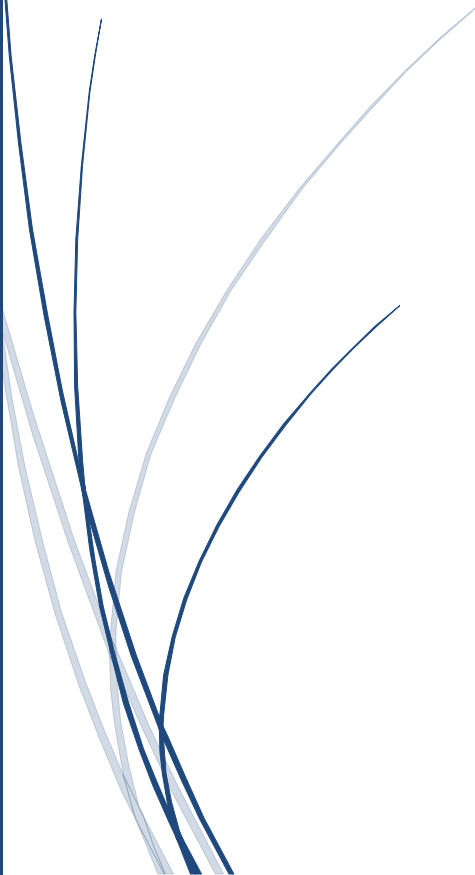




4/11/2018

Documentary of Data Structure

MADE BY : Md Rubel Abedin



Contents

Introduction:	4
Function:	6
Array:.....	7
One dimensional Array Or 1D Array :	7
Two dimensional Array Or 2D Array:	8
Stack:.....	11
Push and pop :	11
Queue:.....	13
enqueue and dequeue :	14
Linked list:	18
Singly Linked List:	18
Doubly Linked List:	19
Circular Linked List:	19
Structure:	20
Tree:	22
Important Terms:.....	22
Binary Tree:	23
Binary Tree Representations:.....	24
Binary Tree Traversals:.....	26
In - Order Traversal (left Child - root – right Child).....	27
Pre - Order Traversal (root – left Child – right Child).....	28
Post - Order Traversal (left Child – right Child - root)	29
Program to Create Binary Tree and display using In-Order Traversal	29
Binary Heap:.....	33
Min-Heap:	33
Max Heap:	36
Graph:	39
Depth First Search (DFS):	40
Breadth first search:.....	45
BFS algorithm:	45
BFS example:	45

BFS pseudo-code:.....	48
BFS code:.....	48
BFS in C :.....	48
Selection Sort:	55
Insertion sort in C :.....	57
Merge sort :.....	58
Binary Search:	65
Reference :	67

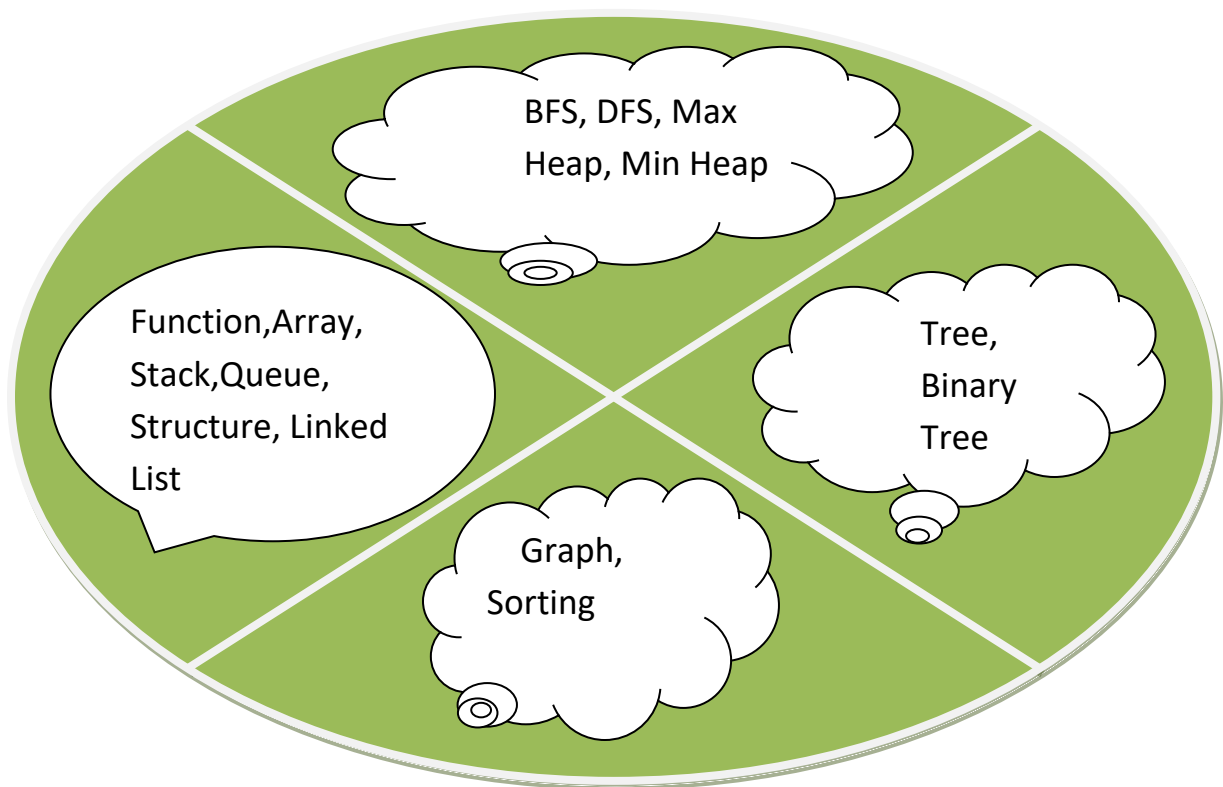
Introduction:

Data Structure is a main Theme or main knowledge of C Language. In Data Structure Course, we know about this topic. Which is below :

Before Mid Term Exam, we know the Topic . Which is :

- About Function
- About Array
 - ✓ 1D Array
 - ✓ 2D Array
- About Stack
- About Queue
- About Structure
- About Linked List

Show This as a graphically it like be a



Now we can Small Describe about this.

Function:

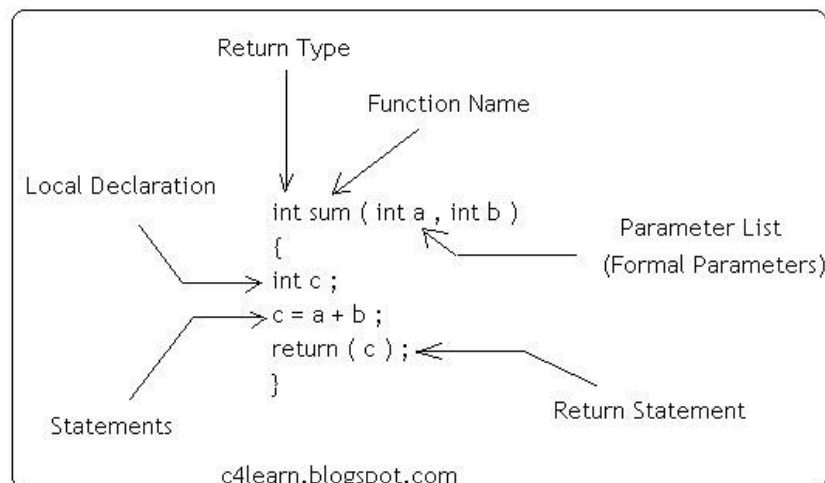
What is function? Everyone say it of his own language. But in C Language, function means that **some group of statement which works when it call**. A **function declaration** tells the compiler about a **function's** name, return type, and parameters. A **function definition** provides the actual body of the **function**.

For Example :

```
int rubelfactory (int product, int salary)
{
    int multiple = product*salary;
    printf("Given Salary : %d",multiple);
    return multiple;
}
```

Here, **rubelfactory** is the function name, before **rubelfactory** or function name given **int** which is **function return type** and after the **int product** and **int salary** which is **function parameters**.

Show The Bellow images clearly know about function.



Array:

Array is the most important section in C language. In C language, Array is the collection of Data which is the same type data and also accessing using a common name. Array works on likely dimensional. Array are three type. Which is :

- ✓ One dimensional Array.
- ✓ Two dimensional Array.
- ✓ Dynamic Array or Three dimensional Array.

