**Exp No. 5**

## Title:

A) Implement Binary Search Tree (BST) ADT using Linked List | B) Construct an Expression Tree from Postfix Expression

## Aim:

To implement (A) Binary Search Tree ADT using linked list nodes and (B) construct an Expression Tree from a given postfix expression, with appropriate traversals.

## Objectives:

• Understand dynamic node-based tree structures.  
• Implement BST operations (insert, search, delete, traversals).  
• Construct an expression tree from postfix using a stack of node pointers.  
• Demonstrate inorder, preorder, and postorder traversals on both trees.

## Theory:

A Binary Search Tree (BST) is a binary tree where for every node, keys in the left subtree are smaller and keys in the right subtree are larger. Efficient search/insert/delete rely on the BST property.  
  
An Expression Tree represents arithmetic expressions as a binary tree: operators are internal nodes and operands are leaves. Given a postfix expression, we can build the tree using a stack: push operands as single-node trees; on an operator, pop two trees, make them children (left = second pop, right = first pop), and push the new tree.

## Algorithms (High Level):

A) BST Operations

• Insert(key): Recurse/iterate: go left if key < node->key, else right; insert at NULL.  
• Search(key): Traverse as per BST property; return node if found.  
• Delete(key): Three cases—(1) leaf, (2) one child, (3) two children (replace with inorder successor).  
• Traversals: Inorder (LNR), Preorder (NLR), Postorder (LRN).

B) Expression Tree from Postfix

• For each token t in postfix:  
 - If operand: create node and push pointer onto stack.  
 - If operator: pop right, then left; create node with op = t; set left/right; push back.  
• Final stack top is the root. Inorder gives infix (add parentheses for clarity).

## Program Code A (C Language): BST using Linked List

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// BST Node structure

typedef struct BSTNode {

    int key;

    struct BSTNode \*left;

    struct BSTNode \*right;

} BSTNode;

BSTNode \*root = NULL;

// Function to create a new BST node

BSTNode\* create\_bst\_node(int key) {

    BSTNode \*new\_node = (BSTNode \*)malloc(sizeof(BSTNode));

    if (new\_node == NULL) {

        printf("Memory allocation failed.\n");

        exit(EXIT\_FAILURE);

    }

    new\_node->key = key;

    new\_node->left = new\_node->right = NULL;

    return new\_node;

}

// Function to insert a key into the BST (Recursive)

BSTNode\* insert\_bst(BSTNode \*node, int key) {

    if (node == NULL) {

        return create\_bst\_node(key);

    }

    if (key < node->key) {

        node->left = insert\_bst(node->left, key);

    } else if (key > node->key) {

        node->right = insert\_bst(node->right, key);

    }

    return node;

}

// Helper function to find the node with the minimum key (Inorder Successor)

BSTNode\* find\_min(BSTNode \*node) {

    BSTNode \*current = node;

    while (current != NULL && current->left != NULL) {

        current = current->left;

    }

    return current;

}

// Function to delete a key from the BST (Recursive)

BSTNode\* delete\_bst(BSTNode \*node, int key) {

    if (node == NULL) {

        return node;

    }

    // 1. Traverse to find the node

    if (key < node->key) {

        node->left = delete\_bst(node->left, key);

    } else if (key > node->key) {

        node->right = delete\_bst(node->right, key);

    } else {

        // 2. Node found: Handle deletion cases

        // Case 1 & 2: Node with only one child or no child

        if (node->left == NULL) {

            BSTNode \*temp = node->right;

            free(node);

            return temp;

        } else if (node->right == NULL) {

            BSTNode \*temp = node->left;

            free(node);

            return temp;

        }

        // Case 3: Node with two children (Find Inorder Successor)

        BSTNode \*temp = find\_min(node->right);

        // Copy the inorder successor's content to this node

        node->key = temp->key;

        // Delete the inorder successor from the right subtree

        node->right = delete\_bst(node->right, temp->key);

    }

    return node;

}

// Traversal Functions (LNR, NLR, LRN)

// Inorder Traversal (LNR: Left, Node, Right) - Gives sorted output

void inorder\_bst(BSTNode \*node) {

    if (node != NULL) {

        inorder\_bst(node->left);

        printf("%d ", node->key);

        inorder\_bst(node->right);

