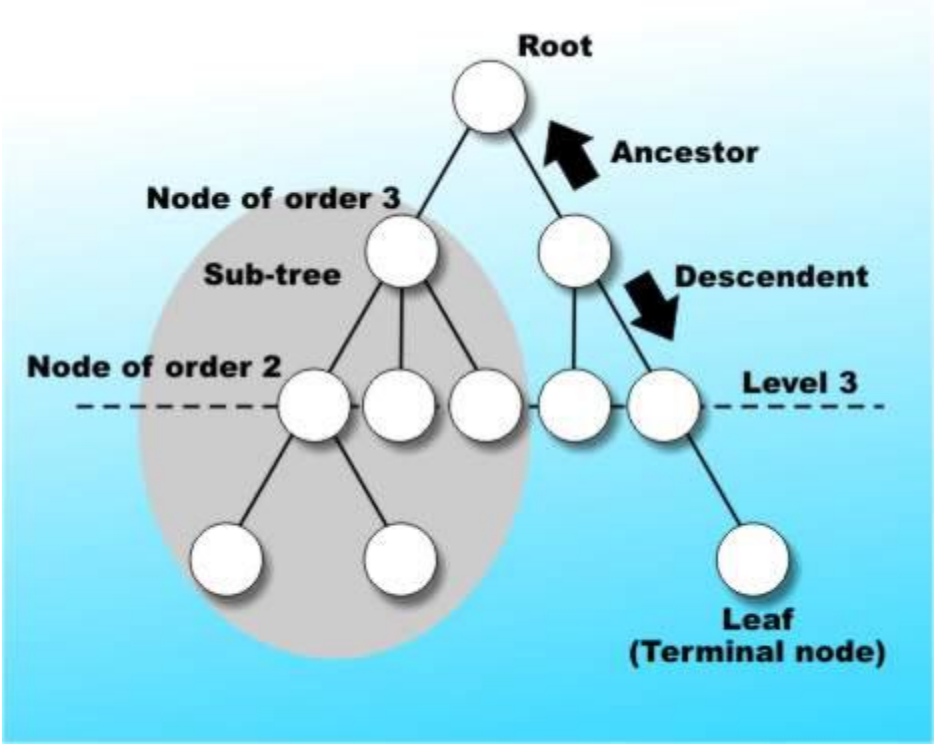
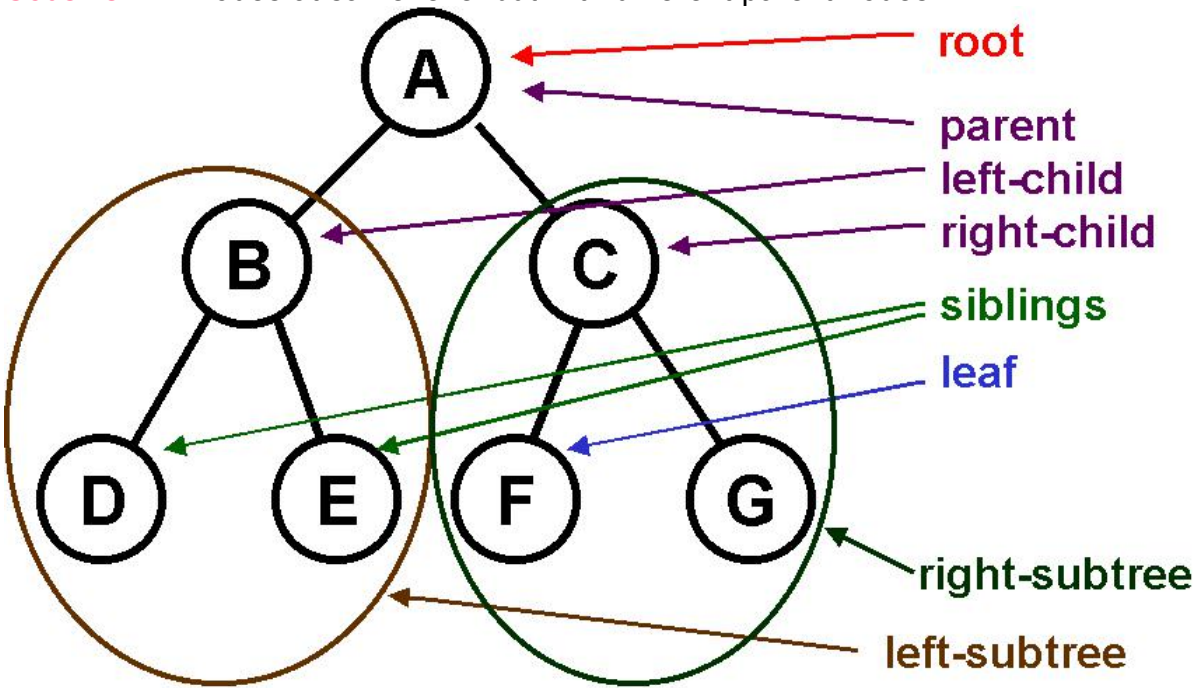
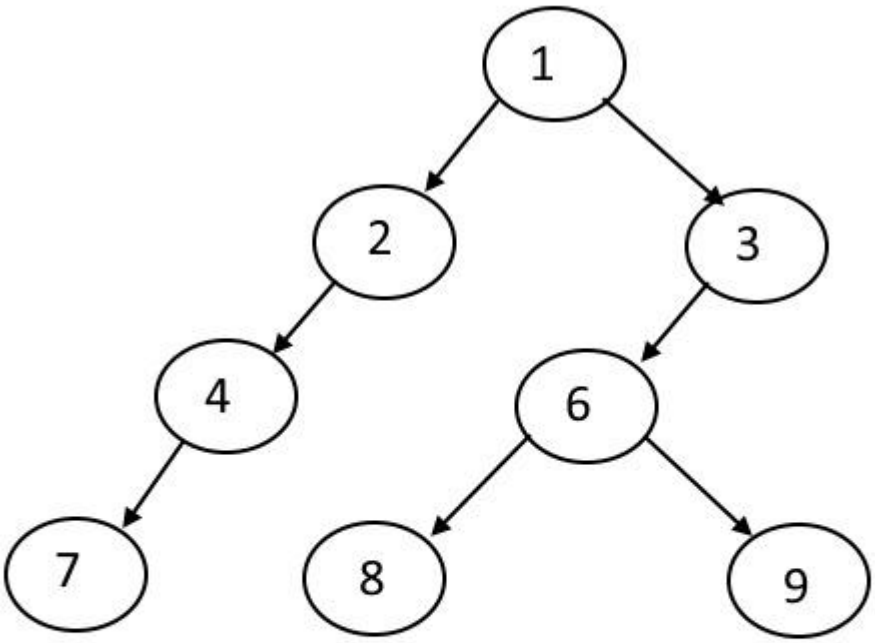
Important Terms



