# 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 small number of unit tests[1] to get you started with testing your code.

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.

[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 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).

**Important:** You should make the capacity of your ArrayQueue be 256. In a professional implementation, the capacity would be specified by a parameter in the constructor, but for this assignment, we'll keep things simple and fix it to be 256.

## 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 listed as type "Microsoft Visual Studio Solution"). This will launch visual studio. If you have multiple versions of Visual Studio installed, then right click on the sln file and choose Open With to make sure you open it in VS 2017.

### Important note:

It is a requirement of this assignment that all operations of all classes, including the Count, IsEmpty and IsFull properties, should 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. Part of your job is to understand what O(1) means. So if you're having trouble understanding O() notation, do feel free to ask us in office hours. But you may not ask if a given piece of code runs in O(1) time. That's up for you to determine!

## 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

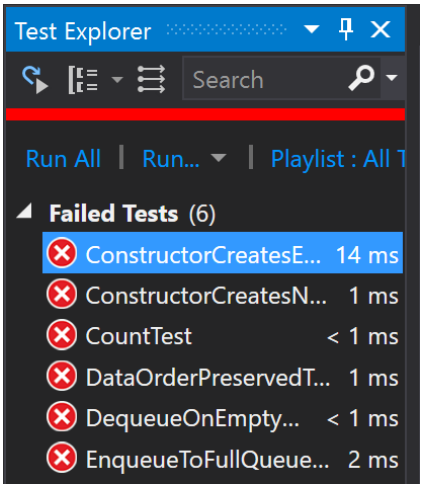
- Add code to the class to add whatever fields you need for the class
- Either initialize the fields in their declarations, or provide a constructor to initialize them. If you write a constructor, make sure it doesn't take any arguments, since the test code assumes there aren't any.
- Fill in the body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

We recommend that you proceed by writing a method, then testing it (see below), then moving on to the next method.

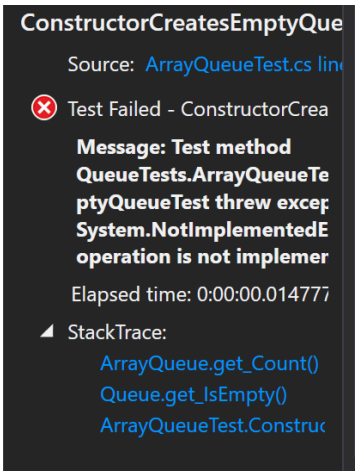
Again, remember that you are required to insure the implementation of IsEmpty and IsFull run in *constant time* (see above). If you find one or more of the tests running prohibitively slowly (see below), do not ignore the test, find out why it's running slowly and fix it.

## Testing the code

We've included a few tests with the assignment, although not nearly enough. To run the tests, 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 Test Explorer window on the left-hand side of the screen. If the tests show up green, you're in good shape. If they show up with X's in red circles:




then those tests have failed. Click on one of the failed tests and look at the bottom of the Test Explorer pane. You'll see an explanation of what went wrong in the process of running that test:



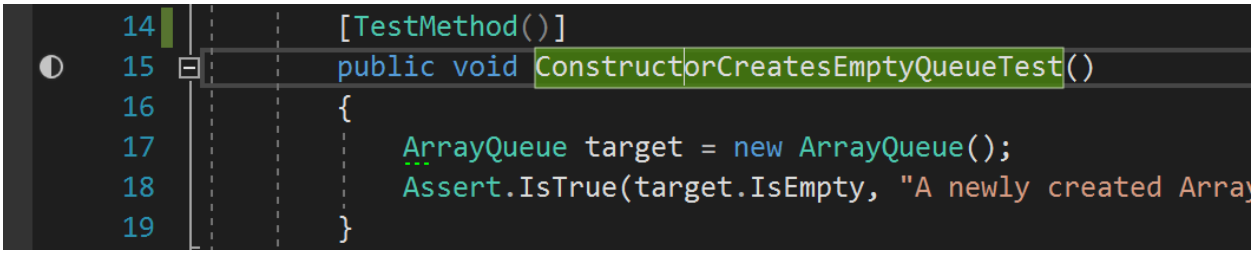

In this case, it's saying that the test failed because the test threw the NotImplementedException, i.e. I hadn't gotten around to writing code for the method being tested.