    }

}

// Preorder Traversal (NLR: Node, Left, Right)

void preorder\_bst(BSTNode \*node) {

    if (node != NULL) {

        printf("%d ", node->key);

        preorder\_bst(node->left);

        preorder\_bst(node->right);

    }

}

// Postorder Traversal (LRN: Left, Right, Node)

void postorder\_bst(BSTNode \*node) {

    if (node != NULL) {

        postorder\_bst(node->left);

        postorder\_bst(node->right);

        printf("%d ", node->key);

    }

}

// Function to free the memory (Postorder-like cleanup)

void free\_bst(BSTNode \*node) {

    if (node != NULL) {

        free\_bst(node->left);

        free\_bst(node->right);

        free(node);

    }

}

int main() {

    printf("--- Binary Search Tree (BST) ADT ---\n");

    // Operations: Insert 50, 30, 70, 20, 40, 60, 80

    root = insert\_bst(root, 50);

    insert\_bst(root, 30);

    insert\_bst(root, 70);

    insert\_bst(root, 20);

    insert\_bst(root, 40);

    insert\_bst(root, 60);

    insert\_bst(root, 80);

    printf("Inorder Traversal (Before Delete): ");

    inorder\_bst(root); // Output: 20 30 40 50 60 70 80

    printf("\n");

    printf("Preorder Traversal: ");

    preorder\_bst(root); // Output: 50 30 20 40 70 60 80

    printf("\n");

    printf("Deleting 70\n");

    root = delete\_bst(root, 70); // Case 2 or 3 deletion

    printf("Inorder Traversal (After Delete 70): ");

    inorder\_bst(root); // Output: 20 30 40 50 60 80 (assuming 80 replaces 70)

    printf("\n");

    printf("Deleting 50 (Root)\n");

    root = delete\_bst(root, 50); // Case 3 deletion (60 replaces 50)

    printf("Inorder Traversal (After Delete 50): ");

    inorder\_bst(root); // Output: 20 30 40 60 80

    printf("\n");

    // Clean up memory

    free\_bst(root);

    root = NULL;

    return 0;

}

## Program Code B (C Language): Construct Expression Tree from Postfix

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include <ctype.h>

// Expression Tree Node structure

typedef struct ExpNode {

    char data;

    struct ExpNode \*left;

    struct ExpNode \*right;

} ExpNode;

// Stack structure for node pointers

#define STACK\_SIZE 50

ExpNode \*stack[STACK\_SIZE];

int top = -1;

// Stack operations

void push(ExpNode \*node) {

    if (top >= STACK\_SIZE - 1) {

        printf("Stack Overflow\n");

        exit(EXIT\_FAILURE);

    }

    stack[++top] = node;

}

ExpNode\* pop() {

    if (top < 0) {

        printf("Stack Underflow\n");

        exit(EXIT\_FAILURE);

    }

    return stack[top--];

}

// Function to create a new Expression Node

ExpNode\* create\_exp\_node(char data) {

    ExpNode \*new\_node = (ExpNode \*)malloc(sizeof(ExpNode));

    if (new\_node == NULL) {

        printf("Memory allocation failed.\n");

        exit(EXIT\_FAILURE);

    }

    new\_node->data = data;

    new\_node->left = new\_node->right = NULL;

    return new\_node;

}

// Function to check if a character is an operator

bool is\_operator(char c) {

    return (c == '+' || c == '-' || c == '\*' || c == '/');

}

// Function to construct the Expression Tree from Postfix

ExpNode\* construct\_expression\_tree(const char \*postfix) {

    top = -1; // Reset stack

    for (int i = 0; postfix[i] != '\0'; i++) {

        char token = postfix[i];

        if (!is\_operator(token)) {

            // Operand: push a single-node tree

            push(create\_exp\_node(token));

        } else {

            // Operator: pop two operands, form a new tree, push back

            ExpNode \*right\_operand = pop(); // First pop is right child

            ExpNode \*left\_operand = pop();  // Second pop is left child

            ExpNode \*operator\_node = create\_exp\_node(token);

            operator\_node->left = left\_operand;

            operator\_node->right = right\_operand;

            push(operator\_node);

        }

    }

    // The final stack top is the root of the Expression Tree

    return pop();

}

// Traversal Functions

// Inorder Traversal (LNR) - gives infix expression with parentheses

void inorder\_exp(ExpNode \*node) {

    if (node == NULL) return;

    // Add parentheses for clarity in the infix expression

    if (is\_operator(node->data)) {

        printf("(");

