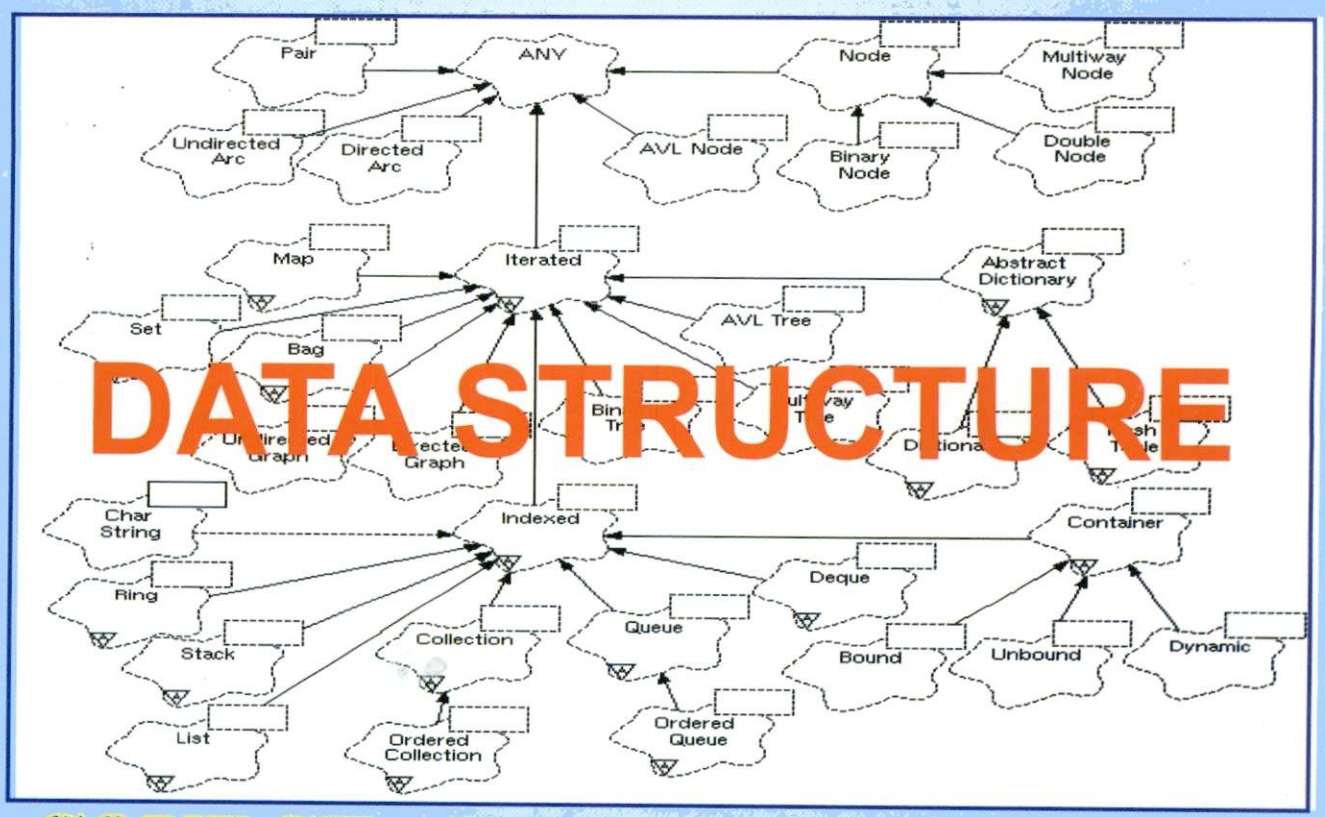
ಕರ್ನಾಟಕ ರಾಜ್ಯ ಮುಕ್ತ ವಿಶ್ವವಿದ್ಯಾನಿಲಯ
ಮಾನಸಗಂಗೋತ್ರಿ, ಮೈಸೂರು – ೫೭೦ ೦೦೬

KARNATAKA STATE OPEN UNIVERSITY
Manasagangotri, Mysore - 570 006

M.Sc. Computer Science

First Semester



DATA STRUCTURE

COURSE: 4

MODULE : 1 - 6

MSCS-504 DATA STRUCTURE

ಉನ್ನತ ಶಿಕ್ಷಣಕ್ಕಾಗಿ ಇರುವ ಅವಕಾಶಗಳನ್ನು ಹೆಚ್ಚಿಸುವುದಕ್ಕೆ ಮತ್ತು ಶಿಕ್ಷಣವನ್ನು ಪ್ರಜಾತಂತ್ರೀಕರಿಸುವುದಕ್ಕೆ ಮುಕ್ತ ವಿಶ್ವವಿದ್ಯಾನಿಲಯ ವ್ಯವಸ್ಥೆಯನ್ನು ಆರಂಭಿಸಲಾಗಿದೆ.

ರಾಷ್ಟ್ರೀಯ ಶಿಕ್ಷಣ ನೀತಿ 1986

The Open University System has been initiated in order to augment opportunities for higher education and as instrument of democrating education.

National Educational Policy 1986

ವಿಶ್ವ ಮಾನವ ಸಂದೇಶ

ಪ್ರತಿಯೊಂದು ಮಗುವು ಹುಟ್ಟುತ್ತಲೇ – ವಿಶ್ವಮಾನವ, ಬೆಳೆಯುತ್ತಾ ನಾವು ಅದನ್ನು 'ಅಲ್ಪ ಮಾನವ'ನನ್ನಾಗಿ ಮಾಡುತ್ತೇವೆ. ಮತ್ತೆ ಅದನ್ನು 'ವಿಶ್ವಮಾನವ'ನನ್ನಾಗಿ ಮಾಡುವುದೇ ವಿದ್ಯೆಯ ಕರ್ತವ್ಯವಾಗಬೇಕು.

ಮನುಜ ಮತ, ವಿಶ್ವ ಪಥ, ಸರ್ವೋದಯ, ಸಮನ್ವಯ, ಪೂರ್ಣದೃಷ್ಟಿ ಈ ಪಂಚಮಂತ್ರ ಇನ್ನು ಮುಂದಿನ ದೃಷ್ಟಿಯಾಗಬೇಕಾಗಿದೆ. ಅಂದರೆ, ನಮಗೆ ಇನ್ನು ಬೇಕಾದುದು ಆ ಮತ ಈ ಮತ ಅಲ್ಲ; ಮನುಜ ಮತ. ಆ ಪಥ ಈ ಪಥ ಅಲ್ಲ ; ವಿಶ್ವ ಪಥ. ಆ ಒಬ್ಬರ ಉದಯ ಮಾತ್ರವಲ್ಲ; ಸರ್ವರ ಸರ್ವಸ್ತರದ ಉದಯ. ಪರಸ್ಪರ ವಿಮುಖವಾಗಿ ಸಿಡಿದು ಹೋಗುವುದಲ್ಲ; ಸಮನ್ವಯಗೊಳ್ಳುವುದು. ಸಂಕುಚಿತ ಮತದ ಆಂಶಿಕ ದೃಷ್ಟಿ ಅಲ್ಲ; ಭೌತಿಕ ಪಾರಮಾರ್ಥಿಕ ಎಂಬ ಭಿನ್ನದೃಷ್ಟಿ ಅಲ್ಲ; ಎಲ್ಲವನ್ನು ಭಗವದ್ ದೃಷ್ಟಿಯಿಂದ ಕಾಣುವ ಪೂರ್ಣದೃಷ್ಟಿ.

ಕುವೆಂಪು

Gospel of Universal Man

Every Child, at birth, is the universal man. But, as it grows, we trun it into "a petty man". It should be the function of education to turn it again into the enlightened "universal man".

The Religion of Humanity, the Universal Path, the Welfare of All, Reconciliation, the Integral Vision - these ***five mantras*** should become View of the Future. In other words, what we want henceforth is not this religion or that religion, but the Religion of Humanity; not this path or that path, but the Universal Path; not the well-being of this individual or that individual, but the Welfare of All; not turning away and breaking off from one another, but reconciling and uniting in concord and harmony; and above all, not the partial view of a narrow creed, not the dual outlook of the material and the spiritual, but the Integral Vision of seeing all things with the eye of the Divine.

Kuvempu

Karnataka State Open University

Manasagangothri, Mysore – 570 006

First Semester M.Sc in Computer Science

Module 1 — Data Structure Classification

Module 2 — Linear data structure

Module 5 Threaded Binary Tree and Forest

Module 6 Sorting and Searching

Publisher

Registrar
Karnataka State Open University
Manasagangotri, Mysore - 6.

Developed by Academic Section, KSOU, Mysore
Karnataka State Open University, 2010

Further information on the Karnataka State Open University Programmes may obtained from the University's office at Manasagangotri, Mysore-6

Printed and Published on behalf of Karnataka State Open University. Mysore-6 by

Registrar (Administration)

Preface

Design of efficient algorithms is very much essential to solve problems efficiently using computers. Selection of appropriate data structures is of 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 prepared to give an overview of the fundamentals of data structures and their importance in solving problems. The concept of various data structures and their applications are dealt independently without considering any programming language of interest. The whole material is organized into six modules each with four units. Each unit lists the objectives of study along with the relevant questions and suggested reading to better understand the concepts.

Module-1 introduces the problem solving technique and the development of efficient algorithms for solving problems. It also gives an overview of the classification of data structures, types of primitive data structures and their representation followed by an introduction to non-primitive linear data structure called array and their representation.

Module-2 introduces non-primitive linear data structures called stacks and queues. This module presents the implementation of stacks and queues using foundational data structure (array). Implementation of circular queue, priority queue and double ended queues are also discussed. Applications of stacks and queues, the concept of recursion and its applications are also demonstrated.

Module-3 introduces a linked representation of linear data structures, their advantages over sequential representation, types of linked lists, implementation of linked lists and some general operations on linked lists.

Module-4 introduces graph and tree data structures with basic terminologies. Representation of graphs and trees based on sequential and linked allocation methods are explained. The module also covers the binary tree, traversal of binary tree and operations on binary tree.

Module-5 highlights the limitations of conventional binary tree and introduces the threaded binary tree. Traversal of threaded binary tree and the difference between binary tree and threaded binary tree is discussed. The module also introduces general tree and forest data

structures, their representation and traversal techniques. Conversion of forest into tree is also discussed in this module.

Module-6 introduces various sorting techniques covering conventional sort, selection sort, insertion sort, binary search based insertion sort, merge sort, quick sort, heap sort and bucket sort. This module also introduces various searching techniques such as linear search, binary Search, depth first search and breadth first search.

We thank everyone who helped directly or indirectly to prepare this material. Without their support this material would not have been a reality.

Dr.D.S.Guru

Dr.H.S.Nagendraswamy

Mr.Manjunath

UNIT -1

INTRODUCTION

Structure

1.0 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 out the properties of an algorithm

1.1 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.

1.2 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 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.

1.3 PROPERTIES OF AN 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.

1.4 SUMMARY

In this unit, we have studied the method of analyzing and solving any given problem. The relationship between algorithm and data structure is given in detail. Considering the problem of prime number as a case study we have demonstrated how to design a suitable algorithm to solve a particular problem. Also, we have learnt the properties that any designed algorithm should posses.

1.5 KEY WORDS

(i) Problem solving

(ii) Algorithm design

(iii)Algorithm Ananlysis

(iv) Finiteness of an algorithm

(v) Definiteness of an algorithm

(vi) Effectiveness of an algorithm

1.6 QUESTIONS

(1). What is an algorithm? Explain with example.

(2). Mention the properties of an algorithm. Explain atleast two in detail.

(3). With a suitable example discuss the relationship between algorithm and data structure.

1.7 EXERCISE

(1). Design and develop an algorithm to find largest of 3 numbers.

(2). Suggest atleast two different types of algorithm to check whether the given number if prime or not. Comment on both algorithms and identity which one is best and justify your answer.

(3). Discuss on steps involved in solving the problem of sorting n numbers.

1.8 REFERENCE

(1). Ellis Horowitz and Sartaj Sahni. Fundamentals of Data Structures. W H Freeman & 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 and Algorithms. Addison Wesley (January 11, 1983

UNIT - 2

DATA STRUCTURE: CLASSIFICATION

Structure

2.0 OBJECTIVES

After reading this unit you are able to

- List out the different types of data structures
- Differentiate between abstract data type and data object
- Elucidate the components of data structures.

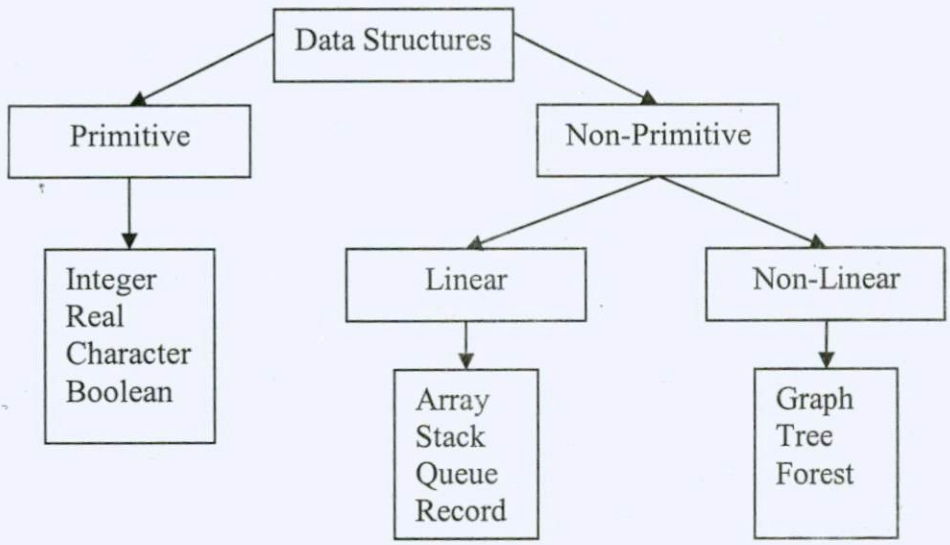
2.1 CLASSIFICATION OF DATA STRUCTURES

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 1.1.



2.2 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 1.2: 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 a designer to plan on the implementation of the data objects and its supportive operations in any programming language.

2.3 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.

2.4 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

2.5 EXERCISE

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

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

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

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

2.6 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 and Algorithms. Addison Wesley (January 11, 1983)

UNIT- 3

PRIMITIVE DATA STRUCTURE

Structure

3.0 OBJECTIVES

After reading this unit you are able to

- Discuss the different ways of representing integer type of data
- Implement the basic operations on integer type of data
- Explain the different ways of representing real number
- Implement the basic operation on real number
- Represent character data types in computers
- Represent boolean type of data in computers

3.1 PRIMITIVE DATA TYPES

Primitive data types are one which can be operated upon by machine level instructions. These are indivisible units treated as atomic values in the storage structure. The following sections describe the various primitive data types and their representations supported by most of the programming languages.

3.1.1 Integer Numbers

It is a whole number with or without a sign. Depending on the nature of the data, sometimes an integer must be stored with a positive or negative sign, such as +10 or –5. Absence of a sign generally indicates a +ve sign. An integer that is stored with a sign is called a *signed number;* an integer that is not stored with a sign is called an *unsigned number*.

More formally, an integer number I is denoted as

$$I = Sd_{n-1}d_{n-2} \ldots d_1d_0$$

Where **ds** are the digits **{0, 1, 2, 3, ... , R-1}** supported by the number system with base (radix) R. Associated with I, there is value which computed as,

$$\text{Value (I)} = S \sum d_k * R^k \text{ for } k = 0 \text{ to } n$$

The integer data objects defined over a domain with a set of functions and axioms can be treated as a primitive data structure as described below.

D: Domain: {-n, n-1, n-2, …, -2, -1, 0, 1, 2, …, n-2, n-1, n}

F: Functions: Iszero(n), Iseven(n), Isodd(n), Isnegative(n), Ispositive(n), Add(n1, n2)

Sub(n1, n2), Mul(n1, n2), Mod(n1, n2) etc.

A: Axioms: Add(Sub(n1,n2),n2) = n1

Sub(Add(n1, n2), n2) = n1

Precessor(Successor(n)) = n

Thus I: (D, F, A) is an abstract data type representing integer as a data structure.

3.1.1.1 Representation of Integers

In the previous section, we learnt about integer as data structure. Now, we shall explain the representation of integers. A storage structure consisting of a number of memory cells to hold an integer number is referred to as integer representation. An integer can be a signed or an unsigned integer. Representation of an unsigned integer is simple and straight forward and a signed integer number can be represented in memory in three different ways as described in the subsequent paragraphs. The figure 3.1 shows the overview of integer representation schemes.

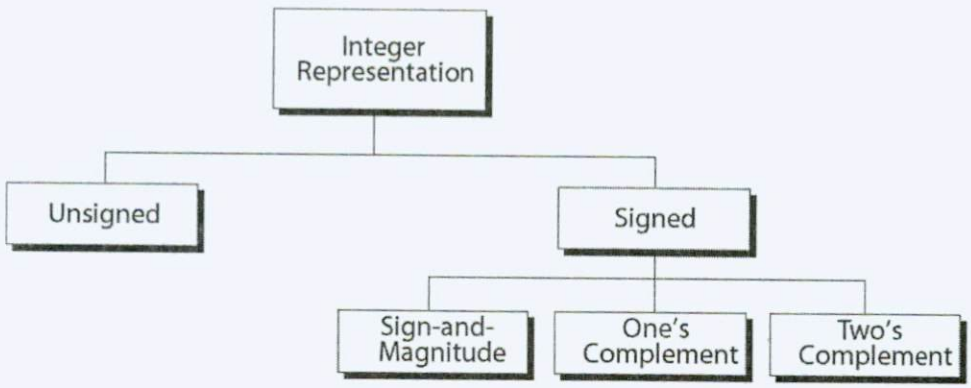


Fig 3.1 Integer representation schemes

Unsigned Integer Representation

In this representation scheme, if n numbers of locations are used to store an integer number then all the cells are used to represent an integer. Therefore, the range of numbers that can be represented using n storage locations is 0 to 2^n-1. For example,

Example 3.1:

Number of Bits	Range
8	0 to 255
16	0 to 65535

Decimal	8-bit allocation	16-bit allocation
7	00000111	0000000000000111
234	11101010	0000000011101010
258	overflow	0000000100000010

24,760	overflow	0110000010111000
1,245,678	overflow	overflow

Signed Magnitude (SM) Representation

In this representation scheme, if n numbers of cells are used to store an integer number, then one cell in the most significant bit position is reserved to store the sign information and the remaining (n-1) number of cells are used to store the magnitude. The cell in the most significant bit position, when represented in binary number system, will contain digit 1 for negative numbers and 0 for positive numbers. The following diagram illustrates the representation of integer number in signed magnitude representation.

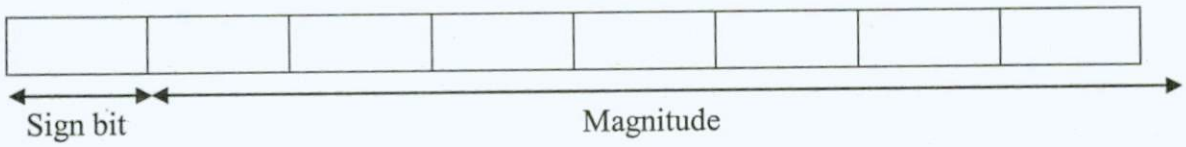


Fig 3.2 storage structure for integer in Sign Magnitude representation

In the above example, 8 cells (bits) are used to represent an integer. Out of 8 cells, one cell is reserved to store the sign and the remaining 7 cells are used for magnitude. The following examples illustrate the range and total size of the permitted domain for various values of R and n.

Example 3.2: For R=10 and n=3

Range: {-99, -98, …, -1, 0, 1, …, 98, 99}

Size: (99 – (-99) +1) = 199

Example 3.3: For R=2 and n=3

Range: {-3, -2,-1, 0, 1, 2, 3}

Size: (3 – (-3) +1) = 7

In general, if there are n cells with R being the radix, then

Range = { - ($R^{n-1} - 1$) to ($R^{n-1} - 1$) }

Expected size = R^n

But, practically, size is $(R^{n-1} - 1) - (-R^{n-1} + 1) + 1 = 2R^{n-1} - 1$. One more observation from the above representation scheme is that for $n = 3$, the actual size in any radix, in general, is R^n. Therefore, there is a loss of data objects in representation which is given by,

$$\text{Loss} = R^n - (2R^{n-1} - 1) = (R - 2)R^{n-1} + 1.$$

This suggests that it is better to work in lower radix (R) with less number of locations (n).

From the above discussion, it is very much clear that the signed magnitude representation has the following drawbacks.

1. Dual representation of zero (Syntactically different but semantically same).
2. Loss is $(R - 2)R^{n-1} + 1$, which says that work in lower radix (R) and lower size (n).
3. Fuzziness in deciding a suitable radix (R) and number of bits (n).
4. Requires special circuit called sign checker and both adder as well as subtractor circuits for realization for arithmetic operators.

The following table gives the range of numbers under varying number of bits used for representation.

Number of bits	Range of Numbers			
8	-127	-0	+0	+127
16	-32767	-0	+0	+32767
32	-2,147,483,647	-0	+0	+2,147,483,647

Note: (i) In sign-and-magnitude representation, the leftmost bit defines the sign of the number. If it is 0, the number is positive. If it is 1, the number is negative

(ii) There are two 0s in sign-and-magnitude representation: positive and negative.

In an 8-bit allocation: +0 → 00000000
-0 → 10000000

Diminished Radix Complement (DRC) Representation

In order to overcome the limitations of the signed magnitude representation scheme, diminished radix complement representation scheme is used. In this representation scheme, a positive integer number (I) is represented as it is but a negative integer number (-I) is represented as $(R^n - 1) - I$.

Let us consider an example to understand this representation scheme. For R = 10 and n = 3, the range of numbers we can represent is -499 to 499. Since the negative

numbers are represented as $(R^n - 1) - I$, the numbers -499, -498, -497, …, -1, -0 are converted using the above formula and represented in DRC scheme as shown below.

-0	-1	-2	•••	-99	•••	-497	-498	-499
999	998	997	•••	900	•••	502	501	500

Therefore, the range of numbers that can be represented in general, in this scheme and the size are given as follows:

Range: $\{ -(R^n/2 - 1) \ldots (R^n/2 - 1) \}$

Size: $R^n/2 - 1 + R^n/2 - 1 + 1 = R^n/2 + R^n/2 - 1 = R^n - 1$.

For R = 10 and $n = 3$, the actual size is 1000, but the size in this representation is 999. Thus the loss is 1. This loss is because of the dual representation of 0 as observed in signed magnitude representation scheme.

Similarly, for R = 2 and $n = 3$, the range of numbers we can represent is -3 to 3 and the size is 7. The following table shows the values in decimal as well as in binary represented using this scheme.

-3	-2	-1	-0	0	1	2	3
100	101	110	111	000	001	010	011

From the above illustrations, it is understood that the actual size is 8, but the size due to this representation scheme is 7 and hence the loss is 1. This can be generalized as shown below.

The actual size is R^n

The size due to this representation scheme is $R^n - 1$

Therefore, the loss is $(R^n - (R^n - 1)) = 1$ (constant). This indicates that the loss in number of data objects to be represented is always 1, and it is independent of R and n. hence, we can work with higher radix R and more number of locations n.

This suggests that better to work in higher radix (R) and in high dimension (n).

From the above discussions, we can draw the following conclusions for this representation scheme:

1. Dual representation of zero (0).
2. Loss is constant. That is 1.
3. Better to work with higher radix (R) with more number of locations (n).
4. Requires a special circuit for complementation; carry checker and only an adder.

The following table gives the range of numbers under varying number of bits used for representation.

Number of bits	The Range of 1's complement Integers			
8	-127	-0	+0	+127
16	-32767	-0	+0	+32767
32	-2,147,483,647	-0	+0	+2,147,483,647

Note: (i) For R=2, DRC is called as 1's complement representation

(ii) For R=10, DRC is called as 9's complement representation.

(iii) In one's complement representation, the leftmost bit defines the sign of the number. If it is 0, the number is positive. If it is 1, the number is negative.

(iv) One's complement means reversing all bits. If you one's complement a positive number, you get the corresponding negative number. If you one's complement a negative number, you get the corresponding positive number. If you one's complement a number twice, you get the original number.

Radix Complement Representation (RC)

In order to overcome the limitations of signed magnitude representation scheme as well as diminished radix complement representation scheme, a radix complement representation scheme is used. In this representation scheme, a positive integer number (I) is represented as it is but a negative integer number (-I) is represented as $(R^n - I.)$.

Let us consider an example to understand the above representation scheme. For R = 10 and $n = 3$, the range of numbers we can represent is -500 to 499. Since the negative

numbers are represented as (R^n – I), the numbers -500, -499, -498, -497, …, -1, -0 are converted using the above formula and represented in RC scheme as shown below.

-0	-1	-2	…	-99	…	-497	-498	-499	-500
000	999	998	…	901	…	503	502	501	500

If we apply the formula (R^n – I) to represent the negative numbers, the negative number 0 result with 000 as shown below.

For R=10, n=3 and I = negative 0,

(R^n – I) = (1000 – 0) = 1000. Because n=3, this value reduces to 000. Thus, both negative 0 as well as positive 0 map to the same 000 notation in the storage. Hence, the dual representation of zero is avoided in this scheme, which results with no loss of size in the storage. In general, the range and size for some R and n in this representation scheme is calculated as follows:

Range: $\{ - R^n/2 , \ldots , R^n/2 - 1\}$

Size: $R^n/2 - 1 - (- R^n/2) + 1 = R^n/2 - 1 + R^n/2 + 1 = R^n$

Therefore, Loss = Actual size – Size due to DRC scheme = (R^n – R^n) = 0

From the above discussions, we can draw the following conclusions for this representation scheme:

1. No dual representation of zero (0).
2. No Loss in size (storage).
3. Requires a special circuit for complementation; and only an adder.

The following table gives the range of numbers under varying number of bits for representation.

Number of bits	The Range of 2's complement Integers		
8	-128	0	+127
16	-32768	0	+32767
32	-2,147,483,648	0	+2,147,483,647

Note:

- For R=2, DRC is called as 2's complement representation
- For R=10, DRC is called as 10's complement representation.
- In two's complement representation, the leftmost bit defines the sign of the number. If it is 0, the number is positive. If it is 1, the number is negative.
- No dual representation for zero.
- In an 8-bit allocation: 0 ➔ 00000000
- Two's complement can be achieved by reversing all bits except the rightmost bits up to the first 1 (inclusive). If you two's complement a positive number, you get the corresponding negative number. If you two's complement a negative number, you get the corresponding positive number. If you two's complement a number twice, you get the original number.
- Two's complement is the most common, important, and widely used representation of integers today.

Summary of Integer Representation

Contents of Memory	Unsigned	Sign magnitude	One's complement	Two's complement
0000	0	+0	+0	+0
0001	1	+1	+1	+1
0010	2	+2	+2	+2
0011	3	+3	+3	+3
0100	4	+4	+4	+4
0101	5	+5	+5	+5
0110	6	+6	+6	+6
0111	7	+7	+7	+7
1000	8	-0	-7	-8
1001	9	-1	-6	-7
1010	10	-2	-5	-6
1011	11	-3	-4	-5
1100	12	-4	-3	4
1101	13	-5	-2	-3
1110	14	-6	-1	-2
1111	15	-7	-0	-1

3.1.3 Real numbers

It is a number with a fraction and with or without a sign. A decimal point is used to indicate the position of the fraction in the real number. For example 589.63 is a real number with an integer part 589 and a fraction 63. The decimal point immediately after the digit 9 separates the integer and fractional parts in the real number. An optional minus (-) sign preceding an integer part can be used to indicate that a real number is negative.

More formally, a real number is denoted as

Decimal point

$$RN = S d_{n-1} d_{n-2} \ldots d_1 d_0 \,.\, d_{-1}\, d_{-2} \ldots d_{-m}$$

Where **d = {0, 1, 2, 3, ... , R-1}** and R is the radix of a number system used.

Associated with RN, then its value is computed by,

$$\text{Value (RN)} = S \sum d_k * R^k \text{ for } k = -m \text{ to } n$$

The real number data objects defined over a domain with a set of functions and axioms can be treated as a primitive data structure as described below.

Domain: {-∞, ... , 0, ... , ∞ }set of all real numbers.

Functions: Iszero(n), Isnegative(n), Ispositive(n), Add(n1, n2)
Sub(n1, n2), Mul(n1, n2), Div(n1, n2)

Axioms: Add(Sub(n1,n2),n2) = n1
Sub(Add(n1, n2), n2) = n1

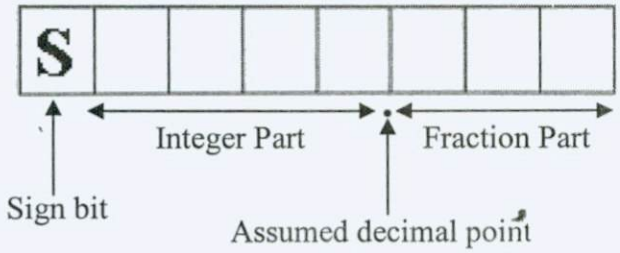
Thus RN:(D,F,A) is a data structure.

3.1.2.1 Representation of Real Numbers

A storage structure consisting of number of memory cells to hold a real number is referred to as real number representation. Any real number has a sign component, an integer component and a fractional component. There are two different ways of representing a real number as described in the following sections.

Fixed Point Representation

In this representation scheme, the position of a decimal point separating the integer part and the fraction part of a real number is fixed. That means, if there are n locations in a storage structure to represent a real number then k locations are used for fractional part, one location is used for sign and the remaining locations are used for integer part. The storage structure representing a real number for $n = 8$ and $k = 3$ is as shown below.



If we want to store the real number 998763482.00567 according to the above storage structure then only partial information is stored leading to high error as shown below.

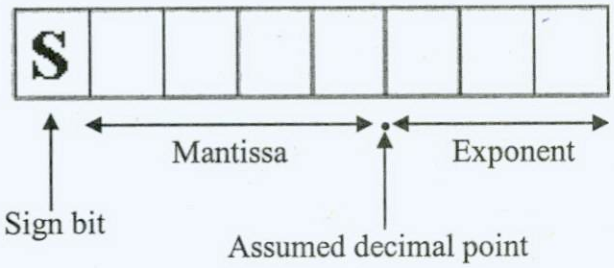
+	3	4	8	2	0	0	5

Because the decimal point is fixed in this representation scheme, four digits (3482) to the left of the decimal point and three digits (005) to the right of the decimal point are considered and the number is represented as shown above. From this example, it is clear that when the real number to be represented has more number of digits in the integer part than the number of cells available in the storage structure, the least

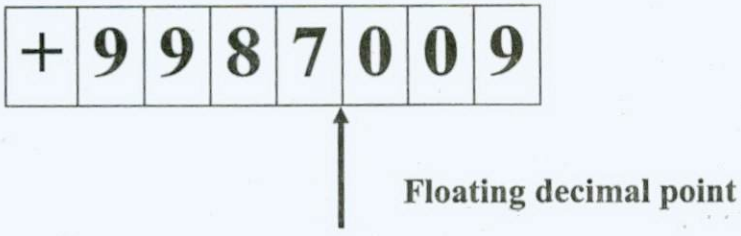
significant digits of the integer part are stored according to the storage structure by discarding most significant digits. Because most significant digits of the integer part are discarded, this representation scheme leads to high error.

Floating Point Representation

In this representation scheme, a real number is represented using a storage structure with three components namely sign, mantissa and exponent. If there are n locations in a storage structure to represent a real number then k locations are used for exponent part, one location is used for sign and the remaining locations are used for mantissa part. The exponent part is in power of the radix of the underlying number system. The storage structure representing a real number for $n = 8$ and $k = 3$ is as shown below.



For example, the real number 998763482.00567 is represented according to the above storage structure as shown below.



The above representation can be interpreted as $0.9987X10^9$, which is equal to 998700000. Since this representation scheme considers the most significant bits for representation, the error is less when compared to fixed point representation scheme. Because the decimal point is moved to the most significant digit position of the integer part of a real number and the exponent is adjusted accordingly, this scheme is called floating point representation scheme. As exponent part is always an integer,

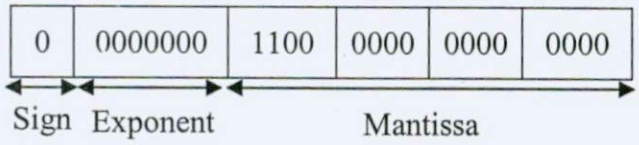
any of the integer representation schemes can be adopted for representing exponent. Let us now consider some example problems to understand the concept.

Examples:

Represent the following numbers in floating point representation using 24 bits out which use one for sign, seven bits for exponent and remaining bits for mantissa.

(a) 0.75

In R = 16 it is 0.C



In R = 8 it is 0.6

0	0000000	110	000	000	000	000	0

In R = 2 it is 0.11

0	0000000	1100000000000000

(b) -943

In R = 16 it is 3AF = 0.3AF X 16^3

1	0000011	0011	1010	1111	0000

In R = 8 it is 1657 = 0.1657 X 8^4

1	0000100	001	100	101	111	000	0

In R = 2 it is $0.1110101111 \times 2^{10}$

1	0001010	1110101111000000

(c) 0.00843

In R = 16 it is $0.022877E = 0.22877E \times 16^{-1}$

0	1000001	0010	0010	1000	0111	Exponent expressed in Signed Magnitude Representation
0	1111110	0010	0010	1000	0111	Exponent expressed in 1's complement
0	1111111	0010	0010	1000	0111	Exponent expressed in 2's complement

In R = 8 it is $0.004241677 = 0.4241677 \times 8^{-2}$

0	1000010	100	010	100	001	1101	Exponent expressed in Signed Magnitude Representation
0	1111101	100	010	100	001	1101	Exponent expressed in 1's complement
0	1111110	100	010	100	001	1101	Exponent expressed in 2's complement

In R = 2 it is $0.022877E = 0.22877E \times 16^{-1}$

0	1000001	0010	0010	1000	0111	Exponent expressed in Signed Magnitude Representation
0	1111110	0010	0010	1000	0111	Exponent expressed in 1's complement
0	1111111	0010	0010	1000	0111	Exponent expressed in 2's complement

Note: Generally, in floating point representation scheme, mantissa part say M, is normalized. A mantissa part is normalized if

$$0.1 \leq M < 1$$

$$R^{-1} \leq M < R^{0}$$

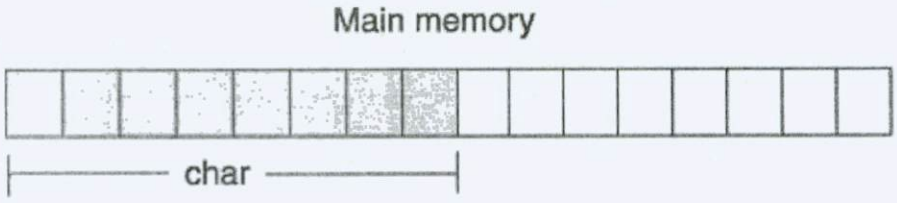
If R=16: $16^{-1} \leq M < 1$, which implies $0.0001 \leq M < 1$, three zeroes are permitted after the radix point in binary equivalent

If R=8: $8^{-1} \leq M < 1$, which implies $0.001 \leq M < 1$, two zeroes are permitted after the radix point in binary equivalent

If R=2: $2^{-1} \leq M < 1$, which implies $0.1 \leq M < 1$, No zeroes are permitted.

3.1.3 Character and its Representation

A character is defined as a symbol used in the alphabet set of a language. A *character* abstract data type is represented as an integer value that corresponds to a character set. A *character set* assigns an integer value to each character, punctuation, and symbols used in a language. Depending on the number of characters to be uniquely identified, *n* number of cells is used in the storage structure in the memory. The figure below shows that 8 locations are used to represent a character in the memory.



For example, the letter *A* is stored in memory as the value 65, which corresponds to the letter *A* in a character set. The computer knows to treat the value 65 as the letter *A* rather than the number 65 because memory was reserved using the char abstract data type. The keyword *char* in a typical programming language tells the computer that the integer stored in that memory location is treated as a character and not a number.

There are two character sets used in programming, the American Standard Code for Information Interchange (ASCII) and Unicode. ASCII is the granddaddy of character sets and uses a byte to represent a maximum of 256 characters. However, a serious

problem was evident after years of using ASCII. Many languages such as Russian, Arabic, Japanese, and Chinese have more than 256 characters in their language. A new character set called Unicode was developed to resolve this problem. Unicode uses 2 bytes to represent each character.

The character data objects defined over a domain with a set of functions and axioms can be treated as a primitive data structure as described below.

Domain: { set of all character used in an application }

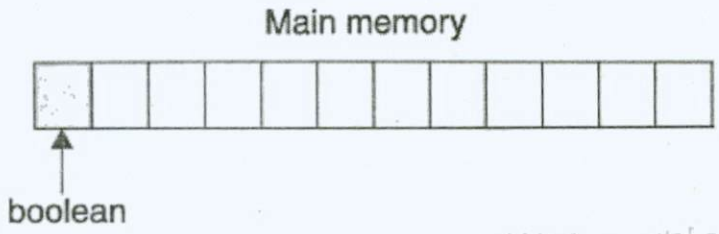
Functions: Ischar(c), Isupper(c), Islower(c), getch(c), putch(c), Comp(c1, c2)
Concat(c1, c2), Ascii(c), upper(c), lower(c).

Axioms: upper(lower(c)) = c, lower(upper(c)) = c

Thus Ch:(D,F,A) is a data structure.

3.1.4 Boolean and its Representation

A Boolean abstract data type reserves memory to store a boolean value, which is a true or false represented as a zero or one. The Boolean data type is used to denote the status of a variable or to know the result of a condition, which is true or false. A Boolean data type requires only one storage location in the memory. We can store as many Boolean values as needed depending on the availability of memory and the requirement by an application. The figure below shows a example storage location in memory for a Boolean type data.



The boolean data objects defined over a domain with a set of functions and axioms can be treated as a primitive data structure as described below.

Domain: { 0, 1 }

Functions: Istrue(c), Isfalse(c)

Axioms: AND(true, false) = false,
AND(false, true) = false,
AND(false, false) = false, AND(true, true) = true,
OR(true, false) = true, OR(false, true) = true,

OR(true, true) = true, OR(false, false) = false,

NOT(true) = false, NOT(false) = true, NOT(NOT(true) = true.

Thus Bl:(D,F,A) is a data structure.

3.2 SUMMARY

In this unit we have studied basic types of primitive data structures, their representation. We saw that integer numbers can be represented as sign magnitude representation, diminished radix representation, radix representation. On the other hand real numbers can be represented either using fixed point representation or floating point representation. Also, we studied merits and demerits of all the representations in detail along with the method to perform basic operations on those primitive data structures. Representing the character and boolean data type is also brought put clearly in this chapter.

3.3 KEY WORDS

(i) Integer data type

(ii) Integer representation

(iii)Real number representation

(iv) Character data representation

(v) Boolean data representation

3.4 QUESTIONS

(1). Explain in detail the different ways of representing the integer numbers.

(2). Bring out the merits and demerits of different representation of integer numbers.

(3). Explain in detail the floating point representation used to represent the real numbers.

(4). Explain in detail the fixed point representation used to represent the real numbers.

(5). Bring out the merits and demerits of different representation of real numbers.

