Tutorial 8: AVL Trees

ELEC 278: Fundamentals of Information Structures

The learning goals for Tutorial 8 are:

- Understand rotations in AVL Trees.
- Implement node insertion and deletion in AVL Trees.

Problem 1. Complete the `insert` and `deleteNode` functions. You can use the utility functions provided.

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

// AVL Tree Node
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int height;
};

// Utility function to get the height of the tree
int height(struct Node* N) {
    if (N == NULL)
        return 0;
    return N->height;
}

// Utility function to get maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Utility function to create a new node
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;  // new node is initially added at leaf
    return node;
}

// Utility function to right rotate subtree rooted with y
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;
}

// Utility function to left rotate subtree rooted with 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;
}

// Get Balance factor of node N
int getBalance(struct Node* N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Utility function to get the minimum value node in the tree
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

// Utility function to print preorder traversal of the tree
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key) {
    // to do: complete
}

// Recursive function to delete a node
struct Node* deleteNode(struct Node* root, int key) {
    // to do: complete
}

// Main function to test above functions
int main() {
    struct Node* root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);
```

```
    printf("Preorder traversal of the constructed AVL tree is:\n");
    preOrder(root);
    printf("\n");

    root = deleteNode(root, 10);
    printf("Preorder traversal after deletion of 10:\n");
    preOrder(root);
    printf("\n");

    return 0;
}
```

Solution.

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

// AVL Tree Node
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int height;
};

struct Node* minValueNode(struct Node*);

// Utility function to get the height of the tree
int height(struct Node* N) {
    if (N == NULL)
        return 0;
    return N->height;
}

// Utility function to get maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Utility function to create a new node
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;  // new node is initially added at leaf
    return node;
}

// Utility function to right rotate subtree rooted with y
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;
}
```

```c
// Utility function to left rotate subtree rooted with 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;
}

// Get Balance factor of node N
int getBalance(struct Node* N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
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  // Equal keys are not allowed in BST
        return node;

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases
    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}
```

```
// Recursive function to delete a node
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL)
        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 == NULL) || (root->right == NULL)) {
            struct Node* temp = root->left ? root->left : root->right;
            if (temp == NULL) {
                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 == NULL)
        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;
}

// Utility function to get the minimum value node in the tree
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}
```

```c
// Utility function to print preorder traversal of the tree
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Main function to test above functions
int main() {
    struct Node* root = NULL;
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("Preorder traversal of the constructed AVL tree is:\n");
    preOrder(root);
    printf("\n");

    root = deleteNode(root, 10);
    printf("Preorder traversal after deletion of 10:\n");
    preOrder(root);
    printf("\n");

    return 0;
}
```