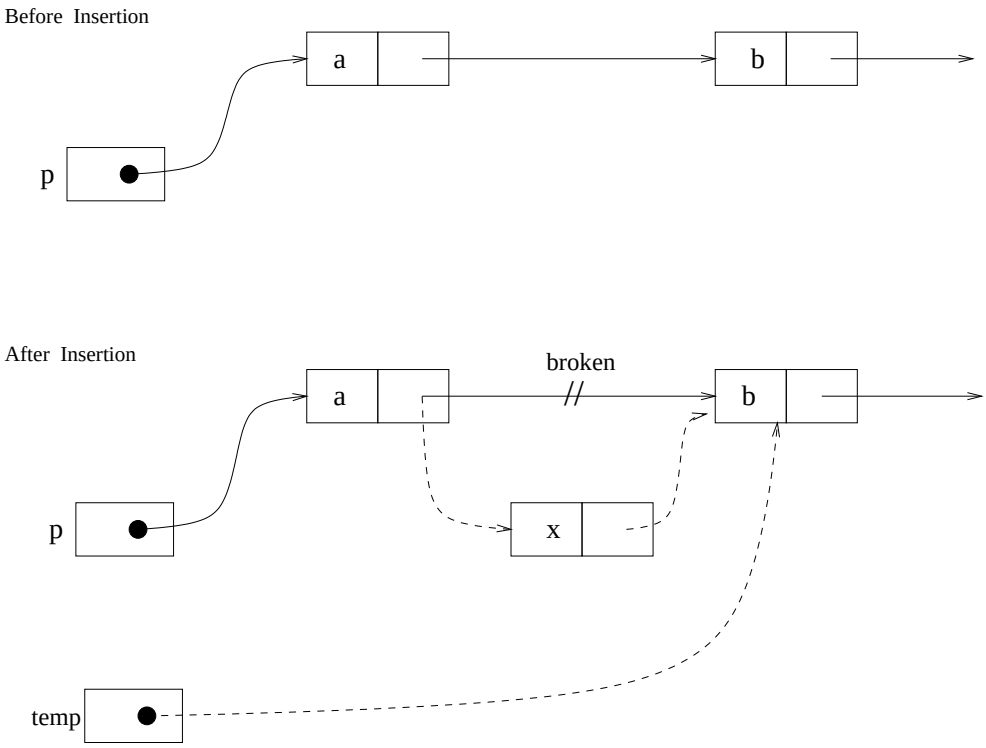


Figure 2.2: Insertion in a singly linked list

  - End (L) is a pointer to the last cell of list L

- If position of a_i is simply a pointer to the cell holding a_i, then
  - Position 1 will be the address in the header
  - end (L) will be a null pointer
- Insert (x, p, L) : See Figure 2.2
- Delete (x, L) : See Figure 2.3

2.2.3 Doubly Linked List Implementation

* makes searches twice as efficient
* needs as many extra pointers as the number of elements (See Figure 2.4); consequently insertions and deletions are more expensive in terms of pointer assignments

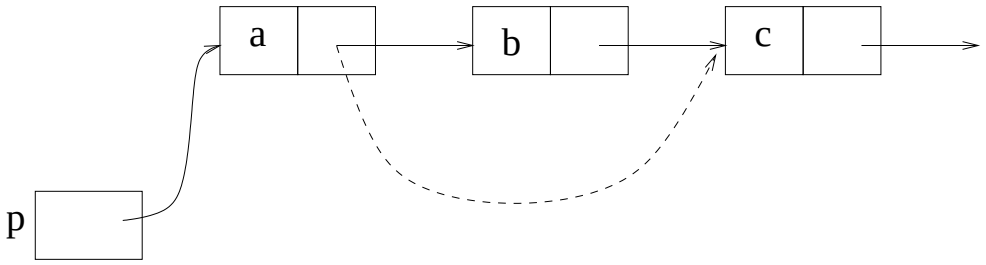


Figure 2.3: Deletion in a singly linked list

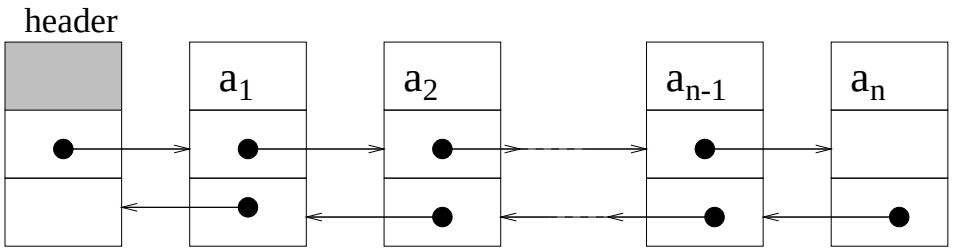


Figure 2.4: A doubly linked list

2.3 Stacks

- Stack is a special kind of list in which all insertions and deletions occur at one end, called the **top**.
- Stack ADT is a special case of the List ADT. It is also called as a LIFO list or a pushdown list.
- Typical Stack ADT Operations:
  1. makenull (S) creates an empty stack
  2. top (S) returns the element at the top of the stack. Same as retrieve (first (S), S)
  3. pop (S) deletes the top element of the stack Same as deletes (first (S), S)
  4. push (x, S) Insert element x at the top of stack S. Same as Inserts (x, first (S), S)
  5. empty (S) returns **true** if S is empty and **false** otherwise
- Stack is a natural data structure to implement subroutine or procedure calls and recursion.
- Stack Implementation : Arrays, Pointers can be used. See Figures 2.5

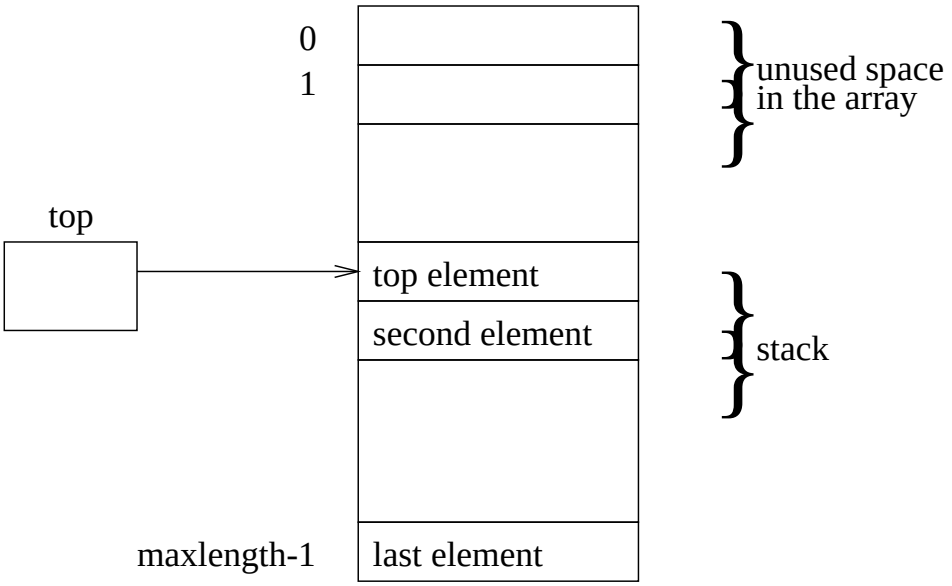


Figure 2.5: An array implementation for the stack ADT

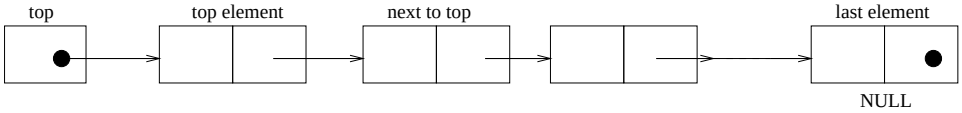


Figure 2.6: A linked list implementation of the stack ADT

and 2.6

- Pointer Implementation of Stacks: The following code provides functions for implementation of stack operations using pointers. See Figures 2.7 and 2.8 for an illustration of push and pop operations on a linked stack.

```
typedef struct node-tag {
        item-type    info ;
        struct    node-tag  * next ;
    } node-type ;

typedef struct stack-tag {
        node-type  * top ;
    } stack-type ;
stack-type   stack ;   /* define a stack */
```

```
    stack-type   * sp = & stack ;   /* pointer to stack */
    node-type   *np ;   /* pointer to a node */

/* makenode allocates enough space for a new node and initializes it */
    node-type * makenode (item-type   item)
      {
        node-type  *p ;
        if ((p = (node-type *) malloc (sizeof
                    (node-type))) = = null)
          error ("exhausted memory") ;
        else  {
            p → info = item ;
            p → next = null ;
       }
       return (p) ;
    }
```

```
    /* pushnode pushes a node onto the top of the linked stack */
      void  pushnode (node-type  *np, stack-type  *sp)
    {
       if (np = = null)
         error ("attempt to push a nonexistent node")
       else  {
           np → next = sp → top ;
           sp → top = np
       }
    }


    void popnode (node-type **np ; stack-type *sp)
      {
        if (sp → top = = null)
          error ("empty stack") ;
        else {
          *np = sp → top ;
          sp → top = (* np) → next ;
          }
      }
```

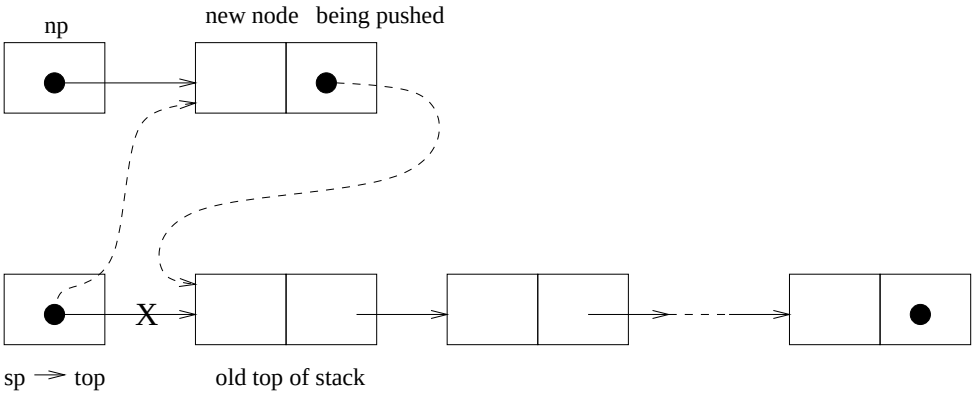


Figure 2.7: Push operation in a linked stack

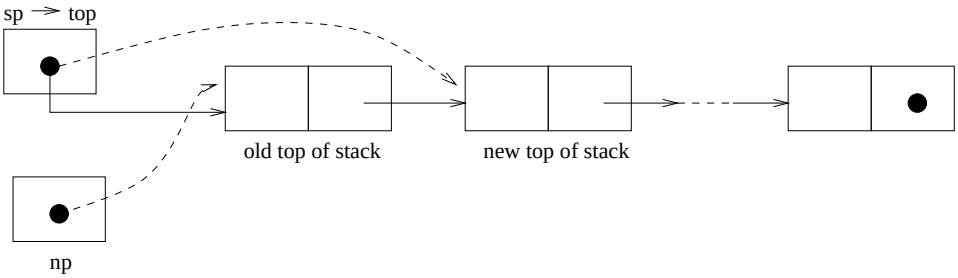


Figure 2.8: Pop operation on a linked stack

```
/* push-make a new node with item and push it onto stack */
    void push (item-type  item ; stack-type  *sp)
    {
      pushnode (makenode (item), sp) ;
    }
  /* pop–pop a node from the stack and return its item */
    void  pop (item-type * item, stack-type *sp)
      {
        node-type * np ;
        popnode (& np, sp) ;
        * item = np → info ;
        free (np) ;
      }
```

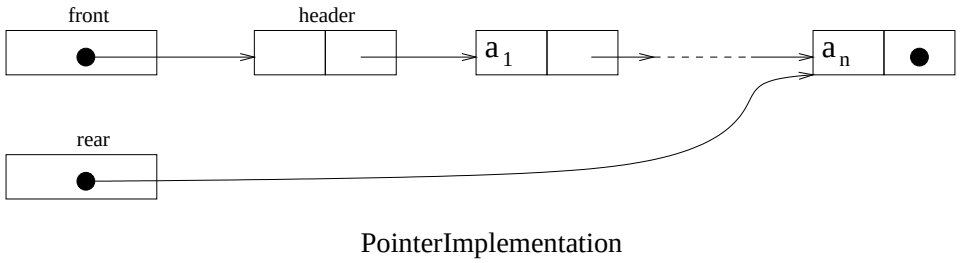


Figure 2.9: Queue implementation using pointers

2.4 Queues

- A queue is a special kind of a list in which all items are inserted at one end (called the **rear** or the **back** or the **tail**) and deleted at the other end (called the **front** or the **head**)
- useful in
  - simulation
  - breadth-first search in graphs
  - tree and graph algorithms
- The Queue ADT is a special case of the List ADT, with the following typical operations
  1. makenull (Q)
  2. front (Q) ≡ retrieve (first (Q), Q)
  3. enqueue (x, Q) ≡ insert (x, end(Q), Q)
  4. dequeue (Q) ≡ delete (first (Q), Q)
  5. empty (Q)
- Implementation : Pointers, Circular array, Circular linked list

2.4.1 Pointer Implementation

See Figure 2.9.

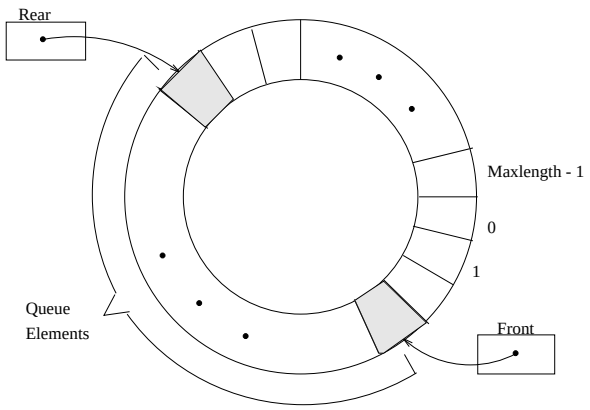


Figure 2.10: Circular array implementation of a queue

2.4.2 Circular Array Implementation

See Figure 2.10.

- Rear of the queue is somewhere clockwise from the front
- To enqueue an element, we move rear one position clockwise and write the element in that position
- To dequeue, we simply move front one position clockwise
- Queue migrates in a clockwise direction as we enqueue and dequeue
- emptiness and fullness to be checked carefully.

2.4.3 Circular Linked List Implementation

- A linked list in which the node at the tail of the list, instead of having a null pointer, points back to the node at the head of the list. Thus both ends of a list can be accessed using a single pointer.

  See Figure 2.11.

- If we implement a queue as a circularly linked list, then we need only one pointer namely tail, to locate both the front and the back.

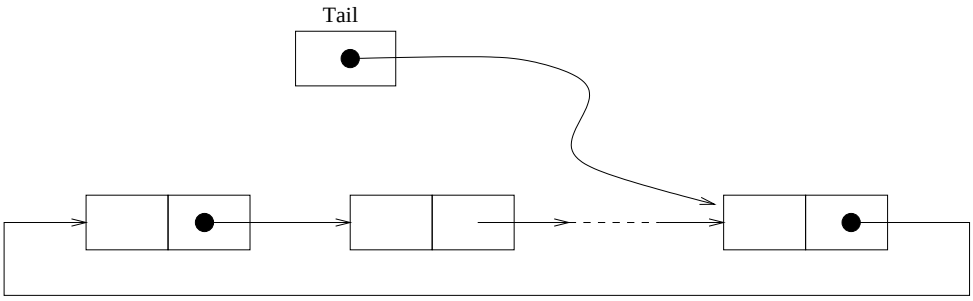


Figure 2.11: Circular linked list implementation of a queue

2.5 To Probe Further

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

2. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.

3. Robert L. Kruse, Bruce P. Leung, and Clovis L. Tondo. *Data Structures and Program design in C*. Prentice Hall, 1991. Indian Edition published by Prentice Hall of India, 1999.

4. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.

5. Duane A. Bailey. *Java Structures: Data Structures in Java for the Principled Programmer*. McGraw-Hill International Edition, 1999.

6. Ellis Horowitz and Sartaz Sahni. *Fundamentals of Data structures*. Galgotia Publications, New Delhi, 1984.

7. Y. Langsam, M.J. Augenstein, and A.M. Tenenbaum. *Data Structures using C and C++*. Second Edition, Prentice Hall, 1996. Indian Edition published by Prentice Hall of India, 2000.

8. Sartaj Sahni. *Data Structures, Algorithms, and Applications in Java*. McGraw-Hill Higher Education, 2000.

9. Thomas A. Standish. *Data Structures in Java*. Addison-Wesley, 1998. Indian Edition published by Addison Wesley Longman, 2000.

2.6 Problems

1. A linked list has exactly n nodes. The elements in these nodes are selected from the set $\{0, 1, \ldots, n\}$. There are no duplicates in the list. Design an $O(n)$ worst case time algorithm to find which one of the elements from the above set is missing in the given linked list.

2. Write a procedure that will reverse a linked list while traversing it only once. At the conclusion, each node should point to the node that was previously its predecessor: the head should point to the node that was formerly at the end, and the node that was formerly first should have a null link.

3. How would one implement a queue if the elements that are to be placed on the queue are arbitrary length strings? How long does it take to enqueue a string?

4. Let A be an array of size n, containing positive or negative integers, with $A[1] < A[2] < \ldots < A[n]$. Design an efficient algorithm (should be more efficient than $O(n)$) to find an i such that $A[i] = i$ provided such an i exists. What is the worst case computational complexity of your algorithm ?

5. Consider an array of size n. Sketch an $O(n)$ algorithm to shift all items in the array k places cyclically counterclockwise. You are allowed to use only one extra location to implement swapping of items.

6. In some operating systems, the *least recently used* (LRU) algorithm is used for page replacement. The implementation of such an algorithm will involve the following operations on a collection of nodes.

   - *use* a node.
   - *replace* the LRU node by a new node.

   Suggest a good data structure for implementing such a collection of nodes.

7. A queue Q contains the items $a_1, a_2, \ldots, a_n$, in that order with a_1 at the front and a_n at the back. It is required to transfer these items on to a stack S (initially empty) so that a_1 is at the top of the stack and the order of all other items is preserved. Using enqueue and dequeue operations for the queue and push and pop operations for the stack, outline an efficient $O(n)$ algorithm to accomplish the above task, using only a constant amount of additional storage.

8. A queue is set up in a circular array $C[0..n-1]$ with *front* and *rear* defined as usual. Assume that $n-1$ locations in the array are available for storing the elements (with the other element being used to detect full/empty condition). Derive a formula for the number of elements in the queue in terms of *rear*, *front*, and n.

9. Let $p_1 p_2 \ldots p_n$ be a stack-realizable permutation of $12 \ldots n$. Show that there do not exist indices $i < j < k$ such that $p_j < p_k < p_i$.

10. Write recursive algorithms for the following problems:

    (a) Compute the number of combinations of n objects taken m at a time.

    (b) Reverse a linked list.

    (c) Reverse an array.

    (d) Binary search on an array of size n.

    (e) Compute gcd of two numbers n and m.

11. Consider the following recursive definition:

$$g(i,j) = \quad i \quad (j=1) \tag{2.1}$$
$$g(i,j) = \quad j \quad (i=1) \tag{2.2}$$
$$g(i,j) = \quad g(i-1,j) + g(i,j-1) \quad \text{else} \tag{2.3}$$

    Design an $O(mn)$ iterative algorithm to compute $g(m,n)$ where m and n are positive integers.

12. Consider polynomials of the form:

$$p(x) = c_1 x^{e_1} + c_2 x^{e_2} \ldots c_1 x^{e_n};$$

    where $e_1 > e_2 \ldots > e_n \geq 0$. Such polynomials can be represented by a linked lists in which each cell has 3 fields: one for the coefficient, one for the exponent, and one pointing to the next cell. Write procedures for

    (a) differentiating,

    (b) integrating,

    (c) adding,

    (d) multiplying

    such polynomials. What is the running time of these procedures as a function of the number of terms in the polynomials?

13. Suppose the numbers 1, 2, 3, 4, 5 and 6 arrive in an input stream in that order. Which of the following sequences can be realized as the output of (1) stack, and (2) double ended queue?

    a) 1 2 3 4 5 6

    b) 6 5 4 3 2 1

    c) 2 4 3 6 5 1

    d) 1 5 2 4 3 6

    e) 1 3 5 2 4 6

2.7 Programming Assignments

2.7.1 Sparse Matrix Package

As you might already know, sparse matrices are those in which most of the elements are zero (that is, the number of non-zero elements is very small). Linked lists are very useful in representing sparse matrices, since they eliminate the need to represent zero entries in the matrix. Implement a package that facilitates efficient addition, subtraction, multiplication, and other important arithmetic operations on sparse matrices, using linked list representation of those.

2.7.2 Polynomial Arithmetic

Polynomials involving real variables and real-valued coefficients can also be represented efficiently through linked lists. Assuming single variable polynomials, design a suitable linked list representation for polynomials and develop a package that implements various polynomial operations such as addition, subtraction, multiplication, and division.

2.7.3 Skip Lists

A skip list is an efficient linked list-based data structure that attempts to implement binary search on linked lists. Read the following paper: William Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, Volume 33, Number 6, pp. 668-676, 1990. Also read the section on Skip Lists in Chapter 4 of this lecture notes. Develop programs to implement search, insert, and delete operations in skip lists.

2.7.4 Buddy Systems of Memory Allocation

The objective of this assignment is to compare the performance of the *exponential* and the *Fibonacci* buddy systems of memory allocation. For more details on buddy systems, refer to the book: Donald E Knuth. *Fundamental Algorithms*, Volume 1 of The Art of Computer Programming, Addison-Wesley, 1968, Second Edition, 1973. Consider an exponential buddy system with the following allowable block sizes: 8, 16, 32, 64, 128, 512, 1024, 2048, and 4096. Let the allowable block sizes for the Fibonacci system be 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2594, and 4191. Your tasks are the following.

1. Generate a random sequence of allocations and liberations. Start with an empty memory and carry out a sequence of allocations first, so that the memory becomes

adequately committed. Now generate a sequence of allocations and liberations interleaved randomly. You may try with 100 or 200 or even 1000 such allocations and liberations. While generating the memory sizes requested for allocations, you may use a reasonable distribution. For example, one typical scenario is:

Range of block size	Probability
8–100:	0.1
100–500:	0.5
500–1000:	0.2
1000–2000:	0.1
2000-4191:	0.1

2. Simulate the memory allocations and liberations for the above random sequences. At various intermediate points, print out the detailed state of the memory, indicating the allocated portions, available portions, and the i-lists. Use *recursive* algorithms for allocations and liberations. It would be excellent if you implement very general allocation and liberation algorithms, that will work for any arbitrary order buddy system.

3. Compute the following performance measures for each random sequence:

   - Average fragmentation.
   - Total number of splits.
   - Total number of merges.
   - Average size of all linked lists used in maintaining the available block information.
   - Number of blocks that cannot be combined with buddies.
   - Number of contiguous blocks that are not buddies.

4. Repeat the experiments for several random sequences and print out the average performance measures. You may like to repeat the experimentation on different distributions of request sizes.

Chapter 3

Dictionaries

A *dictionary* is a container of elements from a totally ordered universe that supports the basic operations of inserting/deleting elements and searching for a given element. In this chapter, we present *hash tables* which provide an efficient implicit realization of a dictionary. Efficient explicit implementations include binary search trees and balanced search trees. These are treated in detail in Chapter 4. First, we introduce the abstract data type **Set** which includes dictionaries, priority queues, etc. as subclasses.

3.1 Sets

- A set is a collection of well defined elements. The members of a set are all different.
- A set ADT can be defined to comprise the following operations:
  1. Union (A, B, C)
  2. Intersection (A, B, C)
  3. Difference (A, B, C)
  4. Merge (A, B, C)
  5. Find (x)
  6. Member (x, A) or Search (x, A)
  7. Makenull (A)

8. Equal (A, B)
9. Assign (A, B)
10. Insert (x, A)
11. Delete (x, A)
12. Min (A) (if A is an ordered set)

- Set implementation: Possible data structures include:
  - Bit Vector
  - Array
  - Linked List
    - Unsorted
    - Sorted

3.2 Dictionaries

- A dictionary is a dynamic set ADT with the operations:
  1. Makenull (D)
  2. Insert (x, D)
  3. Delete (x, D)
  4. Search (x, D)
- Useful in implementing symbol tables, text retrieval systems, database systems, page mapping tables, etc.
- Implementation:
  1. Fixed Length arrays
  2. Linked lists : sorted, unsorted, skip-lists
  3. Hash Tables : open, closed
  4. Trees
     - Binary Search Trees (BSTs)

  - Balanced BSTs
    - AVL Trees
    - Red-Black Trees
  - Splay Trees
  - Multiway Search Trees
    - 2-3 Trees
    - B Trees
  - Tries

- Let n be the number of elements is a dictionary D. The following is a summary of the performance of some basic implementation methods:

	Worst case complexity of			
	Search	Delete	Insert	min
Array	O(n)	O(n)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(n)	O(1)
Unsorted List	O(n)	O(n)	O(n)	O(n)

  Among these, the sorted list has the best average case performance.

- In this chapter, we discuss two data structures for dictionaries, namely Hash Tables and Skip Lists.

3.3 Hash Tables

- An extremely effective and practical way of implementing dictionaries.
- O(1) time for search, insert, and delete in the **average case**.
- O(n) time in the worst case; by careful design, we can make the probability that more than constant time is required to be arbitrarily small.

- Hash Tables
  - Open or External
  - Closed or Internal

3.3.1 Open Hashing

Let:

- U be the universe of keys:
  - integers
  - character strings
  - complex bit patterns
- B the set of hash values (also called the buckets or bins). Let $B = \{0, 1, \ldots, m-1\}$ where $m > 0$ is a positive integer.

A hash function h : U $\rightarrow$ B associates buckets (hash values) to keys.

Two main issues:

1. **Collisions**

   If x_1 and x_2 are two different keys, it is possible that $h(x_1) = h(x_2)$. This is called a collision. Collision resolution is the most important issue in hash table implementations.

2. **Hash Functions**

   Choosing a hash function that minimizes the number of collisions and also hashes uniformly is another critical issue.

Collision Resolution by Chaining

- Put all the elements that hash to the same value in a linked list. See Figure 3.1.

Example:

See Figure 3.2. Consider the keys 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100. Let the hash function be:

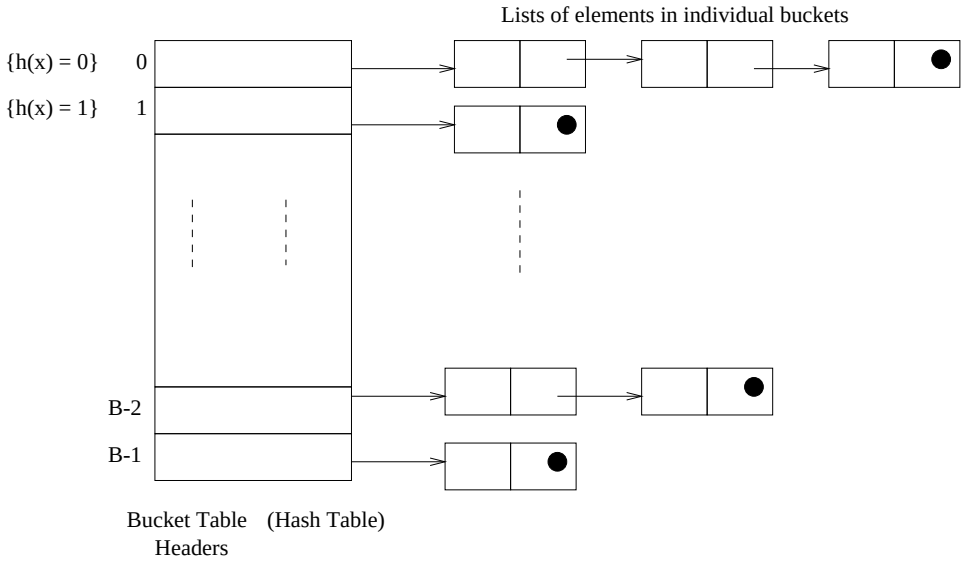


Figure 3.1: Collision resolution by chaining

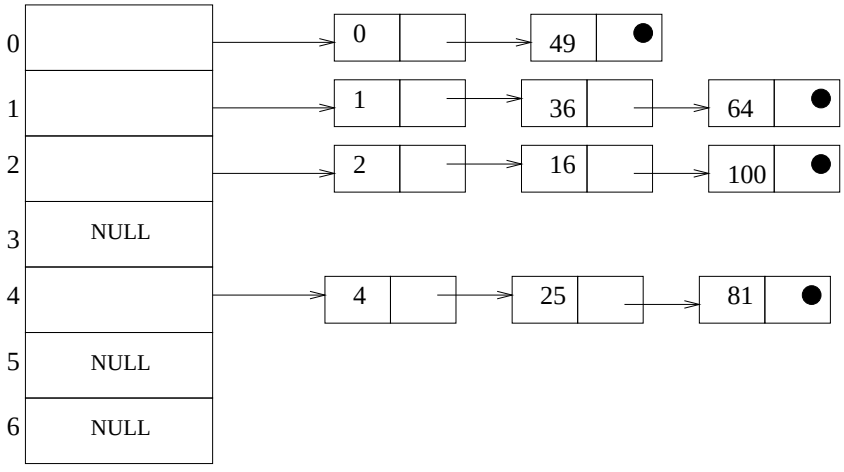


Figure 3.2: Open hashing: An example

$$h(x) = x \% 7$$

- Bucket lists
  - unsorted lists
  - sorted lists (these are better)
- Insert (x, T)
  Insert x at the head of list T[h(key (x))]
- Search (x, T)
  Search for an element x in the list T[h(key (x))]
- Delete (x, T)
  Delete x from the list T[h(key (x))]

Worst case complexity of all these operations is O(n)

In the average case, the running time is $O(1 + \alpha)$, where

$$\begin{aligned} \alpha &= \text{load factor} \triangleq \frac{n}{m} \text{ where} && (3.1) \\ n &= \text{number of elements stored} && (3.2) \\ m &= \text{number of hash values or buckets} \end{aligned}$$

It is assumed that the hash value h(k) can be computed in O(1) time. If n is O(m), the average case complexity of these operations becomes O(1) !

3.4 Closed Hashing

- All elements are stored in the hash table itself
- Avoids pointers; only computes the sequence of slots to be examined.
- Collisions are handled by generating a sequence of **rehash** values.

$$h : \underbrace{U}_{\text{universe of primary keys}} \times \underbrace{\{0, 1, 2, ...\}}_{\text{probe number}} \rightarrow \{0, 1, 2, ..., m-1\}$$

- Given a key x, it has a hash value h(x,0) and a set of rehash values

  $$h(x, 1), h(x,2), \ldots, h(x, m\text{-}1)$$

- We require that for every key x, the probe sequence

  $$< h(x,0), h(x, 1), h(x,2), \ldots, h(x, m\text{-}1)>$$

  be a permutation of <0, 1, ..., m-1>.

  This ensures that every hash table position is eventually considered as a slot for storing a record with a key value x.

Search (x, T)

- Search will continue until you find the element x (successful search) or an empty slot (unsuccessful search).

Delete (x, T)

- No delete if the search is unsuccessful.
- If the search is successful, then put the label DELETED (different from an empty slot).

Insert (x, T)

- No need to insert if the search is successful.
- If the search is unsuccessful, insert at the first position with a DELETED tag.

3.4.1 Rehashing Methods

Denote h(x, 0) by simply h(x).

1. Linear probing

   $$h(x, i) = (h(x) + i) \bmod m$$

2. Quadratic Probing

$$h(x,i) = (h(x) + C_1 i + C_2 i^2) \bmod m$$

where C_1 and C_2 are constants.

3. Double Hashing

$$h(x,i) = (h(x) + i \underbrace{h'(x)}_{\substack{\text{another}\\ \text{hash}\\ \text{function}}}) \bmod m$$

A Comparison of Rehashing Methods

Linear Probing	m distinct probe sequences	Primary clustering
Quadratic Probing	m distinct probe sequences	No primary clustering; but secondary clustering
Double Hashing	m^2 distinct probe sequences	No primary clustering No secondary clustering

3.4.2 An Example:

Assume linear probing with the following hashing and rehashing functions:

$$\begin{aligned} h(x,0) &= x\%7 \\ h(x,i) &= (h(x,0) + i)\%7 \end{aligned}$$

Start with an empty table.

Insert (20, T)
Insert (30, T)
Insert (9, T)
Insert (45, T)
Insert (14, T)

0	14
1	empty
2	30
3	9
4	45
5	empty
6	20

Search (35, T)
Delete (9, T)

0	14
1	empty
2	30
3	deleted
4	45
5	empty
6	20

Search (45, T)
Search (52, T)
Search (9, T)
Insert (45, T)
Insert (10, T)

0	14
1	empty
2	30
3	10
4	45
5	empty
6	20

Delete (45, T)
Insert (16, T)

0	14
1	empty
2	30
3	10
4	16
5	empty
6	20

3.4.3 Another Example:

Let m be the number of slots.

Assume :
- every even numbered slot occupied and every odd numbered slot empty
- any hash value between 0 . . . m-1 is equally likely to be generated.
- linear probing

empty
occupied
empty
occupied
empty
occupied
empty
occupied

Expected number of probes for a successful search = 1

Expected number of probes for an unsuccessful search

$$= \left(\frac{1}{2}\right)(1) + \left(\frac{1}{2}\right)(2)$$

$$= 1.5$$

3.5 Hashing Functions

What is a good hash function?

- Should satisfy the simple uniform hashing property.

  Let U = universe of keys

  Let the hash values be 0, 1, . . . , m-1

Let us assume that each key is drawn independently from U according to a probability distribution P. i.e., for k $\in$ U

$$P(k) = \text{ Probability that } k \text{ is drawn}$$

Then simple uniform hashing requires that

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \text{ for each } j = 0, 1, \ldots, m-1$$

that is, each bucket is equally likely to be occupied.

- Example of a hash function that satisfies simple uniform hashing property:

  Suppose the keys are known to be random real numbers k independently and uniformly distributed in the range [0,1).

  $$h(k) = \lfloor km \rfloor$$

  satisfies the simple uniform hashing property.

  Qualitative information about P is often useful in the design process. For example, consider a compiler's symbol table in which the keys are arbitrary character strings representing identifiers in a program. It is common for closely related symbols, say pt, pts, ptt, to appear in the same program. A good hash function would minimize the chance that such variants hash to the same slot.

- A common approach is to derive a hash value in a way that is expected to be independent of any patterns that might exist in the data.

  – The division method computes the hash value as the remainder when the key is divided by a prime number. Unless that prime is somehow related to patterns in the distribution P, this method gives good results.

3.5.1 Division Method

- A key is mapped into one of m slots using the function

$$h(k) = k \bmod m$$

- Requires only a single division, hence fast
- m should not be :
  - a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest order bits of k
  - a power of 10, since then the hash function does not depend on all the decimal digits of k
  - $2^p - 1$. If k is a character string interpreted in radix 2^p, two strings that are identical except for a transposition of two adjacent characters will hash to the same value.
- Good values for m
  - primes not too close to exact powers of 2.

3.5.2 Multiplication Method

There are two steps:

1. Multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA
2. Multiply this fractional part by m and take the floor.

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

where

$$kA \bmod 1 = kA - \lfloor kA \rfloor$$
$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- Advantage of the method is that the value of m is not critical. We typically choose it to be a power of 2:

  $$m = 2^p$$

  for some integer p so that we can then easily implement the function on most computers as follows:

  Suppose the word size $= w$. Assume that k fits into a single word. First multiply k by the w-bit integer $\lfloor A.2^w \rfloor$. The result is a $2w$ - bit value

  $$r_1 2^w + r_0$$

  where r_1 is the high order word of the product and r_0 is the low order word of the product. The desired p-bit hash value consists of the p most significant bits of r_0.

- Works practically with any value of A, but works better with some values than the others. The optimal choice depends on the characteristics of the data being hashed. Knuth recommends

  $$A \simeq \frac{\sqrt{5}-1}{2} = 0.6180339887\ldots \quad \text{(Golden Ratio)}$$

3.5.3 Universal Hashing

This involves choosing a hash function randomly in a way that is independent of the keys that are actually going to be stored. We select the hash function at random from a carefully designed class of functions.

- Let Φ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, 2, \ldots, m-1\}$.
- Φ is called **universal** if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \Phi$ for which $h(x) = h(y)$ is precisely equal to

  $$\frac{|\Phi|}{m}$$

- With a function randomly chosen from Φ, the chance of a collision between x and y where $x \neq y$ is exactly $\frac{1}{m}$.

Example of a universal class of hash functions:

Let table size m be prime. Decompose a key x into r +1 bytes. (i.e., characters or fixed-width binary strings). Thus

$$x = (x_0, x_1, \ldots, x_r)$$

Assume that the maximum value of a byte to be less than m.

Let $a = (a_0, a_1, \ldots, a_r)$ denote a sequence of $r + 1$ elements chosen randomly from the set $\{0, 1, \ldots, m-1\}$. Define a hash function $h_a \in \Phi$ by

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

With this definition, $\Phi = \sqcup_a \{h_a\}$ can be shown to be universal. Note that it has m^{r+1} members.

3.6 Analysis of Closed Hashing

Load factor α

$$\alpha \triangleq \frac{n}{m} = \frac{\text{\# of elements stored the table}}{\text{total \# of elements in the table}}$$

Assume uniform hashing. In this scheme, the probe sequence

$$< h(k, 0), \ldots, h(k, m-1) >$$

for each key k is equally likely to be any permutation of $< 0, 1, \ldots, m-1 >$

3.6.1 Result 1: Unsuccessful Search

- The expected number of probes in an unsuccessful search $\leq \frac{1}{1-\alpha}$

Proof: In an unsuccessful search, every probe but the last accesses an occupied slot that does not contain the desired key and the last slot probed is empty.

Let the random variable X denote the number of occupied slots probed before hitting an empty slot. X can take values $0, 1, \ldots, n$. It is easy to see that the expected number of probes in an unsuccessful search is $1 + E[X]$.

$$p_i \;=\; P\{\text{ exactly } i \text{ probes access occupied slots}\}, \text{for } i = 0, 1, 2, \ldots$$

for $i > n, p_i = 0$ since we can find at most n occupied slots.

Thus the expected number of probes

$$= 1 + \sum_{i=0}^{n} ip_i \qquad (*)$$

To evaluate (*), define

$$q_i = P\{\text{at least } i \text{ probes access occupied slots}\}$$

and use the identity:

$$\sum_{i=0}^{n} ip_i = \sum_{i=1}^{n} q_i$$

To compute q_i, we proceed as follows:

$$q_1 = \frac{n}{m}$$

since the probability that the first probe accesses an occupied slot is n/m.

With uniform hashing, a second probe, if necessary, is to one of the remaining $m - 1$ unprobed slots, $n - 1$ of which are occupied. Thus

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)$$

since we make a second probe only if the first probe accesses an occupied slot.

In general,

$$\begin{aligned} q_i &= \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-i+1}{m-i+1}\right) \\ &\leq \left(\frac{n}{m}\right)^i = \alpha^i \quad \text{since } \frac{n-j}{m-j} \leq \frac{n}{m} \end{aligned}$$

Thus the expected number of probes in an unsuccessful search

$$\begin{aligned} &= 1 + \sum_{i=o}^{n} ip_i \\ &\leq 1 + \alpha + \alpha^2 + \cdots \\ &= \frac{1}{1-\alpha} \end{aligned}$$

- Intuitive Interpretation of the above:

  One probe is always made, with probability approximately α a second probe is needed, with probability approximately α^2 a third probe is needed, and so on.

3.6.2 Result 2: Insertion

- Insertion of an element requires at most $\frac{1}{1-\alpha}$ probes on average.
- Proof is obvious because inserting a key requires an unsuccessful search followed by a placement of the key in the first empty slot found.

3.6.3 Result 3: Successful Search

- The expected number of probes in a successful search is at most

$$\frac{1}{\alpha}\ \ln\frac{1}{1-\alpha}\ + \frac{1}{\alpha}$$

  assuming that each key in the table is equally likely to be searched for.

  A search for a key k follows the same probe sequence as was followed when the element with key k was inserted. Thus, if k was the $(i+1)$th

key inserted into the hash table, the expected number of probes made in a search for k is at most

$$\frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$$

Averaging over all the n keys in the hash table gives us the expected # of probes in a successful search:

$$\begin{aligned}\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n})\end{aligned}$$

where

$$H_i = \sum_{j=1}^{i}\frac{1}{j} \text{ is the } i^{th} \text{ Harmonic number.}$$

Using the well known bounds,

$$\ln i \leq H_i \leq \ln i + 1,$$

we obtain

$$\begin{aligned}\frac{1}{\alpha}(H_m - H_{m-n}) &\leq \frac{1}{\alpha}[\ln m + 1 - \ln(m-n)] \\ &= \frac{1}{\alpha}\ln\frac{m}{m-n} + \frac{1}{\alpha} \\ &= \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}\end{aligned}$$

3.6.4 Result 4: Deletion

- Expected # of probes in a deletion

$$\leq \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}$$

Proof is obvious since deletion always follows a successful search.

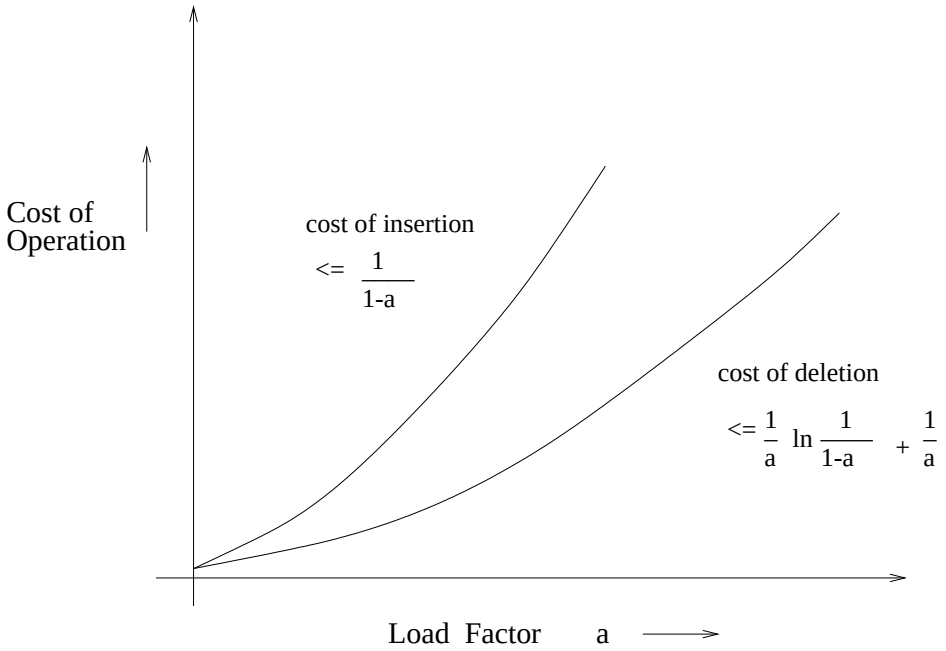


Figure 3.3: Performance of closed hashing

Figure 3.3 depicts the performance of closed hashing for all the four operations discussed above.

3.7 Hash Table Restructuring

- When a hash table has a large number of entries (ie., let us say $n \geq 2m$ in open hash table or $n \geq 0.9m$ in closed hash table), the average time for operations can become quite substantial. In such cases, one idea is to simply create a new hash table with more number of buckets (say twice or any appropriate large number).
- In such a case, the currently existing elements will have to be inserted into the new table. This may call for
  - rehashing of all these key values
  - transferring all the records

  This effort will be less than it took to insert them into the original table.

- Subsequent dictionary operations will be more efficient and can more than make up for the overhead in creating the larger table.

3.8 Skip Lists

REF. William Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, Volume 33, Number 6, pp. 668-676, 1990.

- Skip lists use probabilistic balancing rather than strictly enforced balancing.
- Although skip lists have bad worst-case performance, no input sequence consistently produces the worst-case performance (like quicksort).
- It is very unlikely that a skip list will be significantly unbalanced. For example, in a dictionary of more than 250 elements, the chance is that a search will take more than 3 times the expected time $\leq 10^{-6}$.
- Skip lists have balance properties similar to that of search trees built by random insertions, yet do not require insertions to be random.

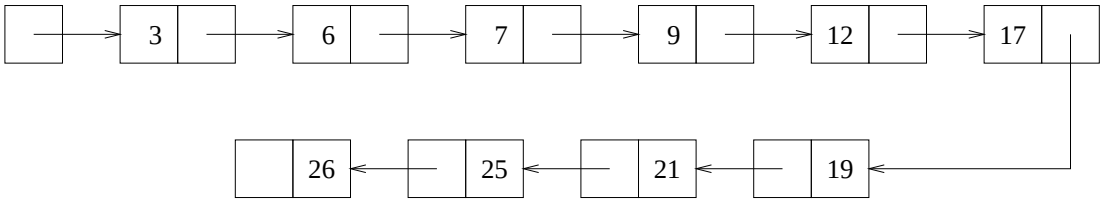


Figure 3.4: A singly linked list

Consider a singly linked list as in Figure 3.4. We might need to examine every node of the list when searching a singly linked list.

Figure 3.5 is a sorted list where every other node has an additional pointer, to the node two ahead of it in the list. Here we have to examine no more than $\lceil \frac{n}{2} \rceil + 1$ nodes.

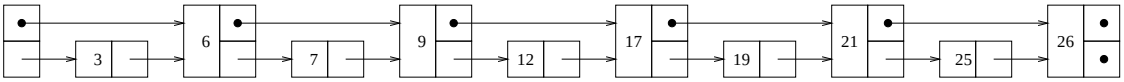


Figure 3.5: Every other node has an additional pointer

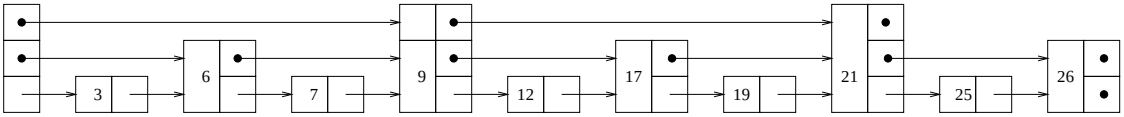


Figure 3.6: Every second node has a pointer two ahead of it

In the list of Figure 3.6, every second node has a pointer two ahead of it; every fourth node has a pointer four ahead if it. Here we need to examine no more than $\lceil \frac{n}{4} \rceil + 2$ nodes.

In Figure 3.7, (every $(2^i)^{th}$ node has a pointer (2^i) node ahead $(i = 1, 2, ...)$; then the number of nodes to be examined can be reduced to $\lceil \log_2 n \rceil$ while only doubling the number of pointers.

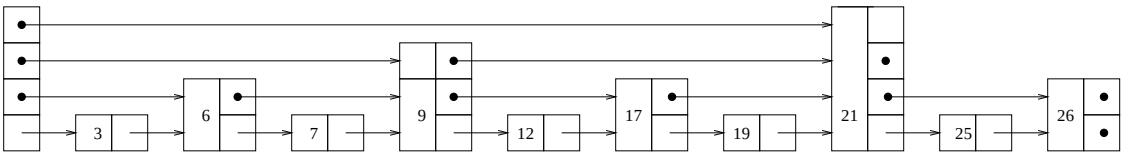


Figure 3.7: Every $(2^i)^{th}$ node has a pointer to a node (2^i) nodes ahead $(i = 1, 2, ...)$

- A node that has k forward pointers is called a *level k* node. If every $(2^i)^{th}$ node has a pointer (2^i) nodes ahead, then

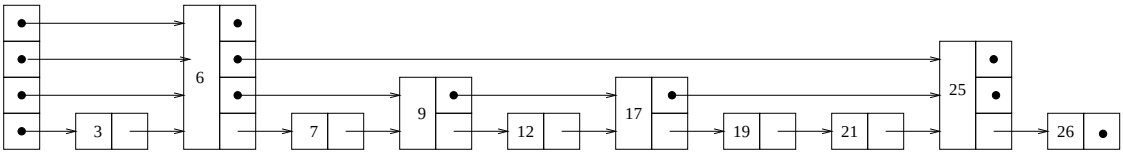


Figure 3.8: A skip list

# of level 1 nodes 50 %

# of level 2 nodes 25 %

# of level 3 nodes 12.5 %

- Such a data structure can be used for fast searching but insertions and deletions will be extremely cumbersome, since levels of nodes will have to change.
- What would happen if the levels of nodes were randomly chosen but in the same proportions (Figure 3.8)?
  - level of a node is chosen randomly when the node is inserted
  - A node's i^{th} pointer, instead of pointing to a node that is 2^{i-1} nodes ahead, points to the next node of level i or higher.
  - In this case, insertions and deletions will not change the level of any node.
  - Some arrangements of levels would give poor execution times but it can be shown that such arrangements are rare.

  Such a linked representation is called a skip list.
- Each element is represented by a node the level of which is chosen randomly when the node is inserted, without regard for the number of elements in the data structure.
- A level i node has i forward pointers, indexed 1 through i. There is no need to store the level of a node in the node.
- Maxlevel is the maximum number of levels in a node.
  - Level of a list = Maxlevel
  - Level of empty list = 1
  - Level of header = Maxlevel

3.8.1 Initialization:

An element NIL is allocated and given a key greater than any legal key. All levels of all lists are terminated with NIL. A new list is initialized so that the level of list = maxlevel and all forward pointers of the list's header point to NIL

Search:

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

Insertion and Deletion:

- Insertion and deletion are through search and splice
- update $[i]$ contains a pointer to the rightmost node of level i or higher that is to the left of the location of insertion or deletion.
- If an insertion generates a node with a level greater than the previous maximum level, we update the maximum level and initialize appropriate portions of update list.
- After a deletion, we check to see if we have deleted the maximum level element of the list and if so, decrease the maximum level of the list.
- Figure 3.9 provides an example of Insert and Delete. The pseudocode for Insert and Delete is shown below.

```
search (list, searchkey);
    { x = list → header;
      for (i = list → level; i ≥ 1;  i − −) {
          while (x → forward[i] → key < searchkey)
```

```
            x = x → forward[i];
    }
    x = x → forward[i];
    if (x → key = searchkey)    return (true)
    else    return  false;
}
```

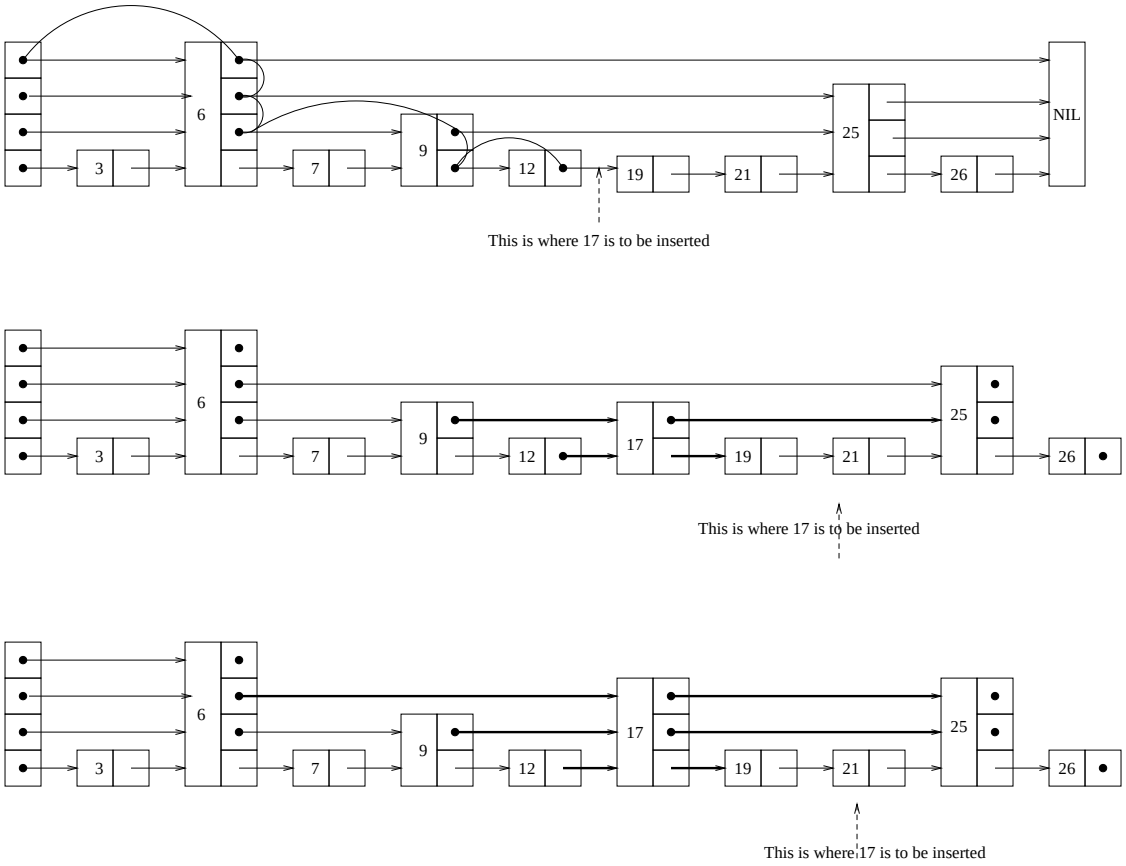


Figure 3.9: A skip list

```
insert (list, searchkey);
{ x = list → header ;
  for (i = list → level; i ≥ 1; i − −)  {
     while (x → forward[i] → key < searchkey)
           x = x → forward[i];
     update[i] = x
  }
```

```
    x = x → forward[1];
    if (x → key = searchkey)  return ("key  already  present")
    else {
        newLevel = randomLevel();
        if newLevel > list → level {
                for (i = list → level + 1; i ≤ newLevel; i ++ )
                    update[i] = list → header;
        }
        x = makenode(newLevel, searchkey);
        for (i = 1, i ≤ newLevel; i++) {
                x → forward[i] = update[i] → forward[i];
                update[i] → forward[i] = x
        }
 } }
delete (list, searchkey);
 { x = list → header ;
   for (i = list → level; i ≥ 1; i – )  {
        while (x → forward[i] → key < searchkey)
                x = x → forward[i];
        update[i] = x
   }
   x = x → forward[1];
   if (x → key = searchkey)  {
        for (i = 1; i ≤ list → level; i ++)  {
                if (update[i] → forward[i] ≠ x)
                    break;
                if (update[i] → forward[i] = x → forward[i];
        }
        free(x)
```

```
While (( list → 1) &&
  (list → header → forward [list+level] = NIL))
                list → level = list → level - 1
```

3.9 Analysis of Skip Lists

In a skiplist of 16 elements, we may have

- 9 elements at level 1
- 3 elements at level 2
- 3 elements at level 3
- 1 element at level 6

- One important question is:
  Where do we start our search? Analysis shows we should start from level $L(n)$ where
  $$L(n) = \log_2 n$$
  In general if p is the probability fraction,
  $$L(n) = \log_{\frac{1}{p}} n$$
  where p is the fraction of the nodes with level i pointers which also have level $(i + 1)$ pointers.
- However, starting at the highest level does not alter the efficiency in a significant way.
- Another important question to ask is:
  What should be MaxLevel? A good choice is
  $$MaxLevel = L(N) = \log_{\frac{1}{p}} N$$
  where N is an upperbound on the number of elements is a skiplist.

