Q.1(a) Explain asymptotic notation with example. [5]

(A) Asymptotic Complexity

f(n) and g(n) are both positive going functions. It can be seen that

till nA g(n) > f(n)
till nB f(n) > g(n)

So if two functions keep on toggling over each other than one cannot decide which one is better. But here after n_0, it is clearly seen than f(n) > g(n). Hence we can say that g(n) is better than f(n) asymptotically i.e. after n_0, $f(n) \geq g(n)$.

Asymptotic Comparison

For smaller values of n, the constants play significant role in the complexity values. Those constants are the byproduct of the assumptions done and are required to be overcome. As the value of n increases, the effect of constants gets vanished. We can see functions fluctuating for the smaller values of n, attains a proper rate as the value of n is increased. Hence the comparison of the algorithms is done for the higher values of inputs, such comparison is known as Asymptotic Comparison. The functions used for the asymptotic comparison are known as asymptotic growth functions. Following are some of the asymptotic growth functions used in algorithm analysis.

1) Big O (Upper Bounding) Function
2) Big (Lower Bounding) Function
3) Theta (Order) Function

Asymptotic Growth Functions

1) Big O (Upper Bounding) Function

A function g(n) = O (f(n)) if and only if, there are 2 constants c and n_0 (both > 0) such that, $0 \leq g(n)$ c . f(n) for all $n < n_0$.

Suppose, $g(n) = 502n^2 + 3n - 7$

$f(n) = 715n^2 + 100n + 10$

Here by making C = 1 and $n_0 = 1$ we get, $0 \leq g(n) \leq c$. f (n). Hence g(n) = O (f (n)) and, it is said that g(n) is upper bounded by f(n). If $g(n) = 502n^2 + 3n - 7$ and $f(n) = n^3 + 200n^2 - 5$. Now we want to prove g(n) = O(f(n)) then C = 1 and $n_0 = 502$. If $f(n) = 0.3n^3 + 200n^2 - 5$ then we can say f(n) =

$O(n^3)$. Hence the highest power term matters. Now suppose $g(n) = 502n^2 + 3n - 7$ and $f(n) = n^2 - 7n - 15$. Then we can show $g(n) = 0(f(n))$ by taking $c = 200000$ (say) and $n = 1$ i.e. $g(n) = O(n^2)$. But $f(n) = 7n + 5$ and $g(n) = 502n^2 + 3n - 7$. Now even if we take $c = 200000$, it cannot be proved that there is some n_0 such that $g(n) < c\ f(n)$ for all $n > n_0$. Hence $g(n) \neq O(f(n))$. The Big O function is used for the worst case complexity.

2) Big Omega (Lower Bounding) Function

A function $g(n) = \Omega(f(n))$, if and only if there are constants c and n_0, such that $0 \leq c\ f(n) \leq g(n)$ for all $n > n_0$. So when $g(n) = \Omega(f(n))$, it is said that g(n) is lower bounded by f(n). The Big Omega function is used for the best case complexity.

3) Theta (Order) Function

A function $g(n) = \theta(f(n))$ if and only if there are positive constants c_1, c_2, n_0 such that $0 < c_1\ f(n) < g(n) < c_2\ f(n)$ for all $n > n_0$. So when $g(n) = \theta(f(n))$ then it is said that g(n) is of the order of f(n). The Theta function is used for the average case complexity.

Q.1(b) Explain expression tree? Give example. [5]

(A) Expression Tree

Priority List

We thus obtain the following list of priorities to reflect our usual customs in evaluating operators:

Operators		Priority
↑ ;	all unary operators	6
×	/ %	5
+	– (binary)	4
= =	! =	3
<	<= > >=	2
&&	| |	1
=		0

Drawing a picture is often an excellent way to gain insight into a problem. For our current problem, the appropriate picture is the *expression tree.* Recall that an expression tree is a binary tree in which the leaves are the simple operands and the interior vertices are the operators. If an operator is binary, then it has nonempty subtrees, that are its left and right operand (either simple operands or subexpressions). If an operator is unary, then only one of its subtrees is nonempty, the one on the left or right according as the operator is written on the right or left of its operand.

Let us determine how to evaluate an expression tree such as, for example, the one shown in panel(a) of figure below. It is clear that we must begin with one of the leaves, since it is only the simple operands for which we know the values when staring.

