```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
char stack[50];
int top = -1;
struct stck
{
    int t;
    int item[500];
};
void Push(struct stck *ptr, int x)
{
    if (ptr->t == 499)
    {
        printf("Stack is overflow\n");
    }
    else
    {
        ptr->t = ptr->t + 1;
        int tp = ptr->t;
        ptr->item[tp] = x;
    }
}

int Pop(struct stck *ptr)
{
    int temp;
    if (ptr->t == -1)
    {
        printf("Error: stack underflow n");
    }
    else
    {
        int t = ptr->t;
        temp = ptr->item[t];
        ptr->t -= 1;
        return temp;
    }
}
bool empty(struct stck *ptr)
{
    return (ptr->t == -1) ? true : false;
}

void NGE(int a[], int n)
{
    int i = 0;
    struct stck s;
    s.t = -1;
    int ele, next;
    Push(&s, a[0]);
    printf("The Next Greater Element of each element are:\n");
    for (i = 1; i < n; i++)
    {
        next = a[i];
```

```
        if (empty(&s) == false)
        {

            ele = Pop(&s);
            while (ele < next)
            {
                printf(" %d --- %d\n", ele, next);
                if (empty(&s) == true)
                    break;
                ele = Pop(&s);
            }

            if (ele > next)
                Push(&s, ele);
        }
        Push(&s, next);
    }

    while (empty(&s) == false)
    {
        ele = Pop(&s);
        next = -1;
        printf(" %d --- %d\n", ele, next);
    }
}

struct Stack
{
    int top;
    unsigned capacity;
    int *array;
};

struct Queue
{
    int front, rear, size;
    unsigned capacity;
    int *array;
};

struct Stack *createStack(unsigned 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 isFull(struct Stack *stack)
{
    return stack->top == stack->capacity - 1;
}

int isEmpty(struct Stack *stack)
```

```
{
    return stack->top == -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];
}

struct Queue *createQueue(unsigned capacity)
{
    struct Queue *queue = (struct Queue *)malloc(sizeof(struct
Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;
    queue->array = (int *)malloc(queue->capacity * sizeof(int));
    return queue;
}

int QueueisFull(struct Queue *queue)
{
    return (queue->size == queue->capacity);
}

int QueueisEmpty(struct Queue *queue)
{
    return (queue->size == 0);
}

void enqueue(struct Queue *queue, int item)
{
    if (QueueisFull(queue))
        return;
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
}

int dequeue(struct Queue *queue)
{
```

```
    if (QueueisEmpty(queue))
        return -1;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

int front(struct Queue *queue)
{
    if (QueueisEmpty(queue))
        return -1;
    return queue->array[queue->front];
}

int rear(struct Queue *queue)
{
    if (QueueisEmpty(queue))
        return -1;
    return queue->array[queue->rear];
}

void pu(char c)
{
    top++;
    stack[top] = c;
}

char po()
{
    if (top == -1)
        return -1;
    else
    {
        return stack[top--];
    }
}
int check(char c)
{
    if ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z') || (c >= '0'
&& c <= '9'))
    {
        return 1;
    }
    return 0;
}
int priority(char c)
{
    if (c == '(')
        return 0;
    if (c == '+' || c == '-')
        return 1;
    if (c == '*' || c == '/')
        return 2;
    if (c == '^')
        return 3;
    return 0;
```

```
}

int main()
{ // Question -1
    // -----------

    printf("Question-1\n");
    printf("\n");
    int num;
    printf("Enter number of elements you want to Insert\n");
    scanf("%d", &num);
    printf("Enter a sequence of number to find the Next Greater
Element \n");
    int a[num];
    for (int k = 0; k < num; k++)
    {
        scanf("%d", &a[k]);
    }
    NGE(a, num);
    printf("\n");

    // Question-2
    // ---------

    /* printf("Question-2\n");
    printf("\n");
    struct Stack *st = createStack(100);
    int expected = 1;
    int fnt;

    struct Queue *q = createQueue(100);

    enqueue(q, 3);
    enqueue(q, 4);
    enqueue(q, 6);
    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 9);
    enqueue(q, 0);
    printf("To check the given array is sorted or not\n");
    while (!QueueisEmpty(q))
    {
        fnt = front(q);
        dequeue(q);

        if (fnt == expected)
            expected++;

        else
        {

            if (isEmpty(st))
            {
                push(st, fnt);
            }

            else if (!isEmpty(st) && peek(st) < fnt)
```

```
        {
            printf("No\n");
            return;
        }

        else
            push(st, fnt);
    }

    while (!isEmpty(st) && peek(st) == expected)
    {
        pop(st);
        expected++;
    }
}

if (expected - 1 == 7 && isEmpty(st))
{
    printf("Yes\n");
}

else
{
    printf("No\n");
}
*/

// Question-3
// -----------

printf("Question-3\n");
printf("\n");
top = -1;
char infix[20];
char *c, x, k;
printf("Enter the Infix expression : \n");
scanf("%s", infix);
c = infix;
printf("The Resultant Prefix Expression is:\n");
while (*c != '\0')
{
    if (check(*c))
        printf("%c", *c);
    else if (*c == '(')
        pu(*c);
    else if (*c == ')')
    {
        k = po();
        while ((x = k) != '(')
        {
            printf("%c", x);
            k = po();
        }
    }
    else
    {
        int pstack = priority(stack[top]);
```

```
            int pe = priority(*c);
            while (pstack >= pe)
            {
                k = po();
                printf("%c", k);
                pstack = priority(stack[top]);
                pe = priority(*c);
            }
            pu(*c);
        }
        c++;
    }

    while (top != -1)
    {
        k = po();
        printf("%c", k);
    }
    printf("\n");

    return 0;
}
```