    }

    inorder\_exp(node->left);

    printf("%c", node->data);

    inorder\_exp(node->right);

    if (is\_operator(node->data)) {

        printf(")");

    }

}

// Preorder Traversal (NLR) - gives prefix expression

void preorder\_exp(ExpNode \*node) {

    if (node == NULL) return;

    printf("%c", node->data);

    preorder\_exp(node->left);

    preorder\_exp(node->right);

}

// Postorder Traversal (LRN) - gives back the postfix expression

void postorder\_exp(ExpNode \*node) {

    if (node == NULL) return;

    postorder\_exp(node->left);

    postorder\_exp(node->right);

    printf("%c", node->data);

}

// Function to free the memory

void free\_exp\_tree(ExpNode \*node) {

    if (node != NULL) {

        free\_exp\_tree(node->left);

        free\_exp\_tree(node->right);

        free(node);

    }

}

int main() {

    printf("--- Expression Tree Construction ---\n");

    const char \*postfix\_exp = "AB+C\*D-";

    printf("Input Postfix: %s\n", postfix\_exp);

    ExpNode \*root = construct\_expression\_tree(postfix\_exp);

    printf("Output:\n");

    printf("Inorder (infix) : ");

    inorder\_exp(root); // Expected: (((A+B)\*C)-D)

    printf("\n");

    printf("Preorder (prefix): ");

    preorder\_exp(root); // Expected: -\*+ABCD

    printf("\n");

    printf("Postorder (postfix): ");

    postorder\_exp(root); // Expected: AB+C\*D-

    printf("\n");

    // Clean up memory

    free\_exp\_tree(root);

    return 0;

}

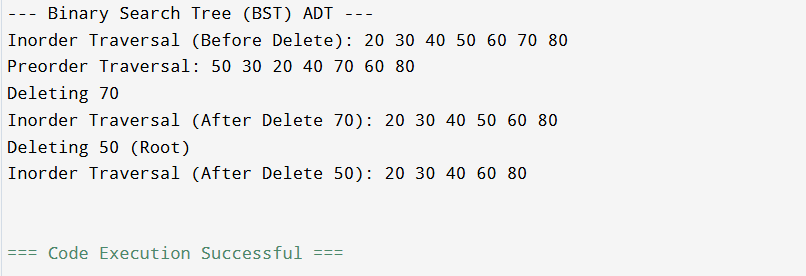
## Sample Input/Output:

BST Sample:  
Operations: Insert 50, 30, 70, 20, 40, 60, 80; Inorder; Delete 70; Inorder  
Output (Inorder before): 20 30 40 50 60 70 80  
Deleting 70  
Output (Inorder after): 20 30 40 50 60 80  
  
Expression Tree Sample:  
Input Postfix: AB+C\*D-  
Output:  
Inorder (infix) : (((A+B)\*C)-D)  
Preorder (prefix): -\*+ABCD  
Postorder : AB+C\*D-

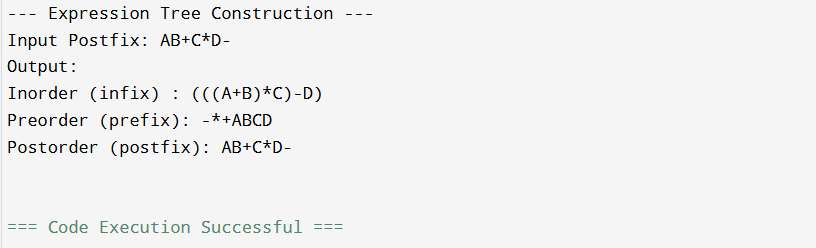
## Result:

Successfully implemented BST ADT with linked nodes and constructed an Expression Tree from postfix expression with correct traversals.

Program A Output:



Program B Output:



## Conclusion:

BSTs provide efficient ordered data operations using structural properties, while expression trees encode operator precedence implicitly. Stack-based construction from postfix is simple and robust.

## Post-Lab Problem: Evaluate an Expression Tree

Objective:

Extend Program B to evaluate the expression tree assuming operands are single-digit non-negative integers.

Description:

After building the tree from a postfix string like '35+2\*', perform a recursive evaluation:  
• If node is a digit -> return its value.  
• Else -> evaluate left and right subtrees and apply the operator at the node.  
Support +, -, \*, /, ^ with integer arithmetic.

Input:

A postfix expression, e.g., '35+26+\*4/'.

