STACK AND QUEUE

CONVERT INFIX TO POSTFIX

CODE:

```
#INCLUDE <STDIO.H>

#INCLUDE <STDLIB.H>

#INCLUDE <CTYPE.H>

#DEFINE MAX 100

TYPEDEF STRUCT {

   INT TOP;

   CHAR ITEMS[MAX];

} STACK;

VOID PUSH(STACK *S, CHAR C) {

   S->ITEMS[++(S->TOP)] = C;

}

CHAR POP(STACK *S) {

   RETURN S->ITEMS[(S->TOP)--];

}

CHAR PEEK(STACK *S) {

   RETURN S->ITEMS[S->TOP];

}

INT ISEMPTY(STACK *S) {

   RETURN S->TOP == -1;

}

INT PRECEDENCE(CHAR OP) {

   SWITCH (OP) {

      CASE '+': CASE '-': RETURN 1;

      CASE '*': CASE '/': RETURN 2;

      CASE '^': RETURN 3;

      DEFAULT: RETURN 0;

   }

}

VOID INFIXTOPOSTFIX(CONST CHAR *INFIX, CHAR *POSTFIX) {

   STACK S;

   S.TOP = -1;
```

```
WHILE (*INFIX) {
    IF (ISALNUM(*INFIX)) {
        *POSTFIX++ = *INFIX;
    } ELSE IF (*INFIX == '(') {
        PUSH(&S, *INFIX);
    } ELSE IF (*INFIX == ')') {
        WHILE (!ISEMPTY(&S) && PEEK(&S) != '(') {
            *POSTFIX++ = POP(&S);
        }
        POP(&S);
    } ELSE {
        WHILE (!ISEMPTY(&S) && PRECEDENCE(PEEK(&S)) >= PRECEDENCE(*INFIX)) {
            *POSTFIX++ = POP(&S);
        }
        PUSH(&S, *INFIX);
    }
    INFIX++;
}
    WHILE (!ISEMPTY(&S)) {
        *POSTFIX++ = POP(&S);
    }
    *POSTFIX = '\0';
}
INT MAIN() {
    CHAR INFIX[MAX], POSTFIX[MAX];
    PRINTF("ENTER INFIX EXPRESSION: ");
    FGETS(INFIX, MAX, STDIN);
    SIZE_T LENGTH = STRLEN(INFIX);
    IF (LENGTH > 0 && INFIX[LENGTH - 1] == '\N') {
        INFIX[LENGTH - 1] = '\0';
    }
    INFIXTOPOSTFIX(INFIX, POSTFIX);
    PRINTF("POSTFIX EXPRESSION: %S\N", POSTFIX);
    RETURN 0;
}
```

Output:

Enter infix expression: A+B*(C-D)/E

Postfix expression: ABCD-*E/+

Array implementation in Queue:

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct {

    int front, rear, size;

    int array[MAX];

} Queue;

void initQueue(Queue *q);

int isEmpty(Queue *q);

int isFull(Queue *q);

void enqueue(Queue *q, int value);

int dequeue(Queue *q);

int peek(Queue *q);

void display(Queue *q);


int main() {

    Queue q;

    initQueue(&q);

    enqueue(&q, 10);

    enqueue(&q, 20);

    enqueue(&q, 30);

    enqueue(&q, 40);

    printf("Queue after enqueuing 10, 20, 30, 40:\n");

    display(&q);

    printf("Front element is %d\n", peek(&q));
```

```
    PRINTF("DEQUEUED ELEMENT IS %D\N", DEQUEUE(&Q));
    PRINTF("DEQUEUED ELEMENT IS %D\N", DEQUEUE(&Q));
    PRINTF("QUEUE AFTER DEQUEUING TWO ELEMENTS:\N");
    DISPLAY(&Q);

    RETURN 0;
}
VOID INITQUEUE(QUEUE *Q) {
    Q->FRONT = 0;
    Q->REAR = -1;
    Q->SIZE = 0;
}
INT ISEMPTY(QUEUE *Q) {
    RETURN Q->SIZE == 0;
}
INT ISFULL(QUEUE *Q) {
    RETURN Q->SIZE == MAX;
}
VOID ENQUEUE(QUEUE *Q, INT VALUE) {
    IF (ISFULL(Q)) {
        PRINTF("QUEUE IS FULL\N");
        RETURN;
    }
    Q->REAR = (Q->REAR + 1) % MAX;
    Q->ARRAY[Q->REAR] = VALUE;
    Q->SIZE++;
}
INT DEQUEUE(QUEUE *Q) {
    IF (ISEMPTY(Q)) {
        PRINTF("QUEUE IS EMPTY\N");
        RETURN -1;
    }
    INT VALUE = Q->ARRAY[Q->FRONT];
    Q->FRONT = (Q->FRONT + 1) % MAX;
```

