END TERM EXAMINATION [MAY-JUNE 2016]
SECOND SEMESTER [MCA]
DATA AND FILE STRUCTURE [MCA–102]

Time : 3 hrs. M.M. : 75

Note: *Attempt any six questions including Q. no. 1 which is compulsory.*

Q.1. (a) Compute the time complexity of the following code segment written in C. (5×5=25)

.....

for(j–1.j<n,j–k*j)Statement1:

Assuming that n is size of input data and *K* is a constant

Ans. The complexity of a loop is considered as **O(Logn)** if the loop variables is divided/multiplied by a constant amount.

So for the given statement:

for(j=1; j<n; j=K*j) Satement 1;

The complexity is **O(Logn).**

Q.1.(b) Implement multiastacks using on 1D array. Write algorithms for push and pop operation in th 1th stack.

Ans. Program for representing multiple stacks in one array

```
#include<stdio.h>
#include<stdlib.h>
#include <conio.h>
#define MAX 10
void push(int,int[],int,int);
int pop(int,int[],int);
void display(int,int[],int);
/*array to top index for each stack*/
int t[MAX];
int top[MAX];
void main()
{
int arr[MAX],n,totstk,index;
int i,j,k=0,span,no,ch,num;
/*Input total elements of the array*/
/*This array may be divided into multiple stacks*/
printf("nEnter size of the array?");
scanf("%d",&n);
/*Input total stacks to be created in above array*/
printf("nEnter total stacks to create?");
scanf("%d",&totstk);
/*t[] array will contains totstk indexes for top*/
/*since total stacks to be created are totstk*/
/*Initializing t[] array with -1*/
for(i=0;i<totstk;i++)
t[i]=-1;
/*computes size of one stack*/
span=n/totstk;
/*store top of each stack in the array*/
```

```
/*this top serves as index for push/pop operations*/
top[k++]=0;
for(i=0;i<totstk-1;i++,k++)
top[k]=top[k-1]+span;
/*displaying elements of indexed array*/
for(i=0;i<totstk;i++)
printf("%dt",top[i]);
do{
clrscr();
printf("nStacks-Menu (n stacks:1 array)");
printf("n1. Push");
printf("n2. Pop");
printf("n3. Display");
printf("n4. Quit");
printf("nnEnter your choice?");
scanf("%d",&ch);
if(ch!=4)
{
printf("nEnter stack number (0 to %d)?",totstk-1);
scanf("%d",&index);
}
switch(ch)
{
case 1:
printf("nEnter number to push?");
scanf("%d",&no);
push(index,arr,span,no);
break;
case 2:
num=pop(index,arr,span);
printf("nPopped element from stack %d is %d",index,num);
break;
case 3:
display(index,arr,span);
break;
case 4:
exit(0);
}
getch();
}while(1);
}
void push(int index,int arr[],int n,int num)
{
int topid,topelement;
topid=top[index];
topelement=t[index];
if(topelement==topid+n-1)
{printf("nStack Full! Overflow.");
return;
```



```
}
topelement=++t[index];
arr[topid+topelement]=num;
}
int pop(int index,int arr[],int n)
{
int topid,topelement;
topid=top[index];
topelement=t[index];
if(topelement==-1)
{printf("nStack Empty! Underflow.");
return -1;
}
t[index]--;
return(arr[topid+topelement]);
}
void display(int index,int arr[],int n)
{
int i,topid,topelement;
topid=top[index];
topelement=t[index];
if(topelement==-1)
{printf("nStack Empty! Underflow.");
return;
}
for(i=topid+topelement;i>=topid;i--)
printf("%dt",arr[i]);
}
```

Q.1. (c) Write a c function that detects a circle within a linear linked list.

Ans. int detectcycle(struct node *head)

```
{
  int flag = 1, count = 1, i;
  struct node *p, *q;
  p = q = head;
  q = q->next;
  while (1)
  {
    q = q->next;
    if (flag)
    {
      p = p->next;
    }
    if (q == p)
    {
      /*Deleting the loop to deallocate the list*/
      q = q->next;
      while (q != p)
      {
        count++;
```

```
            q = q->next;
          }
          q = p = head;
          for (i = 0; i < count; i++)
          {
            q = q->next;
          }
          while (p != q)
          {
            p = p->next;
            q = q->next;
          }
          q->next = NULL;

          return 1;
        }
        else if (q->next == NULL)
        {
          return 0;
        }
        flag = !flag;
      }
    }
```

Q.1. (*d*) Prove that for any non empty binary tree *T*, if n_0 is the number of leaf nodes and n_1 is the number of nodes of degree 2, then $n_0 = n_1 + 1$.

Ans. Proof: Let n be the total no. of nodes in binary tree.

n0 no. of nodes of degree 0

n2 no. of nodes of degree 1

n1 no. of nodes of degree 2

n = n0 + n2 + n1 -----> (1)

All the nodes except the root node has a branch coming into it. Let B be the no. of branches in the binary tree.

n = B + 1 ----->(2)

B = 0.n0 + 1 * n2 + 2*n1 since there would be 0 branches from n0, and 1 branch from n2 nodes and two branches from n1 nodes

B = n2 + 2n1 ----->3

Substituting (3) in (2).

n = n2 + 2n1 + 1 -----> 4.

Equate 1 & 4

n0 + n2 + n1 = n2 + 2n1 + 1

n0 = n1 + 1

Proved.

Q.1. (*e*) Prove that the sum of the degree of the vertices of an undirected graph is twice the number of edges.

Ans. Proof: Since the degree of a vertex is the number of edges incident with that vertex, the sum of the degree counts the total number of times an edge is incident with a vertex. Since every edge is incident with exactly two vertices, each edge gets counted twice, once at each end. Thus the sum of the degrees equal twice the number of edges.

The sum of all the vertex-degrees is equal to twice the number of edges, since each edge contributes exactly 2 to the sum. It is thus an even number. The handshaking lemma is so called because it tells us that if several people shake hands, then the total number of hands shaken must be even – this is precisely because just two hands are involved in each handshake. A useful corollary of the handshaking lemma is the following:

In any graph the number of vertices of odd degree is even.

Q.2. (*a*) Write a C function to evaluate a postfix expression. **(6.5)**

Ans. C Function to Evaluate the Postfix Expression

```
#define SIZE 50          /* Size of Stack */
#include <ctype.h>
int s[SIZE];
int top=-1;      /* Global declarations */
push(int elem)
{                /* Function for PUSH operation */
 s[++top]=elem;
}
int pop()
{                /* Function for POP operation */
 return(s[top—]);
}
main()
{                /* Main Program */
 char pofx[50],ch;
 int i=0,op1,op2;
 printf("\n\nRead the Postfix Expression ? ");
 scanf("%s",pofx);
 while( (ch=pofx[i++]) != '\0')
 {
 if(isdigit(ch)) push(ch-'0'); /* Push the operand */
 else
 {     /* Operator,pop two  operands */
  op2=pop();
  op1=pop();
  switch(ch)
  {
  case '+':push(op1+op2);break;
  case '-':push(op1-op2);break;
  case '*':push(op1*op2);break;
  case '/':push(op1/op2);break;
  }
 }
 }
 printf("\n Given Postfix Expn: %s\n",pofx);
 printf("\n Result after Evaluation: %d\n",s[top]);
}
```

Q.2. (*b*) What do you mean by a sparse matrix? Write procedure to add two sparse matrices. **(6)**

Ans. Sparse Matrix: A matrix in which number of zero entries are much higher than the number of non zero entries is called sparse matrix. The natural method of representing matrices in memory as two-dimensional arrays may not be suitable foe

sparse matrices. One may save space by storing for only non zero entries. For example matrix A (4*4 matrix) represented below where first row represent the dimension of matrix and last column tells the number of non zero values; second row onwards it is giving the position and value of non zero number.

Code:

```
0 0 0 15
0 0 0 0
0 9 0 0
0 0 4 0
```

Here the memory required is 16 elements × 2 bytes = 32 bytes

The above matrix can be written in sparse matrix form as follows:

Code:

```
4 4 3
0 3 15
2 1 9
3 2 4
```

Here the memory required is 12elements × 2 bytes = 24 bytes

Procedure to add two sparse Matrix :

```
Algorithm Sparse_Matrix_Addition(A, B, AROW, BROW, ACOL, BCOL, m, n)
* m,n are number of rows and columns in the matrix
* A is the array containing non-zero elements of first matrix
* B is the array containing non-zero elements of second matrix
* C is the array containing non-zero elements of resultant matrix
* AROW is the row array of first sparse matrix
* BROW is the row array of second sparse matrix
* CROW is the row array of resultant sparse matrix
* ACOL is the column array of first sparse matrix
* BCOL is the column array of second sparse matrix
* CCOL is the column array of resultant sparse matrix
Begin
n = 1
ia = ib = ic = ja = jb = jc = 1
Repeat while (ia <= m)
        if(AROW[ia] == 0 AND BROW[ib] ==0)
                CROW[ic++] = 0
        end if
        else if(AROW[ia] == 0)
                BMAX = BROW[ib++] – 1
                CROW[ic++] = n
                Repeat while(jb <= BMAX)
                        CCOL[jc] = BCOL[jb]
                        C[jc++] = B[jb++]
                        n++
                end Repeat
        end else if
        else if(BROW[ib] == 0)
                AMAX = AROW[ia++] – 1
                CROW[ic++] = n
                Repeat while(ja <= AMAX)
                        CCOL[jc] = ACOL[jb]
                        C[jc++] = A[ja++]
                        n++
                end Repeat
end else if
else
        AMAX = AROW[ia++] – 1
        BMAX = BROW[ib++] – 1
        Repeat while(ja <= AMAX and jb <= BMAX)
                CROW[ic++] = n
                if(ACOL[ja] == BCOL[jb])
                        C[jc] = A[ja] + B[jb++]
                        CCOL[jc++] = ACOL[ja++]
                        n++
                end if
                else if(ACOL[ja] < BCOL[jb])
                        C[jc] = A[ja]
                        CCOL[jc++] = ACOL[ja++]
                        n++
                end else if
                else
                        C[jc] = B[jb]
                        CCOL[jc++] = BCOL[jb++]
                        n++
                end else if
        end Repeat
        Repeat while (ja <=AMAX)
                C[jc] = A[ja]
                CCOL[jc++] = ACOL[ja++]
                n++
        end Repeat
        Repeat while (jb <=BMAX)
                C[jc] = B[jb]
                CCOL[jc++] = BCOL[jb++]
                n++
        end Repeat
end else
ia++
ib++
end Repeat
end Sparse_Matrix_Addition
```


Q.3. (*a*) Write C functions to delete and add a node in a doubly linked list. (6.5)

Ans. Insert a Node at begin in Doubly linked list :

```
void insert_begning(int value)
{
  struct node *var,*temp;
  var=(struct node *)malloc(sizeof(struct node));
  var->data=value;
  if(head==NULL)
```

```
        {
          head=var;
          head->previous=NULL;
          head->next=NULL;
          last=head;
        }
        else
        {
          temp=var;
          temp->previous=NULL;
          temp->next=head;
          head->previous=temp;
          head=temp;
        }
    }
```

Delete a Node from end in Doubly linked list :

```
int delete_from_end()
{
    struct node *temp;
    temp=last;
    if(temp->previous==NULL)
    {
      free(temp);
      head=NULL;
      last=NULL;
      return 0;
    }
    printf("\nData deleted from list is %d \n",last->data);
    last=temp->previous;
    last->next=NULL;
    free(temp);
    return 0;
}
```

Q.3.(*b*) How is a circular queue advantageous over a queue linear? Explain the addition and deletion operations in it. (6)

Ans. Circular Queue (CQ) advantage over the Queue

1. In CQ we utilize memory efficiently. because in queue when we delete any element only front increment by 1, but that position is not used later. so when we perform more add and delete operation, memory wastage increase. But in CQ memory is utilized, if we delete any element that position is used later, because it is circular.

2. Circular queue have less memory consumption as compared to linear queue because while doing insertion after deletion operation it allocate an extra space the first remaining vacant but in circular queue the first is used as it comes immediate after the last.

3. In circular queue the memory of the deleted process can be used by some other new process..

Algorithm for Insert Operation in circular Queue

Step 1: If REAR = SIZE-1 then
REAR = 0
Else

REAR=REAR + 1
Step 2: If FRONT = REAR then
Write ("Circular Queue Overflow")
Step 3: CQ[REAR]=X
Step 4: If FRONT = -1 then
FRONT=0

Algorithm for Delete Operation in circular Queue

Step 1: If FRONT = -1 then
Write ("Circular Queue Underflow")
Step 2: Return (CQ [FRONT])
Step 3: If FRONT = REAR then
FRONT=REAR=-1
Step 4: If FRONT = SIZE-1 then
FRONT=0
Else
FRONT=FRONT+1

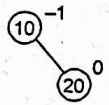
Q.4. (*a*) How AVL tree is useful over a binary search tree? Insert following nodes with key values 10,20,30,42,28,29,50 in the order of their existence. (6.5)

Ans. BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST. In BST, the time complexity of search operation (average case) is taken to be O(log n). But in the worst case, i.e the degenerate trees/ skewed trees time complexity of search operation is O(n) which can also be attained using array or linked list. So, in order to remove this problem, balanced binary search tree (AVL tree) was introduced in which the time complexity of search operation remains O(log n).

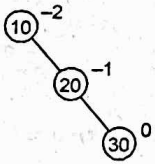
Creation of AVL tree

Insert node 10

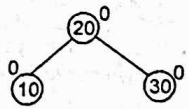
Insert node 20



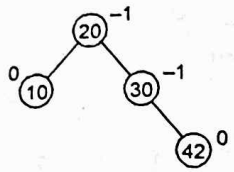
Insert node 30



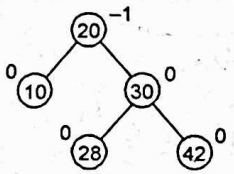
Now the tree is imbalanced so after the LL rotation the new balanced tree is:



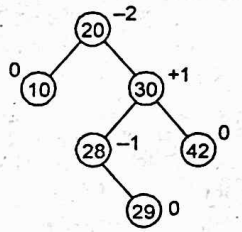
Insert node 42



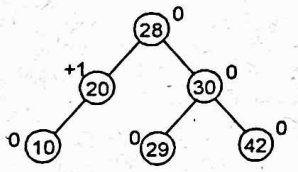
Insert node 28



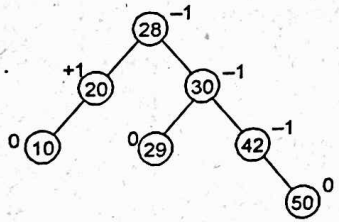
Insert node 29



Now the tree is imbalanced so after te LR rotation the new balanced tree is:



insert node 50



This is the final AVL Tree.

Q.4. (*b*) What is a threaded binary tree? Write a C function to delete a node from this tree. (6)

Ans. What is Threaded Binary Tree:

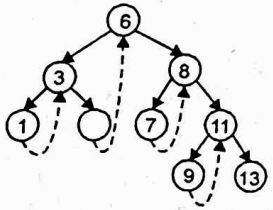
A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.

- We have the pointers reference the next node in an inorder traversal; called threads

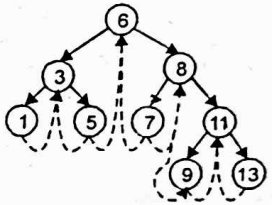
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer.

Types of threaded binary trees:

- **Single Threaded:** each node is threaded towards either the inorder predecessor or successor (left orright) means all right null pointers will point to inorder successor **OR** all left null pointers will point to inorder predecessor.
- **Double threaded:** each node is threaded towards both the inorder predecessor and successor (left **and**right) means all right null pointers will point to inorder successor **AND** all left null pointers will point to inorder predecessor.



Single Threaded Binary Tree



Double Threaded Binary Tree

C Program to delete a Node from Threaded Binary Tree :

```
void del(node *root,node *dummy,int key)
{
 node *temp,*parent,*temp_succ;
 node *search(node *,node *,int,node **);
 int flag=0;
 temp=search(root,dummy,key,&parent);
 if(root==temp)
 {
  cout<<"\n Its Root Node Which Can Not Be Deleted!!";
  return;
 }
 //deleting a node with two children
 if(temp->lth==1&&temp->rth==1)
 {
 parent=temp;
 temp_succ=temp->right;//Finding Inorder successor
 while(temp_succ->lth==1)
 {
        flag=1;
        parent=temp_succ;
        temp_succ=temp_succ->left;
 }
 if(flag==0)
 {
        temp->data=temp_succ->data;
       parent->right=temp_succ->right;
       parent->rth=0;
 }
else//inorder successor is on left subbranch.
              // and it has to be traversed
```

```
    {
        temp->data=temp_succ->data;
        parent->rth=0;
        parent->lth=0;
        parent->left=temp_succ->left;
    }
    cout<<" Now Deleted it!";
    return;
    }
//deleting a node having only one child
//The node to be deleted has left child
if(temp->lth==1 &&temp->rth==0)
{
if(parent->left==temp)
{
    (temp->left)->right=parent;
    parent->left=temp->left;
}
else
{
    (temp->left)->right=temp->right;
    parent->right=temp->left;
}
temp=NULL;
delete temp;
cout<<" Now Deleted it!";
return;
}
//The node to be deleted has right child
if(temp->lth==0 &&temp->rth!=0)
{
if(parent->left==temp)
{
    parent->left=temp->right;
    (temp->right)->left=temp->left;
    (temp->right)->right=parent;
}
else
{
    parent->right=temp->right;
    (temp->right)->left=parent;
}
temp=NULL;
delete temp;
cout<<" Now Deleted it!";
return;
}
//deleting a node which is having no child
if(temp->lth==0 &&temp->rth==0)
{
    if(parent->left==temp)
```

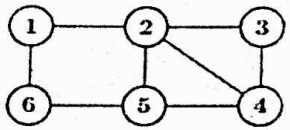


```
    {
      parent->left=temp->left;
      parent->lth=0;
    }
    else
    {
      parent->right=temp->right;
      parent->rth=0;
    }
  cout<<" Now Deleted it!";
  return;
  }
}
```

Q.5. Explain various techniques of a graph representation. Describe Floyd Warshall's algorithm with suitable example. **(12.5)**

Graph Representation in Memory

Ans. There are several possibilities to represent a graph $G = (V,E)$ in memory. Let the set of nodes be $V = \{1,2,\ldots,n)$ with edges $E \subseteq V \times V$, and let $m := |E|$.



