Q.1(a) Give ADT for queue Data Structure. Discuss in brief applications for queue data structure. [5]

(A) Queues

Another important subclass of lists permits deletions to be performed at one end of a list and insertions at the other. The information in such a list is processed in the same order as it was received, that is, on a first-in, first-out (FIFO) or a first-come, first-served (FCFS) basis. This type of list is frequently referred to as a queue. Figure is a representation of a queue illustrating how an insertion is made to the right of the rightmost element in the queue, and how a deletion consists of deleting the leftmost element in the queue. In the case of a queue, the updating operation may be restricted to the examination of the last or end element. If no such restriction is made, any element in the list can be selected. The familiar and traditional example of a queue is a checkout line at a supermarket cash register. The first person in line in (usually) the first to be checked out.

Another perhaps more relevant example of a queue can be found in a time-sharing computer system where many users share the system simultaneously. Since such a system typically has a single central processing unit (called the processor) and one main memory, these resources must be shared by allowing one user's program to execute for a short time, followed by the execution of another user's program, etc., until there is a return to the execution of the initial user's program.

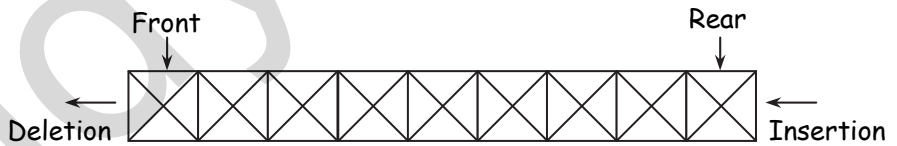


Fig. : Representation of a queue.

The user programs that are waiting to be processed form a waiting queue. This queue may not operate on a strictly first-in, first-out basis, but on some complex priority scheme based on such factors as what compiler is being used, the execution time required, the number of point lines desired, etc. The resulting queue is sometimes called a priority queue.

A final example of a queue is the line of cars waiting to proceed in some fixed direction at an intersection of streets. The deletion of a car corresponds to the first car in the line passing through the intersection, while an insertion to the queue consists of a car joining the end of the line of existing cars waiting to proceed through the intersection.

Q.1(b) Compare quicksort and radix sort on the basis of advantages and disadvantages? [5]

(A)

	Radix Sort	Quick Sort
1)	Negative numbers cannot be used.	Negative numbers can be used.
2)	It is done without recursion.	It can be done using recursion.
3)	It requires extra space other than the input.	It does not require any extra space.
4)	It is not efficient for bulk data.	It is efficient for bulk data.
5)	It uses priority queue.	It does not use priority queue.

Q.1(c) Discuss recursion, write in program in C++ using recursion for generating Fibonacci series. [5]

(A) Recursion

It is one of the design techniques where we try to replace the iterative code with the recursive one. It is the process of calling a function within itself till a terminating condition is not achieved. In many programs recursion makes the code compact. Some of the programs are basically recursive in nature so they should be solved recursively. The iterative solutions of such programs are quite tedious. Recursion though advantageous, proves inefficient with respect to time and space complexity. Calling the same function again and again leads to increase in execution time. For every next pass, the details of the previous pass are pushed on to the stack which increases the space requirements of a program. So recursion is always preferred when either there is no iterative solution for the problem or the iterative solution is tedious.

Tower of Hanoi

```
void towers (int n, char s, char d, char a,
{
    if (n = = 1)
      printf (" move disc from % c to % C \n", s, d, );
    else
    {
        towers (n – 1, s, a, d);
```

```
            towers ( 1, s, d, a);
            towers (n-1, a, d, s);
        }
    }
    void main ( )
    {
        int n;
        printf ("Enter no of disc");
        scanf ("%d", & n);
        towers (n, 'A', 'B', 'C');
    }
```

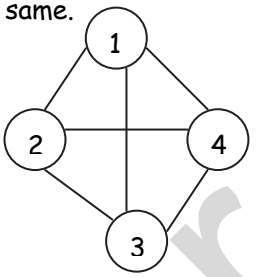
Fibonacci series program in C using recursion

```
#include<stdio.h>
int Fibonacci(int);
main( )
{
    int n, i = 0, c;
    scanf ("%d", &n);
    printf ("Fibonacci series\n");
    for (c = 1; c <= n; c++)
    {
        printf("&d\n", Fibonacci (i));
        i++;
    }
    return 0;
}
int Fibonacci (int n)
{
    if (n = = 0)
        return 0;
    else if (n = =  1);
        return 1;
    else
        return (Fibonacci(n – 1) + Fibonacci(n – 2));
}
```

Q.1(d) What is a graph? Explain methods to represent a graph. [5]

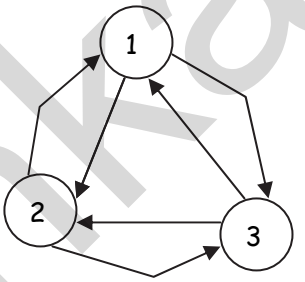
(A) Graph consists of two sets V and E of vertices and edges respectively. So G = (V,E) represents a graph.

