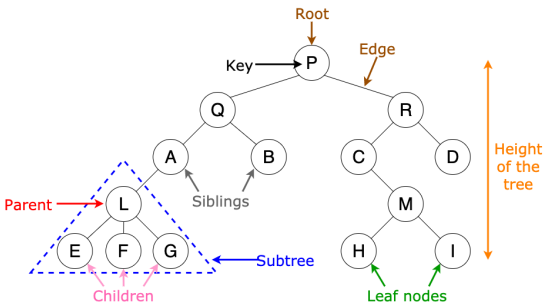
Module 3 | Trees

Importance	★★★
Syllabus	Trees, Types of trees, Binary Trees, Operations on Trees, Tree representations and tree traversals, Application of binary trees. Binary search trees, Operations and Traversals on BST, Applications. Introduction of Height balanced trees, AVL trees

Trees



`Trees:` Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, ‘a’ is a child of ‘f’, and ‘f’ is the parent of ‘a’. Finally, elements with no children are called leaves.

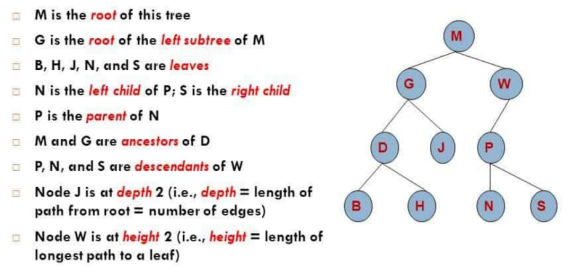
Main applications of trees include:

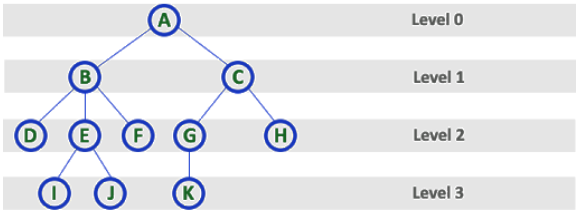
1. Manipulate hierarchical data
2. Make information easy to search (tree traversal)
3. Manipulate sorted lists of data
4. As a workflow for composting digital images for visual effects
5. Router algorithms
6. Form of a multi-stage decision-making

Terminologies of Trees

• An ancestor is any node in the path from the root to the node.
• A descendent is any node in the path below the parent node; that is, all nodes

in the paths
from a given node to a leaf are descendents of that node.
• A path is a sequence of nodes in which each node is adjacent to the next one. Every node in
the tree can be reached by following a unique path starting from the root.
• The level of a node is its distance from the root. Because the root has a zero distance from
itself, the root is at level 0. The children of the root are at level 1, their children are at level 2,
and so forth.
• Siblings are always at the same level, but all nodes in a level are not necessarily siblings.
• The height of the tree is the level of the leaf in the longest path from the root plus 1. By
definition the height of an empty tree is -1. Because the tree is drawn upside down, some
texts refer to the depth of a tree rather than its height.
• A tree may be divided into subtrees. A subtree is any connected structure below the root.
The first node in a subtree is known as the root of the subtree and is used to name the
subtree. Subtrees can also be further subdivided into subtrees.





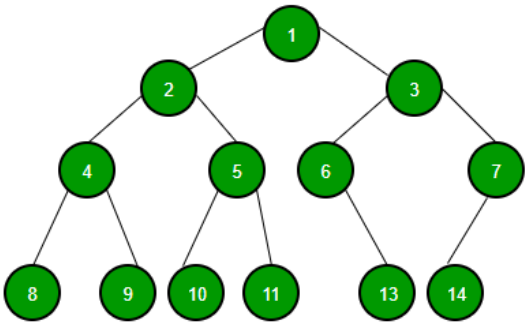
Binary Trees

A binary tree is a hierarchical data structure in which each node has at

most two children generally referred as left child and right child.

