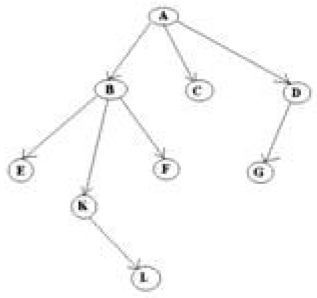
Unit 6
Trees and Graphs

Trees

A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a *nonlinear* data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more subtrees. A tree has following general properties:

- One node is distinguished as a root;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node; A direction is: *parent* -> *children*



A is a parent of B, C, D,
B is called a child of A.
on the other hand, B is a parent of E, F, K

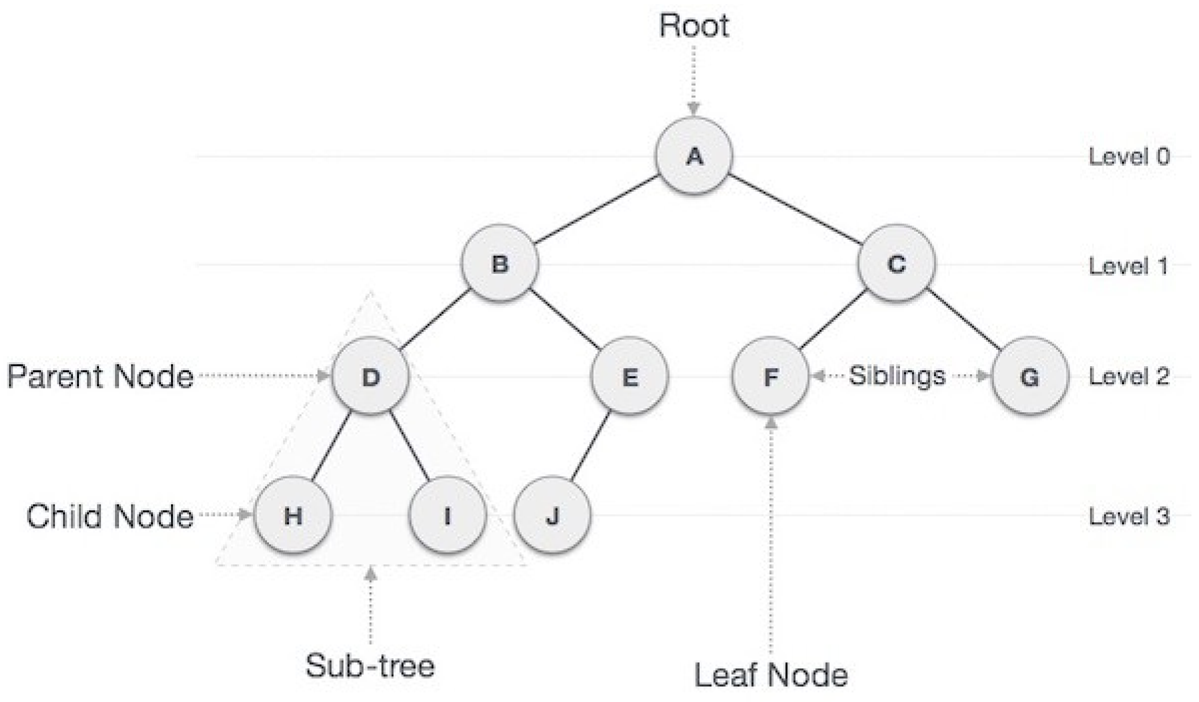
In the above picture, the root has 3 subtrees.

Definitions and Tree Terminologies

Following are the important terms with respect to tree.

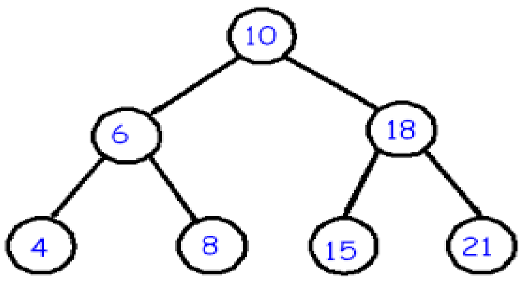
- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.

- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.