- Complexity of search, delete, insert is dominated by the time required to search for the appropriate element. This in turn is proportional to the length of the search path. This is determined by the pattern in which elements with different levels appear as we traverse the list.
- Insert and delete involve additional cost proportional to the level of the node being inserted or deleted.

3.9.1 Analysis of Expected Search Cost

We analyze the search path backwards, traveling up and to the left. We pretend as if the level of a node is being determined only when it is observed while backtracking the search path.

1. From level 1, we rise to level $L(n)$
2. We move leftward
3. We climb from $L(n)$ to maxLevel.

At a particular point in the climb, we are at the i^{th} forward pointer of a node x and we have no knowledge about the levels of nodes to the left of x or about the level of x (other than that the level of x must be $\geq i$). See situation (a) in figure.

- Assume "x" is not the header.
- If level of x is i, then we are in situation (b). If level of x is $> i$, we are in situation (c).
- $Prob\{$ we are in situation $cA\} = p$
- Each time we are in situation c, we climb up one level.

Let

$$C(k) = \begin{array}{l}\text{expected length of a search path that} \\ \text{climbs up } k \text{ levels in an infinite list.}\end{array}$$

Then:

$$\begin{aligned} C(0) = &\ 0 \\ C(k) = &\ p\ \{\text{cost in situation c }\} \\ &\ (1-p)\ \{\text{ cost in situation b }\} \end{aligned}$$

We get

$$\begin{aligned} C(k) = &\ p\ \{1 + C(k-1)\} + (1-p)\{1 + C(k)\} \\ C(k) = &\ \tfrac{1}{p} + C(k-1) \\ C(k) = &\ \tfrac{k}{p} \end{aligned}$$

Our assumption that the list is infinite is pessimistic.

When we bump into the header in our backward climb, we simply climb it up without performing any leftward (horizontal) movements.

This gives us an upperbound of

$$\left(\frac{L(n)-1}{p}\right)$$

On the expected length of a path that climbs up from level 1 to level $L(n)$ in a list of n elements. (because $L(n)-1$ level are being climbed).

The rest of the journey includes

1. leftward movements, the number of which is bounded by the number of elements of level $L(n)$ or higher. his has an expected value $= \frac{1}{p}$
2. We also move upwards from level $L(n)$ to the maximum level in the list.

$$\begin{aligned} p\ \{\ \text{maximum level in the list}\ &> k\} \\ &= 1 - (1-p^k)^n \\ &\leq np^k \end{aligned}$$

$$
\begin{array}{lll}
 & p\{ \text{ an element has a level } > k\} & = p^k \\
\Longrightarrow & p\{ \text{ an element has level } \leq k\} & = 1 - p^k \\
\Longrightarrow & p\{ \text{ all } n \text{ elements have level } \leq k\} & = (1 - p^k)^n \\
\Longrightarrow & p\{ \text{ at least one element has level } \geq k\} & = 1 - (1 - p^k)^n \\
 & \text{Expected maximum level} & \leq np^k \\
\Longrightarrow & \text{Expected maximum level} & \leq L(n) + \frac{1}{1-p}
\end{array}
$$

Putting our results together, we get:

Total expected cost to climb out of a list on n elements

$\leq \frac{L(n)}{p} + \frac{1}{p}$ which is $O(\log n)$.

We can also compute the probability if the actual cost of a search exceeds expected cost by more than a specified ratio.

Examples

1. $p = \frac{1}{2}$; $n = 256$
   p {search will take longer than 3 times the expected cost } $= \overline{10}^6$
2. $p = \frac{1}{2}$; $n = 4096$
   p {search will take longer than 2 times the expected cost } $= \overline{10}^4$
3. $p = \frac{1}{2}$; $n = 1024$
   p {search will take longer than 3 times } $= \overline{10}^{18}$

How do we choose p ?

- Decreasing p increases variability of running times.
- $p = \frac{1}{2}$; generation of random level can be done from a stream of random bits.

- Decreasing p decreases # of pointers per node.
- Overheads are related to $L(n)$ rather than $\frac{L(n)}{p}$.
- $p = \frac{1}{4}$ is probably the best choice, unless variability of running times is an issue.

3.10 To Probe Further

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
2. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.
3. Donald E Knuth. *Fundamental Algorithms*, Volume 1 of The Art of Computer Programming, Addison-Wesley, 1968, Second Edition, 1973.
4. Robert L. Kruse, Bruce P. Leung, and Clovis L. Tondo. *Data Structures and Program design in C*. Prentice Hall, 1991. Indian Edition published by Prentice Hall of India, 1999.
5. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.
6. Ellis Horowitz, Sartaz Sahni, and Rajasekaran. *Fundamentals of Computer Algorithms*. W.H. Freeman and Company, 1998. Indian Edition published by Galgotia Publications, 2000.
7. Donald E. Knuth. *Sorting and Searching*, Volume 3 of The Art of Computer Programming, Addison-Wesley, 1973.
8. Donald E Knuth. *Seminumerical Algorithms*. Volume 2 of The Art of Computer Programming, Addison-Wesley, 1969, Second Edition, 1981.

9. Y. Langsam, M.J. Augenstein, and A.M. Tenenbaum. *Data Structures using C and C++*. Second Edition, Prentice Hall, 1996. Indian Edition published by Prentice Hall of India, 2000.

10. Kurt Mehlhorn. *Sorting and Searching*. Volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

11. William Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, Volume 33, Number 6, pp. 668-676, 1990.

3.11 Problems

1. Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a closed hash table of length $m = 11$ using the primary hashing function $h(k) = k \bmod m$. Illustrate the result of inserting these keys using

   (a) Linear probing.

   (b) Quadratic probing with $h_i(k) = (h(k) + i + 3i^2) \bmod m$.

   (c) Double hashing with rehashing function $h_i(k) = (h(k) + i(1 + k \bmod (m - 1)) \bmod m$.

2. A closed hash table of size m is used to store n items, where $n \leq m/2$.

   (a) Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability that the i^{th} insertion requires strictly more than k probes is at most 2^{-k}.

   (b) Show that for $i = 1, 2, \ldots, n$, the probability that the i^{th} insertion requires more than $2 \log n$ probes is at most n^{-2}.

3. Consider a closed hash table with $2n$ buckets ($n > 0$), employing the hash function $h(x) = x \bmod 2n$. Which of the following rehash strategies would you prefer and why?

   (a) Strategy 1: $h_i(x) = (h(x) + i) \bmod 2n$.

   (b) Strategy 2: $h_i(x) = (h(x) + 2i) \bmod 2n$.

4. Given a set of a maximum of 52 names of 5 characters each and that not more than two names start with the same character, is it possible to find a closed hashing function that operates on a hash table of size 56? Each element of the hash table can be a string of 5 characters. The hashing function should provide for membership test in at most 3 time units. Either exhibit such a hashing function or provide a counter-example. Assume that only upper case alphabetic characters are used in

names and that the hashing function can be computed in one time unit. Comparison of strings also takes one time unit.

5. Consider all two character names with each character being from the set $\{A, B, C, D, E, F, G, H, I, J$ Design an efficient hashing function such that there would be no collisions on insertion. Assume a hash table of size 115.

6. Suppose that we are given a key k to search for in a closed hash table with positions $0, 1, \ldots, m-1$, where m is known to be a power of 2. Let h be a hash function. Consider the following search scheme:

   1. Compute $i = h(k)$ and set $j = 0$.
   2. Probe in position i. Terminate if the required item is found or that position is empty.
   3. Set $j = (j + 1)$ **mod** m and $i = (i + j)$ **mod** m. Return to Step 2.

   Answer the following questions and justify your answers.

   a. Does the algorithm examine every table position in the worst case?

   b. Is this linear probing or quadratic probing or double hashing or none?

   c. What is the distinct number of probe sequences?

7. Given a skip list of 16 elements, compute the (best case) minimum number of probes required for an unsuccessful search in each case below:

   (a) All nodes in the list are level 4 nodes

   (b) There are 14 nodes of level 1 and 2 nodes of level 3

   (c) There is one level 4 node, one level 3 node, and the rest are level 1 nodes

8. The following results have been experimentally observed by William Pugh in his CACM article on skip lists, while comparing the average case performance of skip lists, splay trees, non-recursive AVL trees, and recursive 2-3 trees. Provide an intuitive justification for each of these:

   (a) The non-recursive AVL tree has the least average case complexity for a dictionary search operation

   (b) The skip list outperforms all other data structures for insertions and deletions

   (c) Splay trees perform better than 2-3 trees for insertions

3.12 Programming Assignments

3.12.1 Hashing: Experimental Analysis

The objective of this assignment is to compare the performance of different hash table organizations, hashing methods, and collision resolution techniques.

Generation of Keys

Assume that your keys are character strings of length 10 obtained by scanning a C program. Call these as tokens. Define a token as any string that appears between successive occurrences of a forbidden character, where the forbidden set of characters is given by:

$$F = \{!, ", \#, \%, \&, (,), *, +, -, ., /, :, ;, <, >, =, ?, [,], \{, \}, \ , |, \text{backslash}, \text{comma}, \text{space}\}$$

Exceptions to the above rule are:

- Ignore anything that appears as comment (that is between /* and */)
- Ignore anything that appears between double quotes

That is, in the above two cases, the character strings are not to be taken as tokens. If a string has less than 10 characters, make it up to 10 characters by including an appropriate number of trailing *'s. On the other hand, if the current string has more than 10 characters, truncate it to have the first ten characters only.

From the individual character strings (from now on called as **tokens**), generate a positive integer (from now on called as **keys**) by summing up the ASCII values of the characters in the particular token. Use this integer key in the hashing functions. However, remember that the original token is a character string and this is the one to be stored in the hash table.

Hash Table Methods to be Evaluated

As discussed in the class, the following eleven schemes are to be evaluated.

1. Open with division method for hashing and unsorted lists for chaining
2. Open with division method for hashing and sorted lists for chaining

3. Open with multiplication method for hashing and unsorted lists for chaining
4. Open with multiplication method for hashing and sorted lists for chaining
5. Closed with simple coalesced hashing
6. Closed with linear probing
7. Closed with quadratic probing
8. Closed with double hashing
9. Closed with linear probing and Knuth-Amble method (Keys with same hash value appear in descending order)
10. Closed with quadratic probing and Knuth-Amble method
11. Closed with double hashing and Knuth-Amble method

Hashing Functions

For the sake of uniformity, use the following hashing functions only. In the following, m is the hash table size (that is, the possible hash values are, $0, 1, \ldots, m-1$), and x is an integer key.

1. **Division Method**
$$h(x) = x \ \textbf{mod} \ m$$
2. **Multiplication Method**
$$h(x) = \text{Floor}(m * \text{Fraction}(k * x))$$
where $k = \frac{\sqrt{(5)}-1}{2}$, the Golden Ratio.
3. **Linear Probing**
$$h(x,i) = (h(x,0) + i) \ \textbf{mod} \ m; i = 0, \ldots, m-1$$
4. **Quadratic Probing**
$$h(x,i) = (h(x,0) + i + i^2) \ \textbf{mod} \ m; i = 0, \ldots, m-1$$
5. **Double Hashing**
Use the division method for $h(x)$ and the multiplication method for $h'(x)$.

Inputs to the Program

The possible inputs to the program are:

- m: Hash table size. Several values could be given here and the experiments are to be repeated for all values specified. If nothing is specified, assume all primary numbers between 7 and 97.
- n: The initial number of insertions to be made for setting up a hash table with a particular load factor.
- M: This is a subset of the set $\{1, 2, \ldots, 11\}$ indicating the set of methods to be investigated.
- I: This is a decimal number from which a ternary string is to be generated (namely the radix-3 representation of I). In this representation, assume that a **0** represents the **search** operation, a **1** represents the **insert** operation, and a **2** represents the **delete** operation.
- A C program, from which the tokens are to be picked up for setting up and experimenting with the hash table.

What should the Program Do?

1. For each element of M and for each value of m, do the following.
2. Scan the given C program and as you scan, insert the first n tokens scanned into an initially empty hash table.
3. Now scan the rest of the C program token by token, searching for it or inserting it or deleting it, as dictated by the radix-3 representation of I. Note that the most significant bit (tigit?) is to be considered first while doing this and you proceed from left to right in the radix-3 representation. For each individual operation, keep track of the number of probes.
4. Compute the average number of probes for a typical successful search, unsuccessful search, insert, and delete.
5. Repeat Steps 2, 3, and 4 for each value of M and each value of m.
6. For each individual method investigated, obtain a table that lists the average number of probes for the four cases for various values of m. The table entries can be of the following format:

Hash table size	Average Usearch	Number of Ssearch	Probes for Insert	Delete

7. For each individual value of m, obtain a table that lists the average number of probes for various methods. The table entries can be of the following format:

Hash table method	Average Number of Probes for			
	Usearch	Ssearch	Insert	Delete

3.12.2 Skip Lists: Experimental Analysis

Implement skip list data structure. Analyze the efficiency of search, insert, and delete operations on randomly generated inputs. Carefully take into account all the tradeoffs involved in designing and implementing skip lists. Validate the experimental results with those given in the article by William Pugh (Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, Volume 33, Number 6, pp. 668-676, 1990). Design and carry out experiments to compare the performance of skip lists with that of hash tables.

Chapter 4

Binary Trees

4.1 Introduction

Trees are very useful and important abstractions in Computer Science. They are ideally suited to representing hierarchical relationships among elements in a universe. Efficient implementations of abstract data types such as dictionaries and priority queues are invariably in terms of binary or general trees.

4.1.1 Definitions

We shall first present some definitions and then introduce the Tree ADT.

- **Tree**:

  1. A single node is a tree and this node is the root of the tree.
  2. Suppose r is a node and $T_1, T_2, \ldots, T_k$ are trees with roots $r_1, r_2, \ldots, r_k$, respectively, then we can construct a new tree whose root is r and $T_1, T_2, \ldots, T_k$ are the subtrees of the root. The nodes $r_1, r_2, \ldots, r_k$ are called the children of r.

  See Figure 4.1. Often, it is convenient to include among trees, the **null** tree, which is a tree with no nodes.

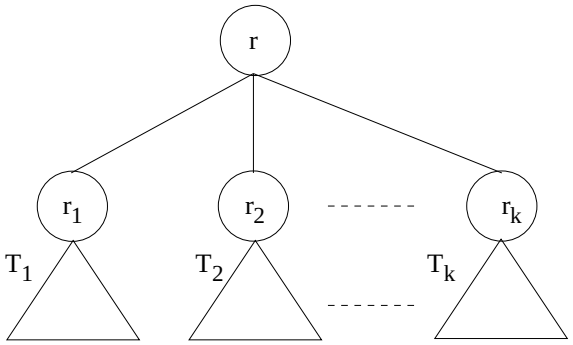


Figure 4.1: Recursive structure of a tree

- **Path**

  A sequence of nodes $n_1, n_2, \ldots, n_k$, such that n_i is the parent of n_{i+1} for $i = 1, 2, \ldots, k-1$. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

- **Siblings**

  The children of a node are called siblings of each other.

- **Ancestor and Descendent**

  If there is a path from node a to node b, then

  a is called an **ancestor** of b

  b is called a **descendent** of a

  If $a \neq b$, then a is a proper ancestor and b is a proper descendent.

- **Subtree**

  A subtree of a tree is a node in that tree together with all its descendents.

- **Height**

  The height of a node in a tree is the length of a longest path from the node to a leaf. The height of a tree is the height of its root.

- **Depth**

  The depth of a node is the length of the unique path from the root to the node.

- **Tree Traversal**

  This is a systematic way of ordering the nodes of a tree. There are three popular schemes (many other schemes are possible):

  1. Preorder
  2. Inorder
  3. Postorder

  All these are recursive ways of listing or visiting all the nodes (each node exactly once)

4.1.2 Preorder, Inorder, Postorder

- If a tree is null, then the empty list is the preorder, inorder, and postorder listing of T
- If T comprises a single node, that node itself is the preorder, inorder, and postorder list of T
- Otherwise
  1. The preorder listing of T is the root of T, followed by the nodes of T_1 in preorder, . . . , and the nodes of T_k in preorder.
  2. The inorder listing of T is the nodes of T_1 in inorder, followed by the root r, followed by the nodes of T_2 in inorder, . . . , and the nodes of T_k in inorder.
  3. The postorder listing of T is the nodes of T_1 in postorder, . . . , the nodes of T_k in postorder, all followed by the root r.
- **Example**: see Figure 4.2.

  Preorder 1,2,3,5,8,9,6,10,4,7

  Postorder 2,8,9,5,10,6,3,7,4,1

  Inorder 2,1,8,5,9,3,10,6,7,4
- Note that the order of appearance of the leaves is the same in all the three schemes. This is true in general.

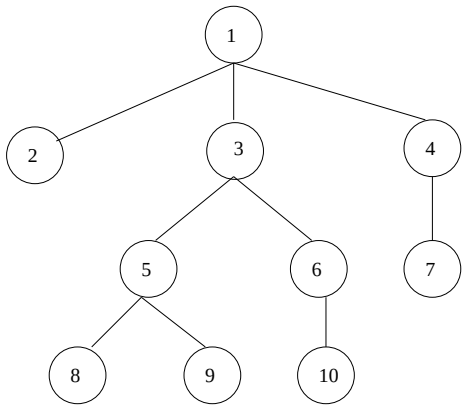


Figure 4.2: Example of a general tree

4.1.3 The Tree ADT

A tree ADT can be defined to comprise the following operations.

1. parent (n, T)
2. lmostchild (n, T)
3. rsibling (n, T)
4. root (T)
5. makenull (n, T)
6. height (n, T)
7. depth (n, T)
8. createi $(v, T_1, T_2, \ldots, T_i)$ creates the tree shown in Figure 4.3

4.1.4 Data Structures for Tree Representation

The data structure chosen should be able to support efficiently the ADT operations given above. Simple schemes include:

1. Arrays

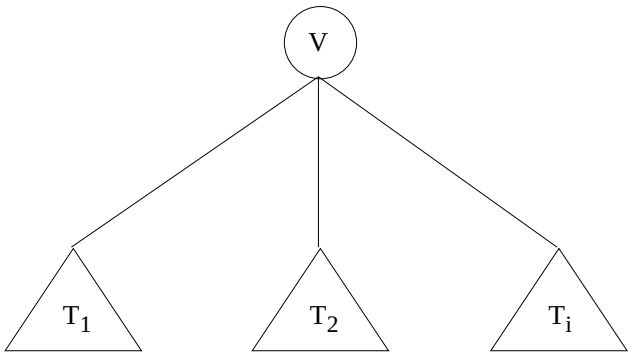


Figure 4.3: A tree with i subtrees

2. Lists of children
3. Leftmost child-right sibling representation

For a detailed discussion of data structures to support Tree ADT operations, refer to the book by Aho, Hopcroft, and Ullman (1983).

4.2 Binary Trees

Definition: A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

- If h = height of a binary tree,
  max # of leaves $= 2^h$
  max # of nodes $= 2^{h+1} - 1$
- A binary tree with height h and $2^{h+1} - 1$ nodes (or 2^h leaves) is called a **full binary tree**
- Figure 4.4 shows several examples of binary trees.
- The nodes of a binary tree can be numbered in a natural way, level by level, left to right. For example, see Figure 4.5.

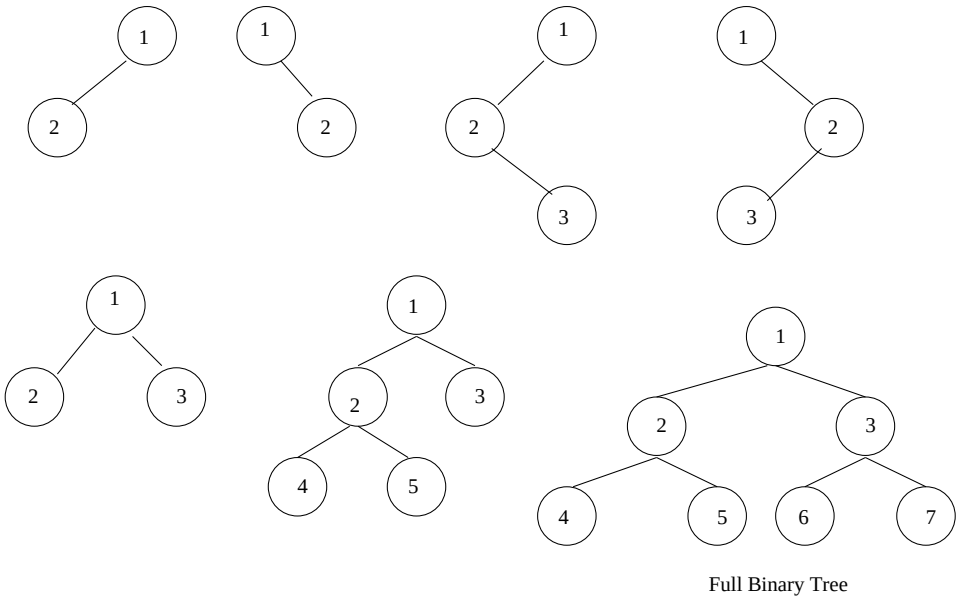


Figure 4.4: Examples of binary trees

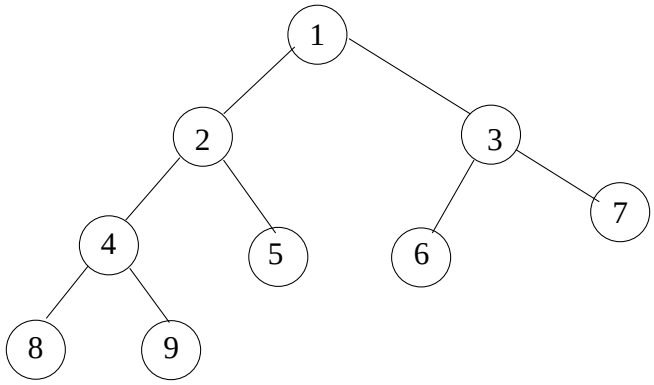


Figure 4.5: Level by level numbering of a binary tree

- A binary tree with n nodes is said to be **complete** if it contains all the first n nodes of the above numbering scheme. Figure 4.6 shows examples of complete and incomplete binary trees.

- Total number of binary trees having n nodes

  = number of stack-realizable permutations of length n

  = number of well-formed parentheses (with n left parentheses and n right parentheses)

  $= \left(\frac{1}{n+1}\right) \begin{pmatrix} 2n \\ n \end{pmatrix}$ Catalan Number

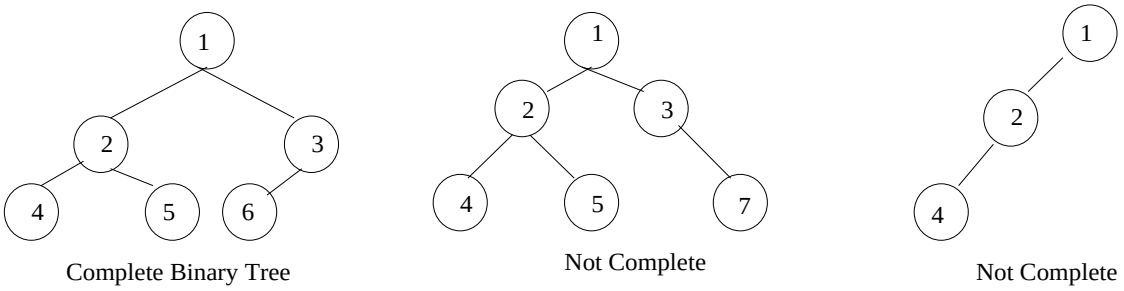


Figure 4.6: Examples of complete, incomplete binary trees

Binary Tree Traversal:

1. **Preorder**

   Visit root, visit left subtree in preorder, visit right subtree in preorder

2. **Postorder**

   Visit left subtree in postorder, right subtree in postorder, then the root

3. **Inorder**

   Visit left subtree in inorder, then the root, then the right subtree in inorder

- Figure 4.7 shows several binary trees and the traversal sequences in preorder, inorder, and postorder.

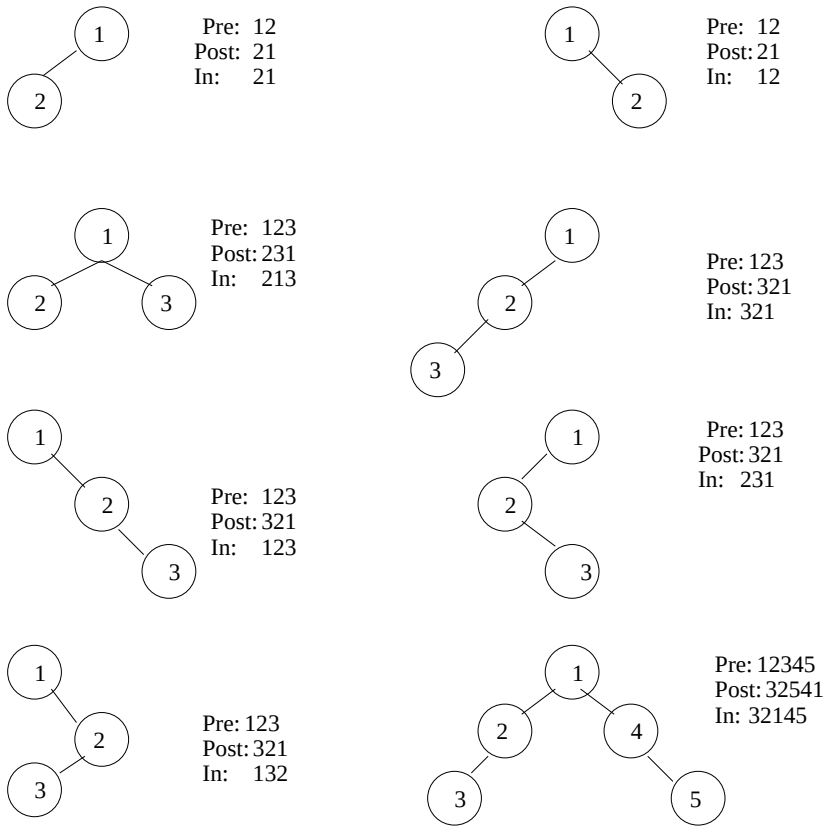


Figure 4.7: Binary tree traversals

- We can construct a binary tree uniquely from preorder and inorder **or** postorder and inorder **but not** preorder and postorder.

Data Structures for Binary Trees:

1. **Arrays**, especially suited for complete and full binary trees
2. **Pointer-based**

Pointer-based Implementation:

```
typedef struct node-tag {
      item-type info ;
      struct node-tag * left ;
      struct node-tag * right ;
    } node-type ;
   node-type  * root ;    /* pointer to root */
```

```
node-type  * p ;      /* temporary pointer */
void  preorder(node-type * root)
 {
  if (root) {
        visit (root) ;
        preorder (root → left) ;
        preorder (root → right) ;
  }
 }
void  inorder (node-type * root)
 {
  if (root) {
        inorder (root → left) ;
        visit (root) ;
        inorder (root → right) ;
  }
 }
void  postorder (node-type * root)
 {
  if (root) {
        postorder (root → left) ;
        postorder (root → right) ;
        visit (root) ;
  }
 }
```

4.3 An Application of Binary Trees: Huffman Code Construction

REF. David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*. Volume 40, Number 9, pp. 1098-1101, 1952.

- Suppose we have messages consisting of sequences of characters. In each message, the characters are independent and appear with a known probability in any given position; the probabilities are the same for

all positions.

e.g. - Suppose we have a message made from the five characters a, b, c, d, e, with probabilities 0.12, 0.40, 0.15, 0.08, 0.25, respectively.

We wish to encode each character into a sequence of 0s and 1s so that no code for a character is the prefix of the code for any other character. This prefix property allows us to decode a string of 0s and 1s by repeatedly deleting prefixes of the string that are codes for characters.

Symbol	Prob	code 1	code 2
a	0.12	000	000
b	0.40	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

In the above, Code 1 has the prefix property. Code 2 also has the prefix property.

- The problem: given a set of characters and their probabilities, find a code with the *prefix property* such that the average length of a code for a character is a minimum.

  Code 1: $(0.12)(3)+(0.4)(3)+(0.15)(3)+(0.08)(3)+(0.25)(3) = 3$

  Code 2: $(0.12)(3)+(0.4)(2)+(0.15)(2)+(0.08)(3)+(0.25)(2) = 2.2$

- Huffman's algorithm is one technique for finding optional prefix codes. The algorithm works by selecting two characters a and b having the lowest probabilities and replacing them with a single (imaginary) character, say x, whose probability of occurrence is the sum of probabilities for a and b. We then find an optimal prefix code for this smaller set of characters, using this procedure recursively. The code for the original character set is obtained by using the code for x with a 0 appended for $''a''$ and a 1 appended for $''b''$.

- We can think of prefix codes as paths in binary trees. For example, see Figure 4.8.

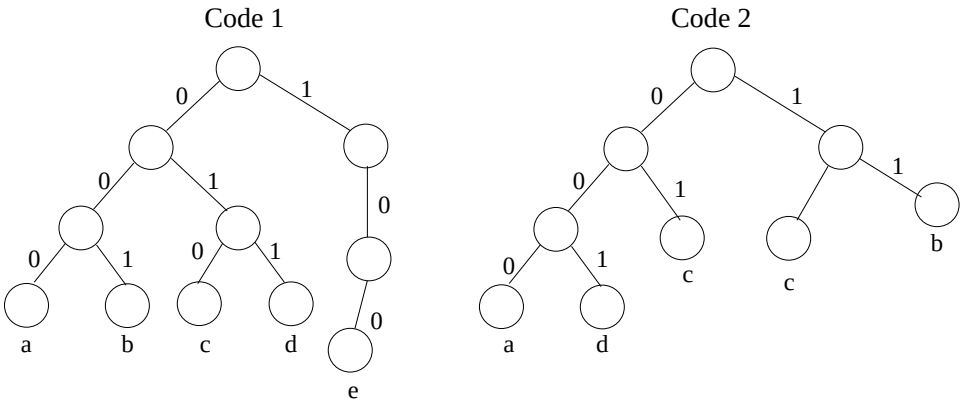


Figure 4.8: Code 1 and Code 2

- The prefix property guarantees that no character can have a code that is an interior node, and conversely, labeling the leaves of any binary tree with characters gives us a code with the prefix property for these characters.
- Huffman's algorithm is implemented using a *forest* (disjoint collection of trees), each of which has its leaves labeled by characters whose codes we desire to select and whose roots are labeled by the sum of the probabilities of all the leaf labels. We call this sum the *weight* of the tree.
- Initially each character is in a one-node tree by itself and when the algorithm ends, there will be only one tree, with all the characters as its leaves. In this final tree, the path from the root to any leaf represents the code for the label of that leaf.
- The essential step of the algorithm is to select the two trees in the forest that have the smallest weights (break ties arbitrarily). Combine these two trees into one, whose weight is the sum of the weights of the two trees. To combine the trees, we create a new node, which becomes the root and has the roots of the two given trees as left and right children (which is which doesn't matter). This process continues until only one tree remains.

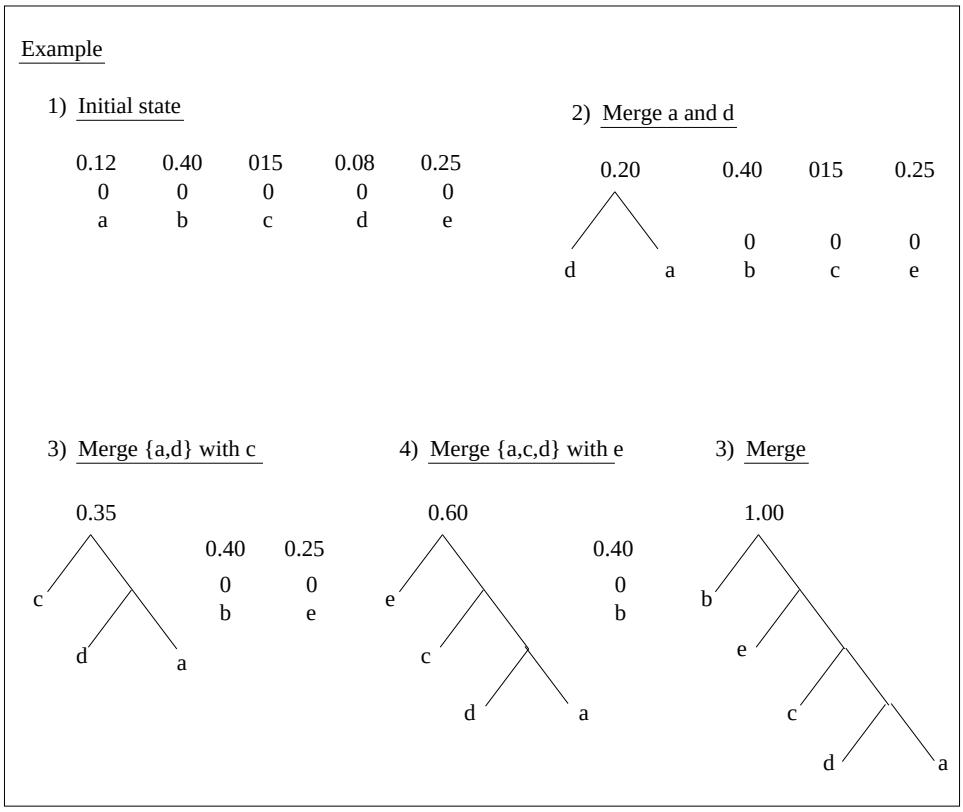


Figure 4.9: An example of Huffman algorithm

- An example of Huffman algorithm is shown in Figure 4.9 for five alphabets.

4.3.1 Implementation

We use three arrays, tree, alphabet, and forest.

- Each element in the array "tree" has 3 fields: lchild, rchild, parent, and represents a node of a tree. This array has information about all nodes in all trees in the forest.
- The array "alphabet" associates with each symbol of the alphabet being encoded, its corresponding leaf.

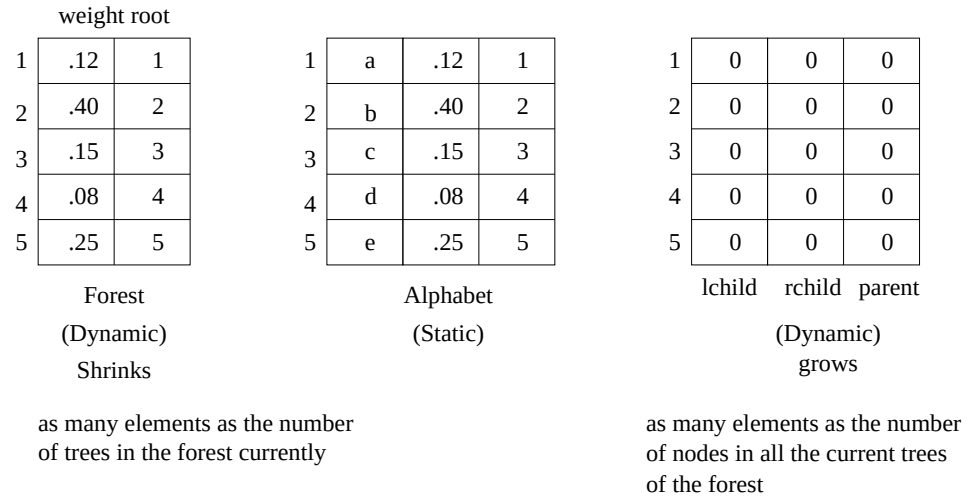


Figure 4.10: Initial state of data structures

- The array "forest" represents the collection of all trees. Initial values of these arrays are shown as follows:
- Figure 4.10 shows the above three data structures for our example.

4.3.2 Sketch of Huffman Tree Construction

1. while (there is more than one tree in the forest) *do* {
2. i = index of tree in forest with smallest weight ;
3. j = index of tree in forest with second smallest weight ;
4. create a new node with

   lchild = forest [i].root ;
   rchild = forest [j].root ;
5. replace tree i in forest by a tree with root as new node, with

   weight = forest [i] → weight + forest[j] → weight ;
6. delete tree j from forest

}

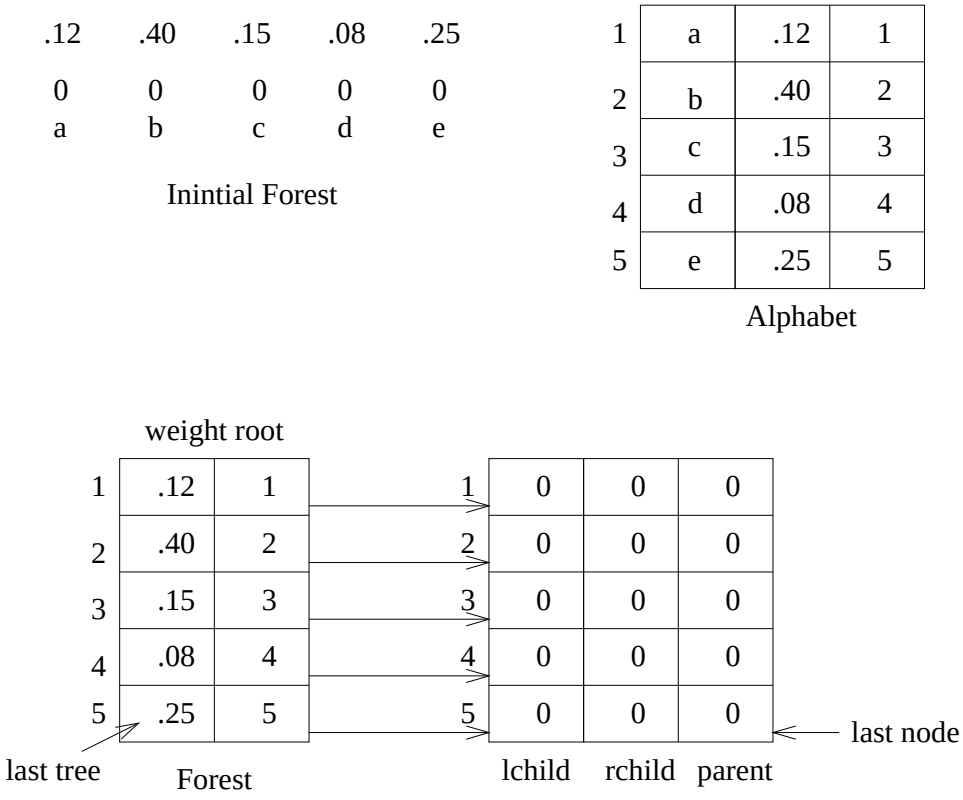
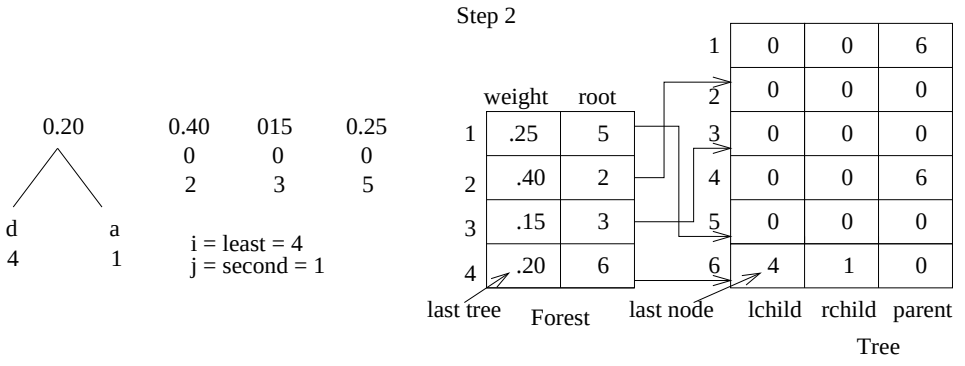


Figure 4.11: Step 1

Line (4) $\longrightarrow$ increases the number of cells of the array *tree*. Lines (5) and (6) $\longrightarrow$ decrease the number of utilized cells of forest. Figures 4.11, 4.12, and 4.13 illustrate different steps of the above algorithm for our example problem.



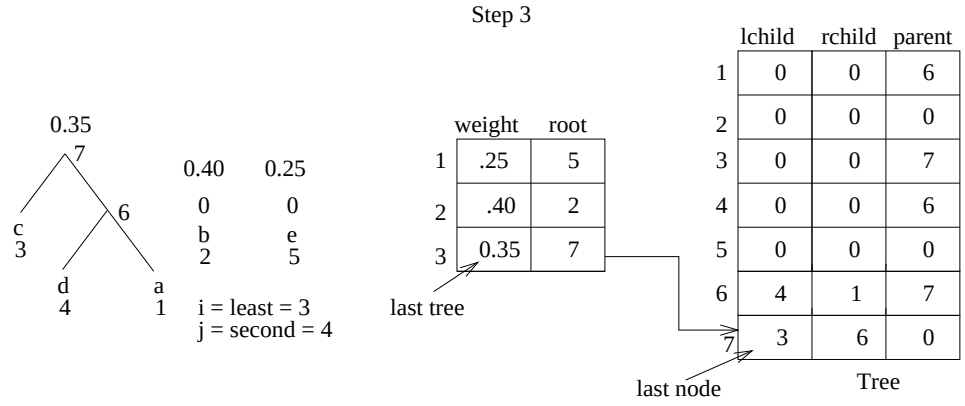


Figure 4.12: Step 2 and Step 3

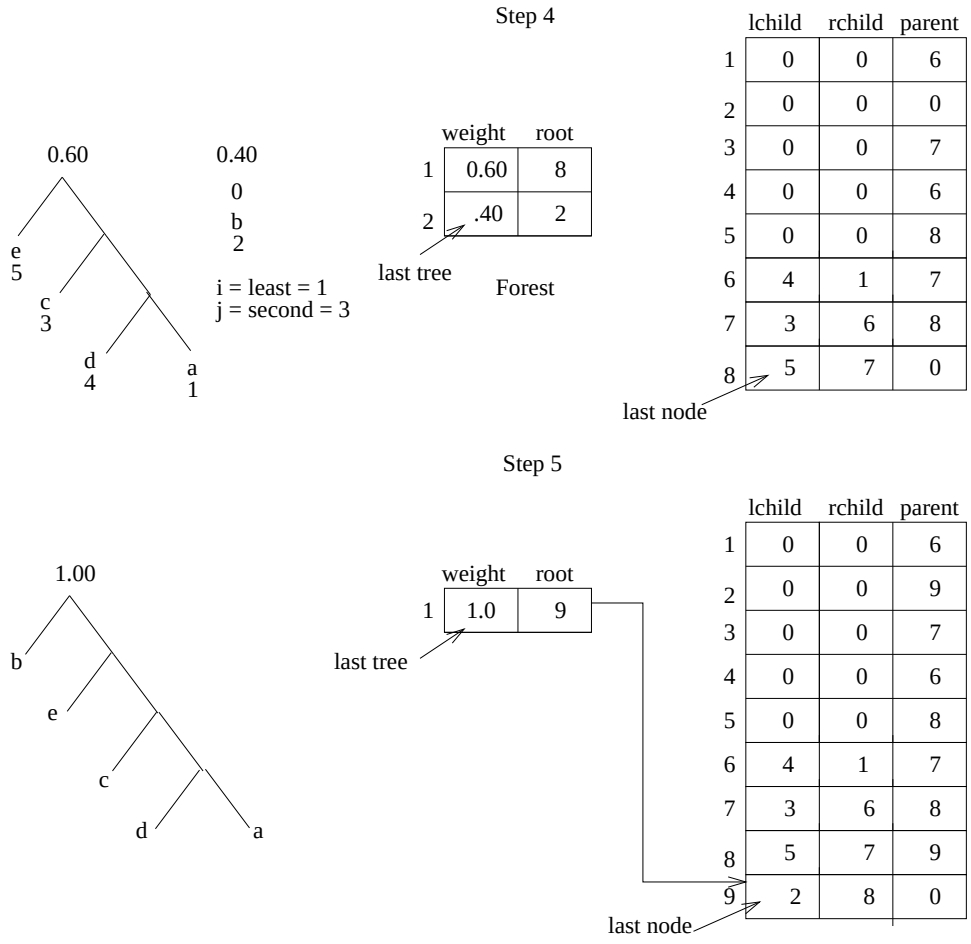


Figure 4.13: Step 4 and Step 5

4.4 Binary Search Tree

- A prominent data structure used in many systems programming applications for representing and managing dynamic sets.
- Average case complexity of Search, Insert, and Delete Operations is O(log n), where n is the number of nodes in the tree.
- **DEF:** A binary tree in which the nodes are labeled with elements of an ordered dynamic set and the following BST property is satisfied: all elements stored in the left subtree of any node x are less than the element stored at x and all elements stored in the right subtree of x are greater than the element at x.
- **An Example**: Figure 4.14 shows a binary search tree. Notice that this tree is obtained by inserting the values 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18 in that order, starting from an empty tree.
- Note that inorder traversal of a binary search tree always gives a sorted sequence of the values. This is a direct consequence of the BST property. This provides a way of sorting a given sequence of keys: first, create a BST with these keys and then do an inorder traversal of the BST so created.
- Note that the highest valued element in a BST can be found by traversing from the root in the right direction all along until a node with no right link is found (we can call that the rightmost element in the BST).
- The lowest valued element in a BST can be found by traversing from the root in the left direction all along until a node with no left link is found (we can call that the leftmost element in the BST).
- **Search** is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the key we are seeking is present, this search procedure will lead us to the key. If the key is not present, we end up in a null link.
- **Insertion** in a BST is also a straightforward operation. If we need to insert an element x, we first search for x. If x is present, there is

nothing to do. If x is not present, then our search procedure ends in a null link. It is at this position of this null link that x will be included.

- If we repeatedly insert a sorted sequence of values to form a BST, we obtain a completely skewed BST. The height of such a tree is $n-1$ if the tree has n nodes. Thus, the worst case complexity of searching or inserting an element into a BST having n nodes is $O(n)$.

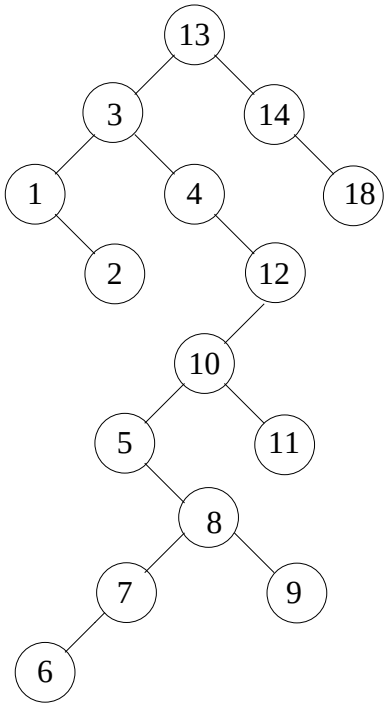


Figure 4.14: An example of a binary search tree

Deletion in BST

Let x be a value to be deleted from the BST and let X denote the node containing the value x. Deletion of an element in a BST again uses the BST property in a critical way. When we delete the node X containing x, it would create a "void" that should be filled by a suitable existing node of the BST. There are two possible candidate nodes that can fill this void, in a way that the BST property is not violated: (1). Node containing highest valued element among all descendants of left child of X. (2). Node

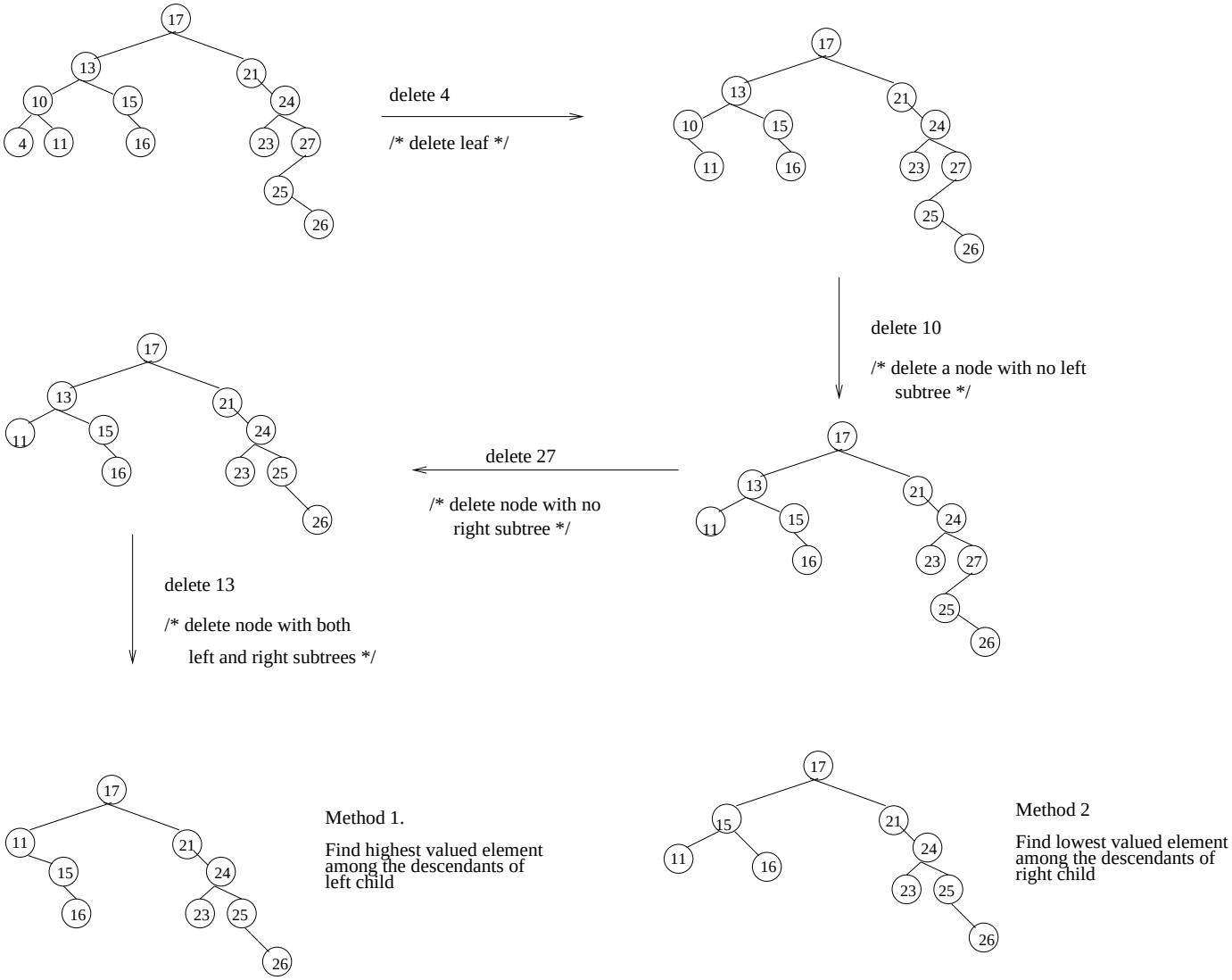


Figure 4.15: Deletion in binary search trees: An example

containing the lowest valued element among all the descendants of the right child of X. In case (1), the selected node will necessarily have a null right link which can be conveniently used in patching up the tree. In case (2), the selected node will necessarily have a null left link which can be used in patching up the tree. Figure 4.15 illustrates several scenarios for deletion in BSTs.

4.4.1 Average Case Analysis of BST Operations

RESULT

If we insert n random elements into an initially empty BST, then the average path length from the root to a node is O(log n)

- Note that the BST is formed by insertions only. Obviously the tree so formed need not be complete.
- We shall assume that all orders of n inserted elements are equally likely. This means that any of the n! permutations is equally likely to be the sequence of keys inserted.

Let

P(n) = average path length in a BST with n nodes (average number of nodes on the path from the root to a node, not necessarily a leaf)

Let

a = first element inserted. This will be the root of the BST. Also this is equally likely to be the first, second . . . , ith, . . . , or nth in the sorted order of the n elements.

Note that P(0) = 0 and P(1) = 1. Consider a fixed i, $0 \leq i \leq n-1$. If i elements are less than a, the BST will look like in Figure 4.16.

- Since all orders for the i small elements are equally likely and likewise for the $(n-i-1)$ larger elements,

  Average path length in the

  - left subtree = P(i)
  - right subtree = P($n-i-1$)

- For a fixed i, let us compute the average path length of the above tree.

  Number of probes if the element a is being sought = 1

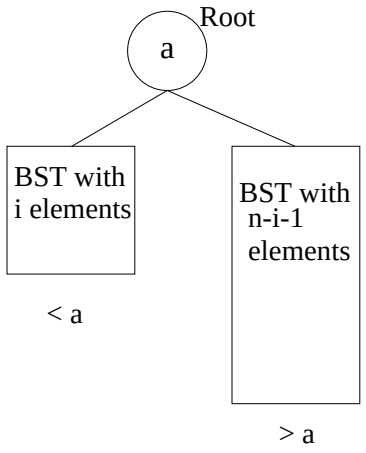


Figure 4.16: A typical binary search tree with n elements

Average number of probes if an element from the LST is sought = 1+P(i)

Average number of probes if an element from the RST is sought = 1 + P($n-i-1$)

Probability of seeking any of the n elements = $\frac{1}{n}$

Thus, average path length for a fixed i

$$\begin{aligned} &= \frac{1}{n}\{1 + i(1 + P(i)) + (n - i - 1)(1 + P(n - i - 1))\} \\ &= 1 + \frac{i}{n}P(i) + \frac{n - i - 1}{n}P(n - i - 1) \\ &= P(n, i), \quad \text{say.} \end{aligned}$$

- Observe that $P(n)$ is given by

$$P(n) = \sum_{i=0}^{n-1} \text{Prob}\{\text{LST has } i \text{ nodes }\}P(n, i)$$

Since the probability that the LST has i elements which is the same as the probability that a is the $(i + 1$th element (where $i = 0, 1, \ldots, n - 1) = \frac{1}{n}$, we have

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1} P(n, i)$$

$$
\begin{aligned}
&= \frac{1}{n}\left\{\sum_{i=0}^{n-1}\left[1+\frac{i}{n}P(i)+\frac{n-i-1}{n}P(n-i-1\right]\right\} \\
&= 1+\frac{1}{n^2}\sum_{i=0}^{n-1}[iP(i)+(n-i-1)P(n-i-1] \\
&= 1+\frac{2}{n^2}\sum_{i=0}^{n-1} iP(i)
\end{aligned}
$$

since

$$\sum_{i=0}^{n-1} iP(i) = \sum_{i=0}^{n-1}(n-i-1)P(n-i-1)$$

- Thus the average path length in a BST satisfies the recurrence:

$$P(n) = 1+\frac{2}{n^2}\sum_{i=0}^{n-1} iP(i)$$

- We shall show that $P(n) \leq 1+4\log n$, by Induction.

Basis:

$P(1)$ is known to be 1. Also the RHS = 1 for n = 1

Induction:

Let the result be true $\forall\ i < n$. We shall show that the above is true for $i = n$.

