KARNATAKA STATE OPEN UNIVERSITY

MANASAGANGOTRI, MYSORE- 570 006

DEPERTAMENT OF STUDIES IN INFORMATION TECHNOLOGY

SCIENCE

M.SC IN INFORMATION

I SEMESTER

ALGORITHMS & DATA STRUCTURES

DATA STRUCTURES AND ALGORITHMS

IS 1.4

KSOU NATIONAL INTERNATIONAL RECOGNITION

Karnataka State Open University (KSOU) was established on 1st June 1996 with the assent of H.E. Governor of Karnataka as a full fledged University in the Academic year 1996 vide Government notification No./EDI/UOV/dated 12th February 1996 (Karnataka State Open University Act – 1992). The Act was promulgated with the object to incorporate an Open University at the State Level for the introduction and promotion of Open University and Distance Education Systems in the education pattern of the State and the Country for the Co-ordination and determination of standard of such systems.

- With the virtue of KSOU Act of 1992, Karnataka State Open University is empowered to establish, maintain or recognize Institutions, Colleges, Regional Centres and Study Centres at such places in Karnataka and also open outside Karnataka at such places as it deems fit.
- All Academic Programmes offered by Karnataka State Open University are recognized by the Distance Education Council (DEC), Ministry of Human Resource Development (MHRD), New Delhi.
- Karnataka State Open University is a regular member of the Association of Indian Universities (AIU), New Delhi, since 1999.
- Karnataka State Open University is a permanent member of Association of Commonwealth Universities (ACU), London, United Kingdom since 1999. Its member code number: ZKASOPENUINI.
- Karnataka State Open University is a permanent member of Asian Association of Open Universities (AAOU), Beijing, CHINA, since 1999.
- Karnataka State Open University has association with Commonwealth of Learning (COL), Vancouver, CANADA, since 2003. COL is an intergovernmental organization created by commonwealth Heads of Government to encourage the development and sharing of open learning distance education knowledge, resources and technologies.

Higher Education To Everyone Everywhere

MSIT – 104

DATA STRUCTURE

&

ALGORITHEMS

Preface

Design of efficient algorithms is very much essential to solve problems efficiently using computers. Since data structures enhance the performance of algorithms, selection of appropriate data structures is critical importance to design efficient algorithms. In other words, good algorithm design must go hand in hand with appropriate data structures for efficient program design to solve a problem irrespective of the discipline or application.

This material is designed to give an overview of the fundamentals of data structures, their complexities and importance in solving problems. The concept of data structures, classification of data structures and their applications are dealt independently without considering any programming language of interest. The whole material is organized into four modules each with four units. Each unit lists the objectives of the study along with the relevant questions, illustrations and suggested reading to better understand the concepts.

Module-1 introduces algorithms, properties of algorithm, notion for programs and some sample programs. Time complexity, space complexity, asymptotic notation, practical complexities, and performance measurement of simple algorithms are also discussed in this module.

Module-2 introduces data structure, primitive data structure, stacks and its application with some examples. Recursion, Queue, a brief introduction to Linked Lists and Some general linked list operations and types like Singly Linked Lists and Circular and Doubly linked lists. Introduction to sorting merge sort, quick sort and selection sort, Binary Search based Insertion Sort, Heap sort, Bucket sort and Searching techniques.

Module-3 introduces the divide-and-conquer strategy of solving a problem. The general structure of divide and conquer strategy and its applicability to solve max-min problem, container loading, 0/1Knapsack problem, minimum cost spanning tree and recurrence relations have been illustrated. The divide-and-conquer strategy to sort a list using merge sort, quick sort techniques and to search a list using binary search technique has also been presented. Complexity of merge sort, quick sort and binary search algorithms are evaluated.

Module-4 introduces graph and tree data structures. Representation of graphs and trees based on sequential and linked allocation and associated algorithms are presented. Traversal of binary trees, operation on binary trees and associated algorithms are discussed. Threaded binary trees and their traversal, representation of forest of trees, traversal of forest, conversion of forest to binary trees and associated algorithms are also discussed in this module.

All the best for the students.

Prof D S Guru & H S Nagendraswamy

Course Design and Editorial Committee	
Prof. K. S. Rangappa Vice Chancellor & Chairperson Karnataka State Open University Manasagangotri, Mysore – 570 006	**Prof. Vikram Raj Urs** Dean (Academic) & Convener Karnataka State Open University Manasagangotri, Mysore – 570 006
Head of the Department – Incharge	**Course Co-Ordinator**
Prof. Kamalesh DIRECTOR IT&Technology Karnataka State Open University Manasagangotri, Mysore – 570 006	**Mr. Mahesha DM** Lecturer, DOS in Information Technology Karnataka State Open University Manasagangotri, Mysore – 570 006

Course Writers		
Dr. D S Guru Associate Professor DOS in Computer Science University of Mysore Manasagangotri, Mysore – 570 006	**Modules 1 and 3**	**Units 1-4 and 9-12**
And		
Dr.H S Nagendraswamy Associate Professor DOS in Computer Science University of Mysore Manasagangotri, Mysore – 570 006	**Modules 2 and 4**	**Units 5-8 and 13-16**

Publisher

Registrar
Karnataka State Open University
Manasagangotri, Mysore – 570 006

Developed by Academic Section, KSOU, Mysore
Karnataka State Open University, 2012

Printed and Published on behalf of Karnataka State Open University, Mysore-6 by the **Registrar (Administration)**

UNIT – 1

INTRODUCTION TO ALGORITHM, PROPERTIES OF ALGORITHM, NOTATION FOR PROGRAMS, SOME SIMPLE PROGRAMS

STRUCTURE

1.0 OBJECTIVES

After studying this unit you should be able to

- Define an algorithm and its characteristics.
- Properties of algorithms.
- Transform an algorithm into a program.

1.1 INTRODUCTION TO ALGORITHM

Computer Science is the field where we study about how to solve a problem effectively and efficiently with the aid of computers. Solving a problem that too by the use of computers requires a thorough knowledge and understanding of the problem. The problem could be of any complex ranging from a simple problem of adding two numbers to a problem of making the computer capable of taking decisions on time in real environment, automatically by understanding the situation or environment, as if it is taken by a human being. In order to automate the task of solving a problem, one has to think of many ways of arriving at the solution. A way of arriving at a solution from the problem domain is called algorithm. Thus, one can have many algorithms for the same problem.

In case of existence of many algorithms we have to select the one which best suits our requirements through analysis of algorithms. Indeed, the design and analysis of algorithms are the two major interesting sub fields of computer science. Most of the scientists do work on these subfields just for fun. We mean to say that these two sub areas of computer science are such interesting areas. Once the most efficient algorithm is selected, it gets coded in a programming language. This essentially requires knowledge of a programming language. And finally, we go for executing the coded algorithm on a machine (Computer) of particular architecture. Thus, the field computer science broadly encompasses,

- Study on design of algorithms.
- Study on analysis of algorithms.
- Study of programming languages for coding algorithms.

- Study of machine architecture for executing algorithms.

It shall be noticed that the fundamental notion in all the above is the term algorithm. Indeed, that signifies its prominence of algorithms, in the field of computer science and thus the algorithm deserves its complete definition. Algorithm means 'process or rules for computer calculation' according to the dictionary. However, it is something beyond that definition.

WHAT IS AN ALGORITHM?

An algorithm, named for the ninth-century Persian mathematician al-KhowArizmi, is simply a set of rules for carrying out some calculation, either by hand or, more usually, on a machine. However, other systematic methods for calculating a result could be included. The methods we learn at school for adding, multiplying and dividing numbers are algorithms, for instance. The most famous algorithm in history dates from well before the time of the ancient Greeks: this is Euclid's algorithm for calculating the greatest common divisor of two integers. The execution of an algorithm must not normally involve any subjective decisions, nor must it call for the use of intuition or creativity. Hence a cooking recipe can be considered to be an algorithm if it describes precisely how to make a certain dish, giving exact quantities to use and detailed instructions for how long to cook it. On the other hand, if it includes such vague notions as "add salt to taste" or "cook until tender" then we would no longer call it an algorithm.
When we use an algorithm to calculate the answer to a particular problem, we usually assume that the rules will, if applied correctly, indeed give us the correct answer. A set of rules that calculate that 23 times 51 is 1170 is not generally useful in practice. However in some circumstances such *approximate algorithms* can be useful. If we want to calculate the square root of 2, for instance, no algorithm can give us an exact answer in decimal notation, since the representation of square root of 2 is infinitely long and nonrepeating. In this case, we shall be content if an algorithm can give us an answer that is as precise as we choose: 4 figures accuracy, or 10 figures, or whatever we want.

Formally, an algorithm is defined to be a sequence of steps, which if followed, accomplishes a particular task. In addition, every algorithm must satisfy the following criteria.

- Consumes zero or more inputs

- Produces at least one output
- Definiteness
- Finiteness
- Effectiveness

The definiteness property insists that each step in the algorithm is unambiguous. A step is said to be unambiguous if it is clear, in the sense that, the action specified by the step can be performed without any dilemma/confusion. The finiteness property states that the algorithm must terminate its execution after finite number of steps (or after a finite time period of execution). That is to say, it should not get into an endless process. The effectiveness property is indeed the most important property of any algorithm. There could be many algorithms for a given problem. But, one algorithm may be effective with respect to a set of constraints and the others may not be effective with respect to the same constraints. However, they may be more effective with respect to some other constraints. In fact, because of the effectiveness property associated with algorithms, finding out most optimal/effective algorithm even for already solved problem is still open for the research community for further research.

To study the effectiveness of an algorithm, we have to find out the minimum and maximum number of operations the algorithm takes to solve the desired problem. The time requirement and space requirement profiling should be done. The profiling may be relatively based on the number of inputs or the number of outputs or on the nature of input. The process of knowing about the minimum cost and the maximum cost (may be in terms of CPU time and memory locations) is called analysis of algorithm. In the subsequent sections, we present methods of analyzing an algorithm.

1.2 PROPERTIES OF ALGORITHM

Any algorithm must possess the following properties:

a) **Finiteness**: The algorithm must terminate with in finite time (a finite number of steps).

b) **Definiteness**: Steps enumerated must be precise and unambiguous.

c) **Effectiveness**: Each step must be easily convertible to a code.

d) **Input**: Algorithm must take zero or more input.

e) **Output**: Algorithm must produce at least one output.

1.3 NOTATION FOR PROGRAMS

It is important to decide how we are going to *describe* our algorithms. If we try to explain them in English, we rapidly discover that natural languages are not at all suited to this kind of thing. To avoid confusion, we shall in future specify our algorithms by giving a corresponding *program.* We assume that the reader is familiar with at least one well-structured programming language such as Pascal. However, we shall not confine ourselves strictly to any particular programming language: in this way, the essential points of an algorithm will not be obscured by relatively unimportant programming details, and it does not really matter which well-structured language the reader prefers.

A few aspects of our notation for programs deserve special attention. We use phrases in English in our programs whenever this makes for simplicity and clarity. Similarly, we use mathematical language, such as that of algebra and set theory, whenever appropriate-including symbols such as and Li introduced in Section 1.4.7. As a consequence, a single "instruction" in our programs may have to be translated into several instructions-perhaps a **while** loop-if the algorithm is to be implemented in a conventional programming language. Therefore, you should not expect to be able to run the algorithms we give directly: you will always be obliged to make the necessary effort to transcribe them into a "real" programming language. Nevertheless, this approach best serves our primary purpose, to present as clearly as possible the basic concepts underlying our algorithms.

To simplify our programs further, we usually omit declarations of scalar quantities (integer, real, or Boolean). In cases where it matters-as in recursive **functions** and **procedures-all** variables used are implicitly understood to be *local* variables, unless the context makes it clear otherwise. In the same spirit of simplification, proliferation of **begin** and **end** statements, that plague programs written in Pascal, is avoided: the range of statements such as **if, while,** or **for,** as well as that of declarations such as **procedure, function,** or **record,** is shown by

indenting the statements affected. The statement **return** marks the dynamic end of a **procedure** or a **function,** and in the latter case it also supplies the value of the **function.**

We do not declare the type of parameters in **procedures** and **functions,** nor the type of the result returned by a **function,** unless such declarations make the algorithm easier to understand. Scalar parameters are passed by value, which means they are treated as local variables within the **procedure** or **function,** unless they are declared to be **var**parameters, in which case they can be used to return a value to the calling program. In contrast, **array** parameters are passed by reference, which means that any modifications made within the **procedure** or **function** are reflected in the array actually passed in the calling statement.

Finally, we assume that the reader is familiar with the concepts of recursion, **record,** and pointer. The last two are denoted exactly as in Pascal, except for the omission of **begin** and **end** in **records.** In particular, pointers are denoted by the symbol " ↑ ".

To wrap up this section, here is a program for multiplication.Here ÷ denotes integer division: any fraction in the answer is discarded. We can compare this program to the informal English description of the same algorithm.

```
function Multiply(m, n)
    result ← 0
    repeat
        if m is odd then result ← result + n
        m ← m ÷ 2
        m ← n + n
    until m = 1
return result
```

1.4 SOME SIMPLE EXAMPLES

In order to understand the process of designing an algorithm for a problem, let us consider the problem of verifying if a given number n is a prime as an example. The possible ways to solve this problem are as follows:

Example 1.1

Algorithm 1: Check for prime

Input: A number (n).

Output: Number n is a prime or not.

Method: For all k = 2, 3, 4, …, (n-1)

If (n mod k <> 0) then n is a prime number

else it is not a prime number.

Algorithm ends

Algorithm 2: Check for prime

Input: A number (n).

Output: Number n is a prime or not.

Method: For all k = 2, 3, 4, …, (n/2)

If (n mod k <> 0) then n is a prime number

else it is not a prime number.

Algorithm ends

Algorithm 3: Check for prime

Input: A number (n).

Output: Number n is a prime or not.

Method: For all k = 2, 3, 4, …, (square-root(n))

If (n mod k <> 0) then n is a prime number

else it is not a prime number.

Algorithm ends

From the above examples, it is understood that there are three different ways to solve a problem. The first method divides the given number *n* by all numbers ranging from 2 to (*n*-1) to decide if it is a prime. Similarly, the second method divides the given number *n* by only those numbers from 2 to (*n*/2) and the third method divides the number *n* by only those from 2 upto square root of *n*. This example helped us in understanding that a problem can be solved in different ways, but any one which appears to be more suitable has to be selected.

If we analyze the above algorithms for a number $n = 100$, the first method takes 98 iterations, the second method takes 49 iterations, and the third method takes only 10 iterations. This shows that the third algorithm is the better one to solve this problem.

Let us consider another problem; searching for a telephone number of a person in the telephone directory. If the telephone numbers are not sequenced in an order, then the searching algorithm has to scan the entire telephone directory one by one till it finds the desired number. In the worst case, the element to be searched may happen to be the last number in the directory, and in such case, the searching algorithm will take as many comparisons as the number of telephone numbers in the directory. The number of comparisons grows linearly as size of the directory increases. This will become a bottle neck if the number of entries in the directory is considerably large. On the other hand if the telephone numbers are arranged in some sequence using an appropriate storage structure then we can devise an algorithm, which takes less time to search the required number. Thus the performance of an algorithm mainly depends on the data structure, which preserves the relationship among the data through an appropriate storage structure.

Friends, we now feel it is a time to understand a bit about the properties of an algorithm.

1.6 SUMMARY

In this unit, we have introduced algorithms. A glimpse of all the phases we should go through when we study an algorithm and its variations was given. In the study of algorithms, the process of designing algorithms, validating algorithms, analyzing algorithms, coding the designed algorithms, verifying, debugging and studying the time involved for execution were presented.

1.7 KEYWORDS

1) Algorithm
2) Properties
3) Asymptotic notation

1.8 QUESTIONS FOR SELF STUDY

1) What is an algorithm? Explain its characteristics?
2) Explain the properties of algorithm?
3) What is meant by notation of programs? Explain.
4) Give some examples on algorithm

1.9 EXCERCISES

1) Design algorithms for the following
   a) To test whether the three numbers represent the sides of a right angle triangle.
   b) To test whether a given point p(x, y) lies on x-axis or y-axis or in I/II/III/IV quadrant.
   c) To compute the area of a circle of a given circumference
   d) To locate a specific word in a dictionary.
   e) To find the first n prime numbers.
   f) To add two matrices.
   g) To multiply two matrices.
   h) To search an element for its presence/absence in a given list of random data elementswithout sorting the list.
   i) To find the minimum and maximum in the given list of data elements without sorting the list.

1.10 REFERENCES

1) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.
2) SartajSahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.
3) Goodman AndHedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

UNIT- 2

SPACE COMPLEXITY, TIME COMPLEXITY, ASYMPTOTIC NOTATION, PRACTICAL COMPLEXITIES, PERFORMANCE MEASUREMENT OF SIMPLE ALGORITHM

STRUCTURE

2.0 OBJECTIVES

After studying this unit you should be able to

- Explain how test algorithms.
- Measure the performance of an algorithm.
- Analyze the space complexity and time complexity of an algorithm.
- Give asymptotic notations for algorithms.

2.1INTRODUCTION

The number of (machine) instructions which a program executes during its running time is called its **time complexity** in computer science. This number depends primarily on the size of the program's input, that is approximately on the number of the strings to be sorted (and their length) and the algorithm used. So approximately, the time complexity of the program "sort an array of n strings by minimum search" is described by the expression $c \cdot n^2$. c is a constant which depends on the programming language used, on the quality of the compiler or interpreter, on the CPU, on the size of the main memory and the access time to it, on the knowledge of the programmer, and last but not least on the algorithm itself, which may require simple but also time consuming machine instructions. (For the sake of simplicity we have drawn the factor 1/2 into c here.) So while one can make c smaller by improvement of external circumstances (and thereby often investing a lot of money), the term n^2, however, always remains unchanged. Performance measurement is the process of executing a correct program on different data sets to measure the time and space that it takes to compute the results. Complexity of a program is generally some function of the instance characteristics.

2.2 SPACE COMPLEXITY

The better the time complexity of an algorithm is, the faster the algorithm will carry out his work in practice. Apart from time complexity, its **space complexity** is also important: This is

essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible, too.

There is often a **time-space-tradeoff** involved in a problem, that is, it cannot be solved with few computing time *and* low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

2.3 TIME COMPLEXITY

The number of (machine) instructions which a program executes during its running time is called its **time complexity** in computer science. This number depends primarily on the size of the program's input, that is approximately on the number of the strings to be sorted (and their length) and the algorithm used. So approximately, the time complexity of the program "sort an array of n strings by minimum selection" is described by the expression $c \cdot n^2$. c is a constant which depends on the programming language used, on the quality of the compiler or interpreter, on the CPU, on the size of the main memory and the access time to it, on the knowledge of the programmer, and last but not least on the algorithm itself, which may require simple but also time consuming machine instructions. (For the sake of simplicity we have drawn the factor 1/2 into c here.) So while one can make c smaller by improvement of external circumstances (and thereby often investing a lot of money), the term n^2, however, always remains unchanged.

2.4 ASYMPTOTIC NOTATIONS

Asymptotic notation is a method of expressing the order of magnitude of an algorithm during the a priori analysis. These order notations do not take into account all program and machine dependent factors i.e., given an algorithm, if it is realized and executed with the aid of different programming languages, then it is obvious to find different performance response for the same algorithm. In addition, if the same program is run on different computers, although the machine speeds are same, their performances may differ. But, the a priori analysis will not have these variations. There are several kinds of mathematical notations that are used in asymptotic representations.

Definition: f(n) = O(g(n)) (read as "f of n equals big oh of g of n"), if and only if there exist two positive, integer constants c and n_0 such that

$ABS(f(n)) \leq C*ABS(g(n))$ for all $n \geq n_0$

In other words, suppose we are determining the computing time, f(n) of some algorithm where n may be the number of inputs to the algorithm, or the number of outputs, or their sum or any other relevant parameter. Since f(n) is machine dependent (it depends on which computer we are working on). An a priori analysis cannot determine f(n), the actual complexity as described earlier. However, it can determine a g(n) such that f(n)=O(g(n)). An algorithm is said to have a computing time O(g(n)) (of the order of g(n)), if the resulting times of running the algorithm on some computer with the same type of data but for increasing values of n, will always be less than some constant times |g(n)|. We use some polynomial of n, which acts as an upper limit, and we can be sure that the algorithm does not take more than the time prescribed by the upper limit. For instance, let us consider the following algorithm,

```
Algorithm: Sum
Input: n. number of values to be added
       A. array of n elements
Output: S. sum of n elements in A
Method :
      (1)   S=0:
      (2)   For i= 1 to n do
      (3)   S= S + A[ i ]:
            For end
      (4)   Output S:
Algorithm ends.
```

In the above algorithm, statement (1) is executed 1 time, statement (2) is executed n+1 times, statement (3) is executed n times, and statement (4) is executed 1 time. Thus, the total time taken is 2n+3.

In order to represent the time complexity of the above algorithm as f(n)=O(n), it is required to find the integer constants c and n0, which satisfy the above definition of O notation. i.e., an algorithm with the time complexity 2n + 3 obtained from a priori analysis can be represented as O(n) because $2n + 3 \leq 3n$ for all $n \geq 3$ here c=3 and $n_0 = 3$.

Some more examples:

The function $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$.

$3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$.

$10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$.

$6 * 2^n + n^2 = O(n^2)$ as $6 * 2^n + n^2 \leq 7 * 2^n$ for all $n \geq 4$.

The most commonly encountered complexities are $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ and $O(2^n)$. Algorithms of higher powers of n are seldom solvable by simple methods. $O(1)$ means a computing time that is constant. $O(n)$ is called linear, $O(n^2)$ is called quadratic, $O(n^3)$ is called cubic and $O(2^n)$ is called exponential. The commonly used complexities can thus be arranged in an increasing order of complexity as follows.

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

If we substitute different values of n and plot the growth of these functions, it becomes obvious that at lower values of n, there is not much difference between them. But as n increases, the values of the higher powers grow much faster than the lower ones and hence the difference increases. For example at $n = 2, 3, 4, \ldots, 9$ the values of 2^n happens to be lesser than n^3 but once $n \geq 10$, 2^n shows a drastic growth.

The O - notation discussed so far is the most popular of the asymptotic notations and is used to define the upper bound of the performance of an algorithm also referred to as the worst case performance of an algorithm. But it is not the only complexity we have. Sometimes, we may wish to determine the lower bound of an algorithm i.e., the least value, the complexity of an algorithm can take. This is denoted by Ω (omega).

Definition: $f(n) = \Omega(g(n))$ (read as "f of n equals omega of g of n) if and only if there exist positive non-zero constants C and n_0, such that for all $ABS(f(n)) \geq C*ABS(g(n))$ for all $n \geq n_0$.

Some Examples: The function $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$.

$3n + 3 = \Omega(n)$ as $3n + 3 \geq 3n$ for $n \geq 1$.

$10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n^3 \geq 1$.

$10n^2 + 4n + 2 = \Omega(n)$ as $10n^2 + 4n + 2 \geq n$ for $n \geq 1$.

$10n^2 + 4n + 2 = \Omega(1)$ as $10n^2 + 4n + 2 \geq 1$ for $n \geq 1$.

In some cases both the upper and lower bounds of an algorithm can be the same. Such a situation is described by the θ-notation.

Definition: $f(n) = \theta(g(n))$ if and only if there exist positive constants C_1, C_2 and n_0 such that for all $n > n_0$, $C_1 |g(n)| \leq f(n) \leq C_2 |g(n)|$

Some Examples: $3n + 2 = \theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$ and $3n + 2 \leq 4n$ for all $n \geq 2$, so $C_1 = 3$ and $C_2 = 4$ and $n_0 = 2$.

2.5 PRACTICAL COMPLEXITIES

We have seen that the time complexity of a program is generally some function of the instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change. We can also use the complexity function to compare two programs P and Q that perform the same task. Assume that program P has complexity $\theta(n)$ and that program Q has complexity $\theta(n^2)$. We can assert that, program P is faster than program Q is for sufficiently largen. To see the validity of this assertion, observe that the actual computing timeof P is bounded from above by cn for some constant c and for all n n>nl, while that of Q is bounded from below by dn2 for some constant d and all n, $n \geq n_2$. Since$cn \leq dn^2$ for $n \geq c/d$, program P is faster than program Q whenever $n \geq \max\{n_1, n_2, c/d\}$.

One should always be cautiously aware of the presence of the phrase *sufficiently large* in the as assertion of the preceding discussion. When deciding which of the two programs to use, we must know whether the n we are dealing with is, in fact, sufficiently large. If program P actually runs in $10^6 n$ milliseconds while program Q runs in n^2 milliseconds and if we always have $n \leq 10^6$, then program Q is the one to use.

To get a feel for how the various functions grow with n, you should study figures 2.1 and 2.2. These figures show that 2^n grows very rapidly with n. In fact, if a program needs 2^n steps for execution, then when n = 40, the number of steps needed is approximately $1.1 * 10^{12}$. On a computer performing 1.000.000,000 steps per second, this program would require about 18.3 minutes. If n = 50, the same program would

run for about 13 days on this computer. When n = 60, about 310.56 years will be required to execute the program, and when n = 100, about $4*10^{13}$ years will be needed. We can conclude that the utility of programs with exponential complexity is limited to small n (typically n < 40).

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Figure 2.1 values of various functions

Programs that have a complexity that is a high-degree polynomial are also of limited utility. For example, if a program needs n^{10} steps, then our 1.000,000,000 steps per second computer needs 10 seconds when n = 10; 3171 years when n = 100; and 3.17 + 1013 years when n = 1000. If the program's complexity had been n^3 steps instead, then the computer would need 1 second when n = 1000, 110.67 minutes when n = 10,000 and 11.57 days when n = 100,000.

Figure 2.2 Plot of various functions

Figure 2.2 above gives the time that a 1,000,000,000 instructions per second computer needs to execute a program of complexity f (n) instructions. One should note that currently only the fastest computers can execute about 1.000,000,000 instructions per second. From a practical standpoint, it is evident that for reasonably large n only programs of small complexity are feasible.

2.6 Performance Measurement

Performance measurement is the process of executing a correct program on different data sets to measure the time and space that it takes to compute the results. Complexity of a program is generally some function of the instance characteristics.

The ultimate test is performed to ensure that the program developed on the basis of the designed algorithm, runs satisfactorily. Testing a program involves two phases viz., debugging and profiling. Debugging is the process of executing a program with sample datasets to determine if the results obtained are satisfactory. When unsatisfactory results are generated, suitable changes are to be incorporated in the program to get the desired results. However, it is pointed out that "debugging can only indicate the presence of errors but not their absence" i.e., a program that yields unsatisfactory results on a sample data set is definitely faulty, but on the other hand a program producing the desirable results on one/more data sets need not be correct. In order to actually prove that a program is perfect, a process of "proving" is necessary wherein the program is analytically proved to be correct and in such cases, it is bound to yield perfect results for all possible sets of data.On the other hand, profiling or performance measurement is the process of executing a correct program on different data sets to measure the time and space that it takes to compute the results. that several different programs may do a given job satisfactorily. But often, especially when large data sets are being operated upon, the amount of space and the computation time required become important. In fact, a major portion of the study of algorithms pertains to the study of time and space requirements of algorithms. The following section discusses the actual way of measuring the performance of an algorithm.

This is the final stage of algorithm evaluation. A question to be answered when the program is ready for execution, (after the algorithm has been devised, made a priori analysis of that, coded into a program debugged and compiled) is how do we actually evaluate the time taken by the program? Obviously, the time required to read the input data or give the output should not be taken into account. If somebody is keying in the input data through the keyboard or if data is being read from an input device, the speed of operation is dependent on the speed of the device, but not on the speed of the algorithm. So, we have to exclude that time while evaluating the programs. Similarly, the time to write out the output to any device should also be excluded. Almost all systems provide a facility to measure the elapsed system time by using stime() or other similar functions. These can be inserted at appropriate places in the program and they act as stop clock measurement. For example, the system time can be noted down just after all the inputs have been read. Another reading can be taken just before the output operations start. The difference between the two readings is the actual time of run of the program. If multiple inputs and outputs are there, the counting operations should be included at suitable places to exclude the I/O operations.It is not enough if this is done for one data set. Normally various data sets are chosen and the performance is measured as explained above. A plot of data size n v/s the actual time can be drawn which gives an insight into the performance of the algorithm.

The entire procedure explained above is called "profiling". However, unfortunately, the times provided by the system clock are not always dependable. Most often, they are only indicative in nature and should not be taken as an accurate measurement. Especially when the time durations involved are of the order of 1-2 milliseconds, the figures tend to vary often between one run and the other, even with the same program and all same input values.

Irrespective of what we have seen here and in the subsequent discussions, devising algorithms is both an art and a science. As a science part, one can study certain standard methods (as we do in this course) but there is also an individual style of programming which comes only by practice.

2.7 SUMMARY

In this unit, we have introduced algorithms. A glimpse of all the phases we should go through when we study an algorithm and its variations was given. In the study of algorithms, the process of designing algorithms, validating algorithms, analyzing algorithms, coding the designed algorithms, verifying, debugging and studying the time involved for execution were presented. All in all, the basic idea behind the analysis of algorithms is given in this unit.

2.8 KEYWORDS

1) Algorithm
2) Space complexity
3) Time complexity
4) Asymptotic notation
5) Profiling

2.9 QUESTIONS FOR SELF STUDY

1) What is an algorithm? Explain its characteristics?
2) How to test algorithms? Why is it needed?
3) Explain Practical complexities of algorithms?
4) What is meant by profiling? Explain.
5) What is meant by space complexity and time complexity of an algorithm? Explain.
6) What is meant by asymptotic notation? Explain.

2.10 EXCERCISES

1) How do we actually evaluate the time taken by the program?
2) Discuss about the Practical complexities considering few example functions.
3) Design algorithms for the following and determine the frequency counts for all statements in thedevised algorithms and express their complexities with the help of asymptotic notations.
4) To test whether the three numbers represent the sides of a right angle triangle.

5) To test whether a given point p(x, y) lies on x-axis or y-axis or in I/II/III/IV quadrant.
7) To compute the area of a circle of a given circumference
8) To locate a specific word in a dictionary.
9) To find the first n prime numbers.
10) To add two matrices.
11) To multiply two matrices.
12) To search an element for its presence/absence in a given list of random data elementswithout sorting the list.
13) To find the minimum and maximum in the given list of data elements without sorting the list.

2.11 REFERENCES

1) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.
2) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.
3) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

UNIT- 3

ANALYZING CONTROL STRUCTURES, USING A BAROMETER, SUPPLEMENTARY EXAMPLES, AVERAGE CASE ANALYSIS, SLOVING RECURRENCES

STRUCTURE

3.0 OBJECTIVES

After studying this unit you should be able to

- Analyze Control Structures.
- Analyze algorithms using Barometer.
- Evaluate recursive algorithms.
- Analyzeaverage case analysis of algorithms.
- Determine the time complexity of algorithms using amortization scheme.
- Solve certain class of recurrences using characteristic equation.

3.1 INTRODUCTION

When you have several different algorithms to solve the same problem, you have to decide which one is best suited for your application. An essential tool for this purpose is the *analysis of algorithms*. Only after you have determined the efficiency of the various algorithms you will be able to make a well-informed decision. But there is no magic formula for analyzing the efficiency of algorithms. It is largely a matter of judgement, intuition and experience. Nevertheless, there are some basic techniques that are often useful, such as knowing how to deal with control structures and recurrence equations. This unit covers the most commonly used techniques and illustrates them with examples.

3.2 ANALYZING CONTROL STRUCTURES

The analysis of algorithms usually proceeds from the inside out. First, we determine the time required by individual instructions (this time is often bounded by a constant); then we combine these times according to the control structures that combine the instructions in the program. Some control structures such as sequencing - putting one instruction after another - are easy to analyze whereas others such as while loops are more subtle. In this unit, we give general

principles that are useful in analyses involving the most frequently encountered control structures, as well as examples of the application of these principles.

Sequencing

Let P_1 and P_2 be two fragments of an algorithm. They may be single instructions or complicated sub algorithms. Let $t_1$and t_2 be the times taken by $P_1$and P_2, respectively. These times may depend on various parameters, such as the instance size. The **sequencing rule** says that the time required to compute *"P_1; P_2"*, that is first $P_1$and then P_2, is simply $t_1 + t_2$. By the maximum rule, this time is in θ (max (t_1, t_2)). Despite its simplicity, applying this rule is sometimes less obvious than it may appear. For example, it could happen that one of the parameters that control t_2 depends on the result of the computation performed by P_1. Thus, the analysis of *" P_1; P_2"* cannot always be performed by considering P_1 and P_2 independently.

"For" loops

Forloops are the easiest loops to analyse. Consider the following loop.

for $i \leftarrow$ **1 to** m **do** $P(i)$

Here and throughout the book, we adopt the convention that when $m = 0$ this is not an error; it simply means that the controlled statement $P(i)$is not executed at all. Suppose this loop is part of a larger algorithm, working on an instance of size n. (Be careful not to confuse m and *n.)* The easiest case is when the time taken by $P(i)$does not actually depend on i, although it could depend on the instance size or, more generally, on the instance itself. Let t denote the time required to compute $P(i)$.In this case, the obvious analysis of the loop is that $P(i)$is performed m times, each time at a cost of t, and thus the total time required by the loop is simply $l = mt$. Although this approach is usually adequate, there is a potential pitfall: we did not take account of the time needed for *loop control.* After all, **for** loop is shorthand for something like the following **while** loop.

$i \leftarrow 1$

```
while i <m do
        P(i)
        i ← i + 1
```

In most situations, it is reasonable to count at unit cost the test $i < m$, the instructions $i \leftarrow 1$ and $i \leftarrow i + 1$, and the sequencing operations **(go to)** implicit in the **while** loop. Let c be an upper bound on the time required by each of these operations. The time l taken by the loop is thus bounded above by

$l \leq$	c	for $i \leftarrow 1$
	$+ (m + 1)c$	for the tests $i \leq m$
	$+ mt$	for the executions of $P(i)$
	$+ mc$	for the executions of $i \leftarrow i + 1$
	$+mc$	for the sequencing operations
$\leq$	$(t+3c)m +2c.$	

Moreover this time is clearly bounded below by mt. If c is negligible compared to t, our previous estimate that f is roughly equal to mt was therefore justified, except for one crucial case: $4l \approx mt$ is completely wrong when $m = 0$ (it is even worse if m is negative!).

Resist the temptation to say that the time taken by the loop is in $\theta(mt)$ on the pretext that the θ notation is only asked to be effective beyond some threshold such as $m > 1$. The problem with this argument is that if we are in fact analyzing the entire algorithm rather than simply the **for** loop, the threshold implied by the θ notation concerns n, the instance size, rather than m, the number of times we go round the loop, and $m = 0$ could happen for arbitrarily large values of n. On the other hand, provided t is bounded *below* by some constant (which is always the case in practice), and provided there exists a threshold no such that $m>1$ whenever $n \geq n_0$.

The analysis of for loops is more interesting when the time t(i) required for $P(i)$ varies as a function of i. (In general, the time required for $P(i)$ could depend not only on i but also on the instance size n or even on the instance itself.) If we neglect the time taken by the loop control, which is usually adequate provided $m>1$, the same **for** loop

for i ←1 to *m* **do** *P(i)*

takes a time given not by a multiplication but rather by a sum: it is $\sum_{i=1}^{m} t(i)$. We illustrate the analysis of **for** loops with a simple algorithm for computing the Fibonacci sequence as shown below.

function *Fibiter(n)*

i ← 1; *j*← 0

for $k \leftarrow 1$ **to** n **do** $j \leftarrow i + j$

$i \leftarrow j - i$

return j

If we count all arithmetic operations at unit cost, the instructions inside the **for** loop take constant time. Let the time taken by these instructions be bounded above by some constant c. Not taking loop control into account, the time taken by the for loop is bounded above by n times this constant: nc. Since the instructions before and after the loop take negligible time, we conclude that the algorithm takes a time in $O(n)$. Similar reasoning yields that this time is also in $O(n)$, hence it is in $O((n)$. We know that it is not reasonable to count the additions involved in the computation of the Fibonacci sequence at unit cost unless n is very small. Therefore, we should take account of the fact that an instruction as simple as "$j - i + j$" is increasingly expensive each time round the loop. It is easy to program long-integer additions and subtractions so that the time needed to add or subtract two integers is in the exact order of the number of figures in the larger operand. To determine the time taken by the k^{th} trip round the loop, we need to know the length of the integers involved. We can prove by mathematical induction that the values of i and j at the end of the k-th iteration are f_{k-1} and f_k, respectively. This is precisely why the algorithm works: it returns the value of j at the end of the n^{th} iteration, which is therefore f_n as required. Moreover, the Moivre's formula tells us that the size of f_k is in $\theta(k)$. Therefore, the k^{th} iteration takes a time $\theta(k - 1) + \theta(k)$, which is the same as $\theta(k)$. Let c be a constant such that this time is bounded above by ck for all $k > 1$. If we neglect the time required for the loop control and for the instructions before and after the loop, we conclude that the time taken by the algorithm is bounded above by

$$\sum_{k=1}^{n} ck = c \sum_{k=1}^{n} k = c \frac{n(n+1)}{2} \in O(n^2).$$

Similar reasoning yields that this time is in $\Omega(n^2)$, and therefore it is in $\theta(n^2)$. Thus it makes a crucial difference in the analysis of *Fibrec* whether or not we count arithmetic operations at unit cost.

The analysis of *for* loops that start at a value other than 1 or proceed by larger steps should be obvious at this point. Consider the following loop for example.

for $i \leftarrow 5$ **to** m **step** 2 **do** $P(i)$

Here, *P(i)* is executed ((m - 5) ÷ 2) + 1 times provided m ≥ 3. (For a **for** loop to make sense, the endpoint should always be at least as large as the starting point *minus* the step).

Recursive calls

The analysis of recursive algorithms is usually straightforward, at least up to a point. Simple inspection of the algorithm often gives rise to a recurrence equation that "mimics" the flow of control in the algorithm. Once the recurrence equation has been obtained, some general techniques can be applied to transform the equation into simpler nonrecursive asymptotic notation. As an example, consider the problem of computing the Fibonacci sequence with the recursive algorithm *Fibrec*.

function *Fibrec*(*n*)
  if *n* < 2 **then return** *n*
  else return *Fibrec*(*n*-1) + *Fibrec*(*n*-2)

Let *T(n)* be the time taken by a call on *Fibrec(n)*. If *n* < 2, the algorithm simply returns *n*, which takes some constant time *a*. Otherwise, most of the work is spent in the two recursive calls, which take time *T(n -1)* and *T(n* - 2), respectively. Moreover, one addition involving f_{n-1} and f_{n-2} (which are the values returned by the recursive calls) must be performed, as well as the control of the recursion and the test "if *n* <2". Let *h(n)* stand for the work involved in this addition and control, that is the time required by a call on *Fibrec (n)* ignoring the time spent inside the two recursive calls. By definition of *T(n)* and *h(n)*,we obtain the following recurrence.

$$T(n) = \begin{cases} a & \text{if } n = 0 \text{ or } n = 1 \\ T(n-1) + T(n-2) + h(n) & \text{otherwise} \end{cases}$$

If we count the additions at unit cost, *h(n)* is bounded by a constant and we conclude that *Fibrec(n)* takes a time exponential in *n*. This is *double* exponential in the size of the instance since the *value* of *n* is exponential in the *size* of *n*.

If we do not count the additions at unit cost, *h(n)* is no longer bounded by a constant. Instead *h(n)* is dominated by the time required for the addition of f_{n-1} and f_{n-2} for sufficiently large *n*. We know that this addition takes a time in the exact order of *n*. Therefore $h(n) \in \theta(n)$. Surprisingly, the result is the same regardless of whether *h (n)* is constant or linear: it is still the

case that $T(n) \in \theta(fn)$. In conclusion, *Fibrec(n)* takes a time exponential in n whether or not we count additions at unit cost! The only difference lies in the multiplicative constant hidden in the θ notation.

"While" and "repeat" loops

While and repeat loops are usually harder to analyze than for loops because there is no obvious a priori way to know how many times we shall have to go round the loop. The standard technique for analyzing these loops is to find a function of the variables involved whose value decreases each time around. To conclude that the loop will eventually terminate, it suffices to show that this value must be a positive integer. (You cannot keep decreasing an integer indefinitely.) To determine how many times the loop is repeated, however, we need to understand better how the value of this function decreases. An alternative approach to the analysis of while loops consist of treating them like recursive algorithms. The analysis of **repeat** loops is carried out similarly.

We shall study *binary search* algorithm, which illustrates perfectly the analysis of while loops. The purpose of binary search is to find an element x in an array $T[1 .. n]$ that is in nondecreasing order. Assume for simplicity that x is guaranteed to appear at least once in T. We require to find an integer i such that $1 < i < n$ and $T[i] = x$. The basic idea behind binary search is to compare x with the element y in the middle of T. The search is over if $x = y$; it can be confined to the upper half of the array if $x > y$; otherwise, it is sufficient to search the lower half. We obtain the following algorithm.

```
function Binary_Search(T[1..n], x)
  i ← 1; j ← n;
  while i < j do
    k ← (i + j) ÷ 2;
    case x < T[k]: j ← k - 1;
    case x = T[k]: i, j ← k {return k};
    case x > T[k]: i ← k + 1;
  return i
```

Recall that to analyze the running time of a **while** loop, we must find a function of the variables involved whose value decreases each time round the loop. In this case, it is natural to consider $j - i + 1$, which we shall call d. Thus d represents the number of elements of T still under

consideration. Initially, $d = n$. The loop terminates when $i \geq j$, which is equivalent to $d \leq 1$. Each time round the loop, there are three possibilities: either j is set to k -1, i is set to k + 1, or both i and j are set to k. Let d and $\hat{d}$ stand respectively for the value of j - i + 1 before and after the iteration under consideration. We use i, j, $\hat{\imath}$ and $\hat{\jmath}$ similarly. If $x<T[k]$, the instruction "$j \leftarrow k$ - 1" is executed and thus , $\hat{\imath} = i$ and $\hat{\jmath} = [(i + j) \div 2] - 1$. Therefore,

$$\hat{d} = \hat{j} - \hat{i} + 1 = j - (i - j) \div 2 \leq j - (i + j - 1) \quad 2 = d \quad 2$$

Similarly, if $x > T[k]$, the instruction "$i \leftarrow k + 1$" is executed and thus

$$\hat{\imath} = [(i + j) \div 2] + 1 \text{ and } \hat{\jmath} = j$$

$$\hat{d} = \hat{j} - \hat{i} + 1 = j - (i - j) \div 2 \leq j - (i + j - 1) \quad 2 = d \quad 2$$

Finally, if $x = T[k]$,then i and j are set to the same value and thus $\hat{d} = 1$; but d was at least 2 since otherwise the loop would not have been reentered. We conclude that $\hat{d} \leq d/2$ whichever case happens, which means that the value of d is at least halved each time round the loop. Since we stop when $d \leq 1$, the process must eventually stop, but how much time does it take?

To determine an upper bound on the running time of binary search, let d_l denote the value of j - i + 1 at the end of the l^{th} trip round the loop for $l \geq 1$ and let $d_o = n$. Since d_l -1is the value of j - i + l before starting the l^{th} iteration, we have proved that $d_l \leq d_{l-1}/2$ for all $l \geq 1$. It follows immediately by mathematical induction that $d_l \leq n / 2^l$. But the loop terminates when $d \leq 1$, which happens at the latest when $l = [\lg n]$. We conclude that the loop is entered at most $[\lg n]$ times. Since each trip round the loop takes constant time, binary search takes a time in $O(\log n)$. Similar reasoning yields a matching lower bound of $\Omega(\log n)$ in the worst case, and thus binary search takes a time in $\theta(\log n)$. This is true even though our algorithm can go much faster in the best case, when x is situated precisely in the middle of the array.

