***Unit.5-Trees And Graphs.***

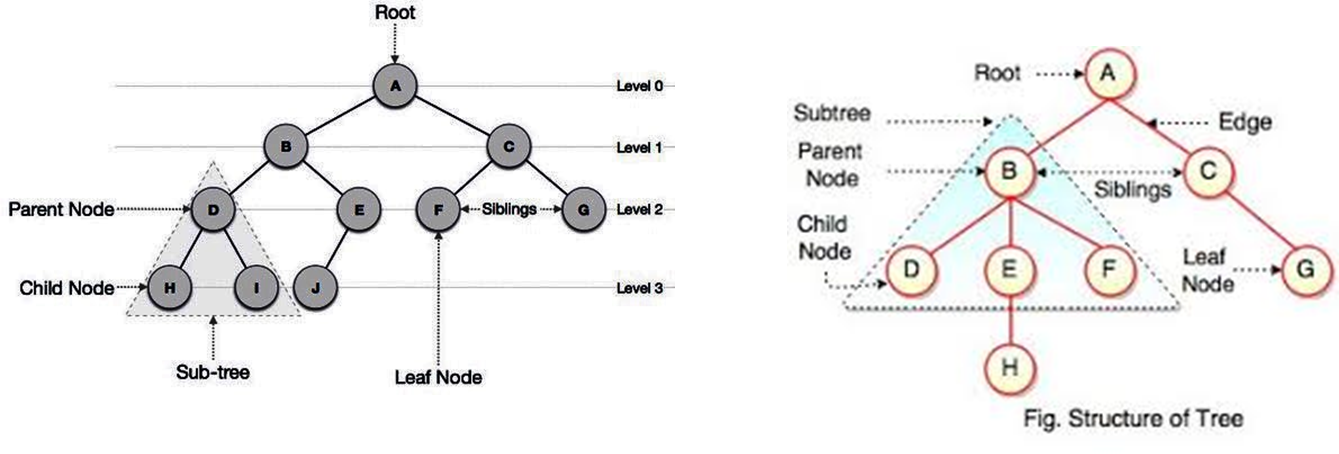
**Introduction of Tree** : -

* Some applications represent information in hierarchical form.
* In such situation we have to represent a **one to many** relationships.
* This hierarchical relationship between data elements can be easily represented using a non-linear data structure called **trees**.
* A tree is a non-linear data structure used to represent data containing a hierarchical relationship between elements. For example - Records, family trees and tables of contents.
* If the elements of the data structures do not form a sequence, it is said to be a non-linear data structure, i.e., there does not exist an order between these elements.
* For example, if we have to store the directory structure of a computer, a linear representation will not be useful. We have to use a non-linear representation i.e. given by **TREE.**

**Common uses of trees are:**  
1. To manipulate hierarchical data.  
2. To make information easily searchable.  
3. To manipulate sorted lists of data.

**Definition of a Tree** : - A tree is a finite set of one or more nodes such that -

1. There is a specially designated node called root.
2. The remaining nodes are partitioned into n > = 0 disjoint sets T1 to Tn where each of those sets is a tree. T1 to Tn are called the sub trees of the root.



**Tree Terminology :-**

1. **Root**: the first node is called as Root Node. In any tree, there must be only one root node. this node does not have any parent.
2. **Null Tree**: A tree with no nodes is a Null Tree.
3. **Node**: In tree data structure, every individual element is called as Node. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure. A node of a tree is an item of information along with the branches to other nodes.
4. **Parent Node**: the node which is predecessor of any node is called as parent node. In simple words, The node having further sub-branches is called parent node.
5. **Child Node**: When a predecessor of a node is parent then all successor nodes are called child nodes. In a tree data structure, the node which is descendant of any node is called as child Node.
6. **Leaf Node/Terminal Node**: In a tree data structure, the node which does not have a child is called as leaf node. Leaf node is a terminal node of a tree.
7. **Siblings:** In a tree data structure, nodes which belong to same Parent are called as siblings
8. **Predecessor:** [**former/previous**]:Every node in binary tree, except the root has unique parent called Predecessor of N. For example, E is predecessor of H in Fig.
9. **Descendents:** The descendents of a node are all those nodes which are reachable from that node e.g. E, F, and H are descendents of B.
10. **Ancestors:** The ancestors of a node are all the nodes along the path from the root to that node. B and A are ancestors of E in Fig.
11. **Degree of a node:** The number of sub trees of a node is called its degree. The degree of A is 2, B is 2, D is 0. The degree of the leaf node is zero.
12. **Degree of the Tree:** The degree of a tree is the maximum degree of the nodes in the tree. The degree of the tree shown in Fig. is 3.
13. **In-degree:** The in-degree of a node is the total number of edges coming to that node. Indegree of root node is always zero.
14. **Out-degree:** The out-degree of a node is the total number of edges going outside from the node. The degree of the leaf node is zero.
15. **level of a Tree** : Level is a rank of tree hierarchy. The whole tree structure is leveled. The level of root node is always at 0. The immediate children of root are at level 1 and their immediate children are at level 2 and so on as shown in Fig.
16. **Depth or Height or degree of a Tree** : The depth of a tree is the length of the longest path from the root to leaf node. In a tree data structure, the total number of edges from root node to a particular node is called as depth of that node.
17. **Edge:** In a tree data structure, the connecting link between any two nodes is called as edge.
18. **Directed Edge:** Edge which has a direction is called directed edge.
19. **Path:** A sequence of consecutive edges is called a path. The path length in tree is a number of edges between nodes from the root to the node. For example in Fig. path length is A, B, E, J or A, C, G.
20. **Branch:** Terminal node is called a leaf and path ending in leaf node is called branch.
21. **Height**: The highest number of nodes that is possible in a way starting from the first node (root) to a leaf node is called the height of tree.
22. **Forest:** Forest is a set of several trees that are not linked to each other

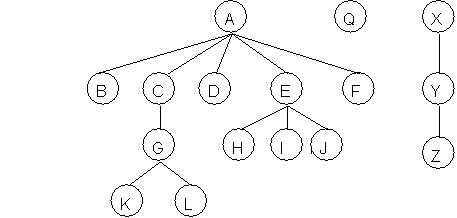
**Types of Trees** : -

The different types of trees are :  
1) General tree  
2) Binary tree

3) Binary search tree  
4) Strictly binary tree  
5) Complete binary tree  
6) Skewed binary tree

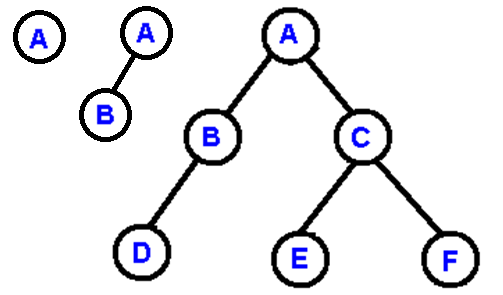
**1) General tree : -**

* In a general tree, number of children per node is not limited to two.
* There is NO fixed structure of a general tree**.**
* Since, the number of children per node can vary greatly; it might not be feasible to make the children direct links in the node.
* Children of a node can be stored in a linked list with parent node storing the address of its leftmost child. And left most child store address of all childs.
* Example : Some nodes may have 1 or no sub-trees. The others may have 3 or 4 sub-trees.



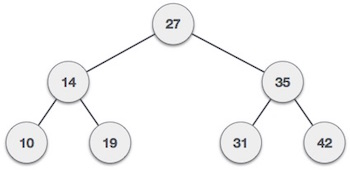
2) **Binary tree : -**

* A Binary tree is a tree where every node can have maximum of two children is called as Binary Tree.
* It is a special type of tree where every node has either **no children, one child or two children**. Child of a node in a binary tree on the left is called left child. Child of a node in a binary tree on the right is called right child.



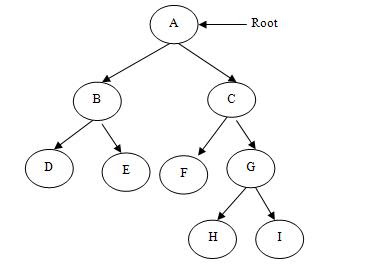
3) **Binary search tree** : -

* A binary search tree is a binary tree in which the nodes are arranged according to their values.
* The left node has a value less than its parent and the right node has a value greater than or equal to the parent node,
* i.e all nodes in the left subtree have values less than the root and
* all the nodes have values greater than or equal to root in right subtree.
* The Left and right sub-trees of node are again binary search trees.



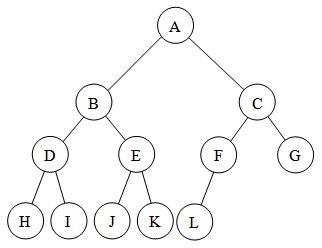
4) **Strictly binary tree** : - [**full binary tree** ][**Extended Binary Tree**]

* When every **non-leaf node** in binary tree is filled with **left and right sub-trees i.e 2 childs**, then the tree is called strictly binary tree.



