Binary Tree Representation

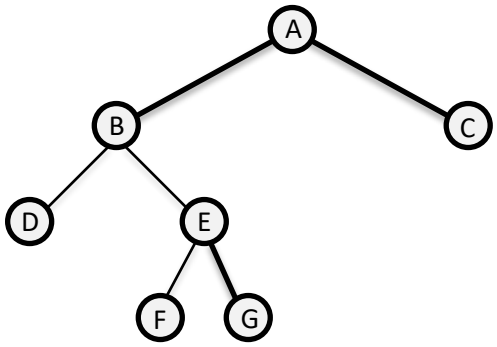
A binary tree can be represented either through an array or a linked list.

Array Representation

A binary tree can be stored sequentially using an array. This type of representation is static as the array of a required size has to be taken before going to store the tree elements into it.

In array representation, the elements are stored level by level, at first the element at level 0 is stored, and then the elements at level 1, from left to right, are stored, and so on. The root node is the first element stored in an array.

For example: The following binary tree is represented in an array as follow:



0	1	2	3	4	5	6
A	B	C	D	E	F	G

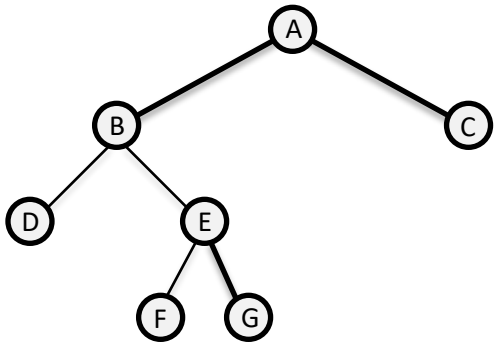
Pictorial Representation **Memory Representaion**

The advantage of an array representation is that it is simple to implement and the elements can be accessed sequentially very easily. The disadvantage of this representation is that before inserting the elements, the array of the required storage has to be declared. As this memory allocation is static, it is not possible to expand or contract the size of a previously declared array.

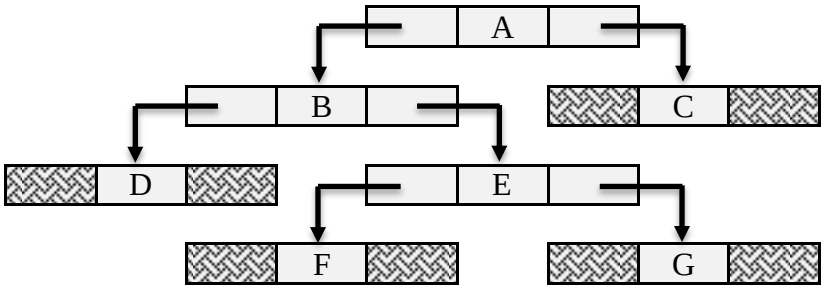
Linked Representation

A linked list representation of a binary tree is as follows:

Each node of a binary tree has three fields- A data field to store information, two address fields to store the addresses of the possible left child and right child. A pictorial representation is given below:



Pictorial Representation



Linked Representation

A linked list representation is dynamic and therefore enhancement of a tree is possible and no memory is wasted. Here, insertion and deletion operations involve to data movement. The drawback of this representation is that pointer fields are involved in each node in addition to the data fields hence more memory is occupied.

The structure of a node is defined as below:

```
struct node
{
        char data;
        struct node *lchild;
        struct node *rchild;
};
```

Here, the first field will store the information, the second field will point to the left child of a node and the third field will point to the right child of a node. If the node has no left child then the left address field will have null value (NULL), if the node has no right child then the right address field will have null value (NULL) stored in it.

Binary Tree Traversals

In linear data structures such as array, linked list, queues, stacks, etc have only one logical way to traverse the data elements from the first node. Unlike this, trees can be traversed in different ways. A binary tree can be traversed in various ways such as preorder traversal, inorder traversal and postorder traversal.

Binary tree traversals are done in non-linear way. In a binary tree the parent nodes, their left and right children are to be traversed. Therefore, six combinations of traversal are possible. If a parent node is denoted by N, the left child by L and the right child by R, then the possible combinations of traversal are NLR, NRL, LNR, LRN, RNL, RLN but only **NLR, LNR** and **LRN** are legal traversals.

These traversals are:

Preorder Traversal (NLR)

1. Visit the root node
2. Traverse the left subtree of the root in preorder
3. Traverse the right subtree of the root in preorder

The function for preorder traversal is given below:

```
void preorder (struct node *);                  /*function declaration*/

preorder (root);                                /*function calling*/

void preorder(struct node *ptr)                 /*function definition*/
{
        if (ptr != NULL)
        {
                printf ("%c", ptr->data);
                preorder(ptr->lchild);
                preorder(ptr->rchild);
        }
}
```

Inorder Traversal (LNR)

1. Traverse the left subtree of the root in inorder
2. Visit the root node
3. Traverse the right subtree of the root in inorder

The function for inorder traversal is given below:

```
void inorder (struct node *);                    /*function declaration*/

inorder (root);                                  /*function calling*/

void inorder(struct node *ptr)                   /*function definition*/
{
        if (ptr != NULL)
        {
                inorder(ptr->lchild);
                printf ("%c", ptr->data);
                inorder(ptr->rchild);
        }
}
```

