Data Structures & Analysis

Chapter 1 : Introduction to Data Structures and Analysis

Q. 1 Define an algorithm. What are characteristics of an algorithm ?

Ans. :

Definition and Characteristics

The field of computer science revolves around writing of programs for several problems for various domains. A program consists of data structures and algorithms. An algorithm is a set of steps required to solve a problem. These steps are performed on a sample data representing an instance of the problem. Thus an algorithm maps a set of input data (from input domain) to a set of output data (in output domain) through a sequence of operations. An algorithm must have the following properties :

(1) **Input :** Input data, supplied externally (zero or more).
(2) **Output :** Result of the program.
(3) **Finiteness :** In every case, algorithm terminates after a finite number of steps.
(4) **Definiteness :** The steps should be clear and unambiguous.
(5) **Effectiveness :** An algorithm should be written using basic instructions. It should be feasible to convert the algorithm in a computer program.

Take the problem of finding the GCD (Greatest Common Divisor) of two positive integers an example.

Inputs to the algorithm are two positive integers. Output is a positive integer which is GCD of two positive integers given as input.

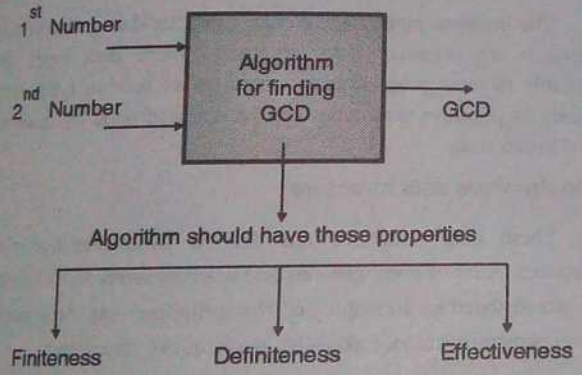


Fig. 1.1

The algorithm is described as a series of steps of basic operations. These steps must be performed in a sequence. Each step of the algorithm is labeled.

Step 1 : Read two positive integers and store them in x and y.
Step 2 : Divide x by y. Let the remainder be r and the quotient be q.
Step 3 : If r is zero then go to step 7.
Step 4 : Assign y to x.
Step 5 : Assign r to y.
Step 6 :goto step 2.
Step 7 : Print y (the required GCD)
Step 8 :stop

The steps mentioned in the above algorithm are simple and unambiguous. Anybody, carrying out these steps will clearly know what to do in each step. Hence, the above algorithm satisfies the **definiteness** property of an algorithm.

Table 1.1 depicts step-wise execution of the above algorithm on two input numbers 15 and 9.

Table 1.1

Steps performed	Value of the variables				
	x	y	r	q	
Step 1	15	9	–	–	Input data supplied externally
Step 2	15	9	6	1	
Step 3	15	9	6	1	
Step 4	9	9	6	1	
Step 5	9	6	6	1	
Step 6	9	6	6	1	
Step 2	9	6	3	1	
Step 3	9	6	3	1	
Step 4	6	6	3	1	
Step 5	6	3	3	1	
Step 6	6	3	3	1	
Step 2	6	3	0	2	
Step 3	6	3	0	2	
Step 7	→ Print the output (GCD) as 3.				

Q. 2 Write an algorithm for adding 10 numbers.

Ans. : Algorithm as a series as steps :

Flow chart

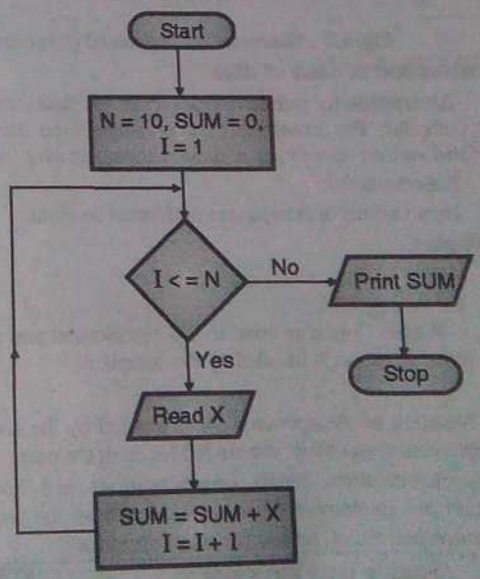


Fig. 1.2 : Flowchart for adding 10 numbers

Step 1 : Assign 10 to N
Step 2 : Assign 0 to SUM

Step 3 : Assign 1 to I
Step 4 : if (I > N) go to step 9
Step 5 : read X
Step 6 : Assign SUM + X To SUM
Step 7 : Assign I + 1 to I
Step 8 : Go to step 4
Step 9 : Print SUM
Step 10 : Stop

Q. 3 What is Abstract Data Type ? Dec. 2013

Ans. :

Abstract data type

The concept of abstraction is commonly found in computer science. A big program is never written as a monolithic piece of program, instead it is broken down in smaller modules (may be called a function or procedure) and each module is developed independently.

When the program is hierarchical organized as shown in the Fig. 1.3, then the "main program" utilizes services at the functions appearing at level 1. Similarly, functions written at level 1 utilizes services of functions written at level 2. Main program uses the services of the next level function without knowing their implementation details. Thus a level of **abstraction** is created. When an abstraction is created at any level, our concern is limited to "what it can do" and not "how it is done".

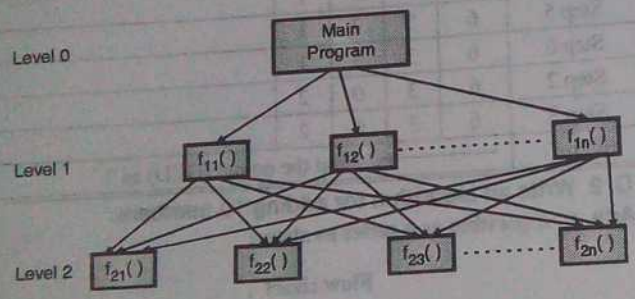


Fig. 1.3 : Hierarchical organized program

Abstraction in case of data

Abstraction for primitive types (char, int, float) is provided by the compiler. For example, we use integer type data and also, perform various operations on them without knowing them :

(1) Representation
(2) How various operations are performed on them.

Example :

```
int x, y, z ;
x = 13 ;
```

Constant 13 is converted to 2's complement and then stored in x. Representation is handled by the compiler

```
x = y + z ;
```

Meaning of the operation '+' is defined by the compiler and its implementation details remain hidden from the user.

Implementation details (representation) and how various operation are implemented remain hidden from the user. User is only concerned about, how to use these operations.

Q. 4 What is data structure ? Dec. 2013

Ans. :

A data structure is merely an instance of an ADT.

An ADT or data structure is formally defined to be a triple (D,F,A) where "D" stands for a set of Domains, "F" denotes the set of operations and "A" represents the axioms defining the functions in "F".

An example of the data structure "Natural Number (NATNO)".

Structure NATNO

Operations

1. ZERO( ) → natno
2. ISZERO(natno) → boolean
3. SUCC(natno) →natno
4. ADD(natno, natno) → natno
5. EQUAL (natno, natno) →boolean

Axioms : For all x, y ∈ natno let

6. ISZERO(ZERO) is true
7. ADD(ZERO,Y) is Y
8. EQUAL(x, ZERO), if ISZERO(x) then true else false

D = { natno, boolean}

F = {ZERO, ISZERO, SUCC, ADD, EQUAL}

A = { Line no 6 to 8 of the structure NATNO}

Q. 5 What are primitive and non-primitive data structure ?

Ans. :

Primitive data structure

The integers, reals, logical data, character data, pointers and reference are primitive data structures. These data types are available in most programming languages as built in type. Data objects of primitive data types can be operated upon by machine level instructions.

Non-Primitive data structure

These data structures are derived from primitive data structures. A set of homogeneous and heterogeneous data elements are stored together. Examples of Non-primitive data structures : Array, structure, union, linked-list, stack, queue, tree, graph.

Some at the most commonly used operations that can be performed on data structures are shown in Fig. 1.4.

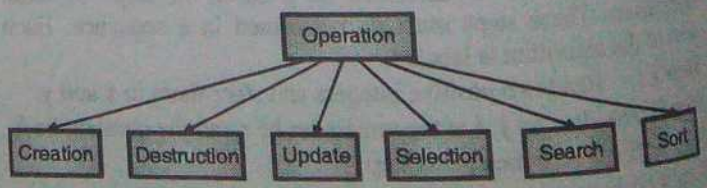


Fig. 1.4 : Data structure operations

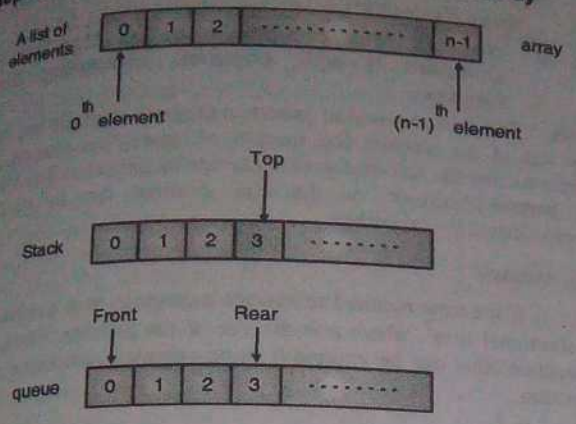
Q. 6 What are linear and non-linear data structure ? May 2015, Dec. 2015, May 2016, Dec. 2016, May 2017

Ans. :

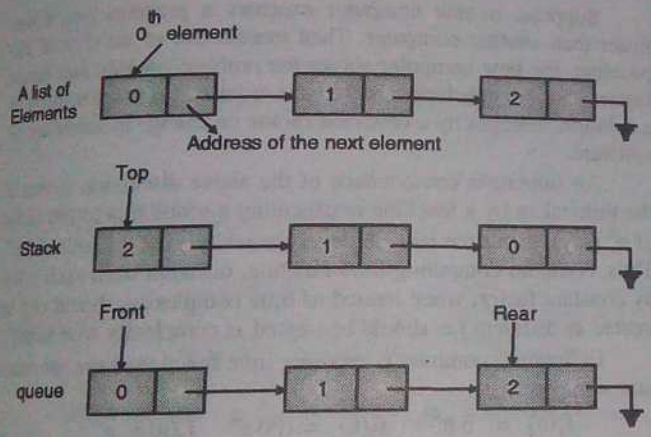
Linear data structure

Elements are arranged in a Linear fashion (one dimension). All one-one relation can be handled through Linear data structures. Lists, stacks and queues are examples of linear data structure.

Representation of Linear data structures in an array

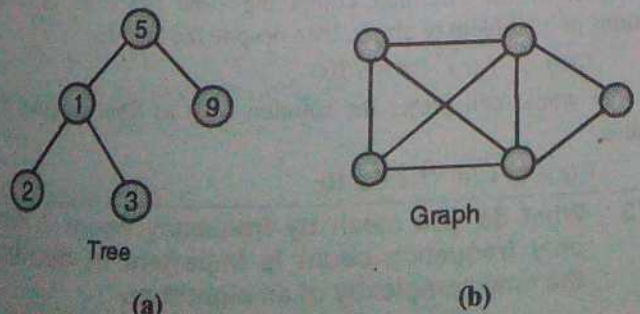


Representation of Linear data structures through Linked structure



Non-Linear data structure

All one-many, many-one or many–many relations are handled through non-linear data structures. Every data element can have a number of predecessors as well as successors. Tree graphs and tables are examples of non-linear data structures.

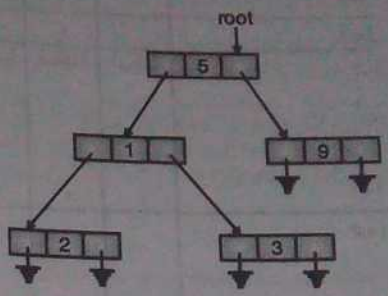


Table

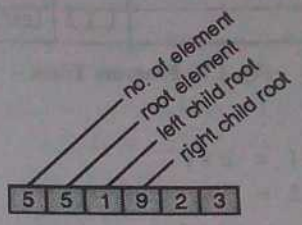
5	9	1
6	4	13
2	5	0
9	6	11

(c)

Fig. 1.5 : Non- Linear data structures



(a) Representation of the binary tree through linked structure



(b) Representation of the binary tree through an array

Fig. 1.6 : Representation of tree of Fig. 1.6(a)

Q. 7 Determine the frequency count for all statements in the following program segment

```
1. i = 1;
2. while ( i <= n)
   {
3. x = x + 1;
4. i = i + 1 ;
   }
```

Ans. :

Statement No.	Frequency
1	1
2	n + 1
3	N
4	n.

Q. 8 Determine the frequency counts for all statements in the following program segment.

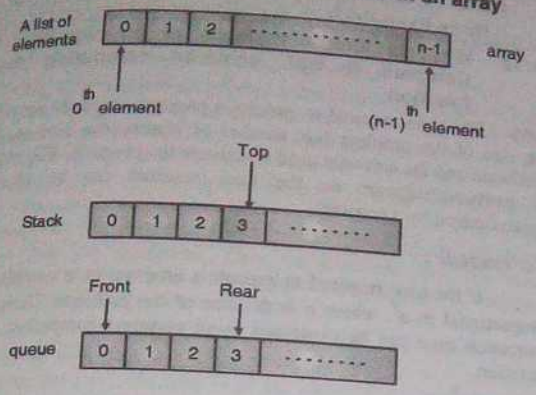
```
1. for (i = 1 ; i <= n ; i++)
2. for (j = 1 ; j <= i ; j++)
3. x = x +1 ;
```

Ans. :

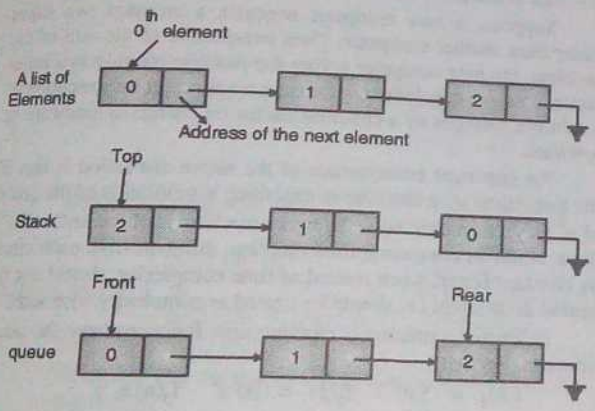
Linear data structure

Elements are arranged in a Linear fashion (one dimension). All one-one relation can be handled through Linear data structures. Lists, stacks and queues are examples of linear data structure.

Representation of Linear data structures in an array

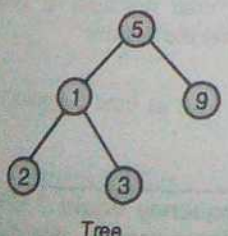


Representation of Linear data structures through Linked structure



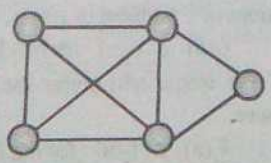
Non-Linear data structure

All one-many, many-one or many–many relations are handled through non-linear data structures. Every data element can have a number of predecessors as well as successors. Tree graphs and tables are examples of non-linear data structures.



Tree

(a)



Graph

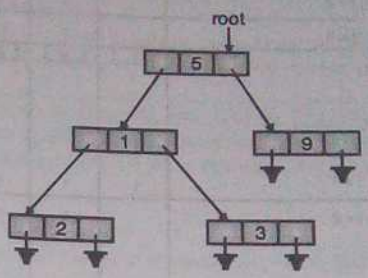
(b)

Table

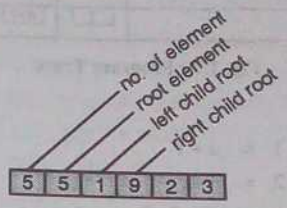
5	9	1
6	4	13
2	5	0
9	6	11

(c)

Fig. 1.5 : Non- Linear data structures



(a) Representation of the binary tree through linked structure



(b) Representation of the binary tree through an array

Fig. 1.6 : Representation of tree of Fig. 1.6(a)

Q. 7 Determine the frequency count for all statements in the following program segment

```
1. l = 1;
2. while ( i <= n)
   {
3. x = x + 1;
4. l = l + 1 ;
   }
```

Ans. :

Statement No.	Frequency
1	1
2	n + 1
3	N
4	n.

Q. 8 Determine the frequency counts for all statements in the following program segment.

```
1. for (i = 1 ; i <= n ; i++)
2. for (j = 1 ; j <= i ; j++)
3. x = x +1 ;
```

Ans. :

		Frequency		
		Statement No.		
		1	2	3
i = 1	j = 1	1	1	1
	2		1	0
		1	2	1
i = 2	j = 1	1	1	1
	j = 2		1	1
	j = 3		1	0
		1	3	2
i = n	j = 1	1	1	1
	j = 2		1	1
	j = n		1	1
	j = n+1		1	0
		1	n+1	n

Total

Fig. 1.7 : Program Trace

Frequency :

Statement No. 1 = $n + 1$

Statement No. 2 = $2 + 3 + \ldots + n + (n+1)$

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

$= n\frac{(n+1)}{2} + n = \frac{1}{2}(n^2 + 3n)$

Statement No. 3 = $1 + 2 + \ldots n = n\frac{(n+1)}{2} = \frac{1}{2}(n^2 + n)$

Q. 9 Define time complexity.

Ans. : Time complexity : Running time of a program can be judged on the basis of factors such as :

1. Input to the program.
2. Size of the program.
3. Machine language instruction set.
4. The machine we are executing on.
5. Time required to execute each machine instruction.
6. The time complexity of the algorithm of the program.

Q. 10 $f(x) = x^2 + 5x$

$g(x) = x^2$ let us take $c = 2$

Ans. :

x	$x^2 + 5x$	$2x^2$	
1	5	2	
2	14	8	$f(x) \le c\,x^2$
5	50	50	For $x \ge 5$
			(asymptotic behavior)

$f(x) \le cg(x)$ for all $x \ge k$ where $c = 2$ and $k = 5$

Q. 11 For the function defined by $f(x) = 5x^3 + 6x^2 + 1$ show that $f(x) = O(x^3)$

Ans. : $f(x) = 5x^3 + 6x^2 + 1 \le 5x^3 + 6x^3 + 1x^3$ for all $x \ge 1$

or, $f(x) \le 12x^3$ for $x \ge 1$

(by replacing each term of x by the highest degree term x^3)

there exist $C = 12$ and $k = 1$ such that

$f(x) \le C\,x^3$ for all $x \ge k$

Hence $f(x)$ is $O(x^3)$

Q. 12 What is time complexity of an algorithm ? Compare it with common computing time function.

Ans. : The time required to execute a program, depends not only the size of the problem (i.e. number of inputs), but also on the hardware and the software used to execute the program. The effect of hardware/software on the time required can be closely approximated by a constant.

For example :

If the time required to execute a program on a machine is proportional to n^2, where n is the size of the problem. Then, the execution time can be expressed using common computing time function.

$f(n) = Kn^2$

$f(n)$ – Execution time

K– A constant, approximating the effect of hardware/software.

n – Size of the problem (number of inputs)

Suppose, a new computer executes a program two times faster than another computer. Then irrespective of the size of the problem, the new computer solves the problem roughly two times faster. Thus we conclude that the time requirement for execution of a solution, changes by a **constant factor** on change in hardware or software.

An important consequence of the above discussion is that if the time taken by a machine in executing a solution is of the order of n^2 (say), then time taken by every machine is of the order of n^2. Thus, common computing time function, different from each other by constant factor, when treated as time complexity, should not be treated as different i.e. should be treated as complexity wise same.

Following, common computing time functions have the same time complexity :

$f_1(n) = 5n^2$ $f_2(n) = 100n^2$ $f_3(n) = n^2$

Asymptotic consideration

When comparing time complexities $f_1(n)$ and $f_2(n)$ of two different solutions of a problem of size n, we need to consider and compare the behaviors of the two functions only for large of n.

For example, if the two functions :

$f_1(n) = 100n^2$ and

$f_2(n) = 10n^3$

represent the common computing time functions of two solutions of a problem of size n, then despite the fact that

$f_1(n) \ge f_2(n)$ for $n \le 10$

We would still prefer the solution $f_1(n)$ as time complexity because.

$f_1(n) \le f_2(n)$ for $n \ge 10$

Q. 13 What do you mean by frequency count ? Why only frequency count is important in deciding the time complexity of an algorithm.

Ans. : Frequency count : Total time taken by a statement in execution depends on :

i) The amount of time a single execution will take.

ii) Number of times the given statement will be executed.

The product of these numbers will be the total time taken by the statement. Number of times a statement will be executed is called its frequency count. Frequency count may vary from data set to data set.

Let us consider an algorithm having a timing complexity given by the function $f(n) = 100\ n^2$

[100 – a constant, n – size of the problem].

The time required for solving a problem, depends not only on the size of the problem but also, on the hardware and software used to execute the solution. The effect of hardware and software on the time required may closely be approximated by a constant.

Suppose, a new computer executes a program two times faster than another computer. Then irrespective of the size of the problem, the new computer solves the problem roughly two times faster than the computer. Thus we conclude that the time requirement for execution of a solution, changes by a constant factor on change in hardware or software.

An important consequence is that if the time taken by one machine in executing a solution is of the order of n^2 (say), then time take by every machine is of the order of n^2. Thus, function different from each other by constant factor, when treated as time complexities, should not be treated as different. i.e. should be treated as complexity wise same.

Following functions have the same time complexities :

$f_1(n) = 5n^2$; $f_2(n) = 100n^2$

$f_3(n) = 1000n^2$; $f_4(n) = n^2$

Time complexity of $f_1(n)$ = Time complexity of $f_2(n)$
= Time complexity of $f_3(n)$
= Time complexity of $f_4(n)$

Q. 14 Greatest Common Divisor (GCD) is defined as :

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

Calculate GCD (3,16)

What are the time and space requirement of your algorithm ?

Ans. : (i) Calculation of GCD (3, 16)

GCD (3, 16) = GCD (16, 3) as m < n
= GCD (3, 16 mod 3) = GCD (3, 1)
= 1, as 3 mod 1 = 0

(ii) Time requirement

This algorithm works by continually finding remainders until 0 is reached. Estimation of running time of the algorithm depends on the length of the sequence of remainders. To get the upper bound on the length of the sequence of remainders, it will be necessary to consider the following relation ;

if m > n, then m mod n < m/2

For example :

$15 \bmod 7 = 1 \left[1 < \frac{15}{2}\right]$; $15 \bmod 9 = 6 \left[6 < \frac{15}{2}\right]$

If m is divided by n, remainder will always be less than m/2. There are two cases.

Case I : $n \le m/2$

Since the remainder is smaller than n.

m mod n < m/2

Case II : n > m/2

Since the remainder will be m–n.

m mod n < m/2

The identity, if m > n, then m mod n < m/2 can be used to conclude that after two iterations of the given algorithm, the remainder is at most half of its original value. Thus the number of iterations is at most 2 log n.

Timing complexity of the algorithm = O (log n)

Space requirement :

Space requirement will be proportional to the maximum length of the stack during recursion. This will again be proportional to log n.

Q. 15 If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ where f_1 and f_2 are positive functions of n, show that the function $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n))$

Ans. :

Since, $f_1(n)$ is $O(g_1(n))$

There exist two positive constants c_1 and k_1 such that

$f_1(n) \le c_1 g_1(n)$ for all $n \ge k_1$ ...(1)

Similarly,

$f_2(n) \le c_2 g_2(n)$ for all $n \ge k_2$ ...(2)

From (1) and (2)

$f_1(n) + f_2(n) \le c_1 g_1(n) + c_2 g_2(n)$ for $n \ge \max(k_1, k_2)$

Case I : $g_1(n) > g_2(n)$

$f_1(n) + f_2(n) \le c_1 g_1(n) + c_2 g_2(n)$ for $n \ge \max(k_1, k_2)$

$\le (c_1 + c_2)\ g_1(n)$ for $n \ge k_1$

$\therefore f_1(n) + f_2(n) = O(g_1(n))$

Case II : $g_2(n) > g_1(n)$

$f_1(n) + f_2(n) \le c_1 g_1(n) + c_2 g_2(n)$ for $n \ge \max(k_1, k_2)$

$\le (c_1 + c_2)\ g_2(n)$ for $n \ge k_2$

$\therefore f_1(n) + f_2(n) = O(g_2(n))$

Thus, we can conclude that :

$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$

Q. 16 Calculate time complexity for given expression.

```
for (k = 0; k< n; k++)
{
    rows[k] = 0;
    for (j = 0; j < n; j++)
    {
        rows[k] = rows [k] + matrix [k] [j];
        total = total + matrix [k] [j];
    }
}
```

Ans. :

The time complexity of the given expression :

```
1.  for (k = 0; k < n; k++)
    {
2.      rows [k] = 0;
3.      for (j = 0; j < n; j++)
        {
4.          rows [k] = rows [k] + matrix [k] [j]
5.          total = total + matrix [k] [j];
        }
    }
```

Frequency count of various statements :

Statement No.	Frequency count	Complexity
1.	n + 1	O(n)
2.	n	O(n)
3.	n (n + 1)	$O(n^2)$
4.	n^2	$O(n^2)$
5.	n^2	$O(n^2)$

∴ **Time complexity** = max (O(n), O(n), $O(n^2)$, $O(n^2)$, $O(n^2)$) = $O(n^2)$

∴ **Space complexity** = space required to store the two arrays rows [] and matrix [] [], plus the memory needed for other variables.

Q. 17 Write short note on : Asymptotic Notation .

Dec. 2016, May 2017

Ans. :

Different algorithm asymptotic notation

(i) The Notation O

The function f(x) is said to be O(g(x)) (pronounced as big – oh) if there exist two positive integer/real number constants C and K such that

$f(x) \leq Cg(x)$ for all $x \geq K$

Example :

if $f(x) = 2x^3 + 3x^2 + 1$

then $f(x) \leq 6x^3$ for all $x \geq 1$

∴ $2x^3 + 3x^2 + 1 = O(x^3)$ as

$f(x) \leq C.\ x^3$ for all $x \geq K$ where C = 6 and K = 1

The notation O provides asymptotic upper bound for a given function.

The purpose of the asymptotic growth rate notations and functions denoted by them is to facilitate the recognition of essential character of a complexity function through some simpler function.

(ii) The Ω Notation

This provides asymptotic lower bound for a given function. A function f(x) is said to be Ω (g(x)) (pronounced as big-omega) if there exist two positive integer/real number constants C and K such that

$f(x) \geq C\ (g(x))$ whenever $x \geq K$

Example :

if. $f(x) = 2x^3 + 3x^2 + 1$

then $f(x) \geq C\ x^3$ for all $x \geq 1$, C = 1

∴ $2x^3 + 3x^2 + 1 = \Omega\ (x^3)$

(iii) The Notation θ

This provides simultaneously both asymptotic upper bound and asymptotic lower bound for a given function.

A function f(x) is said to be θ (g(x)) (pronounced big theta) if, there exist positive constants C_1, C_2 and K such that

$C_2\ g(x) \leq f(x) \leq C_1\ g(x)$ for all $x \geq K$.

For any two functions f(x) and g(x), f(x) = θ (g(x))

If and only if f(x) = O(g(x)) and f(x) = Ω (g(x))

Q. 18 Calculate the worst case time complexity of the following program (Function for binary search) :

```
int binsearch (int a [ ], int i, int j, int x)
{   int k;
    k = (i + j)/2;
    while (i <= j)
    {   if (x > a [k])
            i = k + 1;
        else
            if (x < a [k])
            j = k – 1;
        else
            return (k);
    }   k = (i + j)/2;
        return (– 1);
}
```

Ans. :

Binary search requires that elements are sorted in ascending order. i is the starting index (i.e. 0), j is the index of the last element (n – 1). k gives the index of the centre element

2	5	9	8	11	13	15	← element
0	1	2	3	4	5	6	← index
↑i			↑k			↑j	

i = 0, j = 6, K (0 + 6)/2 = 3

Element to be searched is x.

Inside the while-loop, if the element x is found at the location k (centre), we return k. Inside the while-loop, if the element x is larger than the centre element, we select the second half of the array. Inside the while-loop, if the element x is smaller than the centre element, we select the first half of the array.

2	5	9	8	11	13	15
↑ i = 0		First half	↑ Centre element k = 3	Second half		↑ j = 6

Element to be searched, x = 13

11	13	15	}	Second half of the element		
↑ i = 4		↑ j = 6	}			

In order to calculate time complexity, binary search for n elements must be expressed in terms of binary search of fewer number (<n) of elements.

$f(n) = 1 + f(n/2)$

If there are 128 elements, then the maximum number of elements between i and j after every iteration will be 64, 32, 16, 8, 4, 2, 1, 0, – 1

$128 = 2^7$

Number of iteration required (say h) = 9 = 7 + 2

$\therefore 2^{h+2} = n$

or $\log (2^{h+2}) = \log n$

or $h + 2 = \log n$

$\therefore\ h = \log n - 2 = O(\log n)$

Q. 19 What is recursion ?

Dec. 2013, May 2016, Dec. 2016, May 2017

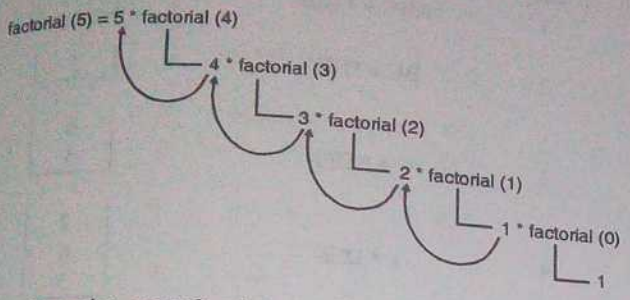
Ans. : Recursion

Recursion is a fundamental concept in mathematics. When a function is defined in terms of itself then it is called a recursive function. Consider the definition of factorial of a positive integer n.

$$\text{Factorial (n)} = \begin{cases} 1 & \text{if (n=0)} \\ n * \text{factorial } (n-1), & \text{otherwise} \end{cases}$$

Function "factorial ( )" is defined in terms of itself for $n > 0$. Value of the function at $n = 0$ is 1 and it is called the base. Recursion terminates on reaching the base.

For example :



Recursion expands when $n > 0$ and it starts winding up on hitting the base ($n = 0$).

Q. 20 Write a C program for GCD using recursion.

Ans. :

GCD of two numbers m and n can be defined as given below.

$$\text{GCD (m, n)} = \begin{cases} \text{GCD (n, m)} & \text{if m < n} \\ n & \text{if m is divisible by n} \\ \text{GCD (n, m \% n)} & \text{otherwise} \end{cases}$$

C-function for GCD

```
int GCD (int m, int n)
  {
        if (m < n)
            return (GCD (n, m));
        if (m%n ==0)
            return (n);
        return (GCD (n, m%n));
  }
```

Q. 21 Explain backtracking.

Ans. : Backtracking

Many a time we have to find an optimal solution to a problem, yet there may be no applicable theory to help us find the optimum. We resort to exhaustive search. Such an exhaustive search technique is known as backtracking.

A practical example of backtracking algorithm is the problem of arranging furniture in a new house. There are many possibilities but only few will be accepted by the owner. Starting with an empty house, each piece of furniture is placed in some part of the room. If all the furniture is placed and the owner is satisfied then the process terminates. When we reach a point where the arrangement is not desirable then we undo the last step and try some other combination. This may force another undo.

Q. 22 Define tail recursion.

Ans. : Tail recursion : Tail recursion refers to a recursive call at the last line. Tail recursion can be eliminated by changing the recursive call to a goto preceded by a set of assignments per function call. This simulates the recursive call because nothing needs to be saved after the recursive call finishes. We can just goto the top of the function with the values that would have been used in a recursive call.

Q. 23 Give difference between recursion and iteration.

Ans. : Difference between recursion and iteration

Sr. No.	Recursion	Iteration
1.	In recursion, a function calls itself.	Iteration is implemented using loop-construct.
2.	An iteration can be implemented using recursion.	It may become quite difficult to implement every type of recursion using a loop. A linear recursion can be implemented using a loop.
3.	A recursion is implemented using a stack.	Iteration does not require a stack.
4.	Every programming language does not support recursion.	Iteration is supported by every programming language.
5.	Most of the natural problems can be expressed using a recursive definition.	It may not be easy to express every type of problem using loops.
6.	Every recursive problem can be converted into an equivalent non-recursive problem with the help of a stack.	Iteration does not require a stack.
7.	Recursive algorithm requires additional memory in the form of a stack.	Additional memory is not required.
8.	Error recovery is difficult to implement in recursive algorithms.	Error recovery can be implemented without much effort in iterative algorithms.

Chapter 2 : Stack

Q. 1 What is stack?

May 2014

Ans. : Stack is a LIFO (last in first out) structure. It is an ordered list of the same type of elements. A stack is a linear list where all insertions and deletions are permitted only at one end of the list. When elements are added to stack it grows at one end. Similarly, when elements are deleted from a stack, it shrinks at the same end.

Fig. 2.1 shows expansion and shrinking of a stack. Initially stack is empty.

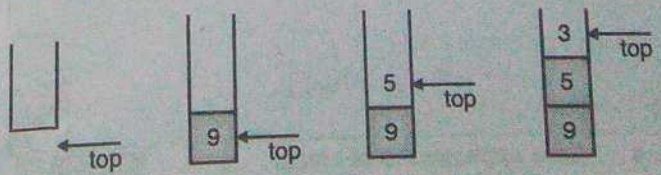


Fig. 2.1: Insertion of 9,5,3 in a stack

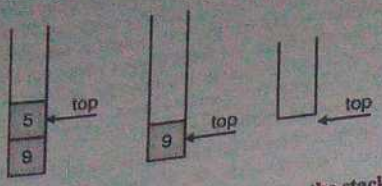


Fig. 2.2 : Deletion of 3 elements from the stack

A variable top, points to the top element of the list.

Q. 2 What are basic operations on stack ?

Ans. :

1. Initialize( ) - Make a stack empty
2. Empty( ) - To determine if a stack is empty or not
3. Full( ) - To determine if a stack is full or not
4. Push( ) - If a stack is not full then push a new element at the top of the stack (similar to insert in a list)
5. POP( ) - If a stack is not empty, then pop the element from its top(similar to delete( ) from a list)
6. display_top() - Returns the top element.

Q. 3 Write a program to implement STACK ADT using array. May 2015

Ans. :

'C' functions for operations on stack.

```
void push()
{
   if(p->top==N –1)
         printf("\noverflow !! can not be inserted");
   else
   {
         p->top=p->top+1;
         p->data[p->top]=x;
   }
}
int pop(stack *p)
{
   int x;
   if(p->top== –1)
         printf("\nUnderflow !!! can not be deleted");
   else
   {
         x=p->data[p->top];
         p->top=p->top –1;
   }
   return(x);
}
void print(stack *p)
{
   int i;
   for(i=p->top;i>=0;i—)
         printf("\n%d",p->data[i]);
}
```

Q. 4 Give applications of stack. May 2014

Ans. : Stack data structure is very useful. Few of its applications are given below :

1. Expression conversion
   (a) Infix to postfix (b) Infix to prefix
   (c) Postfix to infix (d) Prefix to infix
2. Expression evaluation 3. Parsing
4. Simulation of recursion 5. Function call

Q. 5 Evaluate the following postfix expression and show stack after every step in tabular form. Given A = 5, B = 6, C = 2, D = 12, E = 4

Postfix expression : ABC + *DE\–

Ans. :

Step	Input	Stack
1.	ABC + * DE/–	
2.	BC + *DE/–	5
3.	C + * DE/–	6 5
4.	+ * DE/–	2 6 5
5.	* DE/-	8 5
6.	DE/–	40
7.	E/–	12 40
8.	/–	4 12 40
9.	–	3 40
10.	End	37

∴ Value of the expression = 37

Q. 6 Compare stacks and queues.

Ans. : Comparison of stack and queue

Sr. No.	Stack	Queue
1.	Stack is last in first out (LIFO) i.e. which is entered last will be retrieved firstly.	Queue is first out (FIFO) i.e. which is entered first will be served first.
2.	Stack is Linear data structure which follows LIFO.	Queue is Linear data structure which follows FIFO.
3.	Insertion and deletions are possible through one end called top.	Insertions are at the rare end and deletions are from the front end in a queue.
4.	Example : Books in library.	Example : Cinema ticket counter.

Q. 7 Evaluate the following postfix expression. Show all steps : ab * c + d – e + where a = 5, b = 4, c = 10, d = 15 and e = 6.

Ans. :

Step	Expression	Stack
1.	ab*c + d – e +	
2.	b*c + d – e +	5
3.	* c + d – e+	4 5
4.	c + d – e+	20
5.	+ d – e+	10 20
6.	d – e +	30
7.	– e+	15 30
8.	e+	15
9.	+	6 15
10.	End	21

∴ Value of the expression = 21

Q. 8 Evaluate the following postfix expression using stack.

623 + –382 /+*2$ 3 +

Ans. :

623 + –382 /+*2$ 3 + ← Initially stack is empty.