In Un-directed Graph, the pairs (V1, V2) and (V2, V1) are same.
In Directed graph,
(V1, V2) = V1 → V2
(V2, V1) = V2 → V1

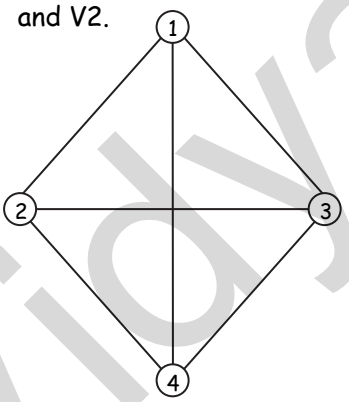


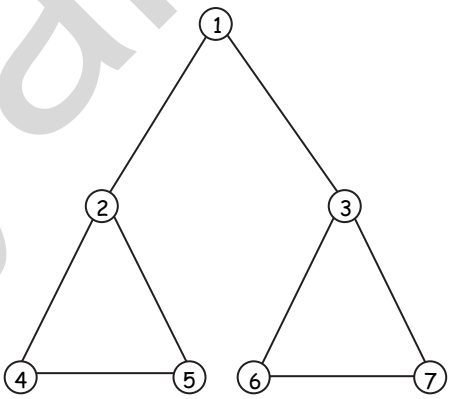
The graph shown in figure is an undirected graph and can be represented as,
V(G1) = {1,2,3,4}
E(G1) = {(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)}

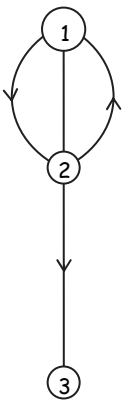
In un-directed graph of n nodes (vertices),
Maximum number of edges = n(n-1)/2
Now this figure shows a directed graph which can be represented as follows :
V(G2) = {1,2,3}
E(G2) = { <1,2>, <2,1>, <1.3>, <3,1>, <2,3>, <3,2> }



In directed graph of n nodes (vertices),
Maximum number of edges = n (n-1)
A 'n' vertices undirected graph with exactly n(n-1) /2 edges is said to be complete.
A 'n' vertices directed graph with exactly n(n-1) edges is said to be complete.
Two nodes V1 and V2 are said to be adjacent if there is an edge between V1 and V2.







Sub-Graph
Sub-graph of G is a graph whose all vertices as well as all edges are included in the Graph G.

Path
A path from Vk to Vi in the graph G is a sequence of vertices Vk, Vx, Vy, Vz, … Vi such that (Vk,Vx), (Vx, Vy), (Vy, Vz), …. Are edges in E(G).
Length of the path = number of edges included in it.

Simple Path

It is a path in which all vertices except possibly the first and the last are distinct.

Path 1 = (1,2) (2,4) (4,3) → 1,2,4,3

Path 2 = (1,2) (2,4) (4,2) → 1,2,4,2

Both the above paths are of length 3. Path 1 is a simple path but path 2 is not a simple path.

Cycle

A cycle is a simple path in which the first and last vertices are same,
e.g. 1,2,4,1 i.e. (1,2) (2,4) (4,1)

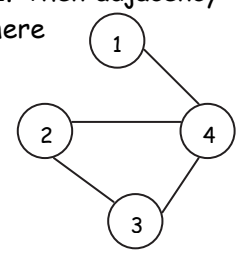
Connected Graph

A graph G is said top be connected if for every pair of distinct vertices Vi and Vj in V(G), there is a path from Vi to Vj in G.

Graph Representations

1) Adjacency Matrix

Let G = (V, E) be a graph of n vertices such that n >=1. Then adjacency matrix of G is a two-dimensional array of n x n say A where
A[i][j] = 1 if edge (Vi, Vj) is present in E(G)
A[i][j] = 0 if edge (Vi, Vj) is absent in E(G).



The graph shown in figure can be represented by,
V(G) = {1,2,3,4}
E(G) = { (1,4), (4,1), (2,3). (3,2), (2,4), (4,2), (3,4), (4,3) }

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

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

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

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

Adj2 = Adj * Adj (Boolean Matrix Multiplication)

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

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

2) Adjacency List Representation

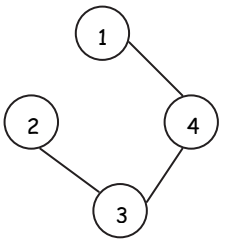
As we have seen earlier that Adjacency matrix representation gives complexity of O(n2). It is because even though fewer edges are present still the traversal cannot be completed till all the n2 elements are traversed.

