DATA STRUCTURES

WENYE LI
CUHK-SZ

OUTLINE

- Trees
- Types of Trees
- Implementation
- Examples

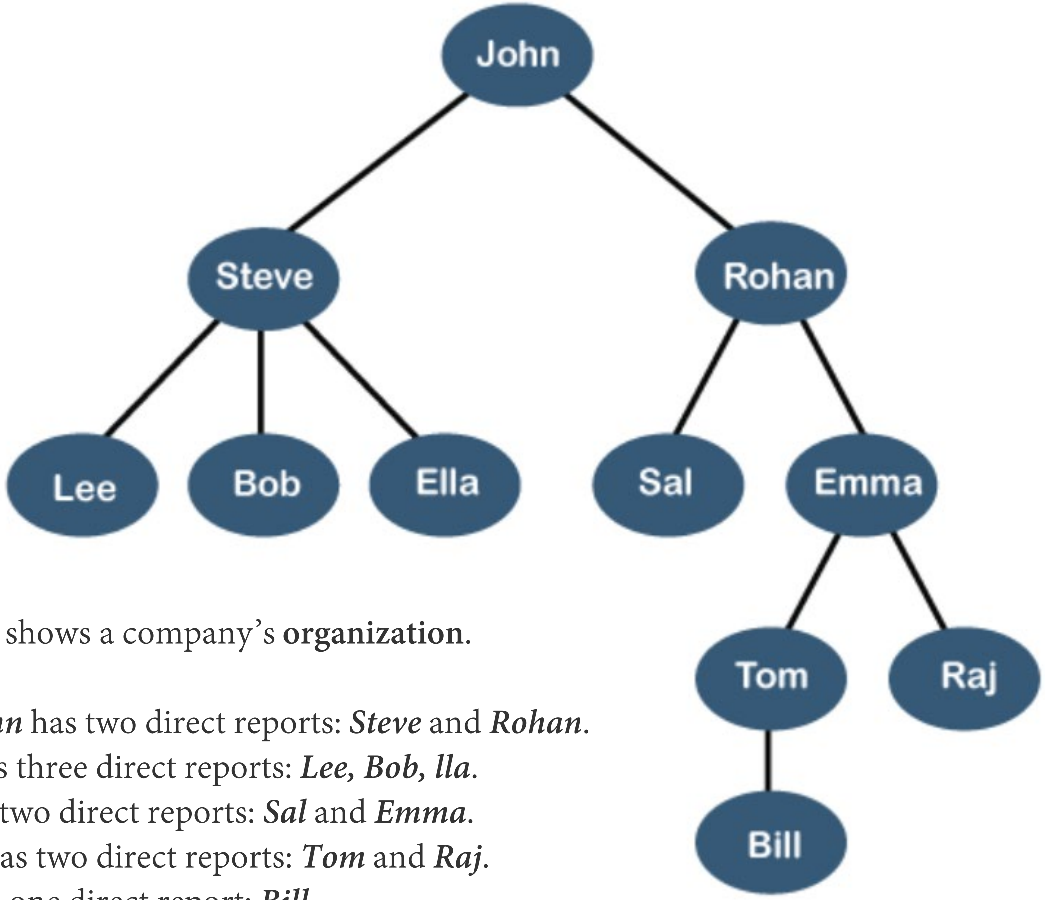
- Linear data structures like an array, linked list, stack and queue
  - all elements are arranged in a sequential manner.

- Factors considered for choosing data structure
  - **What type of data needs to be stored?**
    - It might be possible that a certain data structure can be the best fit for some kind of data.
  - **Cost of operations**
    - For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the binary search. The binary search works very fast for the simple list as it divides the search space into half.
  - **Memory usage**
    - Sometimes, we want a data structure that utilizes less memory.

TREES

- *A **tree*** is a data structure 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.



The tree shows a company's **organization**.

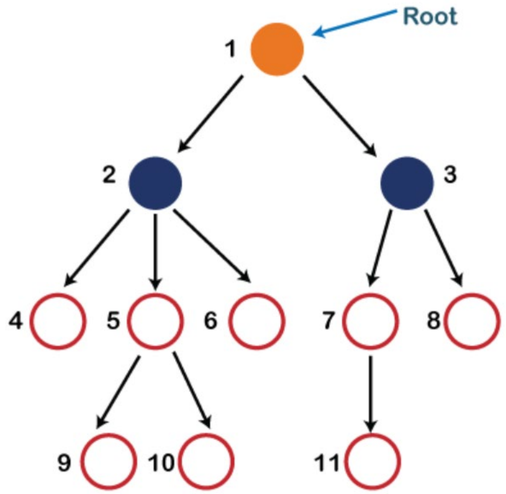
CEO ***John*** has two direct reports: ***Steve*** and ***Rohan***.
Steve has three direct reports: ***Lee, Bob, lla***.
Bob has two direct reports: ***Sal*** and ***Emma***.
Emma has two direct reports: ***Tom*** and ***Raj***.
Tom has one direct report: ***Bill***.

In this structure, the ***root*** is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

TREES

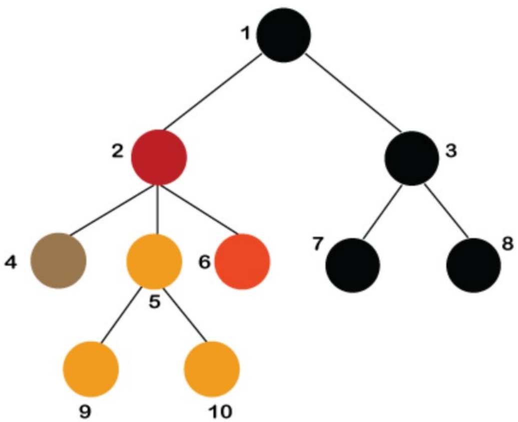
- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a tree are arranged in multiple levels.
- In a tree, the topmost node is known as a root node.
- Each node contains some data, which can be of any type.
- Each node contains the link or reference of other nodes that can be called children.

BASIC TERMS



- 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.
- 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. Leaf nodes can also be called external nodes.
- Internal nodes: A node has at least one child node known as an internal node.
- Ancestor node: An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors, e.g., nodes 1, 2, and 5 are the ancestors of node 10.
- Descendant: The immediate successor of the given node is known as a descendant of a node, e.g., 10 is the descendant of node 5.

PROPERTIES



- Recursive data structure: A tree can be defined as recursively. The root node contains a link to all the roots of its subtrees. The left subtree is shown in yellow, and the right subtree is shown in red. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner.

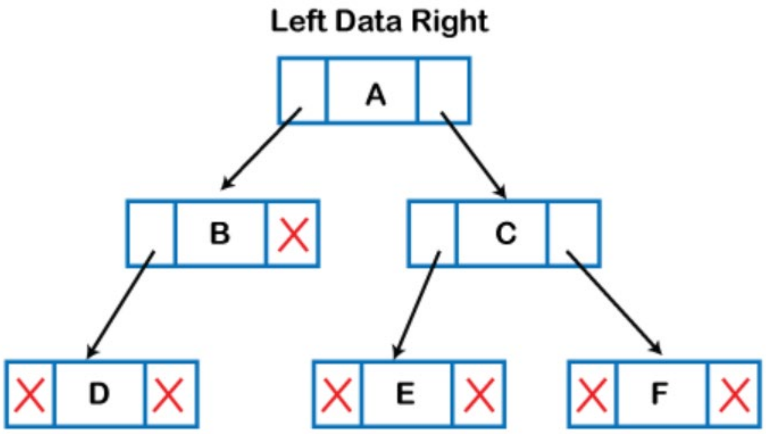
- Number of edges: If there are *n* nodes, then there would *n-1* edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have at least one incoming link known as an edge. There would be one link for the parent-child relationship.

- Depth of node *x*: the length of the path from the root to node *x*, i.e., the number of edges between the root node and node *x*. The root node has *0* depth.

- Height of node *x*: the longest path from the node *x* to the leaf node.

IMPLEMENTATION

- A tree can be created by creating the nodes dynamically with the help of the pointers.
- A node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.
- Note: This structure can only be defined for binary trees because a binary tree have at most two children, and generic trees can have more than two children. The node structure for generic trees would be different as compared to the binary tree.



APPLICATIONS

- Storing naturally hierarchical data: The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- Organize data: It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a log$N$ time for searching an element.
- Trie: It is a special kind of tree that is used to store the dictionary, fast and efficient for dynamic spell checking.
- Heap: It is a tree data structure implemented using arrays. It is used to implement priority queues.
- B-Tree and B+Tree: B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- Routing table: The tree data structure is used to store the data in routing tables in the routers.