1. Adjacency matrix

The adjacency matrix of a graph $G = (V, E)$ is a $|V| \times |V|$ matrix A, where each entry a_{ij} is equal to 1 if there exists an edge $e = (v_i, v_j) \varepsilon E$ and 0 otherwise. In case a weight function $w : E \to R$ is given, then $a_{ij} = w(v_i, v_j)$.

The adjacency matrix of the graph depicted above is

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Note that , since the graph in the example is undirected, the resulting matrix is symmetric. Also since there are no self loops, each entry in the main diagonal is zero.

2. Edge List

In an edge List L, every edge e is stored as a tuple (v_i, v_j) in a sorted list. The edge list of the graph depicted above is:

$$L = \{(1,2), (1,6), (2,3), (2,4), (2,5), (3,4), (4,5) (5,6)\}$$

While being a space-efficient way to store the graph, the access times of an edge list are slower than the access times of an adjacency matrix (for most edges).

3. Adjacency List

An adjacency list is similar to an edge list: for each node a sorted list of adjacement nodes (also termed neighbors) is stored Likewise, this is an efficient way to represent a graph but the access times are slower compared to a matrix (O(lg n) compared to O(1)). One way to solve this issue is to use a hash table for each node, containing all its neighbors.

The adjacency list for the example graph is:

Node	Neighbors
1	{2, 6}
2	{1, 3, 4, 5}
3	{2, 4}
4	{2, 3, 5}
5	{2, 4, 6}
6	{1, 5}

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm works based on a property of *intermediate* vertices of a shortest path. An *intermediate* vertex for a path $p = \langle v_1, v_2, \ldots, v_i \rangle$ is any vertex other than v_1 or v_j.

If the vertices of a graph G are indexed by $\{1, 2, \ldots, n\}$, then consider a subset of vertices $\{1, 2, \ldots, k\}$. Assume p is a minimum weight path from vertex i to vertex j whose intermediate vertices are drawn from the subset $\{1, 2, \ldots, k\}$. If we consider vertex k on the path then either:

k is **not** an intermediate vertex of p (i.e. is not used in the minimum weight path)

$\Rightarrow$ all intermediate vertices are in $\{1, 2, \ldots, k\text{-}1\}$

k is an intermediate vertex of p (i.e. is used in the minimum weight path)

$\Rightarrow$ we can divide p at k giving two subpaths p_1 and p_2 giving $v_i \rightarrow k \rightarrow v_j$

$\Rightarrow$ by Lemma 24.1 (subpaths of shortest paths are also shortest paths) subpaths p_1 and p_2 are shortest paths with intermediate vertices in $\{1, 2, \ldots, k\text{-}1\}$

Thus if we define a quantity $d^{(k)}_{ij}$ as the minimum weight of the path from vertex i to vertex j with intermediate vertices drawn from the set $\{1, 2, \ldots, k\}$ the above properties give the following recursive solution

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

Thus we can represent the optimal values (when $k = n$) in a matrix as

$$D^{(n)} = \left\{d_{ij}^{(n)}\right\} = \{\delta(i, j)\}$$

Algorithm

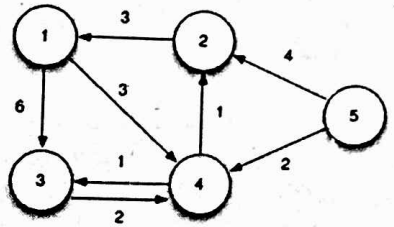
FLOYD-WARSHALL(W)

1. n = W.rows
2. $D^{(0)} = W$
3. $\Pi^{(0)} = \pi^{(0)}_{ij} = \text{NIL}$ if $i=j$ or $w_{ij} = \infty$
   $= i$ if $i \neq j$ and $w_{ij} < \infty$
4. for $k = 1$ to n
5. let $D^{(k)} = (d^{(k)}_{ij})$ be a new $n \times n$ matrix
6. for $i = 1$ to n
7. for $j = 1$ to n
8. $d^{k}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj})$
9. if $d^{(k-1)}_{ij} \leq d\, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}$
10. $\pi^{(k)}_{ij} = \pi^{(k-1)}_{ij}$
11. else
12. $\pi^{(k)}_{ij} = \pi^{(k-1)}_{kj}$
13. return $D^{(n)}$

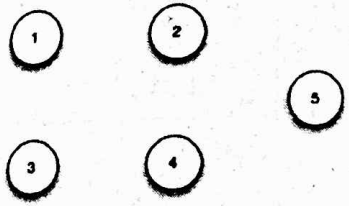
Basically the algorithm works by repeatedly exploring paths between every pair using each vertex as an intermediate vertex.

Since Floyd-Warshall is simply three (tight) nested loops, the run time is clearly $O(V^3)$.

Example



Initialization: ($k = 0$)



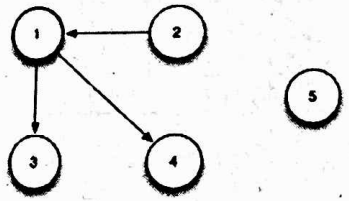
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	∞	∞	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	/	/	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Iteration 1: ($k = 1$) Shorter paths from $2 \rightarrow 3$ and $2 \rightarrow 4$ are found through vertex 1



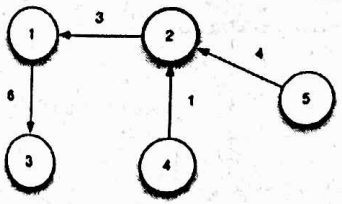
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

Iteration 2: ($k = 2$) Shorter paths from $4 \rightarrow 1$, $5 \rightarrow 1$, and $5 \rightarrow 3$ are found through vertex 2



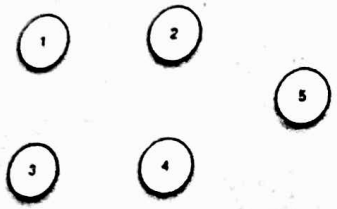
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	4	1	1	0	∞
5	7	4	13	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	2	4	4	/	/
5	2	5	2	5	/

Iteration 3: ($k = 3$) No shorter paths are found through vertex 3



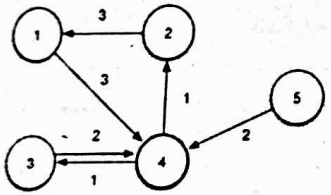
D

	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	4	1	1	0	∞
5	7	4	13	2	0

Π

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	2	4	4	/	/
5	2	5	2	5	/

Iteration 4: ($k = 4$) Shorter paths from $1 \to 2$, $1 \to 3$, $2 \to 3$, $3 \to 1$, $3 \to 2$, $5 \to 1$, $5 \to 2$, $5 \to 3$, and $5 \to 4$ are found through vertex 4



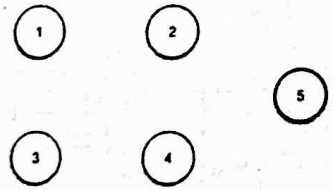
D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

Iteration 5: ($k = 5$) No shorter paths are found through vertex 5



D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

The final shortest paths for all pairs is given by

D

	1	2	3	4	5
1	0	4	4	3	∞
2	3	0	7	6	∞
3	6	3	0	2	∞
4	4	1	1	0	∞
5	6	3	3	2	0

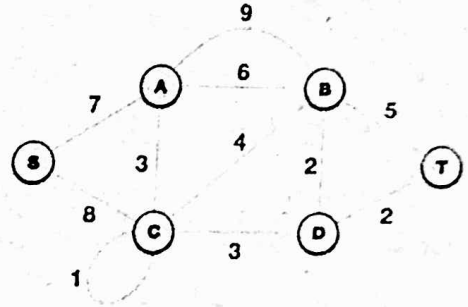
Π

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

Q.6. (a) Explain Kruskal's algorithm of minimum spanning tree with suitable example. **(6.5)**

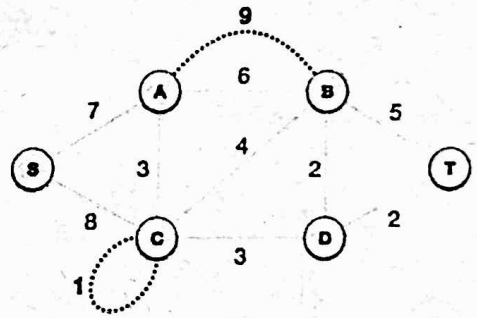
Ans. Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example "

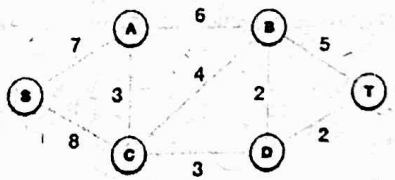


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



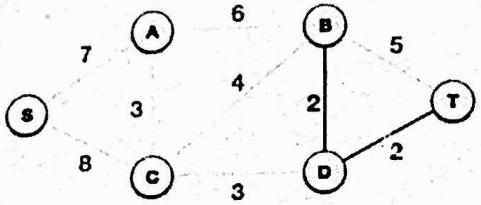
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B,D 2	D,T 2	A,C 3	C,D 3	C,B 4	B,T 5	A,B 6	S,A 7	S,C 8

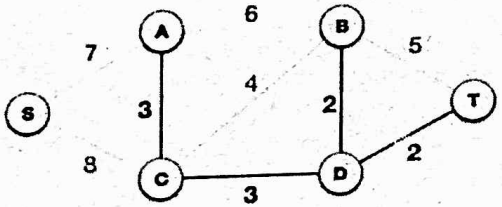
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the lea weight. Throughout, we shall keep checking that the spanning properties remain inta In case, by adding one edge, the spanning tree property does not hold then we sha consider not to include the edge in the graph.

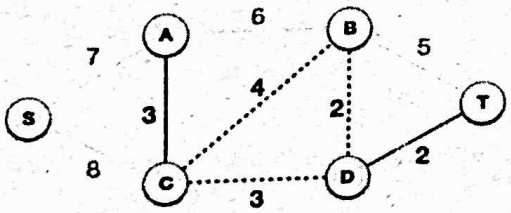


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding the does not violate spanning tree properties, so we continue to our next edge selection.

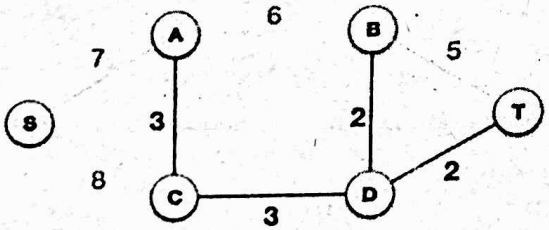
Next cost is 3, and associated edges are A,C and C,D. We add them again.



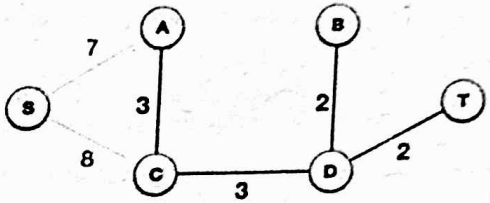
Next cost in the table is 4, and we observe that adding it will create a circuit in th graph.



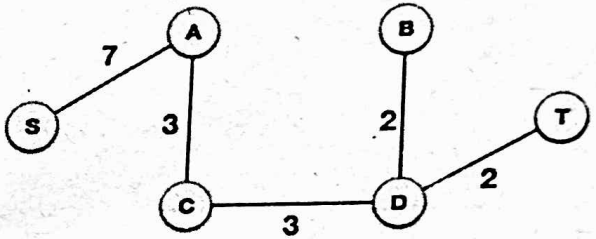
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Q.6. (*b*) What do you mean by a binary search tree? Write procedures for various traversals of this tree. **(6)**

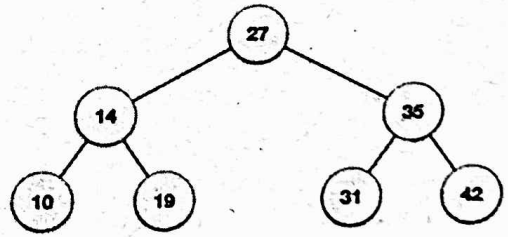
Ans. A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

Following is a pictorial representation of BST "



Binary search tree Traversal : Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree "

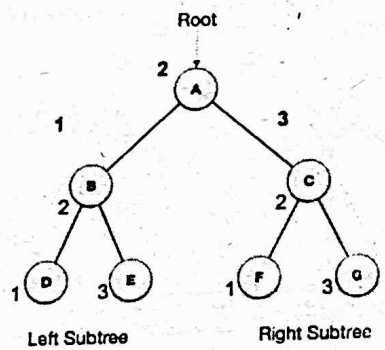
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

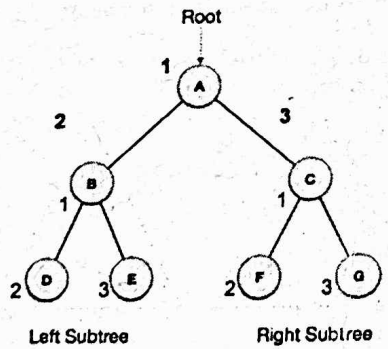


We start from **A**, and following in-order traversal, we move to its left subtree **B**.**B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be "

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

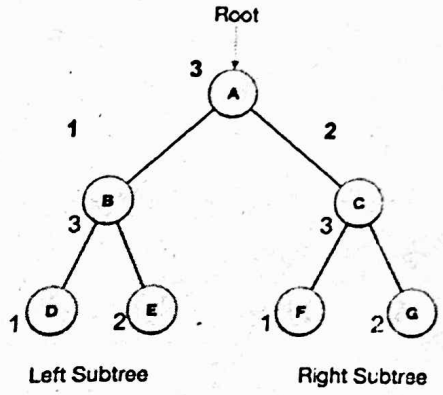


We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be—

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree**B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be "

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Q.7. (*a*) Explain with suitable example the radix sort and compute its time complexity. **(6.5)**

Ans. Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. (Here Radix is 26, 26 letters of alphabet). Specifically, the list of names is first sorted according to the first letter of each names, that is, the names are arranged in 26 classes. Intuitively, one might want to sort numbers on their most significant digit. But Radix sort do counter-intuitively by sorting on the least significant digits first. On the first pass entire numbers sort on the least significant digit and combine in a array. Then on the second pass, the entire numbers are sorted again on the second least-significant digits and combine in a array and so on.

Following example shows how Radix sort operates on seven 3-digits number.

INPUT	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

In the above example, the first column is the input. The remaining shows the list after successive sorts on increasingly significant digits position.

Time complexity of Radix Sort :

The algorithm takes O(n) time per bucket sort.

There are log_10 k = O(log n) bucket sorts.

So the total time is O(n log k).

Q.7. (b) What do you mean by sequential file organization? Explain its advantages and disadvantages. **(6)**

Ans. Sequential File Organization

1. A sequential file is designed for efficient processing of records in sorted order on some search key.
   - Records are chained together by pointers to permit fast retrieval in search key order.
   - Pointer points to next record in order.
   - Records are stored physically in search key order (or as close to this as possible).
   - This minimizes number of block accesses.
   - Example, with *bname* as the search key.
2. It is difficult to maintain physical sequential order as records are inserted and deleted.
   - Deletion can be managed with the pointer chains.
   - Insertion poses problems if no space where new record should go.
   - If space, use it, else put new record in an overflow block.
   - Adjust pointers accordingly.
   - The previous example after an insertion.
   - Problem: we now have some records out of physical sequential order.
   - If very few records in overflow blocks, this will work well.
   - If order is lost, reorganize the file.
   - Reorganizations are expensive and done when system load is low.
3. If insertions rarely occur, we could keep the file in physically sorted order and reorganize when insertion occurs. In this case, the pointer fields are no longer required.

Advantages:

1. Simple to understand
2. Easier to organize, maintain
3. Economical
4. Error in files remain localized

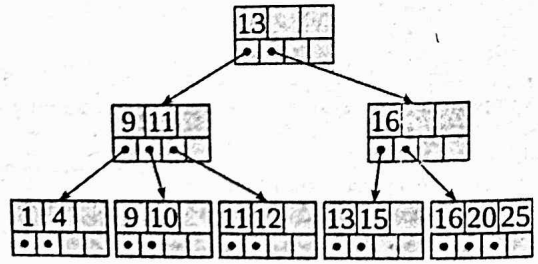
Disadvantages:

1. Entire file has to be processed
2. Transactions must be sorted in a particular sequence before processing
3. Time consuming searching
4. High data redundancy
5. Random enquiries are not possible to handle

Q.8. (a) What is B+ tree indexing? Explain with suitable example. **(6.5)**

The B+-tree, in which each node stores up to d references to children and up to $d - 1$ keys. Each reference is considered "between" two of the node's keys; it references the root of a subtree for which all values are between these two keys.

Here is a fairly small tree using 4 as our value for d.



A B+-tree requires that each leaf be the same distance from the root, as in this picture, where searching for any of the 11 values (all listed on the bottom level) will involve loading three nodes from the disk (the root block, a second-level block, and a leaf).

In practice, d will be larger — as large, in fact, as it takes to fill a disk block. Suppose a block is 4KB, our keys are 4-byte integers, and each reference is a 6-byte file offset. Then we'd choose d to be the largest value so that $4(d - 1) + 6d \le 4096$; solving this inequality for d, we end up with $d \le 410$, so we'd use 410 for d. As you can see, d can be large.

