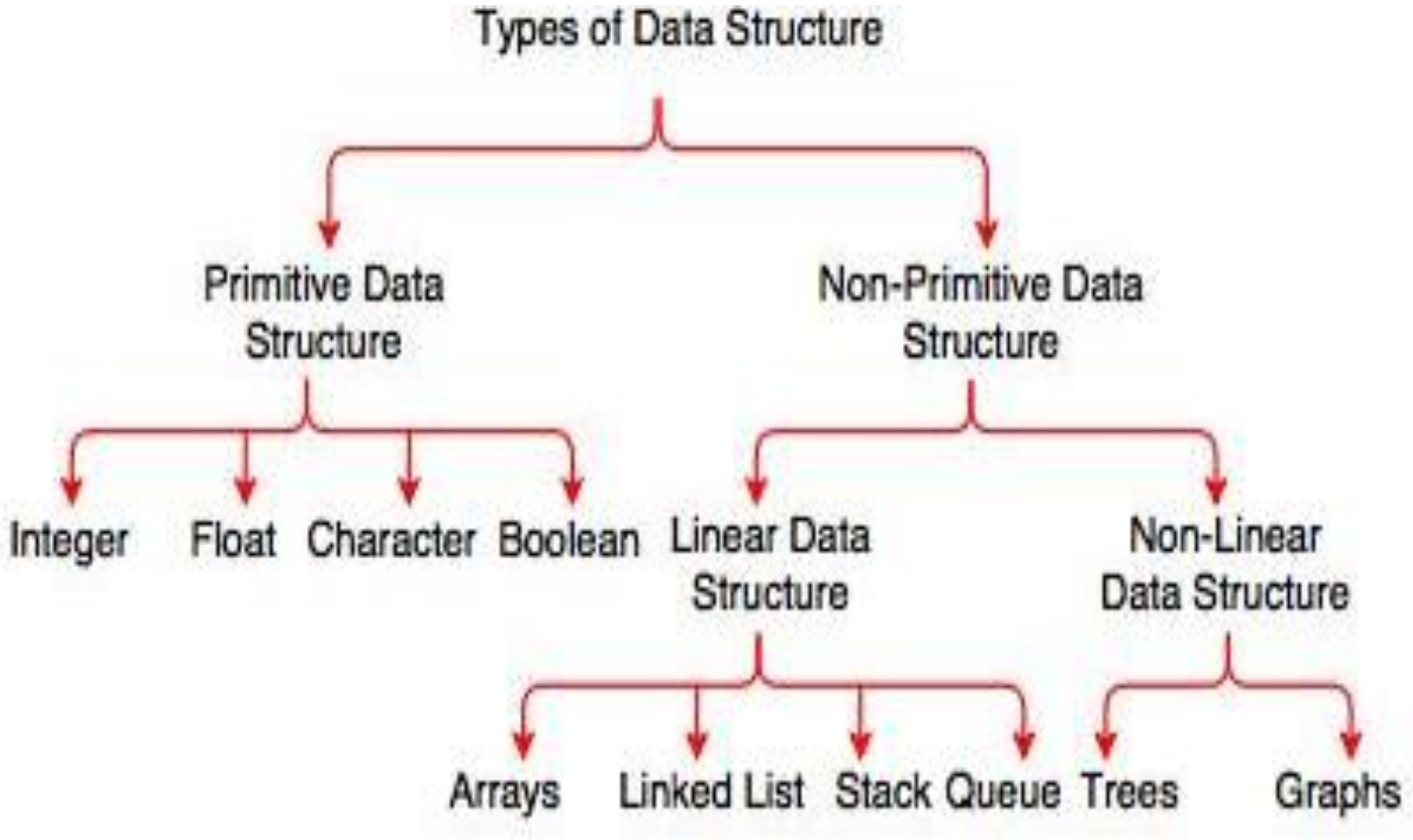
DATA STRUCTURES

UNIT-3

Basics of Tree Terminology and Binary Trees

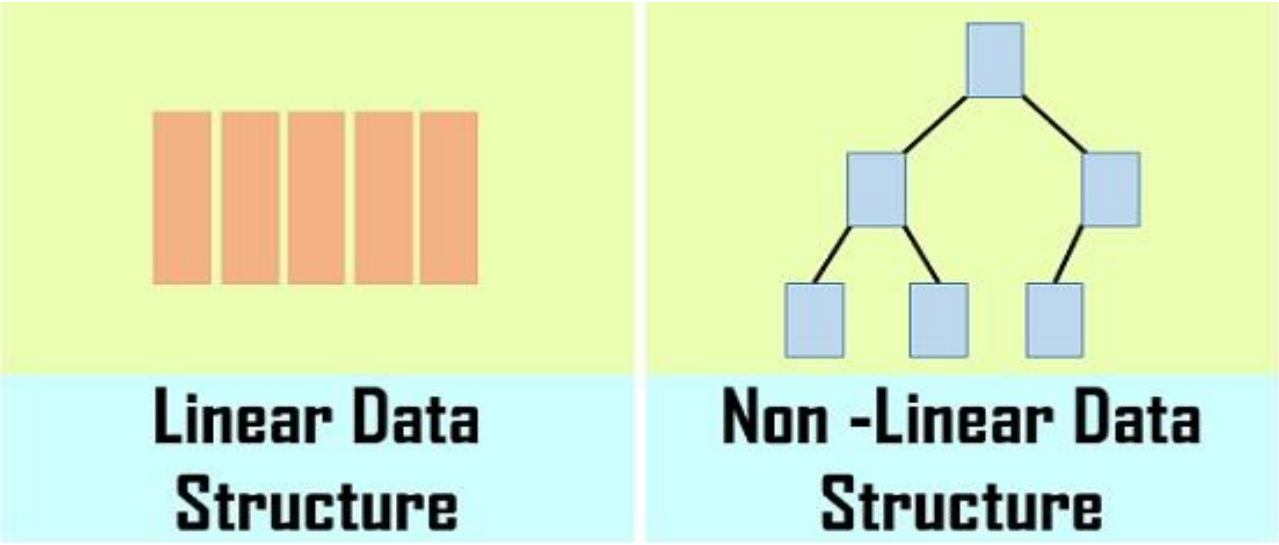
Classification of Data Structures



Linear Vs Non-Linear Data Structures

- In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.