(6). Briefly explain the concept of representing the character type of data

(7). Briefly explain the concept of representing the boolean type of data

3.5 EXERCISE

(1). Represent the real number -87.63×10^{-4} in floating point notation using 24 bits in which the exponent in Excess notation. (1 for sign, 7 for exponent and remaining for mantissa). Mantissa must be normalized mantissa

(a) Binary (b) Octal (c) Hexadecimal

(2). Represent the real number 0.0083412 using 24 bits out of which use 1 bit for sign, 5 bits for exponent and remaining for mantissa (Exponent must be in Excess notation). Give representation in Hexadecimal, Octal and Binary.

3.6 REFERENCE

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

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

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

UNIT – 4

ARRAY

Structure

4.0 OBJECTIVES

After reading this unit you are able to

- Explain the concepts of non-primitive data structure
- Evaluate array as a Abstract Data Type
- Represent array data structures of different dimension
- Discuss row major and column major addressing function

4.1 NON PRIMITIVE DATA STRUCTURES

Non primitive data structures are one which can not be operated upon directly by machine level instructions as whole. These are non atomic or collection of values. A set of objects defined over a domain supporting a set of operations guided by the rules are referred to as non primitive data structures. A non primitive data structure can be

classified as either linear or non linear depending on the organization of the data collection.

4.1.1 Linear Data Structures

A linear data structure is an ordered collection of data elements in which the elements can be labelled by natural numbers in sequence or in other words, a data structure whose data elements can be placed in one-one correspondence with a set of natural numbers so that an element can have at most two adjacent elements is called linear. Arrays, Stacks, Queues, and Records organize the data linearly and thus they are called linear data structures.

4.1.1.1 Array Data Structure

Array is an ordered collection of finite number of homogeneous data elements of fixed size. It is a way to reference a series of memory locations using the same name. Each memory location is represented by an array element. An *array element* is similar to a variable except that it is identified by an index value instead of by name. An *index* value is a number used to identify an array element.

More formally, an array can be defined as an ADT as follows:

Domain: {Application Dependent}

Functions:

- Create an array - CREATEARRAY(A)
- Loading data into an array – LOADARRAY(A)
- Display array elements – DISPLAYARRAY(A)
- Accessing the K^{th} element – ACCESS(A, k)
- Replacing the K^{th} element by e– REPLACE(A, k, e)
- Search for an element e – SEARCH(A, e)
- Compare two elements at m and n locations – COMPARE(A, m, n)

Axioms:

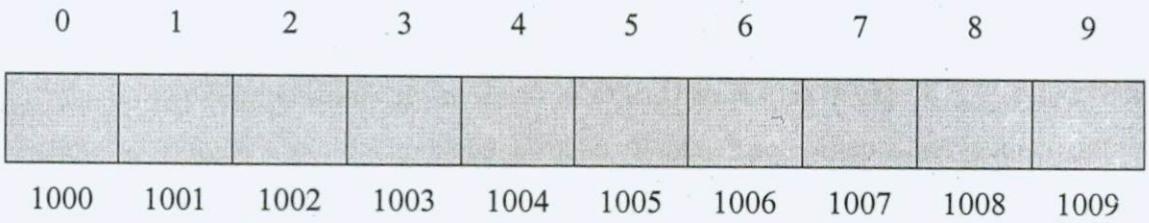
- Should be ordered: Pred(Succ(e)) = e.
- Homogeneous: For all k in A(k), Type(A(k)) is same.

- Size of A is fixed (Static).

Thus Array(Domain, Functions, Axioms) defines array as a data structure.

One-Dimensional Array and its Representation

Logically, an array can be organized as one-dimensional, two-dimensional or multi-dimensional depending on the number of subscripts used to refer an element in the collection. An element in a one-dimensional array can be referred using one subscript. Depending on the number of elements say n, to be stored collectively, an array of *n* contiguous storage locations is defined and the elements are stored. The figure shown below is an array of 10 locations.



As shown above, every location in the memory is identified by a unique value called the address of a memory location. In the above example, the first cell has the address 1000, the second cell has 1001 and so on. The starting cell address is called as the base address of an array. The size of a cell depends on the type of the primitive type of the array. Each element stored is referred using a unique index value called a subscript. The first cell is referred using index value 0, the next cell with 1 and so on. Since the storage locations in an array are contiguous with each cell storing similar type data with uniform size, it is possible to compute the address of any cell by knowing the base address of an array, index value of a cell and the size of the cell. Thus the following expression is used to compute the address of an element in a one-dimensional array.

$$A(k) = B + (k - l) * w$$

Where A is an array name, B is the base address of an array, k is the subscript value, l (generally zero) is the starting value of the subscript used in array indexing, and w is

the size of each cell in bytes. Using the above addressing function, we can access any element in the array with a constant amount of time.

Example 4.1 : In order to understand, how to compute the address of an element in a one dimensional array given its base address B, subscript *k*, and word size *w*, let us consider an array with 10 locations with base address 1000 and word size 1 as describe above.

Consider the addressing function $A(k) = B + (k - l) * w$ (l is 0 in this case)

Address of 1st cell: A(0) = 1000 + (0 – 0) * 1 = 1000

Address of 2nd cell: A(1) = 1000 + (1 – 0) * 1 = 1001

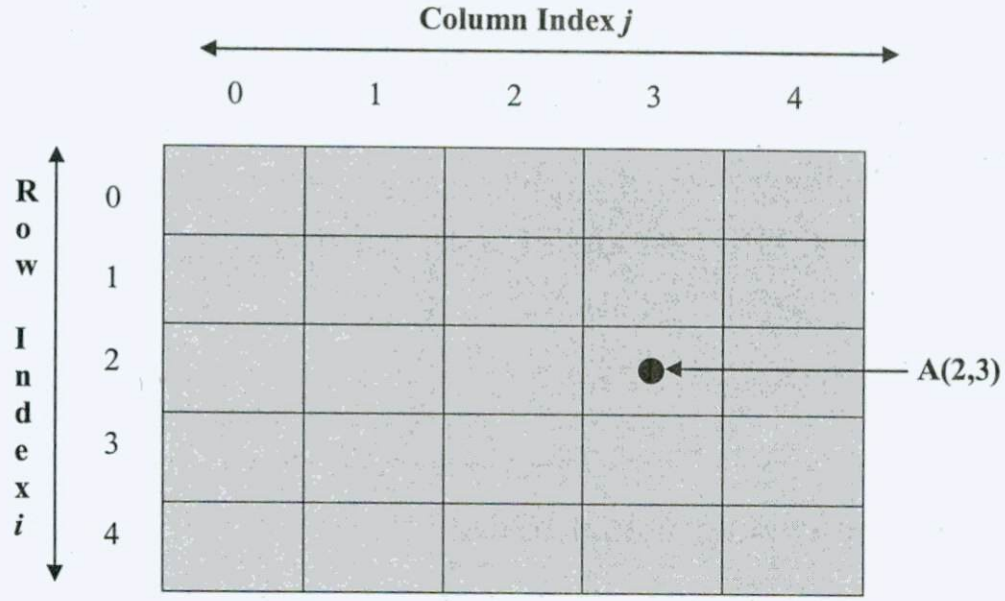
Address of 3rd cell: A(2) = 1000 + (2 – 0) * 1 = 1002

Address of 9th cell: A(9) = 1000 + (9 – 0) * 1 = 1009

Two-Dimensional Array and its Representation

It is often required by most of the applications that an item in a one-dimensional array need to be viewed as a collection of elements rather than a single element. For example, if we want to store the marks scored by 5 students in 3 subjects then we need three one-dimensional arrays each with 5 locations. This requirement can be fulfilled if we logically organize the data as a table of elements with 5 rows and 3 columns. In such an arrangement of data elements, two subscripts are needed to uniquely identify an element – one for row indexing and another for column indexing. This type of logical organization of data is called a two-dimensional array, which is popularly known as a matrix. The following figure shows the logical organization of two-dimensional arrays with 5 rows and 5 columns.

Though a two-dimensional array is logically organized as table of rows and columns, it is represented physically as a list of contiguous locations in the memory each with a unique address. Like 1D-array, address of any location in 2D-array can be computed knowing its base address B, word size W, and the row and column index of a location. Since a 2D-array is physically a list of contiguous locations, there are two ways of addressing a 2D-array, which are usually referred to as Row-Major addressing and Column-Major addressing.



The following expressions are used to compute the address of an element in a 2D-array:

Row-Major Addressing (RMA):

$$A(i, j) = B + \{(i - l_1)(u_2 - l_2 + 1) + (j - l_2)\} * W$$

Column-Major Addressing (CMA):

$$A(i, j) = B + \{(j - l_2)(u_1 - l_1 + 1) + (i - l_1)\} * W$$

Where

- ***B*** is the base address of an array.
- ***W*** is the word size in bytes.
- ***i*** and ***j*** are the subscripts of an element.
- l_1 and u_1 are the lower bound and upper bound of the row index ***i***.
- L_2 and u_2 are the lower bound and upper bound of the column index ***j***.

Three-Dimensional Arrays and its Representation

Sometimes, some applications require that a data need to be organized as a list of two-dimensional arrays. For example, if we want to store the marks scored by 5 students in 3 subjects in 3 tests then we need to logically organize the data as a cube like structure with three dimensions such that one dimension corresponds to the number of students, one dimension corresponds to the number of subjects and one more dimension corresponds to the number of tests. In such an arrangement of data, three subscripts are needed to uniquely identify an element – one for each dimension. This type of logical organization of data is called a three-dimensional array. If i, j, and k denote the three dimensions of an array, then an element is referred as A(i,j,k). The logical organization of the 3D-array can be visualized as shown in Figure 4.1. Though the structure is logically organized as a cube, physically it is organized as a list of contiguous memory locations and hence there are six different ways of computing the address of an element. But the two popular ways are row-major addressing and column-major addressing.

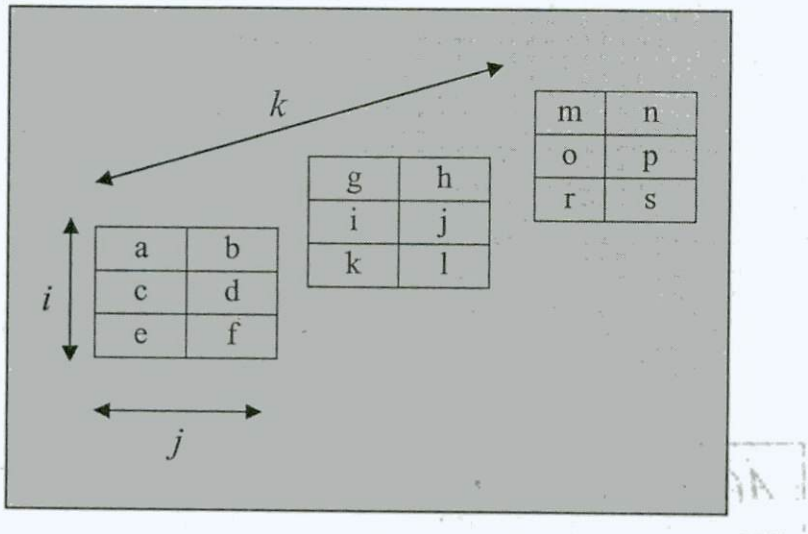


Figure 4.1: Logical organization of elements on 3D array.

Row-Major Addressing:

Column-Major Addressing:

$$A(i, j, k) = B + \{(i - l_1)(u_2 - l_2 + 1)(u_3 - l_3 + 1) + (j - l_2)(u_3 - l_3 + 1) + (k - l_3)\} * W$$

Where

- B is the base address of an array.
- W is the word size in bytes.
- i, j, and k are the subscripts of an element.
- l_1 and u_1 are the lower bound and upper bound of the index i.
- L_2 and u_2 are the lower bound and upper bound of the index j.
- L_3 and u_3 are the lower bound and upper bound of the index k.

N-Dimensional Arrays and its Representation

In theory, the concept of multi dimensional arrays can be extended to any number of dimensions. The number of dimensions required to organize the data depends on the nature of the application and the associated problem to be addressed. In order to refer to an element in an n-dimensional array we need n subscripts. There are $n!$ ways of organizing an n-dimensional array in a physical memory as a list of consecutive elements and hence there are $n!$ ways of computing the address of an element. But the two popular ways are row-major addressing and column-major addressing as discussed earlier in 3D-array.

Row-Major Addressing:

$$
\begin{aligned}
A(i_1,i_2,\ldots,i_n) = B + \{&(i_1 - l_1)(u_2 - l_2+1)(u_3 - l_3+1) \\
&\ldots(u_n - l_n+1) + \\
&(i_2 - l_2)(u_3 - l_3+1)(u_4 - l_4+1) \\
&\ldots(u_n - l_n+1) + \\
&\ldots \\
&(i_{n-1} - l_{n-1})(u_n - l_n+1)+(i_n - l_n)\} * W
\end{aligned}
$$

Similarly, we can obtain an expression to compute the address of an element in Column major addressing.

In general, the above row-major address computing formula can be expressed precisely as follows:

$$A(i_1, i_2, i_3, \ldots, i_n) = B + (\sum(i_j - l_j) * p_j) * w \quad \text{for } j = 1 \text{ to } n$$

$$\text{Where } p_j = \prod(u_k - l_k + 1) \text{ for } k = j+1 \text{ to } n$$

4.2 SUMMARY

In this unit we have studied the basics and types of non primitive data structures. Specifically we have considered linear data structures in which we have particular explained array data structures, their representation for varying dimensions.

4.3 KEYWORDS

(i) Non primitive data structures

(i) Linear data structures

(ii) Array data structures

(iii) Row major addressing

(iv) Column major addressing

4.4 QUESTIONS

(1) Differentiate primitive and non primitive data structures?

(1). Explain in detail array data structure.

(2). Devise an algorithm to compute the address of a three dimensional array at i^{th} row, j^{th} column an p^{th} plane with B as the base address and W as the word size.

4.5 EXERCISE

(1). Consider two dimensional array A ($10 \leq i \leq 13$, $-5 \leq j \leq -1$). Find out the values of the subscripts i and j if A(i, j) is stored at the location whose address is 36 when A is in row major and the element A(i, j) is stored at the

location whose address is 30 when A is in column major. Assume base address equal to 0 and word size as three.

(2). Consider two dimensional array A ($3 \leq i \leq 6, 5 \leq j \leq 9$). Assume that A is in row major ordering with B = 1000 and W = 2. Find the addresses of A(4,6) and A(5,8) using both rows major ordering and column major ordering?

4.6 REFERENCE

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

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

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

UNIT - 5

STACKS

Structure

5.0 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

5.1 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.

5.2 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.

5.3 IMPLEMENTATION OF STACKS

A common and a basic method of implementing stacks is to make use of another fundamental data structure namely, array, which is sequential and static. An

alternative way to realize a stack is to use linked lists, which are non-sequential and dynamic. The following sections describe the stack implementation using arrays.

5.3.1 Array Implementation of a Stack

The fact that stacks are uni-dimensional ordered lists make it convenient to use arrays for its implementation. Arrays, despite their multi-dimensional structure are inherently associated with a one-dimensional consecutive set of memory locations, which are a natural choice for realizing stacks. Figure 1.1 illustrates an array based implem ntation of stacks. The figures shows a stack of six elements D, S, G, H, S, N represented by an array STACK of size 10 elements. In general, we can define a stack, represented by an array of *n* elements depending on the requirement. Therefore, it becomes necessary to signal "STACK OVERFLOW" when we attempt to store more than *n* elements in to a stack and "STACK UNDERFLOW" when we try to remove an element from an empty stack. As shown in the Figure 1.1, one end of the stack called *Bottom of Stack*, is the point from where insertion of elements starts and the other end called *Top of Stack*, which keeps track of the element entered recently. Elements D, S, G, H, S, and N are entered in the same sequence and hence element D is placed at the bottom of the stack and the recently entered element N is placed on the top of the stack.

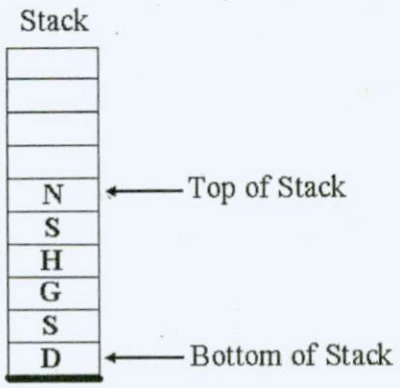


Figure 1.1 A sample stack implemented using array data structure.

5.3.2 Implementation of Stack Operations

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

Algorithm: Create-Stack()

Input: Size of stack, *n*.

Output: Stack S created.

Method: 1. Declare an array of size n.

2. Initialize the pointer TOP to zero.

3. Return.

Algorithm ends.

Algorithm: Isempty(S)

Input: Stack S and a pointer TOP.

Output: True or False.

Method: 1. If (TOP = 0) Then return (True)

Else return (False)

Algorithm ends.

Algorithm: Isfull(S)

Input: Stack S and a pointer TOP.

Output: True or False.

Method: 1. If (TOP = n) Then return (True)

Else return (False)

Algorithm ends.

Algorithm: Push(S)

Input: Stack S and element e to be inserted.

Output: Stack S updated.

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

Else

TOP = TOP + 1

S(TOP) = e.

2. Return.

Algorithm ends.

Algorithm: Pop(S)

Input: Stack S.

Output: Element e.

Method: 1. If (Isempty(S)) Then Display "STACK UNDERFLOW"

Else

e = S(TOP)

TOP = TOP - 1

2. Return.

Algorithm ends.

Algorithm: Top(S)

Input: Stack S and a pointer TOP.

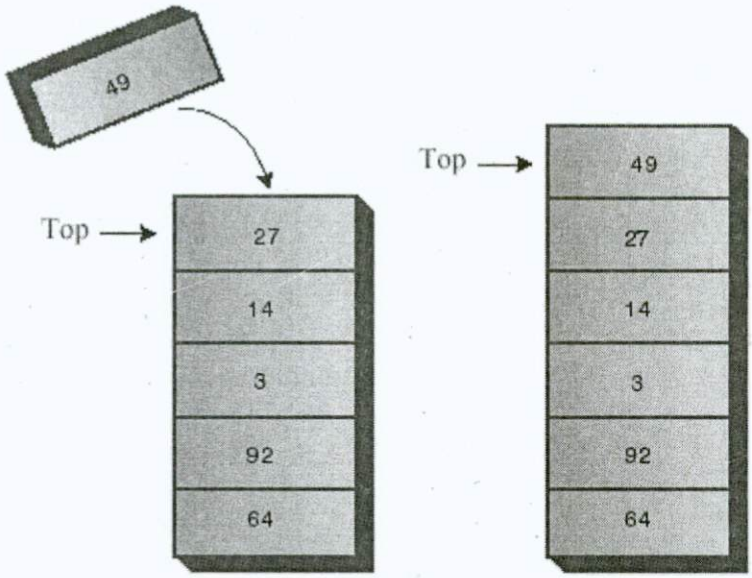
Output: Element on the top of Stack.

Method: 1.If (Isempty(S)) Then Display "STACK UNDERFLOW"

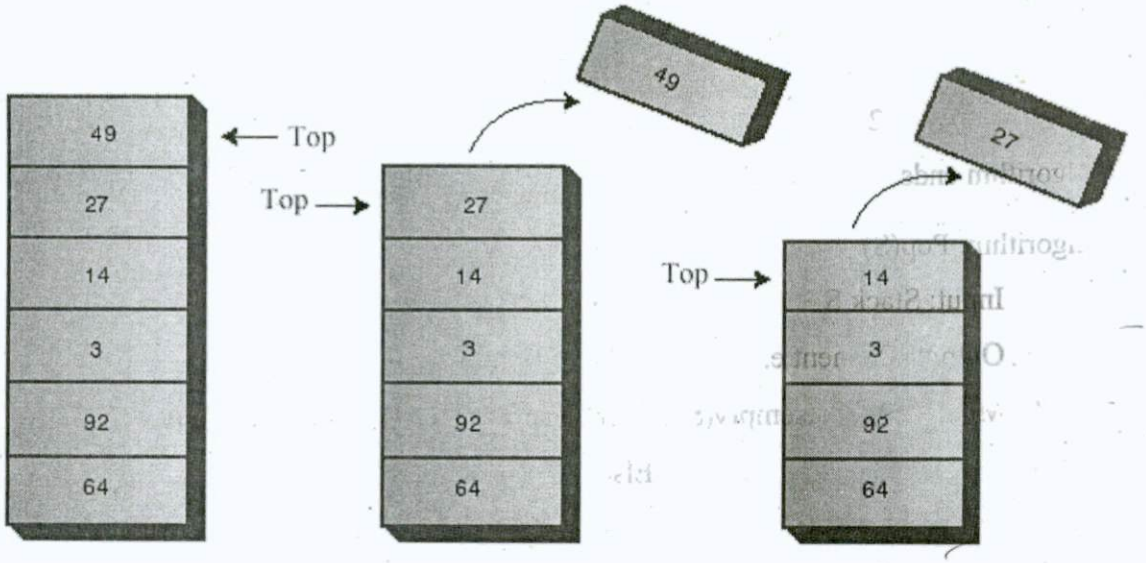
Else return (S(TOP))

Algorithm ends.

(Here the term underflow is used on par with the term overflow)



New item pushed on stack



Two items popped from stack

Figure 1.2 Illustrations of Push and Pop stack operations

Figure 1.2 demonstrate the working principle of pushing an element 47 into the stack and poping of two elements from the stack.

5.4 SUMMARY

In this unit, we have presented the concept of stack data structure and the ways of representing the stack using array as a data structure. In this unit the basic operations and implementation details of stack are demonstrated in the form of algorithms with suitable examples.

5.5 KEY WORDS

(1). Stack data structure

(2). Push

(3). Pope

(4). Top

5.6 QUESTIONS

(1). Explain stack data structure with an example.

(2). Mention the basic operations of stack data structure.

(3). Design and develop an algorithm to push and pop elements from the stack.

5.7 REFERENCES

(1). Ellis Horowitz and Sartaj Sahni. Fundamentals of Data Structures. W H Freeman & 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 and Algorithms. Addison Wesley (January 11, 1983)

UNIT - 6

APPLICATIONS OF STACK

Structure

6.0 OBJECTIVES

After reading this unit you should be able to

- Evaluate different types of expression.
- Explain the mechanism of converting infix expression to equivalent suffix and prefix expressions.

6.1 INTRODUCTION

The concept of stack data structures finds its application in effectively solving various kinds of problems. Realizing nested function calls in programming, conversion of expressions, evaluation of expressions, checking the validity of expressions, finding whether a string is palindrome or not, are few examples to name among several.

6.2 EXPRESSIONS

An expression can be defined as a sequence of operands and operators that reduces to a single value after evaluation. Operands can be constant values or variables and

operators can be symbols such as +, -, *, /, ^ and so on. The operators used in an expression indicate the operations to be performed on the operands involved. Expressions, which involve only arithmetic operators are called arithmetic expressions. More formally, an arithmetic expression can be defined as follows:

1. An identifier is an expression.
2. Constant is an expression.
3. If E_1 and E_2 are expressions then $(E_1 + E_2)$, $(E_1 - E_2)$, $(E_1 * E_2)$, (E_1 / E_2), $(E_1 ^ E_2)$ are all expressions.
4. If E is an expression the (E) is also an expression.

An arithmetic expression can be represented in three different forms namely, infix, postfix, and prefix. We define these three forms of an expression as follows.

6.2.1 Infix Expression

An expression in which an operator is appeared in between two operands is termed as an infix expression. It can be parenthesized or un-parenthesized. Following are the examples for infix expressions.

(i) a + b * c (ii) (a + b) / c (iii) a – b ^ c

6.2.2 Postfix Expression

An expression in which an operator follows the two operands is termed as a postfix expression. No parentheses are used in this form of an expression. It is also called as suffix expression or reverse polish expression. Following are the examples for postfix expressions.

(i) abc* + (ii) ab + / c (iii) abc^ –

6.2.3 Prefix Expression

An expression in which an operator precedes the two operands is termed as a prefix expression. No parentheses are used in this form of an expression. It is also called as polish expression. Following are the examples for prefix expressions.

(i) +a*bc (ii) /+abc (iii) –a^bc

6.3 EVALUATION OF EXPRESSIONS

The process of finding out the value of an expression is called as evaluation of an expression. The following example illustrates the process of evaluation of infix expression.

Example 2.1:

$$E = 9 + \mathbf{3 * 5} - 4 / 2$$
$$= 9 + 15 - \mathbf{4 / 2}$$
$$= \mathbf{9 + 15} - 2$$
$$= \mathbf{24 - 2}$$
$$= 22$$

Example 2.2:

$$E = \mathbf{(9 + 3)} * 5 - 4 / 2$$
$$= \mathbf{12 * 5} - 4 / 2$$
$$= 60 - \mathbf{4 / 2}$$
$$= \mathbf{60 - 2}$$
$$= 58$$

Example 2.33:

$$E = 2\text{^}\mathbf{3\text{^}2}$$
$$= \mathbf{2\text{^}9}$$
$$= 512$$

The above examples illustrate that the process of arithmetic expression evaluation is guided by some rules. Some expressions are given precedence over the other expression and are evaluated first. These rules that determine the order in which different operators are evaluated are called **precedence rules** or **precedence of operators**. In order to determine the precedence of operators, a numeric value is associated with each operator. An operator with highest precedence will have high value and an operator with lowest precedence will have the lowest value. If two or more operators have the same precedence, then the associativity of the operators are considered for evaluation. The order in which the operators with same precedence are evaluated in an expression is called **associativity of the operator**. The Table 2.1 shows arithmetic operators along with their priority values and associativity.

Table 2.1 Priority and Associativity of arithmetic operators

Operator	Priority	Associativity
Exponentiation (^)	3	Right to Left
Multiplication (*)	2	Left to Right
Division (/)	2	Left to Right

Modulo (%)	2	Left to Right
Addition (+)	1	Left to Right
Subtraction (-)	1	Left to Right

The above table shows that an exponentiation operator (^) has high priority value and its associativity is Right to left. That means, when two exponentiation operators are appeared consecutively in an expression then they are evaluated from right to left. Example 2.3 above illustrates this fact. Similarly, all the other arithmetic operators shown in the Table 2.1 are left to right associative. That means, when any of these operators appeared consecutively in an expression with same priority then they are evaluated from left to right. Example 2.1 above illustrates this fact. From the above examples, it is understood that the evaluation of infix expression involves the following steps:

1. Scanning the entire expression iteratively to locate the operator having high priority and to evaluate the sub expression associated with that operator. The number of iterations is equal to the number of operators used in the infix expression.
2. Associativity property of an operator need to be considered while evaluating an expression.
3. Parentheses are required to override the default priority.

Because of the above steps, it is a time consuming task for a machine to evaluate a lengthy expression. This problem can be eliminated if we represent an infix expression either in prefix form or in postfix form. The process of converting an infix expression to its equivalent postfix or prefix requires a stack data structure as explained in the following sections.

6.4 CONVERSION OF INFIX EXPRESSION TO POSTFIX

The process of converting infix expression to postfix requires scanning the expression from left to right searching for an operator with highest priority. If two operators with same priority appeared consecutively in the expression then the associativity property of that operator is considered and the sub expression connected by the operator is converted to postfix form. This post fix form of a sub expression is assigned to a temporary variable and the original sub expression in the infix expression is replaced by this temporary variable. This process is repeated until all the operators in the infix

expression are scanned and the corresponding sub expressions are converted to postfix form resulting with a final intermediate form. The overall postfix form of an infix expression is then obtained through repeated substitution. The above procedure can be explained clearly with an example as follows:

Consider an expression E = (A + (B – C) * D) ^ E + F

(A + **(B – C)** * D) ^ E + F	(B – C) has highest precedence because of parenthesis. The intermediate postfix form is T_1 = B C -
(A + **T_1 * D**) ^ E + F	(T_1 – D) has highest precedence because of parenthesis and *. The intermediate postfix form is T_2 = T_1 D *
(A + T_2) ^ E + F	(A + T_2) has highest precedence because of parenthesis. The intermediate postfix form is T_3 = A T_2 +
T_3 ^ E + F	(T_3 ^ E) has highest precedence because of ^. The intermediate postfix form is T_4 = T_3 E ^
T_4 + F	(The only one operator and hence the intermediate postfix form is T_5 = T_4 F +

Now consider the expression T_4F+ and perform repetitive substitution to get the postfix expression.

T_4 F +

T_3 E ^ F +

A T_2 + E ^ F +

A T_1 D * + E ^ F +

A B C - D * + E ^ F +

Hence, the equivalent postfix expression is **A B C - D * + E ^ F +**

Algorithm: Infix to Postfix

Input: Infix Expression, IE

Output: Postfix Expression, PE

Method:

1. S = Create Stack()

2. Push (S, $)
3. Priority ($) = -1

   Priority (( ) = 0

   Priority (+, -) = 1

   Priority (*, /) = 2

   Priority (^) = 3
4. PE = 0
5. C = get-symbol (IE)
6. While (C ≠ End-of-Character) Do

   Begin

   Switch(C)

   Case1. Operand: PE = concatenate (PE, C)

   Case 2. '(': Push (S, C)

   Case 3. ')': While (Top(S) ≠ '(') Do

   PE = Concatenate (PE, Pop(S))

   End-of-While

   Pop(S)

   Case 4. '^': Push(S, C)

   Otherwise: While (Priority(C) ≤ Priority (Top(S)))

   PE = concatenate (PE, Pop(S))

   End-of-While

   Push(S, C)

   End-of-Switch

   C = get-symbol (IE)

   End-of-While

   While (Top(S) ≠ '$')

   PE = concatenate (PE, Pop(S))

   End-of-While

Algorithm ends.

Now let us consider an example to trace the above algorithm and to obtain the postfix equivalent of an infix expression.

Infix Expression IE	Stack Contents	Postfix Expression PE

A + B * C ^ D	$	NULL
+ B * C ^ D	$	A
B * C ^ D	$ +	A
* C ^ D	$ +	AB
C ^ D	$ + *	AB
^ D	$ + *	ABC
D	$ + * ^	ABC
NULL	$ + * ^	ABCD
NULL	$	ABCD ^ * +

6.5 EVALUATION OF POSTFIX EXPRESSION

In order to valuate the postfix expression, we have to use stack data structure. Since the postfix expression contains only operators and operands, the following procedure is used while evaluating the postfix expression.

1. Give the Postfix expression; scan the symbol from left to right.
2. If the scanned symbol is an operand, place it on to the stack.
3. If the scanned symbol is an operator, pop top two elements from the stack such that the first popped element is operand-2 and second popped element is operand-1.
4. Perform the indicated operation as Result = Operand-1 operator Operand-2.
5. Push the result on to the stack.
6. Repeat the steps 1 to 5 until no more symbols in the Postfix expression.

This procedure can be precisely expressed in the form of an algorithm as follows:

Algorithm: Postfix expression evaluation.

Input: Postfix Expression, PE

Output: Value of the expression, V

Method:

1. C = get-symbol (PE)
2. While (C ≠ NULL) Do

   If (C = Operand) then Push (S, C)

   Else

Op2 = Pop (S)

Op1 = Pop (S)

V = Op1 C Op2

Push (S, V)

End-of-if

C = get-symbol (PE)

End-of-While

V = Pop(S)

3. Stop

Algorithm ends.

Example for evaluation of a postfix notation expression

Infix notation: A + B * C – D

Post Fix notation: A B C * + D -

For A = 2, B = 3, C = 4, and D = 5, the above infix expression would yield 2 + 3 * 4 – 5 the value 9.

Let us see how our stack goes around doing its job

Step 1: First 3 symbols are operands **A B C** i.e. **2, 3, 4**; hence they are pushed on to stack. Stack entries are: **C B A i.e. 4 3 2**

Step 2: Now you will encounter *. Therefore, pop two top most operands from stack and perform the operation and push the result on to stack.

Operands popped: C & B i.e. 4 * 3 = 12

Push result on to stack.

Stack entries at this stage are: **12 2**

Step 3: Now you will encounter +

POP 12 & 2 and Result = **12 + 2**

Push on to stack. Stack entry now will be: **14**

Step 4: Next encounter is **D**. Push it on to stack. Stack entries at this stage: **5 14**

Step 5: Next encounter is **-** . Therefore, pop two entries from stack i.e. **5 & 14**.

Perform the operation **14 – 5 = 9**.

Push the result on to stack: **9**

Step 6: Pop the content of the stack i.e. **9** and assign it to the variable **V**.

Thus, **V = 9.**

Step 7: No more symbols in the postfix expression, therefore, stop of the procedure.

6.6 CONVERSION OF INFIX EXPRESSION TO PREFIX

The procedure followed to convert the given infix expression to its equivalent prefix can be extended to convert the infix expression to its equivalent prefix with little modification. The following example illustrates the procedure.

Consider an expression E = (A + (B – C) * D) ^ E + F

(A + **(B – C)** * D) ^ E + F	(B – C) has highest precedence because of parenthesis. The intermediate prefix form is T_1 = - B C
(A + **T_1 * D**) ^ E + F	(T_1 – D) has highest precedence because of parenthesis and *. The intermediate prefix form is T_2 = * T_1 D
(**A + T_2**) ^ E + F	(A + T_2) has highest precedence because of parenthesis. The intermediate prefix form is T_3 = + A T_2
T_3 ^ E + F	(T_3 ^ E) has highest precedence because of ^. The intermediate prefix form is T_4 = ^ T_3 E
T_4 + F	(The only one operator and hence the intermediate prefix form is T_5 = + T_4 F

Now consider the expression + T_4F and perform repetitive substitution to get the prefix expression.

+ T_4 F

+ ^ T_3 E F

+ ^ + A T_2 E F

+ ^ + A * T_1 D E F

+ ^ + A * - B C D E F

Hence, the equivalent prefix expression is: **+ ^ + A * - B C D E F**

Algorithm: Infix to Prefix

Input: Infix Expression, IE

Output: Prefix Expression, PRE

Method:

7. S = Create Stack()
8. Push (S, $)
9. Priority ($) = -1

   Priority ( )) = 0

   Priority (+, -) = 1

   Priority (*, /) = 2

   Priority (^) = 3
10. PRE = NULL
11. Reverse the given infix expression.
12. C = get-symbol (IE)
13. While (C ≠ End-of-Character) Do

    Begin

        Switch(C)

            Case1. Operand: PRE = concatenate (PRE, C)

            Case 2. ') ': Push (S, C)

            Case 3. ' (': While (Top(S) ≠ ') ') Do

                PRE = Concatenate (PRE, Pop(S))

                End-of-While

                Pop(S)

            Case 4. '^': If (Priority(C) = Priority (Top(S))

                PRE = Concatenate (PRE, C)

                Else Push (S, C)

            Otherwise: While (Priority(C) < Priority (Top(S)))

                PRE = concatenate (PRE, Pop(S))

                End-of-While

                Push(S, C)

        End-of-Switch

        C = get-symbol (IE)

    End-of-While

    While (Top(S) ≠ '$')

        PRE = concatenate (PRE, Pop(S))

End-of-While

Reverse the string PRE to obtain the equivalent prefix expression.

Algorithm ends.

Now let us consider an example to trace the above algorithm and to obtain the prefix equivalent of an infix expression A + B * C ^ D.

Reverse the expression A + B * C ^ D and follow the steps.

Infix Expression IE	Stack Contents	Prefix Expression PRE
D ^ C * B + A	$	NULL
^ C * B + A	$	D
C * B + A	$ ^	D
* B + A	$ ^	D C
B + A	$ *	D C ^
+ A	$ *	D C ^ B
A	$ +	D C ^ B *
A	$ +	D C ^ B * A
NULL	$	D C ^ B * A +

Now, reverse the string to get the equivalent prefix expression: + A * B ^ C D

Let us consider another example to illustrate the procedure A ^ B ^ C * D.

Reverse the expression (D * C ^ B ^ A) and follow the steps.

Infix Expression IE	Stack Contents	Prefix Expression PRE
D * C ^ B ^ A	$	NULL
* C ^ B ^ A	$ *	D
C ^ B ^ A	$ *	D C
^ B ^ A	$ * ^	D C
B ^ A	$ * ^	D C B
^ A	$ * ^	D C B ^
A	$ * ^	D C B ^ A
NULL	$ * ^	D C B ^ A ^ *

Now, reverse the string to get the equivalent prefix expression: * ^ A ^ B C D

Some examples:

1. Convert (A + B) * ((C + D) – E) * F to prefix expression

   * * + A B - + C D E F

2. Convert A + B * C + D – E * F to prefix expression

   - + + A * B C D * E F

3. Convert A - B / (C * D ^ E) to prefix expression

   - A / B * C ^ D E

4. Convert ((A + B) * C – (D – E)) ^ (F + G) to prefix expression

   ^ - * + A B C - D E + F G

5. Convert A ^ B * C - D + E / F / (G + H) to prefix expression.

   + - * ^ A B C D / / E F + G H

Note: Trace the Infix to Prefix conversion algorithm to convert the above infix expressions to its equivalent prefix expression and verify your results.

6.7 EVALUATION OF A PREFIX EXPRESSION

The procedure to evaluate prefix expression is same as postfix expression. Since the prefix expression has operators followed by operands, we scan the prefix expression from right to left and push the operands into the stack until an operator is encountered. When an operator is encountered, remove the top two operands from the stack and

perform the operation using the operator and push the result into the stack. This process is continued until all the symbols in the prefix expression are scanned and the corresponding operations are performed finally storing the result of an expression on top of the stack. Assign the top of the stack to a variable to hold the result of an expression. The following example illustrates the process of evaluating the prefix expression.

Example:

Infix expression: **A - B / (C * D ^ E)**

Equivalent prefix expression: **- A / B * C ^ D E**

Let A = 10, B = 48, C = 3, D = 4, and E = 2. The value of infix expression is computed as 10 – 48 / (3 * **4 ^ 2**) = 10 – 48 / (**3 * 16**) = 10 – **48 / 48** = **10 – 1** = 9

The final result of the expression is **9**.

Let us verify whether we get the above result if we follow the procedure illustrated for evaluating the prefix expression.

Prefix Expression	Symbol scanned	Op2	Op1	Result Op1 op Op2	Stack Contents
$ - 10 / 48 * 3 ^ 4 2	2				2
$ - 10 / 48 * 3 ^ 4	4				2, 4
$ - 10 / 48 * 3 ^	^	4	2	4 ^ 2 = 16	16
$ - 10 / 48 * 3	3				16, 3
$ - 10 / 48 *	*	16	3	16 * 3 = 48	48
$ - 10 / 48	48				48, 48
$ - 10 /	/	48	48	48 / 48	1
$ - 10	10				1, 10
$ -	-	10	1	9	9
$	Thus the result obtained after evaluation is **9**. Same as the above infix expression evaluation.				

6.8 SUMMARY

- A stack allows access to the last item inserted, at the top of the stack.
- The important stack operations are pushing (inserting) an item onto the top of the stack and popping (removing) the item from the top.
- A stack is often helpful in parsing a string of characters, among other applications.
- A stack can be implemented with an array or with another mechanism, such as a linked list.

