

UNIT IV

NON-LINEAR DATA STRUCTURES

Syllabus:

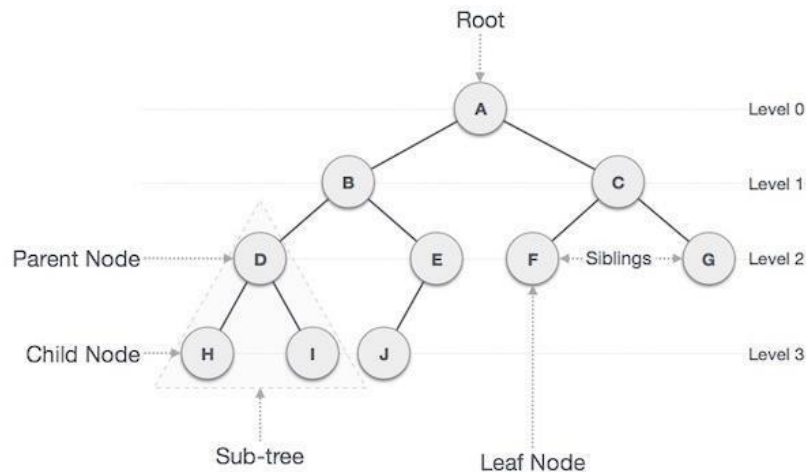
Trees – Binary Trees – Binary tree representation and traversals –Binary Search Trees – Applications of trees. Set representations – Union-Find operations. Graph and its representations – Graph Traversals.

TREES STRUCTURE

Definition:

Tree is a non-linear data structure. It organized the data in hierarchical manner. A tree is a finite set of one or more nodes such that there is a specially designated node called the root node and root node can have zero or more sub trees $T_1, T_2, T_3, \dots, T_n$. Each of whose roots are connected by a directed edge from root R.

Tree is collection of nodes in which the first node is called root and root has many number of sub tree $T_1, T_2, T_3, \dots, T_n$.



Terms:

1. Root

A node which does not have a parent is called as root node.

2. Node

Each data element in the tree is called as node.

3. Leaf node

A path from node n_1 to n_k is defined has sequence of nodes $n_1, n_2, n_3, \dots, n_k$. Such that n_i is a parent of n_{i+1} . Example: $A \rightarrow B \rightarrow E \rightarrow J$

6. Length for a path

Number of edges in the path.

Example: Consider path from A to J is 3

7. Degree

Number of sub trees of the node is called degree.

8. Level

Root is at level 1 then i 's children are at level $2+i$

Example: level

9. Depth

For any node n, the depth n is length of unique path from root to n.

10. Height

For any node n, the height of node n is the length of longest path from n to left.

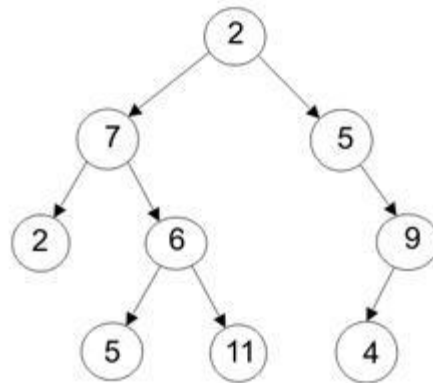
11. Forest

Collection of tree node is known as forest.

BINARY TREE ADT

Definition: -

Binary Tree is a special type of tree in which no node can have most two children. Typically, child nodes of a binary tree on the left is called left child and node on right is called right child. Maximum number of nodes



at level i of a binary tree is 2^{i-1} .

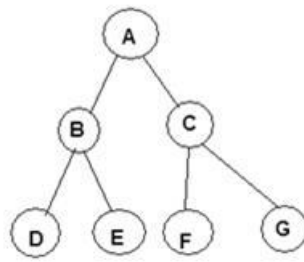
A simple binary tree of size 9 and height 3, with a root node whose value is 2. The above tree is neither a sorted nor a balanced binary tree

Representation of tree.

1. Sequential representation or array representation.
2. Linked representation.

1. Sequential representation or array representation.

The elements are represented using arrays. For any element in position i, the left child is in position 2i, the right child is in position (2i + 1), and the parent is in position (i/2).



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

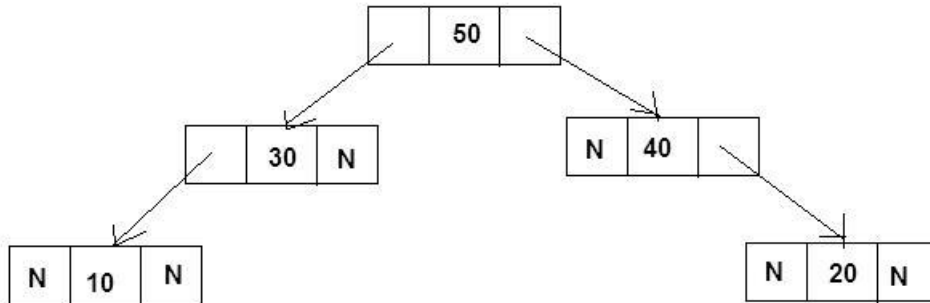
0 1 2 3 4 5 6

Array representation of Binary Tree

2. Linked representation

The elements are represented using pointers. Each node in linked representation has three fields, namely,

- * Pointer to the left subtree
- * Data field
- * Pointer to the right subtree



Linked representation of Binary Tree

Routine of creating tree using linked list.

```

struct tree
{
int data;
struct tree *leftchild;
struct tree *rightchild;
};
  
```

Recursive Traversals of Tree

Traversing means visiting each node at once. Tree traversal is a method for visiting all the nodes in the tree exactly once.