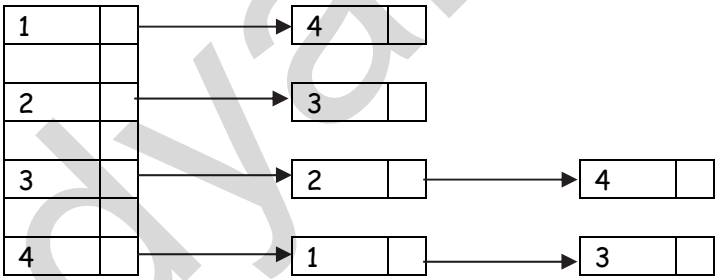
If the matrix is sparse i.e. not many edges are present then the earlier representation will result in huge amount of wastage of space. In such case, we can use the second type of representation for graph known as 'Adjacency List Representation'. With this the performance can be improved to O(e + n) where n is number of vertices and e is number of edges. In adjacency list representation, n rows of matrix are represented by 'n' liked list. Each linked list stores the nodes adjacent to it. Every node will have two fields namely vertex and link.

The class can be defined as follows :

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

The graph in the figure can be represented as follows:





Q.2(a) Write a program in C to implement quicksort algorithm. [8]

(A) Quick Sort

```
void quicksort (int x [ ], int lb, int ub)
{
    if (lb < ub)
    {
        p = partition (x, lb, ub);
        quicksort (x, lb, p–1);
        quicksort (x, pt, ub);
    }
```

```
}
int partition (int x [ ], int lb, int ub)
{
    int val, down , up, t;
    val = x[lb];
    down = lb + 1;
    up = up;
    while (down < = up)
    {
        while (down < = up && x [down] < val)
        down ++;
        while (x[up] > val)
        up – – ;
        if (down < up)
        {
            t = x [down];
            x [down] = x(up);
            x [up] = t;
        }
    }
}
```

Q.2(b) Write a function for generation of inorder, preorder and postorder transversal of a tree. [7]

(A) Tree Traversals

Considering the three parts of the binary tree i.e. root, left sub tree and right sub tree, there are three type traversals for a binary tree by keeping root at different positions.

1) Inorder : In this root is traversed in between the left and right sub tree.
2) Preorder : In this root is traversed prior to both the sub trees.
3) Postorder : In this root is traversed after both the subtrees.

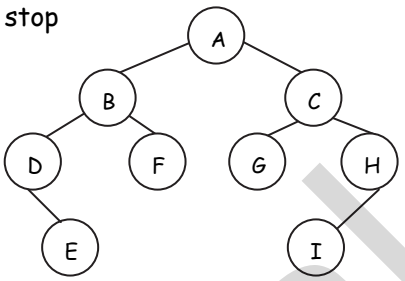
The algorithm can be given as follows.

Function inorder (root)

1) start
2) if root is not null then
   a) call inorder with left pointer of root (root->left)
   b) display the data in root.
   c) call inorder with right pointer of root. (root->right)
3) stop

Function preorder (root)
1) start
2) if root is not null then
   a) display the data in root.
   b) call preorder with left pointer of root (root->left)
   c) call preorder with right pointer of root. (root->right)
3) stop

Function postorder (root)
1) start
2) if root is not null then
   a) call postorder with left pointer of root (root->left)
   b) call postorder with right pointer of root. (root->right)
   c) display the data in root.
3) stop

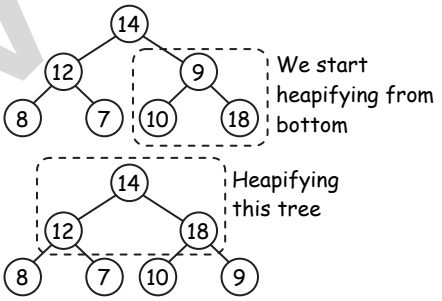


The various traversals for the above tree are as follows :
Inorder : D E B F A G C I H
Preorder : A B D E F C G H I
Postorder : E D F B G I H C A

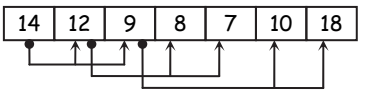
Let us see how the binary tree can be implemented using linked list implementation.

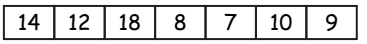
Q.2(c) Consider a list of numbers **[5]**
14,12,9,8,7,10,18
and sort them using heap sort.

