SCHOOL OF COMPUTING

M M VASANTH

CH.SC.U4CSE24227

Week – 6

Date - 09/01/2026

Design and Analysis of Algorithm(23CSE211)

1. ADELSON – VELSKY AND LANDIS TREE (AVL)

Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
} Node;

int getHeight(Node *n) {
    return (n == NULL) ? 0 : n->height;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}
Node* newNode(int key) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = key;
    node->left = node->right = NULL;
    node->height = 1; // New node is initially added at leaf
    return node;
}

Node* rightRotate(Node *y) {
    Node *x = y->left;
    Node *T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
```

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

    return x;
}
Node* leftRotate(Node *x) {
    Node *y = x->right;
    Node *T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    return y;
}
int getBalance(Node *N) {



    return (N == NULL) ? 0 : getHeight(N->left) - getHeight(N->right);
}
Node* insert(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(getHeight(node->left), getHeight(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;
}
void inOrder(Node *root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->key);
        inOrder(root->right);
    }
}

int main() {
    Node *root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
```

```
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("In-order traversal of the constructed AVL tree is: \n");
    inOrder(root);
    printf("\n");



    return 0;
}
```

Output:

```
vasanth@VasanthMM:/mnt/c/Users/vasan$ nano avl.c
vasanth@VasanthMM:/mnt/c/Users/vasan$ gcc avl.c -o avl
vasanth@VasanthMM:/mnt/c/Users/vasan$ ./avl
In-order traversal of the constructed AVL tree is:
10 20 25 30 40 50
vasanth@VasanthMM:/mnt/c/Users/vasan$
```

Time Complexity: O(log N)

Because the tree is strictly balanced, the height is guaranteed to be O(log N). Operations like insertion, deletion, and search take time proportional to the height. Unlike a standard BST, which can degrade to O(N) in the worst case (skewed tree), AVL trees ensure O(log N) even in the worst case.

Space Complexity: O(N)

I used malloc to allocate memory for each of the N nodes. Additionally, each node stores an integer height variable to track balance, which adds a small constant overhead per node compared to a standard BST.