PAGE NO.
DATE: / /

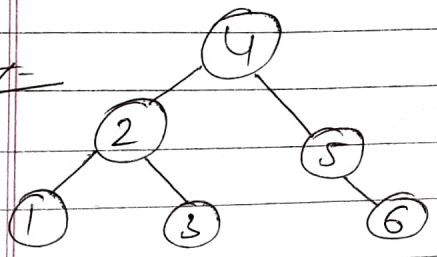
BINARY SEARCH TREE (BST)

What is a BST?

* Binary Tree

a. Left Subtree Nodes < Root
b. Right Subtree Nodes > Root
c. Left and Right Subtrees are also BST with no duplicates.

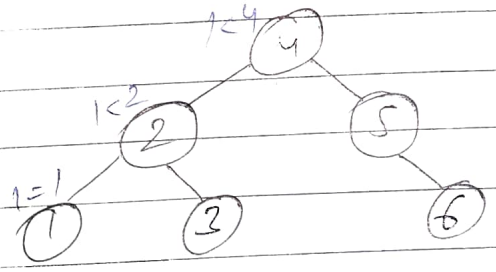
Ex-



Special Property of BST
In order traversal of BST gives a sorted sequence

BST Search

Use Binary Search Algorithm



Key = 1

key > data
go right
key < data
Go left

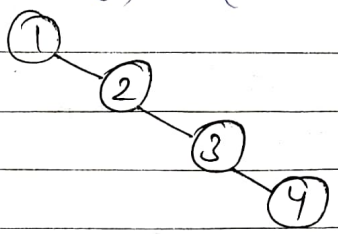
Time complexity = O(H) → Height of tree

Height in balanced BST = log(N) ⇒ Best case

PAGE NO.
DATE: / /

Worst case : $O(N)$ (in skewed tree)

↳ Ex-

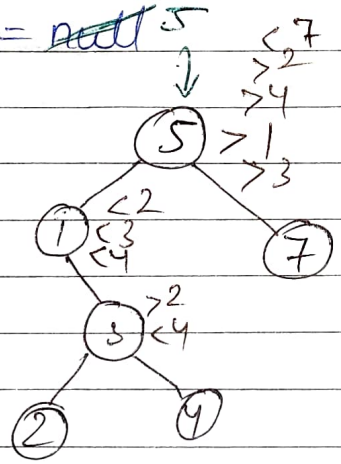


key = 4
$O(H) = O(N)$

Strategy to solve problems
Most problems will be solved using recursion i.e by dividing into subproblems & making recursive calls. on subtree.

Build a BST
values[ ] = {5, 1, 3, 9, 2, 7}

Root = ~~null~~ 5



value[i] > Root
↳ Right Subtree

value[i] < Root
↳ left subtree

PAGE NO.
DATE: / /

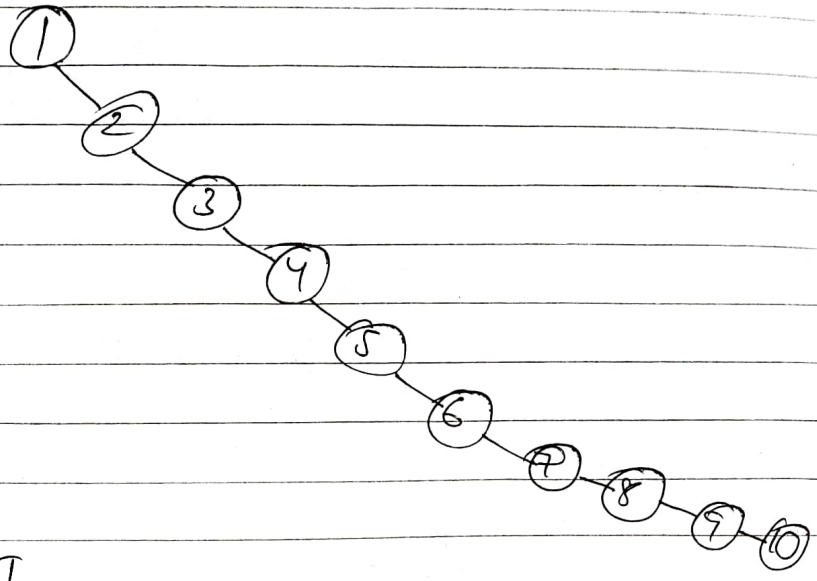
Code:

```
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;
}
psvm (String[] args){
    value[] = {5, 1, 3, 4, 2, 7}; Node root = null;

    for(int i=0; i<values.length; i++){
        root = insert(root, values[i]);
    }
    inorder(root); print root. BST.
}
```

→ func^n for inorder traversal to

PAGE NO.
DATE: / /

BST from 1 to 10

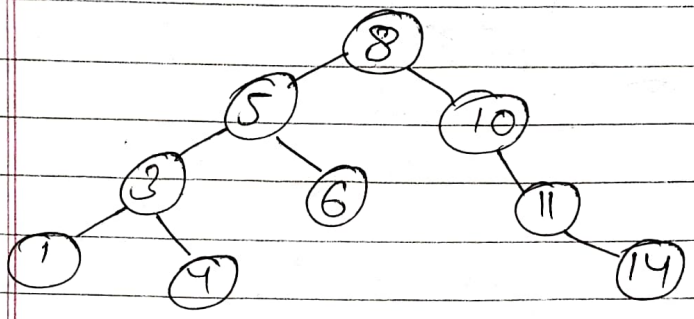


Search in BST

```
public static Node searchBST (Node root, int key){

    if(root == null){
            return null;
    }
    if(root.data == key) return root;
    else if (key < root.data){
                return searchBST(root.left, key);

    } else{
        return searchBST (root.right, key);

    }
}
```

PAGE NO.
DATE: / /

Delete a Node



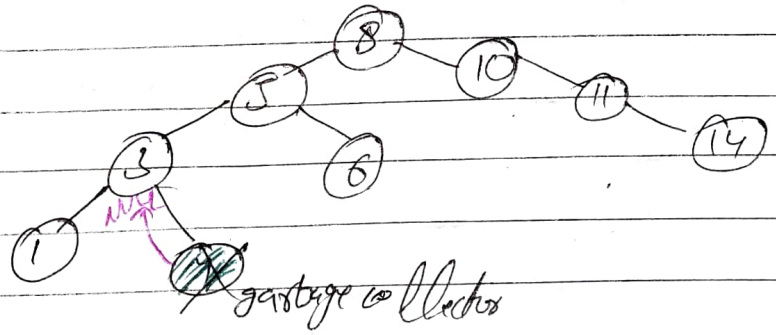
Cases

1. NO child (leaf Node)
2. One child (10, 11)
3. Two children (8, & 3, 5)

1. Search the Node

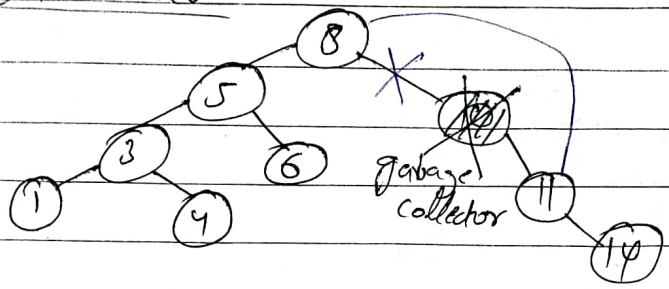
Case 1. Leaf Node

Delete Node & Return null to parent



PAGE NO.
DATE: / /

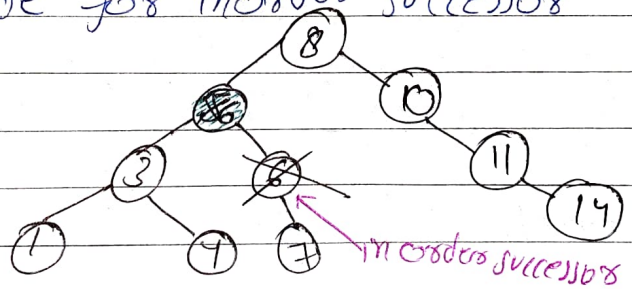
Case2. One child



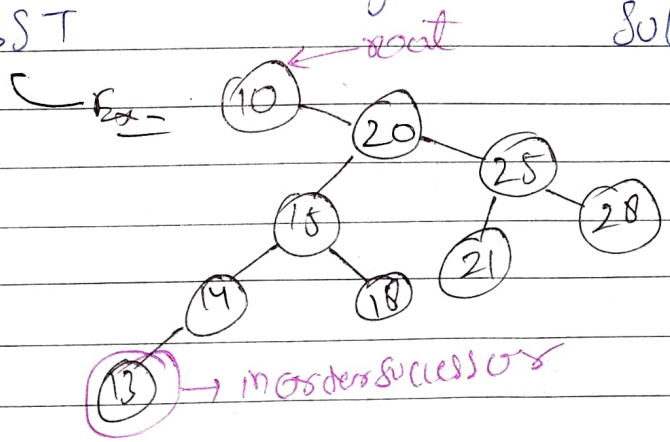
Delete Node & replace with child node

Case3. Two children

Replace value with inorder successor
Delete the node for inorder successor



Inorder successor → Left most node in Right Subtree in BST



* Inorder successor always has zero or 1 child

We need to delete inorder successor using case1 or case2

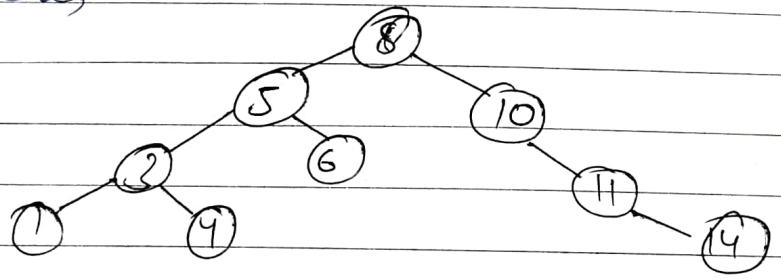
PAGE NO.
DATE: / /

1. find inorder successor
2. replace value with inorder successor
3. Delete the node for inorder successor

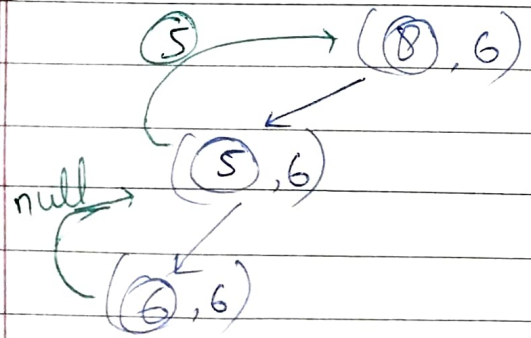
Code:-

```
Public static Node delete(Node root, int val){
    if(root.data

    if(root.data<val){
        root.right = delete(root.right, val);
    }
    else if(root.data >val){
        root.left = delete(root.left, val);
    }else{ //voila
        //case1 - leaf node
        if(root.left == null && root.right == null){
            return null;
        }
        //case2 - single child
        if(root.left == null){
            return root.right;
        }else if(root.right == null){
            return root.left;
        }
```

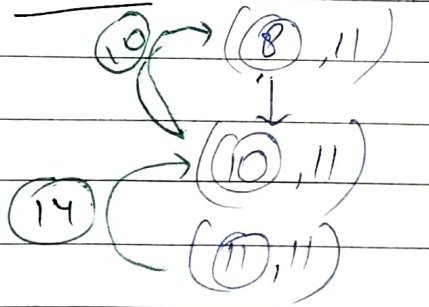
```
//case-3 both children
Node IS = findInorderSuccessor(root.right);
root.data = IS.data;

root.right = delete(root.right, IS.data);

}
return root;
}
public static Node findInOrderSuccessor(Node root){

while(root.left != null)
    root = root.left
}
return root;
}
```



Case ① val = 6

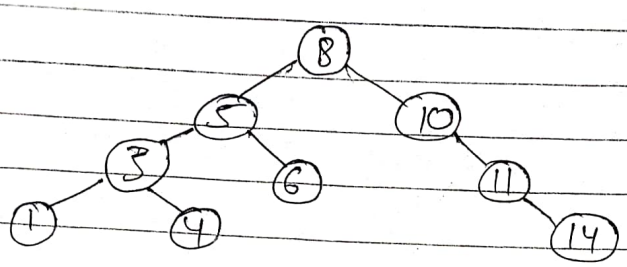


Case ② val = 11



PAGE NO.
DATE: / /

Print in Range



k1 = 5 & k2 = 12

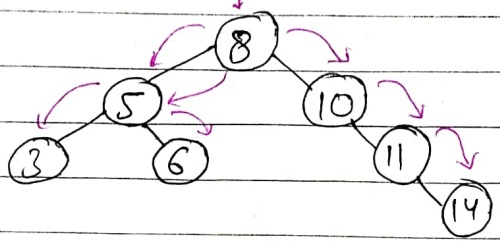
My approach: If k1 < root.data < k2
print root.data
apply same for left
apply same for right

If root.data > k2
apply same for left

If root.data < k1
apply same for right

```
public static void printInRange(Node root, int k1, int k2){
    if(root == null) return;

    if(root.data >= k1 && root.data <= k2){
        System.out.println(root.data + " ");
        printInRange(root.left, k1, k2);
        printInRange(root.right, k1, k2);
    }
    if(root.data > k2){
        printInRange(root.left, k1, k2);
    }
    if(root.data < k1){
        printInRange(root.right, k1, k2);
    }
}
```

Root to leaf Paths



8-5-3
8-5-6
8-10-11-14

~~My Approach~~ ~~Queue~~ ~~?~~

~~while (root != null){~~

Approach: Use ArrayList

8-5-3
8-5-6
8-10-11-14

1. Add Node to path
2. Left subtree
3. Right subtree

When reach leaf → print path
↳ return; → Backtrack and delete node from ArrayList.

Code:-
```
public static void rootToLeafPath (Node root){
    ArrayList<Integer> arr = new ArrayList<>();
    path(root, arr);

public static void path (Node root){
    if(root == null) return;

    arr.add(root.data)
    if(root.left == null && root.right == null){
        for(int i=0; i<arr.size(); i++){
            sout(arr.get(i)+"->");
    }
```

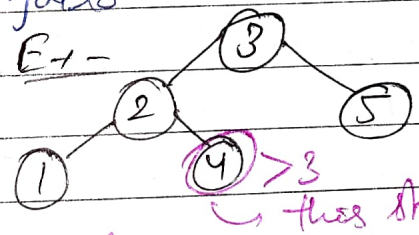
```
        System.out.print("null");
        Sout();
    }
    if(root.left != null || root.right != null){
        arr.add(root.data);
    }
    path(root.left, arr);
    path(root.right, arr);
    arr.removeLast();
```

Validate BST

Approach 1:- Compare root with left & right nodes

But this approach fails

↳ E.g.-



4 > 3

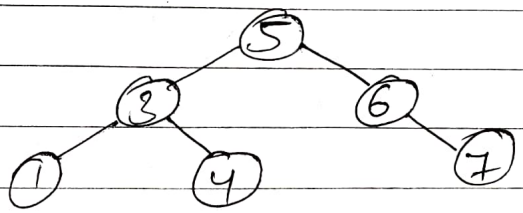
↳ thus should be right

all no.s less than root must be on left

PAGE NO.
DATE: / /

Approach 2 :- If inorder traversal is sorted then it is valid.

Approach 3 :- Check if max value in left subtree < node and check if min value in right is > node

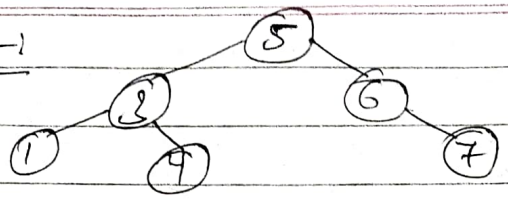


Code :-

```
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);

}
```

PAGE NO.
DATE: / /

Dry Run 1 →



(5, null, null)

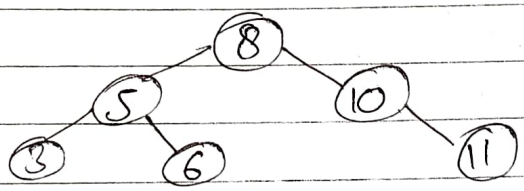
(3, null, 5)

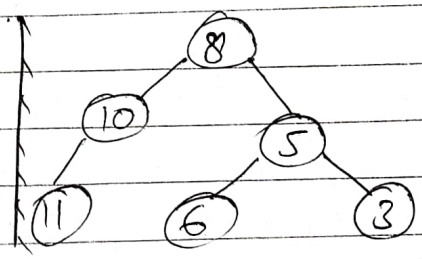
(1, null, 3)

(null, null, 1)

(null, null, 1), n

Mirror A BST



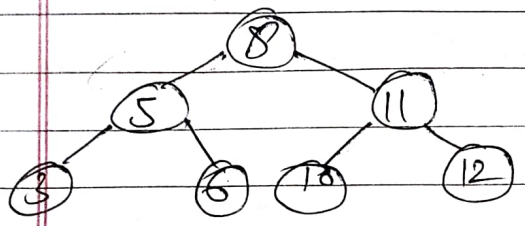



```
public static void Mirroring (Node root){

    if (root == null) return;

    Node temp = root.left;
    root.left = root.right;
    root.right = temp;

    Mirroring (root.left);
    Mirroring (root.right);
}
```

PAGE NO.
DATE: / /

Sorted array to Balanced BST

arr = [3, 5, 6, 8, 10, 11, 12]



→ with minimum possible height.

Approach-: At each step of recursion calculate middle element in array and make it root

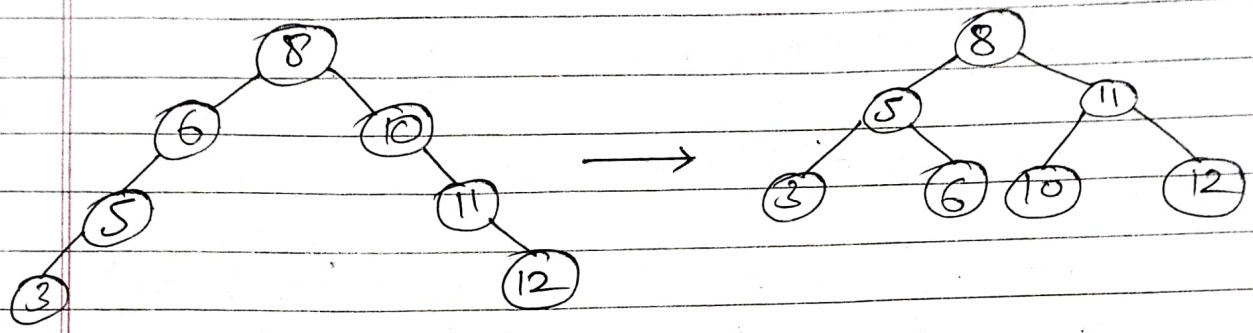
Connect this root to it's left and right.

```
balancedBST (arr, st, end) {
    if (st > end) return null;
    int mid = (st+end)/2

    Node root = new Node(arr[mid])

root.left=balancedBST (arr, st, mid-1)

root.right = balancedBST (arr, mid+1, end)

    return root;
```

Time Complexity : O(n)

PAGE NO.
DATE: / /

Q. Convert BST to Balanced BST



Approach:- We know how to make a balanced BST from a sorted array.

i) Store values from a morder traversal in arraylist

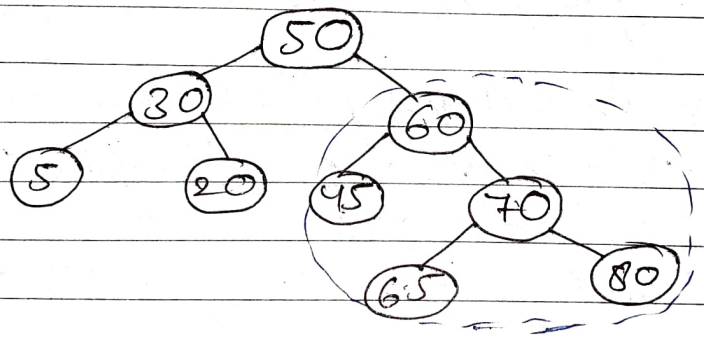
ii) Use this arraylist to create balanced BST.

Code:

```
public static Node toBalancedBST (Node root){
    ArrayList<Node> arr = new ArrayList<>();
    arr = morder(root, arr);
    return balancedBST (arr, 0, arr.size()-1);

public static ArrayList<Node> morder(Node root,
                                     ArrayList<Node> arr){
    if(root == null) return arr;
    morder(root.left, arr);
    arr.add(root);
    morder(root.right, arr);
    return arr;
}
```

```
public static Node balancedBST(ArrayList<Node> arr, int
                                start, int end){
    if(start > end) return null;

    int mid = (start+end)/2;

    Node root = new Node(arr.get(mid).data);
    root.left = balancedBST(arr, start, mid-1);
    root.right = balancedBST(arr, mid+1, end);

    return root;
}
```

Q. Size of Largest BST in BT



Ans = 5

PAGE NO.
DATE: / /

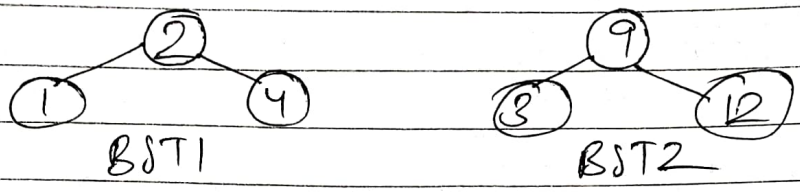
Code-:

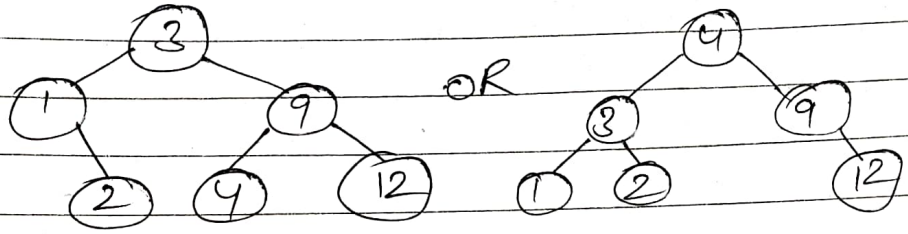
```
public class Solution {
    static class Node {
        ---------
        ---------
        ---------
    }
    static class Info {
        boolean isBST;
        int size;
        int min;
        int max;

        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;

    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){
    return
    MaxBST = Math.max(maxBST, size);

    return new Info(true, size, min, max);

}
return new Info(false, size, min, max);

}
```

PAGE NO.
DATE: / /

Q. Merge 2 BSTs into Balanced BST





Approach:- ① Store inorder seq of BST1 in arr1.

② Store inorder seq. of BST2 in arr2.

③ Merge arr1 & arr2

④ sorted arr → balanced BST

PAGE NO.
DATE: / /

AVL Trees

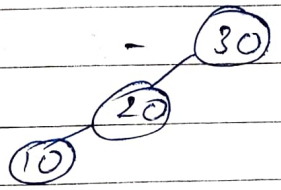
→ Self balancing BST

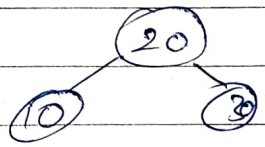
Why we need Balancing?

Operations on BST
- insert
- remove
- search

} O(H) → log n

In skewed BST time complexity = O(n)





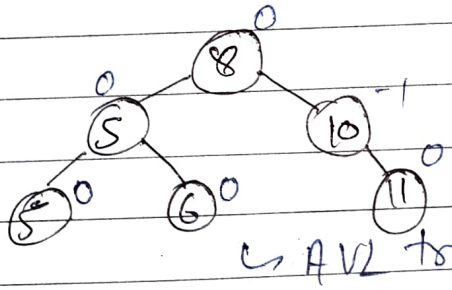
} log n

Property:

i) Balance Factor (bf) = left Height - Right Height = {-1, 0, 1}

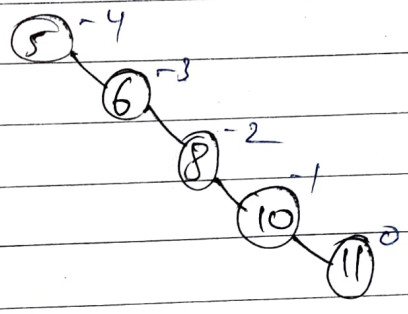
or |bf| < 2

Balanced



↳ AVL tree

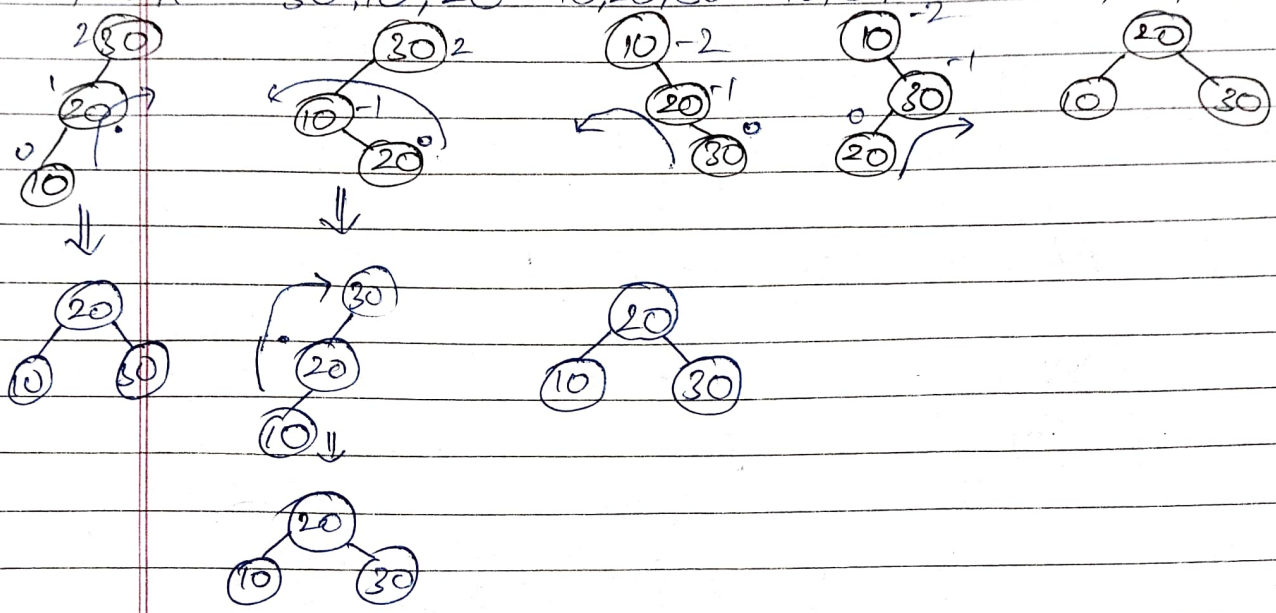
Unbalanced



ii) Rotations

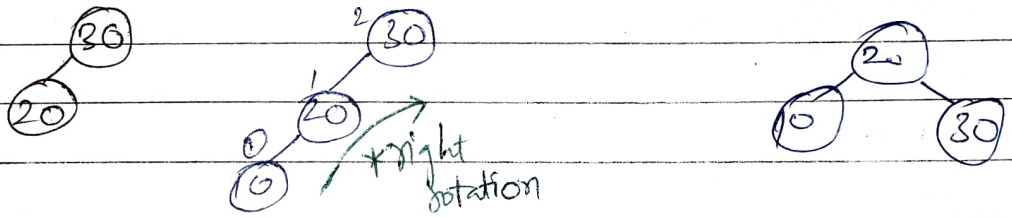
(10,20,30) → n=3 No. of BSTs = n! = 3! = 6

30,20,10 | 30,10,20 | 10,20,30 | 10,30,20 | 20,30,10 | 20,10,30



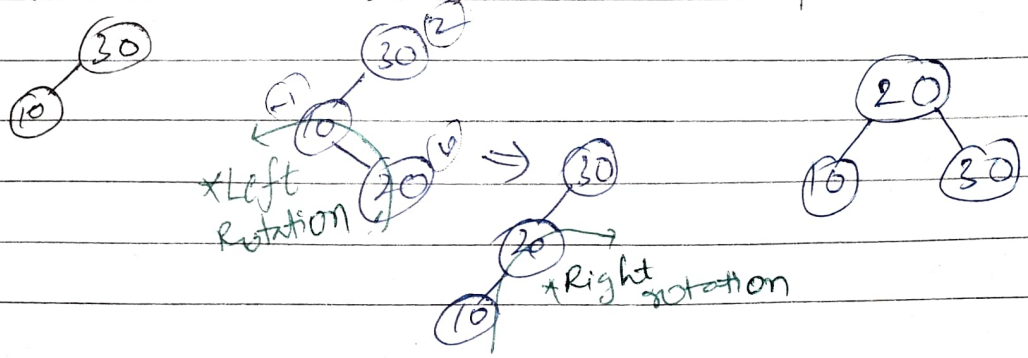
Left Left case (LL case) – Right Rotation

