A Practical Approach to Data Structures and Algorithms

SANJAY PAHUJA

New Age Science

A Practical Approach to

Data Structures and Algorithms

This page intentionally left blank

A Practical Approach to Data Structures and Algorithms

SANJAY PAHUJA
Software Engineer & Trainer

New Age Science Limited
The Control Centre, 11 A Little Mount Sion
Tunbridge Wells, Kent TN1 1YS, UK
www.newagescience.co.uk • e-mail: info@newagescience.co.uk

27 Old Gloucester Street, London, WC1N 3AX, UK
www.newacademicscience.co.uk • e-mail: info@newacademicscience.co.uk

ISBN : 978 1 78183 176 2

British Library Cataloguing in Publication Data

A Catalogue record for this book is available from the British Library

Preface

Every author of a textbook is expected to provide a justification for addition to the existing list at that point of time. I am no exception. I have been teaching "Data Structures and Algorithms" and related courses for over the past ten years, at several institutes. The audience was varied—science and engineering students; undergraduate and graduate students; working executives at junior and senior level.

A very large number of institutes teach a course on "Data Structures and Algorithms". While the core of what is covered in these courses remains the same, but the approaches used are different.

This book elaborates the solutions of the problem exercises of the different standard books. A large number of examples are used throughout the book to illustrate the points so as to convince the reader of its applicability. The book works out practically on data structures.

The aim of this book is to uplift the student self-driven programming in 'C' on data structure such as linked list, stack, queue, trees, graphs and sorting and searching methodology. Each and every topic is described with its complete method, algorithm, function and program.

The first chapter introduces the problems on the concept of data representation, fundamental of data structure, designing algorithm, analyzing algorithm and some evaluation procedure using mathematical notation.

The chapter two deals with elementary concept of 'C' as array, functions, recursions, structures and pointers in respect of programming of all operations on linear or non-linear data structure. This chapter builds confidence to the students, so they can write the program with the help of user-defined functions, pointers and structures.

Chapter three covers arrays, an elementary subject of data structures is dealt with unordered and ordered array with respect to all data structures operations such as insertion, deletion, and traversal. This chapter gives a complete idea of data structures for the subsequent chapters. This chapter also deals with various operations related to matrices.

Chapter four explores problems on linked list, which introduces the linear linked list, circular linked list, doubly linked list, header linked list and multilist. The operations such as insertion,

deletion, and traversal are performed on them. The applications of linked list such as polynomial manipulation, linked directory and multiple-precision arithmetic is introduced in very effective manner.

Chapter five covers algorithms on Stack. This section covers stack representation, stack operations—traversal, insertion, and deletion, multiple stacks. Applications of stacks such as recursion and polish expression and their compilation are discussed.

Chapter sixth deals algorithms on Queue. This section covers queue representation, queue operations—traversal, insertion, and deletion. The circular queue, double ended queues and priority queues are also illustrated.

Chapter seven focusses on algorithms on trees, their representation as sequential and linked list. The binary tree and their traversal schemes are covered in detail. Binary search tree, balanced binary tree, AVL-tree, multi-way, and B-tree are also discussed. The application of tree in searching and evaluating an expression tree is also covered.

Chapter eight explores graph and its application. It includes graphical representation, graph traversal methods, minimum spanning trees, transitive closure and shortest path algorithms.

Chapter ninth introduces sorting algorithms and their analysis. The internal and external sorting methods are discussed. Different sorting methods and their complexity are covered for Selection sort, Bubble sort, Insertion and Heap sort.

Chapter tenth covers searching algorithms and their analysis. It covers basic search techniques, sequential search, binary search, hashing and dynamic memory allocation method-best fit, first fit and worst fit.

Chapter eleventh deals with description of external files. A number of file organizations such as sequential, index-sequential, direct access and multiple-key are discussed and also covers concept of files handling in 'C'.

Appendix A covers conceptual problem solutions of the standard book unsolved problem, problem in the previous examinations of the subject in different universities and some level problems.

The book gives the basic understanding to write program on data structure and their usage in programming.

Sanjay Pahuja

Contents

Chapter 4: Linear Data Structure—Linked List

Chapter 5: Algorithms on Stack

Chapter 6: Algorithms on Queue

Chapter 7: Non-Linear Data Structure: Trees

Chapter 8: Non-Linear Data Structure: Graphs

This page intentionally left blank

Chapter 1

Introduction to Data Structures and Algorithms

1.1 INTRODUCTION TO DATA REPRESENTATION

A computer is a machine that manipulates data. The prime aim of data structures includes the study of how data is organized in a computer, how it can be manipulated, how it can be retrieved, and how it can be utilized.

The basic unit of memory is the bit whose value is either 1 or 0. A single bit can represent the boolean state of result, fail or pass. If we need to represent more than two states in the information then it would require more than 1-bit to represent the information. The data may be numerical data or character strings data or mixed of both.

A high-level programming language defined several data types to represent all types of data in the memory. The memory of a computer is represented by collection of memory addresses. This address is usually numeric and represented in hexadecimal numbers, but for simplicity we shall represent memory addresses in decimal numbers. An address is often called a location, and the contents of the location are the values of data. A location content may be 1-byte, 2-byte, 4-byte or 8-byte long. Every computer has a set of "native" data types and implementation of their operations such as arithmetic operations, addition, subtraction and relational operations. For example, addition of two binary integers is implemented using full adder or half adder logic circuit.

Data Type

A data type is a collection of values (e.g. numeric, character strings) and a set of operations on those values. This mathematical construct may be implemented through hardware or software. But first of all this mathematical construct must be stored in memory locations and then manipulate the required operations and place the result in some memory locations. We can define data type as:

Data Type = Permitted Data Values + Operations

The computer implement operations on binary values. Example is addition of two binary numbers. We can use more effective software implementation on numeric and character strings values through high-level languages such as 'C'.

'C' language supports different types of data, each of which may be represented differently within the computer's memory. Basic or automatic or primitive data types are listed below and their typical memory requirements are also given. Note that the memory requirements for each data type may vary from one C compiler to another.

Data type	*Description*	*Memory requirements*
int	integer number	2-byte
char	single character	1-byte
float	floating-point number (i.e., a number containing a decimal point and/or an exponent)	4-byte
double	double-precision floating-point number (i.e., exponent which may be larger in magnitude)	8-byte

Other data types can be built from one or more primitive data types. The **built-in** data types are enumerated, array, pointer, structure and union. The new structured data types can build from these primitives and built-in data types are linked lists, stacks, queues, trees and graphs and many more. Sometime data types are called data structures.

Consider the following 'C' language declaration:

```
int   a, b, c;
float x, y, z;
```

This declaration reserved six memory locations for six different values. These six locations may be referenced by the identifiers a, b, c, x, y, and z. The contents of the locations reserved for a, b, and c will be interpreted as integers, whereas the contents of the locations reserved for x, y, and z will be interpreted as floating-point numbers. The values of these identifiers are within the allowed range.

Now consider the addition operation on these values declared as:

```
c = a + b;
z = x + y;
```

The compiler that is responsible for translating 'C' programs into machine language will translate the "+" operator in the first statement into integer addition, and place the result into identifier c.

Similarly the "+" operator in the second statement into floating-point addition and place the result into identifier z.

The data types and their operations defined in 'C' language are useful in problem solving. Each programming language has some limitation in representing their values. For example in 'C' primitive data types can only take the range of values according to their memory requirements. It is impossible in 'C' to represent arbitrarily large integers on a computer, since the size of such a machine's memory is finite.

The data types and their operations are implemented through hardware and software data structures. The efficiency of implementation is measured in terms of time and memory space requirements. If a particular application has heavy data processing task, then the speed at which those operations can be performed will be the major factor to evaluate performance. Similarly, if application uses large numbers of structures then the memory space to represent these data structures is the major factor in the efficiency criteria.

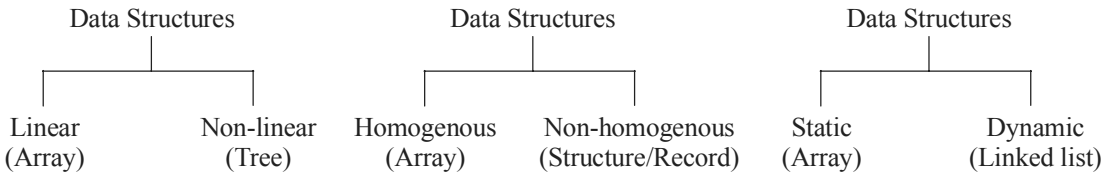
1.2 REVIEW OF DATA STRUCTURES—ARRAY, POINTER, STRUCTURE, LISTS, TREES AND GRAPHS

The **data structure** is a collection of data elements organized in a specified manner in computer's memory (or sometimes on a disk) and there exists an efficient method to store and retrieve individual data elements. Algorithms manipulate the data in these structures in various ways, such as inserting a new data element, searching for a particular element, or sorting the elements. Generally, the data element is of specific data types. We may, therefore, say data are represented that:

Data Structure = Organized Data + Allowed Operations

This is an extension of the concept of data type.

Designing and using data structures is an important programming skill. The data structures can be classified in three ways.



1. *Linear and Non-linear data structures*: In linear data structures the data elements are arranged in a linear sequence like in an array, data processed one by one sequentially. Linear data structures contain following types of data structures:
   - Arrays
   - Linked lists
   - Stacks
   - Queues

   In non-linear data structures, the data elements are not in sequence that means insertion and deletion are not possible in a linear manner. Non-linear data structures contain following types of data structures:
   - Tree
   - Graph
2. *Homogenous and Non-homogenous data structures*: In homogenous data structures the data elements are of same type like an array. In non-homogenous data structures, the data elements may not be of same type like structure in 'C'.
3. *Static and Dynamic data structures*: Static structures are ones whose sizes and structures associated memory location are fixed at compile time, e.g. array. Dynamic structures are ones that expand or shrink as required during the program execution and their associated memory location change, e.g. linked list.

Data Structure Operations

Algorithms manipulate the data that is represented by different data structures using various operations. The following data structure operations play major role in the processing of data:

- *Creating*. This is the first operation to create a data structure. This is just declaration and initialization of the data structure and reserved memory locations for data elements.
- *Inserting*. Adding a new data element to the data structure.
- *Updating*. It changes data values of the data structure.
- *Deleting*. Remove a data element from the data structure.
- *Traversing*. The most frequently used operation associated with data structure is to access data elements within a data structure. This form of operation is called traversing the data structure or visiting these elements once.
- *Searching*. To find the location of the data element with a given value, find the locations of all elements which satisfy one or more conditions in the data structure.
- *Sorting*. Arranging the data elements in some logical order, e.g. in ascending or descending order of students' name.

- *Merging*. Combine the data elements in two different sorted sets into a single sorted set.
- *Destroying*. This must be the last operation of the data structure and apply this operation when no longer needs of the data structure.

Here, we are exploring some real world applications to understand requirements of data structure operations.

Consider AddressBook, which holds a student's name, address and phone number. Choose the appropriate data structures to organized data of AddressBook and then create these structures. The creating of data structure operation stores some initial data in computer's memory. The following operation may require in the AddressBook:

- To insert new students' name, address and phone number in the AddressBook
- To delete old students' name, address and phone number in the AddressBook
- To update existing students' phone number
- To traverse all students' address for mailing
- To search particular phone number of the student
- To arrange AddressBook by student's name in ascending order

In a similar manner airlines/trains reservation system that stores passenger and flight/train information may require several data structures and their operations.

The complete details of these operations are given with each and every data structures in subsequent chapters.

1.3 WHAT IS AN ALGORITHM?

Definition

An algorithm is a finite set of instructions that takes some raw data as input and transforms it into refined data. An algorithm is a tool for solving a well-specified computational problem. Every algorithm must satisfy the following criteria:

- **Input:** In every algorithm, there must be zero or more data that are externally supplied.
- **Output:** At least one data is produced.
- **Definiteness:** Each instruction must be clear and unambiguous (i.e., clearly one meaning).
- **Finiteness:** If we trace out the instructions of an algorithm, then for all cases, the algorithm will terminate after a finite number of steps.
- **Effectiveness:** Every instruction must be feasible and provides some basis towards the solution.

An algorithm is composed of finite set of steps, each may require one or more operations. An algorithm requires the following key points in reference to data.

- Memory requirement for data.
- Language for describing data manipulation.
- The type of operations, which describes what kinds of refined data can be produced from raw data.
- Data structures for representing data.

In fact, a data structure and an algorithm should be thought of as a unit, neither one making sense without the other.

Algorithm Specification

An algorithm can be described in many ways. A natural language such as english can be used to write an algorithm but generally algorithms are written in english-like pseudocode that resembles with high level

languages such as 'C' and pascal. An algorithm does not follow exactly any programming language, so that it does not require any translator. Therefore, we can't run algorithm with the help of compiler/interpreter, just we can dry run to check the results. The following format conventions are used to write an algorithm:

1. **Name of algorithm:** Every algorithm is given an identifying name written in capital letters.
2. **Introductory comment:** The algorithm name is followed by a brief description of the tasks the algorithm performs and any assumptions that have been made. The description gives the name and types of the variables used in the algorithm. Comment begins with // and continues until end of line. Comments specify no action and are included only for clarity.
3. **Steps:** Each algorithm is made of a sequence of numbered steps and each step has an ordered sequence of statements, which describe the tasks to be performed. The statements in each step are executed in a left-to-right order.
4. **Data type:** Data types are assumed simple such as integer, real, char, boolean and other data structures such as array, pointer, structure are used. Array i^{th} element can be described as A[i] and (i, j) element can be described as A[i, j]. Structure data type can be formed as:

```
node =record
{
        data type-1        identifier 1;
        …
        data type-n        identifier n;
        node      *link;
}
```

   link is a pointer to the record type node.
5. **Assignment statement:** The assignment statement is indicated by placing equal (=) between the variable (in left hand side) and expression or value(s) (in right hand side). For example, addition of a and b is assigned in variable a.

```
a = a + b
```

6. **Expression:** There are three expressions: arithmetic, relational and logical. The arithmetic expression used arithmetic operator such as /, *, +, –, relational expression used relational operator such as <, <=, >, >=. <>, = and logical expression used logical operator such as not, or, and. There are two boolean values, true or false.
7. **If statement:** If statement has one of the following two forms:

   (*a*)
```
if  condition
then
statement(s)
```

   (*b*)
```
if  condition
then
statement(s)
else
statement(s)
```

   if the condition is true, statement(s) followed then are to be executed, otherwise if condition is false, statement(s) followed else are to be executed. Here we can use multiple statements, if required.
8. **Case statement:** In general case statement has the form:

   Select case (expression)

```
case value 1:   statement (s)
case value 2:   statement (s)
      .
      .
      .
case value n:   statement (s)
default:
```

the expression is evaluated then a branch is made to the appropriate case. If the value of the expression does not match that of any case, then a branch is made to the default case.

9. **Looping statements:** These statements are used when there is a need of some statements to be executed number of times. These statements also called iterative control statements.

(*a*)
```
while (condition)  do
{
    statement(s)
}
```

As long the condition is true, the statement(s) within while loop are executed. When the condition tests to false then while loop is exited.

(*b*)
```
for variable = start value to final value step increment value do
{
    statement(s)
}
```

This for loop is executed from start value to final value with addition of increment value in start value. In the absence of step clause, assume increment value 1.

(*c*)
```
repeat
{
statement(s)
} until (condition)
```

This repeat loop is executed until the condition is false, and exited when condition becomes true.

10. **Input and output:** For input of data it used read(variable name) statement and for output of data it used write(variable name) statement. If more than one input or output data then we can use comma as separator among the variable names.
11. **Goto statement:** The goto statement causes unconditional transfer of control to the step referenced. Thus statement goto step N will cause transfer of control to step N.
12. **End statement:** The end statement is used to terminate an algorithm. It is usually the last step and algorithm name is written after the end.
13. **Functions:** A function is used to return single value to the calling function. Transfer of control and returning of the value are accomplished by return(value) statement. A function begins as follows: function function_name (parameters list).

1.4 DESIGNING ALGORITHMS

To solve any problem on computer, first of all we have to design efficient algorithm of any computational problems. An algorithm may be an existing procedure, routine, method, process, function or recipe. But we can devise in new one that is more appropriate towards the solution of the problem in respect of memory

and execution time it required. The complete program can be smoothly and perfectly written in the supporting programming language with the help of the algorithm. When designing an algorithm, consider the following key points:

1. Machines that hold data.
2. Language for describing data manipulations.
3. Functions that describe what kinds of refined (e.g. output) data can be produced from raw (e.g. input) data.
4. Structures for representing data.

Effectiveness and efficiency of an algorithm also depends upon the data structures that it can use.

Now we are discussing some common problems and try to learn how to write an algorithm.

Example 1. Consider a problem to find the maximum of n numbers and its location in the vector, n>0.

For the simplicity of the algorithm, consider n integer numbers are stored in a vector A (i.e., single dimensional array).

For example n=10 and integer values of A[ ] = { 12, 43, 9, 3, 46, 25, 20, 15, 32, 39 } and large is a variable which holds always maximum number and j points to the location of maximum value element.

Initially assume that the value of large =A[1] has maximum value (i.e., 12). Compare the subsequent elements of vector A[i] with large and if the value of element is greater than large then assign those values to variable large and location to variable j. This method can be best described by below table.

A[i] →	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
Iteration	<12>									
1		<43>								
2			9							
3				3						
4					<46>					
5						25				
6							20			
7								15		
8									32	
9										39
j→	1	2	2	2	5	5	5	5	5	5

large value is enclosed in angle bracket. The maximum value is 46 and 5th element of vector A stores it.

```
Algorithm MAX(A, n, j)
// Set j so that A[j] is the maximum in A[1:n], and here i is integer, and n>0
Step 1:  large = A[1]; j = 1;
Step 2:  for i = 2 to n do
         {
              if A[i] > large
              then  large = A[i]; j = i;
         }
         end MAX
```

Algorithm 1.1 finding the maximum of n numbers

An algorithm can contain three kinds of variables: local, global and formal parameters. A local variable is one, which is declared in the current function (e.g. i is local variable). A global variable is one which has already been declared, as local to a function which contains the current function (e.g. large is global variable). Formal parameters contained in parameter list following the name of the algorithm. At execution time formal parameter are replaced by the actual parameters. In the above example, A, n , j are the formal parameters.

Example 2. *Euclid's algorithm*: A process for finding the greatest common divisor (GCD) of two numbers. This algorithm is for computing greatest common divisor of two non-negative integers. The essential step which guarantees the validity of this process consists of showing that the greatest common divisor of a and b ($a > b \geq 0$) is equal to a if b is zero and is equal to the greatest common divisor of b and remainder of a divided by b if b is non-zero.

For example, to find the greatest common divisor of two non-negative integers 22 and 8 can be computed as below:

gcd(22, 8) where a=22, b=8 and b is non-zero, so find remainder r= a mod b and assign a=b, and b=r and compute again.

gcd(8, 6) where a=8, b=6 and b is non-zero so find remainder r= a mod b and assign a=b, and b=r and compute again.

gcd(6, 2) where a=6, b=2 and b is non-zero so find remainder r= a mod b and assign a=b, and b=r and compute again.

gcd(2, 0) where a=2, b=0 and b is zero so greatest common divisor is value of a that is 2.

The algorithm can be easily designed on this method.

```
        Algorithm GCD(a, b)
// Assume a > b ≥ 0
Step 1 : if b = 0
        then return(a);
Step 2 : else
// r is the remainder
        r = a mod b;
Step 3 : set a = b; b = r; and go to  step1
        end GCD
```

Algorithm 1.2 Finding the greatest common divisor

Example 3. To design an algorithm for matrix multiplication of two m x n and n x p matrix A and B to form m x p matrix C.

The matrix multiplication of two matrices A and B is possible only when no. of column of matrix A must be equal to no. of row of matrix B.

That means

Matrix C	=	Matrix A	*	Matrix B
(of order m x p)		(of order m x n)		(of order n x p)
m- rows, p- columns		m- rows, n- columns		n- rows, p- columns

The matrix multiplication can be carried by the following formula:

$$C_{ik} = \sum_{j=1}^{j=n} A_{ij} * B_{jk} \quad \text{for } 1 \leq i \leq m \text{ , and } 1 \leq k \leq p$$

where

A_{ij} = Elements of Matrix A

B_{jk} = Elements of Matrix B

C_{ik} = Elements of resultant Matrix C

Consider an example to multiplication of matrix $A_{2,3}$ and matrix $B_{3,2}$. Here the value of m = 2, n = 3 and p = 2, the resultant matrix C must be order of 2 × 2 and matrices data are stored in two dimensional arrays.

C[1,1] = A[1,1] * B[1,1] + A[1,2] * B[2,1] + A[1,3] * B[3,1]

C[1,2] = A[1,1] * B[1,2] + A[1,2] * B[2,2] + A[1,3] * B[3,2]

C[2,1] = A[2,1] * B[1,1] + A[2,2] * B[2,1] + A[2,3] * B[3,1]

C[2,2] = A[2,1] * B[1,2] + A[2,2] * B[2,2] + A[2,3] * B[3,2]

```
Algorithm MAT_MULTIPLY (A, B, C)
// Matrix A order is m × n and Matrix B order is n × p
Step 1 : if number of columns of matrix A = number of rows of matrix B
         then
         for  i = 1 to m do
           for k = 1 to p do
            {
               C[i, k] = 0;
                for  j = 1 to n do
               C[i, k] = C[i, k] +  A[i, j] * B[j, k];

             }
Step 2 : else
         write("Matrix multiplication is not possible and exit");
         end MAT_MULTIPLY
```

Algorithm 1.3: Matrix multiplication

1.5 ANALYZING ALGORITHMS

Why is it necessary to analyze an algorithm? An algorithm analysis measures the efficiency of the algorithm. The efficiency of an algorithm can be checked by:

1. The correctness of an algorithm.
2. The implementation of an algorithm.
3. The simplicity of an algorithm.
4. The execution time and memory space requirements of an algorithm.
5. The new ways of doing the same task even faster.

The analysis emphasizes timing analysis and then, to a lesser extent, spaces analysis. It is very difficult to calculate how fast a problem can be solved. It requires an algorithm execution machine. An algorithm can be executed only of formal models of machines e.g. Turing Machine and Random Access Machine. An algorithm analysis does not depend on computers and programming languages but the program that is coded using the algorithm depends on both. Therefore, programs written in different programming languages using the same algorithm and same computer have different time and space requirements.

In an algorithm analysis, instructions of an algorithm are assumed to be executed one at a time, and so major time of an algorithm depends upon the number of operations it requires.

Given an algorithm to be analyzed, the first task is to determine which operations are involved and what their space and timing requirements. These operations may be arithmetic, comparisons, and assigning values to variables and executing function calls. For the simplicity of analysis, consider that all operations take a fixed amount of time and so their time is bounded by a constant.

An algorithm analysis must cover the sufficient number of data sets that cause the algorithm to exhibit all possible patterns of behaviour.

This requires each statement frequency counts (i.e., the number of times the statement will be executed) and the time for one execution. The total time required to execute that algorithm is the sum of product of these two numbers. But the execution time for each statement depends on both the computer and the programming language.

Here we are determining only frequency counts of each statement of the following codes:

```
x = x + y                    for i = 1 to n do          for i = 1 to n do
                               x= x + y                 for j = 1 to n do
                                                          x= x + y
```

(a) (b) (c)

For each code a statement x = x + y is executed. In code (a), the frequency count is 1. For code (b) the count is n and for code (c) the count is n^2. These are in increasing order of magnitude. The order of magnitude of an algorithm refers to the sum of the frequency counts of all statements of an algorithm. There are several kinds of mathematical notation to represent order of magnitude. One of these is the O (big oh) notation.

The time needed by an algorithm is expressed as a function of the size of a problem is called the time complexity of the algorithm. If an algorithm processes input of size n in time cn^2 for some constant c, the time complexity of the algorithm is $O(n^2)$, read as 'order of n^2'. The time complexity of an algorithm can be defined in terms of two functions. The function f(n) computes the actual time to execute an algorithm and function g(n) representes the time required by the algorithm.

The notation f(n) = O(g(n)) (read as f of n equals big-oh of g of n) has a precise mathematical definition, f(n) = O(g(n)) iff (if and only if) there exist two constants c and n_0 such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$.

The O(g(n)) means that if an algorithm is run on same computer on same type of data but for increasing values of n, the resulting times will always be less than some constant time |g(n)|.

The computing time O(1) is called constant, $O(\log_2 n)$ is logarithmic, O(n) is called linear, $O(n^2)$ is called quadratic, $O(n^3)$ is called cubic, and $O(2^n)$, O(n!) and $O(n^n)$ are called exponential. If an algorithm takes $O(\log_2 n)$ time, it is faster for sufficiently large n, than if it had taken O(n) time. The order of computing times for algorithms are $O(1) \leq O(\log_2 n) \leq O(n) \leq O(n \log_2 n) \leq O(n^2) \leq O(n^3) \leq O(2^n) \leq O(n!) \leq O(n^n)$ for sufficiently large n.

If we have two algorithms which perform the same task, and first has a computing time O(n) and the second one has $O(n \log_2 n)$, then first one is preferred to second one.

To clarify some of these ideas, let us consider an example to compute the n^{th} Fibonacci number. The Fibonacci series starts as

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Each new number is obtained by taking the sum of the two previous numbers in the series. The first number is $F_0 = 0$ and second number is $F_1 = 1$ and in general

$$F_n = F_{n-1} + F_{n-2}$$

The algorithm Fibonacci inputs any non-negative integer n and computes the n^{th} number in the Fibonacci series i.e., F_n.

```
        Algorithm FIBONACCI
        // compute the Fibonacci number Fn
Step 1 : read(n)
Step 2 : if (n< = 1) then
         write(n)
         else
         {
Step 3 : f0 = 0 ; f1 = 1;
Step 4 : for i = 2 to n do
         {
         fn = f0 + f1;
         f0 = f1;
         f1 = fn;
         }
Step 5 : write (fn);
         }
Step 6 : end FIBONACCI
```

Algorithm 1.4 compute the fibonacci number f_n

To analyze the time complexity of this algorithm, we need to consider two cases: (i) n = 0 or 1, and (ii) n > 1. We shall count the total number of statement executions and use this as a measure of the time complexity of the algorithm.

When n = 0 or 1, step 1, step 2 and step 6 get executed once. The step 2 has two statements. Therefore total statement count for this case is 4 (i.e., 1 + 2 + 1 respectively).

When n > 1, step 3 has two assignment statements and get executed once. For loop of the step 4 gets executed n times while the statements in the loop get executed n – 1 times each. The contributions of each step are summarized below for n > 1.

Step 1 :	Frequency count by the statement	Total count for the step
Step 2 :	if condition always false for when n > 1,	n – 1
Thus required n – 1 counts		
Step 3 :	one for each statement	2
Step 4 :	n by for loop and n – 1 by each of three statements.	4n – 3
Step 5 :	1 for write statement and 1 for end of else	2
Step 6 :	1	1

Hence the total count when n > 1	=	5n + 1
And total count when n < = 1	=	2
Thus total count for the algorithm is	=	5n + 3

The order of magnitude of the algorithm can be written as O(n), ignoring the two constants 5 and 3.

The performance of the different order of algorithms, their time complexity and their function values are computed in the below table.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4,294,967,296

The rates of growth of functions are shown in Fig. 1.1. An algorithm that is exponential will work only for very small inputs. The time complexity of order $O(\log_2 n)$, $O(n)$ and $O(n \log_2 n)$ grows much more slowly than the other three. For the large data sets (i.e., value of n is sufficiently large), the algorithms with a complexity greater than $O(n \log_2 n)$ are often impractical. For small n, the exponential function is less time consuming than $O(n^2)$ and $O(n^3)$. Up to n=4, $O(2^n)$ function has less or equal time required as by $O(n^2)$ and up to n=8, $O(2^n)$ function has less time required as by $O(n^3)$.

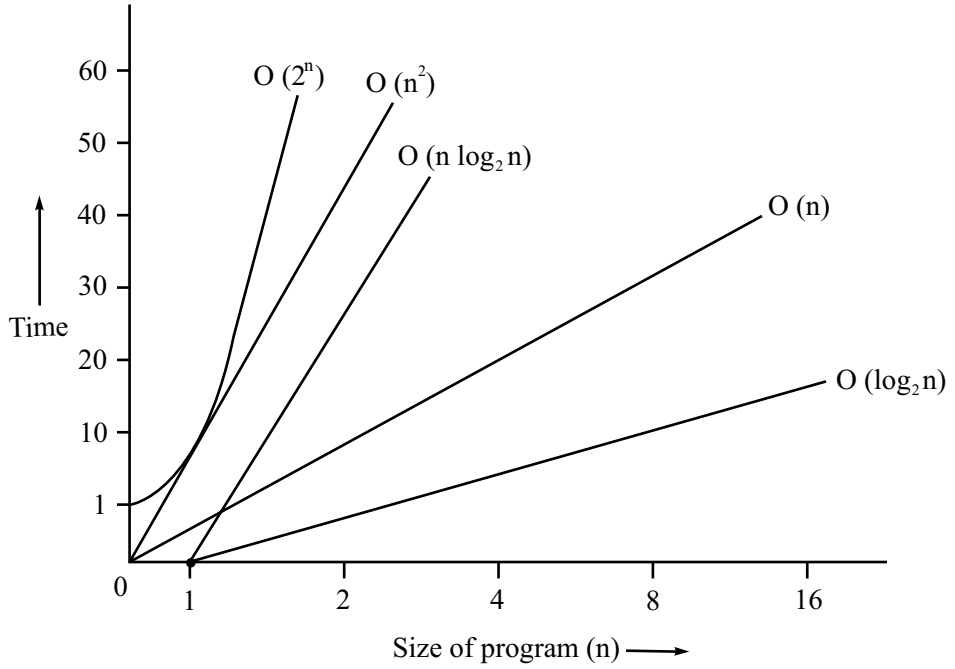


Figure 1.1 *Plot of function values*

1.6 MATHEMATICAL NOTATION AND FUNCTIONS

Summation symbol: Here we introduce the summation symbol Σ (Sigma). Consider a sequence of n-terms $a_1, a_2, \ldots a_n$, then the sums $a_1 + a_2 + \ldots + a_n$ will be denoted as

$$\sum_{1 \le i \le n} a_i$$

Examples are sums of n natural numbers, square of n positive integers and many more. We can write an expression as a sum of series or sequence. We can also call summation of summation.

Consider that a function f(n) denotes the summation of n positive integers. Here function f(n) computes the order of magnitude of an algorithm.

$$f(n) = \Sigma i = 1 + 2 + \ldots + n$$
$$1 \le i \le n$$
$$= n(n+1)/2, \text{ and the order of complexity is } O(n^2)$$

and in similar fashion the summation of square of n positive integers can be expressed as:

$$f(n) = \Sigma i^2 = 1^2 + 2^2 + \ldots + n^2$$
$$1 \le i \le n$$
$$= n(n+1)(2n+1)/6, \text{ and the order of complexity is } O(n^3)$$

Now we want to prove that f(n) is true for all positive numbers n. A mathematical induction does this in following manners.

(*a*) Give a proof that f(1) is true.

(*b*) Give a proof that "If all of f(1), f(2), … ,f(n) are true, then f(n+1) is also true".

This proof should be valid for any positive integer n.

Example 1. Let us consider an example: summation of n odd integer numbers in the following series:

$$1 + 3 + \ldots + (2n - 1) = n^2$$

We can denote as sum of n odd integers

$$\text{Sum} = \Sigma i = n^2$$
$$1 \le i \le 2n, \text{ where i is odd integer}$$

Now consider the following cases:

For n = 1, sum = 1 i.e., 1^2

For n = 2, sum = 1 + 3 i.e., 2^2

For n = 3, sum = 1 +3 + 5 i.e., 3^2

It seems that the sum of n odd numbers is n-square. We can proof it by mathematical induction.

Given a positive number n, following procedure is outlined:

(*a*) f(1) is true, since $1 = 1^2$

(*b*) If all of f(1), f(2), …, f(n) are true, then in particular f(n) is true, so the series holds, addition 2n+1 to both sides we obtained:

$$1 + 3 + \ldots + (2n - 1) + (2n + 1) = n^2 + (2n + 1)$$
$$= (n + 1)^2$$

which proves that f(n+1) is also true. That means sum of (n+1) odd integers is square (n+1).

The method of proof is called a proof by mathematical induction.

The time complexity of this algorithm is $O(n^2)$.

Example 2. The sum of a geometric progression series. Assume that $x <> 1$, $n \ge 0$.

$$a + ax + ax^2 + \ldots + ax^n = \Sigma ax^i \quad (1)$$
$$0 \le i \le n$$
$$= a + \Sigma ax^i \quad (2)$$
$$1 \le i \le n$$
$$= a + x\,\Sigma ax^{i-1} \quad (3)$$
$$1 \le i \le n$$
$$= a + x\,\Sigma ax^i \quad (4)$$
$$0 \le i \le n - 1$$

$$= a + x \sum_{0 \le i \le n} ax^i - ax^{n+1} \quad (5)$$

Comparing the first relation with the fifth, we have

$$\sum_{0 \le i \le n} ax^i = a + x \sum_{0 \le i \le n} ax^i - ax^{n+1}$$

or

$$\sum_{0 \le i \le n} ax^i - x \sum_{0 \le i \le n} ax^i = a - ax^{n+1}$$

$$= (1 - x) \sum_{0 \le i \le n} ax^i = a - ax^{n+1} \quad (6)$$

and thus we obtain the basic formula

$$= \sum_{0 \le i \le n} ax^i = a(1 - x^{n+1})/(1 - x) \text{ for } x < 1 \quad (7)$$

or

$$= \sum_{0 \le i \le n} ax^i = a(x^{n+1} - 1)/(x - 1) \text{ for } x > 1 \quad (8)$$

The time complexity of any algorithm whose frequency count is sum of geometric progression is order of $O(x^n)$.

The example of geometric progression series is

$$2^1 + 2^2 + 2^3 + \ldots 2^7 = 2(2^7 - 1)/(2 - 1) = 2*127 = 254$$

Example 3. The sum of an arithmetic progression series. Assume that $n \ge 0$.

$a + (a + b) + (a + 2b) + \ldots + (a + nb)$

$$= \sum_{0 \le i \le n} (a + bi) \quad (1)$$

$$= \sum_{0 \le n-i \le n} (a + b(n - i)) \quad (2)$$

$$= \sum_{0 \le i \le n} (a + bn - bi) \quad (3)$$

$$= \sum_{0 \le i \le n} (2a + bn) - n \sum_{0 \le i \le n} (a + bi) \quad (4)$$

$$= (n + 1)(2a + bn) - \sum_{0 \le i \le n} (a + bi) \quad (5)$$

Since the first sum was simply a sum of $(n + 1)$ terms which did not depend on i. Now by equating the first and fifth expressions and dividing by 2, we obtain

$$\sum_{0 \le i \le n} (a + bi) = a(n + 1) + (1/2)bn(n + 1)$$

The order of magnitude of this type of algorithms are $O(n^2)$, by ignoring constants a, and b.

Example 4. If function $f(n) = a_m n^m + \ldots + a_1 n + a_0$ is a polynomial of degree m then $f(n) = O(n^m)$.

Proof: Using the definition of function f(n) and a simple inequality

$$|f(n)| \le |a_m| n^m + \ldots + |a_1| n + |a_0|$$
$$\le (|a_m| + |a_m - 1|/n + \ldots + |a_0|/n^m) n^m$$
$$\le (|a_m| + \ldots + |a_0|) n^m, \; n \ge 1$$

Choosing $c = |a_m| + \ldots + |a_0|$ and $n_0 = 1$, the function immediately follows the order $O(n^m)$.

The above function says that if we can describe the frequency count of any algorithm by a polynomial such that f(n), then the time complexity of that algorithm is $O(n^m)$.

Exponents and Logarithms

Let us consider a is positive real number, if n is an integer then a^n is defined by the familiar rules

$$a^0 = 1, \quad a^n = a.\ a.\ a \ldots a \text{ (n times)}$$
$$a^n = a^{n-1}.a \quad \text{if } n>0,$$
$$a^n = a^{n+1}/a \quad \text{if } n<0$$

It is easy to prove by induction that the laws of exponents are valid:

$$a^{x+y} = a^x.\ a^y$$
$$(a^x)^y = a^{xy}$$

where x and y are integers.

Exponents include all rational numbers by defining, for any rational number m/n

$$a^{m/n} = \sqrt[n]{a^m} = (\sqrt[n]{a})^m$$

For example, $125^{2/3} = (\sqrt[3]{125})^2 = 5^2$
$= 25$

Now consider a positive real number y is given, can we find a real number x such that

$$y = b^x$$

For example, what is the value of x in expression $25 = 5^x$, the value of x is definitely 2. But how? The number x is calculated as logarithm of y to base b, and we write this as $x = \log_b y$.

The other examples are

$$\log_2 8 = 3, \text{ since } 2^3 = 8$$
$$\log_{10} 100 = 2, \text{ since } 10^2 = 100$$
$$\log_2 64 = 6, \text{ since } 2^6 = 64$$

Furthermore for any base b,

$$\log_b 1 = 0, \text{ since } b^0 = 1$$
$$\log_b b = 1, \text{ since } b^1 = b$$

The logarithm of a negative number and the logarithm of 0 are not defined.

We can view exponential and logarithm as inverse of each other.

$$f(x) = b^x \quad \text{and} \quad g(x) = \log_b x$$

The term log x shall mean $\log_2 x$ unless otherwise specified.

From the laws of exponents it follows that:

$$\log_b(xy) = \log_b x + \log_b y \qquad \text{if } x>0, y>0$$
$$\log_b(c^y) = y \log_b c \qquad \text{if } c>0$$
$$\log_c x = \log_b x/\log_b c \qquad \text{if } x>0 \text{ e.g } \log_2 10 = \log_{10}10/\log_{10}2$$
$$\log_b x/y = \log_b x - \log_b y \qquad \text{if } x>0, y>0$$

Product: Product of the positive numbers from 1 to n is denoted by n! (read 'n factorial').

That is n! = 1. 2. 3. … (n–1).n

It is also to define 0! = 0

It can be represented by product of numbers as equivalent to summation.

$$n! = \prod i = 1.2.3\ldots. n$$
$$1 \le i \le n$$

For example 4! = 1.2.3.4 and 5! = 1.2.3.4.5

1.7 ASYMPTOTIC NOTATION (O, θ, Ω)

The notation we use to describe the asymptotic running time of an algorithm is defined in terms of function.

O (big oh)-notation

When we have only an asymptotic upper bound we use O (big oh)-notation, for a given function g(n) we denoted by O(g(n)) the set of functions

$O(g(n)) = \{ f(n)$: there exists positive constant c and n_0 such that

$0 \le f(n) \le c\ g(n)$ for all $n > n_0 \}$

Figure 1.2(a) depicted the asymptotic O-notation. All values n to the right of n_0, the value of the function f(n) is on or below g(n).

We use O-notation to give upper bound on a function to within a constant factor. It is used to bind worse case running time of an algorithm. The time complexity $O(n^2)$ bound worse case for insertion sort algorithm.

Thus, we can write the computing time for the following frequency counts of algorithms as follows:

Frequency counts (g(n))	Time complexity (f(n) = O(g(n)))
$10n^3$	$O(n^3)$
$9n^2 + 4n + 1$	$O(n^2)$ as $9n^2 + 4n + 1 \le 10n^2$ for all $n \ge 3$ so $c = 10$ and $n_0 = 3$
$n(n + 1)/2$	$O(n^2)$ as $n^2/2 + n/2 \le n^2$ for all $n \ge 2$ so $c = 1$ and $n_0 = 2$
$n + \log_2 n$	$O(n)$ as $n + \log_2 n \le 2n$ for all $n \ge 2$ so $c = 2$ and $n_0 = 2$
$4*2^n + n^2$	$O(2^n)$ as $4*2^n + n^2 \le 5*2^n$ for all $n \ge 4$ so $c = 5$ and $n_0 = 4$

Here we have considered the higher order frequency counts, remaining lower-order assuming constant.

Ω (Omega)-notation

Ω-notation provides an asymptotic lower bound. For a given function g(n) we denoted by Ω(g(n)) the set of functions.

$\Omega(g(n)) = \{ f(n)$: iff there exists positive constant c and n0 such that

$0 \le c\ g(n) \le f(n)$ for all $n > n_0 \}$

Figure 1.2 (b) depicted the asymptotic Ω-notation. All values n to the right of n_0, the value of the function f(n) is on or above g(n).

Ω-notation describes a lower bound, when we use it to bound the best case running time of an algorithm. The best case running time of insertion sort is Ω(n) as all n inputs are sorted.

Thus, we can write the computing time for the following function counts of algorithms as follows:

Function counts (g(n))	Upper bound time complexity (f(n) = Ω(g(n)))
$10n^3$	$\Omega(n^3)$
$9n^2 + 4n + 1$	$\Omega(n^2)$ as $9n^2 + 4n + 1 \ge 9n^2$ for all $n \ge 1$ so $c = 9$ and $n_0 = 1$
$n(n + 1)/2$	$\Omega(n^2)$ as $n^2/2 + n/2 \ge n^2/2$ for all $n \ge 1$ so $c = 1$ and $n_0 = 1$
$n + \log_2 n$	$\Omega(n)$ as $n + \log_2 n \ge n$ for all $n \ge 2$ so $c = 1$ and $n_0 = 2$
$4*2^n + n^2$	$\Omega(2^n)$ as $4*2^n + n^2 \ge 4*2^n$ for all $n \ge 1$ so $c = 4$ and $n_0 = 1$

θ (Theta)-notation

θ-notation provides an asymptotic average bound. For a given function g(n) we denoted by θ (g(n)) the set of functions.

$\theta(g(n)) = \{ f(n)$: iff there exists positive constants c1, c2 and n0 such that

$0 \le c1\ g(n) \le f(n) \le c2\ g(n)$ for all $n > n_0 \}$

Figure 1.2(c) depicted the asymptotic θ-notation. All values n to the right of n_0, the value of the function f(n) lie at or above c1 g(n) and at or below c2 g(n). Function g(n) is an asymptotically tightly bound for f(n).

θ-notation can be expressed as between worse case O-notation and best case Ω-notation. For insertion sort the complexity may be between Ω(n) as lower bound to $O(n^2)$ upper bound.

Thus, we can write the computing time for the following function counts of algorithms as follows:

Function counts (g(n))	average bound time complexity (f(n) = θ (g(n)))
$3n + 2$	$\theta(n)$ as $3n+2 \geq 3n$ for all $n \geq 2$ and $3n+2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$ and $n_0 = 2$
$9n^2 + 4n + 1$	$\theta(n^2)$ as $9n^2 + 4n + 1 \geq 9n^2$ for all $n \geq 1$ and $9n^2 + 4n + 1 \leq 10n^2$ for all $n \geq 1$ so $c_1 = 9$, $c_2 = 10$ and $n_0 = 3$
$n + \log_2 n$	$\theta(n)$ as $n + \log_2 n \geq n$ for all $n \geq 2$ and $n + \log_2 n \leq 2n$ for all $n \geq 3$ so $c_1 = 1$, $c_2 = 2$ and $n_0 = 3$
$4*2^n + n^2$	$\theta(2^n)$ as $4*2^n + n^2 \geq 4*2^n$ for all $n \geq 1$ and $4*2^n + n^2 \leq 5*2^n$ for all $n \geq 1$ so $c_1 = 4$, $c_2 = 5$ and $n_0 = 1$

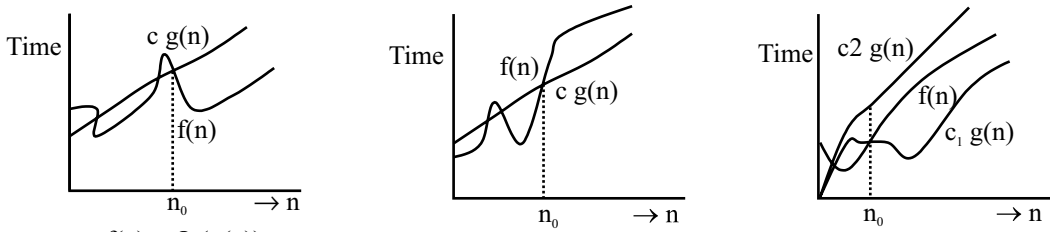


Figure 1.2 *Asymptotic notation*

1.8 PERFORMANCE MEASUREMENT

Performance measurement is concerned with obtaining the memory (or space) and time requirements of a particular algorithm. These quantities depend on the compiler and options used as well as computer on which the algorithm is run.

The run-time complexity of a sequential search, for size of n is O(n) is plotted in Fig. 1.3. So we expect a plot of the times to be straight line.

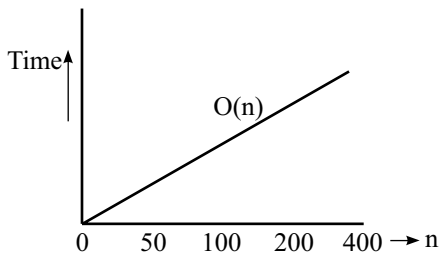


Figure 1.3 *Time complexity graph for sequential search*

Chapter 2

Built-in Data Structure

Here we have discussed some important concept of 'C' language and the use of function. In the course of data structure the array, function, pointer and structure is the basic building block.

2.1 ABSTRACT DATA TYPE

A useful tool for specifying the logical properties of a data type is the abstract data type (ADT). Fundamentally, a data type is a collection of values and set of operations on those values. That collection and those operations form a mathematical construct that may be implemented using a particular hardware or software with data structure.

The term ADT refers to the basic mathematical concept that defines the data types by specifying the mathematical and logical properties of data type or structures. The ADT is a useful guideline to implementers and a useful tool to programmers who wish to use the data type commonly.

An ADT consists of two parts:

- Value definition
- An operator definition

The value definition defines the collection of values for the ADT and consists of two parts: a definition clause and a condition clause.

Immediately following the value definition comes the operator definition. Each operator is defined as an abstract function with three parts: a header, the optional preconditions and the postconditions. The postcondition specifies what the operation does.

2.2 ARRAYS

Arrays are discussed in detail in Chapter 3.

2.3 FUNCTIONS

The 'C' language is accompanied by a number of library functions that carry out various commonly used operations or calculation. However, 'C' also allows programmers to define their own functions for carrying out various individual tasks. The use of programmer/user-defined functions allow a large program to be broken out into smaller ones, each of which has some unique, identifiable fact. The decomposition of a large program into individual program modules (i.e., function) is generally considered to be an important part of good programming practice.

A function is a self-contained program segment that carries out some specific, well-defined task. Or a function is lines of 'C' code grouped into a named unit, written to perform an identifiable activity.

Every 'C' program consists of one or more functions. One of these functions must be main(). Program execution is always beginning by carrying out the statements in main function. If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition cannot be embedded within another. Thus all user-defined function definitions appear before or after the main-function definition.

```
main()
{
statements;
}
```

All C programs must have a main function.

A function will execute its statements whenever the function is "called" from some other portion of the program. The same function can be called from several different places within a program. Once the function has executed its code, the control will be returned to the point from which the function was called. Thus, the use of a function avoids the need for redundant (repeated) programming of the same instructions (i.e., statements).

Another important aspect is that functions are easier to write and understand. Simple functions can be written to do unique specific tasks. Also debugging the program is easier as the program structure is more apparent, due to its modularized form. Each function can individually be tested for all possible inputs. The functions can be put in a library of related functions and used by many programs, thus it can serve as a "building block" to help to build other programs.

Function Prototype and its Declarations

The functions are declared at the beginning of the program through function prototyping. A function prototype is a function declaration that specifies the data types of the arguments. The general form of a function prototype definition is

Data_type function_name(data_type1 parameter_name1, …,data_type n parameter_name n) ;

The data_type specifies the data type of the value which the function will return. The function_name is the name of the function, which has the same naming convention as with identifier names. The parameter names are optional. The parameters are separated by comma, each parameter has its data type specified by data_type. A pair of empty parentheses must follow the function name if the function declaration does not include any parameters or arguments.

When function is declared without any prototype information, the compiler assumes that no information about the parameters is given. Whether the function has no parameters or several parameters does not make a difference. When a function has no parameters, its prototype uses void inside the parentheses.

For example, declares a function as returning no value and has no parameters such as

```
void nothing(void);
```

This indicates that function nothing has no returning value and has no parameters, and any call to that function that has parameters is erroneous.

When a non-prototyped function is called all characters data type identifiers are converted to integers and all floats are converted to doubles. However, if a function is prototyped, the data type mentioned in the prototype are maintained and no type promotions occur.

Function Definition

The general form of function definition is given below:

```
Data_type  function_name(data_type1 parameter_name1, …,data_type n parameter_name n)
{
body of the function
}
```

The first line of a function definition contains the data type specification of the value returned by the function, followed by the function name, and (optionally) a set of arguments, separated by commas and enclosed in parentheses. This line is similar to the function declaration and required parameters name if any parameter exists. These parameters are also known as formal parameters or formal arguments. The formal arguments allow information to be transferred from calling portion of the program to the function but the information actually being transferred by actual parameters or actual arguments at the time of function calling. The identifiers used as formal arguments are "local" in the sense that they are not recognized outside of the function. Hence, the names of the formal arguments may be same as the names of other identifiers that appear outside the function definition.

The body of the function is defined in a compound statement that defines the action to be taken by the function. Like any other compound statement, this statement can contain expression statements, other compound statements, control statements, and so on. As a result, it can also call or access other functions.

The value is returned from the function to the calling portion of the program via the return statement. The return statement also causes control to be returned to the point from which the function was called.

In general terms, the return statement can be used in any of the following ways:

```
return;
return expression;
or
return (expression);
```

The value of the expression is returned to the calling portion of the program. If the expression is omitted, the return statement simply causes control to revert back to the calling portion of the program, without any value transfer. However, a limitation of return is that it can return only one value and its data type must be same as the data type specification of the returned value in the function definition.

A function definition can include multiple return statements, each may be containing a different expression. Functions that include multiple branches often require multiple returns.

Invoking a Function

A function can be invoked or called from any portion of the program except from its definition portion simply by using its name, including the parentheses that follow the name. The parentheses are necessary so that the compiler knows that a function is referred, not a variable and it always ends with a semi-colon.

Example 1. The following program calls a function named output() that writes a message to the screen. The function output() does not return anything and has no parameter.

```
void main()
{
void  output();                 /* function declaration */
output();                       /* function calling */
}
void  output()                  /* function definition */
{
printf(" \n Welcome from C function");
}
```

Example 2. The following program calls a function named numout () that writes a value it receives to the screen.

```
void main()
{
int a = 10;
void numout(int a);                    /* function declaration */
numout( a );                           /* function calling */
}
void numout(int num)                   /* function definition */
{
printf(" \n The value received was %d \n", num);
}
```

In this example, the value contained in the actual variable a is copied into the formal variable num. The function numout() then prints this value.

Example 3. The next program employs the function numout() to print the numbers 1 through 100. The function numout() invoked 100 times.

```
void main()
{
int a;
void numout(int a);                    /* function declaration */
for (a = 1; a< = 100; a++)
numout( a );                           /* function calling */
}
void numout(int num)                   /* function definition */
{
printf(" \n The value received was %d \n ", num);
}
```

Example 4. The following program calls a function named ntoc() that receives any decimal digit and returns its character values. The function returns pointer to characters and invoked ten times.

```
void main()
{
int a;
char  *ntoc(int a);                    /* function declaration */
for (a = 1; a<10; a++)
printf ("\n %d is written in words %s", a , ntoc(a));      /* function calling */
}
char *ntoc(int num)                    /* function definition */
{
        switch (num)
        {
```

```
                case 0: return("zero");
                case 1: return("one");
                case 2: return("two");
                case 3: return("three");
                case 4: return("four");
                case 5: return("five");
                case 6: return("six");
                case 7: return("seven");
                case 8: return("eight");
                case 9: return("nine");
                default: printf(" \n The digit received was invalid");
        }

}
```

Example 5. The following program calls a function named celtofar() converts a temperature from degrees Celsius to Fahrenheit. The function celtofar() takes one integer type parameter and returns a float value.

```
void main()
{
int c;
float f;
float celtofar(int c);                  /* function declaration */
printf("\n Please enter temperature in degree Celsius");
scanf("%d", &c);
f = celtofar(c);                        /* function calling */
printf("\n The temperature in Fahrenheit = %f", f);
}

float celtofar(int c)                   /* function definition */
{

return(9.0/5.0 * c + 32);
}
```

Example 6. The following code illustrated the scope of variables in the function. Variables are basically of three kinds: local, formal and global variables. Local variables are those that are declared inside a function and their scope is limited to within the function. Formal parameters are declared in the definition of function parameters and global variables are declared outside all functions and accessed by all.

```
int ctr;                                        /*global variable */
void main()
{
        void blk1(char ch, int i);              /* function blk1 declaration */
        ctr = 10;
```

```
        blk1('a', 5);                   /* function blk1 called from main function */
        .
        .
}
        void blk1(char ch, int i)          /* function blk1 definition */
{
        void blk2(void);                   /* function blk2 declaration, it is local to blk1*/
        int rtc = 0;                    /* rtc is a local variable */
        if ((ctr > 8) && (i>4))
        {
                rtc++;
                blk2();                 /* function blk2 called  from function blk1*/
        }
        else
                ch = 'f';
}
void blk2(void)                         /* function blk2 definition */
{
        int ctr;                        /* local variable */
        ctr = 0;
}
```

In the above program code, ctr is a global variable and even though it is declared outside main(), and function blk1, it can be referenced to, within them. **Hence, we need to pass the global variable ctr in any function because it is accessed from anywhere in the program**. The variable ctr in function blk2, though, is a local variables and has no connection whatsoever with the global ctr. When the global and local variable have the same name, all references to that name within the block defining the local variable will be associated with the local variable and not the global one.

Storage for global variables is in fixed region of memory set aside for this purpose by the C compiler. Global variables are useful when many functions in a program use the same data.

The above program code executes from main function. The first line set global variable ctr to 10 and second line invoked function blk1 by passing two actual parameters using call by value. The function blk1 definition used two formal parameters to copy these actual parameters, first one is character type ch and second one is integer type i. The function blk1 has defined one local integer variable rtc and checked a test condition upon the global variable and formal parameter i. Here the if block is tested, which returns true and increments the local variable value and invoked a function blk2. Thus function blk2 declared before to invoke in the function blk1 and its scope is also local. This invoking is called nesting of the function, as a function calling another function.

The function blk2 can't invoke from main function, as it is local to blk1 function.

Here the main function invoked function blk1 and blk1 invoked function blk2 and then after executing blk2 the control is transferred to blk1 where it has called blk2. Now function blk1 remaining code has been executed and then control transferred to the main function where it has been called. Now main function executes its remaining code and finishes program execution.

Parameter Passing Methods

1. Passed by Value

In 'C', by default, all function arguments are passed by value. This means that, when arguments are passed to the called function the actual values are copied into the formal parameter memory locations. Any manipulation to be done is done only on these formal parameters. The called function cannot access the actual memory locations of the passed variables and therefore cannot change its value.

Consider the following example:

Example 7.

```
#include <stdio.h>
void main()
{
        int a = 2, b = 4, c = 0;
        int adder(int, int);            /* function declaration */
        printf("\n Before calling adder function a = %d b = %d c = %d", a, b, c);
        c = adder(a , b);
        printf("\n After calling adder function a = %d b = %d c = %d", a, b, c);
}
int adder(int a, int b)         /*function definition, a and b are the formal parameters */
{
        int c;
        c = a + b;
        a *= a;
        b + = 5;
        printf("\n Within adder function a = %d b = %d c = %d", a, b, c);
        return (c);
}
```

The output will be

```
Before calling adder function a = 2 b = 4 c=0
Within adder function a = 4 b = 9 c = 6
After calling adder function a = 2 b = 4 c=6
```

Two integers a and b are passed to the function adder(). It takes them, adds them, squares the first integer, adds 5 to the second integer, prints the resultant and returns the addition of the actual arguments. The variables which are used in the main() and the adder() functions have the same name. But they are stored in different memory locations. The actual variables a and b values are copied to the formal parameters a and b in different memory locations.

The variables a and b in the function adder() are altered from 2 and 4 to 4 and 9, but change does not affect the values of the variables a and b in the main(). This shows that these variables must be stored in different memory locations. The variable c in main() is different from the variable c in adder().

So, arguments are said to be a passed call by value when the value of the variables are passed to the called function and any alterations on this value has no effect on the original value of the passed variable.

2. Passed by Reference

The function is allowed access to the actual memory location of the passed arguments and therefore can change the value of the arguments of the calling routine or function. If we pass the address of a variable to a function, we can declare pointers within the function and assign them to the locations in memory addressed by the parameters. By referencing a variable's address in memory, functions can change the contents as desired.

There may be cases where the values of the arguments of the calling routine have to be changed.

For example, consider the following program that illustrates the parameter passing by reference:

Example 8.

```
#include<stdio.h>
void main()
{
void interchange(float *, float *);                    /* function declaration */
float x, y;
printf("\n Please enter value for x and y :");
scanf("%f%f", &x, &y);
printf("\n Values before function call  x = %f, y = %f", x, y);
interchange(&x, &y);                                   /* function calling by reference */
printf("\n Values after function call  x = %f, y = %f", x, y);
}

void interchange(float *xx, float *yy)                 /* function definition */
{
        float t ;      /* local variable */
        t = *xx ;
        *xx = *yy ;
        *yy = t ;
}
```

The execution of the above program as follows by pressing key Ctrl + F9. The output will be

```
Please enter value for x and y : 4.5  12.3
Values before function call  x = 4.5, y = 12.3
Values after function call  x = 12.3, y = 4.5
```

Two floats a and b addresses are passed to the function interchange(). It takes them, the pointer variable xx will receive the value 1001 (memory location of x), and yy will receive the value 1100 (memory location of y). Since we have declared xx and yy as pointers, we can modify the values contained at the locations referenced by each:

t = *xx;

This assignment places the value reference by the pointer variable xx into a float variable t. Since xx contains the memory location of x, so value of x is copied into variable t and now t has 4.5.

*xx = *yy;

This assignment places the value reference by the pointer variable yy into a memory location referenced by the pointer variable xx. Since yy contains the memory location of y and xx contains the memory location of x, so value of y is copied into the x and now x has 12.3.

```
*yy = t;
```

This assignment places the value of variable t into the memory location referenced by the pointer yy. Since yy contains the memory location of y, so value of t is copied into the y and now y has 4.5.

Some points to remember with reference to function:

- A semicolon is used at the end of the statement when function is called, but not after the function definition.
- Parentheses are compulsory after the function name irrespective of whether the function has arguments or not.
- The returned data type is not compulsory if an int type of value is returned or if no value is returned.
- A function is declared in main(), before it is defined or used, if the function is called before it is defined.
- In 'C', by default, all function arguments are passed by value.
- Variables are basically of three kinds: local, formal and global variables. Local variables are those that are declared inside a function. Formal parameters are declared in the definition of function parameters and global variables are declared outside all functions.
- A function cannot be defined within another function.
- A function prototype is a function declaration that specifies the data types of the returned value and formal arguments. This prototyping must be followed by the function calling and function definition with respect to arguments data types.
- Calling one function from within another is said to be nesting of function calls.
- The main() function returns integer to the calling process.

2.4 STRUCTURES AND UNIONS

A structure is a grouping of variables under one name and is called record. Another user-defined type is the union, which saves the memory.

A structure for a student record is given below:

```
struct  studinfo{
            int rollno;
            char *name;
            float pm;
            }s1, s2;
```

The declaration of structure studinfo consists of student rollno, name, and percentage marks (i.e., pm). The structure variable s1 and s2 can access the member of studinfo by dot (.) reference.

We can read the whole record from the keyboard using scanf statement

```
Scanf("%d%s%f",&s1.rollno,s1.name,&s1.pm);
```

Similarly, printf statement works and other access works.

Array of Structure

An array of structure for a student record is given on the next page.

```
#define MAXSIZE 10
struct  studinfo{
        int rollno;
        char *name;
        float pm;
        }s[MAXSIZE];
```

Here variable s[i] denotes the record of i^{th} students and elements of the record is accessed as s[i].rollno, s[i]. name and s[i].pm

Pointer to Structure

A pointer to structure for a student record is given below:

```
struct  studinfo{
        int rollno;
        char *name;
        float pm;
        }*s;
```

Here s is the pointer to structure and the member of structure is accessed as

s->rollno, s->name, s->pm

We can allocate memory for each record as and when required using malloc statement:

s = (struct studinfo *)malloc(sizeof(struct studinfo));

It is possible to have one structure within another structure. A structure variable either simple, array or pointer can be passed as an argument to another function and also as return type. Array of pointers to structure can also be declared and used when number rows are fixed.

A union is a memory location that is shared by two or more variables, generally of different types, at different times. Defining a union is similar to defining a structure. It is possible to nest unions within unions, unions in structures, and structures in unions.

2.5 POINTERS

What is a Pointer?

A pointer is a variable which contains the address of a memory location of another variable, rather than its stored value. If one variable contains the address of another variable, the first variable is said to **point** to the second. A pointer provides an indirect way of accessing the value of a data item.

Consider two variables v1 and v2 such that v1 has value of 100 and stored in the memory location 1000. If v2 is declared as a pointer to the variable v1, the representation will be as follows:

Memory location	*Variable name*	*Value stored*
1000	v1	100
.	.	.
.	.	.
.	.	.
1200	v2	1000

Variable v2 contains the value 1000 which is nothing but the address of the variable v1.

Pointers can point to variables of other fundamental data types variables like int, char, float, or double or data aggregates like arrays, structures and union,

Why are Pointers Used?

Pointers are used in the situations when passing actual values in function is difficult or not desired. Few of the reasons where pointers can be used are:

- To return more than one values from the function
- To pass arrays and strings more conveniently from one function to another
- To manipulate arrays more easily by moving pointers to them (or to parts of them), instead of moving the arrays themselves
- To allocate memory and access it (dynamic memory allocation)
- To create complex data structures, such as linked list, trees, graphs, where one data structure must contain references to other data structures.

Pointer Variables

If variable is to be used as a pointer, it must be declared as such. The general form for declaring a pointer variable is

```
Data type *name;
```

where data type is any valid data type, and name is the name of the pointer variable. The declaration tells the compiler that name is used to store the address of a value of type data type.

Consider the above example of v1 and v2, since v2 is a pointer which holds the value of an int data type variable v1, as follows:

```
int  v1 = 100;
int  *v2;
```

Now v2 can be used in a program to indirectly access the value of v1 as declared

```
v2 = &v1;
```

places the memory address of v1 into v2, & can be thought of as returning "the address of".

The pointer operator, *, is the complement of &. It is a unary operator and returns the value contained in the memory location pointed to by the pointer variable's value.

Pointers and Arrays

The following declaration creates an array p of integers:

```
int *p;
```

The following declaration and initialization create an array named sname containing 10 names, each of which is of type **char** *.

```
char *sname[10] = {"ramo","hari", "suman", "poonam","sanjay","rajiv","rajesh","rohit", "chetan","rahul"};
```

sname[i] store the i^{th} student name.

Pointers as Function Arguments

Pointers are passed to a function as arguments, to enable data items within the called routine of the program to access variables whose scope does not extend beyond the calling function. When a pointer is passed to a function, the address of a data item is passed to the function making it possible to freely access the contents of the address.

In this way, function arguments permit data items to be altered in the calling routine, enabling a two-way transfer of data between the calling routing and function. When the function arguments are pointers or arrays, a call by reference is made to the function as opposed to a call by value for the variable arguments.

Several declarations involving pointers are shown on the next page.

int f(int *x)**;**	/* f is a function that accepts an argument which is pointer to a integer and returns an integer value. */
int f(char *x[]	/* f is a function that accepts an argument which is array of pointer to an integer and returns an integer value. */
int *f(char x[])	/* f is a function that accepts an argument which is character array and returns a pointer to an integer quantity. */
int *f(char *x[])	/* f is a function that accepts an argument which is array of pointer to an integer and returns a pointer to an integer quantity. */
int * f(struct studinfo *s)	/* f is a function that accepts an argument which is pointer to structure studinfo and returns a pointer to an integer quantity. */
struct studinfo * f(struct studinfo *s[])	/* f is a function that accepts an argument which is array of pointer to structure studinfo and returns a pointer to structure studinfo. */

2.6 MEMORY MANAGEMENT IN 'C'

Consider an example of single dimensional integer array x having 20 elements defined as

```
int x[20];
```

A block of memory enough to store 20 integer values (e.g. 40 bytes) will be reserved in advance. If x is defined as a pointer (i.e., int *x) , sufficient memory can be reserved in advance using malloc function as follows

```
x = malloc(20 * sizeof(int));
```

this will reserve a block of memory for 20 integers. The number 20 is multiplied by sizeof(int) (i.e., 2 bytes) to reserve a block of contiguous memory. The function malloc returns a pointer which is the address location of the starting point of the memory allocated. If enough memory space does not exist, malloc returns a NULL.

```
int *x;
```

we can write a statement for memory allocation for pointer to integer x in the following manner:

```
x = (int *) malloc(20 * sizeof(int));
```

The allocation of memory in this manner, that is, as and when required in a program, is known as Dynamic memory allocation.

Memory allocation for a pointer to structure is given below:

```
struct studinfo{
        int rollno;
        char *name;
        float pm;
        }*s;
```

We can allocate memory for each record as and when required using malloc statement.

```
s = (struct studinfo *)malloc(sizeof(struct studinfo));
```

The function free is used in 'C' to free storage of a dynamically allocated variable. The statement

```
free(s);
```

free the memory occupied by the pointer variable *s, the pointer variable is not deleted. Once the block is freed, an attempt to de-reference a pointer variable yields undefined results. Such a pointer variable is called **dangling pointer**.

Chapter 3

Unordered and Ordered Arrays

An array is a collection of data elements of the same data type arranged contiguously in computer's memory (i.e., main memory). An array is the most commonly used data storage structure. It is built into most of the programming languages.

An array is a linear, homogenous and static data structure. Arrays are basic building block for more complex data structures.

3.1 ONE DIMENSIONAL ARRAYS

Implementing One Dimensional Arrays

Each array element is referred by one or more subscripts (indexes), with each subscript enclosed in square brackets []. Each subscript must be expressed as a nonnegative integer. The number of subscripts determines the dimensionality of the array. For example, x[i] refers to an element in the one-dimensional array x. Similarly, y[i][j] refers to an element in the two-dimensional array y. Higher dimensional arrays can also be formed, by adding additional subscripts in the same manner (e.g. z[i][j][k]– three dimensional array).

The simplest form of an array is one-dimensional unordered array, it is also called vector. The subscript of an array element designates its position in array's ordering. A one-dimensional array definition may be expressed as

Storage class array_type array_name[expression];

The storage class is optional, and the default value for storage class is automatic for arrays that are defined within a function or a block and external for arrays defined outside a function.

The array_type defined the type of values (e.g. integer, float, character, double) of elements of the array and tells the computer how much space is required to hold each value.

The array_name indicates the memory location of the first element of the array.

The expression indicates the size of an array i.e., the number of elements of the array. The expression is usually written as a positive integer constant and determines how many memory locations are allocated in terms of contiguous block of memory.

Consider a one-dimensional array x declared as

```
int x[n];
```

Thus, the array x has storage class automatic, data elements are integer and the array consisting of n-element (i.e., size of the array). In 'C', arrays subscripts (indexes) always begin with 0. Therefore, the first element of the array x is x[0], second element of the array x is x[1], and third element of the array is x[2] and

the array x last element (i.e., n^{th} element) is x[n–1]. The limits of the allowed subscript values are called bounds of the array index. The lowest valid subscript is called the lower bound and highest valid subscript of the array is called the upper bound. The lower bound of the subscript is always 0 and upper bound of the subscript is equal to size of the array–1.

The above declaration of the array x tells computer to allocate a memory block of 2n bytes to store n integers, as in 'C' size of an integer is 2-byte. Here array x required n memory locations and each integer value required 2-byte memory space. To store 10 integer values, integer array required 20 bytes contiguous memory. The memory mapping for one-dimensional array is illustrated in Fig. 3.1. Generally memory addresses are represented in hexadecimal number, but to maintain simplicity we have written in decimal number.

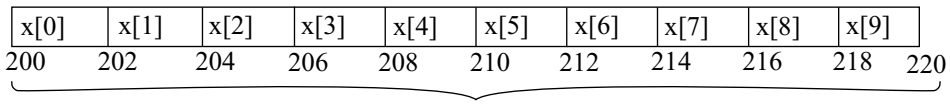


Figure 3.1 *x is 10 elements, one-dimensional array*

The array x has 10 integer elements. Each element size is 2–byte. The array first element x[0] is stored at starting address of the memory space allocated to the array and it is known as base address of the array. The individual array element address can be representing as normal variable address notation rules in 'C'. The following information gives the individual array element address and element number:

Element 1 is x[0] store at memory location 200
Element 2 is x[1] store at memory location 202
Element 3 is x[2] store at memory location 204
Element 4 is x[3] store at memory location 206
Element 5 is x[4] store at memory location 208
Element 6 is x[5] store at memory location 210
Element 7 is x[6] store at memory location 212
Element 8 is x[7] store at memory location 214
Element 9 is x[8] store at memory location 216
Element 10 is x[9] store at memory location 218

Be careful that arrays subscripts begin with 0. The lower bound of array x is 0 and upper bound is 9 i.e., the last element index. **The size of an array** can be calculated as:

Size of an array = upper bound – lower bound + 1

So the size of the x array = 9 – 0 + 1 = 10

To calculate the individual array element address, consider that **the base address of an array** is B and size of each element is S, then the location of I^{th} element would be

= B + (I – 1)*S

To check this formula, apply on the array x and consider the following examples

The address of 1^{st} element of the array x = 200 + (1–1) *2 = 200

The address of 5^{th} element of the array x = 200 + (5–1) *2 = 208

The address of 9^{th} element of the array x = 200 + (9–1) *2 = 216

In the similar manner we can calculate address of any element of one-dimensional array.

Operations on Arrays

The array is a homogenous structure i.e., the elements of an array are of the same type. Following set of operations are defined for this structure:

- Creating an array
- Initializing an array
- Storing an array
- Retrieving an element
- Inserting an element
- Deleting an element
- Searching for an element
- Sorting elements
- Merging
- Printing an array

These operations apply to array of any dimension. Some of the operations are discussed in detail with one-dimensional array and remaining are discussed in the relevant topic.

Creating an Array

Arrays are created in much the same manner as ordinary variables, except that each array name must be accompanied by a size specification (i.e., the number of elements). As we have already defined one-dimensional array, consider some more examples.

int marks[20]; /* marks is a 20-element integer array and this declaration tells computer to allocate a 40 bytes (i.e., 20 *2) contiguous memory block */

char text[80]; /* text is a 80-element character array and this declaration tells computer to allocate a 80 bytes (i.e., 80 *1) contiguous memory block */

static float permarks[20]; /* permarks is defined as a static 20-element floating-point array and this declaration tells computer to allocate an 80 bytes (i.e., 20 *4) contiguous memory block */

It is sometimes convenient to define an array size in terms of a symbolic constant rather than a fixed integer quantity. This makes it easier to modify a program that utilizes an array, since all references to the maximum array size can be altered simply by changing the value of the symbolic constant.

Initializing an Array

Till the automatic array elements are not given any specific values, they are supposed to contain garbage values. If the storage class is declared to be static then all the array elements would have a default initial value of zero. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed in braces and separated by commas.

We can initialize arrays while declaring or defining them. Let us consider the following general form:

storage class array_type array_name[expression] = { value1, value 2, … ,value n};

where value 1 refers to the value of the first array element, value 2 refers to the value of the second element, and so on. The expression, which indicates the number of array elements, is optional when initial values are present.

The following are some array declarations that include the assignment of initial values:

int dice[6] = {1, 2, 3, 4 ,5 ,6 }; /* dice is a 6-element integer array */

float pm[6] ={ 55.5, 60.3, 45.3, 65.3, 54.2, 34.5};/* pm is a 6-element floating-point array */

char vowel[5] = {‘a’, ‘e’, ‘i’, ‘o’ ,‘u’}; /* vowel is a 5-element character array */

The results of these initial assignments, in terms of the individual array elements, are as follows:

dice[0] = 1	pm[0] = 55.5	vowel[0] = ‘a’
dice[1] = 2	pm[1] = 60.3	vowel[1] = ‘e’
dice[2] = 3	pm[2] = 45.3	vowel[2] = ‘i’

dice[3] = 4	pm[3] = 65.3	vowel[3] = 'o'
dice[4] = 5	pm[4] = 54.2	vowel[4] = 'u'
dice[5] = 6	pm[5] = 34.5	

Note that the subscripts (indexes) in an n-element array range from 0 to n –1.

All individual array elements that are not assigned explicit initial values will automatically be set to zero.

The array size need not be specified explicitly when initial values are included as a part of an array definition. With a numerical array, the array size will automatically be set equal to the number of initial values included within the definition. Consider the following array declaration with initialization:

```
int  octel[ ] ={ 0, 1, 2 ,3 ,4 ,5 ,6 ,7}; /* octel is an integer array, size is unspecified */
char color[ ]= "BLUE";/* color is a character array, size is unspecified */
```

Thus, octel will be an 8-element integer array, and color will be a 5-element character array. In case of string, a null character '\0' is automatically added at the end of the string. The individual array elements are

octel[0] = 0	color[0] = 'B'
octel[1] = 1	color[1] = 'L'
octel[2] = 2	color[2] = 'U'
octel[3] = 3	color[3] = 'E'
octel[4] = 4	color[4] = '\0'
octel[5] = 5	
octel[6] = 6	
octel[7] = 7	

The memory map for the array octel is shown below assuming base address is 4012 (in decimal).

octel[0]	octel[1]	octel[2]	octel[3]	octel[4]	octel[5]	octel[6]	octel[7]
0	1	2	3	4	5	6	7
4012	4014	4016	4018	4020	4022	4024	4026

Figure 3.2

Storing an Array

The 'C' programming language stores array element values similar to ordinary variables. We can assign values within the program or values can be input from user through keyboard. Here the section of program code that places values into the array marks elements through user input.

```
for (i =0; i<10; i++)
{
    printf("\n Please enter  marks %d: ", i+1);
    scanf("%d",&marks[i]);      /* The marks is 10–element integer array */
}
```

Analysis

When the array is declared, it allocated 20 bytes contiguous memory block because each of 10 elements would be 2-byte long. And since the array is not being initialized, all ten values present in it would be garbage values.

In the above code, the first time through the for loop, i, has an index 0, so the value read by scanf statement to be stored in the array element marks[0] that is the first element of the array. The loop repeats until the i becomes 9, and the value read by scanf statement to be stored in the array element marks[9] that is the last element of the array.

Remember that in n-element array indexes start from 0 to n–1 that means no array element like marks[10].

The execution of the above section of program code would be

```
Please enter marks 1: 20
Please enter marks 2: 90
Please enter marks 3: 56
Please enter marks 4: 89
Please enter marks 5: 45
Please enter marks 6: 87
Please enter marks 7: 76
Please enter marks 8: 67
Please enter marks 9: 60
Please enter marks 10: 71
```

We can input any integer values for the array marks. Note that to get the output of the above section code we must write a 'C' program including this code.

Traversing One-Dimensional Arrays

Once the element of an array is assigned values, we can retrieve individual array element using subscript, the number in the bracket following the array name. This number specifies the array element position. Array subscripts are numbered starting with 0. For example marks[2] is the third element of the array.

The following section of program code reads out the individual array element and displays its memory location and element numbering.

```
for (i =0; i<10; i++)
printf("\n Element %d at position %d in the array, memory location : %d and value is: %d", i+1, i, &marks[i], marks[i]);
```

The execution of this section of program code would be

```
Element 1 at position 0 in the array, memory location 2212 and value is: 20
Element 2 at position 1 in the array, memory location 2214 and value is: 90
Element 3 at position 2 in the array, memory location 2216 and value is: 56
Element 4 at position 3 in the array, memory location 2218 and value is: 89
Element 5 at position 4 in the array, memory location 2220 and value is: 45
Element 6 at position 5 in the array, memory location 2222 and value is: 87
Element 7 at position 6 in the array, memory location 2224 and value is: 76
Element 8 at position 7 in the array, memory location 2226 and value is: 67
Element 9 at position 8 in the array, memory location 2228 and value is: 60
Element 10 at position 9 in the array, memory location 2230 and value is: 71
```

Note that to get the output of the above section code we must write a 'C' program including the code.

Insertion in One-Dimensional Arrays

Inserting an element

The inserting of an element into the array is easy. We use the normal array syntax

```
marks[0] = 77;
```

If there is no element at position 0 in the array then this assignment statement inserts a new element at that position otherwise we have to shift all the elements one position down. The following criteria is followed to insert an element in an array:

1. The new element must be of same storage class and same data type as the array is declared.
2. The new element can be inserted if the array has free memory locations i.e., the one or more positions of the array are vacant (no element is stored on this position).
3. An element can be inserted as a first, in between, and as last element of an array.

Insert an Element as a Last Element of an Array

Let us consider a character array course and it is declared and initialized as below:

```
char  course[20] ="DATA STRUCTURE"; /* course is a 20element character array */
```

As we know that the array size is fixed and occupies a contiguous memory location when it is declared. The course array size is 15 and it stored a string. The individual elements are initialized as below:

```
course[0]  = 'D'        course[8] = 'U'
course[1]  = 'A'        course[9] = 'C'
course[2]  = 'T'        course[10] = 'T'
course[3]  = 'A'        course[11] = 'U'
course[4]  = ' '        course[12] = 'R'
course[5]  = 'S'        course[13] = 'E'
course[6]  = 'T'        course[14] = '\0'
course[7]  = 'R'
```

Note that string is always terminated by a null character. In the course array 15 elements are stored and there is 5 free-memory locations to insert new elements. To insert an element as a last element of the array then it is just a simple assignment statement. For example to insert character 'S' as a last element (i.e. append an element) in the array, course is as follows:

```
course[15] = 'S' ;
```

This assignment statement appends the character 'S' after the string terminating character '\0'. Thus if we write a printf statement to display the string in the following manner

```
printf("%s",course) ;
```

would display DATA STRUCTURE not DATA STRUCTURES because the string is terminated before the last character 'S'.

If we write a printf statement to display the array course all elements characterwise then the output would be

```
course[0]  = 'D'        course[8]  = 'U'
course[1]  = 'A'        course[9]  = 'C'
course[2]  = 'T'        course[10]  = 'T'
course[3]  = 'A'        course[11]  = 'U'
course[4]  = ' '        course[12]  = 'R'
course[5]  = 'S'        course[13]  = 'E'
course[6]  = 'T'        course[14]  = '\0'
course[7]  = 'R'        course[15]  = 'S'
```

We can append an element in the array until all the positions of the array is full.

Inserting an Element at Start or in Between Position of the Array

To insert an element at start position of the array course, we need to shift all the stored elements one position down.

Consider an example to insert a character 'A' at start position of the array course. We know that the array course position starts from 0 to 19 because course is 20-element array. At the time of initialization 15 elements are stored from position 0 to position 14 of the array course. Now to insert an element at position 0 will require shifting all elements from the position 0 to shift one position down in the array course. Here we shift element at position 14 into position 15, and shift element at position 13 into position 14 and so on. Then in the element course[0], we assigned new element 'A'. The following section of code is written to shift the elements one position down in the array:

```
for(i=15; i>0; i– –)  /* this for loop executes 15 times and shifts the value at position 14 into in
position 15 and so on until i reaches to 0*/
course[i] = course[i–1];
course[0] = 'A' ;
```

The execution of this code is as follows:

course[15] = course[14] = '\0'	course[7] = course[6] = 'T'
course[14] = course[13] = 'E'	course[6] = course[5] = 'S'
course[13] = course[12] = 'R'	course[5] = course[4] = ' '
course[12] = course[11] = 'U'	course[4] = course[3] = 'A'
course[11] = course[10] = 'T'	course[3] = course[2] = 'T'
course[10] = course[9] = 'C'	course[2] = course[1] = 'A'
course[9] = course[8] = 'U'	course[1] = course[0] = 'D'
course[8] = course[7] = 'R'	course[0] = 'A'

Thus, if we write a printf statement to display the string in the following manner

```
printf("%s",course)  ;  /*would display the string ADATA STRUCTURE */
```

Inserting an Element in a Specified Position or as a Specified Element Number

To insert an element at any position in an array is similar to insert an element at start position of an array. Let us consider an example to insert a new element at position 5 or as a sixth element of the array course. Then we need to shift all elements one position down from position 5. Consider the following code

```
for(i=15; i>5; i– –)          /* this for loop executes 10 times and shifts the value at position
    14 into position 15 and so on until i reaches to 5*/
course[i] = course[i–1];
course[5] = 'P' ;
```

The execution of this code is as follows

course[15] = course[14] = '\0'	course[9] = course[8] = 'U'
course[14] = course[13] = 'E'	course[8] = course[7] = 'R'
course[13] = course[12] = 'R'	course[7] = course[6] = 'T'
course[12] = course[11] = 'U'	course[6] = course[5] = 'S'
course[11] = course[10] = 'T'	course[5] = 'P'
course[10] = course[9] = 'C'	

Thus, if we write a printf statement to display the string in the following manner

```
printf("%s",course)  ;  /*would display the string DATA PSTRUCTURE */
```

Deletion from One-Dimensional Arrays

Deleting an element is reverse of inserting an element. To delete the last element of an array is very simple. But deleting the first element or any other elements of an array requires moving all the elements upward to fill up the gap into the array up to the deleting element.

We can delete an element by element number i.e., one greater than its position in the array or by element value.

To delete an element by its value requires searching. The element can only be deleted if it is found in the array. We can delete one or more elements of an array.

As we know an array reserved fixed number of contiguous memory location when it is declared. Deleting an element reduces size of the array, not the memory reserved by the array. The size of an array is calculated as

Size of an array = upper bound – lower bound + 1

In 'C' index start from 0 i.e., lower bound is always 0. Then the size of an array (i.e., number of elements) is upper bound + 1 and each deleting reduces the array size by 1.

Let us consider a simple array declaration and initialization.

```
int  dice[6] = {1, 2, 3, 4, 5, 6}; /* the dice is 6-element integer array */
int nElem = 6; /* nElem denotes the number of elements in the array dice */
```

The individual element of the array dice is as shown below before deleting

```
dice[0] = 1        dice[3] = 4
dice[1] = 2        dice[4] = 5
dice[2] = 3        dice[5] = 6
```

Delete the Last Element

Just reduce the number of elements by 1. Now the number of elements in the array dice reduced to 5 i.e., nElem=5 and stored from position 0 to position 4.

Delete the First Element or Delete from Middle of an Array

To delete the first element of the array dice reduce the number of elements by 1 and requires moving all the elements upward to fill up the gap into the array up to the deleting element. The following section of code is required to delete the first element.

```
for(i = 0; i<5 ;i++) /* this for loop executes 5 times and shifts all the elements of the array dice
   upwards to fill up the gap created by element at position 0 (start element) */
dice[i] = dice[i+1];
```

The execution of the above for loop which deletes the first element would be

```
dice[0]  =  dice[1] = 2
dice[1]  =  dice[2] = 3
dice[2]  =  dice[3] = 4
dice[3]  =  dice[4] = 5
dice[4]  =  dice[5] = 6
```

Now the number of elements in the array dice reduced to 5 i.e., nElem=5 and stored from position 0 to position 4.

In a similar manner, we can delete an element from middle of the array dice. Let us consider to delete element number 4. The element number 4 would be dice[3]. Here the deleting element is in position 3 of the array so the for loop would be as follows:

for(i = 3; i<5 ;i++) /* this for loop executes 2 times and shifts all the elements of the array dice upwards to fill up the gap created by element at position 3 (fourth element) */

dice[i] = dice[i+1];

The execution of the above for loop which deletes the fourth element would be

dice[3] = dice[4] = 5

dice[4] = dice[5] = 6

and remaining elements dice[0], dice[1] and dice[2] have no shifting. Now the number of elements in the array dice reduced to 5 and stored from position 0 to position 4.

To Delete an Element by its Value

The element can only be deleted if it is found in an array. The element to be deleted is searched in an array by any searching technique e.g. sequential search, binary search. Here we are illustrating a sequential search in which search element is compared to each and every element of an array. The following section of code is used to search a deleted element in the array dice:

```
for(i=0;i<6;i++)  /* this for loop is used to traverse the elements of array dice*/
if (dice[i] == sElem)  /* to check the search element i.e., compare sElem with each element of the
dice */
return (i);    /* return the position of the deleted element in the array */
return  (–1)  ;  /* return (–1) to indicate that the element is not in the array */
```

Once the position of deleted element is found in the array then we can delete the element in a similar way as we deleted the element by element number. We can get the element number by adding one in the position of the element in the array.

A complete 'C' program delarray.cpp deletes the element by element number or by value in unordered array.


```
#include <stdio.h>
#define MAXSIZE 100
void main()
{
    int dice[MAXSIZE];
    int sElem, pofElem, nElem;
    int i;
    int posDel(int [], int);
    int delElem(int[], int,int);
    void traverse(int[], int);
    printf("\n How many number of elements do you want to insert [1–%d]: ",MAXSIZE);
    scanf("%d", &nElem);
    for(i = 0;i<nElem;i++)
            {
            printf("\n Pl. input the value of element %d:",i+1);
            scanf("%d",&dice[i]);
            fflush(stdin);
            }
    pofElem = delPos(dice,nElem);
    if (pofElem == –1)
            printf("\nElement is not in the array dice, so can't delete");
    else
```

```
    nElem = delElem(dice,nElem,pofElem);
    traverse(dice,nElem);
    getch();
} /* End of main function */
int delPos(int dice[MAXSIZE],int nElem)
{
    int ch,i,n,sElem,pElem;
    printf("\n Enter your choice");
    printf("\n 1. Delete an element by element number:");
    printf("\n 2. Delete an element by element value:\n");
    scanf("%d", &ch);
    if (ch == 1)
    {
            printf("\n Please Enter the element number to be delete [1–%d]:",nElem);
            scanf("%d", &n);
            if ((n>0) &&(n< = nElem))
                    return(n–1);
    }
    else
    if(ch == 2)
    {
            printf("\n Please Enter the element value to be delete:");
            scanf("%d", &sElem);
    for(i = 0;i<nElem;i++)  /* this for loop is used to traverse the elements of array
                                    dice*/
            if (dice[i] == sElem)  /* to check the search element i.e., sElem with the
                                    dice array elements */
                    return (i); /* return the position of the deleted element in the array */
    }
    return (–1);
}/* end of delPos function */
int delElem(int dice[MAXSIZE],int nElem, int pofElem)
{
    int i,n;
    nElem– –;
    for(i = pofElem; i<nElem ;i++) /* this for loop shifts all the elements of the array dice
    upwards to fill up the gap created by element at position pofElem */
    dice[i] = dice[i+1];
    return nElem;
}
void traverse(int dice[MAXSIZE], int nElem)
{
    int i;
    for(i = 0;i<nElem;i++)
    printf("\nElement %d: %d", i+1,dice[i]);
}
```

Analysis: Program uses two functions, delPos and delElem. The function delPos returns the position of the element to be deleted. The function delElem deletes the element by shifting of the elements not physically. The traverse function displays the element of the array from lower bound index to upper bound index.

If in the array same element is stored more than one locations then the repeated element is called duplicates. The deletion of such duplicates is very necessary in some application.

For an array of n elements, a deletion (assuming no duplicates are allowed) requires average of n/2 comparison to find the element to be deleted. Then shifting the remaining elements upward towards lower bound to fill up that memory location requires n/2 moves. The overall frequency count is n to delete an element. Thus deletion into unordered array required O(n) time.

Searching

Searching refers to the operation of finding the location of search element in an array. To search for an element in the array requires comparison. If any element matches with the search element then return the location of that element in the array. Otherwise appropriate messages are displayed. The search is said to be successful if the search element is found somewhere in the array otherwise it is called unsuccessful.

The most intuitive way to search for a given element e.g., sElem in the array aElem where array aElem contains n elements is to compare sElem with each element of aElem array one by one. That is, first we test whether aElem[0]= sElem, and then we test whether aElem[1]= sElem, and so on.

This type of search is called linear or sequential search. On an average this search requires n/2 comparison. Thus, the time complexity is order of n i.e., O(n).

Other search algorithm is binary search and the time complexity is $O(\log_2 n)$ for an array containing n elements.

Linear Search

This is the simplest technique to find out an element in an unordered array. Suppose aElem is a linear array with n elements and sElem is the element to be searched in the array. In linear search, the search element is compared with the first element of the array, if match is found then search is successful and terminated. Otherwise next element of the array is compared with the search element and this process is continued till the search element found or array is completely exhausted.

A linear search algorithm for unordered array is shown below:

Algorithm LSEARCH(aElem, nElem, sElem)

Here aElem is a linear array with n elements i.e., nElem, sElem is the element to be searched and pofElem is return the position of the search element in the array, if pofElem is –1 then search is unsuccessful.

```
Step 1 :  pofElem = –1
Step 2 :  for i = 0 to nElem–1 do
          {
Step 3 :  if (sElem =aElem[i]) then
          pofElem = i;
          i = nElem;
          }
Step 4 :  return pofElem;
Step 5 :  end LSEARCH
```

The Duplicates Problem

When we design a data structure, we must decide whether elements with duplicate values will be allowed. If a data structure is used to process the employee personnel information system, then the employee number

field can't be duplicates. Because the employee number field is unique i.e., we can't assign the same employee number to two employees. On the other hand, the employee last name field value may be same, because there is possibilities that several employees will have the same last name, so duplicates should be allowed.

Searching with Duplicates

Allowing duplicates element in the array creates the complication in the search algorithm. Even if it finds a match, it must continue looking for possible additional matches until the last element of the array is processed.

We can search up to the first match, 2nd match and so on, depending upon the requirement. These search algorithms always require n comparison to go all the way up to the last element.

Insertion with Duplicates

If duplications are not allowed and there is a possibility the user will attempt to insert the same element twice. So we might need to check every existing element before an insertion.

We have inserted an element without checking of duplicity. Let us consider an application of the employee personnel information system. We can't insert an employee number which is already existing in the array (or list or file). To prevent duplicate insertion requires a complete search for the element to be inserted. If it finds a match then the element can't be inserted, otherwise insert in similar manner as we have already done insertion.

Deletion with Duplicates

Deletion might be more complicated when duplicates are allowed, depending on exactly how 'deletion' is defined. If it means to delete the first match element, this is the same as when no duplicates are allowed. However, if deletion means to delete every match element then the same operation might require multiple deletions. This will require checking all n elements and moving more than n/2 elements. The average number of count depends on how the duplicates are distributed throughout the array.

3.2 TWO-DIMENSIONAL ARRAYS

Implementing Two-Dimensional Arrays

The simplest and most commonly used form of multidimensional arrays is the two-dimensional array. A two-dimensional array can be thought of as an array of two single-dimensional arrays. A typical two-dimensional array is the matrix where exp1 represents row of the matrix and exp2 represents column of the matrix. To locate a particular matrix element the required row and column is determined, and element value is read from the location at which they (the row and column) meet.

A two-dimensional array with m rows and n columns will contain m*n elements and can be defined in the matrix form as logical representation.

	column 1	column 2	… column n
row 1	x[0][0]	x[0][1]	… x[0][n–1]
row 2	x[1][0]	x[1][1]	… x[1][n–1]
.	.	.	.
row m	x[m–1][0]	x[m–1][1]	… x[m–1][n–1]

Now consider the memory mapping for two-dimensional arrays. In two-dimensional array we think of data being arranged in rows and columns. However, computer's memory is arranged as a row of cells (i.e., locations). Thus the rectangular structure of two-dimensional array must be simulated. We first calculate the amount of storage area needed and allocate a block of contiguous memory locations of that size.

There are two ways, Row Major Order and Column Major Order, to simulate two-dimensional array into the computer memory.

Consider a two-dimensional array x declared as

int x[3][3] ; /* x is a 9-element (i.e., 3*3) two-dimensional integer array */

The x indicates the address of first element of the array i.e., x[0][0].

This declaration required 9 contiguous memory locations. The size of each element is two bytes. Thus total memory reserved by the compiler is 18 bytes for the array x.

The logical representation of the array x with 3 rows and 3 columns is as given below:

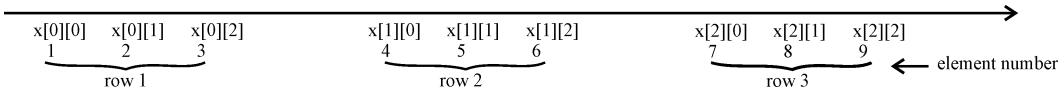
	column 1	column 2	column 3
row 1	x[0][0]	x[0][1]	x[0][2]
row 2	x[1][0]	x[1][1]	x[1][2]
row 3	x[2][0]	x[2][1]	x[2][2]

1. Row Major Order

It stores data in the memory locations row by row. That is, we store first the first row of the array, then the second row of the array and then next and so on.

We know that in computer, memory locations are in sequence, then the following memory representation stores data row by row of the array x declared above.

Contiguous memory locations

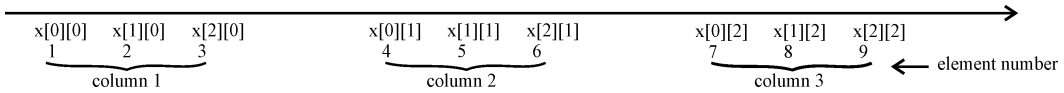


2. Column Major Order

It stores data in the memory locations column by column. That is, we store first the first column of the array, then the second column of the array and then next and so on.

We know that computer memory locations are in sequence, then the following memory representation stores data column by column of the array x declared above.

Contiguous memory locations



It is very important that how the array is stored in memory. In both order the first element of the array is x[0][0], fifth element is x[1][1] and last element of the array is x[2][2]. Therefore in two-dimensional array all the diagonal elements would be in the same locations in row major order and column major order.

Consider some examples to clarify the Row Major Order and Column Major Order.

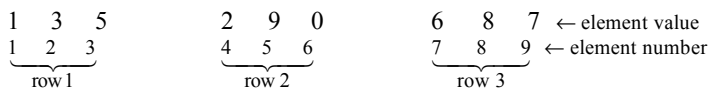
Example 1. Show how the matrix $\begin{Bmatrix} 1 & 3 & 5 \\ 2 & 9 & 0 \\ 6 & 8 & 7 \end{Bmatrix}$ would appear in memory when stored in

(*i*) Row Major Order

(*ii*) Column Major Order.

The above matrix can be represented by two-dimensional array with 3 rows and 3 columns.

(*i*) The memory stored data row by row of the above matrix as



(*ii*) The memory stored data column by column of the above matrix as

$$\underbrace{1\ \ 2\ \ 6}_{\text{column 1}}\quad\underbrace{3\ \ 9\ \ 8}_{\text{column 2}}\quad\underbrace{5\ \ 0\ \ 7}_{\text{column 3}}$$

1 2 3 4 5 6 7 8 9 ← element number

We can ask this problem in a different way.

Example 2. A two-dimensional array with 2 rows and 3 columns stored data in memory as

9 3 6 7 8 2

Show the matrix form of two-dimensional array.

Here the data may be stored in row by row or column by column. If the data is stored in row major order then the memory mapping and the matrix form would be as

$$\underbrace{9\ \ 3\ \ 6}_{\text{row 1}}\quad\underbrace{7\ \ 8\ \ 2}_{\text{row 2}}$$

$$\begin{matrix}\text{row 1} \rightarrow \\ \text{row 2} \rightarrow\end{matrix}\begin{Bmatrix}9 & 3 & 6\\ 7 & 8 & 2\end{Bmatrix}$$

If the data stored in column major order then the memory mapping and the matrix form would be as

$$\underbrace{9\ \ 3}_{\text{column 1}}\quad\underbrace{6\ \ 7}_{\text{column 2}}\quad\underbrace{8\ \ 2}_{\text{column 3}}$$

$$\downarrow\qquad\downarrow\qquad\downarrow$$

$$\begin{Bmatrix}9 & 6 & 8\\ 3 & 7 & 2\end{Bmatrix}$$

In all implementation of pascal and 'C' the storage allocation scheme used is the Row Major Order.

Memory Location of an Element in Two-Dimensional Array

In 'C' the lower bound of each subscript is always 0 and upper bound of each subscript is less than one to its value declared at the time of array declaration. Consider a two-dimensional array x with m rows and n columns i.e., x[m][n] and each element of S size. The base address of the array x is B i.e., the address of first element (x[0][0]).

Memory Location of an Element in Row Major Order

The memory location of an element in memory depends upon the order in which the array elements are stored. The memory location of an element x[i][j] where $0 \le i \le m-1$ and $0 \le j \le n-1$, must skip i rows each having n elements and j elements of $(i+1)^{th}$ row, that is the total number of elements to be skipped is i*n +j.

If the array stores row by row, i.e., Row Major Order in memory locations, then the address of element x[i][j] would be at:

= B + i*n*S + j*S = B + S*(i*n + j)

And thus the element number is = (i*n + j) +1, here 1 is added because element number starts from 1 but the position of an element in the array starts from 0.

For example, a two-dimensional array matA with 4 rows and 6 columns declared as

int matA[4][6] ; /* matA is 4 rows and 6 columns i.e., 4*6 = 24 elements two-dimensional integer array */

Here size of each element is 2 bytes, thus the array required 48 continuous memory locations. Assume that the base address of the array is 2014 (in decimal) byte that means the first element of the array matA is stored at memory location 2014.

Now compute the address or memory location of an element matA[2][3] i.e., the element meets at 3rd row and 4th column in the array matA. Put the values in the above formula to compute the address as

= 2014 + 2*(2*6 + 3)
= 2014 + 2*15
= 2044

and matA[2][3] is the sixteenth element number in the twenty four elements array matA (See below table).

The following table gives the complete memory locations required by the array matA in row major order and element number with index.

	Memory location/address in decimal	*Index*	*Element number*
	2014	matA[0][0]	1
	2016	matA[0][1]	2
R	2018	matA[0][2]	3
o	2020	matA[0][3]	4
w	2022	matA[0][4]	5
1	2024	matA[0][5]	6
	2026	matA[1][0]	7
	2028	matA[1][1]	8
R	2030	matA[1][2]	9
o	2032	matA[1][3]	10
w	2034	matA[1][4]	11
2	2036	matA[1][5]	12
	2038	matA[2][0]	13
	2040	matA[2][1]	14
R	2042	matA[2][2]	15
o	2044	matA[2][3]	16
w	2046	matA[2][4]	17
3	2048	matA[2][5]	18
	2050	matA[3][0]	19
	2052	matA[3][1]	20
R	2054	matA[3][2]	21
o	2056	matA[3][3]	22
w	2058	matA[3][4]	23
4	2060	matA[3][5]	24

Memory Location of an Element in Column Major Order

In column major order data is stored column by column. The memory location of an element x[i][j], where $0 \le i \le m-1$ and $0 \le j \le n-1$, must skip j columns each having m elements and i elements of $(j+1)^{th}$ column, that is the total number of elements to be skipped is j*m + i.

If the array stores in column major order in memory locations then the address of element x[i][j] would be at:

= B + j*m*S + i*S = B + S*(j*m + i)

And thus the element number is = (j*m +i) +1, here 1 is added because element number starts from 1 but the position of an element in the array starts from 0.

Consider the two-dimensional array matA with 4 rows and 6 columns declared as above. Assume that the base address is 2014.

Now compute the address or memory location of an element matA[2][3] i.e., the element meets at 3rd row and 4th column in the array matA. Keep in mind that the elements are stored in memory column by column, thus we must skip 3 columns each having 4 elements and 2 elements of 4th column, then address would be at:

= 2014 + 2*(3*4 + 2)
= 2014 + 2*14
= 2042

and matA[2][3] is the fifteenth element number in the twenty four elements array matA (See below table).

The following table gives the complete memory locations required by the array matA in column major order and element number with index.

	Memory location/address in decimal	*Index*	*Element number*
COL 1	2014	matA[0][0]	1
	2016	matA[1][0]	2
	2018	matA[2][0]	3
	2020	matA[3][0]	4
COL 2	2022	matA[0][1]	5
	2024	matA[1][1]	6
	2026	matA[2][1]	7
	2028	matA[3][1]	8
COL 3	2030	matA[0][2]	9
	2032	matA[1][2]	10
	2034	matA[2][2]	11
	2036	matA[3][2]	12
COL 4	2038	matA[0][3]	13
	2040	matA[1][3]	14
	2042	matA[2][3]	15
	2044	matA[3][3]	16
COL 5	2046	matA[0][4]	17
	2048	matA[1][4]	18
	2050	matA[2][4]	19
	2052	matA[3][4]	20

Contd...

COL 6

2054	matA[0][5]	21
2056	matA[1][5]	22
2058	matA[2][5]	23
2060	matA[3][5]	24

Remembered the order of array in which elements are stored in memory. Here the array element matA[2][3], the element meeting at 3rd row and 4th column is different in row major order from the column order major (See both tables).

Initializing a two-dimensional array is similar to one-dimensional array. Consider the following array definition.

int digit[3][4] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}; /* digit is 3 rows and 4 columns i.e., 12 elements two-dimensional integer array */

In the above initialization of two-dimensional array digit, first assigned the elements of first row then second row and so on i.e., row major order. The number of values cannot exceed the number of elements in the array. If there are too few values within a brace, the remaining elements of the array will be assigned zero.

The individual elements are

digit[0][0] = 0	digit[0][1] = 1	digit[0][2] = 2	digit[0][3] = 3
digit[1][0] = 4	digit[1][1] = 5	digit[1][2] = 6	digit[1][3] = 7
digit[2][0] = 8	digit[2][1] = 9	digit[2][2] = 10	digit[2][3] = 11

Note that the first subscript ranges from 0 to 2 and the second subscript ranges from 0 to 3. A point to remember is that elements will be stored in contiguous locations in memory. The above array digit can be thought of an array of 3 elements, each of which is an array of 4 integers, and will appear as

Row 1				Row 2				Row 3			
0	1	2	3	4	5	6	7	8	9	10	11

The natural order in which the initial values are assigned follows the row major order. This assigned zero to elements of row in cases of few values within a brace. We can alter these assignments by forming groups of initial values enclosed within braces (i.e., {…}). The values within an inner pair of braces will be assigned to the elements of a row, since the second (column) subscript increases most rapidly. If there are too few values within a pair of braces, the remaining elements of that row will be assigned zeros. Now the number of values within each pair of braces cannot exceed the defined row size.

Now consider the following two-dimensional array initialization:

Int rank[3][4] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; /* rank is 3 rows and 4 columns i.e., 12 elements two-dimensional integer array */

This initialization assigns values only to the first three elements in each row. Therefore, the array elements will have the following initial values:

rank[0][0] = 1	rank[0][1] = 2	rank[0][2] = 3	rank[0][3] = 0
rank[1][0] = 4	rank[1][1] = 5	rank[1][2] = 6	rank[1][3] = 0
rank[2][0] = 7	rank[2][1] = 8	rank[2][2] = 9	rank[2][3] = 0

Note that the last element in each row is assigned a value of zero.

Traversing Two-Dimensional Arrays

Traversing of a two-dimensional array can be of two types depending upon the order in which it stored in the memory that is row major order and column major order. Traversing means retrieve or access all elements in the order in which they are stored in memory.

Traversing in row major order

The algorithm to traverse a two-dimensional array in row major order is given below:

```
    Algorithm TROWARR
// Algorithm to traverse a two-dimensional array x with m rows and n columns in row major order
Step 1 :  for i =  0 to m–1 do
            for j = 0 to n–1 do
             write x[i][j]
Step 2 :  end TROWARR
```

Traversing in column major order

The algorithm to traverse a two-dimensional array in column major order is given below:

```
    Algorithm TCOLARR
/* Algorithm to traverse a two-dimensional array x with m rows and n columns in column major order  */
Step 1 :  for j =  0 to n–1 do
            for i = 0 to m–1 do
             write x[i][j]
Step 2 :  end TCOLARR
```

We can also insert an element, delete an element, search an element and sort the two-dimensional arrays similar to one-dimensional arrays.

Matrix Sorting

The matrix or two-dimensional array elements are sorted using bubble sort method. The element at lower bound of the array is smallest element and at the upper bound is largest element. The following function sortmat sorts matrix in ascending order.

```
void sortmat(int row, int col)
{
    int  temp,r = 0, c = 0 ;
    do {
            for(i = 0; i < row; i++)
            {
                    for(j = 0; j< col ; j++)
                    {
                            if(mat[r][c] < mat[i][j])
                            {
                                    temp = mat[r][c];
                                    mat[r][c] = mat[i][j];
                                    mat[i][j] = temp;
                            }
                    }
            }
```

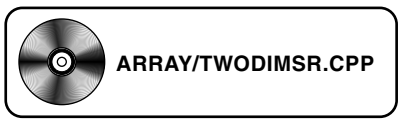
```
        if (c < col–1)
                c++;
        else
        { r++; c = 0;}

        }while (r<row);
    } /* end of sortmat function */
```

The sortmat function has two parameters, row and column of integer type. The function is similar to sorting of single dimension array elements, here two nested loops i and j are corresponding to matrix row and column and one do-while loop nesting these two loops for comparing each element of the matrix.

The do-while loop terminates when number of rows exceeding its limit. When row number changes the column number sets to zero otherwise column number is increment using if-else statement.

A complete 'C' program to sort matrix is given below:

```
/* Sorting a matrix in ascending order i.e., two-dimensional arrays
in row major order. The sort is done with the help of bubble sort method */
 /*  twodimsr.cpp */
  # include<stdio.h>
  #include<stdlib.h>
  #define MAXSIZE 10
int i, j;
int mat[MAXSIZE][MAXSIZE];
void traverse( int, int);
void readmat( int, int);
void sortmat(int, int);
/* traversing function */
void traverse(int row, int col)
{
  for(i = 0; i < row; i++)
  {
      for(j = 0; j < col; j++)
      {
              printf("\t%d", mat[i][j]);
      }
      printf("\n");
  }
}
/* Read matrix function */
void readmat(int row, int col)
{
      for(i = 0 ; i< row; i++)
      {
              for(j = 0 ;  j<col; j++)
```



```
                {
                        printf("\nInput element for :row %d: column %d:", i+1, j+1);
                        scanf("%d", &mat[i][j]);
                }
        }
}
/* Sorting of a matrix */
void sortmat(int row, int col)
{
        int  temp,r = 0, c = 0 ;
        do {
                for(i = 0; i < row; i++)
                {
                        for(j = 0; j< col ; j++)
                        {
                                if(mat[r][c] < mat[i][j])
                                {
                                        temp = mat[r][c];
                                        mat[r][c] = mat[i][j];
                                        mat[i][j] = temp;
                                }
                        }
                }
        if (c < col–1)
                c++;
        else
        { r++; c = 0;}
        }while (r<row);
} /* end of sortmat function */

/* main function */
void main()
{
        int r,c;
        printf("\n Input number of rows:");
        scanf("%d", &r);
        printf("\n Input number of cols:");
        scanf("%d", &c);

        readmat(r, c);
        printf("\n Entered Matrix is as follows:\n");
        traverse(r, c);
        printf("\n Sorted Matrix is as follows:\n");
```

```
    sortmat(r, c);
    traverse(r, c);
    getch();
} /* end of main */
```

3.3 MULTIDIMENSIONAL ARRAYS

Most of the programming languages permit two or more dimensional arrays where an element is referenced by more than one subscripts is called multidimensional. The multidimensional arrays make it easier to represent multidimensional object. For example matrix, rectangle can be represented by two-dimensional array and cube can be represented by three-dimensional array.

Multidimensional arrays are defined much as one-dimensional array, except that a separate pair of square brackets [] is required for each subscript. A two-dimensional array can, thus be represented with two pairs of square brackets, a three-dimensional arrays with three pairs and so on. In general terms, a multidimensional array can be represented as

storage_class data_type array_name[exp1][exp2]…[expN];

where storage_class refers to the storage class of the array, data_type is its data type, array_name is the array name, and exp1, exp2, and expN are positive integer expressions that indicate the number of array elements associated with each subscript. Note that the storage_class is optional and the default value for storage class is automatic for arrays that are defined inside of a function, and external for arrays defined outside of a function.

Three-Dimensional Arrays

A three-dimensional array can be thought of as an array of two-dimensional arrays. A typical three-dimensional array is the cube where first subscripts represent x side of the cube, second subscript shows the y side and third subscript represents z side of the cube. We can think a three-dimensional array as a collection of matrices.

A three-dimensional array with m matrices, each having r rows and c columns, will contain m*r*c elements. The array is declared as

int x[m][r][c] ; /* x is three-dimensional integer array with m matrices, each having r rows and c columns and total number of elements is m*r*c */

The name of an array represents the first element of the array i.e., the array x denotes element x[0][0][x]. The elements for matrix 1 i.e., m = 0, is given below:

	column 1	column 2	…	column c
row 1	x[0][0][0]	x[0][0][1]	…	x[0][0][c–1]
row 2	x[0][1][0]	x[0][1][1]	…	x[0][1][c–1]
.	.	.	.	.
row r	x[0][r–1][0]	x[0][r–1][1]	…	x[0][r–1][c–1]

The subsequent matrix can be represented by incrementing the value of first subscript (i.e., m) by one unit we reach to m–1.

Now consider the memory mapping for three-dimensional arrays. In three-dimensional array we think of data being arranged in array of matrices. However, computer's memory is arranged as a row of cells (i.e., locations). Thus, these structures of three-dimensional array must be simulated. We first calculate the amount of storage area needed by the three-dimensional and allocate a block of contiguous memory locations of that size.

The above declaration required m*r*c continuous memory location. The size of each element is two bytes i.e., S. Thus total memory reserved by the compiler is 2*m*r*c bytes for the array x.

The memory location of an element in memory depends upon the order in which the array elements are stored. The memory location of an element x[i][j][k] where $0 \le i \le m-1$ and $0 \le j \le r-1$, and $0 \le k \le c-1$ must skip i matrices, each having r rows and c columns and then j rows, each having c elements and k columns of $(j+1)^{th}$ row, that is the total number of elements to be skipped is i*r*c + j*c + k.

If the array stores **row by row** of each matrix in memory locations then the address of element x[i][j][k] would be

= B + i*r*c*S + j*c*S + k*S = B + S*(i*r*c + j*c +k)

where B is the base address of the array x i.e., the address of first element (x[0][0][0]).

And thus the element number is = (i*r*c + j*c +k) +1, here 1 is added because element number starts from 1 but the position of an element in the array starts from 0.

For example a three-dimensional array x with 4 matrices, each having 4 rows and 6 columns declared as

int x[4][4][6] ; /* x is three-dimensional integer array with 4 matrices, each having 4 rows and 6 columns i.e., total elements in the array are 4*4*6 = 96 */

Here size of each element is 2 bytes, thus the array required 2*96 =192 continuous memory block. Assume that the base address of the array is 512 (in decimal) bytes that means the first element of the array x (x[0][0][0]) is stored at memory location 512.

Now compute the address or memory location of an element x[3][2][3] i.e., the elements of third matrix, meet at 2^{nd} row and 3^{rd} column in the array x. Keep in mind that the elements are stored in memory row by row, thus we must skip 3 matrices, each having 4 rows and 6 columns, then skip 2 rows, each having 6 elements and 3 columns of 3^{rd} row. The memory location of the element would be at:

= 512 + (3*4*6 + 2*6 + 3)*2
= 512 + 87*2
= 686 bytes

and the element number would be 88.

Keep in mind that array indexes always start from 0 and upper bound is always less than 1 to the expression used for the subscript.

In similar manner we can compute the address of the element of an array's stored data **column by column** in the memory.

The memory location of an element in memory depends upon the order in which the array elements are stored. The memory location of an element x[i][j][k] where $0 \le i \le m-1$ and $0 \le j \le r-1$, and $0 \le k \le c-1$ must skip i matrices, each having r rows and c columns and then k columns, each having r elements and j rows of $(k+1)^{th}$ column, that is the total number of elements to be skipped is i*r*c + k*r + j.

If the array stores column by column of each matrix in memory locations then the address of element x[i][j][k] would be

= B + i*r*c*S + k*r*S + j*S = B + S*(i*r*c + k*r +j)

And thus the element number is = (i*r*c + k*r +j) +1, here 1 is added because element number starts from 1 but the position of an element in the array starts from 0.

Consider the Same Example of Array x as Declared Above

Now compute the address or memory location of an element x[3][2][3] i.e., the element of third matrix meets at 2nd row and 3rd column in the array x. Keep in mind that the elements are stored in memory column by column, thus we must skip 3 matrices, each having 4 rows and 6 columns, then skip 3 columns, each having 4 elements and 2 rows of 4th column. The memory location of the element would be at:

= 512 + (3*4*6 + 3*4 + 2)*2
= 512 + 86*2
= 684 bytes

and the element number would be 87.

Now we can easily locate each element address, element number and its indexes in the three-dimensional arrays.

In similar manner we can compute the memory address, element number and indexes of an element of any dimensional arrays.

Initializing a three-dimensional array is similar to two-dimensional array. Consider the following array definition.

int tables[2][2][3] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}; /* tables is three-dimensional integer array with 2 matrices, each having 2 rows and 3 columns and total number of elements i.e., 2*2*3 = 12 */

In the above initialization of three-dimensional array tables, first assigned the elements of first matrix and within the matrix assigned the first row then second row and so on i.e., row major order. Repeat this process until all elements have been assigned initial value. The number of values cannot exceed the number of elements in the array. If there are too few values within a brace, the remaining elements of the array will be assigned zero.

The individual elements of the array tables are as below:

column 1	column 2	column 3	
tables[0][0][0] = 0	tables[0][0][1] = 1	tables[0][0][2] = 2	/* table 1, row 1 */
tables[0][1][0] = 3	tables[0][1][1] = 4	tables[0][1][2] = 5	/* table 1, row 2 */
tables[1][0][0] = 6	tables[1][0][1] = 7	tables[1][0][2] = 8	/* table 2, row 1 */
tables[1][1][0] = 9	tables[1][1][1] = 10	tables[1][1][2] = 11	/* table 2, row 2 */

All the data structure operations can be implemented in three-dimensional arrays or any multidimensional arrays as we have done with one-dimensional and two-dimensional arrays.

Arrays Handling in Other Languages

In pascal language, array declaration is quite different from 'C'. The general form of type declaration of one-dimensional arrays in pascal is:

```
type
    array_type identifier = array[index_type] of element_type;
where
    array_type identifier indicates the type name;
    index_type specifies the data array of the values to be used as subscripts.
element_type indicates the type of values that are going to be stored in the array.
Let us consider the following declarations:
Var
    x  : array [1..10] of integer;
    y  : array [–5..4] of real;
```

the first declaration tells the computer to allocate enough space for the variable x to store 10 integers. The second declaration tells the computer to allocate enough space for the variable y to store 10 reals. Since a real number takes more space than an integer the storage allocated would not be same.

Note that the lower bound may be any integer number. The number of elements in the array i.e., the size of array can be calculated as

= upper bound – lower bound + 1

Thus in first declaration the size is 10 (10 – 1 + 1) and in second declaration the size is also 10 as follows

= 4 – (–5) +1 = 10

In pascal indexes can be any integer number.

Two-dimensional arrays have two subscripts, three-dimensional arrays have three subscripts and so on. Let us consider the following declarations:

```
Var
    m  : array [1..3, 1..3] of integer;
    n  : array [–5..–2, 0..2] of real;
```

The first declaration tells the computer to allocate space for 9 integer elements for array m, with 3 rows and 3 columns. The second declaration tells the computer to allocate space for 12 real elements for the array n, with 4 rows and 3 columns.

The number of rows and columns can be calculated as we compute the size of one-dimensional array.

Number of rows in array m = upper bound – lower bound + 1 = 3 – 1 + 1
= 3

Number of columns in array m = upper bound – lower bound + 1 = 3 – 1 + 1
= 3

Number of rows in array n = upper bound – lower bound + 1 = –2 – (–5) + 1
= 4

Number of columns in array n = upper bound – lower bound + 1 = 2 – 0 + 1
= 3

We may now generalize the expression for a two-dimensional array

$x[lb_1..ub_1, lb_2..ub_2]$

where lb_1 and lb_2 are lower bounds of the two subscript ranges and ub_1 and ub_2 are the upper bounds of the two subscript ranges.

The logical representation of the array x with m rows and n columns are as given below

	column 1	column 2		column n
row 1	$x[lb_1, lb_2]$	$x[lb_1, lb_2 + 1]$	…	$x[lb_1, ub_2]$
row 2	$x[lb_1 + 1, lb_2]$	$x[lb_1 + 1, lb_2 + 1]$	…	$x[lb_1 + 1, ub_2]$
⋮	⋮	⋮		⋮
row m	$x[ub_1, lb_2]$	$x[ub_1, lb_2 + 1]$	…	$x[ub_1, ub_2]$

The memory location of an element in memory depends upon the order in which the array elements are stored. The memory location of an element x[i,j], where $lb_1 \le i \le ub_1$ and $lb_2 \le j \le ub_2$, must skip $i - lb_1$ rows, each having $(ub_2 - lb_2 + 1)$ elements and $j - lb_2$ elements, which precede element x[i, j].

If the array stores row by row i.e., **Row Major Order** in memory locations then the address of element x[i, j] would be at:

$$= B + (i - lb_2)*(ub_2 - lb_2 + 1)*S + (j - lb_2)*S$$

The row major order varies the subscripts in right (last one) to left (first one) order.

For example consider a two-dimensional array m[1..4,1..6] with 4 rows and 6 columns.

Here size of each element is 2 bytes, thus the array required 48 continuous memory locations. Assume that the base address of the array is 2014 (in decimal) byte that means the first element of the array m is stored at memory location 2014.

Now compute the address or memory location of an element m[2,3] i.e., the element meets at 2nd row and 3rd column in the array m. Put the values in the above formula to compute the address as

$$= 2014 + (2 - 1)*(6 - 1 + 1)*2 + (3 - 1)*2$$
$$= 2014 + 12 + 4$$
$$= 2030$$

Column Major Order

The memory location of an element x[i,j] where $lb_1 \le i \le ub_1$ and $lb_2 \le j \le ub_2$, must skip $(j - lb_2)$ columns, each having $(ub_1 - lb_1 + 1)$ elements and $i - lb_1$ elements, which precede element x[i,j].

If the array stores column by column i.e., Column Major Order in memory locations then the address of element x[i,j] would be at:

$$= B + (j - lb_2)*(ub_1 - lb_1 + 1)*S + (i - lb_1)*S$$

The column major order varies the subscripts in left (first one) to right (last one) order.

For example the elements of a two-dimensional array m[–2..1,–1..1] would be stored in the memory in the sequence as given below:

In Row Major Order		In Column Major Order	
m[–2,–1]	row 1	m[–2,–1]	column 1
m[–2,0]		m[–1,–1]	
m[–2,1]		m[0,–1]	
m[–1,–1]		m[1,–1]	
m[–1,0]	row 2	m[–2,0]	column 2
m[–1,1]		m[–1,0]	
m[0,–1]		m[0,0]	
m[0,0]	row 3	m[1,0]	column 3
m[0,1]		m[–2,1]	
m[1,–1]		m[–1,1]	
m[1,0]	row 4	m[0,1]	
m[1,1]		m[1,1]	

For example consider a two-dimensional array m[1..4, 1..6] with 4 rows and 6 columns.

Here size of each element is 2 bytes, thus the array required 48 continuous memory block. Assume that the base address of the array is 2014 (in decimal) byte that means the first element of the array m is stored at memory location 2014.

Now compute the address or memory location of an element m[2, 3] i.e., the element meets at 2nd row and 3rd column in the array m. Put the values in the above formula of column major order to compute the address as

$$= 2014 + (3 - 1)*(4 - 1 + 1)*2 + (2 - 1)*2$$
$$= 2014 + 16 + 2$$
$$= 2032$$

We may generalize it for an n-dimensional array $x[lb_1..ub_1, lb_2..ub_2,\ldots,lb_n..ub_n]$. The elements would be stored in the following row major order.

$x[lb_1,lb_2,\ldots,lb_n]$
$x[lb_1,lb_2,\ldots,lb_n+1]$
.
.
.
$x[lb_1,lb_2,\ldots,ub_n]$
$x[lb_1,lb_2,\ldots,lb_{n-1}+1,lb_n]$
.
.
.
$x[lb_1,lb_2,\ldots,ub_{n-1},ub_n]$
.
.
.
$x[ub_1,ub_2,\ldots,ub_{n-1},ub_n]$

The elements would be stored in the following column major order.

$x[lb_1,lb_2,\ldots,lb_n]$
$x[lb_1+1,lb_2,\ldots,lb_n]$
.
.
.
$x[ub_1,lb_2,\ldots,lb_n]$
$x[lb_1+1,lb_2+1,\ldots,lb_n]$
.
.
.
$x[ub_1,ub_2,\ldots,lb_{n-1},lb_n]$
.
.
.
$x[ub_1,ub_2,\ldots,ub_{n-1},ub_n]$

The number of elements in the array x will be $(ub_1 - lb_1 + 1)* (ub_2 - lb_2 + 1)*\ldots* (ub_n - lb_n + 1)$. We can compute the memory location of an element of n-dimensional array in a similar manner as we have computed memory location for two-dimensional (i.e., n = 2) and three dimensional arrays (i.e., n = 3).

3.4 ORDERED ARRAYS

An array in which the data elements are stored in ascending order (or descending), that is, with the smallest value at index 0, and each successive index holding a value larger than the index below it. Such an array is called an ordered array. Here we are assuming single dimension array i.e., vector.

Insertion in Ordered Arrays

When we insert an element by its value into this array, the correct location must be found for the insertion, just above a smaller value and just below a larger one. Then all the larger values must shift downward towards upper bound to make a space for the inserted element. Insertion into ordered array required O(n) time as compared to insertion in unordered array required constant time O(1) for array of n-element.

Let us consider an array dice with 6 integer elements. Initially the ordered array consists of 3 elements (i.e., nElem = 3) in ascending order as given below in Fig. 3.3(a).

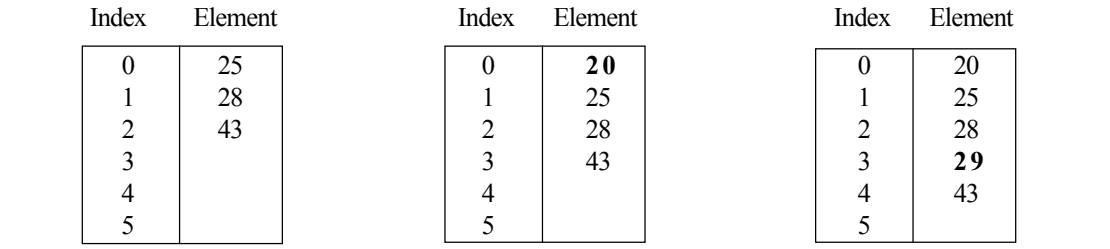


Figure 3.3 (a) *Initially array has 3 elements* (b) *Insert an element value 20* (c) *Insert an element value 29*

Deletion in Ordered Arrays

Deletion in the ordered arrays is similar to deletion in unordered arrays. It required searching. Here an element is found with binary search. Deletion into ordered arrays required O(n) time for the n–element array.

Binary Search for Ordered Arrays

Binary search works for ordered (sorted) arrays and is a very efficient searching technique. It is used to find the location of an element in the given array. The time complexity of binary search is $O(\log_2 n)$ as compare to linear search complexity O(n). The binary search procedure works as follows:

To find an element in an array of N elements, we first set the range of index(es) in which the element to be searched. In this array range specified from lower bound (i.e., 0) to upper bound (i.e., N – 1). Initially the current index, mid, is set to the middle of this range i.e., (0 + N – 1)/2

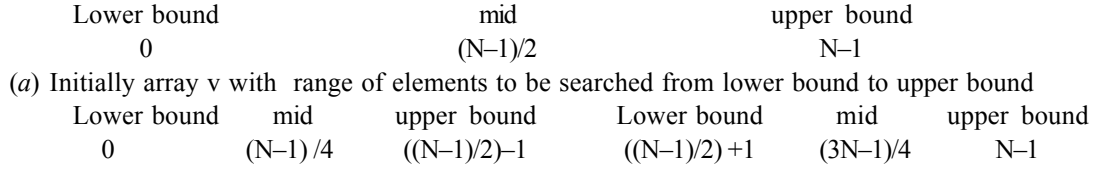
The value of the mid index of the array is compared with the search element. If it matches then we find the element at index mid of the array.

If the search element value is less than the value at mid index of the array then the upper bound of the array is shifted to first half of the array (i.e., upper bound equal to mid – 1). Now search is made in the first half of the array.

If the search element value is greater than the value at mid index of the array then the lower bound of the array is shifted to second half of the array (i.e., lower bound equal to mid +1). Now search is made in the second half of the array.

This procedure is repeated until the required element is found or the search range becomes empty (i.e., upper bound < lower bound).

This procedure is called a binary search because at each step, we reduce the range of the elements to be searched by half and the array is divided into two equal subarrays. The below graphical representation (Fig. 3.4) divided the range in a binary search for N elements array v.



(*a*) Initially array v with range of elements to be searched from lower bound to upper bound

(*b*) New range if search element < v[mid]

(*c*) New range if search element > v[mid]

Figure 3.4 *Dividing the range in logarithmic manner for a binary search.*

We state the binary search algorithm formally as given below:

Algorithm BSEARCH(v, lb, ub, sElem, loc)

/* Here v is a nElem sorted array with lower bound lb and upper bound ub and sElem is an element to be searched and if found then return the index in loc otherwise return–1. The local variables low, mid, high denote, respectively, the beginning, middle and end locations of the array elements */

Step 1 : [Initialize variables]
Set low = lb ; high = ub; mid = (low + high)/2; and loc =–1;

Step 2 : Repeat steps 3 and 4 while low<= high and v[mid] != sElem

Step 3 : if sElem < v[mid] then
high = mid–1;
else
low = mid + 1;

Step 4 : mid = (low + high)/2;

Step 5 : if v[mid] = sElem then
loc =mid;

Step 6 : return (loc);

Step 7 : end BSEARCH

Note that if sElem does not appear in v array, the algorithm eventually arrives at the stage low = mid = high. In next time when high<low, control transfers to Step 5 of the algorithm.

Let us apply this algorithm to an example. Suppose array v contains 10 elements as

10, 15, 22, 27, 34, 41, 51, 56, 70, 77

and that we wish to search for an element value 27. The algorithm works as follows:

Step 1 : Initialize the variables and assign the elements value of the array v.

Elements	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]
	10	15	22	27	34	41	51	56	70	77
Index →	0	1	2	3	4	5	6	7	8	9
	↑ low				↑ mid					↑ high
	(lower bound)									(upper bound)

as mid = (low + high)/2 = (0 + 9)/2 = 4 , and loc = –1

Step 2 : Test loop condition is low<=high? Yes and (v[4] != 27)? Yes (i.e., loop test success)

Step 3 : Is 27<v[4]? Yes, then high = mid –1 = 3

Step 4 : mid = (low + high)/2 = (0 + 3)/2 = 1 and control transfer to Step 2 and current position of the variables in the array is shown below:

	10	15	22	27	34	41	51	56	70	77
Index →	0	1	2	3	4	5	6	7	8	9
	↑ low	↑ mid		↑ high						

Step 2 : Test loop condition is low<=high? Yes and (v[1] != 27)? Yes (i.e., loop test success)

Step 3 : Is 27<v[1]? No, else low = mid +1 =2

Step 4 : mid = (low + high)/2 = (2 +3)/2 = 2 and control transfer to Step 2 and current position of the variables in the array is shown as follows:

	10	15	22	27	34	41	51	56	70	77
Index →	0	1	2	3	4	5	6	7	8	9
			↑ low mid	↑ high						

Step 2 : Test loop condition is low<=high? Yes and (v[2] != 27)? Yes (i.e., loop test success)

Step 3 : Is 27<v[2]? No, else low = mid +1 =3

Step 4 : mid = (low + high)/2 = (3 +3)/2 = 3 and control transfer to Step 2 and current position of the variables in the array is shown below:

	10	15	22	27	34	41	51	56	70	77
Index →	0	1	2	3	4	5	6	7	8	9
				↑ low mid high						

Step 2 : Test loop condition is low<=high? Yes and (v[3] != 27)? No (i.e., loop test fail)

Step 5 : Is v[3] = 27? Yes, loc =mid =3

Step 6 : return (3)

Let us consider a case of unsuccessful search. Search an element value 45 in the above given array v. Here we are demonstrating the search in the following steps.

1. initially low =0, high=9 and mid = (low + high)/2 =(0+9)/2 = 4, v[4]= 34 and loc = –1
2. since 45 > v[4], low has its value changed by low = mid + 1 = 4 + 1 = 5 and hence
   mid = (5 + 9)/2 = 7 and v[7] = 56
3. since 45 <v[7], high has its value changed by high = mid –1 = 7 – 1 = 6 and hence
   mid = (5 + 6)/2 = 5 and v[5] = 41
4. since 45 >v[5], low has its value changed by low = mid +1 = 6 and hence
   mid = (6 + 6)/2 = 6 and v[6] = 51

   (Note that low = mid = high)

Since 45 < v[6], high has its value changed by high = mid –1 = 5. But now low > high, hence search element does not appear in the array v. The graphical representation of the array processing can be shown in Fig. 3.5.

1. Elements	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]
	10	15	22	27	34	41	51	56	70	77
Index →	0↑ low	1	2	3	↑4 mid	5	6	7	8	9↑ high

2.	10	15	22	27	34	41	51	56	70	77
Index →	0	1	2	3	4	↑ 5 low	6	↑ 7 mid	8	↑ 9 high

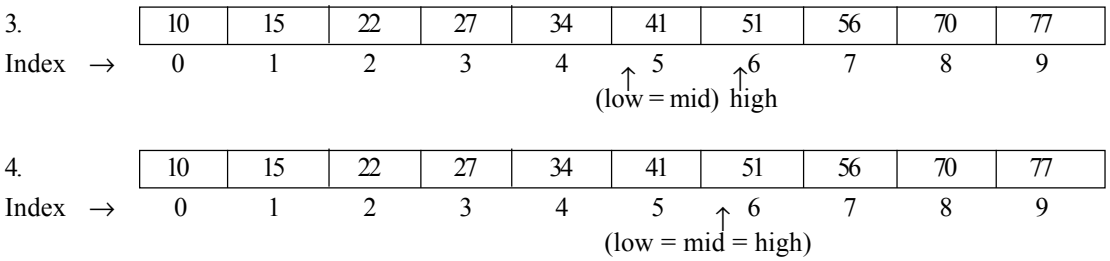


Figure 3.5 *Binary search for an element value 45*

The Pros/cons of Using Ordered Arrays

The biggest advantage of ordered array is that we can speed up search times dramatically using a binary search. The disadvantage of ordered array is that it slows down insertion because all the data elements with a higher value must be moved towards the upper bound. Deletions are slow in both ordered and unordered arrays because elements must be shifted towards the lower bound to fill the gap left by the deleted element. In each insertion upper bound increased by one and in each deletion upper bound of the array decreased by one.

Ordered arrays are therefore useful in the applications where searches are frequent, but insertions and deletions are not. Ordered arrays might be useful for an application of employee personnel information system, where hiring new employees and laying off existing ones would probably be infrequent occurrence compared with accessing an existing employee's information.

Comparison Between Linear Search and Binary Search

A binary search provides a significant speed increase over a linear search. The average number of comparison for input size of n elements for linear search is n/2, whereas binary search required only $\log_2 n$. The following table shows the number of input elements and the number of comparison required for the search algorithms.