A B+-tree maintains the following invariants:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node N with k being the number of keys in N: all keys in the first child's subtree are less than N's first key; and all keys in the ith child's subtree $(2 \le i \le k)$ are between the $(i - 1)$th key of n and the ith key of n.
- The root has at least two children.
- Every non-leaf, non-root node has at least *floor*$(d / 2)$ children.
- Each leaf contains at least *floor*$(d / 2)$ keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.

Q.8. (b) Explain different collision resolution techniques in hashing. **(6)**

Ans. Open Addressing

In Open Addressing all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. (This method is also called **closed hashing**). Well-known probe sequences include:

- Linear probing, in which the interval between probes is fixed (usually 1)
- Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- Double hashing, in which the interval between probes is computed by a second hash function

Linear Probe

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Add the keys 10, 5, and 15 to the previous table.

Hash key = key % table size

2 = 10 % 8

5 = 5 % 8

7 = 15 % 8

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

When using a linear probe, the item will be stored in the next available slot in the table, assuming that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.

If an empty slot is not found before reaching the point of collision, the table is full.

A problem with the linear probe method is that it is possible for blocks of data to form when collisions are resolved. This is known as primary clustering.

This means that any key that hashes into the cluster will require several attempts to resolve the collision.

Quadratic Probe

To resolve the primary clustering problem, quadratic probing can be used. With quadratic probing, rather than always moving one spot, move i^2 spots from the point of collision, where i is the number of attempts to resolve the collision.

89 % 10 = 9

18 % 10 = 8

49 % 10 = 9 – 1 attempt needed – 1^2 = 1 spot

58 % 10 = 8 – 3 attempts – 3^2 = 9 spots

69 % 10 = 9 – 2 attempts – 2^2 = 4 spots

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Limitation: At most half of the table can be used as alternative locations to resolve collisions.

This means that once the table is more than half full, it's difficult to find an empty spot. This new problem is known as secondary clustering because elements that hash to the same hash key will always probe the same alternative cells.

END TERM EXAMINATION [MAY-JUNE 2017]
SECOND SEMESTER [MCA]
DATA AND FILE STRUCTURE [MCA–102]

Time : 3 hrs. **M.M. : 75**

Note: Attempt any five questions including Q. No. 1 which is compulsory.

Q.1. (a) Explain various asymptotic notations for representing time complexity. **(2.5)**

Ans. Asymptotic Notations

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

(1) Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

For example, for a function $f(n)$

$O(f(n)) = \{g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \le c.g(n) \text{ for all } n > n_0.\}$

(2) Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

For example, for a function $f(n)$

$\Omega(f(n)) \ge \{g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \le c.f(n) \text{ for all } n > n_0.\}$

(3) Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows–

$\theta(f(n)) = \{g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0.\}$

Q.1.(b) Differentiate between B+ Tree and B* Tree. **(2.5)**

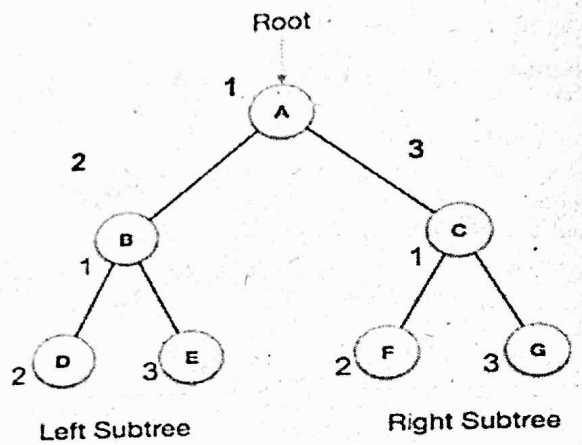
Ans. B+ trees are very common. The major difference between B+ and B* trees is that B+ trees are split into to different child nodes when they are full which means that number of data points in each node in B+ tree is 1/2 of it's maximum capacity. In B* tree the number of data points in each node is at least 2 / 3. Thus the B* trees are more compact compared to B+ trees. This is typically achieved by sharing data points between sibling or neighbouring nodes. When a single node is full it shares data points and only when both the neighbour nodes are full it splits.

However this comes at a cost. Deleting a data point in B* tree is much more complicated than other trees. This is a valid tradeoff because in many use cases the frequency of delete operation is much less.

Q.1.(c) Explain preorder traversal of a binary tree. (2.5)

Ans. Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

Q.1.(d) What do you mean by AVL tree? How the balance factor of a node in an AVL tree is computed. (2.5)

Ans. AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the hieghts of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as balance factor.

Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we are calculating as follows...

Balance factor = height Of Left Subtree - height Of Right Subtree

Q.1. (e) Explain shell short using suitable example. (2.5)

Ans. Shell Sort is a generalized version of insertion sort. It is an in-place comparison sort. Shell Sort is also known as **diminishing increment sort.** This algorithm uses insertion sort on the large interval of elements to sort. Then the interval of sorting keeps on decreasing in a sequence until the interval reaches 1. These intervals are known as **gap sequence.**

This algorithm works quite efficiently for small and medium size array as its average time complexity is near to O(n).

Here are some key points of shell sort algorithm –

- Shell Sort is a comparison based sorting.
- The best case in shell sort is when the array is already sorted. The number of comparisons is less.
- It is an in-place sorting algorithm as it requires no additional scratch space.
- Shell Sort is unstable sort as relative order of elements with equal values may change.
- It is been observed that shell sort is 5 times faster than bubble sort and twice faster than insertion sort its closest competitor.

Q.1.(f) Write any two disadvantages of sequential file allocation method. (2.5)

Ans. Disadvantages of sequential file

Sequential file is time consuming process.

It has high data redundancy.

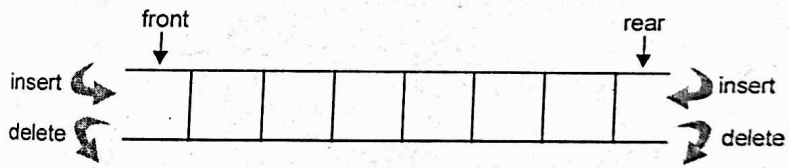
Random searching is not possible.

Q.1.(g) What is dequeue? (2.5)

Ans. DEQUEUE

Double Ended Queue (Dequeue)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



Double Ended Queue can be represented in TWO ways, those are as follows...

Input Restricted Double Ended Queue

Output Restricted Double Ended Queue

Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.

Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.

Q.1.(h) What do you mean by a heap? How is it different from a binary search tree? (2.5)

Ans. A **heap** is a specialized tree-based data structure that satisfies the *heap property:* if P is a parent node of C, then the *key* (the *value*) of P is either greater than or

equal to (*in a max heap*) or less than or equal to (*in a min heap*) the key of C. The node at the "top" of the heap (with no parents) is called the *root* node.

Difference between BST & Heap :

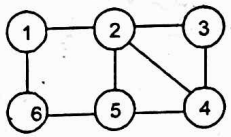
(1) Heap just guarantees that elements on higher levels are greater (for max-heap) or smaller (for min-heap) than elements on lower levels, whereas BST guarantees order (from "left" to "right"). If you want sorted elements, go with BST.

(2) Heap is better at findMin/findMax (O(1)), while BST is good at all finds (O(logN)). Insert is O(logN) for both structures. If you only care about findMin/findMax (e.g. priority-related), go with heap. If you want everything sorted, go with BST.

Q.1.(*i*) Explain adjacency list representation of a graph. **(2.5)**

Ans. Graph Representation in memory

There are several possibilities to represent a graph G = (V.E) in memory. Let the set of nodes be V = (1, 2, ...n) with edges $E \subseteq V \times V$, and let m = |E|.



Adjacency List

An adjacency list is similar to an edge list: for each node a sorted list of adjacent nodes (also termed *neighbors)* is stored. Likewise, this is an efficient way to represent a graph but the access times are slower compared to a matrix (O(lgn) compared to *O*(1)). One way to solve this issue is to use a hash table for each node, containing all its neighbors.

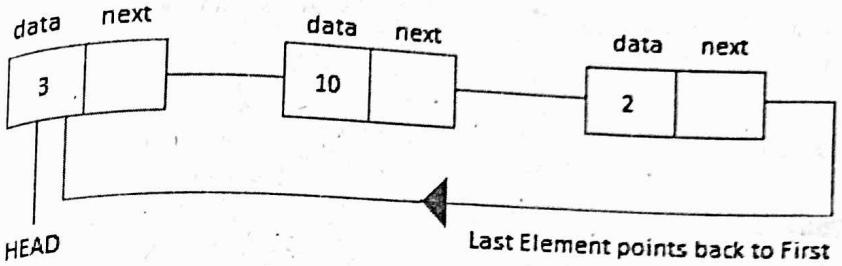
The adjacency list for the example graph is:

Node	Neighbours
1	{2, 6}
2	{1, 3, 4, 5}
3	{2, 4}
4	{2, 3, 5}
5	{2, 4, 6}
6	{1, 5}

Q.1.(j) What is a circular linked list? Write its advantages over a linear linked list. **(2.5)**

Ans. Circular List

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



Circular lists are usually preferable to non-circular ones, for two reasons:

They provide easy access to both ends of a list: notice that we do not need the first pointer in the header, because we can easily reach the first node with L->last->next

The code for processing a circular list is often *simpler* than the code for processing a non-circular list - mainly because you don't have to constantly check if the *next* node is defined - it is always defined.

Q.2.(a) Write algorithms for insertion and deletion of an element in a linear queue. **(6.5)**

Ans. INSERT-ITEM(QUEUE,FRONT,REAR,MAX,ITEM)

This algorithm is used to add or insert item to QUEUE.

1. If (REAR = MAX) then
   a. Display "Queue overflow";
   b. Return;
2. Otherwise
   a. REAR := REAR + 1;
   b. QUEUE(REAR) := ITEM;
3. Return;

REMOVE-ITEM(QUEUE,FRONT,REAR,ITEM)

This algorithm is used to delete an item from QUEUE.

1. If (FRONT = REAR + 1) then
   a. Display "Queue underflow";
   b. Return;
2. Otherwise
   a. ITEM := QUEUE(FRONT);
   b. FRONT := FRONT + 1;
3. Return;

Q.2.(b) Convert the following infix expression into postfix expression. **(6)**

a/b – c + d * e – a * c.

Ans. Infix to postfix expression : $a / b - c + d * e - a * c$

S.No.	Symbol Scanned	Stack	Postfix Expression
1.	a		a
2.	/	/	a
3.	b	/	a , b
4.	-	-	a, b, /
5.	c	-	a, b, /, c
6.	+	+	a, b, /, c, -
7.	d	+	a, b, /, c, -, d
8.	*	+, *	a, b, /, c, -, d
9.	e	+, *	a, b, /, c, -, d, e
10.	-	-	a, b, /, c, -, d, e, *, +
11.	a	-	a, b, /, c, -, d, e, *, +, a
12.	*	-, *	a, b, /, c, -, d, e, *, +, a
13.	c	-, *	a, b, /, c, -, d, e, *, +, a, c
14			a, b, /, c, -, d, e, *, +, a, c, *, -

Q.3.(a) Write ac function that adds two polynomials which are passed as its arguments. **(6)**

Ans. //function to add polynomials

```
int add_addpolynomial( p1, p2, p3, max1, max2)
struct addpolynomial p1[], p2[], p3[];
int max1, max2;
{
 int i,j,k;
 i = j = k = 0;
 while ( i <max1 && j <max2)
 {
  if( p1[i].exp > p2[j].exp)
  {
   p3[k] = p1[i];
   k++;
   i++;
  }
  else
  if( p1[i].exp < p2[j].exp)
  {
   p3[k] = p2[j];
   k++;
   j++;
  }
  else
```



```
  {
   p3[k].exp = p1[i].exp;
   p3[k].coef = p1[i].coef + p2[j].coef;
   i++;
   j++;
   k++;
  }
 }
 while( i <max1 )
 {
  p3[k] = p1[i];
  k++;
  i++;
 }
 while( j <max2 )
 {
  p3[k] = p2[j];
  k++;
  j++;
 }
 return(k);
}
```

Q.3.(b) Prove that the maximum no. of nodes on level i of binary tree is 2^{i-1} for $i \geq 1$ and maximum no. of nodes in a binary tree of depth k in $2k-1$ for $k \geq 1$. **(6.5)**

Ans. Proof :

(1) The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.

(2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Proof:

(1) The proof is by induction on i.

Induction Base: The root is the only node on level $i = 1$. Hence, the maximum number of nodes on level i = 1 is $2^{i-1} = 2^0 = 1$.

Induction Hypothesis: Let i be an arbitrary positive integer greater than 1 Assume that the maximum number of nodes on level $i - 1$ is 2^{i-2}.

Induction Step: The maximum number of nodes on level i – 1 is 2^{i-2} by the hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level i – 1, or 2^{i-1}.

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=1}^{k} \text{(maximum number of nodes on level i)} = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

Q.4. (a) Explain the concept of threaded binary tree. Write algorithm for inserting a node in a threaded binary tree. (6.5)

Ans. Threaded Binary Tree

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node.

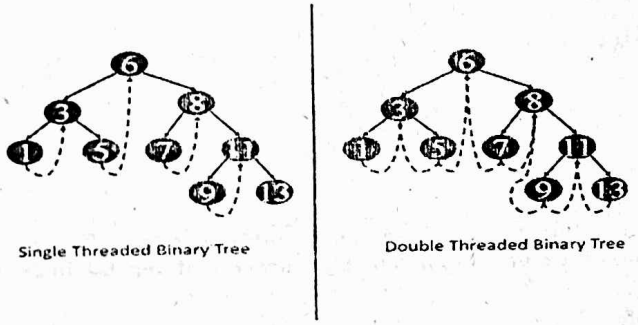
We have the pointers reference the next node in an inorder traversal; called threads.

We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer.

Types of threaded binary trees:

Single Threaded: each node is threaded towards either the in-order predecessor or successor (left **or**right) means all right null point-ers will point to inorder succes-sor **OR** all left null pointers will point to inorder predecessor.

Double threaded: each node is threaded towards both the inorder predecessor and successor (left **and**right) means all right null point-will point to inorder successor **AND** all left null pointers will point to inorder predecessor.



Single and Double threaded binary tree (1)

Insertion in Threaded Binary Tree :

Case 1: Insertion in empty tree

Both left and right pointers of tmp will be set to NULL and new node becomes the root.

```
root = tmp;
tmp -> left = NULL;
tmp -> right = NULL;
```

Case 2: When new node inserted as the left child

After inserting the node at its proper place we have to make its left and right threads points to inorder predecessor and successor respectively. The node which was inorder successor. So the left and right threads of the new node will be-

```
tmp -> left = par ->left;
tmp -> right = par;
```

Before insertion, the left pointer of parent was a thread, but after insertion it will be a link pointing to the new node.


```
par -> lthread = par ->left;
par -> left = temp;
```

Case 3: When new node is inserted as the right child

The parent of tmp is its inorder predecessor. The node which was inorder successor of the parent is now the inorder successor of this node tmp. So the left and right threads of the new node will be-

```
tmp -> left = par;
tmp -> right = par -> right;
```

Before insertion, the right pointer of parent was a thread, but after insertion it will be a link pointing to the new node.

```
par -> rthread = false;
par -> right = tmp;
```

Q.4. (b) Write algorithm for deletion of a node in a binary search tree. (6)

Ans. Binary Search Tree (Deletion)

(1) Node to be deleted is leaf: Simply remove from the tree.

```
      50                    50
     /  \     delete(20)   /  \
    30   70   ---------->  30   70
   / \  / \                \   / \ / \
  20 40 60 80               40 60 80
```

(2) Node to be deleted has only one child: Copy the child to the node and delete the child

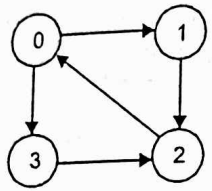
```
      50                    50
     /  \     delete(30)   /  \
    30   70   ---------->  40  70
     \  / \                / \
     40 60 80             60 80
```

(3) Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

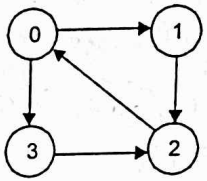
```
      50                    60
     /  \     delete(50)   /  \
    40   70   ---------->  40   70
       / \                    \
      60 80                   80
```

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Q.5.(a) What do you mean by strongly connected graph is the directed graph shown below strongly connected? Last all the simple paths in it. **(6.5)**



Ans. Strongly connected Graph: A Directed graph G is said to be strongly connected, if for each pair u,v of nodes in G there is a path from u to v and there is also a path from v to u. A directed graph is strongly connected if there is a path between any two pair of vertices.



In the Figure, this directed graph is strongly connected, because there is a path from one node to another node.

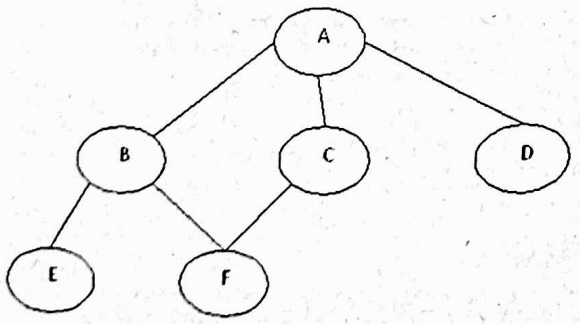
List of all simple paths in the above graph :

(0, 1), (0, 1, 2), (0, 3), (1, 2), (1, 2, 0), (2, 0), (2, 0, 3), (3, 2), (3, 2, 0).

Q.5.(b) Write an algorithm for breath first search of a graph. **(6)**

Ans. Breadth First Search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let's see how BFS traversal works with respect to the following graph:



If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. "A B C D E F". The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

Algorithmic Steps

1. **Step 1**: Push the root node in the Queue.
2. **Step 2**: Loop until the queue is empty.
3. **Step 3**: Remove the node from the Queue.
4. **Step 4**: If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

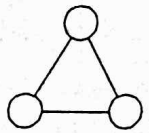
Q.6. (a) Prove that a complete graph with n vertices, has n^{n-2} spanning trees.

