Trees : Level 3
Lesson 3

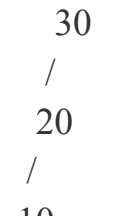
1. normal-bst-to-balanced-bst
2. Insert-a-node-in-a-bst
3. Delete a node from BST
4. implement-change-key-in-bst
5. median-of-bst
6. sum-of-leaf-nodes-in-bst
7. brothers-from-different-root

Normal BST to Balanced BST

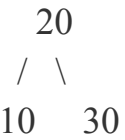
Given a Binary Search Tree, modify the given BST such that it is balanced and has minimum possible height.

Examples :

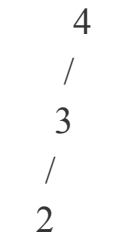
Input:



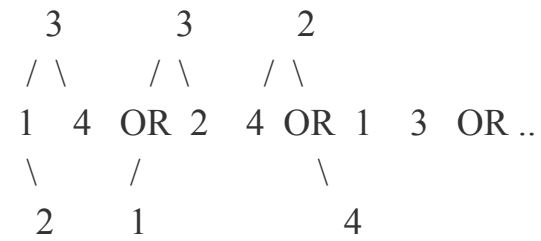
Output:



Input:



Output:



Expected Time Complexity: O(N)
Expected Auxiliary Space: O(N)

Constraints:

1<=N<=200

```
1.  Node*buildtree(vector<Node*>&v,int start, int end)
2.  {
3.    if(start>end)
4.       return NULL;
5.    int mid=(start+end)/2;
6.    Node*head=v[mid];
7.    head->left=buildtree(v,start,mid-1);
8.    head->right=buildtree(v,mid+1,end);
9.    return head;
10. }
11. void inorder(Node*root,vector<Node*>&v)
12. {
13.   if(root==NULL)
14.      Return;
15.   inorder(root->left,v);
16.   v.push_back(root);
17.   inorder(root->right,v);
18. }
19. Node* buildBalancedTree(Node* root)
20. {
21.   vector<Node*>v;
22.   inorder(root,v);
23.   int n = v.size();
24.   return buildtree(v,0,n-1);
25. }
```

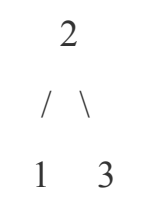
Insert a node in a BST

Given a BST and a key K. If K is not present in the BST, Insert a new Node with a value equal to K into the BST.

Note: If K is already present in the BST, don't modify the BST.

Example 1:

Input:



K = 4

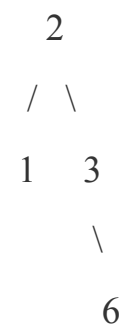
Output: 1 2 3 4

Explanation: After inserting the node 4

Inorder traversal will be 1 2 3 4.

Example 2:

Input:



K = 4

Output: 1 2 3 4 6

Explanation: After inserting the node 4

Inorder traversal of the above tree will be 1 2 3 4 6.

Expected Time Complexity: O(Height of the BST).

Expected Auxiliary Space: O(Height of the BST).

Constraints:

1 <= Number of nodes <= 105

1 <= K <= 106

```
1.  //Function to insert a node in a BST.
2.  Node* insert(Node* root, int Key)
3.  {
4.    if(!root)
5.       return new Node(Key);
6.    if(root->data>Key)
7.       root->left= insert(root->left,Key);
8.    else if(root->data<Key)
9.       root->right = insert(root->right,Key);
10.   return root;
11.   // Your code here
12. }
```

Delete a node from BST

Given a Binary Search Tree and a node value X. Delete the node with the given value X from the BST. If no node with value x exists, then do not make any change.

Example 1:

Input:

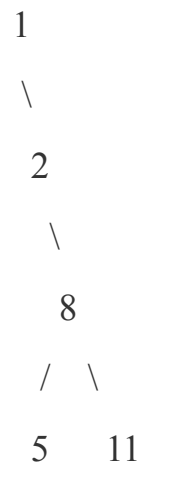
X = 12

Output: 1 2 3

Explanation: In the given input there is no node with value 12 , so the tree will remain the same.

Example 2:

Input:



X = 9

Output: 1 2 4 5 7 8 11 12

Explanation: In the given input tree after

deleting 9 will be

Expected Time Complexity: O(Height of the BST).