6.9 KEY WORDS

(1). Linear data structure

(2). Stack

(3). Infix expression

(4). Postfix expression

6.10 EXERCISES

1. Write a C program using pointers to implement a stack with all the operations.

2. Write a program to convert a given prefix expressions to postfix expression using stacks

3. Write a program to evaluate a postfix expression using stack.

4. Convert following infix expressions to prefix and postfix expressions.

   a) A-B+C b) A*B+C

   c) A+B*C d) A*(B+C)

   e) (A-B)*)*(C+D)^E/F f) (A+B)/(C^ (D-E)+F)- G

   g) A + (((B + C) * (D – E) – F) / G) ^ (H – J)

5. Convert following prefix expressions in to infix expression.

   a) + - * ^ ABCD / / EF+GH

   b) ^ - * + ABC – DE + FG

   c) * * + AB - + CDEF

d) - + + A * BCD * EF

6. Convert following postfix expressions to infix expression

a) AB ^ C * D - EF / GH + / +

b) AB + C * DE - - FG + ^

c) AB + CD + E - * F *

d) ABC * + D + EF * -

7. With A =1, B =2, C=3, and D = 4 evaluate following postfix expressions

a) AB+Cb)

AB-C+DEF- + *

c) AB + C – BA + C / -

d) ABCDE - + * * EF * -

6.11 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

RECURSION

Structure

7.0 OBJECTIVES

After reading this unit you should be able to

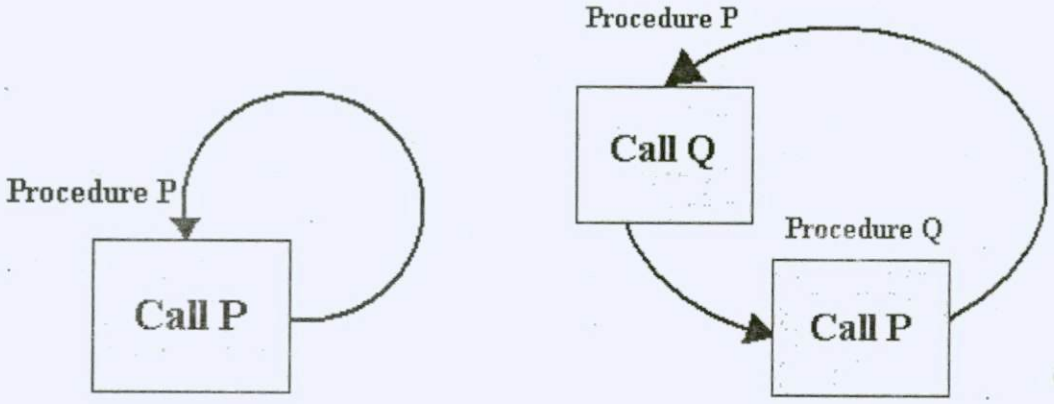
- Define the meaning of recursion
- Explain the working principle of recursion
- Formalize the solution to a problem using recursion
- Discuss the properties of recursive procedure
- Compare and contrast between recursion and iterative procedures

7.1 INTRODUCTION

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 7.1 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 x 4!
4! = 4 x 3!
3! = 3 x 2!
2! = 2 x 1!
1! = 1 x 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.

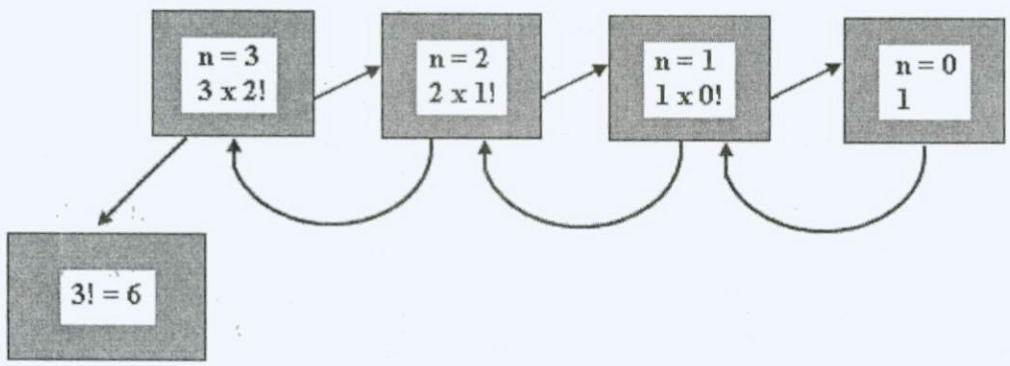


Figure 7.2 Sequence of recursive function calls to compute the factorial of 3

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.

7.2 SOME EXAMPLES OF RECURSIVE FUNCTIONS

(1) Sum of natural numbers: $$\text{N-sum}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 + \text{N-sum}\ (n-1) & \text{if } n > 0 \end{cases}$$

(2) GCD of two numbers:

$$\text{GCD}(m, n) = \begin{cases} \text{GCD}(n, m) & \text{if } m < n \\ \text{m} & \text{if } n = 0 \\ \text{GCD}(n, m \bmod n) & \text{if } m > n \end{cases}$$

(3) Fibonacci number:

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fin}(n-1) + \text{Fib}(n-2) & \text{if } n > 2 \end{cases}$$

(4) Multiplication of numbers:

$$\text{Mul}(m, n) = \begin{cases} 0 & \text{if } m = 0 \text{ or } n = 0 \\ m & \text{if } n = 1 \\ \text{Mul}(m, n-1) + m & \text{otherwise} \end{cases}$$

(5) Maximum of *n* elements:

$$\text{Max}(A[l..u]) = \begin{cases} a = \text{Max}(A[l, (l+u)/2]) > b = \text{Max}(A[(l+u)/2 + 1, u]) \ ?\ a : b & \text{if } l < u \\ A(l) & \text{if } l = u \end{cases}$$

(6) Binary Search:

$$\text{B-Search}(A[l..u], e) = \begin{cases} \text{Mid} = (l + u) / 2 & \\ \text{Found} & \text{if } (e = A(\text{Mid})) \\ \text{B-Search}(A[l..\text{Mid} - 1], e) & \text{if } (e < A(\text{Mid})) \\ \text{B-Search}(A[\text{Mid} + 1, u], e) & \text{if } (e > A(\text{Mid})) \\ \text{Not Found} & \text{if } (l > u) \end{cases}$$

7.3 THE TOWERS OF HANOI PROBLEM

The Towers of Hanoi is an ancient puzzle consisting of a number of disks placed on three needles, as shown in Figure 3.3.

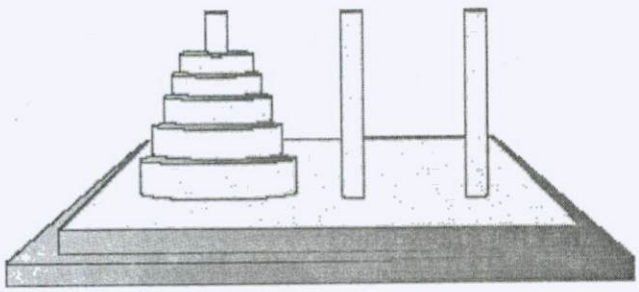


Figure 7.3 The Towers of Hanoi.

The disks all have different diameters and holes in the middle so they will fit over the needles. All the disks start out on column A. The object of the puzzle is to transfer all the disks from needle A to needle C. Only one disk can be moved at a time, and no disk can be placed on a disk that's smaller than itself.

7.3.1 The Recursive Algorithm

The solution to the Towers of Hanoi puzzle can be expressed recursively using the notion of subtrees. Suppose we want to move all the disks from a source tower (S) to a destination tower (D). We have an intermediate tower available (T). Assume that there are n disks on tower S. We can carry out the algorithm as follows:

- Move the subtree consisting of the top $n-1$ disks from S to T.
- Move the remaining (largest) disk from S to D.
- Move the subtree from T to D.

When we begin, the source tower is A, the intermediate tower is B, and the destination tower is C. Figure 3.4 shows the three steps for this situation.

First, the subtree consisting of disks 1, 2, and 3 is moved to the intermediate tower B. Then the largest disk, 4, is moved to tower C. Then the subtree is moved from B to C.

Of course, this doesn't solve the problem of how to move the subtree consisting of disks 1, 2, and 3 to tower B because we can't move a subtree all at once; we must move it one disk at a time. Moving the 3-disk subtree is not so easy. However, it's easier than moving 4 disks.

As it turns out, moving 3 disks from A to the destination tower B can be done with the same 3 steps as moving 4 disks. That is, move the subtree consisting of the top 2 disks from tower A to intermediate tower C; then move disk 3 from A to B. Then move the subtree back from C to B.

How to move a subtree of two disks from A to C? Move the subtree consisting of only one disk (1) from A to B. This is the base case: when we are moving only one disk, we just move it; there is nothing else to do. Then move the larger disk (2) from A to C, and replace the subtree (disk 1) on it. Figure 3.4 illustrated the method of solving the problem.

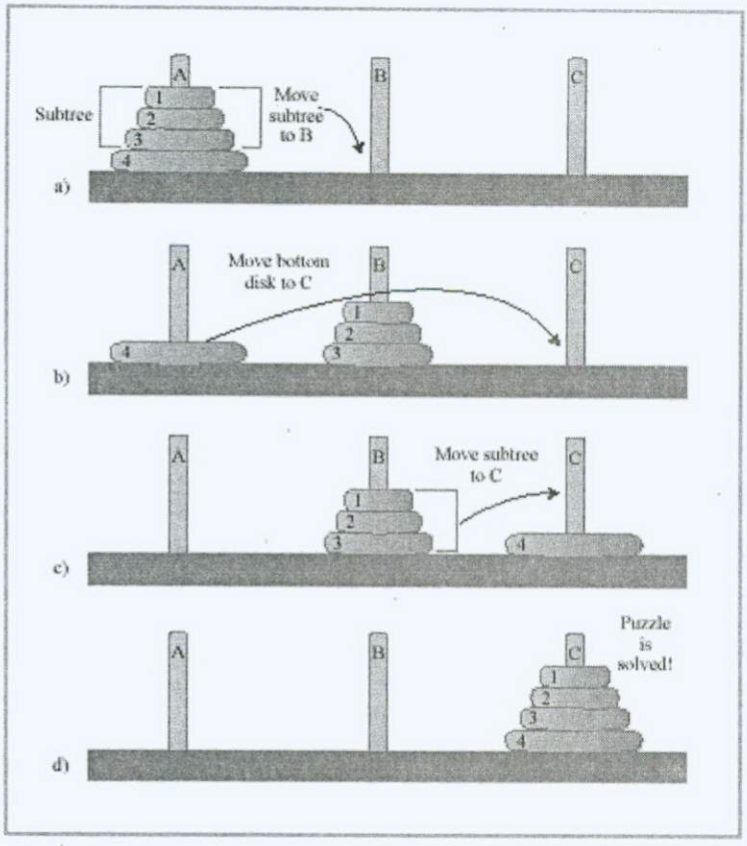


Figure 7.4 Recursive solution to Towers of Hanoi.

The above recursive procedure can be algorithmically expressed as follows:

$$\text{TOH}(n, S, T, D) = \begin{cases} \text{TOH}(n-1, S, D, T) \\ \text{Move a disc from S to D} \\ \text{TOH}(n-1, T, S, D) \end{cases} \quad \text{if } n > 0$$

$$\text{Exit} \quad \text{if } n = 0$$

The above procedure is illustrated by considering Tower-of-Hanoi of 3 discs in Figure 7.5.

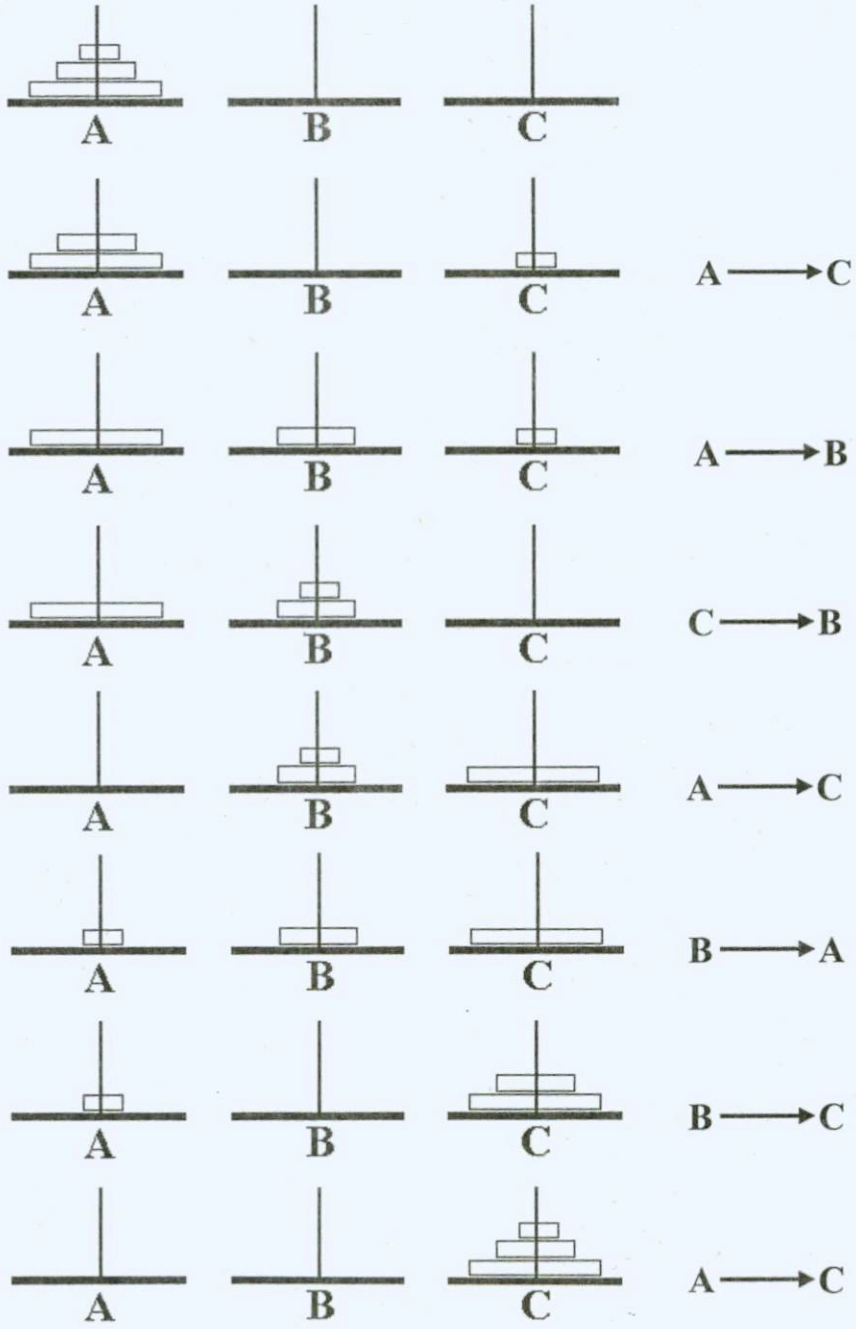


Figure 7.5 Sequence of disc movements in Towers of Hanoi puzzle for $n = 3$.

From the above example, we draw the following observations:

1. No explicit repetition using for, while etc. only implicit repetition.
2. For 2 discs 3 ($2^2 - 1$) movements are required.
3. For 3 discs 7 ($2^3 - 1$) movements are required.
4. For 4 discs 15 ($2^4 - 1$) movements are required.
5. In general, for n discs $2^n - 1$ disc movements are required.

The recursive tree diagram for the Tower-of-Hanoi problem for 3 discs is shown in Figure 3.6.

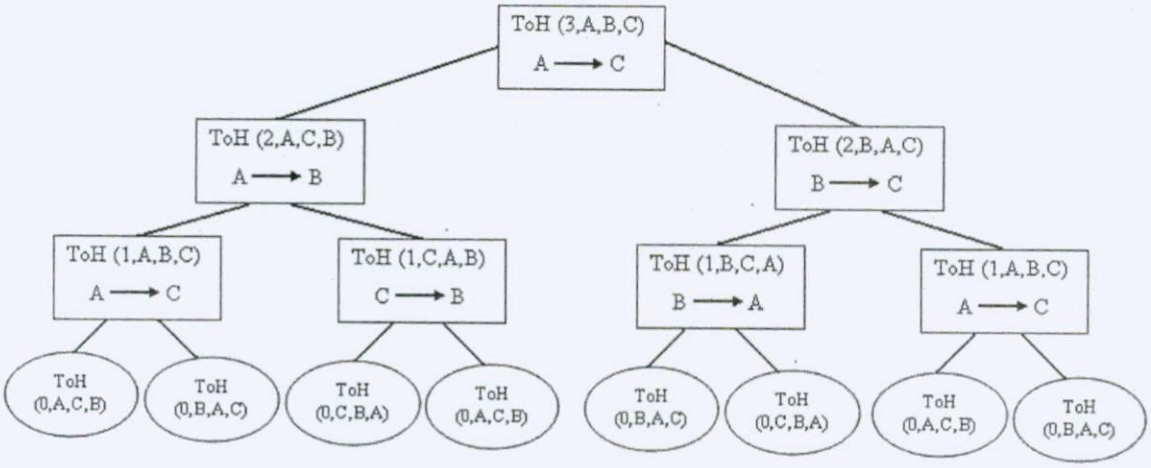


Figure. 3.6 Recursion three diagram for Tower of Hanoi for 3 discs.

In the above diagram, the leaf nodes indicate the termination condition. If we traverse the tree in inorder, we get the sequence of moves as follows:

$$A \rightarrow C, A \rightarrow B, C \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C, A \rightarrow C$$

7.4 EFFICIENCY OF RECURSION

Recursive functions are not efficient in terms of execution time and space utilization when compared to non-recursive functions. When recursive functions make a call directly or indirectly to itself, they have to save function status information such as formal parameters, local variables, return address etc. to ensure proper functioning. Besides, separate copy of the function variables are created for each call, which consumes lot of memory. Once the base criteria is met, the recursive function must restore the most recently saved parameters, local variables, and return addresses and thus most of the time is spent in pushing and popping the necessary items from the stack, which consumes more time. Also, the efficiency of a recursive function depends on depth of recursion i.e. the number of times the function is called recursively to achieve a desired task. When number of calls increases, we may sometimes run out of memory and the process may be terminated abruptly. Recursion is usually used because it simplifies a problem conceptually, not because it's inherently more efficient.

In spite of the above limitations, recursion offers some advantages.

- Recursive functions are clearer and simpler to understand than iterative functions.
- Recursive functions can be easily implemented from recursive definitions.
- Bookkeeping activities like initialization and looping can be avoided in recursive functions.
- Solution to some problems can be easily implemented using recursion and are efficient.

7.5 SUMMARY

- A recursive function calls itself repeatedly, with a different argument value each time.
- Some value of its arguments causes a recursive function to return without calling itself. This is called the *base case*.
- When the innermost instance of a recursive function returns, the process "unwinds" by completing pending instances of the function, going from the latest back to the original call.
- A binary search can be carried out recursively by checking which half of a sorted range the search key is in, and then doing the same thing with that half.
- The Towers of Hanoi puzzle consists of three towers and an arbitrary number of rings.
- The Towers of Hanoi puzzle can be solved recursively by moving all but the bottom disk of a subtree to an intermediate tower, moving the bottom disk to the destination tower, and finally moving the remaining subtree to the destination.
- Any operation that can be carried out with recursion can be carried out with a stack.
- A recursive approach might be inefficient. If so, it can sometimes be replaced with a simple loop or a stack-based approach.

7.6 KEYWORDS

(1). Recursion

(2). Factorial of a number

(3). Tower of Hanoi

(4). Recursive function call

7.7 QUESTIONS

(1). what is recursion? Explain with an example.

(2). What is the difference between iterative procedure and recursion?

(3). Mention the types of recursion and explain each of them using a suitable

7.8 EXERCISE

(1). With a neat recursion tree diagram explain the tower of Hanoi problem for 3 discs.

(2). Write an iterative algorithm to generate Fibonacci series using both iterative and recursive functions. Compare which is the best and justify your answer.

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

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

UNIT- 8

QUEUE

Structure

8.0 OBJECTIVES

After reading this unit you should be able to

- Explain the basics of queue data structure
- Discuss the basic operations on queue
- List out the limitations of linear queue
- Elucidate circular queue and its basic operations
- Explain about priority queue

8.1 INTRODUCTION

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 a data structure that is similar to a stack, except that in a queue the first item inserted is the first to be removed (FIFO). In a stack, as we have seen, the last item 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 is the first person to reach the front of the line and buy a ticket. The last person to line up is the last person to buy a ticket. Figure 8.1 shows how this looks. Queues are used as a programmer's tool similar to stacks. They are also used to model real world situations such as people waiting in line at a bank, airplanes waiting to take off, or data packets waiting to be transmitted over the Internet.

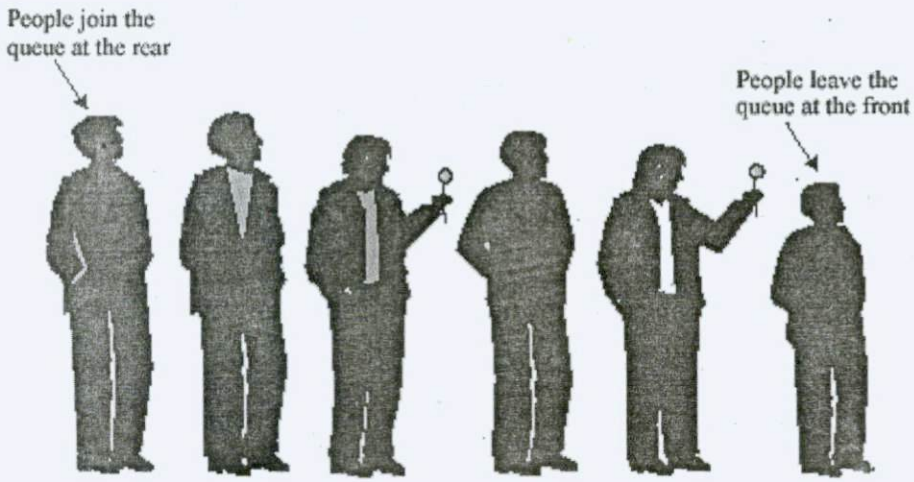


Figure 8.1 A queue of people.

8.2 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:

- sempty(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.

8.3 IMPLEMENTATION OF QUEUES

Like Stacks, Queues can also be realized using an array structures, which is sequential and static. An alternative way to realize a stack is to use linked lists, which are non-sequential and dynamic. The following sections describe the implementation of queues using arrays.

8.4 ARRAY IMPLEMENTATION OF A QUEUE

As discussed for stacks, an array of *n* locations can be defined for realizing a static Queue. The Figure 8.2 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 8.2, initially, the two pointers *front* and *rear* are initialized to -1.

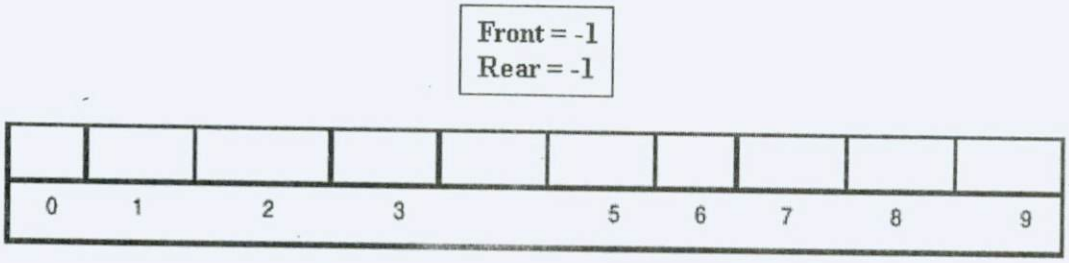


Figure 8.2 An empty Queue of size 10 elements.

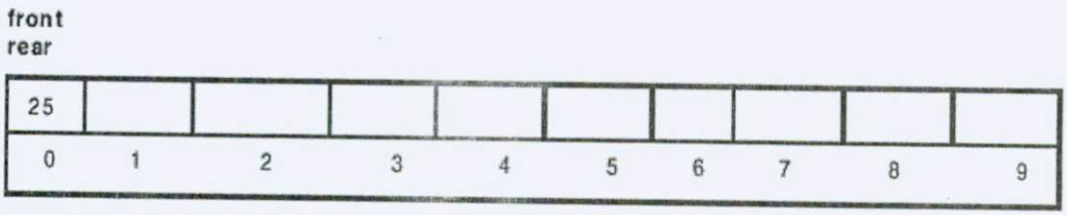


Figure 8.3 Queue with an insertion of an element 25.

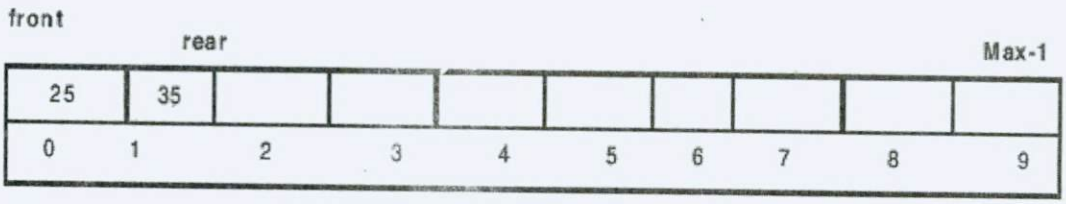


Figure 8.4 Queue with an insertion of two elements 25 and 35.

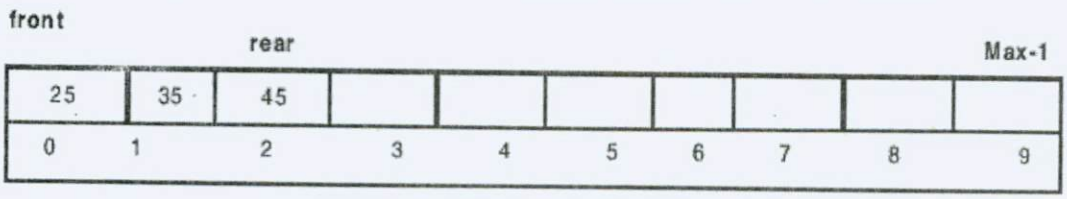


Figure 8.5 Queue with an insertion of three elements 25, 35 and 45.

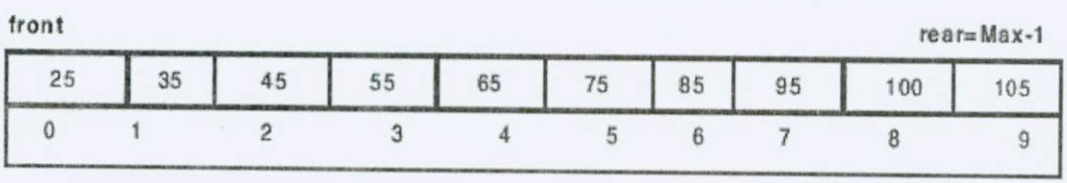


Figure 8.6 Queue is full with insertion of 10 elements.

Look at the Fig. 8.2 above, initially the queue is empty. The condition ***front*** = **-1** and ***rear*** = **-1** initially implies that queue is empty. In Fig. 8.3, element 25 is inserted as first element and both *front* and *rear* are assigned to first position. Fig. 8.4 and 8.5 show the position of *rear* and *front* with each addition to queue. Note that in Fig. 8.3, when first element is inserted, we have incremented *front* by 1, after that

front is never changed during addition to queue process. Observe that *rear* is incremented by one prior to adding an element to the queue; we call this process as ***enque.*** However to ensure that we do not exceed the Max length of the queue, prior to incrementing rear we have to check if the queue is full. This can be easily 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. 8.7 shows status of *front* and *rear* pointers after two deletions from Fig. 8.5.

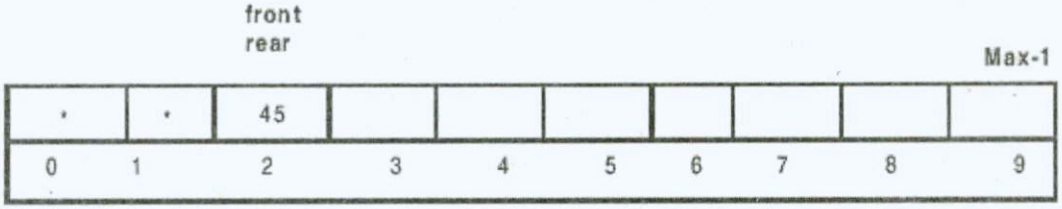


Figure 8.7 Queue after deleting two elements 25 and 35 from Figure 3.5.

Observe that the positions vacated by front marked as * are unusable by queue any further. This is one of the major draw back of linear queue array representation. In Fig. 8.8, we have carried out three deletions for 25, 35, and 45. After deleting 45, we find queue is empty and the condition for ***this queue state is rear is less than front.*** At this stage, we can reinitialize the queue by setting ***front*** = **-1** and ***rear*** = **-1** and thus reclaim the unused positions marked with *.

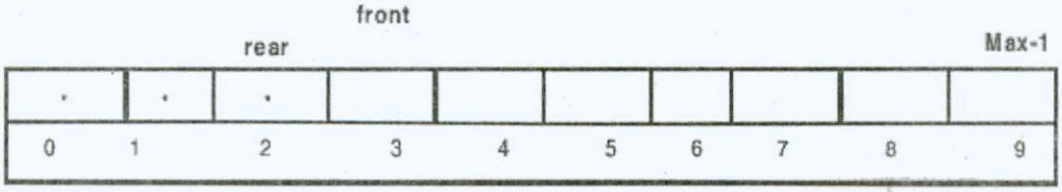


Figure 8.8 Queue after deleting three elements 25, 35 and 45 from Figure 3.5.

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

8.5 IMPLEMENTATION OF QUEUE OPERATIONS

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.

8.6 CIRCULAR QUEUE AND ITS IMPLEMENTATION

The queue structure discussed so far in the above section is generally called as linear queue. This type of queue suffers from one major drawback i.e. when the first element is serviced; the front is moved to next element. However, the position vacated is not available for further use. Thus, we may encounter a situation, wherein the status shows that queue is full even though the queue is physically empty. This situation is shown in fig 8.9.

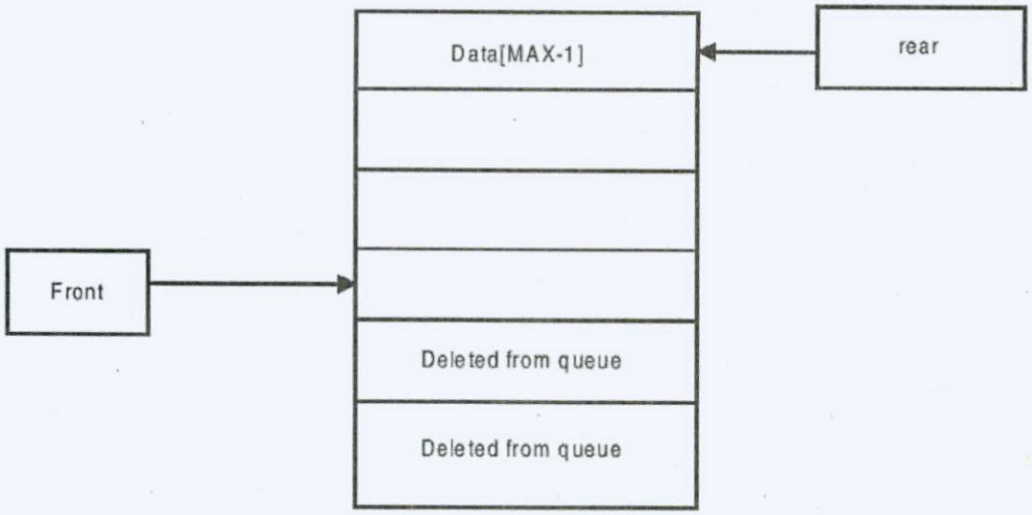


Figure 8.9 Drawback of a linear queue.

In order to overcome the above drawback, we can modify the linear structure in such a way that when the rear pointer has reached the end of the queue, it can point back to the initial position if there are any free locations. A linear queue with such a modification in its operation is called a *circular queue*.

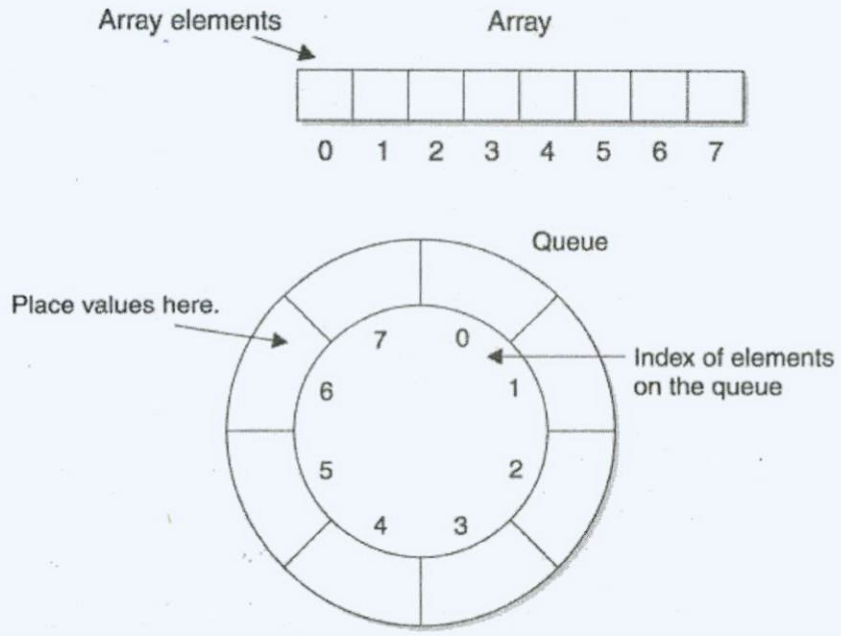


Figure 8.10 An array structure organized as a circular queue.

Figure 8.10 shows an empty conceptual circular queue with a capacity of 8 elements. Figure 8.11a and 8.11b shows status of a queue after inserting elements 55 and 90. The *front* and *rear* pointers are pointing to the positions 0 and 1 respectively.

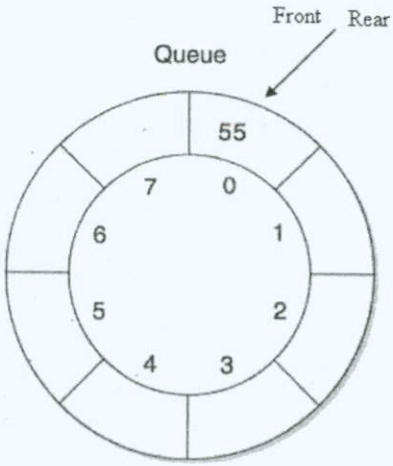


Figure 8.11a Queue with element 55.

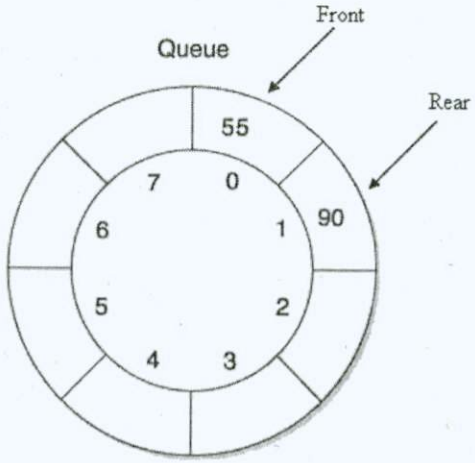


Figure 8.11b Queue with elements 55, 90

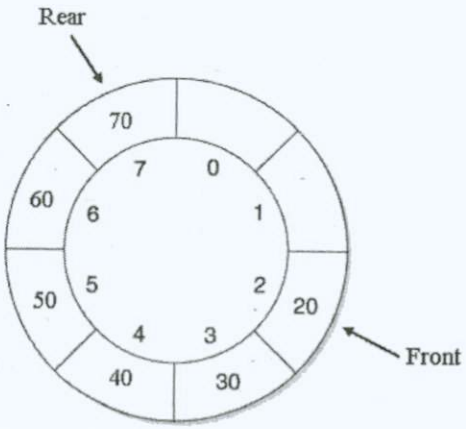


Figure 8.12a. Queue with elements 55, 90 deleted and elements 20, 30, 40, 50, 60 and 70 are inserted to an instance shown in Fig. 3.11b

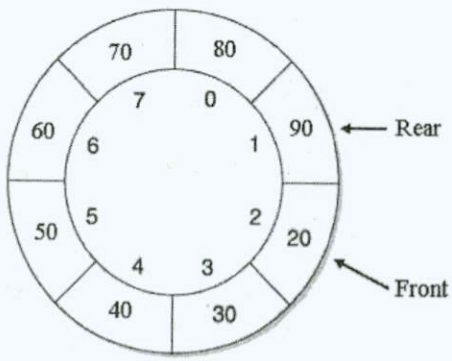


Figure 8.12b. Queue after inserting elements 80 and 90 to an instance shown in Fig. 3.12a.

Figure 8.12a shows an instance of a circular queue, when elements 55 and 90 are deleted from and elements 20, 30, 40, 50, 60 and 70 inserted into an instance shown in

Figure 8.11b. Similarly, Figure 4.12b shows an instance when elements 80 and 90 are inserted into an instance of a queue shown in Figure 8.12a. It is understood from the Figure 8.12b that in a circular queue, it is possible to insert elements to a queue even if the rear pointer has reached the Max position provided if the number of elements is less than the maximum capacity of a queue. Thus the drawback of linear queue structure is eliminated using circular queue structure.

All the operations that we have discussed with linear queue can be extended for circular queue with little modifications as follows. Operations Create-Queue(), Isempty(), Rear(), and Front() as same as linear queue.

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.

8.7 DOUBLE ENDED QUEUES (DEQUEUE)

So far we have discussed that insertion and deletion operations are performed at two different ends in a linear queue structures. But some applications require that both insertion and deletion operations need to be performed at both the ends. That means both insertion and deletion can be performed at front as well as at rear end of the general queue. A modified queue structure, which supports insertion and deletion operations to be performed at both ends of a queue is called a *double-ended queue* or *Dequeue*. Further, Dequeues are classified as insertion-restricted dequeues - when insertion is permitted to take place at only one end where as deletion can be performed at both the ends or deletion-restricted dequeues – when deletion is permitted to take place at only one end where as insertion can be performed at both the ends.

Insert at the front end

In a general queue, insertion is performed at rear end by incrementing the rear pointer. Since dequeue supports insertion and deletion operation to take place at both the ends, we can perform insertion at front end of the queue. To deal with this operation, we need to consider two cases as follows:

Case 1: When queue is empty: In this case, both *front* and *rear* pointers are assigned a value -1 indicating queue empty status. We can insert an item by incrementing rear pointer by 1 as usual and updating both queue pointers as shown in Figure 4.13.

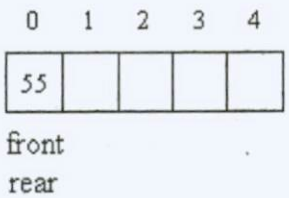
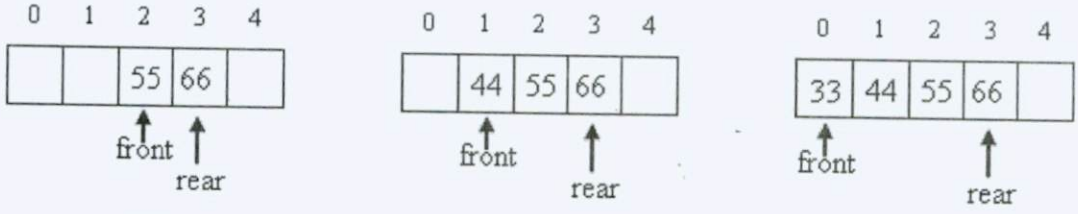


Figure 4.13 After inserting 55

Case 2: When queue is not empty: In this case, both front and rear pointers are assigned values depending on the positions they are pointing to in the queue. We can insert an item at front by decrementing front pointer by 1 as shown in Figure 4.14.