- In a non-linear data structure, data elements are attached in hierarchically manner.



Linear Vs Non-Linear Data Structures

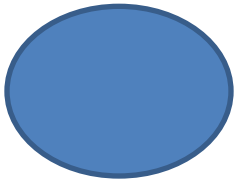
Sr. No.	Key	Linear Data Structures	Non-linear Data Structures
1	Data Element Arrangement	In linear data structure, data elements are sequentially connected and each element is traversable through a single run.	In non-linear data structure, data elements are hierarchically connected and are present at various levels.
2	Levels	In linear data structure, all data elements are present at a single level.	In non-linear data structure, data elements are present at multiple levels.
3	Implementation complexity	Linear data structures are easier to implement.	Non-linear data structures are difficult to understand and implement as compared to linear data structures.
4	Traversal	Linear data structures can be traversed completely in a single run.	Non-linear data structures are not easy to traverse and needs multiple runs to be traversed completely.
5	Memory utilization	Linear data structures are not very memory friendly and are not utilizing memory efficiently.	Non-linear data structures uses memory very efficiently.
6	Time Complexity	Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remain with increase in size.
7	Examples	Array, List, Queue, Stack.	Graph, Map, Tree.

Tree Data Structure

Tree Data Structure

- **Definition:** A Tree is a finite set of one or more nodes such that
  - There is a specially designated node called the root
  - The remaining nodes are partitioned into n>=0 disjoint setsT1,T2,…Tn, where each of these sets is a tree. T1,T2,…Tn are called the subtrees of the root.

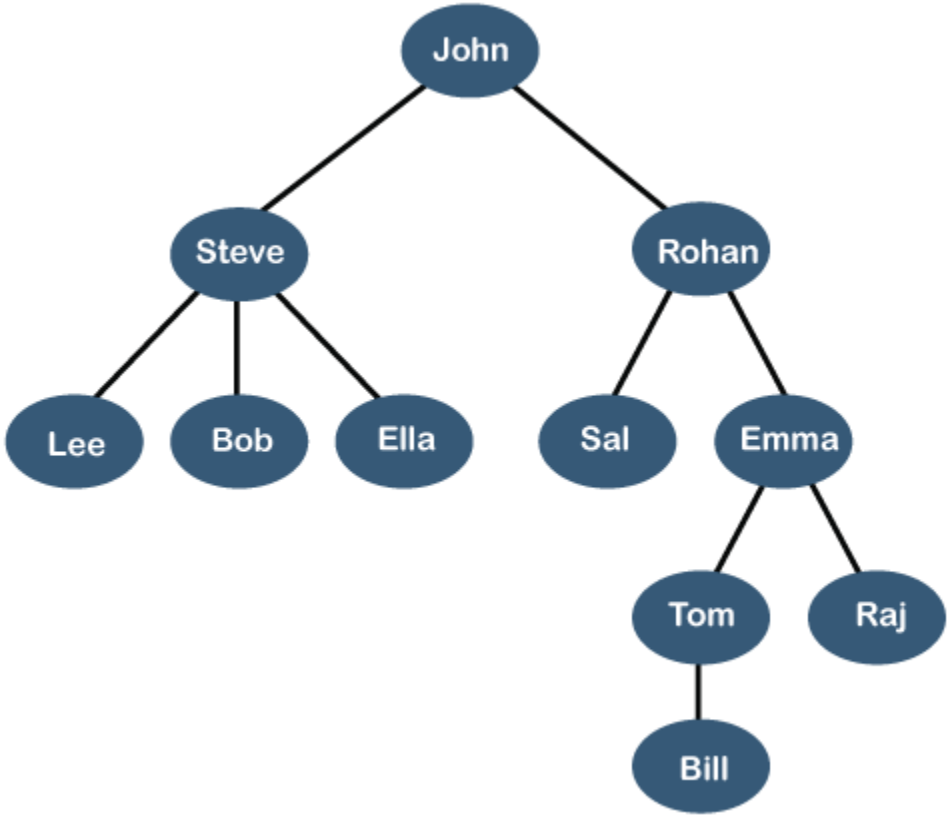
- Non linear data structures can be represented Using nodes and edges



What is a Tree

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:

Tree



Tree Terminology

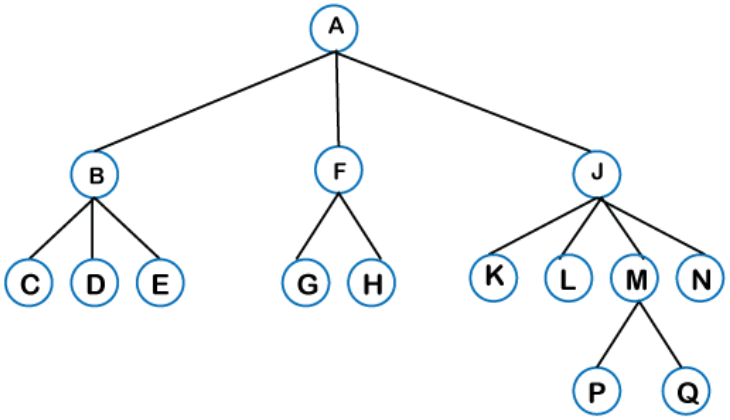
- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent.
- If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an internal
- **Ancestor node:** An ancestor of a node is any predecessor node on a path from the root to that node.
- **Recursive data structure:** The tree is also known as a recursive data structure.