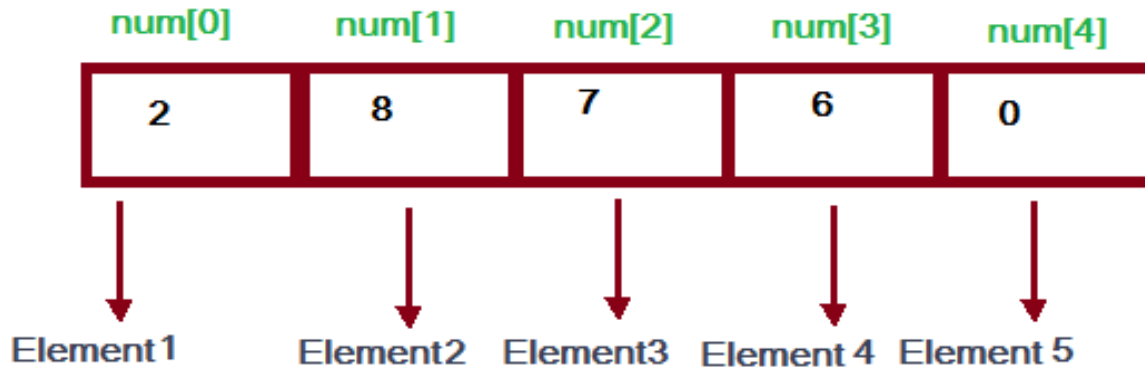
One dimensional Array Or 1D Array :

One Dimensional Array or 1D Array or Single Array is likely as a **linear array or a list**. Accessing its elements involves a **single** subscript which can either represent a row or column index. Most important think of array is array index start at 0 not 1.

For Example :

```
#include<stdio.h>

int main ()
{
    int num[5];
    num[4] = {2,8,7,6,0};
    return 0;
}
```



Here, **num** is an array and num is the 5 int type value. num[5] value firstly is 2 than 8 than 7 than 6 and lastly 0. As array count firstly 0 so the num index 0 or num[0] =2. Simillary num[1]=8, num[2]=7, num[3]=6, num[4]=0 it is input until of number of array declaration. num[5] means the here the num array has 5 block and this block also a value and num[5] of 5 is declaration of array size.

Two dimensional Array Or 2D Array:

Two Dimensional Array or 2D Array is likely as a **Matrix or a table**. A matrix can be represented as a table of rows and columns. We already know when we initialize a normal array (or you can say one dimensional array) during declaration, we need not to specify the size of it. However, that's not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration.

For Example :

```
#include<stdio.h>

int main()
{
```



```

int A[5][4];

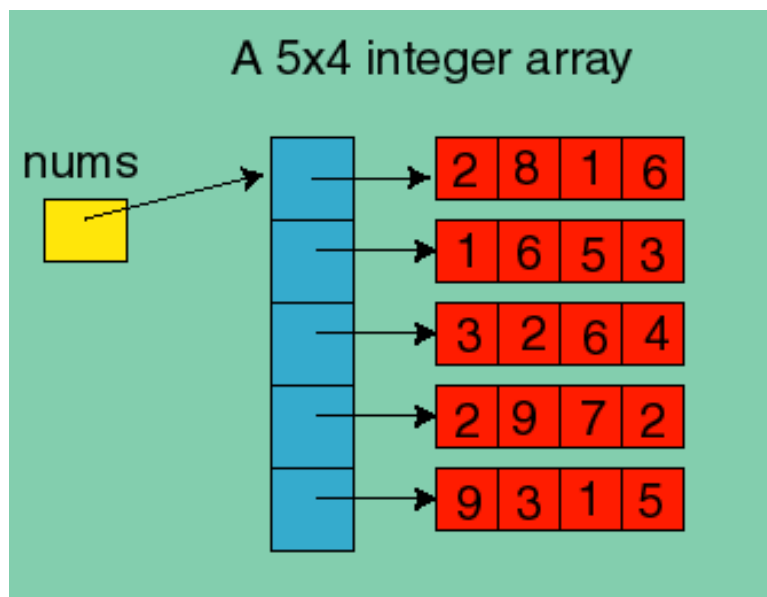
A[5][4]={          {2,8,1,6},
                   {1,6,5,3},
                   {3,2,6,4},
                   {2,9,7,2},
                   {9,3,1,5}
          };

for(i=0; i<5; i++)
{
    for(j=0; j<4; j++)
    {
        printf("%d", A[i][j]);
    }
}

return 0;
}

```

Output is Likely This :



Here, A is an array and this array also Two Dimensional Array because of a declaration. Here A [5][4] declaration means that in A array has 5 row and 4 coloum. This array size is $5 * 4 = 20$ its means than only an array has 20 integer type value.

Stack:

A **stack** is an array or list structure of function calls and parameters used in modern computer **programming** and CPU architecture. Elements in a **stack** are added or removed from the top of the **stack**, in a “last in first, first out” or LIFO order.

Stack are working two type of function. Which are :

- ❖ push ()
- ❖ pop()

push function is working in Stack when Any Element add on Array list. Its means that when add on value on array that time using on push function.

pop function is working when any element on Stack is remove on the list.

Push and pop :

For Example :

```
#include<stdio.h>

int head = 0, Stack [30];

void push ( int data )
{
    Stack[head]=data;
    head ++;
}

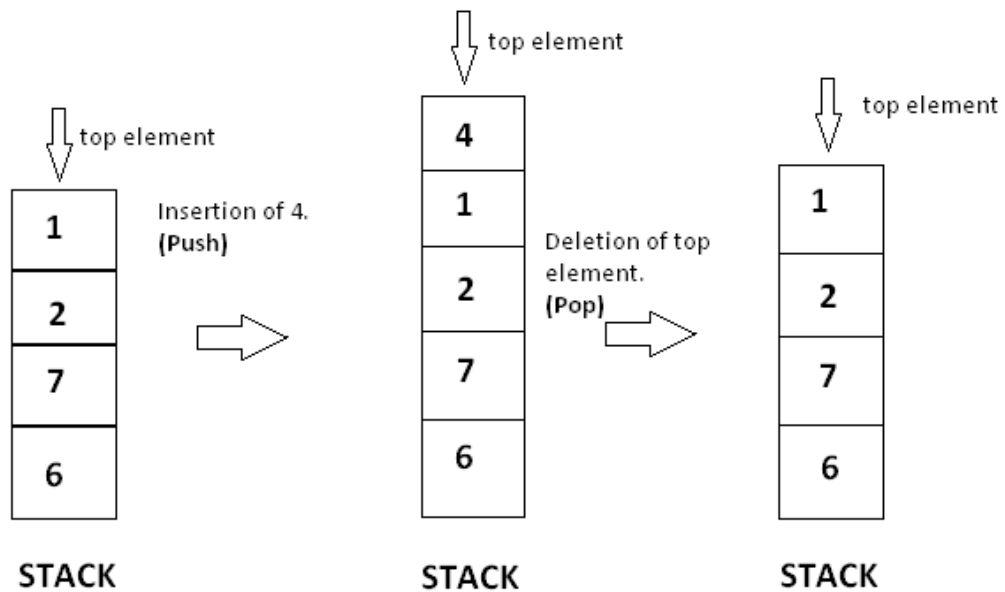
Void pop ()
{
    int value = Stack[head];
```

```

        head --;
    }
    int main()
    {
        push(6);
        push(7);
        push(2);
        push(1);
        push(4);
        pop();
        return 0;
    }

```

Output Likely This:

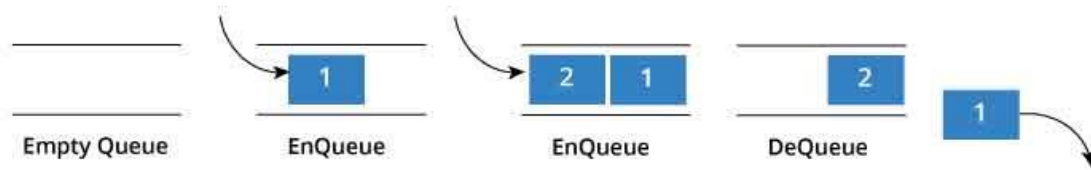


Here we see that firstly 4 elements are push on Stack than push another one which value is 4.so the last value 4 which we add on Stack. So it is the top value on the Stack. Now we pop on the stack and we see that the last value which we add on Stack this value is removing on the Stack. So the Stack list now 6, 7 , 2 , 1.

Queue:

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the **First In First Out (FIFO)** rule - the item that goes in first is the item that comes out first too.