Size of input (n)	*Input size expressed as power of 2*	*Comparisons required in binary search ($\log_2 n$)*	*Comparisons required in linear search (n/2)*
8	2^3	3	4
16	2^4	4	8
32	2^5	5	16
64	2^6	6	32
128	2^7	7	64
256	2^8	8	128
512	2^9	9	256
1024	2^{10}	10	512
32676	2^{16}	16	16338

The difference between binary search and linear search is obvious from the above table. For example searching 10 elements would take an average of five comparisons with a linear search (n/2), and a maximum of four comparisons with a binary search i.e. ($2^3 < 10 < 2^4$).

Now consider an ordered array 'C' program which inserts an element value, removes an element, searches an element with binary search. The complete listing of ordArray.cpp program is given as follows:


```
/*ordArray.cpp is C program to insert, delete and binary search an element in ordered array */
  #include <stdio.h>
  #define MAXSIZE 100
  void main()
  {
            int ordA[MAXSIZE];
            int sElem, pofElem, dElem,iElem;
            int nElem = 0;
            int i,ch;
            int insElem(int[], int, int);
            int delElem(int[], int, int);
            int bSearch(int[], int,int);
            void display(int[], int);
            do      {
                    printf("\n Enter your choice");
                    printf("\n 1. Insert an element");
                    printf("\n 2. Delete an element");
                    printf("\n 3. Search an element");
                    printf("\n 4. Display ordered array elements");
                    printf("\n 5. Exit \n");
                    scanf("%d", &ch);
                    fflush(stdin);
                    switch (ch)
                    {
                        case 1:
                              printf("\n Please Enter the element value to be inserted:");
                              scanf("%d", &iElem);
                              nElem = insElem(ordA, nElem, iElem);
                              break;
                        case 2:
                              printf("\n Please Enter the element value to be deleted:");
                              scanf("%d", &dElem);
                              nElem =delElem(ordA, nElem,dElem);
                              break;
                        case 3:
                              printf("\n Please Enter the element value to be searched:");
                              scanf("%d", &sElem);
                              pofElem = bSearch(ordA, nElem,sElem);
                              If (pofElem == – 1)
                                        (printf ("In Element Not Found");
                              else
                              printf("\n Element found at location %d",pofElem);
                              break;
                        case 4:
```

```
                        display(ordA, nElem);
                    }                                   /* end of switch statement */
    }while (ch! = 5);   /* End of do–while loop */
}/* end of main function */
int insElem(int ordA[MAXSIZE], int nElem, int iElem)
{
            int j,k;

            for(j = 0;j<nElem;j++) /* find where it goes using linear search */
              if (ordA[j] > iElem)
                        break;
            for(k = nElem; k>j;k––)  /* moves bigger ones down towards upper
                    bound  */
            ordA[k] = ordA[k–1];
            ordA[j] = iElem;      /* insert it */
            nElem++;               /* increment size */
            return nElem;
} /* end of insElem function */
int delElem(int ordA[MAXSIZE],int nElem, int delElem)
{
            int i,pofElem;
            pofElem = bSearch(ordA, nElem, delElem);
            if (pofElem == –1)
                    printf("\n Element can't find, not deleted");
                        else
                    {
                    for(i = pofElem; i<nElem ;i++) /* this for loop shifts all the elements of the array
                    dice upwards to fill up the gap created by element at position pofElem */
                    ordA[i] = ordA[i+1];
                    nElem––;
                    }
            return nElem;
}
/* Function to search an element in the ordered array using binary search */
int bSearch(int ordA[MAXSIZE],int nElem, int sElem)
{
int low  = 0;
int high = nElem –1;
int mid = (low + high) /2;
while ((low< = high) && (ordA[mid]! = sElem))
{
        if (sElem < ordA[mid] )
            high = mid –1;
```

```
            else
                    low = mid +1;
            mid = (low + high)/2;
    }
    if (ordA[mid] == sElem)
            return mid;
    else
            return –1;
} /* end of bSearch function */
void display(int ordA[MAXSIZE], int nElem)
{
    int i;
    for(i = 0;i<nElem;i++)
    printf("\nElement %d: %d", i+1,ordA[i]);
}
```

Analysis

In this program, we created four functions:

Function insElem has three formal variables i.e., array ordA, number of elements in the array nElem, and element to be inserted iElem. This function inserts an element in the ordered (here in ascending order) ways using linear search and increased number of elements by one and return it.

Function delElem has three formal variables i.e., array ordA, number of elements in the array nElem, and element to be deleted dElem. This function deletes an element in the ordered array using binary search to find the deleted element in the array and decreased number of elements by one and return it.

Function bSearch has three formal variables i.e., array ordA, number of elements in the array nElem, and element to be searched sElem. This function searches an element in the ordered array using binary search.

Function display has two formal variables i.e., array ordA, number of elements in the array nElem and displays the all array elements.

The main function calls all four functions in a do-while loop. Hence we can insert, delete, search an element as many times as possible.

The program always maintained ascending order.

Modification

This program can be further modified to implement all four functions into two-dimensional array e.g. matrix. We can insert elements in matrix, delete elements from matrix and search elements in a matrix.

3.5 SPARSE MATRIX

Matrices have a relatively higher number of elements having zero values than the non-zero elements. The general types of sparse matrices are shown in Fig. 3.6. The matrix where nonzero elements can only occur on the diagonal or on above or below the diagonal is called a tri-diagonal matrix [see Fig. 3.6(a)]. The matrix where all entries above the main diagonal are zero or equivalently where nonzero elements can only occur on or below the main diagonal is called a lower triangular matrix [see Fig. 3.6(b)]. The matrix where all entries below the main diagonal are zero or equivalently where nonzero elements can only occur on or above the main diagonal is called upper triangular matrix [see Fig. 3.6(c)].

$$\begin{Bmatrix} 2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 2 & 1 \end{Bmatrix} \quad \begin{Bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 & 0 & 0 \\ 4 & -1 & -2 & 0 & 0 & 0 \\ -2 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 7 & 8 & 0 \\ 2 & 1 & 0 & 3 & 2 & 5 \end{Bmatrix}$$

(a) (b)

$$\begin{Bmatrix} 2 & 3 & -1 & -2 & 0 & 1 \\ 0 & 2 & -1 & 2 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & -2 \\ 0 & 0 & 0 & -2 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{Bmatrix}$$

(c)

Figure 3.6 (a) *Tri-diagonal matrix* (b) *Lower triangular matrix* (c) *Upper triangular matrix*

The two-dimensional arrays are always used to store matrix in the memory. But to store sparse matrix as two-dimensional arrays, there would be much wasted memory space. That is, just store only nonzero elements of the matrix.

For example consider tri-diagonal matrix of Fig. 3.6(a), having 36 elements, in which 12 elements are nonzero and remaining 24 elements are zero.

An mxn matrix *A* is said to be *sparse* if many of its elements are zero. A matrix that is not sparse is called *dense* matrix. It is not possible to define an exact boundary between dense sparse matrices. The diagonal and tri-diagonal matrices fit into the category of sparse matrices. In this section, we will consider sparse matrices with an irregular or unstructured nonzero region. Figure 3.7 shows 6 × 7 sparse matrix with ten nonzero elements out of 42 elements.

$$\text{M[][]} = \begin{Bmatrix} 0 & 0 & 6 & 0 & 9 & 0 & 0 \\ 2 & 0 & 0 & 7 & 8 & 0 & 4 \\ 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 5 \end{Bmatrix}$$

Figure 3.7

The nonzero elements are stored

In row 1	M[0][2] = 6	M[0][4] = 9		
In row 2	M[1][0] = 2	M[1][3] = 7	M[1][4] = 8	M[1][6] = 4
In row 3	M[2][0] = 10			
In row 4	M[3][2] = 12			
In row 5	no nonzero element			
In row 6	M[5][3] = 3	M[5][6] = 5		

Representing Sparse Matrices

1. Array representation

In array representation, an array of structure of three elements or three single dimension arrays is defined as

< row, column, element >

is used to store non-zero element.

Row	*Column*	*Element*
0	2	6
0	4	9
1	0	2
1	3	7
1	4	8
1	6	4
2	0	10
3	2	12
5	3	3
5	6	5

The structure declaration for array is as:

```
#define MAXSIZE 100
struct typematrix{
        int row;
        int column;
        int element;
} spmatrix[MAXSIZE];
```

The maximum number of nonzero elements in the sparse matrix is defined by constant MAXSIZE. The array representation will use *(2*(n + l)*sizeof{int) + n*sizeof{T)]* bytes where *n* is the number of nonzero elements and *T* is the data type of elements.

The major limitation of array representation of a sparse matrix is that we need to know the number of nonzero elements in the matrix.

This scheme generally allows faster execution of matrix operations and is more efficient than linked allocation scheme. But the insertion and deletion of matrix element necessitate the displacement of many other elements.

2. An alternate approach is to use a pointer–based representation.

It contains one header node that has four fields—nrow (number of rows), ncol (number of columns), num (number of nonzero elements), and head (a point first row containing at least one nonzero element).

A complete 'C' program to show the sparse matrix for given row, column and its values.

ARRAY/SPARSE.CPP

```
/* program sparse.cpp, is array implementation of sparse matrix. */
#include<stdio.h>
#define MAXSIZE 20
struct typematrix{
        int row;
```

```
        int col;
        int element;
} spmatrix[MAXSIZE];
void main()
{
int i, j, n, row, col, k = 0;
printf("\n Enter matrix dimension :");
scanf("%d%d", &row, &col);
printf("\n Enter number of non–zero elements in sparse matrix:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\n Enter element row, column and its value");
scanf("%d%d%d", &spmatrix[i].row,&spmatrix[i].col,&spmatrix[i].element);
}
printf("\n Sparse matrix is ...");
for(i = 0;i<row;i++)
{
printf("\n");
for(j = 0; j<col;j++)
{
        if ((spmatrix[k].row == i+1) && (spmatrix[k].col == j+1))
        {
                printf("\t%d", spmatrix[k].element);
                k++;
          }
else
        printf("\t0");
    }
  }
} /* end of main */
```

Chapter 4

Linear Data Structure—Linked List

4.1 LINEAR LINKED LIST AND OPERATIONS

Here we will first discuss the term list. What is a list?

A list is a linear sequence of data objects of the same type. Mathematically, a list is a finite sequence of $n \geq 0$ elements, $x_1, x_2, \ldots, x_n$ where x_i are either atoms or lists. The elements x_i, $1 \leq i \leq n$ which are not atoms are said to be the sublists.

Consider the list A=$\{ x_1, x_2, \ldots, x_n\}$, A is the name of the list and n its length and x_1 is the head of the list and x_n is the tail of the list.

Examples of list are a shopping list, a student address list, result list and any real-life events can be present in lists. A few examples are listed below.

A list of states in India = { "Rajasthan", "Uttar Pradesh", "Andhra Pradesh", "Kerala", "Delhi"};

A list of STD Codes = {0141, 011, 022, 033, 044, 040, 0731, 0145};

A list of hotels = {" Taj Hotel", "Mugal Hotel", "Goa Resorts", "Lake Palace"}

In each of the above list shows the atoms, we can access list via element number, e.g. 3rd element of each list respectively be "Andhra Pradesh", 022, "Goa Resorts".

The below list B is the empty or null, that means its length is zero

B = { }

The below list C is of length 2, its first element is the atom 1 and its second element is the linear list {2, 3}.

C = { 1, {2, 3}}

The below list D is of length three where first element is the atom, the second element is the list C and the third element is the list B.

D = {4, C, B}

When we create a list the first addition would be the first element, second addition would be the second element and so on. Thus we can define a linear list is an ordered set consisting of a variable number of elements to which insertion and deletion can be made. A linear list displays the logical adjacency among the list items.

The first element of a list is called head of list and the last element is called the tail of list. The next element to the head of the list is called its successor. The previous element to the tail (if it is not the head of the list) of the list is called its predecessor.

Clearly, a head does not have a predecessor, and a tail does not have a successor. Any other element of the list has both one successor and one predecessor.

The elements in a list are tied together by their successor–predecessor relationship.

A list can be implemented statically or dynamically using array index or pointers (links).

Static Implementation of Lists

It is the simplest implementation. Its size is fixed and allocated at compilation time. A list can be implemented as an array. The array was linearly ordered, and this ordering relation was preserved in the corresponding storage structures by using sequential allocation. There was no need for an element to specify where the next element would be found (See arrays).

If a list is consisting of n nodes (i.e., data items), then we can store the address of each node in a vector (i.e., single dimension array) consisting of n elements. The first element of the vector contains the address of the first node of the list, the second element contains the address of the second node, and so on.

Example 1. Suppose that the list of STD Codes is initialized in the ten elements array list. The elements are sequentially stored in list[0], list[1],…,list[7] and list[8], list[9] are allocated but not used.

List[0]	0141
List[1]	011
List[2]	022
List[3]	033
List[4]	044
List[5]	040
List[6]	0731
List[7]	0145
List[8]	} Allocated but
List[9]	} not used

The predecessor of list[0] (or head) is NULL, list[7] is the tail of the list and has no successor. We may tabulate the predecessors and successors of the other elements of the list as given below:

Data	*Array index*	*Predecessor index*	*Successor index*
0141	0	NULL	1
011	1	0	2
022	2	1	3
033	3	2	4
044	4	3	5
040	5	4	6
0731	6	5	7
0145	7	6	NULL
Unused	8		
locations	9		

Since the elements are sequentially stored, we don't need to store the predecessor and successor indices. Any element in the list can be accessed through its index.

This type of list implementation severally suffers from the heavy data movements when it required insertion and deletion arbitrary from the list.

If the list nodes are not ordered by array ordering, then each must contain within itself a pointer to its successor. Let us consider a similar kind of problem. The ILIM Ahmedabad is going to hold a group discussion for 100 students. They are given an option to select one topic out of four topics for discussion. The organizers are required to prepare four lists according to student group discussion topic.

One way to do this is use four different arrays, each declared to hold the maximum number of students (it may be possible all students will select same topic). This means we have to allocate a total of 4*100 = 400 locations. These result in wastage of lots of space i.e., 300 locations are allocated extra. This is similar to the above static representation of list using array.

Another way is to maintain all four lists in a single dimensional array of maximum size. It required each list start index and end index in the array and each list element must contain a pointer to its successor. Then the list is called linked list as depicted in Fig. 4.1.

Thus each node of the list required two fields. First one is the information field (i.e., Info) and second is the array index value for the successor (i.e., Next). Here nodes of list are placed anywhere in the array instead of being located a fixed location apart as in case of sequential representation. The index of the first node of the list is given in the pointer variable start. The index value for the successor of the last node is –1, which indicated the end of the list. In the linked representation we store the array index of the next element with the data item (i.e., node). Thus the successive elements in the list need not occupy adjacent space in the array. This will make it easier to insert and delete elements in the list. But all nodes of the list must be stored in the block of memory allocated to the array.

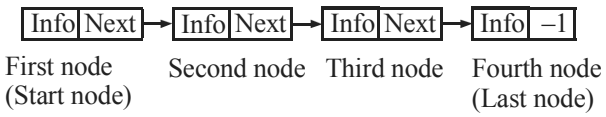


Figure 4.1 *Linked list with four nodes*

Static Implementation of Linked List Using Parallel Arrays

How is the linked list maintained in memory? Each node of the linked list has two parts: first is the element itself and second one is index value of its successor within the array. The easiest way to implement linked list using two parallel arrays: we call them here info and successor, such that info[k] and successor[k] contain information part and the index value for the successive node of the list respectively. The array successor must be integer type and contains only positive values because it stored the index value (i.e., k) of the array. The list pointer variable start contains the location of the beginning of the list.

The following examples of linked lists indicate that the nodes of a list need not occupy adjacent elements in the arrays info and successor, and more than one lists may be maintained in the same linear arrays, info and successor. However, each list must have its own pointer variable giving the locations of its first node.

Example 2. Figure 4.2 illustrates a linked list in memory where each node of the list contains a single character. The arrays info and successor are declared to store ten elements. The array info stored the characters and array successor stored the index value, which is the next node index. The start variable value is 9 i.e., the first node of the list is info[9].

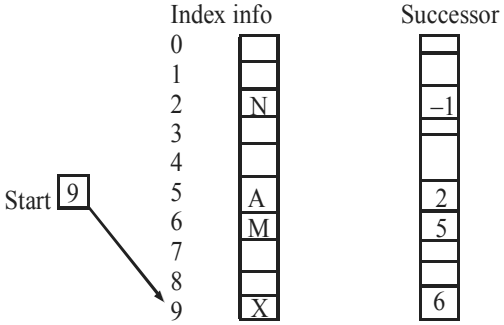


Figure 4.2 *Linked list representation using two parallel arrays, info and successor*

We can obtain the actual list of characters as follows:

Start = 9, so info[9] = X is the first character (first node).

Successor[9] = 6, so info[6] = M is the second character (second node).

Successor[6] = 5, so info[5] = A is the third character (third node).

Successor[5] = 2, so info[2] = N is the fourth character (fourth node).

Successor[2] = –1, negative or illegal index value, so the list has ended.

The list contains the four characters X, M, A, N.

The drawback of the above representation is that it required two arrays. If the information part of a list is recorded (i.e., collection of more than one data items) then it would require a collection of parallel arrays. For example to store an address list of students, where each node of the list contains six data items– name of the student, house number, colony, city, pin code, and state. Here we require seven arrays—six for the information part of the node and one for the successor (i.e., Next) pointer. The best way is to store node of the linked list in some type of record structure e.g. structures in 'C'.

In the array representation of linked list, we can store node of the list in the array of structures.

To declare an array of structures, a structure is first defined, and then an array variable of that type is declared. Like all array variables, arrays of structures begin indexing at 0.

Static Implementation of Linked Lists Using Array of Structures

The node of the linked list contains at least two fields: one for information part and one for next pointer. The structure must contain at least two fields to store the node of a list. In the below declaration of structure, we have considered that maximum nodes in the linked list are 100. The information part is containing one field rank that is integer type, and next pointer is also integer type because it denoted the index of the array.

The node of a list is declared as an array of structures.

```
#define MAXNODES
struct nodetype {
        int rank;          /* Information field  of the node */
        int next;          /* pointer to its successor */
        };
struct nodetype node[MAXNODES];
```

Note that node, next, nodetype, rank are just identifier names and we can consider any suitable name. The node is an array of 100 nodetype structures.

Here the pointer next to a node is represented by an array index. That is, the pointer is an integer between 0 and MAXNODES –1 that references a particular element of the array node. The null pointer is represented by the integer –1.

Example 3. This example illustrates the memory representation of four lists in the node array of structures. We can reference any node element via node[i].rank and next pointer field with node[i].next.

For example, suppose that data items in the list1 = {1, 10, 12}, list 2 = {6, 8 ,14, 7}, list 3 = { 3, 9, 13, 4} and list 4 = {2, 5, 18, 16}. List 1 starts at array index 3, list 2 starts at array index 17, list 3 starts at array index 13, and list 4 starts at array index 8 and i is an integer variable which indicates the index of the array node.

List 1 starts at index 3

Array index	*node[i].rank*	*node[i].next*
3	1	6
6	10	10
10	12	–1

List 2 starts at index 17

Array index	*node[i].rank*	*node[i].next*
17	6	14
14	8	9
9	14	7
7	7	–1

List 3 starts at index 13

Array index	*node[i].rank*	*node[i].next*
13	8	1
1	9	5
5	13	15
15	4	–1

List 4 starts at index 8

Array index	*node[i].rank*	*node[i].next*
8	2	0
0	5	16
16	18	12
12	16	–1

The nodes of a list may be scattered throughout the array node in any arbitrary order. Each node carries within itself the location of its successor until the last node in the list, whose successor/next value is –1, which is the null pointer. The below Fig. 4.3 illustrates an array node that contains four lists and i is the index of the array.

Index		*node[i].rank*	*node[i].next*	
	0	5	16	
	1	9	5	
	2			
list 1 =	3	1	6	
	4			
	5	13	15	
	6	10	10	
	7	7	–1	(i.e., null pointer)
list 4 =	8	2	0	
	9	14	7	
	10	12	–1	

Contd...

	Index	*node[i].rank*	*node[i].next*
	11		
	12	16	–1
list 3 =	13	3	1
	14	8	9
	15	4	–1
	16	18	12
list 2 =	17	6	14

Figure 4.3 *The node is an array of structures which stored four lists*

The array implementation of linked list speeds up the insertion and deletion. Let us consider deleting the data element 6 in list 2. The element 6 is the head of the list 2. We just start the list 2 from the next node at index 14 which will delete the element 6, but does not free the memory occupied by the element.

In similar manner, consider the insertion of an element 17 in list 3 after rank 9 and before rank 13. We can do this by just changing the successor index of rank 9 to 2 and setting the successor index of index 2 to successor index of rank 9. Here we have used the free location at index 2 to store a new element.

Limitations of the Array Implementation of List

1. Memory storage space is wasted, as the memory remains allocated to the array throughout the program execution even few nodes are stored. Because additional nodes are still reserved and their storage can't be used for any purpose.
2. List cannot grow in its size beyond the size of the declared array if required during program execution.
3. The size of array can't be changed after its declaration.
4. Arrays are called dense lists and are said to be static data structures.

An elegant solution to these problems is to allow nodes that are dynamic rather than static. That is, when a node is needed in the list, storage is reserved for it, and when it is no longer needed, the storage is released. Thus the memory storage space for nodes that are no longer in use is available for another purpose in the program. Also, no predefined limit on the number of nodes is established. That means we can expand and shrink the list as and when required.

Dynamic Implementation of Linked Lists

A linked list is a linear data structure to store data elements, called nodes in the memory. The linear order is given by means of pointers. That is, each node must contain two parts: the first part contains the information of the element, which may be a single integer, a character, a string or even a large record. The second part, called pointers or links, contains the address of the next node in the list used to access the next node. A node of a linked list at least required the following representation.

Info	Next

The use of pointers or links to the lists implies that elements that are logically adjacent need not be physically adjacent in the memory. This type of allocation is called linked allocation.

A list has been defined to consist of an ordered set of elements which may increased or decreased as and when required. A simple way to represent a linear list is to expand each node to contain a link or pointer to the next node. This representation is called one–way chain or singly linked linear list as illustrated in Fig. 4.4.

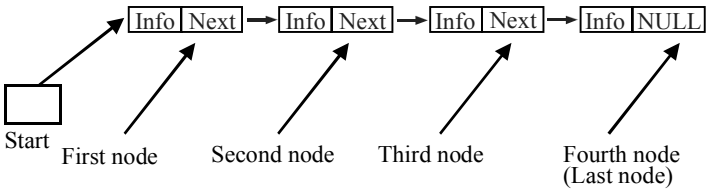


Figure 4.4 *Linear linked list having four nodes*

The linked list is accessed from an external pointer variable, say start, which points (contains the address of) the first node in the list. Hence there is an arrow drawn from start to the first node. This is similar to the array lower bound index (i.e., 0), which indicate the address of the first element of the array.

The last node of list contains a special value in the next address field known as null, which is not a valid address, is called null pointer and indicates the end of the list. This is similar to the array upper bound index, which indicates the address of the last element in the array.

The list with no nodes is called empty list or the null list and is denoted by the null pointer in the list pointer variable start.

[Before to discuss singly linked list, it is utmost requirement to understand dynamically memory allocation and release]

We used pointer variables to implement list pointer. The information part of the node may be a single data item or may consist of multiple data items (e.g., record). Thus the node of a linked list must contain two fields: one for information part and one for next pointer. The best way to declare a linked list in 'C' is by using pointers to structures. Since a linked list is a collection of the nodes (structure) of the same type and hence the node of the linked list can be defined as:

```
struct tag name
{
    data_type   information;   /* information field of the node */
    struct tag name *next;     /* next pointer field of the node */
};
```

The information part of the node may contain any number of data items (fields). The general declaration of a node is:

```
struct tag name
{
    data_type–1   data item–1;   /* information field–1 of the node */
    data_type–2   data item–2;   /* information field–2 of the node */
                …
                …
    data_type–n   data item–n;   /* information field–n of the node */
    struct tag name *next;     /* next pointer field of the node */
};
```

How can a linked list be created? A linked list of student code number having its information as integer data type can be declared as:

```
struct nodetype
{
    int code;
    struct nodetype *next;
};
struct nodetype  *start, *node;
```

Note that next is declared as a pointer to the structure nodetype itself. This means that next points to a location in memory which is of the struct nodetype. This is because next contains the address of the next node in the linked list, and start node is pointer to type of struct nodetype. This type of structure is called self–referential structures.

A node of this type is identical to the nodes of the array implementation except the next field is pointer (containing the address of the next node in the list) rather than an integer (containing the index within an array where the next node in the list is kept).

Now apply the dynamic allocation features to implement linked lists. Instead of declaring an array to represent the maximum number of nodes, nodes are allocated and freed as necessary.

In dynamic memory allocation we use a pointer variable and its size is allocated during the execution of the program. In 'C' malloc() function is used to allocate required memory to pointer variable. The malloc() function simply finds a free block in memory, based on the memory block size requested and then returns a pointer to it. If we execute the statement,

```
node = (struct nodetype *) malloc(sizeof(struct nodetype))
```

A pointer to the address of the allocated memory block is returned by the malloc() function. Note that sizeof is applied a structure type and return the number of bytes required for entire structure.

The allocated memory by malloc() function is not free automatically. For this purpose 'C' has free() function that frees the allocated space. When there is no longer need of a node in the linked list or we explicitly delete a node in the list then the below statement in 'C' frees the memory occupied by the node to available storage i.e., primary memory.

```
free(node);
```

There is no longer a need for managing available memory storage. The system (the computer and compiler) governs the allocating and freeing of nodes. The overflow condition will be detected during the execution of memory allocation functions and they are system dependent. No overflow occurs as long as sufficient storage is available for the linked list nodes, unlike an array implementation where the number of nodes exceed the maximum size of the array.

The major advantage of dynamic implementation is that a set of nodes is not required to be reserved in advance. Even if a program use different types of lists, no storage is allocated for variables until needed.

Consider a student code list {1011, 2022, 3033, 4044}, having four nodes, first node contains 1011, second node contains 2022, third node contains 3033 and fourth node contains 4044 information. The simplest way was just define them as array of structures:

```
struct nodetype node[4];
```

And then node[0] denote first node, node[1] denotes second node, node[2] denotes third node and node[3] denotes the fourth node of the list. In this case the next pointer contains the address of the next node instead of index value of the array node. But this was the form of static implementation. That is, we must know the size of the linked list at the time of compilation.

In case of dynamic implementation, it is necessary that the storage for a node is allocated at the time of execution, as and when needed. This is done using the memory allocation function malloc(). Thus we have defined node and start as pointers to structure nodetype. In order to access the elements of a structure using a pointer to that structure the –> operator is used. For example, the fields code and next can be accessed in any of the following ways after their declaration.

```
node–>code   or (*node).code    /* return the student code number */
node–>next   or (*node).next    /* return the address of the next node in the list */
```

Example 4. We have created two pointer variables start, node to the struct nodetype and initially they have NULL value. Then allocated memory for the first node and assigned the address to the start pointer variable. Then allocated memory for the subsequent node and arranged in the linear order.

The linked list in Fig. 4.5 represents a four nodes student code list {1011, 2022, 3033, 4044}whose elements are located in memory locations 1452, 1460, 1482, and 1496 respectively. The start pointer variable is assigned the address of the first node i.e., 1452. We again emphasize that the only purpose of the links is to specify which node is next in the linear ordering. Remember that the last node in the list should have its pointer, next set to the value NULL.

Here the memory locations are given in decimal number instead of hexadecimal number.

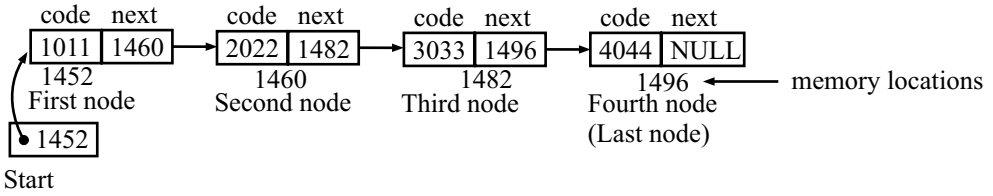


Figure 4.5 *Linear linked list having four nodes*

The memory representation of the above linked list is shown in the Fig. 4.6.

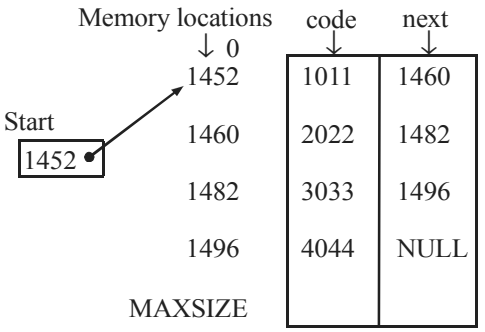
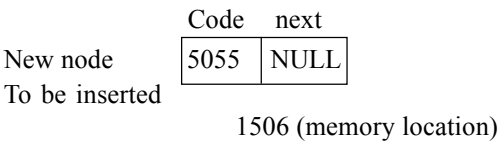
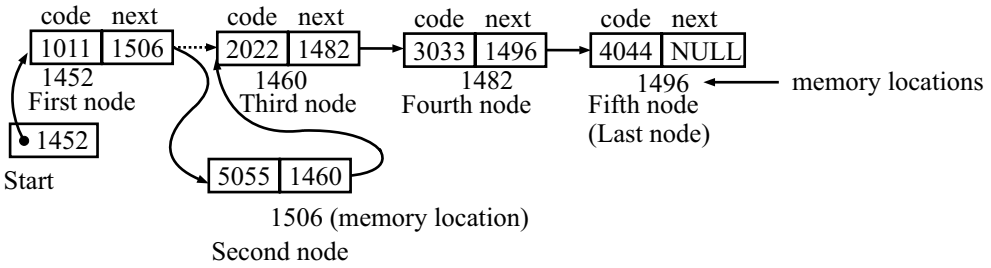


Figure 4.6 *Memory representation of the student code list*

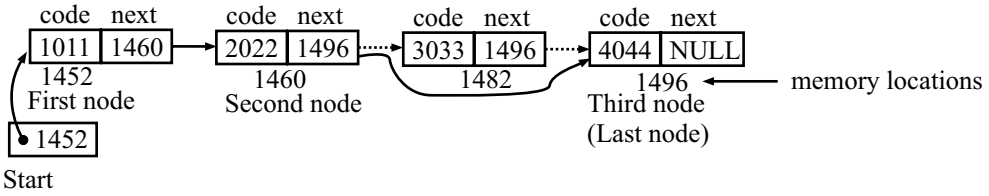
In the linked allocation, an insertion is performed in a straightforward manner. If a new element is to be inserted following the first element, this can be accomplished by merely interchanging pointers. This is depicted in Fig. 4.7, where a new inserted element allocated memory address 1506 and the code value is 5055.



(*a*) A new node to be inserted after node first



(*b*) Insertion is done after first node, thus the logical ordering of the subsequent node is changed



(*c*) Third node deleted

Figure 4.7 *Operations on linear linked list*

Here we have assigned the address in the pointer next of first node to the pointer next of the new node and assigned the address of new node into pointer next of the first node. Thus the arrow direction is made according to pointer position. After insertion, the list would be {1011, 5055, 2022, 3033, 4044} and length of the list would be five.

Using the pointer change as shown in Fig. 4.7 (c) we can perform the deletion of the third element from the original list. Here we have assigned the address in the pointer next of the third node to the pointer next of the second node.

It is clear that the insertion and deletion operations are more efficient when performed on linked lists than on sequentially allocated list i.e., list is stored in an array. But this scheme is inferior when we need to access a particular node of the list. Because it is necessary to follow the links from the first node onward until the desired node is found.

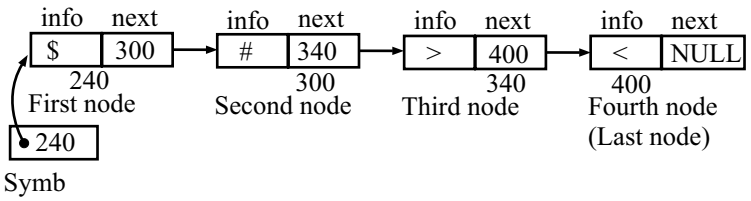
It is easier to join or split two linked lists than it is in case of sequential allocation. This can be accomplished merely by changing pointers and does not require movement of nodes.

The pointers or links consume additional memory, but if only a part of a memory word is being used, then a pointer can be stored in the remaining part. It is possible to group nodes so as to require only one link per several nodes. These two factors make the cost of storing pointers not too expensive in many cases.

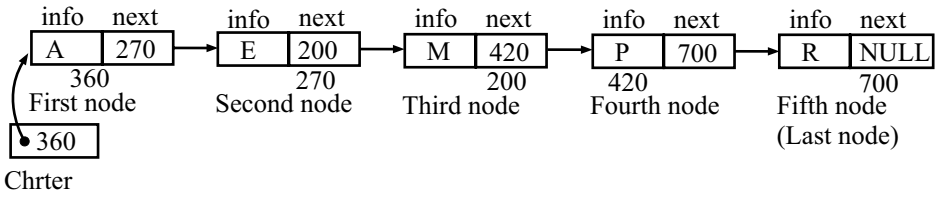
If the application behaviour is dynamic then the linked allocation is more efficient than sequential allocation. If the application behaviour is static then the sequential allocation is preferred. In many applications, both types of allocation are used.

Example 5. Show the memory representation for two-list–symb={ '$', '#', '>', '<'} and chrter = {'A', 'E', 'M', 'P', 'R'}.

The Fig. 4.8 (a) depicts the linked list symb which represents a four-node list whose elements are located in memory locations 240, 300, 340 and 400 respectively. The Fig. 4.8 (b) depicts linked list chrter which represents a five-node list whose elements are located in memory locations 360, 270, 200, 420 and 700, respectively.



(*a*) Linear linked list symb with four nodes



(*b*) Linear linked list chrter with five nodes

Figure 4.8. *Linear linked lists symb and chrter*

Figure 4.9 illustrates the memory representation for the above two-list. Observe that the names of the list are also used as the list pointer variable to represent the first node of the list symb and chrter respectively. The info field is type of character and next pointer field denotes the address of the next node in the list. The memory locations are sequentially given.

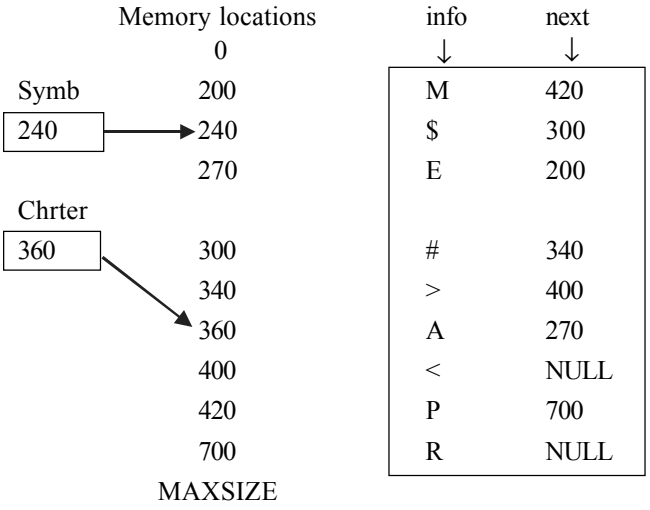


Figure 4.9 *Memory representation of two lists*

Example 6. Suppose that a student address list is stored in a linked list. Each node represents student record, which contains student name, house no., colony, city, state name, and pin code. We can declare the following self-referential structure to create the nodes of the linked list.

```
struct nodetype
{
        char sname[20];
        char hno[10];
        char colony[30];          } information part
        char city[15];
        char state[15];
        int pin;
        struct nodetype *next;
} *start, *node;
```

The start and node is the pointer to struct nodetype. Thus we can create variable number of nodes as and when required.

The Fig. 4.10 illustrates the four nodes of the student address list, whose memory locations are 1800, 1890, 1970 and 2050 respectively. We have assigned some test values to the information part of each node.

Start | 1800 |

D.	Ashish D–32	Patel Nagar	Jaipur	Rajasthan	302011	1890

First node Memory location– 1800

C.	Ashish C–32	Patel Nagar	Jaipur	Rajasthan	302011	1970

Second node Memory location– 1890

B.	Ashish B–32	Patel Nagar	Jaipur	Rajasthan	302011	2050

Third node Memory location– 1970

A.	Ashish A–32	Patel Nagar	Jaipur	Rajasthan	302011	NULL

Fourth node Memory location– 2050

Figure 4.10 *A linear linked list having four nodes*

Traversing a Linked List

Suppose we want to traverse a list in order to process each node exactly once. The traversing algorithm uses a pointer variable ptr which points to the node that is currently being processed. Accordingly, ptr–>next points to the next node to be processed. Thus the assignment moves the pointer to the next node in the list.

```
ptr = ptr–>next
```

The details of the algorithm are as follows:

Initialize pointer ptr to start pointer i.e., the address of the first node of the list. Then process ptr–>info, the information at the first node. Update ptr = ptr–>next, so that ptr points to the second node. Then process the information at the second node. Again update ptr by the assignment ptr = ptr–>next, and then process ptr–>info, the information at the third node, and so on continue until ptr = NULL, which indicates the end of the list.

The algorithm for traversing a linear linked list is given below:

Algorithm TLLIST

[This algorithm traverses a linked list, which is in memory. Applying an operation visit to each element of the list. The pointer variable ptr points to the node currently being processed.]

Step 1 : Set ptr = start; [Initializes pointer ptr to the first node of the linked list]
Step 2 : repeat steps 3 and 4 while ptr <> NULL
Step 3 : Apply operation visit to ptr–>info
Step 4 : Set ptr = ptr–>next; [ptr now points to the next node]
[End of Step 2 loop]
Step 5 : end TLLIST

The algorithm for traversing a linear linked list may be used in several applications/programs. Let us consider a problem to print the element value at each node of a linked list. The procedure for this one may be obtained by simply replacing the step3 of the TLLIST algorithm with the statement–

Write(ptr–>info);

Example 7. Write an algorithm to count the number of node in a linear linked list.

The algorithm to count the number of node in a linked list traverses the linked list in order to count the number of nodes. Here, however, we require an initialization step for a counter variable (e.g. cnt) before traversing algorithm and increment in the counter on traversing each node. Therefore, the algorithm could have been written as follows:

Algorithm COUNTLN

[This algorithm counts node in a linked list, which is in memory. The counter variable cnt is used to store the number of nodes. The pointer variable ptr points to the node currently being processed.]

Step 1 : Set cnt = 0; [initializes counter]
Step 2 : Set ptr = start; [Initializes pointer ptr to the first node of the linked list]
Step 3 : repeat steps 4 and 5 while ptr <> NULL
Step 4 : Set cnt = cnt + 1; [Increases cnt by 1]
Step 5 : Set ptr = ptr–>next; [ptr now points to the next node]
[End of Step 3 loop]
Step 6 : end TLLIST

We can use traversing algorithm in many applications of searching and sorting.

Searching a Linear Linked List

Suppose that a linked list in memory. Suppose a specific data item is given for searching into the list. The searching algorithms find the search item in the list and return the address of the node where the item first appears in the list. There are two searching algorithms: first one is when list is unsorted and second one is when list is sorted.

Linked List is Unsorted

Method

Suppose the data in the linked list are unsorted. We search for an item in the list by traversing the list using a pointer variable ptr and comparing the search item with the contents ptr–>info of each node, one by one of list. If the search item is found then search is successful and returns the location/address of the match node. Otherwise when ptr reaches to end of the list a NULL value is returned, that means search is unsuccessful.

Algorithm LSEARCHL

[The list is unsorted in the memory. This algorithm finds the address/location of the node where a search item (i.e., sItem) first appears in the linked list. If search is unsuccessful then return NULL. A pointer variable loc is used to return the location of the node]

Step 1 : Set ptr = start; [Initializes pointer ptr to the first node of the linked list]

Step 2 : repeat step 3 while ptr <> NULL

Step 3 : if sItem = ptr–>info then
Set loc = ptr; and return; [Search is successful]
else
Set ptr = ptr–>next; [ptr now points to the next node]
[End of if block]
[End of Step2 loop]

Step 4 : Set loc = NULL; [Search is unsuccessful]

Step 5 : return;

The complexity of this algorithm is the same as that of linear search algorithm for linear arrays.

That is, the worse case running time is proportional to the number of elements i.e., n in the list. The average-case running time is approximately proportional to n/2. The search function is given below.

Function

```
void search()
{
struct studinfo *ptr;
        char str[10];
        printf("enter name to be searched");
        scanf("%s",str);
        ptr = start;
      while(ptr! = NULL)
            {
                  if(strcmp(ptr–>name,str) == 0)
                    {
                        printf("\nfound");
                        break;
                    }
                  else
                        ptr = ptr–>next;
            }
       if(ptr == NULL)
              printf("\nnot found");
getch();
}/* end of function */
```

Linked List is Sorted

Suppose the data in the linked list are sorted. Again we search for an item in the list by traversing the list using a pointer variable ptr and comparing the search item with the contents ptr–>info of each node, one

by one of list. If the search item is found then search is successful and returns the location/address of the match node. Otherwise the list traversing is stopped once search item exceeds ptr–>info and NULL value is returned, that means search is unsuccessful.

Algorithm LSEARCHL

[The list is sorted in the memory. This algorithm finds the address/location of the node where a search item (i.e., sItem) first appears in the linked list. If search is unsuccessful then return NULL. A pointer variable loc is used to return the location of the node]

Step 1 : Set ptr = start; [Initializes pointer ptr to the first node of the linked list]
Step 2 : repeat Step 3 while ptr <> NULL
Step 3 : if sItem > ptr–>info then
Set ptr = ptr–>next; [ptr now points to the next node]
else if sItem = ptr–>info then
Set loc = ptr; and return; [Search is successful]
else
Set loc = NULL; and return [sItem now exceeds ptr–>info]
[End of if block]
[End of Step2 loop]
Step 4 : Set loc =NULL; [Search is unsuccessful]
Step 5 : return;

The complexity of this algorithm is still the same as that of other linear search algorithm. That is, the worse case running time is proportional to the number of elements i.e., n in the list. The average–case running time is approximately proportional to n/2. In case of sorted array, we have applied a binary search algorithm and the complexity of this algorithm was proportional to $\log_2 n$. A binary search algorithm cannot be applied to a sorted linked list, since there is no way of indexing the middle element in the list. This property is one of the main drawbacks of using linked list as a data structure.

Inserting into a Linear Linked List

We can insert a new node into a linked list in the following manner:

- Insert as a first node
- Insert as a last node
- Insert in a sorted linked list
- Insert after a given node
- Insert before a given node
- Insert as a middle node

Insertion as a First Node

Method

Let us consider that a linked list of student code list {1011, 2022, 3033, 4044} is stored in the memory. Here the first node of the linked list has information 1011. Let the start pointer holds the address of the first node and pointer ptr denotes the address of the current node. The pointer variable newnode holds the newly inserted node into the linked list. The new node is inserted as the first node, hence the original first node now becomes second node of the list and second node becomes third node of the list and so on.

For example if we insert a new node having information 5055 as the first node, then the list becomes {5055, 1011, 2022, 3033, 4044} as given in Fig. 4.11.

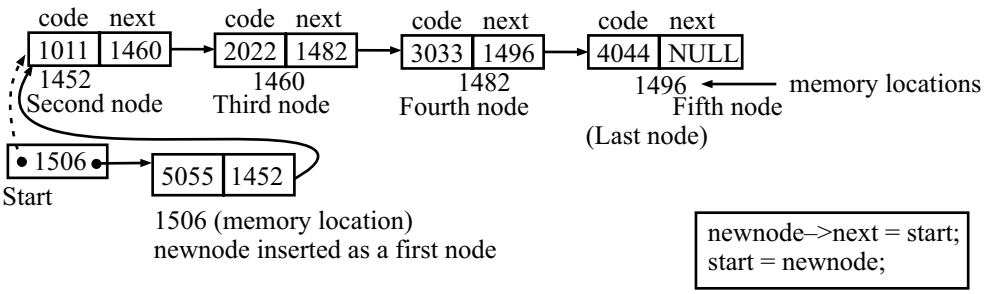


Figure 4.11 *Insertion as a first node in the linear linked list*

Algorithm

The newnode–>next has assigned the address in the start pointer i.e., the address of the original first node (i.e., 1452) and start pointer holds the address of the newnode.

The formal algorithm is given below.

Algorithm INSFIRST

[Let us consider that a linked list is stored in memory. This algorithm inserts a new node as the first node of the linked list. The item information is stored in the new node.]

Step 1 : [allocate memory for a new node]
newnode = allocate free memory using malloc() function

Step 2 : Set newnode–>info = item; [Read value of the information part of a new node]

Step 3 : Set newnode–>next = start; [newnode now points to original first node]

Step 4 : Set start = newnode; [Change pointer start so it points to the new node and newnode becomes the first node of the list]

Step 5 : end INSFIRST

The function insertf inserts a new node as the first node of the linked list.

Function

```
void insertf()
   {
        struct studinfo   *newnode;
        newnode=(struct studinfo *) malloc(sizeof(struct studinfo));
        printf("Enter a new  record : marks and name");
        scanf("%d%s",&newnode–>marks,newnode–>name);
        newnode–>next = NULL;
        if(start == NULL)
                start = newnode;
          else
         {
              newnode–>next = start;
                 start = newnode;
         }
}
```

Insertion as a Last Node

Method

The easiest way of the insertion is to append the new node i.e., added in the last of the list. To append a node into the last of the list, we first traverse the list to reach the last node of the list. The next pointer of the last node holds the NULL value. Now this pointer holds the address of the new inserted node and the next pointer of the new node holds the NULL value.

This process is best explained by the following Fig. 4.12.

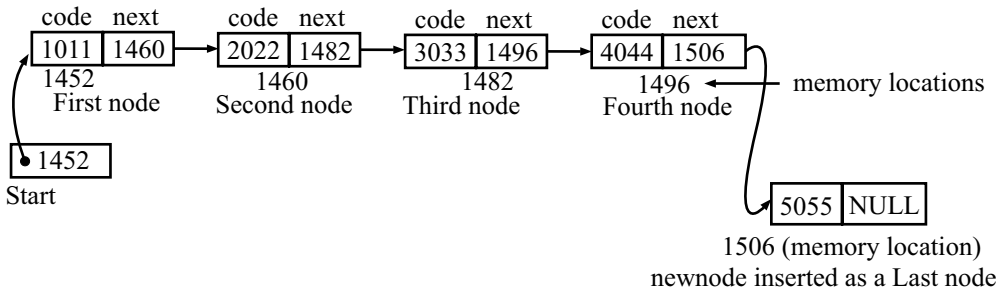


Figure 4.12 *Insertion as a last node in the linear linked list*

Algorithm

Initialize pointer ptr to start pointer i.e., the address of the first node of the list. Update ptr=ptr–>next, so that ptr points to the second node. Again update ptr by the assignment ptr = ptr–>next, and so on. Continue until ptr–>next = NULL, which indicates the end of the list.

Then assign the address of new node into ptr–>next and next pointer of new node assign a NULL value to denote end of the list i.e., the last node of the list.

The formal algorithm is given below:

Algorithm INSLAST

[Let us consider that a linked list is stored in memory. This algorithm inserts a new node as the last node of the linked list. The item information is stored in the new node. If list is empty then new node is inserted as the first node of the list]

Step 1 : [allocate memory for a new node]
newnode = allocate free memory using malloc() function

Step 2 : Set newnode–>info = item; [Read value of the information part of a new node]
Set newnode–>next = NULL; [newnode becomes the last node of the list]

Step 3 : if start=NULL then [if list is empty then insert as a first node]
Set start = newnode;
else
Set ptr = start; [Initializes pointer ptr to the first node of the linked list]

Step 4 : repeat step 5 while ptr–>next <> NULL

Step 5 : Set ptr = ptr–>next; [ptr now points to the next node]
[End of Step 4 loop and now ptr points to the last node of the list]

Step 6 : Set ptr–>next = newnode; [ptr now points to the new node]
[End of the Else structure]

Step 7 : end INSLAST

Function

```
void insertl()
{
struct studinfo *newnode, *ptr;
newnode = (struct studinfo *) malloc (sizeof (struct studinfo));
printf("Enter a new  record : marks and name");
scanf("%d%s",&newnode–>marks,newnode–>name);
newnode–>next = NULL;
if (start == NULL)
      start = newnode;
else
      {
      ptr = start;
     while (ptr–>next ! = NULL)
                        ptr = ptr–>next;
     ptr–>next = newnode;
          }
}
```

Insertion into a Sorted Linked List

Method

To create a linked list that contains the sorted data items in ascending order. To insert an item at a specific location, the algorithm will normally track the starting node, current node and the node just previous to the current node. Here we traverse the list from the start node to the desired node where we want to insert the new node.

Suppose that a node is to be inserted into a sorted linked list. The information part of the inserted node is item. Then item must be inserted between nodes A and B so that

A–>info < item < B–>info

The following is a procedure, which finds the location/address loc of the node A.

Traverse the list, using a pointer variable ptr and comparing item with ptr–>info at each node. While traversing, keep track of the address of the preceding node by using a pointer variable prevptr. Thus prevptr and ptr are updated by the assignments

prevptr = ptr; and ptr = ptr–>next;

The traversing continues as long as ptr–>info > item, or the traversing stops as soon as item ≤ ptr–>info.

Then ptr points to node B, so prevptr will contain the address of the node A.

Here we have given two algorithms: the algorithm FINDLOC finds the address loc of the last node in a sorted list such that loc–>info < item, or sets loc =NULL. The cases where the list is empty or where item < start–>info, so that loc = NULL is set. This algorithm is written as the sub-algorithm in the form of function. Then the function FINDLOC returns the loc pointer.

The second algorithm INSSL calls the algorithm FINDLOC to return the loc pointer. If loc pointer is NULL then insert the node as first node otherwise insert after node with address holds in loc pointer.

The algorithm FINDLOC is written in the form of function.

Algorithm FINDLOC(start, item)

[This algorithm returns the address loc of the last node in a sorted list such that loc–>info < item, or sets loc =NULL]

Step 1 : if start = NULL then
Set loc = NULL; and return(loc); [List is empty]

Step 2 : if item < start–>info then
Set loc = NULL; and return(loc); [the item is smaller than the first node item, thus must be inserted as the first node of the list]

Step 3 : Set prevptr = start; and ptr = start–>next; [Initialize pointers]

Step 4 : repeat Steps 5 and 6 while ptr<> NULL

Step 5 : if item < ptr–>info then
Set loc = prevptr; and return(loc);

Step 6 : Set prevptr = ptr; and ptr = ptr–>next; [Updates pointers]

Step 7 : Set loc = prevptr;

Step 8 : return(loc);

The algorithm INSSL calls the FINDLOC as follows:

Algorithm INSSL

[This algorithm inserts item into a sorted linked list.]

Step 1 : loc = FINDLOC(start, item); [return the address of the node preceding item]

Step 2 : newnode = Allocate memory using malloc() function

Step 3 : Set newnode–>info = item; [Assigned value of the information part of a new node]

Step 4 : if loc = NULL then [Insert as first node]
Set newnode–>next = start; and start = newnode;
else [Insert after node with address loc]
Set newnode–>next = loc–>next; and loc–>next = newnode;
[End of If statement]

Step 5 : end INSSL

The algorithm considers the case of a null or empty list first of all. That is when location loc is NULL then insert the new node as the first node of the list and return the new node address. It also considers the case when new node is preceding the first node in the original list. The new node becomes the first node in the updated list. The next pointer field of the new node is assigned a pointer value which corresponds to the location of the first node of the original list. In other cases, the next pointer field of the new node is assigned the address of the node indicated by next pointer of the location loc and the next pointer field of the loc is assigned a pointer value of new node.

The average time for such an insertion is O(n/2).

A complete 'C' program which inserts element into a sorted linked list is given below:

```
/* Insnllst.cpp program to insert node in sorted linked list as insertion sort */
#include<stdio.h>
#include<malloc.h>
struct nodetype{
    int code;
struct nodetype *next;
```

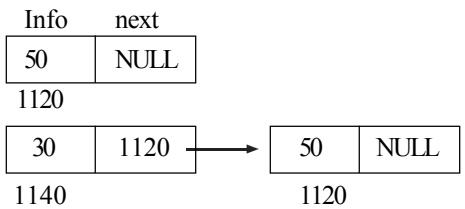
```
}*ptr,*start,*newnode;
int item = 0;
struct nodetype *insertion(struct nodetype *, int);
struct nodetype *findloc(struct nodetype *, int);
void traverse(struct nodetype *);
void main()
{
start = NULL; /* Initially list is empty */
printf("Enter the student code number and type –1 for termination\n");
while(item! = –1)
{
     scanf("%d",&item);
     if (item ! = –1)
               start = insertion(start,item);
}
printf("\n Linked list nodes are : \n");
traverse(start);
getch( );
}/* end of main */

/* Insertion function insert node as sorted order using INSSL Algorithm */
struct nodetype *insertion(struct nodetype *start, int insitem)
{
     struct nodetype *loc;
     newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
     newnode–>code = insitem;
     loc = findloc(start, insitem);
     if (loc == NULL)
     {
     newnode–>next = start;
     start = newnode;
     }
     else
     {
     newnode–>next = loc–>next;
     loc–>next = newnode;
     }
 return (start);
}/* end of the function*/

/* function findloc returns the location of the previous element where the element is inserted */
struct nodetype *findloc(struct nodetype *start, int insitem)
```

```
{
    struct nodetype *prevptr, *ptr;
    if (start == NULL)
      return (NULL);
    if (insitem<start–>code)
            return (NULL);
    prevptr = start;
    ptr     = start–>next;
    while (ptr!= NULL)
    {
    if (insitem < ptr–>code)
            return (prevptr);
    prevptr = ptr;
    ptr = ptr–>next;
    }
 return (prevptr);
}

void traverse(struct nodetype *start)
{
    ptr = start;
    while(ptr! = NULL)
    {
    printf("\t%d", ptr–>code);
    ptr = ptr–>next;
    }
}  /* end of function */
```

We can easily obtain an ordered linear linked list. For example, the sequence of statements

Start = NULL
Start = INSSL(50, start)
Start = INSSL(30, start)
Start = INSSL(40, start)
Start = INSSL(35, start)
Start = INSSL(45, start)

creates a five-element list. A trace of this creation is given in Fig. 4.13.



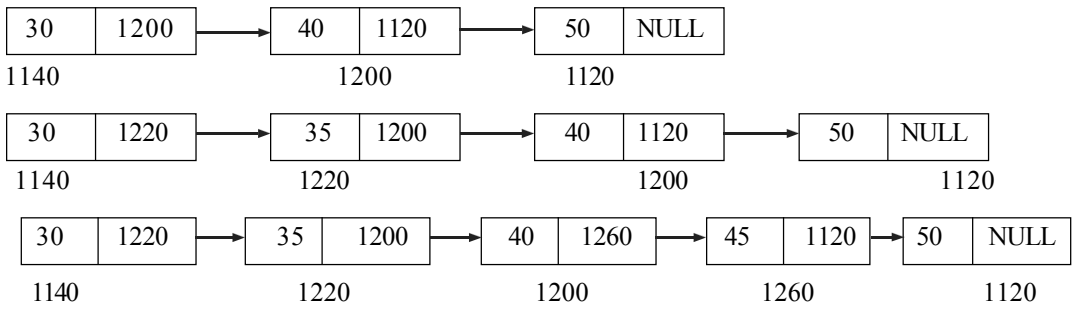


Figure 4.13 *Creation of the ordered linear linked list using Algorithm INSSL*

Insertion of a Node After a Given Node

To insert an item after a given node information, the algorithm will normally track the starting node, current node and the node just previous to the current node. Here we traverse the list from the start node to the desired node where we want to insert the new node.

Let us consider that the linked list is unsorted. The algorithm simply searches the location of the information of the node and inserts the node into proper place.

Insertion of Node as a Node Number

To insert a node as the node number in the linked list, the algorithm traverses the list from the start node to the desired node number where we want to insert the new node.

Algorithm INSNNLST

[This algorithm inserts a node as the node number of the linked list. The variable cnt counts the node number in the list and node number is given as nodenum.]

Step 1 : Set cnt = 0; ptr = start; [Initialize the variables]

Step 2 : if (nodenum = 1) then
newnode–>next = start; start = newnode;
return;

Step 3 : repeat steps 4 and 5 while ptr ! = NULL

Step 4 : if (cnt +1) = nodenum then
Set newnode–>next = ptr–>next; and ptr–>next = newnode;
else

Step 5 : Set ptr = ptr–>next; and cnt = cnt +1;
[End of Step3 loop]

Step 6 : end INSNNLST

The function to insert as node number is given below:

```
void insertion(int nodenum)
{
    struct studinfo *newnode, *loc;
    newnode = (struct studinfo *)malloc(sizeof(struct studinfo));
    printf("Enter a new record : marks and name");
    scanf("%d%s",&newnode–>marks,newnode–>name);
```

```
        loc = findloc(nodenum);
        if (loc == NULL)
        {
        newnode–>next = start;
        start = newnode;
        }
        else
        {
        newnode–>next = loc–>next;
        loc–>next = newnode;
        }
}/* end of the function*/
```

```
struct studinfo *findloc(int nodenum)
{
struct studinfo *prevptr, *ptr;
int cnt = 0;
if (nodenum == 1)
  return (NULL);
prevptr = start;
ptr  = start–>next;
while (ptr! = NULL)
{
if ((cnt+1) == nodenum–1)
      return (prevptr);
else
      {
      prevptr = ptr;
      ptr = ptr–>next;
      cnt = cnt +1;
      }
}
return (prevptr);
}/* end of function */
```

Deletion from a Linear Linked List

Let a linked list in the memory. The algorithms which delete nodes from linked lists come up in various situations. A general deletion algorithm for deleting a node from a linked list is as follows:

1. If the linked list is empty, the deletion is not possible and then the algorithm will print the message UNDERFLOW.
2. Consider the below step3 while end of the list has not been reached and the node has not been found.
3. Keep track of the next node and its predecessor in the list.

4. If the criteria node is found then delete the node from the list.
5. Otherwise "Deletion can't be performed".
6. And in the last-free memory, the memory occupied by the deleted node.

A general deletion criterion is discussed below.

A linked list with a node N between nodes A and B, given in Fig. 4.13(a) Suppose node N is to be deleted from the linked list. The schematic diagram of such a deletion appears in Fig. 4.13(a).

To delete the node N, we have changed the nextpointer field of node A, so that it points to node B.

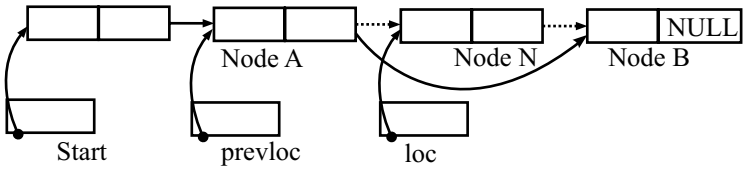


Figure 4.13(a) *Deletion of node N [prevloc–>next = loc–>next]*

There are several cases where we need deletion. Some of the cases are:

- Deletion of the first node
- Deletion of the last node
- Deleting the node with a given item of information
- Deletion of the node as a node number

Deletion of the First Node

Method

Suppose we have a linked list in memory and want to delete first node. The Fig. 4.14 is the schematic diagram of the assignment

start = start–>next;

which effectively deletes the first node from the list.

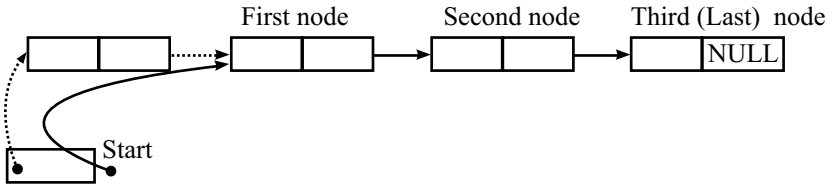


Figure 4.14 *Deletion of first node [start = start–>next]*

The deletion algorithm is given below in the form of function.

Algorithm DELFNLST

[This algorithm deletes the first node of a linked list.]

Step 1 : if start = NULL then
write("UNDERFLOW"); and return [Linked List is empty]
else

Step 2 : Set ptr = start; Set start = start–>next; [Delete the first node]

Step 3 : free(ptr); [Free the memory occupied by the deleted node]

Step 4 : return (start); [return the new address of the start pointer]

Function

```
/* Function deletef deletes node as first node in the linked list */
void deletef()
{
   struct studinfo *ptr;
   ptr = start;
   start = start–>next;
   free(ptr);
} /* end of function*/
```

Deletion of the Last Node

Method

The Figure 4.15 is the schematic diagram of the assignment

```
prevloc–>next = NULL;
```

which effectively deletes the last node from the list.

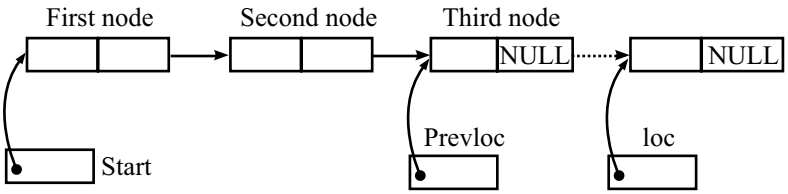


Figure 4.15 *Deletion of last node* *prevloc–>next = NULL; free(loc);*

The deletion algorithm is given below in the form of function.

Algorithm DELLNLST

[This algorithm deletes the last node of a linked list.]

Step 1 : if start = NULL then
write("UNDERFLOW"); and return(NULL) [Linked List is empty]

Step 2 : Set ptr = start ; [Initializes pointer ptr to start of the list pointer]

Step 3 : repeat step 4 while ptr–>next ! = NULL

Step 4 : Set prevptr = ptr; and Set ptr = ptr–>next; [Updates pointer ptr and its predecessor pointer prevptr]

Step 5 : if (start–>next = NULL) then
start = NULL; [List has only first node, thus after deleting the node, the updated list becomes empty]

else
prevptr–>next = NULL; [Sets the preceding node of the last node as a last node in the updated linked list]

Step 6 : free(ptr); [Frees the memory occupied by the deleted node]

Step 7 : return (start);

Function

```
/* Function deletel, deletes node as last node in the linked list */
void deletel()
 {
        struct studinfo *ptr,*prevptr;
        ptr = start;
       while(ptr–>next! = NULL)
         {
              prevptr = ptr;
              ptr = ptr–>next;
         }
        prevptr–>next = NULL;
         free(ptr);
}/* end of function */
```

Deleting the Node with a Given Item of Information

Method

Let us consider that a linked list is in memory. We want to delete a node N with a given item of information from the list. We first find the address/location ptr of the first occurrence of the node N with item information and the location prevptr of the node preceding node N. If N is the first node, we delete the first node of the list and update the start pointer and return. If item does not appear in the list, that means search of the item is unsuccessful, then deletion can't perform.

Traverse the list, using a pointer variable ptr and comparing item with ptr–>info at each node. While traversing, keep track of the location of the preceding node by using a pointer variable prevptr as shown in Fig. 4.13. Thus prevptr and ptr are updated by the assignments

prevptr = ptr; and ptr = ptr–>next;

The traversing continues until ptr–>info != item and ptr != NULL. In other words, the traversing stops when the first occurrence of ptr–>info = item is come. Then ptr contains the location of the node N and prevptr contains the location of the node preceding N.

The below deletion algorithm DELNILST deletes the desire node with the given information in the variable delitem.

Algorithm DELNILST

[This algorithm deletes a node with given information]

Step 1 : if start = NULL then
write("UNDERFLOW"); and return [Linked List is empty]

Step 2 : Set ptr = start; [Initialize pointer]

Step 3 : repeat steps 3 and 4 while ptr–>next ! = NULL and ptr–>info ! = delitem

Step 4 : set prevptr = ptr; and ptr = ptr–>next;

Step 5 : [End of the list?]
if ptr–>info ! = delitem then
write("Node not found"); and return;

Step 6 : [Delete the node]

```
            if (start–>info = delitem) then
                    start = start–>next
            else
                    prevptr–>next = ptr–>next;
```

Step 7 : free(ptr);

Step 8 : return (start);

This algorithm first checks for an underflow. Next, a search is made for the desired node. Initialize pointer ptr to the start pointer of the linked list. Note that the repeat condition in Step 3 fails either when delitem is found or when the end of the list is reached, thus in Step 5, ptr–>info != delitem indicates the end of the list was reached without delitem being found. Step 6 performs the deletion. If delitem is the first node, the second node becomes the new first node in the updated list. This step also handles the situaton in which there is only one node in the list in which case its deletion results in start being set to NULL i.e., start = start–>next, and start–>next is NULL if single node in the list. When delitem is not the first node, its predecessor is set to ptr–>next. Finally ptr frees the allocated memory storage to main memory.

Deletion of the Node as a Node Number

Function:

```
/* Function deleteno, delete node by node number in the linked list */
void deleteno()
  {
      int cnt = 1,no;
      struct studinfo *ptr,*prevptr, *newnode;
     printf("\enter number ...");
      scanf("%d",&no);
    if(start–>next == NULL)
      {
              deletef();
      }
    else
     {
    ptr = start;

    while(cnt!=no)
              {
                      prevptr = ptr;
                      ptr = ptr–>next;
                      cnt++;
              }
    if (ptr–>next == NULL)
              deletel();
      prevptr–>next = ptr–>next;
      free(ptr);
    }
}/* end of function */
```

A complete 'C' program for single linked list is given below.


```
/* program singlell.cpp to perform data structure operations on linear linked list.
The node consists of student record: marks and name */
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct  studinfo{
        int marks;
       char name[10];
        struct studinfo *next;
}*start;
void createlist();
void traverse();
void search();
void insertf();
void insertl();
void deletef();
void deletel();
void inserta();
void insertb();
void deleteno();
void insertion(int);
struct studinfo *findloc(int);

void main()
{
        int c,nn;
                clrscr();
        start = NULL;   /*Initially linked List is empty */

        do {
        printf("\n Enter choice for Link List ....");
        printf("\n 1. Create Link list....");
        printf("\n 2. Insert node as First node.. ");
        printf("\n 3. Delete First node ..");
        printf("\n 4. Delete Last node ..");
        printf("\n 5. Seacrh a node ..");
        printf("\n 6. Insert after node no...");
        printf("\n 7. insert before node no");
        printf("\n 8. Insert node as Last node");
        printf("\n 9. Insert node as node number");
        printf("\n 10. Delete node(given no)");
        printf("\n 11.Traverse List..");
```

```
        printf("\n 12.Exit Lists..");
        printf("\n Enter choice for Link List ....");
        scanf("%d",&c);
        switch (c)
        {
        case 1:
                createlist();
                break;
        case 2:
                insertf();
                break;
        case 3:
                deletef();
                break;
        case 4:
                deletel();
                break;
        case 5:
                search();
                break;
        case 6:
                inserta();
                break;
        case 7:
                insertb();
                break;
        case 8:
                insertl();
                break;
        case 9:
        printf ("\n Enter node number...");
                scanf("%d",&nn);
                insertion(nn);
                break;
        case 10:
                deleteno();
                break;
        case 11:
                traverse();
                break;
        }
   }while (c!=12);
} /* end of main */
```

```
/* function creates the linear list until choice is no */
void createlist()
{
struct studinfo *newnode, *ptr;
char ch;
do{
        newnode=(struct studinfo *)malloc(sizeof(struct studinfo));
       printf("Enter a new  record : marks and name");
       scanf("%d%s",&newnode–>marks,newnode–>name);
      newnode–>next = NULL;
      if (start == NULL)
              start = newnode;
       else
       {
              ptr = start;
              while (ptr–>next ! = NULL)
                      ptr = ptr–>next;
              ptr–>next = newnode;
       }
    fflush(stdin);
    printf("do you want to continue (y/n) ");
    scanf("%c",&ch);
 }while( ch =='y' || ch =='Y');
} /* end of function */

/* function searches the node in the list in sequential manner */
void search()
{
struct studinfo *ptr;
    char str[10];
    printf("enter name to be searched ");
    scanf("%s",str);
    ptr = start;
    while(ptr! = NULL)
       {
              if(strcmp(ptr–>name,str) == 0)
               {
                      printf("\nfound");
                      break;
               }
              else
                      ptr = ptr–>next;
       }
```

```
    if(ptr == NULL)
            printf("\nnot found");
getch();
}/* end of function */
/* Function insertf, inserts node as first node in the linked list */
void insertf()
 {
    struct studinfo   *newnode;
    newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
    printf("Enter a new  record : marks and name");
    scanf("%d%s",&newnode–>marks,newnode–>name);
    newnode–>next = NULL;
    if(start == NULL)
                    start = newnode;
          else
     {
           newnode–>next = start;
            start = newnode;
     }
}/* end of function */
/* Function insertl, inserts node as last node in the linked list */
void insertl()
{
struct studinfo *newnode, *ptr;
newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
printf("Enter a new  record : marks and name");
scanf("%d%s",&newnode–>marks,newnode–>name);
newnode–>next = NULL;
if (start == NULL)
          start = newnode;
else
    {
    ptr = start;
    while (ptr–>next ! = NULL)
                    ptr = ptr–>next;
    ptr–>next = newnode;
     }
}/* end of function */
/* Function deletef, deletes node as first node in the linked list */
void deletef()
{
    struct studinfo *ptr;
    if (start == NULL)
    {printf ("/n list is empty");
    return;
```

```
}
ptr = start;
      start = start–>next;
      free(ptr);
} /* end of function*/
/* Function deletel, deletes node as last node in the linked list */
void deletel()
  {
          struct studinfo *ptr,*prevptr;
          ptr = start;
      if (ptr is next == NULL)
start = NULL;
else
{
}
        while(ptr–>next! = NULL)
           {
                    prevptr = ptr;
                    ptr = ptr–>next;
           }
          prevptr–>next = NULL;
           }
           free(ptr);
}/* end of function */
/* Function inserta, inserts node after a given node in the linked list */
void inserta()
{
   int cnt = 1, no;
          struct studinfo *ptr,*prevptr, *newnode;
            printf("\n enter number ...");
            scanf("%d",&no);
            ptr = start;
            while (cnt ! = no)
            {
            ptr = ptr–>next;
            cnt++;
            }
          newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
          printf("\n Enter a new  record : marks and name");
          scanf("%d%s",&newnode–>marks,newnode–>name);
          newnode–>next = NULL;
            newnode–>next = ptr–>next;
            ptr–>next = newnode;
}/* end of function */
/* Function insertb, inserts node before a given node in the linked list */
void insertb()
 {
```

```
    int cnt = 1, no;
    struct studinfo *ptr,*prevptr, *newnode;
      printf("\n enter number ...");
      scanf("%d",&no);
      ptr = start;
    if(no == 1)
     {
          insertf();
     }
     else
     {
          while(cnt ! = no)
          {
               prevptr = ptr;
               ptr = ptr–>next;
               cnt++;
          }
     newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
     printf("\n Enter a new  record : marks and name");
     scanf("%d%s",&newnode–>marks,newnode–>name);
     newnode–>next = ptr;
     prevptr–>next = newnode;
     }
}/* end of function */

void insertion(int nodenum)
{
    struct studinfo *newnode, *loc;
    newnode = (struct studinfo *)malloc(sizeof(struct studinfo));
    printf("\n Enter a new  record : marks and name");
    scanf("%d%s",&newnode–>marks,newnode–>name);
    loc = findloc(nodenum);
    if (loc == NULL)
    {
    newnode–>next = start;
    start = newnode;
    }
    else
    {
    newnode–>next = loc–>next;
    loc–>next = newnode;
    }
}/* end of the function*/
```

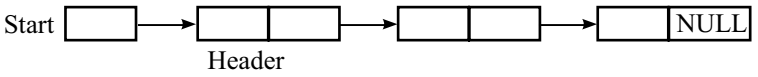
```
struct studinfo *findloc(int nodenum)
{
struct studinfo *prevptr, *ptr;
int cnt = 0;
if (nodenum == 1)
  return (NULL);
prevptr = start;
ptr  = start–>next;
while (ptr! = NULL)
{
if ((cnt+1) == nodenum–1)
      return (prevptr);
else
      {
      prevptr = ptr;
      ptr = ptr–>next;
      cnt = cnt +1;
      }
}
return (prevptr);
}/* end of function */

/* Function deleteno, deletes node by node number in the linked list */
void deleteno()
  {
       int cnt = 1,no;
       struct studinfo *ptr,*prevptr, *newnode;
       printf("\n enter number ...");
       scanf("%d",&no);
      if(no==1)
       {
                deletef();
       }
      else
      {
      ptr = start;

      while(cnt! = no)
                 {
                          prevptr = ptr;
                          ptr = ptr–>next;
                          cnt++;
                 }
      if (ptr–>next == NULL)
                 deletel();

        prevptr–>next = ptr–>next;
        free(ptr);
```

```
    }
}/* end of function */

/* function traveres each node of the linked list */
void traverse()
{
struct studinfo *ptr;
ptr = start;
if (start== NULL)
{
printf ("\n list is empty");
return;
}
while (ptr ! = NULL)
{
printf("\nRecord: marks and name %d  %s\n",ptr–>marks, ptr–>name);
ptr = ptr–>next;
}
getch( );
}
```

Header Linked List

A header linked list is a linked list which always contains a special node, called the header node, at the beginning of the list. The following are two kinds of widely used header lists.

1. A grounded header list is a header list where the last node contains null pointer (see below).
2. A circular header list is a header list where the last node points back to the header node (see Fig. 4.16).



4.2 CIRCULAR LINKED LINEAR LISTS AND OPERATIONS

If we replace the null pointer in the last node of a list with address of its first node, such a list is called a circularly linked linear list or simply a circular list. Figure 4.16 illustrates the schematic diagram of a circular list.

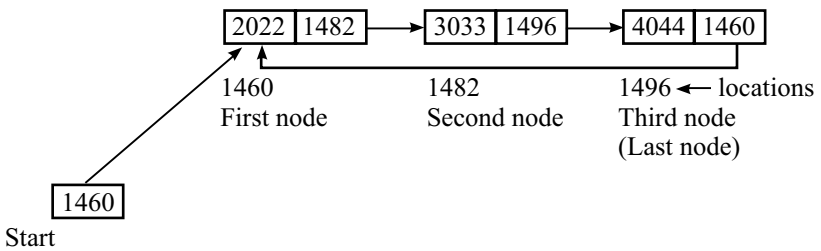


Figure 4.16 *Circular linear linked list*

Circular lists have certain advantages over singly linked lists. The first of these is concerned with the accessibility of a node. In a circular list every node is accessible from a given node. That is, from this given node, all nodes can be reached by merely chaining through the list.

A second advantage concerns the deletion operation. In the deletion algorithm of linear linked list, to find the predecessor requires that a search be carried out by chaining through the nodes from the first node of the list. But this requirement does not exist for a circular list, since the search for the predecessor of node N can be initiated from X itself.

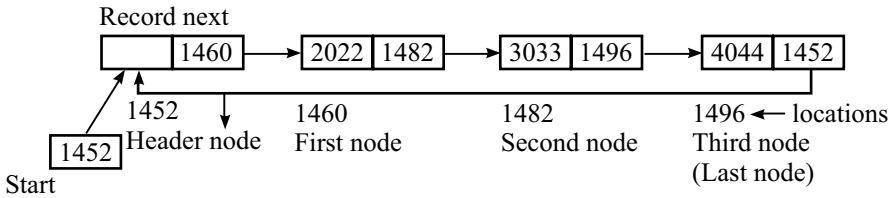
Finally, certain operations on circular lists, such as concatenation and splitting, become more efficient.

The circular list required extra care to detect the end of the list. It may be possible to get into an infinite loop. The detection of the end by placing a special node which can be easily identified in the circular list. This special node is often called the header of the circular list. This technique has an advantage of its own – the list can never be empty.

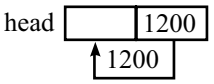
Unless otherwise stated or implied, our header lists will always be circular. Accordingly, the header node also indicates the end of the list.

The representation of a circular list with a header node is given in Fig. 4.17. The pointer variable head denotes the address of the header node (i.e., list head). Note that the information part in the list head node is not used, which is illustrated by shading the field. An empty circular list is represented by having head–>next = head.

Observe that the list pointer start always points to the header node. The head–>next = head indicates that a circular header list is empty.



(*a*) A circular linked linear list with header node



(*b*) Empty circular list with header node

Figure 4.17 *Circular linear linked list with header node*

The first node in a header list is the node following the header node, and the location/address of the first node is start–>next, not start, as with ordinary linked lists.

Circular header lists are mostly used instead of ordinary list because of the following properties:

1. The null pointer is not used, and hence all pointers contain valid addresses.
2. Every (except header) node has a predecessor, so the first node may not require a special case.

Now algorithms for insertion and deletion of a node become simple. Some of the algorithms are discussed here.

1. Traversing a circular header linked list

Let us consider algorithm for traversing a circular header linked list. This algorithm is same as the algorithm for traversing the ordinary linear linked list except that now the algorithm.

(*a*) Initialize pointer ptr with ptr =start–>next not ptr = start in Step 1.

(*b*) The while loop fails when ptr = start not ptr = NULL in Step 2.

The algorithm TRCHLLST for traversing the circular header linked list.

Algorithm TRCHLLST

[This algorithm traverses a circular header linked list, which is in memory. Applying an operation visit to each element of the list. The pointer variable ptr points to the node currently being processed.]

Step 1 : Set ptr = start–>next; [Initializes pointer ptr to the first node of the linked list]
Step 2 : repeat Steps 3 and 4 while ptr ! = start
Step 3 : Apply operation visit to ptr–>info
Step 4 : Set ptr = ptr–>next; [ptr now points to the next node]
[End of Step 2 loop]
Step 5 : end TRCHLLST

2. Searching a given item of information in the circular header linked list

The following is the algorithm LSRCHLST for finding the location of the first occurrence of search item in the circular header linked list.

Algorithm LSRCHLST

[The circular header linked list is unsorted in the memory. This algorithm finds the address/ location of the node where a search item (i.e., sItem) first appears in the linked list. If search is unsuccessful then return NULL. A pointer variable loc is used to return the location of the node.]

Step 1 : Set ptr = start–>next; [Initializes pointer ptr to the first node of the linked list]
Step 2 : repeat Step 3 while ptr–>info != sItem and ptr != start
Step 3 : Set ptr = ptr–>next; [ptr now points to the next node]
[End of Step 2 loop]
Step 4 : if sItem = ptr–>info then
Set loc = ptr; and return; [Search is successful]
else
Step 5 : Set loc =NULL; [Search is unsuccessful]
[End of if block]
Step 6 : return;

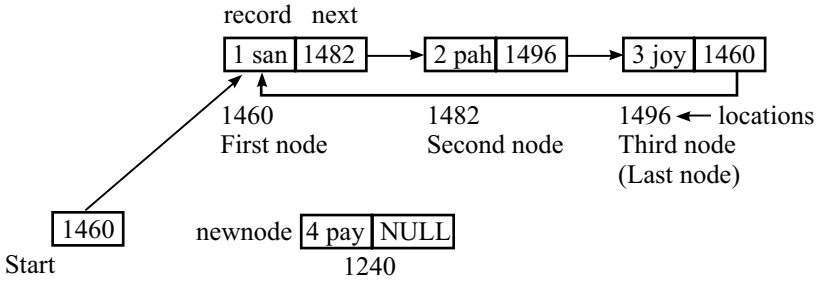
The loop in Step 2 tests two conditions, whereas in the ordinary linked list search algorithm it is one. Because for the ordinary linked lists ptr–>info is not defined when ptr = NULL. This makes it simple.

Insertion Operation

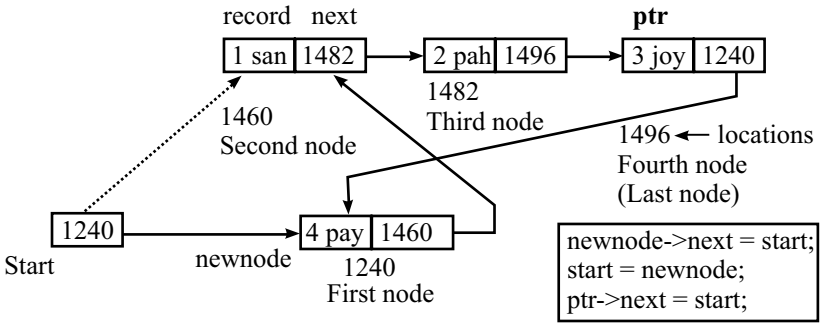
Consider the problem of inserting a node into a circular linear linked list. There are a number of cases which are possible.

1. Inserting a node when list is empty
2. Inserting a node after the right–most node of the list (i.e., as last node)
3. Inserting a node before the left–most node of the list (i.e., as first node)
4. Inserting a node to the left of a given node (i.e., before a node)
5. Inserting a node to the right of a given node (i.e., after a node)
6. Inserting a node as a node number of the list

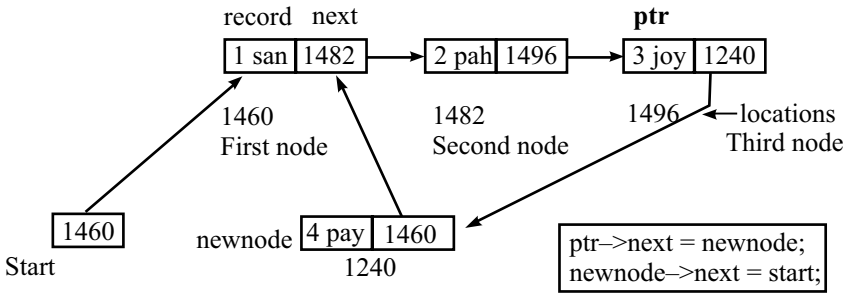
The insertion operation is explained in the below Fig. 4.18



(*a*) Circular single-linked list before a newnode insertion



(*b*) Circular single linked list after a newnode is inserted as first node



Fourth node (last node)

(*c*) Circular single linked list after a newnode is inserted as last node

Figure 4.18 *Insertion operation on circular single linked list*

The remaining insertion operations are similar to single linked list

Some of the functions for insertion operations are given below:

Function: insertf

```
/* function to insert a node as a first node of circular linear linked list */
void insertf()
 {
    struct studinfo   *newnode,*ptr;
    newnode=(struct studinfo *) malloc(sizeof(struct studinfo));
```

```
    printf("Enter a new  record : marks and name");
    scanf("%d%s",&newnode–>marks,newnode–>name);
    if(start == NULL)
            {
                    start = newnode;
                    newnode–>next = newnode;
            }
            else
              {
               ptr = start;
               while(ptr–>next ! = start)
                        ptr = ptr–>next;
              newnode–>next = start;
               start = newnode;
               ptr–>next = start;
      }
}
```

Function: insertl

```
/* Function to insert node as last node in the list */
void insertl( )
{
    struct studinfo *newnode, *ptr;
    newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
    printf("Enter a new  record : marks and name ");
    scanf("%d%s",&newnode–>marks,newnode–>name);
    if (start == NULL)
    {
            start = newnode;
            newnode–>next = newnode;
    }
    else
    {
            ptr = start;
            while (ptr–>next ! = start)
                    ptr = ptr–>next;
            ptr–>next = newnode;
            newnode–>next = start;
    }
} /* end of function */
```

Function: inserta

```
/* function to insert a node after a given node in circular
 linear linked list */
void inserta()
  {
   int cnt = 1, no;
      struct studinfo *ptr,*prevptr, *newnode;
       printf("\n enter number ...");
       scanf("%d",&no);
       ptr=start;
       while (cnt ! = no)
       {
       ptr = ptr–>next;
       cnt++;
       }
      newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
      printf("Enter a new  record : marks and name");
      scanf("%d%s",&newnode–>marks,newnode–>name);
      newnode–>next = ptr–>next;
      ptr–>next = newnode;
}
```

Function: insertb

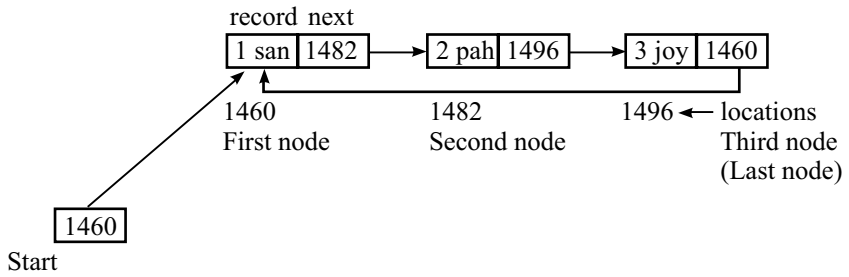
```
/* function to insert a node before a given node in circular
 linear linked list */
void insertb()
 {
      int cnt = 1, no;
      struct studinfo *ptr,*prevptr, *newnode;
       printf("\n enter number ...");
       scanf("%d",&no);
       ptr = start;
      if(no == 1)
       {
                insertf();
       }
       else
       {
                while(cnt ! = no)
                 {
                          prevptr = ptr;
                          ptr = ptr–>next;
```

```
                cnt++;
            }
            newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
            printf("Enter a new  record : marks and name " );
            scanf("%d%s",&newnode–>marks,newnode–>name);
            newnode–>next = ptr;
            prevptr–>next = newnode;
        }
}
```

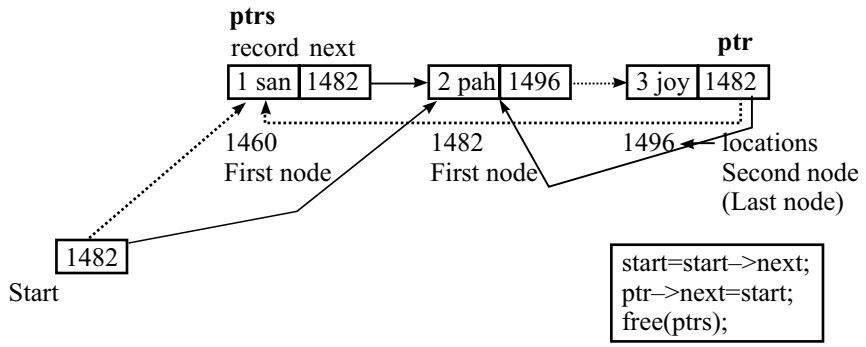
As we have deleted node in single linear lined list, we can also delete the node in circular linked list.

1 Deleting a node after the right-most node of the list (i.e., as last node)
2 Deleting a node before the left-most node of the list (i.e., as first node)
3 Deleting a node to the left of a given node (i.e., before a node)
4 Deleting a node to the right of a given node (i.e., after a node)
5 Deleting a node as a node number of the list

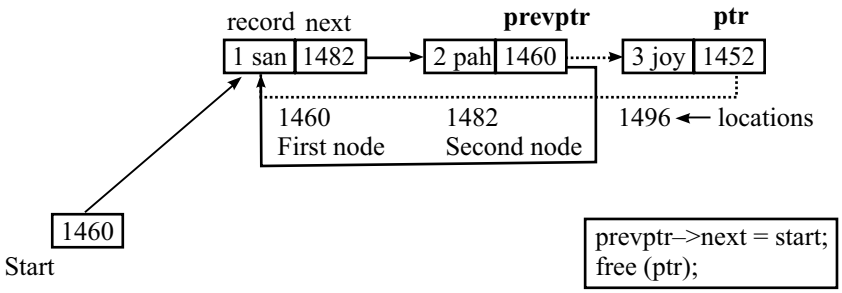
The deletion operations are given in below Fig. 4.19.



(*a*) Circular single-linked list before deletion



(*b*) Circular single-linked list after deletion of first node



(*c*) Circular single linked list after deletion of last node

Figure 4.19 *Deletion in circular linked list*

Function

The functions for deletions are given below.

Function: deletef

```
/* function to delete the first node of circular linear linked list */
void deletef()
 {
   struct studinfo *ptrs,*ptr;
   ptrs = start;
   ptr = start;
  if (start == NULL)
              printf("\n List is empty, element cannot be deleted");
     else
   {
   if (ptr–>next == start)
                  start = NULL;
   else
   {
        while(ptr–>next ! = start)
                          ptr = ptr–>next;
      start = start–>next;
      ptr–>next = start;
   }
   free(ptrs);
  }
}
```

Function: deletel

```
/* function to delete the last node of circular linear linked list */
 void deletel()
  {
     struct studinfo *ptr,*prevptr;
```

```
        ptr=start;
        if (start == NULL)
                printf("\n List is empty, element cannot be deleted");
        else
        {
        if (ptr–>next == start)
                    start = NULL;
        else
        {
                   while(ptr–>next!=start)
                    {
                              prevptr = ptr;
                              ptr = ptr–>next;
                    }
                    prevptr–>next = start;
        }
         free(ptr);
      }
    }
```

A complete 'C' program for circular linked list is given below:

/* Program to perform data structures operation on circular linear linked list. The node consists of a student record: marks and name */


```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct  studinfo{
      int marks;
      char name[10];
struct studinfo *next;
}*start;

void main()
{
int c;
void createlist();
void traverse();
void search();
void insertf();
void insertl();
void deletef();
void deletel();
void inserta();
```

```
void insertb( );
void deleteno( );
clrscr( );
start = NULL;
do {
clrscr ( );
printf("\n Enter choice for Circular Link List .... ");
printf("\n 1. Create Circular Link list.... ");
printf("\n 2. Insert node as First node.. ");
printf("\n 3. Insert node as Last node.. ");
printf("\n 4. Insert after node no...");
printf("\n 5. Insert before node no");
printf("\n 6. Delete First node ..");
printf("\n 7. Delete Last node ..");
printf("\n 8. Delete node(given no)");
printf("\n 9. Search a node ..");
printf("\n 10. Traverse Lists..");
printf("\n 11. Exit Lists..");
printf("\n Enter choice for Link List .... ");
scanf("%d",&c);
switch (c)
{
case 1:
    createlist();
    break;
case 2:
    insertf( );
    break;
case 3:
    insertl( );
    break;
case 4:
inserta( );
    break;
case 5:
    insertb( );
    break;
case 6:
    deletef( );
    break;
case 7:
    deletel( );
    break;
```

```
case 8:
      deleteno( );
      break;
case 9:
      search( );
      break;
case 10:
      traverse( );
      break;
}
}while (c! = 11);
}

/* function to create circular linear linked list */
void createlist( )
{
      struct studinfo *newnode, *ptr;
      char ch;
      do{
      newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
      printf("Enter a new record : marks and name ");
      scanf("%d%s",&newnode–>marks,newnode–>name);
      if (start == NULL)
      {
                start = newnode;
                newnode–>next = newnode;
      }
      else
      {
                ptr = start;
                while (ptr–>next ! = start)
                          ptr = ptr–>next;
                ptr–>next = newnode;
                newnode–>next = start;
      }
      fflush(stdin);
      printf("do you want to continue (y/n) ");
      scanf("%c",&ch);
  }while( ch =='y' || ch =='Y');
}

/* function to insert a node as a first node of circular linear linked list */
void insertf( )
```

```
{
    struct studinfo   *newnode,*ptr;
    newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
    printf("Enter a new  record : marks and name ");
    scanf("%d%s",&newnode–>marks,newnode–>name);
    if(start==NULL)
            {
                    start = newnode;
                    newnode–>next = newnode;
            }
            else
              {
              ptr = start;
              while(ptr–>next ! = start)
                        ptr = ptr–>next;
             newnode–>next = start;
              start = newnode;
              ptr–>next = start;
      }
}
/* function to insert a node as a last node of circular linear linked list*/
void insertl()
{
    struct studinfo *newnode, *ptr;
    newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
    printf("Enter a new  record : marks and name ");
    scanf("%d%s",&newnode–>marks,newnode–>name);
    if (start == NULL)
    {
            start = newnode;
            newnode–>next = newnode;
    }
    else
    {
            ptr =start;
            while (ptr–>next ! = start)
                    ptr = ptr–>next;
            ptr–>next = newnode;
            newnode–>next = start;
    }
} /* end of function */
/* function to insert a node after a given node in circular
 linear linked list */
```

```
void inserta( )
  {
   int cnt = 1, no;
      struct studinfo *ptr,*prevptr, *newnode;
       printf("\n enter number ...");
       scanf("%d",&no);
       ptr=start;
       while (cnt ! = no)
       {
       ptr = ptr–>next;
       cnt++;
       }
      newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
      printf("Enter a new  record : marks and name");
      scanf("%d%s",&newnode–>marks,newnode–>name);
      newnode–>next = ptr–>next;
      ptr–>next = newnode;
}

/* function to insert a node before a given node in circular
 linear linked list */
void insertb( )
 {
      int cnt = 1, no;
      struct studinfo *ptr,*prevptr, *newnode;
       printf("\n enter number ...");
       scanf("%d",&no);
       ptr = start;
      if(no == 1)
       {
                insertf( );
       }
       else
       {
                while(cnt ! = no)
                 {
                          prevptr = ptr;
                          ptr = ptr–>next;
                          cnt++;
                 }
                newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
                printf("Enter a new  record : marks and name");
```

```
                scanf("%d%s",&newnode–>marks,newnode–>name);
                newnode–>next = ptr;
                prevptr–>next = newnode;
        }
}

/* function to delete the first node of circular linear linked list */
void deletef()
 {
    struct studinfo *ptrs,*ptr;
   ptrs = start;
   ptr = start;
  if (start == NULL)
                printf("\n List is empty, element cannot be deleted");
      else
   {
   if (ptr–>next == start)
                           start = NULL;
   else
   {
        while(ptr–>next ! = start)
                             ptr = ptr–>next;
       start = start–>next;
       ptr–>next = start;
   }
   free(ptrs);
  }
}
/* function to delete the last node of circular linear linked list */
 void deletel( )
  {
      struct studinfo *ptr,*prevptr;
      ptr = start;
      if (start == NULL)
                printf("\n List is empty, element cannot be deleted");
      else
      {
      if (ptr–>next == start)
                 start = NULL;
      else
      {
                while(ptr–>next! = start)
                 {
```

```
                        prevptr = ptr;
                        ptr = ptr–>next;
                }
                prevptr–>next = start;
        }
          free(ptr);
    }
}
/* function to delete a node by node number a given node in circular
 linear linked list */
void deleteno( )
  {
       int cnt = 1,no;
       struct studinfo *ptr,*prevptr, *newnode;
       printf("\n enter number ...");
       scanf("%d",&no);
       if (start == NULL)
                printf( " List is empty, element cannot be deleted");
       else
       {
       if((start–>next == start)||(no == 1))
       {
                deletef( );
       }
      else
      {
      ptr = start;

      while(cnt! = no)
                {
                        prevptr = ptr;
                        ptr = ptr–>next;
                        cnt++;
                }
      if (ptr–>next == start)
                deletel( );

        prevptr–>next = ptr–>next;
        free(ptr);
      }
    }
} /* End of function */
```

```
/* Function search finds a record in the circular linear linked list */
void search()
{
    struct studinfo *ptr;
    char str[10];
    printf("enter name to be searched ");
    scanf("%s",str);
    ptr = start;
    while(ptr–>next! = start)
     {
            if(strcmp(ptr–>name,str) == 0)
             {
                    printf("\nfound");
                    break;
             }
            else
                    ptr = ptr–>next;
     }
    if(ptr == NULL)
            printf("\nnot found");
    getch();
}

/* Function traverse traverses circular linear linked list */
void traverse()
{
    struct studinfo *ptr;
    ptr = start;
    if (ptr)
    {
            while (ptr–>next ! = start)
            {
                    printf("\nRecord: marks and name %d  %s\n",ptr–>marks, ptr–>name);
                    ptr = ptr–>next;
            }
            if (ptr)
              printf("\nRecord: marks and name %d  %s\n", ptr–>marks, ptr–>name);
    }
    else
            printf("\nCircular list is empty ");
    getch();
}
```

4.3 DOUBLY LINEAR LINKED LISTS AND OPERATIONS

The single linked list has been restricted to traversing in only one direction (i.e., forward). Even in circular linked list, it is not possible to traverse to delete a node, given only a pointer to that node. Also it is not possible to traverse the list backwards.

In doubly linked linear list we can traverse in both forward and backward directions. That implies that each node must contain two pointer fields. The pointers are used to denote the predecessor and successor of a node. The pointer denoting the predecessor of a node is called left or previous pointer and that denoting successor is called right or next pointer. A list containing this type of node is called a doubly linked list or a two–way chain.

Such a doubly linear list is illustrated in Fig. 4.20, where first are last pointer variables denoting the left–most and right–most nodes in the list, respectively. The left pointer of the left–most node and right pointer of the right–most node are both NULL, indicating the end of the list for each direction. The variables left and right denote the left and right pointers of a node, respectively.

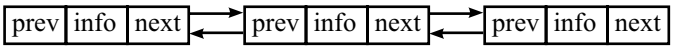


Figure 4.20 *Doubly linked list*

We can declare such a node in 'C' as the array implementation of doubly linear linked list as

```
struct   nodetype {
            int info;
            int prev, next;
}node[MAXSIZE];
```

Here we have created node as an array of structure nodetype with MAXSIZE size.

We can declare such a node in 'C' as the dynamic implementation of doubly linear linked list as

```
struct   nodetype {
            struct   nodetype    *prev;
            int info;
            struct   nodetype   *next;
};
typedef struct nodetype *nodeptr;
```

A doubly linked list may be either linear or circular and it may or may not contain a header node.

Creation of a Doubly Linear Linked List

The function creates doubly linked list node as record of student marks. The two pointer variable prev points to the previous node of the linked list and next pointer variable points to the next node of the linked list. The start is a pointer to structure studinfo points to the start node of the linked list.

```
struct studinfo{
int marks;
struct studinfo *next;
struct studinfo *prev;
}*start;
```

The function to create doubly linked list is createlist as follows:

1. Initially create the first node as the start node as linked list is empty and prev and next pointer of newnode is empty.
2. Append the next newnode to the list and set the prev and next pointer accordingly until choice is yes.

The function createlist in 'C' is given below:

```
void createlist()
{
struct studinfo *newnode, *ptr;
char ch;
do{
newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
printf("Enter a new number : marks ");
scanf("%d",&newnode–>marks);
newnode–>next = NULL;
newnode–>prev = NULL;
if (start == NULL)
    {
    start = newnode;
    }
else
    {
    ptr = start;
    while (ptr–>next!=NULL)
            ptr = ptr–>next;
    ptr–>next = newnode;
    newnode–>prev = ptr;
     }
fflush(stdin);
printf("do you want to continue (y/n) ");
scanf("%c",&ch);
}while( ch =='y' || ch =='Y');
}
```

Insertion in Doubly Linear Linked List

Consider the problem of inserting a node into a doubly linear linked list. There are number of cases possible.

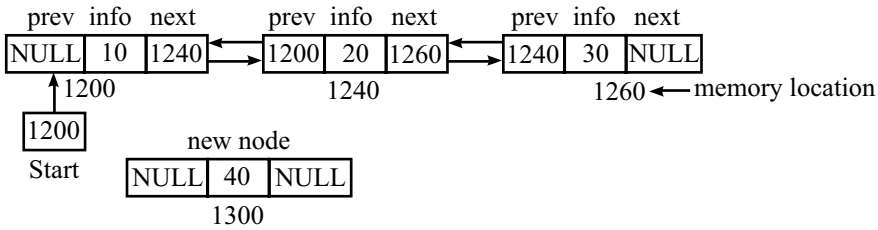
1. Inserting a node when list is empty
2. Inserting a node after the right–most node of the list (i.e., as last node)
3. Inserting a node before the left–most node of the list (i.e., as first node)
4. Inserting a node to the left of a given node (i.e., before a node)
5. Inserting a node to the right of a given node (i.e., after a node)
6. Inserting a node as a node number of the list

A general algorithm for inserting a node to the left of a given node in a doubly linear linked list is as follows:

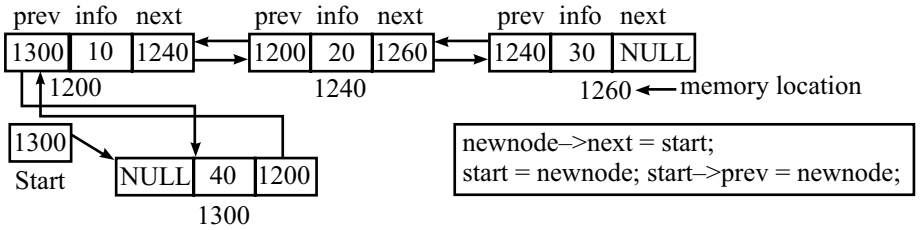
- Allocate the memory for a new node and assign data to its fields.
- If the list is empty then insert the node in the list and update left and right pointers to the list and return.

- If the node is to be inserted at the front of the list then insert the node and update the left pointer to the list and return.
- Otherwise insert the node in the middle of the list and return.

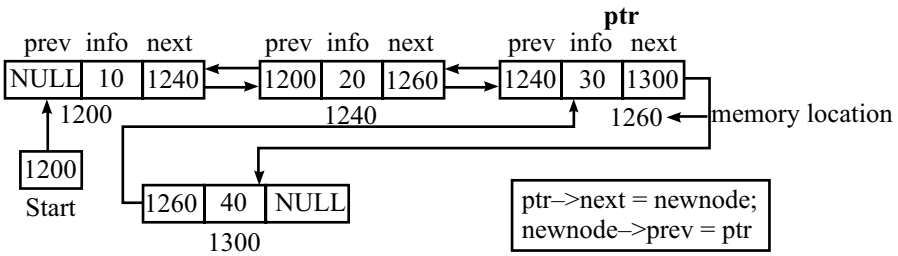
The following Fig. 4.21 illustrates the method of inserting node in the doubly linked list. {10, 20, 30}



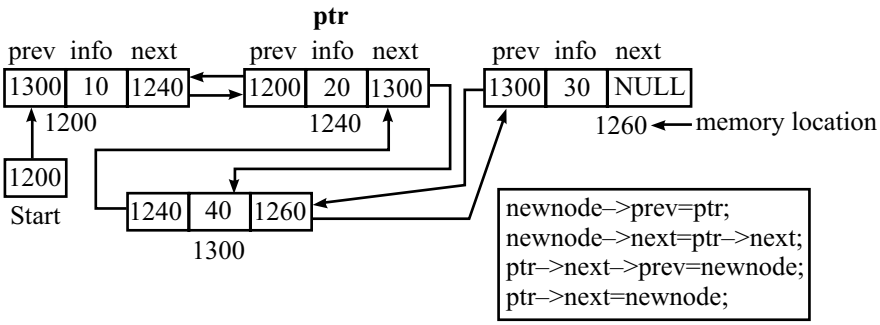
(*a*) List before insertion, memory for newnode has been created and value is assigned {10, 20, 30}.



(*b*) List after insertion as first node {40, 10, 20, 30}



(*c*) List after insertion as last node {10, 20, 30, 40}



(*d*) List after insertion: Inserted after node number 2 {10, 20, 40, 30}

Figure 4.21 *Doubly linked list: insertion of newnode*

The function for the insertion as first node insertf() is given below:

```
void insertf()
 {
   struct studinfo *newnode, *ptr;
     newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
     printf("Enter a new number : marks ");
     scanf("%d",&newnode–>marks);
     newnode–>next = NULL;
     newnode–>prev = NULL;
     newnode–>next = start;
     start–>prev = newnode;
     start=newnode;
 }
```

The function for the insertion as last node insertl() is given below:

```
/* function insertl, insert node as last node in the linked list */
void insertl()
{
     struct studinfo *newnode, *ptr;
     newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
     printf("Enter a new number : marks ");
     scanf("%d",&newnode–>marks);
     newnode–>next = NULL;
     newnode–>prev = NULL;
     if (start == NULL)
                start =newnode;
     else
     {
     ptr = start;
     while (ptr–>next! = NULL)
                ptr = ptr–>next;
     ptr–>next = newnode;
     newnode–>prev = ptr;
     }
}
```

The function for the insertion after a given node number is given below:

```
void inserta()
  {
     int no,cnt = 1;
     struct studinfo *newnode, *ptr;
     printf("insert a node number");
```

```
        scanf("%d",&no);
        ptr = start;
        while((no! = cnt) &&(ptr! = NULL))
         {
                ptr = ptr–>next;
                cnt++;
         }
        newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
        printf("Enter a new number : marks ");
        scanf("%d",&newnode–>marks);
        newnode–>prev = ptr;
        newnode–>next = ptr–>next;
        ptr–>next–>prev = newnode;
        ptr–>next = newnode;
    }
```

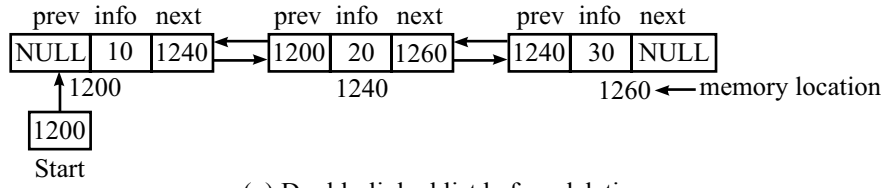
Some of the functions for insertion are given, the remaining functions for insertion are given in the program itself.

Deletion in Doubly Linear Linked List

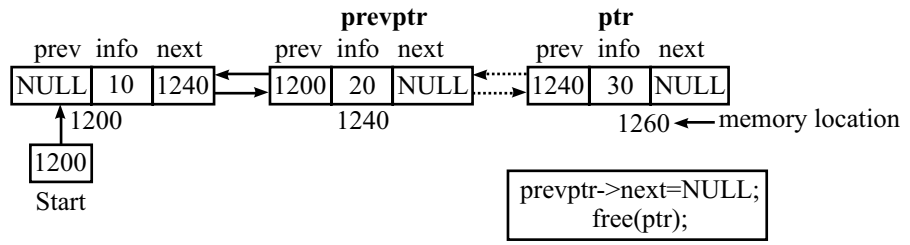
As we have deleted node in single linear lined list, we can also delete the node in doubly linked list.

1. Deleting a node after the right-most node of the list (i.e., as last node)
2. Deleting a node before the left-most node of the list (i.e., as first node)
3. Deleting a node to the left of a given node (i.e., before a node)
4. Deleting a node to the right of a given node (i.e., after a node)
5. Deleting a node as a node number of the list

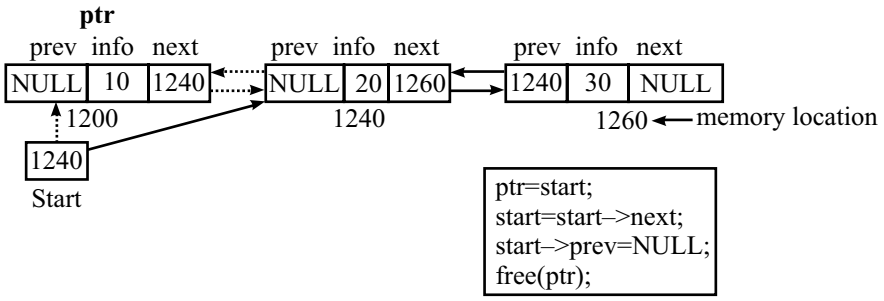
The following Fig. 4.22 illustrates the method of inserting node in the doubly linked list.



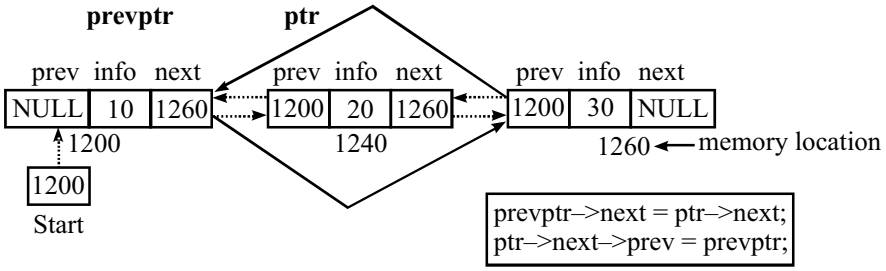
(*a*) Doubly linked list before deletion



(*b*) Doubly linked list after last node deletion



(*c*) Doubly linked list after first node deletion



(*d*) Doubly linked list after node 2 deletion

Figure 4.22 *Deletion in doubly linked list*

Some of the functions to delete the node are given here and rest are within the program.

The deletef function deletes the first node in the doubly linked list.

```
void deletef( )
 {
     struct  studinfo  *ptr;
     if(start == NULL)
     {
               printf("list is empty");
               return;
     }
     ptr = start;
     start = start–>next;
     start–>prev = NULL;
     free(ptr);
 }
```

The function deletel deletes the last node in the doubly linked list.

```
void deletel( )
 {
     struct  studinfo  *ptr,*prevptr;
     ptr = start;
     while(ptr–>next!=NULL)
```

```
    {
            prevptr = ptr;
            ptr = ptr–>next;
    }
  prevptr–>next = NULL;
   free(ptr);

}
```

A complete 'C' program to perform insert, delete and search functions in the doubly linked list is given below:


```
/* program to perform data structure operations on doubly linked list */

#include<stdio.h>
#include<conio.h>
#include<string.h>
struct  studinfo{
     int marks;
     struct  studinfo  *next;
     struct  studinfo  *prev;
     }*start;

void main()
{
int c;
void createlist();
void traverse();
void traverser();
void deletef();
void deletel();
void inserta();
void insertf();
void insertl();
void  search();
void insert();
clrscr();
start = NULL;

do {
printf("\n Enter choice for doubly linked list .... ");
printf("\n 1.Insert in List.... ");
printf("\n 2.Traverse ..");
printf("\n 3.Traverse List in reverse ..");
printf("\n 4.Delete first node ..");
printf("\n 5.Delete last node ..");
```

```
printf("\n 6.Insert as a first node..");
printf("\n 7.Insert as a last node..");
printf("\n 8.Search node..");
printf("\n 9.Insert a node after node no..");
printf("\n 10.Insert a node after a given node ");
printf("\n 11.Exit from list..");
printf("\n Enter choice for list .... ");
scanf("%d",&c);
switch (c)
{
case 1:
     createlist();
     break;
case 2:
     traverse();
     break;
case 3:
     traverser();
     break;
case 4:
     deletef();
     break;
case 5:
     deletel();
     break;
case 6:
     insertf();
     break;
case 7:
     insertl();
     break;
case 8:
     search();
     break;
case 9:
     inserta();
     break;
case 10:
     insert();
     break;
}
}while (c! = 11);
}/* end of main */
```

```
/* function createlist, creates a doubly linked list until choice is not 'y'*/
void createlist()
{
    struct studinfo *newnode, *ptr;
    char ch;
    do{
    newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
    printf("Enter a new number : marks ");
    scanf("%d",&newnode–>marks);
    newnode–>next = NULL;
    newnode–>prev = NULL;
    if (start == NULL)
            start =newnode;
    else
    {
    ptr = start;
    while (ptr–>next! = NULL)
            ptr = ptr–>next;
    ptr–>next = newnode;
    newnode–>prev = ptr;
    }
    fflush(stdin);
    printf("do you want to continue (y/n) ");
    scanf("%c",&ch);
 }while( ch =='y' || ch =='Y');
} /* end of function */

/* function traverse, displays each node of the list */
void traverse( )
{
    struct studinfo *ptr;
    ptr = start;
if (start= = NULL)
{
printf ("\n List is empty");
return;
}
    while (ptr! = NULL)
    {
    printf("\t %d",ptr–>marks);
    ptr = ptr–>next;
    }
    getch( );
}   /* end of function */

/* function traverser, displays each node of the list in reverse*/
void traverser( )
```

```
{
      struct  studinfo  *ptr;
      ptr = start;
if (start== NULL)
{
printf ("\n List is empty");
return;
}
      while (ptr–>next! = NULL)
      ptr = ptr–>next;
      while(ptr!=NULL)
      {
      printf("\t%d",ptr–>marks);
      ptr = ptr–>prev;
      }
}   /* end of function */

/* function deletef, deletes the first node of the linked list */
void deletef( )
 {
      struct  studinfo  *ptr;
      if(start == NULL)
      {
                  printf("list is empty");
                  return;
      }
      ptr = start;
      start = start–>next;
      start–>prev = NULL;
      free(ptr);
}/* end of function */

/* function deletel, deletes the last node of the linked list */
void deletel( )
 {
      struct  studinfo  *ptr,*prevptr;
      ptr = start;
      while(ptr–>next! = NULL)
      {
                  prevptr = ptr;
                  ptr=ptr–>next;
      }
   prevptr–>next = NULL;
    free(ptr);
}/* end of function */

/* function insertf, inserts node as first node in the linked list */
void insertf( )
```

```
 {
   struct studinfo *newnode, *ptr;
     newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
     printf("Enter a new number : marks ");
     scanf("%d",&newnode–>marks);
     newnode–>next = NULL;
     newnode–>prev = NULL;
     newnode–>next = start;
     start–>prev = newnode;
     start = newnode;
}/* end of function */

/* function insertl, inserts node as last node in the linked list */
void insertl()
{
     struct studinfo *newnode, *ptr;
     newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
     printf("Enter a new number : marks ");
     scanf("%d",&newnode–>marks);
     newnode–>next = NULL;
     newnode–>prev = NULL;
     if (start == NULL)
               start =newnode;
     else
     {
     ptr = start;
     while (ptr–>next! = NULL)
               ptr = ptr–>next;
     ptr–>next = newnode;
     newnode–>prev=ptr;
     }
}
/* function searches the given node in the list */
void search()
 {
     int m;
     struct studinfo *ptr;
     printf("\nEnter marks to be searched");
     scanf("%d",&m);
     ptr = start;
     while(ptr! = NULL)
      {
               if(ptr–>marks == m)
                {
                         printf("\nfound");
```

```
                    break;
                }
                else
                    ptr = ptr–>next;
        }
        if(ptr == NULL)
                printf("\n not found");
}/* end of function */

/* function inserta, inserts node after a given node number in
the linked list */
void inserta()
  {
        int no,cnt = 1;
        struct studinfo *newnode, *ptr;
        printf("insert a node number");
        scanf("%d",&no);
        ptr = start;
        while((no! = cnt) &&(ptr! = NULL))
         {
                ptr = ptr–>next;
                cnt++;
         }

        newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
        printf("Enter a new number : marks");
        scanf("%d",&newnode–>marks);

        newnode–>prev = ptr;
        newnode–>next = ptr–>next;
        ptr–>next–>prev = newnode;
        ptr–>next = newnode;

}/* end of function */

/* function insert, inserts node after a given node (info) in the
 linked list */
void insert()
{
        int no;
        struct studinfo *newnode, *ptr;
        printf("insert a mark of a node after which node to be inserted");
        scanf("%d",&no);
```

```
        newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
        printf("Enter a new number : marks ");
        scanf("%d",&newnode–>marks);

        ptr = start;
        while((ptr!=NULL))
         {
         if (ptr–>marks == no)
                        break;

                ptr = ptr–>next;
         }
         if (ptr!=NULL)
         {
          newnode–>prev = ptr;
          newnode–>next = ptr–>next;
          ptr–>next–>prev = newnode;
          ptr–>next = newnode;
          }
          else
                printf("Node not found, cannot insert");
    } /* end of function */
```

Header and Circular doubly linked list. Similarly, circular linear linked list, doubly circular linked list, header doubly linked list, header circular doubly linked list can be implemented.

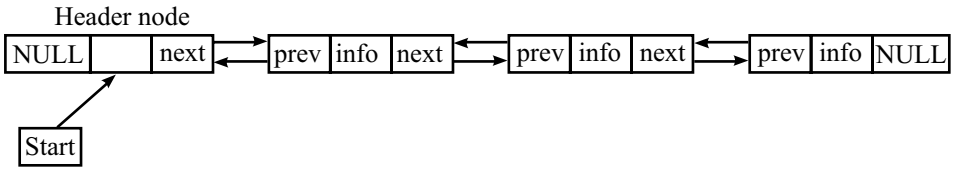


Figure 4.23 *Header grounded doubly linked list*

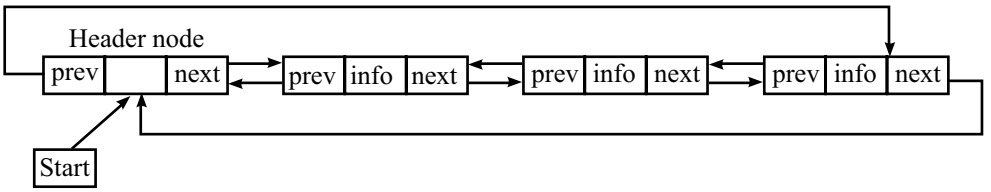


Figure 4.24 *Header circular doubly linked list (3 nodes)*

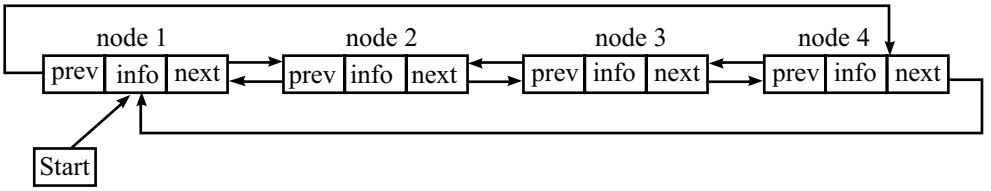


Figure 4.25 *Circular doubly linked list (4 nodes)*

The functions for create, insert, delete in the **header circular doubly linked list** are given below.

Function: insertf

```
/* function insertf, inserts node as first node in the linked list */
void insertf()
 {
   struct studinfo *newnode, *ptr;
     newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
     printf("Enter a new number : info ");
     scanf("%d",&newnode–>info);
     newnode–>next = NULL;
     newnode–>prev = NULL;

     newnode–>next = head–>next;
     newnode–>prev = head;
     head–>next–>prev = newnode;
     head–>next = newnode;

}/* end of function */
```

Function: insertl

```
/* function insertl, inserts node as last node in the linked list */
void insertl()
{
     struct studinfo *newnode, *ptr;
     newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
     printf("Enter a new number : info ");
     scanf("%d",&newnode–>info);
     newnode–>next = NULL;
     newnode–>prev = NULL;
     if (head–>next == NULL)
      {
               head–>next = newnode;
               newnode–>next = head;
               newnode–>prev = head;
               head–>prev = newnode;
      }
     else
     {
     ptr = head;
     while (ptr–>next! = head)
               ptr = ptr–>next;
     ptr–>next = newnode;
```

```
        newnode–>prev = ptr;
        newnode–>next = head;
        head–>prev = newnode;
        }
}
```

Function: deletef

```
/* function deletef, deletes the first node of the linked list */
void deletef()
 {
        struct studinfo *ptr, *ptrs;
        if(head–>next == NULL)
        {
                printf("list is empty");
                return;
        }
        if (head–>next–>next == head)
        {
                head–>next = NULL;
                head–>prev = NULL;
        }
        else
        {
        ptr = head–>next;
        head–>next = ptr–>next;
        ptr–>next–>prev = head;
        }
        free(ptr);
}/* end of function */
```

Function: deletel

```
/* function deletel, deletes the last node of the linked list */
void deletel()
 {
        struct studinfo *ptr,*prevptr;
        ptr = head–>next;
        if(head–>next == NULL)
        {
                printf("list is empty");
                return;
        }
```

```
    if (head–>next–>next == head)
    {
            head–>next = NULL;
            head–>prev = NULL;
    }
    else
    {
    while(ptr–>next! = head)
    {
            prevptr = ptr;
            ptr = ptr–>next;
    }
  prevptr–>next = head;
  head–>prev = prevptr;
  free(ptr);
 }
}/* end of function */
```

A complete 'C' program to perform different data structure operations on **header circular doubly linked** list is given below:

```
/* Program  hcdbllst.c to perform data structure operations on header circular doubly linked list and basic
 functions create, insert, delete, traverse and search are given*/

#include<stdio.h>
#include<conio.h>
#include<string.h>
struct  studinfo{
    int info;
    struct  studinfo  *next;
    struct  studinfo  *prev;
    }*start,  *head;

void main()
{
int c;

void createlist();
void traverse();
void insertf();
void insertl();
void deletef();
void deletel();
void search();
clrscr();
```

```
head = (struct studinfo *) malloc(sizeof(struct studinfo));
head–>next = NULL;  /* Initialize empty header circular linked list */
head–>prev = NULL;
start = head;

do {
printf("\n Enter choice for header circular doubly linked list .... ");
printf("\n 1.Create List.... ");
printf("\n 2.Traverse ..");
printf("\n 3.Insert as a first node..");
printf("\n 4.Insert as a last node..");
printf("\n 5.Delete first node ..");
printf("\n 6.Delete last node ..");
printf("\n 7.Search node..");
printf("\n 8.Exit from list..");
printf("\n Enter choice for list .... ");
scanf("%d",&c);
switch (c)
{
case 1:
     createlist();
     break;
case 2:
     traverse();
     break;
case 3:
     insertf();
     break;
case 4:
     insertl();
     break;
case 5:
     deletef();
     break;
case 6:
     deletel();
     break;
case 7:
     search();
     break;
}
}while (c! = 8);
}/* end of main */
```

```
/* function createlist, creates a doubly linked list until choice is not 'y'*/
void createlist()
{
    struct studinfo *newnode, *ptr;
    char ch;
    do{
    newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
    printf("Enter a new number : info ");
    scanf("%d",&newnode->info);
    newnode->next = NULL;
    newnode->prev = NULL;
    if (head->next == NULL)
     {
            head->next = newnode;
            newnode->next = head;
            newnode->prev = head;
            head->prev = newnode;
     }
    else
    {
    ptr = head;
    while (ptr->next! = head)
            ptr = ptr->next;
    ptr->next = newnode;
    newnode->prev = ptr;
    newnode->next = head;
    head->prev = newnode;
    }
    fflush(stdin);
    printf("do you want to continue (y/n) ");
    scanf("%c",&ch);
 }while( ch == 'y' || ch =='Y');
} /* end of function */

/* function traverse, displays each node of the list */
void traverse()
{
    struct studinfo *ptr;
    ptr = head->next ;
    if (ptr)
    {
            while (ptr->next ! = head)
            {
```

```
                printf("\t %d",ptr–>info);
                ptr = ptr–>next;
                }
                if (ptr)
                        printf("\t %d",ptr–>info);
        }
        else
                printf("\n Header circular doubly lined list is empty" );
        getch();
}   /* end of function */

/* function insertf, inserts node as first node in the linked list */
void insertf()
 {
   struct studinfo *newnode, *ptr;
     newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
     printf("Enter a new number : info ");
     scanf("%d",&newnode–>info);
     newnode–>next = NULL;
     newnode–>prev = NULL;
     newnode–>next = head–>next;
     newnode–>prev = head;
     head–>next–>prev = newnode;
     head–>next = newnode;
}/* end of function */

/* function insertl, inserts node as last node in the linked list */
void insertl()
{
     struct studinfo *newnode, *ptr;
     newnode = (struct studinfo *) malloc(sizeof(struct studinfo));
     printf("Enter a new number : info ");
     scanf("%d",&newnode–>info);
     newnode–>next = NULL;
     newnode–>prev = NULL;
     if (head–>next == NULL)
      {
                head–>next = newnode;
                newnode–>next = head;
                newnode–>prev = head;
                head–>prev = newnode;
      }
```

```
    else
    {
    ptr = head;
    while (ptr–>next! = head)
            ptr = ptr–>next;
    ptr–>next = newnode;
    newnode–>prev = ptr;
    newnode–>next = head;
    head–>prev = newnode;
    }

}

/* function deletef, deletes the first node of the linked list */
void deletef()
 {
    struct studinfo *ptr, *ptrs;
    if(head–>next == NULL)
    {
            printf("list is empty");
            return;
    }
    if (head–>next–>next == head)
    {
            head–>next = NULL;
            head–>prev = NULL;
    }
    else
    {
    ptr = head–>next;
    head–>next = ptr–>next;
    ptr–>next–>prev = head;
    }
    free(ptr);
}/* end of function */

/* function deletel, deletes the last node of the linked list */
void deletel()
 {
    struct studinfo *ptr,*prevptr;
    ptr = head–>next;
    if(head–>next == NULL)
    {
```

```
                printf("list is empty");
                return;
        }
        if (head–>next–>next == head)
        {
                head–>next = NULL;
                head–>prev = NULL;
        }
        else
        {

        while(ptr–>next! = head)
        {
                prevptr = ptr;
                ptr = ptr–>next;
        }
      prevptr–>next = head;
      head–>prev = prevptr;
      free(ptr);
     }
}/* end of function */
/* function searches the given node in the list */
void search()
 {
        int m;
        struct studinfo *ptr;
        printf("\nEnter info to be searched ");
        scanf("%d",&m);
        ptr = head–>next;
        while(ptr! = head)
         {
                if(ptr–>info == m)
                 {
                        printf("\nSuccessfully searched ");
                        break;
                 }
                else
                        ptr = ptr–>next;
         }
        if(ptr == NULL)
                printf("\n Unsuccessful search");
}/* end of function */
```

The remaining functions and algorithms can be easily developed.

4.4 APPLICATIONS OF LINKED LISTS

Linked lists are used in a wide variety of applications. Some of the applications are given below:

1. The polynomial manipulation.
2. To maintain directory of names. This type of problem occurs in compiler construction to store the list of identifiers appearing in a program for maintaining symbol table.
3. To perform arithmetic operations on long integers.
4. To implement sparse matrices.
5. To implement a line editor, where we can keep a linked list of line nodes, each containing a line number, a line of text and a pointer to the next line node.

4.4.1 Polynomial Manipulation

The problem of manipulation of symbolic polynomials is the classical example of the use of list processing. We can perform various operations on polynomials such as addition, subtraction, multiplication, division, differentiation, etc. We can represent any number of different polynomials as long as their combined size does not exceed our block of memory. In general, we want to represent the polynomial of single variable

$$f(x) = a_m x^m + \ldots + a_i x^i + \cdots + a_0$$

of degree m, where a_i are non–zero coefficients with exponents i. Each term will be represented by a node. A node will be of fixed size having three fields, which represent the coefficient and exponent of a term plus a pointer to the next term.

Let us consider the problem of implementing a polynomial in the linked linear lists using arrays. Consider the example of representing a term in a polynomial of a variable x, with coefficient c. Thus a node of the linked list consists of three fields, one each for exponent x and coefficient c, and third one represents a pointer to the next node. This pointer is an index of the array. We can declare three vectors: powerx, coeff and link, each representing a subscript giving the location of the next node in the list.

In a similar fashion, we can represent a term in a polynomial of the variables x, y, and z. It required five vectors: powerx, powery, powerz, coeff, and link, each representing a subscript giving the location of the next node in the list. But it required multiple vectors, instead of these we can declare array of structure with these fields.

The array implementation is static, thus arrays are not always well suited for easily expressing the relationships that exist among data elements.

Assuming that all coefficients are integers, the static implementation of such node with three fields in 'C' is as

```
struct nodetype {
        int     powerx;
        int     coef;
        int     link;
}pterms[MAXTERM];
```

In similar manner, the static implementation of such node with five fields in 'C' is as

```
struct nodetype {
        int     powerx;
        int     powery;
        int     powerz;
        int     coef;
        int     link;
}pterms[MAXTERM];
```

Now consider the dynamic implementation of a polynomial in linear linked lists using pointers.

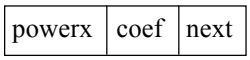
Assuming that all coefficients are integers, the dynamic implementation of such node with three fields in 'C' is as

```
struct nodetype {
        int     powerx;
        int     coef;
        struct nodetype *next;
};
typedef struct nodetype *p;
```

In similar manner, the dynamic implementation of such node with five fields in 'C' is as

```
struct nodetype {
        int     powerx;
        int     powery;
        int     powerz;
        int     coef;
        struct nodetype *next;
};
typedef struct nodetype *p;
```

A term of a polynomial of single variable x, is drawn as below.

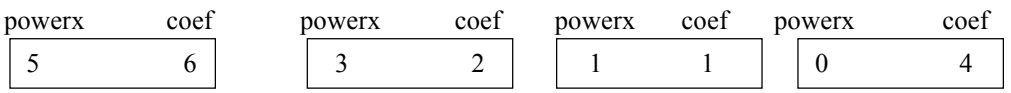


We can view a polynomial such as

$$p = 6x^5 + 2x^3 + x + 4$$

as being represented by a list of terms as shown below

We can replace each term of the polynomial with node of the linked list as below.



We always store terms of the polynomial in descending order of powers. For example, $10x + 2x^4 + 1$ would be stored as $2x^4 + 10x + 1$. This order is maintained to achieve simplicity and greater efficiency in processing.

Consider the example of representing a term of a polynomial in the variables x, y, and z. A typical node is represented as in Fig. 4.26, which consists of five sequentially allocated fields that we collectively refer to as term. The first three fields represent the power of the variables x, y, and z, respectively. The fourth and fifth fields represent the coefficient of the term in the polynomial and the address of the next term in the polynomial, respectively. For example, term $2x^2z$ would be represented.

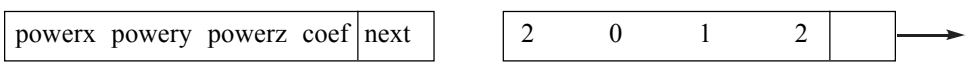


Figure 4.26

We can view a polynomial such as

$$p = 2x^3 + 3xy + y^2 + yz$$

We can replace each term of the polynomial with node of the linked list as

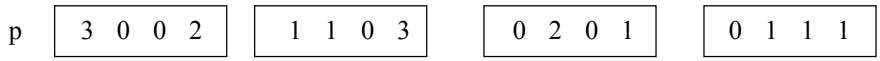


Figure 4.27

Assume that the nodes in the list are to be stored such that the terms with power of variable x are stored first in decreasing order of the power, then terms with power of variable y are stored in decreasing order of the power, then terms with power of variable z are stored in decreasing order of the power.

If a term pointed by p precedes another term indicated by q if p–>powerx is greater than q–>powerx; or, if the powers of x are equal, then p–>powery must be greater than q–>powery; or, if the powers of y are equal, then p–>powerz must be greater than q–>powerz. See Fig. 4.27.

Construction/Creation of a Polynomial as a Linear Linked List

Let us consider the problem of inserting a term of polynomial as a linked list node. The following steps are necessary to accomplish this.

1. Allocate the necessary memory for the new node.
2. Input/Assign the values for the various fields of the new node.
3. The linking of the new node to its successor in the existing list is accomplished by setting the next pointer field of the former to a value giving the location of the later.

Construction of ordered linked list for a single-variable polynomial

We can write an algorithm PINSERT, which inserts a term of a single variable polynomial into a linked list. The algorithm is written in the form of function, which is used to build an ordered linked representation of a polynomial.

Each term of a polynomial contains the information fields powerx and coef, and a pointer field next. This function inserts that term so as to preserve the order of the list. The fields of the new term are denoted by x and c, which correspond to the exponents for x, and the coefficient value of the term. The address of the first term in the list is given by pointer variable start and newnode is the pointer variable for new node. This function invokes a findloc function, which returns the appropriate address for the new node in the ordered list.

Suppose that a term is to be inserted into ordered linked list. The information part of the inserted node are exponents of x and a coefficient c. The term must be inserted between nodes A and B so that

A–>powerx > x > B–>powerx

The following is a procedure, which finds the location/address of the node A.

Traverse the list, using a pointer variable ptr and comparing item with ptr–>powerx at each node. While traversing, keep track of the address of the preceding node by using a pointer variable prevptr. Thus prevptr and ptr are updated by the assignments

prevptr = ptr; and ptr = ptr–>next;

Here we have given two algorithms: the algorithm FINDLOC finds the address loc of the last node in the ordered list such that loc–>powerx > x, or sets loc =NULL. The cases where the list is empty or where x > start–>powerx, so that loc = NULL is set. This algorithm is written as the sub-algorithm in the form of function. Then the function FINDLOC returns the loc pointer.

The second algorithm PINSERT calls the algorithm FINDLOC to return the loc pointer. If loc pointer is NULL then insert the node as first node otherwise insert after node with address holds in loc pointer.

The algorithm FINDLOC is written in the form of function.