AVL Tree | Insert 10 | After Rotation



LR case – Left Rotation then Right Rotation

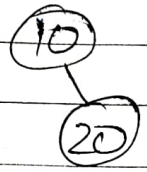
AVL Tree | Insert 20 | After Rotation



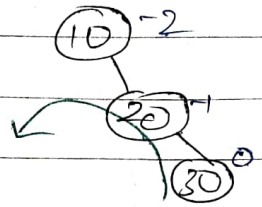
PAGE NO.
DATE: / /

RR case - Left Rotation

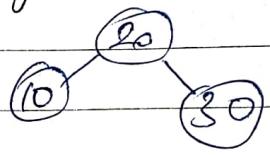
AVL Tree



Insert 30

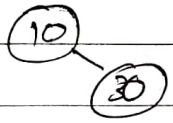


After Rotation

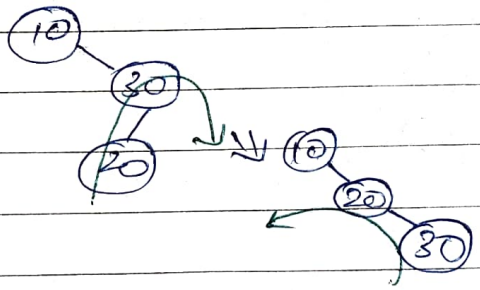


RL case - Right Rotation Left Rotation

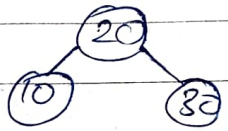
AVL Tree



Insert 20



After Rotation



Steps

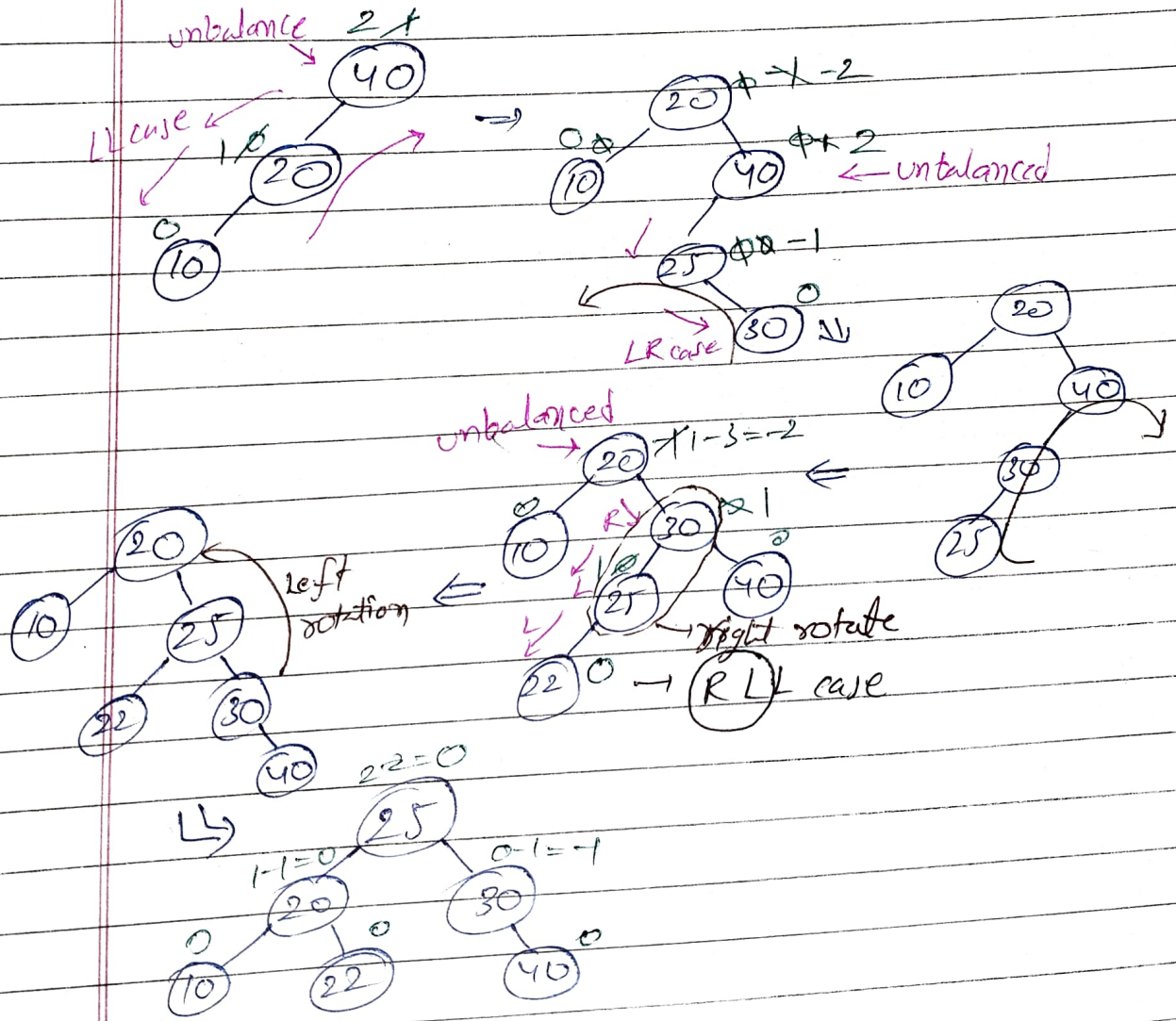
i) Check balance for each node (if imbalanced)
ii) Identify case
iii) rotate

PAGE NO.
DATE: / /

Creating an AVL Tree

1. Insert as BST
2. check balance factor → case → rotate

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



PAGE NO.
DATE: / /