First token is an operand, push 6 on the stack

23 + –382 /+ *2$3 + | 6

Next token in an operand, push 2 on the stack

3 + –382 /+ *2$3 + | 2
6

Next token is an operation, push 3 on the stack.

+ –382 /+ *2$3 + | 3
2
6

Next token is an operator, POP two operands 3 and 2, add them and push the result on the stack.

– 382 /+ *2$3 + | 5
6

Next token is an operator, POP two operands 6 and 5, subtract them and push the result on the stack.

382 /+ *2$3 + | 1

Next token is an operand, push 3 on the stack

82 /+ *2$3 + | 3
1

Next token is an operand, push 8 on the stack

2 /+ *2$3 + | 8
3
1

Next token is an operand, push 2 on the stack

/+ *2$3 + | 2
8
3
1

Next token is an operator, POP two operands 8 and 2, divide them and push the result in the stack.

+ *2 $3 + | 4
3
1

Next token is an operator, POP two operands 3 and 4, add them and push the result in the stack.

*2$3 + | 7
1

Next token is an operator, POP two operands 1 and 7, multiply them and push the result in the stack.

2$3 + | 7

Next token is an operand, push 2 on the stack.

$3 + | 2
7

Next token is an operand, POP two operand 7 and 2, find the power and push the result on the stack.

3 + | 49

Next token is an operand, push 3 on the stack

\+ | 3
49

Next token is an operator, POP two operands 49 and 3, add them and push the result on the stack.

| 52

∴ Value of the expression = 52

Q. 9 Write conversion of an expression from infix to postfix. Explain with example.

Ans. :

Example of an infix expression : (A + B ∧ C) * D + E ∧ 5 where A, B, C, D and E are integer constants and B ∧ C means B^C

Manual method

Step 1 :

In manual evaluation of the above expression, B ^ C must be calculated first. Hence, we convert B ∧ C to its equivalent postfix equivalent i.e. BC∧

(A + B ∧ C) * D + E ∧ 5 – original expression

(A + BC^) * D + E ^ 5 — BC^ should treated as a single integer number (single token)

and A + BC^ should be the next operation to be performed

ABC^ + * D + E ^ 5 — ABC^ + * D is the next operation to be performed

ABC^ + D* + E ^ 5 — E ^ 5 is the next operation to be performed

ABC^ + D* + E5^ — + is the last operation to be performed

ABC^ + D* E5^ + → Final expression.

Algorithmic approach

A + B * C

In the above example, evaluation of * precedes evaluation of + as * has higher precedence over +.

Perform an operation if the current operator has equal or higher precedence over the succeeding operator.

Thus an operator coming on the right of the current operator will determine if the current operation should be performed.

P + Q + R

Same precedence and hence P + Q can be performed

P * Q + R

Higher precedence and hence P * Q can be performed

P * Q ^ R + S

lower precedence — higher precedence.

Sequence in which operations will be performed ^, *, +

All operators must be saved on top of the stack until we get an equal or higher precedence operator

Conversion of P * Q^R + S

Expression	Stack	Output	Remark
P*Q ^ R + S	NULL	–	–
*Q^R + S	NULL	P	Operand must be printed
Q^R + S	*	P	* operation will be performed if the next operator is of lower or equal precedence.
^R + S	*	PQ	Operand must be printed
R + S	*^	PQ	* cannot be performed as ^ has higher precedence
+ S	*^	PQR	Operand must be printed
+ S	NULL	PQR^*	All higher precedence operators (compare to +) are popped and printed and finally the current operator is pushed on top of the stack.
S	+	PQR^*	–
NULL	+	PQR^* S	Operand must be printed

Q. 10 Convert the following expression from infix to postfix using a stack.

a && b ¦¦ c ¦¦ !(e > f)

Ans. :

Sr. No.	Expression	Stack	Output	Comment
1.	a &&b¦¦ c ¦¦ ! (e>f)	NULL	–	Initial condition
2.	&&b ¦¦ c ¦¦ ! (e >f)	NULL	a	Print a
3.	b ¦¦ c ¦¦ ! (e > f)	&&	a	Push &&
4.	¦¦ c ¦¦ ! (e>f)	&&	ab	Print b
5.	¦¦ c ¦¦ ! (e>f)	NULL	ab &&	Pop and print higher precedence operators
6.	c ¦¦ ! (e>f)	¦¦	ab &&	Push the current operator
7.	¦¦ ! (e>f)	¦¦	ab&& c	Print c
8.	¦¦ ! (e>f)	NULL	ab&&c ¦¦	Pop and print equal or higher precedence operators
9.	!(e > f)	¦¦	ab&&c ¦¦	Push the current operator
10.	! (e>f)	¦¦	ab&&c ¦¦	Pop and print equal or higher precedence operators
11.	(e > f)	¦¦ !	ab&&c ¦¦	Push the current operator
12.	e > f	¦¦ ! (	ab&&c ¦¦	'(' should always be pushed.
13.	> f)	¦¦ ! (	ab&&c ¦¦ e	Print e
14.	> f)	¦¦ ! (	ab&&c ¦¦ e	Pop and print equal or higher precedence operators.

Sr. No.	Expression	Stack	Output	Comment
				Remember '(' has lowest precedence
15.	f)	‼ ! (>	ab&&c ‼ e	Push the current operator
16.	)	‼ ! (>	ab&&c ‼ e f	Print f
17.	NULL	‼ !	ab&&c ‼ e f >	When the next input is ')' then all operators until '(' should be popped and printed
18.	NULL	NULL	ab&&c ‼ e f >! ‼	Pop and print all operations.

Q. 11 Convert the following expression into postfix. Show all steps : a + b * c/d – e

Ans. :

Sr. No.	Expression	Stack	Output
1.	a + b * c/d – e	NULL	-
2.	+ b * c/d – e	NULL	a
3.	b * c/d – e	+	a
4.	* c/d – e	+	ab
5.	c/d – e	+ *	ab
6.	/d – e	+ *	abc
7.	d – e	+ /	abc *
8.	– e	+ /	abc * d
9.	e	–	abc * d/+
10.	end	–	abc * d/+ e
11.	end	NULL	abc * d/+ e –

Postfix : abc * d/+ e –

Q. 12 Convert the following infix expressions into postfix expression using stack :

(I) ((A + B)* C–(D – E)) $ (F + G), where $- Exponent

(II) A$B*C – D + E/F/(G + H).

Ans. :

(I) ((A + B)* C – (D – E)) $ (F + G)

Sr. No.	Stack	Infix expression	Postfix expression
1.	Empty	((A + B)* C – (D – E)) $ (F + G)	–
2.	(	(A + B)* C – (D – E)) $ (F + G)	–
3.	((	A + B)* C – (D – E)) $ (F + G)	A
4.	((+	B)* C – (D – E)) $ (F + G)	A
5.	((+	)* C – (D – E)) $ (F + G)	AB+
6.	(	* C – (D – E)) $ (F + G)	AB +
7.	(*	C – (D – E)) $ (F + G)	AB +
8.	(*	– (D – E)) $ (F + G)	AB + C
9.	(*	(D – E)) $ (F + G)	AB + C*
10.	(–	D – E)) $ (F + G)	AB + C *
11.	(–(	– E)) $ (F + G)	AB + C * D
12.	(–(	E)) $ (F + G)	AB + C * D
13.	(–(–	)) $ (F + G)	AB + C * DE
14.	(–(–	) $ (F + G)	AB + C * DE –
15.	(–	$ (F + G)	AB + C * DE – –
16.	Empty	(F + G)	AB + C * DE – –
17.	$	F + G)	AB + C * DE – –
18.	$(	+ G)	AB + C * DE – – F
19.	$(	G)	AB + C * DE – – F
20.	$(+	)	AB * C * DE – – FG
21.	$(+	End	AB + C * DE – – FG +
22.	$	End	AB + C * DE – – FG + $
23.	Empty	End	

(II) A $ B * C – D + E/F/(G + H)

Sr. No.	Stack	Infix expression	Postfix expression
1.	Empty	A $ B * C – D + E/F/(G + H)	–
2.	Empty	$ B * C – D + E/F/(G + H)	A
3.	$	B * C – D + E/F/(G + H)	A
4.	$	* C – D + E/F/(G + H)	AB
5.	*	C – D + E/F/(G + H)	AB $
6.	*	– D + E/F/(G + H)	AB $ C
7.	–	D + E/F/(G + H)	AB $ C *
8.	–	+ E/F/(G + H)	AB $ C * D
9.	+	E/F/(G + H)	AB $ C * D –
10.	+	/F/(G + H)	AB $ C * D – E
11.	+/	F/(G + H)	AB $ C * D – E
12.	+/	/(G + H)	AB $ C * D – EF
13.	+/	(G + H)	AB $ C * D – EF/
14.	+/(	G + H)	AB $ C * D – EF/
15.	+/(	+ H)	AB $ C * D – EF /G
16.	+/(+	H)	AB $ C * D – EF /G

Ans. :

```
/* program for conversion of infix into its postfix form
   operators supported '+,-,*,/,%,^,(,)
   operands supported -- all single character operands */

#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50
typedef struct stack
{
   int data[MAX];
   int top;
}stack;

int precedence(char);
void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *,int );
int top(stack *); //value of the top element
void infix_to_postfix(char infix[],char postfix[]);

void main()
{ char infix[30],postfix[30];
 clrscr();
 printf("\nEnter an infix expression : ");
 gets(infix);
 infix_to_postfix(infix,postfix);
 printf("\nPostfix : %s ",postfix);
 getch();
}
void infix_to_postfix(char infix[],char postfix[])
{  stack s;
   char x;
   int i,j;//i-index for infix[],j-index for postfix
   char token;
   init(&s);
   j=0;
   for(i=0;infix[i]!='\0';i++)
    {    token=infix[i];
         if(isalnum(token))
              postfix[j++]=token;
         else
              if(token == '(')
                   push(&s,'(');
              else
                   if(token == ')')
                     while((x=pop(&s))!='(')
                          postfix[j++]=x;
                   else
                        {

   while(precedence(token)<=precedence(top(&s)) &&
!empty(&s))
                             {
                                  x=pop(&s);
                                  postfix[j++]=x;
                             }
                        push(&s,token);

                        }

   }
   while(!empty(&s))
   {
         x=pop(&s);
         postfix[j++]=x;
   }
postfix[j]='\0';
}
int precedence(char x)
{
   if(x == '(')                    return(0);
   if(x == '+' || x == '-')        return(1);
   if(x == '*' || x == '/' || x == '%') return(2);
   return(3);
}

void init(stack *s)
{
   s->top=-1;
}

int empty(stack *s)
{
   if(s->top==-1)  return(1);
   return(0);
}
int full(stack *s)
{
   if(s->top==MAX-1)   return(1);
   return(0);
}

void push(stack *s,int x)
{
   s->top=s->top+1;
   s->data[s->top]=x;
}

int pop(stack *s)
{
   int x;
   x=s->data[s->top];
```



```
   s->top=s->top
   return(x);
}
int top(stack * p)
{
   return(p->data
}
```

Output

```
Enter infix expre
abc+*d/g+
```

Q. 14 Write c… prefix.

Ans. :

Example of … where A, B, C, D … B^c.

Manual metho…

In manual … calculated first. … ^BC. Hereafter …

Expressi…
(A + B^C)*D
(A + B^C)*D
(A + ^BC)*D
+A^BC*D
*+A^BCD
*+A^BCD

Final …

Algorithm

An i… the help o…

1. Rev…
2. Mal… eve…
3. Co… Re… Ab… from 1 …

(A…

Step 1

Step 2

Step …

E…
5^
^
E

```
    s->top=s->top-1;
    return(x);
}
int top(stack * p)
{
    return(p->data[p->top]);
}
```

Output

```
Enter infix expression:a*(b+c)/d+g
abc+*d/g+
```

Q. 14 Write conversion of an expression from infix to prefix. Explain with example.

Ans. :

Example of a infix expression : (A + B ∧ C) * D + E ∧ 5 where A, B, C, D and E are integer constant and B ∧ C stands for B^C.

Manual method

In manual evaluation of the above expression, B^C must be calculated first. Hence, we convert B^C to its prefix equivalent i.e. ^BC. Hereafter, ^BC should be treated as a single operand.

Expression	After conversion	Comment
(A + B^C)*D + E^5	–	Initial
(A + B^C)*D + E^5	(A + ^BC)*D+E^5	Convert B^5
(A + ^BC)*D+E^5	+A^BC*D+E^5	Convert A + ^BC
+A^BC*D + E^5	*+A^BCD + E^5	Convert +A^BC*D
*+A^BCD + E^5	*+A^BCD + ^E5	Convert E^5
*+A^BCD + ^E5	+*+A^BCD^E5	Convert the rest

Final expression in prefix = +*+A^BCD^E5

Algorithmic approach

An infix expression can be converted into prefix form with the help of following steps :

1. Reverse the infix expression
2. Make every '(' (opening bracket) as ')' (closing bracket) and every ')' as '('
3. Convert the modified expression to postfix form. 4. Reverse the postfix expression

Above algorithm can be explained by carrying out the steps from 1 to 4 on the example :

(A + B ∧ C) * D + E ∧ 5

Step 1 : Reverse the infix expression.

5 ∧ E + D *) C ∧ B + A(

Step 2 : Make every '(' as ')' and every ')' as '('

5 ∧ E + D * (C ∧ B + A)

Step 3 : Convert the expression to postfix form :

Expression (input)	Stack	Output	Comment
5∧E+D*(C∧B+A)	Empty	–	Initially
∧E+D*(C∧B+A)	Empty	5	Print
E+D*(C∧B+A)	∧	5	Push

Expression (input)	Stack	Output	Comment
+D*(C∧B+A)	∧	5E	Push
D*(C∧B+A)	+	5E∧	Pop and Push
*(C∧B+A)	+	5E∧D	Print
(C∧B+A)	+*	5E∧D	Push
C∧B+A)	+*(	5E∧D	Push
∧B+A)	+*(	5E∧DC	Print
B+A)	+*(∧	5E∧DC	Push
+A)	+*(∧	5E∧DCB	Print
A)	+*(+	5E∧DCB∧	Pop and Push
)	+*(+	5E∧DCB∧A	Print
End	+*	5E∧DCB∧A+	Pop until '('
End	Empty	5E∧DCB∧A+*+	Pop everything

Step 4 : Reverse the expression +*+A^BCD^E5

Q. 15 Compute the postfix equivalent of the following infix expression

3 * log (x + 1) – a/2

Ans. :

Compute the postfix equivalent of the following infix expression

3 * log (x + 1) – a/2

After fully parenthesizing the expression, we get

((3*(Log(x+1))) – (a/2))

Now, all operators are moved right to replace their corresponding right parentheses. All parentheses are removed.

3x1 + log* a2/–

Q. 16 Convert the following expression into other two forms.

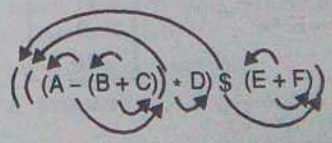
(i) ((A–(B + C))*D) $ (E+F) where $ = exponentiation

(ii) lmn $ q P y/–rs*+

Ans. :

(i) **Expression** : ((A– (B+C))*D) $ (E+F) is in infix form.

Step 1 : After fully parenthesizing the expression, we get



Step 2 : Expression can be converted to postfix form by moving operators to replace their corresponding right parentheses.

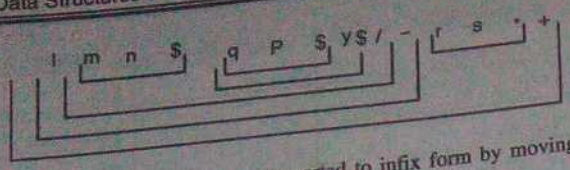
Postfix form : ABC +– D*EF+$

Expression can be converted to postfix form by moving operators to replace their corresponding left parentheses.

Prefix form : $*–A+BCD +EF

(ii) **Expression :** lmn $ q P y/–rs*+ is in postfix form.

Step 1 : After grouping elements in the order of evaluation, we get



Step 2 : Expression can be converted to infix form by moving operators at the center of the group.

infix form : *l* – m $ n/qPy + r * s

expression can be converted of prefix form by moving operators at the beginning of the group.

prefix form : + - *l* / $mn$$qPy *rs

Q. 17 Explain the necessity of representing expression in prefix and postfix notion. For the given postfix expression, evaluate it for the values given. Show stepwise stack contents.
ABC*DEF ^/G* – H * +
A = 6, B = 1, C = 4, D = 16, E = 2, F = 3, G = 2, H = 5
Where ^ = exponential operator.

Ans. : Prefix and Postfix expressions are free from any precedence. They are more suited for mechanization.

Input	Stack	Comment
ABC*DEF^/G* – H*+		Initially
BC*DEF^/G* – H*+	6	PUSH A
C*DEF^/G* – H*+	1 6	PUSH B
DEF^/G – H*+	4 1 6	PUSH C
DEF^/G* – H*+	4 6	4*1
EF^/G* – H *+	16 4 6	PUSH D
F^/G* – H *+	2 16	PUSH E
^/G* – H * +	3 2 16 4 6	PUSH F
/G* – H * +	8 16 4 6	2^3
G* – H * +	2 4 6	16/8
* – H * +	2 2 4 6	PUSH G
– H * +	4 4 6	2*2
H*+	0 6	4 – 4
*+	5 0 6	PUSH H
+	0 6	5*0
End	6	6 + 0

∴ Value of the expression = 6

Chapter 3 : Queue

Q. 1 What is queue ? Dec. 2014

Ans. : Queue

It is a special kind of list, where items are inserted at one end (the rear) and deleted from the other end (front). Queue is a FIFO (First In First Out) list.

We come across the term queue in our day to day life. We see a queue at a railway reservation counter, or a movie theatre ticket counter. Before getting the service, one has to wait in the queue. After receiving the service, one leaves the queue. Service is provided at one end (the front) and people join at the other end (rear).

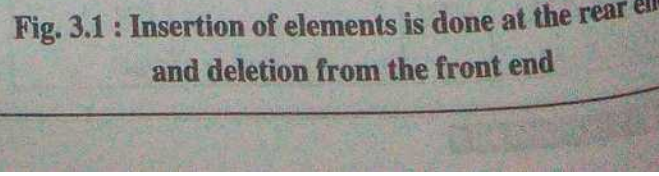


Fig. 3.1 : Insertion of elements is done at the rear end and deletion from the front end

Q. 2 List applications of queue. Dec. 2014

Ans. :

Various features of operating system are implemented using a queue.

(a) Scheduling of processes (Round Robin Algorithm).
(b) Spooling (to maintain a queue of jobs to be printed).
(c) A queue of client processes waiting to receive the service from the server process.

Various application software using non-linear data structure tree or graph requires a queue for breadth first traversal.

Simulation of a real life problem with the purpose of understanding its behaviour. The probable waiting time of a person at a railway reservation counter can be found through the technique of computer simulation if the following concerned factors are known :

(1) Arrival rate
(2) Service time
(3) Number of service counters

Q. 3 Specify ADT for queue. Dec. 2014, Dec. 2016

Ans.:

An array representation of queue requires three entities.

(a) An array to hold queue elements.
(b) A variable to hold the index of the front element.
(c) A variable to hold the index of the rear element.

A queue data type may be defined formally as follows :

```
# define MAX 30
typedef struct queue
{
  int data[MAX];
  int front, rear;
} queue;
```

During initialization of a queue, its front and rear are set to –1.

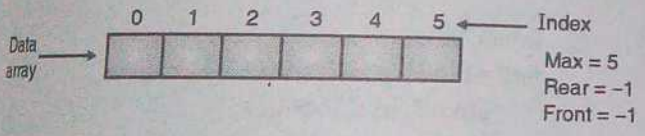


Fig. 3.2 : An empty queue after initialization

Fig. 3..3 shows the status of an queue after insertion of the element '5'.

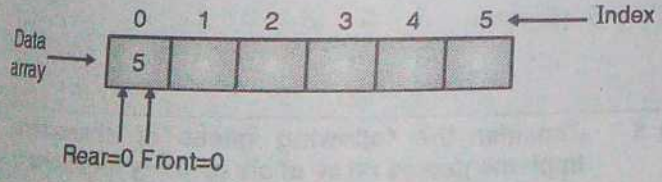


Fig. 3..3 : A queue after insertion of the first element '5'

On subsequent insertions, front remains at the same place, where rear advances.

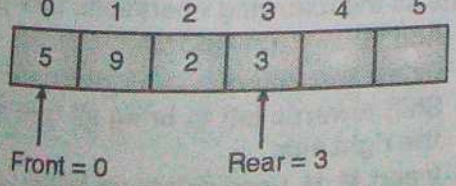


Fig. 3.4 : A queue after insertion of four elements

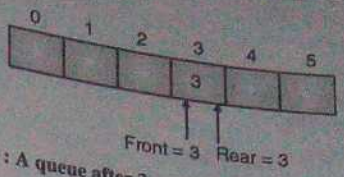


Fig. 3.5 : A queue after 3 successive deletions

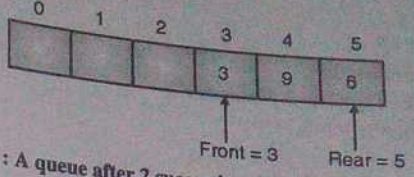


Fig. 3.6 : A queue after 2 successive insertions

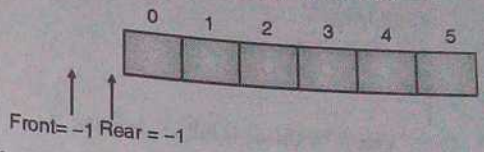


Fig. 3.7 : A queue after 3 successive deletions

Following points can be observed :

(1) If the queue is empty then front = –1 and rear = –1.
(2) If the queue is full then rear = MAX – 1.
(where MAX is the size of the array used for storing of queue elements).
(3) If rear = front then the queue contains just one element.
(4) If rear > front then queue is non-empty.

Problem with the above representation of the queue :

Problem of overflow can be handled by moving the queue elements to their left by number of vacant spaces. This operation could be very time consuming for a large queue.

Q. 4 Mention operations of Queue ADT. Dec. 2016

Ans. :

A set of useful operations on a queue includes :

(1) **initialize( ) :** Initializes a queue by setting the value of rear and front to –1.
(2) **enqueue( ) :** Inserts an element at the rear end of the queue.
(3) **dequeue( ) :** Deletes the front element and returns the same.
(4) **empty( ) :** It returns true(1) if the queue is empty and returns false(0) if the queue is not empty.
(5) **full( ) :** It return true(1) if the queue is full and returns false(0) if the queue is not full.
(6) **print( ) :** Printing of queue elements.

Q. 5 Write a program to implement queue using array. Dec. 2013, Dec. 2014

Ans.:

```
#include<conio.h>
#include<stdio.h>
#define MAX 10
typedef struct Q
{
   int R,F;
   int data[MAX];
} Q;
```

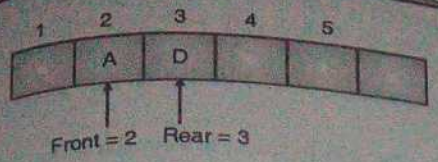
```
void initialise(Q *P);
int empty(Q *P);
int full(Q *P);
void enqueue(Q *P,int x);
int dequeue(Q *P);
void print(Q *P);
void main()
{
   Q q;
   int x,i;
   initialise(&q);
   printf("\nEnter 5 elements :");
   for(i=1;i<=5;i++)
   {
        scanf("%d",&x);
        if(!full(&q))
            enqueue(&q,x);
        else
        {
            printf("\n Queue is full .....exitting");
            exit(0);
        }
   }
   print(&q);
   for(i=1;i<=2;i++)
   {
        if(!empty(&q))
            x=dequeue(&q);
        else
        {
            printf("\ncan not delete...Queue is empty");
            exit(0);
        }
   }
   print(&q);
}
void initialise(Q *P)
{
   P->R=-1;
   P->F=-1;
}
int empty(Q *P)
{
   if(P->R==-1)
        return(1);
   return(0);
}
int full(Q *P)
{
   if(P->R==MAX-1)
        return(1);
   return(0);
}
void enqueue(Q *P,int x)
{
   if(P->R==-1)
   {
        P->R=P->F=0;
        P->data[P->R]=x;
   }
   else
   {
        P->R=P->R+1;
        P->data[P->R]=x;
   }
}
int dequeue(Q *P)
{
   int x;
   x=P->data[P->F];
   if(P->R==P->F)
   {
        P->R=-1;
        P->F=-1;
   }
   else
        P->F=P->F+1;
   return(x);
}
void print(Q *P)
{
   int i;
   if(!empty(P))
   {
        printf("\n");
        for(i=P->F;i<=P->R;i++)
            printf("%d\t",P->data[i]);
   }
}
```

Output

```
Enter 5 elements :  5  4  3  2  1
                    5  4  3  2  1
                    3  2  1
```

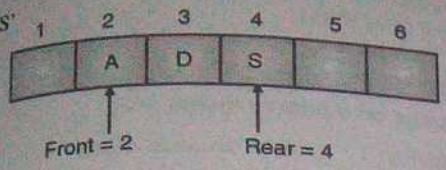
Q. 6 Consider the following queue of characters, implemented as array of six memory locations :-
Front = 2, Rear = 3
Queue : –, A, D, –, –, –
Where '–' denotes empty cell. Describe the queue as the following operations take place
(i) Add 'S' (ii) Add 'J'
(iii) Delete two letters
(iv) Shift towards left to bring all free spaces to the right side
(v) Insert M, H, I and delete on letter.

Ans. : Initial queue :



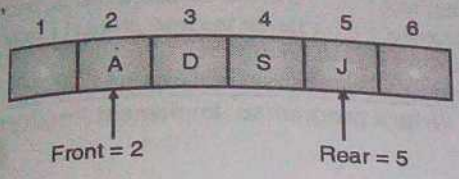
(a)

(i) Add 'S'



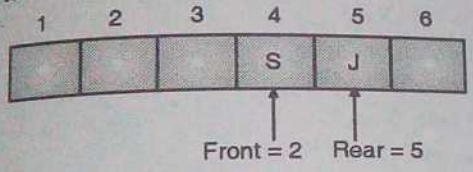
(b)

(ii) Add 'J'



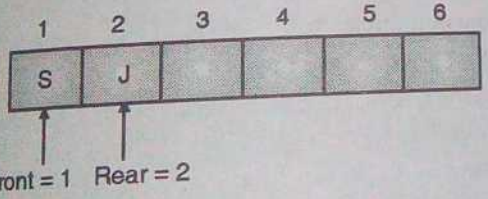
(c)

(iii) Delete two letters



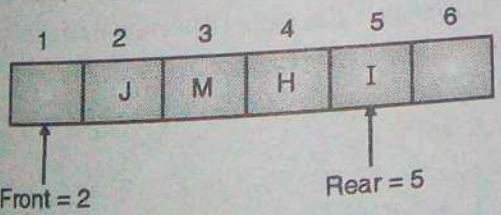
(d)

(iv) Shift towards left to bring all free spaces to the right.



(e)

(v) Insert M, H, I and delete one letter.



(f)

Fig. 3.8

Q.7 Explain queue using a circular array.

Ans. : Queue using circular queue

There is one potential problem with implementation of queue using a simple array. The queue may appear to be full although there may be some space in the queue.

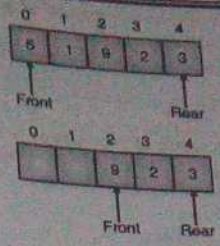


Fig. 3.9 : Queue is full, although locations 0 and 1 are vacant

After insertion of five elements in the array (a queue) as shown in Fig. 3.9,

rear = 4
front = 0
} queue is full

After two successive deletions

rear = 4
front = 2
} queue is full

queue in the Fig. 3.9 is full as there is no empty space ahead of rear. The simple solution is that whenever rear gets to the end of the array, it is wrapped around to the beginning. Now, the array can be thought of as a circle. The first position follows the last element.

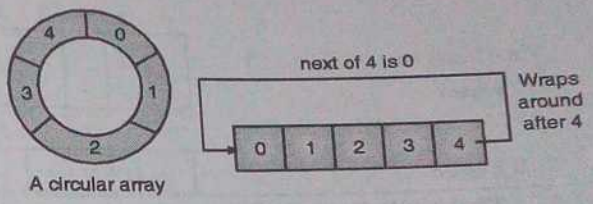
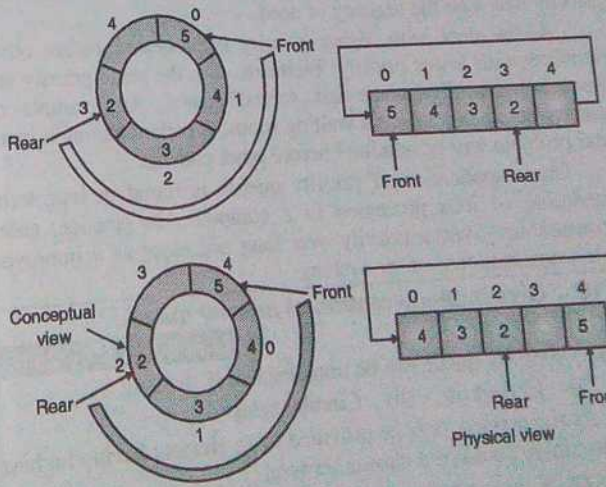


Fig. 3.10 : A circular array

In a circular array, the queue is found somewhere around the circle in consecutive positions.



Conceptual view

Fig. 3.11 : Queue at various places in an array

Q. 8 Consider a circular queue of size 5 having initial status as :

Front	Rear	Circular queue				
		0	1	2	3	4
			P	Q		
1	2					

Show the value of front, rear and the contents of circular queue after every step in tabular form for the following operations :

(1) R is added
(2) Delete 2 letters
(3) S, T, U are added
(4) Three letters are deleted (5) V is added

Ans. :

Operation	Front	Rear	Circular queue (0 1 2 3 4)
Initial	1	2	_ P Q _ _
R is added	1	3	_ P Q R _
Delete 2 letters	3	3	_ _ _ R _
S,T,U are added	3	1	T U _ R S
Three letters are deleted	1	1	_ U _ _ _
V is added	1	2	_ U V _ _

Q. 9 What is priority queue ? May 2015, May 2016, May 2017

Ans. : Priority queue is an ordered list of homogeneous elements. In a normal queue, service is provided on the basis of First-in-first-out. In a priority queue service is not provided on the basis of "first-come-first-served" basis but rather than each element has a priority based on the urgency of need.

An element with higher priority is processed before other elements with lower priority. Elements with the same priority are processed on "first-come-first served" basis. An example of priority queue is a hospital waiting room. A patient having a more fatal problem will be admitted before other patients.

Other application of priority queues is found in long term scheduling of jobs processed in a computer. In practice, short processes are given a priority over long processes as it improves the average response of the system.

Q. 10 Give implementation of priority queue. May 2015, May 2016

Ans. : Priority queue can be implemented both using :

(a) Linked list (b) Circular array

As the service must be provided to an element having highest priority, there could be a choice between

(a) List is always maintained sorted on priority of elements with the highest priority element at the front. Here, deletion is trivial but insertion is complicated as the element must be inserted at the correct place depending on its priority.

(b) List is maintained in the "FIFO" form but the service is provided by selecting the element with highest priority. Deletion is difficult as the entire queue must be traversal to locate the element with highest priority. Here, insertion is trivial (at rear end).

Implementation of a Priority Queue using a Circular Array

Data type for priority queue in a circular array

```
# define MAX 30 /* A queue with maximum of 30 elements */
   typedef struct pqueue
   { int data [MAX];
     int front, rear;
   } pqueue;
```

Operations on a priority queue

i) initialize( ) : Make the queue empty.
ii) empty( ) : Determine if the queue is empty.
iii) full( ) : Determine if the queue is full.
iv) enqueue( ) : Insert an element as per its priority.
v) dequeue( ) : Delete the front element (front element will have the highest priority.
vi) print( ) : Print elements of the queue.