In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".

enqueue and dequeue :

For Example:

```
#include<stdio.h>
#define SIZE 5

void enqueue(int);
void dequeue();
void display();

int items[SIZE], front = -1, rear = -1;

void enqueue ( int value )
{
    if(rear == SIZE-1)
    {
        printf("\nQueue is Full!!");
    }
    else
    {
        if(front == -1)
        front = 0;
        rear++;
        items[rear] = value;
        printf("\nInserted -> %d", value);
    }
}

void dequeue()
```

```

{
    if(front == -1)
    {
        printf("\nQueue is Empty!!");
    }
    else
    {
        printf("\nDeleted : %d", items[front]);
        front++;
        if(front > rear)
            front = rear = -1;
    }
}

```

```

void display(){
    if(rear == -1)
        printf("\nQueue is Empty!!!");
    else{
        int i;
        printf("\nQueue elements are:\n");
        for(i=front; i<=rear; i++)
            printf("%d\t", items[i]);
    }
}

```

```

int main()
{
    //deQueue is not possible on empty queue
    deQueue();

    //enQueue 5 elements
    enQueue(1);
    enQueue(2);
    enQueue(3);
    enQueue(4);
    enQueue(5);

    //6th element can't be added to queue because queue is full
    enQueue(6);

    display();

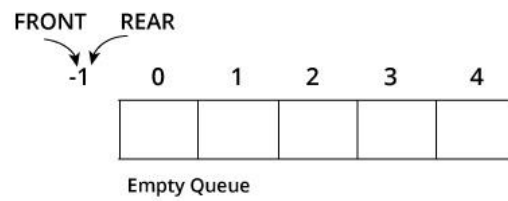
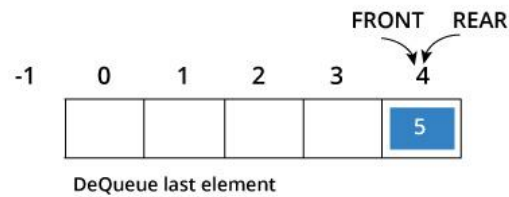
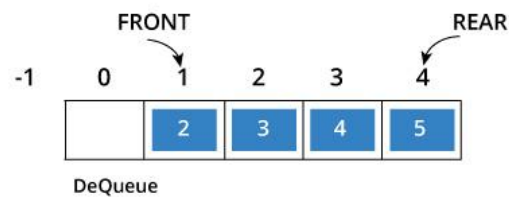
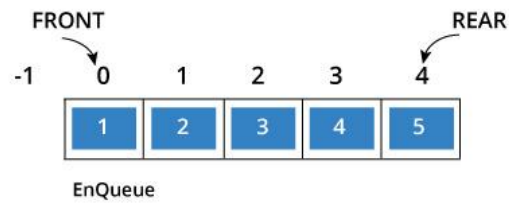
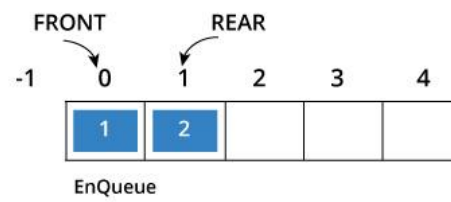
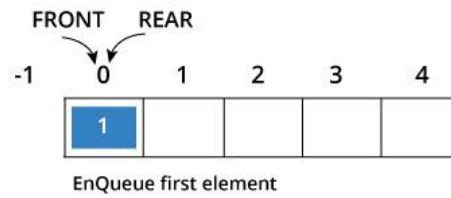
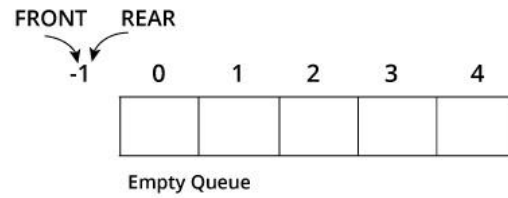
    //deQueue removes element entered first i.e. 1
    deQueue()
}

```

```
        //Now we have just 4 elements
        display();

        return 0;
    }
```

Output is likely that:



Linked list:

Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.

Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.

Linked Lists are used to create trees and graphs



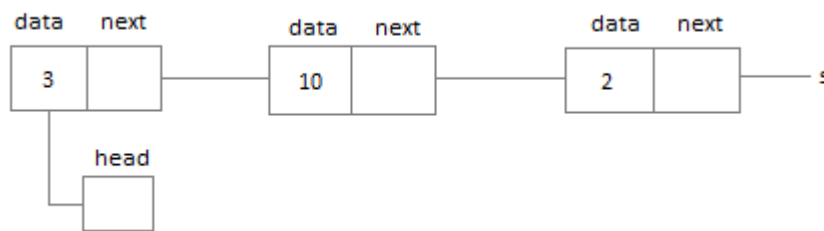
Types of Linked Lists

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Singly Linked List:

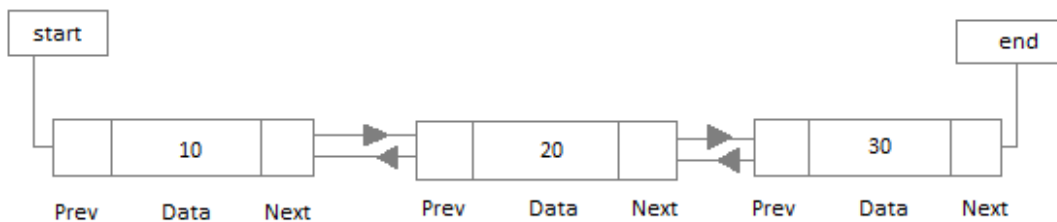
Singly linked lists contain nodes which have a **data** part as well as an **address part next**, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are **insertion, deletion and traversal**



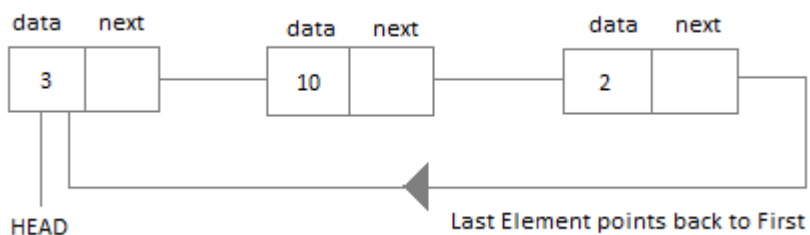
Doubly Linked List:

An a doubly linked list, each node contains a data part and two addresses, one for the **previous node** and one for the **next node**



Circular Linked List:

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain



Structure:

Structure is a collection of variables of different types under a single name

example:

I want to store some information about a person: his/her name, citizenship number and salary. I can easily create different variables name, citNo, salary to store these information separately.

in the future, you would want to store information about multiple persons. Now, i need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2

And i can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

Syntax of structure:

```
struct structure_name
{
    data_type member1;
    data_type member2;
    data_type member;
```

```
};
```

And we can create the structure for a person as mentioned above as:

```
struct person
{
    char name[50];
    int citNo;
    float salary;
};
```

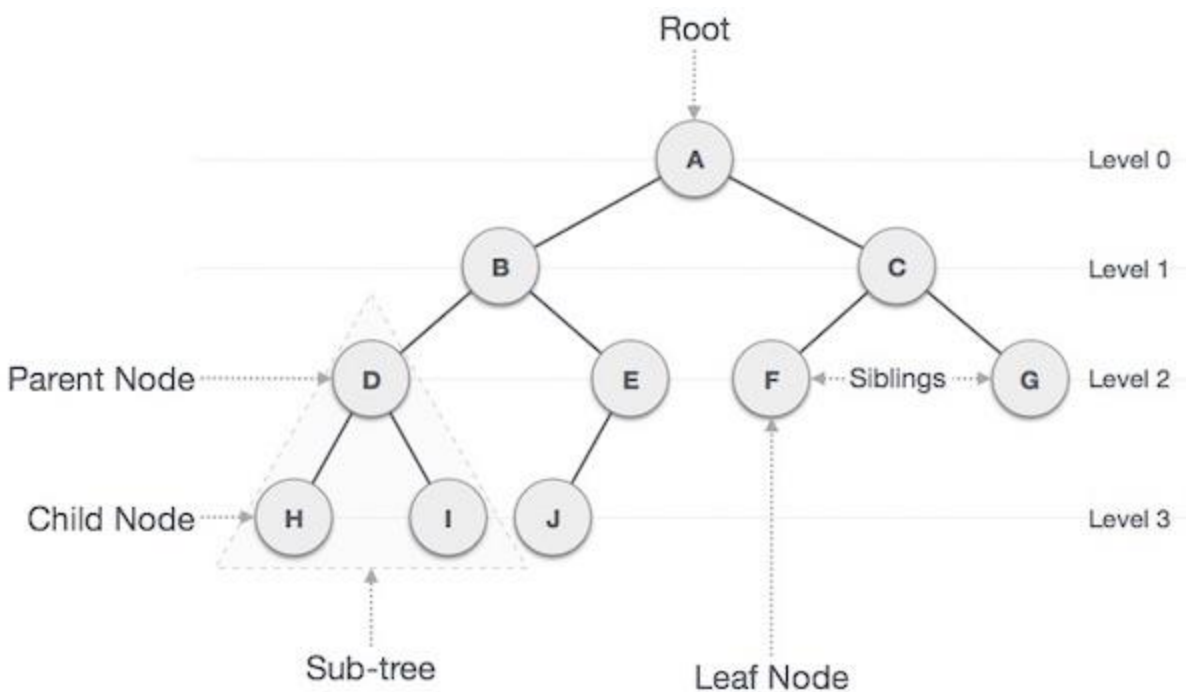
After creating structure we should declare variable like:

```
int main()
{
    struct person person1, person2, person3[20];
    return 0;
}
```

Tree:

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list..



Important Terms:

Following are the important terms with respect to tree.

