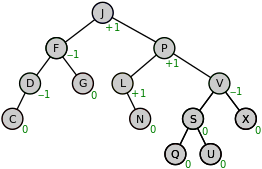
**Ex-1: AVL Trees**

**Program Description:** An AVL tree (Adelson-Velskii and Landis' tree , named after the inventors) is a self- balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node are different by at most one; if at any time they be at variance by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree proceeding to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.



AVL Tree

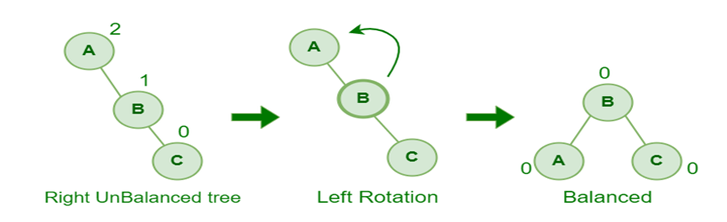
AVL Trees perform self-balancing based on the balance factor (BF). The allowable balance factors are \_1, 0, +1. The mathematical relation for BF is

BF= height of the left subtree - height of the right subtree

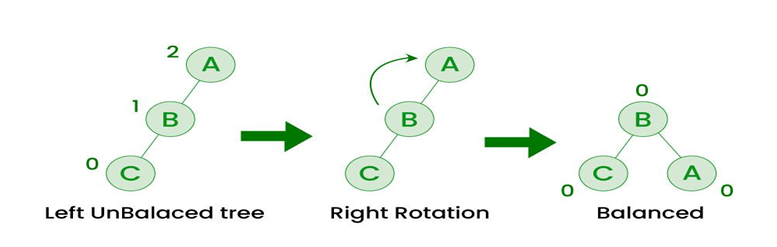
To maintain balance factor, an AVL tree rotate in one of the following four ways:

1. Left rotation (LL) : Single rotation
2. Right rotation (RR) : Single rotation
3. Left-Right rotation (LR): Double rotation
4. Right-Left rotation (RL) : Double rotation

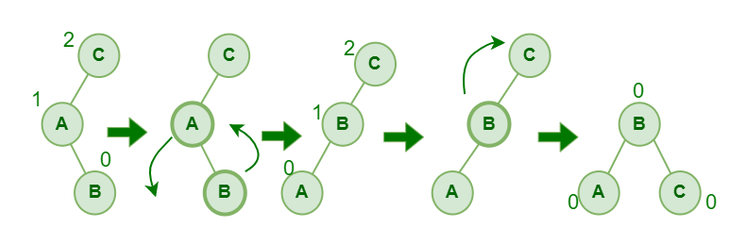
**LL rotation:** When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



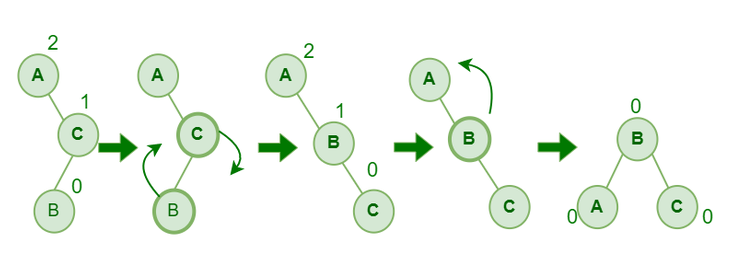
**RR rotation:** If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



**LR rotation:** A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



**RL rotation:** A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



The AVL tree is an abstract data type that contains the following operations:

1. Insertion of a node
2. Deletion of a node
3. Searching a node
4. Displaying the tree nodes
5. **Inserting an element into the AVL Tree:**

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

**Pseudo code for Insertion:**

* Let the newly inserted node be **w** Perform standard**BST** insert for **w**.
* Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y**be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild**of **z** that comes on the path from **w**to **z**.
* Re-balance the tree by performing appropriate rotations on the subtree rooted with**z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
* Following are the possible 4 arrangements:
  + y is the left child of z and x is the left child of y (Left Left Case)
  + y is the left child of z and x is the right child of y (Left Right Case)
  + y is the right child of z and x is the right child of y (Right Right Case)
  + y is the right child of z and x is the left child of y (Right Left Case)

**Algorithm:**

struct Node\* insert(struct 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));

    /\* 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;

}

2.**Deleting an node from the AVL Tree:**

Let w be the node to be deleted

1. Perform standard BST delete for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from [insertion](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/)here.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
   1. y is left child of z and x is left child of y (Left Left Case)
   2. y is left child of z and x is right child of y (Left Right Case)
   3. y is right child of z and x is right child of y (Right Right Case)
   4. y is right child of z and x is left child of y (Right Left Case)

Algorith:

Struct Node\* deleteNode(Struct Node\* root, **int** key)

{

    // STEP 1: PERFORM STANDARD BST DELETE

**if** (root == NULL)

**return** root;

    // If the key to be deleted is smaller

    // than the root's key, then it lies

    // in left subtree

**if** ( key < root->key )

        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater

    // than the root's key, then it lies

    // in right subtree

**else** **if**( key > root->key )

        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then

    // This is the node to be deleted

**else**

    {

        // node with only one child or no child

**if**( (root->left == NULL) ||

            (root->right == NULL) )

        {

            Node \*temp = root->left ?