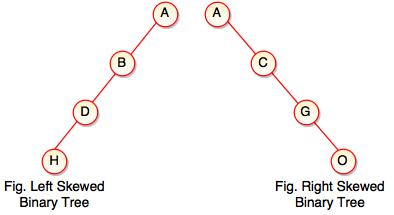
5) **Complete binary tree** : -

* A complete binary tree is a strictly binary tree with all its leaf nodes at the same level tree.
* Means all **non-leaf node** in binary tree is filled with **left and right sub-trees and**  also
* all its leaf nodes at the **same** level tree



6) **Skewed binary tree** : -

* A tree is called skew if  
  All the nodes of a tree are attached to one side only as shown in Fig.



**Representation of Trees**: -

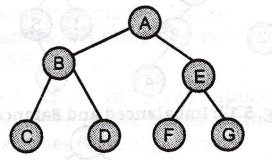
* **Array [Sequential/ Static]** Implementation of Tree : -

**Rules**: -

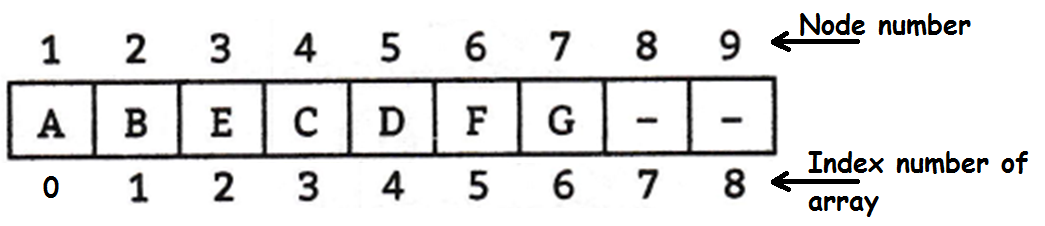
1. If ‘n’ is the number of nodes in the tree.
2. starting from the zero Level where the root is present. The root node is stored in the first memory location as the first element in the array.
3. Its left child is stored at index 2i + 1. if 2i+1 < n, and If 2i+1 > n, then i has no left child.
4. Right child is stored at index 2i+2, if 2i+2<n, If (2i + 2) > n, then i has no right child.
5. To find the parent node of any ith node use i/2. if i≠1,
6. if i=1 then it is root node.

**[Very Imp – when you calculate the left and right child then take ‘i=index value’, and when calculating the parent of any node then take ‘i=node number’.]**

e.g



* Here, the number of levels is 3. Therefore, we need an array of size 23 -1=7 elements. The array representation as shown after applying the rules :



**Advantages of Sequential Representation**: -

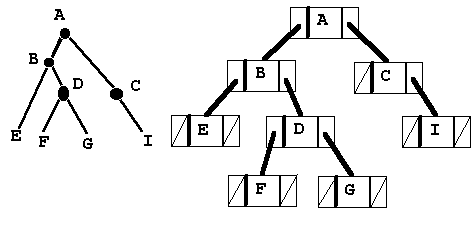
1. Very easy and simple method.
2. Any node can be accessed from any other node by calculating the index.
3. Data are stored without any pointers.
4. Programming languages, where dynamic memory allocation is not possible, e.g BASIC and FORTRAN, array representation is the only means to store a tree.
5. In this method tree searching is faster than linked list.

**Disadvantages of Sequential Representation:** -

1. There is a limitation on its size i.e. we cannot store information of more nodes than the specified array size.
2. If the tree is not an almost complete binary tree, many array positions will be unutilized.
3. Nodes cannot be inserted or deleted. Hence, the linked representation is preferred.

**Linked List Representation of Tree**: -

* In this representation we use singly linked list. In this representation each node requires three fields.
* One for the link of the left child,
* Second field for representing the information associated with the node, and
* The Third is used to represent the link of right child.
* When a node has no child then the corresponding pointer fields are null The left and right field of a node is pointer to left and the right child of that node.

****

struct node

{

int data;

struct node \*left;

struct node \*right;

};

**Advantages of Linked Representation : -**

1. No wastage of memory.
2. also there is no need to have prior knowledge of depth of the tree.
3. Using dynamic memory concept one can create as many memory (nodes) as required.
4. Insertions and deletions operations can be done without moving the other nodes.
5. Processing time reduced.

**Disadvantages of Linked Representation : -**

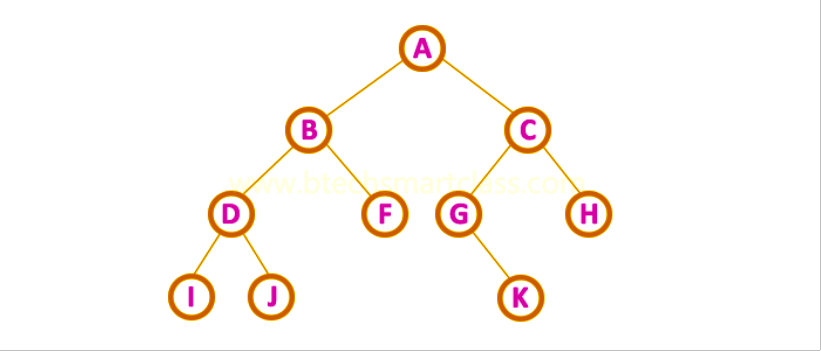
1. This representation does not provide direct access to a node and special algorithms are required.
2. Needs additional space in each node for storing the left and right sub trees.
3. Programming languages like BASIC, FORTRAN which do not support dynamic memory cannot use LL for tree.

**Binary tree Traversal. OR Tree Traversal**: -

* Traversal means visiting every node of a binary tree.
* There are many operations that often are performed on tree such as
  + - search a node,
    - print some information,
    - insert a node,
    - Delete a node etc.
* All such operations need traversal through a tree.
* Also 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.
* There are three types of binary tree traversals.

**In - Order Traversal ( leftChild - root - rightChild ):-**

* In order traversal is also called as symmetric traversal.
* This can be defined as follows:
* 1. Traverse the left sub-tree of root node in in-order.
* 2. Visit the root node.
* 3. Traverse the right sub-tree of root node in in-order.



**In-Order Traversal for above example of binary tree is**

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

**Algorithm:**   
Step 1 : Repeat step 2 to 4

Step 2 : while tree! =NULL.  
Step 3 : INORDER (TREE->LEFT)  
Step 4 : Write "Tree ->Data"  
Step 5: INORDER (Tree->Right)  
 [END of while]  
Step 6 : END

void inorder(struct node \*root)

{

if( root != NULL)

{

inorder(root->left); //recursive function call

printf("%d ",root->data);

inorder(root->right);

}

}

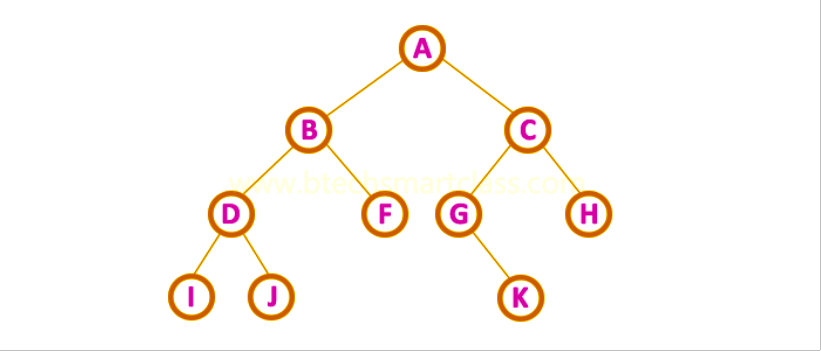
**Pre - Order Traversal ( root - leftChild - rightChild ) : -**

* In preorder traversal, each node is processed before (pre) either of its left subtree or right subtree.
* Preorder traversal is defined as follows:

1. Visit the root.

2. Traverse to the left subtree in preorder.

3. Traverse the right subtree in preorder.



* **Pre-Order Traversal for above example binary tree is**

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

**Algorithm** :  
Step 1 : Repeat step 2 to 4

Step 2 : while tree!=NULL.  
Step 3 : Write "Tree ->Data"  
Step 4 : Preorder (Tree->Left)  
Step 5 : Preorder (Tree->Right)  
 [END of while]  
Step 6 : END.

void preorder(struct node \*root)

{

if( root != NULL)

{

printf("%d ",root->data);

preorder(root->left); //recursive function call

preorder(root->right);

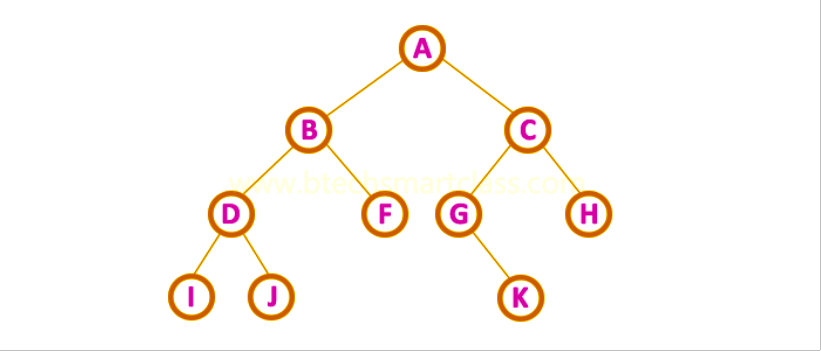
}

}

**Post - Order Traversal ( leftChild - rightChild - root ) : -**

* In Post-Order traversal, the root node is visited after left child and right child.
* This traversal is defined as follows:

1. Traverse the left child (sub-tree) of root node in post-order.
2. Traverse the right child (sub-tree) of root node in post-order.
3. Visit the root 'node'.



