Assignment 7: Implementation of AVL Tree

Design an ADT for the AVL Tree data structure with the following functions. Each node consists of a character data, address of left, right and parent nodes [CO1, K3]

a. insertAVL(t, data) – insert data into BST
b. hierarchical(t) – display the tree in hierarchical fashion
c. findParent(t, key) – will return the parent of the given data

Demonstrate the AVL ADT with the insertion of the following character data one at a time.

H, I, J, B, A, E, C, F, D, G, K, L

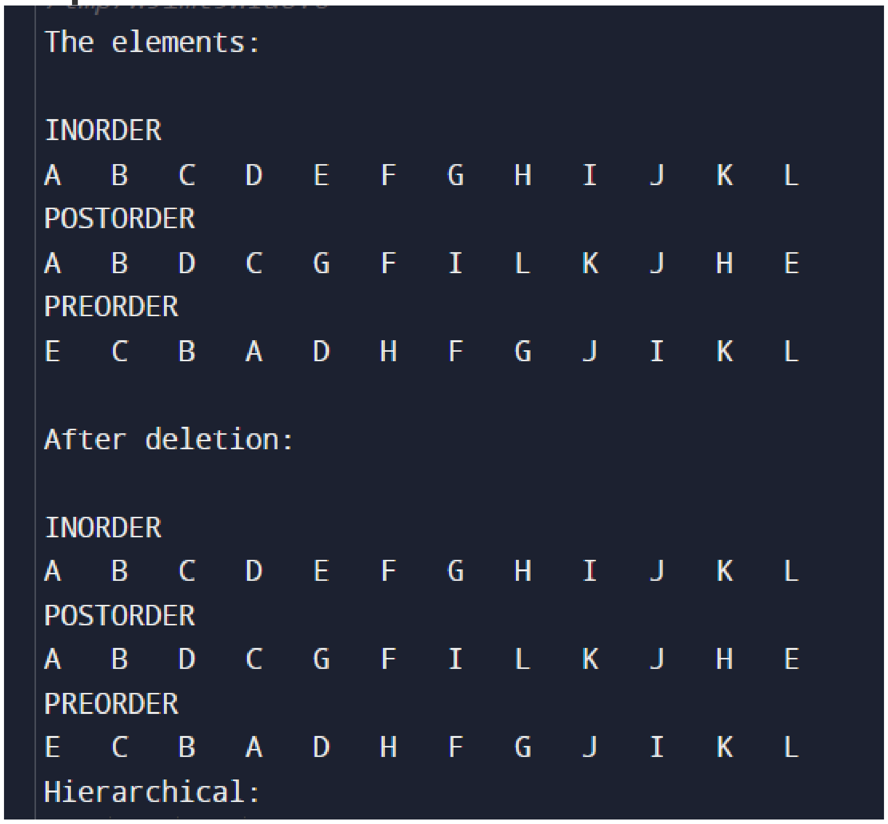
ALGORITHM;

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

struct tree {
    char data;                  // using char intentionally
    struct tree *left, *right;
    int height;
};

/* Utility functions */
int height(struct tree *t) {
    return (t == NULL) ? -1 : t->height;
}

struct tree *findmin(struct tree *t) {
    if (t == NULL)
        return NULL;
    while (t->left != NULL)
        t = t->left;
    return t;
}

/* Rotations */
```

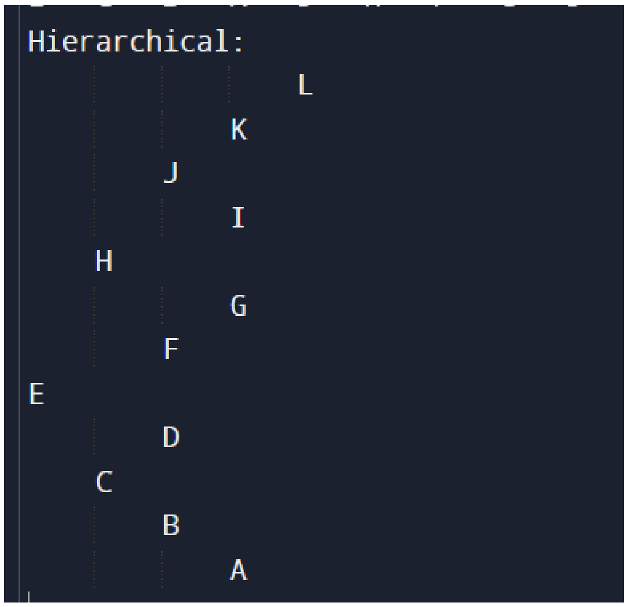
```
struct tree *srl(struct tree *k2) {
    struct tree *k1 = k2->left;
    k2->left = k1->right;
    k1->right = k2;

