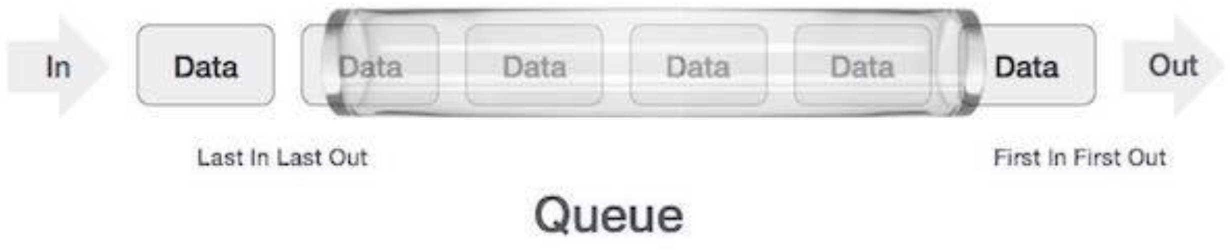
Queue:

- Linear data structure
- First in first out (FIFO/LILO)
- Priority queue



Basic Operations:

enque() : add (store)/ insert an item to the queue

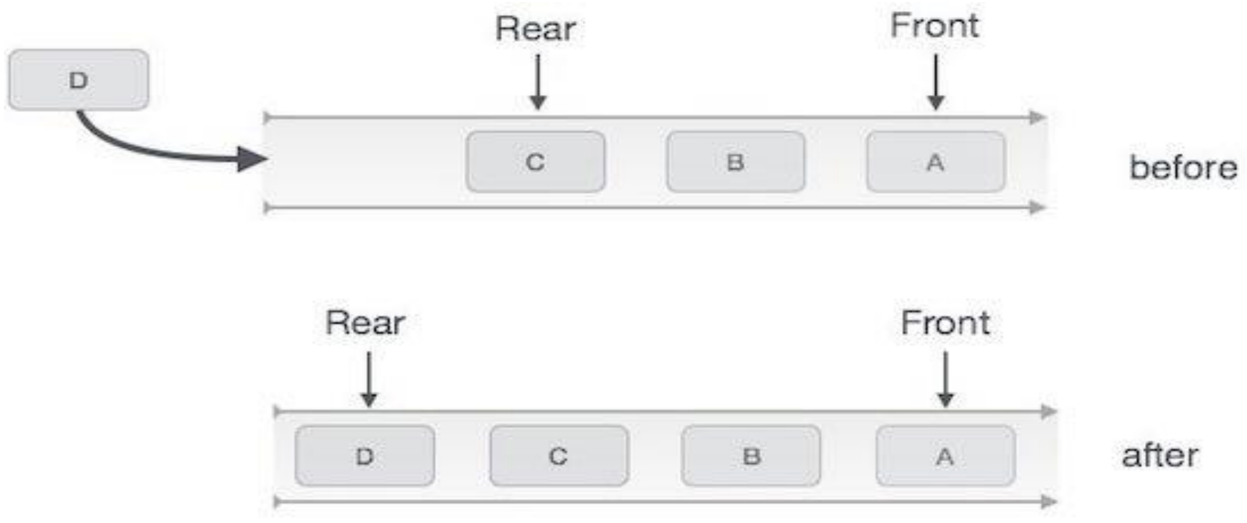
dequeuer() : remove an item from the queue

Enque Operations:

Step – 1: Check if queue is full

Step – 2: If queue is full procedure overflow error

Step – 3: If queue is not full create a temp node and assign node at last



Queue Enqueue

Enque Implementation:

```
void enqueue(int data)

{

        if(rear==full)

        {

                printf("Queue is full!\n");

                return;

        }

        rear++;

        queue[rear]=data;

        printf("%d is enqueue!\n",data);

}
```

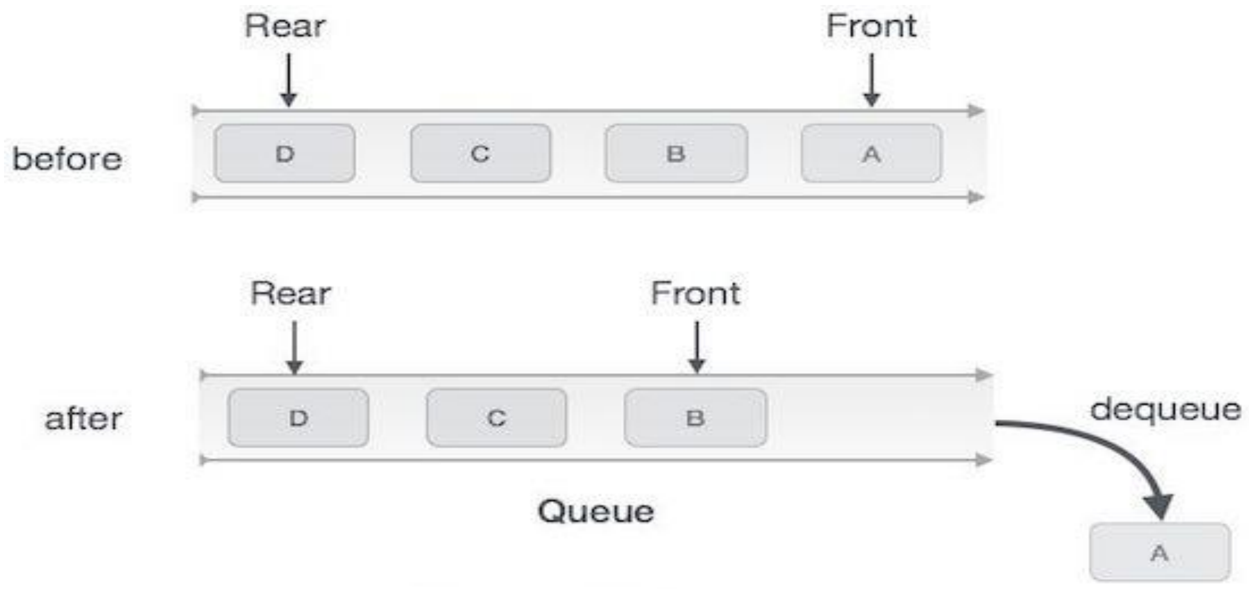
Dequeue Operation:

Step 1 − Check if the queue is empty.

Step 2 − If empty then produce underflow

Step 3 − If not then access the data where front is pointing.

Step 3 – Increment front pointer to point to the next available data element.