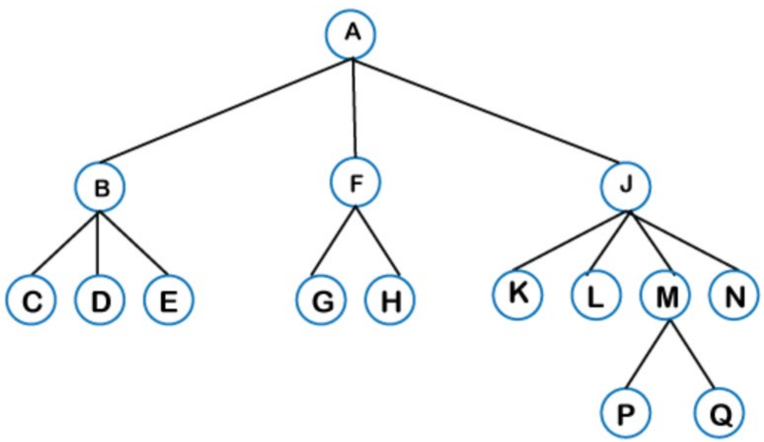
OUTLINE

- Trees
- Types of Trees
- Implementation
- Examples

TYPES OF TREES

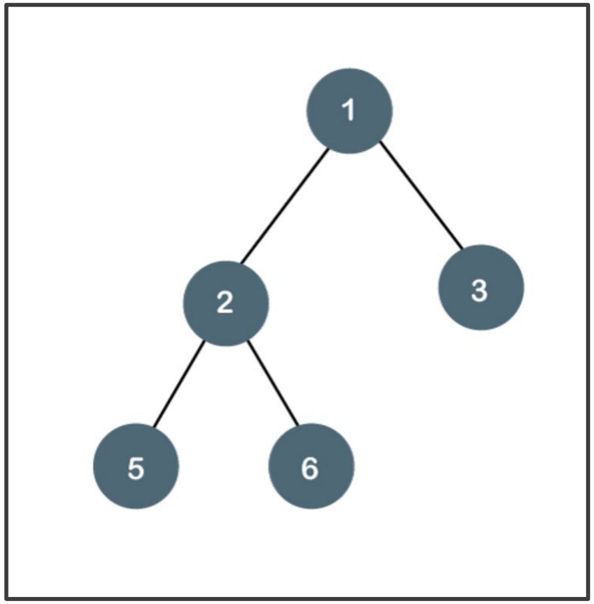
- General Tree
- Binary Tree
- Binary Search Tree
- AVL Tree: in separate slides
- Red-Black Tree: in separate slides

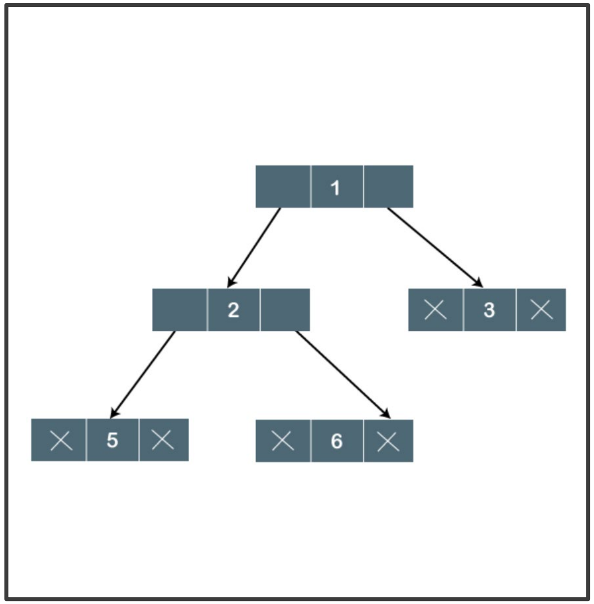
GENERAL TREE



- 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). The topmost node is known as a root node. The children of the parent node are known as subtrees.
- There can be *n* number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.
- Every non-empty tree has a downward edge, and these edges are connected to the nodes known as child nodes. The root node is labeled with level *0*. The nodes that have the same parent are known as siblings.

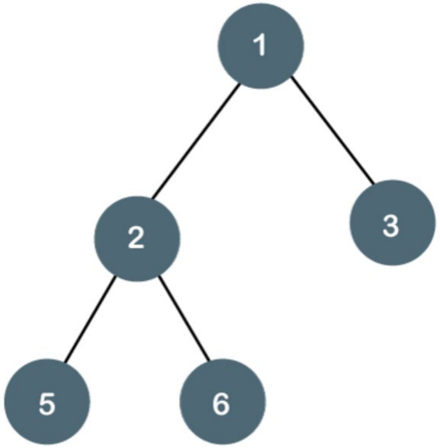
BINARY TREE





- Binary tree means that the node can have maximum two children. Each node can have either 0, 1 or 2 children.
- In the example,
  - Node 1 contains left and right pointers pointing to left and right nodes respectively.
  - Node 2 contains both left and right nodes.
  - Nodes 3, 5 and 6 are leaf nodes; all these nodes contain NULL pointer on both left and right parts.

PROPERTIES OF BINARY TREE



- At each level of *i*, the maximum number of nodes is 2^i.
- The height of a tree is defined as the longest path from the root node to the leaf node.
  - The maximum number of nodes at height 3 is (1+2+4+8) = 15.
  - In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots . 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height *h* is equal to ***h+1***.

- If the number of nodes is minimum, then the height of the tree would be maximum.
- If the number of nodes is maximum, then the height of the tree would be minimum.

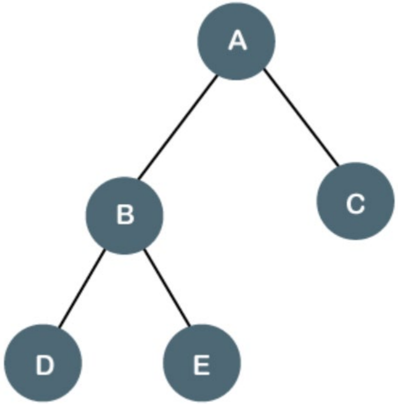
PROPERTIES OF BINARY TREE

- Suppose a binary tree has '*n*' nodes. The minimum height can be computed as:
  - As we know that, $n = 2^{h+1} - 1$ and $n+1 = 2^{h+1}$
  - Taking log on both the sides,
  - $log_2(n+1) = log2(2^{h+1})$
  - $log_2(n+1) = h+1$
  - $h = log_2(n+1) - 1$
- The maximum height can be computed as:
  - As we know that, $n = h+1$
  - $\boldsymbol{h = n-1}$

TYPES OF BINARY TREE

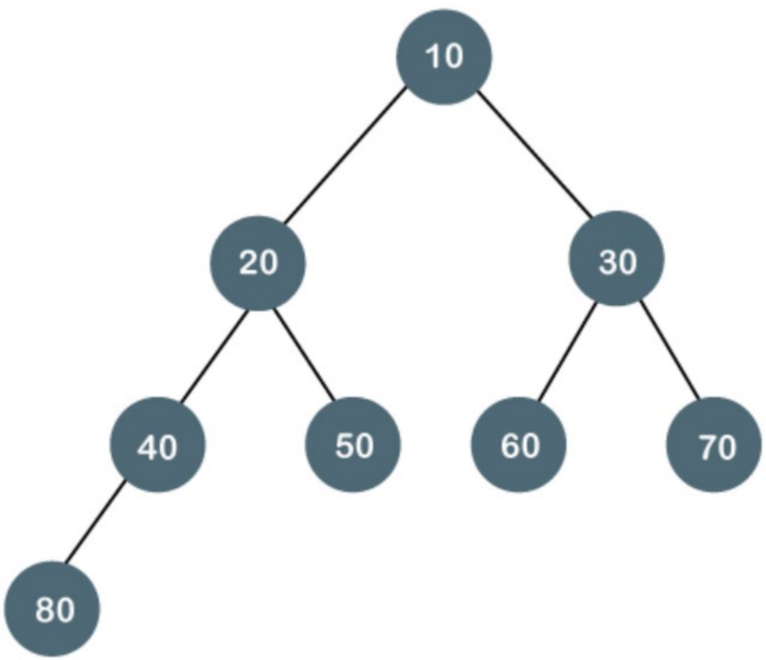
- Full/ proper/ strict Binary tree
- Complete Binary tree
- Perfect Binary tree
- Balanced Binary tree

FULL BINARY TREE



- Each node contains either zero or two children.
- The number of leaf nodes is equal to the number of internal nodes plus 1.
- The maximum number of nodes is $\mathbf{2^{h+1} - 1}$.
- The minimum number of nodes is $\mathbf{2*h + 1}$.
- The minimum height of the full binary tree is $\mathbf{\log_2(n+1) - 1}$.
- The maximum height of the full binary tree can be computed as:
  - $n = 2*h + 1$
  - $n-1 = 2*h$
  - $h = (n-1)/2$

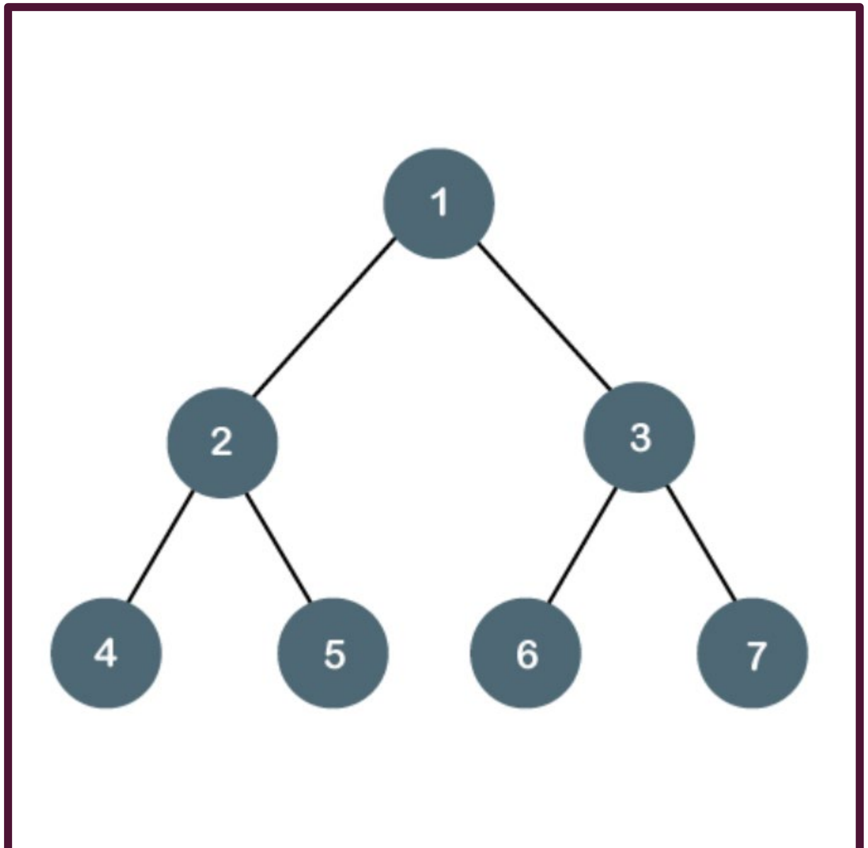
COMPLETE BINARY TREE



- All nodes are completely filled except the last level. In the last level, all nodes must be as left as possible.

