DSA LAB PROGRAMS

1. Write a C program to merge contents of two files containing USNs of students in a sorted order in to the third file such that the third file contains Unique USNs. Program should also display common USNs in both the files.

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


// Function to merge two files and display common strings

void mergeFileAndDisplayCommonStrings(FILE *file1, FILE *file2, FILE *outputFile) {

  char str1[15], str2[15];

  int commonExists = 0;


  // Read initial lines from both files

  char *read1 = fgets(str1, 15, file1);

  char *read2 = fgets(str2, 15, file2);


  // Remove newline characters if present

  if (read1) str1[strcspn(str1, "\n")] = '\0';

  if (read2) str2[strcspn(str2, "\n")] = '\0';


  // Merge files until both are completely read

  while (read1 != NULL && read2 != NULL) {

    int cmp = strcmp(str1, str2);

    if (cmp < 0) {

      fprintf(outputFile, "%s\n", str1);

      read1 = fgets(str1, 15, file1);

      if (read1) str1[strcspn(str1, "\n")] = '\0';

    } else if (cmp > 0) {

      fprintf(outputFile, "%s\n", str2);

      read2 = fgets(str2, 15, file2);

      if (read2) str2[strcspn(str2, "\n")] = '\0';

    } else { // str1 == str2

      fprintf(outputFile, "%s\n", str1);
```

```
            printf("Common string: %s\n", str1);
            commonExists = 1;
            read1 = fgets(str1, 15, file1);
            if (read1) str1[strcspn(str1, "\n")] = '\0';
            read2 = fgets(str2, 15, file2);
            if (read2) str2[strcspn(str2, "\n")] = '\0';
        }
    }

    // Write remaining lines from file1
    while (read1 != NULL) {
        fprintf(outputFile, "%s\n", str1);
        read1 = fgets(str1, 15, file1);
        if (read1) str1[strcspn(str1, "\n")] = '\0';
    }

    // Write remaining lines from file2
    while (read2 != NULL) {
        fprintf(outputFile, "%s\n", str2);
        read2 = fgets(str2, 15, file2);
        if (read2) str2[strcspn(str2, "\n")] = '\0';
    }

    // If no common strings were found
    if (!commonExists) {
        printf("No common strings found\n");
    }
}

int main() {
    FILE *file1, *file2, *outputFile;

    // Open the first input file
    file1 = fopen("USN1.txt", "r");
    if (file1 == NULL) {
        printf("Could not open file1.txt\n");
        return 1;
    }

    // Open the second input file
    file2 = fopen("USN2.txt", "r");
    if (file2 == NULL) {
        printf("Could not open file2.txt\n");
        fclose(file1);
        return 1;
    }

    // Open the output file
    outputFile = fopen("USN3.txt", "w");
    if (outputFile == NULL) {
        printf("Could not open outputFile.txt\n");
        fclose(file1);
        fclose(file2);
        return 1;
    }

    // Call the function to merge and find common strings
    mergeFileAndDisplayCommonStrings(file1, file2, outputFile);

    // Close all files
    fclose(file1);
    fclose(file2);
    fclose(outputFile);

    return 0;
}
```

2.Consider a calculator that needs to perform checking the correctness of parenthesized arithmetic expression and convert the same to postfix expression for evaluation. Develop and execute a program in C using suitable data structures to perform the same and print both the expressions. The input expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and /(divide)

```c
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>


#define MAX 100


// Stack structure

typedef struct {

    char arr[MAX];

    int top;

} Stack;


// Function to initialize the stack

void initStack(Stack* s) {

    s->top = -1;

}


// Function to check if the stack is empty

int isEmpty(Stack* s) {

    return s->top == -1;

}


// Function to check if the stack is full

int isFull(Stack* s) {

    return s->top == MAX - 1;

}


// Function to push an element onto the stack

void push(Stack* s, char val) {
    if (isFull(s)) {
        printf("Stack Overflow\n");
        return;
    }
    s->arr[++(s->top)] = val;
}

