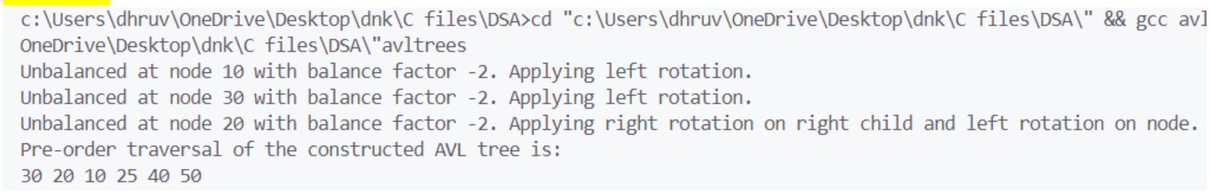
Q1. Implement a function to demonstrate the process of inserting a new node into an AVL tree while maintaining balance factor. Print the balance factor of unbalanced node and rotations applied after every insertion operation.

CODE:

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

struct Node
{
    int data;
    struct Node *left;
    struct Node *right;
    int height;
};

int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

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

struct Node *newNode(int data)
{
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    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)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct Node *insert(struct Node *node, int data)
{
    if (node == NULL)
        return (newNode(data));

    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
        return node;

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

    int balance = getBalance(node);

    if (balance > 1 && data < node->left->data)
    {
        return rightRotate(node);
```

```c
    }

    if (balance < -1 && data > node->right->data)
    {
        return leftRotate(node);
    }

    if (balance > 1 && data > node->left->data)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && data < node->right->data)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

void preOrder(struct Node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->data);
        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);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("Pre-order traversal of the constructed AVL tree is:\n");
    preOrder(root);
    return 0;
}
```
OUTPUT:



```
c:\Users\dhruv\OneDrive\Desktop\dnk\C files\DSA>cd "c:\Users\dhruv\OneDrive\Desktop\dnk\C files\DSA\" && gcc avl
OneDrive\Desktop\dnk\C files\DSA\"avltrees
Unbalanced at node 10 with balance factor -2. Applying left rotation.
Unbalanced at node 30 with balance factor -2. Applying left rotation.
Unbalanced at node 20 with balance factor -2. Applying right rotation on right child and left rotation on node.
Pre-order traversal of the constructed AVL tree is:
30 20 10 25 40 50
```

Q2. Implement the deletion operation for an AVL tree constructed in the above question. Delete a node and show the tree's structure after each deletion and any required rotations.

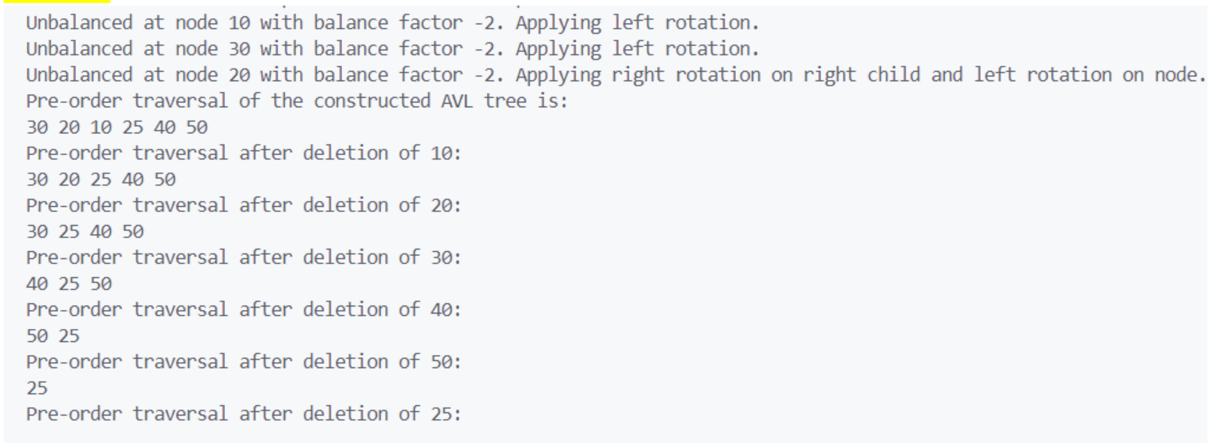
CODE:

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

struct Node {
    int data;
    struct Node *left;
    struct Node *right;
    int height;
};

int height(struct Node *N) {
    if (N == NULL)
        return 0;
    return N->height;
}

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

struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    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) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;

    while (current->left != NULL)
        current = current->left;

    return current;
}

struct Node* insert(struct Node* node, int data) {
    if (node == NULL)
        return(newNode(data));

    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);
    else
```

```c
        return node;

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

    int balance = getBalance(node);

    if (balance > 1 && data < node->left->data) {
        printf("Unbalanced at node %d with balance factor %d. Applying 
right rotation.\n", node->data, balance);
        return rightRotate(node);
    }

    if (balance < -1 && data > node->right->data) {
        printf("Unbalanced at node %d with balance factor %d. Applying 
left rotation.\n", node->data, balance);
        return leftRotate(node);
    }

    if (balance > 1 && data > node->left->data) {
        printf("Unbalanced at node %d with balance factor %d. Applying 
left rotation on left child and right rotation on node.\n", node->data, 
balance);
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && data < node->right->data) {
        printf("Unbalanced at node %d with balance factor %d. Applying 
right rotation on right child and left rotation on node.\n", node-
>data, balance);
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

struct Node* deleteNode(struct Node* root, int data) {
    if (root == NULL)
        return root;

    if (data < root->data)
        root->left = deleteNode(root->left, data);
    else if (data > root->data)
        root->right = deleteNode(root->right, data);
    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->data = temp->data;

            root->right = deleteNode(root->right, temp->data);
        }
    }

    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) {
        printf("Unbalanced at node %d with balance factor %d. Applying
right rotation.\n", root->data, balance);
        return rightRotate(root);
    }

    if (balance > 1 && getBalance(root->left) < 0) {
        printf("Unbalanced at node %d with balance factor %d. Applying
left rotation on left child and right rotation on node.\n", root->data,
balance);
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if (balance < -1 && getBalance(root->right) <= 0) {
        printf("Unbalanced at node %d with balance factor %d. Applying
left rotation.\n", root->data, balance);
        return leftRotate(root);
    }
```

```c
    if (balance < -1 && getBalance(root->right) > 0) {
        printf("Unbalanced at node %d with balance factor %d. Applying
right rotation on right child and left rotation on node.\n", root-
>data, balance);
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

void preOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->data);
        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);
    root = insert(root, 50);
    root = insert(root, 25);

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

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

    root = deleteNode(root, 20);
    printf("Pre-order traversal after deletion of 20:\n");
    preOrder(root);
    printf("\n");

    root = deleteNode(root, 30);
    printf("Pre-order traversal after deletion of 30:\n");
    preOrder(root);
    printf("\n");

    root = deleteNode(root, 40);
    printf("Pre-order traversal after deletion of 40:\n");
    preOrder(root);
    printf("\n");

    root = deleteNode(root, 50);
    printf("Pre-order traversal after deletion of 50:\n");
    preOrder(root);
    printf("\n");

    root = deleteNode(root, 25);
    printf("Pre-order traversal after deletion of 25:\n");
    preOrder(root);
    printf("\n");

    return 0;
}
```