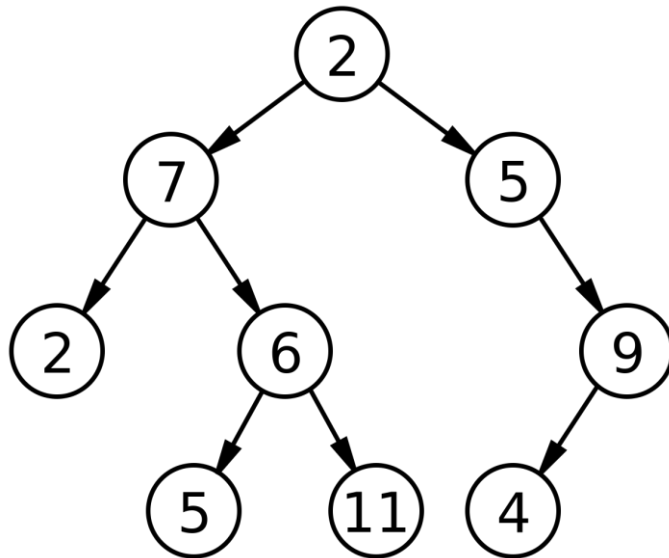
- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – the node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – any node except the root node has one edge upward to a node called parent.
- **Child** – the node below a given node connected by its edge downward is called its child node.
- **Leaf** – the node which does not have any child node is called the leaf node.
- **Sub tree** – Sub tree represents the descendants of a node.
- **Visiting** – Visiting refers to checking the value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Tree:

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.



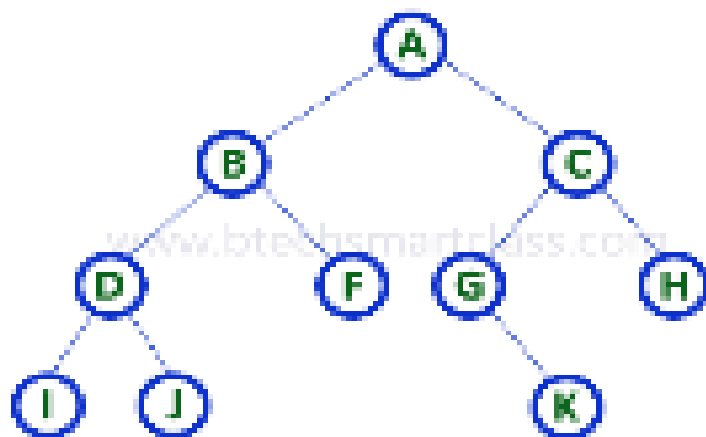
Binary Tree Representations:

A binary tree data structure is represented using two methods. Those methods are as follows...

Array Representation

Linked List Representation

Consider the following binary tree...



Array Representation:

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

Consider the above example of binary tree and it is represented as follows.

A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.

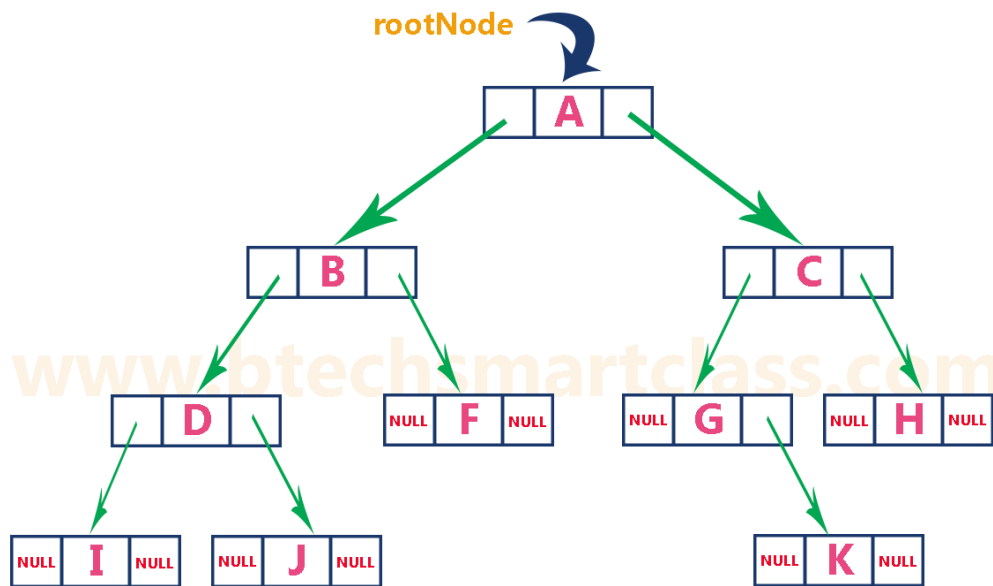
Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure.

Left Child Address	Data	Right Child Address
-------------------------------	-------------	--------------------------------

The above example of binary tree represented using Linked list representation is shown as follows.



Binary Tree Traversals:

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

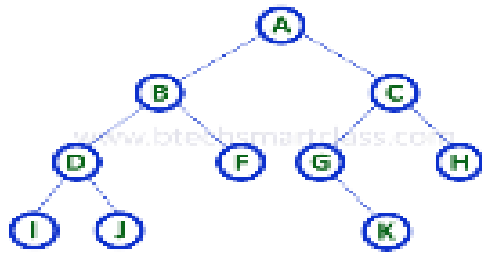
There are three types of binary tree traversals.

In - Order Traversal

Pre - Order Traversal

Post - Order Traversal

Consider the following binary tree.



In - Order Traversal (left Child - root – right Child)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all sub trees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left sub tree. so we try to visit its (B's) left child 'D' and again D is a root for sub tree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a sub tree with root C. So go for left child of C and again it is a sub tree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

Pre - Order Traversal (root – left Child – right Child)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all sub trees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

Post - Order Traversal (left Child – right Child - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most nodes is visited.

Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Program to Create Binary Tree and display using In-Order Traversal

```
#include<stdio.h>

#include<conio.h>

struct Node
{
    int data;
    struct Node *left;
    struct Node *right;
};

struct Node *root = NULL;

int count = 0;
```

```
struct Node* insert(struct Node*, int);
```

```
void display(struct Node*);
```

```
void main()
```

```
{
```

```
    int choice, value;
```

```
    clrscr();
```

```
    printf("\n----- Binary Tree ----- \n");
```

```
    while(1)
```

```
    {
```

```
        printf("\n***** MENU ***** \n");
```

```
        printf("1. Insert\n2. Display\n3. Exit");
```

```
        printf("\nEnter your choice: ");
```

```
        scanf("%d",&choice);
```

```
        switch(choice)
```

```
        {
```

```
            case 1: printf("\nEnter the value to be insert: ");
```

```
                scanf("%d", &value);
```

```
                root = insert(root,value);
```

```
                break;
```

```
            case 2: display(root);
```

```
                break;
```

```
            case 3: exit(0);
```

```

                                default: printf("\nPlease select correct operations!!!\n");
                                }
                        }
}

```

```

struct Node* insert(struct Node *root,int value)
{
    struct Node *newNode;

    newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->data = value;

    if(root == NULL)
    {
        newNode->left = newNode->right = NULL;

        root = newNode;

        count++;

    }
    else {
        if(count%2 != 0)
        {
            root->left = insert(root->left,value);
        }
        else
        {
            root->right = insert(root->right,value);
        }
    }
}

```

```
        }  
    }  
    return root;  
}
```

// display is performed by using Inorder Traversal

```
void display(struct Node *root)  
{  
    if(root != NULL)  
    {  
        display(root->left);  
        printf("%d\t",root->data);  
        display(root->right);  
    }  
}
```


Binary Heap:

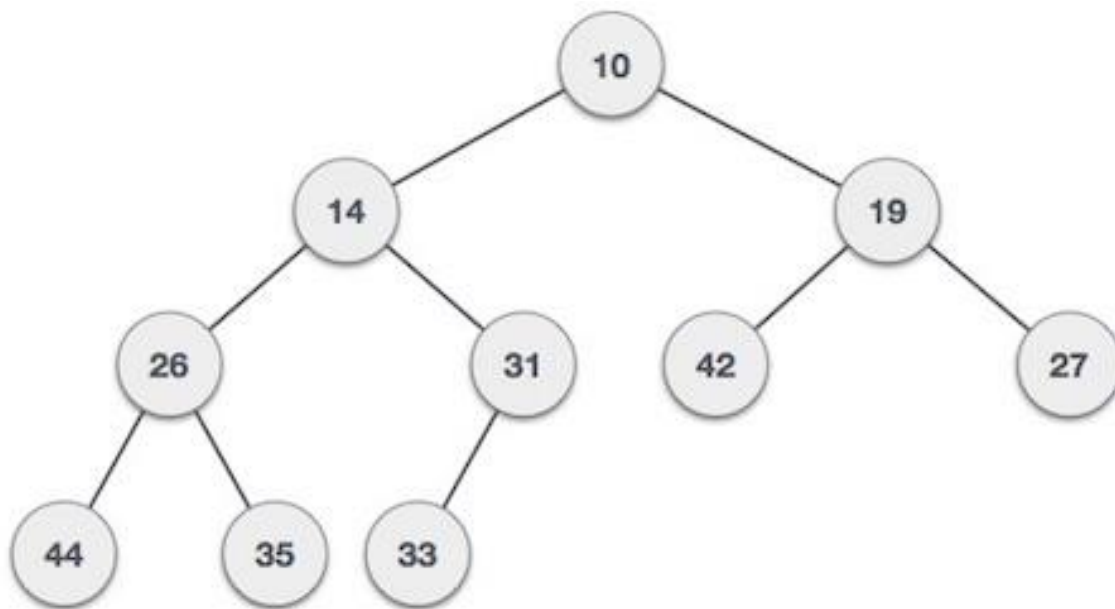
A Binary Heap is a Binary Tree with following properties.