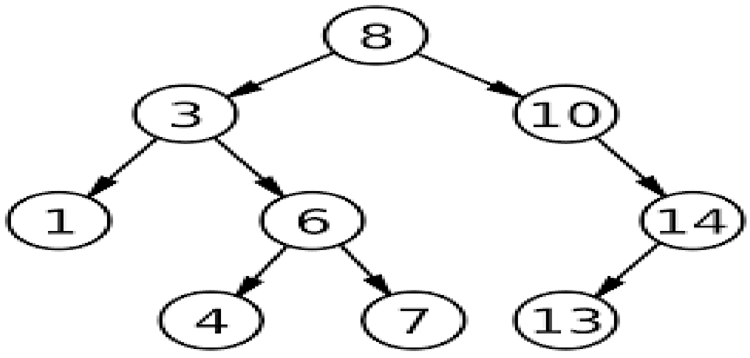
Types of Trees

1.Binary Trees:



A **binary tree** is a **tree data structure** in which each node has at most two children, which are referred to as the left child and the right child.

2.Binary Search Tree:

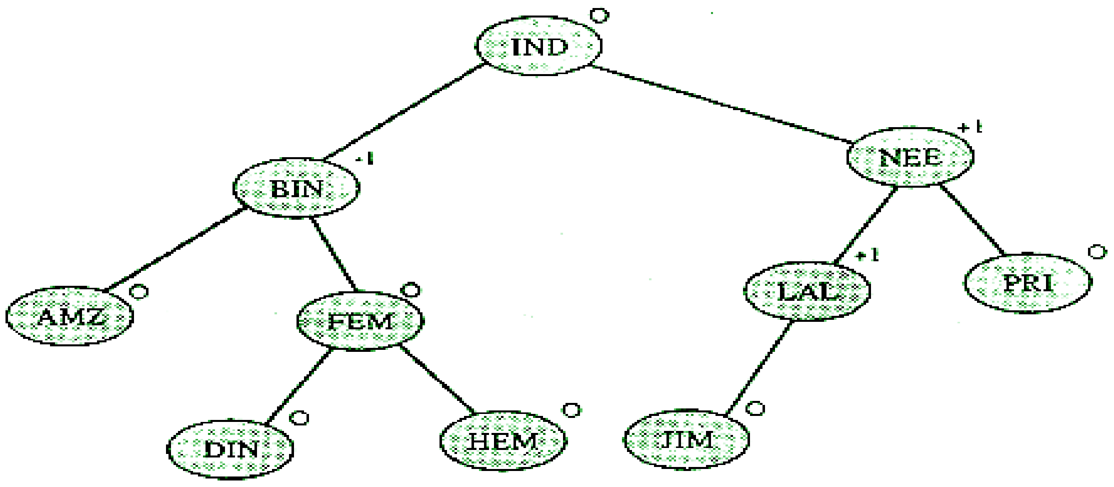


Binary Search Tree follow (Left.Value<Root<Rightchild.value) Rule

*The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient; they are also easy to code.

*Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

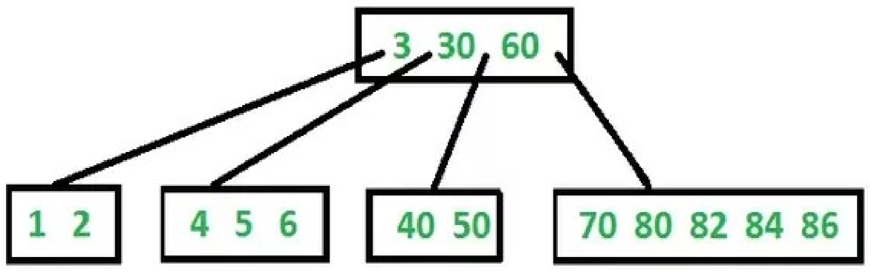
3.AVL TREE:



A **AVL tree** is a self-balancing binary search **tree**. In an **AVL tree**, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.

