Build a queue using a linked list

May 3, 2020

1 Build a queue using a linked list

By now, you may be noticing a pattern. Earlier, we had you implement a stack using an array and a linked list. Here, we're doing the same thing with queues: In the previous notebook, you implemented a queue using an array, and in this notebook we'll implement one using a linked list.

It's good to try implementing the same data structures in multiple ways. This helps you to better understand the abstract concepts behind the data structure, separate from the details of their implementation—and it also helps you develop a habit of comparing pros and cons of different implementations.

With both stack and queues, we saw that trying to use arrays introduced some concerns regarding the time complexity, particularly when the initial array size isn't large enough and we need to expand the array in order to add more items.

With our stack implementation, we saw that linked lists provided a way around this issue—and exactly the same thing is true with queues.

Walkthrough

1.1 1. Define a `Node` class

Since we'll be implementing a linked list for this, we know that we'll need a `Node` class like we used earlier in this lesson.

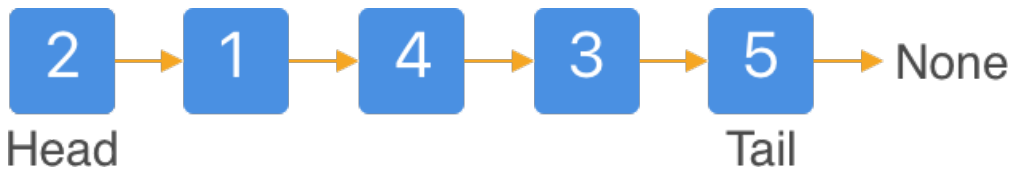
See if you can remember how to do this, and implement it in the cell below.

In [ ]:

Hide Solution

```
In [ ]: class Node:

            def __init__(self, value):
                self.value = value
                self.next = None
```



Linked list.

1.2 2. Create the `Queue` class and its `__init__` method

In the cell below, see if you can write the `__init__` method for our `Queue` class. It will need three attributes: * A `head` attribute to keep track of the first node in the linked list * A `tail` attribute to keep track of the last node in the linked list * A `num_elements` attribute to keep track of how many items are in the stack

In [ ]:

Hide Solution

```
In [ ]: class Queue:

            def __init__(self):
                self.head = None
                self.tail = None
                self.num_elements = 0
```

1.3 3. Add the `enqueue` method

In the cell below, see if you can figure out how to write the `enqueue` method.

Remember, the purpose of this method is to add a new item to the back of the queue. Since we're using a linked list, this is equivalent to creating a new node and adding it to the tail of the list.

Some things to keep in mind: * If the queue is empty, then both the `head` and `tail` should refer to the new node (because when there's only one node, this node is both the head and the tail) * Otherwise (if the queue has items), add the new node to the tail (i.e., to the end of the queue) * Be sure to shift the `tail` reference so that it refers to the new node (because it is the new tail)

```
In [2]: class Queue:

            def __init__(self):
                self.head = None
                self.tail = None
                self.num_elements = 0

            # TODO: Add the enqueue method
```

Hide Solution

```
In [ ]: class Queue:

            def __init__(self):
                self.head = None
                self.tail = None
                self.num_elements = 0

            def enqueue(self, value):
                new_node = Node(value)
```

```
            if self.head is None:
                self.head = new_node
                self.tail = self.head
            else:
                self.tail.next = new_node    # add data to the next attribute of the tail (i
                self.tail = self.tail.next   # shift the tail (i.e., the back of the queue)
            self.num_elements += 1
```

1.4 4. Add the `size` and `is_empty` methods

You've implemented these a couple of times now, and they'll work the same way here: * Add a `size` method that returns the current size of the stack * Add an `is_empty` method that returns `True` if the stack is empty and `False` otherwise

We'll make use of these methods in a moment when we write the `dequeue` method.

```
In [ ]: class Queue:

            def __init__(self):
                self.head = None
                self.tail = None
                self.num_elements = 0

            def enqueue(self, value):
                new_node = Node(value)
                if self.head is None:
                    self.head = new_node
                    self.tail = self.head
                else:
                    self.tail.next = new_node    # add data to the next attribute of the tail (i
                    self.tail = self.tail.next   # shift the tail (i.e., the back of the queue)
                self.num_elements += 1

            # TODO: Add the size method

            # TODO: Add the is_empty method
```

Hide Solution

```
In [ ]: class Queue:

            def __init__(self):
                self.head = None
                self.tail = None
                self.num_elements = 0

            def enqueue(self, value):
                new_node = Node(value)
                if self.head is None:
                    self.head = new_node
```

```
            self.tail = self.head
        else:
            self.tail.next = new_node    # add data to the next attribute of the tail (i
            self.tail = self.tail.next   # shift the tail (i.e., the back of the queue)
        self.num_elements += 1

    def size(self):
        return self.num_elements

    def is_empty(self):
        return self.num_elements == 0
```

1.5 5. Add the `dequeue` method

In the cell below, see if you can add the `deqeueue` method.

Here's what it should do: * If the queue is empty, it should simply return `None`. Otherwise... * Get the value from the front of the queue (i.e., the head of the linked list) * Shift the `head` over so that it refers to the next node * Update the `num_elements` attribute * Return the value that was dequeued

```
In [ ]: class Queue:

    def __init__(self):
        self.head = None
        self.tail = None
        self.num_elements = 0

    def enqueue(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
            self.tail = self.head
        else:
            self.tail.next = new_node    # add data to the next attribute of the tail (i
            self.tail = self.tail.next   # shift the tail (i.e., the back of the queue)
        self.num_elements += 1

    # Add the dequeue method

    def size(self):
        return self.num_elements

    def is_empty(self):
        return self.num_elements == 0
```

Hide Solution

```
In [ ]: class Queue:

            def __init__(self):
                self.head = None
                self.tail = None
                self.num_elements = 0

            def enqueue(self, value):
                new_node = Node(value)
                if self.head is None:
                    self.head = new_node
                    self.tail = self.head
                else:
                    self.tail.next = new_node    # add data to the next attribute of the tail (i
                    self.tail = self.tail.next   # shift the tail (i.e., the back of the queue)
                self.num_elements += 1

            def dequeue(self):
                if self.is_empty():
                    return None
                value = self.head.value          # copy the value to a local variable
                self.head = self.head.next       # shift the head (i.e., the front of the queue)
                self.num_elements -= 1
                return value

            def size(self):
                return self.num_elements

            def is_empty(self):
                return self.num_elements == 0
```

1.6 Test it!

Here's some code you can use to check if your implementation works:

```
In [ ]: # Setup
        q = Queue()
        q.enqueue(1)
        q.enqueue(2)
        q.enqueue(3)

        # Test size
        print ("Pass" if (q.size() == 3) else "Fail")

        # Test dequeue
        print ("Pass" if (q.dequeue() == 1) else "Fail")
```

```
# Test enqueue
q.enqueue(4)
print ("Pass" if (q.dequeue() == 2) else "Fail")
print ("Pass" if (q.dequeue() == 3) else "Fail")
print ("Pass" if (q.dequeue() == 4) else "Fail")
q.enqueue(5)
print ("Pass" if (q.size() == 1) else "Fail")
```

1.7 Time Complexity

So what's the time complexity of adding or removing things from our queue here?

Well, when we use `enqueue`, we simply create a new node and add it to the tail of the list. And when we `dequeue` an item, we simply get the value from the head of the list and then shift the `head` variable so that it refers to the next node over.

Both of these operations happen in constant time—that is, they have a time-complexity of O(1).