Advanced Data Structures and Algorithms Assignment II

Venkatesh E
Indian Institute of Technology
Hyderabad
AI20MTECH14005
ai20mtech14005@iith.ac.in

Abstract

This assignment provides approach to the dynamic version of 2-SUM problem. It also discusses about the choice of data structure along with the reasoning with explanation of both time and space complexity.

OS : MAC
Compiler : g++

1 PROBLEM DESCRIPTION

Consider the 2-SUM problem: given an array of n integers (possibly with repetitions), and a target integer t find if there exist two distinct elements x, y in the array such that **x + y = t**. There are **multiple approaches** to find a **O(nlogn)** solution to this problem. We would like to implement a **dynamic version** of this problem. In this version, you do not know the number of elements in advance. The input arrives as a sequence of operations.

We start with the empty multiset S. Operations are of three types:

1. Insert(k) : Inserts a number k into S.

2. Delete(k): Deletes an instance of the key k from S. If k is not present, no change is made to the data structure. If k is present multiple times, any one instance is deleted.

3. Query (a, b): Prints the number of target values in the closed interval [a, b] such that there are distinct elements x, y in the multiset with x + y = t

Space Constraint : Program should use **atmost O(n) space at any point of time**, where n is the number of elements in the multiset at that point in time

2 APPROACH

Choice of Data Structure : Balanced Binary Search Tree $\longrightarrow$AVL Tree

Since no in-built functions were used, the program has the following functions for performing the operations of insert,delete and query.

Some of the basic concepts which were used to explain concepts of insertion and deletion are as follows:

AVL Trees are self-balanced trees :
i) AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

$$b = H_L(root) - H_R(root)$$

where b is tree balancing factor and $H_L(root) and H_R(root)$ were the height of the left and right subtree from the root node thats being passed.

ii) If b is other than -1,0,1 then some rotations will have to be done inorder to get the tree balanced.

a) We perform the left right rotation when there is a imbalance in the left child of right subtree.

b) We perform the right left rotation when there is a imblance in the right child of left subtree.

c) We perform the right rotation when there is an imbalance in the left child of left subtree.

d) We perform the left rotation when there is an imbalance in the right child of right subtree.

2.1 Why AVL Tree ?

Given that input arrives as a sequence of operation and number of elements is not known in advance. In this case data structures that we could think of were

Dynamic array,Hash Tables,Linked List,Binary Search Tree and Balanced Binary Search Tree(AVL Tree).

Data Structure	Insert	Delete
Dynamic Array	O(n)	O(n)
Single Linked List	O(1)	O(n)
Double Linked List	O(1)	O(n)
Hash Tables	O(n)	O(n)
Binary Search Tree	O(n)	O(n)
Balanced Binary Search Tree	O(logn)	O(logn)

Table 1: Worst Case Time complexity of each operations using AVL Tree

Note :

1. In case of Dynamic array and Hash tables we need to extend the size and copy the content and insert new element if the size of input is more than the space that we alloted.So in worst case to insert it takes O(n) time complexity.To delete an element we will have to search the element and delete which will take O(n) in worst case in both dynamic array and hash tables.

2. From the above table 2 using Balanced Binary Search Tree(AVL Tree) will be comparatively better than Single and Double Linked lists as well as Binary Search Tree. Balanced Binary search tree will be efficient for searching the element.

3. Therefore AVL tree is preferred over other data structures .

Operation	Time Complexity
Insert	O(logn)
Delete	O(logn)
Query	O((b-a)nlogn)

Table 2: Time complexity of each operations using AVL Tree

2.2 Time complexity of Insert

To insert an element k, we need to search where to insert the element in the tree. Since AVL is balanced Binary search tree, to insert the element we will search to insert the element till the height of tree and the time complexity will be **O(logn)** if there are n elements in the tree at the given point of time.

Input : root,element
Returns : root

Pseudo Code :

```
Def Insert(root,element):
    If root is Null:
        Create new node
    If element<root->element:
        root->left=Insert(root->left,element)
    Else:
        root->right=Insert(root->right,element)
    Root->height=1+max(height of left subtree,height of right subtree)
    Find balance factor b_f
    If tree is unbalanced perform rotations to make it balanced
    Return root
```

Figure 1: AVL Tree Insert Function Pseudo Code

Algorithm :
1. Let k be the element to be inserted in the AVL tree.
2. Search for the leaf node by start searching from the root such that if the element in the node is greater than k then continue the search on left subtree and if less than k then continue to search on right subtree and insert the element k as leaf nodes right/left child depends on whether the element k is greater/lesser than that node.
3. Once the element k is inserted check for tree imbalance.
4. If the tree doesn't have any imbalance i.e if the tree imbalance factor of each node is -1,0,1 then return the root.
5. If the tree is not properly balanced rotate the tree as stated before and return the root.

