DSA LAB PROGRAMS _ STACKS:

By Sindhu Sankati AP19110010477 CSE-G

PROGRAM 34

Title: Implementing Stacks using Arrays

Objective:

Write a C program to implement the STACK operation using array as a data structure. Users must be given the following choices to perform relevant tasks.

1. Push an element on to the STACK.
2. Pop and element from the STACK.
3. Peek the STACK.
4. Display the STACK.
5. Exit the program.

Explanation:

Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek: Returns top element of stack

Pseudo Code:

>Push

```
/*Assuming that array-stack can hold maximum N elements*/
/*inital stack index; top = -1*/
  1. Read item
  2. if(top == N-1)
  3.     print(“overflow”)
  4. else
  5.     top++
  6.      stack[top] = item
  7. END
```

>Pop

```
/*Assuming that array-stack can hold maximum N elements*/
  1. if(top == -1)
  2.     print(“underflow”)
  3. else{
  4.      top--
  5.      return stack[top+1]
      }
  6. END
```

>Display

```
1. while(top != -1){
2.     print stack[top]
3.     top--
   }
4. END
```

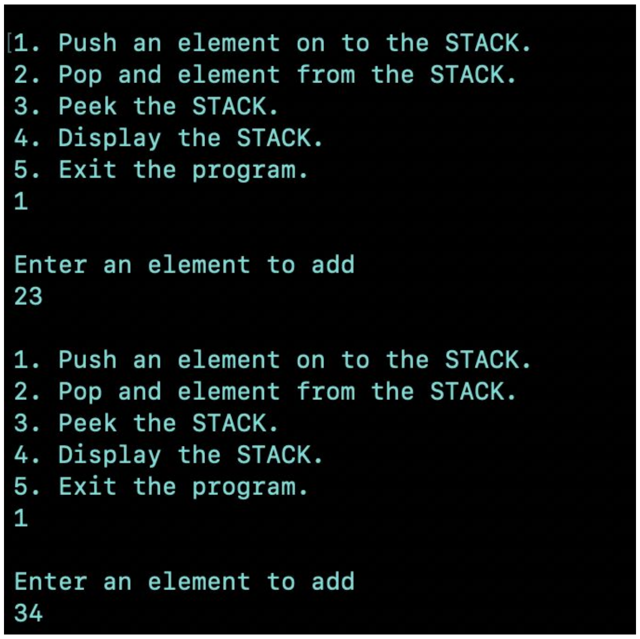
Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define  max 1000//max elements in stack

//defining stack
typedef struct STACK{
     int ar[max];
     int top;
}stack;
//function to push elements into stack
void push(stack *s, int data){
     //overflow condition
     if(s->top >= max-1){
           printf("overflow\n");
           return;
     }
     s->top++;
     s->ar[s->top] = data;
}
//function to pop elements
int pop(stack *s){
     //underflow condition
     if(s->top < 0) {
           printf("Underflow\n");
           return INT_MIN;
           //raise: if element stored == INT_MIN , the function fails
     }
     int temp = s->ar[s->top];
     s->top--;
     return temp;
}
//return the top element without disturbing the top
int peek(stack s){
		return s.ar[s.top];
	}

	//displays elements from top
	void display(stack s){
		int i;
		if(s.top == -1){
			printf("Empty\n");
		}
		for(i =s.top;i>-1;i--){
			printf("%d\n",s.ar[i]);
		}
		printf("\n");
	}

	//main
	int main(int argc, char const *argv[])
	{
		//initialized variables needed
		stack s;
		s.top = -1;
		int choice,data;
		//runs loop till user chooses exit --> 5
		while(1){
			//menu
			printf("\n1. Push an element on to the STACK.\n"
					"2. Pop and element from the STACK.\n"
					"3. Peek the STACK.\n"
					"4. Display the STACK.\n"
					"5. Exit the program.\n");
			scanf("%d",&choice);
			//performs action according the choice
			switch(choice){
				case 1:{
					printf("\nEnter an element to add\n");
					scanf("%d",&data);
					push(&s,data);
					break;
				}
				case 2:{
					data =pop(&s);
					if(data != INT_MIN)
						printf("%d is removed\n",data);
					break;
				}
				case 3:{
```

```c
                        printf("%d is the top\n",peek(s) );
                        break;
                    }
                    case 4:{
                        display(s);
                        break;
                    }
                    case 5:{
                        exit(0);
                        break;
                    }
                    default: printf("no such option choose again\n");
                }
            }
            return 0;
    }
```

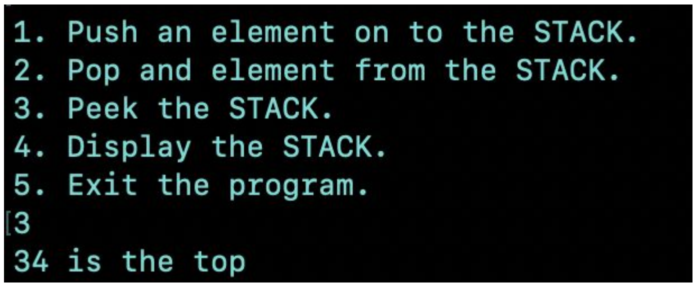
Output:

<u>Push - 2 elements</u>



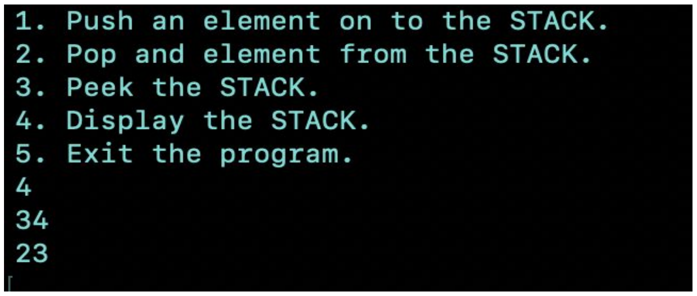
```
1. Push an element on to the STACK.
2. Pop and element from the STACK.
3. Peek the STACK.
4. Display the STACK.
5. Exit the program.
1

Enter an element to add
23

1. Push an element on to the STACK.
2. Pop and element from the STACK.
3. Peek the STACK.
4. Display the STACK.
5. Exit the program.
1

