Prepared by Linan Qiu <lq2137@columbia.edu>

Stacks

Whenever you think of stacks, think of this:

Figure 1: A stack of pancakes

Have you ever tried eating a bottom pancake without going through the top ones? If you do, you're a monster and a hazard to all order and harmony. Besides, have you ever seen IHOP top up pancakes (yes they do that I've tried) from the bottom instead of at the top?

Since the laws of the universe state that you shall eat pancakes in an orderly and respectful manner, you consume only the top ones first and work your way towards the bottom. Similarly, if new pancake comes in, it *stacks* up nicely.

This is exactly what a stack is. It's a. . . stack. Piling onto the stack is called `push` and eating the top pancake is called `pop`.

Implementation-wise, we need something that does this:

```
Start with an empty stack.
```

```
push(A)
[A]
push(B)
[A][B]
push(C)
[A][B][C]
pop()
[A][B] return C
pop()
[A] return B
push(D)
[A][D]
```

Pretty intuitive huh?

Stack is an **abstract data type**. Simply specifying that something should support `push` and `pop` in a *first-in-last-out* (the first pancake to be added to your plate is the last to be eaten) is not really an implementation detail. So let's make Stack an interface:

`AwsmStack`

```
// AwsmStack.java

public interface AwsmStack<T> {

  public void push(T item);

  public T pop();

  public int size();
}
```

This should be pretty no-brainer.

`AwsmLinkedList` as an Implementation of `AwsmStack`

Let's see what we can use to implement a stack.

Turns out Lists (remember that ADT List from the previous chapter?) are perfect candidates for implementing Stacks. We can use `addFirst` to simulate a `push` and `removeFirst` to simulate a `pop`. After all, we can translate the `push` `pop` operations illustrated at the start of this chapter into:

```
Start with an empty stack.

addFirst(A)
[head][A]
addFirst(B)
[head][B][A]
addFirst(C)
[head][C][B][A]
pop()
[head][B][A] return C
pop()
[head][A] return B
push(D)
[head][D][A]
```

Think of the `head` of the linked list as a "cap" on the stack.

This in fact works really well since `addFirst` and `removeFirst` are both $O(1)$, making linked lists the ideal choice for implementing stacks.

We can make a class called `AwsmLinkedStack` to do this:

```java
// AwsmLinkedStack.java

public class AwsmLinkedStack<T> implements AwsmStack<T> {
  private AwsmLinkedList<T> list;

  public AwsmLinkedStack() {
    list = new AwsmLinkedList<>();
  }

  @Override
  public void push(T item) {
    list.addFirst(item);
  }

  @Override
  public T pop() {
    T data = list.getFirst();
    list.removeFirst();
    return data;
  }

  @Override
  public int size() {
    return list.size();
```

```
  }
}
```

Now you may be tempted to use `addLast` and `removeLast`. After all, it makes more intuitive sense to be growing the "butt" of the linked list right? However, while `addLast` can be made $O(1)$ though it is $O(N)$ in the naive implementation, `removeLast` is always $O(N)$. You'd end up with a stack that is $O(N)$ `push` and $O(N)$ `pop`, which is weak as hell.

`AwsmArrayList` as an Implementation of `AwsmStack`

Now before you thought of linked lists, you probably thought of using arrays for a stack. Why not have a pointer (say `size`) that increments as you add more elements? Then every time the array is full, we can simply duplicate the array and... oh wait. This is just our `AwsmArrayList`! And yes you're right. We can use the `AwsmArrayList` to implement a stack.

```java
// AwsmArrayStack.java

public class AwsmArrayStack<T> implements AwsmStack<T> {

  private AwsmArrayList<T> list;

  public AwsmArrayStack() {
    list = new AwsmArrayList<>();
  }

  @Override
  public void push(T item) {
    list.addLast(item);
  }

  @Override
  public T pop() {
    T data = list.getFirst();
    list.removeLast();
    return data;
  }

  @Override
  public int size() {
    return list.size();
  }
}
```

However, this time we use `addLast` and `removeLast` for `AwsmArrayList`. Why? You can answer this. *hint: think of the runtimes for* `addFirst` *and* `removeFirst`*. Remember what happens when we add elements at the front of an array list? We have to shift everything down right? What's the runtime for that? And what's the runtime for* `addLast` *and* `removeLast` This allows us to achieve $O(1)$ runtime for both `push` and `pop`.

In this case, we don't even have to worry about duplicating arrays when the array is full.

This is a great demonstration of how we can recycle previous work. In Columbia, many classes will count this as self-plagiarism. In data structures, we consider this best practice. Do not quote me out of context on this. Let me explain: Reusing previous code / data structures is key to making fewer errors / doing more.