10) CONVERT A NORMAL BST INTO BALANCED BST:

METHOD 1:
The simple solution is to convert the given tree into the AVL tree i.e by using the Left Rotation,Right Rotation, L-R Rotation and R-L Rotation.
This will take time complexity of (N Log N).
(This is not a good approach as writing all the balancing algorithm of AVL Tree is very hectic.)

METHOD 2 (Efficient) :
An Efficient Solution can construct balanced BST in O(n) time with minimum possible height. Below are steps.

1) Traverse given BST in inorder and store result in an array. This step takes O(n) time. Note that this array would be sorted as inorder traversal of BST always produces sorted sequence.
2) Build a balanced BST from the above created sorted array using the recursive approach discussed here. This step also takes O(n) time as we traverse every element exactly once and processing an element takes O(1) time.

CODE FOR THE EFFICIENT APPROACH :(THE FUNCTIONS WRITTEN IN RED BOLD ARE THE MAIN FUNCTIONS)

```
#include<iostream>
#include<string>
#include<vector>
#include<queue>
using namespace std;

class node{
    public:
    int data;
    node *left=NULL;
    node *right=NULL;
};

class BinarySearchTree{
    public:
    node *head=NULL;
    public:
    BinarySearchTree(vector<string> &initialisation){
        queue<node*> q;
        node *root=new node;
        root->data=stoi(initialisation[0]);
        q.push(root);
        int k=1;
        this->head=root;
        while(q.size()!=0 && k+1<initialisation.size()){
            if(initialisation[k]=="N" && initialisation[k+1]=="N"){
                q.pop();
            }
            else if(initialisation[k]=="N" && initialisation[k+1]!="N"){
```

```
            node *temp=new node;
            temp->data=stoi(initialisation[k+1]);
            q.front()->right=temp;
            q.push(temp);
            q.pop();
         }
         else if(initialisation[k]!="N" && initialisation[k+1]=="N"){
            node *temp=new node;
            temp->data=stoi(initialisation[k]);
            q.front()->left=temp;
            q.push(temp);
            q.pop();
         }
         else{
            node *temp1=new node;
            node *temp2=new node;
            temp1->data=stoi(initialisation[k]);
            temp2->data=stoi(initialisation[k+1]);
            q.front()->left=temp1;
            q.front()->right=temp2;
            q.push(temp1);
            q.push(temp2);
            q.pop();
         }
         k=k+2;
      }
      if(k+1==initialisation.size() && q.size()!=0 && initialisation[k]!="N"){
         node *temp=new node;
         temp->data=stoi(initialisation[k]);
         q.front()->left=temp;
      }
   }
   void InOrderTraversal(node *root,vector<int> &v){
      if(root!=NULL){
         InOrderTraversal(root->left,v);
         v.push_back(root->data);
         InOrderTraversal(root->right,v);
      }
   }
   friend int main();
};

node *BalancedBST(vector<int> &v,int low,int high){
   if(low>high){
      return NULL;
   }
   else{
      int mid=(high+low)/2;
      node *t=new node;
      t->data=v[mid];
```

```
        t->left=BalancedBST(v,low,mid-1);
        t->right=BalancedBST(v,mid+1,high);
        return t;
    }
}

void peek(node *root){
    if(root!=NULL){
        peek(root->left);
        cout<<root->data<<" ";
        peek(root->right);
    }
}

node *function(vector<int> &v){
    int n=v.size();
    node *root=BalancedBST(v,0,n-1);
    return root;
}

int main(){
    int n;
    cout<<"\n Enter the number of nodes present in the Binary Search Tree:";
    cin>>n;
    vector<string> initialisation(n);
    cout<<"\n Enter the node values of the Binary Search Tree:";
    for(int i=0;i<n;i++){
        cin>>initialisation[i];
    }
    BinarySearchTree bst(initialisation);

    vector<int> v;
    bst.InOrderTraversal(bst.head,v);
    node *root=function(v);

    cout<<"\n The InOrder Traversal of the balanced BST : ";
    peek(root);

    return 0;
}
```