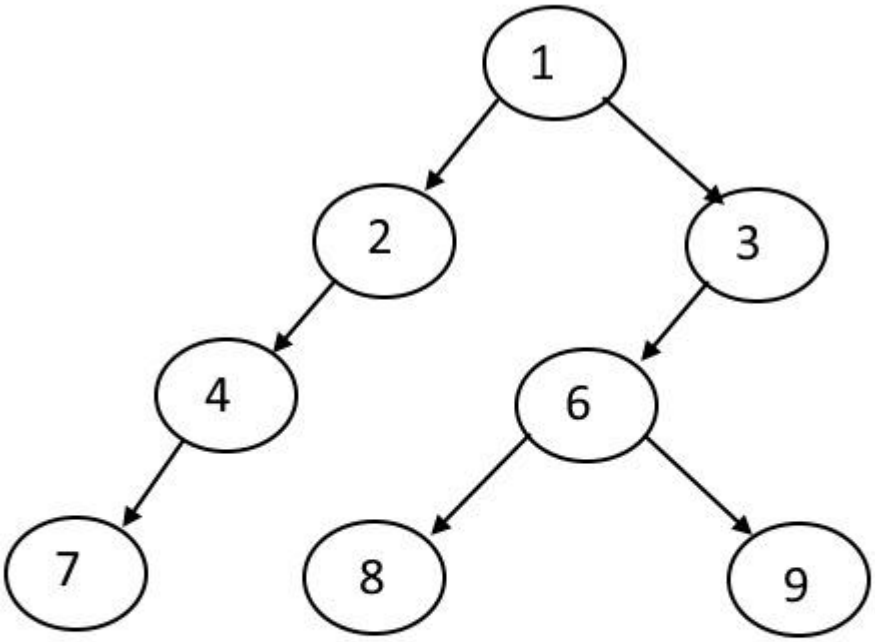
Path − Path refers to the sequence of nodes along the edges of a tree.
Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
Parent − Any node except the root node has one edge upward to a node called parent.
Child − The node 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.
Height - The height of a node is the number of edges on the longest path from the node to a leaf. A leaf node will have a height of 0.
Depth - The depth of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0.
Diameter - The diameter (or width) of a tree is the number of nodes on the longest path between any two leaf nodes.