## Making your code work

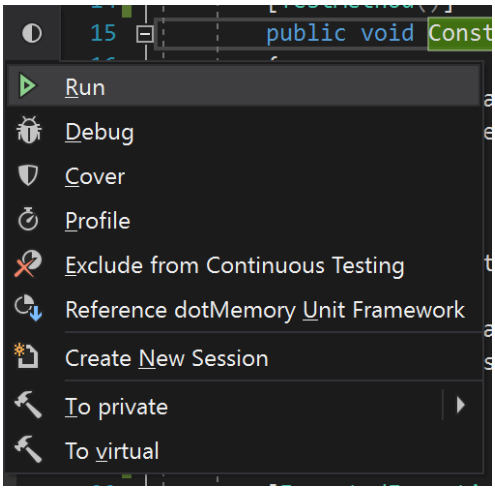
Now double-click the failed test. It will take you to the source code of the test. Look at the code for the test to see what it was trying to do. There's a description of how the testing system works in the appendix. But most of it comes down to calling either Assert.IsTrue() to run a piece of code to see if it returns true (the test fails if it doesn't) or Assert.AreEqual() to run a piece of code and see if it returns the right value (again, failing the test if it doesn't).

You may be able to tell what's wrong just by looking at the code that's being called. But if not, you'll probably want to run the code with debugging turned on. That will mean, for example, that if the code throws and exception, the debugger will pop up the code that threw it and let you inspect the relevant local variables. If you still can't figure out what's going on, then you'll probably want to set some breakpoints or single-step it.

So how do you run a test in debug mode? Look next to the test, and you'll see a half-filled circle:




Right click on the half-filled circle, and you'll get a menu:




Choose Debug and it will run the test with debugging turned on. Note 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). Don't panic because you can't understand why that exception was being triggered by that line; check the line above it to see if it had a throw command.

## Writing more tests

We've written a couple of tests to get you started. 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 an 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

The methods in the testing code have little bracketed expressions before them, like `[TestMethod()]`. These are called **attributes**. 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 the program for all its testing methods so it knows what methods to run. 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.

## Writing more tests

We've provided you with files with "testing classes" (ArrayQueueTest, DequeTest, LinkedListQueueTest) for each of the classes you're going to write. These are just containers for your test methods. We've put some basic tests in one of them, ArrayQueueTests.cs, to get you started. But it's not nearly enough. So now you should start thinking of other tests to write and add them to the appropriate test files.

The basic template for a test looks like this:

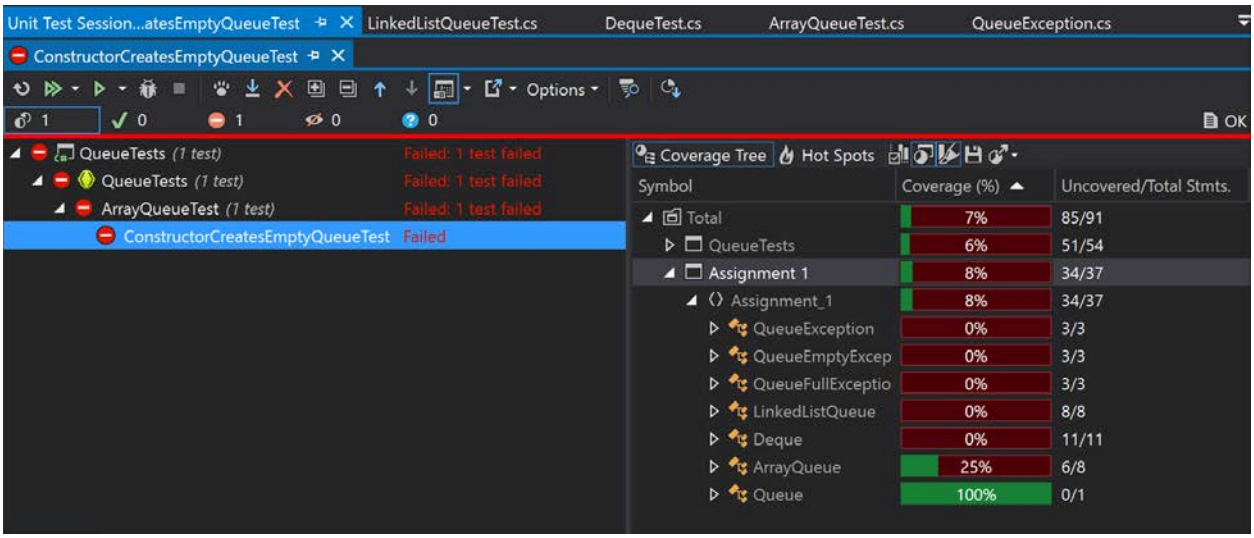
```
[TestMethod]
public void NameOfMyTest() {
  … optionally initialize some things …
    Assert.SomeThing(SomeArguments);
}
```

Add one of these to the appropriate test class for each new test you want to try. Try to think about of every case that could come up and write a test for it.

## Coverage analysis

Of course, you can never be sure you've tested everything you ought to test. But one thing that can help you feel a little more secure is to make sure that your tests collectively run all of your code. That's called *code coverage analysis*. In 111, when you used Racket, you'd see lines of your code highlighted in black. That mean that Racket's coverage tool notice those lines had never been run. ReSharper gives you a similar tool.

