Dynamic Arrays Reading

Tim "Dr. T" Chamillard

What's a Data Structure?

Data structures are ways that we can structure data (duh). In other words, there are lots of ways that we can store and manipulate data, and the way that we choose to store the data can have a significant effect on how efficiently we can manipulate that data.

So is a class a data structure? It certainly organizes data inside the class as instance variables, and it lets us manipulate that data by calling methods. The short answer is no, we usually think about a class as a data *type* rather than a data structure. So when we're using classes to model entities like NPCs and resources, for example, these classes aren't considered data structures.

Arrays are the most basic data structures around, so let's start with them. Since you've already had some practice with arrays, you should remember that arrays simply store a set of values or references to objects in a set of sequentially-numbered locations in memory. We've done some basic array manipulations already, but we'll add some more complicated operations (like remove) and let the array grow if necessary as well.

Now we'll make the distinction between classes and data structures fuzzier, though, because we're going to implement our new array stuff in a `DynamicArray` class. Did I lie to you just a couple paragraphs ago? Only a little! We have lots of different kinds of classes: those that model entities (as discussed above), those that hold collections of data (like the `List` class in `System.Collections.Generic`), those that provide utility methods (like those in `System.Math`), and others. The classes we implement for our data structures will be more like the collections classes than the "model an entity" classes. That makes them data structures, even though we use classes to implement them in C#.

Our Abstract Base Class

Let's start by developing a basic class that acts much like a regular array. We're going to start with some easy stuff for the array, but we'll gradually grow it into a full-blown generic data structure.

The big difference between our classes and a normal array is that we'll let our array grow as necessary as we add elements to it. A data structure with these characteristics is commonly called a *dynamic array*.

You should be thinking at this point that this data structure sounds a lot like a `List` -- and you're right! In practice, you should definitely just use the `List` collection class when you need the functionality described here rather than using the custom classes described here.

Why even bother with this reading, then? First, it gives us a chance to continue developing our understanding of how to code stuff in C#. Also, we need the source code (e.g., the algorithms expressed in C#) for us to do algorithm analysis on different operations. Although we do have access to the source code for the `List` class through the MSDN documentation, it's pretty complicated because it's "industrial strength" (which is a good thing!). We'll use the more basic source code here to work on our algorithm analysis skills instead.

We'll begin by making an `IntDynamicArray` that just holds integers. That will let us focus on the interesting algorithm stuff before making a more generic `DynamicArray` that can hold any data type. Because there are some interesting algorithm differences between dealing with unsorted and sorted arrays, we'll make `IntDynamicArray` an abstract class then make a couple concrete classes called `UnorderedIntDynamicArray` and `OrderedIntDynamicArray`.

Our abstract `IntDynamicArray` is shown below. Notice that I've embedded explanatory text to explain the code; that text obviously isn't part of the source code and wouldn't compile. You can (and should) download the full project code from Blackboard before continuing. Here's the code:

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

namespace ArraysProject
{
    /// <remarks>
    /// Provides a dynamically-sized array of integers
    /// </remarks>
    public abstract class IntDynamicArray
    {
        const int ExpandMultiplyFactor = 2;
        protected int[] items;
        protected int count;
```

The three fields will hold the multiplication factor we'll use to expand the array, the array of the actual integers we have in our dynamic array, and the count of how many things are in the dynamic array. We'll also need to know how many elements the array can currently hold (think of that as the current array capacity), but we can just get that from the `Length` property of the array. Since it's common practice to double the size of the array each time it's expanded rather than add a particular amount, that's the approach we'll take here.

Because in many cases `count` will be less than the capacity, you should realize that the items array will generally hold a combination of "real" values (in elements `0` through `count - 1`) and "dummy" values (in elements `count` through `items.Length - 1`). The only "problem" that causes is that we usually have some wasted space holding dummy values in the array. We could reduce that wasted space by more slowly growing the size of the array (by multiplying the size by 1.5 rather than 2, for example), but that would

mean having to expand the array more often, which we'll see is an expensive operation to do. What do you know – a tradeoff between memory consumption and processing speed! Shocking, isn't it?