OUTPUT:



```
Unbalanced at node 10 with balance factor -2. Applying left rotation.
Unbalanced at node 30 with balance factor -2. Applying left rotation.
Unbalanced at node 20 with balance factor -2. Applying right rotation on right child and left rotation on node.
Pre-order traversal of the constructed AVL tree is:
30 20 10 25 40 50
Pre-order traversal after deletion of 10:
30 20 25 40 50
Pre-order traversal after deletion of 20:
30 25 40 50
Pre-order traversal after deletion of 30:
40 25 50
Pre-order traversal after deletion of 40:
50 25
Pre-order traversal after deletion of 50:
25
Pre-order traversal after deletion of 25:
```

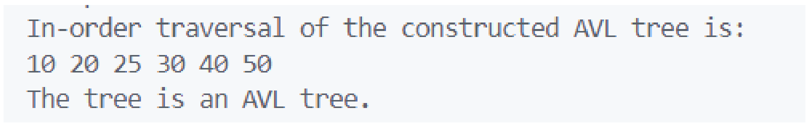
Q3. Implement a function to validate whether a given binary search tree is a valid AVL tree.

CODE:

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

struct Node
{
    int key;
    struct Node *left;
```

```c
    struct Node *right;
    int height;
};

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

int height(struct Node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

struct Node *newNode(int key)
{
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    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 = maximum(height(y->left), height(y->right)) + 1;
    x->height = maximum(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 = maximum(height(x->left), height(x->right)) + 1;
    y->height = maximum(height(y->left), height(y->right)) + 1;

    return y;
}

int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return 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 + maximum(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;
}

bool isBSTUtil(struct Node *node, int min, int max)
{
    if (node == NULL)
        return true;

    if (node->key < min || node->key > max)
        return false;

    return isBSTUtil(node->left, min, node->key - 1) && isBSTUtil(node-
>right, node->key + 1, max);
}

bool isBST(struct Node *node)
{
    return isBSTUtil(node, INT_MIN, INT_MAX);
}

bool isBalanced(struct Node *node)
{
    if (node == NULL)
        return true;

    int leftHeight = height(node->left);
    int rightHeight = height(node->right);

    if (abs(leftHeight - rightHeight) <= 1 && isBalanced(node->left) &&
isBalanced(node->right))
        return true;

    return false;
}

bool isAVL(struct Node *root)
{
    return isBST(root) && isBalanced(root);
}

void inOrder(struct Node *root)
{
```

```c
    if (root != NULL)
    {
        inOrder(root->left);
        printf("%d ", root->key);
        inOrder(root->right);
    }
}

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("In-order traversal of the constructed AVL tree is:\n");
    inOrder(root);
    printf("\n");

    if (isAVL(root))
        printf("The tree is an AVL tree.\n");
    else
        printf("The tree is not an AVL tree.\n");

    return 0;
}
```
OUTPUT:



```
In-order traversal of the constructed AVL tree is:
10 20 25 30 40 50
The tree is an AVL tree.
```