CLO1, CLO3

Adelson-Velsky and Landis (AVL) Trees

An **AVL Tree** is a special type of **Binary Search Tree (BST)** that keeps itself **balanced automatically**. The AVL tree is named after its inventors: **A**delson-**V**elsky, **L**andis

They introduced this tree in 1962 to improve searching time.

In normal BST:

- If we keep inserting numbers in sorted order, the tree becomes skewed (like a straight line).
- Searching becomes slow (like searching in a long list).

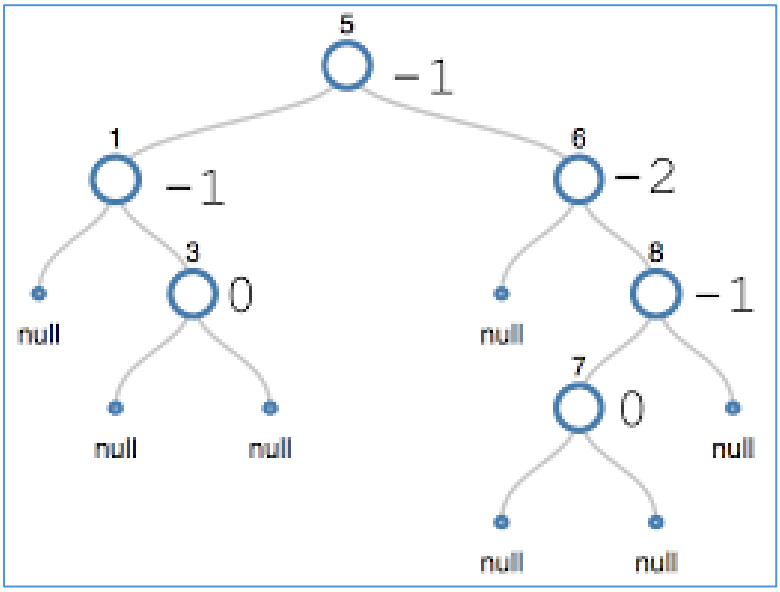
To fix this problem, AVL Trees add a rule:

Rule: The height difference between the left and right subtrees of ANY node must not be more than 1.

This difference is called the **Balance Factor**.

Balance Factor = height(left subtree) – height(right subtree)

- If BF = 0 → Perfectly balanced
- If BF = 1 or -1 → Still okay
- If BF > 1 or BF < -1 → Not balanced → FIX using rotations



Four Types of Rotations

When the tree gets unbalanced, we fix it using these rotations:

Case	Condition	Fix
Left-Left (LL)	Inserted into left subtree of left child	Right Rotation
Right-Right (RR)	Inserted into right subtree of right child	Left Rotation
Left-Right (LR)	Inserted into right subtree of left child	Left + Right Rotation
Right-Left (RL)	Inserted into left subtree of right child	Right + Left Rotation

Real-Life Applications / Uses

AVL Trees are used where **fast searching and frequent updates** are needed.

Real-Life Use Cases

Databases Indexing	AVL trees keep data sorted and balanced Searching becomes very fast
File Systems	(Linux uses similar concepts) To maintain directory structures
Networking	To store routing tables Fast lookup for IP addresses
Games	Storing leaderboard rankings Always sorted, fast insert/search
Memory Management in Operating Systems	Tracking available memory blocks

C++ Code (Simple, Class-Based Implementation)

Below is an extremely simple and clean version of an **AVL Tree in C++** using classes.

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

// Node class
class Node {
public:
    int value;
```

```
    Node* left;
    Node* right;
    int height;

    Node(int v) {
        value = v;
        left = right = NULL;
        height = 1;
    }
};

// AVL Tree class
class AVL {
public:

    // Get height of node
    int getHeight(Node* n) {
        if (n == NULL) return 0;
        return n->height;
    }

    // Get balance factor
    int getBalance(Node* n) {
        if (n == NULL) return 0;
        return getHeight(n->left) - getHeight(n->right);
    }

    // Right Rotation
    Node* rightRotate(Node* y) {
        Node* x = y->left;
        Node* T2 = x->right;
```

```
    x->right = y;
    y->left = T2;

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

    return x;
}

// Left Rotation
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    y->left = x;
    x->right = T2;

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

    return y;
}

// Insert a value
Node* insert(Node* node, int key) {
    // Normal BST insert
    if (node == NULL)
        return new Node(key);

    if (key < node->value)
```

```
    node->left = insert(node->left, key);
else if (key > node->value)
    node->right = insert(node->right, key);
else
    return node; // no duplicates

// Update height
node->height = 1 + max(getHeight(node->left), getHeight(node->right));

// Check balance
int balance = getBalance(node);

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

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

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

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

```
        return node; // balanced
    }

    // Inorder Traversal
    void inorder(Node* root) {
        if (root == NULL) return;
        inorder(root->left);
        cout << root->value << " ";
        inorder(root->right);
    }
};

// Driver Code
int main() {
    AVL tree;
    Node* root = NULL;

    root = tree.insert(root, 30);
    root = tree.insert(root, 20);
    root = tree.insert(root, 40);
    root = tree.insert(root, 10);

    cout << "Inorder Traversal: ";
    tree.inorder(root);
}
```

Output

```
Inorder Traversal: 10 20 30 40

=== Code Execution Successful ===
```

Practice Questions

Q1 Write a C++ program (or complete the given AVL class) to insert the following values into an AVL Tree:

50, 20, 60, 10, 30, 25

After each insertion:

1. Print the **inorder traversal**.
2. Print the **balance factor** of the root.
3. Print which rotation (LL, RR, LR, RL) was applied at that step, if any.

"LL Rotation", "**RR Rotation**", **"LR Rotation"** , **"RL Rotation"**, "**No Rotation**"

Expected Output Example (Format Only):

Insert 50: Inorder = 50 Root BF = 0 Rotation = None **Insert 20: Inorder = 20 50** Root BF = +1 Rotation = None

Q2 Consider an AVL insert operation being performed on the following starting tree:

40

/

20

\

30

This situation requires an **LR (Left–Right) Rotation**.

1. Write the C++ code snippet (ONLY the affected part) showing:
    - First: left rotation on the child
    - Second: right rotation on the parent
2. Draw the tree **before rotation**, **after left rotation**, and **after right rotation**.

   Expected Output Format

Before Rotation:

```
  40
 /
20
 \
 30
```

After Left Rotation (on 20):

```
  40
 /
30
 /
20
```

After Right Rotation (on 40):

```
  30
 / \
20 40
```

Q3 You are given this part of an AVL Tree class:

int getHeight(Node* n);

int getBalance(Node* n);

Now consider the tree:

```
      30
     /  \
   20    40
   /
  10
```

1. Compute the height of every node manually (like the C++ function would).
2. Compute the balance factor of every node using the formula:
   BF = height(left) – height(right)
3. Show how the C++ functions above would return those values.

   Expected Output Format:

```
Height(10) = 1
Height(20) = 2
Height(40) = 1
Height(30) = 3

Balance(10) = 0
Balance(20) = 1
Balance(40) = 0
Balance(30) = 1
```

Q4 Write a simple C++ function inside a class **without using recursion**, which calculates the height of a given node.

Hints:

- You can use max()
- You will access node->left->height and node->right->height safely
- Make sure to check if left or right pointer is NULL

Your output should include:

- Final C++ function
- A small dry run on the following mini tree:

```
  5
 /
3
```

Show how the height of node 5 is calculated step-by-step.

Expected Output:

```
Height(3) = 1
Height(5) = 2
```