```
#region Constructor

/// <summary>
/// Constructor
/// </summary>
protected IntDynamicArray()
{
    items = new int[4];
    count = 0;
}

#endregion
```

We know we'll never be able to explicitly call the constructor to instantiate an object for the abstract class, but because the concrete classes will all do the same thing when their constructors are called, we include the constructor here. Notice that we used protected (rather than our typical public) as our access modifier to say that only classes that inherit from this class can call the constructor.

Also, we can use the `#region` and `#endregion` annotations to mark sections of code. When we do that, we can then click the minus box next to the start of the region in your IDE to hide all the code in that region.

```
#region Properties

/// <summary>
/// Gets the number of elements
/// </summary>
public int Count
{
    get { return count; }
}

#endregion
```

Collection classes provide a `Count` property to return the number of items in the collection. Even though our `IntDynamicArray` really acts like a collection, there's LOTS of other stuff we'd need to do to make it a “real” C# collection (by implementing the `ICollection` interface). Still, the `Count` property is pretty useful, so we provide it here.

```
#region Public methods

public abstract void Add(int item);
public abstract bool Remove(int item);
public abstract int IndexOf(int item);
```

Any concrete class that implements the abstract `IntDynamicArray` class will have to provide implementations for the three methods above. We make them abstract rather

than providing a default implementation of them because the implementations should be different for unordered and ordered arrays.

```
/// <summary>
/// Removes all the items from the IntDynamicArray
/// </summary>
public void Clear()
{
    count = 0;
}

/// <summary>
/// Converts the IntDynamicArray to a comma-separated string of
/// values
/// </summary>
/// <returns>the comma-separated string of values</returns>
public override String ToString()
{
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < count; i++)
    {
        builder.Append(items[i]);
        if (i < count - 1)
        {
            builder.Append(",");
        }
    }
    return builder.ToString();
}

#endregion
```

Both of the methods above work the same way whether the array is unordered or ordered. This is the power of inheritance – we can provide implementations for behavior (i.e., methods) that should be shared by all the child classes in the parent class so we don't have to duplicate it in the child classes.

```
#region Protected methods

/// <summary>
/// Expands the array
/// </summary>
protected void Expand()
{
    int[] newItems = new int[items.Length * ExpandMultiplyFactor];

    // copy elements from old array into new array
    for (int i = 0; i < items.Length; i++)
    {
        newItems[i] = items[i];
    }

    // change to use new array
    items = newItems;
}
```

```
        #endregion
    }
}
```

We'll use the `Expand` method to grow the array when we try to add another integer to an array that's already full. We basically create a new array that's double (or whatever `ExpandMultiplyFactor` is set to) the size of the current array, copy everything from the old array into the new array, and change our old array variable (`items`) to "point to" the new array instead. We didn't want this method to be public, because nobody using the `IntDynamicArray` class should know anything about expanding the structure or when it should happen. We couldn't make the method private either, though, because then the child classes can't call the method when they need to. That's why we used the protected access modifier for the method.

So how do we go about testing our abstract class? The bad news is that because we can't create an object for the abstract class, we need to wait until we have a concrete class that implements the abstract class. We can then test the inherited properties and methods in the concrete class. Don't worry, we'll get there soon!

As mentioned above, we're going to implement concrete classes for both unordered and ordered arrays. Let's start with the unordered class.

An Unordered Concrete Class

Now that we have our abstract base class, we can extend it with a concrete class for unordered dynamic arrays; we'll call that class `UnorderedIntDynamicArray`. The annotated code for that class is:

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

namespace ArraysProject
{
    /// <remarks>
    /// An unordered IntDynamicArray
    /// </remarks>
    public class UnorderedIntDynamicArray : IntDynamicArray
    {
        #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        public UnorderedIntDynamicArray()
            : base()
        {
        }

        #endregion
```

The constructor just calls the constructor in the abstract class.

```
        #region Public methods

        /// <summary>
        /// Adds the given item to the IntDynamicArray
        /// </summary>
        /// <param name="item">the item to add</param>
        public override void Add(int item)
        {
            // expand array if necessary
            if (count == items.Length)
            {
                Expand();
            }

            // add new item and increment count
            items[count] = item;
            count++;
        }
```

Before adding the given item, we need to make sure there's room in the array for it. If there isn't, we expand the array using the `Expand` method in the parent class. We then add the item and increment `count`.