Each node contains three components:

1. Pointer to left subtree
2. Pointer to right subtree
3. Data element



Properties of Binary Tree

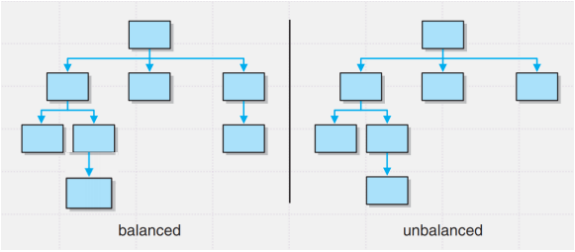
1. The maximum number of nodes at level `l` of a binary tree is 2^l
   Analysis:
   For 1st level l = 0, max no.of nodes = 1
   For 2nd level l = 1, max no.of nodes = 2
   For 3rd level l = 2, max no.of nodes = 4
   For 4th level l = 3, max no.of nodes = 8
   no.of nodes sequence is 1,2,4,8,.....
   i.e 2 powers
   Therefore, 2^l
2. The maximum number of nodes in a binary tree of tree of height `h` is $2^h - 1$
   Analysis:
   From point 1 we can get max no.of nodes in each level, adding them give us required result
   i.e 1+2+4+8+...+ 2^h (consider level 0 as height 1)
   Their sum is in geometric series
   The result would be $2^{l+1} - 1$
   i.e $2^h - 1$
3. In a binary tree with N nodes, minimum possible height or the minimum number of levels is $log_2(N+1)$
   Analysis:
   From point 2 max.nof nodes, N = $2^h - 1$
   N+1 = 2^h
   log both sides
   log(N+1) = h ⇒ h = $log_2(N+1)$
4. A binary tree with L leaves has at least $\lfloor log_2 L \rfloor + 1$ levels
   Analysis:
   Let one side of tree from root be empty

max no.of leaves in l level will be 2^{l-1}
Given L leaves
L ≤ 2^{l-1}
log on both sides
$|log_2 L| \leq l - 1$
$l \geq |log_2 L| + 1$

5. In binary tree where every node has 0 or 2 children, the number of leaf nodes is always one more that nodes with two children
   Analysis:
   max number of leaves of level l is 2^l (Point 2)
   max number of nodes in tree of height h is $2^h - 1 \Rightarrow 2^{l+1} - 1 \Rightarrow 2^l.2 - 1$
   Let,
   L = no.of leaf nodes = 2^l
   N = no.of internal nodes in tree = total nodes - leaf nodes
   N = $2^l.2 - 1 - 2^l$
   N = $2^l(2 - 1) - 1$
   N = $2^l - 1$
   N = $L - 1$
   L = N +1

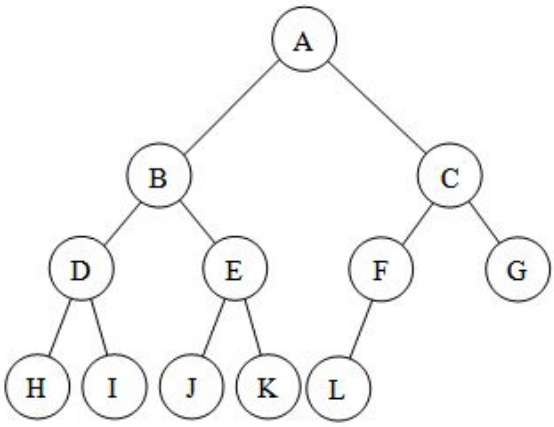
Balanced Trees

A tree is considered balanced if all the leaves of the tree are on the same level or at least within one
level of each other.

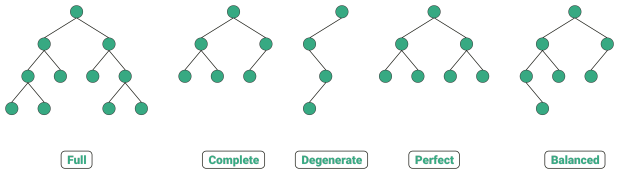


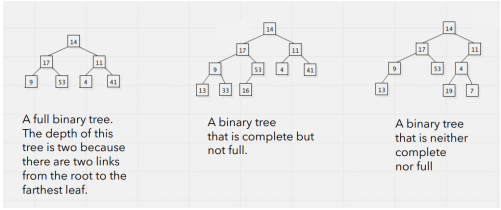
Complete Trees

- A tree is considered complete if it is balanced and all of the leaves at the bottom level are on
  the left side of the tree.
- A complete binary tree has 2k nodes at every level k except the last, where the nodes must be leftmost.
- An n-ary tree is considered full if all the leaves of the tree are at the same level and every node



either is a leaf or has exactly n children.





Node Declaration

```
class Node{
    int key;
    Node left;
    Node right;
    Node(int k){
        key = k;
        right = left = null;
    }
}
```

Height of Tree

```
//Recursive
static int height(Node root){
        if(root == null) return 0;
        return Math.max(height(root.left),height(root.right))+1;
    }
```

Traversals of Tree

1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

• Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

```
static void in_order(Node root){
        if(root == null) return;
```

• In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always
remember that every node may represent a subtree itself.
• If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

```
        in_order(root.left);
        System.out.print(root.key+" ");
        in_order(root.right);
    }
```

> GFG - in-order Traversal Practice

Pre-order Traversal

• In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

```
static void pre_order(Node root){
        if(root == null) return;
        System.out.print(root.key+" ");
        pre_order(root.left);
        pre_order(root.right);
    }
```

> GFG - Pre-order Traversal Practice

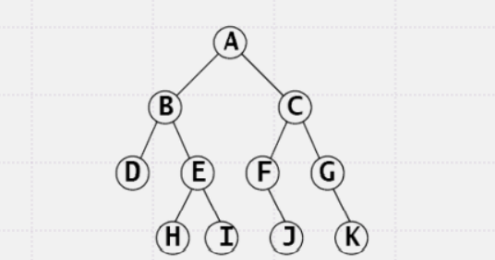
Post-order Traversal

• In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

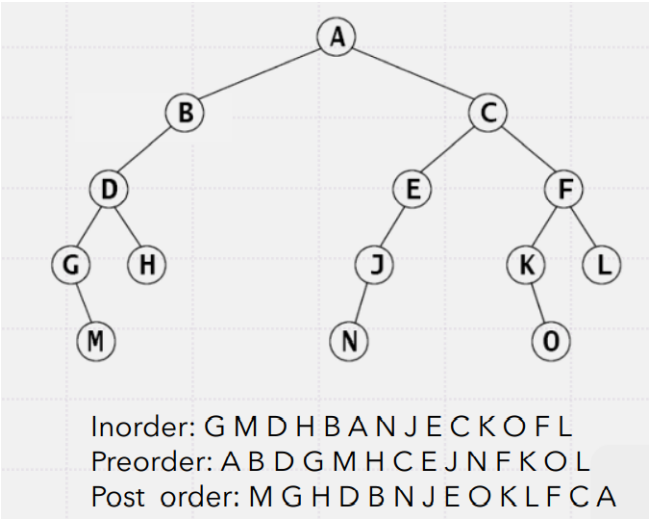
```
static void post_order(Node root){
        if(root == null) return;
        post_order(root.left);
        post_order(root.right);
        System.out.print(root.key+" ");
    }
```

> GFG - Post-order Traversal Practice

> Visualization of Traversals

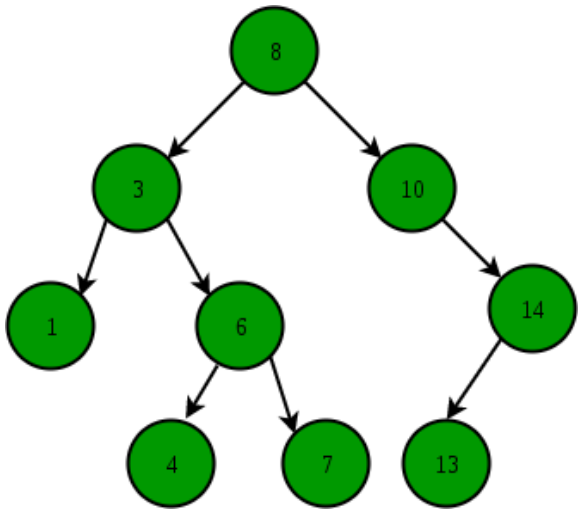


Inorder: D B H E I A F J C G K
Preorder: A B D E H I C F J G K
Post order: D H I E B J F K G C A



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

Binary Search Trees



Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Operations on BSTs

Search

```
public static boolean search(Node root, int x){
        if(root==null)
            return false;
        if(root.key==x)
            return true;
        else if(root.key>x){
            return search(root.left,x);
        }else{
            return search(root.right,x);
        }
    }
```

```
public static boolean search(Node root, int x
        while (root != null) {
            if (root.key == x)
                return true;
            else if (root.key < x)
                root = root.right;
            else
                root = root.left;
        }
        return false;
    }
```

Insert

```
public static Node insert(Node root, int x) {
        if (root == null)
            return new Node(x);
        else if (root.key < x)
            root.right = insert(root.right, x);
        else if (root.key > x)
            root.left = insert(root.left, x);
        return root;
    }
```

```
public static Node insert(Node root, int x) {
        Node temp = new Node(x);
        Node parent = null, curr = root;
        while (curr != null) {
            parent = curr;
            if (curr.key > x)
                curr = curr.left;
            else if (curr.key < x)
                curr = curr.right;
            else
                return root;
        }
        if (parent == null)
            return temp;
        if (parent.key > x)
            parent.left = temp;
        else
            parent.right = temp;
        return root;
    }
```

Delete

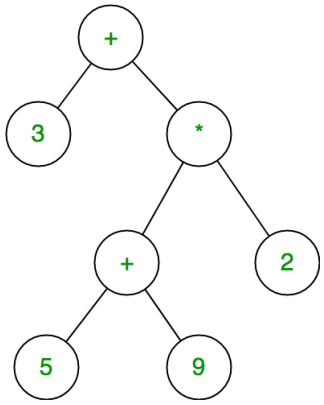
```
public static Node delNode(Node root, int x) {
        if (root == null)
            return root;
        if (root.key > x)
            root.left = delNode(root.left, x);
        else if (root.key < x)
            root.right = delNode(root.right, x);
        else {
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            } else {
                Node succ = getSuccessor(root);
                root.key = succ.key;
                root.right = delNode(root.right, succ.key);
            }
        }
        return root;
    }
```

Expression Tree

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf

node corresponds to the operand.

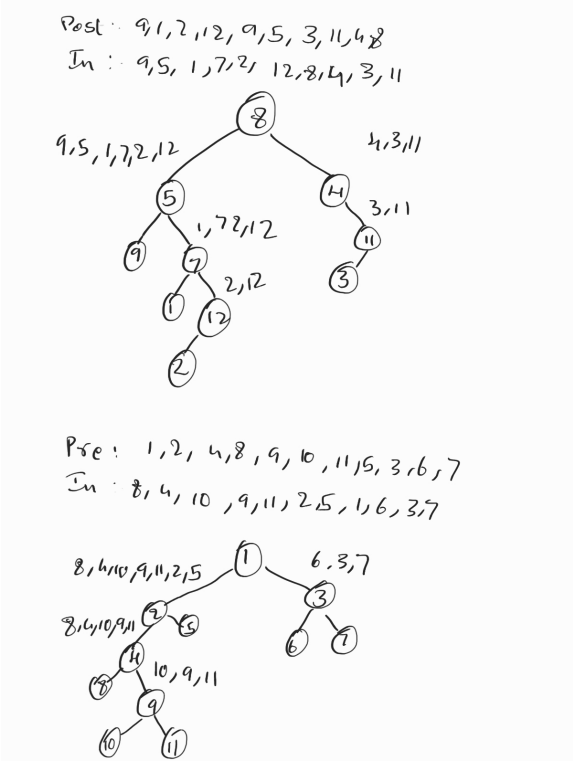
```
Node constructTree(char postfix[]) {
        Stack<Node> st = new Stack<Node>();
        Node t, t1, t2;

        // Traverse through every character of
        // input expression
        for (int i = 0; i < postfix.length; i++) {

            // If operand, simply push into stack
            if (!isOperator(postfix[i])) {
                t = new Node(postfix[i]);
                st.push(t);
            } else // operator
            {
                t = new Node(postfix[i]);

                // Pop two top nodes
                // Store top
                t1 = st.pop();      // Remove top
                t2 = st.pop();

                //  make them children
                t.right = t1;
                t.left = t2;

                // System.out.println(t1 + "" + t2);
                // Add this subexpression to stack
                st.push(t);
            }
        }

        //  only element will be root of expression
        // tree
        t = st.peek();
        st.pop();

        return t;
    }
```