1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min-Heap.

Min-Heap:

Where the value of the root node is less than or equal to either of its children.



```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#define max 100
```

```
int heap[max];
```

```
int size = 0;
```

```
void heapifyUp(){ // last element
```

```
int i = size;
```

```
while(1){ //
```

```
int parent = i/2;
```

```
if (parent >0&& heap[parent] > heap[i]){
```

```
int t = heap[parent];
```

```
    heap[parent] = heap[i];
```

```
    heap[i]=t;
```

```
    i = parent;
```

```
    } else {
```

```
break;
```

```
    }
```

```
    }
```

```
}
```

```
void heapifyDown(){ // top element
```

```
int i = 1;
```

```
while(i<size){
```

```
int c1 = 2*i;
```

```
int c2 = 2*i + 1;
```

```
int t;
```

```
if (c1 <= size) {
```

```
    t = c1;
```

```
    } else {
```

```
break;
```

```
    }
```

```
if (c2 <= size && heap[c1] > heap[c2]){
```

```
    t = c2;
```

```
    }
```

```
if(heap[i] >= heap[t]) break;
```

```
int temp = heap[i];
```

```
    heap[i] = heap[t];
```

```
    heap[t] = temp;
```

```
    i = t;
```

```
    }
```

```
}
```

```
void insert(int key){
```

```
    size = size + 1;
    heap[size] = key;
    heapifyUp();
}
```

```
int returnMin(){
return heap[1];

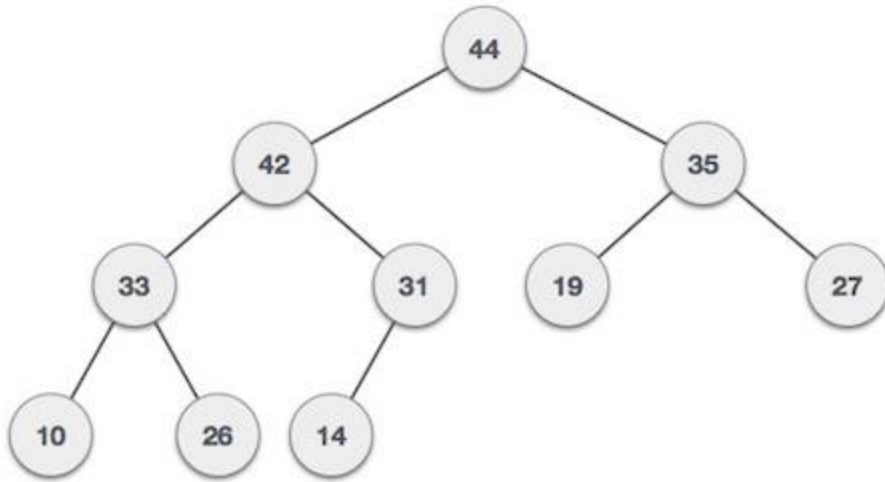
}
```

```
void printHeap(){
int i = 1;
while(i <= size){
    printf("%d ", heap[i]);
    i++;

}
```

Max Heap:

Where the value of the root node is greater than or equal to either of its children.



C code:

```
#include<stdio.h>

void main()
{
    int arr[10], no, i, j, c, heap_root, temp;
    printf("Input number of elements: ");
    scanf("%d",&no);
    printf("\nInput array values one by one : ");
    for(i=0; i < no; i++)
        scanf("%d",&arr[i]);
    for(i=1; i < no; i++)
    {
        c = i;
        do
        {
            heap_root =(c -1)/2;
            /* to create MAX arr array */
            if(arr[heap_root]< arr[c])
            {
```

```

temp = arr[heap_root];
arr[heap_root]= arr[c];
arr[c]= temp;
}
c = heap_root;
}
while(c !=0);
}

```

```

printf("Heap array : ");
for(i =0; i < no; i++)
printf("%d\t ", arr[i]);
for(j = no -1; j >=0; j--)
{
temp = arr[0];
arr[0]= arr[j];
arr[j]= temp;
heap_root =0;
do
{
c =2* heap_root +1;
if((arr[c]< arr[c +1])&& c < j-1)
c++;
if(arr[heap_root]<arr[c]&& c<j)
{
temp = arr[heap_root];
arr[heap_root]= arr[c];

```

```

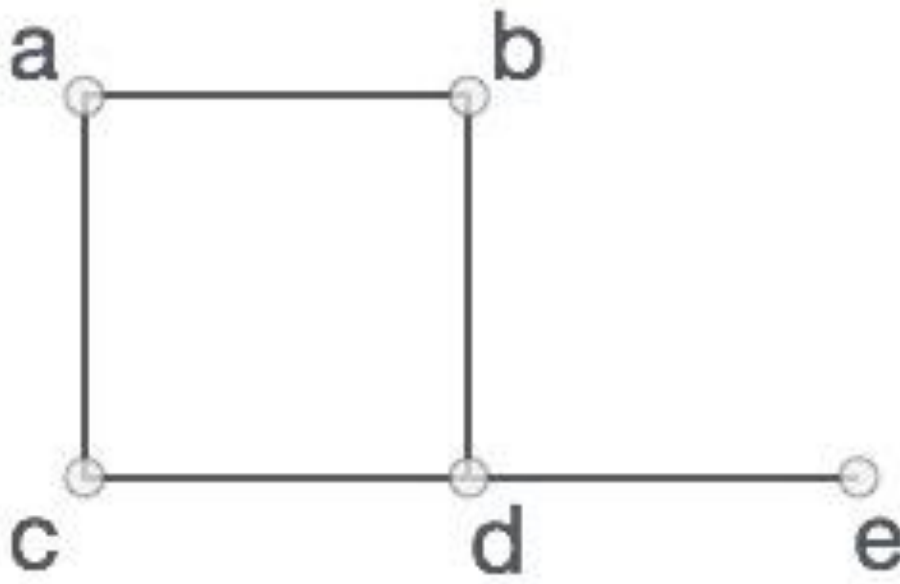
arr[c]= temp;
}
heap_root = c;
} while(c < j);
}
printf("\nSorted array : ");
for(i =0; i < no; i++)
{
printf("\t%d", arr[i]);
}
printf("\n");
return 0;
}

```

Graph:

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



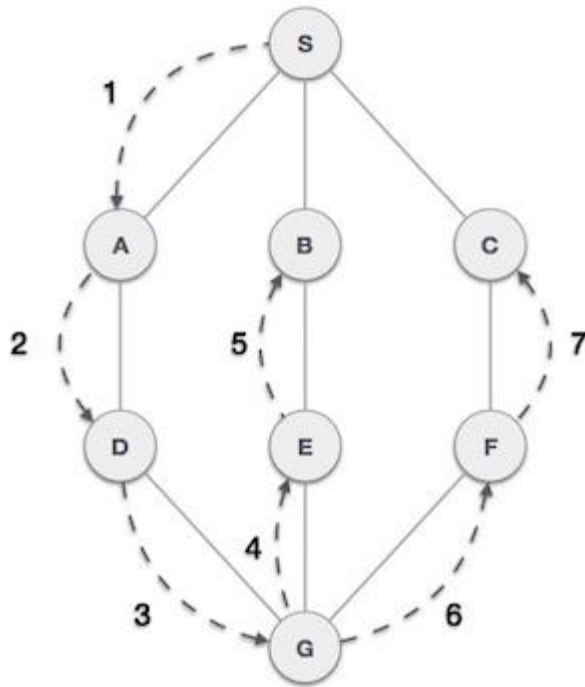
In the above graph,

$V = \{a, b, c, d, e\}$

$E = \{ab, ac, bd, cd, de\}$

Depth First Search (DFS):

Depth First Search (DFS) algorithm traverses a graph in a deathward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