Q. 11 Write a program to implement Priority Queue. Dec. 2016

Ans. :

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct pqueue
{  int data [MAX];
   int rear, front;
} pqueuel
void initialize(pqueue *p);
int empty(pqueue *p);
int full(pqueue *p);
void enqueue(pqueue *p, int x);
int dequeue(pqueue *p);
void print(pqueue *p);
void main( )
{ int x, op, n;
  pqueue q;
  initialize (&q);
do
{
 printf("\n1)create\n2)insert\n3)Delete\n4)print\n5)Quit");
    printf("\n enter your choice:");
    scanf ("%d", &op);
    switch (op)
    {
    case 1 : printf("\n enter no. of elements :");
             scanf("%d", &n);
             initialize (&q);
             printf("enter the data:");
             for (i = 0; i < n; i++)
             {
                scanf ("%d" &x);
```

```
            if (full (&q))
            {       printf("\n queue is full ...");
                    exit(0);
            }
            enqueue (&q, x);
        }
        break;
case 2 : printf("\n enter the element to be inserted");
        scanf ("%d", &x);
        if (full(&q))
        {
            printf("\n queue is full ...");
            exit(0);
        }
        enqueue (&q, x);
        break;
        case 3 : if (empty (&q))
        {
                printf("\n queue is empty ...");
                exit(0);
        }
        x = dequeue (&q);
        printf("\n element = %d", x);
        break;
        case 4 : print(&q);
        break;
        default : break;
        }
    } while (op! = 5);
}
void initialize (pqueue *p)
{ p -> rear = – 1;
  p -> front = – 1;
}
/* A value of rear or front as –1, indicate that the queue is
empty. */
int empty (pqueue *p)
{ if (P->rear == – 1)
        return (1); /* queue is empty */
  return (0); /* queue is not empty */
}
int full (pqueue *p)
{/* if front is next rear in the circular array then the queue
is full */
  if (p -> rear + 1)% MAX == p -> front)
        return (1); /* queue is full */
  return (0);
}
void enqueue (pqueue *p, int x)
{ int i;
  if (full (p))
  printf("\n overflow ...");
  else
  {     /* inserting in an empty queue */
        if (empty (p))
            {       p -> rear = p -> front = 0
                    p -> data [0] = x;
            }
        else
        {
/* move all lower priority data right by one place */
i = p -> rear ;
while (x > p -> data [i])
{       p -> data [(i + 1)%MAX] = p -> data [i];
        /* position i on the previous element */
        i = (i – 1 + MAX) % MAX; /* anticlock wise
movement inside the queue */
        if ((i + 1)% MAX = = p -> front)
        break; /* if all elements have been moved */
}
 /* insert x */
        i = (i + 1)% MAX;
        p -> data [i] = x;
 /* re-adjust rear */
    p -> rear = (p -> rear + 1) % MAX;
 }
}
}
int dequeue (pqueue *p)
{  int x;
   if (empty (p))
        printf("\n underflow ...");
        else
        {    x = p -> data [p -> front];
   if (p -> rear = = p -> front) /* delete last element */
                initialize (p);
            else
            p -> front = (p -> front + 1) % MAX;
        }
   return (x);
   }
void print (pqueue *p)
{  int i, x;
    i = p -> front;
    while ( i! = p -> rear)
    {       x = p data[i];
            printf ("\n%d", x);
            i = (i + 1) % MAX;
    }
    /* print the last data */
            x = p -> data[i];
            printf ("\n%d", x);
    }
```

Q. 12 Define double ended queue. Specify ADT for it. Implement any 2 operation of it. List variants of Double ended queue.

May 2014, Dec. 2014, May 2015

Ans. : Double ended queue

The word dequeue is a short form of double ended queue. It is general representation of both stack and queue and can be used as stack and queue. In a dequeue, insertion as well deletion can be carried out either at the rear end or the front end. In practice, it becomes necessary to fix the type of operation to be performed on front and rear end.

Dequeue can be classified into two types :

(i) Input restricted Dequeue

The following operations are possible in an input restricted dequeue :

(i) Insertion of an element at the rear and
(ii) Deletion of an element from front end
(iii) Deletion of an element from rear end

(ii) Output restricted Dequeue

The following operations are possible in an output restricted dequeue.

(i) Deletion of an element from front end
(ii) Insert ion of an element at the rear end
(iii) Insertion of an element at the front end

There are various methods to implement a dequeue.

(a) Using a circular array
(b) Using a singly linked list.
(c) Using a singly circular linked list.
(d) Using a doubly linked list.
(e) Using a doubly circular linked list.

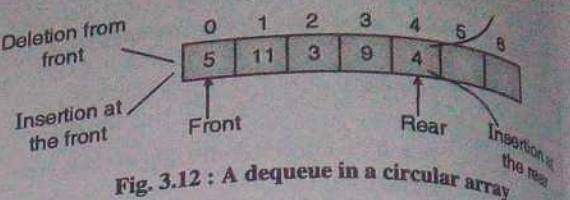


Fig. 3.12 : A dequeue in a circular array

Operations associated with dequeue

(a) empty( ) : Whether the queue is empty ?
(b) full( ) : Whether the queue is full ?
(c) initialize( ) : Make the queue empty
(d) enqueueR( ) : Add item at the rear end of the queue.
(e) enqueueF( ) : Add item at the front end of the queue.
(f) dequeueR( ) : Delete item from the rear end of the queue.
(g) dequeueF( ) : Delete item from the front end of the queue.

Chapter 4 : Linked List

Q. 1 State the advantages of linked list. May 2015

Ans. : Advantages of linked lists

i) Linked list is an example of dynamic data structure. They can grow and shrink during execution of the program.
ii) Representation of linear data structure. (Inline data like polynomial, stack and queue can easily be represented using linked list.
iii) Efficient memory utilization. Memory is not pre-allocated like static data structure. Memory is allocated as per the need. Memory is deallocated when it is no longer needed.
iv) Insertion and deletions are easier and efficient. Insertion and deletion of a given data can be carried out in constant time.

Q. 2 What is Linked List ? Explain.

Dec. 2013, May 2014, Dec. 2014, May 2015, Dec. 2015, May 2016, Dec. 2016

Ans. : Linked list

The linked list consists of a series of structures. They are not required to be stored in adjacent memory locations. Each structure consists of a data field and address field. Address field contains the address of its successors. Fig. 4.1 shows the actual representation of the structure.

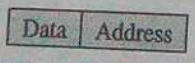


Fig. 4.1 : Representation of the structure

A variable of the above structure type is conventionally known as a **node**. Fig. 4.2 gives a representation of a linked list of three nodes.

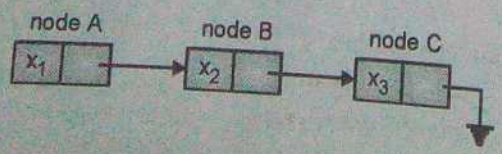


Fig. 4.2 : Linked list

A list consisting of three data x_1, x_2, x_3 is represented using a linked list. Node A stores the data x_1 and the address of the successor (next) node B.

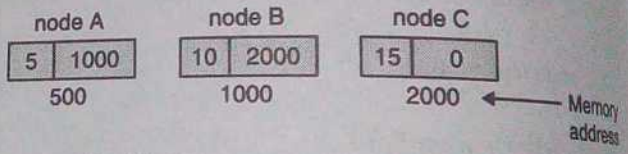


Fig. 4.3 : Memory representation of a linked list

Node B stores the data x_2 and the address of its successor node C. Node C contains the data x_3 and its address field is grounded (NULL pointer), indicating it does not have a successor.

Fig. 4.3 gives a memory representation of the linked list shown in Fig. 4.2. Nodes A, B and C happen to reside at memory locations 500, 1000 and 2000 respectively. $x_1 = 5$, $x_2 = 10$ and $x_3 = 15$.

Node A resides at the memory location 500, its data field contains a value 5 and its address field contains 1000, which is the address of its successor node. Address field of node C contains 0 as it has no successor.

Q. 3 State the different types of Link List.

Dec. 2013, May 2014, Dec. 2014, Dec. 2016

Ans. :

Different types of linked list are

(i) Singly Linked List

In this type of linked list two successive nodes of the linked list are linked with each other in sequential linear manner. Movement in forward direction is possible.

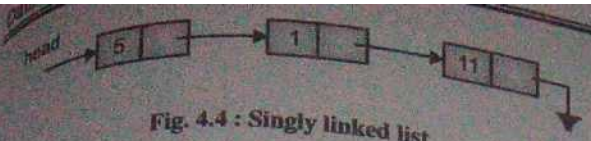


Fig. 4.4 : Singly linked list

(II) Doubly Linked List

In this type of linked list each node holds two-pointer fields. In doubly linked list addresses of next as well as preceding elements are linked with current node.

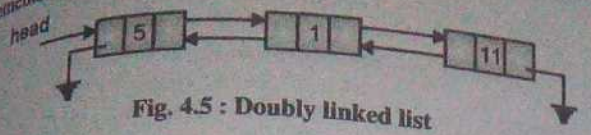


Fig. 4.5 : Doubly linked list

(III) A Circular Linked List

In a circular list the first and the last elements are adjacent. A linked list can be made circular by storing the address of the first node in the next field of the last node.

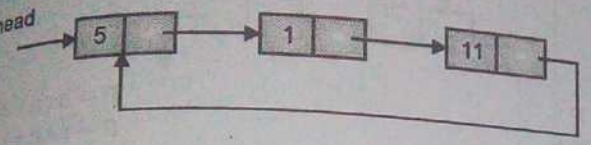


Fig. 4.6 : Circular linked list

Q. 4 Give difference between singly linked list and doubly linked list.

Ans. :

Difference between Singly Linked List and Doubly Linked List

Sr. No.	Singly linked list	Doubly linked list
1.	It has one pointer, pointing to successor.	It has two pointers, one pointing to successor and another pointing to predecessor.
2.	Traversal is possible only in the forward direction.	One can traverse in both forward and backward directions.
3.	Less memory is required by a node.	More memory required by a node.
4.	Fewer pointer adjustment in delete and insert operation.	More pointer adjustment in delete and insert operation.
5.	In singly linked list, each node contains data and link.	In doubly linked list, each node contains data, link to next node and link to previous node.
6	**Applications :** 1. Representation of linear data structure. 2. Representation of Stack. 3. Representation of Queue. 4. Representation of Polynomial. 5. Representation of Sparse matrix.	**Applications :** 1. Representation of dequeue. 2. Memory management and garbage collection.

Q. 5 Write a program to create singly Linked List and display the List. Dec. 2013, Dec. 2015

Ans. :

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
   int data;
   struct node *next;
}node;
node * create(int);
void print(node *);
int count(node *);
void main()
{
   node *HEAD;
   int n,number;
   printf("\n no. of items :");
   scanf("%d",&n);
   HEAD=create(n);
//create function returns the address of first node
   print(HEAD);
   number=count(HEAD);
   printf("\n No of nodes = %d",number);
}
node * create(int n)
{
   node *head,*P;
   int i;
   head=(node*)malloc(sizeof(node));
   head->next=NULL;
   scanf("%d",&(head->data));
   P=head;
   //create subsequent nodes
   for(i=1;i<n;i++)
   {
         P->next=(node*)malloc(sizeof(node));
         //new node is inserted as the next node after P
         P=P->next;
         scanf("%d",&(P->data));
         P->next=NULL;
   }
   return(head);
}
void print(node *P)
{
   while(P!=NULL)
   {
         printf("<- %d ->",P->data);
         P=P->next;
   }
}
```

```
int count(node *P)
{
   int i=0;
   while(P!=NULL)
   {
         P=P->next;
         i++;
   }
   return(i);
}
```

Output

```
no. of items :4
12
13
14
15
<- 12 -> <- 13 -> <- 14 -> <- 15 ->
No of nodes = 4
```

Q. 6 Write 'C' function for insertion of 'n' elements. May 2014, Dec. 2014, May 2015

Ans. :

Inserting a new item, say x, has three situations

1. Insertion at the front of the list.
2. Insertion in the middle of the list.
3. Insertion at the end of the list.

Algorithm for placing the new item at the beginning of a linked list

1. Obtain space for new node.
2. Assign data to the data field of the new node.
3. Set the next field of the new node to the beginning of the linked list.
4. Change the reference pointer of the linked list to point to the new node.

Algorithm for inserting the new data after a node N1

1. Obtain space for new node.
2. Assign value to its data field.
3. Search for the node N1.
4. Set the next field of the new node to point to N1 → next.
5. Set the next field of N1 to point to the new node.

'C' function to insert a data in a linked list after a given data.

```
node *insert(node *head, int x, int key)
{
 /* data x is to be inserted after the key */
/* if key is – 1 then x is to be inserted as a front node */
      node *P, *q;
      /* obtain space for the new node */
      P = (node *) malloc(sizeof(node));
      /* store x in the new node */
      P → data = x;
      if(key = = – 1)
      {
          /* insert the node at the front of the list */
          P → next = head;
          head = P;
      }
      else
      {
          /* search for the key in the linked list */
          q = head;
          while(key! = q → data && q! = NULL)
                q = q → next;
          if(q ! = NULL)
          {
                /* if the key is found */
                P → next = q → next;
                q → next = P;
          }
      }
      return(head);
}
```

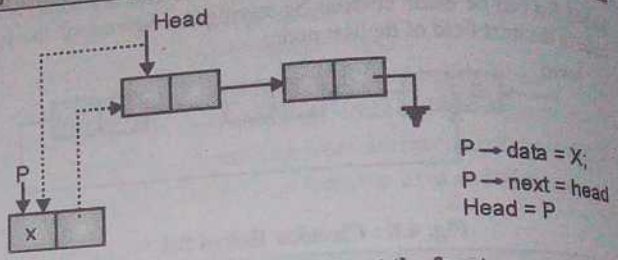


Fig. 4.7 : Insertion at the front

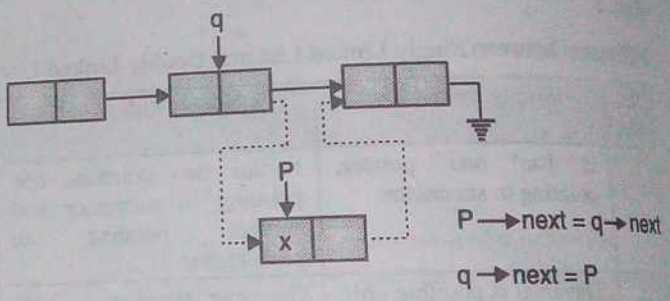


Fig. 4.8 : Insertion in between

Q. 7 Write recursive functions for :
(i) Display SLL forward
(ii) Display SLL reverse

Ans. :

(i) Display SLL forward :

```
void display_forward (node*h)
{
   if (h! = NULL)
         {
               printf("%d", h → data);
               display_forward(h →next);
         }
}
```

(ii) Display SLL reverse :

```
void display_reverse (node*h)
```

```
{
 if(h! = NULL)
  {
    display_reverse(h→next);
    printf("%d", h →data);
  }
}
```

Q. 8 Write a 'C' program to implement a singly Linked List which supports the following operations :
(i) Insert a node in the beginning
(ii) Insert a node in the end
(iii) Insert a node after a specific node
(iv) Deleting a specific node
(v) Displaying the list. May 2017

Ans. :

```
/*Operations on SLL(singly linked list) */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
typedef struct node
 { int data;
  struct node *next;
 }node;
node *create();
node *insert_b(node *head,int x);
node *insert_e(node *head,int x);
node *insert_in(node *head,int x);
node *delete_b(node *head);
node *delete_e(node *head);
node *delete_in(node *head);
node *reverse(node *head);
void search(node *head);
void print(node *head);
node *copy(node *);
int count(node *);
node * concatenate(node *, node *);
void split(node *);
void main()
{ int op,op1,x;
 node *head=NULL;
 node *head1=NULL,*head2=NULL,*head3=NULL;
 clrscr();
 do
  {
  printf("\n\n 1)create\n 2)Insert\n 3)Delete\n 4)Search");
printf("\n 5)Reverse\n 6)Print\n 7)Count\n 8)Copy
\n 9)Concatenate");
   printf("\n 10)Split\n 11)Quit");
   printf("\n Enter your Choice:");
   scanf("%d",&op);
   switch(op)
{ case 1:head=create();break;
```



```
case 2:printf("\n\t 1)Beginning\n\t 2)End\n\t 3)In
between");
        printf("\n Enter your choice : ");
        scanf("%d",&op1);
        printf("\n Enter the data to be inserted : ");
        scanf("%d",&x);
        switch(op1)
        { case 1: head=insert_b(head,x);
            break;
          case 2: head=insert_e(head,x);
            break;
          case 3: head=insert_in(head, x);
            break;
        }
        break;
   case 3:printf("\n\t 1)Beginning\n\t 2)End\n\t 3)In
between");
         printf("\n Enter your choice : ");
         scanf("%d",&op1);
         switch(op1)
         { case 1:head=delete_b(head);
             break;
           case 2:head=delete_e(head);
             break;
           case 3:head=delete_in(head);
             break;
         }
         break;
   case 4:search(head);break;
   case 5:head=reverse(head);
         print(head);
         break;
   case 6: print(head); break;
case 7: printf("\nNo.of node = %d",count(head));
break;   //count
   case 8: head1=copy(head);//copy
          printf("\nOriginal Linked List :");
          print(head);
          printf("\nCopied Linked List :");
          print(head1);
          break;
   case 9:printf("\nEnter the first linked list:");
          head1=create();
          printf("\nEnter the second linked list:");
          head2=create();
          head3=concatenate(head1,head2);
          printf("\nConcatenated Linked List :");
          print(head3);
          break;
          //concatenate
   case 10:printf("\nEnter a linked list : ");
          head1=create();
```

```
            split(head1);
            break;
            //split
        }
    }while(op!=11);
}
node *create()
{ node *head,*p;
 int i,n;
 head=NULL;
 printf("\n Enter no of data:");
 scanf("%d",&n);
 printf("\n Enter the data:");
 for(i=0;i<n;i++)
 {
  if(head==NULL)
  p=head=(node*)malloc(sizeof(node));
  else
  {
  p->next=(node*)malloc(sizeof(node));
  p=p->next;
  }
  p->next=NULL;
  scanf("%d",&(p->data));
 }
 return(head);
}
node *insert_b(node *head,int x)
{  node *p;
  p=(node*)malloc(sizeof(node));
  p->data=x;
  p->next=head;
  head=p;
  return(head);
}
node *insert_e(node *head,int x)
{  node *p,*q;
  p=(node*)malloc(sizeof(node));
  p->data=x;
  p->next=NULL;
  if(head==NULL)
    return(p);
  //locate the last node
  for(q=head;q->next!=NULL;q=q->next);
  q->next=p;
  return(head);
}
node *insert_in(node *head,int x)
{  node *p,*q;
  int y;
  p=(node*)malloc(sizeof(node));
  p->data=x;
  p->next=NULL;
  printf("\n Insert after which number ? : ");
  scanf("%d",&y);
  //locate the the data 'y'
  for(q=head ; q != NULL && q->data != y ;
   q=q->next);
  if(q!=NULL)
   {
   p->next=q->next;
   q->next=p;
   }
  else
    printf("\n Data not found ");
  return(head);
}
node *delete_b(node *head)
{
 node *p,*q;
 if(head==NULL)
  {
   printf("\n Underflow....Empty Linked List");
   return(head);
  }
 p=head;
 head=head->next;
 free(p);
 return(head);
}
node *delete_e(node *head)
{
 node *p,*q;
 if(head==NULL)
   {
   printf("\n Underflow....Empty Linked List");
   return(head);
   }
 p=head;
 if(head->next==NULL)
   { // Delete the only element
    head=NULL;
    free(p);
    return(head);
   }
 //Locate the last but one node
  for(q=head;q->next->next !=NULL;q=q->next)
  p=q->next;
  q->next=NULL;
  free(p);
  return(head);
}
node *delete_in(node *head)
{
```

```
node *p,*q;
int x,i;
if(head==NULL)
 {
 printf("\n Underflow....Empty Linked List");
 return(head);
 }
printf("\n Enter the data to be deleted : ");
scanf("%d",&x);
if(head->data==x)
 { // Delete the first element
  p=head;
  head=head->next;
  free(p);
  return(head);
 }
//Locate the node previous to one to be deleted
 for(q=head;q->next->data!=x && q->next
!=NULL;q=q->next )
  if(q->next==NULL)
  {
   printf("\n Underflow.....data not found");
   return(head);
  }
 p=q->next;
 q->next=q->next->next;
 free(p);
 return(head);
}
void search(node *head)
{ node *p;
 int data,loc=1;
 printf("\n Enter the data to be searched: ");
 scanf("%d",&data);
 p=head;
 while(p!=NULL && p->data != data)
 { loc++;
  p=p->next;
 }
 if(p==NULL)
 printf("\n Not found:");
 else
 printf("\n Found at location=%d",loc);
}
void print(node *head)
{ node *p;
 printf("\n\n");
 for(p=head;p!=NULL;p=p->next)
 printf("%d ",p->data);
}
node *reverse(node *head)
 { node *p,*q,*r;
 p=NULL;
 q=head;
 r=q->next;
 while(q!=NULL)
 {
 q->next=p;
 p=q;
 q=r;
 if(q!=NULL)
  r=q->next;
 }
 return(p);
}
node *copy(node *h)
 {
  node *head=NULL,*p;
  if(h==NULL)
   return head;
  //Copy the first node
  head=p=(node*)malloc(sizeof(node));
  p->data=h->data;
  while(h->next != NULL)
  {
   p->next=(node*)malloc(sizeof(node));
   p=p->next;
   h=h->next;
   p->data=h->data;
  }
  p->next=NULL;
  return (head);
 }
 int count(node *h)
 {
   int i;
   for(i=0; h!=NULL; h=h->next)
    i++;
   return(i);
 }
 node *concatenate( node *h1, node * h2)
   {
     node *p;
     if(h1==NULL)
      return(h2);
     if(h2==NULL)
      return(h1);
     p=h1;
     while(p->next != NULL)
 //go to the end of the 1st linked list
      p=p->next;
      p->next=h2;
      return(h1);
   }
 void split(node *h1)
```

```
{
  node *p,*q,*h2;
  printf("\n Linked list to be split : ");
  print(h1);
  /* linked list will be broken from the centre using the
pointers p and q */
  if(h1==NULL)
   return;
  p=h1;
  q=h1->next;
  while(q!=NULL && q->next != NULL)
   {
   q=q->next->next;
   p=p->next;
/* When q reaches the last node,p will reach the centre
node */
   }
  h2=p->next;
  p->next=NULL;
  printf("\nFirst half : ");
  print(h1);
  printf("\nSecond half : ");
  print(h2);
}
```

Q. 9 Write an algorithm for insertion and traversal in a circular linked list.

May 2014, Dec. 2016, May 2017

Ans. :

In a circular linked list, last node is connected back to the first node. In some applications, it is convenient to use circular linked list. A queue data structure can be implemented using a circular linked list, with a single pointer "rear" as the front node can be accessed through the rear node.

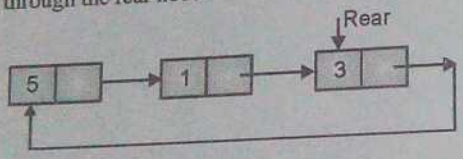


Fig. 4.9

Insertion of a node at the start or end of a circular linked list identified by a pointer to the last node of the linked list takes a constant amount of time. It is irrespective of the length of the linked list.

Algorithm for traversing a circular list is slightly different than the same algorithm for a singly connected linked list because "NULL" is not encountered.

'C' function for inserting a number at the rear of a circular linked list.

```
node * insert_rear(node * rear, int x)
{
        node *P;
        P =(node*) malloc(sizeof(node));
            /* acquire memory for the current data */
        P → data = x;
        if(rear = = NULL)
        {
        /* inserting in an empty linked list */
        rear = P;
        /* node is connected back to the same node */
        P → next = P;
            return(rear);
        }
        else
        {
            P → next = rear → next;
            rear → next = P;
            /* node P is made a part of the circle */
            rear = p
             return(rear)
        }
}
```

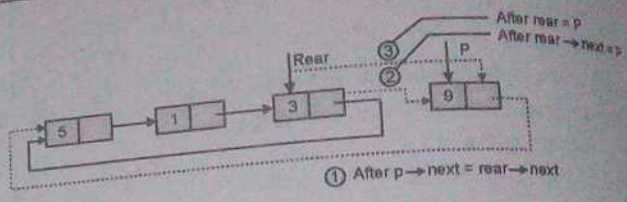


Fig. 4.10 : Insertion of a node at rear in circular linked list

'C' function for inserting a number at the front of the circular linked list.

```
node * insert_front(node * rear, int x)
{
   node *P;
   P =(node *) malloc(sizeof(node));
/* acquire memory */
   P → data = x;
        if(rear = = NULL)
        {
            rear = P;
            P → next = P;
            return(rear);
        }
        else
        {
          P → next = rear → next;
          rear → next = P;
return(rear);  /* rear is not moved after insertion */
        }
}
```

'C' function for traversing a circular linked list.

```
void print(node * rear)
   {
        node *P;
        if(rear ! = NULL)
        {
   P = rear → next;  /* start traversing from the front */
            do
            {
                printf("n%d", P → data);
```

```
            P = P → next;
        } while(P ! = rear → next);
    }
}
```

In a circular linked list, starting and the terminating case for traversal are same. In such a case, do-while is the most suitable construct for traversing.

Q. 10 Write a program in 'C' to implement circular queue using Link-list. May 2014

Ans. :

```
/*To implement circular queue using linked list.*/
#include<stdio.h>
#include<conio.h>
typedef struct node
{
int data;
struct node *next;
}node;
void init(node **R);
void enqueue(node **R,int x);
int dequeue(node **R);
int empty(node *rear);
void print(node *rear);
void main()
{
int x,option;
int n = 0,i;
node *rear;
init(&rear);
clrscr();
do
{
printf("\n 1. Insert\n 2. Delete\n 3. Print\n 4. Quit");
printf("\n your option:    ");
scanf("%d",&option);
switch(option)
{
 case 1 :
    printf("\n Number of Elements to be inserted");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
     scanf("\n %d",&x);
     enqueue(&rear,x);
    }
    break;
 case 2 : if(! empty(rear))
    {
      x = dequeue(&rear);
      printf("\n Element deleted = %d",x);
    }
    else
      printf("\n Underflow..... Cannot deleted");
    break;
 case 3 : print(rear);
    break;
}
}while(option != 4);
getch();
}
void init(node **R)
{
 *R = NULL;
}
void enqueue(node **R,int x)
{
 node *p;
 p = (node *)malloc(sizeof(node));
 p->data = x;
 if(empty(*R))
 {
   p->next = p;
   *R = p;
 }
 else
 {
   p->next = (*R)->next;
   (*R)->next = p;
   (*R) = p;
 }
}
int dequeue(node **R)
{
 int x;
 node *p;
 p = (*R)->next;
 p->data = x;
 if(p->next == p)
 {
   *R = NULL;
   free(p);
   return(x);
 }
 (*R)->next = p->next;
 free(p);
 return(x);
}
void print(node *rear)
{
  node *p;
  if(!empty(rear))
  {
    p = rear->next;
  }
  p = p->next;
```

```
    do
    {
     printf("\n %d",p->data);
     p = p->next;
    }while(p != rear->next);
}
int empty(node *P)
{
    if(P->next == -1)
          return(1);
    return(0);
}
```

Output

```
1. Insert
2. Delete
3. Print
4. Quit
your option :   1

Element to be inserted 4
              12    23    34  45

1. Insert
2. Delete
3. Print
4. Quit
your option :   3
              12    23    34  45

1. Insert
2. Delete
3. Print
4. Quit
your option :   2
Element deleted = 4
1. Insert
2. Delete
3. Print
4. Quit
your option :   3
              23    34    45

1. Insert
2. Delete
3. Print
4. Quit
your option :   4
```

Q. 11 What is doubly linked list ? Dec. 2013

Ans. : Doubly linked list

In a singly linked list, we can easily move in the direction of the link. Finding a node, preceding any node is a time consuming process. The only way to find the node which precedes a node is to start back at the beginning of the list. If we have a problem where moving in either direction is often necessary, then it is useful to have doubly linked lists. Each node has two link fields, one leading in the forward direction and one in the backward direction.

A node in a doubly linked list has at least 3 fields, say "data", "next" and "previous". A doubly linked list is shown in Fig. 4.12.

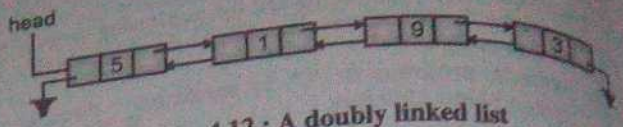


Fig. 4.12 : A doubly linked list

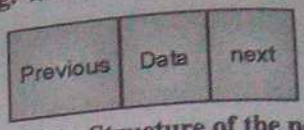


Fig. 4.13 : Structure of the node

A node of a doubly linked list can be defined using the following structure.

```
typedef struct dnode
   {
        int data;
        struct dnode *next, *prev;
   } dnode;
```

Q. 12 Write an algorithm to implement following operations on doubly linked list :

(i) Insertion (All cases)

(ii) Traversal (Forward and Backward) Dec. 2013, May 2017

```
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
typedef struct dnode
{
   int data;
   struct dnode *next,*prev;
}dnode;

dnode * create();
void print_forward(dnode *);
void print_reverse(dnode *);
void main()
{
   dnode *head;
   head=NULL;   // initially the list is empty
   head=create();
   printf("\nElements in forward direction :");
   print_forward(head);
   printf("\nElements in reverse direction :");
   print_reverse(head);
}
dnode *create()
{
   dnode *h,*P,*q;
   int i,n,x;
   h=NULL;
   printf("\nEnter no of elements :");
```

```
scanf("%d",&n);
for(i=0;i<n;i++)
{
    printf("\nEnter next data: ");
    scanf("%d",&x);
    q=(dnode*)malloc(sizeof(dnode));
    q->data=x;
    q->prev=q->next=NULL;
    if(h==NULL)
        P=h=q;
    else
        P->next=q;
        q->prev=P;
        P=q;
}
return(h);
}
void print_forward(dnode *h)
{
  while(h!=NULL)
  {
      printf("<- %d ->",h->data);
      h=h->next;
  }
}
void print_reverse(dnode *h)
{
  while(h->next!=NULL)
      h=h->next;
  while(h!=NULL)
  {
      printf("<- %d ->",h->data);
      h=h->prev;
  }
}
```

Output

```
Enter no of elements : 4
Enter next data: 1
Enter next data: 2
Enter next data: 3
Enter next data: 4
Elements in forward direction :
<-1-> <-2-> <-3-> <-4->
Elements in reverse direction :
<-4-> <-3-> <-2-> <-1->
```


Q. 13 Give applications of linked list.

Ans. : Polynomials as Linked Lists Dec. 2015, May 2016

Representation of polynomial

A polynomial P(x) of degree n is defined by the following expression.

$$P(x) = a_0 + a_1x^1 + a_2x^2 + \ldots\ldots\ldots + a_nx^n$$

Where a_n is any real number and n is an integer. A polynomial can be through as the ordered list of non-zero terms. Each term is a 2-tuple containing power and coefficient.

Thus, the polynomial $5 + 6x^2 - 9x^4$ is represented as ((0,5), (2,6), (4, – 9)). Tuple (0, 5) stands for $5x^0$; the second tuple (2, 6) indicates $6x^2$ and the last tuple denotes $-9x^4$. The linked representation of the said polynomial is shown below.

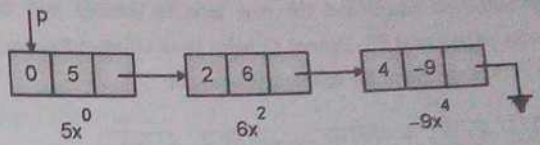


Fig. 4.14

Each term of the polynomial can be defined by the following structure.

```
typedef struct pnode
  {
      float coeff;
      int pow;
      struct pnode * next;
  } pnode;
```

We can treat a polynomial an abstract data type and perform following basic operations

(1) Creating a polynomial
(2) Printing a polynomial
(3) Addition of two polynomials
(4) Multiplication of two polynomials
(5) Evaluation of a polynomial.

Addition of Two Polynomials

In order to add two polynomials denoted by "P" and "q", both the linked lists are scanned term by term. Whenever the powers of the current terms of "P" and "q" are same, their coefficients are added to obtain the coefficient of the term of the new polynomial.

The exponent part of this term is the same as that of the current term of either of "P" or "q". Subsequently, both pointers pointing to the current nodes are advanced to point to the next node. If the power part of one polynomial, say P, is less than a new node is created which is a copy of the term of "P" and this new node is added to the resultant list.

When one of the lists of "P" or "q" is exhausted then the remaining nodes of the other list is simply copied to the resultant list.

First polynomial $5 + 9x + 10x^2 + 13x^5 + 15x^7$ (P1)

Second polynomial $6 + 13x^2 + 9x^3$ (P2)

Resultant polynomial $11 + 9x + 23x^2 + 9x^3 + 13x^5 + 15x^7$ (P1 + P2)

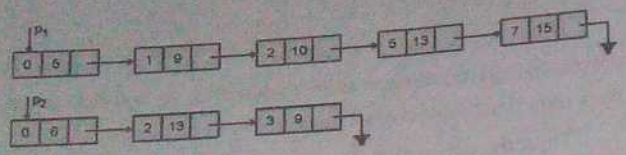


Fig. 4.15 : Representation of two polynomial using linked list

Since, the power of the terms pointed by p1 and p2 are same, coefficients are added and the new term is inserted into the resultant polynomial P3. Pointer r3 helps in inserting, subsequent terms at the rear end of the resultant polynomial.

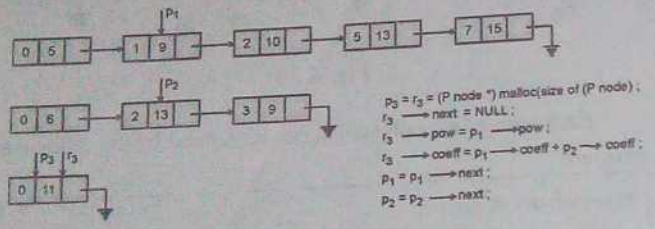


Fig. 4.16(a)

Since, the power of the term pointed by P1 is less than the power of the term pointed by P2, term pointed by P1 is added to P3. Term is added as a next node of r3. P1 and r3 are advanced by a node.

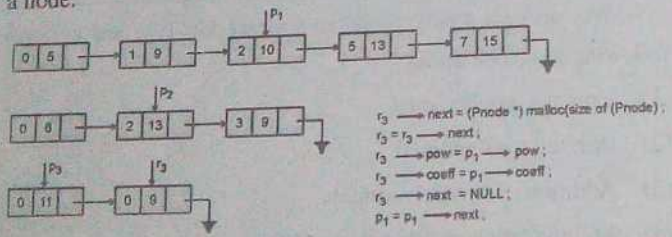


Fig. 4.16 (b)

Coefficients of the terms pointed by P1 and P2 are added and the new term is inserted into P3.

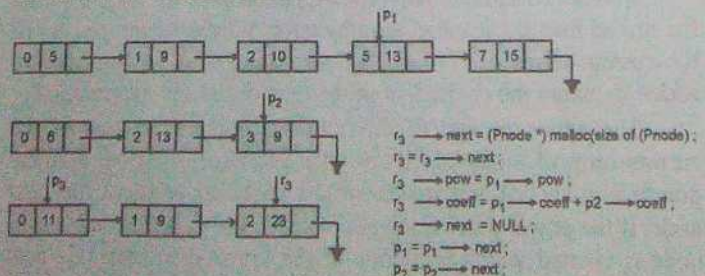


Fig. 4.16(c)

Since the power of the term pointed by P2 is less than the power of the term pointed by P1, term pointed by P2 is inserted into P3.

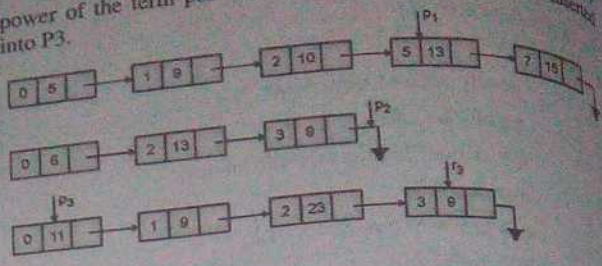


Fig. 4.16 (d)

Since no more terms are left in P2, all the terms of P1 are inserted at the end of P3.

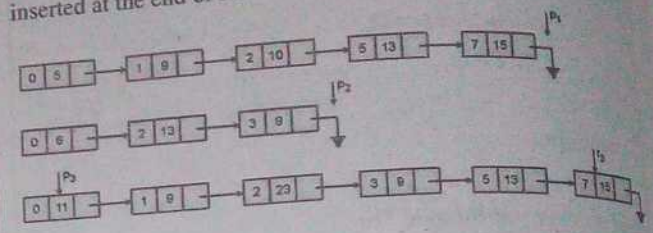


Fig. 4.16(e)

```
while(p1 != NULL)
  {   r3 -> next = (pnode*) malloc(sizeof(pnode));
      r3 = r3 -> next;
      r3 -> pow = p1 -> pow;
      r3 -> coeff = p1 -> coeff;
      r3 -> next = NULL
      p1 = p1 -> next;
  }
```

Q. 14 On a system using first-fit allocation, status of the memory is as follows after initial allocation.

U = used	U	A	U	A	U	A	U	A
A = Available	20K	10K	30K	30K	20K	10K	40K	20K

At what starting address will each of the additional request 20K and 10K be allocated.

Ans. :

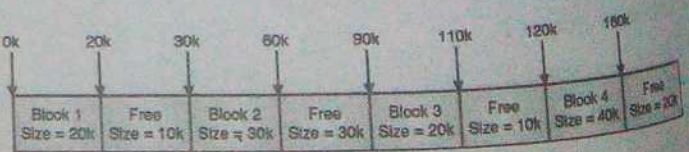


Fig. 4.17 : Initial allocation

A request of 20K will locate the first free block of size ≥ 20K starting at the address 60K. This block will be split into two part, second part will be allocated. Block will be allocated at the address 70K.

Fig. 4.18(a) : After allocation of 20K

An addition request for 10K will locate the first free block of size ≥ 10K starting at address 20K. The entire block will be allocated.

Fig. 4.18(b) : After allocation of 20K, 10K.

Q.15 What do you mean by priority queue? Explain any one application of priority queue with suitable Example.

Ans. :

Priority queue is an ordered list of homogeneous elements. Is a normal queue , service is provided on the basis of First-in-first-out. In a priority queue service is not provided on the basis of first-come-first-served basis but rather each element has a priority based on the urgency of need.

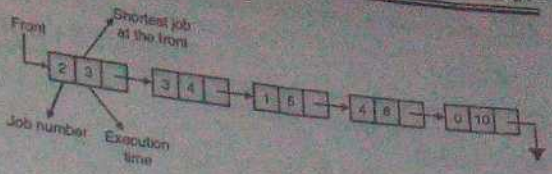
An element with higher priority is processed before other elements with lower priority.

Elements with the same priority are processed on first-come-first-served basis. An example of priority queue is hospital waiting room. A patient having a more fatal problem will be admitted before other patients.

Other application of priority queue is found in long term scheduling of jobs processed in a computer. In practice, short processes are given a priority over long processes as it improves the average response of the system. Let us consider a list of jobs with the execution- time requirements as given below :

Job No	0	1	2	3	4
Execution time	10	5	3	4	6

Job numbers 0 to 4, enter a priority queue as they arrive. Shortest job will have the highest priority and it will be serviced first.



Job no.2 will be completed in 3 time units.

Job no.3 will require additional 4 time units, it will be completed in 7 time units.

Job no.1 will require additional 5 units of time, it will be completed in 12 time units.

Job no. 4 will require additional 6 units of time, it will be completed in 18 time units.

Job no. 0 will require additional 10 units of time, it will be completed in 28 time units.

Structure of node

```
typedef struct node
  { int time ;
    int jobno ;
        struct node *next ;
  } node ;
```

Pseudo code for calculation of turnaround time

```
Priority queue q ; /* q is a priority queue */
initialize q;
Read n /* enter no. of jobs */
for ( i = 0 ; i < n ; i++)
   { read execution_time, jobno ;
        insert(&q, execution_time, jobno);
   }
   elapsed_time = 0 ;
while( ! empty(q))
   { elapsed_time = elapsed_time + time(front node);
        print jobno(front node), elapsed_time;
        delete(&q);
```

Chapter 5 : Sorting and Searching

Q. 1 Define sorting with its types.

Ans. : Sorting

Sorting is a process of ordering a list of elements in either ascending or descending order. Sorting can be divided into two categories.

Internal sorting takes place in the main memory of the computer. Internal sorting can take advantage of the random access nature of the main memory. Elements to be sorted are stored in an integer array.

External sorting is carried on secondary storage. External sorting becomes a necessity if the number of elements to be sorted is too large to fit in main memory. External sorting algorithms should always take into account that movement of data between secondary storage and main memory is best done by moving a block of contiguous elements.

Q. 2 Explain in brief bubble sort.

Ans. : Bubble sort

Bubble sort is one of the simplest and the most popular sorting method. The basic idea behind bubble sort is as a bubble rises up in water, the smallest element goes to the beginning. This method is based on successive selecting the smallest element through exchange of adjacent element.

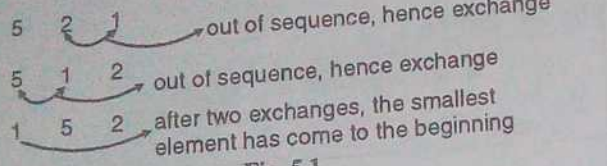


Fig. 5.1

First pass of the bubble sort

Let n be the number of elements in an array a[ ]. The first pass begins with the comparison of a[n – 1] and a[n – 2]. If a[n – 2] is larger than a[n – 1], the two elements are exchanged. The smaller elements, now at a[n – 2] is compared with a[n – 3] and if necessary the elements are exchanged to place the smaller one in a[n – 3]. Comparison progresses backward and after the last comparison of A[1] and A[0] and possible exchange the smallest element will be placed at a[0].

