Khushi panjwani_D10B42 **lab-09**

Aim- to understand the concept of Implementation of binary search tree
Theory-
Binary Search Tree (BST)

A Binary Search Tree (BST) is a specialized binary tree structure that maintains sorted data, enabling efficient insertion, deletion, and lookup operations. Each node in a BST has a maximum of two children, and it follows these specific properties:

Key Properties:

1. Node Structure: Each node contains a key (value), a pointer to the left child, and a pointer to the right child.

2. Ordering:

The left subtree of a node contains only nodes with values less than the node’s value.

The right subtree of a node contains only nodes with values greater than the node’s value.

3. No Duplicate Nodes: Each value in the BST must be unique.

Basic Operations:

1. Insertion:

Start at the root. If the tree is empty, the new node becomes the root.

If the value to be inserted is less than the current node's value, move to the left child; if it's greater, move to the right child.

Repeat this process until you find a null position where the new node can be inserted.

2. Searching:

Similar to insertion, start at the root and compare the target value with the current node's value.

If they match, the search is successful.

If the target value is less, move to the left child; if greater, move to the right child.

Repeat until the value is found or a null pointer is reached.

3. Traversal:

Inorder Traversal: Visits nodes in non-decreasing order (left, root, right).

Preorder Traversal: Visits nodes in the order (root, left, right) which is useful for copying the tree.

Postorder Traversal: Visits nodes in the order (left, right, root) which is useful for deleting the tree.

4. Deletion:

To delete a node, first search for the node in the same way as you would for insertion.

Depending on the number of children:

No Children: Simply remove the node.

One Child: Remove the node and link its parent directly to its child.

Two Children: Find the node's inorder successor (the smallest node in the right subtree), copy its value to the node to be deleted, and recursively delete the inorder successor.

Advantages of BST:

Efficiency: Average-case time complexity for insertion, deletion, and search operations is O(log n) when the tree is balanced.

Sorted Data: Inorder traversal gives a sorted list of values, which is useful for many applications.

Disadvantages of BST:

Unbalanced Trees: If not balanced (due to random insertions and deletions), the tree can degenerate into a linked list, resulting in O(n) time complexity for operations.

Complexity of Balancing: Maintaining balance in a BST can be complex, which is why self-balancing trees like AVL Trees or Red-Black Trees are often used.

Applications:

Databases: Storing sorted data for efficient retrieval.

Memory Management: Allocating and deallocating memory dynamically.

Sets and Maps: Implementing abstract data types like sets and maps.

Input-

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

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

// 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;
    return newNode;
}

// Function to insert a new node into the BST
struct Node* insert(struct Node* root, int value) {
    // If the tree is empty, create a new node and return it
    if (root == NULL) return createNode(value);

    // Otherwise, recur down the tree
    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);
```

```
    return root;
}

// Function to search for a value in the BST
struct Node* search(struct Node* root, int value) {
    // Base case: root is null or the value is found
    if (root == NULL || root->data == value)
        return root;

    // Value is smaller than the root's value
    if (value < root->data)
        return search(root->left, value);

    // Value is greater than the root's value
    return search(root->right, value);
}

// Function to perform in-order traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);  // Visit left subtree
        printf("%d ", root->data);  // Visit root
        inorder(root->right);  // Visit right subtree
    }
}

// Function to find the minimum value node in the tree (used for deletion)
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;

    // Loop down to find the leftmost leaf
    while (current && current->left != NULL)
        current = current->left;

    return current;
}

// Function to delete a node from the BST
struct Node* deleteNode(struct Node* root, int value) {
    // Base case: the tree is empty
    if (root == NULL)
        return root;
```

```
    // Recur down the tree to find the node to delete
    if (value < root->data)
        root->left = deleteNode(root->left, value);
    else if (value > root->data)
        root->right = deleteNode(root->right, value);
    else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children: Get the inorder successor
        struct Node* temp = minValueNode(root->right);

        // Copy the inorder successor's data to this node
        root->data = temp->data;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("In-order traversal of the BST: ");
    inorder(root); // Expected output: 20 30 40 50 60 70 80

    printf("\nAfter deleting 20\n");
```

```
    root = deleteNode(root, 20);
    inorder(root); // Expected output: 30 40 50 60 70 80

    printf("\nAfter deleting 30\n");
    root = deleteNode(root, 30);
    inorder(root); // Expected output: 40 50 60 70 80

    printf("\nAfter deleting 50\n");
    root = deleteNode(root, 50);
    inorder(root); // Expected output: 40 60 70 80

    return 0;
}
```

Output-

```
/tmp/qAKhjQ6lpi.o
In-order traversal of the BST: 20 30 40 50
    60 70 80
After deleting 20
30 40 50 60 70 80
After deleting 30
40 50 60 70 80
After deleting 50
40 60 70 80
```

Conclusion-

A Binary Search Tree is a fundamental data structure in computer science that facilitates efficient data management and retrieval. Understanding its operations and properties is essential for optimizing search algorithms and implementing various applications in programming and software development.