Ans. Proof: Let us consider a complete graph with 3 vertices

No of edges in complete graph with 3 vertices

$$= \frac{n(n-1)}{2} = \frac{3 \times (3-1)}{2} = \frac{3 \times 2}{2} = 3$$

Now we have the graph

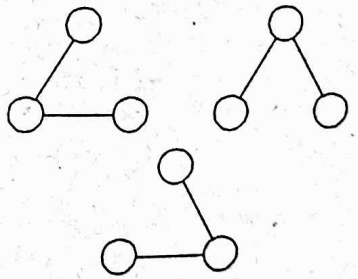


A spanning trees with 3 vertices has only 2 edges.

So, total no of spanning trees with 2 edges out of 3 edges.

$$= {}^3C_2 - \text{no of cycles with two edges}$$

$$= \frac{3!}{2!\,1!} - 0$$



3 Spanning trees

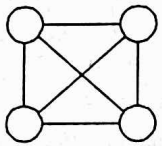
$$= 3 - 0$$

$$= 3$$

$$= 3^1 = 3^{3-2} \quad \text{(Here } n = 3\text{)}$$

If we have a complete graph with 4 vertices then

$$\text{No. of edges} = \frac{4(4-1)}{2} = \frac{4 \times 3}{2} = 6$$



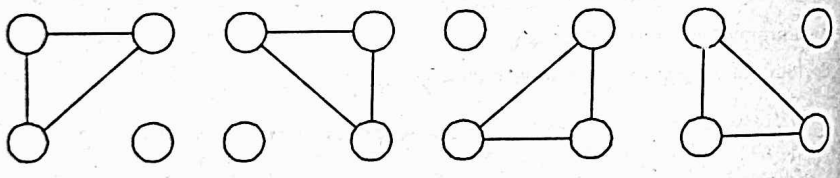
A spanning tree with 4 vertices has 3 edges

Total No. of spanning trees with 3 edges out of 6 edges

$$= {}^6C_3 - \text{no of cycles of length 3}$$

$$= \frac{6!}{3! \times 3!} - \text{no of cycles of length 3}$$

In a complete graph with 4 vertices, we have 4 cycles of length 3.



So,Total no spanning trees with 3 edges out of 6 edges

$$= \frac{6!}{3! \times 3!} - 4$$

$$= \frac{\cancel{6} \times 5 \times 4 \times \cancel{3!}}{\cancel{3} \times \cancel{2} \times 1 \times \cancel{3!}} - 4$$

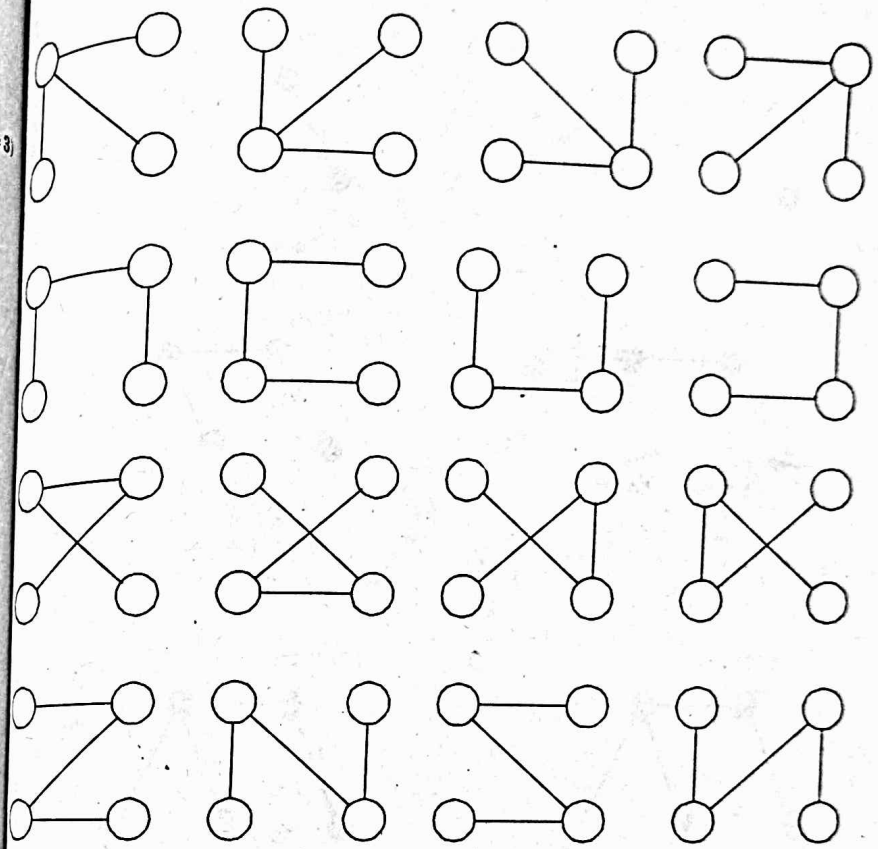
$$= 20 - 4$$

$$= 16$$

$$= 4^2$$

$$= 4^{4-2} \text{ (Here } n = 4\text{)}$$

In general we can say that a complete graph with n vertices has n^{n-2} spanning trees.



All 16 spanning trees of K_4.

Q.6.(b) Write Prim's algorithm for finding the minimum cost spanning tree. (6.5)

Ans. Steps of Prim's Algorithm:

The following are the main 3 steps of the Prim's Algorithm:

1. Begin with any vertex which you think would be suitable and add it to the tree.
2. Find an edge that connects any vertex in the tree to any vertex that is not in the tree. Note that, we don't have to form cycles.
3. Stop when n - 1 edges have been added to the tree.

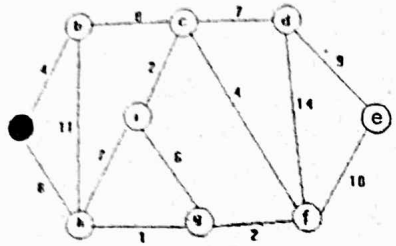
Solved Example

Question: Solve the following by applying Prim's Algorithm.

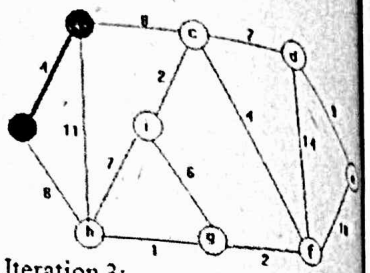
Solution:

By applying Prim's Algorithm, we will find the following iterations:

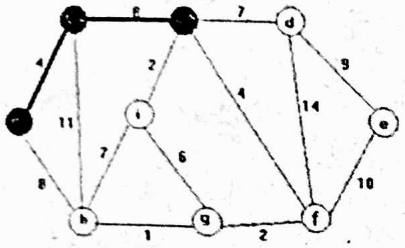
Iteration 0



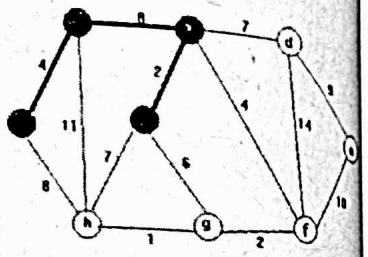
Iteration 1:



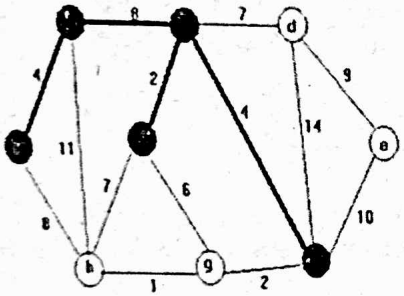
Iteration 2:



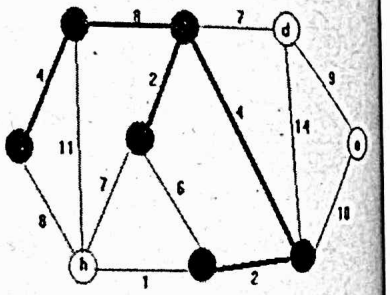
Iteration 3:



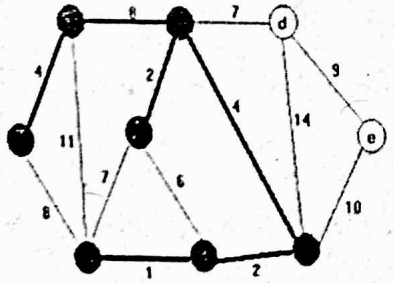
Iteration 4:



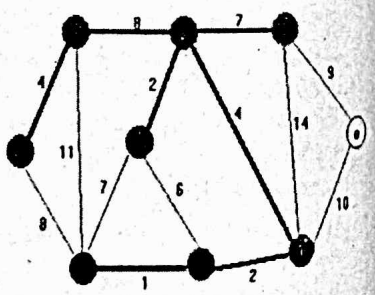
Iteration 5:



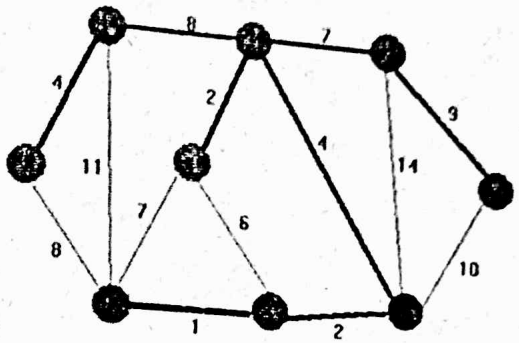
Iteration 6:



Iteration 7:



Iteration 8:



Here, the last iteration number 8 represents the optimal solution of the problem, given by the bold dark black lines, which gives us the minimum length of the path. And, the length would be calculated by adding the numbers written on optimal path. That is, 4 + 8 + 7 + 9 + 2 + 4 + 2 + 1 = 37 is the minimum length.

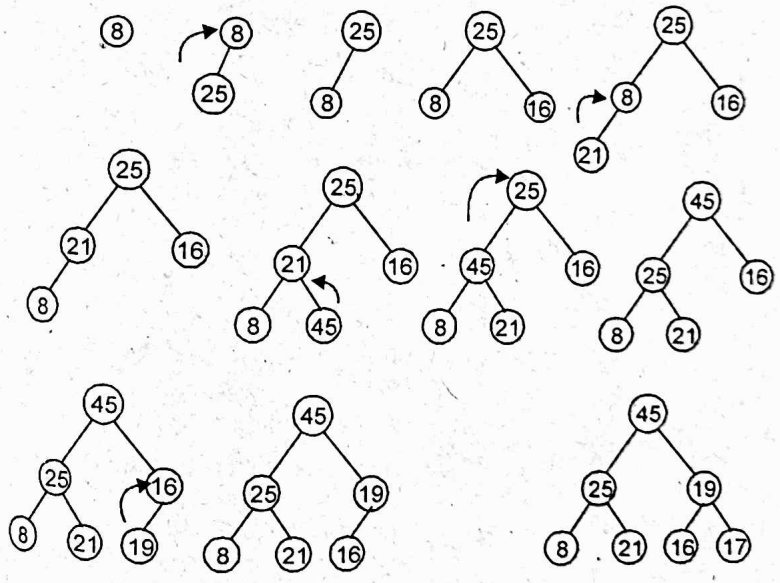
Q.7.(*a*) Short following integers using heap sort: **(6)**

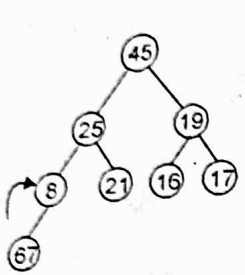
8, 25, 16, 21, 45, 19, 17, 67, 52, 69, 70.

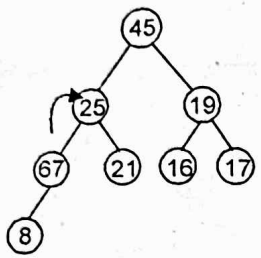
Ans. Heap Sort

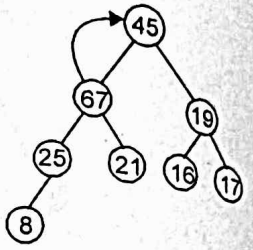
8, 25, 16, 21, 45, 19, 17, 67, 52, 69, 70. Fist we make a more heap

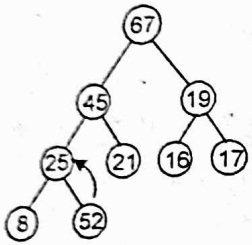
First we make a max heap

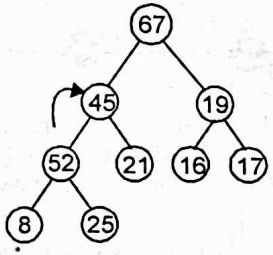


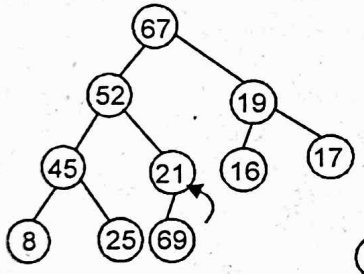


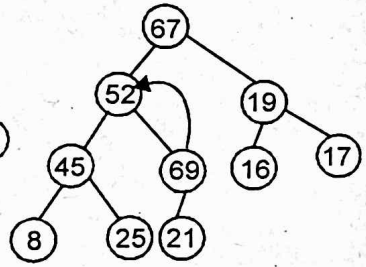


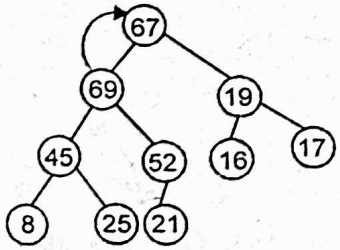


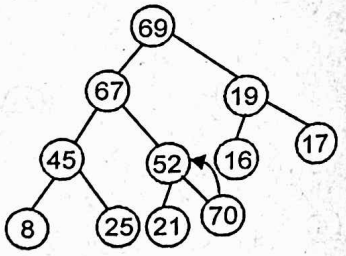


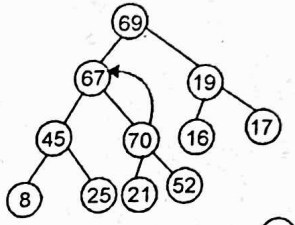


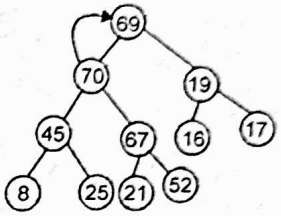


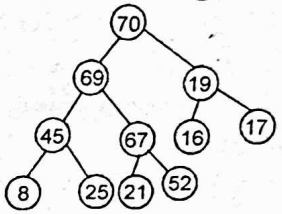










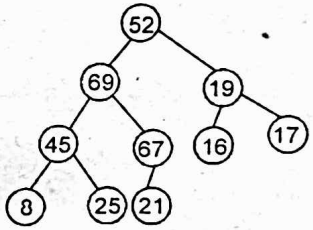


This is the final heap. Now we delete the root element and Replace it with the last element of Heap and store the root element at last position.

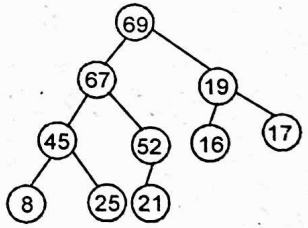
In array

A [10] = 70

The new Heap is



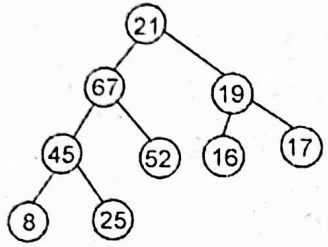
Again aften max heapify the new heap is–



Now again we delete the Root element and replace it with last element of Heap and store the Root Element at second last position of array.

a [9] = 69

The new Heap is



Now we again do the max Heapify the new Heap.

This process is Repeat until all the nodes are not deleted.

So the final elements after sorting are–

8	16	17	19	21	25	45	52	67	69	70

Q.7.(*b*) Explain various parity & error control techniques in files. **(6.5)**

Ans. Error Detecting Codes:

Whenever a message is transmitted, it may get scrambled by noise or data may get corrupted. To avoid this, we use error-detecting codes which are additional data added to a given digital message to help us detect if any error has occurred during transmission of the message.

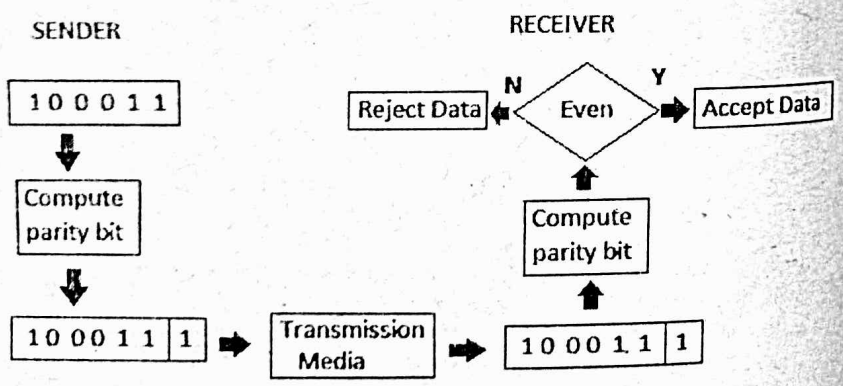
Some popular techniques for error detection are:

1. Simple Parity check

Blocks of data from the source are subjected to a check bit or parity bit generator form, where a parity of :

- 1 is added to the block if it contains odd number of 1's, and
- 0 is added if it contains even number of 1's

This scheme makes the total number of 1's even, that is why it is called even parity checking.



2. Two-dimensional Parity check

Parity check bits are calculated for each row, which is equivalent to a simple parity check bit. Parity check bits are also calculated for all columns, then both are sent along with the data. At the receiving end these are compared with the parity bits calculated on the received data.