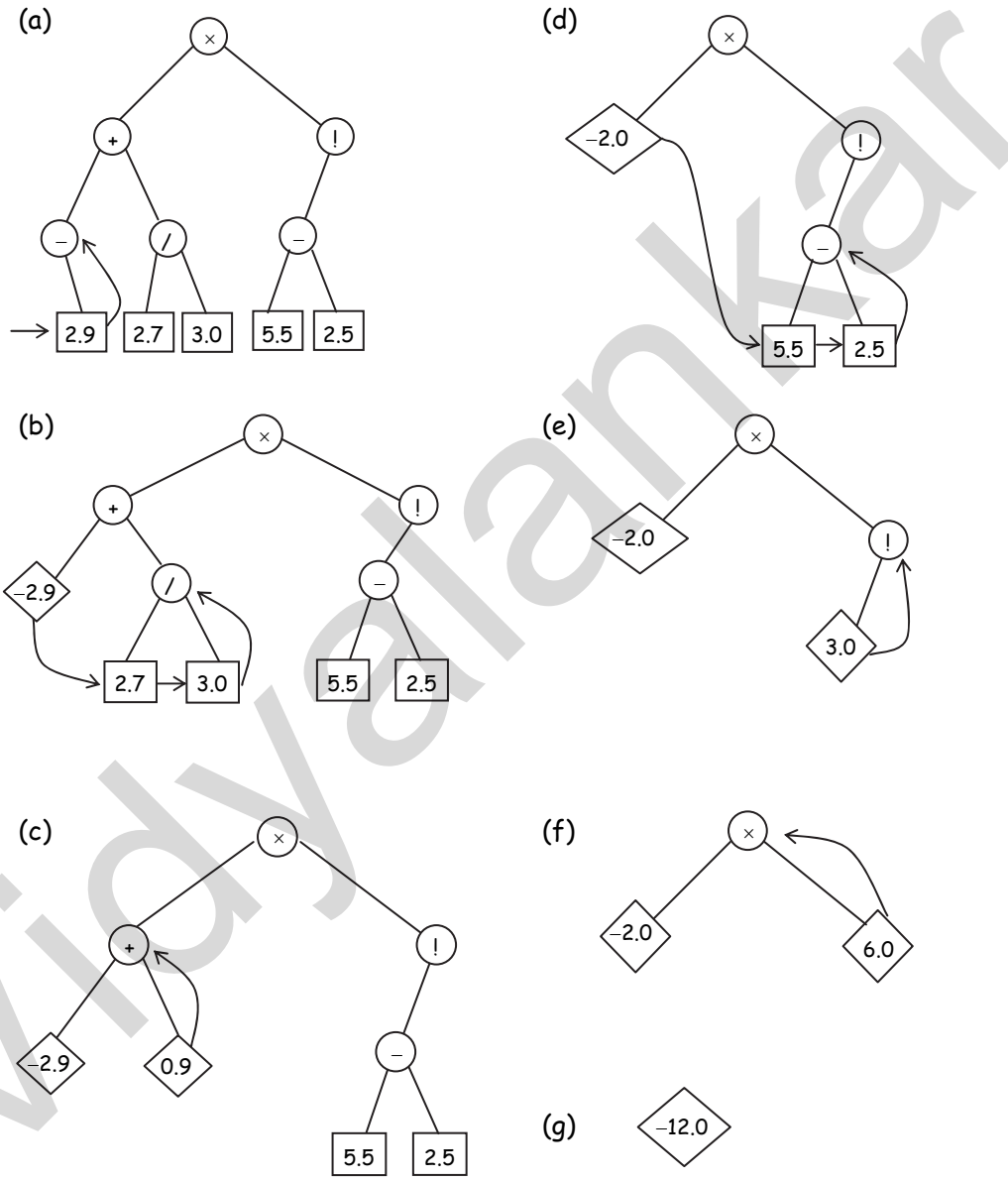


Fig.: Evaluation of an expression tree

To be consistent, let us start with the leftmost leaf, whose value is 2.9. Since, in our example, the operator immediately above this leaf is unary negation, we can apply it immediately and replace both the operator and its operand by the result, –2.9. This step results in the diamond–shaped node in panel (b) of the diagram.

The parent of the diamond–shaped node in panel(b) is a binary operator, and its second operand has not yet been evaluated. We cannot, therefore, apply this operator yet, but must instead consider the next two leaves, as shown by the colored path. After moving past these two leaves, the path moves to their parent operator, which can now be evaluated, and the result placed in the second diamond–shaped node, as shown in panel (c).

At this stage, both operands of the (first) addition are available, so we can perform it, obtaining the simplified tree in panel(d). And so we continue, until the tree has been reduced to a single node, which is the final result. In summary, we have processed the nodes of the tree in the order.

2.9 – 2.7 3.0 / + 5.5 2.5 – ! ×

The general observation is that we should process the subtree rooted at any given operator in the order: "Evaluate its left subtree; Evaluate its right subtree; Perform the operator." (If the operator is unary, then one of these steps is vacuous.) This order is precisely a postorder traversal of the expression tree. We have already observed that the postorder traversal of an expression tree yields the postfix form of the expression, in which each operator is written after its operands, instead of between them.

This simple idea is the key to efficient calculation of expressions by computer. As a matter of fact, our customary way to write arithmetic or logical expressions with the operator between its operands is slightly illogical. The instruction

"Take the number 12 and multiply by ……"

is incomplete until the second factor is given. In the meantime it is necessary to remember both a number and an operation. Form the viewpoint of establishing uniform rules it makes more sense either to write

"Take the numbers 12 and 3; then multiply."

or to write

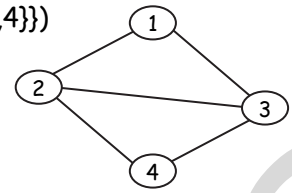
"Do a multiplication. The numbers are 12 and 3."