```
    Q->SIZE--;
    RETURN VALUE;
}
INT PEEK(QUEUE *Q) {
    IF (ISEMPTY(Q)) {
        PRINTF("QUEUE IS EMPTY\N");
        RETURN -1;
    }
    RETURN Q->ARRAY[Q->FRONT];
}
VOID DISPLAY(QUEUE *Q) {
    IF (ISEMPTY(Q)) {
        PRINTF("QUEUE IS EMPTY\N");
        RETURN;
    }
    INT I = Q->FRONT;
    INT COUNT = Q->SIZE;
    WHILE (COUNT--) {
        PRINTF("%D ", Q->ARRAY[I]);
        I = (I + 1) % MAX;
    }
    PRINTF("\N");
}
```

OUTPUT:

```
QUEUE AFTER ENQUEUING 10, 20, 30, 40:
10 20 30 40
FRONT ELEMENT IS 10
DEQUEUED ELEMENT IS 10
DEQUEUED ELEMENT IS 20
QUEUE AFTER DEQUEUING TWO ELEMENTS:
30 40
```

LINKED LIST IMPLEMENTATION:

CODE:

```
#INCLUDE <STDIO.H>
```

```
#INCLUDE <STDLIB.H>

TYPEDEF STRUCT NODE {

   INT DATA;

   STRUCT NODE *NEXT;

} NODE;

TYPEDEF STRUCT {

   NODE *FRONT;

   NODE *REAR;

} QUEUE;

VOID INITQUEUE(QUEUE *Q);

INT ISEMPTY(QUEUE *Q);

VOID ENQUEUE(QUEUE *Q, INT VALUE);

INT DEQUEUE(QUEUE *Q);

INT PEEK(QUEUE *Q);

VOID DISPLAY(QUEUE *Q);


INT MAIN() {

   QUEUE Q;

   INITQUEUE(&Q);

   ENQUEUE(&Q, 10);

   ENQUEUE(&Q, 20);

   ENQUEUE(&Q, 30);

   ENQUEUE(&Q, 40);

   PRINTF("QUEUE AFTER ENQUEUING 10, 20, 30, 40:\N");

   DISPLAY(&Q);

   PRINTF("FRONT ELEMENT IS %D\N", PEEK(&Q));

   PRINTF("DEQUEUED ELEMENT IS %D\N", DEQUEUE(&Q));

   PRINTF("DEQUEUED ELEMENT IS %D\N", DEQUEUE(&Q));

   PRINTF("QUEUE AFTER DEQUEUING TWO ELEMENTS:\N");

   DISPLAY(&Q);

   RETURN 0;

}

VOID INITQUEUE(QUEUE *Q) {

   Q->FRONT = NULL;

   Q->REAR = NULL;
```

```
}
int isEmpty(Queue *q) {
    return q->front == NULL;
}
void enqueue(Queue *q, int value) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    if (isEmpty(q)) {
        q->front = newNode;
        q->rear = newNode;
    } else {
        q->rear->next = newNode;
        q->rear = newNode;
    }
}
int dequeue(Queue *q) {
    if (isEmpty(q)) {
    printf("Queue is empty\n");
    return -1;
}
Node *temp = q->front;
int value = temp->data;
q->front = q->front->next;
if (q->front == NULL) {
    q->rear = NULL;
}
free(temp);
    return value;
}
int peek(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;
    }
```

```
    RETURN Q->FRONT->DATA;
}
VOID DISPLAY(QUEUE *Q) {
    IF (ISEMPTY(Q)) {
        PRINTF("QUEUE IS EMPTY\N");
        RETURN;
    }
    NODE *CURRENT = Q->FRONT;
    WHILE (CURRENT) {
        PRINTF("%D ", CURRENT->DATA);
        CURRENT = CURRENT->NEXT;
    }
    PRINTF("\N");
}
```

OUTPUT:

QUEUE AFTER ENQUEUING 10, 20, 30, 40:

10 20 30 40

FRONT ELEMENT IS 10

DEQUEUED ELEMENT IS 10

DEQUEUED ELEMENT IS 20

QUEUE AFTER DEQUEUING TWO ELEMENTS:

30 40