Expected Auxiliary Space: O(Height of the BST).

```
1.    Node *deleteNode(Node *root,  int X)
2.    {
3.      if(root==NULL)
4.         return NULL;
5.      if(root->data>X)
6.         {
7.            root->left=deleteNode(root->left,X);
8.            return root;
9.         }
10.     else if(root->data<X)
11.        {
12.           root->right=deleteNode(root->right,X);
13.           return root;
14.        }
```

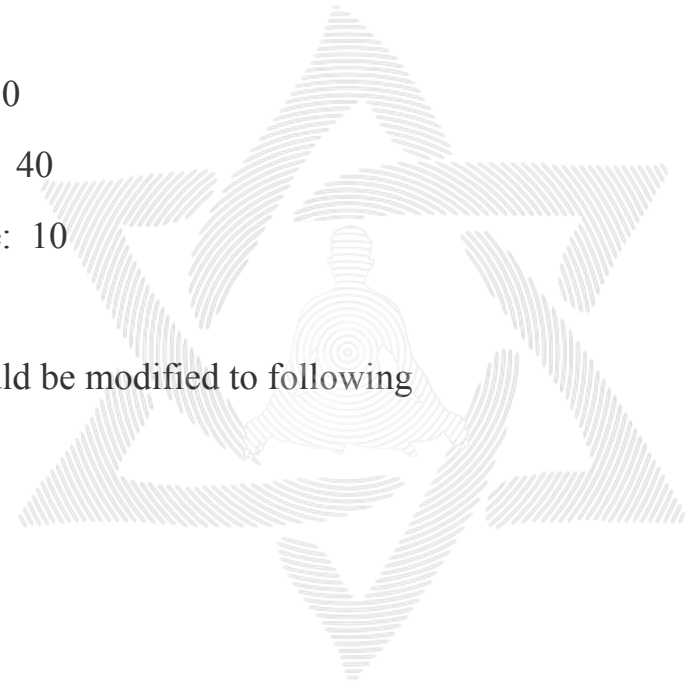
```
15.     if (root->left == NULL) {
16.         Node* temp = root->right;
17.         delete root;
18.         return temp;
19.     }
20.     else if (root->right == NULL) {
21.         Node* temp = root->left;
22.         delete root;
23.         return temp;
24.     }
25.     else
26.     {
27.         Node*insuccP=root;
28.
29.         Node*insucc=root->right;
30.         while(insucc->left)
31.         {
32.             insuccP=insucc;
33.             insucc=insucc->left;
34.         }
35.         if(insuccP==root)
36.             insuccP->right=insucc->right;
37.         else
38.             insuccP->left=insucc->right;
39.
40.         root->data=insucc->data;
41.         delete insucc;
42.
43.         return root;
44.     }
45.     // your code goes here
46. }
```

Change of Key in BST

Given a Binary Search Tree ,change the old key value present in the tree to the given new key value.

Example 1:

Input: Root of below tree



Old key value: 40

New key value: 10

Output: BST should be modified to following

Expected Time Complexity: O(Height of tree)

Expected Auxiliary Space: O(1).

Constraints:

1<=Number of Nodes in tree<=500

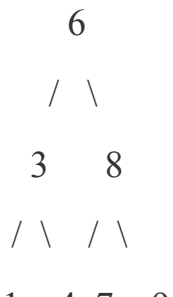
```
1.  void inorder(node* root, vector<int> &p)
2.  {
3.    if(!root)   return;
4.    inorder(root->left,p);
5.    p.push_back(root->key);
6.    inorder(root->right,p);
7.  }
8.  void insert(node* root, vector<int> &p)
9.  {
10.   if(!root)   return;
11.   insert(root->left,p);
12.   root->key=p[0];
13.   p.erase(p.begin());
14.   insert(root->right,p);
15. }
16. struct node *changeKey(struct node *root, int oldVal, int newVal)
17. {
18.       // Code here
19.       vector<int> p;
20.       inorder(root,p);
21.       p.push_back(newVal);
22.       p.erase(remove(p.begin(),p.end(),oldVal));
23.       sort(p.begin(),p.end());
24.       insert(root,p);
25.       return root;
26. }
```

Median of BST

Given a Binary Search Tree of size N, find the Median of its Node values.