Q.1(c) Define a graph. What are methods to represent a graph? [5]

(A) Graph is a data structure with 'V' vertices and 'E' edges such that vertices are connected to each other with edges to form closed structure.

In the graph shown below, G(V,E) is graph which is represented by

G = ({1,2,3,4 } { {1,2},{1,3},{2,3},{2,4},{3,4}})



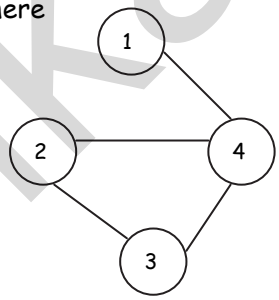
Graph Representations

1) Adjacency Matrix

Let G = (V,E) be a graph of n vertices such that n >=1. Then adjacency matrix of G is a two-dimensional array of n x n say A where

A[i][j] = 1 if edge (Vi, Vj) is present in E(G)

A[i][j] = 0 if edge (Vi, Vj) is absent in E(G).



The graph shown in figure can be represented by,

V(G) = {1,2,3,4}

E(G) = { (1,4), (4,1), (2,3). (3,2), (2,4), (4,2), (3,4), (4,3) }

The same graph can be represented using Adjacency matrix as follows :-

Adj = 0 0 0 0
0 0 1 1
0 1 0 1
0 1 1 1

Adjacency matrix is also known as 'Bit Matrix' as it stores the presence and absence of the edge with the help of bits i.e. 1 for present and 0 for absent.

Now if Adj is the adjacency matrix and Adj[i][j=1 then there exists a path between Vi and Vj.

Adj2 = Adj * Adj (Boolean Matrix Multiplication)

Now if Adj[i][j] =1 then it shows that there exists a path between Vi and Vj of length 2.

Similarly we can have Adj3 = Adj2 * Adj ……. Adjn = Adj(n-1) * Adj.

2) Adjacency List Representation

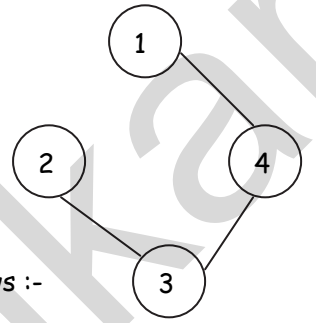
As we have seen earlier that Adjacency matrix representation gives complexity of O(n2). It is because even though fewer edges are present still the traversal cannot be completed till all the n2 elements are traversed.

If the matrix is sparse i.e. not many edges are present then the earlier representation will result in huge amount of wastage of space. In such case, we can use the second type of representation for graph known as 'Adjacency

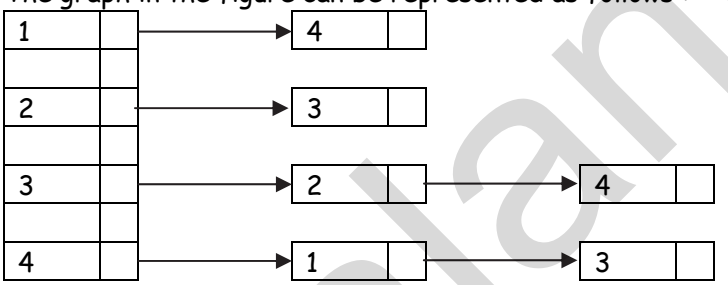
List Representation'. With this the performance can be improved to O(e+n) where n is number of vertices and e is number of edges. In adjacency list representation, n rows of matrix are represented by 'n' liked list. Each linked list stores the nodes adjacent to it. Every node will have two fields namely vertex and link.

The class can be defined as follows :

```
class Node
{
 private int vertex;
 private Node link;
}
```



The graph in the figure can be represented as follows :-



Q.1(d) Define minimum spanning tree. State techniques to compute minimum spanning tree. [5]

(A) Minimum Cost Spanning Tree

Given a connected weighted graph G, it is often desired to create a spanning tree T for G such that the sum of the weights of the tree edges in T is as small as possible. Such a tree is called a minimum cost spanning tree and represents the cheapest way of connecting all the nodes in G.

A greedy method to obtain a minimum cost spanning tree would be to build this tree edge by edge. The next edge to include is chosen according to some optimization criterion. The simplest criterion would be to choose an edge that results in a minimum increase in the sum of the costs of edges so far included. These are two ways to interpret this criterion.

1) The set of edges so far selected should form a tree. Now if A is the set of edges selected so far, then A has to form a tree. The next edge (u,v) to be included in A is a minimum cost edge not in A with the property that A U {(u, v)} also results in a tree. This selection criterion will result in a minimum cost spanning tree. This method is known as Prim's Algorithm.

2) In the second interpretation the edges of the graph are considered in non-decreasing order of cost. The set T of edges so far selected for the spanning tree should be such that it is possible to complete T into a tree. Thus T may not be a tree at all stages in the algorithm. T can be completed into a tree if and only if there are no cycles in T. This method is known as Kruskal's Algorithm.