Go to the ReSharper menu in Visual Studio and choose ReSharper>Unit tests>Cover Unit Tests. This will run all your unit tests, exhaustively logging every line that gets executed. It will pop up a report showing you how much of your code actually got run by the tests:




You want to have ArrayQueue, LinkedListQueue and Deque listed at or near 100%. That means that even if your tests didn't try every possible thing, you at least know they ran every possible line of those files.

But wait, you ask! How do I know which lines were actually executed? Just go back to your source code. Next to every line there will be a little bar that will be color-coded to tell you whether it was run and what the rules were:

```
22      /// <summary>
23      /// Remove object from beginning of queue.
24      /// </summary>
25      /// <returns>Object at beginning of queue</returns>
26      public override object Dequeue()
27      {
28          // Remove this line when you implement this mehtod
29          throw new NotImplementedException();
30      }
31
32      /// <summary>
33      /// The number of elements in the queue.
34      /// </summary>
35      public override int Count
36      {
37          get {
38              // Remove this line when you fill this method in.
39              throw new NotImplementedException();
40          }
41      }
42
```

A green bar means it was executed and every test that executed it succeeded. A red bar means it was executed in a test that failed. A white or gray bar means it never got run at all. Your job is to make sure all the bars are green.

## 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.

## Appendix: more on writing tests

Each test is just a simple method that calls your code and makes **assertions** about what should happen as a result. As in EECS-211, 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 an 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".

These methods can optionally take an error message as an additional argument, in which case the testing rig will display that message rather than a generic "test failed" message, should the assertion fail.

## Marking test methods using attributes

When the test system starts up, it will look at all the files in the Test project and search for methods that start with the annotation:

```
[TestMethod()]
```

Or:

```
[TestMethod]
```

It finds all those methods and runs them. The bracketed annotations are called **attributes** and they they allow the code to search through its own methods at run time to find ones that are annotated with particular attributes. So the equivalent of (check-expect *expectedValue expression*) from EECS-111 is something like this:

```
[TestMethod]
public void nameofmytest() {
   Assert.AreEqual(expectedValue, expression);
}
```

So it's very much like 111, only more verbose. Sometimes you need to have some setup code before you do the real test, for example to make some local variables or do some initialization of an object. If so, you can put that in the method, before the call to Assert.AreEqual:

```
[TestMethod]
public void nameofmytest() {
   … setup code …
   Assert.AreEqual(expectedValue, expression);
}
```

For example, here are some of the tests for array queues from assignment 1:

```
[TestMethod()]
 public void ConstructorCreatesEmptyQueueTest()
 {
     ArrayQueue target = new ArrayQueue();
     Assert.IsTrue(target.IsEmpty,
             "A newly created ArrayQueue should have IsEmpty=true, but doesn't");
 }

 [TestMethod()]
 public void ConstructorCreatesNonFullQueueTest()
 {
     ArrayQueue target = new ArrayQueue();
     Assert.IsFalse(target.IsFull,
             "A newly created ArrayQueue should have IsFull=false, but doesn't");
```

```
}

[TestMethod()]
public void QueueEventuallyFillsText()
{
    ArrayQueue target = new ArrayQueue();
    int i;
    for (i = 0; i < 1000000 && !target.IsFull; i++)
        target.Enqueue(i);
    Assert.IsTrue(target.IsFull,
    "Added 1000000 elements to ArrayQueue and it never registered IsFull=true");
}
```

Notice that these tests specify custom error messages to print, since these are more informative than a generic "test failed" message. But you don't have to do that for your code if you don't want to.

Sometimes, you want to write a test to make sure that a method that's supposed to those an exception does throw an exception. You can do that by adding an ExpectedException attribute:

```
[TestMethod]
[ExpectedException(typeof(ExceptionType))]
public void nameofmytest() {
    … setup code …
    … code that ought to generate the exception…
}
```

The ExpectedException attribute says that the test passes if it throws the specified exception. The Assert.Fail() call says that if the code gets that far, the test has failed (because it didn't throw the exception). For example, here is one of the tests for the array queues in assignment 1:

```
[TestMethod()]
[ExpectedException(typeof(QueueFullException),
                   "Adding to full queue should throw QueueFullException")]
public void EnqueueToFullQueueThrowsQueueFullExceptionTest()
{
    ArrayQueue target = new ArrayQueue();
    Assert.IsFalse(target.IsFull);
    int i;
    for (i = 0; i < 1000000 && !target.IsFull; i++)
        target.Enqueue(i);
}
```

Again, this test is a little fancier in that it includes extra arguments to ExpectedException to provide custom error message for this particular test. Also, the real test we handed out in the code included some more checks t