**Post-Order Traversal for above example binary tree is**   
 **I - J - D - F - B - K - G - H - C – A.**

**Algorithm** :  
Step 1 : Repeat step 2 to 4

Step 2: while tree! = NULL.  
Step 3: POSTORDER (TREE->LEFT)  
Step 4 : POSTORDER (Tree->Right)  
Step 5 : Write "Tree ->Data"  
 [END of while]  
Step 6: END.

void postorder(struct node \*root)

{

if( root != NULL)

{

postorder(root->left); //recursive function call

postorder(root->right);

printf("%d ",root->data);

}

}

**--------------------------------------------------------------------------------------------------------------------------------------------------**

**// C** Program for NON- recursive traversing on BST.

#include <stdio.h>

#include<malloc.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*right;

struct node \*left;

};

struct node \*Create(struct node \*, int);

void NonRec\_Preorder(struct node \*);

void NonRec\_Inorder(struct node \*);

void NonRec\_Postorder(struct node \*);

int main()

{

struct node \*root = NULL;

int choice, item, n, i;

printf("\n\*\*\* Binary Search Tree \*\*\*\n");

printf("\nl. Creation of BST");

printf("\n2. Traverse in Inorder");

printf("\n3. Traverse in Preorder");

printf("\n4. Traverse in Postorder");

printf("\n5. Exit\n");

while(l)

{

printf("\nEnter Your Choice :(1.Create 2.Inorder 3.Preorder 4.Postorder

5.Exit)\n");

scanf("%d",&choice);

switch(choice)

{

case 1:

root = NULL;

printf("Enter number of nodes:\n");

scanf("%d",&n);

for(i = 1; i <= n; i++)

{

printf("\n Enter data for node %d : ", i);

scanf("%d", &item);

root = Create(root, item);

}

break;

case 2:

NonRec\_Inorder(root);

break;

case 3:

NonRec\_Preorder(root);

break;

case 4:

NonRec\_Postorder(root);

break;

case 5:

exit(0);

default:

printf("Wrong Choice !!\n");

}

}

return 0;

}

struct node \*Create(struct node \*root, int item)

{

if(root == NULL)

{

root = (struct node \*)malloc(sizeof(struct node));

root->left = root->right = NULL;

root->data = item;

return root;

}

else

{

if(item < root->data )

root->left = Create(root->left,item); //recursive function call

else if(item > root->data )

root->right = Create(root->right,item);

else

printf(" Duplicate Element is Not Allowed !!!");

return(root);

}

}

void NonRec\_Preorder(struct node \*root)

{

struct node \*stack[100];

int top = 0;

if (root == NULL)

{

return;

}

stack[top++] = root;

while (top != 0)

{

root = stack[--top];

printf("%d % root->data);

if (root->right != NULL)

stack[top++] = root->right;

if (root->left != NULL)

stack[top++] = root->left;

}

}

void NonRec\_Inorder(struct node \*root)

{

struct node \*stack[100];

int top=0;

while(l)

{

while (root != NULL )

{

stack[top++] = root;

root = root->left;

}

if(top==0)

return;

root = stack[--top];

printf("%d ", root->data);

root = root->right;

}

}

void NonRec\_Postorder(struct node \*root)

{

struct

{

struct node \*nodel;

unsigned vleft :1; //l bit will be used to store the value

unsigned vright :1;

}stack[100];

int top = 1;

while ( top >= 1 )

{

// Move to the left subtree if present and not visited

if(root->left != NULL && !stack[top].vleft)

{

stack[top].vleft = 1;

stack[top++].nodel = root;

root = root->left;

continue;

}

// Move to the right subtree if present and not visited

if(root->right != NULL && !stack[top].vright )

{

stack[top].vright - 1;

stack[top++].nodel = root;

root = root->right;

continue;

}

printf("%d ", root->data);

// Clean up the stack

stack[top].vleft = 0;

stack[top],vright = 0;

// Move up

root = stack[--top].nodel;

}

}

---------------------------------------------------------------------------------

**Menu driven program for binary search tree creation and tree traversing. [Recursive]**

#include<stdio.h>

#include<malloc.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*right;

struct node \*left;

};

struct node \*Create(struct node \*, int);

void Inorder(struct node \*);

void Preorder(struct node \*);

void Postorder(struct node \*);

int main()

{

struct node \*root = NULL;

setbuf(stdout, NULL);

int choice, item, n, i;

printf("\n\*\*\* Binary Search Tree \*\*\*\n");

printf("\nl. Creation of BST");

printf("\n2. Traverse in Inorder");

printf("\n3. Traverse in Preorder");

printf("\n4. Traverse in Postorder");

printf("\n5. Exit\n");

while(l)

{

printf("\nEnter Your Choice :(1.Create 2.Inorder 3.Preorder 4.Postorder 5.Exit)\n");

scanf("%d",&choice);

switch(choice)

{

case 1:

root = NULL;

printf("Enter number of nodes:\n");

scanf("%d",&n);

for(i = 1; i <= n; i++)

{

printf("\n Enter data for node %d : ", i);

scanf("%d", &item);

root = Create(root,item);

}

break;

case 2:

Inorder(root);

break;

case 3:

Preorder(root);

Break;

case 4:

Postorder(root);

break;

case 5:

exit(0);

default:

printf("Wrong Choice !!\n");

}

}

return 0;

}

struct node \*Create(struct node \*root, int item)

{

if(root == NULL)

{

root = (struct node \*)malloc(sizeof(struct node));

root->left = root->right = NULL;

root->data = item;

return root;

}

else

{

if(item < root->data )

root->left = Create(root->left,item); //recursive function call

elseif(item > root->data )

root->right = Create(root->right,item);

else

printf(" Duplicate Element is Not Allowed !!!");

return(root);

}

}

void Inorder(struct node \*root)

{

if( root != NULL)

{

Inorder(root->left); //recursive function call

printf("%d ",root->data);

Inorder(root->right);

}

}

void Preorder(struct node \*root)

{

if( root != NULL)

{

printf("%d ",root->data);

Preorder(root->left);//recursive function call

Preorder(root->right);

}

}

void Postorder(struct node \*root)

{

if( root != NULL)

{

Postorder(root->left);//recursive function call

Postorder(root->right);

printf("%d ",root->data);

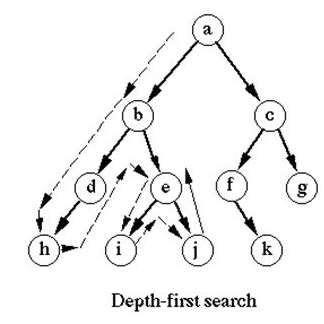
}

}

---------------------------------------------------------------------------------

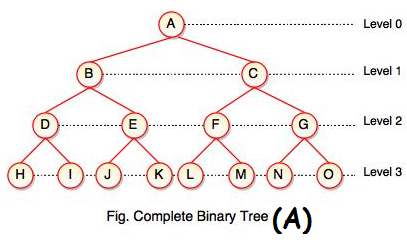
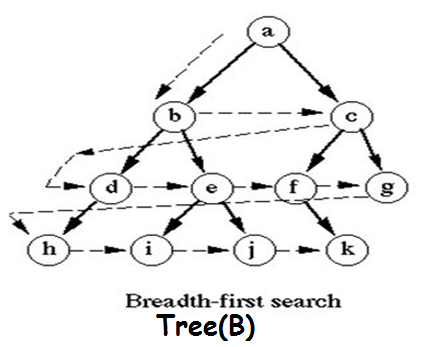
**Depth first Search [DFS] of tree : -**

* In this the data can be search or traverse according to the depth of the tree.
* The DFS is same as the pre-order traversal of the tree.
* The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

****

**Breadth First Search of a tree.[Level Order Tree Traversal] : -**

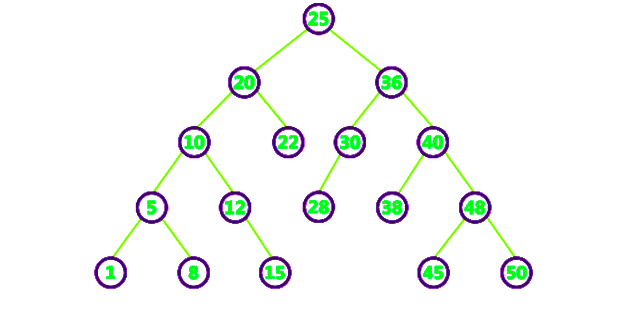
* This method displays all the node level by level of a tree.
* So the BFS of the tree(A) will be – ABCDEFGHIJKLMNO.
* And BFS for tree(B) is – a b c d e f g h I j k

** **

**Binary Search Tree**. :-

* Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.
* It has the following properties.