Original Data

10011001	11100010	00100100	10000100

Row parities

10011001	0
11100010	0
00100100	0
10000100	0
11011011	0

Column parities

100110010	111000100	001001000	100001000	110110110

Data to be sent

3. Checksum

- In checksum error detection scheme, the data is divided into k segments each of m bits.

Original Data

10011001	11100010	00100100	10000100
1	2	3	4

k=4, m=8

Sender

```
1   10011001
2   11100010
   (1)01111011
              1
    01111100
3   00100100
    10100000
    10000100
   (1)00100100
              1
Sum: 00100101
CheckSum: 11011010
```

Reciever

```
1   10011001
2   11100010
   (1)01111011
              1
    01111100
3   00100100
    10100000
    10000100
   (1)00100100
              1
    00100101
    11011010
Sum: 11111111
Complement: 00000000
```

Conclusion: Accept Data

• In the sender's end the segments are added using 1's complement arithmetic to get the sum. The sum is complemented to get the checksum.

• The checksum segment is sent along with the data segments.

• At the receiver's end, all received segments are added using 1's complement arithmetic to get the sum. The sum is complemented.

• If the result is zero, the received data is accepted; otherwise discarded.

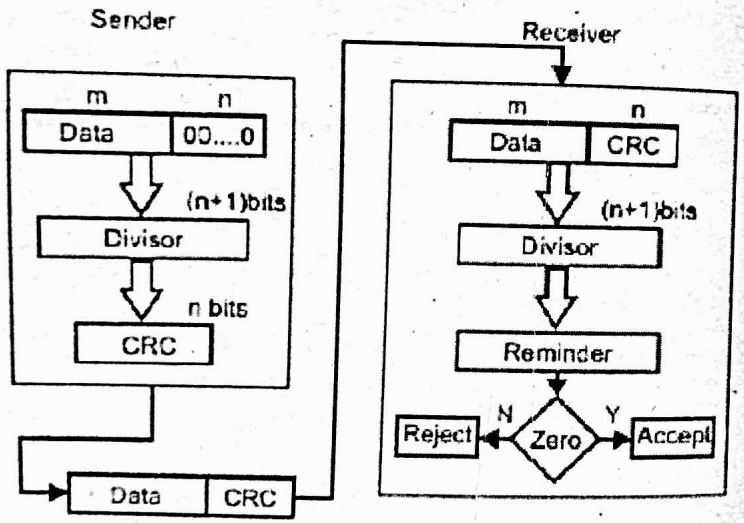
4. Cyclic redundancy check (CRC)

• Unlike checksum scheme, which is based on addition, CRC is based on binary division.

• In CRC, a sequence of redundant bits, called cyclic redundancy check bits, are appended to the end of data unit so that the resulting data unit becomes exactly divisible by a second, predetermined binary number.

• At the destination, the incoming data unit is divided by the same number. If at this step there is no remainder, the data unit is assumed to be correct and is therefore accepted.

• A remainder indicates that the data unit has been damaged in transit and therefore must be rejected.



Q.8. Write short notes on following: **(4+4+4.5 = 12.5)**

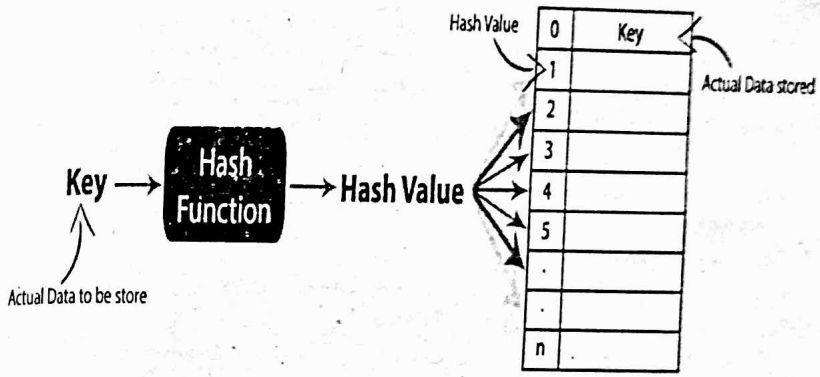
(a) Hashing (b) B* tree (c) Coloring of graphs

Ans. (a) Hashing: Hashing is the process of indexing and retrieving element (data) in a datastructure to provide faster way of finding the element using the hash key.

Here, hash key is a value which provides the index value where tha actual data is likely to store in the datastructure.

In this datastructure, we use a concept called **Hash table** to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a hash function. That means every entry in the hash table is based on the key value generated using a hash function.

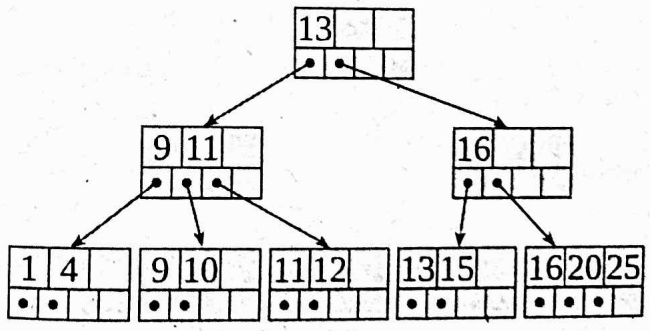
Basic concept of hashing and hash table is shown in the following figure...



Q.8.(b) B* tree

Ans. A B+-tree maintains the following invariants:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node N with k being the number of keys in N: all keys in the first child's subtree are less than N's first key; and all keys in the ith child's subtree $(2 \le i \le k)$ are between the $(i-1)$th key of n and the ith key of n.
- The root has at least two children.
- Every non-leaf, non-root node has at least *floor*$(d / 2)$ children.
- Each leaf contains at least *floor*$(d / 2)$ keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.



Q.8.(c) Coloring of graphs.

Ans. Graph Coloring :

Graph coloring is nothing but a simple way of labelling graph components such as vertices, edges, and regions under some constraints. In a graph, no two adjacent vertices, adjacent edges, or adjacent regions are colored with minimum number of colors. This number is called the **chromatic number** and the graph is called a **properly colored graph**.

While graph coloring, the constraints that are set on the graph are colors, order of coloring, the way of assigning color, etc. A coloring is given to a vertex or a particular region. Thus, the vertices or regions having same colors form independent sets.

Vertex Coloring

Vertex coloring is an assignment of colors to the vertices of a graph 'G' such that no two adjacent vertices have the same color. Simply put, no two vertices of an edge should be of the same color.

Chromatic Number

The minimum number of colors required for vertex coloring of graph 'G' is called as the chromatic number of G, denoted by X(G).

$X(G) = 1$ if and only if G 'is a null graph. If G 'is not a null graph, then $X(G) \geq 2$

END TERM EXAMINATION [MAY-JUNE 2018]
SECOND SEMESTER [MCA]
DATA AND FILE STRUCTURES [MCA–102]

Time : 3 hrs. **M.M. : 75**

Note: Attempt all questions as directed. Internal choice is indicated.

PART–I

Q. 1. Attempt any ten questions. Each question carries equal marks.

(a) Which sorting algorithm is best if the list is already sorted and why? **(2.5)**

Ans. There are many sorting algorithms. Any algorithm that can efficiently sort your problem, is also suitable for already sorted problem. If the data is already sorted, Insertion Sort is the best choice for you as it will complete its operation in linear time. If the list is already sorted there are no swaps and the algorithm will run for only n times for the best case: O(n)

Q. 1. (b) What is the maximum possible number of node in a binary tree at level 6? **(2.5)**

Ans. Level of a node is distance from root to that node. For example, level of root is I and levels of left and right children of root is 2. The maximum number of nodes on level 6 of a binary tree, is 26-1 25 32 nodes.

Q. 1. (c) Which dara structure is applied when dealing with a recursive function? **(2.5)**

Ans. Recursion is an application of Stack. All the recursive calls are maintained in stack during the execution of a program.

Q. 1. (d) Are linked lists considered linear or non linear or non linear data structure? **(2.5)**

Ans. It depends on where you intend to apply linked lists. If you based it on storage, a linked list is considered non-linear. On the other hand, if you based it on access strategies, then a linked list is considered linear.

Q. 1. (e) Differentiate NULL and VOID. **(2.5)**

Ans. NULL and void are two separate concepts:

- When a variable has the value NULL, it means that the variable is not referring to any variable or object. It is a special reserved value of a pointer when we want pointer does not point to any object.

- void is a data type which is used as a return type for a function that does not return a value.

So NULL is a specific value that a variable can have, and void indicates return type of a function

- two completely different things.

We can create a pointer of type void like:

void *p;

Q. 1. (f) What is the advantage of heap over a stack? **(2.5)**

Ans. Basically, the heap is more flexible than the stack. That's because memory space for the heap can be dynamically allocated and de-allocated as needed. However, memory of the heap can at times be slower when compared to that stack.

Q. 1. (g) How do signed and unsigned numbers affect memory. (2.5)

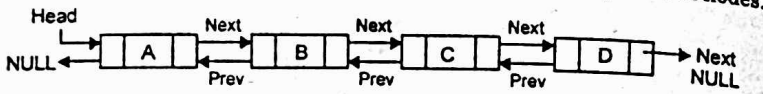
Ans. In the case of signed numbers, the first bit is used to indicate whether positive or negative, which leaves you with one bit short. With unsigned numbers, you have all bits available for that number. The effect is best seen in the number range (unsigned 8 bit number has a raimc 0-255. while 8-bit signed number has a ransc –128 to +127.

Q. 1. (h) What is the minimum number of queues needed when implementing a priority queue? (2.5)

Ans. Two. Minimum number of queues needed to implement priority queue is One queue is used for the actual storing of data, and the other one is used for storing the priorities.

Q. 1. (i) Explain double linked lists. (2.5)

Ans. A doubly-linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.



Advantages of Doubly linked list

(1) A DLL can be traversed in both forward and backward direction.

(2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

(3) We can quickly insert a new node before a given node.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Q. 1. (j) What is the Empty () member method? (2.5)

Ans. Empty ( ): This method used to check whether the stack is empty or not. It can be created for other data structures as well. EmptyQ method for stack is as follows: boo I Empty ()

```
{
    if( top == -1) //stack is empty
    return true;
    else
    return false;
}
```

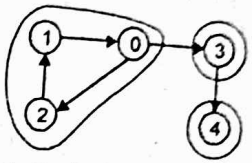
Q. 1. (k) WHat are graphs? What are strongly connected graphs? (2.5)

Ans. A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as:

A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph:



Q. 1. (l) Explain the concept of primitive data structure. (2.5)

Ans. Primitive data structures are those which are predefined way of storing data by the system. And the set of operations that can be performed on these data are also predefined. Primitive Data Structures arc the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Primitive data structures arc char. int. float, double.

PART–II

Q. 2. (a) Compare two functions n^2 and $2^n/4$ for various values of n. Determine when second becomes larger than first. (6.5)

Ans.

f(n) n	n^2	(2^n) /4
1	1	0.5
2	4	1
3	9	2
4	10	4
5	25	8
6	36	16
7	49	32
8	64	64
9	81	128

The above table clearly shows that value of second function becomes larger than e first function when the value of n is 9.

Q. 2. (b) How do you find the complexity of an algorithm? What is the relation tween the time and space complexities of an algorithm? Explain with example. (6)

Ans. Algorithm complexity is a measure which evaluates the order of the count of erations, performed by a given or algorithm as a function of the size of the input data. put this simpler, complexity is a rough approximation of the number of steps necessary execute an algorithm. When we evaluate complexity we speak of order of operation unt, not of their exact count. For example if we have an order of N^2 operations to process elements, then $N^2/2$ and $3*N^2$ are of one and the same quadratic order.

Algorithm complexity is commonly represented with the O(f) notation, also own as asymptotic notation or "Big O notation", where f is the function of the size of e input data.

The asymptotic computational complexity O(f) measures the order of the consumed sources (CPU time, memory, etc.) by certain algorithm expressed as function of the put data si/e. Complexity can be constant, logarithmic, linear, n*log(n), quadratic, cubic, ponential, etc. This is respectively the order of constant, logarithmic, linear and so on,

number of steps, are executed to solve a given problem. For simplicity, sometime instead of "algorithms complexity" or just "complexity" we use the term "running time".

We use different notation to describe limiting behavior of a function:

O-notation:

To denote asymptotic upper bounds, we use O-notation. For a given function g(n), we denote by O(g(n)) (pronounced "big-oh of g of n") the set of functions:

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le c*g(n)$ for all $n \ge n_0\}$

Ω-notation:

To denote asymptotic lower bound, we use Ω-notation. For a given function g(n). We denote by Ω(g(n)) (pronounced "big-omega of g of n").the set of functions:

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le c*g(n) \le f(n)$ for all $\ge n_0\}$

Θ-notation:

To denote asymptotic tight bound, we use Θ-notation. For a given function g(n), we denote by Θ(g(n)) (pronounced "big-there of g of n") the set of functions.

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 * g(n) \le f(n) \le c_2 * g(n)$ for all $n > n_0\}$

Time-Space Trade-off

- The best algorithms to solve a given problem is one that requires less space and less time to complete its execution.
- It is not always possible to achieve both of these objectives because choice of time or space depends on some specific conditions.

Q. 3. (a) Write an algorithm to evaluate a postfix expression. Execute your algorithm as your input ab+cd*f↑. (6)

Ans. Evaluation of a Postfix Expression: Suppose P is a arithmetic expression written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Evaluate ():

Description: Here P is a postfix expression and this algorithm evaluates it.

1. Add a ")" right parenthesis at the end of P.
2. Scan P from left to right and repeat steps 3 & 4 for each element of P until ")" is encountered.
3. If an operand is encountered, push it onto stack.
4. If an operator *opr* is encountered then:
   (a) Pop the top two elements from stack, where A is the top element and B is the next to top element.
   (b) Evaluate B *opr* A.
   (c) Place the result of *(b)* back on stack.
   [End of Step 4 If]
   [End of step 2 For Loop]
5. Set VALUE equal to the top element on the stack.
6. Exit.

Given postfix expression is

ab + cd + *f↑

Assuming a = 7, b = 5, c = 6, d = 2, f = 3

So given expression will be: 75 + 62 + *3 ↑

Symbol Scanned	Operand Stack
7	7
5	7,5
+	12
6	12,6
2	12,6,2
+	12,8
*	96
3	96, 3
↑	884736 Final Value

The given input as postfix expression is: ab + cd + *f↑

Symb	Stack	Evaluation
a	a	
b	a b	
+	pop a and b Push (a + b)	(a + b)
c	(a + b) c	
d	(a + b) cd	
+	pop c and d Push (c + d)	(c + d)
*	pop (a + b) and (c + d)	(a + b) * (c + d)
f	(a + b)* (c + d)f	
↑	pop (a + b)* (c + d) and f	(a + b)* (c + d) ↑ f

The result of evaluation is (a + b)* (c + d)) ↑ f

Q. 3. (b) Two lined lists contain information of the same type in ascending order. Write a module to merge them to a single linked list that is sorted. (6.5)

Ans. /* merges the two sorted linked lists, restricting the coomon elements to occur only once in the final list */

```
struct node
{
int data;
struct node *next;
};
void merge (struct node *p, struct node *q, struct node **s)
{
        struct node *z;
        z = NULL;
```

```
/* if both lists are empty */
if (p == NULL && q == NULL)
        return ;
/* traverse both linked lists till the end. If end of any one list is reached loop is
terminated */
while (p != NULL && q ! = NULL)
/* if node being added in the first node */
if(*s == NULL)
{
        *s = malloc ( sizeof ( struct node ));
        z = *s;
}
else
{
      z -> next =  malloc ( sizeof ( struct node ) ) ;
      z = z -> next;
}
    if ( p -> data <q → data)
    {
    z → data = P → data
    P = P → next
}
else
    {
if (q -> data < p -> data)
{
  z -> data = q -> data;
  q = q -> next;
}
else
{
    if (p -> data == q -> data)
    {
     z -> data = q-> data;
    p = p -> next;
    q = q -> next;
    }
   }
  }
}
/* if end of first list has not been reached */
while (p! = NULL)
{
```



```
        z -> next = malloc (size of (struct node));
        z = z -> next;
        z -> data = p -> data;
        p = p -> next;
}
/* if end of second list has been reached */
while (q! = NULL)
{
        z = -> next = malloc (size of (struct node));
        z = z -> next;
        z -> data = q -> data;
        q = q-> next;
}
z -> next = NULL;
}
```

UNIT–III

Q. 4. (a) What is B-TREE? Define its order and degree with a suitable example of insertion. Find out the B-tree for the following data: A, S, D, H, E, B, M, C, L, L, W, Q and X. (7)

Ans. Definition of B-Tree

B-tree is a multi-way search tree of order n that satisfied the following conditions.

(a) All the non-leaf nodes (except the root node) have at least n/2 children and at the most n children.

(b) The non-leaf root node may have at the most n non-empty child and at least two child nodes.

(c) A B-tree can exist with only one node i.e. the root node containing no child.

(d) If a node has n children then it must have n - 1 values. All the values of a particular node are in increasing order.

(e) All the values that appear on the left most child of a node are smaller than the first value of that node. All the values that appear on the right most child of a node are greater than the last value of that node.

(f) If x and y are any two i^{th} and $(i + 1)^{th}$ values of a node, where x < y, then all the values appearing, on the $(i + 1)^{th}$ sub-tree of that node are greater than x and less than y.

(g) All the leaf nodes should appear on the same level.