// Function to pop an element from the stack
char pop(Stack* s) {
    if (isEmpty(s)) {
        printf("Stack Underflow\n");
        return '\0';
    }
    return s->arr[(s->top)--];
}

// Function to peek the top element of the stack
char peek(Stack* s) {
    if (isEmpty(s)) {
        return '\0';
    }
    return s->arr[s->top];
}

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

// Function to determine operator precedence
int precedence(char op) {
    switch (op) {
        case '^': return 3;
        case '*':
        case '/': return 2;
        case '+':
        case '-': return 1;
        default: return 0;
    }
}

// Function to check if parentheses in the expression are balanced
int areParenthesesBalanced(const char* exp) {
    Stack stack;
    initStack(&stack);

    for (int i = 0; exp[i] != '\0'; i++) {
        if (exp[i] == '(') {
            push(&stack, '(');
        } else if (exp[i] == ')') {
            if (isEmpty(&stack)) {
                return 0; // Unbalanced
            }
            pop(&stack);
        }
    }
    return isEmpty(&stack);
}

// Function to convert infix expression to postfix
void infixToPostfix(const char* infix, char* postfix) {
    Stack stack;
    initStack(&stack);
    int j = 0;

    for (int i = 0; infix[i] != '\0'; i++) {
        char ch = infix[i];

        // If operand, add to postfix
        if (isalnum(ch)) {
            postfix[j++] = ch;
        }
        // If '(', push to stack
        else if (ch == '(') {
            push(&stack, ch);
        }
        // If ')', pop and add to postfix until '(' is found
        else if (ch == ')') {
            while (!isEmpty(&stack) && peek(&stack) != '(') {
                postfix[j++] = pop(&stack);
            }
            pop(&stack); // Remove '('
        }
        // If operator, pop higher or equal precedence operators and push current operator
        else if (isOperator(ch)) {
            while (!isEmpty(&stack) && precedence(peek(&stack)) >= precedence(ch)) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, ch);
        }
    }

    // Pop remaining operators
    while (!isEmpty(&stack)) {
        postfix[j++] = pop(&stack);
    }

    postfix[j] = '\0'; // Null-terminate the postfix expression
}
```

```
int main() {
    char infix[MAX], postfix[MAX];

    printf("Enter an infix expression: ");
    scanf("%s", infix);

    if (!areParenthesesBalanced(infix)) {
        printf("Invalid expression: Unbalanced parentheses\n");
        return 1;
    }

    infixToPostfix(infix, postfix);

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

    return 0;
}
```

3.A calculator needs to evaluate a postfix expression. Develop and execute a program in C using a suitable data structure to evaluate a valid postfix expression. Assume that the postfix expression is read as a single line consisting of non-negative single digit operands and binary arithmetic operators. The arithmetic operators are + (add), - (subtract), * (multiply) and / (divide).

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <math.h>


#define MAX 100


// Stack implementation

typedef struct {

    int items[MAX];

    int top;

} Stack;


void initStack(Stack *s) {

    s->top = -1;

}


int isFull(Stack *s) {

    return s->top == MAX - 1;

}


int isEmpty(Stack *s) {

    return s->top == -1;

}


void push(Stack *s, int value) {

    if (isFull(s)) {

        printf("Stack overflow\n");

        exit(EXIT_FAILURE);

    }
    s->items[++(s->top)] = value;
}


int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        exit(EXIT_FAILURE);
    }
    return s->items[(s->top)--];
}


// Function to evaluate postfix expression
int evaluatePostfix(const char *expression) {
    Stack stack;
    initStack(&stack);


    for (int i = 0; expression[i] != '\0'; i++) {
        char ch = expression[i];


        // If the character is a digit, push it to the stack
        if (isdigit(ch)) {
            push(&stack, ch - '0');
        } else {
            // Pop two operands from stack and perform the operation
            int val2 = pop(&stack);
            int val1 = pop(&stack);


            switch (ch) {
                case '+':
                    push(&stack, val1 + val2);
                    break;
                case '-':
                    push(&stack, val1 - val2);
                    break;
                case '*':
                    push(&stack, val1 * val2);
                    break;
                case '/':
                    if (val2 == 0) {
                        printf("Division by zero error\n");
                        exit(EXIT_FAILURE);
                    }
                    push(&stack, val1 / val2);
                    break;
                case '^':
                    push(&stack, pow(val1, val2));
                    break;
                default:
                    printf("Invalid operator: %c\n", ch);
                    exit(EXIT_FAILURE);
            }
        }
    }
    return pop(&stack);
}
int main() {
    char postfixExpression[MAX];

    printf("Enter a postfix expression: ");
    scanf("%s", postfixExpression);

    int result = evaluatePostfix(postfixExpression);
    printf("The result of the postfix expression is: %d\n", result);

    return 0;
}
```