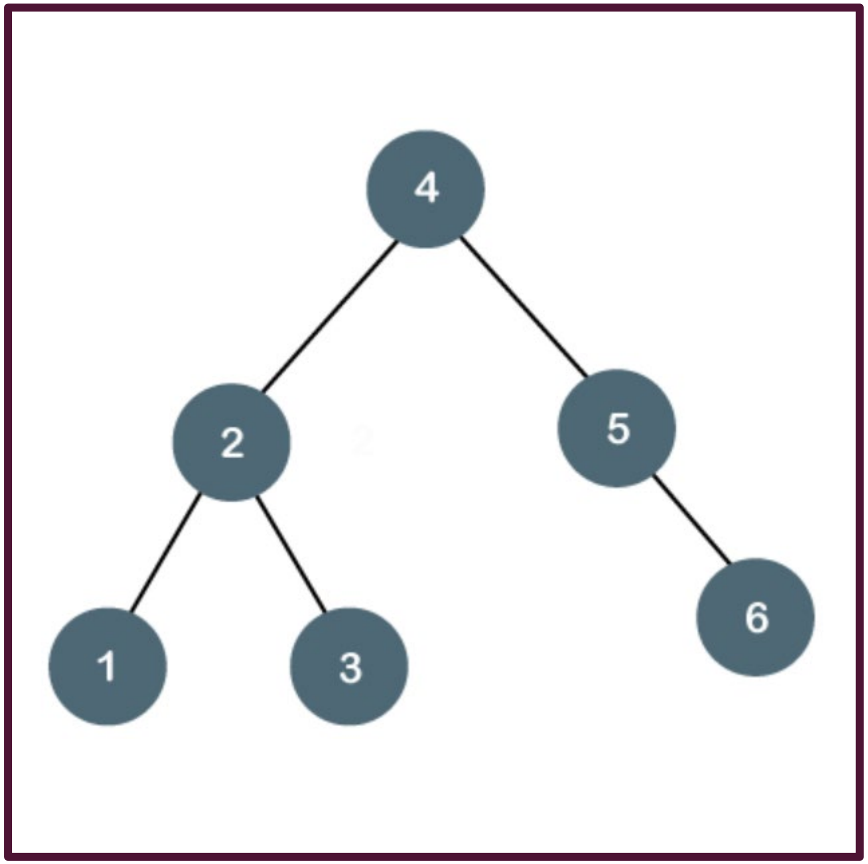
- The maximum number of nodes in complete binary tree is $\mathbf{2^{h+1} - 1}$.
- The minimum number of nodes in complete binary tree is $\mathbf{2^h}$.
- The minimum height of a complete binary tree is $\mathbf{\log_2(n+1) - 1}$.
- The maximum height of a complete binary tree is $\mathbf{\log_2(n)}$.

PERFECT BINARY TREE



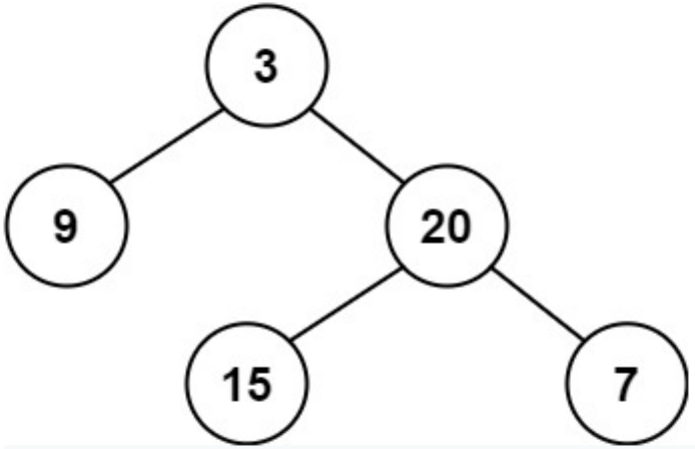
- All internal nodes have 2 children, and all leaf nodes are at the same level.
- All perfect binary trees are complete binary trees as well as full binary trees. But vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

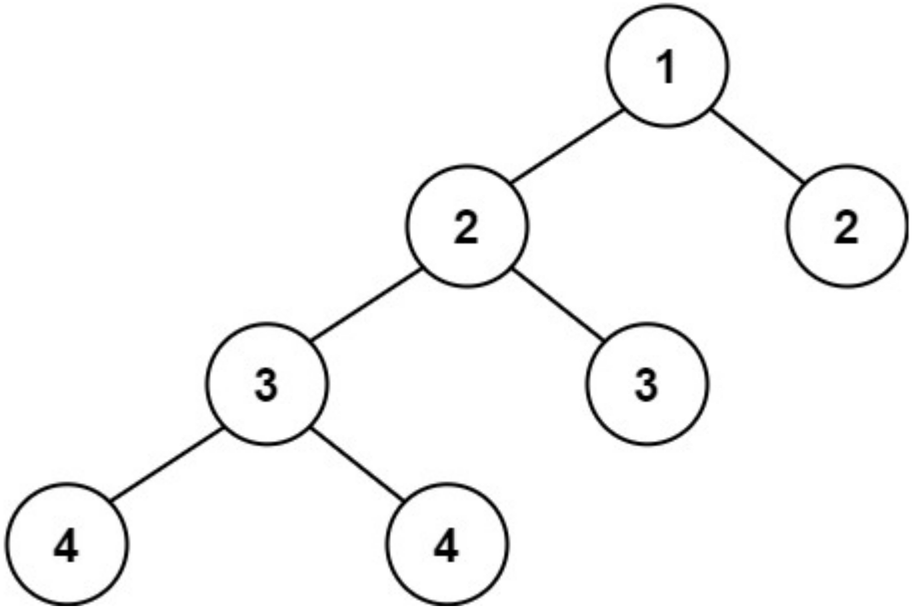
BALANCED BINARY TREE



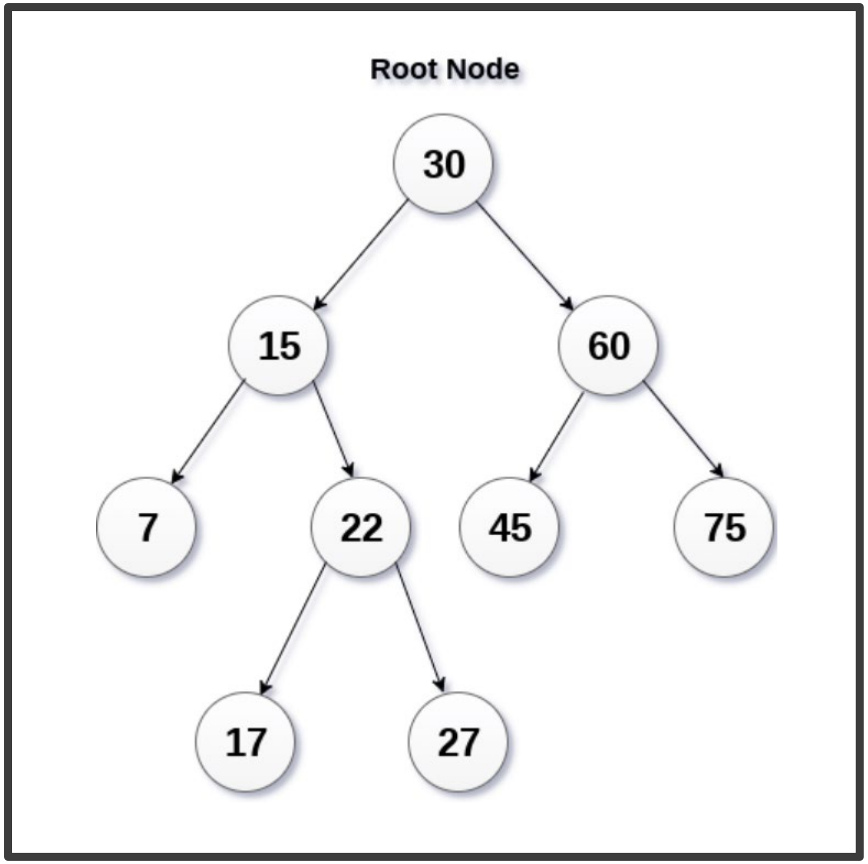
- A binary tree in which the left and right subtrees of every node differ in height by no more than 1.
  - E.g., ***AVL*** and ***Red-Black trees*** are balanced binary tree.

BALANCED OR NOT?





BINARY SEARCH TREE



- A class of binary trees, in which the nodes are arranged in a specific order, also called ordered binary tree.
- The value of all nodes in the left sub-tree is less than the value of the root.
- The value of all nodes in the right sub-tree is greater than or equal to the value of the root.
- The rule is recursively applied to all left and right sub-trees of the root.