Tree Terminology

- **Degree of a node:** Number of subtrees of node.
- **Degree of a tree:** Maximum of the degree of the nodes in the tree
- **Level of node:** root is said to be level 1(sometimes level 0) . If a node is at level ‘x’ then its children are at level ’x+1’.
- **Height or depth of tree:** Maximum level of any node in the tree.

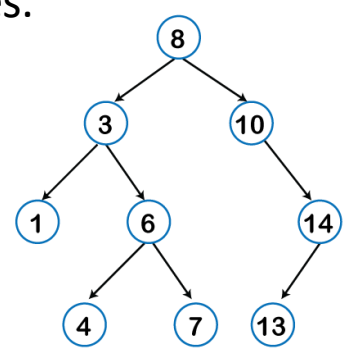
Types of Trees

General tree: The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain).



Binary tree: Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.

Tree



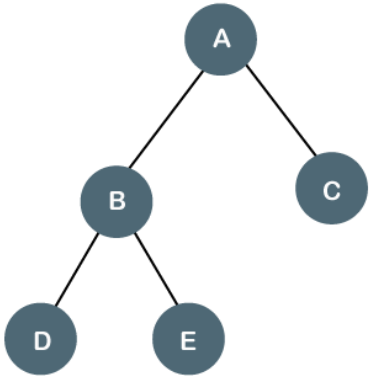
Types of Binary Trees

- There are four types of Binary tree:

    - Full/ proper/ strict Binary tree
    - Complete Binary tree
    - Perfect Binary tree
    - Degenerate Binary tree

1. Full/ proper/ strict Binary tree

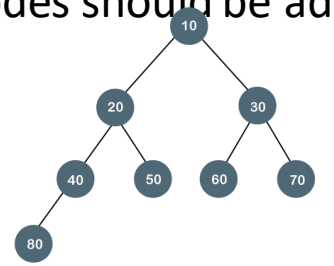
The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children.



Types of Binary Trees

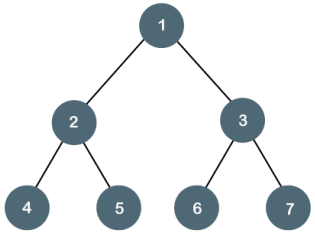
- **Complete Binary Tree**

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.



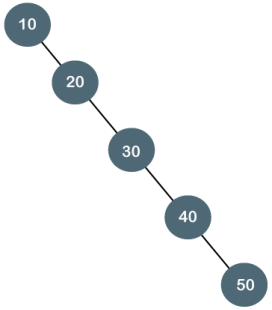
- **Perfect Binary Tree**

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



- **Degenerate Binary Tree**

The degenerate binary tree is a tree in which all the internal nodes have only one children.



How to store a Tree in Memory

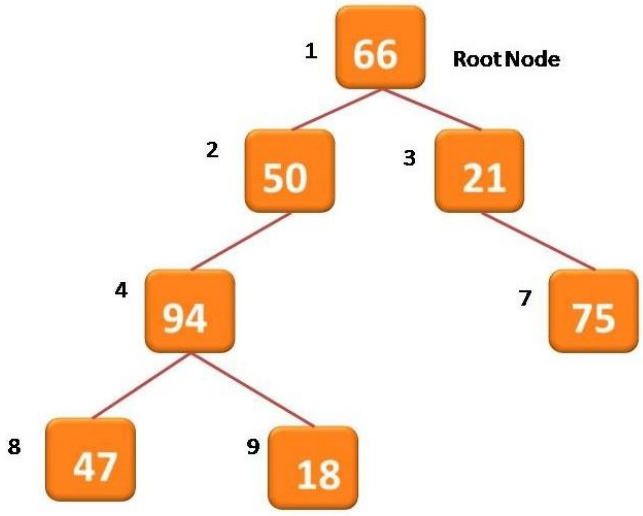
- Using Arrays
- Using Linked List

Memory Representation

- Array

A small and almost complete binary tree can be easily stored in a linear array. Small tree is preferably stored in linear array because searching process in a linear array is expensive. Complete means that if most of the nodes have two child nodes.

To store binary tree in a linear array, you need to consider the positional indexes of the nodes. This indexing must be considered starting with 1 from the root node going from left to right as you go down from one level to other.

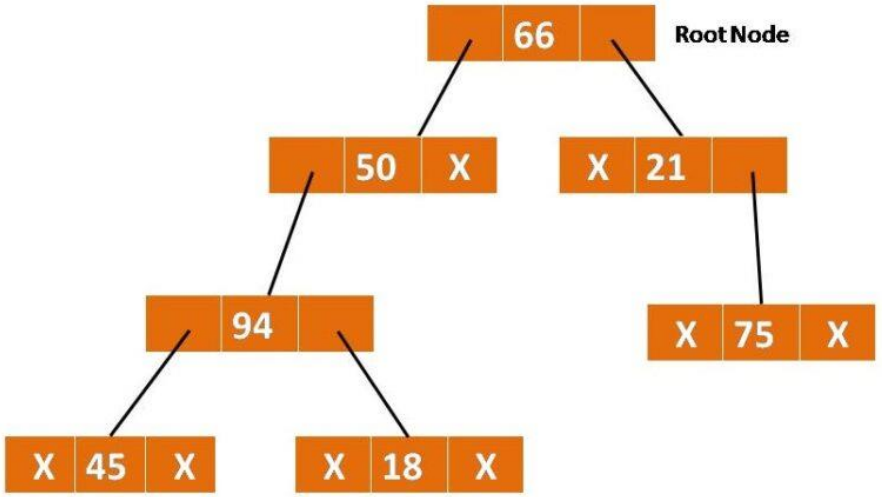


1	2	3	4	5	6	7	8	9
66	50	21	94			75	47	18

Memory Representation

- Linked Representation

In linked and dynamic representation, the linked list data structure is used. Each node constitutes of a data part and two link parts. The two link parts store address of left and right child nodes. Data part is used to store the information about the binary tree element. This is a better representation as nodes can be added or deleted at any location. Memory utilization is better in this binary tree representation.