Q.2(a) Write a program to convert infix to postfix expression. [10]

(A) Translation from Infix form to Polish Form

Very few (if any) programmers habitually write algebraic logical expressions in Polish form, or even in fully bracketed form. To make convenient use of the algorithms we have developed, we must have an efficient method to translate arbitrary expressions from infix form into Polish notation. As a first simplification, we shall consider only the postfix form. Secondly, we shall exclude unary operators that are placed to the right of their operands. Such operators cause no conceptual difficulty, but would make the algorithms more complicated.

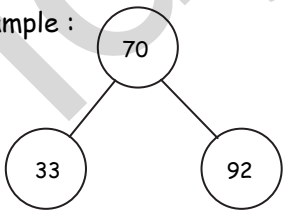
One method that we could use would be to build the expression tree from the infix form and traverse the tree to obtain the postfix form, but, as it turns out, constructing the tree is actually more complicated than constructing the postfix form directly. Since, in postfix form, all operators come after their operands, the task of translation from infix to postfix form is simply.

Q.2(b) Define a binary search tree and write algorithm to implement insertion and deletion. [10]

(A) The binary search tree is based on the binary search algorithm, while creating binary search tree,

values at left subtree < root node value < right subtree values

Example :



Algorithm : (Insertion of a node in a binary tree)

Insertion of a node in a binary tree :

```
void insert (node * root node * new)
{
    if (new → data < root→ data)
```

```
    {
        if (root → left = = NULL)
            root → left = new;
        else
            insert (root → left, new);
    }
    if (new → data > root → data)
    {
        if (root → right = = NULL)
            root → right = New;
        else
            insert (root → right, New);
    }
}
```

Algorithm : Deletion of node in binary search tree.

```
void del (node * root, int key)
{
    node * temp, * parent, * temp_succ;
    temp = search (root, key, & parent);
    /* deleting a node with 2 children */
    if (temp → left !=NULL && temp → right != NULL)
    {
        parent = temp;
        temp_succ = temp → right;
        while (temp_succ → left != NULL)
        {
            parent = temp_succ;
                temp_succ = temp_succ → left;
        }
        temp → data = temp_succ → data;
        parent → right = NULL;
        printf ("Now delete it");
    }
}
```

Q.3(a) Explain quick sort using an example. **[10]**

(A) Quick Sort

Quick Sort is used to sort the bulk amount of data. Here it works on partition technique. In each pass the aim is to place one element called 'pivot' element at its proper position within the array, so that all preceding element will be smaller

than the pivot element. Whereas all the succeeding elements will be greater than the pivot. This process is repeated by selecting different pivot elements using appropriate partition technique. This continues till all the elements in the array are sorted.

Algorithm :

Here an initial array is partitioned into two sub-arrays using a pivot element. Then they can be processed further using iterative or recursive approach. A general algorithm is given below which uses recursive approach.

Function Partition :

1) initialize value with the element at lower bound(lb)
2) initialize down with lb and i with down+1
3) if element at i is less than value then
   a) increment down
   b) swap element at I with element at down
4) increment i
5) if i<n then goto step 3
6) swap element at lb with element at down
7) return down

Function quicksort :

1) Partition the current array into two sub-arrays.
2) Invoke Quick Sort to sort the left sub array.
3) Invoke Quick Sort to sort the right sub array.

Analysis :

Suppose that the position of the pivot element turns out to be the middle position then the complete array of n elements will get divided into 2 arrays of n/2 elements each. Further each of its will get divided which results in 4 arrays of n/4 elements each. This will continue m times where m = log n. So finally there would be n arrays of n/n elements each. So the total number of comparisons can be calculated as follows.

Total = n + 2*(n/2)+4(n/4) + … + n*(n/n) = n + n + n … m times

= n * m = n log n

Hence Quick Sort gives the efficiency of O(n log n). But Quick Sort behaves differently when the array is originally sorted. When the sort starts, x[lb] i.e. x[0] will be compared with all the elements which results in n comparisons. As x[lb] is in its correct position, the array will get divided into two sub-arrays of 0 and n-1 elements. The process will repeat resulting in n-1 comparisons in next pass and we will have n such passes. So the total comparisons for the entire sort can be calculated as follows.

Total = n + (n-1) + (n-2) + … + 1
= n (n-1) /2

Hence it shows that Quick Sort gives efficiency of O(n2) when the array is already sorted.

Sample Output : Initial Array

7	54	29	41	12	5	78	35	22	18

lb = 0 ub = 9

5	7	29	41	12	54	78	35	22	18

lb = 2 ub = 9

5	7	18	12	22	29	78	35	41	54

lb = 2 ub = 4

5	7	12	18	22	29	78	35	41	54

lb = 6 ub = 9

5	7	12	18	22	29	54	35	41	78

lb = 6 ub = 8

5	7	12	18	22	29	41	35	54	78

lb = 6 ub = 7

5	7	12	18	22	29	35	41	54	78

Sorted Array

5	7	12	18	22	29	35	41	54	78

Q.3(b) Define a Queue as ADT. Implement any two operation in it. [10]

(A) Queue as an ADT

The queue is a FIFO data structure in which the element which is inserted can be the first to be removed. The ADT for the queue will be as follows :

AbstractDataType queue

{

Instance : The queue is collection of element in which the element can be inserted one end called rear and elements get deleted from the end called front.