For every insertion we will have to check tree balance factor and rotation to be done if tree is not balanced and return the root node.

Time Complexity : O(logn)

2.3 Time complexity of Delete

To delete an element k,we need to search for that element in the tree. Since AVL is balanced Binary search tree,to delete the element we will try to find the element till the height of tree in worst case and the time complexity will be **O(logn)** if there are n elements in the tree at the given point of time.

Input : root,element
Returns : root

Pseudo Code :

```
Def Delete(root,element):
    If root is Null:
        Return root
    If element<root->element:
        root->left=Delete(root->left,element)
    Else if element>root->element:
        root->right=Delete(root->right,element)
    Else:
        This is the node to be deleted and this node could have no child,single child or two child
        If no child :
            Delete the leaf node
        Else if one child :
            Store the single child in temporary node
            Copy temporary node to root node
            Delete temporary node
        Else if two child:
            Get the inorder successor and store it in root node
    If root==Null:
        Return root
    root->height=1+max(height of left subtree,height of right subtree)
    Find balance factor b_f
    If tree is unbalanced perform rotations to make it balanced
    Return root
```

Figure 2: AVL Tree Delete Function Pseudo Code

Algorithm :
1. Let k be the element to be deleted in the AVL tree.
2. Search for the node to be deleted starting from the root such that if the element is present delete it else search in left sub tree if the element in the node is greater than k and search in right sub tree if the element in the node is lesser than k. Delete the node if it is found. Here since numbers can be repetitive ,in this case only one entry will be deleted. If the node is not found till the leaf node then return the root node and no change in tree as the element is not present in the tree.
3. If the element to be deleted is found,then check for tree balance factor and rotation of trees to be done as stated before if the AVL tree is found to be imbalance.

For every deletion, if the element is found we will have to check the tree balance factor and rotation to be done if tree is not balanced and return the root node.

Time Complexity : O(logn)

2.4 Time complexity of Query

To print the number of target values in the closed interval [a, b] such that there are distinct elements x, y in the multiset with $x + y = t$.The time complexity will be **O(nlogn)** for the algorithm which is as follows.

```
Input : root,a,b
Returns : query result

Pseudo Code :

Def Query(root,a,b):
    query_result =0
    n=count number of elements in the tree at this point of time using traversal techniques
    Create an array of size n
    Inorder traversal(arr,root)
    // the above function stores the sorted order of elements to the array arr
    For x in range(a,b+1): (// to include both a and b)
        For i in range(n-1):
            Found=0
            find=x-arr[i]
            Result=BinarySearch(arr,find,start_index=i+1,end_index=n-1)
            If Result==1:
                query_result+=Result
                Break
    Free(arr);
    Return query_result
```

Figure 3: AVL Tree Query Function Pseudo Code

Algorithm :

1. Given closed interval a and b as input and the expected result is number of target values in the closed interval [a,b] such that there are distinct elements x,y in the multiset with x+y=t.Let the result be 0 initially.

2. At this point of time when the query is passed, find the number of elements in the AVL tree and let it be n. To find the number of elements in the AVL tree we will count the number of nodes which will take **O(n)** time.

3. Create the array named sortedarr of size n.

4. Do inorder traversal and store it in that sortedarr. In Binary search tree doing inorder traversal will fetch us sorted order of elements and it can be done in **O(n)** time.

5. For every element x from a to b,iterate the array from index 0 to n-2,

$$search = x - a[index]$$

Now we find whether the element search is found in array from index i+1 to n-1 using Binary Search. If found return 1 else return -1 which indicates element not found. If element is found stop iterating the array and move on to next element x in closed interval [a,b].
6. So for each element x in range of a to b we try to compute whether $x - a[i]$ in array using binary search which in overall takes **O((b-a)nlogn)** time complexity.