3.3 USING A BAROMETER

The analysis of many algorithms is significantly simplified when one instruction or one test-can be singled out as *barometer.* A barometer instruction is one that is executed at least as often as any other instruction in the algorithm. (There is no harm if some instructions are executed up to a constant number of times more often than the barometer since their contribution is absorbed in the asymptotic notation). Provided the time taken by each instruction is bounded by a constant, the time taken by the entire algorithm is in the exact order of the number of times that the barometer instruction is executed.

This is useful because it allows us to neglect the exact times taken by each instruction. In particular, it avoids the need to introduce constants such as those bounding the time taken by various elementary operations, which are meaningless since they depend on the implementation, and they are discarded when the final result is expressed in terms of asymptotic notation. For example, consider the analysis of *Fibiter* algorithm when we count all arithmetic operations at unit cost. We saw that the algorithm takes a time bounded above by cn for some meaningless constant c, and therefore that it takes a time in $\theta(n)$. It would have been simpler to say that the instruction $j \leftarrow i + j$ can be taken as barometer, that this instruction is obviously executed exactly n times, and therefore the algorithm takes a time in $\theta(n)$. Selection sorting will provide a more convincing example of the usefulness of barometer instructions in the next section.

When an algorithm involves several nested loops, any instruction of the innermost loop can usually be used as barometer. However, this should be done carefully because there are cases where it is necessary to take account of the implicit loop control. This happens typically when some of the loops are executed zero times, because such loops do take time even though they entail no executions of the barometer instruction. If this happens too often, the number of times the barometer instruction is executed can be dwarfed by the number of times empty loops are entered-and therefore it was an error to consider it as a barometer. Consider for instance pigeon-hole sorting. Here we generalize the algorithm to handle the case where the elements to be sorted are integers known to lie between 1 and s rather than between 1 and 10000. Recall that $T[1..n]$is the array to be sorted and $U[1..s]$is an array constructed so that $U[k]$gives the number of times

integer k appears in T. The final phase of the algorithm rebuilds T in nondecreasing order as follows from the information available in U.

```
i ← 0
for k ← 1 to s do
    while U[k] = 0 do
        i ← i + 1
        T[i] ← k
        U[k] ← U[k] - 1
```

To analyze the time required by this process, we use "$U[k]$" to denote the value *originally* stored in $U[k]$since all these values are set to 0 during the process. It is tempting to choose any of the instructions in the inner loop as a barometer. For each value of k, these instructions are executed $U[k]$times. The total number of times they are executed is therefore $\sum_{k=1}^{s} U[k]$. But this sum is equal to n, the number of integers to sort, since the sum of the number of times that each element appears gives the total number of elements. If indeed these instructions could serve as a barometer, we would conclude that this process takes a time in the exact order of n. A simple example is sufficient to convince us that this is not necessarily the case. Suppose $U[k] = 1$ when k is a perfect square and $U[k] = 0$ otherwise. This would correspond to sorting an array T containing exactly once each perfect square between 1 and n^2, using $s = n^2$ pigeon-holes. In this case, the process clearly takes a time in $\Omega(n^2)$ since the outer loop is executed s times. Therefore, it cannot be that the time taken is in $\theta(n)$. This proves that the choice of the instructions in the inner loop as a barometer was incorrect. The problem arises because we can only neglect the time spent initializing and controlling loops provided we make sure to include something even if the loop is executed zero times.

The correct and detailed analysis of the process is as follows. Let a be the time needed for the test $U[k] \neq 0$ each time round the inner loop and let b be the time taken by one execution of the instructions in the inner loop, including the implicit sequencing operation to go back to the test at the beginning of the loop. To execute the inner loop completely for a given value of k takes a time $t_k = (1 + U[k])\, a + U[k]\, b$, where we add 1 to $U[k]$before multiplying by a to take account of the fact that the test is performed each time round the loop *and* one more time to determine that the loop has been completed. The crucial thing is that this time is not zero even

when $U[k] = 0$. The complete process takes a time $c + \sum_{k=1}^{s} (d + t_k)$ where c and d are new constants to take account of the time needed to initialize and control the outer loop, respectively. When simplified, this expression yields $c + (a + d)\ s + (a + b)\ n$. We conclude that the process takes a time in $\theta(n + s)$. Thus the time depends on two independent parameters n and *s;* it cannot be expressed as a function of just one of them. It is easy to see that the initialization phase of pigeon-hole sorting also takes a time in $\theta\ (n + s)$, unless *virtual initialization* is used in which case a time in $\theta(n)$suffices for that phase. In any case, this sorting technique takes a time in $\theta(n + s)$ in total to sort n integers between 1 and s. If you prefer, the maximum rule can be invoked to state that this time is in $\theta(\max\ (n,\ s))$. Hence, pigeon-hole sorting is worthwhile but only provided s is small enough compared to n. For instance, if we are interested in the time required as a function only of the number of elements to sort, this technique succeeds in astonishing linear time if $s \in 0(n)$but it chugs along in quadratic time when $s \in \theta(n^2)$.

Despite the above, the use of a barometer is appropriate to analyze pigeon-hole sorting. Our problem was that we did not choose the proper barometer. Instead of the instructions *inside* the inner loop, we should have used the inner-loop *test* " $U\ [k] \neq 0$ " as a barometer. Indeed, no instructions in the process are executed more times than this test is performed, which is the definition of a barometer. It is easy to show that this test is performed exactly $n + s$ times, and therefore the correct conclusion about the running time of the process follows immediately without need to introduce meaningless constants.

In conclusion, the use of a barometer is a handy tool to simplify the analysis of many algorithms, but this technique should be used with care.

3.4 SUPPLEMENTARY EXAMPLES

In this section, we study several additional examples of analyses of algorithms involving loops, recursion, and the use of barometers.

Selection sort

Let's consider a selection sorting technique as shown below, which is a good example for the analysis of *nested* loops.

```
procedure select(T[1..n])
    for i ← 1 to n − 1 do
        minj ← i; minx ← T[i]
        for j ← i + 1 to n do
            if T[j] < minx then minj ← j
                                minx ← T[j]
        T[minj] ← T[i]
        T[i] ← minx
```

Although the time spent by each trip round the inner loop is not constant, it takes longer time when ***T[j]<minx*** and is bounded above by some constant **c** (that takes the loop control into account). For each value of i, the instructions in the inner loop are executed $n-(i+1)+1 = n-i$ times, and therefore the time taken by the inner loop is $t(i) \leq (n-i)\,c$. The time taken for the i-th trip round the outer loop is bounded above by $b + t(i)$ for an appropriate constant b that takes account of the elementary operations before and after the inner loop and of the loop control for the outer loop. Therefore, the total time spent by the algorithm is bounded above by

$$\begin{aligned}\sum_{i=1}^{n-1} b + (n-i)\,c &= \sum_{i=1}^{n-1} (b + cn) - c\sum_{i=1}^{n-1} i \\ &= (n-1)(b+cn) - cn(n-1)/2 \\ &= \frac{1}{2}cn^2 + \left(b - \frac{1}{2}c\right)n - b,\end{aligned}$$

which is in $o(n^2)$. Similar reasoning shows that this time is also in $\Omega(n^2)$ in all cases, and therefore selection sort takes a time in $\theta(n^2)$ to sort n items.

The above argument can be simplified, obviating the need to introduce explicit constants such as b and c, once we are comfortable with the notion of a barometer instruction. Here, it is natural to take the innermost test "if $T[j] < minx$" as a barometer and count the exact number of times it is executed. This is a good measure of the total running time of the algorithm because none of the loops can be executed zero times (in which case loop control could have been more time consuming than our barometer). The number of times that the test is executed is easily seen to be

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n - i)$$

$$= \sum_{k=1}^{n-1} k = n(n-1)/2.$$

Thus the number of times the barometer instruction is executed is in $\theta(n^2)$, which automatically gives the running time of the algorithm itself.

Insertion Sort

Let's consider one more sorting technique called Insertion Sort for analysis. The procedure for insertion sorting is as shown below.

```
procedure insert(T[1..n])
    for i ← 2 to n do
        x ← T[i]; j ← i − 1
        while j > 0 and x < T[j] do T[j + 1] ← T[j]
                                    j ← j − 1
        T[j + 1] ← x
```

Unlike selection sorting, the time taken to sort n items by insertion depends significantly on the original order of the elements. Here, we analyze this algorithm in the worst case. To analyze the running time of this algorithm, we choose as barometer the number of times the **while** loop condition ($j > 0$ and $x < T[j]$)istested.

Suppose for a moment that i is fixed. Let $x = T[i]$, as in the algorithm. The worst case arises when x is less than $T[j]$for every j between 1 and i - 1, since in this case we have to compare x to $T[i-1]$, $T[i-2]$,..., $T[1]$ before we leave the **while** loop because $j = 0$. Thus the **while** loop test is performed i times in the worst case. This worst case happens for every value of i from 2 to n when the array is initially sorted into descending order. The barometer test is thus performed $\sum_{i=2}^{n} i = n(n-1)/2 - 1$ times in total, which is in $\theta(n^2)$.This shows that insertion sort also takes a time in $\theta(n^2)$ to sort n items in the worst case.

3.5 AVERAGE-CASE ANALYSIS

We saw that insertion sort takes quadratic time in the *worst* case. On the other hand, it is easy to show that it succeeds in linear time in the *best* case. It is natural to wonder about its efficiency *on the average.* For the question to make sense, we must be precise about the meaning of "on the average". This requires us to assume an a priori probability distribution on the instances that our algorithm may be asked to solve. The conclusion of an average-case analysis may depend crucially on this assumption, and such analysis may be misleading if in fact our assumption turns out not to correspond with the reality of the application that uses the algorithm. Most of the time, average-case analyses are performed under the more or less realistic assumption that all instances of any given size are equally likely. For sorting problems, it is simpler to assume also that all the elements to be sorted are distinct.

Suppose we have n distinct elements to sort by insertion and all $n!$permutations of these elements are equally likely. To determine the time taken on average by the algorithm, we could add the times required to sort each of the possible permutations, and then divide by $n!$the answer thus obtained. An alternative approach, easier in this case, is to analyze directly the time required by the algorithm. For any i, $2 \leq i \leq n$, consider the subarray $T[1..i]$. The *partial rank* of $T[i]$ is defined as the position it would occupy if the subarray were sorted. For example, the partial rank of $T[4]$ in [3,6,2,5,1,7,4] is 3 because $T[1..4]$ once sorted is [2,3,5,6]. Clearly, the partial rank of $T[i]$ does not depend on the order of the elements in subarray $T[1..i-1]$. It is easy to show that if all $n!$permutations of $T[1..n]$are equally likely, then the partial rank of $T[i]$is equally likely to take any value between 1 and i, independently for all values of i.

Suppose now that i is fixed, $2 \leq i \leq n$, and that we are about to enter the **while** loop. Inspection of the algorithm shows that subarray $T[1..i-1]$contains the same elements as before the algorithm was called, although they are now in sorted order, and $T[i]$still has its original value since it has not yet been moved. Therefore, the partial rank of $T[i]$is equally likely to be any value between 1 and i. Let k be this partial rank. We choose again as barometer the number of times the **while** loop condition ($j>0$ and $x<T[j]$)is tested. By definition of partial rank, and since $T[1..i-1]$ is sorted, this test is performed exactly $i-k+1$ times. Because each value of k between 1

and i has probability $1/i$ of occurring, the average number of times the barometer test is performed for any given value of i is

$$c_i = \frac{1}{i}\sum_{k=1}^{i}(i-k+1) = \frac{i+1}{2}$$

These events are independent for different values of i. The total average number of times the barometer test is performed when sorting n items is therefore

$$\sum_{i=2}^{n} c_i \quad \sum_{i=2}^{n} \frac{i+1}{2} = \frac{(n-1)(n+4)}{4}$$

which is in $\theta(n^2)$. We conclude that insertion sorting makes on the average about half as many comparisons as in the worst case, but this number is still quadratic.

3.6 AMORTIZED ANALYSIS

Worst-case analysis is sometimes overly pessimistic. Consider for instance a process P that has side effects, which means that P modifies the value of global variables. As a result of side effects, two successive identical calls on P could take a substantially different amount of time. The easy thing to do when analyzing an algorithm that uses P as subalgorithm would be to analyze P in the worst case, and assume the worst happens each time P is called. This approach yields a correct answer assuming we are satisfied with an analysis in O notation but the answer could be pessimistic. Consider for instance the following loop.

for $i \leftarrow 1$ **to** n **do** P

If P takes a time in $\theta(\log n)$in the worst case, it is correct to conclude that the loop takes a time in $O(n \log n)$, but it may be that it is much faster *even in the worst case.* This could happen if Pcannot take a long time ($\Omega(\log n)$)unless it has been called many times previously, each time at small cost. It could be for instance that P takes constant time on the average, in which case the entire loop would be performed in linear time.

Rather than taking the average over all possible inputs, which requires an assumption on the probability distribution of instances, we take the average over *successive* calls. Here the times taken by the various calls are highly dependent. To prevent confusion, we shall say in this context that each call on P takes *amortized* constant time rather than saying that it takes constant time on the average.

Saying that a process takes amortized constant time means that there exists a constant c such that for any positive n and any sequence of n calls on the process, the total time for those calls is bounded above by cn.Therefore, excessive time is allowed for one call only if very short times have been registered previously, *not* merely if further calls would go quickly. Indeed, if a call were allowed to be expensive on the ground that it prepares for much quicker later calls, the expense would be wasted should that call be the final one.

Consider for instance the time needed to get a cup of coffee in a common coffee room. Most of the time, you simply walk into the room, grab the pot, and pour coffee into your cup. Perhaps you spill a few drops on the table. Once in a while, however, you have the bad luck to find the pot empty, and you must start a fresh brew, which is considerably more time-consuming. While you are at it, you may as well clean up the table. Thus, the algorithm for getting a cup of coffee takes substantial time in the worst case, yet it is quick in the amortized sense because a long time is required only after several cups have been obtained quickly. (For this analogy to work properly, we must assume somewhat unrealistically that the pot is full when the first person walks in; otherwise the very first cup would consume too much time.)

A classic example of this behavior in computer science concerns storage allocation with occasional need for "garbage collection". A simpler example concerns updating a binary counter. Suppose we wish to manage the counter as an array of bits representing the value of the counter in binary notation: array $C[1..m]$ represents $\sum_{j=1}^{m} 2^{m-j} C[j]$. For instance, array [0,1,1,0,1,1] represents 27. Since such a counter can only count up to 2^m- 1, we shall assume that we are happy to count modulo 2^m. Here is the algorithm for adding 1 to the counter.

```
procedure count(C[1..m])
    {This procedure assumes m ≥ 1
     and C[j] ∈ {0,1} for each 1 ≤ j ≤ m}
    j ← m + 1
    repeat
        j ← j − 1
        C[j] ← 1 − C[j]
    until C[j] = 1 or j = 1
```

Called on our example [0,1,1,0,1,1], the array becomes [0,1,1,0,1,0] the first time round the loop, [0,1,1,0,0,0] the second time, and [0,1,1,1,0,0] the third time (which indeed represents the value 28 in binary); the loop then terminates with $j = 4$ since $C[4]$ is now equal to 1. Clearly, the algorithm's worst case occurs when $C[j] = 1$ for all j, in which case it goes round the loop m times. Therefore, n calls on *count* starting from an all-zero array take total time in $O(nm)$. But do they take a time in $\theta(nm)$?The answer is negative, as we are about to show that *count* takes constant amortized time. This implies that our n calls on *count* collectively take a time in $\theta(n)$, with a hidden constant that does not depend on m. In particular, counting from 0 to $n = 2^m - 1$can be achieved in a time linear in n, whereas careless worst-case analysis of *count* would yield the correct but pessimistic conclusion that it takes a time in $O(n \log n)$.

There are two main techniques to establish amortized analysis results: the potential function approach and the accounting trick. Both techniques apply best to analyze the number of times a barometer instruction is executed.

Potential Functions

Suppose the process to be analyzed modifies a database and its efficiency each time it is called depends on the current state of that database. We associate with the database a notion of "cleanliness", known as the *potential function* of the database. Calls on the process are allowed to take more time than average provided they clean up the database. Conversely, quick calls are allowed to mess it up. This is precisely what happens in the coffee room! The analogy holds even further: the faster you fill up your cup, the more likely you will spill coffee, which in turn means that it will take longer when the time comes to clean up. Similarly, the faster the process goes when it goes fast, the more it messes up the database, which in turn requires more time when cleaning up becomes necessary.

Formally, we introduce an integer-valued potential function Φof the state of the database. Larger values of Φ correspond to dirtier states. Let ϕ_0 be the value of Φ on the initial state; it represents our standard of cleanliness. Let ϕ_i be the value of Φ on the database after the i^{th} call on the process, and let t_i be the time needed by that call (or the number of times the barometer instruction is performed). We define the *amortized time* taken by that call as

$$\hat{t}_i = t_i + \phi_i - \phi_{i-1}.$$

Thus, $\hat{t}_i$ is the actual time required to carry out the i^{th} call on the process plus the increase in potential caused by that call. It is sometimes better to think of it as the actual time minus the decrease in potential, as this shows that operations that clean up the database will be allowed to run longer without incurring a penalty in terms of their amortized time.

Let $T_n$denote the total time required for the first n calls on the process, and denote the total *amortized* time by $\hat{T}_n$.

$$\begin{aligned}
\hat{T}_n &= \sum_{i=1}^{n} \hat{t}_i = \sum_{i=1}^{n} (t_i + \phi_i - \phi_{i-1}) \\
&= \sum_{i=1}^{n} t_i + \sum_{i=1}^{n} \phi_i - \sum_{i=1}^{n} \phi_{i-1} \\
&= T_n + \phi_n + \phi_{n-1} + \cdots + \phi_1 \\
&\qquad - \phi_{n-1} - \cdots - \phi_1 - \phi_0 \\
&= T_n + \phi_n - \phi_0
\end{aligned}$$

Therefore

$$T_n = \hat{T}_n - (\phi_n - \phi_0).$$

The significance of this is that $T_n \le \hat{T}_n$ holds for all n provided ϕ_n never becomes smaller than ϕ_0. In other words, the total amortized time is always an upper bound on the total actual time needed to perform a sequence of operations, as long as the database is never allowed to become "cleaner" than it was initially. This approach is interesting when the actual time varies

significantly from one call to the next, whereas the amortized time is nearly invariant. For instance, a sequence of operations takes linear time when the amortized time per operation is constant, regardless of the actual time required for each operation. The challenge in applying this technique is to figure out the proper potential function. We illustrate this with our example of the binary counter. A call on *count* is increasingly expensive as the rightmost zero in the counter is farther to the left. Therefore the potential function that immediately comes to mind would be m minus the largest j such that $C[j]=0$. It turns out, however, that this choice of potential function is not appropriate because a single operation can mess up the counter arbitrarily (adding 1 to the counter representing 2^k -2 causes this potential function to jump from 0 to k). Fortunately, a simpler potential function works well: define $\Phi(C)$ as the number of bits equal to 1 in C. Clearly, our condition that the potential never be allowed to decrease below the initial potential holds since the initial potential is zero.

What is the amortized cost of adding 1 to the counter, in terms of the number of times we go round the loop? There are three cases to consider.

- If the counter represents an even integer, we go round the loop once only as we flip the rightmost bit from 0 to 1. As a result, there is one more bit set to 1 than there was before. Therefore, the actual cost is 1 trip round the loop, and the increase in potential is also 1. By definition, the amortized cost of the operation is 1 + 1 = 2.
- If all the bits in the counter are equal to 1, we go round the loop m times, flipping all those bits to 0. As a result, the potential drops from m to 0. Therefore, the amortized cost is $m - m = 0$.
- In all other cases, each time we go round the loop we decrease the potential by 1 since we flip a bit from 1 to 0, except for the last trip round the loop when we increase the potential by 1 since we flip a bit from 0 to 1. Thus, if we go round the loop k times, we decrease the potential k - 1 times and we increase it once, for a net decrease of k - 2. Therefore, the amortized cost is $k - (k - 2) = 2$.

In conclusion, the amortized cost of adding 1 to a binary counter is always exactly equivalent to going round the loop twice, except that it costs nothing when the counter cycles back to zero. Since the actual cost of a sequence of operations is never more than the amortized cost, this

proves that the total number of times we go round the loop when incrementing a counter n times in succession is at most $2n$ provided the counter was initially set to zero.

The Accounting Trick

This technique can be thought of as a restatement of the potential function approach, yet it is easier to apply in some contexts. Suppose you have already guessed an upper bound T on the time spent in the amortized sense whenever process P is called, and you wish to prove that your intuition was correct (Tmay depend on various parameters, such as the instance size). To use the accounting trick, you must set up a virtual bank account, which initially contains zero tokens. Each time P is called, an allowance of Ttokens is deposited in the account; each time the barometer instruction is executed, you must pay for it by spending one token from the account. The golden rule is never to allow the account to become overdrawn. This insures that long operations are permitted only if sufficiently many quick ones have already taken place. Therefore, it suffices to show that the golden rule is obeyed to conclude that the actual time taken by any sequence of operations never exceeds its amortized time, and in particular any sequence of s operations takes a time that is at most Ts.

To analyze our example of a binary counter, we allocate two tokens for each call on *count* (this is our initial guess) and we spend one token each time *count* goes round its loop. The key insight again concerns the number of bits set to 1 in the counter. We leave it for the reader to verify that each call on *count* increases (decreases) the amount available in the bank account precisely by the increase (decrease) it causes in the number of bits set to 1 in the counter (unless the counter cycles back to zero, in which case less tokens are spent). In other words, if there were i bits set to 1 in the counter before the call and $j>0$ bits afterwards, the number of tokens available in the bank account once the call is completed has increased by $j- i$ (counting a negative increase as a decrease). Consequently, the number of tokens in the account is always exactly equal to the number of bits currently set to 1 in the counter (unless the counter has cycled, in which case there are more tokens in the account). This proves that the account is never overdrawn since the number of bits set to 1 cannot be negative, and therefore each call on *count* costs at most two tokens in the amortized sense.

3.7 SOLVING RECURRENCES

The indispensable last step when analyzing an algorithm is often to solve a recurrence equation. With a little experience and intuition most recurrences can be solved by intelligent guesswork. However, there exists a powerful technique that can be used to solve certain classes of recurrence almost automatically. This is the main topic of this section: the technique of the *characteristic equation*.

Intelligent Guesswork

This approach generally proceeds in four stages: calculate the first few values of the recurrence, look for regularity, guess a suitable general form, and finally prove by mathematical induction (perhaps constructive induction) that this form is correct. Consider the following recurrence.

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ 3T(n \div 2) + n & \text{otherwise} \end{cases} \qquad (4.4)$$

One of the first lessons experience will teach you if you try solving recurrences is that discontinuous functions such as the floor function (implicit in $n \div 2$) are hard to analyze. Our first step is to replace $n \div 2$ with the better-behaved *"n/2"* with a suitable restriction on the set of values of n that we consider initially. It is tempting to restrict n to being even since in that case $n \div 2 = n/2$, but recursively dividing an even number by 2 may produce an odd number larger than 1. Therefore, it is a better idea to restrict n to being an exact power of 2. Once this special case is handled, the general case follows painlessly in asymptotic notation.

First, we tabulate the value of the recurrence on the first few powers of 2.

n	1	2	4	8	16	32
$T(n)$	1	5	19	65	211	665

Each term in this table but the first is computed from the previous term. For instance, $T(16) = 3 \times T(8) + 16 = 3 \times 65 + 16 = 211$. But is this table useful? There is certainly no obvious pattern in

this sequence! What regularity is there to look for? The solution becomes apparent if we keep more "history" about the value of $T(n)$.Instead of writing $T(2) = 5$, it is more useful to write $T(2)= 3 \times 1+ 2$.

Then,

$$T(4)= 3 \times T(2)+4 = 3 \times (3 \times 1 + 2)+4 = 3^2 \times 1 + 3 \times 2 + 4.$$

We continue in this way, writing n as an explicit power of 2.

n	$T(n)$
1	1
2	$3 \times 1 + 2$
2^2	$3^2 \times 1 + 3 \times 2 + 2^2$
2^3	$3^3 \times 1 + 3^2 \times 2 + 3 \times 2^2 + 2^3$
2^4	$3^4 \times 1 + 3^3 \times 2 + 3^2 \times 2^2 + 3 \times 2^3 + 2^4$
2^5	$3^5 \times 1 + 3^4 \times 2 + 3^3 \times 2^2 + 3^2 \times 2^3 + 3 \times 2^4 + 2^5$

The pattern is now obvious.

$$\begin{aligned} T(2^k) &= 3^k 2^0 + 3^{k-1} 2^1 + 3^{k-2} 2^2 + \cdots 1\, 2^{k-1} + 3^0 2^k \\ &= \sum_{i=0}^{k} 3^{k-i} 2^i = 3^k \sum_{i=0}^{k} (2/3)^i \\ &= 3^k \times (1 - (2/3)^{k+1}) / (1 - 2/3) \\ &= 3^{k+1} - 2^{k+1} \end{aligned} \qquad (4.5)$$

It is easy to check this formula against our earlier tabulation. By induction (not *mathematical* induction), we are now convinced that the above equation is correct.

With hindsight, the Equation (4.5) could have been guessed with just a little more intuition. For this it would have been enough to tabulate the value of $T(n) + in$ for small values of i, such as $-2 \leq i \leq 2$.

n	1	2	4	8	16	32
$T(n)-2n$	-1	1	11	49	179	601
$T(n)-n$	0	3	15	57	195	633
$T(n)$	1	5	19	65	211	665
$T(n)+n$	2	7	23	73	227	697
$T(n)+2n$	3	9	27	81	243	729

This time, it is immediately apparent that $T(n)+2n$ is an exact power of 3, from which the Equation (4.5) is readily derived.

What happens when n is not a power of 2? Solving recurrence 4.4 exactly is rather difficult. Fortunately, this is unnecessary if we are happy to obtain the answer in asymptotic notation. For this, it is convenient to rewrite Equation 4.5 in terms of $T(n)$rather than in terms of $T(2^k)$.Since $n = 2k$ it follows that $k = \lg n$.

Therefore

$$T(n) = T(2^{\lg n}) = 3^{1+\lg n} - 2^{1+\lg n}$$

Using the fact that $3^{\lg n} = n^{\lg 3}$ it follows that $T(n) = 3n^{\lg 3} - 2n$ (4.6)

when n is a power of 2. Using conditional asymptotic notation, we conclude that $T(n) \in \Theta(n^{\lg 3} \mid n$ is a power of 2). Since $T(n)$is a nondecreasing function (a fact easily proven by mathematical induction) and $n^{\lg 3}$ is a smooth function and $T(n) \in \Theta(n^{\lg 3})$ unconditionally.

Homogeneous Recurrences

We begin our study of the *technique of the characteristic equation* with the resolution of homogeneous linear recurrences with constant coefficients, that is recurrences of the form

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0 \qquad (4.7)$$

where the $t_i$are the values we are looking for. In addition to Equation 4.7, the values of $t_i$on k values of i (usually $0 \le i \le k$ - 1 or $1 \le i \le k$) are needed to determine the sequence. These *initial*

conditions will be considered later. Until then, Equation 4.7 typically has infinitely many solutions. This recurrence is

- *linear* because it does not contain terms of the form $t_{n-i}t_{n-j}$, t_{n-i}^2, and so on;
- *homogeneous* because the linear combination of the t_{n-i} is equal to zero; and
- *with constant coefficients* because the a_i are constants.
- Consider for instance our now familiar recurrence for the Fibonacci sequence.

Consider for instance our now familiar recurrence for the Fibonacci sequence.

$$f_n = f_{n-1} + f_{n-2}$$

This recurrence easily fits the mould of Equation 4.7 after obvious rewriting.

$$f_n - f_{n-1} - f_{n-2} = 0$$

Therefore, the Fibonacci sequence corresponds to a homogeneous linear recurrence with constant coefficients with $k = 2$, $a_0 = 1$and $a_1, = a_2 = -1$.

Before we even start to look for solutions to Equation 4.7, it is interesting to note that any linear combination of solutions is itself a solution. In other words, if f_n,and $g_n$satisfy Equation 4.7, $\sum_{i=0}^{k} a_i f_{n-i} = 0$ and similarly for g_n,and if we set $t_n = cf_n + dg_n$ for arbitrary constants c and d,then t_n,is also a solution to Equation 4.7. This is true because

$$\begin{aligned}
&a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} \\
&= a_0(cf_n + dg_n) + a_1(cf_{n-1} + dg_{n-1}) + \cdots + a_k(cf_{n-k} + dg_{n-k}) \\
&= c(a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k}) + d(a_0 g_n + a_1 g_{n-1} + \cdots + a_k g_{n-k}) \\
&= c \times 0 + d \times 0 = 0.
\end{aligned}$$

This rule generalizes to linear combinations of any number of solutions.

Trying to solve a few easy examples of recurrences of the form of Equation 4.7 *(not* the Fibonacci sequence) by intelligent guesswork suggests looking for solutions of the form

$$t_n = x^n$$

where x is a constant as yet unknown. If we try this guessed solution in Equation 4.7, we obtain

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = 0.$$

This equation is satisfied if x = 0, a trivial solution of no interest. Otherwise, the equation is satisfied if and only if

$$a_0 x^k + a_1 x^{k-1} + \cdots + a_k = 0.$$

This equation of degree k in x is called the *characteristic equation* of the recurrence 4.7 and

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

is called its *characteristic polynomial.*

Recall that the fundamental theorem of algebra states that any polynomial $p(x)$ of degree k has exactly k roots (not necessarily distinct), which means that it can be factorized as a product of k monomials

$$p(x) = \prod_{i=1}^{k} (x - r_i)$$

where the r_i may be complex numbers. Moreover, these r_i are the only solutions of the equation $p(x) = 0$.

Consider any root $r_i$of the characteristic polynomial. Since $p(r_i) = 0$ it follows that $x = r_i$ is a solution to the characteristic equation and therefore $r_i^n$is a solution to the recurrence. Since any linear combination of solutions is also a solution, we conclude that

$$t_n = \sum_{i=1}^{k} c_i r_i^n \tag{4.8}$$

satisfies the recurrence for any choice of constants C_1, C_2, …, C_k.The remarkable fact, which we do not prove here, is that Equation 4.7 has *only* solutions of this form *provided all the r_i are distinct.* In this case, the k constants can be determined from k initial conditions by solving a system of k linear equations in k unknowns.

Example: (Fibonacci) Consider the recurrence

$$f_n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

First we rewrite this recurrence to fit the mould of Equation 4.7.

$$f_n - f_{n-1} - f_{n-2} = 0$$

The characteristic polynomial is

$$x^2 - x - 1$$

whose roots are

$$r_1 = \frac{1+\sqrt{5}}{2} \text{ and } r_2 = \frac{1-\sqrt{5}}{2}.$$

The general solution is therefore of the form

$$f_n = c_1 r_1^n + c_2 r_2^n. \tag{4.9}$$

It remains to use the initial conditions to determine the constants c_1 and c_2. When $n = 0$, Equation 4.9 yields $f_0 = c_1 + c_2$.But we know that $f_0 = 0$. Therefore, $c_1 + c_2 = 0$.Similarly, when $n = 1$, Equation 4.9 together with the second initial condition tell us that $f_1 = c_1 r_1 + c_2 r_2 = 1$. Remembering that the values of $r_1$and r_2 are known, this gives us two linear equations in the two unknowns $c_1$and c_2.

$$\begin{aligned} c_1 &+ c_2 &&= 0 \\ r_1 c_1 &+ r_2 c_2 &&= 1 \end{aligned}$$

Solving these equations, we obtain

$$c_1 = \frac{1}{\sqrt{5}} \text{ and } c_2 = -\frac{1}{\sqrt{5}}.$$

Thus

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right],$$

which is de Moivre's famous formula for the Fibonacci sequence. Notice how much easier the technique of the characteristic equation is than the approach by constructive induction. It is also more precise since all we were able to discover with constructive induction was that "$f_n$grows exponentially in a number close to ϕ"; now we have an exact formula.

3.8 SUMMARY

In this unit, we discussed the techniques to analyze the complexity and performance of algorithms. We analyzed the most frequently encountered control structures such as sequencing, while loops, for loops and recursive calls to study the complexity and performance of algorithms. We learnt that the use of a barometer is a handy tool to simplify the analysis of many algorithms, but this technique should be used with care. Additional examples are considered to analyze the algorithms involving loops, recursion, and the use of barometers. We observed that insertion sorting makes on the average about half as many comparisons as in the worst case and this number is quadratic. The two main techniques to establish amortized analysis results: the potential function approach and the accounting trick are discussed. Both techniques apply best to analyze the number of times a barometer instruction is executed. The indispensable step when analyzing an algorithm is often to solve a recurrence equation. With a little experience and intuition most recurrences can be solved by intelligent guesswork. The characteristic equation is a powerful technique that can be used to solve certain classes of recurrence almost automatically.

3.9 KEYWORDS

1) Control structures
2) Sequencing
3) For loops
4) While loops
5) Recursive calls
6) Barometer
7) Insertion sort
8) Selection sort
9) Time complexity
10) Amortized Analysis
11) Recurrence
12) Accounting trick

3.10 QUESTIONS

1) Explain various loop structures to be considered for analyzing algorithms.
2) How do you analyze recursive algorithms? Explain with an example.
3) How do you determine an upper bound on the running time of binary search algorithm? Explain.
4) What is a barometer instruction? How is it useful to analyze an algorithm? Explain with an example.
5) Explain average case analysis of algorithms considering suitable examples
6) What is meant by amortized analysis? Explain with an example
7) Explain the potential function approach to analyze algorithms
8) Explain intelligent guesswork approach to solve recurrences

3.11 EXCERCISES

1) Determine the complexity of sorting algorithms using amortization scheme.
2) Obtain the general solution for a Fibonacci sequence using characteristic equation approach.
3) Analyze while loops considering Binary search algorithm
4) Analyze the time complexity of selection sort algorithm
5) Analyze the time complexity of insertion sort algorithm

3.8 REFERENCES

4) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.

5) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.

6) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

Unit 4

Searching Technique

Structure

4.0 Objectives

After reading this unit you should be able to

- Discuss the basic concept of searching techniques
- Explain linear search technique
- Discuss the working principle of conventional sorting technique
- Discuss two other sorting techniques such as selection sort and insertion sort
- Simulate the algorithms on any given set of data

4.1 Introduction to Searching Techniques

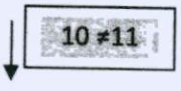
Searching is a technique of finding whether a given element is present in a list of elements. If the search element is present in the list the searching technique should return the index where the given searching element is present in the list. If the search element is not present in the list then the searching technique should return NULL indicating that search element is not present in the list. Similar to sorting there are number of searching technique available in the literature and no single algorithm suits all applications. Some algorithms work faster but require more memory on the other hand some techniques are too fast but they assume that the given list is already in sorted order. On the other hand searching techniques can be classified based on the data structures used to store the list. If array is used, then we must use different searching technique. Searching an element in a non linear data structure requires different searching techniques.

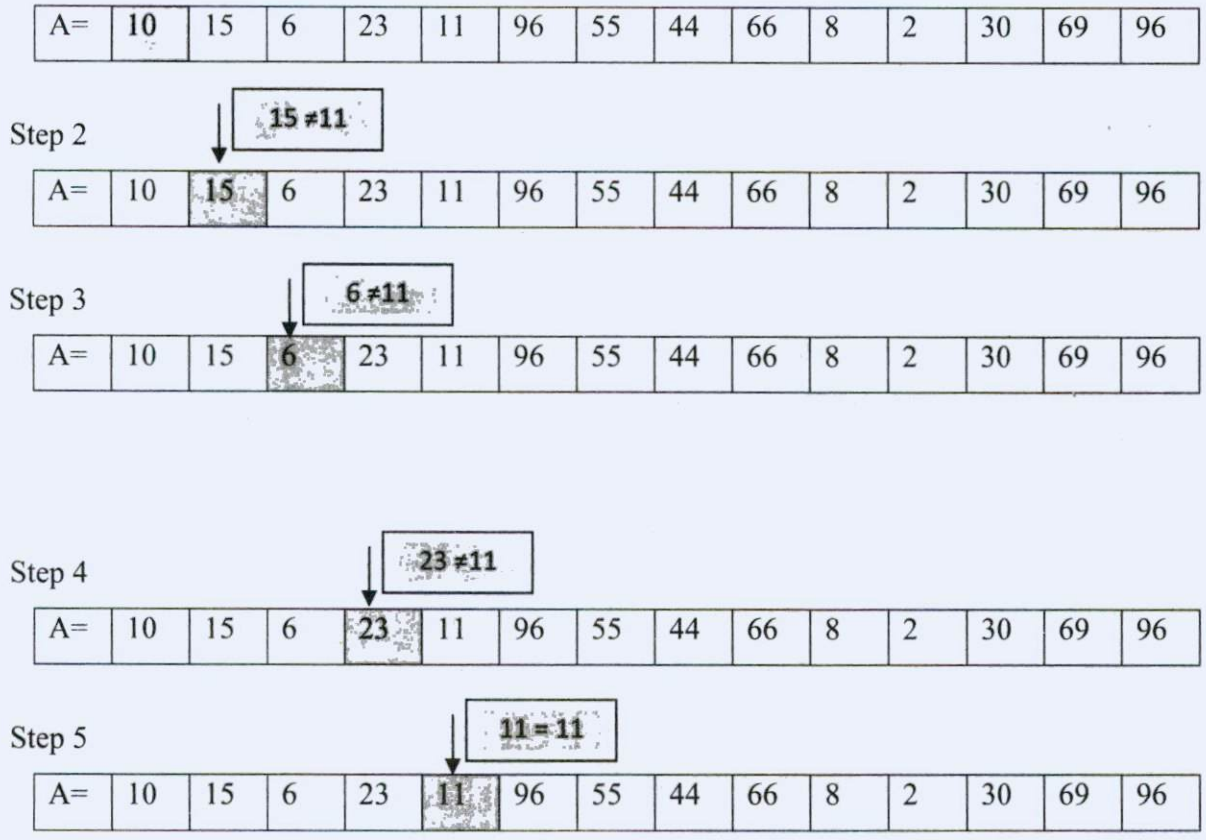
In this unit we present four different searching techniques. First, we present linear search technique which is simplest of all searching technique. Next we present another fast working technique called binary search. Later we present two other searching techniques viz., breadth first search and depth first search technique. The former two are for array type of data structure and later two are for graph data structure.

4.2 Linear Search (Sequential search)

Linear search is a method of searching an element in the list where the given search element 'e' is compared against all elements sequentially until there is atleast one match (i.e., Success) or search process reaches the end of the list without finding any match (i.e., Failure).

Let A = [10 15 6 23 8 96 55 44 66 11 2 30 69 96] and searching element 'e' = 11. Consider a pointer 'i', to begin with the process initialize the pointer 'i' = 1. Compare the value pointed by the pointer with the searching element 'e' = 11. As A(1) = 10 and its is not equal to element 'e' increment the pointer i by i+1. Compare the value pointed by pointer i.e., A(2) = 15 and it is also not equal to element 'e'. Continue the process until the search element is found or the pointer 'i' reaches the end of the list.

Step 1 



Element found in the list A at the position 5.

Now, let us consider the same list with different search element 'e' = 12.

If you search the list using the above mechanism, the pointer moves till the end of the list. This indicates that the given search element 'e' = 12 is not present in the list and returns NULL. The algorithm for linear search is as given below.

Algorithm Linear search

Input A – List of elements
'e' element to be searched
'n' size of the list

Output Success or Unsuccess.

Method

```
For i = 1 : n
        If (A(i) = = e)
                display 'Element e is present at the position i'
                Flag = 1
                break
        If end
For end
```

```
If(Flag = = 0)
        Display 'Element e is not present in the list A'
If end
```

Algorithm ends

The time taken to search an element in the list is 'n+1' in the worst case when the element is not present in the list. In case if element is present the time taken will be the time taken to reach the position of the element. If the element is present at the end of the list of size 'n' the time taken is 'n' and if the element is present in the first position the time required is 1 unit. Linear search or sequential search suits the applications where the size of the list is less. If the size of the list is more, then linear search may perform very poor.

4.3Conventional sort

The basic step in this sorting technique is to bring the smallest element of the unordered list to the recent position. In this technique we will consider two pointers 'i' and 'j'. Initially pointer 'i' will be pointing to first data point and pointer 'j' will be pointing to the next data point. Compare the value pointed by pointers 'i' and 'j', if the value pointed by pointer 'j' is smaller than the value pointed by pointer 'i', then swap those two values else do not swap the values. Increment the pointer 'j' so that it point to the next position. Check whether the value pointed by pointers 'i' and 'j', if the value pointed by pointer 'j' is smaller than the value pointed by pointer 'i', then swap those two values else do not swap the values and increment the pointer 'j' one at a time. Continue the same process until the pointer 'j' reaches the last position. At the end of this process we can see that the smallest element in the list is at the location pointed by the pointer 'i'.

Now increment the pointer 'i' by one and initialize the pointer 'j' to the next location of the pointer 'i'. Continue the above discussed process until the pointer 'i' reaches the last but one position of the list. The algorithm for the conventional list is as given below.

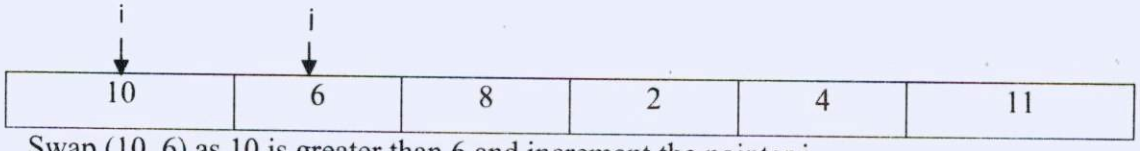
Algorithm	Conventional sort
Input	Unordered list - A
Output	Ordered list – A

Method

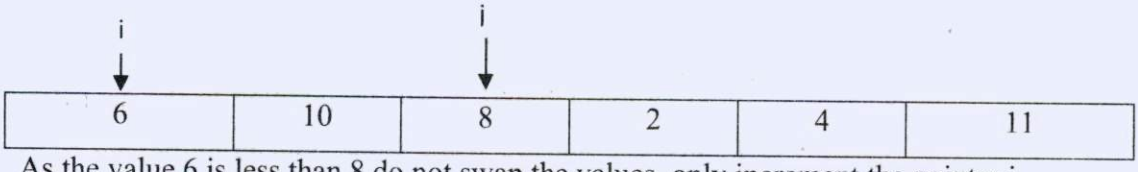
```
For i =  1 to n-1 do
        For j = i + 1 to n do
            if (A(i) > A(j) )
                Swap(A(i), A(j))
            if end
        For end
For end
```

Algorithm ends

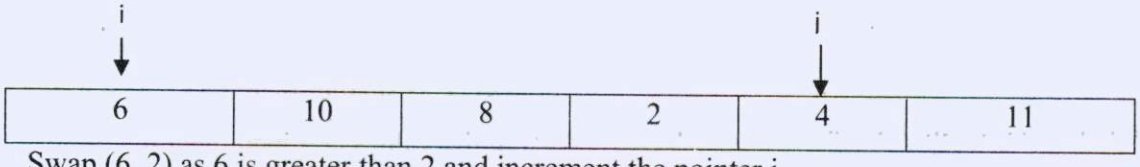
In order to illustrate the conventional sorting technique let us consider an example where a list A={10, 6, 8, 2, 4, 11} contains unordered set of integer numbers.