There are three types of tree traversal techniques, namely

1. Inorder Traversal or symmetric order
2. Preorder Traversal or depth-first order
3. Postorder Traversal

Inorder Traversal

The inorder traversal of a binary tree is performed as

- * Traverse the left subtree in inorder
- * Visit the root
- * Traverse the right subtree in inorder.

Recursive Routine for Inorder Traversal

```

void Inorder (Tree T)
{
if (T!=NULL)
{
  
```

```

Inorder (T->left);
printf(“%d”,T->data);
Inorder (T->right);
}
}

```

Preorder Traversal

The preorder traversal of a binary tree is performed as follows,

- * Visit the root
- * Traverse the left subtree in preorder
- * Traverse the right subtree in preorder.

Recursive Routine for Preorder Traversal

```

void Preorder (Tree T)
{
if (T != NULL)
{
printf(“%d”,T->data);
Preorder (T->left);
Preorder (T->right);
}
}

```

Postorder Traversal

The postorder traversal of a binary tree is performed by the following steps.

- * Traverse the left subtree in postorder.
- * Traverse the right subtree in postorder.
- * Visit the root.

Recursive Routine for Postorder Traversal

```

void Postorder (Tree T)
{
if (T != NULL)
{
Postorder (T->Left);
Postorder (T->Right);
printf(“%d”,T->data);
}
}

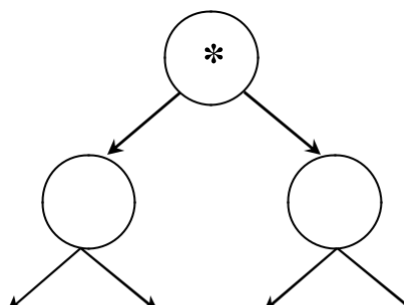
```

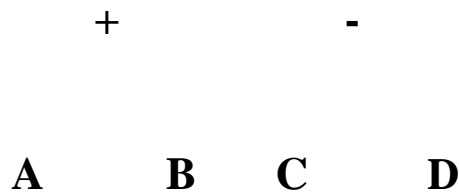
EXPRESSION TREE

An expression tree is tree in which left nodes have the operands and interior node have the operators. Like binary tree, expression tree can also be traversed by inorder, preorder and postorder traversal.

Example:

$(A+B)*(C-D)$





Constructing an Expression Tree

Let us consider postfix expression given as an input for constructing an expression tree by performing the following steps:

1. Read one symbol at a time from the postfix expression and then scan the expression from left to right manner.
2. Check whether the symbol is an operand or operator.
 - a. If the symbol is an operand, create a one - node tree and push a pointer on to the stack.
 - b. If the symbol is an operator pop two pointers from the stack namely T_1 and T_2 and form a new tree with root as the operator and T_2 as a left child and T_1 as a right child. A pointer to this new tree is then pushed onto the stack.
3. Repeated the above steps until reach the end of the expression.
4. The final pointer in the stack is complete expression tree.

Example:

$ABC^*+DE^*F+G^*+$

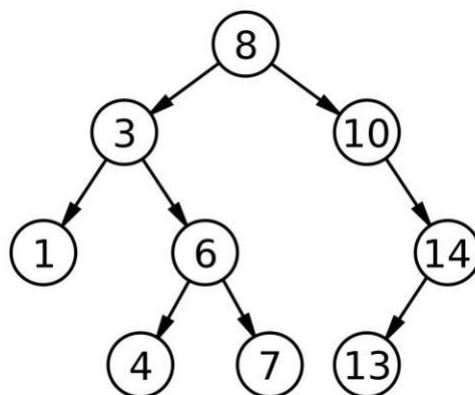
BINARY SEARCH TREE OR SEARCH TREE ADT

Definition: -

When we place constraints on how data elements can be stored in the tree, the items must be stored in such a way that the key values in left subtree of the root less than the key value of the root, and then the key values of all the node in the right subtree of the root are greater than the key values of the root. When this relationship holds in the entire node in the tree then the tree is called as a binary search tree.

The property that makes a binary tree into a binary search tree. That is every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X , and the values of all the keys in its right subtree are larger than the key value in X .

Example:



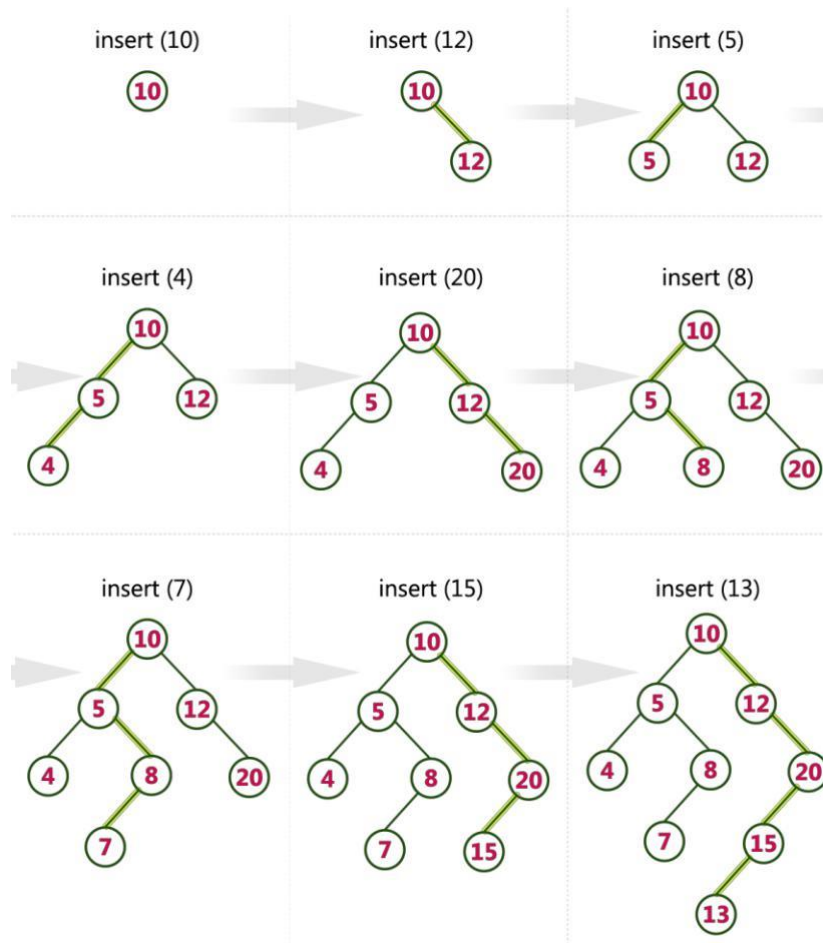
Operations of BST.

