Count smaller elements on right side

Difficulty Level : Hard • Last Updated : 11 Jul, 2022

Write a function to count number of smaller elements on right of each element in an array. Given an unsorted array arr[] of distinct integers, construct another array countSmaller[] such that countSmaller[i] contains count of smaller elements on right side of each element arr[i] in array.

Examples:

```
Input:   arr[] =  {12, 1, 2, 3, 0, 11, 4}
Output:  countSmaller[]  =  {6, 1, 1, 1, 0, 1, 0}

(Corner Cases)
Input:   arr[] =  {5, 4, 3, 2, 1}
Output:  countSmaller[]  =  {4, 3, 2, 1, 0}

Input:   arr[] =  {1, 2, 3, 4, 5}
Output:  countSmaller[]  =  {0, 0, 0, 0, 0}
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Simple)

element and updates countSmaller[].

C++

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

void constructLowerArray(int arr[], int *countSmaller,
                         int n)
{
    int i, j;

    // Initialize all the counts in
    // countSmaller array as 0
    for(i = 0; i < n; i++)
        countSmaller[i] = 0;

    for(i = 0; i < n; i++)
    {
        for(j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[i])
                countSmaller[i]++;
        }
    }
}

// Utility function that prints
// out an array on a line
void printArray(int arr[], int size)
{
    int i;
    for(i = 0; i < size; i++)
        cout << arr[i] << " ";

    cout << "\n";
}

// Driver code
int main()
```

```
    int *low = (int *)malloc(sizeof(int)*n);

    constructLowerArray(arr, low, n);
    printArray(low, n);

    return 0;
}

// This code is contributed by Hemant Jain
```

C

```
void constructLowerArray (int *arr[], int *countSmaller, int n)
{
  int i, j;

  // initialize all the counts in countSmaller array as 0
  for   (i = 0; i < n; i++)
     countSmaller[i] = 0;

  for (i = 0; i < n; i++)
  {
    for (j = i+1; j < n; j++)
    {
       if (arr[j] < arr[i])
         countSmaller[i]++;
    }
  }
}

/* Utility function that prints out an array on a line */
void printArray(int arr[], int size)
{
  int i;
  for (i=0; i < size; i++)
    printf("%d ", arr[i]);

  printf("\n");
}
```

```
  int arr[] = {12, 10, 5, 4, 2, 20, 6, 1, 0, 2};
  int n = sizeof(arr)/sizeof(arr[0]);
  int *low = (int *)malloc(sizeof(int)*n);
  constructLowerArray(arr, low, n);
  printArray(low, n);
  return 0;
}
```

Java

```
class CountSmaller
{
    void constructLowerArray(int arr[], int countSmaller[], int n)
    {
        int i, j;

        // initialize all the counts in countSmaller array as 0
        for (i = 0; i < n; i++)
            countSmaller[i] = 0;

        for (i = 0; i < n; i++)
        {
            for (j = i + 1; j < n; j++)
            {
                if (arr[j] < arr[i])
                    countSmaller[i]++;
            }
        }
    }

    /* Utility function that prints out an array on a line */
    void printArray(int arr[], int size)
    {
        int i;
        for (i = 0; i < size; i++)
            System.out.print(arr[i] + " ");

        System.out.println("");
    }
```

```
        int arr[] = {12, 10, 5, 4, 2, 20, 6, 1, 0, 2};
        int n = arr.length;
        int low[] = new int[n];
        small.constructLowerArray(arr, low, n);
        small.printArray(low, n);
    }
}
```

Python3

```
def constructLowerArray (arr, countSmaller, n):

    # initialize all the counts in countSmaller array as 0
    for i in range(n):
        countSmaller[i] = 0

    for i in range(n):
        for j in range(i + 1,n):
            if (arr[j] < arr[i]):
                countSmaller[i] += 1

# Utility function that prints out an array on a line
def printArray(arr, size):
    for i in range(size):
        print(arr[i],end=" ")
    print()

# Driver code
arr = [12, 10, 5, 4, 2, 20, 6, 1, 0, 2]
n = len(arr)
low = [0]*n
constructLowerArray(arr, low, n)
printArray(low, n)

# This code is contributed by ApurvaRaj
```