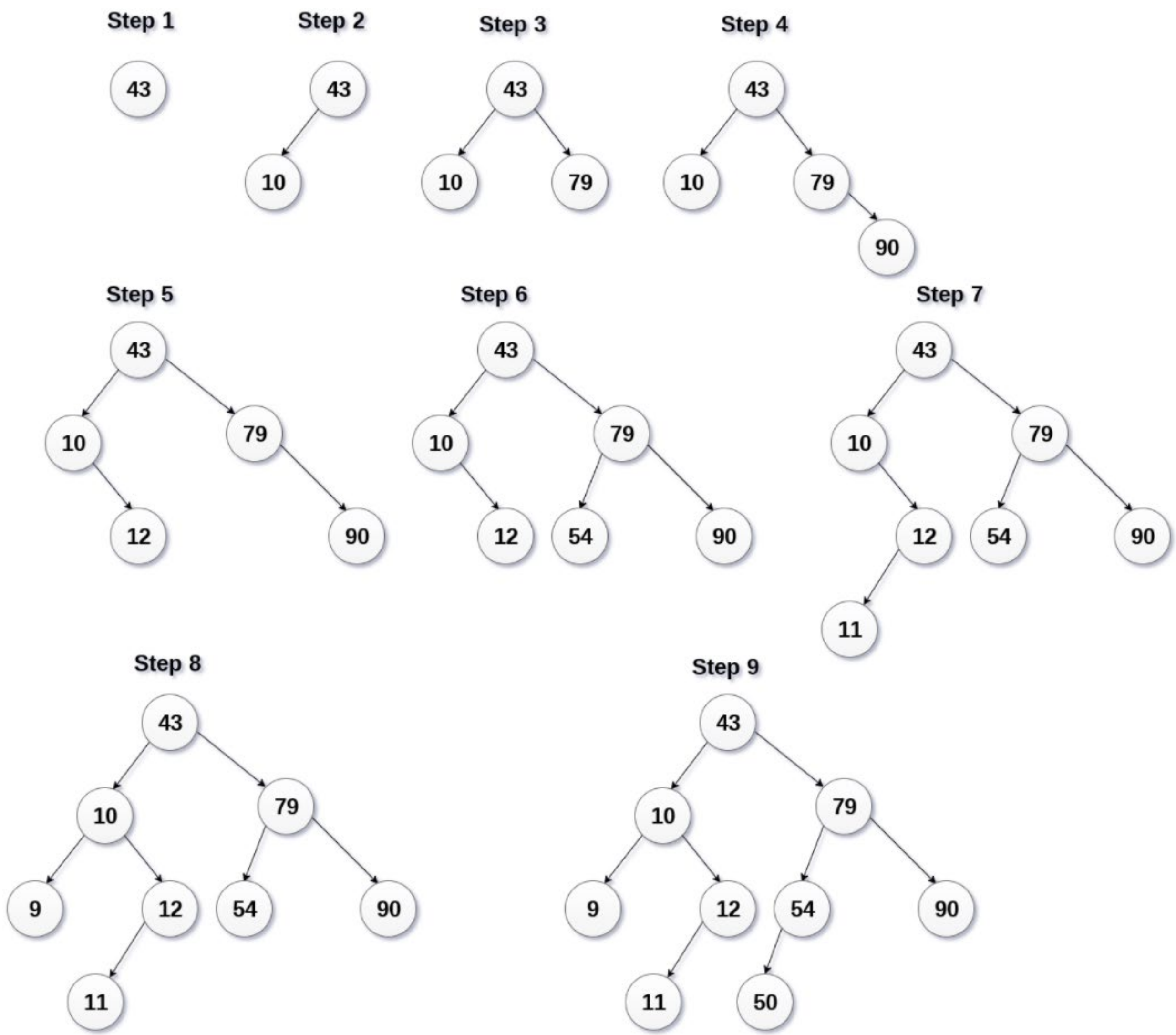
ADVANTAGES OF BST

- Searching is very efficient in a BST since, we get a hint at each step, about which sub-tree contains the desired element.
- BST is considered as efficient data structure in comparison to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $o(\log_2 n)$ time. In worst case, the time it takes to search an element is $0(n)$.
- It also speed up the insertion and deletion operations as comparison to that in array and linked list.

CREATE BST

- **43, 10, 79, 90, 12, 54, 11, 9, 50**

- Insert 43 into the tree as the root of the tree.
- Read next element. If it is less than the root node, insert it as the root of the left sub-tree.
- Otherwise, insert it as the root of the right sub-tree.

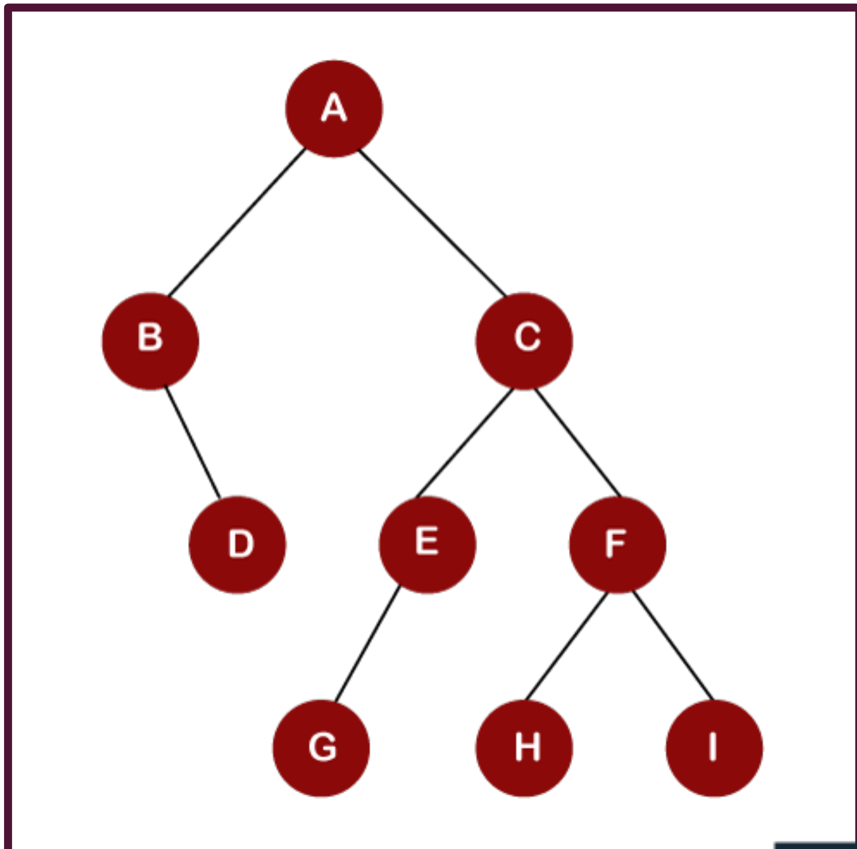


TRAVERSAL OF BINARY TREE

- Tree traversal: traversing or visiting each node of a tree.
  - Linear data structures like stack, queue, linked list have only one way for traversing.

- Tree has various ways to traverse/visit each node.
  - Inorder traversal
  - Preorder traversal
  - Postorder traversal

INORDER TRAVERSAL

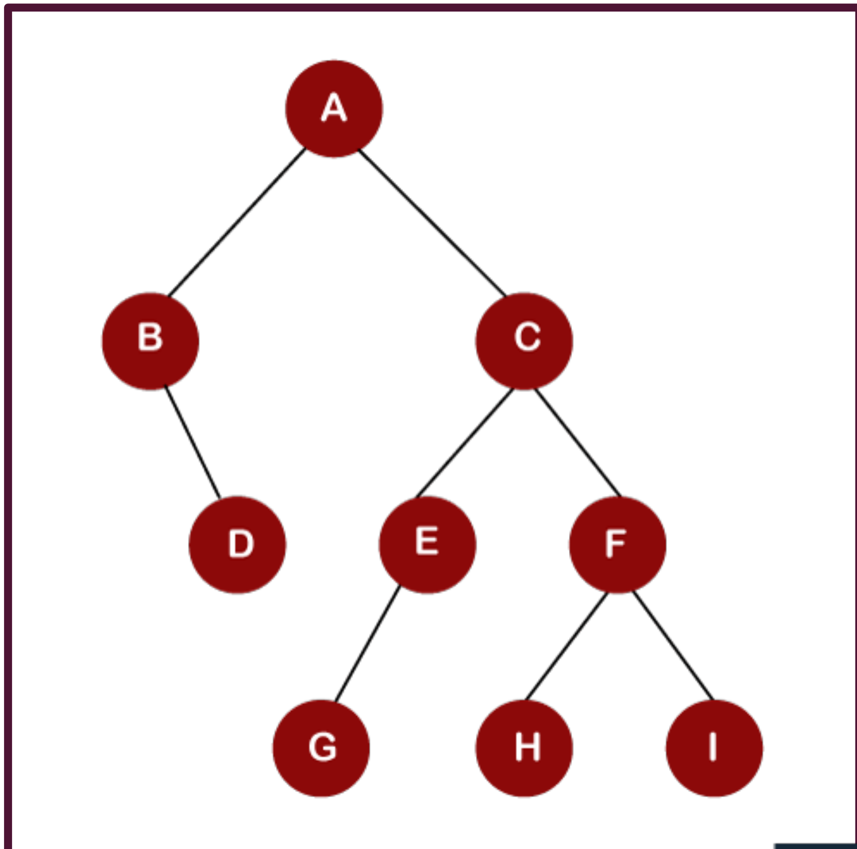


- **Left Root Right**
  - The left subtree of the root is traversed;
  - then the root node;
  - then the right subtree.