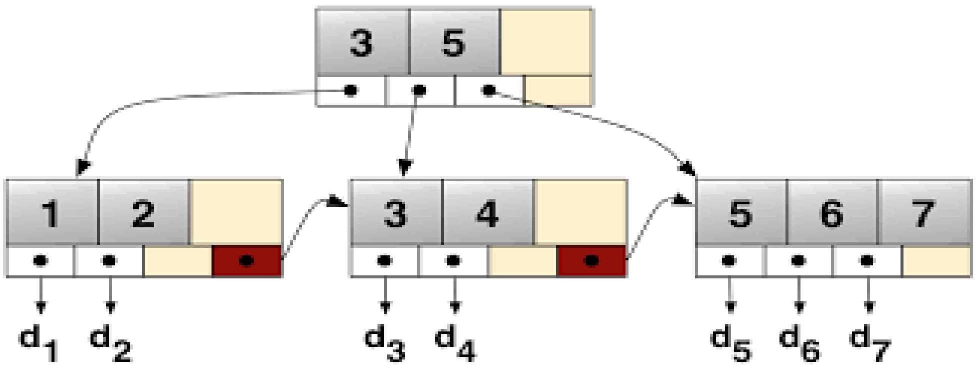
Operations Like Insertion and deletion have low complexity.

4.B-Tree:



A **B-tree** is a self-balancing **tree data structure** that keeps **data** sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The **B-tree** is a generalization of a binary search **tree** in that a node can have more than two children.

5.B+ Tree:



A **B+ tree** is an n-array tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

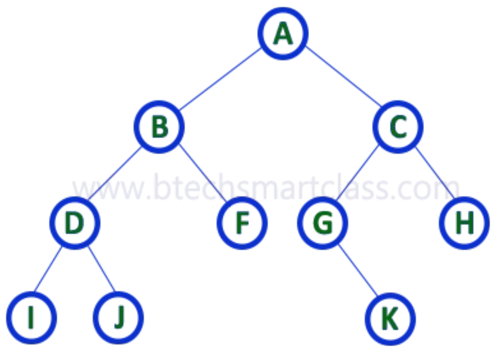
Binary Tree Representation

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



There are different types of binary trees and they are...

Full Binary Tree A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree.

```
               18
           /       \
         15         30
        /  \        /  \
      40    50    100   40


             18
           /    \
         15      20
        /  \
      40    50
     /   \
   30    50



               18
            /     \
          40        30
                   /  \
                100    40
```

In a Full Binary, number of leaf nodes is number of internal nodes plus 1

L = I + 1

Where L = Number of leaf nodes, I = Number of internal nodes

See Handshaking Lemma and Tree for proof.

Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

Following are examples of Complete Binary Trees

```
               18
           /       \
         15         30
        /  \       /  \
      40    50   100   40


               18
           /       \
         15         30
        /  \       /  \
      40    50   100   40
     /  \   /
    8    7 9
```

Practical example of Complete Binary Tree is Binary Heap.

Perfect Binary Tree A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.
Following are examples of Perfect Binaryr Trees.

```
               18
           /       \
         15         30
        /  \       /  \
      40    50   100   40


               18
           /       \
         15         30
```

A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has 2^h – 1 node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree

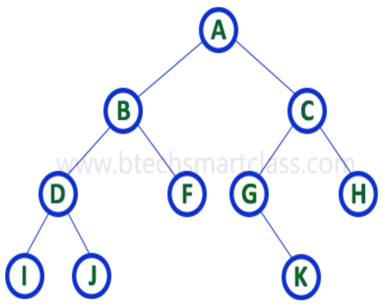
A binary tree is balanced if height of the tree is O(Log n) where n is number of nodes. For Example, AVL tree maintain O(Log n) height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert and delete.

Binary Tree Representation

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation
2. Linked List Representation

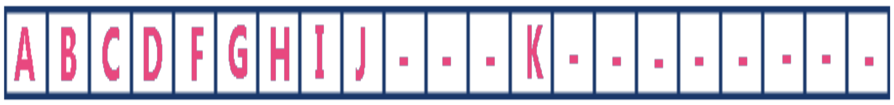
Consider the following binary tree...



1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

Consider the above example of binary tree and it is represented as follows...