C#

```
static void constructLowerArray(int []arr,
                int []countSmaller, int n)
{
    int i, j;

    // initialize all the counts in
    // countSmaller array as 0
    for (i = 0; i < n; i++)
        countSmaller[i] = 0;

    for (i = 0; i < n; i++)
    {
        for (j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[i])
                countSmaller[i]++;
        }
    }
}

/* Utility function that prints out
an array on a line */
static void printArray(int []arr, int size)
{
    int i;
    for (i = 0; i < size; i++)
        Console.Write(arr[i] + " ");

    Console.WriteLine("");
}

// Driver function
public static void Main()
{
    int []arr = new int[]{12, 10, 5, 4,
                          2, 20, 6, 1, 0, 2};
    int n = arr.Length;
    int []low = new int[n];

    constructLowerArray(arr, low, n);
    printArray(low, n);
```

Javascript

```
<script>
    function constructLowerArray(arr, countSmaller, n)
    {
        let i, j;

        // initialize all the counts in
        // countSmaller array as 0
        for (i = 0; i < n; i++)
            countSmaller[i] = 0;

        for (i = 0; i < n; i++)
        {
            for (j = i + 1; j < n; j++)
            {
                if (arr[j] < arr[i])
                    countSmaller[i]++;
            }
        }
    }

    /* Utility function that prints out
    an array on a line */
    function printArray(arr, size)
    {
        let i;
        for (i = 0; i < size; i++)
            document.write(arr[i] + " ");

        document.write("</br>");
    }

    let arr = [12, 10, 5, 4, 2, 20, 6, 1, 0, 2];
    let n = arr.length;
    let low = new Array(n);

    constructLowerArray(arr, low, n);
```

Output:

```
8 7 5 4 2 4 3 1 0 0
```

Time Complexity: O(n^2)
Auxiliary Space: O(1)

Method 2 (Use Self Balancing BST)

A Self Balancing Binary Search Tree (AVL, Red Black,.. etc) can be used to get the solution in O(nLogn) time complexity. We can augment these trees so that every node N contains size the subtree rooted with N. We have used AVL tree in the following implementation.
We traverse the array from right to left and insert all elements one by one in an AVL tree. While inserting a new key in an AVL tree, we first compare the key with root. If key is greater than root, then it is greater than all the nodes in left subtree of root. So we add the size of left subtree to the count of smaller element for the key being inserted. We recursively follow the same approach for all nodes down the root.

Following is the C implementation.

C++

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

#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
```

```
    // with this node
    int size;
};

// A utility function to get
// maximum of two integers
int max(int a, int b);

// A utility function to get
// height of the tree rooted with N
int height(struct node *N)
{
    if (N == NULL)
        return 0;

    return N->height;
}

// A utility function to size
// of the tree of rooted with N
int size(struct node *N)
{
    if (N == NULL)
        return 0;

    return N->size;
}

// A utility function to
// get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

// Helper function that allocates a
// new node with the given key and
// NULL left and right pointers.
struct node* newNode(int key)
{
    struct node* node = (struct node*)
```

```
    // New node is initially added at leaf
    node->height = 1;
    node->size = 1;
    return(node);
}

// A utility function to right rotate
// subtree rooted with y
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct 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;

    // Update sizes
    y->size = size(y->left) + size(y->right) + 1;
    x->size = size(x->left) + size(x->right) + 1;

    // Return new root
    return x;
}

// A utility function to left rotate
// subtree rooted with x
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
```

```
                    height(x->right)) + 1;
    y->height = max(height(y->left),
                    height(y->right)) + 1;