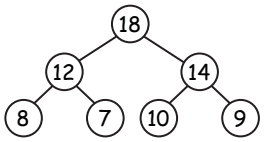
(A) Stage 1 : Heap Construction



Array representation of heap

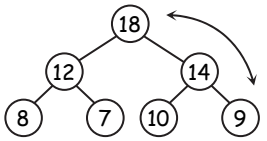






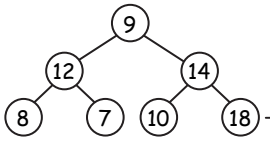
18	12	14	8	7	10	9

Stage 2: Maximum deletion

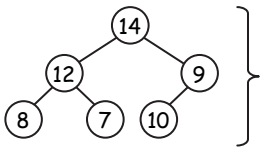


18	12	14	8	7	10	9

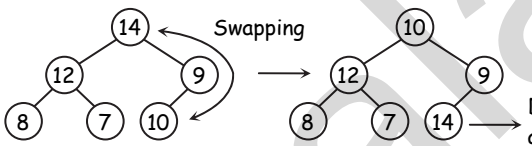
Swap with last node in heap



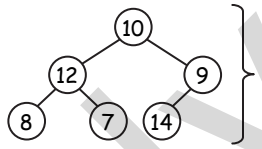
9	12	14	8	7	10	18



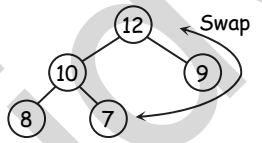
14	12	9	8	7	10	18



10	12	9	8	7	14	18

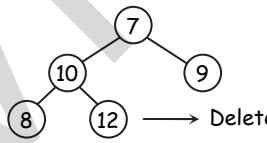


10	12	9	8	7	14	18



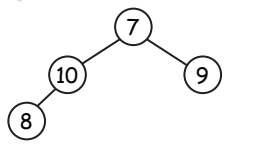
12	10	9	8	7	14	18

Heap to be processed



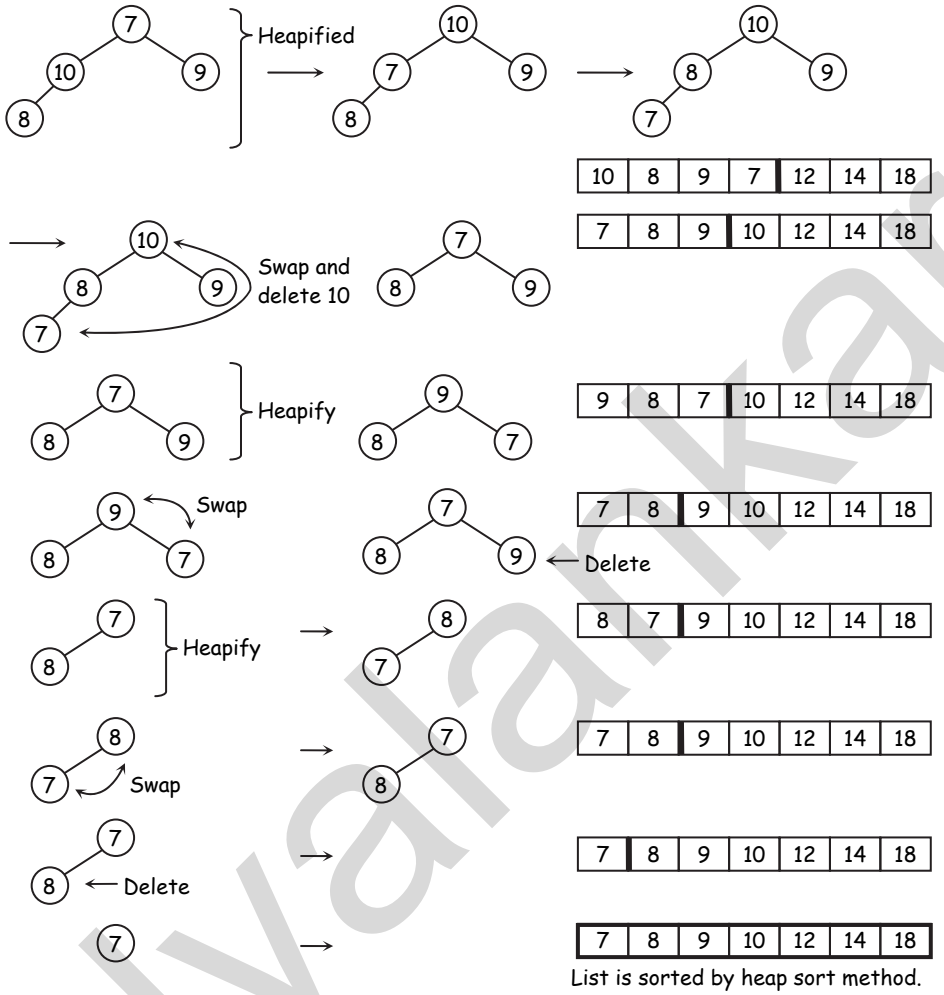
7	10	9	8	12	14	18

Heap to be processed



7	10	9	8	12	14	18

Heap to be processed



Q.3(a) Write a program in C to implement singly linked list supports the following operations **[15]**