Take a moment to think about how the `count` field really works. The field tells us how many items are currently in the array, just as you'd expect from its name. We get another cool use for `count` as well, though. Because arrays are indexed starting with 0, `count` also gives us the location of the next thing we're going to add to the array. That's why we compare `count` to `items.Length` with `==` to see if we should expand the array. For example, if the array size is 5 and `count` is 5 (meaning elements 0 through 4 in the array have "real" values in them), the array is full and we need to expand it before we try to add the item to element 5 of the array. Similarly, that's why we add the item to `items[count]` before incrementing `count` rather than incrementing `count` first (which might seem more intuitive to you).

Finally, because this array is unordered it doesn't matter where we put the new item. Since it's easiest to just add it to the end of the array, that's what we do.

```
        /// <summary>
        /// Removes the first occurence of the given item from the
        /// IntDynamicArray
        /// </summary>
        /// <param name="item">the item to remove</param>
        /// <returns>true if the item is successfully removed, false
        ///     otherwise</returns>
        public override bool Remove(int item)
        {
            // check for given item in array
            int itemLocation = IndexOf(item);
            if (itemLocation == -1)
            {
```

```
                return false;
            }
            else
            {
                // move last element in array here and change count
                items[itemLocation] = items[count-1];
                count--;
                return true;
            }
        }
```

The first thing this method does is get the location of the item to be removed using the `IndexOf` method (shown and discussed below). If the location is -1, this method returns false, saying the item wasn't removed. If we did find the item to be removed, we simply copy the last item in the array on top of the item to be removed then reduce the count of the items in the array by 1. This of course wouldn't work if the array was ordered, but because the order of the items doesn't matter we can do the remove very efficiently this way.

```
        /// <summary>
        /// Determines the index of the given item
        /// </summary>
        /// <param name="item">the item to find</param>
        /// <returns>the index of the item or -1 if it's not found</returns>
        public override int IndexOf(int item)
        {
            // look for first occurrence of item in array
            for (int i = 0; i < count; i++)
            {
                if (items[i] == item)
                {
                    return i;
                }
            }

            // didn't find the item in the array
            return -1;
        }

        #endregion
    }
}
```

The `IndexOf` method does a standard linear search for the given item, checking each of the "real" values in the array. If we find the item, we immediately return the location (the index of the element holding the item in the array) from the method. If we don't find the item, we return -1.

Now that we have a concrete class, we can actually do some testing. The project includes a `TestDynamicArrays` class that will run the test cases against all the classes we develop. The `TestUnorderedIntDynamicArray` class contains the test cases to test the `UnorderedIntDynamicArray` class shown above as well as the properties and methods in the abstract `IntDynamicArray` class. You should look at the code to see all

the details, but the test cases are listed in the table below (the names should be pretty self-explanatory).

Method/Property Tested	Test Cases
`Add`	`TestAddEmptyDynamicArray` `TestAddExpandDynamicArray`

`Remove`	`TestRemoveEmptyDynamicArray`
	`TestRemoveItemFrontOfDynamicArray`
	`TestRemoveItemBackOfDynamicArray`
	`TestRemoveItemInteriorOfDynamicArray`
	`TestRemoveItemNotInDynamicArray`

`IndexOf`	`TestIndexOfEmptyDynamicArray`
	`TestIndexOfFrontOfDynamicArray`
	`TestIndexOfBackOfDynamicArray`

`Count`	`TestCountEmptyDynamicArray`
`Clear`	`TestClearEmptyDynamicArray` `TestClearNonemptyDynamicArray`

Add Method Complexity

Okay, we've avoided the inevitable for long enough. It's time to evaluate the algorithmic complexity for each of the methods we implemented in `UnorderedIntDynamicArray`.

Analyzing the `Add` method is a little trickier than we'd like, because the complexity is different depending on whether or not we need to expand the array. The worst case is certainly when we have to expand the array, but because we hope that expanding the array will be fairly rare, we don't really want to include that extra complexity in the "normal" case. But as we mentioned in the previous reading, we have to do worst case analysis. We'll resolve this by doing 2 analyses: one where the array doesn't have to be expanded and one where the array does need to be expanded.