Enter an element to add
34
```

<u>peek - element</u>



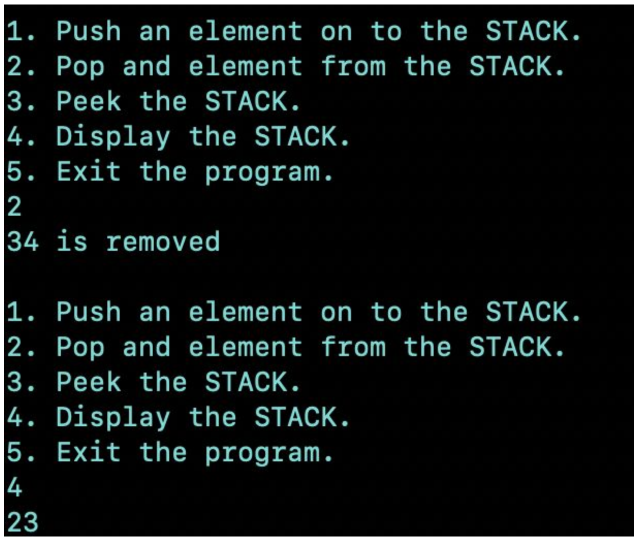
```
1. Push an element on to the STACK.
2. Pop and element from the STACK.
3. Peek the STACK.
4. Display the STACK.
5. Exit the program.
3
34 is the top
```

<u>Display</u>



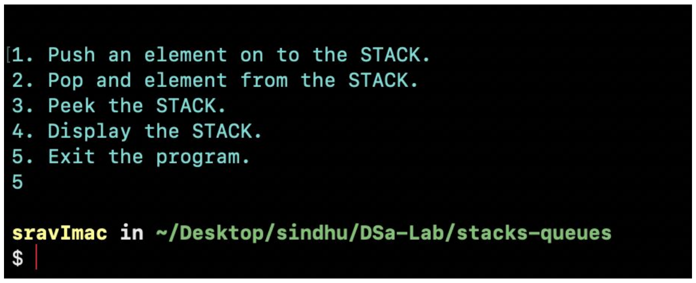
```
1. Push an element on to the STACK.
2. Pop and element from the STACK.
3. Peek the STACK.
4. Display the STACK.
5. Exit the program.
4
34
23
```

Pop and Display:



```
1. Push an element on to the STACK.
2. Pop and element from the STACK.
3. Peek the STACK.
4. Display the STACK.
5. Exit the program.
2
34 is removed

1. Push an element on to the STACK.
2. Pop and element from the STACK.
3. Peek the STACK.
4. Display the STACK.
5. Exit the program.
4
23
```

Exiting program



```
1. Push an element on to the STACK.
2. Pop and element from the STACK.
3. Peek the STACK.
4. Display the STACK.
5. Exit the program.
5

sravImac in ~/Desktop/sindhu/DSa-Lab/stacks-queues
$
```

Conclusion:

We get expected outputs when performing any actions mentioned in objective. (Other observed exceptions are commented in the code).

The time complexity for push, pop , peek is O(1) for best, worst and average cases. And for Display it is O(n).

PROGRAM 35

Title: Reversing a string using Stacks

Objective:

Write a C program to reverse a string using STACK.

Explanation:

When give with a string like “C Programming” it needs to be converted into “gninmmargorp C”. We can achieve this by using STACK data structure since it fools FILO(First In Last Out).

Pseudo Code:

>Reversing

```
/*Assuming that string is less than equal to the max size of the stack*/

1. READ string, INTIALIZE stack, i = 0
2. FOR i < string.size()
3.     push(string[i])/*pushes into stack*/
4. WHILE (a = pop() != ‘0’)
5.     string[i] = a
6. PRINT string
7. END
```

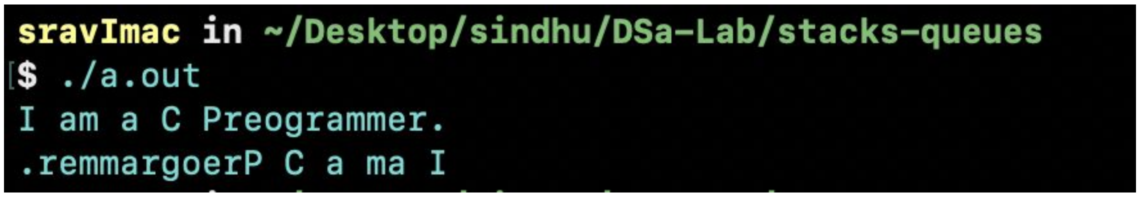
Note: pushing and popping pseudo codes are alredy explained above.

Code:

```c
#include <stdio.h>
#define max 1000
//globally initalized stack
char stack[max]; int top = -1;
//function which return the top element of the global stack
char pop(){

     //underflow condion
     if(top == -1)
           return '0';
     char res = stack[top];
     top--;
     return res;
}
//fuction pushes <char> data into the global stack
void push(char data){
     //overflow condition
     if(top == max-1)
           printf("overflow\n");
     top++;
     stack[top] = data;
```

```
	}
	//main
	int main(){
		char string[max];

		//inputting the string
		scanf("%[^\n]%*c",string);
		int i ;
		//pushes all the elements in the string into stack
		for(i = 0;string[i] !='\0';i++){
			push(string[i]);
		}
		//pops all elements from stack a d intializes to string
		for(i=0;i<max;i++){
			int c = pop();
			if(c == '0')
				break;
			string[i] = c;
		}
		//prints the string after reversing
		printf("%s",string);
		return 0;
	}
```

Output:



```
sravImac in ~/Desktop/sindhu/DSa-Lab/stacks-queues
$ ./a.out
I am a C Preogrammer.
.remmargoerP C a ma I
```

Conclusion:

It prints the output as expected. Exception when the string contains ‘0’ since it was given as the “underflow” condition in pop() function. This can be overcomed by using a linked lists.
It takes O(n) for pushing all elements into the stack and O(n) to pop all the elements. Therefore time complexity is O(2n), where n is the number of characters in the string.

PROGRAM 36:

Title: Conversion of In-Fix to Post-Fix

Objective:

Write a C program to convert the given infix expression to post-fix expression using STACK.

Explanation:

To convert infix expression to postfix expression, we will use the stack data structure. By scanning the infix expression from left to right, when we will get any operand, simply add them to the postfix form, and for the operator and parenthesis, add them in the stack maintaining the precedence of them.

Pseudo Code:

>Post fix conversion

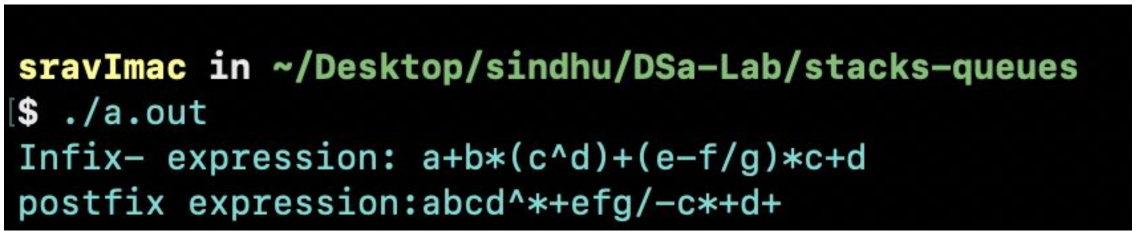
```
//post fix conversion
1. Scan x from left to right and REPEAT steps 3to 0 for each element
   of X UNTIL the STACK is empty:
2. IF an operand is encountred ADD it to EXP
3. IF '(' is encountered, PUSH in to STACK
4. IF an operator is encountered, THEN{

5.     repeatedly POP form STACK and add it to EXP,which has the
   same precedence or higher precedence than operator
6.     PUSH operator to STACK
   }

7. IF a ')' is encountered, THEN
8.       repeatedly POP form STACK and add it to EXP,till '('
   /*Dont add the ')' into the expression*/
   }
9. END
```

Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define max 1000

//intializing stack
typedef struct Stack{
     int top;
     int arr[max];
}stack;
```

```
//pops a element
char pop(stack *s){
     if (s->top != -1)
          return s->arr[s->top--] ;
     return '#';
}
//push a element
void push(stack *s, char op){
     s->arr[++s->top] = op;
}
//preccedence
int Prec(char ch){
     switch (ch) {
     case '+': return 1;
     case '-': return 1;
     case '*': return 2;
     case '/': return 2;
     case '^': return 3;
     }
     return -1;
}
//convert infix two postfix
void in2post(char* exp) {
     int i, k;

     stack s; s.top = -1;

     for (i = 0, k = -1; exp[i]; ++i) {
          //if is a variable
          if ((exp[i] >= 'a' && exp[i] <= 'z') || (exp[i] >= 'A' &&
exp[i] <= 'Z'))
               exp[++k] = exp[i];

          //if a opening bracket
          else if (exp[i] == '(')
               push(&s, exp[i]);

          //if a closing bracket
          else if (exp[i] == ')'){
               //pops all elements form the stack till '('
               while (s.top != -1 && s.arr[s.top] != '(')
                    exp[++k] = pop(&s);
               //if there is no '('
               if (s.top == -1 && s.arr[s.top] != '(')
                    printf("Invalid expression\n");
               //pops '('
            else
                pop(&s);
        }
        //if is a operation
        else {
            //pops till the precendence of stack is greater than
current element
            while (s.top != -1 && Prec(exp[i]) <=
Prec(s.arr[s.top]))
                exp[++k] = pop(&s);
            //pushes the current operation into the stack
            push(&s, exp[i]);
        }
    }
    exp[++k] = pop(&s);

    exp[++k] = '\0';
    printf( "%s", exp );
}
//main
int main() {
    //driver code
    char exp[] = "a+b*(c^d)+(e-f/g)*c+d";
    printf( "Infix- expression: %s\n", exp );
    printf("%s","postfix expression:" );
    in2post(exp);
    return 0;
}
```

Output:



```
sravImac in ~/Desktop/sindhu/DSa-Lab/stacks-queues
[$ ./a.out
Infix- expression: a+b*(c^d)+(e-f/g)*c+d
postfix expression:abcd^*+efg/-c*+d+
```

Conclusion:

The code gives desired outputs as expected. The time complexity of the program is O(n^2) in best, average, worst cases. Not very reliable for large expressions. Space complexity is O(n).

PROGRAM 37:

Title: Conversion of In-Fix to Pre-Fix

Objective:

Write a C program to convert the given infix expression to pre-fix expression using STACK.

Explanation:

When the operators are before operands then it is a prefix espression. We can achive to convert infix to prefix by reversing prefix expression and running through postfix function.

Pseudo Code:

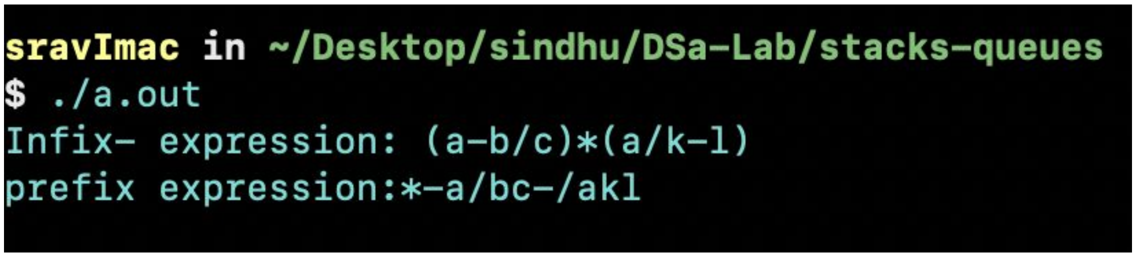
>Pre fix coversion

```
READ string
REVERSE string
/*NOTE while reversing ‘(‘ becomes ‘)’ and ‘)‘ becomes ‘(‘*/
expression = REVERSE(POSTFIX(string))
END
```

Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define max 1000

//intializing stack
typedef struct Stack{
	int top;
	char arr[max];
}stack;
//pops a element
char pop(stack *s){
	if (s->top != -1)
		return s->arr[s->top--] ;
	return '#';
}
//push a element
void push(stack *s, char op){
	s->arr[++s->top] = op;
}
//preccedence
int Prec(char ch){
    switch (ch) {
    case '+': return 1;
    case '-': return 1;
    case '*': return 2;
    case '/': return 2;
    case '^': return 3;
    }
    return -1;
}
//displays elements from top
void display(stack s){
    int i;
    if(s.top == -1){
        printf("Empty\n");
    }
    for(i =s.top;i>-1;i--){
        printf("%c",s.arr[i]);
    }
    printf("\n");
}
//convert infix two prefix
void in2pre(char* exp) {
    int i, n;
    for(n = 0;exp[n];n++);
    stack s; s.top = -1;
    stack pre;pre.top = -1;
    for (i = n-1; i>=0; i--) {
        //if is a variable
        if ((exp[i] >= 'a' && exp[i] <= 'z') || (exp[i] >= 'A' && exp[i]
<= 'Z'))
            push(&pre,exp[i]);
        //if a opening bracket(reverse in prefix) therefore '('' = ')'
        else if (exp[i] == ')')
            push(&s, exp[i]);

        //if a closing bracket
        else if (exp[i] == '('){
            //pops all elements form the stack till '('
            while (s.top != -1 && s.arr[s.top] != ')')
                push(&pre,pop(&s));
            //if there is no '('
            if (s.top == -1 && s.arr[s.top] != ')')
                printf("Invalid expression\n");
            //pops '('
            else
                pop(&s);
        }
        //if is a operation
        else {
            //pops till the precendence of stack is greater than
current element
            while (s.top != -1 && Prec(exp[i]) <= Prec(s.arr[s.top]))
                push(&pre,pop(&s));
            //pushes the current operation into the stack
            push(&s, exp[i]);
        }
    }
    push(&pre,pop(&s));
    display(pre);
}
//main
int main() {
    //driver code
    char exp[] = "(a-b/c)*(a/k-l)";
    printf( "Infix- expression: %s\n", exp );
    printf("%s","prefix expression:" );
    in2pre(exp);
    return 0;
}
```

Output:



```
sravImac in ~/Desktop/sindhu/DSa-Lab/stacks-queues
$ ./a.out
Infix- expression: (a-b/c)*(a/k-l)
prefix expression:*-a/bc-/akl
```

Conclusion:

The code gives desired outputs as expected. The time complexity of the program is O(n^2) in best, average, worst cases. Not very reliable for large expressions. Space complexity is O(n).

PROGRAM 38:

Title: Evaluation of Post-fix and Pre-fix expressions.

Objective:

Write a C program to evaluate the given pre-fix expression, post-fix expression.

Explanation:

Post-fix evaluation: While reading the expression from left to right, push the element in the stack if it is an operand.Pop the two operands from the stack, if the element is an operator and then evaluate it.Push back the result of the evaluation. Repeat it till the end of the expression.
Pre-fix evaluation: We do the same thing as post-fix evalution but we read the expression from right to left.

Pseudo Code:

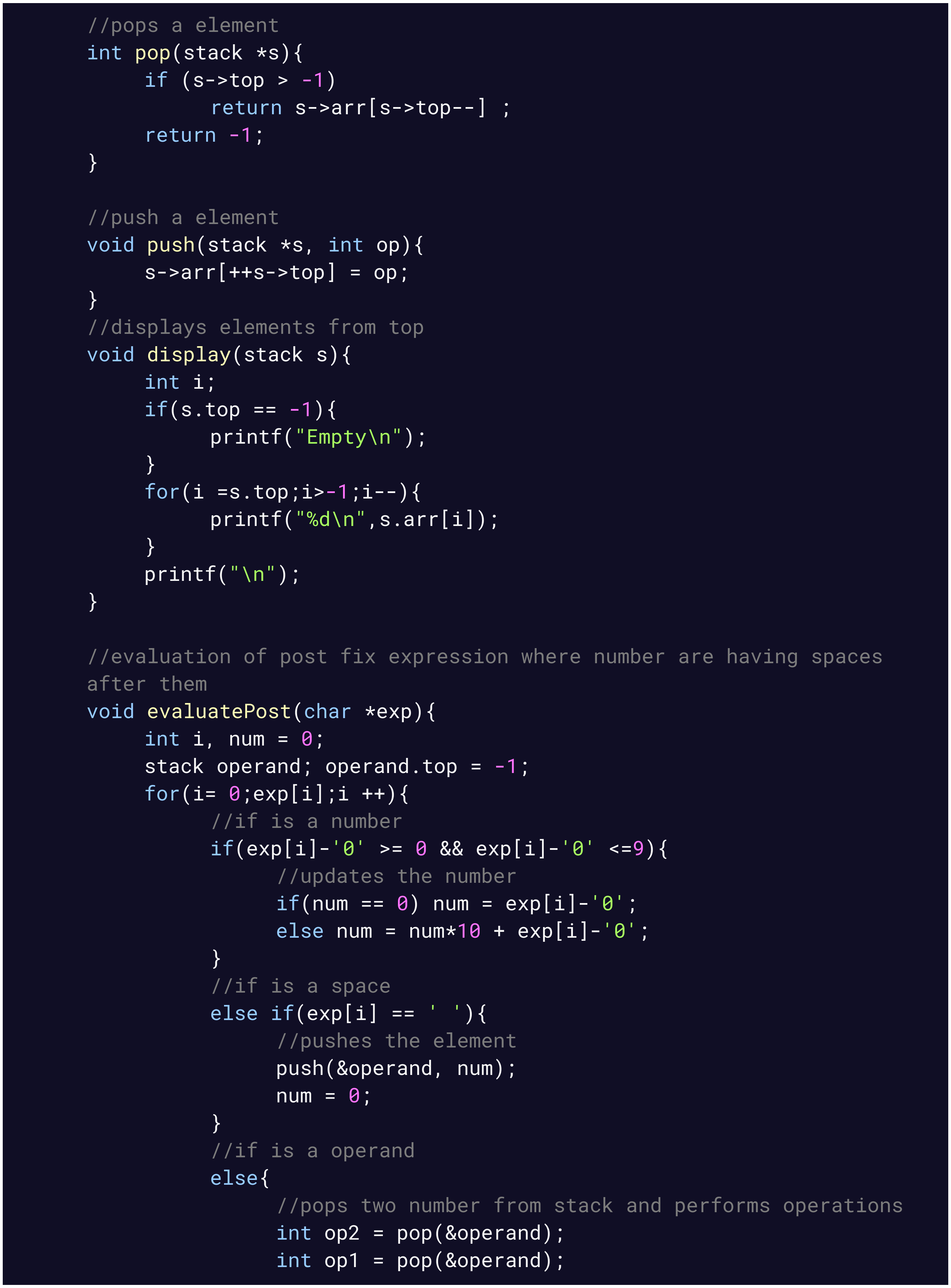
>Post Fix evaluation

```
/*Reading of expression takes place from left to right*/
1. Repeat steps 2 to 8 till the end of expression:
   {
2.     READ the next element
3.     IF element is operand THEN
4.         PUSH the element in the STACK
5.     IF element id operator THEN
      {
6.         POP the element into the STACK
7.         Evaluate the expression formed by the two operands and
   the operator
8.         PUSH the result into STACK
      }
   }
9. END
```

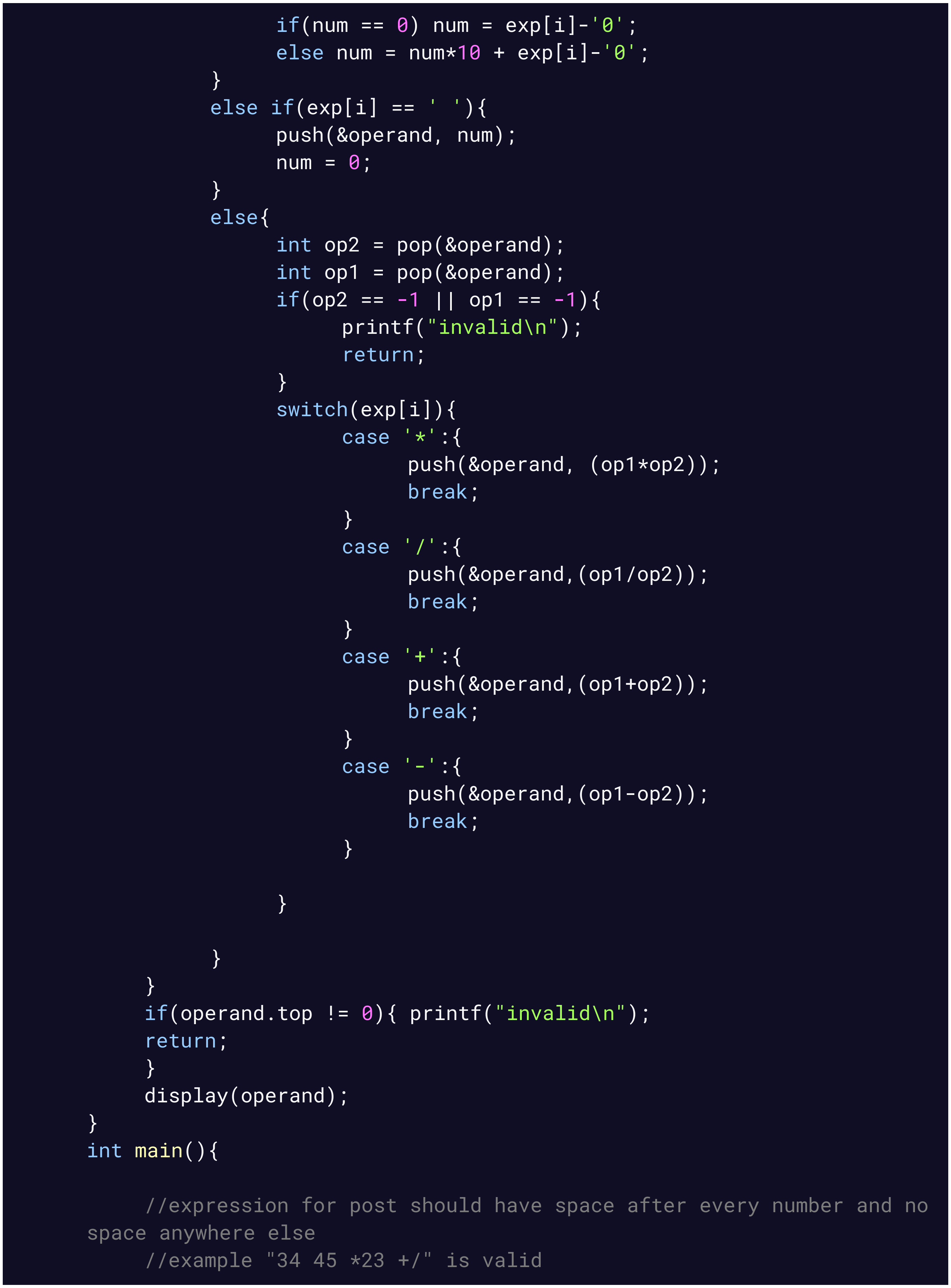
Node: we do the same thing prefix but read expression from right to left

Code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define max 1000

//intializing stack
typedef struct Stack{
    int top;
    int arr[max];
}stack;
```



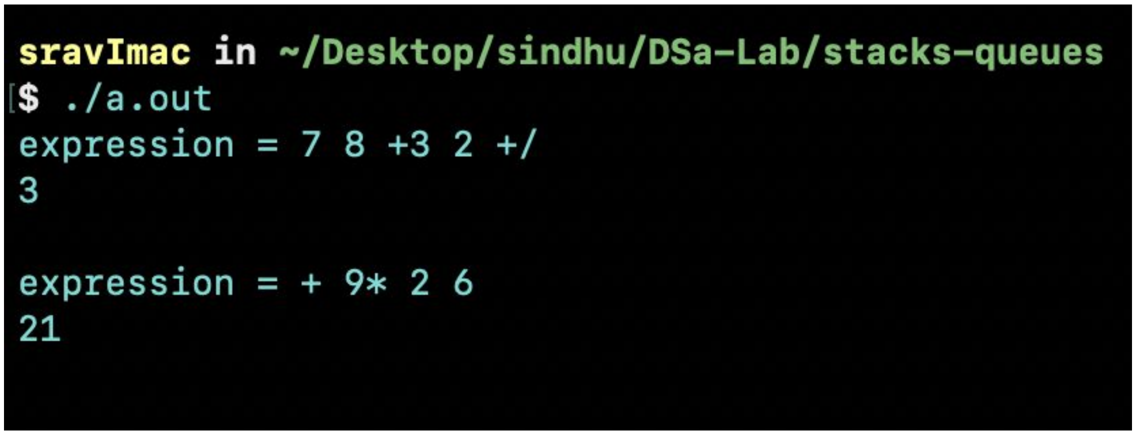
```c
//pops a element
int pop(stack *s){
    if (s->top > -1)
        return s->arr[s->top--] ;
    return -1;
}

//push a element
void push(stack *s, int op){
    s->arr[++s->top] = op;
}
//displays elements from top
void display(stack s){
    int i;
    if(s.top == -1){
        printf("Empty\n");
    }
    for(i =s.top;i>-1;i--){
        printf("%d\n",s.arr[i]);
    }
    printf("\n");
}

//evaluation of post fix expression where number are having spaces after them
void evaluatePost(char *exp){
    int i, num = 0;
    stack operand; operand.top = -1;
    for(i= 0;exp[i];i ++){
        //if is a number
        if(exp[i]-'0' >= 0 && exp[i]-'0' <=9){
            //updates the number
            if(num == 0) num = exp[i]-'0';
            else num = num*10 + exp[i]-'0';
        }
        //if is a space
        else if(exp[i] == ' '){
            //pushes the element
            push(&operand, num);
            num = 0;
        }
        //if is a operand
        else{
            //pops two number from stack and performs operations
            int op2 = pop(&operand);
            int op1 = pop(&operand);
            if(op2 == -1 || op1 == -1){
                printf("invalid\n");
                return;
            }
            //pushes the result into the stack
            switch(exp[i]){

                case '*':{
                    push(&operand, (op1*op2));
                    break;
                }
                case '/':{
                    push(&operand,(op1/op2));
                    break;
                }
                case '+':{
                    push(&operand,(op1+op2));
                    break;
                }
                case '-':{
                    push(&operand,(op1-op2));
                    break;
                }

            }

        }
    }
    //if the stack is not empty then the ex[ression is invalid
    if(operand.top != 0){ printf("invalid\n");
    //return;
    }
    display(operand);
}

//evaluation of pre fix expression where number are having spaces
before them
//is same as post fix expect that it evaluate the expression backward
void evaluatePre(char *exp){
    int n;
    for(n=0;exp[n];n++);
    int i, num = 0;
    stack operand; operand.top = -1;
    for(i = n-1;i>-1;i--){

        if(exp[i]-'0' >= 0 && exp[i]-'0' <=9){
```



```
                if(num == 0) num = exp[i]-'0';
                else num = num*10 + exp[i]-'0';
            }
            else if(exp[i] == ' '){
                push(&operand, num);
                num = 0;
            }
            else{
                int op2 = pop(&operand);
                int op1 = pop(&operand);
                if(op2 == -1 || op1 == -1){
                    printf("invalid\n");
                    return;
                }
                switch(exp[i]){
                    case '*':{
                        push(&operand, (op1*op2));
                        break;
                    }
                    case '/':{
                        push(&operand,(op1/op2));
                        break;
                    }
                    case '+':{
                        push(&operand,(op1+op2));
                        break;
                    }
                    case '-':{
                        push(&operand,(op1-op2));
                        break;
                    }

                }

            }
        }
        if(operand.top != 0){ printf("invalid\n");
        return;
        }
        display(operand);
}
int main(){

    //expression for post should have space after every number and no
space anywhere else
    //example "34 45 *23 +/" is valid
```

```
        //"3 4 5 * 5 / -" is invalid
        char exp[] = "7 8 +3 2 +/";
        evaluatePost(exp);
        //expression for post should have space before every number and
    no space anywhere else
        //example " 34 45* 23+/" is valid
        //" 3 4 5 * 5 / - " is invalid
        char exp2[] = "+ 9* 2 6";
        evaluatePre(exp2);
        return 0;
    }
```

Output:



```
sravImac in ~/Desktop/sindhu/DSa-Lab/stacks-queues
[$ ./a.out
expression = 7 8 +3 2 +/
3

expression = + 9* 2 6
21
```

Conclusion:

We get expected results. But the input is restricted and complex in this algorithimic design. Which can be improved. The time complexity is O(N) for both prefix and postfix evaluation.

PROGRAM 39:

Title: Implementation of Linear Queue using Stacks

Objective:

Write a C program to implement a Linear-Queue, user must choose the following options:

1. Add an element to the Queue – EnQueue.
2. Remove an element from the Queue – DeQueue.
3. Display the elements of the Queue.
4. Terminate the program.

Explanation:

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO).
Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Pseudo Code:

>EnQueue in Array Queue

```
1. IF rear == NULL THEN
   {
2.     rear = front = 0
3.     QUEUE[0] = ITEM
   }
4. ELSE IF rear = N - 1 THEN
5.     PRINT “OverFLow”
6. ELSE
   {
7.     QUEUE[rear + 1] = ITEM
8.     rear = rear + 1
   }
9. END
```

>DeQueue in Array Queue

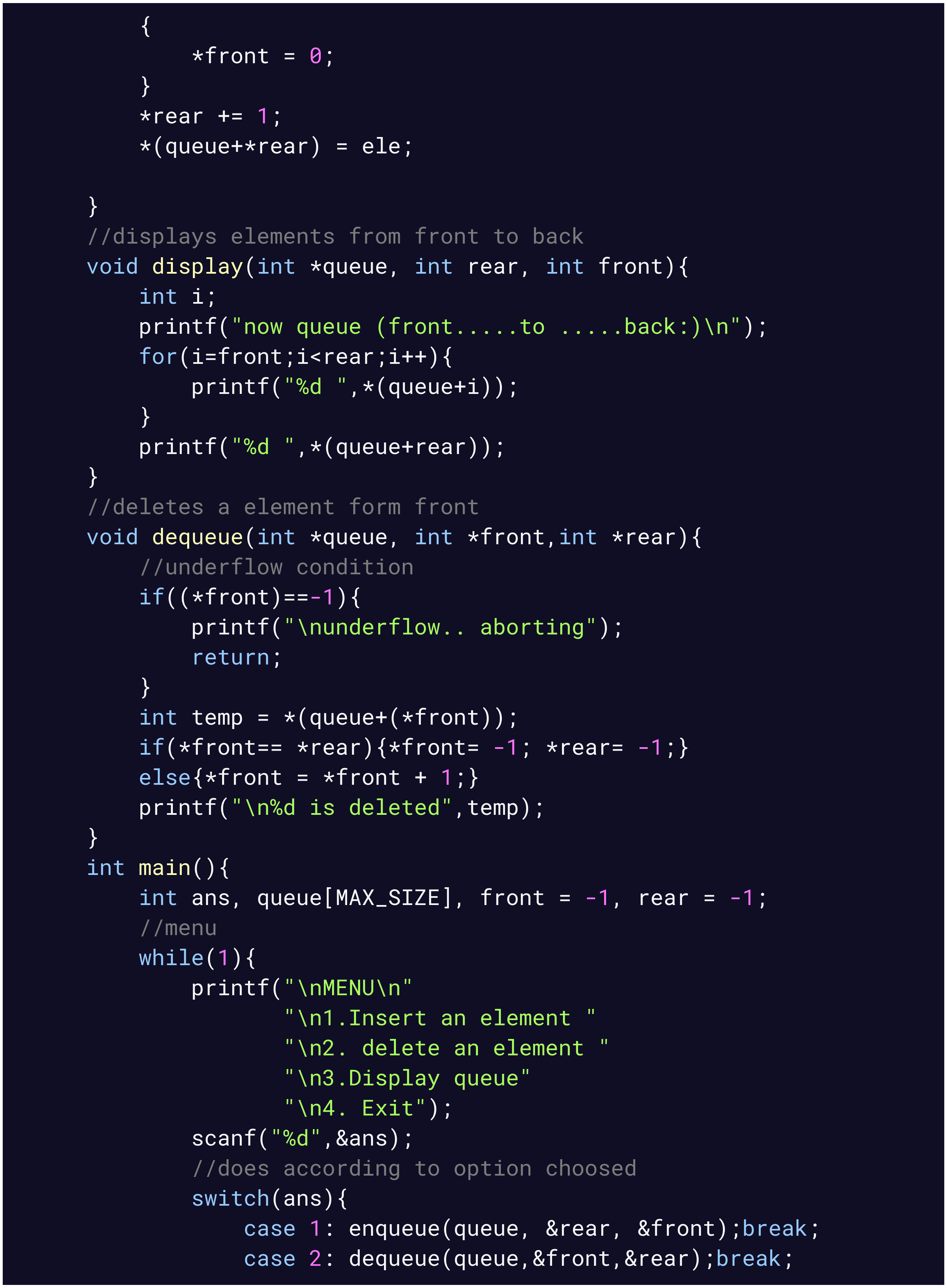
```
1. IF front == NULL THEN
2.     PRINT “Queue Empty”
3. ELSE
   {
4.     ITEM = QUEUE[front]
5.     IF front == rear THEN
6.         front = rear = NULL
7.     ELSE
8.         front = front+1
   }
9. END
```

>Display in Queue

```
5. WHILE (front != rear){
6.     PRINT QUEUE[front]
7.     top++
   }
8. PRINT QUEUE[rear]
9. END
```

Code:

```c
#include <stdio.h>
#include <stdlib.h>
# define MAX_SIZE 100

//adding a eleemt into the queue
void enqueue(int *queue, int *rear,int *front){

    //overflow condition
    if(*rear  == MAX_SIZE-1){

        printf("Overflow\n Aborting...");
        return;
    }
    //inserts a element in rear
    int ele;
    printf("Enter the element to insert");
    scanf("%d",&ele);
    if (*rear == -1)
```

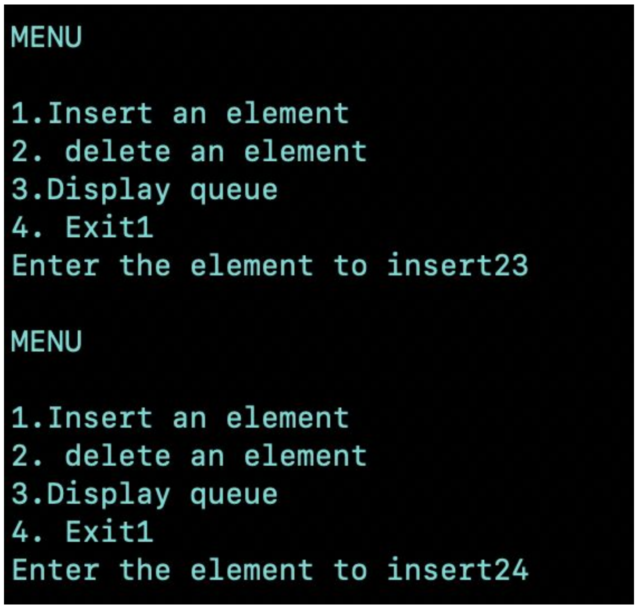


```
    {
        *front = 0;
    }
    *rear += 1;
    *(queue+*rear) = ele;

}
//displays elements from front to back
void display(int *queue, int rear, int front){
    int i;
    printf("now queue (front.....to .....back:)\n");
    for(i=front;i<rear;i++){
        printf("%d ",*(queue+i));
    }
    printf("%d ",*(queue+rear));
}
//deletes a element form front
void dequeue(int *queue, int *front,int *rear){
    //underflow condition
    if((*front)==-1){
        printf("\nunderflow.. aborting");
        return;
    }
    int temp = *(queue+(*front));
    if(*front== *rear){*front= -1; *rear= -1;}
    else{*front = *front + 1;}
    printf("\n%d is deleted",temp);
}
int main(){
    int ans, queue[MAX_SIZE], front = -1, rear = -1;
    //menu
    while(1){
        printf("\nMENU\n"
               "\n1.Insert an element "
               "\n2. delete an element "
               "\n3.Display queue"
               "\n4. Exit");
        scanf("%d",&ans);
        //does according to option choosed
        switch(ans){
            case 1: enqueue(queue, &rear, &front);break;
            case 2: dequeue(queue,&front,&rear);break;
```

```
                case 3: display(queue, rear, front);break;
                case 4: exit(0);break;

            }
        }
        return 0;
    }
```

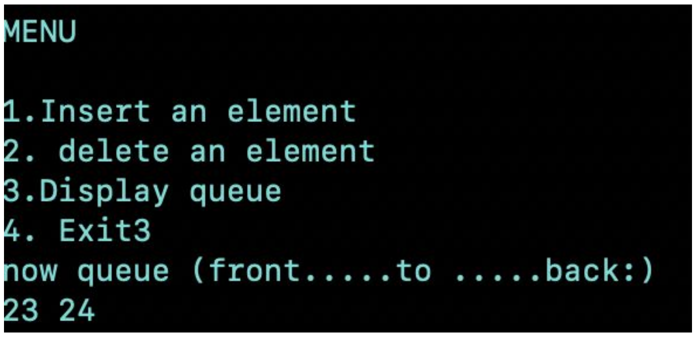
Output:

Enqueue- 2 elements



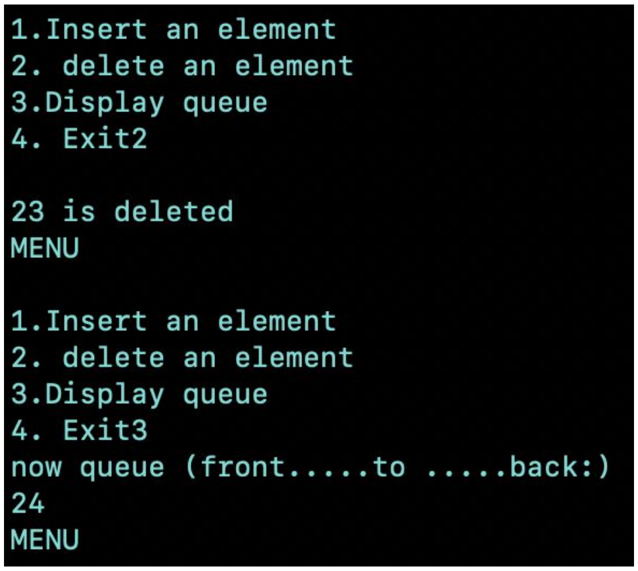
```
MENU

1.Insert an element
2. delete an element
3.Display queue
4. Exit1
Enter the element to insert23

MENU

1.Insert an element
2. delete an element
3.Display queue
4. Exit1
Enter the element to insert24
```

Display



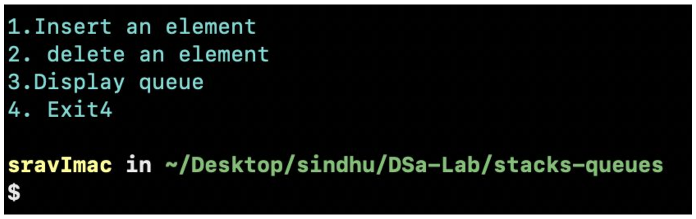
```
MENU

1.Insert an element
2. delete an element
3.Display queue
4. Exit3
now queue (front.....to .....back:)
23 24
```

dequeue and Display:



```
1.Insert an element
2. delete an element
3.Display queue
4. Exit2

23 is deleted
MENU

1.Insert an element
2. delete an element
3.Display queue
4. Exit3
now queue (front.....to .....back:)
24
MENU
```

Exiting program



```
1.Insert an element
2. delete an element
3.Display queue
4. Exit4

sravImac in ~/Desktop/sindhu/DSa-Lab/stacks-queues
$
```

Conclusion:

The code runs as expected. The timecomplexity of deleting and insertion is constant and for displaying it is O(n).

PROGRAM 40:

Title: Implementation of Circular Queue

Objective:

Write a C program to implement a Circular-Queue, user must choose the following options:

1. Add an element to the Queue – EnQueue.
2. Remove an element from the Queue – DeQueue.
3. Display the elements of the Queue.
4. Terminate the program.

Explanation:

A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Pseudo Code:

>EnQueue in Circular Queue

```
/*Assumption that the circular queue is stored in QU with size n*/
  1.   IF (front == 0 AND rear == n-1) or (front == rear +1)THEN
     {
  2.      PRINT “OverFLow”
     }
  3. ELSE
     {
  4.     IF front == NULL THEN
  5.         SET front = 0, rear = 0
  6.     ELSE IF rear = n-1 THEN
  7.         SET rear = 0
  8.     ELSE
  9.         SET rear = (rear + 1)%n
     }
  10.  SET QUEUE[rear] = ITEM
  11.  END
```

>DeQueue in Circular Queue

```
/*Assumption that the circular queue is stored in QU with size n. THis
algorithm will delte an element from the queue and it DITEM*/
/*Check if QU is already empty or not?*/
  1.   IF front == NULL THEN
     {
  2.      PRINT “OverFLow”
  3.      RETURN
     }
  4. ELSE
     {
  5.     SET DITEM = QUEUE[front]
  6.     IF front == rear THEN
  7.     {
  8.         front = rear = NULL
  9.     }
  10.     ELSE IF front = n-1 THEN
  11.         frnt = 0
  12.     ELSE
  13.        front = (front + 1)%n
     }
  14.   END
```

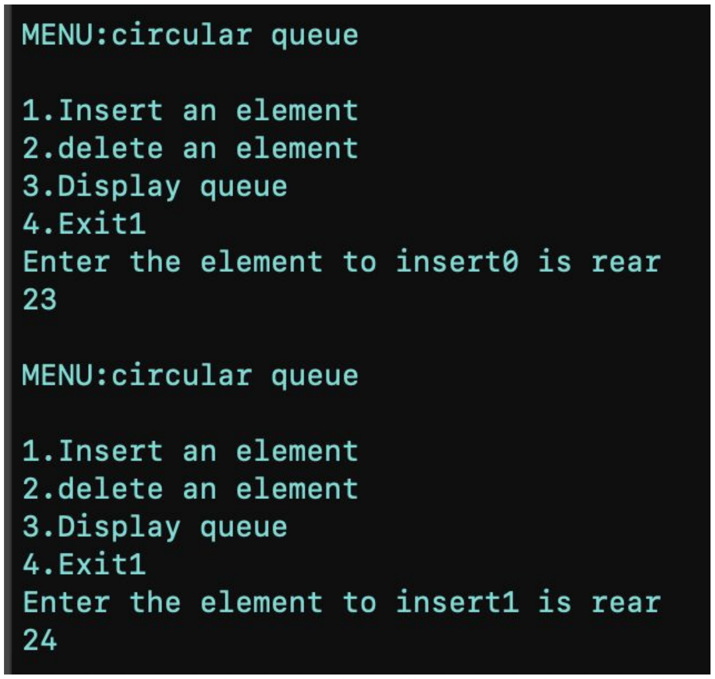
Code:

```c
#include <stdio.h>
#include <stdlib.h>
# define MAX_SIZE 5

//adding a eleemt into the circular queue
void enqueue(int *queue, int *rear,int *front){

    //overflow condition
    if((*rear  == MAX_SIZE-1 && *front == 0)|| (*front== (*rear)+1)){

        printf("Overflow\n Aborting...");
        return;
    }
    //inserts a element in rear
    int ele;
    printf("Enter the element to insert");
    if (*front == -1){
        *front = 0;
    }
    *rear = ((*rear)+1)%MAX_SIZE;
    printf("%d is rear\n",*rear );
    scanf("%d",&ele);
    *(queue+*rear) = ele;

}

//displays elements from front to back of circular queue
void display(int *queue, int rear, int front){
    int i;
    printf("now queue (front.....to .....back:)\n");
    for(i=(front);i!=rear;i= (i+1)%MAX_SIZE){
        printf("%d ",*(queue+i));
    }
    printf("%d ",*(queue+i));
}

//deletes a element form front of circular queue
void dequeue(int *queue, int *front,int *rear){
    //underflow condition
    if((*front)==-1){
        printf("\nunderflow.. aborting");
        return;
    }

    int temp = *(queue+(*front));
    if(*front== *rear){*front= -1; *rear= -1;}
    else{*front = (*front + 1)%MAX_SIZE;}
    printf("\n%d is deleted",temp);

}
int main(){
    int ans, queue[MAX_SIZE], front = -1, rear = -1;
    //menu
    while(1){
        printf("\nMENU:circular queue\n"
               "\n1.Insert an element "
               "\n2.delete an element "
               "\n3.Display queue"
               "\n4.Exit");
        scanf("%d",&ans);
        //does according to option choosed
        switch(ans){
            case 1: enqueue(queue, &rear, &front);break;
            case 2: dequeue(queue,&front,&rear);break;
            case 3: display(queue, rear, front);break;
```

```
                        case 4: exit(0);break;

                   }

              }
              return 0;
         }
```

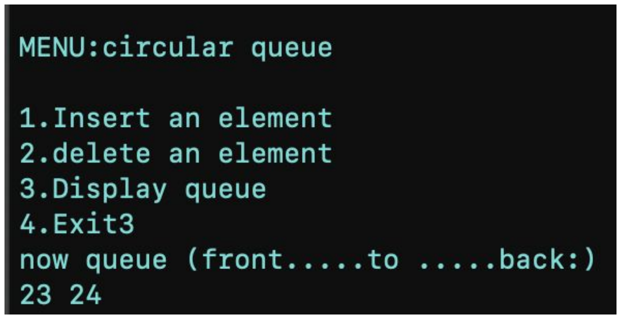
Output:

Enqueue- 2 elements



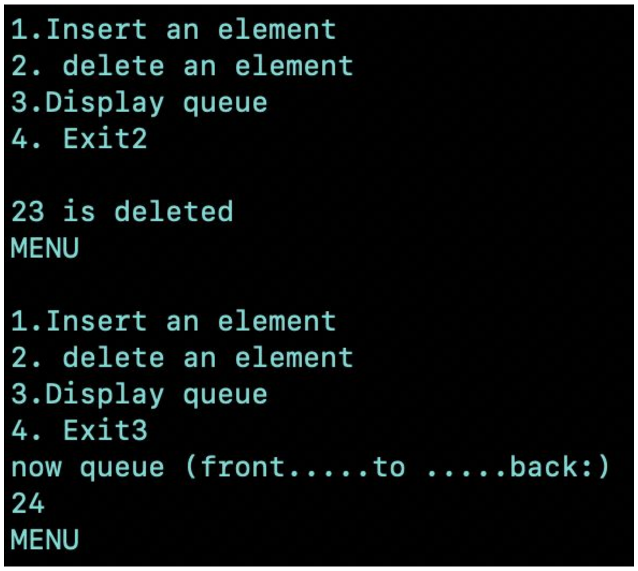
```
MENU:circular queue

1.Insert an element
2.delete an element
3.Display queue
4.Exit1
Enter the element to insert0 is rear
23

MENU:circular queue

1.Insert an element
2.delete an element
3.Display queue
4.Exit1
Enter the element to insert1 is rear
24
```

Display



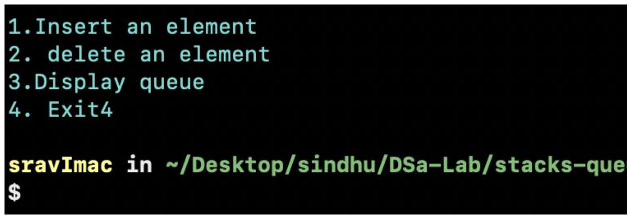
```
MENU:circular queue

1.Insert an element
2.delete an element
3.Display queue
4.Exit3
now queue (front.....to .....back:)
23 24
```

dequeue and Display:



```
1.Insert an element
2. delete an element
3.Display queue
4. Exit2

23 is deleted
MENU

1.Insert an element
2. delete an element
3.Display queue
4. Exit3
now queue (front.....to .....back:)
24
MENU
```

Exiting program



```
1.Insert an element
2. delete an element
3.Display queue
4. Exit4

sravImac in ~/Desktop/sindhu/DSa-Lab/stacks-que
$
```

Conclusion:

The time complexity is same as with linear queue, which is conastant for enqueue and dequeue and o(n) for displaying. But the queue space is utilized.