- INORDER(TREE):
  - Step 1: Repeat Steps 2 to 4 while TREE != NULL
  - Step 2: INORDER(TREE->LEFT)
  - Step 3: Write TREE -> DATA
  - Step 4: INORDER(TREE -> RIGHT)
  - Step 5: END

B	D	A	G	E	C	H	F	I

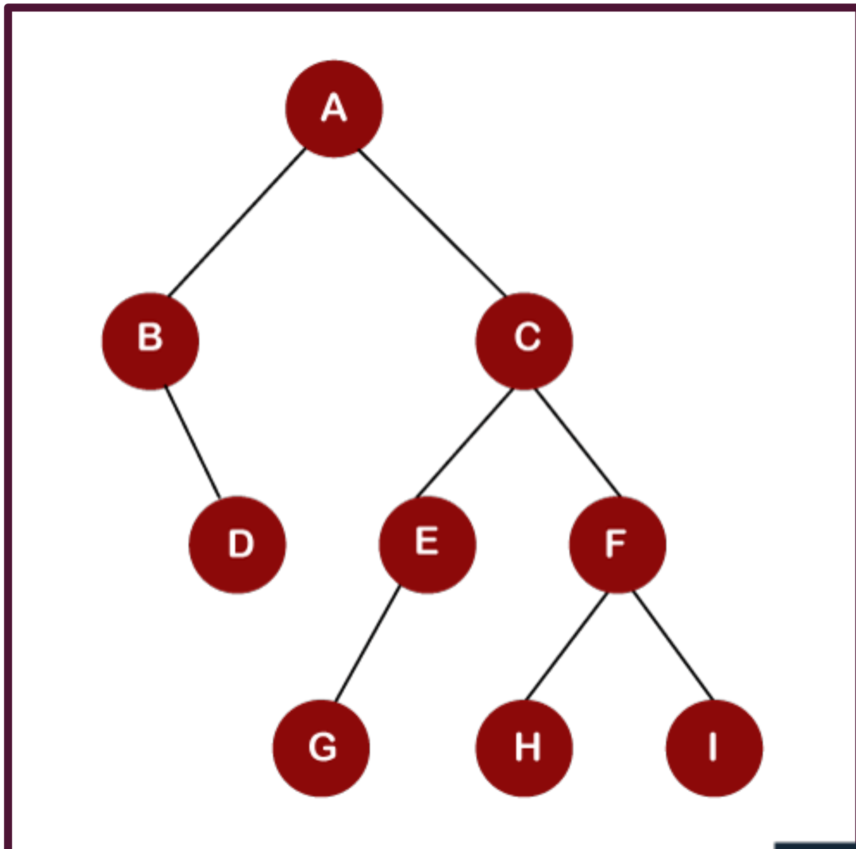
PREORDER TRAVERSAL



- **Root Left Right**
  - Root node of the tree is traversed;
  - then the left subtree;
  - then the right subtree is traversed.
- PREORDER(TREE):
  - Step 1: Repeat Steps 2 to 4 while TREE != NULL
  - Step 2: Write TREE -> DATA
  - Step 3: PREORDER(TREE -> LEFT)
  - Step 4: PREORDER(TREE -> RIGHT)
  - Step 5: END

A	B	D	C	E	G	F	H	I

POSTORDER TRAVERSAL



- **Left Right Root**
  - the left subtree of the root is traversed;
  - then the right subtree;
  - then the root node.

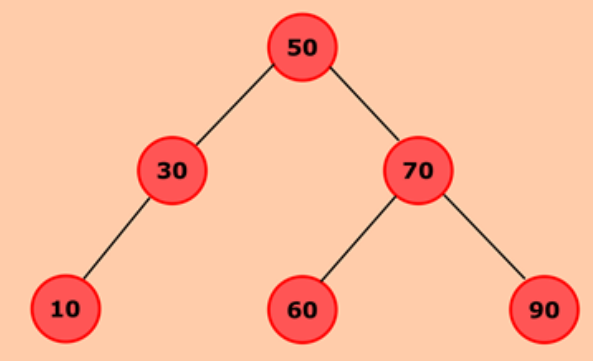
- POSTORDER(TREE):
  - Step 1: Repeat Steps 2 to 4 while TREE != NULL
  - Step 2: POSTORDER(TREE -> LEFT)
  - Step 3: POSTORDER(TREE -> RIGHT)
  - Step 4: Write TREE -> DATA
  - Step 5: END

B	D	A	G	E	C	H	F	I

OUTLINE

- Trees
- Types of Trees
- Implementation
- Examples

BST IMPLEMENTATION

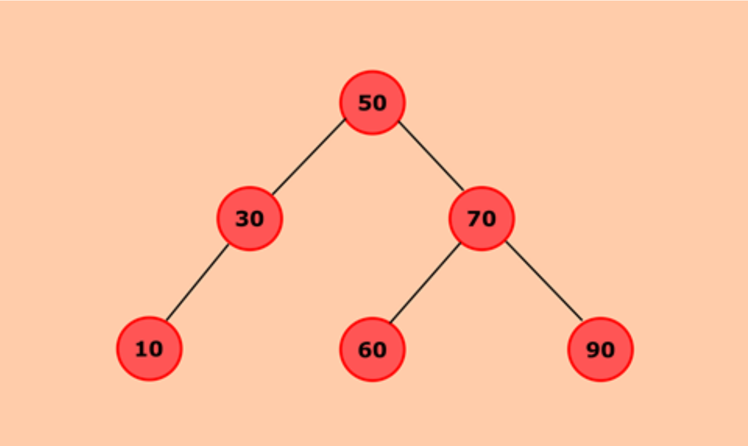


- Define Node class with three attributes: data, left and right.
  - Left represents the left child of the node and right represents the right child of the node.
  - Root represents the root node of the tree and initializes it to null.
- insert(): insert the new value into a BST
  - If the new value is less than the root node, it will be inserted to left subtree; else, to right subtree.
- deleteNode(): delete a particular node from a BST
  - If the node to delete is a leaf node, parent of that node will point to null.
    - If we delete 90, parent node 70 will point to null.
  - If the node to delete has one child node, the child node will become a child node of the parent node.
    - If we delete 30, node 10 which was left child of 30 will become left child of 50.
  - If the node to delete has two children, find minNode with minimum value from the right subtree of the current node. The current node is replaced by minNode.

```java
public class BinarySearchTree {
    //Represent a node of binary tree
    public static class Node {
        int data;
        Node left;
        Node right;
        public Node(int data) {
            //Assign data to the new node, set left and right children to null
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    //Represent the root of binary tree
    public Node root;
    public BinarySearchTree() {
        root = null;
    }
    //insert() will add new node to the binary search tree
    public void insert(int data) {
        //Create a new node
        Node newNode = new Node(data);
        //Check whether tree is empty
        if(root == null) {
            root = newNode;
            return;
        } else {
            //current node point to root of the tree
            Node current = root, parent = null;
            while(true) {
                //parent keep track of the parent node of current node.
                parent = current;
                //If data is less than current's data, node will be inserted to the left of tree
                if(data < current.data) {
                    current = current.left;
                    if(current == null) {
                        parent.left = newNode;
                        return;
                    }
                } else { //If data is greater than current's data, node will be inserted to the right of tree
                    current = current.right;
                    if(current == null) {
                        parent.right = newNode;
                        return;
                    }
                }
            }
        }
    }
```

```java
    //minNode() will find out the minimum node
    public Node minNode(Node root) {
        if (root.left != null)
            return minNode(root.left);
        else
            return root;
    }
    //deleteNode() will delete the given node from the binary search tree
    public Node deleteNode(Node node, int value) {
        if(node == null) {
            return null;
        } else {
            //value is less than node's data then, search the value in left subtree
            if(value < node.data)
                node.left = deleteNode(node.left, value);
            else if(value > node.data) //value is greater than node's data then, search the value in right subtree
                node.right = deleteNode(node.right, value);
            else { //If value is equal to node's data that is, we have found the node to be deleted
                //If node to be deleted has no child then, set the node to null
                if(node.left == null && node.right == null)
                    node = null;
                else if(node.left == null) { //If node to be deleted has only one right child
                    node = node.right;
                } else if(node.right == null) { //If node to be deleted has only one left child
                    node = node.left;
                } else { //If node to be deleted has two children node
                    //then find the minimum node from right subtree
                    Node temp = minNode(node.right);
                    //Exchange the data between node and temp
                    node.data = temp.data;
                    //Delete the node duplicate node from right subtree
                    node.right = deleteNode(node.right, temp.data);
                }
            }
            return node;
        }
    }
    //inorder() will perform inorder traversal on binary search tree
    public void inorderTraversal(Node node) {
        //Check whether tree is empty
        if(root == null) {
            System.out.println("Tree is empty");
            return;
        } else {
            if(node.left != null)
                inorderTraversal(node.left);
            System.out.print(node.data + " ");
            if(node.right != null)
                inorderTraversal(node.right);
        }
    }
```

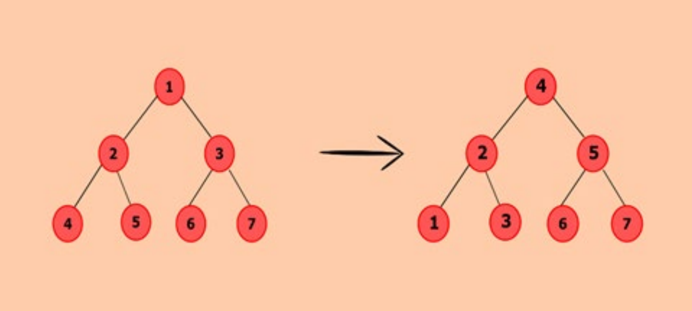
```java
    public static void main(String[] args) {

        BinarySearchTree bt = new BinarySearchTree();
        //Add nodes to the binary tree
        bt.insert(50);
        bt.insert(30);
        bt.insert(70);
        bt.insert(60);
        bt.insert(10);
        bt.insert(90);

        System.out.println("Binary search tree after insertion:");
        //Displays the binary tree
        bt.inorderTraversal(bt.root);

        Node deletedNode = null;
        //Deletes node 90 which has no child
        deletedNode = bt.deleteNode(bt.root, 90);
        System.out.println("\nBinary search tree after deleting node 90:");
        bt.inorderTraversal(bt.root);

        //Deletes node 30 which has one child
        deletedNode = bt.deleteNode(bt.root, 30);
        System.out.println("\nBinary search tree after deleting node 30:");
        bt.inorderTraversal(bt.root);

        //Deletes node 50 which has two children
        deletedNode = bt.deleteNode(bt.root, 50);
        System.out.println("\nBinary search tree after deleting node 50:");
        bt.inorderTraversal(bt.root);
    }
}
```



Output:

```
Binary search tree after insertion:
10 30 50 60 70 90
Binary search tree after deleting node 90:
10 30 50 60 70
Binary search tree after deleting node 30:
10 50 60 70
Binary search tree after deleting node 50:
10 60 70
```