When we don't need to expand the array, the `Add` operation is O(1). Checking if `count == items.Length`, setting `items[count]` to `item`, and incrementing `count` are all constant time operations, so the complexity is O(1). When we do need to expand the array, the `Add` operation is O(n). The constant time operations are still constant time, but what about the `Expand` method we have to call? There are lots of constant time operations in that method too, but we also have the loop to copy all the elements from the old array to the new one. Because that loop executes n times when we're copying n items from one array to the other one, the loop makes the `Expand` operation O(n). Because the `Add` method calls the `Expand` method in the scenario we're analyzing, the `Add` operation is O(n) in this case.

So, what if someone asks the complexity of the `Add` method? Based on our worst case rules, the answer is O(n), but if they let you explain a little more you should talk about the two different scenarios above.

IndexOf Method Complexity

As we said above, the `IndexOf` method does a standard linear search for the given item. In the worst case, the loop in the method executes n times, so `IndexOf` is O(n).

Remove Method Complexity

The first thing the `Remove` method does is call the `IndexOf` method, so we know the `Remove` operation is at least O(n) (though it could be worse). Whether or not we find the item to be removed, the rest of the method consists of constant time operations, so `Remove` is O(n).

An Ordered Concrete Class

It's time to move on to our `OrderedIntDynamicArray` class, a concrete class for ordered dynamic arrays. The annotated code for that class is:

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

namespace ArraysProject
{
    /// <remarks>
    /// An ordered IntDynamicArray
    /// </remarks>
    public class OrderedIntDynamicArray : IntDynamicArray
    {
        #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        public OrderedIntDynamicArray()
            : base()
        {
        }

        #endregion
```

Just like in the previous concrete class, the constructor just calls the constructor in the abstract class.

```
        #region Public methods

        /// <summary>
        /// Adds the given item to the IntDynamicArray
        /// </summary>
        /// <param name="item">the item to add</param>
        public override void Add(int item)
        {
```

```
        // expand array if necessary
        if (count == items.Length)
        {
            Expand();
        }

        // find location at which to add the item
        int addLocation = 0;
        while ((addLocation < count) &&
            (items[addLocation] < item))
        {
            addLocation++;
        }

        // shift array, add new item and increment count
        ShiftUp(addLocation);
        items[addLocation] = item;
        count++;
    }
```

As in the previous class, the first thing we do is expand the array if necessary. Because this array is ordered, we then have to find the right place to insert the given item so we preserve the sorted order of the array. Once we find the right place, we shift all the elements of the array up one space to make room for the item being added. Finally, we add the item and increment `count`.

```
    /// <summary>
    /// Removes the first occurence of the given item from the
    /// IntDynamicArray
    /// </summary>
    /// <param name="item">the item to remove</param>
    /// <returns>true if the item is successfully removed, false
    ///     otherwise</returns>
    public override bool Remove(int item)
    {
        // check for given item in array
        int itemLocation = IndexOf(item);
        if (itemLocation == -1)
        {
            return false;
        }
        else
        {
            // shift all the elements above the removed one down and
            // shift change count
            ShiftDown(itemLocation + 1);
            count--;
            return true;
        }
    }
```

The first thing this method does is get the location of the item to be removed using the `IndexOf` method (shown and discussed below). If the location is -1, this method returns false, saying the item wasn't removed. If we did find the item to be removed, we shift the

rest of the "real" values in the array down one space and reduce the count of the items in the array by 1.

Notice that we couldn't copy the last item in the array on top of the item to be removed like we did for the unordered array. If we did that here, we'd end up screwing up the sorted order of the array because we'd (at least possibly) be putting a larger element between two smaller elements in the array. Although we could do that anyway, then re-sort the array, that would be less efficient than the approach used above.