Consider

$$
\begin{aligned}
P(n) &\leq 1+\frac{2}{n^2}\sum_{i=1}^{n-1} i(1+4\log i) \\
&= 1+\frac{2}{n^2}\sum_{i=1}^{n-1} 4i\log i+\frac{2}{n^2}\sum_{i=0}^{n-1} i \\
&\leq 1+\frac{2}{n^2}\sum_{i=1}^{n-1} 4i\log i+\frac{2}{n^2}\left(\frac{n^2}{2}\right) \quad \text{since } \sum_{i=1}^{n-1} i \leq \frac{n^2}{2}
\end{aligned}
$$

Thus

$$P(n) \leq 2 + \frac{8}{n^2} \sum_{i=1}^{n-1} i \log i$$

Now

$$\begin{aligned}
\sum_{i=1}^{n-1} i \log i &= \sum_{i=1}^{\lceil \frac{n}{2} \rceil - 1} i \log i + \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} i \log i \\
&\leq \sum_{i=1}^{\lceil \frac{n}{2} \rceil - 1} i \log \frac{n}{2} + \sum_{i=\lceil \frac{n}{2} \rceil}^{n-1} i \log n \\
&\leq \frac{n^2}{8} \log \frac{n}{2} + \frac{3n^2}{8} \log n \\
&= \frac{n^2}{2} \log n - \frac{n^2}{8}
\end{aligned}$$

Therefore

$$P(n) \leq 2 + \frac{8}{n^2} \left(\frac{n^2}{2} \log - \frac{n^2}{8} \right) = 1 + 4 \log n$$

A more careful analysis can be done and it can be shown that

$$P(n) \leq 1 + 1.4 \log n$$

4.5 Splay Trees

REF. Daniel D Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, Volume 32, Number 3, pp. 652-686, 1985.

REF. Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, Volume 6, Number 2, pp. 306-318, 1985.

- These are binary search trees which are self-adjusting in the following way:

  Every time we access a node of the tree, whether for retrieval or insertion or deletion, we perform radical surgery on the tree, resulting

in the newly accessed node becoming the root of the modified tree. This surgery will ensure that nodes that are frequently accessed will never drift too far away from the root whereas inactive nodes will get pushed away farther from the root.

- Amortized complexity
  - Splay trees can become highly unbalanced so that a single access to a node of the tree can be quite expensive.
  - However, over a long sequence of accesses, the few expensive cases are averaged in with many inexpensive cases to obtain good performance.
- Does not need heights or balance factors as in AVL trees and colours as in Red-Black trees.
- The surgery on the tree is done using **rotations**, also called as splaying steps. There are six different splaying steps.
  1. Zig Rotation (Right Rotation)
  2. Zag Rotation (Left Rotation)
  3. Zig-Zag (Zig followed by Zag)
  4. Zag-Zig (Zag followed by Zig)
  5. Zig-Zig
  6. Zag-Zag
- Consider the path going from the root down to the accessed node.
  - Each time we move left going down this path, we say we "zig" and each time we move right, we say we "zag."
- Zig Rotation and Zag Rotation
  - Note that a zig rotation is the same as a right rotation whereas the zag step is the left rotation.
  - See Figure 4.17.
- Zig-Zag

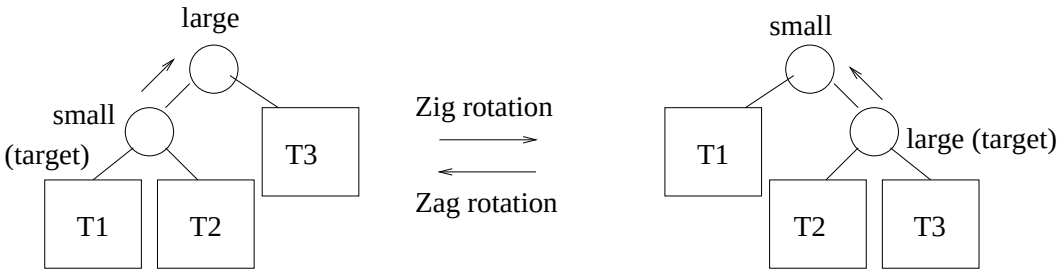


Figure 4.17: Zig rotation and zag rotation

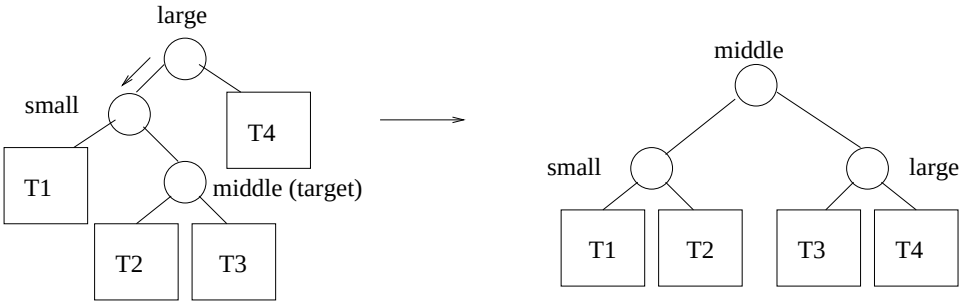


Figure 4.18: Zig-zag rotation

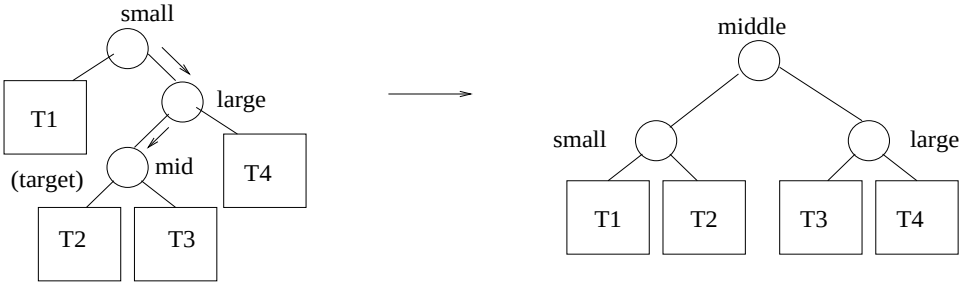


Figure 4.19: Zag-zig rotation

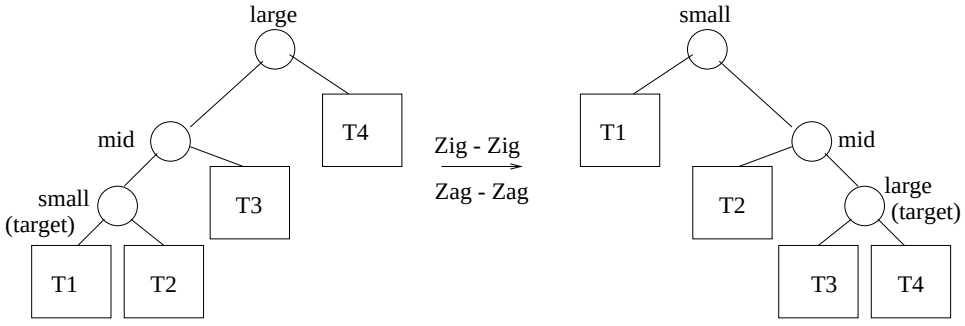


Figure 4.20: Zig-zig and zag-zag rotations

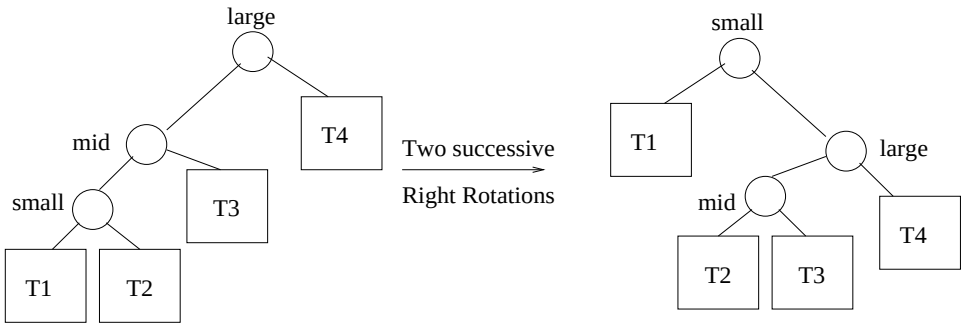


Figure 4.21: Two successive right rotations

  - This is the same as a double rotation in an AVL tree. Note that the target element is lifted up by two levels.
  - See Figure 4.18.

- Zag-Zig
  - This is also the same as a double rotation in an AVL tree.
  - Here again, the target element is lifted up by two levels.
  - See Figure 4.19.

- Zig-Zig and Zag-Zag
  - The target element is lifted up by two levels in each case.
  - Zig-Zig is different from two successive right rotations; zag-zag is different from two successive left rotations. For example, see Figures 4.20, and 4.21.

- See Figure 4.22 for an example

- The above scheme of splaying is called **bottom-up** splaying.

- In **top-down** splaying, we start from the root and as we locate the target element and move down, we splay as we go. This is more efficient.

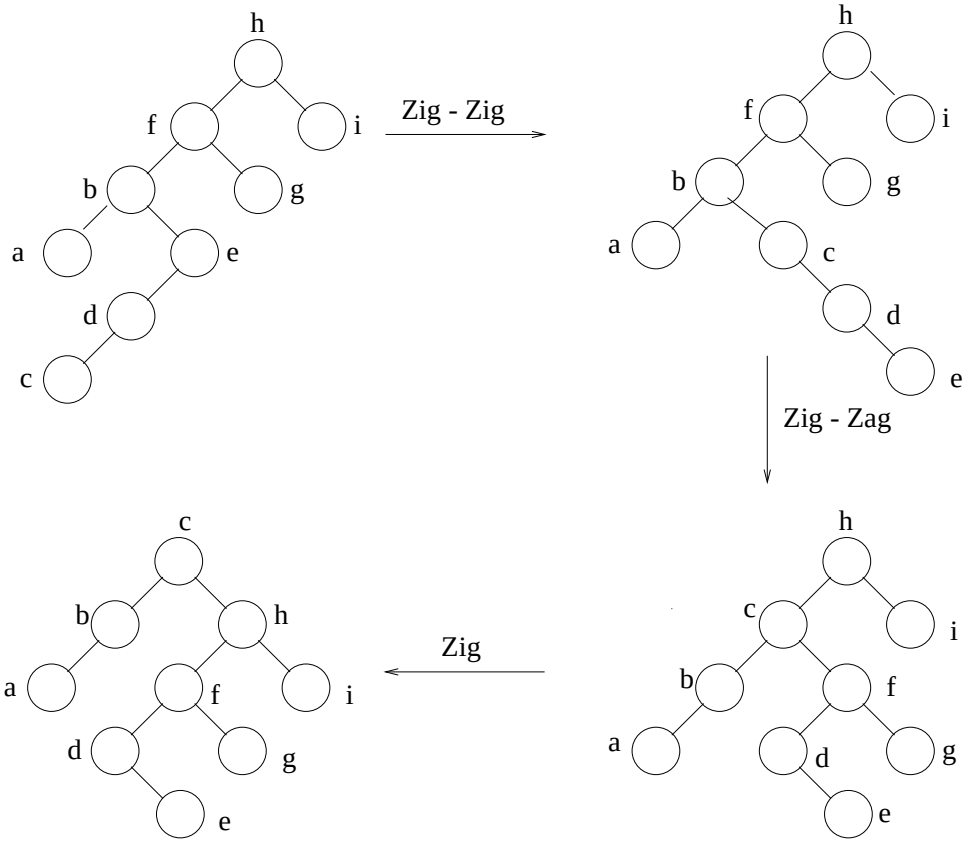


Figure 4.22: An example of splaying

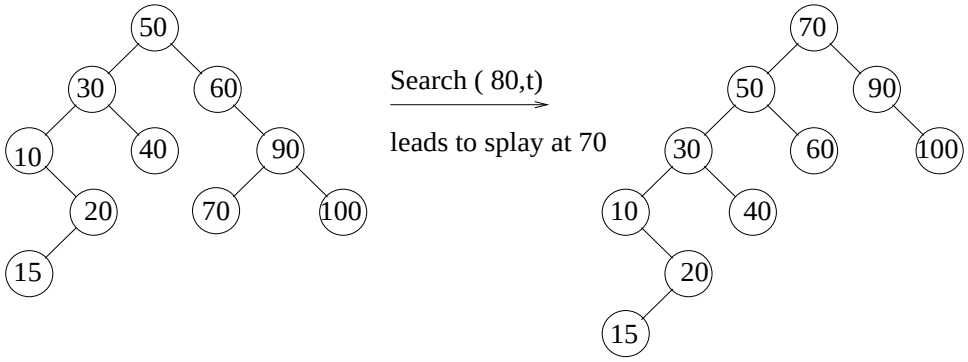


Figure 4.23: An example of searching in splay trees

4.5.1 Search, Insert, Delete in Bottom-up Splaying

Search (i, t)

If item i is in tree t, return a pointer to the node containing i; otherwise return a pointer to the null node.

- Search down the root of t, looking for i
- If the search is successful and we reach a node x containing i, we complete the search by splaying at x and returning a pointer to x
- If the search is unsuccessful, i.e., we reach the null node, we splay at the last non-null node reached during the search and return a pointer to null.
- If the tree is empty, we omit any splaying operation.

Example of an unsuccessful search: See Figure 4.23.

Insert (i, t)

- Search for i. If the search is successful then splay at the node containing i.
- If the search is unsuccessful, replace the pointer to null reached during the search by a pointer to a new node x to contain i and splay the tree at x

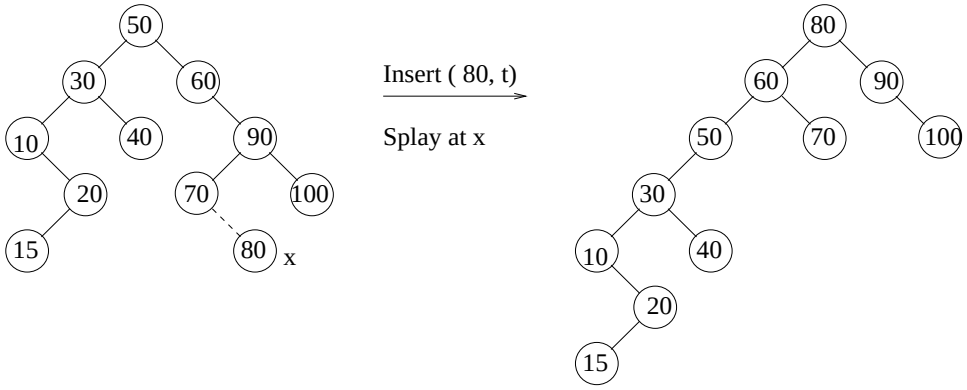


Figure 4.24: An example of an insert in a splay tree

For an example, See Figure 4.24.

Delete (i, t)

- Search for i. If the search is unsuccessful, splay at the last non-null node encountered during search.
- If the search is successful, let x be the node containing i. Assume x is not the root and let y be the parent of x. Replace x by an appropriate descendent of y in the usual fashion and then splay at y.

For an example, see Figure 4.25.

4.6 Amortized Algorithm Analysis

Amortized analysis considers a long sequence of related events rather than a single event in isolation. Amortized analysis gives a worst case estimate of the cost of a long sequence of related events.

4.6.1 Example of Sorting

A list is first sorted and then after some use, a random element is included and then sorted. If we want to do this repeatedly, then amortized analysis will indicate insertion sort rather than heapsort or quicksort.

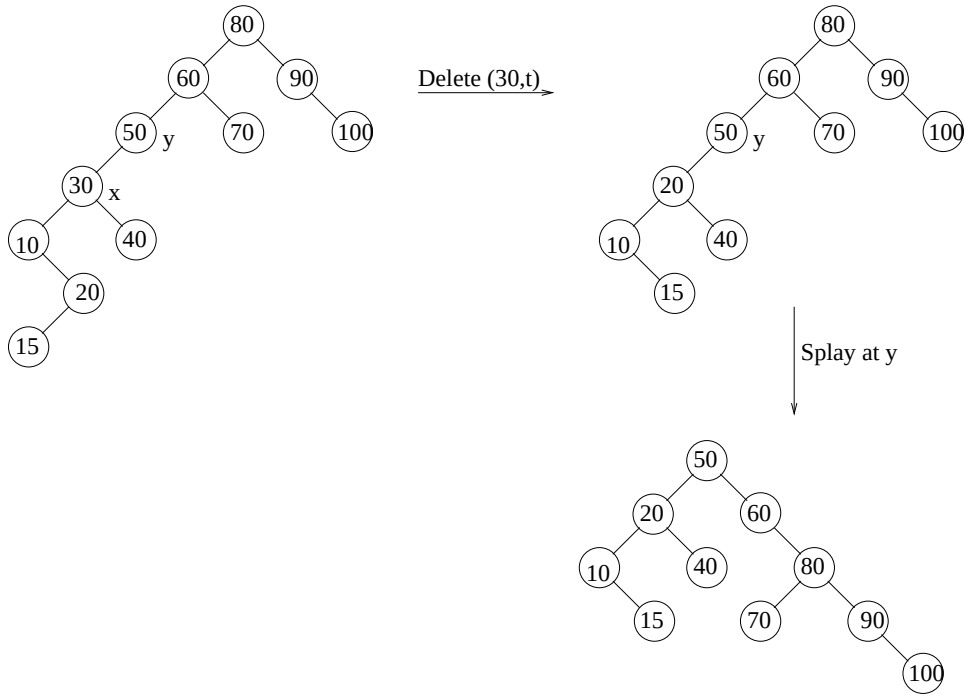


Figure 4.25: An example of a delete in a splay tree

4.6.2 Example of Tree Traversal (Inorder)

- Let the cost of visiting a vertex be the number of branches traversed to reach that vertex from the last one visited.

  Best case cost = 1 Worst case cost = n -1

- If we amortize the cost over a traversal of the entire binary tree, then the cost of going from each vertex to the next is less than 2. To see why, note that every binary tree with n vertices has exactly $n-1$ branches and a complete traversal of the tree goes over each branch exactly twice. Thus the total number of steps in a full traversal $= 2(n-1)$.

- Amortized number of steps from one vertex to the next $= \frac{2(n-1)}{n} < 2$

4.6.3 Credit Balance

- The credit function behaves like the bank balance of a well-budgeted family. It will be large when the next operation is expensive and

smaller when the next operation can be done quickly.

- Consider a sequence of m operations on a data structure. Let

$$t_i = \text{ Actual cost of operation } i \text{ for } \quad 1 \leq i \leq m$$

Let the values of the credit between balance function be

$$C_0, C_1, C_2, \ldots, C_m$$

$$\begin{aligned} C_0 &= \text{Credit balance before the first operation} \\ C_i &= \text{Credit balance after operation } \ i \end{aligned}$$

Amortized Cost

- The amortized cost a_i of each operation i is defined by

$$a_i = t_i + \underbrace{C_i - C_{i-1}}_{\begin{array}{l}\text{change in} \\ \text{the credit} \\ \text{balance during} \\ \text{operation } i\end{array}}$$

choose the credit balance function C_i so as to make the amortized costs a_i as nearly equal as possible, no matter how the actual costs may vary.

We have

$$\begin{aligned} t_i &= a_i - C_i + C_{i-1} \\ \text{Hence } \sum_{i=1}^{m} t_i &= (a_1 - C_1 + C_0) + (a_2 - C_2 + C_1) + \cdots + (a_m - C_m + C_{m-1}) \\ &= \left(\sum_{i=1}^{m} a_i\right) + (C_0 - C_m) \end{aligned}$$

Thus

$$\sum_{i=1}^{m} t_i = \left(\sum_{i=1}^{m} a_i\right) + C_0 - C_m$$

- Thus the total actual cost can be computed in terms of the amortized costs and the initial and final values of the credit balance function.

4.6.4 Example of Incrementing Binary Integers

- Consider an algorithm to continually increment a binary integer by 1. Start at the LSB (least significant bit, i.e. right most bit); while the current bit is 1, change it to 0 and move left, stopping when we reach far left or hit a 0 bit, which we change to 1 and stop.
- For the credit balance, take the total number of 1s in the binary integer:

Step i	Integer	t_i	C_i	a_i
0	0000	-	0	-
1	0001	1	1	2
2	0010	2	1	2
3	0011	1	2	2
4	0100	3	1.	2
5	0101	1	2	2
6	0110	2	2	2
7	0111	1	3	2
8	1000	4	1	2
9	1001	1	2	2
10	1010	2	2	2
11	1011	1	3	2
12	1100	3	2	2
13	1101	1	3	2
14	1110	2	3	2
15	1111	1	4	2
16	0000	4	0	0

4.6.5 Amortized Analysis of Splaying

- As a measure of complexity, we shall take the depth within the tree that the desired node has before splaying.

T	:	BST on which we are performing a splay insertion or retrieval
T_i	:	tree T as it has been transformed after step i of the splaying process, with $T_0 = T$
x	:	any node in T_i
$T_i(x)$	:	subtree of T_i with root x
$\|T_i(x)\|$	:	number of nodes of $T_i(x)$

- We consider bottom-up splaying at a node x, so that x begins somewhere in the tree T but after m splaying steps, ends up as the root of T.

- Rank

  For each step i of the splaying process and each vertex $x \in T$, we define rank at step i of x to be

$$r_i(x) = \underbrace{\log |T_i(x)|}_{\substack{\text{height of } T_i(x) \text{ if it were completely balanced.}}}$$

- Portion out the credit balance of the tree among all its vertices by requiring the following credit invariant to hold:

  For every node x of T and after every step i of splaying, node x has credit equal to its rank $r_i(x)$.

- Define the total credit balance for the tree as

$$C_i = \sum_{x \in T_i} r_i(x)$$

- If the tree is empty or contains only one node, then its credit balance is 0. As the tree grows, its credit balance increases. The credit balance should reflect the work needed to build the tree.

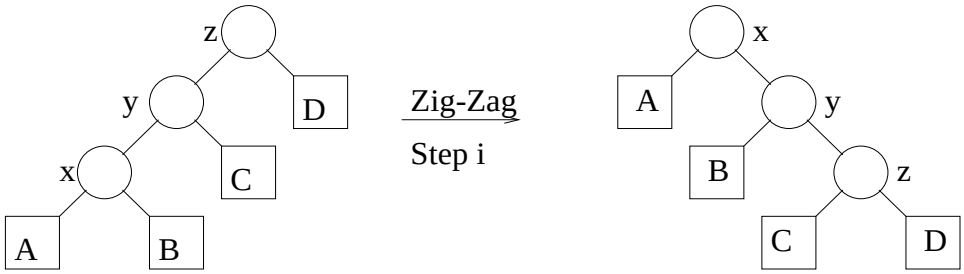


Figure 4.26: A zig-zig at the ith splaying step

- Our goal is to determine bounds on a_i that will allow us to find the cost of the splaying process, amortized over a sequence of retrievals and insertions.
- A useful Result:

  If α, β, γ are positive real numbers with $\alpha + \beta \leq \gamma$, then

  $$\log \alpha + \log \beta \leq 2 \log \gamma - 2 \qquad (*)$$

  This can be easily proved as follows:

  $$\left(\sqrt{\alpha} - \sqrt{\beta}\right)^2 \geq 0 \;\Rightarrow \sqrt{\alpha\beta} \leq \frac{\alpha + \beta}{2} \Rightarrow \sqrt{\alpha\beta} \leq \frac{\gamma}{2}$$
  $$\Rightarrow \log \alpha + \log \beta \leq 2 \log \gamma - 2$$

Result 1

If the i^{th} splaying step is a zig-zig or a zag-zag step at node x, then its amortized complexity a_i satisfies

$$a_i < 3(r_i(x) - r_{i-1}(x))$$

See Figure 4.26.

- The actual complexity t_i of a zig-zig or a zag-zag is 2 units
- Only the sizes of the subtrees rooted at x, y, z change in this step. Hence, all terms in the summation defining C_i cancel against those

for C_{i-1} except those indicated below

$$\begin{aligned} a_i &= t_i + C_i - C_{i-1} \\ &= 2 + r_i(x) + r_i(y) + r_i(z) \\ &\quad -r_{i-1}(x) - r_{i-1}(y) + r_{i-1}(z) \\ &= 2 + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) \qquad (\bullet) \end{aligned}$$

Since $|T_i(x)| = |T_{i-1}(z)|$ as the subtree rooted at z before the splaying step has the same size as that rooted at x after the step.

Let

$\alpha = |T_{i-1}(x)|, \beta = |T_i(z)|$, and $\gamma = |T_i(x)|$. Then observe that $\alpha+\beta \leq \gamma$. This is because

$$\begin{aligned} T_{i-1}(x) &\text{ contains components } x, A, B \\ T_i(z) &\text{ contains components } z, C, D \\ T_i(x) &\text{ contains all these components (plus } y \text{ besides).} \end{aligned}$$

Thus by $(*)$

$$\begin{aligned} & r_{i-1}(x) + r_i(z) \leq 2r_i(x) - 2 \\ \Rightarrow\ & 2r_i(x) - r_{i-1}(x) - r_i(z) - 2 \geq 0 \end{aligned}$$

Adding this to $(\bullet)$, we get

$$a_i \leq 2r_i(x) - 2r_{i-1}(x) + r_i(y) - r_{i-1}(y)$$

Before step i, y is the parent of x so that

$$|T_{i-1}(y)| > |T_{i-1}(x)|$$

After step i, x is the parent of y so that

$$|T_i(x)| > |T_i(y)|$$

Taking logarithms we have

$$r_{i-1}(y) > r_{i-1}(x) \text{ and } r_i(x) > r_i(y)$$

Thus we obtain

$$a_i < 3r_i(x) - 3r_{i-1}(x)$$

Similarly, we can show Results 2 and 3 below:

Result 2

If the i^{th} splaying step is a zig-zag or zag-zig at node x, then its amortized complexity a_i satisfies

$$a_i < 2(r_i(x) - r_{i-1}(x))$$

Result 3

If the i^{th} splaying step is a zig or a zag, then

$$a_i < 1 + (r_i(x) - r_{i-1}(x))$$

Result 4

The total amortized cost over all the splaying steps can be computed by adding the costs of all the splay steps. If there are k such steps, only the last can be a zig or zag and the others are all zig-zag or zig-zag or zag-zig or zag-zag. Since $(*)$ provides the weaker bound, we get,

$$\begin{aligned}
\sum_{i=1}^{k} a_i &= \sum_{i=1}^{k-1} a_i + a_k \\
&\leq \sum_{i-1}^{k-1} 3(r_i(x) - r_{i-1}(x)) + (1 + 3r_k(x) - 3r_{k-1}(x)) \\
&= 1 + 3r_k(x) - 3r_0(x) \\
&\leq 1 + 3r_k(x) \\
&= 1 + 3\log n
\end{aligned}$$

Thus the amortized cost of an insertion or retrieval with splaying in a BST with n nodes does not exceed $1 + 3 \log n$ upward moves of the target node in the tree.

Result 5

Now consider a long sequence of m splay insertions or retrievals. We seek to compute the actual cost of all these m operations.

Let T_j and A_j be the actual cost and amortized cost of the j^{th} operation, where $j = 1, 2, \ldots, m$, now represents a typical insertion or retrieval operation.

We have

$$\sum_{j=1}^{m} T_j = \left(\sum_{j=1}^{m} A_j \right) + C_O - C_m$$

If the tree never has more than n nodes, then C_0 and C_m lie between 0 and $\log n$, so that

$$C_0 - C_m \leq \log n$$

Also we know that

$$A_j \leq 1 + 3 \log n \quad \text{for } j = 1, 2, \ldots, m$$

Thus the above becomes

$$\sum_{j=1}^{m} T_j = m(1 + 3 \log n) + \log n$$

The above gives the total actual time of m insertions or retrievals on a tree that never has more than n nodes.

This gives $O(\log n)$ actual running time for each operation.

4.7 To Probe Further

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

2. Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996. Indian Edition published by Prentice Hall of India, 1998.

3. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.

4. Donald E Knuth. *Fundamental Algorithms*, Volume 1 of The Art of Computer Programming, Addison-Wesley, 1968, Second Edition, 1973.

5. Robert L. Kruse, Bruce P. Leung, and Clovis L. Tondo. *Data Structures and Program design in C*. Prentice Hall, 1991. Indian Edition published by Prentice Hall of India, 1999.

6. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.

7. Duane A. Bailey. *Java Structures: Data Structures in Java for the Principled Programmer*. McGraw-Hill International Edition, 1999.

8. Ellis Horowitz and Sartaz Sahni. *Fundamentals of Data structures*. Galgotia Publications, New Delhi, 1984.

9. Donald E Knuth. *Seminumerical Algorithms*. Volume 2 of The Art of Computer Programming, Addison-Wesley, 1969, Second Edition, 1981.

10. Donald E. Knuth. *Sorting and Searching*, Volume 3 of The Art of Computer Programming, Addison-Wesley, 1973.

11. Y. Langsam, M.J. Augenstein, and A.M. Tenenbaum. *Data Structures using C and C++*. Second Edition, Prentice Hall, 1996. Indian Edition published by Prentice Hall of India, 2000.

12. Kurt Mehlhorn. *Sorting and Searching.* Volume 1 of *Data Structures and Algorithms.* Springer-Verlag, 1984.

13. Sataj Sahni. *Data Structures, Algorithms, and Applications in Java.* McGraw-Hill Higher Education, 2000.

14. Thomas A. Standish. *Data Structures in Java.* Addison-Wesley, 1998. Indian Edition published by Addison Wesley Longman, 2000.

15. Nicklaus Wirth. *Data Structures + Algorithms = Programs.* Prentice-Hall, Englewood Cliffs. 1975.

16. David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* Volume 40, Number 9, pp. 1098-1101, 1952.

17. Daniel D Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, Volume 32, Number 3, pp 652-686, 1985.

18. Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, Volume 6, Number 2, pp.306-318, 1985.

4.8 Problems

4.8.1 General Trees

1. Show in any binary tree that the number of leaves is one more than the number of nodes of degree two.

2. The degree of a node in a tree is the number of children the node has. If a tree has n_1 nodes of degree 1, n_2 nodes of degree 2, ..., n_m nodes of degree m, compute the number of leaves in the tree in terms of $n_1, n_2, \ldots, n_m$.

3. Show that if T is a k-ary tree with n nodes, each having a fixed size, then $nk+1-n$ of the nk link fields are null links ($n \geq 1$).

4. Provide an efficient $O(n)$ time algorithm to determine the height of a tree containing n nodes, represented using parent links. You may build any other supporting data structures, if needed.

5. Associate a weight $w(x) = 2^{-d}$ with each leaf x of depth d in a binary tree T. Show that

$$\sum_x w(x) \leq 1$$

where the sum is taken over all leaves x in T.

6. Consider a complete binary tree with an odd number of nodes. Let n be the number of internal nodes (non-leaves) in the tree. Define the internal path length, I, as the sum, taken over all the internal nodes of the tree, of the depth of each node. Likewise, define the external pathlength, E, as the sum, taken over all the leaves of the tree, of the depth of each leaf. Show that $E = I + 2n$.

7. What is the optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a	b	c	d	e	f	g	h
1	1	2	3	5	8	13	21

Write down the resulting Huffman tree. Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

4.8.2 Binary Search Trees

1. Suppose that we have numbers between 1 and 1000 in a binary search tree and want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined?

   (a) 2, 252, 401, 398, 330, 344, 397, 363.

   (b) 924, 220, 911, 244, 898, 258, 362, 363.

   (c) 925, 202, 911, 240, 912, 245, 363.

   (d) 2, 399, 387, 219, 266, 382, 381, 278, 363.

   (e) 935, 278, 347, 621, 299, 392, 358, 363.

2. In a binary search tree, given a key x, define $successor(x)$ as the key which is the successor of x in the sorted order determined by an inorder tree walk. Define $predecessor(x)$ similarly. Explain how, given x, its successor and predecessor may be found.

3. A binary search tree is constructed by inserting the key values 1, 2, 3, 4, 5, 6, 7 in some order specified by a permutation of 1, ..., 7, into an initially empty tree. Which of these permutations will lead to a complete binary search tree (1 node at level 1, 2 nodes at level 2, and 4 nodes at level 3)?

4. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Investigate if the following is true: $a \leq b \leq c \; \forall \; a \in A$; $b \in B$; $c \in C$. Give a proof or a counter-example as the case may be.

5. Is the operation of deletion in a binary search tree *commutative* in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is so or give a counter-example.

6. Devise an algorithm that takes two values, a and b such that $a \leq b$, and visits all keys x in a BST such that $a \leq x \leq b$. The running time of your algorithm should be $O(N + \log n)$, where N is the number of keys visited and n is the number of keys in the tree.

7. A random binary search tree having n nodes, where $n = 2^k - 1$, for some positive integer k, is to be reorganized into a perfectly balanced binary search tree. Outline an efficient algorithm for this task. The algorithm should not use any memory locations other than those already used by the random binary search tree.

8. Consider three keys, k_1, k_2, k_3, such that $k_1 < k_2 < k_3$. A binary search tree is constructed with these three keys. Depending on the order in which the keys are inserted, five different binary search trees are possible.

   (a) Write down the five binary search trees.

   (b) Let p, q, r be the probabilities of probing for k_1, k_2, k_3, respectively in the given binary search trees ($p+q+r = 1$). Compute the average number of comparisons (or probes) on a successful search for each of the above five binary search trees.

   (c) Define the *optimum* binary search tree as the one for which the average number of comparisons on a successful search is minimum. Determine the range of values of p, q, r, for which the completely balanced search tree is the optimum search tree.

9. Given a set of 31 names, each containing 10 uppercase alphabets, we wish to set up a data structure that would lead to efficient average case performance of successful and unsuccessful searches on this set. It is known that not more than 5 names start with the same alphabet. Also, assume that successful and unsuccessful searches are equally likely and that in the event of a successful search, it is equally likely that any of the 31 names was searched for. The two likely choices for the data structure are:

   - A closed hash table with 130 locations where each location can accommodate one name.
   - A full binary search tree of height 4, where each of the 31 nodes contains a name and lexicographic ordering is used to set up the BST.

Answer the following questions:

(a) What is the hashing function you would use for the closed hash table?

(b) Assume that the computation of the hash function above takes the same time as comparing two given names. Now, compute the average case running time of a search operation using the hash table. Ignore the time for setting up the hash table.

(c) Compute the average case running time of a search operation using the BST. Ignore the time for setting up the BST.

4.8.3 Splay Trees

1. Complete the proof of the following lemmas in the amortized analysis of splay trees:

   (a) If the ith splaying step is a zig-zag step or a zag-zig step at node x, then its amortized complexity a_i satisfies $a_i \leq 2(r_i(x) - r_{i-1}(x))$.

   (b) If the ith splaying step is a zig step or a zag step at node x, then its amortized complexity a_i satisfies $a_i \leq 1 + (r_i(x) - r_{i-1}(x))$.

2. Define a rank function $r(x)$ for the nodes of any binary tree as follows: If x is the root, then $r(x) = 0$; If x is the left child of y, then $r(x) = r(y) - 1$; If x is the right child of y, then $r(x) = r(y)+1$. Define the credit balance of a tree during a traversal to be the rank of the node being visited. For several binary trees, determine the ranks at each node and prepare a table showing the actual cost, credit balance, and amortized cost (in edges traversed) of each step of an inorder traversal.

3. Generalize the amortized analysis for incrementing four digit binary integers to n digit binary integers.

4. A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. Use a suitable credit function and compute the amortized cost per operation.

5. Keys $1, 2, \ldots, n$ are inserted in that order into an empty splay tree, using splay insertions. What will be the resulting tree? Also write down the splay tree that results when the key n is deleted, using splay deletion.

6. Consider a sequence of m accesses (search operations) to an n-node splay tree. Show that the total running time of this sequence of operations is $O((m + n) \log n + m)$.

4.9 Programming Assignments

4.9.1 Huffman Coding

1. **Problem**: Given a set of alphabets and their relative frequencies, find the Huffman binary code using binary trees and an optimal ternary code using ternary trees.
2. **Reading**: Read pages 94-102 of "Data Structures and Algorithms" by Aho, Hopcroft, and Ullman.
3. **Input**: The number of alphabets and relative frequencies of those, for example,

   8 .2 .02 .3 .05 .03 .15 .22 .03

   Assume the alphabets as 1, 2, 3, ..., without loss of generality.
4. **Output**: For each alphabet, print the Huffman binary code and an optimal ternary code. Also print the average length of code.
5. **Data structures**: For binary trees, use the data structures suggested by Aho, Hopcroft, and Ullman. For ternary trees, use similar data structures based on leftmostchild-right sibling representation with nodes organized into cellspace.
6. **Programming language**: Use C or C++or Java. Best practices in programming should be adhered to.

4.9.2 Comparison of Hash Tables and Binary Search Trees

The objective of this assignment is to compare the performance of open hash tables, closed hash tables, and binary search trees in implementing search, insert, and delete operations in dynamic sets. You can initially implement using C but the intent is to use C++ and object-oriented principles.

Generation of Keys

Assume that your keys are character strings of length 10 obtained by scanning a text file. Call these as tokens. Define a token as any string that appears between successive occurrences of a forbidden character, where the forbidden set of characters is given by:

$$F = \{\text{comma}, \text{period}, \text{space}\}$$

If a string has less than 10 characters, make it up into a string of exactly 10 characters by including an appropriate number of trailing *'s. On the other hand, if the current string has more than 10 characters, truncate it to have the first ten characters only.

From the individual character strings (from now on called as **tokens**), generate a positive integer (from now on called as **keys**) by summing up the ASCII values of the characters in the particular token. Use this integer key in the hashing functions. However, remember that the original token is a character string and this is the one to be stored in the data structure.

Methods to be Evaluated

The following four schemes are to be evaluated.

1. Open hashing with multiplication method for hashing and unsorted lists for chaining
2. Open hashing with multiplication method for hashing and sorted lists for chaining
3. Closed hashing with linear probing
4. Binary search trees

Hashing Functions

For the sake of uniformity, use the following hashing functions only. In the following, m is the hash table size (that is, the possible hash values are, $0, 1, \ldots, m-1$), and x is an integer key.

1. **Multiplication Method**

$$h(x) = \text{Floor}(m * \text{Fraction}(k * x))$$

   where $k = \frac{\sqrt{(5)}-1}{2}$, the Golden Ratio.

2. **Linear Probing**

$$h(x, i) = (h(x, 0) + i) \ \textbf{mod} \ m; i = 0, \ldots, m-1$$

Inputs to the Program

The possible inputs to the program are:

- m: Hash table size. Several values could be given here and the experiments are to be repeated for all values specified. This input has no significance for binary search trees.

- n: The initial number of insertions of distinct strings to be made for setting up a data structure.
- M: This is a subset of the set $\{1, 2, 3, 4\}$ indicating the set of methods to be investigated.
- I: This is a decimal number from which a ternary string is to be generated (namely the radix-3 representation of I). In this representation, assume that a **0** represents the **search** operation, a **1** represents the **insert** operation, and a **2** represents the **delete** operation.
- A text file, from which the tokens are to be picked up for setting up and experimenting with the data structure

What should the Program Do?

1. For each element of M and for each value of m, do the following.
2. Scan the given text file and as you scan, insert the first n distinct strings scanned into an initially empty data structure.
3. Now scan the rest of the text file token by token, searching for it or inserting it or deleting it, as dictated by the radix-3 representation of I. Note that the most significant digit is to be considered first while doing this and you proceed from left to right in the radix-3 representation. For each individual operation, keep track of the number of probes.
4. Compute the average number of probes for a typical successful search, unsuccessful search, insert, and delete.

4.9.3 Comparison of Skip Lists and Splay Trees

The objective of this assignment is to compare the performance of splay trees and skip lists in implementing search, insert, and delete operations in dictionaries. The context is provided by symbol table operations in a C compiler.

What is Required?

Assume that your keys are identifiers in a C program (strings that are different from special symbols). Use LEX to design a scanner (lexical analyzer) that will scan the C program and split it into a set of identifiers. Each identifier is a regular expression. Call these identifiers as **tokens** from henceforth. The tokens are to be put it into a symbol table. The following two data structures are to be evaluated.

1. Splay trees with bottom-up splaying

2. Skip lists

Inputs to the Program

The inputs to the program are:

- A C program of sufficient size
- n: The initial number of insertions of distinct tokens to be carried out to set up a base data structure.

What should the Program Do?

Do the following steps for splay tree (skip list):

1. Scan the given C program and as you scan, insert the first n distinct tokens into an initially empty data structure.
2. Now scan the rest of the C program token by token, inserting or deleting the token, by tossing a fair coin. For each individual operation, keep track of the number of probes and number of rotations (if applicable). Count the number of elementary operations for each insert or delete (comparisons, assignments of pointers, etc.)
3. Compute the average number of elementary operations for insert and delete.

Programming Language

C++ or Java is recommended. A serious attempt should be made to use ideas of data abstraction, encapsulation, modularity, inheritance, and polymorphism.

Chapter 5

Balanced Trees

5.1 AVL Trees

Also called as: Height Balanced Binary Search Trees.

REF. G.M. Adel'son-Vel'skii and E.M. Landis. An algorithm for the organization of information. *Soviet Mathematics Monthly* Volume 3, pp.1259-1263, 1962.

Search, Insertion, and Deletion can be implemented in worst case O(log n) time

Definition

An AVL tree is a binary search tree in which

1. the heights of the right subtree and left subtree of the root differ by at most 1
2. the left subtree and the right subtree are themselves AVL trees
3. A node is said to be

left-high	if the left subtree has greater height	/
right-high	if the right subtree has greater height	\
equal	if the heights of the LST and RST are the same	–

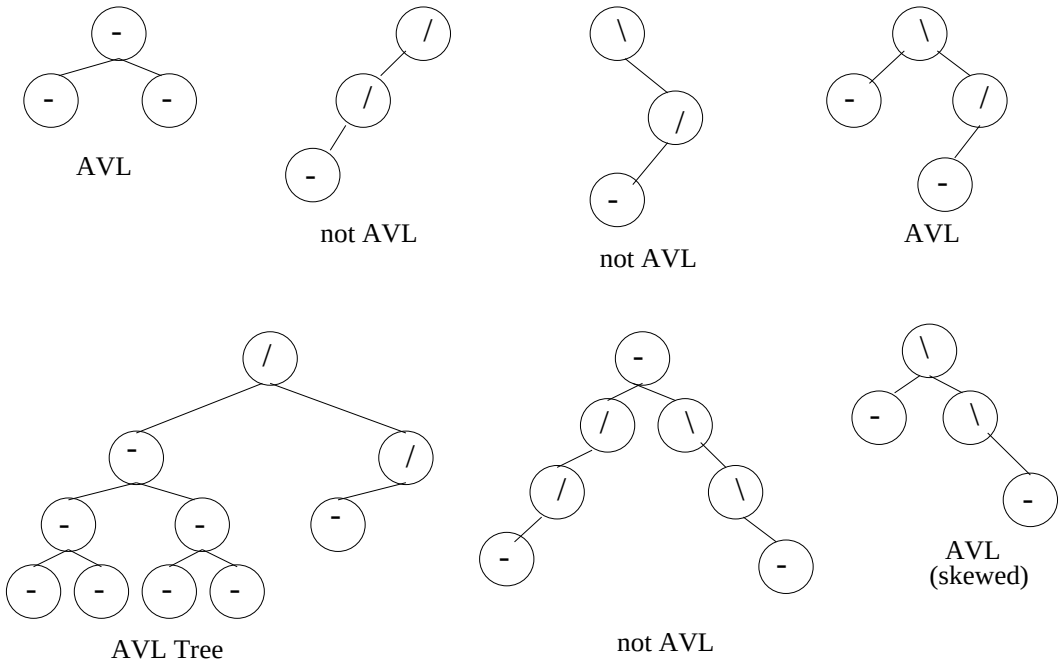


Figure 5.1: Examples of AVL trees

Examples: Several examples of AVL trees are shown in Figure 5.1.

5.1.1 Maximum Height of an AVL Tree

What is the maximum height of an AVL tree having exactly n nodes? To answer this question, we will pose the following question:

What is the minimum number of nodes (sparsest possible AVL tree) an AVL tree of height h can have ?

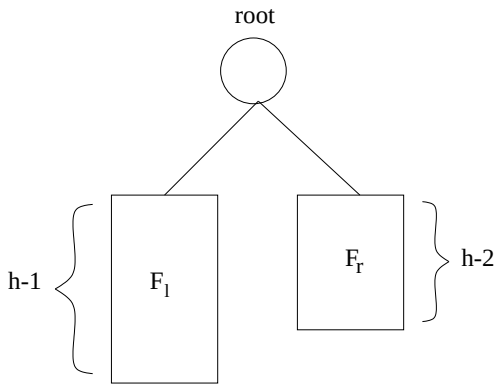


Figure 5.2: An AVL tree with height h

Let F_h be an AVL tree of height h, having the minimum number of nodes. F_h can be visualized as in Figure 5.2.

Let F_l and F_r be AVL trees which are the left subtree and right subtree, respectively, of F_h. Then F_l or F_r must have height h-2.

Suppose F_l has height h–1 so that F_r has height h–2. Note that F_r has to be an AVL tree having the minimum number of nodes among all AVL trees with height of h–1. Similarly, F_r will have the minimum number of nodes among all AVL trees of height h—2. Thus we have

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1$$

where $|F_r|$ denotes the number of nodes in F_r. Such trees are called **Fibonacci trees**. See Figure 5.3. Some Fibonacci trees are shown in Figure 4.20. Note that $|F_0| = 1$ and $|F_1| = 2$.

Adding 1 to both sides, we get

$$|F_h| + 1 = (|F_{h-1}| + 1) + (|F_{h-2}| + 1)$$

Thus the numbers $|F_h| + 1$ are Fibonacci numbers. Using the approximate formula for Fibonacci numbers, we get

$$|F_h| + 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+3}$$

$$\Rightarrow h \approx 1.44 \log |F_n|$$

$\Rightarrow$ The sparsest possible AVL tree with n nodes has height

$$h \approx 1.44 \log n$$

$\Rightarrow$ The worst case height of an AVL tree with n nodes is

$$1.44 \log n$$

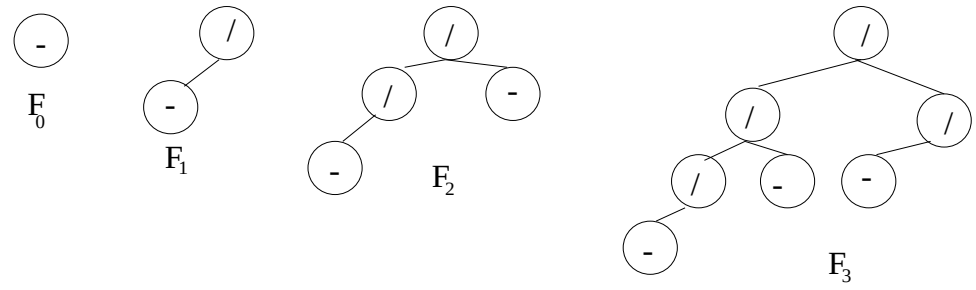


Figure 5.3: Fibonacci trees

5.1.2 AVL Trees: Insertions and Deletions

- While inserting a new node or deleting an existing node, the resulting tree may violate the (stringent) AVL property. To reinstate the AVL property, we use **rotations**. See Figure 5.4.

Rotation in a BST

- Left rotation and right rotation can be realized by a three-way rotation of pointers.

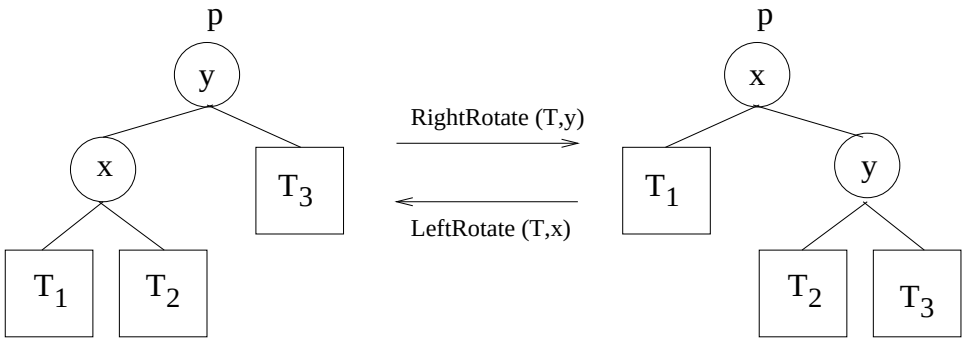


Figure 5.4: Rotations in a binary search tree

- Left Rotation:

temp	=	p → right ;
p → right	=	temp → left ;
temp → left	=	p ;
p	=	temp ;

- Left rotation and right rotation preserve
  - * BST property
  - * Inorder ordering of keys

Problem Scenarios in AVL Tree Insertions

  - – left subtree of node has degree higher by ≥ 2
    - * left child of node is left high (A)
    - * left child or node is right high (B)
  - – right subtree has degree higher by ≥ 2
    - * right child of node is left high (C)
    - * right child or node is right high (D)

- The AVL tree property may be violated at any node, not necessarily the root. Fixing the AVL property involves doing a series of single or double rotations.
- Double rotation involves a left rotation followed or preceded by a right rotation.
- In an AVL tree of height h, no more than $\lceil \frac{h}{2} \rceil$ rotations are required to fix the AVL property.

Insertion: Problem Scenario 1: (Scenario D)

Scenario A is symmetrical to the above. See Figure 5.5.

Insertion: Problem Scenario 2: (Scenario C)

Scenario B is symmetrical to this. See Figure 5.6.

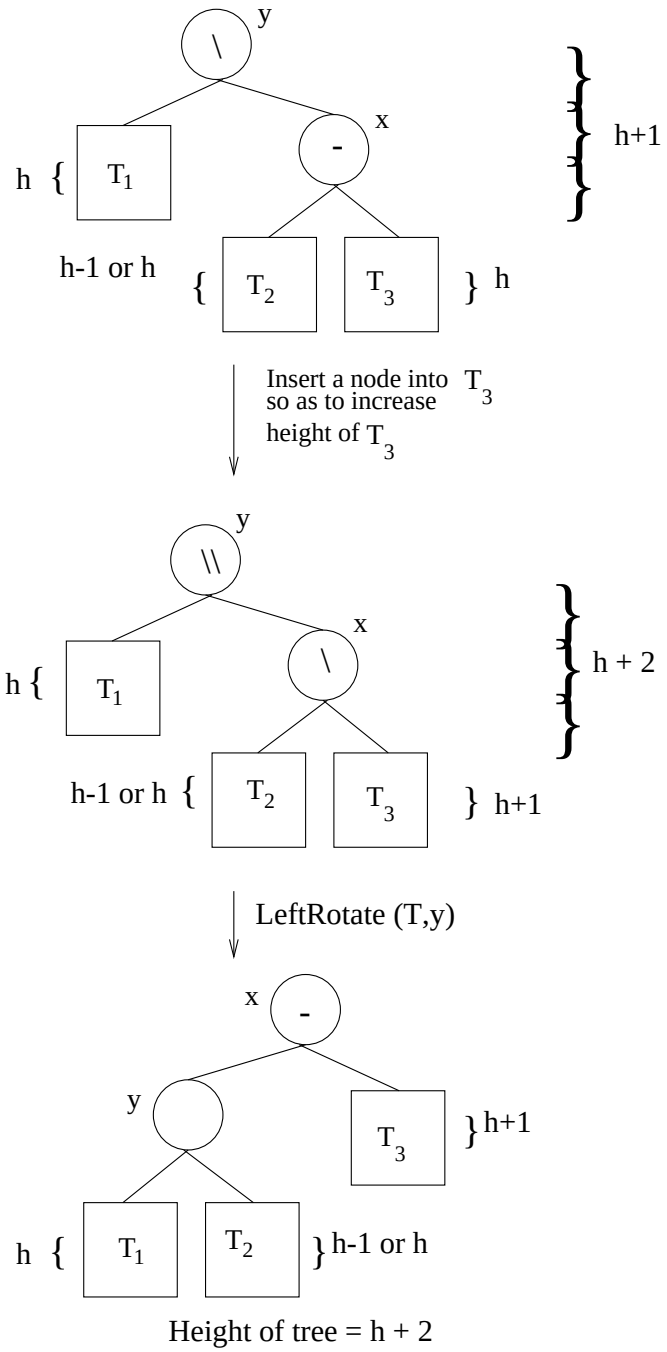


Figure 5.5: Insertion in AVL trees: Scenario **D**

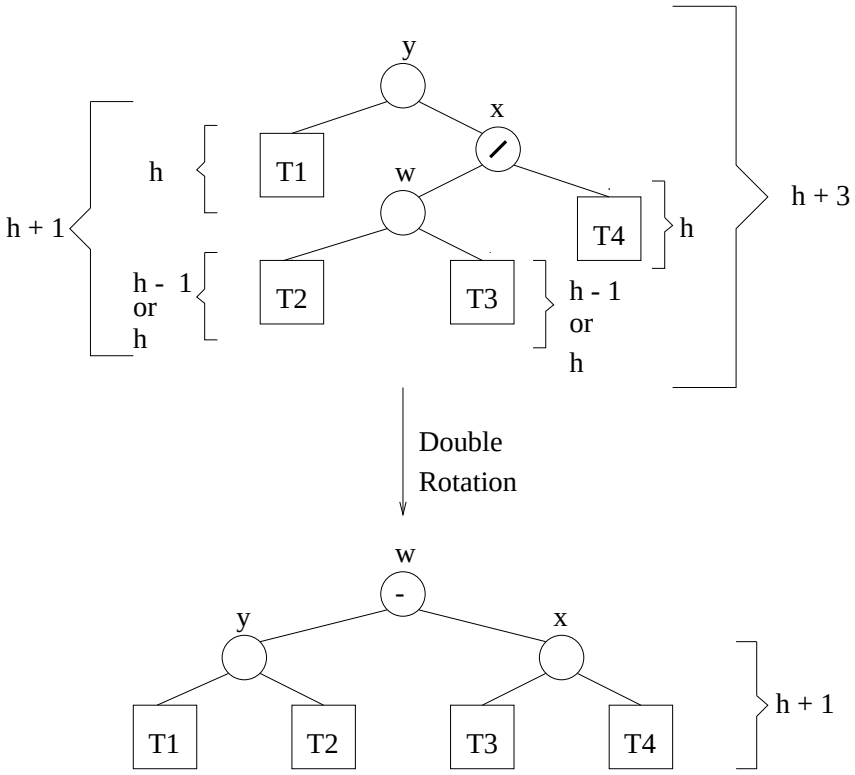


Figure 5.6: Insertion in AVL trees: Scenario **C**

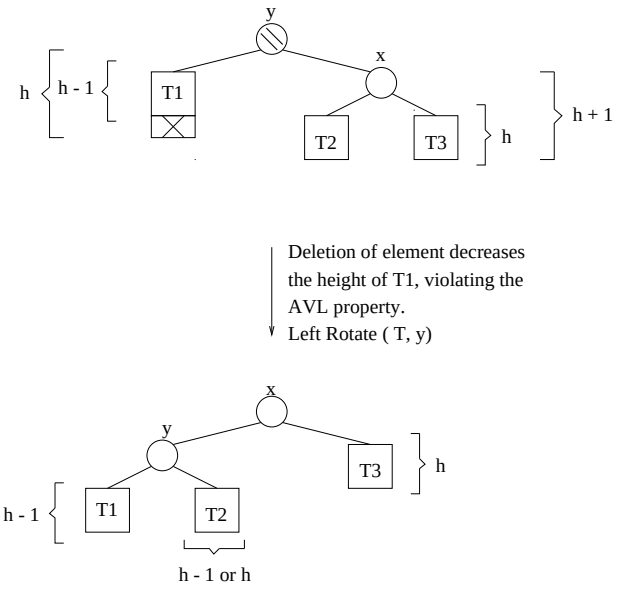


Figure 5.7: Deletion in AVL trees: Scenario **1**

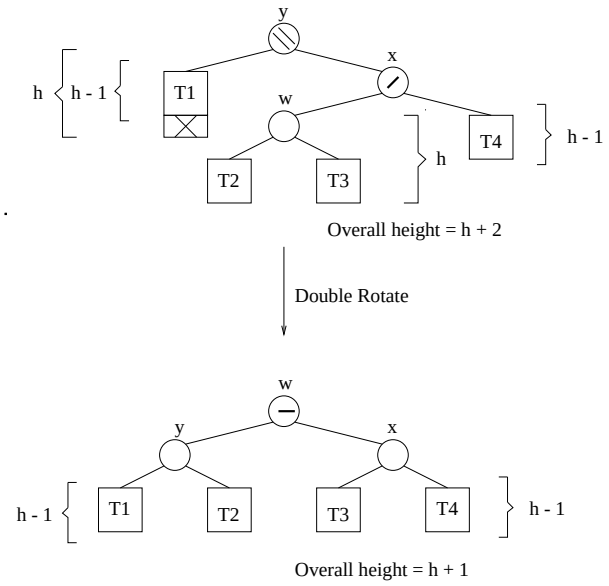


Figure 5.8: Deletion in AVL trees: Scenario **2**

Deletion: Problem Scenario 1:

Depending on the original height of T_2, the height of the tree will be either unchanged (height of $T_2 = h$) or gets reduced (if height of $T_2 = h - 1$). See Figure 5.7.