Time Complexity : O((b-a)nlogn)

2.5 Results

Operation	Expected Time	Amortized Time	Worst Case Time
Insert	O(logn)	O(1)	O(logn))
Delete	O(logn)	O(1)	O(logn))
Query	O((b-a)nlogn)	O((b-a)nlogn)	O((b-a)nlogn)

Table 3: Expected,amortized and worst case time complexity of each operations

3 PROCEDURE TO COMPILE AND RUN THE CODE

Source Files submitted :

```
// importing the libraries
#include <iostream>
using namespace std;

// create a node
class Tree_Node{
    public:
    int element; // element to be stored in the node
    Tree_Node *left_ptr; // left pointer to the node
    Tree_Node *right_ptr; // right pointer to the node
    int height; // height of the tree from that node i.e max(left
    sub tree,right sub tree)+1 from that node
};

/*
2. Insert function

Input : root (root node) , element(element to be inserted)
Returns : root (root node)

Description : This function takes the root of the tree and element
     to be inserted as input.The below function will insert
```

```
the element into the tree and check whether it is balanced. If it is not balanced it rotates(Left rotation or
Right rotation) the tree depending on the tree structure at that point of time
*/
Tree_Node *Insert(Tree_Node *root,int element);

/*
3. Delete function
Input : root (root node) , element(element to be deleted)
Returns : root (root node)

Description :This function takes the root of the tree and element to be deleted as input. The below function will delete the
element from the tree and replace it with inorder successor and check whether the tree is balanced and if it is
not balanced it will make it balance and return the root node
*/
Tree_Node *Delete(Tree_Node *root,int element);

/*
4. Query function

Input : root(root node),a,b
Output : query_result

Description : This function takes the root node and starting and ending range of numbers (a,b) and returns the
count of numbers in that closed interval(a,b) such that there are distinct elements x, y in the array (array is
created when the query is given as input) with x + y = t.
*/
int query(Tree_Node *root,int a,int b);

/*
5. create_node

Input : element (element to be added to the new node)
Returns : new_node(newly created node having the element that is being passed)

Description : This function takes the input as element and it creates the new node and assign the value as element which is
passed to the function. Here tree node is created where element is stored and left and right pointers are made
NULL and height is initialized to 1 for this new node that is created.
*/
Tree_Node *create_node(int element);
```

```
/*
6. heightOfTheTree

Input : root (root node)
Returns : 0 if the root is NULL and root->height if the root is
    not NULL

This function takes the root node as input and returns the height
    if the root is not null
*/
int heightOfTheTree(Tree_Node *root);

/*
7. max

Input : a ,b
Returns : maximum among a and b

Description : This functions takes the input of integers a and b
    and returns maximum among those two integers
*/
int max(int a,int b);

/*
8. AVLBalanceChecker

Input : root(root node)
Returns : 0 if root is NULL else it returns height of left sub
    tree - height of right sub tree

Description : This function takes the root as input and check
    whether the tree is balanced and it is calculated by
tree_balance=height of left subtree - height of right sub tree
*/
int AVLBalanceChecker(Tree_Node *root);

/*
9. Balancing_Trees

Input : root(root node),tree_balance(output that we got from
    AVLBalanceChecker),element(element to be either inserted or
    delted)
Returns : root (root node) after the tree gets balanced

Description : This function takes the input such as root,tree
    balance factor and element(which could be either used for
insertion or deletion). It performs rotation if the tree is
    unbalanced else it returns the root.This performs
```

```
four types of rotations.Right rotation,right left rotation,left rotation and left right rotation depending on
the situation

*/
Tree_Node *Balancing_Trees(Tree_Node *root,int tree_balance,int element);

/*
10. RotateRight
Input : root (root node)
Returns : updated root after right rotation is done

Description : This function takes the root node as input and Right rotation is done .
*/
Tree_Node *RotateRight(Tree_Node *root);

/*
11. RotateLeft

Input : root (root node)
Returns : updated root after left rotation is done

Description : This function takes the root node as input and Left rotation is done .
*/
Tree_Node *RotateLeft(Tree_Node *root);

/*
12 MinNode

Input : root (root node)
Returns : left most child from the root node

This function takes the root node as input and returns the left most child from the root node being passed
*/
Tree_Node *MinNode(Tree_Node *root);

/*
count is used for counting number of elements in the array and it is used in CountElementsInAVLTree function.
It is initialized to 1 having root node by default. If root node is NULL the function CountElementsInAVLTree
will return 0 else it will use this count into consideration for root node on counting the number of elements
in the tree at that point of time.
*/
```

```c
int number_of_elements=1;
/*
13. CountElementsInAVLTree

Input : root(root node)
Returns : count (number of values in the AVL Tree at that point of
     time)

Description : This function takes the root node as input and it
    returns the number of elements in the array at that point of
time when the function is called.
*/
int CountElementsInAVLTree(Tree_Node *root);

/*
ind is made 0 to store the sorted arr elements while doing the
    inorder traversal.It is resetted to 0 after
inorder traversal is done
*/
int ind=0;
/*
14. InOrderTraversal

Input : root(root node)
Performs : Performs inorder traversal and stores the element in
    the sorted order to the array

Description: This function takes the root node and array as input
    and returns the inorder traversal(returns the sorted array)
at this point of time.
*/
void InOrderTraversal(Tree_Node *root,int arr[]);

/*
15. BinarySearch

Input : arr[](sorted array),find(element to be found),s(start
    index of array),e(end index of array)
Returns : 1 if element found else -1

Description : This function takes sorted array,element to be
    searched(find),first index of array and last index of array as
input and returns 1 if the element is found else it returns -1
*/
int BinarySearch(int arr[],int find,int s,int e);


// creating the insert,delete and query functions for AVL Tree