    // Update sizes
    x->size = size(x->left) + size(x->right) + 1;
    y->size = size(y->left) + size(y->right) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;

    return height(N->left) - height(N->right);
}

// Inserts a new key to the tree rotted with
// node. Also, updates *count to contain count
// of smaller elements for the new key
struct node* insert(struct node* node, int key,
                    int *count)
{
    // 1. Perform the normal BST rotation
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key, count);
    else
    {
        node->right = insert(node->right, key, count);

        // UPDATE COUNT OF SMALLER ELEMENTS FOR KEY
        *count = *count + size(node->left) + 1;
    }
```

```
        node->size   = size(node->left) +
                       size(node->right) + 1;

        // 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 following function updates the
    // countSmaller array to contain count of
    // smaller elements on right side.
    void constructLowerArray(int arr[], int countSmaller[],
```

```
    // Initialize all the counts in
    // countSmaller array as 0
    for(i = 0; i < n; i++)
        countSmaller[i] = 0;

    // Starting from rightmost element,
    // insert all elements one by one in
    // an AVL tree and get the count of
    // smaller elements
    for(i = n - 1; i >= 0; i--)
    {
        root = insert(root, arr[i], &countSmaller[i]);
    }
}

// Utility function that prints out an
// array on a line
void printArray(int arr[], int size)
{
    int i;
    cout << "\n";

    for(i = 0; i < size; i++)
        cout << arr[i] <<" ";
}

// Driver code
int main()
{
    int arr[] = {10, 6, 15, 20, 30, 5, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    int *low = (int *)malloc(sizeof(int)*n);

    constructLowerArray(arr, low, n);

    cout <<"Following is the constructed smaller count array";
    printArray(low, n);

    return 0;
```

```
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
    int size; // size of the tree rooted with this node
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree rooted with N
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to size of the tree of rooted with N
int size(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->size;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
    NULL left and right pointers. */
```

```
    node->key   = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1;  // new node is initially added at leaf
    node->size = 1;
    return(node);
}

// A utility function to right rotate subtree rooted with y
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct 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;

    // Update sizes
    y->size = size(y->left) + size(y->right) + 1;
    x->size = size(x->left) + size(x->right) + 1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

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

    //  Update heights
```

```
    x->size = size(x->left) + size(x->right) + 1;
    y->size = size(y->left) + size(y->right) + 1;

    // Return new root
    return y;
}

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

// Inserts a new key to the tree rotted with node. Also, updates *count
// to contain count of smaller elements for the new key
struct node* insert(struct node* node, int key, int *count)
{
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key, count);
    else
    {
        node->right = insert(node->right, key, count);

        // UPDATE COUNT OF SMALLER ELEMENTS FOR KEY
        *count = *count + size(node->left) + 1;
    }


    /* 2. Update height and size of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;
    node->size   = size(node->left) + size(node->right) + 1;

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

```
    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 following function updates the countSmaller array to contain count
// smaller elements on right side.
void constructLowerArray (int arr[], int countSmaller[], int n)
{
  int i, j;
  struct node *root = NULL;

  // initialize all the counts in countSmaller array as 0
  for   (i = 0; i < n; i++)
     countSmaller[i] = 0;

  // Starting from rightmost element, insert all elements one by one in
  // an AVL tree and get the count of smaller elements
  for (i = n-1; i >= 0; i--)
  {
     root = insert(root, arr[i], &countSmaller[i]);
```

```
void printArray(int arr[], int size)
{
  int i;
  printf("\n");
  for (i=0; i < size; i++)
    printf("%d ", arr[i]);
}

// Driver program to test above functions
int main()
{
  int arr[] = {10, 6, 15, 20, 30, 5, 7};
  int n = sizeof(arr)/sizeof(arr[0]);

  int *low = (int *)malloc(sizeof(int)*n);

  constructLowerArray(arr, low, n);

  printf("Following is the constructed smaller count array");
  printArray(low, n);
  return 0;
```

