**WAP to convert a given Infix expression into its equivalent Postfix expression and evaluate it using stack.**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_SIZE 100

struct Stack {

    int top;

    int capacity;

    int\* array;

};

struct Stack\* createStack(int capacity) {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->capacity = capacity;

    stack->top = -1;

    stack->array = (int\*)malloc(stack->capacity \* sizeof(int));

    return stack;

}

int isEmpty(struct Stack\* stack) {

    return stack->top == -1;

}

int isFull(struct Stack\* stack) {

    return stack->top == stack->capacity - 1;

}

void push(struct Stack\* stack, int item) {

    if (isFull(stack)) return;

    stack->array[++stack->top] = item;

}

int pop(struct Stack\* stack) {

    if (isEmpty(stack)) return -1;

    return stack->array[stack->top--];

}

int peek(struct Stack\* stack) {

    if (isEmpty(stack)) return -1;

    return stack->array[stack->top];

}

int precedence(char op) {

    switch(op) {

        case '+':

        case '-':

            return 1;

        case '\*':

        case '/':

            return 2;

        case '^':

            return 3;

        default:

            return -1;

    }

}

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

    struct Stack\* stack = createStack(MAX\_SIZE);

    int i, k;

    for (i = 0, k = -1; infix[i]; ++i) {

        if (isalnum(infix[i])) {

            postfix[++k] = infix[i];

        } else if (infix[i] == '(') {

            push(stack, infix[i]);

        } else if (infix[i] == ')') {

            while (!isEmpty(stack) && peek(stack) != '(') {

                postfix[++k] = pop(stack);

            }

            pop(stack);

        } else {

            while (!isEmpty(stack) && precedence(infix[i]) <= precedence(peek(stack))) {

                postfix[++k] = pop(stack);

            }

            push(stack, infix[i]);

        }

    }

    while (!isEmpty(stack)) {

        postfix[++k] = pop(stack);

    }

    postfix[++k] = '\0';

}

int evaluatePostfix(char\* postfix) {

    struct Stack\* stack = createStack(MAX\_SIZE);

    int i, operand1, operand2;

    for (i = 0; postfix[i]; ++i) {

        if (isdigit(postfix[i])) {

            push(stack, postfix[i] - '0');

        } else {

            operand2 = pop(stack);

            operand1 = pop(stack);

            switch(postfix[i]) {

                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;

            }

        }

    }

    return pop(stack);

}

int main() {

    char infix[MAX\_SIZE], postfix[MAX\_SIZE];

    printf("Enter an infix expression: ");

    scanf("%s", infix);

    infixToPostfix(infix, postfix);

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

    printf("Result: %d\n", evaluatePostfix(postfix));

    return 0;

}

**WAP to convert a given Infix expression into its equivalent Prefix expression and evaluate it using stack.**



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAX\_SIZE 100

struct Stack {

    int top;

    int capacity;

    char\* array;

};

struct Stack\* createStack(int capacity) {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->capacity = capacity;

    stack->top = -1;

    stack->array = (char\*)malloc(stack->capacity \* sizeof(char));

    return stack;

}

int isEmpty(struct Stack\* stack) {

    return stack->top == -1;

}

int isFull(struct Stack\* stack) {

    return stack->top == stack->capacity - 1;

}

void push(struct Stack\* stack, char item) {

    if (isFull(stack)) return;

    stack->array[++stack->top] = item;

}

char pop(struct Stack\* stack) {

    if (isEmpty(stack)) return -1;

    return stack->array[stack->top--];

}

char peek(struct Stack\* stack) {

    if (isEmpty(stack)) return -1;

    return stack->array[stack->top];

}

int precedence(char op) {

    switch(op) {

        case '+':

        case '-':

            return 1;

        case '\*':

        case '/':

            return 2;

        case '^':

            return 3;

        default:

            return -1;

    }

}

void infixToPrefix(char\* infix, char\* prefix) {

    int len = strlen(infix);

    // Reverse the infix expression

    for (int i = 0; i < len / 2; ++i) {

        char temp = infix[i];

        infix[i] = infix[len - i - 1];

        infix[len - i - 1] = temp;

    }

    // Swap '(' and ')' and reverse the precedence of operators

    for (int i = 0; i < len; ++i) {

        if (infix[i] == '(')

            infix[i] = ')';

        else if (infix[i] == ')')

            infix[i] = '(';

