Implementing Queues and Deques

Reminder of cheating policy

You are free to ask other people for help, however

- No one else may edit your code
- You may not copy code from other people
- You may not provide your code to other students; this includes publishing your code on sites like github. If you want to use a source-code control system such as git, use a local repo or a private repo.

Basically don't let other people touch your keyboard, and (obviously) don't copy-and-paste their work.

In addition, for this assignment you will be implementing stacks, queues and deques. It obviously defeats the purpose of the assignment if you use the built-in complex data types of C#. So for this assignment it will also **be considered cheating if you use the built-in collection classes of C# (i.e. LinkedList<T>, List<T>, Queue<T>, ArrayList, etc.)**. You may, of course use arrays and any data types you define yourself.

Northwestern policy requires us to refer you immediately to the Dean if you are suspected of cheating.

Summary

For this assignment, you should implement the following classes

- Array-based queues
- Linked-list-based queues
- Deques using a doubly-linked list

We have provided you with:

- The skeletons of the classes. These include the declarations of methods, but the method bodies all contain the line "throw new NotImplementedException()", which generates an exception complaining that you haven't implemented the method yet.
- A suite of unit tests[1] that will automatically check the correctness of your implementation

So all you should need to do is to fill in the method bodies, run the unit tests, and then fix the tests that fail.

[1] So-called because they test the correctness of individual units (modules) of the program, rather than the correctness of the overall program. The latter are called **integration tests**.

We have asked you to implement a **wider range of methods** for these classes than were discussed in class. In particular, you will be implementing get methods for properties that return the number of items in the queue, and whether it is empty or full.

We have also asked you to implement **error checking** in your add and remove methods. In particular, if the user attempts to remove from an empty queue or deque, you must throw the QueueEmptyException, which you can do by saying (not surprisingly):

```
throw new QueueEmptyException();
```

In the case of ArrayQueues, which have a limited capacity, your Enqueue method should also check whether the queue is full, and if so, throw the QueueFullException (same code as above, but chance the name of the exception).

Using the code

To use the code we have provided,

- Unzip the zip file provided
- Open the file EECS 214 Assignment 1.sln (the file with the little Visual Studio icon with a little 9 inside it. This will launch visual studio.
- If Visual Studio complains that there is an unsupported project type, that means you are using a different version of visual studio. You want Visual Studio 2010 Ultimate. The simpler versions do not support automated testing.

Important note:

It is a requirement of this assignment that the IsEmpty and IsFull properties of all classes you implement run in **constant time**, aka O(1) time. In other words the amount of time they take can't vary depending on the length of the queue. That may require that you override the implementation of IsEmpty rather than use the default implementation that calls Count and compared the result to zero. If you don't handle this properly, then some of the tests will take a ***very long time to run***. If this happens, you need to check the implementations of your IsEmpty and/or IsFull properties to make sure they aren't counting up all the elements in the queue.

Writing the code

We're already written the skeleton of the code for you. Your job is to fill in the blanks in the following files:

- ArrayQueue.cs
- LinkedListQueue.cs
- Deque.cs

You do not need to modify Queue.cs, as this is the parent class of ArrayQueue and LinkedListQueue, it's an abstract class, and we've already written all of it for you.

To fill in the code for a class, open its file and then

- Add code to the class to add whatever fields you need for the class
- Either initialize the fields in their declarations, or provide a constructor to initialize them. If you write a constructor, make sure it doesn't take any arguments, since the test code assumes there aren't any.
- Fill in the body of every method in the file that says "`throw new NotImplementedException();`". Also rememeber to remove the throw statement from the method, or the testing code will be confused and think you haven't implemented the method.

Testing the code

To run the test, select Run>All tests in solution from the Test menu. This will run the tests normally. When you run the tests, the system will compile your code and run through all the tests, displaying their results in the window at the bottom of the screen (where compile errors go).

If there is a problem with one of your tests, then you will probably want to examine its execution using the debugger. To do this, select Debug>All tests in solution from the Test menu instead. This will allow you to set breakpoints, single-step execution, etc. Note however that when using the Debug all tests command your program to pause in the debugger **any time** an exception is thrown, even if the test was deliberately trying to cause the exception to occur (and hence, throwing the exception was what your code had to do to pass the test). So don't panic if you see it throw an exception. You can continue from the exception by pressing F5 or pressing the play button in the toolbar. Note also that when Visual Studio pauses after an exception, it will often pause in the line **after** the throw command (which is what trigger the exception). So again, don't panic because you can't understand why that exception was being trigged by that line – check the line above it to see if it had a throw command.