                         root->left :

                         root->right;

            // No child case

**if** (temp == NULL)

            {

                temp = root;

                root = NULL;

            }

**else** // One child case

            \*root = \*temp; // Copy the contents of

                           // the non-empty child

**free**(temp);

        }

**else**

        {

            // node with two children: Get the inorder

            // successor (smallest in the right subtree)

            Node\* temp = minValueNode(root->right);

            // Copy the inorder successor's

            // data to this node

            root->key = temp->key;

            // Delete the inorder successor

            root->right = deleteNode(root->right,

                                     temp->key);

        }

    }

    // If the tree had only one node

    // then return

**if** (root == NULL)

**return** root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE

    root->height = 1 + max(height(root->left),

                           height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF

    // THIS NODE (to check whether this

    // node became unbalanced)

**int** balance = getBalance(root);

    // If this node becomes unbalanced,

    // then there are 4 cases

    // Left Left Case

**if** (balance > 1 &&

        getBalance(root->left) >= 0)

**return** rightRotate(root);

    // Left Right Case

**if** (balance > 1 &&

        getBalance(root->left) < 0)

    {

        root->left = leftRotate(root->left);

**return** rightRotate(root);

    }

    // Right Right Case

**if** (balance < -1 &&

        getBalance(root->right) <= 0)

**return** leftRotate(root);

    // Right Left Case

**if** (balance < -1 &&

        getBalance(root->right) > 0)

    {

        root->right = rightRotate(root->right);

**return** leftRotate(root);

    }

**return** root;

}

**3. Searching a node:** Search operation as same as BST search

Let’s say we want to search for the number **X,**We start at the root. Then:

* We compare the value to be searched with the value of the root.
  + If it’s equal we are done with the search if it’s smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
* Repeat the above step till no more traversal is possible
* If at any iteration, key is found, return True. Else False.

**Algorithm:**

Struct Node\* searchNode(Struct Node\* root, **int** key)

{

If(root==NULL || root-> value == key)

Return root;

Else

If(kyy < root-> value)

Return searchNode(root->left, key);

Else

Return searchNode(root->right, key);

}

**4. Display AVL Tree nodes:**

**void** Display(**struct** Node \*root)

{

**if**(root != NULL)

    {

**printf**("%d ", root->key);

        Display(root->left);

        Display(root->right);

    }

}

**Program Code:**

Construct AVL tree using vector, lists and perform ADT operations (insertion, deletion, searching, display)

#include <stdio.h> #include <stdlib.h> struct Node {

int key;

struct Node \*left; struct Node \*right; int height;

};

struct Node\* newNode(int key) {

struct Node\* node = (struct Node\*) malloc(sizeof(struct Node)); node->key = key;

node->left = NULL; node->right = NULL; node->height = 1; return node;

}

int height(struct Node\* node) { if (node == NULL)

return 0;

return node->height;

}

int max(int a, int b) { return (a > b) ? a : b;

}

struct Node\* rightRotate(struct Node\* y) { struct Node\* x = y->left;

struct Node\* T2 = x->right; x->right = y;

y->left = T2;

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

}

struct Node\* leftRotate(struct Node\* x) { struct Node\* y = x->right;

struct Node\* T2 = y->left; y->left = x;

x->right = T2;

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

}

int getBalance(struct Node\* node) {

if (node == NULL) return 0;

return height(node->left) - height(node->right);

}

struct Node\* minValueNode(struct Node\* node) { struct Node\* current = node;

while (current->left != NULL) current = current->left;

return current;

}

struct Node\* insert(struct Node\* node, int key) { 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

return node;

node->height = 1 + max(height(node->left), height(node->right)); int balance = getBalance(node);

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

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

if (balance > 1 && key > node->left->key) { node->left = leftRotate(node->left); return rightRotate(node);

}

if (balance < -1 && key < node->right->key) { node->right = rightRotate(node->right); return leftRotate(node);

}

return node;

}

struct Node\* deleteNode(struct Node\* root, int key) { if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key); else if (key > root->key)

root->right = deleteNode(root->right, key); else {

if ((root->left == NULL) || (root->right == NULL)) {

struct Node \*temp = root->left ? root->left : root->right; if (temp == NULL) {

temp = root; root = NULL;

} else

\*root = \*temp; free(temp);

} else {

struct Node\* temp = minValueNode(root->right); root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

}

if (root == NULL) return root;

root->height = 1 + max(height(root->left), height(root->right)); int balance = getBalance(root);

if (balance > 1 && getBalance(root->left) >= 0) return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) { root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && getBalance(root->right) <= 0) return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0) { root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

void Display(struct Node \*root)

{

if(root != NULL){ printf("%4d", root->key); Display(root->left); Display(root->right);

}

}

struct Node\* searchNode(struct Node\* root, int key)

{

if(root==NULL || root->key == key) return root;

else

if(key<root->key)

return searchNode(root->left, key); else if(key>root->key)

return searchNode(root->right, key); else

return NULL;

