BINARY SEARCH TREE - I

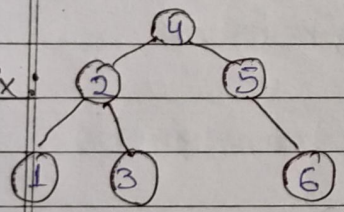
* In BT, for Searching → Time Complexity is O(n). (n = no. of nodes in BT)
* In BST, for Searching → Time Complexity is O(h). (h = height of the Tree)

∴ BST is fast.

* BST is a type of BT, it acquires all properties of BT with some additional properties that are :-
  a) Left Subtrees < Root
  b) Right Subtrees > Root
  c) Left & Right Subtrees are also BST with no Duplicates.

* Inorder Traversal (left → root → right) of BST gives a sorted sequence.

Ex:

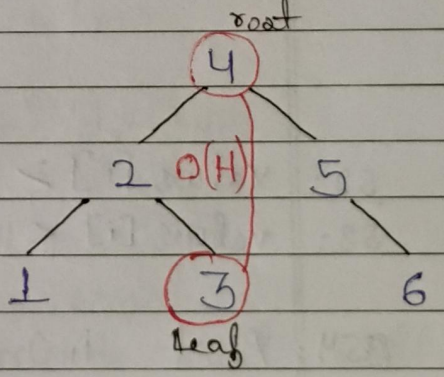


→ In Order Traversal: 1 2 3 4 5 6 (Sorted)

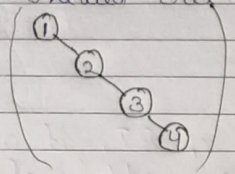
★ BST SEARCH – BST makes search efficient

1. if key == root
   return root
2. if key > root.data
   ~~return~~ Go right
3. if key < root.data
   Go left

key = 3.



* Best Case → BST → $O(H)$
* Average Case → Balanced BST → $\log N << N$
* Worst Case → Skewed Tree → $O(H) = O(N)$

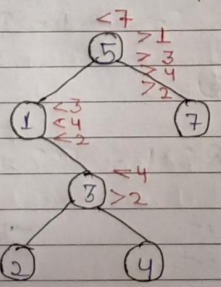


* Most problems will be solved using recursion i.e., by dividing into subproblems & making recursive calls on subtrees.

★ BUILD A BST

* values [] = {5, 1, 3, 4, 2, 7}

S1: Root = null

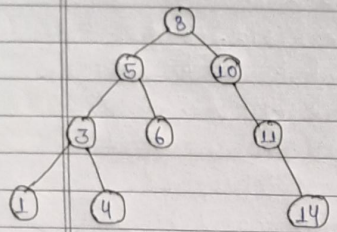


S2: values [i] ≥ root → right subtree
S3: values [i] < root → left subtree

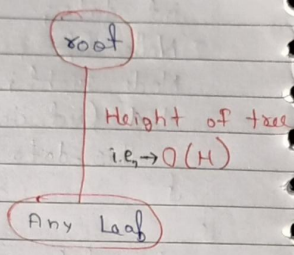
S4: Print inOrder traversal → 1 2 3 4 5 7

⇒
```
static class Node {
  int data;
  Node left;
  Node right;
  Node (int data) {
    this.data = data;
  }
}
public static Node insert (Node root, int val) {
  if (root == null) {
    root = new Node (val);
    return root;
  }
  if (root.data > val) {
    //left subtree
    root.left = insert (root.left, val);
  } else { //right subtree
    root.right = insert (root.right, val);
  }
  return root;
}
public static void inOrder (Node root) {
  if (root == null)
    return;
  inOrder (root.left);
  System.out.print (root.data + " ");
  inOrder (root.right);
}
public static void main (String args[]) {
  int val[] = {5,1,3,4,2,7};
  Node root = null;
  for (int i=0; i<val.length; i++)
    root = insert (root, val[i]);
  inOrder (root);
```

★ SEARCH IN BST

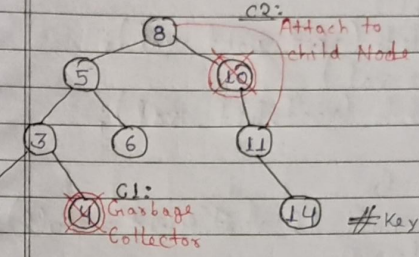


* key = 1




```
public static boolean search (Node root, int key) {
  if (root == null) {
    return false;
  if (root.data == key)
    return true;
  if (root.data > key)
    return search (root.left, key);
  else
    return search (root.right, key);
}
public static void main (String[] args) {
  int val [] = {5, 1, 3, 4, 2, 7}
  int root = null;
  for (int i=0; i< val.length; i++) {
    root = insert (root, val [i]);
  }
  inOrder (root);
  Syso ();
  if (search (root, 7)) {
    Syso ("found");
  } else {
    Syso ("not found");
  }
}
```


★ DELETE A NODE



* Cases to Delete
  C1: No child (Leaf Nodes)
  C2: One Child
  C3: Two Children

Key → Search → Which case → Delete

Case 1: No Child (leaf Nodes) – 1, 4, 6, 14
→ Delete node & return Null to parent.

Case 2: One Child – 10, 11
→ Delete node & replace with child node.

Case 3: Two Children
→ Replace value with inOrder Successor (IS)
→ Delete the node for inOrder Successor

Ex: 1 3 4 5 6 8 10 11 14
IS

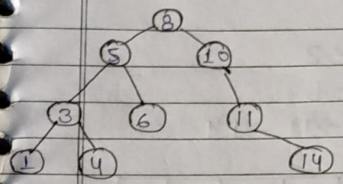
⇒ 1 3 4 6 8 10 11 14

→ InOrder Successor ⟶ ~~Left~~ Left most node in right subtree
in BST ↓
min value in right subtree.

InOrder successor always has 0 or 1 child.

```
⇒ public static Node delete (Node root, int val) {
    if (root.data < val)
        root.right = delete (root.right, val);
    else if (root.data > val)
        root.left = delete (root.left, val);
    else {
        //Case 1 - Leaf node
        if (root.left == null && root.right == null) {
            return null;
        }
        //Case 2 - single child
        if (root.left == null)        if (root.right == null)
            return root.right;            return root.left;
        //Case 3 - Both Children
        Node IS = findInorderSuccessor (root.right);
        root.data = IS.data;
        root.right = delete (root.right, IS.data);
    }
    return root;
}
public static Node findInorderSuccessor (Node root) {
    while (root.left != null) {
        root = root.left;
    }
    return root;
}
public static void main (String[] args)
    int val[] = {8, 5, 3, 1, 4, 6, 10, 11, 14};
    Node root = null;
    for (int i=0; i<val.length; i++)
        root = insert (root, values[i]);
    inorder (root); SYSO(); root = delete (root, 1);
    SYSO(); inOrder (root);
```


★ PRINT IN RANGE



* Input : k1 = 5 & k2 = 12
* Output : 5, 6, 8, 10, 11

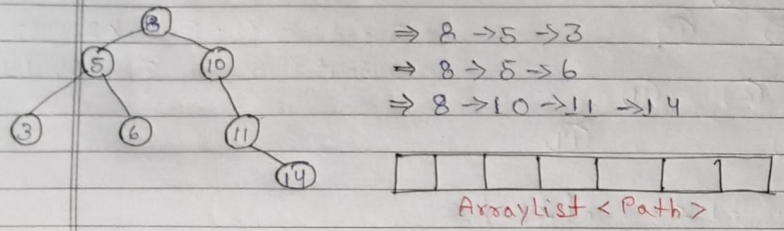
Case 1: k1 <= root <= k2
- Left Subtree
- Right Subtree

Case 2: root > k2
- Right Subtree

Case 3: root < k1
- Left Subtree

```
⇒ public static void printInRange (Node root, int k1, int k2) {
    if (root == null) {
        return;
    }
    if (root.data >= k1 && root.data <= k2) {
        printInRange (root.left, k1, k2);
        System.out.print (root.data + " ");
        printInRange (root.right, k1, k2);
    }
    else if (root.data < k2) {
        printInRange (root.right, k1, k2);
    }
    else {
        printInRange (root.left, k1, k2);
    }
}
```

★ ROOT TO LEAF PATH



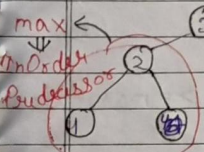
S1: Add (Node) to Path
S2: Left Subtree ⟶ Print Path
S3: Bactracking
S4: Right Subtree ⟶ Print Path
S5: At last, Path will be Empty.

⇒
```
public static void printRoot2Leaf (Node root, ArrayList<Integer> path){
    if (root == null)
        return;
    path.add (root.data);
    if (root.left == null && root.right == null){
        printPath (path);
    printRoot2Leaf (root.left, path);
    printRoot2Leaf (root.right, path);
    path.remove (path.size() - 1);
}
public static void printPath (ArrayList<Integer> path){
    for (int i=0; i<path.size(); i++)
        System.out.print (path.get(i) + "->");
    Syso ("Null");
}
```

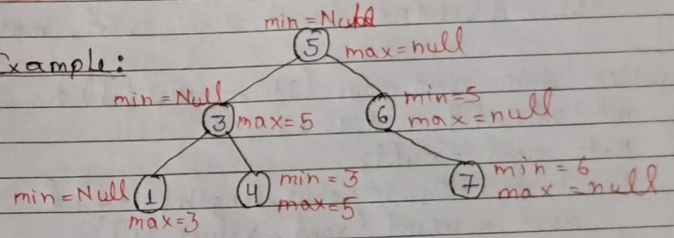
Input: printRoot2Leaf (root, new ArrayList<>());

★ VALIDATE BST - Check the given BST is a valid BST or not.

* Approach: Check if max value in left subtree < node & min value in right subtree > node.



Example:



* Left Subtree ⟶ min = min
  max = root/parent

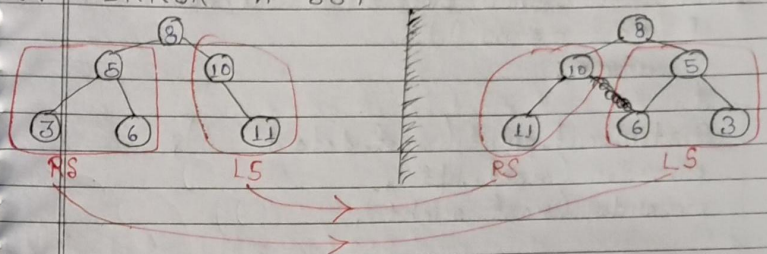
* Right Subtree ⟶ min = root/parent
  max = max

* To check, we know that in BST inOrder traversal is sorted.

=# In this code, we will not write another function to check

```
⇒ public static boolean isValidBST(Node root, Node min, Node max){
    if (root == null) {
        return true;
    }
    if (min != null && root.data <= min.data) {
        return false;
    }
    else if (max != null && root.data >= max.data){
        return false;
    }
    return isValidBST(root.left, min, root)
        && isValidBST(root.right, root, max);
}
public static void main (String args[]) {
    int values[] = {1, 1, 1};
    Node root = null;
    for (int i=0; i<values.length; i++) {
        root = insert(root, values[i]);
    }
    inorder(root);
    Sysо();
    //Checking Valid BST
    if (isValidBST(root, null, null)) {
        SysO("Valid");
    }
    else {
        Syso("Not Valid");
    }
}
```


★ MIRROR A BST



*Swapping leftSubtree with rightSubtree.
//Time Complexity - O(n)

```
⇒ static class Node {
    int data;
    Node left;
    Node right;
    public Node(int data) {
        this.data = data;
        this.left = this.right = null;
    }
}
public static Node createMirror(Node root) {
    if (root == null) {
        return null;
    }
    Node leftMirror = createMirror(root.left);
    Node rightMirror = createMirror(root.right);
    root.left = rightMirror;
    root.right = leftMirror;

    return root;
}
```

//NEXT PAGE

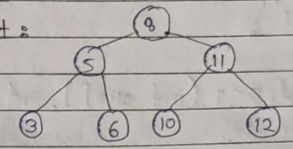
```
public static void preorder (Node root) {
    if (root == null) {
        return;
    }
    System.out.print (root.data + " ");
    preorder (root.left);
    preorder (root.right);
}

public static void main (String args []) {
    Node root = new Node (8);
    root.left = new Node (5);
    root.right = new Node (10);
    root.left.left = new Node (3);
    root.left.right = new Node (6);
    root.right.right = new Node (11);
    root = createMirror (root);
    preorder (root);
}
```


BINARY SEARCH TREES - II

★ SORTED ARRAY TO BALANCED BST

* Input : arr = {3, 5, 6, 8, 10, 11, 12}
* Output :



We have to make Balanced BST by array (min possible height)

* Approach : BST inorder Sequence ⟶ SORTED

⇒ arr [] = {3, 5, 6, 8, 10, 11, 12}

sp=0 mid ep=6

LS ← ⟶ RS

mid = (starting point + ending point) / 2

⇒ 3, 5, 6

sp=0 mid ep=2

LS ← ⟶ RS

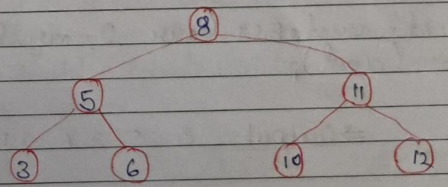
3 (mid) 6 (mid)

⇒ 10, 11, 12

sp=0 mid ep=2

LS ← ⟶ RS

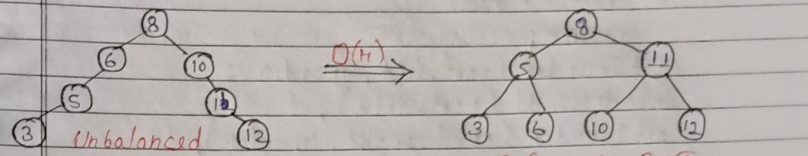
10 (mid) 12 (mid)



```
public static void preorder (Node root) {
    if (root == null)
        return;
    System.out.print (root.data + " ");
    preorder (root.left);
    preorder (root.right);
}
public static Node createBST (int arr[], int st, int end){
    if (st > end)
        return null;
    int mid = (st + end) / 2;
    Node root = new Node (arr[mid]);
    root.left = createBST (arr, st, mid-1);
    root.right = createBST (arr, mid+1, end);
    return root;
}
public static void main (String args[]) {
    int arr[] = {3, 5, 6, 8, 10, 11, 12};
    /*        8
            /   \
           5     11
          / \   /  \
         3   6 10   12
        expected BST
    */
    Node root = createBST (arr, 0, arr.length-1);
    preorder (root);
}
```

⇒ Output: 8 5 3 6 11 10 12

★ CONVERT BST TO BALANCED BST



* Unbalanced BST → InOrder Sequence → Sorted ArrayList → Balanced BST

```
⇒ public static Node balanceBST (Node root) {
    //InOrder Sequence
    ArrayList<Integer> inorder = new ArrayList<>();
    getInOrder (root, inorder);
    //sorted inorder → balanced BST
    root = createBST (inorder, 0, inorder.size() - 1);
    return root;
}
public static Node createBST (ArrayList<Integer> inorder,
                              int st, int end) {
    if (st > end)
        return null;
    int mid = (st + end) / 2;
    Node root = new Node (inorder.get (mid));
    root.left = createBST (inorder, st, mid-1);
    root.right = createBST (inorder, mid+1, end);
    return root;
}
```

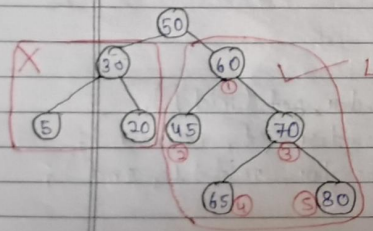
//NEXT PAGE

```
public static void getInorder (Node root, ArrayList<Integer> inorder)
    if (root == null)
        return;
    getInorder (root.left, inorder);
    inorder.add (root.data);
    getInorder (root.right, inorder);
}

public static void main (String args[]) {
    Node root = new Node (8);
    root.left = new Node (6);
    root.left.left = new Node (5);
    root.left.left.left = new Node (3);
    root.right = new Node (10);
    root.right.right = new Node (11);
    root.right.right.right = new Node (12);

    root = balanceBST (root);
    preorder (root);
}
```

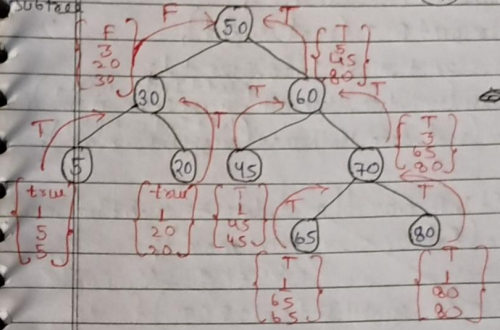
★ SIZE OF LARGEST BST IN BT



∴ ⇒ Size = 5

* Approach : We need 4 informations from the subtrees : ① isBST (boolean) ② size ③ min ④ max — Info Class

Left ⇒ isBST (False) Subtree



Right Subtree ⇒ is BST (True) size = 5

⇒

```
static class Info {
    boolean isBST;
    int size;
    int min;
    int max;
    public Info (boolean isBST, int size, int min, int max) {
        this.isBST = isBST;
        this.size = size;
        this.min = min;
        this.max = max;
    }
}

public static int maxBST = 0;
    // Time Complexity - O(n)
    // Space Complexity - O(1)
```

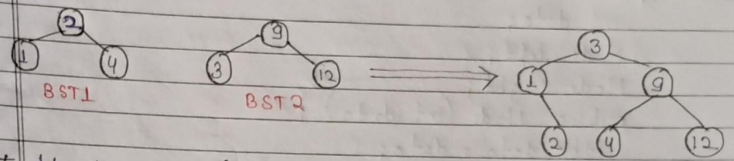
//NEXT PAGE

```
public static Info largestBST (Node root) {
    if (root == null) {
        return new Info (true, 0, Integer.MAX_VALUE, Integer.MIN_VALUE);
    }
    Info leftInfo = largestBST (root.left);
    Info rightInfo = largestBST (root.right);
    int size = leftInfo.size + rightInfo.size + 1;
    int min = Math.min (root.data, Math.min (leftInfo.min, rightInfo.min));
    int max = Math.max (root.data, Math.max (leftInfo.max, rightInfo.max));

    if (root.data <= leftInfo.max || root.data >= rightInfo.min) {
        return new Info (false, size, min, max);
    }
    if (leftInfo.isBST && rightInfo.isBST) {
        maxBST = Math.max (maxBST, size);
        return new Info (true, size, min, max);
    }
    return new Info (false, size, min, max);
}
public static void main (String args []) {
    Node root = new Node (50);
    root.left = new Node (30);
    root.left.left = new Node (5);
    root.left.right = new Node (20);
    root.right = new Node (60);
    root.right.left = new Node (45);
    root.right.right = new Node (70);
    root.right.right.left = new Node (65);
    root.right.right.right = new Node (80);
    Info info = largestBST (root);
    Sysо ("Largest BST size : " + maxBST);
}
```

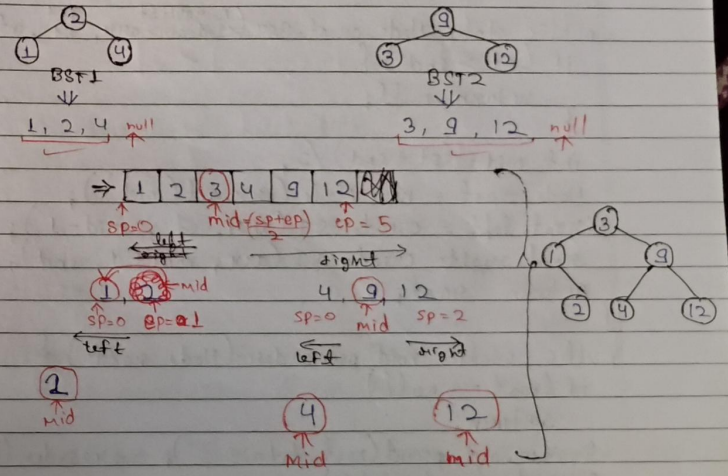
// wrong see code in PC.

★ MERGE TWO BSTs



We know that, if we have Balanced BST as sorted Array or ArrayList then it is converted in as Balanced BST.

S1: BST 1 → inOrder Sequence 1
S2: BST2 → inorder sequence 2
S3: * Sorted1[] = inOrder Sequence 1
    * Sorted2[] = inOrder Sequence 2
S4: Merge Sorted 1[] & Sorted 2[] → Final sorted array
S5: Final Sorted Array ⟹ Balanced BST



// O(n+m) ⟶ Linear TC

⇒
```
public class Node {
  int data;
  Node left;
  Node right;
  public Node (int data) {
    this.data = data;
    this.left = this.right = null;
  }
}

public static void getInorder (Node root, ArrayList<Integer> arr) {
  if (root == null) {
    return;
  }
  getInorder (root.left, arr);
  arr.add (root.data);
  getInorder (root.right, arr);
}

public static Node createBST (ArrayList<Integer> arr, int st, int end) {
  if (st > end) {
    return null;
  }
  int mid = (st + end) / 2;
  Node root = new Node (arr.get (mid));
  root.left = createBST (arr, st, mid-1);
  root.right = createBST (arr, mid+1, end);
  return root;
}

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



```
public static Node mergeBSTs (Node root1, Node root2) {
  //Step-1
  ArrayList<Integer> arr1 = new ArrayList<>();
  getInorder (root1, arr1);
  //Step-2
  ArrayList<Integer> arr2 = new ArrayList<>();
  getInorder (root2, arr2);
  //merge
  int i=0, j=0;
  ArrayList<Integer> finalArr = new ArrayList<>();
  while (i<arr1.size() && j<arr2.size()) {
    if (arr1.get(i) <= arr2.get(j)) {
      finalArr.add (arr1.get(i));
      i++;
    } else {
      finalArr.add (arr2.get(j));
      j++;
    }
  }
  while (i<arr1.size()) {
    finalArr.add (arr1.get(i));
    i++;
  }
  while (j<arr2.size()) {
    finalArr.add (arr2.get(j));
    j++;
  }
  //sorted AL -> Balanced BST
  return createBST (finalArr, 0, finalArr.size()-1);
}
```

* Input & Node root = mergeBSTs (root1, root2);
(main code) preorder (root);

★ AVL TREES

* AVL is a self-balancing tree.
  → automatic AVL हर operation के बाद सभी को check करके balance कर लेता है।

* Why we need balancing BST?
  Operations on BST
  - → Best Case → Balance Tree → O(H)
  - → Average case → O(log(n))
  - → Worst Case → Skewed Tree → O(n)

→ For Best Time complexity we need Balancing

* Property → Balance Factor : It is a property in AVL tree to check balancing.

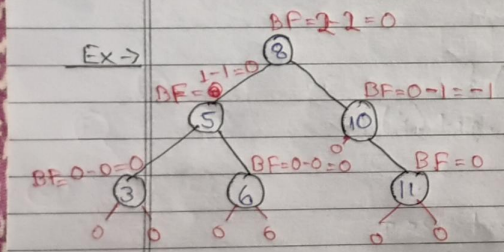
" Balance Factor = height(left subtree) − height(right subtree)

$$\Rightarrow \boxed{BF = |HL - HR| < 2}$$

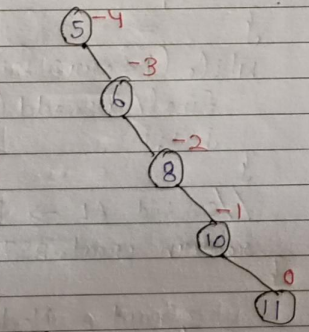
or

$$\Rightarrow \boxed{BF = |HL - HR| \text{ always equal to } \{-1, 0, 1\}}$$

Ex →

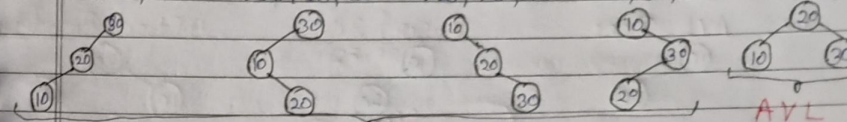


∴ AVL Tree



∴ Not Valid AVL

→ 30, 20, 10 → 30, 10, 20 → 10, 20, 30 → 10, 30, 20 → 20, 10, 30



∴ for {10, 20, 30}, Total BSTs = 6

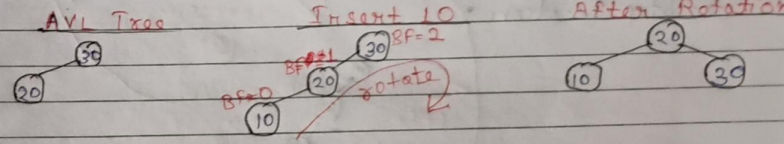
$$\Rightarrow \boxed{\text{for } n, \text{ Total BSTs} = n!}$$

* To Convert NOT AVL to AVL we use Rotations.

* There are 4 cases of rotation in an AVL Tree.

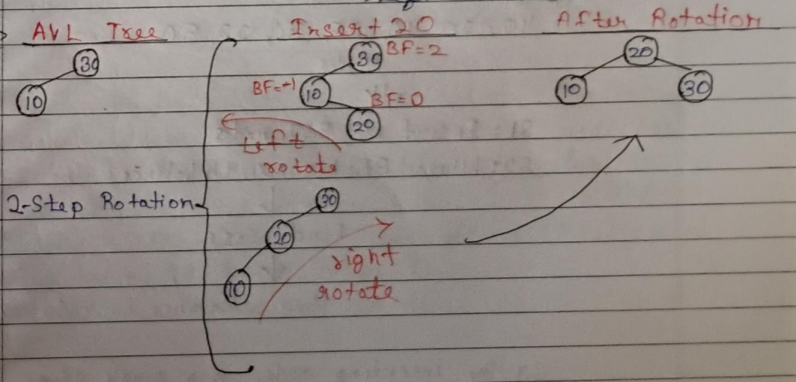
i) Left Left (LL) Case → Right Rotate

Ex →



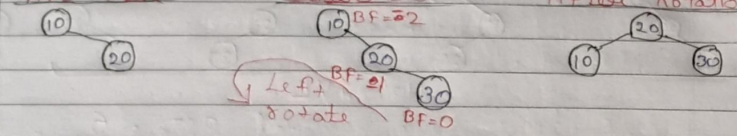
ii) Left Right (LR) Case → Right Rotate
→ Left Rotate

Ex → AVL Tree



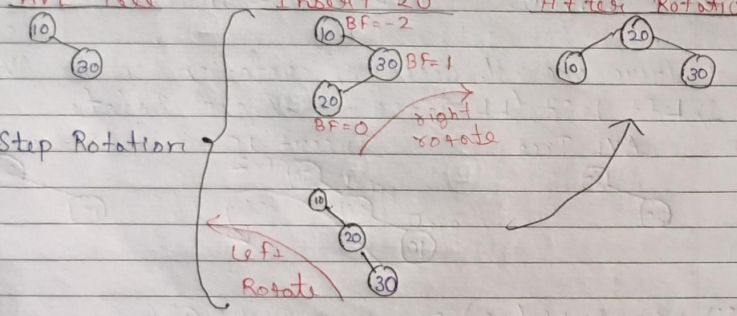
(iii) Right Right (RR) Case → Left Rotate

Ex→ AVL Tree — Insert 30 — After Rotation



iv) Right Left (LR) Case → Right Rotation
→ Left Rotation

Ex→ AVL Tree — Insert 20 — After Rotation



* Creating an AVL Tree

40, 20, 10, 25, 30, 22, 50.
Nodes

↓

S1: Insert as BST nodes

S2: Where BF become Unbalance

↓

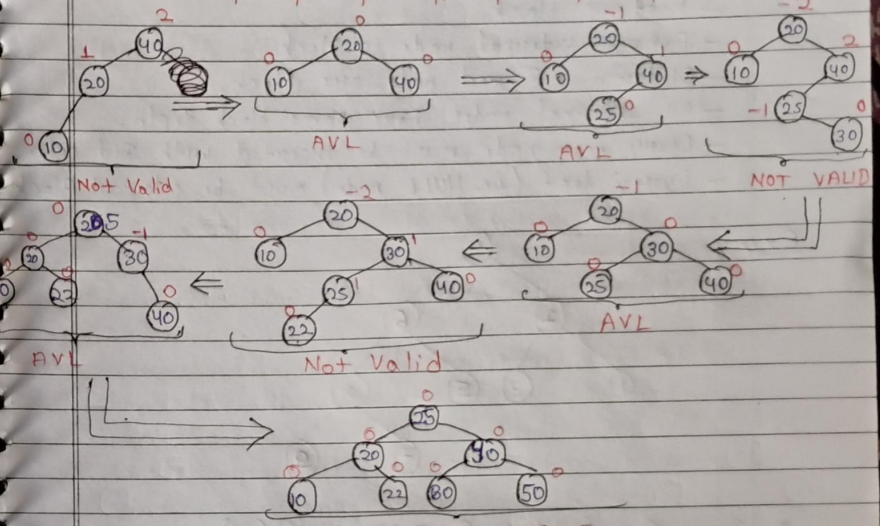
Find Case

↓

then Balance by rotation

S3: By inserting node, in a single steps we have to check balancing.

Ex→ 40, 20, 10, 25, 30, 22, 50



#NOTE: Code of AVL Tree is very big so I have not written in copy you can access using pdfs of vs code.

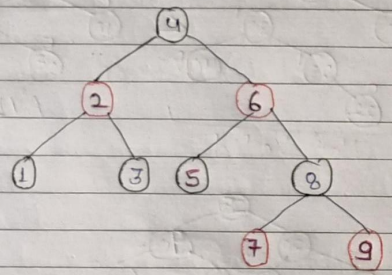
★ RED BLACK TREES – It is a self-balancing binary search tree in which each node has an extra bit, which represents its color. (RED, BLACK)

* Every RBT is a BST but all BST is need not to be RBT.

#NOTE: RBT CODE IS TOO BIG SO FOR CODE → ACCESS PDF.

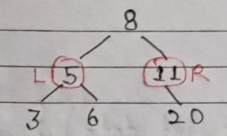
* There are few properties associated with RBT:
  - Root is black.
  - Every external node is black.
  - Children of red node are black.
  - All external nodes have same black depth.
  - Every new node must be inserted with Red color.
  - Every leaf (i.e. NULL node) must be coloured Black.

Ex⇒



* Time Complexity of BST → O(H)
  ↳ Height of Tree
* Height of Red-Black Tree is always O(Log n)
* In RBT, we use two steps to do the balancing:
  i) Recolouring
  ii) Rotation
* Recolouring is the change in color of the node i.e., if it is red then change it to black & vice versa.
* It must be ~~node~~ noted that the color of the NULL node is always black.

Q1. Range Sum of BST - We have a BST consisting of N nodes & two positive integers L & R, the task is to find the sum of values of all the nodes that lie in the range [L, R].



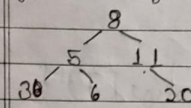
⇒ 14

Sol.

```
static class Node {                    // Time Complexity - O(n)
  int val;                             // Space Complexity - O(n)
  Node left, right;
};
static Node newNode (int item) {
  Node temp = new Node ();
  temp.val = item;
  temp.left = temp.right = null;
  return temp;
}
static int sum = 0;
static int rangeSumBST (Node root, int low, int high) {
  if (root == null)
    return 0;
  Queue <Node> q = new LinkedList <Node> ();
  q.add (root);
  While (q.isEmpty() == false) {
    Node curr = q.peek();
    q.remove();
    if (curr.val >= low && curr.val <= high) {
      ~~curr.val <=~~
      sum += curr.val;
    }
```

```
        if (curr.left != null && curr.val > low)
            q.add(curr.left);
        if (curr.right != null && curr.val < high)
            q.add(curr.right);
    }
    return sum;
}
static Node insert(Node node, int data) {
    if (node == null) {
        return newNode(data);
    }
    if (data <= node.val) {
        node.left = insert(node.left, data);
    } else {
        node.right = insert(node.right, data);
    }
    return node;
}
public static void main(String[] args) {
    Node root = null;
    root = insert(root, 10);
    insert(root, 5);
    insert(root, 15);
    insert(root, 3);
    insert(root, 7);
    insert(root, 18);
    int L = 7, R = 15;
    System.out.print(rangeSumBST(root, L, R));
}
```


Q2. Find the closest element in BST – We have a BST & a target node K. The task is to find the node with minimum absolute difference with given target value K.



Sample Input: 5
Sample output: 5 (difference is 0)

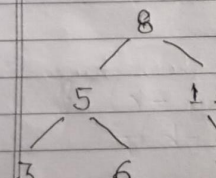
Sample Input: 19
Sample Output: 20 (difference is 0)

Sol.

```
static int min_diff, min_diff_key;
static class Node {
    int key;                    // Time Complexity - O(H)
    Node left, right;           // Space Complexity - O(1)
};
static Node newnode(int key) {
    Node node = new Node();
    node.key = key;
    node.left = node.right = null;
    return (node);
}
static void maxDiffUtil(Node ptr, int k) {
    if (ptr == null)
        return;
    if (ptr.key == k)
        min_diff_key = k;
        return;
    if (min_diff > Math.abs(ptr.key - k)) {
        min_diff = Math.abs(ptr.key - k);
        min_diff_key = ptr.key;
    if (k < ptr.key)
        maxDiffUtil(ptr.left, k);
    else maxDiffUtil(ptr.right, k);
}
```

```
static int maxDiff (Node root, int k) {
  min_diff = 999999999;
  min_diff_key = -1;
  maxDiffUtil (root, k);
  return min_diff_key;
}
public static void main (String args[]) {
  Node root = newNode (9);
  root.right = newNode (17);
  root.left = newNode (4);
  root.left.left = newNode (3);
  root.left.right = newNode (6);
  root.left.right.left = newNode (5);
  root.left.right.right = newNode (7);
  root.right.right = newNode (22);
  root.right.right.left = newNode (20);
  int k = 18;
  Sys.o (maxDiff (root, k));
}
```

Q3. Find kth smallest element in BST — We have the root of a BST & k as input, find kth smallest element in BST.



[3, 5, 6, 8, 11, 20]
0 1 2 (3) 4 (5)

Sample Input 1: 3
Sample Output 1: 8

Sample Input 2: 5
Sample Output 2: 10

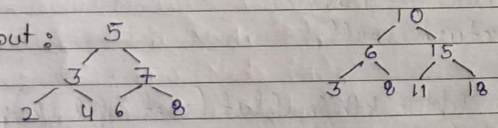
Sol.