1. **Insert a node in beginning**
2. **Insert a node in end**
3. **Deleting a node**
4. **Desplaying the list.**

(A)

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define NULL 0
struct Node
{
    int data;
    struct Node *next;
};
```

```
struct Node *p,*q,*newrec,*first,*last;

void insertElement(int x)
{
    int v,ch;
    struct Node *newrec=(struct Node*)malloc(sizeof(struct Node));
    newrec->data=x;
    newrec->next=NULL;
        if(first==NULL)
    {
        first=newrec;
        last=newrec;
    }
    else
    {
        printf("\nEnter Choice:\n1:At the Beginning\t2:At the End\t3:At
any other place\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: newrec->next=first;
                first=newrec;
                break;
            case 2: last->next=newrec;
                last=newrec;
                break;
            case 3: printf("\nEnter the value after which the node is to be
inserted:\n");
                scanf("%d",&v);
                p=first;
                while(p->data!=v && p!=NULL)
                    p=p->next;
                if(p==NULL)
                printf("\nInsertion not possible...\n");
                else
                {
                    newrec->next=p->next;
                    p->next=newrec;
                    if(p==last)
                    last=newrec;
                }
```

```
                    break;
                    default: printf("\nINVALID CHOICE\n");
                }
            }
        }

        void deleteElement(int x)
        {
            p=first;
            while(p->data!=x && p!=NULL)
                p=p->next;
            if(p==NULL)
            printf("\nDeletion not possible\n");
            else
            {
                if(p==first)
                    first=first->next;
                else
                {
                    q=first;
                    while(q->next!=p)
                    q=q->next;
                    if(p==last)
                    last=q;
                    else q->next=p->next;
                }
                printf("\nDeleted Node=%d\n",p->data);
                free(p);
            }
        }

        void display()
        {
            p=first;
            while(p!=NULL)
            {
                printf("%d ",p->data);
                p=p->next;
            }
        }
```

```
void main()
{
    int x,ch;
    clrscr();
    first=last=NULL;
    do
    {
        printf("\nEnter Choice:\n1:Insert\t2:Delete\t3:Display\t4:Exit\n");
scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\nEnter value to be inserted:\n");
                scanf("%d",&x);
                insertElement(x);
                break;
            case 2: printf("\nEnter value to be deleted:\n");
                scanf("%d",&x);
                deleteElement(x);
                break;
            case 3: display();break;
            case 4: printf("\nThank You...\n");break;
            default: printf("\nINVALID CHOICE\n");
        }
    }
    while(ch!=4);
    getch();
}
```

Q.3(b) Explain Huffmann Algorithm. **[5]**

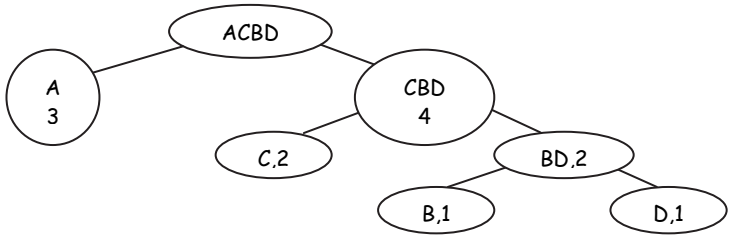
(A) Huffman Algorithm

A binary tree is very useful in Huffman algorithm. Huffman Algorithm is useful in encoding and decoding messages. It gives efficient codes for each character in message depending on its frequencies. So that a character with highest frequency gets smallest code and vice versa. Suppose we have a message ACBDACA. If we keep two digit codes for every symbol as

A	00
B	01
C	10
D	11

Then there are total 7 symbols hence the complete message will require 2 * 7 = 14 bits.

For the same message we can construct a Huffman Tree as follows.



The method of deriving codes for each symbol is to climb up till the root starting from the symbol. If climbed form left then 0 is added into code and if climbed from right then 1 is added to the code. Hence now codes for individual symbols can be derived as follows:

A	0
B	110
C	10
D	111

Then if calculated now the same message will require 11 bits. As the message length increases there would be significant reduction in number of bits.

Q.4(a) Explain file and different file handling operations? [8]

(A) File

Until now, we have been using the functions such as scanf and printf to read and write data. These are console oriented I/O functions which always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small. However, many real–life problems involve large volumes of data and in such situations; the console oriented I/O operations pose two major problems.

1) It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2) The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of files to store data. A file is a place on the disk where a group of related data is stored. Like most other languages, c supports a number of functions that have the ability to perform basic file operations, which include :

- naming a file,
- opening a file,
- reading data from a file,
- writing data to a file, and
- closing a file

Defining and Opening a File

If we want to store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include :

1) File name 2) Data structure 3) Purpose

Filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with the extension. Examples : Input .data

store
PROG. C
Student.c
Text.out

Data structure of a file is defined as **FILE** in the library of standard I/O function definitions. Therefore, all files should be declared as type **FILE** before they are used. **FILE** is a defined data type.

When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data. Following is the general format for declaring and opening a file :