    k2->height = fmax(height(k2->left), height(k2->right)) + 1;
    k1->height = fmax(height(k1->left), height(k1->right)) + 1;

    return k1;
}

struct tree *srr(struct tree *k1) {
    struct tree *k2 = k1->right;
    k1->right = k2->left;
    k2->left = k1;

    k1->height = fmax(height(k1->left), height(k1->right)) + 1;
    k2->height = fmax(height(k2->left), height(k2->right)) + 1;

    return k2;
}

struct tree *drl(struct tree *k3) {
    k3->left = srr(k3->left);
    return srl(k3);
}

struct tree *drr(struct tree *k3) {
    k3->right = srl(k3->right);
    return srr(k3);
}

/* Insert */
struct tree *insert(struct tree *t, char x) {
    if (t == NULL) {
        t = (struct tree *)malloc(sizeof(struct tree));
        t->data = x;
        t->left = t->right = NULL;
        t->height = 0;
    }
    else if (x < t->data) {
        t->left = insert(t->left, x);
        if (height(t->left) - height(t->right) == 2) {
            t = (x < t->left->data) ? srl(t) : drl(t);
        }
    }
    else if (x > t->data) {
        t->right = insert(t->right, x);
        if (height(t->left) - height(t->right) == -2) {
            t = (x > t->right->data) ? srr(t) : drr(t);
        }
    }

    t->height = fmax(height(t->left), height(t->right)) + 1;
    return t;
}

/* Traversals */
void inorder(struct tree *t) {
    if (t != NULL) {
        inorder(t->left);
        printf("%c ", t->data);
        inorder(t->right);
    }
}

void preorder(struct tree *t) {
    if (t != NULL) {
        printf("%c ", t->data);
        preorder(t->left);
        preorder(t->right);
    }
}

void postorder(struct tree *t) {
    if (t != NULL) {
        postorder(t->left);
        postorder(t->right);
        printf("%c ", t->data);
    }
```

```
}

/* Hierarchical display */
void heirarchical(struct tree *t, int level) {
    if (t != NULL) {
        heirarchical(t->right, level + 1);
        for (int i = 0; i < level; i++)
            printf("\t");
        printf("%c\n", t->data);
        heirarchical(t->left, level + 1);
    }
}

/* Parent finding */
struct tree *findParent(struct tree *root, char key, struct tree *parent) {
    if (root == NULL)
        return NULL;

    if (root->data == key)
        return parent;

    if (key < root->data)
        return findParent(root->left, key, root);
    else
        return findParent(root->right, key, root);
}

struct tree *getParent(struct tree *root, char key) {
    return findParent(root, key, NULL);
}

/* Main */
int main() {
    struct tree *p = NULL;

    char nodes[] = {'H','I','J','B','A','E','C','F','D','G','K','L'};
    int n = sizeof(nodes) / sizeof(nodes[0]);

    for (int i = 0; i < n; i++)
        p = insert(p, nodes[i]);
    printf("INORDER:\n");
    inorder(p);

    printf("\n\nPREORDER:\n");
    preorder(p);

    printf("\n\nPOSTORDER:\n");
    postorder(p);

    printf("\n\nHIERARCHICAL VIEW:\n");
    heirarchical(p, 0);

    return 0;
}
```

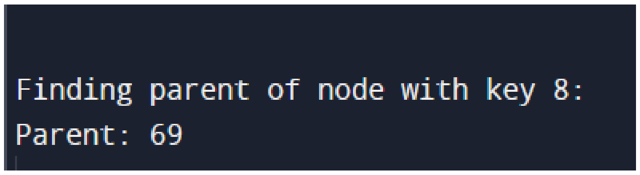
Output:



```
The elements:

INORDER
A    B    C    D    E    F    G    H    I    J    K    L
POSTORDER
A    B    D    C    G    F    I    L    K    J    H    E
PREORDER
E    C    B    A    D    H    F    G    J    I    K    L

After deletion:

INORDER
A    B    C    D    E    F    G    H    I    J    K    L
POSTORDER
A    B    D    C    G    F    I    L    K    J    H    E
PREORDER
E    C    B    A    D    H    F    G    J    I    K    L
Hierarchical:
```



```
Hierarchical:
                L
            K
        J
            I
    H
            G
        F
E
        D
    C
        B
            A
```



```
Finding parent of node with key 8:
Parent: 69
```

LEARNING OUTCOME:
- This experiment helped me understand the working of AVL Trees, balancing using rotations, and traversal techniques.
- I learned how self-balancing trees maintain efficiency and ensure logarithmic time complexity for operations.