    }

    struct Stack\* stack = createStack(MAX\_SIZE);

    int i, k;

    for (i = 0, k = -1; infix[i]; ++i) {

        if (isalnum(infix[i])) {

            prefix[++k] = infix[i];

        } else if (infix[i] == '(') {

            push(stack, infix[i]);

        } else if (infix[i] == ')') {

            while (!isEmpty(stack) && peek(stack) != '(') {

                prefix[++k] = pop(stack);

            }

            pop(stack);

        } else {

            while (!isEmpty(stack) && precedence(infix[i]) < precedence(peek(stack))) {

                prefix[++k] = pop(stack);

            }

            push(stack, infix[i]);

        }

    }

    while (!isEmpty(stack)) {

        prefix[++k] = pop(stack);

    }

    prefix[++k] = '\0';

    // Reverse the prefix expression to get the final result

    len = strlen(prefix);

    for (int i = 0; i < len / 2; ++i) {

        char temp = prefix[i];

        prefix[i] = prefix[len - i - 1];

        prefix[len - i - 1] = temp;

    }

}

int evaluatePrefix(char\* prefix) {

    struct Stack\* stack = createStack(MAX\_SIZE);

    int i, operand1, operand2;

    for (i = 0; prefix[i]; ++i) {

        if (isdigit(prefix[i])) {

            push(stack, prefix[i] - '0');

        } else {

            operand1 = pop(stack);

            operand2 = pop(stack);

            switch(prefix[i]) {

                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;

            }

        }

    }

    return pop(stack);

}

int main() {

    char infix[MAX\_SIZE], prefix[MAX\_SIZE];

    printf("Enter an infix expression: ");

    scanf("%s", infix);

    infixToPrefix(infix, prefix);

    printf("Prefix expression: %s\n", prefix);

    printf("Result: %d\n", evaluatePrefix(prefix));

    return 0;

}

**WAP to implement two stack using array and perform following operations on it. A. PUSH, B. POP, C. StackFull D. StackeEmpty E. Display Stack.**



#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct TwoStacks {

    int top1; // Top index of the first stack

    int top2; // Top index of the second stack

    int array[MAX\_SIZE]; // Array to store elements of both stacks

};

struct TwoStacks\* createTwoStacks() {

    struct TwoStacks\* stacks = (struct TwoStacks\*)malloc(sizeof(struct TwoStacks));

    stacks->top1 = -1; // Initialize top index of the first stack

    stacks->top2 = MAX\_SIZE; // Initialize top index of the second stack

    return stacks;

}

int isStack1Full(struct TwoStacks\* stacks) {

    return stacks->top1 == stacks->top2 - 1;

}

int isStack2Full(struct TwoStacks\* stacks) {

    return stacks->top2 == stacks->top1 + 1;

}

int isStack1Empty(struct TwoStacks\* stacks) {

    return stacks->top1 == -1;

}

int isStack2Empty(struct TwoStacks\* stacks) {

    return stacks->top2 == MAX\_SIZE;

}

void push1(struct TwoStacks\* stacks, int data) {

    if (isStack1Full(stacks)) {

        printf("Stack 1 overflow\n");

        return;

    }

    stacks->array[++stacks->top1] = data;

}

void push2(struct TwoStacks\* stacks, int data) {

    if (isStack2Full(stacks)) {

        printf("Stack 2 overflow\n");

        return;

    }

    stacks->array[--stacks->top2] = data;

}

int pop1(struct TwoStacks\* stacks) {

    if (isStack1Empty(stacks)) {

        printf("Stack 1 underflow\n");

        return -1;

    }

    return stacks->array[stacks->top1--];

}

int pop2(struct TwoStacks\* stacks) {

    if (isStack2Empty(stacks)) {

        printf("Stack 2 underflow\n");

        return -1;

    }

    return stacks->array[stacks->top2++];

}

void displayStack1(struct TwoStacks\* stacks) {

    if (isStack1Empty(stacks)) {

        printf("Stack 1 is empty\n");

        return;

    }

    printf("Stack 1: ");

    for (int i = stacks->top1; i >= 0; i--) {

        printf("%d ", stacks->array[i]);

    }

    printf("\n");

}

void displayStack2(struct TwoStacks\* stacks) {

    if (isStack2Empty(stacks)) {

        printf("Stack 2 is empty\n");

        return;

