PA 2 Stacks and Queues

Due: Friday, March 1 at 11:59 PM

> ⚠ **GradeScope submission**
>
> We will be grading this PA using GradeScope. Our basic test cases will be run against your code every time you submit, so we encourage you to submit early and often! Code which does not compile will not be given any credit. Basic test cases will be available within a few days after the PA release. We will add the grading test cases sometime during the reading week.

Goals and Overview

In this PA you will:

- learn about Stacks, Queues, and Deques
- learn about depth-first-search (DFS) and breadth-first-search (BFS) traversal
- learn about some more advanced C++ constucts, including basic templates and inheritance

Getting the Given Code

Download the source files from pa2.zip, and follow the procedures you learned in lab to move them to your home directory on the remote linux machines.

Part 1: The `Stack`, `Queue`, and `Deque` Classes

> ⚠ **Template Compilation**
>
> Note that the `stack.h`, `queue.h`, and `deque.h` files we've given you include the `.cpp` files at the end, so you should NOT use a `#include "XXX.h"` at the top of the `XXX.cpp` file. We have done it this way so that the template types are instantiated appropriately.

The `Deque` Class

You will write a class named Deque which is a modification of a doubly ended queue structure. While the standard deque allows for insertion and removal at both ends of a contiguous arrangement of data, our deque will allow insertion and removal at one end, but only removal at the other. For convenience, we will refer to the removal-only end as the left end, and the other as the right.

Your deque should be implemented as follows: The underlying data structure will be a C++ standard vector. Following the convention of the queue we discussed in class, you will allow the data to "float" in the vector, with the following difference: If, upon a removal, you discover that the contiguous block of data (whose size is, say, k) will "fit" in the first k positions of the vector, then you should resize down by making a new vector and copying the k pieces of data into that new vector using the `push_back` function. Additions to the structure can use the standard vector functions, and can only occur at the "right" side of the contiguous data (i.e. the position of largest index). (Note that in this implementation the structure is not "circular"– we do not wrap the data using the modulo of the array size.)

The `Stack` Class

You will write a class named `Stack` that works just like the stack you heard about in lecture, with the addition of the `peek()` function. It is declared in the given file `stack.h`. You will implement it in `stack.cpp`.

Your `Stack` class must implement all of the methods mentioned in the given code. Please read the documentation in the header file to see what limitations we have placed on your `Stack` class and what the running times of each function should be.

The `Queue` Class

You will write a class named Queue that works just like the queue you heard about in lecture, with the addition of the `peek()` function. It is declared in the given file `queue.h`. You will implement it in `queue.cpp`.

Your `Queue` class must implement all of the methods mentioned in the given code. Please read the documentation in the header file to see what limitations we have placed on your `Queue` class and what the running times of each function should be.

Testing

We have provided a file `testStackQueue.cpp` which includes a few (albeit trivial) test cases that your code should pass if it is correct. To compile this executable, type:

```
make testStackQueue
```
TERMINAL

These test cases are deliberately insufficient. We encourage you to augment this file with additional test cases, using the provided ones as examples. In particular, we have not provided test cases for your deque class. You will want to add those yourself!

Grading Information for Part 1

The following files are used to grade the stack and queue classes:

- `deque.cpp`
- `stack.cpp`
- `queue.cpp`
- `partners.txt`

All other files (including any testing files you create) will not be used for grading.

Part 2: Flood Fill

For this part of the assignment, you will be writing a number of functions that execute a "flood fill" on a PNG image. To flood fill a region of an image, you specify a point on the image and a fill color (or pattern) and all points similar in color and adjacent to the chosen point are changed to the fill color (or pattern). We will implement two different fill algorithms and three different fill patterns. The two fill algorithms are animated in the pictures above with a rainbow color fill pattern that we implemented.

Functors

Your first task in this part of the project is to implement the mechanism by which colors are selected for the fill pattern. To do this, you will create function objects (or "functors") which will be passed into the fill algorithm and applied to pixels in a PNG. Specifically you will be making two new functors, each of which will be derived classes of the abstract base class called `colorPicker`, which is provided and described in the given code file `colorPicker.h`. We have also given you an example functor so you can see how things are supposed to work. Our functor is called `rainColorPicker`. The `colorPicker` base class has one purely virtual function that you **must** implement in your derived classes, but you may add others if you wish. In particular, you should expect your derived class constructors to initialize the functor with situation-dependent variables.