(a) (b) (c)

Fig. 8.14 (a) Before insertion (b)After inserting 44 (c) After inserting 33

Delete from the rear end

In a general queue, deletion is performed at front end by incrementing the front pointer. Since dequeue supports insertion and deletion operation to take place at both the ends, we can perform deletion from rear end of the queue as shown in Figure in 4.15.

(a) (b) (c)

Fig. 8.15 (a) Before deletion (b)After deleting 66 (c) After deleting 55

Figure 8.15 above shows instances of a dequeue when elements are deleted from the rear end. Observe that the rear pointer is decremented by 1 each time an element is deleted. When all elements are deleted, rear pointer will point to -1 and front still pointing to 0 reaching a condition rear is less than front indicating an empty queue and hence both pointers are initialized to -1 before performing further operations on the queue. Algorithms for implementing the above operations on dequeue can be designed using the above procedures.

8.8 PRIORITY QUEUES

For some applications, it is necessary to organize data items in such a way that elements are inserted or deleted based on some priority. A data structure, which supports such a priority based insertions and deletions is often called as *priority queues*. In this data structure, always an element with highest is processed before processing any of the lower priority elements. If elements in the queue are of same priority, then the elements are processed in FCFS basis. Priority queues are used in implementing job scheduling algorithms used in operating systems, where the jobs with highest priorities have to be processed first. Priority queues are classified as either *Ascending priority queue* or *Descending priority queue*.

In an ascending priority queue, elements are inserted in any order but the element with least priority is deleted first. Similarly, in a descending priority queue, elements are inserted in any order but the element with highest priority is deleted first.

There are different ways to implement priority queues. One such technique is inserting an element into a queue such that elements are arranged in ascending order and always delete an element from the front end. Another technique is inserting elements into a queue such that they occupy their respective positions in the queue based on their priority. That means, element with highest priority is placed at 0^{th} position, element with next highest priority is placed at 1^{st} position and so on. Finally, an element with least priority is placed at the end of the queue. If we remove from front, an element with highest priority is deleted.

8.9 SUMMARY

- Queues and priority queues, like stacks, are data structure usually used to simplify certain 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 and removing the item from the front of the queue.
- A queue can be implemented as a circular queue, which is based on an array in which 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 and removing 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.

8.10 KEYWORDS

(1). Queue

(2). Linear queue

(3). Circular queue

(4). Doubly ended queue

(5). Priority queue

8.11 QUESTIONS

(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.

8.12 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-9

LINKED LISTS, SOME GENERAL LINKED LIST OPERATIONS

Structure:

9.0 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.

9.1 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 of general-purpose data-storage applications.

9.2 LINKED 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 3.1(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 3.1(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 3.2 shows an example linked list of integer numbers.

Figure 3.1 (a) Structure of a node (b) An instance of a node

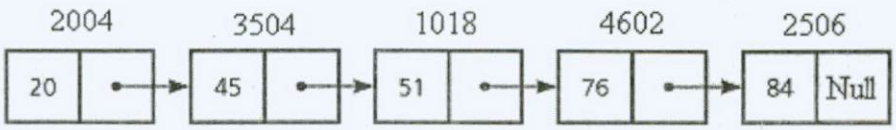


Figure 3.2 Pictorial representation of a linked list.

From the Figure 3.2, 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.

9.3 GENERAL 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.

1. Cr· ·ion 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.

3. 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.

11. 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.

13. 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.

9.4 STATIC 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.	3. 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.

9.5 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.

9.6 KEYWORDS

Linear data structure, Linked lists, Sequential allocation, Dynamic allocation.

9.7 QUESTIONS

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

9.8 REFERENCES

1. Sams Teach Your Self Data Structures and Algorithms in24Hours.
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-10

SINGLY LINKED LISTS AND ITS OPERATIONS

Structure:

10.0 OBJECTIVES

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

- Singly linked lists and their structure.
- Various operations singly linked lists.
- Algorithms to implement these operations.
- Algorithms to implement stacks and queues using linked lists.

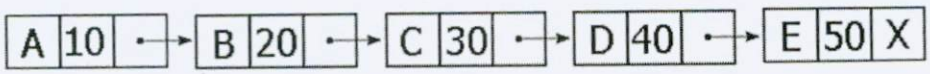
10.1 INTRODUCTION

As discussed earlier in Unit-I, a singly linked list is a collection of nodes, which can be used to effectively implement linear data structures such as stacks and queues. In general, each node in a singly linked list consists of several fields out which one field is

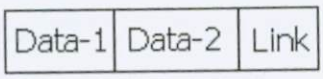
used to hold the address of the next node in a list and all other fields are used to hold the data items of type primitive. In order to implement a singly linked list, we need a mechanism to allocate a memory for each node. A pointer is maintained to a list of freely available memory locations from where we get a memory for creating a node. The memory released when a node is deleted, is added to the available memory list. The following sections describe the various operations performed on singly linked lists and the implementation of stacks and queues using singly linked lists.

10.2 REPRESENTATION OF A SINGLY LINKED LIST

As discussed earlier in Unit-I, a singly linked list is a collection of nodes, which can be used to effectively implement linear data structures such as stacks and queues. In general, each node in a singly linked list consists of several fields out which one field is used to hold the address of the next node in a list and all other fields are used to hold the data items of type primitive. Figure 3.1 shows an example of a singly linked list with three fields. The first two fields hold the data and the third field contains the address of the next node in the list.



(a) Singly linked list



(b) Structure of the node

Fig. 3.1 A singly linked list and its node structure

Every node is a chunk of memory having an address. When a set of data elements to be used by an application are represented using a linked list, each data element is represented by a node. Depending on the information content of the data element, one or more data fields are used in the node. However, in singly linked list only a single link field is used to point to node, which represents its neighbouring element in the list. The last node in the linked list has its link empty. The empty link is normally denoted using a cross mark.

Figure 3.2 shows the physical representation of a linked list where its logical representation is shown in Figure 3.1. Note that the nodes are distributed all over the memory and are not physically contiguous. Also observe that the link field of each node contains the address of the node of its logical neighbour. The link field of the last node is NUL indicated by a cross symbol. In the above example, the node at address 4026 containing the data elements E and 50 is the last node of the linked list. This can be easily understood with the Figure 3.1.

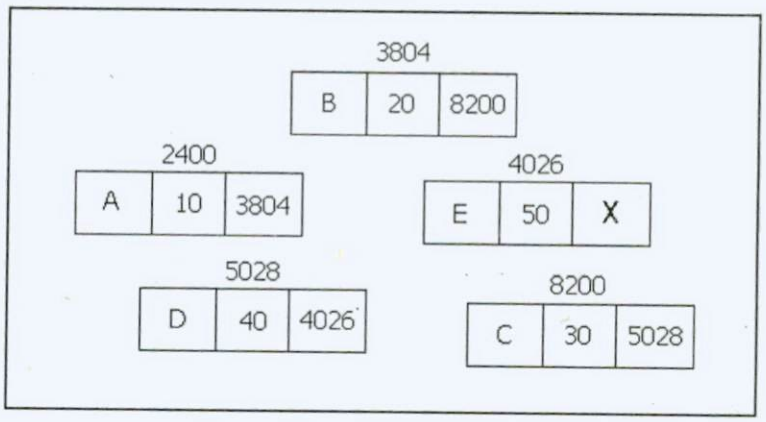


Fig. 3.2 Physical representation of a singly linked list

10.3 OPERATIONS ON SINGLY LINKED LISTS

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:

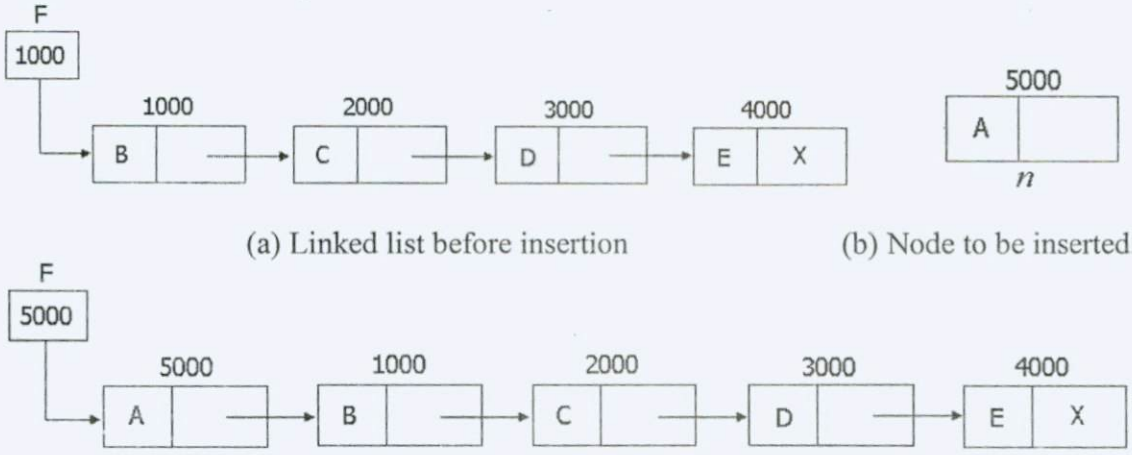
1. n = getnode()
2. n.data = e

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

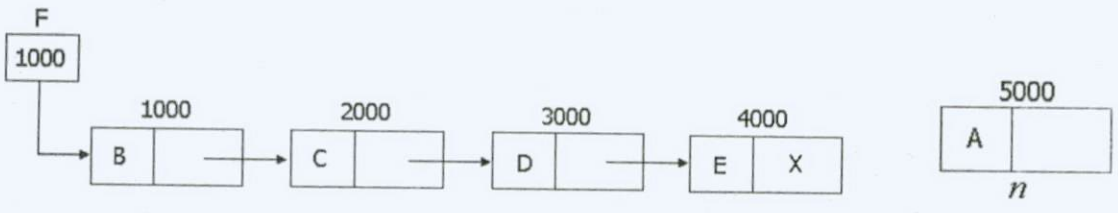
End-of-While

T.Link = n

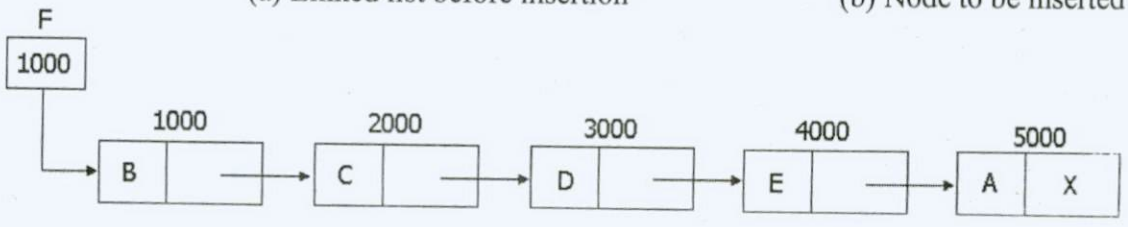
Endif

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

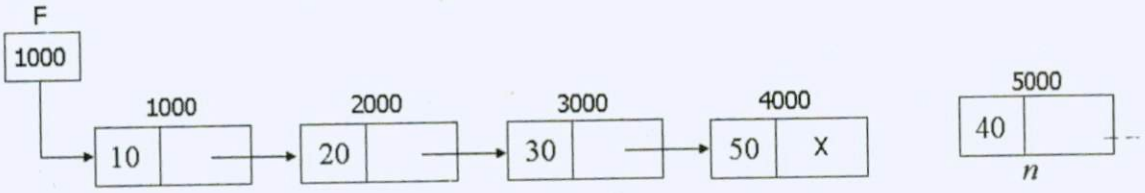
Endif

Endif

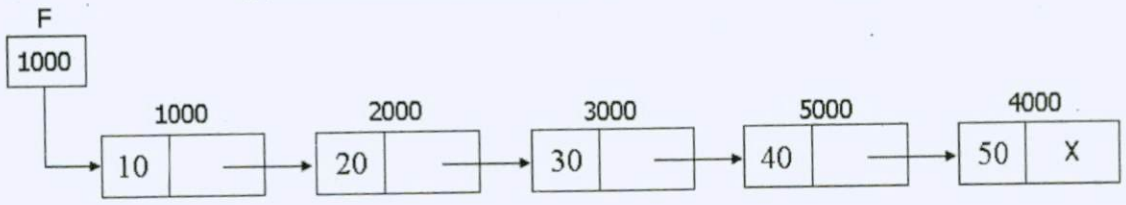
Algorithm ends

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

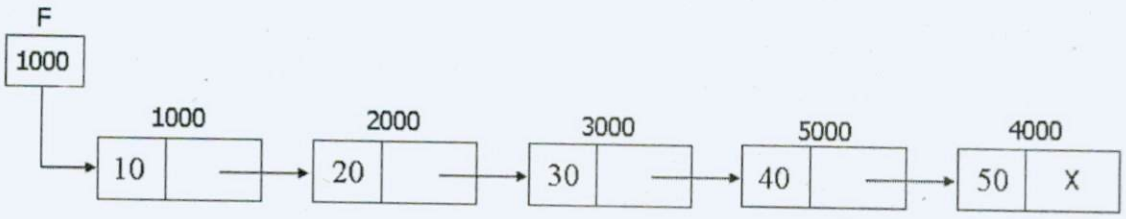
   F = F.link

   Dispose(T)

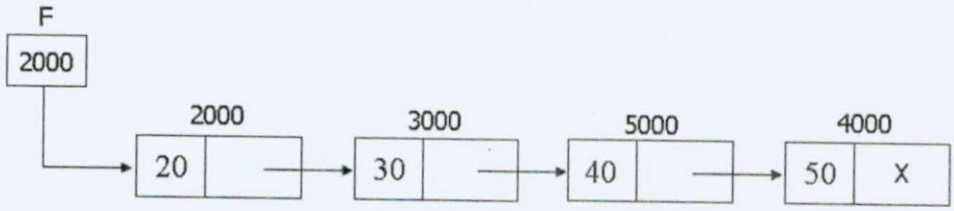
   ifend

Algorithm ends

The above Algorithm is self explanatory. The function Dispose(T) is a logi 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

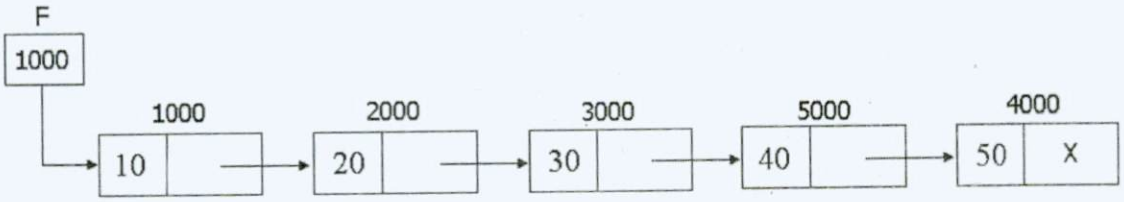
T.link = *n*

endif

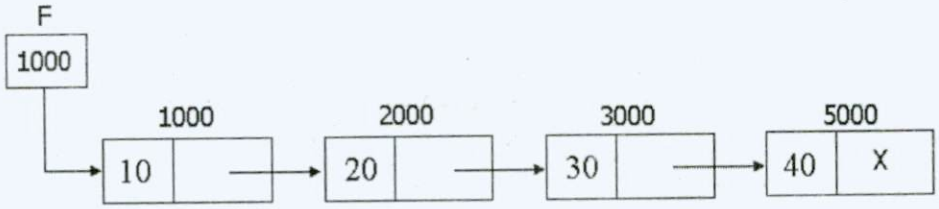
endif

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.

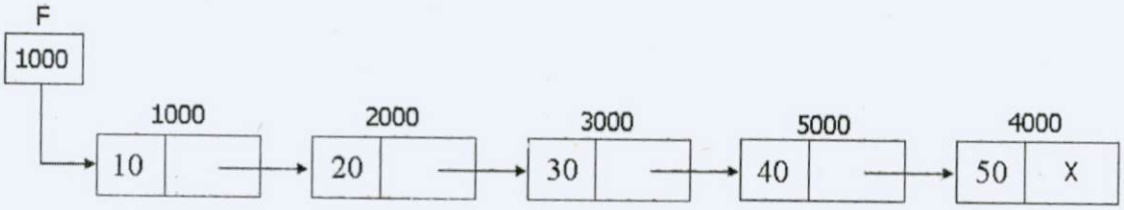
Method:

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

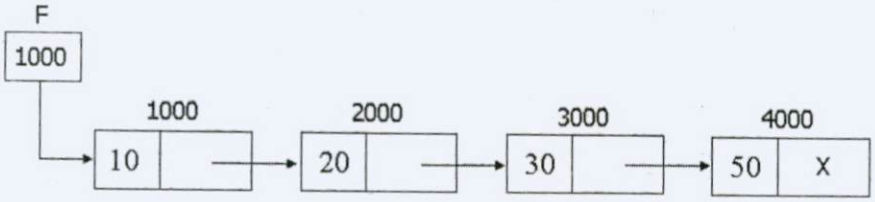
```
        else
            if (F.data = e) then F = F.link
            else
                    T = F
                    While (((T.link).link ≠ Null) and ((T.link).data) ≠ e) Do
                            T = T.Link
                    End-of-While
                    if (T.link = Null) then Display " 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.

The Efficiency of Linked Lists

Insertion and deletion at the beginning of a linked list are very fast. They involve changing only one or two pointers, which takes O(1) time.

Finding or deleting a specified item requires searching through, on the average, half the items in the list. This requires O(N) comparisons. An array is also O(N) for these operations, but the linked list is nevertheless faster because nothing needs to be moved when an item is inserted or removed. The increased efficiency can be significant, especially if a copy takes much longer than a comparison.

Of course, another important advantage of linked lists over arrays is that the linked list uses exactly as much memory as it needs, and can expand to fill all available memory.

The size of an array is fixed when it is created; this usually leads to inefficiency because the array is too large, or to running out of room because the array is too small.

10.5 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().

10.6 KEYWORDS

Linear data structure, Singly Linked lists, Dynamic allocation.

10.7 QUESTIONS

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.

10.8 REFERENCES

1 Sams Teach Your Self Data Structures and Algorithms in24Hours.

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-11

Circular and Doubly Linked Lists

Structure:

11.0 OBJECTIVES

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

- Circular and doubly linked lists.
- Various operations performed on circular and doubly linked lists.
- Algorithms to implement various operations on circular and doubly linked lists.
- Merits and demerits of circular and doubly linked lists.

11.1 INTRODUCTION

In Unit-II, we have studied singly linear linked list. We understood from the discussion that when we reach the end of the list, we cannot come back to the beginning unless the address of the first node is preserved using a temporary pointer variable. This drawback of singly linked list can be eliminated if we make the link field of the last node

to point to the beginning of the list. A linked list organized in such a way that the last node is pointing to the first node of the list is called a *circularly linked list*.

A circularly linked list may be a natural option to represent arrays that are naturally circular, e.g. the corners of a polygon, a pool of buffers that are used and released in FIFO order, or a set of processes that should be time-shared in round-robin order. In these applications, a pointer to any node serves as a handle to the whole list. With a circular list, a pointer to the last node gives easy access also to the first node, by following one link. Thus, in applications that require access to both ends of the list (e.g., in the implementation of a queue), a circular structure allows one to handle the structure by a single pointer, instead of two.

For some applications, it may be necessary to traverse a linked list in both the directions. i.e. from beginning of the list to the end of the list and from end of the list to the beginning of the list. In order to accomplish this task, we need to maintain two singly linked lists with two pointers; one traverses the list from left to right and another from right to left. The problem of traversing a linked list in both the direction can be effectively handled if we incorporate two links into a node structure, which we have considered for singly linked list. This type of data structure enhances the flexibility of data movement. A linked representation that includes two links in every node, each of which points to the nodes on either side of the given node is as known doubly linked list which is discussed in the subsequent sections.

3.2 CIRCULARLY LINKED LIST AND ITS REPRESENTATION

Definition: A linear linked list organized in such a way that link field of the last node contains the address of the first node is called as a ***circularly linked list***. Figure 3.1 illustrates the representation of a circularly linked list.

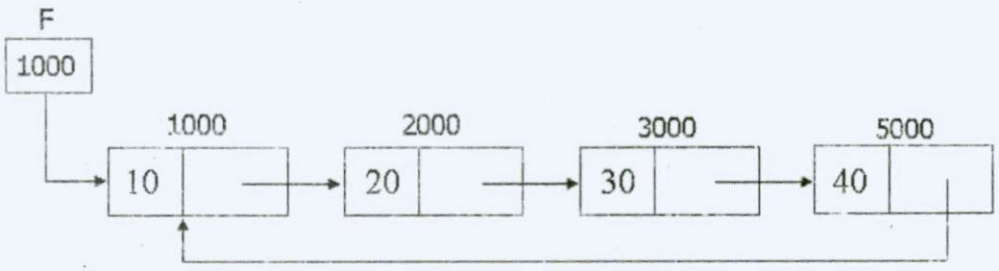
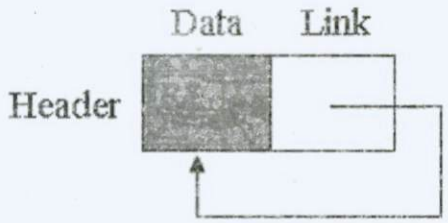
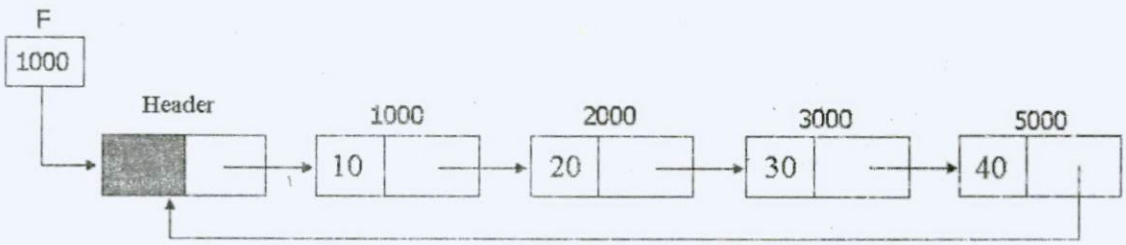


Figure 3.2 Logical representation of a circularly linked list

From the above representation of circularly linked list, it is understood that during processing one has to make sure that one does not get into an infinite loop owing to the circular nature of pointers in the list. A solution to this problem is to designate a special node to act as the head of the list. This node is usually referred to as ***header node***. This header node has its advantage other than pointing to the beginning of a list. The list can never be empty and represented by a hanging pointer (F = Null) as was the case with empty singly linked lists. A circular linked list becomes empty when head points to the head node of the list i.e. (Head.link = Head). A circular linked list with an header node is called a ***headed circularly linked list***. Figure 3.3 shows an empty headed circularly linked list. Figure 3.4 shows the logical representation of a headed circularly linked list.



3.3 An empty circularly linked list



3.4 Logical representation of a headed circularly linked list

Observe that the header node has the same structure as the other nodes in the list. The data field of the header node is unused and is indicated as a shaded field in the pictorial

representation. However, in practical applications these fields may be utilized to represent any useful information about the list relevant to the application.

11.3 OPERATIONS ON CIRCULARLY LINKED LISTS

Let us understand the various primitive operations that can be performed on circularly linked lists.

1. Inserting an element at the beginning of the circularly linked list.

To insert an element at the beginning of the circularly linked list, we need to know the address of the header node and an element to be inserted. The algorithm below is used to implement this operation. Figure 3.5 shows the logical representation of this operation.

Algorithm: Insert-First-CLL

Input: H, address of the header node.

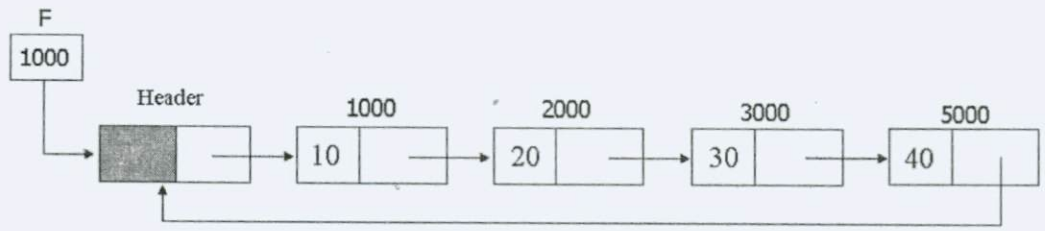
e, element to be inserted.

Output: H, updated.

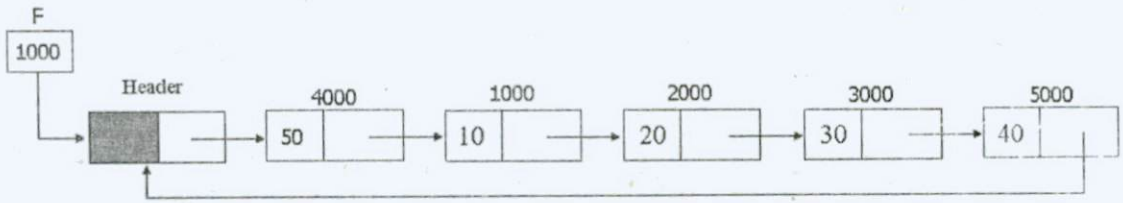
Method:

1. n = Getnode()
2. n.data = e
3. n.link = H.link
4. H.link = n

Algorithm ends



(a) Before insertion



(b) After insertion

Figure 3.5 Pictorial representation of **Insert-First-CLL** operation

2. Inserting an element at the end of the circularly linked list.

To insert an element at the end of the circularly linked list, we need to know the address of the header node and an element to be inserted. The algorithm below is used to implement this operation. Figure 3.6 shows the pictorial representation of this operation.

Algorithm: Insert-Last-CLL

Input: H, address of the header node.

e, element to be inserted.

Output: H, updated.

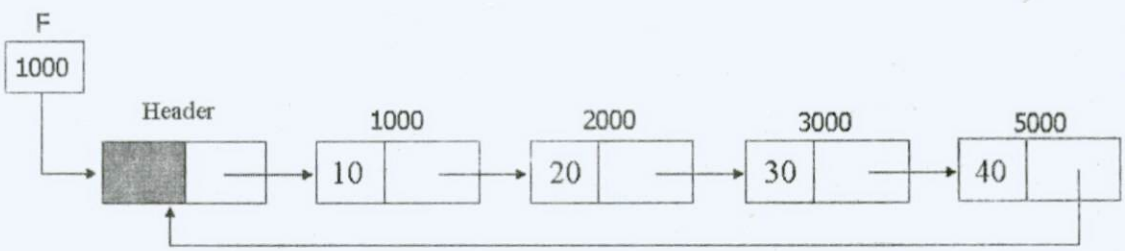
Method:

1. n = Getnode()
2. n.data = e
3. n.link = H
4. T = H
5. While (T.link ≠ H) DO

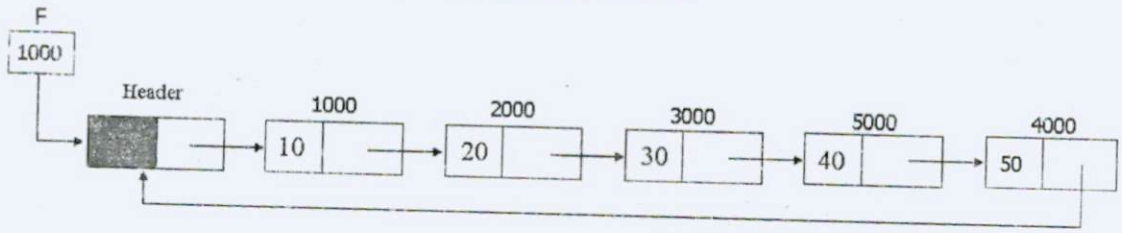
   T = T.link

   End-of-while
6. T.link = n

Algorithm ends



(a) Before insertion



(a) After insertion

Figure 3.6 Pictorial representation of **Insert-Last-CLL** operation

We can observe from the above figure that inserting a node (element) at end of the list is very simple. First, we have to traverse the iist till we reach the end of the list. The last node of the list is the one whose link field contains the address of the header node as shown in the Figure 3.6(a). Once we find this node, its link field is replaced with the address of the new node to be inserted and the address of the header node is copied to the link field of the inserted node.

3. Deleting an element at the beginning of the circularly linked list.

To delete an element at the beginning of the circularly linked list, we need to know the address of the header node. Deletion is done by just replacing the link field of the header node with address contained in the link field of the node to be deleted. The algorithm below is used to implement this operation and is self explanatory. Figure 3.7 shows the pictorial representation of this operation.

Algorithm: Delete-First-CLL

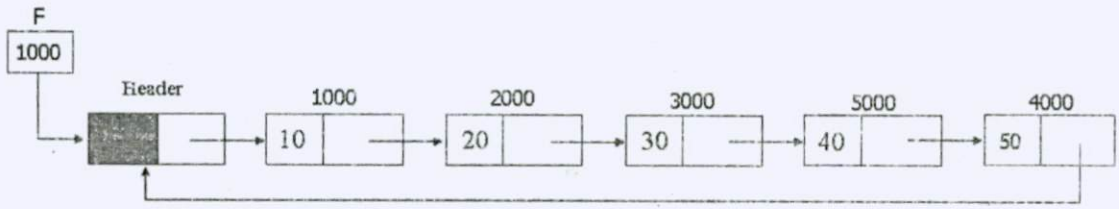
Input: H, address of the header node.

Output: H, updated.

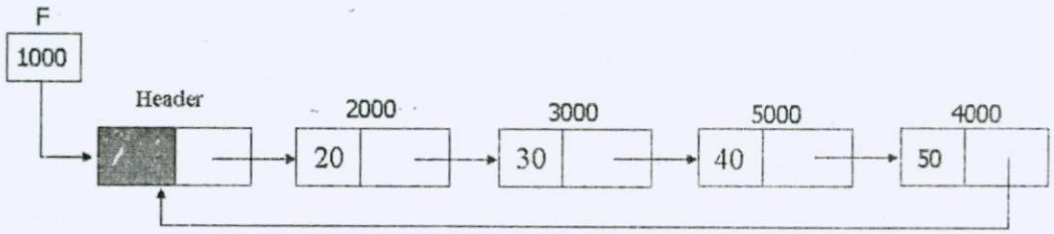
Method:

1. If (H.link = Null) then Display "List is empty"
   else
   H.link = (H.link).link
   endif

Algorithm ends



(a) Before deletion



(a) After deletion

Figure 3.7 Pictorial representation of **Delete-First-CLL** operation

4. Deleting an element at the end of the circularly linked list.

To delete an element at the end of the circularly linked list, first, we have to traverse the list till we reach the node previous to the last node of the list. Deletion is done by just replacing the link field of the last but one node with the address of the header node. The algorithm below is used to implement this operation and is self explanatory. Figure 3.8 shows the pictorial representation of this operation.

Algorithm: Delete-Last-CLL

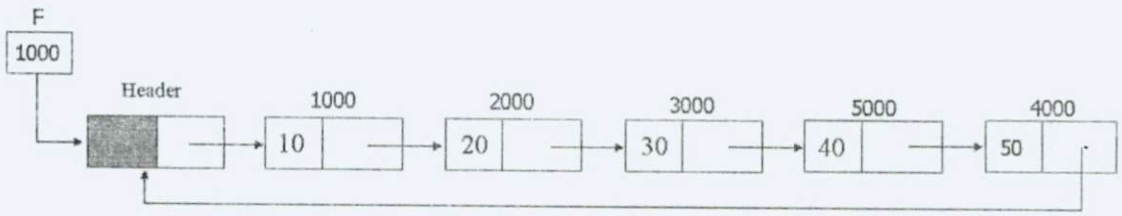
Input: H, address of the header node.

Output: H, updated.

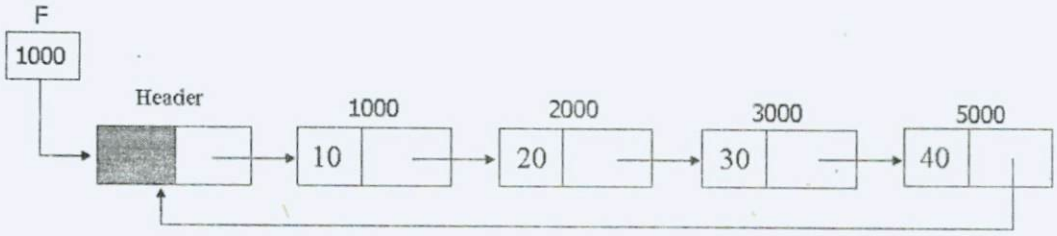
Method:

1. If (H.link = Null) then Display "List is empty"
   else
   T = H
   While ((T.link).link ≠ H) DO
   T = T.link
   End-of-while
2. T.link = H

Algorithm ends



(a) Before deletion



(a) After deletion

Figure 3.8 Pictorial representation of **Delete-Last-CLL** operation

4. Splitting circularly linked list into two lists

We can split the given circularly linked list into two lists such that a node containing the element *e* becomes the first node of the second list. Let $\mathbf{H_1}$ be the address of the header node of the list. To accomplish this task, we have to traverse the list to find the node containing the element *e*. Once found, the link field of the node previous to this node is replaced with the address of the header node $\mathbf{H_1}$. The address of the node containing the element *e* is copied to another header node say $\mathbf{H_2}$ and this list is traversed till we reach the end i.e. the node whose link field contains the address of the header node $\mathbf{H_1}$. Now, the link field of this node is replaced with the address of the header node $\mathbf{H_2}$. Thus we get the two lists with header nodes $\mathbf{H_1}$ and $\mathbf{H_2}$. The following algorithm is used to implement this operation and the Figure 3.9 shows the pictorial representation of this operation.

Algorithm: Split-CLL

Input: H, address of the header node and element *e*.

Output: H_1, H_2, headers.

Method: 1. H_2 = Get-Header()

2. T = H

3. While (((T.link).link ≠ H) and ((T.link).data ≠ *e*)) DO

T = T.link

End-of-while

4. if (T.link ≠ H) then

H_2.link = T.link

T.link = H

T = H_2

While (((T.link).link ≠ H) DO

T = T.link

End-of-while

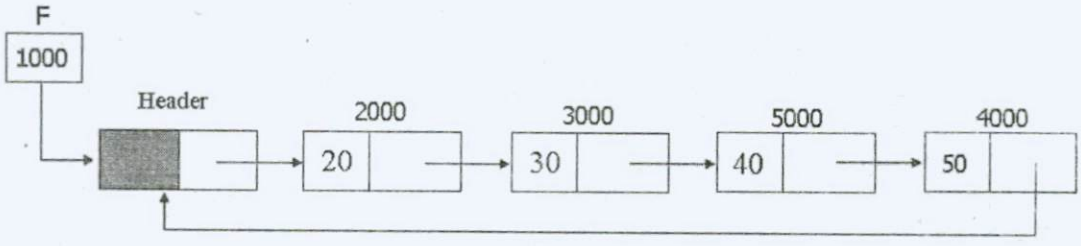
T.link = H_2 ; H_1 = H

Else

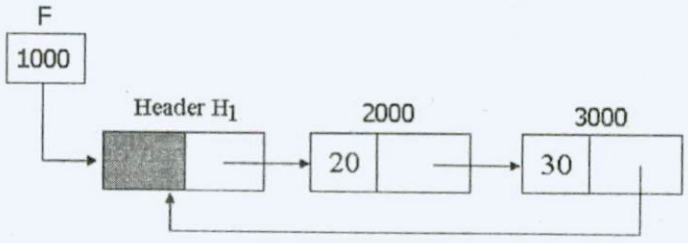
H_2.link = H_2

endif

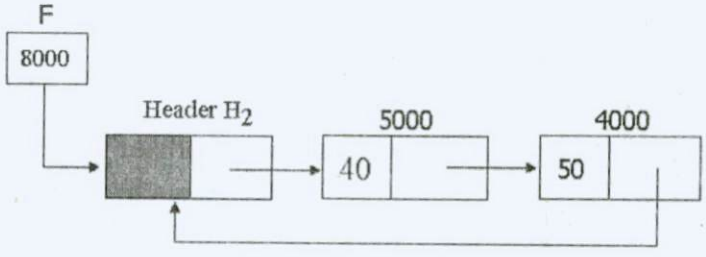
Algorithm ends



(a) Before splitting



(a) List-1 with header H_1



(a) List-2 with header H_2

Figure 3.8 Pictorial representation of **Split-CLL** operation

4. Combining two circularly linked list into a single list

We can combine the given two circularly linked lists in to a single list. This operation is very simple and straight forward when compared to splitting operation. Let $\mathbf{H_1}$ be the address of the header node of the first list and $\mathbf{H_2}$ be the address of the header node of the second list. To accomplish this task, we have to traverse the first list to find the last node of the list i.e. the node whose link field contains the address of the header node $\mathbf{H_1}$. Once found, the link field of this node is replaced with the address of the header node $\mathbf{H_2}$. Then we have to traverse the second list to find the last node of the list i.e. the node whose link field contains the address of the header node $\mathbf{H_2}$. Once found, the link field of this node is replaced with the address of the header node $\mathbf{H_1}$. Now the header node $\mathbf{H_2}$ is no longer required and is disposed. The following algorithm is used to implement this operation.

Algorithm: Combine-CLL

Input: H_1 and H_2, address of the header nodes.

Output: H_1, Header.

Method: 1. $T = H_1$

2. While ($T.link \neq H_1$) DO

   $T = T.link$

   End-of-while

3. $T.link = H_2.link$

4. While ($T.link \neq H_2$) DO

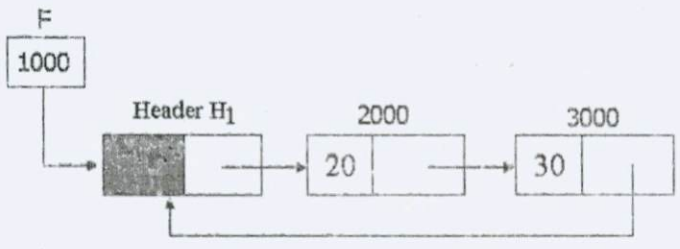
   $T = T.link$

   End-of-while

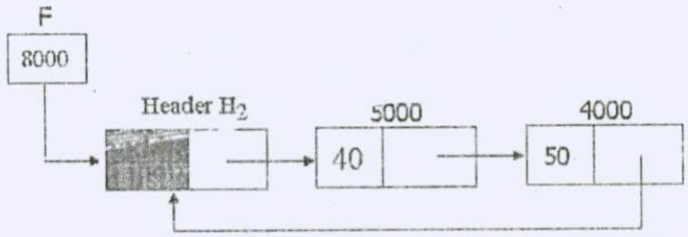
5. $T.link = H_1$

6. Dispose(H_2)

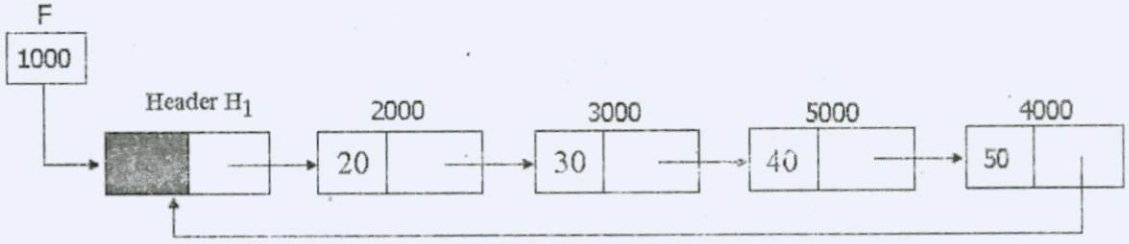
Algorithm ends



(a) List-1 with header H_1



(b) List-2 with header H_2



(c) After combining

11.4 DOUBLY LINKED LIST AND ITS REPRESENTATION

Definition: A linear linked list organized in such a way that every node consists of one or more data fields and two link fields that contain references to the previous and to the next node in the sequence. It can be viewed as two singly-linked lists formed from the same data items, in two opposite orders.

The two links allow walking along the list in either direction with equal ease. Compared to a singly-linked list, modifying a doubly-linked list usually requires changing more pointers, but is sometimes simpler because there is no need to keep track of the address of the previous node. The link fields are often called **forward** and **backwards**, or **next** and **previous**. A pointer to any node of a doubly-linked list gives access to the whole list. Figure 3.10 shows the pictorial representation of a doubly linked list and Figure 3.11 shows the pictorial representation of node structure in doubly linked list

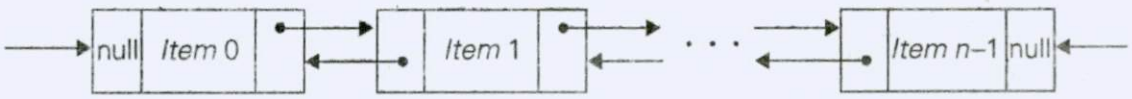


Figure 3.10 Pictorial representation of a doubly linked list.

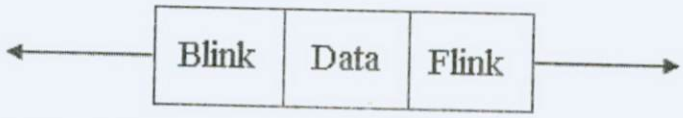


Figure 3.11 Pictorial representation of node structure in doubly linked list.

11.5 OPERATIONS ON DOUBLY LINKED LISTS

Let us understand the various primitive operations that can be performed on doubly linked lists.

1. Inserting an element at the beginning of the doubly linked list.

To insert an element at the beginning of the doubly linked list, we need to know the address of the first and last node of the list and an element to be inserted. The algorithm below is used to implement this operation. Figure 3.12 shows the pictorial representation of this operation.

Algorithm: Insert-First-DLL

Input: F, R, address of first and last nodes.

e, element to be inserted.

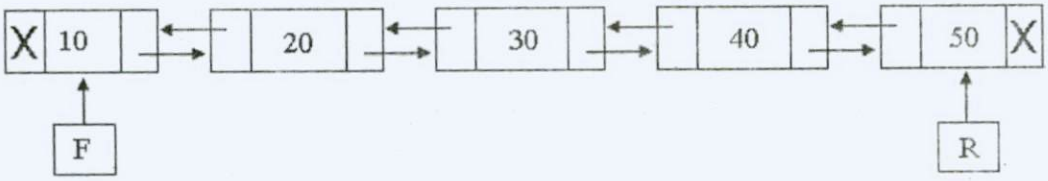
Output: F and R, updated.

Method:

1. n = Getnode()
2. n.data = e
3. n.blink = null
4. n.flink = F
5. If (F = null) then R = n
   Else F.blink = n
   endif
6. F = n

Algorithm ends

(a) List before insertion



(a) List after insertion

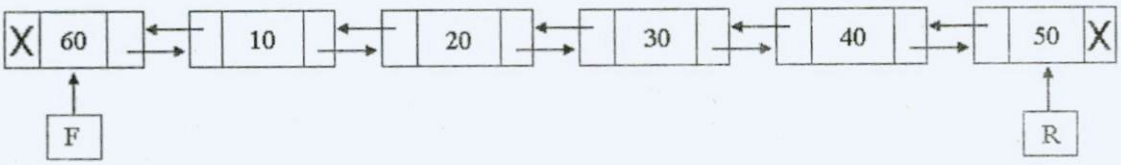


Figure 3.12 The pictorial representation of insertion operation.

From the above pictorial representation, we can understand that inserting an element at the beginning of the doubly linked list requires updation of ***blink*** and ***flink*** pointers of the node to be inserted as well as the node pointed by the **F** pointer. After insertion, the pointer **F** is also updated to point to the inserted node. If the node inserted is the first node then we have to update the pointer **R** as well.

2. Inserting an element at the beginning of the doubly linked list.

To insert an element at the end of the doubly linked list, we need to know the address of the first and last node of the list and an element to be inserted. The algorithm below is used to implement this operation. Figure 3.13 shows the pictorial representation of this operation.

Algorithm: Insert-Last-DLL

Input: F, R, address of first and last nodes

e, element to be inserted.

Output: F and R, updated.

Method:

1. *n* = Getnode()
2. *n*.data = *e*
3. *n*.flink = null
4. *n*.blink = R
5. If (R = null) then F = *n*

Else R.flink = *n*

endif

6. R = *n*

Algorithm ends

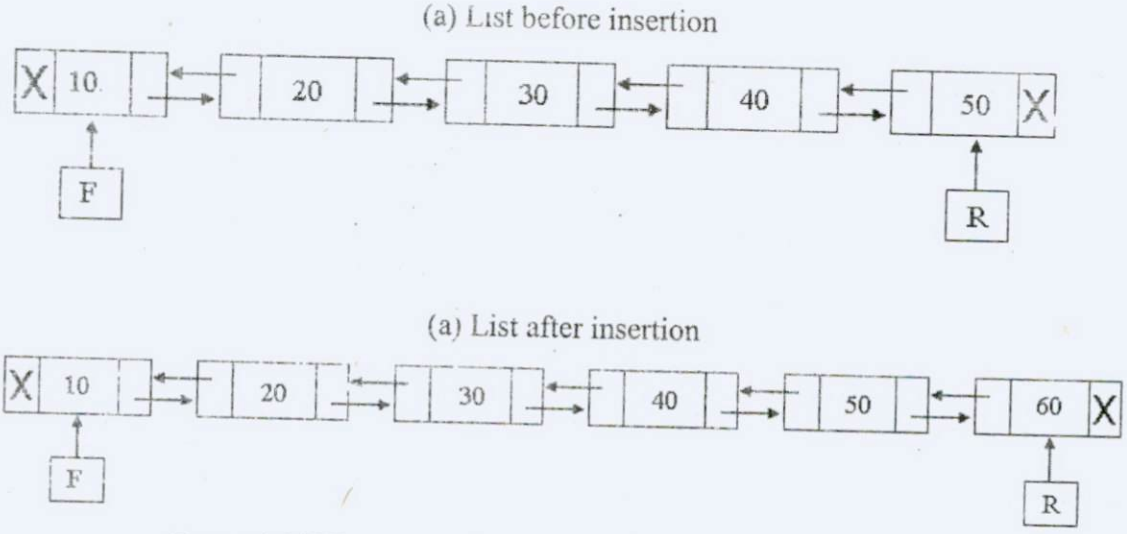


Figure 3.13 The pictorial representation of insertion operation.

From the above pictorial representation, we can understand that inserting an element at the end of the doubly linked list requires updation of ***blink*** and ***flink*** pointers of the node to be inserted as well as the node pointed by the **R** pointer. After insertion, the pointer **R** is also updated to point to the inserted node. If the node inserted is the first node then we have to update the pointer **F** as well.

3. Deleting an element at the beginning of the doubly linked list.

To delete an element at the beginning of the doubly linked list, we need to know the address of the first and last node of the list. The algorithm below is used to implement this operation. Figure 3.14 shows the pictorial representation of this operation.

Algorithm: Delete-First-DLL

Input: F. R, address of first and last nodes

Outpu . F and R, updated.

Meth d:

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

