20. AVL Tree Operations (Insert, Delete, Search)

Aim:
To implement insertion, deletion, and searching in an AVL Tree.

Algorithm (High-level):

1. **Insert:** Insert node like in BST, then check balance factor and perform rotations if needed.

2. **Delete:** Delete node like in BST, then rebalance the tree if needed.

3. **Search:** Traverse the tree like in BST until the key is found or NULL is reached.

CODE:

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

struct Node {
   int key, height;
   struct Node* left;
   struct Node* right;
};

int height(struct Node* N) {
   return N ? N->height : 0;
}

int max(int a, int b) { return (a > b) ? a : b; }

struct Node* newNode(int key) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->key = key;
   node->left = node->right = NULL;
   node->height = 1;
   return node;
}

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

```
    x->right = y;
    y->left = T2;
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

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

int getBalance(struct Node* N) {
    return N ? height(N->left) - height(N->right) : 0;
}

struct Node* insert(struct Node* node, int key) {
    if (!node) 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(height(node->left), height(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;
}

struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current->left) current = current->left;
    return current;
}
```

```
struct Node* deleteNode(struct Node* root, int key) {
    if (!root) return root;
    if (key < root->key) root->left = deleteNode(root->left, key);
    else if (key > root->key) root->right = deleteNode(root->right, key);
    else {
        if (!root->left || !root->right) {
            struct Node* temp = root->left ? root->left : root->right;
            if (!temp) { temp = root; root = NULL; }
            else *root = *temp;
            free(temp);
        } else {
            struct Node* temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }
    if (!root) return root;
    root->height = 1 + max(height(root->left), height(root->right));
    int balance = getBalance(root);

    if (balance > 1 && getBalance(root->left) >= 0) return rightRotate(root);
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }
    if (balance < -1 && getBalance(root->right) <= 0) return leftRotate(root);
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}

struct Node* search(struct Node* root, int key) {
    if (!root || root->key == key) return root;
    if (key < root->key) return search(root->left, key);
    return search(root->right, key);
}

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

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

```
    int choice, key;

    while (1) {
        printf("\n1.Insert 2.Delete 3.Search 4.Display 5.Exit\nEnter choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                root = insert(root, key);
                break;
            case 2:
                printf("Enter key to delete: ");
                scanf("%d", &key);
                root = deleteNode(root, key);
                break;
            case 3:
                printf("Enter key to search: ");
                scanf("%d", &key);
                if (search(root, key)) printf("Key Found\n");
                else printf("Key Not Found\n");
                break;
            case 4:
                printf("PreOrder Traversal: ");
                preOrder(root);
                printf("\n");
                break;
            case 5:
                exit(0);
        }
    }
    return 0;
}
```

Output

```
1.Insert 2.Delete 3.Search 4.Display 5.Exi
Enter choice: 1
Enter key to insert: 20

1.Insert 2.Delete 3.Search 4.Display 5.Exi
Enter choice: 1
Enter key to insert: 15

1.Insert 2.Delete 3.Search 4.Display 5.Exi
Enter choice: 2
Enter key to delete: 15


1.Insert 2.Delete 3.Search 4.Display 5.Exi
Enter choice: 3
Enter key to search: 20
Key Found

1.Insert 2.Delete 3.Search 4.Display 5.Exi
Enter choice: 4
PreOrder Traversal: 20
```

RESULT:

The program successfully executed and displayed the avl tree operations.