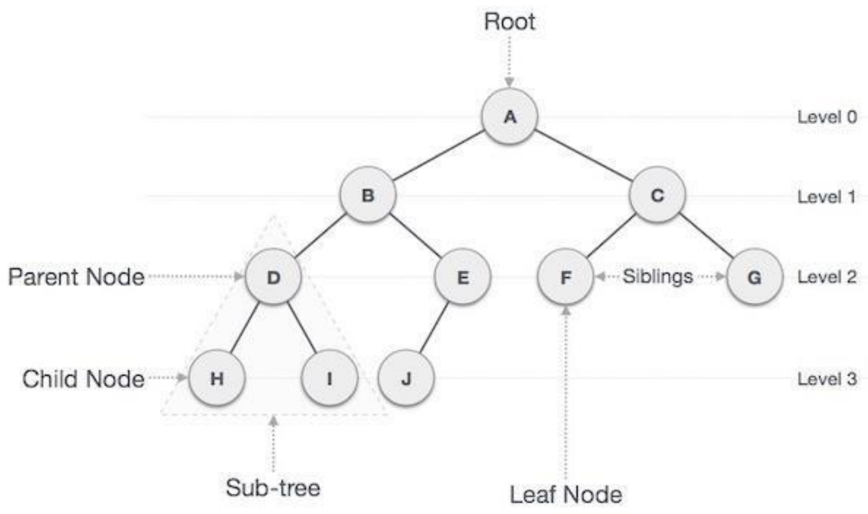
Module III: Trees: Concept of Trees, Tree terminologies**, Binary tree:** Complete and Extended Binary tree, Expression trees, Representation of Binary Tree, Traversing Binary Trees – Preorder, Inorder, Postorder. **Binary Search Tree (BST**): Search, Insertion and Deletion operations, creating a Binary Search Tree. **Graph:** Concept of Graph, Graph terminologies, Graph Traversal – **BFS, DFS**. (20hr)

TREE

- A tree is a data structure consisting of nodes organized as a hierarchy
- Tree represents nodes connected by edges.

Binary Tree

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



-

Following are important terms with respect to tree.

- Path − Path refers to sequence of nodes along the edges of a tree.
- Root − Node at the top of the tree is called root. There is only one root per tree and one path from root node to any node.
- Parent − Any node except root node has one edge upward to a node called parent.

- Child − Node below a given node connected by its edge downward is called its child node.
- Leaf − Node which does not have any child node is called leaf node.
- Subtree − Subtree represents descendents of a node.
- VisiTing − Visiting refers to checking 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 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.

Tree Traversal.

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

In-order Traversal

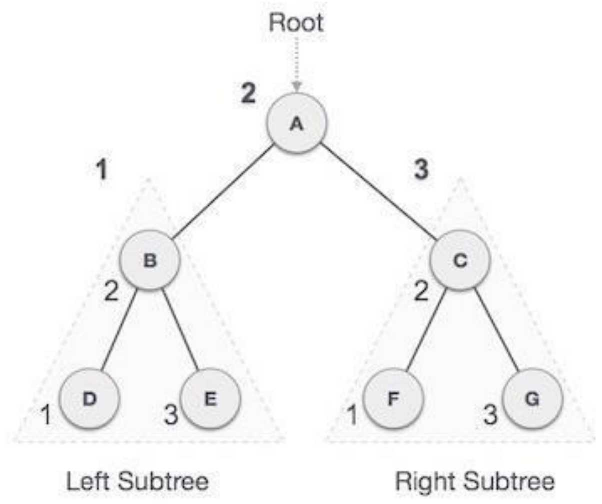
Pre-order Traversal

Post-order Traversal

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

1.Inorder Traversal

- In this traversal method, the left left-subtree is visited first, then root and then the right sub-tree. We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed inorder, the output will produce sorted key values in ascending order.