CONVERT BINARY TREE TO BST



- convertBTBST():
  - Convert binary tree to corresponding array by calling convertBTtoArray().
  - Sort the resultant array from in ascending order.
  - Convert the array to the binary search tree by calling createBST().
  - calculateSize() counts the number of nodes present in the tree.
  - convertBTtoArray() converts binary tree to array representation.
  - createBST() creates a corresponding binary search tree by selecting a middle node of sorted treeArray as it the root node. treeArray is divided into two parts: [0, mid-1] and [mid+1, end]. Recursively find middle node from each array to create left subtree and right subtree respectively.
  - Inorder() displays nodes in inorder fashion, i.e., left child followed by root followed by right child.

```java
import java.util.Arrays;

public class ConvertBTtoBST {

    //Represent a node of binary tree
    public static class Node {
        int data;
        Node left;
        Node right;
        public Node(int data) {
            //Assign data to the new node, set left and right children to null
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    //Represent the root of binary tree
    public Node root;
    int[] treeArray;
    int index = 0;
    public ConvertBTtoBST() {
        root = null;
    }

    //convertBTBST() will convert a binary tree to binary search tree
    public Node convertBTBST(Node node) {
        //Variable treeSize will hold size of tree
        int treeSize = calculateSize(node);
        treeArray = new int[treeSize];

        //Converts binary tree to array
        convertBTtoArray(node);

        //Sort treeArray
        Arrays.sort(treeArray);

        //Converts array to binary search tree
        Node d = createBST(0, treeArray.length - 1);
        return d;
    }
    //calculateSize() will calculate size of tree
    public int calculateSize(Node node) {
        int size = 0;
        if (node == null)
            return 0;
        else {
            size = calculateSize (node.left) + calculateSize (node.right) + 1;
            return size;
        }
    }
```

```java
    //convertBTtoArray() will convert the given binary tree to its corresponding array representation
    public void convertBTtoArray(Node node) {
        //Check whether tree is empty
        if(root == null) {
            System.out.println("Tree is empty");
            return;
        } else {
            if(node.left != null)
                convertBTtoArray(node.left);
            //Adds nodes of binary tree to treeArray
            treeArray[index] = node.data;
            index++;
            if(node.right != null)
                convertBTtoArray(node.right);
        }
    }
    //createBST() will convert array to binary search tree
    public Node createBST(int start, int end) {
        //It will avoid overflow
        if (start > end) {
            return null;
        }

        //Variable will store middle element of array and make it root of binary search tree
        int mid = (start + end) / 2;
        Node node = new Node(treeArray[mid]);

        //Construct left subtree
        node.left = createBST(start, mid - 1);

        //Construct right subtree
        node.right = createBST(mid + 1, end);

        return node;
    }
    //inorder() will perform inorder traversal on binary search tree
    public void inorderTraversal(Node node) {
        //Check whether tree is empty
        if(root == null) {
            System.out.println("Tree is empty");
            return;
        } else {
            if(node.left != null)
                inorderTraversal(node.left);
            System.out.print(node.data + " ");
            if(node.right != null)
                inorderTraversal(node.right);
        }
    }

```

```java
    public static void main(String[] args) {

        ConvertBTtoBST bt = new ConvertBTtoBST();
        //Add nodes to the binary tree
        bt.root = new Node(1);
        bt.root.left = new Node(2);
        bt.root.right = new Node(3);
        bt.root.left.left = new Node(4);
        bt.root.left.right = new Node(5);
        bt.root.right.left = new Node(6);
        bt.root.right.right = new Node(7);

        //Display given binary tree
        System.out.println("Inorder representation of binary tree: ");
        bt.inorderTraversal(bt.root);

        //Converts binary tree to corresponding binary search tree
        Node bst = bt.convertBTBST(bt.root);

        //Display corresponding binary search tree
        System.out.println("\nInorder representation of resulting binary search tree: ");
        bt.inorderTraversal(bst);
    }
}
```

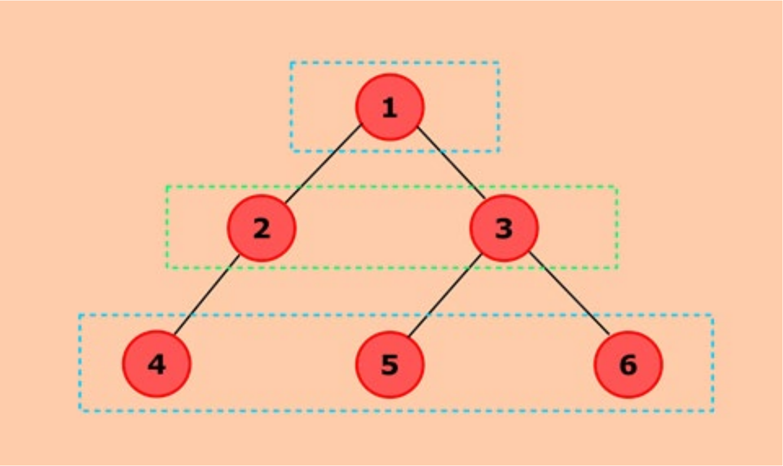
Output:

```
Inorder representation of binary tree:
4 2 5 1 6 3 7
Inorder representation of resulting binary search tree:
1       2 3 4 5 6 7
```

OUTLINE

- Trees
- Types of Trees
- Implementation
- Examples

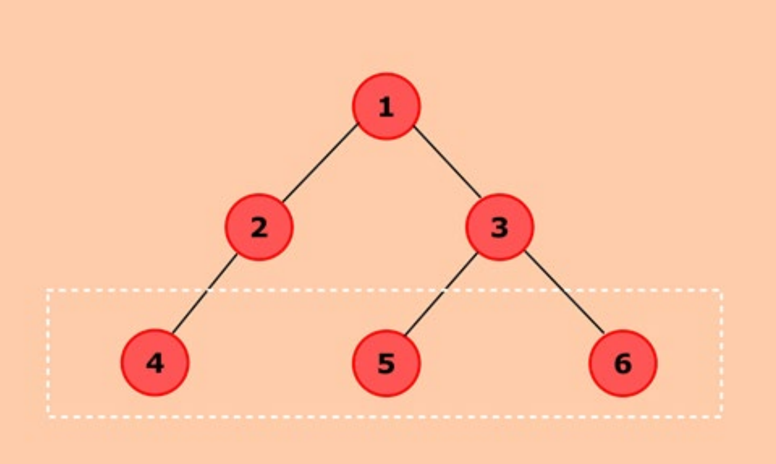
DIFFERENCE BETWEEN SUM OF ODD LEVEL AND EVEN LEVEL NODES OF A BINARY TREE



- Difference = (L1 + L 3 + L5) - (L2 + L4)
  - OddLevelSum = 1 + 4 + 5 + 6 = 16
  - EvenLevelSum = 2 + 3 = 5
  - Difference = |16 - 5| = 11
- difference() :
  - Traverse through the binary tree level wise using Queues.
  - Keep track of current level using the variable currentLevel.
  - If the currentLevel is divisible by 2, then add all values of nodes in currentLevel to variable evenLevel. Else, add all values of nodes to variable oddLevel.
  - Calculate the difference by subtracting value present in evenLevel from oddLevel.

```java
    //difference() will calculate the difference between sum of odd and even levels of binary tree
    public int difference() {
        int oddLevel = 0, evenLevel = 0, diffOddEven = 0;
        //Variable nodesInLevel keep tracks of number of nodes in each level
        int nodesInLevel = 0;
        //Variable currentLevel keep track of level in binary tree
        int currentLevel = 0;
        //Queue will be used to keep track of nodes of tree level-wise
        Queue<Node> queue = new LinkedList<Node>();
        //Check if root is null
        if(root == null) {
            System.out.println("Tree is empty");
            return 0;
        } else {
            //Add root node to queue as it represents the first level
            queue.add(root);
            currentLevel++;
            while(queue.size() != 0) {
                //Variable nodesInLevel will hold the size of queue i.e. number of elements in queue
                nodesInLevel = queue.size();
                while(nodesInLevel > 0) {
                    Node current = queue.remove();
                    //Checks if currentLevel is even or not.
                    if(currentLevel % 2 == 0)
                        //If level is even, add nodes's to variable evenLevel
                        evenLevel += current.data;
                    else
                        //If level is odd, add nodes's to variable oddLevel
                        oddLevel += current.data;

                    //Adds left child to queue
                    if(current.left != null)
                        queue.add(current.left);
                    //Adds right child to queue
                    if(current.right != null)
                        queue.add(current.right);
                    nodesInLevel--;
                }
                currentLevel++;
            }
            //Calculates difference between oddLevel and evenLevel
            diffOddEven = Math.abs(oddLevel - evenLevel);
        }
        return diffOddEven;
    }
```

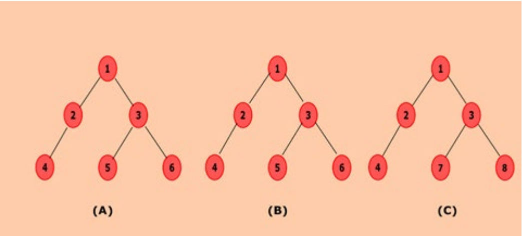
ALL LEAVES IN THE SAME LEVEL?