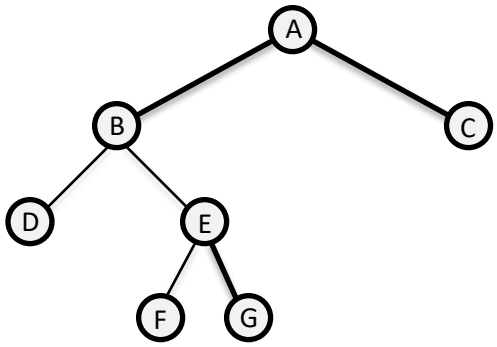
Postorder Traversal (LRN)

1. Traverse the left subtree of the root in postorder
2. Traverse the right subtree of the root in postorder
3. Visit the root node

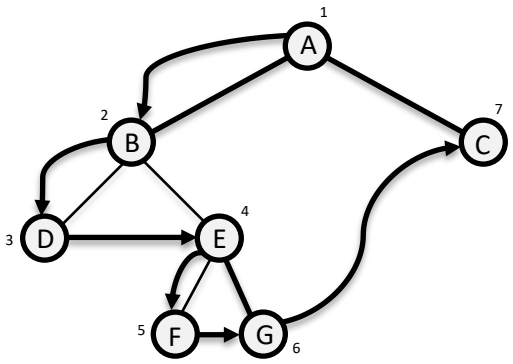
The function for postorder traversal is given below:

```
void postorder (struct node *);      /*function declaration*/

postorder (root);                    /*function calling*/

void postorder(struct node *ptr)     /*function definition*/
{
        if (ptr != NULL)
        {
                postorder(ptr->lchild);
                postorder(ptr->rchild);
                printf ("%c", ptr->data);
        }
}
```

Let us traverse in preorder, inorder and postorder in the given binary tree:

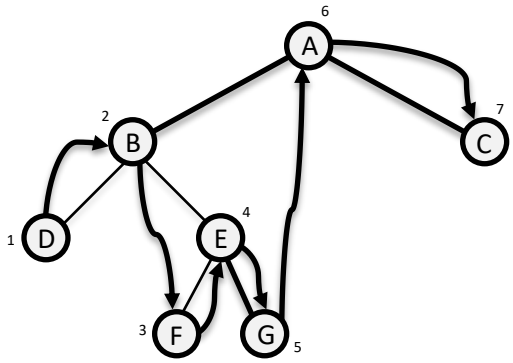


Preorder Traversal



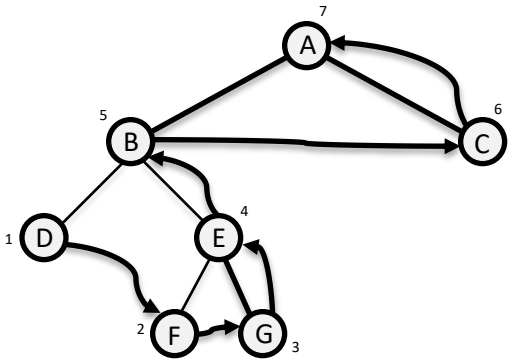
Here, parent node is visited first, then the left child is visited and then the right child is visited. The nodes are visit as ABDEFGC.

Inorder Traversal



Here, the left child is visited first, then the parent node is visited and then the left child is visited. The nodes are visited as DBFEGAC.

Postorder Traversal

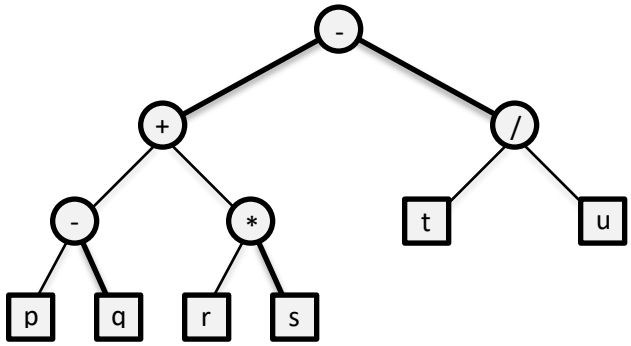


Here, the left child is visited first, then the right child is visited and then the parent node is visited.
The nodes are visit as DFGEBCA.

Let us take another example of an algebraic expression:

p – q + r * s – t / u

At first, construct an expression tree as follows:



The traversals of the above expression tree are as follows:

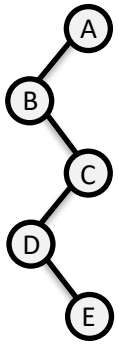
Preorder: - + - p q * r s / t u (Prefix or Polish Notation)

Inorder: p – q + r * s – t / u (Infix Notation)

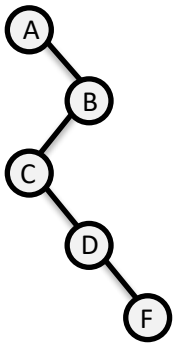
Postorder: p q – r s * + t u / - (Postfix or Reverse Polish Notation)

Question:

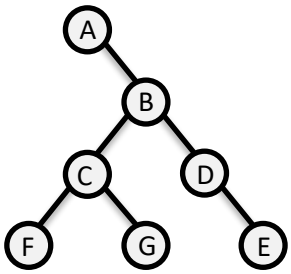
Determine the order in which the vertices of the binary trees will be visited under.
(i) Pre-order (ii) In-order (iii) Post-order



a)



b)



c)