Operation :

1. que_full( ) : Checks whether queue is full or not.
2. que_empty( ) : Checks whether queue is empty or not.
3. Q_insert( ) : Inserts the element in queue from rear end.
4. Q_delete( ) : Delete the element from the queue by front end.

Thus the ADT for queue gives the abstract for what has to be implemented which are the various operations on queue. But it never specifies how to implement them.

Queue Operations :

As we have seen, queue is nothing but the collection of items. Both the end in the queue are having their own functionality. The queue is also called as FIFO i.e. a first in first out data structure. All the elements in the queue are stored sequentially. The various operations on the queue are :

1. Queue overflow
2. Insertion of the element into the queue
3. Queue underflow
4. Deletion of the element from the queue
5. Display of the queue

1. Insertion of element into the queue

   The insertion of any element in the queue will always take place from the rest end.

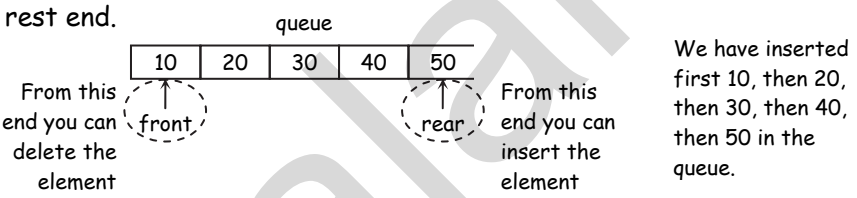


Fig.: Representation of insertion.

Before performing insert operation you must check whether the queue is full or not. If the rear pointer is going beyond the maximum size of the queue then the queue overflow occurs.

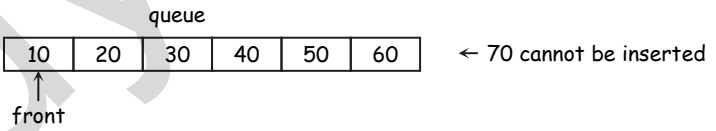


Fig.: Representation the queue overflow.

2. Deletion of the element from queue

   The deletion of any element in the queue takes place by the front end always.

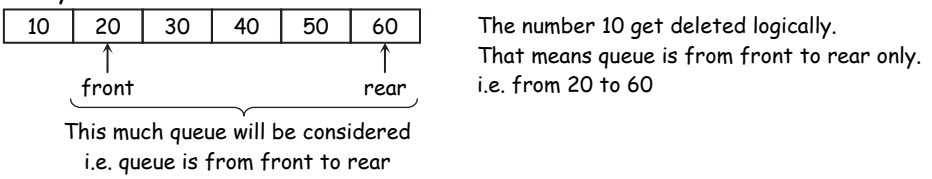
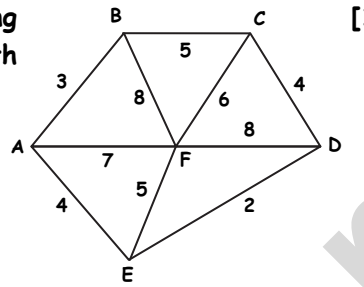


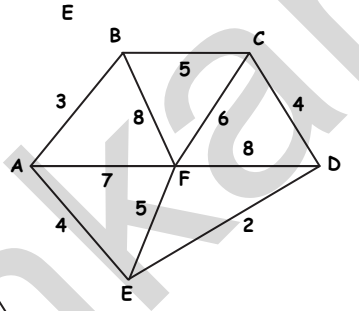
Fig.: Representing the deletion.

Q.4(a) Draw a minimum cost spanning tree using Kruskal's algorithm. Also find its cost with its intermediate steps. **[10]**

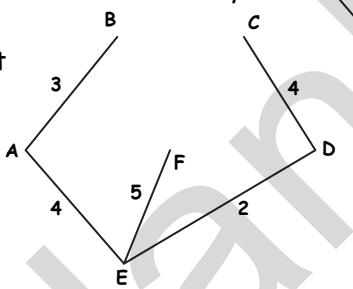


(A) Set the edges in ascending order,

Shortest edges is, 1) ED = 2
2) AB = 3
3) CD = 4
4) AE = 4
5) BC = 5 xx create a cycle.



Select the next edge that does not create a cycle



Thus the total weight of The tree is 18.

Q.4(b) Write a Pseudo Code/algorithm to insert a node in a linked list. **[5]**

(A) General Linked List

```
typedef struct
{
  int data;
  struct node * next;
}
    node;
typedef struct
{
    node * stack;
} head;

void insertbeg (head * t, int ele)
{
    node * p = (node *) malloc (sizeof (node));
    p → data = ele;
    p → next = t → start;
    t → start = p;
}
void insert end (head * t, int ele)
```

```
{
    node * p = (node *) malloc (size of (node));
    p → data = ele;
    p → next = NULL;
    if (t → stack = NULL) // linked list empty// t → start = p;
    else
    {
        node * q = t → stack;
        while (q → next ! = NULL)
            q = q → next;
        q → next = p;
    }
}
```