Width of level - Number of nodes in that level.
Siblings - All nodes with same parent node.
Cousins - All nodes at same level but with different parent nodes.



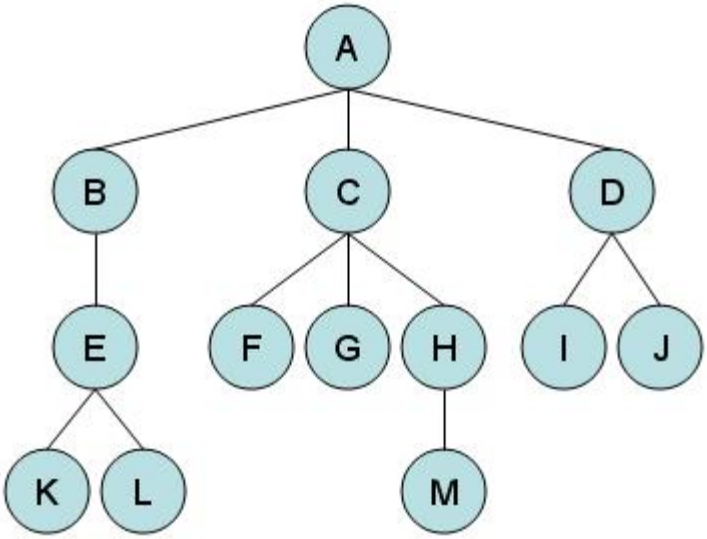


Node 1 and 2 are Siblings?? False
Node 2 and 3 are Siblings?? True
Node 4 and 3 are Siblings?? False
Node 4 and 6 are Siblings?? False
Node 8 and 9 are Siblings?? True



Node 1 and 2 are cousins?? False
Node 2 and 3 are cousins?? False
Node 4 and 3 are cousins?? False
Node 4 and 6 are cousins?? True
Node 7 and 9 are cousins?? True

Generic Tree




```
public class treeNode<T> {
    public T data;
    public ArrayList<treeNode<T>> children ;

    public treeNode(T data){
        this.data = data;
```

```
        children = new ArrayList<>();
    }

    private static Scanner s = new Scanner(System.in);

    public static treeNode<Integer> treeInput(){
        System.out.println("Enter root data : ");
        int data = s.nextInt();
        treeNode<Integer> root = new treeNode<>(data) , newNode = null;

        Queue<treeNode<Integer>> queue = new LinkedList<>();

        queue.add(root);
        while(!queue.isEmpty()){
            treeNode<Integer> temp = queue.poll();
            System.out.println("Enter no. of children of "+temp.data+": ");
            int child = s.nextInt();
            for(int i=0;i<child;i++){
                System.out.println("child "+i+" of "+"data :");
                int childData = s.nextInt();
                newNode = new treeNode<>(childData);
                temp.children.add(newNode);
                queue.add(newNode);
            }
        }
        return root;
    }

    public static void print(treeNode<Integer> root){
        System.out.println("Root data : " + root.data);

        Queue<treeNode<Integer>> queue = new LinkedList<>();
        queue.add(root);

        while(!queue.isEmpty()){
            treeNode<Integer> temp = queue.poll();
            String str = "";
            for(int i=0;i<temp.children.size();i++){
                str = str + temp.children.get(i).data+",";
                queue.add(temp.children.get(i));
            }
            System.out.println(str);
        }
    }

}

//User class
```

```
public class treeUse {

    public static int  largest(treeNode<Integer> root){
        if(root == null){
            return 0;
        }
        int max = root.data;
        for(int i =0;i<root.children.size();i++){
            int temp =largest(root.children.get(i));
            if(temp > max)
                max = temp;
        }
        return max;
    }

    public static void main(String[] args) throws Exception{
        treeNode<Integer> root = treeNode.treeInput();
        treeNode.print(root);
        System.out.println("Largest node data: "+largest(root));
    }

}
```

Sample Input:

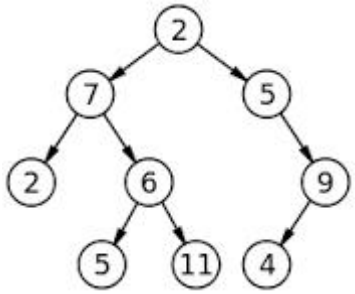
```
Enter root data :
1
Enter no. of children of 1:
3
child 0 of data :
2
child 1 of data :
3
child 2 of data :
4
Enter no. of children of 2:
2
child 0 of data :
5
child 1 of data :
6
Enter no. of children of 3:
0
Enter no. of children of 4:
0
Enter no. of children of 5:
0
Enter no. of children of 6:
0
```

Root 1 children : 2,3,4,
Root 2 children : 5,6,
Root 3 children :
Root 4 children :
Root 5 children :
Root 6 children :

Largest node data: 6

Binary Tree

A binary tree is one in which each node has at most two descendants(0,1 or 2 children) - a node can have just one but it can't have more than two. Clearly each node in a binary tree can have a left and/or a right descendant.




```
public class TreeNode<T> {
   T data;
   Treenode<T> left;
   Treenode<T> right;

   public TreeNode(T data){
      this.data = data;
      left = right = null;
   }
}
```

Full Binary Tree

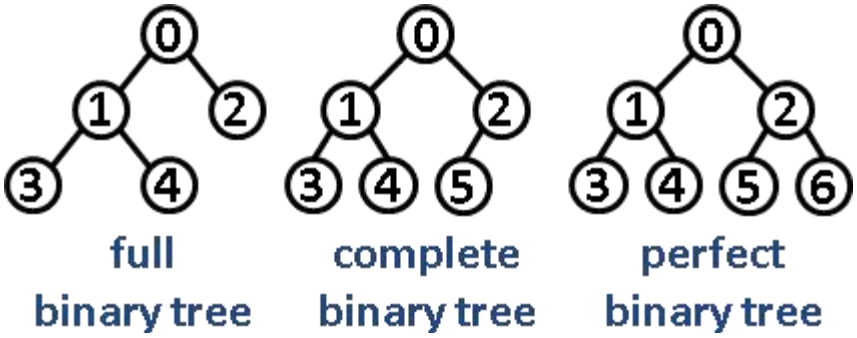
Every node should have exactly 2 nodes except the leaves. It is also called Strict Binary tree or Proper Binary tree.

Complete Binary Tree

It is a binary tree in which every level (except possibly the last) is completely filled, and all nodes are as far left as possible.

Perfect Binary Tree

A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level



Binary Tree Traversal

A Tree is typically traversed in two ways:

Breadth First Traversal OR Level Order Traversal

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key') and explores the neighbor nodes first, before moving to the next level neighbors. Queue is used for BFS.

```
//Level order traversal using Queue
public void printLevelOrder() {
  Queue<TreeNode> queue = new LinkedList<TreeNode>();
  queue.add(root);
  while (!queue.isEmpty()){

    TreeNode tempNode = queue.poll();
    System.out.print(tempNode.data + " ");

          /*Enqueue left child */
    if (tempNode.left != null) {
      queue.add(tempNode.left);
    }

          /*Enqueue right child */
    if (tempNode.right != null) {
      queue.add(tempNode.right);
    }
  }
}
```

Depth First Traversals

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. Stack is used for DFS.

Inorder Traversal (Left-Root-Right)

```
 public void inorder(TreeNode root) {
     if (root == null) {
         return;
     }
     inorder(root.left);
```

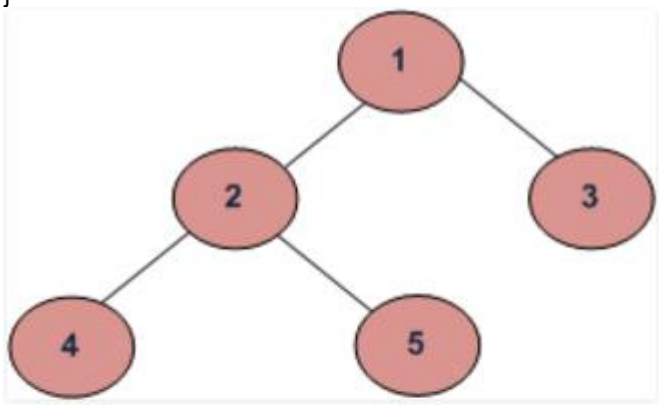
```
    System.out.print(root.element + " ");
    inorder(root.right);
}
```