1. The left subtree of a root node contains only nodes with data less than the root node’s data.
2. The right subtree of a root node contains only nodes with data greater than the root node’s data.
3. The left and right subtree each must also be a binary search tree.



* **In order to create a binary search tree on given data use following steps:**
* Step l Read a data in item variable.
* Step 2 Allocate memory for a new node and store the address in pointer root.
* Step 3 Store the data x in the node root.
* Step 4 If(data<root.data).
* Step 5 Recursively create the left subtree of root and make it the left child of root.
* Step 6 If(data>root.data).
* Step 7 Recursively create the right subtree of root and make it the right child of root.

**Algorithm**: create (root,data)

* Root is initially NULL, data is key which you want to insert
* 1 If root is NULL
* 2. create a node with data
* 3. return
* 4. else
* 5. If data is less than root.data
* 6. root.left=create( root.left, data)
* 7. else
* 8. root.right=create( root.right, data)

**Operations on a Binary Search Tree : -**

The following operations are performed on a binary earch tree...

* **Search**
* **Insertion**
* **Deletion**

**Search Operation in BST : -**

* **Step 1:** Read the search element from the user
* **Step 2:** Compare, the search element with the value of root node in the tree.
* **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
* **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
* **Step 5:** If search element is smaller, then continue the search process in left subtree.
* **Step 6:** If search element is larger, then continue the search process in right subtree.
* **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
* **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
* **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.
* In a binary search tree, the search operation is performed with **O(log n)** time complexity

**Algorithm for Search in BST:**

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;

}

}

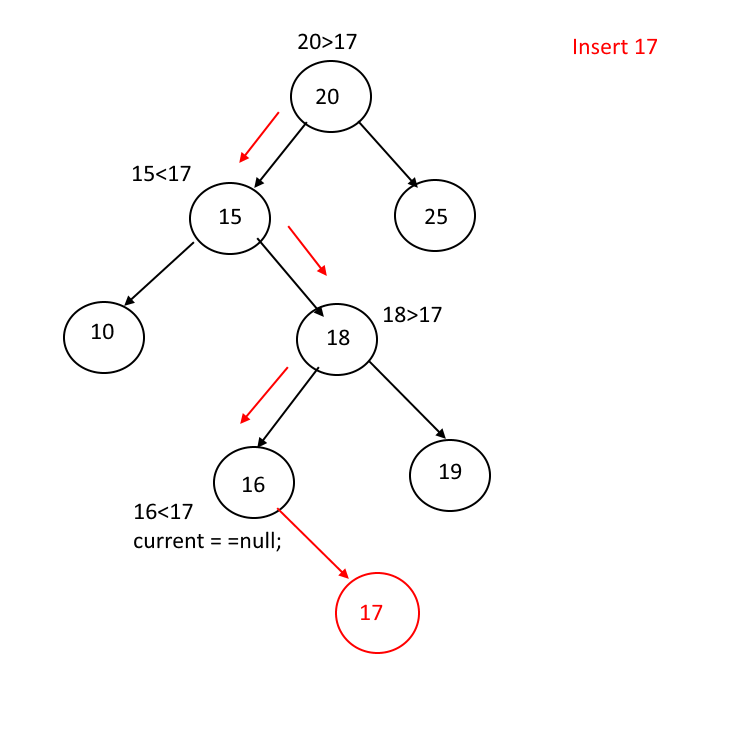
}

return current;

}

**Insertion Operation in BST : -**

* In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...
* **Step 1:** Create a newNode with given value and set its **left** and **right** to **NULL**.
* **Step 2:** Check whether tree is Empty.
* **Step 3:** If the tree is **Empty**, then set set **root** to **newNode**.
* **Step 4:** If the tree is **Not Empty**, then check whether value of newNode is **smaller** or **larger** than the node (here it is root node).
* **Step 5:** If newNode is **smaller** than **or equal** to the node, then move to its **left** child. If newNode is **larger** than the node, then move to its **right** child.
* **Step 6:** Repeat the above step until we reach to a **leaf** node (e.i., reach to NULL).
* **Step 7:** After reaching a leaf node, then isert the newNode as **left child** if newNode is **smaller or equal** to that leaf else insert it as **right child**.



**Algorithm for Insertion in BST:**

void insert(int data)

{

struct node \*tempNode = (struct node\*) malloc(sizeof(struct node));

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL;

//if tree is empty

if(root == NULL) {

root = tempNode;

} else {

current = root;

parent = NULL;

while(1) {

parent = current;

//go to left of the tree

if(data < parent->data) {

current = current->leftChild;

//insert to the left

if(current == NULL) {

parent->leftChild = tempNode;

return;

}

} //go to right of the tree

else {

current = current->rightChild;

//insert to the right

if(current == NULL) {

parent->rightChild = tempNode;

return;

}

}

}

}

}

**Deletion Operation in BST : -**

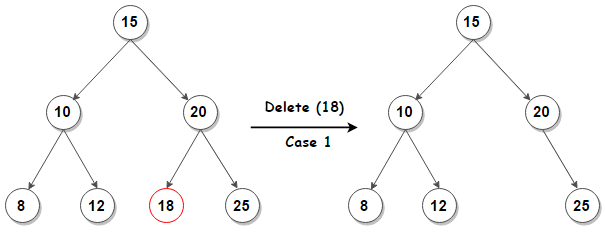
* In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree has follwing three cases...
* **Case 1: Deleting a Leaf node (A node with no children)**
* **Case 2: Deleting a node with one child**
* **Case 3: Deleting a node with two children**
* **Case 1: Deleting a leaf node : -**

We use the following steps to delete a leaf node from BST...

**Step 1:** **Find** the node to be deleted using **search operation**

**Step 2:** simply make the pointer field of parent to NULL (If it is a leaf) and terminate the function. OR

**Step 3:** Delete the node using **free** function (If it is a leaf) and terminate the function.



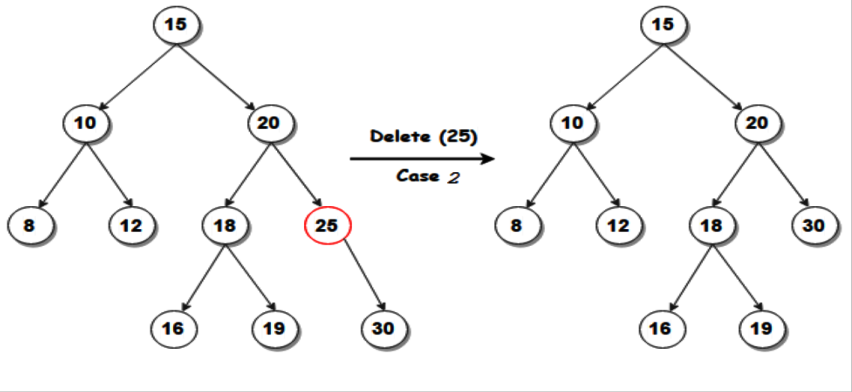
* **Case 2: Deleting a node with one child :-**

We use the following steps to delete a node with one child from BST...

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.



* **Case 3: Deleting a node with two children : -**

We use the following steps to delete a node with two children from BST...