Queue Dequeue

Dequeue Implementation:

```
int dequeue()

{

        int temp;

        if(front==rear)

        {

                printf("Queue is empty!\n");

                return;

        }

        int data=queue[front];

        front++;

        return data;

}
```

Queue Implementation Using Linklist:

```
#include<stdio.h>

#include<stdlib.h>
```

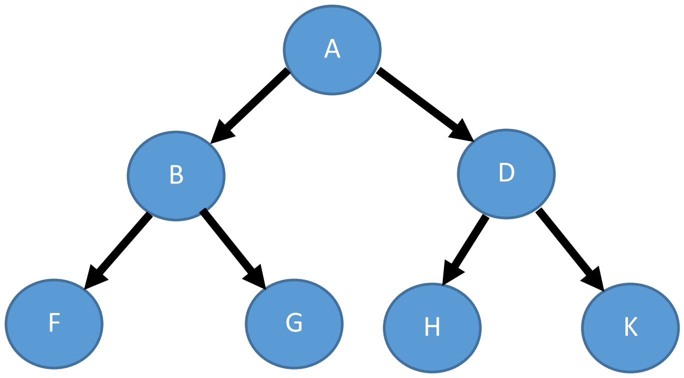
```
struct Node
{
	int data;
	struct Node *next;
};
typedef struct Node queue;
void enque(queue *q,int data)
{
	queue *temp;
	temp=(queue*)malloc(sizeof(queue));
	temp->data=data;
	temp->next=NULL;
	while(q->next!=NULL)
	{
		q=q->next;
	}
	q->next=temp;
	printf("%d is enqueued!\n",data);
}
int dequeue(queue *q)
{
	queue *temp;
	int data;
	if(q->next==NULL)
	{
		printf("Queue is empty!\n");
		return -1;
	}
```

```
        temp=q->next;
        data=temp->data;
        q->next=temp->next;
        free(temp);
        return data;
}
int main()
{
        queue *q;
        q=(queue*)malloc(sizeof(queue));
        q->next=NULL;
        enque(q,10);
        enque(q,5);
        enque(q,11);
        printf("%d\n",dequeue(q));
        printf("%d\n",dequeue(q));
        printf("%d\n",dequeue(q));
        return 0;
}
```

TREE:

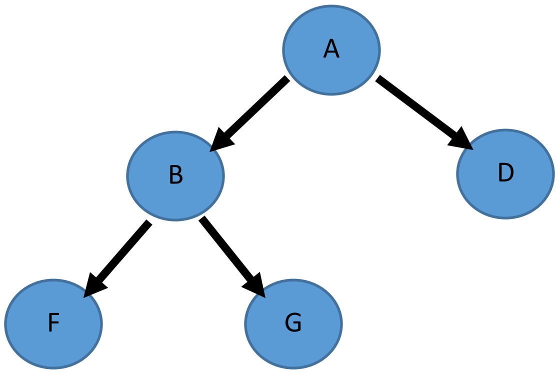
Full Binary Tree:

Binary tree with required number of child as per level



Complete Binary Tree:

Binary tree but childs from left to right



Array to Tree:

0	1	2	3	4
7	5	4	6	9

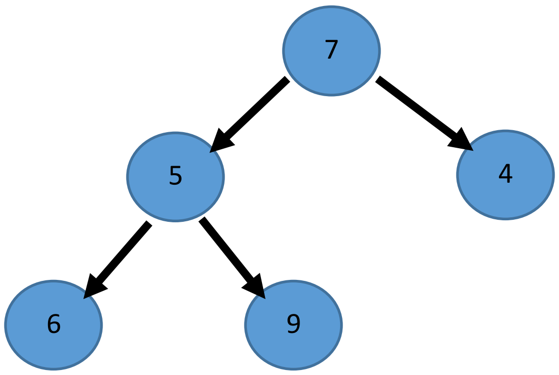
- → X[i] = root;   i=0
- → Left will be = x[2i+1]

  When i=0   x[1]=left

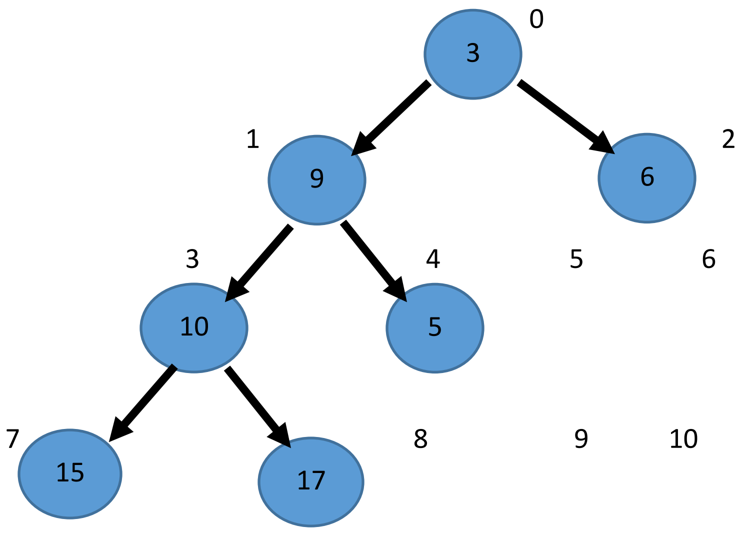
  When i=1   x[3]=left
- → Left will be = x[2i+2]

  When i=0   x[2]=right

  When i=1   x[4]=right



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



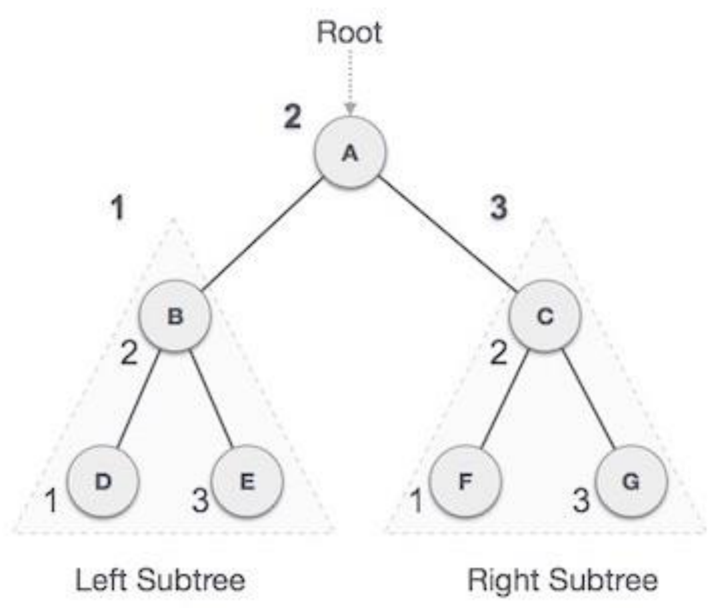
Tree Traversal:

Traversal is a process to visit all the nodes of a tree.

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

In-order Traversal:

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.



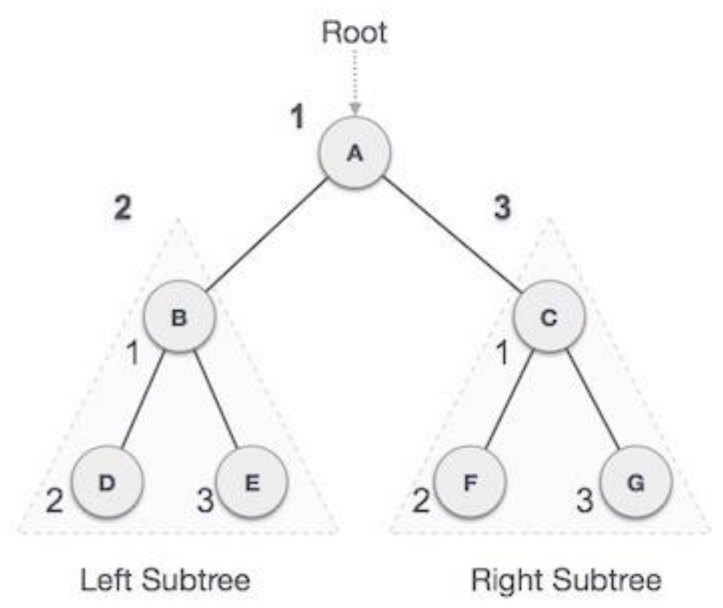
D → B → E → A → F → C → G

In-order Implementation:

```
void inorder(tree *t)
{
        if(t)
        {
                inorder(t->left);
                printf("%d\n",t->data);
                inorder(t->right);
        }
}
```