1. Insertion
2. Deletion
3. Find

4. Find min
5. Find max
6. Retrieve

Notes: when you're constructing the binary tree the given elements are read from first.

Example:



```
struct treenode
{
    int data;
    struct treenode *left;
    struct treenode *right;
};
```

Routine for perform find.

```
struct treenode * find(struct treenode *T,int x)
{
    if(T==NULL)
        return NULL;
```

```

else if(x<T->data)
    return find(T->left,x);
else if(x>T->data)
    return find(T->right,x);
else
    return T;
}

```

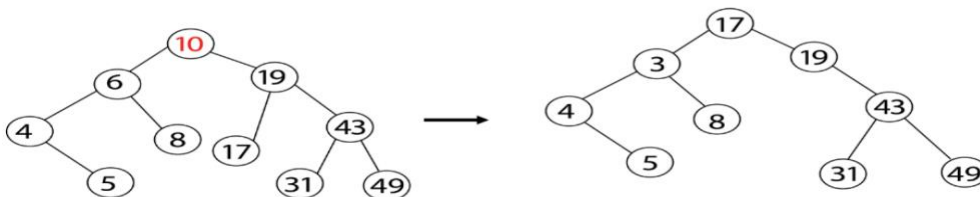
Routine for perform insertion.

```

struct treenode* insert(struct treenode *t,int x)
{
    if(t==NULL)
    {
        t=(struct treenode *)malloc(sizeof(struct treenode));
        t->data=x;
        t->left=t->right=NULL;
        return t;
    }
    else if(x<t->data)
        t->left=insert(t->left,x);
    else if(x>t->data)
        t->right=insert(t->right,x);
    return t;
}

```

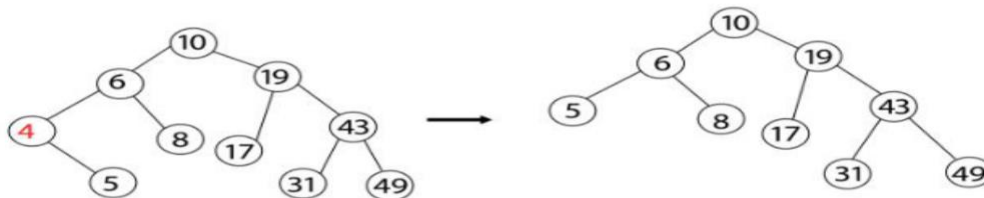
Deletion — I/P = 10
Node deleted to have 2 child.



Before Deletion

After Deletion

Deletion — I/P = 4
Node deleted to have 1 child.



Before Deletion

After Deletion

Routine for perform deletion

```

struct treenode * findmin(struct treenode *t)

```

```

{
  if(t==NULL)
    return NULL; /* There is no element in the tree */
  else if(t->left==NULL) /* Go to the left sub tree to find the min element */
    return t;
  else
    return findmin(t->left);
}

```

```

struct treenode* deletion(struct treenode *t,int x)

```

```

{
  struct treenode *temp;
  if(t==NULL)
    printf("Element not found\n");
  else if(x < t->data )
    t->left = deletion (t->left,x);
  else if(x > t->data )
    t->right = deletion (t->right,x);
  else if(T->right && T->left)
  {
    /* Here we will replace with minimum element in the right sub tree */
    temp = findmin(t->right);
    t->data = temp->data ;
    /* As we replaced it with some other node, we have to delete that node */
    t->right = deletion (t->right,t->data);
  }
  else
  {
    /* If there is only one or zero children then we can directly remove it from the tree and connect its
    parent to its child */
    temp = T;
    if(t->left==NULL)
      t = t->right;
    else if(T->right == NULL)
      t = t->left;
    free(temp); /* temp is longer required */
  }
  return T;
}

```

```

void inorder(struct treenode *t)

```

```

{
  if(t!=NULL)
  {
    inorder(t->left);
    printf("%d \t",t->data);
  }
}

```



```
        inorder(t->right);  
    }  
}
```

Applications of Tree

1. Manipulate hierarchical data.
2. Make information easy to search
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

DIFFERENCE BETWEEN BINARY AND BINARY SEARCH TREES:

BINARY TREE	BINARY SEARCH TREE
It is a tree with only two children	It is also a tree with only two children.
It has no restrictions regarding its children	In this the left child is lesser than the parent and the right child is greater than the parent

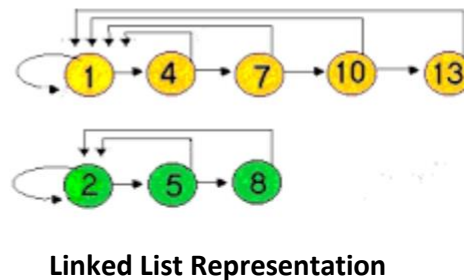
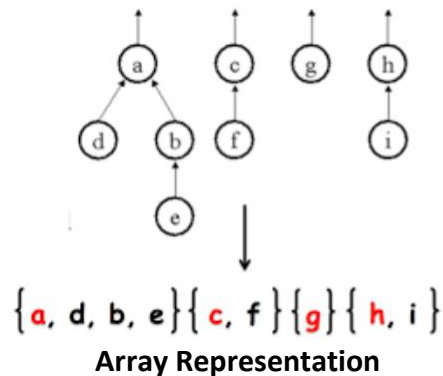
THE DISJOINT SET ADT

A disjoint set data structure keeps note of a set of non-overlapping subsets. It is a data structure that helps us solve the dynamic equivalence problem.