There is a scenario symmetric to this.

Deletion: Problem Scenario 2:

See Figure 5.8. As usual, there is a symmetric scenario.

5.2 Red-Black Trees

- Binary search trees where no path from the root to a leaf is more than twice as long as any other path from the root to a leaf.

REF. R. Bayer. Symmetric binary B-trees: Data Structures and maintenance algorithms, *Acta Informatica*, Volume 1, pp.290-306, 1972.

Definition

A BST is called an RBT if it satisfies the following four properties.

1. Every node is either red or black
2. Every leaf is black (Missing node property)
3. If a node is red, then both its children are black
   (RED constraint)
4. Every simple path from a node to a descendent leaf contains the same number of black nodes
   (BLACK constraint)

Black-Height of a Node

- Number of black nodes on any path from, but not including a node x to a leaf is called the black height of x and is denoted by bh(x).
- Black height of an RBT is the black height of its root.
- Each node of a red black tree contains the following fields.

colour	key	left	right	parent

- If a child of a node does not exist, the corresponding pointer field will be NULL and these are regarded as leaves.
- Only the internal nodes will have key values associated with them.

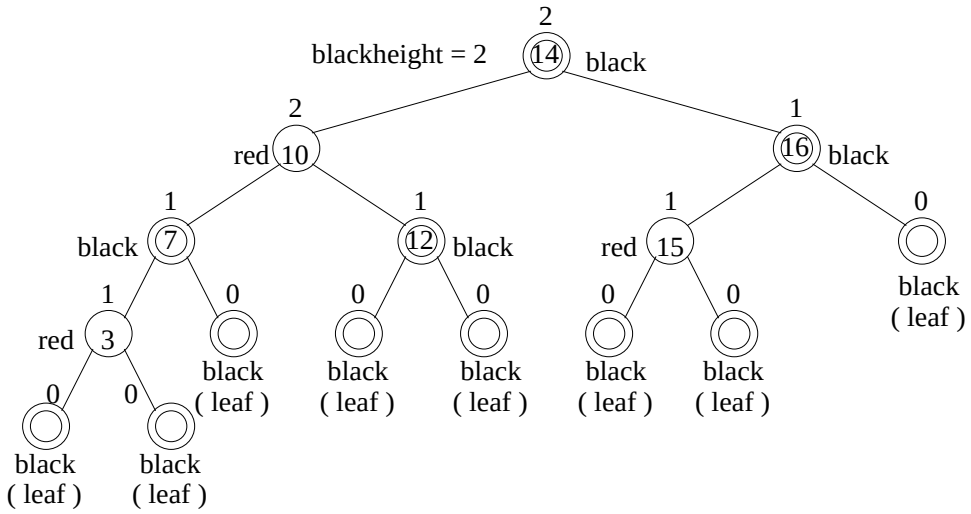


Figure 5.9: A red-black tree with black height 2

- The root may be red or black.

Examples : See Figures 5.9 and 5.10.

5.2.1 Height of a Red-Black Tree

Result 1

In a RBT, no path from a node x to a leaf is more than twice as long as any other path from x to a leaf.

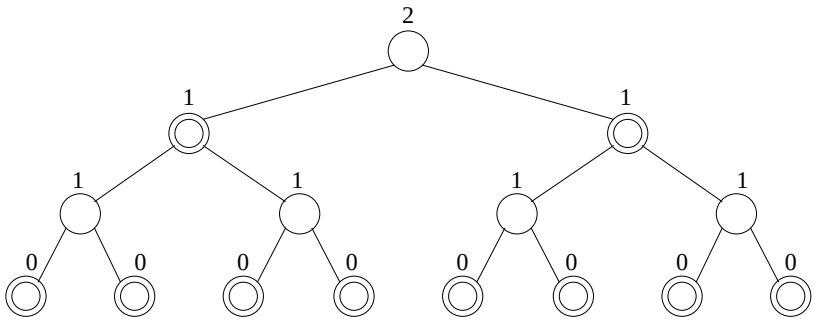
Let bh(x) be the black height of x. Then the length of a longest path from x to a leaf

$$= 2bh(x) \quad \{\text{a path on which red and black nodes alternate}\}$$

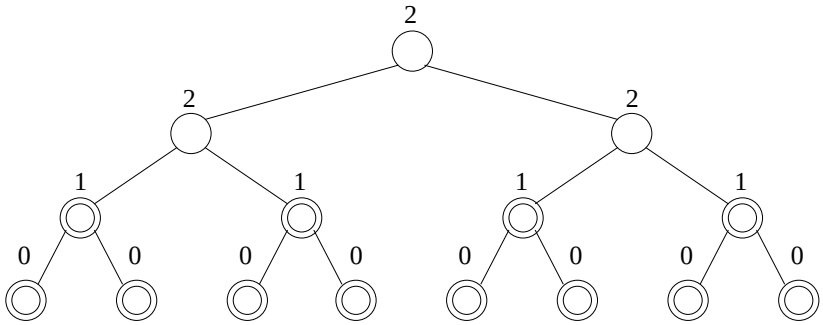
Length of the shortest possible path from x to a leaf

$$bh(x) \quad \{\text{a path that only contains blacks}\}$$

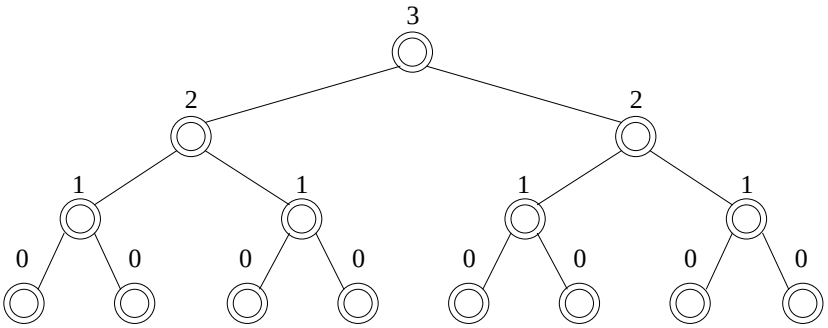
Hence the result.



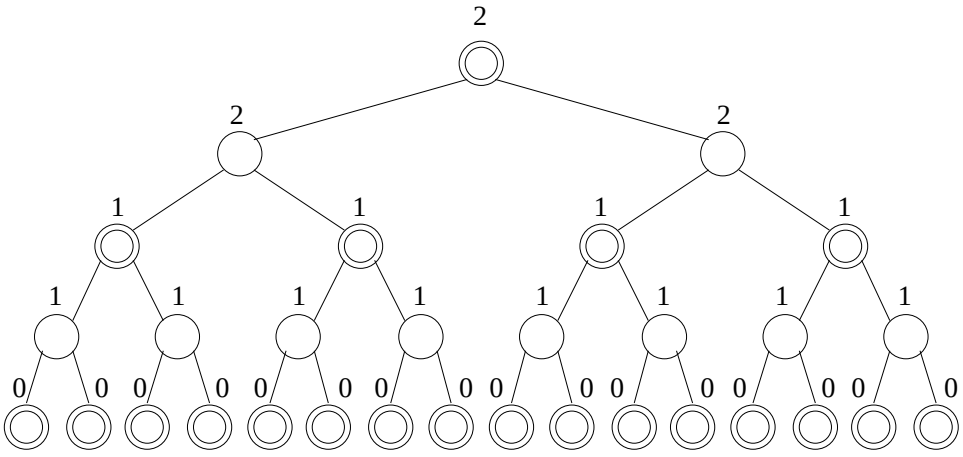
Black height = 2



Black height = 2



Black height = 3



Black height = 2

Figure 5.10: Examples of red-black trees

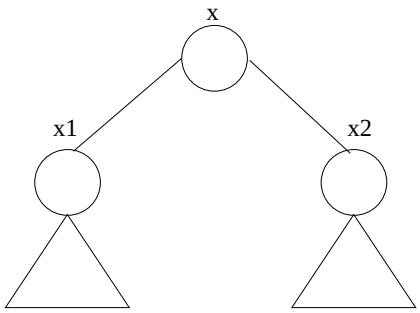


Figure 5.11: A simple red-black tree

Result 2

A red-black tree with n internal nodes has height at most

$$2\log(n+1)$$

Consider any node x and the subtree rooted at x. We will first show that this subtree has at least

$$2^{bh(x)} - 1 \quad \text{internal nodes}$$

We do this by induction on the height of x. If $h(x) = 0$, $bh(x) = 0, x$ is leaf and hence the subtree has no internal nodes, as corroborated by

$$2^0 - 1 = 0$$

Let $h(x) > 0$ and let x_1 and x_2 be its two children (Figure 5.11)

Note that $h(x_1)$, $h(x_2)$ are both $\leq h(x) - 1$. Assume the result to be true for x_1 and x_2. We shall show the result is true for x.

Now,

$$\begin{array}{lcll} bh(x_1) & \leq & bh(x) & \text{and} \quad \geq bh(x) - 1 \\ bh(x_2) & \leq & bh(x) & \text{and} \quad \geq bh(x) - 1 \end{array}$$

Therefore, the tree with root x_1 has at least

$$\left(2^{bh(x)-1} - 1\right) \quad \text{internal nodes}$$

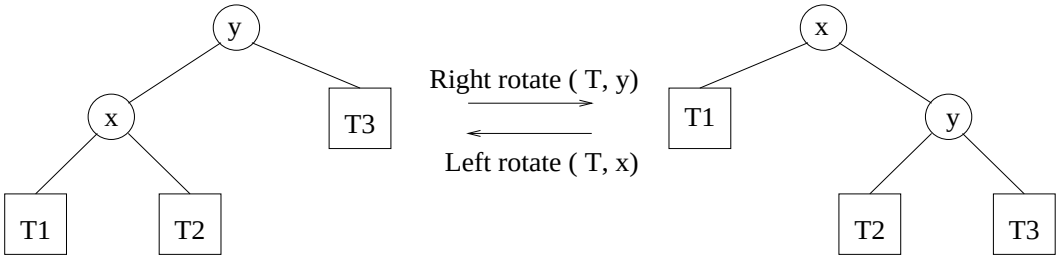


Figure 5.12: Rotations in a red-black tree

whereas the tree with root x_2 has at least

$$\left(2^{bh(x)-1}-1\right) \quad \text{internal nodes}$$

Thus the tree with root x has at least

$$1+2^{bh(x)-1}-1+2^{bh(x)-1}-1=\left(2^{bh(x)}-1\right) \quad \Leftarrow \quad \text{internal nodes}$$

To complete the proof, let h be the height of the tree. Then

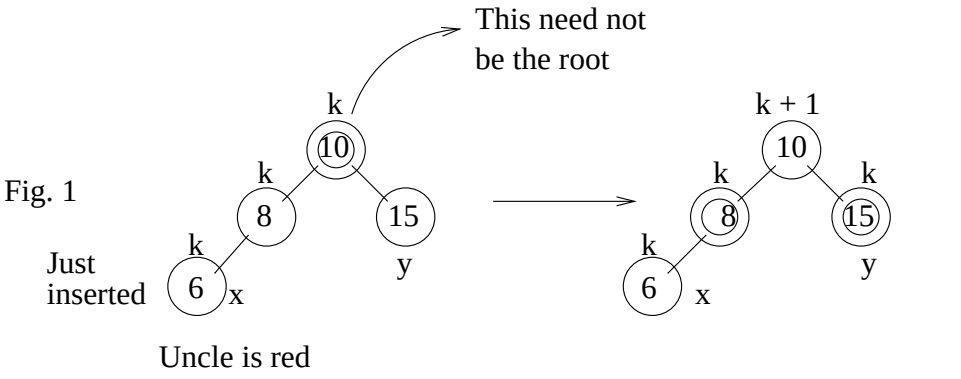
$$bh\ (\text{root}) \ \geq \frac{h}{2}$$

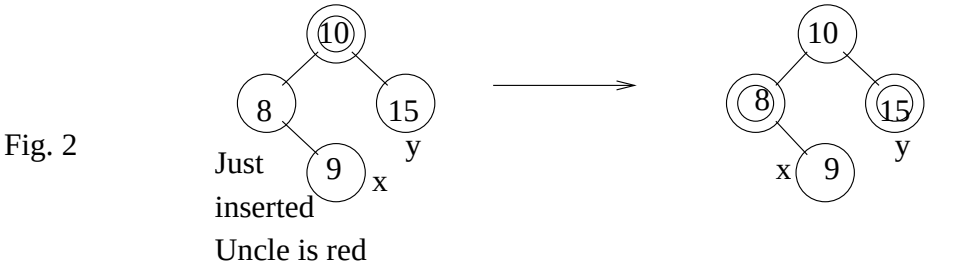
Thus

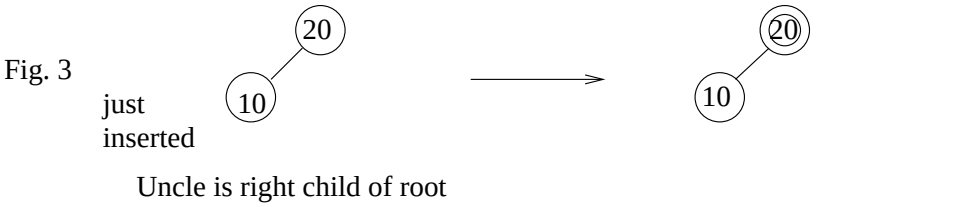
$$\begin{array}{rrl} & n & \geq \quad 2^{h/2}-1 \\ \Rightarrow & h & \geq \quad 2\log(n+1) \end{array}$$

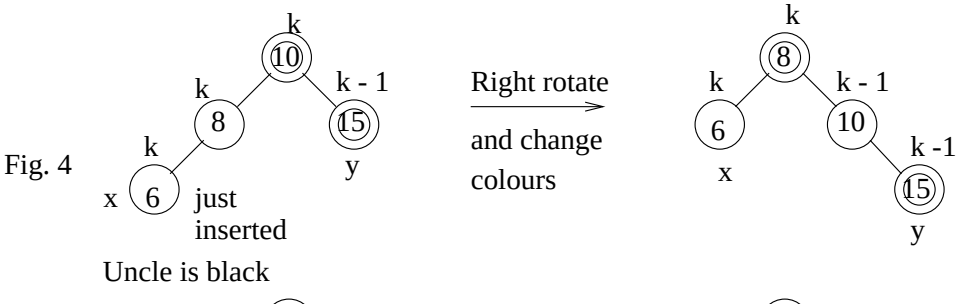
5.2.2 Red-Black Trees: Insertions

- While inserting a node, the resulting tree may violate the red-black properties. To reinstate the violated property, we use
  - **Recolouring** and/or
  - **Rotation** (same as in AVL trees: See Figure 5.12)
    - left
    - right
    - double
- To insert a node x, we first insert the node as if in an ordinary BST and colour it red. If the parent of the inserted node is black, then









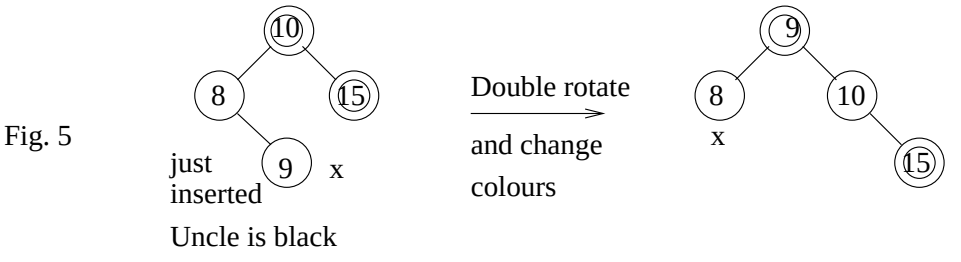


Figure 5.13: Insertion in RB trees: Different scenarios

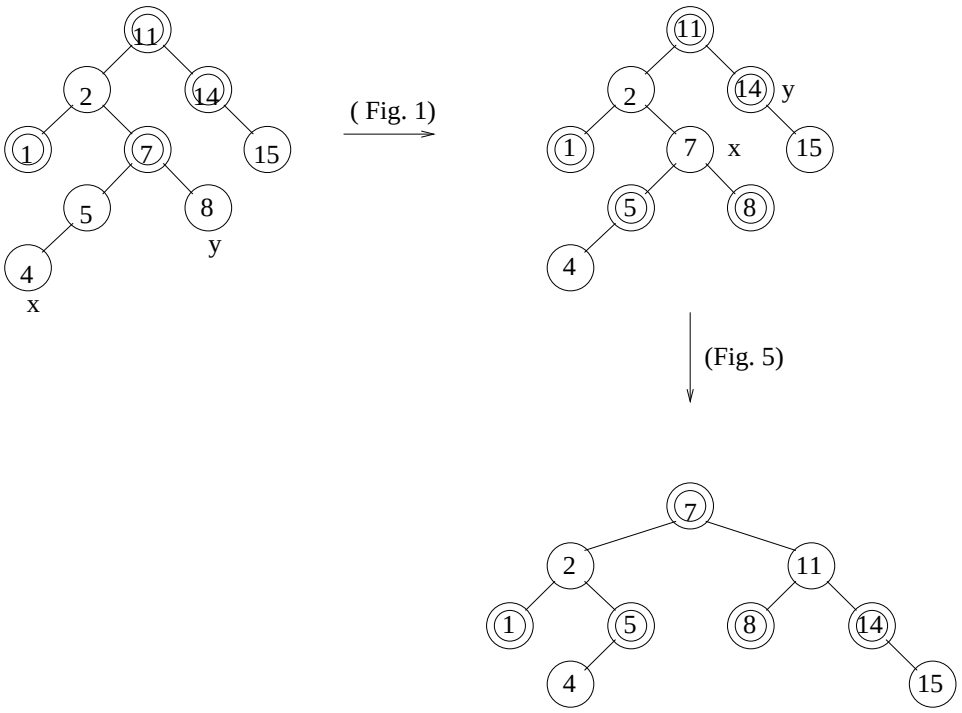


Figure 5.14: Insertion in red-black trees: Example 1

we are done since none of the RB properties will be violated. If the parent is red, then the red constraint is violated. See Figure 5.13.

In such a case, we bubble the violation up the tree by repeatedly applying the recolouring transformation of Figure 1 or Figure 2 until it no longer applies. This either eliminates the violation or produces a situation in which one of the transformations in Figures 3, 4, 5 applies, each of which leaves no violation.

- An insertion requires $O(\log n)$ recolourings and at most two rotations.
- Figures 5.14 and 5.15 illustrate two examples.

5.2.3 Red-Black Trees: Deletion

See Figures 4.33 – 4.37.

First, search for an element to be deleted.

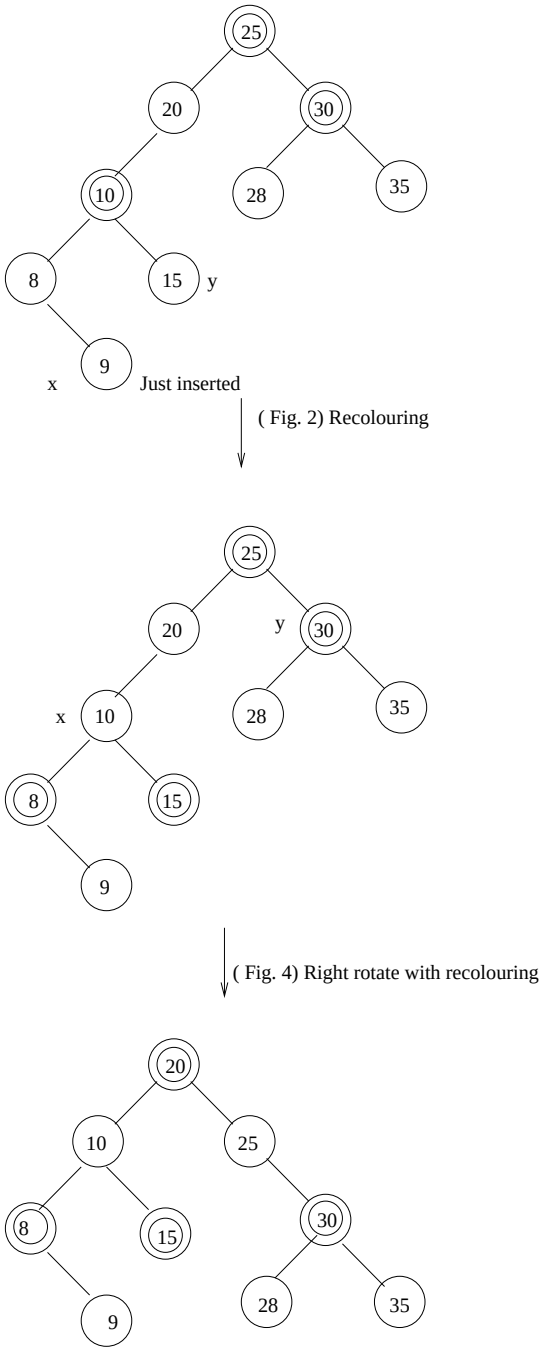


Figure 5.15: Insertion in red-black trees: Example 2

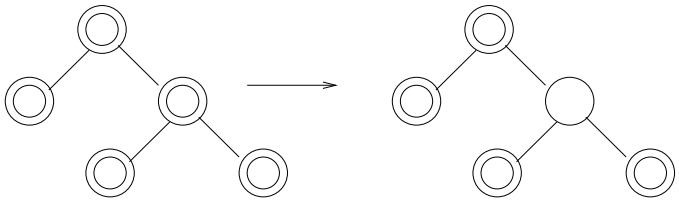


Figure 5.16: Deletion in RB trees: Situation 1

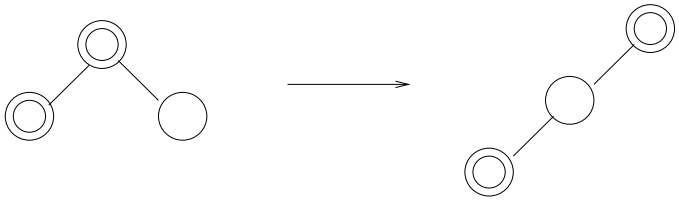


Figure 5.17: Deletion in red-black trees: Situation 2

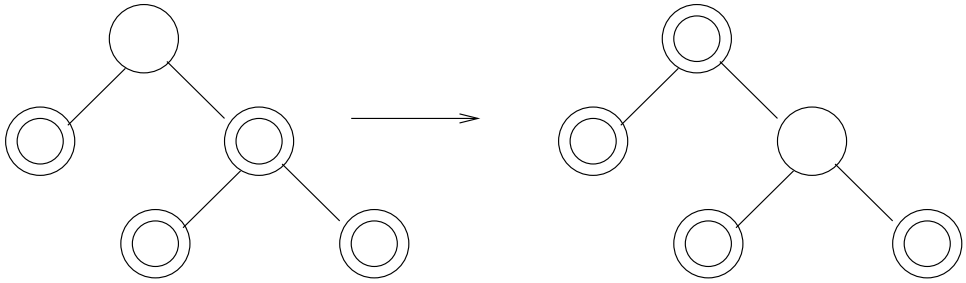


Figure 5.18: Deletion in red-black trees: Situation 3

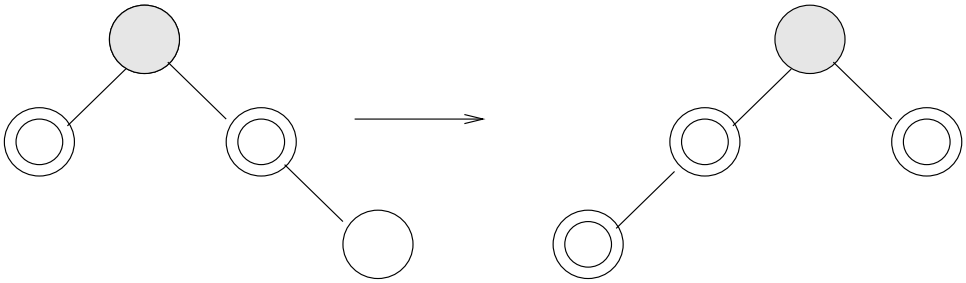


Figure 5.19: Deletion in red-black trees: Situation 4

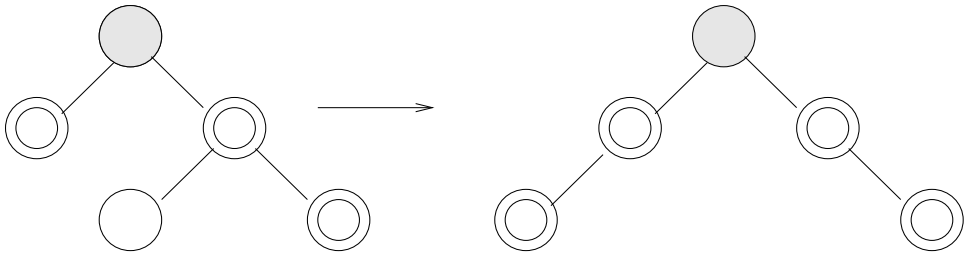


Figure 5.20: Deletion in red-black trees: Situation 5

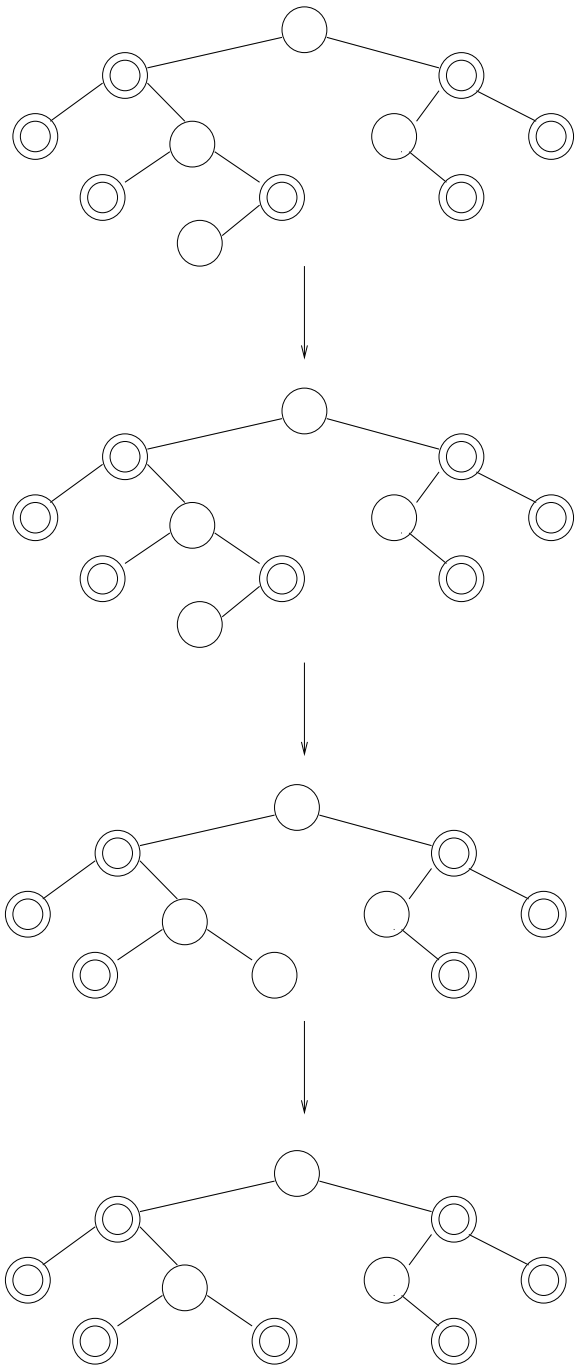


Figure 5.21: Deletion in red-black trees: Example 1

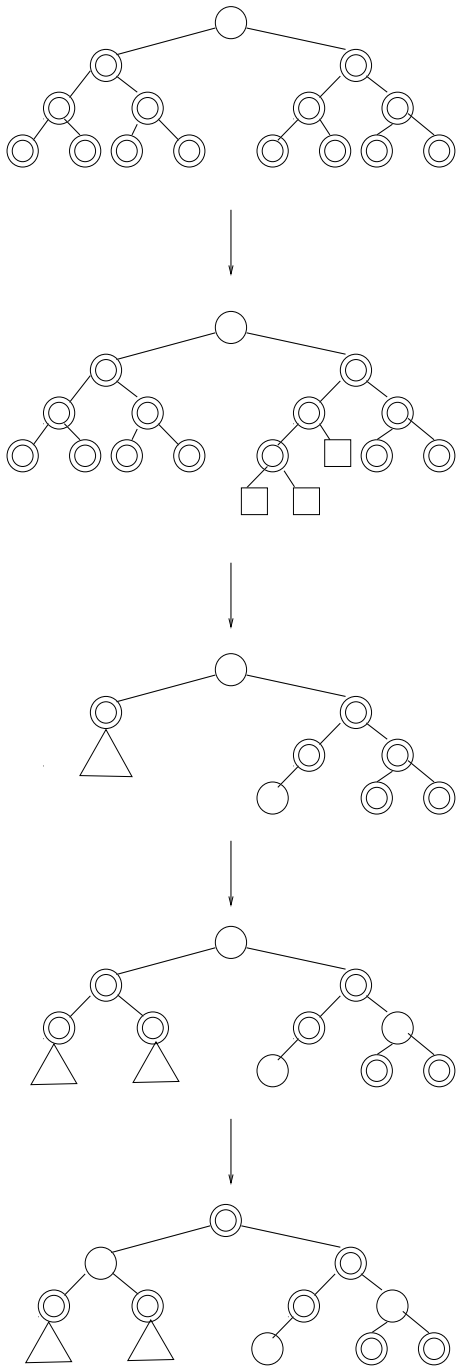


Figure 5.22: Deletion in red-black trees: Example 2

- If the element to be deleted is in a node with only left child, swap this node with the one containing the largest element in the left subtree. (This node has no right child).
- If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right subtree (This node has no left child).
- If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways.

While swapping, swap only the keys but not the colours.

- The item to be deleted is now in a node having only a left child or only a right child. Replace this node with its sole child. This may violate red constraint or black constraint. Violation of red constraint can be easily fixed.
- If the deleted node is block, the black constraint is violated. The removal of a black node y causes any path that contained y to have one fewer black node.
- Two cases arise:

  1. The replacing node is red, in which case we merely colour it black to make up for the loss of one black node.
  2. The replacing node is black.

     In this case, we "bubble" the "shortness" up the tree by repeatedly applying the recolouring transformation of Figure 5.16 until it no longer applies.

  Then we perform the transformation in Figure 5.17 if it applies, followed if necessary by one application of Figure 5.18, Figure 5.19, or Figure 5.20.

- RB-deletion requires O(log n) recolourings and at most 3 rotations.
- See Figures 5.21 and 5.22 for two examples.

5.3 2-3 Trees

Definition

- A 2-3 Tree is a null tree (zero nodes) or a single node tree (only one node) or a multiple node tree with the following properties:
  1. Each interior node has two or three children
  2. Each path from the root to a leaf has the same length.

Fields of a Node :

Internal Node

p_1	k_1	p_2	k_2	p_3

p_1 : Pointer to the first child
p_2 : Pointer to the second child
p_3 : Pointer to the third child
k_1 : Smallest key that is a descendent of the second child
k_2 : Smallest key that is a descendent of the third child

Leaf Node

key	other fields

- Records are placed at the leaves. Each leaf contains a record (and key)

Example: See Figure 5.23

Search

- The values recorded at the internal nodes can be used to guide the search path.
- To search for a record with key value x, we first start at the root. Let k_1 and k_2 be the two values stored here.

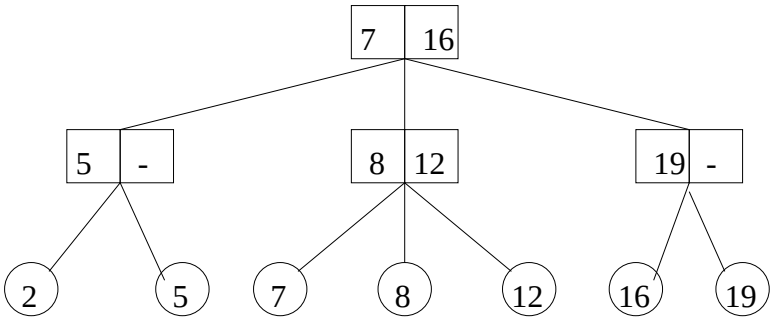


Figure 5.23: An example of a 2-3 tree

1. If $x < k_1$, move to the first child
2. If $x \geq k_1$ and the node has only two children, move to the second child
3. If $x \geq k_1$ and the node has three children, move to the second child if $x < k_2$ and to the third child if $x \geq k_2$.

- Eventually, we reach a leaf. x is in the tree iff x is at this leaf.

Path Lengths

- A 2-3 Tree with k levels has between 2^{k-1} and 3^{k-1} leaves
- Thus a 2-3 tree with n elements (leaves) requires

  at least $1 + \log_3 n$ levels

  at most $1 + \log_2 n$ levels

5.3.1 2-3 Trees: Insertion

For an example, see Figure 5.24.

Insert (x)

1. Locate the node v, which should be the parent of x
2. If v has **two** children,

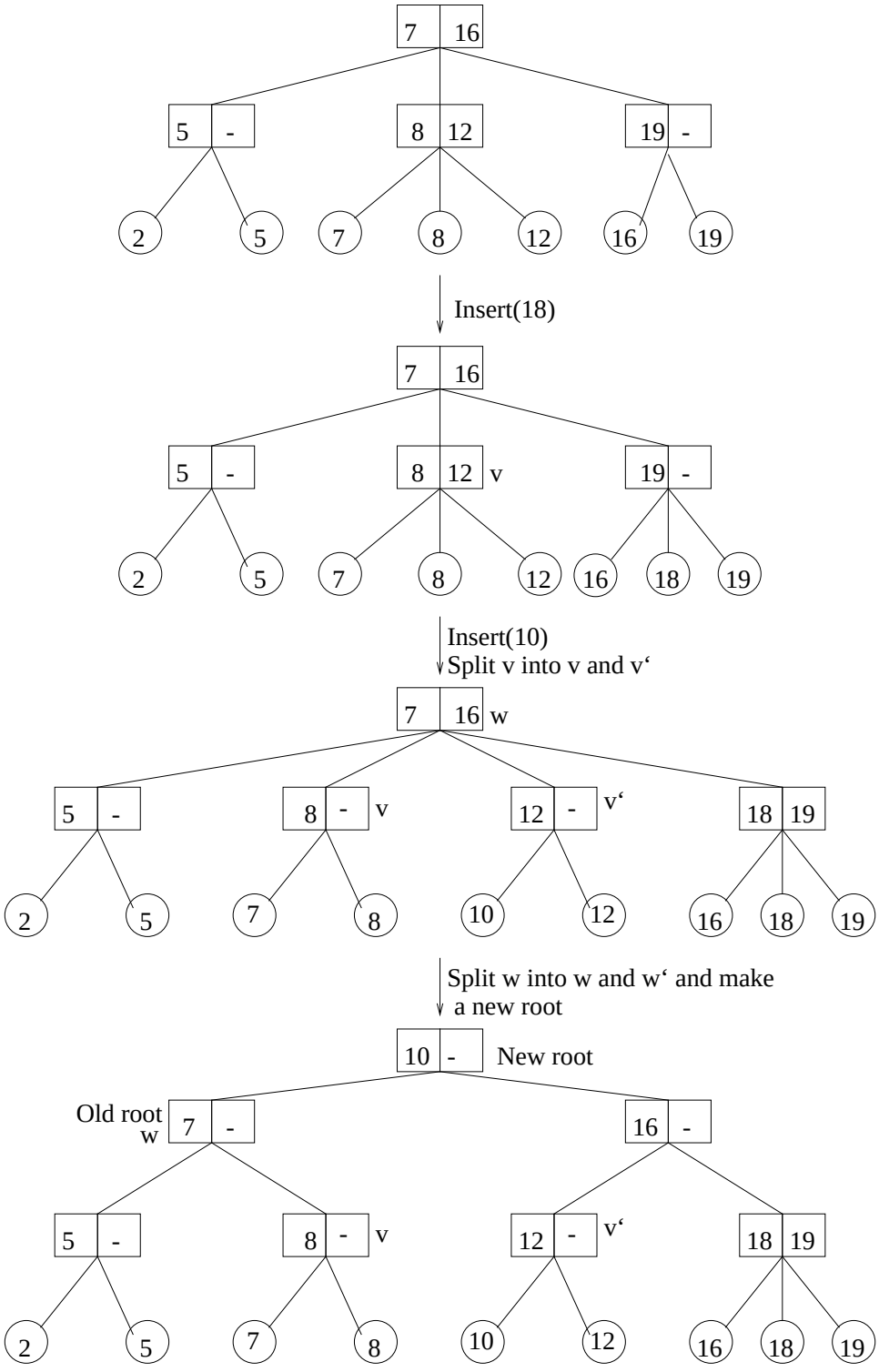


Figure 5.24: Insertion in 2-3 trees: An example

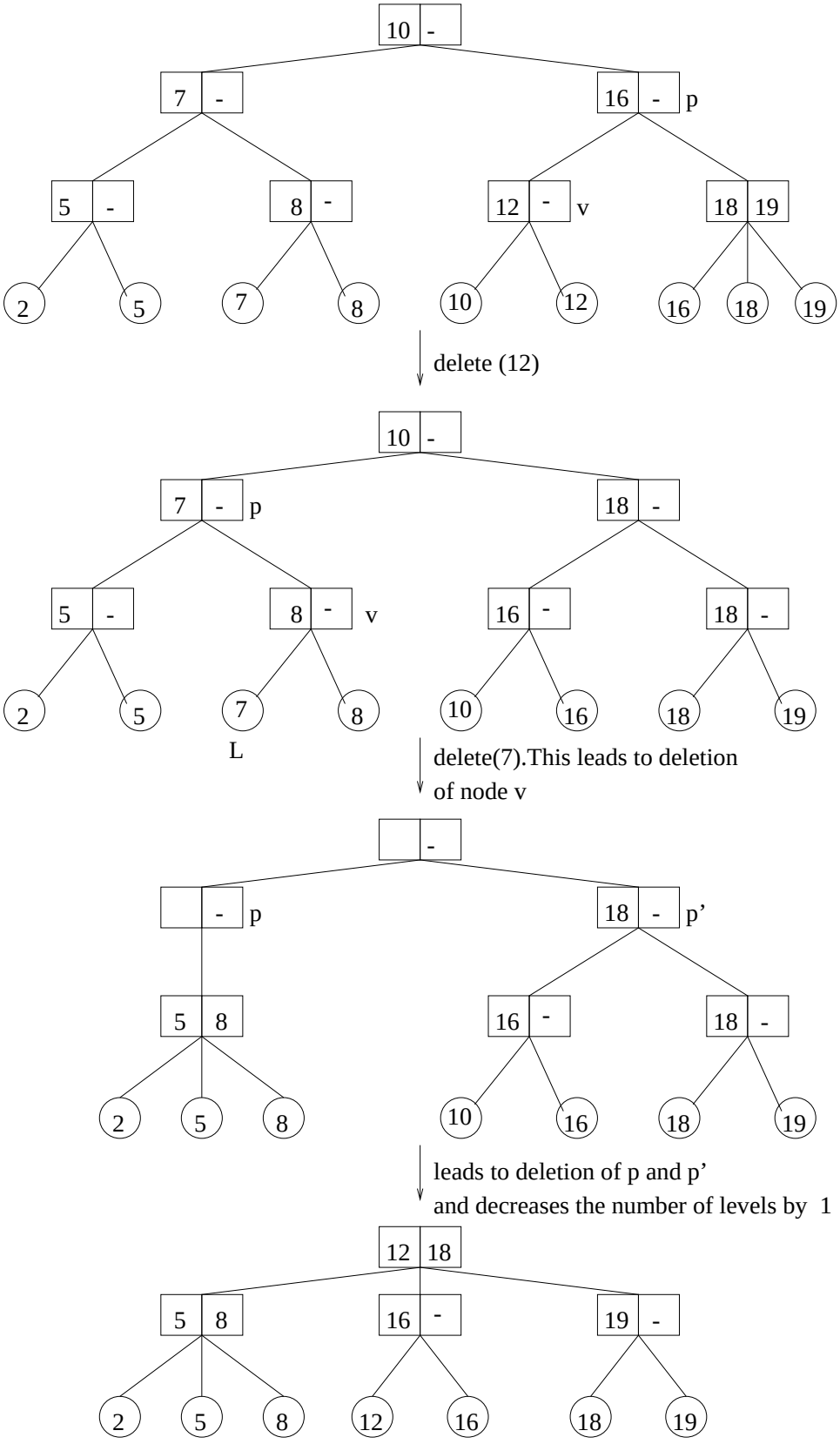


Figure 5.25: Deletion in 2-3 trees: An Example

- make x another child of v and place it in the proper order
- adjust k_1 and k_2 at node v to reflect the new situation

3. If v has **three** children,

   - split v into two nodes v and v'. Make the two smallest among four children stay with v and assign the other two as children of v'.
   - Recursively insert v' among the children of w where

     $$w = \text{parent of } v$$

   - The recursive insertion can proceed all the way up to the root, making it necessary to split the root. In this case, create a new root, thus increasing the number of levels by 1.

5.3.2 2-3 Trees: Deletion

Delete (x)

1. Locate the leaf L containing x and let v be the parent of L
2. Delete L. This may leave v with only one child. If v is the root, delete v and let its lone child become the new root. This reduces the number of levels by 1. If v is not the root, let

   $p =$ parent of v

   If p has a child with 3 children, transfer an appropriate one to v if this child is adjacent (sibling) to v.

   If children of p adjacent to v have only two children, transfer the lone child of v to an adjacent sibling of v and delete v.

   – If p now has only one child, repeat recursively with p in place of v. The recursion can ripple all the way up to the root, leading to a decrease in the number of levels.

Example: See Figure 5.25.

5.4 B-Trees

- Generalization of 2-3 Trees
- Multi-way search tree, very well suited for external storage
- Standard organization for indices in database systems
- Balanced tree, which achieves minimization of disk accesses in database retrieval applications.

REF R. Bayer. Symmetric binary B-trees: Data Structures and maintenance algorithms. *Acta Informatica*, Volume 1, pp.290-306, 1972.

REF R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, Volume 1, Number 3, pp. 173-189, 1972.

REF D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, Volume 11, Number 2, pp 121-137, 1979.

5.4.1 Definition of B-Trees

A B-tree of order m is an m-ary search tree with the following properties:

- The root is either a leaf or has at least two children
- Each node, except for the root and the leaves, has between $\lceil m/2 \rceil$ and m children
- Each path from the root to a leaf has the same length.
- The root, each internal node, and each leaf is typically a disk block.
- Each internal node has up to $(m-1)$ key values and up to m pointers (to children)
- The records are typically stored in leaves (in some organizations, they are also stored in internal nodes)

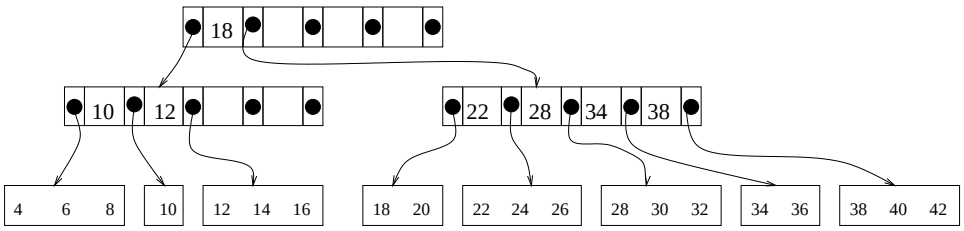


Figure 5.26: An example of a B-tree

Figure 5.26 shows a B-tree of order 5 in which at most 3 records fit into a leaf block .

- A B-tree can be viewed as a hierarchical index in which the root is the first level index.
- Each non-leaf node is of the form

$$(p_1, k_1, p_2, k_2, \ldots, k_{m-1}, p_m)$$

where

p_i is a pointer to the ith child, $1 \leq i \leq m$

k_i Key values, $1 \leq i \leq m-1$, which are in the sorted order, $k_1 < k_2 < \cdots < k_{m-1}$, such that

- all keys in the subtree pointed to by p_1 are less than k_1
- For $2 \leq i \leq m-1$, all keys in the subtree pointed to by p_i are greater than or equal to k_{i-1} and less than k_i
- All keys in the subtree pointed to by p_m are greater than (or equal to) k_{m-1}

5.4.2 Complexity of B-tree Operations

Let there be n records. If each leaf has b records on the average, we have,

$$\text{average number of leaves} = L = \lceil n/b \rceil$$

Longest paths occur if the number of children at every stage $= \lceil m/2 \rceil = l$, say

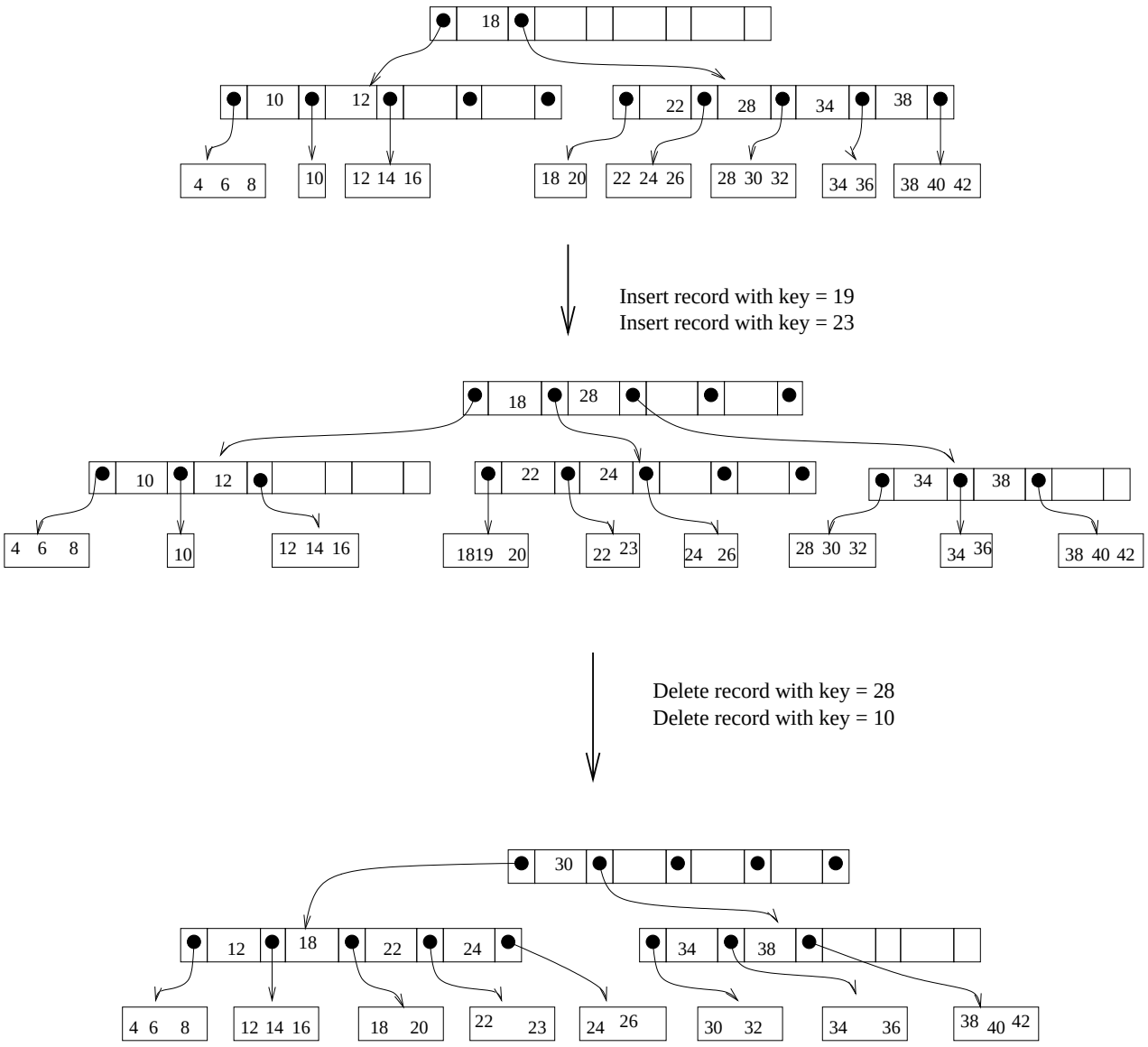


Figure 5.27: Insertion and deletion in B-trees: An example

Average number of nodes that are parents of leaves $= \frac{L}{l}$

Average number of nodes that are second level parents

$$= \left(\frac{L}{l}\right) \Big/ l = \frac{L}{l^2}$$

If h is the level of the leaves, we have

$$\begin{aligned} & \frac{L}{l^{h-1}} \geq 1 \\ \text{or} \quad & h \leq \log_l L \\ \text{i.e.,} \quad & h \leq \log_{\lceil m/2 \rceil} \lceil n/b \rceil \end{aligned}$$

If $n = 10^6$ records (1 million records), $b = 10$, and $m = 100$, we have

$$h \leq 3.9$$

5.4.3 B-Trees: Insertion

Insert (r, x)

Insert a record r with key value $= x$

- Locate the leaf L at which r should belong.
- If there is room for r in L,
  - Insert r into L in the proper order.
  - Note that no modifications are necessary to the ancestors of L.
- If there is no room for r in L,
  - request the file system for a new block L' and move the last half of the records from L to L', inserting r into its proper place in L or L'.

    Let

$$\begin{aligned} P &= \text{parent of } L \\ k' &= \text{smallest key value in } L' \end{aligned}$$

$$\ell' \;=\; \text{pointer to} \quad L'$$

Insert k' and ℓ' in P (recursively)

- If P already has m pointers, insertion of k' and ℓ' into P will cause P to be split and will necessitate an insertion of a key and a pointer in the parent of P.
- The recursive scheme can ripple all the way up to the root causing the root to be split, in which case
  - Create a new root with the two halves of the old root as its two children. This
    - increases the number of levels
    - is the only situation where a node has fewer than $\lceil m/2 \rceil$ children.

Example: See Figure 5.27.

5.4.4 B-Trees: Deletion

Delete (r, x)

Delete a record r with key value $= x$

- Locate the leaf L containing r
- Remove r from L.

  Case 1: L does not become empty after deletion
  - if r is not the first record in L, then there is no need to fix the key values at higher levels.
  - If r is the first record in L, then
    - if L is not the first child of its parent P, then set the key value in P's entry for L to be the new first key value in L
    - if L is the first child of P, the key of r is not recorded in P; locate the lowest ancestor A of P such that L is not the leftmost descendent of A and set the appropriate key value in A.

Case 2: L becomes empty after deletion of r

- Return L to the file system
- Adjust the keys and pointers in P (parent of L) to reflect the removal of L
- If the number of children of P now $< \lceil m/2 \rceil$, examine the node P' immediately to the left or the right. If P' has at least $1 + \lceil m/2 \rceil$ children, distribute the keys and pointers in P and P' evenly between P and P'.
  - Modify the key values for P and P' in the parent of P
  - If necessary, ripple the effects to as many ancestors of P as are affected.
  - If the number of children of P' is exactly $\lceil m/2 \rceil$, we combine P and P' into a single node with $2\lceil m/2 \rceil$ -1 children. Then remove the key and pointer to P' from the parent for P'. If the effects of deletion ripple all the way back to the root, combine the only two children of the root. The resulting combined node is the new root and the old root is returned to the file system. The number of levels decreases by 1.

See Figure 5.27 for an example.

5.4.5 Variants of B-Trees

B^+-Trees

- Internal nodes contain the indices to the records corresponding to the key values $k_1, k_2, \ldots, k_m$ stored at the internal node. This obviates the need for repeating these key values and associated indices at the leaf level.
- More efficient than B-Trees in the case of successful searches.

B*-Trees

- The minimum number of children at each internal node (except the root) is
$$\left\lceil \frac{2m}{3} \right\rceil$$
- Pathlengths are smaller for obvious reasons
- Insertions and deletions are more complex.

5.5 To Probe Further

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
2. Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996. Indian Edition published by Prentice Hall of India, 1998.
3. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.
4. Robert L. Kruse, Bruce P. Leung, and Clovis L. Tondo. *Data Structures and Program design in C*. Prentice Hall, 1991. Indian Edition published by Prentice Hall of India, 1999.
5. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.
6. Duane A. Bailey. *Java Structures: Data Structures in Java for the Principled Programmer*. McGraw-Hill International Edition, 1999.
7. Donald E. Knuth. *Sorting and Searching*, Volume 3 of The Art of Computer Programming, Addison-Wesley, 1973.
8. Y. Langsam, M.J. Augenstein, and A.M. Tenenbaum. *Data Structures using C and C++*. Second Edition, Prentice Hall, 1996. Indian Edition published by Prentice Hall of India, 2000.

9. Kurt Mehlhorn. *Sorting and Searching.* Volume 1 of *Data Structures and Algorithms.* Springer-Verlag, 1984.

10. Sartaj Sahni. *Data Structures, Algorithms, and Applications in Java.* McGraw-Hill Higher Education, 2000.

11. Thomas A. Standish. *Data Structures in Java.* Addison-Wesley, 1998. Indian Edition published by Addison Wesley Longman, 2000.

12. Nicklaus Wirth. *Data Structures + Algorithms = Programs.* Prentice-Hall, Englewood Cliffs. 1975.

13. G.M. Adel'son-Vel'skii and E.M. Landis. An algorithm for the organization of information. *Soviet Mathematics Monthly*, Volume 3, pp.1259-1263, 1962.

14. R. Bayer. Symmetric binary B-trees: Data Structures and maintenance algorithms, *Acta Informatica*, Volume 1, pp.290-306, 1972.