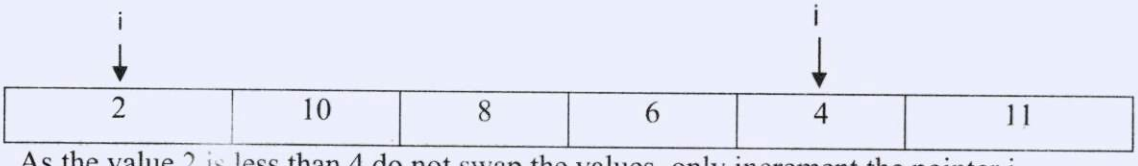
Swap (10, 6) as 10 is greater than 6 and increment the pointer j



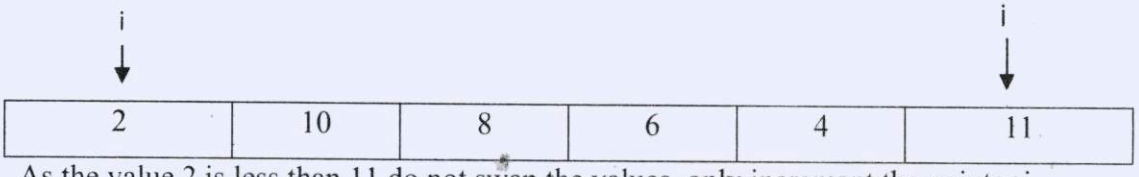
As the value 6 is less than 8 do not swap the values, only increment the pointer j



Swap (6, 2) as 6 is greater than 2 and increment the pointer j



As the value 2 is less than 4 do not swap the values, only increment the pointer j



As the value 2 is less than 11 do not swap the values, only increment the pointer j

This complete one iteration and you can observe that the smallest element is present in the first position. For the second iteration, increment the pointer 'i' such that it points to next location and initialize the pointer 'j' to next position of pointer 'i'. Carry out the same procedure as explained above and it can be observed that at the end of 2nd iteration the list will be as follows.

2	4	10	8	6	11

Continue the same process as explained above and at the end of the process it can be observed that the list will be sorted in ascending order and it looks as follows

2	4	6	8	10	11

If we assume that each step will take 1 unit of time, then the total time taken to sort the considered 6 elements is 5+4+3+2+1 = 15 unit of times. In general if there are n elements in the unordered list the time taken to sort is $(n-1)+(n-2)+(n-3)+...+3+2+1=\frac{n^2-3n+2}{2}$.

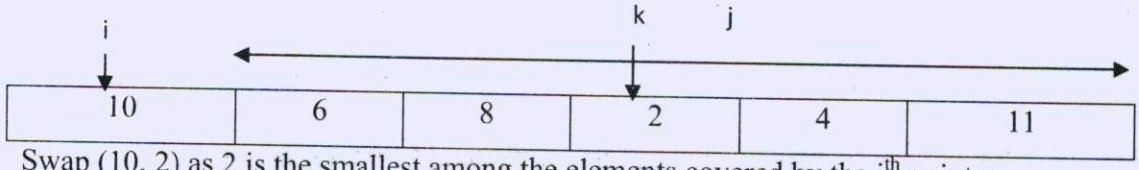
4.4 Selection sort

In case of conventional sort we saw that in each step the smallest element will be brought to the respective positions. In the conventional algorithm it should be noted that there is possible of swapping elements in each step, which is more time consuming part. In order to reduce this swapping time we have another sorting technique called selection sort in which we first select the smallest element in each iteration and swap that smallest number with the first element of the unsorted list part.

Consider the same example which was considered in demonstrating the conventional sorting technique. Similar to conventional sorting technique in this selection sorting technique also we consider same two pointers 'i' and 'j'. As usual 'i' will be pointing to first position and 'j' will be pointing to the next position of 'i'. Find the minimum for all values of 'j' i.e., $i+1 \le j \le n$. Let

the smallest element be present in k^{th} position. If the element pointed by 'i' is larger than the element at k^{th} position swap those values else increment 'i' and set the value of j = i+1.

It should be observed clearly that in case of selection sort there is only one swap where as in case of conventional sorting technique there is possibility of having more than one swapping of elements in each iteration.



Swap (10, 2) as 2 is the smallest among the elements covered by the j^{th} pointer.

2	6	8	10	4	11

Similarly do the process for reaming set and the resultant steps are as follows.

2	6	8	10	4	11
2	4	8	10	6	11
2	4	6	10	8	11
2	4	6	8	10	11
2	4	6	8	10	11
2	4	6	8	11	11

The algorithm for selection sort is as follows.

Algorithm Selection sort
Input Unordered list - A
Output Ordered list – A
Method

```
For i =  1 to n-1 do
        For j = i + 1 to n do
        k = i
                if (A(j) < A(i) )
                    k = j
                if end
        For end
        If(i ≠j)
                    Swap(A(i), A(k))
        If end
```

```
        For end
Algorithm ends
```

4.5 Insertion sort

The basic step in this method is to insert an element 'e' into a sequence of ordered elements e_1, e_2, e_3, .. e_j in such a way that the resulting sequence of size i+1 is also ordered. We start with an array of size 1 which is in sorted order. By inserting the second element into its appropriate position we get an ordered list of two elements. Similarly, each subsequent element will be added into their respective positions obtaining a partial sorted array. The algorithm for the insertion sort is a follows.

```
Algorithm   Insertion sort
Input       Unordered list – A
Output      Ordered list – A
Method

            For i = 2 to n do
                j = i-1, k=A(i)
                While ( (j > 0) and A(j) > k)
                    A(j+1) = A(j)
                    j = j – 1
                While end
                A(j+1) = k
            For end
Algorithm ends
```

To illustrate the working principle of the insertion sort let us consider the same set of elements considered in the above sections.

10	6	8	2	4	11

Consider the first element, as there is only one element it is already sorted.

10	6	8	2	4	11

Now consider the second element 6 and insert that element in the respective position of the sorted list. As 6 is less than 10 insert 6 before the value 10.

6	10	8	2	4	11

The third element is 8 and the value of 8 is greater than 6 and less than 10. Hence insert the element 8 in between 6 and 10.

6	8	10	2	4	11

The fourth element is 2 and it is the smallest among all the elements in the partially sorted list. So insert the value 2 in the beginning of the partially sorted list.

2	6	8	10	4	11

The fifth element is 4 and the value of 4 is greater than 2 and less than remaining elements of the partially sorted list {6, 8, 10} and hence insert the element 4 in between 2 and {6, 8, 10}.

2	4	6	8	10	11

The remaining element is 11 and it is largest of all elements of the partially sorted list {2, 4, 6, 8, 10}, so leave the element in its place only. The finally sorted list is as follows.

2	4	6	8	10	11

The insertion sort works faster than the conventional sort and selection sort. The computation of time taken to sort an unordered set of elements using insertion sort is left as assignment to the students (Refer question number 3).

4.6 Binary search based insertion sort

Binary search based insertionsortis an improvement over insertion sort, which employs a modified version of binary search to sort the list of 'n' element. The central idea of the BSBI algorithm is similar to that of Insertion sort but the method used for searching a position for

inserting an element in the sorted sub-list is binary search instead of linear search. Given the fact that the binary search is more efficient than the linear search, the proposed algorithm exhibits better performance than the insertion sort. The algorithm for BSBI is as follows.

Algorithm: BSBSI (Binary search Based Sorting by Insertion)

Input: n – number of elements

A – Unordered list of 'n' elements

Output: A – Ordered list of 'n' elements

Method:

```
For i = 2 to n do
    X = A(i)
    Posi = Modifief_Bi_Search(X, i+1)
    If(Posi ≠ 0)
        For j = 1 down to (posi+1) do
            A[j] = A[j-1]
        For end
        A[Posi] = x
    If end
For end
```

Algorithm end

The Modifief_Bi_Search is a modified version of the binary search technique. It is used to find the location where the insertion is to be made, but not to find the location of the key element.

Algorithm : Modifief_Bi_Search

Input : x – Element whose position to be found

High – Upper limit of the ordered sub-list

Output: Posi – Position where 'x' is to be inserted

Method:

```
first = 1
do
        mid = (first + high) /2
        if(x ≥ A(mid) and x < A(mid+1))
                posi = mid + 1
        else if (x ≥ A(mid-1) and x < A(mid))
                posi = mid
            else if(A(mid) < x)
                        first = mid +1
                    else
                        high = mid
                    If end
                If end
        If end
    While(first < high)
    If(x > A(first) )
        Posi = 0
    else
        posi = first
    If end
```

Algorithm end

Since BSBI is an improvement on insertion sort by employing the binary search instead of sequential search, the students are asked to experience themselves the working principle of BSBI technique.

4.7 SUMMARY

In this unit we have presented the basics of searching technique. We have presented different sorting techniques such as conventional sort, binary search based insertion sort, selection sort and insertion sort. The algorithm for all the three sorting technique is given in this unit and demonstrated with suitable example.

4.8 KEYWORDS

1) Sorting technique
2) Conventional sort
3) Selection sort
4) Insertion sort
5) Binary search based insertion sort

4.9 QUESTIONS

(1). Designan algorithm to sort given set of n numbers using binary search based insertion sort.

(2). Consider a set A = {10 56 89 42 63 1 9 7 5 23}. With suitable steps demonstrate the working principle of binary search based insertion sort.

(3). Design and develop an algorithm to sort unordered set of element in descending order using (i) Conventional sort (ii) Selection sort (iii) Insertion sort.

(4). Calculate the time taken to sort n elements present in an unordered list using insertion sort.

(5). Sort the given unordered set {12,2,16,30,8,28,4,10,20,6,18} using conventional sort and count the number of swapping took during the sorting process

(6). For the same unordered set given in question 1 sort using selection sort and insertion sort and count the number of swapping too during the sorting process.

4.10 REFERENCES

(1). Ellis Horowitz, Sartaj Sahni, and Dinesh Mehta. Fundamental of Data Structures in C++

(2). Alfred V. Aho , Jeffrey D. Ullman, John E. Hopcroft. Data Structures and Algorithms. Addison Wesley (January 11, 1983)

Unit 5

Concept of Data Structure and its importance: Relationship with algorithms, Classification of data structures.

Structure

5.0 Objectives

After studying this unit you should be able to

- Analyze the steps involved in problem solving methodology
- Explain the relationship between data structure and algorithm
- Differentiate the algorithm based on their efficiency
- List outthe properties of an algorithm

5.1 Problem Solving

Solution to a problem requires a proper sequence of steps with well defined unambiguous actions. To solve a problem using computers, these set of actions need to be transformed into precise instructions using a suitable computer language. These set of precise instructions is referred as a program. Developing a program to solve a simple problem is a straight forward task. But developing a program for a complex problem is not a simple task; it must be assisted by some design aids. Algorithms, flowcharts and pseudo codes are few design aids which assist in developing a program. Among all the techniques, algorithm is the most popular design tool. An algorithm is defined as a finite set of well-defined, effective, unambiguous instructions when followed accomplishes a particular task after a finite amount of time, consuming zero or more inputs producing at least one output.

Development of a solution to a problem, in turn, involves various activities namely, understanding the problem definition, formulating a solution model, designing an algorithm, analysis of an algorithm, and description of an algorithm. Once the clear understanding of a problem is done and the solution model is formulated, an algorithm is designed based on the solution model. It is very much important to structure the data used in solving a problem. Exploring the relationship among the data and their storage structure plays an important role in solving a problem. A tool, which preserves the data and their relationships through an appropriate storage structure is termed as data structure. So selection of an appropriate data structure influences the efficiency of an algorithm. Once the algorithm has been designed, correctness of an algorithm is verified for various data samples and its time and space complexities are measured. This activity is normally referred to as analysis of an algorithm. Documentation of an algorithm for clear understanding is very useful for its implementation using a programming language.

5.2 Development of an Algorithm

In order to understand the process of designing an algorithm for a problem, let us consider the problem of verifying if a given number n is a prime as an example. The possible ways to solve this problem are as follows:

Example 5.1

Algorithm 1: Check for prime

Input: A number (n).

Output: Number n is a prime or not.

Method: For all k = 2, 3, 4, ..., (n-1)

If (n mod k <> 0) then n is a prime number

else it is not a prime number.

Algorithm ends

Algorithm 2: Check for prime

Input: A number (n).

Output: Number n is a prime or not.

Method: For all k = 2, 3, 4, ..., (n/2)

If (n mod k <> 0) then n is a prime number

else it is not a prime number.

Algorithm ends

Algorithm 3: Check for prime

Input: A number (n).

Output: Number n is a prime or not.

Method: For all k = 2, 3, 4, ..., (square-root(n))

If (n mod k <> 0) then n is a prime number

else it is not a prime number.

Algorithm ends

From the above examples, it is understood that there are three different ways to solve a problem. The first method divides the given number *n* by all numbers ranging from 2 to (*n*-1) to decide if it is a prime. Similarly, the second method divides the given number *n* by only those numbers from 2 to (*n*/2) and the third method divides the number *n* by only those from 2 upto square root of *n*. This example helped us in understanding that a problem can be solved in different ways, but any one which appears to be more suitable has to be selected.

If we analyze the above algorithms for a number *n*= 100, the first method takes 98 iterations, the second method takes 49 iterations, and the third method takes only 10 iterations. This shows that the third algorithm is the better one to solve this problem.

Let us consider another problem; searching for a telephone number of a person in the telephone directory. If the telephone numbers are not sequenced in an order, then the searching algorithm has to scan the entire telephone directory one by one till it finds the desired number. In the worst case, the element to be searched may happen to be the last number in the directory, and in such case, the searching algorithm will take as many comparisons as the number of telephone numbers in the directory. The number of comparisons grows linearly as size of the directory increases. This will become a bottle neck if the number of entries in the directory is considerably large. On the other hand if the telephone numbers are arranged in some sequence using an appropriate storage structure then we can devise an algorithm, which takes less time to search the required number. Thus the performance of an algorithm mainly depends on the data structure, which preserves the relationship among the data through an appropriate storage structure.

Friends, we now feel it is a time to understand a bit about the properties of an algorithm.

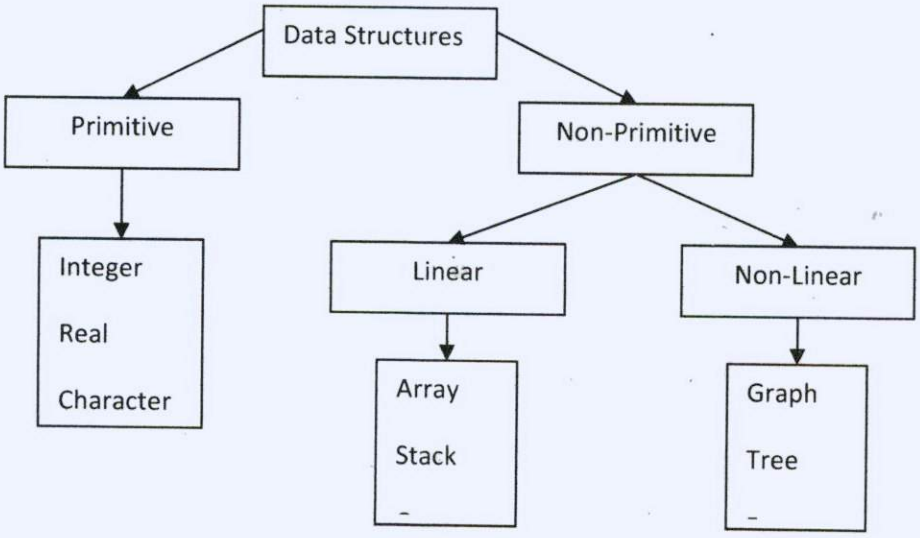
5.3 Classification of data structures

In this section, we present classification of different data structures. Data structures are classified at multiple levels. At the first level they are classified as primitive or non-primitive depending on whether the storage structure contains a single data item or a collection of data items respectively. If it is non-primitive, it is further classified as linear or non-linear depending on the nature of the relationship existing among the data items. If it is linear, it is further classified as

- An Array, if the storage structure contains similar data items and the operations insertion and deletion are not actually defined.

- A Stack, if the storage structure contains similar data items and the operations insertion and deletion are permitted to take place at only one end.
- A Queue, if the storage structure contains similar data items and the insertion operation is performed at one end and the deletion operation is performed at the other end.
- Record, if the storage structure is permitted to contain different types of data items.

A non-linear data structure is further classified as graphs, trees, or forest depending on the adjacency relationship among the elements. Thus the overall classification of data structures can be effectively summarized as shown in Fig 5.5.



5.4 Abstract Data Types

Any individual data item is normally referred to as a data object. For example, an integer or a real number or an alphabetical character can be considered as a data object. The data objects, which comprise the data structure and their fundamental operations, are known as Abstract Data Type (ADT). In other words, an ADT is defined as a set of data objects defined over a domain D, supporting a list of functions F, with a set of axioms A, which are the rules to be satisfied while working on data elements. More formally, it can be defined as a triplet (Domain, Functions, Axioms).

Suppose we want to define the concept of natural number as a data structure, then we have to clearly specifying the domain, functions and axioms.

Example 5.2: Natural numbers as ADT

Domain: {0, 1, 2, 3, 4, 5, . . . , n}

Functions:

Successor (Num) = Num

Iseven (Num) = Boolean

Isodd (Num) = Boolean

Isprime (Num) = Boolean

Lessthan (Num1, Num2) = Boolean

Greaterthan (Num1, Num2) = Boolean

Add (Num1, Num2) = Num3

Subtract (Num1, Num2) = Num3, if (Num1 >= Num2)

Multiply (Num1, Num2) = Num3

Divide (Num1, Num2) = Num3 if (Num2 > 0)

Axioms:

Sub(Add(Num1, Num2), Num2) = Num1

Idiv(Multiply(Num1, Num2), Num2) = Num1

Here the functions are the operations to be done on the natural numbers and the axioms are the rules to be satisfied. Axioms are also called assertions. In this example each natural such as 3, 4 etc is called a data object.

An ADT promotes data abstraction and focuses on what a data structure does rather than how it does. It is easier to comprehend a data structure by means of its ADT since it helps designer to plan on the implementation of the data objects and its supportive operations in any programming language.

5.5 Summary

In this chapter we have studied various types of data structures viz., primitive and non-primitive data structures. We learnt that non-primitive data structures are further divided into linear and non-linear data structures. Definitions of abstract data type, data objects and relationship between abstract data types and data object is demonstrated clearly in this chapter with domain, functions and axioms.

5.6 key words

(1). Primitive data structure

(2). Non-primitive data structure

(3). Linear data structures

(4). Non-linear data structures

(5). Abstract data type

(6). Data object

5.7 Questions for self study

(1). With a neat diagram explain the types of data structures.

(2). What is an abstract data type? Explain with an example.

(3). Mention the difference between abstract data type and data object.

(4). Realize the functions listed for natural number as ADT.

5.8 Reference

(1). Ellis Horowitz and Sartaj Sahni. Fundamentals of Data Structures. W H Freeman and Co (Sd) (June 1983)

(2). Tenenbaum, Langsam, Augenstein. Data Structures Using C. phi

(3). Alfred V. Aho , Jeffrey D. Ullman, John E. Hopcroft. Data Structures andAlgorithms. _____Addison Wesley (January 11, 1983)

Unit 6

Stacks

Structure

6.0 Objectives

After reading this unit you should be able to

- Discuss the basics of stack data structure
- Provide adequate definition to stack data structure
- Implement stack data structure

6.1 Introduction

Solutions to some problems require the associated data to be organized as a linear list of data items in which operations are permitted to take place at only one end of the list. For example, a list of books kept one above another, playing cards, making pancake, storing laundry, wearing bangles, an heap of plates placed one above another in a tray etc. In all these cases, we group things together by placing one

thing on top of another and then removing things one at a time from the top. It is amazing that something this simple is a critical component of nearly every program that is written. The nested function calls in a program, conversion of an infix form of an expression to an equivalent postfix or prefix, computing factorial of a number, and so on can be effectively formulated using this simple principle. In all these cases, it is obvious that the one which recently entered into the list is the one to be operated. Solution to these types of problems is based on the principle Last-In-First-Out (LIFO) or First-In-Last-Out. A logical structure, which organizes the data and performs operations in LIFO or FILO principle, is termed as a Stack.

6.2 Definition of a Stack

Precisely, a stack can be defined as an ordered list of similar data elements in which both insertion and deletion operations are permitted to take place at only one end called top of stack. It is a linear data structure, in which operations are performed on Last-In-First-Out (LIFO) or First-In-Last-Out principle.

More formally, a stack can be defined as an abstract data type with a domain of data objects and a set of functions that can be performed on the data objects guided by a list of axioms.

Domain: {Application dependent data elements}

Functions:

- Create-Stack() – Allocation of memory.
- Isempty(S) – Checking for stack empty: Boolean.
- Isfull(S) – Checking for stack full: Boolean.
- Push(S, e) – Adding an element into a stack: S updated.
- Pop(S) – Removing an element from stack: S updated.
- Top(S) – Displaying an element in the stack.

Axioms:

- Isempty(Create-Stack()): Always True (Boolean value).
- Isfull(Create-Stack()): Always False (Boolean value).
- Isempty(Push(S, e)): Always False (Boolean value).
- Isfull(Pop(S)): Always False (Boolean value).
- Top(push(S, e)): The same element e is displayed.

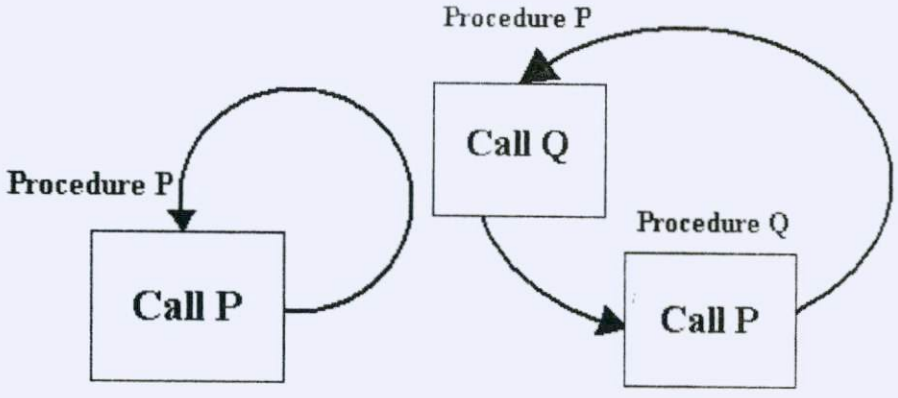
- Pop(Push(S, e): The same element e is removed.

Thus a stack S(D, F, A) can be viewed as an ADT.

6.3Recursion

Recursion is an important concept used in problem solving techniques. Some problems are recursive in nature whose solutions can be better described in terms of recursion. For example, Factorial of a number, generating Fibonacci series, Binary search, Quick sort, Merge sort, GCD of two numbers, Tower of Hanoi etc. to name a few. We can reduce the length and complexity of an algorithm if we can formulate the solution to a problem using the concept of recursion. Conceptually, any recursive algorithm looks very simple and easy to understand. It is a powerful tool that most of the programming languages support. In this chapter, we shall understand the meaning of recursion, how a solution to a problem can be formulated as a recursion, how a recursive function works, properties of recursive procedures along with their merits and demerits and a comparison between recursive and iterative procedures.

Definition: Recursion is a method of defining a problem in terms of itself. That means, if a procedure is defined to solve a problem, then the same procedure is called by itself one or more times to provide a solution to a problem. If **P** is a procedure containing a call to itself or to another procedure that results in a call to itself, then the procedure is said to be **recursive procedure**. In the former case it is called as **direct recursion** and in the later case it is called as **indirect recursion**. Figure 3.1 show these two types of recursive procedures symbolically.



(a) Direct Recursion (b) Indirect Recursion

Figure 3.1 Skeletal recursive procedures

While formulating a recursive procedure, we must understand that (i) there must be one or more criteria, called **base criteria**, where the procedure does not call itself either directly or indirectly and (ii) each time the procedure calls itself directly or indirectly, it must be closer the base criteria.

Example: Let us consider how to compute the factorial of a number using recursive procedure.

We know that $0! = 1$

$1! = 1$

$2! = 2 \times 1 = 1$

$3! = 3 \times 2 \times 1 = 6$

$4! = 4 \times 3 \times 2 \times 1 = 24$

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ and so on.

Therefore, factorial of a number ***n*** is defined as the product of integer values from 1 to ***n***. From the above, we have observed that 5! is given by 5 x 4! and 4! = 4 x 3! and so on. Thus the above procedure can be generalized and the recursive definition to compute ***n!*** can be expressed as below.

$$\text{Fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{Fact}(n-1) & \text{if } n > 0 \end{cases}$$

Thus, $5! = 5 \times 4!$

$4! = 4 \times 3!$

$3! = 3 \times 2!$

$2! = 2 \times 1!$

$1! = 1 \times 0!$

$0! = 1$

1! = 1 x 0! = 1

2! = 2 x 1! = 2

3! = 3 x 2! = 6

4! = 4 x 3! = 24

5! = 5 x 4! = 120

From the above computation, it is observed that the recursive procedure involves decomposing the problem into simpler problems of same type, arrive at the base criteria, which do not involve any recursion, and compute the solution from the previous solutions. The figure 3.2 presents the sequence of recursion calls to compute the factorial of 3.

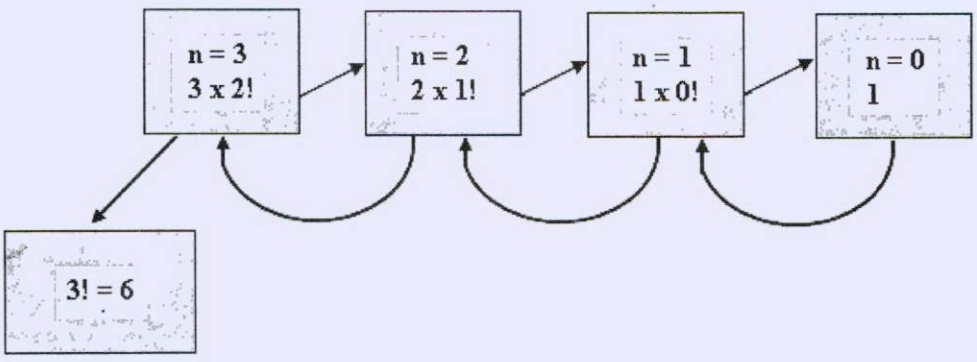


Figure 3.2 Sequence of recursive function calls to compute the factorial of 3

In the above Figure 3.2, each box represents a function, the first function calls the second with n = 2, second function calls the third one with n = 1 and the third function calls the fourth one with n = 0. Since it is the base criteria, no further function calls and it returns the value 1 to the third function. The third function computes the value and returns it to second function. Similarly, second function returns the value to the first function and finally, the function calculates the value and returns it into the main function. Theoretically, the above procedure can be generalized to any value.

In the factorial computation problem, the base case is **Fact (0)**, and the general case is ***n* * Fact (*n* -1)**. It is very much important to understand the following two points while designing any recursive algorithm for a problem.

- **Determining the base case:** Any recursive function must contain a base case, where the function must execute a return statement without a call to it self.
- **Determining the general case:** There must be a general case in any recursive function such that each call must reduce the problem size and must converges to the base case.

6.4 Queue

Queue is an important and familiar concept used in our day to day life. We can see people standing in a queue to board the bus, or standing in a cinema hall to purchase tickets or waiting in a queue for reserving tickets etc. The basic principle followed in these cases is the first come first served and in fact, this is the most pleasing way of serving people when they are in mass requesting a particular service. The concept of first come first served is employed in computer science for solving problems in most of the situations. A queue is adata structure that is similar to a stack,except that in a queue the first iteminserted is the first to be removed (FIFO). In a stack, as we have seen, the lastitem inserted is the first to be removed (LIFO).

A queue works like the line at the movies: the first person to join the rear of the line isthe first person to reach the front of the line and buy a ticket. The last person to line up isthe last person to buy a ticket. Figure 4.1 shows how this looks.Queues are used as a programmer's tool similar to stacks. They are also used to model realworldsituations such as people waiting in line at a bank, airplanes waiting to take off, ordata packets waiting to be transmitted over the Internet.

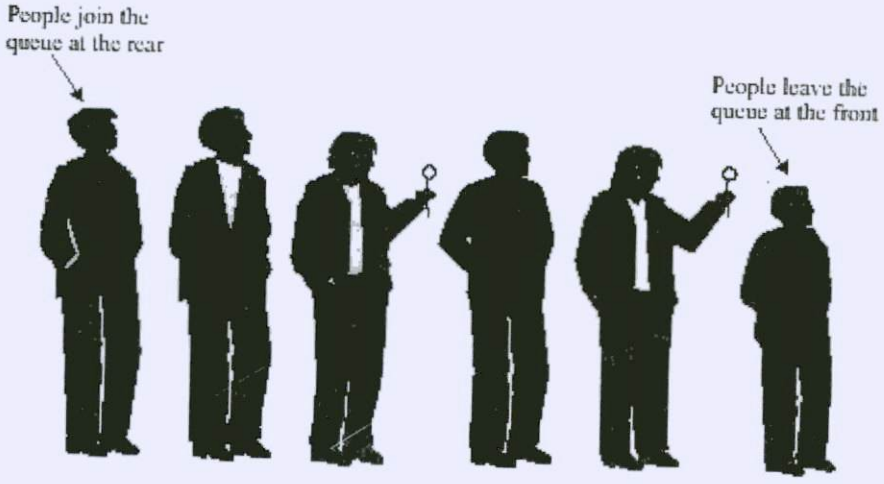


Figure 4.1 A queue of people.

Definition of a queue

We can define a queue more precisely as an ordered list of similar data items in which insertion operation is performed at one end called the *rear end* and the deletion operation is performed at the other end called the *front end*. It is a linear data structure, which works on the basis of LILO or FIFO principle.

More formally, we can define a queue as an abstract data type with a domain of data objects and a set of functions that can be performed on the data objects guided by a list of axioms.

Domain: {Application dependent data elements}

Functions:

- Create-Queue() – Allocation of memory.
- Isempty(Q) – Checking for queue empty: Boolean.
- Isfull(Q) – Checking for queue full: Boolean.
- Insert(Q, *e*) – Adding an element into a queue: Q updated.
- Delete(Q) – Removing an element from a queue: Q updated.
- Front(Q) – Element *e* in the front end of a queue.
- Rear(Q) – Element *e* in the rear end of a queue.

Axioms:

- Isempty(Create-Queue()): Always True (Boolean value).
- Isfull(Create-Queue()): Always False (Boolean value).
- Isempty(Insert(Q, *e*)): Always False (Boolean value).
- Isfull(Delete(Q)): Always False (Boolean value).
- Rear(Insert(Q, *e*)): The same element e is displayed.
- If (Front(Q) = *a*) then Delete(Q) = *a* i.e. Front(Q) = Delete(Q)

Thus a stack Q(D, F, A) can be viewed as an ADT.

Array Implementation of a Queue

As discussed for stacks, an array of *n* locations can be defined for realizing a static Queue. The Figure 3.2 illustrates an array based implementation of a Queue. The figure shows an empty queue of size 10 elements. Since insertions and deletions are performed at two different ends, two pointers namely *front* and *rear* are maintained to realize Queue operations. We insert an element into a queue using *rear* pointer and delete an element from a queue using *front* pointer. Generally, we can define a queue of size of *n* elements depending on the requirement. Therefore, it becomes necessary to signal "QUEUE OVERFLOW" when we attempt to store more than *n* elements in to a queue and "QUEUE UNDERFLOW" when we try to remove an element from an empty queue. As shown in the figure 4.2, initially, the two pointers *front* and *rear* are initialized to -1.

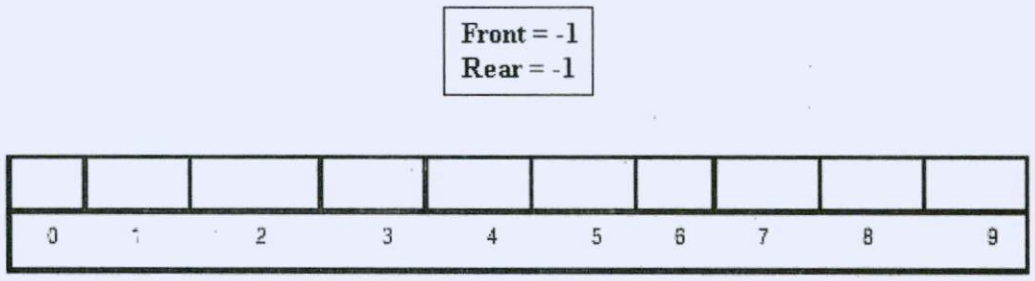


Figure 4.2 An empty Queue of size 10 elements.

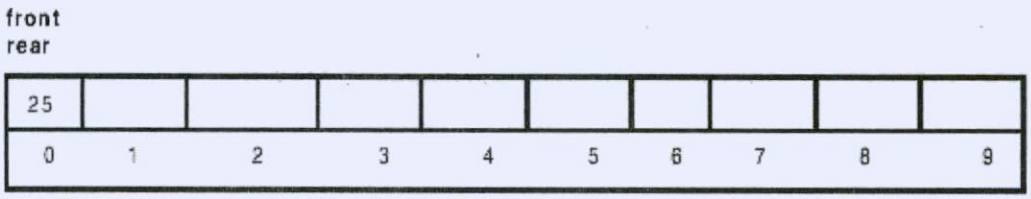


Figure 4.3 Queue with an insertion of an element 25.

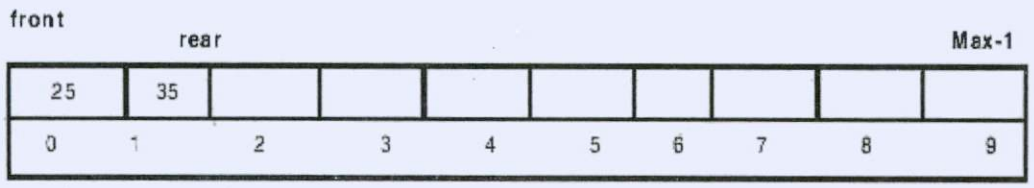


Figure 4.4 Queue with an insertion of two elements 25 and 35.

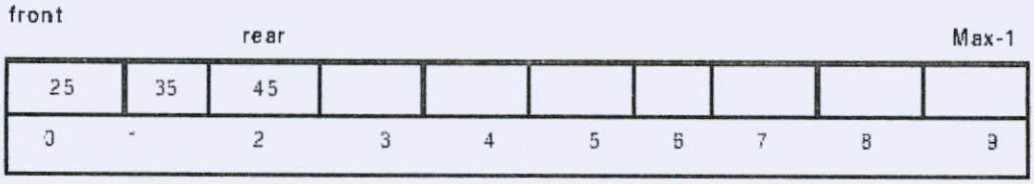


Figure 4.5 Queue with an insertion of three elements 25, 35 and 45.

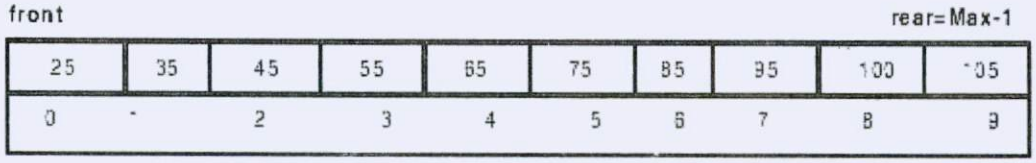


Figure 4.6 Queue is full with insertion of 10 elements.

Look at the Fig. 3.2 above, initially the queue is empty. The condition ***front*** = **-1**and ***rear*** = **-1**initially implies that queueis empty. In Fig. 4.3, element 25 is inserted as first element and both *front* and *rear* are assigned to firstposition. Fig. 4.4and4.5 show the position of *rear* and *front* with each addition to queue. Note thatin Fig. 4.3, when first element is inserted, we have incremented *front* by 1, after that front is neverchanged during addition to queue process. Observe that *rear* is incremented by one prior to adding anelement to the queue; we call this process as ***enque.*** However to ensure that we do not exceed theMax length of the queue, prior to incrementing rear we have to check if the queue is fu is can beeasily achieved by checking the condition ***if (rear = = MAX-1)***.

In a queue, elements are always removed from the front end. The *front* pointer is front incremented by one after removing an element. Fig. 4.7 shows status of *front* and *rear*pointers after twodeletions from Fig. 4.5.

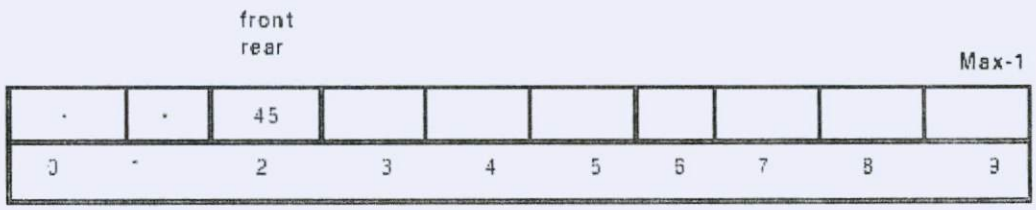


Figure 4.7 Queue after deleting two elements 25 and 35 from Figure 3.5.

Observe that the positions vacated by front marked as * are unusable by queueany further. This is one of the major draw back of linear queue array representation. In Fig. 4.8, wehave carried out three deletions for 25,35,and 45. After deleting 45, we find queue is empty and thecondition for ***this queue state is rear is less than front***. At this stage, we can reinitialize the queueby setting ***front*** = **-1** and ***rear*** = **-1** and thus reclaim the unused positions marked with *.

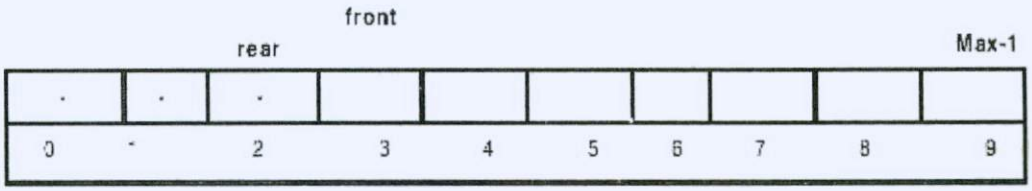


Figure 4.8 Queue after deleting three elements 25, 35 and 45 from Figure 3.5.

Note: Number of elements in the queue at any instance is ***rear*** – ***front*** + **1**.

6.5 Queue Application

We can realize the basic functions related to queue operations using any programming language. The following algorithms will help us to realize the basic functions.

Algorithm: Create-Queue()

Input: Size of queue, ***n***.

Output: Queue **Q** created.

Method: 1. Declare an array of size ***n***.

2. Initialize the pointers ***front*** and ***rear*** to -1.

3. Return.

Algorithm ends.

Algorithm: Isempty(Q)

Input: Queue **Q** and a pointer ***front***.

Output: True or False.

Method: 1. If (*front* = -1) Then return (True)

Else return (False)

Algorithm ends.

Algorithm: Isfull(Q)

Input: Queue **Q** and a pointer ***rear***.

Output: True or False.

Method: 1. If (***rear*= *n* - 1**) Then return (True)

Else return (False)

Algorithm ends.

Algorithm: Rear(Q)

Input: Queue **Q** and a pointer ***rear***.

Output: Element in the rear position of Queue.

Method: 1.If (Isempty(Q)) Then Display "STACK EMPTY"

Else return (Q(***rear***))

Algorithm ends.

Algorithm: Front(Q)

Input: Queue **Q** and a pointer ***front***.

Output: Element in the front position of Queue.

Method: 1.If (Isempty(Q)) Then Display "STACK EMPTY"

Else return (Q(***front***))

Algorithm ends.

Algorithm: Insert(Q)

Input: Queue **Q** and element **e** to be inserted.

Output: Queue **Q** updated.

Method: 1. If (Isfull(S)) Then Display "QUEUE OVERFLOW"

Else

rear = *rear* + 1

Q(*rear*) = e.

If (*front* = -1) Then *front* = 0

2. Return.

Algorithm ends.

Algorithm: Delete(Q)

Input: Queue **Q**.

Output: Element ***e*** deleted.

Method: 1. If (Isempty(S)) Then Display "QUEUE EMPTY"

Else

e = Q(*front*)

If (*front* = *rear*) Then *front* = *rear* = -1

Else *front* = *front* + 1

If-end

If-end

2. Return.

Algorithm ends.

Algorithm: Isfull(Q)

Input: Queue **Q** and a pointer ***rear***.

Output: True or False.

Method: 1. If ((***rear*** mod ***n*+1**) = ***front***) Then return (True)

Else return (False)

Algorithm ends.

Algorithm: Insert(Q)

Input: Queue **Q** and element **e** to be inserted.

Output: Queue **Q** updated.

Method: 1. If (Isfull(S)) Then Display "QUEUE OVERFLOW"

Else

rear = *rear* mod (*n* + 1)

Q(*rear*) = e.

If (*front* = -1) Then *front* = 0

2. Return.

Algorithm ends.

Algorithm: Delete(Q)

Input: Queue **Q**.

Output: Element ***e*** deleted.

Method: 1. If (Isempty(S)) Then Display "QUEUE EMPTY"

Else

e = Q(*front*)

If (*front* = *rear*) Then *front* = *rear* = -1

Else *front* = *front* mod (*n*+ 1)

If-end

If-end

2. Return.

Algorithm ends.

6.6 Summary

- Queues and priority queues, like stacks, are data structure usually used to simplifycertain programming operations.
- In these data structures, only one data item can be immediately accessed.
- A queue, in general, allows access to the first item that was inserted.
- The important queue operations are inserting an item at the rear of the queue andremoving the item from the front of the queue.
- A queue can be implemented as a circular queue, which is based on an array inwhich the indices wrap around from the end of the array to the beginning.
- A priority queue allows access to the smallest (or sometimes the largest) item in the queue.
- The important priority queue operations are inserting an item in sorted order andremoving the item with the smallest key.
- Queues finds its applications in implementing job scheduling algorithms, page replacement algorithms, interrupt handling mechanisms etc. in the design of operating systems.

6.7 Keywords

(1). Queue

(2). Linear queue

(3). Circular queue

(4). Doubly ended queue

(5). Priority queue

6.8 Questions for self study

(1). Explain the basic operations of queue.

(2). Mention the limitation of linear queue with a suitable example

(3). Given a circular array of size n with f and r being the front and rear pointer, work out the mathematical functions to calculate the number of elements present in a circular queue. Note : Consider the f and r at different positions i.e., (i) f < r (ii) f > r and (iii) f = r

(4). What is Dequeue? Explain different types of Dequeue with suitable example

(5). Design an algorithm to insert and delete elements into deletion restricted De-Queue.

(6). What are priority queues? Discuss its applications.

6.9 References

(1). Jean-Paul Tremblay, Paul G. Sorenson, P. G. Sorenson:An Introduction to Data Structures with Applications. Mcgraw-Hill College; 2nd edition (1984).