Preorder Traversal (Root-Left-Right)

```
private void preorder(TreeNode root) {
    if (root == null) {
        return;
    }
    System.out.print(root.element + " ");
    preorder(root.left);
    preorder(root.right);
}
```

Postorder Traversal (Left-Right-Root)

```
private void postorder(TreeNode root) {
    if (root == null) {
        return;
    }
    postorder(root.left);
    postorder(root.right);
    System.out.print(root.element + " ");
}
```




```
BFS and DFSs of above Tree

Breadth First Traversal : 1 2 3 4 5

Depth First Traversals:
      Preorder Traversal : 1 2 4 5 3
      Inorder Traversal  :  4 2 5 1 3
      Postorder Traversal : 4 5 2 3 1
```

Height of Binary Tree

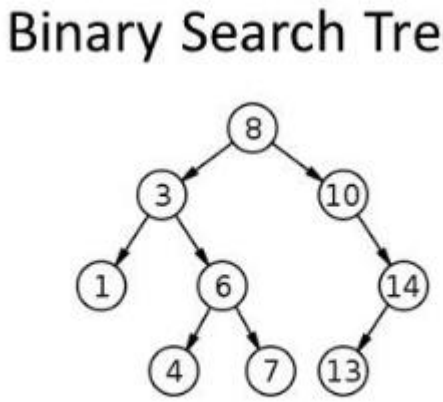
```
public static int height(TreeNode<Integer> root) {
  return root == null ? 0 : Math.max(height(root.left), height(root.right)) + 1;
}
```

Searching in Binary Tree

```
public static TreeNode<Integer> search(TreeNode<Integer> root, Integer data) {
  if (root == null) {
    return null;
  }
  if (root.data.equals(data)) {
    return root;
  }
  TreeNode<Integer> result = search(root.left, data);
  if (result == null) {
    result = search(root.right, data);
  }
  return result;
}
```

Binary Search tree

For a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be less than the data of the root. The data of all the nodes in the right subtree of the root node should be more than the data of the root.



	Average	Worst Case
Space	O(*n*)	O(*n*)
Search	O(log *n*)	O(*n*)
Insert	O(log *n*)	O(*n*)
Delete	O(log *n*)	O(*n*)

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.

```
public class BSTnode<T> {
```

```
private static Scanner s = new Scanner(System.in);

T data;
BSTnode<T> left ;
BSTnode<T> right ;
BSTnode(T data){
   this.data = data;
}
// Insert node by node in BST
public static BSTnode<Integer> insertNode(){
   BSTnode<Integer> root = null;
   System.out.println("Enter root data : ");
   int nodedata = s.nextInt();
   while(nodedata != -1){
      root = insertInBST(root,nodedata);
      System.out.println("Enter another node data : ");
      nodedata = s.nextInt();
   }
   return root;
}

private static BSTnode<Integer> insertInBST(BSTnode<Integer> root, int nodedata){
   if(root == null){
      BSTnode<Integer> newNode = new BSTnode<>(nodedata);
      return newNode;
   }
   if(root.data.intValue() > nodedata){
      root.left = insertInBST(root.left,nodedata);
      return root;
   }else{
      root.right = insertInBST(root.right,nodedata);
      return root;
   }
}

public static<T> void display(BSTnode<T> root){
   if(root == null)
      return;
   System.out.print("\n"+root.data +" :");
   if(root.left != null){
      System.out.print(root.left.data+",");
   }
   if(root.right != null){
      System.out.print(root.right.data);
   }
   display(root.left);
   display(root.right);
}
```

```
}

//User class
public class BSTuse {
    public static void main(String[] args){
            BSTnode<Integer> root = BSTnode.insertNode();
            BSTnode.display(root);
        }
}
```

Sample input:

```
Enter root data :
10
Enter another node data :
14
Enter another node data :
5
Enter another node data :
7
Enter another node data :
4
Enter another node data :
6
Enter another node data :
-1
```

Output:

```
10 :5,14
5 :4,7
4 :
7 :6,
6 :
14 :
```

Searching in BST

To search a given key in Bianry Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

```
public BSTnode<Integer> search(BSTnode<Integer> root, int key){
    if (root==null || root.data.equals(key))
        return root;

    // val is greater than root's key
    if (root.data.intValue() > key)
        return search(root.left, key);

    // val is less than root's key
    return search(root.right, key);}
```