Q.4(c) Write a function to demonstrate depth first search. **[5]**

(A)

```
void dfs (int node, int adj ( ) [10], int n, int visited [ ] )
{
    int i
    visited [node] =1,
    printf ["%d", node + 1);
    for (i = 0; i < n; i++)
       if (adj [node] [i] = = 1 && visited [i] = = 0)
            dfs (i, adj, n, visited);
}
void main( )
{
    int adj [10] [10], visited [10] = {0}, n, e, i, v, vz, node;
    printf (" Enter no of nodes \n");
    scanf (" %d", &n);
    printf ("enter no of edges)n");
    scanf (" %d", & e);
    printf ("Enter %d edges \n", e);
    for (i = 0; i  < e; i++)
    {
        scanf ("% d % d", & v1, & v2);
        adj [v1−1] [v2−1] = adj [v2−1]v1−1] = 1;
    }
    printf (enter starting vertex \n");
    scanf (" %d", & node);
    dfs(node −1, adj, n, visited);
}
```

Q.5(a) Explain priority queue and give its complementation of the same? [10]

(A) Priority Queue

A queue in which we are able to insert items or remove items from any position based on some property (such as priority of the task to be processed) is often referred to as a priority queue. Figure 1(a) represents a priority queue of jobs waiting to use a computer. Priorities of 1, 2 and 3 have been attached to jobs of type real-time, on-line, and batch, respectively. Therefore, if a job is initiated with priority i, it is inserted immediately at the end of the list of other jobs with priority i, for i = 1, 2 or 3. In this example, jobs are always removed from the front of the queue. (In general, this is not a necessary restriction on a priority queue.)

A priority queue can be conceptualized as a series of queue representing situations in which it is known a priori what priorities are associated with queue items. Figure 1(b) shows how the single-priority queue can be visualized as three separate queue, each exhibiting a strictly FIFO behavior. Elements in the second queue are removed only when the first queue is empty, and elements from the third queue are removed only when the first and second queues are empty. This separation of a single-priority queue into a series of queues also suggests an efficient storage representation of a priority queue. When elements are inserted, they are always added at the end of one of the queues as determined by the priority. Alternatively, if a single sequential storage structure is used for the priority queue, then insertion may mean that the new element must be placed in the middle of the structure.

Task identification

R_1	R_2	...	R_{i-1}	O_1	O_2	...	O_{i-1}	B_1	B_2	...	B_{k-1}	...
1	1	...	1	2	2	...	2	3	4	...	3	...

Priority ↑ R_i ↑ R_j ↑ R_k

Fig. 1(a) : A priority queue viewed as a single queue with insertion allowed at any position.

Priority 1

R_1	R_2	...	R_{i-1}	...	← R_i

Priority 2

O_1	O_2	...	O_{i-1}	...	← O_j

Priority 3

B_1	B_2	...	B_{k-1}	...	← B_k

Fig. 1(b) : A priority queue viewed as a setoff queue.

This can require the movement of several items. It is better to split the priority queue into several queues, each having its own storage structure.

Q.5(b) Construct a binary tree for the inorder and post-order traversal. [10]

Inorder : "INFORMATION"

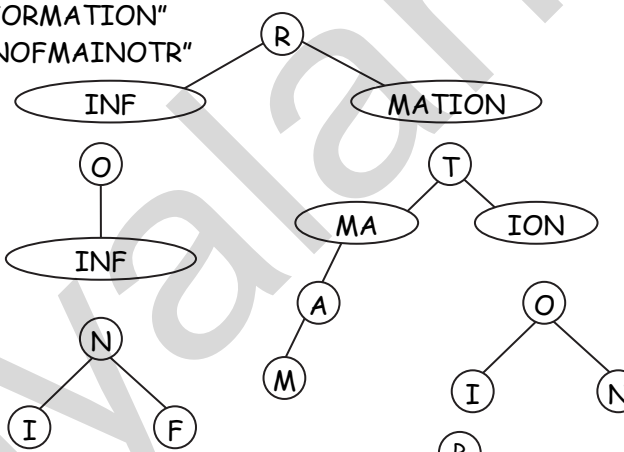
Postorder : "INOFMAINOTR"

(A) Function to traverse a tree in postorder.

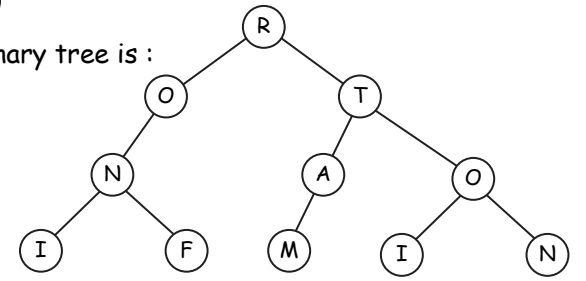
```
void Tree(node * T)
{
    if (T! = Null)
    {
        Postorder T → left;
        Postorder T → right;
        printf("%d", T → data);
    }
}
```

Inorder- "INFORMATION"

Postorder- "INOFMAINOTR"



The binary tree is :



Inorder : INFORMATION

Postorder : INOFMAINOTR

Q.6 Write short notes on any FOUR : [20]

Q.6(a) Write short note on Euclid's algorithm. [5]

(A) Euclid's algorithm

Euclid's algorithm is an efficient method for computing the greatest common divisor (GCD) of two numbers, the largest number that divides both of them without leaving a remainder.

