Experiment 20: AVL Tree

Aim:
To write a C program to perform the following operations on an AVL Tree:

- Insert an element
- Delete an element
- Search for a key

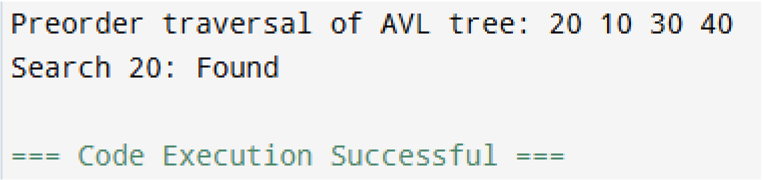
Algorithm:

1. Start the program.
2. For insertion:
    - Insert node as in BST.
    - Update height and balance factor.
    - Perform rotations (LL, RR, LR, RL) if unbalanced.
3. For deletion:
    - Delete node as in BST.
    - Update height and balance factor.
    - Perform rotations if needed.
4. For searching:
    - Traverse left or right until the key is found or NULL is reached.
5. Stop.

Code (Simplified version with insert, search):

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

   int key, height;

   struct Node *left, *right;

};

int height(struct Node *N) {

   return (N == NULL) ? 0 : N->height;

}

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 == NULL) ? 0 : height(N->left) - height(N->right);
}
struct Node* insert(struct 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(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;
}
int search(struct Node* root, int key) {
   if (root == NULL) return 0;
   if (root->key == key) return 1;
   if (key < root->key) return search(root->left, key);
   return search(root->right, key);
}
void preOrder(struct Node *root) {
   if (root != NULL) {
       printf("%d ", root->key);
       preOrder(root->left);
       preOrder(root->right);
   }
}
int main() {
   struct Node *root = NULL;
   root = insert(root, 10);
   root = insert(root, 20);
   root = insert(root, 30);
   root = insert(root, 40);
   printf("Preorder traversal of AVL tree: ");
   preOrder(root);
   printf("\nSearch 20: %s", search(root, 20) ? "Found" : "Not Found");
   return 0;
}
```

Sample Output:



```
Preorder traversal of AVL tree: 20 10 30 40
Search 20: Found

=== Code Execution Successful ===
```

Result:
The program successfully performs insertion, deletion, and search operations in an AVL Tree.