```
/// <summary>
/// Determines the index of the given item using binary search
/// </summary>
/// <param name="item">the item to find</param>
/// <returns>the index of the item or -1 if it's not found</returns>
public override int IndexOf(int item)
{
    int lowerBound = 0;
    int upperBound = count - 1;
    int location = -1;

    // loop until found value or exhausted array
    // buggy code to demonstrate defect
    while (location == -1)
    {
        // find the middle
        int middleLocation = lowerBound +
            (upperBound - lowerBound) / 2;
        int middleValue = items[middleLocation];

        // check for match
        if (middleValue == item)
        {
            location = middleLocation;
        }
        else
        {
            // split data set to search appropriate side
            if (middleValue > item)
            {
                upperBound = middleLocation - 1;
            }
            else
            {
                lowerBound = middleLocation + 1;
            }

            // check to see if the array is exhausted
            if (lowerBound > upperBound)
            {
                break;
            }
        }
    }
    return location;
}
```

```
        #endregion
```

For the unordered array, the `IndexOf` method used an O(n) search to find the index of the given item. Because the array here is sorted, though, we can use a binary search (which we know is O(log n)) to find the index of the given item. O(log n) is better than O(n), so that's what we do!

Caution: there's actually a bug in the above code that I talk about at the very end of the reading. It's fine for now, but you shouldn't copy the code above as a perfect example of a binary search.

```
        #region Private methods

        /// <summary>
        /// Shifts all the array elements from the given index to the end of
        /// the array up one space
        /// </summary>
        /// <param name="index">the index at which to start shifting
        ///     up</param>
        void ShiftUp(int index)
        {
            for (int i = count; i > index; i--)
            {
                items[i] = items[i - 1];
            }
        }

        /// <summary>
        /// Shifts all the array elements from the given index to the end of
        /// the array down one space
        /// </summary>
        /// <param name="index">the index at which to start shifting
        ///     down</param>
        void ShiftDown(int index)
        {
            for (int i = index; i < count; i++)
            {
                items[i - 1] = items[i];
            }
        }

        #endregion
    }
}
```

The `ShiftUp` and `ShiftDown` methods are simply helper methods for the `Add` and `Remove` methods, respectively.

Some of our test cases need to be different for our `TestOrderedIntDynamicArray` class (compared to our `TestUnorderedIntDynamicArray` class) for a couple of reasons. We've already tested the `Count` property and the `Clear` method from the abstract class, so we don't need to test them again here. On the other hand, we need to include more test

cases for the `Add` method to make sure we insert the new item in the right place in the array. We test adding an item at the beginning of the array (already included in the `TestAddExpandDynamicArray` case) and at the end of the array and in the middle of the array as well. Obviously, our expected results in the test cases also change since this array is ordered.

Method/Property Tested	Test Cases
Add	TestAddEmptyDynamicArray TestAddExpandDynamicArray TestAddBackOfDynamicArray TestAddInteriorOfDynamicArray
Remove	TestRemoveEmptyDynamicArray TestRemoveItemFrontOfDynamicArray TestRemoveItemBackOfDynamicArray TestRemoveItemInteriorOfDynamicArray TestRemoveItemNotInDynamicArray
IndexOf	TestIndexOfEmptyDynamicArray TestIndexOfFrontOfDynamicArray TestIndexOfBackOfDynamicArray

Add Method Complexity

We know that the `Add` method interacts with the `Expand` (if it needs to grow the array) and `ShiftUp` methods to add an item to the array. We're going to end up with the same complexity whether or not we need to expand the array, so there's no need to think about two separate scenarios like we did for the unordered array. Let's go through the analysis.

First, in the worst case we need to expand the array, so the first part of the algorithm is O(n). Let's look at the linear search to find out where to put the new item and the `ShiftUp` method together. Linear search is O(n) in the worst case, because the item might belong at the end of the array. What does `ShiftUp` do in this case? It doesn't do any shifting at all. That means that the linear search loop and the `ShiftUp` method together are O(n). When we add the O(n) part of the algorithm to expand the array to the O(n) second part, the Add operation overall is O(n).

But wait, you might be thinking, what about the worst case for `ShiftUp`, where it needs to shift the whole array up one space and is therefore O(n)? In this case, the linear search loop only iterates once, so the second part of the `Add` algorithm is still only O(n). The intuition you should build about that part of the method is that, together, the while loop in the `Add` method and the loop in the `ShiftUp` method only iterate n times total (where n is the number of items in the array before we add the new item). Make sure you understand why this is true.

You might also be thinking that we should use a binary search to find out where the new item should be added, since we know binary search is O(log n). Our current binary search finds where a particular item is, though, and returns -1 if it's not in the array, so we'd have to change that method (or write a new one) to return the location where the

item should go if it's not in the array already. That's a little tricky, but we can certainly do it.

The question is "Is it worth it?" In the worst case for this new idea, the binary search will find that the new item should go at the beginning of the array in O(log n) time. We'll then have to shift the entire array up one space, which will take O(n) time. So in the worst case, our new idea still ends up being O(n). The extra complexity in the search algorithm doesn't change the algorithmic complexity of the `Add` operation, so it's probably not worth doing it. In practice, if we knew we were usually going to add things to the end of the array (in other words, if we were adding the data to the array in order) this would help the actual performance of our code, but that's certainly a special case.

IndexOf Method Complexity

As we said above, the `IndexOf` method does a binary search for the given item. In the worst case, `IndexOf` is therefore O(log n).

Remove Method Complexity

The first thing the `Remove` method does is call the `IndexOf` method, so we know the `Remove` operation is at least O(log n) (though it could be worse). If we find the item to be removed, we shift the rest of the "real" values in the array down one space (by calling the `ShiftDown` method) to retain the ordering in the array. What's the worst case for the `ShiftDown` method? If the item to be removed is at the front of the array, the loop in the `ShiftDown` method needs to iterate n – 1 times, making the `ShiftDown` method O(n) in the worst case. That means we have an O(log n) operation (finding the item to be removed) followed by an O(n) operation (removing the item). That makes `Remove` an O(n) operation.

Making a Generic DynamicArray

We limited our initial dynamic arrays to only hold integers so we could focus on the key points for the dynamic array operations and the algorithm analysis for those operations. We could just keep defining more classes, like a `FloatDynamicArray` that stores floats, a `StringDynamicArray` that stores strings, and so on, but that will get old pretty quickly. When we used the classes found in the `System.Collections.Generic` namespace, we actually provided the type we were going to store in the collection, like:

```
List<GameObject> teddyBears = new List<GameObject>();
```

`List` is called a *generic* collection class because it can hold a collection of any data type we want. We can make the data structure classes we build generic as well, so that's what we're going to do to convert our `IntDynamicArray` to a generic `DynamicArray`. We actually don't have to change too much, but rather than providing fragments of code (which can be confusing) we provide the complete new class here, with comments explaining the changes we made:

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

namespace ArraysProject
{
    /// <remarks>
    /// Provides a dynamically-sized array of a data type
    /// </remarks>
    public abstract class DynamicArray<T>
    {
```

We add the `<T>` after the `DynamicArray` class name to indicate that the class can be instantiated to hold any data type. The `T` isn't an actual data type, of course, it's just a placeholder for whatever data type we decide to use when we instantiate the class.

```
        const int ExpandMultiplyFactor = 2;
        protected T[] items;
        protected int count;
```

We need to change our internal array to hold an array of type `T` rather than `int`.

```
        #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        protected DynamicArray()
        {
            items = new T[4];
            count = 0;
        }

        #endregion
```

We create a new array to hold `T` rather than `int` here.

```
        #region Properties

        /// <summary>
        /// Gets the number of elements
        /// </summary>
        public int Count
        {
            get { return count; }
        }

        #endregion

        #region Public methods

        public abstract void Add(T item);
        public abstract bool Remove(T item);
        public abstract int IndexOf(T item);
```

The three abstract method parameters all change to use parameters of type `T` rather than `int`.

```
        /// <summary>
        /// Removes all the items from the DynamicArray
        /// </summary>
        public void Clear()
        {
            count = 0;
        }

        /// <summary>
        /// Converts the DynamicArray to a comma-separated string of values
        /// </summary>
        /// <returns>the comma-separated string of values</returns>
        public override String ToString()
        {
            StringBuilder builder = new StringBuilder();
            for (int i = 0; i < count; i++)
            {
                builder.Append(items[i]);
                if (i < count - 1)
                {
                    builder.Append(",");
                }
            }
            return builder.ToString();
        }

        #endregion

        #region Protected methods

        /// <summary>
        /// Expands the array
        /// </summary>
        protected void Expand()
        {
            T[] newItems = new T[items.Length * ExpandMultiplyFactor];
```

Our expanded array also has to hold `T` elements rather than `int`s.

```
            // copy elements from old array into new array
            for (int i = 0; i < items.Length; i++)
            {
                newItems[i] = items[i];
            }

            // change to use new array
            items = newItems;
        }

        #endregion
    }
}
```

That's actually pretty good – we only had to change 7 lines of code to make our abstract class generic.