**Step 1:** **Find** the node to be deleted using **search operation**

**Step 2:** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.

**[For deleting the node having 2 childs then we can take either the inorder successor OR the inorder predecessor to be placed at the deleting node.]**

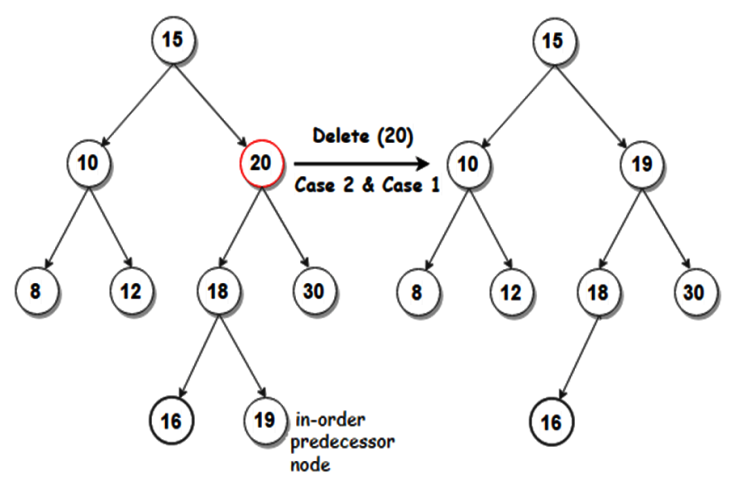
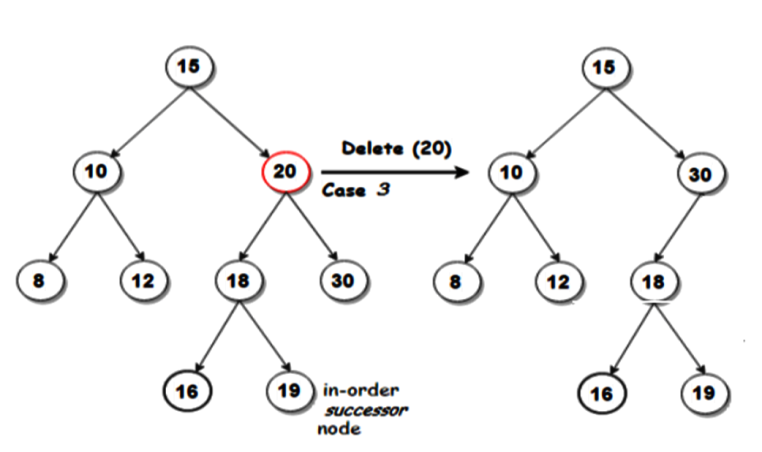
**Step 3:** **Swap** both **deleting node** and node which found in above step.

**Step 4:** Then, check whether deleting node came to **case 1** or **case 2** else goto steps 2

**Step 5:** If it comes to **case 1**, then delete using case 1 logic.

**Step 6:** If it comes to **case 2**, then delete using case 2 logic.

**Step 7:** Repeat the same process until node is deleted from the tree.

**Applications of a Binary Search Tree (BST):**

1 Sorting a set of data.

2. Removing the redundancies from a collection of data.

3. Searching for an element.

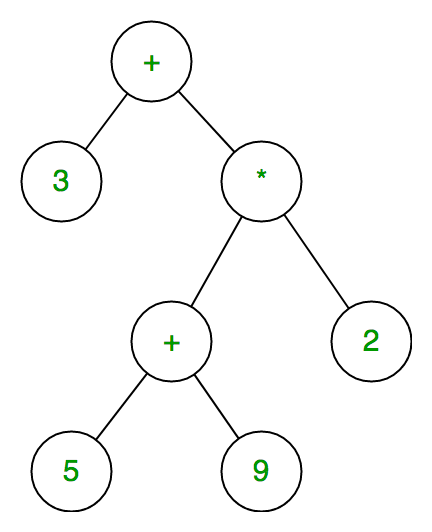
**Expression Tree** : -

* An expression tree is a representation of expressions arranged in a tree-like data structure.
* Expression tree is a binary tree in which each internal node represent operator and each leaf node represent operand.

**Expression has following properties:**

1. Each **leaf** node contains a single operand.
2. Each **non-leaf** node contains a single binary operator.
3. The left and right subtrees of an operator node represent sub expressions that must be evaluated before applying the operator at the root of the sub-tree.
4. The levels of the nodes in the tree indicate their relative precedence of evaluation (we do not need parentheses to indicate precedence).
5. Operations at higher levels of the tree are evaluated after than those below them.
6. The operation at the root is always the last operation performed.

Consider the Expression tree : -

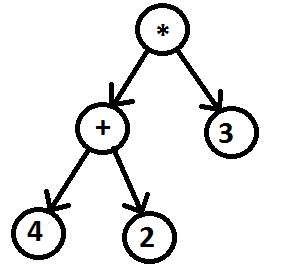


The in-order traversal of expression tree gives you INFIX form of expression tree is : **3 + 5+9\*2**.

The pre-order traversal of expression tree gives you PREFIX form of expression tree is : +**3 \*+592**.

The post-order traversal of expression tree gives you POSTFIX form of expression tree is : **359+2\*+**.

Consider the Expression tree : -

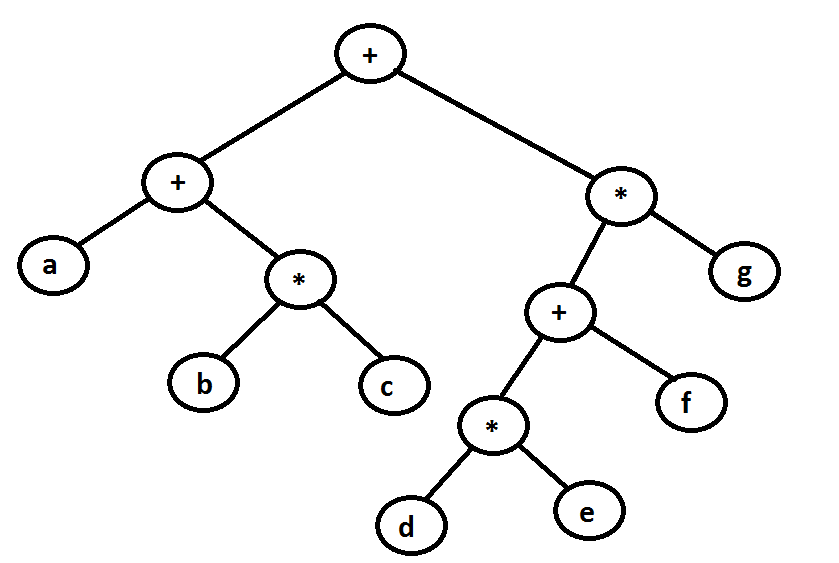


When we apply in-order traversal on given expression tree we will get INFIX expression: 4+2\*3.

When we apply pre-order traversal on given expression tree we will get PREFIX expression: \*+423.

When we apply post-order traversal on given expression tree we will get POST FIX expression: 42+3\*.

**Example 1:** Let consider the expression tree in Fig. 5.51 and evaluate the expression and give the infix, prefix and postfix expression.



**Solution:** Start the evaluation from bottom level.

1. Firstly the expression (d\*e) gets evaluated.
2. Secondly (b\*c) on left side of tree and ((d\*e)+f) on right side of tree get evaluated.
3. Thirdly (a+(b\*c)) on left side of tree and (((d\*e)+f)\*g) on right side of tree get evaluated.
4. Finally (a+(b\*c))+(((d\*e)+f)\*g) get evaluated.

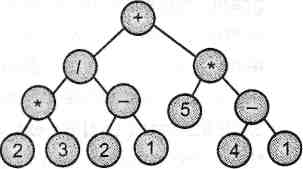
- Above tree represent (a + (b \* c)) + (((d \* e ) + f)*\**g *)* expression.

- Pre-order traversal on given expression tree :++a\*bc\*+\*defg.

- Post-order traversal on given expression tree: abc\* + de\*f + g\*+.

- In-order traversal on given expression tree : a+b\*c+d\*e+f\*g.

**Example 2:** Let consider the following expression tree and evaluate the expression and give the infix, prefix and postfix expression.



**Solution:** Above tree represent ((2\*3)/(2-1)+(5\*(4-1))) expression.

Pre-order traversal on given expression tree: +/\*23 - 21\*5 - 41

Post-order traversal on given expression tree: 23\*21 - /541 - \*+

In-order traversal on given expression tree: 2\*3/2 -1+5\*4 -1

Q. Compares tree and graph data structures.

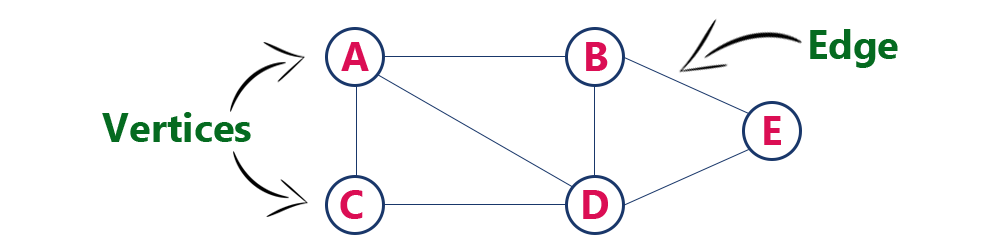
|  |  |  |
| --- | --- | --- |
| Sr. No. | Tree | Graph |
| 1. | A tree is a data structure in which each node is attached to one or more nodes as children. | A graph is a collection of vertices or nodes, which are joined as pairs by lines or edges. |
| 2. | This is a non-linear data structure. | This is also a non-linear data structure. |
| 3. | All the trees can be graphs. | All the graphs are not trees. |
| 4. | The common tree traversal methods are inorder, preorder, postorder traversals | The two common graph traversal method are Breadth First Search (BFS) and Depth First Search (DFS) |
| 5. | It is undirected and connected. | It can be directed or undirected, can be connected or not-connected. |
| 6. | It cannot be cyclic. | It can be cyclic or acyclic. |
| 7. | Give any diagram of Tree. | Give any diagram of Graph. |

**Graph : -**

* A graph is a pictorial representation of the relationships between entities.
* A graph G is a collection of nodes which are called Vertices V, connected in pairs by line segments, called as Edges E.

