```c
//binary tree creation and tree traversals
#include <stdio.h>
#include <stdlib.h>

// Structure for a node in the binary tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Inorder traversal (Left, Root, Right)
void inorder(struct Node* root) {
    if (root == NULL)
        return;

    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

// Preorder traversal (Root, Left, Right)
void preorder(struct Node* root) {
    if (root == NULL)
        return;

    printf("%d ", root->data);
    preorder(root->left);
    preorder(root->right);
}

// Postorder traversal (Left, Right, Root)
void postorder(struct Node* root) {
    if (root == NULL)
        return;

    postorder(root->left);
    postorder(root->right);
    printf("%d ", root->data);
}
```

```
int main() {
    // Creating a binary tree
    struct Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");

    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}

// binary search tree
#include <stdio.h>
#include <stdlib.h>

// Structure for a node in the binary search tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Function to insert a node into the BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }

    if (data < root->data) {
```

```
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }

    return root;
}

// Inorder traversal (Left, Root, Right)
void inorder(struct Node* root) {
    if (root == NULL)
        return;

    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder traversal of the BST: ");
    inorder(root);
    printf("\n");

    return 0;
}
// expression tree
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

// Structure for a tree node
struct Node {
    char data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(char data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
```

```
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Stack structure for storing nodes
struct Stack {
    int top;
    int capacity;
    struct Node** array;
};

// Function to create a stack
struct Stack* createStack(int capacity) {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (struct Node**)malloc(stack->capacity * sizeof(struct Node*));
    return stack;
}

// Stack utility functions
int isFull(struct Stack* stack) { return stack->top == stack->capacity - 1; }
int isEmpty(struct Stack* stack) { return stack->top == -1; }
void push(struct Stack* stack, struct Node* node) { stack->array[++stack->top] =
node; }
struct Node* pop(struct Stack* stack) { return stack->array[stack->top--]; }
struct Node* peek(struct Stack* stack) { return stack->array[stack->top]; }

// Function to check if a character is an operator
int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

// Function to construct the expression tree from postfix expression
struct Node* constructTree(char postfix[]) {
    struct Stack* stack = createStack(100);
    struct Node *t, *t1, *t2;

    // Traverse each character of the postfix expression
    for (int i = 0; postfix[i] != '\0'; i++) {
        // If operand, create a new node and push to stack
        if (isalnum(postfix[i])) {
            t = createNode(postfix[i]);
            push(stack, t);
        }
        // If operator, pop two nodes, make them children of operator node
        else if (isOperator(postfix[i])) {
            t = createNode(postfix[i]);
```

```
            // Pop top two nodes
            t1 = pop(stack); // Right child
            t2 = pop(stack); // Left child

            // Link children to operator node
            t->right = t1;
            t->left = t2;

            // Push this subtree back to stack
            push(stack, t);
        }
    }

    // The final node in the stack is the root of the expression tree
    t = pop(stack);
    return t;
}

// Inorder traversal (Left, Root, Right)
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%c ", root->data);
        inorder(root->right);
    }
}

// Preorder traversal (Root, Left, Right)
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%c ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Postorder traversal (Left, Right, Root)
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%c ", root->data);
    }
}

// Main function to demonstrate
int main() {
    char postfix[] = "ab+ef*g*-"; // Example postfix expression

    struct Node* root = constructTree(postfix);
```

```
    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");

    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
// AVL tree implementation
#include <stdio.h>
#include <stdlib.h>

// Structure for a node of the AVL tree
struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int height;
};

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

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

// 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 = NULL;
    node->right = NULL;
    node->height = 1;  // New node is initially added at leaf
    return node;
}

// Right rotate subtree rooted with y
```

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

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}

// Left rotate subtree rooted with x
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Return new root
    return y;
}

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

// Recursive function to insert a key in the subtree rooted with node
struct Node* insert(struct Node* node, int key) {
    // Perform normal BST insertion
    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;

    // Update height of this ancestor node
    node->height = 1 + max(height(node->left), height(node->right));

    // Get the balance factor of this ancestor node to check whether this node
became unbalanced
    int balance = getBalance(node);

    // If node becomes unbalanced, 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 the (unchanged) node pointer
    return node;
}

// Function to print the tree in pre-order traversal
void preOrder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Driver program
int main() {
    struct Node* root = NULL;

    // Insert nodes
```

```
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

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

    return 0;
}
```

Explanation:

The insert function inserts nodes like a binary search tree (BST) but then checks the balance factor to maintain the AVL property.

Rotations (left and right) are performed to maintain the balance when the tree becomes unbalanced.

The preOrder function is used to display the tree structure.

This program demonstrates AVL tree operations with insertion and balancing to ensure it remains a self-balancing BST.