There's actually another way to make our dynamic arrays hold any type; we could have changed our `int` array and parameters to `Object` and we could have put any object we wanted into the array. In fact, this is how all the collections in the `System.Collections` namespace work! The problem, though, is that you could add `int`s, `string`s, and any other data type you want to the same collection. While this may be what you need in some cases, in most cases you want your collection to hold a single data type.

Why is our way better? Because once you instantiate our generic dynamic array with a particular data type, the compiler will keep you from adding (or removing or finding the index of) any other data type. We get what's called *strong typing*, where the compiler makes sure we're not mixing data types where we shouldn't. The benefits of strong typing are the main reason the generic collections in the `System.Collections.Generic` namespace were added in .NET 2.0.

The Unordered Generic Concrete Class

Now that we have our generic abstract base class, we can extend it with a concrete class for unordered dynamic arrays; we'll call that class `UnorderedDynamicArray`. The only things that change in this class are the class header, the method headers for the three overridden methods, and one line of code in the `IndexOf` method. The class header changes to

```
public class UnorderedDynamicArray<T> : DynamicArray<T>
```

to make the concrete class a generic, and the method headers change to pass `T` parameters rather than `int`s. The line of code changed in the `IndexOf` method requires a bit more discussion.

In the old version of the `IndexOf`, we used the Boolean expression `items[i] == item` to compare the current element of the array to the item we were looking for. This worked fine for ints because `int` is a value type, but remember how == works on reference types? For reference types, == returns true if the two variables being compared point to the exact same object in memory; it compares the addresses referred to by those variables to see if the addresses are equal. More commonly, we want to compare the values contained in the objects themselves. For example, we'd want to say that the strings "Bob" and "Bob" are the same even if they're stored in two different chunks of memory. That's why we override the `Equals` method for the classes we write if we want more than the address comparison for objects of that class. Boy, that's a long way to say I changed that Boolean expression to `items[i].Equals(item)` because our generic class can now hold reference types as well as value types.

So how do we test our new generic class? The first thing I did was change the object instantiations in all the test cases to work with the generic, like so:

```
UnorderedDynamicArray<int> array = new UnorderedDynamicArray<int>();
```

That way, I could confirm that the generic class still worked the way it used to with ints (it did). I then changed everything to test the class with `String` instead to make sure it works with a reference type (it does). The test cases themselves basically stayed the same, I just changed the data type I was working with.

The Ordered Generic Concrete Class

Just like for the `UnorderedDynamicArray`, in the `OrderedDynamicArray` class we need to change the class header, the method headers for the three overridden methods, and one line of code in the `IndexOf` method. We also have to change code in the `Add` and `IndexOf` methods.

For the loop to find the right location for the new item in our original `Add` method, we originally included the while loop shown below:

```
while ((addLocation < count) &&
    (items[addLocation] < item))
{
    addLocation++;
}
```

The first part of the Boolean expression just makes sure we don't move past the end of the "real" values in the array, and the second part makes sure we keep moving up in the array to find the right place for the new item. The second part is what causes us some trouble in our generic class.

Using < to compare to ints works fine, of course, but < isn't typically available for reference types. We resolve this problem by making sure any reference types we want to include in an `OrderedDynamicArray` implement the `IComparable` interface. The `IComparable` interface requires that we implement a `CompareTo` method that compares the object containing the `CompareTo` method (we'll call it "this object" from now on) 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. Specifically, we change our while loop to:

```
while ((addLocation < count) &&
    (items[addLocation].CompareTo(item) < 0))
{
    addLocation++;
```

```
}
```

The loop will keep iterating while the item at `addLocation` in the array is less than the given item, which is exactly what we need.