```
class Node {
  int data;                          // Time Complexity - O(n)
  Node left, right;                  // Space Complexity - O(H)
  Node (int x) {
    data = x;
    left = right = null;
  }
}
static int count = 0;
public static Node insert (Node root, int x) {
  if (root == null)
    return new Node (x);
  if (x < root.data)
    root.left = insert (root.left, x);
  else (x > root.data)
    root.right = insert (root.right, x);
  return root;
}
public static Node kthSmallest (Node root, int k) {
  if (root == null) {
    return null;
  }
  Node left = kthSmallest (root.left, k);
  if (left != null) {
    return left;
  }
  count ++;
  if (count == k) {
    return root;
  return kthSmallest (root.right, k);
}
```

```
public static void printKthSmallest (Node root, int k){
    Node res = kthSmallest (root, k);
    if (res == null)
        Syso ("There are less than k nodes in the BST");
    else
        Syso ("Kth Smallest Element is " + res.data);
}
public static void main (String[] args) {
    Node root = null;
    int keys [] = {20, 8, 22, 4, 12, 10, 14};
    for (int x : keys ){
        root = insert (root, x);
    int k = 3;
    printKthSmallest (root, k);
}
```

Q4. Two Sum BSTs – Given a BST, transform it into a greater sum tree where each node contains sum of all nodes greater than that node.

Sample Input :



x = 16

Sample Output : 3
– There are pairs i.e. 3 –
(5, 11), (6, 10), (8, 8).

Sol.

```
static class Node {
    int data;
    Node left, right;                    //Time Complexity, - O(n1+n2)
    public Node (int data) { //Space Complexity, - O(h1+h2)
        this.data = data;
        left = right = null;
    }
}
static Node root1; static Node root2; int x
static int countPairs (Node root1, Node root2) {
    if (root1 == null || root2 == null)
        return 0;
    Stack <Node> st1 = new Stack <> ();
    Stack <Node> st2 = new Stack <> ();
    Node top1, top2;
    int count = 0;
    while (true) {
        while (root1 != null) {
            st1.push (root1);
            root1 = root1.left;
        }
        while (root2 != null)
            st1.push (root2 != null);
            root2 = root2.right;
        }
        if (st1.empty() || st2.empty()) {
            break;
        }
        top1 = st1.peek();
        top2 = st2.peek();
```

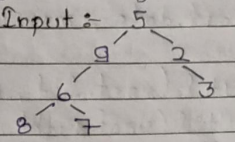
```
if ((top1.data + top2.data) == x) {
    count ++;
    st1.pop();
    st2.pop();
    root1 = top1.right;
    root2 = top2.left;
}
else if ((top1.data + top2.data) < x) {
    st1.pop();
    root1 = top1.right;
}
else {
    st2.pop();
    root2 = top2.left;
}
}
return count;
}
public static void main (String args[]) {
    root1 = new Node(5);
    root1.left = new Node(3);
    root1.right = new Node(7);
    root1.left.left = new Node(2);
    root1.left.right = new Node(4);
    root1.right.left = new Node(6);
    root1.right.right = new Node(8);
    root2 = new Node(10);
    root2.right = new Node(15); root2.left = new Node(6);
    root2.left.left = new Node(3);
    root2.left.right = new Node(8);
    root2.right.left = new Node(11);
    root2.right.right = new Node(18);
```



```
int x = 16;
System.out.println("Pairs = " + countPairs(root1, root2, x));
}
```

Q5. Maximum Sum BST in Binary Tree — We have a binary tree, the task is to print the maximum sum of nodes of a sub-tree which is also a BST.

* Sample Input :-



* Sample Output : 8

Sol.

```
static class Node {
 Node left;
 Node right;
 int data;
 Node (int data)
    this.data = data;
    this.left = this.right = null;
    this.right = null;
};
static class Info{
 int max; int min; boolean isBST; int sum;
 int currmax;
 Info (int m, int mi, boolean is, int su, int cur)
    max = m;
    min = mi;
    isBST = is;
    sum = su
    currmax = cur;
 Info() {}
};
```

```
Static class INT {
    int a;
}
```

```
static Info MaxSumBSTUtil (Node root, INT maxsum){
 if (root == null)
   return new Info (Integer.MIN_VALUE,
                    Integer.MAX_VALUE, true, 0,0);
 if (root.left == null && root.right == null) {
   maxsum.a = Math.max(maxsum.a, root.data);
   return new Info (root.data, root.data, true,
                    root.data , maxsum.a);
 }
 Info L = MaxSumBSTUtil (root.left, maxsum);
 Info R = MaxSumBSTUtil (root.right, maxsum);
 Info BST = new Info();
 if (L.isBST && R.isBST && L.max < root.data
     && R.min > data) {
   BST.max = Math.max(root.data, Math.max(L.max,R.max));
   BST.min = Math.min(root.data, Math.min(L.min,R.min));
   maxsum.a = Math.max(maxsum.a, R.sum + L.sum
                       + root.data);
   BST.sum = R.sum + root.data + L.sum;
   BST.currmax = maxsum.a;
   BST.isBST = true;
   return BST;
 }
 BST.isBST = false;
 BST.currmax = maxsum.a;
 BST.sum = R.sum + root.data + L.sum;
 return BST;
}
```

// Time Complexity - O(n)
// Space Complexity - O(n)


```
static int MaxSumBST (Node root)
  INT maxsum = new INT();
  maxsum.a = Integer.MIN_VALUE;
  return MaxSumBSTUtil (root, maxsum).currmax;
}
public static void main (String args[]) {
  Node root = new Node(5);
  root.left = new Node(14);
  root.right = new Node (3);
  root.left.left = new Node (6);
  root.right.right = new Node (7);
  root.left.left.left = new Node (9);
  root.left.left.right = new Node (1);
  System.out.println (MaxSumBST (root));
}
```