15. R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, Volume 1, Number 3, pp. 173-189, 1972.

16. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, Volume 11, Number 2, pp 121-137, 1979.

17. William Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, Volume 33, Number 6, pp. 668-676, 1990.

18. Daniel D Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, Volume 32, Number 3, pp 652-686, 1985.

19. Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, Volume 6, Number 2, pp.306-318, 1985.

5.6 Problems

5.6.1 AVL Trees

1. Draw all possible binary search trees containing the key values 1, 2, 3, 4. Which of these are AVL trees?

2. In each of the following, insert the keys in the order shown, to build them into AVL trees.

   (a) A, Z, B, Y, C, X, D, W, E, V, F.

   (b) A, B, C, D, E, F, G, H, I, J, K, L.

   (c) A, V, L, T, R, E, I, S, O, K.

   In each case, do the following:

   (a) Delete each of the the keys inserted in LIFO order (last key inserted is first deleted).

   (b) Delete each of the keys inserted in FIFO order (first key inserted is first deleted).

3. An AVL tree is constructed by inserting the key values 1, 2, 3, 4, 5 in some order specified by a permutation of 1, 2, 3, 4, 5, into an initially empty tree. For which of these permutations is there no need to do any rotations at any stage during the insertions?

4. Show that the number of (single or double) rotations done in deleting a key from an AVL tree cannot exceed half the height of the tree.

5.6.2 Red-Black Trees

1. Show that any arbitrary n-node binary tree can be transformed into any other arbitrary n-node binary tree using $O(n)$ rotations.

2. Draw the complete binary search tree of height 3 on the keys $\{1, 2, \cdots, 15\}$. Add the NIL leaves and colour the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

3. Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree. Now show the RB trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 31, 38, 41.

4. Suppose that a node x is inserted into an RB tree and then immediately deleted. Is the resulting RB tree the same as the initial RB tree? Justify your answer.

5. Construct the RB-trees that result by repeated insertions in each of the following cases, starting from an initially empty tree.

   (a) $1, 2, \ldots, 15$

   (b) $8, 4, 12, 6, 14, 2, 10, 7, 11, 5, 9, 3, 8, 1, 15$

   (c) $1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8$

5.6.3 2-3 Trees and B-Trees

1. For a 2-3 tree with n leaves, compute the following:

   (a) Minimum height

   (b) Maximum height

2. For a 2-3 tree with height h, compute the following:

   (a) Minimum number of leaves

   (b) Maximum number of leaves

3. What are the minimum and maximum numbers of internal nodes a B-tree of order m and height h (ie. $(h+1)$ levels) may have?

4. Suppose we insert the keys 1,2, ..., in ascending order into an initially empty 2-3 tree. Which key would cause the leaves to be at level 4 for the first time? (Assume that the root is at level 1). Write down the resulting 2-3 tree.

5. Devise an algorithm to find the k^{th} largest element in a

   (a) 2-3 tree.

   (b) B-tree.

6. Suppose that the keys $1, 2, \ldots, 20$, are inserted in that order into a B-tree with $m = 2$. How many internal nodes does the final B-tree have? Show the final B-tree. Compare this with a binary search tree for the same sequence of inserted elements. Assume that each leaf block can only store one record.

7. Suppose that the keys $1, 2, \ldots, 20$, are inserted in that order into a B-tree with $m = 4$. How many internal nodes does the final B-tree have? Show the final B-tree. Compare this with a 4-way search tree for the same sequence of inserted elements. Assume that each leaf block can only store one record.

8. Given a B-tree, explain how to find

   (a) the minimum key inserted.

   (b) predecessor of a given key stored.

(c) successor of a given key stored.

9. Suppose we have a file of one million records where each record takes 100 bytes. Blocks are 1000 bytes long and a pointer to a block takes 4 bytes. Also each key value needs two bytes. Devise a B-tree organization for this file.

10. Design a B-tree organization if it is required to contain at least one billion keys but is constrained to have a height of 4 (ie. 5 levels). Assume that the number of records for each leaf block is 16.

11. Compute the smallest number of keys, which when inserted in an appropriate order will force a B-tree of order 5 to have exactly m levels (root at level 1 and leaves at level m).

12. A B^*-tree is a B-tree in which each interior node is at least 2/3 full (rather than half full). How do the algorithms for search and delete change for a B^*-tree? What would be the advantages and disadvantages of B^*-trees compared to B-trees?

13. Assume that it takes $a + bm$ milliseconds to read a block containing a node of an m-ary search tree. Assume that it takes $c + d \log_2 m$ milliseconds to process each node in internal memory. If there are n nodes in the tree, compute the total time taken to find a given record in the tree.

14. Suppose that disk hardware allows us to choose the size of a disk block arbitrarily, but that the time it takes to read the disk block is $a+bt$, where a and b are specified constants and t is the degree of a B-tree that uses blocks of the selected size (i.e. each disk block contains $(t-1)$ key values and t pointers). Assume that a total of N records are stored in the B-tree database, with each disk block storing one record.

    (a) Compute the minimum number of disk blocks required by the above B-tree organization, including the blocks for storing the records.

    (b) Assuming that any record is equally likely to be searched for, estimate the total average disk block reading time for accessing a record.

    (c) How do you choose t so as to minimize the total average disk block reading time?

5.7 Programming Assignments

5.7.1 Red-Black Trees and Splay Trees

This assignment involves the implementation of searches, insertions, and deletions in red-black trees and splay trees, and evaluating the performance of each data structure in several scenarios. The intent is to show that these trees are better than ordinary (unbalanced) binary search trees and also to compare red-black trees with splay trees.

Generation of Keys

Assume that your keys are character strings of length 10 obtained by scanning a C program (identical to Programming assignment 1). If a string has less than 10 characters, make it up to 10 characters by including an appropriate number of trailing *'s. On the other hand, if the current string has more than 10 characters, truncate it to have the first ten characters only.

Inputs to the Program

The possible inputs to the program are:

- n: The initial number of insertions of distinct strings to be made for setting up an initial search tree (RBT or ST).
- I: This is a decimal number from which a ternary string is to be generated (namely the radix-3 representation of I). In this representation, assume that a **0** represents the **search** operation, a **1** represents the **insert** operation, and a **2** represents the **delete** operation.
- N: Number of operations to be performed on the data structure.
- A C program, from which the tokens are to be picked up for setting up and experimenting with the search trees.

What should the Program Do?

1. Scan the given C program and as you scan, insert the first n distinct strings scanned into an initial search tree (BST or RBT or ST).
2. For each individual operation, keep track of:
   - Number of probes (comparison of two strings of 10 characters each)
   - Number of pointer assignments (a single rotation involves three pointer assignments)
   - Number of single rotations and number of double rotations (in the case of RBTs and STs)
   - Examining or changing the colour (in a red-black tree)

   Now, compute for the BST, RBT, and ST so created, the following:
   - Height of the search tree

- Total number of probes for all the operations
- Average number of probes (averaged over all the operations) for a typical successful search, unsuccessful search, insert, and delete.
- Total number of pointer assignments
- Total number of single rotations (in the case of RBTs and STs)
- Total number of double rotations (in the case of RBTs and STs)
- Total number of each type of splaying steps (Zig, Zag, Zig-zig, Zig-zag, Zag-zig, Zag-zag) in the case of STs
- Total number of recolourings in the case of RBTs

3. Scan the rest of the C program token by token, searching for it or inserting it or deleting it, as dictated by the radix-3 representation of I and its permutations. You are to carry out a total of N operations.

4. Repeat Step 2 to evaluate the trees over the N operations.

5.7.2 Skip Lists and Binary search Trees

The paper by William Pugh (Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, Volume 33, Number 6, pp. 668-676, 1990) compares the performance of skip lists with AVL trees, splay trees, and 2-3 trees. Implement all these data structures, design experiments on these data structures, and verify the results presented in that article.

5.7.3 Multiway Search Trees and B-Trees

The objective of this assignment is to implement insertions and deletions in *multi-way search trees* and *B-trees*, and evaluate their performance. Three scenarios are considered: insertions alone; deletions alone; and insertions and deletions interleaved randomly.

- **Scenario 1 – Insertions only:**
  Generate a random permutation of n numbers, say $a_1, a_2, \ldots, a_n$. Insert these elements into an initially empty data structure. This process will result in a multi-way search tree and a B-tree, each with n nodes. The *order* of the tree to be constructed, m, is given as an input. Note that a binary search tree is a multi-way search tree with $m = 2$, whereas a 2-3 tree is a B-tree with $m = 3$. Also the number of records per leaf is also to be specified as part of the input. Choose a small number, say 2 or 3, here.

- **Scenario 2 – Deletions only:**
  Start with the tree constructed in Scenario 1 and delete the n elements, one by one, in the following orders:

  - In the order of insertion
  - In the reversed order of insertion
  - In a random order

- **Scenario 3 – Interleaved Insertion and Deletions:**
  Start with the tree constructed in Scenario 1 and do a series of randomly interleaved insertions and deletions. Assume that insertions and deletions are equally likely. Assume that elements currently in the tree and elements not currently in the tree are generated with equal probabilities.

The performance measures are the following:

- Average height of the tree. You should compute it for both MST and BT in all the three scenarios. Also, you should be able to compute the respective heights at any intermediate stage. It should be possible to break execution at any desired point and track the probe sequence for a desired operation.
- Average number of probes required for insertions and deletions. This can be computed for each type of tree in each of the three scenarios.

Repeat the above experiment for several random sequences to obtain better and more credible performance estimates. You will get bonus marks if you implement B^*-trees also.

Chapter 6

Priority Queues

A Priority queue is an important abstract data type in Computer Science. Major operations supported by priority queues are INSERT and DELETEMIN. They find extensive use in

- implementing schedulers in OS, and distributed systems
- representing event lists in discrete event simulation
- implementing numerous graph algorithms efficiently
- selecting k^{th} largest or k^{th} smallest elements in lists (order statistics problems)
- sorting applications

Simple ways of implementing a priority queue include:

- Linked list – sorted and unsorted
- Binary search tree

6.1 Binary Heaps

Heaps (occasionally called as partially ordered trees) are a very popular data structure for implementing priority queues.

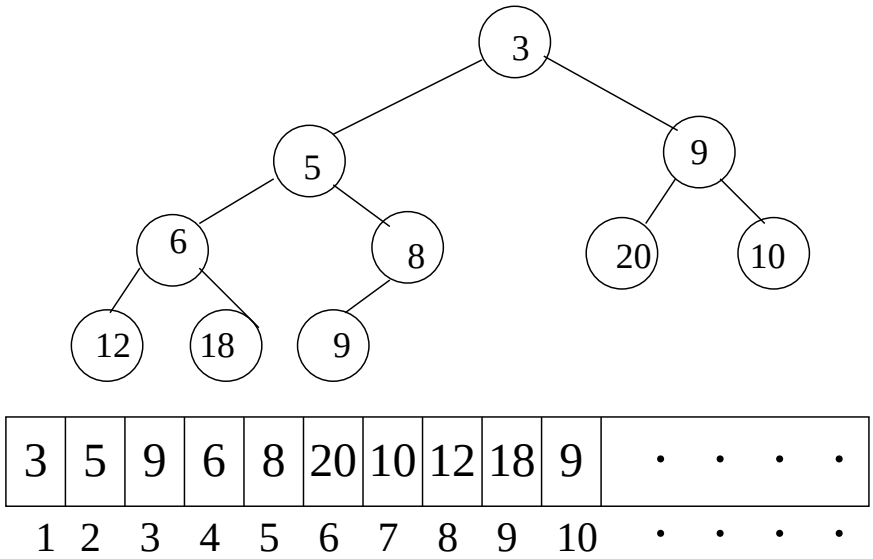


Figure 6.1: An example of a heap and its array representation

- A heap is either a min-heap or a max-heap. A min-heap supports the insert and deletemin operations while a max-heap supports the insert and deletemax operations.
- Heaps could be binary or d-ary. Binary heaps are special forms of binary trees while d-ary heaps are a special class of general trees.
- Binary heaps were first introduced by Williams in 1964.

REF. J.W.J. Williams. Algorithm 232: Heapsort. Communications of the ACM, Volume 7, 1964, pp 347-348

We discuss binary min-heaps. The discussion is identical for binary max-heaps.

DEF. A binary heap is a complete binary tree with elements from a partially ordered set, such that the element at every node is less than (or equal to) the element at its left child and the element at its right child. Figure 6.1 shows and example of a heap.

- Since a heap is a complete binary tree, the elements can be conveniently stored in an array. If an element is at position i in the array,

then the left child will be in position 2i and the right child will be in position $2i + 1$. By the same token, a non-root element at position i will have its parent at position $\lfloor \frac{i}{2} \rfloor$

- Because of its structure, a heap with height k will have between 2^k and $2^{k+1} - 1$ elements. Therefore a heap with n elements will have height $= \lfloor \log_2 n \rfloor$
- Because of the heap property, the minimum element will always be present at the root of the heap. Thus the findmin operation will have worst-case O (1) running time.

6.1.1 Implementation of Insert and Deletemin

Insert

To insert an element say x, into the heap with n elements, we first create a hole in position (n+1) and see if the heap property is violated by putting x into the hole. If the heap property is not violated, then we have found the correct position for x. Otherwise, we "push-up" or "percolate-up" x until the heap property is restored. To do this, we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until x can be placed in the whole. See Figure 6.2 for an example.

Worstcase complexity of insert is O (h) where h is the height of the heap. Thus insertions are O (log n) where n is the number of elements in the heap.

Deletemin

When the minimum is deleted, a hole is created at the root level. Since the heap now has one less element and the heap is a complete binary tree, the element in the least position is to be relocated. This we first do by placing the last element in the hole created at the root. This will leave the heap property possibly violated at the root level. We now "push-down" or "percolate-down" the hole at the root until the violation of heap property is stopped. While pushing down the hole, it is important to slide it down

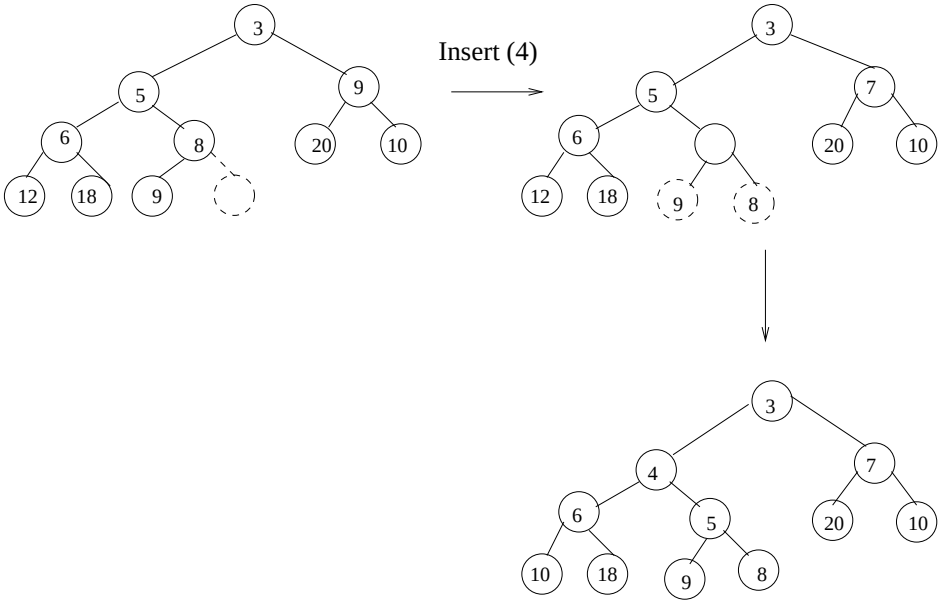


Figure 6.2: Insertion into a heap

to the less of its two children (pushing up the latter). This is done so as not to create another violation of heap property. See Figure 6.3. It is easy to see that the worst-case running time of deletemin is O (log n) where n is the number of elements in the heap.

6.1.2 Creating Heap

Given a set of n elements, the problem here is to create a heap of these elements.

- obvious approach is to insert the n elements, one at a time, into an initially empty heap. Since the worstcase complexity of inserts is O (log n), this approach has a worstcase running time of O (n log n)
- Another approach, purposed by Floyd in 1964, is to use a procedure called "pushdown" repeatedly, starting with the array consisting of the given n elements in the input-order.
  - The function pushdown (first,last) assumes that the elements a[first], ..., a[last] obey the heap property, except possibly the children of

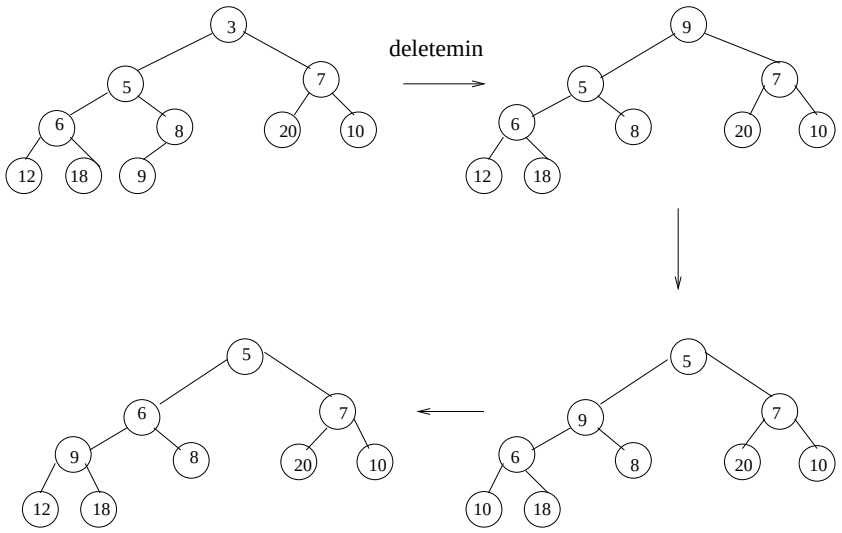


Figure 6.3: Deletemin

a[first]. The function pushes down a[first] until the heap property violation is stopped.

– The following code will accomplish what is required:
for $(i = \frac{n}{2};\ i \geq 1;\ i--)$
pushdown (i,n)

- Figure 6.4 shows an example of this for an array [5 9 4 2 1 6]
- The worstcase complexity of this approach can be shown to b O (n) by virtue of the following result.

Result: For the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h + 1)$.

Proof: The perfect or full binary tree of height h has 1 node at height h, 2 nodes art height (h-1), 4 nodes at height (h-2), ...

Therefore the required sum S is given by

$$\begin{aligned} S &= \sum_{i=0}^{h} 2^i (h - i) \\ &= h + 2(h-1) + 4(h-2) + ... + 2^{h-1} \end{aligned}$$

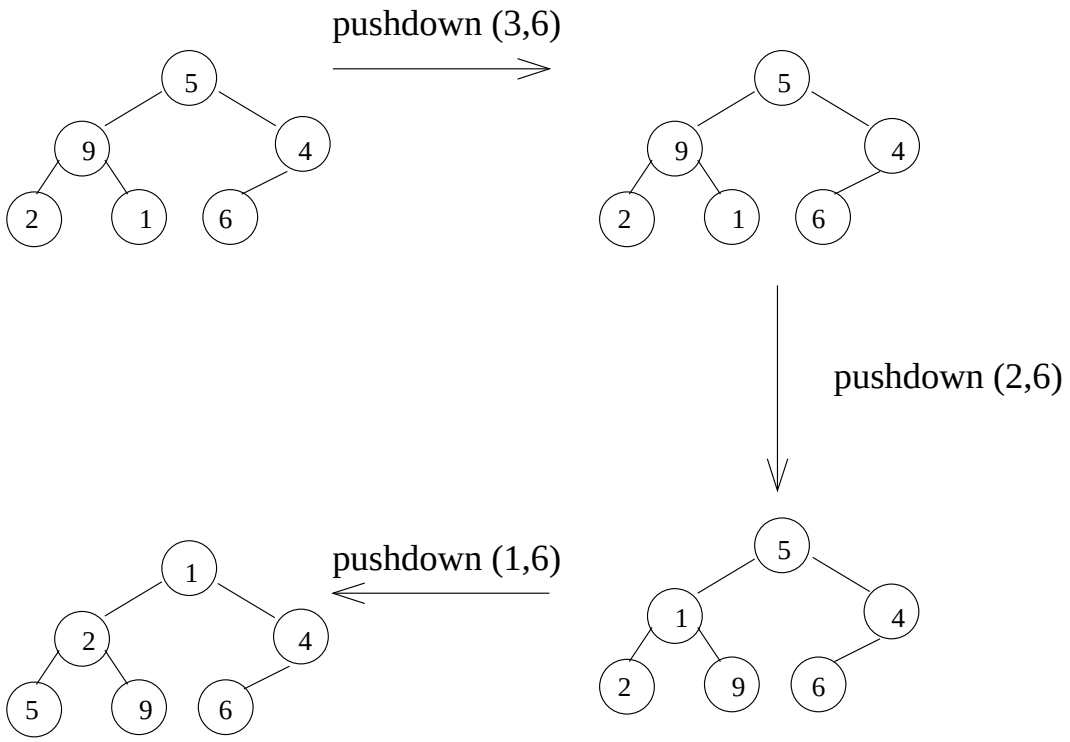


Figure 6.4: Creation of heap

Multiplying both sides by 2 yields

$$2S = \ 2h + 4(h-1) + 8(h-2) + ... + 2^h$$

Subtracting this above equation from the equation prior to that, we obtain

$$S = 2^{h+1} - 1 - (h+1)$$

It is easy to see that the above is an upper bound on the sum of heights of nodes of a complete binary tree. Since a complete binary tree of height h has between 2^h and 2^{h+1} nodes, the above sum is therefore O (n) where n is the number of nodes in the heap.

Since the worstcase complexity of the heap building algorithm is of the order of the sum of heights of the nodes of the heap built, we then have the worst-case complexity of heap building as O (n).

6.2 Binomial Queues

REF. Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, Volume 21, Number 4, pp.309-315, 1978.

REF. Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, Volume 7, Number 3, pp. 298-319, 1978.

- Support merge, insert, and deletemin operations in $O(\log n)$ worstcase time.
- A binomial queue is a forest of heap-ordered trees.
  - Each of the heap-ordered trees is of a constrained form known as a *binomial tree*.

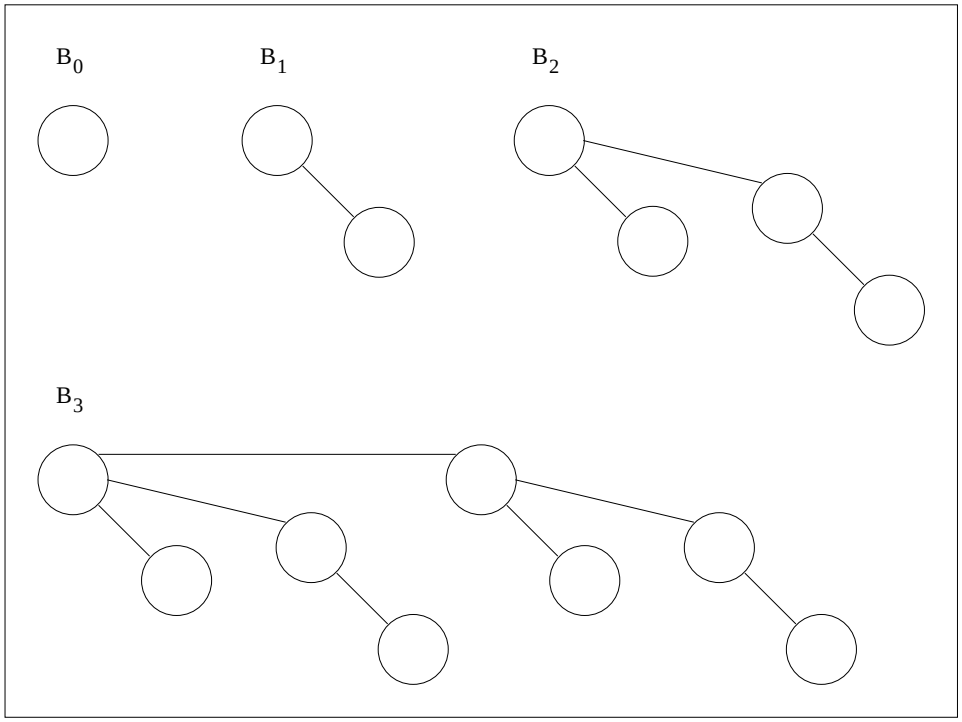


Figure 6.5: Examples of Binomial Trees

- A *binomial tree* of height 0 is a one-node tree; A binomial tree B_k of height k is formed by attaching a binomial tree B_{k-1} to the root of another binomial tree, B_{k-1}.
- See Figure 6.5 for an example of binomial tree.
- A binomial tree B_k consists of a root with children B_0, B_1, ... B_{k-1}.
- B_k has exactly 2^k nodes.
- Number of nodes at depth d is kC_d
- If we impose heap order on the binomial trees and allow at most one binomial tree of any height, we can uniquely represent a priority queue of any size by a forest of binomial trees.
- **Example**: A priority queue of size 13 could be represented by the forest B_3, B_2, B_0. A natural way of writing this is: 1101. See Figure 6.6 for a binomial queue with six elements.

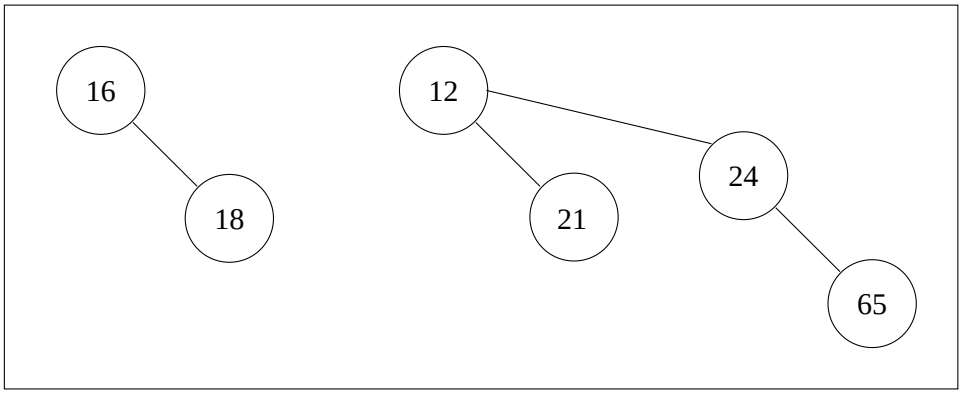


Figure 6.6: A binomial queue H_1 with six elements

6.2.1 Binomial Queue Operations

Find-min

This is implemented by scanning the roots of all the trees. Since there are at most $\log n$ different trees, this leads to a worstcase complexity of $0(\log n)$.

Alternatively, one can keep track of the current minimum and perform find-min in 0(1) time if we remember to update the minimum if it changes during other operations.

Merge

Let H_1: 6 elements

H_2: 7 elements

We are now left with

1 tree of height 0

3 trees of height 2

Note that of the 3 binomial trees of height 2, we could have any pair to get another binomial heap. Since merging two binomial trees takes constant time and there are $0(\log n)$ binomial trees, merge takes $0(\log n)$ in the worstcase. See Figures 6.7 and 6.8 for two examples.

Insertion

This is a special case of merging since we merely create a one-node tree and perform a merge.

Worstcase complexity therefore will be $O(\log n)$.

More precisely; if the priority queue into which the element is being inserted has the property that the smallest nonexistent binomial tree is B_i, the running time is proportional to $i + 1$.

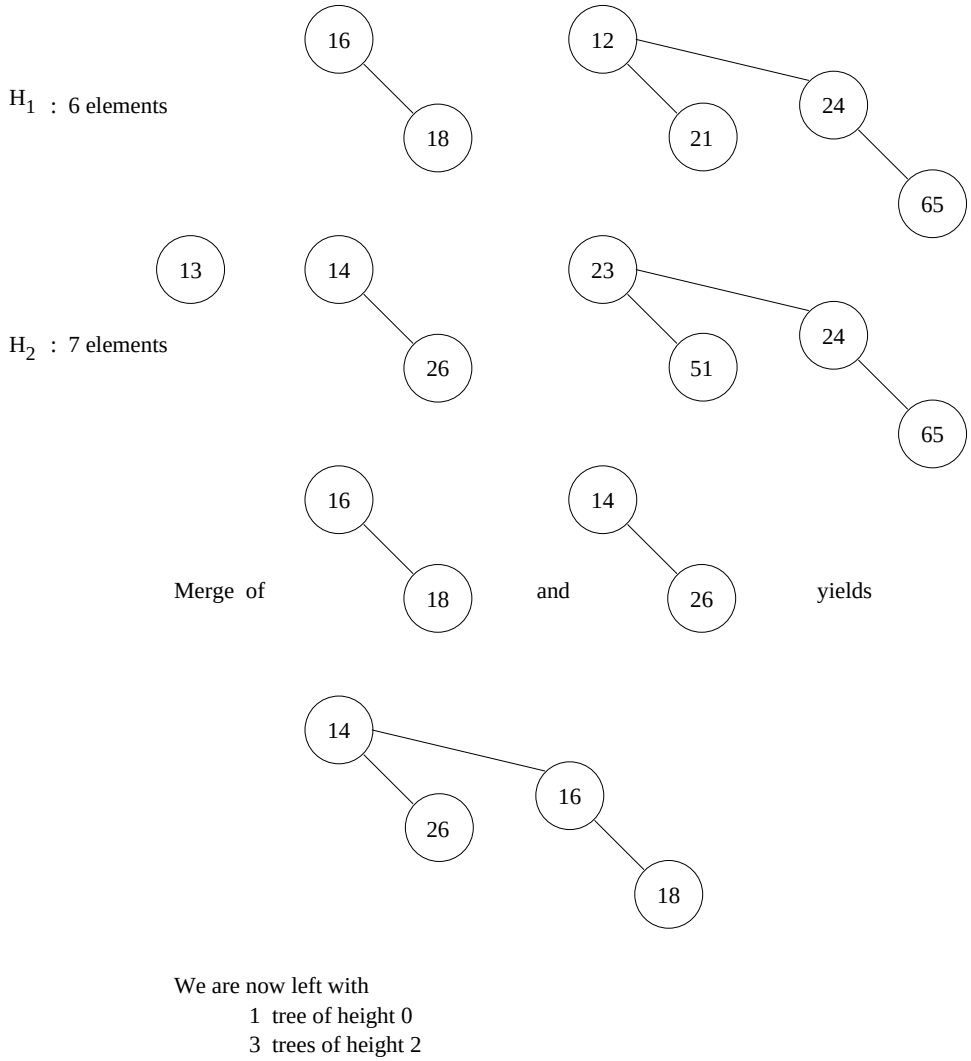


Figure 6.7: Examples of Merging

Eg: Insertion into H_3 (which has 13 elements) terminates in two steps.

Since each tree of a particular degree in a binomial queue is present with probability $\frac{1}{2}$, if we define the random variable X as representing the number of steps in an insert operation, then

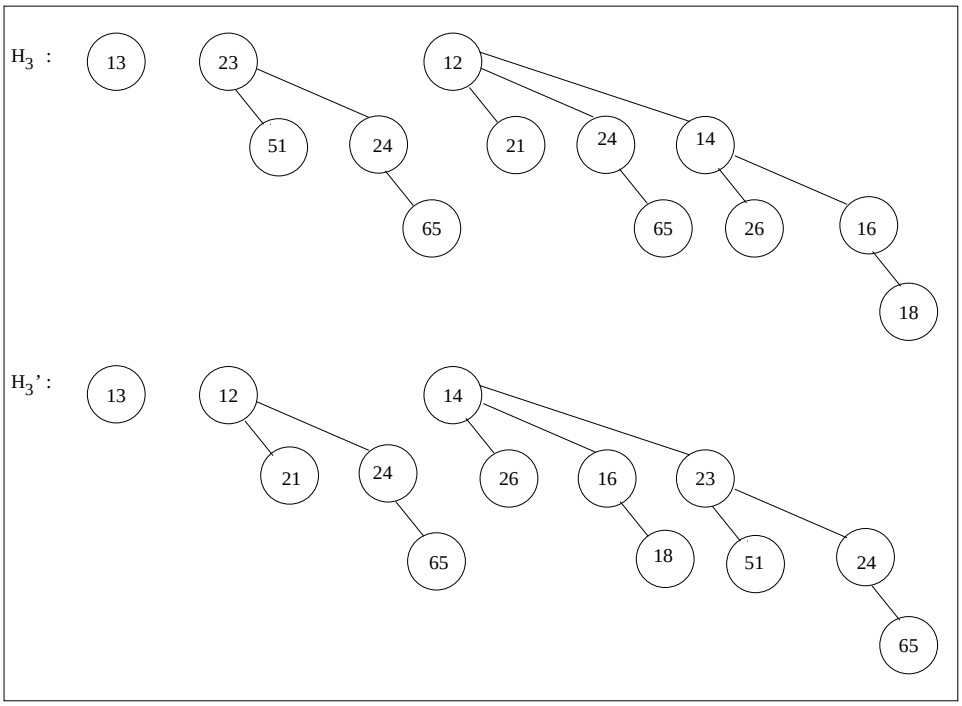


Figure 6.8: Merge of H_1 and H_2

$$
\begin{aligned}
X &= 1 \quad \text{with prob } \tfrac{1}{2} \ (B_0 \text{ not present}) \\
&= 2 \quad \text{with prob } \tfrac{1}{2} \ (B_0 \text{ not present}) \ (B_1 \text{ not present}) \\
&= 3 \quad \text{with prob } \tfrac{1}{8} \\
&\ \ \vdots
\end{aligned}
$$

Thus average number of steps in an insert operation = 2

Thus we expect an insertion to terminate in two steps on the average. Further more, performing n inserts on an initially empty binomial queue will take $0(n)$ worstcase time.

See Figures 6.9 and 6.10 for an example.

Deletemin

- First find the binomial tree with the smallest root. Let this be B_k. Let H be the original priority queue.

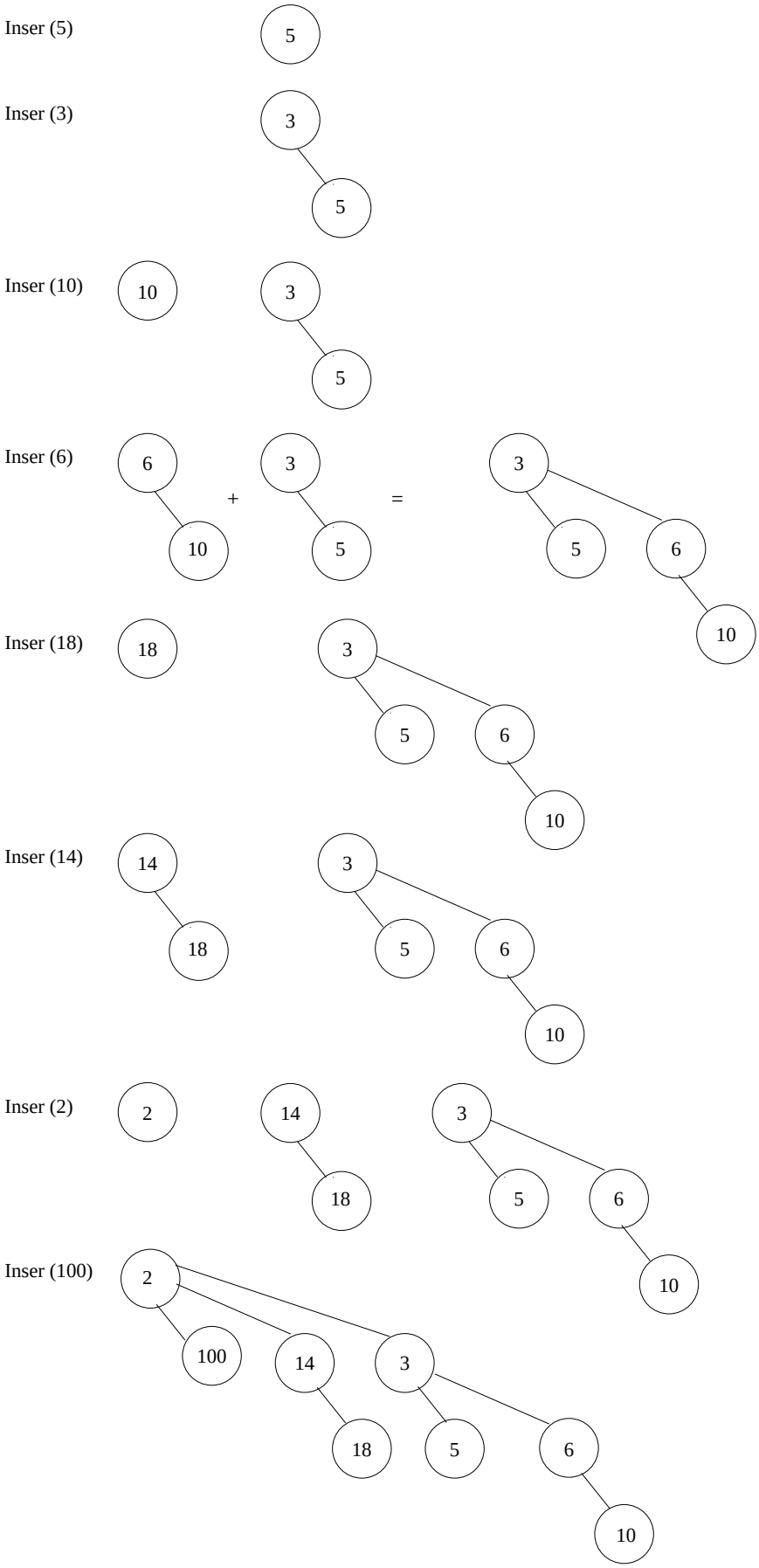


Figure 6.9: Examples of Inserts

- Remove B_k from the forest in H forming another binomial queue $H^{'}$.
- Now remove the root of B_k creating binomial trees B_0, B_1, ..., B_{k-1}, which collectively form a binomial queue $H^{''}$.
- Now, merge $H^{'}$ and $H^{''}$.

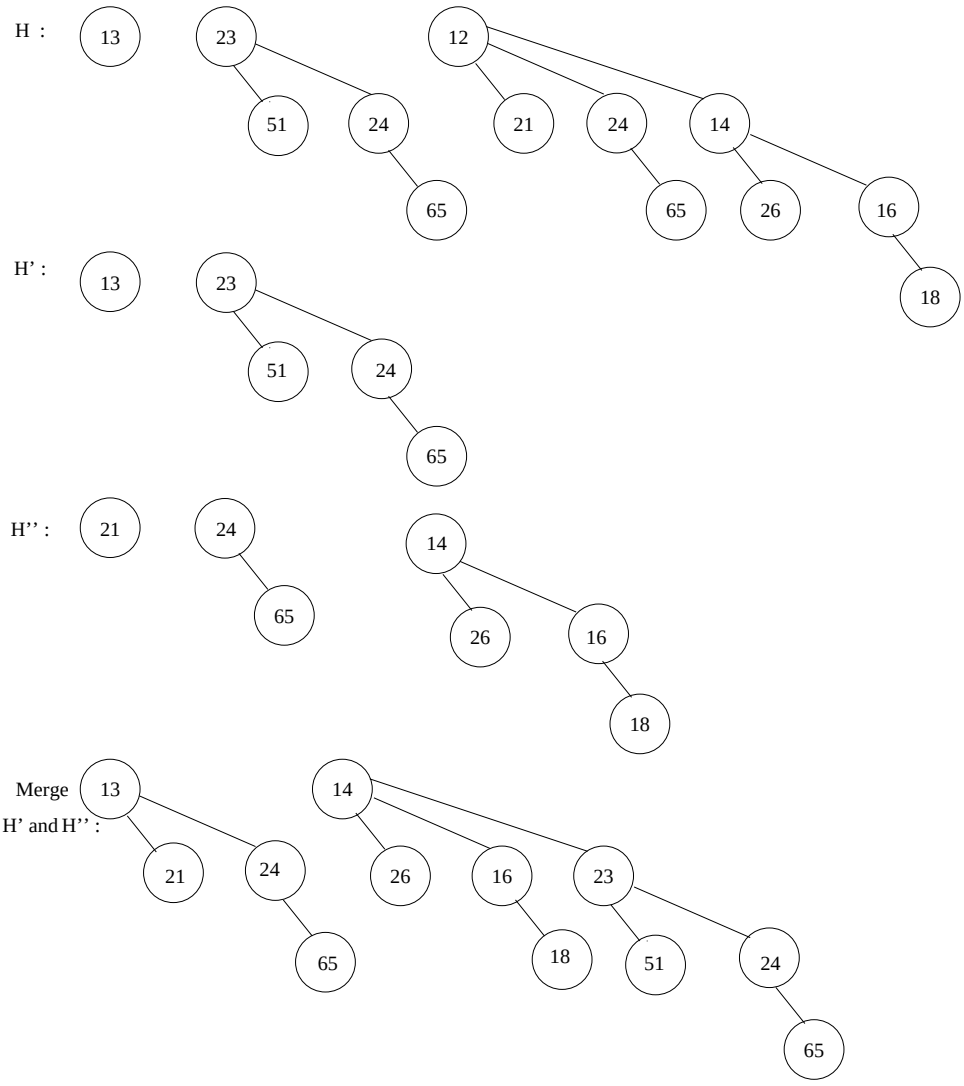


Figure 6.10: Merges of $H^{'}$ and $H^{''}$

It is easy to see that the worstcase complexity of deletemin is $0(\log n)$.

Implementation of a Binomial Queue

- deletemin operation requires ability to find all subtrees of the root. Thus children of each node should be available (say a linked list)
- deletemin requires that the children be ordered by the size of their subtrees.
- we need to make sure it is easy to merge tress. Two binomial trees can be merged only if they have the same size, hence the size of the tree must be stored in the root. While merging, one of the trees becomes the last child of the other, so we should keep track of the last child of each node. A good data structure to use is a circular doubly linked list what each node is of the following form:

data	first child	left sibling	right sibling	rank No. of children

6.2.2 Binomial Amortized Analysis

Amortized Analysis of Merge

- To merge two binomial queues, an operation similar to addition of binary integers is performed:

  At any stage, we may have zero, one, two, or three B_k trees, depending on whether or not the two priority queues contain a B_k tree and whether or not a B_k tree is carried over from the previous step.

  * if there is zero or more B_k tree, it is placed as a tree in the resulting binomial queue.
  * If there are two, they are merged into a B_{k+1} tree and carried over
  * If there are three, one is retained and other two merged.

Result 1:

- A binomial queue of n elements can be built by n successive insertions in $0(n)$ time.
- Brute force Analysis
  Define the cost of each insertions to be
  - 1 time unit + an extra unit for each linking step
    Thus the total will be n units plus the total number of linking steps.
  - The 1st, 3rd, ... and each odd-numbered step requires no linking steps since there is no B_0 present.
  - A quarter of the insertions require only one linking step: 2nd, 6th, 10, ...
  - One eighth of insertions require two linking steps.

  We could do all this and bound the number of linking steps by n.

  The above analysis will not help when we try to analyze a sequence of operations that include more than just insertions.
- Amortized Analysis

  Consider the result of an insertion.
  - If there is no B_0 tree, then the insertion costs one time unit. The result of insertion is that there is now a B_0 tree and the forest has one more tree.
  - If there is a B_0 tree but not B_1 tree, then insertion costs 2 time units. The new forest will have a B_1 tree but not a B_0 tree. Thus number of trees in the forest is unchanged.
  - An insertion that costs 3 time units will create a B_2 tree but destroy a B_0 and B_1, yielding one less tree in the forest.
  - In general, an insertion that costs c units results in a net increase of $2 - c$ trees. Since
    * a B_{c-1} tree is created

$*$ all B_i trees, $0 \leq i \leq c-1$ are removed.

Thus expensive insertions remove trees and cheap insertions create trees.

$$\begin{aligned} \textit{Let} \quad t_i &= \text{ cost of } i^{th} \text{ insertion} \\ c_i &= \text{ no. of trees in forest after } i^{th} \text{ insertion} \end{aligned}$$

We have

$$\begin{aligned} c_0 &= 0 \\ t_i + (c_i - c_{i-1}) &= 2 \end{aligned}$$

Result 2:

- The amortized running times of Insert, Delete-min, and Merge are $0(1)$, $0(\log n)$, and $0(\log n)$ respectively.

  Potential function = # of trees in the queue

  To prove this result we choose:

- Insertion

$$\begin{aligned} t_i &= \text{ actual cost of } i^{th} \text{ insertion} \\ c_i &= \text{ no. of trees in forest after } i^{th} \\ a_i &= t_i + (c_i - c_{i-1}) \\ a_i &= 2 \quad \forall\ i \\ \textstyle\sum_{i=1}^{n} t_i &= \textstyle\sum_{i=1}^{n} a_i - (c_n - c_0) \\ &= 2n - (c_n - c_0) \end{aligned}$$

  As long as $(c_n - c_0)$ is positive, we are done.

  In any case $(c_n - c_0)$ is bounded by $\log n$ if we start with an empty tree.

- Merge:

  Assume that the two queues to be merged have n_1 and n_2 nodes with T_1 and T_2 trees. Let $n = n_1 + n_2$. Actual time to perform merge is given by:

$$\begin{aligned} t_i &= 0(\log n_1 + \log n_2) \\ &= 0(max(\log n_1, \log n_2) \\ &= 0(\log n) \end{aligned}$$

$(c_i - c_{i-1})$ is at most $(\log n)$ since there can be at most $(\log n)$ trees after merge.

- Deletemin:
  The analysis here follows the same argument as for merge.

6.2.3 Lazy Binomial Queues

Binomial queues in which merging is done lazily.

Here, to merge two binomial queues, we simply concatenate the two lists of binomial trees. In the resulting forest, there may be several trees of the same size.

Because of the lazy merge, merge and insert are both worstcase $0(1)$ time.

- Deletemin:
  - Convert lazy binomial queue into a standard binomial queue
  - Do deletemin as in standard queue.

Fibonacci Heaps

Fibonacci heap supports all basic heap operations in $0(1)$ amortized time, with the exception of deletemin and delete which take $0(\log n)$ amortized time.

Fibonacci heaps generalize binomial queues by adding two new concepts:

- A different implementation of decrease-key
- Lazy merging: Two heaps are merged only when it is required.

It can be shown in a Fibonacci heap that any node of rank $r \geq 1$ has at least F_{r+1} descendents.

6.3 To Probe Further

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
2. Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996. Indian Edition published by Prentice Hall of India, 1998.
3. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.
4. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.
5. Ellis Horowitz, Sartaz Sahni, and Rajasekaran. *Fundamentals of Computer Algorithms*. W.H. Freeman and Company, 1998. Indian Edition published by Galgotia Publications, 2000.
6. Y. Langsam, M.J. Augenstein, and A.M. Tenenbaum. *Data Structures using C and C++*. Second Edition, Prentice Hall, 1996.XS Indian Edition published by Prentice Hall of India, 2000.
7. Sataj Sahni. *Data Structures, Algorithms, and Applications in Java*. McGraw-Hill Higher Education, 2000.
8. Thomas A. Standish. *Data Structures in Java*. Addison-Wesley, 1998. Indian Edition published by Addison Wesley Longman, 2000.
9. Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, Volume 7, Number 3, pp. 298-319, 1978.

10. Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, Volume 34, Number 3, pp. 596-615, 1987.

11. Robert W. Floyd. Algorithm 245 (TreeSort). *Communications of the ACM*, Volume 7, pp.701, 1964.

12. Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, Volume 21, Number 4, pp.309-315, 1978.

13. J.W.J. Williams. Algorithm 232 (Heapsort). *Communications of the ACM*, Volume 7, pp.347-348, 1964.

6.4 Problems

1. Show the result of inserting 10, 5, 1, 17, 20, 15, 6, 4, 2, one at a time, into an initially empty binary heap. For the same input, show the result of building the heap by the linear time algorithm discussed.

2. Design an algorithm to find all nodes less than some value, x, in a binary heap. Your algorithm should run in O (m), worst-case time, where m is the number of nodes output.

3. A min-max heap is a data structure that supports both deletemin and deletemax in O (log n) worst-case time. The structure is identical to a binary heap, but the heap order property is that, for any node x, at even depth, the key stored at x is the smallest in its subtree, and for any node x at odd depth, the key stored at x is the largest in its subtree.

   (a) How do we find minimum and maximum elements

   (b) Present an algorithm to insert a new node into the min-max heap

   (c) Give an algorithm to perform deletemin and deletemax

4. Prove that a binomial tree B_k has binomial trees $B_0, B_1, \ldots, B_{k-1}$ as children of the root.

5. Prove that a binomial tree of height k has $stackrel(k)(d)$ nodes at depth d.

6. Present an algorithm to insert m nodes into a binomial queue of n elements in $O(m+\log n)$ worst-case time. Prove your bound.

7. It is required to increment the value of the roots of all binomial trees in a binomial queue. What is the worst-case running time of this in terms of the number of elements in the binomial queue.

8. Design an algorithm to search for an element in a binary heap. What is the worst-case complexity of your algorithm. Can you use the heap property to speedup the search process?

9. Prove that a binomial queue with n items has at most $\log n$ binomial trees, the largest of which contains $2^{\log n}$ items.

10. How do you implement the following operations in a binary heap?

    - Decrease the key at some position
    - Increase the key at some position
    - Remove element at some position

11. What is the worstcase complexity of each problem below, given a binomial queue with n elements? Explain with reason or example in each case.

    (a) Find the second smallest element

    (b) Find the second largest element

12. Consider *lazy* binomial queues where the *merge* operation is lazy and it is left to the *deletemin* operation to convert the lazy binomial queue into a binomial queue in standard form. Obviously, the merge operation is $O(1)$, but the deletemin operation is not worst case $O(\log n)$ anymore. However, show that deletemin is still $O(\log n)$ in amortized time.

6.5 Programming Assignments

6.5.1 Discrete Event Simulation

Priority queues provide a *natural* dats structure to use in event list maintenance in discrete event simulations. The objective of this assignment is to develop event list algorithms using priority queues and use these as a component in a discrete event simulation package. Singly linked lists are commonly used in implementing event lists. One can use binary heaps and binomial queues for better performance.

Write a simulation package for a discrete event system (with a large number of events) and compare the performance of naive linked lists, binary heaps, and binomial queues in event list related aspects of the simulation.

Chapter 7

Directed Graphs

In numerous situations in Computer Science, Mathematics, Engineering, and many other disciplines, *Graphs* have found wide use in representing arbitrary relationships among data objects. There are two broad categories of graphs: Directed graphs (digraphs) and Undirected graphs. In this chapter, we study some important graph-based algorithms in Computer Science and examine data structure related issues.

7.1 Directed Graphs

A directed graph or digraph G comprises

1. a set of vertices V
2. a set of directed edges or arcs, E (an arc is an ordered pair of vertices)

Example: See Figure 7.1

$$V = \{1, 2, 3, 4\}$$
$$E = \{(1,2), (1,3), (2,4), (3,2), (4,3)\}$$

- If there is an arc (v, w), we say w is **adjacent** to v.
- A **path** is a sequence of vertices $v_1, v_2, \ldots v_n$ such that the vertex pairs $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ are arcs in the graph. The **length** of

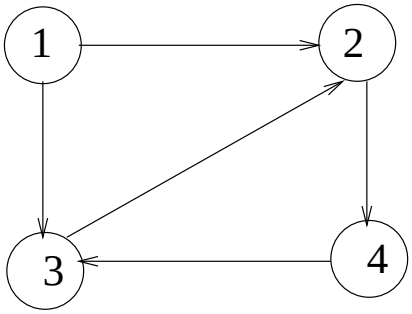


Figure 7.1: A digraph with 4 vertices and 5 arcs

a path is the number of arcs on the path. A single vertex v by itself denotes a path of length zero.

- A path $v_1, v_2, \ldots, v_n$ is said to be **simple** if the vertices are distinct, except possibly v_1 and v_n. If $v_1 = v_n$ and $n > 1$, we call such a path a **simple cycle**.

7.1.1 Data Structures for Graph Representation

1. Adjacency Matrix: The matrix is of order $(n \times n)$ where n is the number of vertices. The adjacency matrix for the graph in Figure 7.2 is

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

2. Adjacency List: Figures 7.2 and 7.3 provide two different adjacency list representations for the graph of Figure 7.1

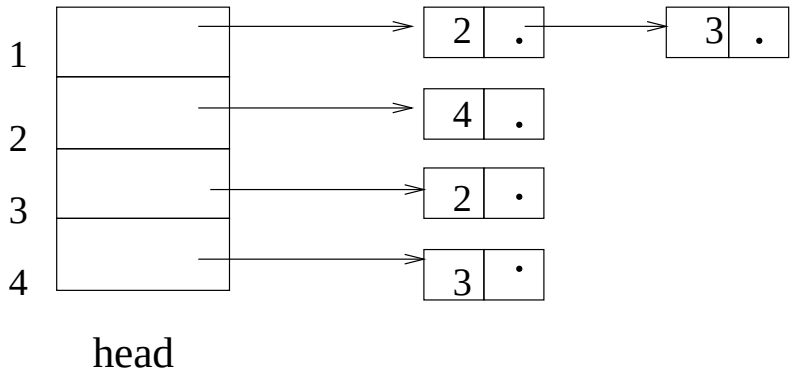


Figure 7.2: Adjacency list representation of the digraph

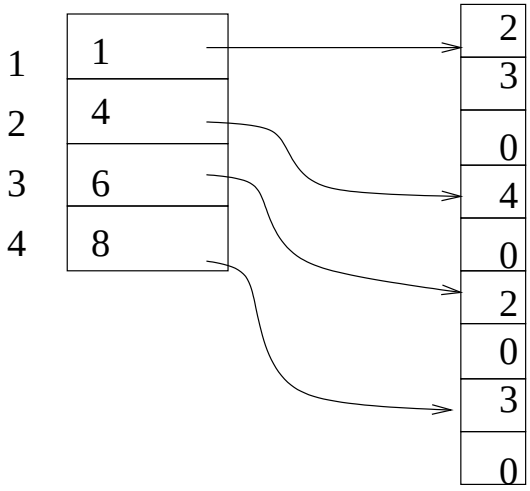


Figure 7.3: Adjacency list using cursors

7.2 Shortest Paths Problem

Given : A digraph $G = (V, E)$ in which each arc has a non-negative cost and one vertex is designated as the **source**.

Problem : To determine the cost of the shortest path from the source to every other vertex in V (where the length of a path is just the sum of the costs of the arcs on the path).

7.2.1 Single Source Shortest Paths Problem: Dijkstra's Algorithm

REF. E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, Volume 1, pp. 269-271, 1959.

- Greedy algorithm
- It works by maintaining a set S of "special" vertices whose shortest distance from the source is already known. At each step, a "non-special" vertex is absorbed into S.
- The absorption of an element of $V - S$ into S is done by a greedy strategy.
- The following provides the steps of the algorithm.

Let

$$\begin{aligned} V &= \{1, 2, \ldots, n\} \text{ and source} = 1 \qquad (7.1) \\ C[i, j] &= \text{Cost of the arc } (i, j) \text{ if the arc } (i, j) \text{ exists; otherwise } \infty \end{aligned}$$

```
{
        S= { 1 };
        for (i = 2; i < n; i++)
            D[i] = C[1,i];
```

```
    for (i=1; i < = n-1; i++)
   {
     choose a vertex w ∈ V–S such that D[w] is a minimum;
     S = S ∪ {w };
     for each vertex v ∈ V–S
       D[v] = min (D[v], D[w] + C[w, v])
    }
}
```

- The above algorithm gives the costs of the shortest paths from source vertex to every other vertex.
- The actual shortest paths can also be constructed by modifying the above algorithm.