```

```
// function to insert a number k into AVL Tree
Tree_Node *Insert(Tree_Node *root,int element){
    // If the root is null then insert the element
    if(root==NULL){
        return create_node(element); // this function creates the tree node with value as the element
    }
    if(element<root->element){
        // checking left sub tree to insert the element
        root->left_ptr=Insert(root->left_ptr,element);
    }
    else{
        // checking right sub tree to insert the element
        root->right_ptr=Insert(root->right_ptr,element);
    }
    // updating the height of the tree in root
    root->height=1+max(heightOfTheTree(root->left_ptr), heightOfTheTree(root->right_ptr));
    // check for balancing of tree in left and right side of the tree
    int tree_balance=AVLBalanceChecker(root);
    // if the tree is imbalanced we rotate the tree to make it balanced
    root=Balancing_Trees(root,tree_balance,element);
    return root;
}

//function to delete the instance of given element from the AVL tree
Tree_Node *Delete(Tree_Node *root,int element){
    if(root==NULL){
        return root;
    }
    // search for element to be deleted in left sub tree
    else if(element<root->element){
        root->left_ptr=Delete(root->left_ptr,element);
    }
    // search for element to be deleted in right sub tree
    else if(element>root->element){
        root->right_ptr=Delete(root->right_ptr,element);
    }
    else {
        if ((root->left_ptr == NULL) ||(root->right_ptr == NULL)) {
            Tree_Node *temp = root->left_ptr ? root->left_ptr : root->right_ptr;
            if (temp == NULL) {
                temp = root;
                root = NULL;
```

```
                }
                else{
                    *root = *temp;
                    free(temp);
                }
        }
        else {
            // getting the inorder successor
            Tree_Node *temp = MinNode(root->right_ptr);
            // storing the inorder successor in root node
            root->element = temp->element;
            // delete the inorder successor
            root->right_ptr = Delete(root->right_ptr,temp->element);
        }
    }
    if(root==NULL){
        return root;
    }
    // updating the height of the tree in root
    root->height=1+max(heightOfTheTree(root->left_ptr),heightOfTheTree(root->right_ptr));
    // check for balancing of tree in left and right side of the tree
    int tree_balance=AVLBalanceChecker(root);
    // balance the tree and return the root if it is unbalanced
    root=Balancing_Trees(root,tree_balance,element);
    return root;
}

//query function to print the number of target values in the closed interval [a,b]
int query(Tree_Node *root,int a,int b){
    int query_result=0;
    int n,search;
    //count the number of elements at this given point of query time
    n=CountElementsInAVLTree(root);
    number_of_elements=1;
    //creating the dynamic memory allocation of size n at this point of time when query is executed
    int* sorted_arr = new int[n];
    //doing inorder traversal and sorted array is stored in sorted_arr
    InOrderTraversal(root,sorted_arr);
    ind=0;
    int found;
    for(int x=a;x<=b;x++){
        found=0;
```

```
        for(int i=0;i<n-1;i++){
            search=x-sorted_arr[i];
            found=BinarySearch(sorted_arr,i+1,n-1,search);
            if(found==1){
                query_result+=found;
                break;
            }
        }
    }
    // after the query result is computed the array that we
    created is deleted
    delete[] sorted_arr;
    return query_result;
}

// sub-functions to compute insert,delete and query functions


// creating the new Node
Tree_Node *create_node(int element){
    Tree_Node *new_node=new Tree_Node();
    new_node->element=element;
    new_node->left_ptr=NULL;
    new_node->right_ptr=NULL;
    new_node->height=1;
    return new_node;
}

// function to return max of two integers
int max(int a,int b){
    if(a<b){
        return b;
    }
    else{
        return a;