The Euclid's algorithm proceeds in a series of steps such that the output of each step is used as an input for the next one. Let K be an integer that counts the steps of the algorithm, starting with zero.
The algorithm can be written in sequence of equations :

$a = q_0 b + r_0$
$b = q_1 r_0 + r_1$
$r_0 = q_1 r_1 + r_2$
$r_1 = q_3 r_2 + r_3$
⋮

Q.6(b) Write short note on Radix Sort. [5]

(A) Radix Sort

This sort works for multiple digits numbers by using Bucket Sort for individual digits. So it keeps 10 buckets representing 0 to 9 digits. All the numbers are rearranged in those buckets using their least significant digit. Then this process is repeated for the remaining digits one by one till the most significant digit.

Algorithm :

1) Create an array a [0 … n-1] elements.
2) Call bucket sort repeatedly on least to most significant digit of each element as the key.
3) Return the sorted array.

Analysis :

The algorithms takes O(n) for each bucket sort as calculated previously and here we require 'd' bucket sorts where d is the number of digits in the largest number say 'k'. So it can be represented as $d = \log_{10} k$.
Hence total comparisons = n* d
= n* log k
Now if number of inputs is large then n nearly approaches k. Hence number of comparisons can be approximated to n log n. So the worst case efficiency of Radix Sort is O(n log n). Like Bucket Sort, here also we need to use a huge array which degrades the space efficiency.

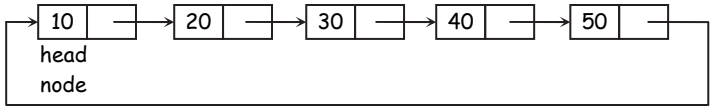
Sample Output : Initial Array

23	814	925	545	2547	21237	7	24727	87	28
7	814	23	925	24727	28	21237	545	2547	87
7	23	28	87	21237	545	2547	24727	814	925
7	23	28	87	545	814	925	21237	2547	24727
7	23	28	87	545	814	925	2547	21237	24727

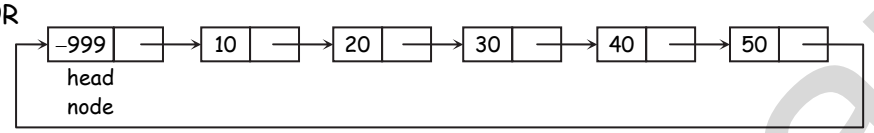
Q.6(c) Write short note on Circular Linked List. [5]

(A) Circular Linked List

The circular linked list is as shown below :



OR



The circular linked list (CLL) is similar to singly linked list accept that the last node next pointer points to first node.

Various operations that can be performed on circular linked list are :

1. Creation of circular linked list.
2. Insertion of a node in circular linked list.
3. Deletion of any node from linked list.
4. Display of circular linked list.

We will see each operation along with some example :

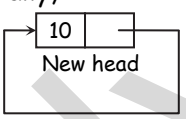
Creation of circular linked list :

```
struct node *Create( )
{
char ans;
int flag = 1;
struct node *head,*New,*temp;
struct node *get node( );
clrscr( );
do
New = get_node( );
        printf("\n\n\n\tEnter The Element\n");
        scanf("%d",&New -> data);
if(flag= =1) /*flag for setting the starting node */
{
    head = New;
    New -> next = head;
    flag = 0, /*rest flag*/
}
else          /*find last node in list */
{
    temp = head,
    while (temp->next i= head)      /*finding the last node */
```

```
            temp=temp->next;        /*temp is a last node */
            temp->next=New;         /*attaching the node */
            New->next=head;         /*each time making the list circular */
}
printf("\n Do you want to enter more nodes?");
}while(ans= = 'y'||ans= ='Y');
return head;
}
/*function used while allocating memory for a node*/
struct node *get_node( );
{
struct node *New;
New = (node *) malloc(sizeof(struct node));
New->next = NULL;
return(New);    /*created node in returned to calling function*/
}
```

Initially we will allocate memory for New node using a function get node( ). There is one variable flag whose purpose is to check whether first node is created or not. That means flag is 1 (set) then first node is not created. Therefore after creation of first node we have to reset the flag (making flag = 0).

Initially,

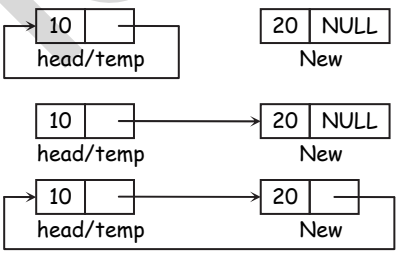


Suppose we have taken element '10' the flag = 1,
head = New,
New → next = head;
flat = 0,

Here variable head indicates starting node.

