Lab 4: Array-Based Stack and Queue

Joshua Fox, Laura Cruz Castro, Diego Aguilar, Cameron Brown

Spring 2024

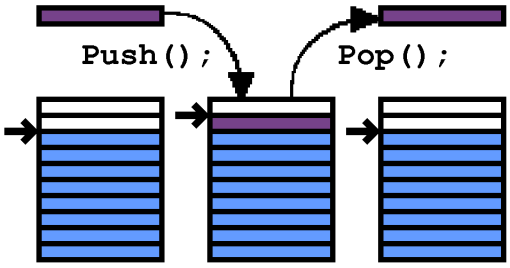
Contents

1 Stack Behavior

Stacks have two basic operations (with many other functions to accomplish these tasks):

- **Push** - Add something to the top of the stack.
- **Pop** - Remove something from the top of the stack and return it.

Figure 1.1: Example of LIFO operations-the data most recently added is the first to be removed

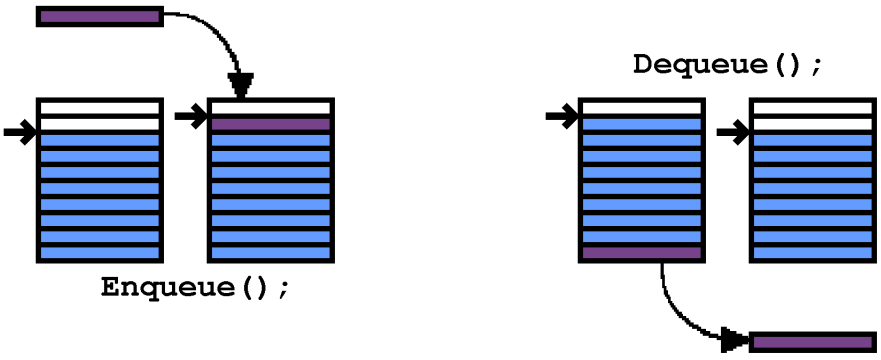


2 Queue Behavior

Like a stack, a queue has two basic operations:

- **Enqueue** - Add something to end of the queue. If this were a line, a new person getting into the line would start at the back.
- **Dequeue** - Remove something from the front of the queue. If this were a line, the person at the start of the line is next.

Figure 2.1: Example of FIFO operations-the newest data is last to be removed



3 Description

Your ABS and ABQ will be **template** classes, and thus will be able to hold any data type. (Many data structures follow this convention—reuse code whenever you can!) Because these classes will be using **dynamic memory**, you must be sure to define The Big Three:

- Copy Constructor
- Copy Assignment Operator
- Destructor

Data will be stored using a **dynamically allocated array** (hence the array-based stack and queue). You may use any other variables/function in your class to make implementation easier.

The nature of containers like these is that they are always changing size. You have 3 elements in a stack, and push() another... now you need space for 4 elements. Use push() to add another, now you need space for 5, etc... If your container is full, you can increase the size by an amount other than one, if you want.

Why increase (or decrease) the size by any amount other than one?

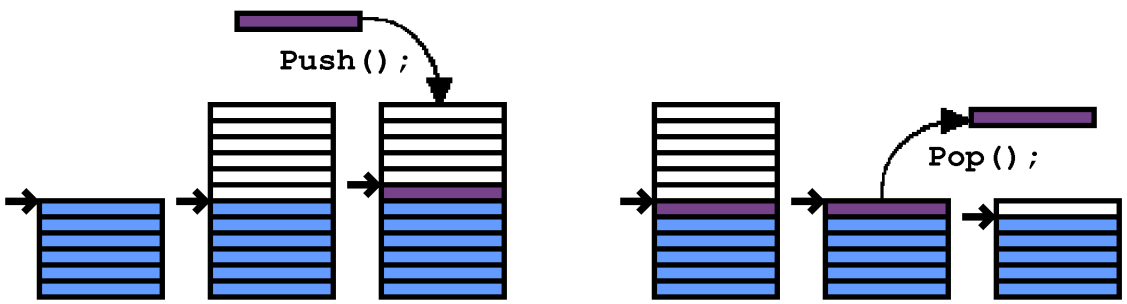
Short answer: performance!

If you are increasing or decreasing the size of a container, it's reasonable to assume that you will want to increase or decrease the size again at some point, requiring another round of allocate, copy, delete, etc.
Increasing the capacity by more than you might need (right now) or waiting to reduce the total capacity allows you to avoid costly dynamic allocations, which can improve performance-especially in situations in which this resizing happens **frequently**. This tradeoff to this approach is that it will use more memory, but this speed-versus-memory conflict is something that programmers have been dealing with for a long time.

By default, your ABS and ABQ will have a **scale factor** 2.0f-store this as a class variable.

1. Attempting to push() or enqueue() an item onto an ABS/ABQ that is full will resize the current capacity to current_capacity*scale_factor.
2. When calling pop() or dequeue(), if the "percent full" (e.g. current size / max capacity) becomes **strictly less** than 1/scale_factor, resize the storage array to current_capacity/scale_factor. **(Note that we are talking about fractions: Integer division is not percise enough!!!)**

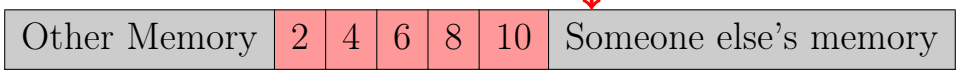
Figure 3.1: An example of the resizing scheme to be implement on a stack.



Resizing arrays What's easy to say isn't usually easy to do in programming. You can't "just" change the size of an array. You have to:

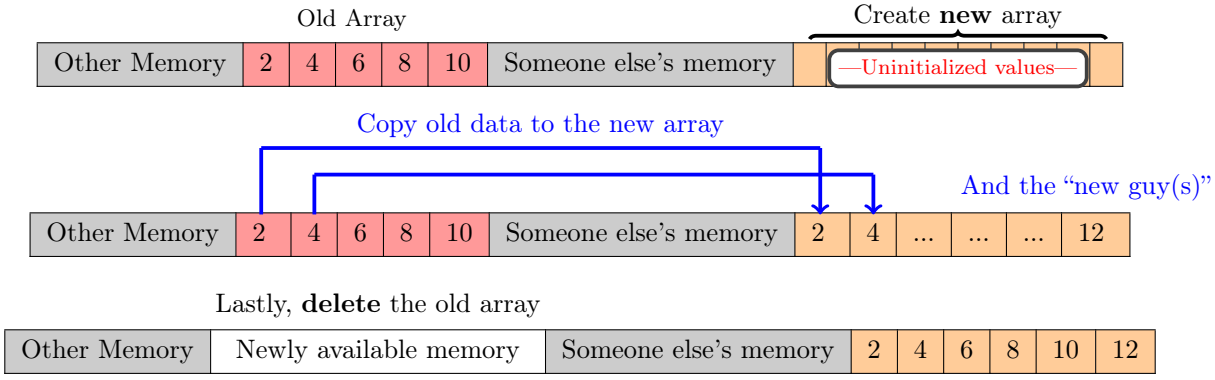
1. Create a new array based on the desired size
2. Copy elements from the old array to the new array (up to the size of the old array, or the capacity of the new array, **WHICHEVER IS SMALLER**).
3. If you were adding something to the array, copy that as well
4. Delete the old array
5. Redirect the pointer to the old array to the new array

```
1 someArray[5] = 12; // bad idea
```



In this scenario, in order to store one more element you would have to:

1. Create another array that was large enough to store all of the old elements plus the new one
2. Copy over all of the data elements one at a time (including the new element, at the end)
3. Free up the old array—no point in having two copies of the data



4 Exceptions

Some of your functions will **throw** exceptions. There are many types of exceptions that can be thrown, but in this case you will simply throw errors of type **runtime_error**. This is a general purpose error to indicate that something went wrong. The basic syntax for throwing an error is simply:

```
1 throw type_of_exception("Message describing the error.");
```

If you wanted to throw a runtime_error exception that said "An error has occurred." you would write:

```
1 throw runtime_error("An error has occurred.");
```

There is also the concept of using try/catch blocks, but for this assignment you will only have to **throw** the exceptions. Checking for such exceptions will be handled by various unit tests on Codio.