Various Representations Of Disjoint Set Adt

Array Representation

Linked List Representation



Array Representation

This representation assigns one position for each element. Each position stores the element and an index to the representative. To make the Find-Set operation fast we store the name of each equivalence class in the array. Thus the find takes constant time, $O(1)$. Assume element a belongs to set i and element b belongs to set j. When we perform Union(a,b) all j's have to be changed to i's. Each union operation unfortunately takes $\Theta(n)$ time. So for n-1 unions the time taken is $\Theta(n^2)$.

Tree Representation

A tree data structure can be used to represent a disjoint set ADT. Each set is represented by a tree. The elements in the tree have the same root and hence the root is used to name the set.

Operations

1. Find
2. Union

The trees do not have to be binary since we only need a parent pointer.

Make-set (DISJ_SET S)

```
int i;
for( i = N; i > 0; i-- )
    p[i] = 0;
```

Initially, after the Make-set operation, each set contains one element. The Make-set operation takes $O(1)$ time. set_type

Find-set(element_type x, DISJ_SET S)

```
if( p[x] <= 0 )
    return x;
else
```

```
    return( find( p[x], S ) );
```

The Find-Set operation takes a time proportional to the depth of the tree. This is inefficient for an unbalanced tree

void Union(DISJ_SET S, set_type root1, set_type root2)

```
    p[root2] = root1;
```

The union operation takes a constant time of $O(1)$

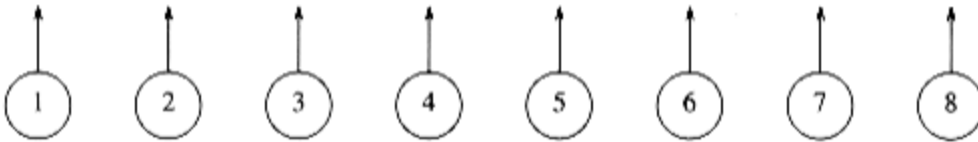


Figure 1.4 Tree representation of disjoint set ADT after Make-set operation

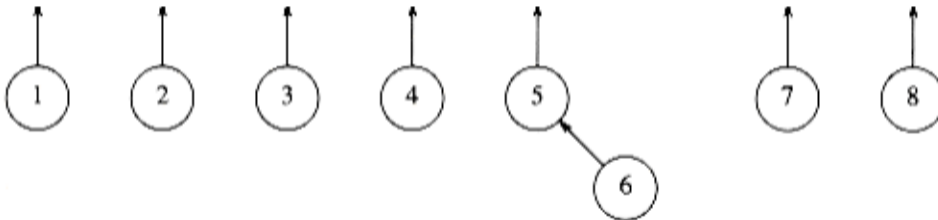


Figure 1.5 Tree representation of disjoint set ADT after union (5,6)

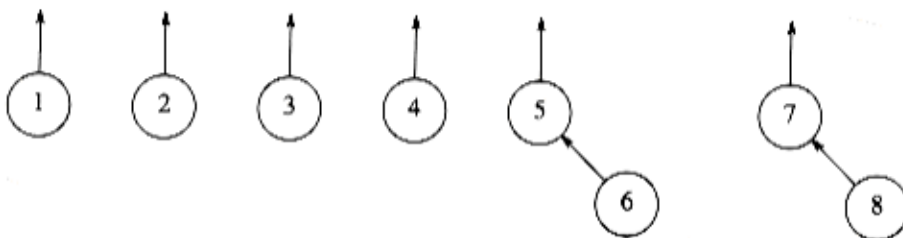
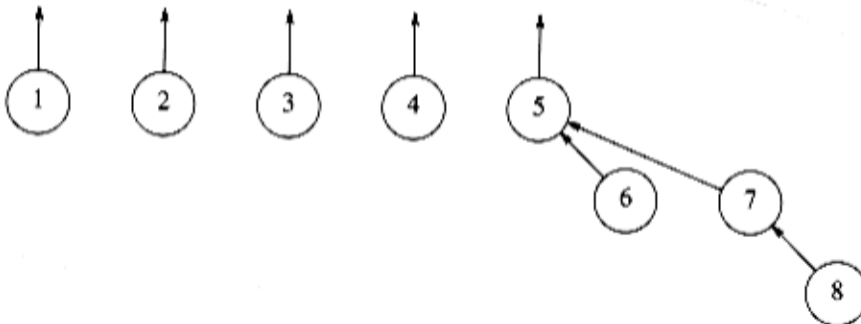


Figure 1.6 Tree representation of disjoint set ADT after union (7,8)