```
File    *fp;
fp = fopen ("filename", "mode");
```

The first statement declares the variable **fp** as a "pointer to the data type **FILE**". As stated earlier, **FILE** is a structure that is defined in the I/O library. The second statement opens the file named filename and assigns an identifier to the **FILE** type pointer **fp**. This pointer which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The mode does this job. Mode can be one of the following :

r open the file for reading only.
w open the file for writing only
a open the file for appending (or adding) data to it.

When trying to open a file, one of the following things may happen :

1) When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2) When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.

3) If the purpose is 'reading', and if it exists, then the file is opened with the current contents safe; otherwise an error occurs.

Many recent compilers include additional modes of operation. They include ;

r+ The existing file is opened to the beginning for both reading and writing.

w+ Same as **w** except both for reading and writing.

a+ Same as **a** except both for reading and writing.

Closing a file

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. In case, there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. Another instance where we have to close a file is when we want to reopen the same file in a different mode. The I/O library supports a function to do this for us. It takes the following form :

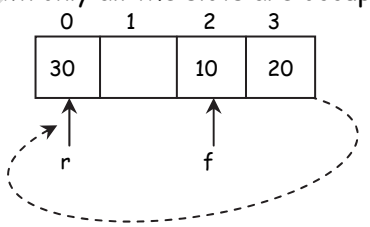
fclose (file_pointer);

Q.4(b) What is a circular queue, implement a circular queue? [7]

(A) Circular Queue

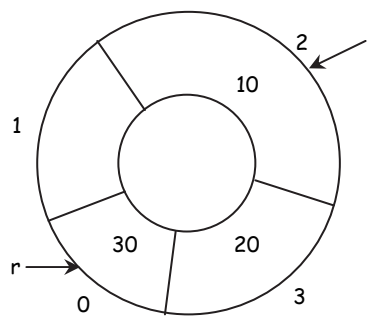
- It's a queue in which rear end and front end are connected together to make it circular.
- In circular queue, overflow is shown only all the slots are occupied.

e.g.



- Implementation

```
#include < stdio.h >
#define MAX 4
typedef struct
{
    int q[MAX], f, r, flag;
} queue;
void insert (queue * t, int ele)
{
    if (t → f = = (t → r + 1)%MAX && f → flag = = 1)
    {
```



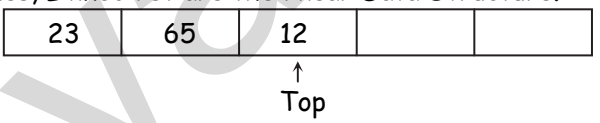
```
            Printf ("circular Queue Overflow\n");
            return;
        }
        t → flag = 1;
        t → r = (t → r + 1)% MAX;
        t → q [t → r] = ele;
}
int isempty (queue * t)
{
        if (t → flag = = 0)
            return 1;
        else return 0;
}
int delete1 (queue * t)
{ int z;
        if (isempty (t) = = 1)
        {
            Printf ("circular Queue Underflow\n");
                return – 1;
        }
        if (t → f = = t → r)
            t → flag = 0;
        z = t → q [t → f];
        t → f = (t → f + 1)% MAX;
            return z;
}
void display (queue * t)
{
        int i;
        if (isempty (t) = = 1)
        {
            Printf (" Circular Queue Empty \n");
            return;
        }
        i = t → f;
        while (1)
        {
            printf ("%d" t → q [t → f];
            if (t → r = = i)
                break;
            i = (i + 1)% MAX;
```