}

int main() {

struct Node\* root = NULL,\*t; int choice, key;

printf("AVL Tree Operations\n"); printf("1. Insert\n");

printf("2. Delete\n"); printf("3.search\n");

printf("4. Display AVL Tree\n"); printf("5. Exit\n");

do {

printf("Enter your choice: "); scanf("%d", &choice); switch (choice) {

case 1:

printf("Enter key to insert: "); scanf("%d", &key);

root = insert(root, key); break;

case 2:

printf("Enter key to delete: "); scanf("%d", &key);

root = deleteNode(root, key); break;

case 3:

printf("Enter key to search: "); scanf("%d", &key);

t = searchNode(root, key); if (t==NULL)

printf("element not found\n"); else

printf("%d found\n",t->key); break;

case 4:

printf("Preorder of AVL Tree:\n"); Display(root);

printf("\n"); break;

case 5:

printf("Exiting...\n"); break;

default:

printf("Invalid choice! Please enter a valid option.\n");

}

} while (choice != 5); Return 0;

}

**Test input and Output:**

**Application Programs on AVL trees**

1. Construct a AVL tree that maintains a 10 students information. The student information like

1. Student roll number
2. Name
3. Class

Perform a search operation that displays a student information based on roll number.

Use appropriate STL C++ functions for implementation (Hint, Map).

#include <iostream> #include <map> #include <string>

// Define the Student structure struct Student {

std::string name;

std::string class\_name; // Avoid conflict with reserved keywords

// Default constructor

Student() : name(""), class\_name("") {}

// Parameterized constructor

Student(const std::string& n, const std::string& c) : name(n), class\_name(c) {}

};

// Function to display student information

void displayStudent(const Student& student) { std::cout << "Name: " << student.name << std::endl;

std::cout << "Class: " << student.class\_name << std::endl;

}

// Main function int main() {

// Create a map with roll number as key and Student as value std::map<int, Student> studentMap;

// Populate the map with student information studentMap[101] = Student("karan", "10A"); studentMap[102] = Student("vasanthi", "10B"); studentMap[103] = Student("adi", "10C"); studentMap[104] = Student("akshara", "10A"); studentMap[105] = Student("abhi", "10B"); studentMap[106] = Student("chandana", "10C");

studentMap[107] = Student("sathwika", "10A"); studentMap[108] = Student("vinuthna", "10B"); studentMap[109] = Student("akanksha", "10C"); studentMap[110] = Student("rushitha", "10A");

// Function to search and display student by roll number auto searchStudent = [&](int rollNumber) {

auto it = studentMap.find(rollNumber); if (it != studentMap.end()) {

std::cout << "Student found!" << std::endl; displayStudent(it->second);

} else {

std::cout << "Student with roll number " << rollNumber << " not found." << std::endl;

}

};

do{

// User input for roll number int rollNumber;

std::cout << "Enter the roll number to search: "; std::cin >> rollNumber;

// Perform the search operation based on user input searchStudent(rollNumber);

}while(ch!=n)

return 0;

}

2. https://www.hackerrank.com/contests/17cs1102/challenges/10-a-height-of-avl-tree

Implement a program to find the Height of AVL Tree.

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

// Definition of AVL Tree Node typedef struct TreeNode {

int key;

struct TreeNode \*left, \*right; int height;

} TreeNode;

// Function to create a new tree node TreeNode\* createNode(int key) {

TreeNode\* node = (TreeNode\*)malloc(sizeof(TreeNode)); node->key = key;

node->left = node->right = NULL; node->height = 1;

return node;

}

// Function to get the height of a node int height(TreeNode \*node) {

if (node == NULL) return 0;

return node->height;

}

// Function to get the balance factor of a node int getBalance(TreeNode \*node) {

if (node == NULL) return 0;

return height(node->left) - height(node->right);

}

// Right rotation

TreeNode\* rightRotate(TreeNode \*y) { TreeNode \*x = y->left;

TreeNode \*T2 = x->right;

// Perform rotation x->right = y;

y->left = T2;

// Update heights

y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right)); x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

// Return new root return x;

}

// Left rotation

TreeNode\* leftRotate(TreeNode \*x) { TreeNode \*y = x->right;

TreeNode \*T2 = y->left;

// Perform rotation y->left = x;

x->right = T2;

// Update heights

x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right)); y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

// Return new root return y;

}

// Function to insert a key into the AVL tree TreeNode\* insert(TreeNode \*node, int key) {

if (node == NULL)

return createNode(key);

if (key < node->key)

node->left = insert(node->left, key); else if (key > node->key)

node->right = insert(node->right, key); else

return node; // Duplicate keys are not allowed

// Update the height of this node

node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));

// Get the balance factor of this node 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;

}

// Function to calculate the height of the AVL tree int getHeight(TreeNode \*root) {

if (root == NULL) return 0;

return root->height;

}

// Main function int main() {

int N; scanf("%d", &N);

TreeNode \*root = NULL; int key;

for (int i = 0; i < N; i++) { scanf("%d", &key); root = insert(root, key);

}

printf("%d\n", getHeight(root));

return 0;

}