C code:

```
#include<stdio.h>

void DFS ( int );

int G[10][10],visited[10],n; //n is no of vertices and graph is sorted in array G[10][10]

void main()
{
    int i,j;

    printf("Enter number of vertices:");

    scanf("%d",&n);

    //read the adjacency matrix
    printf("\nEnter adjacency matrix of the graph:");

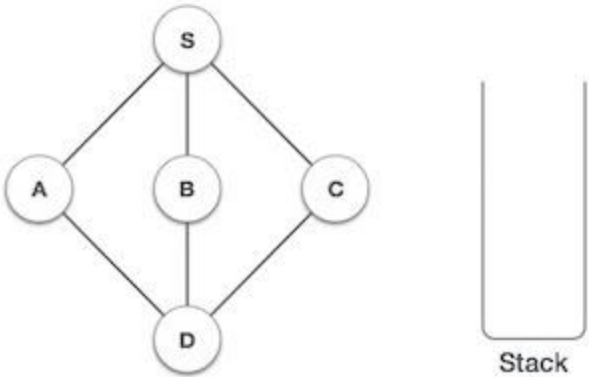
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&G[i][j]);
        }
    }

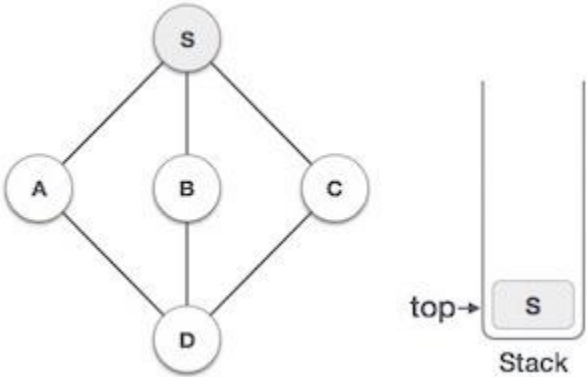
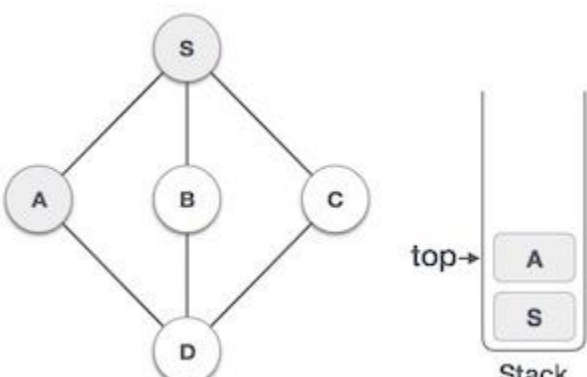
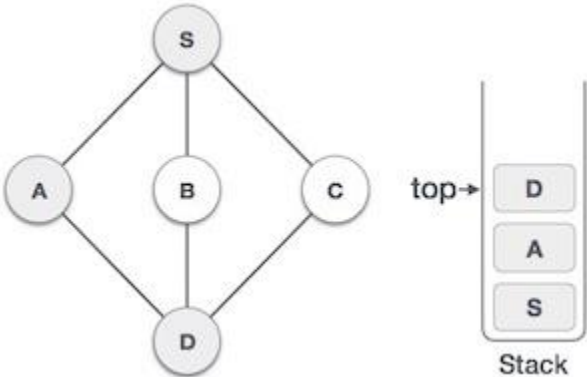
    for(i=0 ; i<n ; i++)
    {
```

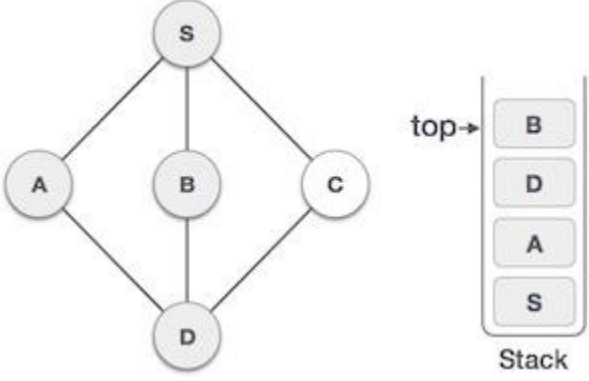
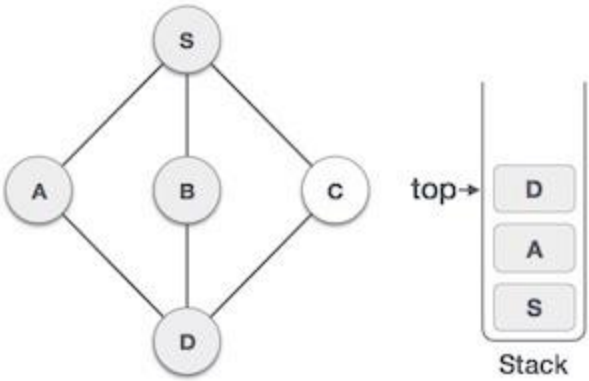
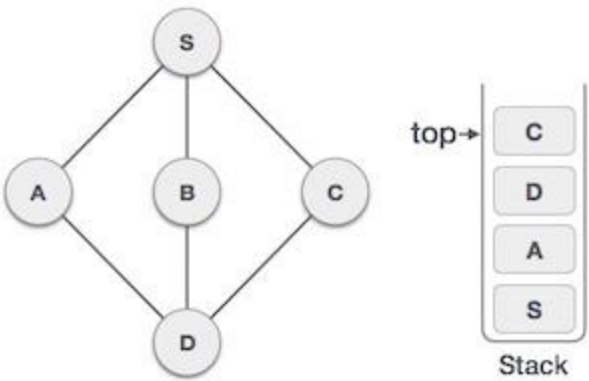
```

Visited [i] = 0 ; DFS(0);
}
}
void DFS( int i)
{
int j; printf("\n%d",i);
visited[i]=1;
for(j=0;j<n;j++)
{
if(!visited[j]&&G[i][j]==1) DFS(j);
}
}

```

Step	Traversal	Description
1		Initialize the stack.

2		<p>Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.</p>
3		<p>Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4		<p>Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.</p>

5		<p>We choose B, mark it as visited and put onto the stack.</p> <p>Here B does not have any unvisited adjacent node. So, we pop B from the stack.</p>
6		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.</p>
7		<p>Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.</p>

Breadth first search:

Traversal means visiting all the nodes of a graph. Breadth first traversal or Breadth first Search is a recursive algorithm for searching all the vertices of a graph or tree data structure. In this article, you will learn with the help of examples the BFS algorithm, BFS pseudocode and the code of the breadth first search algorithm with implementation in C++, C, Java and Python programs.

BFS algorithm:

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

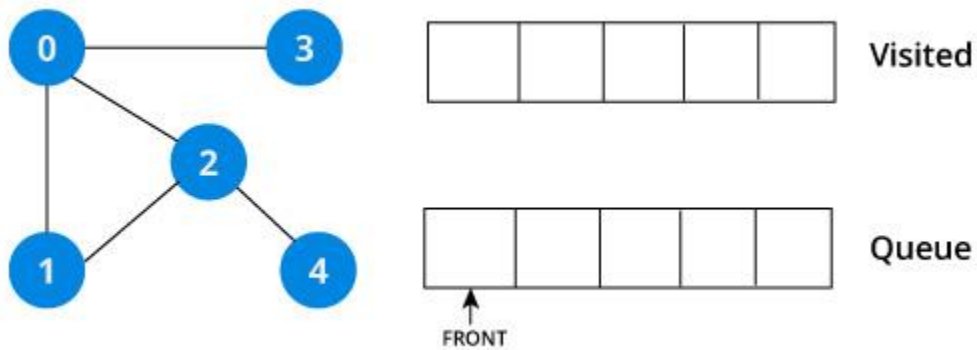
The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

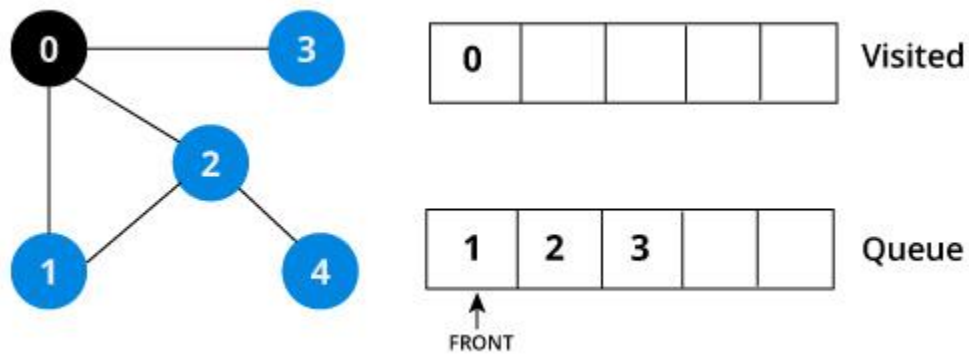
The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

BFS example:

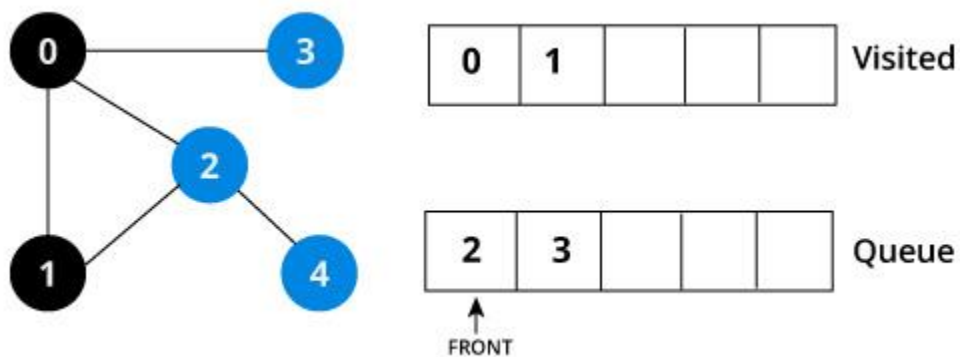
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



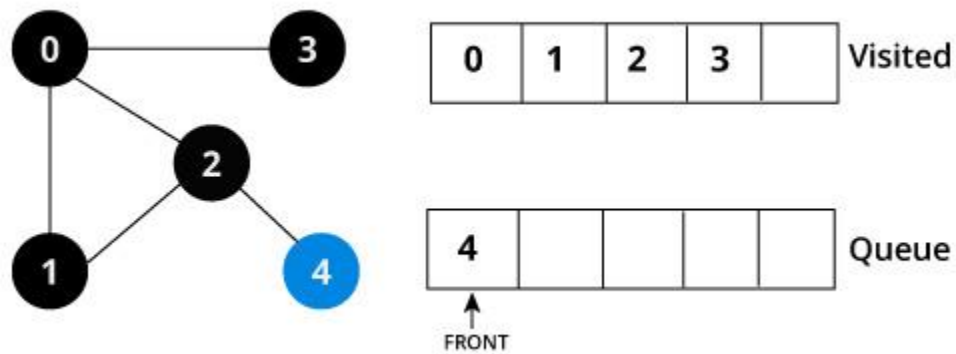
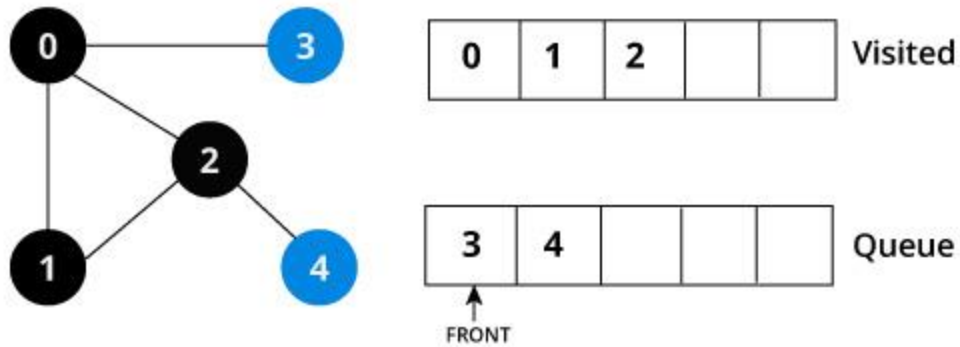
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



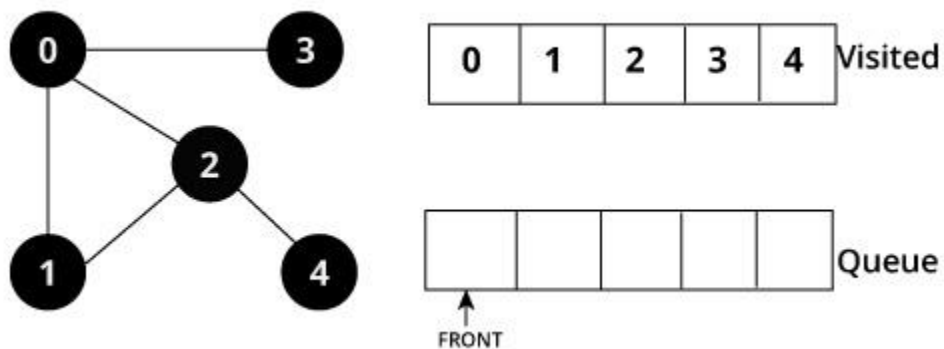
Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Since the queue is empty, we have completed the Depth First Traversal of the graph.

BFS pseudo-code:

```
create a queue Q

mark v as visited and put v into Q

while Q is non-empty

    remove the head u of Q

    mark and enqueue all (unvisited) neighbours of u
```

BFS code:

The code for the Breadth First Search Algorithm with an example is shown below. The code has been simplified so that we can focus on the algorithm rather than other details.

BFS in C :

```
#include<stdio.h>

#include<stdlib.h>

#define SIZE 40

struct queue
{
    int items[SIZE];
    int front;
    int rear;
```



```
};
```

```
struct queue* createQueue();  
void enqueue(struct queue* q, int);  
int dequeue(struct queue* q);  
void display(struct queue* q);  
int isEmpty(struct queue* q);  
void printQueue(struct queue* q);
```

```
struct node  
{  
    int vertex;  
    struct node* next;  
};
```

```
struct node* createNode(int);
```

```
struct Graph  
{  
    int numVertices;  
    struct node** adjLists;  
    int* visited;  
};
```

```
struct Graph* createGraph(int vertices);  
void addEdge(structGraph* graph, int src, int dest);  
void printGraph(structGraph* graph);
```

```
void bfs(structGraph* graph, int startVertex);
```

```
int main()
```

```
{
```

```
    struct Graph* graph = createGraph(6);
```

```
    addEdge(graph, 0, 1);
```

```
    addEdge(graph, 0, 2);
```

```
    addEdge(graph, 1, 2);
```

```
    addEdge(graph, 1, 4);
```

```
    addEdge(graph, 1, 3);
```

```
    addEdge(graph, 2, 4);
```

```
    addEdge(graph, 3, 4);
```

```
    bfs(graph, 0);
```

```
    return 0;
```

```
}
```

```
void bfs(structGraph* graph, int startVertex)
```

```
{
```

```
    struct queue* q = createQueue();
```

```
    graph->visited[startVertex] = 1;
```

```
    enqueue(q, startVertex);
```

```
    while(!isEmpty(q))
```

```

{
    printQueue(q);

    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);

    struct node* temp = graph->adjLists[currentVertex];

    while(temp)
    {
        int adjVertex = temp->vertex;

        if(graph->visited[adjVertex] == 0)
        {
            graph->visited[adjVertex] = 1;
            enqueue(q, adjVertex);
        }
        temp = temp->next;
    }
}

```

```

struct node* createNode(int v)
{
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
}

```

```
        return newNode;
    }
}
```

```
structGraph* createGraph(int vertices)
{
    structGraph* graph = malloc(sizeof(structGraph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}
```

```
void addEdge(structGraph* graph, int src, int dest)
{
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
}
```

```

graph->adjLists[src] = newNode;

// Add edge from dest to src
newNode = createNode(src);
newNode->next = graph->adjLists[dest];
graph->adjLists[dest] = newNode;
}

struct queue* createQueue()
{
    struct queue* q = malloc(sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

int isEmpty(struct queue* q)
{
    if(q->rear == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void enqueue(struct queue* q, int value)

```

```
{  
    if(q->rear == SIZE-1)  
    {  
        printf("\nQueue is Full!!");  
    }  
    else  
    {  
        if(q->front == -1)  
            q->front = 0;  
        q->rear++;  
        q->items[q->rear] = value;  
    }  
}
```

```
int dequeue(struct queue* q)  
{  
    int item;  
    if(isEmpty(q)){  
        printf("Queue is empty");  
        item = -1;  
    }  
    else  
    {  
        item = q->items[q->front];  
        q->front++;  
        if(q->front > q->rear)  
        {
```

```

        printf("Resetting queue");
        q->front = q->rear = -1;
    }
}
return item;
}

void printQueue(struct queue *q)
{
    int i = q->front;

    if(isEmpty(q))
    {
        printf("Queue is empty");
    }
    else
    {
        printf("\nQueue contains \n");
        for(i = q->front; i < q->rear + 1; i++)
        {
            printf("%d ", q->items[i]);
        }
    }
}

```

Selection Sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two sub arrays in a given array.

// C program for implementation of selection sort

```
#include <stdio.h>

void swap(int*xp, int*yp)
{
    inttemp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    inti, j, min_idx;

    // One by one move boundary of unsorted subarray
    for(i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for(j = i+1; j < n; j++)
            if(arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */

void printArray(int arr[], int size)
{
    int i;
    for(i=0; i < size; i++)
    {
        printf("%d ", arr[i]);
        printf("\n");
    }
}
```



```

}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}

```

Insertion sort in C :

Insertion sort in C: C program for insertion sort to sort numbers. This code implements insertion sort algorithm to arrange numbers of an array in ascending order. With a little modification, it will arrange numbers in descending order. Best case complexity of insertion sort is $O(n)$, average and the worst case complexity is $O(n^2)$.

