Count inversions in an array | Set 2 (Using Self-Balancing BST)

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

```
    Two elements a[i] and a[j] form an inversion if
     a[i] > a[j] and i < j. For simplicity, we may
     assume that all elements are unique.

     Example:
     Input:  arr[] = {8, 4, 2, 1}
     Output: 6
     Given array has six inversions (8,4), (4,2),
     (8,2), (8,1), (4,1), (2,1).
```

We have already discussed Naive approach and Merge Sort based approaches for counting inversions.

Time Complexity of the Naive approach is $O(n^2)$ and that of merge sort based approach is O(n Log n). In this post one more O(n Log n) approach is discussed. The idea is to use Self-Balancing Binary Search Tree like Red-Black Tree, AVL Tree, etc and augment it so that every node also keeps track of number of nodes in right subtree.

```
1) Create an empty AVL Tree.  The tree is augmented here
   such that every node also maintains size of subtree
   rooted with this node.

2) Initialize inversion count or result as 0.

3) Iterate from 0 to n-1 and do following for every
   element in arr[i]
     a) Insert arr[i] into the AVL Tree.  The insertion
        operation also updates result.  The idea is to
        keep counting greater nodes when tree is traversed
        from root to a leaf for insertion.

4) Return result.
```

More explanation for step 3.a:

1) When we insert arr[i], elements from arr[0] to arr[i-1] are already inserted into AVL Tree. All we need to do is count these nodes.
2) For insertion into AVL Tree, we traverse tree from root to a leaf by comparing every node with arr[i[]. When arr[i[is smaller than current node, we increase inversion count by 1 plus the number of nodes in right subtree of current node. Which is basically count of greater elements on left of arr[i], i.e., inversions.

Below is C++ implementation of above idea.

```
// An AVL Tree based C++ program to count inversion in an array
#include<bits/stdc++.h>
using namespace std;

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

// 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;
}

/* 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 = new Node;
    node->key   = key;
    node->left   = node->right  = NULL;
    node->height = 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->height = max(height(x->left), 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);
}

// Recursive function to insert a key in the subtree rooted with Node and
```

```
// Inserts a new key to the tree rotted with Node. Also, updates
// *result (inversion count)
struct Node* insert(struct Node* node, int key, int *result)
{
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
    {
        node->left  = insert(node->left, key, result);

        // UPDATE COUNT OF GREATE ELEMENTS FOR KEY
        *result = *result + size(node->right) + 1;
    }
    else
        node->right = insert(node->right, key, result);

    /* 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 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 returns inversion count in arr[]
int getInvCount(int arr[], int n)
{
  struct Node *root = NULL;  // Create empty AVL Tree

  int result = 0;   // Initialize result

  // Starting from first element, insert all elements one by
  // one in an AVL tree.
  for (int i=0; i<n; i++)

     // Note that address of result is passed as insert
     // operation updates result by adding count of elements
     // greater than arr[i] on left of arr[i]
     root = insert(root, arr[i], &result);

  return result;
```

```
}

// Driver program to test above
int main()
{
    int arr[] = {8, 4, 2, 1};
    int n = sizeof(arr)/sizeof(int);
    cout << "Number of inversions count are : "
         << getInvCount(arr,n);
    return 0;
}
```

Output:

```
Number of inversions count are : 6
```

Time complexity of above solution is O(n Log n) as AVL insert takes O(Log n) time.

Counting Inversions using Set in C++ STL.

We will soon be discussing Binary Indexed Tree based approach for the same.