Algorithm FINDLOC(start, x)
[This algorithm returns the address/location for a new term in an ordered list.]
Step 1 : if start = NULL then
return(NULL); [List is empty]
Step 2 : if x > start–> powerx then
return(NULL); [the x is larger than the first node item, thus must be inserted as the first node of the list]
Step 3 : Set prevptr = start; and ptr = start–>next; [Initializes pointers]
Step 4 : repeat steps 5 and 6 while ptr<> NULL
Step 5 : if x > ptr–> powerx then
return(prevptr);
Step 6 : Set prevptr = ptr;and ptr = ptr–>next; [Updates pointers]
Step 7 : return(prevptr);

The algorithm PINSERT calls the FINDLOC as follows:

Algorithm PINSERT(start, x, c)
[This algorithm inserts term into an ordered linked list.]
Step 1 : newnode = Allocate memory using malloc() function
Step 2 : Set newnode–>powerx = x; [Assigns value of the information part of a new node]
Set newnode–>coef = c;
Step 3 : loc = FINDLOC(start, x); [returns the address of the node succeeding x]
Step 4 : if loc = NULL then [Inserts as first node]
Set newnode–>next = start; and start = newnode;
else [Inserts after node with address loc]
Set newnode–>next = loc–>next; and loc–>next = newnode;
[End of If statement]
Step 5 : return(start);

The algorithm considers the case of a null or empty list first of all. That is, when location loc is NULL then insert the new node as the first node of the list and return the new node address. It also considers the case when new node is preceding the first node in the original list. The new node becomes the first node in the updated list. The next pointer field of the new node is assigned a pointer value which corresponds to the location of the first node of the original list. In other cases, the next pointer field of the new node is assigned the address of the node indicated by next pointer of the location loc and the next pointer field of the loc is assigned a pointer value of new node.

The average time for such an insertion is O(n/2).

The complete 'C' program with function is given below.

/* polyiall.cpp */

/* This program inserts a polynomial $3x^2 + 20x + 2$ in a single variable list. The function pinsert constructs the linked representations of these polynomials. Each term consists of two inputs, the first input corresponds to the power of x and the second to the coefficient of the term. This function invokes a findloc function which returns the location of the new term in the ordered list. */

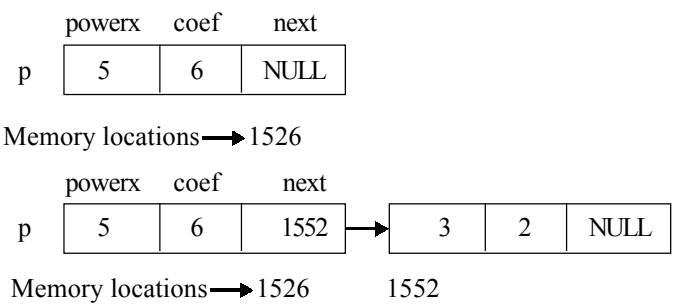
```
#include<stdio.h>
#include<malloc.h>
struct  nodetype{
     int powerx;
     int coef;
     struct nodetype *next;
     }*p,*q,*r, *newnode;
     struct nodetype *pinsert(struct nodetype *, int, int);
     struct nodetype *findloc(struct nodetype *, int);
     void display(struct nodetype *);
void main()
{
     p = NULL;          /* Initially polynomial list is empty */
     p = pinsert(p,2, 3);
     p = pinsert(p,1, 20);
     p = pinsert(p,0, 2);
     printf("\n Polynomial P ");
     display(p);
} /* End of the main function */

/* function, pinsert, inserts after node with address loc */
struct nodetype *pinsert(struct nodetype *start, int x, int c)
{
     struct nodetype *loc;
     newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
     newnode–>powerx = x;
     newnode–>coef = c;
     loc = findloc(start, x);
     if (loc == NULL)
     {
               newnode–>next = start;
               start = newnode;
     }
     else
     {
               newnode–>next = loc–>next;
               loc–>next = newnode;
     }
return (start);
}/* end of the pinsert function*/
/* function, findloc searches the given node in the list*/
struct nodetype *findloc(struct nodetype *start, int x)
{
     struct nodetype *prevptr, *ptr;
```

```
    if (start == NULL)
      return (NULL);
    if (x >start–>powerx)
            return (NULL);
    prevptr = start;
    ptr     = start–>next;
    while (ptr! = NULL)
    {
            if (x > ptr–>powerx)
                    return (prevptr);
            prevptr = ptr;
            ptr = ptr–>next;
    }
return (prevptr);
}/* End of the findloc function */
/* function display, display each term of the polynomial*/
void display(struct nodetype *start)
{
    struct nodetype *ptr;
    ptr = start;
    printf("\n powerx coef");
    while(ptr! = NULL)
    {
            printf("\n %d\t %d", ptr–>powerx , ptr–>coef);
            ptr = ptr–>next;
    }
}   /* End of the display function */
```

We can easily obtain an ordered linear linked list. For example, the sequence of statements

Start = NULL
Start = PINSERT(start, 5, 6)
Start = PINSERT(start, 3, 2)
Start = PINSERT(start, 1, 1)
Start = PINSERT(start, 0, 4)

creates a four–element list. A trace of this creation is given in Fig. 4.28.



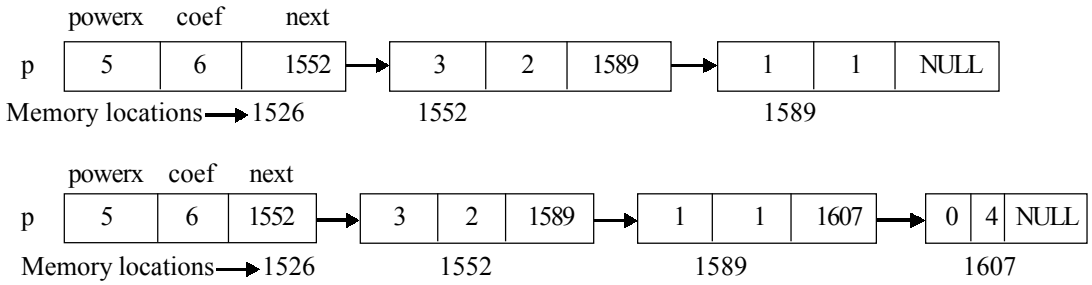


Figure 4.28 *Creation of the ordered linear linked list for a polynomial* $p = 6x^5 + 2x^3 + x + 4$

Construction of Ordered Linked List for a Polynomial with Three Variables

We can write an algorithm PINSERT, which inserts a term of a polynomial having three variables into a linked list. The algorithm is written in the form of function, which is used to build an ordered linked representation of a polynomial.

Each term of a polynomial contains the information fields powerx, powery, powerz and coef, and a pointer field next. This function inserts that term so as to preserve the order of the list. The fields of the new term are denoted by x, y, z and c, which correspond to the exponents for x, y, z and the coefficient value of the term. The address of the first term in the list is given by pointer variable start and newnode is the pointer variable for new node. This function invokes a findloc function, which returns the appropriate location for the new node in the ordered list.

Suppose that a term is to be inserted into ordered linked list. The information part of the inserted node are exponents of x, y, and z and a coefficient c. A new term may be inserted as the first node of the list, last node of the list or in between.

- If the list is empty then new term is inserted as the first node of the list.
- The new term precedes the start term if x is greater than start–>powerx; or, if the powers of x are equal, then y must be greater than start–>powery; or, if the powers of x are equal, and if the powers of y equal, then z must be greater than start–>powerz.
- The new term succeeds the term, which is indicated by pointer variable ptr if x is less than ptr–>powerx; or, if the powers of x are equal, then y must be less than ptr–>powery; or, if the powers of y equal, then z must be less than ptr–>powerz.

The algorithm FINDLOC is written in the form of function.

Algorithm FINDLOC(start, x, y, z)
[This algorithm returns the address/location for the new term in the ordered list.]

Step 1 : if start = NULL then
return(NULL); [List is empty]

Step 2 : if (x > start–>powerx) or((x = start–>powerx) and (y > start–>powery)) or
((x = start–>powerx) and (y = start–>powery) and (z > start–>powerz)) then
return(NULL);

Step 3 : Set ptr = start; [Initializes pointer]

Step 4 : repeat steps 5 and 6 while ptr–>next != NULL

Step 5 : Set a = ptr–>next–>powerx;
Set b = ptr–>next–>powery;
Set c = ptr–>next–>powerz;

Step 6 : If (x < a) or((x = a) and (y > b)) or ((x = a) and (y = b) and (z > c)) then
ptr = ptr–>next;
else
Exitloop;

Step 7 : return(ptr);

The algorithm PINSERT calls the function FINDLOC and has five inputs as follows:

Algorithm PINSERT(start, x, y, z, c)
[This algorithm inserts term into a ordered linked list.]
Step 1 : newnode = Allocate memory using malloc() function
Step 2 : Set newnode–>powerx = x; [Assigned value of the information part of a new node]
Set newnode–>powery = y;
Set newnode–>powerz = z;
Set newnode–>coef = c;
Step 3 : loc = FINDLOC(start, x, y, z); [returns the address of the node succeeding x]
Step 4 : if loc = NULL then [Inserts as first node]
Set newnode–>next = start; and start = newnode;
else [Inserts after node with address loc]
Set newnode–>next = loc–>next; and loc–>next = newnode;
[End of If statement]
Step 5 : return(start);

The algorithm considers the case of a null or empty list first of all. That is, when location loc is NULL then insert the new node as the first node of the list and return the new node address. It also considers the case when new node is preceding the first node in the original list. The new node becomes the first node in the updated list. The next pointer field of the new node is assigned a pointer value which, corresponds to the location of the first node of the original list. In other cases, the next pointer field of the new node is assigned the address of the node indicated by next pointer of the location loc and the next pointer field of the loc is assigned a pointer value of new node.

The average time for such an insertion is O(n/2).

The complete 'C' program with functions is given below:

/* poly3ins.cpp */

/* This program inserts a polynomial in three variables list. The polynomial $x^2y^2z^2 + x^3y^2 + z^2$ is inserted. The function pinsert constructs the linked representations of these polynomials. Each term consists of four inputs, the first three correspond to the powers of x, y, and z, and the fourth to the coefficient of the term. This function invokes a findloc function which return the location of the new term. */


```
#include<stdio.h>
#include<malloc.h>
struct  nodetype{
    int powerx;
    int  powery;
    int powerz;
```

```
    int coef;
    struct nodetype *next;
}*p, *newnode;
struct nodetype *pinsert(struct nodetype *, int, int, int, int);
struct nodetype *findloc(struct nodetype *, int, int, int);
void display(struct nodetype *);

void main()
{
    p = NULL;          /* Initially polynomial p list is empty */
    p=pinsert(p,2, 2, 2, 1);
    p=pinsert(p,3, 2, 0,1);
    p=pinsert(p,0 ,0, 2, 1);
    printf("\n Polynomial P ");
    display(p);
}  /* End of the main function */

/* function pinsert term into a ordered linked list */
struct nodetype *pinsert(struct nodetype *start, int x,int y, int z,int c)
{
    struct nodetype *loc;
    newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
    newnode–>powerx = x;
    newnode–>powery = y;
    newnode–>powerz = z;
    newnode–>coef = c;
    loc = findloc(start, x, y, z);
    if (loc == NULL)
    {
            newnode–>next = start;
            start = newnode;
    }
    else
    {
            newnode–>next = loc–>next;
            loc–>next = newnode;
    }
    return (start);
}/* end of the pinsert function*/
/* function findloc return the address of the node succeeding*/
struct nodetype *findloc(struct nodetype *start, int x, int y, int z)
{
    struct nodetype *ptr;
    int a, b, c;
```

```
    if (start ==NULL)
      return (NULL);
    a = start–>powerx;
    b = start–>powery;
    c = start–>powerz;
    if ((x >a) || ((x ==a) && (y >b)) || ((x ==a) && (y == b) && (z >c)))
            return (NULL);
    ptr     = start;
    while (ptr–>next! = NULL)
    {
            a = ptr–>next–>powerx;
            b = ptr–>next–>powery;
            c = ptr–>next–>powerz;
if ((x <a) || ((x ==a) && (y <b)) || ((x ==a) && (y == b) && (z <c)))
                    ptr = ptr–>next;
            else
                    break;
    }
    return (ptr);
}   /* End of the findloc function */

/* function display display each term the polynomial*/
void display(struct nodetype *start)
{
    struct nodetype *ptr;
    int a, b, c, d;
    ptr = start;
    printf("\n powerx powery powerz coef");
    while(ptr! = NULL)
    {
            a = ptr–>powerx;
            b = ptr–>powery;
            c = ptr–>powerz;
            d = ptr–>coef;
            printf("\n %d\t %d\t %d\t %d",a, b, c, d );
            ptr = ptr–>next;
    }
}  /* End of the display function */
```

We can easily obtain an ordered linear linked list. For example, the sequence of statements
Start = NULL
Start = PINSERT(start, 3, 0, 0, 2)
Start = PINSERT(start, 1, 1, 0, 3)

Start = PINSERT(start, 0, 2, 0, 1)
Start = PINSERT(start, 0, 1, 1, 1)
creates a four-element list. A trace of this creation is given in Fig. 4.29.

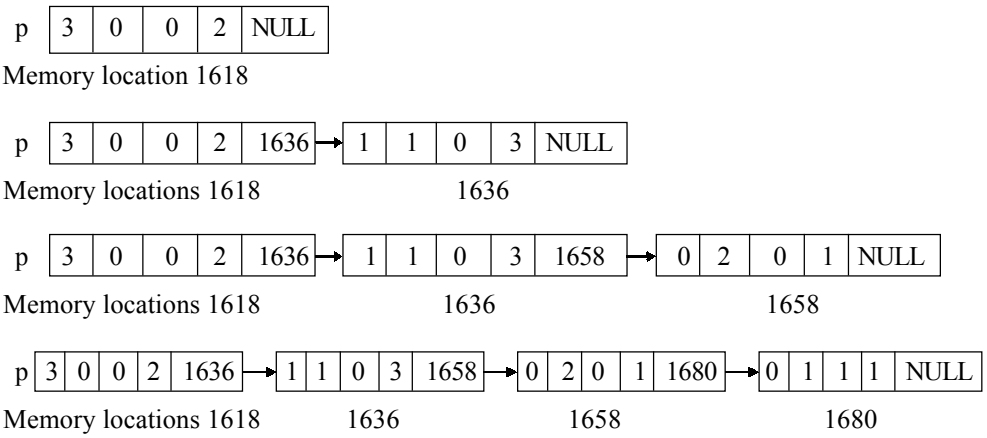


Figure 4.29 *Creation of the ordered linear linked list for a polynomial* $p = 2x^3 + 3xy + y^2 + yz$

Addition of Two Polynomials of Single Variable

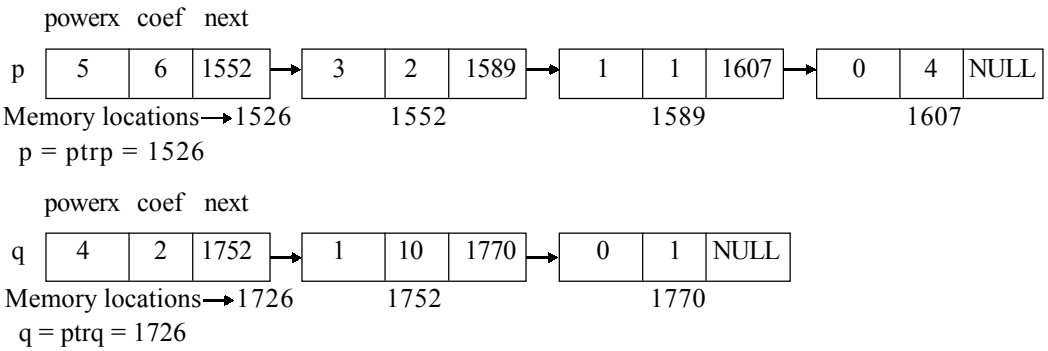
Let us consider the addition of two polynomials of single variable $p = 6x^5 + 2x^3 + x + 4$ and $q = 2x^4 + 10x + 1$. We can visualize this as follows:

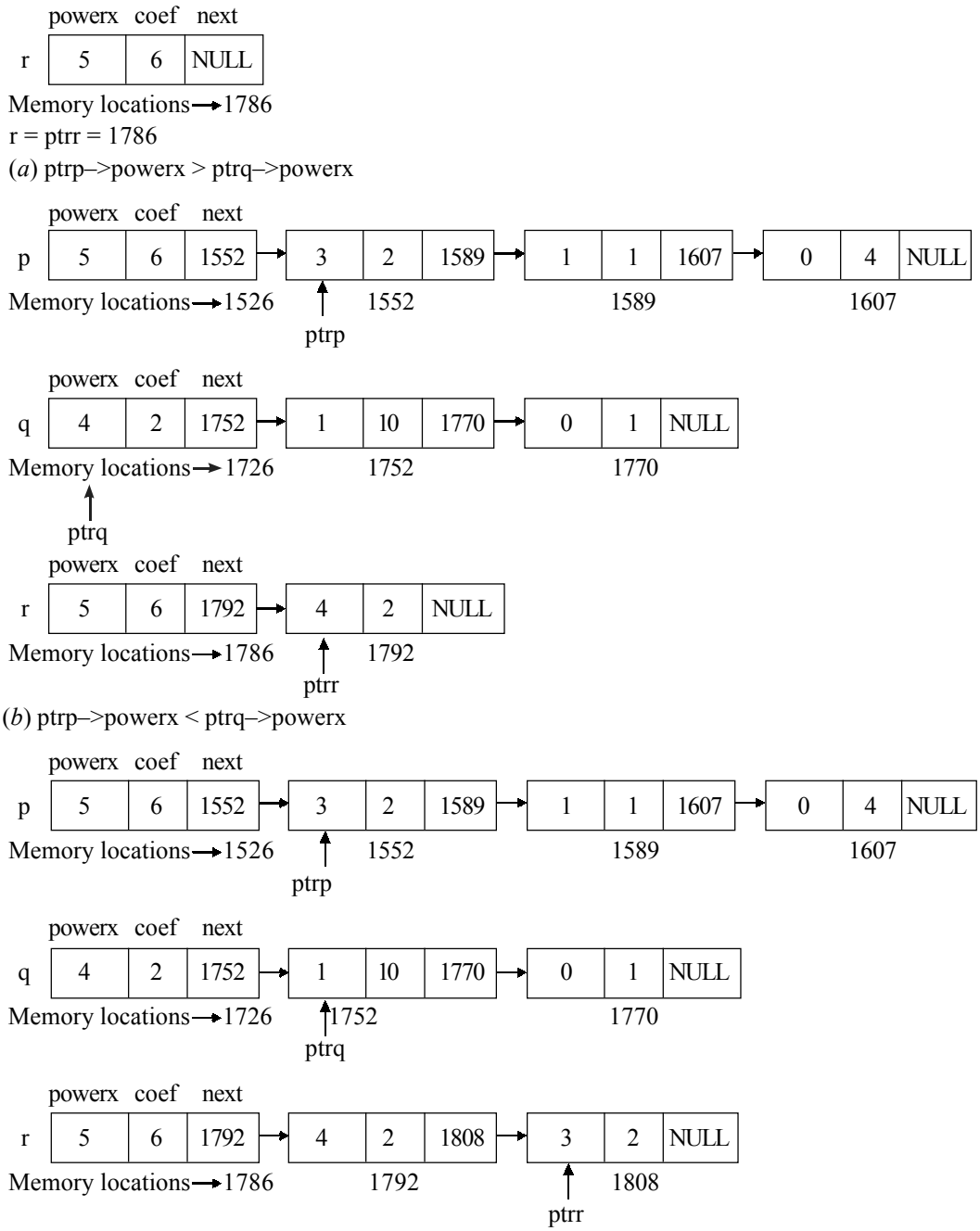
$6x^5 + 2x^3 + x + 4$
$+ \quad 2x^4 + 10x + 1$

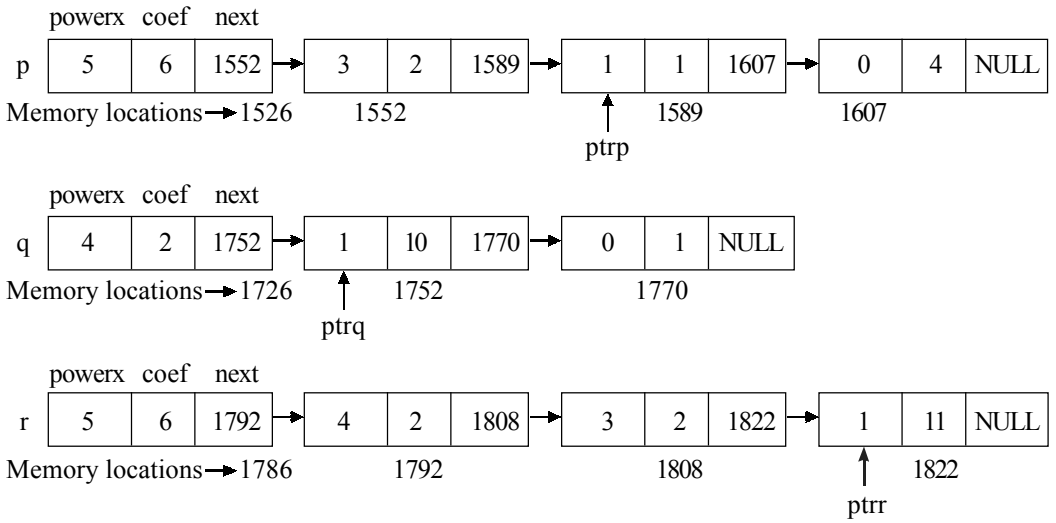
$= 6x^5 + 2x^4 + 2x^3 + 11x + 5$

In order to add two polynomials together we examine their terms starting at the nodes pointed by start pointer variables p and q of the respective polynomials p and q. Two pointers ptrp and ptrq are used to move along the terms of p and q. If the exponents of two terms are equal, then the coefficients are added and if they do not cancel then new term created for the result. If the exponent of the current term in p is less than the exponent of current term of q, then a duplicate of the term of q is created for the result. The pointer ptrq is advanced to the next term. Similar action is taken on p if ptrp–>powerx > ptrq–>powerx.

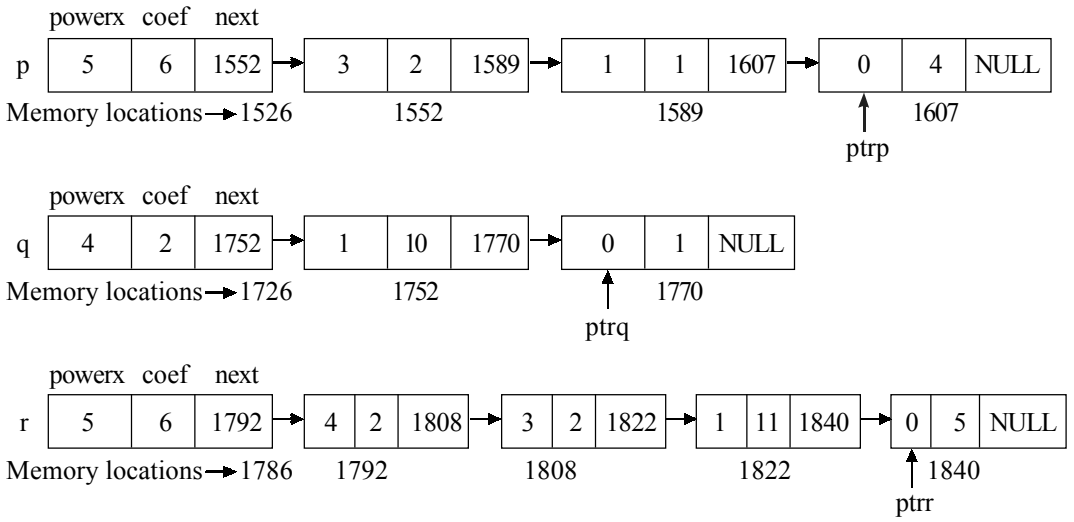
Figure 4.30 illustrates this addition process on the polynomials p and q and resultant polynomial is r = p + q.




powerx coef next
r
5
6
NULL
Memory locations→1786
r = ptrr = 1786
(a) ptrp–>powerx > ptrq–>powerx
powerx coef next
p
5
6
1552
3
2
1589
1
1
1607
0
4
NULL
Memory locations→1526
1552
1589
1607
ptrp
powerx coef next
q
4
2
1752
1
10
1770
0
1
NULL
Memory locations→1726
1752
1770
ptrq
powerx coef next
r
5
6
1792
4
2
NULL
Memory locations→1786
1792
ptrr
(b) ptrp–>powerx < ptrq–>powerx
powerx coef next
p
5
6
1552
3
2
1589
1
1
1607
0
4
NULL
Memory locations→1526
1552
1589
1607
ptrp
powerx coef next
q
4
2
1752
1
10
1770
0
1
NULL
Memory locations→1726
1752
1770
ptrq
powerx coef next
r
5
6
1792
4
2
1808
3
2
NULL
Memory locations→1786
1792
1808
ptrr
(c) ptrp–>powerx > ptrq–>powerx



(*d*) ptrp–>powerx = ptrq–>powerx



(*e*) ptrp–>powerx = ptrq–>powerx

Figure 4.30 *Addition of two polynomials of single variables*

A general algorithm for addition of two polynomials with single variable is given below:

1. Repeat up to step 3 while there are terms in both polynomials yet to be processed
2. Obtain the values for each term
3. If the powers associated with the two terms are equal
   Then if the sum of coefficient of both terms is not zero
   Then insert the sum of the terms into the sum polynomial
   Obtain the next terms in both polynomials to be added
   Else if the power of first polynomial > power of the second
   Then insert the term from first polynomial into sum polynomial

Obtain the next term in the first polynomial
Else insert the term from second polynomial into sum polynomial
Obtain the next term in the second polynomial

4. Copy remaining terms from nonempty polynomial into the sum polynomial
5. Return the address of the sum polynomial

The specific algorithm POLYADD is written in the form of function. This function has as input two ordered linear lists, which represent two polynomials to be added. Each ordered list is constructed by the function PINSERT. The function POLYADD does not alter its input polynomials and the address of the first term of the sum polynomial is returned to the main algorithm, which invokes the function.

Given two polynomials whose first terms are referenced by pointer variables p and q respectively. The third or sum polynomial is represented by pointer variable r. Pointer variables ptrp, ptrq, and ptrr are to keep track polynomials p, q and r respectively.

Algorithm POLYADD

[This algorithm adds two polynomials and returns the address of the first term of the sum polynomial. Consider that the input polynomials are stored in memory.]

```
Step 1 : Set ptrr =  r; ptrp = p ; ptrq = q   ;    [Initially the sum polynomial r is empty, and
                                                    pointer variables ptrp and ptrq denote the first term
                                                    of the polynomials p and q respectively]
Step 2 : repeat step 3 and step 4 while (ptrp !=NULL) and (ptrq !=NULL)
Step 3 :    if  (ptrp–>powerx = ptrq–>powerx)  then
            if  (ptrp–>coef + ptrq–>coef  != 0)  then
                    ptrr  = POLYLAST(ptrr, ptrp–>powerx, ptrp–>coef +ptrq–>coef)
                    Set ptrp = ptrp–>next; and ptrq = ptrq–>next;
            else
Step 4 :    if  (ptrp–>powerx >  ptrq–>powerx)  then
                    ptrr  = POLYLAST (ptrr, ptrp–>powerx, ptrp–>coef)
                    set ptrp = ptrp–>next;
             else
                    r  = POLYLAST (ptrr, ptrq–>powerx, ptrq–>coef)
                    set ptrq = ptrq–>next;
        [End of the loop at step2]
Step 5 : if  (ptrp != NULL) then                    [If  polynomial p does not reach to last term then
                    ptrr = COPY(ptrp, ptrr);         copy all remaining terms into sum polynomial r]
        else                                        [Otherwise copy all remaining terms of
Step 6 :    ptrr = COPY(ptrq, ptrr);                polynomial q into sum polynomial r]
Step 7 :  return (r);                               [Return the address of the first term of the sum
                                                    polynomial]
```

The algorithm POLYADD adds the terms of two polynomials referenced by pointer variables ptrp and ptrq, respectively. This algorithm invokes insertion function POLYLAST and a copying function COPY.

The POLYLAST algorithm is written in the form of function. This function is written for polynomial of single variable. Function POLYLAST(ptrr, x, c) inserts a node at the end of the linked linear list whose address is designated by the pointer ptrr. The global pointer variable r denotes the address of the first node in the list. The fields of the new term are x and c, which correspond to the exponent for x and the coefficient value

of the term, respectively. The newnode is a pointer variable, which contains the address of the new node. The function returns the address of the last node.

Algorithm POLYLAST(ptrr, x, c)

Step 1 : newnode = Allocate memory using malloc() function

Step 2 : Set newnode–>powerx = x; [Assigned values for the term in new node]
Set newnode–>coef = c;

Step 3 : if (ptrr = NULL) then [If the list is empty then add newnode as a first term
Set r = newnode; and set the address of newnode in pointer variable r]
else
Set ptrr–>next = newnode; [otherwise add the new term as the last one]

Set newnode–>next = NULL; [this denotes the end of the list]

Step 4 : return (newnode); [return the last node address]

The COPY algorithm is written in the form of function. Function COPY(ptrp, ptrr) makes a copy of the terms denoted by the pointer variable ptrp to end of the list into the list denoted by the pointer variable ptrr by invoking function POLYLAST. The COPY function returns the address of the last node of the sum polynomial i.e., pointer variable ptrr.

Algorithm COPY(ptrp, ptrr)

Step 1 : repeat step 2 while (ptrp != NULL)

Step 2 : ptrr = POLYLAST(ptrr, ptrp–>powerx, ptrp–>coef); [call POLYLAST function to
ptrp = ptrp–>next; add new term at the end of the list]
[End of the loop at step1]

Step 3 : return (ptrr); [return the last node address]

A complete 'C' program to add two polynomials of single variable is given below:

/* This program adds two polynomials P and Q, and resultant polynomial is R. The polynomials P, Q, and R in one variable are represented by the linked linear lists p, q, and r. The pinsert function inserts polynomials p and q terms in ordered manner and this function invokes the findloc function to find the location of the new term. The function polyadd adds two polynomials p and q by invoking function polylast and copy.

LINKLIST/POLYADD.CPP

P = $3x^4 + 2x^3 + 2$ and Q = $2x^6 - 2x^3 + 2x$*/

```
#include<stdio.h>
#include<malloc.h>
#include<stdlib.h>
struct nodetype{
    int powerx;
    int coef;
    struct nodetype *next;
    }*p,*q,*r, *newnode;
    struct nodetype *pinsert(struct nodetype *, int, int);
    struct nodetype *findloc(struct nodetype *, int);
```

```
    struct nodetype *polyadd(struct nodetype *, struct nodetype *);
    struct nodetype *polylast(struct nodetype *, int, int);
    struct nodetype *copy(struct nodetype *, struct nodetype *);
    void display(struct nodetype *);
void main( )
{
    p = NULL;          /* Initially polynomial p is empty */
    q = NULL;          /* Initially polynomial q  is empty */
    r = NULL;          /* Initially sum polynomial r is empty */
    p = pinsert(p, 4, 3);
    p = pinsert(p, 3, 2);
    p = pinsert(p, 0 , 2);
    q = pinsert(q, 3, –2);
    q = pinsert(q, 1, 2);
    q = pinsert(q, 6, 2);
    r = polyadd(p, q);
    printf("\n Polynomial P ");
    display(p);
    printf("\n Polynomial Q ");
    display(q);
    printf("\n Polynomial R = P + Q ");
    display(r);
    getch( );
}  /* End of main function */

struct nodetype *pinsert(struct nodetype *start, int x, int c)
{
    struct nodetype *loc;
    newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
    newnode–>powerx = x;
    newnode–>coef = c;
    loc = findloc(start, x);
    if (loc == NULL)
    {
            newnode–>next = start;
            start = newnode;
    }
    else
    {
            newnode–>next = loc–>next;
            loc–>next = newnode;
    }
```

```
return (start);
}/* end of the pinsert function*/

struct nodetype *findloc(struct nodetype *start, int x)
{
    struct nodetype *prevptr, *ptr;
    if (start == NULL)
      return (NULL);
    if (x >start–>powerx)
            return (NULL);
    prevptr = start;
    ptr = start–>next;
    while (ptr! = NULL)
    {
            if (x > ptr–>powerx)
                    return (prevptr);
            prevptr = ptr;
            ptr = ptr–>next;
    }
return (prevptr);
} /* End of findloc function */

struct nodetype *polyadd(struct nodetype *p, struct nodetype *q)
{
    struct nodetype *ptrp, *ptrq, *ptrr;
    ptrr = r;
    ptrp = p;
    ptrq = q;

    while ((ptrp !=NULL) && (ptrq !=NULL))
    {
            if (ptrp–>powerx == ptrq–>powerx)
            {
                    if ((ptrp–>coef + ptrq–>coef) != 0)
                            ptrr = polylast(ptrr, ptrp–>powerx, ptrp–>coef + ptrq–>coef);
                    ptrp = ptrp–>next;
                    ptrq = ptrq–>next;

            }
            else if (ptrp–>powerx > ptrq–>powerx)
                    {
                    ptrr = polylast(ptrr, ptrp–>powerx, ptrp–>coef);
```

```
                        ptrp = ptrp–>next;
                        }
                        else
                        {
                        ptrr = polylast(ptrr, ptrq–>powerx, ptrq–>coef);
                        ptrq = ptrq–>next;
                        }
    }
    if (ptrp != NULL)
            ptrr = copy (ptrp, ptrr);
    else
            ptrr = copy (ptrq, ptrr);
return (r);
} /* End of polyadd function */

struct nodetype *polylast(struct nodetype *ptrr, int x, int c)
{
    newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
    newnode–>powerx = x;
    newnode–>coef = c;
    if (ptrr ==NULL)
            r = newnode;
    else
            ptrr–>next =newnode;

    newnode–>next = NULL;
return (newnode);
} /* End of polylast function */

struct nodetype *copy(struct nodetype *ptr, struct nodetype *ptrr)
{
            while (ptr !=NULL)
            {
                    ptrr = polylast(ptrr, ptr–>powerx, ptr–>coef);
                    ptr  = ptr–>next;
            }
return (ptrr);
} /* End of the copy function */

void display(struct nodetype *start)
{
    struct nodetype *ptr;
```

```
    ptr=start;
    printf("\n powerx coef");
    while(ptr!=NULL)
    {
            printf("\n %d\t %d", ptr–>powerx, ptr–>coef );
            ptr = ptr–>next;
    }
}   /* End of the display function */
```

Addition of Two Polynomials of Three Variables

Let us consider the addition of two polynomials of three variables $p = 2x^3 + 2xy + y^2 + z$ and $q = 2x^2 – y^2 + 2yz$. We can visualize this as follows

$$\begin{array}{l} 2x^3 + 2xy + y^2 + z \\ \quad + 2x^2 – y^2 + 2yz \\ \hline \quad 2x^3 + 2x^2 + 2xy + 2yz + z \end{array}$$

The specific algorithm POLY3ADD is written in the form of function. This algorithm is similar to POLYADD, except it adds polynomials of three variables. This function has as input two ordered linear lists, which represent two polynomials to be added. Each ordered list is constructed by the function P3INSERT. The function POLY3ADD does not alter its input polynomials and the address of the first term of the sum polynomial is returned to the main algorithm, which invokes the function.

Given two polynomials whose first terms are referenced by pointer variable p and q respectively. The third or sum polynomial is represent by pointer variable r. Pointer variables ptrp, ptrq, and ptrr are to keep track polynomials p, q and r respectively.

Algorithm POLY3ADD

[This algorithm adds two polynomials and returns the address of the first term of the sum polynomial. Consider that the input polynomials are stored in memory.]

Step 1 : Set ptrr = r; ptrp = p ; ptrq = q ; [Initially the sum polynomial r is empty, and pointer variables ptrp and ptrq denote the first term of the polynomials p and q respectively]

Step 2 : repeat step 3 to step 5 while (ptrp !=NULL) and (ptrq !=NULL)

Step 3 : Set a1 =ptrp–>powerx;
Set a2 =ptrq–>powerx;
Set b1 =ptrp–>powery;
Set b2 =ptrq–>powery;
Set c1 =ptrp–>powerz;
Set c2 =ptrq–>powerz;
Set d1 =ptrp–>coef;
Set d2 =ptrq–>coef;

Step 4 : if (a1 = a2) and (b1 = b2) abd (c1 = c2) then
if (d1 + d2 != 0) then
ptrr = POLYLAST(ptrr, a1, b1, c1, d1+d2)
Set ptrp = ptrp–>next; and ptrq = ptrq–>next;
else

```
Step 5 : if (a1 > a2) or ((a1 =a2) and (b1>b2)) or ((a1 =a2) and (b1 =b2) and (c1>c2)) then
                ptrr = POLYLAST (ptrr, a1, b1, c1, d1)
         set ptrp = ptrp–>next;
         else
                r = POLYLAST (ptrr, a2, b2, c2, d2)
                set ptrq = ptrq–>next;
       [End of the loop at step2]
Step 6 : if (ptrp != NULL) then              [If polynomial p does not reach to last term then
           ptrr = COPY(ptrp, ptrr);  copy all remaining terms into sum polynomial r]
       else                                  [Otherwise copy all remaining terms of
Step 6 :   ptrr = COPY(ptrq, ptrr); polynomial q into sum polynomial r]
Step 7 :  return (r);                        [Return the address of the first term of the sum
                                              polynomial]
```

The algorithm POLY3ADD adds the terms of two polynomials referenced by pointer variable ptrp and ptrq, respectively. This algorithm invokes insertion function POLYLAST and a copying function COPY.

The POLYLAST algorithm is written in the form of function. This function is written for polynomial of three variables. The function has five inputs, one is sum polynomial list pointer variable ptrr, the remaining four are denoted by x, y, z and c, which correspond to the exponents for x, y, z and coefficient of the term. Function POLYLAST(ptrr, x, y, z, c) inserts a node at the end of the linked linear list whose address is designated by the pointer ptrr. The global pointer variable r denotes the address of the first node in the list. The fields of the new term are x and c, which correspond to the exponent for x and the coefficient value of the term, respectively. The newnode is a pointer variable, which contains the address of the new node. The function returns the address of the last node.

```
       Algorithm POLYLAST(ptrr, x, y, z, c)
Step 1 : newnode = Allocate memory using malloc() function
Step 2 : Set newnode–>powerx  = x;          [Assigned values for the term in new node]
         Set newnode–>powerx  = y;
         Set newnode–>powerx  = z;
         Set newnode–>coef    = c;
Step 3 : if (ptrr = NULL) then              [If the list is empty then add newnode as a first term
            Set r = newnode;                 and set the address of newnode in pointer variable r]
         else
           Set ptrr–>next = newnode;        [otherwise add the new term as the last one]
         Set newnode–>next = NULL;[this denotes the end of the list]
Step 4 : return (newnode);                  [return the last node address]
```

The COPY algorithm is written in the form of function. Function COPY(ptrp, ptrr) makes a copy of the terms denoted by the pointer variable ptrp to end of the list into the list denoted by the pointer variable ptrr by invoking the function POLYLAST for the three variables polynomial. The COPY function returns the address of the last node of the sum polynomial i.e. pointer variable ptrr.

Algorithm COPY(ptrp, ptrr)

Step 1 : repeat step 2 while (ptrp != NULL)

Step 2 : ptrr = POLYLAST(ptrr, ptrp–>powerx, ptrp–>powery, ptrp–>powerz, ptrp–>coef);
ptrp = ptrp–>next; [call POLYLAST function to add new term at the end of the list]
[End of the loop at step1]

Step 3 : return (ptrr); [return the last node address]

The complete 'C' program to add three variables polynomial is given below:

/* This program adds two polynomials P and Q, and resultant polynomial is R. The polynomalis P, Q, and R in three variables are represented by the linked linear lists p, q, and r. The pinsert function inserts polynomial p and q terms in ordered manner and this function invokes the findloc function to find the location of the new term. The function poly3add adds two polynomials p and q by invoking functions polylast and copy.

$P = x^3y^2 + x^2 + z^2$ and $Q = -x^3y^2 + xz^3 + y^3z^3$*/


```
#include<stdio.h>
#include<malloc.h>
struct nodetype{
    int powerx;
    int powery;
    int powerz;
    int coef;
    struct nodetype *next;
}*p,*q,*r, *newnode;
struct nodetype *pinsert(struct nodetype *, int, int, int, int);
struct nodetype *findloc(struct nodetype *, int, int, int);
struct nodetype *poly3add(struct nodetype *, struct nodetype *);
struct nodetype *polylast(struct nodetype *, int, int, int, int);
struct nodetype *copy(struct nodetype *, struct nodetype *);
void display(struct nodetype *);

void main()
{
    p = NULL;       /* Initially polynomial P is empty */
    q = NULL;       /* Initially polynomial Q is empty */
    r = NULL;       /* Initially sum polynomial R is empty */

    p=pinsert(p, 2, 0, 0, 1);
    p=pinsert(p,3, 2, 0, 1);
    p=pinsert(p, 0 ,0, 2, 1);
    q=pinsert(q, 3, 2 , 0, –1);
    q=pinsert(q, 0, 3, 3,1);
    q=pinsert(q, 1, 0, 3, 1);
```

```
    r = poly3add(p, q);
    printf("\n Polynomial P ");
    display(p);
    printf("\n Polynomial Q ");
    display(q);
    printf("\n Polynomial R = P + Q ");
    display(r);
}  /* End of main function */

struct nodetype *pinsert(struct nodetype *start, int x, int y, int z, int c)
{
    struct nodetype *loc;
    newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
    newnode–>powerx = x;
    newnode–>powery = y;
    newnode–>powerz = z;
    newnode–>coef = c;
    loc = findloc(start, x, y, z);
    if (loc == NULL)
    {
            newnode–>next = start;
            start = newnode;
    }
    else
    {
            newnode–>next = loc–>next;
            loc–>next = newnode;
    }
    return (start);
}/* end of the pinsert function*/

struct nodetype *findloc(struct nodetype *start, int x, int y, int z)
{
    struct nodetype *prevptr, *ptr;
    int a, b, c;
    if (start == NULL)
      return (NULL);
    a = start–>powerx;
    b = start–>powery;
    c = start–>powerz;
    if ((x >a) || ((x == a) && (y >b)) || ((x == a) && (y == b) && (z >c)))
            return (NULL);
```

```
    ptr = start;
    while (ptr–>next!=NULL)
    {
            a = ptr–>next–>powerx;
            b = ptr–>next–>powery;
            c = ptr–>next–>powerz;

            if ((x <a) || ((x == a) && (y <b)) || ((x == a) && (y == b) && (z <c)))
              ptr = ptr–>next;
            else
                    break;
    }
return (ptr);
} /* End of findloc function */

/* function poly3add adds the terms of two polynomials */
struct nodetype *poly3add(struct nodetype *p, struct nodetype *q)
{
    int a1, a2, b1, b2, c1, c2, d1, d2;
    struct nodetype *ptrp, *ptrq, *ptrr;
    ptrr = r;
    ptrp = p;
    ptrq = q;

    while ((ptrp ! = NULL) && (ptrq ! = NULL))
    {
            a1 = ptrp–>powerx;
            a2 = ptrq–>powerx;
            b1 = ptrp–>powery;
            b2 = ptrq–>powery;
            c1 = ptrp–>powerz;
            c2 = ptrq–>powerz;
            d1 = ptrp–>coef;
            d2 = ptrq–>coef;

            if ((a1 == a2) &&(b1 == b2) &&(c1 == c2))
            {
                    if ((d1 +d2) != 0)
                            ptrr = polylast(ptrr, a1, b1, c1, d1 + d2);
                    ptrp = ptrp–>next;
                    ptrq = ptrq–>next;
            }
            else if ((a1>a2) || ((a1 == a2)&&(b1>b2)) || ((a1 == a2)&&(b1 == b2)&&(c1>c2)))
```

```
                    {
                    ptrr = polylast(ptrr, a1, b1, c1, d1);
                    ptrp = ptrp–>next;
                    }
                    else
                    {
                    ptrr = polylast(ptrr, a2, b2, c2, d2);
                    ptrq = ptrq–>next;
                    }
    }
    if (ptrp != NULL)
            ptrr = copy (ptrp, ptrr);
    else
            ptrr = copy (ptrq, ptrr);
    return (r);
}   /* end of the poly3add function */

struct nodetype *polylast(struct nodetype *ptrr, int x, int y, int z, int c)
{
    newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
    newnode–>powerx = x;
    newnode–>powery = y;
    newnode–>powerz = z;
    newnode–>coef = c;

    if (ptrr ==NULL)
            r = newnode;
    else
            ptrr–>next =newnode;

    newnode–>next = NULL;
    return (newnode);
}/* end of the polylast function*/

struct nodetype *copy(struct nodetype *ptr, struct nodetype *ptrr)
{
    int a, b, c, d;
    a = ptr–>powerx;
    b = ptr–>powery;
    c = ptr–>powerz;
    d = ptr–>coef;
    while (ptr ! = NULL)
```

```
        {
                ptrr = polylast(ptrr, a, b, c, d);
                ptr  = ptr–>next;
        }
        return (ptrr);
    }   /* end of the copy function*/
    void display(struct nodetype *start)
    {
        struct nodetype *ptr;
        int a, b, c, d;
        ptr=start;
        printf("\n powerx powery powerz coef");
        while(ptr!=NULL)
        {
                a = ptr–>powerx;
                b = ptr–>powery;
                c = ptr–>powerz;
                d = ptr–>coef;
                printf("\n %d\t %d\t %d\t %d",a, b, c, d );
                ptr = ptr–>next;
        }
    }   /*end of the display function */
```

Multiplication of Polynomials

The Multiplication of Polynomials is performed by multiplying coefficient and adding the respective power.

4.4.2 Multiple–Precision Arithmetic

Let us consider the problem of performing multiple–precision integer arithmetic. Most digital computers have a certain maximum number of bits or digits for representation of an integer. The number of bits varies from a minimum of 8 to a maximum of 64 in case of 'C' language. The upper bound representing an integer of approximately 10 decimal digits. In certain applications, the size of the integers used may be considerably greater than this upper bound. If we want to perform arithmetic operations such as addition, subtraction or multiplication on these integers, we must find a new way to express them.

Let x–1 be the largest integer that can be stored in a particular computer. Then it is possible to express any integer, say A, using a polynomial expansion of the form:

$$A = \sum_{0 \le i \le k} a_i x^i \qquad \text{where } 0 \le |a_i| < x \text{ for every } i$$

where $a_i = A \bmod x$, and $A = A / x$ until $A = 0$

For example, assume that the largest integer that can be stored in a particular computer is 99999 i.e., x–1, then x would be 100000. An integer A having a value of 1,23456,78910 can be represented in the above form. First of all find the coefficients a_i until A= 0.

$a_0 = A \bmod 100000 = 78910$

and $A = A / 100000 = 123456$

$a_1 = A \bmod 100000 = 23456$
and $A = A / 100000 = 1$
$a_2 = A \bmod 100000 = 1$
and $A = A / 100000 = 0$
$A = a_0 * 100000^0 + a_1 * 100000^1 + a_2 * 100000^2$
$A = 78910 * 100000^0 + 23456 * 100000^1 + 1 * 100000^2$
$= 78910 + 2345600000 + 10000000000$
$= 12345678910$

In this example k =2, a_0 = 78910, a_1 =23456, and a_2 =1. If A is negative integer, then each nonzero a_i will be negative. If A is –10000054321, then it can be represented as

$A = (-54321 * 100000^0) + (0 * 100000^1) + (-1 * 100000^2)$
$= -54321 - 10000000000$
$= -10000054321$

In this section we consider the operation of addition, and we discuss an algorithm for the addition of signed multiple–precision integers. For the algorithm, we require that the above polynomials be represented as linear linked lists.

Consider two polynomials representing integers

$$A = \sum_{0 \le i \le k} a_i x^i \quad \text{where } 0 \le |a_i| < x \text{ for every } i$$

and

$$B = \sum_{0 \le i \le k} b_i x^i \quad \text{where } 0 \le |b_i| < x \text{ for every } i$$

It is required to obtain a polynomial representing the sum

$$S = A + B = \sum_{0 \le i \le k+1} s_i x^i \quad \text{where } 0 \le |b_i| < x \text{ for every } i$$

Note that k is the exponent of the highest–degree term in either polynomial A or B that is nonzero. Thus the exponent of the highest–degree term in sum polynomial would be k+1. In some case, some of the coefficients s_i of S may be zero. As an extreme example, if A is equal to –B or vice-versa, then all coefficients of the polynomial are zero.

Some of the cases are discussed here in adding of two multiple–precision integers.

- If A=0 or B=0, then S is set equal to the number which is nonzero. If A = B = 0, then S=0
- Otherwise $s_i = (a_i + b_i + c_{i-1}) \bmod x$ for $0 \le i \le k$, and initially $c_{-1} = 0$
  and $c_i = (a_i + b_i + c_{i-1}) / x$ for $0 \le i \le k$, and $s_{k+1} = c_k$
  Then $S = \sum_{0 \le i \le k+1} s_i x^i$ where $0 \le |b_i| < x$ for every i, and x =100000

Polynomial Expansion of an Integer

Method

Given the integer A, we can form a single linked list representing its polynomial expansion. Each node in the list consists of a coefficient a_i denoted by coef and a next pointer field denotes the next node in the list. This suggests representing long integers by storing their digits from right to left in a list so that the first node on the list contains the least significant digits (rightmost) and the last node contains the most significant digits (leftmost). However to save space, we keep five digits in each node. Hence long integer variables are used so that numbers as large as 99999 may be kept in each node. The maximum size of an integer is implemented

dependent. In 'C' language we can store the maximum integer value 2147483647 and minimum integer value –2147483646.

We may declare the set of nodes by

```
struct nodetype {
        long int coef;
        struct nodetype *next;
};
```

A list representing the number A = 10111213141516 is given in Figure 4.31.

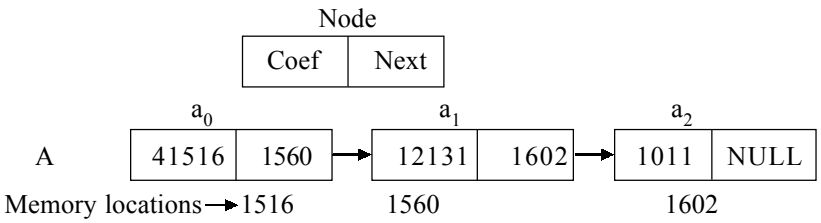


Figure 4.31 *List representation of a multiple–precision integer A = 10111213141516*

The algorithm INSLAST inserts the node coefficient coef at the end of the list.

Algorithm INSLAST(start, s)

Step 1 : create a newnode ;
Step 2 : set newnode–>info = s ; and newnode–>next = NULL;
Step 3 : if (start = NULL) then
Set start = newnode; and return(start);
Step 4 : else
start–>next = newnode; and return(newnode);
Step 5 : end INSLAST

The algorithm ADDINT adds two multiple–precision integers and returns the address of the first node of the resultant integer.

Algorithm ADDINT(sp, sq)

Step 1 : [Initialize the variables]
MAX = 100000L;
ptrp = sp;
ptrq = sq;
ptrs = ss;
carry = 0;
Step 2 : repeat steps up to 5 while ((ptrp ! = NULL) && (ptrq ! = NULL))
Step 3 : total = ptrp–>info + ptrq–>info + carry;
carry = total /MAX;
ptrs = inslast(ptrs, total%MAX);
Step 4 : if ((ptrp = sp) && (ptrq = sq)) then
ss = ptrs;
Step 5 : ptrp = ptrp–>next;
ptrq = ptrq–>next;

```
            [End of while loop at step2]
Step 6 :    if (ptrp !=NULL)then
               ptrq = ptrp;
Step 7 :    if (ptrq !=NULL) then
Step 8 :       repeat step9 while(ptrq!=NULL)
Step 9 :       total = ptrq–>info + carry;
               carry = total / MAX;
               ptrs = inslast(ptrs, total%MAX);
                        ptrq = ptrq–>next;
               [End of while loop at step8]
            [End of If Structure]
Step 10 :   if (carry !=0)
               ptrs = inslast(ptrs, carry);
Step 11 :   return (ss);
```

A complete 'C program for multiple arithmetic precision is given below:


```
/* Program mparlist.cpp for multiple–precision arithmetic. It adds two larger
integers of 10's of digit using linked list structure. */
#include<stdio.h>
#include<malloc.h>
#include<math.h>
#define MAX 100000
struct nodetype{
long int info;
struct nodetype *next;
}*sp,*sq,*ss;
long long int item=0;
struct nodetype *insmpint(struct nodetype *);
struct nodetype *inslast(struct nodetype *, long int);
struct nodetype *addint(struct nodetype *, struct nodetype *);
void display(struct nodetype *);
void main()
{
    struct nodetype *p, *q;
    clrscr();
    p = sp;
    q = sq;
    sp =insmpint(p);
    sq =insmpint(q);
    ss =addint(sp, sq);
    display(ss);
    getch ( );
}
/* function insmpint input multiple precision integers into list */
struct nodetype *insmpint(struct nodetype *ptr)
```

```
{
    struct nodetype *nptr=NULL;
    long int temp;
    printf("\n Enter the multiple precision integer[upto 9 digits] ");
    scanf("%ld",&item);
    while(item !=0)
    {
    temp = item %MAX;
    ptr =inslast(ptr, temp);
    if          (nptr == NULL)
                 nptr = ptr;
    item = item / MAX;
    }
return(nptr);
}
*/ function inslast inserts the node coefficient at the end of the list */
struct nodetype *inslast(struct nodetype *start, long int s)
{
    struct nodetype *newnode;
    newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
    newnode–>info = s ;
    newnode–>next =NULL;
    if (start==NULL)
    {
    start= newnode;
    return(start);
     }
    else
    {
    start–>next =newnode;
    return(newnode);
    }
} /* End of function inslast */
*/ function addint adds two multiple-precision integers */
struct nodetype *addint(struct nodetype *sp, struct nodetype *sq)
{
    long int carry, number, total;
    struct nodetype *ptrp, *ptrq,*ptrs;
    ptrp = sp;
    ptrq = sq;
    ptrs = ss;
    carry = 0;
    while ((ptrp != NULL) && (ptrq !=NULL))
    {
```

```
        total = ptrp–>info + ptrq–>info + carry;
        carry = total /MAX;
        ptrs = inslast(ptrs, total%MAX);
        if ((ptrp == sp) && (ptrq ==sq))
                ss = ptrs;
        ptrp = ptrp–>next;
        ptrq = ptrq–>next;
        }
        if (ptrp !=NULL)
                ptrq = ptrp;
        if (ptrq !=NULL)
        {
        while(ptrq!=NULL)
        {
        total = ptrq–>info + carry;
        carry = total / MAX;
        ptrs = inslast(ptrs, total%MAX);
        ptrq = ptrq–>next;
        }
        }
    if (carry !=0)
        ptrs = inslast(ptrs, carry);
    return (ss);
    }

    void display(struct nodetype *start)
    {
    long long int i=0,sum=0;
    struct nodetype *ptr=start;
    while(ptr!=NULL)
    {
    printf("\n %ld", ptr–>info);
    sum = sum + ptr–>info *pow(MAX,i);
    i++;
    ptr = ptr–>next;
    }
    printf("\n multi precision integer %ld", sum);
    }
```

4.5 SET OPERATION ON LINKED LIST

The set operations on different sets are implemented using linked list.

Consider a set A = { 2, 4, 6, 7 } and set B = {1, 2, 3, 5, 7}

Set A union B = A ∪ B = {1, 2, 3, 4, 5, 6, 7 };

Set A intersection B = A ∩ B = 2, 7}
A difference B = A – B = {4, 6}
B difference A = B – A= {1, 3, 5}

Some of the programs for set operation are given below.

The program for union two sets is given below with needed functions:


```
/* Program unionlst.cpp union two linked list elements */
#include<stdio.h>
#include<malloc.h>
struct nodetype{
    int code;
struct nodetype *next;
}*ptr,*start,*newnode,*start1, *ptr1, *unistart;
int item=0;
struct nodetype *insertion(struct nodetype *, int);
struct nodetype *findloc(struct nodetype *, int);
struct nodetype *createlist(struct nodetype *);
struct nodetype *unilist(struct nodetype *, struct nodetype *);
struct nodetype *copylist(struct nodetype *);
void display(struct nodetype *);
void main()
{
    start = NULL;     /* Initially list is empty */
    start1 = NULL;
    unistart = NULL; /* Initially unistart list is empty */
    do{
    printf("\n Enter element in set A type –1 for termination");
    scanf("%d",&item);
    if (item!= –1)
            start=insertion(start,item);
    }while (item!= –1);
    do{
    printf("\n Enter element in set B type –1 for termination");
    scanf("%d",&item);
    if (item!= –1)
            start1=insertion(start1,item);
    }while (item!= –1);
    printf("\n Element in set A:");
    display(start);
    printf("\n Element in set B:");
```

```
    display(start1);
    unistart = unilist(start, start1);
    printf("\n Union of lists is :");
    display(unistart);
    getch ( );
} /* End of main function */
struct nodetype *insertion(struct nodetype *start, int insitem)
{
    struct nodetype *loc;
    newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
    newnode–>code = insitem;
    loc = findloc(start, insitem);
    if (loc == NULL)
    {
    newnode–>next = start;
    start = newnode;
    }
    else
    {
    newnode–>next = loc–>next;
    loc–>next = newnode;
    }
 return (start);
}/* end of the function*/

struct nodetype *findloc(struct nodetype *start, int insitem)
{
    struct nodetype *prevptr, *ptr;
    if (start == NULL)
      return (NULL);
    if (insitem<start–>code)
            return (NULL);
    prevptr = start;
    ptr = start–>next;
    while (ptr! = NULL)
    {
    if (insitem < ptr–>code)
            return (prevptr);
    prevptr = ptr;
    ptr = ptr–>next;
    }
 return (prevptr);
} /* end of function */
```

```
struct nodetype * unilist(struct nodetype *start, struct nodetype *start1)
{
    struct nodetype *temp;
    if ((start==NULL) && (start1 == NULL))
            printf(" Union of both lists has no item \n");
    else if (start ==NULL)
            unistart = start1;
    else if (start1 == NULL)
            unistart  = start;
    else
    {
    temp = copylist(start);
    unistart = start;
    ptr1= start1;
    while (ptr1 !=NULL)
    {
    ptr = start;
    while ((ptr!=NULL) && (ptr–>code != ptr1–>code))
                    ptr = ptr–>next;
    if (ptr == NULL)
            {
            temp–>next = (struct nodetype *)malloc(sizeof(struct nodetype));
            temp = temp–>next;
            temp–>next = NULL;
            temp–>code = ptr1–>code;
            }
    ptr1 = ptr1–>next;
    }
    }
return (unistart);
} /* end of unilist function */

struct nodetype *copylist(struct nodetype *start)
{
struct nodetype *temp, *uninode;
temp = start;
while (temp–>next! = NULL)
{
            uninode = (struct nodetype *)malloc(sizeof(struct nodetype));
            uninode = temp;
            uninode–>code = temp–>code;
            temp = temp–>next;
}
```

```
return (temp);
}/* end of function */

void display(struct nodetype *start)
{
    ptr=start;
    while(ptr!=NULL)
    {
    printf("\t %d", ptr–>code);
    ptr = ptr–>next;
}
}
```

The program for **merging two lists** using merge function is given below:


```
/* Program mergelst.cpp, merges two sorted list using linear linked list */
#include<stdio.h>
#include<malloc.h>
#include<conio.h>
struct nodetype{
int code;
struct nodetype *next;
}*ptr,*start,*newnode,*start1, *ptr1;
int item=0;
struct nodetype *insertion(struct nodetype *, int);
struct nodetype *findloc(struct nodetype *, int);
struct nodetype *createlist(struct nodetype *);
struct nodetype *merge(struct nodetype *, struct nodetype *);
void display(struct nodetype *);
void main()
{
start = NULL; /* Initially list is empty */
start1 = NULL;
do{
    printf("\n Enter element in List 1 type-1 for termination");
    scanf("%d",&item);
    if (item! = –1);
            start=insertion(start,item);
}while (item! = –1);
do{
    printf("\n Enter element in List 2 type–1 for termination");
    scanf("%d",&item);
    if (item! = –1);
            start1=insertion(start1,item);
```

```
}while (item! = –1);
printf("\n display sort list 1:");
display(start);
printf("\n display sort list 2:");
display(start1);
start = merge(start, start1);
printf("\n display merge list :");
display(start);
getch ( );
}
/* function insertion insert node in ordered list */
struct nodetype *insertion(struct nodetype *start, int insitem)
{
struct nodetype *loc;
newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
newnode–>code = insitem;
loc = findloc(start, insitem);
if (loc == NULL)
{
newnode–>next = start;
start = newnode;
}
else
{
newnode–>next = loc–>next;
loc–>next = newnode;
}
return (start);
}/* end of the function*/
/*function findloc searches the node location in the list*/
struct nodetype *findloc(struct nodetype *start, int insitem)
{
struct nodetype *prevptr, *ptr;
if (start == NULL)
  return (NULL);
if (insitem<start–>code)
     return (NULL);
prevptr = start;
ptr  = start–>next;
while (ptr! = NULL)
{
if (insitem < ptr–>code)
```

```
        return (prevptr);
    prevptr = ptr;
    ptr = ptr–>next;
    }
    return (prevptr);
    }

    /* Merge function merges two lists */
    struct nodetype * merge(struct nodetype *start, struct nodetype *start1)
    {
    if (start == NULL)
        start = start1;
    else
        if (start ! = NULL)
        {
        ptr = start;
        ptr1 = start1;
        if (ptr1–>code <= ptr–>code)
        {
                start1 = start1–>next;
                ptr1–>next = start;
                start  = ptr1;
                ptr = start;
                ptr1 = start1;
        }
        while (start1 ! = NULL)
        {
                while ((ptr–>code < start1–>code) && (ptr! = NULL))
                        {
                        newnode = ptr;
                        ptr = ptr–>next;
                        }
                ptr1 = start1;
                start1 = start1–>next;
                ptr1–>next = newnode–>next;
                newnode–>next = ptr1;
                ptr = ptr1;
        }
      }
    return (start);
    } /* end of merge function */

    void display(struct nodetype *start)
```

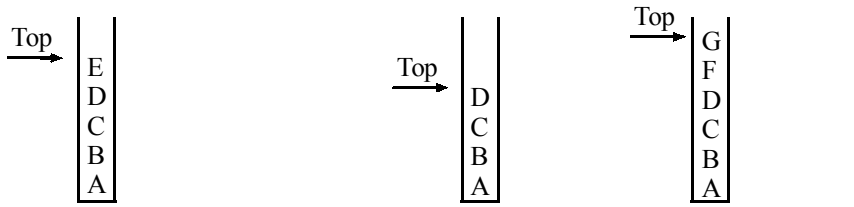
```
{
ptr = start;
while(ptr! = NULL)
{
printf("\n %d", ptr–>code);
ptr = ptr–>next;
}
}
```

Chapter 5

Algorithms on Stack

A stack is an ordered linear list in which all insertions and deletions are made at one end, called the top.

The operations of stack imply that if the elements A, B, C, D, and E are inserted into a stack, in that order, then the first element to be removed (i.e., deleted) must be E. Equivalently, we say that the last element to be inserted into the stack is the first element to be removed. So stack is also called Last In First Out (LIFO) Lists.



(*i*) Insertion of A, B, C, D and E (*ii*) Deletion (*iii*) Insertion of F, G

Figure 5.1 : *Stack*

5.1 REPRESENTATION: USING ARRAY AND LINKED LIST

A stack is an ordered collection of elements. We can declare a variable stack as an array. However, a stack and an array are two entirely different things. The number of elements in an array is fixed and assigned by the declaration for the array. A stack, on the other hand, is fundamentally a dynamic object whose size is constantly changing as items are popped and pushed.

However, although an array cannot be a stack, it can be the home of a stack. That is, an array can be declared large enough for the maximum size of the stack. During the program execution, the stack can grow and shrink within the space reserved for it. One end of the array is fixed bottom of the stack, while the top of the stack constantly shifts as items are popped and pushed.

A stack in 'C' may be declared as a structure containing two objects: an array to hold the elements of the stack, and an integer top to indicate the position of the current stack top within the array. This may be done for stack of integers by the declarations

```
#define MAXSIZE
struct stack {
        int top;
        int items[MAXSIZE];
        };
```

The stack declaration is as

```
struct stack s;
```

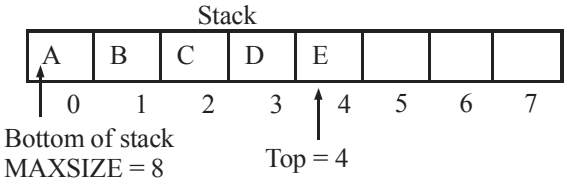
Here, we assume that the elements of the stack s contained in the array s.items[] are integers and maximum elements at the most is MAXSIZE. The stack can contain float, char or any other type of elements. A stack can also contain objects of different types by using 'C' unions.

The identifier top must always be declared as an integer, since its value represents the position within the array items of the topmost stack element.

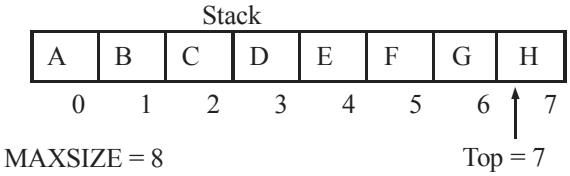
The first or bottom element in the stack is stored at s.items[0], the second at s.items[1] and the ith at s.items[i – 1].

The empty stack contains no elements and can therefore be indicated by top i.e. s.top = –1.

Figure 5.2 shows an array representation of stack (for convenience, the array is drawn horizontally rather than vertically). The size of an array is 8. Initially stack has 5 elements.



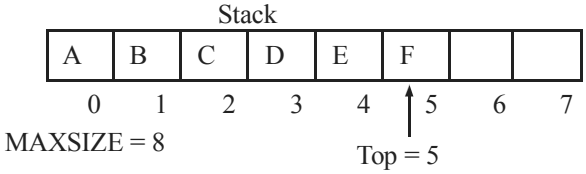
(*i*) Insertion of 3 more elements F, G and H is as given below.



(*ii*) Insert an element I

The element cannot insert as stack is full i.e., top = MAXSIZE – 1.

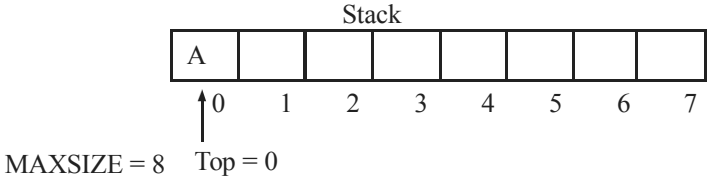
(*iii*) Delete 2 elements



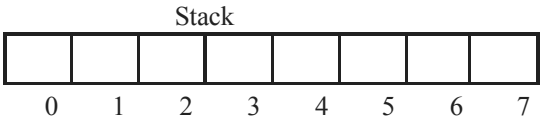
In array implementation actual deletion does not take place. Only top pointer is changed in its position. In the above figure the deleted elements' location is shown empty.

The element of the stack array is displayed from top (i.e. upper bound) to 0 (lower bound) index.

(*iv*) Delete 5 elements



(*v*) Delete one element



MAXSIZE = 8 Top = – 1

Now the top pointer is –1 and stack has become empty.

Figure 5.2 *Stack representation using array*

Linked Representation of Stack

We can represent stacks using one-way or singly linked list. The advantages of linked lists over arrays have already been discussed.

The linked representation of stack is shown in Fig. 5.3. The info fields of nodes hold the elements of the stack and next fields hold pointer to the neighboring element in the stack. The Start pointer of the linked list behaves as the top pointer variable of the stack and NULL pointer of the last node denotes the bottom of stack.

A stack in 'C' may be declared as a structure containing two objects: an item to hold the elements of the stack, and a pointer indicate the position of the next element in the stack. This is similar to single linked list declaration.

This may be done for stack of integers by the declarations

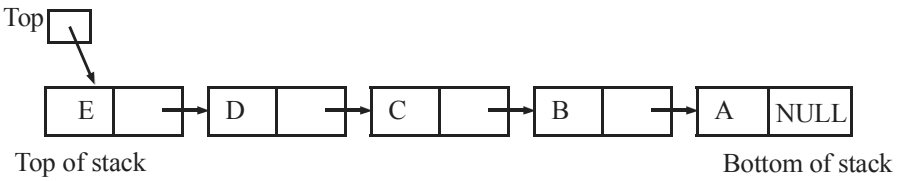
```
struct stack {
        int items;
        struct stack *next;
        };
```

The stack declaration is as

```
struct stack *top;
```

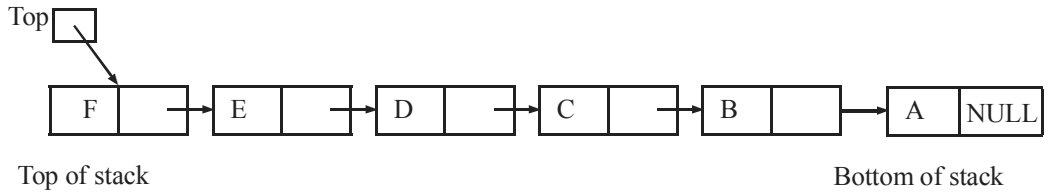
The empty stack contains no element and can be indicated by top =NULL.

Initially linked representation of the stack having five elements A, B, C, D and E are inserted is shown in Fig. 5.3.



An insertion of an element is done at start of the list i.e. top of the stack and deletion operation is undertaken by deleting the node pointed to by the top pointer variable.

(*i*) Stack after insertion of element F.



The condition of stack full depends upon the main memory availability for the linked list at the time of memory allocation for the node by the malloc function.

(*ii*) Stack after deletion of an element

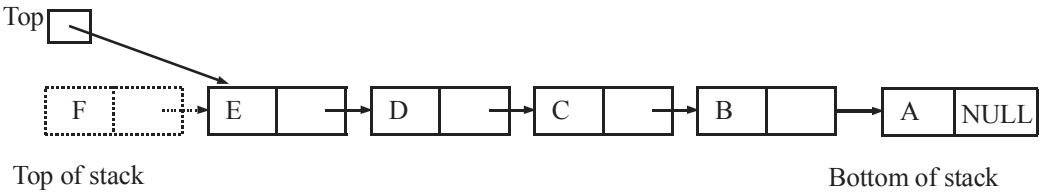


Figure 5.3 *Stack representation using linear linked list.*

The condition of stack empty is when the top pointer variable is NULL.

5.2 PUSH AND POP OPERATIONS

Implementing the Push Operation

The operation of inserting an item onto a stack is called push operation. The simple function to implement push operation using the array representation of a stack. Here, we check whether the array is full before attempting to push element onto the stack. The array is full if s->top = MAXSIZE –1.

The push operation is given below:

```
void push (struct stack *s, int x)
{
        if (s->top == MAXSIZE – 1)
        {
                printf("%s", "stack overflow");
                exit(0);
        }
        else
                s->items[++(s->top)] = x;
        return;
} /* end of push function */
```

Implementing the Pop Operation

The operation of deleting an item onto a stack is called pop operation. The possibility of underflow must be considered in implementing the pop operation.

The pop operation performs two functions.

1. If the stack is empty, print a warning message and halt execution.
2. Remove the top element from the stack and return this element to the calling program.

The pop operation is given below:

```
        int pop (struct stack *s)
        {
```

```
        if (empty(s))
        {
                printf("\n%s", "stack underflow");
                exit(0);
        }
        return(s->items[s->top– –]);
} /* end of pop function */
```

The pop function prints the error message stack underflow and execution halts.
The empty function return true if s->top = –1, otherwise return false.

```
int empty(struct stack *s)
{
        if (s->top == –1)
          return (TRUE);
        else
         return (FALSE);
} /*end of empty function */
```

The underflow condition can be tested during pop operation. The function popandtest pops the stack and returns an indication whether underflow has occurred.

```
void popandtest(struct stack *s, int *p, int *u)
        {
                if (empty(s))
                {
                *p = TRUE;
                return;
                } /* end if */
                *u = FALSE;
                *p =s->items[s->top– –];
                return;
        } /* end of popandtest function */
```

A complete 'C' program to implement these operations using array and linked list is given below:

```
/* The program stackarr.c implement stack operation using structure consist of an array of items */
    #include <stdio.h>
    #include <stdlib.h>
    #define MAXSIZE 80
    #define TRUE 1
    #define FALSE 0
int pop(struct stack *);
void push(struct stack *, int);
void traverse(struct stack *);
int empty(struct stack *);
```

```
struct  stack{
        int top;
        int items[MAXSIZE];
}s;

void main()
{
int c,n;
clrscr();
s.top = –1;
do {
printf("\n Enter choice for Stack ....");
printf("\n 1. Insert node ..");
printf("\n 2. Delete  node ..");
printf("\n 3. Traverse Stack..");
printf("\n 4. Exit Stack..");
printf("\n Enter choice for Stack ....");
scanf("%d",&c);
switch (c)
{
case 1:
        printf("\n Enter items to be push into stack");
        scanf("%d", &n);
        push(&s,n);
        break;
case 2:
        pop(&s);
        break;
case 3:
        traverse(&s);
        break;
}
}while (c!=4);
}
/* function push the integer element into stack s */
void push(struct stack *s, int d)
{
        if (s->top == MAXSIZE – 1)
        {
                printf("%s", "stack overflow");
                exit(0);
        }
```

```
        else
                s->items[++(s->top)] = d;
        return;
} /* end of push function */

/* function pop the integer element from stack s */
int pop(struct stack *s)
{
        if (empty(s))
        {
                printf("\n%s", "stack underflow");
                exit(0);
        }
        return(s->items[s->top– –]);
} /* end of pop function */

int empty(struct stack *s)
{
        if (s->top = = –1)
                return (TRUE);
        else
      return (FALSE);
} /*end of empty function */

void traverse(struct stack *s)
{
 int i;
 for (i =s->top; i>=0;i– –)
        printf(" \t %d",s->items[i]);
 getch();
}/* end of function */
```

The **linked list implementation** for push operation is same in insert element at first in the linear linked list and pop operation similar to delete first (e.g. deletef function) element from linear linked list.

A complete 'C' program for stack as linear linked list is given below:

```
/* Program stackll.cpp to implement stack operation using linked list*/
        #include<stdio.h>
        #include<conio.h>
        #include<string.h>

void display();
void push();
void pop();

struct stack{
int items;
```

STACK/STACKLL.CPP

```
struct stack *next;
}*top;

void main( )
{
int c;
clrscr( );
top = NULL;

do {
printf("\n Enter choice for Stack ....");
printf("\n 1. Insert node ..");
printf("\n 2. Delete  node ..");
printf("\n 3. Display Stack..");
printf("\n 4. Exit Stack ..");
printf("\n Enter choice for Stack ....");
scanf("%d",&c);
switch (c)
{
case 1:
        push( );
        break;
case 2:
        pop( );
        break;
case 3:
        display( );
        break;
}
}while (c!=4);
}

/*end of main function */

void display( )
{
struct stack *ptr;
ptr= top;
while (ptr !=NULL)
{
printf("\n %d \n",ptr->items);
ptr = ptr–>next;
}
getch( );
}
void push( )
```

```
{
        struct stack   *newnode;
        newnode=(struct stack *) malloc(sizeof(struct stack));
        printf("Enter a new  items");
        scanf("%d",&newnode->items);
        newnode->next=NULL;
        if(top==NULL)
                        top = newnode;
       else
        {
              newnode->next = top;
              top = newnode;
        }
}
void pop( )
{
        struct stack *ptr;
        If (top! = NULL)
         ptr=top;
         top=top->next;
        free(ptr);
        else
printf ("\n stack is empty");
}
```

5.3 REPRESENTATION OF EXPRESSION: INFIX, POSTFIX AND PREFIX

Consider the sum of *A* and *B*. We think of applying the *operator* "+" on operands *A* and *B* and write the sum as *A* + *B*. This particular representation is called infix. There are two alternate notations for expressing the sum of *A* and *B* using symbols *A, B,* and +. These are

+AB Prefix
AB+ Postfix

The prefixes "pre-," "post-," and "in-;' refer to the relative position of the operator with respect to the two operands. In prefix notation the operator precedes the operands, in postfix notation the operator follows the two operands, and in infix notation the operator is between the two operands.

The infix notation is known as **Polish notation**, named after the Polish mathematician Jan Lukasiewicz, refers to the notation in which the operator symbol is placed before its two operands.

The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Accordingly, one never needs parentheses when writing expressions in Polish notation.

The postfix notation is known as **Reverse Polish notation.** Again, we never need parentheses to determine the order of the operations in any arithmetic expression written in reverse Polish notation.

Let us now consider some additional examples.

A + B * C

Consider the following normal arithmetic precedence and associatively law for transformation of expressions.

Operator	Precedence
Parentheses ()	4
Exponent (↑ or ^ or $)	3
Multiplication (*) or Division (/)	2
Addition(+) or Subtraction(–)	1

In case of equal precedence of two operators then expression is solved from left to right.

A + B * C	Infix form (convert into postfix form)
A + BC*	Convert the multiplication as higher precedence
ABC*+	Convert the addition and yield the Reverse Polish notation

The underlined expression denotes the operand

A + B * C	Infix form (convert into prefix form)
A + *BC	Convert the multiplication as higher precedence
+A*BC	Convert the addition and yield the Polish notation

The underlined expression denotes the operand

Example 1. Convert the following infix expression into postfix and prefix expression

((A – (B + C)) * D) ↑ (E + F)

Infix expression	((A – (B + C)) * D) ↑ (E + F)
Convert the innermost parentheses	((A – BC+) * D) ↑ (E + F)
Convert the next innermost parentheses	(ABC + – * D) ↑ (E + F)
Convert the next innermost parentheses	ABC + – D * ↑ (E + F)
Convert the next innermost parentheses	ABC + – D * ↑ EF +
Postfix expression	ABC + – D * EF + ↑

Infix expression	((A – (B + C)) * D) ↑ (E + F)
Convert the innermost parentheses	((A – +BC) * D) ↑ (E + F)
Convert the next innermost parentheses	(– A + BC * D) ↑ (E + F)
Convert the next innermost parentheses	*– A + BC D ↑ (E + F)
Convert the next innermost parentheses	*– A + BC D ↑ + EF
Prefix expression	↑*– A + BC D + EF

Example 2. Convert the following infix expression into postfix and prefix expression

(A + B) * (C –D) $ E * F

Infix expression	(A + B) * (C –D) $ E * F
Convert the innermost parentheses	AB + * (C – D) $ E * F
Convert the inner parentheses	AB + * CD – $ E * F
Convert the exponent $	AB + * CD – E $ * F
Convert the multiplication *	AB + CD – E $ * * F
Convert the multiplication *	AB + CD – E $ * F *
Postfix expression	AB + CD – E $ * F *

Infix expression	(A + B) * (C – D) $ E * F
Convert the innermost parentheses	+AB * (C – D) $ E * F
Convert the inner parentheses	+AB * – CD $ E * F
Convert the exponent $	+AB * $ – CD E * F
Convert the multiplication *	* + AB $ – CD E * F
Convert the multiplication *	* * + AB $ – CD E F
Prefix expression	* * + AB $ – CD E F

We give the following additional examples of converting from infix to postfix.

Infix	Postfix
A + B	AB+
A + B – C	AB+C–
(A + B)*(C – D)	AB+CD–*
A ^ B * C – D + E / F / (G + H)	AB ^ C*D–EF / GH+ /+
((A + B) * C –(D – E)) ^ (F + G)	AB + C* DE– – FG + ^
A – B/ (C * D ↑ E)	ABCDE ↑ * / –

The precedence rules for converting an expression from infix to prefix are identical. The only change from postfix conversion is that the operator is placed before the operands rather than after them.

Infix	Prefix
A + B	+AB
A + B – C	– +ABC
(A + B)*(C – D)	* +AB – CD
A ^ B * C – D + E / F / (G + H)	+ – * ^ ABCD / / EF + GH
((A + B) * C –(D – E)) ^ (F + G)	^ – * + ABC – DE + FG
A – B/ (C * D ↑ E)	– A / B * C ↑ DE

5.4 EVALUATION OF THE POSTFIX EXPRESSION

Method

(*i*) Each operator in a postfix string refers to the previous two operands in the string. (Of course, one of these two operands may itself be the result of applying a previous operator.)

(*ii*) Suppose that each time we read an operand we push it onto a stack. When we reach an operator, its operands will be the top two elements on the stack. We can then pop these two elements, perform the indicated operation on them, and push the result on the stack so that it will be available for use as an operand of the next operator.

The following algorithm evaluates an expression in postfix using this method.

Algorithm EvalPostExp

/* Consider the postfix expression E. Opstack is an array to implement stack, variable opnd1, opnd2 contents the value of operand 1 and operand 2. The variable value consists of intermediate results. */

Step 1 : opstack = the empty stack;
/* scan the input string reading one element at a time into symb */

Step 2 : while (symb !=')') {

```
        symb = next input character;
        if (symb is an operand)
                push(opstack, symb);
        else {
                /* symb is an operator */
        opnd2 = pop(opstack);
        opnd1 = pop(opstack);
                value = result of applying symb to opnd1 and opnd2;
                push(opstack, value);
        } /* end else */
    } /* end while */
```

Step 3 : return(pop(opstack));

Each operand is pushed onto the operand stack opstack as it comes in input postfix expression E. Therefore the maximum size of the stack is the number of operands that appear in the input expression.

Example 3. Let us consider the following arithmetic expression E written in postfix notation:

E = 5 6 2 + * 12 4 / –

E = 5	6	2	+	*	12	4	/	–
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)

The elements of E have been labeled from left to right for easy reference. We show the contents of the stack opstack and the variables symb, opnd1, opnd2 and value after each successive iteration of the loop.

The Table below shows the contents of stack as each element of E is scanned. The final result of E is 37.

Symb	***Opnd1***	***Opnd2***	***Value***	***Opstack***
5				5
6				5, 6
2				5, 6, 2
+	6	2	8	5, 8
*	5	8	40	40
12	5	8	40	40, 12
4	5	8	40	40, 12, 4
/	12	4	3	40, 3
–	40	3	37	37

Example 4. Let us consider the following arithmetic expression E written in postfix notation:

E = 6 2 3 + – 3 8 2 / + * 2 ^ 3 +

E = 6	2	3	+	–	3	8	2	/	+	*	2	^	3	+
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)

The elements of E have been labeled from left to right for easy reference. We show the contents of the stack opstack and the variables symb, opnd1, opnd2 and value after each successive iteration of the loop.

The Table on the next page shows the contents of stack as each element of E is scanned. The final result of E is 52.

Symb	*Opnd1*	*Opnd2*	*Value*	*Opstack*
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
^	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Program for Evaluation of Postfix Expression

(*i*) Let us assume in this case that each input line is in the form of a string of digits and operator symbols. We assume that operands are single nonnegative digits, for example, 0, 1, 2, ..., 8, 9. For example, an input line might contain 3 4 5 * + in the first 5 columns followed by an end-of-line character (‘ \ n’). We would like to write a program that reads input lines of this format, as long as there are any remaining, and prints for each line the original input string and the result of the evaluated expression,

(*ii*) Since the symbols are read as characters, we must find a method to convert the operand characters to numbers and the operator characters to operations. For example, we must have a method for converting the character ‘5’ to the number 5 and the character ‘+’ to the addition operation,

(*iii*) The conversion of a character to an integer can be handled easily in C. If **int** is a single digit character in C, the expression *x* - ‘0’ yields the numerical value of that digit. To implement the operation corresponding to an operator symbol, we use function *oper* that accepts the character representation of an operator and two operands as input parameters, and returns the value of the expression obtained by applying the operator to the two operands.

(*iv*) The body of the function is presented below:

```
double eval(char expr[])
{
        int c, position;
        double opndl, opnd2, value;
        struct stack opndstk;
        opndstk.top = –1;
        for (position = 0; (c = expr[position]) != ‘\0’; position++)
        if (c !=’ ‘)
        {
```

```
        if (isdigit(c))
            /* operand– – convert the character representation
            of the digit into double and push it onto  the stack */
            push(&opndstk, (double) (c–'0'));
        else {
            /* operator */
            opnd2 = pop(&opndstk);
            opnd1 = pop(&opndstk);
            value = oper(c, opnd1, opnd2);
            push(&opndstk, value);
            } /* end else */
          } /*end of outer if */
      return(pop(&opndstk));
} /* end eval */
```

A complete 'C' program is given below. The constant MAXSIZE is the maximum size of the input line.

/* Program evalpost.cpp for evaluating a postfix expression such as 2 3 + gives 5. Here we have consider the numeric value is only digit and allowed operations are addition, subtraction, division, multiplication and exponents. For exponent operation we have used ^(carot) operator.*/


```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXSIZE 80
#define TRUE 1
#define FALSE 0
double eval(char[]);
double pop(struct stack *);
void push(struct stack *, double);
int empty(struct stack *);
int isdigit(char);
double oper(int, double, double);

struct stack{
        int top;
double items[MAXSIZE];
};

void main()
{
        char expr[MAXSIZE];
        int position = 0;
        printf ("\n Enter postfix expression");
        while((expr[position++] = getchar()) != '\n');
```

```
        expr[– –position] = '\0';
        printf("%s is the original postfix expression", expr);
        printf ("\n Result is :")
        printf("\n%f", eval(expr));
        getch( );
    } /* end main */
double eval(char expr[])
{
        int c, position;
        double opndl, opnd2, value;
        struct stack opndstk;
        opndstk.top = –1;
        for (position = 0; (c = expr[position]) != '\0'; position++)
        if (c !=' ')
        {
            if (isdigit(c))
                /* operand– – convert the character representation
                of the digit into double and push it onto  the stack */
                push(&opndstk, (double) (c–'0'));
             else {
                /* operator */
                opnd2 = pop(&opndstk);
                opnd1 = pop(&opndstk);
                value = oper(c, opnd1, opnd2);
                push(&opndstk, value);
                } /* end else */
            } /*end of outer if */
           return(pop(&opndstk));
    } /* end eval */
    int isdigit(char symb)
{
        return(symb >= '0' && symb <= '9');
}
double oper(int symb, double op1, double op2)
{
        switch(symb) {
        case '+' : return (op1 + op2);
        case '–' : return (op1 – op2);
        case '*' : return (op1 * op2);
        case '/' : return (op1 / op2);
```

```
            case '^' : return (pow(op1, op2));
            default: printf("\n%s", "illegal operation");
                return FALSE;
} /* end switch */
} /* end oper */
void push(struct stack *s, double d)
{
            if (s->top == MAXSIZE –1)
            {
                  printf("%s", "stack overflow");
                  exit(0);
            }
            else
                        s->items[++(s->top)] = d;
                  return;
} /* end of push function */
double pop(struct stack *s)
{
                  if (empty(s))
                  {
                        printf("\n%s", "stack underflow");
                        exit(0);
                  }
                  return(s->items[s->top– –]);
} /* end of pop function */
int empty(struct stack *s)
{
                  if (s->top == –1)
                       return (TRUE);
                  else
                      return (FALSE);
} /*end of empty function */
```

5.5 TRANSFORMING INFIX EXPRESSION INTO POSTFIX EXPRESSION

In our previous discussion we mentioned that expressions within innermost parentheses must first be converted to postfix so that they can then be treated as single operands. In this fashion parentheses can be successively eliminated until the entire expression is converted. The last pair of parentheses to be opened within a group of parentheses encloses the first expression within that group to be transformed. This **last-in, first-out** behavior should immediately suggest the use of a stack.

Method

(*i*) Consider the two infix expressions $A + B * C$ and $(A + B) * C$, and their respective postfix versions $ABC * +$ and $AB + C *$. In each case the order of the operands is the same as the order of the operands in the original infix expressions. In scanning the first expression, $A + B * C$,

the first operand, *A,* can be inserted immediately into the postfix expression. Clearly the + symbol cannot be inserted until after its second operand, which has not yet been scanned, is inserted. Therefore, it must be stored away to be retrieved and inserted in its proper position. When the operand *B* is scanned, it is inserted immediately after *A*.

(*ii*) Now, however, two operands have been scanned. What prevents the Symbol + from being retrieved and inserted? The answer is, of course, the * symbol that follows, which has precedence over +. In the case of the second expression the closing parenthesis indicates that the + operation should be performed first. Remember that in postfix, unlike infix, the operator that appears earlier in the string is the one that is applied first.

(*iii*) Since precedence plays such an important role in transforming infix to postfix, let us assume the existence of a *function prcd(op1,op2),* where *op1*and *op2* are characters representing operators. This function returns *TRUE* if *op1* has precedence over *op2* when *op1* appears to the left of *op2* in an infix expression without parentheses. *prcd (op1,op2)* returns *FALSE* otherwise. For example, prcd('*' ,'+') and *prcd* ('+' , '+') are *TRUE,* whereas *prcd*('+' ,'*') is *FALSE.*

The following algorithm Infix to Postfix convert an infix expression into postfix expression.

Algorithm Infix to Postfix

```
Step 1 :  opstack = the empty stack;
Step 2 :  while (not end of input) {
          symb = next input character;
Step 3 :  if (symb is an operand)
              add symb to the postfix string;
          else {
Step 4 :      while (!empty(opstack) &&(prcd(pop(opstack), symb)) {
                  topsymb = pop(opstack);
                  add topsymb to the postfix string;
          }/* end of while loop */
Step 5 :  if (empty (opstack) || symb != ')')
              push(opstack, symb);
          else    /*  pop the open parenthesis and discard it */
              topsymb = pop(opstack);
          }/* end of else of step 3 */
      }/* end of while of step 2 */
      /* output any remaining operators */
Step 6 :  while (!empty(opstack)) {
              topsymb = pop(opstack);
              add topsymb to the postfix string;
      } /* end while */
```

The algorithm can be used to convert any infix string to postfix. We summarize the precedence rules for parentheses:

pred('(' ,op) = FALSE for any operator op
pred(op, '(') = FALSE for any operator output other than ')'

pred(op, ‘)’) = TRUE for any operator output other than ‘(’
pred(‘)’, op) = Undefined for any operator output (an attempt to compare two indicates an error)

Example 1. A + B * C

The contents of symb, the postfix string, and opstack are shown after scanning each symbol. opstack is shown with its top to the right.

Symb	*Postfix string*	*opstack*
A	A	
+	A	+
B	AB	+
*	AB	+*
C	ABC	+*
	ABC*	+
	ABC*+	

Example 2. ((A – (B + C)) * D) ^ (E + F)

The contents of symb, the postfix string, and opstack are shown after scanning each symbol. opstack is shown with its top is the right most symbol.

Symb	*Postfix string*	*opstack*
(		(
(		((
A	A	((
–	A	((–
(	A	((– (
B	AB	((– (
+	AB	((– (+
C	ABC	((– (+
)	ABC +	((–
)	ABC + –	(
*	ABC + –	(*
D	ABC + – D	(*
)	ABC + – D *	
^	ABC + – D *	^
(	ABC + – D *	^ (
E	ABC + – D * E	^ (
+	ABC + – D * E	^ (+
F	ABC + – D * E F	^ (+
)	ABC + – D * EF +	^
	ABC + – D * EF + ^	

Example 3. Consider the following arithmetic infix expression E:

E = A + (B * C – (D / E $ F) * G) * H

The symbols of expression E have now labeled from left to right for easy reference. Table below shows the status of opstack, postfix string and symbol which is scanning.

Symb	*Postfix string*	*opstack*
A	A	
+	A	+
(	A	+ (
B	AB	+ (
*	AB	+ (*
C	A B C	+ (*
–	A B C *	+ (–
(	A B C *	+ (– (
D	A B C * D	+ (– (
/	A B C * D	+ (– (/
E	A B C * D E	+ (– (/
^	A B C * D E	+ (– (/ ^
F	A B C * D E F	+ (– (/ ^
)	A B C * D E F ^ /	+ (–
*	A B C * D E F ^ /	+ (– *
G	A B C * D E F ^ / G	+ (– *
)	A B C * D E F ^ / G * –	+
*	A B C * D E F ^ / G * – H	+ *
H	A B C * D E F ^ / G * – H * –	+ *
	A B C * D E F ^ / G * – H * +	

Program to Transforming Infix Expression into Postfix Expression

The following steps are considered:

(*i*) When a closing parenthesis is read, all operators up to the first opening parenthesis must be popped from the stack into the postfix string. This can be done by defining *prcd(op,* ')') as *TRUE* for all operators *op* other than a left parenthesis.

(*ii*) When these operators have been popped off the stack and the opening parenthesis is uncovered special action must be taken. The opening parenthesis must be popped off the stack and the closing parenthesis discarded rather than placed in the postfix string or on the stack.

(*iii*) Let us set *prcd('(', ')')* to *FALSE*. This ensures that upon reaching an opening parenthesis, the loop beginning at line 6 is skipped, so that the opening parenthesis not inserted into the postfix string.

(*iv*) There are two things that we must do before we actually start writing a program. The first is to define precisely the format of the input and output. The second is to construct, or at least define, those routines that the main routine depends upon. We assume that the input consists of strings of characters, one string per input line. The end of each string is signaled by the occurrence of an end of line character ('\n').

(*v*) For the sake of simplicity, we assume that all operands are single-character letters or digits. All operators and parentheses are represented by themselves, and '^' represents exponentiation. The output is a character string.

In transforming the conversion algorithm into a program, we make use of several routines. Among these are empty, pop, push, isoperand and popandtest, all suitably modified so that the elements on the stack are characters. The function isoperand that returns TRUE if its argument is an operand and FALSE otherwise.

Similarly, the prcd function accepts two single-character operator symbols as arguments and return TRUE if the first has precedence over the second when it appears to the left of the second in an infix string and FALSE otherwise.

The program reads a line containing an expression in infix, calls the routine postfix, and prints the postfix string.

/* Program inpostfix.cpp for converting infix expression into a postfix expression . For example infix 2 + 3 is converting postfix 23 + . Here all the single character operands in the initial infix string are digits. The functions used in the program are empty, pop, push, isoperand, prcd and postfix */


```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXSIZE 80
#define TRUE 1
#define FALSE 0
void postfix(char *, char *);
char pop(struct stack *);
void push(struct stack *, char);
int empty(struct stack *);
int isoperand(char);
int prcd(char, char);
void popandtest(struct stack *, char *, int *);

struct stack{
int top;
double items[MAXSIZE];
};

void main()
{
    char infix[MAXSIZE];
    char postr[MAXSIZE];
    int position = 0;
    printf ("\n Enter Infix expression");
    while((infix[position++] = getchar()) != '\n');
    infix[– –position] = '\0';
    printf("%s%s", "is the original Infix expression", infix);
    postfix(infix,postr);
    printf("\n%s", postr);
} /* end main */
```

```
void postfix(char infix[], char postr[])
{
    int und, position, outpos =0;
    char symb, topsymb = '+';
    struct stack opstack;
    opstack.top = -1;          /* the empty stack */
    for (position = 0; (symb = infix[position]) != '\0'; position++)
       if (isoperand(symb))
         postr[outpos++] = symb;
         else {
              popandtest(&opstack, &topsymb, &und);
              while (!und && prcd(topsymb, symb))
              {
                 postr[outpos++] = topsymb;
                 popandtest(&opstack, &topsymb, &und);
              } /* end of while */
              if (!und)
                 push(&opstack, topsymb);
              if (und ||(symb !=')'))
                 push(&opstack, symb);
               else
                 topsymb =pop(&opstack);
    } /* end else */
    while(!empty(&opstack))
          postr[outpos++] = pop(&opstack);
    postr[outpos] = '\0';
return;
} /* end postfix */
int isoperand(char symb)
{
return(symb >= '0' && symb <= '9');
}
double oper(int symb, double op1, double op2)
{
        switch(symb) {
        case '+' : return (op1 + op2);
        case '-' : return (op1 - op2);
        case '*' : return (op1 * op2);
        case '/' : return (op1 / op2);
        case '^' : return (pow(op1, op2));
        default: printf("\n%s", "illegal operation");
                 return FALSE;
} /* end switch */
} /* end oper */
void push(struct stack *s, char d)
{
```

```
        if (s->top == MAXSIZE – 1)
        {
                printf("%s", "stack overflow");
                exit(0);
        }
        else
                s->items[++(s->top)] = d;
        return;
} /* end of push function */
char pop(struct stack *s)
{
        if (empty(s))
        {
                printf("\n%s", "stack underflow");
                exit(0);
        }
        return(s->items[s->top– –]);
} /* end of pop function */
int empty(struct stack *s)
{
        if (s->top == – 1)
            return (TRUE);
        else
             return (FALSE);
} /*end of empty function */
void popandtest(struct stack *s, char *p, int *u)
{
        if (empty(s)) {
                *u = TRUE;
                return;
        } /*end if*/
        *u = FALSE;
        *p =s->items[s->top– –];
        return;
}/* end popandtest */
int prcd(char x, char y)
{
        if ((x ='+') || (x == '/'))
                    return TRUE;
        If ((( x == '+') || (x == '–')) && (yy == '*') || (y == '/')))
                     return FALSE;
        If (x == y)
                     return TRUE;
        If ((y == '–') || (y == '+'))
                     return TRUE;
}
```

The program has one major flaw in that it does not check that the input string is a valid infix expression.

We can now write a program to read an infix string and compute its numerical value. If the original string consists of single digit operands with no letter operands, the following program reads the original string and prints its value.

```
#define MAXSIZE
void main()
{
        char instring[MAXSIZE], postring[MAXSIZE];
        int position = 0;
        double eval();
        while((instring[position++] = getchar()) !='\n')
        ;
        instring[– –position] = \'0';
        printf("%s%s", " infix expression is", instring);
        postfix(instring, postring);
        printf("%s%f\n", "value is ", eval(postring));
} /* end of main */
```

Functions postfix and eval are same as used in program inpostfix.cpp and evalpost.cpp respectively.

5.6 RECURSION

Example 1. Factorial

The factorial function, whose domain is the natural number, can be defined as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1) * (n-2) * \cdots * 1, & \text{if } n > 0 \end{cases}$$

Let's consider the case of 4! since $n > 0$, we use the second clause of the definition

$4! = 4 * 3 * 2 * 1 = 24$

Also $3! = 3 * 2 * 1 = 6$

$2! = 2 * 1 = 2$

$1! = 1$

$0! = 1$

Such definition is called iterative because it calls for the implicit repetition of some process until a certain condition is met.

Let us look again at the definition of *n!*

$4! = 4 * 3 * 2 * 1 = 4 * 3!$

Also $3! = 3 * 2 * 1 = 3 * 2!$

$2! = 2 * 1 = 2 * 1!$

$1! = 1 = 1 * 0!$

or

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

Such a definition which defines an object in terms of a simpler case of itself, is called a recursive definition. Here, *n!* is defined in terms of *(n* – 1)! which in turn is defined in terms of *(n* – 2)! and so on, until finally 0! is reached.

The basic idea, here, is to define a function for all its argument values in a constructive manner by using induction. The value of a function for a particular argument value can be computed in a finite number of steps using the recursive definition, where at each step of recursion, we come nearer to the solution.

The recursive algorithm to compute n! may be easily converted into a 'C' function as follows:

```
int fact(int n)
{
if (n == 0)
        return (1);
else
        return (n * fact(n – 1));
} /* end of fact function */
```

Let us examine the execution of this function, suppose that the calling program contains the statement

```
printf("%d", fact(4));
```

When the calling routine calls fact, the parameter n is set equal to 4, since n is not 0, fact is called a second time with an argument of 3, again n is not 0, fact is called third time with an argument of 2, and so on until n becomes 0. Then function returns and stack is popped the top allocation. The execution of function is given in Fig. 5.4.

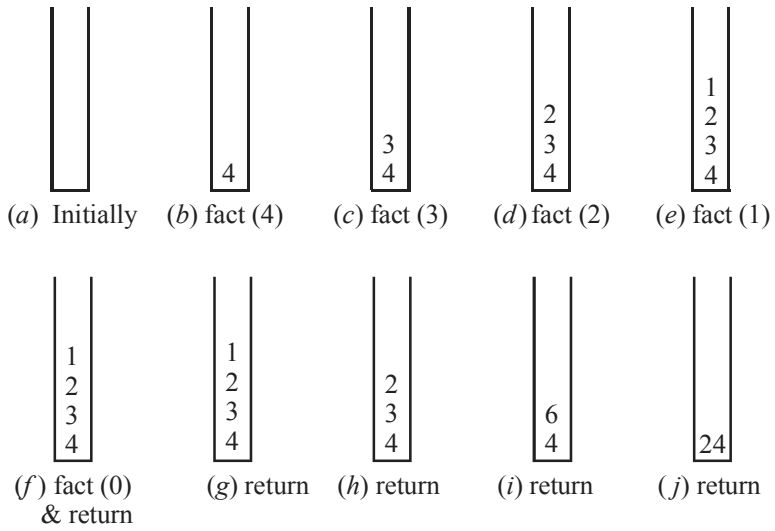


Figure 5.4 *Recursive calls of factorial function*

Example 2. Multiplication of Natural Numbers

Another example of a recursive definition is the definition of multiplication of natural numbers. The product *a * b,* where *a* and *b* are positive numbers, may be written as *a* added to itself *b* times. This definition is iterative. We can write this definition in a recursive way as:

$$a * b = \begin{cases} a & \text{if } b = 1 \\ a * (b - 1) + a & \text{if } b > 1 \end{cases}$$

To evaluate 5 * 3, we first evaluate 5 * 2 and add 5. To evaluate 5 * 2, we first evaluate 5 * 1 and add 5. 5 * 1 is equal to 5 due to first part of the definition.

Therefore

5*3=5*2+5=5*1+5+5=5+5+5=15

This algorithm of multiplication can be easily converted into 'C' function.

```
int multiply(int a, int b)
{
        if ( b ==1)
                return(a);.
        else
                return(multiply(a, b – 1) + a);
}
```

Example 3. The Fibonacci Sequence

The fibonacci sequence is the sequence of integers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...,

Each element, here, is the sum of the two preceding elements.

1 = 0 + 1
2 = 1 + 1
3 = 1 + 2
5 = 2 + 3
8 = 3 + 5

and so on.

We may define the fibonacci sequence by assuming fib(0) = 0, fib(1) =1, as follows:

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

This method can be easily converted into 'C' function.

Function : Fibonacci term

```
int fib(int n)
{
        if (n ==1) || (n ==0)
             return (n);
        a = fib ( n – 1);
         b = fib (n – 2);
  return (a + b);
} /* end of fibo function */
```

We may note that the recursive definition of the fibonacci numbers refers to itself twice so that in computing fib (5) = fib (4) + fib (3), the function is called recursively twice (See Fig. 5.5).

However, the computation of fib (4) again involves determining fib (3), so there is great deal of redundancy in applying this definition. It would be much more efficient to store the intermediate fibonacci values when they are evaluated first time and reuse it each time whenever needed.

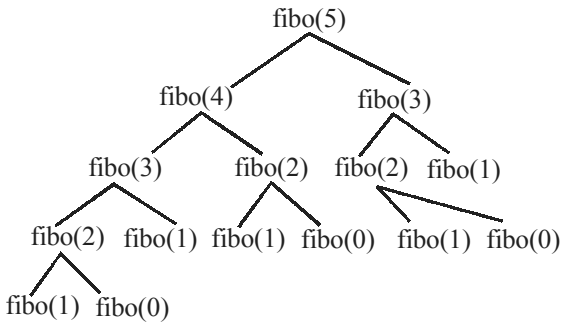


Figure 5.5 *Fibonacci sequence recursive calls for fibo(5)*

The program to find fibonacci term with recursion is quite easy, just call the above recursive function fibo() from the main function is left for exercise.

The program to find the fibonacci term with explicit stack is given below. Here non-recursive function fibos() is written and a stack is maintained explicitly. For stack operation push, pop and empty functions are used.

```
/* The program fibstack.cpp is written in the form of non-recursive form using explicit stack operations
empty, pop and push. It finds the fibonacci term */

#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE 80
#define TRUE 1
#define FALSE 0
int pop(struct stack *);
void push(struct stack *, int);
int empty(struct stack *);
int fibos(int n);
struct stack{
int top;
int items[MAXSIZE];
};

void main()
{
int n;
printf("\n Enter a Fibonacci term number");
scanf("%d",&n);
printf("\n %d term of fibonacci sequence is %d",n, fibos(n–1));
} /* end main */

/*Recursive function fibo() is written in form of stack */
```

```
int fibos(int n)
{
int sum = 0;
struct stack s;
s.top = 0;
while (s.top>=0)
{
        if ((n ==0) || (n==1))
                sum = sum + n;
        else if (n > 1)
        {
                push(&s, n–1);
                push(&s, n–2);
        }
        n = pop(&s);
}
return(sum);
}
/* function push the integer element into stack s */
void push(struct stack *s, int d)
{
        if (s->top = = MAXSIZE – 1)
        {
                printf("%s", "stack overflow");
                exit(0);
        }
        else
                s->items[++(s->top)] = d;
        return;
} /* end of push function */
/* function pop the integer element from stack s */
int pop(struct stack *s)
{
        if (empty(s))
        {
                printf("\n%s", "stack underflow");
                exit(0);
        }
        return(s->items[s->top– –]);
} /* end of pop function */
int empty(struct stack *s)
```

```
{
        if (s->top == – 1)
            return (TRUE);
        else
            return (FALSE);
} /*end of empty function */
```

Example 4. Finding the Greatest Common Divisor
The greatest common divisor of two integers is defined as follows by the Euclid's algorithm

$$GCD(a, b) = \begin{cases} GCD(b, a) & \text{if } (b > a) \\ a & \text{if } (b = 0) \\ GCD(b, \text{mod}(a, b)) & \text{otherwise} \end{cases}$$

Here mod(a, b) is the remainder on dividing a by b. The second case is when the second argument is zero. In this case, the greatest common divisor is equal to the first argument.

If the second argument is greater than the first, the order of argument is interchanged. Finally, the GCD is defined in terms of itself. Note that the size of the argument is getting smaller as mod *(a, b)* will eventually be reduced to zero in a finite number of steps.

A 'C' function for computing GCD can be written as:

Function : Greatest Common Divisor

```
int GCD (int a, int b)
{
int x;
if (b > a)
      return (GCD (b, a));
if (b == 0)
      return(a);
else
    return (GCD (b, (a %b)));
}
```

Consider an example to compute GCD(28, 8).

It calls GCD(8, 28 % 8) i.e. GCD(8, 4)

It calls GCD(4, 8%4) i.e. GCD(4, 0) which returns 4 as greatest common divisor.

Example 5. The Towers of Hanoi Problem
Another complex recursive problem is that of towers of Hanoi. The problem has a historical basis in the ritual of the ancient Tower of Brahma. This is a problem which is not specified in terms of recursion and we have to see how we can use recursion to produce a logical and elegant solution. The problem is as follows:

There are *n* disks of different sizes and there are three needles *A, B* and C. All the n disks are placed on a needle (A) in such a way that a larger disk is always below a smaller disk as shown in Fig. 5.6.

The other two needles are initially empty. The aim is to move the *n* disks to the second needle *(C)* using the third needle *(B)* as a temporary storage. The rules for the movement of disks is as follows:

(*a*) Only the top disk may be moved at a time.

(*b*) A disk may be moved from any needle to any other needle.

(*c*) A larger disk may never be placed upon a smaller disk.

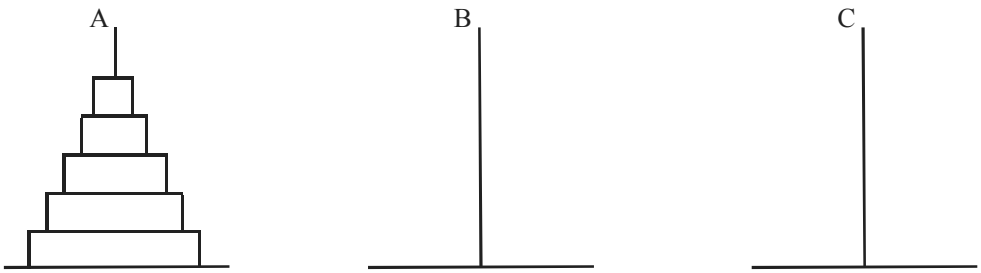
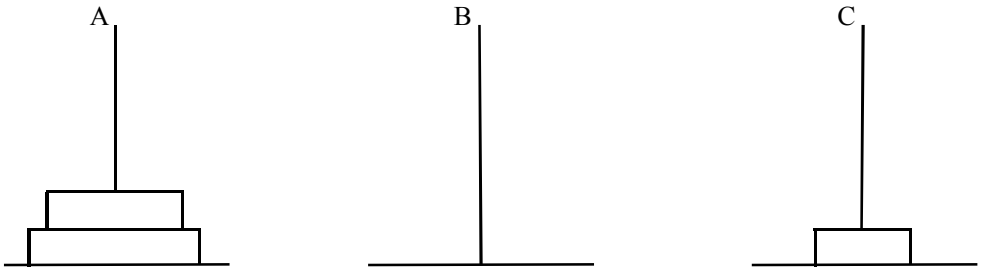


Figure 5.6 *Tower of Hanoi*

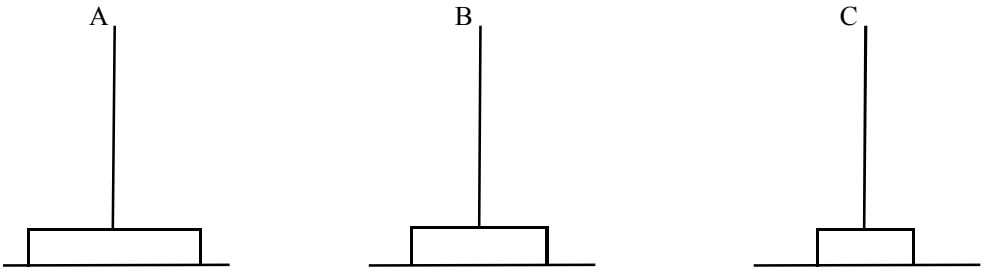
Let us see how to solve this problem. If there is only one disk, we merely move it from A to C. If there are two disks, we move the top disk to B and move the second disk to C and finally move the disk from B to C.

In general we can move *(n* – 1) disks from A to B and then move the nth disk *from A* to C, finally we can move *(n* – 1) disks from B to C.

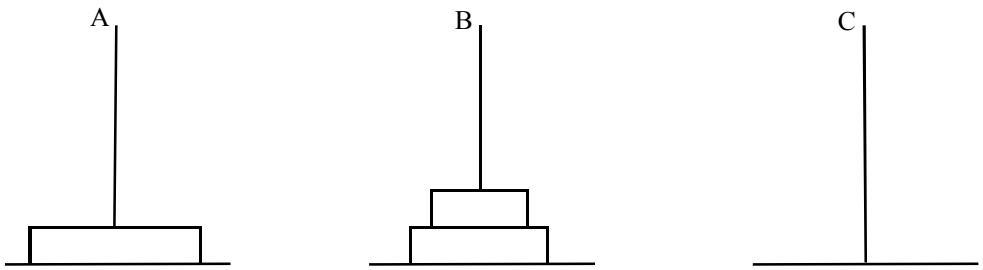
To understand it better, let *n* be 3. The following figures (Fig. 5.7) illustrate the disk movements.



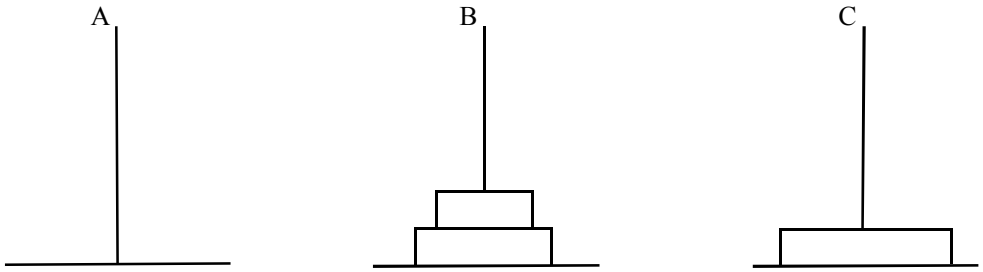
(*i*) Move disk 1 from needle A to needle C



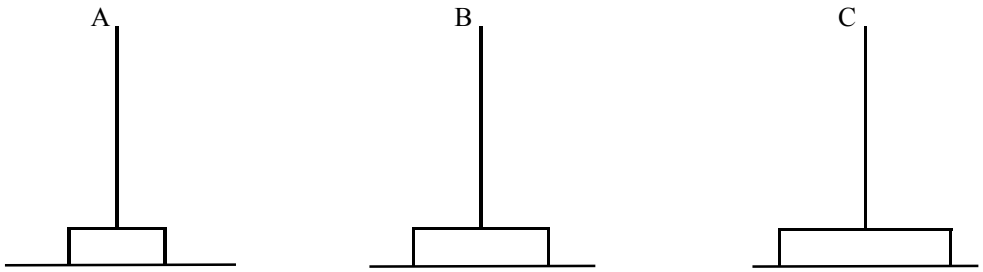
(*ii*) Move disk 2 from needle A to needle B



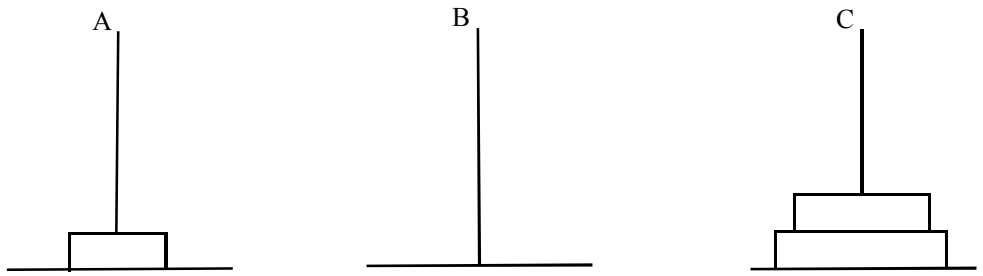
(*iii*) Move disk 1 from needle C to needle B



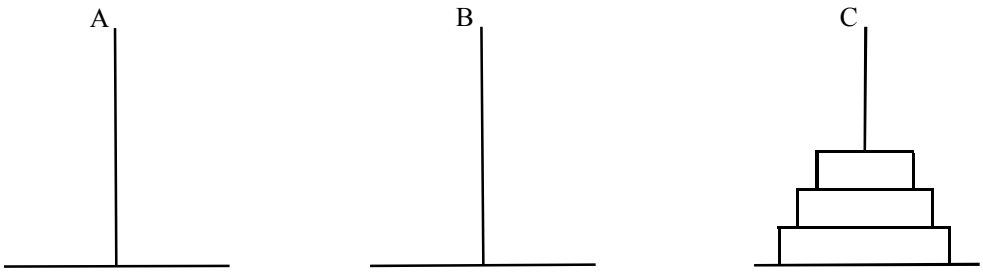
(*iv*) Move disk 1 from needle A to needle C



(*v*) Move disk 1 from needle B to needle A



(*vi*) Move disk 1 from needle B to needle C



(*vii*) Move disk 1 from needle A to needle C

Figure 5.7 *Tower of Hanoi for three disks.*

Therefore, a recursive solution can be formulated to solve the problem of Hanoi to move *n* disks from *A* to C using *B* as auxiliary.

(*a*) If n = 1**,** move the single disk from *A* to C and return.

(*b*) Move the top *n* – 1 disks from *A* to B using C as temporary.

(*c*) Move the remaining disk from *A* to C.

(*d*) Move the *n* – 1 disks from *B* to C, using *A* as temporary.

Now, let us convert this algorithm to C program. For that let us decide about the input and output to the program. As an input to the program, we must give *n,* the number of disks. Apart from number of disks, we must also specify the needles which are to act as the initial needle from which we are moving disks, the final needle to which we are moving disks, and temporary needle.

We must also decide about the names of the disks. Let the numbering itself represent the names of the disks. The smallest disk which is on top of the needle is numbered 1, the next disk is 2 and so on. Such that the largest disk is represented by number *n.*

The output from the program could be a list of statements as:

move disk nn from needle *x* to needle *y*.

The solution of the problem then would be to perform action according to the output statement in exactly the same order that they appear in the following program

STACK/HANOI.CPP

```
/* Tower of Hanoi problem with implicit recursion*/
/* HANOI.CPP */

# include<stdio.h>

void towers(char , char , char , int );

/* Definition of the Tower Of Hanoi generator function */

void towers(char needle1, char needle2, char  needle3, int n)
{
        if( n <= 0)
                printf("\n Illegal entry");

        /*If only one disk, make the move and return */
        if(n == 1)
        {
```

```
            printf("\n Move Disk 1 from needle %c to needle %c", needle1, needle2);
            return;
        }
        /* Move top n – 1 disks from A to B, using C as auxiliary */
            towers(needle1, needle3, needle2, n – 1);
        /* Move remaining disk from A to C */
            printf("\n Move Disk %d from needle %c to needle %c",n, needle1, needle2);
        /* Move  n – 1 disks from B to C, using A as auxiliary */
            towers(needle3, needle2, needle1, n – 1);
    } /* end of towers */
    /* main function */
    void main()
    {
        int n;
        printf("\n Input the number of disc:");
        scanf("%d", &n);
        printf("\n Tower of Hanoi for %d DISCs", n);
        towers('A', 'C', 'B', n);
    }
```

Output of the program:

```
Input the number of disk: 4
Tower of Hanoi for 4 DISCs
Move Disk 1 from needle A to needle B
Move Disk 2 from needle A to needle C
Move Disk 1 from needle B to needle C
Move Disk 3 from needle A to needle B
Move Disk 1 from needle C to needle A
Move Disk 2 from needle C to needle B
Move Disk 1 from needle A to needle B
Move Disk 4 from needle A to needle C
Move Disk 1 from needle B to needle C
Move Disk 2 from needle B to needle A
Move Disk 1 from needle C to needle A
Move Disk 3 from needle B to needle C
Move Disk 1 from needle A to needle B
Move Disk 2 from needle A to needle C
Move Disk 1 from needle B to needle C
```

Verify the steps and check that it does not violate any of the rules of the problem at any stage. We may note that it is quite possible to develop a nonrecursive solution to the problem of Towers of Hanoi directly from the problem statement.

Example 6. Write a 'C' program to remove a recursion from the following function using stack.

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

Program is written using stack functions push, pop and empty. The non-recursive function for abc is written. A sum variable is used to addition of return values.

/* The program recstack.cpp implement the above function. The function abc() is written in the form of non-recursive form using explicit stack operations empty, pop and push */


```
#include <stdio.h>
#include <stdlib.h>
#define MAXSIZE 80
#define TRUE 1
#define FALSE 0
int pop(struct stack *);
void push(struct stack *, int);
int empty(struct stack *);
int abc(int n);
struct stack{
int top;
int items[MAXSIZE];
};
void main()
{
int n;
printf("\n Enter a number");
scanf("%d",&n);
printf("\n Result is %d", abc(n));
} /* end main */
/*Recursive function abc() is written in form of stack */
int abc(int n)
{
int sum = 0;
struct stack s;
s.top = 0;
while (s.top>=0)
{
    if (n > 0)
    {
        push(&s, n-1);
```

```
                push(&s, n – 2);
        }
        if (n<=0)
                sum = sum + 1;
        n = pop(&s);
    }
    return(sum);
    }
/* function push the integer element into stack s */
void push (struct stack *s, int d)
{
        if (s->top == MAXSIZE –1)
        {
                printf("%s", "stack overflow");
                exit(0);
        }
        else
                s->items[++(s->top)] = d;
        return;
} /* end of push function */
/* function pop the integer element from stack s */
int pop(struct stack *s)
{
        if (empty(s))
        {
                printf("\n%s", "stack underflow");
                exit(0);
        }
        return(s->items[s->top– –]);
} /* end of pop function */
int empty(struct stack *s)
{
        if (s->top == –1)
                return (TRUE);
        else
                return (FALSE);
} /*end of empty function */
```

Output:
Enter a number: 5
Result is 13

Chapter 6

Algorithms on Queue

A queue is an ordered linear list in which all insertions take place at one end, the rear, whereas deletions are made at other end, the front.

The operations of a queue require that the first element that is inserted into the queue is the first one to be removed. Thus queues are known as First In First Out (FIFO) lists.

6.1 REPRESENTATION: USING ARRAY AND LINKED LIST

A queue is an ordered collection of elements. We can declare a variable queue as an array. However, a queue and an array are two entirely different things. The number of elements in an array is fixed and assigned by the declaration for the array. A queue, on the other hand, is fundamentally a dynamic object whose size is constantly changing as items are deleted and inserted.

However, although an array cannot be a queue, it can be the home of a queue. That is, an array can be declared large enough for the maximum size of the queue. During the program execution, the queue can grow and shrink within the space reserved for it. One end of the array is called start of the queue i.e. front end, while the last end of the queue is rear end and both ends constantly shift as items are deleted and inserted.

A queue in 'C' may be declared as a structure containing three objects: an array to hold the elements of the queue, an integer rear to indicate the position of the current queue end within the array and an integer front to indicate the position of the start of queue within the array. This may be done for queue of integers by the declarations

```
#define MAXSIZE 10
struct queue {
        int front;
        int rear;
        int items[MAXSIZE];
        };
```

The queue declaration is as

```
struct queue q;
```

Here, we assume that the elements of the queue q contained in the array q.items[] are integers and maximum elements at the most is MAXSIZE. The queue can contain float, char or any other type of elements. A queue can also contain objects of different types by using 'C' unions.

The identifier front and rear must always be declared as an integer, since its value represents the position within the array items of the queue element.

The first or starting element in the queue is stored at s.items[0], the second at q.items[1] and the ith at q.items[i−1].

The empty queue contains no elements and can therefore be indicated by q.front = –1 and q.rear = –1.

Figure 6.1 shows an array representation of queue (for convenience, the array is drawn horizontally rather than vertically). The size of an array is 8. Initially queue has 5 elements.

The operations of queue imply that if the elements A, B, C, D, and E are inserted into a queue, in that order, then the first element to be removed (i.e. deleted) must be A.

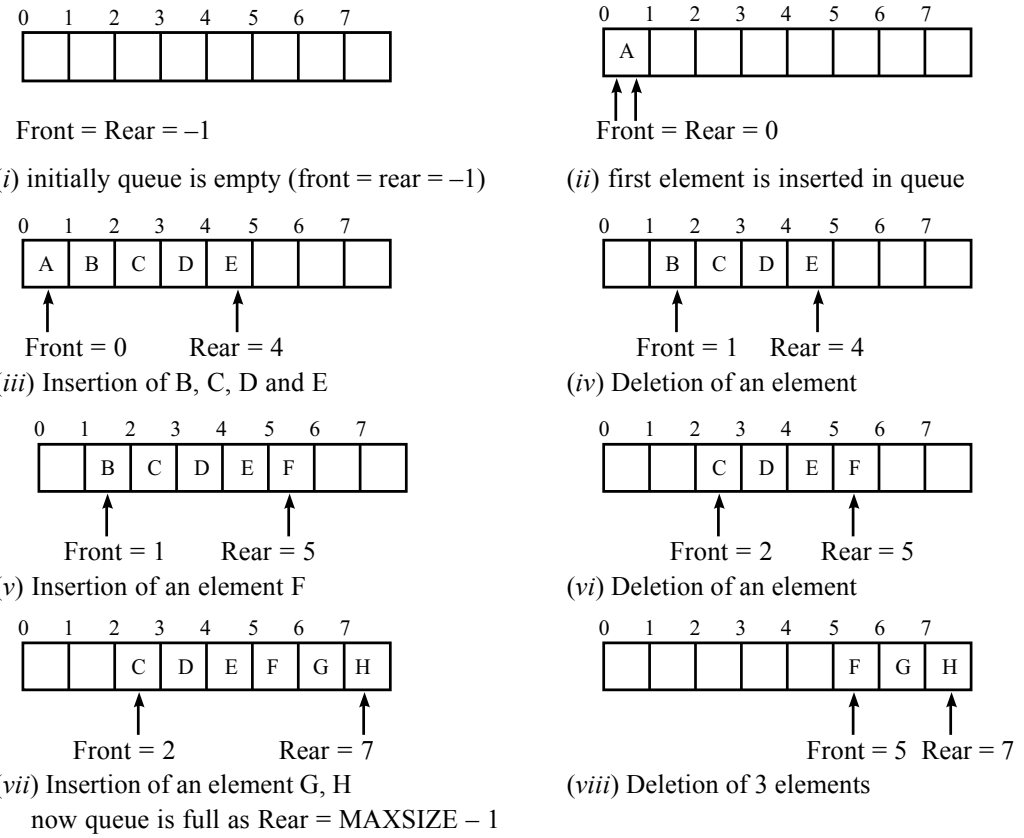


Figure 6.1 *Operations on Queue (MAXSIZE = 8)*

In array implementation actual deletion does not take place. Only front pointer is changed in its position. In the above figure the deleted elements' location is shown empty. In simple queue, the memory space is wasted.

The element of the queue array is displayed from front (i.e. lower bound) to rear (i.e. upper bound) index.

Linked Representation of Queue

We can represent queues using one-way or singly linked list. The advantages of linked lists over arrays have already been discussed.

The linked representation of queue is shown in Fig. 6.2. The info fields of the nodes hold the elements of the queue and next fields hold pointer to the neighboring element in the queue. The start pointer of the linked list behaves as the **front pointer** variable of the queue and NULL pointer of the last node denotes the **rear pointer** variable of the queue.

A queue in 'C' may be declared as a structure containing two objects: an item to hold the elements of the queue and a pointer indicate the position of the next element in the queue. This is similar to single linked list declaration.

This may be done for queue of integers by the declarations

```
struct queue {
        int items;
        struct queue *next;
        };
```

The queue declaration is as

```
struct queue *front, *rear;
```

The empty queue contains no elements and can be indicated by front = rear = NULL.

Linked representation of the queue is shown in Fig. 6.2.

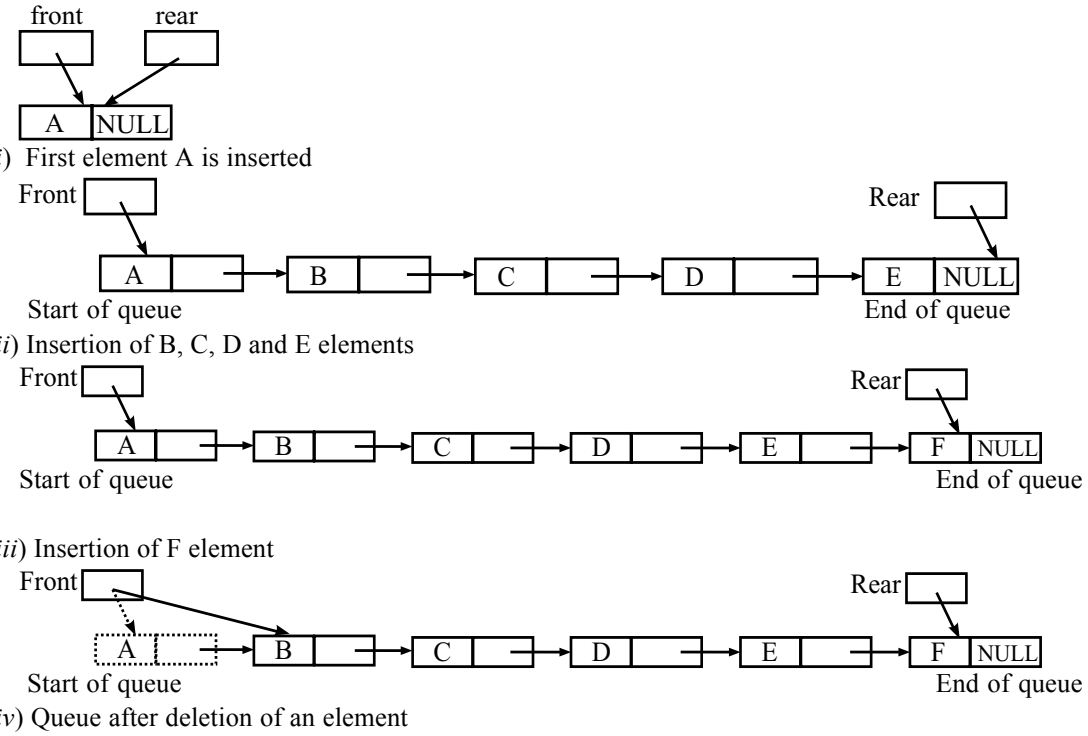


Figure 6.2 *Queue representation using linear linked list.*

The condition of queue fully depends upon the main memory availability for the linked list at the time of memory allocation for the node by the malloc function.

6.2 INSERTION AND DELETION OPERATIONS

Implementing the Insert Operation

The operation of inserting an item onto a queue is called insert operation. The simple function to implement insert operation using the array representation of a queue. Here, we check whether the array is full before attempting to insert element onto the queue. The array is full if q–>rear = MAXSIZE –1.

Whenever an element is inserted into the queue, the rear pointer variable value is increased by 1 (i.e. rear = rear + 1;).

The insert operation is written in the form of queueins function as given below:

The function needs two parameters, one is reference to queue structure and other is element to be inserted in the queue and it does not return anything. Thus, function declaration is

```
void queueins(struct queue * ,int ) ;/ * function declaration */
```

The function definition is given below.

```
void queueins(struct queue q ,int x) / function definition */
{
/*check for queue full, if so give a warning message and return */
if (q–>rear==MAXSIZE –1)
 {
  printf("Queue full\n");
   return;
 }
else
{
 /* check for queue empty, if so initialized front and rear pointers to zero. Otherwise increment rear pointer
by 1.  */
if (q–>front == –1)
 { q–>front = 0; q–>rear = 0;}
 else
   q–>rear = q–>rear + 1 ;
q–>items[q–>rear] = x;            }
} /* End of queueins function */
```

The function first checks the queue full condition and if queue is full it returns. If queue is empty i.e. the first element will be inserted in the queue so front and rear pointers set to 0 index. Otherwise rear pointer is increased by one.

Implementing the Delete Operation

The operation of deleting an item onto a queue is called delete operation. The possibility of underflow must be considered in implementing the delete operation.

The delete operation performs two functions:

1. If the queue is empty, print a warning message and halt execution.
2. Remove the first element from the queue and return this element to the calling program.

Whenever an element is deleted from the queue, the front pointer variable value is increased by 1 (i.e. front = front + 1;). However, in real life application of queue front pointer does not move. The front pointer contains the location of the front (i.e., start) element of the queue. In case of deletion whole queue elements advance their position by one location. But this requires lots of shifting in the array, so we have implemented insertion by increasing front pointer index value by 1.

The delete operation is given in the form of queuedel function as follows:

The function needs one parameter, a return type. The parameter is reference to queue structure and it returns deleted element value. Thus function declaration is

```
int queuedel(struct queue *);  /* function declaration */

int queuedel(struct queue *q)            /*function definition */
 {
 int x;

/*check for queue empty, if so raise a warning message and return*/

 if (q–>front == –1)
     printf(" Queue is empty\n");
 x = q–>items[q–>front];

/* if both pointers at the same position then reset the queue to empty position  otherwise increment front pointer by 1.*/

 if (q–>front == q–>rear)
{ q–>front = –1; q–>rear = –1;}
 else
   q–>front = q–>front +1;
return x;
}
```

The delete function prints the error message queue underflow and execution halts.