    }

    printf("Stack 2: ");

    for (int i = stacks->top2; i < MAX\_SIZE; i++) {

        printf("%d ", stacks->array[i]);

    }

    printf("\n");

}

void displayBothStacks(struct TwoStacks\* stacks) {

    displayStack1(stacks);

    displayStack2(stacks);

}

int main() {

    struct TwoStacks\* stacks = createTwoStacks();

    push1(stacks, 10);

    push1(stacks, 20);

    push1(stacks, 30);

    push2(stacks, 40);

    push2(stacks, 50);

    push2(stacks, 60);

    displayBothStacks(stacks);

    pop1(stacks);

    pop2(stacks);

    displayBothStacks(stacks);

    return 0;

}

**WAP to implement following by using stack.**



**A. Factorial of a given number B. Generation of Fibonacci series**

A.Factorial

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Stack {

    int top;

    int capacity;

    int\* array;

};

struct Stack\* createStack(int capacity) {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->capacity = capacity;

    stack->top = -1;

    stack->array = (int\*)malloc(stack->capacity \* sizeof(int));

    return stack;

}

int isEmpty(struct Stack\* stack) {

    return stack->top == -1;

}

int isFull(struct Stack\* stack) {

    return stack->top == stack->capacity - 1;

}

void push(struct Stack\* stack, int data) {

    if (isFull(stack)) {

        printf("Stack overflow\n");

        return;

    }

    stack->array[++stack->top] = data;

}

int pop(struct Stack\* stack) {

    if (isEmpty(stack)) {

        printf("Stack underflow\n");

        return -1;

    }

    return stack->array[stack->top--];

}

int factorial(int n) {

    struct Stack\* stack = createStack(MAX\_SIZE);

    int fact = 1;

    for (int i = 1; i <= n; i++) {

        push(stack, i);

    }

    while (!isEmpty(stack)) {

        fact \*= pop(stack);

    }

    return fact;

}

int main() {

    int n;

    printf("Enter a number: ");

    scanf("%d", &n);

    printf("Factorial of %d is %d\n", n, factorial(n));

    return 0;

}

B.Fibonacci

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

struct Stack {

    int top;

    int capacity;

    int\* array;

};

struct Stack\* createStack(int capacity) {

    struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

    stack->capacity = capacity;

    stack->top = -1;

    stack->array = (int\*)malloc(stack->capacity \* sizeof(int));

    return stack;

}

int isEmpty(struct Stack\* stack) {

    return stack->top == -1;

}

int isFull(struct Stack\* stack) {

    return stack->top == stack->capacity - 1;

}

void push(struct Stack\* stack, int data) {

    if (isFull(stack)) {

        printf("Stack overflow\n");

        return;

    }

    stack->array[++stack->top] = data;

}

int pop(struct Stack\* stack) {

    if (isEmpty(stack)) {

        printf("Stack underflow\n");

        return -1;

    }

    return stack->array[stack->top--];

}

void generateFibonacci(int n) {

    struct Stack\* stack = createStack(MAX\_SIZE);

    int a = 0, b = 1, c;

    printf("Fibonacci series up to %d terms: ", n);

    for (int i = 1; i <= n; i++) {

        printf("%d ", a);

        push(stack, a);

        c = a + b;

        a = b;

        b = c;

    }

    printf("\n");

    printf("Fibonacci series retrieved from stack: ");

    while (!isEmpty(stack)) {

        printf("%d ", pop(stack));

    }

    printf("\n");

}

int main() {

    int n;

    printf("Enter the number of terms for Fibonacci series: ");

    scanf("%d", &n);

    generateFibonacci(n);

    return 0;

}

* **Write a Program to implement circular double ended queue where user can add and remove the elements from both front and rear of the queue**

#include <stdio.h>

#include <stdlib.h>

#define MAX 5

typedef struct {

    int arr[MAX];

    int front;

    int rear;

} Deque;

void initialize(Deque \*dq) {

    dq->front = -1;

    dq->rear = -1;

}

int isFull(Deque \*dq) {

    return ((dq->front == 0 && dq->rear == MAX - 1) || dq->front == dq->rear + 1);

}

int isEmpty(Deque \*dq) {

    return (dq->front == -1);

}

void insertFront(Deque \*dq, int item) {

    if (isFull(dq)) {

        printf("Queue Overflow\n");

        return;