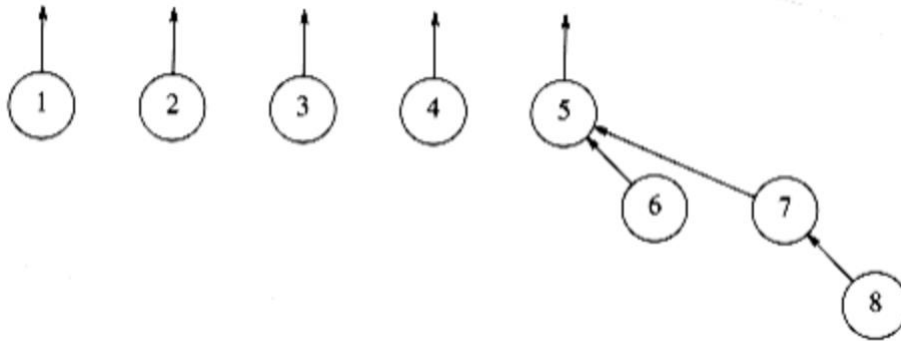


Figure 1.7 Tree representation of disjoint set ADT after union (5,7)

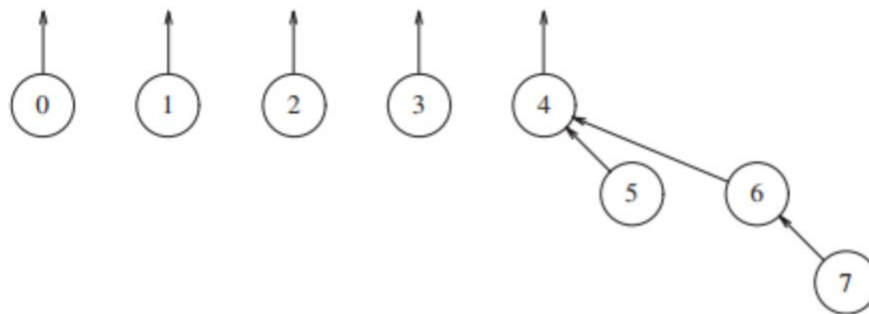


Figure 8.4 After union(4,6)

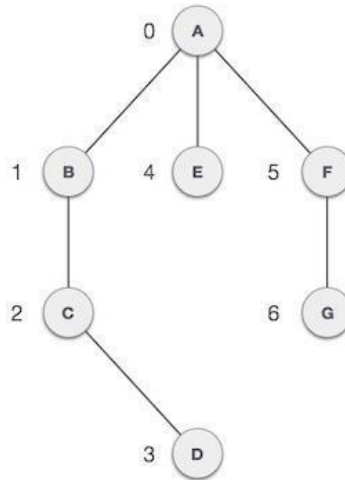
-1	-1	-1	-1	-1	4	4	6
0	1	2	3	4	5	6	7

Figure 8.5 Implicit representation of previous tree

Graphs

Introduction to graphs

A graph is a data structure that is used to represent a relational data e.g. a set of terminal in network or roadmap of all cities in a country. Such complex relationship can be represented using graph data structure. A graph is a structure made of two components, a set of vertex V and the set of edges E . Therefore, a graph is $G = (V, E)$ where G is graph. The graph may be directed or undirected. Vertices are referred to as nodes and the arc between the nodes are referred to as Edges.



Terms associated with graphs

1. Directed graph

A directed graph G is also called digraph which is the same as multigraph expect that each edges e in g is assigned a direction or in other words each edge in G is identified with an order pair (U,V) of node in G rather than an unordered pair

2. Undirected graph

An undirected graph g is a graph in which each edge e is not assigned a direction.

Undirected Graph

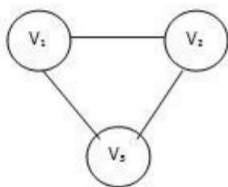


Figure 1: An Undirected Graph

Directed Graph

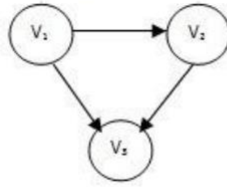
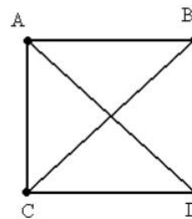
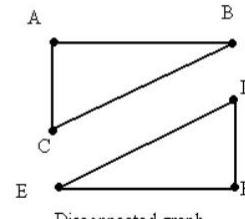


Figure 2: A Directed Graph



Connected graph



Disconnected graph

3. Connected graph

A graph is called connected if there is a path from any vertex to any other vertex.

Strongly Connected Graph If there is a path from every vertex to every other vertex in a directed graph then it is said to be strongly connected graph. Otherwise, it is said to be weakly connected graph.

4. Multiple edges

5. Distinct edges e and e' are called multiple edges if they connected the same and point.

Ex: $e=(U,V)$ then $e'=(U,V)$

6. Loop

An edges e is called loop if it has identical and points. $E=(U,U)$

7. Path

A path is a sequence of distinct vertices, each adjacent to the next. The length of such a path is number of edges on that path.

8. Cycle

A path from a node to itself is called cycle. Thus, a cycle is a path in which the initial and final vertices are same.

Acyclic Graph A directed graph which has no cycles is referred to as acyclic graph. It is abbreviated as DAG [Directed Acyclic Graph]

Note: a graph need not be a tree but a tree must be graph

9. Degree, incidence and adjacent

A vertex V is incident to an edges e if V is one of the two vertices in the order pair of vertices that constitute e .

The degree of a vertex is the number of edges incident to it.

The **indegree** of vertex V is the number of edges that have V as head and the **outdegree** of vertex V is number of edges that have v as the tail.

A vertex V is **adjacent** to vertex U if there is an edge from U to V . if V is adjacent to U , V is called a **successor** of U , and U a **predecessor** of V .

10. Weighted graph

A weighted graph is a graph in which edges are assigned weighted. Weights of an edge are called as cost.

11. complete graph

If there is an edges from each vertices to all other vertices in graph is called as completed graph.

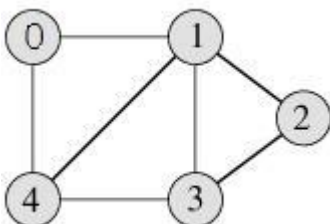
Graph and its representations

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.