The empty function return true if q–>front = –1, otherwise return false.

```
#define TRUE 1
#define FALSE 0
int empty(struct queue *q)
{
     if (s–>front == –1)
       return (TRUE);
     else
      return (FALSE);
} /*end of empty function */
```

The complete program for linear queue is given below:

/* Program queue.cpp perform insertion, deletion operations on a queue which is implemented using array in structure. */

QUEUE/QUEUE.CPP

```
#include<stdio.h>
#define MAXSIZE 5
struct  queue{
     int items[MAXSIZE];
     int rear;
     int front;
}q;

void queueins(struct queue *,int);       /* function declaration */
```

```
int  queuedel(struct queue *); /* function declaration */
void display(struct queue *);  /* function declaration */

void queueins(struct queue *q ,int x)   /* function definition */
{
/*check for queue full, if so give a warning message and return */

if (q–>rear == MAXSIZE –1)
 {
  printf("Queue full\n");
   return;
 }
else
{
/* check for queue empty, if so initialized front and rear pointers to zero. Otherwise increment rear
    pointer */
 if (q–>front == –1)
 { q–>front = 0; q–>rear = 0;}
 else
   q–>rear = q–>rear+1;
 q–>items[q–>rear] = x;
}
} /* End of queueins function */

int queuedel(struct queue *q)
 {
 int x;
/*check for queue empty, if so raise a warning message and return*/

 if (q–>front == –1)
       printf(" Queue is empty\n");
 x = q–>items[q–>front];

/* if both pointer at the same position then reset the queue to empty position  otherwise increment front
pointer by 1.*/
 if (q–>front == q–>rear)
 { q–>front = –1; q–>rear = –1;}
 else
   q–>front = q–>front + 1;
return x;

}

void display(struct queue *q)
 {
 int i;
 if (q–>front ! = –1)
```

```
 if (q–>front < = q–>rear)
  for( i = q–>front;i <=q–>rear;i++)
     printf("%d          ",q->items[i]);
} /* end of display function */

void main()
 {
     int ch,t;
     q.rear = –1;
     q.front = –1; /* Initially queue is empty */

     do{
     printf("\n Operations on Linear Queue");
     printf(" \n 1. Insert \n 2. Delete\n 3. Display \n 4. Exit:\n");
     printf("\n Select operation");
     scanf("%d",&ch);
     switch(ch)
     {
      case 1:
                printf("\nEnter data");
                scanf("%d",&t);
                queueins(&q,t);
                break;
      case 2:
                queuedel(&q);
                break;
      case 3:
                display(&q);
                break;
      case 4:   break;
        }
  }while(ch!=4);

} /* end of main */
```

Analysis of Program

Program includes three functions: queueins for inserting element in the queue, queuedel for deleting an element from queue and display function to display the current elements in the queue.

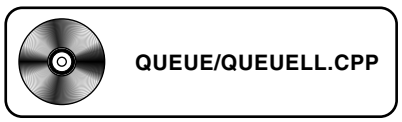
It displays a menu for insert, delete and display operations on queue.

The limitation of linear queue is that if the last position (i.e., MAXSIZE –1) is occupied, it is not possible to insert an element in queue even though some locations are vacant at front end of the queue.

However, this limitation can be overcome if we consider the next index after MAXSIZE – 1 is 0. The resulting queue is known as circular queue.

A complete 'C' program to implement queue using linked list is given on the next page, which saves memory as nodes are allocated memory if they needed, no wastes of memory.

/* Program queuell.cpp to implement queue operation using linked list */




```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct queue{
int items;
struct queue *next;
}*rear,*front;

void main( )
{
int c;
void insertion( );
void display( );
void deletion( );
clrscr( );
front = rear = NULL;

do {
printf("\n Enter choice for Queue ....");
printf("\n 1. Insert in Queue.... ");
printf("\n 2. Delete in Queue ..");
printf("\n 3. Display Queue..");
printf("\n 4. Exit from Queue..");
printf("\n Enter choice for Queue ....");
scanf("%d",&c);
switch (c)
{
case 1:
     insertion( );
     break;
case 2:
     deletion( );
     break;
case 3:
     display( );
     break;
case 4:
     break;
}
}while (c!=4);
}
void insertion( )
{
struct queue *newnode, *ptr;
```

```
newnode = (struct queue *) malloc(sizeof(struct queue));
printf(" Enter items ");
scanf("%d",&newnode->items);
newnode–>next = NULL;
if (rear = = NULL)
      {
      rear = newnode;
      front = newnode;
      }
else
      {
      ptr = rear;
      rear–>next = newnode;
      rear = newnode;
        }
}
void display()
{
struct queue *ptr;
ptr = front;
while (ptr ! = NULL)
{
printf(" %d ",ptr–>items);
ptr = ptr–>next;
}
getch();
}
void deletion()
 {
   struct queue *ptr;
  if (front = = NULL)
                {
                printf("\n Queue is empty");
                rear = NULL;
                return;
                }
   ptr = front;
   front = front–>next;
   free(ptr);
 }
```

6.3 CIRCULAR QUEUE

An efficient queue representation can be obtained by taking an array and treating it as circular. Elements are inserted by increasing the pointer variable rear to the next location in the array. When rear = MAXSIZE –1,

the next element is entered at items[0] in case when no element is at index 0. The variable front also increased as deletion and in same fashion as rear variable.

The Fig. 6.3 of circular queue is given below for n elements.

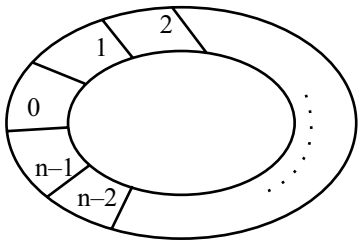


Figure 6.3 *Circular queue*

Figure 6.4 shows how a queue is maintained by a circular array items with n = 5 (i.e., MAXSIZE = 5) memory locations. Circular queue also occupies consecutive memory locations.

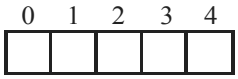
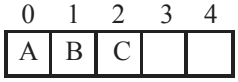
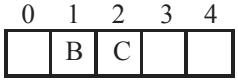
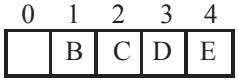
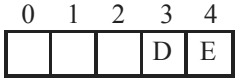
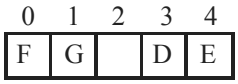
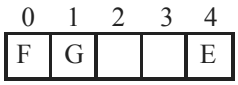
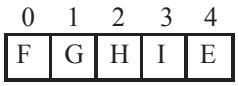
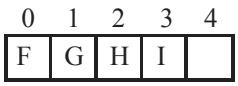
(*i*)	Initially queue empty	rear = –1, front = –1	
(*ii*)	A, B and C inserted	rear = 2, front = 0	
(*iii*)	Deletion	rear = 2, front = 1	
(*iv*)	D and E inserted	rear = 4, front = 1	
(*v*)	Two deletions	rear = 4, front = 3	
(*vi*)	F and G inserted	rear = 1, front = 3	
(*vii*)	deletion	rear = 1, front = 4	
(*viii*)	H and I inserted	rear = 3, front = 4	
(*ix*)	deletion	rear= 3, front = 0	

Figure 6.4 *Operation of circular queue*

Operations on Circular Queue

Implementing the Insert Operation

The operation of inserting an item onto a queue is called insert operation. The simple function to implement insert operation using the array representation of a queue. Here, we check whether the array is full before attempting to insert element onto the queue.

The conditions of circular queue full is either one of the below:

1. (front = 0) and (rear = MAXSIZE –1).
2. front = rear + 1.

Whenever an element is inserted into the queue, the rear pointer variable value is increased by 1 (i.e. rear = (rear + 1) % MAXSIZE;). So that after rear pointer variable values MAXSIZE – 1, it increased by one to denote rear value 0.

The insert operation is written in the form of queueins function as given below:

The function needs two parameters, one is reference to queue structure and other is element to be inserted in the queue and it does not return anything. Thus function declaration is same in linear queue and function definition is given below:

```
void queueins(struct queue *q,int x)    /* function definition */
{
/* check for queue full if front pointer at index 0 and rear pointer at MAXSIZE –1. Or front is just behind
rear i.e. front = rear +1 */
if (((q–>front = = 0) &&(q->rear = = MAXSIZE–1))||(q–>front == q–>rear + 1))
 {
  printf("Queue full\n");
   display(q);                                    /* function calling */
   return;
 }
else
{
 if (q–>front = = –1)
 { q–>front =0; q–>rear = 0;}
 else
   q–>rear = (q–>rear + 1) % MAXSIZE;
 q–>items[q–>rear] = x;
}
} /* End of queueins function */
```

The delete operation is given in the form of queuedel function as below:

The function needs one parameter, a return type. The parameter is reference to queue structure and it returns deleted element value. Thus, function declaration is

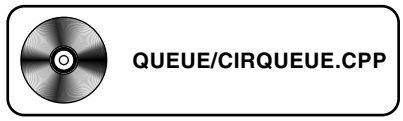
```
int queuedel(struct queue *);  /* function declaration */
```

and function definition is given below:

```
int queuedel(struct queue *q)            /* function definition */
 {
 int x;
 /* check for queue empty when front pointer is at index–1. */
 if (q–>front = = –1)
       printf(" Queue is empty\n");
 x = q–>items[q–>front];
```

```
/* check for front equal to rear means queue has become empty and reinitiates front and rear to –1.
Otherwise front is increased by 1 in modulus way */
 if (q–>front == q–>rear)
 { q–>front = –1; q–>rear = –1;}
 else
   q–>front = (q–>front + 1) % MAXSIZE;
return x;
}
```

```
/* Program cirqueue.cpp perform insertion, deletion operation on a circular queue which is implemented
using array in a queue structure . */
```




```
#include<stdio.h>
#define MAXSIZE 5
struct  queue{
      int items[MAXSIZE];
      int rear;
      int front;
}q;
void queueins(struct queue *,int);       /* function declaration */
int queuedel(struct queue *);            /* function declaration */
void display(struct queue *);            /* function declaration */

void queueins(struct queue *q,int x)     /* function definition */
{
/* check for queue full if  front pointer at index 0 and rear pointer at MAXSIZE –1. Or front is just behind
rear i.e. front = rear +1 */
if (((q–>front == 0) &&(q–>rear == MAXSIZE –1))||(q–>front == q–>rear + 1))
 {
  printf("Queue full\n");
   display(q);                                   /* function calling */
   return;
 }
else
{
 if (q–>front == –1)
 { q–>front = 0; q–>rear = 0;}
 else
   q–>rear = (q–>rear + 1) % MAXSIZE;
 q–>items[q–>rear] = x;
}
} /* End of queueins function */

int queuedel(struct queue *q)            /* function definition */
 {
```

```
int x;
/* check for queue empty when front pointer is at index –1. */
if (q–>front = = –1)
      printf(" Queue is empty\n");
x = q–>items[q–>front];
/* check for front equal to rear means queue has become empty and reinitiates front and rear to –1.
Otherwise front is increased by 1 in modulus way */
if (q–>front == q–>rear)
{ q–>front = –1; q–>rear = –1;}
else
  q–>front = (q–>front + 1) % MAXSIZE;
return x;
}
void display(struct queue *q)                    /* function definition */
{
int i;
if (q–>front ! = –1)
 if (q–>front < = q–>rear)
  for(i = q–>front;i<=q–>rear; i ++)
     printf("%d     ",q–>items[i]);
 else
 {
 for( i = q–>front;i<MAXSIZE;i++)
  printf("%d     ",q–>items[i]);
 for( i = 0; i< = q–>rear;i ++)
  printf("%d     ",q–>items[i]);
 }
} /* end of display function */
void main()
{
    int ch,t;
    q.rear = –1;
    q.front = –1; /* Initially queue is empty */
    do{
    printf("\n Operations on Circular Queue [Maximum size of queue is 5]");
    printf(" \n 1. Insert \n 2. Delete\n 3. Display \n 4. Exit:\n");
    printf("\n Select operation");
    scanf("%d",&ch);
    switch(ch)
    {
```

```
        case 1:
                printf("\nEnter data");
                scanf("%d",&t);
                queueins(&q,t);                 /* function calling */
                break;
        case 2:
                queuedel(&q);                   /* function calling */
                break;
        case 3:
                display(&q);                    /* function calling */
                break;
        case 4: break;
          }
    }while(ch! = 4);

} /* end of main */
```

Analysis of Program

Program perform the operations on circular queue. The function of linear queue has been changed to perform these operations. These functions are: queueins for inserting element in the queue, queuedel for deleting an element from queue and display function to display the current elements in the queue.

It displays a menu for insert, delete and display operations on queue.

Parameters and return type of function are not changed. Only queue full condition is changed and front and rear pointer is increased by one in modulus form so that after index MAXSIZE –1 its value is 0 not MAXSIZE.

The limitation of circular queue is that the number of memory locations are fixed, so size can't be changed in any representation either in array or in linked list.

6.4 DEQUES

A deque is double-ended queue, in a linear list in which elements can be added or removed at either end but not in middle. There are two variations of a deque:

1. Input-restricted deque

This form of deque allows insertions only one end but allows deletions at both ends.

2. Output-restricted deque

This form of deque allows insertions at both ends but allows deletions at one end.

There are various ways to representing a deque in memory. We will assume our deque is maintained by a circular array deque with pointers left and right, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. Fig. 6.5 shows deque operations with their pointer positions.

(*i*) Initially deque is empty left = –1

(*ii*) A, B and C inserted in right end right = 2, left = 0

0	1	2	3	4
A	B	C		

(*iii*) Deletion from right end right = 1, left = 0

0	1	2	3	4
A	B			

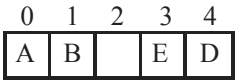
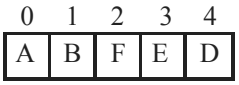
(*iv*) Inserted D, E in left end	right = 1, left = 3	
(*v*) Inserted F in right end	right = 2, left = 3 [now queue is full]	

Figure 6.5 *Operations of deque*

Operations on DEQUEs

Implementing Insertion Operation

The operation of inserting an item into a deque is called insert operation. We can insert element into both ends of the queue that are represented by left and right variables.

1. **Insertion in Left end of the queue.** The left pointer is changed in its value according to the following conditions:
   (*a*) If left = right +1, then queue is full and insertion can't take place.
   (*b*) If left pointer is –1, then the first element is inserted in the array deque and assigned value of both left and right pointers to 0.
   (*c*) If left is at index 0, then assigned left = MAXSIZE –1 otherwise decrement left by one i.e. left = left –1.
2. **Insertion in Right end of the queue.** The right pointer is changed in its value according to the following conditions:
   (*a*) If left = right +1, then queue is full and insertion can't take place.
   (*b*) If left pointer is –1, then the first element is inserting in the array deque and assigned value of both left and right pointers to 0.
   (*c*) If right is at index MAXSIZE –1, then assigned right = 0 otherwise increment right by one i.e. right = right +1.

To implement insertion, we have written a insdeque function, which takes two parameters, one for the elements to be inserted and other if a flag value indicating the operation is from left end or right end of the queue. The function does not return anything.

Implementing Deletion Operation

The operation of deleting an item into a deque is called delete operation. We can delete element into both ends of the queue that are represented by left and right variables.

1. **Deletion in Left end of the queue.** The left pointer is changed its value according to the following conditions:
   (*a*) If left = –1, then queue is empty and deletion can't take place.
   (*b*) If left = right, then array deque becomes empty and assigned value of both left and right pointers to –1.
   (*c*) If left is at index MAXSIZE –1, then assigned left = 0 otherwise increment left by one i.e. left = left + 1.
2. **Deletion in Right end of the queue.** The right pointer is changed its value according to the following conditions:
   (*a*) If left = –1, then queue is empty and deletion can't take place.
   (*b*) If left = right, then array deque becomes empty and assigned value of both left and right pointers to –1.

(*c*) If right is at index 0, then assigned right = MAXSIZE –1 otherwise decrement right by one i.e. right = right –1.

Elements of Deque are displayed from left end to right end index.

The conditions of deque full is left = right + 1 and the condition of deque empty is left = –1.

To implement deletion, we have written a deldeque function, which takes one parameter flag that indicates the operation is from left end or right end of the queue. The function returns a value for the deleted element.

A complete program with insertion and deletion functions from both ends of the queue is given below.

/* Program deque.cpp to implement double ended queue operations using circular array deque. Program performs insertions and deletions from both ends of the queue which is represented by left and right pointer variables*/

QUEUE/DEQUE.CPP

```
#include<stdio.h>
#define MAXSIZE 5
int deque[MAXSIZE];
int right = –1;
int left = –1;

void insdeque(int, int);
int deldeque(int);
void display();

/* Function insdeque insert an element into left or right end  of deque */
void insdeque(int x, int f)
{
/* check for deque full */
if (left == right + 1)
 {
  printf("Queue full\n");
   return;
 }
else
{
 if (left == –1)       /* queue is empty */
 { left = 0; right = 0; deque[left] = x; return; }
 else if (f == 1)      /* insert in right end */
      {
       right = (right + 1) % MAXSIZE;
       deque[right] = x;
       }
   else
   {
   if (left == 0)      /* insert in left end */
        left = MAXSIZE –1;
```

```
      else
        left = left –1;
      deque[left] = x;
    }
}
} /* End of insdeque function */

/*Function deldeque delete an element from left or right end of the deque*/
int deldeque(int f)
{
 int x;
 if (left == –1)
      {
       printf(" Queue is empty\n");
       return –1;
      }
 if (left == right)            /* when queue is full */
 {x = deque[left]; left = –1; right = –1;}
 else
   {
  if (f == 0)
   {
   x = deque[left];         /* delete from left end */
   left = (left + 1) % MAXSIZE;
   }
   else
   {
   x = deque[right];
   if (right == 0)          /* delete from right end */
        right =  MAXSIZE –1;
   else
       right = right –1;
   }
 }
return x;
}

void display()
{
 int i;
 if (left ! = –1)
  if (left <= right)
   for( i = left;i< = right;i++)
      printf(" %d       ", deque[i]);
```

```
 else
 {
 for( i = left;i<MAXSIZE;i++)
  printf("%d      ", deque[i]);
 for( i = 0;i< = right;i++)
  printf("%d      ", deque[i]);
 }
} /* end of function */

void main( )
 {
    int ch,t;
  do{
    printf("\n DEQUE (Double Ended Queue):— Opertions ");
    printf(" \n 1. Insert in left \n 2. Insert in right \n 3. Delete in left\n");
    printf("4. Delete in right\n 5. Display elements \n 6. Exit.\n");
    scanf("%d",&ch);
    switch(ch)
    {
     case 1:
                printf("enter data ");
                scanf("%d",&t);
                insdeque(t,0);
                break;
     case 2:
                printf("enter data ");
                scanf("%d",&t);
                insdeque(t,1);
                break;
     case 3:
                deldeque(0);
                break;
     case 4:
                deldeque(1);
                break;
     case 5:
                display( );
                break;
     case 6:  break;
     } /* end of switch */
  }while(ch!=6);    /* end of do -while */
} /* end of main( ) */
```

6.5 PRIORITY QUEUE

Any data structure that supports the following operations efficiently is called priority queue:

1. Searching of an element with minimum or maximum property
2. Insertion
3. Deletion of an element with minimum or maximum property.

A priority queue is a collection of elements such that each element has been assigned an explicit or implicit priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed to the order in which they were inserted to the queue.

The priority of element is decided by its value is known as implicit priority and priority number is assigned with each element is known as explicit priority.

The application of priority queue in timesharing operating system: programs of higher priority are processed first, and programs with the same priority form a standard queue.

There are two types of priority queues: an ascending priority queue and a descending priority queue.

An **ascending priority queue** is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.

A **descending priority queue** is a collection of items into which items can be inserted arbitrarily and from which only the largest item can be removed.

There are various ways of maintaining a priority queue in memory. These are:

- One way linked list (with or without priority number)
- Multiple queues, one for each priority (i.e. priority number is needed)
- Maximum or minimum heap (no need of priority number, key value itself decides the priority)

The simplest way to represent a priority queue is an unordered linear list. Suppose that we have n elements in this queue and to delete the element with maximum key. If the list is represented sequentially, insertions are most easily performed at the end of this list. Hence, the insert time is O(1). A deletion requires a search for an element with the largest key, followed by its deletion. So each deletion takes O(n) time.

An alternative is to use an ordered linear list. The elements are in nondecreasing order if sequential representation is used. The deletion time for this one is O(1) but insertion time now is O(n).

In case of heap, both insertions and deletions can be performed in $O(\log_2 n)$ time.

One-way List Representation of a Priority Queue

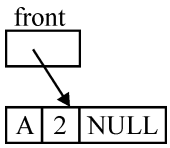
One way to maintain a priority queue in memory is one way list, we are maintaining **ascending priority queue** as follows:

1. Each node in the list contains three items of information: an information field Info, a priority number prt and a next pointer.
2. A node x precedes a node y in the list when x has higher priority than y or when both have same priority but x was inserted to the list before y.

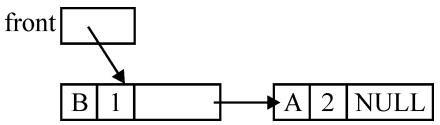
The following operations are performed:

(*i*) Searching of an element with maximum priority
(*ii*) Insertion
(*iii*) Deletion of an element with maximum priority

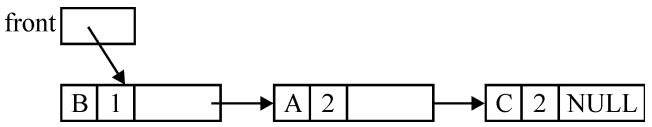
The first element in the list is the element with maximum priority. So deletion needs only O(1) time, but insertion needs O(n) time. Only one pointer is required i.e. start, denotes the first node in the linked list.



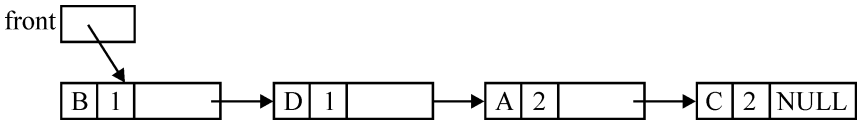
(*i*) First element A is inserted with priority number is 2



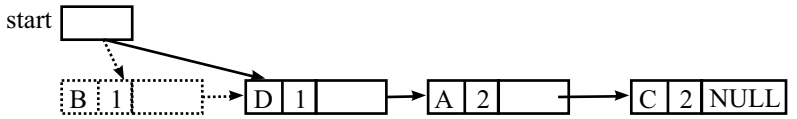
(*ii*) Element B with priority number 1 is inserted



(*iii*) Element C with priority number 2 is inserted



(*iv*) Element D with priority number 1 is inserted



(*v*) Deletion of element with highest priority (i.e. 1)

Figure 6.6 *Ascending priority queue with ordered linear list*

The algorithm INSSL in Chapter 4, is an insertion sort, in which node is inserted according to its value in ascending order. The same algorithm can be used for ascending priority queue, only change is that one more field of priority number is added and the nodes are inserted according to their priority numbers.

The algorithm INAPQUEUE insert node in ascending priority queue.

The algorithm INAPQUEUE as follows:

```
        Algorithm INAPQUEUE
        [This algorithm inserts item into a linked list which is sorted according to node priority].
Step 1 : loc = FINDLOC(front, item);          [return the address of the node preceding item]
Step 2 : newnode = Allocate memory using malloc() function
Step 3 : Set newnode->info = item;            [Assigned value of the information part of a new node]
         Set newnode->prt =prtno;
Step 4 :   if loc = NULL then                 [Insert as first node]
                Set  newnode–>next = front; and front = newnode;
         else                                 [Insert after node with address loc]
            Set  newnode->next = loc->next; and loc->next = newnode;
         [End of If statement]
```

Step 5 : end INAPQUEUE

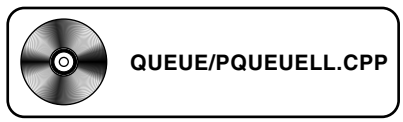
The algorithm FINDLOC is written in the form of function.

Algorithm FINDLOC(front, item)
[This algorithm returns the address loc of the last node in a sorted list such that loc->info < item, or sets loc = NULL]
Step 1 : if front = NULL then
Set loc = NULL; and return(loc); [List is empty]
Step 2 : if item < front–>prt then
Set loc = NULL; and return(loc); [the item is smaller than the first node item, thus must be inserted as the first node of the list]
Step 3 : Set prevptr = front; and ptr = front–>next; [Initialize pointers]
Step 4 : repeat Steps 5 and 6 while ptr<> NULL
Step 5 : if item < ptr->prt then
Set loc = prevptr; and return(loc);
Step 6 : Set prevptr = ptr; and ptr = ptr->next; [Updates pointers]
Step 7 : Set loc = prevptr;
Step 8 : return(loc);

The algorithm to delete an element in ascending priority queue is similar to delete the first node in the linked list as given in Chapter 4.

The complete program is given below.

/* Program pqueuell.cpp to implement priority queue operation using linked list. Each element has been assigned a priority in such a way (i) An element of higher priority is processed before any element of lower priority. */




```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct queue {
int items;
int prt;
struct que *next;
{ *rear, *front;

void main ( )
{
int c, item=0, prtitem=0;
void insertion (int, int);
void display ( );
void deletion ( );
clrscr ( );
front=rear=NULL;
do {
printf ("\n Enter choice for Priority Queue ....");
printf ("\n 1. Insert in Queue .....");
printf ("\n 2. Delete in Queue ..");
```

```
printf ("\n 3. Display Queue ..");
printf ("\n 4. Exit from Queue ..");
printf ("\n Enter choice for Priority Queue ....");
fflush (stdin);
scanf ("%d", &c);
switch (c)
 {
case 1 :
        printf ("\n Enter the process number and its priority");
        scanf ("%d%", & item, & prtitem);
        break;
case 2 :
        deletion ( );
        break;
case 3 :
        display ( );
        break;
case 4 :
        break;
}
} while (c!=4);
}   /* end of main  */

void insertion (int insitem, int prtitem)
{
        struct queue *ptr, *newnode, *loc;
        struct queue *findloc (struct queue *, int);
        newnode = (struct queue *)malloc (sizeof (struct queue));
        newnode–>items = insitem;
        newnode–>prt = prtitem;
        newnode–>next = NULL;
        if (front = = NULL)
        {
          rear = newnode;
          front = newnode;
        }
        else
        {
        loc = findloc (front, prtitem);
        if (loc == NULL)
        {
        newnode –>next = front;
        front = newnode;
        }
        else
        {
```

```
            newnode–>next = loc–>next;
            loc–>next = newnode;
            }
            }
            }/* end of the function */
    struct queue * findloc (struct queue *front, int prtitem)
    {
            struct queue *prevptr, *ptr;
            if (front == NULL)
                  return (NULL);
            if (prtitem<front–>prt)
                        return (NULL);
            prevptr =front;
            ptr =front–>next;
            while (ptr!=NULL)
            {
            if (prtitem < ptr–>prt)
                   return (prevptr);
            prevptr =ptr;
            ptr = ptr –>next;
            }
    return (prevptr);
    } /* end of findloc function */

    void display ( )
    {
    struct queue *ptr;
    ptr= front;
    printf ("/n Process number  priority /n");
    while {ptr !=NULL)
    {   printf (" %d/n ", ptr–>items, ptr–>prt);
    ptr = ptr–>next;
    }
    getch ( );
    }  /* end of display function */

    void deletion ( )
        {
            struct queue *ptr;
            if (front == NULL)
                        {
                         printf ("\n Queue is empty ");
                         return ;
            ptr=front;
            front=front–>next;
            free (ptr);
           } /* end of deletion function */
```

6.6 MULTIPLE QUEUES

Figure 6.7 shows the scheme to represent multiple queues in a same array queue[] of size n. Each queue i is allocated a predefined space bounded by array indices. Here a vector front and rear denotes the indices for each queue.

Queue 1 indices from front[0] to rear[0]
Queue 2 indices from front[1] to rear[1]
Queue 3 indices from front[2] to rear[2]
Queue 4 indices from front[3] to rear[3]
Queue 5 indices from front[4] to rear[4]

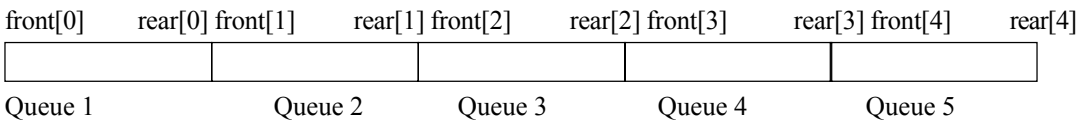


Figure 6.7 *Representation of five queues by a single array.*

The most common use of multiple queues is implementation of priority queue. In that case a queue is maintained for each priority. In order to process an element of the priority queue, element from first nonempty highest priority number queue is accessed.

In order to insert a new element to the priority queue, the element is inserted in an appropriate queue for given priority number.

The first queue has highest priority number elements, 2nd queue has next higher priority number elements and so on.

Consider that we have processes of five different priority numbers. So five queues are maintained in the queue array. Each queue has fixed size.

Operations on Multiple Queues

The items are inserted in queues according to their priority, priority 1 items are inserted in queue 1, and priority 2 items are inserted in queue 2 and so on. The deletion first takes place from first non-empty queue.

A complete 'C' program to implement multiple queues with priority is given below:

```
/* Program multique.cpp perform insertion, deletion operations on a multiple queue with priority which
is implemented using array queue structure. We have assumed four different priority processes define
MAXQUEUE = 4 and maximum number of items in each queue is 5  */

#include<stdio.h>
#define MAXSIZE 20
#define MAXQUEUE 4
struct  queue{
            int items[MAXSIZE];
            int rear[MAXQUEUE];
            int front[MAXQUEUE];
}q;
void queueins(struct queue *,int,int);
int  queuedel(struct queue *);
void display(struct queue *);

void queueins(struct queue *q ,int x, int p)
{
```

QUEUE/MULTIQUE.CPP

```
/*check for queue full, if so give a warning message and return */
if (q–>rear[p–1] == 5*p –1)
 {
  printf("Queue with priority %d is full\n", p–1);
   return;
 }
else
{
/* check for queue empty, if so initialized front and rear pointer to zero. Otherwise increment rear pointer */
 if (q–>front[p–1] == 5*(p–1)–1)
 { q–>front[p–1] = 5*(p–1); q–>rear[p–1] = 5*(p–1);}
 else
   q–>rear[p–1] = q->rear[p–1] +1;
 q–>items[q–>rear[p–1]] = x;
}
} /* End of queueins function */

int queuedel(struct queue *q)
 {
 int x, i;
/*check for queue empty, if so raise a warning message and return*/
  for (i = 0;i<4;i++)
   {
       if (q–>front[i] == 5*i – 1)
            {
                  printf(" Queue with priority %d is empty\n",i+1);
                   continue;
            }
      /*if both pointers at the same position then reset the queue to
       empty position  otherwise increment front pointer by 1.*/
       if (q–>front[i] == q–>rear[i])
                  { q–>front[i] = 5 *i–1; q->rear[i] =5*i –1;}
       else
        {
          x = q–>items[q–>front[i]];
          q–>front[i] = q–>front[i] +1;
        }
    printf("\n Items deleted from queue %d",  i+1);
     break;
  }
return x;
}
void display(struct queue *q)
 {
 int i,j;
 for(i = 0;i<4; i++)
 {
```

```
 if (q–>front[i] ! = –1)
  {
   printf("\n Items in queue with priority %d–> ", i+1);
  if (q–>front[i] < = q–>rear[i])
    for( j = q–>front[i];j< = q–>rear[i];j++)
      printf ("%d ",q–>items[j]);
 }
 }
} /* end of display function */
void main()
 {
     int ch,t, i,prt;
     for(i = 0;i<MAXQUEUE;i++)
     {
      q.rear[i] = 5*i –1;
      q.front[i] =5*i –1; /* Initially set all queues to empty */
     }
     do{
     printf("\n Operations on Multiple Queues To implement Priority");
     printf(" \n 1. Insert \n 2. Delete\n 3. Display \n 4. Exit:\n");
     printf("\n Select operation");
     scanf("%d",&ch);
     switch(ch)
     {
      case 1: do{
                printf("\nEnter process number and its priority (1..4) [–1 for termination]");
                scanf("%d%d",&t,&prt);
                if ((t ! = –1) &&((prt>0) && (prt<5)))
                        queueins(&q,t,prt);
                else
                        break;
                }while(1);
                 break;
      case 2:
                 queuedel(&q);
                 break;
      case 3:
                 display(&q);
                 break;
      case 4:  break;
        }
 }while(ch!=4);
} /* end of main */
```

Analysis of Program

Program multique.cpp used three functions queueins to insert an item into corresponding to its queues, queuedel to delete an item from first non-empty queues. The maximum size of each queue is 5 and 4 queues are maintained. The first element to be deleted is always from first non-empty queue.

Chapter 7

Non-Linear Data Structure: Trees

7.1 GENERAL CONCEPT

Tree represents hierarchical relationship. A tree is a finite set of one or more nodes such that there is a specially designated node called the root and the remaining nodes are partitioned into n>= 0 disjoint sets T1,… , Tn, where each of these sets is a tree. The sets T1….Tn are called the subtree of root.

$V \rightarrow$ Vertex
$T = \{V,E\}$ $E \rightarrow$ Edge
$R \subset V$ $R \rightarrow$ Root

Root does not have a parent, but each one of the other nodes has a parent node associated to it. A node may or may not have children. A node that has no children is called leaf node.

Degree: The number of subtree of a node is called as its degree.

Terminal node: Nodes with no children. It can be also called as leaf node whose degree is zero.

Non-terminal node: Nodes with at least one child.

Sometimes non-terminal nodes are called internal nodes and terminal nodes as external nodes.

Ancestor and descendant: All nodes in the path from root to that node are known as Ancestor node. For example, node B, A are the ancestors of node E. If there is path from n_i to n_j, then n_i is the ancestor of n_j and n_j is the descendant of n_i.

The tree, Figure 7.1, ancestors of node E are A and B and descendants of node B are E and F.

A **path** in a tree is a list of distinct vertices in which successive vertices are connected by edges in the tree. There is exactly one path between the root and each of the other nodes in the tree.

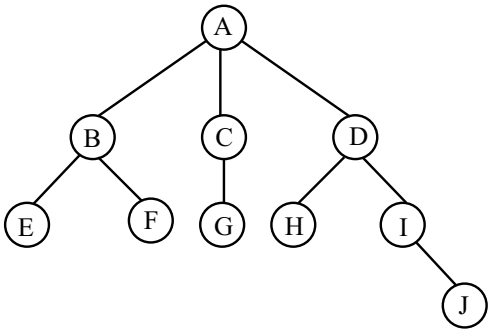
A path from vertex A to H is A–D–H of length 2.

Sibling: Children of same parent are known as siblings. Root does not have sibling. For example in the shown tree B, C and D are siblings.

Height: Height of tree is the number of edges from root to its farthest leaf node. Thus all leaves are at height zero. The height of the tree is same as the height of the root.

The **depth** of any node n_i is the length of the path from root to n_i. Depth of a node is sometimes also referred to as **level** of a node.

Each node, except the root, has a unique parent, and every edge connects a node to its parent. Therefore, a tree with N nodes has N–1 edges.



Level	Node	Degree
0	A	3
1	{B, C, D}	{2, 1, 2}
2	I	1
3	E, F, G, H and J have degree 0	

Figure 7.1 *A General tree*

- Node A is the root node.
- E, F, G, H, J are the Terminal nodes.
- The nodes E, F are siblings of subtree B, so as H, I are of subtree D.
- Node G has no sibling as the parent node has no other child node.
- Height of the tree is 3.

A set of trees is called a forest. For example, if we remove the root node A and the edges connecting it from the tree, then we have forest consisting of three trees rooted at B, C and D.

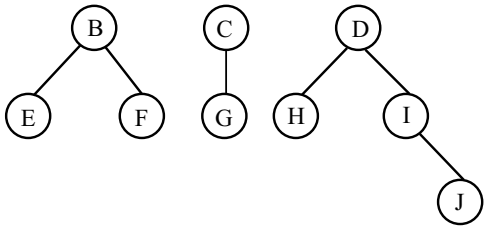


Figure 7.2 *A forest (3 sub-trees)*

Examples

- Organization structure of a corporation

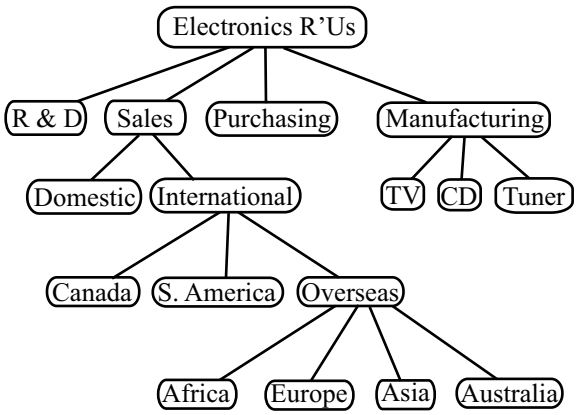


Figure 7.3 *A hierarchical tree structure of an organization*

Level : Level is (number of ancestor+1) for the given node. All the sibling nodes have same level. Root always be on 1st level.

Height of any node is the number of edges to farthest node in left or right subtree. For example in the following tree:

- Node 2 has height 2
- Node 3 and 4 have height 1.
- Nodes 5,6,7,8 have height = 0.

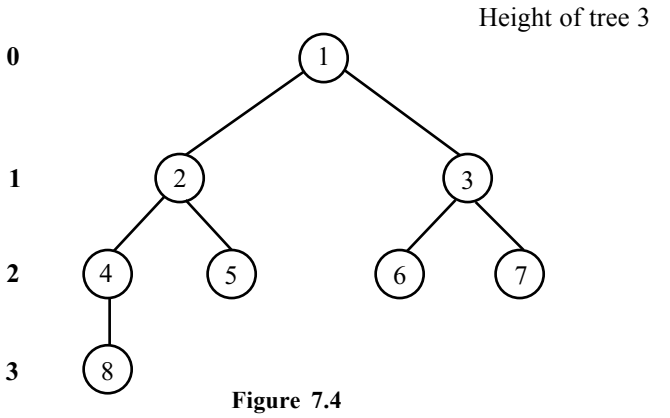


Figure 7.4

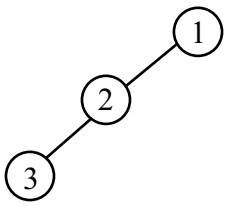
We are discussing binary tree first, then we will discuss general tree and its conversion.

7.2 BINARY TREE

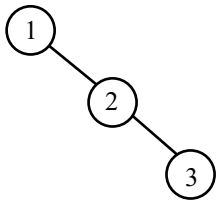
A binary tree is a set of finite sets of nodes that is empty or consists of root and two disjoint binary trees called the left and right subtree. The left or right subsets are themselves binary trees. A left or right sub–tree can be **empty.**

Types of Binary Tree:

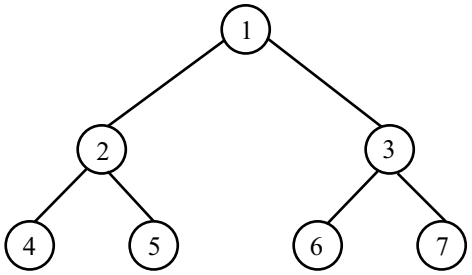
1. Left aligned Binary tree:



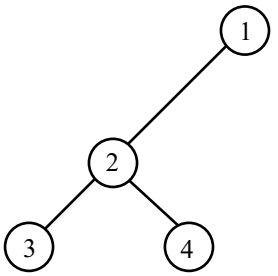
2. Right aligned Binary tree:



3. Complete Binary tree:



4. Left Heavy:



5. Right Heavy:

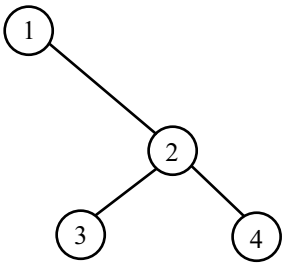


Figure 7.5 *Different binary trees*

Number of nodes at level		Level	Depth
1	1	0	**0**
2	2 3	**1**	**1**
4	4 5 6 7	**2**	**2**

Figure 7.6

Height (h) = 2

Number of nodes in complete binary tree $= 2^{h+1}-1$

$= 2^3-1$

$= 7$

Number of nodes at each level $= 2^{level}$

- The maximum number of nodes on level i of a binary tree is 2^i. For the shown binary tree number of nodes on each level are:
  Nodes at level 2 → 2^2
  $= 4$
  Nodes at level 1 → 2^1
  $= 2$
  Nodes at level 0 → 2^0
  $= 1$
- The maximum number of nodes in a binary tree of depth **k** is 2^k where $k > 0$.
- If a binary tree contains 'm' nodes at level 'L', then it contains at most '2m' nodes at level 'L+1'.
- Therefore, total number of nodes in complete binary tree is
  $$n = 2^0 + 2^1 + 2^2 + \cdots + 2^d = 2^{(d+1)} - 1$$
- Find d = ?
  $$d = \log_2(n+1) - 1$$
- If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $1 \le i \le n$, we have:
  (a) parent[i] is at i/2 if i <> 1. When i = 1, i is the root and has no parent.
  (b) lchild[i] is at 2*i if 2*i ≤n, if 2*i > n, i has no left child.
  (c) rchild[i] is at 2*i +1 if 2*i +1 ≤n, if 2*i +1 > n, i has no right child.

7.3 SEQUENTIAL AND LINKED LIST REPRESENTATION OF BINARY TREE

Sequential Representation

Suppose T is a binary tree that is complete or partially complete. The easy way of maintaining T in memory is sequential representation. This representation uses only a linear array BTREE as follows:

(*i*) The root of tree is stored in BTREE[0].
(*ii*) If a node occupies BTREE[i], then left child is stored in BTREE[2*i +1] and its right child is stored in BTREE[2*i +2]
(*iii*) If the corresponding tree does not exist then its pointer is null.

The sequential representation of the binary tree T in Fig. 7.4 appears in Fig. 7.7. It requires 16 locations in the BTREE array even though T has only 8 nodes.

	1	2	3	4	5	6	7	8	0	0	0	0	0	0	0	0	0
i→	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

BTREE

Figure 7.7 *Sequential representation*

In general, the sequential representation of a binary tree with depth d will require an array with approximately 2^{d+1} elements or locations.

Linked List Representation

There are two types of implementation:

1. Array Implementation
2. Dynamic memory allocation implementation

1. Array Implementation

```
Struct  nodetype
        {
        int info;
        int left;
        int right;
}node[MAXSIZE];
```

MAXSIZE define the maximum number of nodes in the binary tree. Each node of binary tree consists of the following field

(*i*) node[i].info contains the data at the node i.

(*ii*) node[i].left contains the location of the left child of node i.

(*iii*) node[i].right contains the location of the right child of node i.

The root contains the location of the root of the tree T. If any subtree is empty, then the corresponding pointer will contain the null value. If the tree T itself is empty, then root will contain null value. In the array implementation null value is denoted by 0 (See Fig. 7.8).

Node[i]	Info	Left	Right
0			
1	6	0	0
2	2	6	5
3			
4	1	2	7
5	5	0	0
6	4	9	0
7	3	1	8
8	7	0	0
9	8	0	0

Figure 7.8 *Memory representation of tree using array of structure (MAXSIZE = 10)*

2. Dynamic Memory Allocation Implementation

```
struct  nodetype
    {
          int info;
         struct nodetype *lchild;
         struct nodetype *rchild;
  }*node;
```

In binary tree each node has two child pointers, lchild pointer contains the address of left child of the node and rchild pointer contains the address of right child of the node. If the child of the node does not exist then respective pointer consists of null value.

Memory is allocated memory for each node using malloc function in 'C' programming.

node = (struct nodetype *)malloc(sizeof(struct nodetype));

The concept and addressing scheme of linked allocation is already discussed in detail in Chapter 4.

Consider a binary tree T in Fig. 7.4. Fig. 7.9 shows how linked representation may appear in memory.

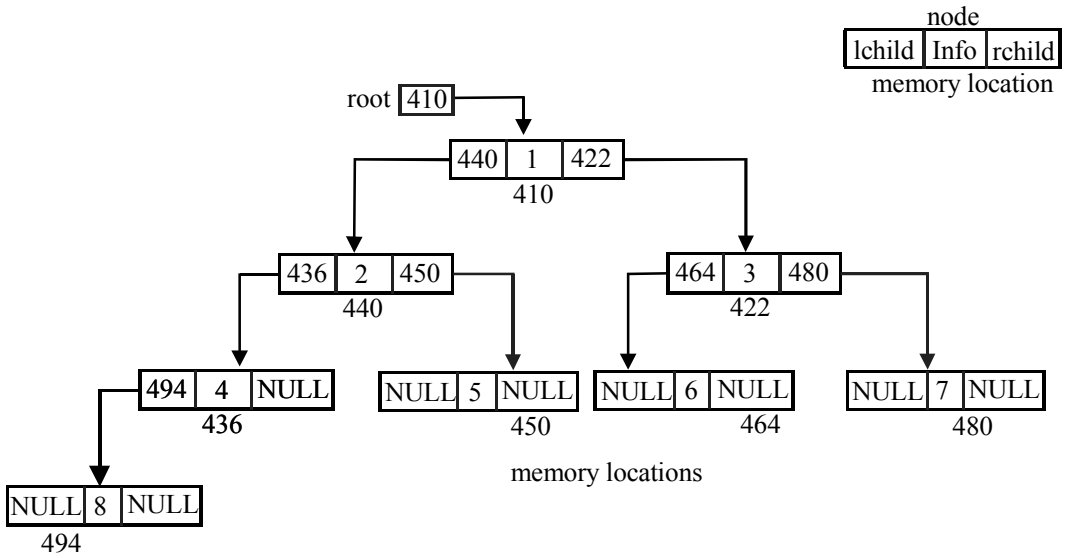


Figure 7.9 *Linked representation of binary tree (Empty pointer has null value)*

7.4 BINARY TREE TRAVERSAL ALGORITHM: RECURSIVE AND NON-RECURSIVE

Another common operation is to traverse a binary tree that is to pass through the tree, enumerating each of its nodes once. Here different ordering is used for traversal in different cases. We have defined four traversal methods.

The methods are all defined recursively, so that traversing a binary tree involves visiting the root and traversing its left and right subtrees. The only difference among the methods is to the order in which these three operations are performed.

1. Preorder (also known as depth first order) (left subtree → right subtree → visit node)
2. Inorder (or symmetric order) (left subtree → visit node → right subtree)
3. Postorder (left subtree → right subtree → visit node)
4. Backward inorder (right subtree → node → left subtree)

Preorder Traversing

```
Algorithm Preorder(root)      //Recursive
      if ( root ! = NULL)
Step 1 :  {     print data;
Step 2 :        Preorder (root → lchild);
Step 3 :        Preorder (root → rchild);
          }
Step 4 :        end Preorder
```

Algorithm Preorder(root) //Non Recursive or Iterative

```
/*
 A binary tree T is in memory and array stack is used to temporarily hold address of nodes
*/
```

Step 1 : [Initialize top of stack with null and initialize temporary pointer to nodetype is ptr]

```
Top = 0;
stack[Top] = NULL;
ptr = root;
```

Step 2 : repeat steps 3 to 5 while ptr ! = NULL

Step 3 : visit ptr → Info

Step 4 : [Test existence of right child]

```
if ( ptr → rchild ! = NULL)
{
        Top = Top + 1;
        stack[Top] = ptr → rchild;
}
```

Step 5 : [Test for left child]

```
if (ptr → lchild ! = NULL)
        ptr = ptr → lchild;
else
    [Pop from stack]
        ptr = stack[Top];
        Top = Top–1;
[End of loop in Step 2.]
```

Step 6 : end Preorder.

Consider Fig. 7.10:

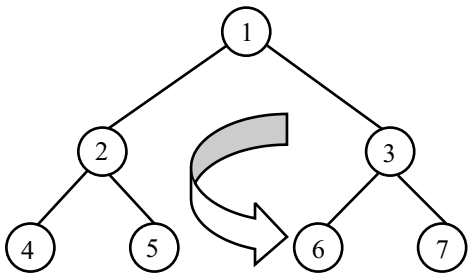


Figure 7.10

Preorder sequence is as follows

{1, 2, 4, 5, 3, 6, 7 }

Function: Iterative function for preorder traversal is given below:

```
void ipreorder(struct node *ptr)
{
struct node *stk[100];
```

```
    int tos = –1;
    do
    {
            while(ptr! = NULL)
            {
            printf("%d, ",ptr–>info);
            stk[++tos] = ptr;
            ptr = ptr–>lchild;
      }
      if(tos! = –1)
      {
            ptr = stk[tos– –];
            ptr = ptr–>rchild;
      }
}while(tos! = –1 || ptr! = NULL);
}
```

Inorder Traversing

Algorithm Inorder(root) //Recursive

Step 1 : if (root ! = NULL)
{
call Inorder (root → lchild);
print data;
call Inorder (root → rchild);
}

Step 2 : end Inorder

Algorithm Inorder(root) // Non Recursive or Iterative

```
/*
A binary tree is in memory and array stack is used to temporarily hold address of nodes.
*/
```

Step 1 : [Initially push NULL on stack and initialize ptr]
Top = 0;
stack[Top] = NULL;
ptr = root;

Step 2 : [In Inorder first we visit left child]
while (ptr ! = NULL)
{
Top = Top+1;
stack[Top] = ptr;
ptr = ptr → lchild;
}

Step 3 : ptr = stack[Top];
Top = Top–1;
Step 4 : repeat steps 5 to 7 while(ptr ! = NULL)
Step 5 : print ptr → Info;
Step 6 : if (ptr → rchild ! = NULL)
ptr = ptr → rchild;
goto step 2;
Step 7 : ptr = stack[Top];
Top = Top–1;
[End of while loop.]
Step 8 : end Inorder.

Consider Fig. 7.11

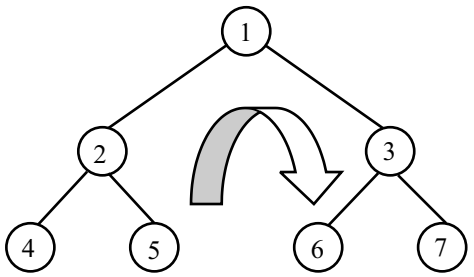


Figure 7.11

Inorder sequence is as follows:
{ 4, 2, 5, 1, 6, 3, 7 }

Function: Iterative function for inorder traversal is given below:

```
void iinorder(struct node *ptr)
{
struct node *stk[100];
int tos = –1;
do
{
        while(ptr! = NULL)
        {
                stk[++tos] = ptr;
                ptr = ptr–>lchild;
        }
        if(tos! = –1)
        {
                ptr = stk[tos– –];
                printf("%d ",ptr–>info);
                ptr = ptr–>rchild;
```

```
        }

    }while(tos! = –1 || ptr! = NULL);
    }
```

Postorder Traversing

Algorithm Postorder(root) //Recursive

Step 1 : if (root ! = NULL)
{
call Postorder(root → lchild);
call Postorder(root → rchild);
print data;
}

Step 2 : End Postorder

Algorithm Postorder(root) //Non Recursive or Iterative

/*
Binary tree is in memory and array stack is used to temporarily hold address of node.
*/

Step 1 : [Push NULL to stack and initialize ptr]
Top = 0;
stack[Top] = NULL;
ptr = root;

Step 2 : repeat steps 3 to 5 while(ptr! = NULL)

Step 3 : Top = Top+1;
stack[Top] = ptr;

Step 4 : if (ptr → rchild ! = NULL)
Top = Top + 1;
stack[Top] = ptr → rchild;

Step 5 : ptr = ptr → lchild;

Step 6 : [pop from stack]
ptr = stack[Top];
Top = Top–1;

Step 7 : while(ptr>0)
{
print ptr → Info;
ptr = stack[Top];
Top = Top–1;
}

Step 8 : if ptr<0 then
{
ptr = ptr → rchild;
go to Step 2;
}

Step 9 : end Postorder
Consider Fig. 7.12

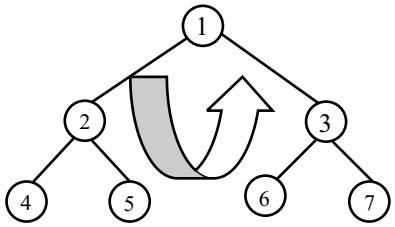


Figure 7.12

Postorder sequence is as follows:
{ 4, 5, 2, 6, 7, 3, 1 }

Function: Iterative function for postorder traversal is given below:

```
void ipostorder(struct node *ptr)
{
struct node *stk[100];
int tos = –1;
int visited,i,visit[10];
do
{
        while(ptr! = NULL)
        {
                printf("%d", ptr–>info);
                stk[++tos] = ptr;
                ptr = ptr–>lchild;
        }
        if(tos! = –1)
        {
                ptr = stk[tos– –];
                visited = 0;
                for(i = 0;i<tos;i++)
                        if(visit[i] = = ptr–>rchild)
                        {
                                visited = 1;
                                break;
                        }
                if(visited = = 1)
                {
                        printf("%d", ptr–>info);
                        ptr = NULL;
```

```
                }
                else
                {
                        stk[++tos] = ptr;
                        ptr = ptr–>rchild;
                        visit[ptr] = 1;
                }
                ptr = ptr–>rchild;
            }
    }while(tos! = –1 || ptr! = NULL);
    }
```

Backward Inorder Traversing

Algorithm BackInorder(root) //Recursive

Step 1 : if (root ! = NULL)
{
call Back Inorder (root → rchild);
print data;
call Back Inorder (root → lchild);
}

Step 2 : end BackInorder

Algorithm BackInorder(root) // Non Recursive

/*
A binary tree is in memory and array stack is used to temporary hold address of nodes.
*/

Step 1 : [Initially push NULL on stack and initialize ptr]
Top = 0;
stack[Top] = NULL;
ptr = root;

Step 2 : [In Backward Inorder first we visit right child]
while (ptr ! = NULL)
{
Top = Top+1;
stack[Top] = ptr;
ptr = ptr → rchild;
}

Step 3 : ptr = stack[Top];
Top = Top–1;

Step 4 : repeat steps 5 to 7 while(ptr ! = NULL)

Step 5 : print ptr → Info;

Step 6 : if (ptr → lchild ! = NULL)

```
            ptr = ptr → lchild;
            goto Step 2;
Step 7 :    ptr = stack[Top];
            Top = Top–1;
            [End of while loop.]
Step 8 :    end BackInorder.
```

Consider Fig. 7.13

Backward Inorder sequence is as follows:

{ 7, 3, 6, 1, 5, 2, 4 }

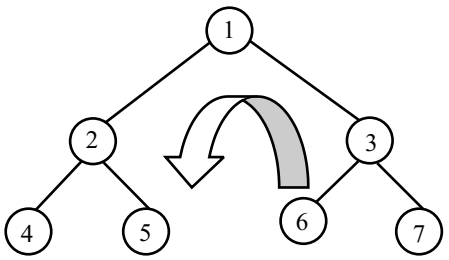
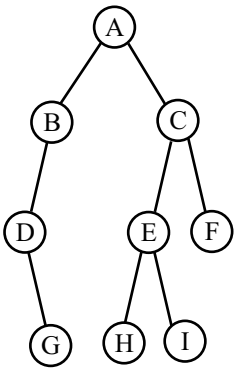
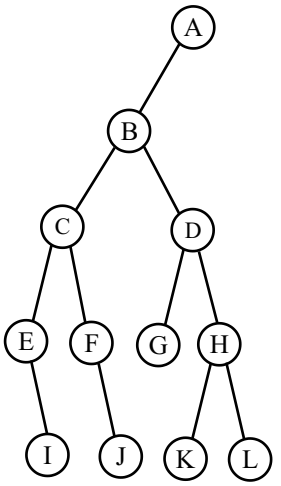


Figure 7.13

Examples: Binary Tree Traversing (see Fig. 7.14)





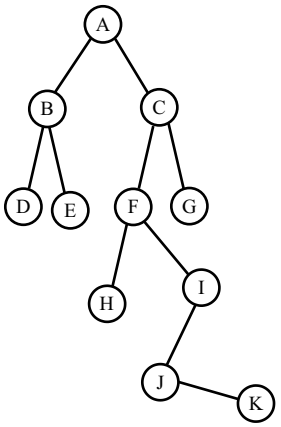
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F
Postorder	G	D	B	H	I	E	F	C	A
Backward	F	C	I	E	H	A	B	G	D

(*i*)

Preorder	A	B	C	E	I	F	J	D	G	H	K	L
Inorder	E	I	C	F	J	B	G	D	K	H	L	A
Postorder	I	E	J	F	C	G	K	L	H	D	B	A
Backward	A	L	H	K	D	G	B	J	F	C	I	E

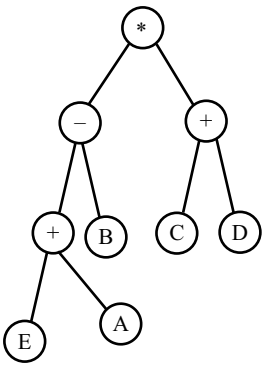
(*ii*)

(***Figure 7.14–contd...***)



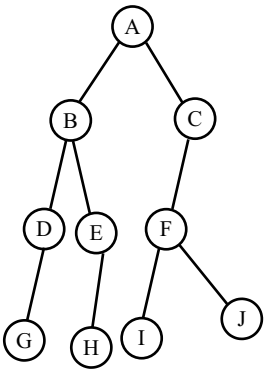
Preorder	A	B	D	E	C	F	H	I	J	K	G
Inorder	D	B	E	A	H	F	J	K	I	C	G
Postorder	D	E	B	H	K	J	I	F	G	C	A
Backward	G	C	I	K	J	F	H	A	E	B	D

(*iii*)



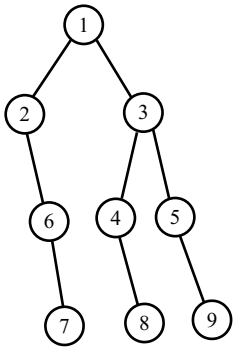
Preorder	*	–	+	E	A	B	+	C	D
Inorder	E	+	A	–	B	*	C	+	D
Postorder	E	A	+	B	–	C	D	+	*
Backward	D	+	C	*	B	–	A	+	E

(*iv*)



Preorder	A	B	D	G	E	H	C	F	I	J
Inorder	G	D	B	H	E	A	I	F	J	C
Postorder	G	D	H	E	B	I	J	F	C	A
Backward	C	J	F	I	A	E	H	B	D	G

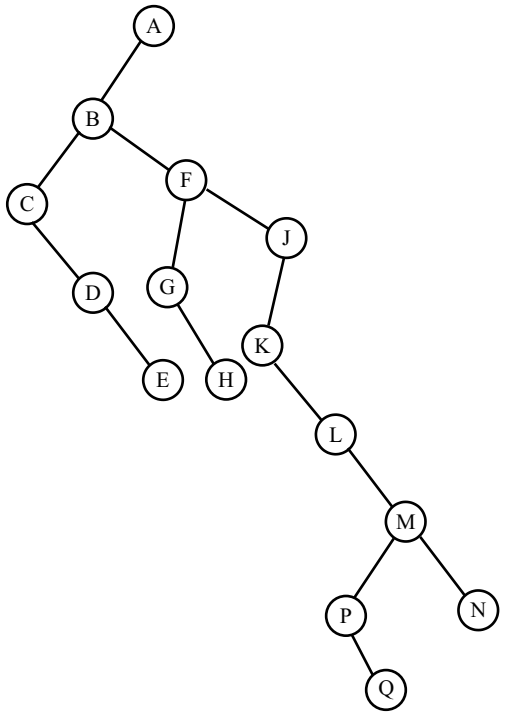
(*v*)



Preorder	1	2	6	7	3	4	8	5	9
Inorder	2	6	7	1	4	8	3	5	9
Postorder	7	6	2	8	4	9	5	3	1
Backward	9	5	3	8	4	1	7	6	2

(*vi*)

(***Figure 7.14–contd...***)



(*vii*)

Preorder		A	B	C	D	E	F	G	H		J	K	L	M	P	Q	N
Inorder		C	D	E	B	G	H	F	K	L	P	Q	M	N	J	A	
Postorder	E	D	C	H	G	Q	P	N	M	L	K	J	F	B	A		
Backward	A	J	N	M	Q	P	L	K	F	H	G	B	E	D	C		

Figure 7.14 *Examples of binary tree traversals*

Example 1. A binary tree T has 9 nodes. The inorder and preorder traversals of T yield the following sequence of nodes:

Inorder	E	A	C	K	F	H	D	B	G
Preorder	F	A	E	K	C	D	H	G	B

Draw the tree.

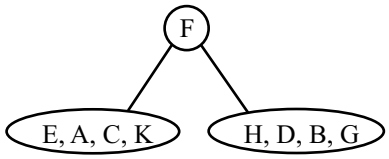
The tree T is drawn from its root downward as follows in different stages in Fig. 7.15.

(*i*) The root of T is obtained by choosing the first node in its preorder. Thus F is the root of T.

(*ii*) The inorder of T to find the nodes in the left subtree T of F. Thus T_1 consists of nodes E, A, C, K. Then the left child of F is obtained by choosing the first node in the preorder of T_1. Thus A is the left child of F.

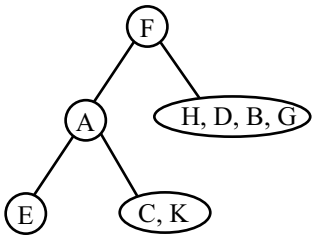
(*iii*) Similarly, the right subtree T_2 of F consists of the nodes H, D, B, and G. D is the right child of F. Repeating the above process with each new node is given below:

***Step* 1:**



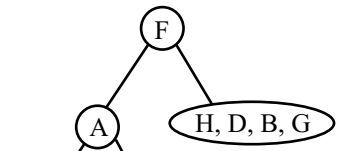
F is the root of tree

***Step* 2:**



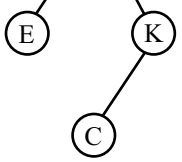
A becomes left child of T

***Step* 3:**



E is the left child of A

***Step* 4:**

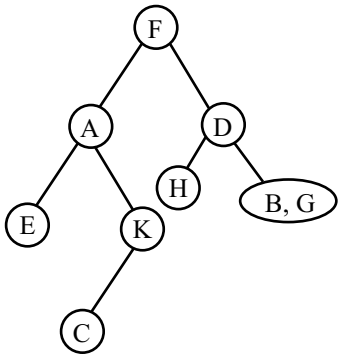


The next node in preorder K becomes right child of node A

***Step* 5:**

C is the left child of K

***Step* 6:**

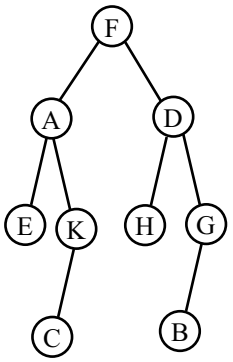


D becomes right child of F

(***Figure 7.15–contd...***)

***Step* 7:** Next node H becomes left child of D.

***Step* 8:**



Next node G becomes right child of D.

***Step* 9:** Next node B is the left child of G.

Figure 7.15 *construction of binary tree from inorder and preorder sequence*

The postorder sequence of tree is

visit left subtree, visit right subtree then visit rooted node.

E C K A H B G D F

Example 2. Consider the algebraic expression E = $(2x+y)*(5a–b)^3$

(*a*) Draw the tree T which corresponds to the expression E.

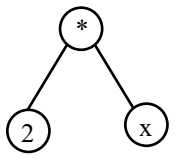
(*b*) Find the prefix polish expression P which is equivalent to E, find the preorder of T.

(*a*) First convert the expression into arithmetic expression

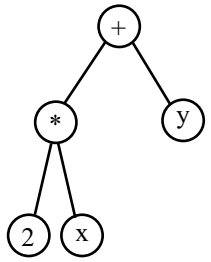
E = (2 * x+y) * (5 * a–b) ↑ 3

The binary tree is constructed by solving expression according to precedence and associative law (see Fig. 7.16).

2 * x ⇒

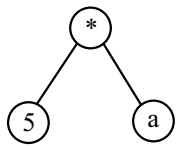


2 * x + y ⇒

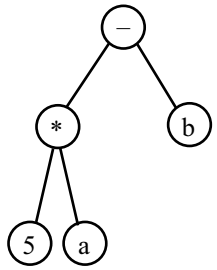


(***Figure 7.16–contd...***)

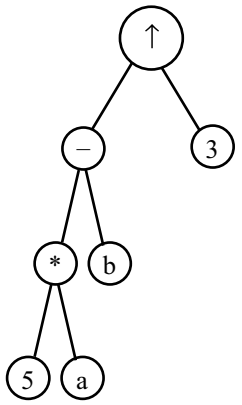
5 * a ⇒



5 * a –b ⇒



(5 * a – b) ↑ 3 ⇒



(2 * x + y) * (5 * a – b) ↑ 3 ⇒

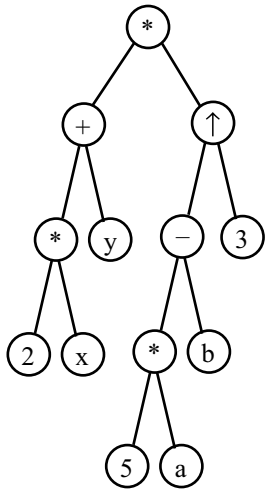


Figure 7.16

(*b*) There is no difference between the prefix polish expression P and the preorder of T

* + * 2 x y ↑ – * 5 a b 3

Here ↑ is symbol of exponent.

Example 3. Suppose the following sequence list the nodes of a binary tree T in inorder and postorder.

Inorder	2	6	7	1	4	8	3	5	9
Postorder	7	6	2	8	4	9	5	3	1

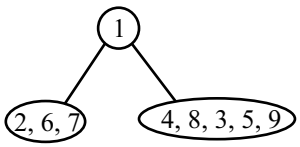
Draw the binary tree.

(*i*) The root of tree T in postorder traversal is last node i.e., 1.

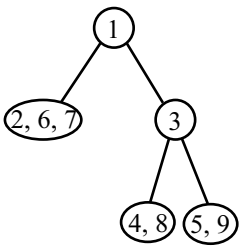
(*ii*) The inorder of T to find the nodes in the left subtree T_1 of root node 1. Thus T_1 consists of nodes 2, 6 and 7.

(*iii*) Similarly, the right subtree T_2 of root node 1 consists of the nodes 4, 8, 3, 5 and 9.

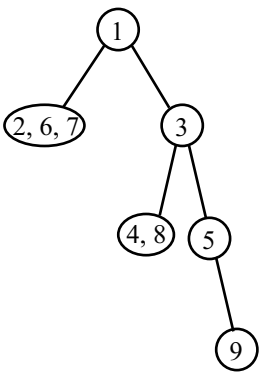
Repeat the above process, while traversing postorder node by node in right to left order (see Fig. 7.17).

1. 

1 is the root of T

2. 

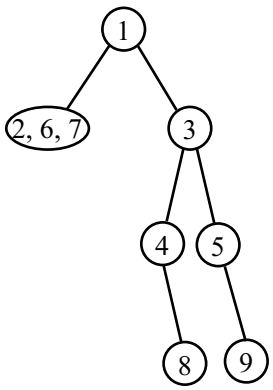
3 is the right child of 1

3. 

5 becomes the right child of 3

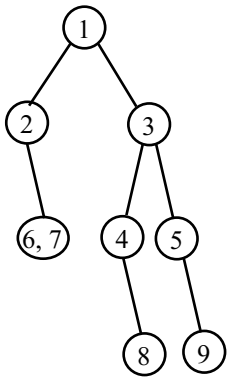
4. Next node 9 is right child of 5

(*Figure 7.17–contd...*)

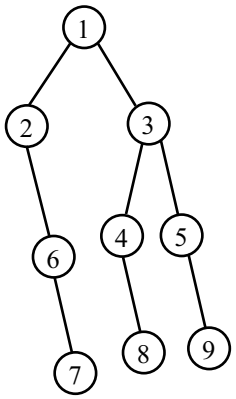
5. 

Next node 4 becomes the left child of 3

6. Next node 8 becomes the right child of 4

7. 

Next node 2 becomes the left child of 1

8. 

Next node 6 becomes the right child of 2

9. Next node 7 becomes the right child of 6

Figure 7.17

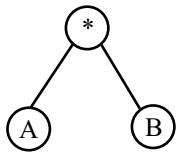
Example 4. Construct the expression binary tree for the following infix expression

A * B + C * D + E.

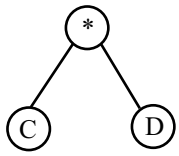
The tree is constructed by solving one operator at a time with the law of precedence and associatively.

The whole process is explained below in Fig. 7.18.

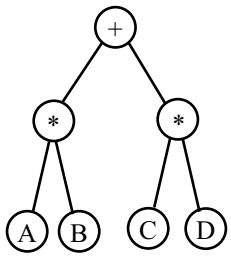
(*i*) A * B ⇒



(*ii*) C * D ⇒



(*iii*) A * B + C * D



(*iv*) A * B + C * D + E

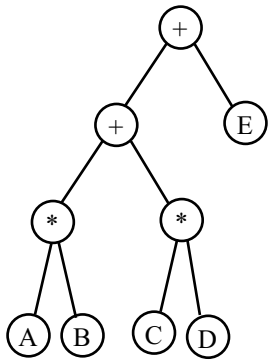


Figure 7.18

The prefix polish notation is equivalent to preorder traversal of T

+ + * A B * C D E

The postfix polish notation is equivalent to postorder traversal of T

AB * CD * + E +

7.5 THREADED BINARY TREE TRAVERSAL

The wasted NULL links in the storage representation of binary tree can be replaced by threads. A binary tree is threaded according to a particular traversal order. For example, the threads for the inorder traversal of a tree are pointers to its higher nodes. For this traversal order, if the left link of a node P is normally NULL, then this link is replaced by the address of the predecessor of P.

Similarly, a normally NULL right link is replaced by the address of the successor of the node in question.

We can distinguish structure link with a thread by positive address and negative address respectively.

Header Threaded Node

Consider a Binary Tree T is maintained in memory by means of a linked representation. Sometimes an extra special node, called a header node, is added to the beginning of T. Then HEAD node will point to the root T. The left points of header node will point to the root T.

Suppose a Binary Tree T is empty. Then T will contain a header node, but the left points of the header node

Left [Head] = NULL

will indicate an empty tree.

The HEAD node is simply another node that serves as the predecessor and successor of the first and last tree nodes with respect to inorder traversal.

Another way of distinguishing a thread link from a structure link is to have a separate boolean flag for each of the left and right pointers.

LPTR	LTHREAD	DATA	RTHREAD	RPTR

LTHREAD = {TRUE} – denote left thread link
LTHREAD = {FALSE} – denote structural link
RTHREAD = {FALSE} – denote structural link
RTHREAD = {TRUE} – denote right thread link

We distinguish thread with structural link by negative address.

Given a threaded tree for a particular order of traversal. It is a relatively simple task to develop algorithm to obtain the predecessor or successor nodes of some particular node P.

The algorithm is given for inorder traversal

Algorithm INS(X)

/* X is address of node in a threaded BT. The algorithm returns the address of its inorder successor. P is the temporary pointer variable. */

Step 1 : [Return the right pointer of the given node if a thread]
P = X–>rchild;
if (rchild–>X) < 0) then
return P;

Step 2 : [Branch left repeatedly until a left thread]
while (p–>lchild)>0 do
P = P–>lchild;

Step 3 : [Return address of successor]
return P;

Algorithm INP(X)

/* For a given X, the address of node in threaded BT. This algorithm returns address of its predecessor is returned */

Step 1 : [Return the left child of a given node of a thread]
P = X–>lchild;
if ((X–>lchild) < 0) then
return P;
Step 2 : [Branch right subtree repeatedly until a right thread]
while (P–>rchild) > 0 then
P = P–>rchild;
Step 3 : [Return address of predecessor]
return P;

The successor subalgorithm can be used repeatedly to traverse the threaded tree in inorder.

Algorithm TINORDER
Step 1 : [initialize]
P = HEAD;
Step 2 : [Traverse threaded tree in inorder]
while TRUE do
P = INS(P);
if P = HEAD then
exit;
else
write (P–>info];

Consider the binary tree T given in Fig. 7.19.

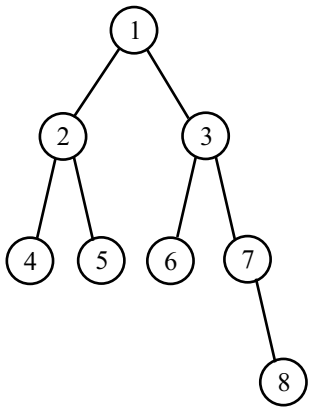


Figure 7.19 *A binary tree*

There are two types of binary tree threading:
- One way threading
  - Right threaded binary tree-thread is placed in the right child of nodes if they are NULL
  - Left threadded binary tree-thread is placed in the left child of nodes if they are NULL.
- Two way threading-thread is placed in the left child of nodes if they are NULL

The one way inorder right threading of T is given in Fig. 7.20. There is a thread in the right child of the node if they are Null.

Inorder sequence

4 2 5 1 6 3 7 8

right child of node 4 points to its successor node i.e. 2.

right child of node 5 points to its successor node i.e. 1. The right child of node 6 points to its successor node i.e., 3.

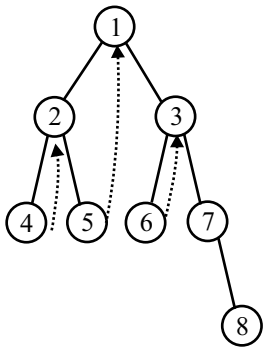


Figure 7.20

Now construct two way inorder threading of T in Fig. 7.21. The NULL pointer if exists in rchild of node is replaced by a thread pointer that points to its inorder successor and NULL pointer if exists in lchild of node is replaced by a thread pointer that points to its inorder predecessor.

The left child of node 5 points to its inorder predecessor, node 2.
The right child of node 5 points to its inorder successor node 1.
Similarly done for the other nodes.

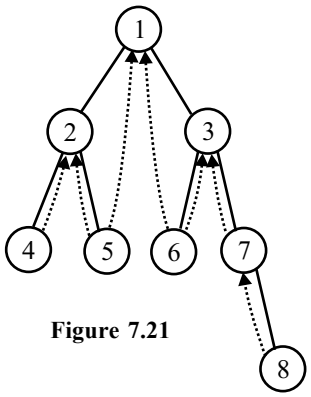


Figure 7.21

Two way threading with header node is given in Fig. 7.22.

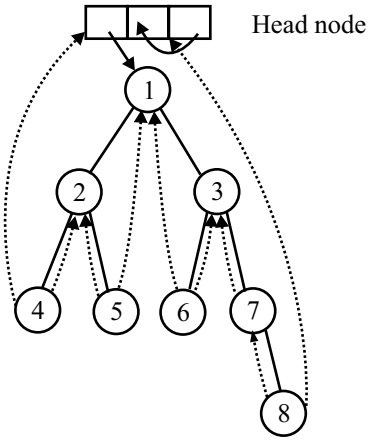


Figure 7.22

Preorder Threaded Tree

Again consider the similar tree for preorder threading (Fig. 7.23).

Predorder sequence
1 2 4 5 3 6 7 8

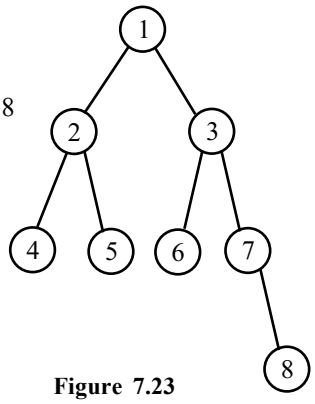


Figure 7.23

One way preorder right threaded tree is given in Fig. 7.24.

The right thread for nodes 4, 5, & 6 is placed as their right child is NULL. The node 4 right thread points to its preorder successor i.e., node 5, and node 5 right thread points to node 3, and node 6 right thread points to node 7.

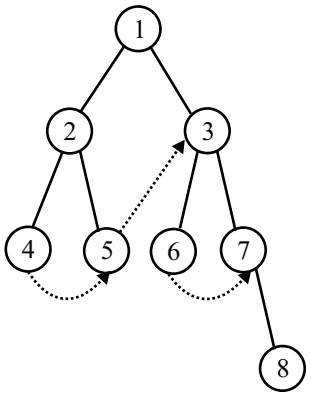


Figure 7.24

Two way preorder threaded tree is given in Fig. 7.25.

The left thread for nodes 4, 5, 6, 7 and 8 is also placed as their left child is null.
The left thread for node 4 points to node 2.
The left thread for node 5 points to node 4.
The left thread for node 6 points to node 3.
The left thread for node 7 points to node 6.
The left thread for node 8 points to node 7.

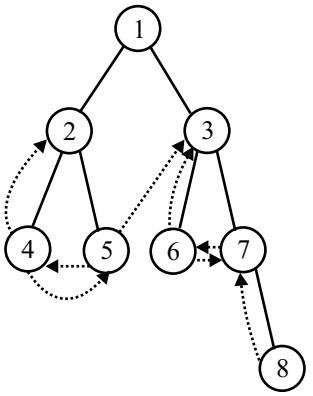


Figure 7.25

Post-Order Threaded Tree

Consider the following tree for post-order threading as in Fig. 7.26.

Post-order sequence
4 5 2 6 8 7 3 1

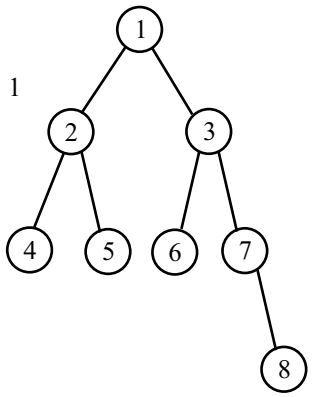


Figure 7.26

One way postorder right threaded tree is given in Fig. 7.27.

The right thread for nodes 4, 5, 6 & 8 is placed as their right child pointer is null.
The right thread for node 4 points to node 5.
The right thread for node 5 points to node 2.
The right thread for node 6 points to node 8.
The right thread for node 8 points to node 7.

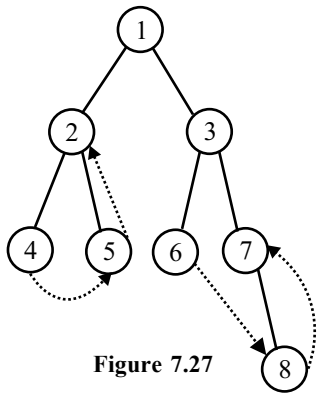


Figure 7.27

Two way postorder threaded tree is given in Fig. 7.28.

The left thread for nodes 5, 6, 8 & 7 is placed as their left child pointer is null.
The left thread for node 5 points to node 4.
The left thread for nude 6 points to node 2.
The left thread for node 8 points to node 6.
The left thread for node 7 points to node 8.

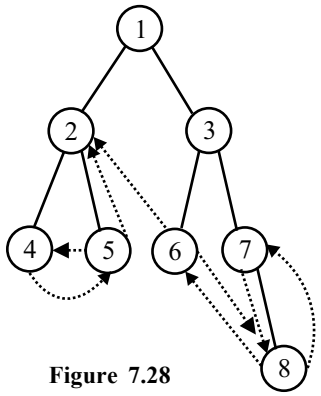


Figure 7.28

7.6 GENERAL TREE AND ITS CONVERSION

A binary tree T' is not a special case of general tree T. The differences are:

1. A binary tree T' may be empty but a general tree is nonempty.
2. Suppose a node N has only one child. Then the child is distinguished as a left child or right child in binary tree T', but no such distinction exists in a general tree T.

The second difference is illustrated in trees T_1 & T_2 in Fig. 7.29

A forest F is defined as an ordered collection of zero or more distinct trees.

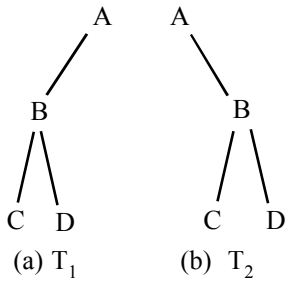


Figure 7.29

Memory Representation of General Tree

1. Suppose T is a general tree. T will be maintained in memory by means of a linked representation, which uses three parallel arrays. INFO, CHILD (or DOWN) and SIBL (or HORZ) and a pointer variable ROOT as follows. First of all, each node N of T will correspond to a location K such that
   - INFO [K] contains the data at node N.
   - CHILD [K] contains the location of the first child of node N.
   - The condition CHILD [K] = NULL indicates that N has no children.
   - SIBL [K] contains the location of the next sibling of node N.
   - The condition SIBL [K] = NULL means that N is the last child of its parent.

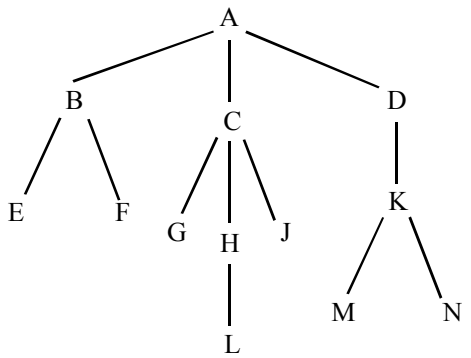


Figure 7.30 *A general tree*

The array representation of the tree in Fig. 7.30 is given below:

root
2

	INFO	CHILD	SIBL
1			
2	A	3	0
3	B	15	4
4	C	6	16
5		13	
6	G	0	7
7	H	11	8
8	J	0	0
9	N	0	0
10	M	0	9
11	L	0	0
12	K	10	0
13		0	
14	F	0	0
15	E	0	14
16	D	12	0

Figure 7.31 *Array representation of the general tree in Fig. 7.30*

A Node May be Declared as a Dynamic Variable as Below:

```
struct treenode{
    int info;
    struct treenode *father;
    struct treenode *son;
    struct treenode *next;
    };
```

If all traversals are from a node to its sons, the father field may be omitted. Figure 7.32 (a) illustrates the representations of the trees of Fig. 7.32 (b).

```
struct treenode{
    int info;
    struct treenode *son;
    struct treenode *next;
    };
```

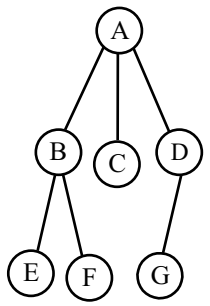


Figure 7.32 (a)

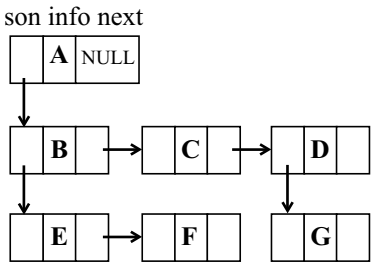


Figure 7.32 (b)

General Tree Traversals

General tree also supports four ways to traversal tree as similar to binary tree.

- Inorder traversal
- Preorder traversal
- Postorder traversal
- Backward Inorder traversal

The function for each traversal is given below:

Function Inorder traversal

```
void tintrav(struct treenode *t)
{
if (t)
{
        tintrav(t–>son);
        printf("\t %d", t–>info);
        tintrav(t–>next);
}
}/* end of tintrav function */
```

Inorder traversal of general tree in Fig. 7.32 is E, F, B, C, G, D, A.

Function Preorder traversal

```
void tpretrav(struct treenode *t)
{
if (t)
{
        printf("\t %d", t–>info);
        tpretrav(t–>son);
        tpretrav(t–>next);
}
}/* end of tpretrav function */
```

Preorder traversal of general tree in Fig. 7.32 is A, B, E, F, C, D, G

Function Postorder traversal

```
void tpostrav(struct treenode *t)
{
if (t)
{
        tpostrav(t–>son);
        tpostrav(t–>next);
        printf("\t %d", t–>info);
}
} /* end of tpostrav function */
```

Postorder traversal of general tree in Fig. 7.32 is E, F, G, D, C, B, A

Function Backward inorder

```
void tbintrav(struct treenode *t)
{
if (t)
{
        tbintrav(t–>next);
        printf("\t %d", t–>info);
        tbintrav(t–>son);
}
}/* end of tbintrav function */
```

Backward inorder traversal of general tree in Fig. 7.32 is A, D, G, C, B, F, E

A complete 'C' program to create, insert, search traversals and display general tree is given below:


```
/* Program tree.cpp construct a general tree. Program consists of insert, search, display, searchinfo,
and recursive traversal functions.  */
#include<stdio.h>
#include<conio.h>

struct  treenode
{
        int info;
        struct  treenode  *son;
        struct treenode *next;
};

/* Search function, find the treenode by its value in the tree */
struct treenode *search(struct treenode *root,int val,int nnum)
{
        struct treenode *ret  =  NULL;
        if(root–>info  =  = val)
                return root;
        if(root–>son! = NULL)
        {
                nnum++;
                ret  =  search(root–>son,val,nnum + 1);
        }
        if(root–>next! = NULL)
        {
                nnum++;
                ret  =  search(root–>next,val,nnum + 1);
        }
return ret;
} /* end of function */
/* searchinfo function, find the treenode by its value in the tree and return
```

```
1 for successful search and 0 for unsuccessful search*/
int searchinfo(struct treenode *root,int val,int nnum)
{
        int ret = 0;
        if(root–>info = = val)
        {
                printf("Treenode found");
                return 1;
        }
        if(root–>son! = NULL)
        {
                nnum++;
                ret = searchinfo(root–>son,val,nnum+1);
        }
        if(root–>next! = NULL)
        {
                nnum++;
                ret = searchinfo(root–>next,val,nnum+1);
        }
 return ret;
} /* end of function */

/* Function insert the node into its proper position */
void insert(struct treenode *root)
{
        int finfo,val;
        struct treenode *ptr = NULL,*tmp;
        printf("\n\n Enter the Father info:");
        scanf("%d",&finfo);
        ptr = search(root,finfo,0);
        if(ptr! = NULL)
        {
                if(ptr–>son! = NULL)
                {
                        ptr = ptr–>son;
                        while(ptr–>next! = NULL)
                                ptr = ptr–>next;

                        tmp = (struct treenode *)malloc(sizeof(struct treenode));
                        printf("Enter the Node Info:");
                        scanf("%d",&val);
                        tmp–>info = val;
                        tmp–>next = NULL;
                        tmp–>son = NULL;
```

```
                    ptr–>next = tmp;
                }
            else
                {
                    tmp = (struct treenode *)malloc(sizeof(struct treenode));
                    printf("Enter the Node Info:");
                    scanf("%d",&val);
                    tmp–>info = val;
                    tmp–>next = NULL;
                    tmp–>son = NULL;
                    ptr–>son = tmp;
                }
        }
        else
        {
            printf(" Node not found!!!");
            getch();
        }
}   /* end of insert */

/* Function displays the nodes in the tree */
void display(struct treenode *root, struct treenode *father,int nnum)
{
        if(root! = NULL)
        {
            if(father! = NULL)
                printf(" father node = %d: child node = %d
\n",father–>info,root–>info);
            else
                printf(" ROOT = %d \n",root–>info);

        }
        if(root–>son! = NULL)
        {
            nnum++;
            display(root–>son,root,nnum);
        }
        if(root–>next! = NULL)
            display(root–>next,father,++nnum);
} /* end of display */

/* function traverse the nodes of tree as inorder */
void tintrav(struct treenode *t)
{
```

```
if (t)
{
        tintrav(t–>son);
        printf("\t %d", t–>info);
        tintrav(t–>next);
}
}/* end of tintrav function */

/* function traverse the nodes of tree as preorder */
void tpretrav(struct treenode *t)
{
if (t)
{
        printf("\t %d", t–>info);
        tpretrav(t–>son);
        tpretrav(t–>next);

}
}/* end of tpretrav function */

/* function traverse the nodes of tree as postorder */
void tpostrav(struct treenode *t)
{
if (t)
{

        tpretrav(t–>son);
        tpretrav(t–>next);
        printf("\t %d", t–>info);

}
} /* end of tpostrav function */

/* function traverse the nodes of tree as backward inorder */
void tbintrav(struct treenode *t)
{
if (t)
{
        tpretrav(t–>next);
        printf("\t %d", t–>info);
        tpretrav(t–>son);
}
}/* end of tbintrav function */

/* main function */
void main()
{
        struct treenode *root;
```

```
int ch,val;
root  =  NULL;  /* empty tree */
do
{
printf("\n 1. Create Root.");
printf("\n 2. Exit.");
printf("\n\n Enter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
        clrscr();
        printf("Enter root info:");
        scanf("%d",&val);
        root = (struct treenode *)malloc(sizeof(struct treenode));
        root–>info = val;
        root–>next = NULL;
        root–>son = NULL;
        do
        {
        clrscr();
        printf ("\n General Tree Operations");
        printf("\n 1. Insert node.");
        printf("\n 2. Search node.");
        printf("\n 3. Display Tree.");
        printf("\n 4. Inorder Traversal.");
        printf("\n 5. Preorder Traversal.");
        printf("\n 6. Postorder Traversal.");
        printf("\n 7. Backward Inorder Traversal.");
        printf("\n 8. Exit.");
        printf("\n\n Enter your choice:");
        fflush (stdin);
        scanf("%d",&ch);
        switch(ch)
        {
        case 1:
                insert(root);
                break;
        case 2:
                printf("\n\n Enter the node info:");
                scanf("%d",&val);
                        if(!searchinfo(root,val,0))
```

```
                                        printf("\nNode not found");
                        getch();
                        break;
                case 3:
                        display(root,NULL,0);
                        getch();
                        break;
                case 4:
                        tintrav(root);
                        getch();
                        break;
                case 5:
                        tpretrav(root);
                        getch();
                        break;
                case 6:
                        tpostrav(root);
                        getch();
                        break;
                case 7:
                        tbintrav(root);
                        getch();
                        break;
                case 8:
                        break;
                default:
                        printf("\n Wrong choice!!!");
                        getch();
                }
            }while(ch! = 8);
        ch = 2;
        break;
        case 2:
                break;
        default:
                printf("\nWrong choice!!!");
                break;
    } /* end of outer switch */
}while(ch! = 2);
} /* end of main */
```

Conversion of General Tree into Binary Tree (BT)

Suppose T is a general tree. Then we may assign a unique BT T^1 to T as follows. First of all nodes of the binary tree T^1 will be same as the nodes of the general tree T and the root of T^1 will be the root of T. Let N be an arbitrary node of the BT T^1. Then the left child of N will be the first child of the node N in the general tree and right child of T^1 will be the next sibling of N in the general tree T.

The Fig. 7.33 below is binary tree of general tree of Fig. 7.30.

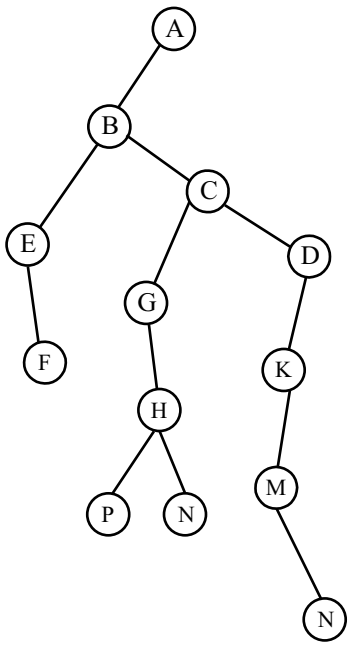


Figure 7.33 *Binary tree for general tree of Fig. 7.30*

Converting Forest (i.e., Set of Trees) into Binary Tree

Consider son as corresponding to the left pointer of a binary tree node and next as corresponding its right pointer. This method actually represents a general ordered tree as binary tree.

Consider a forest in Fig. 7.34.

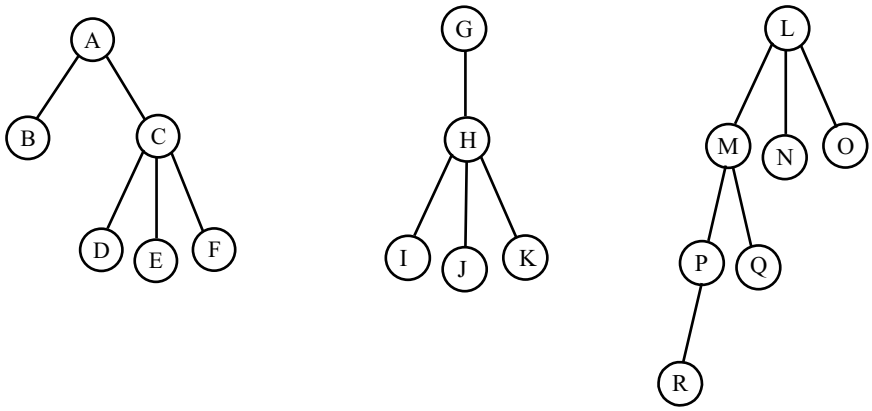


Figure 7.34

The binary tree for corresponding forest is given below in Fig. 7.35.

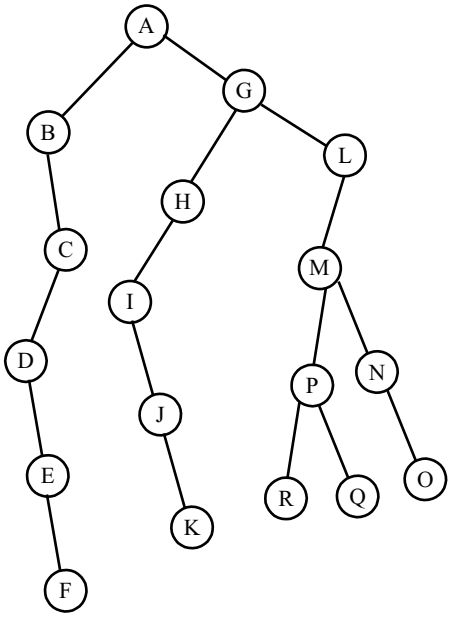


Figure 7.35 *Binary tree*

7.7 BINARY SEARCH TREE (BST)

The binary search tree properties are given below:

- It is Binary tree
- All items in the left subtree are less than the root
- All items in the right subtree are greater than the root
- Each subtree is also a binary search tree (i.e., recursive definitions)
- Does not allow duplicate values.

Definition: A binary search tree, BST, is an ordered binary tree T such that either it is an empty tree or

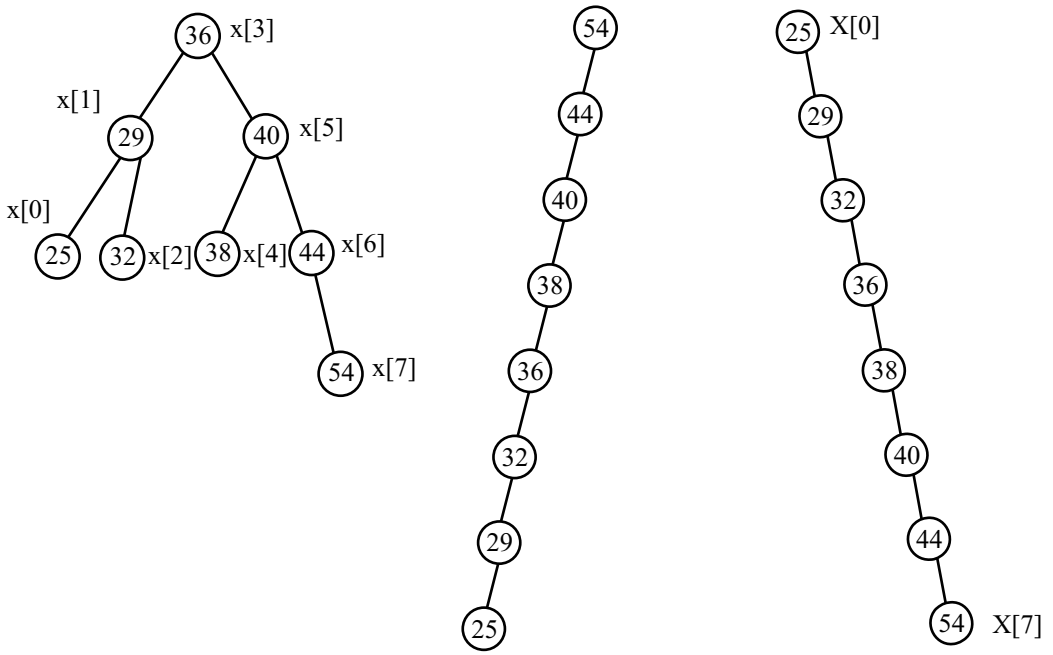
- each element value in its left subtree is less than the root value.
- each element value in its right subtree is greater than or equal to the root value, and
- left and right subtrees are again binary search trees.

The inorder traversal of such a binary tree gives the set elements in ascending order.

A sorted array can be produced from a BST by traversing the tree in inorder and inserting each element sequentially into array as it is visited. It may be possible to construct many BSTs from given sorted array. Some of the possible BSTs with different roots are given below in Fig. 7.36.

Consider a sorted array

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
x[ ] =	25	29	32	36	38	40	44	54



(*a*) BST representation BST balanced (*b*) BST left heavy unbalanced (*c*) BST right heavy unbalanced

Figure 7.36

Insertion in BST

The following algorithm searches as BST and inserts a new key into the tree if the search is unsuccessful into Fig. 7.36 (*a*).

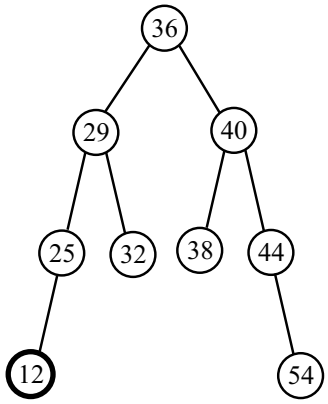


Figure 7.36 (*d*) *Insert a new element whose value is 12*

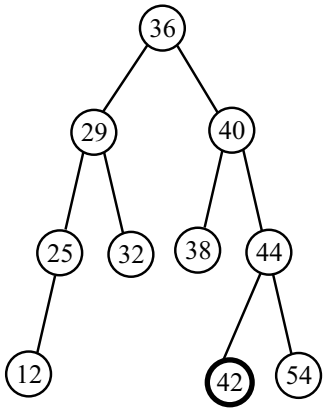


Figure 7.36 (*e*) *Insert a new element whose value is 42*

Note that after a new key is inserted, the tree retains the property of being sorted in an inorder traversal.

Deleting from a BST

(*i*) Delete node whose value is 32 in BST of Fig. 7.36 (*e*) is given below in Fig. 7.37(*i*)

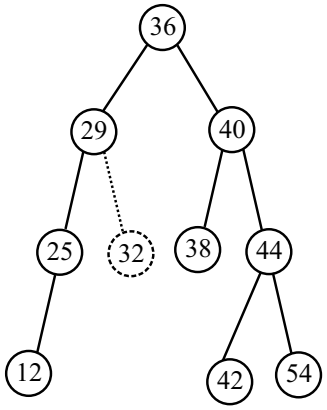


Figure 7.37 (*i*)

(*ii*) Delete node whose value is 40 in BST of Fig. 7.36 (*e*)

First replace the deleted node to its inorder successor i.e., 42 and then delete the node.

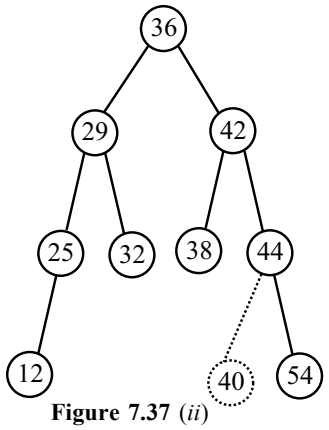


Figure 7.37 (*ii*)

(*iii*) Delete the node whose value is 25 in Fig. 7.36 (*e*) of BST
Just adjust the node 12 in the position of 25 and remove it.

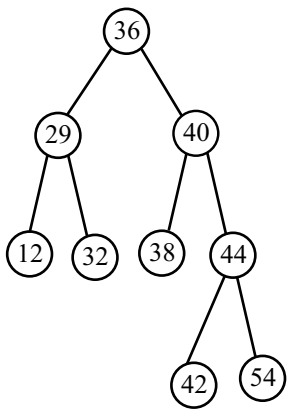


Figure 7.37 (*iii*)

Example 1. Suppose the following list of letters is inserted in order into an empty BST

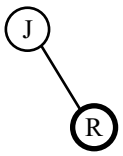
J R D G T E M H P A F Q

(*a*) Find the final tree T (*b*) find the inorder traversal of T.

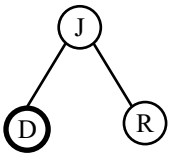
Solution (*a*) Insert the nodes one after the other, maintaining BST property as given in Fig. 7.38 as follows:

(*i*) Insert node J in empty BST  J becomes root of BST

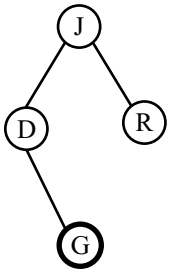
(*ii*) Insert node R



(*iii*) Insert node D

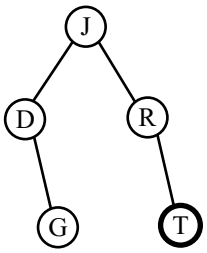


(*iv*) Insert node G

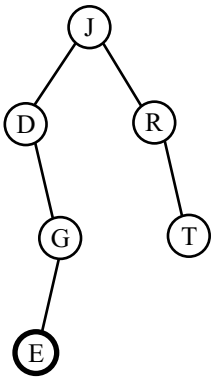


(***Figure 7.38–contd...***)

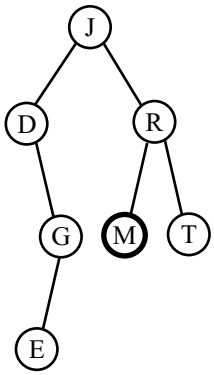
(*v*) Insert node T



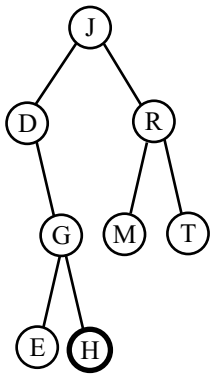
(*vi*) Insert node E



(*vii*) Insert node M

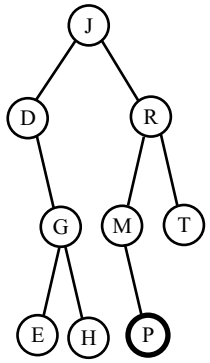


(*viii*) Insert node H

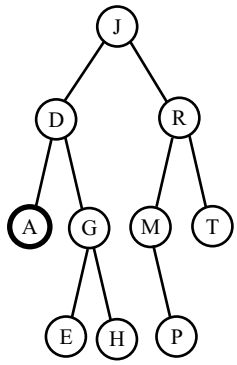


(***Figure 7.38–contd...***)

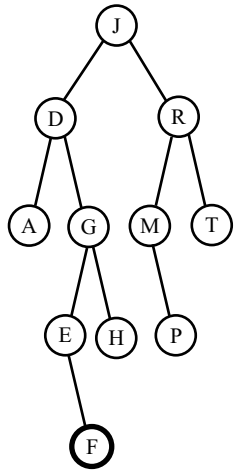
(*ix*) Insert node P



(*x*) Insert node A



(*xi*) Insert node F



(***Figure 7.38–contd...***)

(*xii*) Insert node Q

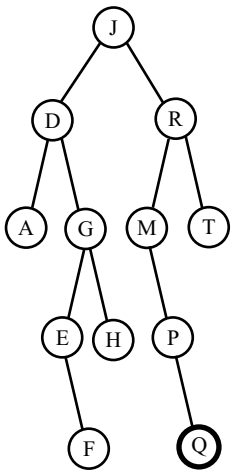


Figure 7.38 *Construction of binary search tree*

(*b*) The inorder traversal of T follows

A D E F G H J M P Q R T

Example 2. Suppose the following numbers are inserted in order into an empty BST

50 25 75 22 40 60 80 90 15 30

Draw the tree T

Solution Numbers are inserted one by one maintaining BST property as given below in Fig. 7.39.

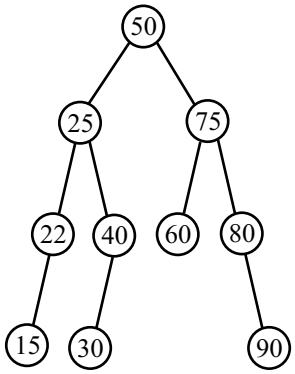


Figure 7.39 *Binary search tree T*

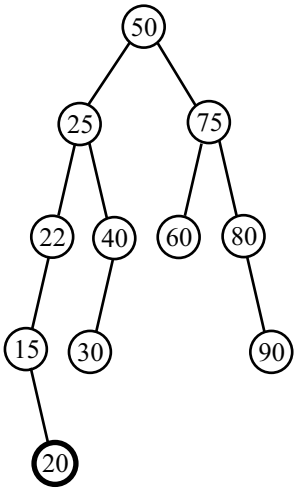
Applying the following operation on original tree T (That is, the operations are applied independently, not successively as given in Fig. 7.40).

(*a*) Node 20 is added

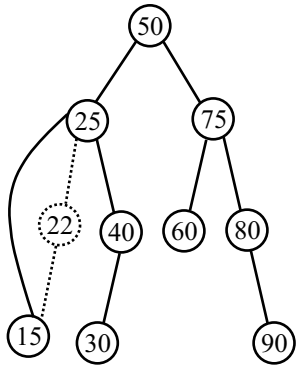
(*b*) Node 22 is deleted

(*c*) Node 88 is added

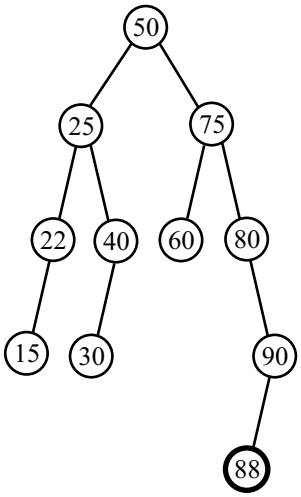
(*d*) Node 75 is deleted
(*e*) Node 35 is added
(*f*) Node 50 is deleted



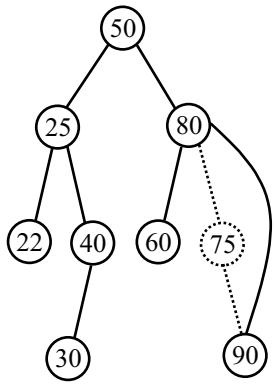
(*a*)



(*b*)



(*c*)



(*d*)

(*Figure 7.40–contd...*)

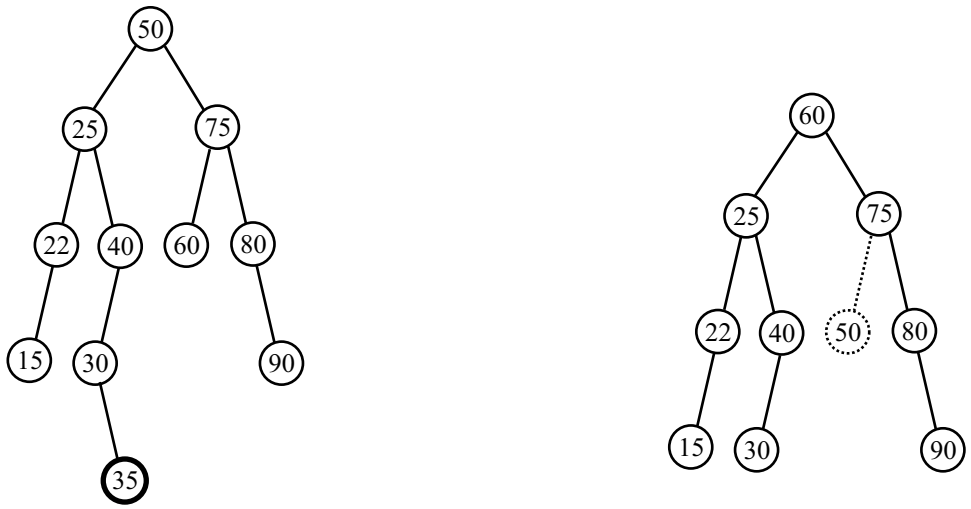


Figure 7.40 *Operations of binary search tree of Fig. 7.39*

Searching a Binary Search Tree

Since the definition of a binary search tree is recursive, it is easiest to describe a recursive search method.

Suppose we wish to search for an element with key x. We begin at the root. If the root is NULL, then the search tree contains no nodes and the search is unsuccessful. Otherwise, we compare x with the key in the root. If x equals this key, then the search terminates successfully. If x is less than the key in the root, then search in the left subtree. If x is larger than the key in the root, only the right subtree needs to be searched.

The recursive algorithm for searching is given below:

```
        Algorithm BTRECSRC(p, x)
        /* recursive search of a binary search tree and p denotes root of the tree */
Step 1 : if (p = NULL) then
             return –1;
Step 2 : else if (x = p –>Info) then
             return p;
Step 3 : else if (x < p–>Info) then
             return(BTRECSRC(p–>lchild, x);
             else
             return(BTRECSRC(p–>rchild, x);
```

The non–recursion algorithm for binary search is given below:

```
        Algorithm BTSRCH(p, x)
        /* Iterative search of a binary search tree and p denotes root of the tree */
Step 1 : repeat step2 while (p ! = NULL) and (p–>Info ! = x)
Step 2 :     if (p–>Info > x) then
                 p = p–>lchild;
             else
```

```
                    p = p–>rchild;
            [end of while loop at Step 1]
```
Step 3 : return (p);

The function btsrch searches a node in the binary search tree

```
/* The btsrch function finds the given node in the binary tree */
void btsrch(struct nodetype *p, int x)
{
  while((p! = NULL) &&(p–>info ! = x))
  {
   if (p–>info >x)
          p = p–>lchild;
  else
          p = p–>rchild;
  }
if (p = = NULL)
    printf("\n Search unsuccessful");
else
  printf("\n Search successful at node address %d",p);
}
```

Insertion into a Binary search tree

To insert a new element x, we must first verify that its key is different from those of existing nodes.

Algorithm BTREEINS(x)

Step 1 : [initialize variables]
Set found = 0; set p = root;
Step 2 : repeat step 3 while (p ! = NULL) and (!found)
Step 3 : parent = p;
if (p–>Info = x) then
Set found = 1;
else if (x < p–>Info) then
Set p = p–>lchild;
else
Set p = p–>rchild;
[end of while loop at Step 2]

Step 4 : if (!found) then
Set p = allocate memory for a node of binary tree
Set p–>lchild = NULL;
Set p–>Info = x;
Set p–>rchild = NULL;
Step 5 : if (root ! = NULL) then
if (x <parent–>Info) then

```
                    Set parent–>lchild = p;
                else
                    Set parent–>rchild = p;
                [End of If Structure]
            else
                Root = p;
            [End of If Structure of Step 5]
        [End of If Structure of Step 4]
```

Step 6 : end BTREEINS

The function for insertion into binary search tree is given below:

```
/* The function btreeins inserts a node value x in the binary search tree */
void btreeins(int x)
{
struct nodetype *p, *parent;
int found = 0;
p = root;
while((p! = NULL) &&(!found))
{
   parent = p;
   if (x = = p–>info)
            found = 1;
 else if (x < p–>info )
            p = p–>lchild;
 else
            p = p–>rchild;
}
if (!found)
{
p = (struct nodetype *)malloc(sizeof(struct nodetype));
p–>info = x;
p–>lchild = NULL;
p–>rchild = NULL;
if (root! = NULL)
            {
            if ( x<parent–>info)
                parent–>lchild = p;
            else
                parent–>rchild = p;
            }
else
               root = p;
}
} /* End of function btreeins */
```

Deletion of a node from a Binary Search Tree

To delete an element x, we must first verify that its key exists into the binary search tree.

Case A:

1. When the node to be deleted is a leaf node of the binary search tree.
2. When the node to be deleted has a single child, either left child or right child.

Case B:

1. When the nodes to be deleted have both left and right children.

The algorithm BTREEDEL calls the algorithm DELCASEA and DELCASEB. The pointer variable dnode and pardnode denotes the location of the deleted node and its parent respectively. The pointer variable succ and parsucc denotes the inorder successor of the node to be deleted and parent of the successor respectively.

Algorithm BTREEDEL(x)

Step 1 : call btsrch(x);
Step 2 : if(dnode = NULL)then
write(" Not found"); and exit;
Step 3 : if (dnode–>rchild ! = NULL) and (dnode–>lchild! = NULL) then
Call delcaseB();
else
Call delcaseA(dnode, pardnode);
Step 4 : free(dnode);

Function btreedel

```
void btreedel(int x)
{
btsrch(x);
if(dnode = = NULL)
        {
        printf("\n not found");
        return;
        }
if ((dnode–>rchild ! = NULL) && (dnode–>lchild! = NULL))
        delcaseB();
else
   delcaseA(dnode, pardnode);
free(dnode);
} /* End of function btreedel */
```

Algorithm BTSRCH(x)

Step 1 : Set dnode = root;
Step 2 : while(dnode! = NULL) and (dnode–>info ! = x)
Step 3 : Set pardnode = dnode;
if (dnode–>info >x) then
Set dnode = dnode–>lchild;

```
            else
                    Set dnode = dnode–>rchild;
[End of while loop at step 2]
```

Function btsrch

```
/* The btsrch function searches a node in the Binary Search Tree */
void btsrch(int x)
{
dnode = root;
   while((dnode! = NULL) &&(dnode–>info ! = x))
   {
 pardnode = dnode;
 if (dnode–>info >x)
            dnode = dnode–>lchild;
 else
            dnode = dnode–>rchild;
 }
} /* End of function btsrch */
```

Algorithm DELCASEA(dnode, pardnode)

Step 1 : if (dnode–>lchild = NULL) and (dnode–>rchild = NULL) then
Set child = NULL;
else if (dnode–>lchild)! = NULL then
Set child = dnode–>lchild;
else
Set child = dnode–>rchild;

Step 2 : if (pardnode ! = NULL) then
if (dnode = pardnode–>lchild)
Set pardnode–>lchild = child;
else
Set pardnode–>rchild = child;
else
root = child;

Step 3 : end DELCASEA

Function delcaseA

```
/* The delcaseA function deletes the node with one child or as leaf node */
void delcaseA(struct nodetype *dnode,struct nodetype *pardnode)
{
struct nodetype *child;
if ((dnode–>lchild = = NULL) &&(dnode–>rchild = = NULL))
        child = NULL;
else if ((dnode–>lchild)! = NULL)
                child = dnode–>lchild;
```

```
        else
                child = dnode–>rchild;
if (pardnode ! = NULL)
        if (dnode = = pardnode–>lchild)
                pardnode–>lchild = child;
        else
                pardnode–>rchild = child;
else
                root = child;
}/* End of delcaseA function */
```

Algorithm DELCASEB

Step 1 : [Initialize pointers]
Set p = dnode–>rchild; and set q = dnode;
Step 2 : while (p–>lchild) ! = NULL
Step 3 : Set q = p; and p = p–>lchild;
[End of while loop at step 2]
Step 4 : Set succ = p; and parsucc = q;
Srep 5 : call DELCASEA(succ, parsucc);
Step 6 : if (pardnode ! = NULL) then
if (dnode = pardnode–>lchild) then
pardnode–>lchild = succ;
else
pardnode–>rchild = succ;
else
root = succ;
Step 7 : Set succ–>lchild = dnode–>lchild; and succ–>rchild = dnode–>rchild;
Step 8 : end DELCASEB

Function delcaseB

```
/* The delcaseB function deletes the node with two child  */
void delcaseB()
{
struct nodetype *p,*q, *succ, *parsucc;
p = dnode–>rchild;
q = dnode;
while ((p–>lchild) ! = NULL)
{
        q = p;
        p = p–>lchild;
}
succ = p;
parsucc = q;
delcaseA(succ, parsucc);
```

```
if (pardnode ! = NULL)
        if (dnode = = pardnode–>lchild)
                pardnode–>lchild = succ;
        else
        pardnode–>rchild = succ;
else
        root = succ;
succ–>lchild = dnode–>lchild;
succ–>rchild = dnode–>rchild;
}/* End of delcaseB function */
```

The height function finds the height of the binary search tree as follows:

Function height

```
/* The height function returns the height of the binary tree */
int height(struct nodetype *btreeh, int level)
{
if (btreeh)
{
        height(btreeh–>lchild, level+1);
        height(btreeh–>rchild, level+1);
}
if ( level> m)
        m = level;
return (m–1);
}
```

A complete 'C' program to perform insertion, deletion, searching, tree traversals is given below with the help of the respective functions.

```
/* Create Binary TREE  and Find height of the tree, and search the tree*/
/* bstreeop.cpp performs various operations such as insert, delete, search,
find height of tree, and tree traversals */
```

TREE/BSTREEOP.CPP

```
   # include<stdio.h>
   # include<malloc.h>
struct  nodetype
{
        int info;
        struct nodetype *lchild;
        struct nodetype *rchild;
}*root,*dnode,  *pardnode;
int k = 0, m = 0;
void btreeins(int);
```

```
void btsrch(struct nodetype *, int );
void display(struct nodetype *, int);
void inorder(struct nodetype *);
void postorder(struct nodetype *);
void preorder(struct nodetype *);
void btsrchn(int);
void delcaseA(struct nodetype *,struct nodetype *);
void delcaseB();
void btreedel(int);

int height(struct nodetype *,int);
void main()
{
int n = 0,ch,x, dn;
clrscr();
root = NULL; /* empty tree */
do{

printf("\n Enter choice for tree");
printf("\n 1.Create tree/Insertion");
printf("\n 2.Display tree");
printf("\n 3.Height of tree");
printf("\n 4.Search the desired value");
printf("\n 5.Inorder traversing of tree");
printf("\n 6.Postorder traversing of tree");
printf("\n 7.Preorder traversing of tree");
printf("\n 8. Deletion");
printf("\n 9 Exit");
printf("\n Enter choice for tree operation");
scanf("%d",&ch);

switch(ch)
{
   case 1:
        while(n! = –1)
        {
              printf("\n Enter node, type(–1) to terminate");
              scanf("%d",&n);
              if (n! = –1)
                  btreeins(n);
        }
    n = 0;
        break;
```

```
 case 2:
                display(root,0);
                break;
 case 3:
                printf("\n height is %d",height(root,0));
                break;
 case 4:
                printf("\n  Enter node to be search");
                scanf("%d",&x);
                btsrch(root,x );
                break;
 case 5:
                inorder(root);
                break;
 case 6:
                postorder(root);
                break;
 case 7:
                preorder(root);
                break;
 case 8:        printf("\n Enter the node to be deleted");
                scanf("%d",&dn);
                btreedel(dn);
                        printf("\n Binary Search Tree after Deletion of node %d is as follows",dn);
                display(root,1);
                break;
 case 9:        break;
   default:   printf("\n Wrong choice !");
        }
}while(ch! = 9);

} /* end of the main function */

/* The function btreeins inserts a node value x in the binary tree */
void btreeins(int x)
{
struct nodetype *p, *parent;
int found = 0;
p = root;
while((p! = NULL) &&(!found))
{
 parent = p;
 if (x = = p–>info)
          found = 1;
else if (x < p–>info)
```

```
        p = p–>lchild;
 else
        p = p–>rchild;
}
if (!found)
{
p = (struct nodetype *)malloc(sizeof(struct nodetype));p–>info = x;
p–>lchild = NULL;
p–>rchild = NULL;
if (root! = NULL)
        {
        if ( x<parent–>info)
                parent–>lchild = p;
        else
                parent–>rchild = p;
        }
else
        root = p;
}
} /* End of function btreeins */
/* The btsrch function finds the given node in the binary tree */
void btsrch(struct nodetype *p, int x)
{
 while((p! = NULL) &&(p–>info ! = x))
 {
 if (p–>info >x)
        p = p–>lchild;
 else
        p = p–>rchild;
 }
if (p = = NULL)
   printf("\n Search unsuccessful");
else
   printf("\n Search successful at node address %d",p);
}
/* The height function returns the height of the binary tree */
int height(struct nodetype *btreeh, int level)
{
if (btreeh)
{
        height(btreeh–>lchild, level+1);
        height(btreeh–>rchild, level+1);
```

```
}
if ( level> m)
        m = level;
return (m–1);
}
void inorder(struct nodetype *bt)
{
if (bt)
{
        inorder(bt–>lchild);
        printf(“  %d ”,bt–>info);
        inorder(bt–>rchild);
}
}
void preorder(struct nodetype *bt)
{
if (bt)
{
        printf(“  %d”,bt–>info);
        preorder(bt–>lchild);
        preorder(bt–>rchild);
}
}
void postorder(struct nodetype *bt)
{
if (bt)
{
        postorder(bt–>lchild);
        postorder(bt–>rchild);
        printf(“  %d”,bt–>info);
}
}
/* Display function displays the binary tree in tree forms */
/* The function btreedel deletes a node from binary search tree */
void btreedel(int x)
{
btsrchn(x);
if(dnode = = NULL)
        {
        printf(“\n not found”);
```

```
        return;
        }
if ((dnode–>rchild ! = NULL) && (dnode–>lchild! = NULL))
        delcaseB();
else
   delcaseA(dnode, pardnode);
free(dnode);
} /* End of function btreedel */

/* The btsrch function searches a node in the Binary Search Tree */
void btsrchn(int x)
{
dnode = root;
 while((dnode! = NULL) &&(dnode–>info ! = x))
 {
 pardnode = dnode;
 if (dnode–>info >x)
        dnode = dnode–>lchild;
 else
        dnode = dnode–>rchild;
 }
} /* End of function btsrch */

/* The delcaseA function deletes the node with one child or as leaf node */
void delcaseA(struct nodetype *dnode,struct nodetype *pardnode)
{
struct nodetype *child;
if ((dnode–>lchild = = NULL) &&(dnode–>rchild = = NULL))
        child = NULL;
else if ((dnode–>lchild)! = NULL)
                child = dnode–>lchild;
        else
                child = dnode–>rchild;
if (pardnode ! = NULL)
        if (dnode = = pardnode–>lchild)
                        pardnode–>lchild = child;
        else
                pardnode–>rchild = child;
else
                root = child;
}/* End of delcaseA function */

/* The delcaseB function deletes the node with two child  */
void delcaseB()
```

```
{
struct nodetype *p,*q, *succ, *parsucc;
p = dnode–>rchild;
q = dnode;
while ((p–>lchild) ! = NULL)
{
        q = p;
        p = p–>lchild;
}
succ = p;
parsucc = q;
delcaseA(succ, parsucc);

if (pardnode ! = NULL)
        if (dnode = = pardnode–>lchild)
                pardnode–>lchild = succ;
        else
        pardnode–>rchild = succ;
else
        root = succ;
succ–>lchild = dnode–>lchild;
succ–>rchild = dnode–>rchild;
}/* End of delcaseB function */

void display(struct nodetype *start, int t)
{
int i = 1;
struct nodetype *ptr  =  start;
if (ptr)
{
display (ptr–>rchild,t+1);
printf("\n");
for(i = 0;i< = t;i++)
        printf("  ");

printf("%d",ptr–>info);
display (ptr–>lchild, t+1);
}
}
```

Binary Search Trees vs. Arrays

(*i*) Complexity of searching $O(\log_2 N)$.

(*ii*) Better insertion time: $O(\log_2 N)$ vs. $O(N)$

(*iii*) Better deletion

(*iv*) What is worse?

BST requires more memory space as two pointer references left and right child for each data element.

Applications of Binary Search Trees

- Sorting: We can sort data by reading it, item by item, and constructing a binary search tree as we go
- When we have read all the data, we output it by doing inorder traversal of the tree
- If there is any possibility that the data items are nearly sorted already, or nearly in reverse order, then it is important to use an AVL tree, otherwise we end up with a very unbalanced tree!
- The time complexity of this method of sorting is, in general, $O(n\log_2 n + n)$ i.e., $O(n\log n)$.

7.8 HEIGHT BALANCED TREES: AVL

- A balanced tree is a binary search tree whose every node above the last level has non–empty left and right subtree.
- Since the number of nodes in a complete binary tree of height h is $2^{h+1} - 1$, a binary tree of n elements is balanced if: $2^h - 1 < n \leq 2^{h+1} - 1$.

Balanced trees give the best search/insert times for a given number of nodes

(*i*) Finding an element, inserting, and removing in a balanced tree containing n elements are $O(\log n)$ operations

(*ii*) Binary search trees can become unbalanced and, in the worst case, these operations then become $O(n)$

(*iii*) There are algorithms for balancing binary search trees — they tend not to be very useful because we usually need to keep the tree balanced when we do insertions and deletions, rather than balancing it once and for all.

A type of binary search tree which is nearly as good as a balanced tree for time complexity of the operations, and whose structure we can maintain as insertions and deletions proceed, is the **AVL tree.**

AVL trees are named after the Russian mathematicians G.M. Adelson–Velskii and E.M. Landis, who discovered them in 1962.

An AVL tree is a binary search tree in which the heights of the left and right *subtrees* of the root differ by at most 1, and in which the left and right subtrees of the root are again AVL trees

Balance Factor (bf) = $H_L - H_R$

An empty binary tree is an AVL tree. A nonempty binary tree T is an AVL tree iff given $|H_L - H_R| <= 1$. $H_L - H_R$ is known as the *balance factor* (bf) and for an AVL tree the balance factor of a node can be either 0, 1, or –1.

An AVL search tree is a binary search tree which is an AVL tree but vice–versa is not true.

AVL search trees like binary search trees are represented using a linked representation. However, every node consists of its balance factor. Fig. 7.41 shows the representation of an AVL tree. The number against each node represents its balance factor. The node indicates node number not key.

```
struct nodetype
    {
        int info;
        int bf;
        struct nodetype *lchild;
        struct nodetype *rchild;
    }*node;
```

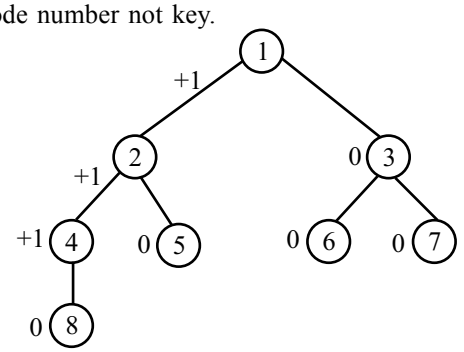


Figure 7.41(*a*) **:** *AVL tree with balancing factor*

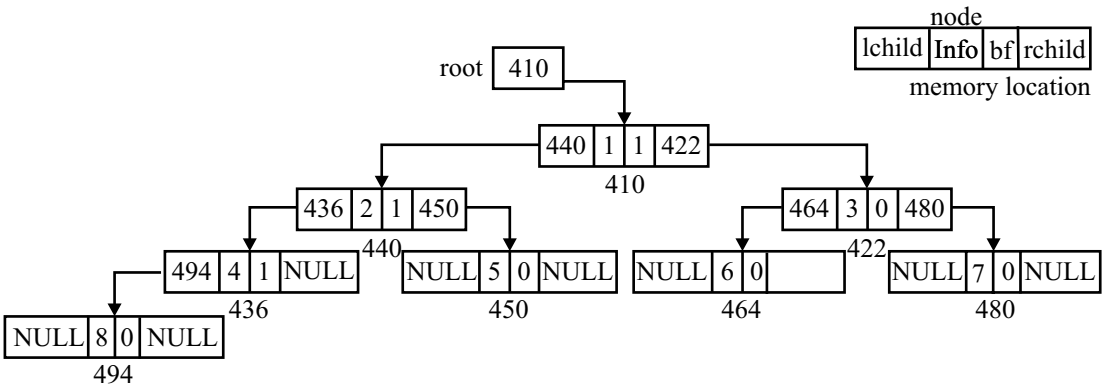


Figure 7.41 (*b*) *Linked representation of binary tree (Empty pointer has null value)*

Searching an AVL Search Tree

Searching an AVL search tree for an element is exactly similar to the method used in binary search tree.

Insertion in an AVL Search Tree

Inserting an element into an AVL search tree in its first phase is similar to that of the binary search tree. However, if after insertion of the element, the balance factor of any tree is affected so as to render the binary search tree unbalanced, we resort to techniques called *Rotations* to restore the balance of the search tree.

To perform rotations it is necessary to identify a specific node A whose BF(A) is neither 0, 1 –1, and which is the nearest ancestor to the inserted node on the path from the inserted node to the root. This implies that all nodes on the path from the inserted node to A will have their factors to be either 0, 1, or –1. The rebalancing rotations classified as LL, LR, RR are illustrated below, based on the position of the inserted node with reference to A.

LL rotation: Inserted node is in the left subtree of left subtree of node A

RR rotation: Inserted node is in the right subtree of right subtree of node A

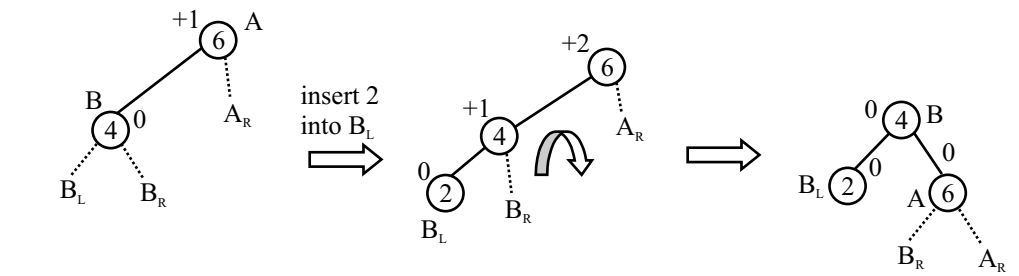
LR rotation: Inserted node is in the right subtree of left subtree of node A

RL rotation: Inserted node is in the left subtree of right subtree of node A

Each of the rotations is explained with an example.

(i) LL rotation

The new element X is inserted in the left subtree of left subtree of A, the closest ancestor node whose BF(A) becomes +2 after insertion. To rebalance the search tree, it is rotated so as to allow B to be the root with B_L and A to be its left subtree and right child, and B_R and A_R to be the left and right subtrees of A. Observe how the rotation results in a balanced tree (See Fig. 7.42).



Initially, B_L : Left subtree of B is NULL, B_R : Right subtree of B is NULL, A_R : Right subtree of A is NULL and h : Height is 1. The pointer movement is done as below:

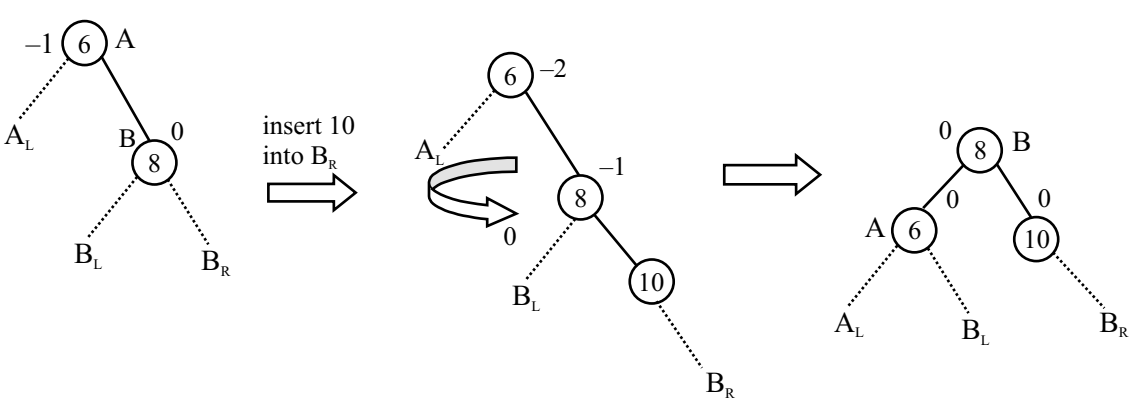
A–>lchild = B–>rchild

B–>rchild = A

Figure 7.42 *LL rotation*

(ii) RR rotation

Here the new element X is in the right subtree of A. The rebalancing rotation pushes B up to the root with A as its left child and B_R as its right subtree, and A_L and B_L as the left and right subtrees of A. Observe the balanced height after the rotation (See Fig. 7.43).



Initially, B_L : Left subtree of B is NULL, B_R : Right subtree of B is NULL, A_L : Left subtree of A is NULL and h : Height is 1. The pointer movement is done as below.

A–>rchild = B–>lchild

B–>lchild = A

Figure 7.43 *RR rotation*

(iii) LR and RL rotations

The balancing methodology of LR and RL rotations are similar in nature but are mirror images of one another.

In this case, the BF values of nodes A and B after balancing are dependent on the BF value of node C after insertion.

