# Lab 3 – Array-Based Stack and Queue

## Overview

In this assignment, you will be implementing your own *Array-Based Stack* (ABS) and *Array-Based Queue (ABQ)*. A **stack** is a linear data structure which follows the Last-In, First-Out (LIFO) property. LIFO means that the data most recently added is the first data to be removed. (Imagine a stack of books, or a stack of papers on a desk—the first one to be removed is the last one placed on top.)

A **queue** is another linear data structure that follows the First-In, First-Out (FIFO) property. FIFO means that the data added first is the first to be removed (like a line in a grocery store—the first person in line is the first to checkout).

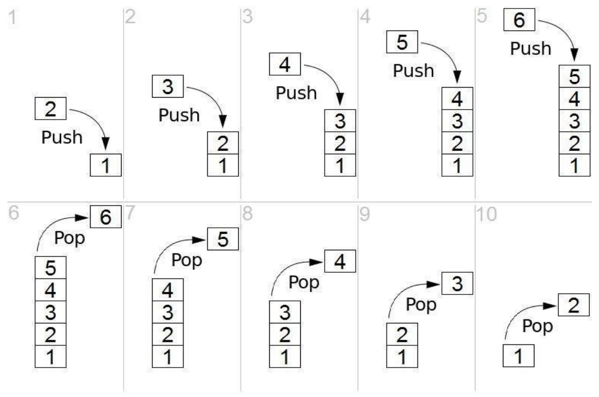
Both of these are data structures—some sort of class to store information. What makes them different is how they store and access the information. Adding the same values (say, the numbers 1-10) to either data structure would result in different ordering/output of the information. Why you might choose one or the other depends on the specific needs of a program.

## New keywords/language concepts

- Templates – write code to work with multiple data types
- The Big Three – Copy Constructor, Copy Assignment Operator, and Destructor – necessary when working with dynamic memory
- Exceptions – Used to indicate something went wrong in an application. Unhandled exceptions will cause a program to terminate
- Deep copy – Creating a copy of dynamically allocated memory requires new memory to be allocated before copying values

## Stack Behavior

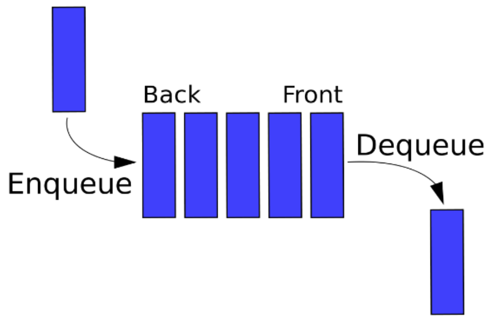
Stacks have two basic operations (with many other functions to accomplish these tasks):
**Push** – Add something to the top of the stack.
**Pop** – Remove something from the top of the stack and return it.



**Example of LIFO operations-the data most recently added is the first to be removed**

## 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—for whatever the people are lining up for.



**Example of FIFO operations-the newest data is last to be removed**

## 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?

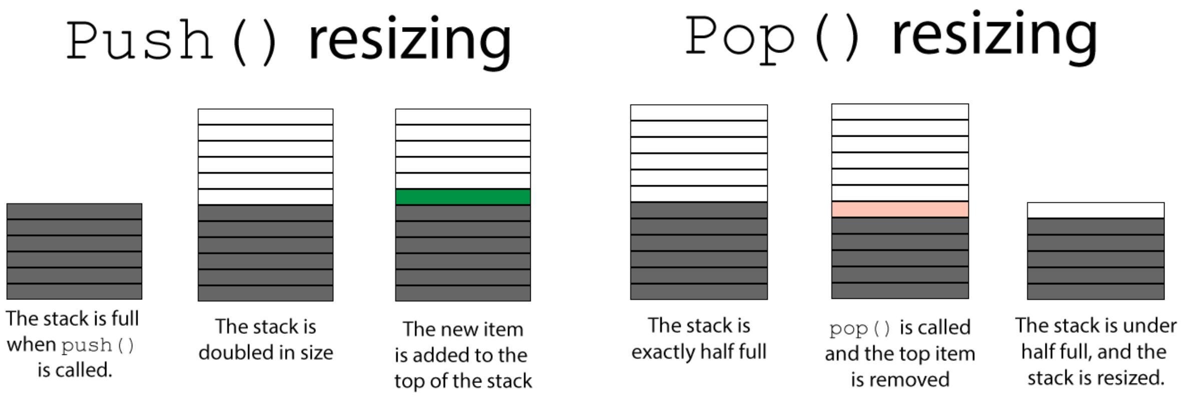
Short answer: performance!

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.


Push() resizing
Pop() resizing
The stack is full when push() is called.
The stack is doubled in size
The new item is added to the top of the stack
The stack is exactly half full
pop() is called and the top item is removed
The stack is under half full, and the stack is resized.

**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

## 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:

```
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:

```
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 zyBooks.

## Stack Functions

Your ABS must support the following methods:

| **Method** | **Description** |
|---|---|
| `ABS()` | Default constructor. Maximum capacity should be set to 1, and current size set to 0; |
| `ABS(int capacity)` | Constructor for an ABS with the specified starting maximum capacity. |
| `ABS(const ABS &d)` | Copy Constructor |
| `ABS &operator=(const ABS &d)` | Assignment operator. |
| `~ABS()` | Destructor |
| `void push(T data)` | Add a new item to the top of the stack and resize if necessary. |
| `T pop()` | Remove the item at the top of the stack, resizes if necessary, and return the value removed.<br>Throws a **runtime_error** if the stack is empty. |
| `T peek()` | Return the value of the item at the top of the stack, without removing it.<br>Throws a **runtime_error** if the stack is empty. |
| `unsigned int getSize()` | Returns the current number of items in the ABS. |
| `unsigned int getMaxCapacity()` | Returns the current max capacity of the ABS. |
| `T* getData()` | Returns the array representing the stack. |

Additional methods may be added as deemed necessary.

## Queue Functions

Your ABQ must support the following functions

| **Method** | **Description** |
|---|---|
| `ABQ()` | Default constructor. Maximum capacity should be set to 1, and current size set to 0; |
| `ABQ(int capacity)` | Constructor for an ABQ with the specified starting maximum capacity. |
| `ABQ(const ABS &d)` | Copy Constructor |
| `ABQ &operator=(const ABQ &d)` | Assignment operator. |
| `~ABQ()` | Destructor |
| `void enqueue(T data)` | Add a new item to end of the queue and resizes if necessary. |
| `T dequeue()` | Remove the item at front of the queue, resizes if necessary, and return the value removed.<br>Throws a **runtime_error** if the queue is empty. |
| `T peek()` | Return the value of the item at the front of the queue, without removing it.<br>Throws a **runtime_error** if the queue is empty. |
| `unsigned int getSize()` | Returns the current number of items in the ABQ. |
| `unsigned int getMaxCapacity()` | Returns the current max capacity of the ABQ. |
| `T* getData()` | Returns the array representing the queue. |

Additional methods may be added as deemed necessary.