Theorem: Dijkstra's algorithm finds the shortest paths from a single source to all other nodes of a weighted digraph with positive weights.

Proof: Let V = 1, 2, ..., n and 1 be the source vertex. We use mathematical induction to show that

(a) If a node i ($\neq 1$) $\in S$, then D[i] gives the length of the shortest path from the source to i.

(b) if a node $i \notin S$, then D[i] gives the length of the shortest special path from the source to i.

Basis: Initially S = 1 and hence (a) is vacuously true. For $i \in S$, the only special path from the source is the direct edge if present from source to i and D is initialized accordingly. Hence (b) is also true.

Induction for condition (a)

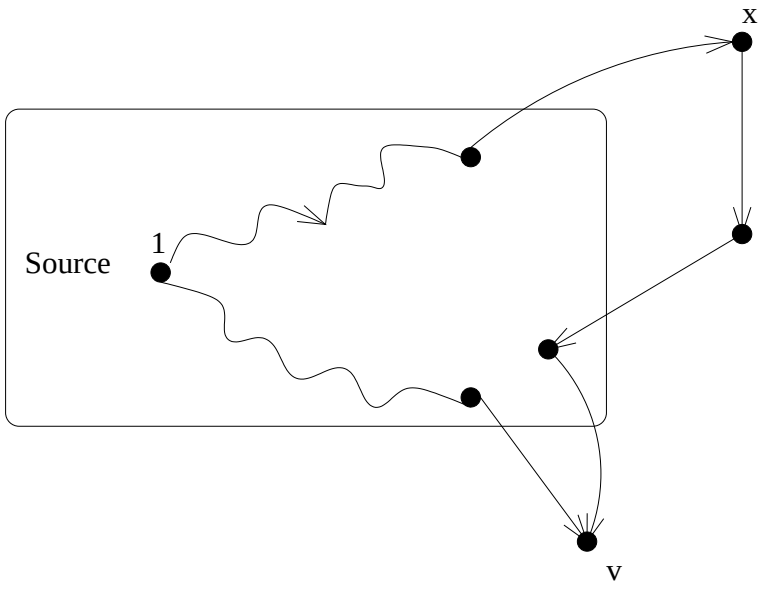


Figure 7.4: The shortest path to v cannot visit x

- The induction hypothesis is that both (a) and (b) hold just before we add a new vertex v to S.
- For every node already in S before adding v, nothing changes, so condition (a) is still true.
- We have to only show (a) for v which is just added to S.
- Before adding it to S, we must check that D[v] gives the length of the shortest path from source to v. By the induction hypothesis, D[v] certainly gives the length of the shortest special path. We therefore have to verify that the shortest path from the source to v does not pass through any nodes that do not belong to S.
- Suppose to the contrary. That is, suppose that when we follow the shortest path from source to v, we encounter nodes not belonging to S. Let x be the first such node encountered (see Figure 7.4). The initial segment of the path from source to x is a special path and by part (b) of the induction hypothesis, the length of this path is D[x]. Since edge weights are no-negative, the total distance to v via x is greater than or equal to D[x]. However since the algorithm has chosen v ahead of x, $D[x] \geq D[v]$. Thus the path via x cannot be shorter than the shortest special path leading to v.

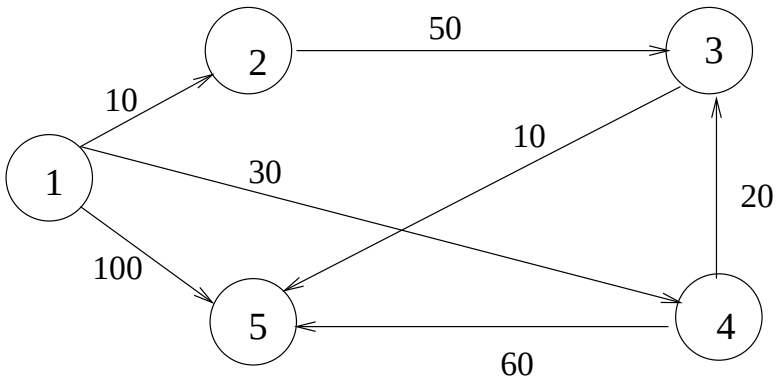


Figure 7.5: A digraph example for Dijkstra's algorithm

Induction step for condition (b): Let $\omega \neq v$ and $\omega \in S$. When v is added to S, these are two possibilities for the shortest special path from source to w:

1. It remains as before
2. It now passes through v (and possibly other nodes in S)

In the first case, there is nothing to prove. In the second case, let y be the last node of S visited before arriving at w. The length of such a path is D[y] + C[y,w].

- At first glance, to compute the new value of d[w], it looks as if we should compare the old value of D[w] with D[y] + C[y,w] for every $y \in S$ (including v)
- This comparison was however made for all $y \in S$ except v, when y was added to S in the algorithm. Thus the new value of D[w] can be computed simply by comparing the old value with D[v] + C[v,w]. This the algorithm does.

When the algorithm stops, all the nodes but one are in S and it is clear that the vector D[1], D[2], ..., D[n]) contains the lengths of the shortest paths from source to respective vertices.

Example: Consider the digraph in Figure 7.5.

Initially:

$$S = \{1\} \quad D[2] = 10 \quad D[3] = \infty \quad D[4] = 30 \quad D[5] = 100$$

Iteration 1

Select $w = 2$, so that $S = \{1, 2\}$

$$\begin{aligned} D[3] &= \min(\infty, D[2] + C[2,3]) = 60 && (7.2) \\ D[4] &= \min(30, D[2] + C[2,4]) = 30 && (7.3) \\ D[5] &= \min(100, D[2] + C[2,5]) = 100 \end{aligned}$$

Iteration 2

Select $w = 4$, so that $S = \{1, 2, 4\}$

$$\begin{aligned} D[3] &= \min(60, D[4] + C[4,3]) = 50 && (7.4) \\ D[5] &= \min(100, D[4] + C[4,5]) = 90 \end{aligned}$$

Iteration 3

Select $w = 3$, so that $S = \{1, 2, 4, 3\}$

$$D[5] = \min(90, D[3] + C[3,5]) = 60$$

Iteration 4

Select $w = 5$, so that $S = \{1, 2, 4, 3, 5\}$

$$\begin{aligned} D[2] &= 10 && (7.5) \\ D[3] &= 50 && (7.6) \\ D[4] &= 30 && (7.7) \\ D[5] &= 60 \end{aligned}$$

Complexity of Dijkstra's Algorithm

With adjacency matrix representation, the running time is $O(n^2)$ By using an adjacency list representation and a **partially ordered tree** data structure for organizing the set $V - S$, the complexity can be shown to be

$$O(e \log n)$$

where e is the number of edges and n is the number of vertices in the digraph.

7.2.2 Dynamic Programming Algorithm

REF. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

REF. Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, Volume 16, Number 1, pp. 87-90, 1958.

Consider a directed acyclic graph (digraph without cycles) with non-negative weights on the directed arcs.

Given a destination vertex z, the problem is to find a shortest cost path from each vertex of the digraph. See Figure 7.6.

Let

$$C(i,j) = \text{Cost of the directed arc from vertex } i \text{ to vertex } j \ (\infty \text{ in case there is no link}) \quad (7.8)$$

$$J(i) = \text{Optimal cost of a path from vertex } i, \text{ to the destination vertex } z \quad (7.9)$$

$J(i)$ Satisfies :

$$J(z) = 0$$

and if the optimal path from i to z traverses the link (i,j) then

$$J(i) = C(i,j) + J(j)$$

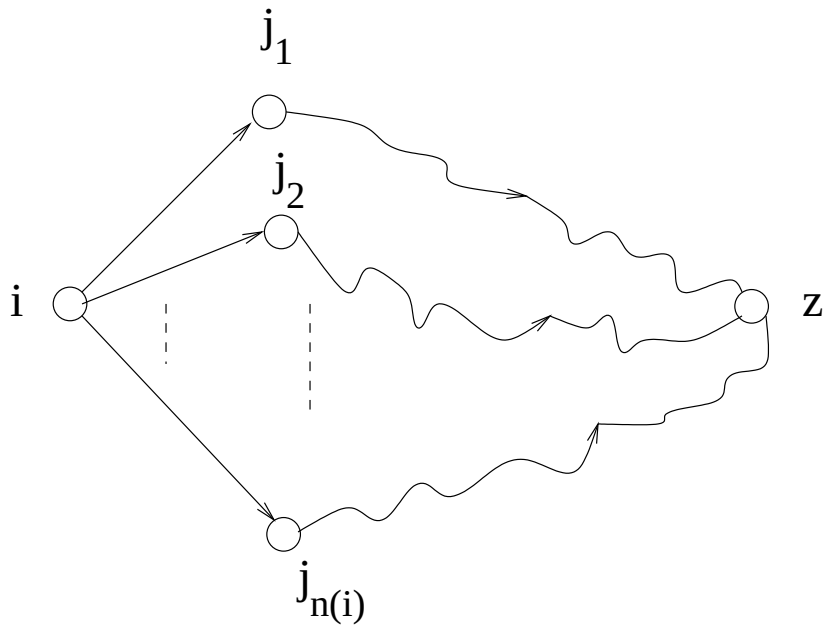


Figure 7.6: Principle of dynamic programming

Suppose that for each vertex j such that the link (i, j) exists, the optimal cost $J(j)$ is known. Then, the principle of DP immediately implies:

$$J(i) = \min_j [C(i, j) + J(j)]$$

A DP algorithm based on the above observations:

1. Set $J(z) = 0$. At this point, this is the only node whose cost has been computed.
2. Consider vertices $i \in V$ such that
   - $J(i)$ has not yet been found
   - for each vertex j such that a link (i, j) exists, $J(j)$ is already computed

   Assign $J(i)$ according to

   $$J(i) = \min_j [C(i, j) + J(j)]$$

3. Repeat Step 2 until all vertices have their optimal costs computed.

Example: Consider the digraph in Figure 7.7

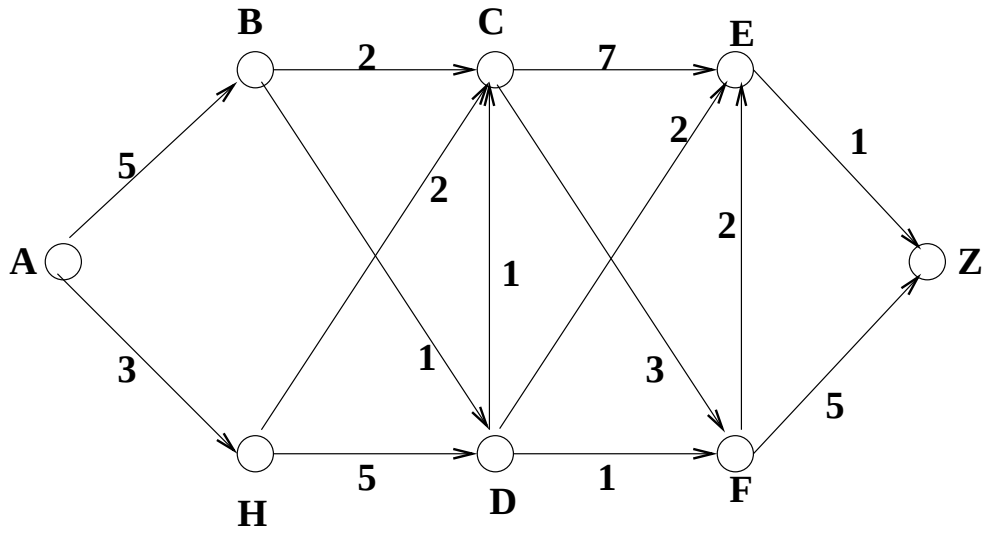


Figure 7.7: An example digraph to illustrate dynamic programming

Step 1. $J(Z) = 0$

Step 2. E is the only vertex such that $J(j)$ is known $\forall$ j such that $C(E, j) \neq 0$

$$J(E) = C(E, Z) + J(Z) = 1$$

Step 3. Select F.

$$\begin{aligned} J(F) &= \min \begin{cases} C(F, Z) + J(Z) = 5 \\ C(F, E) + J(E) = 3 \end{cases} \\ &= \min(5, 3) = 3 \end{aligned} \tag{7.10}$$

This yields the optimal path $F \rightarrow E \rightarrow Z$.

Step 4. Select C.

$$\begin{aligned} J(C) &= \min \begin{cases} C(C, E) + J(E) = 8 \\ C(C, F) + J(F) = 6 \end{cases} \\ &= \min(8, 6) = 6 \end{aligned} \tag{7.11}$$

This yields the optimal path $C \rightarrow F \rightarrow E \rightarrow Z$.

Step 5. Select D

$$J(D) = \min \begin{cases} C(D,C) + J(C) = 7 \\ C(D,E) + J(E) = 3 \\ C(D,F) + J(F) = 4 \end{cases} = 3$$

This yields the optimal path $D \to E \to Z$.

$\vdots$

Eventually, we get all the optimal paths and all the optimal costs.

Complexity of the Algorithm

- It is easy to see that the algorithm has a worst case complexity of $O(n^2)$, where n is the number of vertices.
- Limitation of the algorithm
  - doesn't work if there are cycles in the digraph.

7.2.3 All Pairs Shortest Paths Problem: Floyd's Algorithm

REF. Robert W Floyd. Algorithm 97 (Shortest Path). *Communications of the ACM*, Volume 5, Number 6, pp. 345, 1962.

Given : A digraph $G = (V, E)$ in which each arc $v \to w$ has a nonnegative cost $C[v, w]$

To find : For each ordered pair of vertices (v, w), the smallest length of any path from v to w

Floyd's Algorithm

- This is an $O(n^3)$ algorithm, where n is the number of vertices in the digraph.

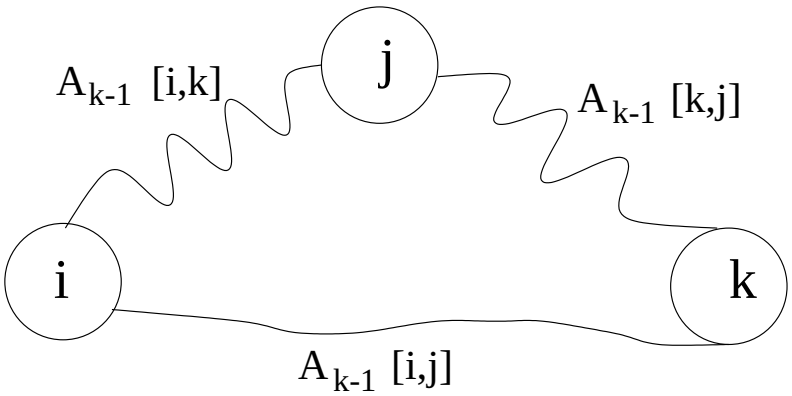


Figure 7.8: Principle of Floyd's algorithm

- Uses the principle of Dynamic Programming
- Let $V = \{1, 2, \ldots, n\}$. The algorithm uses a matrix $A[1..n][1..n]$ to compute the lengths of the shortest paths.
- Initially,

$$\begin{aligned} A[i,j] &= C[i,j] \text{ if } i \neq j \\ &= 0 \quad\quad \text{if } i = j \end{aligned} \tag{7.12}$$

  Note that $C[i,j]$ is taken as ∞ if there is no directed arc from i to j.

- The algorithm makes n passes over A. Let $A_0, A_1, \ldots, A_n$ be the values of A on the n passes, with A_0 being the initial value. Just after the $k-1^{th}$ iteration,

$$A_{k-1}[i,j] = \text{Smallest length of any path from vertex } i \text{ to vertex } j \tag{7.13}$$
$$\text{that does not pass through the vertices } k, k+1, \ldots n \tag{7.14}$$
$$\text{(i.e. that only passes through possibly } 1, 2, \ldots\ k-1)$$

  See Figure 7.8

- The k^{th} pass explores whether the vertex k lies on an optimal path from i to j, $\forall\ i, j$
- We use

$$A_k[i,j] = \min \begin{cases} A_{k-1}[i,j] \\ A_{k-1}[i,k] + A_{k-1}[k,j] \end{cases}$$

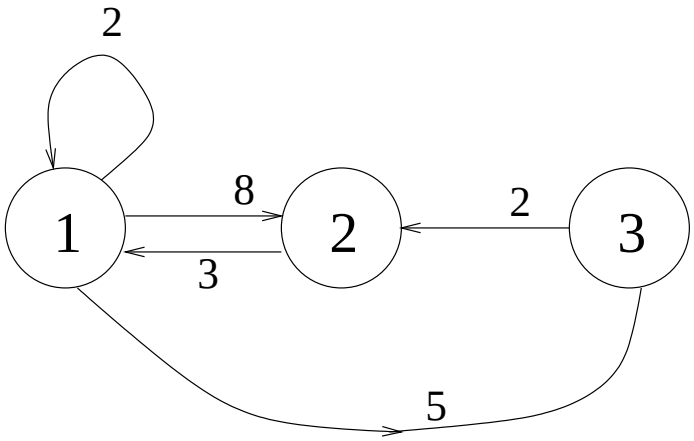


Figure 7.9: A digraph example for Floyd's algorithm

- The algorithm:

```
void Floyd (float C[n-1][n-1], A[n-1][n-1])
 {  int i, j, k;
    { for (i = 0, i ≤ n - 1; i++)
       for (j = 0; j ≤ n - 1, j++)
          A[i, j] = C[i, j];
    for (i = 0; i ≤ n - 1; i++)
          A[i, i] = 0;
    for (k = 0; k ≤ n - 1; k++);
     {
       for (i = 0; i ≤ n - 1; i++)
        {
          for (j = 0; j ≤ n - 1, j++)
             if (A[i, k] + A[k, j] < A[i, j])
                A[i, j] = A[i, k] + A[k, j]
        }
       }
     }
 }
```

Example: See Figure 7.9.

$$C = \begin{bmatrix} 2 & 8 & 5 \\ 3 & \infty & \infty \\ \infty & 2 & \infty \end{bmatrix}$$

$$\begin{aligned} A_0 &= \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix}; \quad A_1 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix} \\ A_2 &= \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix}; \quad A_3 = \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix} \end{aligned} \tag{7.15}$$

- With adjacency matrix representation, Floyd's algorithm has a worst case complexity of $O(n^3)$ where n is the number of vertices
- If Dijkstra's algorithm is used for the same purpose, then with an adjacency list representation, the worst case complexity will be $O(ne \log n)$. Thus if e is $O(n^2)$, then the complexity will be $O(n^3 \log n)$ while if e is $O(n)$, then the complexity is $O(n^2 \log n)$.

7.3 Warshall's Algorithm

REF. Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, Volume 9, Number 1, pp. 11-12, 1962.

Given a digraph G = (V,E), determine
for each $i, j \epsilon V$ ether or not there exists a path of length one
or more from vertex i to vertex j

Def: Given the adjacency matrix C of any digraph C = (v,E), the matrix A is called the transitive closure of C if $\forall\ i,\ j\ \epsilon\ V$,

$A[i,j]$ $= 1$ if there is a path of length one or more from vertex i to vertex j
$= 0$ *otherwise*

Warshall's algorithm enables to compute the transitive closure of the adjacency matrix f any digraph. Warshall's algorithm predates Floyd's algorithm and simple uses the following formula in the k^{th} passes of Floyd's algorithm:

$$A_k[i,j] = A_{k-1}[i,j] \vee (A_{k-1}[i,k] \wedge A_{k-1}[k,j])$$

where the matrix elements are treated as boolean values 0 or 1 and the symbols $\vee$ and $\wedge$ denote "logical or" and "logical and" respectively. It is easy to see that Warshall's algorithm has a worst case complexity of $O(n^3)$ where n is the number of vertices of the graph.

7.4 Depth First Search and Breadth First Search

- An important way of traversing all vertices of a digraph, with applications in many problems.
- Let $L[v]$ be the adjacency list for vertex v. To do a depth first search of all vertices emanating from v, we have the following recursive scheme:

```
void dfs (vertex v)
  {
    vertex w;
    mark v as visited;
    for each vertex w ∈ L[v]
      dfs(w)
  }
```

- To do a dfs on the entire digraph, we first unmark all vertices and do a dfs on all unmarked vertices:

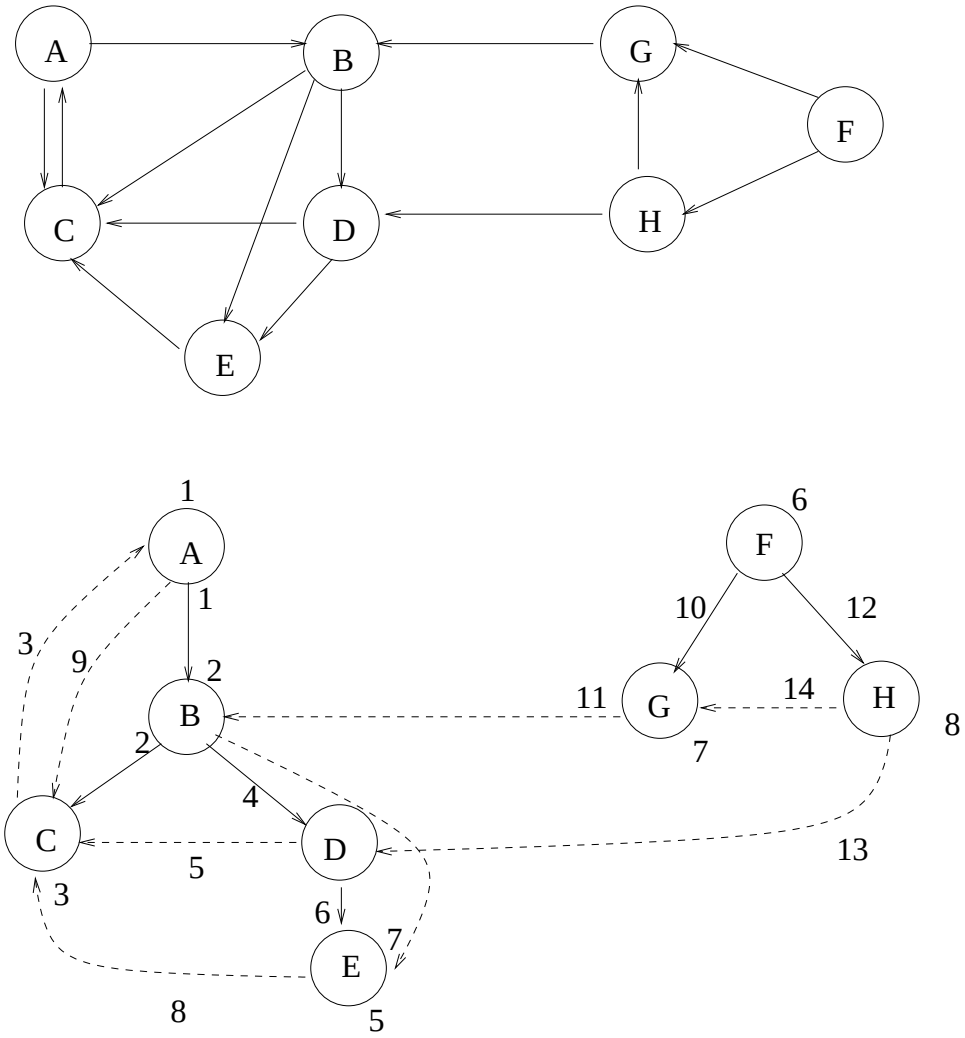


Figure 7.10: Depth first search of a digraph

```
{
    for v ∈ V
      mark v as unvisited;
    for v ∈ V
      if (v is unvisited)
        dfs(v);
 }
```

Example: See Figure 7.10.

DFS Arcs:

- Tree Arcs: $a \to b, N(a) < N(b)$ leading to unvisited vertices
- Non-Tree Arcs:
  - back arcs: $a \to b, N(a) \geq N(b)$, b is an ancestor
  - forward arcs: $a \to b, N(a) < N(b)$, b is a proper descendant
  - cross arcs: $a \to b$. Neither ancestor nor descendant

7.4.1 Breadth First Search

- Level by level traversal
- A queue data structure is used
- The complexity of both DFS and BFS is O(e).

Implementation of Breadth-First Search

```
void bfs (v)   /* visits all vertices connected to v
                  in breadth-first fashion */
```

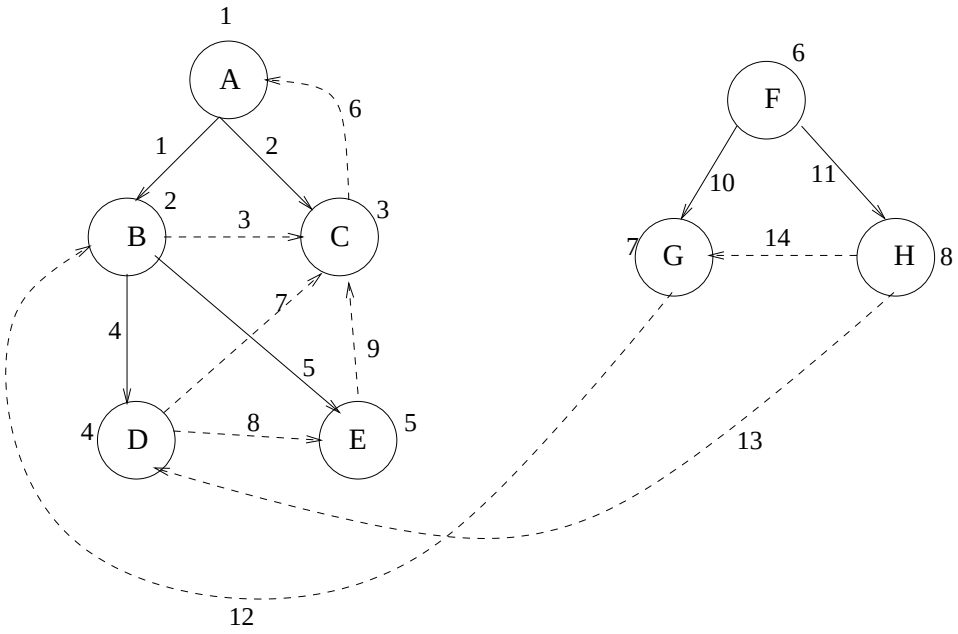


Figure 7.11: Breadth-first search of the digraph in Figure 7.11

```
vertex x, y;
vertexqueue Q;
{
  mark [v] = visited ;
  enqueue (v, Q);
  while (Q not empty)
   { x = front (Q) ;
     dequeue (Q) ;
     for (each vertex y adjacent to x)
       if  (mark[y] = unvisited)
           {
            mark[y] = visited ;
            enqueue(y,Q) ;
            insert ((x,y), T)
           }
   }
}
```

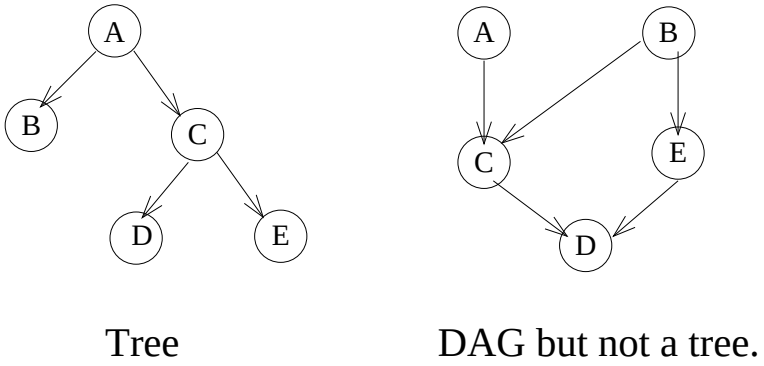


Figure 7.12: Examples of directed acyclic graphs

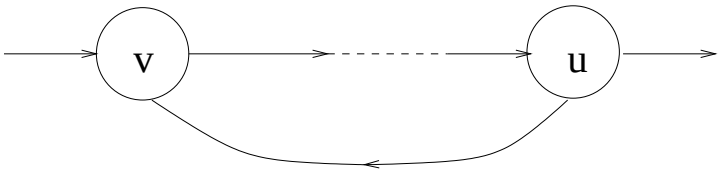


Figure 7.13: A cycle in a digraph

Example: See Figure 7.11.

7.5 Directed Acyclic Graphs

- Digraph without any cycles.
- More general than trees
- See Figure 7.12 for some examples
- Useful in many applications, such as
  1. Representing syntactic structure of arithmetic expressions
  2. Representing task graphs
  3. Precedence relations in many scheduling applications

7.5.1 Test for Acyclicity

Result:

A digraph is acyclic if and only if its first search does not have back arcs.

Proof:

First we prove that

backarc $\Longrightarrow$ cycle.

Let $(u \Longrightarrow w)$ be a backarc. This means that w is an ancestor of v. Thus $(w, ..., v, w)$ will be a cycle in the digraph.

Next we show that

cycle $\Longrightarrow$ backarc.

Suppose G is cyclic. Consider a cycle and let v be the vertex with the lowest *dfnumber* on the cycle. See Figure 7.13. Because v is on a cycle, there is a vertex u such that $(u \Longrightarrow v)$ is an edge. Since v has the lowest *dfnumber* among all vertices on the cycle, u must be an descendant of v.

- it can not be a tree arc since $dfnumber(v) \leq dfnumber(u)$
- it can not be a forward arc for the same reason
- it can not be a cross arc since v and u are on the same cycle.

Note that the above test for acyclicity has worst case complexity $O(e)$.

7.5.2 Topological Sort

Topological sort is a process of assigning a linear ordering to the vertices of a DAG so that if there is an arc from vertex i to vertex j, then i appears before j in the linear ordering

- Useful in scheduling applications
- **Example**: Consider the DAG in Figure 7.14. A topological sort is given by: B, A, D, C, E. There could be several topological sorts for a given DAG
- Topological sort can be easily accomplished by simply including an additional statement in the depth first search procedure of the given graph.

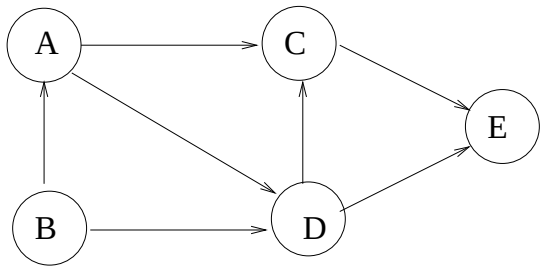


Figure 7.14: A digraph example for topological sort

- Let number [vertex] be the number that we assign in topological sort. We use a global integer variable n, whose initial value is zero.

```
int n = 0;
   void topsort (vertex v);
        /* assigns numbers to vertices accessible from v in
        reverse topological order */
     vertex w;
   {
      mark[v] = visited;
      for (w ∈ L[v])
        if (mark[w] == unvisited)
           topsort (w);
      number[v] = n+1
   }
```

- This technique works because a DAG has no back arcs.

Consider what happens when the DFS leaves a vertex x for the last time. The only arcs emanating from v are tree, forward, and cross arcs. But all these arcs are directed towards vertices that have been completely visited by the search and therefore precede x in the order being constructed.

Figure 7.15: Strong components of a digraph

7.5.3 Strong Components

REF. Rao S. Kosaraju. Unpublished. 1978.

REF. Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, Volume 1, Number 2, pp. 146-160, 1972.

- A strongly connected component of a digraph is a maximal set of vertices in which there is a path from any one vertex to any other vertex in the set. For an example, see Figure 7.15.

- Let $G = (V, E)$ be a digraph. We can partition V into equivalence classes $V_i, 1 \leq i \leq r$, such that vertices v and w are equivalent if there is a path from v to w and a path from w to v.

  Let $E_i, 1 \leq i \leq r$, be the set of arcs with head and tail both in V_i.

  Then the graphs (V_i, E_i) are called the strong components of G.

  A digraph with only one strong component is said to be strongly connected.

- Depth-first-search can be used to efficiently determine the strong components of a digraph.

- Kosaraju's (1978) algorithm for finding strong components in a graph:

  1. Perform a DFS of G and number the vertices in order of completion of the recursive calls.

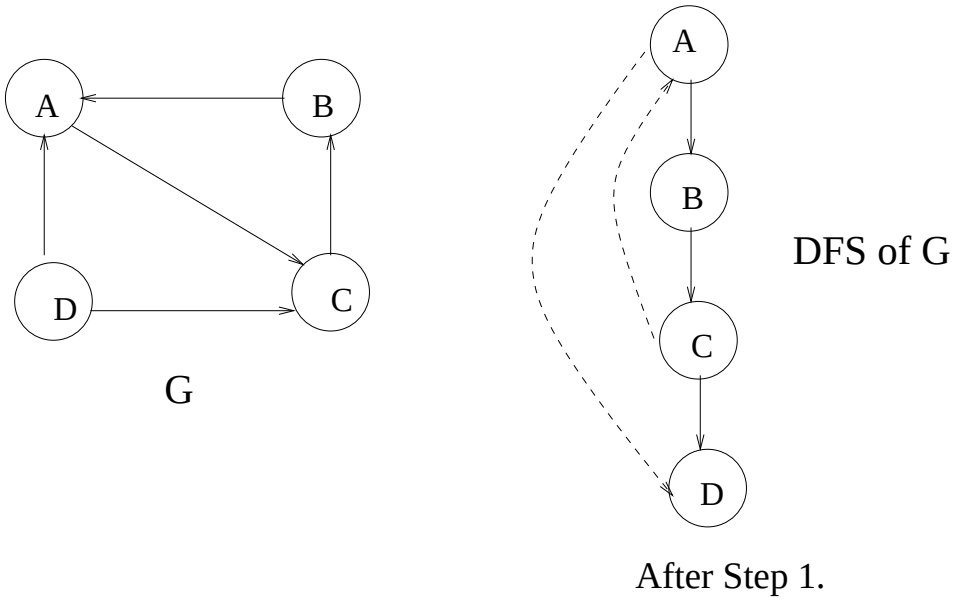


Figure 7.16: Step 1 in the strong components algorithm

2. Construct a new directed graph G_r by reversing the direction of every arc in G.
3. Perform a DFS on G_r starting the search from the highest numbered vertex according to the numbering assigned at step 1. If the DFS does not reach all vertices, start the next DFS from the highest numbered remaining vertex.
4. Each tree in the resulting spanning forest is a strong component of G.

Example: See Figures 7.16 and 7.17

Proof of Kosaraju's Algorithm

- First we show: If v and w are vertices in the same strong component, then they belong to the same spanning tree in the DFS of G_r.
  v and w in the same strong component

$$\begin{aligned} &\Longrightarrow \exists \text{ a path in } G \quad \text{from v to w and} \\ &\qquad\qquad\qquad\quad \text{from w to v} \\ &\Longrightarrow \exists \text{ a path in } G \quad \text{from w to v and} \\ &\qquad\qquad\qquad\quad \text{from v to w} \end{aligned}$$

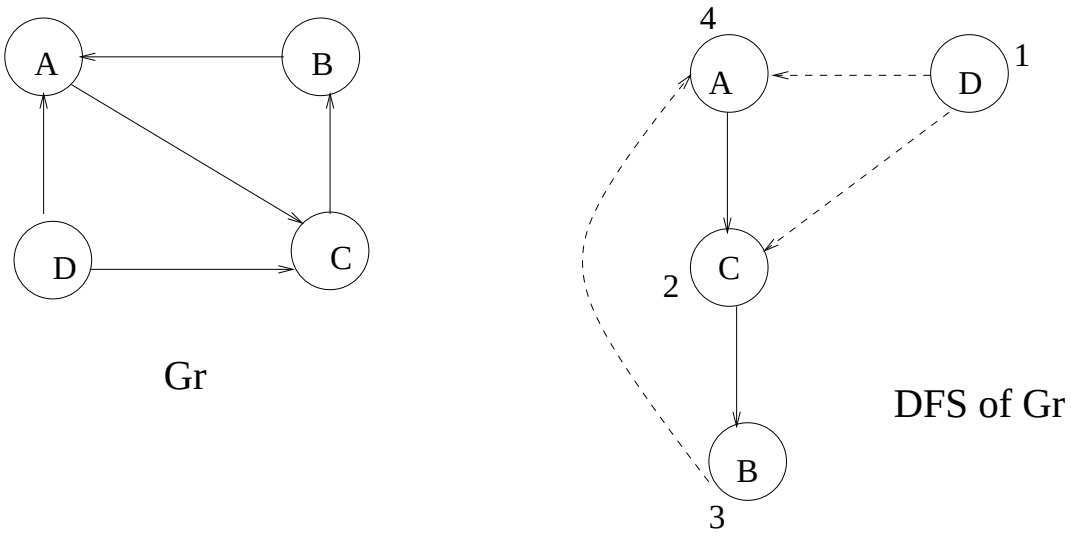


Figure 7.17: Step 3 in the strong components algorithm

Let us say in the DFS of G_r we start at some x and reach v or w. Since there is a path from v to w and vice versa, v and w will end up in the same spanning tree (having root x).

- Now we show: If v and w are in the same spanning tree in the DFS of G_r, then, they are in the same strong component of G.

  Let x be the root of the spanning tree containing v and w.

  $$\begin{aligned} &\Longrightarrow v \text{ is a descendant of } && \text{x} \\ &\Longrightarrow \exists \text{ a path in } G_r && \text{from x to v} \\ &\Longrightarrow \exists \text{ a path in } G && \text{from v to x} \end{aligned}$$

  In the DFS of G_r, vertex v was still unvisited when the DFS at x was initiated (since x is the root)

  $\Longrightarrow$ xx has a higher number than v
  (since DFS of G_r starts from highest numbered (remaining) vertex).
  $\Longrightarrow$ In the DFS of G, the recursive call at v terminated before the recursive call at x did. (due to the numbering done in step 1)

$$\begin{matrix} u \rightsquigarrow x \\ + \end{matrix} \tag{7.16}$$

Recursive call at v terminates before that of x in G (7.17)

In the DFS of G, there are two possibilities.

1) Search of v occurs before x (7.18)
2) Search of x occurs before v (7.19)

(1), (3) $\Longrightarrow$ DFS of v ... invokes DFS of x back to v.
$\Longrightarrow$ search at x would start and end before the search starts at v ends.
$\Longrightarrow$ recursive call at v terminates after the call at x contradicts (2).

$\Longrightarrow$ search of x occurs before v.
(4), (2) $\Longrightarrow$ v is visited during the search of x.
$\Longrightarrow$ v is descendant of x.
$\Longrightarrow$ $\exists$ a path from x to v.
$\Longrightarrow$ x and v are in the same strong component.

`Similarly`, x and w are in the same strong component.
$\Longrightarrow$ Similarly, v and w are in the same strong component.

7.6 To Probe Further

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

2. Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996. Indian Edition published by Prentice Hall of India, 1998.

3. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.

4. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.

5. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *The Design*

and Analysis of Computer Algorithms, Addison-Wesley, 1974.

6. Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Third Edition, 2000. Indian Edition published by Pearson Education Asia, 2000.

7. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

8. Gilles Brassard and Paul Bratley. *Algorithmics : Theory and Practice*. Prentice-Hall, 1988.

9. Ellis Horowitz, Sartaz Sahni, and Rajasekaran. *Fundamentals of Computer Algorithms*. W.H. Freeman and Company, 1998. Indian Edition published by Galgotia Publications, 2000.

10. Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*. Volume 2 of *Data Structures and Algorithms*, Springer-Verlag, 1984.

11. Robert Sedgewick. *Algorithms*. Addison-Wesley, Second Edition, 1988.

12. Nicklaus Wirth. *Data Structures + Algorithms = Programs*. Prentice-Hall, Englewood Cliffs. 1975.

13. Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*. Volume 16, Number 1, pp. 87–90, 1958.

14. E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, Volume 1, pp 269-271, 1959.

15. Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, Volume 17, pp 449-467, 1965.

16. Robert W Floyd. Algorithm 97 (Shortest Path). *Communications of the ACM*, Volume 5, Number 6, pp. 345, 1962.

17. John E Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, Volume 16, Number 6 pp.372-378, 1973.

18. Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, Volume 1, Number 2, pp.146-160, 1972.

19. Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, Volume 9, Number 1, pp.11-12, 1962.

7.7 Problems

1. Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why doesn't the proof of Dijkstra's algorithm go through when negative-weight edges are allowed?

2. Give an example of a four node directed graph with some negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Also, give an example of a four node directed graph with some negative-weight edges for which Dijkstra's algorithm always produces correct answers. In either case, justify your answer.

3. Explain how to modify Dijkstra's algorithm so that if there is more than one minimum path from source to a destination vertex, then a path with the fewest number of edges is chosen.

4. Suppose that in implementing Dijkstra's shortest path algorithm, one uses an AVL tree rather than a partially ordered tree for representing the *dynamic set* of non-special vertices. What will be the worst case complexity of the algorithm if an adjacency list representation is used for the digraph? Would you still prefer the partially ordered tree implementation?

5. We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two vertices.

6. Write a program to find the longest path in a directed acyclic graph. What is the complexity of the algorithm?

7. Describe a mathematical model for the following scheduling problem: Given tasks $T_1, T_2, \ldots, T_n$, which require time $t_1, t_2, \ldots, t_n$ to complete, and a set of constraints, each of the form T_j *must be completed prior to the start of* T_i, find the minimum time necessary to complete all the tasks assuming unlimited number of processors to be available.

8. In a depth-first search of a directed graph $G = (V, E)$, define $d(v)$ as the timestamp when v is visited for the first time and $f(v)$ the timestamp when the search finishes examining the adjacency list of v. Show that an edge $(u, v) \in E$ is

   (a) a tree edge or forward edge if and only if $d[u] < d[v] < f[v] < f[u]$.

   (b) a back edge if and only if $d[v] < d[u] < f[u] < f[v]$.

   (c) a cross edge if and only if $d[v] < f[v] < d[u] < f[u]$.

9. An Euler Tour of a connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

   (a) Show that G has an Euler tour if and only if $indegree(v) = outdegree(v)$ $\forall v \in V$.

   (b) Describe an $O(e)$ algorithm to find an Euler tour of G if one exists, where e is the number of edges.

10. Design an efficient algorithm to determine whether a given DAG is a tree.

11. Let G= (V, E) be a digraph. Define a relation R on V such that uRv if and only if u and v lie on a common (not necessarily simple) cycle. Prove that R is an equivalence relation on V.

7.8 Programming Assignments

7.8.1 Implementation of Dijkstra's Algorithm Using Binary Heaps and Binomial Queues

The objective of this assignment is to compare the performance of binary heaps and binomial queues in implementing the single source shortest path algorithm of **Dijkstra**, which computes the costs and paths of the shortest cost paths from a source vertex to every other vertex in a labeled digraph with non-negative weights. Note that the dynamic set $V - S$ is implemented using binary heaps or binomial queues.

Input graph

The graph that is input to the algorithm is either through a simple text file or is generated randomly. In the first case, assume the input to be in the following format:

- Line 1: Number of vertices in the graph
- Line 2: Source vertex (an integer between 1 and n, where n is the number of vertices
- Line 3: List of immediate neighbours of Node 1 with the weights on associated arcs
- Line 4: List of immediate neighbours of Vertex 2 with the weights on associated arcs

- *etc* $\cdots$

In the second case, to generate a random digraph, assume three inputs:

1. Number of vertices, n
2. Average degree of a node
3. Range of (integer) weights to be randomly generated for the directed arcs

Choose an appropriate data structure for the graph.

What is to be done ?

Implement Dijkstra's algorithm, using binary heaps and using binomial queues. Make sure to use the standard array data structure for binary heaps and an efficient, appropriate data structure for binomial queues (for example, look at the book by Alan Weiss). Attempt to do it in C++ but C is also good enough. As usual, special care should be taken to structure your program according to best practices in software engineering: use of good abstractions, smart algorithms, discipline in coding, documentation, provision for exceptions (for example, negative weights should be detected immediately; errors in input should be flagged asap; etc.).

For the input graph and the source vertex, print the following:

- shortest cost path to each vertex and the cost of this shortest path
- Execution times of the program with binary heaps and binomial queues
- Total number of heap operations executed with binary heaps and binomial queues (this includes: probes, swaps, link traversals, etc.)

7.8.2 Strong Components

Finding strong components of a given digraph is an important practical problem. You have studied Kosaraju's algorithm for this purpose. There are other algorithms available in the literature for determining the strong components. Many of the references lister earlier can be consulted for this. Implement Kosaraju's and at least one other algorithm for determining strong components and compare the performance of the two algorithms.

Chapter 8

Undirected Graphs

8.1 Some Definitions

- $G = (V, E)$

  Each **edge** is an unordered pair of vertices

- v and w are **adjacent** if (v, w) or equivalently (w, v) is an edge. The edge (v, w) is incident upon the vertices v and w.

- A **path** is a sequence of vertices $v_1, v_2, \ldots, v_n$, such that (v_i, v_{i+1}) is an edge for $1 \leq i < n$. A **path** is **simple** if all vertices on it are distinct, except possibly v_1 and v_n. The path has length $n - 1$ and connects v_1 and v_n.

- A graph is **connected** if every pair of vertices is connected.

- A **subgraph** $G' = (V', E')$ of a graph $G = (V, E)$ is a graph such that

  1. $V' \subseteq V$
  2. E' contains some edges (u, v) of E such that both u and v are in V'

  If E' contains **all** edges $(u, v) \in E$ for which $u, v \in V'$, the subgraph G' is called an **induced subgraph**.

- A **connected component** of a graph is an induced subgraph which is connected and which is not a proper subgraph of any other connected subgraph of G (i.e., maximal connected induced subgraph).

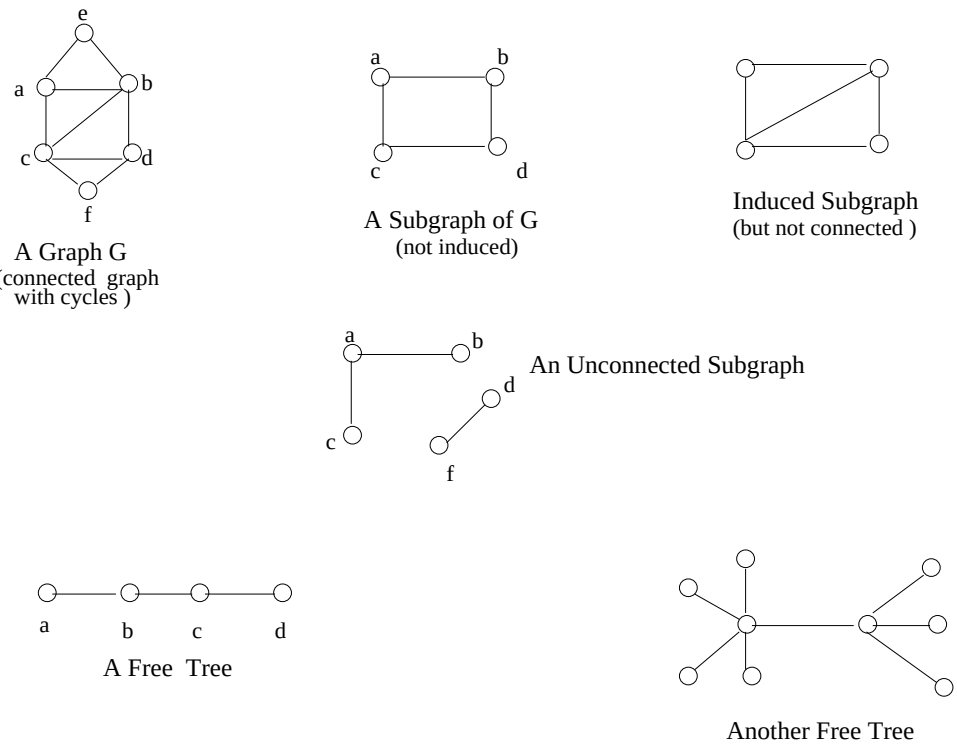


Figure 8.1: Examples of undirected graphs

- A **simple cycle** is a simple path of length ≥ 3, that connects a vertex to itself.
- A graph is **cyclic** if it contains at least one (simple) cycle.
- A **free tree** is a connected, acyclic graph.
- Observe that
  1. Every free tree with n vertices contains exactly $n - 1$ edges
  2. If we add any edge to a free tree, we get a cycle.
- See Figure 8.1 for several examples.

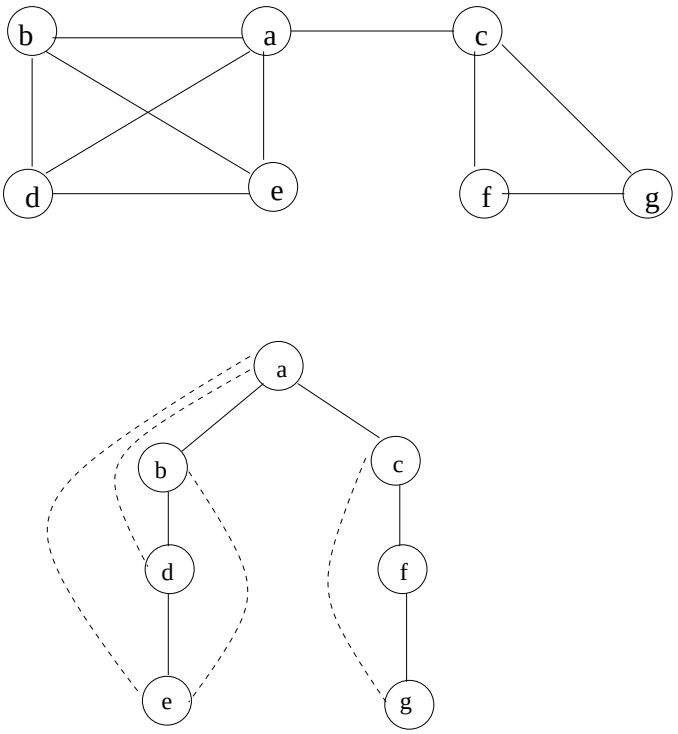


Figure 8.2: Depth-first search of an undirected graph

8.2 Depth First and Breadth First Search

See Figure 8.2 for an example. Assume the following adjacency lists.

Vertex	Adj. List
a	(b, c, d, e)
b	(a, d, e)
c	(a, f, g)
d	(a, b, e)
e	(a, b, d)
f	(c, g)
g	(c, f)

- DFS of an undirected graph involves only two types of arcs.
  1. Tree arcs.
  2. Back arcs (there is no distinction between back arcs and forward arcs)
- Cross arcs also don't exist because given any two vertices, if there exists an arc between them, then one of them will be an ancestor and

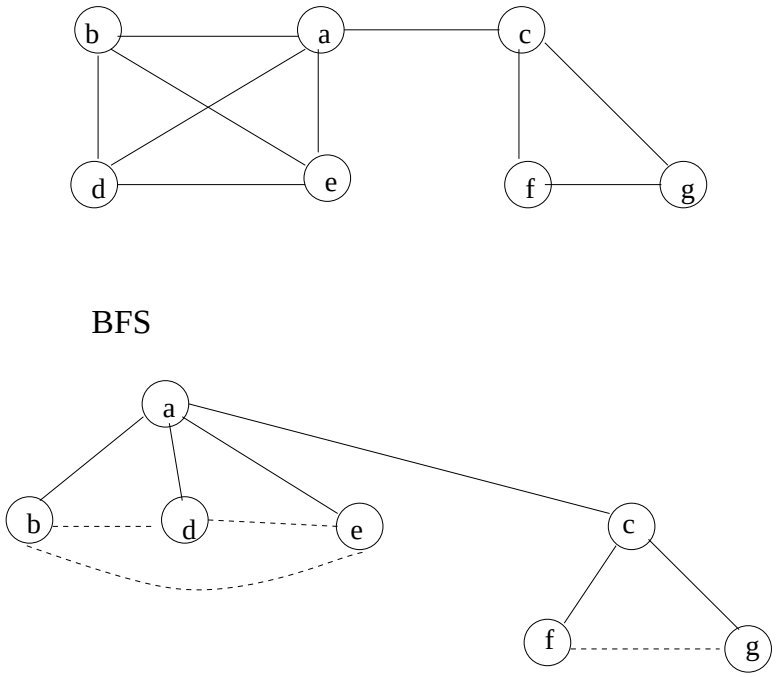


Figure 8.3: Breadth-first search of an undirected graph

the other a descendant in the DFS. Thus all arcs that would have been cross arcs in the DFS of a digraph would become tree arcs in the case of an undirected graph.

8.2.1 Breadth-first search of undirected graph

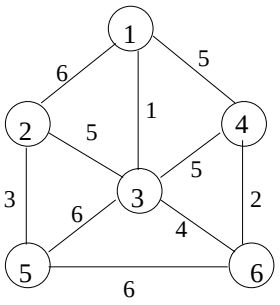
Example: See Figure 8.3.

Assume the adjacency lists:

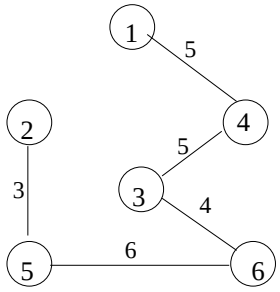
Vertex	Adj. List
a	(b, c, d, e)
b	(a, d, e)
c	(a, f, g)
d	(a, b, e)
e	(a, b, d)
f	(c, g)
g	(c, f)

8.3 Minimum-Cost Spanning Trees

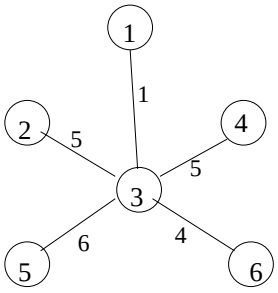
Let $G = (V, E)$ be a connected graph in which each edge $(u, v) \in E$ has an associated cost $C(u, v)$.



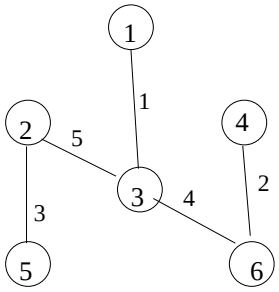
A connected graph.



A spanning tree with cost = 23



Another spanning tree with cost 21



MST, cost = 15

Figure 8.4: Spanning trees in a connected graph

- A **Spanning Tree** for G is a subgraph of G that it is a free tree connecting all vertices in V. The cost of a spanning tree is the sum of costs on its edges.
- An **MST** of G is a spanning tree of G having a minimum cost.
- See Figure 8.4 for several examples.

8.3.1 MST Property

Suppose $G = (V, E)$ is a connected graph with costs defined on all $e \in E$. Let U be some proper subset of V.

If (u, v) is an edge of lowest cost such that $u \in U$ and $v \in V - U$, then there exists an MST that includes (u, v) as an edge. See Figure 8.5.