    }

    if (dq->front == -1) {

        dq->front = 0;

        dq->rear = 0;

    } else if (dq->front == 0) {

        dq->front = MAX - 1;

    } else {

        dq->front = dq->front - 1;

    }

    dq->arr[dq->front] = item;

}

void insertRear(Deque \*dq, int item) {

    if (isFull(dq)) {

        printf("Queue Overflow\n");

        return;

    }

    if (dq->front == -1) {

        dq->front = 0;

        dq->rear = 0;

    } else if (dq->rear == MAX - 1) {

        dq->rear = 0;

    } else {

        dq->rear = dq->rear + 1;

    }

    dq->arr[dq->rear] = item;

}

void deleteFront(Deque \*dq) {

    if (isEmpty(dq)) {

        printf("Queue Underflow\n");

        return;

    }

    printf("Element deleted from front is %d\n", dq->arr[dq->front]);

    if (dq->front == dq->rear) {

        dq->front = -1;

        dq->rear = -1;

    } else if (dq->front == MAX - 1) {

        dq->front = 0;

    } else {

        dq->front = dq->front + 1;

    }

}

void deleteRear(Deque \*dq) {

    if (isEmpty(dq)) {

        printf("Queue Underflow\n");

        return;

    }

    printf("Element deleted from rear is %d\n", dq->arr[dq->rear]);

    if (dq->front == dq->rear) {

        dq->front = -1;

        dq->rear = -1;

    } else if (dq->rear == 0) {

        dq->rear = MAX - 1;

    } else {

        dq->rear = dq->rear - 1;

    }

}

void display(Deque \*dq) {

    int i;

    if (isEmpty(dq)) {

        printf("Queue is empty\n");

        return;

    }

    printf("Queue elements are:\n");

    if (dq->front <= dq->rear) {

        for (i = dq->front; i <= dq->rear; i++)

            printf("%d ", dq->arr[i]);

    } else {

        for (i = dq->front; i < MAX; i++)

            printf("%d ", dq->arr[i]);

        for (i = 0; i <= dq->rear; i++)

            printf("%d ", dq->arr[i]);

    }

    printf("\n");

}

int main() {

    Deque dq;

    initialize(&dq);

    int choice, item;

    while (1) {

        printf("\n1. Insert at front\n");

        printf("2. Insert at rear\n");

        printf("3. Delete from front\n");

        printf("4. Delete from rear\n");

        printf("5. Display\n");

        printf("6. Quit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("Input the element for insertion in front: ");

                scanf("%d", &item);

                insertFront(&dq, item);

                break;

            case 2:

                printf("Input the element for insertion in rear: ");

                scanf("%d", &item);

                insertRear(&dq, item);

                break;

            case 3:

                deleteFront(&dq);

                break;

            case 4:

                deleteRear(&dq);

                break;

            case 5:

                display(&dq);

                break;

            case 6:

                exit(0);

            default:

                printf("Wrong choice\n");

        }

    }

    return 0;

}

* **Write a Program to implement multiple two queues using array and perform following operations on it. A. Addq, B. Delq, C. Display Queue.**

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

typedef struct {

int arr[MAX];

int front1, rear1;

int front2, rear2;

} TwoQueues;

void initialize(TwoQueues \*queues) {

queues->front1 = -1;

queues->rear1 = -1;

queues->front2 = MAX;

queues->rear2 = MAX;

}

int isFull1(TwoQueues \*queues) {

return (queues->rear1 + 1 == queues->front2);

}

int isFull2(TwoQueues \*queues) {

return (queues->rear2 - 1 == queues->rear1);

}

int isEmpty1(TwoQueues \*queues) {

return (queues->front1 == -1);

}

int isEmpty2(TwoQueues \*queues) {

return (queues->front2 == MAX);

}

void Addq1(TwoQueues \*queues, int item) {

if (isFull1(queues)) {

printf("Queue 1 Overflow\n");

return;

}

if (queues->front1 == -1) {

queues->front1 = 0;

}

queues->rear1++;

queues->arr[queues->rear1] = item;

}

void Addq2(TwoQueues \*queues, int item) {

if (isFull2(queues)) {

printf("Queue 2 Overflow\n");

return;

}

if (queues->front2 == MAX) {

queues->front2 = MAX - 1;