The purpose of any `colorPicker` is to take an image coordinate as its `(x,y)` parameters to `operator()`, and return the color it would suggest the client recolor the source PNG to. So in the examples above, the `colorPicker` is returning rainbow colors, based on the number of iterations in the fill algorithm.

Your task is to implement the `stripeColorPicker` and `borderColorPicker` and `customColorPicker` function objects. Refer to the given code for implementation details: they **must** overwrite the abstract **`operator()`**!

- **`stripeColorPicker`** The first pattern will be a simple stripe flood fill. Its behavior is described in the given code, and sample images are given.
- **`borderColorPicker`** The second fill pattern will leave most of the region unchanged, but it will draw a border around the edge of the region. Again, the specification is described in the given code.
- **`customColorPicker`** The third exercise will be a fill pattern of your own creation!! You must create files `customColorPicker.h` and `customColorPicker.cpp` to change the pixels in the fill using some interesting pattern you design. You'll also give us the `.gif` images created by your custom work. Note that we'll leave it to you to figure out how to change your `Makefile`, `filler.h`, and `filler.cpp` to accommdate the new color picker.

The Fill Algorithms

The animations in the .gif images above are the result of **breadth-first-search** (BFS) and **depth-first-search** (DFS), respectively, both with a rainbow fill. We have given you the rainbow fill functions as an example for the others.

You will be implementing both DFS and BFS, each with the three different fill `patterns` described above.

You must implement **all** of the functions inside the `filler` namespace. Refer to the documentation in the header file for implementation details.

Testing

> ⚠ **ImageMagick**
>
> Note that you must have ImageMagick installed if you're developing on your personal machine. You can test for this requirement by running
>
> ```
> convert
> ```
> TERMINAL
>
> in your terminal. Windows users should visit ImageMagick to download the latest binary. OSX users can use homebrew or MacPorts to install it if missing. Linux users likely already have it installed. If not, then you can install it via your system's package manager.

We have provided a file `testFills.cpp` which includes several test cases that your code should pass if it is correct. To compile this executable, type:

```
make testFills
```
TERMINAL

This will create executable `testFills`.

One important thing to note about this PA is that the `.gif` compression may take some time. In general, on the remote machines via ssh, the solution code runs in somewhere from 5-20 seconds when all the tests are enabled and completed and the server is under a light load. It is advisable to comment out the tests for things you have already completed, so that you don't have to wait as long (and use the CPU up) for things you already did. That said, the main point is that if you code doesn't finish immediately don't fret. Also note that as more traffic is happening on the server things will slow down. This may be an incentive to not wait until the last minute and have to try to run your code when everyone else is too!

The `testFills` program will generate images and animated gifs in the `images/` folder, which can be diffed with the solution images in `soln_images/`. For animated gifs, the frames making up the animation are placed in `frames/`, and can be diffed with their corresponding expected frames in `soln_frames/`.

You can check your color picker outputs by redirecting `testFills`' output to a file and `diff`ing that with our solution output:

```
./testFills all > testFills.out
diff testFills.out soln_testFills.out
```
TERMINAL

You can also diff each frame created for the animation, which will probably be useful for debugging:

```
diff frames/bfssolid00.png soln_frames/bfssolid00.png
```
TERMINAL

These test cases are deliberately insufficient. We encourage you to augment this file with additional test cases, using the provided ones as examples.

Handing in Your Code

To facilitate anonymous grading, **do not** include any personally-identifiable information in any of your source files. Instead, before you hand in this assignment, create a file called `partners.txt` that contains only the CSIDs of people in your collaboration group (if it exists), one per line. If you worked alone, include only your own CSID in this file. We will be automatically processing this information, so do not include anything else in the file. If we must manually correct your submission, you may lose points. As always, if you're working in a group, each group member must hand in the assignment. Failure to cite collaborators violates our academic integrity policy; we will be aggressively pursuing such violators.

You will submit your work to gradeScope as you did for PA1.

Grading Information

The following files are used to grade PA 2:

- `deque.cpp`
- `stack.cpp`
- `queue.cpp`
- `stripeColorPicker.cpp`
- `stripeColorPicker.h`
- `borderColorPicker.cpp`
- `borderColorPicker.h`
- `customColorPicker.cpp`
- `customColorPicker.h`
- `filler.h`
- `filler.cpp`
- `customDFS.gif`
- `customBFS.gif`
- `partners.txt`

All other files will not be used for grading.

Good luck!