Creating a Queue with O(1) Operations

Learn how to create a queue with constant-time enqueue and dequeue operations.

Creating a Queue with O(1) Operations

NOTE

For this problem, familiarity with queues, hash tables, and linked lists would help.

Instructions

Create a queue data structure with `O(1)` insertion, deletion, and size calculation.

Queue

A queue (https://en.wikipedia.org/wiki/Queue_(abstract_data_type)) is a data structure that keeps track of data in the order in which it was entered. Items are inserted into the back of the queue and removed from the front of the queue.

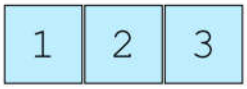
A real-world analogy would be a line at a grocery store. Everyone enters the line at the back. The first person who enters will be the first person served.

Common operations available to perform on a queue are:

- `enqueue`, or adding someone to the back of the queue
- `dequeue`, or removing someone from the front of the queue
- `size`, or checking the number of items in the queue

A common interview question is to create a queue that can perform each of these operations in constant-time. There are two common ways to do this.

Feel free to try it yourself first. There are many correct ways to complete this, so we won't have prepared tests for it.




```
class Queue {
    constructor() {
        // Your code here
    }

    enqueue(item) {
        // Your code here
    }

    dequeue() {
        // Your code here
    }

    get size() {
        // Your code here
        // Using a getter function is not required
    }
}
```

Solution 1 - Hash Table

This is the easier way to create a queue with constant-time operations. It's fairly straightforward. Let's see how we would do this.

In many JavaScript engines, an object is implemented as a hash table. A hash table has constant-time insertion and removal of items.

We'll use an object to keep track of the order of items in our queue. In many JavaScript engines, an object is implemented as a hash table. A hash table has constant-time insertion and removal of items.

By using an object, we're effectively storing our data in a hash table.

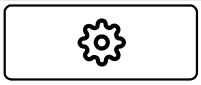
```
class Queue {
    constructor() {
        this._storage = {};
        this._firstIndex = 0;
        this._lastIndex = 0;
    }

    enqueue(item) {
        this._storage[this._lastIndex] = item;
        this._lastIndex++;
    }

    dequeue() {
        if(this.size === 0) {
            console.warn('Attempting to dequeue from an empty queue');
            return undefined;
        }

        const itemToReturn = this._storage[this._firstIndex];
        delete this._storage[this._firstIndex];
        this._firstIndex++;
        return itemToReturn;
    }

    get size() {
        return this._lastIndex - this._firstIndex;
    }
}
```



How it Works

`constructor`

We're creating a variable to store all of our data. This is an object, or hash table. We create two other variables to keep track of our start and end indexes.

`enqueue`

We insert an item into our object. The item's key will be `lastIndex`, which we then increment.

`dequeue`

We retrieve the item located at the first index and store it in a variable. We delete that item from our object and then return it.

`size`

This is a simple subtraction of the first index from the last index.

Time

Every method above has `O(1)` time complexity. We never loop through data. We perform a fixed, unchanging number of operations for each method call.

Space

Aside from data inserted into the queue, we track only the start and end index. This means that our queue requires `O(1)` memory in addition to the items passed in.

Solution 2 - Linked List

To implement our queue, we can also use a linked list. A linked list is characterized as having constant-time removal and addition at both the front and back of the list. This is exactly what we want in our queue.

In an interview, we'd generally be able to assume that we have a linked list available; the goal of this problem is to test that we can create a queue, not a linked list, so we can imagine that we have one. In that case, our queue will be extremely simple.

Assuming we have a linked list class with `size`, `addToTail`, and `addToHead` methods, we can create a queue class with the following code.


```
class Queue extends LinkedList {
    constructor() {
        super();
        this.enqueue = this.addToTail;
        this.dequeue = this.removeFromHead;
    }
}
```

That's really it. There's nothing else we need to do. The linked list has all of the functionality built-in and all we need to do is rename two methods.

Writing the Linked List Yourself

If you'd like to be ambitious, you can create the linked list yourself. Here's what it might look like.

```
        const newNode = {
            value: item,
            next: null
        };

        if(this._length === 0) {
```

```
            this.head = newNode;
        } else {
            this.tail.next = newNode;
        }

        this.tail = newNode;
        → this._length++;
    }

    removeFromHead() {
        if(this._length === 0) {
            console.warn('Attempting to remove from an empty list');
            return undefined;
        }

        const itemToReturn = this.head.value;
        this.head = this.head.next;
        → this._length--;
        return itemToReturn;
    }

    → get size() {
    →   →   return this._length;
    }
}
```

Time Complexity

Again, each method is `O(1)` as we don't loop through anything and we perform a fixed number of operations in each method.

Conclusion

If you are not very familiar with data structures, much of this lesson likely passed over your head. Try coming back to it after learning more about queues, stacks, and hash tables.

In the next lesson, we'll go over an even more restrictive way to create a queue, but one that Google has asked in an interview.

Back

Next

Balanced Brackets

Creating a Queue with Limited Data S...

Completed

Report an Issue