4.Write a C program to simulate the working of Messaging System in which a message is placed in a Queue by a Message Sender, a message is removed from the queue by a Message Receiver, which can also display the contents of the Queue.

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX 100   // Maximum number of messages in the queue

#define MSG_SIZE 256 // Maximum size of a single message


// Define the queue structure

typedef struct {

    char messages[MAX][MSG_SIZE]; // Array to store messages

    int front;                 // Index of the front of the queue

    int rear;                  // Index of the rear of the queue

    int count;                 // Number of messages in the queue

} MessageQueue;


// Initialize the queue

void initQueue(MessageQueue *q) {

    q->front = 0;

    q->rear = -1;

    q->count = 0;

}


// Check if the queue is full

int isFull(MessageQueue *q) {

    return q->count == MAX;

}


// Check if the queue is empty

int isEmpty(MessageQueue *q) {

    return q->count == 0;

}
```

```
// Enqueue a message
void enqueue(MessageQueue *q, char *message) {
    if (isFull(q)) {
        printf("Queue is full! Cannot add more messages.\n");
        return;
    }
    q->rear = (q->rear + 1) % MAX;
    strncpy(q->messages[q->rear], message, MSG_SIZE);
    q->messages[q->rear][MSG_SIZE - 1] = '\0'; // Ensure null-termination
    q->count++;
    printf("Message added to the queue.\n");
}

// Dequeue a message
void dequeue(MessageQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty! No messages to remove.\n");
        return;
    }
    printf("Message removed: %s\n", q->messages[q->front]);
    q->front = (q->front + 1) % MAX;
    q->count--;
}

// Display the contents of the queue
void displayQueue(MessageQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty!\n");
        return;
    }
    printf("Messages in the queue:\n");
    for (int i = 0; i < q->count; i++) {
        int index = (q->front + i) % MAX;
        printf("%d: %s\n", i + 1, q->messages[index]);
    }
}

int main() {
    MessageQueue queue;
    initQueue(&queue);

    int choice;
    char message[MSG_SIZE];

    do {
        printf("\nMessaging System:\n");
        printf("1. Add Message\n");
        printf("2. Remove Message\n");
        printf("3. Display Messages\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // Consume newline character

        switch (choice) {
            case 1:
                printf("Enter the message: ");
                fgets(message, MSG_SIZE, stdin);
                message[strcspn(message, "\n")] = '\0'; // Remove trailing newline
                enqueue(&queue, message);
                break;

            case 2:
                dequeue(&queue);
                break;
            case 3:
                displayQueue(&queue);
                break;

            case 4:
                printf("Exiting Messaging System.\n");
                break;

            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 4);

    return 0;
}
```

5. Consider a super market scenario where sales manager wants to search for the customer details using a customer-id. Customer information like (custid, custname, & custphno) are stored as a structure, and custid will be used as hash key. Develop and execute a program in C using suitable data structures to implement the following operations:

a. Insertion of a new data entry. b. Search for customer information using custid. c. Display the records. (Demonstrate collision and its handling using linear probing method).

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define TABLE_SIZE 10

#define NAME_SIZE 50

#define PHONE_SIZE 15


// Define the customer structure

typedef struct {

    int custid;

    char custname[NAME_SIZE];

    char custphno[PHONE_SIZE];

} Customer;


// Define the hash table entry

typedef struct {

    Customer customer;

    int isOccupied; // 0 for empty, 1 for occupied

} HashTableEntry;


// Hash table

HashTableEntry hashTable[TABLE_SIZE];


// Initialize the hash table

void initHashTable() {

    for (int i = 0; i < TABLE_SIZE; i++) {

        hashTable[i].isOccupied = 0;
    }
}

// Hash function
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Insert a new customer into the hash table
void insertCustomer(int custid, char *custname, char *custphno) {
    int index = hashFunction(custid);
    int originalIndex = index;
    int i = 0;

    while (hashTable[index].isOccupied) {
        if (hashTable[index].customer.custid == custid) {
            printf("Customer ID %d already exists!\n", custid);
            return;
        }
        index = (originalIndex + ++i) % TABLE_SIZE;
        if (index == originalIndex) {
            printf("Hash table is full! Cannot insert new customer.\n");
            return;
        }
    }

    hashTable[index].customer.custid = custid;
    strncpy(hashTable[index].customer.custname, custname, NAME_SIZE);
    strncpy(hashTable[index].customer.custphno, custphno, PHONE_SIZE);
    hashTable[index].isOccupied = 1;

    printf("Customer inserted successfully at index %d.\n", index);
}
```

```
// Search for a customer by customer ID
void searchCustomer(int custid) {
    int index = hashFunction(custid);
    int originalIndex = index;
    int i = 0;

    while (hashTable[index].isOccupied) {
        if (hashTable[index].customer.custid == custid) {
            printf("Customer found at index %d:\n", index);
            printf("ID: %d, Name: %s, Phone: %s\n",
                   hashTable[index].customer.custid,
                   hashTable[index].customer.custname,
                   hashTable[index].customer.custphno);
            return;
        }
        index = (originalIndex + ++i) % TABLE_SIZE;
        if (index == originalIndex) {
            break;
        }
    }

    printf("Customer ID %d not found!\n", custid);
}

// Display all records in the hash table
void displayRecords() {
    printf("\nHash Table Records:\n");
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i].isOccupied) {
            printf("Index %d: ID: %d, Name: %s, Phone: %s\n",
                   i, hashTable[i].customer.custid,
                   hashTable[i].customer.custname,
                hashTable[i].customer.custphno);
        } else {
            printf("Index %d: Empty\n", i);
        }
    }
}

int main() {
    initHashTable();

    int choice, custid;
    char custname[NAME_SIZE], custphno[PHONE_SIZE];

    do {
        printf("\nSupermarket Customer Management:\n");
        printf("1. Insert Customer\n");
        printf("2. Search Customer\n");
        printf("3. Display Records\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        getchar(); // Consume newline character

        switch (choice) {
            case 1:
                printf("Enter Customer ID: ");
                scanf("%d", &custid);
                getchar();
                printf("Enter Customer Name: ");
                fgets(custname, NAME_SIZE, stdin);
                custname[strcspn(custname, "\n")] = '\0'; // Remove trailing newline
                printf("Enter Customer Phone: ");
                fgets(custphno, PHONE_SIZE, stdin);
                custphno[strcspn(custphno, "\n")] = '\0'; // Remove trailing newline
                insertCustomer(custid, custname, custphno);
                break;

            case 2:
                printf("Enter Customer ID to search: ");
                scanf("%d", &custid);
                searchCustomer(custid);
                break;

            case 3:
                displayRecords();
                break;

            case 4:
                printf("Exiting program.\n");
                break;

            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 4);

    return 0;
}
```

6.Consider a warehouse where the items have to be arranged in an ascending order. Develop and execute a program in C using suitable data structures to implement warehouse such that items can be traced easily.

```
#include <stdio.h>

#include <stdlib.h>


// Define a structure for a node

typedef struct Node {

    int itemId;

    struct Node *next;

} Node;


// Function to create a new node

Node *createNode(int id) {

    Node *newNode = (Node *)malloc(sizeof(Node));

    if (newNode == NULL) {

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

        exit(1);

    }

    newNode->itemId = id;

    newNode->next = NULL;

    return newNode;

}


// Function to insert a node in ascending order

Node *insertNodeSorted(Node *head, int id) {

    Node *newNode = createNode(id);


    // If the list is empty or the new node should be the first

    if (head == NULL || head->itemId > id) {

        newNode->next = head;

        return newNode;

    }
    // Traverse the list to find the correct position
    Node *current = head;
    while (current->next != NULL && current->next->itemId < id) {
        current = current->next;
    }

    newNode->next = current->next;
    current->next = newNode;
    return head;
}

// Function to display all nodes
void displayNodes(Node *head) {
    if (head == NULL) {
        printf("The warehouse is empty.\n");
        return;
    }

    printf("Items in the warehouse:\n");
    while (head != NULL) {
        printf("ID: %d\n", head->itemId);
        head = head->next;
    }
}

// Function to search for a node by ID
void searchNode(Node *head, int id) {
    while (head != NULL) {
        if (head->itemId == id) {
            printf("Item found: ID: %d\n", head->itemId);
            return;
        }
        head = head->next;
    }
    printf("Item with ID %d not found.\n", id);
}

// Main function to manage the warehouse
int main() {
    Node *warehouse = NULL;
    int choice, id;

    do {
        printf("\nWarehouse Management:\n");
        printf("1. Add Item\n");
        printf("2. Display Items\n");
        printf("3. Search Item\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter Item ID: ");
                scanf("%d", &id);
                warehouse = insertNodeSorted(warehouse, id);
                printf("Item added successfully.\n");
                break;

            case 2:
                displayNodes(warehouse);
                break;

            case 3:
                printf("Enter Item ID to search: ");
                scanf("%d", &id);
                searchNode(warehouse, id);
                break;

            case 4:
                printf("Exiting program.\n");
                break;

            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 4);

    // Free allocated memory
    while (warehouse != NULL) {
        Node *temp = warehouse;
        warehouse = warehouse->next;
        free(temp);
    }

    return 0;
}
```

7. Consider a polynomial addition for two polynomials. Develop and execute a program in C using suitable data structures to implement the same.

```
#include <stdio.h>

#include <stdlib.h>


struct Node {

    int coeff;

    int exp;

    struct Node* next;

};


// Function to create a new node

struct Node* createNode(int coeff, int exp) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->coeff = coeff;

    newNode->exp = exp;

    newNode->next = NULL;

    return newNode;

}


// Function to insert a node at the end of the polynomial

void insertTerm(struct Node** poly, int coeff, int exp) {

    struct Node* newNode = createNode(coeff, exp);

    if (*poly == NULL) {

        *poly = newNode;

    } else {

        struct Node* temp = *poly;

        while (temp->next != NULL) {

            temp = temp->next;

        }

        temp->next = newNode;

    }

}
```

```
// Function to add two polynomials
struct Node* addPolynomials(struct Node* poly1, struct Node* poly2) {
    struct Node* result = NULL;
    struct Node *temp1 = poly1, *temp2 = poly2;

    while (temp1 != NULL && temp2 != NULL) {
        if (temp1->exp > temp2->exp) {
            insertTerm(&result, temp1->coeff, temp1->exp);
            temp1 = temp1->next;
        } else if (temp1->exp < temp2->exp) {
            insertTerm(&result, temp2->coeff, temp2->exp);
            temp2 = temp2->next;
        } else {
            int sumCoeff = temp1->coeff + temp2->coeff;
            if (sumCoeff != 0) {
                insertTerm(&result, sumCoeff, temp1->exp);
            }
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
    }

    while (temp1 != NULL) {
        insertTerm(&result, temp1->coeff, temp1->exp);
        temp1 = temp1->next;
    }

    while (temp2 != NULL) {
        insertTerm(&result, temp2->coeff, temp2->exp);
        temp2 = temp2->next;
    }
    return result;
}

// Function to print the polynomial
void printPolynomial(struct Node* poly) {
    if (poly == NULL) {
        printf("0");
        return;
    }

    struct Node* temp = poly;
    while (temp != NULL) {
        if (temp->coeff > 0 && temp != poly) {
            printf(" + ");
        }
        printf("%dx^%d", temp->coeff, temp->exp);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Node *poly1 = NULL, *poly2 = NULL, *result = NULL;

    // Inserting terms into first polynomial: 5x^3 + 4x^2 + 2x + 1
    insertTerm(&poly1, 5, 3);
    insertTerm(&poly1, 4, 2);
    insertTerm(&poly1, 2, 1);
    insertTerm(&poly1, 1, 0);

    // Inserting terms into second polynomial: 3x^3 + 2x^2 + x + 4
    insertTerm(&poly2, 3, 3);
    insertTerm(&poly2, 2, 2);
    insertTerm(&poly2, 1, 1);
    insertTerm(&poly2, 4, 0);

    // Adding the two polynomials
    result = addPolynomials(poly1, poly2);

    // Printing the result
    printf("First Polynomial: ");
    printPolynomial(poly1);
    printf("Second Polynomial: ");
    printPolynomial(poly2);
    printf("Resultant Polynomial: ");
    printPolynomial(result);

    return 0;
}
```

8. Develop and execute a program in C to perform following operations on binary search tree: a. To count number of non terminal nodes. b. To count number of terminal nodes. c. To count nodes with degree 2. d. To count total number of nodes.

```
#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 = 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 {

        root->right = insert(root->right, data);

    }

    return root;

}
```

```
// Function to count the total number of nodes
int countTotalNodes(struct Node* root) {
    if (root == NULL) {
        return 0;
    }
    return 1 + countTotalNodes(root->left) + countTotalNodes(root->right);
}

// Function to count the number of non-terminal nodes (nodes with at least one child)
int countNonTerminalNodes(struct Node* root) {
    if (root == NULL) {
        return 0;
    }
    int count = 0;
    if (root->left != NULL || root->right != NULL) {
        count = 1;
    }
    return count + countNonTerminalNodes(root->left) + countNonTerminalNodes(root->right);
}
// Function to count the number of terminal nodes (leaf nodes)
int countTerminalNodes(struct Node* root) {
    if (root == NULL) {
        return 0;
    }
    if (root->left == NULL && root->right == NULL) {
        return 1;
    }
    return countTerminalNodes(root->left) + countTerminalNodes(root->right);
}
// Function to count the number of nodes with degree 2 (nodes with both left and right children)
int countNodesWithDegree2(struct Node* root) {
    if (root == NULL) {
        return 0;
    }
    int count = 0;
    if (root->left != NULL && root->right != NULL) {
        count = 1;
    }
    return count + countNodesWithDegree2(root->left) + countNodesWithDegree2(root->right);
}
int main() {
    struct Node* root = NULL;

    // Inserting nodes into the BST
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    // Count total number of nodes
    printf("Total number of nodes: %d\n", countTotalNodes(root));

    // Count non-terminal nodes
    printf("Number of non-terminal nodes: %d\n", countNonTerminalNodes(root));

    // Count terminal nodes (leaf nodes)
    printf("Number of terminal nodes: %d\n", countTerminalNodes(root));

    // Count nodes with degree 2
    printf("Number of nodes with degree 2: %d\n", countNodesWithDegree2(root));

    return 0;
}
```

9. Develop and execute a program in C using suitable data structures to create a binary tree for an expression. The tree traversals in some proper method should result in conversion of original expression into prefix, infix and postfix forms. Display the original expression along with the three different forms also

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>


// Structure for a node in the expression tree

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 = newNode->right = NULL;

    return newNode;

}


// Function to check if the character is an operand (A-Z, a-z, or digits)

int isOperand(char ch) {

    return isalpha(ch) || isdigit(ch);

}


// Function to create the expression tree from a given expression

struct Node* constructTree(char* expression, int* index) {

    // If the expression has ended

    if (expression[*index] == '\0')

        return NULL;


    char ch = expression[*index];
    (*index)++;

    struct Node* node = createNode(ch);