(2). Tenenbaum, Langsam, Augenstein. Data Structures Using C. phi

(3). Alfred V. Aho , Jeffrey D. Ullman, John E. Hopcroft. Data Structures and Algorithms. Addison Wesley (January 11, 1983)

Unit-7

Linked Lists, Some general linked list operations

Structure:

7.0 Objectives

After studying this unit, we will be able to explain the following:

- Concept of Linked allocation.
- Advantages of linked allocation over sequential allocation.
- Linked list data structure.
- Types of linked lists.
- Implementation of linked lists.
- Various operations on linked lists.

7.1 Introduction

As discussed in earlier chapters, the linear data structures such as stacks and queues can be realized using sequential allocation technique i.e. arrays. Since arrays represent contiguous locations in memory, implementing stacks and queues using arrays offers several advantages.

- Accessing data is faster. Since arrays works on computed addressing, direct addressing of data is possible and hence data access time is constant and faster.
- Arrays are easy to understand and use. Hence implementing stacks and queues is simple and straight forward.
- In arrays, the data, which are logically adjacent are also physically adjacent and hence a loss of data element does not affect other part of the list.

However, the sequential allocation technique i.e. arrays suffers from serious drawbacks.

- Arrays are static in nature. A fixed size of memory is allocated for a program before execution and cannot be changed during its execution. The amount of memory required by most of the applications cannot be predicted in advance. If all the memory allocated to an application is not utilized then it results in wastage of memory space. If an application requires more than the memory allocated then additional amount of memory cannot be allocated during execution.
- For realizing linear data structures using arrays, sufficient amount of contiguous storage space is required. Sometimes, even though enough storage space is available in the memory as chunks of contiguous locations, it can be used because they are not contiguous.
- Insertion and deletion operations are time consuming and tedious tasks, if linear data structures are implemented using contiguous allocation technique – arrays.

In order to overcome the above limitations of contiguous allocation techniques, the linear data structures such as stacks and queues can be implemented using linked allocation technique. Linked list structures can be used to realize linear data structures efficiently and is a versatile mechanism suitable for use in many kinds ofgeneral-purpose data-storage applications.

7.2Linked lists

A linked representation of a data structure known as a *linked list* is a collection of *nodes*. Each node is a collection of fields categorized as *data items* and *links*. The data item fields hold the information content or the data to be represented by the node. The link fields hold the addresses of the neighboring nodes or the nodes associated with the given node as dictated by the application. Figure 7.1(a) below, shows the general structure of a node in a linked list, where the field, which holds data item is called as *Info* and the field, which holds the address is called as a *Link* and Figure 7.1(b) shows an instance of a node where the info field contains an integer number **90** and the address field contains the address **2468** of the next node in a list. Figure 7.2 shows an example linked list of integer numbers.

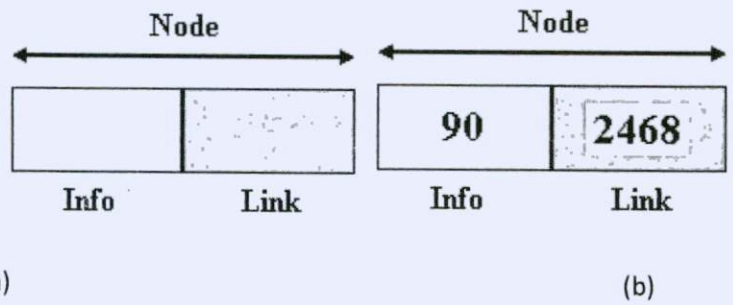


Figure 7.1 (a) Structure of a node (b) An instance of a node

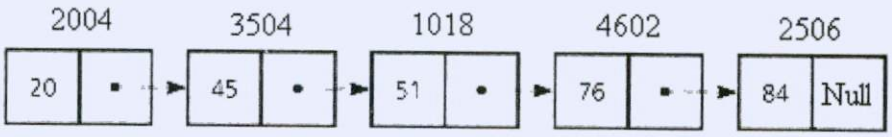


Figure 7.2 Pictorial representation of a linked list.

From the Figure 7.2, we have observed that unlike arrays no two nodes in a linked list need be physically contiguous. All the nodes in a linked list data structure may in fact be strewn across the storage memory making effective use of a little available space to represent a node.

In order to implement linked lists, we need to use the following mechanisms:

- A mechanism to frame chunks of memory into nodes with the desired number of data items and fields.

- A mechanism to determine which nodes are free and which nodes have been allocated for use.
- A mechanism to obtain nodes from the free storage area for use.
- A mechanism to return or dispose nodes from the reserved area to the free area after use.

We can realize the above mechanisms using the features like records or structures to define nodes and dynamic memory allocation functions to get chunks of memory and other related functions supported by most of the programming languages.

Irrespective of the number of data items fields, a linked list is categorized as singly linked list, doubly linked list, circularly singly linked list, circularly doubly linked list.

7.3General Linked List Operations

Irrespective of the types of linked list, we can perform some general operations such as

- Creation of a linked list.
- Displaying the contents of a linked list.
- Inserting an element at the front or at the rear end of a linked list.
- Inserting an element at the specified position of a linked list.
- Inserting an element in to a sorted linked list.
- Deleting an element from a linked list at the front or at the rear end.
- Deleting an element form a linked list at the specified position.
- Deleting an element *e* from a linked list if present.
- Searching for an element *e* in the linked list.
- Reversing a linked list.
- Concatenation of two linked lists.
- Merging of two ordered linked lists.
- Splitting a linked list.

7.Creation of a linked list:

Creation of a linked list is fairly a simple task. The steps include getting a free node, copying the data item into the information field of a node and updating the link field depending on whether the new node is inserted at the beginning or at the end.

2. Displaying the contents of a linked list:

To display the contents of a linked list, we start with the pointer containing the address of the first node of a linked list and follow the links to visit each node and to display the data item contained in the information field of a node. We continue the process until the last node of the list is reached. Since the link field of the last node of a linked list contains the NULL value, the process ends indicating the end of a linked list.

7. Inserting an element at the front:

Inserting an element into a linked list at the front is a simple task. First, we need to verify whether the list to which we are inserting an element is empty or not. This is done by verifying the pointer, which contains the address of the first node in the linked list. If this pointer contains the value NULL then the list is empty otherwise it is not empty.

The following sequence of steps is used to insert an element into a linked list at the front end when the list is empty:

1. Get a free node.
2. Copy the data into information field of the new node.
3. Copy the address of the new node into the pointer pointing to the list.
4. Place the NULL value into the link field of the node.

The following sequence of steps is used to insert an element into a linked list at the front end when the list is not empty:

1. Get a free node.
2. Copy the data into information field of the new node.
3. Copy the address contained in the pointer to the link field of the new node.
4. Copy the address of the new node into the pointer
5. Place the NULL value into the link field of the new node.

4. Inserting an element at the rear:

To insert an element into a linked list at the rear end, we need to consider two cases (i) when the list is empty (ii) when the list contains some elements. The first case is similar to inserting an element to a list at front when the list is empty. But in the second case, we need to traverse through the linked

list until we encounter the last node of the linked list and the following steps are needed to insert an element.

1. Get a free node.
2. Copy the data into information field of the new node.
3. Copy the address of the new node into link field of the last node.
4. Place the NULL value into the link field of the new node.

5. Inserting an element at the specified position of a linked list:

To insert an element into a linked list at the specified position, we need to traverse through the linked list until we find the right position. The position of the node to be inserted is tracked using a temporary pointer variable. Once the right position is found, the following steps are needed to insert an element at that position.

1. Get a free node.
2. Copy the data into information field of the new node.
3. Copy the address contained in the link field of the temporary pointer variable into the link field of the new node.
4. Copy the address of the new node into link field of the node pointed by the temporary pointer variable.

6.Inserting an element in to a sorted linked list:

To insert an element into a sorted linked list, we may come across the following three cases:

(i) Inserting at the front end: This is same as discussed earlier above.

(ii) Inserting at the rear end: This is same as discussed earlier above.

(iii) Inserting in the middle: This case involves traversal of a linked list from the beginning, comparing the data item stored in the information field of the node with the data item contained in the information field of the node to be inserted. If the linked list is sorted in ascending order, the process stops when the data item to be inserted is less than the data item contained in the linked list node and the node is inserted at that position. Similarly, if the linked list is sorted in descending order, the process stops when the data item to be inserted is greater than or equal to the data item contained in the linked list node and the node with the data item is inserted at that position.

7. Deleting an element from a linked list at the front or at the rear end:

Deleting an item form a linked list at the front is a simple and a straight forward task. The pointer pointing to the first node of a linked is made to point to the next node. Thus by updating the address stored in the pointer, we can remove the first node from a linked list.

Similarly, to delete a node from a linked list at the rear end, we need to traverse the linked list till we reach end of the list. However, it is necessary to remember the node preceding to the last node using a pointer. We delete the current node by placing the NULL value in the link field of the preceding node.

8. Deleting an element form a linked list at the specified position:

In order to accomplish the above task, we traverse a linked list form the beginning using a temporary pointer to identify the position in the linked list. Once the position is found, we remove the node at this position by placing the address contained in the link field of the current node into the link field of the preceding node. A pointer is maintained to keep tack of the preceding node.

9. Deleting an element *e*form a linked list:

To delete an element *e* from a linked list, we traverse a linked list form the beginning using a temporary pointer searching for a node containing the element *e*. We do so by comparing the data item stored in the information field of a node with the element *e*. If we found the node, then we remove it by copying the address stored in the link field of the node to the link field of the previous node. A pointer is maintained to keep track of the previous node.

10. Reversing a linked list:

Reversing a linked list means obtaining a new linked list from a given linked list such that the first node of an original list becomes the last node in new list, second node becomes the last but one node and so on. Finally, last node of the original list becomes the first node in the reversed linked list.

17. Concatenation of two linked lists:

Concatenation of two linked lists is nothing but joining the second list at the end of the first list. To accomplish this task, we traverse the first list from the beginning to the end using a temporary pointer variable and copying the address of the first node of the second list to the link field of the last node of the first list.

12. Merging of two ordered linked lists.

Given two ordered list, we can obtain a single ordered list by merging the two lists. To perform this task, we need to maintain three pointers one for list-1, one for list-2 and one for the merged list. We then traverse list-1 and list-2 from the beginning comparing the information content of the first node of the first list with the information content of the first node of the second list. If the data item contained in the first node of the first list is less than the data item contained in the first node of the second list then the address of the first node of the first list is copied to the pointer pointing to the merged list and the pointer pointing to the first list is incremented to point to next node otherwise the address of the first node of the second list is copied to the pointer pointing to the merged list and the pointer pointing to the second list is incremented to point to next node. This process is continued until we reach the end of first list or the end of second list. If we encounter the end of first list before the second list then the remaining nodes of the second list are directly copied to the merged list. Otherwise, the remaining nodes of the first list are directly copied to the merged list.

17. Splitting a linked list:

Splitting a linked list means obtaining two lists form the given single list. We can split a list either by specifying the position of the node or the information content of the node. To split a list at a specified position like 1, 2, 3, etc., we need to traverse a linked list to find the position of the node to split. Once the position of the node is found, the address of this node is copied to a pointer variable, which points to the second list and a NULL value is copied to the link field of the node previous to the current node. A Similar procedure is followed to split a list based on the information content of the node. But in this case, we traverse a linked list, searching for a node, which contains the required data item and then we split the list as described above.

7.4Static allocation Vs Linked allocation

Static allocation technique	Linked allocation technique
1. Memory is allocated during compile time.	1. Memory is allocated during execution time.
2. The size of the memory allocated is fixed.	2. The size of the memory allocated may vary.
3. Suitable for applications where data size is fixed and known in advance.	7. Suitable for applications where data size is unpredictable.
4. Execution is faster.	4. Execution is slow.
5. Insertion and deletion operations are strictly not defined. It can be done conceptually but inefficient way.	5. Insertion and deletion operations are defined and can be done more efficiently.

Summary

- Sequential data structures suffer from the drawbacks of inefficient implementation of insert/delete operations and inefficient usage of memory.
- A linked representation serves to rectify these drawbacks. However, it calls for the implementation of mechanisms such as Getnode() and Free() to reserve a node for use and return the node to the available list of memory after use, respectively.
- For certain applications linked allocation is more useful and efficient and for some other applications sequential allocation is more useful and efficient.

Keywords

Lineardata structure, Linked lists, Sequential allocation, Dynamic allocation.

Questions for self study

1. What are linear data structures? Explain briefly.
2. What is meant by sequential allocation? Mention the advantages and disadvantages of sequential allocation.
3. What are linked lists? Explain with an illustrative example.
4. What is the need for linked representation of lists?
5. What are advantages of linked representation over sequential representation? Explain briefly.
6. Mention the various operations that can be performed on linked lists.
7. Describe the insertion and deletion operations on linked lists considering all possible cases.
8. Briefly describe the merging and splitting operations on linked lists.

References

1. Sams Teach Your Self Data Structuresand Algorithmsin24Hours.
2. C and Data Structures by Practice- Ramesh, Anand and Gautham.
3. Data Structures Demystified by Jim Keogh and Ken Davidson. McGraw-Hill/Osborne © 2004.
4. Data Structures and Algorithms: Concepts, Techniques and Applications by GAV Pai. Tata McGraw Hill, New Delhi.

Unit-8

Application of linked lists : Polynomial operations, Sparse Matrix multiplication associated algorithms

Structure:

8.1 Application of Linked Lists

Is there some real life, intuitive example of the plain ole' singly Linked List like we have with inheritance? The typical textbook Linked List example shows a node with an integer and a pointer to the next, and it just doesn't seem very useful. The main Applications of Linked Lists are : In Dynamic Memory management , In Symbol Tables, Representing Sparse Matrix. For representing Polynomials. It means in addition/subtraction /multipication.. of two polynimials.

8.2 Polynomial Operations

Eg:p1=2x^2+3x+7 and p2=3x^3+5x+2

p1+p2=3x^3+2x^2+8x+9

* In Dynamic Memory Management

In allocation and releasing memory at runtime.

*In Symbol Tables

in Balancing paranthesis

* Representing Sparse Matrix

8.2 Polynomial Operations

Sparse matrices, which are common in scientific applications, are matrices in which most elements are zero. To save space and running time it is critical to only store the nonzero elements. A standard representation of sparse matrices in sequential languages is to use an array with one element per row each of which contains a linked-list of the nonzero values in that row along with their column number. A similar representation can be used in parallel. In NESL a sparse matrix can be represented as a sequence of rows, each of which is a sequence of `(column-number, value)` pairs of the nonzero values in the row. The matrix

8.4 Associated Algorithms

We have briefly described the various operations that can be performed on linked list, in general, in Unit-I. In fact, we can realize all those operations in singly linked list. The following sections describe the logical representation of all these operations along with their associated algorithms.

1. Inserting a node at the front end of the list:

Algorithm: Insert-First-SLL

Input: F, address of first node.

e, element to be inserted.

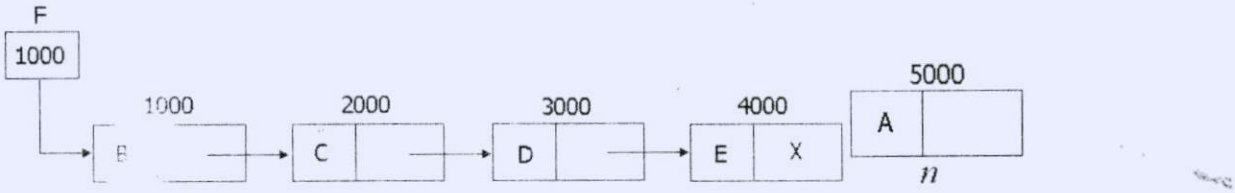
Output: F, updated.

Method:

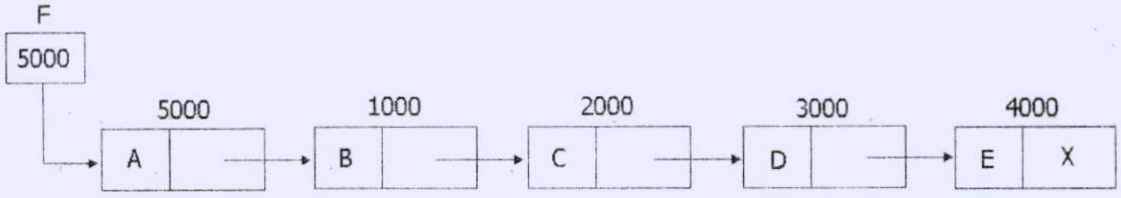
1. *n*= getnode()
2. *n*.data = *e*
3. *n*.link = F
4. F = *n*

Algorithm ends

In the above algorithm, *n* indicates the node to be added into the list. The getnode() function allocates the required memory locations to a node *n* and assigns the address to it. The statement *n*.data = *e* copies the element into the data field of the node *n*. The statement *n*.link = F copies the address of the first node into the link field of the node *n*. Finally, the address of the node n is placed in the pointer variable F, which always points to the first node of the list. Thus an element is inserted into a singly linked list at the front. The Figure 3.3 below shows the logical representation of the above operation. In figure 3.3(a) the pointer F contains the address of the first node i.e. 1000. Figure 3.3(b) shows the node *n* to be inserted, which has its address 5000 holding the data A. Figure 3.3(c) shows the linked list after inserting the node *n* at the front end of the list. Now, F contains the address of the new node i.e. 5000.



(a) Linked list before insertion (b) Node to be inserted.



(c) Linked list after insertion

Figure 3.3 Logical representation of the insertion operation.

2. Inserting a node at the rear end of the list:

Algorithm: Insert-Last-SLL

Input: F, address of first node.

e, element to be inserted.

Output: F, updated.

Method: 1. *n*= getnode()

5.*n*.data = *e*

6.*n*.link = Null

7. if (F = Null) then F = *n*

else

T = F

While (T.link ≠ Null) Do

T = T.Link

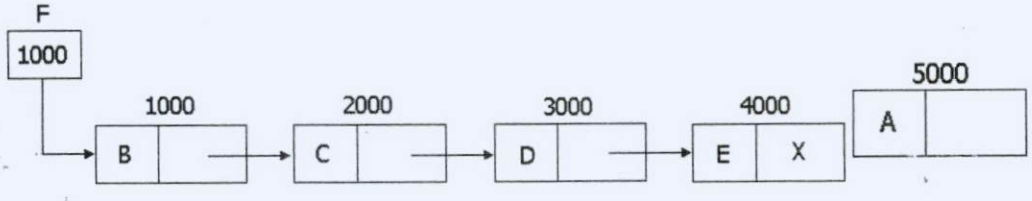
End-of-While

T.Link = *n*

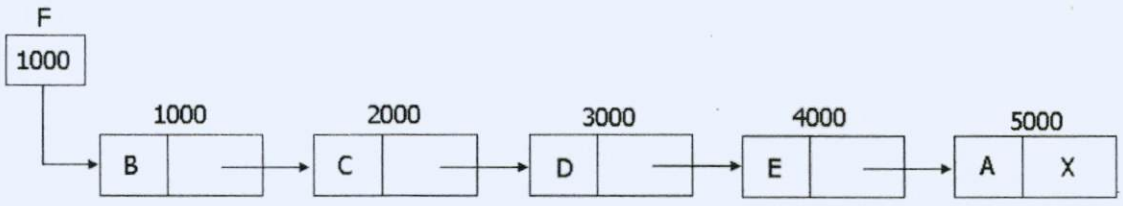
Endif

Algorithm ends

Algorithm above illustrates the insertion of node at the end of a singly linked list. Observe that the link field of the node to be inserted is Null. This is because the inserted node becomes the end of the list, whose link field always contains the Null value. If the list is empty, the inserted node becomes the first node and hence the pointer F will hold the address of this inserted node. Otherwise, the list is traversed using a temporary variable till it reaches the end of the list. This is accomplished using a statement T = T.Link. Once the last node is found, the address of the new node is copied to the link field. Figure 3.4 below shows the logical representation of inserting a node at the end of the linked list.



(a) Linked list before insertion (b) Node to be inserted



(c) Linked list after insertion

Figure 3.4 Logical representation of the insertion operation

3. Inserting a node into a sorted singly linked list:

Algorithm: Insert-Sorted-SLL

Input: F, address of first node.

e, element to be inserted.

Output: F, updated.

Method: 1. *n* = getnode()

2. *n*.data = *e*

3. if (F = Null) then *n*.link = Null

F = *n*

else

If (e ≤ F.data) then n.link = F

F = n

Else

T = F

While (T.link ≠ Null) and (T.link.data < e) Do

T = T.Link

End-of-While

n.link = T.link

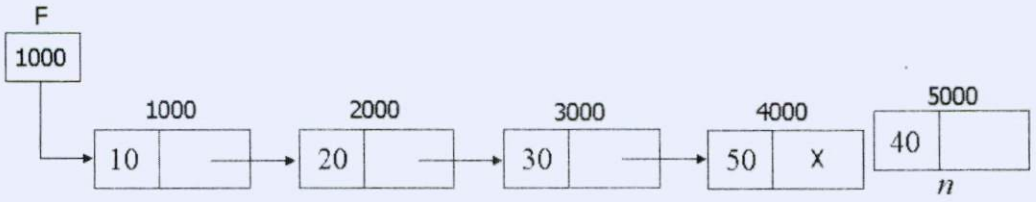
T.link = *n*

Endif

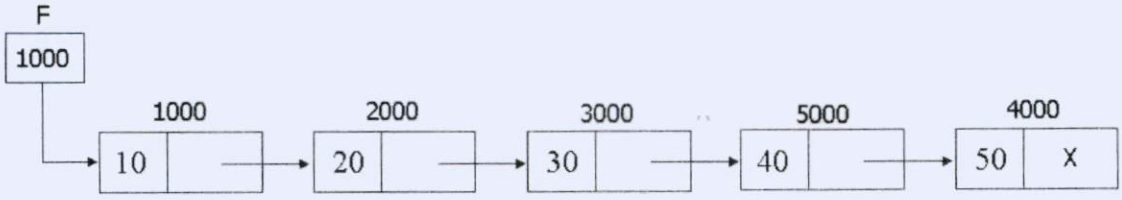
Endif

Algorithm ends

The above algorithm illustrates the operation of inserting an element into a sorted linked list. This operation requires the information about the immediate neighbour of the current element. This is done using the statement T.link.data. This is because, we need to insert an element between two nodes. The rest of the statements are self explanatory and can be traced with an example for better understanding of the algorithm. Figure 3.5 below shows the logical representation of this operation.



(a) Linked list before insertion (b) Node to be inserted.



(c) Linked list after insertion

Figure 3.5 Logical representation of the insertion operation.

4. Deleting a first node from a singly linked list:

Algorithm: Delete-First-SLL

Input: F, address of the first node.

Output: F, updated.

Method:

1. if (F = Null) then Display "List is empty"

   else

   T = F

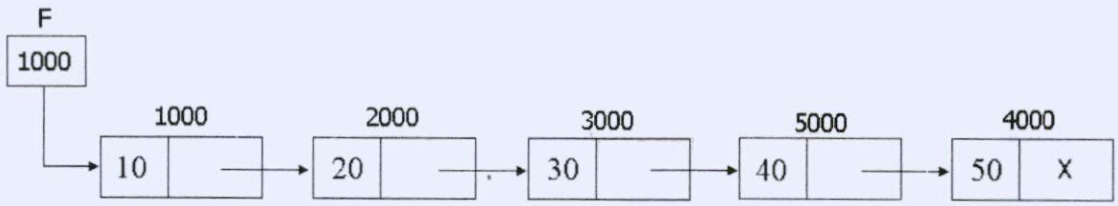
F = F.link

Dispose(T)

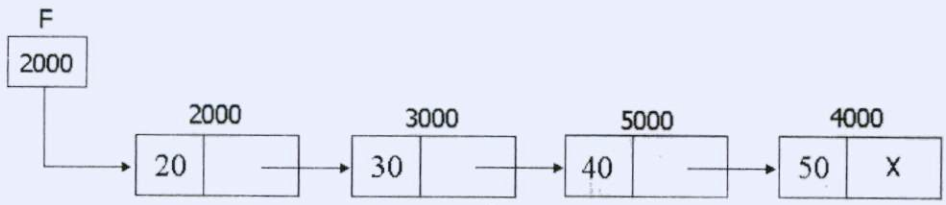
Ifend

Algorithm ends

The above Algorithm is self explanatory. The function Dispose(T) is a logical deletion. It releases the memory occupied by the node and freed memory is added to available memory list. Figure 3.6 below shows the logical representation of this operation.



(a) Linked list before deletion



(a) Linked list after deletion

Figure 3.6 Logical representation of the deletion operation

5. Deleting a node at the end of a singly linked list:

Algorithm: Delete-Last-SLL

Input: F, address of the first node.

Output: F, updated.

Method: 1. if (F = Null) then Display " List is empty"

else

If (F.link = Null) then Dispose(F)

F = Null

else

T = F

While ((T.link).link ≠ Null) Do

T = T.Link

End-of-While

T.link = Null

T.link = *n*

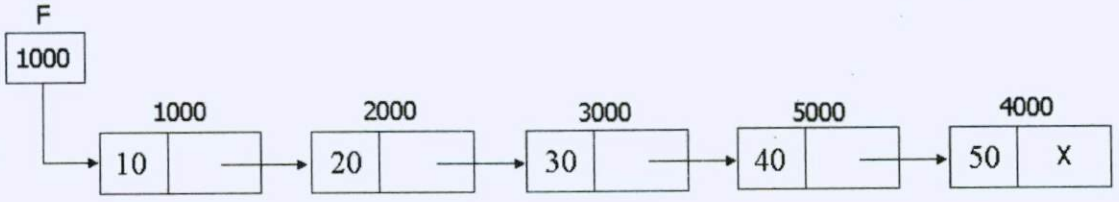
endif

endif

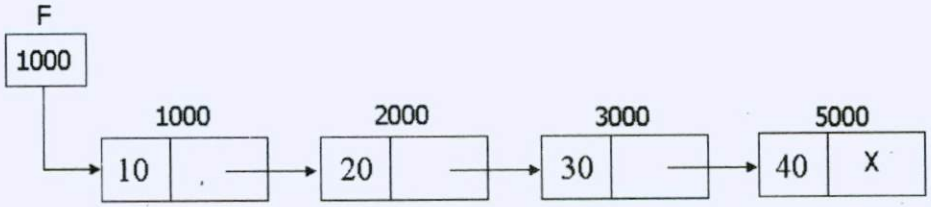
Algorithm ends

From the algorithm above, we understand that we may come across three cases while deleting an element from a list. When the list is empty, we just display an appropriate message. When a list contains only one element, we remove it and the pointer holding the address of the first node will be assigned a Null value. When a list contains more than one element, we traverse through the linked list to reach the

end of the linked list. Since we need to update the link field of the node, predecessor to the node to be deleted, it requires checking the link field of the next node being in the current node. This is accomplished using the statement (T.link).link. Figure 3.7 below shows the logical representation of this operation.



(a) Linked list before deletion



(a) Linked list after deletion

Figure 3.7 Logical representation of the deletion operation

6. Deleting a node with a given element from a singly linked list:

Algorithm: Delete-Element-SLL

Input: F, address of the first node.

e, element to be deleted.

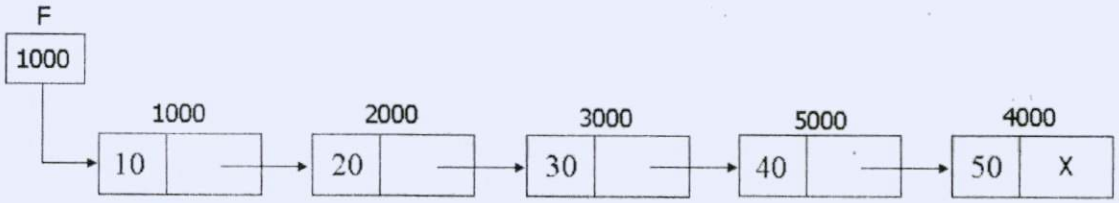
Output: F, updated.

Method:

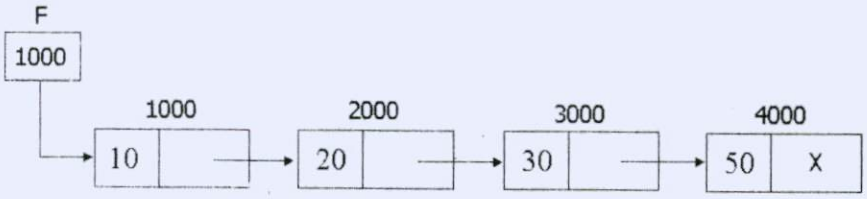
```
1. if (F = Null) then Display " List is empty"
   else
      if (F.data = e) then F = F.link
      else
            T = F
            While (((T.link).link ≠ Null) and ((T.link).data) ≠ e) Do
                  T = T.Link
            End-of-While
            if (T.link = Null) then Display " Element e is not found"
            else
                  V = T.link
                  T.link = (T.link).link
                  Dispose(V)
            endif
   endif
endif
```

Algorithm ends

The above algorithm is self explanatory as described in the earlier algorithm. Figure 3.8 below shows the logical representation of this operation.



(a) Linked list before deletion



(a) Linked list after deleting a node containing element 40

Note: The combination of **Insertion-First-SLL** and **Deletion-First-SLL** or the combination of **Insert-Last-SLL** and **Delete-Last-SLL** can be used to realize the basic stack operation using singly linked list.

8.5 Summary

- A singly linked list is the simplest of a linked representation.
- We can realize the operations of stack data structure using linked list functions: insert-rear(), delete-front() and display().
- We can realize the operations of queue data structure using linked list functions: insert-front(), delete-rear() and display().

8.6 Keywords

Lineardata structure, Singly Linked lists, Dynamic allocation.

8.7 Questions for self study

1. What are singly linked lists? Explain with an illustrative example.
2. Mention the various operations performed on singly linked lists.
3. Briefly explain the implementation of stack using singly linked list.
4. Briefly explain the implementation of queue using singly linked list.
5. Design an algorithm to delete all the nodes with a given element from a singly linked list.
6. Design an algorithm to delete alternative occurrence of an element from a singly linked list.
7. Design an algorithm to delete an element e from a sorted singly linked list.
8. Design an algorithm to delete all occurrence of an element from a sorted singly linked list.
9. Design an algorithm to reverse a given singly linked list.
10. Bring out the differences between sequential and linked allocation techniques.

8.8 References

5. Sams Teach Your Self Data Structuresand Algorithmsin24Hours.
6. C and Data Structures by Practice- Ramesh, Anand and Gautham.
7. Data Structures Demystified by Jim Keogh and Ken Davidson. McGraw-Hill/Osborne © 2004.
8. Data Structures and Algorithms: Concepts, Techniques and Applications by GAV Pai. Tata McGraw Hill, New Delhi.

Unit – 9

Concept of Divide and Conquer Strategy

STRUCTURE

9.0 OBJECTIVES

After studying this unit you should be able to

- Explain the divide-and-conquer methodology for solving problems.
- Formulate divide-and-conquer strategyto solve a large problem.
- Critically evaluate the complexity of divide-and-conquer algorithms.

9.1 INTRODUCTION

The divide-and-conquer strategy so successfully used by monarchs and colonizers may also be applied to the development of efficient computer algorithms. In this context, divide-and-conquer is a technique for designing algorithms that consists of decomposingthe instance to be solved into a number of smaller subinstances of the sameproblem, solving successively and independently each of these subinstances, andthen combining the subsolutions thus obtained to obtain the solution of the originalinstance. Two questions that naturally spring to mind are "Why would anyonewant to do this?" and "How should we solve the subinstances?" The efficiency ofthe divide-and-conquer technique lies in the answer to this latter question.

9.2 GENERAL STRUCTURE

More formally, given a function, to compute on n-inputs, we split the inputs into k distinct subsets (distinct because no two subsets share the same data, or each of the n input values will be in exactly one subset), $1 \leq k \leq n$. This gives us k sub problems, identical to the original problem but with lesser of inputs. Each of them is solved and the sub solutions are recombined to get the originally required solution. However, combining need not always mean, simply the union of individual solutions it can even be the selecting a subset from the set of solutions.

Generally, Divide and Conquer involves four steps

1) Divide
2) Conquer [Initial Conquer]
3) Combine
4) Conquer [Final Conquer].

In precise, forward journey is divide and backward journey is Conquer. A general binary divide and conquer algorithm is:

Algorithm: Divide and Conquer

Input: P, Q, the lower and upper limits of the data size

Output: solution

Method:

```
If (SMALL(P,Q))
    Solve(P,Q)
Else
    M ← divide(P,Q)
    Combine (Divide and Conquer (P,M), Divide and Conquer (M+1,Q))
If end
```

Algorithm ends.

Sometimes, this type of algorithm is known as control abstract algorithms as they give an abstract flow. This control abstraction, apart from giving us the framework of divide and conquer methodology also gives us a methodology that we follow repeatedly during this course. SMALL is a function which takes two parameters P and Q and decides whether the input size (Q-P+1) is small enough so that the answer can be computed without further splitting. How exactly is this decided depends on the particular problem on hand, but what we look at this stage is that, given the lower limit P and upper limit Q of a given problem this Boolean function can decide whether the problem needs to be split further or not. If the problem is small enough, then the function SMALL returns a value "True" and then the problem is solved by invoking a solution yielding function SOLVE. If the problem is not small enough, then SMALL returns 'False' and the problem is divided further. This is done by another function DIVIDE, which returns a value *m*, such that the given problem with the range from P to Q is divided into two parts from P to *m* and*m*+1 to Q. Now the actual solution of the problem is the combination of the solution values from P to *m* and from *m*+1 to Q. Of course, if these are not small enough, they are divided again recursively. Finally the solutions of all these are combined to get back the required solution. This way of breaking down the problem has found wide application in sorting, selection and searching algorithms.

. Now, we introduce another property, which we keep using repeatedly in this course i.e., analyzing the complexity of the algorithms under divide and conquer strategy. Let us say, there are *n* parameters to be taken care of. Now, if there is a method which can work for solving by considering all at once then n is obviously small. Otherwise, if *n* is big, then *n* will be divided

into two parts. Common sense tells us it will be two equal parts (this need not always be true), if possible these parts are solved separately and recombined. Let us say the function T(n) gives us the time required to solve the problem for n values then T(n) = g(n) if n is small, where, g(n) is the time required to complete the task using the given algorithm. Otherwise, T(n) = 2T(n/2) + f(n) if n is not small enough.

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ is small} \\ 2T\left(\frac{n}{2}\right) + f(n) & \text{Otherwise} \end{cases}$$

These types of representations are called recurrence relations. In this case, we divide n into two parts of size n/2 each and try to solve them separately. So, each part needs T(n/2) units of time. Since the data set is divided into two parts, it takes 2 x T(n/2) units. But, we also need time to divide the problem into two parts using DIVIDE and time to combine the solutions using COMBINE, So, an additional time f(n) is required. So, the total time needed will be 2T(n/2) + f(n).

Note that we are using T(n/2), not g(n/2), because we are not yet sure, whether n/2 is small enough. If not, it is further divided to two parts each, so that the total becomes four parts of size n/4 each and so on. For each break and combine operation, additional time f( ) is to be spent.

Since divide-and-conquer algorithms decompose a problem instance into several smaller independent instances, divide-and-conquer algorithms may be effectively run on a parallel computer: the independent smaller instances can be worked on by different processors of the parallel computer.

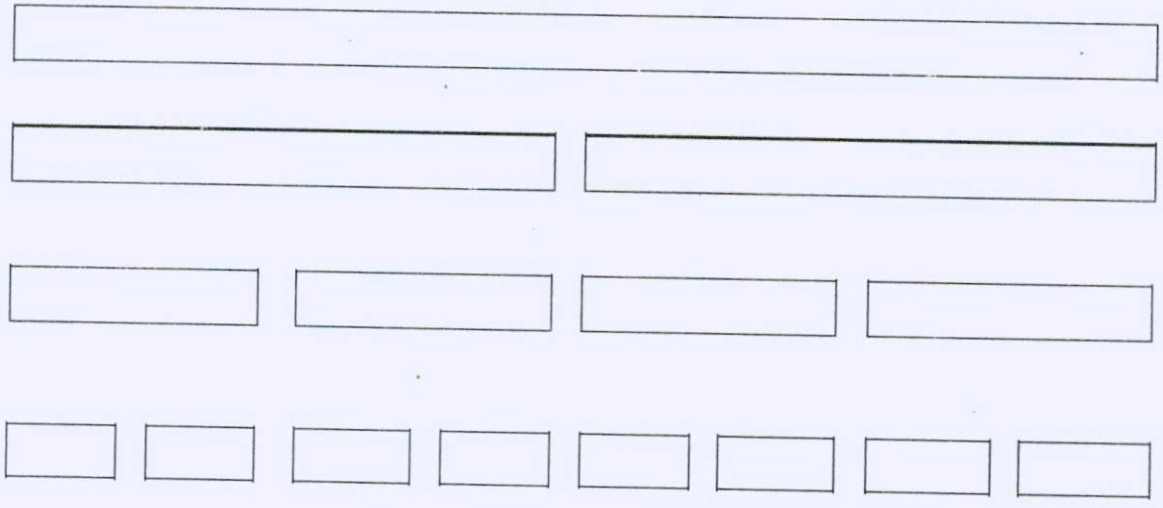
This module develops the mathematics needed to analyze the complexity of frequently occurring divide-and-conquer algorithms and proves that the divide-and-conquer algorithms for the minmax problem is optimal by deriving lower bounds on the complexity of the problem. The derived lower bounds agree with the complexity of the divide-and-conquer algorithms for these problems.

In subsequent sections, we see a few standard examples to illustrate application of the divide andconquer strategy. In each of these cases, we also evaluate the complexity of the algorithm so devised.

9.3 APPLICATIONS

Max-Min Search

Let us consider a simple problem that can be solved by the use of divide and conquer technique. Max-Min search problem aims at finding the smallest as well as the biggest element in a vector A of n elements. By the steps of Divide and Conquer, the vector can be divided into sub-problem as shown below.



The search for the maximum and minimum has now reduced to comparison of 2 numbers in the worst case. i.e., the divide and conquer strategy divides the list recursively until a stage is reached where each sub problem has utmost 2 elements which can be easily compared. The divide and conquer strategy based algorithm is given below.

Algorithm: Max-Min(p, q, max, min)

Input: p, q, the lower and upper limits of the datasetmax, min, two variables to return the maximum and minimum values in the list.

Output: the maximum and minimum values in the data set.

Method :

```
If (p = q) Then
        max = a(p)
        min = a(q)
    Else
        If ( p - q-1) Then
            If a(p) > a(q) Then
                max = a(p)
                min = a(q)
            Else
                max = a(q)
                min = a(p)
            If End
        Else
            m ← (p+q)/2
            max-min(p,m,max1,min1)
            max-min(m+1,q,max2,min2)
            max ← large(max1,max2)
            min ← small(min1,min2)
        If End
    If End
```

Algorithm Ends.

Illustration

Consider a data set with 9 elements {82, 36, 49, 91, 12, 14, 06, 76, 92}. Initially the max and minvariables have null values. In the first call, the list is broken into two equal halves. The list is again broken down into two. This process is continued till the length of the list is either two or one. Then the maximum and minimum values are chosen from the smallest list and these values are returned to the preceding step where the length of the list is slightly big. This process is continued till the entire list is searched. A good way of keeping track of recursive calls is to build

a tree by adding a node each time a new call is made. On the list of data elements considered, the tree of Fig 9.1 is produced.

Complexity

In analyzing the time complexity of this algorithm, we once again concentrate on the number of element comparisons. If T(n) represents the number of comparisons, then the recurrence relation for the above algorithm can be written as

$$T(n) = \begin{cases} T(n/2) + T(n/2) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

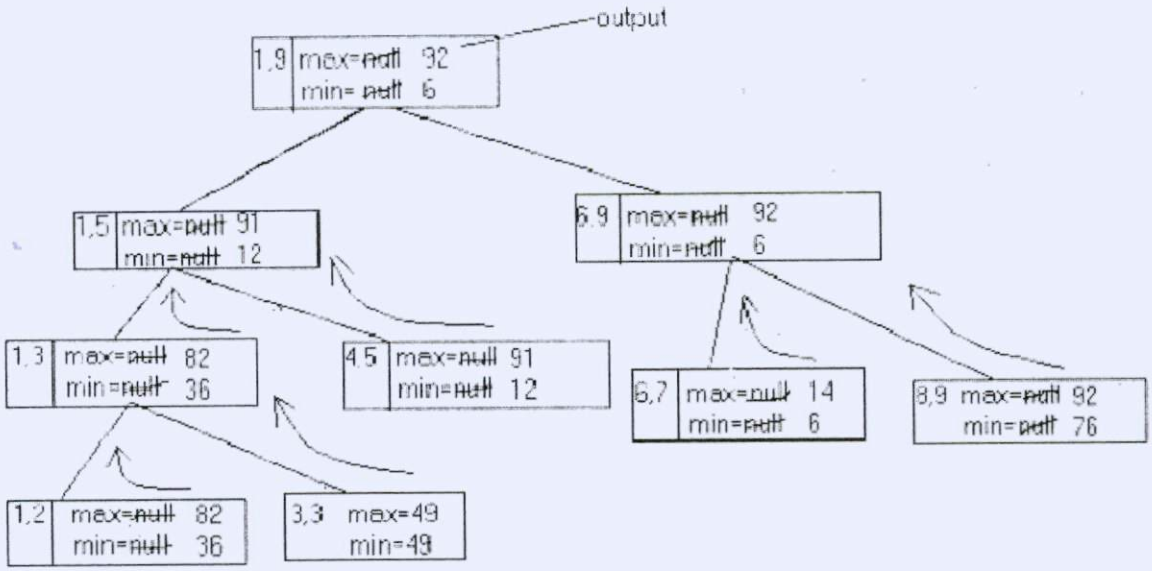


Figure 9.1

When n is a power of 2, i.e., $n=2^k$ for some positive integer k, then

$$
\begin{aligned}
T(n) &= 2T(n/2) + 2 \\
&= 2(2T(n/4) + 2) + 2 \\
&= 4T(n/4) + 4 + 2 \\
&\vdots \\
&= 2^{k-1}T(2) + \sum_{i=1}^{k-1} 2^i \\
&= 2^{k-1} - 2^k - 2 \\
&= 3n/2 - 2
\end{aligned}
$$

Note that $3n/2$ -2 is the best, average and worst case number of comparisons when n is a power of two.

Solving Recurrence Equations

Recurrences go hand in hand with the divide-and-conquer paradigm, because theygive us a natural way to characterize the running times of divide-and-conquer algorithms.A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs. For example, the worst-case running time T(n)of the MERGE-SORT procedure can be described by the recurrence as

$$
T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1, \end{cases} \tag{9.1}
$$

whose solution we claimed to be $T(n) = \Theta(n \lg n)$.

Recurrences can take many forms. For example, a recursive algorithm mightdivide subproblems into unequal sizes, such as a 2/3-to-1/3 split. If the divide andcombine steps take linear time, such an algorithm would give rise to the recurrence

$$T(n) = T(2n/3) + T(n/3) + \Theta(n).$$

Subproblems are not necessarily constrained to being a constant fraction ofthe original problem size. For example, a recursive version of linear searchwould create just one subproblem

containing only one elementless than the original problem. Each recursive call would take constanttime plus the time for the recursive calls it makes, yielding the recurrence

$$T(n) = T(n-1) + \Theta(1).$$

There are three methods for solving recurrences:

- In the ***substitution method***, we guess a bound and then use mathematical inductionto prove our guess correct.
- The ***recursion-tree method*** converts the recurrence into a tree whose nodesrepresent the costs incurred at various levels of the recursion. We use techniquesfor bounding summations to solve the recurrence.
- The ***master method*** provides bounds for recurrences of the form

$$T(n) = aT(n/b) + f(n). \tag{9.2}$$

where a ≥1, b > 1, and $f(n)$is a given function. Such recurrences arisefrequently. A recurrence of the form in equation (9.2) characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is 1/b thesize of the original problem, and in which the divide and combine steps togethertake $f(n)$time.

