Queues

Tim "Dr. T" Chamillard

Introduction

A queue is a data structure with specific access characteristics. Think of getting in line to get into a concert. If you were in the UK (and many other places), that line would be called a queue, and the queue data structure works like the line. We can add someone to the end of the line; when we do so, we say we *enqueue* someone. We can remove the person at the front of the line; we say we *dequeue* that person. We can also look at the person at the front of the queue without removing them from the queue; this is called peeking at the queue.

Because the first thing we added to a queue is the first thing we remove from the queue, a queue is a FIFO (First In First Out) data structure. We can implement the queue using an array, a list, or a variety of other underlying structures, but any queue implementation needs to enforce FIFO access.

The most common use of queues in the game domain is probably storing information about players waiting to enter a particular game or match under the assumption that player skill level doesn't matter. We could also use a message queue to make sure messages are displayed in the order in which they were sent. We can use a specialized form of queue called a priority queue to ensure that critical messages are displayed before non-critical messages. We use a priority queue in our Battle Paddles game to help the AI process the most important items first.

The `System.Collections.Generic` namespace in C# provides a `Queue` class that does the same things we'll do below and more. The underlying implementation in that class is an array rather than a list, but we get the same behavior from Enqueue, Dequeue, Peek, and Count. It's important to work through how to actually implement a queue to deepen your understanding of the data structure and the operation complexities of that data structure, but you should simply plan to use the `Queue` class when you need a queue (and you will).

Although queues can expose many operations, we'll implement a queue that provides a Count property and Enqueue, Dequeue, and Peek methods. Let's get to work.

Instance Variables and the Constructor

Let's start building our queue by adding the instance variables and a constructor. We're of course going to make our queue generic so we can store any data type in the queue.

Here's the start of our `Queue` class:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Queues
{
    /// <summary>
    /// A queue
    /// </summary>
    /// <typeparam name="T">type for queue elements</typeparam>
    class Queue<T>
    {
        List<T> contents = new List<T>();

        #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        public Queue()
        {
        }

        #endregion
```

The only field we need is something to hold the contents of the queue. We're using a list for this, but you have a variety of choices for that data type.

Count

The `Count` property simply returns the number of items currently on the queue. Because this is exactly the same as the number of items in our `contents` field, this is a pretty simple property:

```
        #region Properties

        /// <summary>
        /// Gets the number of items in the queue
        /// </summary>
        public int Count
        {
            get { return contents.Count; }
        }

        #endregion
```

Notice that we can get the value of this property but we can't set it. This makes sense, because we shouldn't be able to directly manipulate this count. The only way we can (indirectly) change this property's value is by enqueueing items on and dequeueing items from the queue.

Algorithmic Complexity

The documentation for the `List Count` property says "Retrieving the value of this property is an O(1) operation." That means the get for our `Count` property is also O(1).

Enqueue

The `Enqueue` method adds a new item to the back of the queue. Here's the method:

```
#region Public methods

/// <summary>
/// Adds an item to the back of the queue
/// </summary>
/// <param name="item">the item to add to the queue</param>
public void Enqueue(T item)
{
    contents.Add(item);
}
```

Algorithmic Complexity

The complexity for the `List Add` method is just like for the `Add` method for our dynamic arrays. If we don't have to grow the list, this is an O(1) operation, but if we do have to grow the list this is an O(n) operation.

Dequeue

The `Dequeue` method removes the front item from the queue and returns it. If the queue is empty, the method throws an `InvalidOperationException`. Here's the code:

```
/// <summary>
/// Removes the item from the front of the queue
/// </summary>
/// <returns>the removed item</returns>
public T Dequeue()
{
    // throw exception if try to dequeue from an empty queue
    if (contents.Count == 0)
    {
        throw new InvalidOperationException(
            "Can't dequeue from an empty queue");
    }
    else
    {
        // retrieve front of queue, remove, and return
        T item = contents[0];
        contents.RemoveAt(0);
        return item;
    }
}
```

Algorithmic Complexity

Accessing the 0th element of the list is an O(1) operation. Unfortunately, `RemoveAt` is an O(n) operation, where n = Count – the index of the element we want to remove. Because we're always removing the 0th element of the list, we always get the worst case behavior from the `RemoveAt` method since n = Count every time. Dequeue is therefore an O(n) operation.

Peek

The `Peek` method also returns the first item on the queue, but it does NOT remove that item from the queue. Like the `Dequeue` method, if the queue is empty the `Peek` method throws an `InvalidOperationException`. Here's the code:

```
/// <summary>
/// Peeks at the front item on the queue
/// </summary>
/// <returns>the front item</returns>
public T Peek()
{
    // throw exception if try to peek at an empty queue
    if (contents.Count == 0)
    {
        throw new InvalidOperationException(
            "Can't peek at an empty queue");
    }
    else
    {
        // return front of queue
        return contents[0];
    }
}
```

Algorithmic Complexity

As noted above, accessing the 0th element of the list is an O(1) operation, so Peek is an O(1) operation.

Testing the Queue Class

You should take a look at the TestQueue code in the provided project to see how we can thoroughly test the `Queue` class. You should pay close attention to how I build the expected results for each of the test cases, especially in TestDequeue and TestPeek.

In general, here's what we test in that class:

Queue Property/Method	Tested Characteristics
Constructor	Count is 0 Queue has no items

Count	Correct based on # of items in queue
Enqueue	Count increases by 1 Contents reflect new item at back of queue
Dequeue (empty queue)	InvalidOperationException thrown
Dequeue (non-empty queue)	Correct item returned Count decreases by 1 Contents reflect item removed from front of queue
Peek (empty queue)	InvalidOperationException thrown
Peek (non-empty queue)	Correct item returned Count doesn't change Contents of queue don't change

Priority Queues

The introduction section above discusses some uses of priority queues, which we haven't explored yet. A priority queue works the same as a regular queue, but with one additional constraint – items with higher priority are added closer to the front of the queue. Essentially, the queue keeps the items in the queue sorted in priority order.

How do we know the priority of a particular item we're adding to a priority queue? The standard way to do this is by making sure any reference types we want to include in a priority queue implement the `IComparable` interface. Recall that we did the same thing for our `OrderedDynamicArray` so we could keep that array sorted; since a priority queue also needs to be sorted, we use the same approach here.

As a reminder, the `IComparable` interface requires that we implement a `CompareTo` method that compares this object to another object of the same type. If this object is less than the other object `CompareTo` returns a number less than 0, if the objects are equal `CompareTo` returns 0, and if this object is greater than the other object `CompareTo` returns a number greater than 0. We don't have to worry about how "less than", "equal to", and "greater than" are actually defined – that happens inside the `CompareTo` method implementation – but we can easily use the defined `CompareTo` behavior without worrying about the details. For our priority queue, we'll assume lower numbers mean higher priority, but we could of course simply change the sign of the `CompareTo` result to treat higher numbers as higher priorities.

C# doesn't provide a priority queue class, so you have to implement one on your own if you need one!

Instance Variables and the Constructor

Let's start building our priority queue by adding the instance variables and a constructor. We're of course going to make our queue generic so we can store any data type on the queue; note the use of the constraint (the `where T:IComparable` part of the header) to say that any T we use to instantiate the `PriorityQueue` class has to implement the `IComparable` interface.

Here's the start of our `PriorityQueue` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Queues
{
    /// <summary>
    /// A priority queue
    /// </summary>
    /// <typeparam name="T">type for queue elements</typeparam>
    class PriorityQueue<T> where T : IComparable
    {
        List<T> contents = new List<T>();

        #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        public PriorityQueue()
        {
        }

        #endregion
```

Count, Dequeue, and Peek

The `Count` property and the `Dequeue` and `Peek` methods work exactly the same as they did for the queue we already implemented, and they have the same algorithmic complexity as well.

Enqueue

The `Enqueue` method adds a new item to the priority queue. Unlike the standard queue, though, we can't just add the new item to the back of the queue. Instead, we need to add it to the queue in the appropriate place. Here's the method:

```
        /// <summary>
        /// Adds an item to the the queue
        /// </summary>
        /// <param name="item">the item to add to the queue</param>
        public void Enqueue(T item)
        {
            // find location at which to add the item
            int addLocation = 0;
            while ((addLocation < Count) &&
                (contents[addLocation].CompareTo(item) < 0))
            {
```

```
            addLocation++;
        }

        // insert the item in the list at the appropriate location
        contents.Insert(addLocation, item);
    }
```

The loop to find the correct location for the item to be added is similar to the loop we used in the `Add` method for our `OrderedDynamicArray` class. We then use the `List Insert` method to insert the item in the appropriate location in the list.

Algorithmic Complexity

Our analysis here is similar to the `Add` method for our `OrderedDynamicArray`. The linear search at the beginning of our `Enqueue` method is O(n) in the worst case, because the item might belong at the end of the queue. The `List Insert` method is also O(n), where n is the size of the list. The `List` class may also need to expand the size of the list (in case it's already full), which is also an O(n) operation. The Enqueue operation is therefore an O(n) operation.

Testing the Priority Queue Class

You should take a look at the TestPriorityQueue code in the provided project to see how we can thoroughly test the `PriorityQueue` class. Most of the test methods are the same as for the queue testing, but I did change the `TestEnqueue` method. That method now takes an additional parameter giving the expected string representation of the new queue, which makes it much easier to specify that the queue stays sorted.

In general, here's what we test in that class:

Queue Property/Method	**Tested Characteristics**
Constructor	Count is 0 Queue has no items
Count	Correct based on # of items in queue
Enqueue	Count increases by 1 Contents reflect new item at correct location in queue
Dequeue (empty queue)	InvalidOperationException thrown
Dequeue (non-empty queue)	Correct item returned Count decreases by 1 Contents reflect item removed from front of queue
Peek (empty queue)	InvalidOperationException thrown
Peek (non-empty queue)	Correct item returned Count doesn't change Contents of queue don't change