Pre-order Traversal:

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



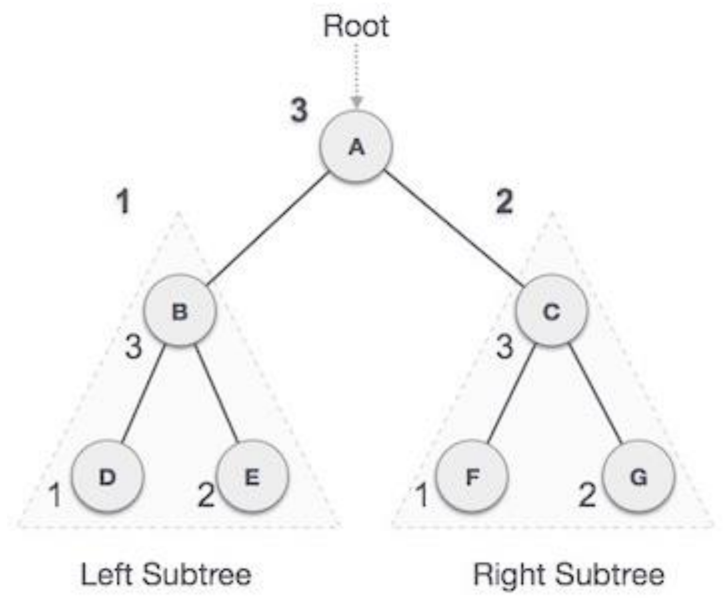
A → B → D → E → C → F → G

Pre-order Implementation:

```
void preorder(tree *t)

{

        if(t)

        {

                printf("%d\n",t->data);

                preorder(t->left);

                preorder(t->right);

        }

}
```

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.



D → E → B → F → G → C → A

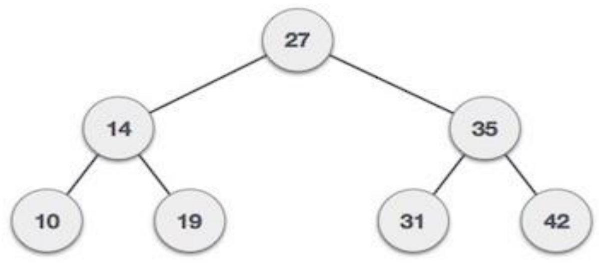
Post-order Implementation:

```
void postorder(tree *t)

{

        if(t)

        {

                postorder(t->left);

                postorder(t->right);

                printf("%d\n",t->data);

        }

}
```

Binary Search Tree (BST):

→ Left child < Node

→ Right child > Node



Basic Operations:

Insert:

To insert a data element first search from the root node. If the data is less than the key value then search an empty location in the left subtree and insert the data. Otherwise, search empty location in the right subtree and insert the data.

Implementation:

```
void insert(tree **t, int val)

{

        tree *temp=NULL;

        if(!(*t))

        {

                temp=(tree*)malloc(sizeof(tree));

                temp->left=temp->right=NULL;

                temp->data=val;

                *t=temp;

                return;

        }

        if(val<(*t)->data)

        {

                insert(&(*t)->left,val);

        }

        else if(val>(*t)->data)

        {

                insert(&(*t)->right,val);

        }

}
```

Search:

Whenever searching an element start from the root node. Then if the data is less than the key value, then search in the left subtree. Else if the data is greater than the key value search in the right subtree.

Search Implementation:

```
void search(tree *t,int data)
{
        if(t!=NULL)
        {
                if(t->data==data)
                {
                        printf("\nYes\n");
                }
                else if(t->data > data)
                {
                        search(t->left,data);
                }
                else if(t->data < data)
                {
                        search(t->right, data);
                }
        }
        else
        {
                printf("\nNo\n");
        }
}
```

Heap:

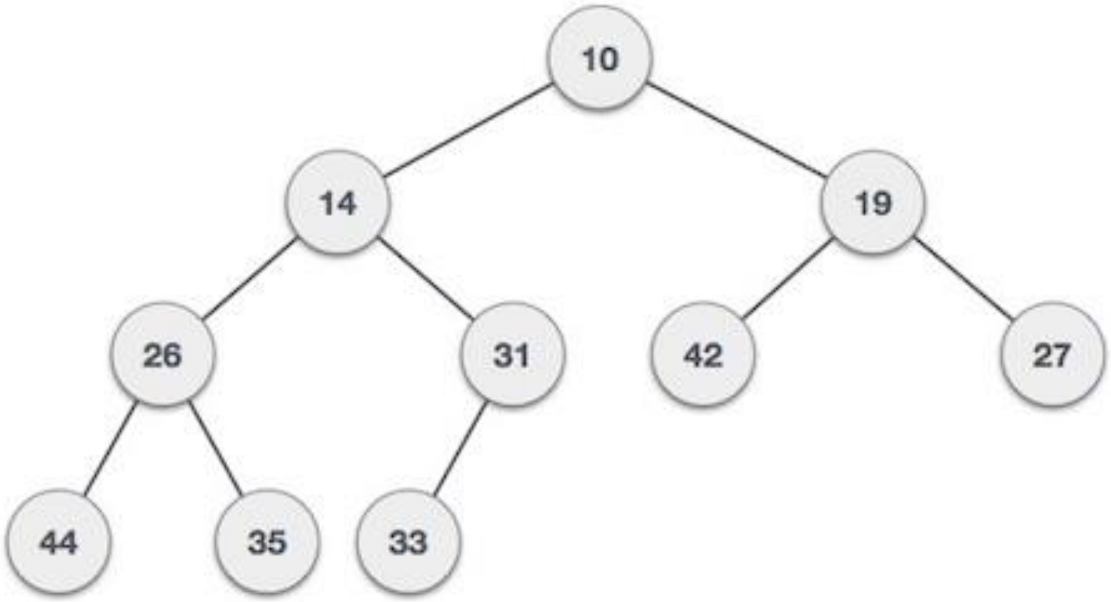
A special case of complete binary where root-node element is compared with its child and arranged accordingly

- → Root >= left – right
  [max heap]

→ Root <= left – right
[min heap]

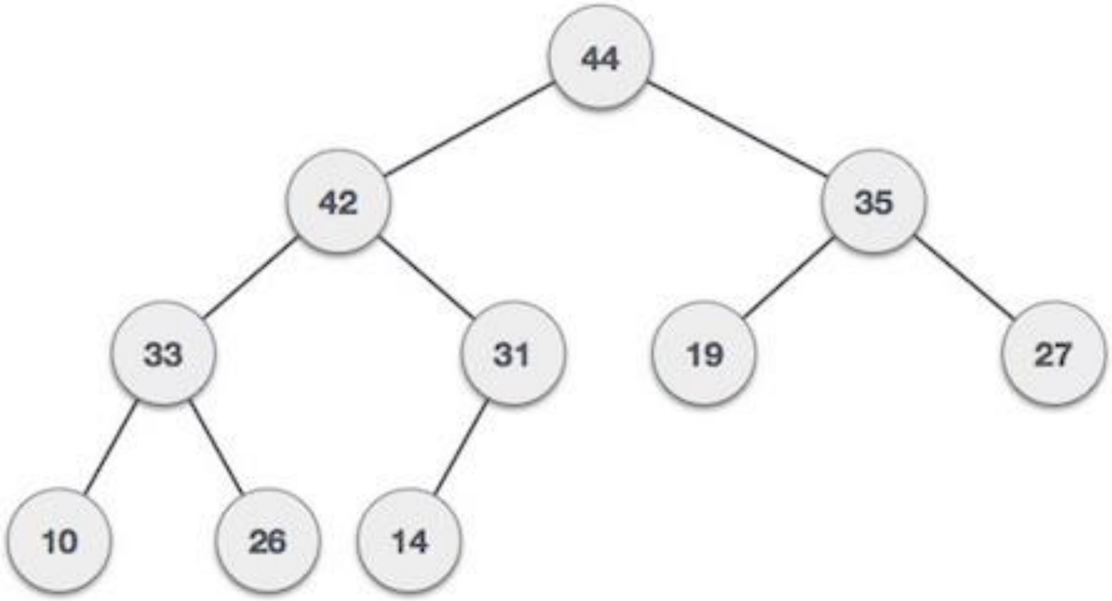
Min Heap:

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



Max Heap:

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



Max Heap Construction:

Step 1 – Create a new node at the end of heap.