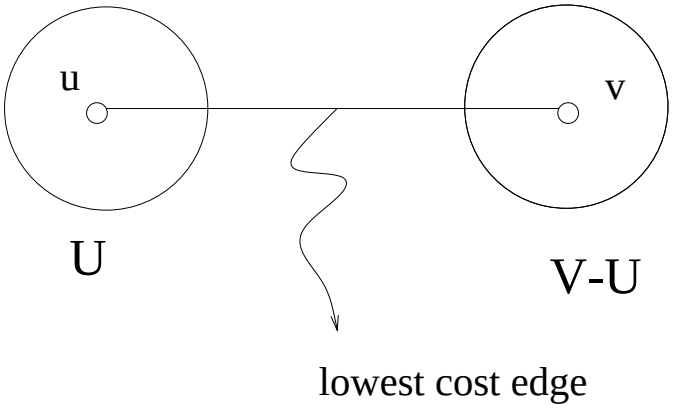


Figure 8.5: An illustration of MST property

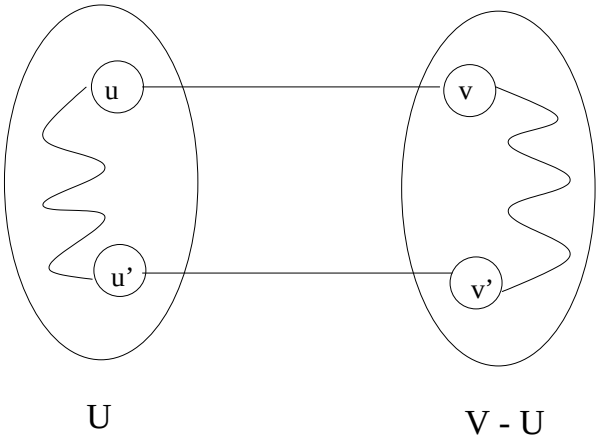


Figure 8.6: Construction of a minimal spanning tree

Proof of MST Property

Suppose to the contrary that there is no MST for the graph G which includes the edge (u, v).

Let T be any MST of G. By our assumption, T does not contain (u, v). Adding (u, v) to T will introduce a cycle since T is a free tree. This cycle involves (u, v). Therefore there is a path from v to u that does not pass through this edge. This means that $\exists$ another edge (u', v') in T such that $u' \in U$ and $v' \in V - U$. See Figure 8.6.

Deleting edge (u', v') breaks the cycle and yields a spanning tree T' whose cost is certainly $\leq$ that of T since $C(u, v) \leq C(u', v')$. Thus we have constructed an MST that includes (u, v).

To illustrate, consider the graph in Figure 8.7 and refer to Figures 8.8

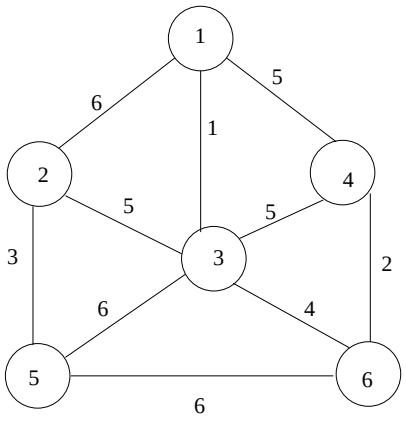


Figure 8.7: An example graph for finding an MST

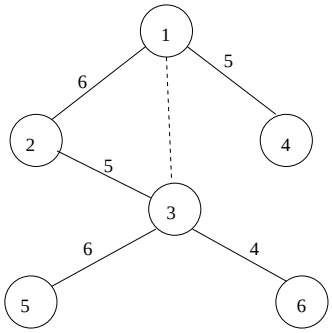


Figure 8.8: A spanning tree in the above graph, with cost 26

and 8.9. Consider the sets:

$$\begin{aligned} U &= \{1, 2, 5\} \\ V - U &= \{3, 4, 6\}. \end{aligned}$$

Spanning tree with cost = 26. Now, least cost edge from U to $V - U$ is (1,3).
By including (1,3) in the above spanning tree, a cycle will form (for example, 1-2-3-1). Let us replace the edge (2,3) by the edge (1,3).

This has yielded a ST with cost = 22

DEF. A set of edges T in a connected graph promising if it can be extended to produce a minimal spanning tree for the graph.

- By definition, $T = \phi$ is always promising since a weighted connected graph always has at least one MST.

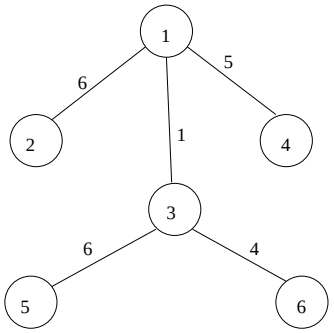


Figure 8.9: Another spanning tree, but with cost 22

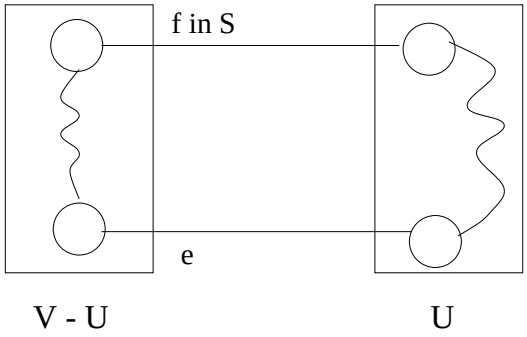


Figure 8.10: Illustration of MST Lemma

- Also, if a promising set of edges T is a spanning tree, then it must be an MST.

Def: An edge is said to **leave** a given set of nodes if exactly one end of this edge is in the set.

MST Lemma: Let

- $G = (V, E)$ be weighted connected graph
- $U \subset V$ a strict subset of nodes in G
- $T \subseteq E$ a promising set of edges in E such that no edge in T leaves U
- e a least cost edge that leaves U

Then the set of edges $T' = T \cup \{e\}$ is promising.

Proof

Since T is promising, it can be extended to obtain an MST, says S. If $e \in S$, there is nothing to prove.

If $e \in S$, then we add edge e to S, we create exactly one cycle (since S is a spanning tree). In this cycle, since e leaves U there exists at least one other edge, f say, that also leaves U (otherwise the cycle could not close). See Figure 8.10.

If we now remove f, the cycle disappears and we obtain a new tree R that spans G.

Note that $R = (S \cup \{e\}) - \{f\}$

Also note that weight of $e \leq$ weight of f since e is a least cost edge leaving U.

Therefore R is also an MST and it includes edge e. Furthermore $T \subseteq R$ and so can be extended to the MST R. Thus T is a promising set of edges.

8.3.2 Prim's Algorithm

This algorithm is directly based on the MST property. Assume that $V = \{1, 2, \ldots, n\}$.

REF. R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, Volume 36, pp. 1389-1401, 1957.

```
{
        T = φ;
        U = { 1 };
        while (U ≠ V)
          {
            let (u, v) be the lowest cost edge
            such that u ∈ U and v ∈ V − U;
```

$T = T \cup \{(u, v)\}$
$U = U \cup \{v\}$
}
}

- See Figure 8.11 for an example.
- $O(n^2)$ algorithm.

Proof of Correctness of Prim's Algorithm

Theorem: Prim's algorithm finds a minimum spanning tree.

Proof: Let G = (V,E) be a weighted, connected graph. Let T be the edge set that is grown in Prim's algorithm. The proof is by mathematical induction on the number of edges in T and using the MST Lemma.

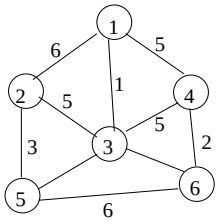
Basis: The empty set ϕ is promising since a connected, weighted graph always has at least one MST.

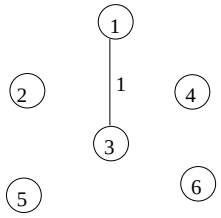
Induction Step: Assume that T is promising just before the algorithm adds a new edge e = (u,v). Let U be the set of nodes grown in Prim's algorithm. Then all three conditions in the MST Lemma are satisfied and therefore T U e is also promising.
When the algorithm stops, U includes all vertices of the graph and hence T is a spanning tree. Since T is also promising, it will be a MST.

Implementation of Prim's Algorithm

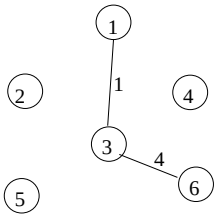
Use two arrays, **closest** and **lowcost**.

- For $i \in V - U$, closest[i] gives the vertex in U that is closest to i
- For $i \in V - U$, lowcost[i] gives the cost of the edge (i, closest(i))

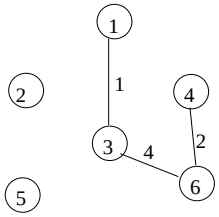




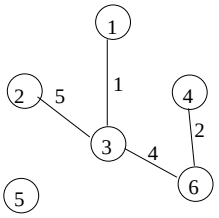
Iteration 1. U = {1}



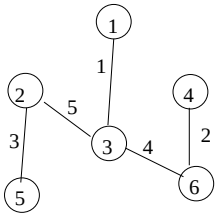
Iteration 2. U = {1,3}



Iteration 3. U = {1,3,6}



Iteration 4. U = {1,3,6,4}



Iteration 5. U = {1,3,6,4,2}

Figure 8.11: Illustration of Prim's algorithm

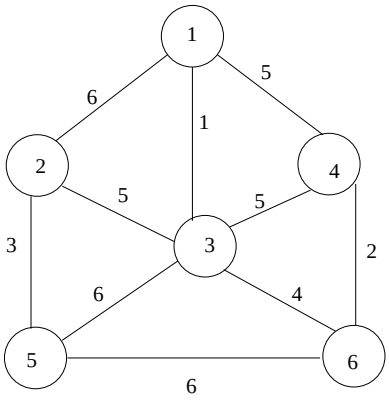


Figure 8.12: An example graph for illustrating Prim's algorithm

- At each step, we can scan lowcost to find the vertex in $V - U$ that is closest to U. Then we update lowcost and closest taking into account the new addition to U.
- Complexity: $O(n^2)$

Example: Consider the digraph shown in Figure 8.12.

Step 1

$U = \{1\}$ $V - U = \{2, 3, 4, 5, 6\}$

closest		lowcost
$V - U$	U	
2	1	6
3	1	1
4	1	5
5	1	∞
6	1	∞

Select vertex 3 to include in U

Step 2

$U = \{1, 3\}$ $V - U = \{2, 4, 5, 6\}$

closest		lowcost
$V - U$	U	
2	3	5
4	1	5
5	3	6
6	3	4

Now select vertex 6

Step 3

$U = \{1, 3, 6\}$ $V - U = \{2, 4, 5, 6\}$

closest		lowcost
$V - U$	U	
2	3	5
4	6	2
5	3	6

Now select vertex 4, and so on

8.3.3 Kruskal's Algorithm

REF. J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, Volume 7, pp. 48-50, 1956.

- Complexity is $O(e \log e)$ where e is the number of edges. Can be made even more efficient by a proper choice of data structures.
- Greedy algorithm
- Algorithm:

Let $G = (V, E)$ be the given graph, with $|V| = n$
{
Start with a graph $T = (V, \phi)$ consisting of only the
vertices of G and no edges; /* This can be viewed as n
connected components, each vertex being one connected component */
Arrange E in the order of increasing costs;
for $(i = 1, i \leq n - 1, i + +)$
{ Select the next smallest cost edge;
if (the edge connects two different connected components)
add the edge to T;
}
}

- At the end of the algorithm, we will be left with a single component that comprises all the vertices and this component will be an MST for G.

Proof of Correctness of Kruskal's Algorithm

Theorem: Kruskal's algorithm finds a minimum spanning tree.

Proof: Let G = (V, E) be a weighted, connected graph. Let T be the edge set that is grown in Kruskal's algorithm. The proof is by mathematical induction on the number of edges in T.

- We show that if T is promising at any stage of the algorithm, then it is still promising when a new edge is added to it in Kruskal's algorithm
- When the algorithm terminates, it will happen that T gives a solution to the problem and hence an MST.

Basis: $T = \phi$ is promising since a weighted connected graph always has at least one MST.

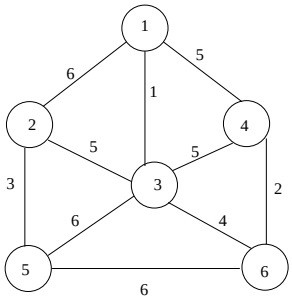
Induction Step: Let T be promising just before adding a new edge $e = (u, v)$. The edges T divide the nodes of G into one or more connected components. u and v will be in two different components. Let U be the set of nodes in the component that includes u. Note that

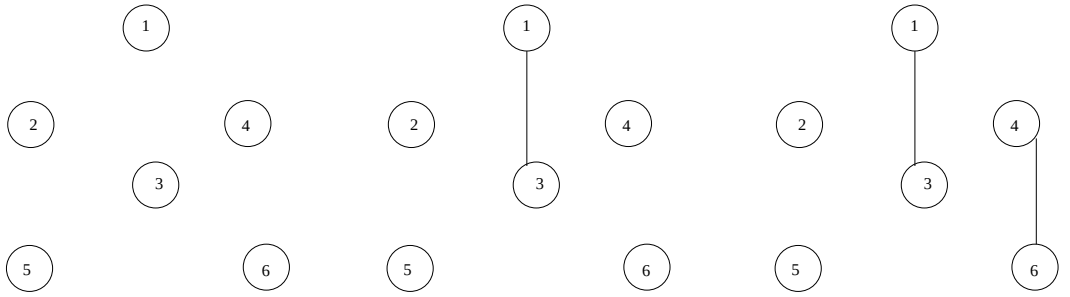
- U is a strict subset of V
- T is a promising set of edges such that no edge in T leaves U (since an edge T either has both ends in U or has neither end in U)
- e is a least cost edge that leaves U (since Kruskal's algorithm, being greedy, would have chosen e only after examining edges shorter than e)

The above three conditions are precisely like in the MST Lemma and hence we can conclude that the $T \cup \{e\}$ is also promising. When the algorithm stops, T gives not merely a spanning tree but a minimal spanning tree since it is promising.

- **Program**

```
void kruskal (vertex-set V; edge-set E; edge-set T)
  int ncomp;  /* current number of components */
  priority-queue edges  /* partially ordered tree */
  mfset components;  /* merge-find set data structure */
  vertex u, v; edge e;
  int nextcomp;  /* name for new component */
  int ucomp, vcomp;  /* component names */
   {
    makenull (T); makenull (edges);
    nextcomp = 0; ncomp = n;
```





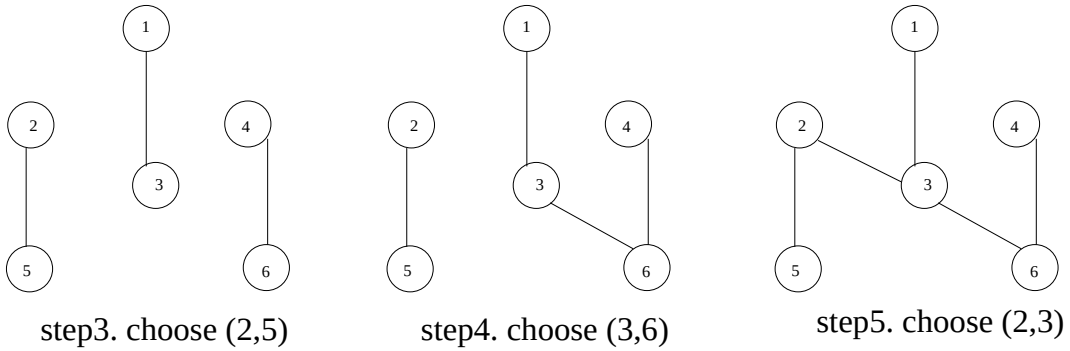


Figure 8.13: An illustration of Kruskal's algorithm

```
    for (v ∈ V) /* initialize a component to have one vertex of V */
      { nextcomp++ ;
        initial (nextcomp, v, components);
      }
    for (e ∈ E)
      insert (e, edges);  /* initialize priority queue of edges */
      while (ncomp > 1)
      {
        e = deletemin (edges);
        let e = (u, v);
        ucomp = find(u, components);
        vcomp = find(v, components);
        if (ucomp! = vcomp)
          {
          merge (ucomp, vcomp, components);
           ncomp = ncomp − 1;
      }
    }
}
```

Implementation

- Choose a partially ordered tree for representing the sorted set of edges
- To represent connected components and interconnecting them, we need to implement:
  1. MERGE (A, B, C) . . . merge components A and B in C and call the result A or B arbitrarily.
  2. FIND (v, C) . . . returns the name of the component of C of which vertex v is a member. This operation will be used to determine whether the two vertices of an edge are in the same or in different components.

3. INITIAL (A, v, C) . . . makes A the name of the component in C containing only one vertex, namely v

- The above data structure is called an MFSET

Running Time of Kruskal's Algorithm

- Creation of the priority queue
  - * If there are e edges, it is easy to see that it takes $O(e \log e)$ time to insert the edges into a partially ordered tree
  - * $O(e)$ algorithms are possible for this problem
- Each deletemin operation takes $O(\log e)$ time in the worst case. Thus finding and deleting least-cost edges, over the while iterations contribute $O(\log e)$ in the worst case.
- The total time for performing all the merge and find depends on the method used.

$O(e \log e)$	without path compression
$O(e\alpha(e))$	with the path compression, where
$\alpha(e)$	is the inverse of an Ackerman function.

Example: See Figure 8.13.
E = {(1,3), (4,6), (2,5), (3,6), (3,4), (1,4), (2,3), (1,2), (3,5), (5,6) }

8.4 Traveling Salesman Problem

REF. Eugene L Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B.Shmoys, editors. *The Traveling Salesman Problem.* John Wiley & Sons, 1985.

- Tour (Hamilton) (Hamiltonian cycle)

  Given a graph with weights on the edges a **tour** is a simple cycle that includes all the vertices of the graph. For examples of tours, see Figure 6.30.

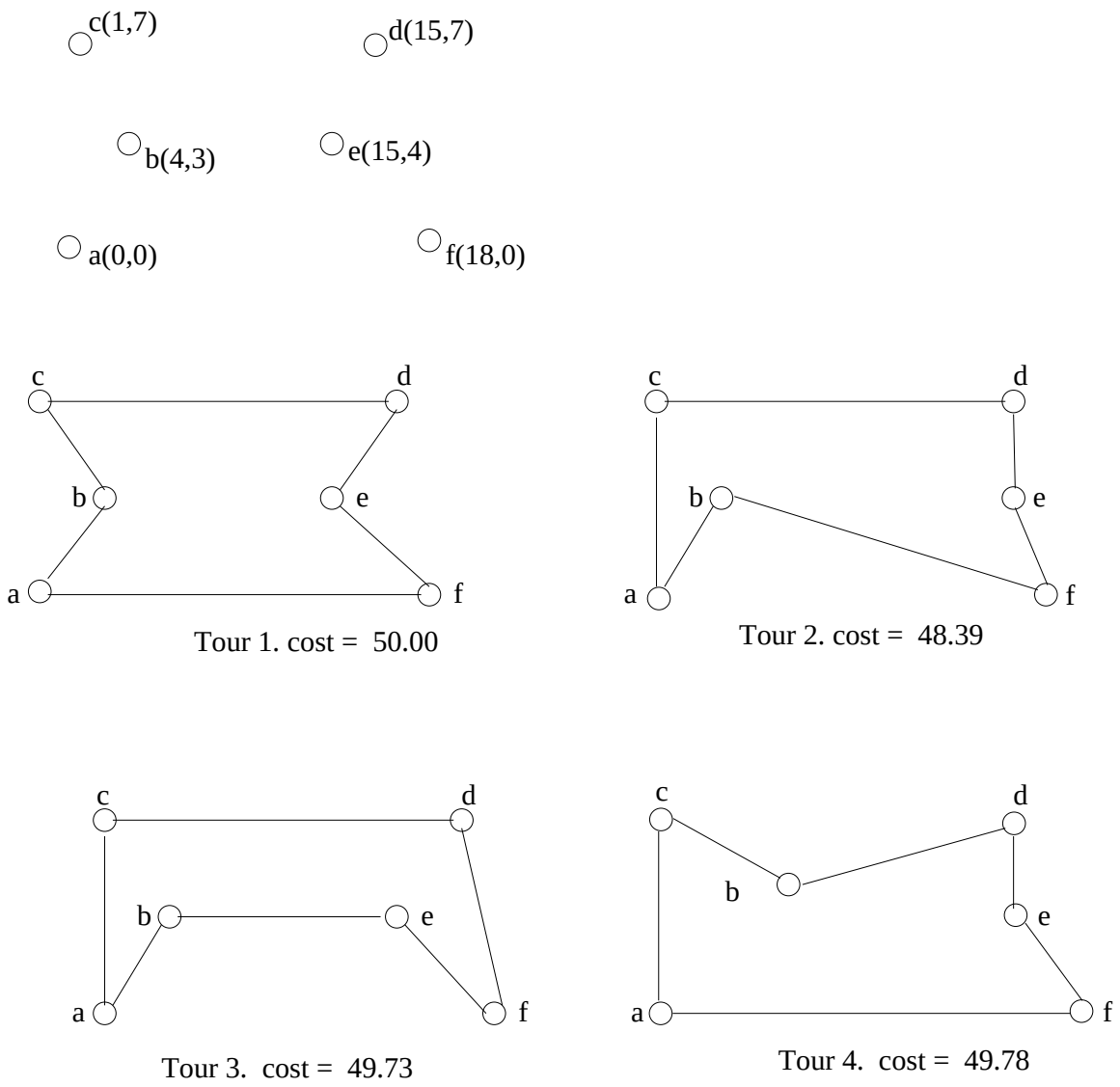


Figure 8.14: A six-city TSP and some tours

- TSP

  Given a graph with weights on the edges, find a tour having a minimum sum of edge weights.

- NP-hard problem

8.4.1 A Greedy Algorithm for TSP

- Based on Kruskal's algorithm. It only gives a suboptimal solution in general.

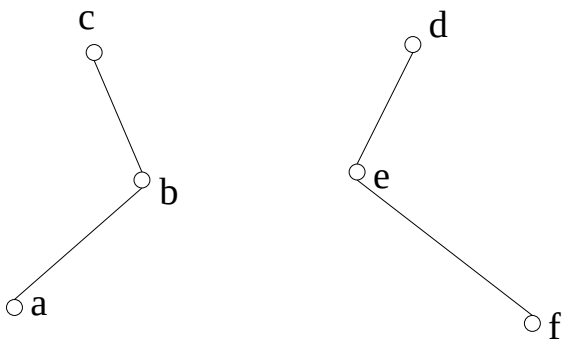


Figure 8.15: An intermediate stage in the construction of a TSP tour

- Works for complete graphs. May not work for a graph that is not complete.
- As in Kruskal's algorithm, first sort the edges in the increasing order of weights.
- Starting with the least cost edge, look at the edges one by one and select an edge only if the edge, together with already selected edges,
  1. does not cause a vertex to have degree three or more
  2. does not form a cycle, unless the number of selected edges equals the number of vertices in the graph.

Example:

Consider the six city problem shown in Figure 8.14. The sorted set of edges is

$\{((d,e),3),((b,c),5),((a,b),5),((e,f),5),((a,c),7.08),((d,f),\sqrt{58}),$
$((b,e),\sqrt{22}),((b,d),\sqrt{137}),((c,d),14),\ldots((a,f),18)$

See Figures 8.15 and 8.16.

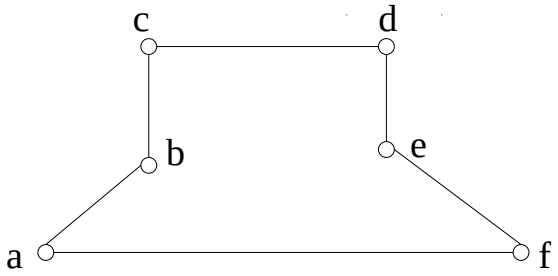


Figure 8.16: A TSP tour for the six-city problem

Select (d, e)	
Select (a, b)	
Select (b, c)	
Select (e, f)	
Reject (a, c)	since it forms a cycle with (a, b) and (b, c)
Reject (d, f)	since it forms a cycle with (d, e) and (e, f)
Reject (b, e)	since that would make the degree of b equal to 3
Reject (b, d)	for an identical reason
Select (c, d)	
.	
.	
.	
Select (a, f)	

$\Downarrow$

This yields a total cost = 50, which is about 4% from the optimal cost.

8.4.2 Optimal Solution for TSP using Branch and Bound

Principle

Suppose it is required to minimize an objective function. Suppose that we have a method for getting a lower bound on the cost of any solution among those in the set of solutions represented by some subset. If the best solution found so far costs less than the lower bound for this subset, we need not explore this subset at all.

Let S be some subset of solutions. Let

$$\begin{aligned} L(S) &= \text{a lower bound on the cost of any solution belonging to } S \\ \text{Let } C &= \text{cost of the best solution} \end{aligned}$$

found so far

If $C \leq L(S)$, there is no need to explore S because it does not contain any better solution.

If $C > L(S)$, then we need to explore S because it may contain a better solution.

A Lower Bound for a TSP

Note that:

Cost of any tour

$$= \frac{1}{2} \sum_{v \in V} \begin{array}{l}\text{(Sum of the costs of the two tour} \\ \text{edges adjacent to } v)\end{array}$$

Now:

The sum of the two tour edges adjacent to a given vertex v

$$\geq \text{ sum of the two edges of least cost adjacent to } v$$

Therefore:

Cost of any tour

$$\geq \frac{1}{2} \sum_{v \in V} \begin{array}{l}\text{(Sum of the costs of the two least cost} \\ \text{edges adjacent to } v)\end{array}$$

Example: See Figure 8.17.

Node	**Least cost edges**	**Total cost**
a	(a, d), (a, b)	5
b	(a, b), (b, e)	6
c	(c, b), (c, a)	8
d	(d, a), (d, c)	7
e	(e, b), (e, f)	9

Thus a lower bound on the cost of any tour

$$= \frac{1}{2}(5 + 6 + 8 + 7 + 9) = 17.5$$

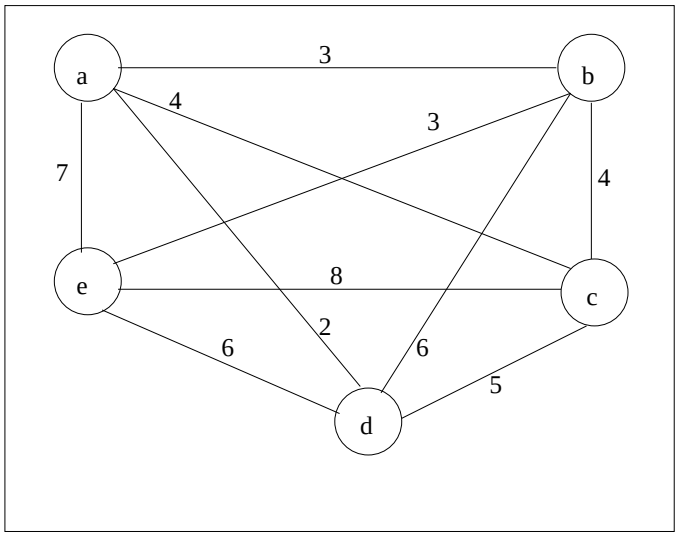


Figure 8.17: Example of a complete graph with five vertices

A solution Tree for a TSP instance: (edges are considered in lexicographic order): See Figure 8.18

- Suppose we want a lower bound on the cost of a subset of tours defined by some node in the search tree.

  In the above solution tree, each node represents tours defined by a set of edges that must be in the tour and a set of edges that may not be in the tour.

- These constraints alter our choices for the two lowest cost edges at each node.

  e.g., if we are constrained to include edge (a, e), and exclude (b, c), then we will have to select the two lowest cost edges as follows:

a	(a, d), (a, e)	9
b	(a, b), (b, e)	6
c	(a, c), (c, d)	9
d	(a, d), (c, d)	7
e	(a, e), (b, e)	10

  Therefore lower bound with the above constraints = 20.5

- Each time we **branch**, by considering the two children of a node, we try to infer additional decisions regarding which edges must be included or excluded from tours represented by those nodes. The rules we use for these inferences are:

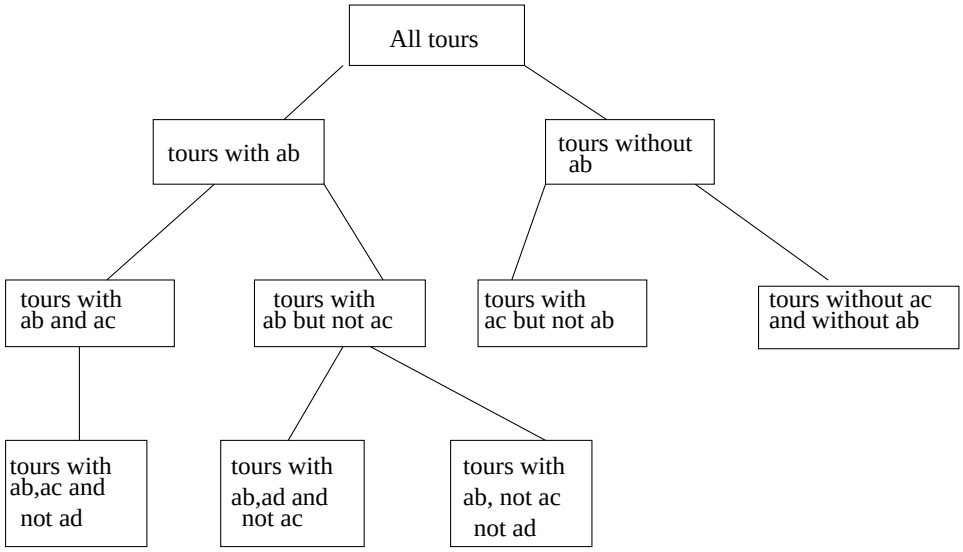


Figure 8.18: A solution tree for a TSP instance

1. If excluding (x, y) would make it impossible for x or y to have as many as two adjacent edges in the tour, then (x, y) must be included.
2. If including (x, y) would cause x or y to have more than two edges adjacent in the tour, or would complete a non-tour cycle with edges already included, then (x, y) must be excluded.

- See Figure 8.19.
- When we branch, after making what inferences we can, we compute lower bounds for both children. If the lower bound for a child is as high or higher than the lowest cost found so far, we can "prune" that child and need not consider or construct its descendants.

  Interestingly, there are situations where the lower bound for a node n is lower than the best solution so far, yet both children of n can be pruned because their lower bounds exceed the cost of the best solution so far.
- If neither child can be pruned, we shall, as a heuristic, consider first the child with the smaller lower bound. After considering one child, we must consider again whether its sibling can be pruned, since a new best solution may have been found.

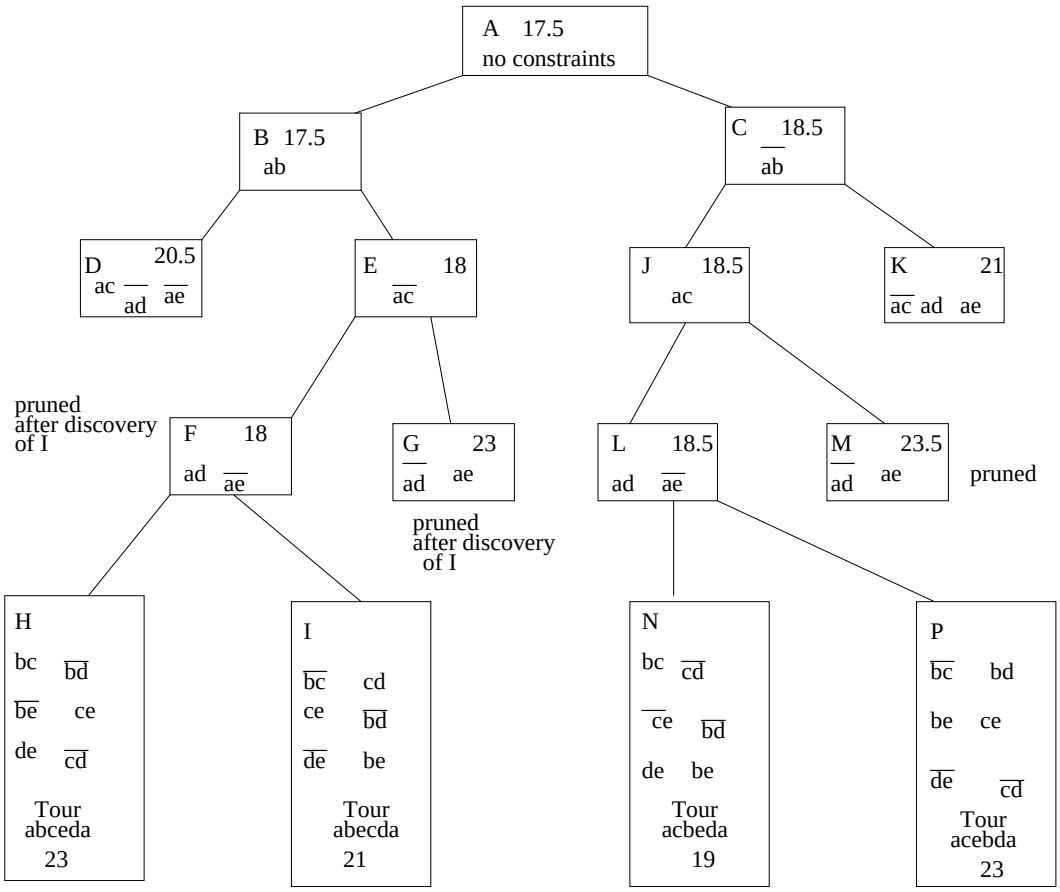


Figure 8.19: Branch and bound applied to a TSP instance

8.5 To Probe Further

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

2. Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996. Indian Edition published by Prentice Hall of India, 1998.

3. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.

4. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.

5. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

6. Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Third Edition, 2000. Indian Edition published by Pearson Education Asia, 2000.

7. Ellis Horowitz, Sartaz Sahni, and Rajasekaran. *Fundamentals of Computer Algorithms*. W.H. Freeman and Company, 1998. Indian Edition published by Galgotia Publications, 2000.

8. Eugene L Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B.Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

9. Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*. Volume 2 of *Data Structures and Algorithms*, Springer-Verlag, 1984.

10. Robert Sedgewick. *Algorithms*. Addison-Wesley, Second Edition, 1988.

11. Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, Volume 17, pp 449-467, 1965.

12. John E Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, Volume 16, Number 6 pp.372-378, 1973.

13. J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, Volume 7, pp 48-50, 1956.

14. R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, Volume 36, pp.1389-1401, 1957.

15. Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, Volume 1, Number 2, pp.146-160, 1972.

8.6 Problems

1. Consider an undirected graph G=(V, E), with n vertices. Show how a one dimensional array of length $\frac{n(n-1)}{2}$ can be used to represent G.

2. Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. The algorithm should run in O(|V|) time independent of |E|.

3. Design an algorithm to enumerate all simple cycles of a graph. How many such cycles can there be? What is the time complexity of the algorithm?

4. Let (u, v) be a minimum-weight edge in a graph G. Show that (u, v) belongs to some minimum spanning tree of G.

5. Let e be maximum-weight edge on some cycle of $G = (V, E)$. Prove that there is a minimum spanning tree of $G' = (V, E - \{e\})$ that is also a minimum spanning tree of G.

6. For an undirected graph G with n vertices and e edges, show that $\sum_{i=1}^{n} d_i = 2e$ where d_i is the degree of vertex i (number of edges incident on vertex i).

7. The diameter of a tree $T = (V, E)$ is defined by $\max_{u,v \in V} \delta(u, v)$ where $\delta(u, v)$ is the shortest-path distance from vertex u to vertex v. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of the algorithm.

8. An *Eulerian walk* in an undirected graph is a path that starts and ends at the same vertex, traversing every edge in the graph exactly once. Prove that in order for such a path to exist, all nodes must have even degree.

9. Two binary trees T_1 and T_2 are said to be *isomorphic* if T_1 can be transformed into T_2 by swapping left and right children of (some of the) nodes in T_1. Design an efficient algorithm to decide id two given trees are isomorphic. What is the time complexity of your algorithm?

8.7 Programming Assignments

8.7.1 Implementation of Some Graph Algorithms

This assignment involves five problems.

1. Generate a *connected* graph with n vertices and a certain number of edges, decided by an average degree d of each vertex. The connected graph generated should have several *spanning trees* and several *Hamiltonian paths*. Also, generate random costs for all the edges.

2. Find a minimum cost spanning tree (MST) for the above graph, using an efficient implementation of Prim's algorithm. Compute the execution time of this program.

3. Find an MST using Kruskal's algorithm. In the implementation, use the *heap* data structure to organize the set of edges and the *MFSET* data structure to implement union's and find's. Compare the execution time with that of Prim's algorithm.

4. Find a *second best MST* by extending the Kruskal's algorithm, using the following property: If T is an MST, then there exist an edge $(u, v) \in T$ and an edge $(x, y) \notin T$ such that $(T - \{(u, v)\})U\{(x, y)\}$ is a second best MST.

5. Find a *Hamiltonian path* using a greedy strategy based on Kruskal's algorithm. You can compare the cost of this path to that of a minimal cost Hamiltonian path (found by exhaustive enumeration) for values of n up to 10. In fact, you can do a little better by using the *branch and bound* method to compute the minimal cost Hamiltonian path.

6. Provide examples of connected graphs for situations specified below. If an example cannot exist for the situation, provide reasons.

   (a) A graph in which a maximum cost edge is a part of every MST in the graph

   (b) A graph in which a maximum cost edge is never a part of any MST

   (c) A graph in which a least cost edge is not a part of any MST

7. Let T be a minimum spanning tree of a connected graph G. Prove that there exist edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

8.7.2 Traveling Salesman Problem

This assignment involves implementing the following five problems.

1. Generate a *complete* graph with n vertices and random costs for all the edges. Let the costs be positive integers uniformly distributed between 1 and 100.
2. Find a minimum cost spanning tree (MST) for the above graph, using an efficient implementation of Prim's algorithm. Compute the running time of this program.
3. Find an MST using Kruskal's algorithm. In the implementation, use the *partially ordered tree* data structure to organize the set of edges and the *MFSET* data structure to implement union's and find's. Compare the running time with that of Prim's algorithm.
4. Find a *Hamiltonian path* using a greedy strategy based on Kruskal's algorithm.
5. Find an optimal *Hamiltonian path* for the above graph using the *branch and bound* methodology.

It should be possible to input any desired graph and carry out steps 2, 3, 4, and 5 above. Assume the following input format: Let the set of vertices be $V = \{1, \ldots, n\}$ Then the input file would be: n, Cost of edge $(1, 2)$, ..., Cost of edge $(1, n)$, Cost of edge $(2, 3)$, ..., Cost of edge $(2, n)$, Cost of edge $(3, 4)$, ..., Cost of edge $(n - 1, n)$.

Chapter 9

Sorting Methods

The function of sorting or ordering a list of objects according to some linear order is so fundamental that it is ubiquitous in engineering applications in all disciplines. There are two broad categories of sorting methods: **Internal** sorting takes place in the main memory, where we can take advantage of the random access nature of the main memory; **External** sorting is necessary when the number and size of objects are prohibitive to be accommodated in the main memory.

The Problem:

- Given records $r_1, r_2, \ldots, r_n$, with key values $k_1, k_2, \ldots, k_n$, produce the records in the order

  $$r_{i_1}, r_{i_2}, \ldots, r_{i_n},$$

  such that

  $$k_{i_1} \leq k_{i_2} \leq \ldots \leq k_{i_n}$$

- The complexity of a sorting algorithm can be measured in terms of
  - number of algorithm steps to sort n records
  - number of comparisons between keys (appropriate when the keys are long character strings)
  - number of times records must be moved (appropriate when record size is large)
- Any sorting algorithm that uses comparisons of keys needs at least O(n log n) time to accomplish the sorting.

Sorting Methods

Internal (In memory)	**External Appropriate for secondary storage**
quick sort	
heap sort	mergesort
bubble sort	radix sort
insertion sort	polyphase sort
selection sort	
shell sort	

9.1 Bubble Sort

Let $a[1], a[2], \ldots, a[n]$ be n records to be sorted on a key field "key". In bubble sort, records with low keys (light records) bubble up to the top.

```
{
    for (i = 1; i <= n - 1; i++)
      {
        for (j = n; j >= i + 1; j--)
            if (a[j].key < a[j - 1].key)
              swap the records a[j] and a[j - 1]
      }
}
```

At the end of the $(i-1)^{st}$ iteration, we have

$$\underbrace{a[1] \ \ldots \ a[i-1]}_{\text{sorted as in the final order}} \quad \underbrace{a[i] \ \ldots \ a[n]}_{\text{unsorted}}$$

In the i^{th} iteration, the i^{th} lowest key bubbles up to the i^{th} position and

we get

$$\underbrace{a[1] \ \ldots \ a[i-1]a[i]}_{\text{sorted}} \quad \underbrace{a[i+1] \ \ldots \ a[n]}_{\text{unsorted}}$$

Worst case complexity and also average case complexity is $O(n^2)$.

9.2 Insertion Sort

Here in the i^{th} iteration, the record $a[i]$ is inserted into its proper position in the first i positions.

$\underbrace{a[1] \ \ldots \ a[i-1]}$	$\underbrace{a[i] \ \ldots \ a[n]}$
sorted (may not be as in the final order)	Insert $a[i]$ into an appropriate place

Let us assume, for the sake of convenience, a fictitious record $a[0]$ with key value $= -\infty$.

```
{
    for (i = 2; i <= n; i++)
      {
        j = i ;
        while (key of a[j] < key of a[j-1])
          {
            swap records a[j] and a[j-1];
            j = j - 1
          }
      }
  }
```

- Exhibits the worst case performance when the initial array is sorted

in reverse order.

- Worst case and average case performance is $O(n^2)$

9.3 Selection Sort

i^{th} Iteration

$$\underbrace{A[1]A[2]\ \cdots\ A[i-1]}_{\text{sorted as in the final sequence}} \qquad \underbrace{A[i]\ \cdots\ A[n]}_{\text{select least among these and swap with } A[i]}$$

$$\Downarrow$$

$$\underbrace{A[1]A[2]\ \cdots\ A[i]}_{\text{sorted as in final sequence}} \quad \underbrace{A[i+1]\ \cdots\ A[n]}_{\text{yet to be sorted}}$$

```
{
  for (i = 1, i ≤ n − 1; i++)
  {
    lowindex = i; lowkey = A[i] → key;
      for (j = i + 1; j ≤ n; j++)
        if (A[j] → key < lowkey)
          {
            lowkey = A[j] → key;
            lowindex = j
          }
      swap (A[i], A[lowindex])
  }
}
```

- Number of swaps $= n - 1$
- Number of comparisons $= \frac{n(n-1)}{2}$

9.4 Shellsort

- Invented by David Shell. Also called as Diminishing Increments sort.

REF. Shell. A High-speed Sorting Procedure.
Communications of the ACM, Volume 2, Number 7, pp. 30-32,1959.

- The algorithm works by comparing elements that are distant; the distance between comparisons decreases as the algorithm runs until the last phase, in which adjacent elements are compared.
- Uses an increment sequence, $h_1, h_2, ..., h_t$. Any increment sequence will do as long as $h_1 = 1$, however some choices are better than others.
- After a phase, using some increment h_k, for every i, we have

$$a[i] \leq a[i + h_k] \quad whenever \quad i + h_k \leq n$$

That is, all elements spaced h apart are sorted and the sequence is said to be
$h_k - sorted$

- The following shows an array after some phases in shellsort.

OriginalArray	81	94	11	96	12	35	17	95	28	58	41	75	15
After5 – Sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After3 – Sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After1 – Sort	11	12	15	17	28	35	41	58	75	81	94	95	96

- An important property of Shellsort:
  - A sequence which is h_k - sorted that is then h_{k-1} - sorted will remain h_k - sorted.

This means that work done by early phases is not undone by later phases.

- The action of an h_k - sorted is to perform an insertion sort on h_k independent subarrays.
- Shell suggested the increment sequence

$$h_t = \lfloor \frac{n}{2} \rfloor; h_k = \lfloor \frac{h_{k=1}}{2} \rfloor$$

For various reasons, this turns out to be a poor choice of increments. Hibbard suggested a better sequence; $1, 3, 7, ...2^k - 1$.

- Worst-case running time of Shellsort, using shell's increments, has been shown to be $O(n^2)$
- Worst-case running time of Shellsort, using Hibbard's increments, has been shown to be $O(n^{1.5})$
- Sedgewick has proposed several increment sequences that give an $O(n^{\frac{4}{3}}$ worst-case running time.
- Showing the average -case complexity of shellsort has proved to be a formidable theoretical challenge.
- Far more details, refer [weiss94, chapter 7, 256-260] and [knuth 73].

9.5 Heap Sort

Worst case as well as average case running time is $O(n \log n)$

REF. Robert W. Floyd. Algorithm 245 (TreeSort). *Communications of the ACM*, Volume 7, pp. 701, 1964.

REF. J.W.J. Williams. Algorithm 232 (Heapsort). *Communications of the ACM*, Volume 7, pp. 347-348, 1964.

- **Heap:** Recall that a heap is a complete binary tree such that the weight of every node is less than the weights of its children.

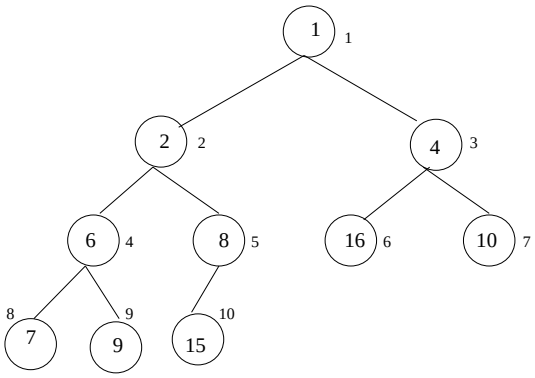


Figure 9.1: Example of a heap

- A heap with n elements can be conveniently represented as the first n elements of an array. Furthermore, the children of $a[i]$ can be found in $a[2i]$ (left child) and $a[2i+1]$ (right child)
- See Figure 9.1 for an example
- Generic heap sort algorithm:

```
{
  Insert all records to form a heap S;
  while (S is not empty)
    {
      y = min (S);
      print the value of y;
      delete y from S;
    }
}
```

Heap sort crucially uses a function called pushdown (first, last).

This assumes that the elements a[first], a[first + 1], ..., a[last] obey the heap property, except possibly the children of a[first]. The function pushes

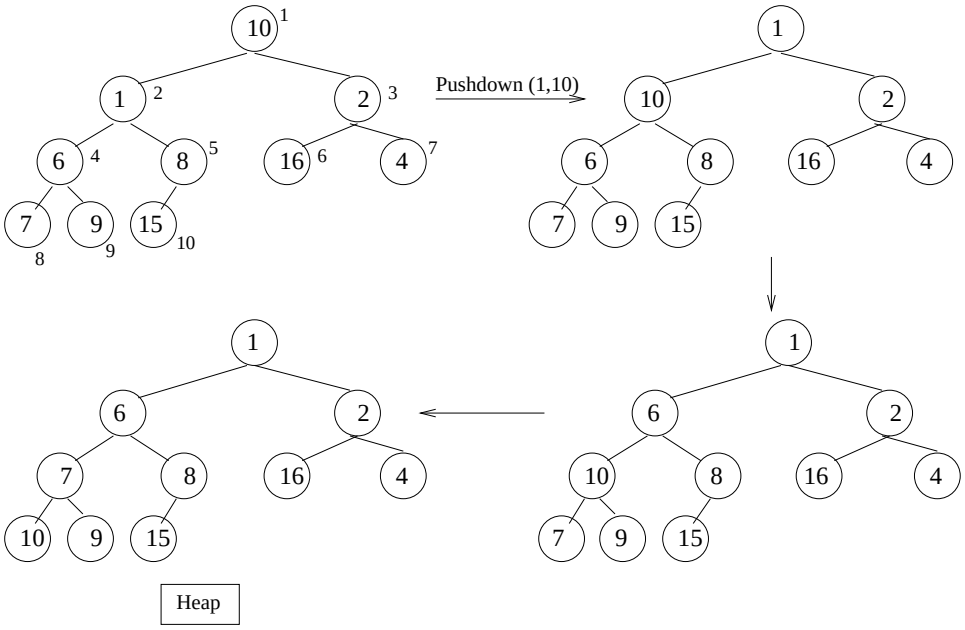


Figure 9.2: Illustration of some heap operations

a[first] down until the heap property is restored.

Example: See Figure 9.2. Here, $a[1], \ldots, a[10]$ is a heap except that the children of $a[1]$ violate the heap property.

Heapsort Algorithm

```
{
    for (i = n/2, i >= 1; i--)
      pushdown (i, n);   /* initial heap construction */
    for (i = n; i >= 2; i--)
     {
      swap records a[i] and a[1];
      pushdown (1, i - 1)
     }
  }
```

- It can be shown that the initial heap construction takes O(n) time in the worst case.

- The sorting portion takes worst case $O(n \log n)$ time.
- Heap sort can also be used for computing order statistics, ie., k^{th} lowest in a list of records.

9.6 Quick Sort

REF. C.A.R. Hoare. Algorithm 63 (Partition) and Algorithm 65 (find). *Communications of the ACM*, Volume 4, Number 7, pp. 321-322, 1961.

REF. C.A.R. Hoare. Quicksort. *The Computer Journal*, Volume 5, Number 1, pp. 10-15, 1962.

REF. Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, Volume 21, Number 10, pp. 847-857, 1978.

- Divide and conquer algorithm designed by CAR Hoare in 1962.
- Worst case $O(n^2)$ time, but average case $O(n \log n)$ time. Better average case performance than heap sort.

9.6.1 Algorithm:

To sort the records $a[i], a[i+1], \ldots, a[j]$, in place.

quicksort (i, j)

```
{
 if (a[i] ··· a[j] contain at least two distinct keys)
  {
    let v be the larger of the first two distinct keys;
    Partition a[i] ··· a[j] so that for some k between i + 1 and j,
      a[i] ··· a[k − 1] all have keys < v, and
      a[k] ··· a[j] all have keys ≥ v;
    quicksort (i, k − 1);
    quicksort (k, j);
  }
```

}

$$\underbrace{a[i] \ \cdots \ a[k-1]}_{\text{keys } <v} \quad \underbrace{a[k] \ \cdots \ a[j]}_{\text{keys } \geq v}$$

v is called the pivot. It could be any element such that the resulting partition desirably has two equal sized groups.

9.6.2 Algorithm for Partitioning

```
int partition (int i; int j; key-type pivot);
    /* partitions the elements a[k] ··· a[j],
    wrt the pivot and returns the position k */
{
  int  ℓ, r;
    {
      ℓ = i;   /* ℓ starts from left end */
      r = j;   /* r starts from right end */
      do
        swap the records a[ℓ] and a[r];
        while (a[ℓ] . key < pivot)
            ℓ = ℓ + 1;
        while (a[r] . key ≥ pivot)
            r = r − 1;
     while (ℓ ≤ r);
    return (ℓ);
    }
}
```

- For an example, see Figure 9.3.
- Worst case arises when the input is already sorted: $O(n^2)$
- Average case : $O(n \log n)$

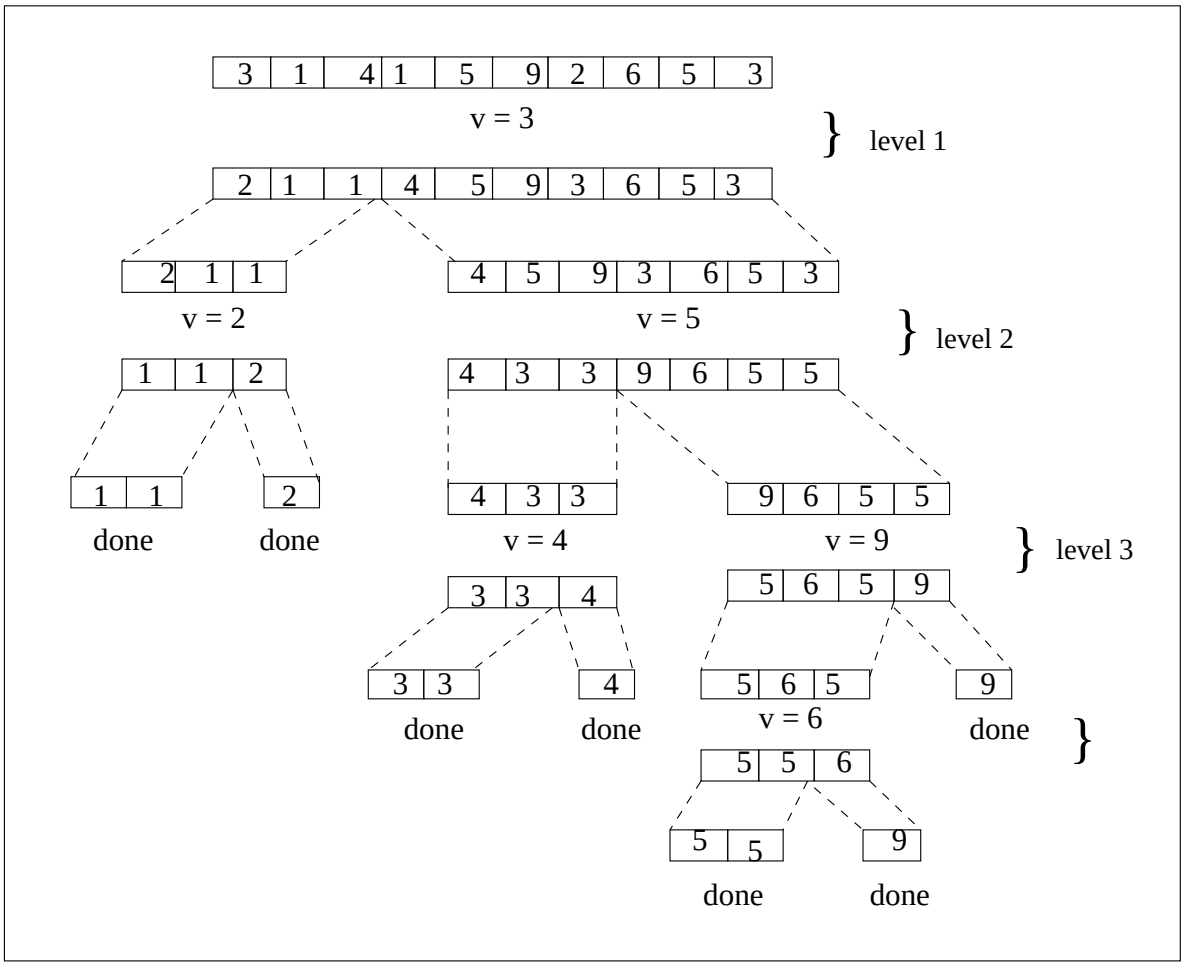


Figure 9.3: Quicksort applied to a list of 10 elements

9.6.3 Quicksort: Average Case Analysis

Assume that all initial orderings of the keys are equally likely;

- Assume that the keys are distinct

  Note that the presence of equal keys will make the sorting easier, not harder.

- Also assume that when we call quicksort (i, j), all orders for $A[i] \cdots A[j]$ are equally likely.

  Let $T(n)$ = average time taken by quicksort to sort n elements

- $T(1) = C_1$ where C_1 is some constant.

- Recall that the pivot is the larger of the first two elements.
- When $n > 1$, quicksort splits the subarray, taking $C_2 n$ time, where C_2 is another constant.

Also, since the pivot is the larger of the first two elements, left groups tend to be larger than the right groups.

The left group can have i elements where $i = 1, 2, \ldots, n-1$, since the left group has at least one element and the right group also has at least one element.