Let us consider the order of B-Tree as 5. So for creating a B-tree of order 5

Min. No. of children for a non-leaf node = $\frac{n}{2} = \frac{5}{2} = 3$

Max. No. of children for a non-leaf node = n = 5

Min. No. of values for a non-leaf node = 3 – 1 = 2

Max. No. of values for a non-leaf node = 5 – 1 = 4

Min. No. of children for a root node = 2

Max. No. of children for a root node = 5

Min. No. of values for a root node = 2 – 1 = 1
Max. No. of values for a root node = 5 – 1 = 4

Split a node if it contains more than max. no. of values allowed from $\frac{n}{2} = \frac{5}{2} = 3rd$ value in the node by moving 3rd value to the parent node and by dividing that node in two values.

Ist half contains $\frac{n}{2} - 1 = 3 - 1 = 2$ values and 2nd half contain $n - \frac{n}{2} = 5 - 3 = 2$ values.

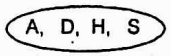
Given data
A, S, D, H, E, B, M, C, I, L, K, W, Q and X

Step 1: Insert A

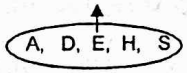
Step 2: Insert S

Step 3: Insert D

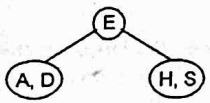
Step 4: Insert H



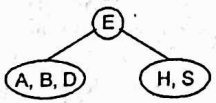
Step 5: Insert E



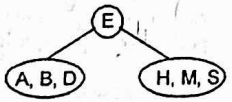
Now, the node contains more than the max. no. of values, so split the node from $\frac{5}{2} = 3rd$ value as shown below:



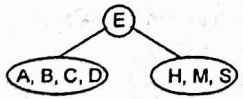
Step 6: Insert B



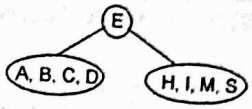
Step 7: Insert M



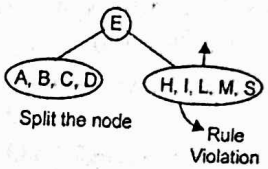
Step 8: Insert C

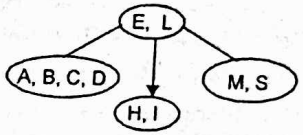


Step 9: Insert I

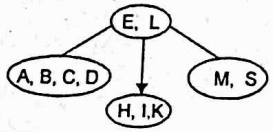


Step 10: Insert L

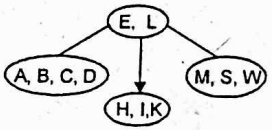




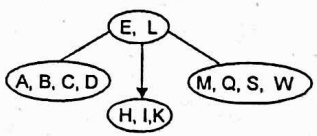
Step 11: Insert K



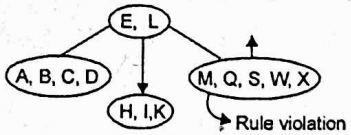
Step 12: Insert W



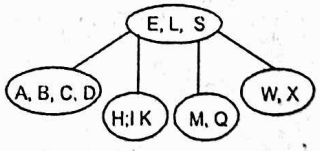
Step 13: Insert Q



Step 14: Insert X



Split the node again



This is the Final B-Tree

Q. 4.(b) What are threaded binary tree? Why they are needed in terms of memory? Make the threaded binary tree for the given data 22, 4, 12, 7, 14, 34, 23, 1, 3, 24, 13, 21, 34, 2. **(5.5)**

Ans. "A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node."

There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

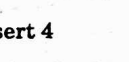
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

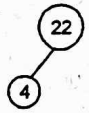
The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

Given data is

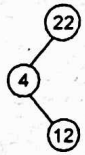
22, 4, 12, 7, 14, 34, 23, 1, 3, 24, 13, 21, 34, 2

Step 1: Insert 22 

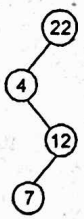
Step 2: Insert 4



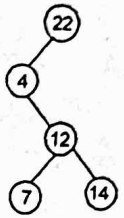
Step 3: Insert 12



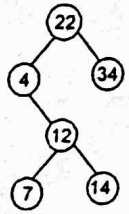
Step 4: Insert 7



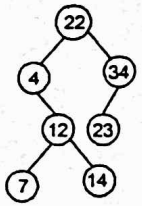
Step 5: Inser'14



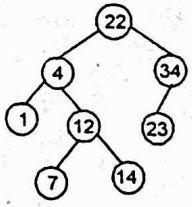
Step 6: Insert 34



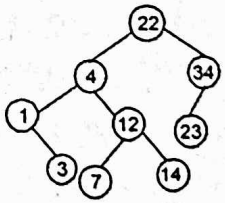
Step 7: Insert 23



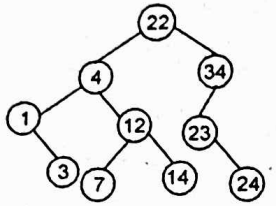
Step 8: Insert 1



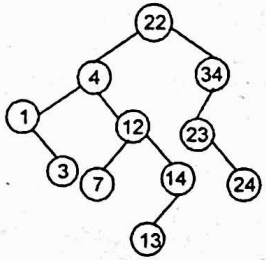
Step 9: Insert 3



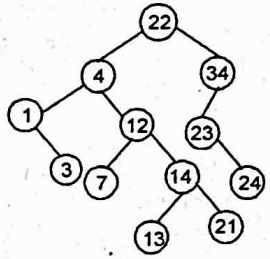
Step 10: Insert 24



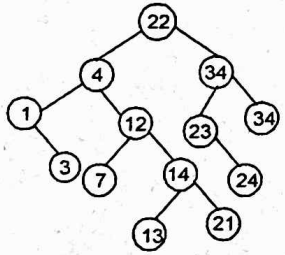
Step 11: Insert 13



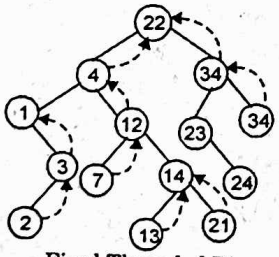
Step 12: Insert 21



Step 13: Insert 34



Step 14: Insert 2



Final Threaded Binary tree

Q. 5. (a) What is B-Tree? Define its order and degree with a suitable example of insertion. Prepare a B-Tree from the given information of order: 4, 22, 4, 12, 7, 14, 34, 23, 1, 3 24, 13, 21, 34, 2.

Ans. Definition of B-Tree

B-tree is a multi-way search tree of order n that satisfies the following conditions.

(a) All the non-leaf nodes (except the root mode) have at least n/2 children and at the most n children

(b) The non-leaf root node may have at the most an non-empty child and at least two child nodes.

(c) A B-tree can exist with only one node i.e., the root node containing no child.

(d) If a node has a children then it must have n - 1 values. All the values of a particular node are in increasing order.

(e) All the values that appear on the left most child of a node are smaller than the first value of that node. All the values that appear on the right most child of a node are greater than the last value of that node.

(f) If x and y are any two i^{th} and $(i + 1)^{th}$ values of a node, where x < y. then all the values appearing on the $(i + 1)^{th}$ sub-tree of that node are greater than x and less than y.

(g) All the leaf nodes should appear on the same level.

Let us consider the order of B-Tree as 5. So for creating a B-tree of order 5

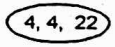
Given data is

4, 22, 4, 12, 7 14, 34, 23, 1, 3, 24, 13, 21, 34, 2

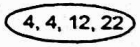
Step 1: Insert 4

Step 2: Insert 22

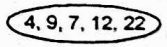
Step 3: Insert 4



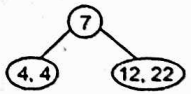
Step 4: Insert 12



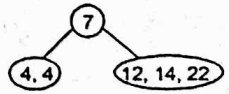
Step 5: Insert 7



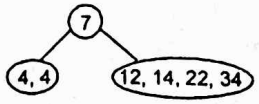
Now, split the node from the 3rd value.



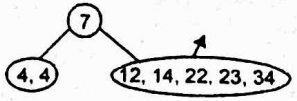
Step 6: Insert 14



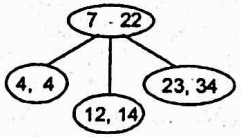
Step 7: Insert 34



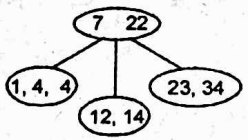
Step 8: Insert 23



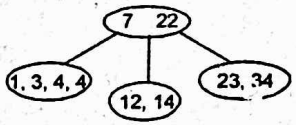
Split again



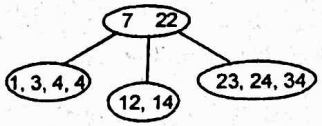
Step 9: Insert 1



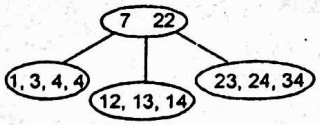
Step 10: Insert 3



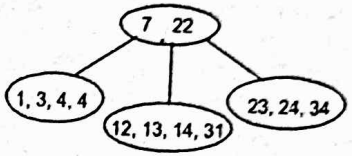
Step 11: Insert 24



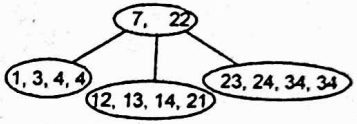
Step 12: Insert 13



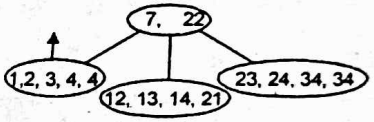
Step 13: Insert 21



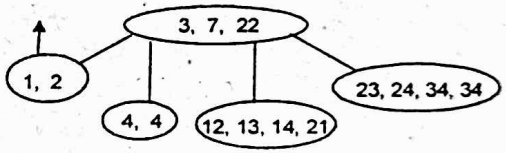
Step 14: Insert 34



Step 15: Insert 2



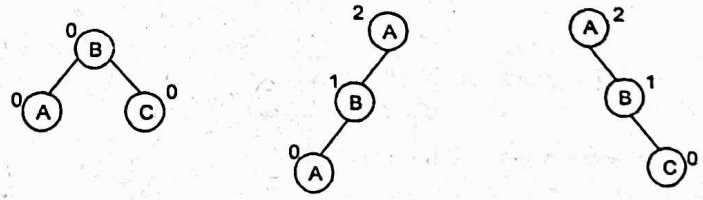
After Splitting



Final B-Tree of order – 5.

Q. 5. (b) What is AVL tree? Describe tree rotation in AVL tree (Right and left) with diagram and examples. (5.5)

Ans. Named after their inventor Adelson, Velski & Landis, AVI, trees are height balancing brnary search tree. AVI, tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor. Here we see that the llrst tree is balanced and the next two trees arc not balanced



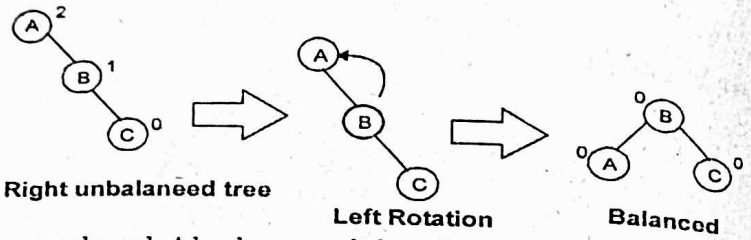
Balanced **Not balanced** **Not balanced**

In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2T In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVI, tree permits difference (balance factor) to be only I .

BalanceFactor = height(left-sutree) - height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

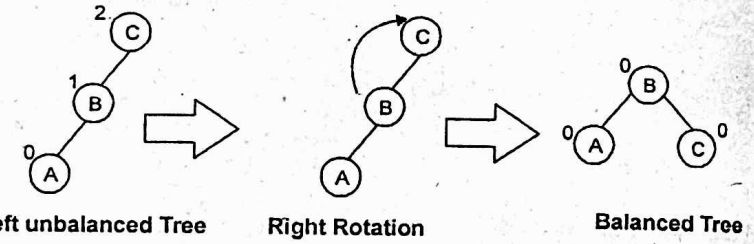
Left Rotation: If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree. then we perform a single left rotation



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

Right Rotation:

AVI. tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by perfoming a right rotation.

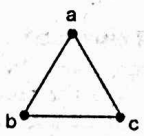
PART–IV

Q. 6. (a) Draw the complete undirected graphs on one, two, three, four and five vertices, Prove that the number of edges in an n vertex complete graph is n(n-1)/2 **(5.5)**

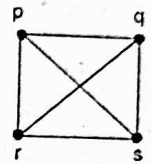
Ans. Below is a complete graph with N = 1 vertex.

Below is a complete graph with N = 2 vertices.

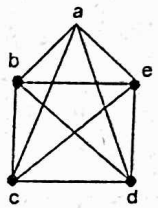
Below is a complete graph with N = 3 vertices.



Below is a complete graph with N = 4 vertices.



Below is a complete graph with N = 5 vertices.



Prove that the number of edges in an n vertex complete graph is n(n-1)/2.

Proof: This is easy to prove by induction. If n : 1, zero edges are required, and 1(1 – 0)/2 = 0. Assume that a complete graph with k vertices has k(k – 1)/2. When we add the (k + 1)st vertex, we need to connect it to the k original vertices, requiring k additional edges. We will then have k(k – 1)/2 + k = (k + 1) ((k +1) – 1)/2 vertices, and we are done.

Q. 6. (b) What is Quick sort? Sort the following array using quick sort method: 24, 56, 47, 35, 10, 90, 82, 31. **(7)**

Ans. Quicksort is a sorting algorithm which uses the idea of divide and conquer. This algorithm chooses or selects an element known as pivot element, finds its position in the array such that it divides the array into the two sub arrays. The number of elements in the left sub array are less than and the elements in the right sub array are greater than the dividing element. Then the algorithm is repeated individually for these two sub array. It is also called partition-exchange sort.

The given data is :

24 56 47 35 10 90 82 31

Pass 1:-	(10)	24	(56	47	35	90	82	31)
Pass 2:	10	24	(56	47	35	90	82	31)
Pass 3:	10	24	(47	35	31)	56	(90	82)
Pass 4:	10	24	(35	31)	47	56	(90	82)
Pass 5:	10	24	(31)	35	47	56	(90	82)
Pass 6:	10	24	31	35	47	56	(90	82)
Pass 7:	10	24	31	35	47	56	(82)	90
Pass 8:	10	24	31	35	47	56	82	90

PART–IV

Q. 7. (a) The following values are to be sorted in a hash table: 25, 42, 96, 101, 102, 162, 197. Describe how the values are hashed by using division method of hashing with a table size of 7. Use chaining as the method of collision resolution. **(6.5)**

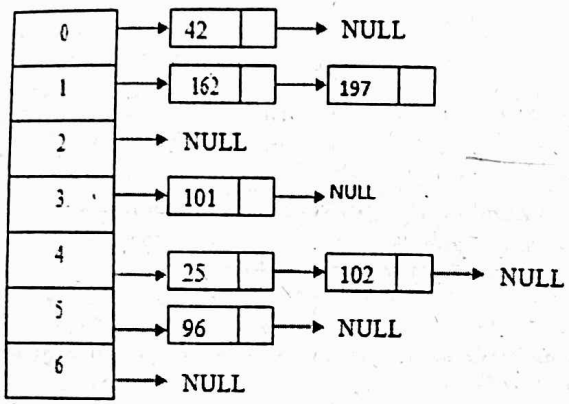
Ans. Given Table size = 7

25%7=4

42%7=0

96%7=5
101%7=3
102%7=4
162%7=1
197%7=1

So collision resolution can be resolved using chaining method as shown below:



Q. 7. (b) Consider the following specification of a Graph G. **(6)**

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

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

(i) Draw and undirected graph

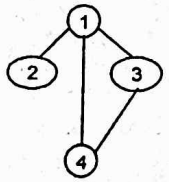
(ii) Draw its adjacency matrix

Ans. Given specification of a Graph G is

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

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

(i) Undirected Graph:



Undirected Graph as per the list of edges given

(ii)

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

Adjacency matrix for the above undirected graph.

Q. 8. (a) How do Binary Files work? Explain with example? **(6)**

Ans. Binary files are very similar to arrays of structures, except the structures are in a disk file rather than in an array in memory. Because the structures in a binary file are on disk, you can create very large collections of them (limited only by your available disk space). They are also permanent and always available. The only disadvantage is the slowness that comes from disk access time.

Binary files have two features that distinguish them from text files:

- You can jump instantly to any structure in the file, which provides random access as in an array.
- You can change the contents of a structure anywhere in the file at any time.

Binary files also usually have faster read and write times than text files, because a binary image of the record is stored directly from memory to disk (or vice versa). In a text file, everything has to be converted back and forth to text, and this takes time.

All electronic information, regardless of the format, is ultimately stored in a binary form—as a series of bits (zeroes and ones). However, the same value can be recorded as a binary value in a number of different ways.

For example, given the number 12345, we could store it as individual characters 1, 2, 3, 4, and 5, using one byte for each character:

00110001 00110010 00110011 00110100 00110101

Alternatively, we could store the number as a four-byte integer

00111001 00110000 00000000 00000000

Binary formats are consequently much harder to write software for, which results in there being less software available to do the job.

However, some binary formats are easier to read than others. Given that a description is necessary to have any chance of reading a binary file, proprietary formats, where the file format description is kept private, are extremely difficult to deal with. Open standards become more important than ever.

C supports the file-of-structures concept very cleanly. Once you open the file you can read a structure, write a structure, or seek to any structure in the file. This file concept supports the concept of a file pointer. When the file is opened, the pointer points to record 0 (the first record in the file). Any read operation reads the currently pointed-to structure and moves the pointer down one structure. Any write operation writes to the currently pointed-to structure and moves the pointer down one structure. Seek moves the pointer to the requested record.

Example: Reading and writing the records in a Binary File

```
struct rec
{
  int x,y,z;
};
/* Writing the records in a Binary Files */
  f=fopen("junk","w");
  for (i=1;i<=10; i++)
  {
    r.x=i;
    fwrite(&r,sizeof(struct rec),1,f);
```

```
}
fclose(f);
/* Reading the records in a Binary Files */
f=fopen("junk","r");
  for (i=1;i<=10;i++)
{
  fread(&r,sizeof(struct rec),1,f);
  printf("%d\n",r.x);
}
fclose(f);
```

Q. 8. (b) What is a text file? How do text files differ from other type of files? **(6.5)**

Ans. A text file is the one where data is stored as a stream of characters that can be processed sequentially. In the text format, data are organized into lines terminated by newline character. The text files are in human readable form and they can be created and read using any text editor. Since text files process only characters, they can read or write only one character at a time. A new line may be converted to carriage return and line feed character. Figure below shows the data transfer in text file.