    }
}

// this function returns the balanced tree
Tree_Node *Balancing_Trees(Tree_Node *root,int tree_balance,int
    element){
    if(tree_balance>1){
        if(element<root->left_ptr->element){
            return RotateRight(root);
        }
        else if(element>root->left_ptr->element){
            root->left_ptr=RotateLeft(root->left_ptr);
            return RotateRight(root);
        }
```

```
    }
    else if(tree_balance<-1){
        if(element<root->right_ptr->element){
            root->right_ptr=RotateRight(root->right_ptr);
            return RotateLeft(root);
        }
        else if(element>root->right_ptr->element){
            return RotateLeft(root);
        }
    }
    return root;
}

// height of the tree from the given root node
int heightOfTheTree(Tree_Node *root){
    return (root==NULL) ? 0: root->height;
}

//check for balanced binary search tree to verify whether tree is balanced or not
int AVLBalanceChecker(Tree_Node *root){
    return (root==NULL) ? 0 : heightOfTheTree(root->left_ptr)-heightOfTheTree(root->right_ptr);
}

// to get the left mode child from the root node passed
Tree_Node *MinNode(Tree_Node *root){
    return ((root==NULL)||(root->left_ptr==NULL)) ? root : MinNode(root->left_ptr);
}

//Rotating the tree left
Tree_Node *RotateLeft(Tree_Node *node){
    Tree_Node *l1=node->right_ptr;
    Tree_Node *l2=l1->left_ptr;
    l1->left_ptr=node;
    node->right_ptr=l2;
    node->height=1+max(heightOfTheTree(node->left_ptr),heightOfTheTree(node->right_ptr));
    l1->height=1+max(heightOfTheTree(l1->left_ptr),heightOfTheTree(l1->right_ptr));
    return l1;
}

// Rotating the tree right
Tree_Node *RotateRight(Tree_Node *node){
    Tree_Node *r1=node->left_ptr;
    Tree_Node *r2=r1->right_ptr;
    r1->right_ptr=node;
```

```
    node->left_ptr=r2;
    node->height=1+max(heightOfTheTree(node->left_ptr),heightOfTheTree(node->right_ptr));
    r1->height=1+max(heightOfTheTree(r1->left_ptr),heightOfTheTree(r1->right_ptr));
    return r1;
}

// count the number of elements in AVL Tree when query is passed
int CountElementsInAVLTree(Tree_Node *root){
    if(root==NULL){
        return 0;
    }
    if(root->left_ptr!=NULL){
        number_of_elements+=1;
        number_of_elements=CountElementsInAVLTree(root->left_ptr);
    }
    if(root->right_ptr!=NULL){
        number_of_elements+=1;
        number_of_elements=CountElementsInAVLTree(root->right_ptr);
    }
    return number_of_elements;
}

// inorder traversal to get the sorted elements till this query point
void InOrderTraversal(Tree_Node *root,int arr[]){
    if(root==NULL){
        return;
    }
    InOrderTraversal(root->left_ptr,arr);
    arr[ind]=root->element;
    ind+=1;
    InOrderTraversal(root->right_ptr,arr);
}

// binary search to find the index for query operation
int BinarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return 1;

        // If element is smaller than mid, then
```

```
        // it can only be present in left subarray
        else if (arr[mid] > x)
            return BinarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return BinarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}


//main function
// 1 . Start from here
int main() {
  Tree_Node *root=NULL; // initialize the root node to null
  // initializing the variables
  char c;
  int k,a,b;
  int out;
  // running the while loops till we see char 'E' as it indicates end of input streaming
  while(c!='E'){
      cin>>c;
      // read the character input which could be 'I','Q','D' Or 'E'
      if(c=='I'){
          cin>>k; // get the element to be inserted
          // calling the insert function and inserting the element k into the tree is done using this function
          root=Insert(root,k);
      }
      else if(c=='Q'){
          cin>>a>>b; // get the closed interval a and b
          out=query(root,a,b);
          cout<<out<<"\n";
      }
      else if(c=='D'){
          cin>>k; // get the element to be deleted
          root=Delete(root,k);
      }
  }
  return 0;
}
```