```
        }
        printf ("/n");
    }


    void main( )
    {
        queue x;
        x. f = 0;
        x. r = –1;
        x. flag = 0;

        insert (&x, 10);
        insert (&x, 20);
        display (&x);
        printf ("%d ", deletel (&x));
        display (&x);
    }
```

Q.4(c) Explain linear and non-linear data structures. [5]

(A) Linear and Non-linear Data Structure

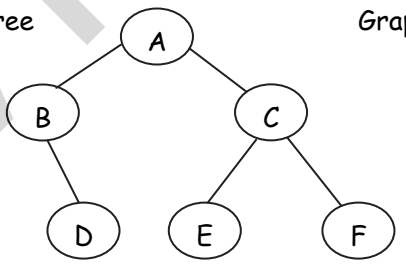
The Data Structure which are arranged in linear fashion i.e. One-One relation can be handled through linear data Structure. For example, Arrays, stacks, Queues, Linked list are the linear Data Structure.



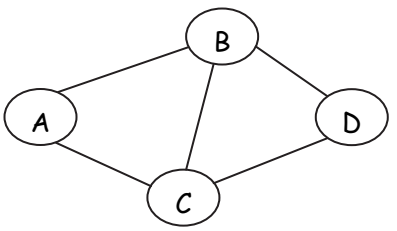
Stack

The Data Structure which establishes one-many OR many-many relations are called Non-linear Data Structure. In Non Linear Data Structure transversal techniques are used to access the data elements. The place at which the data is stored is called as **Node.**

Tree



Graph



Q.5(a) Explain DFS in detail with an example? [10]

(A) Depth First Search

The procedure of performing DFS on an undirected graph can be as follows : The starting vertex v is visited. Next an unvisited vertex w adjacent to v is selected and a depth first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex w adjacent to it and initiate a depth first search from w. the search terminates when no unvisited vertex can be reached from any of the visited ones.

Algorithm : Given an undirected graph G=(V,E) with n vertices and an array visited[n] initially set to false, this algorithm, dfs (v) visits all vertices reachable from v. Visited is a global array.

```
dfs (v)
visited [v] = true
for each vertex w adjacent to v
if not visited [w]
then dfs(w)
```

DFS

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

```
}
 printf (enter starting vertex \n");
 scanf (" %d", & node);
dfs(node –1, adj, n, visited);
}
```

Q.5(b) Explain BFS in detail with an example? [10]

(A) Breadth First Search

Starting at vertex v and making it as visited, BFS visits next all unvisited vertices adjacent to v. then unvisited vertices adjacent to there vertices are visited and so on.

Algorithm : A breadth first search of G is carried out beginning at vertex v as bfs (v). All vertices visited are marked as visited [i]=true. The graph G and arrat visited are global and visited is initialized to false. Initialize, addqueue, emptyqueue, deletequeue are the functions to handle operations on queue.

```
bfs (v)
initialize queue q
visited [v] = true
addqueue(q,v)
while not emptyqueue
    for all vertices w adjacent to v do
        if not visited [w] then
            addqueue (q,w)
            visited [w]=true
     deletequeue
```

Breadth First Search

```
void bfs (int node, int adj [ ] [ ], int n, int visted [ ])
{
 int q[20], f = –1, r = –1, nd, i;
 visited [node] = 1;
 q[++r] = node
 while (f ! = r)
 {
 nd = q[++ f ];
 printf ("%d", nd + 1);
 for (i = 0; i < n; i++)
  if (adj [nd] [i] = = 1 && visited [i] = = o)
   {
```