As a variation, the first pass could begin comparison with a[0] and a[1]. If a[0] is larger than a[1], the two elements are exchanged. Larger one will be placed in a[1] after the first comparison. Comparison can work forward and after the last comparison of a[n – 2] and a[n – 1], the larger element will be placed in a[n –1].

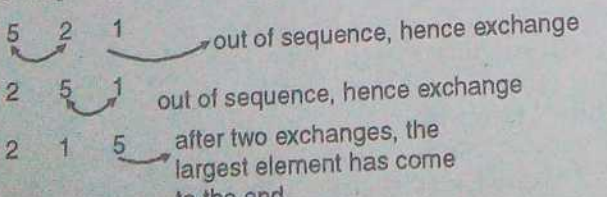


Fig. 5.2

First pass of the bubble sort with comparison in the forward direction.

The **second pass** is an exact replica of the first pass except that this time, the pass ends with the comparison and possible exchange of a[n – 3] and a[n – 2]. After the end of second pass, the second largest element will be placed at a[n – 2].

Bubble sort requires a total of n – 1 passes. If n – 1 elements are arranged (one element each pass) in ascending order from a[1] to a[n – 1], smallest element will finally left at a[0].

Q. 3 Explain in brief selection sort.

Ans. :

Selection sort

Selection sort is a very simple sorting method. In the i^{th} pass we select the element with lowest value among a[i], a[i + 1], ..., a[n – 1] and we swap it with a[i]. As a result, after i passes (pass number 0 to i – 1) first i elements will be in sorted order. Selection sort can be described by

```
for(i = 0; i < n – 1; i ++)
    select the smallest element among
        a[i], …. , a[n –1] and
    swap it with a[i];
```

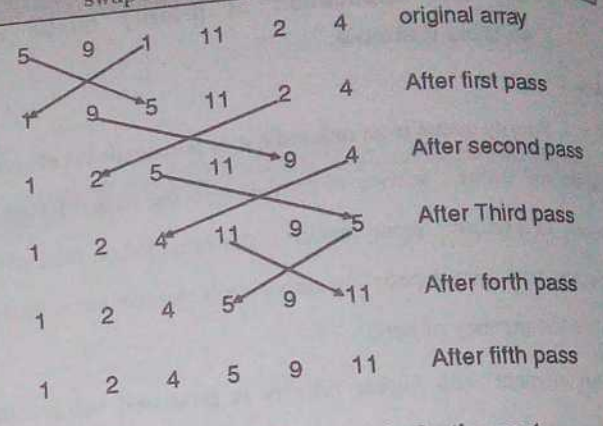


Fig. 5.3 : Illustration of selection sort

Analysis of selection sort

Selection sort is not data sensitive. In i^{th} pass, n – i comparisons will be needed to select the smallest element.

Thus, the number of comparisons needed to sort an array having n elements.

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

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

Q. 4 Compare the three sorting algorithms
(a) Bubble sort (b) Selection sort
(c) Insertion sort

Ans. :

A sorting algorithm can be judged on the basis of following parameters :

1. **Simplicity of algorithm**
   All the three sorting algorithms are equally simple to write.
2. **Timing complexity**
   Bubble sort and selection sort are not data sensitive. Both of them have a timing requirement of $O(n^2)$.
   Insertion sort is data sensitive. It works much faster if the data is partially sorted.
   Best case behaviour (when input data is sorted) = O(n).
   Worst case behaviour (when input data is in descending order) = $O(n^2)$.

3. **Sort stability**
All the three sorting algorithms are stable.
4. **Storage requirement**
No additional storage is required. All of them are in-place sorting algorithms.
5. All of them can be used for internal as well as external sorting.

Q. 5 Explain in brief insertion sort. Dec. 2013, May 2017

Ans. : Insertion Sort

An element can always be placed at a right place in sorted list of element for example

List of element s(sorted) 5 9 10 15 20

Element to be placed = 6

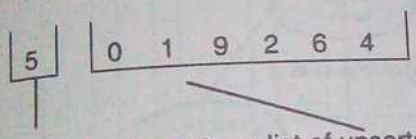
If the element 6 is to be inserted in a sorted list of elements (5, 9, 10, 15, 20), its rightful place will be between 5 and 9. Elements with value > 6 should be moved right by one place. Thus creating a space for the incoming element.

5 | 9 10 15 20 → Moved right by 1 place

5 6 9 10 15 20 → Element 6 is inserted between 5 and 9

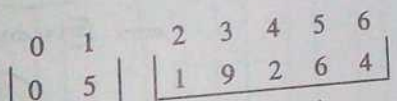
Insertion sort is based on the principle of inserting the element at its correct place in a previously sorted list. It can be varied from 1 to n – 1 to sort the entire array.

Index - 0 1 2 3 4 5 6 Initial unsorted list

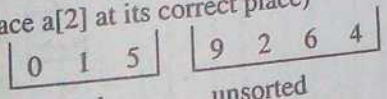


A list of sorted element (a list of single element is always sorted) a list of unsorted element

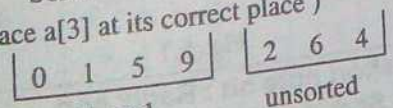
1st iteration (place element at location '1' i.e. a[1], at its correct place)



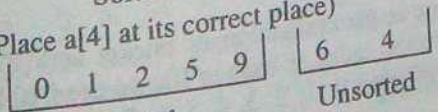
2nd iteration (place a[2] at its correct place)



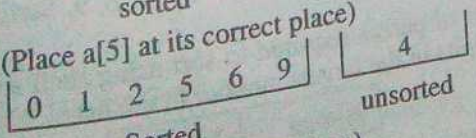
3rd iteration (place a[3] at its correct place)



4th iteration (Place a[4] at its correct place)



5th iteration (Place a[5] at its correct place)



6th iteration (Place a[6] at its correct place)

0 1 2 4 5 6 9

Fig. 5.4 : Sorting of elements using insertion sort

Q. 6 Write a program to sort an array using insertion sort algorithm. May 2015

Ans. :

```
#include<conio.h>
#include<stdio.h>
void insertion_sort(int[],int);
void main()
{
   int a[50],n,i;
   printf("\nEnter no of elements :");
   scanf("%d",&n);
   printf("\nEnter array elements :");
   for(i=0;i<n;i++)
         scanf("%d",&a[i]);
   insertion_sort(a,n);
   printf("\nSorted array is :");
   for(i=0;i<n;i++)
         printf("%d",a[i]);
   getch();
}
void insertion_sort(int a[],int n)
{
   int i,j,temp;
   for(i=1;i<n;i++)
   {
         temp=a[i];
         for( j=i-1;j>=0 && a[ j]>temp;j--)
            a[ j+1]=a[ j];
         a[ j+1]=temp;
   }
}
```

Output

```
Enter no of elements :  5
Enter array elements :  57  89  64  56  77
333
Sorted array is      :  56 57 64 77  89  333
```

Q. 7 Explain quick sort using an example. Write algorithm for it and comment on its complexity. May 2014, Dec. 2014, May 2015

Ans. :

Quick sort

Quick sort is the fastest internal sorting algorithm with the time complexity = O(n log n). The basic algorithm to sort an array a[ ] of n elements can be described recursively as follows :

1. If n < = 1, then return
2. Pick any element V in a[ ]. This is called the pivot. Rearrange elements of the array by moving all elements $x_i > V$ right of V and all elements $x_i \leq V$ left of V. If the place of the V after re-arrangement is j, all elements with value less than V, appear in a[0], a[1] … a[j – 1] and all those with value greater than V appear in a[j + 1] … a[n – 1].
3. Apply quick sort recursively to a[0] ….. a[j – 1] and to a[j + 1] …. a[n – 1]

Entire array will thus be sorted by as selecting an element V.

(a) partitioning the array around V.

(b) recursively, sorting the left partition.

(c) recursively, sorting the right partition.

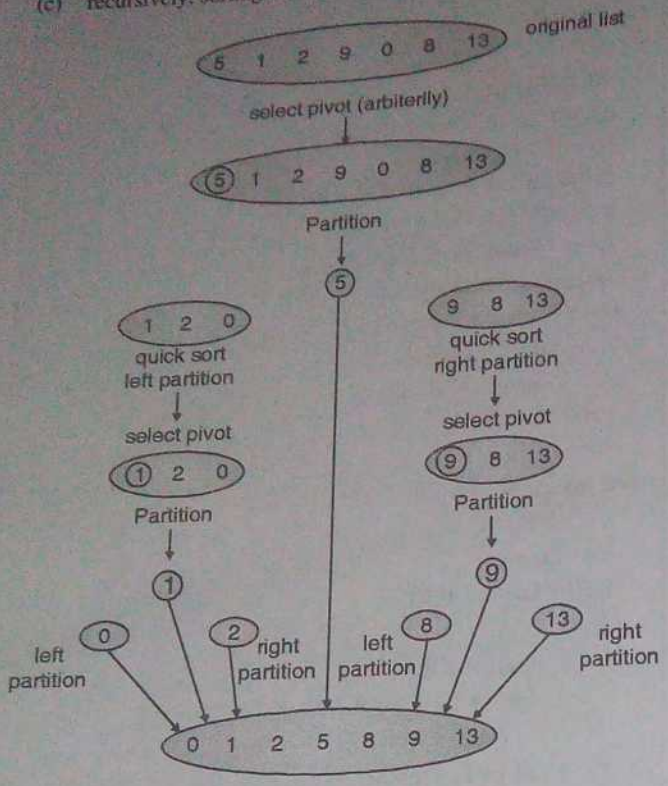


Fig. 5.5 : Illustration of quick sort

Q. 8 Sort the following elements using quick sort.
27, 76, 17, 9, 57, 90, 45, 100, 79

Ans. :

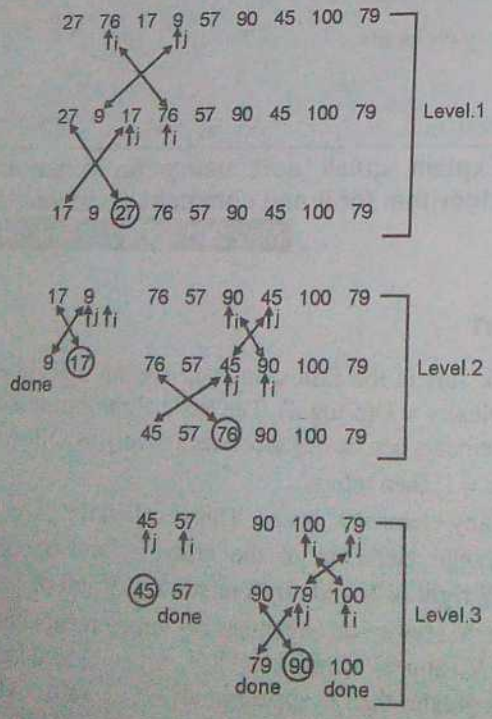


Fig. 5.6

Sorted data = 9 17 27 45 57 76 79 90 100

Q. 9 Sort the following list of numbers using Quick sort, show the result stepwise.

Ans. :

Data after each pass

Pass	Data
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

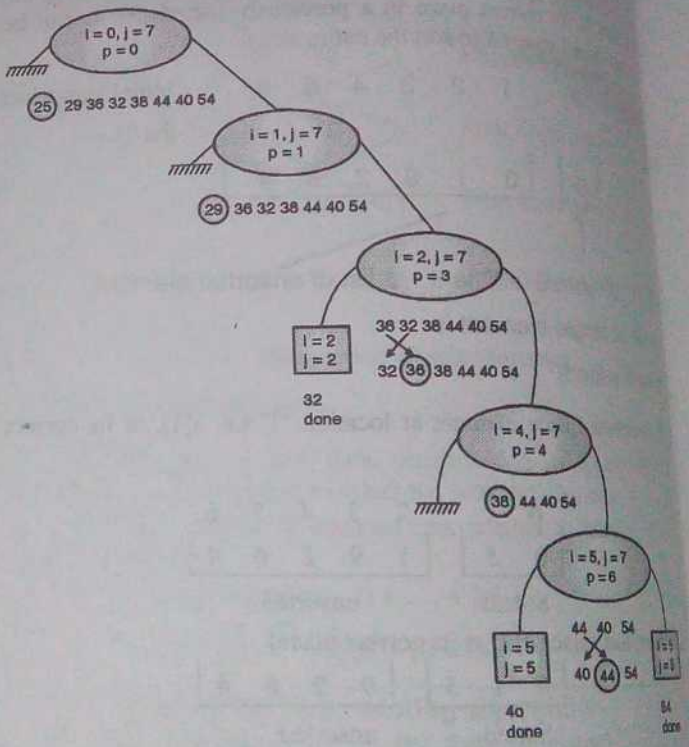


Fig. 5.7

Q. 10 Write short note on : Radix sort May 2016

Ans. : Radix Sort

Radix sort is generalization of bucket sort. To sort decimal numbers, where the radix or base is 10, we need 10 buckets. These buckets are numbered 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Sorting is done in passes.

Number of passes required to sort using shell sort is equal to number of digits in the largest number in the list.

Range	Passes
0 to 99	2 passes
0 to 999	3 passes
0 to 9999	4 passes

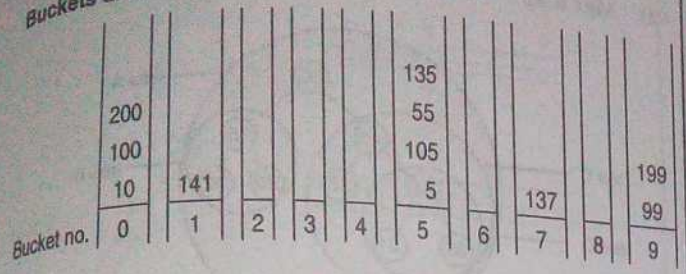
In the first pass, numbers are sorted on least significant digit. Numbers with the same least significant digit are stored in the same bucket.

In the 2nd pass, numbers are sorted on the second least significant digit. At the end of every pass, numbers in buckets are merged to produce a common list. Number of passes depends on the range of numbers being sorted. The following example shows the action of radix sort.

Initial numbers :

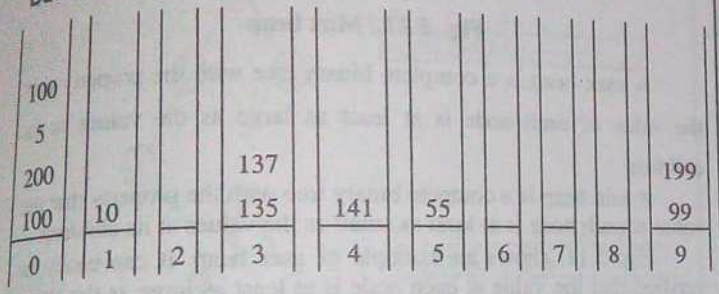
10 5 99 105 55 100 135 141 137 200 199

Buckets after 1st pass



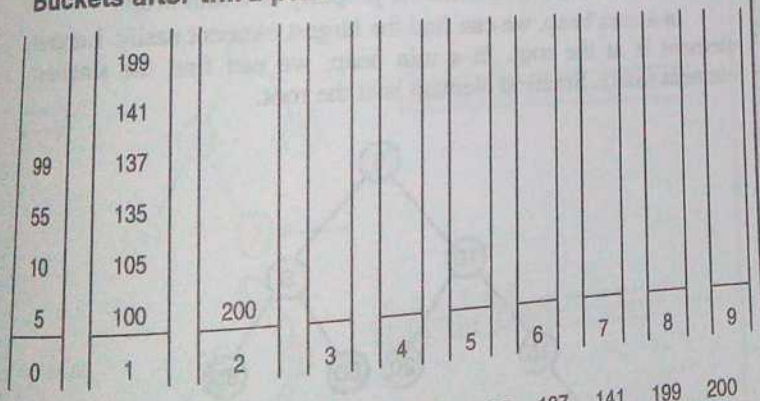
Merged list = 10 100 200 141 5 105 55 135 137 99 199

Buckets after 2nd pass



merged list = 100 200 5 105 10 135 137 141 55 99 199

Buckets after third pass



Merged list = 5 10 55 99 100 105 135 137 141 199 200

Q. 11 Write short note on : Merge sort. Write an algorithm for it and comment on its complexity. Dec. 2015, May 2016

Ans. :

Two-Way Merge Sort

Merge sort runs in O(N log N) running time. It is a very efficient sorting algorithm with near optimal number of comparisons. It is best described using a recursive algorithm. Basic operation in merge sort is that of merging of two sorted lists into one sorted list. Merging operation has a linear time complexity.

Recursive algorithm used for merge sort comes under the category of divide and conquer technique. An array of n elements is split around its centre. Producing two smaller arrays. After these two arrays are sorted independently, they can be merged to produce the final sorted array. The process of splitting and merging can be carried recursively till there is only one element in the array. An array with 1 element is always sorted. Let us try to understand the method through an example.

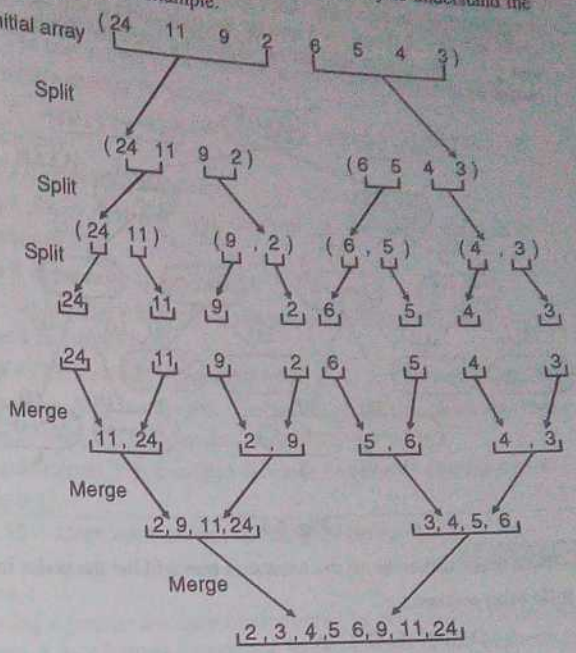


Fig. 5.8

Given array has 8 elements. Index of the first element is i = 0, index of the last element j = 7. In order to divide the above list around the middle element, the index of the centre element $k = \frac{i+j}{2} = \frac{0+7}{2} = 3$.

Merge sort is applied recursively to left half of the list from i = 0 to j = 3.

After sorting of the left half of the list. Right half of the list is sorted from i = 4 to j = 7 recursively using merge sort.

After both the lists are sorted, left list from i = 0 to j = 3 and right list from i = 4 to j = 7, these two lists are merged to produce a single sorted array.

'C' function for merge sort.

```
void mergesort(int a[ ], int i, int j)
  {
      int k;
      if(i < j)
      {
          k =(i + j)/2;
          mergesort(a, i, k);
          mergesort(a, k+1, j);
          merge(a, i, k, j);
      }
  }
```

Recursive function merge sort can be seen as postorder traversal of the binary tree :
Where traversal of the left subtree is given by – mergesort (a, i, k);
And traversal of the right subtree is given by –
mergesort(a, k + 1 , j);
Visiting a node is given by merge(a, i, k, j).

If the root node starts with an array (24, 11, 9, 2, 6, 5, 4, 3) having eight elements. Value of i and j for the root node will be i = 0 and j = 7 respectively. After split, value of i and j for the left child will be i = 0 and j = 3 and for the right child it will be i = 4 and j = 7. Recursion will go on expanding until the list can no longer be divided.

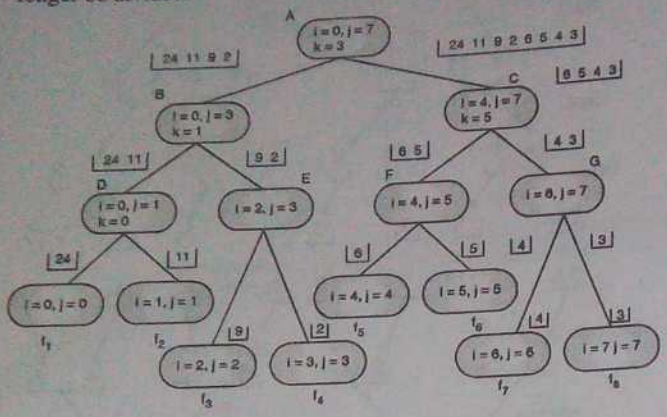


Fig. 5.9

Post order traversal on the recursion tree will list the nodes in the following sequence :

D E B F G C A

Visit D[merge (24) (11) giving (11, 24)]

Visit E[Merge (9) (2) giving (2, 9)]

Visit B[Merge (11, 24)(2, 9) giving (2, 9, 11, 24)]

Visit F[Merge (6) (5) giving (5, 6)]

Visit G[Merge (4) (3) giving (3, 4)]

Visit C[Merge (5, 6) (3, 4) giving (3, 4, 5, 6)]

Visit A[Merge (2, 9, 11, 24), (3, 4, 5, 6) giving (2, 3, 4, 5, 6, 9, 11, 24)]

Q. 12 Define heaps with its types.

Ans. :

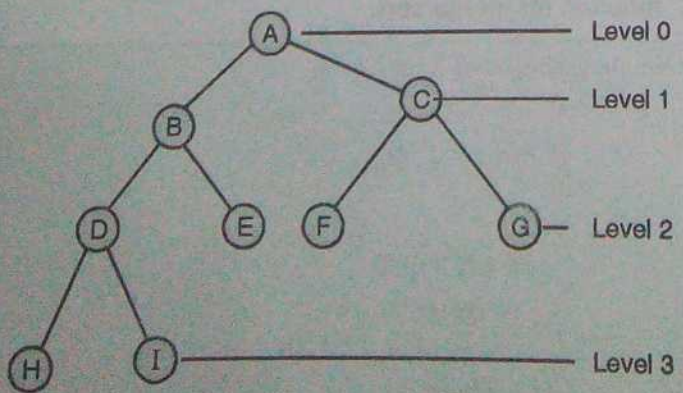


Fig. 5.10 : Complete binary tree

A heap is a **complete** binary tree. In a complete binary tree, elements are filled level by level from left to right. Thus the bottom level may not be completely filled.

A complete binary tree is shown in the Fig. 5.10. First three levels (i.e. level 0, level 1 and level 2) are completely filled. Elements at the level 3 are getting added from left to right.

Types of Heaps

(a) Max heap (b) Min heap

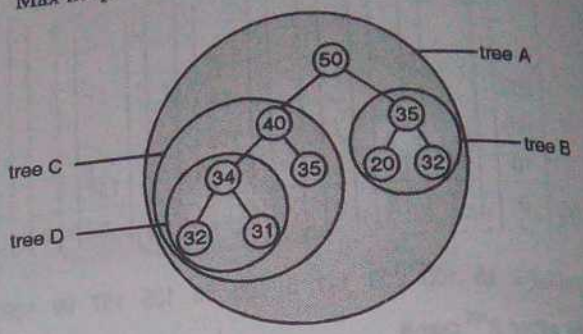


Fig. 5.11 : Max heap

A max heap is a complete binary tree with the property that the value of each node is at least as large as the values at its children.

A min heap is a compete binary tree with the property that the value at each node is at least as small as the values at its children.

Fig. 5.11, shows an example of max heap. It can easily be verified that the value at each node is at least as large as the value at its children.

In Fig. 5.12, value at each node is at least as small as the value at its children, it satisfies the property of a min heap.

In a max heap, we can find the largest element easily. Largest element is at the root. In a min heap, we can find the smallest element easily. Smallest element is at the root.

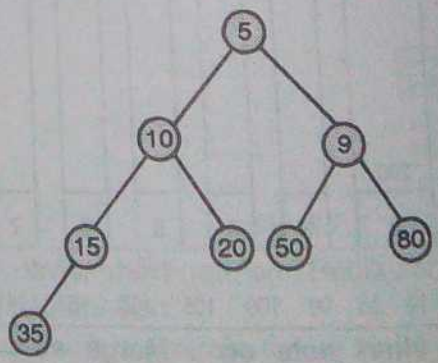
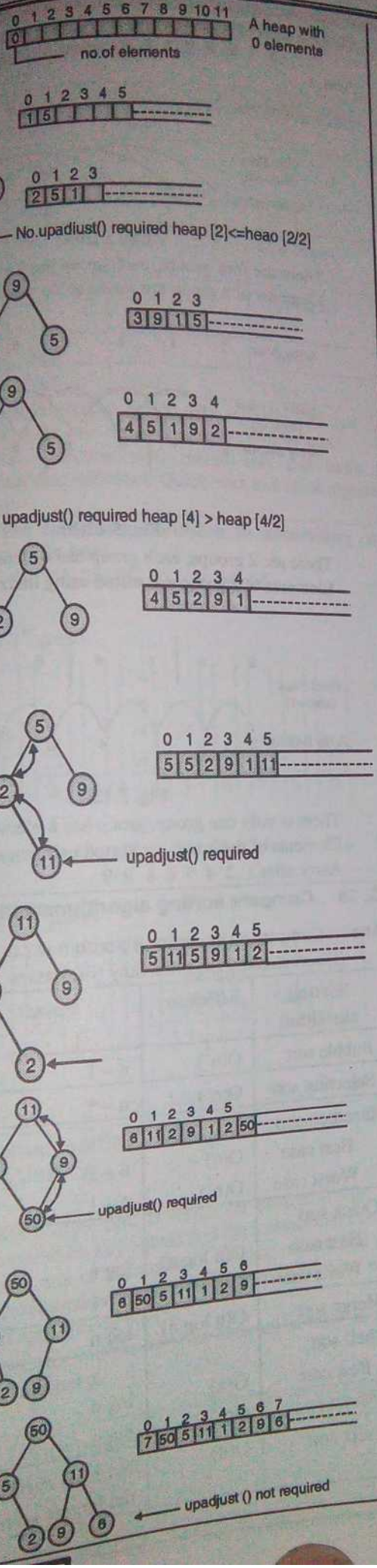
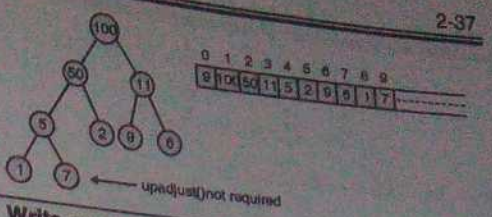


Fig. 5.12 : Min heap

Q. 13 Create a max heap with following elements : 5, 1, 9, 2, 11, 50, 6, 100, 7

Ans. : Let us assume that heap is represented using an array heap[12].





Q. 14 Write an algorithm to implement Heap- sort

Ans. : Application of heap sort Dec. 2016

The best known application of heap is in sorting. It is uses a very simple strategy for sorting.

A heap can be sorted through repeated application of deletemax();

Step 1 : Construct a max-heap.

Step 2 : Swap A[1] and A[n], this will put the largest element of the array in A[n].

Step 3 : Reduce the heap size by 1.

i.e. heap[0] = heap[0] – 1; largest element is in the array but it is not a part of the heap.

Step 4 : The new tree represented by A[1 .. n – 1] may no longer be a heap, as A[1] is changed. The new tree can be converted to a heap by using the function downadjust().

Step 5 : Repeat step 2 to step 4, while number of elements in the heap > 1.

Q. 15 Comment on complexity of heap sort. Dec. 2016

Ans. :

Sorting algorithm has been implemented in two phases. In the first phase, a max-heap is created and in the second phase element are sorted through delete max operation.

Complexity of heap creation

Let us assume that heap contains N element.

Number of elements at the last level (worst case) = N / 2.

Number of elements at the last but one level (worst case) = N / 4.

:

:

Number of elements at the 0^{th} level = 1

In the function create (), n/2 elements are down-adjusted. If the element to be down-adjusted is at height h then it may require 2h comparisons.

$\frac{N}{4}$ elements are at height = 1, requiring $2 \times \frac{N}{4} \times 1$ comparisons.

$\frac{N}{8}$ elements are at height = 2, requiring $2 \times \frac{N}{8} \times 2$ comparisons.

$\frac{N}{16}$ elements are at height = 3, requiring $2 \times \frac{N}{8} \times 3$ comparisons.

:

:

$$\therefore \text{ Total number of comparisons} = \frac{2N}{4} + \frac{2N}{8} \times 2 + \frac{2N}{16} \times 3 + \dots$$

$$= \frac{2N}{4}\left[\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \frac{4}{2^3} + \dots\right]$$

Let us say, $S = \frac{2N}{4}\left[\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \frac{4}{2^3} + \dots\right]$

$$\therefore \frac{2S}{N} = \frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \frac{4}{2^3} + \dots \quad \dots (1)$$

$$\therefore \frac{2S}{N} \times \frac{1}{2} = \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \dots \quad \dots (2)$$

from Equation (1) – Equation (2)

$$\frac{2S}{N} - \frac{S}{N} = \frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \dots$$

$$\therefore \frac{S}{N} = 2$$

$$\therefore S = 2N = O(N)$$

Complexity of Sorting

Worst case complexity of downadjust ( ) is ( ).

$O(\log_2 N)$ [$\log_2 N$ is the height of a heap with N elements].

Hence the total complexity of the heap sort algorithm =

$$\sum_{i=1}^{N} \log_2 i = \log_2 1 + \log_2 2 + \dots + \log_2 N = \log_2 (1 \times 2 \times 3 \times \dots \times N) = \log_2 (N!).$$

$\log_2 (N!)$ can be approximated to $O(N \log_2 N)$ for large value of N.

Overall complexity = max $(O(N), O(N \log_2 N)) = O(N \log_2 N)$

Q. 16 Explain in brief shell sort. Write an algorithm to implement shell sort. Dec. 2013, Dec. 2015

Ans. : Shell sort

This method makes repeated use of insertion sort. If an array of n elements is to be sorted using shell sort then sorting requires that a number d_i called increment should be chosen before every pass. d_i should be less than n. d_i should diminish with every pass and for the last pass d_i should be 1.

Initial value of increments d_i can be taken as n/2 and in subsequent passes d_i should be halved.

i.e.

$d_1 = n/2$

$d_2 = d_1/2$

$d_3 = d_2/2$

$d_{i+1} = d_i/2$

If an array a[0] … a[7], has 8 elements then the initial value of increment $d_1 = 8/2 = 4$ with increment as 4, original array will be divided into four sub-lists :

1st sub list : (a [0], a[0 + 4])

2nd sub list : (a [1], a[1 + 4])

3rd sub list : (a [2], a[2 + 4])

4th sub list : (a [3], a[3 + 4])

Each of these lists is sorted independently using insertion sort.

In the second pass, the value of increment d_2 is taken as

$d_2 = d_1/2 = 4/2 = 2$

Array will be divided into two sub lists :

1st sub list : (a[0], a[0 + 2], a[0 + 4], a[0 + 6])

2nd sub list : (a[1], a[1 + 2], a[1 + 4], a[1 + 6])

each of these lists is sorted independently using insertion sort.

In the third pass the value of increment d_3 is taken as

$d_3 = d_2/2 = 2/2 = 1$

Since, the value of increment is 1, entire array with elements (a[0], a[1], a[2] … a[6], a[7]) will be sorted using insertion sort.

Q. 17 Sort the following sequence of integers (5 1 9 8 2 4 6 9) using shell sort

Ans. :

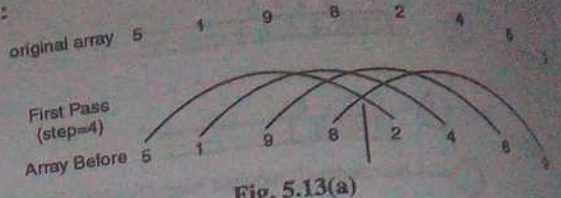


Fig. 5.13(a)

There are four groups, each group has 2 elements. Elements of a group are sorted using insertion sort.

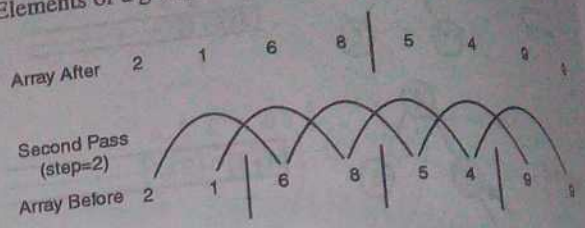


Fig. 5.13(b)

There are 2 groups, each group has 4 elements. Elements of a group are sorted using insertion sort.

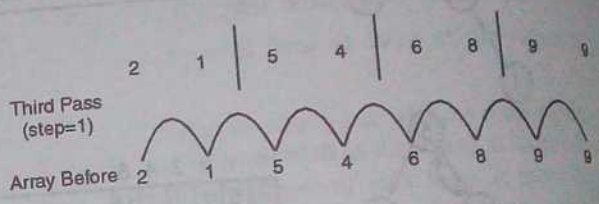


Fig. 5.13(c)

There is only one group, group has 8 elements. Elements of the group are sorted using insertion sort.

Array after 1 2 4 5 6 8 9 9

Q. 18 Compare sorting algorithms with its parameters.

Ans. : Comparison of sorting algorithm w.r.t.

(i) sort stability (ii) efficiency (iii) passes

Sorting algorithm	Efficiency	Passes	Sort stability
Bubble sort	$O(n^2)$	n – 1	Stable
Selection sort	$O(n^2)$	n – 1	Stable
Insertion sort			Stable
Best case	O(n)	n – 1	
Worst case	$O(n^2)$	n – 1	
Quick sort			Unstable
Best case	O(n log n)	log n	
Worst case	$O(n^2)$	n – 1	
Merge sort	O(n log n)	log n	Stable
Shell sort			Unstable
Best case	O(n)	log n	
Worst case	$O(n^2)$	log n	
Radix sort	O(n)	No. of digits in the largest number	Stable

Q. 19 Give best-case, worst-case and average-case analysis of sorting algorithm .

Ans. :

	Best-case	Worst-case	Average-case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Quick sort	O(n log n)	$O(n^2)$	O(n log n)
Merge sort	O(n log n)	O(n log n)	O(n log n)
Shell sort	O(n)	$O(n^2)$	$O(n^2)$
Radix sort	O(n)	O(n)	O(n)

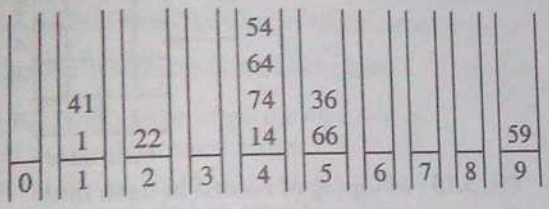
If the sorting algorithm is not data sensitive, its best case, worst case and average case timing behaviour will be same.

Bubble sort, selection sort, merge sort and radix sort algorithms are not data sensitive. Quick sort and shell algorithms are data sensitive.

Q. 20 Sort the following numbers in ascending order using radix sort
14, 1, 66, 74, 22, 36, 41, 59, 64, 54

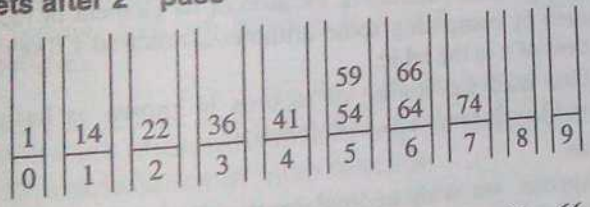
Ans. :

Buckets after 1st pass



merged list = 1 41 22 14 74 64 54 66 36 59

Buckets after 2nd pass



merged list = 14 22 36 41 54 59 64 66 74

Q. 21 "External sorting calls for internal sorting as well". Justify.

Ans. :

Every sorting algorithm is based on passes. Inside a pass, records are compared on key values and the records can be shuffled based on the outcome of comparison. In case of external sorting, records to be compared are read into memory variables and they are re-written at appropriate places.

Pseudo code for bubble sort for external sorting is given to explain the above concept.

```
n ← no of records in the file ;
rec1, rec2, : record type ;
for (i = 1 ; i < n ; i ++)
    for (j = 0 ; j < n – i ; j ++)
    {
        rec1 ← read the jth record ;
        rec2 ← read the (j + 1)th record ;
        if (rec1 · key > rec2 · key)
        {
            write rec2 at jth place ;
            write rec1 at (j + 1) th place ;
        }
    }
```


Q. 22 Write note on searching algorithms. Dec. 2016

Ans. :

Searching is a technique of finding an element in a given list of elements. List of elements could be represented using an :

(a) Array (b) Linked List
(c) Binary tree (d) B-tree
(e) Heap

Elements could also be stored in file. Searching technique should be able to locate the element to be searched as quickly as possible. Many a time, it is necessary to search a list of records to identify a particular record. Usually, each record is uniquely identified by its key field and searching is carried out on the basis of key field. If search results in locating the desired record then the search is said to be successful. Otherwise, the search operation is said to be unsuccessful. The complexity of any searching algorithm depends on number of comparisons required to find the element. Performance of searching algorithm can be found by counting the number of comparisons in order to find the given element.