Structure of a node for binary tree

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

Create a Binary Tree

```
struct node *create()
{
  struct node *temp;
temp = (struct node *)malloc(sizeof(struct node));
  WRITE Press 0 to exit;
  WRITE Press 1 for new node;
  WRITE Enter your choice ;
  READ choice;
  if(choice==0)
   {  return 0;  }
else
{ WRITE Enter the data:;    Read data;
  temp->data = data;
Write Enter the left child for data;
  temp->left = create();
Write Enter the right child for data;
  temp->right = create();
return temp;
} }
```

```
main()
  {
    struct node *root;
    root = create();
  }
```

Tree Traversals

- Three Tree Traversals
  - Preorder
  - Postorder
  - Inorder
  - Level Order

Preorder Traversal

Preorder Traversal

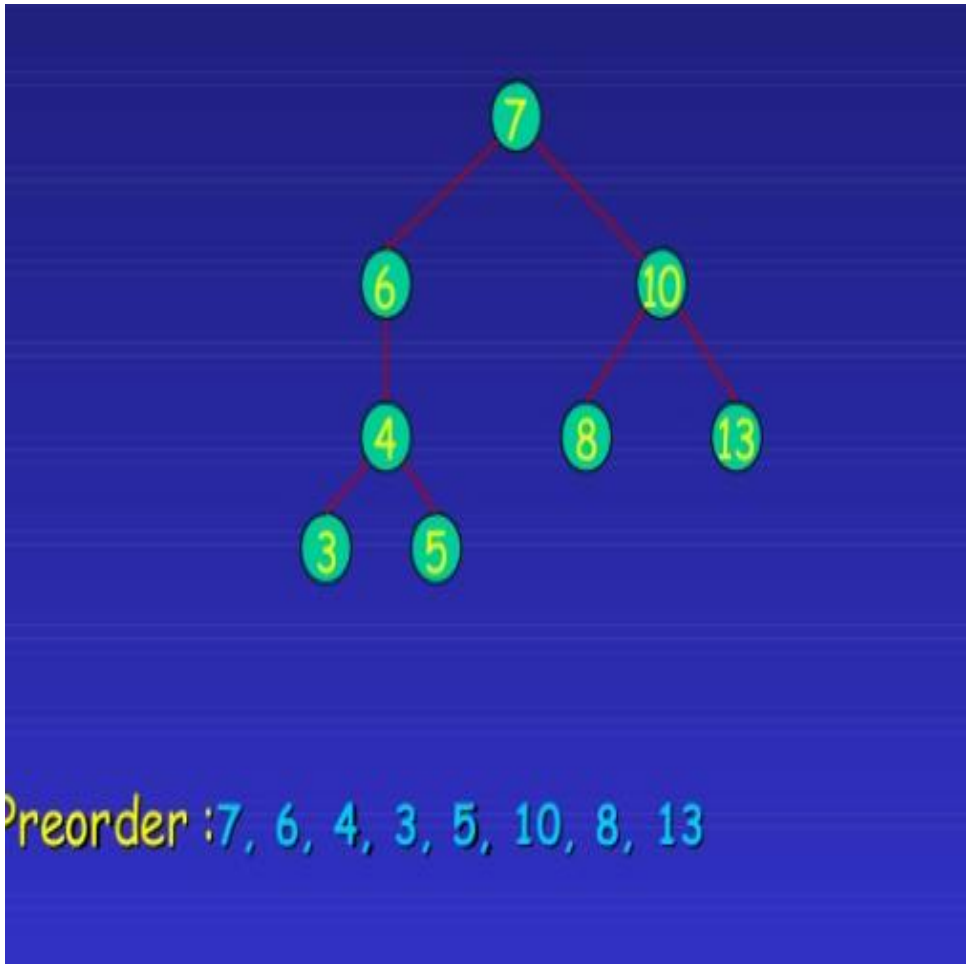
- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants

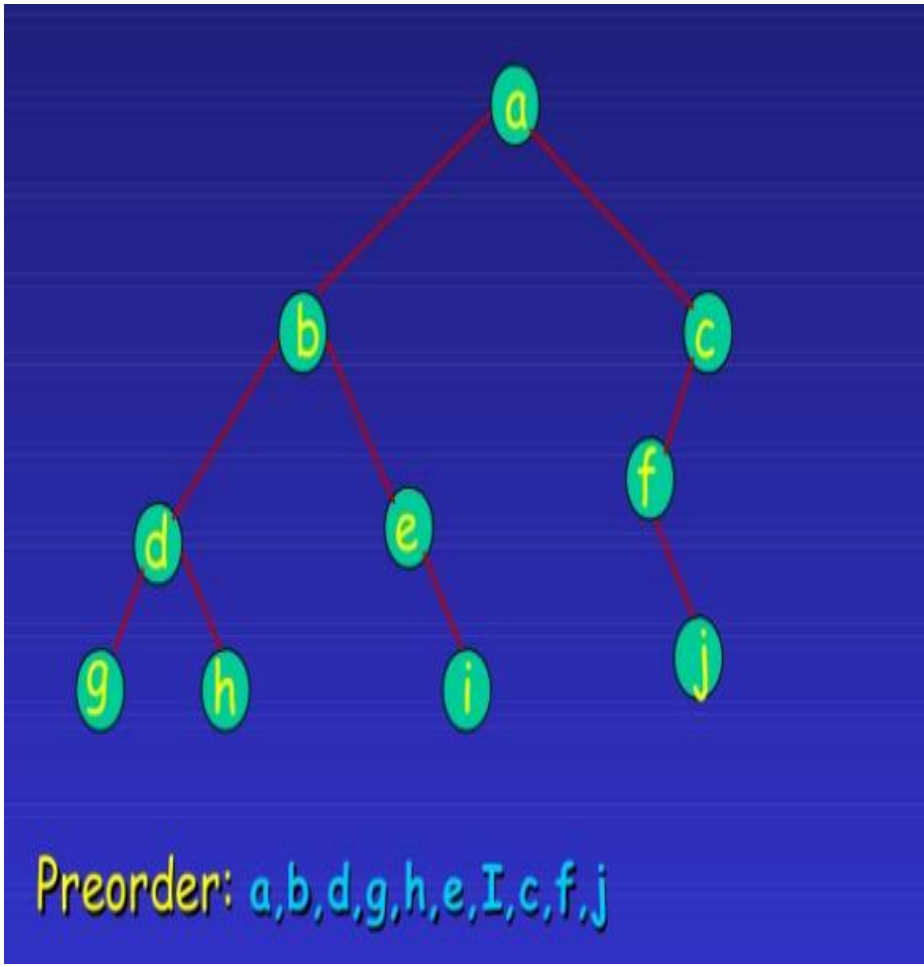
1. Visit the root
2. Traverse the left subtree in preorder
3. Traverse the right subtree in preorder

Preorder traversal

- node, left, right

Examples for Preorder Traversal





Postorder Traversal

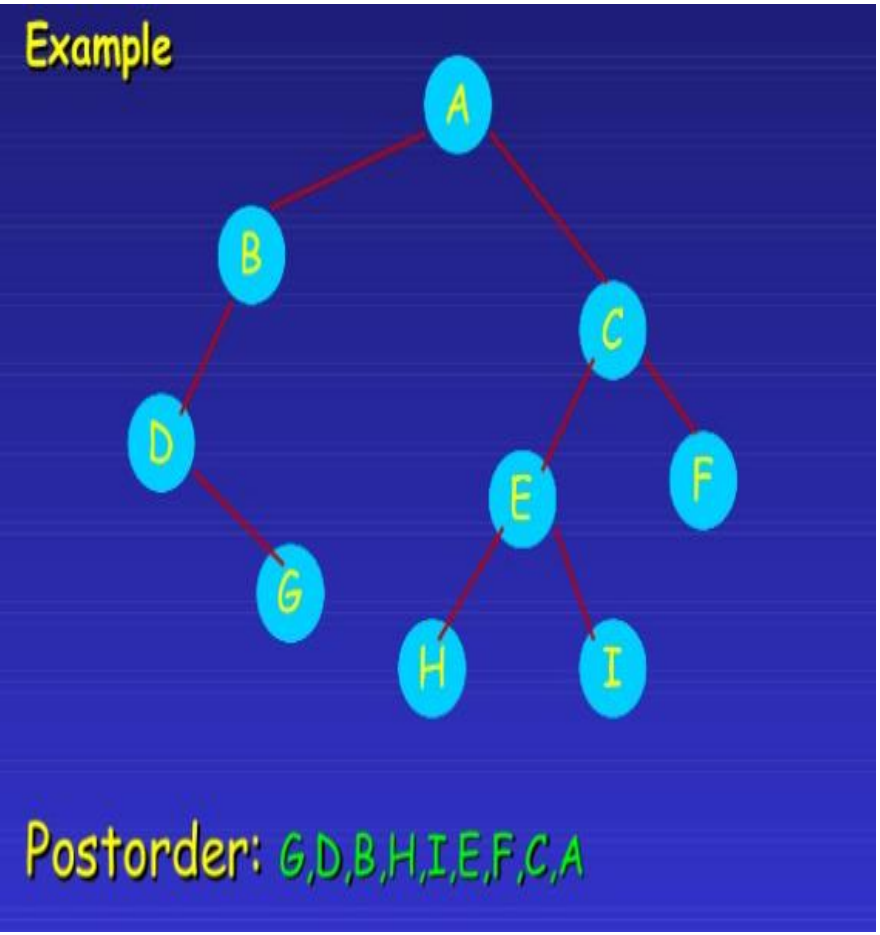
- In a postorder traversal, a node is visited after its descendants