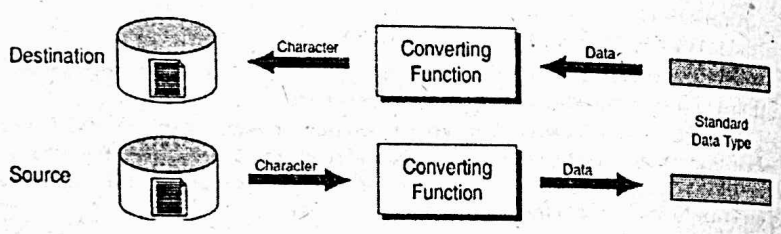


Fig. Reading and Writing Text Files

Because of these translations, the number of characters written / read may not be the same as the number of characters written / read on the external device. Therefore, there may not be a one – to – one relationship between the characters written / read into the file and those stored on the external devices.

For example, the data 2345 requires 4 bytes with each character occupying exactly one byte. A text file cannot contain integers, floating – point numbers etc. converted to their character – equivalent formats.

Text File	Binary File
1. Human readable format.	1. Not in human readable format.
2. Data is stored as lines of characters with each line terminated by / n which may be translated into carriage return + line feed.	2. Data is stored on the disk in the same way as it is represented in the computer memory.
3. Translation of new line character to carriage return + line feed.	3. No translation of any type.
4. Data can be read using any of the text editors.	4. Data can be read only by specific programs written for them.
5. The number of characters written / read may not be the same as the number of characters written/read on the external device such as disk.	5. The number of characters written / read is same as the number of characters written / read on the external device such as disk.
6. There may not be a one – to- one relationship between the characters written / read into the file and those stored on the external devices.	6. There is one – to – one relationship between the characters written / read into the file and those stored on the external devices.
7. The data 2345 requires 4 bytes with each character requiring one byte. So, it is stored as 0 × 32, 0 × 33, and 0 × 34, 0 × 35 (ASCII values) thus requiring 4 bytes.	7. The data 2345 requires 2 bytes and stored as 0 × 0929. The data 2345 is requiring 2 bytes.

Q. 9. (a) Explain Inverted files with example. **(6)**

Ans. Inverted Files:

- Inverted files are similar to multi-lists.
- The difference between multi-lists and inverted files is that in multi-lists records with the same key value are linked together along with the link information being kept in individual record. But in case of inverted files this link information is kept in the index itself.

For example:

437	BBB
488	EEE
689	DDD
695	AAA
778	CCC

Fig. 1 (a) Roll_No Index

Fifth	BBB, CCC
Tenth	EEE, DDD, AAA

Fig. 1 (b) Class Index

Female	BBB, DDD, CCC
Male	EEE, AAA

Fig. 1 (c) Sex Index

First class	EEE
Second class	AAA, DDD, CCC
Pass class	BBB

Fig. 1 (d) Marks Index

formatted text, such as older Microsoft Word document files, contain the text of the document but also contain formatting information in binary form.

Binary files are usually thought of as being a sequence of bytes, which means the binary digits (bits) are grouped in eights. Binary files typically contain bytes that are intended to be interpreted as something other than text characters. Compiled computer programs are typical examples; indeed, compiled applications are sometimes referred to, particularly by programmers, as binaries. But binary files can also mean that they contain images, sounds, compressed versions of other files, etc.—in short, any type of file content whatsoever.

Some binary files contain headers, blocks of metadata used by a computer program to interpret the data in the file. The header often contains a signature or magic number which can identify the format. For example, a GIF file can contain multiple images, and headers are used to identify and describe each block of image data. The leading bytes of the header would contain text like GIF87a or GIF89a that can identify the binary as a GIF file. If a binary file does not contain any headers, it may be called a flat binary file.

END TERM EXAMINATION [MAY–2019]
SECOND SEMESTER [MCA]
DATA AND FILE STRUCTURE [MCA-102]

Time : 3 hrs. **M.M. : 75**

Note :- Attempt five questions in all including Q.No. 1 which is compulsory. Select one question from each unit.

Q.1. Write any five applications of stacks.

Ans. Applications of Stack **(2.5)**

- **Expression Evaluation:** Stack is used to evaluate prefix, postfix and infix expressions.
- **Expression Conversion:** An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.
- **Syntax Parsing:** Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.
- **Backtracking:** Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.
- **Parenthesis Checking:** Stack is used to check the proper opening and closing of parenthesis.
- **Function Call:** Stack is used to keep information about the active functions or subroutines.

Q.1. (b) Define queue and what are the operartions that can be performed on queues? **(2.5)**

Ans. A queue is a linear list in which elements may be added only at one end known as rear end of the queue and elements may be removed only at the other end known as front end of the queue. The name "queue" likely comes from the everyday use of the term. Queue is a linear data structure. **Consider:** queue of people waiting at a bus stop. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first. Clearly, the first person in the line is the first person to leave. Thus queues are also called first-in first-out (FIFO) lists. Another example of a queue is a batch of jobs waiting to be processed, assuming no job has higher priority than the others.

Operations performed on Queues:-

The operations on a queue are

1. insert(queue, element): The arguments of the function are the queue and element is the item to be inserted into the queue at the rear end. Insertion of an element in a queue also known as enqueue() operation.

2. delitem = remove(queue): This function deletes the front element from the queue **queue** and sets **delitem** to its contents. Deletion is known as dequeue() operation.

3. empty(queue): The function returns a false or a true depending on whether or not the queue contains any elements.

The **insert** function can always be performed since there is no limit on the number of elements in a queue in Linked List representation only.

The **remove** function can be performed only if the queue is non-empty. The result of an attempt to remove an element from an empty queue is called the **underflow**.

The **empty** operation can always be performed.

Q.1. (c) Differentiate between full binary tree & complete binary tree? **(2.5)**

Ans. A complete binary tree is a binary tree in which every level of the binary tree is completely filled except the last level. In the unfilled level, the nodes are attached starting from the left-most position. A full binary tree is a tree in which every node in the tree has two children except the leaves of the tree. Full binary tree is a binary tree in which every node in the tree has exactly zero or two children.

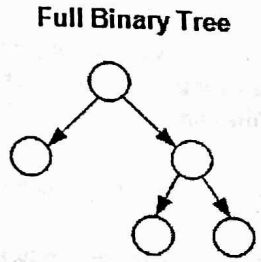


Fig.1. Full Binary Tree

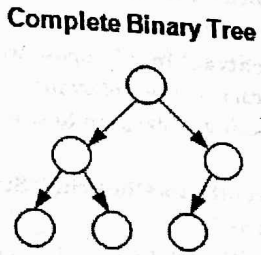


Fig.2. Complete Binary Tree

In a full binary tree, if you know the number of total nodes or the number of leaves or the number of internal nodes, you can find the other two very easily. But a complete binary tree does not have a special property relating these three attributes.

Q.1. (d) List the properties of B-tree? **(2.5)**

Ans. Definition of B-Tree

B-tree is a multi-way search tree of order n that satisfies the following conditions:

(a) All the non-leaf node (except the root node) have at least n/2 children and at the most n children.

(b) The non-leaf root node may have at he most n non-empty child and atleast two child nodes.

(c) A B-tree can exist with only one node i.e. the root node containing no child.

(d) If a node has n children then it must have n – 1 values. All the values of particular node are in increasing order.

(e) All the values that appear on the left most child of a node are smaller than the first value of that node. All the values that appear on the right most child of a node are greater than the last value of that node.

(f) If x and y are any two i^{th} and $(i + I)^{th}$ values of a node, where x < y, then all the values appearing on the $(i + 1)^{th}$ sub-tree of that node are greater than x and less than y.

(g) All the leaf nodes should appear on the same level.

Q.1. (e) Write short note on bubble sort algorithm. **(2.5)**

Ans. Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Bubble Sort ():

Description: Here **A** is an unsorted array having **N** elements.

1. Repeat For J = 1 to N
2. Repeat For K = 1 to N-J
3. If (A[K] > A[K+1]) Then
4. Interchange A[K] and A[K+1]

[End of If]

[End of Step 2 For Loop]

[End of Step 1 For Loop]

5. Exit

Q.1. (f) What are the merits and demerits of binary search? **(2.5)**

Ans. Merits of Binary Search:

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Disadvantages of Binary Search:

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor(because of its random access nature).

Q.1. (g) Explain critical path **(2.5)**

Ans. Critical path is the sequential activities from start to the end of a project. Although many projects have only one critical path, some projects may have more than one critical paths depending on the flow logic used in the project.

If there is a delay in any of the activities under the critical path, there will be a delay of the project deliverables.

Most of the times, if such delay is occurred, project acceleration or re-sequencing is done in order to achieve the deadlines.

Critical path method is based on mathematical calculations and it is used for scheduling project activities. In the critical path method, the critical activities of a program or a project are identified. These are the activities that have a direct impact on the completion date of the project.

Q.1. (h) What is meant by activity network? **(2.5)**

Ans. An Activity Network identifies the activity sequences and dependencies critical to the success of a project. Using an Activity Network, project management is able to identify the sequence of activities that should provide the most efficient time and duration estimate for the completion of a project. An Activity Network depicts the sequencing of the activities, the activities that must be completed before other activities can start (in series), the activities that can be performed at the same time (in parallel), and those activities critical to a project's completion time.

Q.1. (i) What are the various disadvantages of sequential fiel oragnization? **(2.5)**

Ans. Disadvantages of sequential file organization

- The sorting does not remove the need to access other records as the search looks for particular records.
- Sequential records cannot support modern technologies that require fast access to stored records.
- The requirement that all records be of the same size is sometimes difficult to enforce.

Q.1. (j) Explain polyphose merge. **(2.5)**

Ans. A polyphase merge sort is a variation of bottom up Merge sort that sorts a list using an initial uneven distribution of sub-lists (runs), primarily used for external sorting, and is more efficient than an ordinary merge sort when there are less than 8 external working files (such as a tape drive or a file on a hard drive). A polyphase merge sort is not a stable sort.

For N less than 8 working files, a polyphase merge sort achieves a higher effective run count reduction factor by unevenly distributing sorted runs between N-1 working files; each iteration merges runs from N-1 working files onto a single output working

file. When the end of one of the N-1 working files is reached, then it becomes the new output file and what was the output file becomes one of the N-1 working input files, starting a new iteration of polyphase merge sort; each iteration merges only a fraction of the dataset (about 1/2 to 3/4), except for the last iteration which merges all of the dataset into a single sorted run. The initial distribution is set up so that only one input working file is emptied at a time, except for the final merge iteration which merges N-1 single runs from the N-1 input working files to the single output file, resulting in a single sorted run, the sorted dataset.

UNIT-I

Q.2. (a) Write various steps to transform the following postfix expression to infix expression ABCDE –+$*EF* **(8)**

Ans. Given postfix expression is: ABCDE–+$* EF*

S. No.	Symbol Scanned	Final Stack
1	A	A
2	B	AB
3	C	ABC
4	D	ABCD
5	E	ABCDE
6	–	ABC, (D-E)
7	+	AB, C+(D-E)
8	$	A, B$(C+(D-E))
9	*	A*B$(C+(D-E))
10	E	A*B$(C+(D-E)), E
11	F	A*B$(C+(D-E)), E, F
12	*	A*B$(C+(D-E)), (E*F)
13	-	A*B$(C+(D-E))–(E*F) **[Final Infix Expression]**

Q.2. (b) Differentiate between multistack and multiqueue. **(4.5)**

Ans. Multi-Stack

If we need two or more stacks in a program, then it casn be accomplished in two ways. One way is to have a separate array for each stack in the program. This approach has a disadvantage that if one stack needs to store larger number of elements than the specified size and another stack has less number of elements, then it is not possible to store the elements of the first stack in the second stack in spite of the vacant space. This problem can be solved by having a single arrary of sufficient size to hold two or more stacks. For simplicity, the representation of two stacks in a single array. The two stacks can be represented efficiently in the same array if one stack grows from left to

right and other grows from right to left. The memory is also utilized more efficiently in this case.

For example, consider an array stack [MAX] to hold two stacks, say stack1 and stack 2. Two top variables Top1 and Top2 are required to represent the top of two stacks. Initially, Top1 is set to –1, and Top2 is set to MAX to represent empty stacks. The condition of overflow in this case occurs when the combined size of both the stacks will exceed MAX. For this, we use a variable count that keeps track of the number of elements stored in the array. Initially, count is set to 0. If the value of count exceeds the value of MAX, and an attempt is made to insert a new element, the overflow occurs.

To push an element in stack1, Top1 is incremented by one and the element is inserted at that position. On the other hand, to push an element in stack2, Top2 is decremented by one and the element is inserted at that position. Top pop an element from stack1, the element at the position indicated by Top1 is assigned to a local variable, and then Top1 is decremented by one. On the other hand, to pop an element from stack2, the element at the position indicated by Top2 is assigned to a local variable, and then Top2 is incremented by one. Figure shows an array of size MAX to hold two stacks stack 1 and stack 2, where stack 1 grows from left to right and stack 2 grows from right of left.

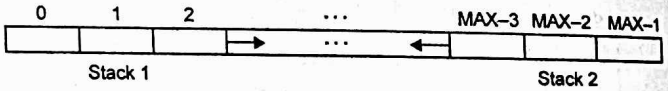


Fig. Representing two Stacks by an Array of Size MAX

```
struct multi-stack
{
int item[MAX];
int Top1, Top2;
int count;
int sno; /*sno indicate the stack number (1 or 2)*/
};
```

Multi queue is data structure in which multiple queues are maintained. This type of data structures are utilized for process scheduling. We might use one dimensional array or multidimensional array to illustrated a multiple queue.

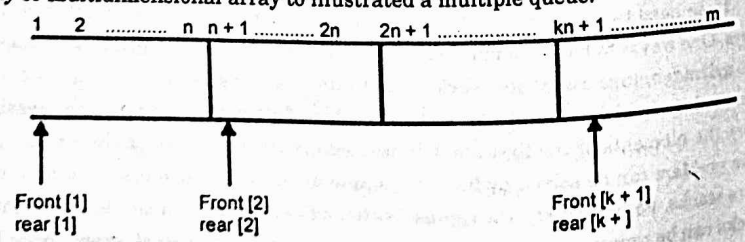


Fig. Multiple Queues in an array

A multi queue implementation by using a single dimensional array along m elements is illustrated in figure. Each of queues contains n elements that are mapped to a linear array of m elements.

Q.3. Define doubly linked list. Write the procedure to implement following operations on a doubly linked list. (12.5)

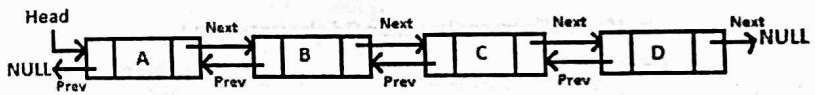
(i) Insert an element at beginning.

(ii) Insert an element at a particular location

(iii) Delete an element from beginning.

(iv) Delete an element form a particular location.

Ans. A doubly-linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.




```
/* Function to insert an element at the beginning of the doubly linked list */
void d_addatbeg ( struct dnode **s, int num )
{
  struct dnode *q ;

  /* create a new node */
  q = malloc ( sizeof ( struct dnode ) ) ;

  /* assign data and pointer to the new node */
  q -> prev = NULL ;
  q -> data = num ;
  q -> next = *s ;

  /* make new node the head node */
  ( *s ) -> prev = q ;
  *s = q ;
}

/* Function to insert an element at a particular location in doubly linked list*/
void d_addafter ( struct dnode *s, int loc, int num )
```

```
{
    struct dnode *temp , *q;
    int i ;
    q = s;

    /* skip to desired portion */
    for ( i = 1 ; i < loc - 1 ; i++ )
    {
        q = q -> next ;
        /* if end of linked list is encountered */
        if ( q == NULL )
        {
            printf ( "\nThere are less than %d elements", loc );
            return ;
        }
    }
    /* insert new node */
    temp = malloc ( sizeof ( struct dnode ) ) ;
    temp -> data = num ;
    temp -> prev = q ;
    temp -> next = q -> next ;
    temp -> next -> prev = temp ;
    q -> next = temp ;
}

/* deletes an element from beginning or from a particular location from the doubly
linked list */
void d_delete ( struct dnode **s, int num )
{
    struct dnode *q = *s ;

    /* traverse the entire linked list */
    while ( q != NULL )
    {
        /* if node to be deleted is found */
        if ( q -> data == num )
```



```
        {
            /* if node to be deleted is the first node */
            if ( q == *s )
            {
                *s = ( *s ) -> next ;
                ( *s ) -> prev = NULL ;
            }
            else
            {
                /* if node to be deleted is the last node */
                if ( q -> next == NULL )
                    q -> prev -> next = NULL ;
                else
                /* if node to be deleted is any intermediate node */
                {
                    q -> prev -> next = q -> next ;
                    q -> next -> prev = q -> prev ;
                }
                free ( q ) ;
            }
            return ; /* return back after deletion */
        }
        q = q -> next ; /* go to next node */
    }
    printf ( "\n%d not found.", num ) ;
}
```

UNIT-II

Q.4. (a) Construct a binary tree whose order of needs in is given as below.