Q. 23 Explain binary search.

Ans. : Binary search

Linear search has a time complexity O(n). such algorithms are not suitable for searching when number of elements is large. Binary search exhibits much better timing behaviour in case of large volume of data with timing complexity $O(\log_2 n)$.

Number of comparisons for $n = 2^{20}$ (1 million)

Sequential search (in worst case) = 2^{20} comparisons.

Binary search (in worst case) = $\log_2 2^{20} = 20$ comparisons.

Linear search (sequential search) may need 1 million comparisons for searching an element in an array having 1 million elements. Binary search will require, just 20 comparisons for the same task. Binary search uses a much better method of searching. Binary search is applicable only when the given array is sorted.

This method makes a comparison between the "key" (element to be searched) and the middle element of the array.

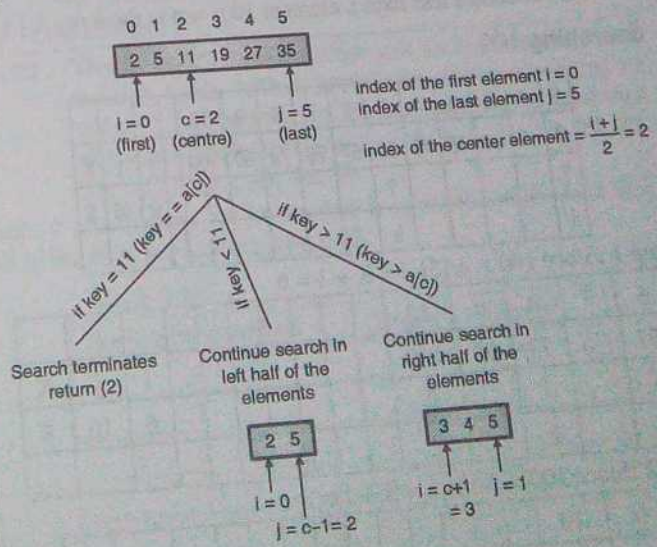


Fig. 5.14

Since elements are sorted, comparisons may result in either a match or comparison could be continued with either left half of elements or right half of the elements.

Left half of elements could be selected by simply making j = c–1

Right half of element could be selected by simply making i = c+1

Process of selecting either the left half or the right half continues until the element is found or element is not there.

Q. 24 Apply binary search on the following numbers stored in array from A [0] to A [10]
9, 17, 23, 38, 45, 50, 57, 76, 79, 90, 100
to search numbers – 10 to 100.

Ans. :

Searching 10

0	1	2	3	4	5	6	7	8	9	10					
9	17	23	38	45	50	57	76	79	90	100			i	j	k
↑					↑					↑			0	10	5
i					k					j					

Step 1 : Since 10 < A [5], j = K – 1 = 4

0	1	2	3	4									i	j	k
9	17	23	38	45									0	4	2
↑		↑		↑											
i		k		J											

Step 2 : Since 10 < A [2], j = K – 1 = 1

0	1											
9	17								i	j	k	
↑↑	↑								0	1	0	
i k	j											

Step 3 : Since 10 > A [0], j = K + 1 = 1

				i	j	k
1				1	1	1
17						
↑ ↑ ↑						
i j k						

Step 4 : Since 10 < A [1], j = K – 1 = 0

As i becomes less than j, element 10 is not in the array A [ ]

Searching 100

0	1	2	3	4	5	6	7	8	9	10				
9	17	23	38	45	50	57	76	79	90	100		i	j	K
↑					↑					↑		0	10	5
i					k					j				

Step 1 : Since 100 > A [5], i = K + 1 = 6

6	7	8	9	10										
57	76	79	90	100								i	j	K
↑		↑		↑								6	10	8
I		k		j										

Step 2 : Since 100 > A [8], i = K + 1 = 9

9	10													
90	100											i	j	K

↑↑	↑								9	10	9
i	j										
k											

Step 3 : Since 100 > A [9], i = K + 1 = 10

10										i	j	K
100										10	10	10
↑ ↑ ↑												
i j k												

Since, the element to be searched is found at A [10], search terminates with a success.

Q. 25 What is hashing ?

Ans. :

There is widely used technique for storing of data called "hashing". It does away with the requirement of keeping data sorted (as in binary search) and its best case timing complexity is of constant order (O(1)). In its worst case, hashing algorithm starts behaving like linear search.

Best case timing behaviour of searching using hashing = O(1)

Worst case timing Behaviour of searching using hashing = O(n).

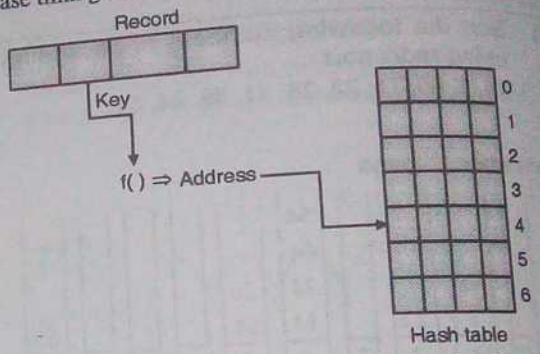


Fig. 5.15 : Mapping of record in hash table

Since, there is a large gap between its best case O(1) and worst case O(n) behaviour. It should be implemented properly to get an average case behaviour close to O(1). In hashing , the record for a key value "key". is directly referred by calculating the address from the key value. Address or location of an element or record, x, is obtained by computing some arithmetic function f.f(key) gives the address of x in the table.

Table used for storing of records is known as hash table. Function f(key)is known as hash function.

Example

Suppose, we wish to implement a hash table for a set of records where the key is a member of set. Set K of strings.

K = {"aaa", "bbb", "ccc", "ddd" "eee", "fff", "ggg"}

A function f : key → Index is given by the following table:

N	f(x)
"aaa"	0
"bbb"	1
"ccc"	2
"ddd"	3
"eee"	4
"fff"	5
"ggg"	6

Hash table can be implement using an array of records of length n = 7. To store a record with key x, we simply store it at position f(x) in the array. Similarly, to locate the record having key = x, we simply cheek to see if it is found at position f(x).

Q. 26 Explain different forms of hashing.

Ans. : There are two different forms of hashing.

(a) Open hashing or external hashing

(b) Close hashing or internal hashing

Open or external hashing, allows records to be stored in unlimited space (could be a hard disk). It places no limitation on the size of the tables. Closed or internal hashing, uses a fixed space for storage and thus limits the size of hash table.

(a) Open Hashing Data Structure

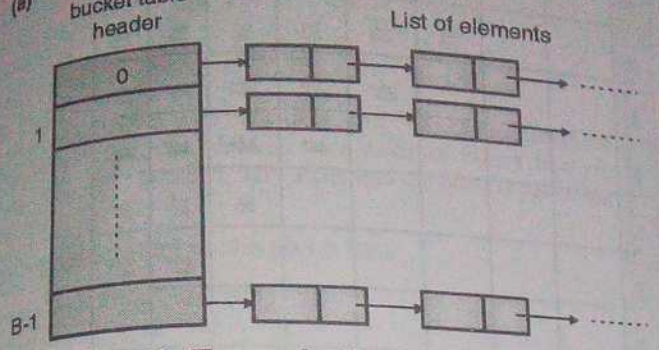


Fig. 5.16 : The open hashing data organization

Fig. 5. 16 gives the basic data structure for open hashing.

The basic idea is that the records [elements] are partitioned into B classes, numbered 0,1,2.. B–1. Hashing function f(x) maps a record with key n to an integer value between 0 and B–1. if a record is mapped to location 1 then we say the record is mapped to bucket 1 or the record belongs to class 4. Each bucket in the bucket table is the head of the linked list of records mapped to that bucket.

(b) Closed Hashing Data Structure

A closed hash table keeps the elements in the bucket itself. Only one element can be put in the bucket. If we try to place an element in the bucket f(n) and find it already holds an element, then we say that a collision has occurred. In case of collision, the element should be rehashed to alternate empty location $f_1(x)$, $f_2(x)$, ... within the bucket table. In closed hashing, collision handling is a very important issue.

0	b
1	
2	
3	a
4	c
5	d
6	
7	

Fig. 5.17 : Partially filled hash table

Q. 27 What are characteristics of a good hash function ?

Ans. : Characteristics of hash function

(i) A good hash function avoids collisions.

(ii) A good hash function tends to spread keys evenly in the array.

(iii) A good hash function is easy to compute.

(iv) There are many hashing functions like division method, midsquare methods, folding method, digit analysis, length dependent method, algebraic coding and multiplicative hashing.

Q. 28 What are collision resolution strategies ? Explain.

Ans. : Collision Resolution Strategies (Synonym Resolution)

Collision resolution is the main problem in hashing. If the element to be inserted is mapped to the same location, where an element is already inserted then we have a collision and it must be resolved. There are several strategies for collision resolution. The most commonly used are :

(a) Separate chaining – used with open hashing.

(b) Open addressing – used with closed hashing.

(a) Separate Chaining

In this strategy, a separate list of all elements mapped to the same value is maintained. Fig. 5.18 shows an implementation of separate chaining. Separate chaining is based on collision avoidance. If memory space is tight, separate chaining should be avoided. Additional memory space for links is wasted in storing address of linked elements.

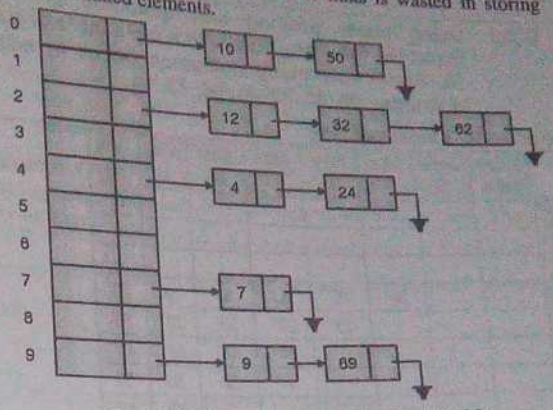


Fig. 5.18 : A separate chaining hash table

Hashing function should ensure even distribution of elements among buckets, otherwise the timing behavior of most operations on hash table will deteriorate.

In the Fig. 5.18 : A simple hash function hash (x)= x mod 10 is taken. Bucket table has a size of 10.

To perform a **find**, hash function is used to find the list to be traversed for searching. Then the list is traversed in normal manner. To perform an **insert**, the appropriate list is traversed to ensure that the element to be inserted is not present in the list. If the element is not present then it is inserted at the front.

(b) Open Addressing

Separate chaining requires additional memory space for pointers. Open addressing hashing is an alternate method of handling collision. In open addressing, if a collision occurs, alternate cells are tried until an empty cell is found. Because all the data elements are stored inside the table, a larger memory space is needed for open addressing. Generally, the load factor should be below 0.5 for open addressing hashing. There are three commonly used collision resolution strategy in open addressing.

(1) Linear probing (2) Quadratic probing

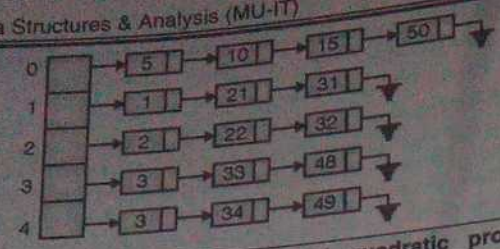
(3) Double hashing.

Q. 29 The integers gives below are to be inserted in a hash table with 5 locations using chaining to resolve collisions. Construct hash table and use simplest hash function.1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50

Ans. : An element can be mapped to a location in the hash table using the mapping function key %10.

Hash table

Hash table locations	Mapped elements
0	5, 10, 15, 50
1	1, 21, 31
2	2, 22, 32
3	3, 33, 48
4	4, 34, 49



Q. 30 Using linear probing and quadratic probing insert the following values in a hash table of size 10. Show how many collisions occur in each technique :99, 33, 23, 44, 56, 43, 19

Ans. : 1. Linear probing

	Empty table	After 99	After 33	After 23	After 44	After 56	After 43	After 19
0								19*
1								
2								
3			33	33	33	33	33	33
4				23*	23*	23*	23*	23*
5					44*	44*	44*	44*
6						56	56	56
7							43*	43*
8								
9		99	99	99	99	99	99	99

* = Collision
No. of collisions = 4

2. Quadratic probing

	Empty table	After 99	After 33	After 23	After 44	After 56	After 43	After 19
0								19*
1								
2								
3			33	33	33	33	33	33
4				23*	23*	23*	23*	23*
5					44*	44*	44*	44*
6						56	56	56
7							43*	43*
8								
9		99	99	99	99	99	99	99

* = Collision
No. of collisions = 4

Chapter 6 : Trees

Q. 1 Define binary tree .

Ans. : Binary tree

A tree is binary if each node of the tree can have maximum of two children. Moreover, children of a node of binary tree are ordered.

One child is called the "left" child and the other is called the "right" child. An example of binary tree is shown in Fig. 6.1.

Node A has two children B and C. Similarly, nodes B and C, each have one child name G and D respectively.

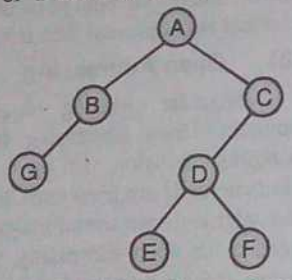
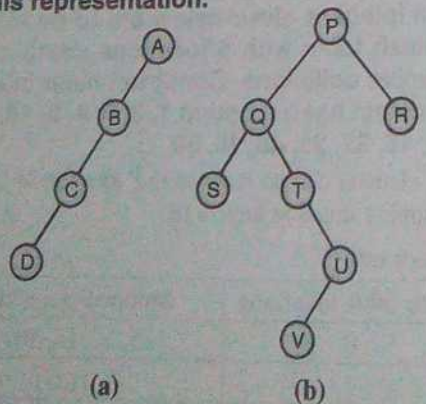


Fig. 6.1 : A binary tree

Q. 2 Explain array representation of binary trees using the Fig. 6.2.State and explain limitations of this representation.



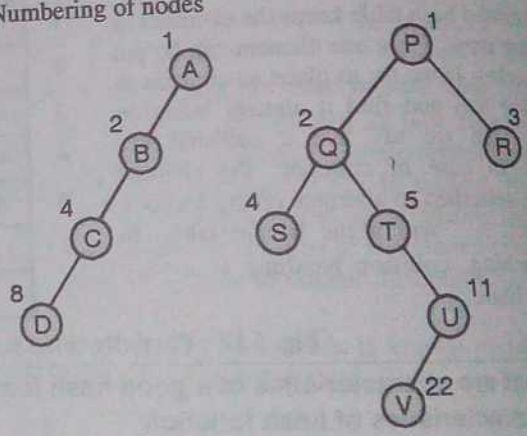
(a) (b)

Fig. 6.2

Ans.

In order to represent a tree in a single one-dimensional array, the nodes are numbered sequentially level by level from left to right. Even empty nodes are numbered. Location number zero of the array can be used to store the size of the tree in terms of total number of nodes (existing or not existing).

Step 1 : Numbering of nodes



Step 2 : Representation :

Left tree

0	1	2	3	4	5	6	7	8
15	A	B		C				D

Right tree

0	1	2	3	4	5	11	22
31	P	Q	R	S	T	U	V

Array representation is less efficient for a sparse tree. In array representation, memory is allocated even for empty nodes. Array representation also suffers from the problem of underflow. The size of the array is fixed during compile time.

Q. 3 Define a binary tree. Show the sequential representation of the binary tree given.

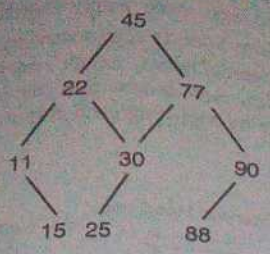


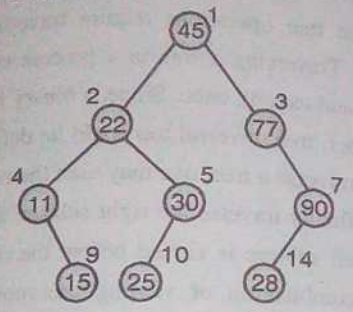
Fig. 6.3

Ans. :

A tree is binary if each node of the tree can have maximum of two children. Moreover, children of a node of binary tree are ordered. One child is called "left" child and the other is called the "right" child.

Array representation of the given tree

Step 1 : Numbering of nodes :



Step 2 : Representation

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	45	22	77	11	30		90		15	25				28	

Q. 4 Explain : (i) Degree of tree (ii) Height of tree (iii) Depth of tree.

May 2015, Dec. 2015, May 2016

Ans. :

(i) **Degree of tree :** Number of sub trees of a node.

(ii) **Height of tree :** Height of a tree is height of the root node.

(iii) **Depth of tree :** The depth of a node is the number of edges from the node to the tree's root node. Depth of the tree is the depth of deepest node.

Q. 5 Define general tree.

Ans. : General Tree

In a general tree, number of children per node is not limited to two. Since, the number of children per node can vary greatly, it might not be feasible to make the children direct links in the node. Children of a node can be stored in a linked list with parent node storing the address of its leftmost child.

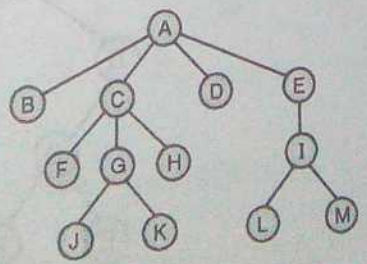


Fig. 6.4 : A tree

Q. 6 What is the necessity of converting a tree into binary tree ? Given the following tree :

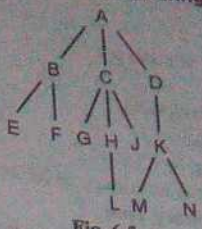


Fig. 6.5

Covert it into a binary tree and list down the steps for the same.

Ans. :

Algorithms for binary tree are simple and widely used. Thus it is easy to give a formal treatment to a binary tree. A tree can be converted into a binary tree using leftmost child right sibling relation.

The left pointer points to the leftmost child.

The right pointer points to its next sibling.

Equivalent binary tree

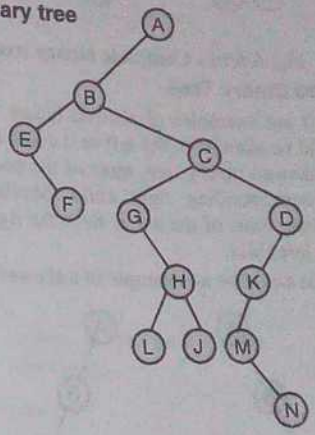


Fig. 6.5(a)

Q. 7 Explain types of binary tree

May 2014

Ans. : Types of Binary Tree

(i) Full Binary Tree

A binary tree is said to be full binary tree if each of its node has either two children or no child at all. Every level is completely filled. Number of node at any level i in a full binary tree is given by 2^i. A full binary tree is shown in Fig. 6.6(a).

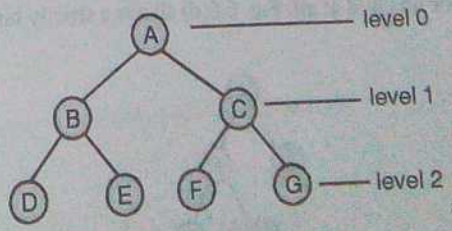


Fig. 6.6(a) : Full binary tree of depth 3

Total number of nodes in a full binary tree of height

$$h = 2^0 + 2^1 + 2^2 \ldots + 2^h = 2^{h+1} - 1$$

The height of the tree shown is Fig. 6.6.1(a) is 2 and number of nodes is given by

$$2^{2+1} - 1 = 8 - 1 = 7.$$

Hence number of nodes $n = 2^{h+1} - 1$

Or $2^{h+1} = n + 1$

Taking log on both sides.

$h + 1 = \log_2 (n + 1)$

$\therefore h = (\log_2 (n + 1)) - 1$

Tournament tree is an example of full binary tree.

(ii) Complete Binary Tree

A complete binary tree is defined as a binary tree where

(i) All leaf nodes are on level n or n – 1.

(ii) Levels are filled from left to right.

Examples of a complete binary tree are shown in Fig. 6.6(b). Heap is an example of complete binary tree.

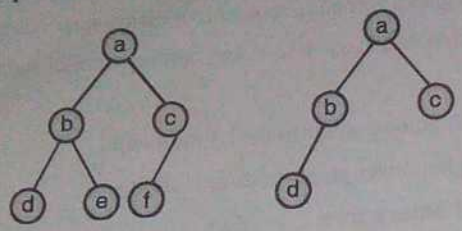


Fig. 6.6(b) : Complete binary trees

(iii) Skewed Binary Tree

Fig. 6.6(c) are examples of skewed binary trees. A skewed binary tree could be skewed to the left or it could be skewed to the right. In a left skewed binary tree, most of the nodes have the left child without corresponding right child. Similarly, in a right skewed binary tree, most of the nodes have the right child without a corresponding left child.

Binary search tree could be an example of a skewed binary tree.

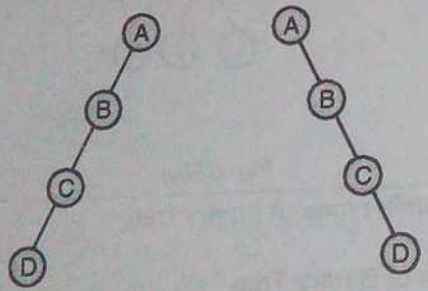


Fig. 6.6(c) : Skewed Binary trees

(iv) Strictly Binary Tree

If every non-terminal node in a binary tree consists of non-empty left subtree and right subtree, then such a tree is called strictly binary tree. In other words, a node will have either two children or no child at all. Fig. 6.6(d) shows a strictly binary tree.

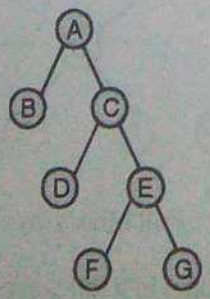


Fig. 6.6(d) : Strictly binary tree

(v) Extended Binary Tree (2-Tree)

In an extended tree, each empty subtree is replaced by a failure node. A failure node is represented by □. Nodes with children are called internal nodes, and the nodes with 0 children are called external nodes. Any binary tree can be converted into extended binary tree by replacing each empty subtree by a failure node.

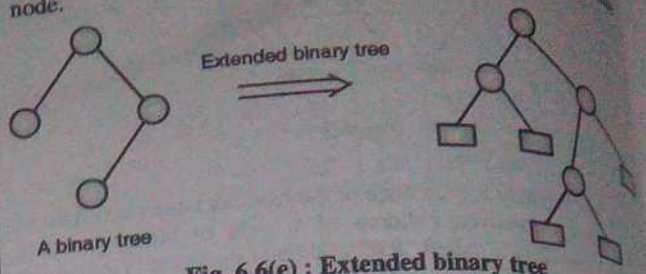


Fig. 6.6(e) : Extended binary tree

Q. 8 What are binary tree traversal techniques ? Explain with example.

Ans. : Binary tree traversal techniques

Most of the tree operations require traversing a tree in a particular order. Traversing a tree is a process of visiting every node of the tree and exactly once. Since, a binary tree is defined in a recursive manner, tree traversal too could be defined recursively. For example, to traverse a tree, one may visit the root first, then the left subtree and finally traverse the right subtree. If we impose the restriction that left subtree is visited before the right subtree then three different combination of visiting the root, traversing left subtree, traversing right subtree is possible.

1. Visit the root, traverse, left subtree, traverse right subtree.
2. Traverse left subtree, visit the root, traverse right subtree.
3. Traverse left subtree, traverse right subtree, visit the root.

These three techniques of traversal are known as preorder, inorder and postorder traversal of a binary tree.

(i) Preorder Traversal (Recursive)

The functioning of preorder traversal of a non-empty binary tree is as follows :

1. Firstly, visit the root node (visiting could be as simple as printing the data stored in the root node).
2. Next, traverse the left subtree in preorder.
3. At last, traverse the right-subtree in preorder.

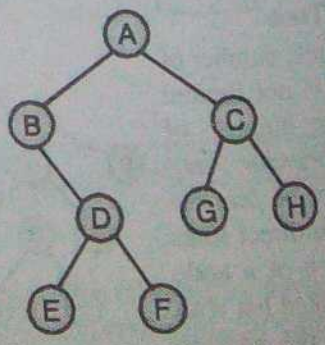


Fig. 6.7(a) : A sample binary tree

Stepwise preorder traversal of tree is shown in Fig. 6.7(b) is given below :

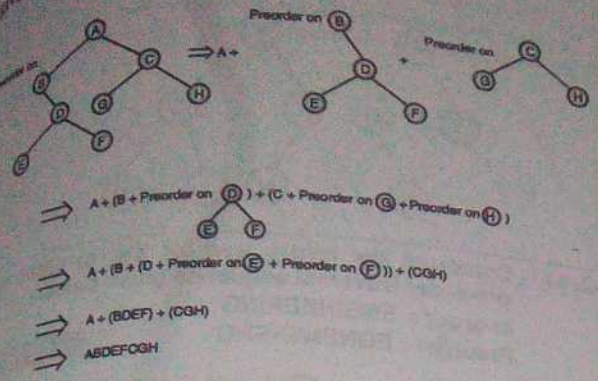


Fig. 6.7(b)

(ii) Inorder Traversal (Recursive)

The functioning of inorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in inorder.
2. Next, visit the root node.
3. At last, traverse the right subtree in inorder.

Stepwise inorder traversal of tree shown in Fig. 6.7(c)

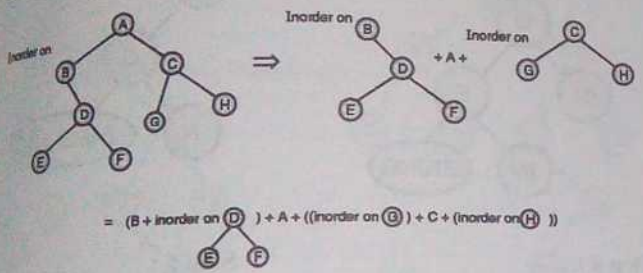


Fig. 6.7(c)

(iii) Postorder Traversal (Recursive)

The functioning of postorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in postorder.
2. Next, traverse the right subtree in postorder.
3. At last, visit the root node.

Stepwise postorder traversal of tree shown in Fig. 6.7(d) is given below.

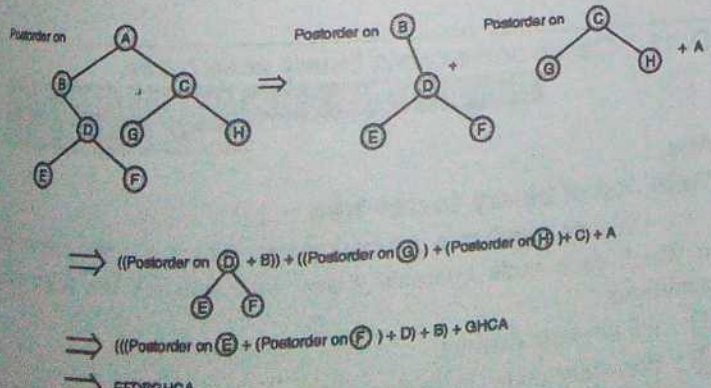


Fig. 6.7(d)

Q. 9 Traverse the following binary tree into preorder and inorder and postorder with reason.

May 2014

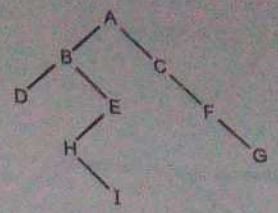


Fig. 6.8

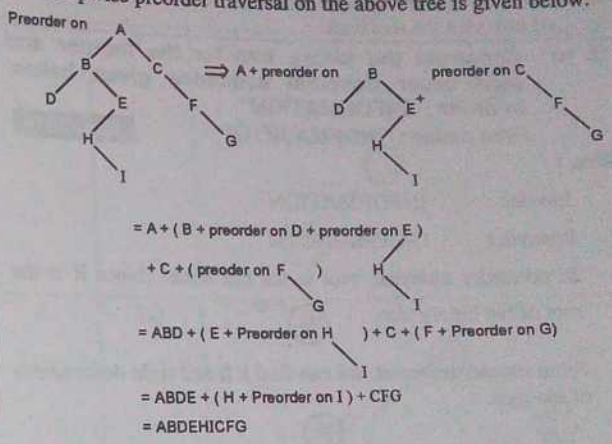
Ans. :

Preorder traversal

The following of preorder traversal of a non-empty binary tree is as follows :

1. Firstly, visit the root node.
2. Next, traverse the left subtree in preorder.
3. At last, traverse the right subtree in preorder.

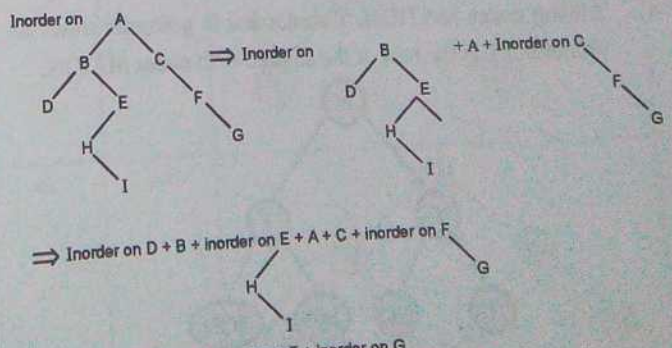
Stepwise preorder traversal on the above tree is given below.



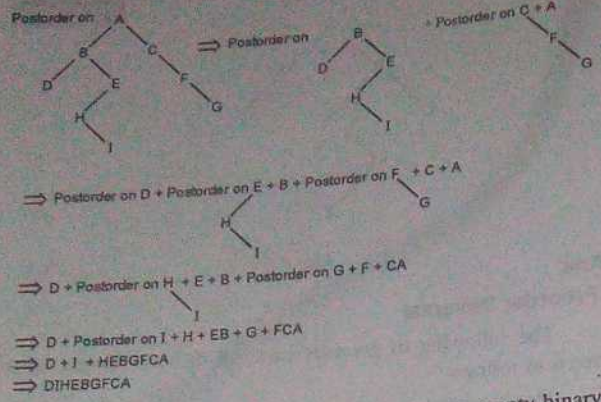
Inorder traversal

The functioning of inorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in inorder.
2. Next, visit the root node.
3. At last, traverse the right subtree in inorder.



Postorder traversal



The functioning of postorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in postorder.
2. Next, traverse the right subtree in postorder.
3. At last, visit the root node.

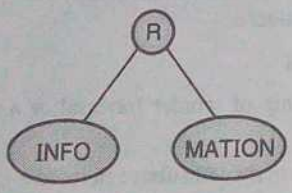
Q. 10 Construct the binary tree for the inorder and post order traversal sequence given below.
In order : "INFORMATION"
Post order : "INOFMAINOTR" May 2016

Ans. :

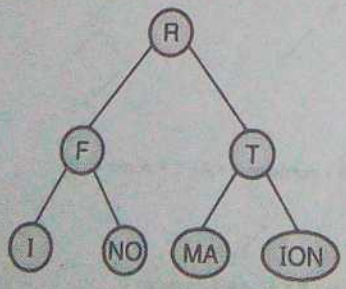
Inorder : INFORMATION

Postorder : INOFMAINOTR

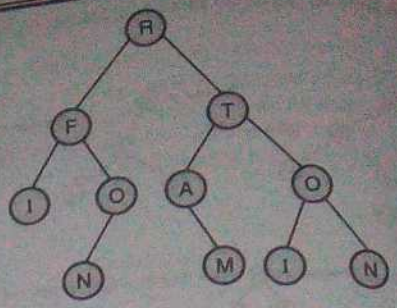
1. In postorder traversal, root is the last node. Hence R is the root of the binary tree. (R — root)
2. From inorder traversal, we can find left and right descendants of the root.



3. Among nodes INFO, F comes last in postorder traversal. Therefore, F is the root of the subtree with nodes INO.
4. Among nodes MATION, T comes last in postorder traversal. Therefore, T is the root of the subtree with nodes MAION.



5. Among nodes NO, O comes last in postorder. Among nodes MA, A comes last in postorder among nodes ION, O comes last in postorder. Therefore O is the root of subtree NO, A is the root of subtree MA and O is the root of subtree ION.

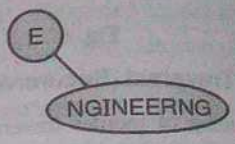


Q. 11 Construct the binary tree for the, in older and pre-order traversal sequence given below :
In order : ENGINEERING
Preorder : EGNENIIRENG May 2017

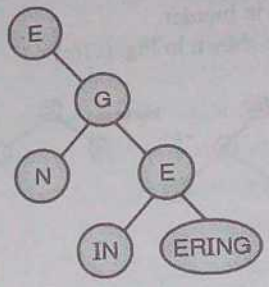
Ans. :

Step 1 :

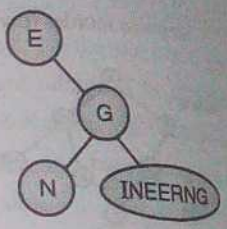
Root is the first element in the preorder sequence



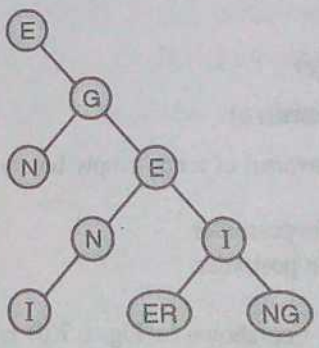
Step 2 :



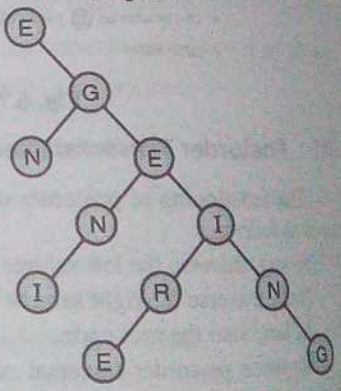
Step 3 :



Step 4 :



Step 5 :



Q. 12 Define and explain binary search tree.
Dec. 2013, Dec. 2014, May 2015, Dec. 2015, May 2016, Dec. 2016

Ans. :

Definition of binary search tree

A binary search tree is a binary tree, which is either empty or in which each node contains a key that satisfies the following conditions :

(1) All keys are distinct.

(2) For every node, X, in the tree, the values of all the keys in its left subtree are smaller than the key value in X.

(3) For every node, X, in the tree, the values of all the keys in its right subtree are larger than the key value in X.

Binary search tree finds its application in searching. In Fig. 6.9, the tree on the left is a binary search tree. Tree on the right is not a BST. Left subtree of the node with key 8 has a key with value 9.

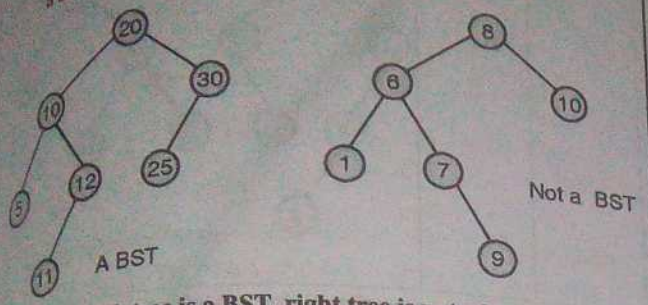
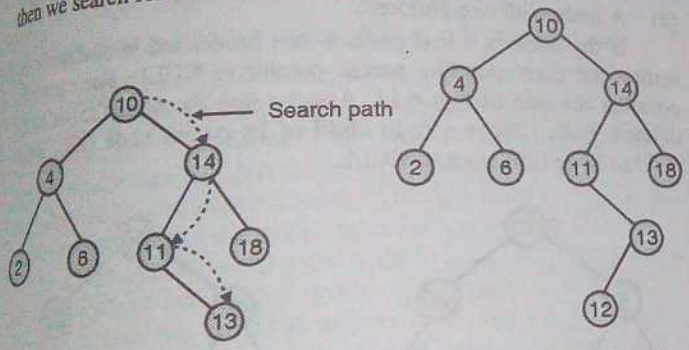


Fig. 6.9 : Left tree is a BST, right tree is not a binary search tree

Q. 13 Write algorithm to implement insertion operation Dec. 2013, May 2015, Dec. 2015, May 2016

Ans. : Insert operation

The function insert (T, x), adds element x to an existing binary search tree. T is tested for NULL, if so, a new node is created to hold x. T is made to point to the new node. If the tree is not empty then we search for x as in find() operation.



