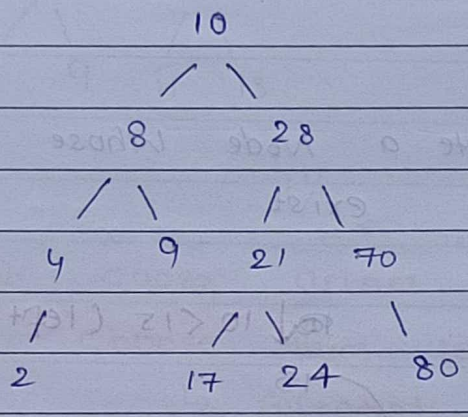
Day - 48

(Balanced Binary Search Tree)

problem : Delete a Node from BST :

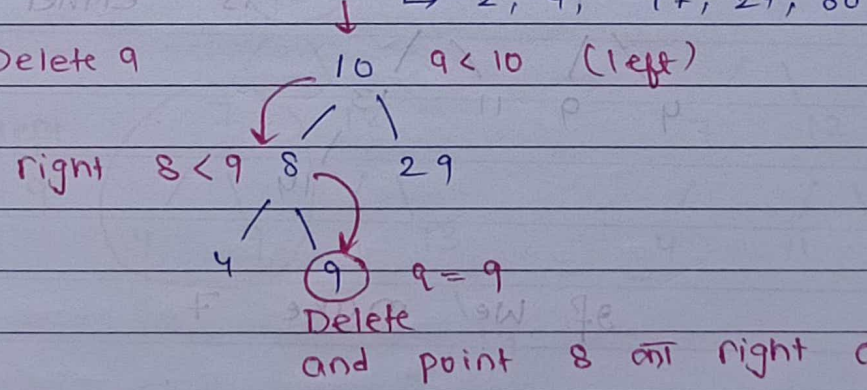


case 1 : leaf Node to be delete

↳ directly delected

↳ 2, 9, 17, 24, 80

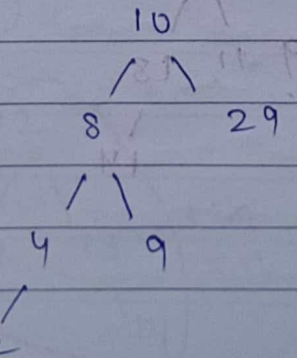
Delete 9

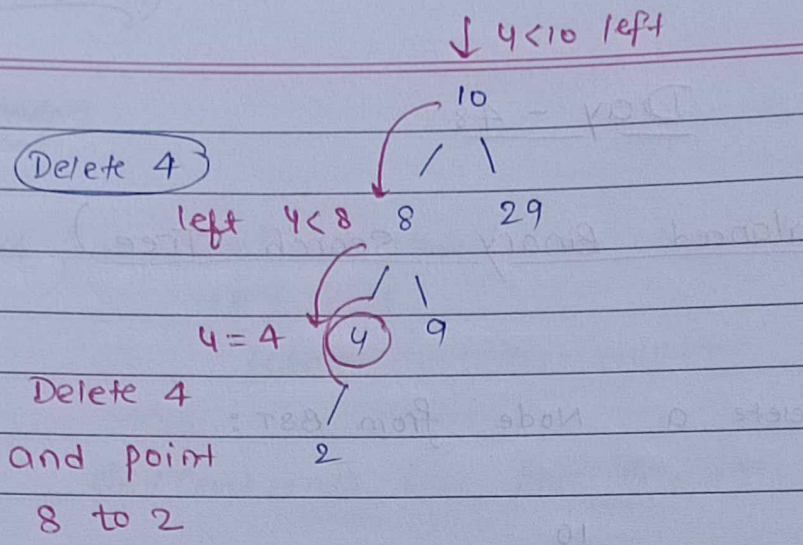


case 11 : Delete a Node whose 1 child exist.

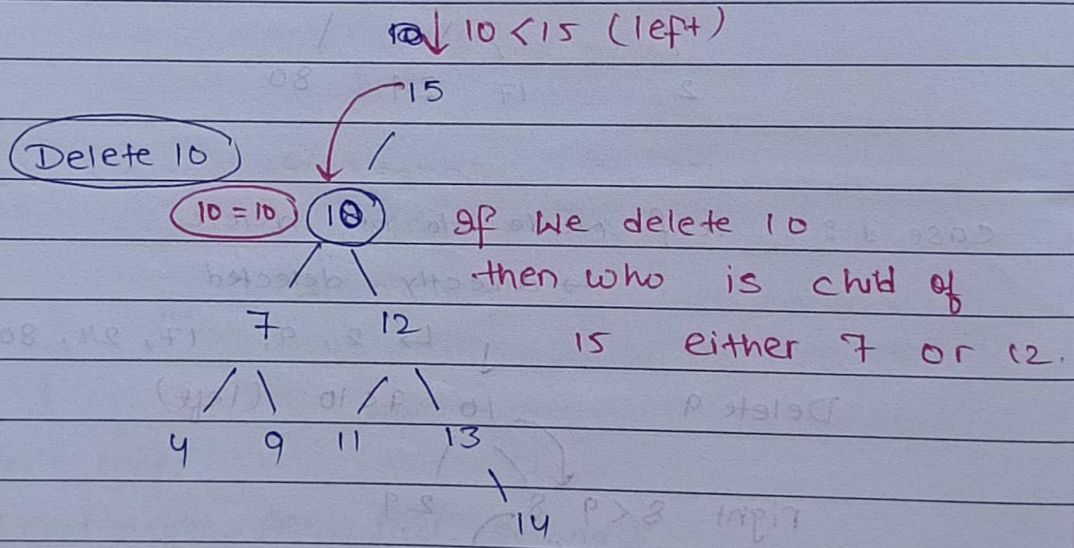
↳ 4, 70

Delete 4

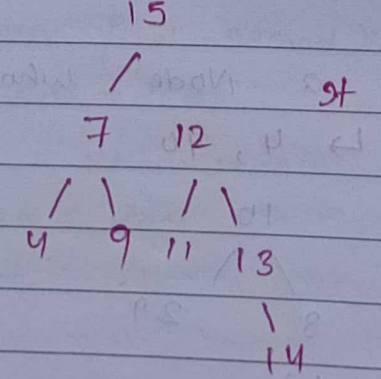




Case III : Delete a Node whose both child exist

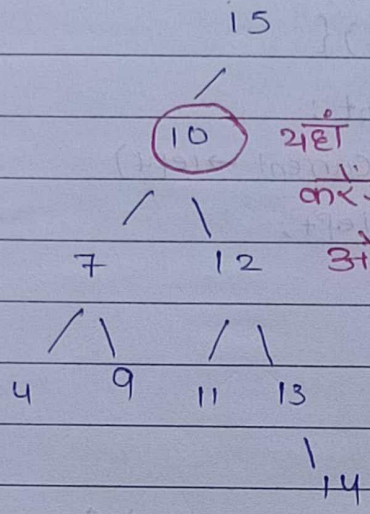


If we make 7



It can't be possible.

So, we do:

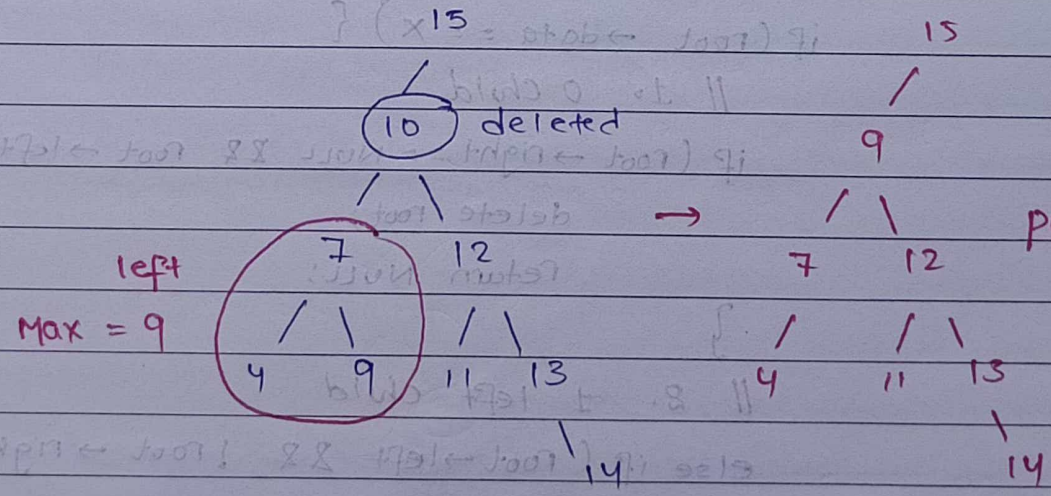


यहाँ पर हम एक ऐसा value put करेंगे जिसका सारा left छोटा हो और सारा right बड़ा हो।

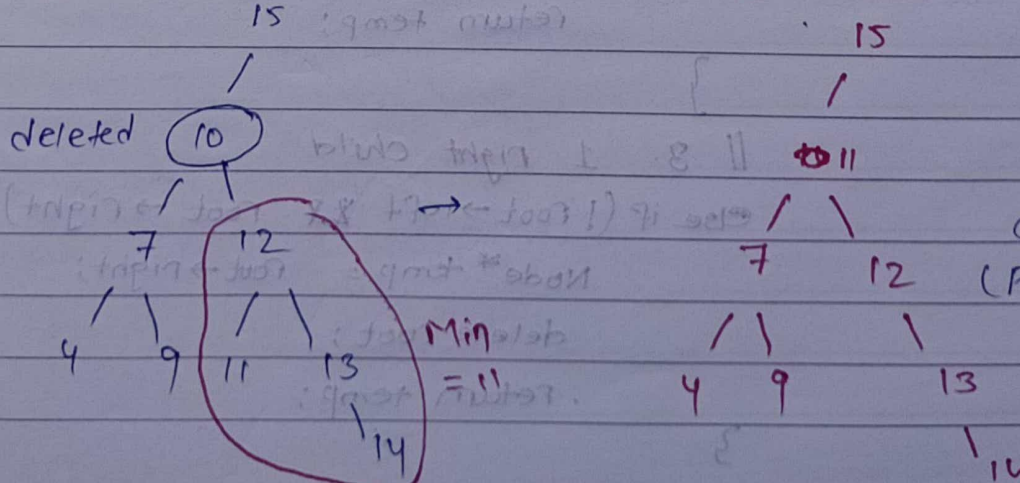
Here, two cases arises

↳ (1) Choose from left subtree

↳ Greatest from left subtree



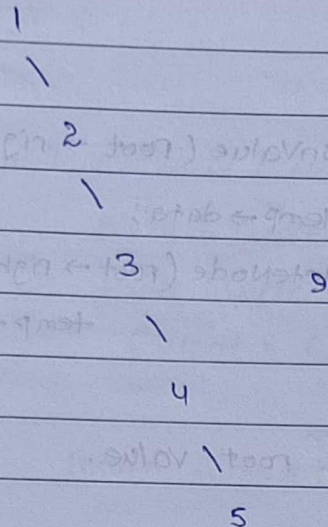
(2) Choose Minimum from right subtree.



Code :

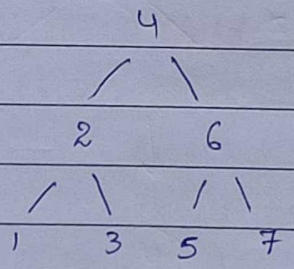
```
Node* minValue (Node* root){
    Node* Current = root;
    While (Current && Current->left)
     Current = Current->left;

    return Current;
}


Node *deleteNode (Node *root, int x){
        if (root == NULL)
          return NULL;

  //1. X value is equal to root.
        if (root->data == x){
          // 1. 0 child
          if (root->right == NULL && root->left == NULL){
              delete root;
              return NULL;
          }
          // 2. 1 left child
          else if (root->left && !root->right){
              Node* temp = root->left;
              delete root;
              return temp;
          }
          // 3. 1 right child
          Else if (!root->left && root->right){
              Node* temp = root->right;
              delete root;
              return temp;
          }
```

```
// 4. 2 child
        else {
            Node * temp = minValue (root -> right);
            root -> data = temp -> data;
            root -> right = deleteNode (root -> right,
                                        temp -> data);
        }
    }
// 2. x value is less than root value.
    else if ( x < root -> data ) {
        root -> left = deleteNode (root -> left, x);
    }
// 3. x value is greater than root value.
    else {
        root -> right = deleteNode (root -> right, x);
    }

    return root;
}
```



On this if we want to search 7, then it takes O(N) time.

But, after arranging:



On this we can search is O(log N)

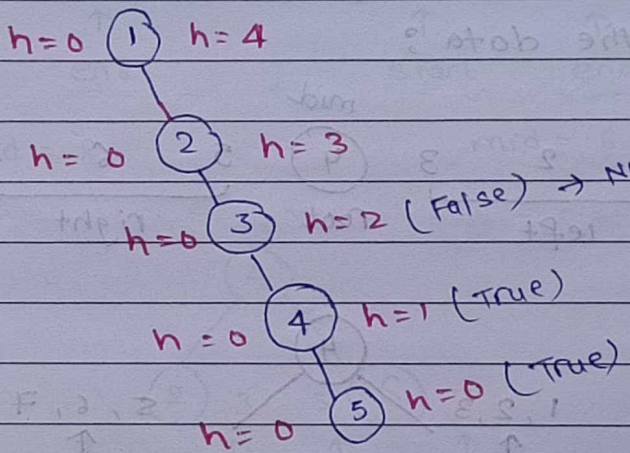
This tree is called Balanced Binary search tree.

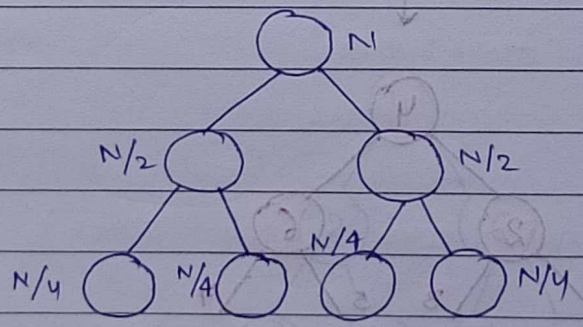
Balanced Binary search tree

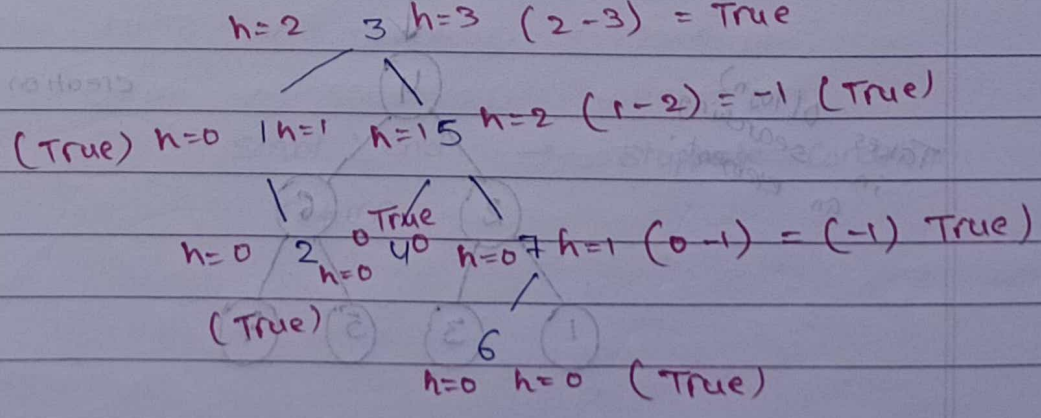
A binary tree is balanced if the height of tree is $O(\log N)$ where n is the number of nodes.

$$-1 \leq (\text{left height} - \text{right height}) \leq 1$$

Then, tree is ~~ppe~~ perfect. Balanced tree.



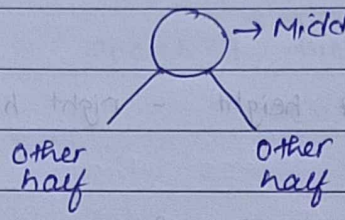




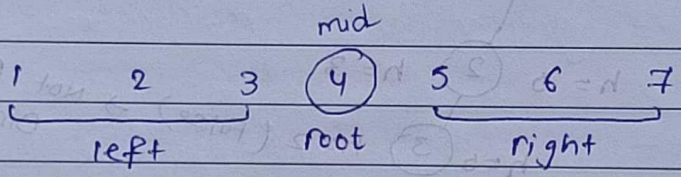
Balanced Binary search tree.

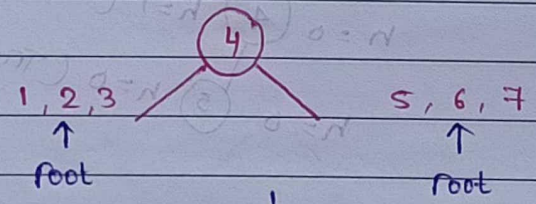
How we Create a Balanced Binary Search tree :

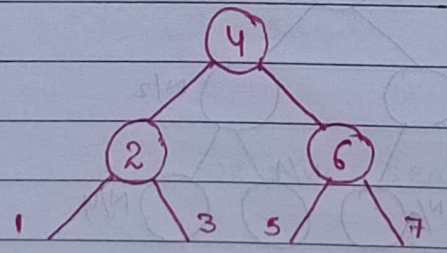
7 4 1 6 5 2 3

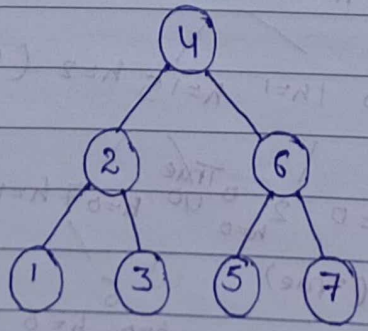


Sort the data :









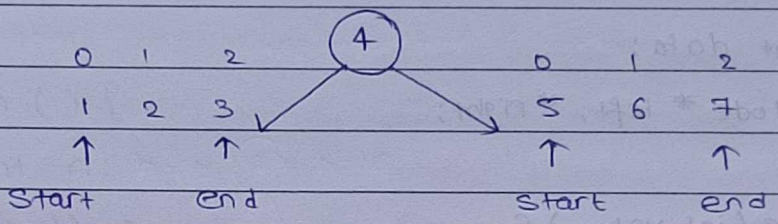
Takes O(logn) searching in an element.

creation : O(nlogn)

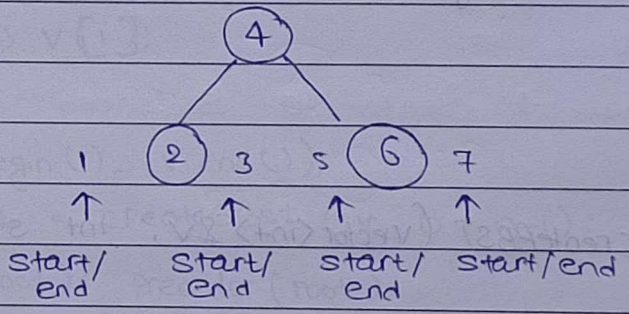
0	1	2	3	4	5	6
1	2	3	4	5	6	7
↑						↑
start						end

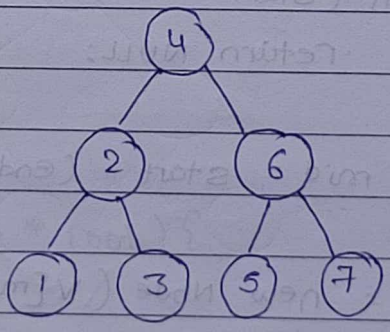
mid = start + (end - start) / 2;

mid = 4



mid = 2 mid = 6





start > end → Stoppage Condition.

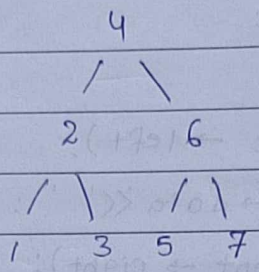
return

Code :

```
#include <iostream>
using namespace std;

~~main (void)~~

    ~~vector~~

Class Node {
    public :
    int data;
    Node * left, *right;

  Node (int value) {
      data = value;
      left = right = NULL;
    }
}

Node * CreateBBST (vector<int> &V, int start, int end) {

            if (start > end)
              return NULL;

          int mid = start + (end - start) / 2;

    Node * root = new Node (V[mid]);
    root → left = CreateBBST ( V, start, mid - 1);
    root → right = CreateBBST (V, mid + 1, end);
     return root;
}
```

```
Void inorder ( Node * root){
        if (!root)
            return;

        inorder (root -> left);
        cout << root -> data << " ";
        inorder (root -> right);
}

int main (){
    int n;
    cin >> n;
    vector <int> v(n);
    for (int i = 0; i<n; i++)
        cin >> v[i];

    sort (v.begin(), v.end());
    Node * root = createBBST (v, 0, n-1);
    inorder(root); Preorder(root);
    return 0;
}


Void preorder ( Node * root ){
        if (!root)
            NULL;
        cout << root -> data << " ";
        preorder( root -> left );
        preorder ( root -> right);
    }
```

preorder : 4 2 1 3 6 5 7



Inorder check about sorted.

preorder can check balance of tree.

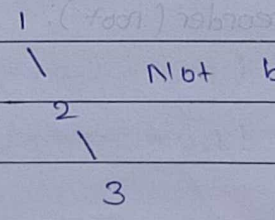
→ On the same problem we have two condition

① you don't have to sort

② create at Runtime.

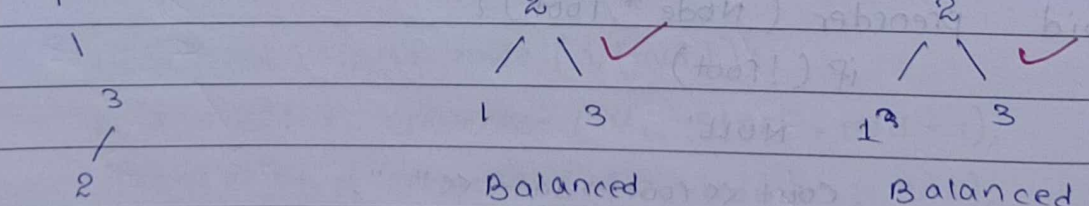
Then how to create a Balanced Binary search tree

1, 2, 3



Not balanced.

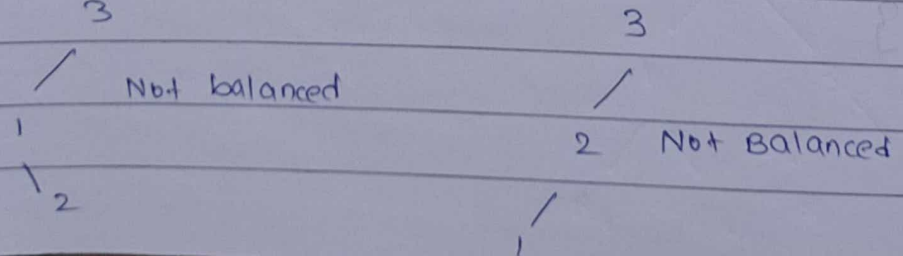
1 3 2 2 1 3 2 3 1



Balanced Balanced

Not balanced

3 1 2 3 2 1



Not balanced Not Balanced

How we balance, where we have problem :

① left side ↓root

1

\R

2

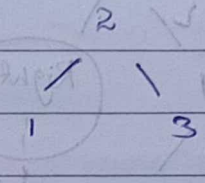
\R

3

R-R unbalance

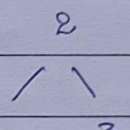
↳ then Move Root to left.

1 को तोड़ कर नीचे ले आओ ⇒



② ↓root

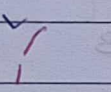
3 right side fall ⇒ Break. 3 put down.



2

L-L unbalance

↳ the Move root to right



③ ↓root

3

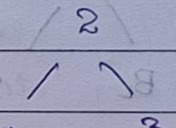
L/

1

R\

2

⇒



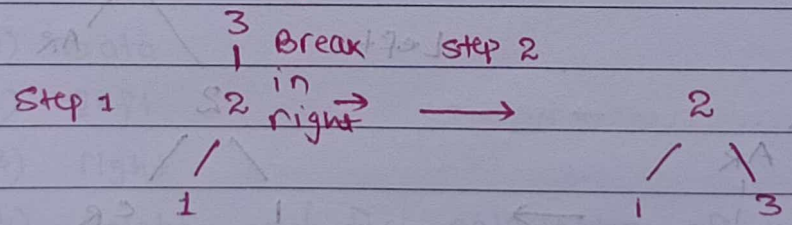
L-R unbalance

↳ Move root→left to left.

↳ Move root to right.

Break in two step

1 को left



3

Break in right ⇒ step 2

Step 1

2

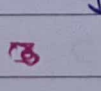
1

⟶

2

1 3

④ ↓root



1

\R

3

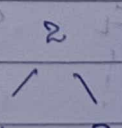
/L

2

R-L unbalance

↳ Move root ⇒ right to right

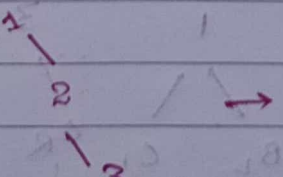
↳ Move root to left.



2

1 3

Step 1 :

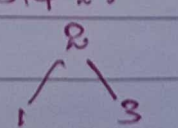


1

2

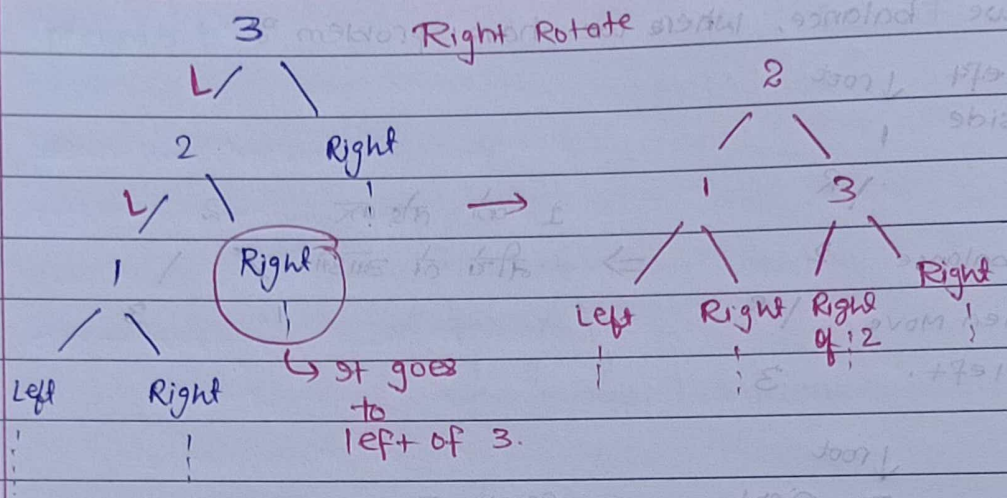
⟶

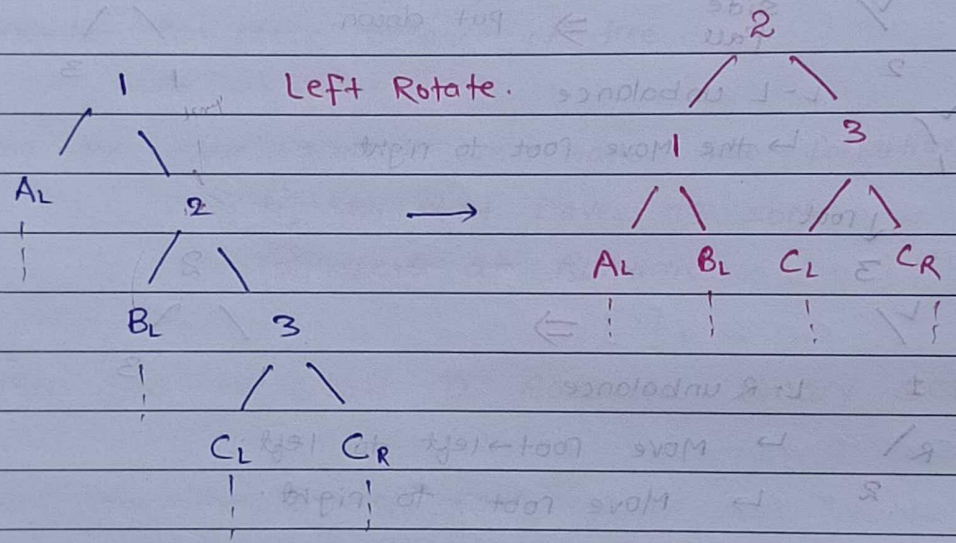
Step 2 :