**Definition :** -

* Generally, a graph **G** is represented as **G = ( V , E )**, where **V is set of vertices** and **E is set of edges**.
* Sets of vertices are represented as V(G) and
* sets of edges are represented as E(G).
* So we can represent a graph as G = (V, E).



**Example**

The above is a graph with 5 vertices and 6 edges.  
This graph G can be defined as G = ( V , E )  
Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.

**Application of Graphs** :

It is used to describe a wide variety of relationships between objects.

* For example the following information can be represented by graphs :
* Family trees in which the member nodes have an edge from parent to each of their children.
* Transportation networks in which nodes are airports, intersections, ports etc. The edges can be airline flights, one-way roads, shipping routes etc.

**Graph Terminology : -**

**Vertex : -**

* A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

**Edge : -**

* An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is an edge with cost on it.
4. **Undirected Graph : -**A graph with only undirected edges is said to be undirected graph.
5. **Directed Graph : -** A graph with only directed edges is said to be directed graph.
6. **Mixed Graph : -**A graph with undirected and directed edges is said to be mixed graph.
7. **End vertices or Endpoints : -** The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.
8. **Origin : -** If an edge is directed, its first endpoint is said to be origin of it.
9. **Destination : -**If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.
10. **Adjacent : -** If there is an edge between vertices A and B then both A and B are said to be adjacent.
11. **Incident : -** An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.
12. **Outgoing Edge : -** A directed edge is said to be outgoing edge on its orign vertex.
13. **Incoming Edge : -** A directed edge is said to be incoming edge on its destination vertex.
14. **Degree : -** Total number of edges connected to a vertex is said to be degree of that vertex.
15. **Indegree : -** Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
16. **Outdegree : -** Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.
17. **Parallel edges or Multiple edges : -** If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.
18. **Self-loop : -** An edge (undirected or directed) is a self-loop if its two endpoints coincide.
19. **Simple Graph : -** A graph is said to be simple if there are no parallel and self-loop edges.
20. **Path : -** A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

**Graph Representations**

Graph data structure is represented using following representations...

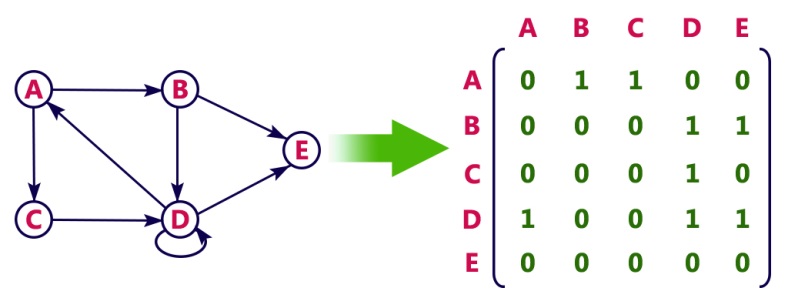
1. **Adjacency Matrix. [Sequential Representation.]**
2. **Adjacency List. [Linked Representation.]**
3. **Adjacency Matrix. [Sequential Representation.]**

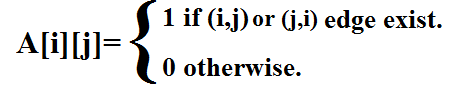
* Graph can be represented using a matrix of size total number of vertices by total number of vertices.
* Means graph with 4 vertices can be represented using a matrix of 4X4 class.
* In this matrix, rows and columns both represents vertices.
* This matrix is filled with either 1 or 0.
* Here, 1 represents there is an edge from row vertex to column vertex and
* 0 represents there is no edge from row vertex to column vertex.

****

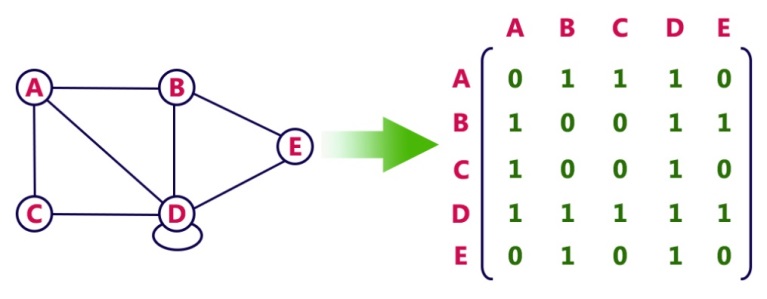
**Assume : -**

|  |  |
| --- | --- |
| G | Graph |
| V | Set of Vertices |
| (i,j) | Edge connecting Vi and Vj |
| A [ i , j ] | Adjacency matrix element |
| i , j | Vertices |

****



* **If G is Undirected graph then : -**



**Program to represent graph as Adjacency Matrix. [Sequential Representation.]**

#include <stdio.h>

#define MAX 10

void degree(int adj[][MAX],int x,int n)

{

int i,incount=0, outcount =0;

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

{

if( adj[x][i] ==1)

outcount++;

if( adj[i][x] ==l)

incount++;

}

printf( The indegree of the node %d is %d\n",x, incount++);

printf("The outdegree of the node %d is %d\n",x,outcount++);

}

int main()

{

int adj[MAX][MAX],n,I,j;

setbuf(stdout, NULL);

printf("Enter the total number of nodes in graph");

scanf("%d\&n);

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

for(j=0;j<n; j++)

{

printf("Enter Edge from %d to %d,(l: Edge 0: No edge) \n", i, j); scanf("%d",&adj[i][j]);

}

for(i=0ji<n; i++)

{

degree( adj, i, n);

}

return 0;

}

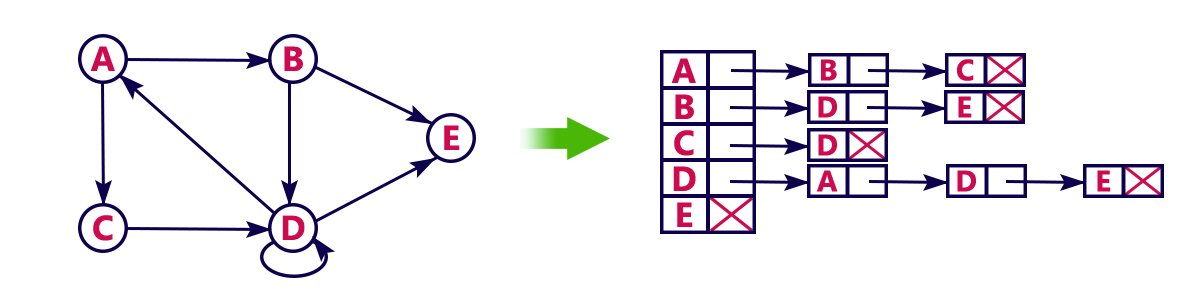
**Advantages of Array representation of Graph:**

1. Simple and easy to understand.
2. Graphs can be constructed at run-time.
3. Efficient for dense (lots of edges) graphs.
4. Simple and easy to program.
5. Most of the implicit graph problems can be easily represented as Adjacency Matrix with little or no pre-processing.
6. Adapts easily to different kinds of graphs.

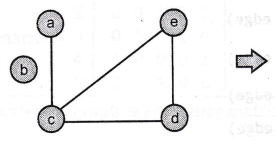
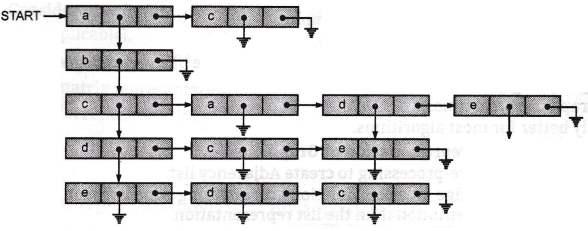
**Disadvantages of Array representation of Graph:**

1. Adjacency matrix consumes huge amount of memory for storing big or large graphs.
2. Adjacency matrix requires huge efforts for adding/removing a vertex.
3. The matrix representation of graph does not keeps track of the information related to the nodes.
4. Requires that graph access be a command rather than a computation.
5. The excessive amount of empty cells also reduce the efficiency of some of the Traversal algorithms.
6. **Adjacency List Implementation of graph. [Linked Representation.]**

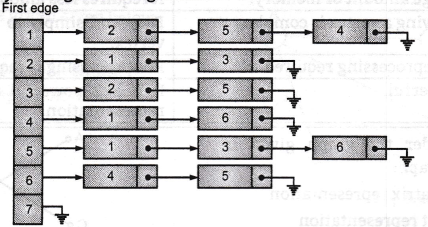
* For **Directed graph**, In this representation, every vertex of graph contains list of its adjacent vertices.