- isSameLevel(): check whether all leaves of given binary tree are at same level or not
  - It checks whether the root is null, which means the tree is empty.
  - If the tree is not empty, traverse through the tree and check for leaf node whose left and right children are null.
  - CurrentLevel will keep track of current level being traversed.
  - When the first leaf node is encountered, store the value of currentLevel in variable level.
  - Traverse recursively through all level, check for subsequent leaf nodes. If currentLevel of all leaf is equal to the value stored in level then, all leaves are at same level.

```java
public class LeafLevel {
    //Represent a node of binary tree
    public static class Node {
        int data;
        Node left;
        Node right;
        public Node(int data) {
            //Assign data to the new node, set left and right children to null
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }
    //Represent the root of binary tree
    public Node root;
    //It will store level of first encountered leaf
    public static int level = 0;
    public LeafLevel() {
        root = null;
    }
    //isSameLevel() will check whether all leaves of the binary tree is at same level or not
    public boolean isSameLevel(Node temp, int currentLevel ) {
        if(root == null) { //Check whether tree is empty
            System.out.println("Tree is empty");
            return true;
        } else {
            if(temp == null) //Checks whether node is null
                return true;
            if(temp.left == null && temp.right == null) {
                if(level == 0) { //If first leaf is encountered, set level to current level
                    level = currentLevel ;
                    return true;
                } else //Checks whether the other leaves are at same level of that of first leaf
                    return (level == currentLevel);
            }
            //Checks for leaf node in left and right subtree recursively.
            return  (isSameLevel(temp.left, currentLevel + 1) && isSameLevel(temp.right, currentLevel + 1));
        }
    }
    public static void main (String[] args) {
        LeafLevel bt = new LeafLevel();
        //Add nodes to the binary tree
        bt.root = new Node(1);
        bt.root.left = new Node(2);
        bt.root.right = new Node(3);
        bt.root.left.left = new Node(4);
        bt.root.right.left = new Node(5);
        bt.root.right.right = new Node(6);
        //Checks whether all leaves of given binary tree is at same level
        if(bt.isSameLevel(bt.root, 1))
            System.out.println("All leaves are at same level");
        else
            System.out.println("All leaves are not at same level");
    }
}
```

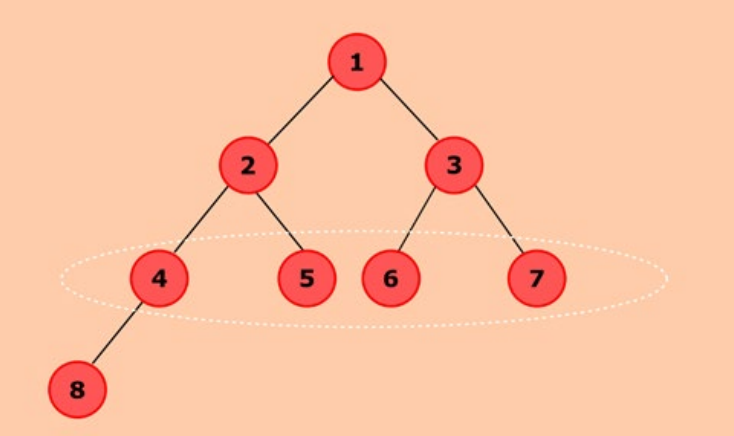
TWO TREES ARE IDENTICAL?



- areIdenticalTrees(): check two trees are identical or not?
  - If root nodes of both the trees are null, they are identical.
  - If the root node of only one tree is null, trees are not identical, return false.
  - If root node of none of the tree is null,
    - check whether data of both the nodes are equal,
    - and then recursively check the left subtree and right subtree of one tree is identical to another or not.

```java
public class IdenticalTrees {
    //Represent the node of the binary tree
    public static class Node {
        int data;
        Node left;
        Node right;
        public Node(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }
    public Node root; //Represent the root of the binary tree
    public IdenticalTrees() {
        root = null;
    }
    //areIdenticalTrees() finds whether two trees are identical or not
    public static boolean areIdenticalTrees(Node root1, Node root2) {
        //Checks if both the trees are empty
        if(root1 == null && root2 == null)
            return true;
        //Trees are not identical if root of only one tree is null thus, return false
        if(root1 == null && root2 == null)
            return true;
        //If both trees are not empty, check whether the data of the nodes is equal
        //Repeat the steps for left subtree and right subtree
        if(root1 != null  && root2 != null) {
            return ((root1.data == root2.data) &&
                    (areIdenticalTrees(root1.left, root2.left)) &&
                    (areIdenticalTrees(root1.right, root2.right)));
        }
        return false;
    }
    public static void main(String[] args) {
        IdenticalTrees bt1 = new IdenticalTrees(); //Adding nodes to the first binary tree
        bt1.root = new Node(1);
        bt1.root.left = new Node(2);
        bt1.root.right = new Node(3);
        bt1.root.left.left = new Node(4);
        bt1.root.right.left = new Node(5);
        bt1.root.right.right = new Node(6);
        IdenticalTrees bt2 = new IdenticalTrees(); //Adding nodes to the second binary tree
        bt2.root = new Node(1);
        bt2.root.left = new Node(2);
        bt2.root.right = new Node(3);
        bt2.root.left.left = new Node(4);
        bt2.root.right.left = new Node(5);
        bt2.root.right.right = new Node(6);
        //Displays whether both the trees are identical or not
        if(areIdenticalTrees(bt1.root, bt2.root))
            System.out.println("Both the binary trees are identical");
        else
            System.out.println("Both the binary trees are not identical");
    }
}
```

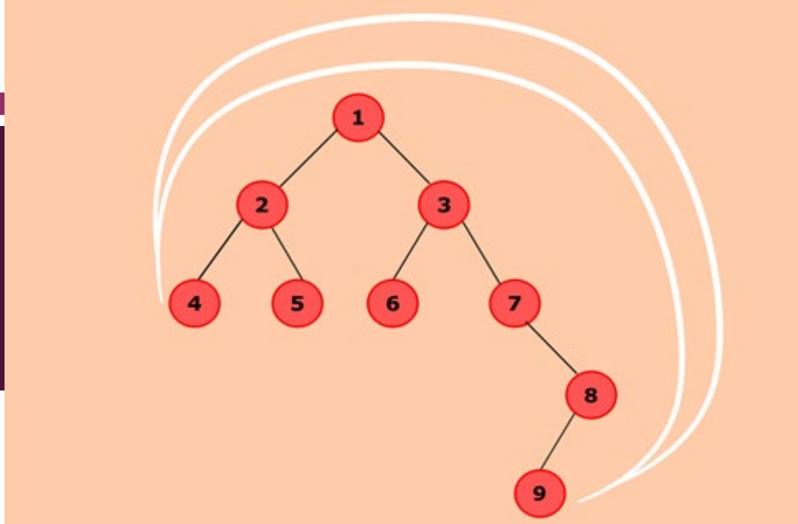
MAXIMUM WIDTH OF A BINARY TREE



- The maximum width of the binary tree is 4 denoted by white ellipse.
- findMaximumWidth(): find out the maximum width of the given binary tree
  - Variable maxWidth stores the maximum number of nodes present in any level.
  - The queue is used for traversing binary tree level-wise.
  - Check whether the root is null, which means the tree is empty.
  - If not empty, add the root node to queue. Variable nodesInLevel keeps track of the number of nodes in each level.
  - If nodesInLevel > 0, remove the node from the front of the queue and add its left and right child to the queue. For the first iteration, node 1 will be removed and its children nodes 2 and 3 will be added to the queue. In the second iteration, node 2 will be removed, its children 4 and 5 will be added to the queue and so on.
  - MaxWidth stores max(maxWidth, nodesInLevel). At any given point of time, it represents the maximum number of nodes.
  - This continues till all the levels of the tree is traversed.

```java
import java.util.LinkedList;
import java.util.Queue;
public class BinaryTree {
    //Represent the node of binary tree
    public static class Node {
        int data;
        Node left;
        Node right;
        public Node(int data) {
            //Assign data to the new node, set left
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    //Represent the root of binary tree
    public Node root;
    public BinaryTree() {
        root = null;
    }
    //findMaximumWidth() will find out the maximum width of the given binary tree
    public int findMaximumWidth() {
        int maxWidth = 0;
        //Variable nodesInLevel keep tracks of number of nodes in each level
        int nodesInLevel = 0;
        //queue will be used to keep track of nodes of tree level-wise
        Queue<Node> queue = new LinkedList<Node>();
        //Check if root is null, then width will be 0
        if(root == null) {
            System.out.println("Tree is empty");
            return 0;
        } else {
            //Add root node to queue as it represents the first level
            queue.add(root);
            while(queue.size() != 0) {
                //Variable nodesInLevel will hold the size of queue i.e. number of elements in queue
                nodesInLevel = queue.size();
                //maxWidth will hold maximum width.
                //If nodesInLevel is greater than maxWidth then, maxWidth will hold the value of nodesInLevel
                maxWidth = Math.max(maxWidth, nodesInLevel);
                //If variable nodesInLevel contains more than one node
                //then, for each node, we'll add left and right child of the node to the queue
                while(nodesInLevel > 0) {
                    Node current = queue.remove();
                    if(current.left != null)
                        queue.add(current.left);
                    if(current.right != null)
                        queue.add(current.right);
                    nodesInLevel--;
                }
            }
        }
        return maxWidth;
    }
    public static void main(String[] args) {
        BinaryTree bt = new BinaryTree();
        //Add nodes to the binary tree
        bt.root = new Node(1);
        bt.root.left = new Node(2);
        bt.root.right = new Node(3);
        bt.root.left.left = new Node(4);
        bt.root.left.right = new Node(5);
        bt.root.right.left = new Node(6);
        bt.root.right.right = new Node(7);
        bt.root.left.left.left = new Node(8);
        //Display the maximum width of given tree
        System.out.println("Maximum width of the binary tree: " + bt.findMaximumWidth());
    }
}
```