So how do we make sure that "any reference types we want to include in an `OrderedDynamicArray` implement the `IComparable` interface?" We actually do this by adding a *constraint* to the class header:

```
public class OrderedDynamicArray<T> : DynamicArray<T> where T:IComparable
```

The constraint (the `where T:IComparable` part of the header) says that any `T` we use to instantiate the `OrderedDynamicArray` class has to implement the `IComparable` interface; otherwise, the compiler will give us an error. Pretty slick, huh? We also could have placed this constraint on the `DynamicArray` class instead, but that would have been the wrong choice because we were able to implement our `UnorderedDynamicArray` class just fine without `T` implementing the `IComparable` interface. We only need this constraint to be satisfied for ordered dynamic arrays, so we only require the constraint on our `OrderedDynamicArray` class.

As mentioned above, we also need to change our `IndexOf` method to use the `CompareTo` method. The changes are shown below, but you can also take a look at the actual code to see them in context.

`int middleValue = items[middleLocation];` changed to
`T middleValue = items[middleLocation];`
`middleValue == item` changed to `middleValue.CompareTo(item) == 0`
`middleValue > item` changed to `middleValue.CompareTo(item) > 0`

I used the same approach to test this new generic class as I did for testing `UnorderedDynamicArray`. I changed the object instantiations in all the test cases to work with the generic, like so:

```
OrderedDynamicArray<int> array = new OrderedDynamicArray<int>();
```

That way, I could confirm that the generic class still worked the way it used to with ints (it did). By the way, I could only do this because `Int32` (the underlying structure for the `int` data type) implements the `IComparable` interface. I then changed everything to test the class with `String` instead to make sure it works with a reference type – and it broke! That's the whole point of test cases, of course – they help us find defects in our code.

When I ran my `TestRemoveEmptyDynamicArray` test case, I got a `NullReferenceException` at the following line of code:

```
if (middleValue.CompareTo(item) == 0)
```

Why did this happen? For an empty array, `upperBound` is set to -1, so `middleLocation` is set to 0 (you should figure out why by looking at the code, taking integer rounding into account). Now we set `middleValue` to `items[middleLocation]`. Because `items` is an array of Strings (we instantiated the class using `OrderedDynamicArray<String>`), which are reference types, and because we never set the 0th element of the array, middleValue is null. When we try to call the `CompareTo` method on the null object, we get the exception.

We have two questions to answer: "Why didn't we see this error for ints?" and "How can we fix it?" We didn't see the error for ints because the default value for an `int` is 0, not null. We could therefore "safely" compare 0 to whatever we were looking for in the array before breaking out of the search loop when lower bound was greater than upper bound. This actually wasn't safe, though! If you go to the `TestOrderedIntDynamicArray` class and change the `TestIndexOfEmptyDynamicArray` method to look for 0 instead of 42, the test case fails because the method thinks it found 0 at location 0 of the array! That just goes to show that testing isn't perfect, and can actually never guarantee that our code is free of defects.

So how do we fix this problem? By changing our while loop Boolean expression from

```
while (location == -1)
```

to

```
while ((location == -1) &&
    (lowerBound <= upperBound))
```

we can remove this defect. That also makes the following if statement unnecessary, so we can remove that as well:

```
if (lowerBound > upperBound)
{
    break;
}
```

Finding and fixing this defect actually made our code a bit easier to read, and that's a good thing too.

Note: I left the defect in the `OrderedIntDynamicArray` class so you could see the test case (looking for 0 in an empty array) fail if you wanted to. I did add comments to the code showing how to fix it like I did in `OrderedDynamicArray`, though.

Conclusion

So there you have it. We built an abstract data structure class (essentially, a collection, though not quite as formal) around a simple `int` array and added the ability to expand that array as necessary. We also provided unordered and ordered concrete

implementations of that abstract class to maintain unordered and ordered arrays. Finally, we extended those integer classes to be generic instead so we could build dynamic arrays that can contain any data type. For the sorted arrays, that data type has to implement the `IComparable` interface to let us order the array properly.

Although arrays are one of the most simple data types, this reading should have convinced you that we can use them to provide pretty interesting data structures.