Solution Review: Implement a Queue Using Stacks

In this lesson, we'll review two different approaches to solve the 'Implement a Queue Using Stacks' challenge.

We'll cover the following

Solution # 1: Two Stacks Working in `enqueue()`

main.py

Stack.py

```
1  # We can use 2 stacks for this purpose,main_stack to store original
2  # and temp_stack which will help in enqueue operation.
3  # Main thing is to put first entered element at the top of main stack
```

```
4
5  from Stack import MyStack
6
7  class newQueue:
8      # Can use size from argument to create stack
9      def __init__(self):
10         self.main_stack = MyStack()
11         self.temp_stack = MyStack()
12
13     # Inserts Element in the Queue
14     def enqueue(self, value):
15         # Push the value into main_stack in O(1)
16         if self.main_stack.is_empty() and self.temp_stack.is_empty():
17             self.main_stack.push(value)
18             print(str(value) + "init main enqueued")
19         else:
20             while not self.main_stack.is_empty():
21                 self.temp_stack.push(self.main_stack.pop())
22             # inserting the value in the queue
23             self.main_stack.push(value)
24             print(str(value) + "temp enqueued")
25             while not self.temp_stack.is_empty():
26                 self.main_stack.push(self.temp_stack.pop())
27
28     # Removes Element From Queue
```

In this approach, we are using two stacks. The `main_stack` stores the queue elements while the `temp_stack` acts as a temporary buffer to provide queue functionality.

We make sure that after every `enqueue` operation, the newly inserted value is at the bottom of the main stack. Before insertion, all the other elements are transferred to `temp_stack` and, naturally, their order is reversed. The new element is added into the empty `main_stack`. Finally, all the elements are pushed back into `main_stack` and `temp_stack` becomes empty.

The `dequeue` operation simply pops out the element at the top of `main_stack` that is the oldest element in the stack.

We can observe that the meat of the implementation lies in e ue i it a costly operation.

Time Complexity#

`enqueue()`

Whenever a value is enqueued, all the elements are transferred to `temp_stack` and then back to `main_stack`. Hence, for **n** elements in our queue, the runtime complexity of the `enqueue` operation is **O(n)**.

`dequeue()`

The `dequeue` operation takes **constant** time since it involves one pop of the stack.

Solution # 2: Two Stacks Working in `dequeue()`

main.py

Stack.py

```
#We can use 2 stacks for this purpose,main_stack to store original values
#and temp_stack which will help in enqueue operation.
#Main thing is to put first entered element at the top of main_stack

from Stack import MyStack

class NewQueue:

  # Can use size from argument to create stack
  def __init__(self):
    self.main_stack = MyStack()
    self.temp_stack = MyStack()
```

```
        # Inserts Element in the Queue
  def enqueue(self,value):
    # Push the value into main_stack in O(1)
    self.main_stack.push(value)
    print (str(value) + " enqueued")

        # Removes Element From Queue
  def dequeue(self):
                # If both stacks are empty, end operation
    if not self.temp_stack.is_empty():
        front = self.temp_stack.pop()
        print(str(front) + " dequeued")
        return front
    if self.temp_stack.is_empty() and self.main_stack.is_empty():
        return None
      # Transfer all elements to temp_stack
    while not self.main_stack.is_empty():
      self.temp_stack.push(self.main_stack.pop())
    # Pop the first value. This is the oldest element in the queue
    front = self.temp_stack.pop()
    print(str(front) + " dequeued")
    return front

if __name__ == "__main__" :
  queue = NewQueue()
  for i in range(5):
    queue.enqueue(i+1)
  print("----------")
  for i in range(5):
    queue.dequeue()
```

Here, we make `dequeue` the more expensive operation. `enqueue` works in constant time as we simply push the value into `main_stack`.

In `dequeue`, we first check if `temp_stack` is empty. If it is, then we move all the elements of `main_stack` to `temp_stack` (given that `main_stack` is not empty). This would bring the oldest element to the top of `temp_stack`. Now, all we have to do is pop off the top value.

However, if `temp_stack` was not empty at the beginning, then we would nc transfer any elements. The top value in `temp_stack` would simply be popped

off and returned.

Time Complexity#

`enqueue()`

In the second approach, `enqueue` operation takes **constant** time.

`dequeue()`

`dequeue` is **O(n)** if `temp_stack` is empty because in that case, we have to transfer all the elements to it. However, it takes *O(1)* as `temp_stack` is not empty. This solution is more efficient than the previous one because, each time, we perform one transfer instead of two, and sometimes we do not need to transfer at all.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. See how

Back

Challenge 4: Implement a Queue Usin...

Next

Challenge 5: Sort Values in a Stack

Completed

Report an Issue