Insertion sort algorithm implementation in C

/* Insertion sort ascending order */

```

#include <stdio.h>

int main()
{
    int n, array[1000], c, d, t;

    printf("Enter number of elements\n");
    scanf("%d", &n);

```

```

printf("Enter %d integers\n", n);
for (c = 0; c < n; c++)
scanf("%d", &array[c]);
for (c = 1 ; c <= n - 1; c++)
{
    d = c;
    while ( d > 0 && array[d-1] > array[d])
    {
        t = array[d];
        array[d] = array[d-1];
        array[d-1] = t;
        d--;
    }
}
printf("Sorted list in ascending order:\n");
for (c = 0; c <= n - 1; c++)
{
    printf("%d\n", array[c]);
}
return 0;
}

```

Merge sort :

Given that the merge function runs in $\Theta(n)$ time when merging n elements, how do we get to showing that merge Sort runs in $\Theta(n \log_2 n)$ time? We start by

thinking about the three parts of divide-and-conquer and how to account for their running times. We assume that we're sorting a total of n elements in the entire array.

The divide step takes constant time, regardless of the sub array size. After all, the divide step just computes the midpoint q of the indices p and r . Recall that in big- Θ notation, we indicate constant time by $\Theta(1)$.

The conquer step, where we recursively sort two sub arrays of approximately $n/2$ elements each, takes some amount of time, but we'll account for that time when we consider the sub problems.

The combine step merges a total of n elements, taking $\Theta(n)$ time.

If we think about the divide and combine steps together, the $\Theta(1)$ running time for the divide step is a low-order term when compared with the $\Theta(n)$ running time of the combine step. So let's think of the divide and combine steps together as taking $\Theta(n)$ time. To make things more concrete, let's say that the divide and combine steps together take cn time for some constant c .

To keep things reasonably simple, let's assume that if $n > 1$, n is greater than, 1, then n is always even, so that when we need to think about $n/2$, it's an integer.

(Accounting for the case in which n is odd doesn't change the result in terms of big- Θ notation.) So now we can think of the running time of merge Sort on an n -element sub array as being the sum of twice the running time of merge Sort on an $(n/2)$ -element sub array (for the conquer step) plus cn (for the divide and combine steps—really for just the merging).

Now we have to figure out the running time of two recursive calls on $n/2$ elements. Each of these two recursive calls takes twice of the running time of merge Sort on an $(n/4)$ -element sub array (because we have to halve $n/2$ plus $cn/2$ to merge. We have two sub problems of size $n/2$, and each takes $cn/2$ time to merge, and so the total

time we spend merging for sub problems of size $n/2$ is $2 \cdot cn/2 = cn$.
 $2 \cdot cn/2 = cn$, dot, c, n, slash, 2, equals, c, n.

Let's draw out the merging times in a "tree":

Computer scientists draw trees upside-down from how actual trees grow. A tree is a graph with no cycles (paths that start and end at the same place). Convention is to call the vertices in a tree its nodes. The root node is on top; here, the root is labeled with the n sub array size for the original n -element array. Below the root are two child nodes, each labeled $n/2$ to represent the sub array sizes for the two sub problems of size $n/2$.

Each of the sub problems of size $n/2$ recursively sorts two sub arrays of size $(n/2)/2$, or $n/4$. Because there are two sub problems of size $n/2$, there are four sub problems of size $n/4$. Each of these four sub problems merges a total of $n/4$ elements, and so the merging time for each of the four sub problems is $cn/4$. Summed up over the four sub problems, we see that the total merging time for all sub problems of size $n/4$ is $4 \cdot cn/4 = cn$.
 $4 \cdot cn/4 = cn$, dot, c, n, slash, 4, equals, c, n:

What do you think would happen for the sub problems of size $n/8$? There will be eight of them, and the merging time for each will be $cn/8$, for a total merging time of $8 \cdot cn/8 = cn$.
 $8 \cdot cn/8 = cn$, dot, c, n, slash, 8, equals, c, n:

As the sub problems get smaller, the number of sub problems doubles at each "level" of the recursion, but the merging time halves. The doubling and halving cancel each other out, and so the total merging time is cn at each level of recursion. Eventually, we get down to sub problems of size 1: the base case. We have to spend $\Theta(1)$ time to sort sub arrays of

size 1, because we have to test whether $p < r$, is less than, r , and this test takes time. How many sub arrays of size 1 are there? Since we started with n elements, there must be n of them. Since each base case takes $\Theta(1)$ time, let's say that altogether, the base cases take $\Theta(n)$ time:

Now we know how long merging takes for each sub problem size. The total time for merge Sort is the sum of the merging times for all the levels. If there are $\log_2 n$ levels in the tree, then the total merging time is $\log_2 n \cdot \Theta(n)$, dot, $\Theta(n)$. So what is $\log_2 n$? We start with sub problems of size n and repeatedly halve until we get down to sub problems of size 1. We saw this characteristic when we analyzed binary search, and the answer is $\log_2 n + 1$

2

$\log_2 n + 1$, equals, $\log_2 n$, plus, 1. For example, if $n=8$, equals, 8, then $\log_2 n + 1 = 4$

2

$\log_2 n + 1 = 4$, equals, 4, and sure enough, the tree has four levels: $n = 8, 4, 2, 1$. The total time for merge Sort, then, is $\Theta(n \log_2 n)$

2

$\log_2 n + 1$, left parenthesis, $\log_2 n$, plus, 1, right parenthesis. When we use big- Θ notation to describe this running time, we can discard the low-order term ($+1$) and the constant coefficient (c), giving us a running time of $\Theta(n \log_2 n)$

2

$\Theta(n \log_2 n)$, as promised.

One other thing about merge sort is worth noting. During merging, it makes a copy of the entire array being sorted, with one half in low Half and the other half in high Half. Because it copies more than a constant number of elements at some time, we say that merge sort does not work in place. By contrast, both selection sort and insertion sort do work in place, since they never make a copy of more than a constant number of array elements at any one time. Computer scientists like to consider whether an algorithm works in place, because there are some systems where space is at a premium, and thus in-place algorithms are preferred.

Merge sort is a sorting technique based on divide and conquer technique. With the worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms.

Implementation in C

We shall see the implementation of merge sort in C programming language here –

```
#include <stdio.h>
#define max 10;
int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];
void merging(int low, int mid, int high)
{
    int l1, l2, i;

    for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++)
    {
        if(a[l1] <= a[l2])
        {
            b[i] = a[l1++];
        }
    }
}
```

```

        else
        {
            b[i] = a[l2++];
        }
    }
    while(l1 <= mid)
        b[i++] = a[l1++];
    while(l2 <= high)
        b[i++] = a[l2++];
    for(i = low; i <= high; i++)
        a[i] = b[i];
}

void sort(int low, int high)
{
    int mid;
    if(low < high)
    {
        mid = (low + high) / 2;
        sort(low, mid);
        sort(mid+1, high);
        merging(low, mid, high);
    }
    else
    {
        return 0;
    }
}

```

```

    }

    int main()
    {
        int i;
        printf("List before sorting\n");
        for(i = 0; i <= max; i++)
        {
            printf("%d ", a[i]);
        }
        sort(0, max);
        printf("\nList after sorting\n");
        for(i = 0; i <= max; i++)
        {
            printf("%d ", a[i]);
        }
    }

```

If we compile and run the above program, it will produce the following result –

Output:

List before sorting :

10 14 19 26 27 31 33 35 42 44 0

List after sorting :

0 10 14 19 26 27 31 33 35 42 44

Binary Search:

This code implements binary search in C language. It can only be used for sorted arrays, but it's fast as compared to linear search. If you wish to use binary search on an array which isn't sorted, then you must sort it using some sorting technique say merge sort and then use the binary search algorithm to find the desired element in the list. If the element to be searched is found then its position is printed. The code below assumes that the input numbers are in ascending order.

C programming code for binary search

```
#include <stdio.h>

int main()
{
    int c, first, last, middle, n, search, array[100];
    printf("Enter number of elements\n");
    scanf("%d",&n);
    printf("Enter %d integers\n", n);
    for (c = 0; c < n; c++)
    {
        scanf("%d",&array[c]);
    }
    printf("Enter value to find\n");
    scanf("%d", &search);
    first = 0;
```

```

last = n - 1;
middle = (first+last)/2;
while (first <= last)
{
    if (array[middle] < search)
    {
        first = middle + 1;
    }
    else if (array[middle] == search)
    {
        printf("%d found at location %d.\n", search, middle+1);
        break;
    }
    else
    {
        last = middle - 1;
        middle = (first + last)/2;
    }
    if (first > last)
        printf("Not found! %d isn't present in the list.\n", search);
}
return 0;
}

```

Reference :

1. BtechSmartClass.com
2. tutorialpoint.com
3. Stackoverflow.com
4. W3resource.com
5. Programize.com
6. Programmingsimplified.com