```
        visited [i] = 1;
        q [ + + r] = I;
         }
       }
    }
```

Q.6(a) Write a program in C to implement deletion of a node from binary tree. The program should consider all cases. [10]

(A) Insertion into a Search Tree

The first part of treesort is to build a sequence of nodes into a binary search tree. We can do so by starting with an empty binary tree and inserting one node at a time into the tree, always making sure that the properties of a search tree are preserved. The first case, inserting a node into an empty tree, is easy. We need only make root point to the new node. If the tree is not empty, then we must compare the key with the one in the root. If it is less, then the new node must be inserted into the left subtree; if it is more, then it must be inserted into the right subtree. If the keys are equal, then our assumption that no two nodes have the same key is violated.

Deletion from a Search Tree

We have already obtained an algorithm that adds a new node to the search tree, and it can be used to update the tree as easily as to build it from scratch. But we have not yet considered how to delete a node from the tree. If the node to be deleted is a leaf, then the process is easy: We need only replace the link to the deleted node by NULL. The process remains easy if the deleted node has only one subtree: We adjust the link from the parent of the deleted node to point to its subtree.

When the item to be deleted has both left and right subtrees nonempty, however, the problem is more complicated. To which of the subtrees should the parent of the deleted node now point ? What is to be done with the other subtree? This problem is illustrated in figure, together with one possible solution. (An exercise outlines another, sometimes better solution). What we do is to attach the right subtree in place of the deleted node, and then hang the left subtree onto an appropriate node of the right subtree.

To which node of the right subtree should the former left subtree be attached? Since every key in the left subtree precedes every key of the right subtree, it must be as far to the left as possible, and this point can be found by taking left branches until an empty left subtree is found.

Q.6(b) Hash the flowing using table of size 11 use any two collision resolution techniques [10]

23, 35, 10, 71, 67, 32, 1000, 18, 10, 90, 44.

(A) HASHING

Hashing is one of the most efficient searching techniques. But it uses large amount of extra space. It uses a hash function and creates its own hash table from the set of input values. Using the hash function, the position of the original value (key) within the hash table is calculated. While searching, the search element is also hashed using hash function to get the hash key. Then the corresponding hash table entry is checked. If the value is present then we declare success otherwise failure. But this can happen only in the absence of collision which is a rare case. A collision is occurred when two or more keys hashed to same hash key. This problem is handled by any of the collision handling techniques available and discussed later. Let us first summarize the important terms used in the concept of hashing.

1) Hash Table : It is the array which stores the input elements according to their hash keys.

2) Hash Function : It is the function used to convert an input value into its corresponding hash key which indicates the place for that key in hash table. So if k is the original key and 'h' is the hash function then h(k) is its hash key. One would like to have a hash function which is easy to compute and even capable of generating non-colliding hash keys.

3) Collision : The collision occurs when two or more keys hashed to the same hash key. Suppose k1 and k2 are two keys such that h(k1) = h(k2) then they result in collision. Its quite obvious that two keys cannot be stored at the same place in hash table, hence the first key arriving is stored at the proper place and the place of the second key is decided by the 'Collision Handling Technique' used.

Let us take some examples of Hash Functions.

1) Mid square

This function is calculated by squaring the original key and then selecting appropriate number of bits form the middle of the squared number to obtain the hash key. The number of bits to select depends on the size of the hash table. If 'k' bits are selected then the size of hash table is required to be minimum of 2^k. The size of the hash table has to be a power of 2 if this procedure is adopted.

Even we can select the middle digits to form the hash key. So if we decided to select 'k' digits then the hash table size should be 10^k.

For E.g.- Original Key = 23 Square = 529 Hash Key = 05
Original Key = 45 Square = 2025 Hash Key = 20

2) Modulo N

In this method we use a simple division technique. We decide a number N for division. Each key is divided by N and the remainder obtained is used as the hash key. If k is the original key then the function can be defined as follows.

h(k) = h % N

This will result in hash keys ranging from 0 to N-1 and hence needs to have hash table with size N. Preferably a prime number is selected as N.

For Example : Suppose N = 13
Original Key = 23 Hash Key = 10
Original Key = 45 Hash Key = 06

3) Folding

In this method, the original element is partitioned into several equal parts except the last one. These parts are then added together to form the hash key for that element. There are two ways for doing that addition namely shift folding and folding at the boundaries. In shift folding, except the last; all parts are shifted so that the least significant bit of each part lines up with the corresponding bit of the last part. The different parts are then added together to form the hash key. In the folding at boundaries method, the original element is folded at the boundaries and digits falling into the same position are added together.

Let us take one example, suppose the number to be hashed is 234764365

Dividing it by equal parts of length 2.

P1= 23, P2= 47, P3= 64, P4= 36, P5= 5

Shift Folding	Folding at the boundaries	
23	23	
47	74	Reverse of 47
64	64	
36	63	Reverse of 36
5	5	
175	229	

For the above number, the hash key generated by Shift folding would be 175 and by Folding at boundaries would be 229. As here the size of the table is difficult to predict, hence the hash key generated by this technique can be rehashed using some functions like Modulo N.

4) Digit Analysis

This method is very useful when all the elements to be stored are known in advance. Each element is interpreted as a number using some radix. Using this radix all the digits of all elements are examined and the digits having very skewed representation are deleted. This continues till the number of digits left is small enough to give an index of the array which can be used as hash key. The digits to drop are selected such that the hash keys generated would be as non-conflicting as possible.

For example
1205, 5287, 6247, 355, 7225, 6347, 7273, 23, 415, 347

Dropping the hundredths place
105, 587, 647, 55, 725, 647, 773, 23, 15, 47

Dropping the unit place
10, 58, 64, 5, 72, 64, 77, 2, 1, 4

❑ ❑ ❑ ❑ ❑