2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale. See [this](#) for more applications of graph. Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix

2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, a

slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

The adjacency matrix for the above example graph is:

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

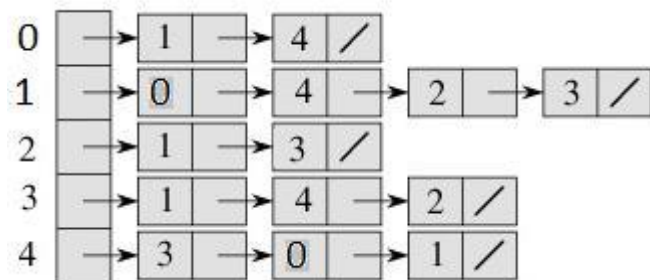
Adjacency Matrix Representation of the above graph

Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done $O(1)$.

Cons: Consumes more space $O(V^2)$. Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

Adjacency List:

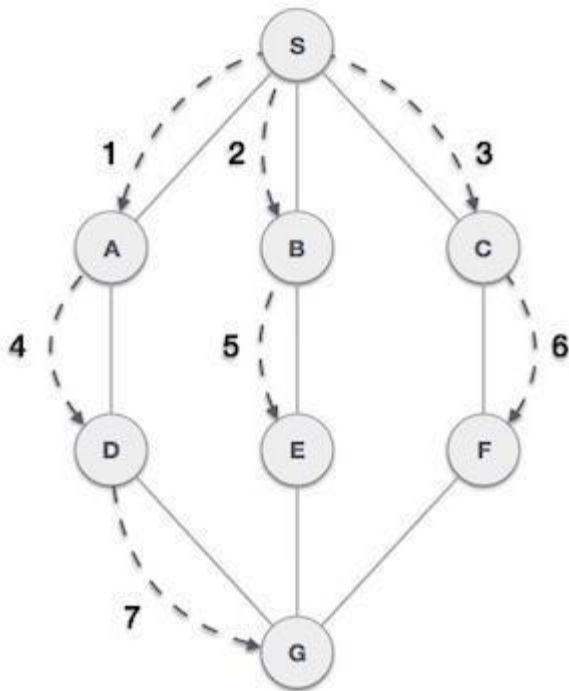
An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be $array[]$. An entry $array[i]$ represents the linked list of vertices adjacent to the i th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Adjacency List Representation of the above Graph

Data Structure - Breadth First Traversal

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

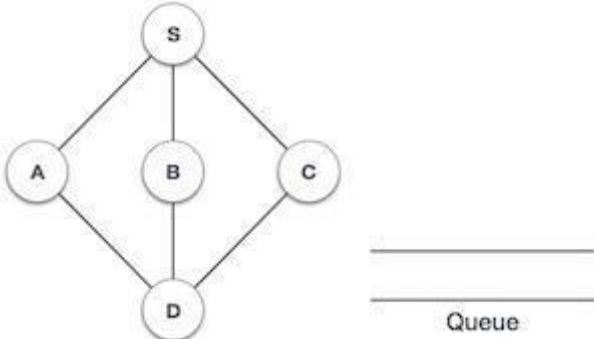
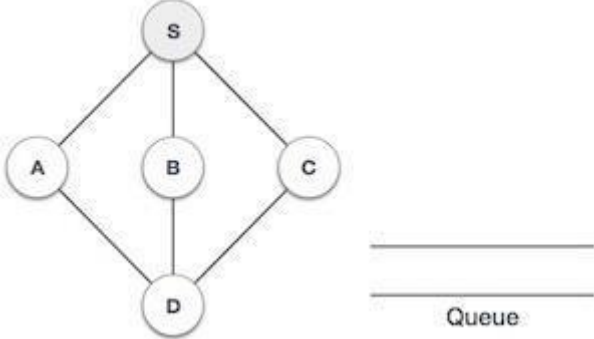
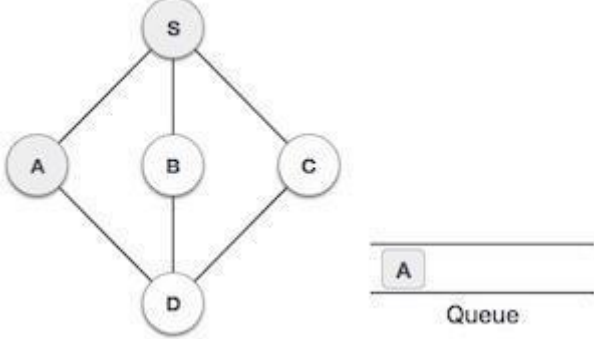
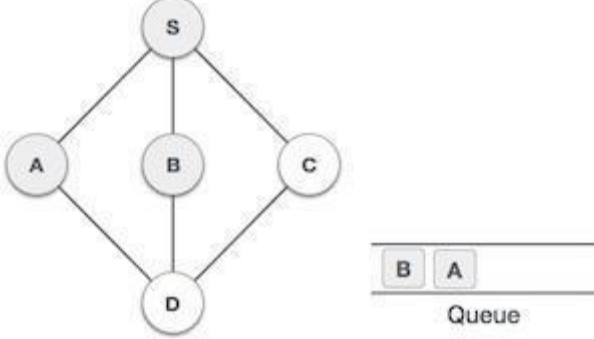
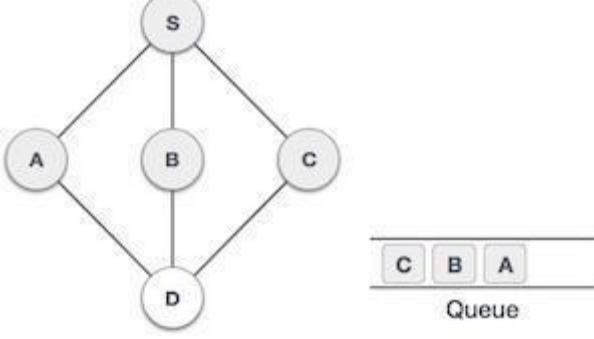