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

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −

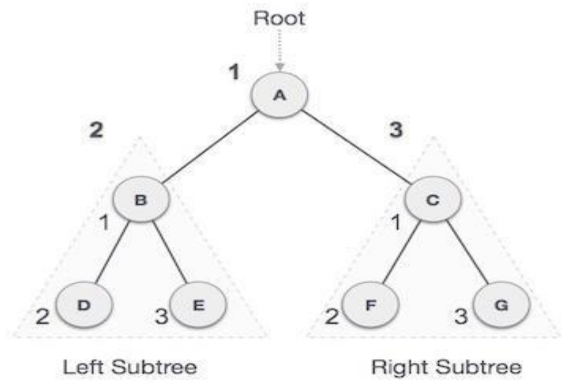
Step 1 − Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

2.Preorder Traversal

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



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

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed – Step

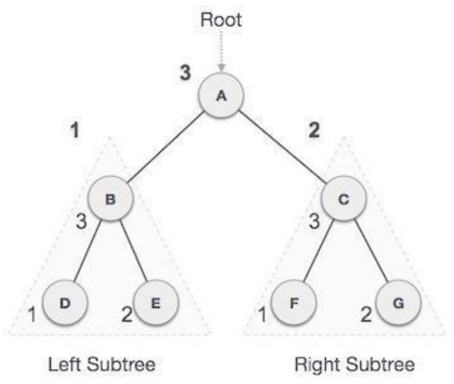
1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

3.Postorder Traversal

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



D->E->B->F->G->C->A

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree
Step 3- Visit root node.

Binary Search Tree

- Binary Search tree exhibits a special behavior. A node's left child must have value less than its parent's value and node's right child must have value greater than it's parent value.
- Node
- A tree node should look like the below structure. It has data part and references to its left and right child nodes.
- A binary search tree (BST) is a tree in which all nodes follows the below mentioned properties −
    1. The left sub-tree of a node has key less than or equal to its parent node's key.
    2. The right sub-tree of a node has key greater than or equal to its parent node's key.
- Thus, a binary search tree (BST) divides all its sub-trees into two segments; left sub-tree and right sub-tree and can be defined as −
- left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

Node

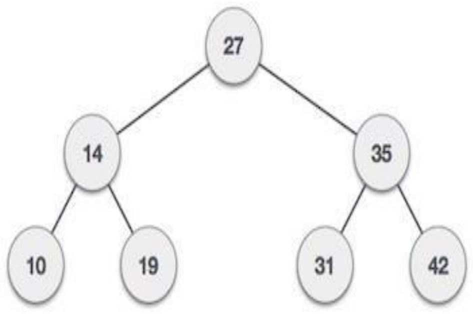
Define a node having some data, references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

Representation OF BST

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has key and associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

An example of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and higher valued keys on the right sub-tree.

BST Basic Operations

The basic operations that can be performed on binary search tree data structure, are following −

1. Insert − insert an element in a tree / create a tree.
2. Search − search an element in a tree.
3. Preorder Traversal − traverse a tree in a preorder manner.
4. Inorder Traversal − traverse a tree in an inorder manner.
5. Postorder Traversal − traverse a tree in a postorder manner.

Algorithm

```
struct node* search(int data){
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data){

      if(current != NULL) {
         printf("%d ",current->data);

         //go to left tree
         if(current->data > data){
            current = current->leftChild;
         }  //else go to right tree
         else {
            current = current->rightChild;
         }

         //not found
         if(current == NULL){
            return NULL;
         }
      }
   }

   return current;
}
```

_Example of creating a binary search tree

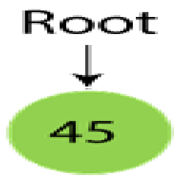
Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

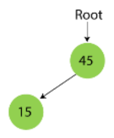
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45



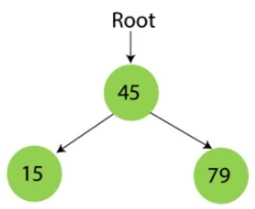
Step 2 - Insert 15.

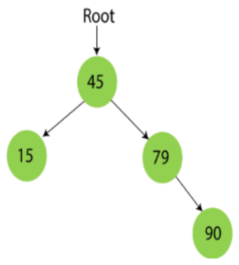
As 15 is smaller than 45, so insert it as the root node of the left subtree.



_Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



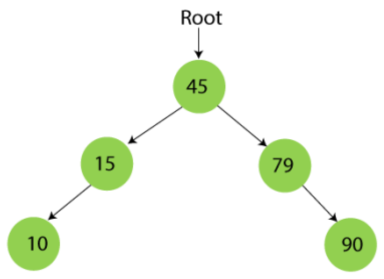


Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.