Listing 1: **C++ Code for Dynamic 2 SUM Problem** makefile - Notepad

File Edit Format View Help

```
CC = g++

program: Assignment2.cpp
	$(CC) Assignment2.cpp
run:
	./a.out
clean:
	rm a.out
```

Figure 4: **makefile**

```
➜ Downloads make
g++ Assignment2.cpp
➜ Downloads make run
./a.out
I
10
I
-21
I
2
I
2
D
-20
D
-21
Q
-20 40
2
Q
100 200
0
E
➜ Downloads
```

Figure 5: **Output**

Compilation command to create the executable : **make**

Run command :**make run**

Output is executed in terminal and it is as shown below in figure 5. Output is highlighted using yellow color.

4 ADDITIONAL INFORMATION

Description of Functions :

1. TreeNode *Insert(TreeNode *root,int element)

Input : root (root node) , element(element to be inserted)
Returns : root (root node)

Description : This function takes the root of the tree and element to be inserted as input.The below function will insert the element into the tree and check whether it is balanced. If it is not balanced it rotates(Left rotation or Right rotation) the tree depending on the tree structure at that point of time

2. TreeNode *Delete(TreeNode *root,int element)

Input : root (root node) , element(element to be deleted)
Returns : root (root node)

Description : This function takes the root of the tree and element to be deleted as input. The below function will delete the element from the tree and replace it with inorder successor and check whether the tree is balanced and if it is not balanced it will make it balance and return the root node

3. int query(TreeNode *root,int a,int b)

Input : root(root node),a,b
Returns : queryresult

Description : This function takes the root node and starting and ending range of numbers (a, b) and returns the count of numbers in that closed interval $[a, b]$ such that there are distinct elements x, y in the array (array is created when the query is given as input) with $x + y = t$.

4. TreeNode *createnode(int element)

Input : element (element to be added to the new node)
Returns : newnode(newly created node having the element that is being passed)

Description : This function takes the input as element and it creates the new node and assign the value as element which is passed to the function. Here tree node is created where element is stored and left and right pointers are made NULL and height is initialized to 1 for this new node that is created.

5. int heightOfTheTree(TreeNode *root)

Input : root (root node)
Returns : 0 if the root is NULL and root-¿height if the root is not NULL

Description : This function takes the root node as input and returns the height if the root is not null.
6. int max(int a,int b)

Input : a ,b
Returns : maximum among a and b

Description : This functions takes the input of integers a and b and returns maximum among those two integers .

7. int AVLBalanceChecker(TreeNode *root)

Input : root(root node)
Returns : 0 if root is NULL else it returns height of left sub tree - height of right sub tree

Description : This function takes the root as input and check whether the tree is balanced and it is calculated by treebalance=height of left subtree - height of right sub tree

8. TreeNode *BalancingTrees(TreeNode *root,int treebalance,int element)

Input : root(root node),treebalance(output that we got from AVLBalanceChecker), element(element to be either inserted or delted)
Returns : root (root node) after the tree gets balanced

Description : This function takes the input such as root,tree balance factor and element(which could be either used for insertion or deletion). It performs rotation if the tree is unbalanced else it returns the root.This performs four types of rotations.Right rotation,right left rotation,left rotation and left right rotation depending on the situation.

9. TreeNode *RotateRight(TreeNode *root)

Input : root (root node)
Returns : updated root after right rotation is done

Description : This function takes the root node as input and Right rotation is done .

10. TreeNode *RotateLeft(TreeNode *root)

Input : root (root node)
Returns : updated root after left rotation is done

Description : This function takes the root node as input and Left rotation is done .

11. TreeNode *MinNode(TreeNode *root)

Input : root (root node)
Returns : left most child from the root node

Description : This function takes the root node as input and returns the left most child from the root node being passed.

12. int CountElementsInAVLTree(TreeNode *root)

Input : root(root node)
Returns : count (number of values in the AVL Tree at that point of time)

Description : This function takes the root node as input and it returns the number of elements in the array at that point of time when the function is called.

13. void InOrderTraversal(TreeNode *root,int arr[])

Input : root(root node)
Performs : Performs inorder traversal and stores the element in the sorted order to the array

Description : This function takes the root node and array as input and returns the inorder traversal(returns the sorted array) at this point of time.

14. int BinarySearch(int arr[],int find,int s,int e)

Input : arr[](sorted array),find(element to be found),s(start index of array),e(end index of array)
Returns : 1 if element found else -1

Description : This function takes sorted array,element to be searched(find),first index of array and last index of array as input and returns 1 if the element is found else it returns -1.

5 References

AVL Tree-Wikipedia
AVL Tree-Concept of rotations
AVL Tree-Insertion and Deletion time complexity