IF BF (C) = 0 after insertion then BF(A) = BF(B) = 0, after rotation

IF BF (C) = –1 after insertion then BF(A) = 0, BF(B) = 1, after rotation

IF BF (C) = 0 after insertion then BF(A) = –1, BF(B) = 0, after rotation

(a) LR rotation

The new element C is inserted in the right subtree of left subtree of A, the closest ancestor node whose BF(A) becomes +2 after insertion. To rebalance the search tree, it is rotated so as to allow C to be the root with B and A to be its left child and right child respectively. Observe how the rotation results in a balanced tree (See Fig. 7.44).

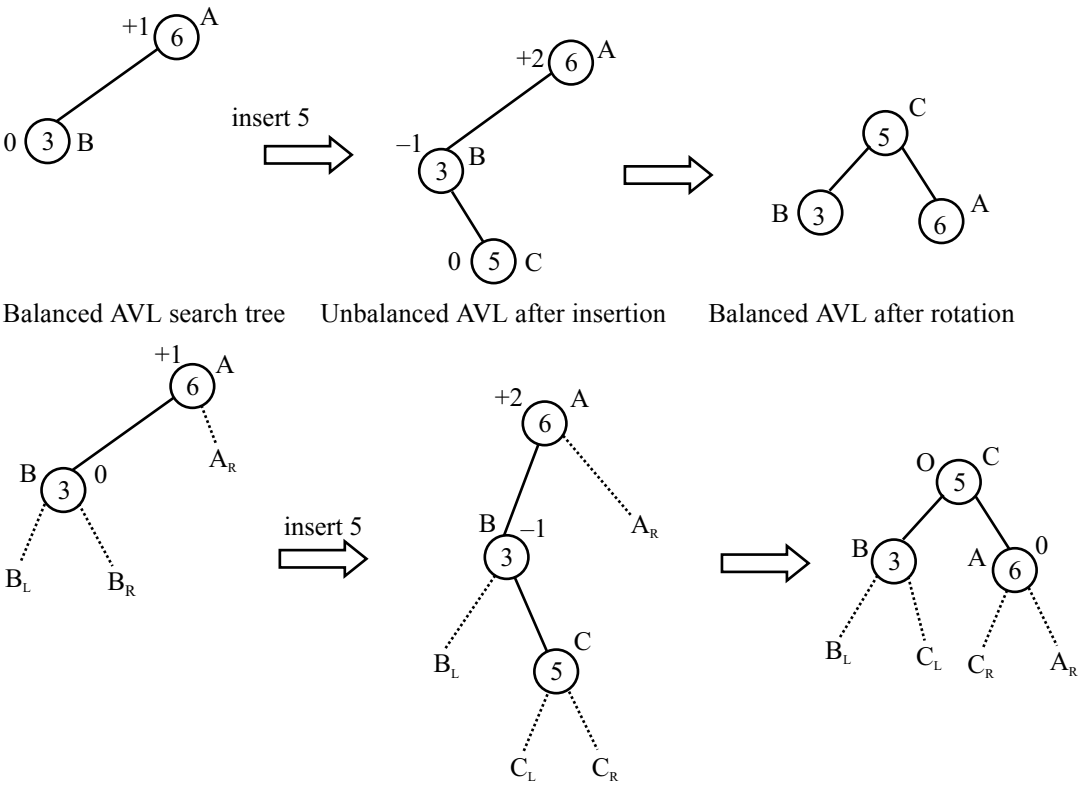


Figure 7.44 *LR rotation*

Initially, B_L : Left subtree of B is NULL, A_R : Right subtree of A is NULL , C_L Left subtree of C is NULL, C_R : Right subtree of C is NULL and h : Height is 2.

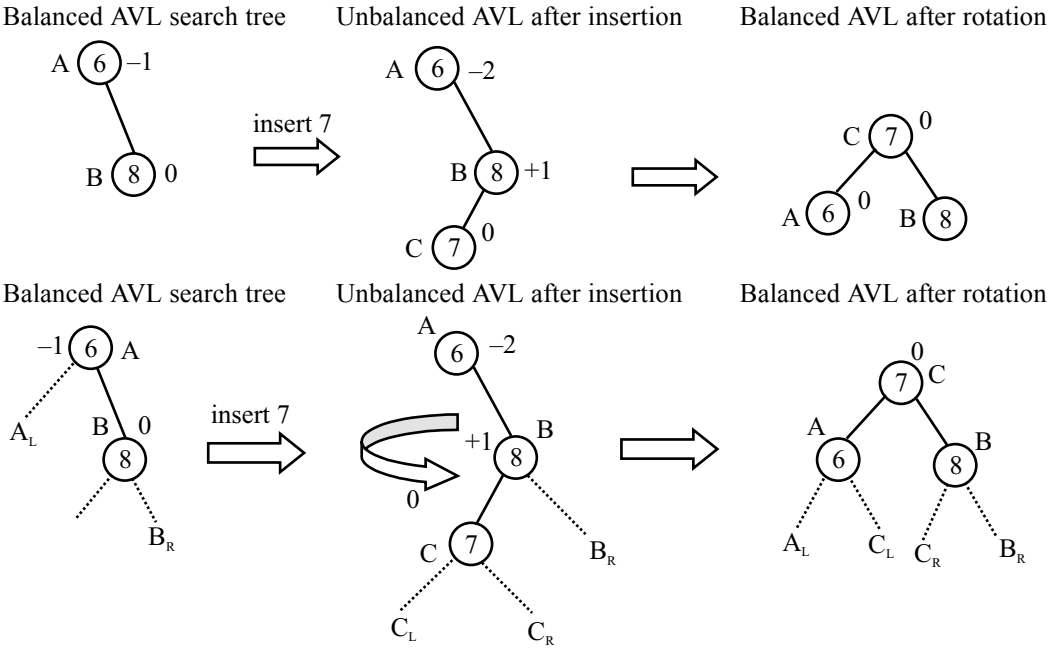
The pointer movement is done as below.

```
C = B–>rchild;
B–>rchild = C–>lchild;
A–>lchild = C–>rchild;
C–>lchild = B;
C–>rchild = A;
```

Note that balancing factor of node depends on height of subtrees.

(b) RL rotation

The new element C is inserted in the left subtree of right subtree of A, the closest ancestor node whose BF(A) becomes –2 after insertion. To rebalance the search tree, it is rotated so as to allow C to be the root with A and B to be its left child and right child. Observe how the rotation results in a balanced tree (see Fig. 7.45).



Initially, B_L : Left subtree of B is NULL, B_R : Right subtree of B is NULL, A_R : Right subtree of A is NULL. The pointer movement is done as below.

```
C = B–>lchild;
B–>lchild = C–>rchild;
A–>rchild = C–>lchild;
C–>rchild = B;
C–>lchild = A;
```

Figure 7.45 *RL rotation*

Note that balancing factor of node depends on height of subtrees.

Amongst the rotations, LL and RR rotations are called as *single rotations* and LR and RL are known as *double rotations* since LR can be accomplished by RR followed by LL rotation and RL can be accomplished by LL followed by RR rotation. The time complexity of an insertion operation in an AVL tree is given by *O(height) = O(log n)*.

Function for insertion and balancing the tree is given below:

```
/* function avlinsrt inserts the node into AVL tree and makes it balanced */
void avlinsrt(int x)
{
        struct nodetype *newnode, *p, *q, *b;
        int d, unbal;
        if (root = = NULL)
        {
        newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
        newnode–>code = x;
        newnode–>lchild = NULL;
```

```
newnode–>rchild = NULL;
newnode–>bf = 0;
root = newnode;
}
else
{
q = avlsrch(x);
        if (!found)
        {
/* insert and rebalance. x is not in tree and may be inserted as the
appropriate child of q */
        newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
        newnode–>code = x;
        newnode–>lchild = NULL;
        newnode–>rchild = NULL;
        newnode–>bf = 0;
        if ( x <q–>code)
                q–>lchild = newnode;
        else
                q–>rchild = newnode;
/* Adjust balance factors of nodes on path from ancestor to q. Note that by the definition of
ancestor, all nodes on this path must have balance factors of 0 and so will change to –+1. The
value of d = +1 implies x is inserted in left subtree of ancestor, and d = –1 implies x is inserted
in right subtree of ancestor. */
        if (x > ancestor–>code)
        {
        p = ancestor–>rchild;
        b = p;
        d = –1;
        }
        else
        {
        p = ancestor–>lchild;
        b = p;
        d = +1;
        }
        while (p! = newnode)
        {
        if (x >p–>code)
        {
                p–>bf = –1; p = p–>rchild;
        }
        else
        {
```

```
                    p–>bf = +1; p = p–>lchild;
                }
        }   /* end of while loop */
         /* [Is tree unbalanced] */
        unbal = 1;
        if (ancestor–>bf = = 0)
        {       ancestor–>bf = d;           unbal = 0;          }
        if (ancestor–>bf + d = =  0)
        {       ancestor–>bf = 0;           unbal = 0;          }
        if (unbal)    /* tree unbalanced, determine rotation type */
        {
                    if (d = = +1)           /* Left imbalance */
                        leftbal(b);
                    else
                    if (d = = –1)           /* Right imbalance */
                        rightbal(b);
                    rebal(b);
                } /* end of if unbal */
         } /* end of if not found */
} /* end of root = NULL */
} /* End of function avlinsrt */
```

```
/* function rebal is right balance the tree */
void rebal(struct nodetype *b)
{
        /* Subtree with root b has been rebalanced and is the new subtree */
                if ( paranc = = NULL)
                        root = b;
                else if ( ancestor = = paranc–>lchild)
                                paranc–>lchild = b;
                        else if (ancestor = = paranc–>rchild)
                                        paranc–>rchild = b;
 }
```

```
/* function avlsrch search the node */
struct nodetype *avlsrch(int x)
{
struct nodetype *p;
struct nodetype *q;
p = root;  q = NULL;
paranc = NULL;    ancestor = root;   found = 0;
        while ((p! = NULL)&& (!found))
        {
                if (p–>bf ! = 0)
                {       ancestor = p; paranc = q; }
```

```
                if (x <p–>code)
                {   q = p;   p = p–>lchild; }
                else if (x > p–>code)
                        {       q = p;          p = p–>rchild;          }
                        else
                                {       found = 1; }
        } /* End of While Loop */
return (q);
}
```

```
/* function leftbal balance the left branch of the tree */
void leftbal(struct nodetype *b)
{
struct nodetype *c;
        if (b–>bf  = = +1)    /* Rotation type left to left */
        {
                printf("\n Left to Left rotation");
                ancestor–>lchild = b–>rchild;
                b–>rchild = ancestor;
                ancestor–>bf = 0;           b–>bf = 0;
        }
        else
        {
                printf("\n Left to Right rotation");
                c = b–>rchild;
                b–>rchild = c–>lchild;
                ancestor–>lchild = c–>rchild;
                c–>lchild = b;
                c–>rchild = ancestor;
                switch (c–>bf)
                {
                        case +1: ancestor–>bf = –1; /* Left to Right rotation for b */
                                                b–>bf = 0;
                                                break;
                        case –1: ancestor–>bf = 0;  /* Left to Right rotation for c */
                                                b–>bf = 1;
                                                break;
                        case 0: ancestor–>bf = 0;/* Left to Right rotation for ancestor */
                                                b–>bf  = 0;
                                                break;
                } /* end of switch structure */
                c–>bf = 0;  b = c;          /* b is a new root */
        }/* end of else of b–>bf = +1 */
} /* End of function leftbal */
```

```
/* function rightbel balances the right branch of the tree */
void rightbal(struct nodetype *b)
{
struct nodetype *c;
        if (b–>bf = = –1)    /* Rotation type is right to right */
        {
                printf("\n Right to Right rotation");
                ancestor–>rchild = b–>lchild;
                b–>lchild = ancestor;
                ancestor–>bf = 0;        b–>bf = 0;
        }
        else
        {
                printf("\n Right to Left rotation");
                c = b–>lchild;
                b–>lchild = c–>rchild;
                ancestor–>rchild = c–>lchild;
                c–>rchild = b;
                c–>lchild = ancestor;
                switch (c–>bf)
                {
                        case +1: ancestor–>bf = –1; /* Right to Left rotation for b */
                                        b–>bf = 0;
                                        break;
                        case –1: ancestor–>bf = 0;  /* Right to Left rotation for c */
                                        b–>bf = 1;
                                        break;
                        case 0: ancestor–>bf = 0;  /* Right to left rotation for
                                                ancestor  */
                                        b–>bf = 0;
                                        break;
                        } /* end of switch structure */
                        c–>bf = 0;          b = c;    /* b is a new root */
      }/* end of else of b–>bf = –1 */
} /* End of function rightbal */
```

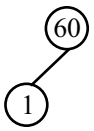
Example 1. Construct an AVL search tree by inserting the following elements in the order of their occurrence.

60, 1, 10, 30, 100, 90, 80

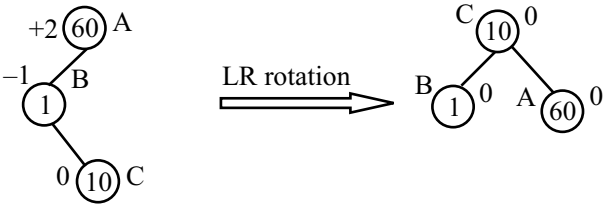
The elements are inserted one by one in the binary search tree maintaining its property. Fig. 7.46 gives the various stages in AVL search tree.

(*i*) Insert 60

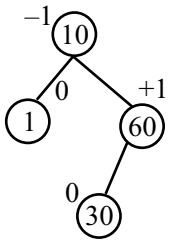
(*ii*) Insert 1



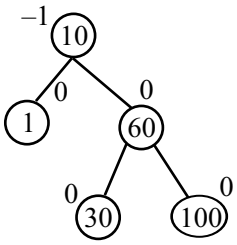
(*iii*) Insert 10



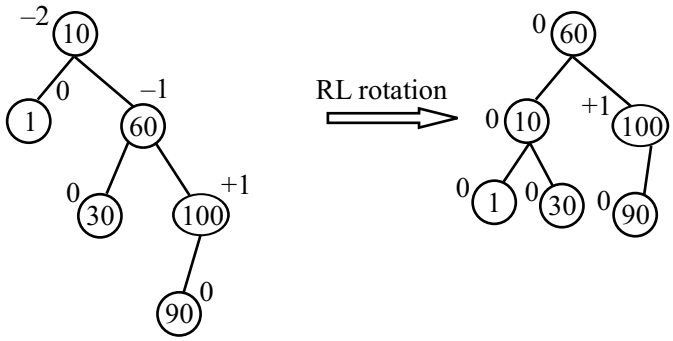
(*iv*) Insert 30



(*v*) Insert 100



(*vi*) Insert 90



(***Figure 7.46–contd...***)

(*vii*) Insert 80

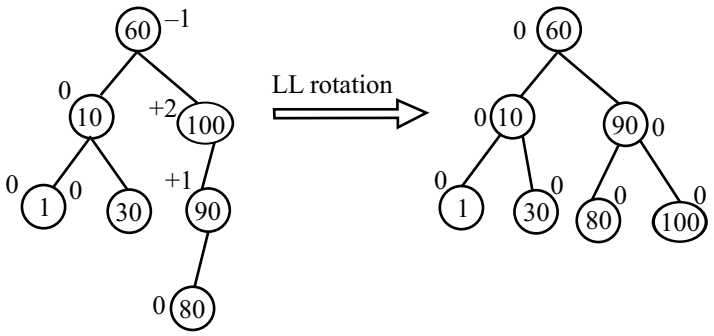


Figure 7.46 *Insertions in AVL trees*

Deletion in an AVL search tree

The deletion of an element in an AVL search tree proceeds for deletion of an element in a binary search tree. However, in the event of imbalance due to deletion, one or more rotations need to be applied to balance the AVL tree.

On deletion of a node X from the AVL tree, let A be the closest ancestor node on the path from X to the root node, with a balance factor of +2 or –2. To restore balance the rotation is first classified as L or R depending on whether the deletion occurred on the left or right subtree of A.

Now depending on the value of BF(B) where B is the root of the left or right subtree of A, the R or L imbalance is further classified as R0, Rl and R–l or L0, L1 and L–l. The three kinds of R rotations are illustrated with examples. The L rotations are but mirror images of these rotations.

R0 Rotation

If BF(B) = 0, the R0 rotation is executed as illustrated in Fig. 7.47(*a*).

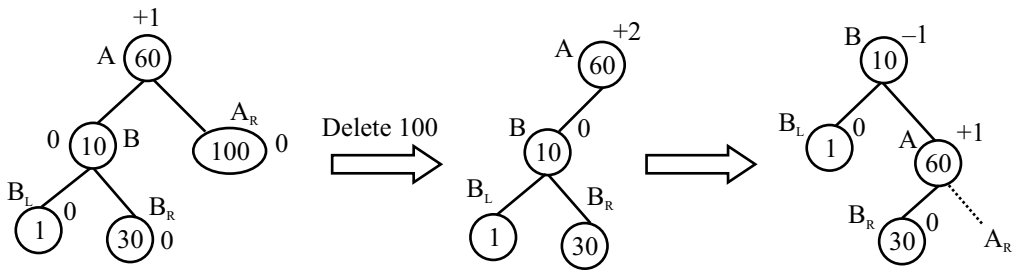


Figure 7.47(*a*)

If BF(B) = +1, the R1 rotation is executed as illustrated in Fig. 7.47(*b*).

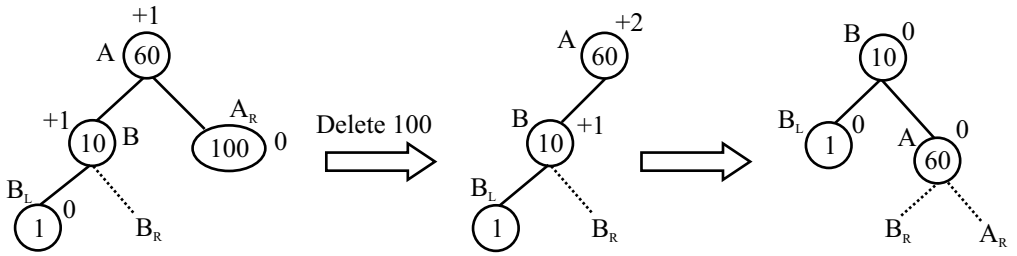


Figure 7.47(*b*)

If BF(B) = –1, the R–1 rotation is executed as illustrated in Fig. 7.47(*c*).

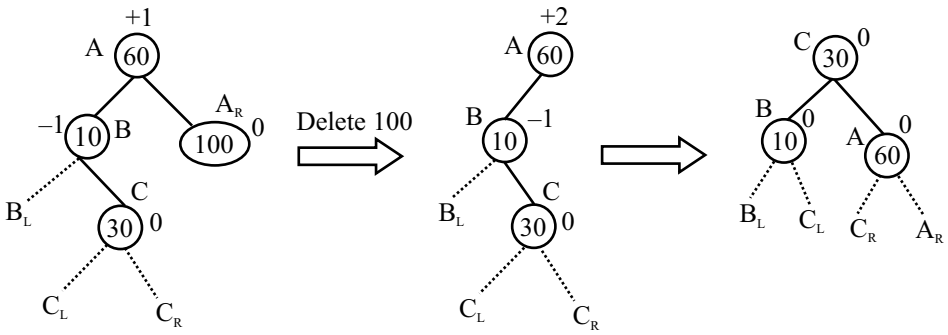


Figure 7.47(*c*) *Deletion in AVL tree*

Observe that LL and R0 rotations are identical. Though LL and R1 are also identical they yield different balance factors. LR and R–1 are identical.

L0 Rotation

If BF(B) = 0, the L0 rotation is executed as illustrated in Fig. 7.48 (*a*).

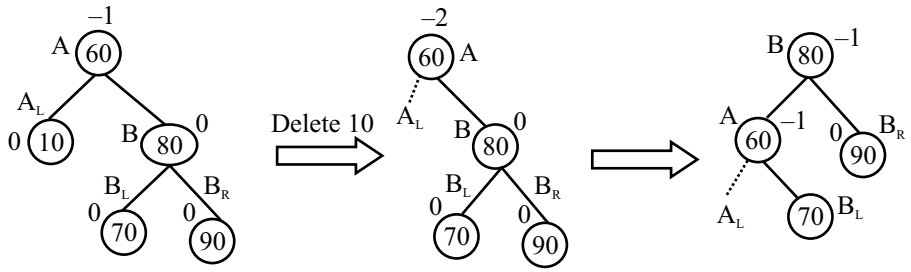


Figure 7.48 (*a*)

If BF(B) = +1, the L1 rotation is executed as illustrated in Fig. 7.48 (*b*).

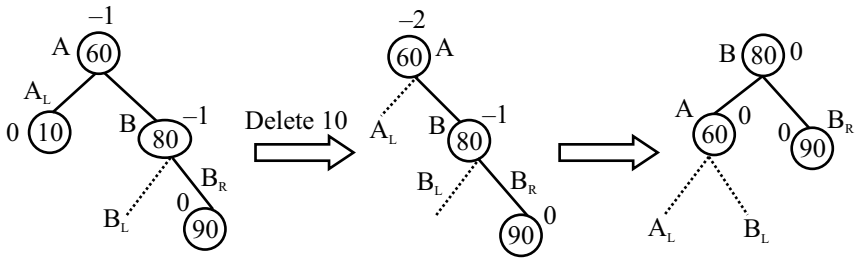


Figure 7.48 (*b*)

If BF(B) = –1, the L–1 rotation is executed as illustrated in Fig. 7.48 (*c*).

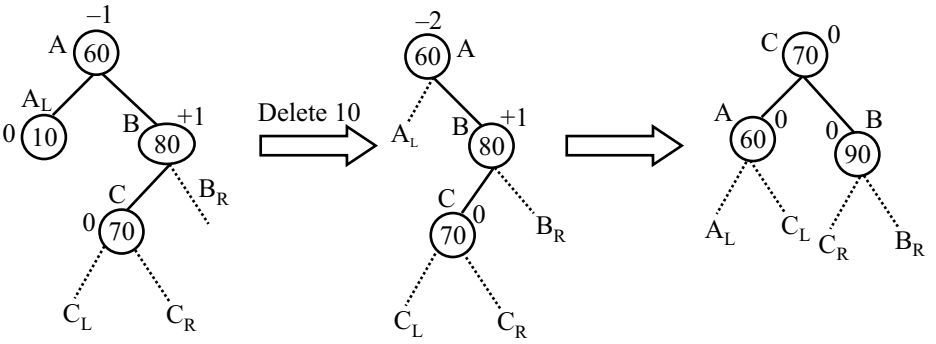


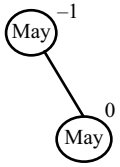
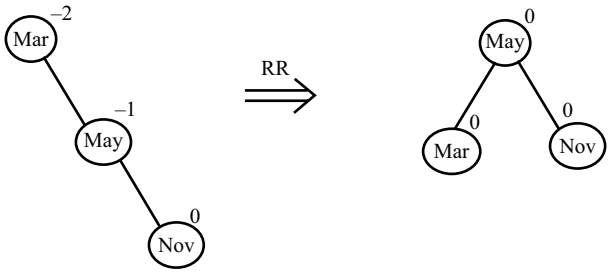
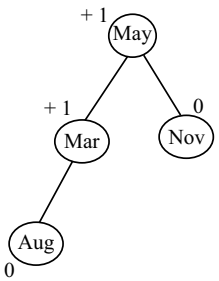
Figure 7.48 (*c*)

Observe that RR and L0 rotations are identical. Though RR and L1 are also identical they yield different balance factors. RL and L–1 are identical.

Example 2. Construct AVL tree for the following set of months.

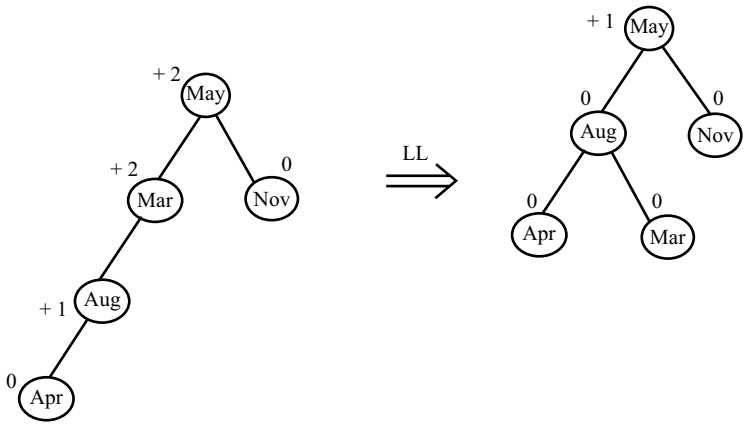
March, May, November, August, April, January, December, July, February, June, October, September.

The below Fig. 7.49 gives various stages of AVL tree insertions, first three letters is given for month name.

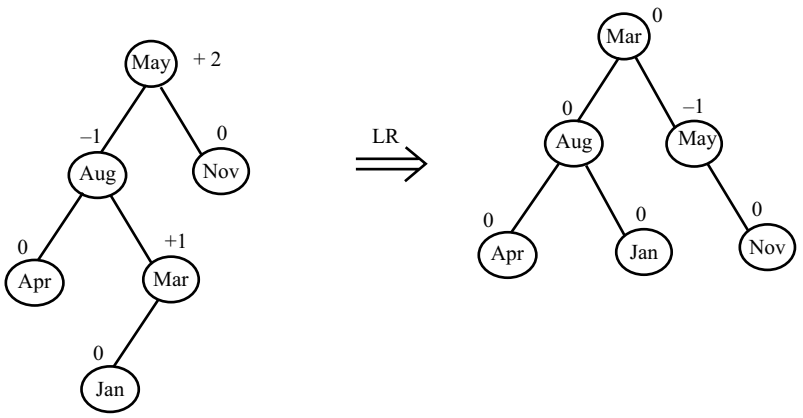
	Inserted Month	After Insertion	After Rebalancing
(*i*)	Mar		No rebalancing needed.
(*ii*)	May		No rebalancing needed.
(*iii*)	Nov		
(*iv*)	Aug		No rebalancing needed.

(***Figure 7.49–contd...***)

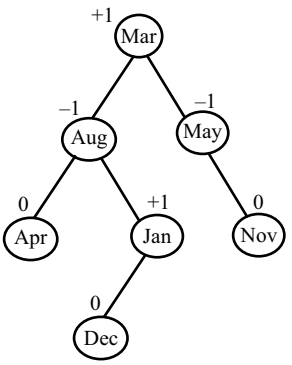
(*v*) Apr



(*vi*) Jan



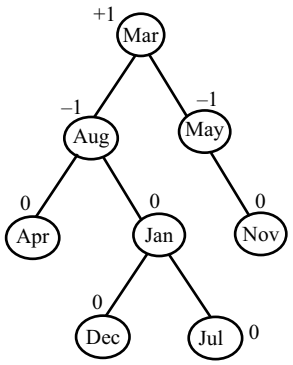
(*vii*) Dec



No rebalancing needed.

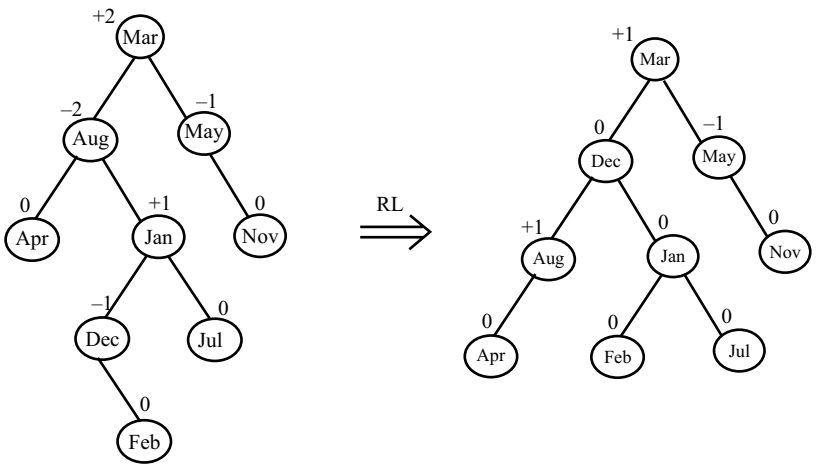
(***Figure 7.49–contd...***)

(*viii*) Jul

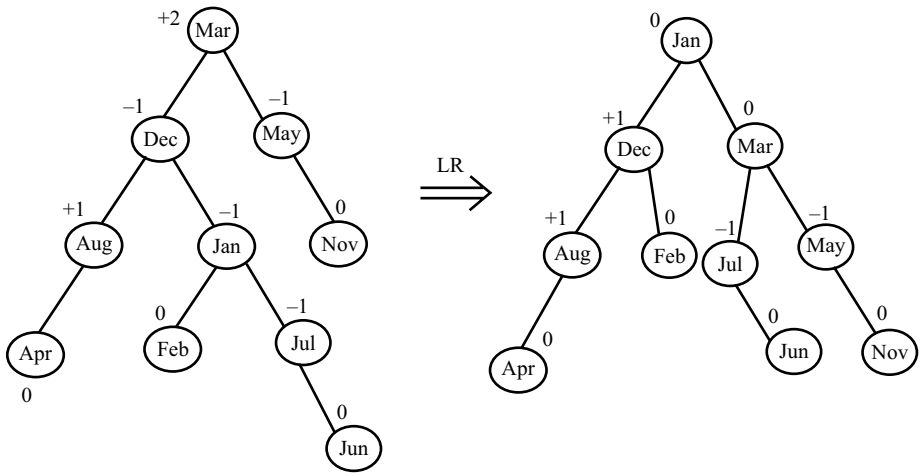


No rebalancing needed.

(*ix*) Feb

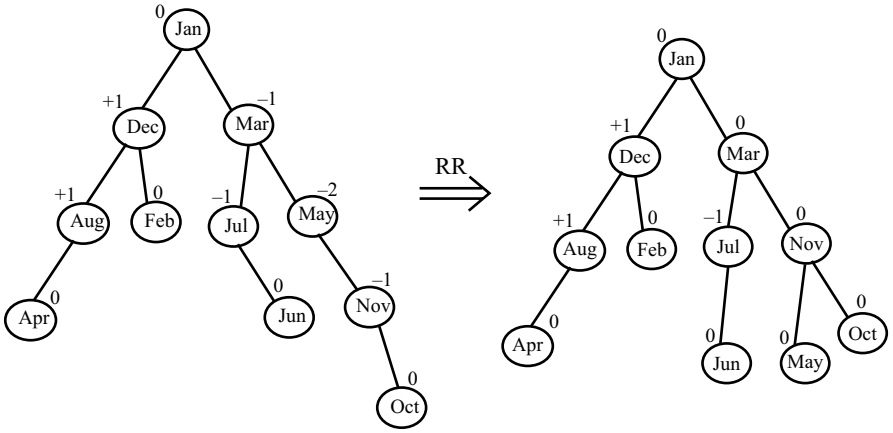


(*x*) Jun

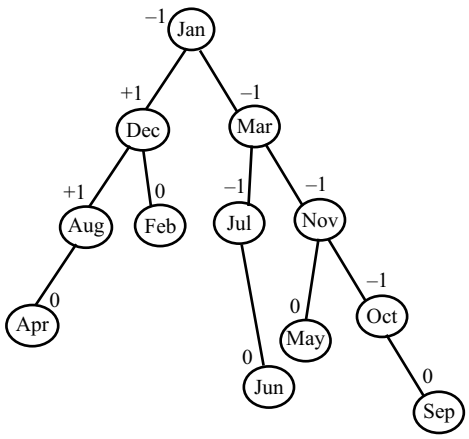


(***Figure 7.49–contd...***)

(*xi*) Oct



(*xii*) Sep



No rebalancing needed.

Figure 7.49

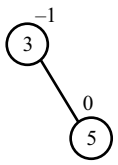
Example 3. Construct the AVL tree for the following set of element to be inserted one by one in Fig. 7.50.

3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10

(*i*) 3

No rebalancing needed

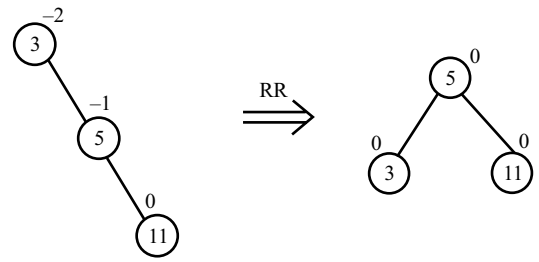
(*ii*) 5



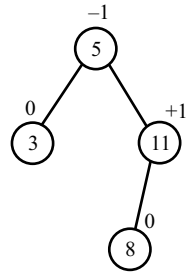
No rebalancing needed

(***Figure 7.50–contd...***)

(*iii*) 11

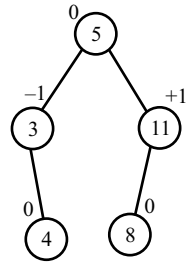


(*iv*) 8



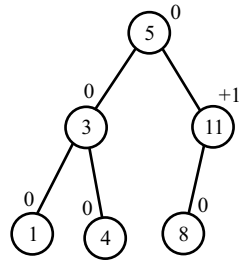
No rebalancing needed

(*v*) 4



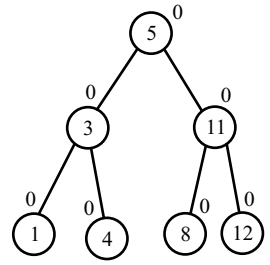
No rebalancing needed.

(*vi*) 1



No rebalancing needed.

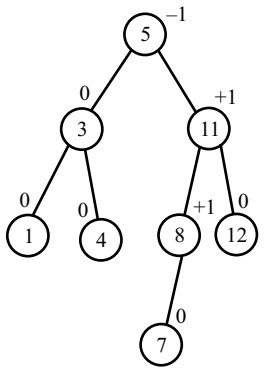
(*vii*) 12



No rebalancing needed.

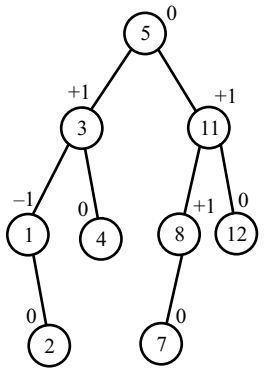
(***Figure 7.50–contd...***)

(*viii*) 7



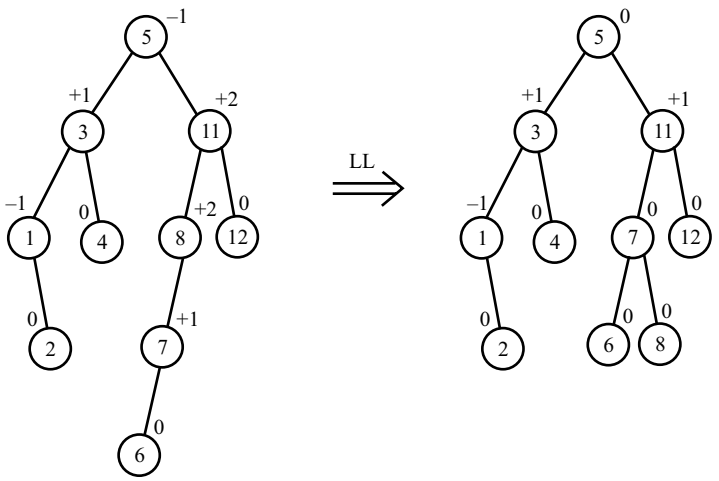
No rebalancing needed.

(*ix*) 2



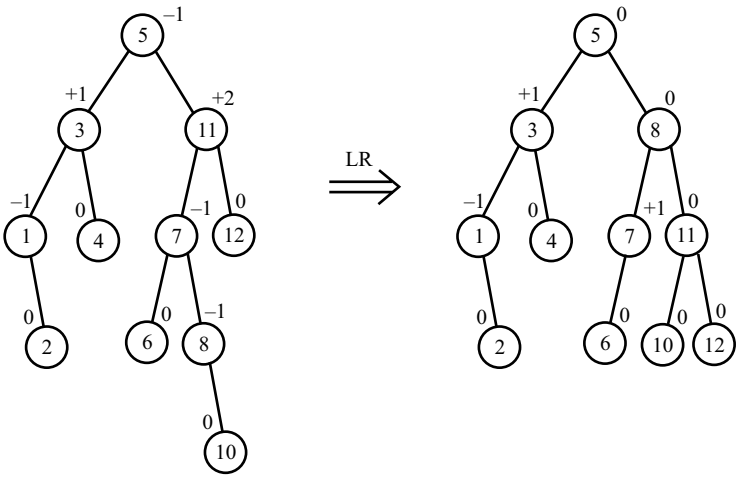
No rebalancing needed.

(*x*) 6



(***Figure 7.50–contd...***)

(*xi*) 10



Which is height balance tree.

Figure 7.50 *AVL tree*

7.9 B-TREES

A B–Tree is a balanced M–way tree. A node of the tree may contain many records or keys and pointers to children. A B–tree is also known as the balanced sort tree. It finds its use in external sorting. It is not a binary tree.

To reduce disk accesses, several conditions of the tree must be true.

- The height of the tree must be kept to a minimum;
- There must be no empty subtrees above the leaves of the tree;
- The all leaves of the tree must be on the same level; and
- All nodes except the leaves must have at least some minimum number of children.

B –Tree of order M has the following properties:

1. Each node has a maximum of M children and a minimum of *M/2* children and for root any number from 2 to the maximum.
2. Each node has one fewer keys than children with a maximum of M–1 keys.
3. Keys are arranged in a defined order within the node. All keys in the subtree to the left of a key are predecessors of the key and that on the right are successors of the key.
4. When a new key is to be inserted into a full node, the node is split into two nodes, and the key with the median value is inserted in the parent node. In case the parent node is the root, a new root is created.
5. All leaves are on the same level, i.e. there is no empty subtree above the level of the leaves.

While root and terminal nodes are special cases, normal nodes have between M/2 and M children. For example, a normal node of tree of order 11 has at least 6 and at most 11 children.

B-Tree Insertion

Method

1. First the search for the place where the new record must be placed is done. If the node can accommodate the new record insertion is simple. The record is added to the node with an appropriate pointer so that number of points remain one more than the number of records. If the node overflows because there is an upper bound in the size of node, splitting is required.

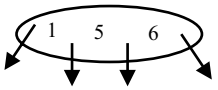
2. The node is split into three parts. The middle record is passed upward and inserted into the parent, leaving two children behind where there was one before. Splitting may propagate up the tree because the parent into which a record is to be split in its child node may overflow. Therefore it may also split. If the root is required to be split, a new root is created with just two children, and the tree grows taller one level.

Example 1. Consider building a B–tree of degree 4 that is a balanced four–way tree where each node can hold three data values, and have four branches (i.e. pointers). Suppose it needs to contain the following values:

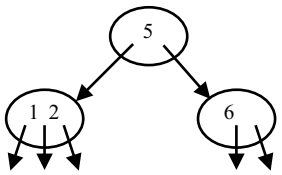
1 5 6 2 8 11 13 18 20 7 9

The following stages of B–tree of order–4 is created in Fig. 7.51.

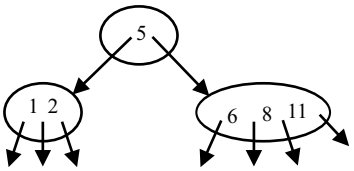
1. The first value 1 is placed in a new node, which can accommodate the next two, values also i.e.



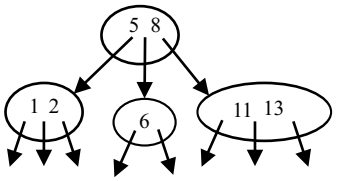
2. When the fourth value 2 is to be added, the node is split at a median value 5 into two leaf nodes with a parent at 5.



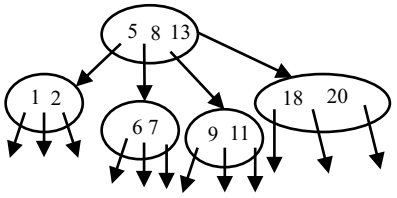
3. The following item 8 is to be added in a leaf node. A search for its appropriate place is the node containing 6. Next, 11 is also placed in the same. So we have



4. Now 13 is to be added. But the right leaf node, where 13 finds appropriate place, is full. Therefore, it is split at median value 8 and thus it moves up to the parent. Also it splits up to make two nodes.



5. The remaining items may also be added following above procedure. The final result is



Note that the tree built up in this manner is balanced, having all of its leaf nodes at one level. Also the tree appears to grow at its root, rather than at its leaves as was the case in binary tree.

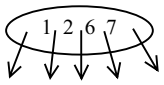
Figure 7.51 *Creation of B-tree*

Example 2. Insert the following elements in order–5 of B-tree.

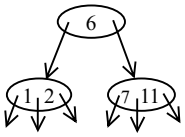
1, 7, 6, 2, 11, 4, 8, 13, 10, 5, 19,
9, 18, 24, 3, 12, 14, 20, 21, and 16.

The following Fig. 7.52 shows various stages of B-tree of order-5 constructions.

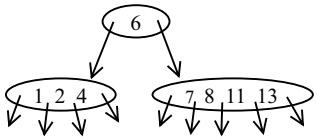
(*i*) Insert 1, 7, 6, 2



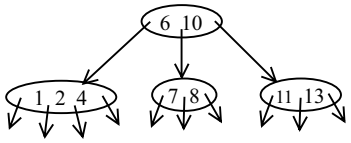
(*ii*) Insert 11 split the root into two sets.



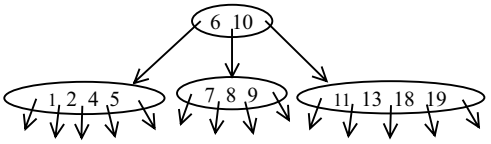
(*iii*) Insert 4, 8, 13



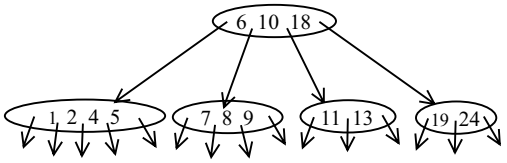
(*iv*) Insert 10 move the median towards root



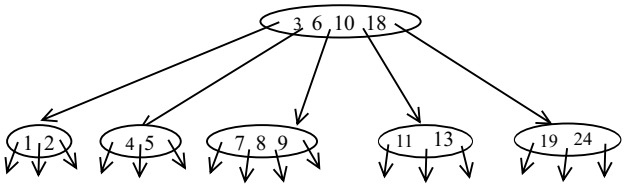
(*v*) Insert 5, 19, 9, 18 does not change leaves level.



(*vi*) Insert 24 required shifting of median value towards root.

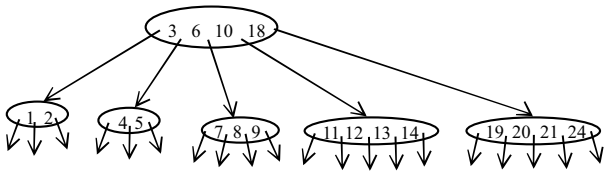


(*vii*) Insert 3 need splitting



(***Figure 7.52–contd...***)

(*viii*) Insert 12, 14, 20, 21



(*ix*) Insert 16 need splitting and shifting.

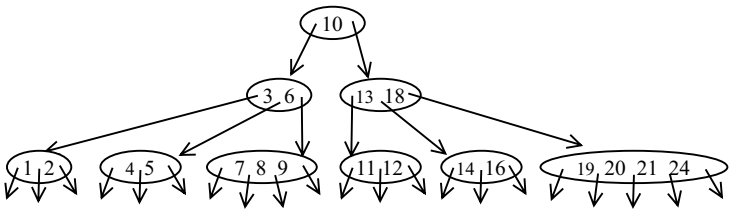


Figure 7.52 *Insertion in B-Tree*

B-Tree Deletion

As in the insertion method, the record to be deleted is first searched for.

If the record is in a terminal node, deletion is simple. The record along with an appropriate pointer is deleted.

Method

1. If the record is not in terminal node, it is replaced by a copy of its successor, that is, a record with a next, higher value. The successor of any record not at the lowest level will always be in a terminal node. Thus in all cases deletion involves removing a record from a terminal node.

2. If on deleting the record the new node size is not below the minimum, the deletion is over. If the new node size is lower than the minimum an underflow occurs.

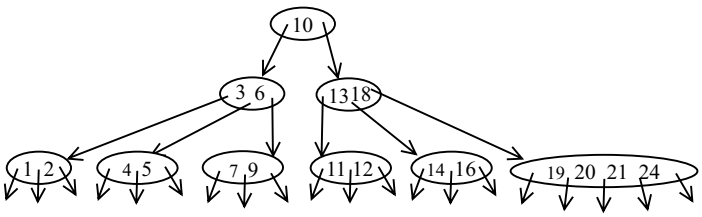
3. Redistribution is carried out if either of adjacent siblings contains more than the minimum number of records. For redistribution, the contents of the node which has less than minimum records, the contents of its adjacent sibling which has more than minimum records, and the separating record from parent are collected. The central record from this collection is written back to parent. The left and right halves are written back to the two siblings.

4. In case the node with less than minimum number of records has no adjacent sibling that is more than minimally full, concatenation is used. In this case the node is merged with its adjacent sibling and the separating record from its parent.

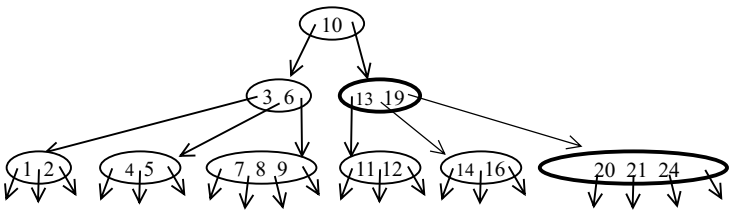
5. It may be solved by redistribution or concatenation.

Example 3. The process of deletion in the previous B-Tree of order-5 in Fig. 7.52 is illustrated in Fig. 7.53.

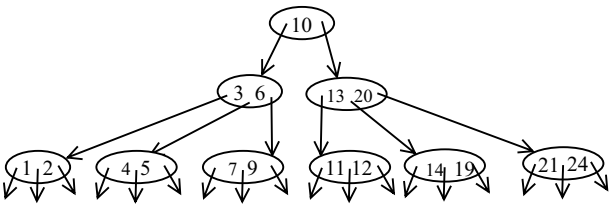
(*i*) The first deletion of 8 is from a leaf with more than the minimum number of elements and hence leaves no problem.



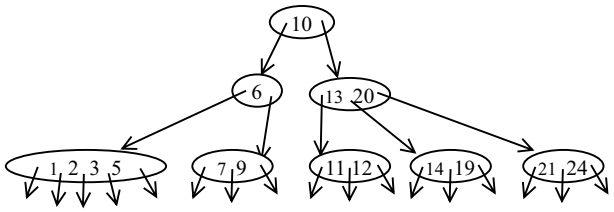
(*ii*) The deletion of 18 is not from leaf and therefore the immediate successor is placed at 18 (i.e. 19).



(*iii*) The deletion of 16 leaves its nodes with too few elements. The element 19 from parent node is therefore brought down and replaced by the element 20.

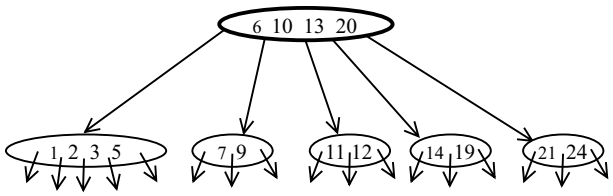


(*iv*) The deletion of 4 leaves the node with too few elements and neither of its sibling nodes can spare an element. The node therefore is combined with one of the siblings and with the median element from the parent node as illustrated in the following figure.



Again combined the upper two level.

(***Figure 7.53–contd...***)



It reduces the height of tree also.

Figure 7.53 *B-tree deletion.*

7.10 APPLICATIONS OF TREES

(*i*) The manipulation of arithmetic expression
(*ii*) Construction and maintenance of symbol table
(*iii*) Syntax analysis

(*i*) **The manipulation of arithmetic expression** – There is a close relationship between binary trees and formulas in prefix or suffix notation.

We can write an infix formula as binary tree, where a node has an operator as a value and where the left and right subtrees are left and right operand of that operator. The leaves of the tree are the variables and constants in the expression.

Consider an expression a * (–b) + c ↑ 2. The binary tree is as follows in figure 7.54.

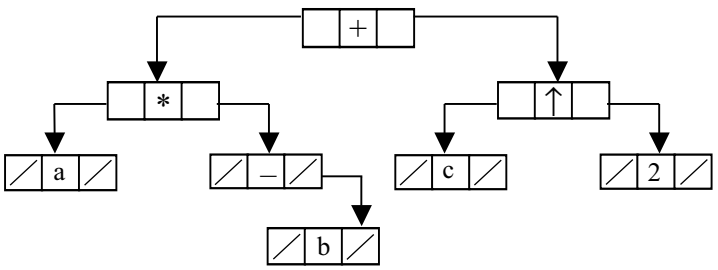


Figure 7.54 *Expression a * (–b) + c ↑ 2 as binary tree*

Each node of tree consists of left child pointer (lchild) and a right child pointer (rchild) and an info field. The values of info field associated with the operators +, –,*, /, ↑ and unary are 1, 2, 3, 4, 5 and 6 respectively. For a leaf node, however, info denotes a variable or a constant.

The right child of leaf is used to give the address in the symbol table which corresponds to that variable or constant.

This choice makes the processing of such tree simple.

The symbol table contains the name of the variable (symbol) or constant and its value as in Fig. 7.54.

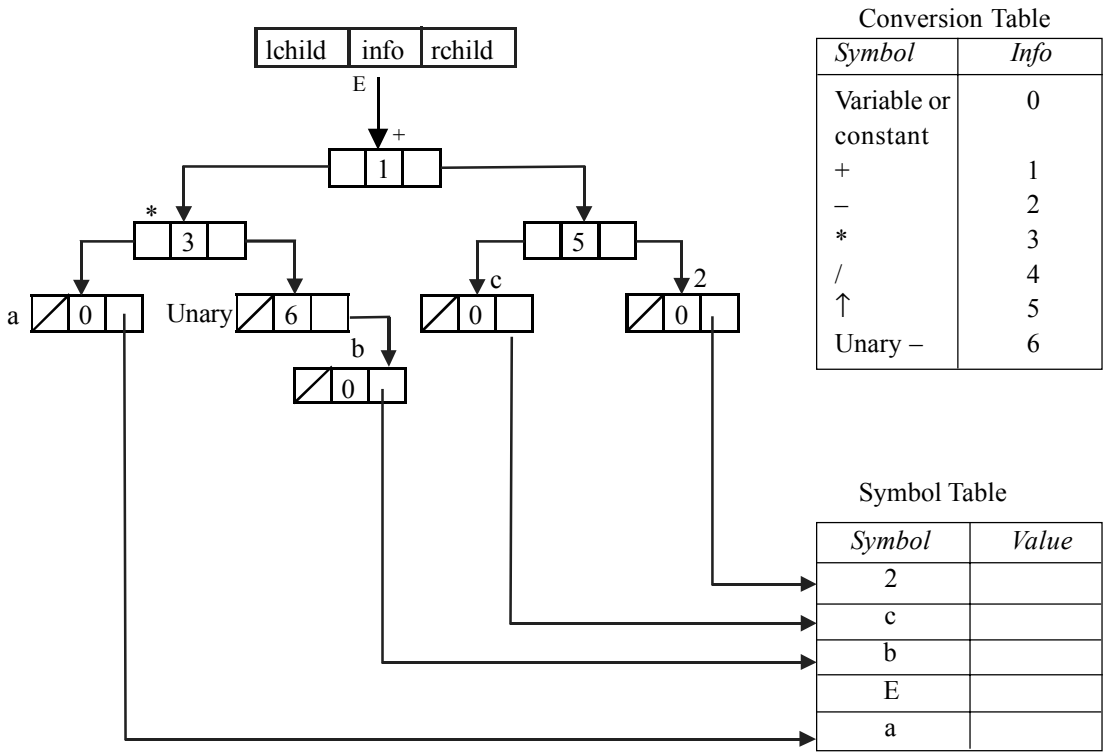


Figure 7.55 *A binary tree representation of an expression.*

Let us consider the evaluation of an expression, which is represented by a binary tree. That is, we want to obtain the value of this expression. The recursive function is given below:

```
EVAL (E)
/* Pointer E denotes the address of the root node, this function returns the value of this expression.
PTR is a local pointer variable */
{
        switch (E→info)
        {
        case 0 : PTR  =  (E→rchild)   /* a variable or constant */
                        return (value (PTR));
        case 1 : return ((EVAL(E→lchild) + EVAL(E→rchild)); /* additional operator */
        case 2 : return ((EVAL(E→lchild) – EVAL(E→rchild)); /* subtraction operator */
        case 3 : return (EVAL(E→lchild)) * EVAL(E→rchild));/*multiplication operator */
        case 4 : return ((EVAL(E→lchild) / EVAL(E→rchild)); /* division operator */
        case 5 : return (EVAL(E→lchild)) ↑EVAL(E→rchild)); /* exponent operator */
        case 6 : return (–EVAL(E→rchild)); /* a unary operator */
        default : printf("Invalid expression");
                        return (0);
       }
  }
```

(*ii*) **Symbol table construction** — As an application of binary trees, we will maintain a stack–implemented tree structured symbol table. The two required operations that must be performed on a symbol table are insertion and look–up, each of which involves searching. The binary tree structure does both very easily.

A binary tree is constructed with typical nodes of the form

lchild	Symbol	Info	rchild

where lchild and rchild are pointer fields, symbol is the field for the character string which is the identifier character name, and info is additional information about the identifier, such as its type.

Here the similar function is used as BST insertion where no duplicate entries are inserted and BST search function is applied for look-up operation.

(*iii*) **Syntax analysis** — Syntax analysis is used to display the structure of a sentence in a language and is used in defining unambiguous languages.

The concept of a syntax tree and its relationship to the parse of a sentence of a language are not discussed here.

Here for the sake of simplicity, the syntax tree for some expression is illustrated.

Consider the simple grammar G, (i.e., context-free grammar), which has the following productions

(1) $S \rightarrow S * S$

(2) $S \rightarrow S + S$

(3) $S \rightarrow a$

where a is a terminal symbol and S is start symbol.

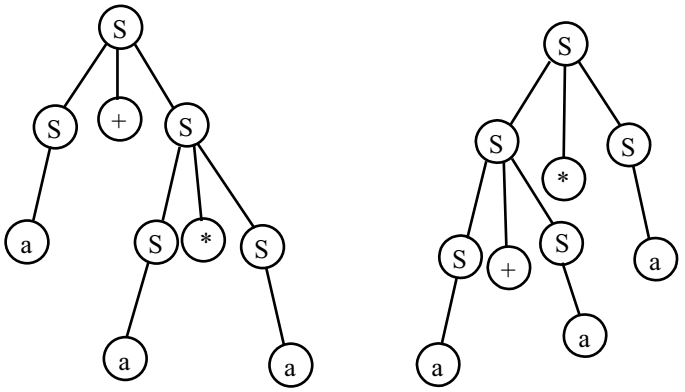
Let us find the derivation for the sentence a + a * a. One such derivation is

$$
\begin{aligned}
S &\Rightarrow S + S \\
&\Rightarrow S + S * S && \text{as } S \rightarrow S * S \\
&\Rightarrow a + S * S && S \rightarrow a \\
&\Rightarrow a + a * S && S \rightarrow a \\
&\Rightarrow a + a * a && S \rightarrow a
\end{aligned}
$$

The syntax tree for this derivation is given in Fig. 7.55.

The another derivation is

$$
\begin{aligned}
S &\Rightarrow S * S \\
&\Rightarrow S + S * S && \text{as } S \rightarrow S+S \\
&\Rightarrow S + S * a && S \rightarrow a \\
&\Rightarrow S + a * a && S \rightarrow a \\
&\Rightarrow a + a * a && S \rightarrow a
\end{aligned}
$$



(*i*) (*ii*)

Figure 7.56 *Syntax tree*

The sentence generated by this grammar is ambiguous as there exists more than one syntax trees for it.

The top-down parsing and bottom up parsing can be implemented with stack using recursive nature of the tree.

Chapter 8

Non-Linear Data Structure: Graphs

8.1 PROPERTIES OF GRAPHS

A graph G consists of two sets V and E. The set V is a finite, non-empty set of vertices. The set E is a set of pairs of vertices, these pairs are called *edges*.

Graph G can be represented as G =(V, E)

- A graph is *nodes* joined by *edge i.e., A set of nodes V and a set of edges E*
- A node is defined by its name or label.
- An edge is defined by the two nodes which it connects, plus optionally:
  - An order of the nodes (*direction*)
  - A *weight or cost* (usually a number)
- Two nodes are *adjacent* if they are connected by an edge
- A node's *degree* is the number of its edges

Graph can be directed or undirected

- In undirected graph pair of vertices representing vertices are unordered. Thus, the pair (u, v) and (v, u) representing the same edge.
  For example (undirected graph), see Figure 8.1.
  V(G) = {1,2,3,4}
  E(G) = {(1,2),(2,4),(4,3),(3,1)}

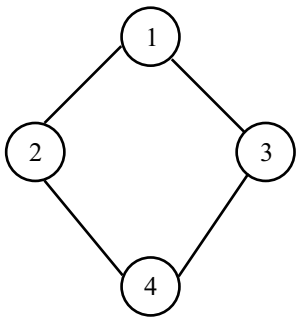


Figure 8.1 *Undirected graph (unweighted)*

- In directed graph each edge is represented by direct pair **(u, v)** where **u** is a tail and **v** is a head of edge. Thus **(u, v)** and **(v, u)** are two distinct edges.
  For example:– (directed graph) See Figure 8.2.
  V(G)={1,2,3,4} E(G)={(1,2),(2,1)(2,4)(4,2),(4,3)(3,4),(3,1)(1,3)}

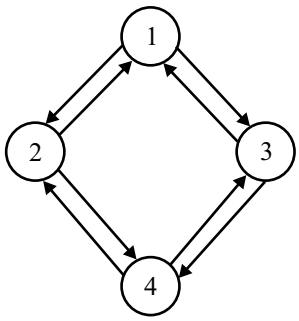
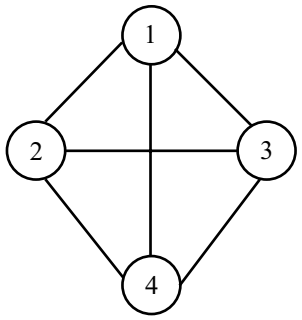


Figure 8.2 *Directed graph (unweighted)*

- In an undirected graph with **n** vertices the maximum number of edges = **n(n–1)/2** then graph is said to be a complete graph.
  For example: See Figure 8.3

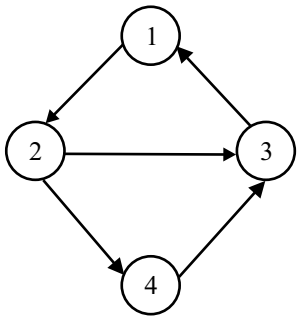


Number of nodes = 4
Number of edges = 4 (4–1)/2=6

V(G)={1,2,3,4} E={(1,2),(2,4),(4,3),(3,1),(1,4),(2,3)}

Figure 8.3 *Complete undirected graph*

- **PATH** :– Path can be defined as sequence of vertices **(u1, u2, u3.....v)** in such a way that there are **(u1, u2),(u2, u3),.....(uk, v)** edges in G(E).
- Any graph is said to be strongly connected if a path exists between each of the vertices of graph. In directed graph each edge is represented by a direct pair **(u, v)** where u is a tail, v is a head.
- *Indegree*: In directed graph each of the vertices has indegree defined as number of edges for which v is head.
- *Outdegree*: In directed graph each vertex has outdegree defined as number of edges for which v is tail.
- *length*: the number of edges in the path
- *cost*: the sum of the weights on each edge in the path
- *cycle*: a path that starts and finishes at the same node
- An *acyclic* graph contains no cycles
- Digraphs are usually either *densely* or *sparsely* connected
  - *Densely*: the ratio of number of edges to number of nodes is large
  - *Sparsely*: the above ratio is small



For vertex 2
Indegree = 1
Outdegree = 2

$e = \left(\sum_{I=1}^{n} di \right) / 2$ where e→edges

n→number of vertices

di→degree of vertices

Figure 8.4 *Directed graph and its degree*

8.2 REPRESENTATION OF GRAPHS

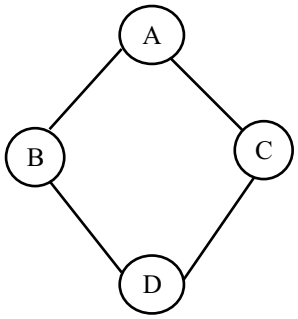
Graphs can be represented in following ways:

1. Adjacency matrix
2. Adjacency list

1. **Adjacency matrix** : let G=(V,E) be a graph with n vertices, n>=1. The adjacency matrix of G is a two-dimensional n*n array (i.e., matrix), say with the property that a[i,j]=1 if and only if there is an edge(i,j) in E(G). The element a(i,j)=0 if there is no such edge in G.

   For weighted or cost graphs, the position in the matrix is the weight or cost.

   For example, Figure 8.5 shows undirected graph and its adjacency matrix.

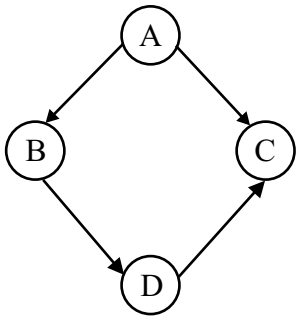


Adjacency matrix:

	Column			
Row	A	B	C	D
A	0	1	1	0
B	1	0	0	1
C	1	0	0	1
D	0	1	1	0

Figure 8.5 *Undirected graph and its adjacency matrix*

Example: Figure 8.6 shows directed graph and its adjacency matrix.

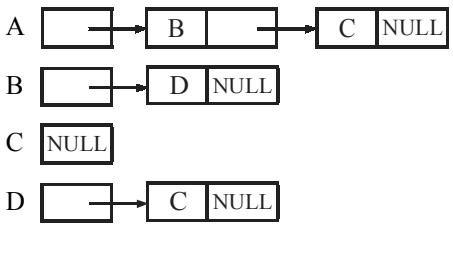


Adjacency matrix:

	Column			
Row	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	0	0	1	0

Figure 8.6 *Directed graph and its adjacency matrix*

2. **Adjacency list** : In this representation of graphs, the n rows of adjacency matrix are represented as n linked lists. There is one list for each vertex in G. See Fig. 8.7.
   For weighted graphs, include the weight/cost in the elements of the list



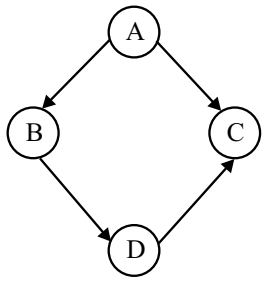


Figure 8.7 *Directed graph and its adjacency list*

Comparing the two representations

(*i*) Space complexity
- Adjacency matrix is $O(|n|^2)$
- Adjacency list is $O(n + |E|)$
  $|E|$ is the number of edges in the graph

(*ii*) Static versus dynamic representation
- An adjacency matrix is a *static* representation: the graph is built 'in one go', and is difficult to alter once built.
- An adjacency list is a *dynamic* representation: the graph is built incrementally, thus is more easily altered during run-time.

8.3 TRAVERSAL ALGORITHMS — DEPTH FIRST SEARCH, BREADTH FIRST SEARCH

Many graph applications require to visit all the vertices in a graph. A fundamental problem concerning graph is the reachability problem. In the simplest form it requires to determine whether there exists a path among the vertices. In case of a graph, we can specify any arbitrary vertex as the starting vertex. In the given graph **G = (V, E),** we want to visit all vertices in G that are reachable from vertex v, where v ⊂V.

Problems with the Graph Traversal:

(*i*) No First Node. So there must be a way to find out the starting node of the graph. User can enter the starting point or there can be several other methods to find out the starting point.

(*ii*) When traversing a graph, we must be careful to avoid going round in circles. We do this by marking the vertices which have already been visited. List of visited nodes can also be maintained.

(*iii*) No natural order among the successor of a particular node.

(*iv*) Nodes which are not connected or to which there is no path.

We have two common search methods:

1. Breadth first search
2. Depth first search

1. Breadth First Search (BFS) Method: (1) In breadth first search we start at vertices v and mark it is having been reached. All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. (2) Then vertex **v** has now been explored. The new vertices that have not been visited are put onto the end of the list of unexplored vertices. (3) The vertex first in the list is the next to be explored. This exploration continues until no unexplored vertex is left. The list of unexplored vertices as a queue can be represented by queue. It is based on **FIFO** system.

Breadth first search for the graph in Figure 8.8 is as follows in queue.

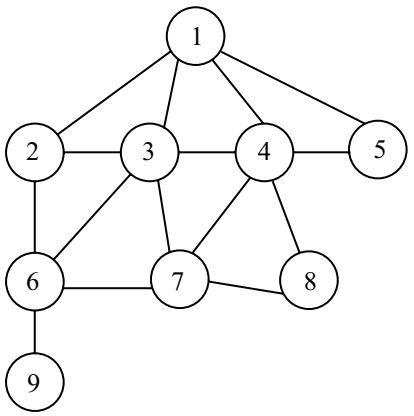


Figure 8.8 *Undirected graph, for finding the breadth first search and depth first search sequence*

The steps of over search follow as start node 1

1. Initially, add 1 to queue as follows:
   Front = 1 Queue:1
   Rear = 1
2. Delete the front element 1 from queue and add adjacent nodes of 1 to queue which is not visited.

Front = 2 Queue:2,3,4,5
Rear =5

3. Delete the front element 2 from queue and add adjacent nodes of 2 to queue which is not visited.
   Front = 3 Queue:3,4,5,6
   Rear = 6
4. Delete the front element 3 from queue and add adjacent nodes of 3 to queue which is not visited.
   Front = 4 Queue:4,5,6,7
   Rear = 7
5. Delete the front element 4 from queue and add adjacent nodes of 4 to queue which is not visited.
   Front = 5 Queue:5,6,7,8
   Rear = 8
6. Delete the front element 5 from queue and add adjacent nodes of 5 to queue which is not visited.
   Front = 6 Queue:6,7,8
   Rear = 8
7. Delete the front element 6 from queue and add adjacent nodes of 6 to queue which is not visited.
   Front = 7 Queue:7,8,9
   Rear = 9
8. Delete the front element 7 from queue and add adjacent nodes of 7 to queue which is not visited.
   Front = 8 Queue:8,9
   Rear = 9
9. Delete the front element 8 from queue and add adjacent nodes of 8 to queue which is not visited.
   Front = 9 Queue:9
   Rear = 9
10. Delete the front element 9 from queue and add adjacent nodes of 9 to queue which is not visited.
    Front = 0 Queue:0
    Rear = 0

Breadth first search is 1, 2, 3, 4, 5, 6, 7, 8, 9

Algorithm of Breadth First Search:

1. Non Recursive
2. Recursive

Non Recursive: A breadth first search of G carried out beginning at vertex g for any node i, visited[i] = 1 if i has already been visited. Initially, no vertex is visited so for all vertices visited [i] = 0.

```
        Algorithm BFS(g)
Step 1 : h = g;
Step 2 : visited[g] = 1;
Step 3 :  repeat
           {
Step 4 : for all vertices  w  adjacent  from h do
           {
             if (visited[w] = 0)  then
             {
```

```
                add w to q;  // w is unexplored
                visited[w] = 1;
            }                    // end of the if
          }       // end of the for loop
```

Step 5 : if q is empty then return;
Delete h from q;
} until(false);
// end of the repeat

Step 6 : end B F S.

Function

```
void bfs(int a[MAXSIZE][MAXSIZE], int g, int n)
{
    int i,j, visited[MAXSIZE],h;
    for(i = 0;i<n;i++)
    visited[i] = 0;
    h = g ;
    visited[h] = 1;
    printf("\n BFS sequence with start vertex is –>");
    while(1)
    {
    printf(" %d ", h);
     for (j = 0; j<n; j++)
      {
            if ((a[h][j] == 1) && (visited[j] = = 0))
                  {
                        queueins(&q, j);
                        visited[j] = 1;
                  }
     }
            if (empty(&q))
                  return;
            else
                  h = queuedel(&q);

    } /* end of while */
} /* end BFS */
```

Recursive:

Algorithm BFS(g)
/ *A breadth first search of G carried out beginning at vertex g for any node i, visited[i] = 1 if i has already been visited. Initially, no vertex is visited so for all vertices visited[i] = 0. */

Step 1 : visited[g] = 1;

Step 2 : for each vertex w adjacent from g do
{
if(visited[w] = 0) then
BFS(w);
}

Step 3 : end BFS.

The time complexity of BFS algorithm is O(n + |E|) if G is represented by adjacency lists, where n is number of vertices in G and |E| number of edges in G. If G is represented by adjacency matrix, then $T(n,e) = O(n^2)$ and space complexity remains S(n,e) = O(n) same.

A complete 'C' program for breadth first search traversal is given below:


```
/* Program bfs.cpp traverses a graph which is input in the form of adjacency matrix in breadth first search
non–recursive method. It uses the basic concept of queue data structure */
# include<stdio.h>
#define MAXSIZE 10
struct queue{
    int items[MAXSIZE];
    int rear;
    int front;
}q;
void queueins(struct queue *,int);
int queuedel(struct queue *);
int empty(struct queue *);
void adjmat(int[MAXSIZE][MAXSIZE], int);
void bfs(int[MAXSIZE][MAXSIZE],int,int);
void main()
{
    int a[MAXSIZE][MAXSIZE], v,g;
    q.front =–1; q.rear = –1;    /* Initialize queue to empty */
    printf("enter number of vertices in graph\n");
    scanf("%d",&v);
    adjmat(a,v);
    printf("Enter start vertex [0–%d]\n",v–1);
    scanf("%d",&g);
    bfs(a,g,v );
    getch();
} /* end of main */

/* Input function */
void adjmat(int a[MAXSIZE][MAXSIZE], int n)
{
    int i, j;
    printf("\n Input adjacency matrix of order %dx%d\n",n,n);
```

```
    for(i = 0; i < n; i++)
            for(j = 0; j < n; j++)
                    scanf("%d", &a[i][j]);
}

void bfs(int a[MAXSIZE][MAXSIZE], int g, int n)
{
    int i,j, visited[MAXSIZE],h;
    for(i = 0;i<n;i++)
    visited[i] = 0;
    h = g ;
    visited[h] = 1;
    printf("\n BFS sequence with start vertex is –>");
    while(1)
    {
    printf(" %d", h);
     for (j = 0; j<n; j++)
      {
            if ((a[h][j] == 1) && (visited[j] == 0))
                    {
                            queueins(&q, j);
                            visited[j] = 1;
                    }
     }
            if (empty(&q))
                    return;
            else
                    h = queuedel(&q);

    } /* end of while */
} /* end BFS */

void queueins(struct queue *q ,int x)
{
/*check for queue full, if so give a warning message and return */
if (q–>rear = = MAXSIZE –1)
 {
  printf("Queue full\n");
   return;
 }
else
{
```

```
/* check for queue empty, if so initialize front and rear pointers to zero.
 Otherwise increment rear pointer*/
 if (q–>front = = –1)
 { q–>front = 0; q–>rear = 0;}
 else
   q–>rear = q–>rear+1 ;
 q–>items[q–>rear] = x;
}
} /* End of queueins function */

int queuedel(struct queue *q)
 {
      int x;
      x = q–>items[q–>front];
/* if both pointers at the same position then reset the queue to empty position
 otherwise increment front pointer by 1.*/
       if (q–>front = = q–>rear)
       { q–>front = –1; q–>rear = –1;}
       else
         q–>front = q–>front +1 ;
   return x;
}
int empty(struct queue *q)
{
       if (q–>front = = –1)
                return 1;
       else
                return 0;
}
```

2. Depth First Search (DFS) Method: (1) A depth first search of a graph differs from a breadth first search in that the exploration of a vertex k is suspended as soon as a new vertex is reached.

(2) At this time the exploration of the new vertex g begins and its exploration is also suspended as soon as a new vertex is reached.

(3) The above process is repeated recursively and when this new vertex has been explored, the previous vertex exploration continues.

(4) The search terminates when all reached vertices have been fully explored.

The list of unexplored vertices as a stack can be represented by stack. It is based on **LIFO** system. Depth first search for the graph in Fig. 8.8 is as follows in stack.

The steps of over search follow as start node 1

1. Initially, push 1 onto stack as follows:
   Stack: 1

2. Pop the top element 1 from stack and push onto stack all adjacent nodes of 1 which is not visited.
   Stack: 2,3,4,5
3. Pop the top element 5 from stack and push onto stack all adjacent nodes of 5 which is not visited.
   Stack: 2,3,4
4. Pop the top element 4 from stack and push onto stack all adjacent nodes of 4 which is not visited.
   Stack: 2,3,7,8
5. Pop the top element 8 from stack and push onto stack all adjacent nodes of 8 which is not visited.
   Stack: 2,3,7
6. Pop the top element 7 from stack and push onto stack all adjacent nodes of 7 which is not visited.
   Stack: 2,3,6
7. Pop the top element 6 from stack and push onto stack all adjacent nodes of 6 which is not visited.
   Stack: 2,3,9
8. Pop the top element 9 from stack and push onto stack all adjacent nodes of 9 which is not visited.
   Stack: 2,3
9. Pop the top element 3 from stack and push onto stack all adjacent nodes of 3 which is not visited.
   Stack: 2
10. Pop the top element 2 from stack and push onto stack all adjacent nodes of 2 which is not visited.
    Stack: empty
    Depth first search sequence is 1, 5, 4, 8, 7, 6, 9, 3, 2

Note that DFS sequence may depend on the order in which direction we go in depth, it may be different also. Here for the simplicity we have assumed the depth from the order of nodes as inserted in adjacency list of the graph. The last node of the list represents the top of the stack, so in the stack the top element is explored subsequently.

Algorithm of Depth First Search:

1. Non Recursive
2. Recursive

A depth first search of G carried out beginning at vertices g for any node i, visited[i] =1 if i has already been visited. Initially, no vertex is visited so for all vertices visited[i] = 0.

Algorithm DFS(g)
/* Non Recursive */
Step 1 : h = g;
Step 2 : visited[g] = 1;
Step 3 : repeat
{

```
Step 4 :  for all vertices  w  adjacent  from h do
            {
              if (visited[w] = 0)  then
              {
                    push w  to  s;   //  w is unexplored
                   visited[w] = 1;
              }                        //  end of the if
            }        // end of the for loop
Step 5 :  if  s is empty  then  return;
             pop  h from s;
          }  until(false);
           //  end of the repeat
Step 6 :  end  DFS.
```

The function for depth first search is given below:

```
void dfs(int a[MAXSIZE][MAXSIZE], int g, int n)
{
    int i,j, visited[MAXSIZE],h;
    for(i = 0;i<n;i++)
    visited[i] = 0;
    h = g ;
    visited[h] = 1;
    printf("\n DFS sequence with start vertex is –>");
    while(1)
    {
    printf(" %d ", h);
     for (j = 0; j<n; j++)
      {
            if ((a[h][j] == 1) && (visited[j] == 0))
                  {
                        push(&s, j);
                        visited[j] = 1;
                  }
     }
            if (empty(&s))
                  return;
            else
                  h = pop(&s);

    } /* end of while */
} /* end DFS */
```

```
        Algorithm DFS(g)
        /* Recursive */
Step 1 :  visited[g] = 1;
```

Step 2 : for each vertex w adjacent from g do
{
if(visited[w] = 0) then
DFS(w);
}
Step 3 : end DFS.

The time complexity of DFS algorithm is O(n + |E|) if G is represented by adjacency lists, where n is number of vertices in G and |E| number of edges in G. If G is represented by adjacency matrix, then $T(n,e) = O(n^2)$ and space complexity remains S(n,e) = O(n) same.

BFS and DFS are two fundamentally different search methods. In BFS a vertex is fully explored before the exploration of any other node begins. The next vertex to explore is the first unexplored vertex remaining. In DFS the exploration of a vertex is suspended as soon as a new unexplored vertex is reached. The exploration of this vertex is immediately begun.

A complete 'C' program for depth first search traversal is given below:

/* Program dfs.cpp is depth first search traversal for any graph whose input is in the form of adjacency matrix and its uses stack operation for non–recursive implementation */

GRAPH/DFS.CPP

```
# include<stdio.h>
#define MAXSIZE 10
#define TRUE 1
#define FALSE 0
int pop(struct stack *);
void push(struct stack *, int);
int empty(struct stack *);
struct stack{
    int top;
    int items[MAXSIZE];
}s;
void adjmat(int[MAXSIZE][MAXSIZE], int);
void dfs(int[MAXSIZE][MAXSIZE],int,int);
void main()
{
    int a[MAXSIZE][MAXSIZE], v,g;
    s.top = –1;     /* Initialize stack to empty */
    printf("Enter number of vertices in graph\n");
    scanf("%d",&v);
    adjmat(a,v);
    printf("Enter start vertex [0–%d]\n",v–1);
    scanf("%d",&g);
    dfs(a,g,v );
    getch( );
} /* end of main */
```

```
/* Input function */
void adjmat(int a[MAXSIZE][MAXSIZE], int n)
{
    int i, j;
    printf("\n Input adjacency matrix of order %dx%d\n",n,n);
    for(i = 0; i < n; i++)
            for(j = 0; j < n; j++)
                    scanf("%d", &a[i][j]);
}
void dfs(int a[MAXSIZE][MAXSIZE], int g, int n)
{
    int i,j, visited[MAXSIZE],h;
    for(i = 0;i<n;i++)
    visited[i] = 0;
    h = g ;
    visited[h] = 1;
    printf("\n DFS sequence with start vertex is –>");
    while(1)
    {
    printf(" %d ", h);
     for (j = 0; j<n; j++)
      {
            if ((a[h][j] == 1) && (visited[j] == 0))
                    {
                            push(&s, j);
                            visited[j] = 1;
                    }
     }
            if (empty(&s))
                    return;
            else
                    h = pop(&s);

    } /* end of while */
} /* end DFS */

/* function pushes the integer element into stack s */
void push(struct stack *s, int d)
{
    if (s–>top = = MAXSIZE –1)
    {
            printf("%s", "stack overflow");
```

```
            exit(0);
        }
        else
            s–>items[++(s–>top)] = d;
        return;
    } /* end of push function */

    /* function pop the integer element from stack s */
    int pop(struct stack *s)
    {
        if (empty(s))
        {
            printf("\n%s", "stack underflow");
            exit(0);
        }
        return(s–>items[s–>top– –]);
    } /* end of pop function */

    int empty(struct stack *s)
    {
        if (s–>top = = –1)
          return (TRUE);
        else
         return (FALSE);
    } /*end of empty function */
```

Example of Directed Graph: The graph traversal techniques for connected directed graph is also breadth first search and depth first search. In directed graph, each edge has direction so it may be possible that a path is available from vertices i to j but not from j to i or there is no path from vertex i to j.

The breadth first search and depth first search sequence for the directed graph in Fig. 8.9 is given below.

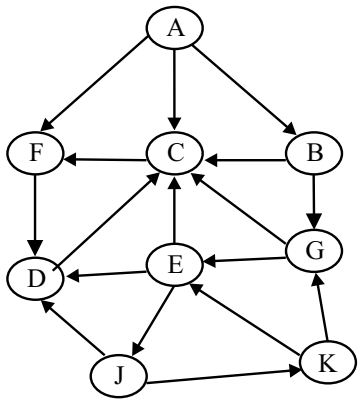


Figure 8.9 *Directed Graph, to find breadth first search and depth first search*

Breadth First Search

The steps of over search follow as start node A.

1. Initially, add A to queue as follows:
   Front = A Queue: A
   Rear = A
2. Delete the front element A from queue and add to queue adjacent nodes of A which is not visited.
   Front = B Queue: B, C, F
   Rear = F
3. Delete the front element B from queue and add to queue adjacent nodes of B which is not visited.
   Front = C Queue: C, F, G
   Rear = G
4. Delete the front element C from queue and add to queue adjacent nodes of C which is not visited.
   Front = F Queue: F,G
   Rear = G
5. Delete the front element F from queue and add to queue adjacent nodes of F which is not visited.
   Front = G Queue: G, D
   Rear = D
6. Delete the front element G from queue and add to queue adjacent nodes of G which is not visited.
   Front = D Queue: D, E
   Rear = E
7. Delete the front element D from queue and add to queue adjacent nodes of D which is not visited.
   Front = E Queue: E
   Rear = E
8. Delete the front element E from queue and add to queue adjacent nodes of E which is not visited.
   Front = J Queue: J
   Rear = J
9. Delete the front element J from queue and add to queue adjacent nodes of J which is not visited.
   Front = K Queue: K
   Rear = K
10. Delete the front element K from queue and add to queue adjacent nodes of K which is not visited.
    Front = 0 Queue: EMPTY
    Rear = 0
    Breadth first search sequence is A, B, C, F, G, D, E, J, K

Depth First Search:

The steps of over search follow as start node A

1. Initially, push A onto stack as follows
   Stack: A
2. Pop the top element A from stack and push onto stack all adjacent nodes of A which is not visited.
   Stack: B, C, F

2. Pop the top element F from stack and push onto stack all adjacent nodes of F which is not visited.
   Stack: B, C, D
3. Pop the top element D from stack and push onto stack all adjacent nodes of D which is not visited.
   Stack: B, C
4. Pop the top element C from stack and push onto stack all adjacent nodes of C which is not visited.
   Stack: B
5. Pop the top element B from stack and push onto stack all adjacent nodes of B which is not visited.
   Stack: G
6. Pop the top element G from stack and push onto stack all adjacent nodes of G which is not visited.
   Stack: E
7. Pop the top element E from stack and push onto stack all adjacent nodes of E which is not visited.
   Stack: J
8. Pop the top element J from stack and push onto stack all adjacent nodes of J which is not visited.
   Stack: K
9. Pop the top element K from stack and push onto stack all adjacent nodes of K which is not visited.
   Stack: empty
   DFS sequence is A, F, D, C, B, G, E, J, K.

8.4 MINIMUM COST SPANNING TREE

Definition: Let G = (V, E) be an undirected connected graph. A subgraph T = (V, E') of G is a spanning tree of G iff T is a tree.

The cost of a spanning tree is the sum of the costs of the edges in that tree. There are two greedy methods to find the spanning tree whose cost is minimum called minimum cost spanning tree.

A greedy method to obtain a minimum cost spanning tree would build this tree edge by edge. The next edge to include is chosen according to some optimization criterion. There are two possible ways for optimization criterion due to Prim's and Kruskal's algorithm. These algorithms are recommended for undirected graph.

In the first, the set of edges so far selected form a tree. Thus, if T is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be included in T is a minimum cost edge not in T with the property that A U { (v, u)} is also a tree. The corresponding algorithm is known as **Prim's algorithm.**

In the second optimization criteria the edges of the graph are considered in nondecreasing order of cost. The set T of edges so far selected for the spanning tree be such that it is possible to complete T into a tree. Thus T may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges T can be completed into a tree iff there are no cycles in T. This method is known as **Kruskal's algorithm.**

(1) Prim Method

Step 1 : Select the minimum cost edge and include in minimum spanning tree (MST) T.

Step 2 : Select Adjacent edges to {T} which is minimum in cost, include in T, until [n–1] edges.

Step 3 : And no cycles.

Prim's Algorithm: In Prim's algorithm we use the following steps:

Prim Algorithm (E, Cost, n, T)

/* E is the set of edges in G. cost[n][n] is the cost adjacency matrix of an n vertices graph such that cost(i,j) is either a positive real number or infinity if no edge(i,j) exists. A minimum spanning tree is computed and stored as a set of edges in the array T[n][2], edge in MST is T[i][1] to T[i][2].*/

Step 1 : Consider an edge (k, l) with minimum cost.

Step 2 : mincost = cost[k][l];

Step 3 : T[1][1] = k; T[1][2] = l;

Step 4 : for i = 1 to n do { /*initialize NEAR vector */

Step 5 : if cost [i][l]< cost[i][k] then
NEAR[i] = l;
else
NEAR[i] = k;

Step 6 : /*end of for at step4 */

Step 7 : NEAR[k] = NEAR[l] = 0;

Step 8 : for i = 2 to n–1 do{ /*find n–2 additional edges for T */
Let j be an index such that NEAR[j] ! = 0 and cost[j][NEAR[j]] is minimum.

Step 9 : T[i][1] = j; T[i][2] = NEAR[j];
mincost = mincost + cost[j][NEAR[j]];
NEAR[j] = 0;

Step 10 : for k = 1 to n do { /*update NEAR */
If NEAR[k] ! = 0 and cost [k][NEAR[k]] > cost[k][j]
then NEAR[k] = j;

Step 11 : } /*end of for at step 10*/

Step 12 : } /*end of for at step 8*/

Step 13 : if mincost > = Infinity then print "no spanning tree."; / * infinity is a very high number */
else
return mincost;

Step 14 : end Prim

The time complexity of prim's is $O(n^2)$ where n is the number of vertices in the graph, since the algorithm contains a nested for loop.

Prim's algorithm can be made more efficient by maintaining the graph using adjacency lists and keeping a priority queue of the nodes not in the partial tree. The first inner loop can then be replaced by removing the minimum distance node from the priority queue and adjusting the queue.

The second inner loop simply traverses on adjacency list and adjust the position of any nodes whose distance is modified in the priority queue.

The order of this implementation is $O(n + |E|) \log_2 n$, where |E| denotes the number of edges in the graph.

The algorithm will start with a tree that includes only a minimum cost edge of G. Then, edges will be added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included and the cost of (i, j), cost(i, j) is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree.

In order to determine this edge(i, j) efficiently, we shall associate with each vertex j is not included in the tree a value NEAR[j]. NEAR[j] is a vertex in the tree such that cost(j, NEAR[j]) is minimum among all choices for NEAR[j]. We have defined NEAR[j] = 0 for all vertices j that are already in the tree. The next edge to include is defined by the vertex j such that NEAR[j] <>0 (j not already in the tree) and cost(j, NEAR[j]) is minimum.

1. Select the minimum cost edge (k, l) i.e., (3,4) and include it in minimum spanning tree T.
2. Remainder of the spanning tree is built up edge by edge. Select (j, NEAR[j]) as the next edge to include.
3. Update NEAR[ ] vector and repeat the process.

The time required by PRIM is $O(n^2)$ where n is the number of vertices in the graph G.

Example 1. Explain the working of Prim's and Kruskal's methods on the undirected graph of figure 8.10.

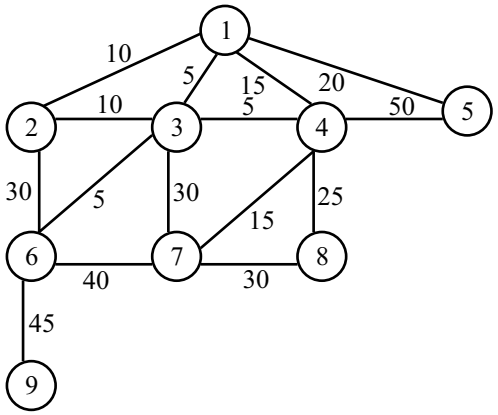
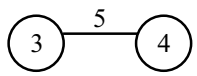
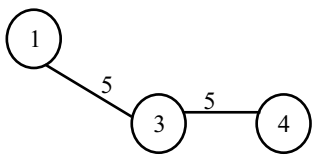
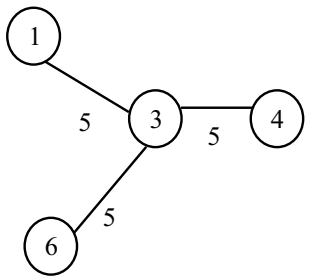


Figure 8.10 *Undirected graph G to compute MST.*

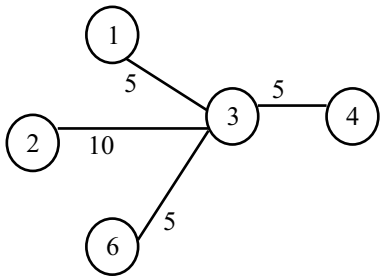
Initially minimum cost spanning tree T is NULL. The different stages in Prim's is given in Fig. 8.11 below.

Edge	Cost	Spanning tree (T)
(3, 4)	5	
$T = T \cup \{3, 4\}$		
(3, 1)	5	
$T = T \cup \{3, 1\}$		

(3, 6) 5
T = T ∪ {3, 6}

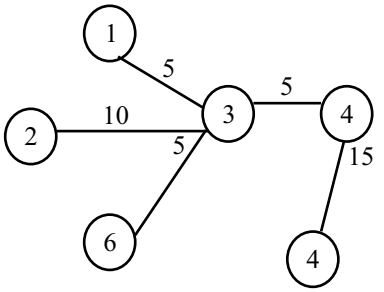


(3, 2) 10
T = T ∪ {3, 2}



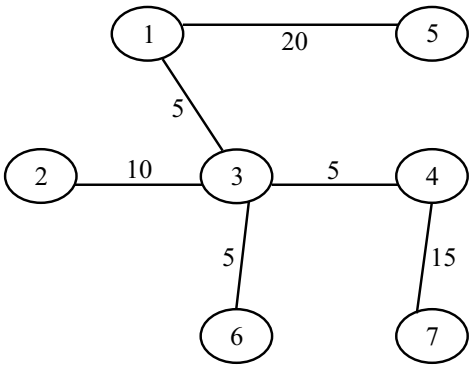
(1, 2) 10 Reject (due to formation of cycle)

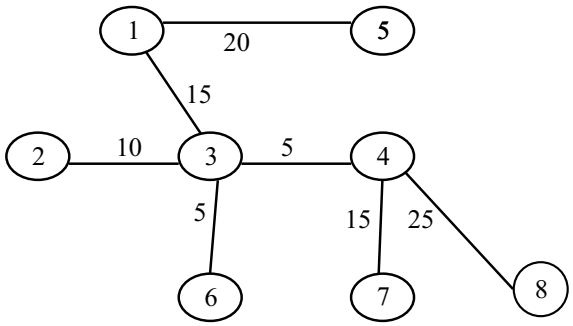
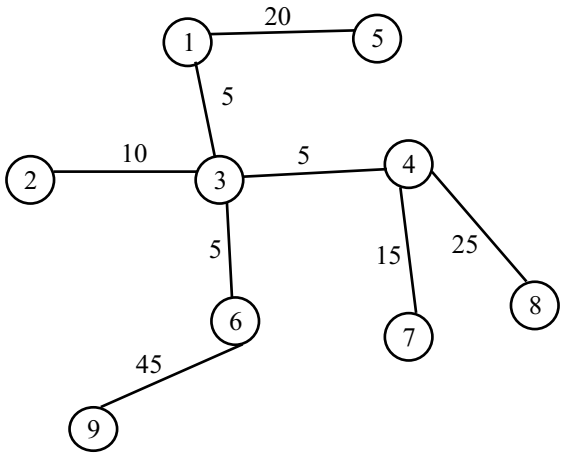
(4, 7) 15
T = T ∪ {4, 7}



(1, 4) 15 Reject (due to formation of cycle)

(1, 5) 20
T = T ∪ {1, 5}



(3, 7)	20	Reject (due to formation of cycle)
(4, 5)	20	Reject (due to formation of cycle)
(4, 8) T = T ∪ {4, 8}	25	
(2, 6)	30	Reject (due to formation of cycle)
(3, 7)	30	Reject (due to formation of cycle)
(6, 9) T = T ∪ {6, 9}	45	

Total cost of minimum spanning tree is 130.

Figure 8.11 *Prim's stage to compute MST*

(2) Kruskal Algorithm

In Kruskal algorithm, cost of edges are maintained as a minimum heap and each vertex is assigned to a distinct set (and hence to a distinct tree). T is the set of edges to be included in the minimum cost spanning tree, while i is the number of edges in T. The tree T can be represented as a sequential list using two- dimensional array T[n][2].

The edge (u,v) may be added to T by the assignment T[i][1] = u; and T[i][2] = v;

In the next step, edges are removed from heap one by one in nondecreasing order of cost.

If one vertex of an edge (u,v) is not in T that means it has distinct set, then edge is included in T.

The set containing u and v can be combined. If (u =v) the edge is discarded as the inclusion in T will create a cycle.

In last step, check that i # n–1 iff the graph G is not connected.

The time complexity of Kruskal's algorithm is $O(|E| \log_2|E|)$, where |E| denotes the number of edges in the graph.

Kruskal Method

In Kruskal method we use the following method:

Step 1 : Construct a minimum heap of the edges according to their cost.

Step 2 : Delete the minimum cost edge from heap and accept the edge if no cycle, otherwise reject the edge.

Step 3 : Repeat steps 1 and 2 until when |E|=|V–1| or heap empty.

```
Kruskal Algorithm(E, cost, n, T)
```

/* E is the set of edges in G. cost[n][n] is the cost adjacency matrix of an n vertices graph such that cost(i,j) is either a positive real number or infinity if no edge(i,j) exists. A minimum spanning tree is computed and stored as a set of edges in the array T[n][2], edge in MST is T[i][1] to T[i][2].*/

Step 1 : Construct a heap out of the edge costs using HEAPIFY.

Step 2 : for i = 1 to n do
PARENT[i] = –1 /*each vertex in a different tree set */

Step 3 : i = mincost = 0; /* Initialize variables */

Step 4 : while i<n–1 and heap not empty do
Delete a minimum cost edge(u,v) from the heap and reheapify using ADJUST.

Step 5 : j = FIND(u);
k = FIND(v);

Step 6 : if j <>k then
i = i + 1;
T[i][1] = u;
T[i][2] = v;
mincost = mincost + cost[u][v];
UNION(j, k);

Step 7 : repeat /* end of while at step4 */

Step 8 : if i <>n–1 then print "no spanning tree";
else return (mincost);

Step 9 : end Kruskal

Algorithm HEAPIFY(a, n)

Step 1 : for(i = n/2;i> = 1;i– –)
ADJUST(a, i, n);

Step 2: end HEAPIFY

```
        Algorithm ADJUST(a, i, n)
Step 1 :    j = 2i;
            Item = a[i];
Step 2 :    while(j< = n)
              {
                 if(j<n) and (a[j]<a[j+1])
                        j = j+1;
               if(item> = a[j])
                    break;
                        a[j/2] = a[j];
               } /* end of while at step2*/
                 j = 2j;
              a[j/2] = item;
Step 3 :    end ADJUST

        Algorithm UNION(i, j)
Step 1 :    PARENT[i] = j;
Step 2 :    end UNION

        Algorithm FIND(i)
Step 1 :    while(PARENT[i] > = 0) do i = PARENT[i];
            return i;
Step 2 :    end FIND
```

1. Initially, a minimum heap is constructed using heapify function with minimum cost edge is root of the tree. Each vertex is assigned to a distinct set (and hence to a distinct tree). T is the set of edges to be included in the minimum cost spanning tree, while i is the number of edges in T. T itself may be represented is a two-dimensional array T[n][2]. Edge (u,v) may be added to T by the assignment T [i][1] = u and T [i][2] = v.
2. Remove the root node and reheap the remaining set and determine j = FIND (u) and k = FIND (v). If j <> k then vertex u and v are different sets and edge(u,v) is included into T.
3. The sets containing u and v are combined using UNION. If u = v the edge(u,v) is discarded as its inclusion into T will create a cycle.
4. If i<>n–1 then no spanning tree iff graph G is not connected. Otherwise mincost return minimum cost of tree and T consists of edges of minimum cost.

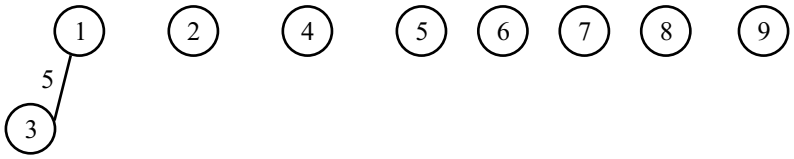
The complexity of Kruskal is $O(|E| \log_2|E|)$ when |E| number of edges.

The spanning tree obtained is shown below.

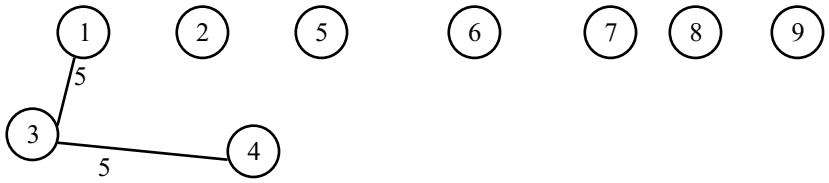
(*i*) Initially, each vertex is assigned to a distinct set (and hence to a distinct tree). Here, 9 distinct trees, each one having single vertex itself as root of the tree.

(*ii*) Create minimum heap for the edges with their cost and remove minimum cost edge i.e., (1, 3) and is included in T as both vertices are from distinct tree set.

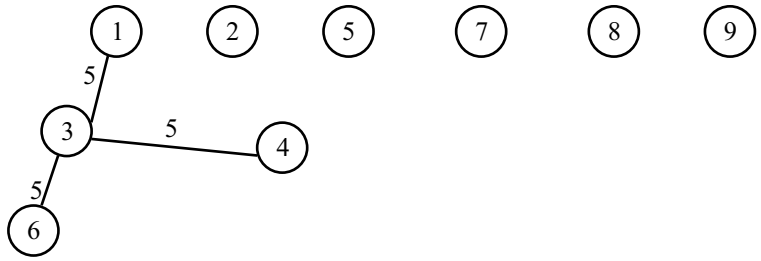
(*Figure 8.12–contd...*)



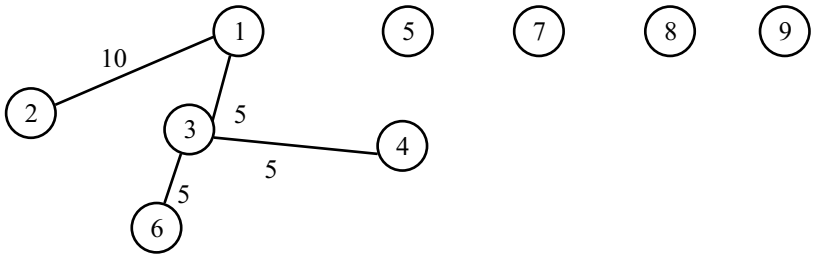
(*iii*) Reheapify the remaining edges with their cost and remove minimum cost edge i.e., (3, 4) and is included in T as both vertices are from distinct tree set.



(*iv*) Reheapify the remaining edges with their cost and remove minimum cost edge (3, 6) and is included in T as both vertices are from distinct tree set.

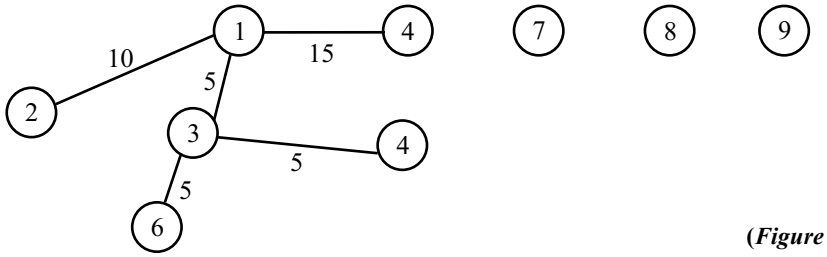


(*v*) Reheapify the remaining edges with their cost and remove minimum cost edge i.e., (1, 2) and is included in T as both vertices are from distinct tree set.



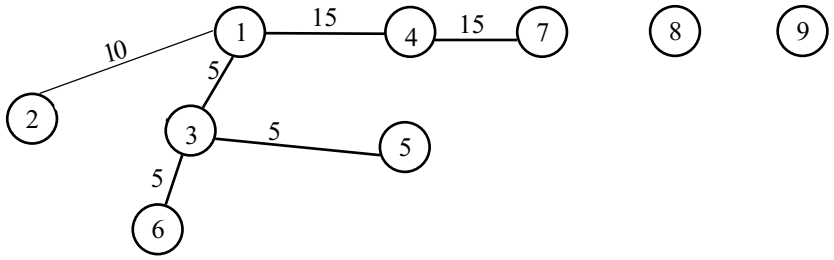
(*vi*) Reheapify the remaining edges with their cost and remove minimum cost edge i.e., (2, 3) and in not included in T as both vertices are from the same tree set.

(*vii*) Reheapify the remaining edges with their cost and remove minimum cost edge i.e., (1, 4) and is included in T as both vertices are from distinct tree set.



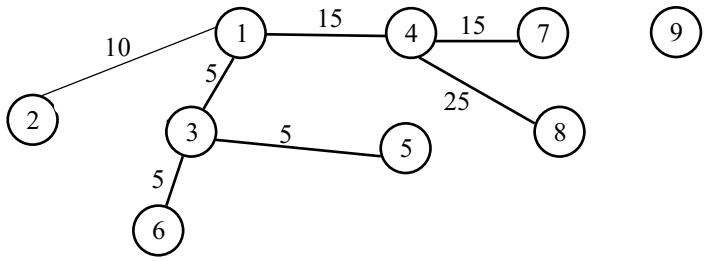
(***Figure 8.12–contd...***)

(*viii*) Reheapify the remaining edges with their cost and remove minimum cost edge, i.e., (4, 7) and is included in T as both vertices are from distinct tree set.



(*ix*) Reheapify and remove the edge i.e., (1, 5) and is not included in T as both vertices are from the same tree set.

(*x*) Repeat the above process, and edge i.e., (4, 8) and is included in T as both vertices are from distinct tree set.



(*xi*) Reheapify and remove edge (2, 6) and is not included in T as both vertices are from the same tree set.

(*xii*) Reheapify and remove edge (3, 7) and is not included in T as both vertices are from the same tree set.

(*xiii*) Reheapify and remove edge (7, 8) and is not included in T as both vertices are from the same tree set.

(*xiv*) Reheapify and remove edge (6, 7) and is not included in T as both vertices are from the same tree set.

(*xv*) Reheapify and remove edge (6, 9) and is included (i.e., union) with T as both vertices are in distinct tree set.

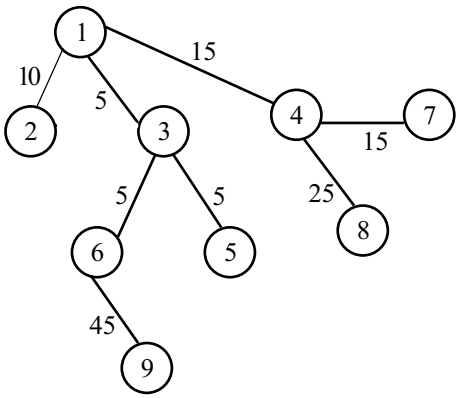


Figure 8.12 *Various stages in Kruskal's algorithm for Figure 8.10*

Now the while loop will terminate as number of edges in T is 8 and minimum cost of spanning tree is 125.

Example 2. Consider the following directed graph in Fig. 8.13, find the minimum cost spanning tree using Prim's algorithm and Kruskal's algorithm.

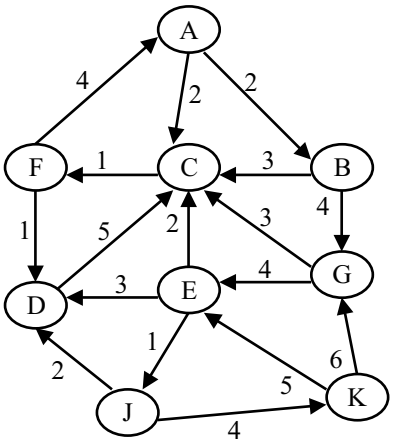
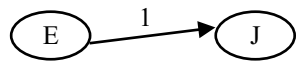
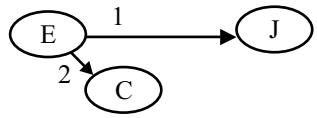


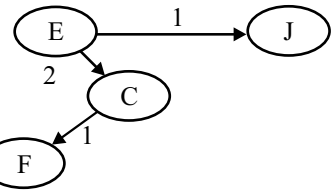
Figure 8.13 *Directed graph G*

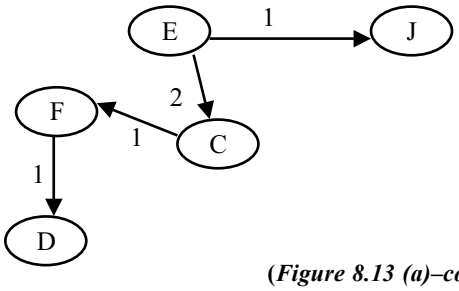
Prim's Algorithm

	Edge	Cost	Spanning tree
1.	(E, J)	1	
2.	(E, C)	2	
3.	(C, F)	1	
4.	(F, D)	1	



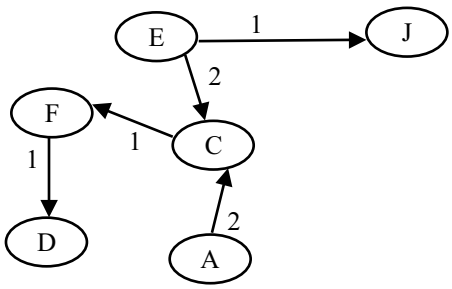




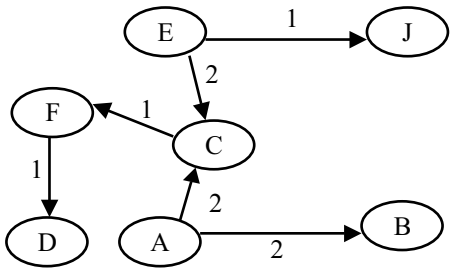


(***Figure 8.13 (a)–contd...***)

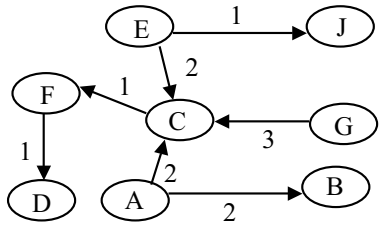
5. (A, C) 2



6. (A, B) 2



7. (G, C) 3



8. (J, K) 4

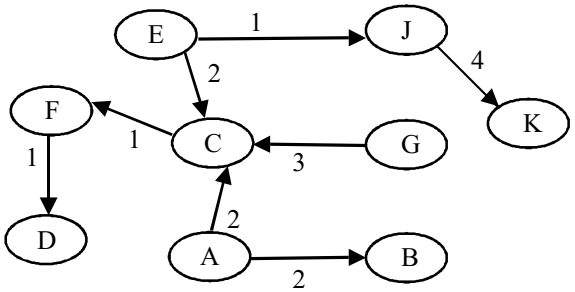
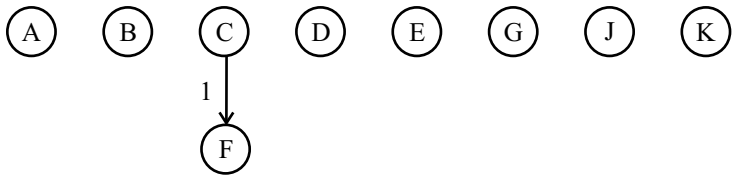


Figure 8.13 (*a*) *Prim's algorithm, different stages.*

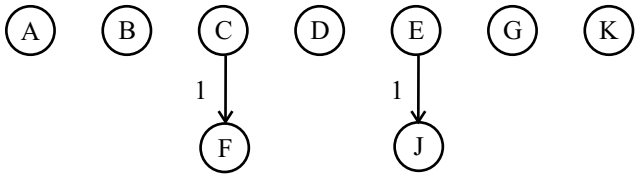
The spanning tree obtained by applying Kruskal algorithm is given below.

(*i*) Initially, each vertex is assigned to a distinct set (and hence to a distinct tree). Here, 9 distinct trees, each one having single vertex itself as root of the tree.

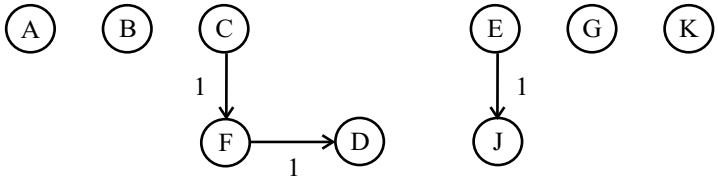
(*ii*) Select minimum cost edge (C, F) and is included in MST T.



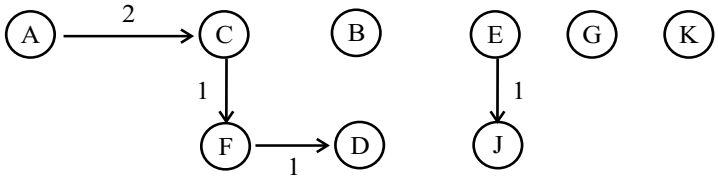
(*iii*) Apply the same procedure as we did for undirected graph. The next edge (E, J) is included



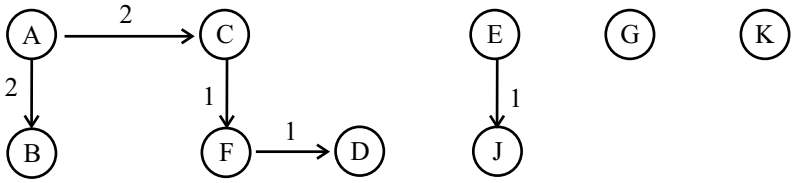
(*iv*) The next edge (F, D) is included in T



(*v*) The next edge (A, C) is included in T

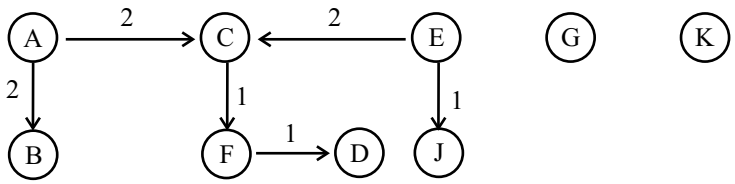


(*vi*) The next edge (A, B) is included in T

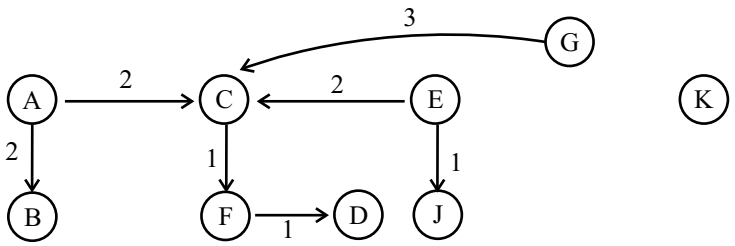


(***Figure 8.13 (b)–contd...***)

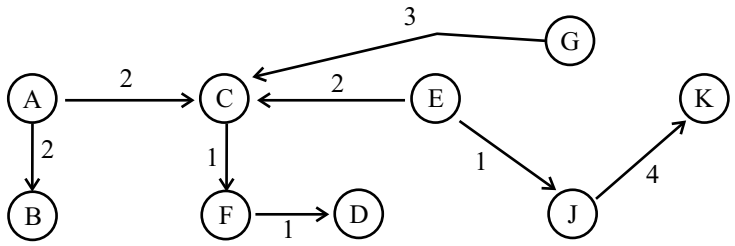
(*vii*) The next edge (E, C) is included in T



(*viii*) The next edge (J, D) is not included as both vertices are from the same true set.
(*ix*) The next edge (B, C) is not included as both vertices are from the same tree set.
(*x*) The next edge (E, D) is not included as both vertices are from the same tree set.
(*xi*) The next edge (G, C) is included in T



(*xii*) The next edge (B, G) is not included as both vertices are from the same tree set.
(*xiii*) The next edge (F, A) is not included as both vertices are from the same tree set.
(*xiv*) The next edge (G, E) is not included as both the vertices are from the same tree set.
(*xv*) The next edg (J, K) is included in T.



This is the spanning tree as (n–1) i.e., 8 edges are included in T where n is number of vertices i.e. 9.

Fig. 8.13 (*b*) *Kruskal algorithm, different stages*

8.5 BICONNECTIVITY

A vertex v in a connected graph G is an **articulation point** if and only if the deletion of vertex v together with all edges incident to v disconnects the graph into two or more nonempty components.

A graph G is **biconnected** if and only if it contains no articulation points. The presence of articulation point in a connected graph can be an undesirable feature in many cases. For example, if graph G represents a communication network with the vertices representing communication stations i that is an articulation point would result in the loss of communication point other than i too.

On the other hand, if G has no articulation point, then if any station i fails, we can still communicate between every two stations not including station i.

We developed an efficient algorithm to test whether a connected graph is biconnected. For the cases of graph that is not biconnected, this algorithm will identify all the articulation points.

Lemma: Two biconnected components can have at most one vertex in common and this vertex is an articulation point as described in Fig. 8.14.

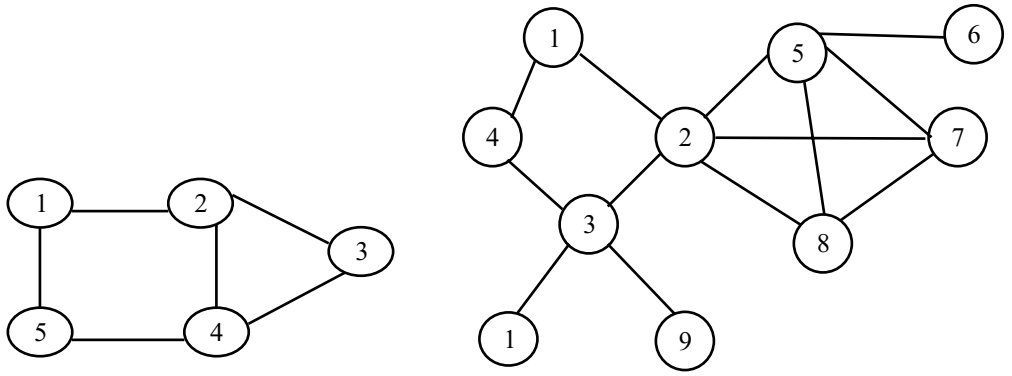


Figure 8.14 *(i) A biconnected graph (ii) Two biconnected component, vertex 2 is an articulation vertex.*

Lemma: Let G = (V, E) be a connected undirected graph, and let S = (V,T) be depth first spanning tree for G. Vertex x is an articulation point if and only if either

1. x is the root and x has more than one son or
2. x is not the root, and for some sons of x there is no back edge between any descendent of s (inclosed s itself) and a proper ancestor for G.

Now consider the problem of identifying the articulation points and biconnected components of a connected graph G with n > = 2 vertices. Considering a depth first spanning tree of G efficiently solves the problem.

Fig. 8.15 (*ii*) shows a depth first spanning tree of the graph of Fig. 8.15 (*i*). The numbers inside each vertex correspond to the order in which a depth first search visits these vertices and are referred to as the depth first numbers (dfns) of the vertex. Thus, dfn[v1] = 1, dfn[v2] = 2, dfn[v6] = 4 as given below.

DEPTH FIRST SEARCH vertices	V1	V2	V4	V6	V7	V8	V9	V5	V3
DFN number	1	2	3	4	5	6	7	8	9

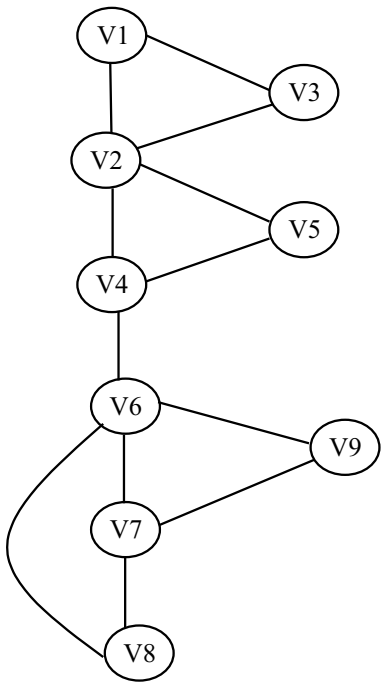
In Figure 8.15 (*ii*) solid edges form the depth first spanning tree. These edges are called **tree edges** and broken edges are called **back edges.**

The simple rule to identify articulation points, defines L[u] as follows:

L[u] = min { dfn[u], min{L[w] where w is a child of u}, min { dfn[w] where (u, w) is a back edge}}

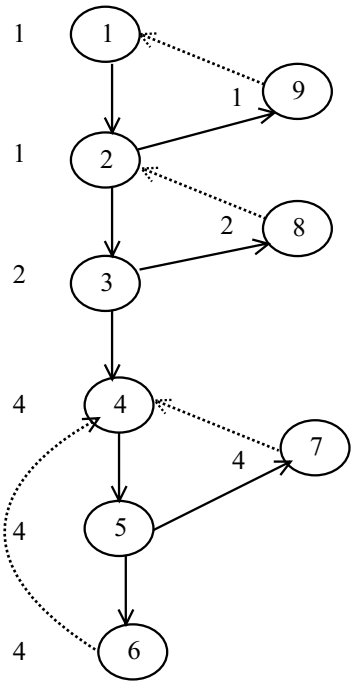
The L[u] is the lowest depth first number(dfn) that can be reached from u using a path of descendents followed by at most one back edge.

For spanning tree of Fig. 8.15 (*ii*) the L values are L[1–9] = {1, 1, 2, 4, 4, 4, 4, 2, 1}. Vertex V4 is an articulation point as child of V4 (e.g., V6) has L[V6] = 4 and dfn[V4] = 3. The biconnected components are shown in Fig. 8.15 (*iii*).

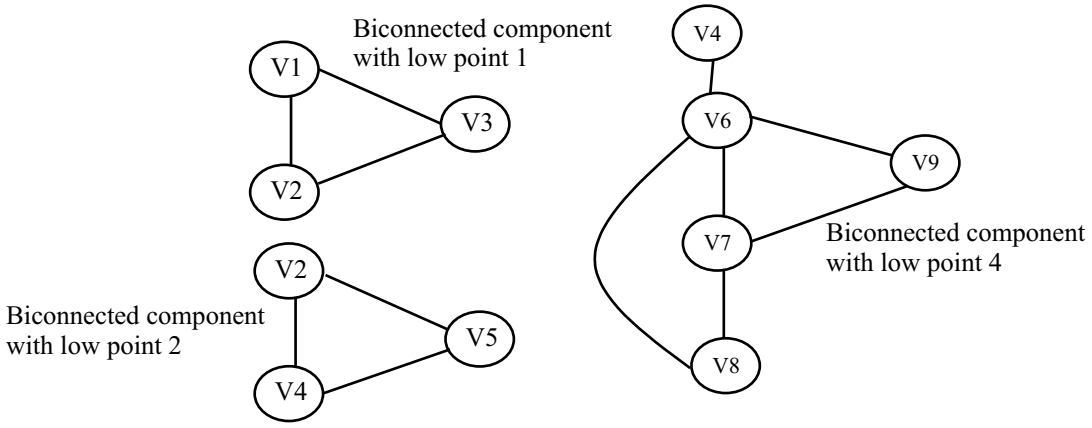


(*i*) Graph

low point number L[u]



(*ii*) DF number with low point L



(*iii*) Biconnected components of Graph with low points 1, 2 and 4.

Figure 8.15 *Biconnectivity and articulation point*

Algorithm Biconnectivity(a,f)

/* a is a start vertex for DEPTH FIRST SEARCH, f is its parents if any in the depth first spanning tree. It is assumed that the global array dfn is initially zero and n is number of vertices in G. Global variable count is initialized to 1*/

```
Step 1 :  dfn[a] = count;
          L[a] = dfn[a];
          count = count+1;
Step 2 :  for each vertex w adjacent from a do
           {
                    if(a<>w) and (dfn[w]<dfn[a]) then
                    add[a,w] to the top of  the stack s;
                    if(dfn[w] = 0) then
                    {
                      if(L[w]> = dfn[a]) then
                       {
                         write ("New bicomponent");
                           repeat
                               {
                                  delete an edge from the top of stack s;
                                    let this edge be(x,y);
                                  write(x,y);
                                }until(((x,y) = (a,w)) or ((x,y) = (w,a)));
                    } /* end of if */
                            Biconnectivity(w,a)  /* w is unvisited, calls biconnectivity recursively */
                              L[a] = min(L[a],L[w]);
                    }/* end of if */
                              else if (w<>f) then L[a] = min(L[a],dfn[w]);
            } /* end of for loop */
Step 3 :  end Biconnectivity
```

The time complexity of the algorithm is O(n + e), where e is the number of edges and n number of vertices in graph G.

Example 1. Find the biconnected component and articulation point for the following undirected graph in Fig. 8.16.

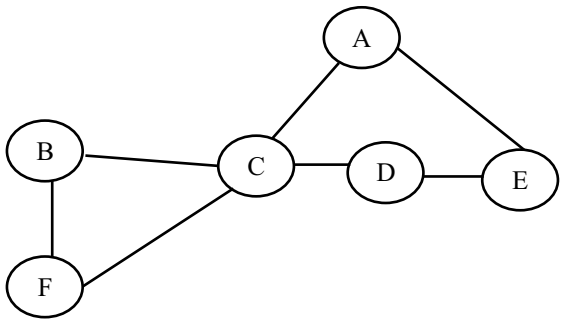


Figure 8.16 *Undirected graph*

Depth first search sequence from vertex A is A, C, B, F, D, E

The depth first number in the depth first search tree is given below in Fig. 8.16 (*i*)

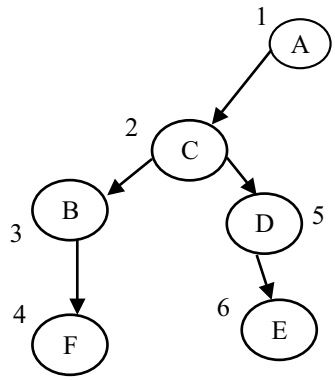


Figure 8.16 (*i*) *DFN number*

Compute the low point L[u] is the lowest depth first number (dfn) that can be reached from u using a path of descendents followed by at most one back edge.

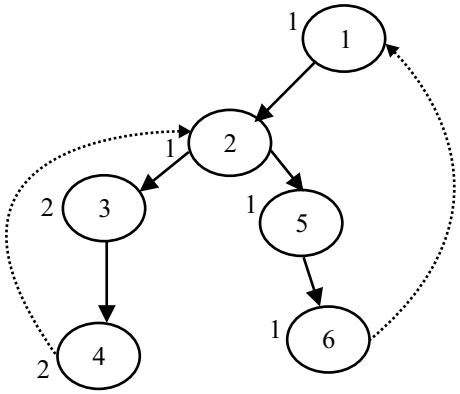
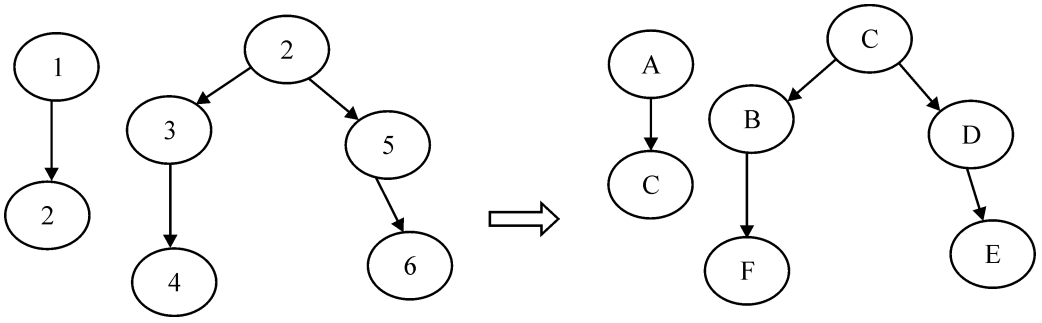


Figure 8.16 (*ii*) *Low point computation*

Thus the vertex 2 is the only articulation point. Therefore, two biconnected components of the graph as given in the Fig. 8.17.



(*i*) (*ii*)

Figure 8.17 *Biconnected components (i) with DFN number (ii) with the original graph*

8.6 STRONG CONNECTIVITY

A graph is said to be strongly connected if there exists a path from each vertex to every other vertex in the graph. As an application of breadth first search consider the problem of obtaining a spanning tree for an undirected graph G. The graph G has a spanning tree iff G is connected. A complete traversal of the graph can be made by repeatedly calling BFS algorithm with a new unvisited starting vertex. This algorithm is known as breadth first traversal. The algorithm is given below:

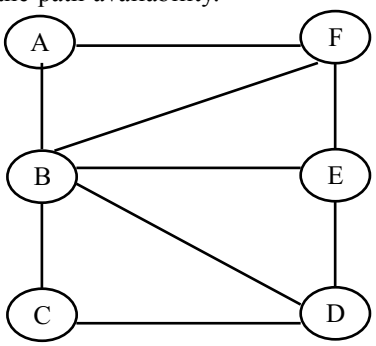
```
Algorithm BFT( G, n)
/* Breadth first traversal of undirected connected graph G, n is the number of vertices and visited is
global array initially consists of zero */
Step 1 :  for i = 1 to n do
                visited[i] = 0;
Step 2 :  for i = 1 to n do
               if (visited[i] = 0) then
                              BFS(i);
               end if
Step 3 :  end BFT
```

A depth first traversal of a graph is carried out by repeatedly calling DFS, with a new unvisited starting vertex each time. The algorithm for this DFT differs from BFT only in that the call to BFS(i) is replaced by a call to DFS(i).

The algorithm is given below:

```
Algorithm DFT(G, n)
/* Depth first traversal of undirected connected graph G, n is the number of vertices and visited is global
array initially consists of zero */
Step 1 :  for i = 1 to n do
                visited[i] = 0;
Step 2 :  for i = 1 to n do
               if (visited[i] = 0) then
                              DFS(i);
                    end if
Step 3 :  end DFT
```

Consider an undirected connected graph as given below in Fig. 8.18. To check the strong connectivity, find the spanning tree for each vertex. We can construct spanning trees either by breadth first search or depth first search for each node to check the path availability.



(***Figure 8.18–contd...***)

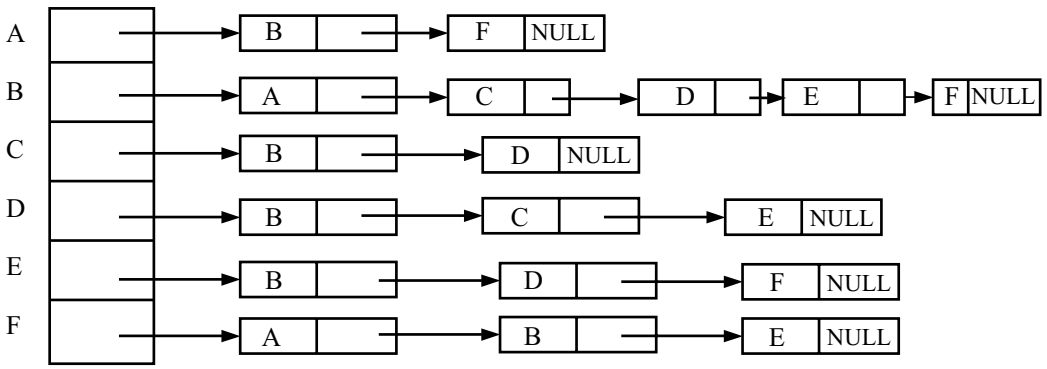
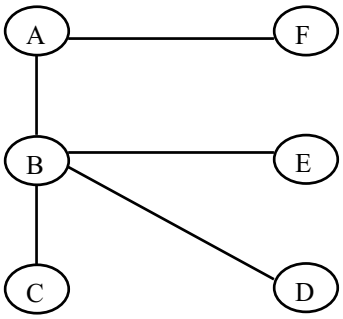
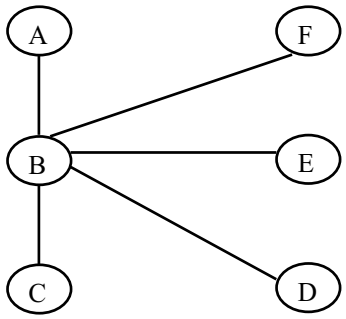


Figure 8.18 *Graph and its adjacency list*

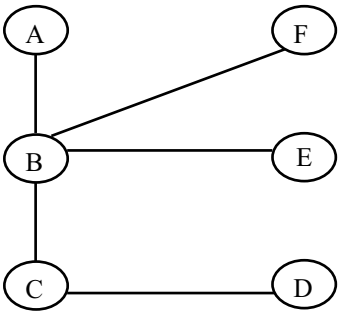
The spanning trees obtained using a breadth first search are calling breadth first spanning trees.



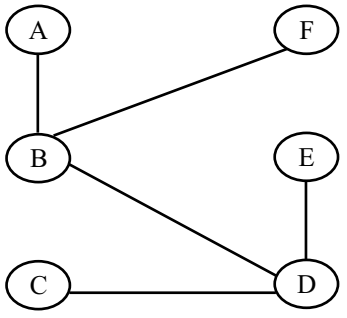
(*i*) BFS(a)



(*ii*) BFS(b)

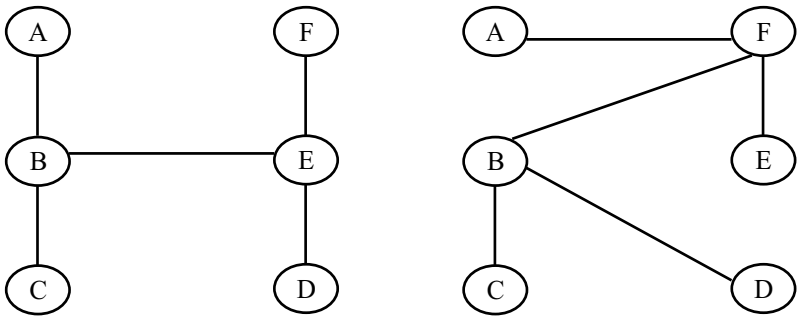


(*iii*) BFS(c)



(*iv*) BFS(d)

(*Figure 8.19–contd...*)

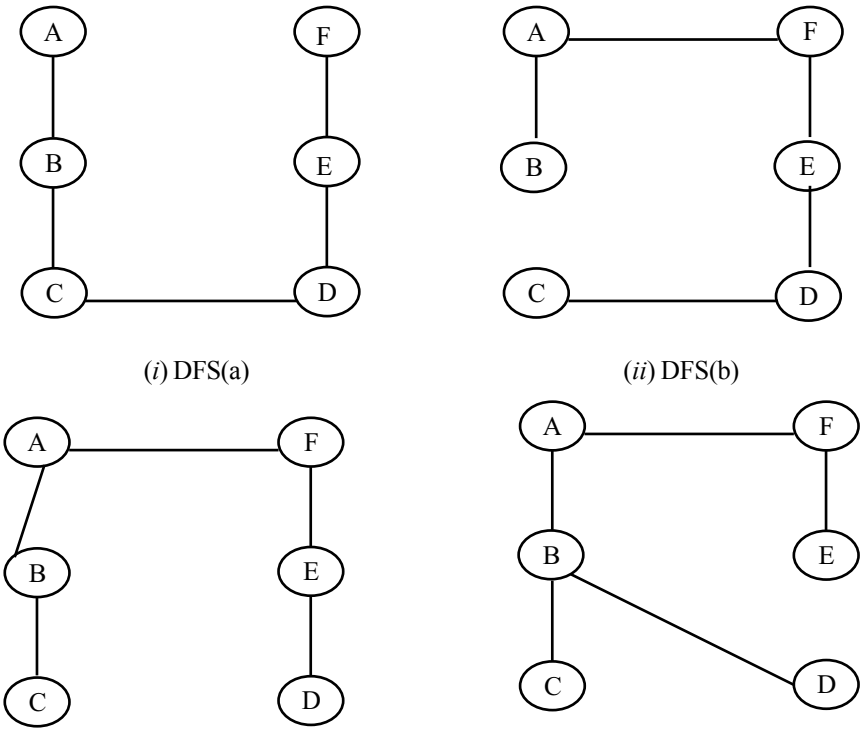


(*v*) BFS(e) (*vi*) BFS(f)

Figure 8.19 *BFS spanning trees for graph in figure 8.18*

The graph is strongly connected as the spanning tree is constructed from each vertex.

