Lab Experiment No. 7

Objective

Implementation and analysis of Splay Trees and comparison with basic data structures.

Theory

Splay Trees are a type of self-adjusting binary search tree where recently accessed elements are moved to the root through a process called **splaying**. This ensures that frequently accessed elements are quick to access again, improving performance in real-world scenarios with non-uniform access patterns.

Each operation (search, insertion, deletion) involves a **splay** step to bring the accessed node to the root. This adaptive nature can lead to amortized time complexities of **O(log n)**.

Splay Trees are often compared to:

- Binary Search Trees (BST)
- AVL Trees
- Red-Black Trees because of their dynamic balancing characteristics.

Algorithm:

1. Splay Operation (Core Mechanism)

Input: Node x to be accessed
Output: Node x becomes the new root

Steps:

1. While x is not the root:
   - Let p be the parent of x and g the grandparent of x (if it exists).
   - Depending on the relationship between x, p, and g, perform one of the following:
     - **Zig** (single rotation): x is child of root (p is root).
     - **Zig-Zig** (double rotation): x and p are both left or both right children.
     - **Zig-Zag** (double rotation): x is a left child and p is a right child, or vice versa.
2. Continue until x is the root.

2. Search Operation

Input: Key k
Output: Node with key k, or NULL if not found.
Steps:

1. Traverse the tree as in a standard BST search.
2. If node with key k is found or a leaf is reached:
   - **Splay** the last accessed node to the root.
3. Return the result.

3. Insert Operation

Input: Key k, Value v
Output: Updated tree with new node inserted.
Steps:

1. If tree is empty, insert the node as root.
2. Else:
   - Traverse as in a BST to find the insertion point.
   - Insert the new node as a leaf.
   - **Splay** the newly inserted node to the root.

4. Delete Operation

Input: Key k
Output: Updated tree with node removed.
Steps:

1. Search for the node with key k.
2. If found:
   - **Splay** the node to the root.
   - If the left and right subtrees exist:
     - Save a reference to the right subtree.
     - Remove the root.
     - Splay the maximum node of the left subtree to the root.
     - Attach the saved right subtree as the new root’s right child.
   - If only one subtree exists, make it the new root.
3. If node not found, exit.

Code

T-5.1. Implementation using Arrays. Depth-First Search (DFS)

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

// Node structure
struct Node {
    int key;
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

// Right rotate
Node* rightRotate(Node* x) {
    Node* y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// Left rotate
Node* leftRotate(Node* x) {
    Node* y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// Splay function
Node* splay(Node* root, int key) {
    if (!root || root->key == key)
        return root;

    // Left subtree
    if (key < root->key) {
        if (!root->left) return root;

        // Zig-Zig (Left Left)
        if (key < root->left->key) {
            root->left->left = splay(root->left->left, key);
            root = rightRotate(root);
        }
        // Zig-Zag (Left Right)
        else if (key > root->left->key) {
            root->left->right = splay(root->left->right, key);
            if (root->left->right)
                root->left = leftRotate(root->left);
        }

        return root->left ? rightRotate(root) : root;
    }
```

```

// Delete function
Node* deleteKey(Node* root, int key) {
    if (!root) return nullptr;

    root = splay(root, key);
    if (root->key != key) return root;  // Key not found

    Node* temp;
    if (!root->left) {
        temp = root->right;
    } else {
        temp = splay(root->left, key);
        temp->right = root->right;
    }
    delete root;
    return temp;
}

// Inorder traversal
void inorder(Node* root) {
    if (!root) return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

// Structured test
int main() {
    Node* root = nullptr;

    cout << "\n--- Splay Tree Operations ---\n";
    cout << "\nInserting elements: 10, 20, 30, 40, 50,
25\n";
    int keys[] = {10, 20, 30, 40, 50, 25};
    for (int key : keys) {
        root = insert(root, key);
        cout << "Inserted: " << key << "\n";
    }

    cout << "\nInorder traversal after insertions:\n";
    inorder(root);
    cout << "\n";

    cout << "\nSearching for key 30:\n";
    root = splay(root, 30);
    cout << "Root after splay: " << root->key << "\n";

    cout << "\nDeleting key 20:\n";
    root = deleteKey(root, 20);
```

```
    // Right subtree
    else {
        if (!root->right) return root;

        // Zag-Zig (Right Left)
        if (key < root->right->key) {
            root->right->left = splay(root->right->left, key);
            if (root->right->left)
                root->right = rightRotate(root->right);
        }
        // Zag-Zag (Right Right)
        else if (key > root->right->key) {
            root->right->right = splay(root->right->right,
key);
            root = leftRotate(root);
        }

        return root->right ? leftRotate(root) : root;
    }
}

// Insert function
Node* insert(Node* root, int key) {
    if (!root) return new Node(key);

    root = splay(root, key);
    if (root->key == key) return root;  // Duplicate not
inserted

    Node* newNode = new Node(key);
    if (key < root->key) {
        newNode->right = root;
        newNode->left = root->left;
        root->left = nullptr;
    } else {
        newNode->left = root;
        newNode->right = root->right;
        root->right = nullptr;
    }
    return newNode;
}
```

```
    inorder(root);
    cout << "\n";

    cout << "\nDeleting key 10:\n";
    root = deleteKey(root, 10);
    inorder(root);
    cout << "\n";

    return 0;
}
```

Sample Output

--- Splay Tree Operations ---

Inserting elements: 10, 20, 30, 40, 50, 25
Inserted: 10
Inserted: 20
Inserted: 30
Inserted: 40
Inserted: 50

Inserted: 25

Inorder traversal after insertions:
10 20 25 30 40 50

Searching for key 30:
Root after splay: 30

Deleting key 20:
10 25 30 40 50

Deleting key 10:
25 30 40 50

Complexity Analysis

Table 7.1 Analysis of time complexity for Splay Tree.

Operation	Average Case	Worst Case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Conclusion

Splay Trees provide an efficient and elegant solution for dynamic set operations where frequent access patterns are skewed. Though the worst-case time for individual operations is linear, their amortized complexity remains logarithmic, making them practical in many scenarios. Compared to strictly balanced trees like AVL or Red-Black Trees, Splay Trees often offer better real-world performance without explicit balancing logic.