MAXIMUM DISTANCE IN A BINARY TREE



- nodesAtMaxDistance(): find out the nodes which are present at the maximum distance
- calculateSize(): count the number of nodes present in the tree.
- convertBTtoArray(): convert the binary tree to its array representation by traversing the tree and adding elements to treeArray.
- getDistance(): calculate the distance of a given node from the root.
- LowestCommonAncestor(): find out the lowest common ancestor for two nodes.
- FindDistance(): calculate the distance between two nodes.

```java
import java.util.ArrayList;
public class MaxDistance {
    public static class Node {
        int data;
        Node left;
        Node right;
        public Node(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    //Represent the root of binary tree
    public Node root;
    int[] treeArray;
    int index = 0;
    public MaxDistance() {
        root = null;
    }
    //calculateSize() will calculate size of tree
    public int calculateSize(Node node) {
        int size = 0;
        if (node == null) return 0;
        else {
            size = calculateSize (node.left) + calculateSize (node.right) + 1;
            return size;
        }
    }
    //convertBTtoArray() will convert binary tree to its array representation
    public void convertBTtoArray(Node node) {
        if(root == null) { //Check whether tree is empty
            System.out.println("Tree is empty");
            return;
        } else {
            if(node.left != null) convertBTtoArray(node.left);
            //Adds nodes of binary tree to treeArray
            treeArray[index] = node.data;
            index++;
            if(node.right != null) convertBTtoArray(node.right);
        }
    }
    //getDistance() will find distance between root and a specific node
    public int getDistance(Node temp, int n1) {
        if (temp != null) {
            int x = 0;
            if ((temp.data == n1) || (x = getDistance(temp.left, n1)) > 0
                    || (x = getDistance(temp.right, n1)) > 0) {
                //x will store the count of number of edges between temp and node n1
                return x + 1;
            }
            return 0;
        }
        return 0;
    }
```

```java
    //lowestCommonAncestor() will find out the lowest common ancestor for nodes node1 and node2
    public Node lowestCommonAncestor(Node temp, int node1, int node2) {
        if (temp != null) {
            //If root is equal to either of node node1 or node2, return root
            if (temp.data == node1 || temp.data == node2) {
                return temp;
            }
            //Traverse through left and right subtree
            Node left = lowestCommonAncestor(temp.left, node1, node2);
            Node right = lowestCommonAncestor(temp.right, node1, node2);
            //If node temp has one node(node1 or node2) as left child and one node(node1 or node2) as right child
            //Then, return node temp  as lowest common ancestor
            if (left != null && right != null) {
                return temp;
            }
            //If nodes node1 and node2 are in left subtree
            if (left != null) {
                return left;
            }
            //If nodes node1 and node2 are in right subtree
            if (right != null) {
                return right;
            }
        }
        return null;
    }
    //findDistance() will find distance between two given nodes
    public int findDistance(int node1, int node2) {
        //Calculates distance of first node from root
        int d1 = getDistance(root, node1) - 1;
        //Calculates distance of second node from root
        int d2 = getDistance(root, node2) - 1;
        //Calculates lowest common ancestor of both the nodes
        Node ancestor = lowestCommonAncestor(root, node1, node2);
        //If lowest common ancestor is other than root then, subtract 2 * (distance of root to ancestor)
        int d3 = getDistance(root, ancestor.data) - 1;
        return (d1 + d2) - 2 * d3;
    }
```

```java
    //nodesAtMaxDistance() will display the nodes which are at maximum distance
    public void nodesAtMaxDistance(Node node) {
        int maxDistance = 0, distance = 0;
        ArrayList<Integer> arr = new ArrayList<>();
        //Initialize treeArray
        int treeSize = calculateSize(node);
        treeArray = new int[treeSize];
        //Convert binary tree to its array representation
        convertBTtoArray(node);
        //Calculates distance between all the nodes present in binary tree and stores maximum distance in variable maxDistance
        for(int i = 0; i < treeArray.length; i++) {
            for(int j = i; j < treeArray.length; j++) {
                distance = findDistance(treeArray[i], treeArray[j]);
                //If distance is greater than maxDistance then, maxDistance will hold the value of distance
                if(distance > maxDistance) {
                    maxDistance = distance;
                    arr.clear();
                    //Add nodes at position i and j to treeArray
                    arr.add(treeArray[i]);
                    arr.add(treeArray[j]);
                } else if(distance == maxDistance) {
                    //If more than one pair of nodes are at maxDistance then, add all pairs to treeArray
                    arr.add(treeArray[i]);
                    arr.add(treeArray[j]);
                }
            }
        }
        //Display all pair of nodes which are at maximum distance
        System.out.println("Nodes which are at maximum distance: ");
        for(int i = 0; i < arr.size(); i = i + 2) {
            System.out.println("( " + arr.get(i) + "," + arr.get(i + 1) + " )");
        }
    }

    public static void main(String[] args) {
        MaxDistance bt = new MaxDistance();
        //Add nodes to the binary tree
        bt.root = new Node(1);
        bt.root.left = new Node(2);
        bt.root.right = new Node(3);
        bt.root.left.left = new Node(4);
        bt.root.left.right = new Node(5);
        bt.root.right.left = new Node(6);
        bt.root.right.right = new Node(7);
        bt.root.right.right.right = new Node(8);
        bt.root.right.right.right.left = new Node(9);
        //Finds out all the pair of nodes which are at maximum distance
        bt.nodesAtMaxDistance(bt.root);
    }
}
```

THE HUFFMAN CODE

- An algorithm that uses a binary tree in a surprising way to compress data.
- Huffman code, after David Huffman who discovered it in 1952.
- Data compression is important in many situations.
  - An example is sending data over the Internet, where, especially over a dial-up connection, transmission can take a long time.

- Each character in a normal uncompressed text file is represented by one byte (ASCII code) or by two bytes (Unicode, which is designed to work for all languages.)
  - Every character requires the same number of bits.

Character	Decimal	Binary
A	65	01000000
B	66	01000001
C	67	01000010
…	…	…
X	88	01011000
Y	89	01011001
Z	90	01011010

THE HUFFMAN CODE

Frequency Table

Character	Count
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
Space	4
Linefeed	1

Huffman Code

Character	Code
A	010
E	1111
I	110
S	10
T	0110
U	01111
Y	1110
Space	00
Linefeed	01110

- Reduce the number of bits that represent the most-used characters.
  - E is the most common letter, so it is reasonable to use as few bits as possible to encode it.
  - On the other hand, Z is seldom used, so using a large number of bits is not so bad.
- Rule: No code can be the prefix of any other code.
  - For example, if E is 01, and X is 01011000, then anyone decoding 01011000 wouldn't know if the initial 01 represented an E or the beginning of an X.

CREATE A HUFFMAN TREE

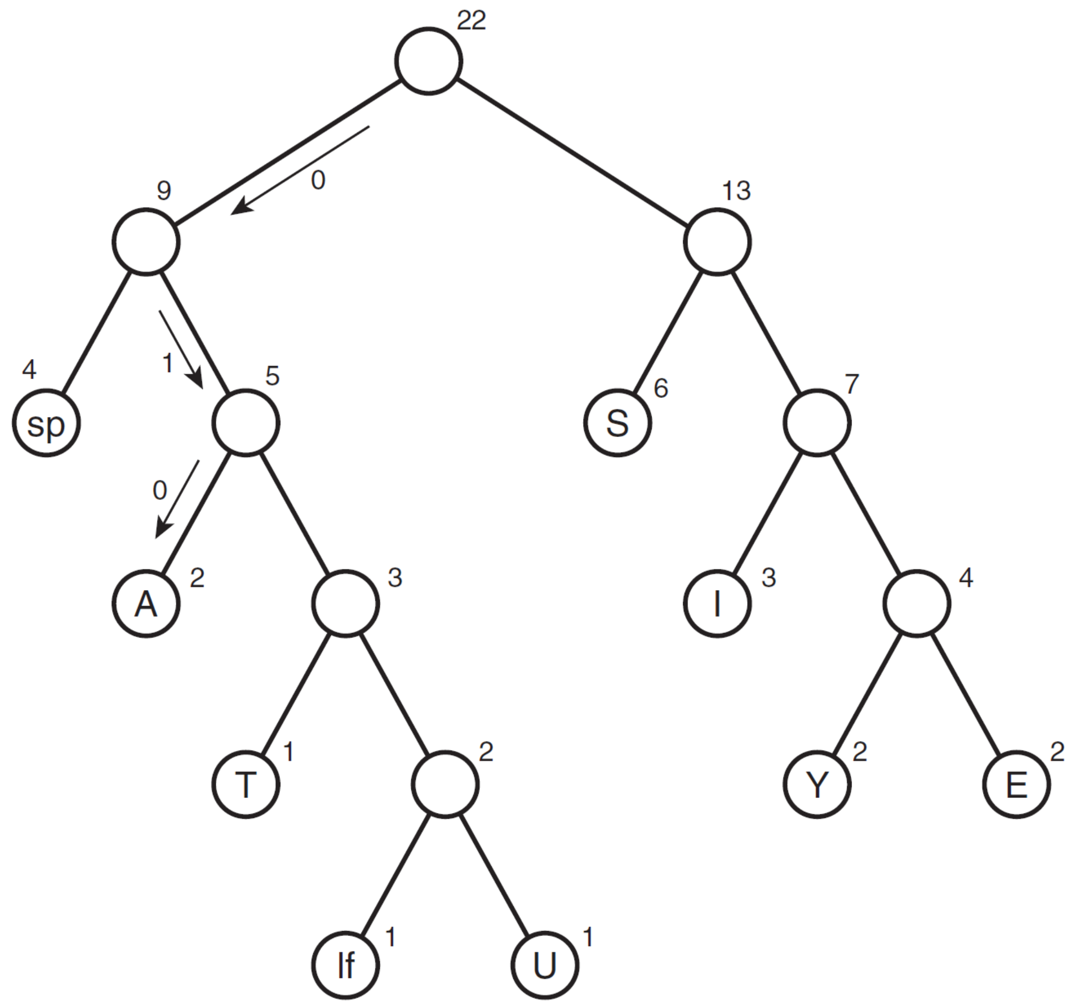
- Make a Node object for each character used in the message.
  - Each node has two data items: the character and that character's frequency in the message.
- Make a tree object for each of these nodes. The node becomes the root of the tree.
- Insert these trees in a priority queue,
  - ordered by frequency, with the smallest frequency having the highest priority.

- Keep repeating the following steps, unitil there is only one tree left in the queue.
  - Remove two trees from the priority queue, and make them into children of a new node.
    - The new node has a frequency that is the sum of the children's frequencies.
  - Insert this new three-node tree back into the priority queue.

SUSIE SAYS IT IS EASY

Frequency Table

Character	Count
A	2
E	2
I	3
S	6
T	1
U	1
Y	2
Space	4
Linefeed	1



THANKS