Java

```
import java.util.*;

class GFG{

  // An AVL tree node
  static class node
  {
    int key;
    node left;
    node right;
    int height;

    // size of the tree rooted
    // with this node
    int size;
  };
```

```
// height of the tree rooted with N
static int height(node N)
{
  if (N == null)
    return 0;

  return N.height;
}

// A utility function to size
// of the tree of rooted with N
static int size(node N)
{
  if (N == null)
    return 0;

  return N.size;
}

// A utility function to
// get maximum of two integers
static int max(int a, int b)
{
  return (a > b)? a : b;
}

// Helper function that allocates a
// new node with the given key and
// null left and right pointers.
static node newNode(int key)
{
  node node = new node();
  node.key   = key;
  node.left   = null;
  node.right  = null;

  // New node is initially added at leaf
  node.height = 1;
  node.size = 1;
  return(node);
}
```

```
{
  node x = y.left;
  node T2 = x.right;

  // Perform rotation
  x.right = y;
  y.left = T2;

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

  // Update sizes
  y.size = size(y.left) + size(y.right) + 1;
  x.size = size(x.left) + size(x.right) + 1;

  // Return new root
  return x;
}

// A utility function to left rotate
// subtree rooted with x
static node leftRotate(node x)
{
  node y = x.right;
  node T2 = y.left;

  // Perform rotation
  y.left = x;
  x.right = T2;

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

  // Update sizes
  x.size = size(x.left) + size(x.right) + 1;
```

```
}

// Get Balance factor of node N
static int getBalance(node N)
{
  if (N == null)
    return 0;

  return height(N.left) - height(N.right);
}

// Inserts a new key to the tree rotted with
// node. Also, updates *count to contain count
// of smaller elements for the new key
static node insert(node node, int key,
                   int count)
{
  // 1. Perform the normal BST rotation
  if (node == null)
    return(newNode(key));

  if (key < node.key)
    node.left  = insert(node.left, key, count);
  else
  {
    node.right = insert(node.right, key, count);

    // UPDATE COUNT OF SMALLER ELEMENTS FOR KEY
    countSmaller[count] = countSmaller[count] + size(node.left) + 1;
  }


  // 2.Update height and size of this ancestor node
  node.height = Math.max(height(node.left),
                         height(node.right)) + 1;
  node.size   = size(node.left) +
    size(node.right) + 1;

  // 3. Get the balance factor of this
  // ancestor node to check whether this
  // node became unbalanced
```

```
  // 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 following function updates the
// countSmaller array to contain count of
// smaller elements on right side.
static void constructLowerArray(int arr[],
                                int n)
{
  int i, j;
  node root = null;

  // Initialize all the counts in
  // countSmaller array as 0
  for(i = 0; i < n; i++)
    countSmaller[i] = 0;
```

```
    for(i = n - 1; i >= 0; i--)
    {
      root = insert(root, arr[i],i);
    }
  }

  // Utility function that prints out an
  // array on a line
  static void printArray(int arr[], int size)
  {
    int i;
    System.out.print("\n");

    for(i = 0; i < size; i++)
      System.out.print(arr[i] +" ");
  }

  // Driver code
  public static void main(String[] args)
  {
    int arr[] = {10, 6, 15, 20, 30, 5, 7};
    int n = arr.length;

    countSmaller = new int[n];

    constructLowerArray(arr, n);

    System.out.print("Following is the constructed smaller count array");
    printArray(countSmaller, n);

  }
}
```