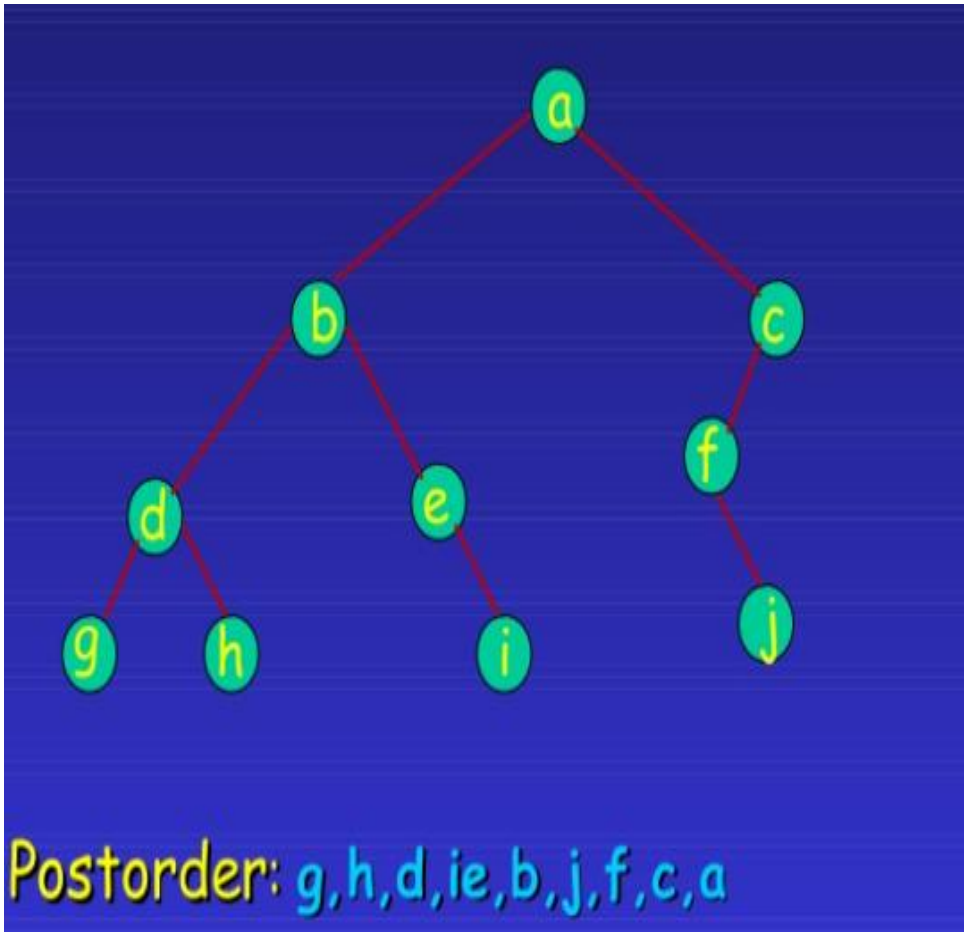
1. Traverse the left subtree in postorder
2. Traverse the right subtree in postorder
3. Visit the root

Postorder traversal

- left, right, node

Examples for Postorder Traversal





Inorder Traversal

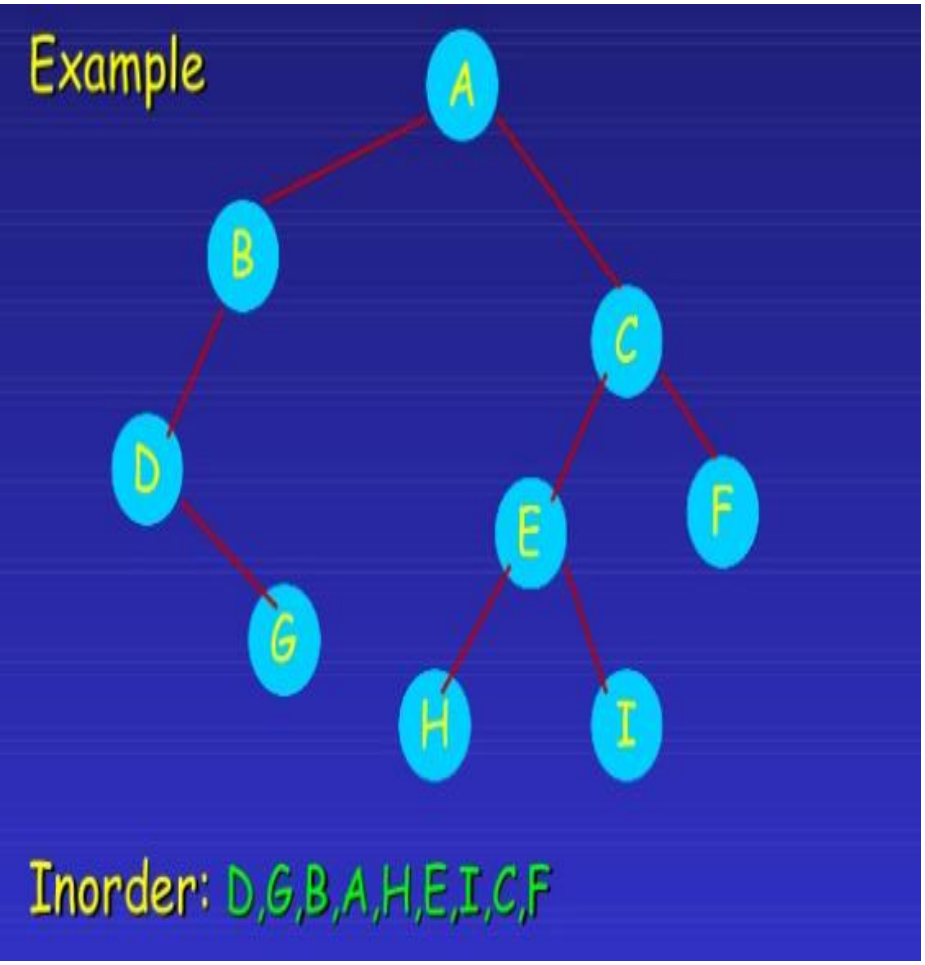
Inorder traversal

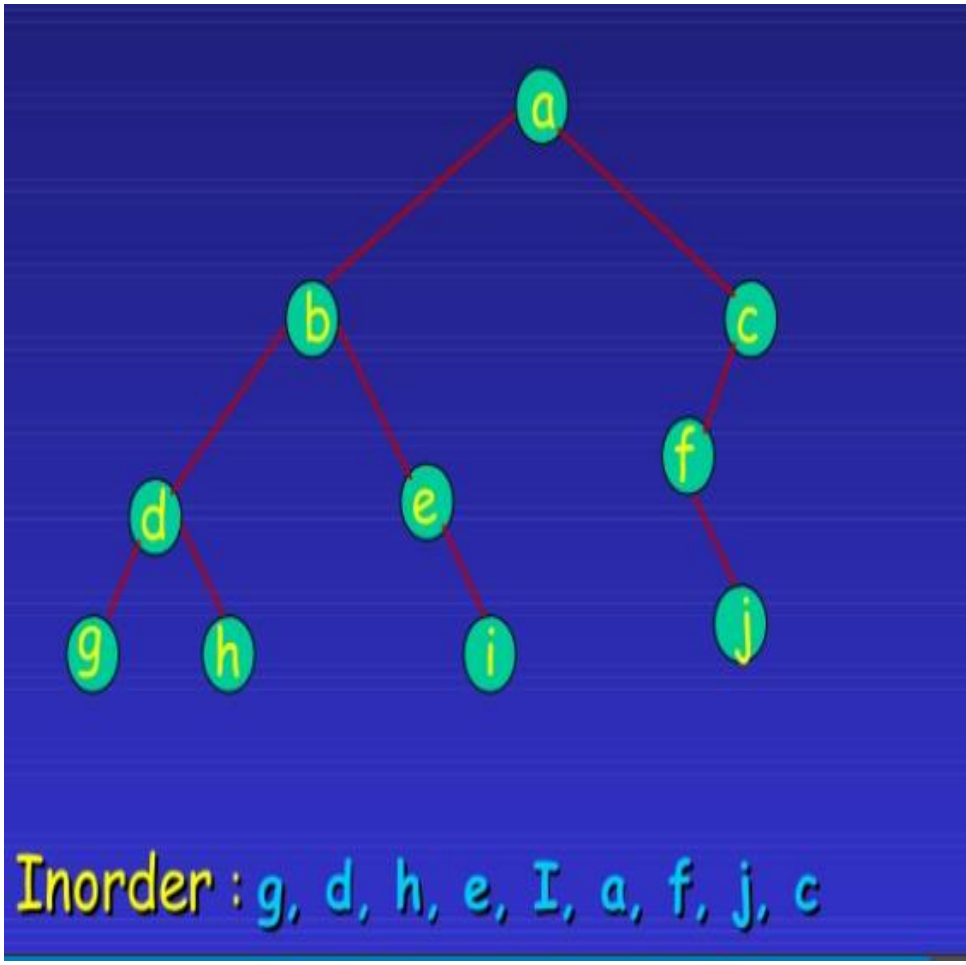
1. Traverse the left subtree in inorder
2. Visit the root
3. Traverse the right subtree in inorder