Technicalities in Recurrences

In practice, we neglect certain technical details when we state and solve recurrences.For example, if we call MERGE-SORT on $n$elements when $n$is odd, weend up with subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$. Neither size is actually $n/2$,because $n/2$ is not an integer when $n$is odd. Technically, the recurrence describingthe worst-case running time of MERGE-SORT is really

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n > 1. \end{cases} \tag{9.3}$$

Boundary conditions represent another class of details that we typically ignore.Since the running time of an algorithm on a constant-sized input is a constant,the recurrences that arise from the running times of algorithms generally have $T(n) = \Theta(1)$ for sufficiently small n. Consequently, for convenience, we shallgenerally omit statements of the boundary conditions of recurrences and assumethat T(n)is constant for small n. For example, we normally state recurrence (9.1) as

$$T(n) = 2T(n/2) + \Theta(n) . \tag{9.4}$$

without explicitly giving values for small n. The reason is that although changingthe value of T(1)changes the exact solution to the recurrence, the solution typicallydoesn't change by more than a constant factor, and so the order of growth isunchanged.

When we state and solve recurrences, we often omit floors, ceilings, and boundaryconditions. We forge ahead without these details and later determine whetheror not they matter. They usually do not, but you should know when they do. Experiencehelps, and so do some theorems stating that these details do not affect theasymptotic bounds of many recurrences characterizing divide-and-conquer algorithms.

The complexity of many divide-and-conquer algorithms is given by a recurrence of the form

$$t(n) = \begin{cases} t(1) & n = 1 \\ a * t(n/b) + g(n) & n > 1 \end{cases} \tag{9.5}$$

where a and b are known constants. We shall assume that $t(1)$ is known and that n is a power of b (i.e., $n = b^k$). Using the substitution method, we can show that

$$t(n) = n \log_b^a [t(1) + f(n)] \tag{9.6}$$

where $f(n) = \sum h(b^j)$ and $h(n) = g(n)/n \log_b^a$ for $j = 1$ to n.

Table 9.1 tabulates the asymptotic value of $f(n)$ for various values of $h(n)$. This table allows us to easily obtain the asymptotic value of $t(n)$ for many of the recurrences we encounter when analyzing divide-and-conquer algorithms.

Table 9.1 $f(n)$ values for various $h(n)$ values

$h(n)$	$f(n)$
$O(n^r)$, $r < 0$	$O(1)$
$\Theta((\log n)^i)$, $i \geq 0$	$\Theta((\log n)^{i+1}) / (i + 1)$, $i \geq 0$

$\Omega(n^r)$, $r > 0$	$\Theta(h(n))$

Let us consider some examples using this table. The recurrence for binary search when n is a power of 2 is

$$t(n) = \begin{cases} t(1) & n = 1 \\ t(n/2) + c & n > 1 \end{cases} \quad (9.7)$$

Comparing this recurrence with the following recurrence

$$t(n) = \begin{cases} cn \log n & n = 90 \\ t(n/9) + t(7n/8) + cn & n \geq 90 \end{cases} \quad (9.8)$$

we see that $a = 1$, $b = 2$, and $g(n) = c$. Therefore, $\log_b(a) = 0$, and $h(n) = g(n)/n^{\log_b a} = c = c(\log n)^0 = \Theta((\log n)^0)$. From Table 9.1, we obtain $f(n) = (\log n)$.

Therefore, $t(n) = n^{\log_b a}(c + \Theta(\log n) = \Theta(\log n)$.

For the merge sort recurrence, we obtain $a = 2$, $b = 2$, and $g(n) = cn$. So $\log_b a = 1$, and $h(n) = g(n)/n = c = \Theta((\log n)^0)$. Hence, $f(n) = \Theta(\log n)$ and $t(n) = n(t(1) + \Theta(\log n) = \Theta(n \log n)$.

As another example, consider the recurrence

$$t(n) = 7t(n/2) + 18n^2,\ n \geq 2 \text{ and } n \text{ a power of } 2$$

That corresponds to the recurrence for Strassen's matrix-multiplication method with $k = 1$ and $c = 18$. We obtain $a = 7$, $b = 2$ and $g(n) = 18n^2$. Therefore, $\log_b a = \log_2 7 \approx 2.81$ and $h(n) = 18n^2/n^{\log_2 7} = 18n^{2 - \log_2 7} = O(n^r)$ where $r = 2 - \log_2 7 < 0$. Therefore, $f(n) = O(1)$. The expression for $t(n)$ is $t(n) = n^{\log_2 7}(t(1) + O(1)) = c + \Theta(n^{\log_2 7})$ as $t(1)$ is assumed to be a constant.

As a final example, consider the following recurrence equation

$t(n) = 9t\ (n/3) + 4n^6$, $n \geq 3$ and n a power of 3

comparing the above recurrence equation with the equation 9.8, We obtain $a = 9$, $b = 3$ and $g(n) = 4n^2$. Therefore, $\log_b a = 2$ and $h(n) = 4n^6 / n^2 = 4n^4 = \Omega(n^4)$ and we obtain $f(n) = \Theta(h(n)) = \Theta(n^4)$.Therefore, $t(n) = n^2\ (t(1) + \Theta(n^4) = \Theta(n^6)$ as $t(1)$ may be assumed to be a constant.

9.4 SUMMARY

In this unit, we have presented Divide and Conquer strategy, a technique for designing algorithms that consistof decomposing the instance to be solved into a number of smaller sub instances of the same problem,solving successively and independently each of these sub instances, and then combining the sub solutionsthus obtained, to obtain the solution of the original instance. The Divide and Conquer strategy to solve a min-max problem has been illustrated. We have also explained the formulation of recurrence equations to a problem and to solve such equations using divide and conquer strategy. Complexity of divide and conquer strategy to solve recurrence equations has also been presented.

9.5 KEYWORDS

4) Divide-and-Conquer
5) Max-Min Search
6) Recurrence Relation
7) Complexit

9.6 QUESTIONS FOR SELF STUDY

14) Explain divide-and-conquer strategy to solve complex problems.
15) Explain min-max problem and divide-and-conquer method of solving it.
16) What is recurrence relation? Explain various methods of solving recurrence relations.
17) Discuss the technicalities of recurrence relations.
18) Discuss the complexity of recurrence relations to solve merge-sort and Strassen's matrix-multiplication methods.

9.7 REFERENCES

4) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.

5) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.

6) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

Unit – 10

Merge Sort, Quick Sort, Binary Search and their Complexities

STRUCTURE

10.0 OBJECTIVES

After studying this unit you should be able to

- Explain the divide-and-conquer methodology for sorting a list of problem.
- Formulate divide-and-conquer strategyto sort a list using merge sort, quick sort techniques and to search a list using binary search technique.
- Critically evaluate the complexity of merge sort, quick sort and binary search algorithms.

10.1MERGE SORT METHOD

Sorting is a process of arranging a set of elements in some order. We can apply the divide-and-conquer method to the sorting problem. In this problem we must sort n elements into non decreasing order. The divide-and-conquer method suggests sorting algorithms with the following general structure: If n is 1,terminate; otherwise, partition the collection of elements into two of more sub collections; sort each; combine the sorted sub collections into a single sorted collection.

The merge sort is a classic case of use of the divide and conquers methodology. The basic concept goes something like this.Consider a series of n numbers, say A(1), A(2) ……A(n/2) and A(n/2 + 1), A(n/2 + 2) ……. A(n).Suppose we individually sort the first set and also the second set. To get the final sorted list, we merge thetwo sets into one common set.We first look into the concept of arranging two individually sorted series of numbers into a commonseries using an example:

Let the first set be A = {3, 5, 8, 14, 27, 32}.

Let the second set be B = {2, 6, 9, 15, 18, 30}.

The two lists need not be equal in length. For example the first list can have 8 elements and the second5. Now we want to merge these two lists to form a common list C. Look at the elements A(1) and B(1),A(1) is 3, B(1) is 2. Since B(1) < A(1), B(1) will be the first element of C i.e., C(1)=2. Now compareA(1) =3 with B(2) =6. Since A(1) is smaller than B(2), A(1) will become the second element of C. Thus C= {2, 3}.

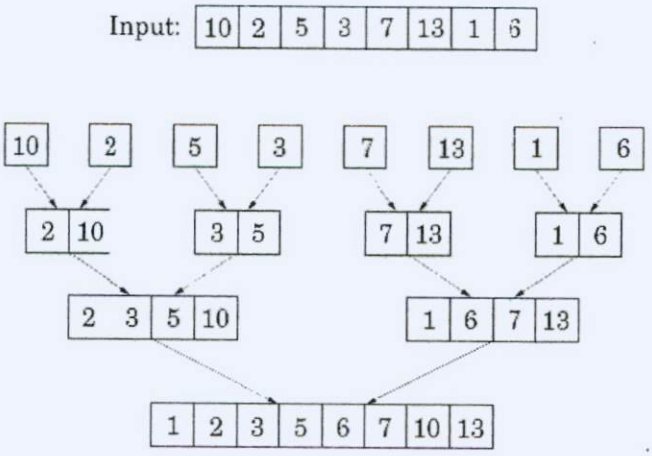
Similarly compare A(2) with B(2), since A(2) is smaller, it will be the third element of C and so on. Finally,C is built up as C= {2, 3, 5, 6, 8, 9, 14, 15, 18, 27, 30, 32}.

However the main problem remains. In the above example, we presume that both A and B are originallysorted. Then only they can be merged. But, how do we sort them in the first? To do this and show theconsequent merging process, we look at the following example.

Consider the series A= (7 5 15 6 4). Nowdivide A into 2 parts (7, 5, 15) and (6, 4). Divide (7, 5, 15) again as ((7, 5) and (15)) and (6, 4) as ((6) (4)).Again (7, 5) is divided and hence ((7, 5) and (15)) becomes (((7) and (5)) and (15)).

Now since every element has only one number, we cannot divide again. Now, we start merging them,taking two lists at a time. When we merge 7 and 5 as per the example above, we get (5, 7) merge this with15 to get (5, 7, 15). Merge this with 6 to get (5, 6, 7, 15). Merging this with 4, we finally get (4, 5, 6, 7 and15). This is the sorted list.

The above procedure can be diagrammatically shown for a different input data as follows.



Algorithm: MERGESORT

Input : low. high. the lower and upper limits of the list to be sorted

A. the list of elements

Output : A. Sorted list

Method :

If (low<high)

mid← (low − high)/2

MERGESORT(low. mid)

MERGESORT (mid. high)

MERGE(A. low. mid. high)

If end

Algorithm ends

As shown in the above algorithm, we need to design two algorithms to formulate the merge sort procedure. The main algorithm is a recursive algorithm which calls at times the other algorithm calledMERGE. The algorithm MERGE does the merging operation as discussed above.

You may recall that this algorithm runs on lines parallel to the binary search algorithm. Each time it divides the list (low, high) into two lists(low, mid) and (mid+1, high). But later, calls for merging the two lists.

Algorithm: MERGE
Input:low, mid, high, limits of two lists to be merged
A, the list of elements
Output: B, the merged and sorted list
Method:

```
h   low, i = low, j = mid + 1
While ((h ≤ mid) and (j ≤ high)) do
        If (A(h) ≤ A(j) )
                B(i) = A(h)
                h = h + 1
        else
                B(i) = A(j)
                j = j+1
        If end
        i = i+1
While end
If (h > mid)
        For k = j to high
                B(i) = A(k)
                i = i + 1
        For end
Else
        For k = h to mid
                B(i) = A(k)
                i = i+1
        For end
If end
```

Algorithm ends

The first portion of the algorithm works exactly similar to the explanation given earlier, except thatinstead of using two lists A and B to fill another array C, we use the elements of the same array A[low,mid]and A[mid+1,high] to write into another array B.

Now it is not necessary that both the lists from which we keep picking elements to write into B shouldget exhausted simultaneously. If the fist list gets exhausted earlier, then the elements of the second list aredirectly written into B, without any comparisons being needed and vice versa. This aspect will be takencare of by the second half of the algorithm.

Complexity

We can safely presume that the time required for MERGE operation is proportional to n.

Hence the time taken by mergesort is,

$$T(n) = \begin{cases} a & \text{if } n = 1 \quad a \text{ is a const.} \\ 2T(n/2) + cn & \text{if } n > 1 \quad c \text{ is a const} \end{cases}$$

That is, when n=1, then mergesort requires a constant time a. However, in a more general case, we keepdividing n into $n/2$, $n/4$, ... , etc. Assuming $n = 2^k$, where k is an integer (in a practical case n need notexactly be equal to 2^k, but k is an integer such that 2^k is the next higher integer nearer to n. For exampleif n=30, k will be 5, because$2^k = 32$ will be the next power of 2 nearest to 30).

We can write

$$T(n) = 2 (2T (n / 4) + Cn / 2) + Cn$$
$$= 4T(n /4 + 2)Cn$$
$$= 4(2T(n/8)+Cn/4) + 2Cn$$
$$= .....$$
$$= 2^k T(1) + kCn$$
$$= an + Cn \log n$$

This directly gives us the complexity of $T(n) = O(n \log n)$. There can be no distinction like best,average or worst case, since all of them need the same amount of time.

10.2QUICK SORT METHOD

This is another method of sorting that uses a different methodology to arrive at the same sorted result.It "Partitions" the list into 2 parts (similar to merge sort), but not necessarily at the centre, but at anarbitrarily "pivot" place and ensures that all elements to the left of the pivot element are lesser than theelement itself and all those to the right of it are greater than the element. Consider the following example.

75	80	85	90	95	70	65	60	55

To facilitate ordering, we add a very large element, say 1000 at the end. We keep in mind that this iswhat we have added and is not a part of the list.

75	80	85	90	95	70	65	60	55	1000
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

Quick sort again uses the technique of divide-and-conquer. We proceed as Follows:

- Pick an arbitrary element of the array (the pivot).
- Divide the array into two subarrays, those that are smaller and those that are greater (the partition phase).
- Recursively sort the subarrays.
- Put the pivot in the middle, between the two sorted subarrays to obtain the final sorted array.

In merge sort, it was easy to divide the input (we just picked the midpoint),but it was expensive to merge the results of the sorted left and right subarrays. In quick sort, dividing the problem into sub problems could be computationally expensive (as we analyze partitioning below), but putting the results back together is immediate. This kind of trade-off is frequent in algorithm design.

Now consider the first element. We want to move this element 75 to its correct position in the list. Atthe end of the operation, all elements to the left of 75 should be less than 75 and those to the right shouldbe greater than 75. This we do as follows:

Start from A(2) and keep moving forward until an element which is greater than 75 is obtained.Simultaneously start from A(10) and keep moving backward until an element smaller than 75 is obtained.

75	80	85	90	95	70	65	60	55	1000
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

Now A(2) is larger than A(1) and A(9) is less than A(1). So interchange them and continue theprocess.

75	55	85	90	95	70	65	60	80	1000
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

Again A(3) is larger than A(1) and A(8) is less than A(1), so interchange them.

75	55	60	90	95	70	65	85	80	1000
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

Similarly A(4) is larger than A(1) and A(7) is less than A(1), interchange them

75	55	60	65	95	70	90	85	80	1000
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

In the next stage A(5) is larger than A(1) and A(6) is lesser than A(1), after interchanging we have

75	55	60	65	70	95	90	85	80	1000
A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

In the next stage A(6) is larger than A(1) and A(5) is lesser than A(1), we can see that the pointershave crossed each other, hence Interchange A(1) and A(5).

70	55	60	65	75	95	90	85	80	1000

A(1)	A(2)	A(3)	A(4)	A(5)	A(6)	A(7)	A(8)	A(9)	A(10)

We have completed one series of operations. Note that 75 is at its proper place. All elements to its leftare lesser and to its right are greater.

Next we repeat the same sequence of operations from A(1) to A(4) and also between A(6) to A(10).This we keep repeating till single element lists are arrived at.

Now we suggest a detailed algorithm to do the same. As before, two algorithms are written. Themain algorithm, called QUICKSORT repeatedly calls itself with lesser and lesser number of elements. However,the sequence of operations explained above is done by another algorithm called PARTITION.

Algorithm: QUICKSORT

Input: p, q, the lower and upper limits of the list of elements A to be sorted

Output: A, the sorted list

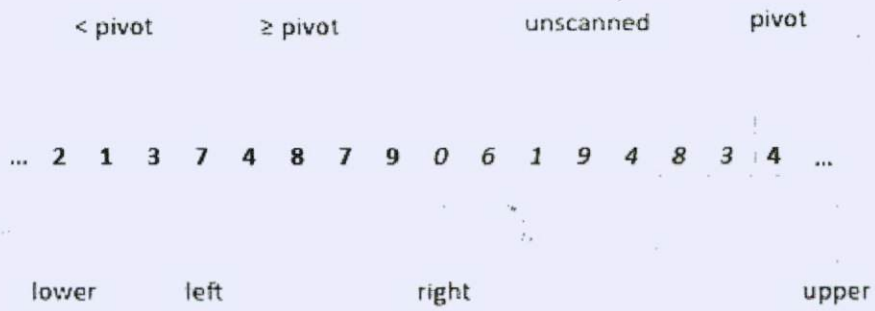
Method:

```
If (p < q)
        j = q+1;
        PARTITION (p, j)
        QUICKSORT(P, j-1)
        QUICKSORT(j+1, q)
If end
```

Algorithm ends

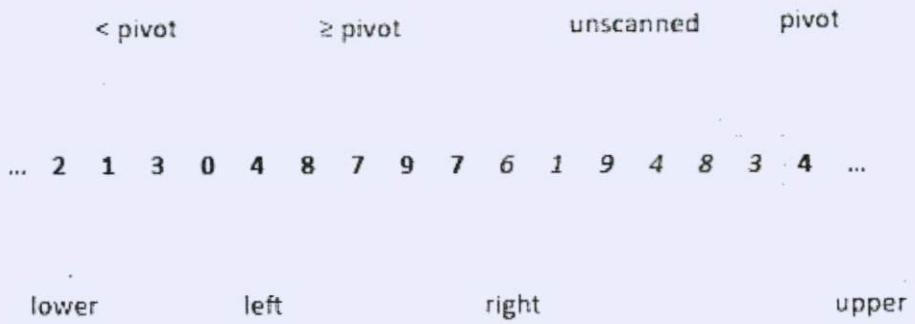
Partitioning

The trickiest aspect of quick sort is the partitioning step, in particular since we want to perform this operation in place. Once we have determined the pivot element, we want to divide the array segment into four different sub segments as illustrated in this diagram.

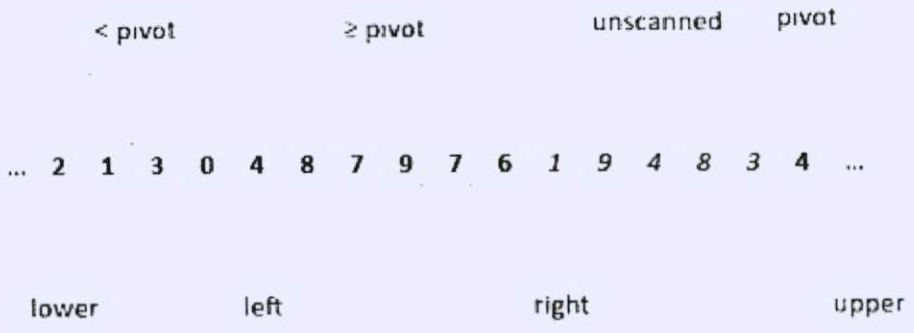


We fix lower and upper as they are when partition is called. The segment A [lower..left) contains elements known to be less than the pivot, the segment A[left..right) contains elements greater or equal to the pivot, and the element at A[upper−1] is the pivot itself. The segment from A[right..upper−1] has not yet been scanned, so we don't know yet how these elements compare to the pivot.

We proceed by comparing A[right] with the pivot. In this particular example, we see that A[right] < pivot. In this case we swap the element with the element at A [left] and advance both left and right, resulting in the following situation.



The other possibility is that A[right] ≥ pivot. In that case we can just advance the right index by one and maintain the invariants without swapping any elements. The resulting situation would be the following.

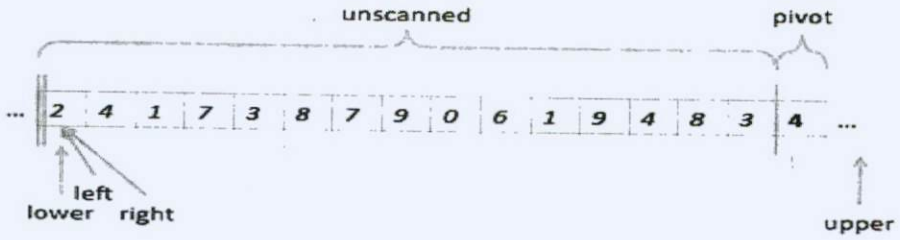


When right reaches upper − 1, the situation will look as follows:

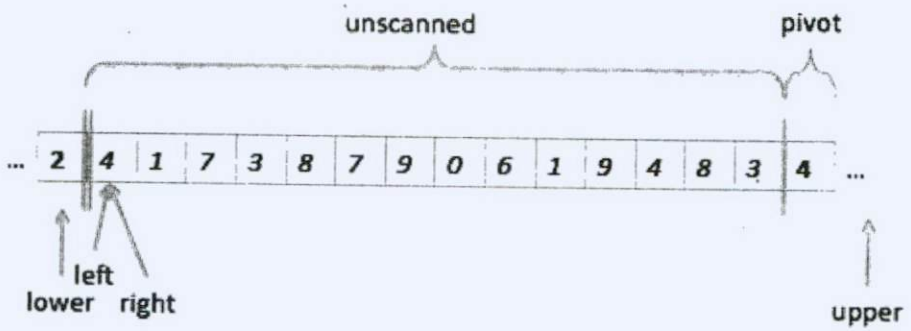
We can now just swap the pivot with A[left], which is known to be greater or equal to the pivot.The resulting array segment has been partitioned, and we return left as the index of the pivot element.

Throughout this process, we have only ever swapped two elements of the array. This guarantees that the array segment after partitioning is a permutation of the segment before. However, we did not consider how to start this algorithm. We begin by picking a random

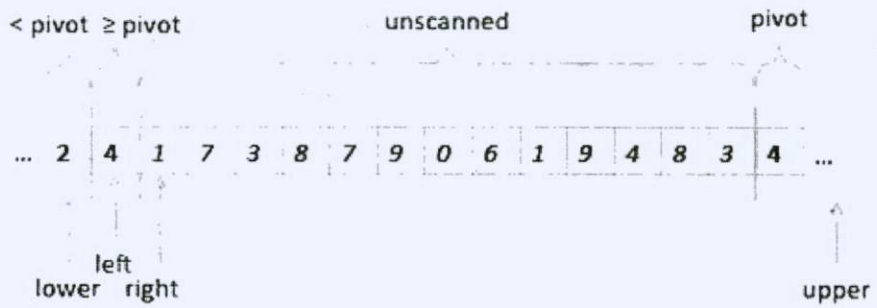
element as the pivot and then swapping it with the last element in the segment. We then initialize left and right to lower. We then have the situationwhere the two segments with smaller and greater elements than the pivot are still empty.



In this case (where left = right), if A[right] ≥ pivot then we can increment right as before, preserving the invariants for the segments. However, if A[left] < pivot, swapping A[left] with A[right] has no effect. Fortunately, incrementing both left and right preserves the invariant since the element we just checked is indeed less than the pivot.



If left and right ever separate, we are back to the generic situation we discussed at the beginning. In this example, this happens in the next step.



If left and right always stay the same, all elements in the array segment are strictly less than the pivot, excepting only the pivot itself. In that case, too, swapping A[left] and A[right] has no effect and we return left = upper – 1 as the correct index for the pivot after partitioning.

Algorithm: PARTITION

Input: m, the position of the element whose actual position in the sorted list has to be found

p, the upper limit of the list

Output: the position of m^{th} element

Method:

```
v = A(m);
i = m;
Repeat
        Repeat
                i = i+1
        Until (A(i) ≥ v);
        Repeat
                p = p - 1
        Until (A(p) ≤ v);
        If (i < p)
                INTERCHANGE(A(i), A(p))
        Ifend
Until (i ≥ p)
A(m) = A(p) ;
A(p) = v;
```

Algorithm ends

We see that in the worst case there are $n - 1$ significant recursive calls on an array of size n. The k^{th} recursive call has to sort a subarray of size k, which proceeds by partitioning, requiring $O(k)$ comparisons. This means that, overall, for some constant c we have

$$c\sum_{j=0}^{n-1} j = c\frac{n(n-1)}{2} = O(n^2)$$

comparisons. Here we used the fact that $O(p(n))$ for a polynomial $p(n)$ is always equal to the $O(n^k)$ where k is the leading exponent of the polynomial. This is because the largest exponent of a polynomial will eventually dominate the function, and big-O notation ignores constant coefficients.

So in the worst case, quick sort has quadratic complexity. How can we mitigate this? If we always picked the median among the elements in the subarray we are trying to sort, then half the elements would be less and half the elements would be greater. So in this case there would be only $\log(n)$ recursive calls, where at each layer we have to do a total amount of n comparisons, yielding an asymptotic complexity of $O(n \log n)$.

Unfortunately, it is not so easy to compute the median to obtain the optimal partitioning. It turns out that if we pick a random element, it will be on average close enough to the median that the expected running time of algorithm is still $O(n \log n)$.

We should really make this selection randomly. With a fixed-pick strategy, there may be simple inputs on which the algorithm takes $O(n^2)$ steps. For example, if we always pick the first element, then if we supply an array that is already sorted, quick sort will take $O(n^2)$ steps (and similarly if it is "almost" sorted with a few exceptions)! If we pick the pivot randomly each time, the kind of array we get does not matter, the expected running time is always the same, namely $O(n \log n)$.This is an important use of randomness to obtain a reliable average case behavior.

10.3 BINARY SEARCH

Binary search is an efficient technique for searching in a sorted array. It works by comparing a searching element 'e' with the array's middle element A(mid). If they match, the

algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array **if e < A(mid)** and for the second half **if e > A(mid).**

Consider the same example of sequential search and let us apply Binary search technique. Let A be the list and 'e'=11 be the searching element. As binary search technique assumes that the list will be in sorted order, sorting of elements in the list will be the first step.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A=	10	15	6	23	11	96	55	44	66	8	2	30	69	96

Let Low = 1; High = 14 be the initial values, where Low is the starting position and High is the last position of the list.

Sort the given list using any of the sorting techniques. In this illustrative we assume that we have sorted list of A.

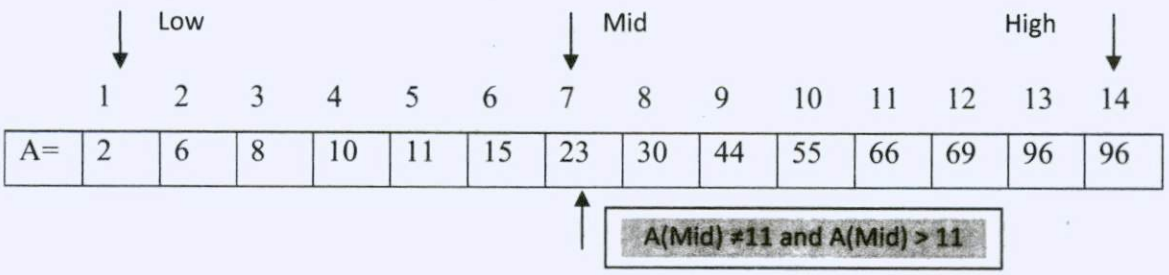
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A=	2	6	8	10	11	15	23	30	44	55	66	69	96	96

Compute the mid of list; Mid = (Low + High)/2;

Mid = (1+14)/2 = 15/2 = 7.5 ≈ 7.

Check whether A(Mid) = = Search element.

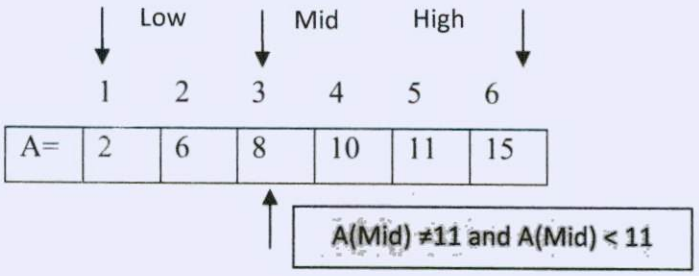
A(7) = 23 and it is not equal to search element 11.



Since the search element is not equal to A(mid) and A(mid) is greater than the search element, the search element should be present in the first half of the list. Consider first half as the new list

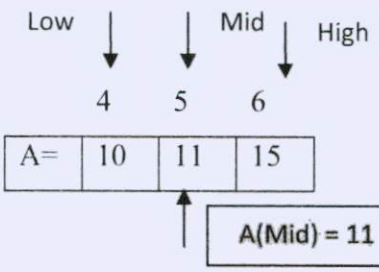
and continue the same procedure neglecting the second half of the list. For further processing initialization are as follows

High = Mid – 1; i.e., High =7 – 1 = 6; Mid = (Low+High)/2; i.e., Mid = (1+6)/2 = 7/2 ≈3. The value of the Low remains unchanged ie., Low = 1.



Since the search element is not equal to A(mid) and A(mid) is less than the search element, the search element should be present in the second half of the list. Consider second half as the new list and continue the same procedure neglecting the first half of the list. For further processing initialization are as follows

Low = Mid+1, i.e., Mid =3+1 = 4; Mid = (Low+High)/2; i.e., Mid = (4+6)/2 = 10/2 = 5. In this case the value of High remains same i.e., High = 6.



As A(Mid) is equal to search element, it can be announced that Search element is found in position 5. The recursive algorithm for binary search is as follows.

Algorithm: Binary Search

Input: A – List of elements
'e' element to be searched
'n' size of the list
Low = 1, High = n;

Output: Success or Unsuccess.

Method:

```
While (Low <= High)
        Mid = (Low+High)/2;
        If(A(Mid) = = e)
                Display 'Element found in Mid Position'
        Else
                If(A(Mid) > E)
                        High = Mid – 1
                        Binary Search(A(Low : High),e)
                Else
                        Low = Mid+1
                        Binary Search(A(Low : High),e)
                If end
        If end
        While end
```

Algorithm ends

Time complexity

To analyze the complexity of the above designed algorithm, we concentrate on the number of element comparisons.

1. The best case is when we find the element in the first iteration itself i.e., the complexity is O(1)
2. The average case is when we find the element after some iterations, since each iteration reduces the number of elements to half, given n elements, we can divide it by log n times i.e., limits of the search, i.e., O(log n) is the complexity.
3. Even in the worst case, we would be able to get to the element of log n operations. So the worst case complexity is also O(log n). If we are not able to find the element, that also becomes known after log n operations. So the unsuccessful search also has a complexity of O(log n).

10.4 SUMMARY

In this unit, we have presented the two sorting techniques and one searching technique, which use the Divide and Conquer strategy.The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. It breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list. The merge sort is a comparison sort and has an algorithmic complexity of $O(n \log n)$.

The Quick Sort selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sub lists are recursively sorted until the list gets completely sorted. The time complexity of this algorithm is $O(n \log n)$.

Binary search is an efficient searching technique, which can be applied to a sorted list of elements. The essence of the binary search technique is that we look for the middle item in the list and compare it with the search element. If it matches, we stop the search process; otherwise, we check whether the search element lies in the first half of the list or in the second half. Then we recursively apply the searching process till the element is found or the list exhausts. Binary search is one of the $O(\log n)$ searching algorithms, which makes it quite efficient for searching large data volumes.

10.5 KEYWORDS

1) Divide-and-Conquer
2) Merge Sort
3) Quick Sort
4) Binary Search
5) Complexity

10.6 QUESTIONS FOR SELF STUDY

1) Explain merge sort technique with an example.
2) Explain the procedure followed in quick sort technique with an example.

3) Explain binary search technique with an example.

4) Obtain the time complexity for merge sort, quick sort and binary search.

10.7 REFERENCES

1) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.

2) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.

3) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

Unit – 11

Concept of Greedy Method and Optimization Problems

STRUCTURE

11.0 OBJECTIVES

After studying this unit you should be able to

- Analyze the optimal solution to a problem.

- Explain the greedy method to construct an optimal solution to a problem.
- Design greedy algorithm for a problem.
- Analyze the complexity of an algorithm.

11.0 INTRODUCTION

Greedy method is a method of choosing a subset of a dataset as the solution set that result in some profit. Consider a problem having n inputs. We are required to obtain a solution which is a series of subsets that satisfy some constraints or conditions. Any subset, which satisfies these constraints, is called a feasible solution. It is required to obtain a feasible solution that maximizes or minimizes an objective function. This feasible solution finally obtained is called optimal solution. The concept is called Greedy because at each stage we choose the "best" available solution i.e., we are "greedy" about the output.

In greedy strategy, one can devise an algorithm that works in stages, considering one input at a time and at each stage, a decision is taken on whether the data chosen results with an optimal solution or not. If the inclusion of a particular data, results with an optimal solution, then the data is added into the partial solution set. On the other hand, if the inclusion of that data results with infeasible solution then the data is eliminated from the solution set.

Stated in simple terms, the greedy algorithm suggests that we should be "greedy" about the intermediate solution i.e., if at any intermediate stage k different options are available to us, choose an option which "maximizes" the output.

The general algorithm for the greedy method is

- Choose an element e belonging to the input dataset D.
- Check whether e can be included into the solution set S, If yes solution set is Union(S, e)
- Continue until s is filled up or D is exhausted whichever is earlier.

Sometimes the problem under greedy strategy could be to select a subset out of given n inputs, and sometimes it could be to reorder the n data in some optimal sequence.

The control abstraction for the subset paradigm is as follows:

Algorithm: GREEDY

Input: A, a set containing *n* inputs

Output: S, the solution subset

Method: S = { }

```
For I = 1 to n do
    x = SELECT (A(i))
    If (FEASIBLE (S, x))
        S = UNION(S, x)
    If end
For end
return(S)
```

Algorithm ends

SELECT selects the best possible solution (or input) from the available inputs and includes it in the solution. If it is feasible (in some cases, the constraints may not allow us to include it in the solution, even if it produces the best results), then it is appended to the partially built solution. The whole process is repeated till all the options are exhausted.

In the next few sections, we look into few optimization problems of the Greedy method.

11.2 THE PROBLEM OF OPTIMAL STORAGE ON TAPES

We know, when large quantities of data or program need to be backed up, the best way is to store them on tapes. For our discussion, let us presume that they are programs. However, they can be any other data also. (Though with the advent of high capacity CDs, the importance of tapes as storage media can be said to have reduced, but they have not vanished altogether).

Consider *n* programs that are to be stored on a tape of length *L*. Each program P_i is of length l_i where *i* lies between 1 and *n*. All programs can be stored on the tape iff the sum of the lengths of the programs is at most *L*. It is assumed that, whenever a program is to be retrieved the tape is initially positioned at the start end. That is, since the tape is a sequential device, to access the i^{th} program, we will have to go through all the previous (*i*-1) programs before we get to *i*, i.e.,

to go to a useful program (of primary interest), we have to go through a number of useless (for the time being) programs. Since we do not know in what order the programs are going to be retrieved, after accessing each program, we come back to the initial position.

Let t_i be the time required for retrieving program ii where programs are stored in the order $\{P_1, P_2, P_3, \ldots, P_n\}$. Thus to get to the i^{th} program, the required time is proportional to

$$t_i = \sum_{k=1}^{i} l_k$$

If all programs are accessed with equal probability over a period of time, then the average retrieval time (mean retrieval time)

$$\text{MRT} = \frac{1}{n} \sum_{k=1}^{n} t_k$$

To minimize this average retrieval time, we have to store the programs on the tape in some optimal sequence. Before trying to devise the algorithm, let us look at the commonsense part of it. The first program (program no. 1) will have to be passed through, the maximum number of times, as irrespective of the program that we are interested in, we have to pass this. Similarly, the second program has to be passed through in all cases, except when we are interested in the first program. Likewise it can be concluded that the first, second, ... , n^{th} programs are scanned/passed through with decreasing frequency. Hence, it is better to store the shortest program at the beginning. Due to the fact that the first program is passed/scanned, maximum number of times, if we store the smallest one, it adds the smallest possible amount to the time factor.

Similarly the second program will have to be the second smallest and so on. Precisely, the programs to be stored on the tape are to be arranged according to the increasing lengths. The smallest program which is passed/scanned through the maximum number of times, adds less over head. The largest of programs is at the end of the tape, but will hardly be passed through, except when we are actually interested in the same.

From the above discussion one can observe that the MRT can be minimized if the programs are stored in an increasing order i.e., $l_1 \leq l_2 \leq l_3 \ldots \leq l_n$. Hence the ordering defined

minimizes the retrieval time. Here, the solution set obtained is not a subset of data but is the data set itself in a different sequence.Now we take a numerical example to see whether this works.

Illustration

Assume that 3 files are given. Let the length of files A, B and C be 7, 3 and 5 units respectively. All these three files are to be stored on to a tape S in some sequence that reduces the average retrieval time.

The table shows the retrieval time for all possible orders.

Order of recording	Retrieval time	MRT
ABC	7+(7+3)+(7+3+5)=32	$\frac{32}{3}=10.667$
ACB	7+(7+5)+(7+5+3)=34	$\frac{34}{3}=11.333$
BAC	3+(3+7)+(3+7+5)=28	$\frac{28}{3}=9.333$
BCA	3+(3+5)+(3+5+7)=26	$\frac{26}{3}=8.333$
CAB	5+(5+7)+(5+7+3)=32	$\frac{32}{3}=10.667$
CBA	5+(5+3)+(5+3+7)=28	$\frac{28}{3}=9.333$

Though this discussion looks fairly straight forward enough not to need an algorithm, it provides us certain insights. In this case, we are trying to optimize on the time of accessing, which is nothing but the lengths of programs to be traversed. Hence, we try to optimize on the length of the part of the scan we scan at each stage - i.e. we are "greedy" about the time spent in retrieving a file of our interest. We actually build the list in stages. At each stage of storing the programs choose the least costly of available options - i.e. the program of shortest length. Though the program is simple enough, we write a small algorithm to implement the same.The procedure can also be used to append programs on more than one tape. The first tape gets the smallest set of the programs (in terms of their lengths) and so on.

Complexity

The greedy method simply requires us to store the programs in nondecreasing order of their lengths. This ordering can be done in O(n log n) time using efficient sorting algorithm like heap sort.

Algorithm: TAPE STORAGE

Input : n. the number of programs

$l_1, l_2, l_3, \ldots, l_n$. the lengths of the programs to be stored

Output: minimum mean retrieval time yielding sequence

Method :

Sort the list in the non decreasing order of lengths

For i = 1 to n do

Choose the i^{th} program from the sorted list and store it on the tape

For end

Algorithm ends

11.3 A THIRSTY BABY

Assume there is a thirsty, but smart, baby, who has access to a glass of water, a carton of milk, etc., a total of n different kinds of liquids. Let a_i be the amount of ounces in which the i^{th} liquid is available. Based on her experience, she also assigns certain satisfying factor, s_i, to the i^{th} liquid. If the baby needs to drink t ounces of liquid, how much of each liquid should she drink?

Let x_i, $1 \le i \le n$, be the amount of the i^{th} liquid the baby will drink. The solution for this thirsty baby problem is obtained by finding real numbers x_i, $1 \le i \le n$, that maximize $\sum_{i=1}^{n} s_i x_i$ subject to the constraints that $\sum_{i=1}^{n} x_i = t$ and for all $1 \le i \le n$, $0 \le x_i \le a_i$. We notice that if $\sum_{i=1}^{n} a_i < t$ then this instance will not be solvable.

A specification

Input: n, t, s_i, a_i, $1 \le i \le n$. n is an integer, and the rest are positive reals.

Output: If $\sum_{i=1}^{n} a_i \ge t$, output is a set of real numbers x_i, $1 \le i \le n$, such that $\sum_{i=1}^{n} s_i x_i$ is maximum, $\sum_{i=1}^{n} x_i = t$, and for all $1 \le i \le n$, $0 \le x_i \le a_i$.

In this case, the constraints are $\sum_{i=1}^{n} x_i = t$, and for all $1 \le i \le n$, $0 \le x_i \le a_i$, and the optimization function is $\sum_{i=1}^{n} s_i x_i$.

Every set of xi that satisfies the constraints is a feasible solution, and if it further maximizes $\sum_{i=1}^{n} s_i x_i$ is an optimal solution.

11.4 LOADING PROBLEM

A large ship is to be loaded with containers of cargos. Different containers, although of equal size, will have different weights. Let w_i be the weight of the i^{th} container, $1 \le i \le n$, and the capacity of the ship is c, we want to find out how could we load the ship with the maximum number of containers.

Let $x_i \in \{0, 1\}$. If $x_i = 1$, we will load the i^{th} container, otherwise, we will not load it. We wish to assign values to x_i's is such that $\sum_{i=1}^{n} x_i \le C$, and $x_i \in \{0, 1\}$. The optimization function is $\sum_{i=1}^{n} x_i$.

11.5 CHANGE MAKING

A child buys a candy bar at less than one buck and gives a $1 bill to the cashier, who wants to make a change using the fewest number of coins. The cashier constructs the change in stages, in each of which a coin is added to the change.

The greedy criterion is as follows: At each stage, increase the total amount as much as possible. To ensure the feasibility, such amount in no stage should exceed the desired change. For example, if the desired change is 67 cents. The first two stages will add in two quarters. The

next one adds a dime, and following one will add a nickel, and the last two will finish off with two pennies.

11.6 MACHINE SCHEDULING

We are given an infinite supply of machines, and n tasks to be performed in those machines. Each task has a start time, s_i, and finish time, t_i. The period $[s_i, t_i]$ is called the processing interval of task i. Two tasks i and j might overlap, e.g., [1, 4] overlaps with [2, 4], but not with [4, 7].

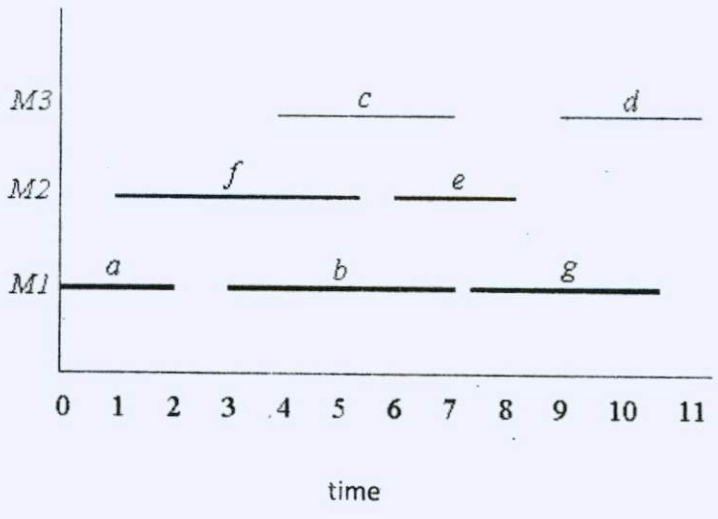
A feasible assignment is an assignment in which no machine is given two overlapped tasks. An optimal assignment is a feasible one that uses fewest numbers of machines.