3 + ((5+9)*2)

Evaluating an Expression from Its Postfix Representation
To evaluate an expression represented in postfix, scan the representation from left to right:
1. Create a stack for operands.
2. Repeat steps 3 9 until the end of representation is reached.
3. Read the next token t from the representation.
4. If it is an operand, push its value onto the stack.
5. Otherwise, do steps 6 9:
6. Pop a from the stack.
7. Pop b from the stack.
8. Evaluate c = a t b.
9. Push c onto the stack.
10. Return the top element on the stack

Binary tree form given pre order or post order and in-order

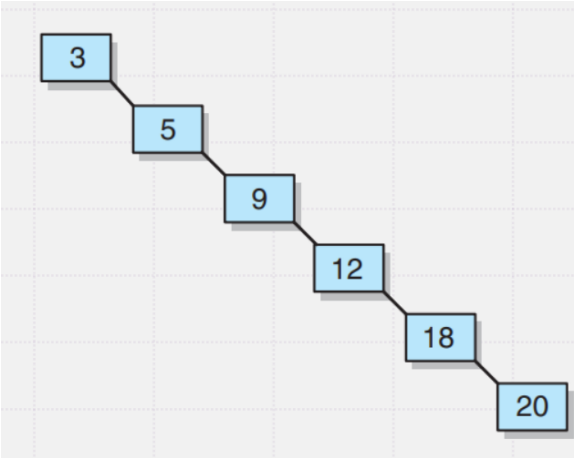


AVL Trees aka Self-Balancing Trees

Self Balancing Trees

Our goal instead is to keep the maximum path length in the tree at or near log2n. There are a variety of algorithms available for balancing or maintaining balance in a tree. There are brute force methods, which are not elegant or efficient, but get the job done. For example, we could write an in-order traversal of the tree to an array and then use a recursive method (much like binary search) to insert the middle element of the array as the root, and then build balanced left and right subtrees.

Although such an approach would work, there are more elegant



solutions, such as AVL trees and red/black trees, which we examine later.

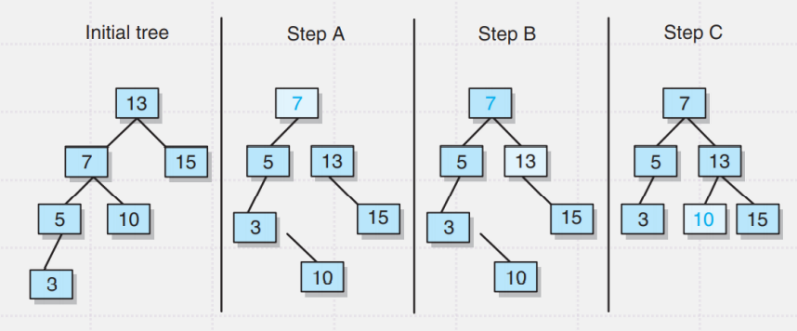
Right Rotations

The maximum path length in this tree is 3, and
the minimum path length is 1. With only 6 elements in the tree, the maximum path
length should be $log_2 6$, or 2. To get this tree into balance, we need to

✓ Make the left child element of the root the new root element.

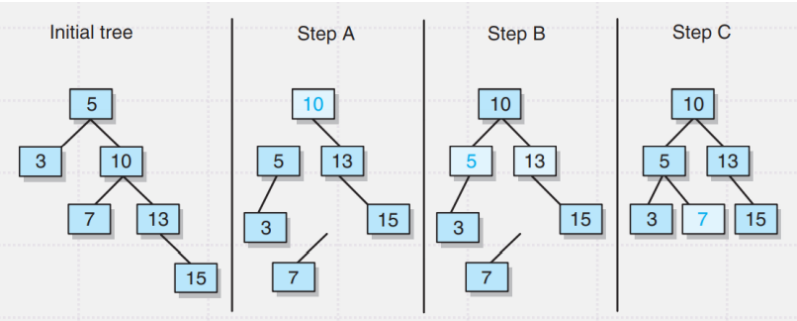
✓ Make the former root element the right
child element of the new root.

✓ Make the right child of what was the left child of the former root the new left child
of the former root.



Left Rotations

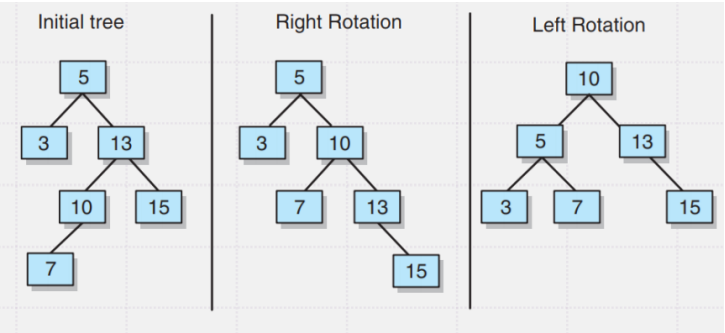
Again, the maximum path length in this tree is 3 and the minimum path length is 1.
However this time the larger path length is in the right subtree of the right child of the root.



✓ To get this tree into balance, we need to Make the right child element of the root the new root element.
✓ Make the former root element the left child element of the new root.
✓ Make the left child of what was the right child of the former root the new right child of the former root.
This left rotation is often referred to as a left rotation of the right child around the parent.

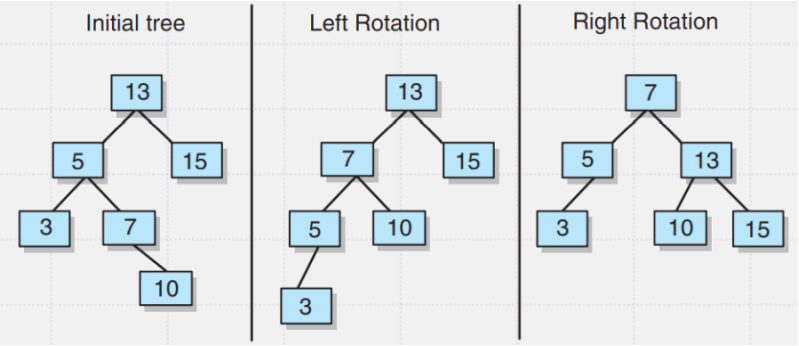
Right-Left Rotations

Unfortunately, not all imbalances can be solved by single rotations. If the imbalance is caused by a long path length in the left subtree of the right child of the root, we must first perform a right rotation of the left child of the right child of the root around the right child of the root, and then perform a left rotation of the resulting right child of the root around the root



Left-Right Rotations

Similarly, if the imbalance is caused by a long path length in the right subtree of the left child of the root, we must first perform a left rotation of the right child of the left child of the root around the left child of the root, and then perform a right rotation of the resulting left child of the root around the root.

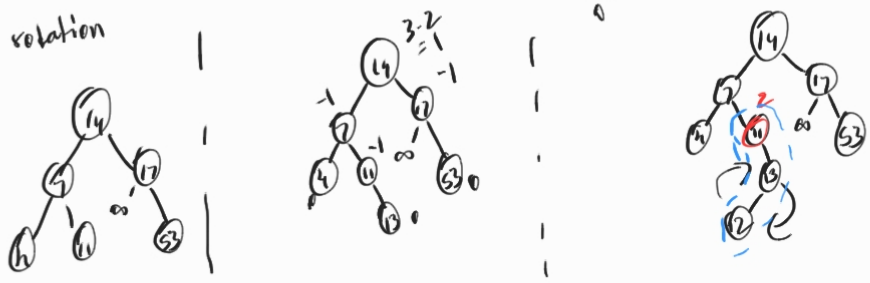


Construct AVL

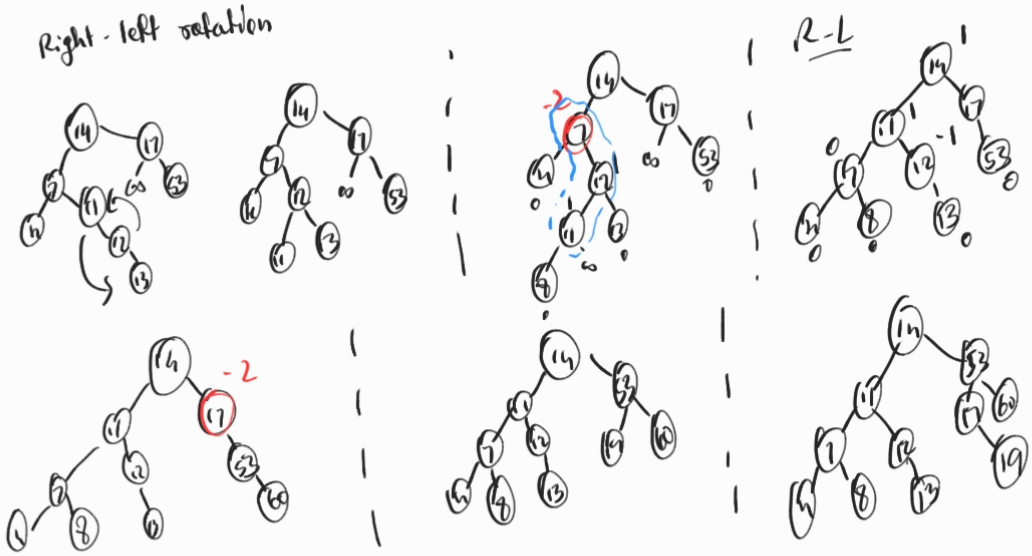
Q Construct an AVL tree from following array of nos

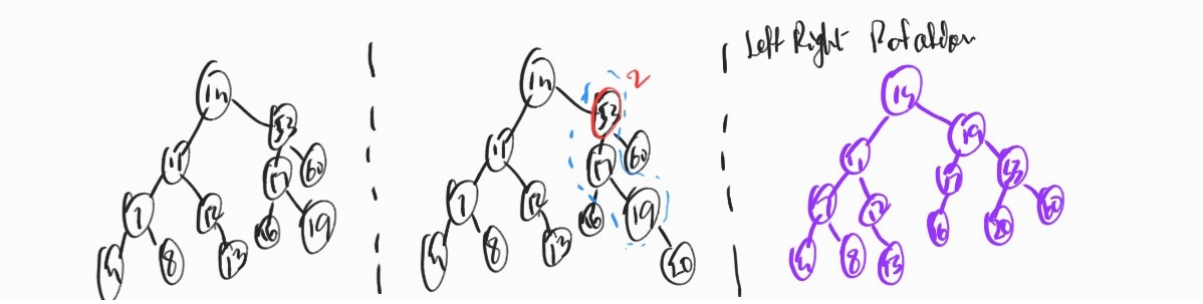
14, 17, 4, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

Right rotation



Right - left rotation





> https://www.geeksforgeeks.org/binary-tree-data-structure/
> Practice Problems - GFG
> Quizzes - GFG