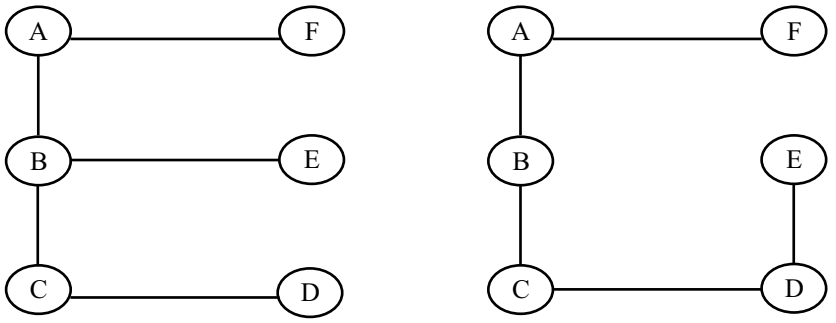
The spanning trees obtained using a depth first search are calling depth first spanning trees. The depth first search of the graph is as given below using adjacency lists.



(*i*) DFS(a) (*ii*) DFS(b)

(*iii*) DFS(c) (*iv*) DFS(d)

(*Figure 8.20–contd...*)



(*v*) DFS(e) (*vi*) DFS(f)

Figure 8.20 *DFS spanning trees for graph in figure 8.18*

The graph is strongly connected as the spanning tree is constructed from each vertex.

8.7 TRANSITIVE CLOSURE ALGORITHM

Transitive Closure

The graph G^+ which has the same vertex set as G, but has an edge from v to w if and only if there is a path (of length 0 or more) from v to w in G, is called the (reflexive) transitive closure of G.

A problem is closely related to shortest path. The cost of a path is defined to be the sum of the costs of the edges in the path. The shortest path problem is to find for each ordered pair of vertex (v, w) the lowest cost of any path from v to w.

Definition

A closed semi ring is a system (S,+,.,0,1) where S is a system, + (logical or) and **.** (logical and) are binary operators on S and elements are 0 and 1.

Let us assume that a graph (not weighted) is completely described by its adjacency matrix A. Consider a logical expression (A[i][k] && A[k][j]) its value is true if and only if the values of both A[i][k] and A[k][j] are true; which implies that there is an edge from node i to node k and edge from node k to node j. Thus A[i][k] and A[k][j] equal to true if and only if there is a path of length 2 from i to j passing through node k.

Consider a matrix A^2 such that A^2[i][j] is the value of the foregoing expression, A^2 is called the path matrix of length 2. Matrix A^2 indicates whether or not there is a path of length 2 between node i and j.

The matrix A^2 is the product of matrix A with itself, with numerical multiplication is replaced by logical and (.) operations, and addition is replaced by logical or (+) operation. Matrix A^2 is said to be boolean product of matrix A itself.

Similarly, define matrix A^3, the path matrix of length 3, as the boolean product of A^2 and A. Matrix A^3[i][j] equals to true if and only if there is a path of length 3 from nodes i to j.

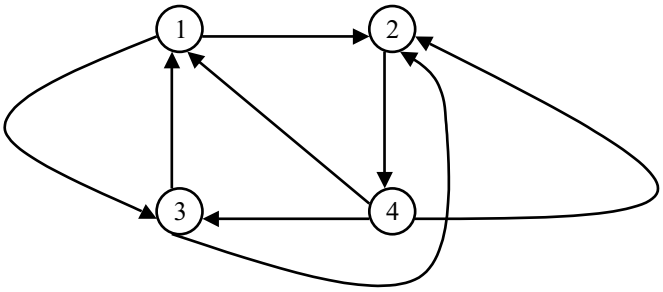
Now we can construct a matrix path of n such that path[i][j] is true if and only if there is some path from vertex i to vertex j of any length, maximum of n.

$$Path[i][j] = A[i][j] + A^2[i][j] + \ldots + A^n[i][j]$$

This is because if there is a path of length m > n, then there is a cycle in path, by removing the cycle we can obtain path of length n.

The matrix path is often called the transitive closure of the adjacency matrix A.

Consider the following directed graph in Fig. 8.21, compute the path matrix for the following.



A^1

	1	2	3	4
1	0	1	1	0
2	0	0	0	1
3	1	1	0	0
4	1	1	1	0

A^2

	1	2	3	4
1	1	0	0	1
2	1	1	1	0
3	0	1	1	1
4	1	1	1	1

A^3

	1	2	3	4
1	1	1	1	1
2	1	1	0	0
3	1	1	1	1
4	1	1	1	0

A^4

	1	2	3	4
1	1	0	0	1
2	0	1	1	0
3	0	1	1	0
4	1	1	1	1

Path matrix

Path[i][j] = $A^1 + A^2 + A^3 + A^4$

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1

Figure 8.21 *Transitive closure of the graph as path matrix*

The efficient way to compute transitive closure of a given graph is by **Warshal's algorithm**.

Let us define the matrix $path^k$ such that $path^k[i][j]$ is true if and only if there is a path from node i to node j that does not pass through any nodes numbered higher than k.

How to compute $path^{k+1}$ from $path^k$ matrix? The $path^{k+1}[i][j]$ equals true if and only if one of the following two conditions holds.

(*i*) $path^k[i][j]$ = true

(*ii*) $path^k[i][k+1]$ = true && $path^k[k+1][j]$ = true

An algorithm to obtain the matrix $path^k$ from matrix $path^{k-1}$ based on above condition as follows.

```
for(i = 0 ; i< n; i++)
        for( j = 0; j<n; j++)
                path^k[i][j] = path^(k-1)[i][j] + (path^(k-1)[i][k] . path^(k-1)[k][j])
```

Transitive Closure Function

The following 'C' function computes the closure.

```
void  tranclos(int A[n][n] , int path[n][n])
{
int i, j  , k;
for (i = 0; i <n; i++)
     for (j = 0; j <n; j++)
               path[i][j] = A[i][j];  /* initial adjacency matrix of graph as path matrix */
for (k = 0; k <n; k++)
for (i = 0; i <n; i++)
               if (path[i][k] == 1)
                         for (j = 0; j <n; j++)
                                   path[i][j] = path[i][j] || path[k][j];
} /* end tranclos */
```

The efficiency of transitive closure is $O(n^3)$.

8.8 SHORTEST PATH ALGORITHMS

In a weighted or cost graph, or network, it is frequently desired to find the shortest path among the vertices. The shortest path is defined as a path from i to j such that the sum of the costs of the edges on the path is minimized. We assume a cost function C_{ij} is the cost of the edge from i to j. If there is no edge from i to j, C_{ij} is set to an arbitrarily large value to indicate the infinite cost i.e., no path from i to j.

We have given four algorithms to compute shortest path from among the vertices depending upon the costs type and required paths.

1. Dijkstra's Algorithm

If all costs are positive, the Dijkstra's algorithm determines the shortest path from vertex i to j.

Here we discuss the **Dijkstra's algorithm** to find the shortest path from source to all vertices.

The main idea of Dijkstra's algorithm is to keep identifying the closest nodes from the source node in order of increasing path cost. The algorithm is iterative. At the first iteration the algorithm finds the closest node from the source node, which must be the neighbour of the source node if link costs are positive. At the second iteration the algorithm finds the second–closest node from the source node. At the third iteration the third–closest node must be the neighbour of the first two closest nodes and so on. Thus at the k^{th} iteration, the algorithm will have determined the k closest nodes from the source node.

Dijkstra's algorithm can be described as follows:

V = set of vertices in the graph

s = source node (i.e., vertex)

S = set of vertices currently included in shortest path, initially consisting of source vertex

l (w, v) = link cost from node w to node v, the cost is ∞ if the nodes are not directly connected

D(n) = least cost, from s to n nodes that is currently known to the algorithm.

The steps for the algorithm:

1. Initialize
   set D[S] = 0;
2. for each node v in V except source node do D[v] = l(s, v);
3. while S# V do // select minimum cost node

4. begin
5. Choose a node w in V–S such that D[w] is a minimum
6. Add w to S;
7. for each in V–S do // iterate for all nodes to re-compute the cost
   D[v] = Min (D[v], D[w] + l(w, v));
   end of loop
   end Dijkstra

For example, consider an undirected graph Fig. 8.22, where s is node one and set of nodes (1 to 6) is V, cost of w to v node is given. Compute the shortest path from source to all destinations.

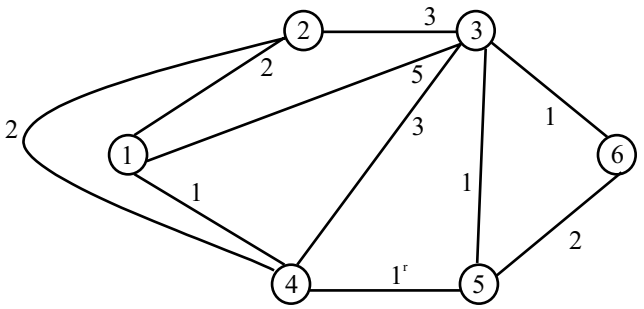


Figure 8.22

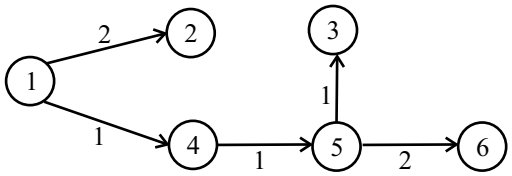
Initialize S = {1} the source node, D[1] = 0, and D[i] is 2, 5, 1, ∞ , ∞ for node 2, 3, 4, 5, and 6 respectively. In next iteration select the minimum cost link node that is node 4 and re-compute the cost for all nodes according to the algorithm (line 3–7).

The below Table lists all possible paths from source to all destinations with their cost and select the minimum.

Iteration	*Nodes travel currently* (*S*)	*D[2]*	*Path*	*D[3]*	*Path*
1.	{1}	2	1–2	5	1–3
2.	{1,4}	2	1–2	4	1–4–3
3.	{1,2,4}	[2]	1–2	4	1–4–3
4.	{1,2,4,5}	[2]	1–2	3	1–4–5–3
5.	{1,2,3,4,5}	[2]	1–2	[3]	1–4–5–3
6.	{1,2,3,4,5,6}	[2]	1–2	[3]	1–4–5–3

D(4)	*Path*	*D[5]*	*Path*	*D[6]*	*Path*
1	1–4	∞	–	∞	–
[1]	1–4	2	1–4–5	∞	–
[1]	1–4	2	1–4–5	∞	–
[1]	1–4	[2]	1–4–5	4	1–4–5–6
[1]	1–4	[2]	1–4–5	4	1–4–5–6
[1]	1–4	[2]	1–4–5	[4]	1–4–5–6

Note that square box indicates that node is mark permanent.



Example 1. Using Dijkstra's algorithm, find out a shortest path to all other nodes 1 through 6 for the graph in Fig. 8.23.

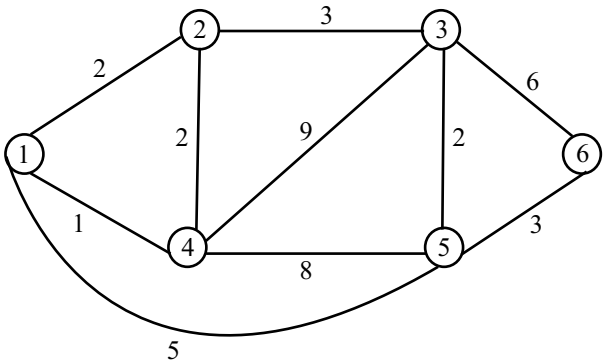


Figure 8.23

Dijkstra's algorithm can find least cost node 1 through node 6 as follows:

Initially, D_j = link cost from node w to node v, the cost is ∞ if the nodes are not directly connected. For example cost $D_2 = 2$, cost from node 1 to node 2, $D_3 = \infty$, as node 1 and node 3 are not directly connected and so on.

D_j = least cost, from s to j nodes that is currently known to the algorithm. The least cost computed $D_j = \min(D_j, D_i + l(i, j))$ in number of iteration to include the nodes in the source node. In iteration first minimum cost node 4 is added and the minimum costs for rest of nodes are computed. For example $D_3 = \min(D_3, D_4 + l(4,3)) = \min(\infty, 1+9)$ and so on. In next iteration minimum cost node 2 is added and the minimum costs for rest of nodes are computed. For example now $D_3 = \min(D_3, D_2 + l(2,3)) = \min(10, 2+3)$ and so on. In iteration 4, we get least cost through node 1 to node 6. For example the least cost between node 1 and node 6 is 8 through path 1–5–6.

The Fig. 8.24 depicts least cost computation for Dijkstra's algorithm.

Iteration	*Nodes currently*	D_2	D_3	D_4	D_5	D_6
Initial	1	2	∞	1	5	∞
1	1,4	2	10	[1]	5	∞
2	1,2,4	[2]	5	[1]	5	∞
3	1,2,3,4	[2]	[5]	[1]	5	11
4	1,2,3,4,5	[2]	[5]	[1]	[5]	9
5	1,2,3,4,5,6	[2]	[5]	[1]	[5]	[8]

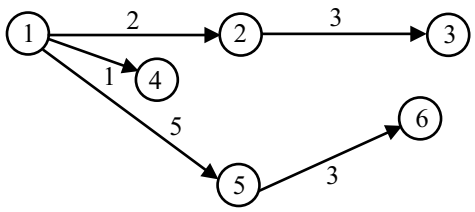


Figure 8.24 *Shortest path tree from node 1 to other nodes*

2. Bellman-Ford Algorithm

The **Bellman-Ford algorithm** computes single-source shortest paths in a weighted graph (where some of the edge weights may be negative). Dijkstra's algorithm accomplishes the same problem with a lower running time, but requires edge weights to be non-negative. Thus, Bellman-Ford is usually used only when there are negative edge weights.

Bellman-Ford runs in O(VE) time, where V and E are the number of vertices and edges.

Applications in routing

A distributed variant of Bellman-Ford algorithm is used in the Routing Information Protocol (RIP). The algorithm is distributed because it involves a number of nodes (routers) within an Autonomous system, a collection of IP networks typically owned by an ISP. It consists of the following steps:

1. Each node calculates the distances between itself and all other nodes within the AS and stores this information as a table.
2. Each node sends its table to all neighbouring nodes.
3. When a node receives distance tables from its neighbours, it calculates the shortest routes to all other nodes and updates its own table to reflect any changes.

The main disadvantages of Bellman-Ford algorithm in this setting are:

- Does not scale well.
- Changes in network topology are not reflected quickly since updates are spread node-by-node.
- Counting to infinity.

Example 1. Find the shortest path cost for destination vertex 6 of the graph in Fig. 8.23 using Bellman-Ford algorithm.

Bellman-Ford algorithm is to find both the minimum cost from each node to node 6 (the destination) and next node along the shortest path. Let us label each node i by (n, D_i) where n is the next node along the current shortest path and D_i is the current minimum cost from node i to the destination.

$D_i = \min (C_{ij} + D_j), \ \forall j \neq i$

Initially, $D_i = \infty, \ \forall i \neq d$ hence D_1, D_2, D_4 is ∞ and $D_3 = 6, D_5 = 3$

$D_d = 0$ (i.e. $D_6 = 0$)

In first iteration $D_1 = \min(C_{12} + D_2, C_{13}+D_3, C_{14}+D_4, C_{15} + D_5)$, where j range 2 to 5

$D_2 = \min(C_{21}+D_1, C_{23} +D_3, C_{24}+D_4, C_{25}+D_5)$, and j = 1,3,4,5

$D_3 = \min(C_{31} + D_1, C_{32}+D_2, C_{34}+D_4, C_{35} +D_5)$, where j ranges 1,2,4,5

$D_4 = \min(C_{41}+D_1, C_{42} +D_2, C_{43}+D_3, C_{45}+D_5)$, and j = 1,2,3,5

$D_5 = \min(C_{51}+D_1, C_{52} +D_2, C_{53}+D_3, C_{54}+D_4)$, and j = 1,2,3,4

If the next node is not defined, we set n to –1.

Iteration	*1 (n, D_1)*	*2 (n, D_2)*	*3 (n, D_3)*	*4 (n, D_4)*	*5 (n, D_5)*
Initial	–1, ∞	–1, ∞	6, 6	–1, ∞	6, 3
1	5, 8	3, 9	5, 5	5, 11	6, 3
2	5, 8	3, 8	5, 5	1, 9	6, 3
3	5, 8	3, 8	5, 5	1, 9	6, 3

Figure 8.25 *Shortest path computation using Bellman-Ford algorithm*

From the Fig. 8.25 draw the shortest path from each node to the destination node. The next node from node 1 is 5, from node 2 to node 3, from node 3 to node 5, node 4 to 1 and from node 5 to node 6.

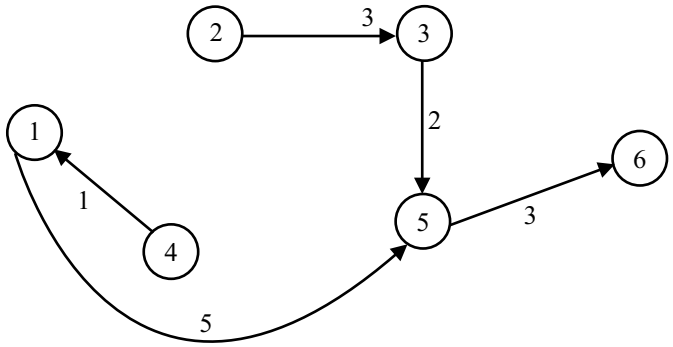


Figure 8.26 *Shortest path tree to node 6*

3. Floyd-Warshall Algorithm

The Floyd-Warshall algorithm takes as input an adjacency matrix representation of a weighted, directed graph (V, E). The weight of a path between two vertices is the sum of the weights of the edges along that path. The edges *E* of the graph may have negative weights, but the graph must not have any negative weight cycles. The algorithm computes, for each pair of vertices, the minimum weight among all paths between the two vertices. The running time complexity is $O(|V|^3)$.

The algorithm is based on the following observation: Assuming the vertices of a directed graph *G* are V = {1, 2, 3, 4, ..., *n*}, consider a subset {1, 2, 3, ..., *k*}. For any pair of vertices *i*, *j* in *V*, consider all paths from *i* to *j* whose intermediate vertices are all taken from {1, 2, ..., *k*}, and *p* is a minimum weight path from among them. The algorithm exploits a relationship between path *p* and shortest paths from *i* to *j* with all intermediate vertices in the set {1, 2, ..., *k*}. The relationship depends on whether or not *k* is an intermediate vertex of path *p*.

Applications and Generalizations

The Floyd-Warshall algorithm can be used to solve the following problems, among others:

- Shortest paths in directed graphs (Floyd's algorithm). For this to work, the weights of all edges are set to the some positive number. That number is usually chosen to be one, so that the weight of a path coincides with the number of edges along the path.
- Transitive closure of directed graphs (Warshall's algorithm). In Warshall's original formulation of the algorithm, the graph is unweighted and represented by a Boolean adjacency matrix. Then the addition operation is replaced by logical conjunction (AND) and the minimum operation by logical disjunction (OR).

- Finding a regular expression denoting the regular language accepted by a finite automation (Kleene's algorithm).
- Inversion of real matrices (Gauss-Jordan algorithm).
- Optimal routing: In this application one is interested in finding the path with the maximum flow between two vertices. This means that, rather than taking minima as in the pseudocode above, one instead takes maxima. The edge weights represent fixed constraints on flow. Path weights represent bottlenecks; so the addition operation above is replaced by the minimum operation.
- Testing whether an undirected graph is bipartite.

Warshal's algorithm to compute transitive closure can be modified to compute shortest path among all pairs of vertices of a weighted or cost graph as below.

```
for(i = 0 ; i< n; i++)
        for( j = 0; j<n; j++)
                C^k[i][j] = min (C^k–1[i][j], (C^k–1[i][k] + C^k–1[k][j]);
```

(Replace + by , and . by + and use a minimum cost function in transitive closure algorithm).

Similarly the Warshal's algorithm to compute shortest path is obtained just few modification in transitive closure implementation.

The following 'C' function computes the shortest path cost matrix C.

```
void  Warshal(int A[n][n], int C[n][n])
{
int i, j, k;
for (i = 0; i <n; i++)
     for (j = 0; j <n; j++)
          if (A[i][k] = = 0)
               C[i][j] = 32676;
          else
               C[i][j] = A[i][j];
for (k = 0; k <n; k++)
for (i = 0; i <n; i++)
     for (j = 0; j <n; j++)
          {/* Cij^k = min( Cij^k–1 ,Cik^k–1 +Ckj^k–1);*/
          if (C[i] [j] < = C[i] [k] + C[k] [j])
           C[i] [j] = C[i] [j];
      else
          C[i] [j] = C[i] [k] + C [k] [j];
          }
} /* end Warshal */
```

Example 1. Compute the shortest path among all pairs of vertices for the graph below in Fig. 8.27 using Warshal's algorithm.

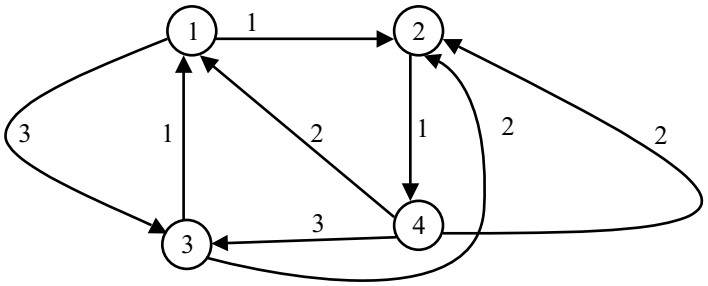


Figure 8.27

The initial cost matrix is as follows:

	1	2	3	4
1	0	1	3	0
2	0	0	0	1
3	1	2	0	0
4	2	2	3	0

Adjacency matrix A

	1	2	3	4
1	∞	1	3	∞
2	∞	∞	∞	1
3	1	2	∞	∞
4	2	2	3	∞

Cost matrix C^0

The cost matrix C^1, C^2, C^3 is computed as follows:

for k = 1 and i = 1, j = 1,2,3,4

$Cij^K = \min(C_{ij}^{k-1}, C_{ik}^{k-1} + C_{kj}^{k-1})$

$C^1_{11} = \min(C^0_{11}, C^0_{11} + C^0_{11}) = \min(\infty, \infty + \infty) = \infty$

$C^1_{12} = \min(C^0_{12}, C^0_{11} + C^0_{12}) = \min(1, \infty + 1) = 1$

$C^1_{13} = \min(C^0_{13}, C^0_{11} + C^0_{13}) = \min(3, \infty + 3) = 3$

$C^1_{14} = \min(C^0_{14}, C^0_{11} + C^0_{14}) = \min(\infty, \infty + \infty) = \infty$

k = 1 i = 2, j = 1,2,3,4

$C^1_{21} = \min(C^0_{21}, C^0_{21} + C^0_{11}) = \min(\infty, \infty + \infty) = \infty$

$C^1_{22} = \min(C^0_{22}, C^0_{21} + C^0_{12}) = \min(\infty, \infty + 1) = \infty$

$C^1_{23} = \min(C^0_{23}, C^0_{21} + C^0_{13}) = \min(\infty, \infty + 3) = \infty$

$C^1_{24} = \min(C^0_{24}, C^0_{21} + C^0_{14}) = \min(1, \infty + \infty) = 1$

k = 1 i = 3, j = 1,2,3,4

$C^1_{31} = \min(C^0_{31}, C^0_{31} + C^0_{11}) = \min(1, 1 + \infty) = 1$

$C^1_{32} = \min(C^0_{32}, C^0_{31} + C^0_{12}) = \min(2, 1 + 1) = 2$

$\mathbf{C^1_{33} = \min(C^0_{33}, C^0_{31} + C^0_{13}) = \min(\infty, 1 + 3) = 4}$

$C^1_{34} = \min(C^0_{34}, C^0_{31} + C^0_{14}) = \min(\infty, 1 + \infty) = \infty$

k = 1 i = 4, j = 1,2,3,4

$C^1_{41} = \min(C^0_{41}, C^0_{41} + C^0_{11}) = \min(2, 2 + \infty) = 2$

$C^1_{42} = \min(C^0_{42}, C^0_{41} + C^0_{12}) = \min(2, 2 + 1) = 2$

$C^1_{43} = \min(C^0_{43}, C^0_{41} + C^0_{13}) = \min(3, 2 + 3) = 3$

$C^1_{44} = \min(C^0_{44}, C^0_{41} + C^0_{14}) = \min(\infty, 2 + \infty) = \infty$

After first iteration cost matrix C^1 is as follows:

	1	2	3	4
1	∞	1	3	∞
2	∞	∞	∞	1
3	1	2	**4**	∞
4	2	2	3	∞

cost matrix C^1

In second iteration compute cost matrix C^2 from C^1

for k = 2 i = 1, j = 1,2,3,4

$C^2_{11} = \min(C^1_{11}, C^1_{12} + C^1_{21}) = \min(\infty, 1 + \infty) = \infty$

$C^2_{12} = \min(C^1_{12}, C^1_{12} + C^1_{22}) = \min(1, 1 + \infty) = 1$

$C^2_{13} = \min(C^1_{13}, C^1_{12} + C^1_{23}) = \min(3, 1 + \infty) = 3$

$\mathbf{C^2_{14} = \min(C^1_{14}, C^1_{21} + C^1_{24}) = \min(\infty, 1 + 1) = 2}$

k = 2 i = 2, j = 1,2,3,4

$C^2_{21} = \min(C^1_{21}, C^1_{22} + C^1_{21}) = \min(\infty, \infty + \infty) = \infty$

$C^2_{22} = \min(C^1_{22}, C^1_{22} + C^1_{22}) = \min(\infty, \infty + \infty) = \infty$

$C^2_{23} = \min(C^1_{23}, C^1_{22} + C^1_{23}) = \min(\infty, \infty + \infty) = \infty$

$C^2_{24} = \min(C^1_{24}, C^1_{22} + C^1_{24}) = \min(1, \infty + 1) = 1$

k = 2 i = 3, j = 1,2,3,4

$C^2_{31} = \min(C^1_{31}, C^1_{32} + C^1_{21}) = \min(1, 2 + \infty) = 1$

$C^2_{32} = \min(C^1_{32}, C^1_{32} + C^1_{22}) = \min(2, 2 + \infty) = 2$

$C^2_{33} = \min(C^1_{33}, C^1_{32} + C^1_{23}) = \min(4, 2 + \infty) = 4$

$\mathbf{C^2_{34} = \min(C^1_{34}, C^1_{32} + C^1_{24}) = \min(\infty, 2 + 1) = 3}$

k = 2 i = 4, j = 1,2,3,4

$C^2_{41} = \min(C^1_{41}, C^1_{42} + C^1_{21}) = \min(2, 2 + \infty) = 2$

$C^2_{42} = \min(C^1_{42}, C^1_{42} + C^1_{22}) = \min(2, 2 + \infty) = 2$

$C^2_{43} = \min(C^1_{43}, C^1_{42} + C^1_{23}) = \min(3, 2 + \infty) = 3$

$C^2_{44} = \min(C^1_{44}, C^1_{42} + C^1_{24}) = \min(\infty, 2 + 1) = 3$

After second iteration cost matrix C^2 is as follows:

	1	2	3	4
1	∞	1	3	**2**
2	∞	∞	∞	1
3	1	2	4	**3**
4	2	2	3	**3**

cost matrix C^2

In third iteration compute cost matrix C^3 from C^2

for k = 3 i = 1, j = 1,2,3,4

$C^3_{11} = \min(C^2_{11}, C^2_{13} + C^2_{31}) = \min(\infty, 3 + 1) = 4$

$C^3_{12} = \min(C^2_{12}, C^2_{13} + C^2_{32}) = \min(1, 3 + 2) = 1$

$C^3_{13} = \min(C^2_{13}, C^2_{13} + C^2_{33}) = \min(3, 3 + \infty) = 3$

$C^3_{14} = \min(C^2_{14}, C^2_{13} + C^2_{34}) = \min(2, 3 + 3) = 2$

for k = 3 i = 2, j = 1,2,3,4

$C^3_{21} = \min(C^2_{21}, C^2_{23} + C^2_{31}) = \min(\infty, \infty + 1) = \infty$

$C^3_{22} = \min(C^2_{22}, C^2_{23} + C^2_{32}) = \min(\infty, \infty + 2) = \infty$

$C^3_{23} = \min(C^2_{23}, C^2_{23} + C^2_{33}) = \min(\infty, \infty + 4) = \infty$

$C^3_{24} = \min(C^2_{24}, C^2_{23} + C^2_{34}) = \min(1, \infty + 3) = 1$

for k = 3 i = 3, j = 1,2,3,4

$C^3_{31} = \min(C^2_{31}, C^2_{33} + C^2_{31}) = \min(1, 4 + 1) = 1$

$C^3_{32} = \min(C^2_{32}, C^2_{33} + C^2_{32}) = \min(2, 4 + 2) = 2$

$C^3_{33} = \min(C^2_{33}, C^2_{33} + C^2_{33}) = \min(4, 4 + 4) = 4$

$C^3_{34} = \min(C^2_{34}, C^2_{33} + C^2_{34}) = \min(3, 4 + 3) = 3$

for k = 4 i = 4, j = 1,2,3,4

$C^3_{41} = \min(C^2_{41}, C^2_{43} + C^2_{31}) = \min(2, 3 + 1) = 2$

$C^3_{42} = \min(C^2_{42}, C^2_{43} + C^2_{32}) = \min(2, 3 + 2) = 2$

$C^3_{43} = \min(C^2_{43}, C^2_{43} + C^2_{33}) = \min(3, 3 + 4) = 3$

C^3_{44} = min (C^2_{44}, C^2_{43}+ C^2_{34}) = min(3 , 3 + 3) = 3

After third iteration cost matrix C^3 is as follows:

	1	2	3	4
1	4	1	3	2
2	∞	∞	∞	1
3	1	2	4	3
4	2	2	3	3

cost matrix C^3

In fourth iteration compute cost matrix C^4 from C^3

for k = 4 i = 1, j = 1,2,3,4

C^4_{11} = min (C^3_{11}, C^3_{14}+ C^2_{41}) = min(4, 2 + 2) = 4

C^4_{12} = min (C^3_{12}, C^3_{14}+ C^3_{42}) = min(1, 2 + 2) = 1

C^4_{13} = min(C^3_{13}, C^3_{14}+ C^3_{43}) = min(3, 2 + 3) = 3

C^4_{14} = min(C^3_{14}, C^3_{14}+ C^3_{44}) = min(2, 3 + 3) = 2

k = 4 i = 2, j = 1,2,3,4

C^4_{21} = min (C^3_{21}, C^3_{24}+ C^3_{41}) = min(∞, 1 + 2) = 3

C^4_{22} =min(C^3_{22}, C^3_{24}+ C^3_{42}) = min(∞, 1 + 2) = 3

C^4_{23} = min(C^3_{23}, C^3_{24}+ C^3_{43}) = min(∞, 1 + 3) = 4

C^4_{24} = min (C^3_{24}, C^3_{24}+ C^3_{44}) = min (1, 1 + 3) = 1

k = 4 i = 3, j = 1,2,3,4

C^4_{31} = min(C^3_{31}, C^3_{34}+ C^3_{41}) = min(1, 4 + 1) = 1

C^4_{32} = min(C^3_{32}, C^3_{34}+ C^3_{42}) = min(2, 4 + 2) = 2

C^4_{33} = min(C^3_{33}, C^3_{34}+ C^3_{43}) = min(4, 4 + 4) = 4

C^4_{34} = min(C^4_{34}, C^4_{34}+ C^3_{44}) = min(3, 4+ 3) = 3

k = 4 i = 4, j = 1,2,3,4

C^4_{41} = min(C^3_{41}, C^3_{44}+ C^3_{41}) = min(2, 3 + 1) = 2

C^4_{42} = min(C^3_{42}, C^3_{44}+ C^3_{32}) = min(2, 3 + 2) = 2

C^4_{43} = min(C^3_{43}, C^3_{44}+ C^3_{43}) = min(3, 3 + 3) = 3

C^4_{44} = min (C^3_{44}, C^3_{44}+ C^3_{44}) = min(3, 3 + 3) = 3

After fourth iteration cost matrix C^4 is as follows.

	1	2	3	4
1	4	1	3	2
2	3	3	4	1
3	1	2	4	3
4	2	2	3	3

cost matrix C^4

4. Johnson's Algorithm

Johnson's algorithm is a way to solve the all–pairs shortest path problem in a sparse, weighted, directed graph.

First, it adds a new node with zero weight edge from it to all other nodes, and runs the Bellman–Ford algorithm to check for negative weight cycles and finds $h(v)$, the least weight of a path from the new node to node v. Next it reweighs the edges using the nodes' $h(v)$ values. Finally for each node, it runs Dijkstra's algorithm and stores the computed least weight to other nodes, reweighs using the nodes' $h(v)$ values, as the final weight. The time complexity is $O(V^2 \log V + VE)$.

8.9 APPLICATIONS OF GRAPH

Graph theory has a very wide range of applications in engineering, in physical, social and biological sciences, in linguistics and in numerous other areas.

A graph can be used to represent almost any physical situation involving discrete objects and a relationship among them. The following are few examples.

(*i*) **PERT and related techniques:** A directed graph is a natural way of describing, representing, and analyzing complex projects, which consist of man interrelated activities. The project might be, for example, the design and construction of power dam or the design and erection of an apartment building.

There are a number of management techniques such as PERT (Program Evaluation and Review Techniques) and CPM (Critical Path Method) which employ a graph as the structure on which analysis is based.

A PERT graph is a finite digraph with no parallel edges or cycles, in which there is exactly one source (i.e., a node whose indegree is 0) and one sink (i.e., a node whose outdegree is 0). Each edge in the graph is assigned a weight (time) value. The directed edges are meant to represent activities, with the directed edge joining nodes, which represent the start time and the finish time of the activity. The weight value of each edge is taken to be the time it takes to complete the activity.

One such graph is represented in Fig. 8.28. Each node is called an event and it represents a point in time. In particular, node v_1 denotes the start of the entire project (its source) and v_6 its completion (its sink). The number represents the number of days required to do that particular activity.

The most common solution of system analysis and its activities is represented by PERT chart to compute critical path. A critical path is a path from the source node to the sink node such that if an activity on the path is delayed by an amount t, then entire project scheduled is delayed by t.

We can compute an algorithm to determine critical path for PERT graph in Fig. 8.28.

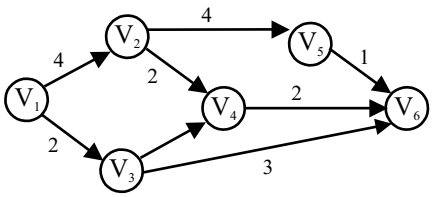


Figure 8.28 *A PERT graph*

(*ii*) **Electrical network problems:** Electrical network analysis and synthesis are mainly the study of network topology. The topology of a network is studied by means of its graph. In the graph of electrical network vertices represent the junctions, and branches (which consist of electrical elements) are represented by edges.

(*iii*) **Topological sort:** A topological sort of a directed graph without cycles, also known as directed acycle graph or DAG, G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (x, y), then x appears before in the ordering.

A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

Directed acyclic graphs are used in many applications to indicate precedence among events such as code optimization techniques of compiler.

To find topological sort of DAG, the general structure of the algorithm is as follows:

1. Perform DFS on G for each vertex.
2. As each vertex is finished, insert at front of a linked list.
3. Print the elements of linked list in order.

(*iv*) **Minimum spanning tree (MST):** A spanning tree for a graph, G = (V, E), is a subgraph of G that is a tree and contains all the vertices of G. In a weighted graph, the weight of a graph is the sum of the weights of the edges of the graph. An MST for a weighted/cost graph is a spanning tree with minimum cost.

There are main applications where minimum spanning tree is needed:

- To find cheapest way to connect a set of terminals, where terminals may represent cities, electrical, electronic components of a circuit, computers, or premises by using roads, wires or wireless, or telephone lines. The solution to this is an MST, which has an edge for each possible connection weighed by the cost of that connection.
- The routing problems to find path among the system over Internet, also need MST.
- The most common solution for MST is Prim's and Kruskal's algorithms.

(*v*) **Shortest path:** A path from source vertices v to w is shortest path if there are shortest paths from v to u and u to w with lower costs. The shortest paths are not necessarily unique.

The most common shortest path problem is in traveling salesman. Another example is airline, which gives the cities having direct flights and the total time of flights and air fare. The shortest path finds which route provides minimum air fare.

Various shortest path algorithms discussed in this chapter are Dijkstra, Bellman-Ford, Warshall and Johnson.

Chapter 9

Sorting Algorithms and Their Analysis

The concept of an ordered set of elements is one that has considerable impact on many applications. For example, the process of finding a telephone number in a telephone directory. This process, called a search, is simplified as the names in the directory are listed in alphabetical order.

Another example is the result of your examinations published in sorted order by roll number, list of student names and order of rank in your examination.

9.1 INTERNAL AND EXTERNAL SORTING

A sort can be classified as being internal if the records (or elements) that it is sorting are in main memory, or external if some of the records that it is sorting are in auxiliary or secondary storage. Example of external sorting is a telephone directory. Here a record is a collection of fields or items and a key is usually a field of the record. A file is a collection of records. The file can be sorted on one or more keys. In the example of the telephone directory, the key upon which the file is sorted is the name field of the record. Each record also contains fields for an address and telephone number.

Internal sorting methods are to be used when a file to be sorted is small enough so that the entire sort can be carried out in main memory.

Internal sorting methods:

- Selection sort
- Bubble sort
- Insertion sort
- Quick sort
- Merge sort
- Heap sort
- Radix sort

The time required to read and write is not considered to be significant in evaluating the performance of internal sorting methods. The three important efficiency criteria are:

- use of storage space
- use of computer time
- programming effort.

External sorting methods are employed to sort records of file which are too large to fit in the main memory of the computer. These methods involve as much as external processing (e.g. Disk I/O) as compared to processing in the CPU.

The sorting requires the involvement of external devices such as magnetic tape, magnetic disk due to the following reasons:

1. The cost of accessing an element is much higher than any computational costs.
2. Depending upon the external device, the method of access has different restrictions.

External storage devices can be categorized into two types based on access method. These are sequential access devices (e.g. magnetic tapes) and random access devices (e.g. disks).

External sorting depends to a large extent on system considerations like the type of device and the number of such devices that can be used at a time.

The lists of algorithms are K-way merge, selection tree and polyphone merging. The general method of external sorting is the merge sort.

9.2 SORTING PROBLEM

Sorting refers to the operation of arranging data in some given order, such as increasing or decreasing, with numerical data or alphabetically, with character data.

Sorting is applied to a file of records or set of elements input at the time of execution of the program. Each record in a file can contain many fields, but there may be some field(s) whose values uniquely determine the records in the file, called key of the file, sorting the file with respect to a particular key K. In some of the applications, we want to sort the data on several keys.

Example 1. Suppose an array contains 8 integer elements as follows:

```
int x[8] ={77, 31, 44, 40, 61, 55, 32, 34};
```

After sorting in ascending order, array x must appear in memory as follows:

```
x[ ] ={31, 32, 34, 40, 44, 55, 61, 77};
```

Example 2. Suppose an array name contains 8 student names as follows:

```
char *name [8] = {"sanjay", "rajesh", "poonam", "ash", "mona", "dina", "mpati", "mtech"}
```

After sorting, array name must appear in memory as follows:

{"ash", "dina", "mona", "mpati", "mtech", "poonam", "rajesh", "sanjay"}

Example 3. Suppose the personnel file of a company contains the following data on each of its employees:

Name, designation, and salary

Suppose a structure employee contains 8 records of the file as follows:

Name	*Designation*	*Salary*
Kapil	GM	25000
Anil	AM	20000
S.Ram	GM	25000
Dayal	DM	20200
Rakesh	GM	25000
Amit	AM	20000
Kapil	AM	20000
Anil	GM	25000

Sorting the file with respect to name key will yield a different order of the records than sorting the file with respect to the designation or salary. Here the name of the employee is same so another key is required to sort the file.

After sorting the data with respect to name key the result is as follows:

Name	*Designation*	*Salary*
Amit	GM	25000
Anil	AM	20000
Anil	GM	25000
Dayal	DM	20200
Kapil	**GM**	**25000**
Kapil	**AM**	**20000**
Rakesh	GM	25000
S.Ram	GM	25000

Sorting with Multiple Keys

Sorting the file with respect to name and then designation yields the following result:

Name	*Designation*	*Salary*
Amit	GM	25000
Anil	AM	20000
Anil	GM	25000
Dayal	DM	20200
Kapil	**AM**	**20000**
Kapil	**GM**	**25000**
Rakesh	GM	25000
S.Ram	GM	25000

Sorting file with respect to designation and name yield the different result as follows:

Name	*Designation*	*Salary*
Anil	AM	20000
Kapil	AM	20000
Dayal	DM	20200
Amit	GM	25000
Anil	GM	25000
Kapil	GM	25000
Rakesh	GM	25000
S.Ram	GM	25000

Similarly, file can be sorted on several fields at the same time but the result will be different only if the first sorted field/key has duplicate values and next field than also has duplicate values and so on.

Complexity of Sorting Algorithms

The complexity of a sorting algorithm measures the execution time as a function of the number n of elements to be sorted. Usually each algorithm made the following operations:

(*i*) Comparisons—which test whether x[i] >x[j]

(*ii*) Interchanges—which switch the contents of x[i] and x[j]

(*iii*) Assignments—which set x[i]=x[j] or t=x[i].

Normally, the complexity function measures only the number of comparisons, since the number of other operations is at most a constant factor of the number of comparisons.

9.2.1 Bubble Sort

This is the most widely known sorting by comparison method. The basic idea underlying the bubble sort is to pass through the list of elements (e.g. file) sequentially several times. Each pass consists of comparing each element in the array (or file) with its successor (x[i] with x[i+1]) and interchanging the two elements if they are not in proper order.

Consider the following array x containing 8 integer elements in memory.

```
int x[8] ={25, 57, 48, 37, 12, 92, 86, 33};
```

The following comparisons are made on the first iteration:

x[0] with x[1]	(25 with 57)	No interchange
x[1] with x[2]	(57 with 48)	Interchange
x[2] with x[3]	(57 with 37)	Interchange
x[3] with x[4]	(57 with 12)	Interchange
x[4] with x[5]	(57 with 92)	No interchange
x[5] with x[6]	(92 with 86)	Interchange
x[6] with x[7]	(92 with 33)	Interchange

Thus, after the first iteration, the array x is in the order 25 48 37 12 57 86 33 92

Notice that after this first iteration, the largest element (in this case 92) is in its proper position within the array x. In general, x[n–i] will be in its proper position after iteration i.

This method is called the bubble sort because each number slowly "bubbles" up to its proper position.

The following comparisons are made on the second iteration:

x[0] with x[1]	(25 with 48)	No interchange
x[1] with x[2]	(48 with 37)	Interchange
x[2] with x[3]	(48 with 12)	Interchange
x[3] with x[4]	(48 with 57)	No interchange
x[4] with x[5]	(57 with 86)	No interchange
x[5] with x[6]	(86 with 33)	Interchange
x[6] with x[7]	(86 with 92)	No interchange

Notice that 86 has now found its way to the second highest position.

The order after 2nd iteration is 25 37 12 48 57 33 86 92.

Since each iteration places a new element into its proper position, a set of n elements requires no more than n–1 iterations.

The complete set of iterations is the following:

Iteration 0 (Original set)	25	57	48	37	12	92	86	33
Iteration 1	25	48	37	12	57	86	33	**92**
Iteration 2	25	37	12	48	57	33	**86**	**92**
Iteration 3	25	12	37	48	33	**57**	**86**	**92**
Iteration 4	12	25	37	33	**48**	**57**	**86**	**92**
Iteration 5	12	25	33	**37**	**48**	**57**	**86**	**92**
Iteration 6	12	25	**33**	**37**	**48**	**57**	**86**	**92**
Iteration 7	12	**25**	**33**	**37**	**48**	**57**	**86**	**92**

What is the Efficiency of the Bubble Sort?

There are n–1 iterations and n–1 comparisons on each iteration. Thus, the total number of comparisons is (n–1) * (n–1) = n^2–2n+1, which is O(n^2). Of course, the number of interchanges depends on the original order of the file. However, the number of interchanges cannot be greater than the number of comparisons.

The algorithm of bubble sort in the form of 'C' function is given below:

Algorithm Bubble (x[],n)

```
/* x is array of n integers, which is to be sort */
{
for (i = 0; i<n–1;i++)
        {
            for (j = 0; j < n–1; j++)
                if(x[j]>x[j+1])
                    {
                        t = x[j];
                        x[j] = x[j+1];
                        x[j+1] = t;
                    }       /* end of if and for loop*/
            }      /* end of outer for loop */
        }     /*end of bubble */
```

However, there are some obvious improvements to it. First, since all the elements in positions greater than or equal to n–i are already in proper position after iteration i, they need not be considered in successive iterations. Thus in the first iteration n–1 comparisons are made, in second iteration n–2 comparisons are made and so on. The number of comparisons on iteration i is n–i.

Second, eliminate the unnecessary iterations. By keeping a record of whether or not interchanges are made in a given pass it can be determined whether any further iterations are necessary.

Thus, if there are k iterations, the total number of comparisons is (n–1)+….(n–k) which equals (2kn–k^2–k)/2.

It can be shown that the average number of iterations, k, is O(n), so that the entire formula is still O(n^2).

In our example 5 iterations are required. So total number of comparisons are

(2*5*8–25–5)/2 = (80–30) = 50/2 = 25

But, the most of time in the program's execution goes in number of interchanges rather than number of comparisons.

The number of interchanges in our example are:

In iteration 1	5 interchanges
In iteration 2	3 interchanges
In iteration 3	2 interchanges
In iteration 4	1 interchange
In iteration 5	0 interchanges

Total interchanges are 11.

The modified algorithm of bubble sort in the form of 'C' function, which also returns the number of comparisons required for sorting is given below:

```
Algorithm MBubble (x[],n)
/* x is array of n integers, which is to be sort */
{
for (i = 0; i<n–1;i++)
        {
                for (j = 0; j<n–i–1; j++)
                        {
                nc++;           /* nc counts the number of comparisons required */
                if(x[j]>x[j+1])
                        {
                                t = x[j];
                                x[j] = x[j+1];
                                x[j+1] = t;
                        }       /* end of if  */
                }/* end of for loop*/
         }      /* end of outer for loop */
return nc;
}     /*end of Mbubble */
```

A complete program for modified bubble sort is given below:


```
/* Program mbubble.c sort the list of n integers element into ascending
order with minimum number of iteration and comparisons */
   #include<conio.h>
   #include<stdio.h>
   #define MAXSIZE 10
int mbubble(int x[],int n)
{
int i,j,t, nc = 0;
for(i = 0;i<n;i++)
        {
                for(j = 0;j<n–i–1;j++)
                {
                        nc++;
                        if(x[j]>x[j+1])
                        {
```

```
                    t = x[j];
                    x[j] = x[j+1];
                    x[j+1] = t;
                } /* end of if */
            } /* end of inner loop */
        } /* end of outer loop */
return nc;
} /* end of mbubble */

void main()
{
int x[MAXSIZE],n,i, nc;

clrscr();

printf("Enter the number of elements: [1–10]");
scanf("%d",&n);

printf("\nEnter the elements: \n");
for(i = 0;i<n;i++)
scanf("%d",&x[i]);

nc = mbubble(x,n);

printf("\nThe sorted output required %d comparisons\n", nc);
for(i = 0;i<n;i++)
printf("\n%d",x[i]);

getch();
}
```

Example 1. Consider the following sets of elements are in memory, which is sorted in ascending order. Now, find the number of comparisons to sort the set in ascending order.

25 29 32 36 38 40 44 54

The following comparisons are made on the first iteration:

x[0] with x[1]	(25 with 29)	No interchange
x[1] with x[2]	(29 with 32)	No interchange
x[2] with x[3]	(32 with 36)	No interchange
x[3] with x[4]	(36 with 38)	No interchange
x[4] with x[5]	(38 with 40)	No interchange
x[5] with x[6]	(40 with 44)	No interchange
x[6] with x[7]	(44 with 54)	No interchange

As no interchange is required in first iteration, so no more iteration is required. The number of comparisons in first iteration required is n –1 i.e., 8 – 7 = 7.

Example 2. Find the number of comparisons to sort the above set of elements in descending order. The only change in the function is use "<" operator instead of ">" operator for comparisons. The following comparisons are made on the first iteration:

x[0] with x[1]	(25 with 29)	Interchange
x[1] with x[2]	(25 with 32)	Interchange
x[2] with x[3]	(25 with 36)	Interchange
x[3] with x[4]	(25 with 38)	Interchange
x[4] with x[5]	(25 with 40)	Interchange
x[5] with x[6]	(25 with 44)	Interchange
x[6] with x[7]	(44 with 54)	Interchange

After first iteration the set is as follows:

29 32 36 38 40 44 54 25

The complete set of iterations are given below:

Original set	25	29	32	36	38	40	44	54
Iteration 1	29	32	36	38	40	44	54	**25**
Iteration 2	32	36	38	40	44	54	**29**	**25**
Iteration 3	36	38	40	44	54	**32**	**29**	**25**
Iteration 4	38	40	44	54	**36**	**32**	**29**	**25**
Iteration 5	40	44	54	**38**	**36**	**32**	**29**	**25**
Iteration 6	44	54	**40**	**38**	**36**	**32**	**29**	**25**
Iteration 7	54	**44**	**40**	**38**	**36**	**32**	**29**	**25**

Thus the number of comparisons

$$n-1 + n-2 + n-3 + n-4 + n-5 + n-6 + 1$$
$$= 7 + 6 + 5 + 4 + 3 + 2 + 1$$
$$= \frac{7 * 8}{2} = 28$$

and number of interchanges are also same as comparisons.

9.2.2 Selection Sort

A selection sort is one in which successive elements are selected in order and placed into their proper sorted positions.

The selection sort implements the descending priority queue as an unordered array.

The selection sort consists entirely of a selection phase in which the largest of the remaining elements is repeatedly placed in its proper position, i, at the end of the array.

The large element is interchanged with the element x[i].

The first iterations make n–1 comparisons and second iteration makes n–2, and so on. Therefore, there is a total of n–1 + n–2 + n–3 + …….. +1 = n* (n–1)/2 comparisons, which is $O(n^2)$, although it is faster than the bubble sort.

The number of interchanges is always n–1. The function for selection is given below:

```
void selectsort (int x[], int n)
{
/* x is an array of integers consists of n elements, consider pos, and large are two temporary
variables. */
```

```
        int i, j, large;
        for (i = n–1; i>0; i– –){
                large = x[0];
                pos = 0;
                for (j = 1; j< = i; j++)
                        if (x[j] > large){
                        large = x[j];
                        pos = j;
                } /* end of if and inner for loop*/
         x[pos] = x[i];
        x[i] = large;
        } /*end of outer for loop */
} /*end of selection sort*/
```

There is no improvement if the input set of elements is either completely sorted or unsorted.

A complete 'C' program to implement selection sort is given below:


```
/* Program selsort.c is selection sort, sorts the list of integer elements */
   #include<stdio.h>
   #define MAXSIZE 10

void selection(int x[], int n)
{
/* x is an array of integers consists of n elements, consider pos,
and large are two temporary variables. */
int i, j, large,pos,nc = 0;
for (i = n–1; i>0; i– –){
        large = x[0];
        pos = 0;
        for (j = 1; j< = i; j++)
        {
                nc++;
                        if (x[j] > large){
                                large = x[j];
                                pos = j;
                        } /* end of if*/
        } /* end of for loop */
        x[pos] = x[i];
        x[i] = large;
        } /*end of outer for loop */
        printf("\n Number of comparisons required = %d",nc);
   } /*end of selection sort*/

   void main( )
   {
```

```
    int x[MAXSIZE],n, i;
    printf("\n Enter number of elements[1–10]");
    scanf("%d",&n);
    printf("\n Unsorted list is as follows\n");
    for (i = 0; i < n; i++)
    {     printf("\n Enter element %d ",i+1);
          scanf("%d",&x[i]);
    }
    selection(x, n);
    printf("\n Sorted list is as follows\n");
    for (i = 0; i < n; i++)
    printf(" %d ", x[i]);

}
```

Consider a set of elements in array x as follows:

25 57 48 37 12 92 86 33

The following comparisons are made in first iteration:

x[1] with large	(57 with 25)	large=57, pos=1
x[2] with large	(48 with 57)	No change
x[3] with large	(37 with 57)	No change
x[4] with large	(12 with 57)	No change
x[5] with large	(92 with 57)	large=92, pos=5
x[6] with large	(86 with 92)	No change
x[7] with large	(33 with 92)	No change

and the assignment statement x[pos] = x[i] i.e., x[5] = x[7] and x[7] = 92 is executed.

Elements after first iteration in the array x is as follows:

25 57 48 37 12 33 86 92

The following comparisons are made in second iteration:

x[1] with large	(57 with 25)	large = 57, pos = 1
x[2] with large	(48 with 57)	No change
x[3] with large	(37 with 57)	No change
x[4] with large	(12 with 57)	No change
x[5] with large	(33 with 57)	No change
x[6] with large	(86 with 57)	large = 86, pos = 6

and assignment statement x[pos] = x[i] i.e., x[6] = 86

After second iteration the array x is as follows:

25 57 48 37 12 33 86 92

The complete iterations are given below:

Original set	25	57	48	37	12	92	86	33
Iteration 1	25	57	48	37	12	33	86	**92**
Iteration 2	25	57	48	37	12	33	**86**	**92**
Iteration 3	25	33	48	37	12	**57**	**86**	**92**
Iteration 4	25	33	12	37	**48**	**57**	**86**	**92**
Iteration 5	25	33	12	**37**	**48**	**57**	**86**	**92**
Iteration 6	25	12	**33**	**37**	**48**	**57**	**86**	**92**
Iteration 7	12	**25**	**33**	**37**	**48**	**57**	**86**	**92**

Thus, total number of comparisons required is n*(n–1)/2 = 8*(8–7)/2 = 28

The number of comparisons in selection sort is independent of the original order of an element. But the number of interchanges and assignments does depend on the original order of the element on the original order of the elements in the array x.

Example 1. Consider the following set of elements order in memory, which is sorted in ascending order. Now, find the number of comparisons to sort the set in ascending order:

25 29 32 36 38 40 44 54

The following comparisons are made on the first iteration:

Assigned large = x[0], pos = 0.

x[1] with large	(29 with 25)	large = 29, pos = 1
x[2] with large	(32 with 29)	large = 32, pos = 2
x[3] with large	(36 with 32)	large = 36, pos = 3
x[4] with large	(38 with 36)	large = 38, pos = 4
x[5] with large	(40 with 38)	large = 40, pos = 5
x[6] with large	(44 with 40)	large = 44, pos = 6
x[7] with large	(54 with 44)	large = 54, pos = 7

x[7] = x[7] and x[7] = 54

After first iteration the order is as follows:

25 29 32 36 38 40 44 54

The complete set of iterations is as follows:

Iteration 1	25	29	32	36	38	40	44	**54**
Iteration 2	25	29	32	36	38	40	**44**	**54**
Iteration 3	25	29	32	36	38	**40**	**44**	**54**
Iteration 4	25	29	32	36	**38**	**40**	**44**	**54**
Iteration 5	25	29	32	**36**	**38**	**40**	**44**	**54**
Iteration 6	25	29	**32**	**36**	**38**	**40**	**44**	**54**
Iteration 7	25	**29**	**32**	**36**	**38**	**40**	**44**	**54**

Thus, the number of comparisons

= n–1 + n–2 + ……. +1 = 7+6+5+4+3+2+1 = (8*7)/2 = 28

Example 2. Find the number of comparisons for above set to sort in descending order. The following changes are required in selection sort function:

(1) Place the first smallest element value in the last index of the array x.

(2) Place the second smallest element value in the second last index of the array x and repeat the process until index 1.

In function change large variable with small and in If condition use less than (<) operator instead of greater than (>).

In the first iteration, the following comparisons are made:

small = x[0];	pos = 0;	
x[1] with small	(29 with 25)	No assignment
x[2] with small	(32 with 25)	No assignment
x[3] with small	(36 with 25)	No assignment
x[4] with small	(38 with 25)	No assignment
x[5] with small	(40 with 25)	No assignment
x[6] with small	(44 with 25)	No assignment
x[7] with small	(54 with 25)	No assignment

Thus, the number of comparisons requires in first iteration is 7 but no assignment is required. Interchange the small value with last index value of the array

x[0] = x[i] i.e. x[0] = 54
x[7] = small i.e. x[7] = 25

After first iteration the array x is as follows:

54 29 32 36 38 40 44 25

In the subsequent iteration the following conditions arise:

Original set	25	29	32	36	38	40	44	54
Iteration 1	54	29	32	36	38	40	44	**25**
Iteration 2	54	44	32	36	38	40	**29**	**25**
Iteration 3	54	44	40	36	38	**32**	**29**	**25**
Iteration 4	54	44	40	38	**36**	**32**	**29**	**25**
Iteration 5	54	44	40	**38**	**36**	**32**	**29**	**25**
Iteration 6	54	44	**40**	**38**	**36**	**32**	**29**	**25**
Iteration 7	54	**44**	**40**	**38**	**36**	**32**	**29**	**25**

Again total number of comparisons are (8*7)/2 = 28. Only the changes are in number of assignment and interchanges.

Modified Selection Sort

The selection sort procedure can also be modified to reduce the number of comparisons. The modification is done using a flag variable. The outer for loop can be broken if the inner for loop if statement is true for all iterations of the loop. That means now the remaining elements to be sorted is in sorting order and we don't need more iterations.

The function for modified selection sort is just introduce a flag variable and it checks the number of times the large variable value is changed in the inner for loop. If it is equal to the number of iterations in the inner loop then outer for loop is broken otherwise some assignment statement is written as in selection sort.

The function and complete 'C' program for modified selection sort are given below:

```
void mselection(int x[], int n)
{
/* x is an array of integers consists of n elements, consider pos, and large are two temporary variables. */
```

```
int i, j, large,pos,nc = 0,flag = 1;
for (i = n–1; i>0; i– –){
        large = x[0];
        pos = 0;
        for (j = 1; j< = i; j++)
        {
        nc++;
                if (x[j] > large){
                        large = x[j];
                        pos = j;
                        flag++;
                } /* end of if*/
        } /* end of for loop */
        if (flag ! = j)
        {
                x[pos] = x[i];
                x[i] = large;
                flag = 1;
        }
        else
                break;
        } /*end of outer for loop */
        printf("\n Number of comparisons for modified selection sort is
                = %d",nc);
} /*end of mselection sort*/
```

A complete 'C' program for modified selection sort is given below:

```
/* Program mselsort.c is modified selection sort, sorts the list of integer
elements with minimum number of comparisons*/
   #include<stdio.h>
   #define MAXSIZE 10

void mselection(int x[ ], int n)
{
/* x is an array of integers consists of n elements, consider pos,
and large are two temporary variables. */
int i, j, large,pos,nc = 0,flag = 1;
for (i = n–1; i>0; i– –){
        large = x[0];
        pos = 0;
        for (j = 1; j< = i; j++)
        {
                nc++;
```

```
                    if (x[j] > large){
                        large = x[j];
                        pos = j;
                        flag++;
                    } /* end of if */
        } /* end of for loop */
        if (flag ! = j)
        {
                    x[pos] = x[i];
                    x[i] = large;
                    flag = 1;
        }
        else
                break;
        } /*end of outer for loop */
        printf("\n Number of comparisons for modified selection sort is = %d",nc);
} /*end of selection sort*/

void main()
{
        int x[MAXSIZE],n, i;
        printf("\n Enter number of elements[1–10]");
        scanf("%d",&n);
        printf("\n Unsorted list is as follows\n");

        for (i = 0; i < n; i++)
        {       printf("\n Enter element %d ",i+1);
                scanf("%d",&x[i]);
        }
        mselection(x, n);
        printf("\n Sorted list is as follows\n");

        for (i = 0; i < n; i++)
                printf(" %d ", x[i]);

}
```

9.2.3 Insertion Sort

Insertion sort is one that sorts a set of elements by inserting elements into an existing sorted set. The simple method for insertion sort is given below:

1. x[0] by itself is sorted as only element
2. x[1] is inserted before or after x[0] so that x[0], x[1] is sorted
3. x[2] is inserted into its proper place with x[0], x[1], so that x[0], x[1], x[2] is sorted and so on up to x[n–1] is inserted into proper place so that x[0], x[1], x[2]…..x[n–1] is sorted.

The function for insertion sort is given below:

```
void inssort (int x[], int n)
{
        int i, j, temp;
/*initially x[0] itself is sorted set of one element.  After each iteration of the following loop, the
elements x[0] through x[i] are in order */
        for (i = 1; i<n; i++)   /*insert x[i] into the sorted set */
                {
                temp = x[i];
                /*move down 1 position all element greater than temp */
                for (j = i–1; j> = 0 && temp <x[j]; j– –)
                        x[j+1] = x[j];
                /* insert temp at proper position */
                        x[j+1] = temp;
                } /* end for loop */
} /* end of inssort function*/
```

If the initial file is sorted, only one comparison is made on each iteration, so that the sort is O(n).

If the set of elements is initially sorted to the reverse order, the sort is $O(n^2)$, since the total number of comparison is

$$n–1 + n–2 + \ldots\ldots\ldots 3+2+1 = n*(n–1)/2 = O(n^2)$$

However, the insertion sort is still usually better than the bubble sort. The space requirements for the sort consists of only a temporary variable, temp.

The speed of the sort is improved using a binary search to find the proper position for x[i] in the sorted set. Another improvement can be made using array link of pointers. This reduces the time requirement for insertion but not the time required for searching for the proper position. The space or memory requirements are also increased because of extra link array. This sort is called list insertion sort and may be viewed as a general selection sort in which the priority queue is implemented by an ordered list.

Both selection sort and insertion sort are more efficient than bubble sort. Selection sort requires fewer assignments than insertion sort but more comparisons.

Sort the following set of elements:

25 29 38 36 32 54 44 40

Initially x[0] = 25 is sorted set of 1 element

Iteration 1 is as follows:

temp = x[1];	(temp = 29)
temp with x[0]	(29 with 25) no move down

after iteration 1 the set of elements is as follows:

25 29 38 36 32 54 44 40

Iteration 2 as follows:

temp = x[2]	(temp = 38)
temp with x[1]	(38 with 29) no move down

after iteration 2 the set of elements is as follows:

25 29 38 36 32 54 44 40

Iteration 3 as follows:

temp = x[3]	(temp = 36)	
temp with x[2]	(36 with 38)	move down
temp with x[1]	(36 with 29)	no move down

After iteration 3 set is as follows:

25 29 36 38 32 54 44 40

Iteration 4 as follows:

temp = x[4]	(temp = 32)	
temp with x[3]	(32 with 38)	move down
temp with x[2]	(32 with 36)	move down
temp with x[1]	(36 with 29)	no move down

After iteration 4 th set of elements is as follows:

25 29 32 36 38 54 44 40

In iteration 5 the comparisons are as follows:

temp = x[5]	(temp = 54)	
temp with x[4]	(54 with 38)	no move down

The set is as follows:

25 29 32 36 38 54 44 40

Iteration 6 is as follows:

temp = x[6]	(temp = 44)	
temp with x[5]	(44 with 54)	move down
temp with x[4]	(44 with 38)	no move down

The set is as follows:

25 29 32 36 38 44 54 40

Iteration 7 is as follows:

temp = x[7]	(temp = 40)	
temp with x[6]	(40 with 54)	move down
temp with x[5]	(40 with 44)	move down
temp with x[4]	(40 with 38)	no move down

The set is as follows:

25 29 32 36 38 40 44 54

The total number of comparisons is sum of comparisons in each iteration, which is number of move down as follows:

= 1+2+2+3+1+2+3 = 14

The complete 'C' program for insertion sort is given below:

SORT/INSSORT.C

```
/* Program inssort.c is insertion sort, sorts the list of integer elements */
  #include<stdio.h>
  #define MAXSIZE 10
void inssort (int x[], int n)
{
      int i, j, temp, nc = 0;
/*initially x[0] itself is sorted set of one element.  After each iteration
of the following loop, the elements x[0] through x[i] are in order */
```

```
        for (i = 1; i<n; i++)   /*insert x[i] into the sorted set */
                {
                temp = x[i];
                /*move down 1 position all elements greater than temp */
                for (j = i–1; j> = 0 && temp <x[j]; j– –)
                {
                        x[j+1] = x[j];
                        nc++;
                }
                /* insert temp at proper position */
                        x[j+1] = temp;
                } /* end of for loop */
        printf("\n Number of comparisons required = %d", nc);
} /* end of inssort function*/

void main()
{
        int x[MAXSIZE],n, i;
        printf("\n Enter number of elements[1–10]");
        scanf("%d",&n);
        printf("\n Unsorted list is as follows\n");

        for (i = 0; i < n; i++)
        {       printf("\n Enter element %d",i+1);
                scanf("%d",&x[i]);
        }
        inssort(x, n);
        printf("\n Sorted list is as follows\n");

        for (i = 0; i < n; i++)
                printf(" %d ", x[i]);

}
```

9.2.4 Shell Sort

Most significant improvement over simple insertion sort, is named after its developer Donald Shell.

This method sorts separate subset of original set of elements. This subset contains every k^{th} element of the original set. The value of k is called increment. After the first k subsets are sorted, a new smaller value of k is chosen and the file is again partitioned into a new set of subsets of elements. Each of these again sorted. Eventually, the value of k is set to 1 so that the only one subset consisting of the entire file is sorted.

A decreasing sequence of increments is fixed at the start of sorting. The last value in this sequence must be 1. For example, if the array x contains the following elements:

25 29 38 36 32 54 44 40

and the sequence (4,2,1) is chosen, the following subsets are sorted on each iteration.

First iteration (increment = 4)

(x[0], x[4])
(x[1], x[5])
(x[2], x[6])
(x[3], x[7])

Second iteration (increment = 2)

(x[0], x[2], x[4], x[6])
(x[1], x[3], x[5], x[7])

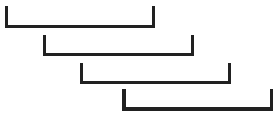
Third iteration (increment = 1)

(x[0], x[1], x[2], x[3], (x[4], x[5], x[6], x[7])

The shell sort is illustrated below:

Original set 25 29 38 36 32 54 44 40

Pass 1 increment = 4 25 29 38 36 32 54 44 40



Pass 2 increment = 2 25 29 32 36 38 40 44 54

Pass 3 increment = 1 25 29 32 36 38 40 44 54

Sorted file 25 29 32 36 38 40 44 54

The function for shell sort requires an array which contains increments value inc, and number of increments nofinc.

```
void shelsort (int x[], int n, int inc[], int nofinc)
{
        int i, j, k, span, temp;
        static int nc = 0;
        for (i = 0;i< nofinc; i++){
        /* span is the size of the increment */
        span = inc[i];
        for (j = span;j<n; j++){
        /* insert element x[j] into its proper within its subsets */
              nc++;
              temp = x[j];
              for (k = j–span; k> = 0 && temp< x[k]; k = k–span)
                    x[k+span] = x[k];
          x[k+span] = temp;
```

```
        }/*end of j inner for loop */
      }/* end of i outer for loop */
      printf("\n Number of comparisons required = %d", nc);
}/* end of shelsort */
```

The idea behind the shell sort is a simple one. Since the first increment used by the shell sort is large, the individual subsets are quite small, so that the simple insertion sorts on those subsets are fairly fast. Each sort of a subset causes the entire file to be moved nearly sorted. The efficiency analysis of shell sort approximated by $O(n(\log_2 n)^2)$ of an appropriate sequence increments is used.

The number of comparisons required for above set is the sum of all comparisons for each subset sorting in each increment. The computation is as follows:

		Number of comparisons
Iteration 1	subset 1 = {25, 32}	1
Increment = 4	subset 2 = {29, 54}	1
	subset 3 = {38, 44}	1
	subset 4 = {36, 54}	1

After first iteration, set of elements in array x is as follows:

x[]= 25 29 38 36 32 54 44 40

Iteration 2

Increment = 2 subset 1 = {25, 38, 32, 44}

Now apply insertion sort on these elements and compute the number of comparisons.

Initially element 25 is sorted

Next element 38 required no move down, so only one comparison required.

Next element 32 required 2 comparisons to insert in subset, so the output is {25, 32, 38, 44}

Next element 44 required only 1 comparison to insert in last.

Thus total comparisons for this set is 0+1+2+1 = 4 and sorted subset is placed in original set.

Subset 2 = {29, 36, 54, 40}

Now apply insertion sort on these elements and compute the number of comparisons.

Initially element 29 is sorted

Next element 36 required no move down, so only one comparison required.

Next element 54 required no move down, so only one comparison required.

Next element 40 required 2 comparisons.

Thus, total comparisons for this set is 0+1+1+2 = 4.

For this iteration, number of comparisons is 8 and sorted subset is placed in original set.

After iteration 2 the array x is as follows:

x[] = 25 29 32 36 38 40 44 54

Iteration 3

Increment = 1 Only one set is there, apply simple insertion sort comparisons. As the set is already sorted, so each iteration of insertion required only one comparison, so total number of comparisons to sort this whole set is n–1 = 8–1 =7.

Total number of comparisons required for shell sort is 4+8+7 = 19

Example 1. Repeat the shell sort procedure for the above set with sequence (5, 3, 1) of increments.

The following subsets are sorted on iteration and compute the number of comparisons.

Iteration 1	subset 1 = (x[0], x[5])	{25, 54}
Increment = 5	subset 2 = (x[1], x[6])	(29, 44)
	subset 3 = (x[2], x[7])	(38, 40)
	subset 4 = (x[3])	(36)
	subset 5 = (x[4])	(32)

Only subsets 1, 2, and 3 require only one comparison to sort them, thus total number of comparisons for this iteration is 3

After first iteration the array x is as follows:

25	29	38	36	32	54	44	40
x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]

Iteration 2	subset 1 = (x[1], x[3], x[6])	{25, 36, 44}
Increment = 3	subset 2 = (x[1], x[4], x[7])	(29, 32, 40)
	subset 3 = (x[2], x[5])	(38, 54)

The subset 1 requires 2 comparisons, subset 2 requires 2 comparisons and subset 3 requires one comparison to sort the elements. Total of 2+2+1 = 5 comparisons for this iteration and array x is as follows:

25	29	38	36	32	54	44	40

Iteration 3	Only subset is the original set of
Increment = 1	set 1 = {x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7]}

25	29	38	36	32	54	44	40

Initially, element x[0] i.e., 25 is primarily sorted.

Insertion of element 29 required 1 comparison.

Insertion of element 38 required 1 comparison.

Insertion of element 36 required 2 comparisons.

Insertion of element 32 required 3 comparisons.

Insertion of element 54 required 1 comparison.

Insertion of element 44 required 2 comparisons.

Insertion of element 40 required 3 comparisons.

Thus total of 1+1+2+3+1+2+3 = 13 comparisons required.

The array x after this iteration is as follows:

25	29	32	36	38	40	44	54

Total number of comparisons for shell sort is 3+5+13 = 21

The number of comparisons required is more than previous one.

The actual time requirements for a specific sort depend on the number of elements in the increment array inc and on their actual values.

In general, the shell sort is recommended for average set of several hundred elements.

A complete 'C' program for shell sort is given below:

```
/* Program shelsort.c is shell sort, sorts the list of integer elements */
  #include<stdio.h>
  #define MAXSIZE 10
```

SORT/SHELSORT.C

```
void shelsort (int x[], int n, int inc[], int nofinc)
{
        int i, j, k, span, temp;
        static int nc = 0;
        for (i = 0;i< nofinc; i++){
        /* span is the size of the increment */
        span = inc[i];
        for (j = span;j<n; j++){
        /* insert element x[j] into its proper within its subsets */
        nc++;
        temp = x[j];
        for (k = j–span; k> = 0 && temp< x[k]; k = k–span)
                x[k+span] = x[k];
  x[k+span] = temp;
                }/*end of j inner for loop */
        }/* end of i outer for loop */
        printf("\n Number of comparisons required = %d", nc);
}/* end of shell sort */
void main()
{
        int x[MAXSIZE],n, i, nofinc,inc[MAXSIZE/2];
        printf("\n Enter number of elements[1–10]");
        scanf("%d",&n);
        printf("\n Unsorted list is as follows\n");
        for (i = 0; i < n; i++)
        {       printf("\n Enter element %d",i+1);
                scanf("%d",&x[i]);
        }
        printf("\n Enter number of increments for shell sort[1–%d]",n);
        scanf("%d",&nofinc);
        printf("\n Enter the incremetns value [%d –1] \n",n);
        for (i = 0; i < nofinc; i++)
        {       printf("\n Enter Increment %d",i+1);
                scanf("%d",&inc[i]);
        }
        shelsort(x, n,inc,nofinc);
        printf("\n Sorted list is as follows\n");
        for (i = 0; i < n; i++)
                printf(" %d ", x[i]);
}
```

9.2.5 Address Calculation Sort

The sorting by address calculation sort is sometimes called sorting by hashing. In this method a hash function is applied on sorted field (i.e., key).

In applying a hashing function to the sorting process, a particular kind of hashing function is required. The hashing function h with the property

$$x_1 < x_2 \text{ implies that } h(x_1) \leq h(x_2)$$

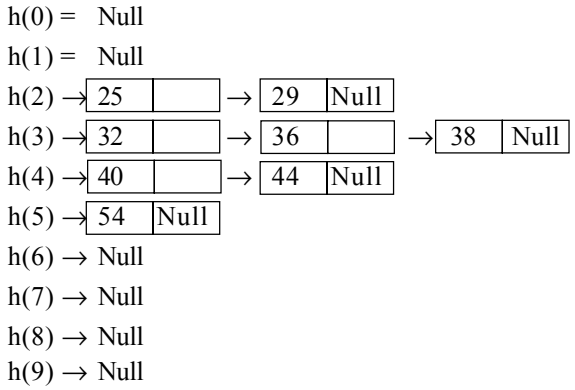
Where x_1 and x_2 is element value.

A function that exhibits this property is called a non-decreasing, or order preserving hashing function. When such a function is used to hash a particular value (i.e., key) into a particular number location to which some previous elements have already been hashed (that is, a collision occurs), then the new value is placed in the set of colliding elements, so as to preserve the order of the values.

Example 1. Consider the set of elements

25 29 38 36 32 54 44 40

Now, apply a hash function h, which creates ten subsets, one for each of the ten possible first digits. The function h(i) consists of the elements whose first digit (i.e., Most Significant Digit [MSD]) is i. Each of the subset is maintained as a sorted linked list of the original array elements. After applying hashing function to each of the element of array x elements are placed in the proper linked list as follows:



These sorted linked lists are concatenated to produce the sorted result,

25 29 32 36 38 40 44 54

The implementation of address calculation sort is required to maintain linked list with above hashing function. The implementation totally depends upon the hashing function.

9.2.6 Radix Sort

This sort is based on the values of the actual digits in the positional representations of the numbers being sorted. The radix sorting does not require any comparisons. It uses queue data structure to implement sorting process. It needs ten queues according to ten digits in the decimal numbers and concatenation of queues from given the sorted output.

Suppose that we perform the method on set of elements for each digit, beginning with the least significant digit and ending with most significant digit. Place the element into one of ten queues, depending on the value of the digit currently being processed. Then concatenate each queue to original set starting with the queue

of numbers with a 0 digit and ending with the queue of numbers with digit 9. Perform these actions for each digit, with the least significant and ending with most significant, then resultant set becomes sorted. This sorting method is called the **radix sort**.

Example 1. Consider a set of elements stored in queue x = {25, 29, 38, 36, 32, 54, 44, 40}, apply radix sort.

x = 25 29 38 36 32 54 44 40

Insert the elements in queues based on least significant digit. For example, element 25 is inserted in queue 5 as its least significant digit is 5 and so on as given below:

	Front	Rear
Q[0]	4**0**	
Q[1]		
Q[2]	3**2**	
Q[3]		
Q[4]	5**4**	4**4**
Q[5]	2**5**	
Q[6]	3**6**	
Q[7]		
Q[8]	3**8**	
Q[9]	2**9**	

After first pass queue x is:

40 32 54 44 25 36 38 29

In next pass insert elements in queues based on most significant digit:

	Front	Rear
Q[0]		
Q[1]		
Q[2]	**2**5	**2**9
Q[3]	**3**2	**3**6 **3**8
Q[4]	**4**0	**4**4
Q[5]	**5**4	
Q[6]		
Q[7]		
Q[8]		
Q[9]		

After second pass the set is sorted: 25 29 32 36 38 40 44 54

The algorithm for radix sort is given below:

Algorithm radixsort(x)

Step 1 : for k = 1 to number of digits do

Step 2 : for i = 0 to n – 1 do

temp = x[i];

d = k^{th} digit of temp;

place temp at rear of Q[d];

/* end of for loop at step2 */

Step 3 : for j = 0 to 9 do
Delete elements of Q[j] and place in next sequential position of queue x;
/* end of for loop at step1 */

Step 4 : end radixsort

The time complexity for radix sorting method clearly depends on the number of digits (m) and number of elements (i.e., n) in the set. Thus, the sort required O(m *n) time. Sometimes number of digits m may be $\log_2 n$, then the sort required $O(n\log_2 n)$ time.

The sort does require memory space to store pointers to the fronts and rears of the queues in addition to an extra field (i.e., next) in each record to be used as pointer in the linked lists. If the number of digits is large it is sometimes more efficient to sort the set by first applying the radix sort to the most significant digits and then using straight insertion sort on the rearranged set.

Example 2. Consider one example where elements consist of different number of digits in their values.

x = {121, 345, 2, 4343, 12, 34, 5, 1231, 231, 3212}

Insert the elements in queues based on least significant digit. For example element 121 is inserted in queue 1 as its least significant digit is 1, element 345 is inserted in queue 5 as its least significant digit is 5 and so on as given below.

	Front		Rear
Q[0]			
Q[1]	12**1**	123**1**	23**1**
Q[2]	**2**	1**2**	321**2**
Q[3]	434**3**		
Q[4]	3**4**		
Q[5]	34**5**	**5**	
Q[6]			
Q[7]			
Q[8]			
Q[9]			

After first pass queue x is:

x = 121 1231 231 2 12 3212 4343 34 345 5

In second pass place the elements whose 2nd digit present in queue x into Q[] according to 2nd digit of the element. Remaining elements in queue x = {2 5 }

	Front		Rear
Q[0]			
Q[1]	**1**2	32**1**2	
Q[2]	1**2**1		
Q[3]	12**3**1	2**3**1	**3**4
Q[4]	43**4**3	3**4**5	
Q[5]			
Q[6]			
Q[7]			
Q[8]			
Q[9]			

Now concatenate the queues from Q[0] to Q[9] in the queue x, after second pass queue x is:

x = 2 5 12 3212 121 1231 231 34 4343 345

In third pass place the elements whose 3rd digit present in queue x into Q[] according to 3rd digit of the element. Remaining elements in queue x = {2 5 12 34}

	Front		Rear
Q[0]			
Q[1]	**1**21		
Q[2]	3**2**12	1**2**31	**2**31
Q[3]	4**3**43	**3**45	
Q[4]			
Q[5]			
Q[6]			
Q[7]			
Q[8]			
Q[9]			

Now concatenate the queues from Q[0] to Q[9] in the queue x, after third pass queue x is:

x = 2 5 12 34 121 3212 1231 231 4343 345

In fourth pass place the elements whose 4th digit present in queue x into Q[] according to 4th digit of the element. Remaining elements in queue x = {2 5 12 34 121 231 345}

	Front	Rear
Q[0]		
Q[1]	**1**231	
Q[2]		
Q[3]	**3**212	
Q[4]	**4**343	
Q[5]		
Q[6]		
Q[7]		
Q[8]		
Q[9]		

Now concatenate the queues from Q[0] to Q[9] in the queue x, after fourth pass queue x is:

x = 2 5 12 34 121 231 345 1231 3212 4343

A complete 'C' program for radix sort is given below.

SORT/RADIX.C

```
/*A radix.c is sort program for radix or bucket sorting. */
  #include<stdio.h>
  #include<conio.h>
void main()
{
        int a[20],n,lrg,j,i,k,p;
        int digcnt,divsr,buktcnt[10],bukt[10][10],r;
        clrscr();
        printf("**********************************\n");
```

```
printf("RADIX/BUCKET SORT\n");
printf("**********************************\n");
printf("How many elements(Maximum:20)\n");
scanf("%d",&n);
printf("Enter the elements\n");
for(i = 0;i<n;i++)
scanf("%d",&a[i]);
i = 0;
lrg = a[i];
/*find the largest element in array*/
while(i<n)
{
        if(a[i]>lrg)
                lrg = a[i];
                i++;
}
/*count the no of digit in largest no*/
digcnt = 0;
while(lrg>0)
{
        digcnt = digcnt+1;
        lrg = lrg/10;
}
i = 1;
divsr = 1;
while(i< = digcnt)
{
        j = 0;
        while(j<10)
        {
                buktcnt[j] = 0;
                j++;
        }
j = 0;
        while(j<n)
        {
                r = (a[j]/divsr)%10;
                bukt[r][buktcnt[r]] = a[j];
                buktcnt[r]++;
                j++;
        }
```

```
            /*collect all element in order*/
            j = 0;
            p = 0;
            while(j<10)
            {
                  k = 0;
                  while(k<buktcnt[j])
                  {
                        a[p] = bukt[j][k];
                        p++;
                        k++;
                  }
            j++;
            }
      i++;
      divsr = divsr*10;
      }
      clrscr();
      printf("\nSORTED ELEMENTS.....\n");
      for(i = 0;i<n;i++)
            printf("\n%d",a[i]);
getch();
}
```

OUTPUT:

```
**********************************
        RADIX\BUCKET SORT
**********************************
how many elements in array
6
enter the element of array
215 319 75 124 166 8

SORTED ELEMENTS.....................
8          75          124          166          215          319
```

9.2.7 Merge Sort

Merging is the process of combining two or more sorted lists into a third sorted list. Suppose array x is a list with n1 elements and array y with n2 elements and they are merged into a third array z with n1 + n2 elements. This procedure is called simple merge.

Consider two sorted lists of array x and array y consist of following elements:

Initially i = 0, j = 0, k = 0

```
x[] :  29    32    38    40
y[] :  25    36    44    54    86    90
```

Initially array z is empty. Here i, j, and k are indices for arrays x, y and z respectively.

	Comparison	Elements in array z	
x[0] with y[0]	(29 with 25)	25	Assigned y[0] into z[0] & increment j index and k index.
x[0] with y[1]	(29 with 36)	25 29	Assigned x[0] into z[1] and increment i, and k index.
x[1] with y[1]	(32 with 36)	25 29 32	Assigned x[1] into z[2] and increment i and k
x[2] with y[1]	(38 with 36)	25 29 32 36	Assigned y[1] into z[3] and increment j and k
x[2] with y[2]	(38 with 44)	25 29 32 36 38	Assigned x[2] into z[4] and increment i and k
x[3] with y[2]	(40 with 44)	25 29 32 36 38 40	Assigned x[3] into z[5] and increment k and now array x is completely merged in array 2

Copy elements of array y from index 2 to upper bound of it i.e., 5 into array z

Result is 25 29 32 36 38 40 44 54 86 90

Number of comparisons required for this set is 6.

The function for merging two lists is given below:

```
void mergelist (int x[], int y[], int z[], int n1, int n2, int n3)
{
        int i = 0, j = 0, k = 0;/* indices for array x, y, z */
        if (n1+n2 ! = n3) {
                printf("array bound incompatible \n");
                exit(1);
                }
        for (k = 0; i< = n1–1 && j < = n2–1; k++)
        if (x[i] < y[j])
                z[k] = x[i++];
        else
                z[k] = y[j++];
        while (i< = n1–1)
                z[k++] = x[i++];
        while (j< = n2–1)
                z[k++] = y[j++];
} /* end mergelist */
```

This merge can be further enhanced to sort the n lists of single element to get the sorted list. This is called **two-way merge sort**. This is one of the "divide and conquer" classes of algorithm. This algorithm sorts the n elements array x using an auxiliary array y. The basic idea into this is to divide the set into a number of subsets and merge them to get a single sorted set.

A complete 'C program for merge list is given below:

SORT/MERGLIST.C

```
/* Program merglist.c merges the two sorted lists into a single list
 of integer elements */
   #include<stdio.h>
   #define MAXSIZE 20
void mergelist (int x[], int y[], int z[], int n1, int n2, int n3)
{
        int i = 0, j = 0, k = 0;/* indices for array x, y, z */
        if (n1+n2 ! = n3) {
                printf("array bound incompatible \n");
                exit(1);
                }
        for (k = 0; i< = n1–1 && j < = n2–1; k++)
        if (x[i] < y[j])
                z[k] = x[i++];
        else
                z[k] = y[j++];
        while (i< = n1–1)
                z[k++] = x[i++];
        while (j< = n2–1)
                z[k++] = y[j++];
} /* end mergelist */
void main()
{
        int x[MAXSIZE], y[MAXSIZE], z[MAXSIZE], n1,n2, n3, i;
        printf("\n Enter number of elements in sorted list1");
        scanf("%d",&n1);
        printf("\n Sorted list is as follows\n");
        for (i = 0; i < n1; i++)
        {       printf("\n Enter element %d",i+1);
                scanf("%d",&x[i]);
        }
        printf("\n Enter number of elements in sorted list2");
        scanf("%d",&n2);
        printf("\n Sorted list is as follows\n");
        for (i = 0; i < n2; i++)
        {       printf("\n Enter element %d",i+1);
                scanf("%d",&y[i]);
        }
        n3 = n1 + n2;
        mergelist(x, y,z, n1,n2,n3);
```

```
        printf("\n Sorted list is as follows\n");
        for (i = 0; i < n3; i++)
                printf(" %d ", z[i]);

}
```

The **recursive implementation** of 2-way merge sort divides the set of elements into 2 subsets, x[0], x[n/2–1] and x[n/2] ….. x[n–1]. Each set is individually sorted and the resulting sequences are merged to produce a single sorted sequence of n elements.

The illustrative implementation of 2-way merge sort finally divides the input in n sets of size 1. These are merged to get n/2 sets of size 2. These n/2 sets are merged pairwise and so on till a single set of sorted elements is obtained.

We give here the recursive implementation of 2-way merge. The function for merge sort is given below.

```
/* Mergesort funtion divides the set size into half of size recursively then call merge function */
void mergesort(int low ,int high,int a[])
{
int mid;
if (low < high)
{
        mid = (low+high)/2;
        mergesort(low,mid,a);
        mergesort(mid+1,high,a);
        merge(low,mid,high,a);
        return;
}
} /* End of mergesort function */

/* Merge function merges the two sorted sets, one set is low to mid  and second set is mid+1 to high */
void merge(int low,int mid,int high,int a[])
{
        int h,b[MAXSIZE],i,j,k;
        h = low;i = low;j = mid+1;
        printf("\n Recursion calls :%d %d %d",low,mid,high);
        while((h< = mid) && (j< = high))
        {
                if (a[h]< = a[j])
                        { b[i] = a[h]; h = h+1; }
                else
                        { b[i] = a[j]; j = j+1; }
                i = i+1;
        }
        if (h>mid)
```

```
            for(k = j;k< = high;k++)
            { b[i] = a[k]; i = i+1; }
        else
            for(k = h;k< = mid;k++)
                { b[i] = a[k]; i = i+1; }
        for (k = low;k< = high;k++)
            a[k] = b[k];
return;
} /* End of merge function */
```

Example 1. Consider the array of eight integer elements x = {25 29 38 36 32 44 40 54}

The function mergesort begins by dividing x into two subsets of size four. The elements in x [0] to x[3] are then divided into two subsets of size two. They are further divided into size of one and now the merging begins.

Note that no actual movement of data occurred.

The mergesort function is explained easily with binary tree where two parameters of array x, lower bound low and and upper bound high, are used as in Figure 9.1.

Initial value of low is 0 and high is 7.

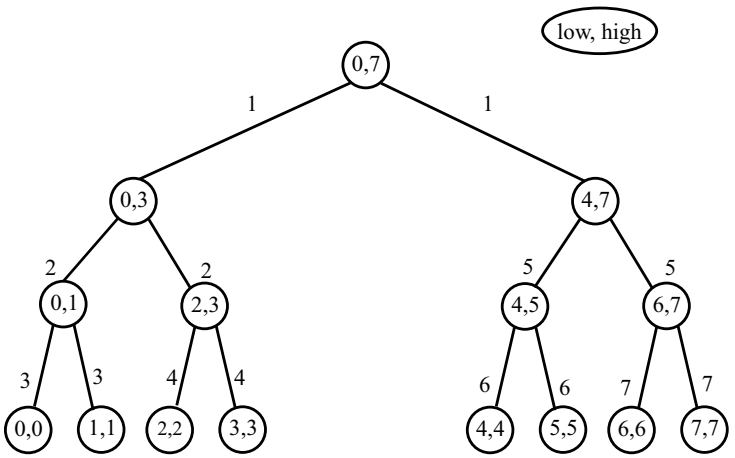


Figure 9.1 *The order of recursive division is numbered on tree edges.*

Division 1 mergesort(0, 7)	25 29 38 36 32 44 40 54
Division 2 mergesort(0, 3)	25 29 38 36
Division 3 Mergesort(0,1)	25 29
Call of Merge (0, 0, 1)	25 29
Division 4 mergesort(2,3)	38 36

Recursive calls of merge (2,2,3)	36 38
Recursive calls of merge (0,1,3)	25 29 36 38
Division 5 mergesort(4,7)	32 44 40 54
Division 6 mergesort(4,5)	32 44
Recursive calls of merge(4,4,5)	32 44
Division 7 mergesort(6,7)	40 54
Recursive calls of merge(6,6,7)	40 54
Recursive calls of merge(4,5,7)	32 44 40 54
Recursive calls of merge(0,3,7)	25 29 32 36 38 40 44 54

The merge function has 3 parameters for array x: lower bound, middle index, and upper bound. The tree is representing calls to function merge of mergesort is given below in Fig. 9.2

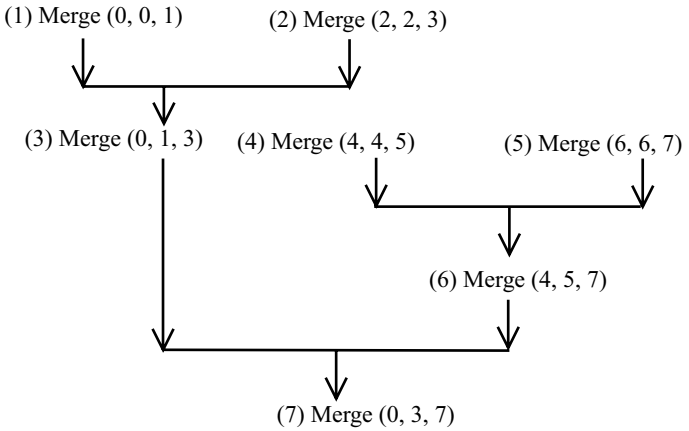


Figure 9.2 *Process of merge*

The complete 'C' program for merge sort is given below:

```
/* mergesor.c describes the sorting process using recursion, which
merges two sorted sets. The elements set is divided into
recursively half of the set size repeatedly and then merged. */
```

SORT/MERGESOR.C

```
#include<stdio.h>
#define MAXSIZE 10

/* Mergesort funtion divides the set size into half of size recursively
then calls merge function */
void mergesort(int low ,int high,int a[8])
{
void merge(int,int,int,int[]);
int mid;
if (low < high)
{
        mid = (low+high)/2;
        mergesort(low,mid,a);
        mergesort(mid+1,high,a);
        merge(low,mid,high,a);
        return;
}
} /* End of mergesort function */

/* Merge function merges the two sorted sets, one set is low to mid  and
second set is mid+1 to high */
void merge(int low,int mid,int high,int a[])
{
        int h,b[MAXSIZE],i,j,k;
        h = low;i = low;j = mid+1;
        printf("\n Recursion calls :%d %d %d",low,mid,high);
        while((h< = mid) && (j< = high))
        {
                if (a[h]< = a[j])
                        { b[i] = a[h]; h = h+1; }
                else
                        { b[i] = a[j]; j = j+1; }
                i = i+1;
        }
        if (h>mid)
                for(k = j;k< = high;k++)
                        { b[i] = a[k]; i = i+1; }
        else
                for(k = h;k< = mid;k++)
                        { b[i] = a[k]; i = i+1; }
        for (k = low;k< = high;k++)
                a[k] = b[k];
return;
```

```
} /* End of merge function */
void main()
{
int a[MAXSIZE],i,j,k,low=0,high,mid,n;
void mergesort(int,int,int[]);
printf("\nEnter the number of elements to be sorted \n");
scanf("%d",&n);
printf(" Enter unsorted list elements \n");
for(i = 0;i<n;i++)
   {
   printf("\n Enter element %d", i+1);
   scanf("%d",&a[i]);
   }
high = n–1;
mergesort(low,high,a);
printf("The merge output is ...\n");
for(i = 0;i<n;i++)
   printf("\t %d",a[i]);
} /* End of main function */
```

9.2.8 Quick Sort

This is the most widely used internal sorting algorithm. In its basic form, it was invented by C.A.R. Hoare. Its popularity lies in the ease of implementation, moderate use of resources and acceptable behaviour for a variety of sorting cases.

The basis of quick sort is "divide and conquer". The division into two subsets is made such that the sorted subsets do not need to be later merged.

The recursive implementation is as follows:

1. Choose the first element of array x as the pivot number.
2. Rearrange the set so that this element is in the proper position i.e. all preceding elements have lesser or equal value and all succeeding elements have a greater value than this element.
   a. x[0], x[1] ……. x[i–1] in the set 1
   b. x[i]
   c. x[i+1], x[i+2]……. x[n–1] in the set 2
3. Repeat steps 1 & 2 for subsets 1 and 2 till array x is sorted.

The step 2 of the above implementation is as follows.

(*i*) Choose x[m] the dividing element i.e., pivot element and m is lower bound index (in 'C' it is 0).

(*ii*) From the left end of the array (x[0]) onwards) scan till an element x[i] is found whose value is greater than x[m].

(*iii*) From right end of array (x[n–1] backwards) scan till an element x[j] is found whose value is less than x[m].

(*iv*) Swap x[i] and x[j]

(*v*) Continue steps (ii), (iii) & (iv) till the scan pointers i and j cross.

(*vi*) Now interchange the x[m] with x[j] at this stage, set is partitioned according to pivot element & set 1 has lesser or equal value elements and set 2 has higher value elements to the pivot element value.

The function for quick sort is given below.

Example 1. Consider the following set of elements in array x to be sorted in ascending order.

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
x[] =	25	29	36	32	38	44	40	54

The lower bound index is 0 and upper bound is 7. The step2 of above implementation is applied.

Initially, quick sort (x, 0, 7) is called, which calls a partition function to create two distinct sets.

				x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]	i	j
step2 (*i*)	pivot element												
	v = x[0]	x[] =		25	29	36	32	38	44	40	54	0	7
step2 (*ii*) and (*iii*)		x[] =		25	29	36	32	38	44	40	54		

i j

After first partition the distinct sets are

25 29 36 32 38 44 40 54

Now first set has only one element so no more partitioning. Apply the same procedure on set 2 elements for partitioning.

	Pivot element										i	j
step2	(*i*) v = x[1]	x[] =	25	29	36	32	38	44	40	54	1	7
step2	(*ii*) & (iii)	x[] =	25	29	36	32	38	44	40	54	1	1

i j

After partition set 2 into two subsets are

25 29 36 32 38 44 40 54

Now again partition subset 2 of set 2

	Pivot element										i	j
step2	(*i*) v = x[2]	x[] =	25	29	36	32	38	44	40	54	2	7
step2	(*ii*) & (*iii*)	x[] =	25	29	36	32	38	44	40	54	4	3

j i

No swapping needed. The element x[2] is interchanged with x[j] and partition result is

x[] = 25 29 36 32 38 44 40 54

step2 (*vi*) Again partition subset 2 required further partitioning.

	Pivot element										i	j
step2	(*i*) v = x[4]	x[] =	25	29	32	36	38	44	40	54	4	7
step2	(*ii*) & (*iii*)	x[] =	25	29	32	36	38	44	40	54	5	5

i j

Result of partition is

x[] = 25 29 32 36 38 44 40 54

Again subset 2 is partitioned.

										i	j
Pivot element											
step2 (*i*) v = x[5]	x[] =	25	29	32	36	38	44	40	54	5	7
step2 (*ii*) & (*iii*)	x[] =	25	29	32	36	38	44	40	54	7	6

step2 (*vi*) Interchange x[j] with x[5] the result is

x[] = 25 29 32 36 38 40 44 54

Now recursion stop as no more partition required and resultant set is sorted. The procedure is summarized in the table below:

Table : Behaviour of Quick Sort (Recursive Algorithm)

Partition Call	*x[0]*	*x[1]*	*x[2]*	*x[3]*	*x[4]*	*x[5]*	*x[6]*	*x[7]*
Initially	{**25**	29	36	32	38	44	40	54}
1	25	{**29**	36	32	38	44	40	54}
2	25	29	{**36**	32	38	44	40	54}
3	25	29	{32}	36	{**38**	44	40	54}
4	25	29	32	36	38	{**44**	40	54}
5	25	29	32	36	38	{40}	44	{54}
Result sorted	25	29	32	36	38	40	44	54

Bold face numbers are pivot numbers.

The recursive function for quick sort first divides elements into two sets according to its pivot element. It uses a quickr function, which calls a partition function for dividing the set according to pivot number.

The function quickr has three parameters: array a, lower bound and upper bound indexes of the array as follows.

```
/* function quickr call recursively array until lower bound is smaller than
upper bound */
void quickr(int x[],int p,int q,int n)
{
int j;

if(p<q)
{
        j = partition(x,p,q,n);
        quickr(x,p,j–1,n);
        quickr(x,j+1,q,n);
}
}/* end of quickr */
```

The partition function works as follows.

```
int partition(int x[],int m,int p,int n)
/*
x[] if the file to be sorted. m denotes the file's lower bound
and p denotes its upper bound.
*/
{
int v = x[m],i = m,j = p,temp;
static  pass;

while(i<j)
{
        while(x[i]< = v && i<p)
                i++;

        while(x[j]>v)
                j– –;

        if(i<j)
        {
                temp = x[i];
                x[i] = x[j];
                x[j] = temp;
        }
}

x[m] = x[j];
x[j] = v;

printf(" Completion of Pass %d ",++pass);
display(x,n,j);

return j;
}/* end of partition */
```

A complete 'C' program to sort set of elements in array is given below:


```
/* Program quick_re.c is recursive quicksort method to sort an array
of elements */
#include<conio.h>
#include<stdio.h>
#define MAXSIZE 20
void display(int x[],int n,int pivot)
{
int i;

for(i = 0;i<n;i++)
```

```
{
        if(i == pivot)
                printf("< %d >",x[i]);
        else
                printf("%d ",x[i]);

}
printf("\n\n");
}
int partition(int x[],int m,int p,int n)
/*
The file to be sorted is in array x [ ], m denotes the file's lower bound
and p denotes its upper bound.
*/
{
int v = x[m],i = m,j = p,temp;
static  pass;
while(i<j)
{
        while(x[i]< = v && i<p)
                i++;
        while(x[j]>v)
                j– –;
        if(i<j)
        {
                temp = x[i];
                x[i] = x[j];
                x[j] = temp;
        }
}
x[m] = x[j];
x[j] = v;
printf("Completion of Pass %d",++pass);
display(x,n,j);
return j;
}/* end of partition */
/* function quickr call recursively array until lower bound is smaller than
upper  bound */
void quickr(int x[],int p,int q,int n)
{
int j;
```

```
if(p<q)
{
        j = partition(x,p,q,n);
        quickr(x,p,j–1,n);
        quickr(x,j+1,q,n);
}
}/* end of quickr */

void main()
{
int x[MAXSIZE],n,i;
clrscr();

printf("Enter the number of elements:[1–20]");
scanf("%d",&n);

printf("\nEnter the elements: \n");
for(i = 0;i<n;i++)
scanf("%d",&x[i]);
printf("\n");

quickr(x,0,n–1,n);

printf("\nThe sorted output:\n");
for(i = 0;i<n;i++)
printf("\n%d",x[i]);

getch();
}/* end of main */
```

Analysis of Quick Sort

For analyzing quick sort, we shall count only number of element comparisons.

The partition function involves n comparisons with the pivot or dividing element; with at most n/2 swaps. The swap pointers i and j scan the array from left and right side respectively till they cross. Hence the complexity of comparison in one partition is O(n).

So the complexity of quicksort depends on number of partition performed. If the pivot is such that set is divided into two subsets of the same size, then the complexity of quick sort is similar to merge sort. In this case, complexity is $O(n \log_2 n)$.

In the worst case, each partitioning will divide the set into two subsets of 1 and n–1 size. Then partition would be called n–1 times. So the number of comparison required $O(n^2)$.

Now consider the stack space needed by the recursion. In the worst case the maximum depth of recursion may be n–1.

The amount of stack space needed may be reduced to $O(\log_2 n)$ by using iterative version of quick sort in which smaller of the two subsets x[0] to x[j–1] and x[j+1] to x[n–1] is always sorted first. Also, second recursive call may be replaced by some assignment statements and jump to start of function.

The iterative quick sort function calls partition function to partition the set into two disjoint sets on the basis of pivot or partitioning element as previously done.

Example 2. Consider the previous data set. The lower bound p = 0, and upper bound q = 7.

Iteration 1

(*i*) The quick sort function calls partition function if p<q (i.e., 0 <7) and partition function returns an index j where the set is split into two subsets. After partitioning the array is as follows:

scan pointers

x[] = **25** 29 32 36 38 40 44 54 j = 0

(*ii*) Compare (j–p) with (q–j), (0–0) with (7–0) i.e., 0 < 7, store the lower bound and upper bound index value for set 2 into stk array. stk [ ] = 1 7 tos = 2

and q = j–1 i.e., q = –1

As (p<q) i.e., (0<–1) is false so while loop will not be terminated.

(*iii*) tos = 2; no return, need more iteration.

(*iv*) Popped the stack value into upper bound and lower bound q and p variables i.e., q = 7, p = 1. The stk array is empty. stk [ ] = NULL, tos = 0

Iteration 2

Now again execution enter in while loop is p<q i.e., 1<7

(*i*) Call the partition function

25 29 36 32 38 44 40 54 j = 1

(*ii*) Compare (j–p) with (q–j) i.e. 7– 0 < 7– 1. Now store the lower bound and upper bound values for partition subset 2 into stk array

stk [ ] = 2 7 tos = 2 and q = j–1 i.e., q = 0

and while loop is terminated.

(*iii*) tos = 2; no return, need more iteration.

(*iv*) Popped the stack value into q and p i.e. q = 7, p = 2. The stk array becomes empty and tos = 0

Iteration 3

(*i*) **25 29 {32} 36** {38 44 40 54} j = 3

(*ii*) Compare (j–p) with (q–j) i.e., (3–0< 7–4). Store the lower bound and upper bound values for further partitioning subset 2 with stk array.

stk [ ] = 5 7 tos = 2 and q = 2

Test p<q i.e., 0<2 is true. Call the partition function and return index j.

This process is continued until the p<q and final result will be sorted as

25 29 32 36 38 44 40 54

The non–recursive quick sort function is given below. It uses explicit stack, here a stk array is used to place element in stack. The function quicki is the iterative function for quick sort.

```
void quicki(int x[],int n)
{
int p=0,q=n–1,j,stk[20],tos=0,num=0;
do
{
while(p<q)
{
j = partition(x,p,q);
```

```
num++;
if((j–p)<(q–j))
{
        stk[tos++] = j+1;
        stk[tos++] = q;
        q = j–1;
}
else
{
        stk[tos++] = p;
        stk[tos++] = j–1;
        p = j+1;
}
}
if(tos = = 0)
{
        printf("\nComparisons: %d\n",num);
        return;
}
p = stk[– –tos];
q = stk[– –tos];
}while(1);
}
```

A complete 'C' program can be written very easily using this quicki function and the partition function for quick sort non-recursive version.

9.2.9 Heap Sort

Before discussing heap sort, we will begin by defining a new structure, the heap. We can define minimum heap and maximum heap.

A max (or min) heap is a complete binary tree with the property that the value at each node is at least as large as (or as small as) the values of its children (if they exist). This is called heap property.

A heap is a complete balanced binary tree in which each node satisfies the heap property and every leaf is of depth d or d–1.

The definition of a max heap implies that one of the largest elements is at the root of the heap. If the elements are distinct, then the root contains the largest item. A max (or min) heap can be implemented using an array.

A convenient data structure for a heap is an array say x, where x[i] is the element stored at the root, and x[2*i+1] and x[2*i+2] are the elements stored at the left and right children (if they exist) of the node at which element x[i] is stored.

Assume that n elements are initially stored in the array x, x[i], $0 \le i \le n–1$. To build the heap, the elements in x are rearranged to satisfy the heap property:

$$x[i] \ge x[2*i+1] \text{ for } 0 \le i \le n/2, \text{ and } x[i] \ge x[2*i+2] \text{ for } 0 \le i \le n/2.$$

We can also write this condition for the node at i index such that x[i] ≤ x[i/2] for 0 ≤ i ≤ n–1. Here i/2 is integer division.

We will now study the operations possible on a heap and see how these can be combined to generate a sorting algorithm.

Construction of a Heap

We can construct heap in two ways:

1. Top-down heap construction
   - Insert elements/nodes one by one into an initially empty heap. Keeping the heap property/ condition incorporate at all steps.
2. Bottom-up heap construction
   - Build a heap with the nodes in the order given.
   - From the rightmost nodes satisfy the heap condition.

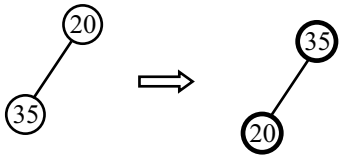
Example 1. Build a heap of the following nodes having integer value using both methods of construction:

20, 35, 9, 26, 49, 78, 2, 46

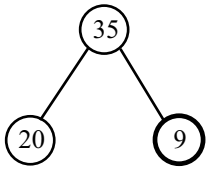
Top-down construction is as given in Figure 9.3.

1. The first node 20 is inserted as the root of the tree

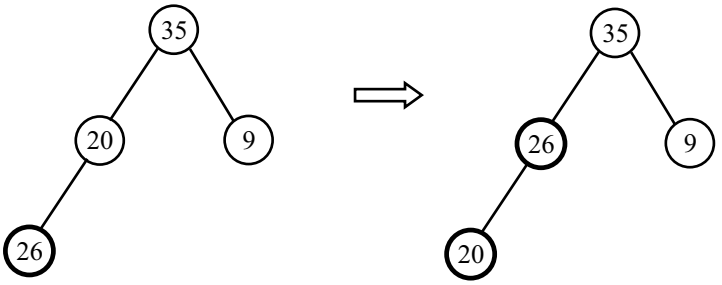
2. Insert second node 35, to the left of the node 20. But it violates the max heap property, so first interchange the node value.



3. Insert third node 9 to the right of node 35

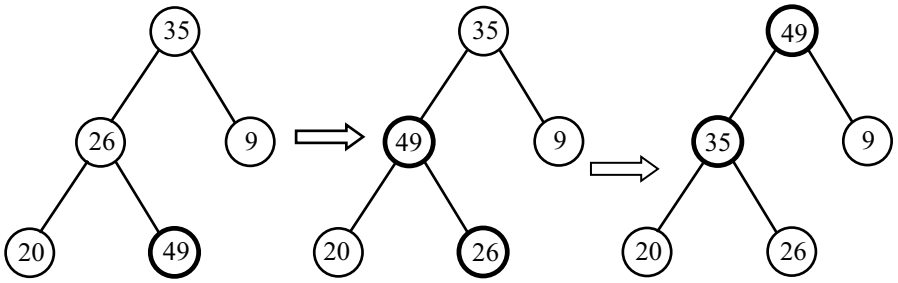


4. Insert fourth node 26, to the left of the node 20

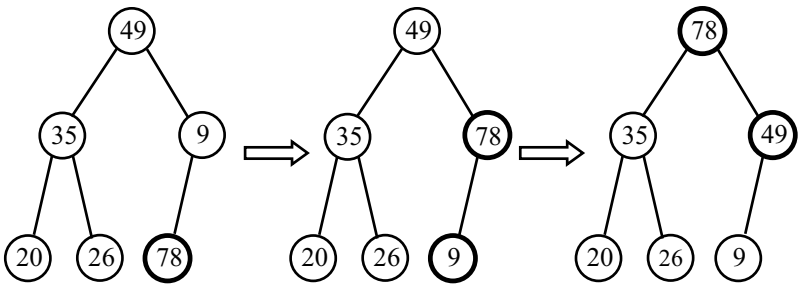


(***Figure 9.3–contd...***)

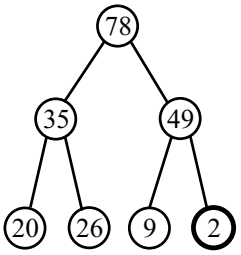
5. Insert fifth node 49 to the right of node 26



6. Insert sixth node 78 to the left of node 9



7. Insert node 2 to the right of node 49



8. Insert node 46 to the left of node 20

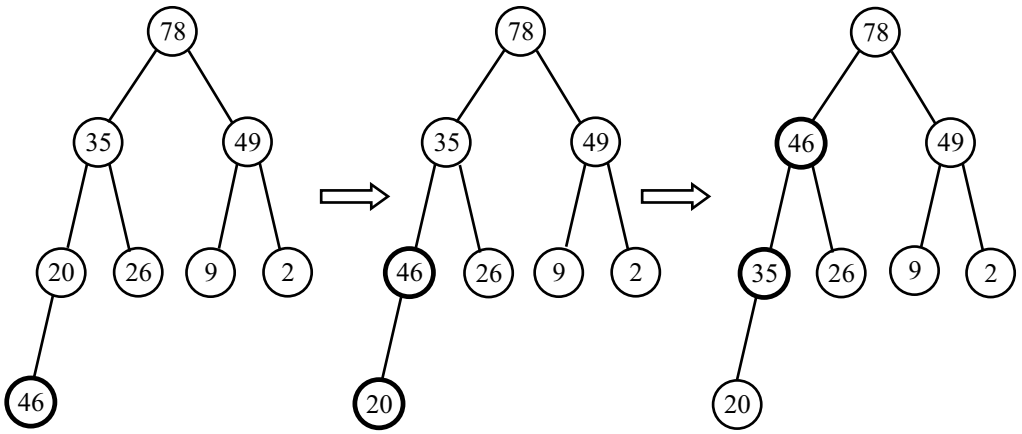


Figure 9.3 *Heap construction*

Algorithm for construction of maximum heap using top down approach

The function to insert a node into initially empty heap is given below.

```
/* function inserts the element into heap and heap is maintained in array a */
void insert(int a[],int n)
{
int i = n, item = a[n];
while((i>1) && (a[i/2]<item))
{
        a[i] = a[i/2];
        i = i/2;
}
a[i] = item;
}
```

Bottom-Up Heap Construction

The elements are stored in the array as their order. The first element is the root i.e., index 0 and at index 2*i+1 left child and at position 2*i+2 right child and so on forming complete binary tree as in Figure 9.4.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
20	35	9	26	49	78	2	46

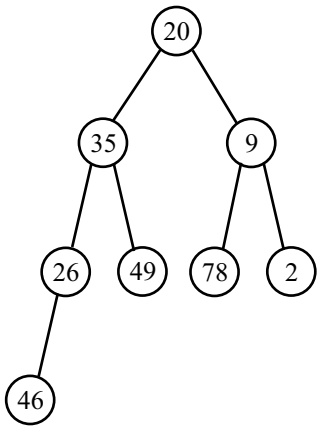
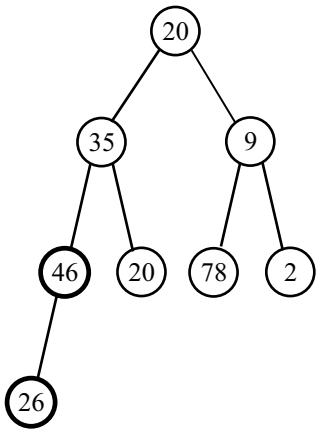


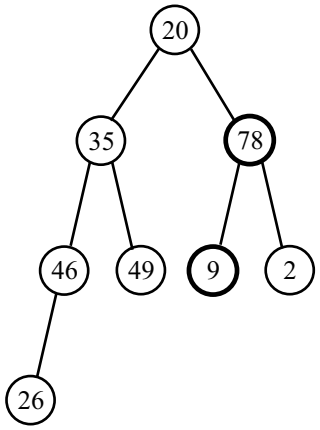
Figure 9.4

From the rightmost nodes check the heap condition and progress of heap is illustrated in Figure 9.5.

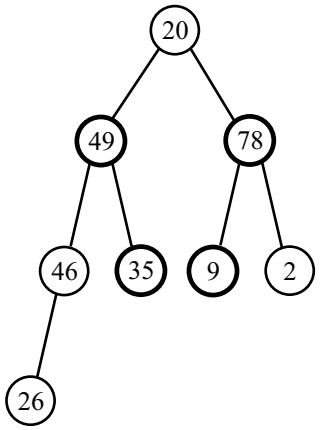
1. The last node is 46, which is greater than its parent node 26. Thus it interchanges in the following manner.



2. Node 2 needs no interchanges.
3. Node 78 needs interchanges.

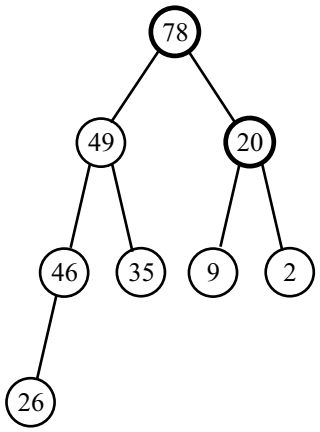


4. Node 49 needs interchanges.



5. Node 46 needs no interchanges.
6. Node 78 needs interchanges.

(*Figure 9.5–contd...*)



7. Node 49 needs no interchanges.

Figure 9.5 *Heap construction (Bottom up approach)*

Function for Heap Construction Using Bottom-Up Approach

Given n elements in array x[], we can create a heap by applying adjust function. It is easy to see that leaf nodes are already heap. So we can begin by calling adjust for the parents of leaf nodes and then level by level, until the root is reached. The function heapify call adjust function for i = n/2 to 1 and rearrange the heap to maintain its property.

```
void heapify(int x[ ],int n)
{
int i;
for(i = n/2;i>0;i– –)
        adjust(x,i,n);
}
void adjust(int a[],int i,int n)
{
int j = 2*i,item = a[i];
while(j< = n)
{
if((j<n) && (a[j]<a[j+1]))
        j = j+1;
if(item> = a[j])
        break;
        a[j/2] = a[j];
        j = 2*j;
}
a[j/2] = item;
}
```

These two are basic fundamental methods of heap construction. Now we will use top down method for heap sort.

Heap Sort Methods

1. Top-down heap sort: It is based on top-down approach for heap construction.

2. Bottom-up heap sort: It is based on bottom-up approach for heap construction.

1. Top-down approach for heap sort: Once the elements of array x have been arranged into a heap, elements are removed one at a time from the root node. This is done by interchanging x[0] with x[n – 1] and rearranging x[0], x[1], ..., x[n–2] into a maximum heap. Next x[0] and x[n–2] are interchanged and elements x[0], x[1], ..., x[n–3] are rearranged into a heap and so on until heap consists of one element at that point. x[0], x[1],... ,x[n–1] is the sorted sequence.

Example 2. Consider the above example of top-down approach for construction of heap.

The elements after nodes are arranged in maximum heap and are stored in array x as in Figure 9.6.

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
78	46	49	35	26	9	2	20

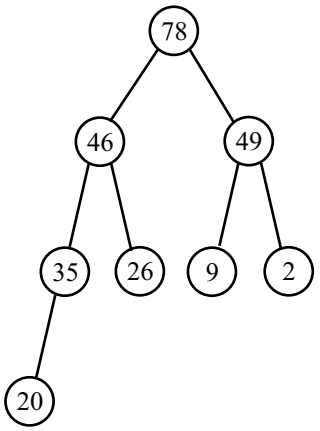
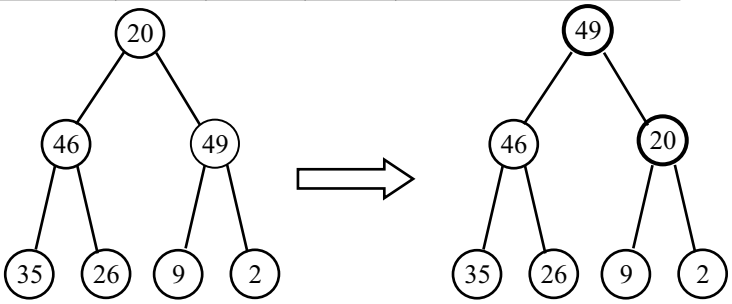


Figure 9.6

Now remove the root element 78, interchange with last element of the heap 20 & reconstruct the heap for elements x[0] to x[6].

Array – x

0	1	2	3	4	5	6	7
20	46	49	35	26	9	2	78

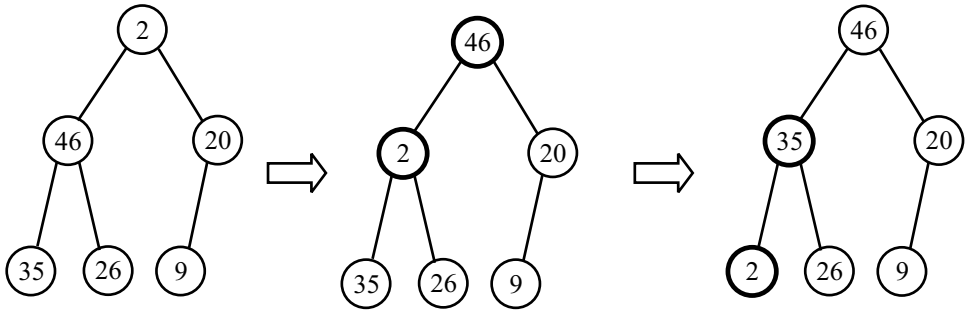


Now node element 20 is less than its both left and right children so replace it with large child element i.e., right child 49. Now it satisfies heap condition. The array representation of the heap is

0	1	2	3	4	5	6	7
49	46	20	35	26	9	2	78

Remove root node element 49 and interchange with element x[n–2], i.e., with 2 and reconstruct heap for x[0] to x[5].

0	1	2	3	4	5	6	7
2	46	20	35	26	9	49	78

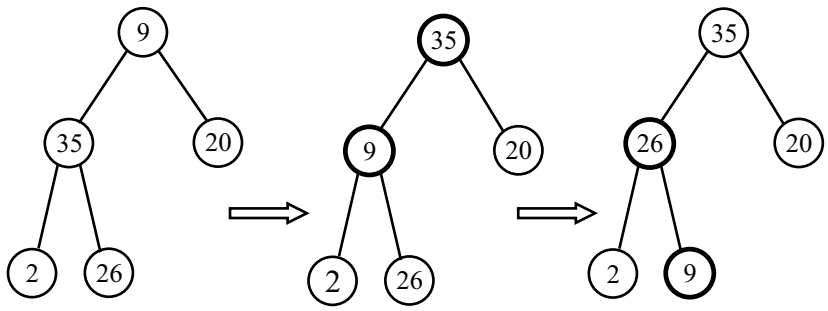


Now

0	1	2	3	4	5	6	7
46	35	20	2	26	9	49	78

Remove root node element 46 and interchange with element x[n–3], i.e., with 9 and reconstruct heap for x[0] to x[4].

0	1	2	3	4	5	6	7
9	35	20	2	26	46	49	78

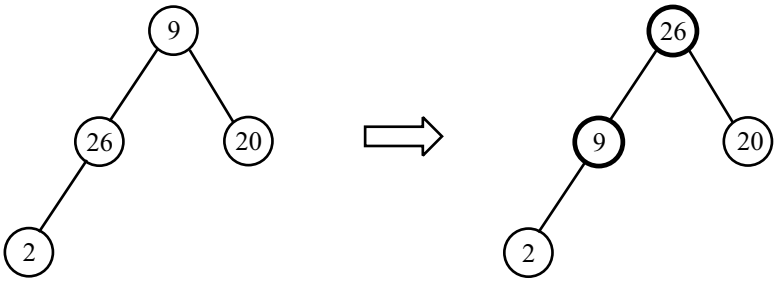


Now array x is:

0	1	2	3	4	5	6	7
35	26	20	2	9	46	49	78

Remove root node element 35 and interchange with element x[n–4], i.e., with 9 and reconstruct heap for x[0] to x[3].

0	1	2	3	4	5	6	7
9	26	20	2	35	46	49	78

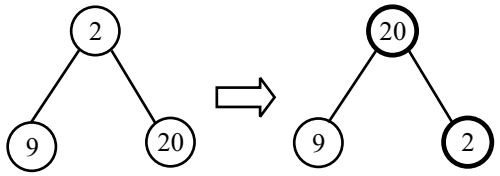


Now array x is:

0	1	2	3	4	5	6	7
26	9	20	2	35	46	49	78

Remove root node element 26 and interchange with element x[n–5], i.e., with 2 and reconstruct heap for x[0] to x[2].

0	1	2	3	4	5	6	7
2	9	20	26	35	46	49	78

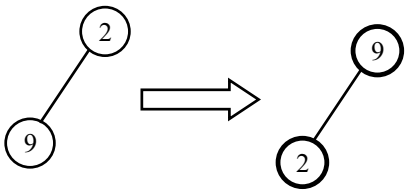


Now array x is:

0	1	2	3	4	5	6	7
20	9	2	26	35	46	49	78

Remove root node element 20 and interchange with element x[n–6], i.e., with 2 and reconstruct heap for x[0] to x[1].

0	1	2	3	4	5	6	7
2	9	20	26	35	46	49	78



Now array x is:

0	1	2	3	4	5	6	7
9	2	20	26	35	46	49	78

Remove root node element 9 and interchange with element x[1], i.e., with 2 and now array x is sorted.

0	1	2	3	4	5	6	7
2	9	20	26	35	46	49	78

The function to sort elements in heap using top-down approach is given below. The function hsorttd sort the elements in array a into ascending order using maximum heap construction.

The function hsorttd call insert and delmax function, where delmax function calls adjust function to maintain the max heap property.

```
void hsorttd(int a[],int n)
{
int i;
for (i = 1;i< = n;i++)
        insert(a,i);
for(i = n;i>0;i– –)
        delmax(a,i);
}
void adjust(int a[],int i,int n)
{
int j = 2*i,item = a[i];
while(j< = n)
{
        if((j<n) && (a[j]<a[j+1]))
                j = j+1;
        if(item> = a[j])
                break;
        a[j/2] = a[j];
        j = 2*j;
}
a[j/2] = item;
}
void delmax(int a[],int n)
{
int x;
x = a[1],a[1] = a[n];
a[n] = x;
adjust(a,1,n–1);
}
```

In worse case the number of executions of the while loop is proportional to the number of levels in the heap. Thus if there are n elements in the heap inserting a new element takes ($\log_2 n$) time in the worse case.

A complete 'C' program to sort n elements using top-down approach is given below:


```
/* Program heap_t_d.c is heap sort using top down heap construction method */
   #include<conio.h>
   #include<stdio.h>
void insert(int a[],int n)
{
int i = n,item = a[n];
while((i>1) && (a[i/2]<item))
{
        a[i] = a[i/2];
        i = i/2;
}
a[i] = item;
}
void adjust(int a[],int i,int n)
{
int j = 2*i,item = a[i];

while(j< = n)
{
        if((j<n) && (a[j]<a[j+1]))
                j = j+1;
        if(item> = a[j])
                break;
        a[j/2] = a[j];
        j = 2*j;
}
a[j/2] = item;
}
void delmax(int a[],int n)
{
int x;
x = a[1],a[1] = a[n];
a[n] = x;
adjust(a,1,n–1);
}
void hsorttd(int a[],int n)
{
```

```
int i;
for (i = 1;i< = n;i++)
        insert(a,i);
for(i = n;i>0;i– –)
        delmax(a,i);
}
void main()
{
int x[10],n,i;
clrscr();
printf("Enter the number of elements: ");
scanf("%d",&n);
printf("\nEnter the elements: \n");
for(i = 1;i< = n;i++)
scanf("%d",&x[i]);
hsorttd(x,n);
printf("\nThe sorted output:\n");
for(i = 1;i< = n;i++)
printf("\n%d",x[i]);
getch();
}
```

2. Bottom-up approach for heap sort: It is possible to devise an algorithm that can perform n inserts in O(n) time rather than $O(n\log_2 n)$. This reduction is achieved by an algorithm that regards any array a as a complete binary tree and works from the leaves up to the root level by level. At each level, the left and right subtrees of any nodes are heap. Only the value in the root node may violate the heap property. The method is known as bottom-up heap sort. The function for this heap sort is given below.

```
void hsortbu(int a[],int n)
{
int i,t;
heapify(a,n);
for(i = n;i>1;i– –)
{
t = a[i];
a[i] = a[1];
a[1] = t;
adjust(a,1,– –n);
}
}
```

The function hsortbu calls heapify and adjust functions to construct heap using bottom-up approach and then it interchanges the first and last index values in the array.

However, heapify requires that all the elements be available before heap creation begins. Using insert function, we can add a new element into this heap at any time.

A complete 'C' program to sort n elements using bottom-up approach is given below:


```
/* Program heap_b_u.c is heap sort method using bottom up heap construction method. It uses
heapify, adjust and hsortbu functions */
  #include<conio.h>
  #include<stdio.h>

void adjust(int a[],int i,int n)
{
int j = 2*i,item = a[i];

while(j< = n)
{
        if((j<n) && (a[j]<a[j+1]))
                j = j+1;

        if(item> = a[j])
                break;

        a[j/2] = a[j];
        j = 2*j;
}
a[j/2] = item;
}

void heapify(int x[],int n)
{
int i;

for(i = n/2;i>0;i– –)
        adjust(x,i,n);
}

void hsortbu(int a[],int n)
{
int i,t;

heapify(a,n);

for(i = n;i>1;i– –)
{
        t = a[i];
        a[i] = a[1];
        a[1] = t;
        adjust(a,1,– –n);
```

```
}
}
void main()
{
int x[10],n,i;
clrscr();
printf("Enter the number of elements:");
scanf("%d",&n);
printf("\nEnter the elements: \n");
for(i=1;i<=n;i++)
scanf("%d",&x[i]);
hsortbu(x,n);
printf("\nThe sorted output:\n");
for(i=1;i<=n;i++)
printf("\n%d",x[i]);
getch();
}
```

Example 3. Find the number of comparisons for bottom up heap sort for following set of data.

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
x[]=	25	29	36	32	38	44	40	54

(*i*) Initially hsortbu function calls heapify function to construct maximum heap using bottom up technique, the heapify function further calls adjust function to maintain heap property.
Initial binary tree is shown in Figure 9.7(i).

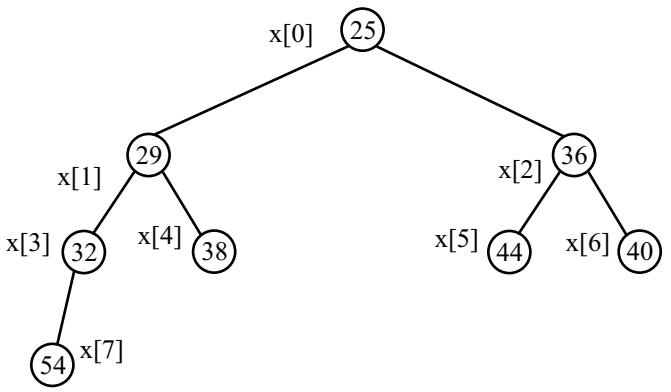
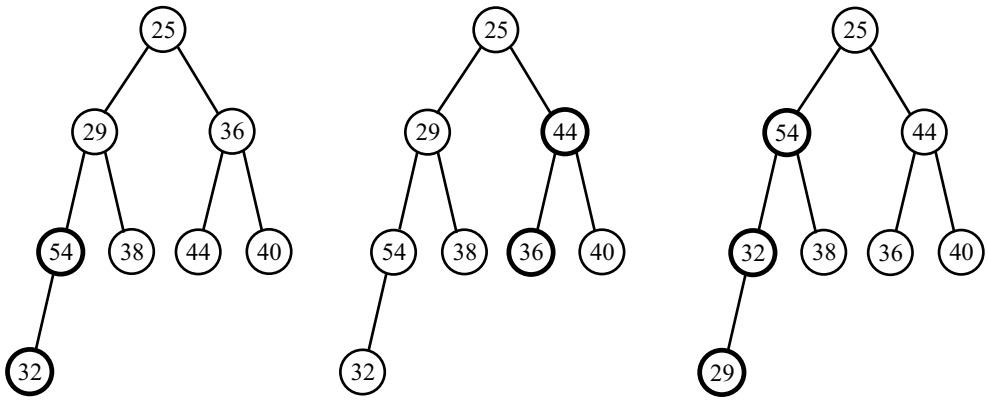


Figure 9.7 (*i*)

The leaf nodes are already heaps. Adjust the heap level by level, until the root is reached.

(*ii*) The first call to adjust function has at i = n/2 (i.e i = 7/2 = 3). The two elements 32 and 54 are rearranged to form a heap as in Figure 9.7(*ii*).



(*ii*) (*iii*) (*iv*)

Figure 9.7

(*iii*) Subsequently adjust with i = 2, rearrange nodes 44, 36, and 40 as in Figure (iii).

(*iv*) As with i=1, rearrange nodes 29, 54, and 38.

(*v*) With i=0 rearrange the below all nodes. So binary tree is in heap.

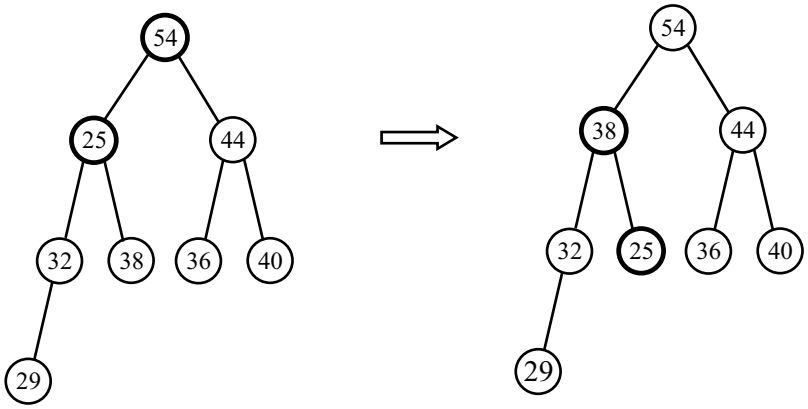


Figure 9.7 (*v*)

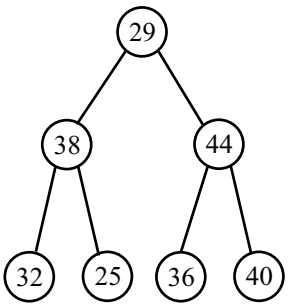
Now the resultant array x is

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
x[] =	54	38	44	32	25	36	40	29

Interchange the x[0] and x[7] and call adjust for x[0] to x[6] elements and heapify it.

x[ ] = 29 38 44 32 25 36 40 54

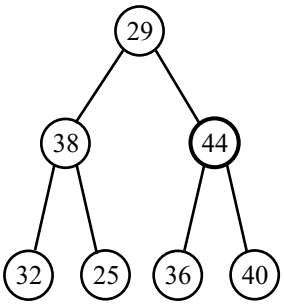
Initial heap for the elements x[0] to x[6] is in Figure 9.8(i).