}

queues->rear2--;

queues->arr[queues->rear2] = item;

}

void Delq1(TwoQueues \*queues) {

if (isEmpty1(queues)) {

printf("Queue 1 Underflow\n");

return;

}

printf("Element deleted from Queue 1 is %d\n", queues->arr[queues->front1]);

if (queues->front1 == queues->rear1) {

queues->front1 = -1;

queues->rear1 = -1;

} else {

queues->front1++;

}

}

void Delq2(TwoQueues \*queues) {

if (isEmpty2(queues)) {

printf("Queue 2 Underflow\n");

return;

}

printf("Element deleted from Queue 2 is %d\n", queues->arr[queues->front2]);

if (queues->front2 == queues->rear2) {

queues->front2 = MAX;

queues->rear2 = MAX;

} else {

queues->front2--;

}

}

void displayQueue1(TwoQueues \*queues) {

if (isEmpty1(queues)) {

printf("Queue 1 is empty\n");

return;

}

printf("Queue 1 elements are:\n");

for (int i = queues->front1; i <= queues->rear1; i++) {

printf("%d ", queues->arr[i]);

}

printf("\n");

}

void displayQueue2(TwoQueues \*queues) {

if (isEmpty2(queues)) {

printf("Queue 2 is empty\n");

return;

}

printf("Queue 2 elements are:\n");

for (int i = queues->front2; i >= queues->rear2; i--) {

printf("%d ", queues->arr[i]);

}

printf("\n");

}

int main() {

TwoQueues queues;

initialize(&queues);

int choice, item, queueNumber;

while (1) {

printf("\n1. Addq\n");

printf("2. Delq\n");

printf("3. Display Queue\n");

printf("4. Quit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter queue number (1 or 2): ");

scanf("%d", &queueNumber);

printf("Input the element to add: ");

scanf("%d", &item);

if (queueNumber == 1) {

Addq1(&queues, item);

} else if (queueNumber == 2) {

Addq2(&queues, item);

} else {

printf("Invalid queue number\n");

}

break;

case 2:

printf("Enter queue number (1 or 2): ");

scanf("%d", &queueNumber);

if (queueNumber == 1) {

Delq1(&queues);

} else if (queueNumber == 2) {

Delq2(&queues);

} else {

printf("Invalid queue number\n");

}

break;

case 3:

printf("Enter queue number (1 or 2): ");

scanf("%d", &queueNumber);

if (queueNumber == 1) {

displayQueue1(&queues);

} else if (queueNumber == 2) {

displayQueue2(&queues);

} else {

printf("Invalid queue number\n");

}

break;

case 4:

exit(0);

default:

printf("Wrong choice\n");

}

}

return 0;

}

**WAP to perform addition of two polynomials using singly linked list.**



* **Write an iterative Reverse() function that reverses a list by rearranging all the next pointers and the head pointer. Ideally, Reverse() should only need to make one pass of the list.**

#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* next;

};

void Reverse(struct Node\*\* headRef) {

    struct Node\* prev = NULL;

    struct Node\* current = \*headRef;

    struct Node\* next = NULL;

    while (current != NULL) {

        next = current->next;  // Store next node

        current->next = prev;  // Reverse current node's pointer

        prev = current;        // Move pointers one position ahead

        current = next;

    }

    \*headRef = prev;  // Update head to new front

}

void push(struct Node\*\* headRef, int newData) {

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

    newNode->data = newData;

    newNode->next = \*headRef;

    \*headRef = newNode;

}

void printList(struct Node\* head) {

    struct Node\* temp = head;

    while (temp != NULL) {

        printf("%d ", temp->data);

        temp = temp->next;

    }

    printf("\n");

}

int main() {

    struct Node\* head = NULL;

    push(&head, 5);

    push(&head, 4);

    push(&head, 3);

    push(&head, 2);

    push(&head, 1);

    printf("Original List: \n");

    printList(head);

    Reverse(&head);

    printf("Reversed List: \n");

    printList(head);

    return 0;

}

**WAP to create doubly linked list and perform following operations on it. A) Insert (all cases) 2. Delete (all cases).**



#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* next;

    struct Node\* prev;

};

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;

}

void insertAtBeginning(struct Node\*\* headRef, int data) {

    struct Node\* newNode = createNode(data);

    newNode->next = \*headRef;

    if (\*headRef != NULL) {

        (\*headRef)->prev = newNode;

