3

Stacks and Queues

Questions on stacks and queues will be much easier to handle if you are comfortable with the ins and outs of the data structure. The problems can be quite tricky, though. While some problems may be slight modifications on the original data structure, others have much more complex challenges.

▸ Implementing a Stack

The stack data structure is precisely what it sounds like: a stack of data. In certain types of problems, it can be favorable to store data in a stack rather than in an array.

A stack uses LIFO (last-in first-out) ordering. That is, as in a stack of dinner plates, the most recent item added to the stack is the first item to be removed.

It uses the following operations:

- `pop()`: Remove the top item from the stack.
- `push(item)`: Add an item to the top of the stack.
- `peek()`: Return the top of the stack.
- `isEmpty()`: Return true if and only if the stack is empty.

Unlike an array, a stack does not offer constant-time access to the `i`th item. However, it does allow constant-time adds and removes, as it doesn't require shifting elements around.

We have provided simple sample code to implement a stack. Note that a stack can also be implemented using a linked list, if items were added and removed from the same side.

```
public class MyStack<T> {
  private static class StackNode<T> {
    private T data;
    private StackNode<T> next;

    public StackNode(T data) {
      this.data = data;
    }
  }

  private StackNode<T> top;

  public T pop() {
    if (top == null) throw new EmptyStackException();
    T item = top.data;
```

```
      top = top.next;
      return item;
   }

   public void push(T item) {
      StackNode<T> t = new StackNode<T>(item);
      t.next = top;
      top = t;
   }

   public T peek() {
      if (top == null) throw new EmptyStackException();
      return top.data;
   }

   public boolean isEmpty() {
      return top == null;
   }
}
```

One case where stacks are often useful is in certain recursive algorithms. Sometimes you need to push temporary data onto a stack as you recurse, but then remove them as you backtrack (for example, because the recursive check failed). A stack offers an intuitive way to do this.

A stack can also be used to implement a recursive algorithm iteratively. (This is a good exercise! Take a simple recursive algorithm and implement it iteratively.)

▸ Implementing a Queue

A queue implements FIFO (first-in first-out) ordering. As in a line or queue at a ticket stand, items are removed from the data structure in the same order that they are added.

It uses the operations:

- `add(item)`: Add an item to the end of the list.
- `remove()`: Remove the first item in the list.
- `peek()`: Return the top of the queue.
- `isEmpty()`: Return true if and only if the queue is empty.

A queue can also be implemented with a linked list. In fact, they are essentially the same thing, as long as items are added and removed from opposite sides.

```
public class MyQueue<T> {
   private static class QueueNode<T> {
      private T data;
      private QueueNode<T> next;

      public QueueNode(T data) {
         this.data = data;
      }
   }

   private QueueNode<T> first;
   private QueueNode<T> last;

   public void add(T item) {
        QueueNode<T> t = new QueueNode<T>(item);
        if (last != null) {
           last.next = t;
        }
        last = t;
        if (first == null) {
           first = last;
        }
     }

     public T remove() {
        if (first == null) throw new NoSuchElementException();
        T data = first.data;
        first = first.next;
        if (first == null) {
           last = null;
        }
        return data;
     }

     public T peek() {
        if (first == null) throw new NoSuchElementException();
        return first.data;
     }

     public boolean isEmpty() {
        return first == null;
     }
  }
```

It is especially easy to mess up the updating of the first and last nodes in a queue. Be sure to double check this.

One place where queues are often used is in breadth-first search or in implementing a cache.

In breadth-first search, for example, we used a queue to store a list of the nodes that we need to process. Each time we process a node, we add its adjacent nodes to the back of the queue. This allows us to process nodes in the order in which they are viewed.

Interview Questions

3.1 **Three in One:** Describe how you could use a single array to implement three stacks.

Hints: #2, #12, #38, #58

pg 227

3.2 **Stack Min:** How would you design a stack which, in addition to `push` and `pop`, has a function `min` which returns the minimum element? `Push`, `pop` and `min` should all operate in `O(1)` time.

Hints: #27, #59, #78

pg 232

3.3 **Stack of Plates:** Imagine a (literal) stack of plates. If the stack gets too high, it might topple. Therefore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure `SetOfStacks` that mimics this. `SetOfStacks` should be composed of several stacks and should create a new stack once the previous one exceeds capacity. `SetOfStacks.push()` and `SetOfStacks.pop()` should behave identically to a single stack (that is, `pop()` should return the same values as it would if there were just a single stack).

FOLLOW UP

Implement a function `popAt(int index)` which performs a pop operation on a specific sub-stack.

Hints: #64, #81

pg 233

3.4 **Queue via Stacks:** Implement a `MyQueue` class which implements a queue using two stacks.

Hints: #98, #114

pg 236

3.5 **Sort Stack:** Write a program to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure (such as an array). The stack supports the following operations: `push`, `pop`, `peek`, and `isEmpty`.

Hints: #15, #32, #43

pg 237

3.6 **Animal Shelter:** An animal shelter, which holds only dogs and cats, operates on a strictly "first in, first out" basis. People must adopt either the "oldest" (based on arrival time) of all animals at the shelter, or they can select whether they would prefer a dog or a cat (and will receive the oldest animal of that type). They cannot select which specific animal they would like. Create the data structures to maintain this system and implement operations such as `enqueue`, `dequeueAny`, `dequeueDog`, and `dequeueCat`. You may use the built-in `LinkedList` data structure.

Hints: #22, #56, #63

pg 239

Additional Questions: Linked Lists (#2.6), Moderate Problems (#16.26), Hard Problems (#17.9).

Hints start on page 653.