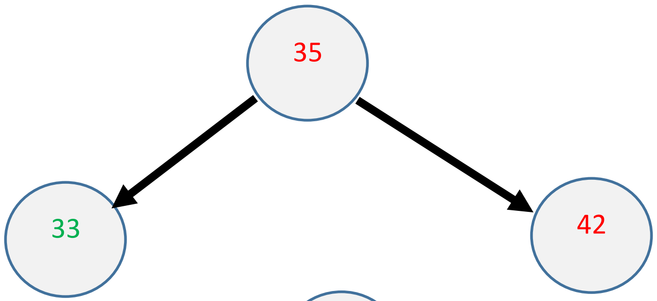
Step 2 – Compare the value of this child node with its parent.

Step 3 – If value of parent is less than child, then swap them.

Step 4 – Repeat step 2 & 3 until Heap property holds.

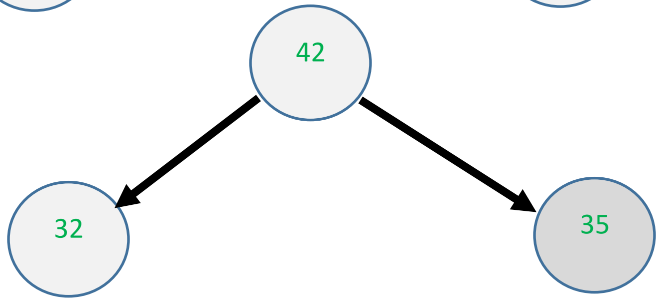
Input → 35 33 42 10 14 19 27 44

Input → 35 33 42 10 14 19 27 44

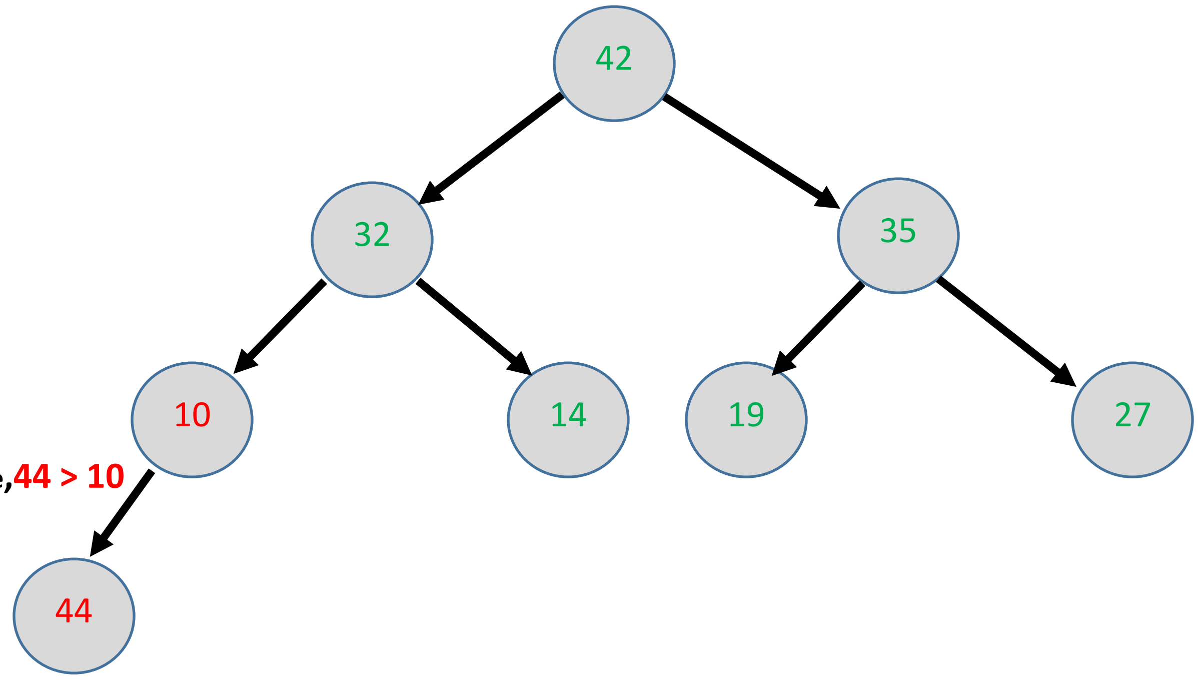


Here, “**42 > 35**”

Now we swap 42&35.