Level
0
1
2

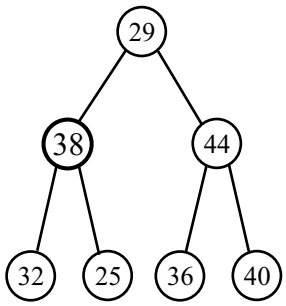
(*i*)

The adjust function calls for elements at levels 2, 1 and 0 to heap as in Figures 9.8 (ii), (iii) & (iv).



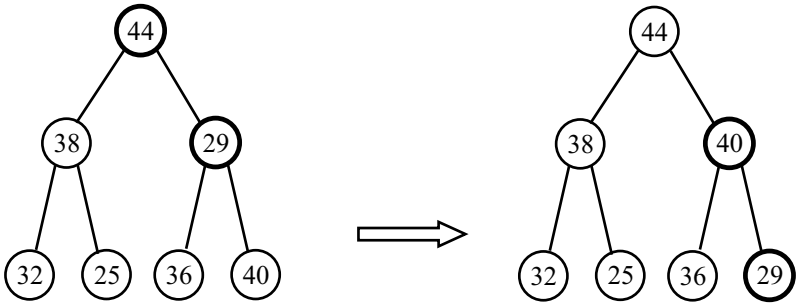
(*ii*)

No change as 44 follows heap property



(*iii*)

No change as 38 follows heap property



(*iv*)

Interchange as 29 does not follows heap property

Figure 9.8

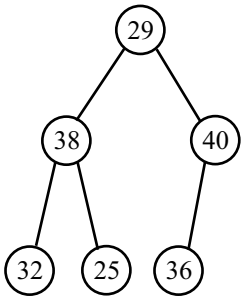
The resultant array is

x[] = 44 38 40 32 25 36 29 54

Now interchange x[0] with x[6], again repeat the whole process.

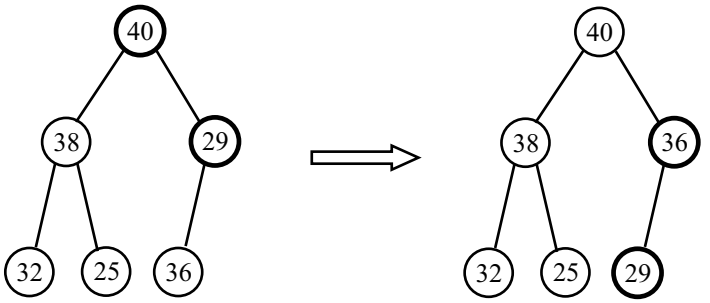
x[] = 29 38 40 32 25 36 44 54

Initial heap for remaining element is given below:



Level
0
1
2

No change in level 2 and level 1 elements as they maintained heap property.
The heap for root node is

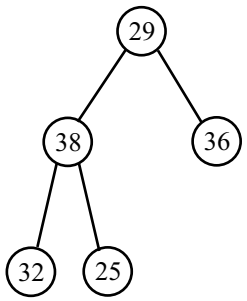


The array is x[] = 40 38 36 32 25 29 44 54

Interchange x[0] with x[5], the resultant array is

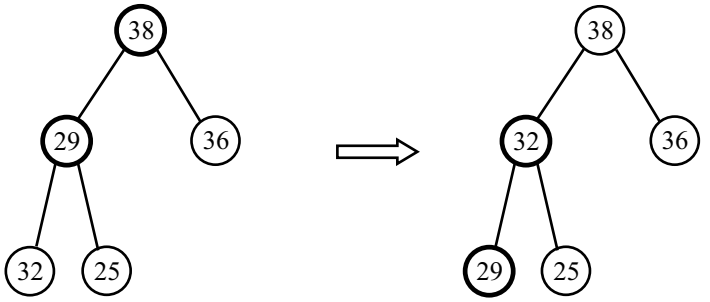
x[] = 29 38 36 32 25 40 44 54

Again heap the elements and adjust



Level
0
1
2

As leaves are themselves heaps, so level 1 nodes are in heap. So heap the element at level 0 i.e. root.



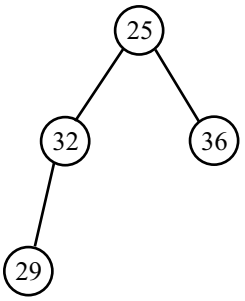
The result of array x is

x[] = 38 32 36 29 25 40 44 54

Interchange x[0] with x[4] the array is

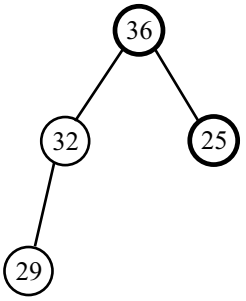
x[] = 25 32 36 29 38 40 44 54

Again apply heapify and adjust



Level
0
1
2

As leaves nodes are themselves heap, Level 1 node is exhibiting the proportion of heap. So root node needs to be heapified.



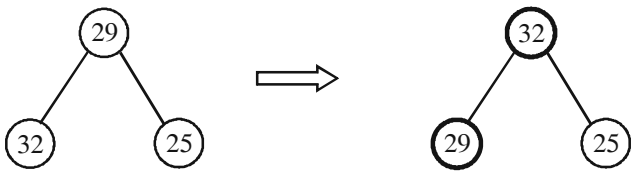
The array is

x[] = 36 32 25 29 38 40 44 54

Interchange x[0] with x[3]

x[ ] = 29 32 25 36 38 40 44 54

And apply heapsort on x[0] to x[2]



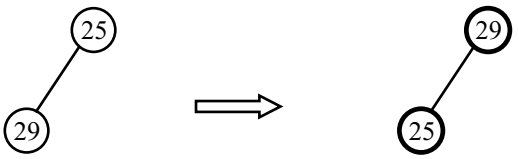
The array x is

x[] = 32 29 25 36 38 40 44 54

Interchange x[0] with x[2]

x[] = 25 29 32 36 38 40 44 54

Again apply heapsort



Array x is

x[] = 29 25 32 36 38 40 44 54

Interchange x[0] with x[1]

x[ ] = 25 29 32 36 38 40 44 54

Result is sorted.

Example 4. Construct heap and heapsort using top-down approaches

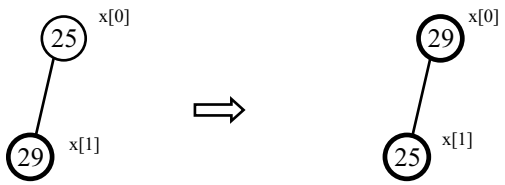
x[] = 25 29 38 32 36 44 40 54

Fig. 9.9 shows the data is moved in array until heap is created. Trees in the left column represent state of the array x before each call of function insert. Trees in the right column shows how the array altered by insert to produce a heap. The array is drawn as a complete binary tree in Fig. 9.9.

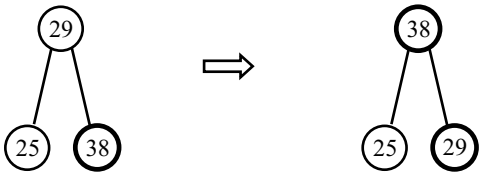
Left Column Right Column

(*i*) Insert 25

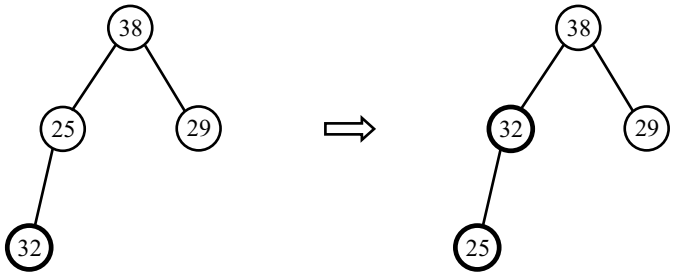
(*ii*) Insert 29



(*iii*) Insert 38

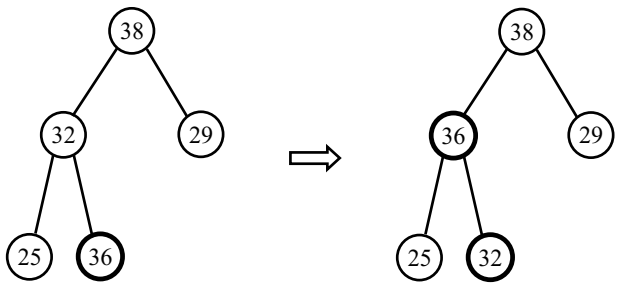


(*iv*) Insert 32

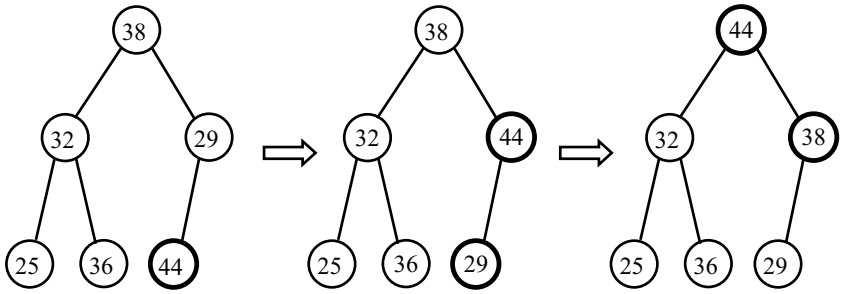


(***Figure 9.9–contd...***)

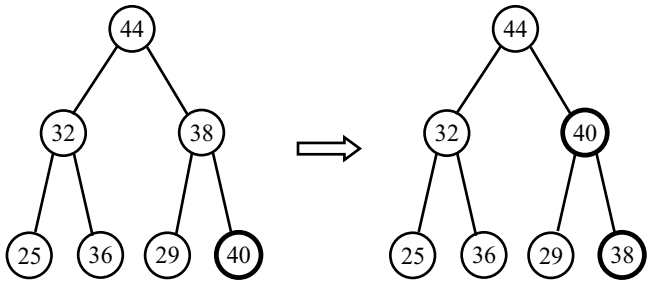
(*v*) Insert 36



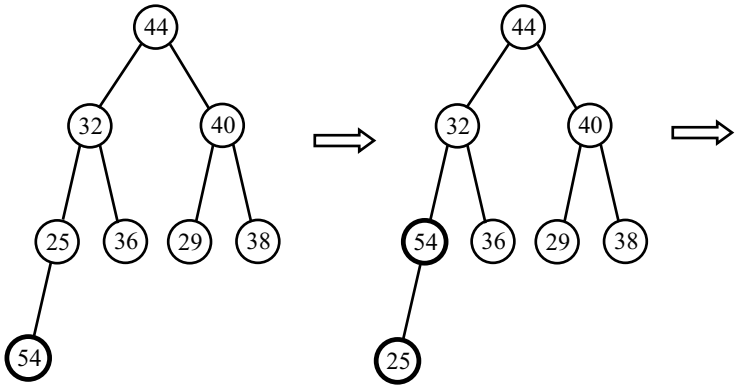
(*vi*) Insert 44



(*vii*) Insert 40



(*viii*) Insert 54



(***Figure 9.9–contd...***)

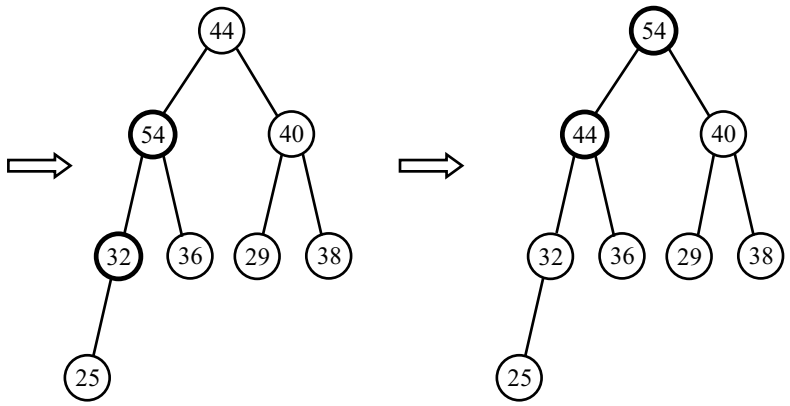


Figure 9.9 *Heap construction (top-down approach)*

The array x is

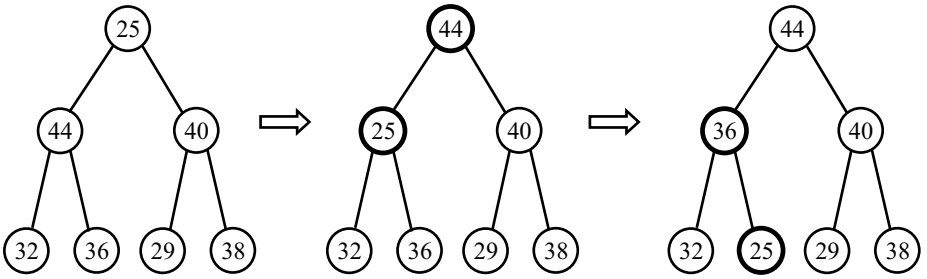
(*i*)

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
x[ ] =	54	44	40	32	36	29	38	25

Call delmax function, which interchanges x[0] with x[7] and the array x is

x[ ] =	25	44	40	32	36	29	38	54

Now apply the adjust function of the remaining elements set to get heap.



The array x is

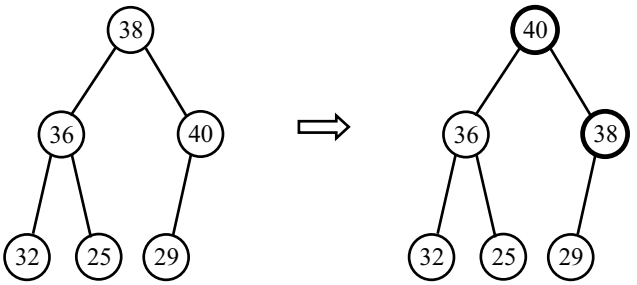
(*ii*)

x[ ] =	44	36	40	32	25	29	38	54

Call delmax function, which interchanges x[0] with x[6] and the array x is

x[ ] =	38	36	40	32	25	29	44	54

Adjust the set to get the heap



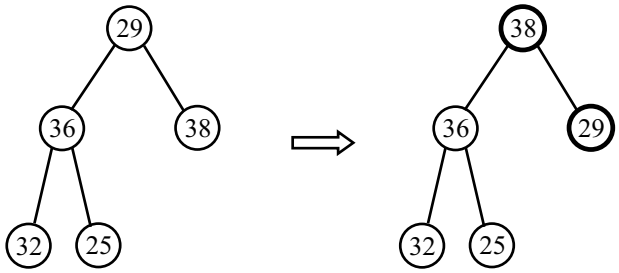
The array x is

(*iii*) x[] = 40 36 38 32 25 29 44 54

Call delmax to interchange x[0] with x[5] and the array is

x[] = 29 36 38 32 25 40 44 54

Adjust the remaining set to get the heap.



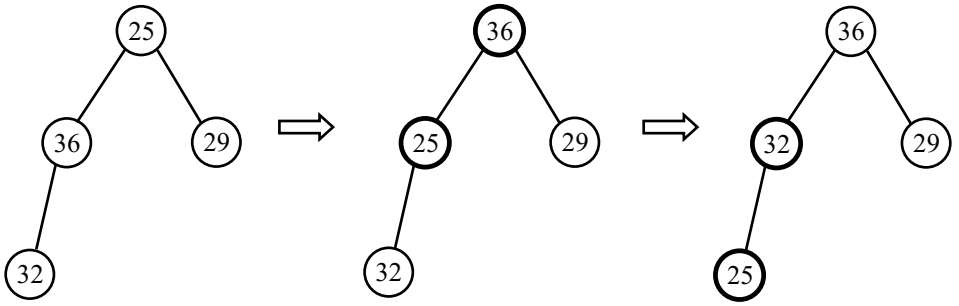
The array x is

(*iv*) x[] = 38 36 29 32 25 40 44 54

Call delmax function to interchange x[0] with x[4] and the array is

x[] = 25 36 29 32 38 40 44 54

Adjust the remaining set to get the heap.



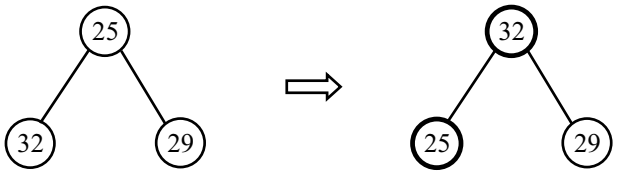
The array x is

(*v*) x[] = 36 32 29 25 38 40 44 54

Call delmax to delete (i.e. interchange) x[0] with x[3] and the array is

x[] = 25 32 29 36 38 40 44 54

Adjust the remaining set to get the heap.



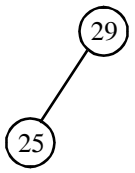
The array x is

(*vi*) x[] = 32 25 29 36 38 40 44 54

Call delmax to interchange X[0] with x[1] and the array is

x[] = 29 25 32 36 38 40 44 54

Adjust the remaining set to get the heap.



It is max heap

(*vii*) The array x is

x[] = 29 25 32 36 38 40 44 54

Call delmax to interchange x[0] with x[1] and the array is

x[ ] = 25 29 32 36 38 40 44 54

Hence the sorted set is obtained.

The maximum number of comparisons needed is $O(n \log_2 n)$

Summary of Sorting Methods

The sorting methods discussed so far are summarized in below table. The parameter n denotes the number of elements (or keys) and m denotes the number of digit in a key. It is used in radix sort.

The goodness of any sorting algorithm depends upon the properties such as the number, size, distribution and order of keys in the set. The amount of memory available in performing the sort may also be an important factor.

In summary, the selection, bubble or insertion sort can be used if the number of elements in set is small. If n is large and keys are short (i.e. m is low), the radix sort can perform well.

With a large n and long keys, heap sort, quick sort or merge sort can be used.

If the set of elements almost sorted then quick sort should be avoided. When the keys, after hashing, are uniformly distributed over the interval [1,m], then an address calculation sort is key good method.

Both merge sort and quick sort are methods based on 'divide–and–conquer' that involve splitting the set into two parts, sorting the two parts separately, and then joining the two sorted halves together. In merge sort, the splitting is easy and joining is hard. In quick sort, the splitting is hard (partitioning) and joining is easy (as the few halves and the pivot automatically form a sorted array).

Insertion sort may be considered a special case of merge sort in which the two halves consist of a single element and the remainder of the array. Selection sort may be considered a special case of quick sort in which the set is partitioned into one half consisting of the largest element alone and second half consisting of the remainder of the array.

Comparison of Sorting Method (in approximation) Time Complexity

Algorithm	*Average*	*Worst case*	*Space usage*
Selection Sort	$n^2/2$	$n^2/2$	In place
Bubble Sort	$n^2/2$	$n^2/4$	In place
Insertion Sort	$n^2/4$	$n^2/2$	In place
Merge Sort	$O(n\log_2 n)$	$O(n\log_2 n)$	Extra n entries
Quick Sort	$O(n\log_2 n)$	$O(n^2)$	Extra $\log_2 n$ entries
Heap Sort	$O(n\log_2 n)$	$O(n\log_2 n)$	In place
Radix Sort	$O(m+n)$	$O(m+n)$	Extra space for links
Address Calculation Sort	$O(n)$	$O(n^2)$	Extra space for links
Shell Sort	$O(n(\log_2 n)^2)$	$O(n^2)$	Extra space for increment array

Chapter 10

Searching Techniques

10.1 SEQUENTIAL SEARCH

This is the simplest technique to find out an element in an unordered array. Suppose aElem is a linear array with n elements and sElem is the element to be searched in the array. In linear search, the search element is compared with the first element of the array, if match is found then search is successful and terminated. Otherwise next element of the array is compared with the search element and this process is continued till the search element is found or array is completely exhausted.

A linear search algorithm for unordered array is shown below:

Algorithm LSEARCH(aElem, nElem, sElem)

/* Here aElem is a linear array with n elements i.e. nElem, sElem is the element to be searched and pofElem is return the position of the search element in the array, if posElem is –1 then search is unsuccessful */

Step 1 : pofElem = – 1
Step 2 : for i = 0 to nElem – 1 do
{
Step 3 : if (sElem = aElem[i]) then
pofElem = i;
i = nElem;
}
Step 4 : return pofElem;
Step 5 : end LSEARCH

The time complexity of sequential search in worst case is O(n), worst case occurs when one must search through the entire list of elements n. The best case of successful search is O(1), means element to be searched is the first element in the list.

For unsuccessful search, the time complexity is O(n), as all the elements in the list has to be processed.

10.2 BINARY SEARCH

Binary search is already discussed in Chapter 3: Ordered Arrays. Let us consider an example to search an element from a given sorted list of 10 elements. The underline elements are low, mid and high index in the array.

Find an element whose key is 80.

Comparison										
1 (unsuccessful)	10	20	30	40	50	60	70	80	90	100
2 (successful)	10	20	30	40	50	60	70	80	90	100

Find an element whose key is 50.

Comparison										
1 (successful)	10	20	30	40	50	60	70	80	90	100

Find an element whose key is 25.

Comparison										
1 (unsuccessful)	10	20	30	40	50	60	70	80	90	100
2 (unsuccessful)	10	20	30	40	50	60	70	80	90	100
3 (unsuccessful)	10	20	30	40	50	60	70	80	90	100

From above examples it is clear that the running time for the worst case is approximately equal to $O(\log_2 n)$. Successful search for best case is O(1).

For unsuccessful search, best case, worst case, and average case time complexity is $O(\log_2 n)$.

The priority queue and dictionary is good application for searching an element.

10.3 HASHING

In all searching algorithms considered so far, the location of an element is determined by a sequence of comparisons. The number of comparisons depend on the data structure and search algorithm used. For example:

Unsorted sequential array Implementation

- **insert**: add to back of array; O(1)
- **find**: search through the keys one at a time, potentially all of the keys; O(n)
- **remove**: find + replace removed node with last node; O(n)

Sorted sequential array Implementation

- **insert**: add in sorted order; O(n)
- **find**: binary search; $O(\log_2 n)$
- **remove**: find, remove node and shuffle down; O(n)

Linked list (unsorted or sorted) Implementation

- **insert**: add to front; O(1) *or O(n) for a sorted list*
- **find**: search through potentially all the keys, one at a time; O(n) *still O(n) for a sorted list*
- **remove**: find, remove using pointer alterations; O(n)

AVL tree Implementation

- An AVL tree, ordered by key
- **insert**: a standard insert; $O(\log_2 n)$
- **find**: a standard find, O(log n)
- **remove**: a standard remove; $O(\log_2 n)$

However there is a table organization and search technique in which insert, delete and search are possible in O(1) time. The concept lies here.

If each key (i.e. field) is to be retrieved in a single access, the location of the record within the table can depend only on the key, not on the locations of other keys, as in a tree. The most efficient way to organize such a table is as an array. If keys are integers, the keys themselves can serve as indices to the array.

What is necessary is some method of converting a key into an integer within a limited range. Ideally no two keys should be converted into the same integer. Unfortunately, such an ideal method usually does not exist.

A function that transforms a key into a table index is called a **hash function**. If h is a hash function and k is a key, h(k) is called the hash of key and is the index at which a record with the key k should be placed.

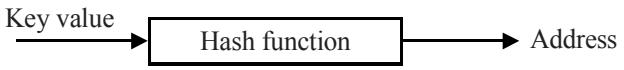


Figure 10.1 *Mapping from a key value to an address location*

The following key points are summarized:

- An array in which records are ***not*** stored consecutively — their place of storage is calculated using the key and a *hash function*
- Hashed key: the result of applying a hash function to a key
- Keys and entries are scattered throughout the array

Let us consider the example of a company with an inventory file which consists of parts number, parts name and quantity.

The original location of records with their index number is given below:

Index no.	***Parts number (key)***	***Parts name***	***Quantity***
0	123909	Floppy disk	200
1	123232	Hard disk	50
2	213421	CD drive	100
3	392827	SCSI	10
4	322486	Monitor	50
5	Empty		
6			
7			
8			
9			

After applying hash function h(k) = key mod 10 or key % 10 can produce any indices between 0 and 9, depending on the value of key k. The below table gives the direct location for key value:

Index no. (key % 10)	***Parts number (key)***	***Parts name***	***Quantity***
0	Empty		
1	213421	CD drive	100
2	123232	Hard disk	50
3	Empty		
4			

Contd...

Index no. (key % 10)	*Parts number (key)*	*Parts name*	*Quantity*
5			
6	322486	Monitor	50
7	392827	SCSI	10
8	Empty		
9	123909	Floppy disk	200

10.3.1 Hash Functions

The main characteristics of hash function h is:

- It should be possible to compute it efficiently
- It should distribute the keys uniformly across the locations i.e. it should keep the number of collisions as minimum as possible.

Some of the popular hash functions are:

1. Division Method

In division method, key k to be mapped into n locations or indices by finding the modulus or remainder of key k by n.

H(k) = key mod n or key % n

It requires only a single division (as reminder operation need first division) operation, so this hashing is quite fast.

The inventory example has used division method.

Consider a searching of a record with key is 392827. By using division hash function, the index where the record is stored is (392827 % 10) = 7.

2. Multiplication Method

Multiplication method has two steps:

(*i*) The key value k is multiplied by a constant C in the range 0 < C < 1 and extract the fractional part of the value k *C e.g. (k * C – (int)(k *C))

(*ii*) The fractional value obtained above is multiplied by n (i.e. number of indices) and the floor of the result is taken as the hash value or address of the key k.

The hash function is

H(k) = floor(n *(k * C – (int)(k *C)))

Example 1. Consider the inventory file, and n = 10, and constant C preferred value is 0.618, let us map the all keys to its location according to this hash function.

For key = 392827

= floor (10 * (392827 * 0.618 – (int)(392827 * 0.618)))

= floor (10 * (0.904))

= floor (9.04)

= 9

Similar calculation has been carried out for each key of the inventory file as address shown in the table below:

*floor (n *(k * C – (int)(k *C)))*	*Parts number (key)*	*Parts name*	*Quantity*
0	Empty		
1	213421	CD drive	100
2	Empty		
3	123232	Hard disk	50
4	Empty		
5			
6			
7	123909	Floppy disk	200
8	Empty		
9	392827	SCSI	10

The record with key 322486 is mapped to address location 3, which is already occupied by key 123232, so there is a collision. Thus this record can't be inserted.

3. Midsquare Method

This method also operates in two steps:

(*i*) The square of the key value k is taken

(*ii*) According to value of n, number of digit extracted from the above square as address.

Example 2. Consider the inventory file and n =10, so we will extract one middle digit from k^2.

The keys are mapped as below:

k	213421	123232	322486	392827	123909
k^2	45548**5**23241	15186**1**25824	103997**7**220196	15431**3**051929	15353**4**40281
h(k)	5	1	7	3	4

The hash values are obtained by taking 6^{th} digit from the k^2. The below table shows the inventory file where keys are mapped according to midsquare function.

Index no. (6^{th} digit of k^2)	*Parts number (key)*	*Parts name*	*Quantity*
0	Empty		
1	123232	Hard disk	50
2	Empty		
3	392827	SCSI	10
4	123909	Floppy disk	200
5	213421	CD drive	100
6	Empty		
7	322486	Monitor	50
8	Empty		
9			

There is no collision as each key is mapped in different locations/indices.

4. Folding Method

This method also operates in two steps:

(*i*) The key value k is divided into number of parts, k_1, k_2, .. k_r, where each part has the same number of digits except the last part, which can have lesser digits.

(*ii*) These parts are added together and hash value is obtained by ignoring the last carry., if any.

Example Consider the inventory file, here each key is divided into a single digit as number indices are 10. Add all parts and extract the least significant digit as indices of the key.

k	213421	123232	322486	392827	123909
Parts	2, 1, 3, 4, 2, 1	1, 2, 3, 2, 3, 2	3, 2, 2, 4, 8, 6	3, 9, 2, 8, 2, 7	1, 2, 3, 9, 0, 9
Sum of parts	13	13	25	31	24
hash (k)	3	3	5	1	4

There is collision as key 213421 and 123232 occupied the same hash value or location 3. The second key 123232 is not inserted in the below table.

Index no. LSD (sum of digits of key k)	***Parts number (key)***	***Parts name***	***Quantity***
0	Empty		
1	392827	SCSI	10
2	Empty		
3	213421	CD drive	100
4	123909	Floppy disk	200
5	322486	Monitor	50
6	Empty		
7			
8			
9			

*LSD–least significant digit.

10.3.2 Collision Resolution Techniques

A collision occurs when more than one keys map to same hash value in the hash table: Therefore we need some mechanism to handle them. The following schemes resolve the collisions:

1. Collision resolution by open addressing
2. Collision resolution by separate chaining

The performance of these methods depends on load factor i.e ratio p = m /n, where m is the number of records in the file and n is the number of hash addresses.

The efficiency of a hash function with a collision resolution technique is measured by the average number of probes (key comparisons) needed to find the location of the record with a given key k. The efficiency depends mainly on the load factor p and the following conditions of search:

(*i*) S(p) = average number of comparisons for a successful search

(*ii*) U(p) = average number of comparisons for an unsuccessful search

1. Open Addressing: Linear Probing

A simple approach to resolve collisions is to store the colliding record into the next available memory location. This technique is known as linear probing.

Linear probing resolved hash collisions by sequentially searching a hash table beginning at the location returned by the hash function (using circular array).

The linear probing uses the following hash function:

$h(k, i) = (h'(k) + i) \bmod n$ for $i = 0, 1, 2, \ldots n-1$

where n is the size of the hash table and $h'(k) = k \bmod n$ the basic hash function, and i is the probe number.

The number of comparisons for successful and unsuccessful search is given below.

$$S(p) = \frac{1}{2}\left(1 + \frac{1}{1-p}\right) \quad \text{and} \quad U(p) = \frac{1}{2}\left(1 + \frac{1}{(1-p)^2}\right)$$

Example 1. Let us insert the keys 99, 33, 23, 44, 56, 43, 19 into a hash table of size n =10 and hash function is division method i.e h(k) = k mod 10. The linear probing hash function is calculated as below. The keys are stored at location that is underlined.

k	99	33	23	44	56	43	19
h(k,0)	9	3	3	4	6	3	9
h(k,1)	-	-	4	5	-	4	0
h(k,2)	-	-	-	-	-	5	-
h(k,3)	-	-	-	-	-	6	-
h(k,4)	-	-	-	-	-	7	-

Now several keys map to the memory locations 3 and 9 so there is collision.

(*i*)

0	1	2	3	4	5	6	7	8	9
			33	23					99

For probe 0, insert key 23 at location 3 but there is collision. For probe 1 the address is 4, now insert the key.

(*ii*)

0	1	2	3	4	5	6	7	8	9
			33	23	44				99

For probe 0, insert key 44 at location 4 but there is collision. For probe 1 the address is 5, now insert the key.

(*iii*)

0	1	2	3	4	5	6	7	8	9
			33	23	44	56			99

For probe 0, insert key 56 at location 6 as there is no collision.

(*iv*)

0	1	2	3	4	5	6	7	8	9
			33	23	44	56			99

For probe 0, insert key 43 at location 3 but there is collision. For probe 1 the address is 4 but there is again collision. For probe 2 the address is 5 but there is again collision. For probe 3 the address is 6 but there is again collision. For probe 4 the address is 7, now insert the key as this location is free.

(*v*)

0	1	2	3	4	5	6	7	8	9
19			33	23	44	56	43		99

For probe 0 insert key 19 at location 9 but there is collision. For probe 1 the address is 0, now insert the key as this location is free.

A complete 'C' program to implement linear probing is given below:

SEARCH/HASHLP.C

```
/* Program hashlp.c inserts keys into an array with linear probing
technique of collsion resolution technique. The maximum number of keys
can be inserted is equal to MAXSIZE. The function hashlp inserts the key
into hash value using division method. */

#include<stdio.h>
#define MAXSIZE 10
int h[MAXSIZE];
void hashlp(int , int[]);
void hashlpsr(int key, int h[MAXSIZE]);
void display(int h[MAXSIZE]);
void main()
{
int i, key,ch ;
for(i = 0;i<MAXSIZE;i++)
    h[i] = –1;
do{
    printf("\n\n Program for insertion/searching keys with linear probing");
    printf("\n 1. Insert keys");
    printf("\n 2. Search key");
    printf("\n 3. Display keys");
    printf("\n 4. Exit");
    printf("\n Select operation");
    scanf("%d",&ch);
    switch(ch)
    {
    case 1: do{
                printf("\n Enter key value [type–1 for termination]");
                    scanf("%d",&key);
                    if (key ! = –1)
                            hashlp(key,h);
                    }while(key! = –1);
                    display(h);
```

```
                    break;
    case 2:   printf("\n Enter search key value");
                    scanf("%d",&key);
                    hashlpsr(key,h);
                    break;
    case 3: display(h);

                break;
    case 4: break;
    }
    }while(ch! = 4);
}  /* end of main function */

/* function hashlp find a location for key and insert it */
void hashlp(int key, int h[MAXSIZE])
{
 int loc;
 loc = key % MAXSIZE;
 while (h[loc] ! = –1)
      loc = ++loc % MAXSIZE;
 h[loc] = key;
} /* end of hashlp function */

/* function hashlpsr find a location for a key  */
void hashlpsr(int key, int h[MAXSIZE])
{
 int loc;
 loc = key % MAXSIZE;
 while ((h[loc] ! = key) && (h[loc] ! = –1))
      loc = ++loc % MAXSIZE;
 if (h[loc] ! = –1)
     printf("\n Search successful at index %d",loc);
 else
     printf("\n Search unsuccessful");
} /* end of hashlpsr function */

void display(int h[MAXSIZE])
{
int i;
printf("\n List of keys, –1 indicate that the location is empty \n");
for (i = 0;i<MAXSIZE;i++)
     printf("%d",h[i]);
} /* end of display function */
```

Linear probing is very easy to implement but it suffers from a problem of clustering. After several insertion and deletion there is a clustering. If the hash addresses 2, 3, 4, 5, and 6 are filled then the key with these addresses will be stored at 7. Here a cluster means a block of occupied addresses and primary clustering refers to many such blocks separated by free locations.

Therefore, once clusters are formed there are more chances that subsequent insertions will also end up in one of the cluster and may increase the size of cluster. Thus, increasing the number of probes is required to find a free location.

To avoid problem of clustering, the two other methods are quadratic probing and double hashing.

Quadratic Probing

Suppose for key k has the hash address h(k) = loc. Then, in case of collision instead of searching the locations with addresses loc, loc + 1, loc + 2, loc+ 3, we linearly search the locations with addresses

loc, loc + 1, loc + 4, loc + 9, loc + 16,… loc + i^2,..

up to one complete cycle from the collision point.

The quadratic probing uses the following hash function:

h(k, i) = (h'(k) + i^2) mod nfor i = 0, 1, 2, …

Example 1. Let us insert the keys 99, 33, 23, 44, 56, 43, 19 into a hash table of size n =10 and hash function is division method i.e h(k) = k mod 10. The quadratic probing hash function is calculated as below. The keys are stored at location that is underlined.

k	99	33	23	44	56	43	19
h(k,0)	<u>9</u>	<u>3</u>	3	4	<u>6</u>	3	9
h(k,1)	-	-	<u>4</u>	<u>5</u>	-	4	<u>0</u>
h(k,2)	-	-	-	-	-	<u>7</u>	-

Now several keys map to the memory locations 3 and 9 so there is collision.

(*i*)

0	1	2	3	4	5	6	7	8	9
			33	23					99

For probe 0, insert key 23 at location 3 but there is collision. For probe 1 the address is 4, now insert the key.

(*ii*)

0	1	2	3	4	5	6	7	8	9
			33	23	44				99

For probe 0, insert key 44 at location 4 but there is collision. For probe 1 the address is 5, now insert the key.

(*iii*)

0	1	2	3	4	5	6	7	8	9
			33	23	44	56			99

For probe 0, insert key 56 at location 6 as no collision.

(*iv*)

0	1	2	3	4	5	6	7	8	9
			33	23	44	56	43		99

For probe 0, insert key 43 at location 3 but there is collision. For probe 1 the address is 4 but there is again collision. For probe 2 the address is 7, now insert the key as this location is free.

(*v*)

0	1	2	3	4	5	6	7	8	9
19			33	23	44	56	43		99

For probe 0, insert key 19 at location 9 but there is collision. For probe 1 the address is 0, now insert the key as this location is free.

A complete 'C' program to implement quadratic probing is given below:

```
/* Program hashqp.c inserts keys into an array with quadratic probing
technique of collsion resolution technique. The maximum number of keys
can be inserted is equal to MAXSIZE. The function hashqp inserts the key
into hash value using division method. */
```

SEARCH/HASHQP.C

```
#include<stdio.h>
#define MAXSIZE 10
int h[MAXSIZE];
void hashqp(int , int[]);
void hashqpsr(int key, int h[MAXSIZE]);
void display(int h[MAXSIZE]);
void main()
{
int i, key,ch ;
for(i = 0;i<MAXSIZE;i++)
    h[i] = –1;
do{
    printf("\n\n Program for insertion/searching keys with linear probing");
    printf("\n 1. Insert keys");
    printf("\n 2. Search key");
    printf("\n 3. Display keys");
    printf("\n 4. Exit");
    printf("\n Select operation");
    scanf("%d",&ch);
    switch(ch)
    {
    case 1: do{
                printf("\n Enter key value [type – 1 for termination]");
                    scanf("%d",&key);
                    if (key ! = –1)
                            hashqp(key,h);
                    }while(key! = –1);
                    display(h);
                    break;
    case 2:   printf("\n Enter search key value");
```

```
                    scanf("%d",&key);
                    hashqpsr(key,h);
                    break;
    case 3: display(h);
                break;
    case 4: break;
    }
    }while(ch! = 4);
} /* end of main function */

/* function hashqp find a location for key and insert it */
void hashqp(int key, int h[MAXSIZE])
{
 int loc, i = 1;
 loc = key % MAXSIZE;
 while (h[loc] ! = –1)
      {
       loc = (key % MAXSIZE + i*i) % MAXSIZE;
       i++;
       }
 h[loc] = key;
} /* end of hashqp function */

/* function hashqpsr find a location for a key  */
void hashqpsr(int key, int h[MAXSIZE])
{
 int loc;
 loc = key % MAXSIZE;
 while ((h[loc] ! = key) && (h[loc] != –1))
       loc = ++loc % MAXSIZE;
 if (h[loc] ! = –1)
      printf("\n Search successful at index %d",loc);
 else
      printf("\n Search unsuccessful");
} /* end of hashqpsr function */

void display(int h[MAXSIZE])
{
int i;
printf("\n List of keys, –1 indicate that the location is empty \n");
for (i = 0;i<MAXSIZE;i++)
      printf("%d",h[i]);
} /* end of display function */
```

Quadratic probing also suffers milder form of clustering called secondary clustering.

Double Hashing

Double hashing is one of the best methods available for open addressing. Double hashing uses a function of the form

$h(k, i) = (h_1(k) + i*h_2(k)) \bmod n$ for i = 0, 1, 2, … n–1

where n is the size of the hash table, $h_1(k) = k \bmod m$ and $h_2(k) = k \bmod n$ are two auxiliary hash functions. Here n is chosen lesser than n (i.e. n–1, or n–2).

Therefore, for a given key k, the first address is loc and the successive probes are offset from previous locations by the amount $h_2(k)$ modulo n.

Double hashing represents an improvement over linear or quadratic probing as $O(n^2)$ probe sequence are used rather than O(n), since each possible pair $<h_1(k), h_2(k)>$ yields a distinct probe sequence.

Example 1. Let us insert the keys 99, 33, 23, 44, 56, 43, 19 into a hash table of size n =10 and first hash function is division method i.e., $h_1(k) = k \bmod 10$ and second hash function is $h_2(k) = k \bmod 9$. The double probing hash function as calculated as below. The keys are stored at location that is underlined.

k	99	33	23	44	56	43	19
$h(k,0) = h_1(k)$	9	3	3	4	6	3	9
$h(k,1) = (h_1(k) + h_2(k)) \bmod n$	-	-	8	-	-	0	0
$h(k,2) = (h_1(k) + 2*h_2(k)) \bmod n$	-	-	-	-	-	-	1

Now several keys map to the memory locations 3 and 9 so there is collisions.

(*i*)

0	1	2	3	4	5	6	7	8	9
			33	44		56			99

For probe 0, keys 99, 33, 44 and 56 are inserted.

0	1	2	3	4	5	6	7	8	9
43			33	44		56		23	99

(*ii*) For probe 1, keys 23 and 43 are inserted.

0	1	2	3	4	5	6	7	8	9
43	19		33	44		56		23	99

(*iii*) For probe 2, key 19 is inserted.

A complete 'C' program to implement double hashing is given below:

```
/* Program hashdh.c inserts keys into an array with double hashing
technique of collsion resolution technique. The maximum number of keys
can be inserted is equal to MAXSIZE. The function hashdh inserts the key
into hash value using two hash functions h1(key) and h2(key)
of division method. */

#include<stdio.h>
#define MAXSIZE 10
```

SEARCH/HASHDH.C

```
int h[MAXSIZE];
void hashdh(int , int[]);
void hashdhsr(int key, int h[MAXSIZE]);
void display(int h[MAXSIZE]);
void main()
{
int i, key,ch ;
for(i = 0;i<MAXSIZE;i++)
     h[i] = –1;
do{

     printf("\n\n Program for insertion/searching keys with double hashing");
     printf("\n 1. Insert keys");
     printf("\n 2. Search key");
     printf("\n 3. Display keys");
     printf("\n 4. Exit");
     printf("\n Select operation");
     scanf("%d",&ch);
     switch(ch)
     {
     case 1: do{
                printf("\n Enter key value [type–1 for termination]");
                     scanf("%d",&key);
                     if (key ! = –1)

                          hashdh(key,h);
                     }while(key! = –1);
                     break;
     case 2:   printf("\n Enter search key value");
                     scanf("%d",&key);
                     hashdhsr(key,h);
                     break;
     case 3: display(h);

                break;
     case 4: break;
     }
     }while(ch! = 4);
}

/* function hashdh find the location for a key and insert it */
void hashdh(int key, int h[MAXSIZE])
{
 int loc, i = 0;
```

```
 loc = key % MAXSIZE;
 while (h[loc] ! = –1)
       loc = (loc + ++i*(key % (MAXSIZE –1))) % MAXSIZE;
 h[loc] = key;
 display(h);
} /* end of function */

/* function hashdhsr find the location for a key */
void hashdhsr(int key, int h[MAXSIZE])
{
 int loc,i = 0;
 loc = key % MAXSIZE;
 while ((h[loc] ! = key) && (h[loc] ! = –1))
       loc = (loc + ++i*(key % (MAXSIZE–1))) % MAXSIZE;
 if (h[loc] != –1)
      printf("\n Search successful at index %d",loc);
 else
      printf("\n Search unsuccessful");
} /* end of hashdhsr function */

void display(int h[MAXSIZE])
{
int i;
printf("\n List of keys, –1 indicate that the location is empty \n");
for (i = 0;i<MAXSIZE;i++)
      printf("%d",h[i]);
}/* end of display function */
```

2. Collision Resolution by Separate Chaining

In this scheme, all the records whose keys hash to the same location are put in a linked list. The location i in the hash-table contains a pointer to the head of the linked list of all the records that hashes to value i. If there is no such record that hash to location i, the location i contains NULL value.

Suppose that hash function produces values between 0 to MAXSIZE–1. Then an array bucket of header nodes of size MAXSIZE is declared. The array bucket[i] points to the list of all records whose keys hash into i.

In searching for a record, the list head bucket[i] is accessed and the list that it initiates is traversed. If the record is not found, it is inserted at the end of the list.

For programming, we can use the linked list implementation with pointer. Here we have declared with array of pointer to structure.

```
#define MAXSIZE 10
struct chain{
int keys;
struct chain *next;
```

}*bucket[MAXSIZE];

The linked list with bucket [0] consists of all keys with their hash value 0. The linked list with bucket [1] consists of all keys with their hash value 1. The linked list with bucket [2] consists of all keys with their hash value 2 and so on.

The average number of probes, using chaining, for a successful search and for an unsuccessful search are known to be the following approximate values:

$S(p) \cong 1 + p/2$ and $U(p) \cong e^{-p} + p$

Here load factor p =n/m where n is number of memory locations in hash and m is number of keys to be mapped.

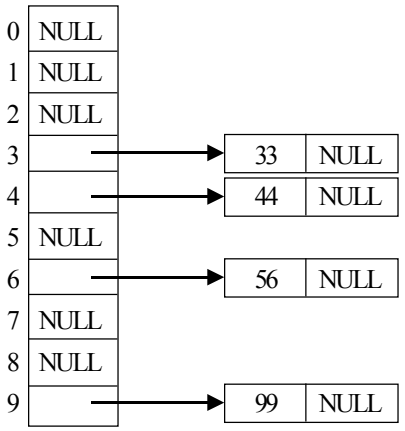
Example 1. Let us insert the keys 99, 33, 23, 44, 56, 43, 9 into a hash table of size n =10. The hash function h(k) = k mod 10 is used.

The keys are stored linked list bucket[] .

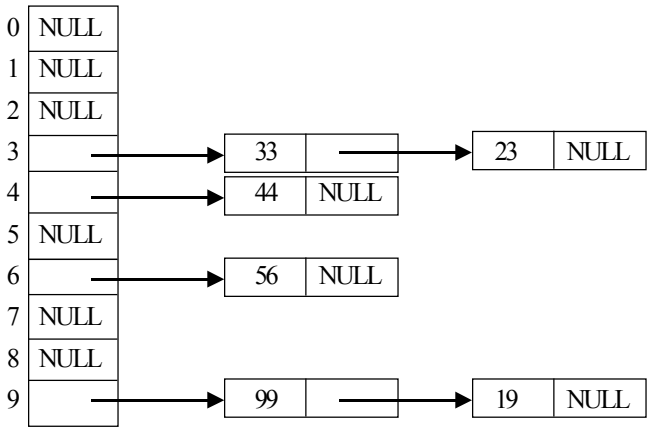
k	99	33	23	44	56	43	19
h(k)	9	3	3	4	6	3	9

Initially chained hash table bucket is empty. The Fig. 10.2 shows the keys insertion in separate chaining

(*i*) In probe 0, keys 99, 33, 44 and 56 are inserted in their respective list.



(*ii*) In probe 1, the keys 23 and 19 are inserted.



(iii) In probe 2, the key 43 is inserted.

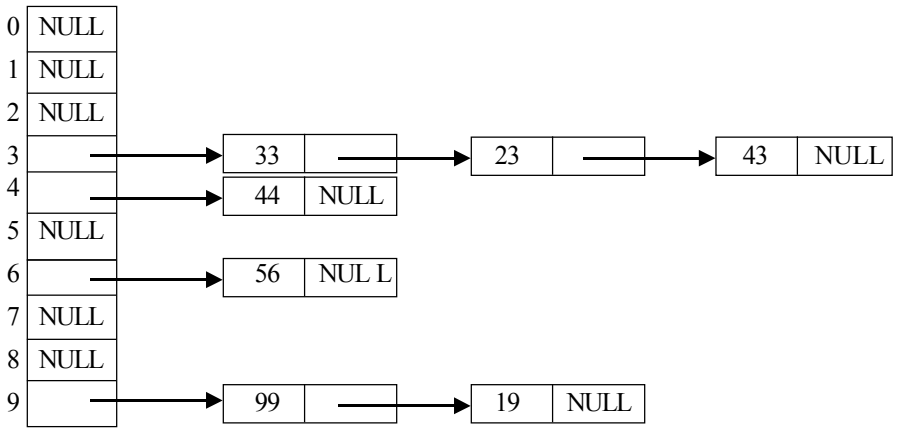


Figure: 10.2 *chaining implemented as linked list.*

The complete 'C' program to implement the collision resolution with separate chaining is given below. The program used the createlist function to implement hash function and function display the buckets elements.

/* Program hashscr.c implements the collision resolution with separate chaining the help of linked list. The function createlist creates MAXSIZE number of separate buckets using division method hash function */


```
#include<stdio.h>
#include<malloc.h>
#define MAXSIZE 10
struct chain{
    int key;
    struct chain *next;
}*ptr,*bucket[MAXSIZE],*newnode;

int item = 0;
void createlist(struct chain *bucket[MAXSIZE]);
void display(struct chain *bucket[MAXSIZE]);

void main()
{
    int ch,i,blk;
    for(i = 0;i<MAXSIZE;i++)
            bucket[i] = NULL; /* Initially list is empty */
    createlist(bucket);
    display(bucket);
    getch( );

}/* end of main function */
```

```
void createlist(struct chain *bucket[MAXSIZE])
{
int h;
     printf("Enter keys and type –1 for termination\n");
     while(item! = –1)
     {
               scanf("%d",&item);
               if (item ! = –1)
               {
                 h = item % MAXSIZE;
                 printf("h = %d",h);
                 if (bucket[h] == NULL)
                          {
                          ptr = (struct chain *)malloc(sizeof(struct chain));
                          bucket[h] = ptr;  /* bucket[h] now points to the first
                                                                   node of the list*/
                          ptr–>key = item;
                          ptr–>next = NULL;
                          }
                           else
                          {
                          ptr = bucket[h];
                          while(ptr–>next! = NULL)
                                   ptr = ptr->next;
                        newnode = (struct chain *)malloc(sizeof(struct chain));
                          newnode–>key = item;
                          newnode–>next = NULL;
                          ptr–>next = newnode;
                          } /* End of Else structure */
                 } /* End of If structure */
     } /* end of outer while */
} /* End of function createlist */

void display(struct chain *bucket[MAXSIZE])
{
 int i;
 printf("\n List of keys with separate chaning");
 for(i = 0;i<MAXSIZE;i++)
 {
     ptr = bucket[i];
     if (ptr)
               printf("\n List of keys in bucket %d–>",i);
     else
```

```
            printf("\n Bucket %d is Empty",i);
    while(ptr! = NULL)
    {
            printf("%d", ptr–>key);
            ptr = ptr->next;
    }
}
} /* end of display function */
```

Selecting Good Hash Function

(*i*) One way to minimize collisions is to use data structures that has more memory location than is actually needed for the number of elements, in order to increase the range of the hash function. However, allocating an array that contains a large number of empty locations wastes lots of memory.

(*ii*) To design a new hash function to minimize the collisions. The goal is to distribute the elements as uniformly as possible throughout the hash table.

10.4 DYNAMIC MEMORY ALLOCATION

Two fundamental concerns in the design of a memory allocation are speed of allocation and efficient utilization of memory. In dynamic memory allocation, the request of variable size blocks of memory by the programs. Each time that a request is made for memory, a free area large enough to accommodate the size requested must be allocated. The most common method for keeping track of the free blocks is to use a linear linked list.

Each free block contains a field containing the size of the block and a field containing a pointer to next free block. A global pointer freeblk points to the first free block on this list. Figure 10.3 illustrates the free block list in main memory.

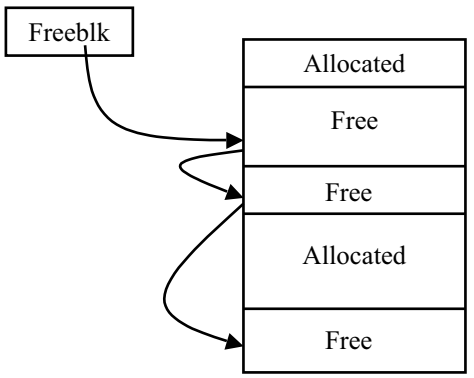


Figure 10.3 *Allocation status of memory (i.e. form of heap)*

There are several methods of selecting the free block to use when requesting memory:

- First fit
- Best fit
- Worst fit

In the **first fit** method the free list is traversed sequentially to find the first free block whose size is larger than or equal to the amount of memory requested. Once the block is found, it is removed from the

list (if it is equal to the amount requested) or is split into two partitions (if it is greater than the amount requested). The first portion remains on the list and the second is allocated.

The following first fit allocation function returns the address of a free block of memory of size n in the variable alloc and set null if no such block is available.

```
    void firstfit(int n)
    {
    p = freeblk;        /* assign the first block address from free list to variable pointer p */
    alloc = NULL:
    q = NULL;
    while( p ! = NULL) && (p–>size <n)
    {
            q = p;
            p = p–>next;
    } /* end of while */
    if (p! = NULL)              /*  check for a block large enough */
    {
            s = p–>size;
alloc = p + s – n;              /* alloc contain the address of the desired block */

if (s == n)             /* remove the block from the free list */
   if (q == NULL)
    freeblk = p–>next;
    else
         q–>next  = p->next;
else                    /* adjust the size of the remaining free block */
   p->size = s – n;
     } /* end if */
} /* end function */
```

The **best fit** method obtains the smallest free block whose size is greater than or equal to n. A function bestfit to obtain such a block by traversing the entire free list as follows:

```
void bestfit(int n)
    {
    struct nodetype *p, *q, *r, *rq;
    int s, rsize = 32676;
    p = freeblk;        /* assign the first block address from free list to variable pointer p */
    alloc = NULL;
    q = NULL;                   /* q is one block behing p*/
    r = NULL;                   /* r points to the desired block */
    rq = NULL;                  /* rq is one block behind r */
    while (p != NULL)
```

```
    {
            if ((p–>size > = n) &&(p–>size <rsize))
            {
                    r = p;
                    rq = q;
                    rsize = p->size;
            } /* end if */
            q = p;
            p = p –>next;
    } /* end of while */
    if (r! = NULL)              /*  check for a block large enough */
    {
– alloc = r + rsize –n; /* alloc contain the address of the desired block */
     if (rsize == n)            /* remove the block from the free list */
            if (rq == NULL)
                    freeblk = r–>next;
            else
                    rq–>next = r–>next;
      else                      /* adjust the size of the remaining free
                                        block */
            r–>size = rsize – n;
      } /* end if */
} /* end function */
```

The another method of allocating blocks of memory is the **worst fit** method. In this method the system always allocates a portion of the largest free block whose size is greater than or equal to n.

The only change in worst fit from best fit method is that it allocates the largest free block to fit the requested block. The variable **rsize** initialize to 0 and the first if condition in the outer while loop is changed to **p–>size >rsize** from **p–>size <rsize**.

The **worst fit function** is as given below:

```
void worstfit(int n)
    {
    struct nodetype *p, *q, *r, *rq;
    int s, rsize = 0;
    p = freeblk;        /* assign the first block address from free
list to variable pointer p */
    alloc = NULL;
    q = NULL;                   /* q is one block behing p*/
    r = NULL;                   /* r points to the desired block */
    rq = NULL;                  /* rq is one block behind r */
    while (p ! = NULL)
    {
```

```
                if ((p–>size > = n) &&(p–>size <= rsize))
                {
                        r = p;
                        rq = q;
                        rsize = p–>size;
                } /* end if */

                q = p;
                p = p–>next;
        } /* end of while */
        if (r!= NULL)                   /*  check for a block large enough */
        {
alloc = r + rsize –n;   /* alloc contain the address of the desired block */
         if (rsize == n)                /* remove the block from the free list */
                 if (rq == NULL)
                        freeblk = r–>next;
                 else
                         rq–>next  = r–>next;
          else                          /* adjust the size of the remaining free
block */
                r–>size = rsize – n;
         } /* end if */
} /* end function */
```

For example, if memory consists initially of blocks of sizes 200, 300, and 100, the sequence of requests 150, 100, 125, 100, 100 can be satisfied by the worst fit method but not by either the first fit or best fit methods. The below figure 10.4 explains the allocation by each method.

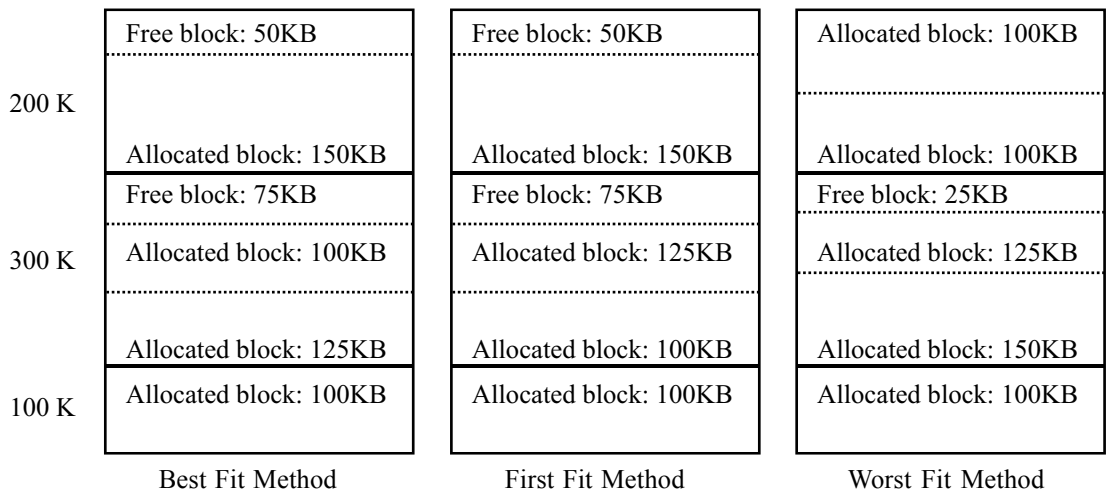


Figure 10.4 *Dynamic contiguous memory allocation*

Both best fit and worst fit allocation methods can't allocate memory to last process of 100KB, but in worst fit allocation it is done. Here first fit and best fit suffer from internal fragmentation. Even though they have two free blocks of size 75 and 50KB but noncontiguous. Therefore, one task of 100KB can't be allocated.

A complete 'C' program to implement dynamic memory allocation is given below:

/* Program freelist.cpp implements the dynamic contiguous memory allocation scheme first fit, best fit and worst fit with the help of linked list. The list of free blocks is maintained and allocated using linked list. */


```
#include<stdio.h>
#include<malloc.h>
struct  nodetype{
     int size;
     struct  nodetype  *next;
}*ptr,*freeblk,*newnode, *alloc;

int item = 0;
struct  nodetype  *createlist(struct  nodetype  *);
void firstfit(int);
void bestfit(int);
void worstfit(int);
void  display(struct  nodetype  *);

void main()
{
     int ch,blk;
     freeblk–>next = NULL;      /* Initially list is empty */
     printf("\n Program  for dynamic memory allocation");
     do{
               printf("\n 1. Create a Free Memory Blocks List");
               printf("\n 2. First fit");
               printf("\n 3. Best fit");
               printf("\n 4. Worst fit");
               printf("\n 5. List of free blocks");
               printf("\n 6. Exit");
               printf("\n  Enter your choice");
               scanf("%d",&ch);
               switch(ch)
               {
               case 1: freeblk = createlist(freeblk);
                                 display(freeblk);
                                 break;
case 2: printf("\n Enter list of memory blocks request [type –1 for termination]");
                                 do{
```

```
                    scanf("%d",&blk);
                    if (blk >0)
                        firstfit(blk);
                    }while(blk!= –1);
                    display(freeblk);
                    break;
        case 3:     printf("\n Enter list of memory blocks request [type –1 for termination]");
                    do{
                    scanf("%d",&blk);
                    if (blk >0)
                        bestfit(blk);
                    }while(blk!= –1);
                    display(freeblk);
                    break;
        case 4:     printf("\n Enter list of memory blocks request [type –1 for termination]");
                    do{
                    scanf("%d",&blk);
                    if (blk >0)
                        worstfit(blk);
                    }while(blk!= –1);
                    display(freeblk);
                    break;
        case 5:     display (freeblk);
        case 6:     break;
        } /* end of switch */
    }while(ch! = 6); /*end of while */
}/* end of main function */

struct nodetype * createlist(struct nodetype *freeblk)
{
    printf("Enter block sizes and type –1 for termination\n");
    while(item!= –1)
    {
        scanf("%d",&item);
        if (item != –1)
        {
          if (freeblk == NULL)
                {
ptr = (struct nodetype *)malloc(sizeof(struct nodetype));
freeblk = ptr;  /* freeblk now points to the first node of the list*/
                    ptr–>size = item;
```

```
                    ptr–>next = NULL;
                    }
                else
                    {
                    while(ptr–>next! = NULL)
                       ptr = ptr–>next;
        newnode = (struct nodetype *)malloc(sizeof(struct nodetype));
                    newnode–>size = item;
                    newnode–>next = NULL;
                    ptr–>next = newnode;
                    } /* End of Else structure */
          } /* End of If structure */
    } /* end of outer while */
 return(freeblk);
} /* End of function createlist */

void display(struct nodetype *freeblk)
{
printf("\n List of free memory blocks");
ptr = freeblk;
while(ptr! = NULL)
{
printf("\n %d", ptr–>size);
ptr = ptr–>next;
}
}
void firstfit(int n)
     {
     struct nodetype *p, *q;
     int s;
     p = freeblk;        /* assign the first block address from free
list to variable pointer p */
     alloc = NULL;
     q = NULL;
     while ((p ! = NULL) && (p–>size <n))
     {
               q = p;
               p = p–>next;
     } /* end of while */
     if (p! = NULL)              /*  check for a block large enough */
     {
       s = p–>size;
```

```
        alloc = p + s – n;                    /* alloc contain the address of the
desired block */
      if (s == n)                    /* remove the block from the free list */
       if (q == NULL)
                freeblk = p–>next;
        else
                 q–>next  = p–>next;
      else                  /* adjust the size of the remaining free block */
                p–>size = s – n;
         } /* end if */
} /* end function */

void bestfit(int n)
      {
      struct nodetype *p, *q, *r, *rq;
      int s, rsize = 32676;
      p = freeblk;        /* assign the first block address from free
list to variable pointer p */
      alloc = NULL;
      q = NULL;                     /* q is one block behing p*/
      r = NULL;                     /* r points to the desired block */
      rq = NULL;                    /* rq is one block behind r */
      while (p != NULL)
      {
                if ((p->size >= n) &&(p->size <rsize))
                {
                          r = p;
                          rq = q;
                          rsize = p–>size;
                } /* end if */

                q = p;
                p = p–>next;
      } /* end of while */
      if (r! = NULL)                /*  check for a block large enough */
      {
alloc = r + rsize –n;   /* alloc contain the address of the desired block */
        if (rsize == n)               /* remove the block from the free list */
                 if (rq == NULL)
                          freeblk = r–>next;
                 else
                           rq–>next  = r–>next;
```

```
        else                        /* adjust the size of the remaining free
block */
                r–>size = rsize – n;
        } /* end if */
} /* end function */

void worstfit(int n)
     {
     struct nodetype *p, *q, *r, *rq;
     int s, rsize = 0;
     p = freeblk;        /* assign the first block address from free
list to variable pointer p */
     alloc = NULL;
     q = NULL;                  /* q is one block behing p*/
     r = NULL;                  /* r points to the desired block */
     rq = NULL;                 /* rq is one block behind r */
     while (p ! = NULL)
     {
               if ((p–>size > = n) &&(p–>size <=rsize))
               {
                         r = p;
                         rq = q;
                         rsize = p–>size;
               } /* end if */

               q = p;
               p = p–>next;
     } /* end of while */
     if (r! = NULL)              /*  check for a block large enough */
     {
               alloc = r + rsize – n;        /* alloc contain the address of the
desired block */
       if (rsize == n)           /* remove the block from the free list */
                if (rq == NULL)
                         freeblk = r–>next;
                else
                          rq–>next  = r–>next;
        else                        /* adjust the size of the remaining free
block */
               r–>size = rsize – n;
        } /* end if */
} /* end function */
```

Analysis of program: Program uses four functions. The function createlist create a list of elements consisting of free blocks. The function firstfit, bestfit and worstfit allocate free memory to the requested blocks according to first fit, best fit and worst fit allocation method. The function display the available list of free memory blocks.

Internal and External Fragmentation

The phenomenon in which there are many small noncontiguous free blocks is called external fragmentation because free space is wasted outside allocated blocks. This contrasts with internal fragmentation, in which free space (i.e. memory) is wasted within allocated blocks. To avoid the problem of internal and external fragmentation in the use of free lists, we merge all free memory block into a single free block, or to combine some adjoining free blocks into a free block of larger size. We describe two techniques: memory compaction and buddy system.

Memory Compaction

In this approach memory allocations are changed such that all free blocks can be united into a single free block. This is achieved by relocating throughput memory blocks to the programs and data existing in the allocated blocks of memory. The below figure explains the compaction for the best fit method as given in Fig. 10.5.

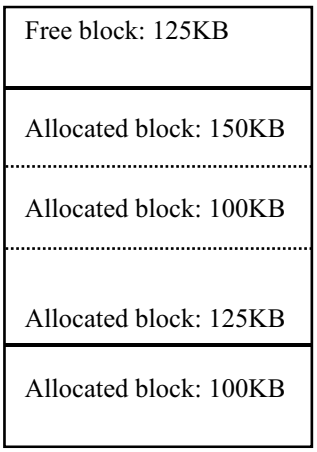


Figure 10.5 *Compaction*

After memory compaction a free block of 125KB is available and now we can allocate 100KB process to it.

Buddy System

An alternative method of handling the fragmentation problem is to keep separate free list for blocks of different sizes. Each list contains free blocks of only one specific size. Adjacent free blocks of smaller size may be removed from their lists, combined into free blocks of larger size, and placed on the larger size free list. These larger blocks can then be used intact to satisfy a request for a larger amount of memory or they can be split once more into their smaller constituent blocks to satisfy several smaller requests.

Chapter 11

File Structures

11.1 DEFINITION AND CONCEPT

We will now define the terms of the hierarchical structure of stored data collections i.e file.

(*i*) ***Field*:** It is an elementary data item characterized by its size, length and type
Example:
Name : a character type size 20
Age: : a numeric type

(*ii*) ***Record*:** It is collection of related fields. It has no identifying name.

(*iii*) ***File*:** Data is organized for storage in files. A file is collection of similar, related records. It has an identifying name.

(*iv*) ***Index*:** An index file corresponds to a data file. It's record contain a key field and a pointer to that record of the data file which has the same value of the key field.

The fundamental operations that are performed on files are:

- *Creation of a file:* A file is initially created as a structure of fields and the data is stored record wise into the empty file.
- *Updation of a file*: The contents of a file may be constantly updated. The updation includes insertion of new record occurrence, modification of existing record and deletion of existing record.
- *Retrieval from a file*: Retrieval is the extraction of meaningful information from file. This retrieval may be in the form of a query.
- *Maintenance of a file*: A file may be maintained by restructuring a file or by reorganizing a file. Restructuring a file means changing the structure of a file within the same file organization techniques i.e., changing the field width or adding new field. Rearranging of a file means changing from one file organization to another.

Files may also classified according to the function they perform in an information system.

- *Master file*: A master file represents a static view of some aspect of an organization at a particular point of time. A master file in bank may contain the details of each account number operating in the bank and details about the account holder.
- *Transaction file:* A transaction file contains information about the changes which are to be applied to a master file.
- *Program file*: A program file contains instruction for data processing. This data may be stored in other files or in the main memory.
- *Report file*: A report file contains data in the form of a report.

- *Text file*: Text file contains alphanumeric and graphics data input using a text editor program.
- *Binary file*: Binary files allow to write numeric data to the disk files in less number of bytes as compared to text files.

11.2 FILE ORGANIZATION

The concept of file organization refers to the manner in which data records are arranged on file storage medium. File organization can most simply be defined as the method of storing data records in a file and subsequent implications on the way these records can be accessed. The factors involved in selecting a particular file organization for uses are:

- Economy of storage
- Easy of retrieval
- Convenience of updates
- Reliability
- Security
- Integrity

The most common file organizations are:

1. **Sequential File.** Sequential file organization has one access key and only sequential access.
   Data records are stored in some specific sequence e.g. order of arrival, value of keyfields etc. Records of a sequential file cannot be accessed at random i.e to access the n^{th} record, one must traverse the preceding n–1 records.
2. **Relative File.** Relative file organization has one access key and support for direct access.
   Each record has a fixed place in relative file. Each record must have associated with it an integer key value that will help identify this slot. Therefore, this key will be used for *insertion and retrieval* of the record. Random as well as sequential access is possible. Relative files can exist only on random access devices like disks.
3. **Direct File.** These are similar to relative files, except that the key value need not be an integer. The user can specify keys that make sense to his application.
4. **Index Sequential File.** Index sequential file organization has one access key and no direct access and sequential access.
   An index is added to the sequential file to provide random access. An overflow area needs to be maintained to permit insertion in sequence.
5. **Indexed File.** In this file organization, no sequence is imposed on the storage of records in the data file, therefore, no overflow area is needed. However, the index is maintained in strict sequence. Multiple indexes are allowed on a file to improve access.

Sequential Files

A sequentially organized file may be stored on either a serial access or a direct access storage medium.

Structure

To provide the 'sequence' required a 'key' must be defined for the data records. Usually a field whose values can uniquely identify data records is selected as the key. If a single field cannot fulfil this criterion, then a combination of fields can serve as the key. For example in a file that keeps student records, a key could be student id.

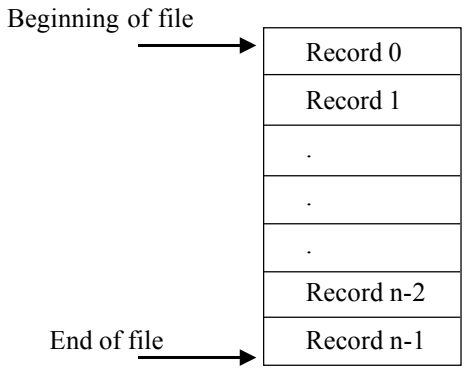


Figure 11.1 *Structure of a sequential file*

Operations

(*i*) **Insertion:** Records must be inserted at the place directed by the sequence of the key. As it is obvious, direct insertions into the main data file would lead to frequent rebuilding of the file. Reserving overflow areas in the file for insertions could mitigate this problem. But this leads to wastage of space and also the overflow areas may also be filled.

(*ii*) **Deletion:** Deletion is the reverse process of insertion. The space occupied by the record should be freed for use. Usually deletion (like insertion) is not done immediately. The concerned record (along with a marker or 'tombstone' to indicate deletion) is written to a transaction file. At the time of merging the corresponding data record will be dropped from the primary data file.

(*iii*) **Updation:** Updation is a combination of insertion and deletions. The record with the new values is inserted and the earlier version deleted. This is also done using transaction files.

(*iv*) **Retrieval:** User programs will often retrieve data for viewing prior to making decisions, therefore, it is vital that this data reflects the latest state of the data if the merging activity has not yet taken place.

Retrieval is usually done for a particular value of the key field. Before returning to the user, the data record should be merged with the transaction record (if any) for that key value.

The other two operations 'creation' and 'deletion' of files are achieved using simple programming language statements.

Advantages

- Easy to handle.
- Involve no overhead.
- Can be stored on tapes as well as disks.
- Records in a sequential file can be of varying length.

Disadvantages

- Updates are not easily accommodated.
- By definition, random access is not possible.
- All records must be structurally identical. If a new field has to be added, then every record must be rewritten to provide space for the new field.
- Continuous areas may not be possible because both the primary data file and the transaction file must be looked during merging.

Application

Sequential files are most frequently used in commercial batch oriented data processing applications where there is the concept of a master file to which details are added periodically. Example is payroll applications.

Direct File Organization

It offers an effective way to organized data when there is a need to access individual records directly.

To access a record directly (or random access) a relationship is used to translate the key value into a physical address. This is called the mapping function h

h(key value) = address.

A calculation is performed on the key value to get an address. This address calculation technique is often termed as hashing and mapping function is called hash function.

We have already discussed hashing in details in Chapter 10.

Advantages

- Records can be accessed out-of-sequence, randomly.
- Well suited for interactive application.
- Support updation operation in place.

Disadvantages

- Can only be stored on disks.
- Involved overhead in address calculation.
- Records can only be of fixed length.

Index Sequential File Organization

When there is need to access records sequentially by some key value and also to access records directly by the same key value, the collection of records may be organized in an effective manner called Index Sequential Organization.

To implement the concept of indexed sequential file organizations, we consider an approach in which the index part and data part reside on separate file. The index file has a tree structure and data file has a sequential structure. Since the data file is sequenced, it is not necessary for the index to have an entry for each record.

An indexed sequential file provides the combination of access types that are supported by a sequential file and direct file (or relative file).

When the new records are inserted in the data file, the sequence of records needs to be preserved and also the index is accordingly updated.

Two approaches used to implement indexes are static indexes and dynamic indexes.

As the main data file changes due to insertions and deletions, the static index contents may change but the structure does not change. In case of dynamic indexing approach, insertions and deletions in the main data file may lead to changes in the index structure.

There are several approaches to structuring both the index and sequential data portions of an indexed sequential file. The most common approach is to build the index as a tree of key values. The tree used is typically a variation on the B-tree, known as B^+-tree.

A B^+-tree has the same structure as B-tree except that additional set of pointers is used to link the leaf nodes in sequential order. Thus, a B^+-tree has two parts: the index set is the interior nodes and sequence set is the leaves. Thus, keys can be accessed efficiently both directly and sequentially.

Advantages

- Records can be accessed sequentially and randomly.
- Supports interactive as well as batch-oriented applications.
- Support updation operation in place.

Disadvantages

- Can only be stored on disks.
- Involved more overheads in the form of maintenance of indexes.
- Records can only be of fixed length.

11.3 FILES IN 'C'

'C' language supports standard I/O and system I/O for reading/writing a file. Standard I/O is more commonly used. The user has no control over the buffer size in this case. Standard I/O supports both binary as well as text files. The data is represented as character strings in text files. For example, the number 30000 will be written as a character string "30000" in a text file. This requires 5 bytes for its storage. But an integer requires only two bytes of storage, if it is less than 32767. It is possible to store the number using 2 bytes only if the file is opened in binary mode. So, 'C' language supports the following types of files and the I/O commands depends upon the type of file being used:

- Standard I/O text files
- Standard I/O binary files
- System I/O files

Standard I/O Text Files

As described earlier in this chapter, a file has to be opened before any I/O operation can be performed on it. 'C' files can be opened in different modes:

1. Read mode designated by 'r' opens a file in read mode and we can only read from a file.
2. Write mode designated by 'w' opens a file for writing. If a file is already existing and contains some data, it will be over written.
3. To avoid over writing, a file may be opened in append mode designated by *'a'* which allows the user to write at the end of an existing file without destroying the contents of the file.

Opening and Closing a File: The **FILE** is a data type in C and a pointer variable must be defined to point to the file. For example, the statement:

FILE *fp;

Declares a pointer variable *fp* which points to a file. A file can then be opened using fopen command. The general form of the **fopen()** statement is

<file_pointer> = fopen«(file_name>, <access_mode)»;

where the <file_name> is a string of characters forming a valid file name. A file name can have a maximum of 8 characters and an extension of three characters. And <access_mode> is one of ''w', 'a' or 'r' as just described.

fopen() opens a file and returns a pointer to a file if the operation was successful. Therefore, fopen() must be equated to a file pointer. In case the file is not opened due to some error (e.g. lack of space on the disk or the file does not already exist etc.) it returns the value NULL.

We can write the following program segment to open a file named testfile in write mode:

FILE/FILEW.C

```
/* Program filew.c to open a file for writing */

#include<stdio.h>
main()
{
     FILE *fp;
  if ((fp = fopen("\testfile.txt","w")) ==NULL)
   {
     printf("Unable to open file \n");
     printf("Program terminated \n");
     exit(0);
     }
   else
      printf("\file opened succesfully\n");
} /* end of main */
```

We may note that it is no point in continuing with the program if the file cannot be opened. So, it would be better to exit from the program and look for error. This can be done by using exit() function which is defined in header file stdlib.h. This function terminates a program. Exit(0) takes an integer argument which informs the compiler about the way the process is to be terminated. Argument 0 in exit(0) allows the program to terminate grace-fully after closing all open files and to do other house keeping work.

The file must be closed at the end of its use. The statement

```
fclose(fp);
```

closes file testfile which was opened in the above example. We may note that the function fclose () takes one argument which is a file pointer. It is also possible to close all open files with a single statement. The statement

```
fcloseall();
```

will close all open files. We may note that the function takes no arguments.

Writing to a File: Once a file is opened in write mode, we can start writing records on to the file. We may recall that if file is opened in write mode, it will destroy the existing data. A file may also be opened in append mode and then information can be written at the end of file. We can write on a file using following functions:

1. **fputc( ):** Writes a single character to a file. It takes two arguments, the character to be written and the file pointer of the file to which character is to be written. For example, the statement

   ```
   fputc(c, fp);
   ```

   writes a character c to the file *testfile* which we have already opened in 'w' mode with file pointer fp. The general format of fputc() is

   ```
   fputc«(character>, <file-pointer)»;
   ```

2. **fputs( ):** Writes a string to a file. It takes two arguments, the address of the string to be written and the file pointer of the file to which the string is to be written. For example, the statement

   ```
   fputs(line, fp);
   ```

writes a character string *line* to the file *testfile* using file pointer f*p*. The general format of fputs() is

fputs(<string>, <file-pointer>);

3. **fprintf( ):** Writes character strings and values of C variables to a file. It takes three arguments, file-pointer of the file to which information is written, format string which is to be written and the list of variables. For example, the statement

fprintf(fp,"%d%d%c\n", a, b, c);

writes a integer a, followed by a integer b followed by a single character c to the file pointed by the file pointer f*p*. The general syntax of the fprintf() is

fprintf(<file-Pointer>, "<output_format>", <Variable list>);

We may note that all the three functions described above assumes that a file has already been opened either in write mode or in append mode. Once a file is written, it may be closed so that it can be reopened in read mode to allow the user to read this information.

Reading from a File: Once a file is opened in read mode, data can be read from the file. When a file is opened, the file pointer is positioned to the start of the file. We can read from a file by using the following functions:

1. **fgetc():** Reads a single character to a file. It takes single arguments, the file pointer of the file to which character is to be read. For example, the statement

c = fgetc(c);

will read a character c from the file *testfile* which we have already opened in *'r'* mode with file pointer f*p*. The general format of fputc() is

fgetc<file-pointer>;

2. **fgets( ):** Reads a line of text from a file. It requires three arguments, the name of the character string, the maximum number of characters to be read and the file pointer to the file from which the information is to be read. For example, the statement

fgets(line, 80, fp);

reads characters from file which is pointed to by file pointer *fp* and store the characters in an array named line[]. The compiler continues to read characters from the file as long as new line character (' \n') is not encountered in the file or the number of characters read does not exceed 80. In addition, a NULL will be appended to the end of the string to convert the array line[] to a valid C string. We may note that the length of the string to be read must be at least one more than the maximum number of characters to be read.

3. **fscanf( ):** Reads characters from a file and converts the string into values of 'C' variables according to the format specified. It requires three arguments, file pointer, and format string and the list of variables. For example, the statement

fscanf(fp,"%d%d%c", x, y, z);

reads the record from file pointed to by *fp* and puts it in the variables listed. We may note that a blank space is considered to be a delimiter by function fscanf(). The general format of fscanf() is

fscanf(<file-Pointer>, "<input_format>", <variable_list>);

We may note that all the three functions described above for reading from a file assumes that a file is already open in read mode. Once a file is read, it may be closed.

Standard I/O Binary Files

A text file requires more storage space for numerical data. We can store the same number in the computer memory in 2 bytes, if the number is an integer less than 32767 and in 4 bytes if it is a real number declared as a float variable by using binary mode. But we cannot read the numbers by using MS DOS type command. We need special commands to read the contents of binary files. If a file is opened in binary mode, it can be accessed randomly.

We can open a file in binary mode by writing *"rb", "wb"* or *"ab"* in the *fopen* statement in read, write or append mode, e.g

```
fpb = fopen("\file.c", "rb");
```

open a file named file.c with file pointer fpb in the binary mode.

It is often desirable to read or write a structure. To write a structure on a file, we have to use fprint statement for every element of the structure. But 'C' provides another statement called fwrite which allows a block of record to be written onto a file with a single statement. The syntax of the fwrite() statement is as given of below

```
fwrite(<&rec>, <bytes>, <n>, <fpb>);
```

where rec is the name of the record to be written, bytes is the number of bytes required to store the record, *n* is the number of records to be written and *fpb* is the file pointer.

In order to get the number of bytes required to store the record, 'C' provides another function called *sizeof()*. Sizeof(y) returns the number of bytes required for storing the variable *y.* Using this function, we can rewrite the syntax for fwrite() as follows

```
fwrite (<& rec>, <sizeof (rec), <n>, <fpb>);
```

for example, consider the following structure definition:

```
struct  person{
char name[20];
int id;
float salary;
} man;
```

We can write one record of man structure by using the following statement:

```
fwrite(&man, sizeof(man), 1, bptr);
```

This statement assumes that file is already open in binary mode for writing using fpb by the following statement.

```
fpb =fopen("\file.c", "wb");
```

Let us write a complete 'C' program which reads the name of a person, his id and salary from the keyboard and writes the same to a binary file.

```
/* Program filebwk.c write a records of a person in binary file
from keyboard*/
#include<stdio.h>
```

FILE/FILEBWK.C

```
#include<stdlib.h>
#include<math.h>
#include<conio.h>
void main()
{
    struct  person
    {
            char name[20];
            int id;
            float salary;
    }man;
    char  ans;
    char string[20];
    FILE *fpb;
    if ((fpb = fopen("\person.txt","wb")) ==NULL)
    {
            printf("Unable to open file \n");
            printf("Program terminated \n");
            exit(0);
    }
    do
    {
    printf("\n Enter person name");
    gets(man.name);
    printf("\n Enter person id number");
    gets(string);
    man.id =atoi(string);
    printf("\n Enter person salary");
    gets(string);
    man.salary =atof(string);
    if(fwrite(&man, sizeof(man),1,fpb)==0)
    {
            printf("No record written \n");
            exit(0);
    }
    printf("\n Do you want to input more records y/n");
    ans = getch();
    }while(ans!='n');
}/* End of main */
```

The person name, id and salary are read as character strings and id is converted to integer by using atoi() function and salary is converted to real number using atof() function. The structure is then written to the file in one stroke by using fwrite() statement which writes all the fields into the file.

As this file was opened in binary mode, we need another function **fread()** to read the records of this file. The syntax of the fread ( ) statement is as given below:

fread(<&rec>, <bytes>, <n>, <fpb>);

where rec is the name of the record, bytes is the number of bytes required to store the record, *n* is the number of records to be read and *fpb* is the file pointer. This statement assumes that the file is already open in the binary mode for reading by the following statement:

fpb = fopen ("XYZ", "rb");

Let us write a complete program which reads the person name, his *id* and salary from the file named "\person.txt" which was created by us in the last program

FILE/FILEBRD.C

```
/* Program filebrd.c read a records of a person in binary file
and display*/
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<math.h>
void main()
{
    struct person
    {
            char name[20];
            int id;
            float salary;
    }man;
    char ans;
    char string[20];
    FILE *fpb;
    clrscr();
    if ((fpb = fopen(".....\person.txt","rb")) = =NULL)
    {
            printf("Unable to open file \n");
            printf("Program terminated \n");
            exit(0);
    }
    if(fread(&man, sizeof(man),1,fpb)==0)
    {
            printf("No record read \n");
            exit(0);
    }
    while(!feof(fpb))
    {
```

```
            printf("\n%s %d %f\n", man.name,man.id,man.salary);
            fread( &man, sizeof(man),1,fpb);
        }
        printf("\Press any key to return");
        getch();

    }/* End of main */
```

This program uses a function *feof* () which check whether the end of a file has been reached or not. The syntax of this function is as given below:

feof(<fpb>);

The function return a positive values if the file has reached its end otherwise it returns 0. This function is useful when a file is read sequentially from beginning to end.

A binary file may be accessed randomly by using function **offset()** and **fseek()**. When we open a file using fopen(), the file pointer points to the beginning of the file. Whenever we write a record to a file, the file pointer moves to the next record automatically. When we write again, it starts writing from the location where file pointer is pointing. Similarly when we read a record from a file, the file pointer moves to the end of the record automatically after we have read.

By using fseek() we can change the file pointer such that it points to the beginning of the record we want to read. The syntax of fseek() function is as given below:

fseek(<fpb>, <offset>, <mode>);

fseek() takes three arguments. The first is a file pointer, second is the offset which gives number of bytes from a particular place from where reading is to start and the third is the mode which gives a number which indicate the position from which offset is to be measured. The possible values of mode are 0, 1 or 2. 0 means beginning of the file, 1 means current position of the file and 2 means the end of the file. Offset is declared as long integer.

For example the statement

fseek(fpb, (n – 1) *x, 0);

access the n^{th} record from the beginning where *x* is the number of bytes required for storing each record of the file opened by file pointer fpb.

If a file is positioned past the actual end of the file and a write is performed, then the size of the file will be extended and logical gap will be placed in the file. If a read is attempted past the end of the file, an error will occur.

While performing any file operation, an error may occur. It is always good to check these errors and terminate the program gracefully in case of an error. There are two functions in C which allow us to check the error. These are ferror() and perror().

Function ferror() returns 0 if there is no error and return nonzero value if there is an error. It takes a file pointer as an argument. In case of an error give the message, close the file and exit gracefully. The other function perror() gives the error message automatically, in case of an error.

System Input/Output Files

For system level Input/Output the file has to be opened with function open() before doing any input/output operation. The syntax of open() is as given below:

open(<f_name> , <flag>)

open() takes two arguments, the first is the file name and second is a flag which represents a file characteristic. Flag can take any of the values listed below:

Flag	Meaning
O_TEXT	open a file in text mode
O_BINARY	open a file in binary mode
O_APPEND	open a file in append mode
O_RDWR	open a file in both read and write mode
O_READONLY	open a file in read only mode
O_WRONLY	open a file in write only mode
O_CREAT	create a new file for write mode

The function returns an integer known as file handle. If it returns a negative number, that means, the file has not been opened because of some error. The attribute flag may contain relation operators and (&&) and or (||). For example, the statement

handle = open ("C:\sanjay\test.c",O_BINARY && O_READONLY)

opens the file *c:\sanjay\test.c* in binary mode for reading only. Handle should be declared as an integer. The routines for system I/O are defined in header file fcntl.h. Therefore, this file must be included in the program for system I/O

To read a file opened with function open() we may use function read(). The syntax of the statement is as given below

read(<handle>, <buffer>, <buffersize>);

The function read() takes three arguments. First is the file handle returned by function open(), second is the address of an array of characters in which the characters read from the file are stored and third is the buffer size which is the maximum number of bytes desired to be read. It returns the number of bytes actually read. This tells us how full the buffer is.

We may note that in system I/O, the programmer selects the buffer size. When the buffer is full, the whole thing in the buffer is written. The programmer has to take them out of the buffer before reading

To write a file opened with function open(), we may use function write(). The syntax of the statement is as given below:

write(<handle>, <buffer>, <buffersize>);

This function takes three arguments, first is the handle, second is the address of the array used as a buffer and third is the maximum number of bytes we want to write.

Appendix

Conceptual Problem Solutions

Problem 1. Prove the following by mathematical induction.

(*i*) $\sum i^2 = n^2$

$1 \le i \le n$ where i is odd

(*ii*) $\sum i^2 = n(n+1)(2n+1)/6$

$1 \le i \le n$

Solution. (*i*) We can prove it by mathematical induction.

Given a positive number n, following procedure is outlined:

(*a*) f(1) is true, since $1 = 1^2$

(*b*) If all of f(1), f(2), …, f(n) are true, then in particular f(n) is true, so the series holds, addition 2n+1 to both sides we obtained:

$$1+3+\ldots+(2n-1)+(2n+1) = n^2+(2n+1)$$
$$= (n+1)^2$$

Which proves that f(n+1) is also true. That means sum of (n+1) odd integers is square of (n+1).

The method of prove is called a proof by mathematical induction.

(*ii*) We can it by mathematical induction.

Given a positive number n, following procedure is outlined:

(*a*) f(1) is true, since $1(1+1)(2*1+1)/6 = 1^2$

(*b*) If all of f(1), f(2), …, f(n) are true, then in particular f(n) is true, so the series holds, addition n+1 term to both sides we obtained:

$$1^2+2^2+3^2\ldots+n^2+(n+1)^2 = n(n+1)(2n+1)/6+(n+1)^2$$
$$= (n+1)(n+1+1)(2*(n+1)+1)/6$$

put m in place of (n +1).

$$= m(m+1)(2m+1)/6$$

Which proves that f(m) is also true.

The method of proof is called a proof by mathematical induction.

Problem 2. A two dimensional array x[5][6] is stored with base address 201. What is the address of x[2][4].

(*i*) In column major order (*ii*) In row major order

Solution. (*i*) If the array x[m][n] is stored in column major order in memory locations then the address of element x[i][j] would be at:

B+j*m*S+i*S = B+S*(j*m+i), where B is the base address, S is the size of the element (assume 1 byte) and m is number of rows.

= 201+1*(4*5+2)
= 201+22
= 223

(*ii*) If the array is stored in row major order in memory locations then the address of element x[i][j] would be at:

B+i*n*S+j*S = B+S*(i*n+j), where B is the base address, S is the size of the element (assume 1 byte) and n is number of columns.

= 201+1*(2*6+4)
= 201+16
= 217

Problem 3. A three dimensional array x[5][6][7] is stored with base address 400. What is the address of x[2][3][4].

(*i*) In row by row
(*ii*) In column by column

Solution. (*i*) If the array x[m][r][c] is stored row by row of each matrix in memory locations then the address of element x[i][j][k] would be:

= B+i*r*c*S+j*c*S +k*S = B+S*(i*r*c+j*c +k), where B is the base address, S is the size of the element (assume 1 byte) and r is number of rows, c is number of columns in matrix m.

= 400+1*(2*6*7+3*7+4)
= 400+109
= 509

(*ii*) If the array x[m][r][c] is stored column by column of each matrix in memory locations then the address of element x[i][j][k] would be:

= B+i*r*c*S+k*r*S +j*S = B+S*(i*r*c+k*r +j), where B is the base address, S is the size of the element (assume 1 byte) and r is number of rows, c is number of columns in matrix m.

= 400+1*(2*6*7+4*6+3)
= 400+111
= 511

Problem 4. A two-dimensional array x[–5..4, 0..5] is stored with base address 201. How many number of elements in the array x? What is the address of x[2,4]?

(*i*) In row major order
(*ii*) In column major order

Solution. The number of elements in array x = number of rows * number of columns

Number of rows in array x = upper bound – lower bound+1
= 4–(–5)+1 = 10

Number of columns in array x = upper bound – lower bound+1
= 5–0 +1 = 6

The number of elements in array x = 10 * 6 = 60

(*i*) If the array is stored row by row i.e., Row Major Order in memory locations then the address of element x[i,j] would be at:

= $B+(i-lb_2)*(ub_2- lb_2 +1)*S+(j-lb_2)*S$ (assume size S is of one byte)
= 201+(2–0) * (5–0+1) *1+(4–0)*1
= 201+12+4
= 217

(*ii*) If the array is stored column by column i.e. Column Major Order in memory locations then the address of element x[i,j] would be at:

$= B+(j-lb_2)*(ub_1 - lb_1+1)*S+(i-lb_1)*S$ (assume size S is of one byte)

$= 201+(4-0)*(4-(-5)+1)*1+(2-(-5))*1$

$= 201+40+7$

$= 248$

Problem 5. (*a*) How are 2–D arrays implemented in memory? Derive the formula for finding the location of element LOC(A[i, j]) in 2–D array, using base–address technique.

(*b*) Give the algorithms (both) for the evaluation of an expression using stacks.

(*c*) Write equivalent postfix expressions for:–

(*i*) (A+B^D)/(E–F)+G

(*ii*) A*(B+D)/ E–F*(G+H/K)

Solution. (*a*) A two–dimensional array with m rows and n columns will contain m*n elements and can be defined in the matrix form as logical representation.

	column 1	column 2	…	column n
row 1	x[0][0]	x[0][1]	…	x[0][n–1]
row 2	x[1][0]	x[1][1]	…	x[1][n–1]
. .	.	.	.	.
row m	x[m–1][0]	x[m–1][1]	…	x[m–1][n–1]

Now consider the memory mapping for two–dimensional arrays. In two-dimensional array we think of data being arranged in rows and columns. However, computer's memory is arranged as a row of cells (i.e. locations). Thus, the rectangular structure of two–dimensional array must be simulated. We first calculate the amount of storage area needed and allocate a block of contiguous memory locations of that size.

There are two ways—Row Major Order and Column Major Order—to simulate two-dimensional array in to the computer memory.

Consider a two-dimensional array x declared as:

intx[3][3]; /* x is a 9–element (i.e. 3*3) two-dimensional integer array */

The x indicate the address of first element of the array i.e. x[0][0].

This declaration required 9 contiguous memory location. The size of each element is two bytes. Thus total memory reserved by the compiler is 18 bytes for the array x.

The logical representation of the array x with 3 rows and 3 columns is as given below:

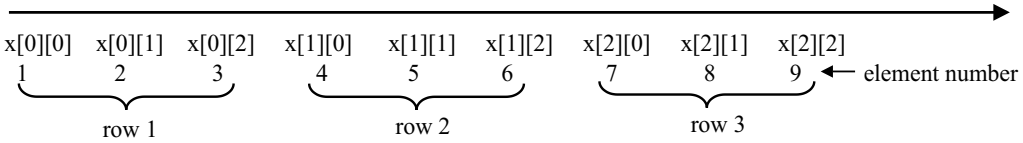
	column 1	column 2	column 3
row 1	x[0][0]	x[0][1]	x[0][2]
row 2	x[1][0]	x[1][1]	x[1][2]
row 3	x[2][0]	x[2][1]	x[2][2]

1. Row Major Order

It stores data in the memory locations row by row. That is, we store first the first row of the array, then the second row of the array and then next and so on.

We know that in computer, memory locations are in sequence, then the following memory representation stores data row by row of the array x declared above.

Contiguous memory locations



If the array store row by row i.e. Row Major Order in memory locations then the address of element x[i][j] would be at:

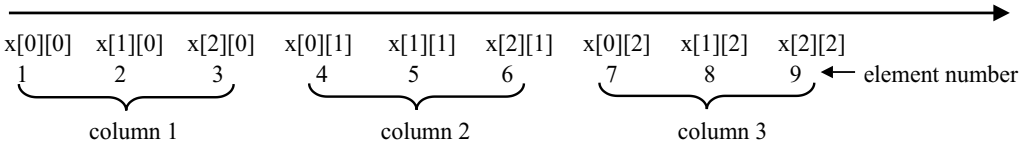
= B+i*n*S+j*S = B+S*(i*n+j) where B is the base address.

2. Column Major Order

It stores data in the memory locations column by column. That is, we store first the first column of the array, then the second column of the array and then next and so on.

We know that computer memory locations are in sequence, then the following memory representation stores data column by column of the array x declared above.

Contiguous memory locations



It is very important that, how the array is stored in memory. In both order the first element of the array is x[0][0], fifth element is x[1][1] and last element of the array is x[2][2]. Therefore, in two-dimensional array all the diagonal elements would be in the same locations in row major order and column major order.

If the array store columnwise i.e. Column Major Order in memory locations then the address of element x[i][j] would be at:

= B+j*m×s+i*S = B+S*(j×m+i)

(*b*) The following algorithm evaluates an expression in postfix using this method.

Algorithm EvalPostExp

/* Consider the postfix expression E. Opstack is an array to implement stack, variable opnd1, opnd2 contents the value of operand 1 and operand 2. The variable value consist of intermediate results. */

Step 1 : opstack = the empty stack;
/* scan the input string reading one element at a time into symb */

Step 2 : while (symb ! =')') {
symb = next input character;
if (symb is an operand)
push(opstack, symb);
else {
/* symb is an operator */
opnd2 = pop(opstack);
opndl = pop(opstack);
value = result of applying symb to opndl and opnd2;
push(opstack, value);
} /* end else */
} /* end while */

Step 3 : return(pop(opstack));

Each operand is pushed onto the operand stack opstack as it comes in input postfix expression E. Therefore, the maximum size of the stack is the number of operands that appear in the input expression.

(*c*) Convert the following infix expression into postfix and prefix expression:

(*i*) (A+B^D)/(E–F)+G

Infix expression	(A+B^D)/(E–F)+G
Convert the inner parentheses and first ^	(A+BD^)/(E–F)+G
Convert the inner parentheses	ABD^+/(E–F)+G
Convert the inner parentheses	ABD^+/EF–+G
Convert the exponent/	ABD^+EF–/+G
Convert the ADDITION+	ABD^+EF–/ G+
Postfix expression	ABD^+EF–/ G+

(*ii*) A*(B+D)/ E–F*(G+H/K)

Infix expression	A * (B+D)/E–F*(G+H/K)
Convert the inner parentheses	A * BD+ / E–F*(G+H/K)
Convert the inner parentheses and its /	A * BD+ / E–F*(G+HK/)
Convert the inner parentheses and its +	A * BD+ / E–F*GHK/+
Convert the *	ABD+* /E–F*GHK/+
Convert the /	ABD+* E/–F *GHK/+
Convert the *	ABD+* E/–FGHK/+*
Convert the –	ABD+*E/ FGHK/+*–
Prefix expression	ABD+*E/FGHK+*–

Problem 6. (*a*) Compare array representation and linked representation of data on the basis of the following points. Give explanation to each:

(*i*) Time to access an element

(*ii*) Efficient use of memory

(*iii*) Deletion of a data item

(*iv*) Adding a new data item

(*b*) How is an array stored in main memory? Discuss. Consider the following 'C' statement:

int a[10][20];

Assume a = 0 x 2000, size of each word = 4 bytes and row major form of storage; compute address of a[5][6].

Solution (*a*)

1. Memory storage space is wasted, as the memory remains allocated to the array throughout the program execution, even few nodes are stored. Because additional nodes are still reserved and their storage can't be used for any purpose.
2. List can not grow in its size beyond the size of the declared array if required during program execution.
3. The size of array can't be changed after it's declaration.
4. Arrays are called dense lists and linked list are said to be static data structures.
   (*i*) *Time to access an element*: The array directly access the required element. If we want access 6^{th} element in the single dimension array x, then x[5] will give the result. In case of linked list we need to traverse the 6 nodes then only get the result.

(*ii*) *Efficient use of memory*: In array memory storage space is wasted, as the memory remains allocated to the array throughout the program execution, even few nodes are stored. Because additional nodes are still reserved and their storage can't be used for any purpose. In linked list as and when nodes are created their memory is allocated.

(*iii*) *Deletion of item*: In array it require almost O(n) moves where as in linked list it required only few pointer movements.

(*iv*) *Adding a new data item*: In array addition of new item required movement of O(n) items whereas in linked list it require only few pointer movements. No need to actual shift of elements or nodes.

(*b*) If the array is stored row by row i.e. Row Major Order in memory locations then the address of element a[i][j] would be at:

= B+i*n*S+j*S = B+S*(i*n+j) where B is the base address, and size of element is 2 byte as element is integer.

Given the number of rows in the array e.g. m = 10

and the number of columns in the array e.g. n= 20

and base address is a = 0x2000, size of each word = 4 bytes and row major form of storage; compute address of a[5][6]. The index location of a[5][6] is

= 2(5*20+6)

= 212 bytes

Number of words required = 212/4 = 53 words

So the address of a[5][6] is = 0x2000+53

= 0x2053

Problem 7. Discuss the advantages, if any, of a two-way list over a one way list for each of the following operations:

(*i*) Traversing the list to process each node

(*ii*) Deleting a node whose location is given

(*iii*) Searching an unsorted list for a given element item

(*iv*) Searching a sorted list for a given element item

(*v*) Inserting a node before the node with a given location

(*vi*) Inserting a node after the node with a given location

Solution (*i*) There is no advantage except two-way list can transverse in forward and backward direction.

(*ii*) The location of the preceding node is needed. The two-way list contains this information, whereas with a one-way list we must traverse the list.

(*iii*) There is no advantage.

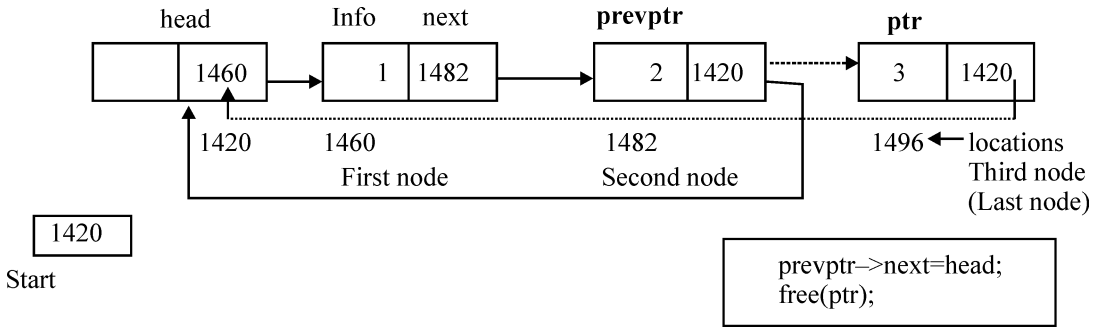
(*iv*) There is no advantage unless we know the item must appear at the end of the list, in that case we traverse the list backward.

(*v*) The location of the preceding node is needed. The two-way list contains this information, whereas with a one-way list we must traverse the list. So it is advantages.

(*vi*) There is no advantage.

Problem 8. Suppose list is a header circular list in memory. Write an algorithm which deletes the last node from the list.

Solution. The below figure explains the deletion of last node in header circular linked list:



Algorithm DELLNHCLST(head)
/* The algorithm DELLNHCLST deletes the last node from the header circular linked list. The head is pointer variable which is head of the list. */

Step 1 : if (head–>next = NULL) then /* List is empty */
write(" Underflow");
exit
end if

Step 2 : ptr = head–>next;
prevptr = NULL;

Step 3 : while (ptr–>next ! = head)/*Traverses list seeking the last node */
prevptr = ptr;
ptr = ptr–>next;

Step 4 : if (prevptr = NULL) then
head–>next = NULL;
else
prevptr–>next = head;
end if

Step 5 : free (ptr);

Step 6 : end DELLNHCLST

Problem 9. (*a*) What are header linked lists? What are the different types of header linked lists, where are they usually used?
(*b*) What are Deques and priority queues? What is their use?
(*c*) Give algorithms for insertion and deletion in a doubly linked list.

Solution. (*a*) Header Linked List

A header linked list is a linked list which always contains s special node, called the header node, at the beginning of the list. The following are two kinds of widely used header lists:

1. A grounded linear header list is a header list where the last node contains null pointer (see, Fig. 1).
2. A circular linear header list is a header list where the last node points back to the header node (see Fig. 2).
3. A grounded doubly header list is a header list where the last node contains null pointer (see Fig. 3).

4. A circular doubly header list is a header list where the last node points back to the header node (see Fig. 4).

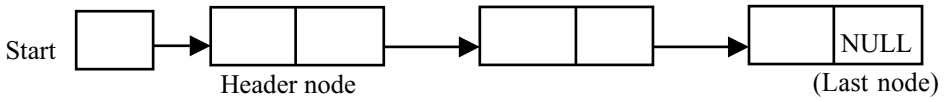


Figure 1 *A grounded linear list with header node*

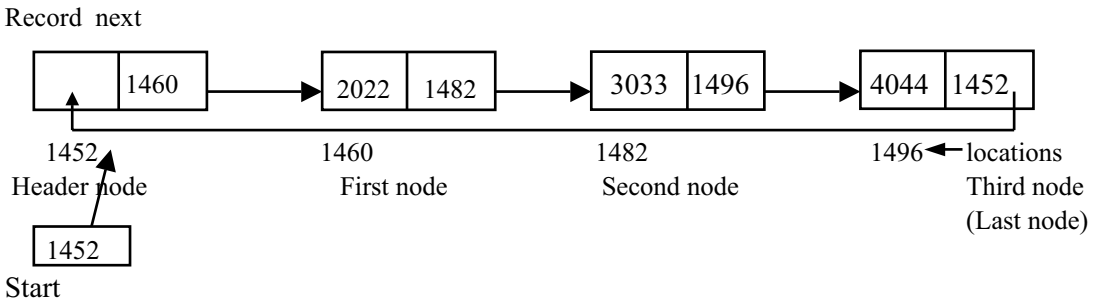


Figure 2 *A Circular linked linear list with header node*

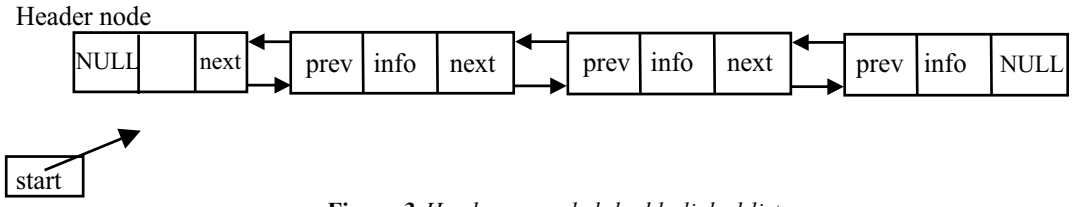


Figure 3 *Header grounded doubly linked list*

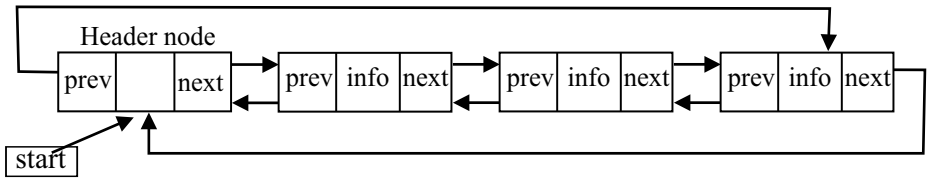


Figure 4 *Header circular doubly linked list*

By using the head node, we can realize a certain degree of symmetry in the structure by making list circular. Note that the next link of the right most node contains the address of the head node and the prev link of the head node points to the right most node. The empty list is represented by both next and prev links of the head node point to itself in case of list is doubly linked list.

(*b*) Consider the problem of inserting a node into a doubly linear linked list. There are number of cases possible.

1. Inserting a node when list is empty
2. Inserting a node after the right-most node of the list (i.e. as last node)
3. Inserting a node before the left-most node of the list (i.e. as first node)
4. Inserting a node to the left of a given node (i.e. before a node)

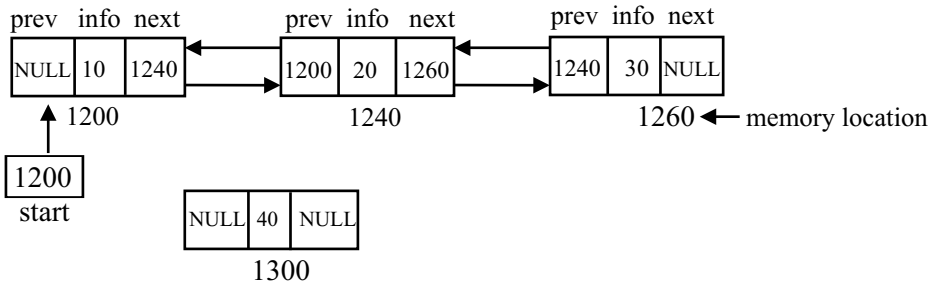
5. Inserting a node to the right of a given node (i.e. after a node)
6. Inserting a node as a node number of the list

A general algorithm for inserting a node to the left of a given node in a doubly linear linked list is as follows:

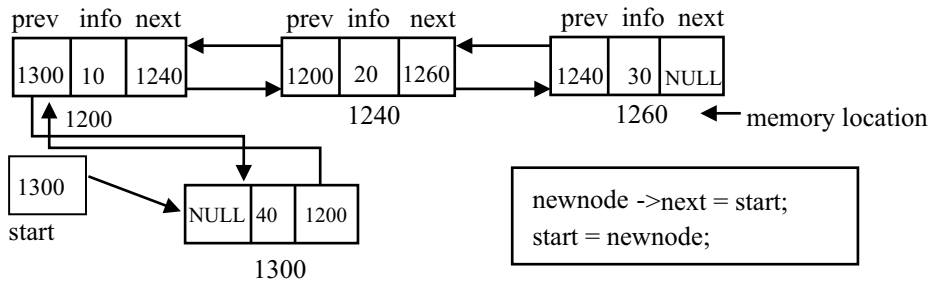
- Allocate the memory for a new node and assign data to its fields.
- If the list is empty then insert the node in the list and update left and right pointers to the list and return.
- If the node is to be inserted at the front of the list then insert the node and update the left pointer to the list and return.
- Otherwise insert the node in the middle of the list and return.

The following Fig. 5 illustrates the method of inserting node in the doubly linked list.

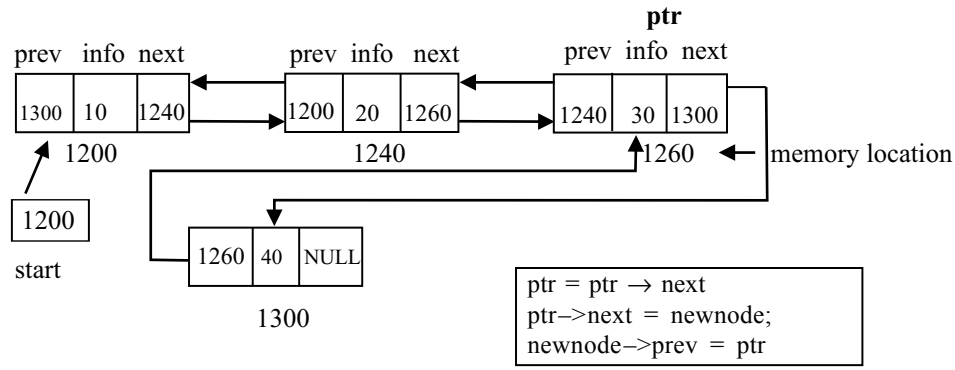
(*a*) Before insertion, memory for newnode has been created and value is assigned.



(*b*) After insertion as first node



(*c*) After insertion as last node



(*d*) After insertion: Inserted after node number 2

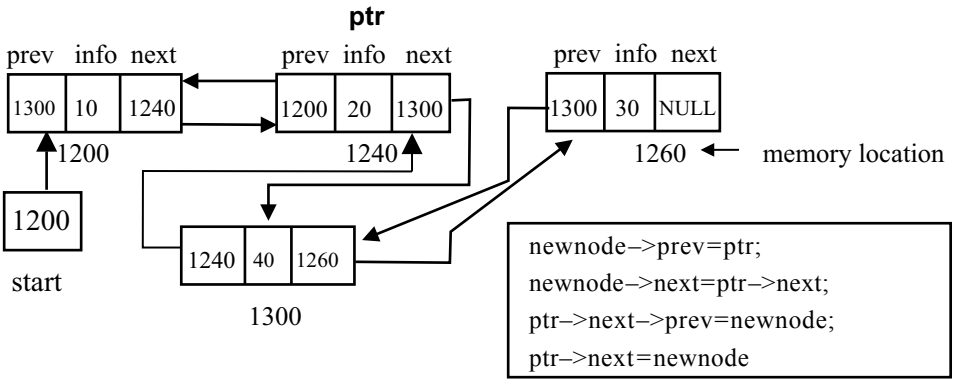


Figure 5 *Doubly linked list: insertion of newnode*

The algorithm for the insertion as first node insertf() is given below:

```
        Algorithm insertf(start, item)
Step 1 :  newnode–>info = item;
          newnode–>next = NULL;
          newnode–>prev = NULL;
Step 2 :
          newnode–>next = start;
          start = newnode;
 Step 3 : end Insertf
```

The Algorithm for the insertion as last node insertl() is given below:

```
Algorithm insertl(start, item)
/* Algorithm insertl, insert node as last node in the linked list */
Step 1 :  newnode–>info = item;
          newnode–>next = NULL;
          newnode–>prev = NULL;
Step 2 :  if (start = = NULL)then
                        start  = newnode;
step 3 :  else
                ptr = start;
Step 4 :        while (ptr–>next! = NULL)
                        ptr = ptr–>next;
                ptr–>next = newnode;
                newnode–>prev = ptr;
          /* end of else in step3 */
Step 5 : end insertl
```

Deletion in Doubly Linear Linked List

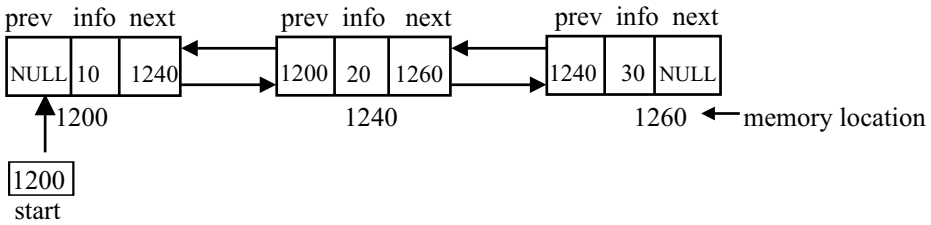
We can also delete the node in doubly linked list.

1. Deleting the right-most node of the list (i.e. as last node)
2. Deleting the left-most node of the list (i.e. as first node)

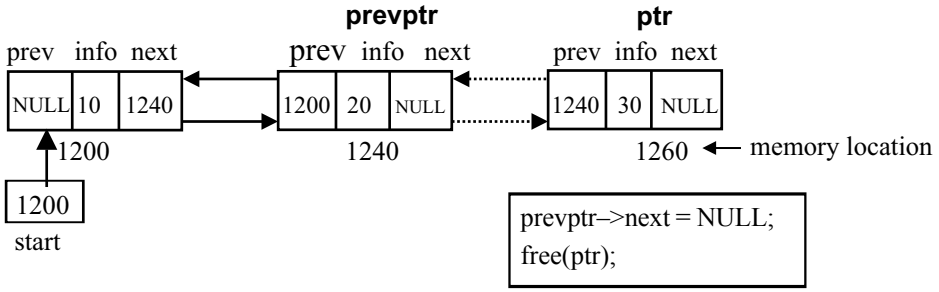
3. Deleting a node to the left of a given node (i.e. before a node)
4. Deleting a node to the right of a given node (i.e. after a node)
5. Deleting a node as a node number of the list

The following Fig. 6 illustrates the method of deleting node in the doubly linked list.

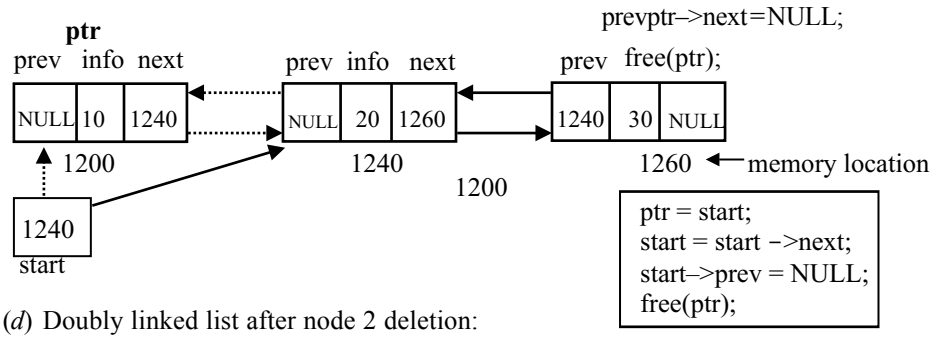
(*a*) Doubly linked list before deletion



(*b*) Doubly linked list after last node deletion:



(*c*) Doubly linked list after first node deletion.



(*d*) Doubly linked list after node 2 deletion:

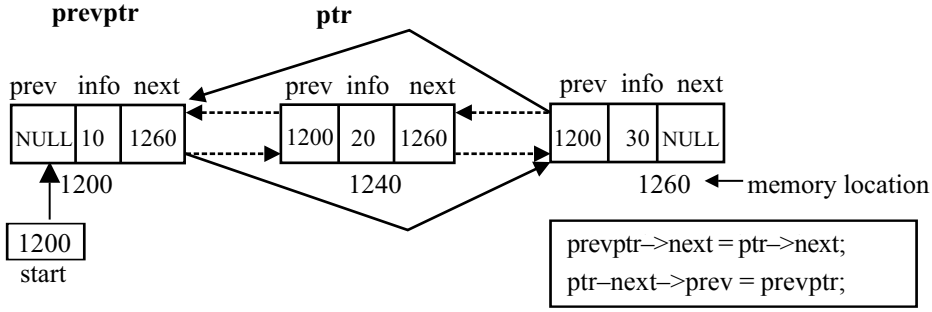


Figure 6 *Deletion in doubly linked list*

The deletef algorithm deletes the first node in the doubly linked list.

```
        Algorithm deletef(start)
Step 1 : if(start = = NULL)
                printf("list is empty");
                return;
Step 2 : ptr = start;
         start = start–>next;
         start–>prev = NULL;
         free(ptr);
Step 3 : end deletef
```

The algorithm deletel delete the last node in the doubly linked list.

```
        Algorithm deletel(start)
Step 1 : ptr = start;
Step 2 : while(ptr–>next! = NULL)
                prevptr = ptr;
                ptr = ptr–>next;
Step 3 : prevptr–>next = NULL;
         free(ptr);
Step 4 : end deletel
```

(*c*) A deque is double-ended queue, is a linear list in which elements can be added or removed at either end but not in middle. There are two variations of a deque:

1. Input-restricted deque
   This form of deque allows insertions only one end but allows deletions at both ends.
2. Output-restricted deque

This form of deque allows insertions at both ends but allows deletions at one end.

There are various ways to represent a deque in memory. We will assume our deque is maintained by a circular array deque with pointers left and right, which points to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. Figure 7 shows deque operations with their pointer positions.

(*i*) Initially deque is empty left = –1

(*ii*) A, B and C inserted in right end right = 2, left = 0

0	1	2	3	4
A	B	C		

(*iii*) Deletion from right end right = 1, left = 0

0	1	2	3	4
A	B			

(*iv*) Inserted D, E in left end right = 1, left = 3

0	1	2	3	4
A	B		E	D

(***Figure 7–contd...***)

(*v*) Inserted F in right end	right = 2, left = 3 [now queue is full]	0 1 2 3 4 A B F E D	

Figure 7 *Deque*

Priority Queue

Any data structure that supports the following operations efficiently is called priority queue.

1. Searching of an element with minimum or maximum property
2. Insertion
3. Deletion of an element with minimum or maximum property

A priority queue is a collection of elements such that each element has been assigned an explicit or implicit priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed to the order in which they were inserted to the queue.

The priority of element is decided by its value known as implicit priority and priority number is assigned with each element known as explicit priority.

The application of priority queue in timesharing operating system: programs of higher priority are processed first, and programs with the same priority form a standard queue.

There are two types of priority queues: an ascending priority queue and a descending priority queue.

An **ascending priority queue** is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.

A **descending priority queue** is a collection of items into which items can be inserted arbitrarily and from which only the largest item can be removed.

The use of priority queues as heap implementation is in sorting.

Problem 10. (*a*) What is the advantage of circular queue over a linear queue? Write the algorithm to delete a data item into a circular queue.

(*b*) An input restricted deque is a queue in which deletions can be made from either and but insertions are permitted from one end only.
Develop an algorithm for
(*i*) Addition; and
(*ii*) Deletion from specified end.

Solution. (*a*) The limitation of linear queue is that if the last position (i.e., MAXSIZE –1) is occupied, it is not possible to insert an element in queue even though some locations are vacant at front end of the queue. However, this limitation can be overcome if we consider the next index after MAXSIZE–1 is 0. The resulting queue is known as circular queue.

Algorithm queuedel(q)

Step 1 : /* check for queue empty when front pointer is at index –1. */
if (q–>front = = –1) then

```
                write("Queue is empty");
        end if
Step 2 : x = q–>items[q–>front];
 /* check for front equal to rear means queue has become empty and reinitiate front and rear to –1.
Otherwise front increases by 1 in modulus way */
Step 3 : if (q–>front = q–>rear) then
                q–>front = –1; q–>rear = –1;
        else
                q–>front = (q–>front +1) % MAXSIZE;
        end if
Step 4 : return x;
```

(*b*) Input restricted deque

The operation of inserting an item into a deque is called insert operation. We can insert element into right end of the queue that represented by right variable.

Insertion in Right end of the queue: The right pointer has changed its value according to the following conditions:

(*a*) If left = right+1, then queue is full and insertion can't take place.

(*b*) If left pointer is –1, then the first element is inserting in the array deque and assigned value of both left and right pointer to 0.

(*c*)If right is at index MAXSIZE –1, then assigned right = 0 otherwise increment right by one i.e., right = right+1.

```
        Algorithm insdeque(int x)
Step 1 : /* check for deque full */
                if (left = right+1) then
                        write("Queue full");
                        return;
Step 2 :                else
                if (left = –1) then
                 left = 0; right = 0; deque[left] = x; return;
Step 3 :         else
                        right = (right+1) % MAXSIZE;
                        deque[right] = x;
                 end if
        end if /* end of step 1 if */
Step 4 :  end insdeque
```

The operation of deleting an item into a deque is called delete operation. We can delete element into both end of the queue that represented by left and right variable.

1. **Deletion in Left end of the queue:** The left pointer changes its value according to the following conditions.

   (*a*) If left = –1, then queue is empty and deletion can't take place.

   (*b*) If left = right, then array deque become empty and assigned value of both left and right pointer to –1.

(*c*) If left is at index MAXSIZE –1, then assigned left = 0 otherwise increment left by one i.e. left = left+1.

2. Deletion in Right end of the queue: The right pointer changes its value according to the following conditions.

(*a*) If left = –1, then queue is empty and deletion can't take place.

(*b*) If left = right, then array deque become empty and assigned value of both left and right pointer to –1.

If right is at index 0, then assigned right = MAXSIZE –1 otherwise decrement right by one i.e. right = right – 1.

/*Algorithm deldeque delete an element from left or right end of the deque. A flag f = 0 indicate the left side deletion and f = 1 indicate the right side deletion in the deque*/

```
          Algorithm deldeque(int f)
Step 1 :  if (left = –1) then
                printf(" Queue is empty\n");
                return –1;
          end if
Step 2 :  if (left = right)then
          x = deque[left]; left = –1; right = –1;
Step 3 :  else
                if (f = 0) then
                     x = deque[left];
                    left = (left+1) % MAXSIZE;
Step 4 :  else
                x = deque[right];
                if (right = 0)then
                      right =  MAXSIZE –1;
                else
                      right = right –1;
                end if
           end if /* end of if of step3*/
end if /* end of if of step2 */
Step 5 :  return x;
```

Problem 11. (*a*) Consider two linked lists:

$$X_1 = (x_1, x_2, x_3 \ldots\ldots x_m)\ x_i \le x_{i+1}$$

$$X_2 = (x_1, x'_2, x'_3 \ldots\ldots x'_m)\ x'_i \le x'_{i+1}$$

Develop an algorithm to merge the two lists such that $x_i \le x_{i+1}$ in the merged list as well.

(*b*) Two stacks are to be mapped onto one an array. Find the positions of two stacks in the array. Also develop an algorithm to push a data element into the specified stack. The operation must return an error only if there is no space left in the array.

(*c*) Write an algorithm to match parentheses using stack.

Solution. (*a*) The merge algorithm merges two sorted list which is pointed by start and start1 pointer variable. The algorithm is written in form of 'C' function and it return the address of the merge list.

```
Algorithm merge(start, start1)
{
if (start = = NULL)
        start = start1;
else
        if (start ! = NULL)
        {
        ptr = start;
        ptr1 = start1;
        if (ptr1–>code < ptr–>code)
        {
                start1 = start1–>next;
                ptr1–>next = start;
                start = ptr1;
                ptr = start;
                ptr1 = start1;
        }
        while (start1 ! = NULL)
        {
                while ((ptr–>code < start1–>code) && (ptr! = NULL))
                        {
                        newnode = ptr;
                        ptr = ptr–>next;
                        }
                ptr1 = start1;
                start1= start1–>next;
                ptr1–>next = newnode–>next;
                newnode–>next = ptr1;
                ptr = ptr1;
        }
    }
return (start);
} /* end of merge */
```

(*b*) It is possible to keep two stacks in a single array. One grows from lower bound index other from upper bound index. The top1 variable indicates stack 1 and top2 variable indicates stack 2. The following push 'C' function performs the operations. The value of s = 1 indicate the insertion is stack 1 and value of s = 2 indicate the insertion in stack 2.

```
        void push(x, s)
        {
                if ((s! = 1) || (s! = 2))
                        {
```

```
                printf("\n Wrong choice");
            return;
            }
        if (s == 1)
        {
        if (top1 >= top2 –1)
        {
            printf("Stack 1 is in overflow");
            return;
        }
top1++;
stack[top1] = x;
return;
}
else
{
        if (top2 < = top1+1)
        {
        printf("\n Stack2 is overflow");
        return;
        }
top2– –;
stack[top2] = x;
return;
}
} /* end of push function */
```

(*c*) We want to ensure that the parenthesis of right and left parenthesis.

- There are an equal number of right and left parentheses.
- Every right parenthesis is preceded by a matching left parenthesis.

Let us define the parenthesis count at a particular point in an expression as the number of left parentheses minus the number of right parentheses that have been encountered in scanning the expression from its left end up to that particular point. The two conditions that must hold for matching.

1. The parenthesis count at the end of the expression is 0. This implies that no scopes have been left open or that exactly as many right parentheses as left parentheses have been found.
2. The parenthesis count at each point in the expression is non negative. This implies that no right parenthesis is encountered for which a matching left parenthesis had not previously been encountered.

Some example of parenthesis count is given below:

Counts → 0 0 1 2 2 2 2 1 1 1 0 1 2 3 3 3 3 2 2 2 1 1 1 0

A + ((B – A) / D) (((A * B) / B) + B)

Parenthesis count at various points of strings

A stack may be used to keep track of not only how many scopes have been opened but also their types.

Algorithm matchparenthesis

Step 1 : count = 0

Step 2 : valid = true; /* assume the string is valid */

```
        s.top = –1;       /* stack s is empty */
Step 3 : while (! Eof string )
        {
        Read the next symbol symb of the string;
Step 4 : if (symb = '(' || symb = '[' || symb = '{') then
                count++;
                push (s, symb);
        end if
Step 5 : if (symb = ')' || symb = ']' || symb = '}' )
                if (empty(s))
                                valid = false;
                else
                        i = pop(s);
                        if (i is not the matching opener of symb)
                        valid = false;

                end if
        end if
Step 5 : if (!empty(s)) then
                valid = false;
        end if
Step 6 : if (valid) then
                write("The string is valid");
        else
                write("The string is not valid");
        end if
```

Problem 12. (*a*) What are Circular Queues? How they differ from Sequential Queues? Write algorithms for Addition and Deletion to a circular queue.

(*b*) Write Infix equivalent of the Prefix expression:

+–* ^ A B C D/E/F+GH

Solution. (*a*) An efficient queue representation can be obtained by taking an array and treating it as circular. Elements are inserted by increasing the pointer variable rear to the next location in the array. When rear = MAXSIZE –1, the next element is entered at items[0] in case when no element is at index 0. The variable front also increased as deletion and in same fashion as rear variable.

The Fig. 8 of circular queue is given below for n elements.

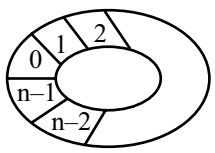


Figure 8 *Circular queue*

The operation of inserting an item onto a queue is called insert operation. The simple function to implement insert operation using the array representation of a queue. Here, we check whether the array is full before attempting to insert element onto the queue.

The conditions of circular queue full is either one of the below:

1. (front = 0) and (rear = MAXSIZE –1).
2. front = rear+1.

Whenever an element is inserted into the queue, the rear pointer variable value is increased by 1 (i.e. rear = (rear+1) % MAXSIZE;). So that after rear pointer variable values MAXSIZE – 1, it increased by one to denote rear value 0.

Insertion in circular queue is given in the form of 'C' function queueins

```
void queueins(struct queue *q,int x)    /* function definition */
{
/* check for queue full if  front pointer at index 0 and rear pointer at MAXSIZE –1. Or front is just
behind rear i.e. front = rear +1 */
if (((q–>front = = 0) &&(q–>rear = = MAXSIZE –1))||(q–>front = = q–>rear+1))
     {
          printf("Queue full\n");
          display(&q);                              /* function calling */
          return;
     }
else
{
     if (q–>front = = –1)
     { q–>front = 0; q–>rear = 0;}
     else
          q–>rear = (q–>rear+1) % MAXSIZE;
     q–>items[q–>rear] = x;
}
} /* End of queueins function */
```

The delete operation is given in the form of queuedel function as below:

The function needs one parameter, a return type. The parameter is reference to queue structure and it return deleted element value. Thus function declaration is:

```
int queuedel(struct queue *q)          /* function definition */
     {
     int x;
     /* check for queue empty when front pointer is at index –1. */
     if (q–>front = = –1)
          printf(" Queue is empty\n");
     x = q–>items[q–>front];
 /* check for front equal to rear means queue has become empty and reinitiate front and rear to –1.
Otherwise front is increased by 1 in modulus way */
```

```
    if (q–>front = = q–>rear)
    { q–>front = –1; q–>rear = –1;}
    else
        q–>front = (q–>front +1) % MAXSIZE;
return x;
}
```

(*b*) The infix expression is written from the prefix expression is follows:

Prefix +–* ^ A B C D/E/F+GH

Apply ^ operator between two successive operand A and B.	+–*A ^ B C D/E/F+GH
Next apply * operator	+–A ^ B * C D/E/F+GH
Next apply – operator	+A ^ B * C – D/E/F+GH
Next apply + operator	+A ^ B * C – D/E/F G +H
Next apply + operator	A ^ B * C – D / E / F+G +H
Infix expression	A ^ B * C – D / E / F+G +H

Problem 13. (*a*) What is recursion? Give the two essential conditions for recursion to happen?

(*b*) How is recursion implemented?

(*c*) Give the algorithm for recursive binary search.

(*d*) What are the various methods of accessing a file? List advantages and disadvantages of each.

Solution. (*a*) A function that contains a function call to itself, or a function call to a second function which eventually causes the first function to be called, is known as a recursive function.

The factorial function, whose domain is the natural numbers, can be recursive defined as

$$\text{fact}(n) = \begin{cases} 1, \text{ if } n = 0 \\ n * \text{fact } (n - 1), \text{ otherwise} \end{cases}$$

There are two important conditions that must be satisfied by any recursive function:

1. Each time a function calls itself (either directly or indirectly), it must be "nearer" in some sense, to a solution. In the case of factorial function, each time that the function calls itself, its argument decrements by one, so the argument of the function gets smaller.
2. There must be a decision criterion for stopping the process or computation. In the case of the factorial function, the value of n must be zero.

(*b*) The general algorithm model for any recursive function contains the following steps:

1. Save the parameters, local variables, and return address.
2. If the base criterion has been reached, then perform the final computation and go to step 3; otherwise, perform the partial computation and go to step 1 (initiate a recursive call).
3. Restore the most recently saved parameters, local variables, and return address.

(*c*) We state the binary search algorithm formally as given below:

```
Algorithm BSEARCH(v, lb, ub, sElem)
/* Here v is a nElem sorted array with lower bound lb and upper bound ub and sElem is an element
to be searched and if found then return the index otherwise return –1. The local variable low, mid, high
denote respectively, the beginning, middle and end locations of the array elements */
```

```
Step 1 :  Initialize variables
          Set  low = lb; high = ub; and loc = –1;
Step 2 :  If (low > high)
               return (–1);
Step 3 :  mid = (low+high)/2;
Step 4 :  if  sElem = v[mid]
               return(mid);
Step 5 :  if (sElem < v[mid] )
               BSEARCH(v, low, mid – 1, sElem);
          else
               BSEARCH(v, mid+1, high, sElem);
Step 6 :  end BSEARCH
```

(*d*) The most common file organizations are:

1. Sequential Files

Sequential file organization has one access key and only sequential access. Data records are stored in some specific sequence, e.g. order of arrival, value of keyfields, etc. Records of a sequential file cannot be accessed at random i.e., to access the n^{th} record, one must traverse the preceding n–1 records.