Let us fix i and try to compute the probability:

$$\text{P\{ left group has } i \text{ elements\}}$$

Now, left group has i elements

$\Rightarrow$ pivot must be the $(i+1)^{\text{st}}$ element among the n elements

If the pivot is in position 1, then the element in position 2 is one of the i smaller elements and vice-versa.

P{ Position 1 contains one of the i smaller elements }

$$= \left(\frac{i}{n-1}\right) \left(\frac{1}{n}\right)$$

Similarly,

P{ Position 2 contains one of the i smaller elements}

$$= \left(\frac{1}{n}\right) \left(\frac{i}{n-1}\right)$$

Therefore,

P{ left group has i elements}

$$= \frac{2i}{n(n-1)}$$

This leads to

$$T(n) \leq C_2 n + \sum_{i=1}^{n-1} \frac{2i}{n(n-1)} \{T(i) + T(n-i)\}$$

Using

$$\sum_{i=1}^{n-1} f(i) = \sum_{i=1}^{n-1} f(n-i),$$

we get

$$T(n) \leq C_2 n + \frac{1}{n-1} \sum_{i=1}^{n-1} \{T(i) + T(n-i)\}$$

The above expression is in the form it would have been if we had picked a truly random pivot at each step.
The above simplifies to:

$$T(n) \leq C_2 n + \frac{2}{n-1} \sum_{i=1}^{n-1} T(i)$$

The above is the recurrence that one would get if all sizes between 1 and $n-1$ for the left group were equally likely. Thus picking the larger of the two elements doesn't really affect the size distribution.
We shall guess the solution

$$T(n) \leq Cn \log n$$

for some constant C and prove its correctness using induction.

Basis:

$$n = 2 \Rightarrow Cn \log n = 2C$$

which is correct.

Induction Step:

Assume $T(i) \leq Ci \log i \quad \forall\ i < n$.

$$T(n) \ \leq \ C_2 n + \frac{2C}{n-1} \sum_{i=1}^{n-1} i \log i$$

$$\begin{aligned}
&\leq C_2 n + \frac{2C}{n-1} \sum_{i=1}^{n/2} i \log i + \frac{2C}{n-1} \sum_{i=\frac{n}{2}+1}^{n-1} i \log i \\
&\leq C_2 n + \frac{2C}{n-1} \sum_{i=1}^{n/2} i \log \frac{n}{2} + \frac{2C}{n-1} \sum_{i=\frac{n}{2}+1}^{n-1} i \log n \\
&\leq C_2 n + Cn \log n - \frac{Cn}{4} - \frac{Cn}{2(n-1)}, \text{ after simplification}
\end{aligned}$$

Picking $C \geq 4C_2$, we have,

$$C_2 n - \frac{Cn}{4} \geq 0$$

Thus $T(n)$ is $O(n \log n)$

9.7 Order Statistics

- Given a list of n records, and an integer k, find the record whose key is the k^{th} in the sorted order of keys.
- Note that

$$\begin{aligned}
K &= 1 \text{ corresponds to finding the minimum} \\
K &= n \text{ corresponds to finding the maximum} \\
K &= \frac{n}{2} \text{ corresponds to finding the median}
\end{aligned}$$

9.7.1 Algorithm 1

One can arrange the n keys into a heap and pick the k^{th} largest in k steps, where each step takes logarithmic time in the number of elements of the heap.

Since the construction of initial heap can be accomplished in worst case $O(n)$ time, the worst case complexity of this algorithm is:

$$O(n + k \log n)$$

Therefore if $k \leq \frac{n}{\log n}$ or $k \geq \frac{n}{\log n}$, the above algorithm will have $O(n)$ worst case complexity.

9.7.2 Algorithm 2

Variation of quicksort.

Select (i, j, k) $/*$ finds the k^{th} element among $A[i] \cdots A[j]$ $*/$

{

pick a pivot element v;

partition $A[i] \cdots A[j]$ so as to get

$A[i] \cdots A[m-1]$ with keys $< v$ and

$A[m] \cdots A[j]$ with keys $\geq v$;

if $(k \leq m-1)$

Select $(i, m-1, k)$

else

Select $(m, j, k-m+i)$

}

- Worst case complexity of the above algorithm is $O(n^2)$ (as in quicksort).
- Average Case:
  - Note that select calls itself only once at a time whereas quicksort called itself twice each time.
  - On an average, select calls itself on a subarray half as long as the subarray. To be conservative, suppose that each call of select is on an array $\left(\frac{9}{10}\right)^{\text{th}}$ the size on previous call. Then,

$$T(n) \leq Cn + T\left(\frac{9n}{10}\right)$$

which can be shown to be $O(n)$.

9.7.3 Algorithm 3

Worst case linear time algorithm due to

Blum, Floyd, Pratt, Rivest, Tarzan.

REF. Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, Volume 7, Number 4, pp. 448-461, 1973.

- This is also a variation of quicksort and the main idea is to find a good pivot.
- Assume that all the elements are distinct. The algorithm works otherwise too.

1. Divide the n elements into groups of 5 leaving aside between 0 and 4 elements that cannot be placed in a group.

   Sort each group of 5 elements by any algorithm and take the middle element from each group. This yields

   $$\lfloor n/5 \rfloor \text{ medians.}$$

2. Use the SELECT algorithm to find the **median** of these $\lfloor n/5 \rfloor$ elements. Choose this median as the pivot.

- Note that this pivot is in position $\lfloor \frac{n+5}{10} \rfloor$
- Also, the pivot exceeds $\lfloor \frac{n-5}{10} \rfloor$ of the middle elements and each of these middle elements exceeds two elements. Thus, the pivot exceeds at least

  $$3\lfloor \frac{n-5}{10} \rfloor \text{ elements.}$$

  Also, by a similar argument, the pivot is less than at least

  $$3\lfloor \frac{n-5}{10} \rfloor \text{ elements.}$$

- If $n \geq 75$,

  $$3\lfloor \frac{n-5}{10} \rfloor \geq 3\lfloor \frac{70}{10} \rfloor = 21 > \frac{75}{4}$$

  In other words,

  $$n \geq 75 \Rightarrow 3\left\lfloor \frac{n+5}{10} \right\rfloor \geq \frac{n}{4}$$

  This would mean that, if $n \geq 75$, the pivot is greater than at least $\frac{n}{4}$ elements and less than at least $\frac{n}{4}$ elements. Consequently, when we

partition an array with this pivot, the k^{th} element is isolated to within a range of at most $\frac{3n}{4}$ of the elements.

Implementation

```
keytype select (int i, j, k);
  /* returns key of the k^th largest element among A[i] ··· A[j] */
   {
    if ((j − i) < 75)
      find the k^th largest by some simple algorithm
    else {
          for (m = 0; m ≤ (j − i − 4)/5; m++)
           {
           find the third element among A[i+5*m] ··· A[i+5*m+4]
            and swap it with A[i + m];
            pivot = select (i, (j − i − 4)/5, (j − i − 4)/10)
            m = partition (i, j, pivot);
            if (k ≤ m − i)
              return (select (i, m − 1, k))
            else
              return (select (m, j, (k − (m − i)))
           }
        }
   }
```

The worst case complexity of the above algorithm can be described by

$$\begin{aligned} T(n) &\leq C_1 \quad \text{if } n \leq 75 \\ &\leq C_2 n + T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) \quad \text{if } n > 75 \end{aligned}$$

Let us guess the solution $T(n) = Cn$ for $n > 75$. For $n \geq 75$, assume

$T(m) \leq Cm$ for $m < n$. Then

$$\begin{aligned} T(n) &\leq C_2 n + \left(\frac{Cn}{5}\right) + \left(\frac{3Cn}{4}\right) \\ &\leq C_2 n + \frac{19}{20}Cn \end{aligned}$$

Thus $T(n)$ is $O(n)$.

- Instead of groups of 5, let us say we choose groups of 7. It is easy to show that the worst case complexity is again $O(n)$. In general, any size of the groups that ensures the sum of the two arguments of $T(.)$ to be less than n will yield $O(n)$ worst case complexity.

9.8 Lower Bound on Complexity for Sorting Methods

Result 1

The worst case complexity of any sorting algorithm that only uses key comparisons is

$$\Omega(n \log n)$$

Result 2

The average case complexity of any sorting algorithm that only uses key comparisons is

$$\Omega(n \log n)$$

The above results are proved using a **Decision Tree** which is a binary tree in which the nodes represent the status of the algorithm after making some comparisons.

Consider a node x in a decision tree and let y be its left child and z its right child. See Figure 9.4.

Basically, y represents a state consisting of the information known at x

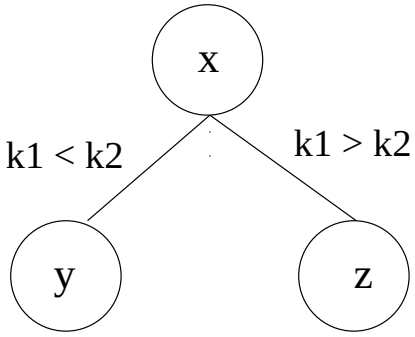


Figure 9.4: A decision tree scenario

plus the fact that the key k_1 is less than key k_2. For a decision tree for insertion sort on 3 elements, see Figure 9.5.

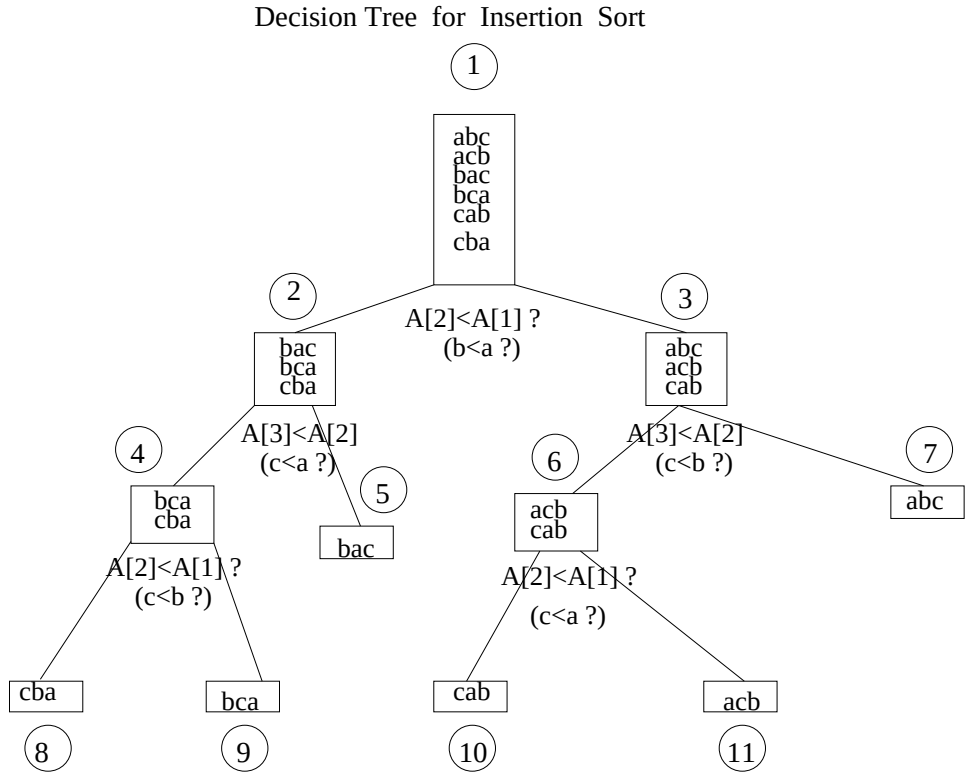


Figure 9.5: Decision tree for a 3-element insertion sort

9.8.1 Result 1: Lower Bound on Worst Case Complexity

- Given a list of n distinct elements, there are $n!$ possible outcomes that represent correct sorted orders.

$\Downarrow$

- any decision tree describing a correct sorting algorithm on a list of n elements will have at least $n!$ leaves.
- In fact, if we delete nodes corresponding to unnecessary comparisons and if we delete leaves that correspond to an inconsistent sequence of comparison results, there will be exactly $n!$ leaves.

The length of a path from the root to a leaf gives the number of comparisons made when the ordering represented by that leaf is the sorted order for a given input list L.

- The worst case complexity of an algorithm is given by the length of the longest path in the associated decision tree.
- To obtain a lower bound on the worst case complexity of sorting algorithm, we have to consider all possible decision trees having $n!$ leaves and take the minimum longest path.

In any decision tree, it is clear that the longest path will have a length of at least $\log n!$

Since

$$\begin{aligned} n! &\sim \left(\frac{n}{e}\right)^n \\ \log n! &\sim n \log n \end{aligned}$$

More Precisely,

$$\begin{aligned} n! &\geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \\ \textbf{or } \log(n!) &\geq \frac{n}{2}\log\frac{n}{2} \\ &= \frac{n}{2}\log n - \frac{n}{2} \end{aligned}$$

Thus any sorting algorithm that only uses comparisons has a worst case

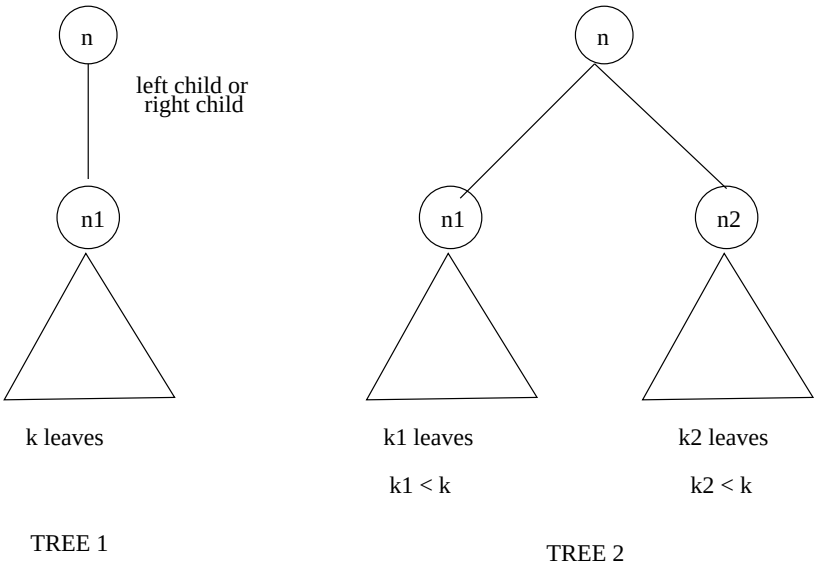


Figure 9.6: Two possibilities for a counterexample with fewest nodes

complexity

$$\Omega(n \log n)$$

9.8.2 Result 2: Lower Bound on Average Case Complexity

We shall show that in any decision tree with K leaves, the average depth of a leaf is at least log K
We shall show the result for any binary tree with K leaves.

Suppose the result is not true. Suppose T is the counterexample with the fewest nodes.
T cannot be a single node because $\log 1 = 0$. Let T have k leaves. T can only be of the following two forms. Now see Figure 9.6.

Suppose T is of the from Tree 1. The tree rooted at n_1, has fewer nodes than T but the same number of leaves and the hence an even smaller counterexample than T. Thus T cannot be of Tree 1 form.

Suppose T is of the form of Trees 2. The trees T_1 and T_2 rooted at n_1 and

n_2 are smaller than T and therefore the

$$\begin{aligned} \text{Average depth of } T_1 &\geq \log k_1 \\ \text{Average depth of } T_2 &\geq \log k_2 \end{aligned}$$

Thus the average depth of T

$$\begin{aligned} &\geq \frac{k_1}{k_1+k_2}\log k_1 + \frac{k_2}{k_1+k_2}\log k_2 + 1 \\ &= \frac{k_1}{k}\log k_1 + \frac{k_2}{k}\log k_2 + \left(\frac{k_1}{k}+\frac{k_2}{k}\right) \\ &= \frac{1}{k}\left(k_1 \log 2k_1 + k_2 \log 2k_2\right) \\ &\geq \log k \quad \begin{array}{l}\text{since the minimum value of} \\ \text{the above is attained at} \\ k_1 = k_2 \text{ giving the value } k\end{array} \end{aligned}$$

This contradicts the premise that the average depth of T is $< \log k$.

Thus T cannot be of the form of Tree 2.

Thus in any decision tree with $n!$ leaves, the average path length to a leaf is at least

$$\log(n!) \sim O(n \log n)$$

9.9 Radix Sorting

- Here we use some special information about the keys and design sorting algorithms which beat the $O(n \log n)$ lower bound for comparison-based sorting methods.
- Consider sorting n integers in the range 0 to n^2 - 1. We do it in two phases.

Phase 1: We use n bins, one for each of the integers 0, 1, $\cdots, n-1$. We

place each integer i on the list to be sorted into the bin numbered

$$i \textbf{ mod } n$$

Each bin will then contain a list of integers leaving the same remainder when divided by n.

At the end, we concatenate the bins in order to obtain a list L.

Phase 2: The integers on the list L are redistributed into bins, but using the bin selection function:

$$\lfloor \frac{i}{n} \rfloor$$

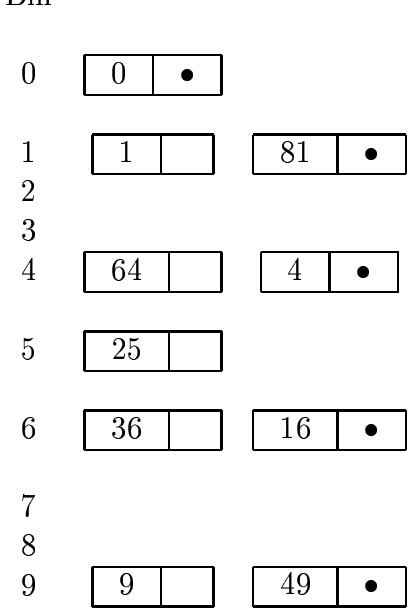
Now append integers to the ends of lists. Finally, concatenate the lists to get the final sorted sequence.

Example

n = 10

Initial list : 36, 9, 0, 25, 1, 49, 64, 16, 81, 4

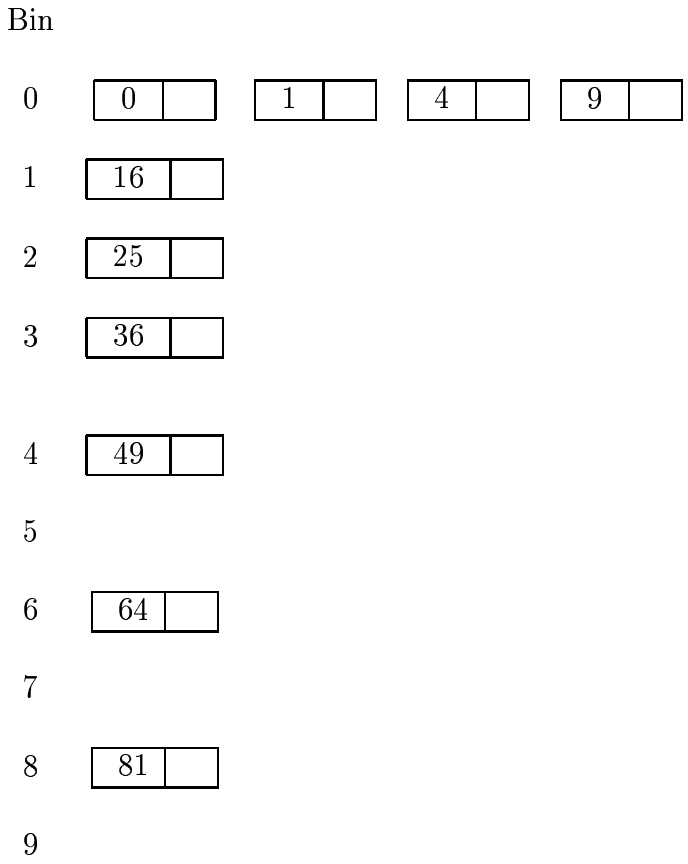
Phase 1 : bin = i **mod** 10
= right most digit of i



Concatenation would now yield the list:

L: 0, 1, 81, 64, 4, 25, 36, 16, 9, 49

Phase 2 : bin $= \lfloor i/10 \rfloor$
= right most digit of i



The concatenation now yields:

0, 1, 4, 9, 25, 36, 49, 64, 81

In general, assume that the key-type consists of k components $f_1, f_2, \cdots, f_k$, of type $t_1, t_2, \cdots, t_k$.

Suppose it is required to sort the records in lexicographic order of their keys. That is,

$$(a_1, a_2, \ldots, a_k) < (b_1, b_2, \ldots, b_k)$$

if one of the following holds:

1. $a_1 < b_1$
2. $a_1 = b_1; a_2 < b_2$
3. $a_1 = b_1; a_2 = b_2; a_3 < b_3$

   .

   .

   .

k. $a_1 = b_1; \ldots; a_{k-1} = b_{k-1}; a_k < b_k$

Radix sort proceeds in the following way.

First binsort all records first on f_k, the least significant digit, then concatenate the bins lowest value first.

Then binsort the above concatenated list on f_{k-1} and then concatenate the bins lowest value first.

In general, after binsorting on $f_k, f_{k-1}, \ldots, f_i$, the records will appear in lexicographic order if the key consisted of only the fields $f_i, \ldots, f_k$.

```
void radixsort;
  / * Sorts a list A of n records with keys consisting of fields
  f1, ..., fk of types t1, ..., tk. The function uses k
  arrays B1, ..., Bk of type array [ti] of list-type, i = 1, ..., k,
  where list-type is a linked list of records */
 {
  for (i = k; i ≥ 1; i − −)
    {
     for (each value v of type ti)
       make Bi[v] empty; /* clear bins */
     for (each value r on list A)
       move r from A onto the end of bin Bi[v],
       where v is the value of the field fi of the key of r;
     for (each value v of type ti from lowest to highest)
       concatenate Bi[v] onto the end of A
    }
```

```
}
```

- Elements to be sorted are presented in the form of a linked list obviating the need for copying a record. We just move records from one list to another.
- For concatenation to be done quickly, we need pointers to the end of lists.

Complexity of Radix Sort

- Inner loop 1 takes $O(s_i)$ time where s_i = number of different values of type t_i
- Inner loop 2 takes $O(n)$ time
- Inner loop 3 times $O(s_i)$ time

  Thus the total time

$$
\begin{aligned}
&= \sum_{i=1}^{k} O(s_i + n) \\
&= O\left(kn + \sum_{i=1}^{k} s_i\right) \\
&= O\left(n + \sum_{i=1}^{k} s_i\right) \quad \text{assuming } k \text{ to be a constant}
\end{aligned}
$$

- For example, if keys are integers in the range 0 to $n^k - 1$, then we can view the integers as radix $-n$ integers each k digits long. Then t_i has range $0 \cdots n-1$ for $1 \leq i \leq k$.

  Thus $s_i = n$ and total running time is $O(n)$
- similarly, if keys are character strings of length k, for constant k, then $s_i = 128$ (say) for $i = 1, \ldots, k$ and again we get the total running time as $O(n)$.

9.10 Merge Sort

- Divide and conquer algorithm
- $O(n \log n)$ worst case performance
- Useful in external sorting applications.

Algorithm

---

list-type mergesort (list-type L; **int** n)
{
if $(n = 1)$
return (L)
else {
split L into two halves L_1 and L_2;
return (merge (mergesort $\left(L_1, \frac{n}{2}\right)$, mergesort $\left(L_2, \frac{n}{2}\right)$)
}
}

Let $T(n)$ be the running time of mergesort on a list of size n. Then.

$$\begin{aligned} T(n) &\leq C_1 \quad (n = 1) \\ &\leq \underbrace{2T\left(\frac{n}{2}\right)}_{\substack{\text{divide} \\ \text{and} \\ \text{conquer}}} + \underbrace{C_2 n}_{\text{merge}} \quad (n > 1) \end{aligned}$$

It can be shown that $T(n)$ is $O(n \log n)$.

Mergesort as an External Sorting Procedure

- Assume that the data to be sorted is stored in a file
- The essential idea is to organize a file into progressively larger **runs** where:

  A **run** is a sequence of records $r_1, \ldots, r_k$, such that $r_i \leq r_{i+1}$ for $i = 1, \ldots, k-1$.

- We say a file, $r_1, r_2, \ldots, r_m$, of records is organized into **runs** or length k if: $\forall\ i \leq 1$ such that $ki \leq m$, the sequence

$$\left(r_{k(i-1)+1}, r_{k(i-1)+2}, \ldots, r_{ki}\right)$$

  is a run of length k and furthermore if m is not divisible by k and $m = pk+q$ where $q < k$, the sequence $(r_{m-q+1}, \ldots, r_m)$ which is called the **tail** is a run of length q.

Example

The following is organized into runs of length 3 with the tail having length 2.

7	15	29	8	11	13	16	22	31	5	12

Basic step

Consider two files f_1 and f_2 organized into runs of length k. Assume that

1. If n_i is the number of runs (including tails) of $f_1 (i = 1, 2)$, then $|n_1 - n_2| \leq 1$
2. At most one of f_1 and f_2 has a tail
3. The one with a tail (if any) has at least as many runs as the other.

In mergesort, we read one run from each of f_1 and f_2, merge the runs into a run of length 2k and append it to one of two files g_1 and g_2. By alternating between g_1 and g_2, these files would be organized into runs of length $2k$, satisfying (1), (2), and (3).

Algorithm

- First divide all n records into two files f_1 and f_2 as evenly as possible. f_1 and f_2 can be regarded as organized into runs of length 1.
- Merge the runs of length 1 and distribute them into files g_1 and g_2 which will now be organized into runs of length 2.
- Now make f_1 and f_2 empty and merge g_1 and g_2 into f_1 and f_2 creating runs of length 4.

Repeat $\cdots$

Example

f_1	28	3	93	54	65	30	90	10	69	8	22
f_2	31	5	96	85	9	39	13	8	77	10	

runs of length 1

$\Downarrow$ merge into g_1 and g_2

g_1	28	31	93	96	9	65	13	90	69	77	22
g_2	3	5	54	85	30	39	8	10	8	10	

runs of length 2

$\Downarrow$ merge into f_1 and f_2

f_1	3	5	28	31	9	30	39	65	8	10	69	77
f_2	54	85	93	96	8	10	13	90	22			

runs of length 4

$\Downarrow$ merge into g_1 and g_2

g_1	3	5	28	31	54	85	93	96	8	10	22	69	77
g_2	8	9	10	13	30	39	65	90					

runs of length 8

$\Downarrow$ merge into f_1 and f_2

f_1 3 5 8 9 10 13 28 30 31
39 54 65 85 90 93 96

f_2 8 10 22 69 77

runs of length 16

$\Downarrow$ merge into g_1 and g_2

g_1 will be left with the sorted sequence

- After i passes, we have two files consisting of runs of length 2^i. If $2^i \geq n$, then one of the two files will be empty and the other will contain a single run of length n (which is the sorted sequence).

  Therefore number of passes $= \lceil \log n \rceil$

- Each pass requires the reading of two files and the writing of two files, all of length $\sim \frac{n}{2}$.

  Total number of blocks read or written in a pass $\sim \frac{2n}{b}$

  (where b is the number of records that fit into a block)

  Total number of block reads or writes for the entire sorting$\} \sim O\left(\frac{n \log n}{b}\right)$.

- The procedure to merge reads and writes only one record at a time. Thus it is not required to have a complete run in memory at any time.

- Mergesort need not start with runs of length 1. We could start with a pass that, for some appropriate k, reads groups of k records into main memory, sorts them (say quicksort) and writes them out as a run of length k.

Multiway Mergesort

- Here we merge m files at a time. Let the files $f_1, \ldots, f_m$ be organized as runs of length k.

  We can read m runs, one from each file, and merge them into one run of length mk. This run is placed on one of m files $g_1, \ldots, g_m$, each getting a run in turn.

- The merging process will involve each time selecting the minimum among m currently smallest unselected records from each file.

  By using a priority queue that supports insert and delete in logarithmic time (e.g., partially ordered tree), the process can be accomplished in $O(\log m)$ time.

- Number of passes - $\lceil \log_m n \rceil$
  Effort in each pass $= O(n \log_2 m)$
  overall complexity $\sim O(n \log_2 m . \log_m n)$

- **Advantages**

  - We save by a factor of $\log_2 m$, the number of times we read each record
  - We can process data $O(m)$ times faster with m disk units.

9.11 To Probe Further

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

2. Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996. Indian Edition published by Prentice Hall of India, 1998.

3. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.

4. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.

5. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

6. Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Third Edition, 2000. Indian Edition published by Pearson Education Asia, 2000.

7. Ellis Horowitz, Sartaz Sahni, and Rajasekaran. *Fundamentals of Computer Algorithms*. W.H. Freeman and Company, 1998. Indian Edition published by Galgotia Publications, 2000.

8. Donald E Knuth. *Fundamental Algorithms*, Volume 1 of The Art of Computer Programming, Addison-Wesley, 1968, Second Edition, 1973.

9. Donald E. Knuth. *Sorting and Searching*, Volume 3 of The Art of Computer Programming, Addison-Wesley, 1973.

10. Y. Langsam, M.J. Augenstein, and A.M. Tenenbaum. *Data Structures using C and C++*. Second Edition, Prentice Hall, 1996. Indian Edition published by Prentice Hall of India, 2000.

11. Kurt Mehlhorn. *Sorting and Searching.* Volume 1 of *Data Structures and Algorithms.* Springer-Verlag, 1984.

12. Nicklaus Wirth. *Data Structures + Algorithms = Programs*. Prentice-Hall, Englewood Cliffs. 1975.

13. Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, Volume 7, Number 4, pp.448-461, 1973.

14. Robert W. Floyd. Algorithm 245 (TreeSort). *Communications of the ACM*, Volume 7, pp.701, 1964.

15. C.A.R. Hoare. Algorithm 63 (Partition) and Algorithm 65 (find). *Communications of the ACM*, Volume 4, Number 7, pp 321-322, 1961.

16. C.A.R. Hoare. Quicksort. *The Computer Journal*, Volume 5, Number 1, pp.10-15, 1962.

17. Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, Volume 21, Number 10, pp.847-857, 1978.

18. Shell. A High-speed Sorting Procedure. *Communications of the ACM*, Volume 2, Number 7, pp. 30-32,1959.

9.12 Problems

1. Consider the following sorting methods: Bubble sort, Insertion Sort, Selection sort, Shell sort, Merge sort, Quick sort, and Heap sort.

   - Sort the following keys using each of the above methods:
     22, 36, 6, 79, 26, 45, 2, 13, 31, 62, 10, 79, 33, 11, 62, 26
   - A sorting algorithm is said to be *stable* if it preserves the original order of records with equal keys. Which of the above methods are stable?
   - Suppose you are to sort a list L comprising a sorted list followed by a few random elements. Which of the above sorting methods would you prefer and why?

2. Show that if k is the smallest integer greater than or equal to $n + \log_2 n - 2$, k comparisons are necessary and sufficient to find the largest and second largest elements of a set of n distinct elements.

3. Design an algorithm to find the two smallest elements in an array of length n. Can this be done in fewer than $2n - 3$ comparisons?

4. Show how to find the minimum and maximum elements in an array using only $(2n - 3)$ comparisons, where n is the size of the array.

5. Show that any sorting algorithm that moves elements only one position at a time must have time complexity $\Omega(n^2)$.

6. Design an efficient algorithm to find all duplicates in a list of n elements.

7. Find a sorting method for four keys that is optimal in the sense of doing the smallest possible number of key comparisons in the worst case. Find how many comparisons your algorithm does in the average case.

8. Suppose we have a set of words, i.e., strings of the letters a–z, whose total length is n. Show how to sort these in $O(n)$ time. Note that if the maximum length of a word is constant, then bin sort will work. However, you must consider the case where some of the words are very long.

9. Suppose we have a sorted array of strings $s_1, \ldots, s_n$. Write an algorithm to determine whether a given string x is a member of this sequence. What is the time complexity of your algorithm as a function of n and the length of x?

10. Design an algorithm that will arrange a contiguous list of real numbers so that all the items with negative keys will come first, then those with nonnegative keys. The final list need not be sorted.

11. Design an algorithm that will rearrange a list of integers as described in each case below.

- All even integers come before all odd integers.
- Either all the integers in even-numbered positions will be even or all integers in the odd-numbered positions will be odd. First, prove that one or the other of these goals can be achieved, although it may not be possible to achieve both goals at the same time.

12. Suppose that the splits at every level of quicksort are in the proportion $1-\alpha$ to α, where $0 < \alpha \leq \frac{1}{2}$ and α is a constant. Show that the minimum depth of a leaf in the recursion tree of quicksort is approximately $-\frac{\log n}{\log \alpha}$ and the maximum depth is approximately $-\frac{\log n}{\log(1-\alpha)}$. (Ignore the integer round off).

13. Recall the algorithm Select(i, j, k) that finds the kth element in the sorted order of the elements $a[i], a[i+1], ..., a[j]$. Choose the pivot as follows. Divide the elements into groups of 3 (leaving aside between 0 and 2 elements that cannot be placed in a group), sort each group, and take the middle element from each group; a total of say, p, middle elements will result. Choose the median of these p elements as the pivot. Let $T(n)$ be the time taken by a call to Select on n elements. Write down an appropriate recurrence relation for $T(n)$. Is $T(n)$, $O(n)$?

14. In the above problem, choose the pivot as follows. Divide the elements into groups of 7, leaving aside between 0 and 6 elements that cannot be placed in a group. Sort each group and take the middle element from each group. Choose the median of these middle elements as the pivot.

    Let $T(n)$ be the time taken by a call to *select* on n elements. Write down an appropriate recurrence for $T(n)$ and show that it is $O(n)$.

15. A d-ary heap is like a binary heap, but in stead of two children, nodes have d children.

    - How would you represent a d-ary heap in an array?
    - What is the height of a d-ary heap of n elements in terms of n and d?
    - Give an efficient implementation of deletemin and insert operations and analyze the running time in terms of d and n.

16. Consider constructing a heap by first forming a complete binary tree and then repeatedly applying the *pushdown* procedure. What input permutations of $(1, 2, 3, 4, 5)$ are transformed into $(1, 2, 4, 3, 5)$ by this process?

17. Give a permutation of $1, 2, \ldots, 8$, which when input to the quicksort algorithm will produce the best possible performance of the quicksort algorithm (assume that the larger of the first two keys is selected as the pivot for partitioning).

18. Suppose we have an array of n data records such that the key of each record has the value 0 or 1. Outline a worst case linear time algorithm to sort these records *in place*, using only an additional amount of storage equal to that of one record. Is your algorithm stable? Justify.

19. Outline an $O(n \log k)$ algorithm to merge k sorted lists into a single sorted list, where n is the total number of elements in all the input lists.

20. Outline an efficient algorithm, using the binomial queue data structure to **merge** k sorted lists into a single sorted list. What is the worst-case complexity of your algorithm (in terms of k and n), if n is the total number of elements in all the input lists.

21. Suppose we have an array of n data records such that the key of each record has the value 0, 1, or 2. Outline a worst case linear time algorithm to sort these records *in place*, using only an additional amount of storage equal to that of one record. Is your algorithm stable? Justify.

22. Write down a decision tree for sorting three elements a, b, c using *bubble sort*, with proper annotations on the nodes and edges of the tree.

9.13 Programming Assignments

9.13.1 Heap Sort and Quicksort

Implement heapsort and quicksort. Design and carry out an experiment to compare their average case complexities.

Chapter 10

Introduction to NP-Completeness

10.1 Importance of NP-Completeness

Most algorithms we have studied so far have polynomial-time running times. According to Cormen, Leiserson, and Rivest, polynomial-time algorithms can be considered tractable for the following reasons.

(1) Although a problem which has a running time of say $O(n^{20})$ or $O(n^{100})$ can be called intractable, there are very few practical problems with such orders of polynomial complexity.

(2) For reasonable models of computation, a problem that can be solved in polynomial time in one model can also be solved in polynomial time on another.

(3) The class of polynomial-time solvable problems has nice closure properties (since polynomials are closed under addition, multiplication, etc.)

The class of NP-complete (Non-deterministic polynomial time complete) problems is a very important and interesting class of problems in Computer Science. The interest surrounding this class of problems can be attributed to the following reasons.

1. No polynomial-time algorithm has yet been discovered for any NP-complete problem; at the same time no NP-complete problem has been shown to have a super polynomial-time (for example exponential time) lower bound.

2. If a polynomial-time algorithm is discovered for even one NP-complete problem, then all NP-complete problems will be solvable in polynomial-time.

It is believed (but so far no proof is available) that NP-complete problems do not have polynomial-time algorithms and therefore are intractable. The basis for this belief is the second fact above, namely that if any single NP-complete problem can be solved in polynomial time, then every NP-complete problem has a polynomial-time algorithm. Given the wide range of NP-complete problems that have been discovered to date, it will be sensational if all of them could be solved in polynomial time.

It is important to know the rudiments of NP-completeness for anyone to design "sound" algorithms for problems. If one can establish a problem as NP-complete, there is strong reason to believe that it is intractable. We would then do better by trying to design a good approximation algorithm rather than searching endlessly seeking an exact solution. An example of this is the TSP (Traveling Salesman Problem), which has been shown to be intractable. A practical strategy to solve TSP therefore would be to design a good approximation algorithm. This is what we did in Chapter 8, where we used a variation of Kruskal's minimal spanning tree algorithm to approximately solve the TSP. Another important reason to have good familiarity with NP-completeness is many natural interesting and innocuous-looking problems that on the surface seem no harder than sorting or searching, are in fact NP-complete.

10.2 Optimization Problems and Decision Problems

NP-completeness has been studied in the framework of *decision problems*. Most problems are not decision problems, but optimization problems (where some value needs to be minimized or maximized). In order to apply the theory of NP-completeness to optimization problems, we must recast them as decision problems. We provide an example of how an optimization problem can be transformed into a decision problem.

Example: Consider the problem SHORTEST-PATH that finds a shortest path between two given vertices in an unweighted, undirected graph

$G = (V, E)$. An instance of SHORTEST-PATH consists of a particular graph and two vertices of that graph. A solution is a sequence of vertices in the graph, with perhaps an empty sequence denoting that no path exists. Thus the problem SHORTEST-PATH is a relation that associates each instance of a graph and two vertices with a solution (namely a shortest path in this case). Note that a given instance may have no solution, exactly one solution, or multiple solutions.

A decision problem PATH related to the SHORTEST-PATH problem above is : Given a graph $G = (V, E)$, two vertices $u, v \in V$, and a non-negative integer k, does a path exist in G between u and v whose length is at most k ?

Note that the decision problem PATH is one way of casting the original optimization problem as a decision problem. We have done this by imposing a bound on the value to be optimized. This is a popular way of transforming an optimization problem into a decision problem.

If an optimization problem is easy then its related decision problem is easy as well. Similarly, if we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard.

10.3 Examples of some Intractable Problems

10.3.1 Traveling Salesman Problem

A Hamiltonian cycle in an undirected graph is a simple cycle that passes through every vertex exactly once.

Optimization Problem: Given a complete, weighted graph, find a minimum-weight Hamiltonian Cycle.

Decision Problem: Given a complete, weighted graph and an integer k, does there exist a Hamiltonian cycle with total weight at most k.

10.3.2 Subset Sum

The input is a positive integer C and n objects whose sizes are positive integers $s_1, s_2, \ldots, s_n$.

Optimization Problem: Among all subsets of objects with sum at most C, what is the largest subset sum?

Decision Problem: Is there a subset of objects whose sizes add up to exactly C?

10.3.3 Knapsack Problem

This is a generalization of the subset sum problem. Consider a knapsack of capacity C where C is a positive integer and n objects with positive integer sizes $s_1, s_2, \ldots, s_n$ and positive integer profits $p_1, p_2, \ldots, p_n$.

Optimization Problem: Find the largest total profit of any subset of the objects that fits in the knapsack.

Decision Problem: Given k, is there a subset of the objects that fits in the knapsack and has total profit at least k?

10.3.4 Bin Packing

Suppose we have unlimited number of bins each of unit capacity and n objects with sizes $s_1, s_2, \ldots, s_n$ where the sizes $s_i (i = 1, \ldots, n)$ are rational numbers in the range $0 < s_i \leq 1$.

Optimization Problem: Determine the smallest number of bins into which the objects can be packed and find an optimal packing.

Decision Problem: Given an integer k, do the objects fit in k bins?

10.3.5 Job Shop Scheduling

Suppose there are n jobs, $J_1, J_2, \ldots, J_n$ to be processed one at a time in non-preemptive fashion. Let the processing time be $t_1, t_2, \ldots, t_n$ and the due dates be $d_1, d_2, \ldots, d_n$. A schedule for the jobs is a permutation π of $1, 2, \ldots, n$. Job $J_i (i = 1, \ldots, n)$ will incur a penalty of $p_i (i = 1, \ldots, n)$ if it misses the due-date in the given schedule π. If processed within the due-date, the penalty is taken as zero. The processing times, due-dates, and penalties are all positive integers. The penalty of a schedule is the sum of penalties incurred by jobs processed according to that schedule.

Optimization Problem: Determine the minimum possible penalty for a schedule and find an (optimal) schedule that achieves the minimum penalty.

Decision Problem: Given a non-negative integer k, does there exist a schedule with penalty at most k?

10.3.6 Satisfiability

A propositional variable is one that can be assigned the value *true* or *false*. A literal is a propositional variable or its negation. A clause is a sequence of literals separated by the logical OR operator. A propositional formula is said to be in conjunctive normal form (CNF) if it consists of a sequence of clauses separated by the logical AND operator. For example,

$$(p \vee q \vee s) \wedge (\overline{q} \vee r) \wedge (\overline{p} \vee \overline{q} \vee r \vee \overline{s})$$

A truth assignment for a set of propositional variables is an assignment of *true* or *false* value to each propositional variable. A truth assignment is said to satisfy a formula if it makes the value of the formula *true*.

3-SAT (Decision Problem): Given a CNF formula in which each clause is permitted to contain at most three literals, is there a truth assignment to its variables that satisfies it?

10.4 The Classes P and NP

An algorithm is said to be polynomially bounded if its worst-case complexity is bounded by a polynomial function of the input size. A problem is said to be polynomially bounded if there is a polynomially bounded algorithm for it.

P is the class of all decision problems that are polynomially bounded. The implication is that a decision problem X $\in$ **P** can be solved in polynomial time on a deterministic computation model (such as a deterministic Turing machine).

NP represents the class of decision problems which can be solved in polynomial time by a non-deterministic model of computation. That is, a decision problem X $\in$ **NP** can be solved in polynomial-time on a non-deterministic computation model (such as a non-deterministic Turing machine). A non-deterministic model can make the right guesses on every move and race towards the solution much faster than a deterministic model.

A deterministic machine, at each point in time, executes an instruction. Depending on the outcome of executing the instruction, it then executes some next instruction, which is unique. A non-deterministic machine on the other hand has a choice of next steps. It is free to choose any that it wishes. For example, it can always choose a next step that leads to the best solution for the problem. A non-deterministic machine thus has the power of extremely good, optimal guessing.

As an example, let us consider the decision version of TSP: Given a complete, weighted graph and an integer k, does there exist a Hamiltonian cycle with total weight at most k?

A smart non-deterministic algorithm for the above problem starts with a vertex, guesses the correct edge to choose, proceeds to the next vertex, guesses the correct edge to choose there, etc. and in polynomial time discovers a Hamiltonian cycle of least cost and provides an answer to the above problem. This is the power of non-determinism. A deterministic algorithm here will have no choice but take super-polynomial time to answer the above question.

Another way of viewing the above is that given a candidate Hamiltonian cycle (call it *certificate*), one can verify in polynomial time whether the

answer to the above question is YES or NO. Thus to check if a problem is in **NP**, it is enough to prove in polynomial time that any YES instance is correct. We do not have to worry about NO instances since the program always makes the right choice.

It is easy to show that **P** $\subseteq$ **NP**. However, it is unknown whether **P** = **NP**. In fact, this question is perhaps the most celebrated of all open problems in Computer Science.

10.5 NP-Complete Problems

The definition of NP-completeness is based on reducibility of problems. Suppose we wish to solve a problem X and we already have an algorithm for solving another problem Y. Suppose we have a function T that takes an input x for X and produces T(x), an input for Y such that the correct answer for X on x is yes if and only if the correct answer for Y on T(x) is yes. Then by composing T and the algorithm for y, we have an algorithm for X.

- If the function T itself can be computed in polynomially bounded time, we say X is polynomially reducible to Y and we write X $\leq_p$ Y.
- If X is polynomially reducible to Y, then the implication is that Y is at least as hard to solve as X. i.e. X is no harder to solve than Y.
- It is easy to see that

  X $\leq_p$ Y and Y $\in$ **P** implies X $\in$ **P**.

10.5.1 NP-Hardness and NP-Completeness

A decision problem Y is said to be NP-hard if X $\leq_p$ Y $\forall$ X $\in$ **NP**. An NP-hard problem Y is said to be NP-complete if Y $\in$ **NP**. **NPC** is the standard notation for the class of all NP-complete problems.

- Informally, an NP-hard problem is a problem that is at least as hard as any problem in **NP**. If, further, the problem also belongs to **NP**, it would become NP-complete.

- It can be easily proved that if any NP-complete problem is in **P**, then **NP** = **P**. Similarly, if any problem in **NP** is not polynomial-time solvable, then no NP-complete problem will be polynomial-time solvable. Thus NP-completeness is at the crux of deciding whether or not **NP** = **P**.

- Using the above definition of NP-completeness to show that a given decision problem, say Y, is NP-complete will call for proving polynomial reducibility of each problem in **NP** to the problem Y. This is impractical since the class **NP** already has a large number of member problems and will continuously grow as researchers discover new members of **NP**.

- A much more practical way of proving NP-completeness of a decision problem Y is to discover a problem X $\in$ **NPC** such that Y $\leq_p$ X. Since X is NP-complete and $\leq_p$ is a transitive relationship, the above would mean that Z $\leq_p$ Y $\quad \forall \quad$ Z $\in$ **NP**. Furthermore if Y $\in$ **NP**, then Y is NP-complete.

  The above is the standard technique used for showing the NP-hardness or NP-completeness of a given decision problem. For example, all the decision problems given in Section 10.2 can be shown to be NP-complete by the above technique.

- An interesting question is: how was the first member of **NPC** found? Stephen Cook showed the NP-completeness of the problem 3-SAT by directly proving that X $\leq_p$ 3-SAT $\forall$ X $\in$ **NP**.

- If a problem is NP-complete, it does not mean that all hopes are lost. If the actual input sizes are small, an algorithm with, say, exponential running time may be acceptable. On the other hand, it may still be possible to obtain near-optimal solutions in polynomial-time. Such an algorithm that returns near-optimal solutions (in polynomial time) is called an approximation algorithm. Using Kruskal's algorithm to obtain a suboptimal solution to the TSP (see Chapter 8) is an example of this.

10.6 To Probe Further

This chapter has presented a first level introduction to the notion of NP-completeness. We have consciously avoided a technical discussion of the topics since that would be the subject of a more advanced course. For a rigorous treatment of the subject, you should consult the following references.

The following book is a classic source on the theory of NP-completeness and also contains a catalogue of NP-complete problems discovered until 1979.

REF. M.R. Garey and D.S. Johnson. Computers and Intractability : A Guide to the Theory of NP-completeness. W.H. Freeman, 1979.

The class **P** was first introduced by Cobham in 1964,

REF. Alan Cobham. The intrinsic computational difficulty of functions. Proceedings of the 1964 Congress for Logic, Methodology, and Philosophy of Sciences, North-Holland, 1964, pp.24-30.

The class **P** was also independently introduced in 1965 by Edmonds, who also conjectured $\mathbf{P} \neq \mathbf{NP}$ for the first time.

REF. Jack Edmonds. Paths, trees, and flowers. Canadian Journal of Mathematics, Volume 17, 1965, pp. 449-467.

The notion of NP-completeness was first proposed in 1971 by Stephen Cook who also gave the first NP-completeness proof for 3-SAT.

REF. Stephen Cook. The complexity of theorem proving procedures. Proceedings of the Third Annual ACM Symposium on Theory of Computing, 1971, pp. 151-158.

The technique of polynomial reductions was introduced in 1972 by Karp, who also demonstrated a rich variety of NP-complete problems.

REF. Richard M Karp. Reducibility among combinatorial problems. In Complexity of computer computations, edited by R.E. Miller and J.W Thather, Plenum Press, 1972 pp. 85-103.

This chapter has liberally drawn from the material presented in two books: Chapter 36 of the book by Cormen, Leiserson, and Rivest and Chapter 13 of the book by Sara Baase and Allen Van Gelder. These two books are a rich source of NP-complete problems and NP-completeness proofs.

10.7 Problems

1. Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

2. A Hamiltonian path in a graph is a simple path that visits every vertex exactly once. Show that the decision problem of determining whether or not there is a Hamiltonian path from a given vertex u to another given vertex v belongs to **NP**

3. Show that the $\leq_p$ relation is a transitive relation on the set of problems.

4. Given an undirected graph, the Hamiltonian cycle problem determines whether a graph has a Hamiltonian cycle. Given that the Hamiltonian cycle problem for undirected graphs is NP-Complete, what can you say about the Hamiltonian cycle problem for directed graphs? Provide reasons for your answer.

Chapter 11

References

11.1 Primary Sources for this Lecture Notes

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

2. Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, 1996. Indian Edition published by Prentice Hall of India, 1998.

3. Thomas H. Cormen, Charles E. Leiserson, and Donald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series, 1990. Indian Edition published in 1999.

4. Donald E Knuth. *Fundamental Algorithms*, Volume 1 of The Art of Computer Programming, Addison-Wesley, 1968, Second Edition, 1973.

5. Robert L. Kruse, Bruce P. Leung, and Clovis L. Tondo. *Data Structures and Program design in C*. Prentice Hall, 1991. Indian Edition published by Prentice Hall of India, 1999.

6. Mark Allen Weiss. *Data Structures and Algorithm Analysis in C++*. Benjamin-Cummings, 1994. Indian Edition published in 1998.

11.2 Useful Books

1. Alfred V Aho, John E. Hopcroft, and Jeffrey D Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

2. Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, Third Edition, 2000. Indian Edition published by Pearson Education Asia, 2000.

3. Duane A. Bailey. *Java Structures: Data Structures in Java for the Principled Programmer*. McGraw-Hill International Edition, 1999.

4. Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

5. Jon L. Bentley. *Writing Efficient Programs*, Prentice-Hall, 1982.

6. Jon L. Bentley. *Programming Pearls*, Addison-Wesley, 1986.

7. Jon L. Bentley. *More Programming Pearls*, Addison-Wesley, 1988.

8. Gilles Brassard and Paul Bratley. *Algorithmics : Theory and Practice*. Prentice-Hall, 1988.

9. Michael R. Garey and David S Johnson. *Computers and Intractability: A Guide to Theory of NP-Completeness*. W.H. Freeman, 1979.

10. R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-wesley, Reading, 1990. Indian Edition published by Addison-Wesley Longman, 1998.

11. Ellis Horowitz and Sartaz Sahni. *Fundamentals of Data structures*. Galgotia Publications, New Delhi, 1984.

12. Ellis Horowitz, Sartaz Sahni, and Rajasekaran. *Fundamentals of Computer Algorithms*. W.H. Freeman and Company, 1998. Indian Edition published by Galgotia Publications, 2000.

13. Donald E Knuth. *Fundamental Algorithms*, Volume 1 of The Art of Computer Programming, Addison-Wesley, 1968, Second Edition, 1973.

14. Donald E Knuth. *Seminumerical Algorithms*. Volume 2 of The Art of Computer Programming, Addison-Wesley, 1969, Second Edition, 1981.

15. Donald E. Knuth. *Sorting and Searching*, Volume 3 of The Art of Computer Programming, Addison-Wesley, 1973.

16. Y. Langsam, M.J. Augenstein, and A.M. Tenenbaum. *Data Structures using C and C++*. Second Edition, Prentice Hall, 1996. Indian Edition published by Prentice Hall of India, 2000.

17. Eugene L Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B.Shmoys, editors. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

18. Kurt Mehlhorn. *Sorting and Searching*. Volume 1 of *Data Structures and Algorithms*. Springer-Verlag, 1984.

19. Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*. Volume 2 of *Data Structures and Algorithms*, Springer-Verlag, 1984.

20. Kurt Mehlhorn. *Multidimensional Searching and Computational Geometry*. Volume 3 of *Data Structures and Algorithms*, Springer-Verlag, 1984.

21. Sartaj Sahni. *Data Structures, Algorithms, and Applications in Java*. McGraw-Hill Higher Education, 2000.

22. Robert Sedgewick. *Algorithms*. Addison-Wesley, Second Edition, 1988.

23. Thomas A. Standish. *Data Structures in Java*. Addison-Wesley, 1998. Indian Edition published by Addison Wesley Longman, 2000.

24. Nicklaus Wirth. *Data Structures + Algorithms = Programs*. Prentice-Hall, Englewood Cliffs. 1975.

11.3 Original Research Papers and Survey Articles

1. G.M. Adel'son-Vel'skii and E.M. Landis. An algorithm for the organization of information. *Soviet Mathematics Monthly*, Volume 3, pp.1259-1263, 1962.

2. R. Bayer. Symmetric binary B-trees: Data Structures and maintenance algorithms, *Acta Informatica*, Volume 1, pp.290-306, 1972.

3. R. Bayer and E.M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, Volume 1, Number 3, pp. 173-189, 1972.

4. Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*. Volume 16, Number 1, pp. 87–90, 1958.

5. Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, Volume 7, Number 4, pp.448-461, 1973.

6. Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, Volume 7, Number 3, pp. 298-319, 1978.

7. Alan Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, pages 24-30, North-Holland, 1964.

8. D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, Volume 11, Number 2, pp 121-137, 1979.

9. Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151-158, 1971.

10. E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, Volume 1, pp 269-271, 1959.

11. Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, Volume 17, pp 449-467, 1965.

12. Robert W Floyd. Algorithm 97 (Shortest Path). *Communications of the ACM*, Volume 5, Number 6, pp. 345, 1962.

13. Robert W. Floyd. Algorithm 245 (TreeSort). *Communications of the ACM*, Volume 7, pp.701, 1964.

14. Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, Volume 34, Number 3, pp. 596-615, 1987.

15. C.A.R. Hoare. Algorithm 63 (Partition) and Algorithm 65 (find). *Communications of the ACM*, Volume 4, Number 7, pp 321-322, 1961.

16. C.A.R. Hoare. Quicksort. *The Computer Journal*, Volume 5, Number 1, pp.10-15, 1962.

17. John E Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, Volume 16, Number 6 pp.372-378, 1973.

18. David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* Volume 40, Number 9, pp. 1098-1101, 1952.

19. Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*. pages 85-103, Plenum Press, 1972.

20. J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, Volume 7, pp 48-50, 1956.

21. R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, Volume 36, pp.1389-1401, 1957.

22. William Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, Volume 33, Number 6, pp. 668-676, 1990.

23. Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, Volume 21, Number 10, pp.847-857, 1978.

24. Daniel D Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, Volume 32, Number 3, pp 652-686, 1985.

25. Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, Volume 1, Number 2, pp.146-160, 1972.

26. Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, Volume 6, Number 2, pp.306-318, 1985.

27. Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, Volume 21, Number 4, pp.309-315, 1978.

28. Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, Volume 9, Number 1, pp.11-12, 1962.

29. J.W.J. Williams. Algorithm 232 (Heapsort). *Communications of the ACM*, Volume 7, pp.347-348, 1964.