We line up tasks in nondecreasing order of s_i's, and call a machine *old*, if it has been assigned a task. otherwise, call it *new*. A greedy strategy could be the following: At each stage, if an old machine becomes available by the start time of a task, assign the task to this machine; otherwise, assign it to a new one.

Example: Given seven tasks, their start time, as well as their finish time as follow:

task	*a*	*b*	*c*	*d*	*e*	*f*	*g*
start	0	3	4	9	7	1	6
finish	2	7	7	11	4	5	8

Then, by the aforementioned strategy, we can assign the tasks to machines in the following way:



It may be implemented in $\theta(n \log n)$, by using a minHeap of availability times for the old machines.

11.7 SUMMARY

In this unit, we have introduced the optimization problems. In an optimization problem, we are given a set of constraints and an optimization function. Solutions that satisfy the constraints are called feasible solutions. A feasible solution for which the optimization function has the best possible value is called an optimal solution. We have considered some optimization problems and understand how to provide optimal solution to these problems.

11.8 KEYWORDS

1) Optimization problem
2) Feasible solution
3) Optimal solution
4) Greedy Algorithm
5) Complexity

11.9 QUESTIONS FOR SELF STUDY

1) What is an optimization problem? Explain.
2) What is a greedy method? Explain the general algorithm for the greedy method.
3) Explain the problem of optimal storage on tapes with an example.
4) Define a thirsty baby problem and obtain an optimal solution for this problem.
5) Formulate the loading problem as an optimization problem.
6) What is machine scheduling problem? How do you obtain optimal solution for this problem? Explain.

11.10 REFERENCES

1) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.
2) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.
3) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

Unit – 12

Applications of Greedy Method

STRUCTURE

12.0 OBJECTIVES

After studying this unit you should be able to

- Analyze the container loading problem.
- Design the greedy algorithm to the container loading problem.
- Analyze the Knapsack problem.
- Design the greedy algorithm to the Knapsack problem.
- Analyze the Minimum Cost Spanning Tree problem.
- Design the greedy algorithm to the Minimum Cost Spanning Tree problem.
- Analyze the complexity of these algorithms.

12.1 CONTANER LOADING

A large ship is to be loaded with containers of cargos. Different containers, although of equal size, will have different weights. Let w_i be the weight of the i^{th} container, $1 \leq i \leq n$, and the capacity of the ship is c, we want to find out how could we load the ship with the maximum number of containers.

The ship may be loaded in stages; one container per stage. At each stage we need to decide which container to load. For this decision we may use the greedy criterion: *From the remaining containers, select the one with least weight.* This order of selection will keep the total weight of the selected containers minimum and hence leave maximum capacity for loading more containers. Using the greedy algorithm just outlined, we first select the container that has least weight, then the one with the next smallest weight, and so on until either all containers have been loaded or there is not enough capacity for the next one.

Example 6.1: Suppose that $n = 8$, $[w_1, \ldots, w_8] = [100, 200, 50, 90, 150, 50, 20, 80]$, and $c = 400$. When the greedy algorithm is used, the containers are considered for loading in the order 7, 3, 6, 8, 4, 1, 5, 2. Containers 7, 3, 6, 8, 4, and 1 together weigh 390 units and are loaded. The available capacity is now 10 units, which is inadequate for any of the remaining containers.

In the greedy solution we have $[x_1, , , , , x_8] = [1, 0, 1, 1, 0, 1, 1, 1]$ and $\Sigma x_i = 6$.

Theorem 6.1: The greedy algorithm generates optimal loadings.

Proof: We can establish the optimality of the greedy solution in the following way:

Let x = $[x_1, x_2, x_3, \ldots, x_n]$ be the solution produced by the greedy algorithm.

Let y = $[y_1, y_2, y_3, \ldots, y_n]$ be any feasible solution.

We shall show that $\Sigma x_i \geq \Sigma y_i$ for $i = 1$ to n

Without loss of generality, we may assume that the containers have been ordered so that $w_i \leq w_{i+1}$, $1 \leq i \leq n$. We shall transform y in several steps into x. Each step of transformation will produce a new y that is feasible and for

which Σy_i for $i = 1$ to n is no smaller than before the transformation. As a result, $\Sigma x_i \geq \Sigma y_i$ for $i = 1$ to n, initially.

From the way greedy algorithm works, we know that there is a k in the range $[0, n]$ such that $x_i = 1$, $i \leq k$, and $x_i = 0$, $i > k$.

Find the least integer j in the range $[1, n]$ such that $x_i \neq y_j$. If no such j exists, then $\Sigma x_i = \Sigma y_i$ for $i = 1$ to n. If such a j exists, then $j \leq k$, as otherwise y is not a feasible solution. Since $x_i \neq y_j$ and $x_i = 1$, $y_j = 0$. Set y_j to 1. If the resulting y denotes as infeasible solution, there must be an l in the range $[j+1, n]$ for which $y_l = 1$. Set y_l to 0. As $w_j \leq w_l$, the resulting y is feasible. Also, the new y has at least as many ones as the old y.

By using this transformation several times, we can transform y into x. As each transformation produces a new y that has at least as many ones as the previous y, x has at least as many ones as the original y had.

12.2 THE KNAPSACK PROBLEM

Greedy method is best suited to solve more complex problems such as a knapsack problem. In a knapsack problem there is a knapsack or a container of capacity M, n items where each item i is of weight w_i and is associated with a profit P_i. The problem of knapsack is to fill the available items into the knapsack so that the knapsack gets filled up and yields a maximum profit. If a fraction x_i of object i is placed into the knapsack, then a profit $P_i x_i$ is earned. The constraint is that all chosen objects should sum up to M. Obviously, the bag cannot carry all the objects if $M < \sum_{i=1}^{n} w_i$ Otherwise the problem becomes trivial and one can put all the objects into the bag and obtain the maximum profit which is equal to $\sum_{i=1}^{n} P_i$ We can formulate the knapsack problem as Maximize, $\sum_{i=1}^{n} p_i x_i$ Subject to $\sum_{i=1}^{n} w_i x_i \leq M$ where $(0 \leq x_i \leq 1)$ $(1 \leq i \leq n)$.

Though, we talk of storing the items whose total weight is less than M, we are actually interested in storing items such that the weights become equal to M (because total weight less than M means, we can add some more and get more profit. The only constraint is that we cannot exceed M, the capacity of the bag). We presume that we keep filling the bag with complete objects, until the last object which will be chosen to see that the remaining space gets filled up.

What is that we will be "greedy" about? In the previous case of tape storage, we were greedy about the time spent for accessing a program of our interest. We chose, at each stage, the program which occupied the smallest possible space. In this case, however, we see several options.

We can choose the object that takes up the least capacity of the bag, just as in the case of tapes, we mean, we can choose at each stage the object with the least weight that is we intend to fill the bag as slowly as possible. We can also think of choosing the objects that is most profitable that is we can choose the most profitable of the remaining object to be put into the knapsack first.

Common sense, again suggests that each of these is not a very good option. The object that has the least weight may also have very low profit. In fact we may save something on sack capacity, but may end up loosing the maximum profits. Conversely, the object with the highest profit may also have high weight and hence fill the knapsack all alone.

Thus, we should choose a mechanism that is a mean of the two, profit and weight. We should fill the knapsack as slowly as possible, still increasing the profits as fast as possible. That is, we should not simply choose the lighter objects or more profitable objects, but choose those objects that give the best profit/weight ratio. Specifically, we have to choose those objects which have the profit/weight ratio to be high. Thus, in order to get a maximum profit, the knapsack has top be filled with the objects by ordering them in the decreasing order of their profit/weight ratio.

Illustration

Consider a knapsack problem of finding the optimal solution where, M= 15, the profits associated (P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3) and (W_1, W_2, W_3, W_4, W_5, W_6, W_7) = (2, 3, 5, 7, 1, 4, 1).

As explained earlier, in order to find the feasible solution, one can follow three different strategies.

Strategy 1: Non-increasing profit values

Let (X_1, X_2, X_3, X_4, X_5, X_6, X_7) represent the items with profit (10, 5, 15, 7, 6, 18, 3) then the sequence of objects with non-increasing profit is (X_5, X_3, X_1, X_4, X_5, X_2, X_7).

Item chosen for inclusion	Quantity of item included	Remaining space in M	P_iX_i
X_6	1 full unit	15-4=11	18*1=18
X_3	1 full unit	11-5=6	15*1=15
X_1	1 full unit	6-2=4	10*1=10
X_4	4/7 unit	4-4=0	4/7*7=04

Profit = 47 units

The solution set is (1, 0, 1, 4/7, 0, 1, 0).

Strategy 2: Non-decreasing weights

The sequence of objects with non-decreasing weights is (X_5, X_7, X_1, X_2, X_6, X_3, X_4).

Item chosen for inclusion	Quantity of item included	Remaining space in M	P_iX_i
X_5	1 full unit	15-1=14	6*1=6
X_7	1 full unit	14-1=13	3*1=3
X_1	1 full unit	13-2=11	10*1=10
X_2	1 full unit	11-3=8	5*1=05
X_6	1 full unit	8-4=4	18*1=18
X_3	4/5 unit	4-4=0	4/5*15=12

Profit = 54 units

The solution set is (1, 1, 4/5, 0, 1, 1, 1).

Strategy 3: Maximum profit per unit of capacity used

(This means that the objects are considered in decreasing order of the ratio Pi/Wi)

X_1: $P_1/W_1 = 10/2 = 5$

X_2: $P_2/W_2 = 5/3 = 1.66$

X_3: $P_3/W_3 = 15/5 = 3$

X_4: $P_4/W_4 = 7/7 = 1$

X_5: $P_5/W_5 = 6/1 = 6$

X_6: $P_6/W_6 = 18/4 = 4.5$

X_7: $P_7/W_7 = 3/1 = 3$

Hence, the sequence is (X_5, X_1, X_6, X_3, X_7, X_2, X_4)

Item chosen for inclusion	Quantity of item included	Remaining space in M	P_iX_i
X_5	1 full unit	15-4=11	15*1=18
X_1	1 full unit	15-1=14	6*1=6
X_6	1 full unit	14-2=12	10*1=10
X_3	1 full unit	12-4=8	18*1=18
X_7	1 full unit	8-5=3	15*1=15
X_2	1 full unit	3-1=2	3*1=3
X_4	2/3 unit	2-2=0	2/3*5=3.33

Profit = 55.33 units

The solution set is (1, 2/3, 1, 0, 1, 1, 1).

In the above problem it can be observed that, if the sum of all the weights is $\leq M$ then all $X_i = 1$, is an optimal solution. If we assume that the sum of all weights exceeds M, all X_i's cannot be one Sometimes it becomes necessary to take a fraction of some items to completely fill the knapsack. This type of knapsack problems is a general knapsack problem. The algorithm designed to accomplish the task of knapsack is as follows.

Algorithm: Greedy Knap

Input : $(X_1, X_2, X_3, \ldots, X_n)$ the n items to be filled into the knapsack

$(P_1, P_2, P_3, \ldots, P_n)$, the profits associated with the n items

(W_1, W_2, W_3, W_n), the weights associated with n items

M, capacity of the knapsack

Output: S, sequence of items yielding maximum profit

Method :

For i = 1 to n do

Compute PW[i] = P_i / W_i

For end

Sort the objects X_i's such that

$P_i/W_i \geq P_{i+1} / W_{i+1}$

S[ ] = 0;

Cr = M // Cr indicates the remaining sack capacity and is

For i = 1 to n do

If ($Cr - W_i \geq C_r$)

S[i] = 1;

$Cr = Cr - W_i$

Else

S[i] = Cr / W_i

$Cr = Cr - Cr / W_i$

If end

For end

Return(S)

Algorithm ends

Complexity

Disregarding the time taken to initially sort the items based on their increasing weights or decreasing profits or the profit/weight ratios the complexities of all three strategies is O(n) time.

12.3MINIMUM COST SPANNING TREE

Let $G = (N, A)$be a connected, undirected graph where N is the set of nodes and A is the set of edges. Each edge has a given nonnegative *length*. The problem is to find a subset T of the edges of G such that all the nodes remain connected when only the edges in T are used, and the sum of the lengths of the edges in T is as small as possible. Since G is connected, at least one solution must exist. If G has edges of length 0, then there may exist several solutions whose total length is the same but that involve different numbers of edges. In this case, given two solutions with equal total length, we prefer the one with least edges. Even with this proviso, the problem may have several different solutions of equal value. Instead of talking about length, we can associate a *cost* to each edge. The problem is then to find a subset T of the edges whose total cost is as small as possible. Obviously this change of terminology does not affect the way we solve the problem.

Let $G' = (N, T)$be the partial graph formed by the nodes of G and the edges in T,and suppose there are n nodes in N.A connected graph with n nodes must have at least $n - 1$ edges, so this is the minimum number of edges there can be in T. On the other hand, a graph with n nodes and more than $n - 1$ edges contains at least one cycle. Hence if G' is connected and T has more than $n - 1$edges, we can remove at least one of these without disconnecting G', provided we choose an edge that is part of a cycle. This will either decrease the total length of the edges in T,or else leave the total length the same (if we have removed an edge with length 0) while decreasing the number of edges in T. In either case the new solution is preferable to the old one. Thus a set T with n or more edges cannot be optimal. It follows that T must have exactly $n - 1$ edges, and since G' is connected, it must therefore be a tree.

The graph G' is called a *minimum spanning tree* for the graph G. This problem has many applications. For instance, suppose the nodes of G represent towns, and let the cost of an edge $\{a, b\}$be the cost of laying a telephone line from a to b. Then a minimum spanning tree of G corresponds to the cheapest possible network serving all the towns in question, provided only direct links between towns can be used (in other words, provided we are not allowed to build telephone exchanges out in the country *between* the towns). Relaxing this condition is equivalent

to allowing the addition of extra, auxiliary nodes to G. This may allow cheaper solutions to be obtained.

At first sight, at least two lines of attack seem possible if we hope to find a greedy algorithm for this problem. Clearly our set of candidates must be the set A of edges in G. One possible tactic is to start with an empty set T,and to select at every stage the shortest edge that has not yet been chosen or rejected, regardless of where this edge is situated in G. Another line of attack involves choosing a node and building a tree from there, selecting at every stage the shortest available edge that can extend the tree to an additional node. Unusually, for this particular problem both approaches work! Before presenting the algorithms, we show how the general schema of a greedy algorithm applies in this case.

- The candidates, as already noted, are the edges in G.
- A set of edges is a solution if it constitutes a spanning tree for the nodes in N.
- A set of edges is feasible if it does not include a cycle.
- The selection function we use varies with the algorithm.
- The objective function to minimize is the total length of the edges in the solution.

We also need some further terminology. We say a feasible set of edges is *promising* if it can be extended to produce not merely a solution, but an optimal solution to our problem. In particular, the empty set is always promising (since an optimal solution always exists). Furthermore, if a promising set of edges is already a solution, then the required extension is vacuous, and this solution must itself be optimal. Next, we say that an edge *leaves* a given set of nodes if exactly one end of this edge is in the set. An edge can thus fail to leave a given set of nodes either because neither of its ends is in the set.

We intend to build a least cost spanning tree, stage by stage, using the greedy method. Obviously at each stage, we choose the edge with the least weight from amongst the available edges. The other precaution we should take is that there should be no loops.

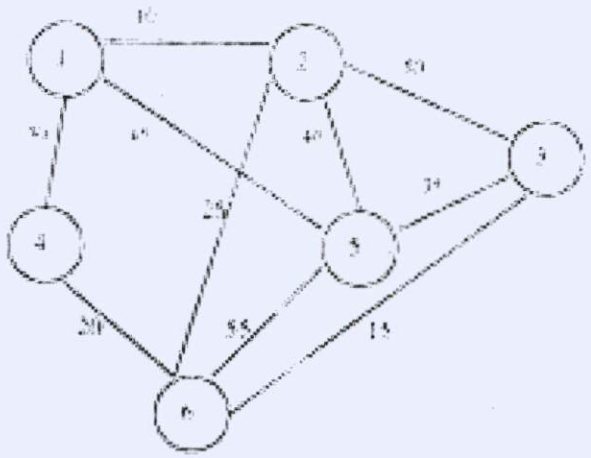
With this, in mind, we look at two algorithms Prim's and Kruskal's respectively. They work on greedy principle, and are similar except that they choose the next edge slight differently.

Prim's algorithm

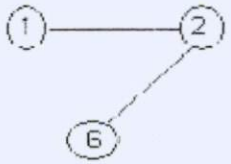
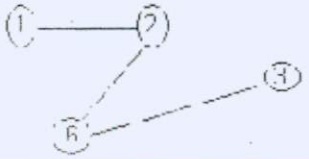
Prim's algorithm starts with the least cost edge. Then, it chooses another edge that is adjacent to this edge and is of least cost and attaches it to the first edge. The process continues as follows:

1) At each stage, choose an edge that is adjacent to any of the nodes of the partially constructed spanning tree and the least weighted amongst them.
2) If the node selected above forms a loop/circuit, then reject it and select the next edge that satisfies the criteria.
3) Repeat the process (n-1) times for the graph with n vertices.

To further clarify the situation let us trace the application of Prim's algorithm to the following graph.



Now we see how the Prim's algorithm works on it.

Edge	Cost	Spanning tree
(1,2)	10	
(2,6)	25	
(3,6)	15	
(4,6)	20	
(1, 4)	30	Forms a loop, so reject
(3,5)	35	

Now we shall devise the formal algorithm.

Algorithm: PRIM

Input : n. the no. of vertices

E. the edge set of graph G

V. the vertex set

COST. cost of each edge in terms of cost adjacency matrix

Output : T. the spanning tree

Mincost. minimum cost of T

Method :

T={ }

Let (k, l) be the edge with the minimum cost

mincost = COST (k, l)

$T = T \cup (V_k, V_l)$

While (V-T ≠ null) do

For i = all vertices in T do

For j= V - T do

Find minimum of COST(i, j) matrix

$T = T \cup V_j$

Mincost = cost(i, j)

For end

For end

While end

Algorithm ends

Complexity

The algorithm, though appears to be fairly complex, can be looked up as made up of several parts. At each stage, the nearest edge (indicating the edge with the least weight that

connects an outside vertex to a vertex that is in the partially built up spanning tree) is identified and added to the tree. It can be seen that the complexity of the algorithm is q (n^2). The complexity can be reduced to O((n + |E|) log n). You are expected to refer additional books and obtain more information about this.

Kruskal's Algorithm

The Kruskal's algorithm differs from Prim's in the following manner. It does not insist on nearness to a vertex already existing in the partial spanning tree. As long as the new incoming low cost edge does not form a loop, it is included in the tree. A broad outline of the algorithm can be listed as follows:

- Choose an edge with the lowest cost. Add it to the spanning tree. Delete it from the set of edges.
- From the set of edges choose the next low cost edge. Try it on the partial spanning tree. If no loop is created, add it to the spanning tree, otherwise discard. In either case, delete it from the set of edges.
- Repeat the operation till (n-1) edges are picked up from the set of edges and added to the spanning tree which spans over the vertex set V.

Algorithm: Kruskal

Input: n. the no. of vertices

E. the edge set of graph G

V. the vertex set

COST. cost of each edge in terms of cost adjacency matrix

Output: T. the spanning tree

Mincost. minimum cost of T

Method :

Sort the edges in increasing order of their costs.

T={ }

Let (k. l) be the first edge with the minimum cost

mincost = COST (k. l)

$T = T \cup (V_k, V_l)$

While (V-T ··· null) do

Take next edge (k. l) in sorted list of edges

If insertion of (k. l) does not form a circuit with T

T = Mincost = COST(i. j)

Else

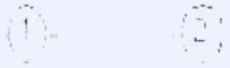
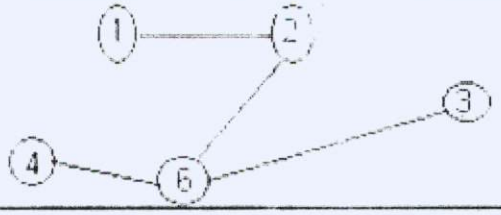
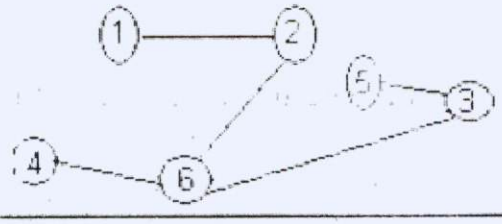
Remove (k. l)

If end

While end

Algorithm ends

We now see its effect on the graph we have considered for the Prim's algorithm.

Edge	Cost	Spanning tree
(1,2)	10	
(5,6)	15	
(4,6)	20	
(2,6)	25	
(1, 4)	30	Forms a loop, so reject
(3,5)	35	

Complexity

Since kruskal's method works on the basis of sorting the edges based on their weights, the complexity in the worst case is O(|E| log |E|). This complexity can ' reduced to O(|V| + |E|).

12.4 SUMMARY

In this unit, we have described three problems where greedy strategy is used to provide optimal solution. In the container loading problem, we used the greedy strategy in such a way that we first select the container that has least weight, then the one with the next smallest weight, and so on until either all containers have been loaded or there is not enough capacity for the next one. We have also proved that the greedy algorithm generates optimal loadings. We considered the knapsack problem and the greedy strategy to provide optimal solution. Several greedy strategies for the knapsack problem are possible. In each of these strategies, the knapsack is packed in several stages. In each stage, one object is selected for inclusion into the knapsack using greedy criterion. In the case of minimum cost spanning tree problem, we intend to build a least cost spanning tree, stage by stage, using the greedy method. Obviously at each stage, we choose the edge with the least weight from amongst the available edges. With this, in mind, we described two algorithms Prim's and Kruskal's respectively, which work on greedy principle.

12.4 KEYWORDS

1) Container loading
2) Knapsack problem
3) Minimum cost spanning tree
4) Optimal solution
5) Greedy strategy
6) Complexity

12.5 QUESTIONS FOR SELF STUDY

1) Explain the container loading problem?
2) Show that the greedy algorithm generates optimal loadings.
3) Explain the Knapsack problem and obtain an optimal solution for this problem.
4) Explain the three greedy strategies used to solve Knapsack problem.
5) Illustrate the greedy algorithm to solve a Knapsack problem with an example.

6) Explain the problem of minimum cost spanning tree.

7) Explain the Prim's algorithm to solve minimum cost spanning tree problem.

8) Explain the Kruskal's algorithm to solve minimum cost spanning tree problem.

12.6 REFERENCES

1) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.

2) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.

3) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

Unit-13

Graph as a Data Structure, Graph Representation Based on Sequential Allocation and Linked Allocation

STRUCTURE:

13.0 OBJECTIVES

After studying this unit, we will be able to explain the following:

- Basic terminologies of graph
- Graph Data Structure.
- Graph Representation based on Sequential Allocation
- Graph Representation based on Linked Allocation.

13.1 INTRODUCTION

In Unit-1 of module-1, we have defined non-linear data structure and we mentioned that trees and graphs are the examples of non-linear data structure. To recall, in non-linear data structures unlike linear data structures, an element is permitted to have any number of adjacent elements.

Graph is an important mathematical representation of a physical problem, for example finding optimum shortest path from a city to another city for a traveling sales man, so as to minimize the cost. A graph can have unconnected node. Further there can be more than one path between two nodes. Graphs and directed graphs are important to computer science for many real world applications from building compilers to modeling physical communication networks. A graph is an abstract notion of a set of nodes (vertices or points) and connection relations (edges or arcs) between them.

13.2 BASIC DEFINITIONS

Definition1: A graph G =(V,E) is a finite nonempty set V of objects called vertices together with a (possibly empty) set E of unordered pairs of distinct vertices of G called edges.

Definition2:A digraph G = (V,E) is a finite nonempty set V of verticestogether with a (possibly empty) set E of ordered pairs of vertices of G calledarcs

An arc that begins and ends at a same vertex u is called a loop. We usually (but not always) disallow loops in our digraphs. By being defined as a set, Edoes not contain duplicate (or multiple) edges/arcs between the same two vertices. For a given graph (or digraph) G, we also denote the set of vertices byV (G) and the set of edges (or arcs) by E (G) to lessen any ambiguity.

Definition3: The order of a graph (digraph) G = (V, E) is |V| sometimesdenoted by |G| and the size of this graph is |E|

Sometimes we view a graph as a digraph where every unordered edge $(u, v)$is replaced by two directed arcs (u, v) and (v,u). In this case, the size of a graphis half the size of the corresponding digraph.

Definition 4: A *walk* in a graph (digraph) G is a sequence of vertices $v_0,v_1...v_n$ such that for all $0 \leq i<n$, (v_i,v_{i+1}) is an edge (arc) in G. The length of the walk $v_0,v_1...v_n$ is the number n. A *path* is a walk in

which no vertex is repeated. A *cycle* is a walk (of length at least three for graphs) in which $v_0 = v_n$ and no other vertex is repeated; sometimes, it is understood, we omit v_n from the sequence.

In the next example, we display a graph G_1 and a digraph G_2 both of order 5. The size of the graph G_1 is 6 where $E(G_1) = \{(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 4)$ while the size of the graph G_2 is 7 where $E(G_2) = \{(0, 2), (1, 0), (1, 2), (1, 3), (3, 1), (3, 4), (4, 2)\}$.

A pictorial example of a graph G_1 and a digraph G_2 is given in figure 13.1

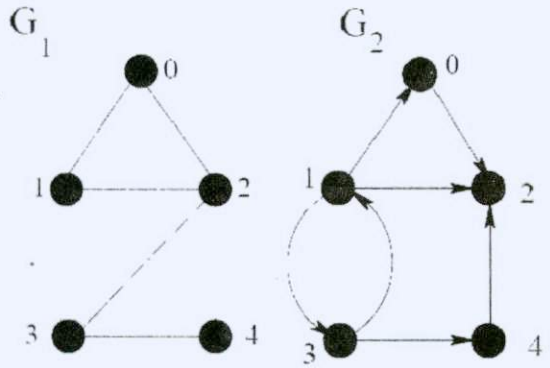


Figure 13.1 A Graph G_1 and a digraph G_2

Example 1: For the graph G_1 of Figure 13.1, the following sequences of vertices are classified as being walks, paths, or cycles.

$v_0, v_1 \ldots v_n$	is walk?	is path?	is cycle?
0 1 2 3 4	Yes	Yes	No
0 1 2 0	Yes	No	Yes
0 1 2	Yes	Yes	Yes
0 3 2	No	No	No
0 1 0	Yes	No	No

Example 2: For the graph G_1 of Figure 13.1, the following sequences of vertices are classified as being walks, paths, or cycles.

$v_0, v_1 \ldots v_n$	is walk?	is path?	is cycle?
0 1 2 3 4	No	No	No
0 2 4	No	No	No
3 1 2	Yes	Yes	No

1 3 1	Yes	No	Yes
3 1 3 1 0	Yes	No	No

Definition 5: A graph G is *connected* if there is a path between all pairs of vertices *u* and *v* of V(G). A digraph G is *strongly connected* if there is a path from vertex *u* to vertex *v* for all pairs *u* an *v* in V(G).

In Figure 13.1, the graph G_1 is connected by the digraph G_2 is not strongly connected because there are no arcs leaving vertex 2. However, the underlying graph G_2 is connected.

Definition 6: In a graph, the *degree* of a vertex *v*, denoted by *deg(v)*, is the number of edges incident to *v*. For digraphs, the *out-degree* of a vertex *v* is the number of arcs $\{(v, x) \in E \mid x \in V\}$ incident from *v* (leaving *v*) and the *in-degree* of vertex *v* is the number of arcs $\{(v, x) \in E \mid x \in V\}$ incident to *v* (entering *v*).

For a graph, the in-degree and out-degree's are the same as the degree. For out graph G_1, we have deg(0) = 2, deg(2) = 4, deg(3) = 2 and deg(4) = 2. We may concisely write this as a degree sequence (2, 2, 4, 2, 2) id there is a natural ordering (e.g., 0, 1, 2, 3, 4) of the vertices. The in-degree sequence and out-degree sequence of the digraph G_2 are (1, 1, 3, 1, 1) and (1, 3, 0, 2, 1), respectively. The degree of a vertex of a digraph is sometimes defined as the sum of its in-degree and out-degree. Using this definition, a degree sequence of G_2 would be (2, 4, 3, 3, 2).

Definition 7: A *weighted graph*is a graph whose edges have weights. These weights can be thought as cost involved in traversing the path along the edge. Figure 13.2 shows a weighted graph.

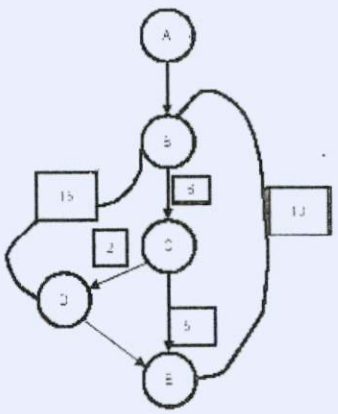


Figure 13.2 A weighted graph

Definition 8: If removal of an edge makes a graph disconnected then that edge is called *cutedge* or *bridge*.

Definition 9: If removal of a vertex makes a graph disconnected then that vertex is called *cutvertex*.

Definition 10: A connected graph without a cycle in it is called a *tree*. The pendent vertices of a tree are called leaves.

Definition 11: A graph without self loop and parallel edges is called a simple graph.

Definition 12: A graph which can be traced without repeating any edge is called an Eulerian graph. If all vertices of a graph happen to be even degree then the graph is called an Eulerian graph.

Definition 13: If two vertices of a graph are odd degree and all other vertices are even then it is called open Eulerian graph. In open Eulerian graph the starting and ending points must be odd degree vertices.

Definition 14: A graph in which all vertices can be traversed without repeating any edge but can have any number of edges is called Hamiltonian graph.

Definition 15: Total degree of a graph is twice the number of edges. That is, the total degree = 2* |E|

Corollary: Number of odd degree vertices of a graph is always even.

- Total degree = Sum of degrees of all vertices = 2 * |E| = Even.
- Sum of degrees of all even degree vertices + Sum of degrees of all odd degree vertices = Even.
- Even + Sum of vertices of all odd degree vertices = Even.
- Sum of vertices of all odd degree vertices = Even – Even = Even.

13.3 GRAPH DATA STRUCTURE

We can formally define graph as an abstract data type with data objects and operations on it as follows:

Data objects: A graph G of vertices and edges. Vertices represent data objects.

Operations:

- **Check-Graph-Empty(G):** Check if graph G is empty - Boolean function
- **Insert-Vertex(G, V):** Insert an isolated vertex V into a graph G. Ensure that vertex V does not exist in G before insertion.
- **Insert-Edge(G, *u*, *v*):** Insert an edge connecting vertices *u*, *v* into a graph G. Ensure that an edge does not exist in G before insertion.
- **Delete-Vertex(G, V):** Delete vertex V and all the edges incident on it from the graph G. Ensure that such a vertex exists in the graph G before deletion.
- **Delete-Edge(G, *u*, *v*):** Delete an edge from the graph G connecting the vertices *u*, *v*. Ensure that such an edge exists before deletion.
- **Store-Data(G, V, Item):** Store Item into a vertex V of graph G.
- **Retrieve-Data(G, V, Item):** Retrieve data of a vertex V in the graph G and return it in Item.
- **BFT(G):** Perform Breath First Traversal of a graph.
- **DFT(G):** Perform Depth First Traversal of a graph.

13.4 REPRESENTATION OF GRAPHS

A graph is a mathematical structure and it is required to be represented as a suitable data structure so that very many applications can be solved using digital computer. The representation of graphs in a computer can be categorized as (i) *sequential representation* and (ii) *linked representation* The sequential representation makes use of an array data structure where as the linked representation of a graph makes use of a singly linked list as its fundamental data structure.

Sequential Representation of Graphs

The sequential or the matrix representations of graphs have the following methods:

- Adjacency Matrix Representation
- Incidence Matrix Representation

Adjacency Matrix Representation

A graph with n nodes can be represented as n x n Adjacency Matrix A such that an element A_{ij}

A_{ij} = 1 if there is an edge between nodes i and j

0 Otherwise

Note that the number of 1s in a row represents the out degree of a node. In case ofundirected graph, the number of 1s represents the degree of the node.Total number of 1s in the matrix represents number of edges. Figure 13.3(a) shows a graph and Figure 13.3(b) shows its adjacency matrix.

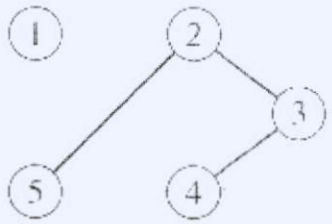
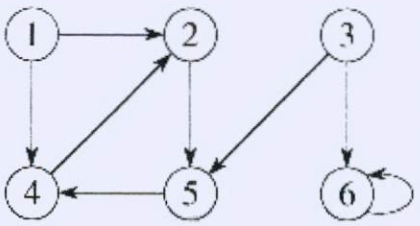


Figure 13.3(a) Graph

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Figure 13.3(b) Adjacency matrix

Figure 13.4(a) shows a digraph and Figure 13.4(b) shows its adjacency matrix.



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Figure 13.4(a) Digraph

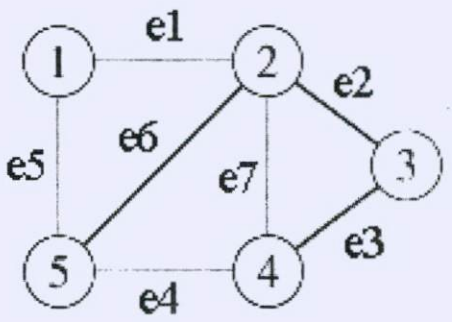
Figure 13.4(b) Adjacency matrix

Incidence Matrix Representation

Let G be a graph with *n* vertices and *e* edges. Define an *n* x *e* matrix M = [m_{ij}] whose *n* rows corresponds to *n* vertices and e columns correspond to *e* edges, as

$$A_{ij} = \begin{cases} 1 & e_j \text{ incident upon } v_i \\ 0 & \text{Otherwise} \end{cases}$$

Matrix M is known as the *incidence matrix* representation of the graph G. Figure 1.5(a) shows a graph and Figure 1.5(b) shows its incidence matrix.



	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_1	1	0	0	0	1	0	0
v_2	1	1	0	0	0	1	1
v_3	0	1	1	0	0	0	0
v_4	0	0	1	1	0	0	1
v_5	0	0	0	1	1	1	0

Figure 1.5(a) Undirected graph

Figure 1.5(b) Incidence matrix

The incidence matrix contains only two elements, 0 and 1. Such a matrix is called a *binary matrix* or a *(0, 1)-matrix*.

The following observations about the incidence matrix can readily be made:

1. Since every edge is incident on exactly two vertices, each column of in an incidence matrix has exactly two1's.
2. The number of 1's in each row equals the degree of the corresponding vertex.
3. A row with all 0's, therefore, represents an isolated vertex.

Linked Representation of Graphs

The linked representation of graphs is referred to as adjacency list representation and is comparatively efficient with regard to adjacency matrix representation. Given a graph G with n vertices and e edges, the adjacency list opens n head nodes corresponding to the n vertices of graph G, each of which points to a singly linked list of nodes, which are adjacent to the vertex representing the head node. Figure 1.6(a-b) shows an undirected its linked representation. Similarly, Figure 1.7(a-b) shows a digraph and its linked representation.

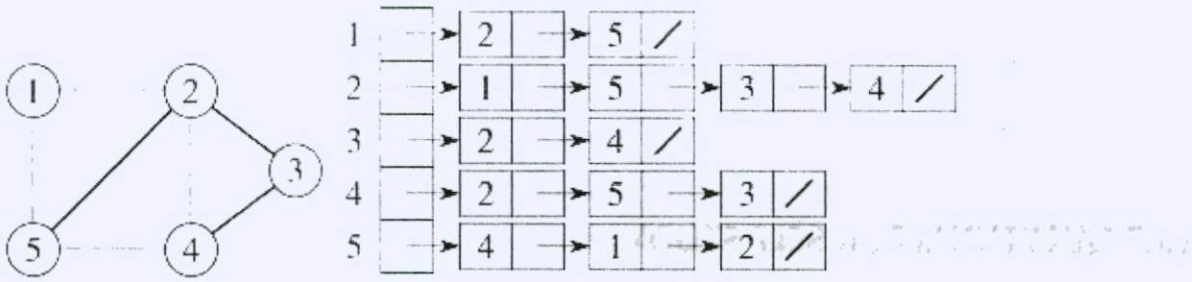


Figure 1.6(a) Undirected graph

Figure 1.6(b) Linked representation of a graph

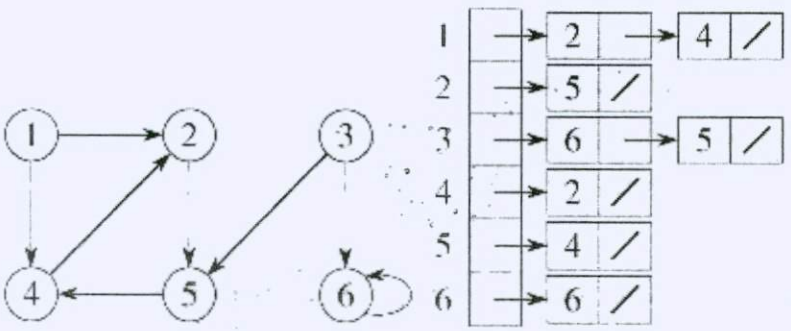


Figure 1.7(a) Digraph

Figure 1.7(b) Linked representation of a graph

13.5 SUMMARY

- Graphs are non-linear data structures. Graph is an important mathematical representation of a physical problem.
- Graphs and directed graphs are important to computer science for many real world applications from building compilers to modeling physical communication networks.
- A graph is an abstract notion of a set of nodes (vertices or points) and connection relations (edges or arcs) between them.
- The representation of graphs in a computer can be categorized as (i) *sequential representation* and (ii) *linked representation.*
- The sequential representation makes use of an array data structure where as the linked representation of a graph makes use of a singly linked list as its fundamental data structure.

13.6 KEYWORDS

Non-linear data structures, Undirected graphs, Directed graphs, Walk, Path, Cycle, Cutedge, Cutvertex, In-degree, Out-degree, Pendent edge, Eulerian graph, Hamiltonian graph, Adjacency matrix, Incidence matrix.

13.7 QUESTIONS FOR SELF STUDY

1) Define graph and directed graph.
2) Define walk, path and cycle with reference to graph.
3) Define connected graph and strongly connected graph and give examples.
4) Define in-degree and out-degree of a graph and give some examples.
5) Define cutedge, cutvertex, pendent vertex, Hamiltonian graph, Eulerian graph.
6) Show that the number of odd degree vertices of a graph is always even.
7) Explain graphs as a data structure.
8) Explain two different ways of sequential representation of a graph with an example.
9) Explain the linked representation of an undirected and directed graph.

13.7 REFERENCES

1) Sartaj Sahni, 2000, Data structures, algorithms and applications in C++, McGraw Hill international edition.
2) Horowitz and Sahni, 1983, Fundamentals of Data structure, Galgotia publications
3) Narsingh Deo, 1990, Graph theory with applications to engineering and computer science,Prentice hall publications.
4) Tremblay and Sorenson, 1991, An introduction to data structures with applications, McGrawHill edition.
5) C and Data Structures by Practice- Ramesh, Anand and Gautham.
6) Data Structures and Algorithms: Concepts, Techniques and Applications by GAV Pai. Tata McGraw Hill, New Delhi.

Unit-14

Binary Trees, Representation of Binary Trees based on Sequential and Linked Allocation Method

STRUCTURE:

14.0 OBJECTIVES

After studying this unit, we will be able to

- Explain the basic terminologies of trees.
- Discuss the importance of Binary Tree Data Structure.
- Describe the representation of binary trees based on Sequential Allocation.
- Explain the linked representation of binary trees.

- List out the advantages of representing Binary Trees using linked allocation.

14.1 INTRODUCTION

In Unit-13, we have discussed graphs, which are non-linear data structures. Similarly, trees are also non-linear data structures, which are very useful in representing hierarchical relationships among the data items. For example, in real life, if we want to express the relation ship exists among the members of the family then we use non linear structures like trees. Organizing the data in a hierarchical structure plays a very important role for most of the applications, which involve searching. Trees are the most useful and widely used data structure in Computer Science in the areas of data storage, parsing, evaluation of expressions, and compiler design.

14.2 DEFINITION AND BASIC TERMINOLOGIES OF TREES

Definition: A tree is defined as a finite set of one or more nodes such that

(i) there is a specially designated ***node*** called the ***root*** and

(ii) the rest of the nodes could be partitioned in to t disjoint sets ($t \geq 0$) each set representing a tree T_i, $i = 1, 2, 3, \ldots, t$ known as ***subtree*** of the tree.

A node in the definition of the tree represents an item of information and the links between the nodes termed as branches, represent an association between the items of information. Figure 2.1 shows a tree.

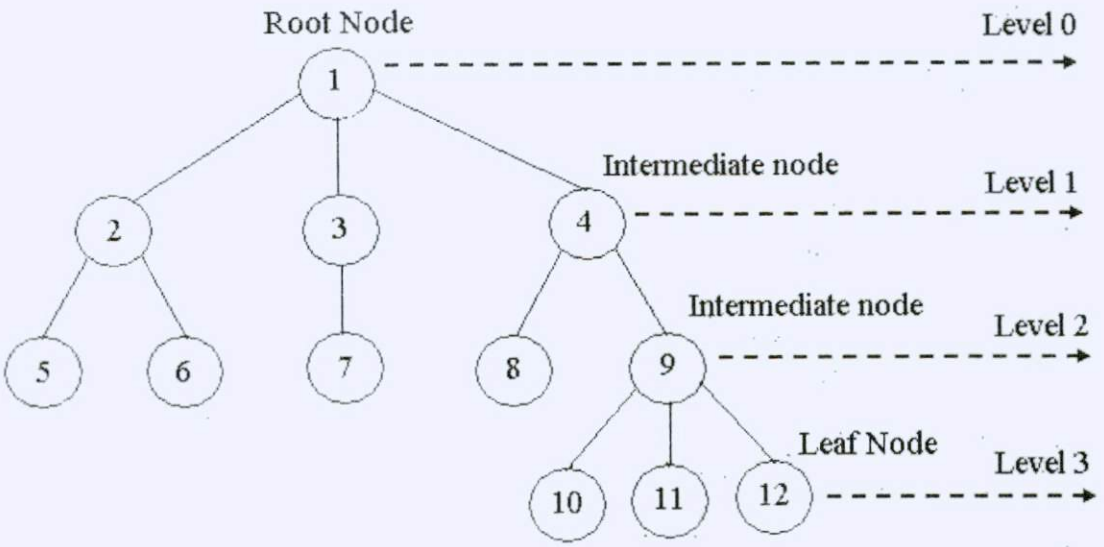


Figure 2.1 An example tree

In the above figure, node 1 represents the root of the tree, nodes 2, 3, 4 and 9 are all intermediate nodes and nodes 5, 6, 7, 8, 10, 11 and 12 are the leaf nodes of the tree. The definition of the tree emphasizes on the aspect of (i) *connectedness* and (ii) absence of *loops* or *cycles*. Beginning from the root node, the structure of the tree permits connectivity of the root to every other node in the tree. In general, any node is reachable from any where in the tree. Also, with branches providing links between the nodes, the structure ensures that no set of nodes link together to form a closed loop or cycle.