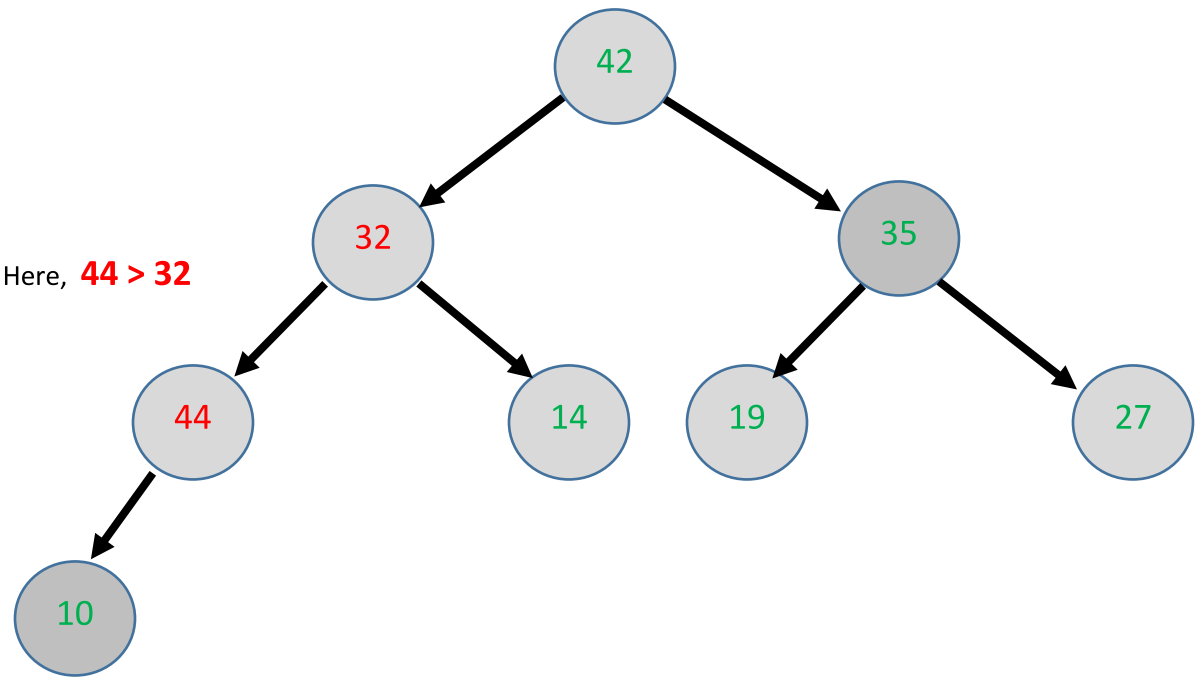


Input → 35 33 42 10 14 19 27 44

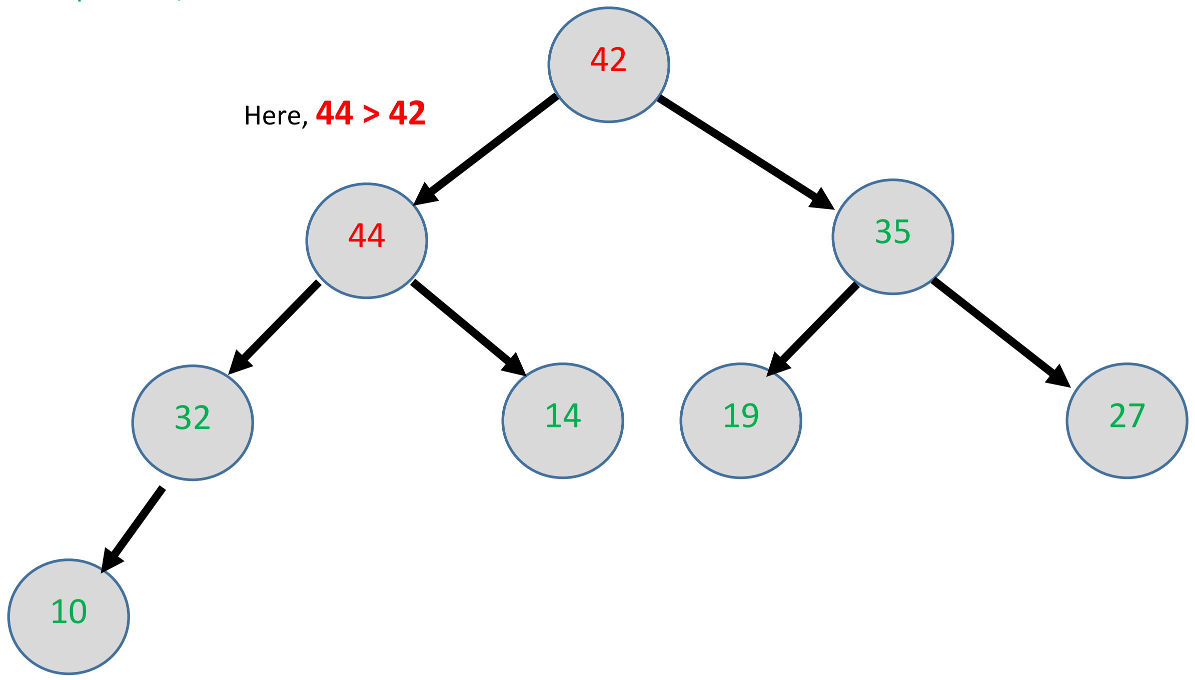


here,44 > 10

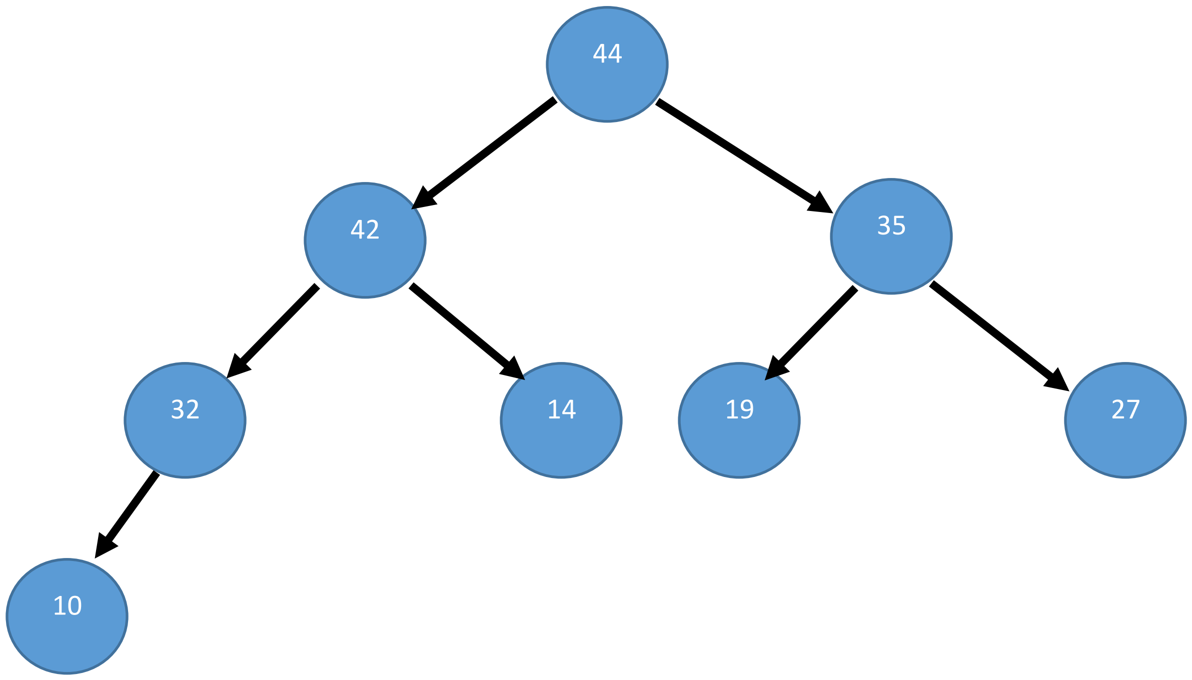
now we swap 44&10,



Now we swap 44&32,



Now we swap 44&42,



Time Complexity:

Calculate T(n) : no of operations

1. int x = 0; 2 operations (declaration & assign)

   int sum = 0; 2 operations (declaration & assign)

   T(n) = 2+2

   = 4

2. int i, sum, n; 3 operations (3 declaration)

   sum = 0, n=5; 2 operations (assign)

   for(i = 0; i < n; i++) n operation [looping n times]

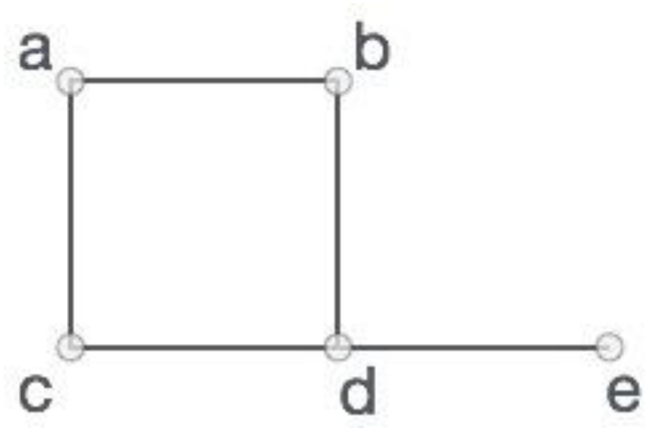
   sum + =i; 2 operations [inside each loop]

   T(n) = 3+2+(n*2)

   = 5+2n

Graph:

- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links.
- Interconnected objects are called vertices (vertex)
- Links are connect the vertices are called edges.

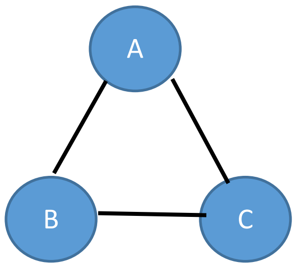


In the above graph,

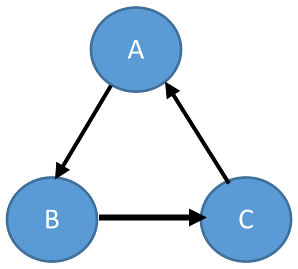
V = {a, b, c, d, e}

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

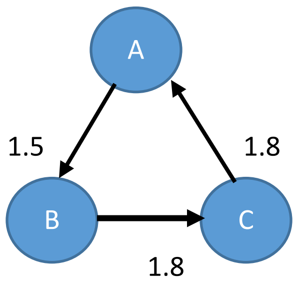
Different Types of Graph:



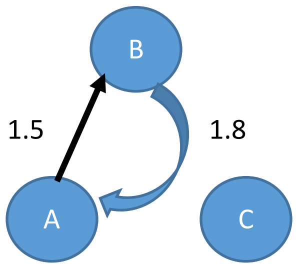
Unweighted connected graph



Unweighted directed cycle connected



Directed connected weighted



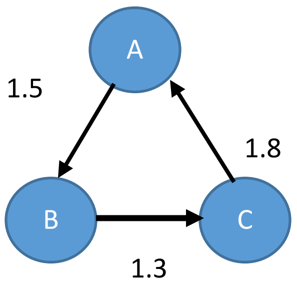
Directed weighted disconnected

Presenting a Graph in Memory:

- → Adjacency matrix
- → Adjacency list

Adjacency matrix:

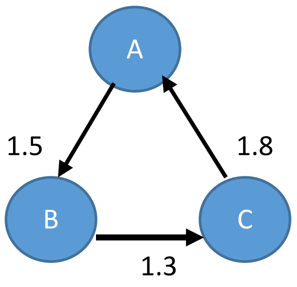
- → Based on array

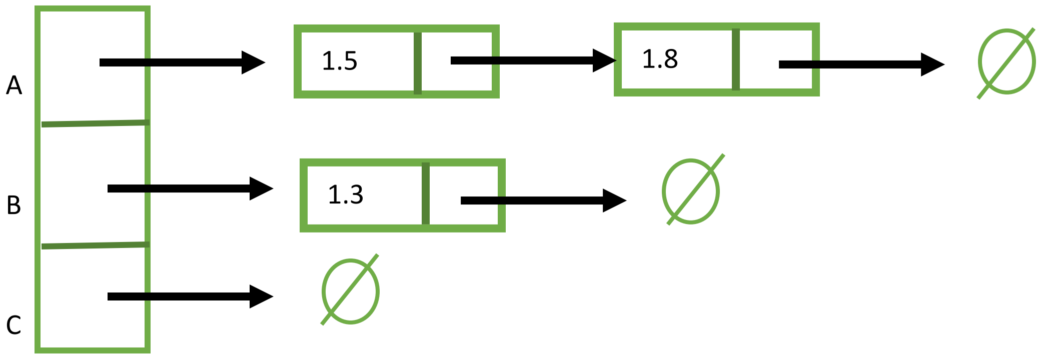


	A	B	C
A		1.5	1.8
B			1.3
C			

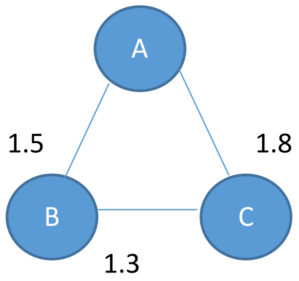
Adjacency list:

- → Using link list



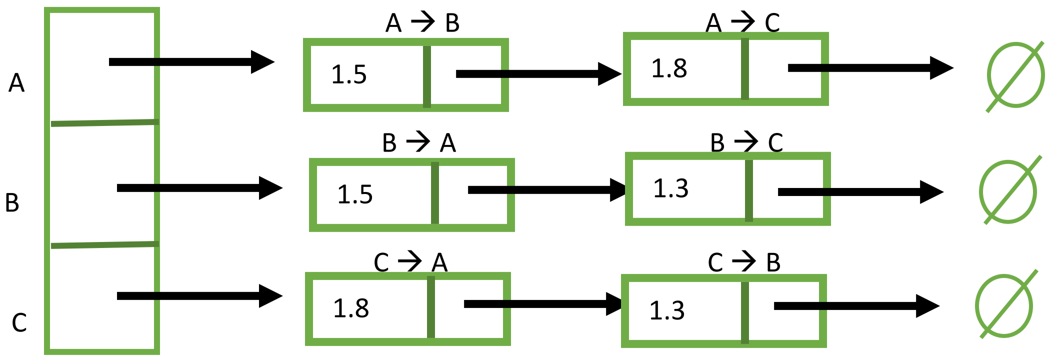


#Adjacency List for undirected graph:



	A	B	C
A		1.5	1.8
B	1.5		1.3
C	1.8	1.3	

*Adjacency list for undirected graph is always symmetric.

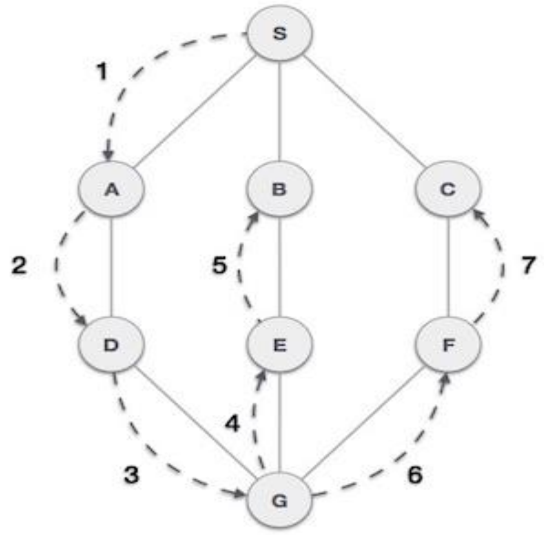


Graph Travarsal:

DFS(depth first search):

In DFS we start from a origin node then we go deeper and deeper until we find an end node. After an end node found we find another path to another end.

Uses a stack to remember to get the next vertex:



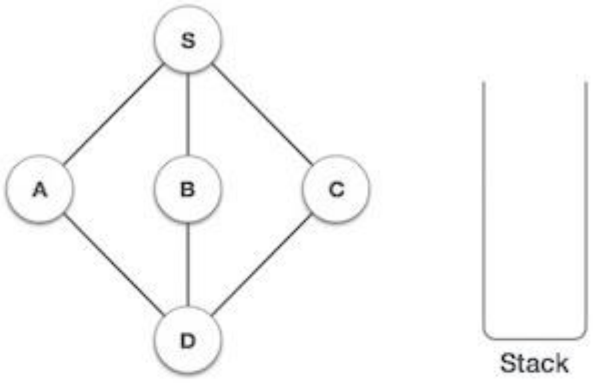
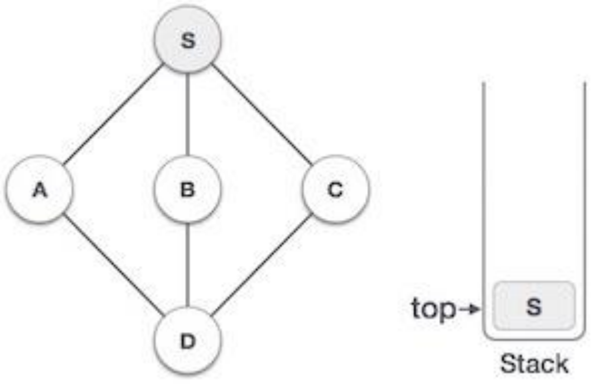
Here, we start from S. then visit A, then D, then G. now we got two way one is to E and other is to F. we visit E, then B. now there are no further unvisited vertex. So we backtrack to G and visit F, then C. now there are no unvisited vertex.

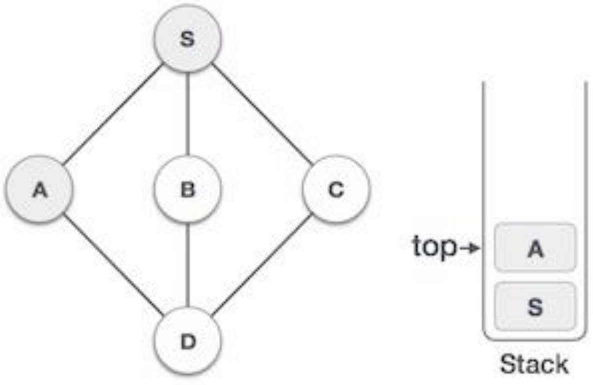
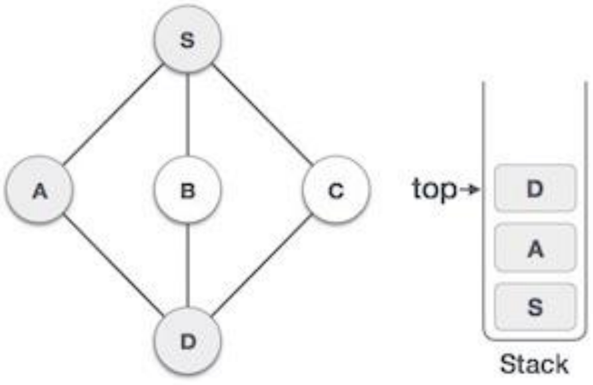
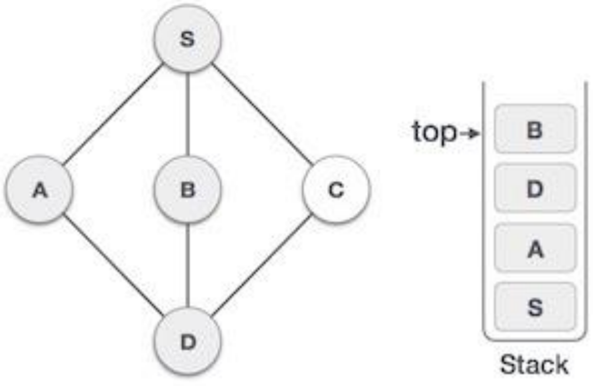
RULES OF TRAVERSAL IN DFS:

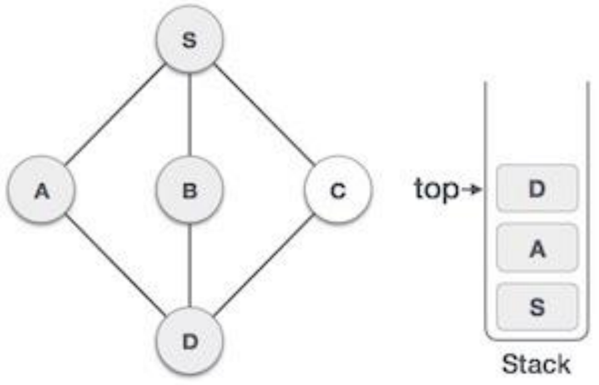
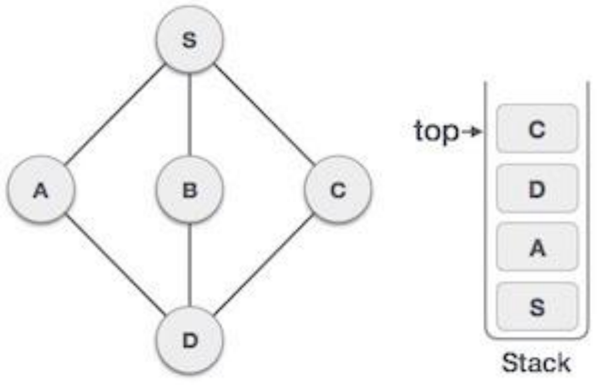
Rule 1 – Start from a vertex then Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1.		Initialize the stack.
2.		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.

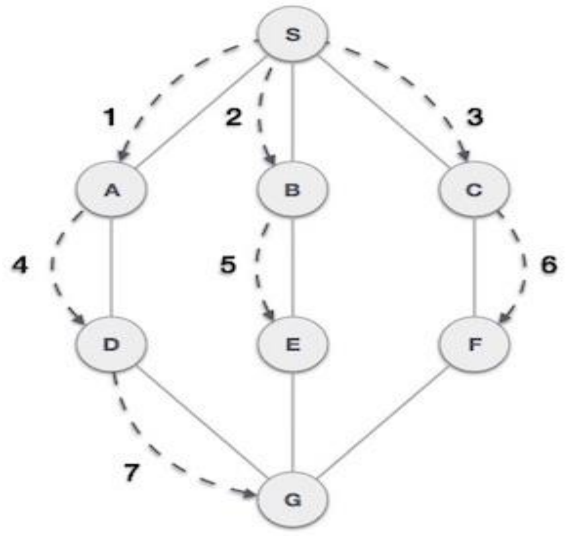
3.		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4.		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6.		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.
7.		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

BFS(BREATH FIRST SEARCH):

In BFS firstly we visit each nodes adjacent nodes before visiting their grandchild or further away nodes



Here we start from S then visit A then backtrack to S then visit B, then again backtrack to S and visit C.

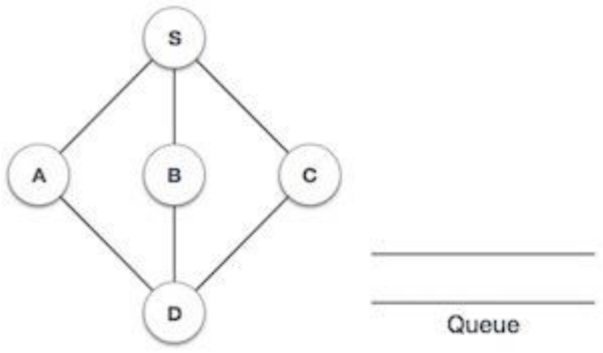
Then we backtrack to A and visit D, after that we backtrack to B and visit E, then we backtrack to C and visit F. after visiting them we again backtrack to D and visit G.