Advantages

- Easy to handle
- Involve no overhead
- Can be stored on tapes as well as disks
- Records in a sequential file can be of varying length

Disadvantages

- Updates are not easily accommodated.
- By definition, random access is not possible.
- All records must be structurally identical. If a new field has to be added, then every record must be rewritten to provide space for the new field.
- Continuous areas may not be possible because both the primary data file and the transaction file must be looked during merging.

2. Direct File Organization

It offers an effective way to organize data when there is a need to access individual records directly.

To access a record directly (or random access) a relationship is used to translate the key value into a physical address. This is called the mapping function h

h(key value) = address.

A calculation is performed on the key value to get an address. This address calculation technique is often termed as hashing and mapping function is called hash function.

We have already discussed hashing in details in Chapter 10.

Advantages

- Records can be accessed out-of-sequence, randomly
- Well suited for interactive application
- Support updation operation in place

Disadvantages

- Can only be stored on disks
- Involved overhead in address calculation
- Records can only be of fixed length

3. Index Sequential File

Index sequential file organization has one access key and no direct access and sequential access. An index is added to the sequential file to provide random access. An overflow area needs to be maintained to permit insertion in sequence.

Advantages

- Records can be accessed sequentially and randomly.
- Supports interactive as well as batch-oriented applications.
- Supports updation operation in place.

Disadvantages

- Can only be stored on disks
- Involved more overheads in the form of maintenance of indexes
- Records can only be of fixed length

Problem 14. (*a*) Explain the main parts of a recursive algorithm. Write a recursive program/algorithm to add from 1 to n where n is input of the program.

(*b*) What is sparse matrix? How it is represented in memory? Explain algorithm to add two sparse matrices, how it is more efficient than general addition algorithm.

Solution. (*a*) There are two important parts that must be satisfied by any recursive function:

1 Each time a function calls itself (either directly or indirectly), it must be "nearer" in some sense, to a solution. In the case of factorial function, each time that the function calls itself, its argument is decrements by one, so the argument of the function is getting smaller.

2. There must be a decision criterion for stopping the process or computation. In the case of the factorial function, the value of n must be zero.

The recursive function for addition is given below:

```
        Algorithm add(n)
Step 1 : if (n = 1)
            return 1;
Step 2 : if (n>1)
            return( n+add(n–1));
Step 3 : end add
```

(*b*) An mxn matrix *A* is said to be *sparse* if many of its elements are zero. A matrix that is not sparse is called *dense* matrix. It is not possible to define an exact boundary between dense sparse matrices. The diagonal and tridiagonal matrices fit into the category of sparse matrices. In this section, we will consider sparse matrices with an irregular or unstructured nonzero region. Fig. 9 shows 6 × 7 sparse matrix with ten nonzero elements out of 42 elements.

$$M[][] = \begin{Bmatrix} 0 & 0 & 6 & 0 & 9 & 0 & 0 \\ 2 & 0 & 0 & 7 & 8 & 0 & 4 \\ 10 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 5 \end{Bmatrix}$$

Figure 9

One of the basic methods for storing such a sparse matrix is to store nonzero elements in a one-dimensional array and to identify each array element with row and column indices as shown below in Fig. 10.

The i^{th} element of vector A is the matrix element with row and column indices row[i] and column[i]. A more efficient representation in terms of storage requirements and access time to the rows of the matrix. The row vector is changed so that its i^{th} element is the index to the first of the column indices for the elements in row i of the matrix.

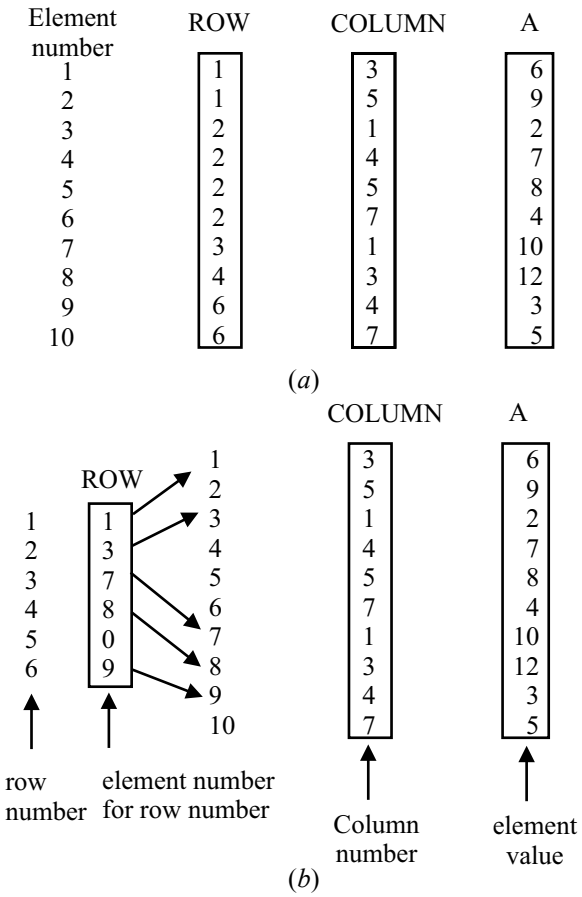


Figure 10 *Sequential representation of sparse matrices*

The representation of matrix A in Fig. 10 has cut-storage requirements by more than half. For large matrices the conservation of storage is very significant.

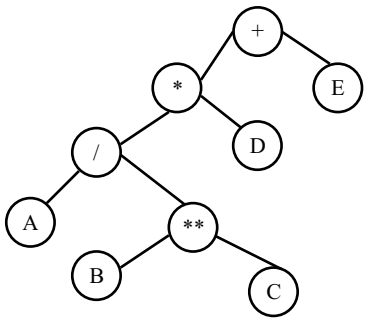
Sequential allocation schemes for representing sparse matrices generally allow faster execution of matrix operations and are more storage efficient than linked allocation schemes.

The matrix addition for the above sequential representation is much faster than the conventional matrix addition.

A general algorithm for adding two matrices is as follows:

1. Repeat through step 8 until all rows have been processed.
2. Obtain current row indices for matrix A and B.
3. Obtain starting position in vectors of next row in matrix A and B.
4. Repeat thru step 6 until either the end of matrix A's or B's row is reached.
5. If elements exist in the same column in matrix A and B then sum elements then move onto the next column in matrix A and B
   else if matrix A column number is less than matrix B's column number then move on to the next column in matrix A
   else move on to the next column in matrix B
6. Add new element to matrix C
7. If the end of matrix A's row has not been reached then
   Add remaining elements in row to matrix C
8. If the end of matrix's B row has not been reached then
   Add remaining elements in row to matrix C.

Problem 15. (*a*) Write an output after traversing of a given tree by Inorder and Preorder traversing methods.



(*b*) Explain the different methods of binary tree representation.

Solution. (*a*) In order traversal of the tree is A / B ** C * D+E

(*b*) Pre order traversal of the tree is +* / A ** B C D E

1. Sequential Representation

Suppose T is a binary tree that is complete or partially complete. The easy way of maintaining T in memory is sequential representation. This representation uses only a linear array BTREE as follows:

(*i*) The root of tree is stored in BTREE[0].

(*ii*) If a node occupies BTREE[i], then left child is stored in BTREE[2*i +1] and its right child is stored in BTREE[2*i +2]

(*iii*) If the corresponding tree does not exist then its pointer is null.

In general, the sequential representation of a binary tree with depth d will require an array with approximately 2^{d+1} elements or locations.

2. Linked List Representation

Two types of implementation:

(*i*) Array Implementation

(*ii*) Dynamic memory allocation implementation

(*i*) **Array Implementation**

```
Struct  nodetype
        {
                int info;
                int left;
                int right;
        }node[MAXSIZE];
```

MAXSIZE define the maximum number of nodes in the binary tree. Each node of binary tree consist of following field

(*i*) node[i].info contains the data at the node i.

(*ii*) node[i].left contains the location of the left child of node i.

(*iii*) node[i].right contains the location of the right child of node i.

The root contain the location of the root of the tree T. If any subtree is empty, then the corresponding pointer will contain the null value. If the tree T itself is empty, then root will contain null value. In the array implementation null value is denoted by 0.

(*ii*) **Dynamic Memory Allocation Implementation**

```
struct  nodetype
        {
                int info;
                struct nodetype *lchild;
                struct nodetype *rchild;
        }*node;
```

In binary tree each node has two child pointers, lchild pointer contains the address of left child of the node and rchild pointer contain the address of right child of the node. If the child of the node does not exists then respective pointer consist of null value.

Memory is allocated for each node using malloc function in 'C' programming.

```
node = (struct nodetype *)malloc(sizeof(struct nodetype);
```

Problem 16. What is a binary search tree? Write a recursive "C" function to search for an element in a given binary search tree. Write a non-recursive version for the same.

Solution. A binary search tree, BST, is an ordered binary tree T such that either it is an empty tree or

- each element value in its left subtree less than the root value
- each element value in its right subtree is greater than or equal to the root value, and
- left and right subtrees are again binary search trees

The inorder traversal of such a binary tree gives the set elements in ascending order.

A sorted array can be produced from a BST by traversing the tree in inorder and inserting each element sequentially into array as it is visited. It may possible to construct many BSTs from given sorted array.

Searching a Binary Search Tree

Since the definition of a binary search tree is recursive, it is easiest to describe a recursive search method.

Suppose we wish to search for an element with key x. We begin at the root. If the root is NULL, then the search tree contains no nodes and the search is unsuccessful. Otherwise, we compare x with the key in the root. If x equals this key, then the search terminate successfully. If x is less than the key in the root, then search in the left subtree. If x is larger than the key in the root, only the right subtree needs to be searched.

The recursive algorithm for searching is given below:

```
            Algorithm BTRECSRC(p, x)
            /* recursive search of a binary search tree */
Step 1:     if (p = NULL) then
                return –1;
Step 2:     else if (x = p–>Info) then
                return p;
Step 3:     else if (x < p–>Info) then
                return(BTRECSRC(p–>lchild, x);
                else
                return(BTRECSRC(p–>rchild, x);
```

The non–recursion algorithm for binary search is given below:

```
            Algorithm BTSRCH(p, x)
            /* Iterative search of a binary search tree */
Step 1:     repeat step2 while (p ! = NULL) and (p–>Info ! = x)
Step 2:         if (p–>Info > x) then
                    p = p–>lchild;
                else
                    p = p–>rchild;
            [end of while loop at Step1]
Step 3:     return (p);
```

Problem 17. What is an AVL tree? Insert the following integers to an initially empty AVL tree of integers:

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

Show the tree after each insertion specifying the rotations (if any) involved.

Solution. An AVL tree is a binary search tree in which the heights of the left and right *subtrees* of the root differ by at most 1, and in which the left and right subtrees of the root are again AVL trees

Balance Factor (bf) = $H_L - H_R$

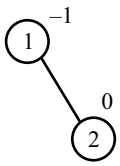
An empty binary tree is an AVL tree. A non empty binary tree T is an AVL tree iff given $|H_L - H_R| <= 1$. $H_L - H_R$ is known as the *balance factor* (BF) and for an AVL tree the balance factor of a node can be either 0, 1, or –1.

An AVL search tree is a binary search tree which is an AVL tree but vice-versa is not true.

(*i*) 1 

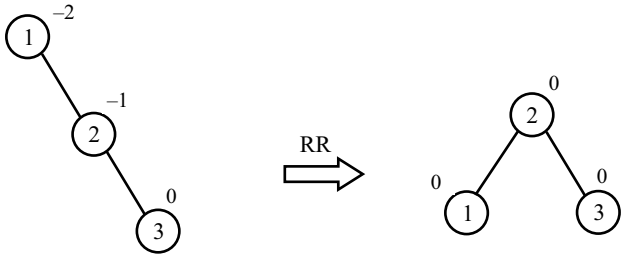
No rebalancing needed

(*ii*) 2

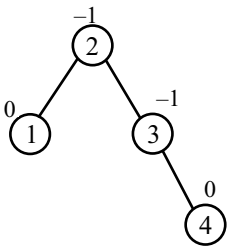


No rebalancing needed

(*iii*) 3

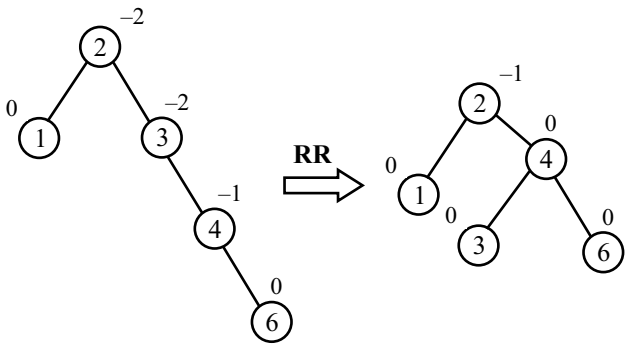


(*iv*) 4

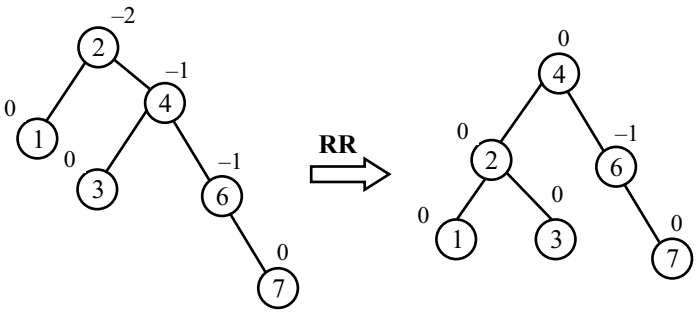


No rebalancing needed

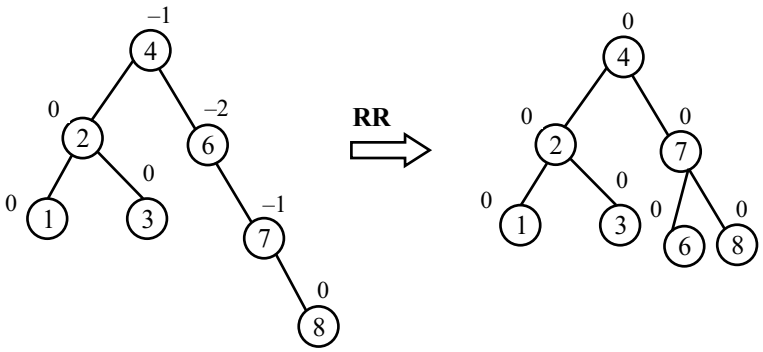
(*v*) 6



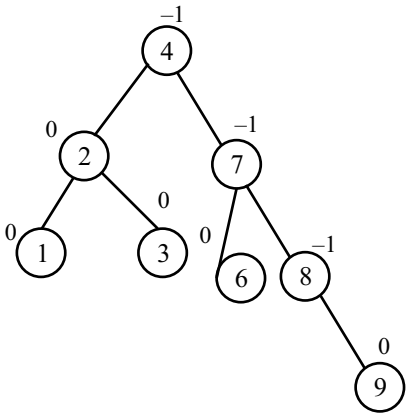
(*vi*) 7



(*vii*) 8

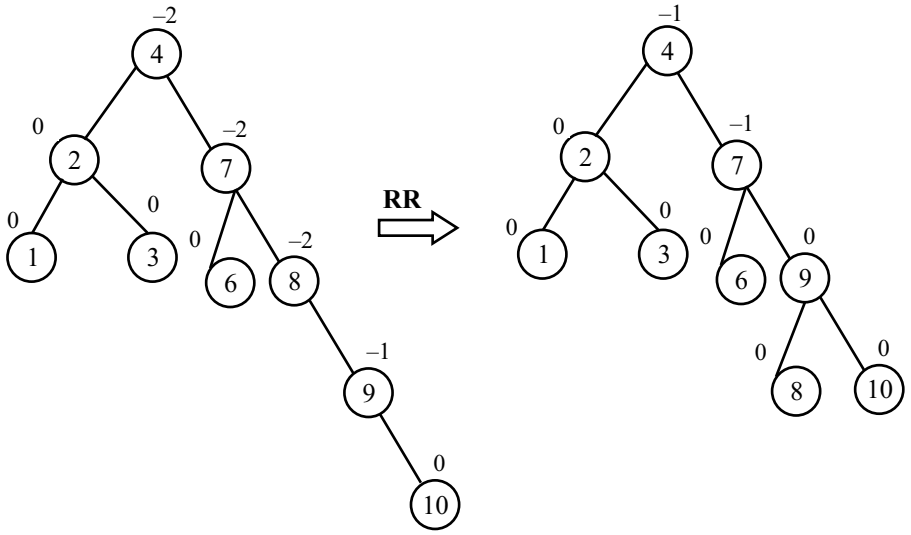


(*viii*) 9

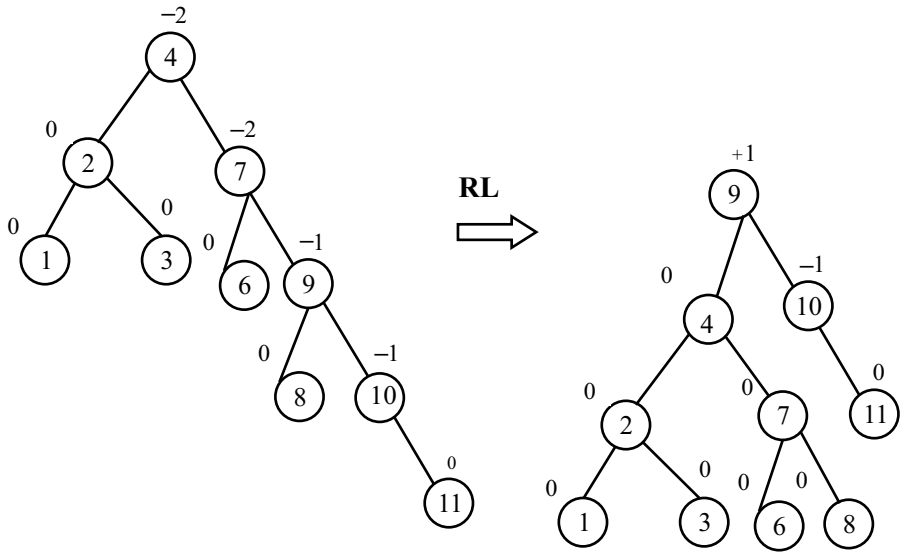


No rebalancing needed

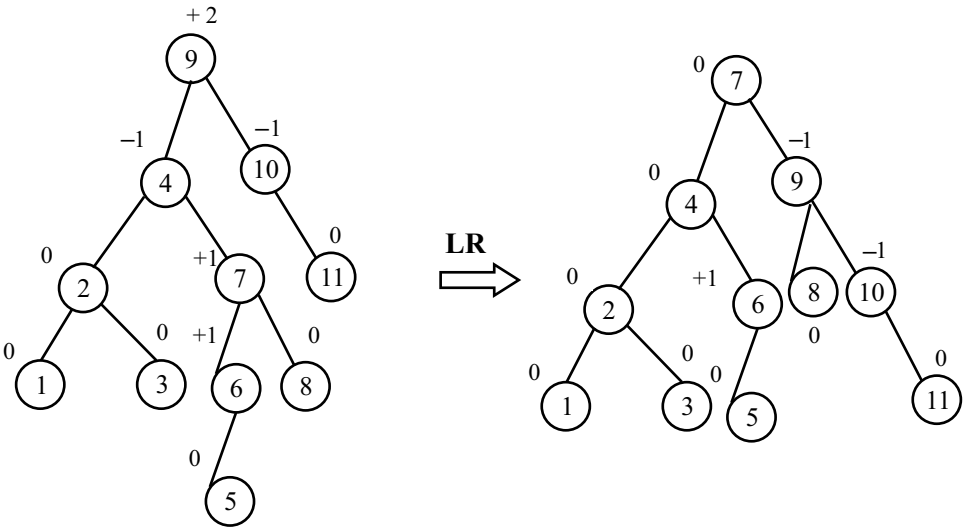
(*ix*) 10



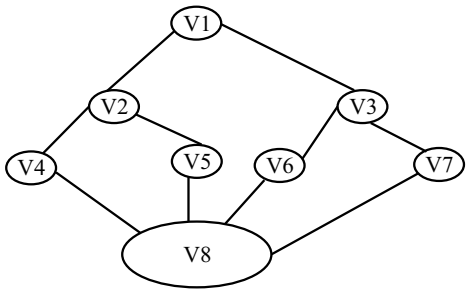
(*x*) 11



(*xi*) 5

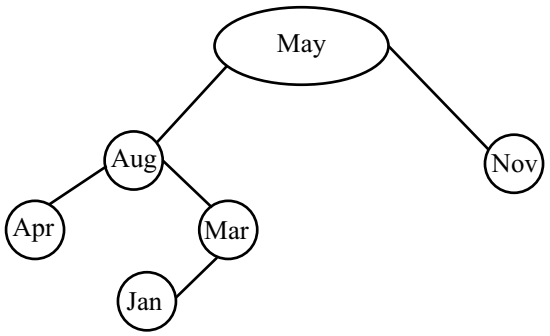


Problem 18. (*a*) Draw the adjacency list representation for the following graph:



(*b*) Using above representation, develop an algorithm to traverse a graph in depth first search (dfs) order. Clearly mention the data structures and other procedures used. Find the sequence in which nodes of the above figure will be traversed.

(*c*) Consider the following tree

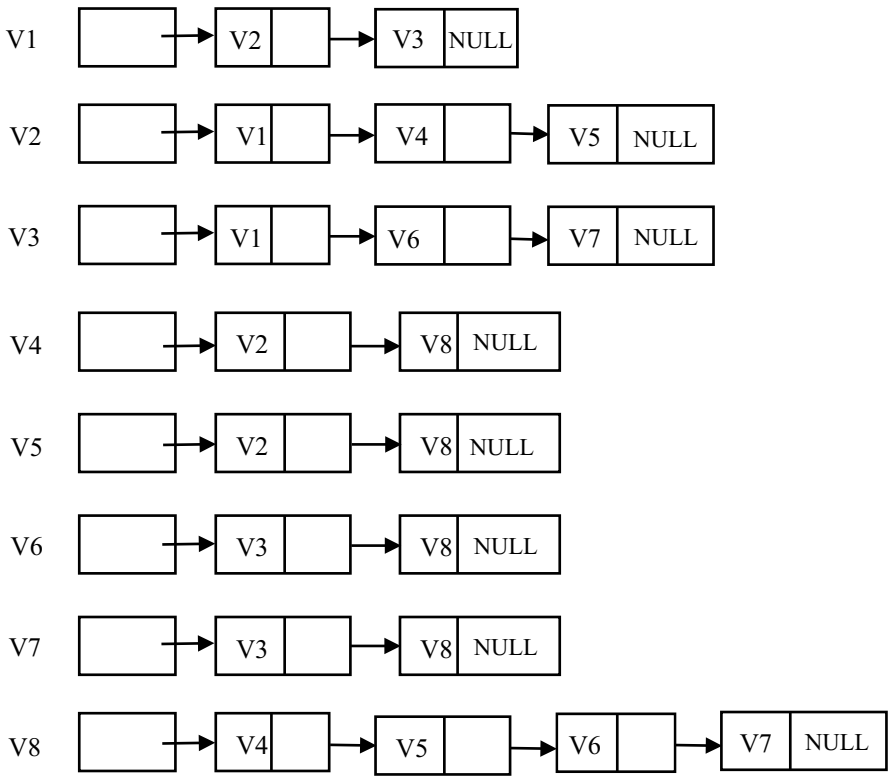


(*i*) Identify the point of unbalance in the tree;

(*ii*) Compute balancing factors of all the nodes in the tree;

(*iii*) Convert the binary tree into an AVL tree. (Draw only the tree showing the sequence in which pointers will be manipulated. No procedure to be written.)

Solution (*a*) **Adjaceny list :** In this representation of graphs, the n rows of adjaceny matrix are represented as n linked lists. There is one list for each vertex in G.



(*b*) **Depth First Search (DFS):** In depth first search we start at vertices v and mark it as having been reached. All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. Then vertex **v** has now been explored. The newly vertex have not visited are put onto the end of the list of unexplored vertices. The vertex first in the list is the next to be explored. This exploration continues until no unexplored vertex is left. The list of unexplored vertices as a stack and can be represented by stack. It is based on **LIFO** system.

A depth first search of G carried out beginning at vertices g for any node i, visited[i] =1 if i has already been visited. Initially, no vertex is visited so for all vertices visited[i] = 0.

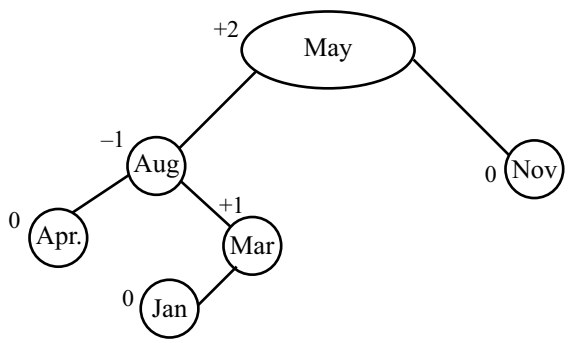
```
        Algorithm DFS(g)
        /* Non  Recursive */
Step 1: h = g;
Step 2: visited[g] = 1;
Step 3:  repeat
              {
Step 4 : for all vertices  w  adjacent  from h do
              {
                 if (visited[w] = 0)  then
                 {
                        push w  to  s;   //  w is unexplored
                        visited[w] = 1;
                 }                    //  end of the if
                 }        // end of the for loop
Step 5: if  s is empty  then  return;
                    pop  h from s;
            }  until(false);
 //  end of the  repeat
Step 6: end  DEPTH  FIRST SEARCH.
```

The sequence of Depth first search for start vertex V1 is: V1, V2, V4, V8, V5, V6, V3, V7.

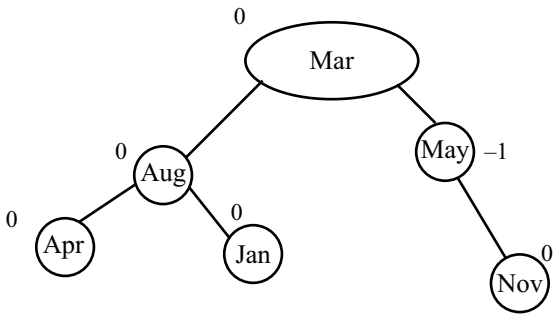
(*c*)

(*i*) The imbalance is at node May.

(*ii*) The balancing factor is as given below:



(*iii*) Balance the above binary search tree with LR rotation.



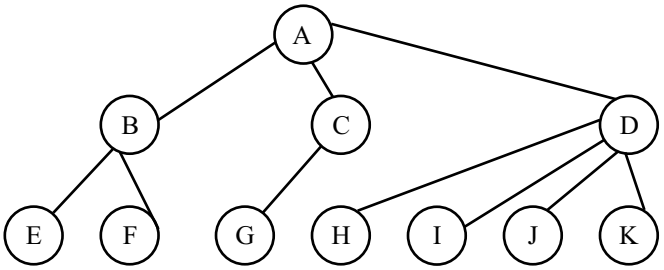
Problem 19. (*a*) What is a Threaded Binary Tree? How does it differ from Binary Tree? Write algorithm for INORDER traversal of Threaded Binary Tree.

(*b*) Write C procedure for non-recursive preorder traversal is defined in the following manner:

(*i*) Visit the root r

(*ii*) Visit each child in preorder (taking children) from left to right.

For example, preorder traversal of the following tree is A, B, E, F, C ,G, D, H, I, J, K.



Solution. (*a*) The wasted NULL links in the storage representation of binary tree can be replaced by threads. A binary tree is threaded according to a particular traversal order. For example, the threads for the inorder traversal of a tree are pointers to its higher nodes. For this traversal order, if the left link of a node P is normally NULL, then this link is replaced by the address of the predecessor of P. Similarly, a normally NULL right link is replaced by the address of the successor of the node in question.

It differs from binary tree in the following respect:

1. The left and right child pointers in the threaded binary tree denotes the predecessor and successor node if they are empty.
   In binary tree these node's pointers contain NULL values.
2. The structure for threaded tree is similar to binary tree except a flag which indicates the pointer is thread or structural link.
3. Given a threaded tree for a particular order of traversal, it is a relatively simple task to develop algorithm to obtain the predecessor or successor nodes of some particular node P.

Algorithm TINORDER

Step 1: [initialize]
P = HEAD;

Step 2: [Traverse threaded tree in inorder]
while TRUE do
P = INS(P); /* it call INS algorithm which return the inorder successor */
if P = HEAD then
exit;
else
write (P–>info);

Algorithm INS(X)
/* X is address of node in a threaded BT. The algorithm returns the address of its inorder successor. P is the temporary pointer variable. */

Step 1: [Return the right pointer of the given node if a thread]
P = X–>rchild;
if (rchild–>X) < 0) then
return (P);

Step 2: [Branch left repeatedly until a left thread]
while (P–>lchild)>0 do
P = P–>lchild;

Step 3: [Return address of successor]
return P;

(*b*) **Algorithm Preorder(root)** //Non Recursive
/* A binary tree T is in memory and array stack is used to temporarily hold address of nodes */

Step 1: [Initialize top of stack with null and initialize temporary pointer to nodetype is ptr]
Top = 0;
stack[Top] = NULL;
ptr = root;

Step 2: repeat steps 3 to 5 while ptr ! = NULL

Step 3: visit ptr → Info

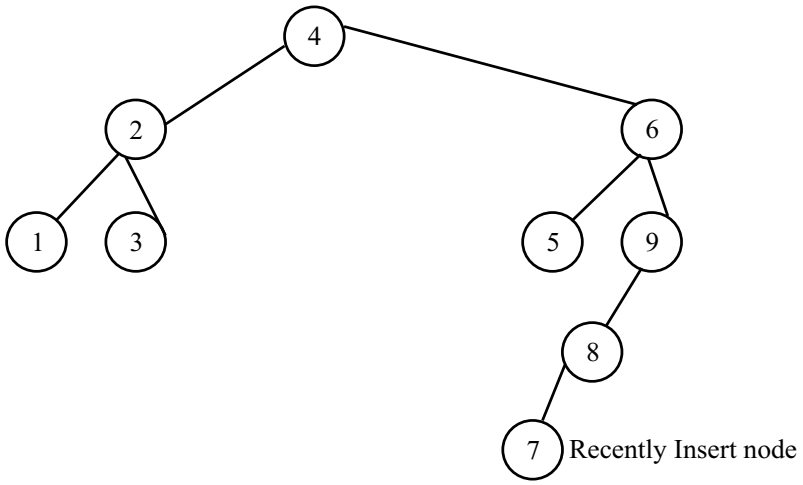
Step 4: [Test existence of right child]
if (ptr → rchild ! = NULL)
{
Top = Top+1;
stack[Top] = ptr → rchild;
}

Step 5: [Test for left child]
if (ptr → lchild ! = NULL)
ptr = ptr → lchild;
else
[Pop from stack]

```
            ptr = stack[Top];
            Top = Top–1;
        [End of loop in Step 2.]
```

Step 6: end Preorder.

Problem 20. (*a*) Consider the following tree:



(*i*) Compute the balancing factor of all the nodes in the tree.

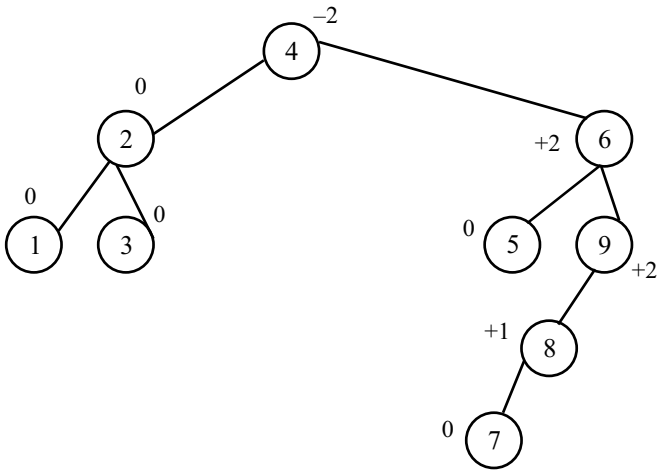
(*ii*) Identify the type of rotation in the tree.

(*iii*) Convert the binary tree into an AVL tree (Show rotations)

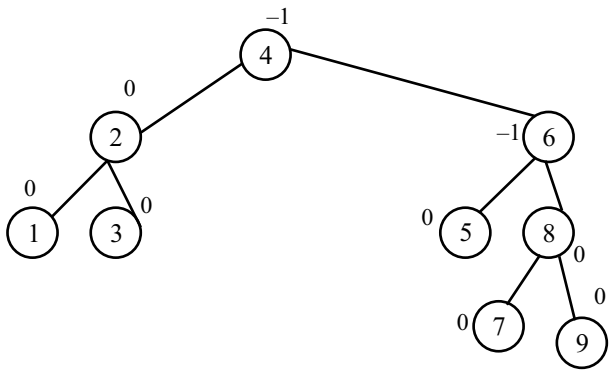
(*b*) Write an algorithm for deleting a node from binary search tree, which has exactly one child. The algorithm should first search for that node and point appropriate pointers.

(*c*) Write an algorithm that uses binary search tree to find duplicate numbers in an input file.

Solution. (*a*) (*i*) The balancing factor for each node is given below:



(*ii*) & (*iii*) The left rotation requires as inserted node is in the left subtree of the ancestor node 9 which is imbalance. This LL rotation needed as below, hence tree become AVL tree.



(*b*) To delete an element x, we must first verify that its key exists into the binary search tree. The below algorithm BTSRCH search the node to be deleted with information x. The node to be deleted is pointed by dnode and its parent node is denoted by pardnode.

Algorithm BTSRCH(x)

Step 1: Set dnode = root;
Step 2: while(dnode! = NULL) and (dnode–>info ! = x)
Step 3: Set pardnode = dnode;
if (dnode–>info >x) then
Set dnode = dnode–>lchild;
else
Set dnode = dnode–>rchild;
[End of while loop at step 2]

The below algorithm btreedel delete the node which has exactly one child.

Algorithm BTREEDEL(x)

Step 1: call btsrch(x); /* search the node to be deleted in the tree */
Step 2: if(dnode = NULL)then
write("Not found"); and exit;
Step 3: if (dnode–>lchild)! = NULL then
Set child = dnode–>lchild;
else
Set child = dnode–>rchild;
Step 4: if (pardnode ! = NULL) then
if (dnode = pardnode–>lchild)
Set pardnode–>lchild = child;
else
Set pardnode–>rchild = child;
else
root = child;
Step 5: end BTREEDEL

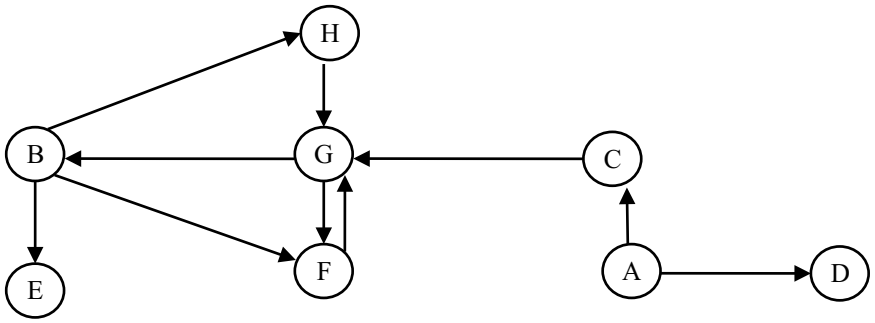
(*c*) To verify that an element with duplicate information in the binary search tree, we must first verify that its key is same from those of existing nodes.

The below algorithm BTREEDUP check the element with information x exist or not in the tree. The root pointer variable points the address of the root node in the binary search tree.

```
        Algorithm BTREEDUP(x)
Step 1: [initialize variables]
        set found = 0; set p = root;
Step 2: repeat step 3 while (p ! = NULL) and ( !found)
Step 3:     parent = p;
            if (p–>Info = x) then
                Write ("Node is duplicate");
            else if (x < p–>Info) then
                Set p = p–>lchild;
                else
                Set p = p–>rchild;
        [end of while loop at Step 2]
Step 4: end BTREEDUP
```

Problem 21. (*a*) Discuss the graph representation methods in detail.

(*b*) Differentiate between Depth First Traversal and Breadth First Traversal of graph. Draw the Breadth First Traversal of the following graph. (start from Vertex A)



Solution. (*a*) Graph can be represented in following ways:

1. Adjacency matrix
2. Adjacency list

1. *Adjacency matrix*: let G = (V, E) be a graph with n vetices, n> = 1. The adjacency matrix of G is a two-dimensional n*n array (i.e., matrix), say with the property that a[i,j] = 1 if and only if there is an edge(i,j) is in E(G). The element a(i,j) = 0 if there is no such edge in G.
   For weighted or cost graphs, the position in the matrix is the weight or cost.

Adjacency matrix:

Row \ Column	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	0	0	1	0

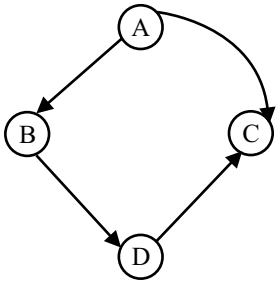


Figure 11 *Directed graph and its adjacency matrix*

2. *Adjacency list*: In this representation of graphs, the n rows of adjacency matrix are represented as n linked lists. There is one list for each vertex in G. See Fig. 12.
   For weighted graphs, include the weight/cost in the elements of the list.

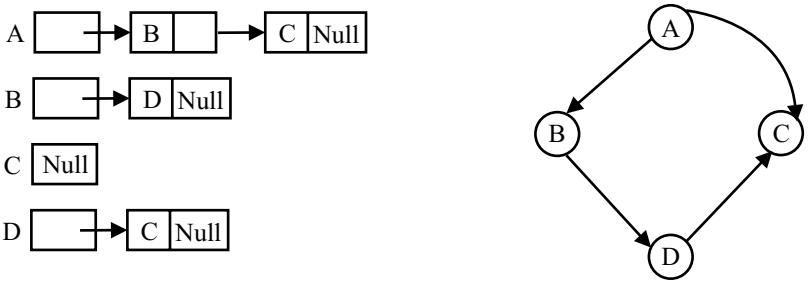
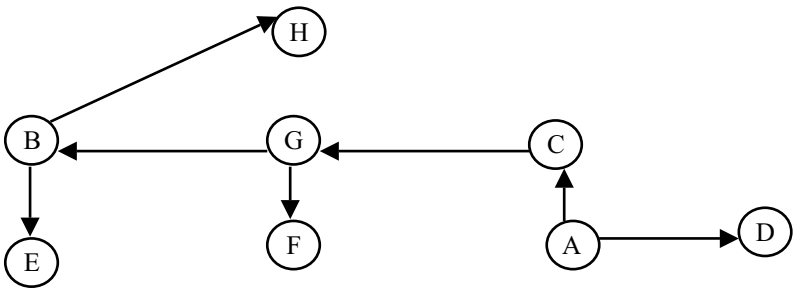


Figure 12 *Directed graph and its adjacency list*

(*b*) BFS and DFS are two fundamentally different search methods. In BFS a vertex is fully explored before the exploration of any other node begins. The next vertex to explore is the first unexplored vertex remaining. In DFS the exploration of a vertex is suspended as soon as a new unexplored vertex is reached. The exploration of this vertex immediately begins.
The time complexity of BFS algorithm is O(n+|E|) if G is represented by adjacency lists, where n is number of vertices in G and |E| number of edges in G. If G is represented by adjacency matrix, then T(n, e) = $O(n^2)$ and space complexity remain S(n,e) = O(n) same.

The time complexity of DFS algorithm is O(n+|E|) if G is represented by adjacency lists, where n is number of vertices in G and |E| number of edges in G. If G is represented by adjacency matrix, then T(n, e) = $O(n^2)$ and space complexity remain S(n, e) = O(n) same.



The BFS sequence is A, C, D, G, B, F, E, H

Problem 22. Trace the quicksort algorithm for the following data.

6, 5, 4, 7, 12, 11, 9, 10, 8, 2

Solution. Take x[0] as pivot number and place it at its proper position by swapping the elements.

Table: Behaviour of Quicksort (Recursive Algorithm)

Partition Call	*x[0]*	*x[1]*	*x[2]*	*x[3]*	*x[4]*	*x[5]*	*x[6]*	*x[7]*	*x[8]*	*x[9]*
Initially	**{6**	5	4	7	12	11	9	10	8	2)
1	**{2**	5	4}	6	{12	11	9	10	8	7}
2	2	**{5**	4}	6	{12	11	9	10	8	7}
3	2	{4}	5	6	**{12**	11	9	10	8	7}
4	2	4	5	6	**{11**	9	10	8	7}	12
5	2	4	5	6	**{9**	10	8	7}	11	12
6	2	4	5	6	**{7**	8}	9	10	11	12
7	2	4	5	6	7	{8}	9	10	11	12
Result sorted	2	4	5	6	7	8	9	10	11	12

Bold face numbers are pivot number.

Problem 23. (*a*) Write a routine that reverses a singly linked list without using any additional node.

(*b*) Write a function to add two double variables polynomials, using linked list representation.

Solution. (*a*) The reverse function used few pointers variable to reverse the single linked list without adding any node in the list. The ptr pointer variable denotes the current node position, prevptr pointer variable denotes the previous node position and rstart pointer variable store the address of the last node of the linked list and start pointer variable points the starting node address of the list.

```
void reverse()
{
while (start–>next ! = NULL)
{
        ptr = start;
        prevptr = NULL;
        while (ptr–>next ! = NULL)
        {
        prevptr = ptr;
        ptr = ptr–>next;
        }
        if (rstart = = NULL) /* store the last node address into rstart pointer variable */
                rstart = ptr;
        ptr–>next = prevptr;  /* last node denotes the address of previous node */
        prevptr–>next = NULL; /* make the previous node as last node of the linked list */
}       /* end of while */
        start = rstart;
}
```

(*b*) Addition of two Polynomials of two variables

Let us consider the addition of two polynomials of three variables

$p = 2x^3+2xy+y^2$ and $q = 2x^2 - y^2+2y$.

We can visualize this as follows

$p = 2x^3+ 2xy+y^2$

$q = + 2x^2 - y^2+2y$

$r = 2x^3+2x^2+2xy+2y$

The specific algorithm POLY2ADD is written in the form of function. This function has as input two ordered linear lists, which represent two polynomials to be added. Given two polynomials whose first terms are referenced by pointer variable p and q respectively. The third or sum polynomial is represented by pointer variable r. Pointer variable ptrp, ptrq, and ptrr are to keep track polynomial p, q and r respectively.

Algorithm POLY2ADD

[This algorithm adds two polynomials and returns the address of the first term of the sum polynomial. Consider that the input polynomials are stored in memory].

Step 1: Set ptrr = r; ptrp = p; ptrq = q; [Initially the sum polynomial r is empty, and pointer variables ptrp and ptrq denote the first term of the polynomials p and q, respectively]

Step 2: repeat step 3 to step 5 while (ptrp ! = NULL) and (ptrq ! = NULL)

Step 3:
```
Set a1 = ptrp–>powerx;
Set a2 = ptrq–>powerx;
Set b1 = ptrp–>powery;
Set b2 = ptrq–>powery;
Set d1 = ptrp–>coef;
Set d2 = ptrq–>coef;
```

Step 4:
```
If (a1 = a2) and (b1 = b2) then
    If (d1+d2 ! = 0) then
        ptrr = POLYLAST(ptrr, a1, b1 d1+d2)
    Set ptrp = ptrp–>next; and ptrq = ptrq–>next;
else
```

Step 5:
```
If (a1 > a2) or ((a1 = a2) and (b1>b2)) or ((a1 = a2) and (b1 = b2))) then
        ptrr = POLYLAST (ptrr, a1, b1 d1)
        set ptrp = ptrp–>next;
    else
        r = POLYLAST (ptrr, a2, b2 d2)
        Set ptrq = ptrq–>next;
[End of the loop at step 2]
```

Step 6:
```
if (ptrp ! = NULL) then[If polynomial p does not reach to last term then
    ptrr = COPY(ptrp, ptrr);          copy all remaining terms into sum polynomial r]
else                                  [Otherwise copy all remaining terms of
```

Step 7: ptrr = COPY(ptrq, ptrr); polynomial q into sum polynomial r]

Step 8: return (r); [Return the address of the first term of the sum Polynomial]

The algorithm POLY2ADD adds the terms of two polynomials referenced by pointer variable ptrp and ptrq, respectively. This algorithm invokes insertion function POLYLAST and a copying function COPY.

The POLYLAST algorithm is written in the form of function. This function is written for polynomial of two variables. The function has four input, one is sum polynomial list pointer variable ptrr, remaining three denoted by x, y and c, which correspond to the exponents for x, y and coefficient of the term. Function POLYLAST(ptrr, x, y, c) insert a node at the end of the linked linear list whose address is designated by the pointer ptrr. The global pointer variable r denotes the address of the first node in the list. The fields of the new term are x and c, which correspond to the exponent for x and the coefficient value of the term, respectively. The newnode is a pointer variable, which contains the address of the new node. The function returns the address of the last node.

Algorithm POLYLAST(ptrr, x, y, c)

Step 1: newnode = Allocate memory using malloc() function

Step 2: Set newnode–>powerx = x; [Assigned values for the term in new node]
Set newnode–>powerx = y;
Set newnode–>coef = c;

Step 3: if (ptrr = NULL) then [If the list is empty then add newnode as a first term
Set r = newnode; and set the address of newnode in pointer variable r]
else
Set ptrr–>next = newnode; [otherwise add the new term as the last one]

Set newnode–>next = NULL; [this denote the end of the list]

Step 4: return (newnode); [return the last node address]

The COPY algorithm is written in the form of function. Function COPY(ptrp, ptrr) makes a copy of the terms denoted by the pointer variable ptrp to end of the list into the list denoted by the pointer variable ptrr by invoking the function POLYLAST for the three variables polynomial. The COPY function return the address of the last node of the sum polynomial i.e. pointer variable ptrr.

Algorithm COPY(ptrp, ptrr)

Step 1: repeat step 2 while (ptrp ! = NULL)

Step 2: ptrr = POLYLAST(ptrr, ptrp–>powerx, ptrp–>powery, ptrp–>coef);
ptrp = ptrp–>next; [call POLYLAST function to add new term at the end of the list]
[End of the loop at step1]

Step 3 **:** return (ptrr); [return the last node address]

Problem 24. (*a*) Explain shell sort. Write a procedure to sort a data sequence using shell sort. Show how the data sequence 32, 2, 3, 12, 31, 53, 22 will be sorted using shell sort.

(*b*) Compare the behaviour of quick sort and merge sort sorting algorithms on the basis of the following points:

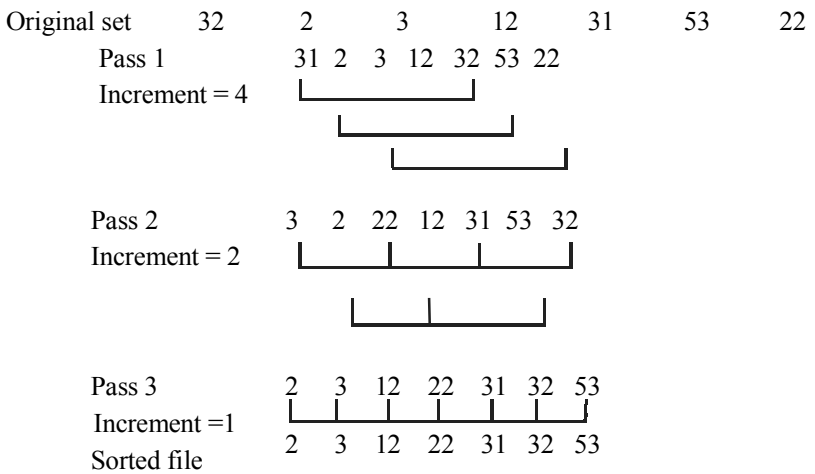
(*i*) No. of comparisons required (average and worst case);
(*ii*) Additional space required;
(*iii*) Time complexity (average and worst case)
(*iv*) Stability of algorithm

Solution: (*a*) This method sorts separate subset of original set of elements. This subset contains every k^{th} element of the original set. The value of k is called increment. After the first k subsets are sorted, a new smaller value of k is chosen and the file is again partitioned into a new set of subsets of elements. Each of these again sorted. Eventually, the value of k is set to 1 so that the only one subset consisting of the entire file is sorted.

The function for shell sort requires an array containing increments value inc, and number of increments nofinc.

```
void shelsort (int x[], int n, int inc[], int nofinc)
{
    int i, j, k, span, temp;
    static int nc = 0;
    for (i = 0;i< nofinc; i++){
    /* span is the size of the increment */
    span = inc[i];
    for (j = span;j<n; j++){
    /* insert element x[j] into its proper within its subsets */
        nc++;
        temp = x[j];
        for (k = j–span; k> = 0 && temp< x[k]; k = k–span)
            x[k+span] = x[k];
        x[k+span] = temp;
        }/*end of j inner for loop */
      }/* end of i outer for loop */
      printf("\n Number of comparisons required = %d", nc);
    }/* end of shelsort */
```

The shell sort is illustrated below. A decreasing sequence of increments <4, 2, 1> is fixed at the start of sorting.



(*b*) (*i*) The number of comparisons required for merge sort for average and worst case is:

$$f(n) <= n \log_2 n$$

The number of comparisons required for quick sort for average case $n\log_2 n$ and in worst case it is n^2.

(*ii*) The merge sort required additional space of O(n) and quick sort does not required any extra space.

(*iii*) Merge sort time complexity (average and worst case) is same i.e. $O(n\log_2 n)$.

(*iv*) Stability of merge sort is higher than quick sort. However quicksort is often the fastest available because of its low overhead and its average $O(n\log_2 n)$ behaviour. Another advantage of quicksort is locality of reference. That is, over a short period of time all array accesses are to one or two relatively small portions of the array.

Both mergesort and quicksort are methods that involve splitting the file into two parts, sorting the two parts separately, and then joining the two sorted halves together. In mergesort, the splitting is trivial (simply taking two halves) and joining is hard (merging the two sorted files).

In quicksort, the splitting is hard (partitioning) and the joining is trivial (the two halves and the pivot automatically form a sorted array).

Problem 25. (*a*) How Linked Lists tackle the manipulation of Symbolic Polynomials? Explain using algorithm the addition of two polynomials.

(*b*) Write a C implementation to add the data fields of a Linear Linked List and store the result at the end of the list.

Solution. (*a*) The problem of manipulation of symbolic polynomials is the classical example of the use of list processing. We can perform various operations on polynomials such as addition, subtraction, multiplication, division, differentiation, etc. We can represent any number of different polynomials as long as their combined size does not exceed our block of memory. In general, we want to represent the polynomial of single variable

$$f(x) = a_m x^m + \ldots + a_1 x^1 + a_0$$

of degree m, where a_i are non-zero coefficients with exponents i. Each term will be represented by a node. A node will be of fixed size having three fields, which represent the coefficient and exponent of a term plus a pointer to the next term.

Addition of Two Polynomials of Single Variable

Let us consider the addition of two polynomials of single variable

$$p = 6x^5 + 2x^3 + x + 4 \text{ and}$$
$$q = 2x^4 + 10x + 1.$$

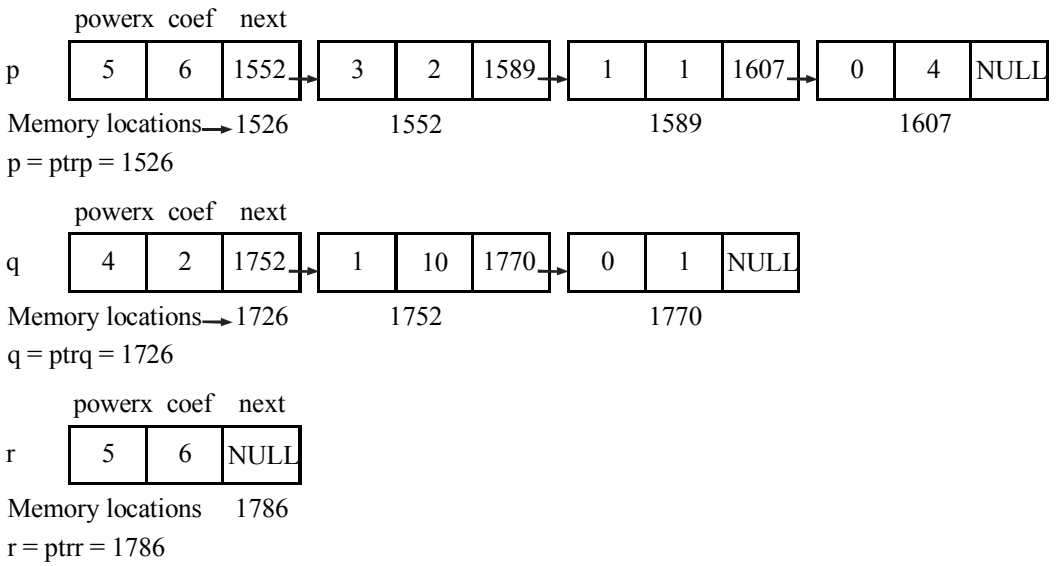
We can visualize this as follows:

$$\begin{array}{l} 6x^5 + 2x^3 + x + 4 \\ + \; 2x^4 + 10x + 1 \\ \hline = 6x^5 + 2x^4 + 2x^3 + 11x + 5 \end{array}$$

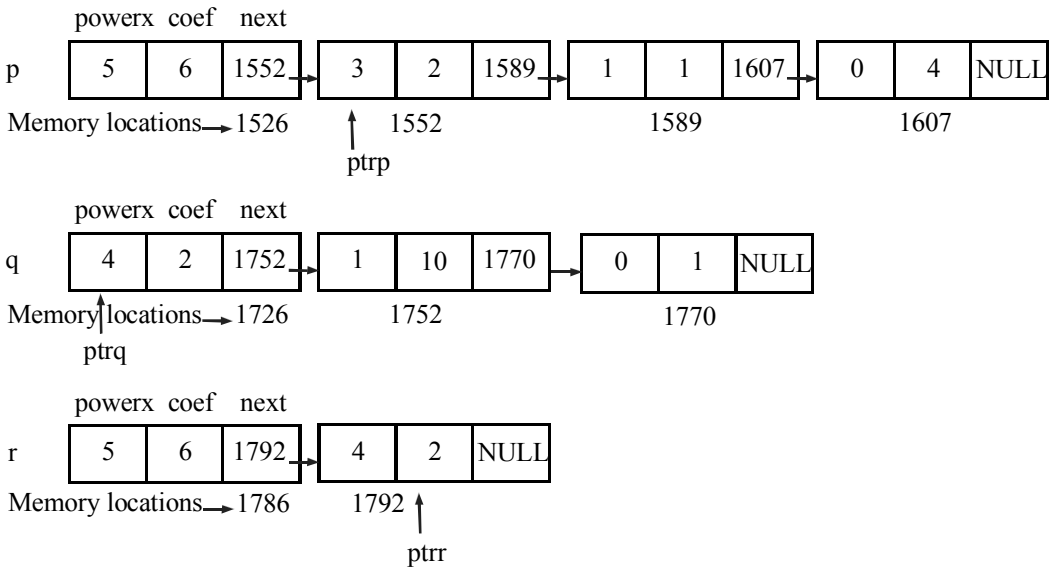
In order we examine their terms starting at the nodes pointed by start pointer variable p and q of the respective polynomial p and q. Two pointers ptrp and ptrq are used to move along the terms of p and q. If the exponents of two terms are equal, then the coefficients are added and if they does not cancel then new term created for the result. If the exponent of the current term in p is less than the exponent of current term of q, then a duplicate of the term of q is created for the result. The pointer ptrq is advanced to the next term. Similar action is taken on p if ptrp–>powerx > ptrq–>powerx.

Figure 13 illustrates this addition process on the polynomials p and q and resultant polynomial is r = p+q.

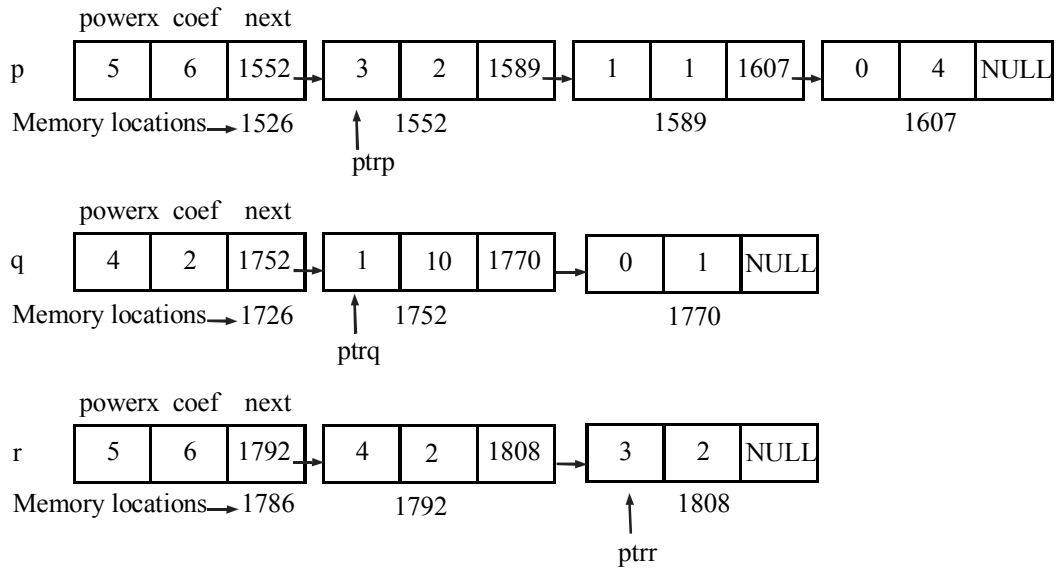
(*i*) ptrp–>powerx > ptrq–>powerx



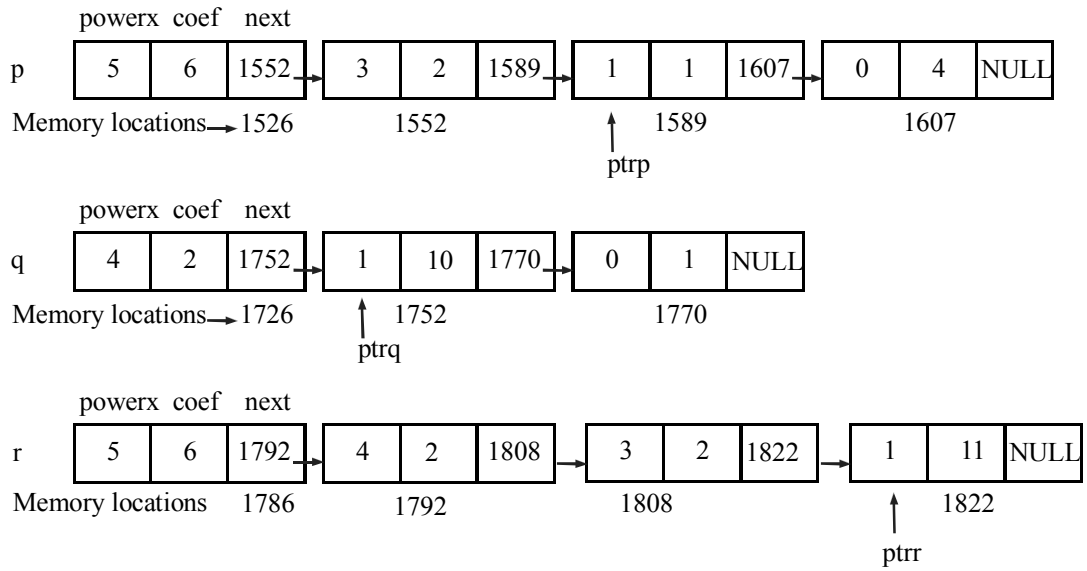
(*ii*) ptrp–>powerx < ptrq–>powerx



(*iii*) ptrp–>powerx > ptrq–>powerx



(*iv*) ptrp–>power = ptrq–>powerx



(*v*) ptrp–>powerx = ptrq–>powerx

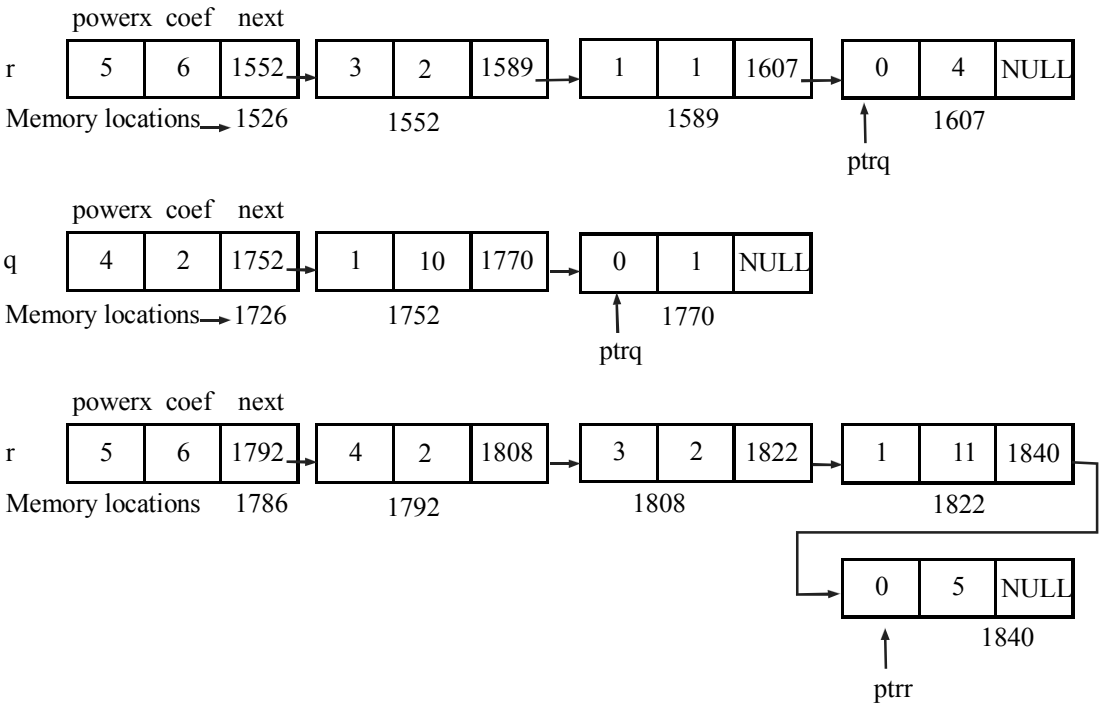


Figure 13 *Addition of two polynomials of single variables*

A general algorithm for addition of two polynomials with single variable is given below:

1. Repeat up to step 3 while there are terms in both polynomials yet to be processed
2. Obtain the values for each term
3. If the powers associated with the two terms are equal
   Then if the sum of coefficient of both terms is not zero
   Then insert the sum of the terms into the sum polynomial
   Obtain the next terms in both polynomials to be added
   Else if the power of first polynomial > power of the second
   Then insert the term from first polynomial into sum polynomial
   Obtain the next term in the first polynomial
   Else insert the term from second polynomial into sum polynomial
   Obtain the next term in the second polynomial
4. Copy remaining terms from nonempty polynomial into the sum polynomial
5. Return the address of the sum polynomial

(*b*) The function addnode add the field information in the nodes of linked list and store it into the last node.

```
void addnode(){int info = 0;node = start;
while(node–>next! = NULL)
{info = info+node–>code;
```

```
node = node–>next;}
node–>code = info;}
```

Problem 26. (*a*) How the sorting algorithms can be compared? Give a chart to show time and space complexity of various sorting algorithm.

(*b*) Why Quick sort is called as "Quick"? Show quick sort method for sorting the following data and find the number of comparisons.

13, 5, 3, 7, 10, 11, 9, 12

Solution. (*a*) **Summary of Sorting Methods**

The sorting methods discussed so far are summarized in below table. The parameter n denotes the number of elements (or keys) and m denotes the number of digit in a key. It is used in radix sort.

The goodness of any sorting algorithm depends upon the properties such as the number, size, distribution and order of keys in the set. The amount of memory available in performing the sort may also be an important factor.

In summary, the selection, bubble or insertion sort can be used if the number of elements in set is small. If n is large and keys are short (i.e. m is low), the radix sort can perform well.

With a large n and long keys, heap sort, quick sort or merge sort can be used.

If the set of elements, almost sorted then quick sort should be avoided. When the keys, after hashing, are uniformly distributed over the interval [1,m], then an address calculation sort is key good method.

Both merge sort and quick sort method are based on 'divide-and-conquer' that involve splitting the set into two parts, sorting the two parts separately, and then joining the two sorted halves together. In merge sort, the splitting is easy and joining is hard. In quick sort, the splitting is hard (partitioning) and joining is easy (as the few halves and the pivot automatically form a sorted array).

Insertion sort may be considered a special case of merge sort in which the two halves consists of a single element and the remainder of the array. Selection sort may be considered a special case of quick sort in which the set is partitioned into one half consisting of the largest element alone and second half consisting of remainder of the array.

Comparison of Sorting Method: (in approximation)

Algorithm	*Average*	*Worst case*	*Space usage*
Selection Sort	$n^2/2$	$n^2/2$	In place
Bubble Sort	$n^2/2$	$n^2/4$	In place
Insertion Sort	$n^2/4$	$n^2/2$	In place
Merge Sort	$O(n\log_2 n)$	$O(n\log_2 n)$	Extra n entries
Quick Sort	$O(n\log_2 n)$	$O(n^2)$	Extra $\log_2 n$ entries
Heap Sort	$O(n\log_2 n)$	$O(n\log_2 n)$	In place
Radix Sort	O(m+n)	O(m+n)	Extra space for links
Address Calculation Sort	O(n)	$O(n^2)$	Extra space for links
Shell Sort	$O(n(\log_2 n)^2)$	$O(n^2)$	Extra space for increment array

(*b*) In quick sort, the splitting is hard (partitioning) and joining is easy (as the few halves and the pivot automatically form a sorted array).

The quicksort algorithm is summarized below:

Partition call	*x[0]*	*x[1]*	*x[2]*	*x[3]*	*x[4]*	*x[5]*	*x[6]*	*x[7]*
Initially	{**13**	5	3	7	10	11	9	12}
1	**{5**	3	7	10	11	9	12}	13
2	{3}	5	{**7**	10	11	9	12}	13
3	3	5	7	**{10**	11	9	12}	13
4	3	5	7	{9}	10	**{11**	12}	13
5	3	5	7	9	10	11	{12}	13
sorted set	3	5	7	9	10	11	12	13

Bold face numbers are pivot number.

Problem 27. (*a*) Explain selection sort algorithm.

(*b*) Write a function QUICKCOUNT(A, N, NUMB) which sorts the array A with N elements and which also counts the NUMB of comparisons.

Solution. (*a*) A selection sort is one in which successive elements are selected in order and placed into their proper sorted positions.

The selection sort implements the descending priority queue as an unordered array.

The selection sort consists entirely of a selection phase in which the largest of the remaining elements is repeatedly placed in its proper position, i, at the end of the array.

The large element is interchanged with the element x[i].

The function for selection is given below:

```
void selectsort (int x[], int n)
{
/* x is an array of integers consists of n elements consider pos, and large is two temporary
variable. */
int i, j, large;
for (i = n–1; i>0; i– –){
        large = x[0];
        pos = 0;
        for (j = 1; j< = i; j++)
                if (x[j] > large){
                large = x[j];
                pos = j;
          } /* end of if and inner for loop*/
        x[pos] = x[i];
        x[i] = large;
         } /*end of outer for loop */
} /*end of selection sort*/
```

(*b*)

```
/* Function quickcount call recursively array until lower bound is smaller than upper bound and
   count  the number of comparisons.*/
void quickcount(int x[],int n,int lb, int ub,int numb)
{
int j, p = lb, q = ub;
int v = x[p],i = p,j = q,temp;
if(p<q)
{
        while(i<j)
{
numb++;
        while(x[i]< = v && i<p)
                i++;
        while(x[j]>v)
                j– –;
        if(i<j)
        {
                temp = x[i];
                x[i] = x[j];
                x[j] = temp;
        }
        }
        x[m] = x[j];
        x[j] = v;
quickcount(x,p,j–1,n);
quickcount(x,j+1,q,n);
}
printf("\n Number of comparisons = %d",numb);
}/* end of quickcount */
```

Problem 28. (*a*) Define minimum spanning tree. Write Prim's minimum spanning Tree algorithm.

(*b*) Write a procedure to compute length of shortest path of a directed graph using dynamic Programming method.

Solution. (*a*) **Definition:** Let G = (V, E) be an undirected connected graph. A sub graph T = (V, E') of G is a spanning tree of G iff T is a tree.

The cost of a spanning tree is the sum of the costs of the edges in that tree. There are two greedy methods to find the spanning tree whose cost is minimum called minimum cost spanning tree.

A greedy method to obtain a minimum cost spanning tree would build this tree edge by edge. The next edge to include is chosen according to some optimization criterion. There are two possible ways for optimization criterion due to Prim's and Kruskal's algorithm These algorithms are recommended for undirected graph.

In the first, the set of edges so far selected form a tree. Thus, if T is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be included in T is a minimum cost edge not in T with the property that A U { (v, u)} is also a tree. The corresponding algorithm is known as **Prim's algorithm.**

Prim's Algorithm: In Prim's algorithm we use the following steps:

Prim Algorithm (E, Cost, n, T)

/* E is the set of edges in G. cost[n][n] is the cost adjacency matrix of an n vertices graph such that cost(i,j) is either a positive real number or infinity if no edge(i,j) exist. A minimum spanning tree is computed and stored as a set of edges in the array T[n][2], edge in MST is T[i][1] to T[i][2].*/

Step 1: Consider an edge (k, l) with minimum cost.
Step 2: mincost = cost[k][l];
Step 3: T[1][1] = k; T[1][2] = l;
Step 4: for i = 1 to n do { /*initialize NEAR vector */
Step 5: if cost [i][l]< cost[i][k] then
NEAR[i] = l;
else
NEAR[i] = k;
Step 6: /*end of for at step4 */
Step 7: NEAR[k] = NEAR[l] = 0;
Step 8: for i = 2 to n–1 do{ /*find n–2 additional edges for T */
Let j be an index such that NEAR[j] # 0 and cost[j][NEAR[j]] is minimum.
Step 9: T[i][1] = j; T[i][2] = NEAR[j];
mincost = mincost+cost[j][NEAR[j]];
NEAR[j] = 0;
Step 10: for k=1 to n do { /*update NEAR */
if NEAR[k] # 0 and cost [k][NEAR[k]] > cost[k][j]
then NEAR[k] = j;
Step 11: } /*end of for at step 10*/
Step 12: } /*end of for at step 8*/
Step 13: if mincost > = Infinity then print "no spanning tree";
else
return mincost;
Step 14: end Prim

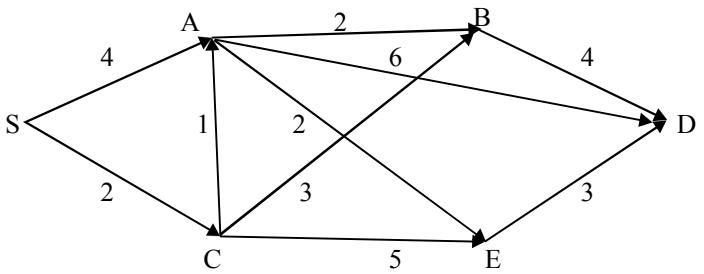
(*b*) Warshal algorithm to compute transitive closure compute shortest path among the all pairs of vertices of graph up to n length. This method is based on dynamic programming. It compute the path length. The following 'C' function compute the path:

```
void  tranclos(int A[n][n], int path[n][n])
{
        int i, j  , k;
        for (i = 0; i <n; i++)
                for (j = 0; j <n; j++)
```

```
                path[i][j] = A[i][j];  /* initial adjacency matrix of graph as path matrix */
for (k = 0; k <n; k++)
        for (i = 0; i <n; i++)
                if (path[i][k] = = 1)
                        for (j = 0; j <n; j++)
                                path[i][j] = path[i][j] || path[k][j];
} /* end tranclos */
```

The efficiency of transitive closure is $O(n^3)$

Problem 29. Write down Dijkstra's algorithm for solving the single source shortest path problem on a weighted, directed graph. Run the algorithm on the following graph for finding the shortest path from S to D.



Give a simple example of a directed graph for which Dijkstra's algorithm produces incorrect answer.

Solution. The main idea of Dijkstra's algorithm is to keep identifying the closest nodes from the source node in order of increasing path cost. The algorithm is iterative. At the first iteration the algorithm finds the closest node from the source node, which must be the neighbor of the source node, if link costs are positive. At the second iteration the algorithm finds the second-closest node from the source node. At the third iteration the third-closest node must be the neighbor of the first two closest nodes and so on. Thus at the kth iteration, the algorithm will have determined the k closest nodes from the source node.

Dijkstra's algorithm can be described as follows:

V = set of vertices in the graph

s = source node (i.e. vertex)

S = set of vertices currently included in shortest path, initially consist of source vertex s.

l (w, v) = link cost from node w to node v, the cost is ∞ if the nodes are not directly connected

D(n) = least cost, from s to n nodes that is currently known to the algorithm.

The steps for the algorithm:

Initialize

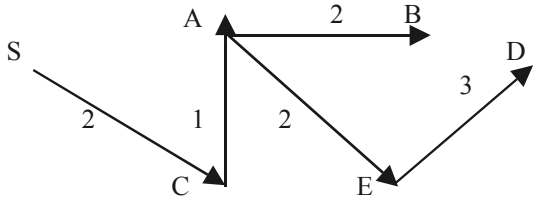
1. set D[s] = 0;
2. for each node in V except source node do D[v] = l (s, v);
3. while S# V do // select minimum cost node
4. begin
5. Choose a node w in V–S such that D[w] is a minimum
6. Add w to S;
7. for each in V–S do // iterate for all nodes to re-compute the cost

D[v] = Min (D[v], D[w]+l(w, v));
end
end Dijkostra

Initialized S is the source node, D[S] = 0, and D[i] is 4, 2, ∝, ∝, ∝ for node A, C, B, E, D respectively. In next iteration select the minimum cost link node that is node C and re-compute the cost for all nodes according to the algorithm.

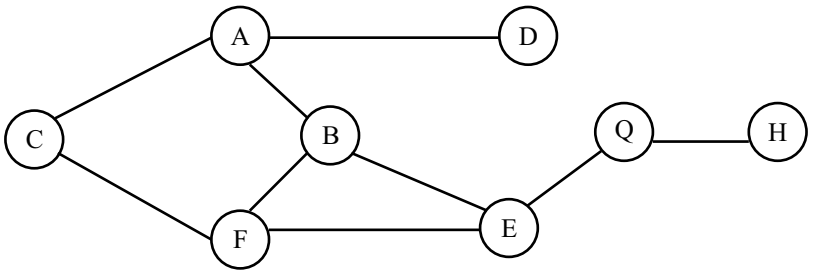
The following figure illustrates the process. The dark rectangle node is marked permanent and others remaining temporary.

Iteration	*Nodes travel currently*	*D[A]*	*D[C]*	*D[B]*	*D[E]*	*D[D]*
1.	{S}	4	2	∝	∝	∝
2.	{S, C}	3	2	5	7	∝
3.	{S, C, A}	3	2	5	5	9
4.	{S, C, A, B}	3	2	5	5	9
5.	{S, C, A, B, E}	3	3	5	5	8
6.	{S, C, A, B, E, D}	3	3	5	5	8



The shortest path from vertex S to D is S–C–A–E–D of 8 length.

Problem 30. Consider the graph 'G' in figure below:



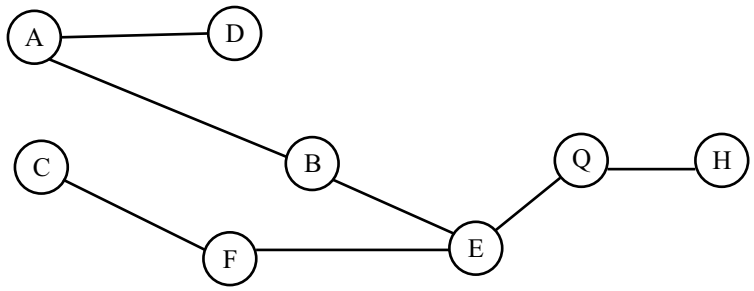
(*i*) Find the adjacency matrix of G

(*ii*) Find the order in which vertices of G are processed using a depth first search algorithm beginning at vertex A.

Solution. (*i*) Adjacency matrix

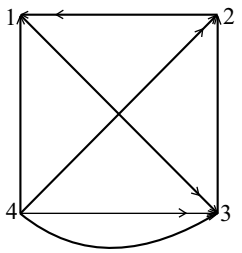
	A	B	C	D	E	F	H	Q
A	0	1	1	1	0	0	0	0
B	1	0	0	0	1	1	0	0
C	1	0	0	0	0	1	0	0
D	1	0	0	0	0	0	0	0
E	0	1	0	0	0	1	0	1
F	0	1	1	0	1	0	0	0
H	0	0	0	0	0	0	0	1
Q	0	0	0	0	1	0	1	0

(*ii*)



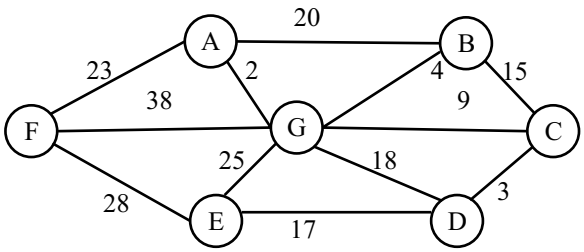
The depth first search sequence is A, B, E, F, C, Q, H, D where A is start vertex.

Problem 31



(*a*) Execute BFS algorithm on the above graph with 3 as the starting node.

(*b*) Explain Kruskal's algorithm for finding minimum cost spanning tree in a graph. Trace your algorithm on the following graph:



Solution. (*a*) The steps of over search follow as start node 3

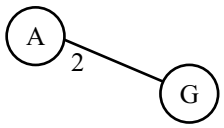
1. Initially, add node 3 to empty queue as follows
   Front = 3 Queue:3
   Rear = 3
2. Delete the front element 3 from queue and add adjacent nodes of 3 to queue which is not visited.
   Front = 2 Queue:2
   Rear = 2
3. Delete the front element 2 from queue and add adjacent nodes of 2 to queue which is not visited.
   Front = 1 Queue:1
   Rear = 1
4. Delete the front element 1 from queue and add adjacent nodes of 1 to queue which is not visited.
   Front = NULL Queue:Empty
   Rear = NULL

The breadth first search is over at 3, 2, 1. It can't travel node 4.

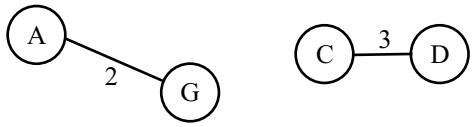
(*b*) The spanning tree obtained is shown below.

(*i*) Initially, each vertex is assigned to a distinct set (and hence to a distinct tree). Here, 9 distinct trees, each one having single vertex itself as root of the tree.

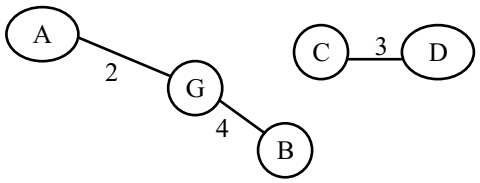
(*ii*) Create minimum heap for the edges with their cost and remove minimum cost edge i.e. (A, G) and is included in T as both vertices are from distinct tree set.



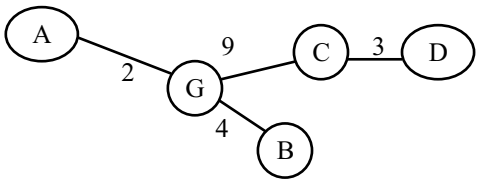
(*iii*) Reheapify the remaining edges with their cost and remove minimum cost edge i.e. (C,D) and is included in T as both vertices are from distinct tree set.



(*iv*) Reheapify the remaining edges with their cost and remove minimum cost edge i.e. (G,B) and is included in T as both vertices are from distinct tree set.

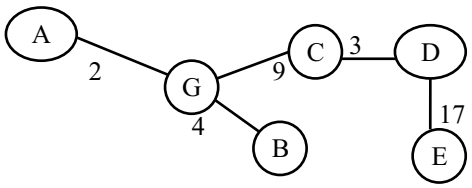


(*v*) Reheapify the remaining edges with their cost and remove minimum cost edge, i.e. (G, C) and is included in T as both vertices are from distinct tree set.



(*vi*) Reheapify the remaining edges with their cost and remove minimum cost edge, i.e. (B, C) and is not included in T as both vertices are not from distinct tree set.

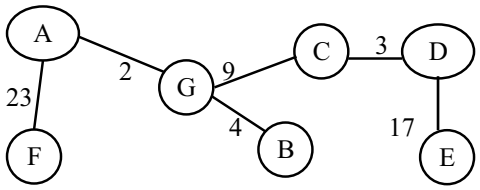
(*vii*) Reheapify the remaining edges with their cost and remove minimum cost edge, i.e. (E, D) and is included in T as both vertices are from distinct tree set.



(*viii*) Reheapify the remaining edges with their cost and remove minimum cost edge, i.e. (G, D) and is not included in T as both vertices are not from distinct tree set.

(*ix*) Reheapify the remaining edges with their cost and remove minimum cost edge, i.e. (A, B) and is not included in T as both vertices are not from distinct tree set.

(*x*) Reheapify the remaining edges with their cost and remove minimum cost edge, i.e. (A, F) and is included in T as both vertices are from distinct tree set.



That is minimum spanning tree.

Problem 32. (*a*) Write binary search algorithm and trace it to search element 91 in the following list:
13, 30, 62, 73, 81, 88, 91

What limitations do we have in binary search?

(*b*) Construct an expression tree for the expression $(-b^2+\sqrt{b^2-4ac})/2a$

Show all the steps and give pre order, in order and post order traversals of the expression tree so formed.

Solution. (*a*) We state the binary search algorithm formally as given below:

Algorithm BSEARCH(v, lb, ub, sElem, loc)

/* Here v is a nElem sorted array with lower bound lb and upper bound ub and sElem is an element to be searched and if found then return the index in loc otherwise return –1. The local variable low, mid, high denote respectively, the beginning, middle and end locations of the array elements */

Step 1 : Initialize variables
Set low = lb; high = ub; mid = (low+high)/2; and loc = –1;
Step 2 : repeat steps 3 and 4 while low< = high and v[mid] != sElem
Step 3 : if sElem < v[mid] then
high = mid –1;
else
low = mid+1;
Step 4 : mid = (low+high)/2;
Step 5 : if v[mid] = sElem then
loc = mid;
Step 6 : return (loc);
Step 7 : end BSEARCH

Note that if sElem does not appear in v array, the algorithm eventually arrives at the stage low = mid = high. In next time when high<low, control transfers to Step 5 of the algorithm.

Let us apply this algorithm to an example. Suppose array v contains 7 elements as

13, 30, 62, 73, 81, 88, 91

and that we wish to search for an element value 91. The algorithm works as follows:
mid = (low+high)/2
Initialize vector v as

(1) Check v[mid] = 91, it is not so search reduces from low = mid +1 to high

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
v[] =	13	30	62	73	81	88	91
	low = 0			mid = 3			high = 6

(2) Check v[mid] = 91, it is not so search reduces from low = mid +1 to high

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
v[] =	13	30	62	73	81	88	91
					low = 4, mid = 5, high = 6		

(3) Check v[mid] = 91, it is not so search reduces from low = mid +1 to high

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
v[] =	13	30	62	73	81	88	91
						low = 5, high = 6, mid = 5	

(4) Check v[mid] = 91, it matches so searches is successful.

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
v[] =	13	30	62	73	81	88	91

low = 6,
high = 6,
mid = 6,

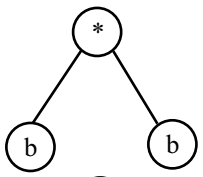
The limitation of binary search is that it can only apply on sorted or ordered data. The binary search can't apply on linked list. The most suitable structure for binary search is array, but the insertion and deletion in array is very costly and ineffective.

(*b*) First convert the expression into arithmetic expression

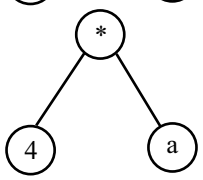
$(-b^2+\sqrt{b^2-4ac})/2a$ = (– b * b + √(b*b – 4 *a*c))/(2*a)

The tree is constructed by solving one operator at a time with the law of precedence and associatively. The whole process is explained below in figure:

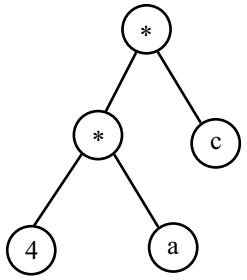
(*i*) b * b ⇒



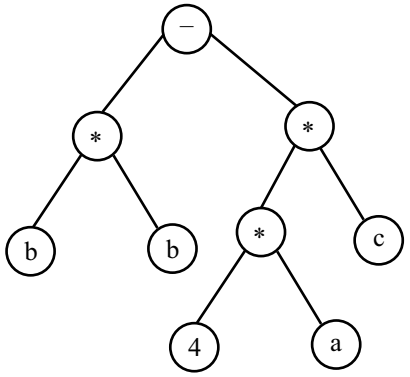
(*ii*) 4 * a ⇒



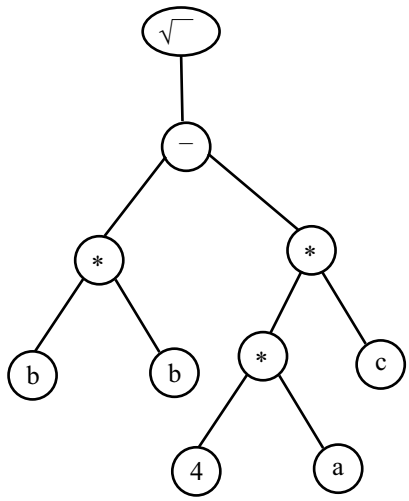
(*iii*) 4*a*c



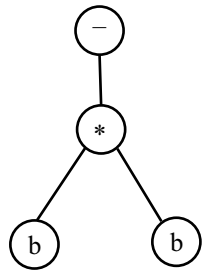
(*iv*) b*b – 4*a*c



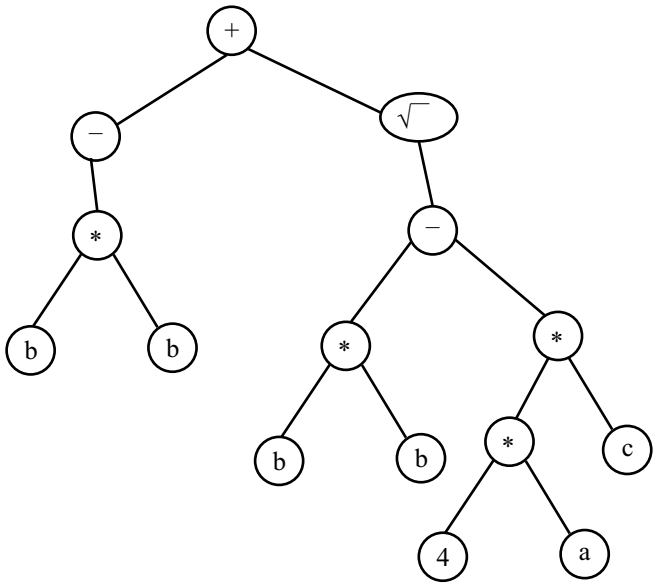
(*v*) √(b*b – 4*a*c)



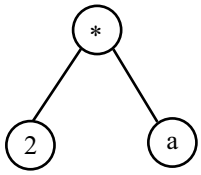
(*vi*) – b * b



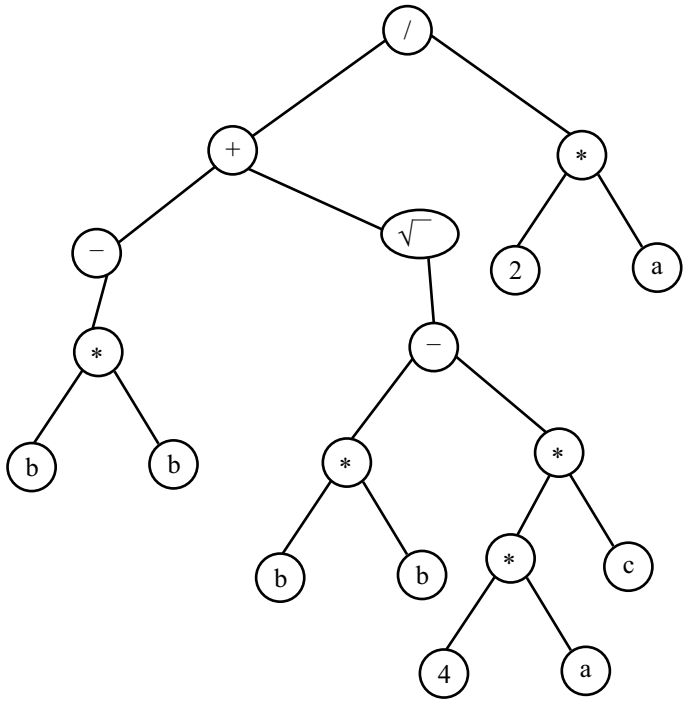
(*vii*) (– b*b+ √(b*b – 4*a*c))



(*viii*) 2 * a



(*ix*) (– b *b+ √(b*b – 4*a*c)) /2*a



Expression tree for (– b *b+√(b*b – 4*a*c))/2*a

The pre order traversal of the tree is /+–*b b √ – * b b * * 4 a c * 2 a

The post order traversal of the tree is b b * – b b * 4 a * c * – √+2 a * /

Problem 33. Consider the following random sequence of data

$X_1, X_2, X_3, \ldots . X_M$, –MININT <= X_i<= MAXINT

and all X's need not be distinct.

We are to find the frequency of occurrence of each data item and to be printed in ascending order of value of data item.

(*i*) Develop a suitable data representation. Give reasons for your choice.

(*ii*) Develop an algorithm to compute the frequency of occurrence.

(*iii*) Write a procedure to print the data and frequency of occurrence in tabular form as required.

(*iv*) Find the time complexity of the procedures in (*ii*) and (*iii*).

(*v*) Can we use the same data representation if the table is to be printed in descending order? If yes how? If no, suggest modification in data representation. (Do not write any procedures, etc).

Solution. (*i*) It is given that the input sequence is random and the occurrence of data item may not be distinct. We can use binary search tree as data representation with one modification that the node will store the frequency of occurrence of data item. As in order traversal of the tree will give the ascending order of data item.

```
struct   nodetype{
        int info;
        int focc;  /* frequency of occurrence of data item whose value is info */
        struct nodetype *lchild;
        struct nodetype *rchild;
}
```

(*ii*) We can develop an algorithm for this just by modifying the binary tree search insertion algorithm. Each new element x, we must first verify that its key is different from those of existing nodes and its frequency of occurrence is one and the element those exist in the tree we just increase its frequency of occurrence by one.

The algorithm BTREEINSF is given below:

Algorithm BTREEINSF(x)

***Step 1*:** [initialize variables]
Set found = 0; set p = root;

***Step 2*:** repeat step 3 while (p != NULL) and (!found)

***Step 3*:**
```
        parent  = p;
        if (p–>Info = x) then
                Set p–>focc = p–>focc +1;
                Set found = 1;
        else if (x < p–>Info) then
                Set p = p–>lchild;
                else
                Set p = p–>rchild;
[end of while loop at step 2]
```

***Step 4*:**
```
if (!found) then
        Set p  =  allocate memory for a node of binary tree
        Set p–>lchild = NULL;
        Set p–>Info = x;
        Set p–>focc = 1;
        Set p–>rchild = NULL;
```

***Step 5*:**
```
        if (root != NULL) then
                if (x <parent–>Info) then
                        Set parent–>lchild = p;
                else
                        Set parent–>rchild = p;
                [End of If Structure]
        else
```

```
                    root  = p;
                [End of If Structure of Step 5]
            [End of If Structure of Step 4]
```

***Step 6*:** end BTREEINSF

(*iii*) The algorithm to print the data item and its frequency of occurrence in ascending order is obtained by in order traversal of binary search tree.

```
void inordero(struct nodetype *bt)
{
printf("\n———————————————");
if (bt)
{
inordero(bt–>lchild);
printf("\n  %d   %d \n",bt–>info, bt–>focc);
inordero(bt–>rchild);
}
}
```

(*iv*) The time complexity of algorithm in (ii) involve insertion of a new node into binary search tree and if node exist then it updates its frequency of occurrence.
Complexity of searching $O(\log_2 m)$.
Complexity of insertion time: $O(\log_2 m)$
To insert m elements the complexity of insertion is $O(m\log_2 m)$
The complexity of algorithm in (iii) is just traversal of each element once. That is O(n).

(*v*) **Yes.** We can use same data representation for descending order. Instead of in-order traversal we use backward in order traversal to print the data item in descending order.
The following procedure to achieve the descending order.
The algorithm to print the data item and its frequency of occurrence in descending order is obtained by backward in order traversal of binary search tree. It traversal right subtree, node and then left subtree recursively.

```
void backinordero(struct nodetype *bt)
{
printf("\n———————————————");
if (bt)
{
backinordero(bt–>rchild);
printf("\n  %d   %d \n",bt–>info, bt–>focc);
backinordero(bt–>lchild);
}
}
```

Problem 34. (*a*) What is recursion? How are values computed for a recursively defined function? Explain with an example? Clearly name the data structure required and operations on the data structures.

(*b*) Convert the following infix expression into prefix expression:

(*i*) A * B /C

(*ii*) A /B ** C+D * E – A * C

Also compute (i) in prefix form for A = 1, B = 4 and C = 2.

Solution. (*a*) Consider an example of factorial

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

Such a definition which defines an object in terms of a simpler case of itself, is called a recursive definition. Here, *n!* is defined in terms of $(n-1)!$ which in turn is defined in terms of $(n-2)!$ and so on, until finally 0! is reached

The basic idea, here is to define a function for all its argument values in a constructive manner by using induction. The value of a function for a particular argument value can be computed in a finite number of steps using the recursive definition, where at each step of recursion, we come nearer to the solution.

To compute factorial of 4 as follows:

4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1!= 1

which further return the factorial of 1, then return factorial of 2 , then 3 and finally factorial of 4.

2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1
= 24

The data structure for recursion is stack.

The operations on stack are pop and push.

(*b*)

(*i*) A * B /C

Convert the infix expression into prefix by converting one by one operator.

*A B / C
/ * A B C

Prefix expression / * A B C

(*ii*) A /B ** C+D * E – A * C

Convert the infix expression into prefix by converting one by one operator.

A / **B C +D * E – A * C
/ A **B C +D * E – A * C
/ A **B C +*D E – A * C
/ A **B C +*D E – *A C

+ / A **B C *D E – *A C

–+/ A **B C *D E *A C

Prefix expression –+/ A **B C *D E *A C

The prefix form for A * B /C where A = 1, B = 4 and C = 2 is:

= / * 1 4 2

= 2

Problem 35. (*a*) Two queues are to be represented in a single array. How ? Explain with neat diagram. Also write the implementation of insertion and deletion algorithm in 'C'.

(*b*) Give algorithm for deletion of a node in a queue using linked representations. Show the critical values of pointer clearly.

Solution. (*a*) It is possible to keep two queues in a single array. One grows from lower bound index other from upper bound index. The front1 and rear1 variable indicates queue 1 and initialize to –1. The front2 and rear2 variable indicates queue 2 and initialize to MAXSIZE.

Queue Condition

Empty

The empty queue1 contains no elements and can therefore be indicated by q.front1 = –1 and q.rear1 = –1.

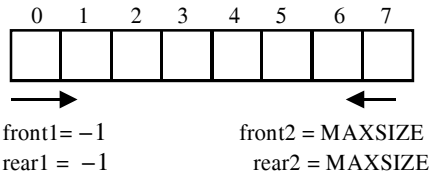
The empty queue2 contains no elements and can therefore be indicated by q.front2 = MAXSIZE and q.rear2 = MAXSIZE.

Full

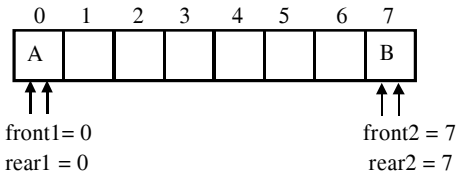
The queue1 is full when rear1 = rear2 –1. The queue2 is full when rear2 = rear1 + 1.

Below figure shows an array representation of queue (for convenience, the array is drawn horizontally rather than vertically). The size of an array is 8.

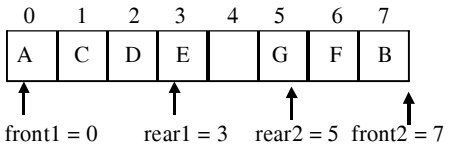
The operations of queue imply that if the elements A, B, C, are inserted into a queue1 and element D and E is inserted in queue2 , in that order, then the first element to be removed (i.e deleted) from queue1 must be A and from queue2 is D



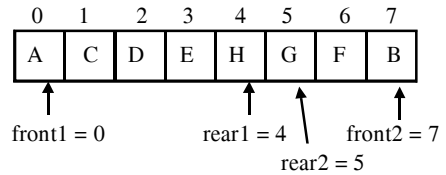
(*i*) initially both queues are empty



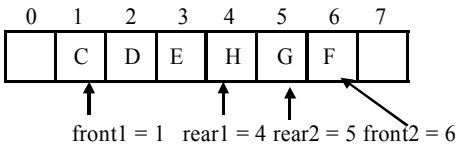
(*ii*) Element A & B is inserted in queue1 and queue 2 respectively



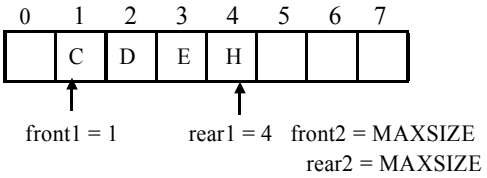
(*iii*) Insertion of C, D and E in queue 1 and element F and G is inserted in queue 2



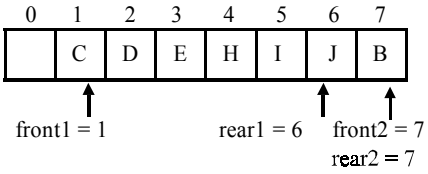
(*iv*) Insertion of H in queue 1 & insertion of an element in queue 2 raises queues full condition



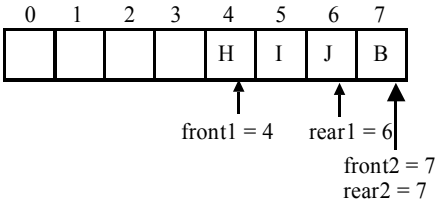
(*v*) Deletion in queue1 and queue2



(*vi*) Deletion of two elements in queue2, the queue2 became empty



(*vii*) Insertion of I and J in queue1 & B in queue2



(*viii*) Deletion of three elements in queue1

The following 'C' program implement the insertion and deletion in the queues. The queueins1 function insert into queue1 and queueins2 function insert into queue2. The deleteq1 delete element from queue1 and deleteq2 delete element from queue2.

The display function traverse the both queues.

```
/* Program twoqueue.c perform insertion, deletion operation on queues which are implemented in a
 array. The queue1 start from lower bound index and queue2 start from upper bound index. Queues
is full when rear1 = rear2 –1. */

   #include<stdio.h>
   #define MAXSIZE 5
   struct  queue{
         int items[MAXSIZE];
         int rear1, rear2;
         int front1, front2;
}q;

void queueins1(struct queue *q ,int x)
{
/*check for queue1 full,if so give a warning message and return */

if (q–>rear1 == q–>rear2–1)
   {
   printf("Queue 1 full\n");
   return;
}
else
{
/* check for queue empty, if so initialized front and rear pointer to zero.
```

```
Otherwise increment rear pointer*/
if (q–>front1 == –1)
{ q–>front1 = 0; q–>rear1 = 0;}
else
    q–>rear1 = q–>rear1+1;
q–>items[q–>rear1] = x;
}
} /* End of queueins function */

int queuedel1(struct queue *q)
 {
 int x;
/*check for queue empty, if so raise a warning message and return*/

if (q–>front1 == –1)
    printf(" Queue is empty\n");
 x = q–>items[q–>front1];

/* if both pointer at the same position then reset the queue to empty position  otherwise increment front
pointer by 1.*/

 if (q–>front1 == q–>rear1)
 { q–>front1 = –1; q–>rear1 = –1;}
 else
          q–>front1 = q–>front1 +1 ;
return x;

}
void queueins2(struct queue *q ,int x)
{
/*check for queue2 full, if so give a warning message and return */

if (q–>rear2 == q–>rear1+1)
    {
    printf("Queue 2 full\n");
  return;
 }
else
{
/* check for queue empty, if so initialized front and rear pointer to zero.
 Otherwise increment rear pointer*/
 if (q–>front2 == MAXSIZE)
 { q–>front2 = MAXSIZE–1; q–>rear2 = MAXSIZE –1;}
 else
   q–>rear2 = q–>rear2 –1 ;
 q–>items[q–>rear2] = x;
```

```
}
} /* End of queueins function */

int queuedel2(struct queue *q)
    {
     int x;
/*check for queue empty, if so raise a warning message and return*/

 if (q–>front2 == MAXSIZE)
          printf(" Queue is empty\n");
 x = q–>items[q–>front2];

 /* if both pointer at the same position then reset the queue to empty position  otherwise increment front
pointer by 1.*/

 if (q–>front2 == q–>rear2)
 { q–>front2 = MAXSIZE; q–>rear2 = MAXSIZE;}
 else
    q–>front2 = q–>front2 –1;
return x;

}

void display(struct queue *q)
    {
    int i;
     if (q–>front1 ! = –1)
          if (q–>front1 < = q–>rear1)
          {
          printf("\n Queue 1 data\n");
          for( i = q–>front1;i< = q–>rear1;i++)
                printf("%d      ",q–>items[i]);
    }
 if (q–>front2 ! = MAXSIZE)
    if (q–>front2 > = q–>rear2)
    {
    printf("\n Queue 2 data\n");
    for( i = q–>front2;i> = q–>rear2;i– –)
          printf("%d       ",q–>items[i]);
    }

} /* end of display function */

void main()
 {
          int ch,t;
          q.rear1 = –1;
```

```
        q.front1 = –1; /* Intially queue1 is empty */
        q.rear2 = MAXSIZE;
        q.front2 = MAXSIZE; /* Intially queue2 is empty */
        do{
        printf("\n Operations on Two Linear Queues in a Array");
        printf(" \n 1. Insert in queue 1");
        printf("\n 2. Insert in queue 2");
        printf("\n 3. Delete in queue 1");
        printf("\n 4. Delete in queue 2");
        printf("\n 5. Display Queues");
        printf("\n 6. Exit");
        printf("\n Select operation ");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1 :
                printf("\nEnter data");
                scanf("%d",&t);
                queueins1(&q,t);
                break;
        case 2 :
                printf("\nEnter data");
                scanf("%d",&t);
                queueins2(&q,t);
                break;
        case 3 :
                queuedel1(&q);
                break;
        case 4 :
                queuedel2(&q);
                break;
        case 5 :
                display(&q);
                break;
        case 6 : break;
        }
  }while(ch! = 6);
} /* end of main */
```

(*b*) The below deletion function deletes the first element of the queue and when front = NULL, it returns queue empty message and exit.

The front pointer variable points to the first element of the queue and advance its pointer when a element is delete. The function for deletion of a node is given below:

```
void deletion()
{
    struct queue *ptr;
     if (front == NULL)
            {
            printf("\n Queue is empty");
            return;
            }
    ptr = front;
    front = front–>next;
    free(ptr);
}
```

Problem 36. (*a*) Write a quicksort algorithm which chooses the median of first, the last and n/2 key values as the pivot. (Total no. of records are n).

(*b*) What is external sorting? Discuss any algorithm suitable for external sorting.

Solution. (*a*) In quicksort algorithm the pivot number is the median value of first, last and n/2 key values. We can write the algorithm just modifying the original quicksort algorithm where the element at first index is the pivot number.

1. Compute pivot number v in the partition algorithm as follows:
   mv = (x[m] +x[p]+x[(m+p)/2]) /3;
   mv is the median value, x is the array, m is first index, p is last index and (m+p)/2 is mid value index.
2. And now no interchange between piovt number v and x[m]. Therefore, eliminate the last two statements.

The modified algorithm is given below:

```
/* function quickr call recursively array until lower bound is smaller than upper bound */
void quickr(int x[],int p,int q,int n)
{
int j;

if(p<q)
{
    j = partition(x,p,q,n);
    quickr(x,p,j–1,n);
    quickr(x,j+1,q,n);
}
}/* end of quickr */

int partition(int x[],int m,int p,int n)
/*
Consider the records in array x[] for the file to be sorted, m denotes the file's lower bound
```

```
and p denotes its upper bound.
*/
{
int v, mv,i = m,j = p,temp;
static  pass;
mv = (x[m] +x[p]+x[(m+p)/2]) /3;
while(i<j)
{
      while(x[i]< = v && i<p)
            i++;
      while(x[j]>v)
            j– –;
      if(i<j)
      {
            temp = x[i];
            x[i] = x[j];
            x[j] = temp;
      }
}
/*
x[m] = x[j];
x[j] = v;
*/
printf(" Completion of Pass %d ",++pass);
display(x,n,j);
return j;
}/* end of partition */
```

(*b*) External sorting method employed to sort records of file which too large to fit in the main memory of the computer. These methods involve as much as external processing (e.g. Disk I/O) as compared to processing in the CPU.

The sorting required the involvement of external device such as magnetic tape, magnetic disk due to the following reasons:

1. The cost of accessing an element is much higher than any computational costs.
2. Depending upon the external device, the method of access has different restrictions.

External storage devices can be categorized into two types based on access method. There are sequential access devices (e.g. magnetic tapes) and random access devices (e.g. disks).

External sorting depends to a large extent on system considerations like the type of device and the number of such devices that can be used at a time.

The lists of algorithms are K-way merge, selection tree and polyphone merging. The general methods for external sorting in the merge sort.

K-way merge strategy

Assuming we have r initial runs (i.e. number of elements). The number of passes required to sort a file using k-way merge is $O(\log_k r)$. Consider an example of two-way merge as illustrated in below figure.

In the example, a buffer size of b = 3 is chosen. The first pass is required to create the initial runs of length 3. From this point the merging process is used to create progressively longer runs until the entire file is ordered.

The storage requirements to perform the merging are two input buffers and one output buffer.

The general strategy for the merging process involves the following two steps:

1. First as many initial runs as possible – let the number of runs be r.
2. Merge the r runs, k at a time, using a merging algorithm that is a generalization of the simple two-way merging.

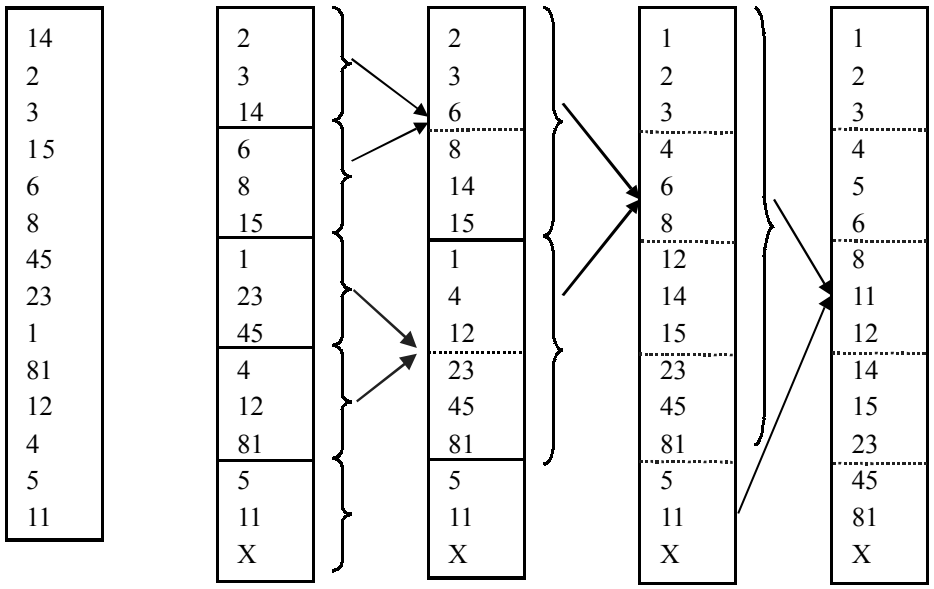
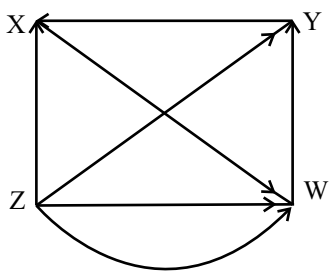


Figure *k-way merge*

Problem 37. Consider the following graph G–



(*i*) Find all simple paths from x to z
(*ii*) Find all simple paths from x to y
(*iii*) Find indeg (y) and outdeg (y)
(*iv*) Find the adjacency matrix A of graph G
(*v*) Find the path matrix P of G using powers of A
(*vi*) Also find the path matrix of G by Warshall's algorithm. Give algorithms whenever required
(*vii*) Is G strongly connected? Give reason for your answer.

Solution. (*i*) The simple paths from vertex x to z are paths of length one, two, three and four.
x – y – w – z of length 3
x – w – z of length 2
(*ii*) The simple paths from vertex x to y are paths of length one, two, three and four.
x – y of length 1
x – w – z – y of length 3
(*iii*) The indegree of y is 2 and outdegree is 1.
(*iv*) Adjacency matrix of graph G is as follows:

A: Adjacency matrix

	X	Y	W	Z
X	0	1	1	0
Y	0	0	1	0
W	0	0	0	1
Z	1	1	1	0

(*v*) Path matrix P of G is as given below:

$P = A^1+A^2+A^3+A^4$

$A^2 = A * A$

	X	Y	W	Z
X	0	0	1	1
Y	0	0	0	1
W	1	1	1	0
Z	0	1	1	1

$A^3 = A^2 * A$

	X	Y	W	Z
X	1	1	1	1
Y	1	1	1	0
W	0	1	1	1
Z	1	1	1	1

$A^4 = A^2 * A^2$

	X	Y	W	Z
X	1	1	1	1
Y	1	1	1	1
W	1	1	1	1
Z	1	1	1	1

$P = A^1+A^2+A^3+A^4$

	X	Y	W	Z
X	1	1	1	1
Y	1	1	1	1
W	1	1	1	1
Z	1	1	1	1

(*vi*) The efficient way to compute transitive closure of a given graph is by **Warshal's algorithm**. Let us define the matrix $path^k$ such that $path^k[i][j]$ is true if and only if there is a path from node i to node j that does not pass through any nodes numbered higher than k.

The following 'C' function compute the closure.

```
void  tranclos(int A[n][n] , int path[n][n])
{
        int i, j, k;
        for (i = 0; i <n; i++)
                for (j = 0; j <n; j++)
                        path[i][j] = A[i][j];  /* initial adjacency matrix of graph as path matrix */
        for (k = 0; k <n; k++)
        for (i = 0; i <n; i++)
                        if (path[i][k] == 1)
                                for (j = 0; j <n; j++)
                                        path[i][j] = path[i][j] || path[k][j]
        } /* end tranclos */
```

Initial path matrix is equal to adjacency matrix A.

	X	Y	W	Z
X	0	1	1	0
Y	0	0	1	0
W	0	0	0	1
Z	1	1	1	0

For k = 0, i = 0 to n–1

Path matrix is similar to above initial path matrix and for the remaining iteration calculate the path matrix as follows:

For k = 1, i = 0 to n–1 and j = 0 to n–1

Path[0][0] = path[0][0] || path[1][0] = 0 || 0 = 0
Path[0][1] = path[0][1] || path[1][1] = 1 || 0 = 1
Path[0][2] = path[0][2] || path[1][2] = 1 || 1 = 1
Path[0][3] = path[0][3] || path[1][3] = 0 || 0 = 0

Path[1][0] = path[1][0] || path[1][0] = 0 || 0 = 0
Path[1][1] = path[1][1] || path[1][1] = 0 || 0 = 0
Path[1][2] = path[1][2] || path[1][2] = 1 || 1 = 1
Path[1][3] = path[1][3] || path[1][3] = 0 || 0 = 0

Path[2][0] = path[2][0] || path[1][0] = 0 || 0 = 0
Path[2][1] = path[2][1] || path[1][1] = 0 || 0 = 0
Path[2][2] = path[2][2] || path[1][2] = 0 || 1 = 1
Path[2][3] = path[2][3] || path[1][3] = 1 || 0 = 1

Path[3][0] = path[3][0] || path[1][0] = 1 || 0 = 1
Path[3][1] = path[3][1] || path[1][1] = 1 || 0 = 1
Path[3][2] = path[3][2] || path[1][2] = 1 || 0 = 1
Path[3][3] = path[3][3] || path[1][3] = 1 || 1 = 1 and remaining is left for exercise.

(*viii*) Graph G is strongly connected, as there is path from each vertex to another vertex as shown by the path matrix.

Problem 38. (*a*) Store the following values in a hash table–

25, 41, 96, 101, 102, 162, 197, 201

use the division method of hashing with a table size of 11. Use the sequential method of resolving collisions.

(*b*) What do you understand by BigO notation? Out of the sorting techniques studied by you, which sorts are $O(n^2)$ and which are $O(n\log_2 n)$? What conditions can make an $O(n^2)$ sort run faster than an $O(n\log_2 n)$ sort.

Solution. (*a*) In division method, key k to be mapped into n locations or indices by finding the modulus or remainder of key k by n.

H(k) = key mod n or key % n

Here n = 11

So h(k) is computed as below:

Key	h(k) = key%n
25	25 % 11 = 3
41	41 % 11 = 8
96	96 % 11 = 8 **collision**
101	101 % 11 = 2
102	102 % 11 = 3 **collision**

162 162 % 11 = 8 **collision**
197 197 % 11 = 10
201 201 % 11 = 3 **collision**

It suffers from severe collision. The sequential or linear probing can solve the collision.

Linear probing resolved hash collisions by sequentially searching a hash table beginning at the location returned by the hash function (using circular array).

The linear probing uses the following hash function.

$$h(k, i) = (h'(k)+i) \bmod n \qquad \text{for } i = 0, 1, 2, \ldots n-1$$

where n is the size of the hash table and h'(k) = k mod n the basic hash function, and i is the probe number.

k	25	41	96	101	102	162	197	201
h(k,0)	3	8	**8**	2	**3**	**8**	10	**3**
h(k, 1)	–	–	9	–	4	**9**		**4**
h(k, 2)	–	–	–	–	–	**10**		5
h(k, 3)	–	–	–	–	–	0		–
h(k, 4)	–	–	–	–	–	–		–

Underline and bold number represent collision and after successive probing we get the location for the keys.

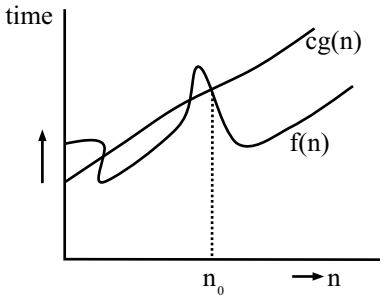
(*b*) O(big oh)–notation:

When we have only an asymptotic upper bound we use notation O(big-oh) for a given function g(n) we denoted by O(g(n)) the set of functions.

$$O(g(n)) = \{ f(n) : \text{there exist positive constant c and } n_0 \text{ such that}$$
$$0 \le f(n) \le c\, g(n) \text{ for all } n > n_0 \}$$

The below figure depicted the asymptotic O–notation. All values n to the right of n_0, the value of the function f(n) is on or below g(n).

We use O–notation to give upper bound on a function to within a constant factor. It is used to bind worse case running time of an algorithm. The time complexity $O(n^2)$ bound worse case for insertion sort algorithm.



f(n) = O(g(n))

Big O–notation

The below sorting algorithm perform sorting in $O(n^2)$ and $O(n\log_2 n)$.

Sorting algorithm	Complexity
Selection Sort	$O(n^2)$
Bubble Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Merge Sort	$O(n\log_2 n)$
Quick Sort	$O(n\log_2 n)$
Heap Sort	$O(n\log_2 n)$
Address calculation sort	$O(n^2)$
Shell sort	$O(n^2)$

The modified bubble sort can take O(n) time when the set elements to be sorted in the same order as we required.

In case of insertion sort, if the initial file is sorted, only one comparison is made on each iteration, so that the sort is O(n).

The modified selection sort can take O(n) time when the set elements to be sorted in the same order as we required.

That means all three cases the sorted data is input and we need the same output.

In address calculation sort if the keys are hashed in order then it take O(n) time in sorting.

Problem 39. (*a*) Suppose a binary tree T is in memory. Write a nonrecursive procedure for each of the following:

(*i*) Finding the number of nodes in T

(*ii*) Finding the number of terminal nodes in T

(*b*) Suppose we are hashing integers with a T-bucket hash table using the hash function h(i) = i mod 7.

(*i*) Show the resulting open hash table if the perfect cubes, 1, 8, 27, 64, 125, 216, 343 are inserted.

(*ii*) Repeat part (i) using a closed hash table with linear resolution of collisions.

Solution. (*a*) (*i*) The non-recursive procedure to compute the number of nodes in a binary tree is given below. The algorithm is based on non-recursive preorder traversing of T.

Algorithm NumofNode(T) //Non Recursive

/* A binary tree T is in memory and array stack is used to temporarily hold address of nodes. The counter node counts the number of nodes*/

Step 1 : [Initialize top of stack with null and initialize temporary pointer to nodetype is ptr]
node = 0;
Top = 0;
stack[Top] = NULL;
ptr = T;

Step 2 : repeat step 3 to 5 while ptr ! = NULL

Step 3 : node = node +1;

```
Step 4 :  [Test existence of right child]
          if (ptr → rchild ! = NULL)
          {
                Top = Top+1;
                stack[Top] = ptr → rchild;
          }
Step 5 : [Test for left child]
          if (ptr → lchild ! = NULL)
                ptr = ptr → lchild;
          else
            [Pop from stack]
                ptr = stack[Top];
                Top = Top–1;
          [End of loop in Step 2.]
Step 6 :  write ("Number of nodes =", node);
Step 7 :  end NumofNode.
```

(*ii*) The non-recursive procedure to compute the number of terminal nodes in a binary tree is given below. The terminal node has no child. The algorithm is based on non-recursive preorder traversing of T.

Algorithm NumofTNode(T)//Non Recursive
/* A binary tree T is in memory and array stack is used to temporarily hold address of nodes. The counter tnode counts the number of terminal nodes*/

```
Step 1 :  [Initialize top of stack with null and initialize temporary pointer to nodetype is ptr]
          tnode = 0;
          Top = 0;
          stack[Top] = NULL;
          ptr = T;
Step 2 :  repeat step 3 to 5 while ptr ! = NULL
Step 3 :  if (ptr–>lchild == NULL) &&  (ptr–>rchild) == NULL)
                    tnode = tnode +1;
Step 4 :  [Test existence of right child]
          if (ptr → rchild ! = NULL)
          {
                    Top=Top+1;
                    stack[Top] = ptr → rchild;
          }
Step 5 :  [Test for left child]
          if (ptr → lchild ! = NULL)
                    ptr = ptr → lchild;
          else
            [Pop from stack]
```

```
                ptr = stack[Top];
                Top = Top–1;
        [End of loop in Step 2.]
```

Step 6: write ("Number of terminal nodes =", tnode);

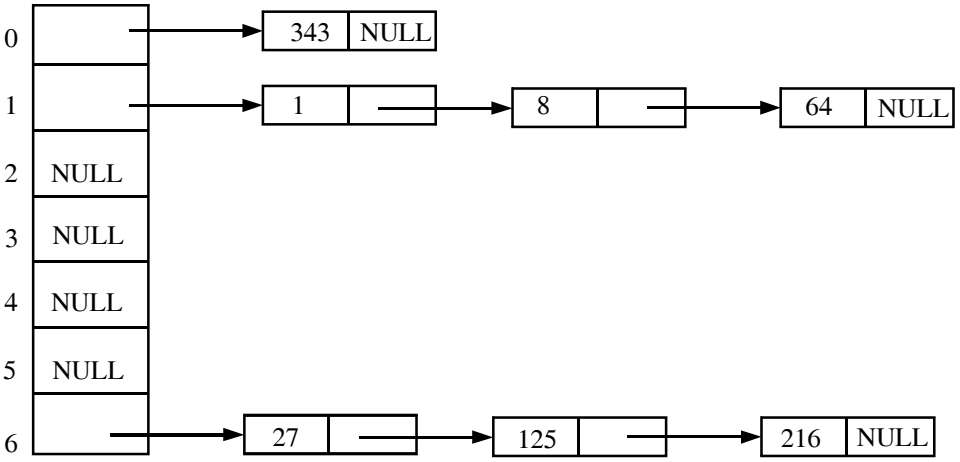
Step 7: end NumofNode.

(*b*)

(*i*) The keys are stored linked list bucket[] .

k	1	8	27	64	125	216	343
h(k) = k mod 7	1	1	6	1	6	6	0

Initially chained hash table bucket is empty.



(*ii*) The linear probing uses the following hash function.

$h(k, i) = (h'(k)+i) \bmod n$ for i = 0, 1, 2, … n–1

where n is the size of the hash table and h'(k) = k mod n the basic hash function, and i is the probe number.

k	1	8	27	64	125	216	343
h(k,0) = k mod7	1	**1**	6	**1**	**6**	**6**	0
h(k,1)	–	2	–	**2**	**0**	**0**	–
h(k,2)	–	–	–	3	**1**	**1**	–
h(k,3)	–	–	–	–	**2**	**2**	–
h(k,4)	–	–	–	–	**3**	**3**	–
h(k,5)	–	–	–	–	4	**4**	–
h(k,6)	–	–	–	–	–	5	–

The final hash table is given below:

Key	h (k)	Number of Probes
1	1	0
8	2	1
27	6	0
64	3	2
125	4	5
216	5	6
343	0	0

Problem 40. (*a*) Write recursive algorithm for each to determine:

(*i*) the number of nodes in a binary tree

(*ii*) the sum of the contents of all nodes in a binary tree

(*iii*) the depth of a binary tree

(*b*) Draw all the possible nonsimilar trees T where:

(*i*) T is a binary tree with 3 nodes

(*ii*) T is a 2-tree with 4 external nodes

Solution. (*a*) (*i*) The following algorithm counts the number of nodes in a binary tree T.

```
        Algorithm countnode(T)      //Recursive
        /* count is initialized to zero */
Step 1 : if ( T != NULL)
        {
              count = count +1;
              countnode (T → lchild);
              countnode (T → rchild);
        }
Step 2 : write ("Number of nodes", count);
Step 3 : end countnode
```

(*ii*) The following algorithm shows the sum of content of nodes in a binary tree T:

```
        Algorithm sumofnode(T)      //Recursive
        /* sum is initialize to zero */
Step 1 : if (T ! = NULL)
        {
              sum = sum+T–>info;
              sumofnode (T → lchild);
              sumofnode (T → rchild);
         }
Step 2 : write ("Sum of nodes contents:", sum);
```

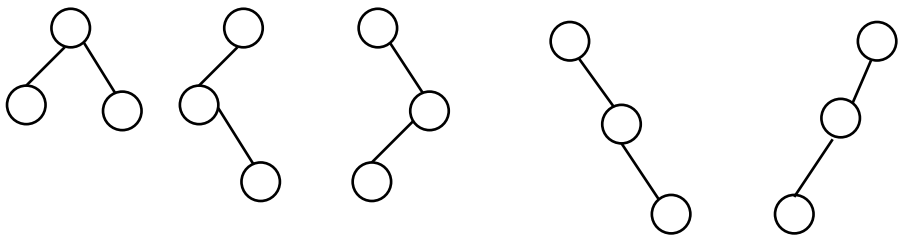
Step 3 : end sumofnode

(*iii*) The depth function return the depth of the binary tree

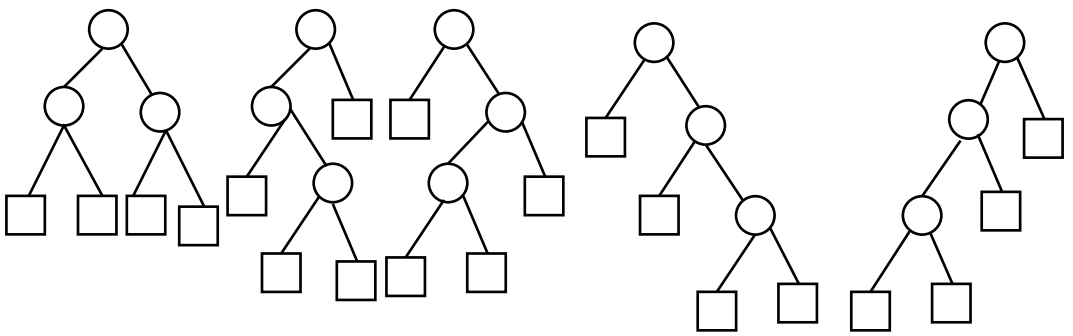
```
int depth(struct nodetype *T, int level)
{
if (T)
{
        depth(T–>lchild, level+1);
        depth(T–>rchild, level+1);
}
if ( level> m)          /* m is global variable initialize to zero */
        m = level;

return (m–1);
}
```

(*b*)

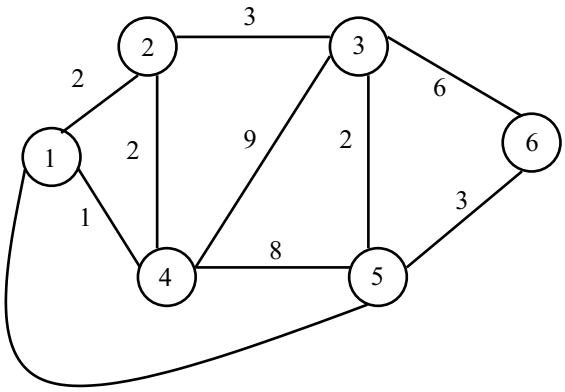
(*i*) There are five such trees, which are shown below:



(*ii*) Each 2-tree with 4 external nodes is determined by binary tree with 3 nodes, i.e., by a tree in part (i). thus there are five such trees, which are given below:



Problem 41. Using Dijkastra's shortest path algorithm, find out a least cost path to all other nodes 1 through 6 for the following graph.



Solution. Dijkastra's routing algorithm find the minimum distance from a source to all destinations. Here, we are finding minimum distance from source node 1 to destination nodes 2, 3, 4, 5 and 6.

Step 1 : Make node l as permanent and all destination nodes as temporary.

Step 2 : Find the distance from node 1 to each destination node. The distance for node 2 is l (1,2) = 2, for node 3 is l(1,3) = ∞, as there is no direct edge, and so on.

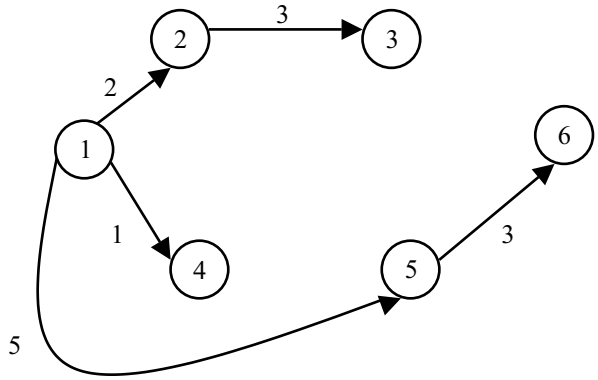
Step 3 : Mark the minimum distance node as permanent. Now find the distance from node 1 to all other temporary nodes through the new permanent node and select the minimum distance between new one and old one.

Step 4 : Repeat step 4 until all nodes marked permanent.

The following table shows the calculations:

Nodes mark permanent	***Minimum distance from node 1 to destination nodes (2, 3, 4, 5, 6)***				
	D[2]	***D[3]***	***D[4]***	***D[5]***	***D[6]***
1	2	∞	1	5	∞
1, 4	2	3	1	5	∞
1, 4, 2	2	3	1	5	∞
1, 4, 2 ,3	2	3	1	5	11
1, 4, 2, 3, 5	2	3	1	5	8
1, 4, 2, 3, 5, 6	2	3	1	5	8

The shortest path tree for route 1 to through 6 to all other nodes.



This page intentionally left blank

Index

This page intentionally left blank

CD–Index

This book contains CD which includes source code of different C\C++ programs. These Programs are helpful to the users to implement and use different types of data structure algorithms. This CD has 9 folders and each folder has a set of program related to a topic. Below are the details of the folders and programs in those folders.

Folder Name	Program Name	Description	Page No.
	POLY3ADD.CPP	Adds two polynomial of 3 variables	157
	POLY3INS.CPP	Inserts a polynomial of 3 variables	144
	POLYADD.CPP	Adds two polynomial	151
	POLYIALL.CPP	Inserts a polynomial in a single variable list	140
	SINGLELL.CPP	Data structure on linear linked list	93
	UNIONLST.CPP	Union two linear linked list elements	167
5. Queue	CIRQUEUE.CPP	Circular queue using array	219
	DEQUE.CPP	Double ended queue using array	223
	MULTIQUE.CPP	Multiple queue with priority using array queue	231
	PQUEUELL.CPP	Priority queue using linked list	228
	QUEUE.CPP	Linear queue	212
	QUEUELL.CPP	Queue operation using linked list	215
6. Search	FREELIST.CPP	Implements dynamic memory allocation	455
	HASHDH.C	Inserts keys into an array with double hashing	445
	HASHLP.C	Inserts keys into hash value using division method	440
	HASHQP.C	Inserts keys into an array with quadratic probing	443
	HASHSCR.C	Collision resolution technique for linked list	449
7. Sort	BUBBLE.C	Bubble sort	not in Book
	HEAP_B_U.C	Heap sort using bottom up heap	421
	HEAP_T_D.C	Heap sort using top down heap	419
	Insertion.C	Insertion sort	not in Book
	INSSORT.C	Insertion sort for 10 integers in an array	384
	MBUBBLE.C	Sort the list of integers into ascending order using bubble sort	374
	MERGES.C	Simple merge sort	not in Book
	MERGESOR.C	Merges two sort sets	400
	MERGLIST.C	Merges two sorted lists into one list	397
	MSELSORT.C	Modified selection sort	not in Book

Folder Name	Program Name	Description	Page No.
	QUICK.C	Quick sort in ascending order	not in Book
	QUICK_NR.C	Non-recursive quick sort	not in Book
	QUICK_RE.C	Recursive quick sort	405
	RADIX.C	Radix or bucket sort	393
	SELECT~1.C	Selection sort	not in Book
	SELSORT.C	Sorts the list using selection sort	377
	SHELSORT.C	Shell sort	388
8. Stack	EVALPOST.CPP	Evaluating a postfix expression	187
	FIBSTACK.CPP	To find Fibonacci term with explicit stack	199
	HANOI.CPP	Tower of Hanoi with implicit recursion	204
	INPOSTFIX.CPP	Convert Infix to Postfix	193
	RECSTACK.CPP	To remover recursion from stack	206
	STACKLL.CPP	Program as linear linked list	180
9. Tree	AVLINDEL.CPP	Insert Nodes in AVL TREE	not in Book
	BSTREEOP.CPP	Create Binary TREE and find height of the tree	285
	BTREECRT.CPP	Create Binary TREE and find height of the tree, and search the tree	not in Book
	BTREEDEL.CPP	Insert nodes, delete nodes in binary search tree	not in Book
	TREE.CPP	Construct a general tree	264

COPYRIGHT