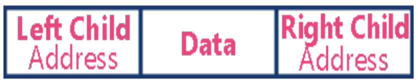


A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

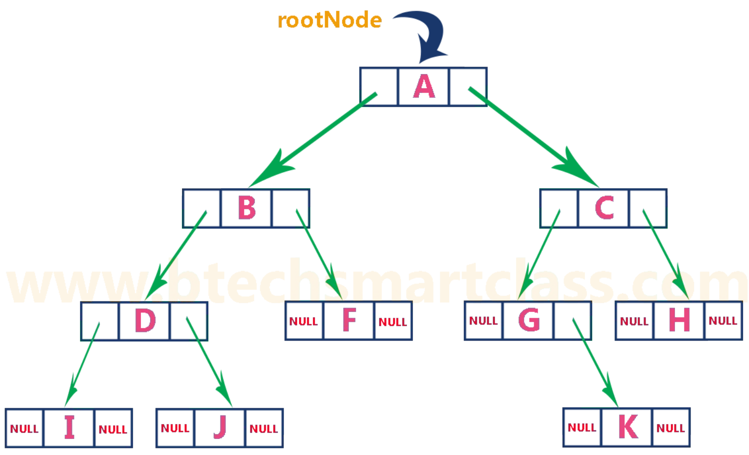
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.

2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
In this linked list representation, a node has the following structure...



The above example of binary tree represented using Linked list representation is shown as follows...



Binary Tree Operations

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.
Binary Tree Representation in C: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.
A Tree node contains following parts.
1. Data
2. Pointer to left child
3. Pointer to right child

In C, we can represent a tree node using structures. Below is an example of a tree node with an integer data.

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

First Simple Tree in C

Let us create a simple tree with 4 nodes in C. The created tree would be as following.

```
      tree
      ----
       1    <-- root
     /   \
    2     3
   /
  4
```

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

/* newNode() allocates a new node with the given data and NULL
left and
   right pointers. */
struct node* newNode(int data)
{
  // Allocate memory for new node
  struct node* node = (struct node*)malloc(sizeof(struct node));

  // Assign data to this node
  node->data = data;

  // Initialize left and right children as NULL
  node->left = NULL;
  node->right = NULL;
  return(node);
}


int main()
{
  /*create root*/
  struct node *root = newNode(1);
  /* following is the tree after above statement

        1
      /   \
     NULL  NULL
  */


  root->left        = newNode(2);
  root->right       = newNode(3);
  /* 2 and 3 become left and right children of 1
           1
         /   \
        2      3
     /    \    /  \
    NULL NULL NULL NULL
  */


  root->left->left  = newNode(4);
  /* 4 becomes left child of 2
           1
       /       \
      2          3
    /   \       /  \
   4    NULL  NULL  NULL
  /  \
NULL NULL
*/

  getchar();
  return 0;
}
```

Binary Tree Traversal

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.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal

3. Post - Order Traversal

1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

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

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

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

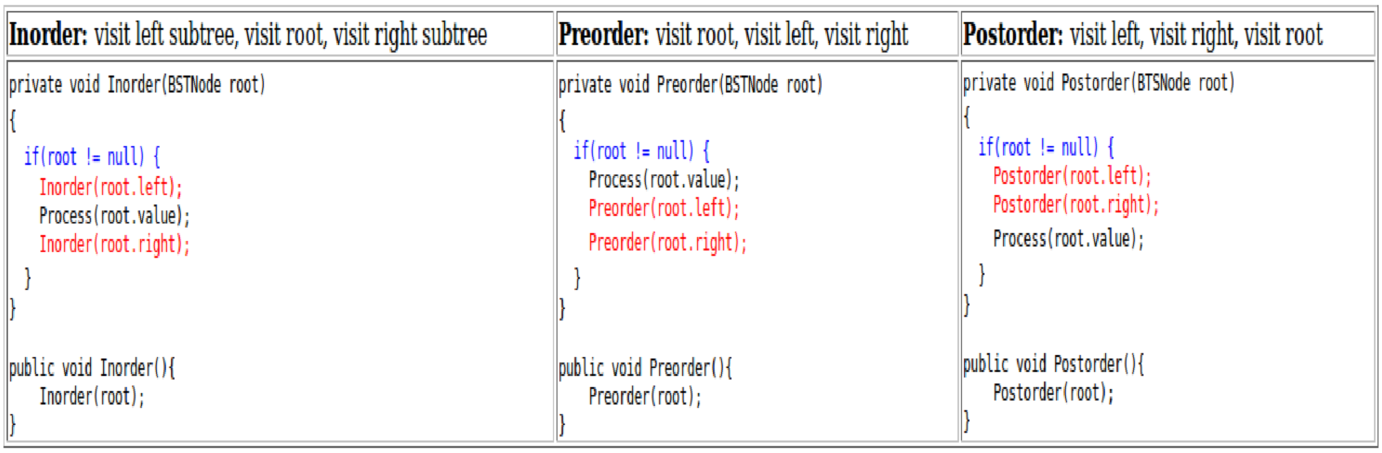
2. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

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



Inorder: visit left subtree, visit root, visit right subtree	**Preorder:** visit root, visit left, visit right	**Postorder:** visit left, visit right, visit root
`private void Inorder(BSTNode root) { if(root != null) { Inorder(root.left); Process(root.value); Inorder(root.right); } }` `public void Inorder(){ Inorder(root); }`	`private void Preorder(BSTNode root) { if(root != null) { Process(root.value); Preorder(root.left); Preorder(root.right); } }` `public void Preorder(){ Preorder(root); }`	`private void Postorder(BTSNode root) { if(root != null) { Postorder(root.left); Postorder(root.right); Process(root.value); } }` `public void Postorder(){ Postorder(root); }`

<u>Binary Search Tree</u>

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

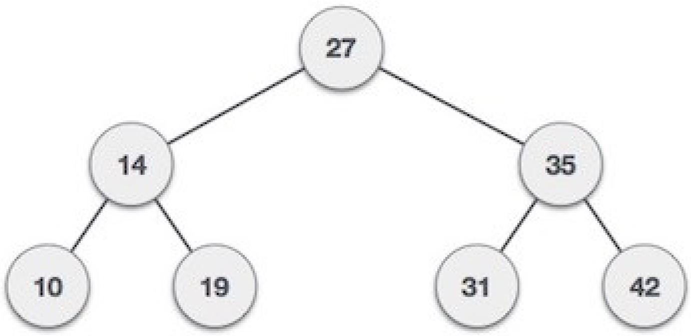
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