Again, remember that you are required to insure that the implementations of IsEmpty and IsFull run in constant time (see above). If you find one or more of the tests running prohibitively slowly, do not ignore the test, find out why it's running slowly and fix it.

Making your code work

Once you've implemented your queues, test the code, scroll through the list of tests, looking for any that failed. When you find a test that failed, double-click it. This will display more information about the test and why it failed. Hopefully it will be relatively self-explanatory, but if not, you can click the hyperlink in the window and it will bring you to the actual code performing the test so that you can see precisely what it was doing.

The question now is why the test failed. In many cases, it may be obvious. For example if the test was checking whether your code properly threw a QueueEmptyException when trying to dequeue from an

empty queue, you may say “oh, nuts! I forgot to check for that!”. In that case, you can just add the check, rerun the tests, and hopefully the test will pass now.

If it’s not obvious what’s going wrong, then

- Click the hyperlink to bring you to the code for the test that failed.
- Place a breakpoint at the beginning of the test
- Rerun the tests. The debugger should stop at the beginning of the test.
- Now single-step through the code (using F10 and F11, see previous lectures), and examine the values of the variables until you see something that’s awry. Hopefully, that will come soon.
- Remove the breakpoint and fix error you found in the code.
- Rerun the test

Now repeat this process until all the tests pass. Once that happens, you know your assignment is correct and you can hand it in.

How the tests work

Note: For this assignment, we’re providing you with the tests. However, for future assignments, you will need to write your own. So it’s worth taking some time to learn how the tests we’re providing to you work.

Each test is just a simple method that calls your code and makes **assertions** about what should happen as a result. An assertion is a piece of code that checks a condition. If the condition is true, then execution continues, but if it’s false, the code throws and exception. The testing system works by running each test method, and keeping track of which of them has assertions that fail. For those whose assertions fail, it remembers the assertion that failed and displays it alongside the name of the assertion in the list at the bottom of the screen.

Microsoft’s testing framework supports a number of assertions, all of which are methods of the magic class Assert:

- Assert.IsTrue(boolean)
  Continues execution if it’s argument is true, otherwise fails and marks the test as having failed.
- Assert.IsFalse(boolean)
  Same, but the test passes if the boolean is false.
- Assert.AreEqual<Type>(x, y)
  Test passes if x and y are equial. The <Type> part tells what type x and y are.
- Assert.Fail()
  Never passes. It’s a way of saying “execution should never get to this point, so if it does, then something’s wrong”.

Attributes

Note: This paragraph is for the curious. You can safely ignore it if you want. The methods in the code have little bracketed expressions before them, like `[TestMethod()]`. These are called **attributes**. You can safely ignore them, although please don't delete them. For what it's worth, these are **metadata** (data about the program rather than the instructions for the program itself) that's used by the testing system. They allow the testing system to ask .NET for all the methods in the testing class that are tagged with the TestMethod attribute (this is how it figures out what the tests are). There is also another attribute called `ExpectException`, which allows the programmer to say that the whole point of a given test is that it's supposed to generate a particular exception and so the test passes if it does generate that exception, and fails if it doesn't.

Turning the assignment in

To turn in your assignment, please do the following:

- In Visual Studio, chose "Clean Solution" from the Build menu. This will remove all the binary files from the solution, leaving only the source code. That will make your assignment smaller and easier to upload.
- Close Visual Studio
- Right click on the folder icon for your assignment and choose the "Compressed (zipped) folder" selection from the Send To menu.
    - **IMPORTANT:** Many of you are using commercial compression software such as **WinRar**. You may use these programs to make a zip file, if you prefer. However, **you must specifically turn you program in as a ZIP file**. Do not turn in other file formats such as RAR, TAR, TGZ, etc.. **Non-ZIP files will be returned ungraded** and your assignment will be **counted as late until you submit a ZIP file**.
- Return to this assignment on Canvas and upload your zip file as your submission.