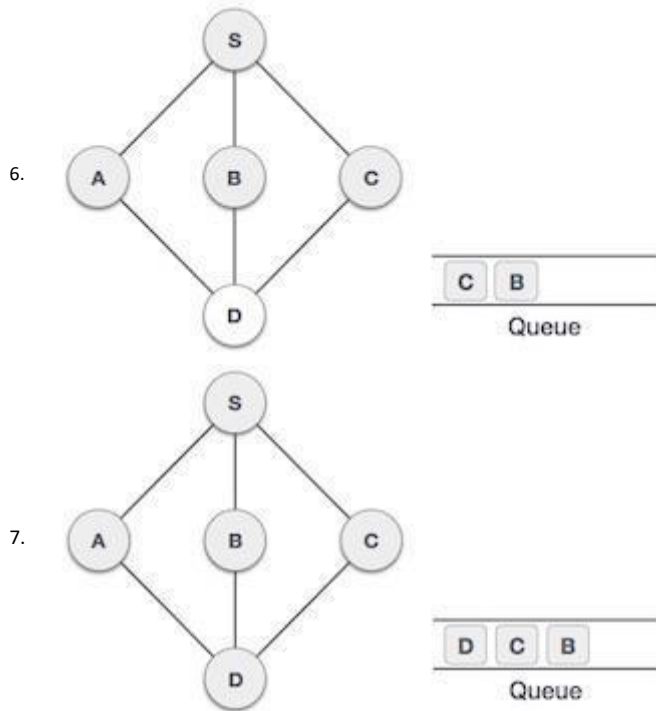
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

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.

```
void BFS(int v1)
{
front=rear=-1;
for(int i = 0; i < n; i++)
visited[i] = 0;
visited[v1] = 1;
Q[++rear]=v1;
while(front!=rear())
{
v1 = Q[++front];
printf("%d ", v1);
// Get all adjacent vertices of the dequeued vertex v1
// If a adjacent has not been visited, then mark it visited
// and enqueue it
for(int v2=0;v2<n;v2++)
{
if(g[v1][v2]==1&&visited[v2]==0)
{
visited[v2] = 1;
Q[++rear]=v2; } } } }
```


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.



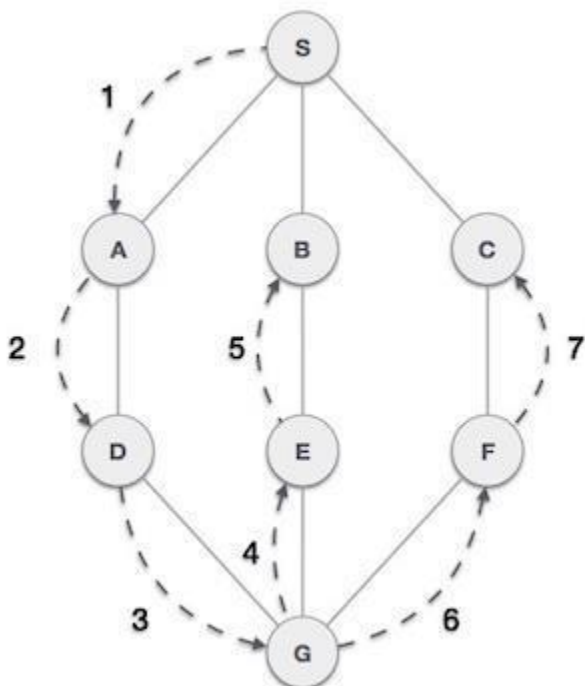
Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we 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.

Data Structure - Depth First Traversal

Depth First Search (DFS) algorithm 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 the 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 the following rules.