Inorder traversal

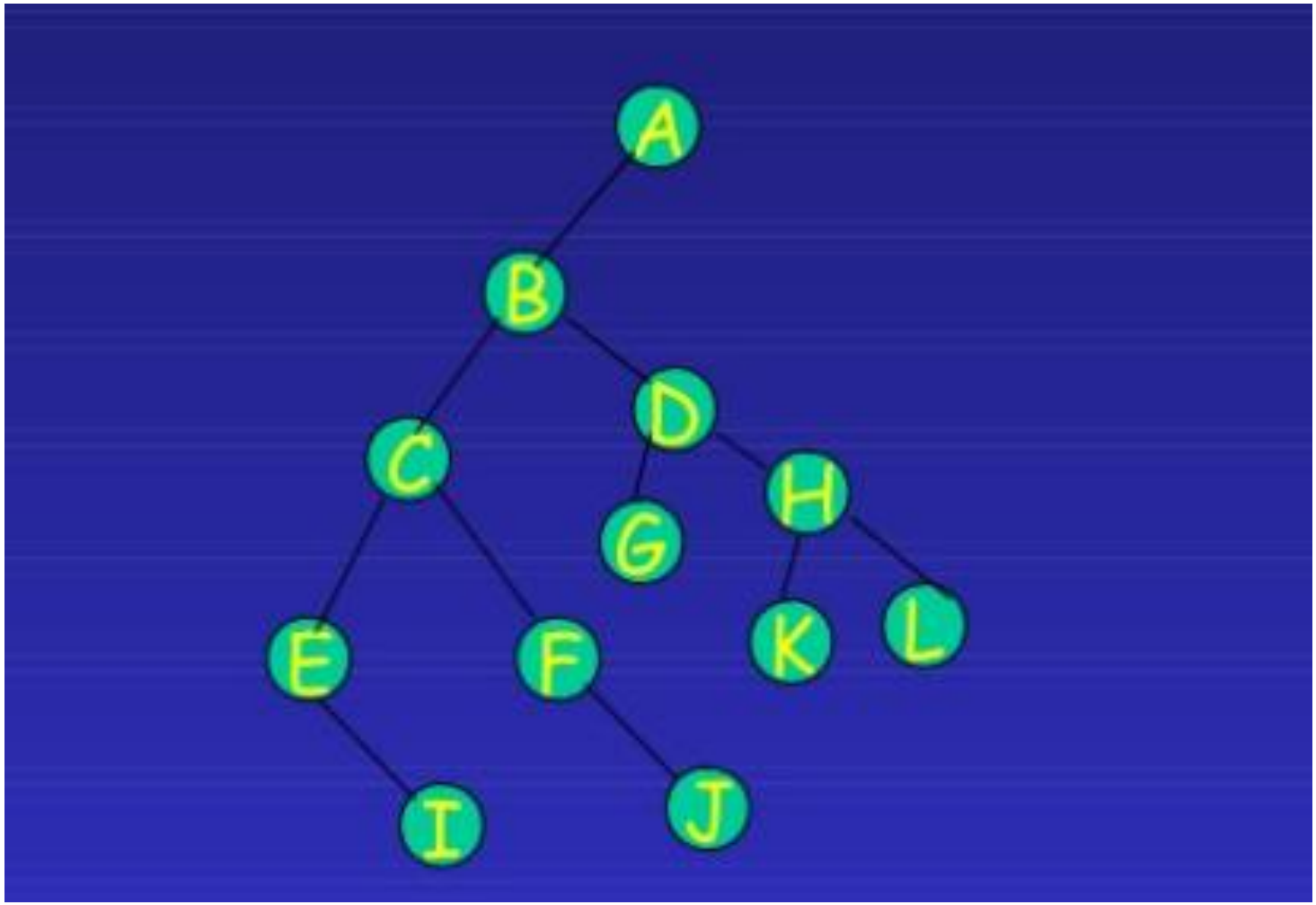
- left, node, right

Examples for Inorder Traversal





Tree Traversals: Example for work



Algorithm for Inorder Traversal

```
Algorithm inorder(Node *root)
{
    if (root == NULL)
        return;
    inorder(root->left);
    write root->data;
    inorder(root->right);
}
```

Algorithm for Preorder Traversal

```
Algorithm preorder(Node *root)
{
   if (root == NULL)
      return;

   write root->data;
   preorder(root->left);
   preorder(root->right);
}
```

Algorithm for Postorder Traversal

```
Algorithm postorder(Node *root)
{
    if (root == NULL)
        return;

    postorder(root->left);
    postorder(root->right);
    write root->data;
}
```

Exercise on Binary Trees

- Given the following tree traversals Construct a Binary Tree

  - Inorder: E I C F J B G D K H L A
  - Preorder: A B C E I F J D G H K L

  - Inorder: E M F B D L C A J G N H K O I
  - Preorder: A B E F M C D L G J N H K O I

Additional Stuff

- Implementation of Non-Recurcive tree traversals

Inorder Tree Traversal without Recursion

Step 1: Create an empty stack S.

Step 2: Initialize current as root node data value

Step 3: If current is NULL go to step 4.

Else a) Push the current node to S and

b) set current = current->left

c) go to step 3.

Step 4: If current is NULL and stack is not empty then

- a) Pop the top item from stack.
- b) Print the popped item, set current = popped_item->right
- c) Go to step 3.

Step 5: If current is NULL and stack is empty then we are done.

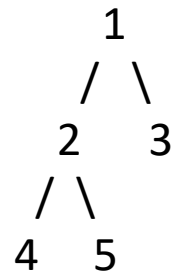
Example for Inorder Tree Traversal without Recursion

Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left
until current is NULL
current -> 1
push 1: Stack S -> 1
current -> 2
push 2: Stack S -> 2, 1
current -> 4
push 4: Stack S -> 4, 2, 1
current = NULL

Let us consider the below tree for example



Step 4 pops from S
a) Pop 4: Stack S -> 2, 1
b) print "4"
c) current = NULL /*right of 4 */ and go to step 3
Since current is NULL step 3 doesn't do anything.

Example for Inorder Tree Traversal without Recursion

Step 4 pops again.

a) Pop 2: Stack S -> 1

b) print "2"

c) current -> 5/*right of 2 */ and go to step 3

Step 3 pushes 5 to stack and makes current NULL

Stack S -> 5, 1

current = NULL

Step 4 pops from S

a) Pop 5: Stack S -> 1

b) print "5"

c) current = NULL /*right of 5 */ and go to step 3

Since current is NULL step 3 doesn't do anything

Step 4 pops again.

a) Pop 1: Stack S -> NULL

b) print "1"

c) current -> 3 /*right of 1 */

Step 3 pushes 3 to stack and makes current NULL

Stack S -> 3

current = NULL

Step 4 pops from S

a) Pop 3: Stack S -> NULL

b) print "3"

c) current = NULL /*right of 3 */

Iterative Inorder

```
Algorithm iterative-Inorder (mynode *root)
 {
   while(root != NULL)
  {
     while (root != NULL)
     {
         if (root->right != NULL)
         {
           save[top++] = root->right;
         }
         save[top++] = root;
         root = root->left;
     }

     root = save[--top];
     while(top != 0 && root->right == NULL)
     {
        printf("[%d] ", root->value);
        root = save[--top];
     }

     printf("[%d] ", root->value);
     root = (top != 0) ? save[--top] : (mynode *) NULL;
  }
 }
```

Iterative Preorder

```
void iterativePreorder(mynode *root)
 {
   mynode *save[100];
   int top = 0;

   if (root == NULL)
   {
     return;
   }

   save[top++] = root;
   while (top != 0)
   {
     root = save[--top];

     printf("[%d] ", root->value);

     if (root->right != NULL)
       save[top++] = root->right;
     if (root->left != NULL)
       save[top++] = root->left;
   }
 }
```

Iterative Postorder

```
void iterativePostorder(mynode *root)
{
  struct
  {
    mynode *node;
    unsigned vleft :1;   // Visited left?
    unsigned vright :1;  // Visited right?
  }save[100];

  int top = 0;

  save[top++].node = root;

  while ( top != 0 )
  {
     /* Move to the left subtree if present and not visited */
     if(root->left != NULL && !save[top].vleft)
     {
        save[top].vleft = 1;
        save[top++].node = root;
        root = root->left;
        continue;
     }
     /* Move to the right subtree if present and not visited */
     if(root->right != NULL && !save[top].vright )
     {
        save[top].vright = 1;
        save[top++].node = root;
        root = root->right;
        continue;
     }

     printf("[%d] ", root->value);

     /* Clean up the stack */
     save[top].vleft = 0;
     save[top].vright = 0;

     /* Move up */
     root = save[--top].node;
   }
}
```