Some Properties of Tree

1. There is one and only one path between every pair of vertices in a tree, T.
2. A tree with *n* vertices has *n*-1 edges.
3. Any connected graph with n vertices and *n*-1 edges is a tree.
4. A graph is a tree if and only if it is minimally connected.

Therefore a graph with *n* vertices is called a tree if

1. G is connected and is circuit less, or
2. G is connected and has *n*-1 edges, or
3. G is circuit less and has *n*-1 edges, or
4. There is exactly one path between every pair of vertices in G, or
5. G is a minimally connected graph.

There are several basic terminologies associated with trees. There is a specially designated node called the ***root node***. The number of subtrees of a node is known as the ***degree*** of the node. Nodes that have zero degree are called ***leaf nodes*** or ***terminal nodes***. The rest of them are called ***intermediate nodes***. The nodes, which hang from branches emanating from a node, are called as ***children*** and the node from which the branches emanate is known as the ***parent node***. Children of the same parent node are referred to as ***siblings***. The ***ancestors*** of a given node are those nodes that occur on the path from the root to the given node. The ***degree*** of a tree is the maximum degree of the node in the tree. The ***level*** of

the node is defined by letting the root node to occupy ***level 0***. The rest of the nodes occupy various levels depending on their association. Thus, if parent node occupies ***level i*** then, its children should occupy ***level i+1***. This renders a tree to have a ***hierarchical structure*** with root occupying the top most level of 0. The ***height*** or ***depth*** of a tree is defined to be the maximum level of any node in the tree. A ***forest*** is a set of zero or more disjoint trees. The removal of the root node from a tree results in a forest.

14.3 BINARY TREES

A binary tree has the characteristic of all nodes having at most two branches, that is, all nodes have a degree of at most 2. Therefore, a binary tree can be empty or consist of a root node and two disjointed binary trees termed left subtree and right subtree. Figure 2.2 shows an example binary tree.

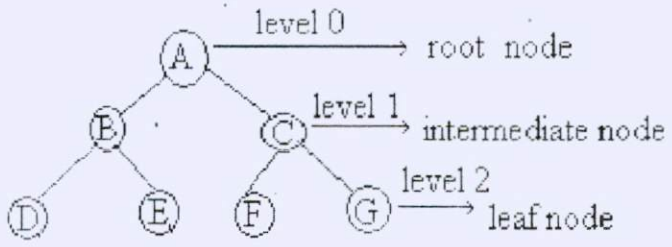


Figure 2.2 An example binary tree

The number of levels in the tree is called the "**depth**" of the tree. A "**complete**" binary tree is one which allows sequencing of the nodes and all the previous levels are maximally accommodated before the next level is accommodated. i.e., the siblings are first accommodated before the children of any one of them. And a binary tree, which is maximally accommodated with all leaves at the same level is called "**full**" binary tree. A full binary tree is always complete but a complete binary tree need not be full. Fig. 2.2 is an example for a full binary tree and Figure 2.3 illustrates a complete binary tree.

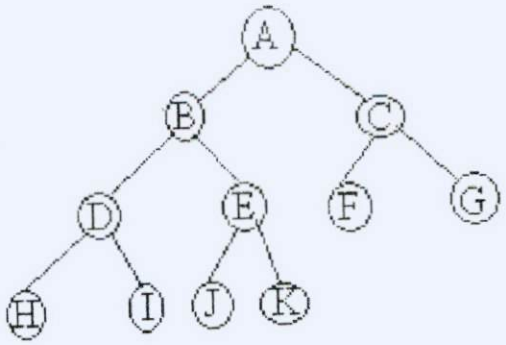


Figure 2.3 A complete binary tree

The maximum number of vertices at each level in a binary tree can be found out as follows:

At level 0: 2^0 number of vertices

At level 1: 2^1 number of vertices

At level 2: 2^2 number of vertices

...

At level i: 2^i number of vertices

Therefore, maximum number of vertices in a binary tree of depth 'l' is:

$$2^0 + 2^1 + 2^2 + \ldots + 2^l$$

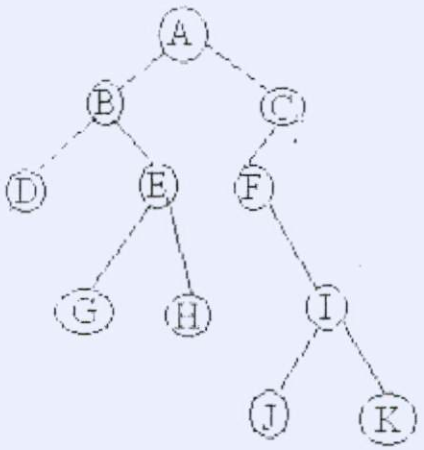
$$\text{i.e., } \sum 2^k = 2^{l+1} - 1 \quad \text{for} k = 0 \text{ to } l$$

14.4 REPRESENTATION OF A BINARY TREE

Binary Tree can be represented using sequential as well as linked data structures. In sequential data structures, we have two ways of representing the binary tree. One is through the use of Adjacency matrices and the other is through the use of Single dimensional array representation.

Adjacency Matrix Representation

A two dimensional array can be used to store the adjacency relations very easily and can be used to represent a binary tree. In this representation, to represent a binary tree with *n* vertices we use *n*×*n* matrix. Figure 2.4(a) shows a binary tree and Figure 2.4(b) shows its adjacency matrix representation.



	A	B	C	D	E	F	G	H	I	J	K
A		L	R								
B				L	R						
C						L					
D											
E							L	R			
F									R		
G											
H											
I										L	R
J											
K											

(a) A binary tree

(b) Adjacency matrix representation

Figure 2.4 A binary tree and its adjacency matrix representation

Here, the row indices correspond to the parent nodes and the column corresponds to the child nodes. i.e., a row corresponding to the vertex v_i having the entries 'L' and 'R' indicate that v_i has its left child, the index corresponding to the column with the entry 'L' and has its right child, the index corresponding to the column with the entry 'R'. The column corresponds to vertex $v_i$with no entries indicate that it is the root node. All other columns have only one entry. Each row may have 0, 1 or 2 entries.Zero entry in the row indicates that the corresponding vertex v_i is a leaf node, only one entry indicates that the node has only one child and two entries indicate that the node has both the left and right children. The entry "L" is used to indicatethe left child and "R" is used to indicate the right child entries.

From the above representation, we can understand that the storage space utilization is not efficient. Now, let us see the space utilization of this method of binary tree representation. Let '*n*' be the number of vertices. The space allocated is *n* x *n* matrix. i.e., we have n^2number of locations allocated,

but we have only n-1 entries in the matrix. Therefore, the percentage of space utilization is calculated as follows:

$$\frac{n-1}{n^2} \cong n = \frac{1}{n} \times 100\%$$

The percentage of space utilized decreases as n increases. For large 'n', the percentage of utilization becomes negligible. Therefore, this way of representing a binary tree is not efficient in terms of memory utilization.

Single Dimensional Array Representation

Since the two dimensional array is a sparse matrix, we can consider the prospect of mapping it onto a single dimensional array for better space utilization. In this representation, we have to note the following points:

- The left child of the i^{th} node is placed at the $2i^{th}$ position.
- The right child of the i^{th} node is placed at the $(2i+1)^{th}$ position.
- The parent of the i^{th} node is at the $(i/2)^{th}$ position in the array.

If l is the depth of the binary tree then, the number of possible nodes in the binary tree is 2^{l+1}-1. Hence it is necessary to have 2^{l+1}-1 locations allocated to represent the binary tree.

If 'n' is the number of nodes, then the percentage of utilization is

$$\frac{n-1}{2^{l+1}-1} \times 100$$

Figure 2.5shows a binary tree and Figure 2.6 shows its one-dimensional array representation.

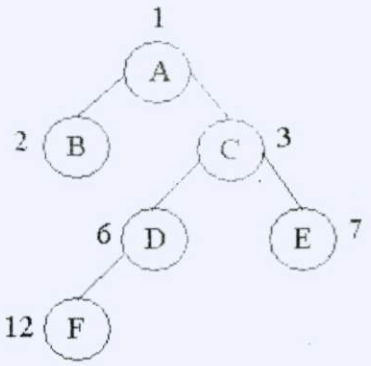


Figure 2.5 A binary tree

1	2	3	4	5	6	7	8	9	10	11	12
A	B	C			D	E					F

Figure 2.6 One-dimensional array representation

For a complete and full binary tree, there is 100% utilization and there is a maximum wastage if the binary tree is right skewed or left skewed, where only $l+1$ spaces are utilized out of the $2^{l+1} - 1$ spaces.

$$\text{i.e., } \frac{l+1}{2^{l+1}-1} \times 100$$

An important observation to be made here is that the organization of the data in the binary tree decides the space utilization of the representation used.

14.4 LINKED REPRESENTATION OF BINARY TREES

In previous section, we have discussed the representation of a binary tree using arrays (sequential allocation) and we understand the merits and demerits of sequential representation of binary trees. In this section, we will be discussing the linked list representation of a binary tree and their advantages.

The linked representation of a binary tree has the node structure shown in Figure 2.7. Here, the node besides the DATA field needs two pointers LCHILD and RCHILD to point to the left and right child

nodes respectively. The tree is accessed by remembering the pointer to the root node of the tree. Figure 2.8 shows an example binary tree and Figure 2.9 shows its linked representation.

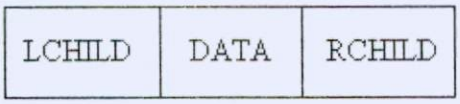


Figure 2.7 Structure of Node in Binary Tree

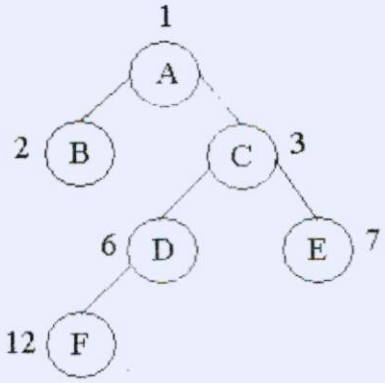


Figure 2.8 A Binary Tree

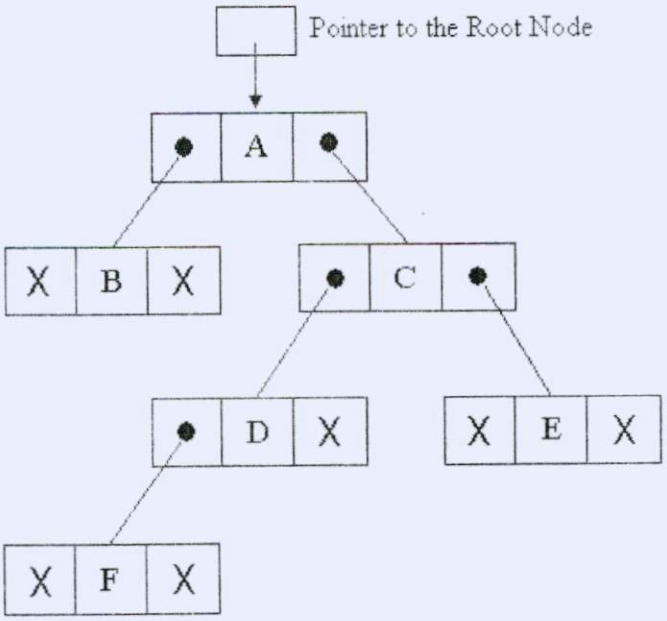


Figure 2.9 Linked representation of a Binary Tree

In the binary tree T shown in Figure 2.9, LCHILD(T) refers to the node storing B and RCHILD(T) refers to the node storing C and so on. The following are some of the important observations regarding the linked representation of a binary tree:

- If a binary tree has *n* nodes then the number of pointers used in its linked representation is 2 * *n*
- The number of null pointers used in the linked representation of a binary tree with *n* nodes is *n* + 1.

However, in linked representation, it is difficult to determine a parent given a child node. In any case, if an application so requires, a fourth field PARENT may be included in the structure.

Figure 2.10 illustrates the node structure of a binary tree, which incorporates the parent information and Figure 2.11 shows an instance of a binary tree with additional parent information.

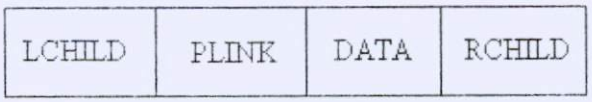


Figure 2.10 Node structure with parent link

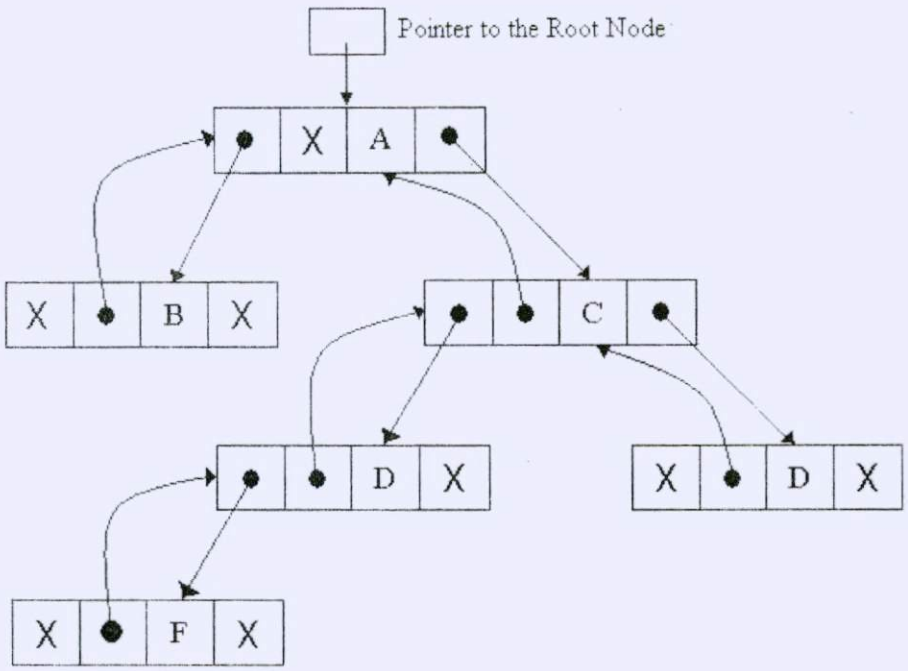


Figure 2.11 Linked representation of a binary tree with access to parent node.

From Figure 2.10, we can observe that the PLINK field of the node contains the address of its parent node. If we represent a binary tree with this node structure as shown in Figure 2.11, we can find the parent node of any given node by just following the PLINK pointer. Also observe that the PLINK field of the root node is null, which indicates that there is no parent node a root node.

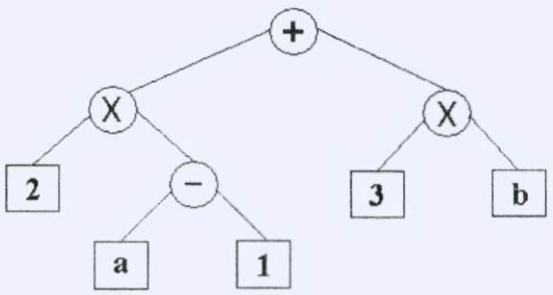
In this representation, ordering of nodes is not mandatory and we require a header to point to the root node of the binary tree. If there are '*n*' nodes to be represented, only '*n*' node-structures will be allocated. This means that there will be 2*n* link fields. Out of 2*n* link fields only (*n*-1) will be actually used to point to the next nodes and the remaining are wasted.Therefore the percentage of utilization is given as:

$$\frac{n-1}{2n} = \frac{1}{n} \times 100 = 50\%$$

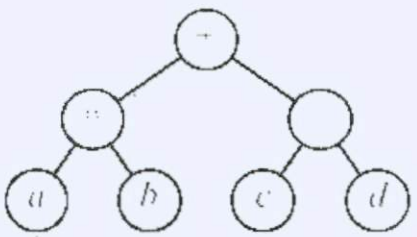
The linked representation of a binary tree discussed above can be used to effectively represent the arithmetic expressions and decision process as described below.

Arithmetic Expression Tree

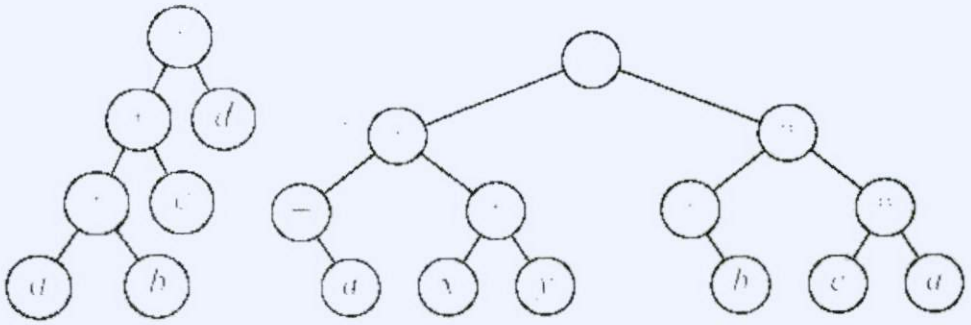
Binary tree associated with an arithmetic expression is called an arithmetic expression tree. The internal nodes represent the operators and the external nodes represent the operands. Figure 2.12 shows the arithmetic expression trees.



(a) (2 *(a – 1) +(3 *b))



(b) (a * b) + (c / d)



(c) ((a + b) + c) + d)

(d) ((-a) + (x + y)) / ((+ b) * (c * a))

Figure 2.12 Example expression trees

Decision Tree

Binary tree associated with a decision process is called a decision tree. The internal nodes represent the questions with yes/no answer and the external nodes represent the decisions. Figure 2.13 shows the decision tree for the biggest of three numbers.

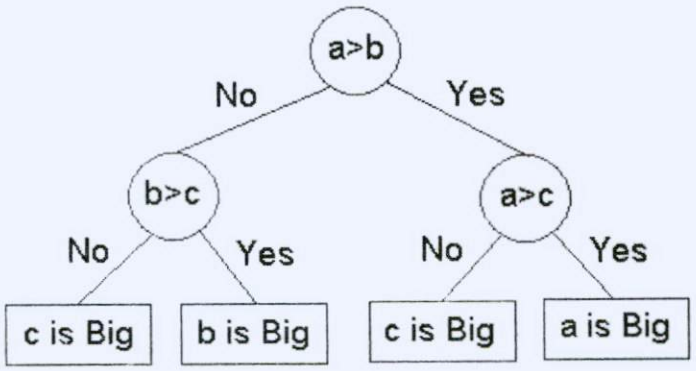


Figure 2.13 Decision tree

14.5 SUMMARY

- Trees and binary trees are non-linear data structures, which are inherently two dimensional in structures.

- While trees are non empty and may have nodes of any degree, a binary tree may be empty r hold nodes of degree at most two.
- The terminologies of root node, height, level, parent, children, sibling, ancestors, leaf or terminal nodes and non-terminal nodes are applicable to both trees and binary trees.
- While trees are efficiently represented using linked representations, binary trees are represented using both array and linked representations.
- The deletion and insertion operations can be performed efficiently in the linked representation of binary trees. But such operations are difficult in case of array based representations of a binary tree.

14.6 KEYWORDS

Trees, binary trees, non-linear data structures, root node, height, level, parent, children, sibling, ancestors, leaf or terminal nodes, non-terminal nodes, linked representation, array representation, expression tree, decision tree, left child, right child.

14.7 QUESTIONS FOR SELF STUDY

1) Define tree and binary tree.
2) Differentiate complete and full binary trees
3) What is the maximum number of nodes in a binary tree of level 7, 8 and 9
4) Explain the two techniques used to represent a binary tree based on sequential allocation method.
5) What are the advantages of single dimensional array based representation over adjacency matrix representation of a binary tree?
6) Describe the node structure used to represent a node in a binary tree.
7) Explain linked representation of a binary tree with an example.
8) What are the advantages of linked representation over array based representation of a binary?
9) How do you know the parent node of a given node in linked representation of a binary tree? Explain the linked list structure with an example.
10) What are expression trees and decision trees? Explain with suitable examples.

14.8 REFERENCES

1) Sartaj Sahni, 2000, Data structures, algorithms and applications in C++, McGraw Hill international edition.
2) Horowitz and Sahni, 1983, Fundamentals of Data structure, Galgotia publications
3) Horowitz and Sahni, 1998, Fundamentals of Computer algorithm, Galgotia publications.
4) Narsingh Deo, 1990, Graph theory with applications to engineering and computer science, Prentice hall publications.
5) Tremblay and Sorenson, 1991, An introduction to data structures with applications, McGraw Hill edition.
6) Dromey R. G., 1999, How to solve it by computers, Prentice Hall publications, India.

Unit-15

Traversal of Binary Trees and Operations on Binary Trees

STRUCTURE:

15.0 OBJECTIVES

After studying this unit, we will be able to

- Explain the traversal of binary trees.
- List out the various ways of traversing a binary tree.
- Discuss the various operations on binary trees.
- Evaluate the usefulness of binary search trees.

15.1 INTRODUCTION

In this unit, we will introduce an important operation performed on binary trees called traversal. Traversal is the process of visiting all the vertices of the tree in a systematic order. Systematic means that every time the tree is traversed it should yield the same result.This processis not as commonly used as finding, inserting, and deleting nodes. One reasonfor this is that traversal is not particularly fast. But traversing a tree hassome surprisingly useful applications and is theoretically interesting. In addition, we discuss some other important operations on binary trees.

15.2 TRAVERSAL OF A BINARY TREE

A traversal of a binary tree is where its nodes are visited in a particular but repetitive order, rendering a linear order of nodes or information represented by them. There are three simple ways to traverse a tree. They are called *preorder,inorder*, and *postorder*. In each technique, the left subtree is traversed recursively, the right subtree is traversed recursively, and the root is visited. What distinguishes the techniques from one another is the order of those three tasks. The following sections discuss these three different ways of traversing a binary tree.

Preorder Traversal

In this traversal, the nodes are visited in the order of root, left child and then right child.

- Process the root node first.
- Traverse left sub-tree.
- Traverse right sub-tree.

Repeat the same for each of the left and right subtrees encountered. Here, the leaf nodes represent the stopping criteria. The pre-order traversal sequence for the binary tree shown in Figure15.1 is: A B D E H I C F G

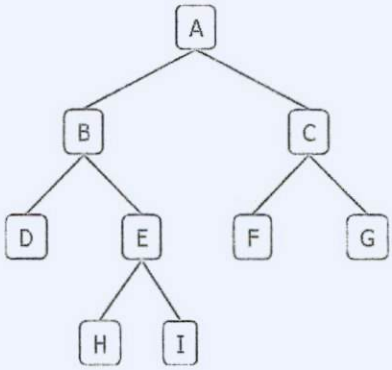


Figure15.1 A binary tree

Inorder Traversal

In this traversal, the nodes are visited in the order of left child, root and then right child. i.e., the left sub-tree is traversed first, then the root is visited and then the right sub-tree is traversed. The function must perform onlythree tasks.

- Traverse the left subtree.
- Process the root node.
- Traverse the right subtree.

Remember that visiting a node means doing something to it: displaying it, writing it to a file and so on.The inorder traversal sequence for the binary tree shown in Figure15.1 is: D B H E I A F C G.

Postorder Traversal

In this traversal, the nodes are visited in the order of left child, right child and then the root. i.e., the left sub-tree is traversed first, then the right sub-tree is traversed and finally the root is visited. The function must perform the following tasks.

- Traverse the left subtree.
- Traverse the right subtree.
- Process the root node.

The postorder traversal sequence for the binary tree shown in Figure4.1 is: D H I E B F G C A.

Traversal of a Binary Tree Represented in an Adjacency Matrix

The steps involved in traversing a binary tree from the adjacency matrix representation, firstly requires finding out the root node. Then it entails to traverse through the left subtree and then the right subtree in specific orders. In order to remember the nodes already visited, it is necessary to maintain a stack data structure. Thus, following are the steps involved in traversing trough the binary tree given in an adjacency matrix representation.

- Locate the root (the column sum is zero for the root)
- Display
- Push in to the stack
- Scan the row in search of 'L' for the left child information
- Pop from the stack
- Scan the row in search of 'R' for the right child information
- Check if array IsEmpty().
- Stop

Sequencing the above stated steps helps us in arriving at preorder, inorder and postorder traversal sequences.

Binary Tree Traversal from 1D Array Representation

Preorder Traversal

Algorithm: Preorder Traversal

Input: A[], one dimensional array representing the binary tree

i, the root address //initially *i*=1

Output: Preorder sequence

Method:

If (A[*i*] ≠ 0)

Display(A[*i*])

Preorder Traversal (A, 2*i*)

Preorder Traversal (A, 2*i* +1)

If end

Algorithm ends

Inorder Traversal

Algorithm: Inorder Traversal

Input: A[], one dimensional array representing the binary tree

i, the root address //initially *i*=1

Output: Inorder sequence

Method:

If (A[*i*] ≠ 0)

Inorder Traversal (A, 2*i*)

Display(A[*i*])

Inorder Traversal (A, 2*i* +1)

If end

Algorithm ends

Postorder Traversal

Algorithm: Postorder Traversal

Input: A[], one dimensional array representing the binary tree

i, the root address //initially *i*=1

Output: Postorder sequence

Method:

If (A[*i*] ≠ 0)

Postorder Traversal (A, 2*i*)

Postorder Traversal (A, 2*i* +1)

Display(A[*i*])

If end

Algorithm ends

Binary Tree Traversal in Linked Representation

We have already studied that every node of a binary tree in linked representation has a structure which has links to the left and right children. The algorithms for traversing the binary tree in linked representation are given below.

Algorithm: Preorder Traversal

Input:*bt*, address of the root node

Output: Preorder sequence

Method:

If(*bt*≠ NULL)

Display([*bt*].data)

Preorder Traversal([*bt*].Lchild)

Preorder Traversal([*bt*].Rchild)

If end

Algorithm ends.

Algorithm: Inorder Traversal

Input:*bt*, address of the root node

Output: Inorder sequence

Method:

```
If(bt≠ NULL)
        Inorder Traversal([bt].Lchild)
        Display([bt].data)
        Inorder Traversal([bt].Rchild)
If end
```

Algorithm ends.

Algorithm: Postorder Traversal

Input:*bt*, address of the root node

Output: Postorder sequence

Method:

```
If(bt≠ NULL)
        Postorder Traversal([bt].Lchild)
        Postorder Traversal([bt].Rchild)
        Display([bt].data)
If end
```

Algorithm ends.

15.3 OPERATIONS ON BINARY TREE

We have already discussed an important operation performed on binary trees called traversal. The various other operations that can be performed on binary trees are discussed as follows.

Insertion

To insert a node containing an item into a binary tree, first we need to find the position for insertion. Suppose the node pointed to by ***temp*** has to be inserted whose information field contains the item ***J*** as shown in Figure 15.2, we need to maintain an array say ***D***, which contains only the directions where the node temp has to be inserted.

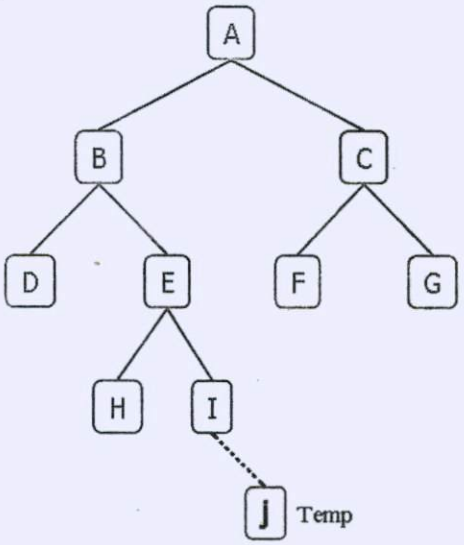


Figure 15.2 To insert a item ***J***

If ***D*** contains '**LRLR**', from the root node, first move towards left (**L**), then right (**R**), then left (**L**) and finally move towards right (**R**). If the pointer points to ***null*** at that position, node ***temp*** can be inserted otherwise, it cannot be inserted. To achieve this, one has to start from the root node. Let us use two pointers ***prev*** and ***cur*** where ***prev*** always points to parent node and ***cur*** points to child node. Initially ***cur*** points to root node and ***prev*** points to ***null***. To start with one can write the following statements.

prev = null

cur = root

Now, keep updating the node pointed to by ***cur*** towards left if the direction is '**L**' otherwise, update towards right. Once all directions are over, if current points to ***null***, insert the node ***temp*** towards left or right based on the last direction. Otherwise, display error message. This procedure can be algorithmically expressed as follows.

Algorithm: Insert Node

Input: ***root***, address of the root node

e, element to be inserted

D, direction array

Output: Tree updated

Method:

1. temp = Getnode()
2. temp.info = e
3. temp.llink = temp.rlink = null
4. if (root = null) return temp // Node inserted for the first time
5. prev = null, cur = root, k = 0
6. While (k < strlen(D)) DO

   If (cur = null) exit

   prev = cur

   if (D[k] = 'L') then

   cur = cur.llink

   else

   cur = cur.rlink

   Whileend

7. If ((cur ≠ null) OR k ≠ strlen(D)) then

   Display "Insertion not possible"

Free(temp)

Return(root)

8. If (D[k – 1] = 'L') then
   prev.llink = temp

   Else

   prev.rlink = temp

   Ifend

9. root
10. stop

Algorithm ends

Searching

To search for an item in a tree, we can traverse a tree in any of the (inorder, preorder, postorder) order to visit the node. As we visit the node, we can compare the item to be searched with the data item stored in information field of the node. If found then the search is successful otherwise, search is unsuccessful. A recursive inorder traversal technique used for searching an item in binary tree is presented below.

Algorithm: Search(item, root, flag)

Input: item, data to be searched

root, address of the root node

flag, status variable

Output: Item found or not found

Method:

1. if (root = null then
   flag = false

exit

ifend

2. Search (item, root.llink, flag)
3. if (item = root.info) then
   flag = true

   exit

   ifend

4. Search (item, root.rlink, flag)
5. if (flag = true) then display "Data item is found"
   else display "Data item is not found"

   ifend

6. stop

Algorithm ends

Deletion

Deletion of a node from a binary tree involves searching for a node which contains the data item. If such a node is found then that node is deleted; otherwise, appropriate message is displayed. If the node to be deleted is a leaf node then the deletion operation is a simple task. Otherwise, appropriate modifications need to be done to update the binary tree after deletion. This operation is explained in detail considering another form of a binary tree called *binary search tree*.

Binary Search Tree

Binary Search Tree (BST) is an ordered Binary Tree in that it is an empty tree or value of root node isgreater than all the values in Left Sub Tree(LST) and less than all the values of Right Sub Tree (RST).Right and Left sub trees are again binary sub trees by themselves.Figure15.3(a) shows an example binary tree where as Figure 15.3(b) is not a binary search tree but a binary tree.

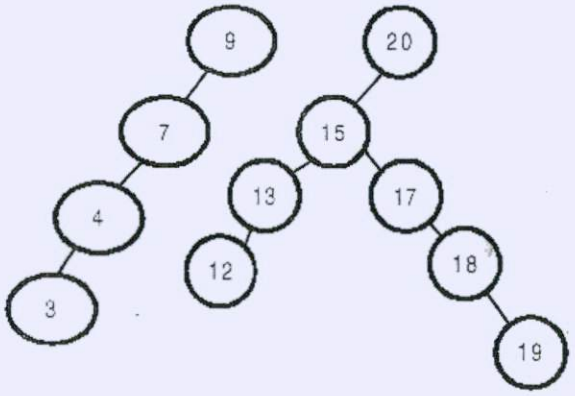


Figure 15.3(a) A binary search tree

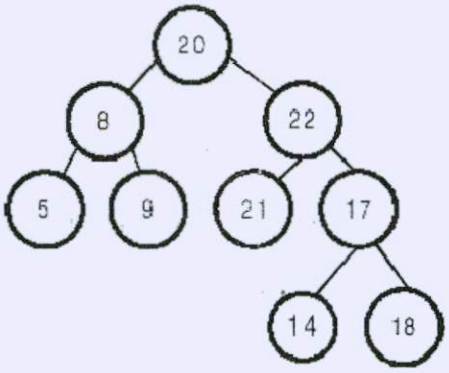


Figure 15.3(b) Not a binary search tree

We will be using BST structure to demonstrate features of Binary Trees. The operations possible on abinary tree are

- Create a Binary Tree
- Insert a node in a Binary tree
- Delete a node in a Binary Tree
- Search for a node in Binary search Tree

Algorithm: Creating Binary Tree

Step 1: Do step 2 to 3 till stopped by the user

Step 2: Obtain a new node and assign value to the node

Step 3: Insert on to a Binary Search tree

Step 4: return

Algorithm: Insertion of node into a Binary Search Tree (BST)

InsertNode (node, value)

Check if Tree is empty

if (empty then Enter the node as root

```
else    // find the proper location for insertion
        if (value < value of current node)
                If (left child is present)
                        InsertNode( LST, Value)
                ifend
        else
                allocate new node and make LST pointer point to it
        ifend
        else if (value > value of current node)
                if ( right child is present)
                        InsertNode( RST, Value);
                else
                        allocate new node and make RST pointer point to it
                ifend
        ifend
ifend
```

Figure 15.4 (a – b) illustrates an insertion operation.

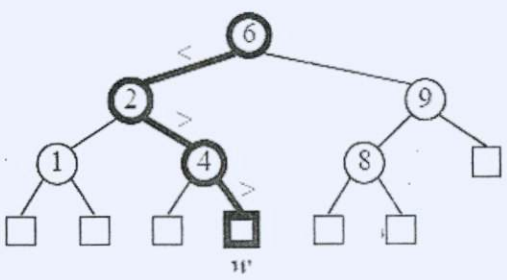


Figure 15.4(a) Before insertion

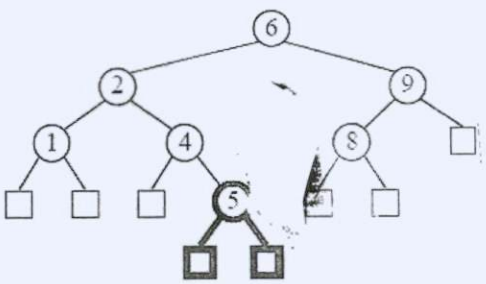


Figure 15.4(b) After insertion of item 5

Deleting a node from a Binary Search Tree

There are three distinct cases to be considered when deleting a node from a BST. They are

a)Node to be deleted is a leaf node.

Make its parent to point to NULL and free the node. Forexample, to delete node 4 the right pointer of its parent node 5 is made to point to NULL and freenode4. Figure 15.5 shows an instance of delete operation.

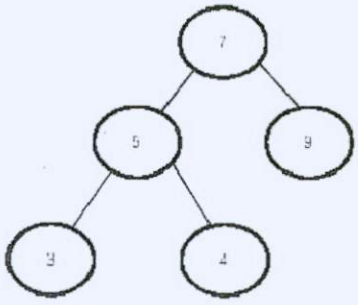


Figure 15.5 Deleting a leaf node 4

(b) Deleting a node with one child only, either left child or Right child.

For example, deletenode 9 that has only a right child as shown in Figure 15.6. The right pointer of node 7 is made to point to node 11. Thenew tree after deletion is shown in Figure 15.7.

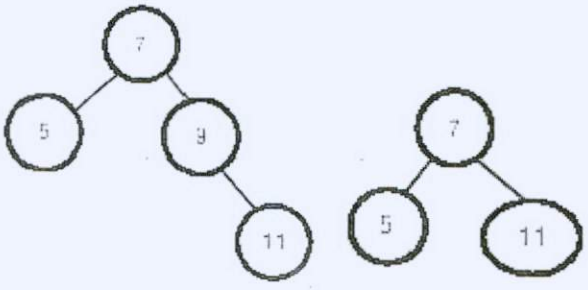


Figure 4.6Deletion of node with only one child Figure 4.7New tree after deletion

(c) Node to be deleted is an intermediate node.

- To perform operation**RemoveElement(*k*),** wesearch for key ***k***
- Assume key ***k*** is in the tree,and let ***v*** be the nodestoring ***k***
- If node ***v*** has a leaf child ***w***,we remove ***v*** and ***w*** from thetree with operation RemoveAboveExternal(***w***)
- Example: Remove 4

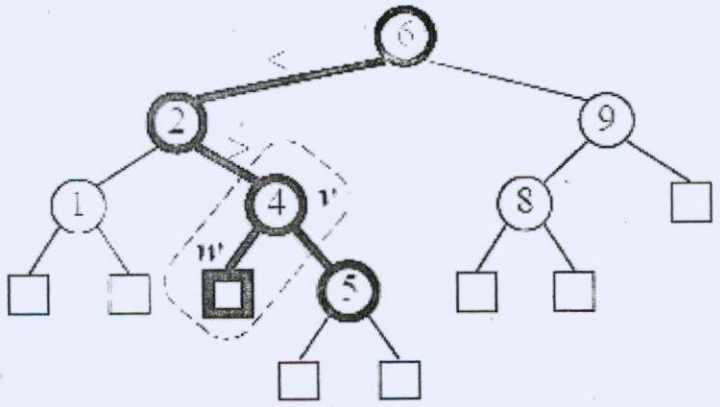


Figure 15.8 Before deletion

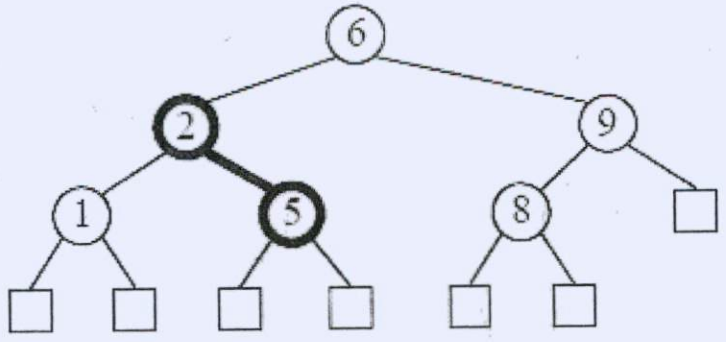


Figure 15.9 After deletion

Searching for a node in a Binary Search Tree

- To search for a key ***k***,we trace a downwardpath starting at the root

- The next node visiteddepends on theoutcome of thecomparison of *k* withthe key of the currentnode
- If we reach a leaf, thekey is not found and wereturn NO_SUCH_KEY
- Example: findElement(4)

Algorithm: findElement (k, v)

```
if T.isExternal (v)
        return NO_SUCH_KEY
if k <key(v)
        return findElement(k, T.leftChild(v))
else if k = key(v)
        return element(v)
else { k >key(v) }
        return findElement(k, T.rightChild(v))
```

Algorithm ends

Figure 15.10 shows an instance of searching for a node containing element 4.

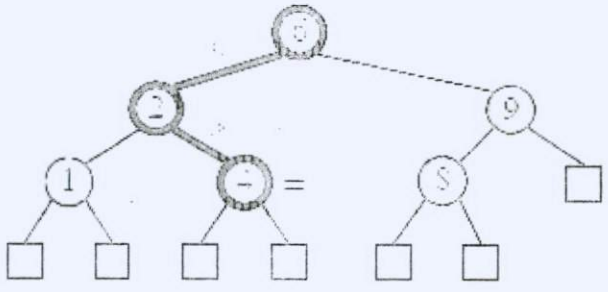


Figure 15.10 Searching operation

15.4 SUMMARY

- Traversing a tree means visiting all its nodes in some order.
- The simple traversals are preorder, inorder, and postorder.
- An inorder traversal visits nodes in order of ascending keys.
- Preorder and postorder traversals are useful for parsing algebraic expressions, among other things.
- All the common operations on a binary search tree can be carried out in O(log *n*)time.

15.5 KEYWORDS

Traversing, preorder, inorder, postorder, insertion, deletion, searching, binary search tree, left subtree, right subtree.

15.6QUESTIONS FOR SELF STUDY

1) What is meant by traversing a binary tree? Explain the various binary tree traversal techniques.
2) Describe the inorder, preorder and procedure to traverse a binary tree represented in adjacency matrix.
3) Describe the inorder, preorder and postorder procedure to traverse a binary tree represented in one dimensional array representation.
4) Explain the recursive inorder, preorder and postorder traversal of a binary tree in linked representation.
5) Describe binary search tree with an example diagram.
6) Explain the insertion, deletion and searching operations on binary search trees.
7) Design algorithms to traverse a binary tree represented using linked data structure.
8) Design algorithms to perform insertion, deletion and searching operations on binary search trees.

15.7 REFERENCES

1) Sartaj Sahni, 2000, Data structures, algorithms and applications in C++, McGraw Hill international edition.
2) Horowitz and Sahni, 1983, Fundamentals of Data structure, Galgotia publications
3) Horowitz and Sahni, 1998, Fundamentals of Computer algorithm, Galgotia publications.
4) Narsingh Deo, 1990, Graph theory with applications to engineering and computer science, Prentice hall publications.
5) Tremblay and Sorenson, 1991, An introduction to data structures with applications, McGraw Hill edition.
6) G A V Pai, Data Structures and Algorithms: Concepts, Techniques and Applications, The McGraw-Hill Companies.

Unit 16

Threaded Binary Tree and its Traversal, Representation of Forest of Trees, Traversal of Forests, Conversion of Forest to Binary Tree

STRUCTURE

16.0 OBJECTIVES

- List out the limitations of conventional binary tree
- Explain the concept of threaded binary tree
- Discuss the traversal technique for threaded binary tree
- Differentiate between binary tree and threaded binary tree
- Explain the basic concepts of tree
- Differentiate between binary tree and a general tree
- Convert a given general tree to binary tree
- Explain the concept of forest
- Traverse the given forest in preorder
- Traverse the given forest in inorder
- Traverse the given forest in postorder

16.1 LIMITATIONS OF BINARY TREE REPRESENTATION

In the last unit we have discussed about non-linear data structures. We have understood that the trees can be represented either in the form of sequential allocation or in the form of linked list allocation. Consider a tree of n nodes represented using linked allocation. As we all know that each node will be having two pointers or links i.e., left link pointing to left child and right link pointing to right child. Since we have n nodes in the tree, totally we have 2*n pointers or links in the tree. Out of 2*n nodes the linked list representation will use (n-1) links. So the number of unused links in linked representation is 2*n – (n-1) = 2*n-n+1 = n+1. The memory space of n+1 links is unnecessarily wasted. This is one of the major limitations of the binary tree.

On the other hand if we consider recursive traversal of binary tree which internally uses stack require more memory and time. Using the unused n+1 pointers we can efficiently redesign binary tree which can be used for faster traversal using efficiently the memory.

Let us illustrate the above discussion with an example. Figure 16.1 shows a sample binary tree represented using linked allocation. The three contains 11 nodes and hence we have 2*11 pointers which is equal to 22 pointers. Among 22 pointers only 10 pointers are used and remaining 12 pointers are unused as they are the pointers of leaf nodes. These unused pointers can be efficiently used to traverse to their successor or predecessor nodes using threaded binary tree which reduce the traversal time also. In the next section we explain the concept of threaded binary tree.