Output:

The evaluated integer result.

Constraints:

• Operands are digits 0–9. • Expression is valid. • Division is integer division.

Sample I/O:

Input: 35+26+\*4/  
Output: 16

Hints:

• Write int eval(node) that returns value of subtree.  
• Use switch-case on operator in internal nodes.  
• Reuse the tree built from postfix in Program B.

Post Lab Code:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <math.h>

#include <stdbool.h>

// Expression Tree Node structure

typedef struct ExpNode {

    char data;

    struct ExpNode \*left;

    struct ExpNode \*right;

} ExpNode;

// Stack structure for node pointers (reused from Program B)

#define STACK\_SIZE 50

ExpNode \*stack[STACK\_SIZE];

int top = -1;

// Stack operations (minimal for this problem)

void push(ExpNode \*node) {

    if (top >= STACK\_SIZE - 1) exit(EXIT\_FAILURE);

    stack[++top] = node;

}

ExpNode\* pop() {

    if (top < 0) exit(EXIT\_FAILURE);

    return stack[top--];

}

// Function to create a new Expression Node

ExpNode\* create\_exp\_node(char data) {

    ExpNode \*new\_node = (ExpNode \*)malloc(sizeof(ExpNode));

    if (new\_node == NULL) exit(EXIT\_FAILURE);

    new\_node->data = data;

    new\_node->left = new\_node->right = NULL;

    return new\_node;

}

// Function to check if a character is an operator

bool is\_operator(char c) {

    return (c == '+' || c == '-' || c == '\*' || c == '/' || c == '^');

}

// Function to construct the Expression Tree from Postfix

ExpNode\* construct\_expression\_tree\_eval(const char \*postfix) {

    top = -1;

    for (int i = 0; postfix[i] != '\0'; i++) {

        char token = postfix[i];

        if (!is\_operator(token)) {

            // Operand (digit): push a single-node tree

            push(create\_exp\_node(token));

        } else {

            // Operator

            ExpNode \*right\_operand = pop();

            ExpNode \*left\_operand = pop();

            ExpNode \*operator\_node = create\_exp\_node(token);

            operator\_node->left = left\_operand;

            operator\_node->right = right\_operand;

            push(operator\_node);

        }

    }

    return pop();

}

// --- Evaluation Function (Post-Lab Core) ---

// Function to recursively evaluate the expression tree

int evaluate\_expression\_tree(ExpNode \*node) {

    if (node == NULL) {

        return 0;

    }

    // Check if the node is a leaf (operand/digit)

    if (isdigit(node->data)) {

        // Convert char digit to int value

        return node->data - '0';

    }

    // Node is an internal node (operator)

    int left\_val = evaluate\_expression\_tree(node->left);

    int right\_val = evaluate\_expression\_tree(node->right);

    switch (node->data) {

        case '+': return left\_val + right\_val;

        case '-': return left\_val - right\_val;

        case '\*': return left\_val \* right\_val;

        case '/':

            if (right\_val == 0) {

                printf("Error: Division by zero\n");

                exit(EXIT\_FAILURE);

            }

            return left\_val / right\_val; // Integer division

        case '^':

            // Use pow for exponentiation

            return (int)pow((double)left\_val, (double)right\_val);

        default:

            printf("Error: Invalid operator %c\n", node->data);

            exit(EXIT\_FAILURE);

            return 0;

    }

}

// Function to free the memory

void free\_exp\_tree(ExpNode \*node) {

    if (node != NULL) {

        free\_exp\_tree(node->left);

        free\_exp\_tree(node->right);

        free(node);

    }

}

int main() {

    printf("--- Post-Lab: Evaluate an Expression Tree ---\n");

    // Input Postfix: 35+26+\*4/

    const char \*postfix\_exp = "35+26+\*4/";

    printf("Input Postfix: %s\n", postfix\_exp);

    // 1. Construct the Expression Tree

    ExpNode \*root = construct\_expression\_tree\_eval(postfix\_exp);

    // 2. Evaluate the Tree

    int result = evaluate\_expression\_tree(root);

    // Expected Steps:

    // (3+5) = 8

    // (2+6) = 8

    // 8 \* 8 = 64

    // 64 / 4 = 16

    printf("Output: %d\n", result); // Expected: 16

    // Clean up memory

    free\_exp\_tree(root);

    return 0;

}

Post Lab Output:

