**Develop a Program in C for the following:**

**a) Declare a calendar as an array of 7 elements (A dynamically Created array) to represent**

**7 days of a week. Each Element of the array is a structure having three fields. The first**

**field is the name of the Day (A dynamically allocated String), The second field is the**

**date of the Day (A integer), the third field is the description of the activity for a**

**particular day (A dynamically allocated String).**

**b) Write functions create(), read() and display(); to create the calendar, to read the data**

**from the keyboard and to print weeks activity details report on screen.**

**is this correct for that code**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_DAYS 7

#define MAX\_STRING\_LENGTH 50

// Structure to represent a day's details

struct Day {

    char \*dayName;

    int date;

    char \*activity;

};

// Function to create the calendar

struct Day \*create() {

    struct Day \*calendar = (struct Day \*)malloc(MAX\_DAYS \* sizeof(struct Day));

    for (int i = 0; i < MAX\_DAYS; i++) {

        calendar[i].dayName = (char \*)malloc(MAX\_STRING\_LENGTH \* sizeof(char));

        calendar[i].activity = (char \*)malloc(MAX\_STRING\_LENGTH \* sizeof(char));

    }

    return calendar;

}

// Function to read data from the keyboard

void read(struct Day \*calendar) {

    for (int i = 0; i < MAX\_DAYS; i++) {

        printf("Enter day name: ");

        scanf("%s", calendar[i].dayName);

        printf("Enter date: ");

        scanf("%d", &calendar[i].date);

        printf("Enter activity: ");

        scanf("%s", calendar[i].activity);

    }

}

// Function to display week's activity details report on screen

void display(struct Day \*calendar) {

    printf("\nWeek's Activity Details:\n");

    for (int i = 0; i < MAX\_DAYS; i++) {

        printf("Day: %s, Date: %d, Activity: %s\n", calendar[i].dayName, calendar[i].date, calendar[i].activity);

    }

}

// Function to deallocate memory

void deallocateMemory(struct Day \*calendar) {

    for (int i = 0; i < MAX\_DAYS; i++) {

        free(calendar[i].dayName);

        free(calendar[i].activity);

    }

    free(calendar);

}

int main() {

    struct Day \*weeklyCalendar;

    // Creating the calendar

    weeklyCalendar = create();

    // Reading data from the keyboard

    read(weeklyCalendar);

    // Displaying week's activity details

    display(weeklyCalendar);

    // Deallocate memory

    deallocateMemory(weeklyCalendar);

    return 0;

}

2 Develop a Program in C for the following operations on Strings. a. Read a main String (STR), a Pattern String (PAT) and a Replace String (REP) b. Perform Pattern Matching Operation: Find and Replace all occurrences of PAT in STR with REP if PAT exists in STR. Report suitable messages in case PAT does not exist in STR Support the program with functions for each of the above operations. Don't use Built-in functions.

#include <stdio.h>

#include <string.h>

#define MAX\_LENGTH 100

// Function to find and replace pattern in string

void findAndReplace(char \*str, char \*pattern, char \*replace) {

    int strLength = strlen(str);

    int patLength = strlen(pattern);

    int repLength = strlen(replace);

    for (int i = 0; i <= strLength - patLength; i++) {

        int j;

        // Check for pattern match starting from position i

        for (j = 0; j < patLength; j++) {

            if (str[i + j] != pattern[j]) {

                break;  // Pattern mismatch

            }

        }

        if (j == patLength) {

            // Pattern found at position i

            // Shift characters to the right for replacement

            for (int k = strLength - 1; k >= i + patLength; k--) {

                str[k + repLength - patLength] = str[k];

            }

            // Replace pattern with replacement string

            for (int k = 0; k < repLength; k++) {

                str[i + k] = replace[k];

            }

            // Update string length after replacement

            strLength = strLength + repLength - patLength;

            // Move 'i' to the end of replacement string

            i = i + repLength - 1;

        }

    }

}

int main() {

    char mainString[MAX\_LENGTH];

    char pattern[MAX\_LENGTH];

    char replace[MAX\_LENGTH];

    printf("Enter the main string: ");

    scanf("%s", mainString);

    printf("Enter the pattern to find: ");

    scanf("%s", pattern);

    printf("Enter the replacement string: ");

    scanf("%s", replace);

    // Perform pattern matching and replacement

    findAndReplace(mainString, pattern, replace);

    printf("Resultant string after replacement: %s\n", mainString);

    return 0;

}

3 Develop a menu driven Program in C for the following operations on STACK of Integers

(Array Implementation of Stack with maximum size MAX)

a. Push an Element on to Stack

b. Pop an Element from Stack

c. Demonstrate how Stack can be used to check Palindrome

d. Demonstrate Overflow and Underflow situations on Stack

e. Display the status of Stack

f. Exit

Support the program with appropriate functions for each of the above operations

#include <stdio.h>

#include <stdbool.h>

#define MAX 100

// Stack structure

struct Stack {

    int items[MAX];

    int top;

};

// Initialize the stack

void initialize(struct Stack \*s) {

    s->top = -1;

}

// Check if the stack is full

bool isFull(struct Stack \*s) {

    return s->top == MAX - 1;

}

// Check if the stack is empty

bool isEmpty(struct Stack \*s) {

    return s->top == -1;

}

// Push operation to add an element onto the stack

void push(struct Stack \*s, int value) {

    if (isFull(s)) {

        printf("Stack Overflow - Cannot push element %d\n", value);

    } else {

        s->top++;

        s->items[s->top] = value;

        printf("Element %d pushed to the stack\n", value);

    }

}

// Pop operation to remove an element from the stack

int pop(struct Stack \*s) {

    if (isEmpty(s)) {

        printf("Stack Underflow - Cannot pop element\n");

        return -1;

    } else {

        int removed = s->items[s->top];

        s->top--;

        return removed;

    }

}

// Display elements in the stack

void display(struct Stack \*s) {

    if (isEmpty(s)) {

        printf("Stack is empty\n");

    } else {

        printf("Stack elements: ");

        for (int i = 0; i <= s->top; i++) {

            printf("%d ", s->items[i]);

        }

        printf("\n");

    }

}

// Check if a string is a palindrome using stack

bool isPalindrome(char \*str) {

    struct Stack s;

    initialize(&s);

    int length = 0;

    while (str[length] != '\0') {

        push(&s, str[length]);

        length++;

    }

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

        if (pop(&s) != str[i]) {

            return false;

        }

    }

    return true;

}

int main() {

    struct Stack stack;

    initialize(&stack);

    int choice, element;

    char str[MAX];

    do {

        printf("\nMENU\n");

        printf("1. Push element\n");

        printf("2. Pop element\n");

        printf("3. Check palindrome\n");

        printf("4. Display stack\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter element to push: ");

                scanf("%d", &element);

                push(&stack, element);

                break;

            case 2:

                element = pop(&stack);

                if (element != -1) {

                    printf("Popped element: %d\n", element);

                }

                break;

            case 3:

                printf("Enter a string to check palindrome: ");

                scanf("%s", str);

                if (isPalindrome(str)) {

                    printf("The string is a palindrome\n");

                } else {

                    printf("The string is not a palindrome\n");

                }

                break;

            case 4:

                display(&stack);

                break;

            case 5:

                printf("Exiting the program\n");

                break;

            default:

                printf("Invalid choice\n");

        }

    } while (choice != 5);

    return 0;

}

Develop a Program in C for converting an Infix Expression to Postfix Expression. Program should support for both parenthesized and free parenthesized expressions with the operators: +, -, \*, /, % (Remainder), ^ (Power) and alphanumeric operands.

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <string.h>

#define MAX 100

// Stack structure for operators

struct Stack {

    char items[MAX];

    int top;

};

// Function to initialize the stack

void initialize(struct Stack \*s) {

    s->top = -1;

}

// Function to check if the stack is full

bool isFull(struct Stack \*s) {

    return s->top == MAX - 1;

}

// Function to check if the stack is empty

bool isEmpty(struct Stack \*s) {

    return s->top == -1;

}

// Function to push an element onto the stack

void push(struct Stack \*s, char value) {

    if (isFull(s)) {

        printf("Stack Overflow\n");

    } else {

        s->top++;

        s->items[s->top] = value;

    }

}

// Function to pop an element from the stack

char pop(struct Stack \*s) {

    if (isEmpty(s)) {

        return '\0';

    } else {

        char removed = s->items[s->top];

        s->top--;

        return removed;

    }

}

// Function to check if a character is an operator

bool isOperator(char ch) {

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

}

// Function to get the precedence of an operator

int getPrecedence(char ch) {

    if (ch == '^')

        return 3;

    else if (ch == '\*' || ch == '/' || ch == '%')

        return 2;

    else if (ch == '+' || ch == '-')

        return 1;

    else

        return -1;

}

// Function to convert infix expression to postfix expression

void infixToPostfix(char \*infix, char \*postfix) {

    struct Stack stack;

    initialize(&stack);

    int i = 0, j = 0;

    while (infix[i] != '\0') {

        char current = infix[i];

        if ((current >= 'a' && current <= 'z') || (current >= 'A' && current <= 'Z') || (current >= '0' && current <= '9')) {

            postfix[j++] = current;

        } else if (current == '(') {

            push(&stack, current);

        } else if (current == ')') {

            while (!isEmpty(&stack) && stack.items[stack.top] != '(') {

                postfix[j++] = pop(&stack);

            }

            if (!isEmpty(&stack) && stack.items[stack.top] != '(') {

                printf("Invalid expression\n");

                return;

            } else {

                pop(&stack);

            }

        } else if (isOperator(current)) {

            while (!isEmpty(&stack) && getPrecedence(current) <= getPrecedence(stack.items[stack.top])) {

                postfix[j++] = pop(&stack);

            }

            push(&stack, current);

        }

        i++;

    }

    while (!isEmpty(&stack)) {

        postfix[j++] = pop(&stack);

    }

    postfix[j] = '\0';

}

int main() {

    char infix[MAX];

    char postfix[MAX];

    printf("Enter the infix expression: ");

    scanf("%s", infix);

    infixToPostfix(infix, postfix);

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

    return 0;

}

Develop a Program in C for the following Stack Applications

a. Evaluation of Suffix expression with single digit operands and operators: +, -, \*, /, %, ^

b. Solving Tower of Hanoi problem with n disks

1. Program for Suffix Expression Evaluation:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 100

// Stack structure

struct Stack {

    int items[MAX];

    int top;

};

// Function to initialize the stack

void initialize(struct Stack \*s) {

    s->top = -1;

}

// Function to check if the stack is full

bool isFull(struct Stack \*s) {

    return s->top == MAX - 1;

}

// Function to check if the stack is empty

bool isEmpty(struct Stack \*s) {

    return s->top == -1;

}

// Function to push an element onto the stack

void push(struct Stack \*s, int value) {

    if (isFull(s)) {

        printf("Stack Overflow\n");

    } else {

        s->top++;

        s->items[s->top] = value;

    }

}

// Function to pop an element from the stack

int pop(struct Stack \*s) {

    if (isEmpty(s)) {

        printf("Stack Underflow\n");

        return -1;

    } else {

        int removed = s->items[s->top];

        s->top--;

        return removed;

    }

}

// Function to evaluate suffix expression

int evaluateSuffixExpression(char \*suffix) {

    struct Stack stack;

    initialize(&stack);

    int i = 0;

    while (suffix[i] != '\0') {

        char current = suffix[i];

        if (current >= '0' && current <= '9') {

            push(&stack, current - '0');

        } else {

            int operand2 = pop(&stack);

            int operand1 = pop(&stack);

            switch (current) {

                case '+':

                    push(&stack, operand1 + operand2);

                    break;

                case '-':

                    push(&stack, operand1 - operand2);

                    break;

                case '\*':

                    push(&stack, operand1 \* operand2);

                    break;

                case '/':

                    push(&stack, operand1 / operand2);

                    break;

                case '%':

                    push(&stack, operand1 % operand2);

                    break;

                case '^':

                    int result = 1;

                    for (int i = 0; i < operand2; i++) {

                        result \*= operand1;

                    }

                    push(&stack, result);

                    break;

                default:

                    printf("Invalid operator\n");

                    return -1;

            }

        }

        i++;

    }

    return pop(&stack);

}

int main() {

    char suffix[MAX];

    printf("Enter the suffix expression: ");

    scanf("%s", suffix);

    int result = evaluateSuffixExpression(suffix);

    printf("Result: %d\n", result);

    return 0;

}

b. Program for Tower of Hanoi Problem:

#include <stdio.h>

// Function to solve Tower of Hanoi problem

void towerOfHanoi(int n, char from\_rod, char to\_rod, char aux\_rod) {

    if (n == 1) {

        printf("Move disk 1 from rod %c to rod %c\n", from\_rod, to\_rod);

        return;

    }

    towerOfHanoi(n - 1, from\_rod, aux\_rod, to\_rod);

    printf("Move disk %d from rod %c to rod %c\n", n, from\_rod, to\_rod);

    towerOfHanoi(n - 1, aux\_rod, to\_rod, from\_rod);

}

int main() {

    int num\_disks;

    printf("Enter number of disks: ");

    scanf("%d", &num\_disks);

    printf("Steps to solve Tower of Hanoi with %d disks:\n", num\_disks);

    towerOfHanoi(num\_disks, 'A', 'C', 'B');

    return 0;

}

Develop a menu driven Program in C for the following operations on Circular QUEUE of

Characters (Array Implementation of Queue with maximum size MAX)

a. Insert an Element on to Circular QUEUE

b. Delete an Element from Circular QUEUE

c. Demonstrate Overflow and Underflow situations on Circular QUEUE

d. Display the status of Circular QUEUE

e. Exit

Support the program with appropriate functions for each of the above operations

#include <stdio.h>

#include <stdbool.h>

#define MAX 5

// Circular Queue structure

struct CircularQueue {

    char items[MAX];

    int front, rear;

};

// Function to initialize the Circular Queue

void initialize(struct CircularQueue \*q) {

    q->front = -1;

    q->rear = -1;

}

// Function to check if the Circular Queue is full

bool isFull(struct CircularQueue \*q) {

    return (q->front == 0 && q->rear == MAX - 1) || (q->rear == (q->front - 1) % (MAX - 1));

}

// Function to check if the Circular Queue is empty

bool isEmpty(struct CircularQueue \*q) {

    return q->front == -1;

}

// Function to insert an element into the Circular Queue

void insert(struct CircularQueue \*q, char value) {

    if (isFull(q)) {

        printf("Queue Overflow - Cannot insert element %c\n", value);

    } else {

        if (q->front == -1) {

            q->front = 0;

        }

        q->rear = (q->rear + 1) % MAX;

        q->items[q->rear] = value;

        printf("Element %c inserted into the queue\n", value);

    }

}

// Function to delete an element from the Circular Queue

char delete(struct CircularQueue \*q) {

    char removed;

    if (isEmpty(q)) {

        printf("Queue Underflow - Cannot delete element\n");

        return '\0';

    } else {

        removed = q->items[q->front];

        if (q->front == q->rear) {

            q->front = -1;

            q->rear = -1;

        } else {

            q->front = (q->front + 1) % MAX;

        }

        return removed;

    }

}

// Function to display the status of the Circular Queue

void display(struct CircularQueue \*q) {

    if (isEmpty(q)) {

        printf("Queue is empty\n");

    } else {

        int i = q->front;

        printf("Circular Queue elements: ");

        do {

            printf("%c ", q->items[i]);

            i = (i + 1) % MAX;

        } while (i != (q->rear + 1) % MAX);

        printf("\n");

    }

}

int main() {

    struct CircularQueue queue;

    initialize(&queue);

    int choice;

    char value;

    do {

        printf("\nMENU\n");

        printf("1. Insert element\n");

        printf("2. Delete element\n");

        printf("3. Display queue status\n");

        printf("4. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter element to insert: ");

                scanf(" %c", &value);

                insert(&queue, value);

                break;

            case 2:

                value = delete(&queue);

                if (value != '\0') {

                    printf("Deleted element: %c\n", value);

                }

                break;

            case 3:

                display(&queue);

                break;

            case 4:

                printf("Exiting the program\n");

                break;

            default:

                printf("Invalid choice\n");

        }

    } while (choice != 4);

    return 0;

}

Develop a menu driven Program in C for the following operations on Singly Linked List

(SLL) of Student Data with the fields: USN, Name, Programme, Sem,

PhNo

a. Create a SLL of N Students Data by using front insertion.

b. Display the status of SLL and count the number of nodes in it

c. Perform Insertion / Deletion at End of SLL

d. Perform Insertion / Deletion at Front of SLL(Demonstration of stack)

e. Exit

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Structure for Student Data

struct Student {

    char USN[20];

    char Name[50];

    char Programme[50];

    int Sem;

    long long PhNo;

    struct Student \*next;

};

// Function to create a new Student node

struct Student \*createStudent(char USN[], char Name[], char Programme[], int Sem, long long PhNo) {

    struct Student \*newStudent = (struct Student \*)malloc(sizeof(struct Student));

    if (newStudent == NULL) {

        printf("Memory allocation failed\n");

        return NULL;

    }

    strcpy(newStudent->USN, USN);

    strcpy(newStudent->Name, Name);

    strcpy(newStudent->Programme, Programme);

    newStudent->Sem = Sem;

    newStudent->PhNo = PhNo;

    newStudent->next = NULL;

    return newStudent;

}

// Function to insert a Student node at the front of the list

void insertAtFront(struct Student \*\*head, struct Student \*newNode) {

    if (\*head == NULL) {

        \*head = newNode;

    } else {

        newNode->next = \*head;

        \*head = newNode;

    }

    printf("Student data inserted at the front\n");

}

// Function to display the status of the SLL and count the number of nodes

void displayAndCount(struct Student \*head) {

    if (head == NULL) {

        printf("List is empty\n");

        return;

    }

    struct Student \*temp = head;

    int count = 0;

    printf("SLL of Students Data:\n");

    while (temp != NULL) {

        printf("USN: %s, Name: %s, Programme: %s, Sem: %d, PhNo: %lld\n", temp->USN, temp->Name, temp->Programme, temp->Sem, temp->PhNo);

        temp = temp->next;

        count++;

    }

    printf("Number of nodes in the list: %d\n", count);

}

// Function to insert a Student node at the end of the list

void insertAtEnd(struct Student \*\*head, struct Student \*newNode) {

    if (\*head == NULL) {

        \*head = newNode;

    } else {

        struct Student \*temp = \*head;

        while (temp->next != NULL) {

            temp = temp->next;

        }

        temp->next = newNode;

    }

    printf("Student data inserted at the end\n");

}

// Function to delete a Student node from the front of the list

void deleteAtFront(struct Student \*\*head) {

    if (\*head == NULL) {

        printf("List is empty - Cannot delete\n");

    } else {

        struct Student \*temp = \*head;

        \*head = (\*head)->next;

        free(temp);

        printf("Student data deleted from the front\n");

    }

}

// Function to delete a Student node from the end of the list

void deleteAtEnd(struct Student \*\*head) {

    if (\*head == NULL) {

        printf("List is empty - Cannot delete\n");

    } else if ((\*head)->next == NULL) {

        free(\*head);

        \*head = NULL;

        printf("Student data deleted from the end\n");

    } else {

        struct Student \*temp = \*head;

        while (temp->next->next != NULL) {

            temp = temp->next;

        }

        free(temp->next);

        temp->next = NULL;

        printf("Student data deleted from the end\n");

    }

}

// Function to free memory allocated to the linked list

void freeLinkedList(struct Student \*\*head) {

    struct Student \*temp;

    while (\*head != NULL) {

        temp = \*head;

        \*head = (\*head)->next;

        free(temp);

    }

}

int main() {

    struct Student \*head = NULL;

    int choice, Sem;

    char USN[20], Name[50], Programme[50];

    long long PhNo;

    do {

        printf("\nMENU\n");

        printf("1. Create a Student (Front Insertion)\n");

        printf("2. Display status and count of Students\n");

        printf("3. Insert Student at End\n");

        printf("4. Delete Student at End\n");

        printf("5. Insert Student at Front (Demonstration of stack)\n");

        printf("6. Delete Student at Front (Demonstration of stack)\n");

        printf("7. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Enter USN, Name, Programme, Sem, PhNo: ");

                scanf("%s %s %s %d %lld", USN, Name, Programme, &Sem, &PhNo);

                insertAtFront(&head, createStudent(USN, Name, Programme, Sem, PhNo));

                break;

            case 2:

                displayAndCount(head);

                break;

            case 3:

                printf("Enter USN, Name, Programme, Sem, PhNo: ");

                scanf("%s %s %s %d %lld", USN, Name, Programme, &Sem, &PhNo);

                insertAtEnd(&head, createStudent(USN, Name, Programme, Sem, PhNo));

                break;

            case 4:

                deleteAtEnd(&head);

                break;

            case 5:

                printf("Enter USN, Name, Programme, Sem, PhNo: ");

                scanf("%s %s %s %d %lld", USN, Name, Programme, &Sem, &PhNo);

                insertAtFront(&head, createStudent(USN, Name, Programme, Sem, PhNo));

                break;

            case 6:

                deleteAtFront(&head);

                break;

            case 7:

                printf("Exiting the program\n");

                break;

            default:

                printf("Invalid choice\n");

        }

    } while (choice != 7);

    // Free memory allocated to the linked list before exiting

    freeLinkedList(&head);

    return 0;

}

Develop a Program in C for the following operationson Singly Circular Linked List (SCLL)

with header nodes

a. Represent and Evaluate a Polynomial P(x,y,z) = 6x2y2z-4yz5+3x3yz+2xy5z-2xyz3

b. Find the sum of two polynomials POLY1(x,y,z) and POLY2(x,y,z) and store the

result in POLYSUM(x,y,z)

Support the program with appropriate functions for each of the above operations

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

struct Term {

    int coeff;

    int x\_exp;

    int y\_exp;

    int z\_exp;

    struct Term \*next;

};

// Function to create a new term in the polynomial

struct Term \*createTerm(int coeff, int x\_exp, int y\_exp, int z\_exp) {

    struct Term \*newTerm = (struct Term \*)malloc(sizeof(struct Term));

    if (newTerm == NULL) {

        printf("Memory allocation failed\n");

        return NULL;

    }

    newTerm->coeff = coeff;

    newTerm->x\_exp = x\_exp;

    newTerm->y\_exp = y\_exp;

    newTerm->z\_exp = z\_exp;

    newTerm->next = NULL;

    return newTerm;

}

// Function to insert a term at the end of the circular linked list

void insertTerm(struct Term \*\*head, struct Term \*newTerm) {

    if (\*head == NULL) {

        \*head = newTerm;

        newTerm->next = \*head;

    } else {

        struct Term \*temp = \*head;

        while (temp->next != \*head) {

            temp = temp->next;

        }

        temp->next = newTerm;

        newTerm->next = \*head;

    }

}

// Function to display the polynomial

void displayPolynomial(struct Term \*head) {

    if (head == NULL) {

        printf("Polynomial is empty\n");

        return;

    }

    struct Term \*temp = head;

    printf("Polynomial: ");

    do {

        printf("%dx^%dy^%dz^%d", temp->coeff, temp->x\_exp, temp->y\_exp, temp->z\_exp);

        temp = temp->next;

        if (temp != head) {

            printf(" + ");

        }

    } while (temp != head);

    printf("\n");

}

// Function to evaluate the polynomial for given x, y, z values

int evaluatePolynomial(struct Term \*head, int x, int y, int z) {

    if (head == NULL) {

        printf("Polynomial is empty\n");

        return 0;

    }

    int result = 0;

    struct Term \*temp = head;

    do {

        result += temp->coeff \* pow(x, temp->x\_exp) \* pow(y, temp->y\_exp) \* pow(z, temp->z\_exp);

        temp = temp->next;

    } while (temp != head);

    return result;

}

// Function to free memory allocated to the linked list

void freeLinkedList(struct Term \*\*head) {

    if (\*head == NULL) {

        return;

    }

    struct Term \*temp = \*head;

    struct Term \*next;

    do {

        next = temp->next;

        free(temp);

        temp = next;

    } while (temp != \*head);

    \*head = NULL;

}

int main() {

    struct Term \*poly1 = NULL; // Polynomial 1

    struct Term \*poly2 = NULL; // Polynomial 2

    // Representing the first polynomial: 6x^2y^2z - 4yz^5 + 3x^3yz + 2xy^5z - 2xyz^3

    insertTerm(&poly1, createTerm(6, 2, 2, 1));

    insertTerm(&poly1, createTerm(-4, 0, 1, 5));

    insertTerm(&poly1, createTerm(3, 3, 1, 1));

    insertTerm(&poly1, createTerm(2, 1, 5, 1));

    insertTerm(&poly1, createTerm(-2, 1, 1, 3));

    printf("First Polynomial:\n");

    displayPolynomial(poly1);

    // Evaluation of the first polynomial for x=2, y=3, z=4

    int x = 2, y = 3, z = 4;

    int result = evaluatePolynomial(poly1, x, y, z);

    printf("Result of polynomial evaluation for x=%d, y=%d, z=%d: %d\n", x, y, z, result);

    // Free memory allocated to the first polynomial

    freeLinkedList(&poly1);

    return 0;

}

Develop a menu driven Program in C for the following operations on Binary Search Tree

(BST) of Integers .

a. Create a BST of N Integers: 6, 9, 5, 2, 8, 15, 24, 14, 7, 8, 5, 2

b. Traverse the BST in Inorder, Preorder and Post Order

c. Search the BST for a given element (KEY) and report the appropriate message

d. Exit

#include <stdio.h>

#include <stdlib.h>

// Structure for the Binary Search Tree node

struct Node {

    int data;

    struct Node \*left;

    struct Node \*right;

};

// Function to create a new node for BST

struct Node \*createNode(int key) {

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

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        return NULL;

    }

    newNode->data = key;

    newNode->left = newNode->right = NULL;

    return newNode;

}

// Function to insert a key into BST

struct Node \*insert(struct Node \*root, int key) {

    if (root == NULL) {

        return createNode(key);

    }

    if (key < root->data) {

        root->left = insert(root->left, key);

    } else if (key > root->data) {

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

    }

    return root;

}

// Function to perform inorder traversal of BST

void inorderTraversal(struct Node \*root) {

    if (root != NULL) {

        inorderTraversal(root->left);

        printf("%d ", root->data);

        inorderTraversal(root->right);

    }

}

// Function to perform preorder traversal of BST

void preorderTraversal(struct Node \*root) {

    if (root != NULL) {

        printf("%d ", root->data);

        preorderTraversal(root->left);

        preorderTraversal(root->right);

    }

}

// Function to perform postorder traversal of BST

void postorderTraversal(struct Node \*root) {

    if (root != NULL) {

        postorderTraversal(root->left);

        postorderTraversal(root->right);

        printf("%d ", root->data);

    }

}

// Function to search for a key in BST

void search(struct Node \*root, int key) {

    if (root == NULL) {

        printf("Key not found\n");

        return;

    }

    if (root->data == key) {

        printf("Key found in BST\n");

    } else if (key < root->data) {

        search(root->left, key);

    } else {

        search(root->right, key);

    }

}

int main() {

    struct Node \*root = NULL;

    int choice, key;

    // Create BST with given elements

    int keys[] = {6, 9, 5, 2, 8, 15, 24, 14, 7, 8, 5, 2};

    int numKeys = sizeof(keys) / sizeof(keys[0]);

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

        root = insert(root, keys[i]);

    }

    do {

        printf("\nMENU\n");

        printf("1. Traverse BST in Inorder\n");

        printf("2. Traverse BST in Preorder\n");

        printf("3. Traverse BST in Postorder\n");

        printf("4. Search element in BST\n");

        printf("5. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Inorder traversal: ");

                inorderTraversal(root);

                printf("\n");

                break;

            case 2:

                printf("Preorder traversal: ");

                preorderTraversal(root);

                printf("\n");

                break;

            case 3:

                printf("Postorder traversal: ");

                postorderTraversal(root);

                printf("\n");

                break;

            case 4:

                printf("Enter element to search: ");

                scanf("%d", &key);

                search(root, key);

                break;

            case 5:

                printf("Exiting the program\n");

                break;

            default:

                printf("Invalid choice\n");

        }

    } while (choice != 5);

    return 0;

}

Develop a Program in C for the following operations on Graph(G) of Cities

a. Create a Graph of N cities using Adjacency Matrix.

b. Print all the nodes reachable from a given starting node in a digraph using DFS/BFS

method

#include <stdio.h>

#include <stdbool.h>

#define MAX 20

// Function to initialize the adjacency matrix with zeros

void initializeGraph(int adjMatrix[MAX][MAX], int n) {

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            adjMatrix[i][j] = 0;

        }

    }

}

// Function to add an edge between two cities in the graph

void addEdge(int adjMatrix[MAX][MAX], int src, int dest) {

    adjMatrix[src][dest] = 1;

}

// Function to perform Depth-First Search (DFS) in the graph

void DFS(int adjMatrix[MAX][MAX], int n, int start, bool visited[]) {

    printf("%d ", start);

    visited[start] = true;

    for (int i = 0; i < n; i++) {

        if (adjMatrix[start][i] && !visited[i]) {

            DFS(adjMatrix, n, i, visited);

        }

    }

}

// Function to perform Breadth-First Search (BFS) in the graph

void BFS(int adjMatrix[MAX][MAX], int n, int start, bool visited[]) {

    int queue[MAX];

    int front = -1, rear = -1;

    printf("%d ", start);

    visited[start] = true;

    queue[++rear] = start;

    while (front != rear) {

        int current = queue[++front];

        for (int i = 0; i < n; i++) {

            if (adjMatrix[current][i] && !visited[i]) {

                printf("%d ", i);

                visited[i] = true;

                queue[++rear] = i;

            }

        }

    }

}

int main() {

    int adjMatrix[MAX][MAX];

    int numCities, srcCity;

    printf("Enter the number of cities: ");

    scanf("%d", &numCities);

    initializeGraph(adjMatrix, numCities);

    // Add edges between cities (Example: considering a few connections)

    addEdge(adjMatrix, 0, 1); // Example edge: City 0 to City 1

    addEdge(adjMatrix, 0, 2); // Example edge: City 0 to City 2

    addEdge(adjMatrix, 1, 3); // Example edge: City 1 to City 3

    addEdge(adjMatrix, 2, 4); // Example edge: City 2 to City 4

    addEdge(adjMatrix, 3, 4); // Example edge: City 3 to City 4

    printf("Enter the starting city (0 to %d): ", numCities - 1);

    scanf("%d", &srcCity);

    // Performing DFS

    printf("DFS traversal starting from city %d: ", srcCity);

    bool visitedDFS[MAX] = {false};

    DFS(adjMatrix, numCities, srcCity, visitedDFS);

    printf("\n");

    // Performing BFS

    printf("BFS traversal starting from city %d: ", srcCity);

    bool visitedBFS[MAX] = {false};

    BFS(adjMatrix, numCities, srcCity, visitedBFS);

    printf("\n");

    return 0;

}

Given a File of N employee records with a set K of Keys (4-digit) which uniquely determine

the records in file F. Assume that file F is maintained in memory by a Hash Table (HT) of m

memory locations with L as the set of memory addresses (2-digit) of locations in HT. Let the

keys in K and addresses in L are Integers. Develop a Program in C that uses Hash function H:

K →L as H(K)=K mod m (remainder method), and implement hashing

technique to map a given key K to the address space L. Resolve the collision (if any) using

linear probing.

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

// Structure for employee records

struct EmployeeRecord {

    int key;

    // Other employee details can be added here

};

// Structure for Hash Table Entry

struct HashTableEntry {

    int key;

    int address;

};

// Function to initialize hash table entries

void initializeHashTable(struct HashTableEntry HT[], int m) {

    for (int i = 0; i < m; i++) {

        HT[i].key = -1;

        HT[i].address = -1;

    }

}

// Function to perform hashing using remainder method (mod m)

int hashFunction(int key, int m) {

    return key % m;

}

// Function to insert a key into the hash table using linear probing for collision resolution

void insert(struct HashTableEntry HT[], int m, int key, int address) {

    int hashIndex = hashFunction(key, m);

    while (HT[hashIndex].key != -1) {

        hashIndex = (hashIndex + 1) % m;

    }

    HT[hashIndex].key = key;

    HT[hashIndex].address = address;

}

// Function to display the hash table

void displayHashTable(struct HashTableEntry HT[], int m) {

    printf("\nHash Table:\n");

    printf("Index\tKey\tAddress\n");

    for (int i = 0; i < m; i++) {

        printf("%d\t%d\t%d\n", i, HT[i].key, HT[i].address);

    }

}

int main() {

    struct EmployeeRecord employeeRecords[MAX];

    struct HashTableEntry hashTable[MAX];

    int n, m;

    printf("Enter the number of employee records: ");

    scanf("%d", &n);

    printf("Enter the number of memory locations in the hash table: ");

    scanf("%d", &m);

    // Input employee records with unique keys

    printf("Enter %d employee keys:\n", n);

    for (int i = 0; i < n; i++) {

        scanf("%d", &employeeRecords[i].key);

    }

    // Initialize hash table entries

    initializeHashTable(hashTable, m);

    // Insert keys into the hash table using hashing and linear probing

    for (int i = 0; i < n; i++) {

        int address = hashFunction(employeeRecords[i].key, m);

        insert(hashTable, m, employeeRecords[i].key, address);

    }

    // Display the hash table

    displayHashTable(hashTable, m);

    return 0;

}