    }

    \*headRef = newNode;

}

void insertAtEnd(struct Node\*\* headRef, int data) {

    struct Node\* newNode = createNode(data);

    if (\*headRef == NULL) {

        \*headRef = newNode;

        return;

    }

    struct Node\* temp = \*headRef;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    temp->next = newNode;

    newNode->prev = temp;

}

void insertAfter(struct Node\* prevNode, int data) {

    if (prevNode == NULL) {

        printf("The given previous node cannot be NULL\n");

        return;

    }

    struct Node\* newNode = createNode(data);

    newNode->next = prevNode->next;

    prevNode->next = newNode;

    newNode->prev = prevNode;

    if (newNode->next != NULL) {

        newNode->next->prev = newNode;

    }

}

void deleteFromBeginning(struct Node\*\* headRef) {

    if (\*headRef == NULL) {

        printf("List is empty\n");

        return;

    }

    struct Node\* temp = \*headRef;

    \*headRef = (\*headRef)->next;

    if (\*headRef != NULL) {

        (\*headRef)->prev = NULL;

    }

    free(temp);

}

void deleteFromEnd(struct Node\*\* headRef) {

    if (\*headRef == NULL) {

        printf("List is empty\n");

        return;

    }

    struct Node\* temp = \*headRef;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    if (temp->prev != NULL) {

        temp->prev->next = NULL;

    } else {

        \*headRef = NULL;

    }

    free(temp);

}

void deleteNode(struct Node\*\* headRef, struct Node\* delNode) {

    if (\*headRef == NULL || delNode == NULL) {

        printf("The given node cannot be deleted\n");

        return;

    }

    if (\*headRef == delNode) {

        \*headRef = delNode->next;

    }

    if (delNode->next != NULL) {

        delNode->next->prev = delNode->prev;

    }

    if (delNode->prev != NULL) {

        delNode->prev->next = delNode->next;

    }

    free(delNode);

}

void printList(struct Node\* head) {

    struct Node\* temp = head;

    while (temp != NULL) {

        printf("%d ", temp->data);

        temp = temp->next;

    }

    printf("\n");

}

int main() {

    struct Node\* head = NULL;

    insertAtBeginning(&head, 1);

    insertAtBeginning(&head, 2);

    insertAtBeginning(&head, 3);

    printf("List after inserting at beginning: ");

    printList(head);

    insertAtEnd(&head, 4);

    insertAtEnd(&head, 5);

    printf("List after inserting at end: ");

    printList(head);

    insertAfter(head->next, 6);

    printf("List after inserting after the second node: ");

    printList(head);

    deleteFromBeginning(&head);

    printf("List after deleting from beginning: ");

    printList(head);

    deleteFromEnd(&head);

    printf("List after deleting from end: ");

    printList(head);

    deleteNode(&head, head->next);

    printf("List after deleting the second node: ");

    printList(head);

    return 0;

}

**WAP to merge two sorted Doubly linked lists and display their result.**



#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* next;

    struct Node\* prev;

};

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;

}

void insertAtEnd(struct Node\*\* headRef, int data) {

    struct Node\* newNode = createNode(data);

    if (\*headRef == NULL) {

        \*headRef = newNode;

        return;

    }

    struct Node\* temp = \*headRef;

    while (temp->next != NULL) {

        temp = temp->next;

    }

    temp->next = newNode;

    newNode->prev = temp;

}

struct Node\* mergeSortedLists(struct Node\* head1, struct Node\* head2) {

    if (head1 == NULL) return head2;

    if (head2 == NULL) return head1;

    struct Node\* mergedHead = NULL;

    if (head1->data <= head2->data) {

        mergedHead = head1;

        head1 = head1->next;

    } else {

        mergedHead = head2;

        head2 = head2->next;

    }

    struct Node\* mergedTail = mergedHead;

    while (head1 != NULL && head2 != NULL) {

        if (head1->data <= head2->data) {

            mergedTail->next = head1;

            head1->prev = mergedTail;

            head1 = head1->next;

        } else {

            mergedTail->next = head2;

            head2->prev = mergedTail;

            head2 = head2->next;

        }

        mergedTail = mergedTail->next;

    }

    if (head1 != NULL) {

        mergedTail->next = head1;

        head1->prev = mergedTail;