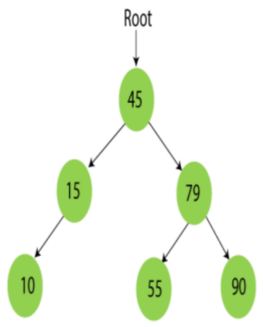
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



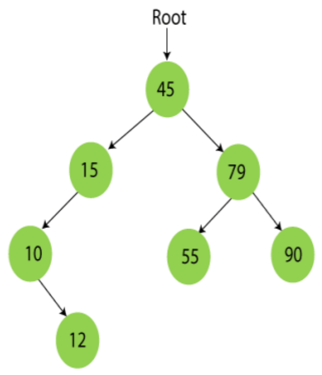
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



Step 7 - Insert 12.

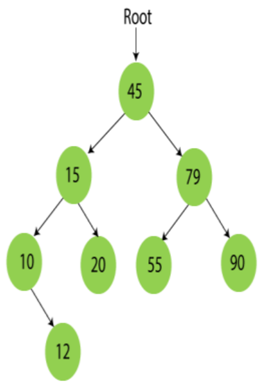
12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree



of 10.

Step 8 - Insert 20.

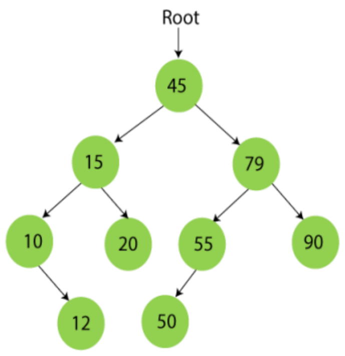
20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree



.

Step 9: insert 50

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

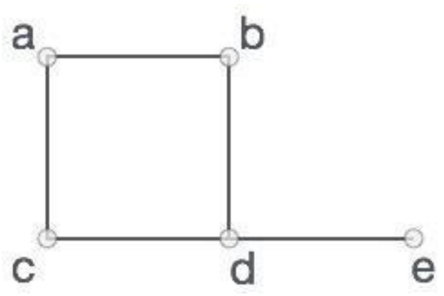
Algorithm

```
1. Create a new BST node and assign values to it.
2. insert(node, key)
   i) If root == NULL,
       return the new node to the calling function.
   ii) if root=>data < key
       call the insert function with root=>right and assign the return value in root=>right.
      root->right = insert(root=>right,key)
   iii) if root=>data > key
       call the insert function with root->left and assign the return value in root=>left.
       root=>left = insert(root=>left,key)
3. Finally, return the original root pointer to the calling function.
```

Graph Data Structure

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}

Graph Data Structure

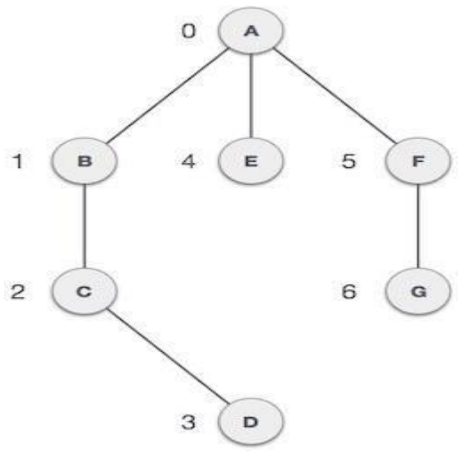
Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.

- **Edge** − Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.

- **Path** − Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph which are following.

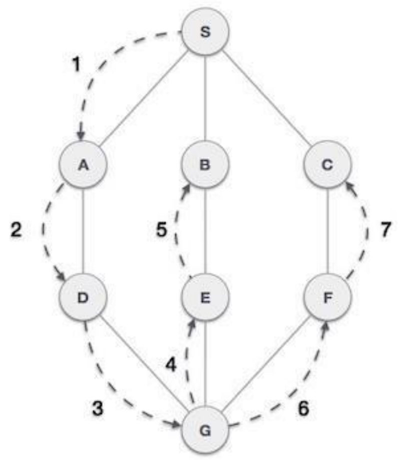
- **Add Vertex** − add a vertex to a graph.
- **Add Edge** − add an edge between two vertices of a graph.
- **Display Vertex** − display a vertex of a graph.