Inorder : DBHEAIFJCG

Preorder : ABDEHCFIJG **(8)**

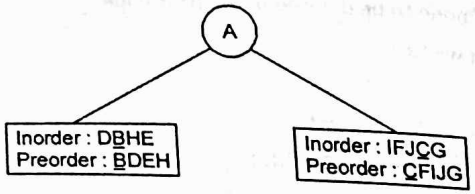
Ans. Given:

Inorder : DBHEAIFJCG

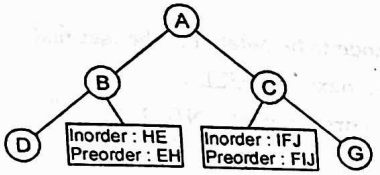
Preorder : ABDEHCFIJG

Step 1 : From preorder traversal, we can find the root of the tree as 'A' and inorder traversal is used to find left & right subtree,

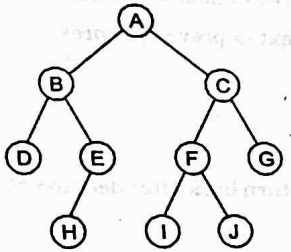
So, we will have



Step 2 :



Step 3 :



Final Binary Tree

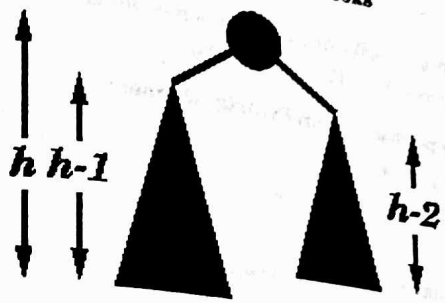
Q.4. (b) What is a AVL tree? Explain the advantages of AVL trees. **(4.5)**

Ans. An **AVL tree** is another balanced binary search tree. Named after their inventors, Adelson-Velskii and Landis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an **O(logn)** search time. Addition and deletion operations also take **O(logn)** time.

Definition of an AVL tree

An AVL tree is a binary search tree which has the following properties:

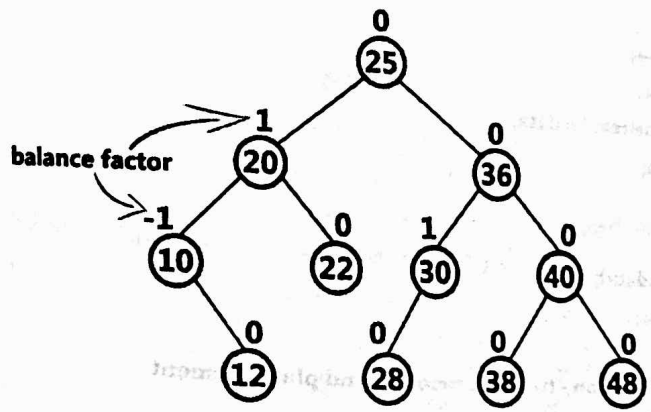
1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.



Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.

One of the advantage of AVL tree is better search times for keys. The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is O(h) where h is height of the tree. Since AVL tree is balanced, the height is O(Log n). So time complexity of AVL delete is O(Log n).

Example of AVL Tree



The above tree is a binary search tree and every node is satisfying balance factor condition.

Q.5. Explain in detail the various operations that can be perfomed on priority queue. **(12.5)**

Ans. Priority queue contains data items which have some preset priority. While removing an element from a priority queue, the data item with the highest priority is

removed first. In a priority queue, insertion is performed in the order of arrival and deletion is performed based on the priority.

Operations performed on Priority Queue:

```
//Create Function
void create()
{
    front = rear = -1;
}

// Function to perform insertion by priority
void insert_by_priority(int data)
{
    if (rear >= MAX - 1)
    {
printf("\nQueueOverflow");
return;
}
    if ((front == -1) && (rear == -1))
{
front++;
rear++;
pri_que[rear] = data;
return;
}
 else
check(data);
 rear++;
}
//Check Function - to check priority and place element
void check(int data)
{
int i,j;
for (i = 0; i <= rear; i++)
{
if (data >= pri_que[i])
{
```



```
for (j = rear + 1; j > i; j– –)
{
pri_que[j] = pri_que[j – 1];
}
pri_que[i] = data;
return;
}
}
pri_que[i] = data;
}
// Function to perform deletion by priority
void delete_by_priority(int data)
{
int i;
if ((front==-1) && (rear==-1))
{
printf("\nQueue is empty no elements to delete");
return;
}
for (i = 0; i <= rear; i++)
{
if (data == pri_que[i])
{
for (; i < rear; i++)
{
pri_que[i] = pri_que[i + 1];
}
pri_que[i] = -99;
rear– –;
if (rear == -1)
front = -1;
return;
}
}
printf("\n%d not found in queue to delete", data);
}
//Traversal operation
void traverse_pqueue()
```

```
{
if ((front == -1) && (rear == -1))
{
printf("\nQueue is empty");
return;
}
 for (; front <= rear; front++)
 {
 printf(" %d ", pri_que[front]);
 }
 front = 0;
 }
```

Unit-III

Q.6. (a) Write an algorithm to sort the elements using mergesort? (10)

Ans. Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(A, BEG, MID, END) is key process that assumes that A [BIG..MID] and A[MID+1..END] are sorted and merges the two sorted sub-arrays into one.

Merge Sort (A, BEG, END):

Description: Here **A** is an unsorted array. **BEG** is the lower bound and **END** is the upper bound.

1. If (BEG < END) Then
2. Set MID = (BEG + END) / 2
3. Call Merge Sort (A, BEG, MID)
4. Call Merge Sort (A, MID + 1, END)
5. Call Merge Array (A, BEG, MID, END)

   [End of If]
6. Exit

Merge Array (A, BEG, MID, END)

Description: Here **A** is an unsorted array. **BEG** is the lower bound, **END** is the upper bound and **MID** is the middle value of array. **B** is an empty array.

1. Repeat For I = BEG to END
2. Set B[I] = A[I] [Assign array A to B]

   [End of For Loop]

3. Set I = BEG, J = MID + 1, K = BEG
4. Repeat While (I <= MID) and (J <= END)
5. If (B[I] <= B[J]) Then [Assign smaller value to A]
6. Set A[K] = B[I]
7. Set I = I + 1 and K = K + 1
8. Else
9. Set A[K] = B[J]
10. Set J = J + 1 and K = K + 1

    [End of If]

    [End of While Loop]
11. If (I <= MID) Then [Check whether first half has exhausted or not]
12. Repeat While (I <= MID)
13. Set A[K] = B[I]
14. Set I = I + 1 and K = K + 1

    [End of While Loop]
15. Else
16. Repeat While (J <= END)
17. Set A[K] = B[J]
18. Set J = J + 1 and K = K + 1

    [End of While Loop]

    [End of If]
19. Exit

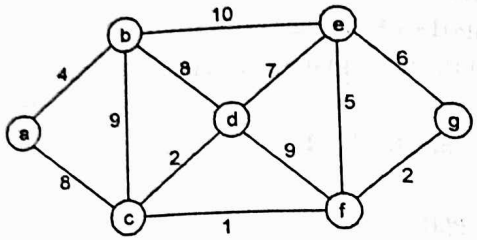
Q.6. (b) Write a short note on Hash Table. (2.5)

Ans. Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

A hash table is made up of two parts: an array (the actual table where the data to be searched is stored) and a mapping function, known as a hash function. The hash function is a mapping from the input space to the integer space that defines the indices of the array. In other words, the hash function provides a way for assigning numbers to the input data such that the data can then be stored at the array index corresponding to the assigned number.

Q.7. Find the minimum spanning tree of the following graph using Kruskal's Alogorithm. (12.5)



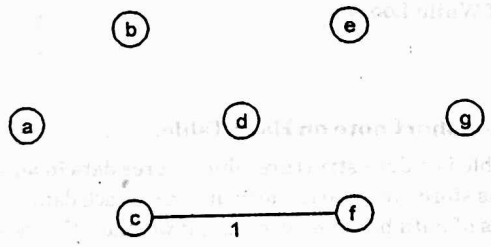
Ans. Kruskal's Algorithm:

- Kruskal's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Kruskal's algorithm, the given graph must be weighted, connected and undirected.

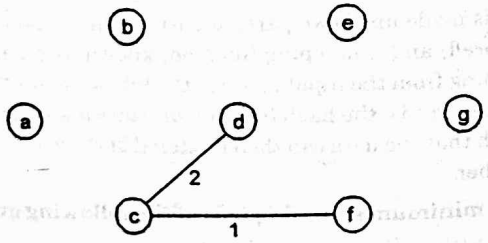
Thumb rule to construct MST using Kruskal's algorithm:

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

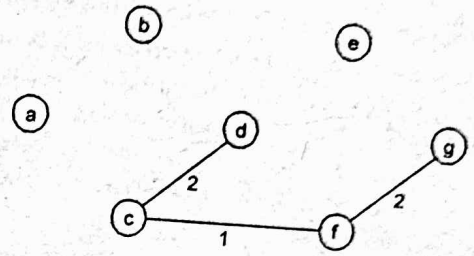
Step : 1 Add 'cf'



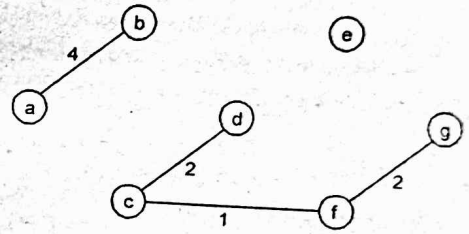
Step : 2 Add 'cd'



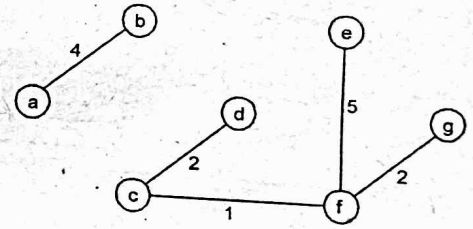
Step : 3 Add 'fg'



Step : 4 Add 'ab'

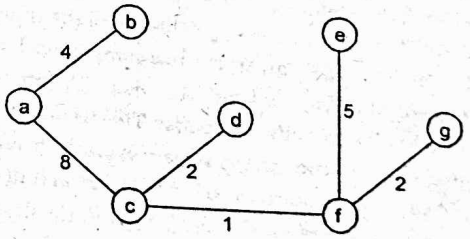


Step : 5 Add 'ef'



Step : 6 Next edge 'eg' with weight 6 needs to leave as it makes closed path. Similarly edge 'de' with weight 7 needs to leave.

Now, the next edges are 'ac' and 'bd' both with the weight 8



Final minimum spanning tree,

UNIT-IV

Q.8. (a) Differentiate between double buffering and block buffering. (4)

Ans. When several books need to be transferred from disk to main memory and all the block addresses are known, several buffers can be reserved in main memory to speed up the transfer. While one buffer is being read or written, the CPU can process data in the other buffer because an independent disk I/O processor (controller) exists that, once started, can proceed to transfer a data block between memory and disk independent of and in parallel to CPU processing.

Fig.1 illustrates how two processes can proceed in parallel. Processes A and B are running concurrently in an interleaved fashion, whereas processes C and D are running concurrently in a parallel fashion. When a single CPU controls multiple processes, parallel execution is not possible. However, the processes can still run concurrently in an interleaved way. Buffering is most useful when proces s can run concurrently in a parallel fashion, either because a separate disk I/O processor is available or because multiple CPU processors exist.

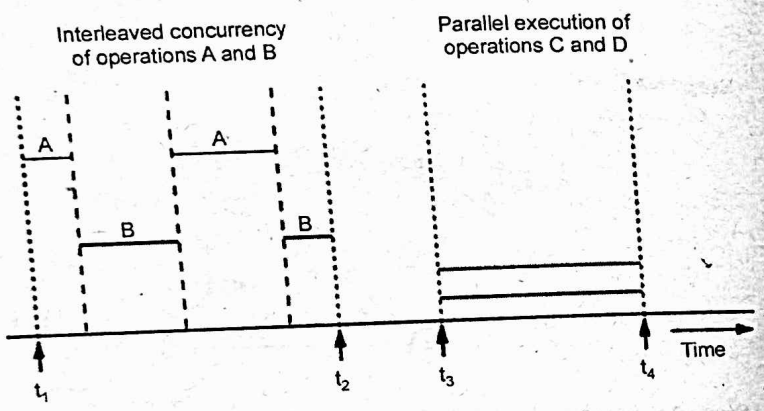


Fig.1. Interleaved concurrency versus parallel execution.

Fig. 2 Illustrates how reading and processing can proceed in parallel when the time required to process a disk block in memory is less than the time required to read he next block and fill a buffer. The CPU can start processing a block once its transfer to main memory is completed; at the same time, the disk I/O processor can be reading and transferring the next block into a different buffer. This technique is called double duffering and can also be used to read a continuous stream of blocks from disk to memeory. Double buffering permits continuous reading or writing of data on consecutive disk blocks, which eliminates the seek time and rotational delay for all but the first block transfer. Moreover, data is kept ready for processing, thus reducing the waiting time in the programs.

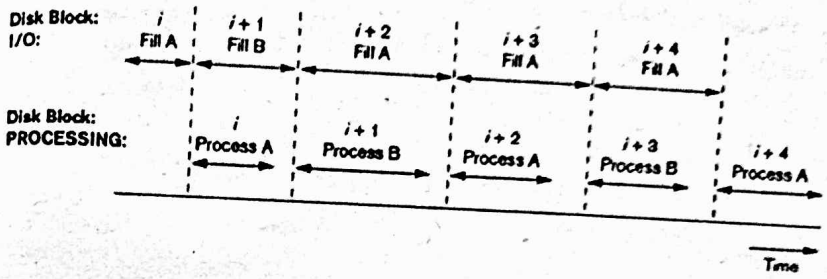


Fig. 2. Use of two buffers, A and B, for reading from disk.

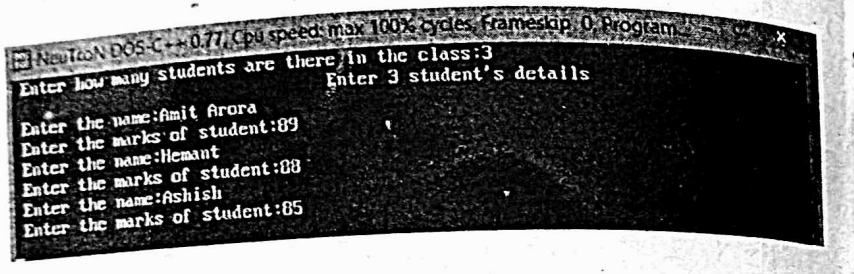
Q.8. (b) Write a C program to read name and marks of n number of students from user and store them in a file. (8.5)

Ans. Program to transfer student data in a file

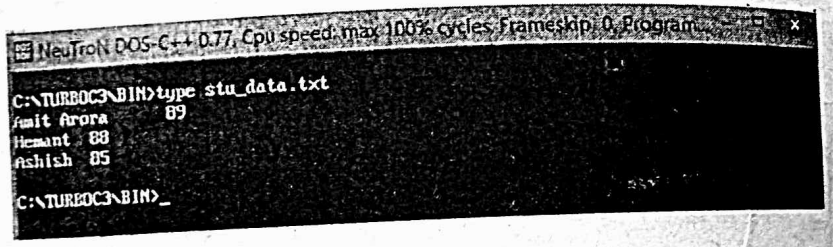
```
#include<stdio.h>
struct student
{
  char name[30];
  int marks;
};
int main()
{
  int i, n;
  struct student s[50];
  FILE *fp;
clrscr();
fp = fopen("Stu_data.txt", "w");
  printf(" Enter how many students are there in the class:");
  scanf("%d", &n);
  printf("\t\t\t\t Enter %d student's details\n", n);
  for(i=0;i<n;i++)
  {
    printf(" Enter the name:");
    fflush(stdin);
    gets(s[i].name);
    printf(" Enter the marks of student:");
```

```
    scanf("%d", &s[i].marks);
    fprintf(fp, "%s\t%d\n", s[i].name, s[i].marks);
    }
fclose(fp);
return 0;
}
```

Input:



Output:



Q.9. Explain any six functions that are used for handling sequential files in C language. **(12.5)**

Ans. Sequential File Handling Functions:

Function	Syntax	Example
fopen - Open file	FILE * fopen(char * filename, char*mode) Filename – string filename with path Mode – mode of opening r read only w write only	FILE * infile ; Infile = fopen("a.txt","r"); FILE *outfile = fopen("b.txt","w"); if(infile==NULL ||

	a append file
	+ read and write
	(r+/w+/a+)
	Return value – **FILE pointer** on success
	NULL on failure
		outfile==NULL)
		{
		{printf("error");
		}

fclose – close the open file	fclose(FILE *fptr) fptr – FILE * of the open file Returns void	fclose(infile);

fgetc – read one char from file	int fgetc(FILE *fptr)
	fptr – FILE * of the open file
	returns – character read
	-1 on eof or error
		while((ch=fgetc())!=-1)
		putchar(ch);
		//display content of file
		//on screen

fputc – write one character to the file	int fputc(int ch,FILE *fptr) ch – character to write fptr – open file pointer returns character written EOF if there is error	for(ch='A';ch<='Z';ch++) fputc(ch,fptr);

fseek – move the file pointer by given offset	int fseek(FILE*ptr,long offset,
	
	int whence)
	
	ptr – file pointer
	offset – offset in bytes from third parameter
	whence – SEEK_SET – from beginning of file
	SEEK_CUR – from current position
	SEEK_END – from end of file
	Returns – zero on success
	Non-zero on failure
		if(fseek(infile,4L,SEEK_SET)==0)
		{
		char ch=fgetc(infile);
		printf("The fourth char of the file is %c\n",ch);
		}

ftell – get the position of file pointer	int ftell(FILE *ptr)
	
	ptr – FILE pointer
	
	Returns – position of file pointer-1 on error
		FILE *ptr = fopen(b[1], "r");
		fseek(ptr,0L,SEEK_END);
		int size_of_file = ftell(ptr);

rewind–moves the file pointer to the beginning of the file	void rewind(FILE *ptr) ptr – FILE pointer.	rewind(infile); int n = fscanf(infile,"%d",&n);