

DSA Lab

Name: Siddhi Parekh

Reg No.: 221071047

Batch: C

SY Comps

Experiment No.: 4

Aim:

To implement a stack with the help of 2 queues.

Theory:

Here Queues follow 'FIFO' which means First in First Out, i.e. the first element Enqueued into the Queue is the first element to get Dequeued from the Queue, when given the command. Whereas Stack follows 'LIFO' which means Last In First Out, i.e. the first element to get pushed into the stack is the last element to get popped from it.

We can use 2 Queues following 'FIFO' to make one Stack following 'LIFO'.

Push: To insert an element into the stack

Pop: To remove an element from the stack

Enqueue: To insert an element into the Queue

Dequeue: To remove an element from the Queue

EXAMPLE:

2 Queue - 1 stack

Let's say we take 2 Queues
We Enqueue [12, 5, 7, 19, 8] in Queue 1.
Let Queue 2 remain Empty.

Q₁: [12 | 5 | 7 | 19 | 8]

Q₂: [] [] [] [] []

Queue \rightarrow 'FIFO' \rightarrow First In First Out.

\therefore If we dequeue, the first element to move out would be [12]

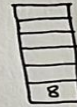
Let us dequeue elements from Q₁ and Enqueue into Q₂ till rear = 0 & front = 0.

The element(s) remaining would be dequeued & printed out in the output.

Q₁: [] [] [] [] []

Q₂: [12 | 5 | 7 | 19 |]

Output: 8



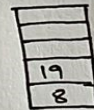
Again we dequeue all elements from Q₂ and Enqueue into Q₁ till rear & front = 0.

The element(s) remaining would be dequeued & printed out in the output.

Q₁: [12 | 5 | 7 |] []

Q₂: [] [] [] [] []

Output: 8 19

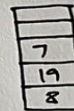


Dequeue elements from Q₁ & enqueue into Q₂ till rear & front = 0, remaining element(s) are dequeued & printed out in the output.

Q₁: [] [] [] [] []

Q₂: [12 | 5 |] [] []

Output: 8 19 7



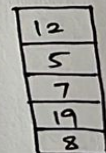
eg: dequeue [7] \rightarrow enqueue in Q₁, dequeue [5] from Q₂ & print

Later dequeue [12] from Q₁ again and print.

\therefore Q₁: [] [] [] [] []

Q₂: [] [] [] [] []

Output: 8 19 7 5 12



ALGORITHM:

Algorithm:

Step 1. Start

Step 2. Create a class named Stack.

Step 3. Initialize the number of elements in a queue and values of rear & front.

Step 4. Giving conditions to the code, if the stack is full (isfull()) or if the stack is empty(isempty()). These conditions are given with the help of rear and front.

Step 5. Create functions named push & pop using front and rear pointers.

Step 6. Give the command to push the element into the queue.

Step 7. If queue is full print "Stack overflow".

Step 8. Give the command to pop the element from the queue.

Step 9. If queue is empty print "Stack Underflow"

Step 10. Now, in the main function define 2 queues, q1 and q2.

Step 11. Enqueue elements into the q1 as many as you want till the Queue is full.

Step 12. Now dequeue all the elements from the q1 till rear = 0 and enqueue them into q2.

Step 13. The remaining element (The last element) is to be dequeued and printed in the output.

Step 14. Now repeat step 12 for q2 .

Step 15. This process continues till both the queues gets underflowed i.e till the queue gets empty.

Step 16. End.

After following this we find that both the queues get underflowed (empty) and in the output we find all the elements in a stack.

CONCLUSION:

This theoretical approach demonstrates how you can implement a stack using two queues efficiently. It also highlights the trade-offs between push and pop operations and helps you appreciate the underlying principles of data structure design and algorithmic efficiency.

CODE:

```
#include<iostream>
using namespace std;

class que
{
    int a[100];
    int front = -1;
    int rear=-1;

    friend class stack_using_queues;

public:

    bool isfull()
    {
        if(rear==99)
        {
            return true;
        }

        else
```

```
{  
return false;  
}  
}
```

```
bool isempty()  
{  
if((rear==-1 && front==-1) || (front>rear))  
{  
return true;  
}
```

```
else  
{  
return false;  
}  
}
```

```
void insert(int data)  
{  
if(isfull())  
{  
cout<<"Queue is full"<<endl;  
}
```

```
else  
{  
rear++;  
a[rear]=data;
```

```
if(rear==0)  
{  
    front=0;  
}  
}
```

```
}
```

```

int del()
{
    if(isempty())
    {
        cout<<"Queue is empty"<<endl;
    }

    else
    {
        return a[front++];
    }
}
};

```

```

class stack_using_queues
{
    que q1,q2;
public:
    void push(int element)
    {
        if(q1.isfull())
        {
            cout<<"Stack overflow"<<endl;
        }

        else
        {
            q1.insert(element);
        }
    }

    void pop()
    {
        while(!q1.isempty())
        {
            int x = q1.del();

```

```

        if(!q1.isEmpty())
        {
            q2.insert(x);
        }

        else
        {
            cout<<x<<endl;
        }
    }
}

void func()
{
    q1.rear=-1;
    q1.front=-1;

    while(!q2.isEmpty())
    {
        q1.insert(q2.del());
    }

    q2.rear=-1;
    q2.front=-1;
}

};

int main()
{
    stack_using_queues s1;
    s1.push(20);
    s1.push(30);
    s1.push(40);
    cout<<"Displaying last element of the stack"<<endl;
    s1.pop();
}

```

```

        s1.func();
        s1.push(50);
        s1.push(60);
        cout<<"Again Displaying last element of the modified stack"<<endl;
        s1.pop();
        s1.func();

    return 0;
}

```

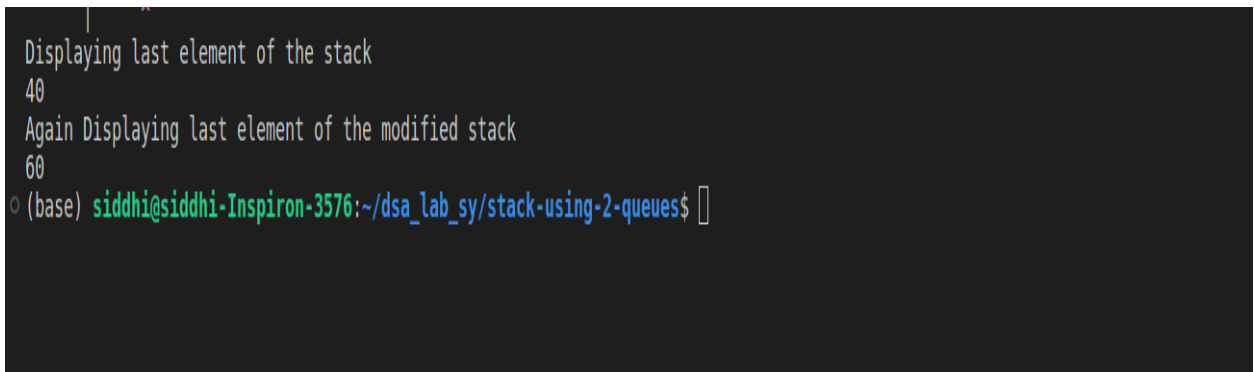
INPUT:

```

s1.push(20);
s1.push(30);
s1.push(40);
s1.pop();
s1.func();
s1.push(50);
s1.push(60);
s1.pop();
s1.func();

```

OUTPUT:



```

|
Displaying last element of the stack
40
Again Displaying last element of the modified stack
60
o (base) siddhi@siddhi-Inspiron-3576:~/dsa_lab_sy/stack-using-2-queues$

```