(a) Before insertion of the new key 12 (b) After inserting 12

Fig. 6.10 Insertion operation into a binary search tree

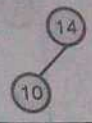
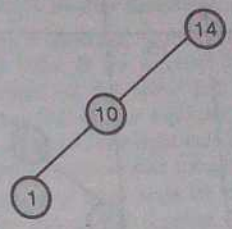
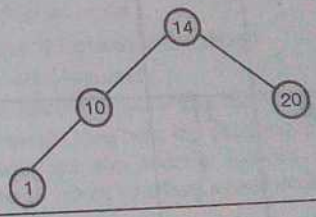
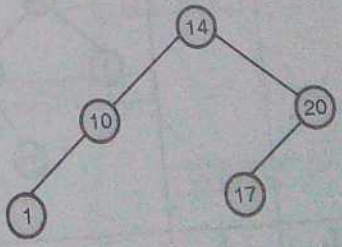
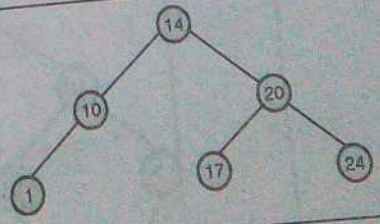
If x is already there in the then insert() operation terminates without insertion as a BST is not allowed to have duplicate keys. If we find a NULL pointer during the find() operation. We replace it by a pointer to a new node holding x. Fig. 6.10 shows the insert operation.

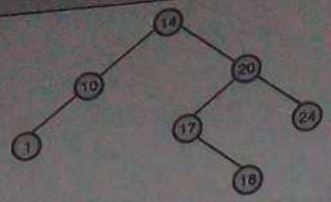
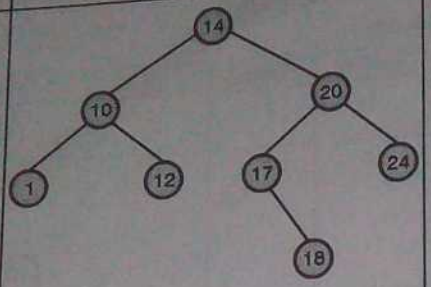
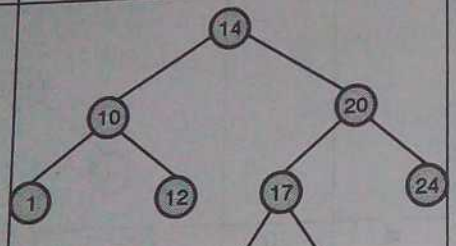
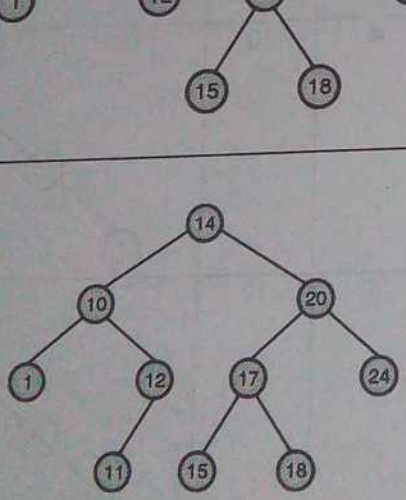
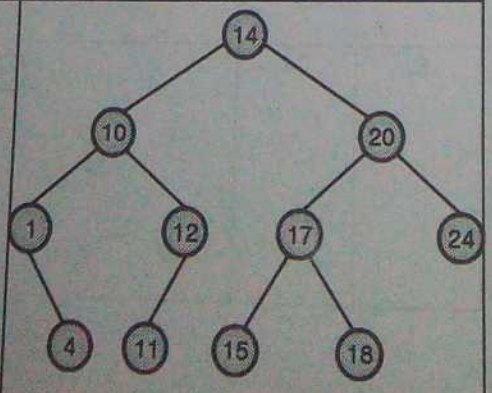
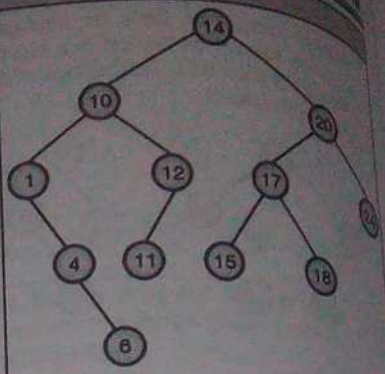
'C' Function for insert() – Recursive

```
BSTnode *insert(BSTnode *T,int x)
{
  if(T==NULL)
  {
        T=(BST *)malloc(sizeof(BSTnode));
        T->data=x;
        T->left=NULL;
        T->right=NULL;
        return(T);
  }
                                    // insert in right subtree
  {
        T->right=insert(T->right,x);
        return(T);
  }
  T->left=insert(T->left,x);   //insert in left subtree
  return(T);
}
```

Q. 14 Construct the Binary Search Tree for the following set of data : 14, 10, 1, 20, 17, 24, 18, 12, 15, 11, 4, 6 Dec. 2016

Ans. :

Sr. No.	Data	Tree
1.	14	
2.	10	
3.	1	
4.	20	
5.	17	
6.	24	

Sr. No.	Data	Tree
7.	18	
8.	12	
9.	15	
10.	11	
11.	4	
12.	6	

Q. 15 Write the algorithm for deletion of a node, in Binary Search Tree. Explain all the three cases of traversals.

Dec. 2013, May 2015, Dec. 2015, May 2016, May 2017

Ans. : Delete operation

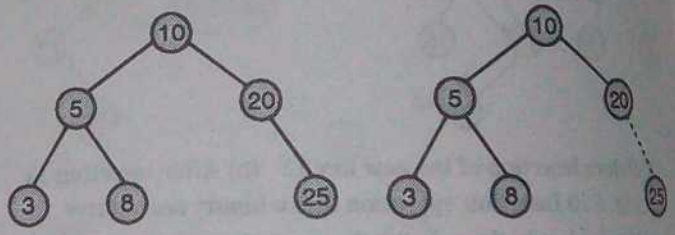
In order to delete a node, we must find the node to be deleted. The node to be deleted may be :

(a) A leaf node

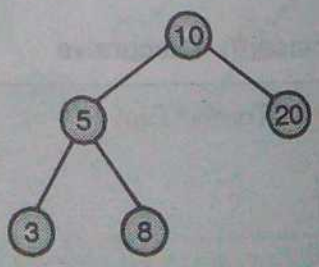
(b) A node with one child

(c) A node with two children.

If the node is a leaf node, it can be deleted immediately by setting the corresponding parent pointer to NULL. For example, consider the tree of Fig. 6.11. Assume that the node (25) is to be deleted. node (25) is a right child of its parent node (20). Right child of node (20) is set to NULL.



(a) Before deletion (b) Deletion

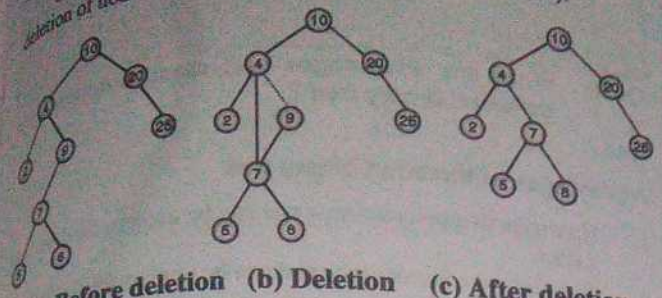


(c) After deletion

Fig. 6.11 : Deletion of node (25), a leaf node

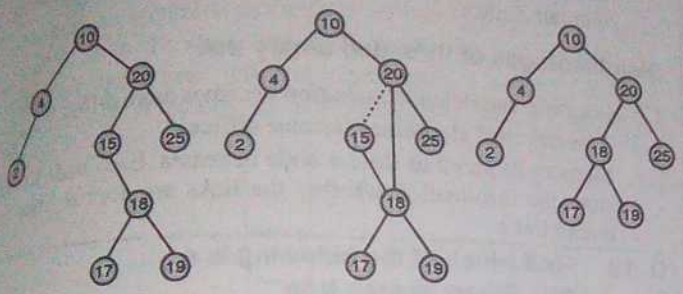
Even when the node to be deleted has one child, it can be deleted easily. If a node q is to be deleted and it is right child of its parent node p. The only child of q will become the right child of p after deletion of q. Similarly, if a node q is to be deleted and it is left child of its parent node p. The only child of q will become the left child of p after deletion.

For example, consider the tree of Fig. 6.12. Assume that node (9) is to be deleted. Since node (9) is a right child of its parent node (4). The only child subtree of node (9), with node (7) will become the right subtree of node (4) after deletion. Fig. 6.13 shows deletion of node (15). It is left child of its parent node (20).



(a) Before deletion (b) Deletion (c) After deletion

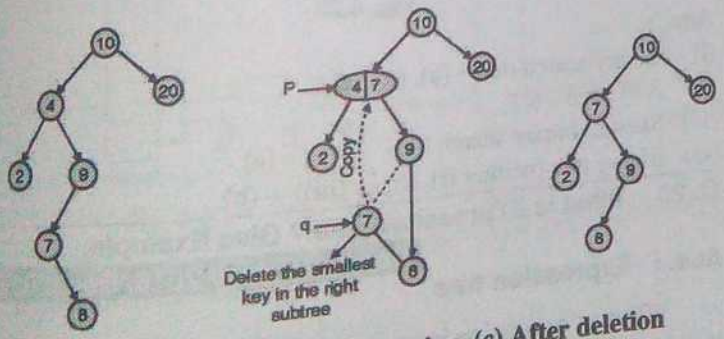
Fig. 6.12 : Deletion of a node (9) with one child



(a) Before deletion (b) Deletion (c) After deletion

Fig. 6.13 : Deletion of a node (15) with one child

The case in which the node to be deleted has two children is a bit complicated. The general strategy is to replace the data of this node with the smallest data of the right subtree (inorder successor) and then delete the smallest node in the right subtree. The smallest node in right subtree will either be a leaf node or a node of degree 1. As a first step, the node with smallest value in the right subtree of P (address of node (4)) is found and its address is stored in q. Content of node q is copied in node P. As a second step, the node q is deleted, a node with one child.. For example, consider the tree of Fig. 6.14. Assume that node (4) is to be deleted. Node (7) is the smallest node in the right subtree of node (4). Value 7 is copied in the node P (earlier node (4)) as shown in the Fig. 6.14(b). Now, the node q is deleted.



(a) Before deletion (b) Deletion (c) After deletion

Fig. 6.14 : Deletion of a node (4) with two children

Q. 16 Explain the concept of threaded binary search tree. Show the declaration of a node in threaded binary search tree . May 2014, Dec. 2015

Ans. : Threaded Binary Trees (TBT)

In a linked representation of a binary tree, there are more null links than actual pointers. These null links can be replaced by pointers, called threads to other nodes. A left null link of a node is replaced with the address of its inorder predecessor. Similarly, a right null link of a node is replaced with the address of its inorder successor.

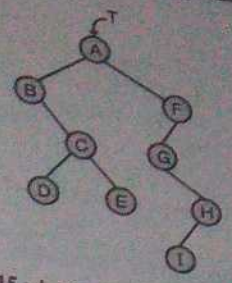


Fig. 6.15 : A sample tree before threading

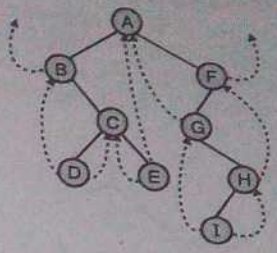


Fig. 6.16 : Tree of Fig. 6.16 after threading

The tree T of Fig. 6.16 has 9 nodes and 10 null links, which have been replaced by thread links. If we traverse the tree in inorder the nodes will be visited in the order BDCEAGIHF. Consider a node D. The left null link of D is replaced with a the link pointing to its inorder predecessor. The right null link of D is replaced with a thread link pointing to its inorder successor.

In the memory representation of a tree node we must be able to distinguish between threads and normal pointers. This can be done by adding two extra fields *l*bit and rbit.

*l*bit of a node = 1 left child is normal

*l*bit of a node = 0 left link is replaced with a thread

rbit of a node = 1 right child is normal

rbit of a node = 0 right link is replaced with a thread

In the Fig. 6.17, two threads have been left dangling. Node B has no inorder predecessor and the node F has no inorder successor. This problem can be solved by taking a head node.

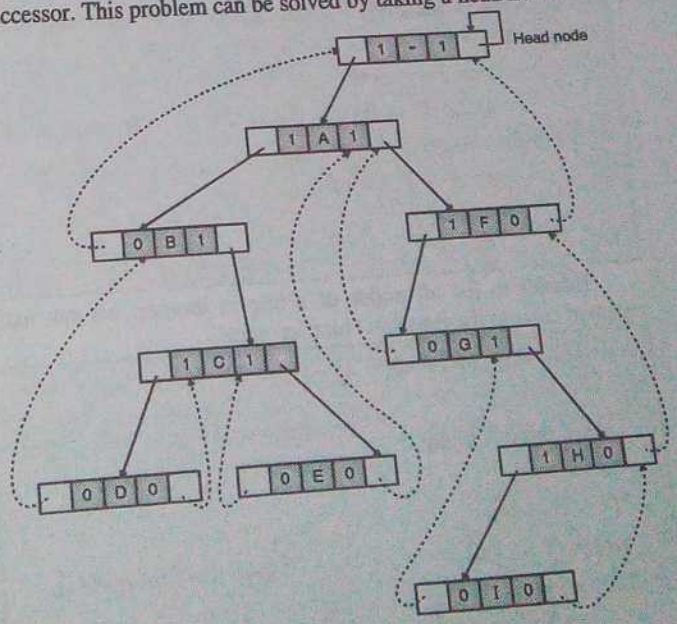


Fig. 6.17 : Initial status of head node

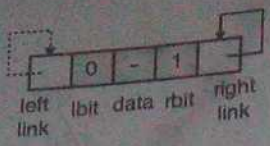


Fig. 6.18 : Memory representation of the TBT of Fig. 6.17

Q. 17 Write a function for inorder traversal of threaded binary search tree. Dec. 2015

Ans. : In order Traversal of a TBT

In a TBT, it is easy to find the inorder successor of any node without using a stack. If the right child of a node is not a thread then the leftmost child in the right subtree will be the inorder successor. If the right link is a thread, then the thread itself points to the successor.

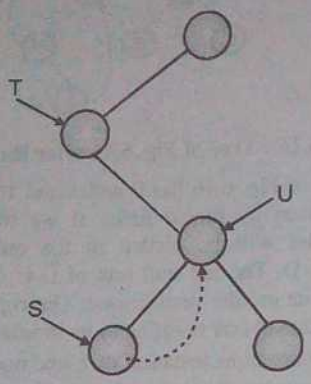


Fig. 6.19 : node S is inorder successor of T node U is inorder successor of S

Function "inorder-succ" returns the address of the inorder successor of any arbitrary node.

```
TBTnode * inorder_succ(TBTnode * P)
{
        if (P → rbit = = 0)
            return (P → right);          /* right link is a
thread link */
        P = P → right;
        while (P → lbit = =1 )           /* leftmost child
of the right subtree */
            P = P → left;
        return (P);
}
```

Inorder to list all nodes of a tree in inorder, we can make repeated calls to the function "inorder_succ".

```
void Tinorder (TBTnode * T)
{
   TBT node * head;
   head = T;
   T = T → left;
   while (T → lbit = = 1)        /* goto the first node in
inorder sequence */
     T = T → left;
  while (T! = head)
  {
        printf ("\n%c", T → data);
        T = inorder_succ (T);
  }
}
```

Q. 18 What are advantages and disadvantages of threaded binary tree ?

Ans. :

Advantages of threaded binary tree

1. Non-recursive preorder traversal can be implemented without a stack.
2. Non-recursive inorder traversal can be implemented without a stack.
3. Non-recursive postorder traversal can be implemented without a stack.

Disadvantages of threaded binary tree

1. Insertion and deletion operation becomes more difficult.
2. Tree traversal algorithms become difficult.
3. Memory required to store a node increases. Each node has to store the information whether the links are normal links or thread links.

Q. 19 Find which of the following is a :
(i) Binary search tree
(ii) AVL tree
(iii) Skewed binary search tree
(iv) Binary tree (neither(i), (ii) and (iii))

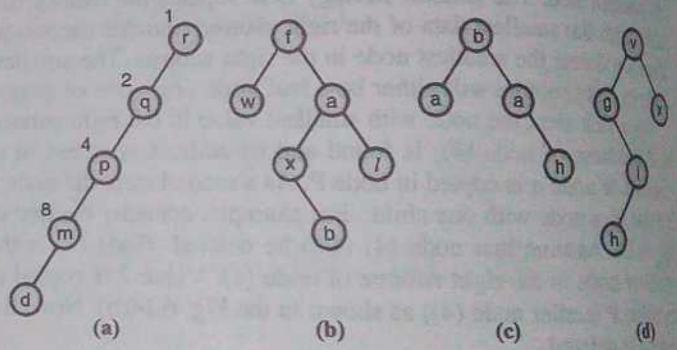


Fig. 6.20

Ans. :

(i) Binary search tree – (a), (c), (d)
(ii) AVL tree – (c)
(iii) Skewed binary search tree – (a)
(iv) Binary tree (neither (i), (ii) or (iii)) – (b)

Q. 20 What is Expression Tree ? Give Example. Dec. 2013, May 2016, Dec. 2016

Ans. : Expression tree

When an expression is represented through a tree, it is known as an expression tree. The leaves of an expression tree are operands, such as constants or variables names and all internal nodes contain operations. Fig. 6.21 gives an example of an expression tree.

(a + b * c) * e + f

A preorder traversal on the expression tree gives prefix equivalent of the expression. A postorder traversal on the expression tree gives postfix equivalent of the expression.

Prefix (expression tree of Fig. 6.21) = + * * a * b c e f

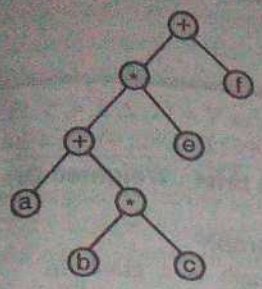


Fig. 6.21

Postfix (expression tree of Fig. 6.21) = a b c * + e * f +

Algorithm

We read our expression one symbol at a time. If the symbol is an operand, we create one node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop pointer to two trees T_2 and T_1 from the stack and form a new tree whose root is the operator and whose left and right children point to T_1 and T_2 respectively. A pointer to this new tree is then pushed onto the stack.

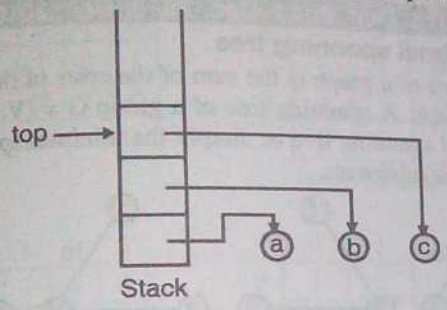


Fig. 6.22

As an example, suppose the input is abc * + e * f +

The first three symbols are operands, so we create one-node trees push pointers to them onto a stack. Next, 4 * is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

Next, a + is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

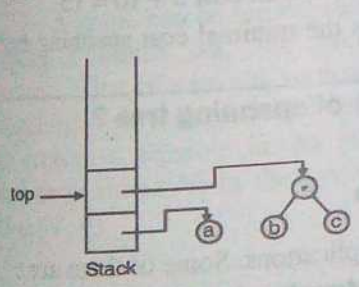


Fig. 6.23

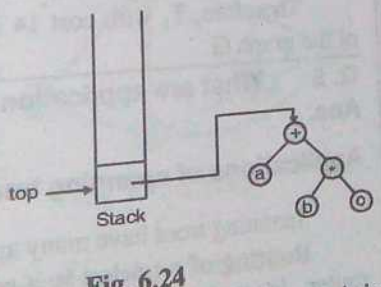


Fig. 6.24

Next, e is read, one node tree is created and a pointer to it is pushed onto the stack.

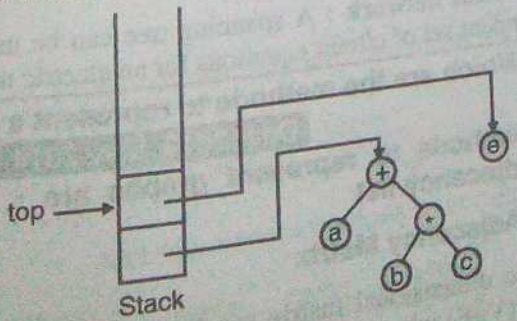


Fig. 6.25

Next, a * is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

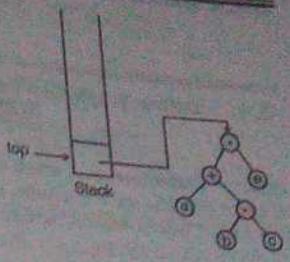


Fig. 6.26

Continuing, f is read, a one node tree is created and a pointer to it is pushed onto the stack.

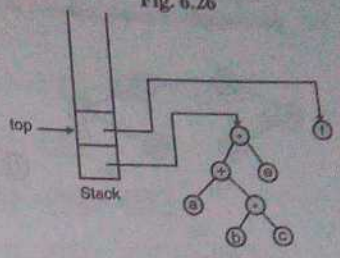


Fig. 6.27

Finally, a + is read, two trees are merged, and a pointer to the final tree is pushed on the stack.

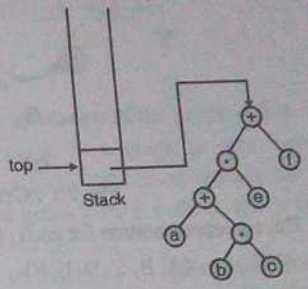
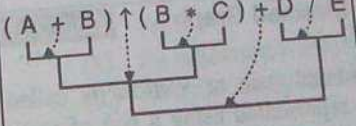


Fig. 6.28

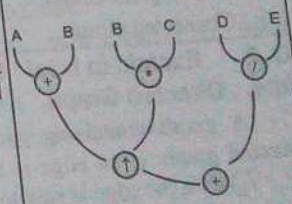
Q. 21 Construct the binary tree for the following expression : (A + B) ↑ (B * C) + D / E

Ans. :

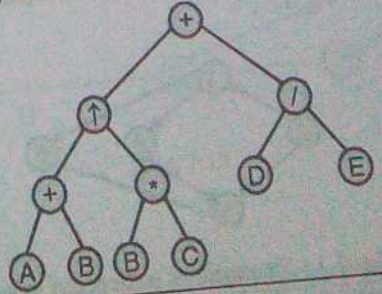
Step 1 : Group elements as per the sequence of evaluation. [This step is similar to fully parenthesizing an expression].



Step 2 : Move the operator at the center of the group.



Step 3 : Inverting the structure, we get



Q. 1 Define Graph. List its types with example. May 2014, Dec. 2014, May 2015, Dec. 2015, May 2017

Ans. : Definition : A graph G is a set of vertices (V) and set of edges (E). The set V is a finite, nonempty set of vertices. The set E is a set of pair of vertices representing edges.

G = (V, E); V(G) = Vertices of graph G

E(G) = Edges of graph G

An example of graph is shown in Fig. 7.1

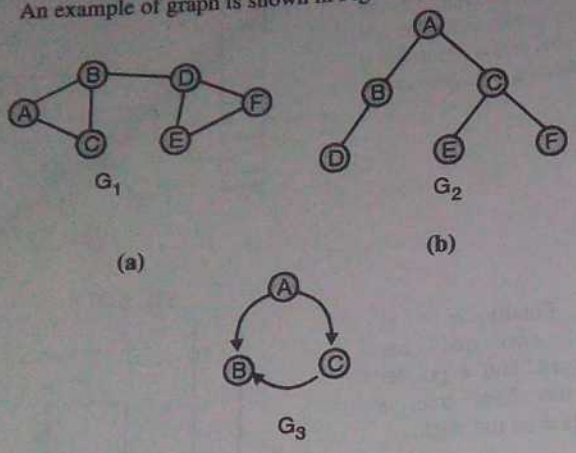


Fig. 7.1 : Graphs

The set representation for each of these graphs is given by

$V(G_1)$ = {A, B, C, D, E, F}

$V(G_2)$ = {A, B, C, D, E, F}

$V(G_3)$ = {A, B, C}

$E(G_1)$ = {(A, B),(A, C),(B, C),(B, D),(D, E),(D, F),(E, F)}

$E(G_2)$ = {(A,B),(A, C),(B, D),(C, E),(C, F)}

$E(G_3)$ = {(A, B),(A, C),(C, B)}

Types of graph

(i) Undirected Graph (ii) Directed Graph

(iii) A Complete Graph (iv) Weighted Graph

(v) Connected Graph

Q. 2 Explain in brief : Directed Graph Dec. 2013

Ans. : Directed Graph

A graph containing ordered pair of vertices is called a directed graph. If an edge is represented using a pair of vertices (V_1, V_2) then the edge is said to be directed from V_1 to V_2.

The first element of the pair, V_1 is called the start vertex and the second element of the pair, V_2 is called the end vertex. In a directed graph, the pairs (V_1, V_2) and (V_2, V_1) represent two different edges of a graph. Example of a directed graph is shown in Fig. 7.2.

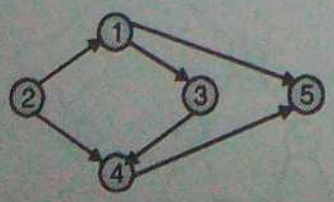


Fig. 7.2 : Example of a directed graph

The set of vertices V = {1, 2, 3, 4, 5, 5}. The set of edges E = {(1, 3), (1, 5), (2, 1), (2, 4), (3, 4),(4, 5)}.

Q. 3 Explain in brief : Weighted Graph Dec. 2013, 2 Marks

Ans. : Weighted Graph

A weighted graph is a graph in which edges are assigned some value. Most of the physical situations are shown using weighted graph. An edge may represent a highway link between two cities. The weight will denote the distance between two connected cities using highway. Weight of an edge is also called its cost. The graph of Fig. 7.3 is an example of a weighted graph.

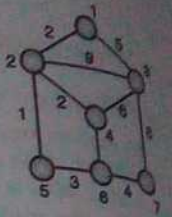


Fig. 7.3 : A weighted graph

Q. 4 Explain in brief : Minimal spanning tree

Dec. 2013, Dec. 2014, Dec. 2015, Dec. 2016, May 2017

Ans. : Minimal spanning tree

The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a group G = (V, E) is called a minimal cost spanning tree or simply the minimal spanning tree of G if its cost is minimum.

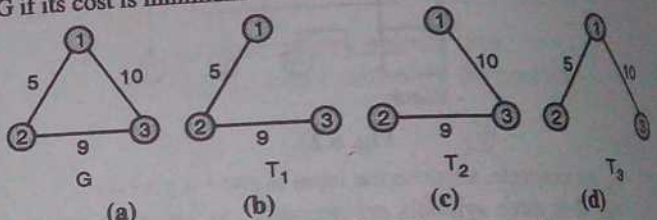


Fig. 7.4 : An example of minimal spanning tree

G → A sample weighted graph

T_1 → A spanning tree of G with cost 5 + 9 = 14

T_2 → A spanning tree of G with cost 7 + 9 = 19

T_3 → A spanning tree of G with cost 5 + 10 = 15

Therefore, T_1 with cost 14 is the minimal cost spanning tree of the graph G.

Q. 5 What are applications of spanning tree ?

Ans. :

Applications of spanning tree

Spanning trees have many applications. Some of them are :

Routing of a packet in a network : A node can represent a router, located in a city and the link between routers can be represented using an edge. A spanning tree can represent a network with the minimum number of links.

Electrical network : A spanning tree can be used to obtain an independent set of circuit equations for an electric network.

Q. 6 Which are the methods to represent a graph ? Dec. 2013, May 2014, Dec. 2014

Ans. : Methods to represent graphs are : Adjacency Matrix, adjacency list

(i) Adjacency Matrix

A two dimensional matrix can be used to store a graph. A graph G = (V, E) where V = {0, 1, 2, ... n – 1} can be represented using a two dimensional integer array of size n × n.

int adj[20][20]; can be used to store a graph with 20 vertices.
adj[i][j] = 1, indicates presence of edge between two vertices i and j
= 0, indicates absence of edge between two vertices i and j

A graph is represented using a square matrix.

Adjacency matrix of an undirected graph is always a symmetric matrix, i.e. an edge (i, j) implies the edge (j, i).

Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicates a directed edge from vertex i to vertex j.

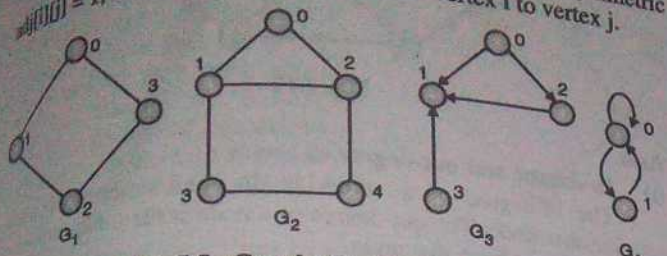


Fig. 7.5 : Graphs G_1, G_2, G_3 and G_4

	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0

G_1 (Undirected graph)

	0	1	2	3	4
0	0	1	1	0	0
1	1	0	1	1	0
2	1	1	0	0	1
3	0	1	0	0	1
4	0	0	1	1	0

G_2 (Undirected graph)

	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	1	0	0
3	0	1	0	0

G_3 (Directed graph)

	0	1
0	1	1
1	1	0

G_4 (With self loop)

Fig. 7.6 : Adjacency matrix representation of graphs G_1, G_2, G_3 and G_4 of Fig. 7.5

(ii) Adjacency List

A graph can be represented using a linked list. For each vertex, a list of adjacent vertices is maintained using a linked list. It creates a separate linked list for each vertex V_i in the graph G = (V, E)

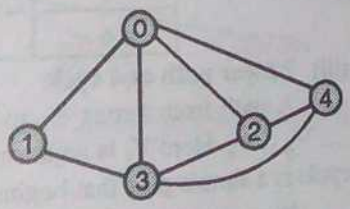
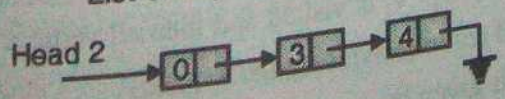


Fig. 7.7: A graph

List of adjacent vertices to vertex 0

List of adjacent vertices to vertex 1

List of adjacent vertices to vertex 2



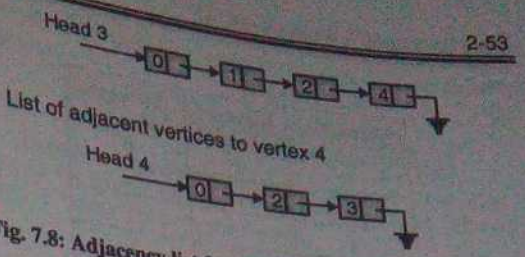


Fig. 7.8: Adjacency list for each vertex of graph of Fig. 7.7

Adjacency list of a graph with n nodes can be represented by an array of pointers. Each pointer points to a linked list of the corresponding vertex. Fig. 7.9 shows the adjacency list representation of graph of Fig. 7.7.

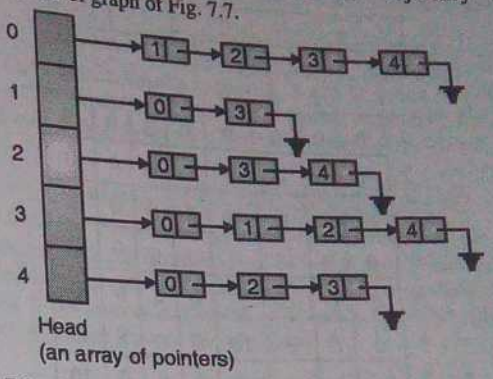


Fig. 7.9 : Adjacency list representation of the graph of Fig. 7.7

Adjacency list representation of a graph is very memory efficient when the graph has a large number of vertices but very few edges. For an undirected graph with n vertices and e edges, total number of nodes will be n + 2e.

If e is large then due to overhead of maintaining pointers, adjacency list representation does not remain cost effective over adjacency matrix representation of a graph. Degree of a node in an undirected graph is given by the length of the corresponding linked list. Finding in-degree of a directed graph represented using adjacency list will require O(e) comparisons. Lists pointed by all vertices must be examined to find the indegree of a node in a directed graph.

Checking the existence of an edge between two vertices i and j is also time consuming. Linked list of vertex i must be searched for the vertex j.

Q. 7 For the following graph obtain :

(i) The in degree and out degree of each vertex

(ii) Its adjacency matrix

(iii) Its adjacency list representation.

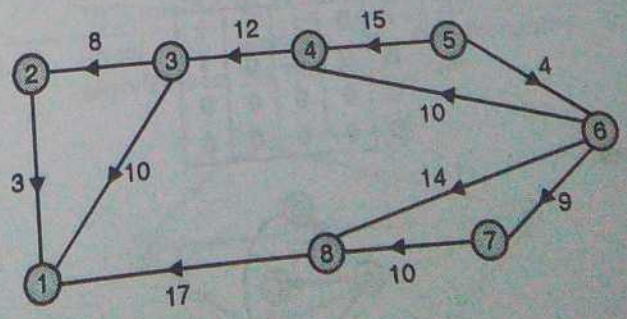


Fig. 7.10

Ans. :

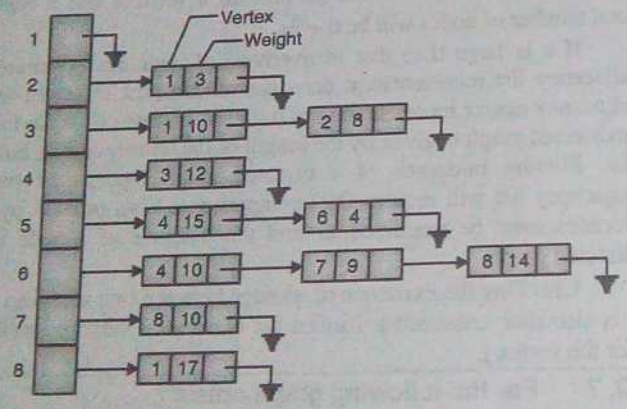
(1)

Vertex No.	Indegree	Outdegree
1	3	0
2	1	1
3	1	2
4	2	1
5	1	1
6	1	3
7	1	1
8	2	1

(2)

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	3	0	0	0	0	0	0	0
3	10	8	0	0	0	0	0	0
4	0	0	12	0	0	0	0	0
5	0	0	0	15	0	4	0	0
6	0	0	0	10	0	0	9	14
7	0	0	0	0	0	0	0	10
8	17	0	0	0	0	0	0	0

(3) **Adjacency list**



Q. 8 For the adjacency matrix given below, draw the corresponding graph.

	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	0	0	0
D	0	0	0	0

Ans. :

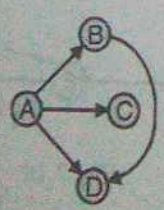


Fig. 7.11

Q. 9 Explain the following terms with example from the given graph.
(i) In-degree and out-degree of vertex.
(ii) Adjacent vertices
(iii) linear path and cycle.

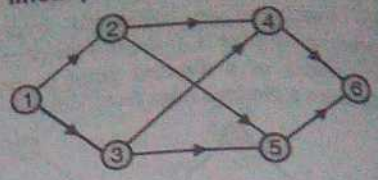


Fig. 7.12

Ans. :

(i) In-degree and out-degree of vertex

The in-degree of a vertex is the total number of edges entering that node. The out-degree of a node is the total number of edges going out from that node.

Node No.	In-degree	Out-degree
1	0	2
2	1	2
3	1	2
4	2	1
5	2	1
6	2	0

(ii) Adjacent vertices

A vertex V_2 is said to be adjacent to V_1, if there is an edge from V_1 to V_2.

Node No.	Adjacent vertices
1	2, 3
2	4, 5
3	4, 5
4	6
5	6
6	–

(iii) Linear path and cycle

A path from vertex V_1 to V_n is sequence of vertices V_1, V_2 .... V_{n-1}, V_n. Here V_1 is adjacent to V_2,... V_{n-1} is adjacent to V_n. A cycle is a simple path that begins and ends at the same vertex.

There are 4 paths between 1 and 6.

$1 \rightarrow 2 \rightarrow 4 \rightarrow 6$

$1 \rightarrow 2 \rightarrow 5 \rightarrow 6$

$1 \rightarrow 3 \rightarrow 5 \rightarrow 6$

$1 \rightarrow 3 \rightarrow 4 \rightarrow 6$

The graph does not have a cycle.

Q.10 Explain DFS with algorithm.

May 2014, Dec. 2015, May 2016

Ans. :

Depth First Search (DFS)

It is like preorder traversal of tree. Traversal can start from any vertex, say V_i. V_i is visited and then all vertices adjacent to V_i are traversed recursively using DFS.

DFS (G, 1) is given by

(a) Visit (1)
(b) DFS (G, 2)
DFS (G, 3)
DFS (G, 4)
DFS (G, 5)
} all nodes adjacent to 1

Since, a graph can have cycles. We must avoid re-visiting a node. To do this, when we visit a vertex V, we mark it visited. A node that has already been marked as visited, should not be selected for traversal. Marking of visited vertices can be done with the help of a global array visited[]. Array visited[] is initialized to false (0).

