```
/*
Since the CPP file format was not supported by the uploading website, here I
am sending the program in a PDF file.
*/
/*
Suppose we are given an integer array nums and want to return a new counts
array. The counts array has the property where counts[i] is the number of
smaller elements to the right of nums[i].

Example:

Input: [5,2,6,1]
Output: [2,1,1,0]
Explanation:
To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
------------------------------------------------------------
Solution:
I solved this problem with the help of a helper data structure: AVL Trees.
So, the core idea is to make the initial array into a pair of its value and
its index: vec.push_back(make_pair(nums[i],i))
Then sort this vector based on its first number. Then, start from the
beginning of this vector and insert things into the AVL tree for the indexes
and update the height and nodes on the left and nodes on the right for the
tree and update the relevant nodes in the tree after eash insertion. This
will take log n for the tree to complete. Also we will search for the node
that we just inserted and we count the number of nodes on the right of each
node, since those were smaller nodes that were added before the current node
and should be counted as the answer for that node. And then just put the
number into the index that we just entered to the tree.

What I want to show you in this code is:
1) I correctly identify the algorithm.
2) I use correct data structures and can manipulate them based on the needs.
3) I can define my own data structures.
4) The flow of the code I wrote is readable and understandable.

*/
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

class Smallers {
public:
    struct comp{
        template<typename T>
        bool operator()(const T& e1, const T& e2)const{
            return e1>e2;
        }
    };
    class Node {
        public:
        int key;
        Node *left;
        Node *right;
        int heightL = 0;
        int heightR = 0;
        int height;
    };
    vector<int> countSmaller(vector<int>& nums) {
        vector<int> ret(nums.size(),0);
        vector<pair<int,int>> vec;
        for (int i = 0;i<nums.size();i++){
            vec.push_back(make_pair(nums[i],i));
        }
        sort(vec.begin(),vec.end());
        int count = 0;
        Node *st = NULL;
        for (int i = 0;i<vec.size();i++){
            // cout<<"i::"<<vec[i].second<<" :: "<<vec[i].first<<endl;
            st = insert(st, vec[i].second);
            ret[vec[i].second] = dfs(st,vec[i].second);
            // cout<<endl;
        }

        // pr(st);

        return ret;
    }
    void pr(Node* root){
        if (root==NULL)
            return;
        pr(root->left);
        cout<<" , "<< root->key << " :L: " << root->heightL << " :R: "<<
root->heightR << " , ";
        pr(root->right);
    }
    int dfs(Node* root, int search){
        if (root->key == search){
            return (root->heightR);
        }
        if (root->key < search){
            return dfs(root->right, search);
        }
        if (root->key > search){
            return root->heightR + 1 + dfs(root->left, search);
        }
        return 0;
    }



    // A utility function to get the
    // height of the tree
    int height(Node *N)
    {
        if (N == NULL)
            return 0;
        return N->height;
    }
    int heightL(Node *N)
```

```
{
    if (N == NULL)
        return 0;
    return N->heightL;
}
int heightR(Node *N)
{
    if (N == NULL)
        return 0;
    return N->heightR;
}


/* Helper function that allocates a
   new node with the given key and
   NULL left and right pointers. */
Node* newNode(int key)
{
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially
                      // added at leaf
    return node;
}

// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
Node *rightRotate(Node *y)
{
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
                    height(y->right)) + 1;
    x->height = max(height(x->left),
                    height(x->right)) + 1;

    y->heightL = (T2)? heightL(T2) + heightR(T2) + 1 : 0;
    x->heightR = (y)? heightL(y) + heightR(y) + 1 : 0;
    // Return new root
    return x;
}

// A utility function to left
// rotate subtree rooted with x
// See the diagram given above.
Node *leftRotate(Node *x)
{
    Node *y = x->right;
```

```
        Node *T2 = y->left;

        // Perform rotation
        y->left = x;
        x->right = T2;

        // Update heights
        x->height = max(height(x->left),
                        height(x->right)) + 1;
        y->height = max(height(y->left),
                        height(y->right)) + 1;

        x->heightR = (T2)? heightL(T2) + heightR(T2) + 1 : 0;
        y->heightL = (x)? heightL(x) + heightR(x) + 1 : 0;

        // Return new root
        return y;
    }

    // Get Balance factor of node N
    int getBalance(Node *N)
    {
        if (N == NULL)
            return 0;
        return height(N->left) - height(N->right);
    }

    // Recursive function to insert a key
    // in the subtree rooted with node and
    // returns the new root of the subtree.
    Node* insert(Node* node, int key)
    {
        /* 1. Perform the normal BST insertion */
        if (node == NULL)
            return(newNode(key));

        if (key < node->key)
            node->left = insert(node->left, key);
        else if (key > node->key)
            node->right = insert(node->right, key);
        else // Equal keys are not allowed in BST
            return node;

        /* 2. Update height of this ancestor node */
        node->height = 1 + max(height(node->left),
                            height(node->right));
        /* 2. Update height of this ancestor node */
        node->heightL = (node->left)? 1 + heightL(node->left) + heightR(node-
>left): 0;
        /* 2. Update height of this ancestor node */
        node->heightR = (node->right)? 1 + heightL(node->right) +
heightR(node->right): 0;


        /* 3. Get the balance factor of this ancestor
            node to check whether this node became
            unbalanced */
        int balance = getBalance(node);

        // If this node becomes unbalanced, then
        // there are 4 cases

        // Left Left Case
        if (balance > 1 && key < node->left->key)
            return rightRotate(node);

        // Right Right Case
        if (balance < -1 && key > node->right->key)
            return leftRotate(node);

        // Left Right Case
        if (balance > 1 && key > node->left->key)
        {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }

        // Right Left Case
        if (balance < -1 && key < node->right->key)
        {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }

        /* return the (unchanged) node pointer */
        return node;
    }
};


/*
The sole purpose of the following main, is to showcase the class.
*/
int main(){
      vector<int> v = {5,2,6,1};
      Smallers* obj = new Smallers();
      vector<int> res = obj->countSmaller(v);
      cout<<"The vector::"<<endl;
      for (int i = 0;i<v.size()-1;i++){
            cout<<v[i]<<",";
      }
      cout<<v.back()<<endl;

      cout<<"Has the following count smaller for the elements (in the same
order as the array)::"<<endl;

      for (int i = 0;i<res.size()-1;i++){
            cout<<res[i]<<",";
      }
      cout<<res.back()<<endl;
      return 0;
}
```