C#

```
using System;
public class GFG {

  // An AVL tree node
  public class node {
```

```
  // size of the tree rooted
  // with this node
  public int size;
};

static int[] countSmaller;
static int count;

// A utility function to get
// height of the tree rooted with N
static int height(node N) {
  if (N == null)
    return 0;

  return N.height;
}

// A utility function to size
// of the tree of rooted with N
static int size(node N) {
  if (N == null)
    return 0;

  return N.size;
}

// A utility function to
// get maximum of two integers
static int max(int a, int b) {
  return (a > b) ? a : b;
}

// Helper function that allocates a
// new node with the given key and
// null left and right pointers.
static node newNode(int key) {
  node node = new node();
  node.key = key;
  node.left = null;
  node.right = null;
```

```
  return (node);
}

// A utility function to right rotate
// subtree rooted with y
static node rightRotate(node y) {
  node x = y.left;
  node T2 = x.right;

  // Perform rotation
  x.right = y;
  y.left = T2;

  // Update heights
  y.height = Math.Max(height(y.left), height(y.right)) + 1;
  x.height = Math.Max(height(x.left), height(x.right)) + 1;

  // Update sizes
  y.size = size(y.left) + size(y.right) + 1;
  x.size = size(x.left) + size(x.right) + 1;

  // Return new root
  return x;
}

// A utility function to left rotate
// subtree rooted with x
static node leftRotate(node x) {
  node y = x.right;
  node T2 = y.left;

  // Perform rotation
  y.left = x;
  x.right = T2;

  // Update heights
  x.height = Math.Max(height(x.left), height(x.right)) + 1;
  y.height = Math.Max(height(y.left), height(y.right)) + 1;

  // Update sizes
  x.size = size(x.left) + size(x.right) + 1;
```

```
}

// Get Balance factor of node N
static int getBalance(node N) {
  if (N == null)
    return 0;

  return height(N.left) - height(N.right);
}

// Inserts a new key to the tree rotted with
// node. Also, updates *count to contain count
// of smaller elements for the new key
static node insert(node node, int key, int count) {
  // 1. Perform the normal BST rotation
  if (node == null)
    return (newNode(key));

  if (key < node.key)
    node.left = insert(node.left, key, count);
  else {
    node.right = insert(node.right, key, count);

    // UPDATE COUNT OF SMALLER ELEMENTS FOR KEY
    countSmaller[count] = countSmaller[count] + size(node.left) + 1;
  }

  // 2.Update height and size of this ancestor node
  node.height = Math.Max(height(node.left), height(node.right)) + 1;
  node.size = size(node.left) + size(node.right) + 1;

  // 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 && kev < node.left.kev)
```

```
    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 following function updates the
// countSmaller array to contain count of
// smaller elements on right side.
static void constructLowerArray(int []arr, int n) {
  int i, j;
  node root = null;

  // Initialize all the counts in
  // countSmaller array as 0
  for (i = 0; i < n; i++)
    countSmaller[i] = 0;

  // Starting from rightmost element,
  // insert all elements one by one in
  // an AVL tree and get the count of
  // smaller elements
  for (i = n - 1; i >= 0; i--) {
    root = insert(root, arr[i], i);
  }
}

// Utility function that prints out an
// array on a line
```

```
    for (i = 0; i < size; i++)
      Console.Write(arr[i] + " ");
  }

  // Driver code
  public static void Main(String[] args) {
    int []arr = { 10, 6, 15, 20, 30, 5, 7 };
    int n = arr.Length;

    countSmaller = new int[n];

    constructLowerArray(arr, n);

    Console.Write("Following is the constructed smaller count array");
    printArray(countSmaller, n);

  }
}
```

Output:

```
Following is the constructed smaller count array
3 1 2 2 2 0 0
```

Time Complexity: O(nLogn)
Auxiliary Space: O(n)

Method 3 (Using BST with 2 extra fields)

Another approach to solve the above problem would be to use a simple Binary Search Tree with 2 extra fields:
1) to hold the elements on the left side of a node
2) to store the frequency of element.

In this approach, we traverse the input array from the ending to the begging and add the elements into the BST.

current node, if we are moving to right side of the current node.