```
Else
    T = F
    F = F.flink
    If (F = null) then R = null
    Else
            F.blink = null
    Endif
    Dispose(T)
Endif
```

Algorithm ends

Deleting the first node from a doubly linked list is a simple task. We need the address of the first (**F**) and the last node (**R**) of the list. If (F=**null**) means the list is empty. Otherwise, **F** is made to point to next node using the statement **F = F.flink** thus deleting the first node. After deletion, if (F=null) then R is also null. That means if there is only one element then after deletion, the list will become empty; otherwise, the node next to the one deleted will become the first node of the list and hence its ***blink*** will become null. That is **F.blink = null**. Finally, the memory occupied by the node is released. The Figure 3.14 shows the pictorial representation of this operation.

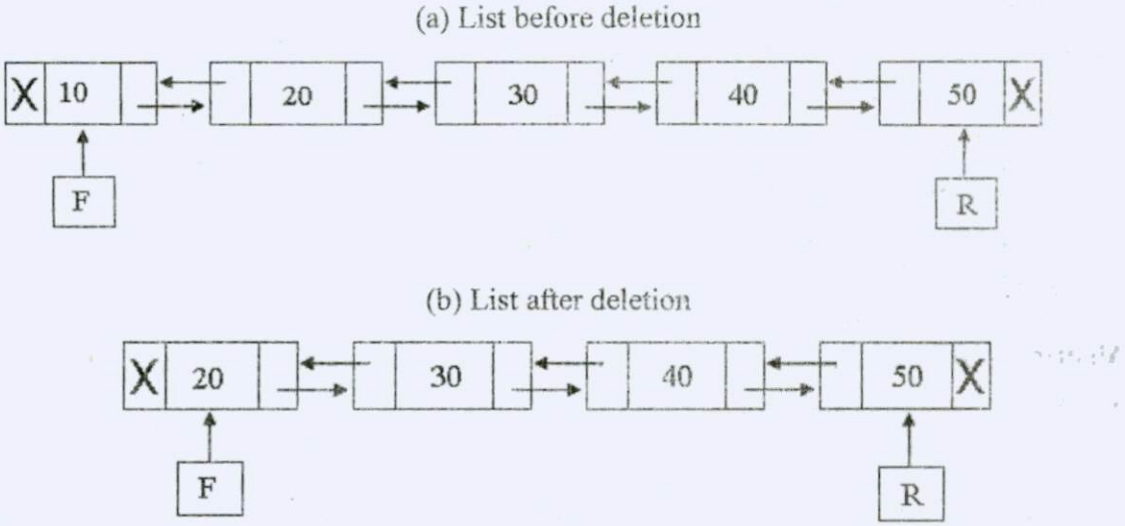


Figure 3.14 The pictorial representation of deletion operation.

4. Deleting an element at the end of the doubly linked list.

To delete an element at the end of the doubly linked list, we need to know the address of the first and last node of the list. The algorithm below is used to implement this operation. Figure 3.15 shows the pictorial representation of this operation.

Algorithm: Delete-Last-DLL

Input: F, R, address of first and last nodes

Output: F and R, updated.

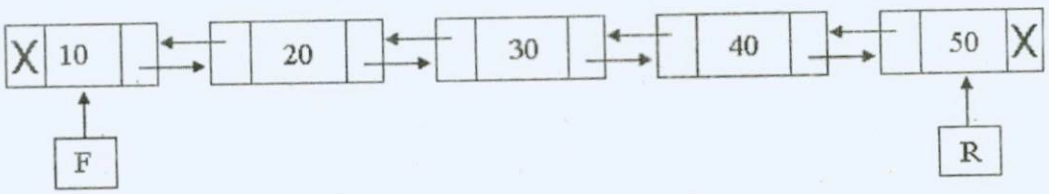
Method:

```
1. If (R=null) then Display "List is empty")
   Else
      T = R
      R = R.blink
      If (R = null) then F = null
      Else
            R.flink = null
      Endif
      Dispose(T)
   Endif
```

Algorithm ends

Deleting the last node from a doubly linked list is a simple task. We need the address of the first (**F**) and the last node (**R**) of the list. If (**R=null**) means the list is empty. Otherwise, **R** is made to point to next node using the statement **R = R.blink** thus deleting the last node. After deletion, **if (R=null) then F** is also made to null. That means if there is only one element then after deletion, the list will become empty; otherwise, the node previous to the one deleted will become the last node of the list and hence its ***flink*** will become null. That is **R.flink = null**. Finally, the memory occupied by the node is released. The Figure 3.15 shows the pictorial representation of this operation.

(a) List before deletion



(b) List after deletion

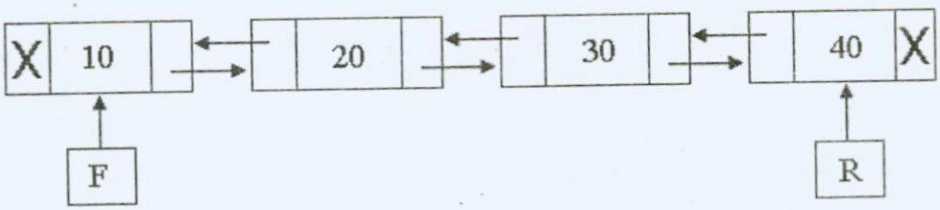


Figure 3.15 The pictorial representation of deletion operation.

5. Inserting an element into a sorted doubly linked list.

To insert an element *e* into a sorted doubly linked list, we need to know the address of the first and last node of the list. If the elements in the linked list are arranged in ascending order, then we have to traverse the list starting with the first node, searching for a node whose element is greater than or equal to the element *e*. If found then the element *e* is inserted as a node predecessor to this node and the link fields of the current node the neighbouring nodes are updated. The algorithm below is used to implement this operation. Figure 3.16 shows the pictorial representation of this operation.

Algorithm: Insert-Sort-DLL

Input: F, R, address of first and last nodes

e, element to be inserted.

Output: F and R, updated.

Method:
1. *n* = Getnode()
2. *n*.data = e
3. if (F=null) then

   n.blink = null

   n.flink = null

   F = *n*

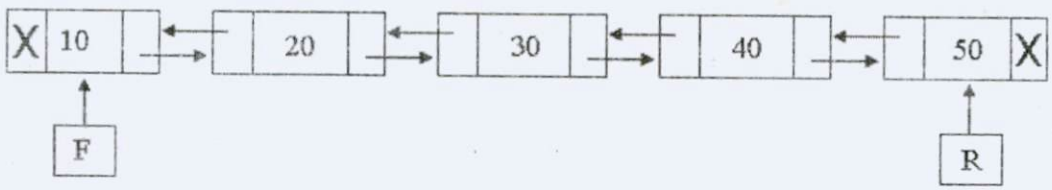
   R = *n*

   else

```
            if (e ≤ F.data) then
                n.flink = F
                n.blink = null
                F.blink = n
                F = n
            else
                T = F
                While ((T.flink ≠ null) and (T.flink).data < e) DO
                        T = T.flink
                End-of-while
                n.flink = T.flink
                n.blink = T
                T.flink = n
                If (T = R) then R = n
                  else (n.flink).blink = n
                endif
            endif
        endif
```

Algorithm ends

(a) List before insertion



(b) List after insertion

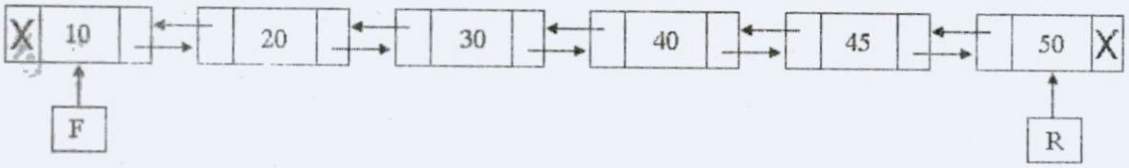


Figure 3.16 The pictorial representation of insertion operation.

6. Deleting an element *e* from a doubly linked list.

To delete an element *e* from a doubly linked list, we need to know the address of the first and last node of the list. We have to traverse the list starting with the first node, searching for a node whose element is equal to the element *e*. To delete the node containing element *e*, we must be at the node predecessor to this node. After deletion the link fields of the neighbouring nodes are updated. The algorithm below is used to implement this operation. Figure 3.17 shows the pictorial representation of this operation.

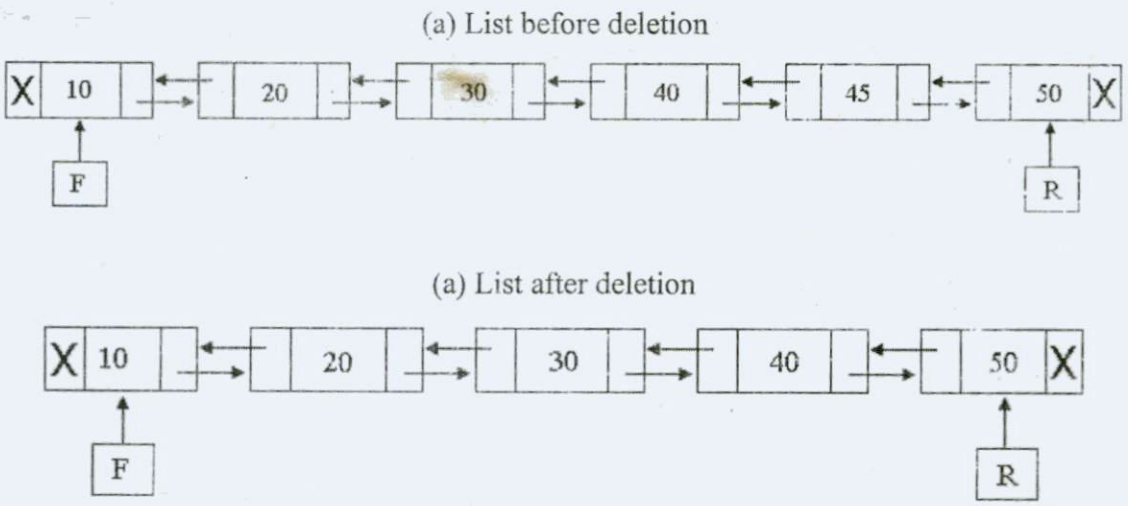


Figure 3.17 The pictorial representation of deletion operation.

Algorithm: Delete-Element-DLL

Input: F, R, address of first and last nodes
e, element to be inserted.

Output: F and R, updated.

Method: 1 if (F=null) then Display "List is empty"
else
if (*e* = F.data) then
F = F.flink
if (F = null) then R = null
else
F.blink = null
endif