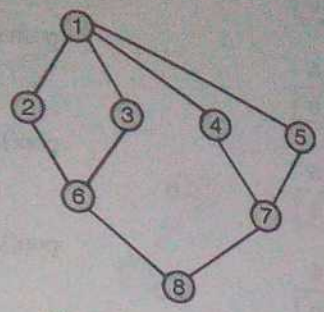


Fig. 7.13: Graph G

Q. 11 Write algorithm to traverse a graph using Depth First Search.

May 2014, Dec. 2014, May 2015, Dec. 2015

Ans. :

Algorithm for DFS

```
n ← number of nodes
(i) Initialize visited[] to false (0)
    for (i = 0; i < n; i++)
    visited[i] = 0;
(ii) void DFS (vertex i)  [DFS starting from i]
    {
    visited[i] = 1;
    for each w adjacent to i
        if(! visited[w])
            DFS(w);
    }
```

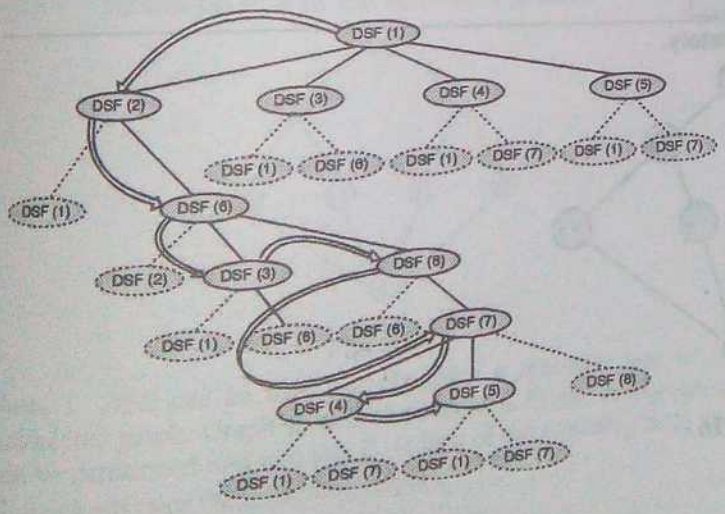


Fig. 7.14 : DFS traversal on graph

DFS traversal on graph of Fig. 7.14.

DSF(i)

Node i can be used for recursive traversal using DFS().

DFS(i)

Node i is already visited

Traversal start from vertex 1. Vertices 2, 3, 4 and 5 are adjacent to vertex 1. Vertex 1 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	0	0	0	0	0	0	0

Out of the adjacent vertices 2, 3, 4 and 5, vertex number 2 is selected for further traversal. Vertices 1 and 6 are adjacent to vertex 2. Vertex 2 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	0	0	0	0	0	0

Out of the adjacent vertices 1 and 6, vertex 1 has already been visited. Vertex number 6 is selected for further traversal. Vertices 2, 3 and 8 are adjacent to vertex 6. vertex 6 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	0	0	0	1	0	0

Out of the adjacent vertices 2, 3 and 8, vertex 2 is already visited. Vertex number 3 is used for further expansion. Vertices 1 and 6 are adjacent to vertex 3. Vertex 3 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	0	0	1	0	0

Vertices 1 and 6 are already visited, therefore it goes back to vertex 6. (Vertex 6 is predecessor of vertex 3 in DFS() sequence). Out of the adjacent vertices 2, 3 and 8, 2 and 3 are visited. It selects vertex 8 for further expansion. Vertices 6 an 7 are adjacent to vertex 8. Vertex 8 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	0	0	1	0	1

Out of the adjacent vertices 6 and 7, vertex 6 is already visited. Vertex number 7 is used for further expansion. Vertices 4, 5 and 8 are adjacent to vertex number 7. Vertex 7 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	0	0	1	1	1

Out of the adjacent vertices 4, 5 and 8, vertex 4 is selected for further expansion. Vertices 1 and 7 are adjacent to vertex 4. Vertex 4 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	1	0	1	1	1

Adjacent vertices 1 and 7 are already visited. It goes back to vertex 7 and selects the next unvisited node 5 for further expansion. Vertex 5 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	1	1	1	1	1

DFS, traversal sequence → 1, 2, 6, 3, 8, 7, 4, 5

Q. 12 Show the working of non-recursive DFS algorithm on the following graph.

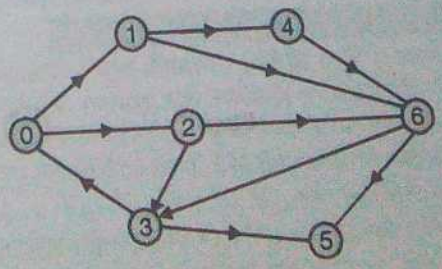


Fig. 7.15

Ans. :

Stack contents	Visited[ ] (0 1 2 3 4 5 6)	Vertex visited	Action
NULL	0 0 0 0 0 0 0	–	initial
↓ top: 0	0 0 0 0 0 0 0	–	push the initial vertex
↓ top: 1 2	1 0 0 0 0 0 0	0	pop(), visit(), push adjacent vertices
↓ top: 1 3 6	1 0 1 0 0 0 0	0, 2	pop(), visit(), push adjacent vertices
↓ top: 1 3 3 5	1 0 1 0 0 0 1	0, 2, 6	pop(), visit(), push adjacent vertices
↓ top: 1 3 3	1 0 1 0 0 1 1	0, 2, 6, 5	pop(), visit()
↓ top: 1 3	1 0 1 1 0 1 1	0, 2, 6, 5, 3	pop()
↓ top: 1	1 0 1 1 0 1 1	0, 2, 6, 5, 3	pop(), visit(), push adjacent vertices
↓ 4	1 1 1 1 0 1 1	0, 2, 6, 5, 3, 1	visit
NULL	1 1 1 1 1 1 1	0, 2, 6, 5, 3, 1, 4	complete

Q. 13: Find DFS for the given graph show each pass separately.

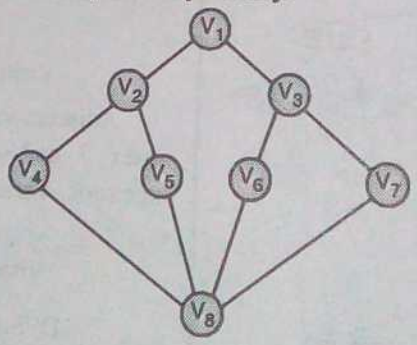
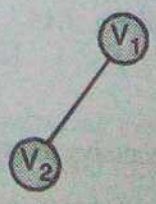


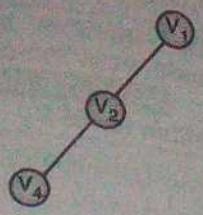
Fig. 7.16

Ans. :

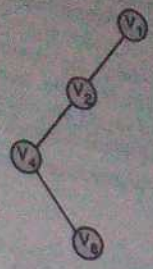
Step 1 : 

Step 2 : 

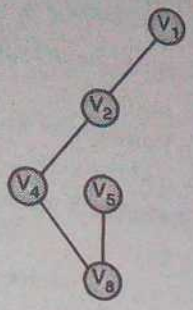
Step 3 :



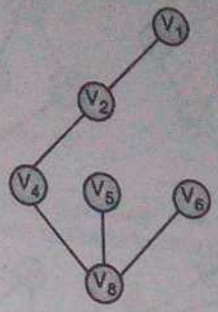
Step 4 :



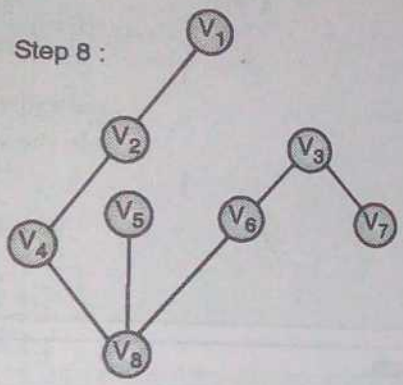
Step 5 :



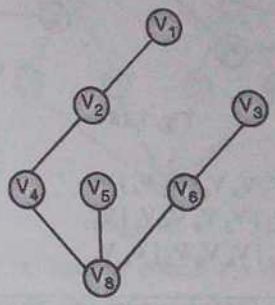
Step 6 :



Step 8 :



Step 7 :



Q. 14 What is DFS spanning tree ? Draw the DFS spanning tree of the following graph.

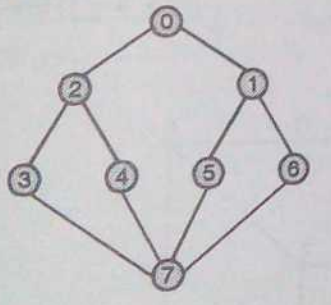


Fig. 7.17

Ans. : DFS can be used for obtaining a spanning tree of an undirected graph. Graph should be connected. Edges of a graph G can be partitioned into two sets E_1 (edges of the spanning tree) and E_2 (back edges or the remaining edges).

E_1 ← Null set (initially)

E_2 ← A set of all edges of the graph G.

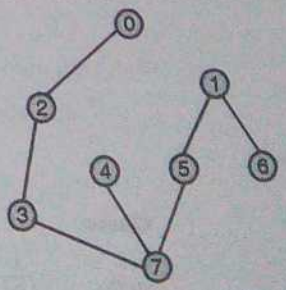
In the algorithm for DFS(), whenever a recursive call is made inside the if statement, a new edge will be added to E1.

```
"for each w adjacent to i
if(! visited[w])
DFS(w);"
```

Edge(i, w) is added to E1. The same edge is also deleted from E2.

$$E1 = E1 \cup \{(i, w)\}$$
$$E2 = E2 - \{j, w\}$$

Edges of E_1 will form a spanning tree as DFS will visit each node of the graph and each node will be visited exactly once.



DFS spanning tree of the graph

Q. 15 Explain BFS.

May 2014, Dec. 2015, May 2016, May 2017

Ans. : Breadth First Search (BFS) : It is another popular approach used for visiting the vertices of a graph. This method starts from a given vertex V_0. V_0 is marked as visited. All vertices adjacent to V_0 are visited next. Let the vertices adjacent to V_0 are V_{10}, V_3, V_{12} ... V_{1n}. V_{11}, V_{12} ... V_{1n} are marked as visited. All unvisited vertices adjacent to V_{11}, V_{12} .....V_{1n} are visited next. The method continues until all vertices are visited. The algorithm for BFS has to maintain a list of vertices which have been visited but

not explored for adjacent vertices. The vertices which have been visited but not explored for adjacent vertices can be stored in queue.

Initially the queue contains the starting vertex. In every iteration, a vertex is removed from the queue and its adjacent vertices which are not visited as yet are added to the queue. The algorithm terminates when the queue becomes empty.

Fig. 7.18 gives the BFS sequence on various graphs.

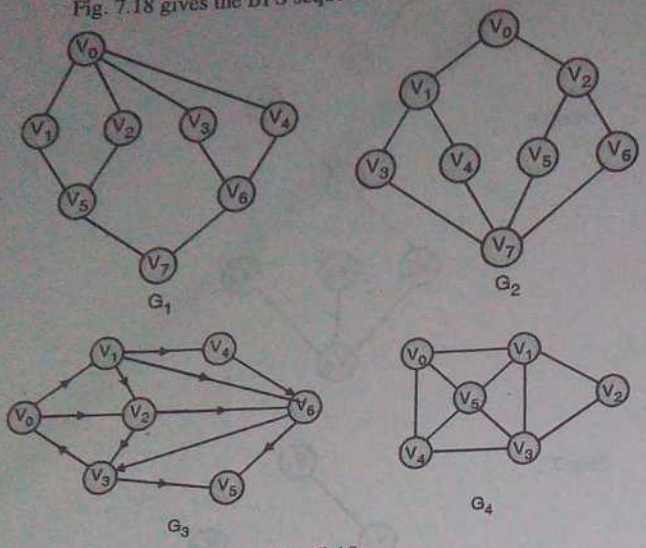


Fig. 7.18

BFS sequence :

$G_1 \rightarrow V_0 \mid V_1\ V_2\ V_3\ V_4 \mid V_5\ V_6 \mid V_7$

$G_2 \rightarrow V_0 \mid V_1\ V_2 \mid V_3\ V_4\ V_5\ V_6 \mid V_7$

$G_3 \rightarrow V_0 \mid V_1\ V_2 \mid V_4\ V_6\ V_3 \mid V_5\ V_6$

$G_4 \rightarrow V_0 \mid V_1\ V_4\ V_5 \mid V_2\ V_3$

Fig. 7.18(a) : BFS traversal on G_1, G_2, G_3 and G_4

Q. 16 Write algorithm to traverse a graph using Breadth First Search

May 2014, May 2015, Dec. 2015, Dec. 2016

Ans. : Algorithm for BFS

```
/* Array visited[] is initialize to 0 */
/* BFS traversal on the graph G is carried out beginning at
vertex V */
void BFS(int V)
{
        q : a queue type variable;
        initialize q;
        visited[v] = 1;      /* mark v as visited */
        add the vertex V to queue q;
        while(q is not empty)
        {
            v ← delete an element from the queue;
            for all vertices w adjacent from V
            {
                if(!visited[w])
                {
                    visited[w] = 1;
                    add the vertex w to queue q;
                }
            }
        }
}
```

Q. 17 Show the working of BFS algorithm on the following graph.

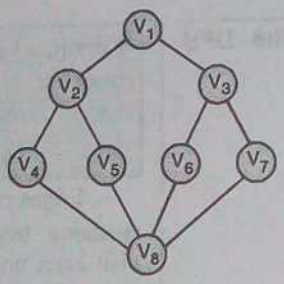


Fig. 7.19

Ans. :

Queue	Visited[ ] (1 2 3 4 5 6 7 8)	Vertex visited	Action
NULL	0 0 0 0 0 0 0 0	–	–
V_1	1 0 0 0 0 0 0 0	V_1	add (q, V_1) visit (V_1)
V_2 V_3	1 1 1 0 0 0 0 0	V_1 V_2 V_3	delete (q), add and visit adjacent vertices
V_3 V_4 V_5	1 1 1 1 1 0 0 0	V_1 V_2 V_3 V_4 V_5	delete (q), add and visit adjacent vertices

Queue	Visited[ ] 1 2 3 4 5 6 7 8	Vertex visited	Action
$V_1 V_5 V_6 V_7$	1 1 1 1 1 1 1 0	$V_1 V_2 V_3 V_4 V_5 V_6 V_7$	delete (q), add and visit adjacent vertices
$V_5 V_6 V_7 V_8$	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q), add and visit adjacent vertices
$V_6 V_7 V_8$	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q)
$V_7 V_8$	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q), add and visit adjacent vertices
V_8	1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q)
NULL	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	algorithm terminates

Q. 18 For the graph given below, generate adjacency list and perform BFS and DFS.

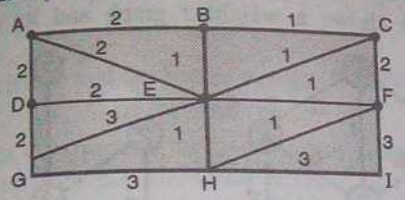
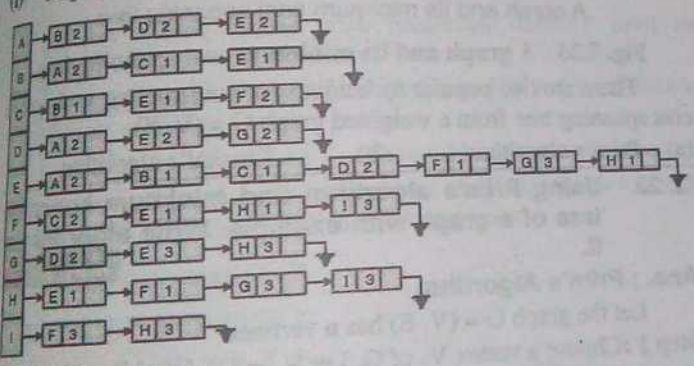


Fig. 7.20

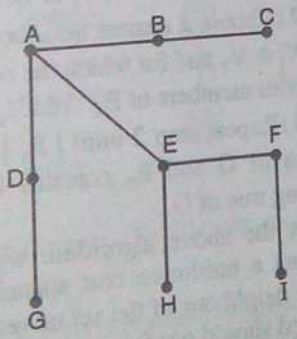
Ans. :

(i) Adjacency list :



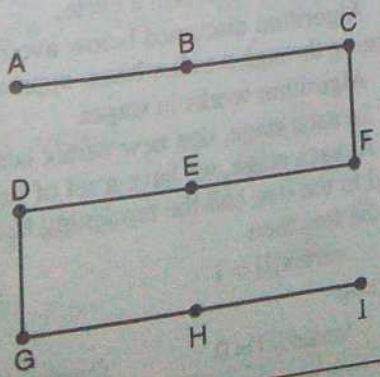
(ii) BFS sequence :

A B D E C G F H I



(iii) DFS sequence :

A B F E D G H I



Q. 19 What is BFS spanning tree. Draw the BFS spanning tree of the following graph.

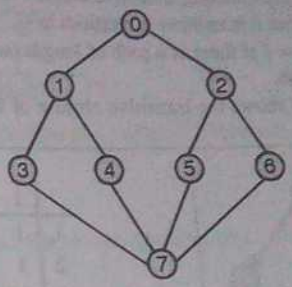


Fig. 7.21

Ans. : BFS can be used for obtaining a spanning tree of an undirected graph. Graph should be connected. A set E of edges of spanning tree can be created during BFS traversal of the graph.

In the algorithm for BFS traversal,

```
loop inside the algorithm for BFS
while(q is not empty)
{
        v ← delete an element from the queue;
        for all vertices w adjacent from V
            if(! visited[w])
            {
                visited[w] = 1;
                {
                    add the vertex w to queue q;
                }
            }
}
```

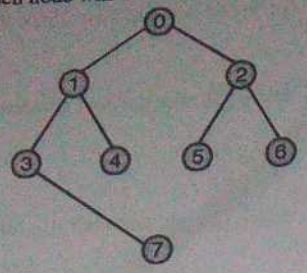
whenever a new vertex w is added to queue,

an edge (V, w) is added to E.

$E = E \cup \{(V, w)\}$

Edges of E will form a spanning tree as BFS will visit each node of the graph and each node will be visited only once.

BFS spanning tree



Q. 20 What is transitive closure ?

Ans. :

In many problems, it may be necessary to determine whether there exists a path from the vertex i to vertex j or simply whether the vertex i is connected to vertex j. Transitive closure of a graph G, represented using an adjacency matrix A[][] can defined as follows :

TC[][], representing transitive closure of G is a matrix of the size n × n where n is number of vertices in G.

TC[i][j] = 1 if there is a path of length one or more from i to j and 0 otherwise.

Fig. 7.23 shows the transitive closure of the directed graph of Fig. 7.22.

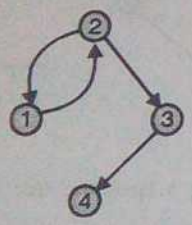


Fig. 7.22 : A digraph

	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	0	0	0	1
4	0	0	0	0

Fig. 7.23 : Transitive closure

Transitive closure of graph represented using an adjacency matrix A is given by

$A + A^2 + A^3 + \ldots A^n$, where n is number of vertices

Q. 21 Compute the transitive closure of the graph of Fig. 7.24 using matrix multiplication.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 7.24 : adjacency matrix of the graph

Ans. :

$$A^2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Transitive closure of $A = A + A^2 + A^3 + A^4$

$$= \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$+ \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Transitive closure of a graph can be used to check whether a graph is connected. A graph is said to connected (undirected graph) or strongly connected (directed graph) if there is no '0' in the transitive closure matrix of the graph. Graph of Fig. 7.24 is not strongly connected as there is no path between vertices

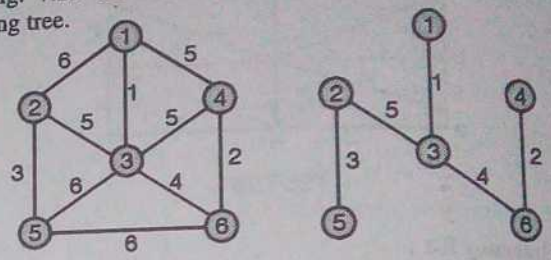
(3, 1), (3, 2), (3, 3), (4, 1), (4, 2), (4, 3) and (4, 4)

Q. 22 Define minimum cost spanning tree.

Ans. : Minimum cost spanning tree

With applications of weighted graphs, it is often necessary to find a spanning tree for which the total weight of the edges in the tree is as small as possible. Such a spanning tree is called a minimal spanning tree or minimum cost spanning tree.

Fig. 7.25 shows a weighted graph and its minimum-cost spanning tree.



A graph and its minimum cost spanning tree

Fig. 7.25 : A graph and its minimum cost spanning tree

There are two popular techniques for constructing a minimum cost spanning tree from a weighted graph G = (V, E).

(a) Prim's algorithm (b) Kruskal's algorithm

Q. 23 Using Prim's algorithm find minimum spanning tree of a graph with example. Write algorithm of it. May 2014

Ans. : Prim's Algorithm

Let the graph G = (V, E) has n vertices.

Step 1 : Choose a vertex V_1 of G. Let $V_T = \{V_1\}$ and $E_T = \{\}$.

Step 2 : Choose a nearest neighbour V_i of V_T that is adjacent to V_j, $V_j \in V_T$ and for which the edge (V_i, V_j) does not form a cycle with members of E_T. Add V_i to V_T and add (V_i, V_j) to E_T.

Step 3 : Repeat step 2 until $| E_T | = n - 1$. Then V_T contains all n vertices of G and E_T contains the edges of the minimum cost spanning tree of G.

In the above algorithm, we begin at any vertex of G and construct a minimum cost spanning tree by adding an edge to a nearest neighbour of the set of vertices already linked. The edge to be added should not form a cycle.

Algorithm discussed below avoids cycle in the minimum cost spanning through a rather better approach.

Algorithm works in stages.

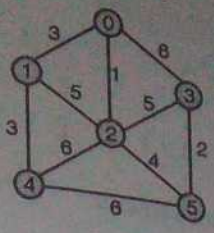
In each stage, one new vertex is added through an edge to the tree. At each stage, we have a set of vertices that have already been added to the tree and the remaining that have not been. If a vertex i is in the tree then

```
vertex[i] = 1
else
    vertex[i] = 0        //vertex i is not in the tree.
```

We find a new vertex V to be added to the tree, such that (u, V) is an edge in the graph u is in the tree and V is not (u, V) is the smallest weight edge among the remaining edge.

Two arrays distance[] and from[] are used to find the new vertex V to be added to the tree.

Initial values of visited[], distance[] and from[] are shown in the Fig. 7.26.



0	1	2	3	4	5	visited
1	0	0	0	0	0	

0	1	2	3	4	5	distance
0	3	1	6	∞	∞	

0	1	2	3	4	5	from
0	0	0	0	0	0	

◎ → Node added to the tree

Fig. 7.26 : Graph and the initial values of visited[], distance[], from[] and the tree

We start constructing the spanning tree beginning with the vertex 0. Array distance[] shows the distance of every vertex from the vertex 0. Array from[] says that the distances in the distance[] array are from the vertex 0.

1st iteration :

Select a new vertex with minimum distance from the spanning tree.

V = 2, from the distance[] array, distance[2] = 1 is minimum

u = 0, as from[2] = 0;

An edge (u, V) with weight 1 should be added to the tree.

At this stage, distance[] should be updated. Distance[], gives the minimum distance of vertices from the tree.

If(Distance(i) > cost[i][V])

Distance[i] = cost[i][V];

0	1	2	3	4	5	Visited
1	0	1	0	0	0	

0	1	2	3	4	5	Distance
–	3	–	5	6	4	

0	1	2	3	4	5	From
–	0	–	2	2	2	

Configuration of tables and the tree after 1st iteration.

2nd iteration

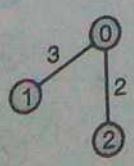
V = 1, u = 0[distance[1] = 3, is the minimum distance]

edge (1, 0) with weight 3 is added to the tree distance[] is updated.

0	1	2	3	4	5	Visited
1	1	1	0	0	0	

0	1	2	3	4	5	Distance
–	–	–	5	3	4	

0	1	2	3	4	5	From
–	–	–	2	1	2	



3rd iteration

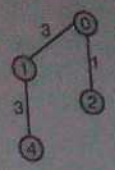
V = 4, u = 1[distance[4] = 3 is the minimum distance]

edge (4, 1) with weight 3 is added to the tree. Distance[] is updated.

0	1	2	3	4	5	Visited
1	1	1	0	1	0	

0	1	2	3	4	5	Distance
–	–	–	5	–	4	

0	1	2	3	4	5	From
–	–	–	2	–	2	



Configuration after 3rd iteration

4th iteration

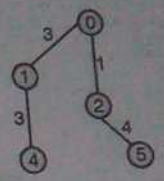
V = 5, u = 2[distance[5] = 4 is the minimum distance]

edge (5, 2) with weight 4 is added to the tree. Distance[] is updated.

0	1	2	3	4	5	Visited
1	1	1	0	1	1	

0	1	2	3	4	5	Distance
–	–	–	2	–	–	

0	1	2	3	4	5	From
–	–	–	5	–	–	



Configuration after 4th iteration

5th iteration

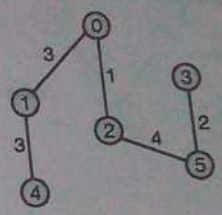
V = 3, u = 5

edge (3, 5) is added to the spanning tree.

0	1	2	3	4	5	Visited
1	1	1	1	1	1	

0	1	2	3	4	5	Distance
–	–	–	–	–	–	

0	1	2	3	4	5	From
–	–	–	–	–	–	



Configuration after 5th iteration

Q. 24 Find MST for the following graph using Prim's algorithm. Show various steps. Dec. 2013

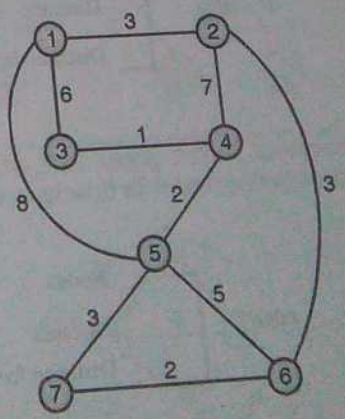


Fig. 7.27

Ans. : Step 1 :

edge

Nodes	1	2	3	4	5	6	7
Distance	-	3	6	∞	8	∞	∞
Distance from	-	1	1	1	1	1	1

↑

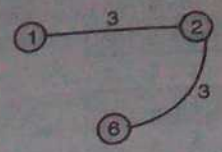
Node at minimum distance is selected.

Step 2 :

edge

Nodes	1	2	3	4	5	6	7
Distance	-	3	6	7	8	3	∞
Distance from	-	1	1	2	1	2	1

↑

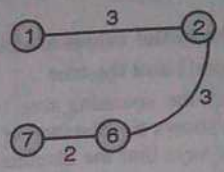


Step 3 :

edge

Nodes	1	2	3	4	5	6	7
Distance	-	3	6	7	5	3	2
Distance from	-	1	1	2	6	2	6

↑

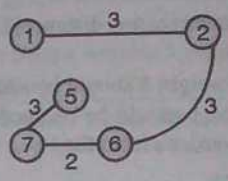


Step 4 :

edge

Nodes	1	2	3	4	5	6	7
Distance	-	3	6	7	3	3	2
Distance from	-	1	1	2	7	2	6

↑

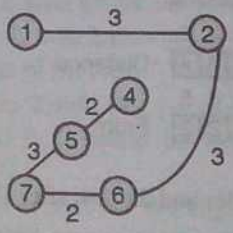


Step 5 :

edge

Nodes	1	2	3	4	5	6	7
Distance	-	3	6	2	3	3	2
Distance from	-	1	1	5	7	2	6

↑

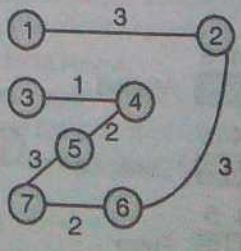


Step 6 :

edge

Nodes	1	2	3	4	5	6	7
Distance	-	3	1	2	3	3	2
Distance from	-	1	4	5	7	2	6

↑



Q. 25 Using Prim's algorithm find minimum spanning tree for the following graph. Dec. 2015

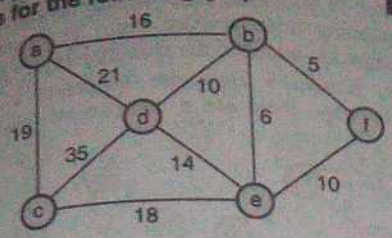
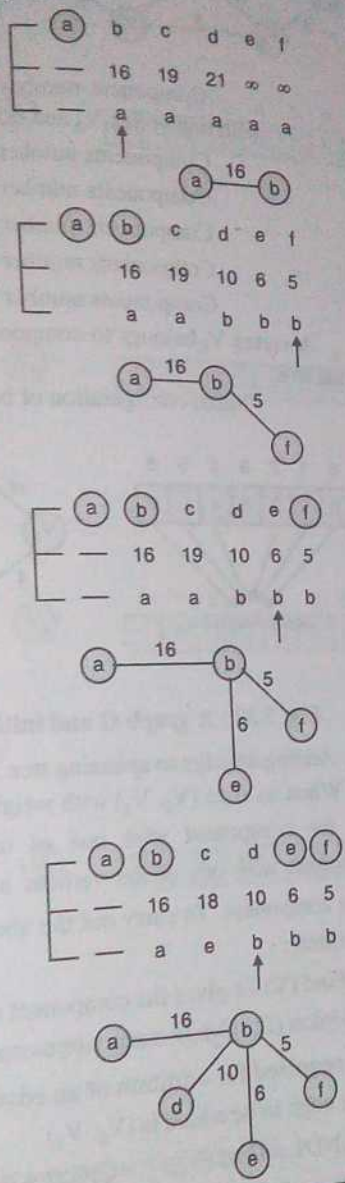


Fig. 7.28

Ans. :

Prim's Algorithm



Q. 26 Using Prim's algorithm find minimum spanning tree for the following graph : May 2016

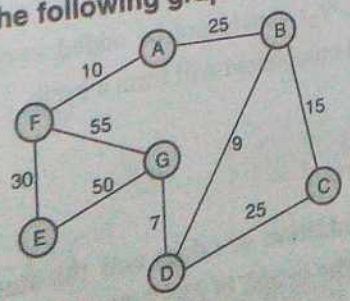
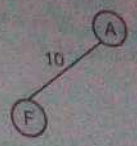


Fig. 7.29

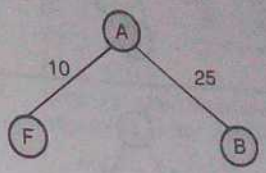
Ans. : Prim's algorithm :

Step 1 : Initial node A is added

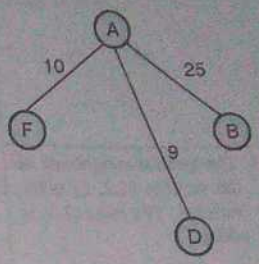
Step 2 : Nearest node F is added



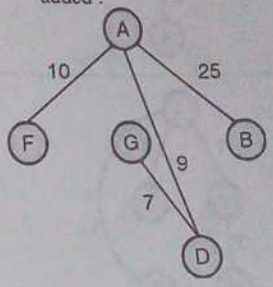
Step 3 : Nearst node B is added :



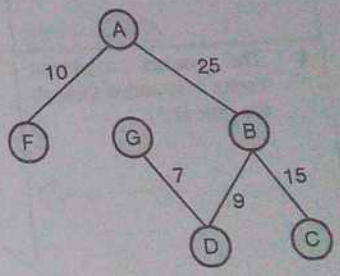
Step 4 : Nearest node D is added :



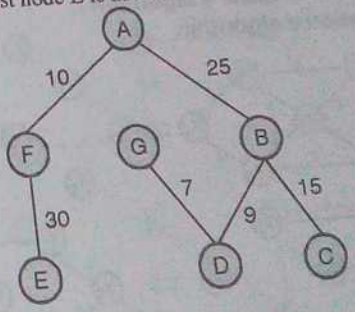
Step 5 : Nearest note G is added :



Step 6 : Nearest node C is added :



Step 7 : Nearest node E is added :



Q. 27 Draw the MST using Prim's Algorithm and find out the cost with all intermediate steps. May 2017

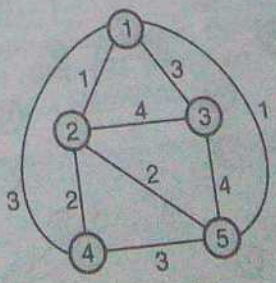
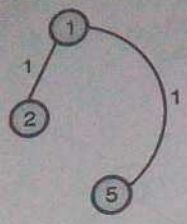
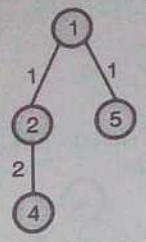
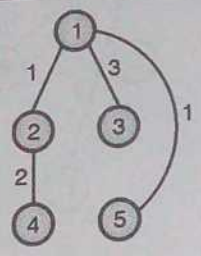


Fig. 7.30

Ans. :

MST using Prim's algorithm :

1. The vertex 1 with its nearest neighbour is added to the spanning tree	
The nearest neighbour of the vertices (1,2) is 5. The edge (1,5) is added	
3. The nearest neighbour of the vertices (1,2, 5) is the vertex 4. The edge (2,4) is added.	
4. The nearest neighbour vertex 3 is added through the edge (1,3).	

Q. 28 Explain Kruskal's algorithm .

Ans. : Kruskal's algorithm

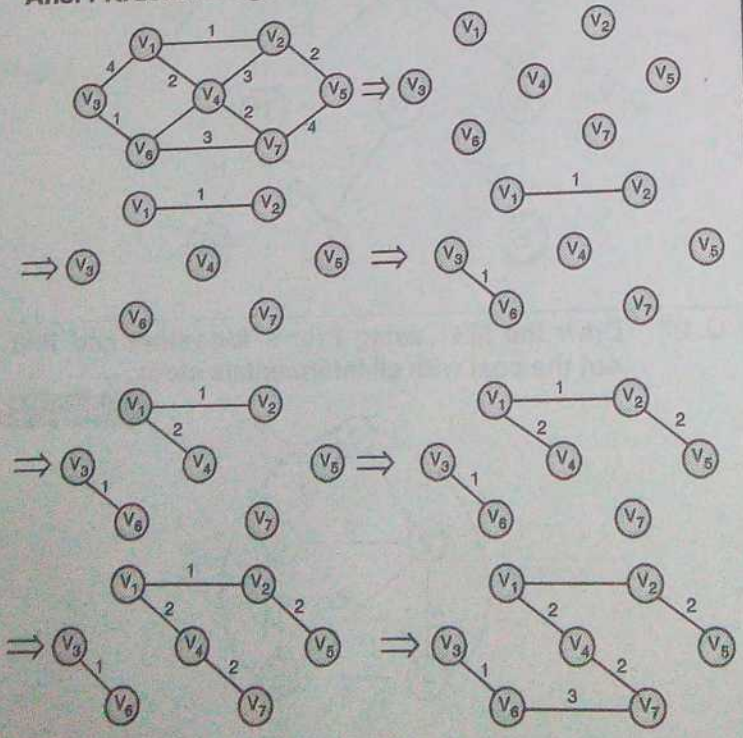
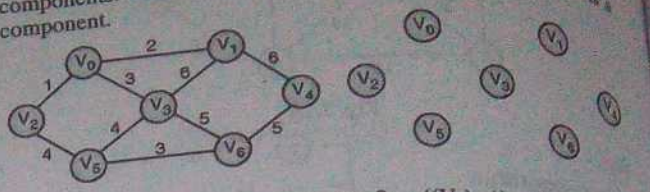


Fig. 7.31: A graph G and its minimum cost spanning tree

It is another method for finding the minimum cost spanning tree of the given graph. In Kruskal's algorithm, edges are added to the spanning tree in increasing order of cost. If the edge E forms a cycle in the spanning, it is discarded. Fig. 7.31 shows the sequence in which the edges are added to the spanning tree.

Formally, Kruskal's algorithm maintains a collection of components. Initially, there are n components. Every node is a component.



Component number 0 = $((V_0), \phi)$ –[A component with one vertex V_0 and no edges]

Components number 1 = $((V_1), \phi)$

Components number 2 = $((V_2), \phi)$

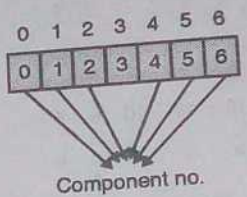
Components number 3 = $((V_3), \phi)$

Components number 4 = $((V_4), \phi)$

Components number 5 = $((V_5), \phi)$

A vertex V_i belongs to component number k if belongs[V_i] is equal to k.

Initial configuration of belongs[] is



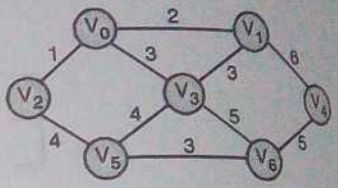


Fig. 7.32 : A graph G and initial list of components

Adding an edge to spanning tree merges two components into one. When an edge (V_0, V_2) with weight 1 is added to the spanning tree, the component with one of the vertices as V_0 and the component with one of the vertices as V_2 will be merged into a single component. To carry out the above operation, two functions are written :

Find (V) → gives the component number of the vertex V

Union (C_1, C_2) → merges two components C_1 and C_2 into C_1.

Steps required for addition of an edge to spanning tree

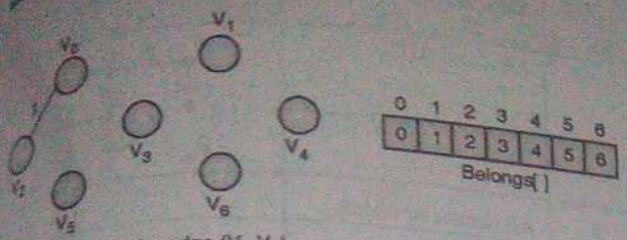
Let the edge to be added is (V_0, V_2)

```
CND1 = Find (V0);      CND2 = Find (V2)
if (CND1 = = CND2)
{
    edge (V0, V2) should not be added as adding an edge to a
    connected component will form a cycle
}
else
    union (V0, V2)
```

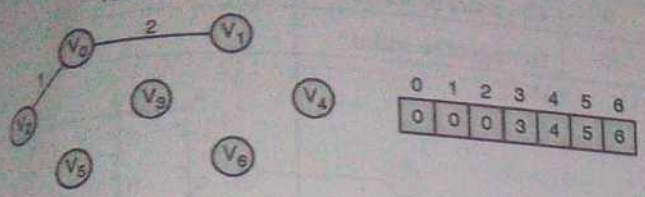
Stepwise **addition of edges and the status of belongs[] at every stage for the graph** of Fig. 7.32.

Edges are added in the increasing sequence of their weights.

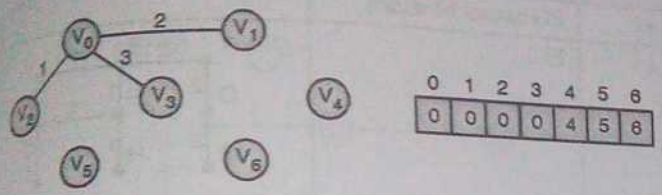
Add the edge (V_0, V_2)



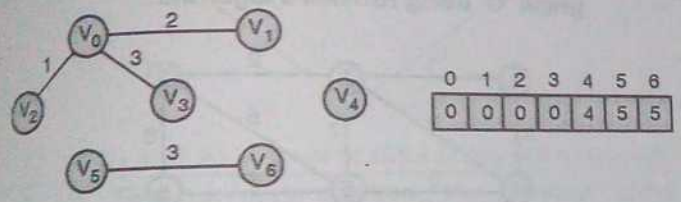
Add the edge (V_0, V_1)



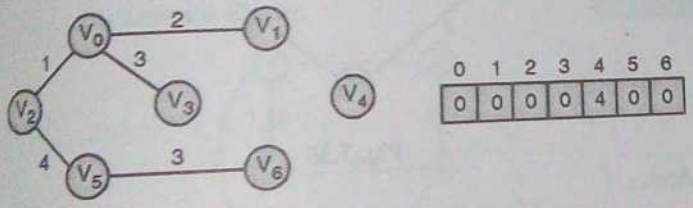
Add the edge (V_0, V_3)



Add the edge (V_0, V_6)



Add the edge (V_2, V_5)



Add the edge (V_3, V_5)

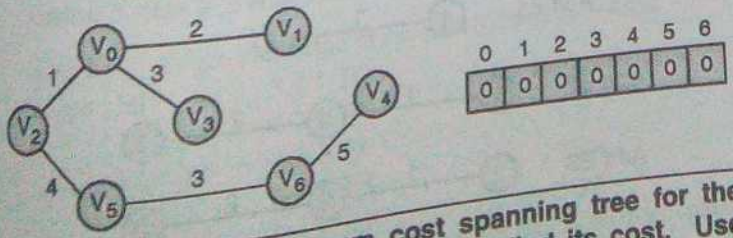
Cannot be added as the component number of V_3 is equal to the component number of V_5.

Add the edge (V_3, V_6)

Cannot be added as the component number of V_3 is equal to the component number of V_6.

Add the edge (V_4, V_6)

Add the edge (V_0, V_6)



Q. 29 Draw the minimum cost spanning tree for the graph given below and also find its cost. Use Kruskal algorithm.

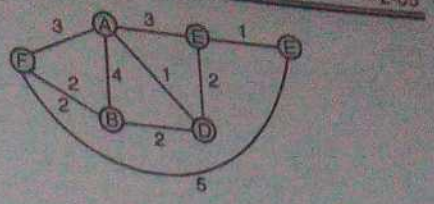


Fig. 7.33

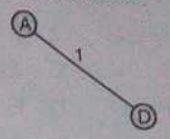
Ans. :

1. Edges are sorted in ascending order on weight.
   Edges :

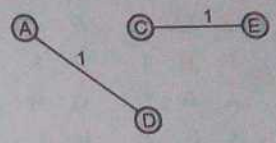
E1	: A D 1	E5	: B F 2	E9	: E F 5
E2	: C E 1	E6	: A C 3		
E3	: B D 2	E7	: A F 3		
E4	: C D 2	E8	: A B 4		

2. Edges are added in sequence E1 to E9 to spanning tree. If an edge forms a cycle, it is discarded.

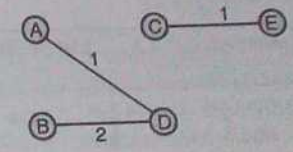
Add E1



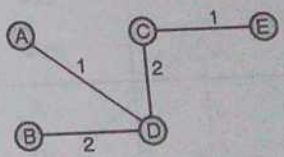
Add E2



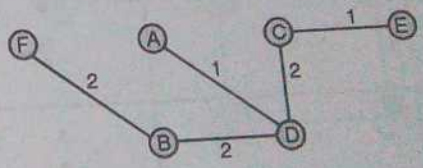
Add E3



Add E4



Add E5



Cost of the minimum cost spanning tree
$= 2 + 2 + 2 + 1 + 1 = 8$

Q. 30 For the graph given below, draw the adjacency matrix and find out minimum spanning tree using Kruskal's algorithm.

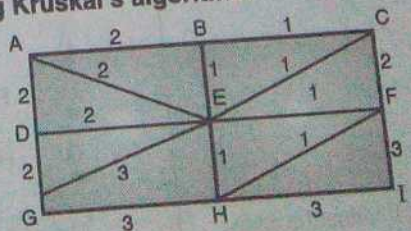


Fig. 7.34

Ans. :

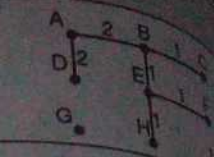
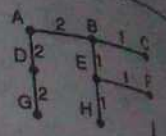
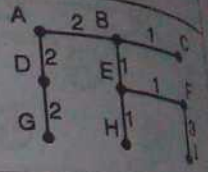
(a) Adjacency matrix representation :

	A	B	C	D	E	F	G	H	I
A	0	2	0	2	2	0	0	0	0
B	2	0	1	0	1	0	0	0	0
C	0	1	0	0	1	2	0	0	0
D	2	0	0	0	2	0	2	0	0
E	2	1	1	2	0	1	3	1	0
F	0	0	2	0	1	0	0	1	3
G	0	0	0	2	3	0	0	3	0
H	0	0	0	0	1	1	3	0	3
I	0	0	0	0	0	3	0	3	0

(b) Spanning tree using Kruskal's algorithm :

List of edges (sorted on weight)

B	C	1	A	E	2
B	E	1	C	F	2
C	E	1	D	E	2
E	F	1	D	G	2
E	H	1	E	G	3
F	H	1	F	I	3
A	B	2	G	H	3
A	D	2	H	I	3

Sr. No.	Edge selected	List of components
1.	Initial	A B C D E F G H I
2.	BC 1	A, B 1 C, D E F, G H I
3.	BE 1	A, B 1 C, D E 1 F, G H I
4.	CE 1 cannot be added	
5.	EF 1, EH 1	A, B 1 C, D E 1 1 F, G H 1 I
6.	FH 1 cannot be added	
7.	AB 2 and AD 2	
8.	AE cannot be added	
9.	CF cannot be added	
10.	DE cannot be added	
11.	DG 2	
12.	EG cannot be added	
13.	FI 3	

Q. 31 : Obtain minimum spanning tree for the following graph 'G' using Kruskal's algorithm.

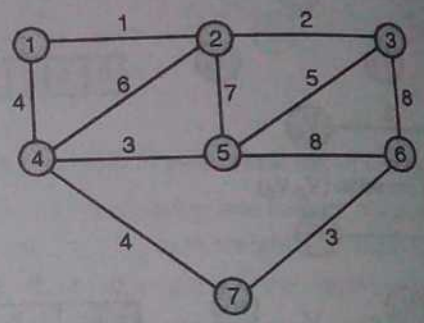


Fig. 7.35

Ans. :. :

1. Edges are sorted in ascending order on weight.

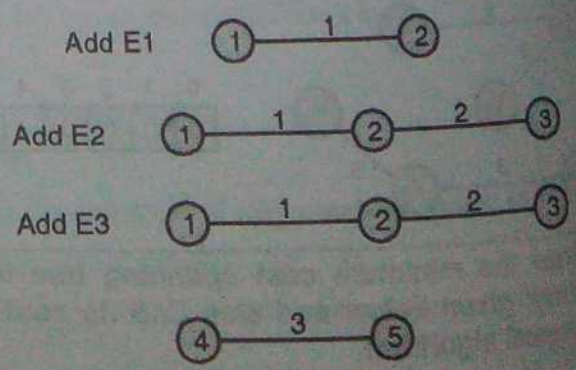
Edges :

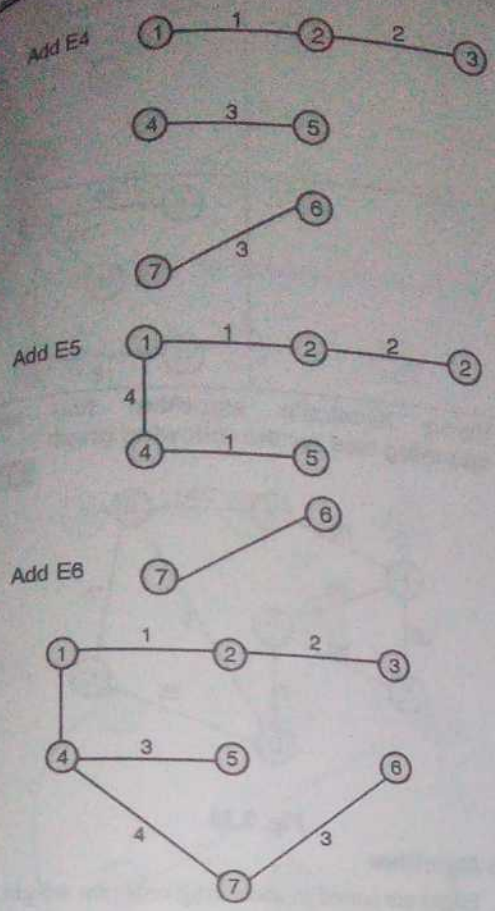
E1 : 1 2 1 E2 : 2 3 2 E3 : 4 5 3 E4 : 6 7 3

E5 : 1 4 4 E6 : 4 7 4 E7 : 3 5 5 E8 : 2 4 6

E9: 2 5 7 E10 : 3 6 8 E11: 5 6 8

2. Edges are added in sequence E1 to E11 to spanning tree. If an edge forms a cycle, it is discarded.





E7, E8, E9, E10, E11 can not be added as they will form a cycle.

Q. 32 Find minimum spanning tree for graph given in Fig. 7.36 using Kruskal's algorithm. Show various steps. Dec. 2013

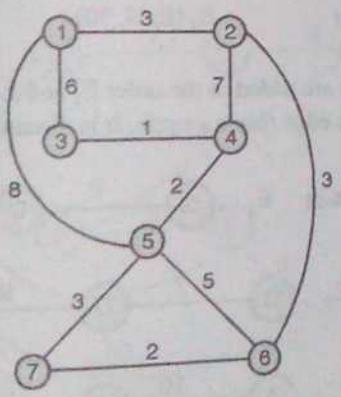


Fig. 7.36

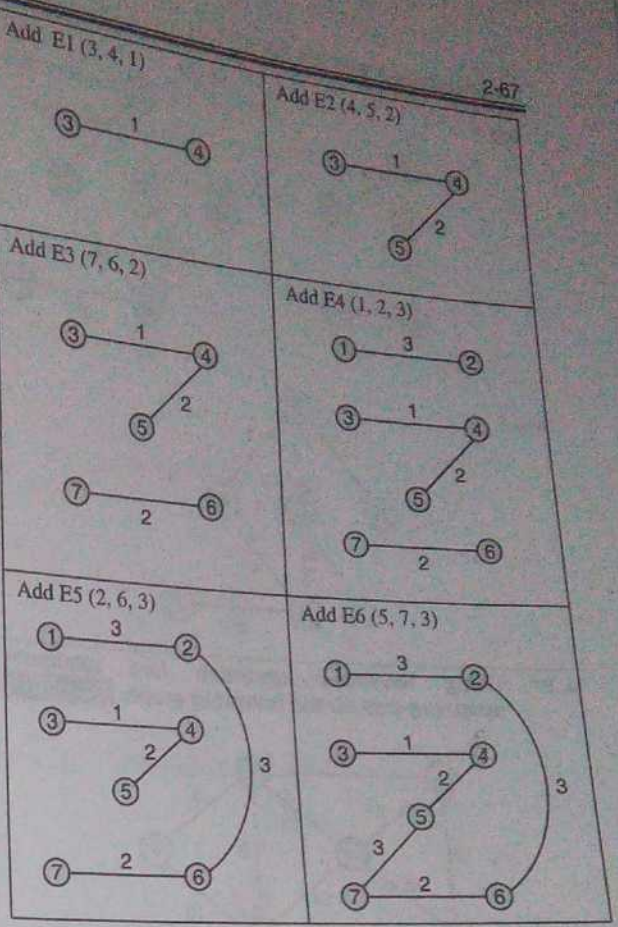
Ans. :

Using Kruskal's algorithm

Step I : Edges are sorted in ascending order on weight

Edges : E1	:	3 4 1	E2	:	4 5 2
E3	:	6 7 2	E4	:	1 2 3
E5	:	2 7 3	E6	:	1 3 6
E7	:	5 7 5	E8	:	1 5 8
E9	:	2 4 7	E10	:	

Step II : Edges are added in sequence E1 to E10 to spanning tree. If an edges forms a cycle, it is discarded.



Other edges can not be added as they will form a cycle.

Q. 33 Find the minimum spanning tree for the given graph using Kruskal's algorithm. Also find its cost with all intermediate steps May 2015

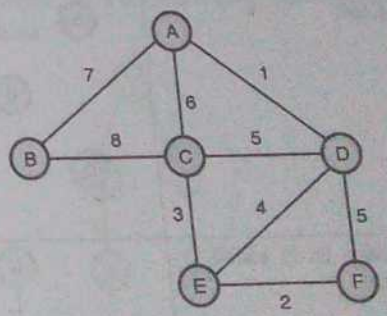
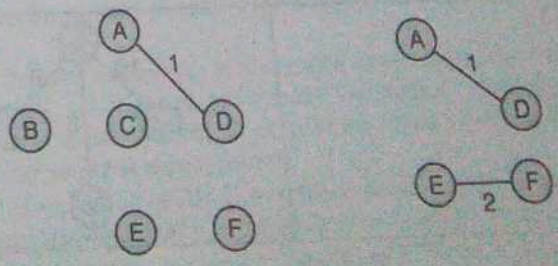


Fig. 7.37

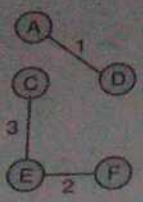
Ans. :

Step 1

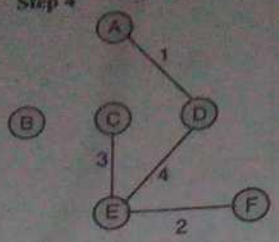
Step 2



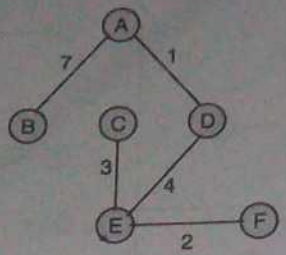
Step 3



Step 4



Step 5



Q. 34 Using Kruskal's algorithm find minimum spanning tree for the following graph. Dec. 2015

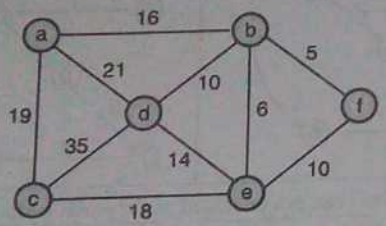
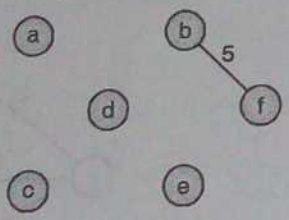
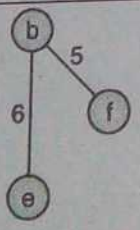
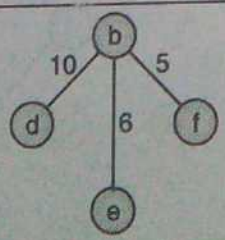
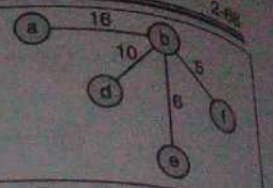
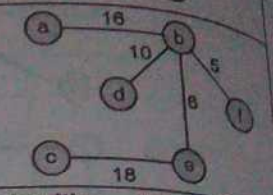


Fig. 7.38

Ans. :

Step 1 : Edge bf is added to spanning tree	
Step 2 : Edge be is added to spanning tree	
Step 3 :	

Step 4 :	
Step 5 :	

Q. 35 Using Kruskal's algorithm find minimum spanning tree for the following graph : May 2016

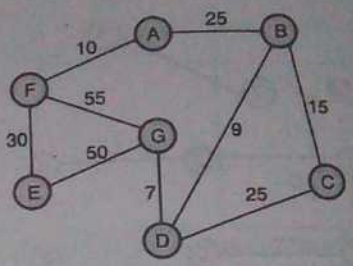


Fig. 7.39

Kruskals Algorithm

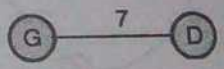
Step 1 : Edges are sorted in ascending order on weight :

Edges

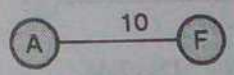
E_1 (G, D, 7)	E_2 (B, D, 9)
E_3 (A, F, 10)	E_4 (B, C, 15)
E_5 (A, B, 25)	E_6 (C, D, 25)
E_7 (F, E, 30)	E_8 (E, G, 50)
E_9 (F, G 55)	

Step 2 : Edges are added in the order E_1 and E_9 to spanning tree. If an edge forms a cycle. It is discarded :

Add E_1

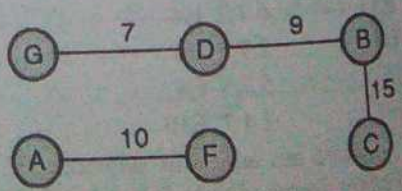


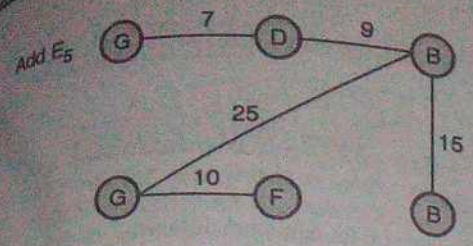
Add E_3



Add E_2

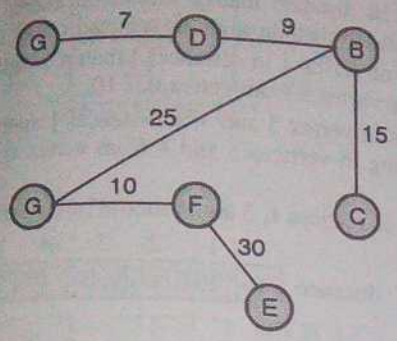
Add E_4



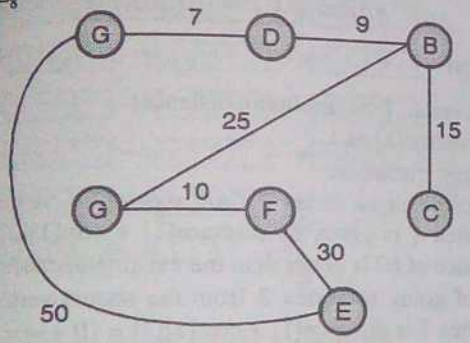


Add E_6 (C, D, 25) can't be added, as it forms a cycle.

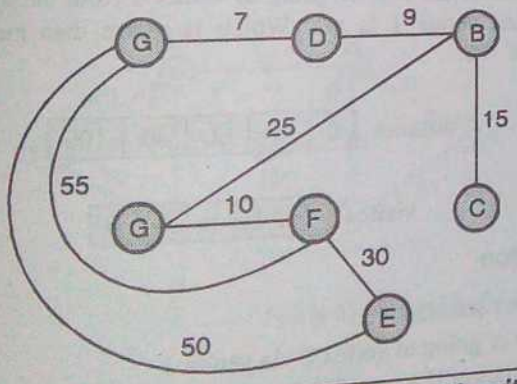
Add E_7



Add E_8



Add E_9



Q. 36 Draw the MST using Kruskal's Algorithm and find out the cost with all intermediate steps. May 2017

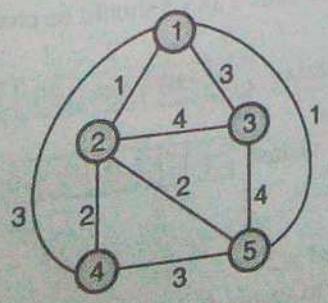


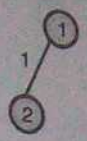
Fig. 7.40

Ans. :

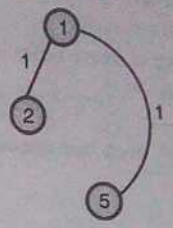
MST using Kruskal's algorithm

Elements are added in the order of their weight.

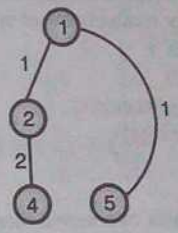
1. **The edge (1,2) is added.**



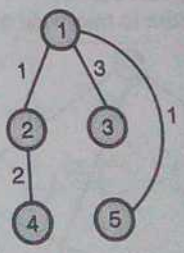
2. **The edge (1,5) is added.**



3. **The edge (2,4) is added.**



4. **The edge (2,5) will form a loop. The edge (1,3) is added.**



Other edges can not be added.

Q. 37 Explain Dijkstra's Algorithm

Ans. : Dijkstra's Algorithm

Let G = (V,E) be a graph with n vertices. The problem is to find out the shortest distance from a vertex to all other vertices of a graph.

Dijkstra's algorithm is also called single source shortest path algorithm. It is based on "greedy" technique. The algorithm maintains a list 'visited[ ]' of vertices, whose shortest distance from the source is already known.

If visited[1], equals 1, then the shortest distance of vertex i is already known. Initially, visited[i] is marked as, for source vertex.

At each step, we mark visited[V] as 1 Vertex V is a vertex at shortest distance from the source vertex. At each step of the

algorithm, shortest distance of each vertex is stored in an array 'distance[ ]'.

Algorithm :

(1) Create cost matrix C[ ][ ] from adjancency matrix adj[ ][ ]. C[i][j] is the cost of going from vertex i to vertex j. If there is no edge between vertices i and j then C[i][j] is infinity.

(2) Array visited [ ] is initialized to zero.

```
for (i = 0; i < n; i++)
    visited[i] = 0;
```

(3) If the vertex 0 is the source vertex then visited[0] is marked as 1.

(4) Create the distance matrix, by storing the cost of vertices from vertex no. 0 to n – 1 from the source vertex 0.

```
for (i = 1; i < n; i++)
    distance[i] = cost[0][i];
```

Initial, distance of source vertex is taken as 0.

i.e. distance[0] = 0;

(5) for (i = 1; i < n; i++)

Choose a vertex w, such that distance[w] is minimum and visited[w] is 0.

Mark visited[w] as 1.

Recalculate the shortest distance of remaining vertices from the source. Only, the vertices not marked as 1 in array visited[ ] should be considered for recalculation of distance.

i.e. f/or each vertex v

if (visited[v] = = 0)

distance[v] = min (distance[v], distance[w] + cost[w][v])

Timing Complexity

The program contains two nested loops each of which has a complexity of O(n). n is number of vertices. So the complexity of algorithm is $O(n^2)$.

Q. 38 Show the working of the Dijkstra Algorithm on the graph given below. (Source vertex is taken as 0)

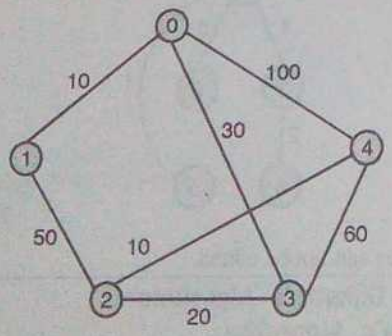


Fig. 7.41

Ans. : Initial configuration of :

(a) Cost matrix

	0	1	2	3	4
0	∞	10	∞	30	100
1	10	∞	50	∞	∞
2	∞	50	∞	20	10
3	30	∞	20	∞	60
4	100	∞	10	60	∞

distance matrix

0	1	2	3	4
0	∞	∞	∞	∞

visited

0	1	2	3	4
0	0	0	0	0

1st Iteration

(a) Select vertex 0 (at minimum distance)

(b) Mark visited[0] as 1

(c) Re-adjust distances of vertices not marked as 1 in visited[ ]. Distance in distance matrix should be altered if there is a better distance path through the selected vertex 0.

Distance of vertex 1 in distance[ ] matrix is infinity. But the cost of going to vertex 1 from vertex 0 is 10.

Distances of vertex 3 and 4 in distance[ ] matrix are ∞. But the cost of going to vertices 3 and 4 from vertex 0 is 30 and 100 respectively.

Distance of vertices 1, 3 and 4 should be modified.

distance

0	1	2	3	4
0	10	∞	30	100

visited

1	0	0	0	0

2nd Iteration

Select vertex 1 (at minimum distance)

Mark visited[1] as 1.

Re-adjust distances.

Cost of going to vertex 2 form the source vertex 0, via the selected vertex 1 is given by distance[1] + cost[1][2] = 10 + 50 = 60. Distance of 60 is better than the existing distance of ∞.

Cost of going to vertex 3 from the source vertex 0, via the selected vertex 1 is distance[1] + cost[1][3] = 10 + ∞ = ∞. Which is worse than the existing distance of 30.

Similarly, the cost of going to vertex 4 from the source, via the selected vertex 1 is ∞. Which is worse than the existing distance of 100.

distance

0	1	2	3	4
0	10	60	30	100

visited

1	1	0	0	0

3rd Iteration

Vertex selected = 3

Cost of going to vertex 2 via vertex 3

= distance[3] + cost[3][2] = 30 + 20 = 50

Cost of going to vertex 4 via vertex 3

= distance[3] + cost[3][4] = 30 + 60 = 90

Distances of vertices 2 and 4 should be changed.

distance

0	1	2	3	4
0	10	50	30	90

visited

1	1	0	1	0

4th Iteration

Vertex selected = 2

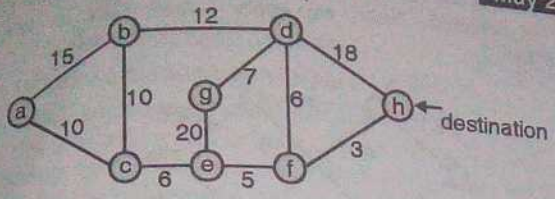
Cost of going to vertex 4 via vertex 2

= distance[2] + cost[2][4] = 50 + 10 = 60

Distance of vertex 4 should be changed.

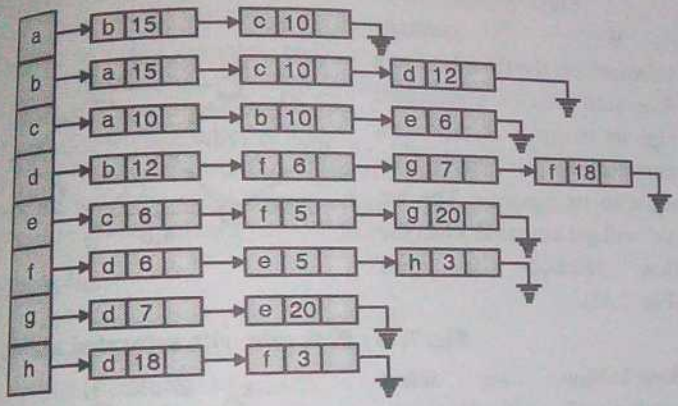
distance	0	10	50	30	60	← final distances
visited	1	1	1	1	0	

Q. 39 Represent the following graph using adjacency list and find the shortest path using Dijkstra's algorithm. Write all the sequence of steps used in the algorithm. May 2014



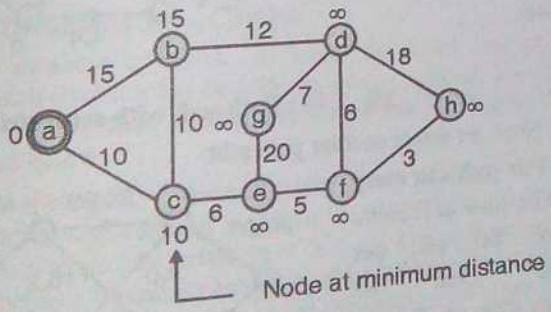
Ans. :

(a) Adjacency list representation :

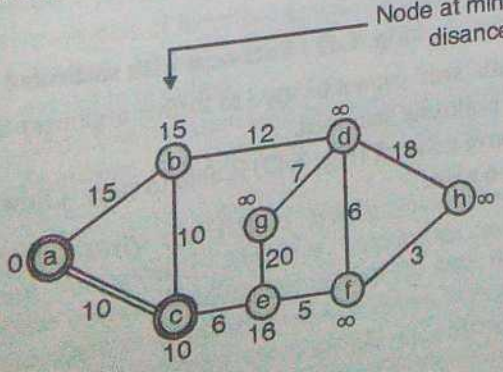


(b) Shortest path using Dijkstra's algorithm :

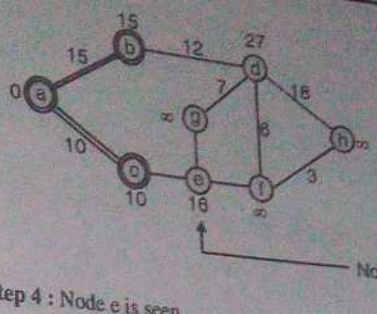
Step 1 : Node a is seen.



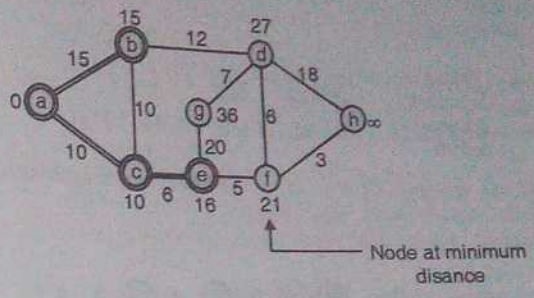
Step 2 : Node c is seen.



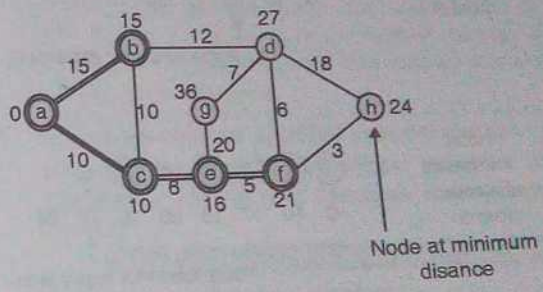
Step 3 : Node b is seen.



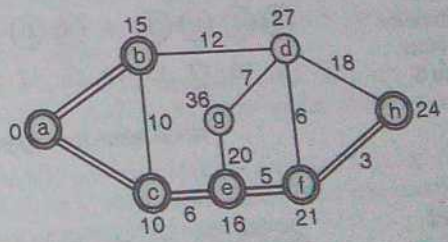
Step 4 : Node e is seen.



Step 5 : Node f is seen.



Step 6 : Node h is seen.



∴ Distance of h = 24

Path from a to h : a → c → e → f → h.

Q. 40 Find the shortest path using Dijkstra's Algorithm. May 2017

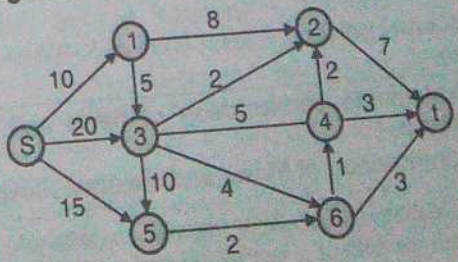


Fig. 7.42

Ans. :

Distance is the minimum distance from the starting mode.

1. Visited minimum distance from S

(S)	1	2	3	4	5	6	t
0	10	∞	20	∞	15	∞	∞

↑ Node taken for expansion

2. Visited minimum distance from S

(S)	(1)	2	3	4	5	6	t
0	10	18	15	∞	15	∞	∞

↑ Node taken for expansion

3. Visited minimum distance from S

(S)	(1)	2	(3)	4	5	6	t
0	10	17	15	20	15	19	∞

↑ Node taken for expansion

4. Visited minimum distance from S

(S)	(1)	2	(3)	4	(5)	6	t
0	10	17	15	20	15	17	∞

↑ Node taken for expansion

5. Visited minimum distance from S

(S)	(1)	(2)	(3)	4	(5)	6	t
0	10	17	15	20	15	17	24

↑ Node taken for expansion

6. Visited minimum distance from S

(S)	(1)	(2)	(3)	4	(5)	(6)	t
0	10	17	15	18	15	17	20

↑ Node taken for expansion

7. Visited minimum distance from S

(S)	(1)	(2)	(3)	(4)	(5)	(6)	t
0	10	17	15	18	15	17	20

∴ Shortest path from S to t is given by S → 5 → 6 → t with length = 20.

Q. 41 Explain the construction of flows and maximal flows.

Ans. : Construction of Flows and Maximal Flows

The construction of flow through a network is being explained with the help of a sample directed network shown in the Fig. 7.43.

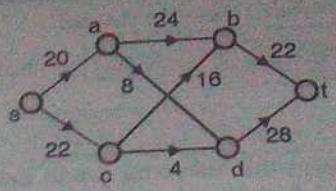


Fig. 7.43 : A sample directed network

Step 1 : Select an arbitrary path from the source to sink. A path 'sabt' is selected. Now, we apply a unit flow on this path (Fig. 7.44)

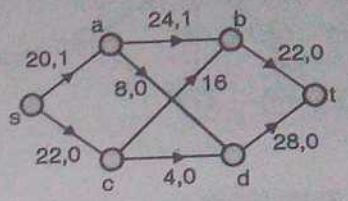


Fig. 7.44 : A unit flow through the path sabt.

We continue incrementing the flow by 1 at a time until there is a saturated pipe on the path. A pipe gets saturated when it carries flow equal to its capacity. The arc 'sa' will get saturated when the flow reached 20 units (Fig. 7.45).

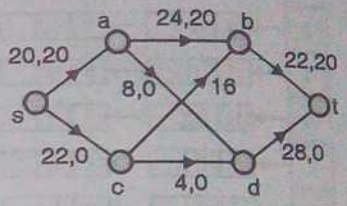


Fig. 7.45 : Path sabt with saturated arc sa.

Step 2 : Now, we select another path 'scdt'. This path can carry a flow of 4 units as the arc 'cd' will get saturated on a flow of 4 units (Fig. 7.46).

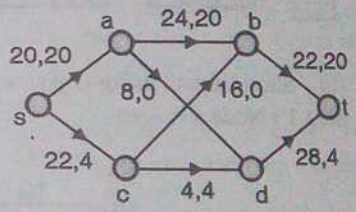


Fig. 7.46 : Path scdt with saturated arc cd

Step 3 : Now, we select another path scbt.

This path can carry an addition flow of 2 units. The arc 'bt' will get saturated after an additional flow of 2 units (Fig. 7.47)

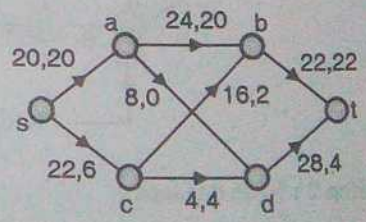


Fig. 7.47 : Path scbt with saturated arc bt.

The path 'sadt' cannot be used to further argument the flow as the arc 'sa' is already saturated.

The above network (Fig. 7.47) is able to carry a flow of $f_{sa} + f_{sc} = 20 + 6$ units.

❑❑❑