    } else if (head2 != NULL) {

        mergedTail->next = head2;

        head2->prev = mergedTail;

    }

    return mergedHead;

}

void printList(struct Node\* head) {

    struct Node\* temp = head;

    while (temp != NULL) {

        printf("%d ", temp->data);

        temp = temp->next;

    }

    printf("\n");

}

int main() {

    struct Node\* head1 = NULL;

    struct Node\* head2 = NULL;

    insertAtEnd(&head1, 1);

    insertAtEnd(&head1, 3);

    insertAtEnd(&head1, 5);

    insertAtEnd(&head2, 2);

    insertAtEnd(&head2, 4);

    insertAtEnd(&head2, 6);

    printf("First Sorted List: ");

    printList(head1);

    printf("Second Sorted List: ");

    printList(head2);

    struct Node\* mergedHead = mergeSortedLists(head1, head2);

    printf("Merged Sorted List: ");

    printList(mergedHead);

    return 0;

}

**Implement Push and POP operations of STACK on Doubly linked lists**



#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* next;

    struct Node\* prev;

};

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;

}

void push(struct Node\*\* topRef, int data) {

    struct Node\* newNode = createNode(data);

    if (\*topRef != NULL) {

        newNode->next = \*topRef;

        (\*topRef)->prev = newNode;

    }

    \*topRef = newNode;

    printf("Pushed %d onto the stack\n", data);

}

int pop(struct Node\*\* topRef) {

    if (\*topRef == NULL) {

        printf("Stack underflow\n");

        return -1;

    }

    struct Node\* temp = \*topRef;

    int poppedData = temp->data;

    \*topRef = temp->next;

    if (\*topRef != NULL) {

        (\*topRef)->prev = NULL;

    }

    free(temp);

    printf("Popped %d from the stack\n", poppedData);

    return poppedData;

}

void printStack(struct Node\* top) {

    struct Node\* temp = top;

    printf("Stack: ");

    while (temp != NULL) {

        printf("%d ", temp->data);

        temp = temp->next;

    }

    printf("\n");

}

int main() {

    struct Node\* top = NULL;

    push(&top, 10);

    push(&top, 20);

    push(&top, 30);

    printStack(top);

    pop(&top);

    pop(&top);

    printStack(top);

    pop(&top);

    pop(&top);

    return 0;

}

**Implement ADD and DELETE operations of QUEUE on Doubly linked lists**



#include <stdio.h>

#include <stdlib.h>

struct Node {

    int data;

    struct Node\* next;

    struct Node\* prev;

};

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;

}

void add(struct Node\*\* frontRef, struct Node\*\* rearRef, int data) {

    struct Node\* newNode = createNode(data);

    if (\*rearRef == NULL) {

        \*frontRef = \*rearRef = newNode;

    } else {

        (\*rearRef)->next = newNode;

        newNode->prev = \*rearRef;

        \*rearRef = newNode;

    }

    printf("Added %d to the queue\n", data);

}

int delete(struct Node\*\* frontRef, struct Node\*\* rearRef) {

    if (\*frontRef == NULL) {

        printf("Queue underflow\n");

        return -1;

    }

    struct Node\* temp = \*frontRef;

    int deletedData = temp->data;

    \*frontRef = (\*frontRef)->next;

    if (\*frontRef != NULL) {

        (\*frontRef)->prev = NULL;

    } else {

        \*rearRef = NULL;

    }

    free(temp);

    printf("Deleted %d from the queue\n", deletedData);

    return deletedData;

}

void printQueue(struct Node\* front) {

    struct Node\* temp = front;

    printf("Queue: ");

    while (temp != NULL) {

        printf("%d ", temp->data);

        temp = temp->next;

    }

    printf("\n");

}

int main() {

    struct Node\* front = NULL;

    struct Node\* rear = NULL;

    add(&front, &rear, 10);

    add(&front, &rear, 20);

    add(&front, &rear, 30);

    printQueue(front);

    delete(&front, &rear);

    delete(&front, &rear);

    printQueue(front);

    delete(&front, &rear);

    delete(&front, &rear);

    return 0;

}

**Write a Program to create a Binary Tree and perform following non-recursive operations on it. a. Preorder Traversal b. Count Leaf Nodes c. Count total no. of nodes d. Display height of a tree.**