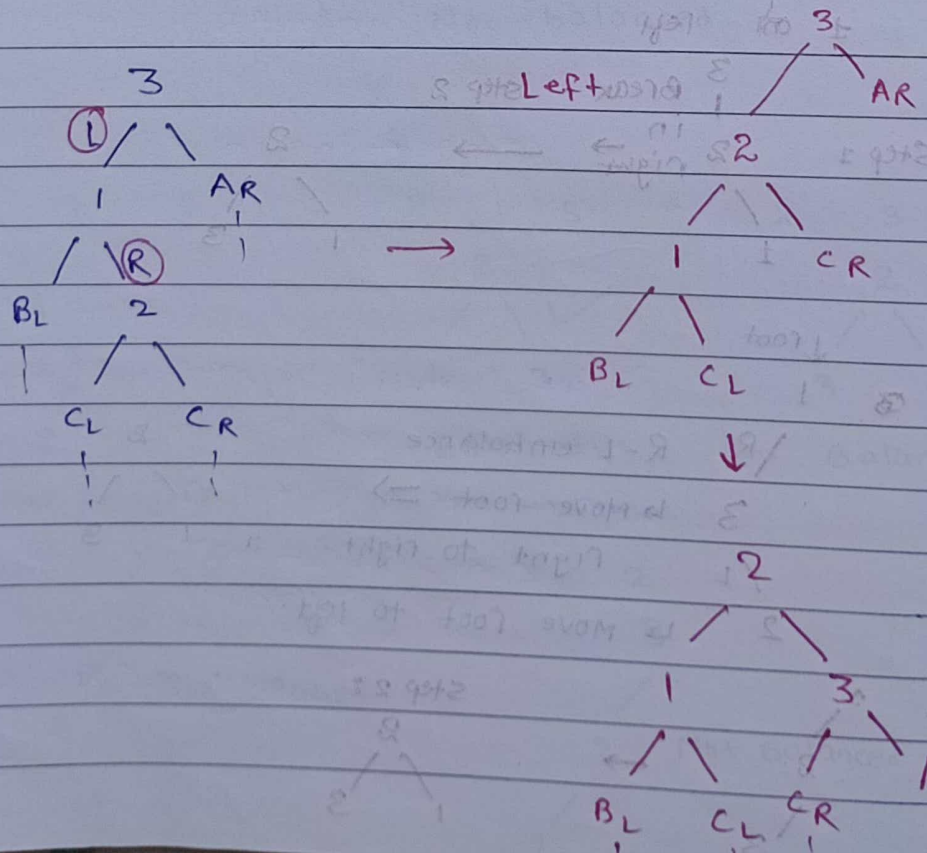


2

1 3







classmate
Date
Page

Two function

1) Rotate left
2) Rotate Right

```
Rotate_left (Node * root) {
        Node* temp = root → right;
        root → right = temp → left;
        temp → left = root;
        return temp;
    }

Rotate_right (Node * root) {
        Node * temp = root → left;
        root → left = temp → right;
        temp → right = root;
        return temp;
    }
```

→ Node creation :

1) Data
2) left
3) right
4) height (for calculation of height on each step).

→ find Element right position ⇒ add
→ Return → check Balance
→ If Not Balance, Balance it.

Code :

```
#include <iostream>
using namespace std;

class Node {
    public:
        int data, height;
        Node * left, * right;

    Node (int value) {
        data = value;
        height = 1;
        left = right = NULL;
    }
};

int getHeight (Node * root) {
    if (!root)
        return 0;

    return root -> height;
}

Node * rotateRight (Node * root) {
    Node * temp = root -> left;
    root -> left = temp -> right;
    temp -> right = root;
    updateHeight (root);
    updateHeight (temp);
    return temp;
}
```

```
Node * rotateLeft (Node * root) {
        Node* temp = root->right;
        root->right = temp->left;
        temp->left = root;
        updateHeight(root);
        updateHeight(temp);
    return temp;
}

int main() {
    Node * root = NULL;
}

Node * insertBST (Node * root, int value) {
        if (!root)
            return new Node(value);

        if (value < root->data)
            root->left = insertBST (root->left, value);

        else
            root->right = insertBST (root->right, value);

    return root;
}

Node * Balance (Node * root) {

        // Complete Yourself

    }

int main() {
        Node * root = NULL;
        int value;
```

```
While (1) {
    cin >> value;
    if (value != -1)
        root = insertBST (root , value);
    else
        break;
};
```

```
~~Node * Balance (Node * root) {~~
    ~~if (! root)~~
        ~~return NULL;~~
    inorder (root);
    preorder (root);
return 0;
}
```