    // If the node is an operand, return it
    if (isOperand(ch)) {
        return node;
    }

    // Otherwise, it's an operator, so we need to build the tree recursively
    node->left = constructTree(expression, index);
    node->right = constructTree(expression, index);

    return node;
}

// Inorder traversal (Infix notation)
void inorder(struct Node* root) {
    if (root == NULL)
        return;
    if (root->left) printf("(");
    inorder(root->left);
    printf("%c", root->data);
    inorder(root->right);
    if (root->right) printf(")");
}

// Preorder traversal (Prefix notation)
void preorder(struct Node* root) {
    if (root == NULL)
        return;
    printf("%c", root->data);
    preorder(root->left);
    preorder(root->right);
}

// Postorder traversal (Postfix notation)
void postorder(struct Node* root) {
    if (root == NULL)
        return;
    postorder(root->left);
    postorder(root->right);
    printf("%c", root->data);
}

int main() {
    char expression[100];
    int index = 0;

    // Input expression (example: +A*BC)
    printf("Enter the expression: ");
    scanf("%s", expression);

    // Construct the expression tree
    struct Node* root = constructTree(expression, &index);

    // Display the original expression and the three forms
    printf("\nOriginal Expression: %s\n", expression);

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

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

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

    return 0;
}
```

10. Develop and execute a program in C using suitable data structures to perform Searching a data item in an ordered list of items in both directions and implement the following operations: a. Create a doubly linked list by adding each node at the start. b. Insert a new node at the end of the list. c. Display the content of a list. Consider an integer number as a data item.

```
#include <stdio.h>

#include <stdlib.h>


// Structure for a doubly linked list node

struct Node {

    int data;

    struct Node* next;

    struct Node* prev;

};


// Function to create a new node

struct Node* createNode(int data) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    newNode->data = data;

    newNode->next = NULL;

    newNode->prev = NULL;

    return newNode;

}


// Function to add a node at the start of the doubly linked list

void insertAtStart(struct Node** head, int data) {

    struct Node* newNode = createNode(data);

    newNode->next = *head;


    if (*head != NULL) {

        (*head)->prev = newNode;

    }


    *head = newNode;

}
```

```c
// Function to insert a new node at the end of the doubly linked list
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);

    if (*head == NULL) {
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->prev = temp;
}

// Function to display the content of the doubly linked list from head to tail
void displayList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    printf("List content (head to tail): ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
```

```
}

// Function to search for a data item in the list (head to tail)
int searchForward(struct Node* head, int data) {
    struct Node* temp = head;
    while (temp != NULL) {
        if (temp->data == data) {
            return 1; // Found
        }
        temp = temp->next;
    }
    return 0; // Not found
}

// Function to search for a data item in the list (tail to head)
int searchBackward(struct Node* head, int data) {
    if (head == NULL) {
        return 0; // List is empty
    }

    struct Node* temp = head;
    // Traverse to the last node
    while (temp->next != NULL) {
        temp = temp->next;
    }

    // Search backward
    while (temp != NULL) {
        if (temp->data == data) {
            return 1; // Found
        }
        temp = temp->prev;
    }
    return 0; // Not found
}
int main() {
    struct Node* head = NULL;

    // Create a doubly linked list by adding nodes at the start
    insertAtStart(&head, 10);
    insertAtStart(&head, 20);
    insertAtStart(&head, 30);

    // Insert a new node at the end of the list
    insertAtEnd(&head, 40);
    insertAtEnd(&head, 50);

    // Display the list content
    displayList(head);
 // Search for an item from head to tail
    int searchItem = 40;
    if (searchForward(head, searchItem)) {
        printf("Item %d found in the list (head to tail).\n", searchItem);
    } else {
        printf("Item %d not found in the list (head to tail).\n", searchItem);
    }
// Search for an item from tail to head
    searchItem = 50;
    if (searchBackward(head, searchItem)) {
        printf("Item %d found in the list (tail to head).\n", searchItem);
    } else {
        printf("Item %d not found in the list (tail to head).\n", searchItem);
    }

    return 0;
}
```