```
else
    T = F
    While ((T.flink ≠ null) and (T.flink).data ≠ e) DO
            T = T.flink
      End-of-while
      if (T = R) then Display "Element e is not found"
      else
          T.flink = (T.flink).flink
          if (T.flink = null) then R = T
          else
              (T.flink).blink  = T
          endif
      endif
  endif
endif
```

Algorithm ends

11.6 SUMMARY

- A circularly linked list is an enhancement of the singly linked list representation, in that the nodes are circularly linked. This provides better flexibility in handling delete operation
- The doubly linked list has one or more data item fields but two link fields blink and flink, respectively pointing to the predecessor and successor of the node. Though the list exhibits the advantages of greater flexibility and efficient delete operation, it suffers from the drawback of increased storage requirement and data movement operations when compared to singly linked lists.

11.7 KEYWORDS

Linear data structure, circularly linked lists, doubly linked lists, header node.

11.8 QUESTIONS

1. What are circular linked lists? Explain with an illustrative example.
2. Mention the various operations performed on circularly linked lists.
3. What is an header? Why is it required? Explain with an example.
4. What are the advantages of circularly linked lists?
5. Design an algorithm to delete all the nodes with a given element from a circularly linked list.
6. Design an algorithm to delete alternative occurrence of an element from a circularly linked list..
7. What are doubly linked lists? Explain with an example.
8. What are the advantages and disadvantages of doubly linked lists over singly linked lists?
9. Design an algorithm to split a given doubly linked list into two such that the element ***e*** becomes the first node of the second list.
10. Design an algorithm to merge to sorted doubly linked list in to a single list

11.9 REFERENCES

1. Sams Teach Your Self Data Structures and Algorithms in24Hours.
2. C and Data Structures by Practice- Ramesh, Anand and Gautham.
3. Data Structures and Algorithms: Concepts, Techniques and Applications by GAV Pai. Tata McGraw Hill, New Delhi.

*****

Unit-12

Applications: Polynomial Operations, Dictionary Construction and Sparse Matrix Representation

Structure:

12.0 OBJECTIVES

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

- Representation of a polynomial using linked lists.
- Addition of two polynomials using linked lists.
- Construction of a dictionary using linked lists
- Representation of a sparse matrix using linked lists
- Multiplication of sparse matrix using linked lists.

12.1 INTRODUCTION

In Unit-III, we have studied circularly and doubly linked lists. We discussed how to overcome few drawbacks of the singly linearly linked list using circularly and doubly

linked lists. The linked lists can be used to effectively implement other linear as well as non linear data structures. Also linked lists can be used to perform operations on long positive numbers, manipulation of polynomials, construction of symbol tables, multiplication of sparse matrices etc. In the following sections, we discuss some of these applications of linked lists.

12.2 REPRESENTATION OF A POLYNOMIAL

In order to understand how to represent a polynomial using linked list, let us consider a polynomial of only one variable **X**.

$$P_1: 5X^3 + X^2 + 3X + 6$$

Each term in this polynomial consists of a co-efficient and power of **X**. So, a polynomial can be easily represented using a linked list where each term can be considered as a node consists of three fields as follows:

- **CoefX :** To store the coefficient of **X**.
- **PowX:** To store the power of **X**.
- **Link:** To store the address of the next node in the list.

Figure 4.1 shows the general structure of a node to hold a term of a polynomial and Figure 4.2 shows the pictorial representation of a polynomial using linked list.

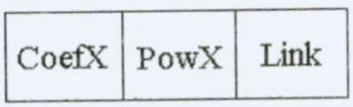


Figure 4.1 Node structure to hold a term

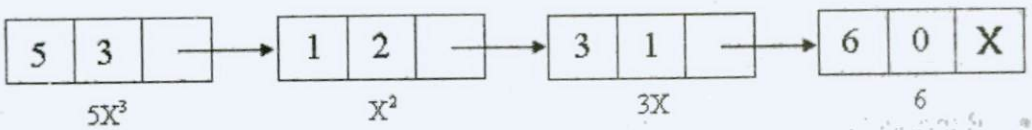


Figure 4.2 Representing a polynomial $5X^3 + X^2 + 3X + 6$ using linked list

Similarly, we can represent a polynomial of two variables or three variables. A node representing a polynomial of two variables consists of four fields out of which one field is for coefficient, one field is for power of X, one field is for power of Y and one field for

link Figure 4.3 shows the general structure of a node to hold a term of a polynomial and Figure 4.4 shows the pictorial representation of a polynomial using linked list.

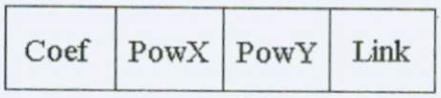


Figure 4.3 Node structure to hold a term

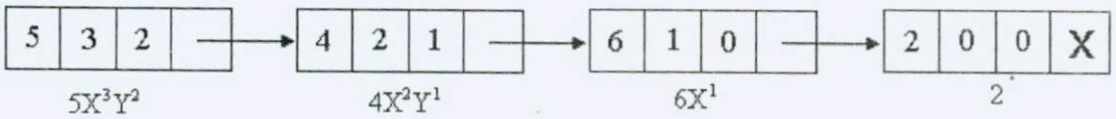


Figure 4.4 Representing a polynomial $5X^3Y^2 + 4X^2Y^1 + 6X^1 + 2$ using linked list

12.3 ADDITION OF POLYNOMIALS

From previous discussion, we understood how to represent a polynomial using linked list. Now let us understand how to perform addition of two polynomials.

Let $\mathbf{P_1: 2x^6 + x^3 + 5x + 4}$ and

$\mathbf{P_2: 7x^6 + 8x^5 - 9x^3 + 10x^2 + 5}$

be two polynomials over a variable **x**. We can add these two polynomials $\mathbf{P_1 + P_2}$ to obtain the resulting polynomial $\mathbf{P_3}$ as,

$$\mathbf{P_3 = P_1 + P_2 = 9x^6 + 8x^5 - 8x^3 + 10x^2 + 5x + 9}$$

To perform addition of two polynomials using linked lists, we need to represent these to polynomials using linked lists as discussed in the earlier section, and then the addition is achieved by adding the coefficients of the nodes of like powers of variable x in lists P_1, P_2 and adding new node reflecting this operation in the resultant list P_3. The algorithm to perform this operation is presented below.

Let H_1, H_2 are the start pointers of the singly linked lists representing polynomials P_1, P_2. Also, T1, T2 are the two temporary pointers initially pointing to P_1 and P_2 respectively.

Algorithm: Polynomial Addition

Input: T1, T2, Start address of the two polynomials P_1 and P_2

Output: The resultant polynomial P_3

Method:

1. If (T_1.PowX = T_2.PowX) then

        If (T_1.Coef + T_2.Coef) ≠ 0) then

            n = Getnode()

            n.Coef = T_1.Coef + T_2.Coef

            n.PowX = T_2.PowX

            n.Link = null

        Endif

    Else

    If (T_1.PowX < T_2.PowX) then

        If (T_1.Coef + T_2.Coef) ≠ 0) then

            n = Getnode()

            n.Coef = T_2.Coef

            n.PowX = T_2.PowX

            n.Link = null

        Endif

    Else

    If (T_1.PowX > T_2.PowX) then

        If (T_1.Coef + T_2.Coef) ≠ 0) then

            n = Getnode()

            n.Coef = T_1.Coef

            n.PowX = T_1.PowX

            n.Link = null

        Endif

    Endif

2. If any one of the lists during the course of addition of terms has exhausted its nodes earlier than the other list, then the nodes of the other list are simply appended to list P_3 in the order of their occurrence in their original list.

Note: 1. Since each term of a polynomial is a node in a singly linked list, we can create a linked list representing a polynomial by repeatedly invoking a function **Insert-Last-SLL()**, which is discussed earlier in unit-II.

2. Similarly, we can design an algorithm to display a polynomial as well as to evaluate a polynomial.

12.5 REPRESENTATION OF SPARSE MATRIX

In this section, we shall discuss about the sparse matrix and its representation using linked lists. First we shall define what is a spare matrix?

Sparse matrix: A sparse matrix is a matrix with zeroes as the dominating elements. That means, a matrix containing more number of zero elements than non zero elements. Figure 4.5 shows an example of a matrix and sparse matrix.

2	4	6	8
1	2	0	4
0	3	5	9
7	0	8	5

(a) Matrix

2	0	0	0
0	0	0	0
0	0	5	0
0	0	8	0

(b) Sparse matrix

Figure 4.5 Matrix and a sparse matrix

A matrix consumes a lot of space in memory, if we use contiguous allocation using arrays. This type of allocation is efficient in terms of memory for a general matrix but not efficient for sparse matrix. Thus, a 1000 x 1000 matrix needs 1 million storage locations in memory. Since a sparse matrix contains more number of zero elements than non zero elements, there will be a wastage of lot of memory locations in case of large sparse matrix. This problem can be alleviated if we use linked representation for realizing sparse matrix.

Consider a sparse matrix shown in Figure 4.5(b). The node structure for the linked representation of the sparse matrix is shown in Figure 4.6. Each non-zero element of the matrix is represented using this node structure. Here ROW, COL, DATA fields record the row, column and the value of the non-zero element in the matrix. The RIGHT link points to the node holding the next non-zero value in the same row of the matrix. The DOWN link points to the node holding the next non-zero value in the same column of the matrix. Thus, each non-zero value is linked to its row wise and column wise non-zero neighbour.

This way we can ignore the representation of zero values in the matrix. Now each of the fields connect together to form a singly linked list with a head node. Thus all the nodes representing non-zero elements of a row in the matrix link themselves (through RIGHT link) to form a singly linked list with a head node. The number of such lists is equal to the number of rows in the matrix, which contain at least one non-zero element. Similarly, all the nodes representing the non-zero elements of a column in the matrix link themselves through (DOWN link) to form a singly linked list with a head node. The number of such lists is equal to the number of columns in the matrix, which contain at least one non-zero element. All the head nodes are also linked together to form a singly linked list. The head nodes of the row lists have their COL fields to be zero and the head nodes of the column lists have their ROW fields to be zero. The head node of all head nodes, indicated by START, stores the dimension of the original matrix in its ROW, COL fields. Figure 4.7 shows the linked representation of the sparse matrix shown in Figure 4.5(b).

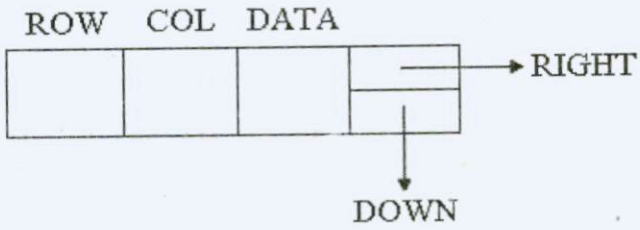


Figure 4.6 Node structure representing an element in a sparse matrix.

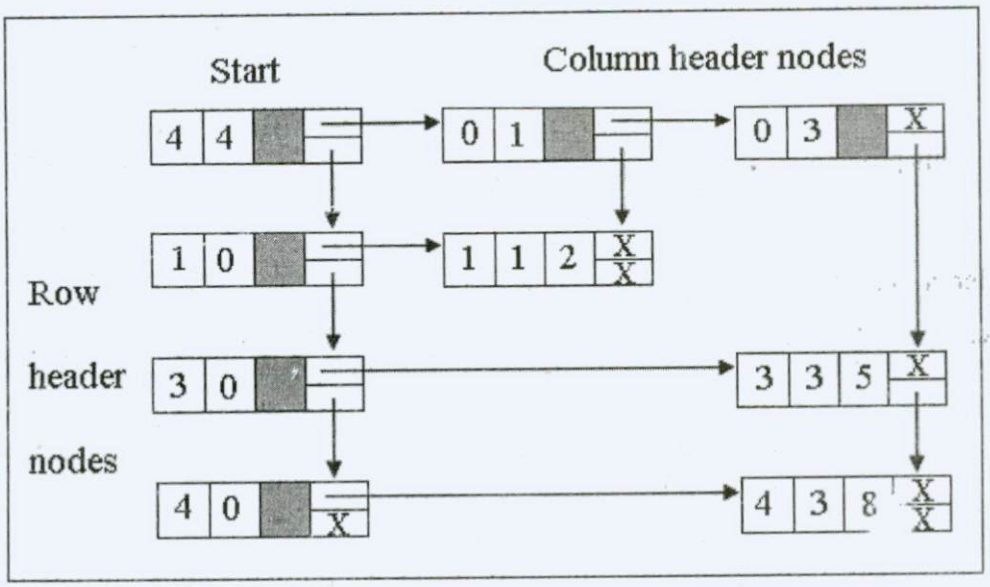
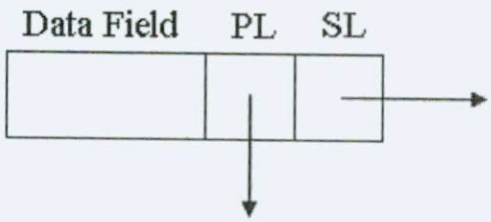


Figure 4.7 Linked representation of the sparse matrix shown in Figure 4.5(b).

12.5 DICTIONARY CONSTRUCTION

Construction of a dictionary using linked lists is a simple and a straight forward task. First, we need to define a node structure with one data field to store the dictionary, and two link fields out which one holds the address of the list containing the words with similar meaning and another link field to contain the address of the list of words in the dictionary. Figure 4.8 shows the node structure of a linked list used in the dictionary construction and Figure 4.9 shows a sample dictionary constructed using linked lists.



In above node structure, PL stands for Primary Link. It holds the address of the next node containing the word in the dictionary list. Similarly, SL stands for Secondary Link. It holds the address of the next node containing the word with similar meaning. Thus, the SL link field always points to the words with similar meaning where as the PL link points to the list of words considered for creating a dictionary.

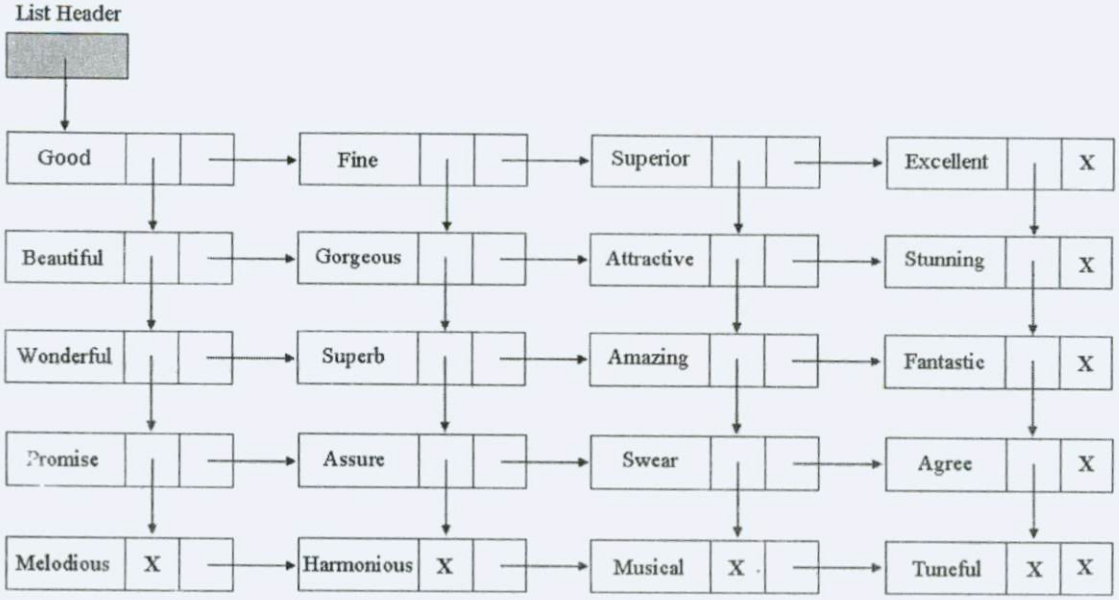


Figure 4.9 Sample dictionary constructed using linked lists.

Figure 4.9 shows a sample dictionary of five words. A list header node contains the address of the first node in the dictionary from where we can start our searching process to find the meaning for any word in the dictionary. For example, if we want to know the meaning for a word **Promise**, we need to start with the header node. First, we compare the word **Promise** with the word stored in the first node i.e. **Good**. Since they are not equal, we move on to the next node using the link filed **PL**, then we compare with the word stored in the next node i.e. **Beautiful**. Again, they are not same. We continue like this searching for a word in the word list using primary link **PL**. Once we find the word, we traverse through the secondary link **SL** to find the words with similar meaning. In the above example dictionary, the word **Promise** is found in the fourth node. Then, we follow the **SL** link of this node to get all the words with similar meaning. The SL link field of the last word in the dictionary is ***null***. Similarly, the PL link fields of all the nodes that appear at the end of the similar word list are ***null***. The Figure 4.9 above illustrates this fact.

12.6 SUMMARY

- A singly linked list can be used to represent a polynomial with one or more variables. Addition of two polynomials can be performed using such a representation.
- Linked lists can also be used to construct symbol table and dictionary.
- A sparse matrix can be represented efficiently using linked lists.

12.7 KEYWORDS

Polynomial representation, Polynomial addition, Dictionary construction, Spare matrix representation.

12.8 QUESTIONS

1. What is a polynomial? How do you represent a polynomial using linked list?
2. Explain addition of two polynomials with an example.
3. Give an algorithm to perform addition of two polynomials.

4. What is a sparse matrix? What is the drawback of representing a sparse matrix using sequential allocation method?
5. How do you represent a sparse matrix using linked lists? Explain with an illustrative example.

12.9 REFERENCES

1. Sams Teach Your Self Data Structures and Algorithms in24Hours.
2. C and Data Structures by Practice- Ramesh, Anand and Gautham.
3. Data Structures and Algorithms: Concepts, Techniques and Applications by GAV Pai. Tata McGraw Hill, New Delhi.

*****

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-I 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 vertices together with a (possibly empty) set E of ordered pairs of vertices of G called arcs
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, E does 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 by V (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| sometimes denoted 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 graph is half the size of the corresponding digraph.

Definition 4: A *walk* in a graph (digraph) G is a sequence of vertices $v_0, v_1 \ldots v_n$ such that for all $0 \leq i < n$, (v_i, v_{i+1}) is an edge (arc) in G. The length of the walk $v_0, v_1 \ldots v_n$ 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 1.1

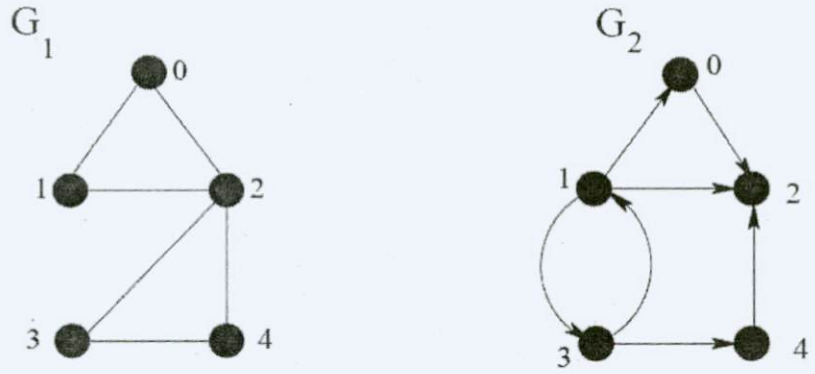


Figure 1.1 A Graph G_1 and a digraph G_2

Example 1: For the graph G_1 of Figure 1.1, 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 1.1, 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 1.1, 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 1.2 shows a weighted graph.

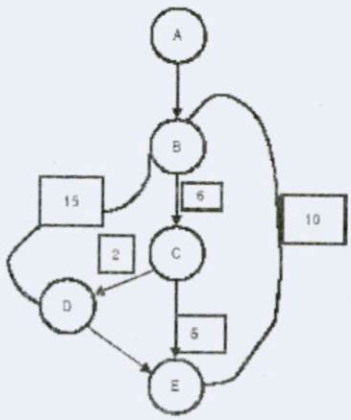


Figure 1.2 A weighted graph

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}

$$A_{ij} = \begin{cases} 1 & \text{if there is an edge between nodes } i \text{ and } j \\ 0 & \text{Otherwise} \end{cases}$$

Note that the number of 1s in a row represents the out degree of a node. In case of undirected graph, the number of 1s represents the degree of the node. Total number of 1s in the matrix represents number of edges. Figure 1.3(a) shows a graph and Figure 1.3(b) shows its adjacency matrix.

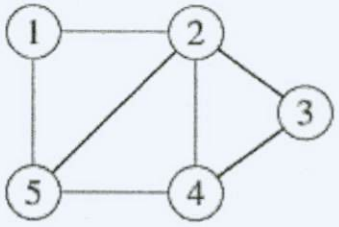


Figure 1.3(a) Graph

	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 1.3(b) Adjacency matrix

Figure 1.4(a) shows a digraph and Figure 1.4(b) shows its adjacency matrix.

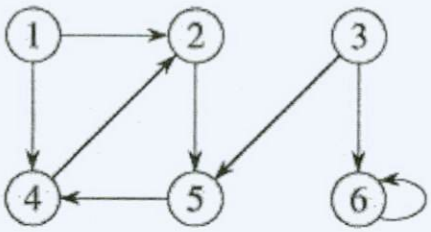


Figure 1.4(a) Digraph

	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 1.4(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

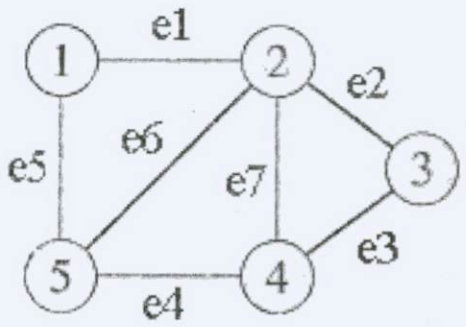


Figure 1.5(a) Undirected graph

Figure 1.5(b) Incidence matrix

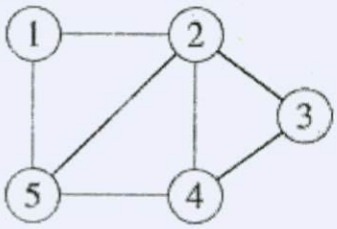
The incidence matrix contains only two elements, 0 and 1. Such a matrix is called a *binary matrix* or a *(0, 1)-matrix*.

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.



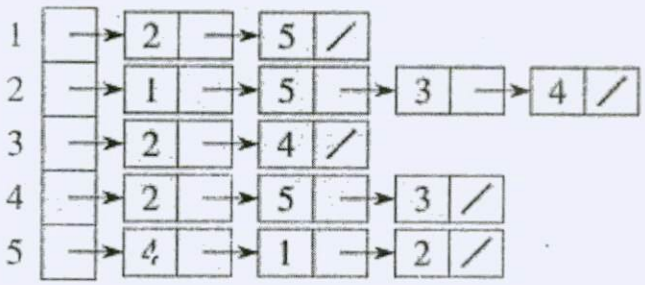


Figure 1.6(a) Undirected graph

Figure 1.6(b) Linked representation of a graph

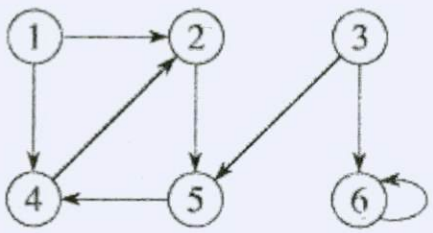


Figure 1.7(a) Digraph

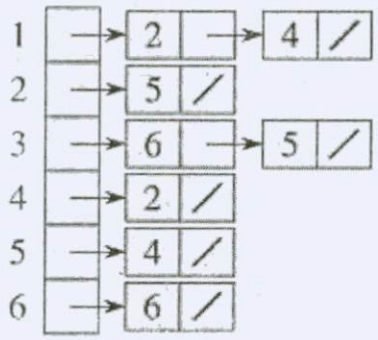


Figure 1.7(b) Linked representation of a graph

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

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, McGraw Hill 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 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.

14.1 INTRODUCTION

In Unit-I, 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.

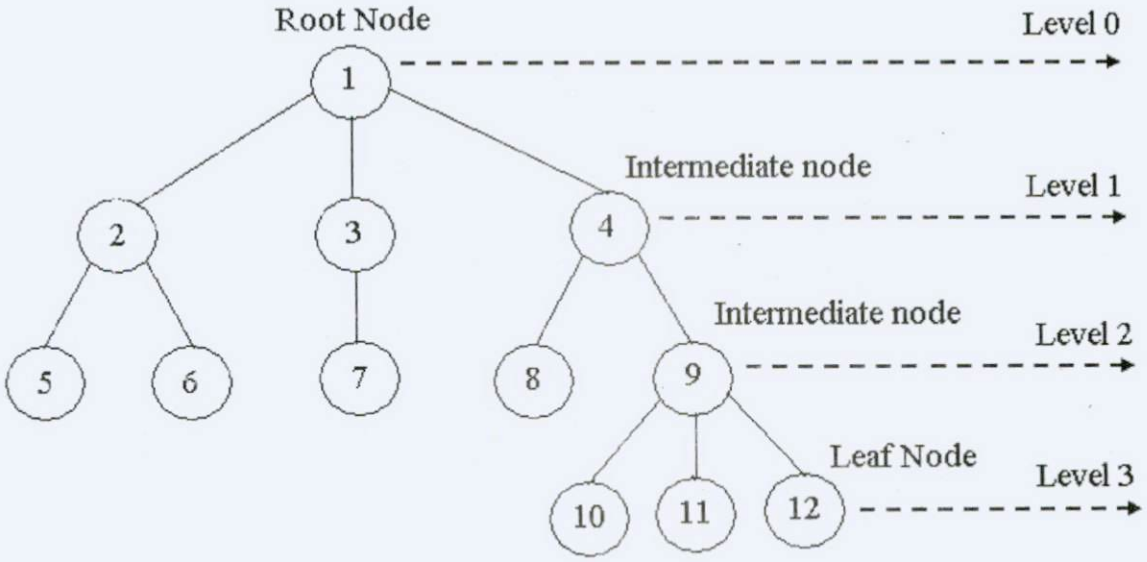


Figure 2.1 An example tree

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 TREE

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.

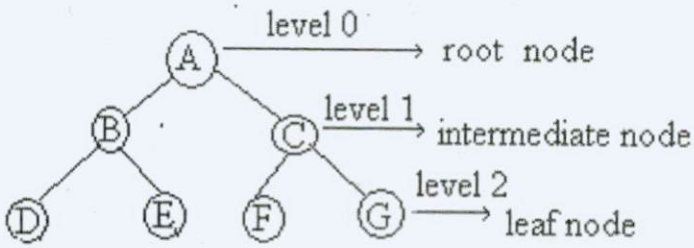


Figure 2.2 An example binary tree

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.

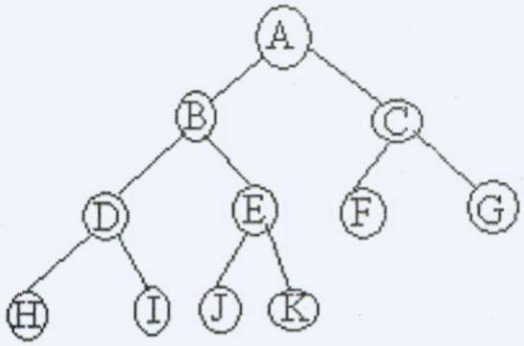


Figure 2.3 A complete binary tree

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:

$$2^0 + 2^1 + 2^2 + \ldots + 2^l$$

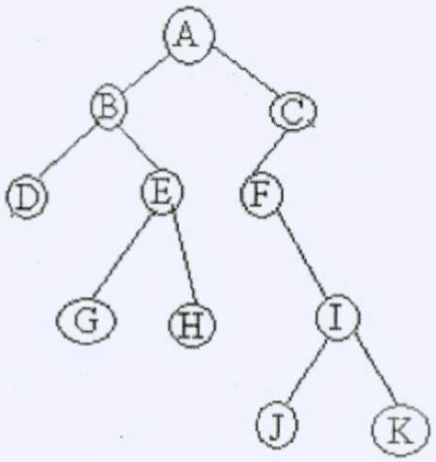
i.e., $\sum 2^k = 2^{l+1} - 1$ for $k = 0$ to l

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 $n \times n$ 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 indicate the 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^2 number 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

$$\frac{n-1}{2^{l+1}-l}\times 100$$

Figure 2.5 shows a binary tree and Figure 2.6 shows its one-dimensional array representation.

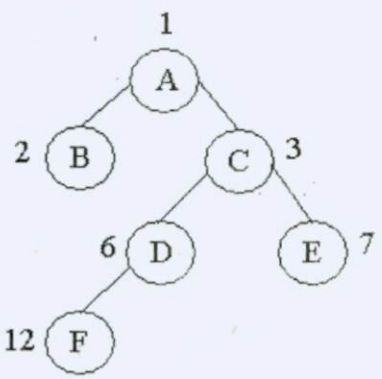


Figure 2.5 A binary tree

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.

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.

14.5 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.

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.

14.7 QUESTIONS

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?

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

REPRESENTATION OF BINARY TREES BASED ON LINKED ALLOCATION METHOD

Structure:

15.0 OBJECTIVES

After studying this unit, we will be able to

- Explain the linked representation of binary trees.
- List out the advantages of representing Binary Trees using linked allocation.
- Binary tree as a data structure.

15.1 INTRODUCTION

In Unit-II of this module, 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 unit, we will be discussing the linked list representation of a binary tree and their advantages.

15.2 LINKED REPRESENTATION OF BINARY TREES

The linked representation of a binary tree has the node structure shown in Figure 3.1. 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 3.2 shows an example binary tree and Figure 3.3 shows its linked representation.

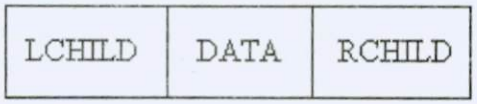


Figure 3.1 Structure of Node in Binary Tree

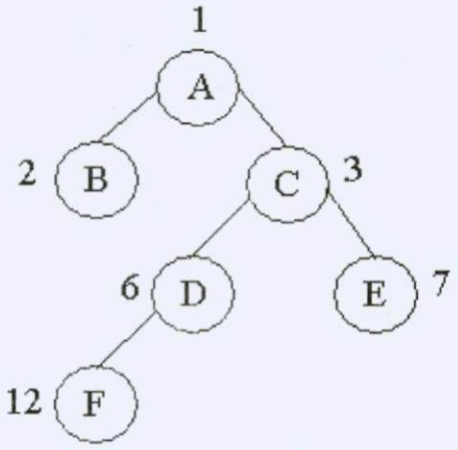


Figure 3.2 A Binary Tree

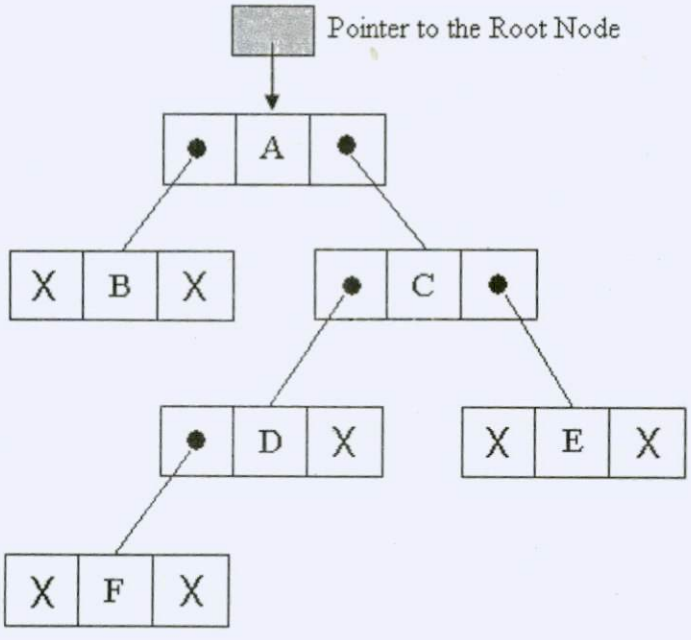


Figure 3.3 Linked representation of a Binary Tree

In the binary tree T shown in Figure 3.3, 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 3.4 illustrates the node structure of a binary tree, which incorporates the parent information and Figure 3.5 shows an instance of a binary tree with additional parent information.

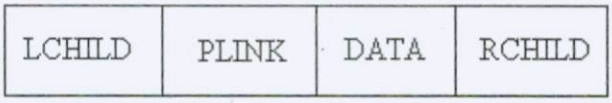


Figure 3.4 Node structure with parent link

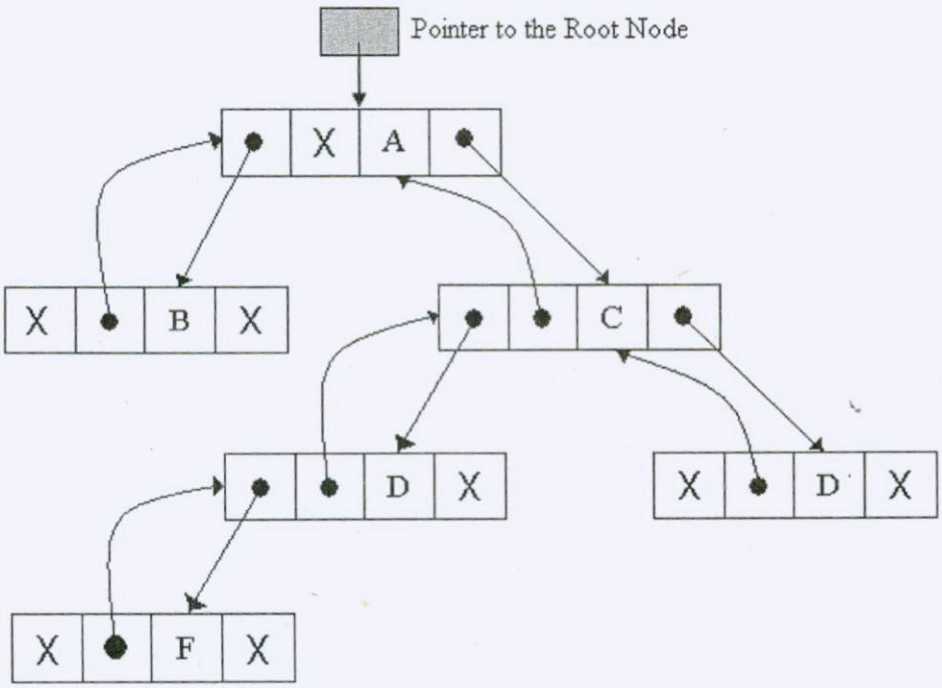


Figure 3.5 Linked representation of a binary tree with access to parent node.

From Figure 3.4, 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 3.5, 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.

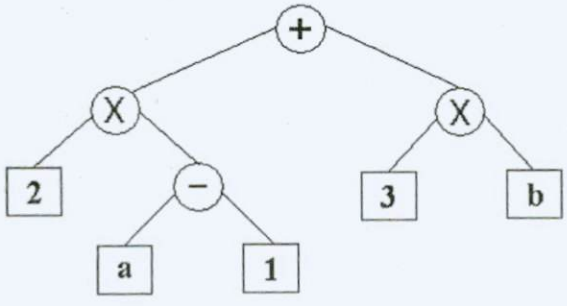
In this representation, ordering of nodes is not mandatory and we require a header to point to the root node of the binary tree. If there are 'n' nodes to be represented, only 'n' node-structures will be allocated. This means that there will be $2n$ link fields. Out of $2n$ link fields only (n-1) will be actually used to point to the next nodes and the remaining are wasted. Therefore the percentage of utilization is given as:

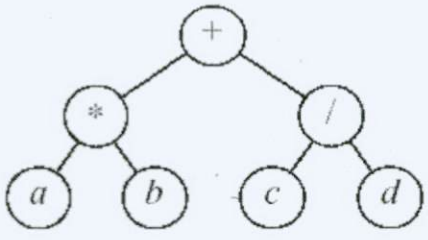
$$\frac{n-1}{2n} = \frac{1}{n} \times 100 = 50\%$$

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

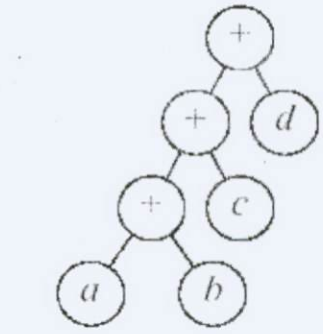
Binary tree associated with an arithmetic expression is called an arithmetic expression tree. The internal nodes represent the operators and the external nodes represent the operands. Figure 3.6 shows the arithmetic expression trees.

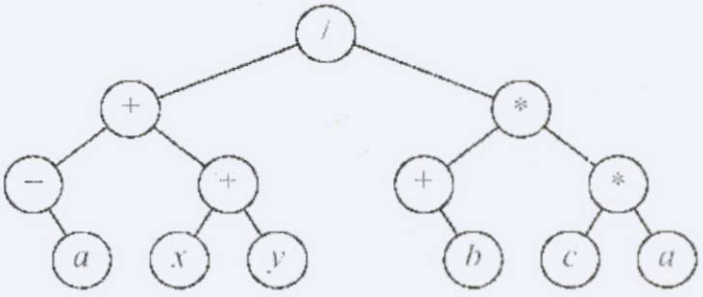




(a) (2 * (a − 1) + (3 * b))

(b) (a * b) + (c / d)





(c) ((a + b) + c) + d)

(d) ((-a) + (x + y)) / ((+ b) * (c * a))

Figure 3.6 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 3.7 shows the decision tree for the biggest of three numbers.

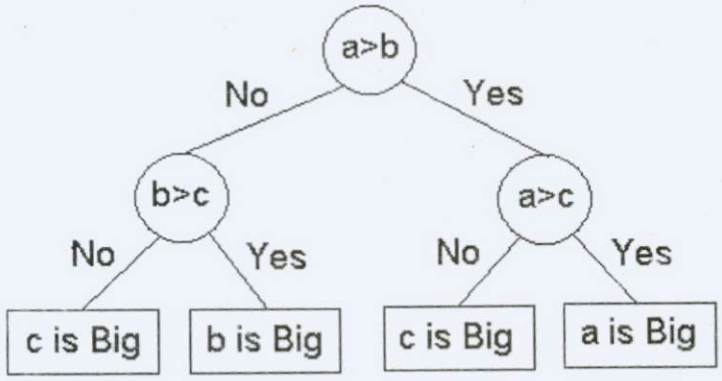


Figure 3.7 Decision tree

15.3 BINARY TREE AS A DATA STRUCTURE

A collection of vertices is said to be a binary tree if there is a special vertex called root and the remaining vertices can be partitioned into utmost two such that each partition

is a binary tree. To describe a binary tree as a data structure, we need to define its domain, a set of functions permitted on binary tree and the axioms associated with it.

Domain: Application Dependent

Functions:

Create-BT() ---------- Allocation

Construction(BT_1, BT_2, data) ---------- BT

Traversal(BT, option) ---------- Sequence

Heap(BT) ---------- Heap

Root(BT) ---------- Data

Lchild(BT) ---------- BT

Rchild(BT) ---------- BT

Search(BT, data) ---------- Boolean

Insertion(BT, *e*) ---------- Updated BT

Deletion(BT, e) ---------- Updated BT

Isempty(BT) ---------- Boolean

Isfull(BT) ---------- Boolean

Axioms:

Isempty(Create-BT()) ---------- True

Isfull(Create-BT()) ---------- False

Root(Construction(BT_1, BT_2, *e*)) ---------- *e*

Lchild(Construction(BT_1, BT_2, *e*)) -------- BT_1

Rchild(Construction(BT_1, BT_2, *e*)) -------- BT_2

Search((Insertion(BT, *e*), *e*) ---------- True

BT(D, F, A): Binary tree as a data structure.

15.4 SUMMARY

In this unit, we have introduced the binary trees. Trees form the core of non-linear data structure and Binary tree helps in storing the data more efficiently although the access is not as simple as arrays. The linked representation of a binary tree is described and the binary as a data structure is discussed. 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.

15.5 KEYWORDS

Binary tree, linked representation, expression tree, decision tree, left child, right child.

15.6 QUESTIONS

1. Describe the node structure used to represent a binary.
2. Explain linked representation of a binary tree with an example.
3. What are the advantages of linked representation over array based representation of a binary?
4. 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.
5. What are expression trees and decision trees? Explain with suitable examples.
6. Describe binary tree as a data structure.

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

TRAVERSAL OF BINARY TREES AND OPERATIONS ON BINARY TREES

Structure:

16.0 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.

16.1 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 process is not as commonly used as finding, inserting, and deleting nodes. One reason for this is that traversal is not particularly fast. But traversing a tree has some surprisingly useful applications and is theoretically interesting. In addition, we discuss some other important operations on binary trees.

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 Figure 4.1 is: A B D E H I C F G

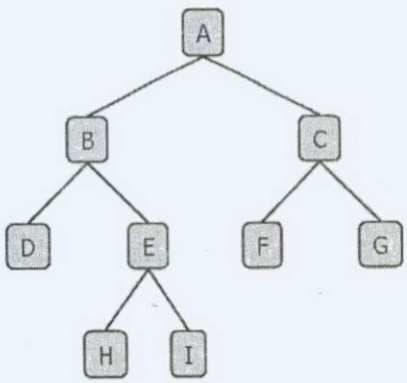


Figure 4.1 A binary tree

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 only three 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 Figure 4.1 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 Figure 4.1 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 ($A[i] \neq 0$)

Display ($A[i]$)

Preorder Traversal (A, $2i$)

Preorder Traversal (A, $2i + 1$)

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 ($A[i] \neq 0$)

Inorder Traversal (A, $2i$)

Display ($A[i]$)

Inorder Traversal (A, $2i + 1$)

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 ($A[i] \neq 0$)

Postorder Traversal (A, $2i$)

Postorder Traversal (A, $2i$ +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 ($bt \neq$ 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 ($bt \neq$ 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 ($bt \neq$ NULL)

Postorder Traversal ([bt].Lchild)

Postorder Traversal ([bt].Rchild)

Display ([bt].data)

If end

Algorithm ends.

16.3 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 4.2, we need to maintain an array say ***D***, which contains only the directions where the node temp has to be inserted.

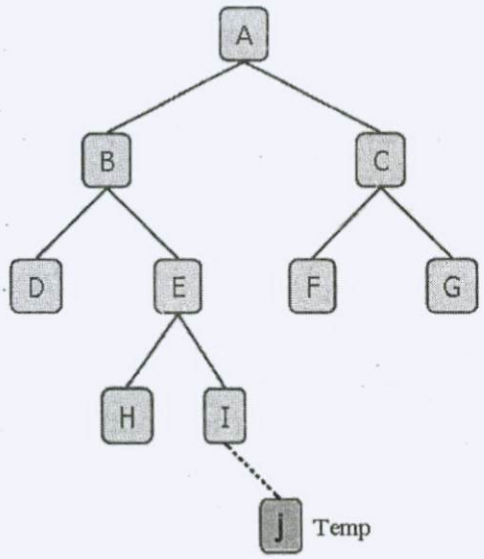


Figure 4.2 To insert a item ***J***

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 is greater 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. Figure 4.3(a) shows an example binary tree where as Figure 4.3(b) is not a binary search tree but a binary tree.

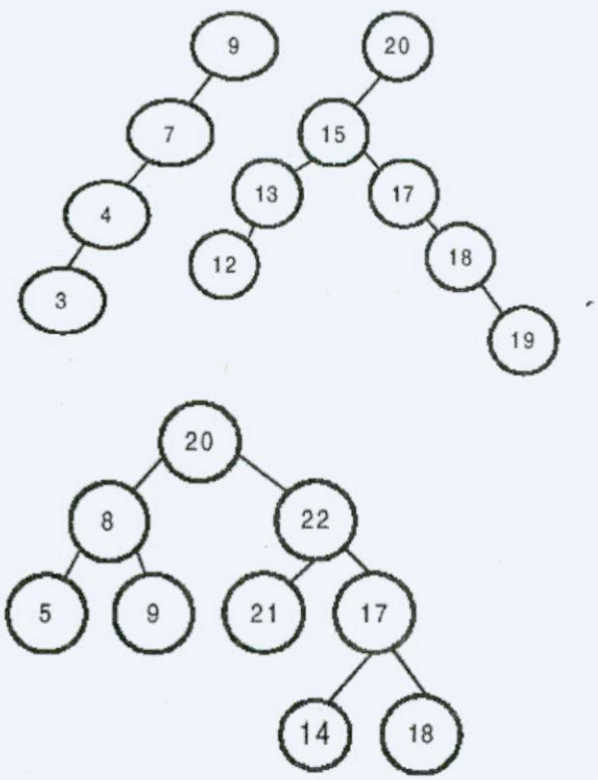


Figure 4.3(a) A binary search tree

Figure 4.3(b) Not a binary search tree

We will be using BST structure to demonstrate features of Binary Trees. The operations possible on a binary 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 4.4 (a – b) illustrates an insertion operation.

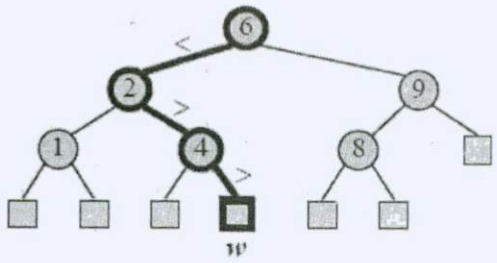


Figure 4.4(a) Before insertion

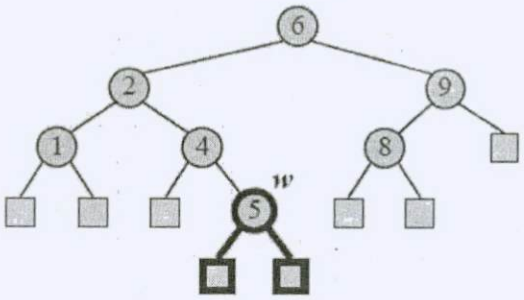


Figure 4.4(b) After insertion of item 5

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. For example, to delete node 4 the right pointer of its parent node 5 is made to point to NULL and free node 4. Figure 4.5 shows an instance of delete operation.

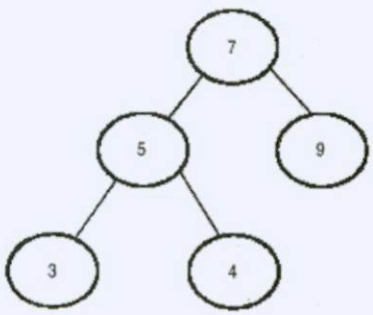


Figure 4.5 Deleting a leaf node 4

(b) Deleting a node with one child only, either left child or Right child.

For example, delete node 9 that has only a right child as shown in Figure 4.6. The right pointer of node 7 is made to point to node 11. The new tree after deletion is shown in 4.7.

VOP- 4 -64

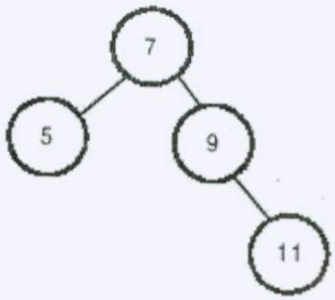


Figure 4.6 Deletion of node with only one child deletion

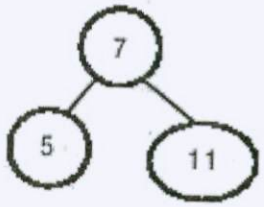


Figure 4.7 New tree after deletion

(c) Node to be deleted is an intermediate node.

- To perform operation **RemoveElement(*k*),** we search for key ***k***
- Assume key ***k*** is in the tree, and let ***v*** be the node storing ***k***
- If node ***v*** has a leaf child ***w***, we remove ***v*** and ***w*** from the tree with operation RemoveAboveExternal(***w***)
- Example: Remove 4

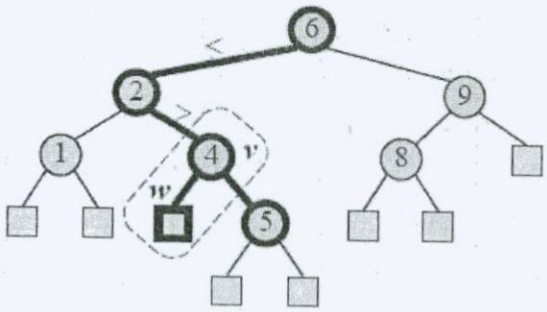


Figure 4.8 Before deletion

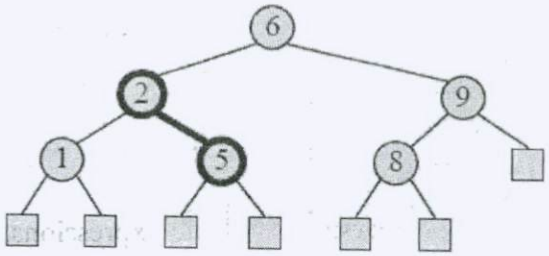


Figure 4.9 After deletion

Searching for a node in a Binary Search Tree

- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found and we return 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 4.10 shows an instance of searching for a node containing element 4.

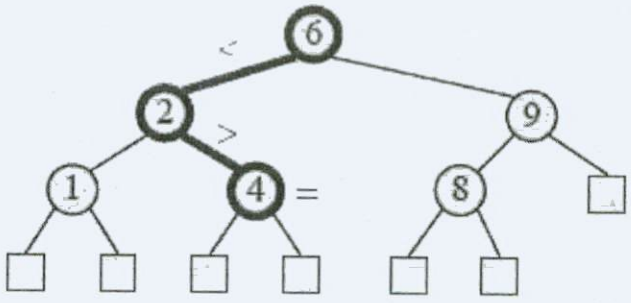


Figure 4.10 Searching operation

16.4 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.

16.5 KEYWORDS

Traversing, preorder, inorder, postorder, insertion, deletion, searching, binary search tree, left subtree, right subtree.

16.6 QUESTIONS

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.

16.7 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 -17

THREADED BINARY TREE AND ITS TRAVERSAL

Structure

17.0 OBJECTIVES

After reading this unit you should be able to

- 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

17.1 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 on sequential allocation or in the form of linked list allocation. Consider a tree of n nodes represented using linked list 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/ links in the tree. Out of 2*n nodes the linked list representation will use (n-1) links. So, the total number of unused links in linked representation is 2*n – (n-1) = 2*n-n+1 = n+1. The memory for these 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 traversing of binary tree which internally uses stack require more memory and time. Using the unused n+1 pointer's 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 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.

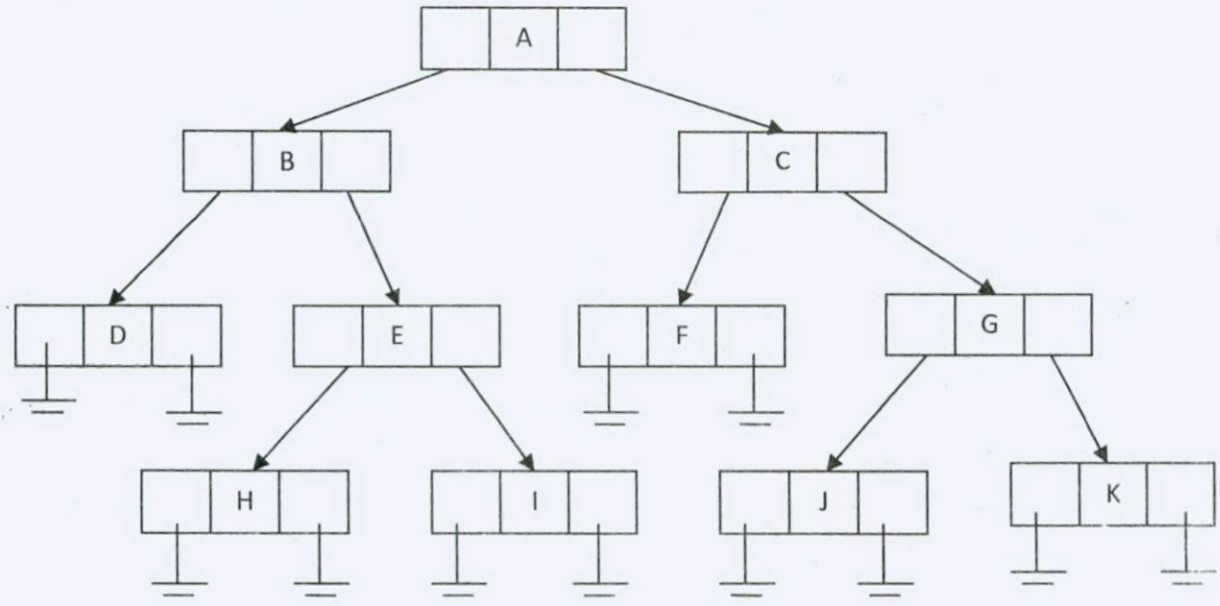


Figure 1. A sample binary tree with 11 nodes.

17.2 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.

17.2 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 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 identified by looking into 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 2. The new links created will be called as thread and it has shown in dotted lines.

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 a 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.

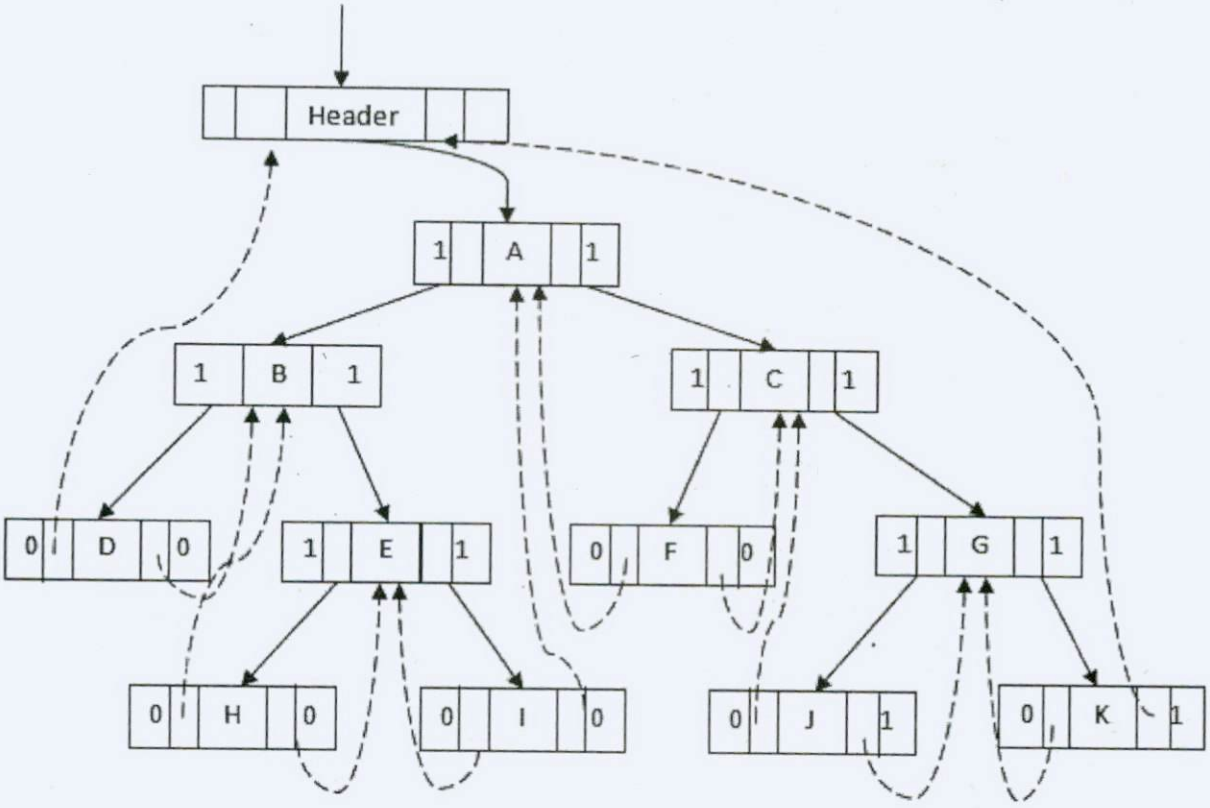


Figure 2: Inorder threaded binary tree

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 3.

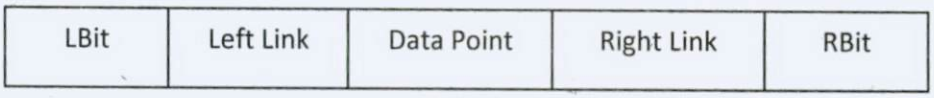


Figure 3. Structure of a node in a threaded binary tree

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.

3. 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.

17.3 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.

4 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.

17.4 DIFFERENCE BETWEEN BINARY TREE AND THREADED BINARY TREE

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

17.5 SUMMARY

In this unit we have studied the limitations of conventional binary tree and inorder 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.

17.6 KEYWORDS

(1). Threaded binary tree

(2). Inordered threaded binary tree traversal

17.7 QUESTIONS

(1) Design an algorithm to find the predecessor node in an inorder threded binary tree.

(2) Suggest an algorithm to construct preorder threaded binary tree.

(3) Suggest an algorithm to construct postorder threaded binary tree.

17.7.1 TEXT BOOK

(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 - 18

REPRESENTATION OF FOREST OF TREES

Structure:

18 OBJECTIVES

After reading this unit you should be able

- 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

18.2 INTRODUCTION TO GENERAL TREES

In module 4 we have already understood the concept 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 $T_1, T_2, \ldots, T_n$, where each T_i (i = 1, 2, …, n) is a tree; $T_1, T_2, \ldots, T_n$ are called sub tree of the root.

This definition implies that a tree must contain at least one node and the number of nodes are 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 1. shows two general tree structures. It should be noted that the node can have more than two child node.

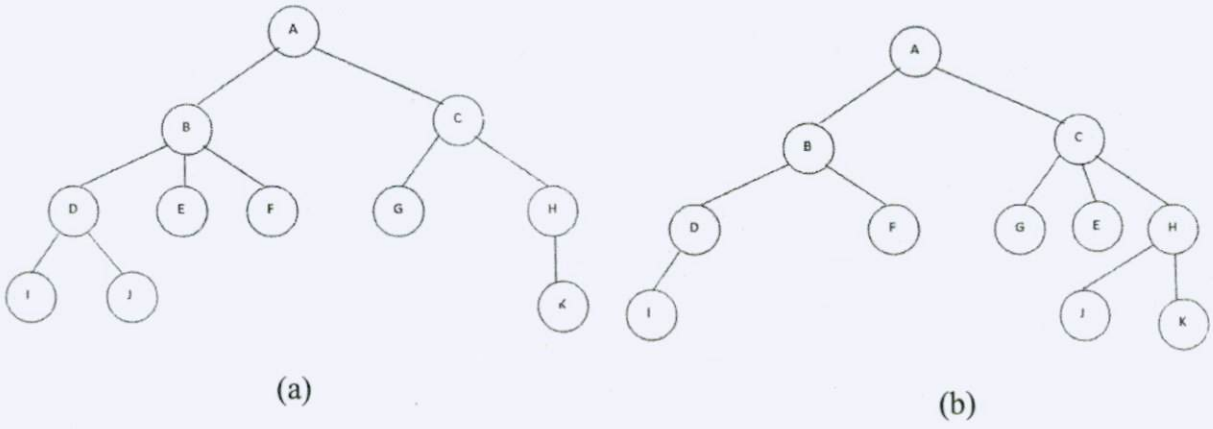


Figure. 1 Two general trees.

18.3 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.

18.2.1 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 2.

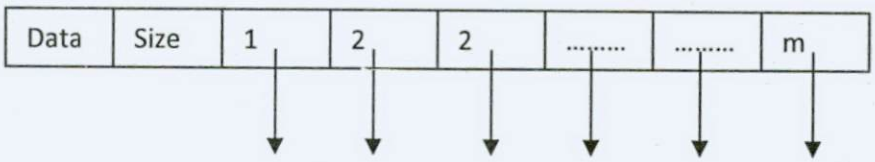


Figure 2. Node structure for a node in linked representation of a tree.

For the Figure 1(a), the equivalent tree represented using linked list representation is shown in Figure 3. 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.

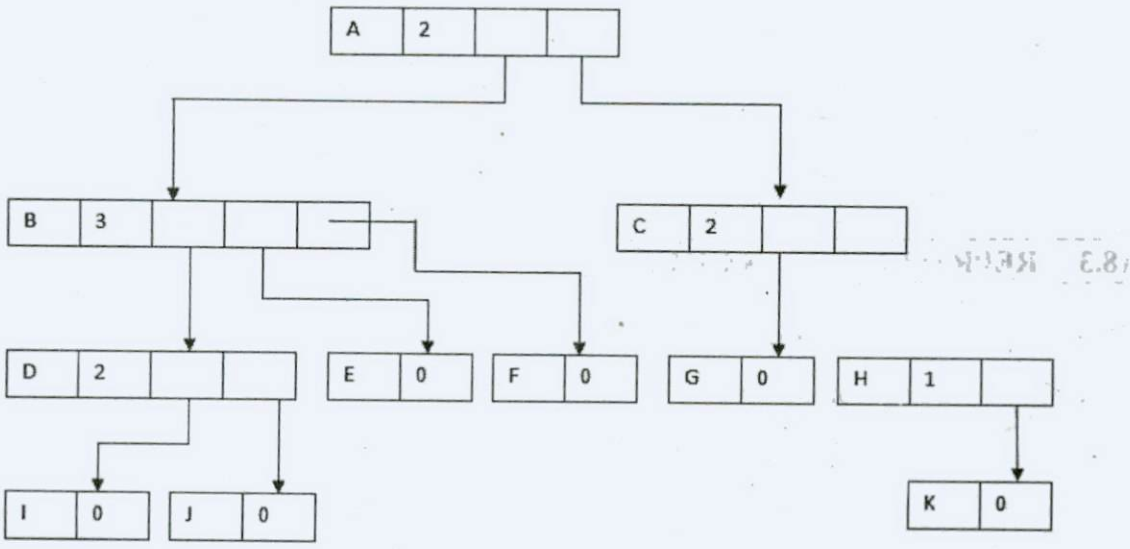


Figure 3. A tree represented using linked list representation

18.4 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 1(a). Figure 4(a) shows the equivalent binary tree of a general tree in Figure 1(a). Figure 4(b) shows tilted 45^0 clockwise of Figure 4(a).

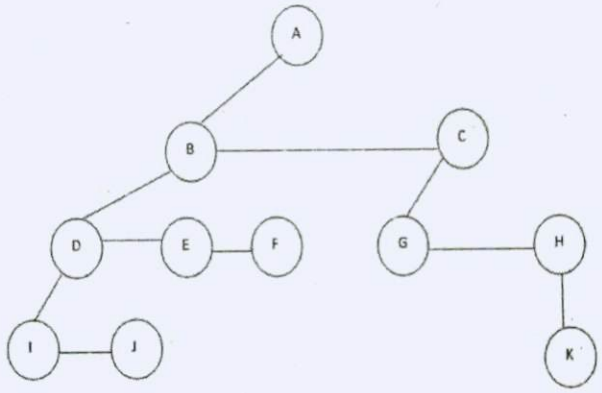


Figure 4(a). Equivalent binary tree obtained after step 1

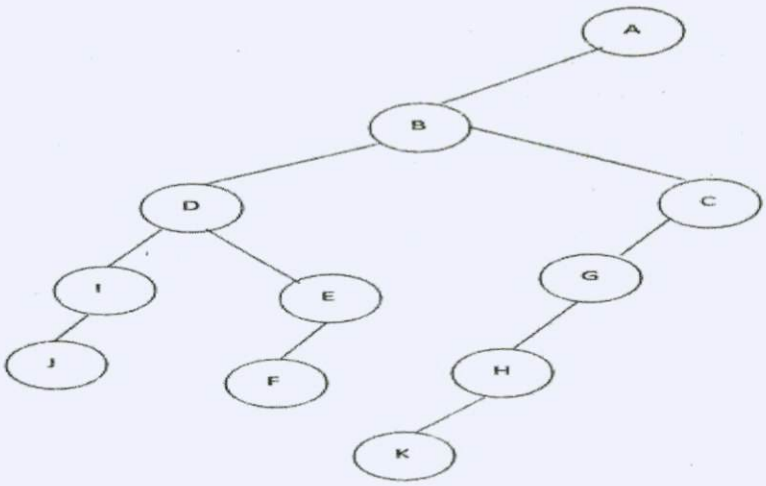


Figure 4(b). Equivalent binary tree obtained after step 2

18.5 FOREST

A forest is nothing but a collection of disjoint trees. Figure 5 shows a forest

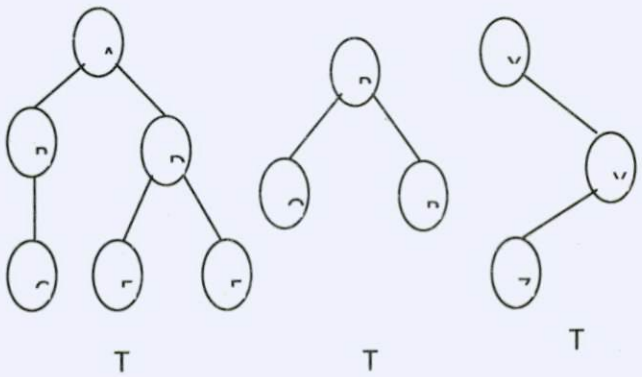


Figure 5: A forest F

Here, F is a forest which consists of three trees T_1, T_2, and T_3.

18.6 SUMMARY

In this chapter we have introduced 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 is also presented in this unit.

18.7 KEYWORDS

(1). General tree

(2). Linked list representation of tree

18.8 QUESTIONS

(1) What is the difference between binary tree and a general tree?

(2) With a neat diagram explain the linked list representation of general tree.

(3) Consider a general tree of your own choice and convert it into its equivalent binary tree.

(4) What is forest?

18.9 TEXT BOOK

(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 -19

TRAVERSAL OF FOREST

Structure

19.0 OBJECTIVES

After reading this unit you should be able to

- Traverse the given forest in preorder
- Traverse the given forest in inorder
- Traverse the given forest in postorder

- Introduction to Forest traversal

In the last unit we have learnt about representation of trees using linked list allocation, their traversal and converting a general tree into its equivalent binary tree. Also we have learnt

about forest i.e., collection of trees. 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 unit we shall study about traversal of forest ie.., 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 1 to traverse in different orders.

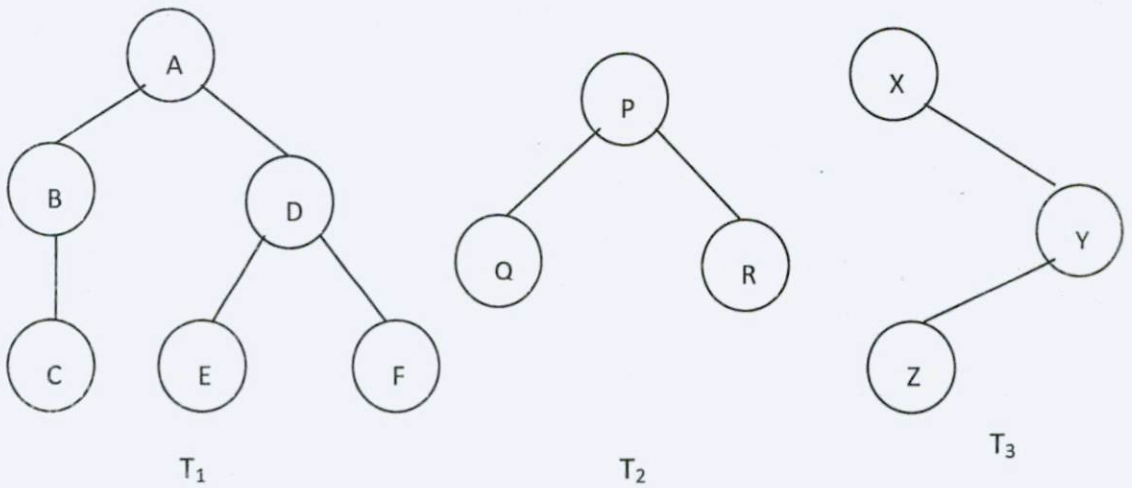


Figure 1. A forest F

19.1.1.1 Preorder traversal of forest

The preorder 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

19.1.2 INORDER TRAVERSAL OF FOREST

The inorder 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

19.3 POSTORDER TRAVERSAL OF FOREST

The postorder 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

19.4 SUMMARY

In this unit we have presented ways of traversing forest. Similar to binary tree traversal we can traverse forest in preorder, inorder and postorder.

19.4 KEYWORDS

(1). Forest traversal

(2). Preorder forest sequence

(3). Inorder forest sequence

(4). Postorder forest sequence

19.5 QUESTIONS

(1) Design recursive algorithm to traverse forest in preorder

(2) Design recursive algorithm to traverse forest in inorder

(3) Design recursive algorithm to traverse forest in postorder

19.6 REFERENCE

(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- 20

CONVERSION OF FOREST TO BINARY TREE

Structure

20.1 OBJECTIVES

After reading this unit you should be able

- Explain recursive way of converting forest to its equivalent binary tree
- Discuss iterative way of converting forest to its equivalent binary tree

20.2 INTRODUCTION

In unit 2 of this module 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. Figure 1 shows a forest with three trees. 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.

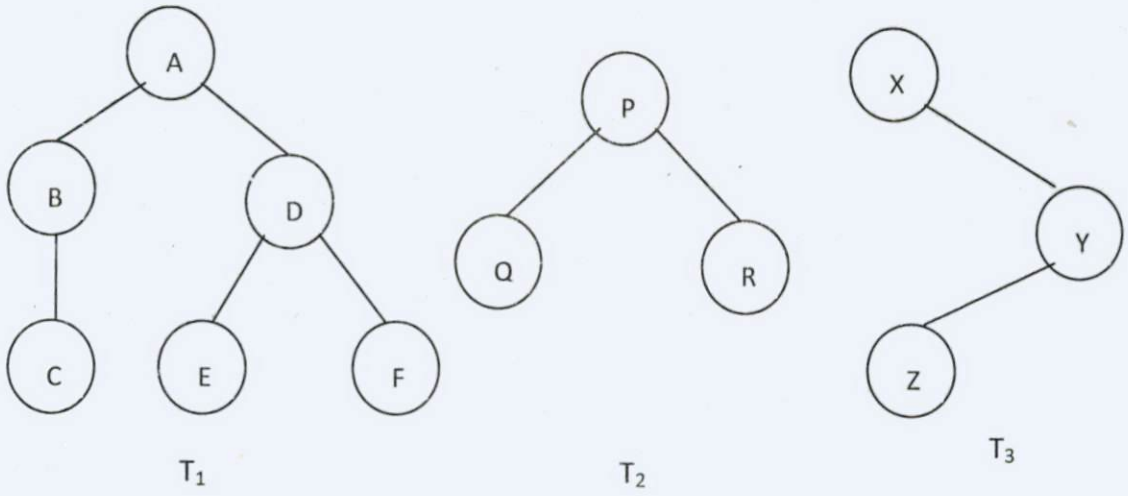


Figure 1: A forest F

20.3 RECURSION BASED APPROACH

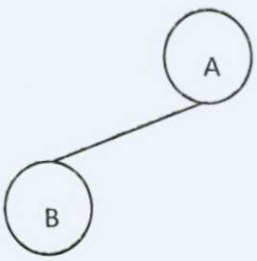
In order to represent forest in computer we have to convert it into its equivalent binary tree. In this section we shall see how to convert using recursion discussed in module 3.

The steps involved in converting any forest F into its corresponding binary tree are as follows

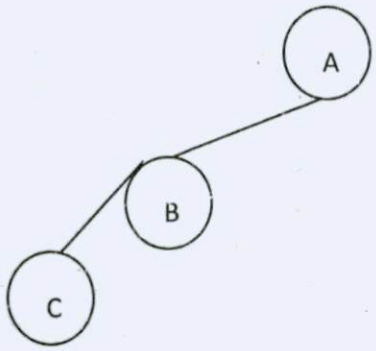
- Convert each tree T_i (i=1, 2, .., n) into its corresponding binary tree $T_i^{|}$. Note that in such that $T_i^{|}$, the right link is empty.
- Assume that the root of T_1 will be the root of binary tree equivalent of forest. Then Add $T_{i+1}^{|}$ as the right sub-tree of T_i for all i = 1, 2, …, n-1.

For illustration purpose, let us consider the transformation of forest F as shown in Figure 1. Step by step illustration is given below.

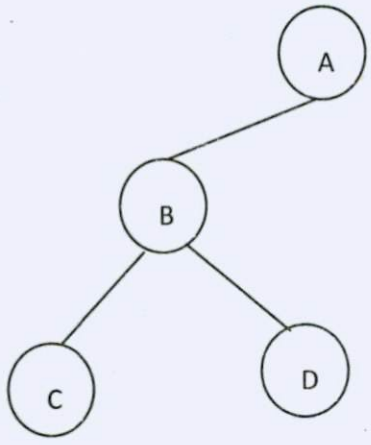
Roots of the binary tree



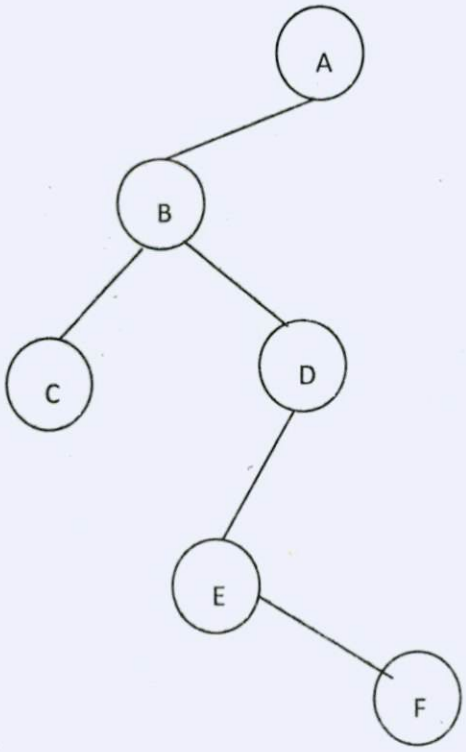
Since B is the first left child of A in the T_1 make it as the first left child.



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

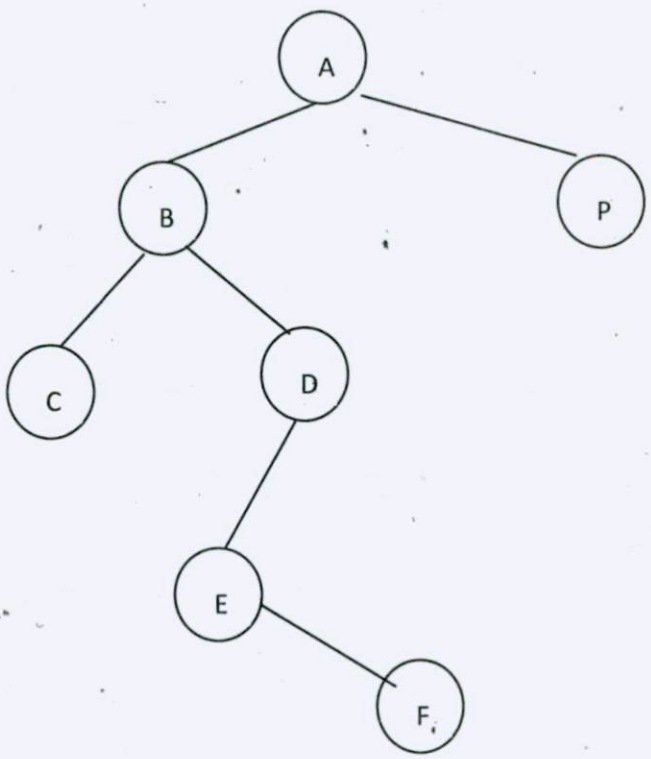


Node D has E and F has two child nodes which are leaf node.

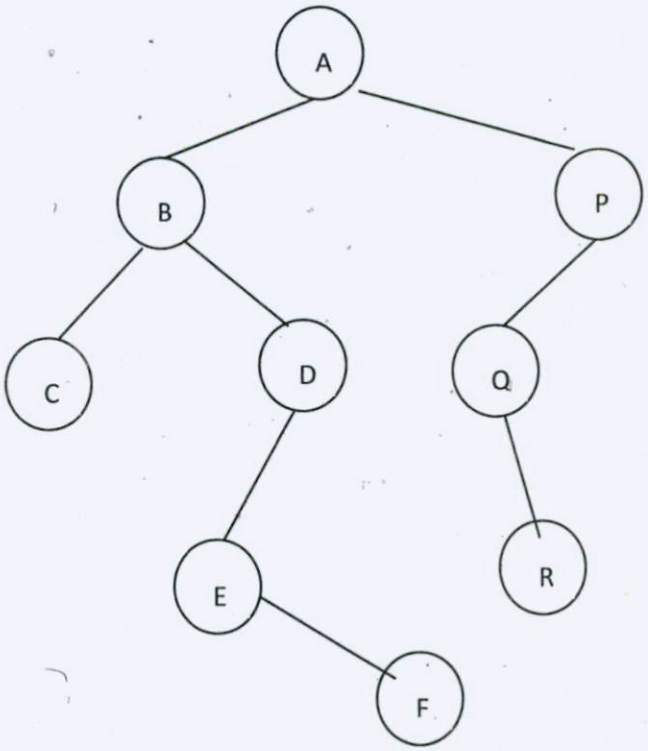
The first node is E make E as left child of D. Since D has no child nodes and has F as its sibling make it as right node of E.

This completes First sub tree.

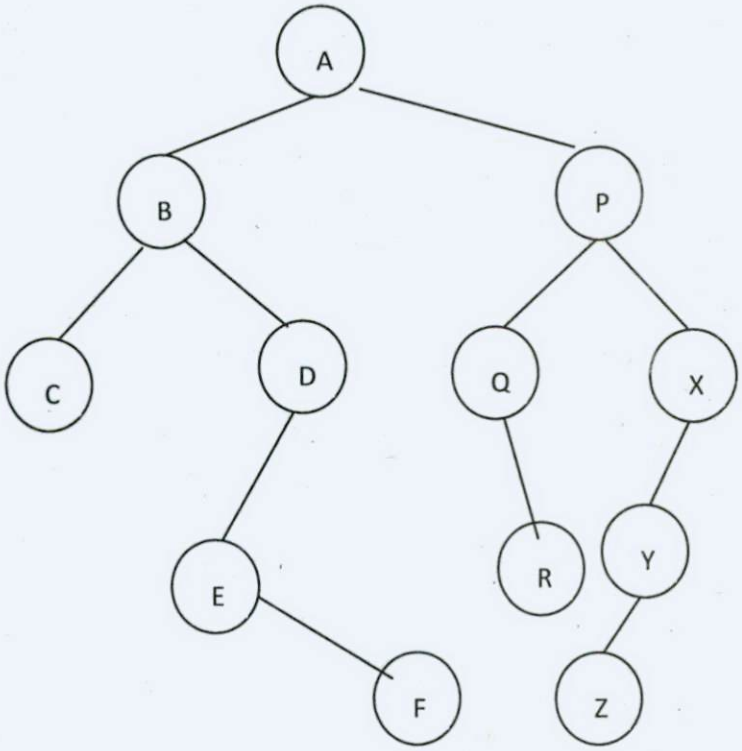
Now consider Tree T_2. The first node of T_2 is P. Make node P as r ht child of root node A.



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

Algorithm to convert a given forest to its equivalent binary tree

Algorithm ForestBinaryTree

:

Input n – number of trees

$F = \{ T_1, T_2, \ldots T_n\}$

Output bt – Binary tree

Method

If($n \neq 0$) then

root(bt) = root(T_1)

LeftSubTree(bt) = ForestBinartTree(m, T_{11}, T_{12}, …, T_{1m})

RightSubTree(bt) = ForestBinaryTree(n-1, T_2, T_3, …, T_n)

If end

Algorithm ends

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 an

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.

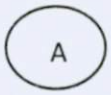
Let us consider the forest considered in Figure 1 and illustrate the steps involved in converting forest to its equivalent binary tree.

The preorder and inorder sequence of the forest F in Figure 1 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

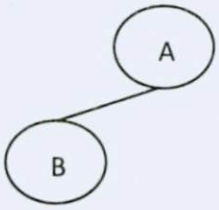
The first element of the preorder sequence is A, make this as a root of the binary tree



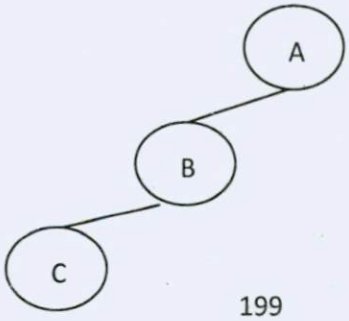
Search for the element A in the preorder sequence. The elements which are left to element A will form left subtree and elements which are right of the binary tree will form the right subtree of the binary tree.

C	B	E	F	D	**A**	Q	R	P	Y	Z	X
Left subtree						Right subtree					

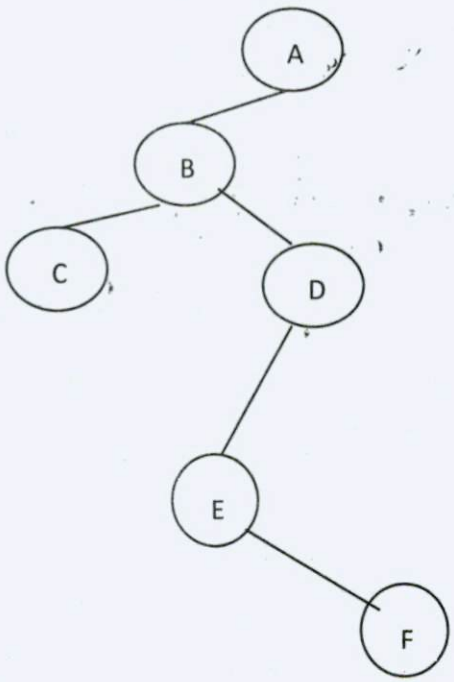
The next element in the preorder sequence is B, make it as the root of the left sub tree



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. Finally the binary tree will be as shown in Figure 2.

20.4 SUMMARY

In this unit 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 in this unit.

20.5 KEYWORDS

(1). Threaded binary tree

(2). Threaded binary tree traversal

20.6 QUESTIONS

(1) 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

(2) 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

20.7 Text book

(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 - 21

INTRODUCTION TO SORTING

Structure

21.0 OBJECTIVES

After reading this unit you should be able to

- Explain the basic concept of sorting techniques
- Discuss the working principle of conventional sorting technique
- List out the limitations of conventional sorting technique
- Discuss two other sorting techniques such as selection sort and insertion sort

21.1 INTRODUCTION TO SORTING TECHNIQUES

Sorting is a process of arranging a set of unordered elements in some predefined ordered. The simplest way of ordering any unordered set of elements in some predefined orders is to consider randomly one element at a time and array them, continue this procedure for all elements in the set. If the desired ordered set is obtained stop the procedure else continue

with other combinations. This process shall be continued until the desired order is obtained. The important thing to be noted in any sorting is to select any desired property on which the sorting will be performed and that property shall be called as sorting key.

For example consider an unordered set of records arranged sequentially containing details of students. If we want to arrange the details of the students based on the alphabetical order of the student's name, then we need to sort the records based on the student's name and student name will be treated as sorting key. Consider Table 1.1 where we have nine students and their details which are stored in an unordered way. If we want to store those details in an ordered way then we need to sort the entire table using any one of the attribute value as a sorting key.

Table 1.1: Unordered details of students

Register Number	Student Name	Course	City	Date of birth
KSOU123	Harish	M.A	Mysore	13/03/1983
KSOU125	Mallikarjuna	M.Com	Davanageri	21/12/1978
KSOU121	Nagasundara	M.Sc	Nanjanagud	12/02/1980
KSOU120	Sharath	M.A	Mandya	18/06/1988
KSOU122	Vinay	M.Sc	Thumkur	11/02/1985
KSOU124	Imran	M.Com	Challkere	13/01/1983
KSOU126	Anamica	M.A	Bangalore	01/01/1985
KSOU127	Gowri	M.Sc	Mysore	06/04/1986
KSOU128	Nayana	M.Com	Mandya	05/12/1986

Table 1.2 shows an ordered table containing details of students. In this case table is sorted based on the alphabetical order of the students and hence the sorting key is student name.

Table 1.1: Details of students ordered by their name in alphabetical order

Register Number	Student Name	Course	City	Date of birth
KSOU126	Anamica	M.A	Bangalore	01/01/1985
KSOU127	Gowri	M.Sc	Mysore	06/04/1986
KSOU123	Harish	M.A	Mysore	13/03/1983
KSOU124	Imran	M.Com	Challkere	13/01/1983

KSOU125	Mallikarjuna	M.Com	Davanageri	21/12/1978
KSOU121	Nagasundara	M.Sc	Nanjanagud	12/02/1980
KSOU128	Nayana	M.Com	Mandya	05/12/1986
KSOU120	Sharath	M.A	Mandya	18/06/1988
KSOU122	Vinay	M.Sc	Thumkur	11/02/1985

Note:

1. The sorting key can be created from two or more sort keys. The first key is termed as primary sort key and second is called as secondary sort key and so on.
2. In the literature we can a good number of sorting techniques which are classified as (i) internal techniques (ii) External techniques. Internal techniques are those which can be used when the records are small enough to be sorted within the main memory and external techniques are those which can be used when the records are huge and requires main memory and secondary memory to sort entire set of records. In this unit we shall consider only internal techniques.

In the following section we present three simple sorting techniques viz., conventional sorting technique, selection sorting technique and insertion sorting technique. In all the techniques discussed in the following sections we consider an unordered set of integer numbers. While sorting we can sort in ascending order (i.e., smallest to largest) or in descending order (i.e., largest to smallest). For simplicity in all the sections we have considered sorting of integer numbers in ascending order and the sorting in descending order is left as assignment for the students for practice.

21.2 CONVENTIONAL 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

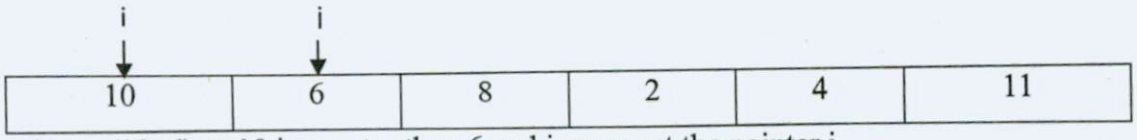
the pointer 'j' one at a time. Continue the same process until the pointer 'j' reaches the last position. At the end of this process we can see that the smallest element in the list is at the location pointed by the pointer 'i'.

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.

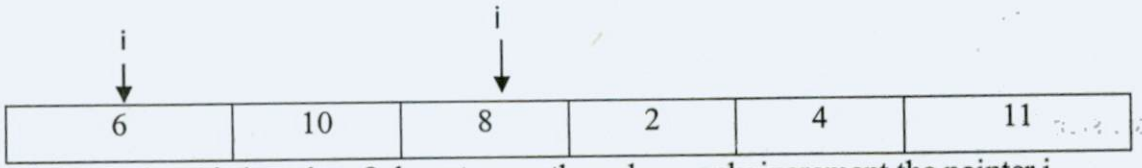
```
Algorithm   Conventional sort
Input       Unordered list - A
Output      Ordered list – A
Method

            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
```

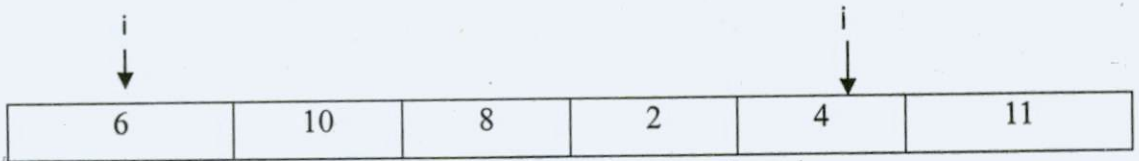
In order to illustrate the conventional sorting technique let us consider an example where a list A={10, 6, 8, 2, 4, 11} contains unordered set of integer numbers.



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



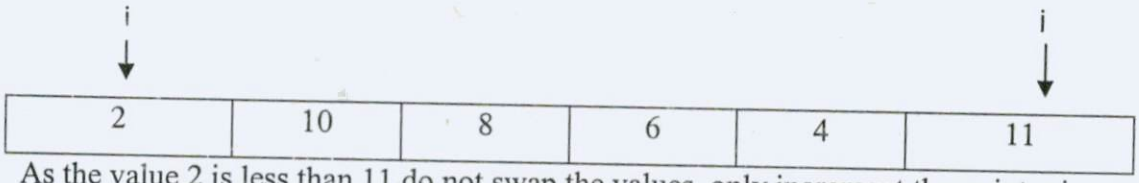
As the value 6 is less than 8 do not swap the values, only increment the pointer j



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

2	10	8	6	4	11

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

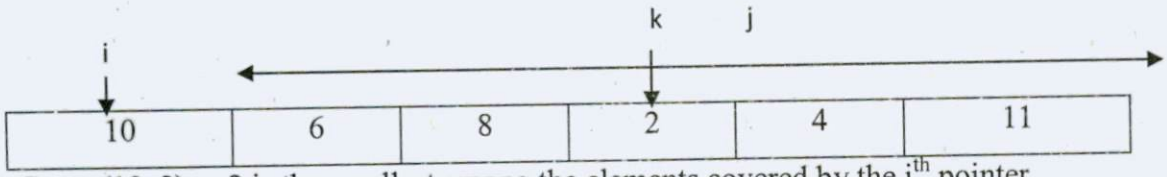
If we assume that each step will take 1 unit of time, then the total time taken to sort the considered 6 elements is 5+4+3+2+1 = 15 unit of times. In general if there are n elements in the unordered list the time taken to sort is $(n-1)+(n-2)+(n-3)+...+3+2+1=\frac{n^2-3n+2}{2}$.

21.3 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

21.3 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).

21.5 APPLICATIONS OF SORTING

Applications of sorting can find in many areas. Some applications of sorting are

(1). Speeding up the searching process is one of the most important applications of the sorting technique. Example. To search an element for its existence in any list, applying sorting prior to searching makes the process to work faster.

(2). In many of the statistical procedures sorting is necessary. Example, to find median of any set of elements sorting is must.

(3). To find duplicate elements present in the list.

21.6 SUMMARY

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

21.7 KEYWORDS

(1). Sorting technique

(2). Conventional sort

(3). Selection sort

(4). Insertion sort

21.8 QUESTIONS

(1). Define sorting. Mention the applications of sorting with respect the computer science field.

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

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

(4). 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

(5). 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.

21.9 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 - 22

BINARY SEARCH BASED INSERTION SORT, MERGE SORT, QUICK SORT

Structure

22.0 OBJECTIVES

After reading this unit you should be able

- Discuss the basic concepts and implementation details of all the three techniques
- Simulate the algorithms on any given set of data

22.1 BINARY SEARCH BASED INSERTION SORT

Binary search based insertion sort is 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.

22.2 MERGE SORT

The basic concept of merge sort is 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 the two sets into one common set.

We first look into the concept of arranging two individually sorted series of numbers into a common series 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 second 5. 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 compare A(1) =3 with B(2) =6. Since A(1) is smaller then B(2), A(1) will become the second element of C. C[ ] = {2, 3}

Similarly compare A(2) with B(2), since A(2) is smaller, it will be the third element 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 & B are originally sorted. Then only they can be merged. But, how do we sort them in the first? To do this and show the consequent merging process, we look at the following example. Consider the series A= (7 5 15 6 4). Now divide 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 with 15 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 and 15). This is the sorted list.

You are now expected to take different sets of examples and see that the method always works.

We design two algorithms in the following. The main algorithm is a recursive algorithm (somewhat similar to the binary search algorithm that we saw earlier) which calls at times the other algorithm called MERGE. The algorithm MERGE does the merging operation as discussed earlier.

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

Algorithm: Merge

Input: low, mid, high, limits of two lists to be merged i.e., A(low, mid) and A(mid+1, high)

A, the list of elements

Output: B, the merged and sorted list

Method:

```
h = low, i = low, j = mid + 1;
While ((h dŠ mid) and (j dŠ high)) do
        If (A(h) dŠ A(j) )
                B(i) = a(h);
                h = h+1;
        else
                B(i) = A(j);
                j = j+1;
        If end
        i = i+1;
        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

While end

Algorithm ends

22.2 QUICK SORT

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 an arbitrarily "pivot" place and ensures that all elements to the left of the pivot element are lesser than the element 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 is what we have added and is not a part of the list.

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

Now consider the first element. We want to move this element 75 to its correct position in the list. At the end of the operation, all elements to the left of 75 should be less than 75 and those to the right should be 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.

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

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

process.

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

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

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

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

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

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

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

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

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

We have completed one series of operations. Note that 75 is at its proper place. All elements to its left are lesser and to it's 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. The
main 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

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: 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) e> v);
                Repeat
                p = p - 1
                Until (A(p) d" v);
        If (i < p)
                INTERCHANGE(A(i), A(p))
        If end
        Until (i e" p)
        A(m) = A(p) ;
        A(p) = v;
```

Algorithm ends

22.3 SUMMARY

In this unit we have studied three sorting techniques such as binary search based insertion sort, merge sort and quick sort. Each sorting technique has been illustrated with an example and designed corresponding algorithm.

22.4 KEYWORDS

(1). Binary search based insertion sort
(2). Merge sort
(3). Merge
(4). Quick sort
(5). Partition algorithm

22.5 QUESTIONS

(1). Design an algorithm to sort given set of n numbers using binary search based insertion sort.
(2). Design an algorithm to sort given set of n numbers using merge sorting technique
(3). Design an algorithm to sort given set of n number using quick sort.

(4). 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.

(5). Consider a set A = {45 1 69 85 23 4 89 56 96 54 2}. With necessary steps demonstrate how merge sort works.

(6). Consider a set A = {56 89 74 12 63 5 9 21 66 69 1}. Show the working principle of quick sort technique.

22.6 REFERENCES

(1). Anany Levitin. The design and analysis of algorithms. Pearson Education

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

UNIT- 23

HEAP SORT, BUCKET SORT

Structure

23.0 OBJECTIVES

After reading this unit you should be able to

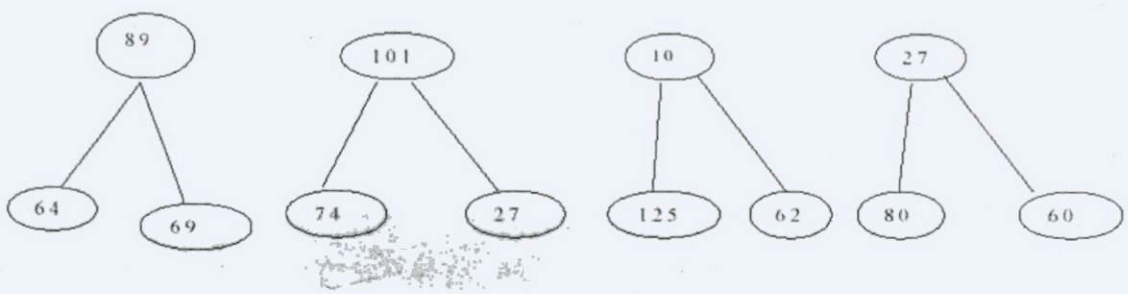
- Discuss the basic concepts and implementation details of heap sort.
- Explain the basic concepts and implementation details of bucket sort

i

23.1 HEAP SORT

We now look at the concept of heap sort - a method of arranging numbers in ascending order. A heap is defined as a collection of numbers, normally arranged in the form of a tree - A binary tree to be more precise (students can look into a later block to look at the concept of binary trees). The condition is that the parent node is always larger than the child nodes. For example in the following figures the first two represent heaps, while the others do not.

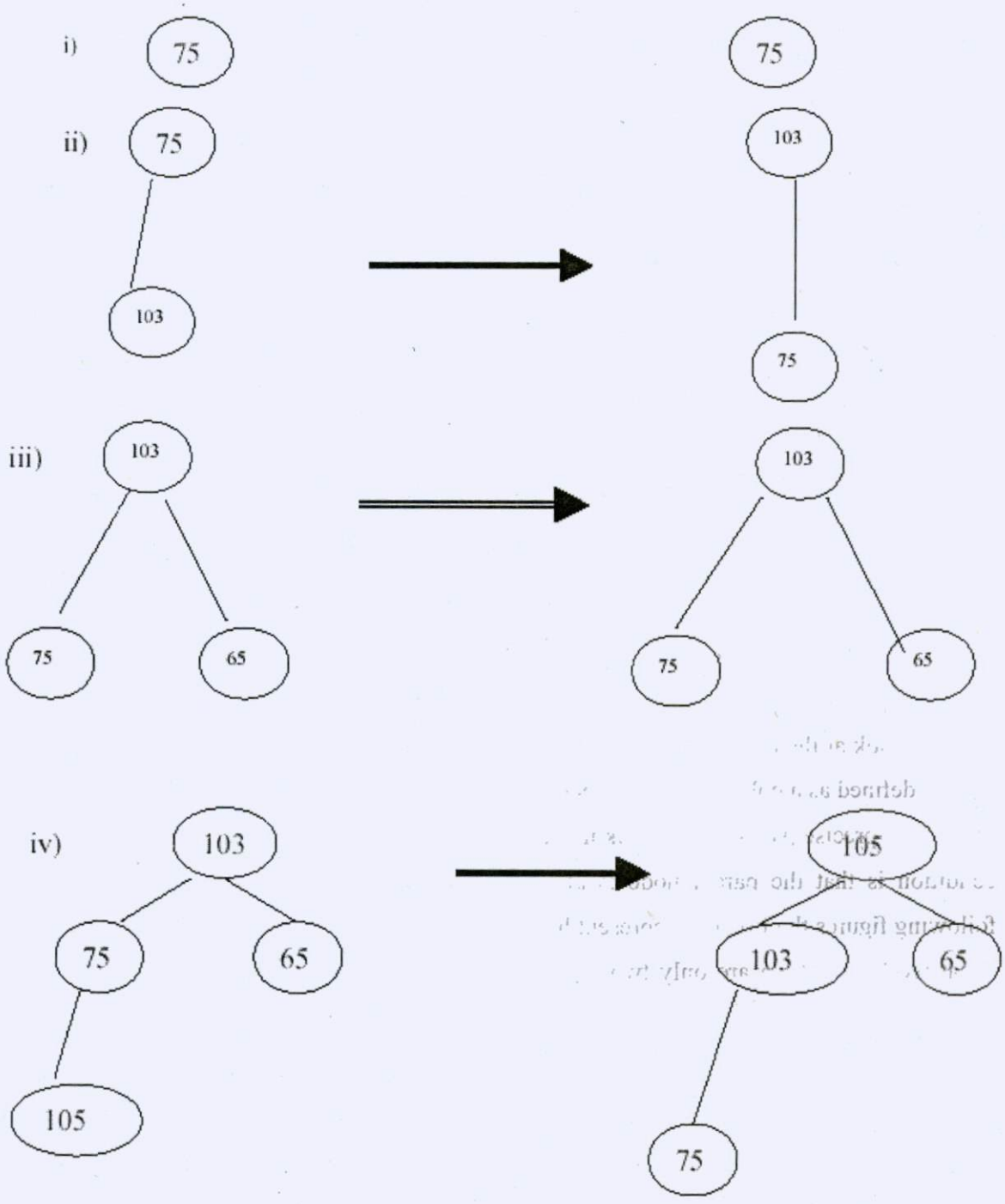
What we have shown are only two levels, but the same can be rearranged for any number of



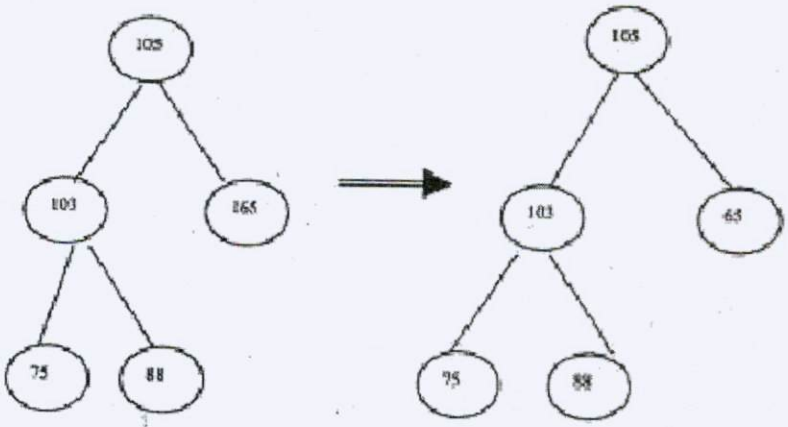
levels and any number of elements.

To insert an element into the heap, one adds it "at the bottom" of the heap and then compares it with its parent, grandparent, great grandparent and so on, until it is less than or equal to one of these values.

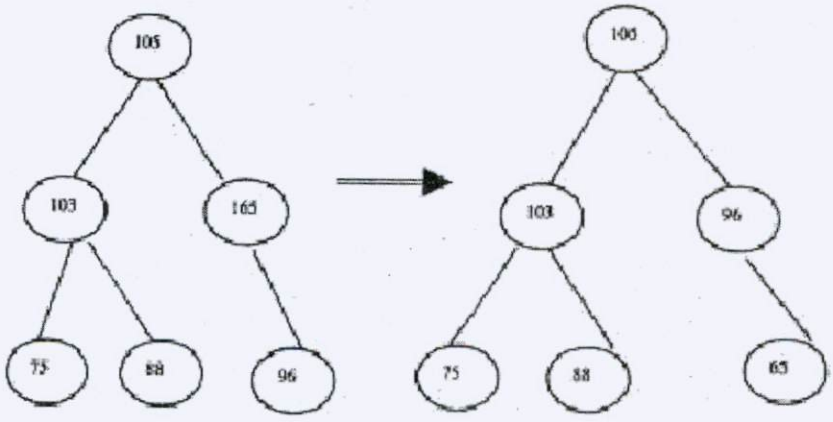
Let us consider the incoming numbers in the sequence (75, 103, 65, 105, 88, 96, 100). Now we want to construct a heap for the same

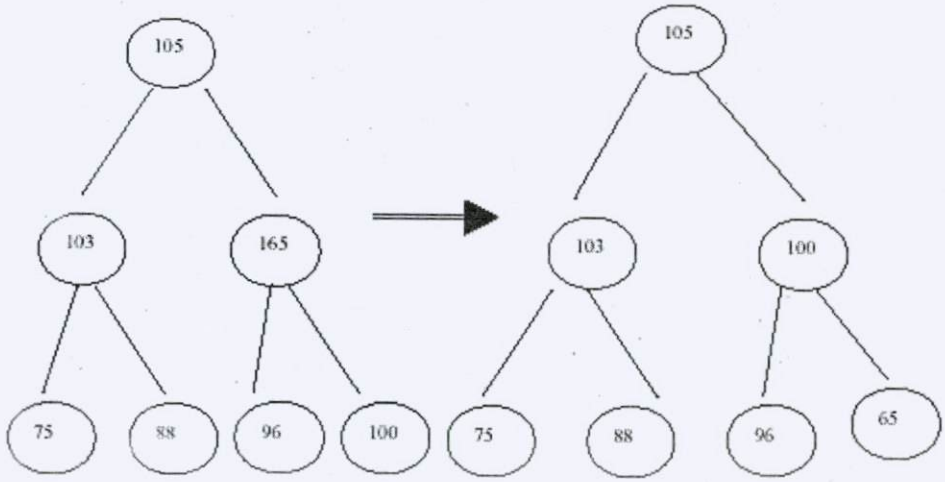


v)


105
103
165
75
88
165
103
65
75
88

vi)


105
103
165
75
88
96
100
103
96
75
88
65


105
103
165
75
88
96
100
105
103
100
75
88
96
65

Note that though we have ended up in a balanced tree in this case, it may not be so always. Now take out the largest element 105, put it at the end of the sorted list and rearrange the heap. 103 comes to top, remove it, rearrange the heap again etc.

We now write algorithms to do the same. The first one is the most important procedure that actually creates the heap. The other two call on this to produce the required sorted array.

Algorithm Insert

Input: A, An unordered set

N, size of set A

Output:

Insert A[n] into the heap which is stored in A[1: n-1]

Method:

```
i=n
item=A[n];
While ( (i>n) and (A[ i!/2 ] < item)) do
        A[ i ] = A[i/2];
        i =i/2;
While end
A[i]=item;
```

Algorithm end

Algorithm Adjust

Input: A, i, n

Output: Updated A

Method:

```
j=2i
item=A[i]
While (j<=n) do
        If ((j<=n) and (A[j] < A[j+1])) then
                j=j+1
        //compare left and right child and let j be the right //child
        If (item >= A[i]) then break
        // a position for item is found
        A[i/2]=A[j]
```

```
            j=2i
        While end
        A[j/2]=item;
Algorithm end
```

Algorithm: Delmax

Input: A n x

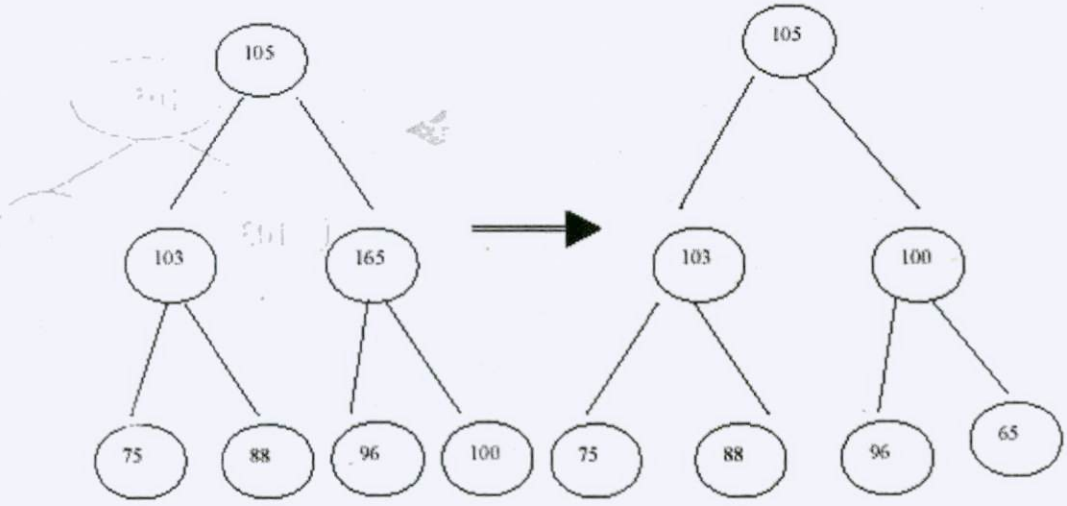
```
// Delete the maximum from the heap A[1..n] and store it in x
        If (n=0) then
                Write ('heap is empty");
        If end
        x=A[1];
        A[1] =A[n];
        Adjust (A, 1, n-1);
        Return (true);
```

Algorithm end

To delete the maximum key from the max heap, we use an algorithm called Adjust. Adjust takes as

input the array A[ ] and integer I and n. It regards A[1..n] as a complete binary tree. If the sub trees

rooted at 2I and 2I+1 are max heaps, then adjust will rearrange elements of A[ ] such that the tree rooted at I is also a max heap. The maximum elements from the max heap A[1..n] can be deleted



by deleting the root of the corresponding complete binary tree. The last element of the array, i.e. A[n], is copied to the root, and finally we call Adjust(A, 1, n-1).

Algorithm: sort

Input: Unsorted list - A, size of the list - n

Output: Sorted list A

Method:

```
For i=1 to n do
        Insert (A, i)
For end
For i= n to 1 step –1 do
        Delmax (A, i, x);
        A[i] =x;
For end
```

Algorithm end

23.2 BUCKET SORT

Bucket sort is possibly the simplest distribution sorting algorithm. The essential requirement is that the size of the universe from which the elements to be sorted are drawn is a small, fixed constant, say m.

For example, suppose that we are sorting elements drawn from **{0, 1, . . ., m-1}**, i.e., the set of integers in the interval **[0, *m*-1]**. Bucket sort uses m counters. The i^{th} counter keeps track of the number of occurrences of the i^{th} element of the universe. The figure below illustrates how this is done.

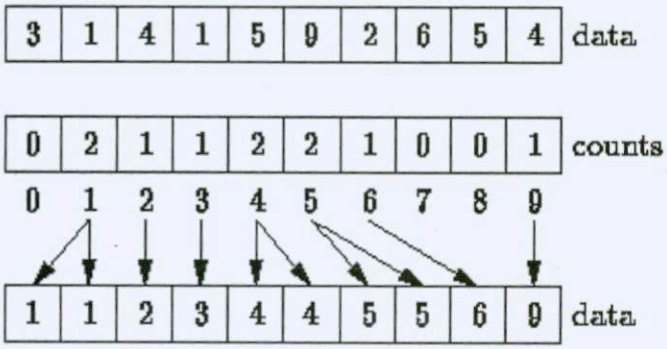


Figure: Bucket Sorting

In the figure above, the universal set is assumed to be **{0, 1, . . ., 9}**. Therefore, ten counters are required-one to keep track of the number of zeroes, one to keep track of the number of ones, and so on. A single pass through the data suffices to count all of the elements. Once the counts have been determined, the sorted sequence is easily obtained. E.g., the sorted sequence contains no zeroes, two ones, one two, and so on.

It works as follows:

1. Set up an array of initially empty "buckets" the size of the range.
2. Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket.
4. Put elements from non-empty buckets back into the original array.

23.3 SUMMARY

In this unit we have presented the heap sort and bucket sort. The corresponding algorithms are presented in the unit with suitable example.

23.4 KEYWORDS

(1). Heap

(2). Heap sort

(3). Bucket sort

23.5 QUESTIONS

(1). Design an algorithm to sort n number using heap sort. Consider a set A = {12,2,16,30,8,28,4,10,20,6,18}, illustrate the designed algorithm on set A.

(2). For the same unordered set given in question 1 sort using bucket sort

23.6 TEXT BOOK

(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 - 24

SEARCHING TECHNIQUE

Structure

24.0 OBJECTIVES

After reading this unit you should be able to

- Discuss the basic concept of searching techniques
- Explain linear search technique
- Elucidate binary search technique
- Design algorithms for depth first and breadth first search technique

24.1 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

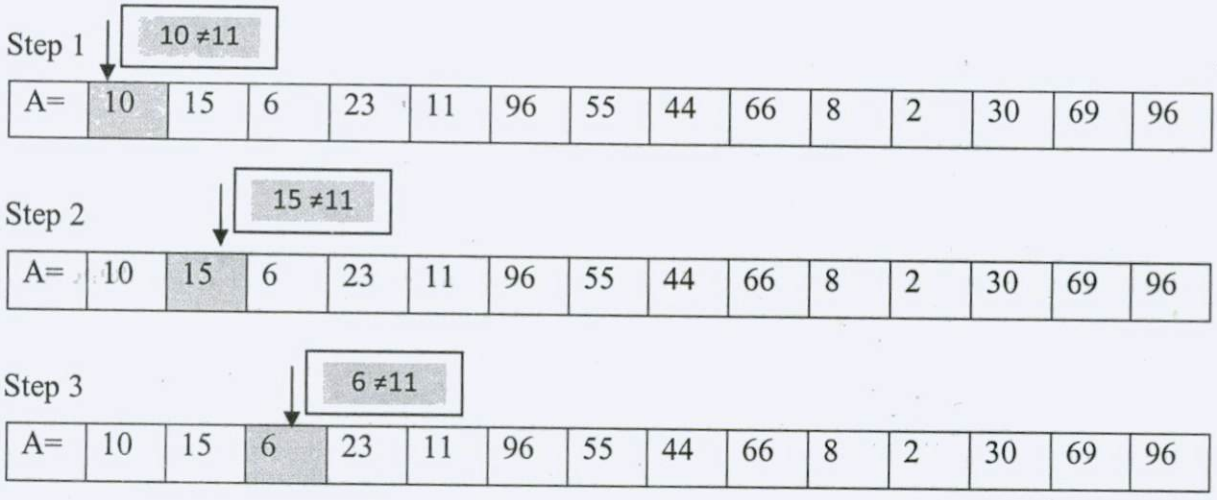
given list is already in sorted order. On the other hand searching techniques can be classified based on the data structures used to store the list. If array is used, then we must use different searching technique. Searching an element in a non linear data structure requires different searching techniques.

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.

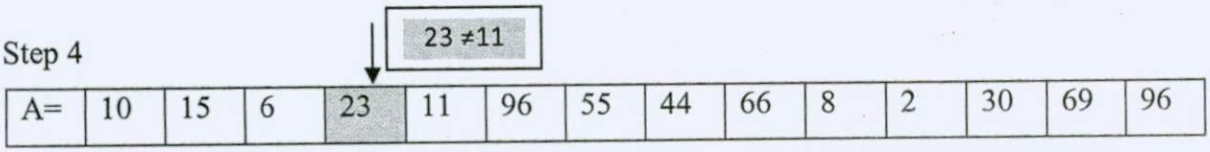
24.2 LINEAR SEARCH (SEQUENTIAL SEARCH)

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).

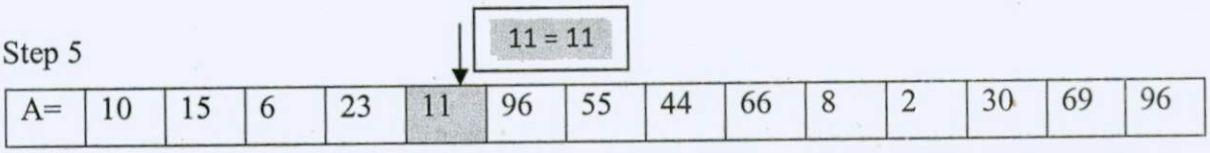
Let A = [10 15 6 23 8 96 55 44 66 11 2 30 69 96] and searching element 'e' = 11. Consider a pointer 'i', to begin with the process initialize the pointer 'i' = 1. Compare the value pointed by the pointer with the searching element 'e' = 11. As A(1) = 10 and its is not equal to element 'e' increment the 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 4



Step 5



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.

24.3 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.

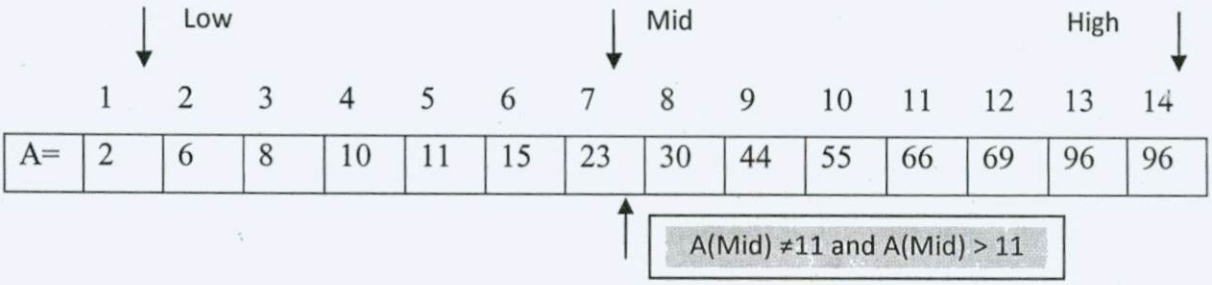
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A=	2	6	8	10	11	15	23	30	44	55	66	69	96	96

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

Mid = (1+14)/2 = 15/2 = 7.5 ≈ 7.

Check whether A(Mid) = = Search element.

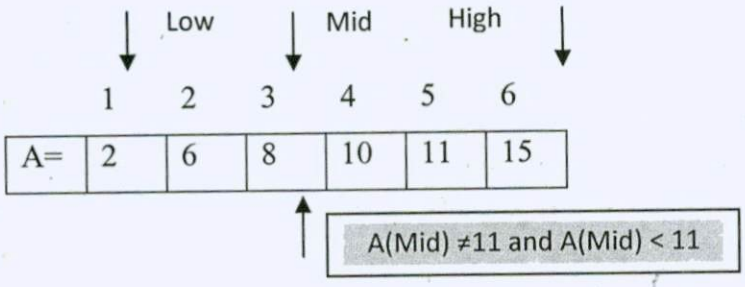
A(7) = 23 and it is not equal to search element 11.



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

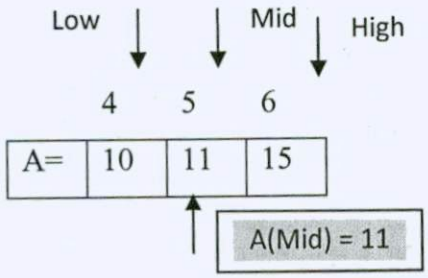
the new list and continue the same procedure neglecting the second half of the list. For further processing initialization are as follows

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

The techniques that were discussed in above two sections are only for linear type of structures (Specifically for arrays). In next two sections we present two different approaches called Depth first search and breadth first search algorithms, which works on graph data structures. Normally these two approaches are used to search a node in a graph or to traverse any undirected graph. Traversing a graph without revisiting the same node is a difficult task as it involves loops, circuits. In this unit we consider the problem of searching a node in a graph.

24.4 DEPTH FIRST SEARCH

Depth first search algorithm starts visiting nodes of a graph arbitrarily, marking that node as visited node. Soon after visiting any node (current node) we consider any of its adjacent nodes as next node for traversal and the current node address will be stored in stack data structure and traverse to the next adjacent node. The same thing is processed until no node can be processed further. If there are any nodes which are not visited then backtracking is used until all the nodes are visited. In depth first search stack will be used as a storage structure to store information about the nodes which will be used during backtracking.

Before knowing how to search a node in a graph using depth first search we need to understand how depth first search can be used for traversal of graph. Consider a graph G as shown in Figure 1(a). The traversal starts with node 1 (Figure 1(b)), mark the node as traversed (Gray shading is used to indicate that the node is traversed) and push the node

number 1 into the stack. As it has only one adjacent node 4 we will move to node number 4. Mark the node number 4 Figure 1(c) and push 4 into the stack. For node number 4 there are 2 adjacent nodes i.e., 2 and 5.

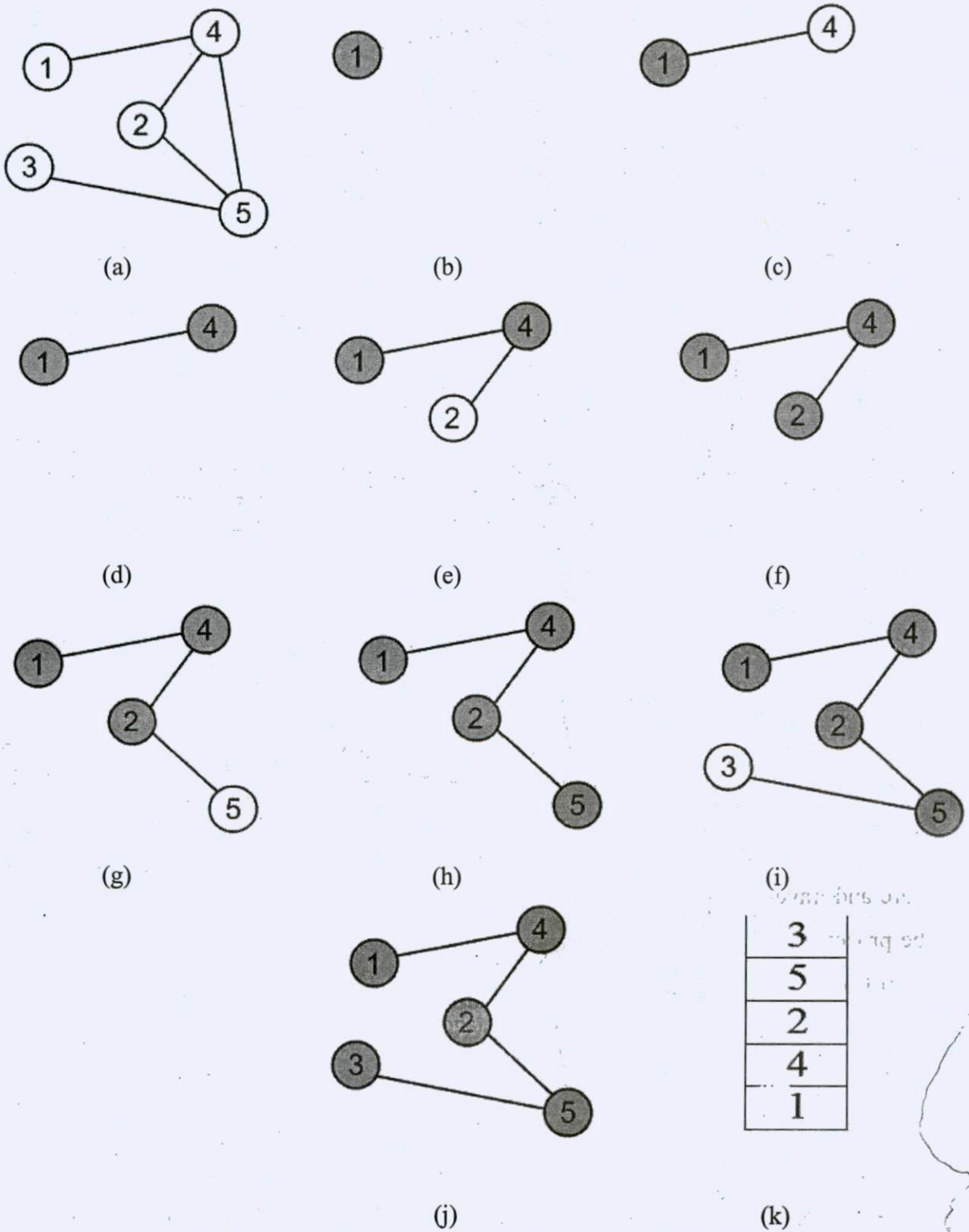
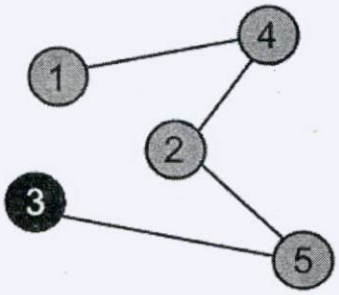
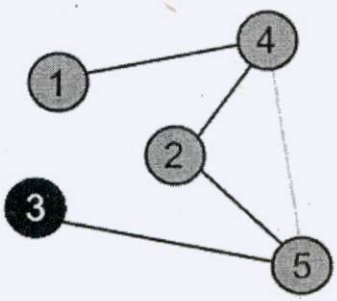


Figure 1: Traversal of a graph using depth first search algorithm

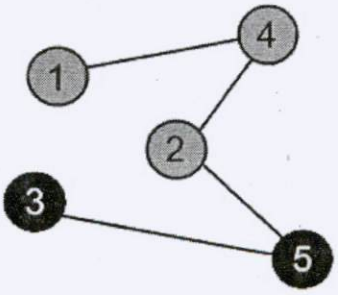
Select one node arbitrarily (For implementation purpose we can select the node with smallest number) and move to that node, in this case we will move to node 2 and push the node number 2 to stack. Similarly we will move to node 5 from node 2, pushing 5 into stack and then move to node 3 from node 5 and push node 3 into the stack (Figure (1(d, e, f, g, h, I, j)). Figure 1(k) shows the elements present in the stack at the end. From node 3 there is no possibility to traverse further. From this point onwards we will backtrack to check whether there are any other nodes which are not traversed. Pop the top node 3 from stack. Now, check is there any possibility to traverse from the element present in the top of the stack. The top element is 5 and there is an edge with has not been traversed from the node 5 (See Figure 2(b), the line marked in red color is untraversed edge). This edge leads to 4 which has been already visited and there is no other possibility for traversing from node 5, pop node 5 from the stack. Do the same process and at the end there will be no elements in the stack indicating that all the vertices of the graph have been traversed.



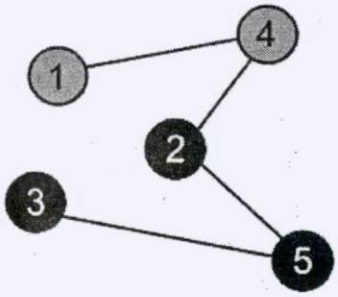
(a)



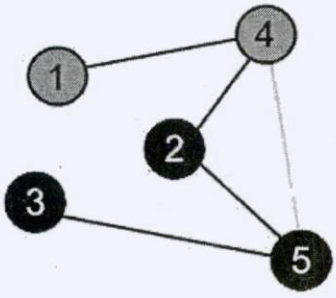
(b)



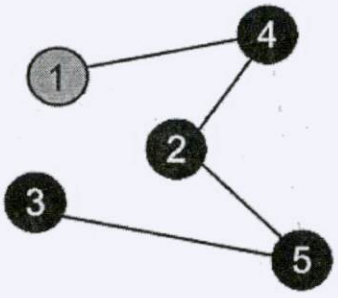
(c)



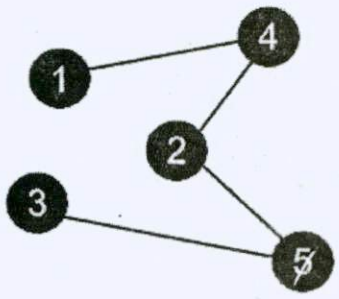
(d)



(e)



(f)



(g)

Figure 2. Backtracking operations for the depth first search algorithm

Figure 1 and Figure 2 demonstrated the depth first search for traversal purpose. The same technique can be used to search an element in the graph. Given a graph with n nodes we can search whether given node is present in the graph or not. Each time we visit a node we check whether that node is same as the search node, if it is stop the procedure declaring that the node is present else push that node into the stack and traverse until you the stack become empty.

Let us consider a tree example and illustrate the working principle of the depth first search. Let the search element be F.

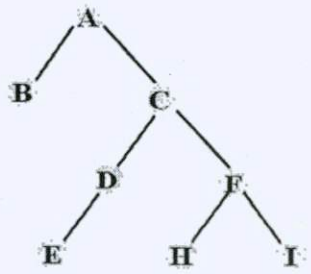


Figure 3. A binary tree with 8 nodes.

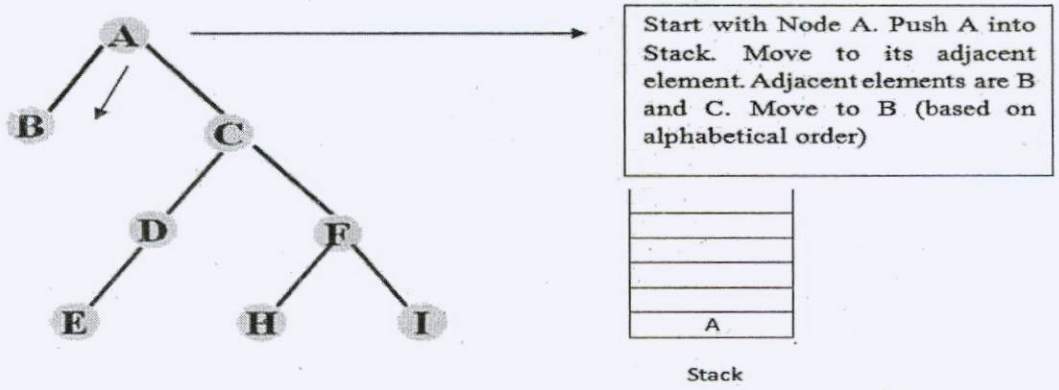


Figure 4(a)

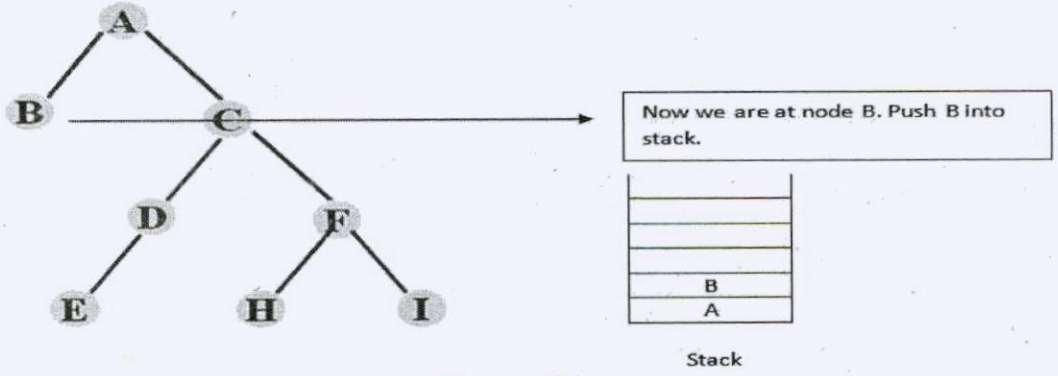


Figure 4(b).

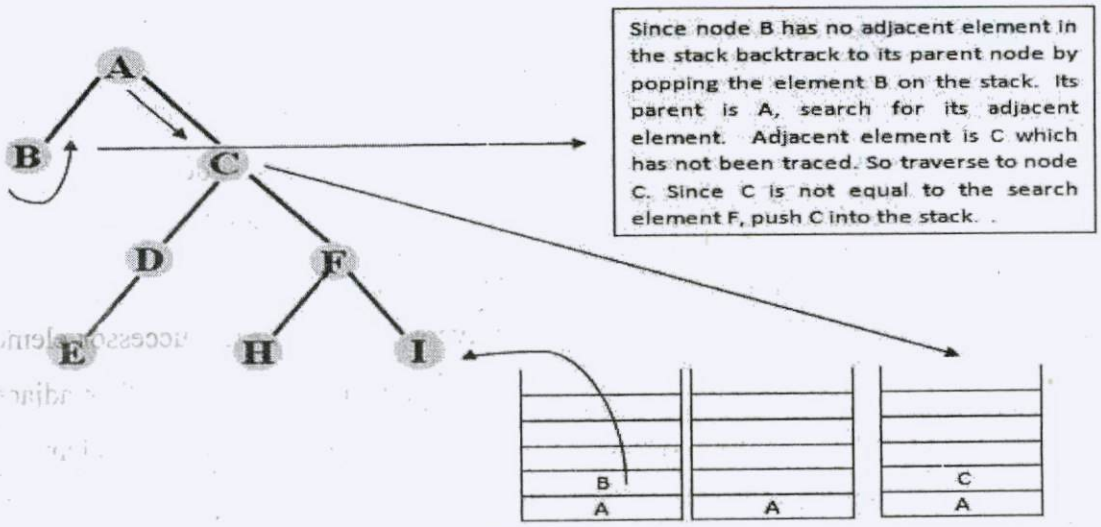


Figure 4(c)

hence remove D from the Queue. The next element in the Queue is F, find out is successor i.e., {H, I}. Insert them into the queue and mark them as visited. Once again the element F has no successor so remove it from queue and check for next element in the queue. The next element is E and E has no successor remove it and next elements are H and I. traverse them in the same way.

For searching an element using breadth first search, similar to depth first search we traverse the graph using breadth first traversal and while traversing the graph if a node same as search element occurs we declare that search element is present in the graph.

24.6 SUMMARY

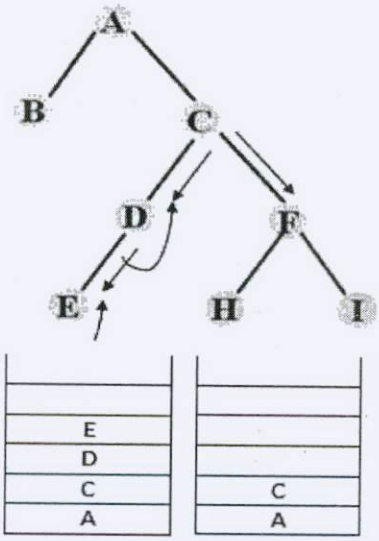
In this unit we have presented the basics of searching technique. We have presented four different searching techniques such as linear search, binary search, depth first search and depth first search. The first two are for conventional type of data whereas last two are for searching of elements stored in the form of graph.

24.7 KEYWORDS

(1). Searching technique
(2). Linear search
(3). Binary search
(4). Depth first search
(5). Breadth first search

24.8 QUESTIONS

(1). Design an algorithm to search an element 'e' from a given list using linear search technique.
(2). Design and develop an algorithm to find a given element 'e' using binary search method. Discuss how it is efficient than linear search method.
(3). Mention the difference between depth first search and breadth first search algorithm.
(4). Mention the applications of searching algorithms?
(5). Consider a list A= {12,2,16,30,8,28,4,10,20,6,18}. Check whether element 6 is present in the list using binary search technique. Illustrate the searching technique



Node C has two adjacent elements D and E, Move to D. Check whether D is equal to search elemen: F. Since D not equal to F move its adjacent element i.e., E. Push D into the Stack. Since E is not equal to its search element check for its adjacent element. Since E has no adjacent elements backtrack. Pop element E from stack. We have D in the top of the stack. Check whether D has any other untraversed node. Since we do not have any untraversed node from D back track to its parent node C by popping top element D. From C we can move to Node F. Node F is equal to search element. Search element is found stop the process.

Figure 4(d)

Figure 4(a) – 4(d) : Various steps in depth first search algorithm.

Note: Depth first search method uses stack as a data structure.

24.5 BREADTH FIRST SEARCH

Analogous to depth first search which search the nodes from top to bottom fashion postponing the traversal of adjacent elements, the breadth first search algorithm first traverse adjacent nodes of a starting node, then all unvisited nodes in a connected graph will be traversed in the same manner.

It is convenient to use a queue to trace the operation of breadth first search. The queue is initialized with the traversal's starting node, which is marked as visited. On each iteration, the algorithm identifies all unvisited nodes that are adjacent to the front node, marks them as visited, and adds them to the queue; after that front node is removed from the queue.

Let us consider the same example of tree traversal Figure 3.

Starting node is A, Insert A into queue mark A as traversed. Move to its successor element {B, C}, push them to queue and mark them as traversed. Since there is no other adjacent element to node A, remove A from which is first element in the queue. The next element in the queue is B, check for its successor node. Since B has no successor elements remove B from the Queue. The next element in the queue is C, find its successor elements i.e., {D, F}. Insert them into the queue and correspondingly marks the, as traversed. Since C has no other elements as its successor remove C from the Queue. The next element in the queue is D, its successor is E insert it into the queue and mark it as traversed. Now, D has no successor node

24.9 REFERENCES

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

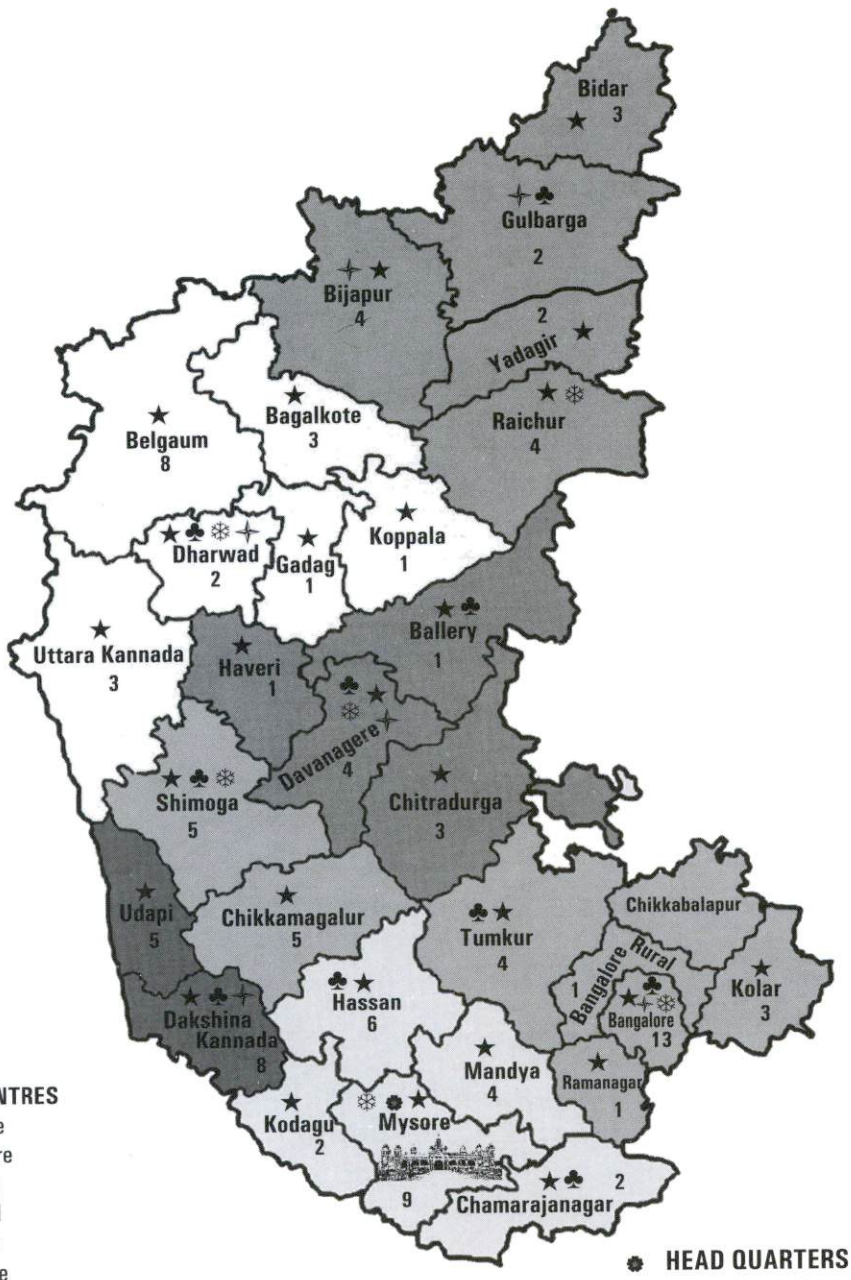
(2). Alfred V. Aho , Jeffrey D. Ullman, John E. Hopcroft. Data Structures and Algorithms. Addison Wesley (January 11, 1983)

NOTES

ಆದೇಶ ಸಂಖ್ಯೆ : ಕರಾಮುವಿ/ಸಿಪಾವಿ/2-1311/2010-2011 ದಿನಾಂಕ : 16/11/2010
ಒಳಪುಟ : 60 GSM ಮ್ಯಾಪ್ಲಿತೊ ಮತ್ತು ರಕ್ಷಾಪುಟ : 220 GSM ಆರ್ಟ್ ಕಾರ್ಡ್
ಮುದ್ರಕರು : ವಿನಾಯಕ ಆಫ್‌ಸೆಟ್ ಪ್ರಿಂಟರ್ಸ್, ಬೆಂಗಳೂರು – 560 070 ಪ್ರತಿಗಳು : 500

Karnataka State Open University

Manasagangotri Mysore - 570 006



♣ **REGIONAL CENTRES**

Bangalore
Davanagere
Gulbarga
Dharwad
Shimoga
Mangalore
Tumkur
Hassan
Chamarajanagar
Bellary

❁ **HEAD QUARTERS**

★ **Total Study Centres : 111**

♣ **Regional Centres : 10**

❄ **B.Ed Study Centres : 10**

⟡ **M.Ed Study Centres : 08**

VOP-05-316-500