Khushi panjwani_D10B42 **lab-10**

Aim- To understand the concept and implementation of avl tree

Theory-
AVL Tree Theory

An AVL Tree is a self-balancing binary search tree (BST) that ensures the heights of the left and right subtrees of any node differ by at most one. It was invented by Georgy Adelson-Velsky and Evgenii Landis in 1962 and is named after their initials.

Key Characteristics:

1. Binary Search Tree Property:

Each node follows the standard binary search tree rules, where the left child contains values less than the parent node, and the right child contains values greater than the parent node.

2. Balance Factor:

The balance factor of a node is defined as the difference between the heights of its left and right subtrees:

\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}

- **-1** ≤ Balance Factor ≤ **1**

This property ensures that the tree remains balanced, minimizing the height and thus optimizing search times.

3. Height:

The height of an AVL tree with nodes is guaranteed to be . This logarithmic height results in efficient operations.

Operations:

Khushi panjwani_D10B42 **lab-10**

1. Insertion:

Insertion in an AVL tree is similar to insertion in a standard binary search tree.

After inserting a new node, the tree checks the balance factor of the affected nodes and performs rotations as needed to restore balance.

2. Deletion:

Deletion also follows the same approach as in a standard BST, but after removing a node, the tree must check balance factors and perform rotations to maintain the AVL property.

3. Rotations:

To restore balance, the AVL tree uses rotations:

Right Rotation: Used when a left-heavy tree becomes unbalanced (left-left case).

Left Rotation: Used when a right-heavy tree becomes unbalanced (right-right case).

Left-Right Rotation: A combination of a left rotation followed by a right rotation, used for a left-heavy tree where a right child is inserted (left-right case).

Right-Left Rotation: A combination of a right rotation followed by a left rotation, used for a right-heavy tree where a left child is inserted (right-left case).

Advantages:

Self-Balancing: The AVL tree maintains a balanced state through rotations, ensuring efficient search, insertion, and deletion operations.

Fast Lookups: Because of its balanced nature, AVL trees provide faster lookups compared to unbalanced binary search trees, especially in scenarios where frequent insertions and deletions occur.

Disadvantages:

Complexity: AVL trees require more complex algorithms for balancing compared to simpler tree structures, such as standard binary search trees or red-black trees.

Overhead: The need to maintain balance factors and perform rotations introduces overhead during insertions and deletions, which can be slightly slower compared to other balanced trees in specific scenarios.

Applications:

Database Indexing: AVL trees are used in databases for indexing, allowing for quick data retrieval while handling frequent updates.

Memory Management: They can manage memory allocations and deallocations efficiently, maintaining balance as memory is allocated and freed.

Dynamic Sets and Maps: AVL trees are useful in implementing dynamic sets and maps where frequent insertions, deletions, and lookups are required.

Input-

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

// Definition of the node structure
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
    int height;
};

// Function to get the height of a node
int height(struct Node *node) {
    return node ? node->height : 0;
}

// Function to get the balance factor of a node
int getBalance(struct Node *node) {
    return node ? height(node->left) - height(node->right) : 0;
}

// Function to create a new node
```

```
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    newNode->height = 1; // New node is initially added at leaf
    return newNode;
}

// Right rotation
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 = 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 the new root
    return x;
}

// Left rotation
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 = 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 the new root
    return y;
}

// Function to insert a value into the AVL tree
struct Node* insert(struct Node* node, int value) {
```

```
    // Perform the normal BST insertion
    if (node == NULL) return createNode(value);

    if (value < node->data)
        node->left = insert(node->left, value);
    else if (value > node->data)
        node->right = insert(node->right, value);
    else // Duplicate values are not allowed
        return node;

    // Update the height of this ancestor node
    node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) :
height(node->right));

    // Get the balance factor of this ancestor node
    int balance = getBalance(node);

    // If the node becomes unbalanced, then there are 4 cases

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

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

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

    // Right Left Case
    if (balance < -1 && value < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // Return the (unchanged) node pointer
    return node;
}

// Function to perform inorder traversal
```

```
void inorder(struct Node *root) {
    if (root) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = NULL;

    // Inserting values into the AVL tree
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 10); // Causes Right Rotation

    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 60); // Causes Left Rotation

    printf("Inorder traversal of the AVL tree: ");
    inorder(root); // Output: 10 20 30 40 50 60

    return 0;
}
```

Output-

```
/tmp/IrE0iLpvfb.o
Inorder traversal of the AVL tree: 10 20 30
    40 50 60
```

Conclusion:

An AVL Tree is a robust data structure that combines the principles of binary search trees with self-balancing techniques to ensure efficient performance for a variety of operations. Its properties make it particularly suitable for applications requiring dynamic data management while maintaining sorted order, allowing for quick search operations even in the presence of frequent updates. Understanding AVL trees is essential for computer science students and professionals working with data structures and algorithms.