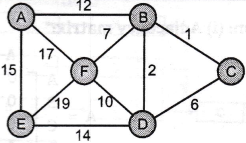
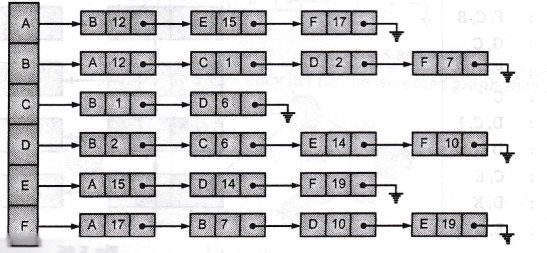
* **For Un-Directed Graph : -**

Example 1: Consider the undirected graph in Fig and provide adjacency list.

Example 2: Consider the weighted undirected graph in Fig and provide adjacency list.

**Advantages of Linked** List **representation of Graph:**

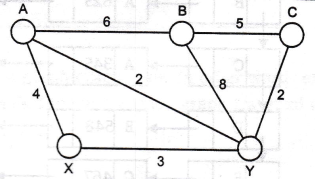
1. Less storage for sparse (few edges) graphs.
2. Easy to store additional information in the data structure like vertex degree, edge weight etc.
3. Better space usage.
4. Better graph traversal times.
5. Generally better for most algorithms.

**Disadvantages of Linked List representation of Graph:**

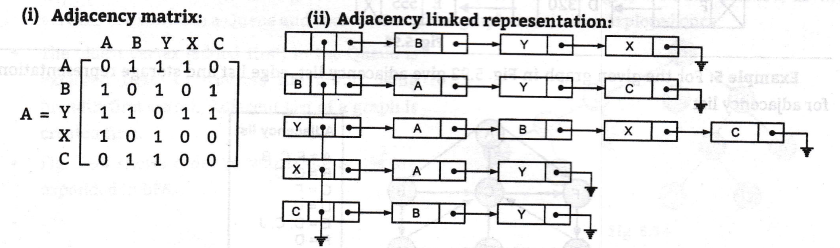
1. Generally takes some pre-processing to create Adjacency list.
2. Algorithms involving edge creation, deletion and querying edge between two vertices are better way with matrix representation than the list representation.
3. Adding/removing an edge to/from adjacent list is not so easy as for adjacency matrix.

Following table compares array and linked representation of graph:

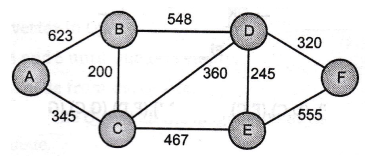
|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Array Representation of Graph** | **Linked Representation of Graph** |
| **1.** | This method uses matrix (array) in system memory. | This method uses linked list. |
| **2.** | This method is sequential in nature. | This method is linked (list) in nature. |
| **3.** | Indegree, outdegree, path calculations can be carried out easily. | Addition, deletion operations can be carried out easily. |
| **4.** | Execution time high. | Execution time low. |
| **5.** | It requires large amount of memory. | It requires less amount of memory. |
| **6.** | Adding/removing a vertex is complex. | Easy and simply to add/remove or delete a vertex. |
| **7.** | There is no preprocessing required. | Pre-processing is required. |
| **8.** | It is more powerful. | It is less powerful as compared to matrix representation. |

**Example 3:** Consider the graph given below For the graph :  
(i) Give adjacency matrix representation.   
(ii) Give adjacency list representation.

Ans : -



Example4: Create an adjacency matrix and adjacency list for the weighted graph shown in fig.

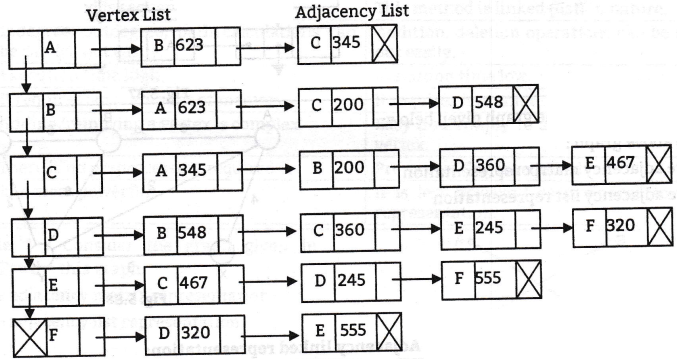


Ans : -

(i) Adjacency Matrix:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** | **F** |
| **A** | **0** | **623** | **345** | **0** | **0** | **0** |
| **B** | **623** | **0** | **200** | **548** | **0** | **0** |
| **C** | **345** | **200** | **0** | **360** | **467** | **0** |
| **D** | **0** | **548** | **360** | **0** | **245** | **320** |
| **E** | **0** | **0** | **467** | **245** | **0** | **555** |
| **F** | **0** | **0** | **0** | **320** | **555** | **0** |

(ii) Adjacency List:



**Graph traversal** : -

* Graph traversal is technique used for searching a vertex in a graph.
* The graph traversal is also used to decide the order of vertices to be visit in the search process.
* A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.
* There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

**DFS (Depth First Search): -**

* DFS traversal of a graph, produces a spanning tree as final result.
* Spanning Tree is a graph without any loops.
* We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

**Algorithm:**

Step 1 : Initialize all the vertices.

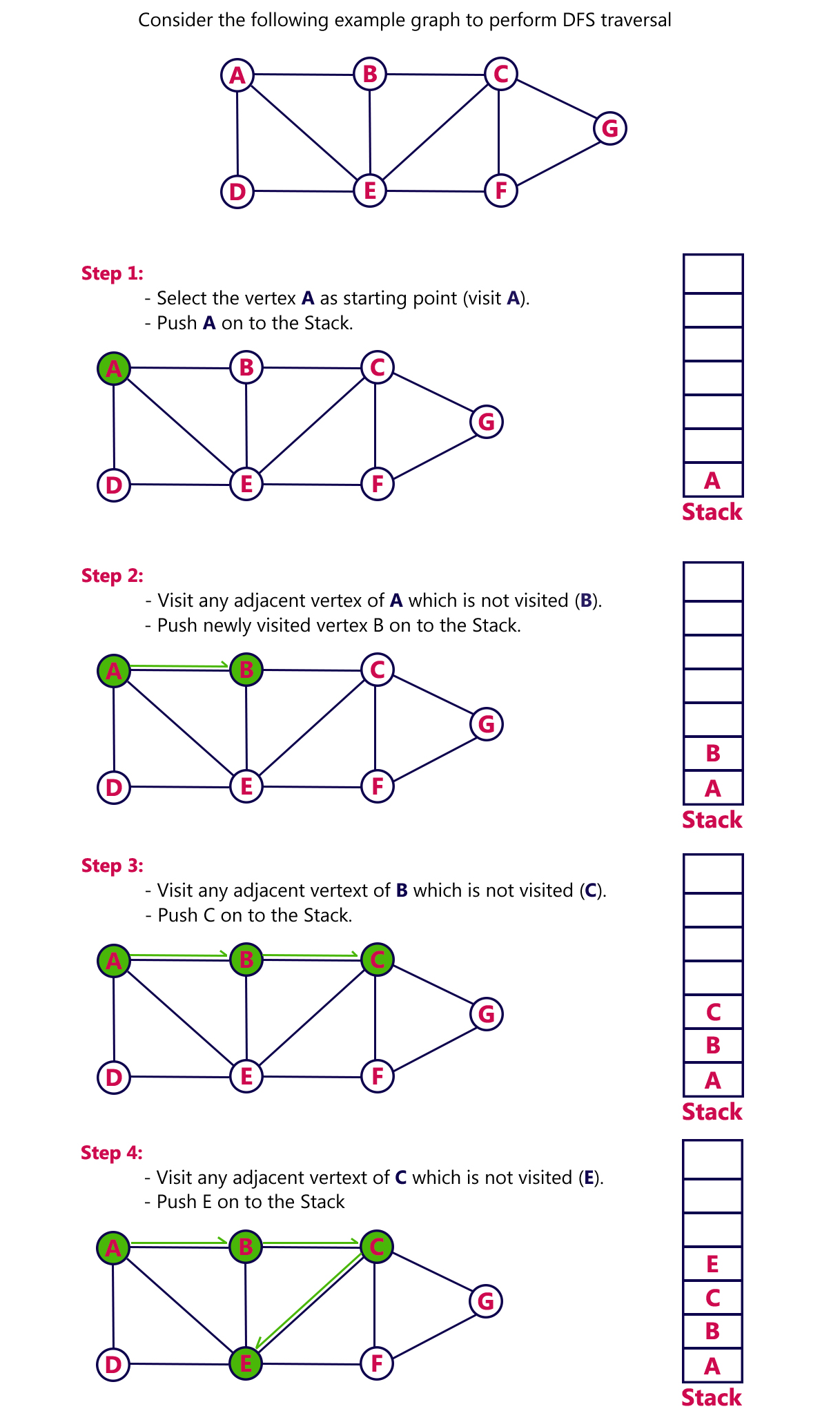
Step 2 : Push starting vertex onto the Stack.

