232. Implement Queue using Stacks

May 2, 2016 | 108.7K views

Average Rating: 4.33 (70 votes)

Implement the following operations of a queue using stacks.

- push(x) -- Push element x to the back of queue.
- pop() -- Removes the element from in front of queue.
- peek() -- Get the front element.
- empty() -- Return whether the queue is empty.

Example:

```
MyQueue queue = new MyQueue();

queue.push(1);
queue.push(2);
queue.peek();  // returns 1
queue.pop();   // returns 1
queue.empty(); // returns false
```

Notes:

- You must use *only* standard operations of a stack -- which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.
- Depending on your language, stack may not be supported natively. You may simulate a stack by using a list or deque (double-ended queue), as long as you use only standard operations of a stack.
- You may assume that all operations are valid (for example, no pop or peek operations will be called on an empty queue).

Summary

This article is for beginners. It introduces the following ideas: Queue, Stack.

Solution

Queue is **FIFO** (first in - first out) data structure, in which the elements are inserted from one side - `rear` and removed from the other - `front`. The most intuitive way to implement it is with linked lists, but this article will introduce another approach using stacks. Stack is **LIFO** (last in - first out) data structure, in which elements are added and removed from the same end, called `top`. To satisfy **FIFO** property of a queue we need to keep two stacks. They serve to reverse arrival order of the elements and one of them store the queue elements in their final order.

Approach #1 (Two Stacks) Push - $O(n)$ per operation, Pop - $O(1)$ per operation.

Algorithm

Push

A queue is FIFO (first-in-first-out) but a stack is LIFO (last-in-first-out). This means the newest element must be pushed to the bottom of the stack. To do so we first transfer all `s1` elements to auxiliary stack `s2`. Then the newly arrived element is pushed on top of `s2` and all its elements are popped and pushed to `s1`.

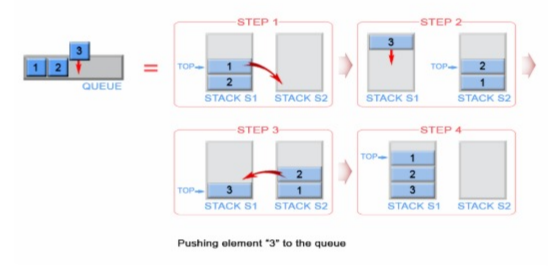

Figure 1. Push an element in queue

Java

```java
private int front;

public void push(int x) {
    if (s1.empty())
        front = x;
    while (!s1.isEmpty())
        s2.push(s1.pop());
    s2.push(x);
    while (!s2.isEmpty())
        s1.push(s2.pop());
}
```

Complexity Analysis

- Time complexity : $O(n)$.

Each element, with the exception of the newly arrived, is pushed and popped twice. The last inserted element is popped and pushed once. Therefore this gives $4n+2$ operations where n is the queue size. The `push` and `pop` operations have $O(1)$ time complexity.

- Space complexity : $O(n)$. We need additional memory to store the queue elements

Pop

The algorithm pops an element from the stack `s1`, because `s1` stores always on its top the first inserted element in the queue. The front element of the queue is kept as `front`.

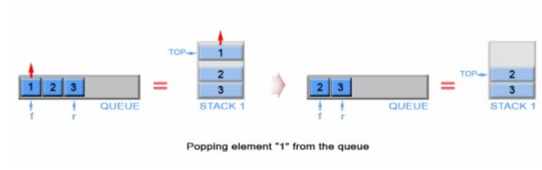

Figure 2. Pop an element from queue

Java

```java
// Removes the element from the front of queue.
public void pop() {
    s1.pop();
    if (!s1.empty())
        front = s1.peek();
}
```

Complexity Analysis

- Time complexity : $O(1)$.
- Space complexity : $O(1)$.

Empty

Stack `s1` contains all stack elements, so the algorithm checks `s1` size to return if the queue is empty.

```java
// Return whether the queue is empty.
public boolean empty() {
    return s1.isEmpty();
}
```

Time complexity : $O(1)$.

Space complexity : $O(1)$.

Peek

The `front` element is kept in constant memory and is modified when we push or pop an element.

```java
// Get the front element.
public int peek() {
  return front;
}
```

Time complexity : $O(1)$. The `front` element has been calculated in advance and only returned in `peek` operation.

Space complexity : $O(1)$.

Approach #2 (Two Stacks) Push - $O(1)$ per operation, Pop - Amortized $O(1)$ per operation.

Algorithm

Push

The newly arrived element is always added on top of stack `s1` and the first element is kept as `front` queue element

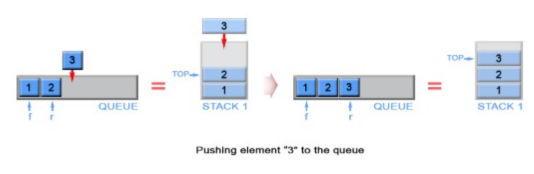

Figure 3. Push an element in queue

Java

```java
private Stack<Integer> s1 = new Stack<>();
private Stack<Integer> s2 = new Stack<>();

// Push element x to the back of queue.
public void push(int x) {
    if (s1.empty())
        front = x;
    s1.push(x);
}
```

Complexity Analysis

- Time complexity : $O(1)$.

Appending an element to a stack is an O(1) operation.

- Space complexity : $O(n)$. We need additional memory to store the queue elements

Pop

We have to remove element in front of the queue. This is the first inserted element in the stack `s1` and it is positioned at the bottom of the stack because of stack's `LIFO (last in - first out)` policy. To remove the bottom element from `s1`, we have to pop all elements from `s1` and to push them on to an additional stack `s2`, which helps us to store the elements of `s1` in reversed order. This way the bottom element of `s1` will be positioned on top of `s2` and we can simply pop it from stack `s2`. Once `s2` is empty, the algorithm transfer data from `s1` to `s2` again.

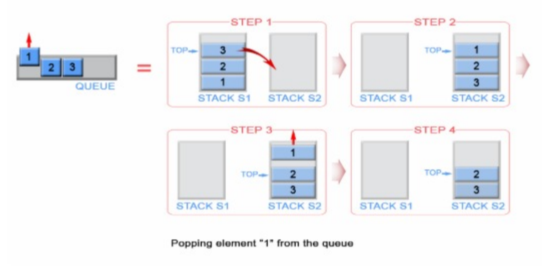

Figure 4. Pop an element from stack

Java

```java
// Removes the element from in front of queue.
public void pop() {
    if (s2.isEmpty()) {
        while (!s1.isEmpty())
            s2.push(s1.pop());
    }
    s2.pop();
}
```

Complexity Analysis

- Time complexity: Amortized $O(1)$, Worst-case $O(n)$.

In the worst case scenario when stack `s2` is empty, the algorithm pops n elements from stack s1 and pushes n elements to `s2`, where n is the queue size. This gives $2n$ operations, which is $O(n)$. But when stack `s2` is not empty the algorithm has $O(1)$ time complexity. So what does it mean by Amortized $O(1)$? Please see the next section on Amortized Analysis for more information.

- Space complexity : $O(1)$.

Amortized Analysis

Amortized analysis gives the average performance (over time) of each operation in the worst case. The basic idea is that a worst case operation can alter the state in such a way that the worst case cannot occur again for a long time, thus amortizing its cost.

Consider this example where we start with an empty queue with the following sequence of operations applied:

$$push_1, push_2, \ldots, push_n, pop_1, pop_2 \ldots, pop_n$$

The worst case time complexity of a single pop operation is $O(n)$. Since we have n pop operations, using the worst-case per operation analysis gives us a total of $O(n^2)$ time.

However, in a sequence of operations the worst case does not occur often in each operation - some operations may be cheap, some may be expensive. Therefore, a traditional worst-case per operation analysis can give overly pessimistic bound. For example, in a dynamic array only some inserts take a linear time, though others - a constant time.

In the example above, the number of times pop operation can be called is limited by the number of push operations before it. Although a single pop operation could be expensive, it is expensive only once per `n` times (queue size), when `s2` is empty and there is a need for data transfer between `s1` and `s2`. Hence the total time complexity of the sequence is : `n` (for push operations) + `2*n` (for first pop operation) + `n - 1` (for pop operations) which is $O(2*n)$.This gives $O(2n/2n)$ = $O(1)$ average time per operation.

Empty

Both stacks `s1` and `s2` contain all stack elements, so the algorithm checks `s1` and `s2` size to return if the queue is empty.

```java
// Return whether the queue is empty.
public boolean empty() {
    return s1.isEmpty() && s2.isEmpty();
}
```

Time complexity : $O(1)$.

Space complexity : $O(1)$.

Peek

The `front` element is kept in constant memory and is modified when we push an element. When `s2` is not empty, front element is positioned on the top of `s2`

```java
// Get the front element.
public int peek() {
    if (!s2.isEmpty()) {
            return s2.peek();
    }
    return front;
}
```

Time complexity : $O(1)$.

The `front` element was either previously calculated or returned as a top element of stack `s2`. Therefore complexity is $O(1)$

Space complexity : $O(1)$.

Analysis written by: @elmirap.

Rate this article:

Previous Next

Comments: 31 Sort By

Type comment here... (Markdown is supported)

Preview Post

sfdye 860 September 24, 2018 8:36 PM

Python 3 implementation for both approaches:

Approach 1: push `O(n)`, pop `O(1)`

```
class MyQueue:
```

Read More

28 Share Reply

SHOW 3 REPLIES

terrible_whiteboard 633 May 19, 2020 7:24 AM

I made a video if anyone is having trouble understanding the solution (clickable link)
https://youtu.be/BI3JzvNyV3o

Read More

22 Share Reply

SHOW 1 REPLY

k3000001 79 October 8, 2017 5:50 AM

Algo #2: I'm not seeing how this case will work if you push immediately after you pop().

21 Share Reply

SHOW 2 REPLIES

milanocookie 111 July 29, 2018 7:45 AM

Slightly modified code for solution #2. Made the push method more clean.

```
class MyQueue {
    private Stack<Integer> s1 = new Stack<>();
    private Stack<Integer> s2 = new Stack<>();
```

Read More

5 Share Reply

wanders 35 March 11, 2019 5:22 AM

I am not an expert on amortized analysis, but the analysis for `pop()` in Approach #2 relies on the sequence of operations given. If instead the sequence is `push()` n times, then alternate `pop()` and `push()` n times, each `pop()` operation still costs n operations, and hence the average cost is O(n).

2 Share Reply

SHOW 1 REPLY

gauravjsh127 2 January 23, 2018 4:31 AM

Python solution:

from Queue import LifoQueue

Read More

2 Share Reply

mkl75now 7 April 9, 2020 3:04 AM

For approach 2, pop() operation time complexity, why are we dividing as in O(2n/2n)?

1 Share Reply

_bella_zhao 3 February 19, 2019 3:22 PM

```
class MyQueue {
private:
    stack<int> s1,s2;
```

Read More

1 Share Reply

wqmbisheng 8 October 23, 2017 9:30 AM

As for time complexity analysis of approach 2, why is it O(2n/2n) ?
I think it might be O(4n/2n), though the final result is also O(1).

1 Share Reply

SHOW 4 REPLIES

Vijay0753 10 April 17, 2020 11:26 PM

Why use two stacks?

```
def __init__(self):
    """
    Initialize your data structure here.
```

Read More

0 Share Reply

SHOW 3 REPLIES

‹ 1 2 3 4 ›