```
left_subtree (keys)  ≤  node (key)  ≤  right_subtree (keys)
```

Representation

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

Following is a pictorial representation of BST −



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

Basic Operations

Following are the basic operations of a tree −

- Search − Searches an element in a tree.
- Insert − Inserts an element in a tree.
- Pre-order Traversal − Traverses a tree in a pre-order manner.
- In-order Traversal − Traverses a tree in an in-order manner.
- Post-order Traversal − Traverses a tree in a post-order manner.

Node

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

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

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

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

   while(current->data != data){

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

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

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

Insert Operation

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

Algorithm

```
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;
            }
         }
      }
   }
}
```

Threaded Binary Tree

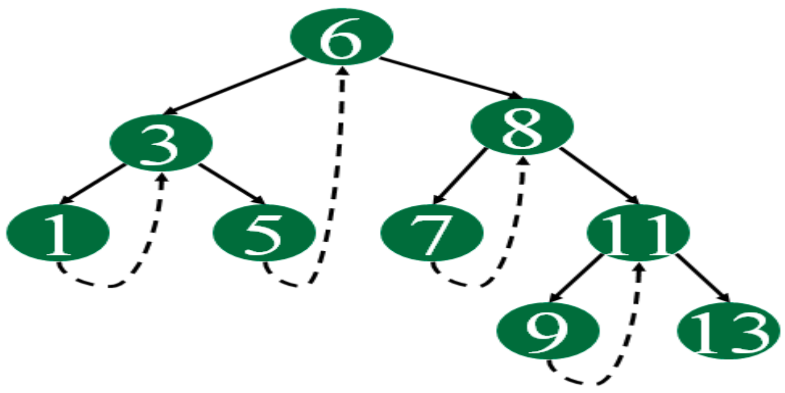
Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



C representation of a Threaded Node
Following is C representation of a single threaded node.

