Implement Queue using Stacks

A Queue is defined by its property of **FIFO**, which means First in First Out, i.e the element which is added first is taken out first. This behaviour defines a queue, whereas data is actually stored in an **array** or a **list** in the background.

What we mean here is that no matter how and where the data is getting stored, if the first element added is the first element being removed and we have implementation of the functions `enqueue()` and `dequeue()` to enable this behaviour, we can say that we have implemented a Queue data structure.

In our previous tutorial, we used a simple **array** to store the data elements, but in this tutorial we will be using **Stack data structure** for storing the data.

While implementing a queue data structure using stacks, we will have to consider the natural behaviour of stack too, which is **First in Last Out**.

For performing **enqueue** we require only **one stack** as we can directly **push** data onto the stack, but to perform **dequeue** we will require **two Stacks**, because we need to follow queue's FIFO property and if we directly **pop** any data element out of Stack, it will follow LIFO approach(Last in First Out).

Implementation of Queue using Stacks

In all we will require two Stacks to implement a queue, we will call them `S1` and `S2`.

```
class Queue {

    public:

    Stack S1, S2;



    //declaring enqueue method

    void enqueue(int x);



    //declaring dequeue method

    int dequeue();

}
```
Copy

In the code above, we have simply defined a class `Queue`, with two variables `S1` and `S2` of type `Stack`.

We know that, Stack is a data structure, in which data can be added using `push()` method and data can be removed using `pop()` method.

You can find the code for `Stack` class in the Stack data structure tutorial.

To implement a queue, we can follow two approaches:

1. **By making the `enqueue` operation costly**
2. **By making the `dequeue` operation costly**

---

1. Making the Enqueue operation costly

In this approach, we make sure that the oldest element added to the queue stays at the **top** of the stack, the second oldest below it and so on.

To achieve this, we will need two stacks. Following steps will be involved while enqueuing a new element to the queue.

NOTE: First stack(`S1`) is the main stack being used to store the data, while the second stack(`S2`) is to assist and store data temporarily during various operations.

1. If the queue is empty(means `S1` is empty), directly **push** the first element onto the stack `S1`.
2. If the queue is not empty, move all the elements present in the first stack(`S1`) to the second stack(`S2`), one by one. Then add the new element to the first stack, then move back all the elements from the second stack back to the first stack.
3. Doing so will always maintain the right order of the elements in the stack, with the 1st data element staying always at the **top**, with 2nd data element right below it and the new data element will be added to the bottom.

This makes removing an element from the queue very simple, all we have to do is call the `pop()` method for stack `S1`.

---

2. Making the Dequeue operation costly

In this approach, we insert a new element onto the stack `S1` simply by calling the `push()` function, but doing so will will push our first element towards the bottom of the stack, as we insert more elements to the stack.

But we want the first element to be removed first. Hence in the `dequeue` operation, we will have to use the second stack `S2`.

We will have to follow the following steps for `dequeue` operation:

1. If the queue is empty(means `S1` is empty), then we return an error message saying, the queue is empty.
2. If the queue is not empty, move all the elements present in the first stack(`S1`) to the second stack(`S2`), one by one. Then remove the element at the **top** from the second stack, and then move back all the elements from the second stack to the first stack.
3. The purpose of moving all the elements present in the first stack to the second stack is to reverse the order of the elements, because of which the first element inserted into the queue, is positioned at the top of the second stack, and all we have to do is call the `pop()` function on the second stack to remove the element.

NOTE: *We will be implementing the **second approach**, where we will make the `dequeue()` method costly.*

Adding Data to Queue - enqueue()

As our Queue has a Stack for data storage instead of arrays, hence we will be adding data to Stack, which can be done using the push() method, therefore enqueue() method will look like:

```
void Queue :: enqueue(int x)

{

    S1.push(x);

}
```

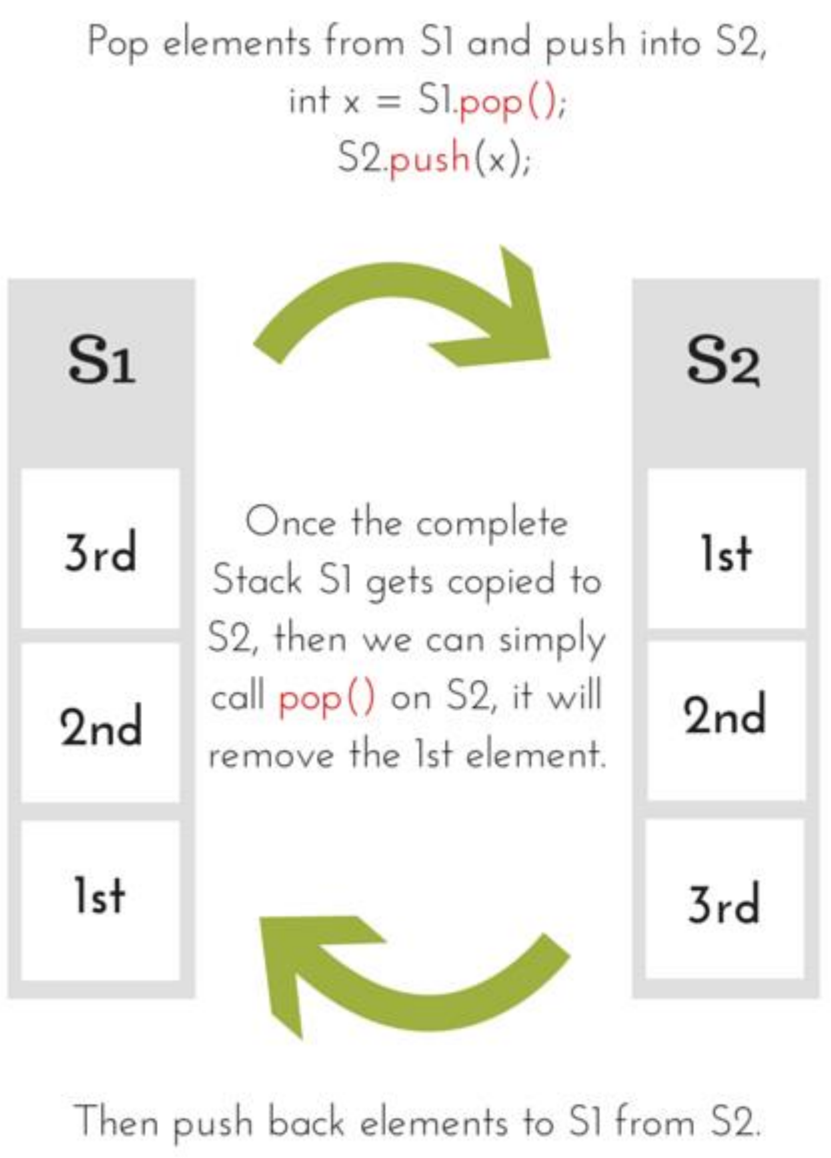
Copy

That's it, new data element is enqueued and stored in our queue.

Removing Data from Queue - dequeue()

When we say remove data from Queue, it always means taking out the element which was inserted first into the queue, then second and so on, as we have to follow the **FIFO approach**.

But if we simply perform S1.pop() in our **dequeue** method, then it will remove the Last element inserted into the queue first. So what to do now?


Pop elements from S1 and push into S2,
int x = S1.pop();
S2.push(x);
S1
3rd
2nd
1st
Once the complete Stack S1 gets copied to S2, then we can simply call pop() on S2, it will remove the 1st element.
S2
1st
2nd
3rd
Then push back elements to S1 from S2.

As you can see in the diagram above, we will move all the elements present in the first stack to the second stack, and then remove the **top** element, after that we will move back the elements to the first stack.

```
int Queue :: dequeue()

{

    int x, y;

    while(S1.isEmpty())

    {

        // take an element out of first stack

        x = S1.pop();

        // insert it into the second stack

        S2.push();

    }


    // removing the element

    y = S2.pop();


    // moving back the elements to the first stack

    while(!S2.isEmpty())

    {

        x = S2.pop();

        S1.push(x);

    }


    return y;

}
```

Copy

Now that we know the implementation of `enqueue()` and `dequeue()` operations, let's write a complete program to implement a queue using stacks.

Implementation in C++(OOPS)

We will not follow the traditional approach of using pointers, instead we will define proper classes, just like we did in the Stack tutorial.

```
/*  Below program is written in C++ language  */
```

```
# include<iostream>

using namespace std;

// implementing the stack class
class Stack
{
    int top;
    public:
    int a[10];  //Maximum size of Stack
    Stack()
    {
        top = -1;
    }

    // declaring all the function
    void push(int x);
    int pop();
    bool isEmpty();
};

// function to insert data into stack
void Stack::push(int x)
{
    if(top >= 10)
    {
        cout << "Stack Overflow \n";
    }
    else
```

```cpp
    {
        a[++top] = x;
        cout << "Element Inserted into Stack\n";
    }
}

// function to remove data from the top of the stack
int Stack::pop()
{
    if(top < 0)
    {
        cout << "Stack Underflow \n";
        return 0;
    }
    else
    {
        int d = a[top--];
        return d;
    }
}

// function to check if stack is empty
bool Stack::isEmpty()
{
    if(top < 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}

// implementing the queue class
class Queue {
    public:
    Stack S1, S2;

    //declaring enqueue method
    void enqueue(int x);

    //declaring dequeue method
    int dequeue();
};

// enqueue function
void Queue :: enqueue(int x)
{
    S1.push(x);
    cout << "Element Inserted into Queue\n";
}

// dequeue function
int Queue :: dequeue()
{
    int x, y;
    while(!S1.isEmpty())
    {
        // take an element out of first stack
        x = S1.pop();

        // insert it into the second stack

        S2.push(x);

    }


    // removing the element

    y = S2.pop();


    // moving back the elements to the first stack

    while(!S2.isEmpty())

    {

        x = S2.pop();

        S1.push(x);

    }


    return y;

}


// main function

int main()

{

    Queue q;

    q.enqueue(10);

    q.enqueue(100);

    q.enqueue(1000);

    cout << "Removing element from queue" << q.dequeue();


    return 0;

}
```
Copy