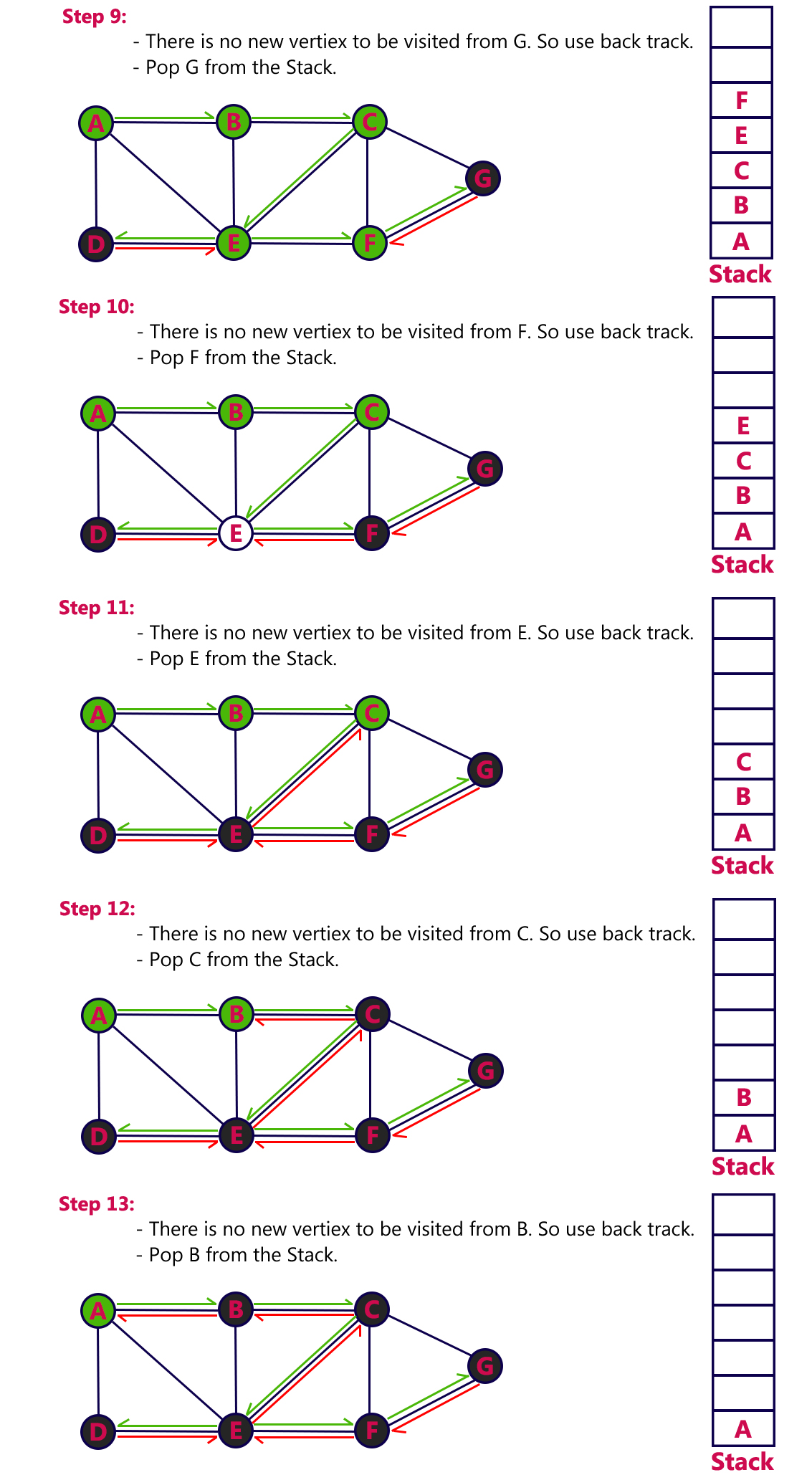
Step 3 : Repeat Steps 4 and 5 until Stack is empty.

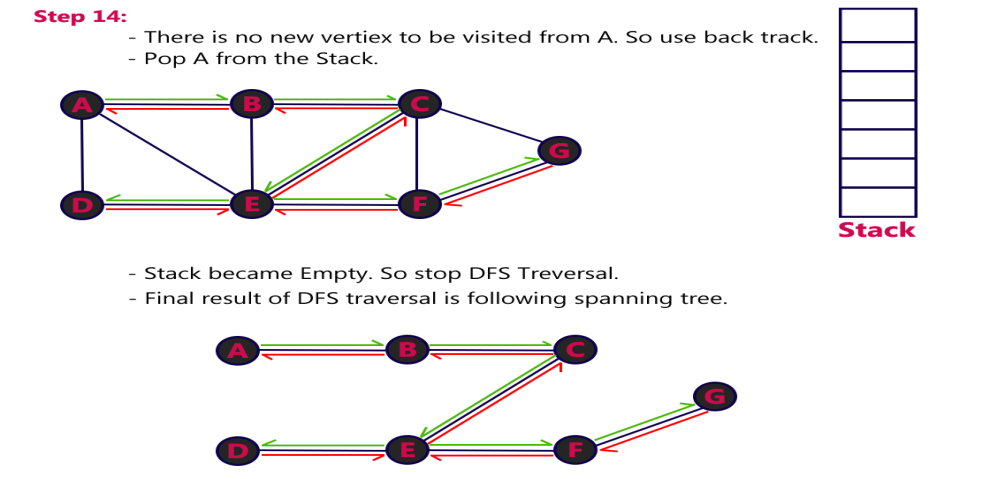
Step 4 : Pop the top vertex from the Stack.

Step 5 : Push onto stack all the neighbours of Popped vertex.

Step 6 : Exit.







**BFS (Breadth First Search)** : -

* BFS traversal of a graph, produces a **spanning tree** as final result.
* **Spanning Tree** is a graph without any loops.
* We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.
* We use the following steps to implement BFS traversal...

**Algorithm**:

**Step 1** : Initialize all the vertices and create adjacent list.

**Step 2** : Put Starting vertex in Queue.

**Step 3** : Repeat Step 4 and 5 until Queue is empty.

**Step 4** : Remove front node from the Queue.

**Step 5** : Add to the rear of the Queue all the neighbours of the deleted node that are not visited or added in a Queue.

**Step 5** : Exit.



|  |  |  |
| --- | --- | --- |
| Sr. No. | **DFS** | **BFS** |
| 1. | DFS stands for Depth First Search. | BFS stands for Breadth First Search. |
| 2. | DFS uses Stack implementation i.e. LIFO. | BFS uses Queue implementation i.e. FIFO. |
| 3. | DFS is faster. | Slower than DFS. |
| 4. | DFS requires less memory. | BFS uses a large amount of memory. |
| 5. | DFS is easy to implement in a procedural  language. | DFS is complex or hard to implement in a procedural language. |
| 6. | The aim of DFS algorithm is to traverse the  graph in such a way that it tries to go far from the root node. | The aim of BFS algorithm is to traverse the graph as close as possible to the root node. |
| 7. | Key ideas:  (i) Mark all vertices as "unvisited".  (ii) Visit first vertex.  (iii) Recursively visit its "unvisited  neighbours. | Key ideas:  (i) (undirected) i.e., mark all vertices as "unvisited".  (ii) Initialize a queue (to empty).  (iii) Find an unvisited vertex and apply breadth-first search to it.  (iv) Add the vertex's neighbours to the queue.  (v) Repeat extract a vertex from the queue, and add its "unvisited" neighbours to the queue. |
| 8. | It is an algorithm for traversing or searching a graph. Intuitively, one starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking. | It is a graph search algorithm that begins at  the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal. |
| 9. |  |  |

**Applications of Graphs:**

1. **Analysis of Electrical Networks:** Graphs, digraphs are used to analyze electrical networks. Electrical circuits can be represented by graphs.
2. **Study of Molecular Structure:** The study of molecular structure of chemical compounds requires graphs.
3. **Used for representation of Airlines Routes:** To represent airline route system airports are considered to be vertices and flights to be edges and distance or ticket price to be weights.
4. **Networks:** Computer networks or communication networks can be represented by graphs. Here, vertices are the computers and edges are communications lines. Weight could be bandwidth.
5. **Topological Sort:** Graphs are also used to sort elements using the topological sort algorithm.
6. **Finding Shortest Paths:** Graphs can be used to find shortest path between any two vertices, when it has several routes.
7. **Project Management:** Graphs are also used to Compute project completion time, delays, early start and late finish times for a project, which is made up of several tasks.
8. **State Transition Diagrams.** The nodes represent states, and the edges represent legal moves from state to state. For example, we could use a graph to represent legal moves in a game of tic-tac-toe.