Now as flag = 0, we can further create the nodes and attach them as follows:

When we have taken element '20'



Q.6(d) Write short note on Merge Sort. **[5]**

(A) Merge Sort

Algorithm :

1) Create array a[0 … n-1] of n elements.
2) Consider size equal to unity.
3) Subdivide the array in sub-arrays of 'size'.
4) Merge these sub arrays in pairs of two such that they remain sorted.
5) Double the size.
6) if size <= array size then goto step 3
7) Return the sorted array.

Analysis : Here if the size of the input is n then the number of passes required is m such that $n = 2^m$.
Hence log n = m. So number of passes are m = log n.

Maximum number of comparisons made in each pass is n. Hence total comparisons can be calculated as follows:

Total comparisons = Comparisons in each pass * No. of passes
= n * m
= n* log n

Hence the worst case complexity of Merge Sort is O(n log n). Here the space complexity is not good. This sort gives very good time efficiency but requires extra space other than the array of input elements. The array 'temp' which of the same size as the input array is used while sorting. Generally a sorting technique which does not use extra space other than the set of input array, is known as 'In place Sorting' technique. Merge Sort is not 'In Place Sorting' technique.

Sample Output : Initial Array

7	54	29	41	12	5	78	35	22	18
7	54	29	41	5	12	35	78	18	22
7	29	41	54	5	12	35	78	18	22
5	7	12	29	35	41	54	78	18	22
5	7	12	18	22	29	35	41	54	78

Q.6(e) Write short note on searching algorithm. **[5]**

(A) Searching algorithm

Searching algorithm is an algorithm that retrieves information stored within same data structure. The appropriate search algorithm depends upon data structure being searched.

Search algorithms can be classified based on the mechanism of searching linear search checks after record for one associated target key in a linear fashion. Binary or half interval searches repeatedly target the centre of the search structure and divide the search space into half comparison search algorithm improve in linear searching by successively improving and eliminating the records based on comparison of keys.

Finally hashing directly maps keys to record based on a hash function.
Searching functions/algorithms can also be evaluated on the basis of complexity or maximum theoretical run time. Binary search function has maximum time complexity of O(log n) or logarithmic time.

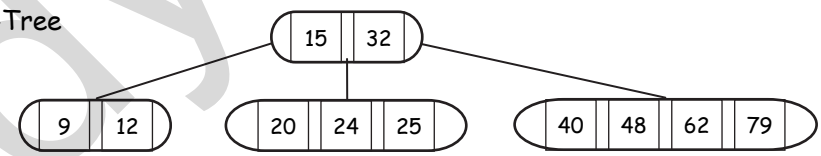
Q.6(f) Write short note on B-tree. [5]

(A) B-tree

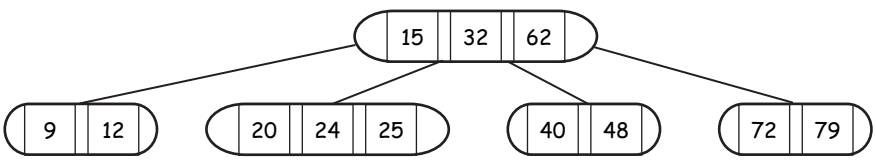
B tree is a balanced multi-way search tree. A B tree of order n is defined as a tree in which each node (except root) has at least n div 2 keys and maximum of n-1 keys and n sons. All leaves of B-tree are always at the same level. This also implies that time needed to search any key in such a tree is always constant and worst case is never achieved. Insertion of any new keys always takes place at leaf. If leaf is already full i.e. if it already has n-1 keys then the node is split in two different nodes of n div 2 keys each and the median is sent up to the immediate ancestor. If the ancestor is also filled then the same procedure is repeated again on the ancestor. Therefore B-tree always grows in upward direction.

Let us take one example to see the construction of B-tree from the available data.

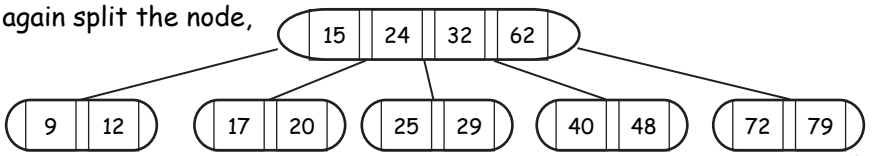
Initial B-Tree



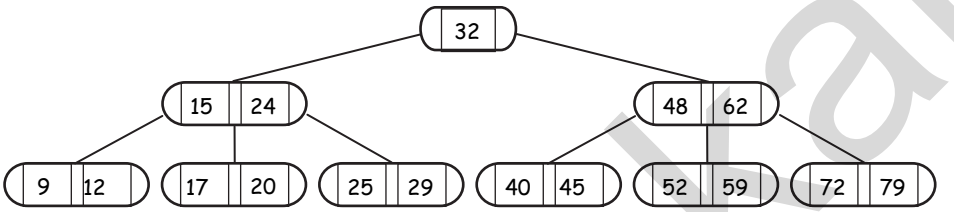
Inserting 72 in the tree will result in splitting the node. As 62 being the mean,



It will be sent up. Insertion of 29 will be straight forward. But insertion of 17 will again split the node,



Now insertion of 52 and 59 will not result in splitting but then insertion 45 will split the node and the resulting B tree will be as follows.



❑ ❑ ❑ ❑ ❑