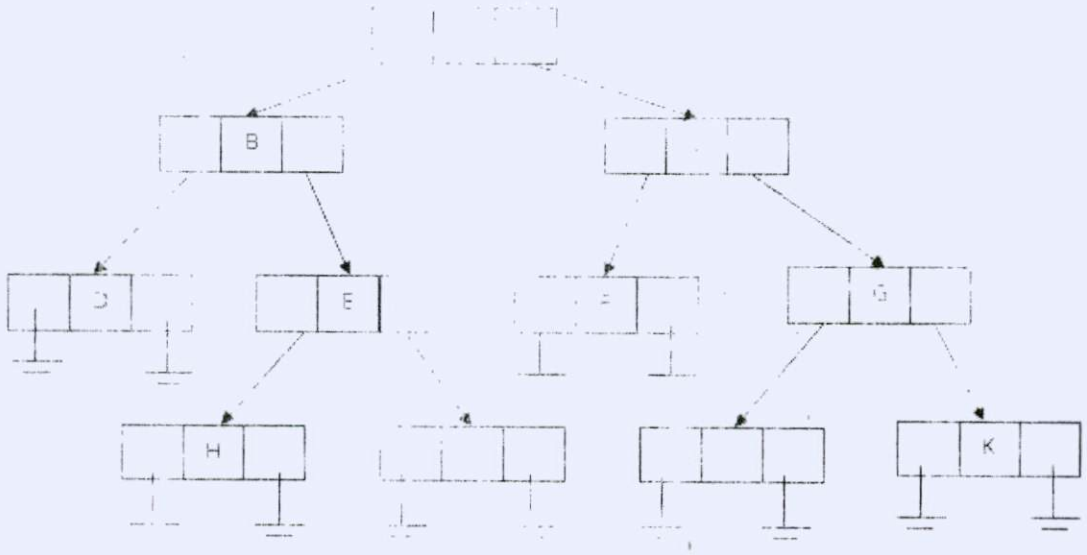


Figure 16.1 A sample binary tree with 11 nodes

16.2 THREADED BINARY TREE

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node. It should be noted that threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal. Threaded binary tree can be of three types Inorder threaded binary tree, preorder threaded binary tree and post order threaded binary tree.

16.3 INORDER THREADED BINARY TREE

As we discussed in the previous section traversal of binary tree in inorder requires stack to store the detail of successor and also we have mentioned that the pointers of the leaf nodes are wasted. Utilizing this we can redesign the binary tree such that the leaf nodes pointers can be used to point to their successor and predecessor using thread concept. To generate an inorder threaded binary tree we first generate inorder sequence of the binary tree. For the given binary tree in Figure 16.1 the inorder sequence is 'D B H E I A F C J G K'. Check for the leaf node in the binary tree, the first leaf node (from left to right) is D, its successor is B (note: predecessor and successor can be ... in inorder

sequence of the tree) and it has no predecessor. Create a right link from node D to B and create a left link to header node as shown in Figure 16.2. The new links created will be called as thread and it has shown in dotted lines.

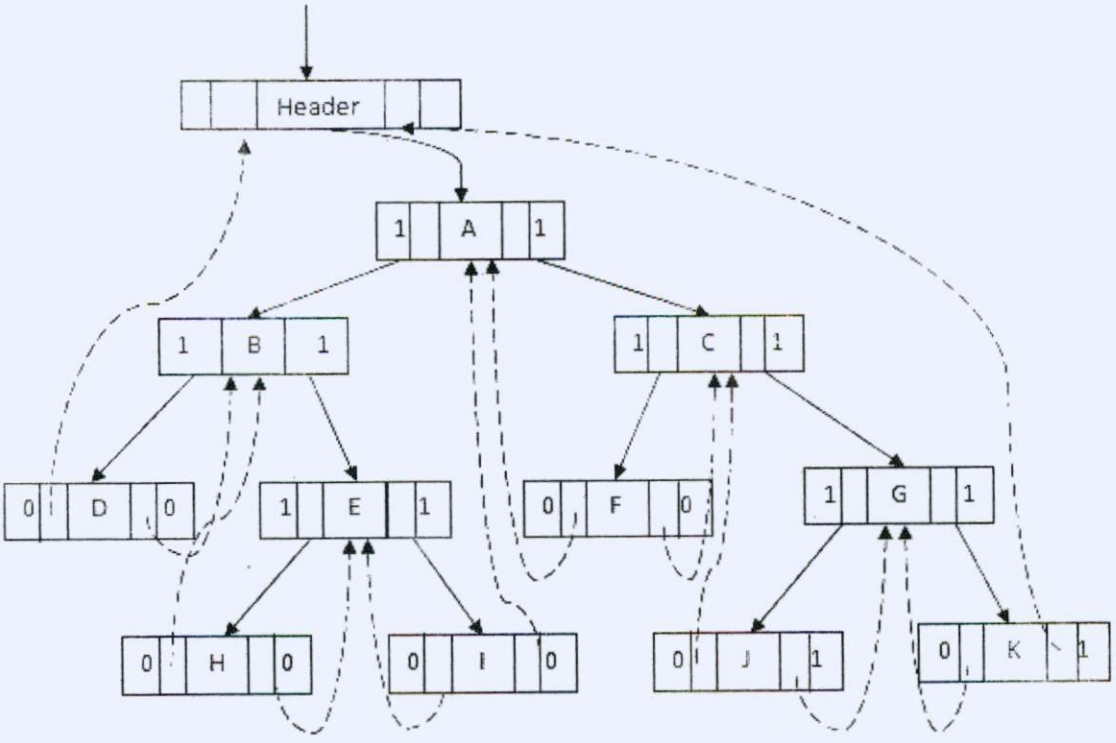


Figure 16.2: Inorder threaded binary tree

The next element in the inorder sequence is B. Since B has both left and right child it is has actual pointer which is shown in solid lines. The next node is H which is a leaf node and the predecessor and successor of H are B and E respectively. So create a left link from H to B and create a right link from H to E. The next node is E which is not a leaf node and hence it has actual pointer. The next node in the inorder sequence is I which is a leaf node. Its predecessor is E and successor is A, create a left link from I to E and right link to A. Following the same procedure create links to the remaining nodes. At the end of the process we will get an inorder threaded binary tree. Using the obtained threaded binary tree the traversal of the tree becomes faster as we can traverse without using recursive functions.

Note:

1. To differentiate between original link and the thread we add extra two data point to each node and hence each node will be having three data points and two links. The left data point is referred as LBit and the right data point is referred as RBit. Structure of a node in a threaded binary tree is given as in Figure 16.3.

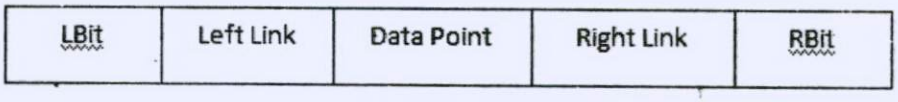


Figure 16.3: Structure of a node in a threaded binary tree

2. If the left most data point is 1, it indicates as original link and if it is 0 it indicates as a thread. Similar to right most bit also.

Threaded binary tree requires use of additional data points to indicate whether the link is actual pointer or thread. When compared to conventional binary tree threaded binary tree requires more memory, but in terms of traversal the time taken to traverse the binary tree is less compared to conventional binary tree.

16.4 INORDER THREADED BINARY TREE TRAVERSAL

The next question is how to traverse a threaded binary tree. To traverse a threaded binary tree we need to first design an algorithm to locate the successor of a given node. The algorithm to locate the successor of a given node is as follows.

Algorithm: **InorderThreadedSuccessor**

Input: bt – Address of threaded binary tree

n – a node whose successor is to be identified

Output: S, the successor of n^{th} node

Method: S = [n].Rchild

If ([n]. RBit == 1) then

While ([S]. LBit = = 1)

S = [S].LChild

While end

If end

Algorithm end

Using the above algorithm we can generate the inorder sequence of the given binary tree. Find the first node of the tree. The identified first node will be given as input to the above algorithm, and the output will be the successor of the input node. Repetitively the obtained output will be given as input to the algorithm. After the last node is given as input to the above algorithm we will get an inorder sequence of the binary tree.

The algorithm to generate inorder sequence from a inorder threaded binary tree is as follows.

Algorithm: **InorderThreadedBinaryTree**

Input: H – Address of header node

Output: Inorder sequence

Method: S = InorderThreadedSuccessor(H)

While (S ≠ H)

Disply(S)

S = InorderThreadedSuccessor(S)

While end

Algorithm end

Note:

1 The first statement in the algorithm gives the first node in the inorder sequence

2 The last executed statement gives the last node in the inorder sequence.

3 Similarly we can generate preordered and postordered threaded binary tree. In that we need to identify the predecessor of a given element.

To get the predecessor of a given node we use the same strategy used to find the successor. Instead of looking into RBit and LChild we look for LBit and RChild. In the successor algorithm, just replace RChild by LChild and RBit by LBit and LChild by RChild.

The difference between conventional binary tree and threaded binary tree is summarized as follows.

Sl. No	Binary Tree	Threaded binary tree
1	Do not have a header node	Has a header node
2	No extra bits are required to distinguish between a thread and a actual pointer	Extra bits are required to distinguish between a thread and a actual pointer
3	Traversal essentially require stack operation and hence work very slow	Traversal do not require stack operation

4	Any traversal is possible	Only one traversal is possible. If the tree is inorder threaded binary tree then inorder traversal is possible. If preorder or postorder traversal is required then the threaded binary tree must be preorder or postorder.
5	51% of link fields is wasted	Wastage is exploited to speed up traversal

16.5 INTRODUCTION TO GENERAL TREES

We have already understood the concept of graph and tree. Specifically in previous module more emphasis was given on a particular type of tree called binary tree where a node could have a maximum of two child nodes. In this unit we shall consider general type of tree where a node can have any number of child nodes. In its most abstract sense, a tree is simply a data structure capable of representing a hierarchical relationship between a parent node and an unlimited number of children nodes.

Let us recall the definition of a tree once again. A tree is a finite set of one or more nodes such that:

1. There is a specially designated node called the root.
2. Remaining nodes are partitioned into n (n>0) disjoint sets T_1, T_2, ..., T_n, where each T_i (i = 1, 2, ..., n) is a tree; T_1, T_2, ..., T_n are called sub tree of the root.

This definition implies that a tree must contain at least one node and the number of nodes is finite. A tree is always rooted in the sense that there will be a special node termed as 'root'. A tree is always directed because one can move from the root node to any other node and from any node to its successor. But the reverse is not possible. Figure 16.4 shows two general tree structures. It should be noted that the node can have more than two child nodes.

(a) (b)

Figure 16.4 Two general trees

16.6 REPRESENTATION OF GENERAL TREE

Since, in a general tree, a node may have any number of children, the implementation of a general tree is more complex than that of a binary tree. Now, we will discuss how linked list representation can be used for representing general trees.

Linked representation of trees

With this representation, we will restrict the maximum number of children for each node. Suppose, the maximum number of children allowed is m. Then the structure of node will be appeared as shown in Figure 16.5.

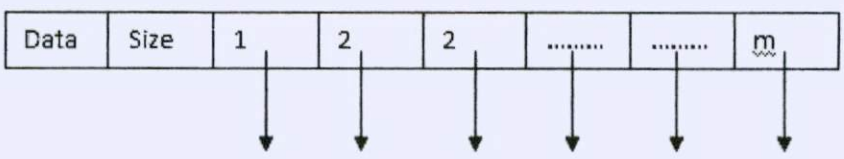


Figure 16.5 Node structure for a node in linked representation of a tree

For the Figure 16.4(a), the equivalent tree represented using linked list representation is shown in Figure 16.6. It should be noted that root node has 2 child nodes and hence the size is 2 and have 2 links and the same thing should be followed for other nodes also. The node B has 3 child nodes and hence node B has 3 pointers links which points to its child nodes D, E, and F. If we consider a leaf node it will be having zero in its size indicating that it has no child nodes. For example consider the node I which has 0 as it size. Similar can be observed in the case of node J, E, F, G, and K.

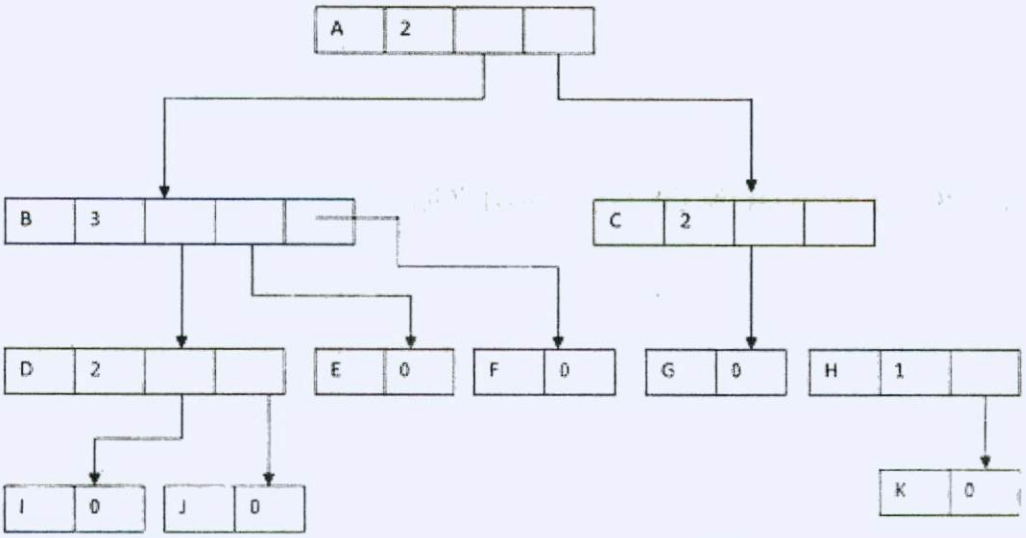


Figure 16.6 A tree represented using linked list representation

16.7 BINARY TREE REPRESENTATION OF TREES

In this section we shall now show how a tree can be represented by a binary tree. In fact, it is more practical and efficient way to represent a general tree using a binary tree, and it is also the fact that every tree can be uniquely represented by a binary tree. Hence, for the computer implementation of a tree, it is better to consider its binary tree representation.

Correspondence between a general tree (T) and its binary tree (T^1) representation are based on the following:

- All the nodes of the binary tree T^1 will be the same as the nodes of the general tree T
- The root of T^1 will be the root of T
- Parent – child relationship follows the property called left-most child next-right sibling. This means that the left child of any node N in T^1 will be the first child of the node N in the general tree T, and the right child of N in T^1 will be the next sibling of N in the general tree T^1.

Thus, given an ordered tree we can obtain its corresponding binary tree. Two steps have to be followed to do this. First, we delete all the branches originating in every node except the left-most branch. Then we draw edges from a node to the node on its immediate right, if any, which is situated at the same level. Second, for any particular node, we choose its left and right children in the following manner: the left child is the node, we choose its left and right node, and the right child is the node to the immediate right of the given node on the same horizontal line. Alternatively, this second step can be carried out by rotating all the vertical and horizontal edge so obtained in the first step roughly by 45^0 clockwise. It also can be observed that, such a binary tree will not have a right sub-tree.

As an illustration, let us consider the case of representation of tree considered in Figure 16.4(a). Figure 16.7(a) shows the equivalent binary tree of a general tree. Figure 16.4(b) shows tilted 45^0 clockwise of Figure 16.7(a).

Figure 16.7(a). Equivalent binary tree obtained after step 1

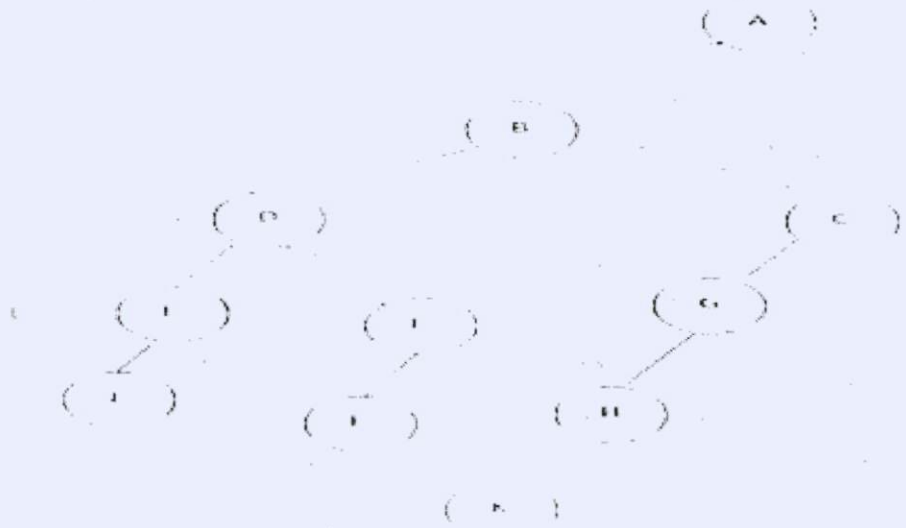


Figure 16.7(b) Equivalent binary tree obtained after step 2

16.8 REPRESENTATION OF FOREST

A forest is nothing but a collection of disjoint trees. Figure 16.8 shows a forest

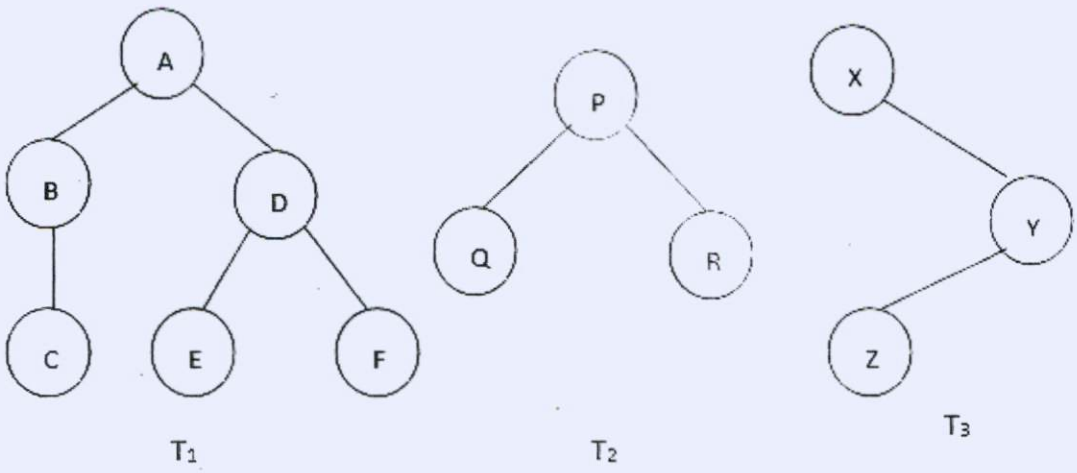


Figure 16.8 A forest F

Here, F is a forest which consists of three trees T_1, T_2, and T_3.

16.9 TRAVERSAL OF FOREST

Similar to traversal of binary tree the forest can also be traversed, the same recursive techniques that have been used to traverse binary tree cab be used to traverse a set of trees. In this section, we shall study about traversal of forest that is, given a set of trees { T_1, T_2, ... T_n} how to traverse all the tress in preorder, inorder and postorder.

Traversal of a forest depends on the way in which they are listed.

- The roots of the first tree is the list will be the first component of the forest
- All the subtrees of the first tree forms the second component of the forest
- Remaining (n-1) trees form the third component of the forest.

Throughout this unit we shall consider the forest F with three trees shown in Figure 16.9 to traverse in different orders.

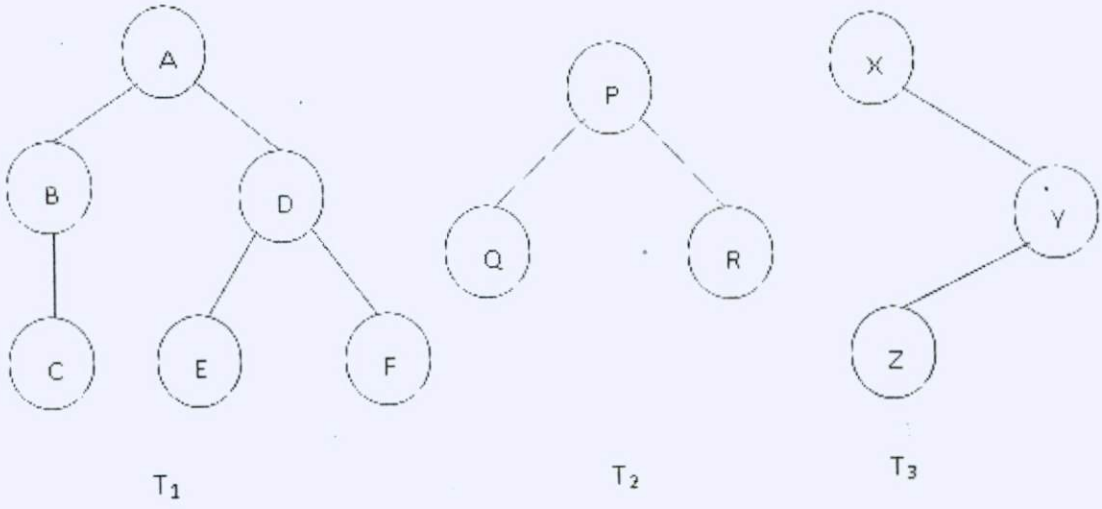


Figure 16.9 A forest F

Preorder traversal of forest

Thepreorder traversal of a forest F can be defined recursively as follows:

- Visit the root of the first tree
- Traverse though all the subtree in preorder

- Then traverse throughout the remaining tree in Preorder

The preorder sequence of the forest F is A B C D E F P R Q X Y Z

Inorder traversal of forest

Theinorder traversal of a forest F can be defined recursively as follows:

- Traverse through all subtrees of the first subtree in inorder
- Visit the root of the first
- Traverse through the remaining trees in Inorder

The inorder sequence of the forest F is C B E F D A Q R P Y Z X

Postorder traversal of forest

Thepostorder traversal of a forest F can be defined recursively as follows:

- Traverse through all subtrees of the first in postorder
- Traverse through the remaining trees in postorder
- Vist the root of the first

The postorder sequence of the forest F is C F E D B A R O P Z Y X

16.10 CONVERSION OF FOREST TO BINARY TREE

In the previous section, we have understood the concept of forest. For the sake of completion let us see the definition of the forest once again. A forest is nothing but a collection of disjoint trees as shown in Figure 16.9. Here, F is a forest which consists of three trees T_1, T_2, and T_3. In the last section we saw how to traverse forest in preorder, inorder and postorder. The best way to represent the forest in the computer is to convert the forest into its equivalent binary tree. Any forest can be converted into its equivalent binary tree in two ways. The first is using recursion and the second method is using preorder and inorder sequence of the forest.

Recursion based approach

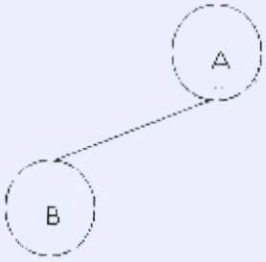
In order to represent forest in computer we have to convert it into its equivalent binary tree. In this section we shall see how to convert using recursion discussed in module 3.

The steps involved in converting any forest F into its corresponding binary tree are as follows

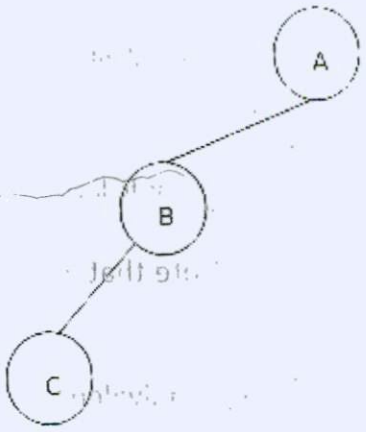
- Convert each tree T_i(i=1, 2, .., n) into its corresponding binary tree T_i^l. Note that in such that T_i^l, the right link is empty.
- Assume that the root of T_1 will be the root of binary tree equivalent of forest. Then Add T_{i+1}^l as the right sub-tree of T_i for all i = 1, 2, ..., n-1.

For illustration purpose, let us consider the transformation of forest F as shown in Figure 16.9. Step by step illustration is given below.

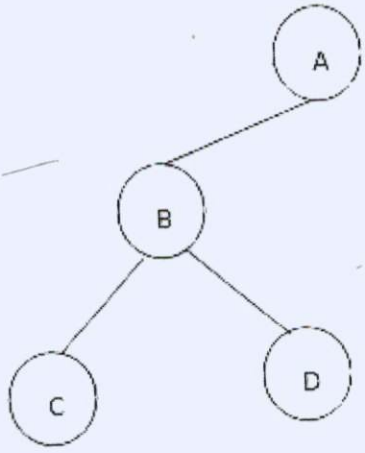
Roots of the binary tree



Since B is the first left child of A in the T_1 make it as the first left child.



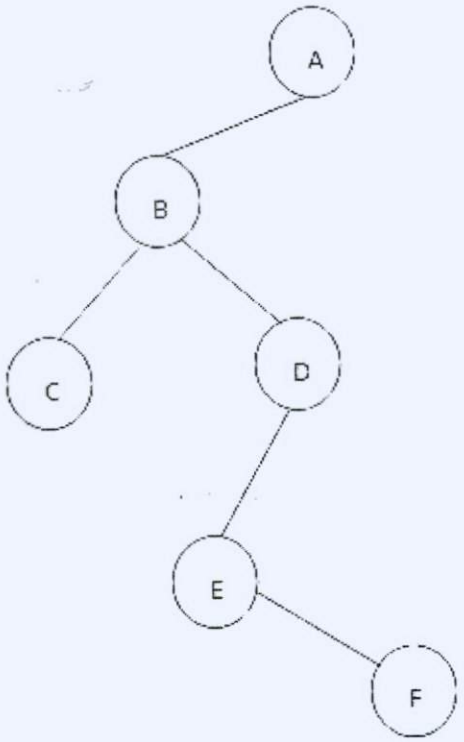
Node B has only one child so make it as left. Since node c is a leaf node in T_1, it will be leaf node in the binary tree also.



Now check for any siblings of B. Node B has only D as its siblings so make it as right child

Node D has E and F has two child nodes which are leaf node.

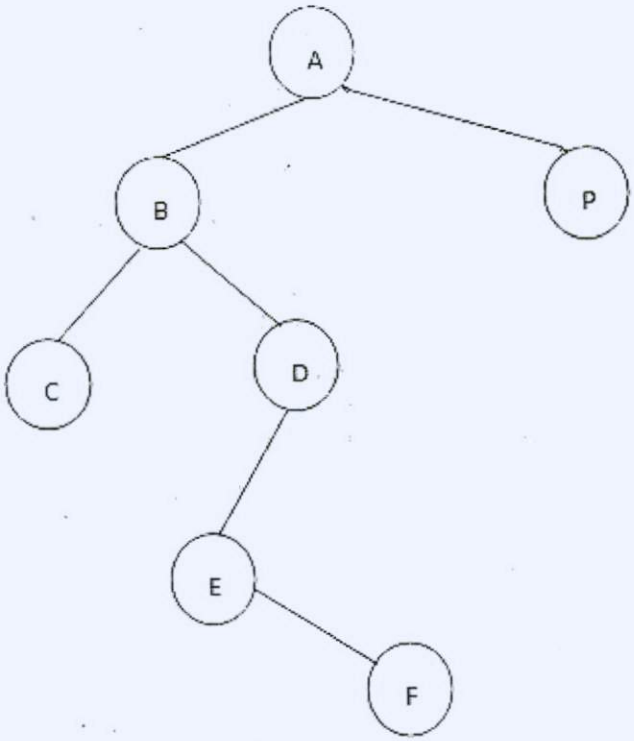
The first node is E make E as left child of D. Since D has no



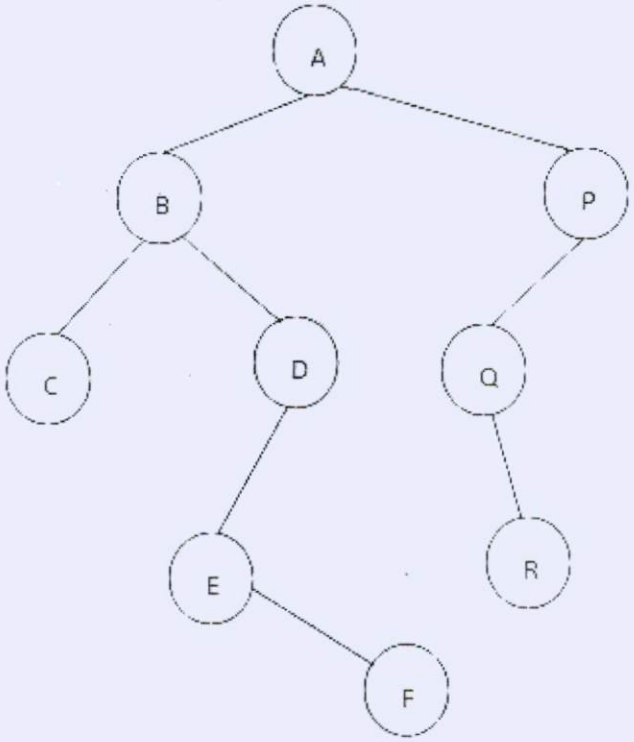
child nodes and has F as its sibling make it as right node of E.

This completes First sub tree.

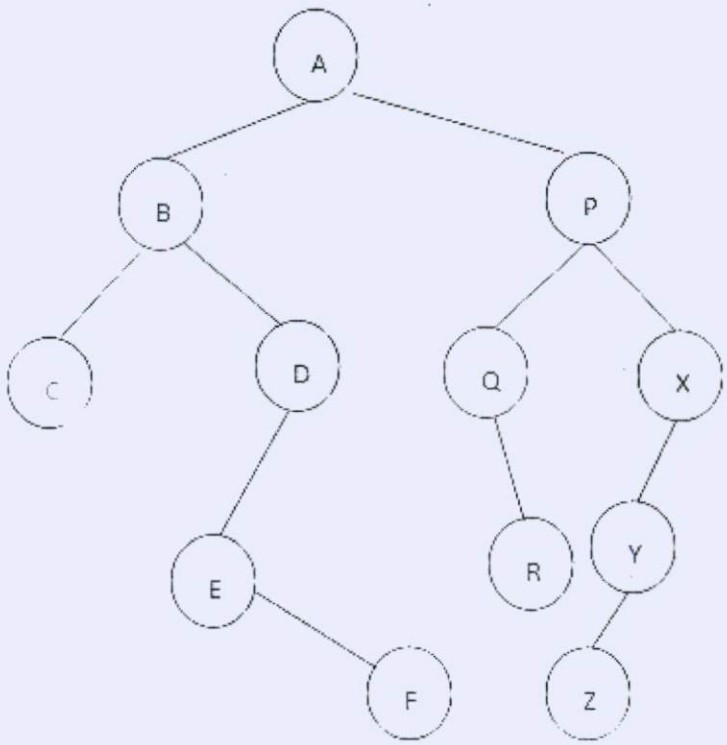
Now consider Tree T_2. The first node of T_2 is P. Make node P as right child of root node A.



Now consider Tree T_2. The first node of T_2 is P. Make node P as right child of root node A.



T_2 has only two child nodes Q and P. Both P and Q are siblings. So make Q as left child of P and make R as right child of Q. This completes Tree T_2.



Tree T_3 has X as the root node make it as the right child of P. Since X has only Y as its child make it as left child. Node Y as Z as it child so make it as left child of Y.

This complete the process and gives the binary tree

Preorder and inorder sequence based approach

Another approach to convert a given forest to its equivalent binary tree is by using the preorder and inorder sequence of the forest. In the unit 3 we saw that how to traverse and generate preorder, inorder and postorder sequence of a forest. Using preorder and Inorder sequence of a forest it is possible to generate a binary tree. The steps are as follows.

Step 1 Generate preorder and inorder sequence of the forest F.

Step 2 The first element of the preorder sequence will be the root of the first tree and this element will be treated as the root of the binary tree B which we need to construct. Let X be the first element of the preorder sequence.

Step 3 Now, consider inorder sequence, search for the element X in the inorder sequence. The elements which are at left to X will form left subtree of the binary tree B and the elements which are at right to X will form right subtree of the binary tree B.

Step 4. Now consider only left part of the inorder sequence as a new sequence and scan for the next element in the preorder sequence and make it as the left child of the root X.

Step 5 In the new sequence you search for the left child of the binary tree in the preorder sequence. The elements which are left of the second element will be the left subtree of the left child and elements which are right will form right subtree of the left child of the binary tree.

Step 6 Repeat Step 4 to Step 5 until you complete the obtained new sequence

Step 7 Now consider only right part of the inorder sequence as a new sequence and scan for the next element in the preorder sequence and make it as the right child of the root X.

Step 8 In the new sequence you search for the right child of the binary tree in the preorder sequence. The elements which are left of the second element will be the left subtree of

the right child and elements which are right will form right subtree of the right child of the binary tree.

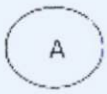
Step 9 Repeat step 4 to 8 until you process all the elements in the inorder sequence.

Let us consider the forest considered in Figure 16.9 and illustrate the steps involved in converting forest to its equivalent binary tree. The preorder and inorder sequence of the forest F in Figure 16.9 is as follows:

Preorder sequence: A B C D E F P R Q X Y Z

Inorder sequence: C B E F D A Q R P Y Z X

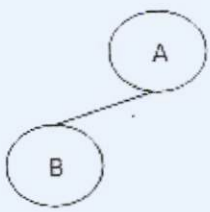
The first element of the preorder sequence is A, make this as a root of the binary tree



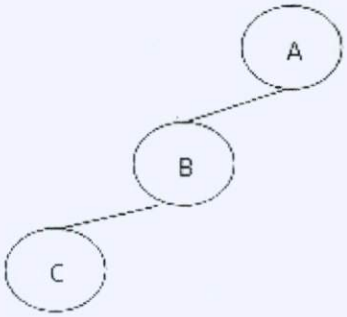
Search for the element A in the preorder sequence. The elements which are left to element A will form left subtree and elements which are right of the binary tree will form the right subtree of the binary tree.

C	B	E	F	D	A	Q	R	P	Y	Z	X
Left subtree						Right subtree					

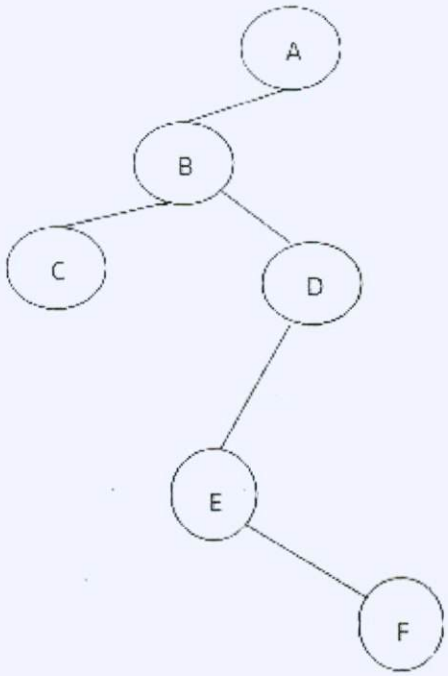
The next element in the preorder sequence is B, make it as the root of the left sub tree



Consider C B E F D as a new sequence. Search for B in this sequence The elements which are left to B in new sequence will form left subtree and elements which are right to element B forms right subtree of element B. In the new sequence only one element 'c' is present this will form the left child of B. Scan for the next element in the preorder sequence, the next element is C. Since C do not have any unprocessed elements to its left and right it becomes the leaf node of the binary tree.



Scan for the next element in the preorder sequence and it is D. Since D is right of B make it as the right child of node B. Element D has E and F to its left nad hence they becomes the left child nodes. Since D do not have any elements to its right which are processed the node D will not have right subtree.



Similarly, do the process to the elements which are right to element A and obtain the right subtree.

16.11SUMMARY

In this unit we have studied the limitations of conventional binary tree. To overcome those limitations, we have modified binary tree called threaded binary tree. In this unit we have also designed the algorithm to traverse the threaded binary tree in inorder sequence. The difference between conventional binary tree and threaded binary tree is also brought clearly.

The concept of general tree, their representation and how to convert a given general tree to its equivalent binary tree is also presented in this unit. A brief introduction about forest and different ways of traversing forest is also presented. Similar to binary tree traversal we can traverse forest in preorder, inorder as well as in postorder.

We have studied a recursive and sequence based approach to convert a given forest into its equivalent binary tree. We have also studied how to design algorithms for converting forest to a binary tree.

16.11 KEYWORDS

Threaded binary tree, inorder, preorder, postorder threaded binary tree traversal, general tree, linked list representation of tree, forest.

16.12 QUESTIONS FOR SELF STUDY

1) Design an algorithm to find the predecessor node in an inorder threaded binary tree.
2) Suggest an algorithm to construct preorder threaded binary tree.
3) Suggest an algorithm to construct postorder threaded binary tree.
4) What is the difference between binary tree and a general tree?
5) With a neat diagram explain the linked list representation of general tree.
6) Consider a general tree of your own choice and convert it into its equivalent binary tree.

7) What is forest? Explain with an example.

8) Discuss the preorder, inorder and postorder traversal of a forest.

9) Design an algorithm to convert a given forest into its equivalent binary tree using recursive approach. Consider a forest with atleast three tress and 15 nodes of your chose and trace out the algorithm

10) Design an algorithm to convert a given forest into its equivalent binary tree using preorder and inorder sequence of the forest. Consider a forest with atleast three tress and 15 nodes of your chose and trace out the algorithm.

16.13 REFERENCES

1) Ellis Horowitz, Sartaj Sahni, and Dinesh Mehta. Fundamental of Data Structures in C++

2) Alfred V. Aho , Jeffrey D. Ullman, John E. Hopcroft. Data Structures and Algorithms. Addison Wesley (January 11, 1983)

3) Sartaj Sahni, 2000, Data structures, algorithms and applications in C++, McGraw Hill international edition.

4) Horowitz and Sahni, 1983, Fundamentals of Data structure, Galgotia publications

5) Horowitz and Sahni, 1998, Fundamentals of Computer algorithm, Galgotia publications.

6) Narsingh Deo, 1990, Graph theory with applications to engineering and computer science, Prentice hall publications.

7) Tremblay and Sorenson, 1991, An introduction to data structures with applications, McGraw Hill edition.

8) G A V Pai, Data Structures and Algorithms: Concepts, Techniques and Applications, The McGraw-Hill Companies.

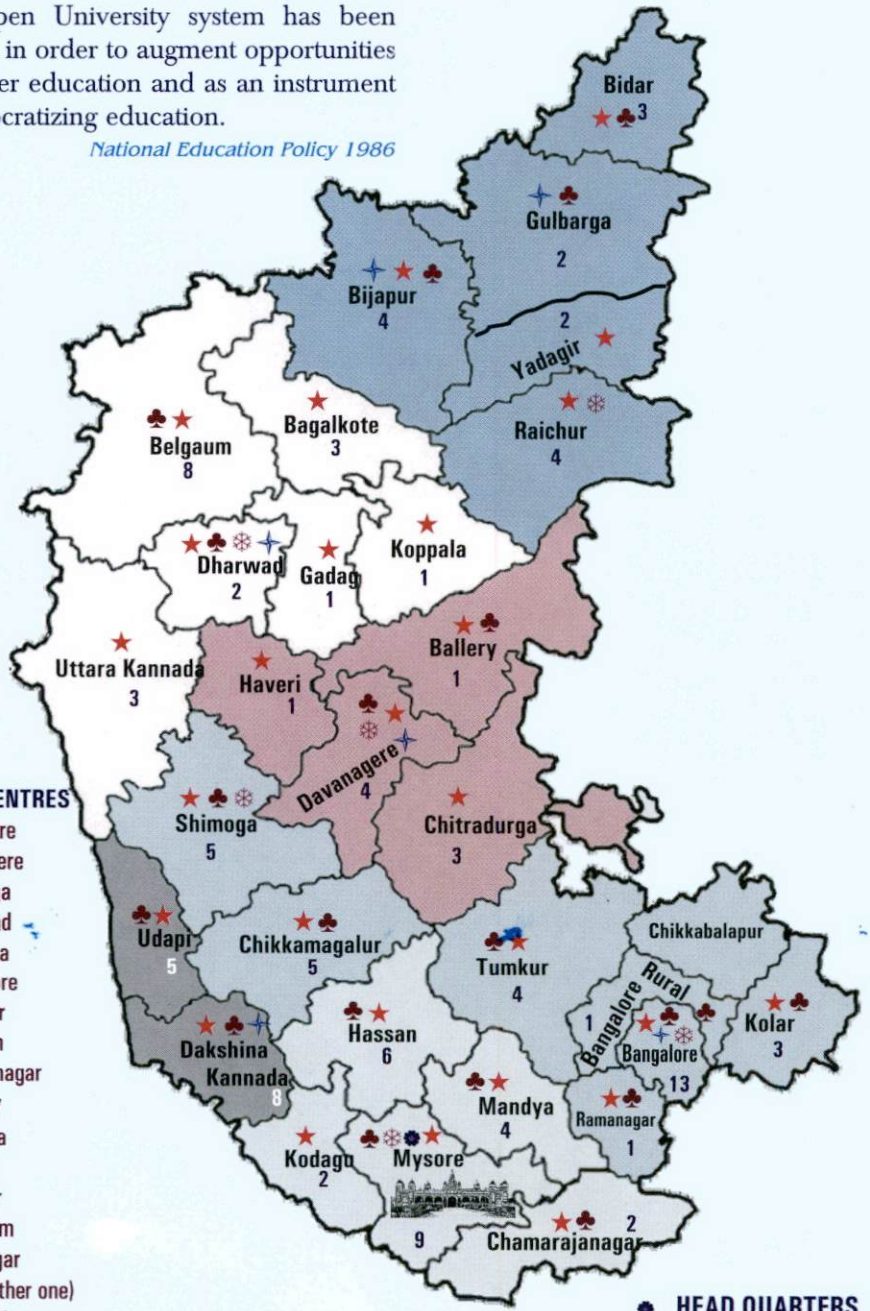
ಆದೇಶ ಸಂಖ್ಯೆ : ಕರಾಮುವಿ/ಅಸಾವಿ/2-1614/2012-13 ದಿನಾಂಕ : 02-08-2012
ಒಳಪುಟ : 60 GSM ಮಾಪ್ಲಿತೊ ಮತ್ತು ರಕ್ಷಾ ಪುಟ : 220 GSM 'ಆರ್ಟ್ ಕಾರ್ಡ್'
ಮುದ್ರಕರು : ಶ್ರೀ ಸುಬ್ರಮಣ್ಯೇಶ್ವರ ಬುಕ್ ಡಿಪೋ, ಬೆಂಗಳೂರು - 560 002, ಪ್ರತಿಗಳು - 200

Karnataka State Open University

Manasagangotri Mysore - 570 006

The Open University system has been initiated in order to augment opportunities for higher education and as an instrument of democratizing education.

National Education Policy 1986



REGIONAL CENTRES

Bangalore
Davanagere
Gulbarga
Dharwad
Shimoga
Mangalore
Tumkur
Hassan
Chamarajanagar
Bellary
Mandya
Kolar
Bijapur
Belagaum
Ramanagar
Bangalore (another one)
Chikmagalur
Udupi
Karwar
Bidar
Mysore

HEAD QUARTERS

Total Study Centres : 123

Regional Centres : 21

B.Ed Study Centres : 10

M.Ed Study Centres : 06

KSOU

Higher Education to everyone everywhere

ಉನ್ನತ ಶಿಕ್ಷಣ ಎಲ್ಲರಿಗೂ ಎಲ್ಲೆಡೆ

ಕರ್ನಾಟಕ ರಾಜ್ಯ ಮುಕ್ತ ವಿಶ್ವವಿದ್ಯಾನಿಲಯ

ಮಾನಸಗಂಗೋತ್ರಿ, ಮೈಸೂರು – 570 006

Karnataka State Open University

Manasagangotri, Mysore - 570 006 Website : www.ksoumysore.edu.in