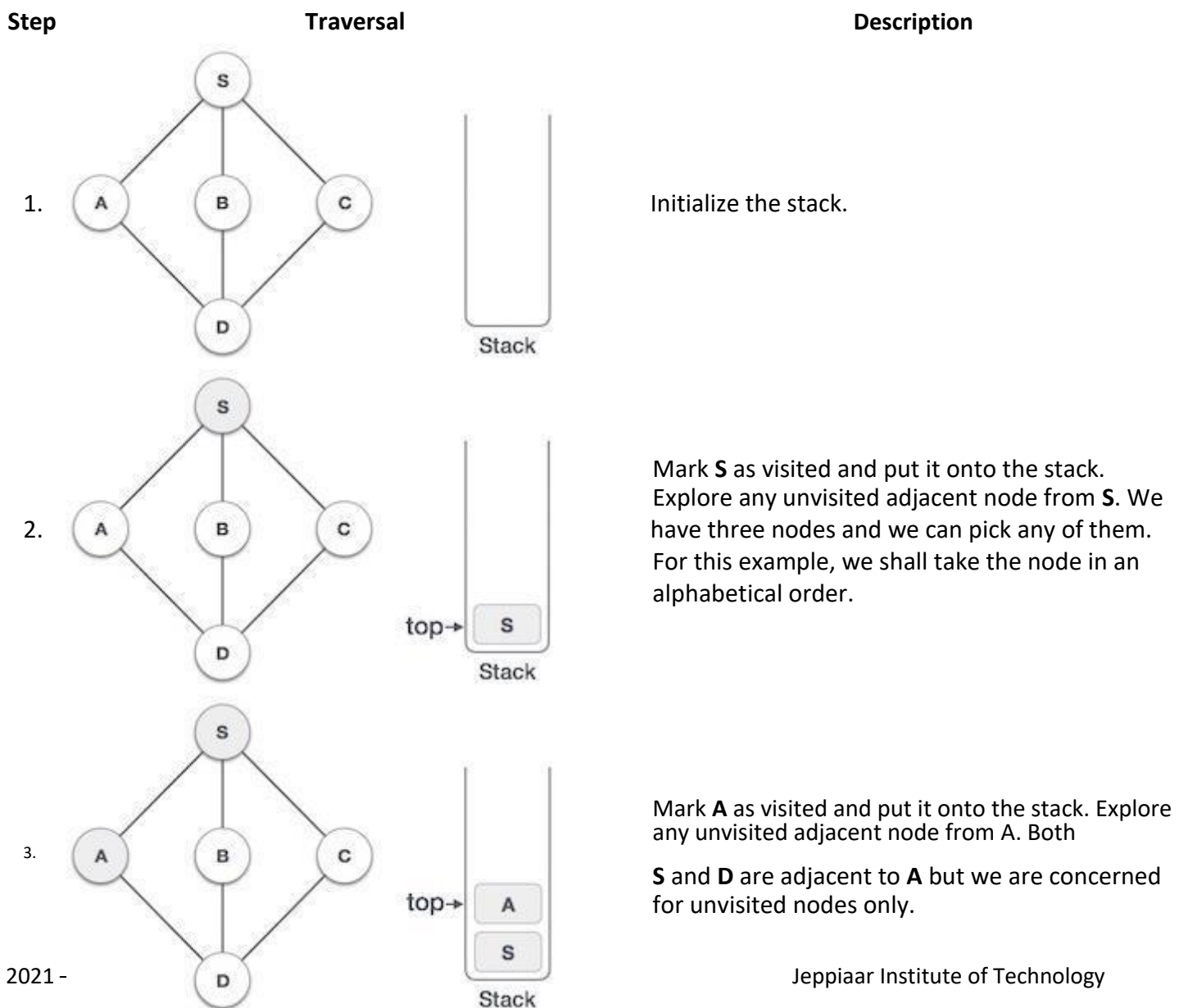
Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

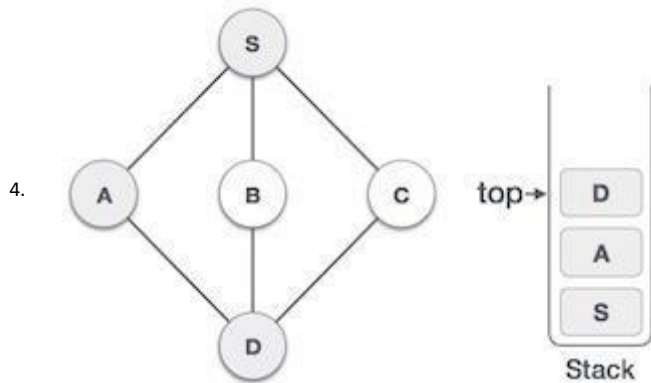
Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

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

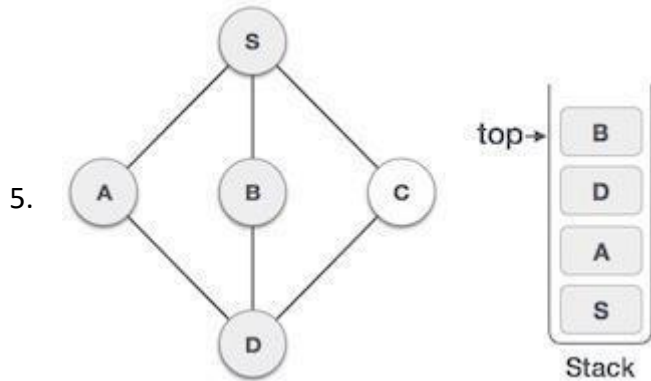
```
void dfs( Vertex v )
{
v.visited = true;
for each Vertex w adjacent to v if( !w.visited )
    dfs( w );
}
```

Figure Template for depth-first search (pseudocode)

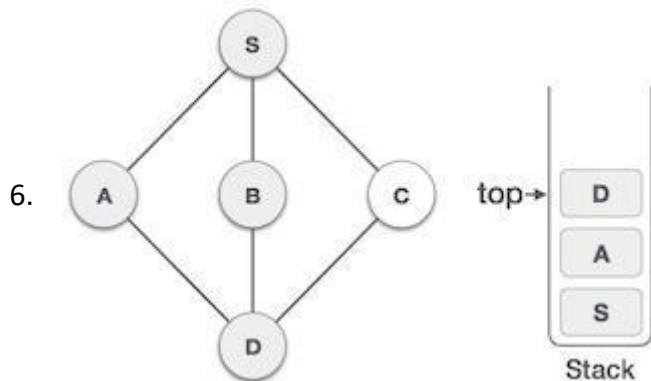




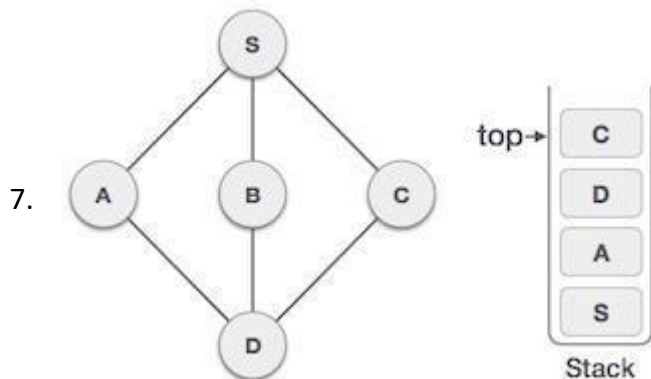
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.



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.



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.



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.