Example 1:

Input:



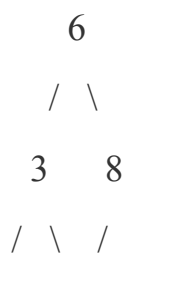
Output: 6

Explanation: Inorder of Given BST will be:

1, 3, 4, 6, 7, 8, 9. So, here the median will be 6.

Example 2:

Input:



Output: 5

Explanation:Inorder of Given BST will be:

1, 3, 4, 6, 7, 8. So, here median will

(4 + 6)/2 = 10/2 = 5.

Expected Time Complexity: O(N).

Expected Auxiliary Space: O(Height of the Tree).

Constraints:

1<=N<=1000

```
float findMedian(struct Node *root)
{
  vector<int>v;
  inorder(root,v);

  int n = v.size();

  if(n%2)
  {
     int mid = n/2;
     return (float)(v[mid]);
  }
  else
  {
     int mid1 =n/2;
     return (float)((float)(v[mid1]+v[mid1-1])/2);
  }
   //Code here
}
```

Sum of leaf nodes in BST

Given a Binary Search Tree, find the sum of all leaf nodes. BST has the following property (duplicate nodes are possible):

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than or equal to the node's key.

Input:

The first line of input contains a single integer T denoting the number of test cases. Then T test cases follow. Each test case consists of two lines. The first line of each test case consists of integer N, denoting the number of elements in the BST. The second line of each test case consists of N space-separated integers denoting the elements in the BST.

Output:

For each test case, in a new line, print the sum of leaf nodes.

Constraints:

1 <= T <= 103

1 <= N <= 105

Example:

Input:

2

6

67 34 82 12 45 78

1

45

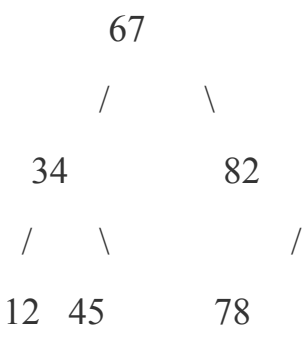
Output:

135

45

Explanation:

In the first test case, the BST for the given input will be-



Hence, the required sum= 12 + 45 + 78 = 135

```
1.  void calculate(Node*root,int &sum)
2.  {
3.    if(root==NULL)
4.       return;
5.    if(!root->left && !root->right)
6.    {
7.       sum+=root->data;
8.    }
9.    calculate(root->left,sum);
10.   calculate(root->right,sum);
11. }
12. int sumOfLeafNodes(Node *r )
13. {
14.   int sum=0;
15.   calculate(r,sum);
16.   return sum;
17.  }
```

Brothers From Different Roots

Given two BSTs containing N1 and N2 distinct nodes respectively and given a value x. Your task is to complete the function countPairs(), which returns the count of all pairs from both the BSTs whose sum is equal to x.

Examples:

Input : BST 1: 5

```
        /   \
       3     7
      / \   / \
     2  4  6   8
```

BST 2: 10

```
        /   \
       6     15
      / \   /  \
     3  8  11  18
```

x = 16

Output : 3

The pairs are:

(5, 11), (6, 10) and (8, 8)

Input:

The function takes three arguments as input, first the reference pointer to the root(root1) of the BST1, then reference pointer to the root(root2) of the BST2 and last the element X.

There will be T test cases and for each test case the function will be called separately.

Output:

For each test case, print the required number of pairs on a new line.

Constraints:

1<=T<=100

1<=N<=103

Example:

Input:

2

7

5 3 7 2 4 6 8

7

10 6 15 3 8 11 18

16

6

10 20 30 40 5 1

5

25 35 10 15 5

30

Output:

3

2

```
1.  void bst(Node*root,unordered_set<int>&s)
2.  {
3.    if(root==NULL)
4.       return;
5.    bst(root->left,s);
6.    s.insert(root->data);
7.    bst(root->right,s);
8.  }
9.  int countPairs(Node* root1, Node* root2, int x)
10. {
11.   unordered_set<int>s1,s2;
12.   bst(root1,s1);
13.   bst(root2,s2);
14.
15.
16.   int count=0;
17.   for(auto i:s1)
18.   {
19.      if(s2.find(x-i)!=s2.end())
20.         count++;
21.   }
22.   return count;
23.   // Code here
24. }
```