**Write a Program to create a Binary Tree and perform following non-recursive operations on it. a. Inorder Traversal b. Mirror Image c. Count total no. of nodes d. Display height of a tree.**

**Write a Program to create a Binary Tree and perform following non-recursive operations on it. a. Postorder Traversal b. Display Leaf Nodes c. Count total no. of Leaf nodes d. Display height of a tree.**

**Write a Program to create a Binary Tree and perform following non-recursive operations on it. a. Level-wise Traversal b. Display Leaf Nodes c. Count total no. of Leaf nodes d. Display height of a tree.**

**Write a Program to create a Binary Tree and perform following non-recursive operations on it. a. Levelwise display b. Mirror image c. Display height of a tree.**



**Write a program to illustrate operations on a BST holding numeric keys. The menu must include: • Insert • Delete • Find**



**• Level wise Display**

**Write a program to illustrate operations on a BST holding numeric keys. The menu must include: • Insert • Mirror Image • Find • Height of the tree**



**Write a Program to create a Binary Tree and perform following Non-recursive operations on it. a. Postorder Traversal c. Display Leaf Nodes d. Mirror Image**



**Write a Program to create a Binary Search Tree and perform following non-recursive operations on it. a. Inorder Traversal b. Display Number of Leaf Nodes c. Mirror Image**



**Write a Program to create a Binary Search Tree and perform following non-recursive operations on it. a. Preorder Traversal b. Display total Number of Nodes C. Display Leaf nodes.**



**Write a Program to create a Binary Search Tree and perform deletion of a node from it. Also display the tree in nonrecursive postorder way.**



**Write a Program to create a Binary Search Tree and display it levelwise. Also perform deletion of a node from it.**



**Write a Program to create a Binary Search Tree and display its mirror image with and without disturbing the original tree. Also display height of a tree using nonrecursion.**



**Write a program to efficiently search a particular employee record by using Tree data structure. Also sort the data on emp-id in ascending order.**



**Write a Program to create Inorder Threaded Binary Tree and Traverse it in Preorder way.**

**Write a Program to create Inorder Threaded Binary Tree and Traverse it in Inorder way.**



**Write a Program to implement AVL tree and perform different rotations on it and display it Levelwise.**



**Write a Program to accept a graph from user and represent it with Adjacency Matrix and perform BFS traversals on it.**

**Write a Program to accept a graph from user and represent it with Adjacency Lists and perform BFS traversals on it.**

**Write a Program to accept a graph from user and represent it with Adjacency Matrix and perform DFS traversals on it.**

**Write a Program to accept a graph from user and represent it with Adjacency Lists and perform DFS traversals on it.**

**Write a Program to implement Prim’s algorithm to find minimum spanning tree of a user defined graph. Use Adjacency List to represent a graph.**



**Write a Program to implement Kruskals’s algorithm to find minimum spanning tree of a user defined graph. Use Adjacency Matrix to represent a graph.**

**Write a Program to implement Kruskal’s algorithm to find minimum spanning tree of a user defined graph. Use Adjacency List to represent a graph.**

**Write a Program to implement Dijkstra’s algorithm to find shortest distance between two nodes of a user defined graph. Use Adjacency List to represent a graph**

**Write a Program to implement Dijkstra’s algorithm to find shortest distance between two nodes of a user defined graph. Use Adjacency Matrix to represent a graph**

|  |
| --- |
| **WAP to implement Heap sort on 1D array of Student structure (contains student\_name, student\_roll\_no, total\_marks), with key as student\_roll\_no. And count the number of swap performed.** |
| **WAP to implement Quick sort on 1D array of Employee structure (contains employee\_name, emp\_no, emp\_salary), with key as emp\_no. And count the number of swap performed.** |
| **Assume that an array A with n elements was sorted in an ascending order, but two of its elements swapped their positions by a mistake while maintaining the array. Write a code to identify the swapped pair of elements and their positions in the asymptotically best possible time. [Assume that all given elements are distinct integers.]** |
| **Implement following hashing Techniques by assuming suitable input and Table Size.**   1. **Linear Probing With Chaining Without Replacement**   **Also mention number of collisions occurred while inserting a Data in hash table.** |
| **Implement following hashing Techniques by assuming suitable input and Table Size.**   1. **Linear Probing With Chaining With Replacement**   **Also mention number of collisions occurred while inserting a Data in hash table.** |
|  |
|  |
|  |