```
struct Node
{
    int data;
    Node *left, *right;
    bool rightThread;
}
```

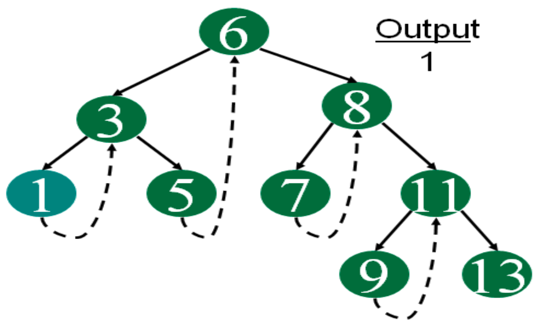
Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

Inorder Taversal using Threads

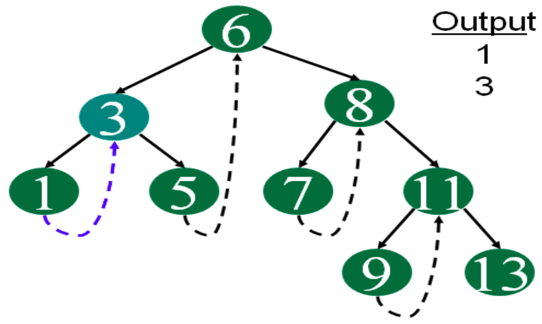
Following is C code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in a tree rooted with n
struct Node* leftMost(struct Node *n)
{
```

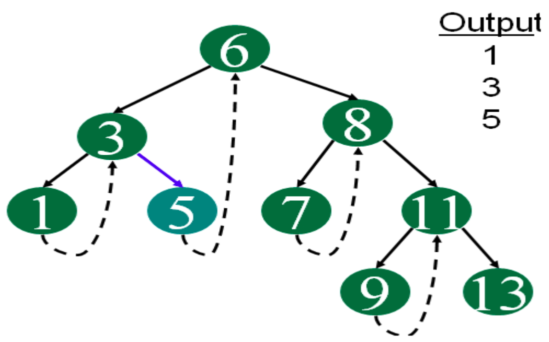
Following diagram demonstrates inorder order traversal using threads.



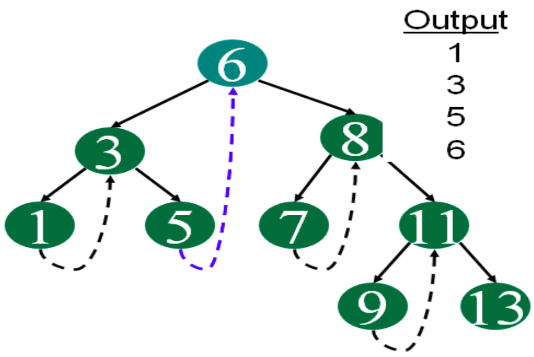
Start at leftmost node, print it



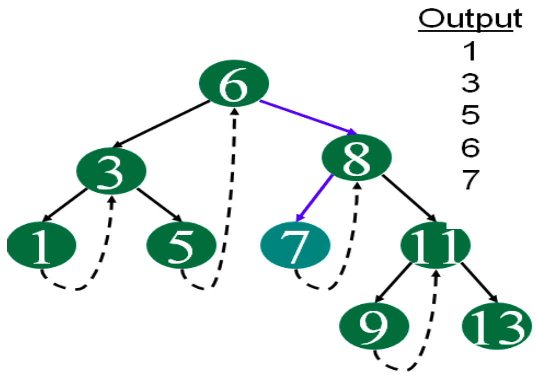
Follow thread to right, print node



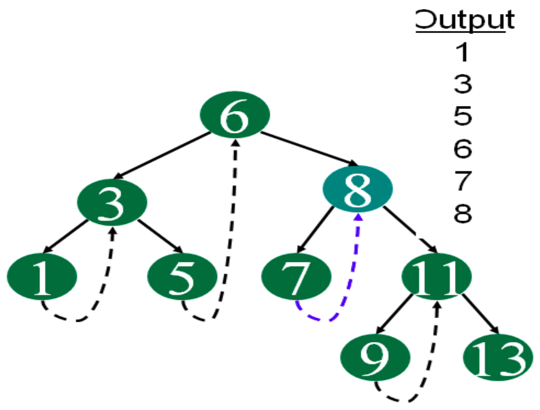
Follow link to right, go to leftmost node and print