To know more about Graph, please read Graph Theory Tutorial. We shall learn traversing a graph in coming chapters

Graph Traversal (DFS and BFS)

Depth First Search algorithm(DFS).

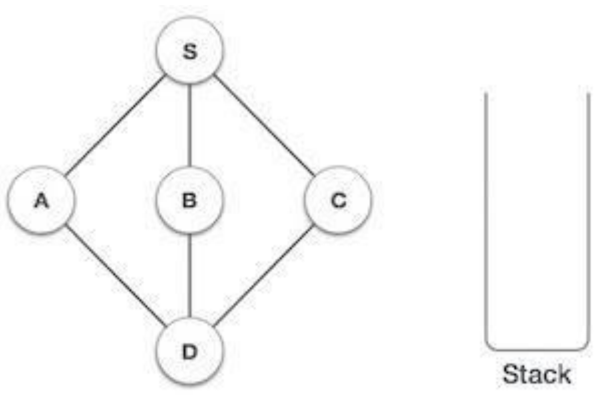
Depth First Search algorithm(DFS) traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

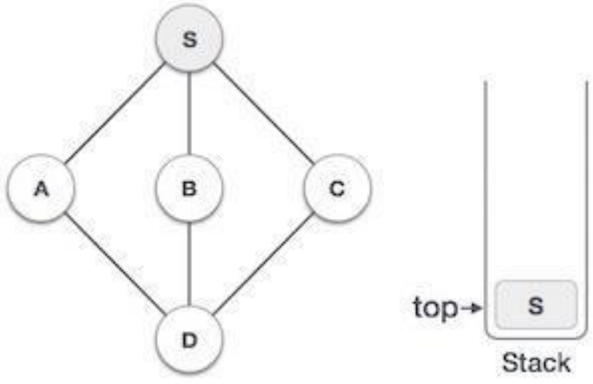
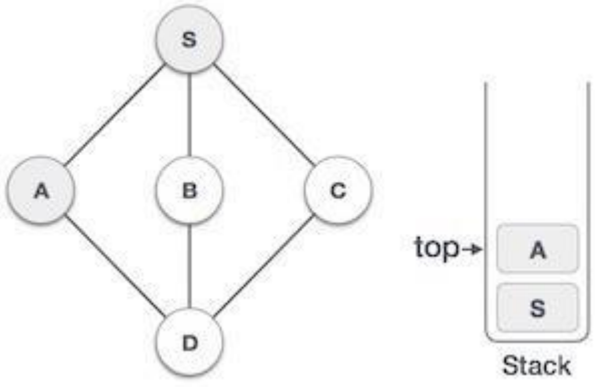
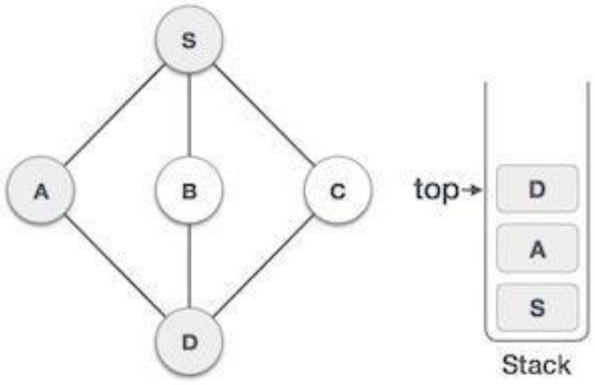


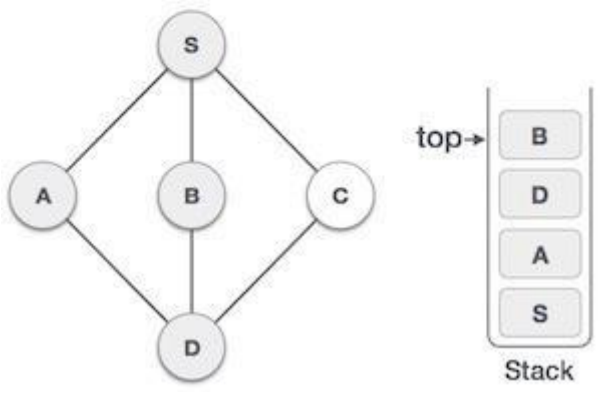
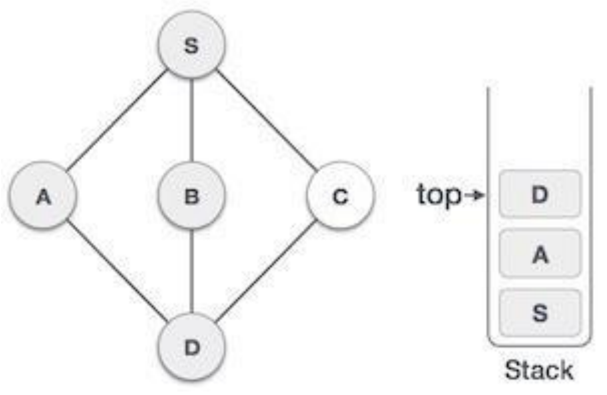
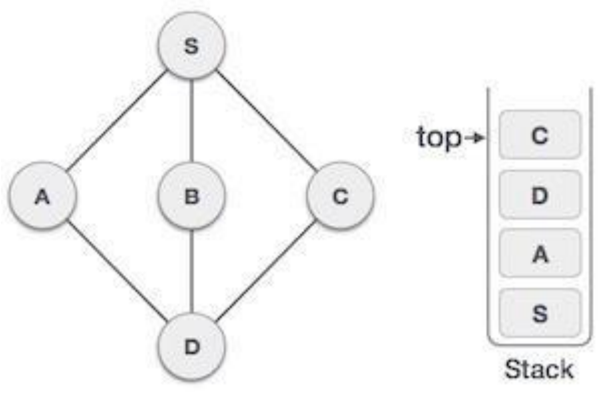
As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G.

It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from 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 stack is empty.

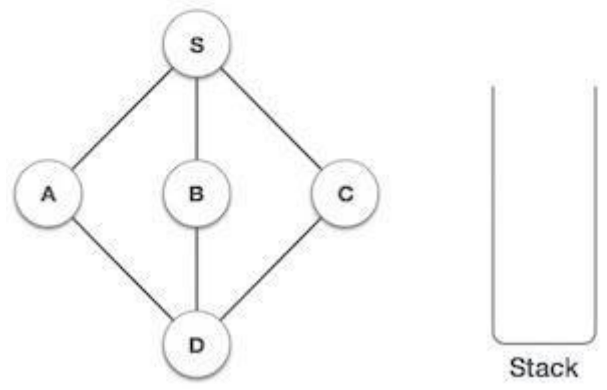
Step	Traversal	Description
1.		Initialize the stack

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 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 visited and put onto the stack. Here we have **B** and **C** nodes which are adjacent to **D** and both are unvisited. But we shall again choose in alphabetical order.

5.		We choose **B**, mark it visited and put onto stack. Here **B** does not have any unvisited adjacent node. So we pop **B** from the stack.
6.		We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of stack.
7.		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it 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 which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

We shall not see the implementation of Depth First Traversal (or Depth First Search) in C programming language. For our reference purpose we shall follow our example and take this as our graph model −



Implementation in C

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
 #define MAX 5
 struct Vertex {
char label;    bool visited;
};
 //stack variables
 int stack[MAX];  int top = -1;
//graph variables
 //array of vertices struct Vertex* lstVertices[MAX];
 //adjacency matrix int adjMatrix[MAX][MAX];
 //vertex count int vertexCount = 0;
 //stack functions
 void push(int item) {     stack[++top] = item;
}   int pop() {
return stack[top--];
}   int peek() {
return stack[top];
 }  bool
 isStackEmpty() {
 return top == -1;
 }

 //graph functions

 //add vertex to the vertex list void addVertex(char label) {    struct
 Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
 vertex->label = label;      vertex->visited = false;
 lstVertices[vertexCount++] = vertex;
 }

 //add edge to edge array void
 addEdge(int start,int end) {
 adjMatrix[start][end] = 1;
 adjMatrix[end][start] = 1;
 }

 //display the vertex void displayVertex(int
 vertexIndex) {    printf("%c
 ",lstVertices[vertexIndex]->label); }
```

```
//get the adjacent unvisited vertex int getAdjUnvisitedVertex(int vertexIndex)
{    int i;     for(i = 0; i<vertexCount; i++) {
if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false) {
return i;
      }
   }
 return -1;
 }  void
 depthFirstSearch() {
 int i;

    //mark first node as visited    lstVertices[0]-
 >visited = true;

    //display the vertex
 displayVertex(0);

    //push vertex index in
 stack    push(0);
 while(!isStackEmpty()) {
       //get the unvisited vertex of vertex which is at top of the stack
 int unvisitedVertex = getAdjUnvisitedVertex(peek());

       //no adjacent vertex found
 if(unvisitedVertex == -1) {          pop();
 }else {          lstVertices[unvisitedVertex]-
 >visited = true;
displayVertex(unvisitedVertex);
push(unvisitedVertex);
      }
   }


    //stack is empty, search is complete, reset the visited flag
 for(i = 0;i < vertexCount;i++) {       lstVertices[i]->visited
 = false;
    }
 }
   int main() {    int i, j;     for(i = 0;
 i<MAX; i++) // set adjacency {       for(j
 = 0; j<MAX; j++) // matrix to 0
 adjMatrix[i][j] = 0;
    }
 addVertex('S');   // 0
 addVertex('A');   // 1
 addVertex('B');   // 2
 addVertex('C');   // 3
 addVertex('D');   // 4
      addEdge(0, 1);    // S
 - A    addEdge(0, 2);    //
 S - B    addEdge(0, 3);
 // S - C    addEdge(1, 4);
 // A - D
```

```
   addEdge(2, 4);    // B - D
addEdge(3, 4);    // C - D

   printf("Depth First Search: ");

depthFirstSearch();
    return
0;
}
```

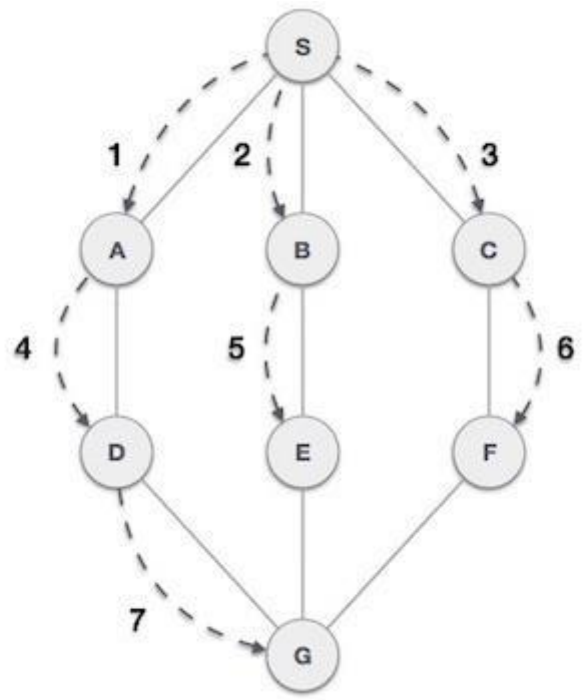
If we compile and run the above program then it would produce following result –

Output

```
Depth First Search: S A D B C
```

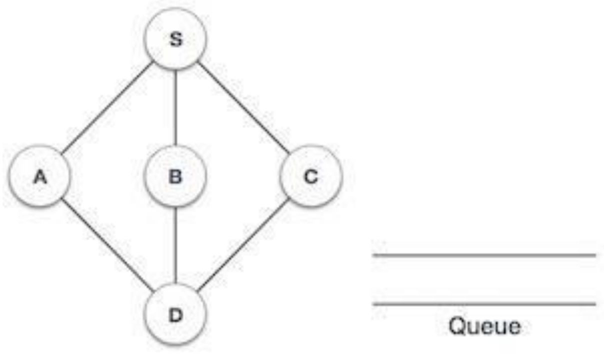
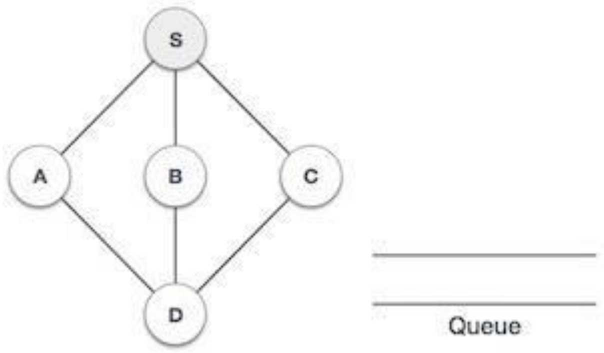
Breadth First Search algorithm(BFS)

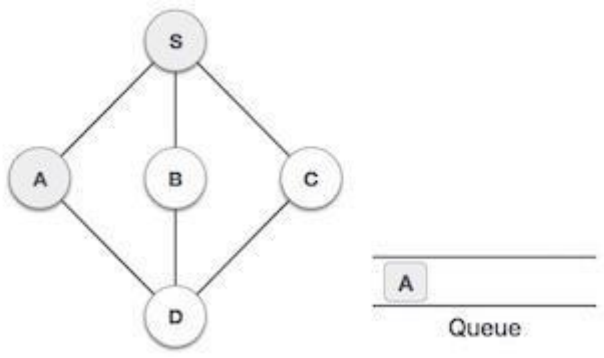
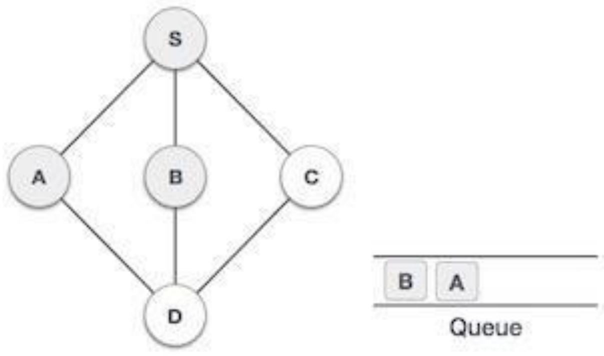
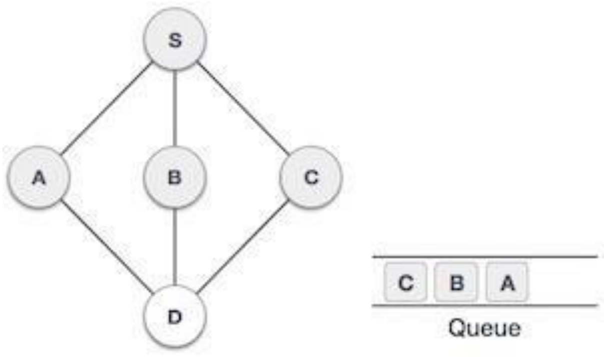
Breadth First Search algorithm(BFS) traverses a graph in a breadthwards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

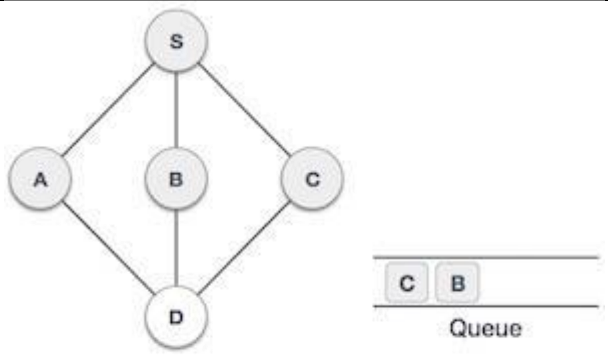
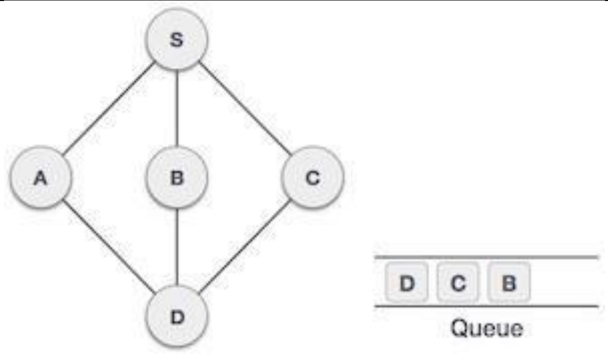


As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex found, remove the first vertex from queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until queue is empty.

Step	Traversal	Description
1.		Initialize the queue.
2.		We start from visiting **S** (starting node), and mark it visited.

3		We then see unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A** mark it visited and enqueue it.
4		Next unvisited adjacent node from **S** is **B**. We mark it visited and enqueue it.
5		Next unvisited adjacent node from **S** is **C**. We mark it 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 visited and enqueue it.

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