Stacks & Queues Exercises

Download our starter code <../dsa-stacks-queues.zip>.

Queues

Make a ***Queue*** class. It should include methods for enqueuing, dequeuing, peeking, and checking if the queue is empty.

Make it throw an error if you try to dequeue from an empty queue.

Stacks

Make a ***Stack*** class. It should include methods for pushing, popping, peeking, and checking if the stack is empty.

Make it throw an error if you try to pop from an empty stack.

Further Study: Composition

You've probably noticed that your ***Stack*** and ***Queue*** classes involve a lot of duplicate code compared to your ***LinkedList*** class. One way to avoid this problem is to use a ***LinkedList*** class internally to manage your stack or your queue:

```
class Queue {
  constructor() {
    this.size = 0;
    this.first = null;
    this.last = null;
    this._list = new LinkedList();
  }
}
```

Re-implement your classes by using a ***LinkedList*** internally to manage the data structure. Then make a new version using an array instead of a linked list. Which do you prefer?

Further Study Build: Deque

For extra practice with linked lists, build a deque using a doubly-linked list. Make sure it includes all of the expected methods for a deque.

Challenges

For these challenges, use either a stack or a queue (or a combination of both!)

Browser Back/Forward

Design how you could design a browser back/forward system using two stacks, so that you can visit a series of sites (Google, Yahoo, EBay, go back to Yahoo, then forward again to EBay, then onto Apple, and so on).

Write pseudo-code for this.

String Reversal

Write a function that reverses a string by handling one letter at a time. You cannot use an arrays, nor can you use any string-reversal built-in method.

Balanced Brackets?

Write a function that is passed a string which can contain any text, including different kinds of brackets: `{} [] ()`.

It should examine the string and decide if the string is "balanced" — a balanced string is one where the different kinds of brackets are properly balanced, such that you never close an bracket that isn't opened, is out of order, or end up with unclosed brackets.

Examples of balanced strings:

- `hello` *(no brackets)*
- `(hi) [there]`
- `(hi [there])`
- `(((hi)))`

Imbalanced:

- `(hello` *(bracket left open at end)*
- `(nope]` *(wrong type closed)*
- `((ok) [nope)]` *(closed out of order)*

Josephus Survivor

This is a classic algorithm problem, based on a Biblical-era tale.

Imagine a group of 10 people in a circle, numbered 1 to 10. If we started at the first person (#1) and killed every three people, it would look like this:

```
1  2  3  4  5  6  7  8  9  10
      !        !        !
```

This continues, though, looping around again, starting with where we left of at #10 (we'll mark the freshly-removed as red/! and the previously-removed in striked-out gray/X):

```
1  2  3  4  5  6  7  8  9  10
   !  X        X  !     X
```

And again, starting where that left off, at #8, and continuing:

```
1  2  3  4  5  6  7  8  9  10
!  X  X        X  X  !  X

1  2  3  4  5  6  7  8  9  10
X  X  X     !  X  X  X  X

1  2  3  4  5  6  7  8  9  10
X  X  X     X  X  X  X  X  !
```

At this point, only #4 remains, so that person would be our "survivor".

Write an algorithm that, given a number of people, and the "skip", which person will be the survivor.

For example:

```
find_survivor(10, 3) // 4
```

There are different ways you can solve this, but a good solution uses one of the structures covered in this exercise.

Want a hint on the data structure?

Linked List

You could solve this in other ways, but using a linked list (or a doubly-linked list) is often a good way to solve this problem. You can do so by making the list "circular"—having the last item in the linked list point back to the first item.

This will let you traverse the list, removing items until one remains.

Calculator

In this exercise, you'll build a "polish notation calculator".

Polish notation is a different way to write an artithmetic expression. For example, instead of writing ***1 + 2 * 3***, as we would in normal ("infix") style, we could write it with the operators to the left of their arguments. This expression would become ***+ 1 * 2 3***. You can read a polish notation expression backwards to see exactly what it does — in this case, multiply 2 times 3, and add that result to 1.

Let's try this out:

```
calc("+ 1 2")  # 1 + 2 == 3

calc("* 2 + 1 2")  # 2 * (1 + 2) == 6

calc("+ 9 * 2 3")   # 9 + (2 * 3) == 15
```

Let's make sure we have non-commutative operators (subtraction and division) working:

```
calc("- 1 2")  # 1 - 2 == -1

calc("- 9 * 2 3")  # 9 - (2 * 3) == 3

calc("/ 6 - 4 2")  # 6 / (4 - 2) == 3
```

Want a hint?

Data Structure

You'll want to turn the expression into a string and work through it *backwards*. Do this by hand and see how it feels.

Hacker Rack Challenges

Pick and work on challenges from HackerRank:

- Queues <https://www.hackerrank.com/domains/data-structures?filters%5Bsubdomains%5D%5B%5D=queues>
- Stacks <https://www.hackerrank.com/domains/data-structures?filters%5Bsubdomains%5D%5B%5D=stacks>

Solution

See Our solution <solution/index.html>.