Follow thread to right, print node



Follow link to right, go to leftmost node and print

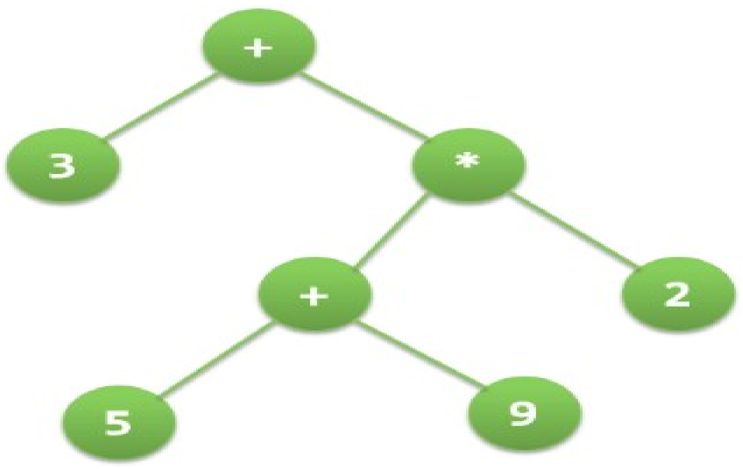


Follow thread to right, print node

continue same way for remaining node.....

Expression Tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree:

```
Let t be the expression tree
If  t is not null then
      If t.value is operand then
                Return  t.value
      A = solve(t.left)
      B = solve(t.right)

      // calculate applies operator 't.value'
      // on A and B, and returns value
      Return calculate(A, B, t.value)
```

Construction of Expression Tree:

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

1) If character is operand push that into stack

2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

//Implementation of Expression Tree

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct tree
{
 char data;
 struct tree *left,*right;
};
int top=-1;

struct tree *stack[20];
struct tree *node;

void push(struct tree *node)// to push operator and operands in tree
{
  ++top;
 stack[top]=node;
}

struct tree *pop()//to pop elements for evaluation
{
 return(stack[top--]);
}

//check() function checks if the element is operator or operand
int check(char ch)
{
 if(ch=='+' || ch=='-' || ch=='*' || ch=='/')
   return 2;//for operator return 2
 else
   return 1;//for operand return 1
}
int cal(struct tree *node)
{
 int ch;
 ch=check(node->data);//check if the node is operator or opreand

 if(ch==1)//if element is operand
   return node->data-48;
/*convert numerbers which we have stored as string into int...refer ASCII Table e.g. ASCII value of
1 is 49 so node->data-48 is 49-48=1, 50-48=2 where 50 is ASCII value of 2*/

 else if(ch==2)//if element is operator
 {
  if(node->data=='+')
    return cal(node->left)+cal(node->right);
  if(node->data=='-')
    return cal(node->left)-cal(node->right);
  if(node->data=='*')
    return cal(node->left)*cal(node->right);
  if(node->data=='/')
    return cal(node->left)/cal(node->right);
 }
}

//inorder traversal
```

```
void inorder(struct tree *node)
{
 if(node!=NULL)
 {
  inorder(node->left);
  printf("%c ",node->data);
  inorder(node->right);
 }
}
```

//if the element is operand then a node is crerated having left and right ptr set to null and it is pused on stack

```
void operand(char b)
{
 node=(struct tree*)malloc(sizeof(struct tree));
 node->data=b;
 node->left=NULL;
 node->right=NULL;
 push(node);
}
```

//if the element is operator then a node is created and we will pop two node elements i.e operands from stack and aasign it to that operator. then we will push the node on the tree.

```
void operater(char a)
{
 node=(struct tree*)malloc(sizeof(struct tree));
 node->data=a;
 node->right=pop();
 node->left=pop();
 push(node);
}

void main()
{
 int i,p,k,ans;
 char s[20];
 printf("\nEnter a postfix Expression : ");
 scanf("%s",s);
 for(i=0;s[i]!='\0';i++)//expression tree is created using this loop
 {
  p=check(s[i]);//check if the element is operator or operand

  if(p==1)
   operand(s[i]);
  else if(p==2)
   operater(s[i]);
  else
   exit(0);
 }
 ans=cal(stack[top]);
 printf("\nValue of the postfix Expression you entered is %d.",ans);
 printf("\nThe inorder traversal of the tree is:-\n");
 inorder(stack[top]);
 printf("\n");
}
```

Applications of tree data structure

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system
———————

```
     /   <-- root
  /      \
...        home
      /          \
   ugrad        course
    /          /    |    \
  ...        cs101 cs112 cs113
```

2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for insertion/deletion.

4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Following are the common uses of tree.
1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms