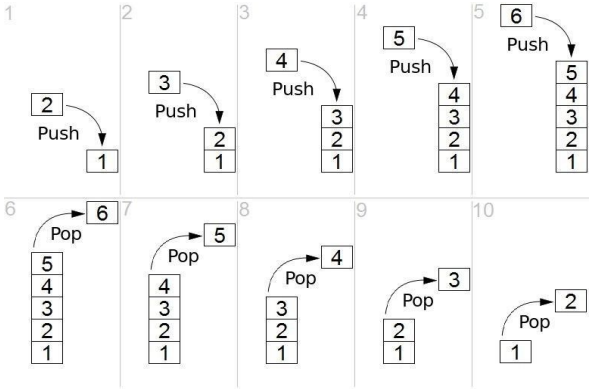
Lab 5 – Array-Based Stack and Queue

Overview

In this assignment, you will be implementing your own *Array-Based Stack* (ABS) and *Array-Based Queue*. 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).

Stack Behavior

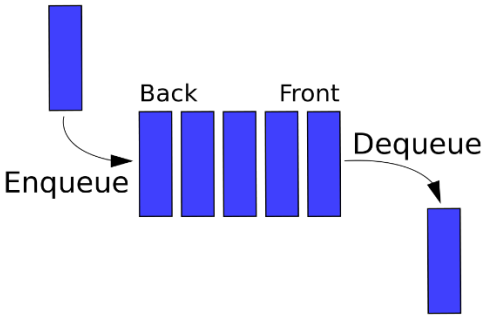
Push – Add something to the top of the stack.
Pop – Remove something from the top of the stack and return it,



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

Queue Behavior

Enqueue – Add something to end of the queue.
Dequeue – Remove something from the front of the queue.



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!) As with previous classes that use dynamic memory, you must be sure to define The Big Three: The Copy Constructor, the Assignment Operator, and the 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.

By default, your ABS and ABQ will have a scale factor 2.0f.

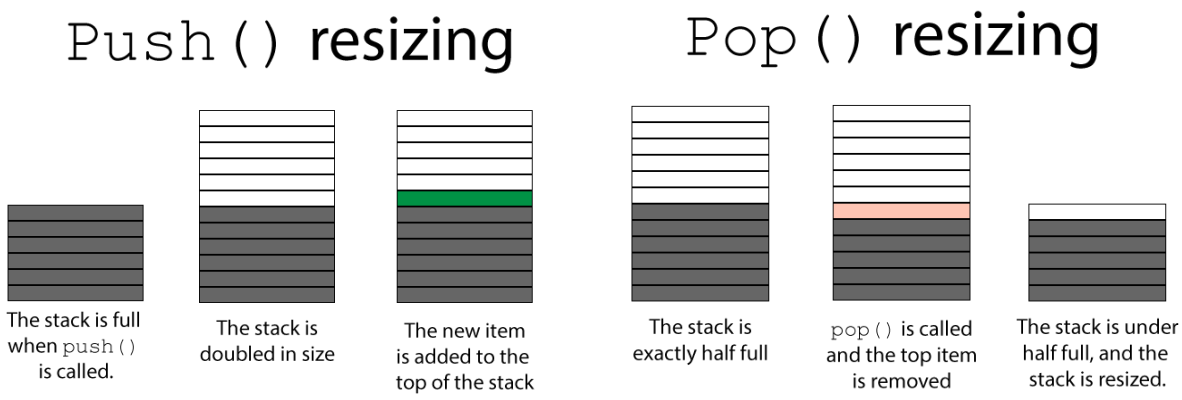
1. Attempting to push() or enqueue() an item onto an ABS/ABQ that is full will resize the current capacity to current_capacity*scale_factor.
2. When calling pop() or dequeue(), if the "percent full" (e.g. current size / max capacity) becomes **strictly less** than 1/scale_factor, resized the storage array to current_capacity/scale_factor.

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.



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

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. Throws -1 if the stack is empty.
`T peek()`	Return the value of the item at the top of the stack, without removing it. Throws -1 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.

Addition 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. Throws -1 if the queue is empty.
`T peek()`	Return the value of the item at the front of the queue, without removing it. Throws -1 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.

Addition methods may be added as deemed necessary.

S

Extra Credit (1%)

You can earn up to 1% extra credit (toward your final course grade) for this lab by doing everything listed below. DO NOT work on extra credit until your standard lab scored full points. If your standard lab does not score full points, you cannot earn any extra credit.

Update ABS/ABQ

Update your ABS and ABQ to use a scale factor other than 2.0f, and modify any methods that should behave differently as a result:

- ABS(int capacity, float scale_factor) : Constructor for an ABS with the specified starting capacity, and a custom scale factor.
- ABQ(int capacity, float scale_factor) : Constructor for an ABQ with the specified starting capacity, and a custom scale factor.

Add an extra attribute to your ABS and ABQ, total_resizes, which is a count of how many times the ABS/ABQ has been resized. Add an accessor for this attribute:

- unsigned int getTotalResizes(): Returns the total number of times the ABS has been resized.

Analysis Report

Create a testing file that can perform the following tasks, each with an ABS/ABQ that starts with a max capacity of 2:

- push *N* items onto an empty ABS.
- pop *N* items off an ABS with *N* items already on it.
- enqueue *N* items onto an empty ABQ.
- dequeue *N* items off an ABQ with *N* items already on it.

For each of the tasks, record:

- How long it takes to perform the task
    - Use the <chrono> or <ctime> libraries
- How many resizes were performed during the task

Do the tasks for each of the following possible combinations of Scale Factor and N*:

Scale Factor	N
1.5	10 000 000
2.0	30 000 000
3.0	50 000 000
10.0	75 000 000
100.0	100 000 000

Depending on your computer specs, a significantly high N may take a very long time or crash the program. If necessary, you may pick 5 different values of N but be sure to vary their size decently to get the most interesting results.

You should have 100 different sets of data (4 tasks * 5 scale factors * 5 Ns).

Graph the data for each scale factor, with N being the independent variable and time being the dependent variable. Include the number of resizes for each task somewhere on the graph as well.

You should have 10 graphs, one for each scale factor per ABS/ABQ.

Write a 1-2 page analysis of your results. Make note of any trends in the data. Answer the following questions in your report.

- How does N affect the time it takes for?
- What are the effects of changing scale factor?
- How do both of these affect the number of times the ABS will be resized?
- What seems to be the best scale factor, and why?
- How can you explain differences between the performances of your ABS and ABQ?

Submission (on Canvas):

- A **well organized** PDF containing relevant graphs, and your analysis.
- Your updated ABS.h file