Once we place an element in it's correct position, we can return it's this sum value

---

C++14

```
#include<bits/stdc++.h>
using namespace std;

// BST node structure
class Node{

public:
    int val;
    int count;
    Node* left;
    Node* right;

    // Constructor
    Node(int num1, int num2)
    {
        this->val = num1;
        this->count = num2;
        this->left = this->right = NULL;
    }
};

// Function to addNode and find the smaller
// elements on the right side
int addNode(Node*& root, int value,
                         int countSmaller)
{

    // Base case
    if (root == NULL)
    {
        root = new Node(value, 0);
        return countSmaller;
```

```
        addNode(root->right,
                value,
                countSmaller + 1);
    }
    else
    {
        root->count++;
        return addNode(root->left, value,
                       countSmaller);
    }
}

// Driver code
int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    int data[] = { 10, 6, 15, 20, 30, 5, 7 };
    int size = sizeof(data) / sizeof(data[0]);
    int ans[size] = {0};

    Node* root = NULL;

    for(int i = size - 1; i >= 0; i--)
    {
        ans[i] = addNode(root, data[i], 0);
    }

    for(int i = 0; i < size; i++)
        cout << ans[i] << " ";

    return 0;
}

// This code is contributed by divyanshu gupta
```

Python3

```
class Node:
    def __init__(self,val):
```

```
            # denotes number of times (frequency)
            # an element has occurred.
            self.elecount = 1

            # denotes the number of nodes on left
            # side of the node encountered so far.
            self.lcount = 0

    class Tree:
        def __init__(self,root):
            self.root = root
        def insert(self,node):

            """This function helps to place an element at
                its correct position in the BST and returns
                the count of elements which are smaller than
                the elements which are already inserted into the BST.
            """
            curr = self.root
            cnt = 0
            while curr!=None:
                prev = curr
                if node.val>curr.val:

                    # This step computes the number of elements
                    # which are less than the current Node.
                    cnt += (curr.elecount+curr.lcount)
                    curr=curr.right
                elif node.val<curr.val:
                    curr.lcount+=1
                    curr=curr.left
                else:
                    prev=curr
                    prev.elecount+=1
                    break
            if prev.val>node.val:
                prev.left = node
            elif prev.val<node.val:
                prev.right = node
            else:
                return cnt+prev.lcount
```

```
    ans = [0]
    for i in range(n-2,-1,-1):
        ans.append(t.insert(Node(arr[i])))
    return reversed(ans)

# Driver function for above code
def main():
    n = 7
    arr = [10, 6, 15, 20, 30, 5, 7]
    print(" ".join(list(map(str,constructArray(arr,n)))))
if __name__ == "__main__":
    main()

# Code Contributed by Tarun Gudipati
```

Output:

```
3 1 2 2 2 0 0
```

Time Complexity: $O(n^2)$ as add step can take $O(n)$ time.

Auxiliary Space: $O(n)$

Count smaller elements on right side using Set in C++ STL

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Count smaller elements on right side using Set in C++ STL

Count smaller elements on right side and greater elements on left side using Binary Index Tree

RECOMMENDED ARTICLES

Page : 1 2 3

01 Count smaller elements on right side and greater elements on left side using Binary Index Tree
09, Aug 19

02 Count smaller elements on right side using Set in C++ STL
06, May 18

03 Count of smaller elements on right side of each element in an Array using Merge sort
04, Feb 20

04 Count array elements having at least one smaller element on its left and right side
09, Nov 20

05 Find the farthest smaller number in the right side
17, Sep 19

06 Delete array elements which are smaller than next or become smaller
22, Mar 18

07 Count the number of elements which are greater than any of element on right side of an array
02, Sep 19

08 Count of larger elements on right side of each element in an array
26, Jan 21

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Hard

Easy | Normal | Medium | Hard | Expert

Improved By : Sam007, TarunGudipati, ApurvaRaj, DivyanshuGupta2, hemantjain99, decode2207, amartyaghoshgfg, Rajput-Ji, v3nom

Article Tags : AVL-Tree, Self-Balancing-BST, Arrays

Practice Tags : Arrays, AVL-Tree

Improve Article | Report Issue

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

A-143, 9th Floor, Sovereign Corporate Tower,
Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org