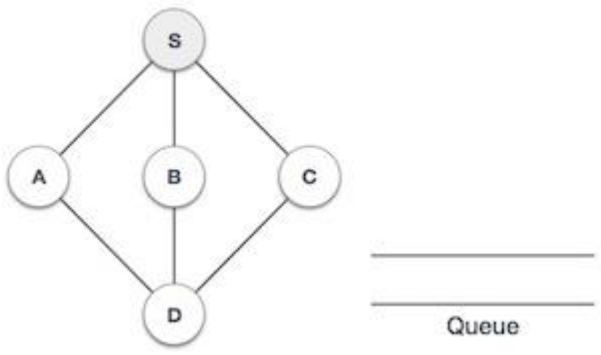
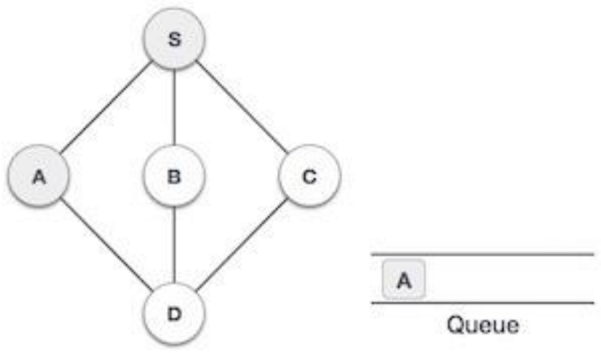
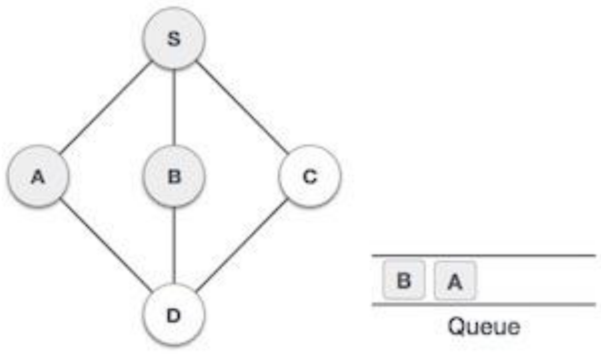
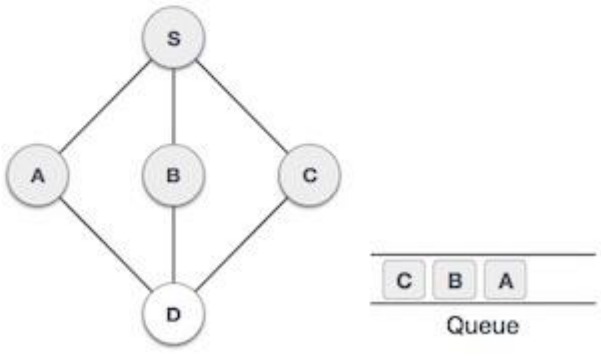
RULES:

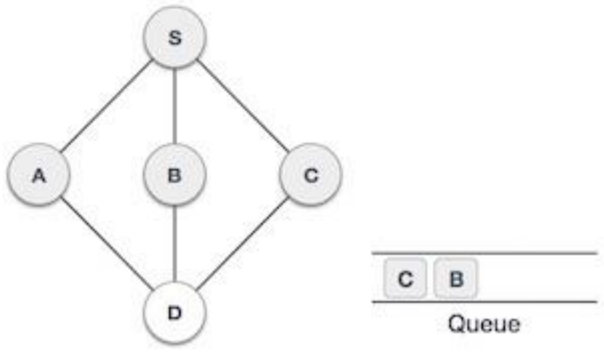
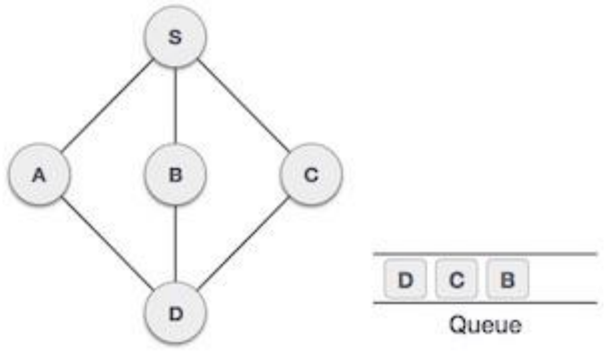
Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1.		Initialize the queue.

2.		We start from visiting **S** (starting node), and mark it as visited.
3.		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4.		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5.		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

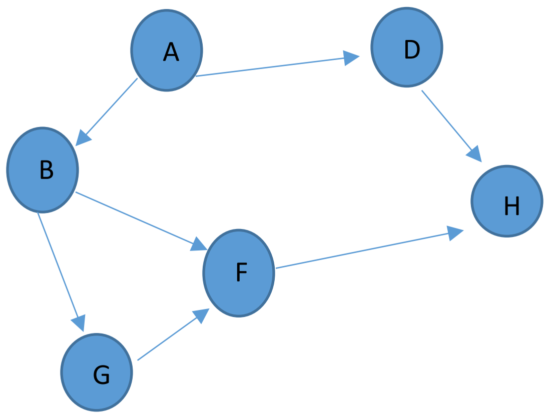
6.		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7.		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, there are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

TOPOLOGICAL ORDERING:

This type of order is based on indegree zero. Here we start with a vertex having indegree 0.

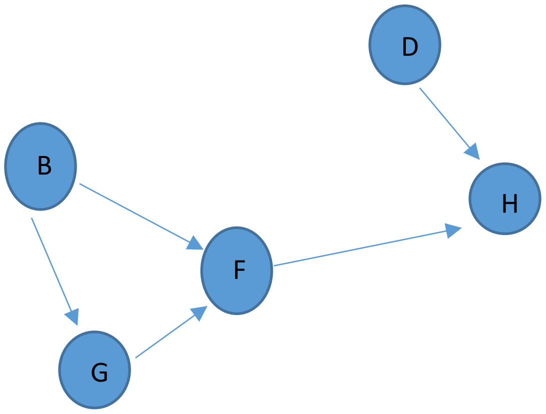
Step 1:



A" has indegree zero. So, we take A and remove it from the graph.

So, {A,

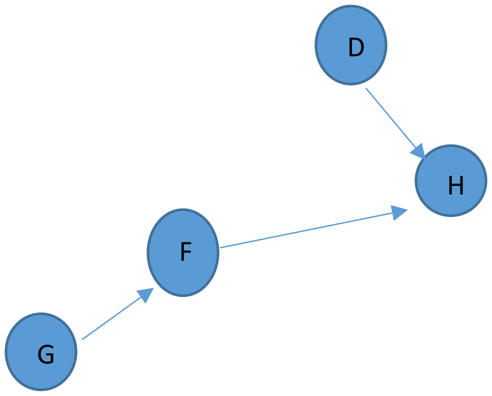
STEP 2:



Here both B&D have indegree zero. So we will take B as B has higher outdegree(2). Then we will remove B from graph.

So, {A,B..

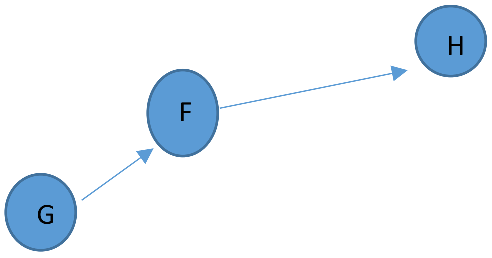
STEP 3:



Here both D&G have indegree zero. we will take D. Then we will remove D from graph.

So, {A,B,D..

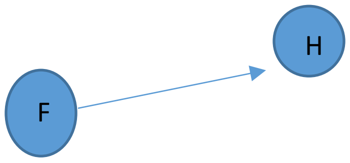
STEP 4:



we will take G. Then we will remove G from graph.

So, { A,B,D,G..

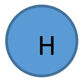
STEP 5:



we will take F. Then we will remove F from graph.

So, { A,B,D,G,F..

STEP 6:



we will take H. Then we will remove H from graph.

So, { A,B,D,G,F,H}

HERE THE ORDER BECOMES { A, B, D, G, F, H }