System.Collections.Generic.SortedSet: who needs another collection anyway?

By Bill Wagner

The .NET 4.0 library includes an additional collection class: The SortedSet<T>. At first glance, it doesn't seem to provide anything new. You may wonder why we need yet another way to store objects. In this article, I'll explain the drivers behind creating a new collection type, how the SortedSet<T> stores its objects.

You'll learn how to make a wise first choice for which collection type you should use, and how to create code that minimizes the impact if later measurements turn out to be incorrect.

Collections are just a place for your stuff (to paraphrase George Carlin). You put objects in collections, and you can retrieve objects from collections. Some of the types support specialized operations. Other differences are because some collections have very different performance characteristics for different operations. The SortedSet<T> was added because its performance characteristics are very different than HashSet<T>. SortedSet<T> is not universally faster than HashSet<T>. Some options will be faster, others will be slower. That's why you must make a smart initial selection, and then continue to benchmark your results.

Understanding Big O notation

The performance characteristics for the collection classes are represented using Big O notation. Big O notation provides some general guidance about how much time particular operations take. That's good because Big O notation will provide you with some guidance about the relative performance characteristics of operations using the different collections classes. Unfortunately, too many people mis-understand what Big O notation describes. That leads some people to misuse it, substituting general rules for actual measurements. A full discussion of Big O notation and a strict definition is outside the scope of one article. It is covered in most computer science texts on algorithms and data structures. I am providing a brief discussion as a refresher.

Big O notation describes the relationship between the number of elements in a collection and the time it takes for a given operation. It represents how the time taken will change based on how many elements are operated upon. For example, an operation that takes the same time regardless of the number of items in a collection is O(1), or "order of 1 time bound". For example, retrieving the first item of an array is O(1). No matter how many elements an array contains, retrieving the first element will take roughly the same time.

In most modern algorithms, you'll find one of five main cases:

O(1). This is constant time bound. The time for an algorithm is unaffected by the number of elements in the collection.

O(ln n). This is logarithmic bound. The time for an algorithm increases with the logarithm of the number of elements in the collection.

O(n). This is linear bound. The time for an algorithm increases linearly with the number of elements in the collection.

O(n ln n). The time for an algorithm to complete increases with the number of elements times the log of the number of elements.

O(n^2). This is quadratic bound. The time for an algorithm to complete increases as the square of the number of elements in the collection.

I've ordered these cases in increasing time *for the most general case*. That means you'll usually want to pick a collection where the most common operations are those that have performance characteristics higher up the list. More importantly, Big O notation describes how an algorithm's performance is affected by the number of elements being operated on. It does not describe the absolute performance characteristics for different operations. It's entirely possible that for your average scenarios, and the number of elements in those scenarios that you a quadratic algorithm will be faster than a linear algorithm. We are discussing performance here, and the only absolute correct answer is to measure it.

The differences between the Sets

Now that we've discussed the basics of Big O notation, you're ready to understand more about why the .NET framework developers added the new SortedSet<T> collection class. The HashSet<T> performs very well for add and search operations. It has O(1) for add in most cases. When the internal capacity must be increased, it is an O(n) operation. Increasing capacity is infrequent. Any search operation (Contains, Remove, and similar operations) are O(1). That's great. However, on the minus side, the HashSet<T> is not a sorted collection. Therefore, enumerating the elements in a sorted order forces you to copy the items to a different collection (like a List<T>) and sort the resulting list. You could construct a LINQ query to order the elements, however internally that query will likely use some form of temporary storage to create the sorted sequence. That means every sort will be an expensive operation. Sort is typically an O(n ln n) operation, Also, because the HashSet<T> does not have a sort method, you'll also have increased memory pressure and time cost to copy the elements.

The SortedSet<T> has different characteristics. The sorted set ensures that the elements in the set are always in sorted order. Every Add operation places the new element in the correct location in the set. That means Add is an O(ln n) operation. The SortedSet<T> must perform a binary search to find the correct location for the new element. The search happens on any of the search actions (Contains, Remove, etc). Those operations also have an O(ln n) performance characteristic.

That sounds like the SortedSet<T> is always slower than the HashSet<T>. No one would use it if it was always slower. SortedSet<T> is much faster for iterating the set in sorted order. It's already in the correct order, so the enumeration becomes an O(n) operation. SortedSet<T> will typically be faster than HashSet<T> when the majority of your operations require enumerating the set in one particular order. If, instead, most of the operations are searching, you'll find better performance using the HashSet<T>. The frequency of insert operations also has an effect on which collection would be better. The more frequently insert operations occur, the more likely HashSet<T> will be faster.

Minimizing Exposure to change

Throughout this article, I've been clearly avoiding specific recommendations on the absolute performance of the different set classes. That's because it can't be done. You'll need to measure as you code your application. Measuring helps you know what changes you should make to improve performance. To leverage that knowledge, you need to structure your application so that you have the freedom to change your mind as the results change with new features.

The sample I created shows how both the number of objects and the type of operation has a large impact on the performance of your application. In order to show both collections with the same operations, you'll also see how to isolate the collection type and make changes later in your development cycle.

The test needs to show the differences in performance as the collection grows in size. That's pretty simple. In addition, I need to show how having a sorted collection affects performance. I wrote two different test methods. The first adds several items to a collection, then searches for items in that collection. The second test adds several items to the collection, and then enumerates the collection in sorted order. In both cases, the tests are repeated 50 times to get a more consistent result. Here's the code for the first test:

```
static TimeSpan RunTestOne(ISet<double> collection, int numItemsToTest)

{

   Stopwatch timer = new Stopwatch();

   timer.Start();

   // perform the test 50 times:

   for (int counter = 0; counter < 50; counter++)

   {

      collection.Clear();

      // add some random items

      Random rnd = new Random();

      var sequence = from n in Enumerable.Range(1, numItemsToTest)

                  select rnd.NextDouble() * n;

      foreach (var item in sequence)
```

```
            collection.Add(item);


        // search for 1000 random items
        var sequence2 = from n in Enumerable.Range(1, 1000)
                        select rnd.NextDouble() * n;
        bool found = false;
        foreach (var item in sequence2)
        {
            found &= collection.Contains(item);
        }
    }
    timer.Stop();
    return timer.Elapsed;
}
```

The code adds a number of items to the set. Then, it performs 1000 searches for random elements in the set. It returns the time elapsed. Notice that the code uses the ISet<T> interface which is implemented by both the HashSet<T> and the SortedSet<T> class. That helps isolate which collection type you are using in your application. The ISet<T> interface is new in .NET 4.0, and provides a common abstraction for both of the HashSet<T> and SortedSet<T> classes.

The results for this first test look like this:

<< Note: This would be good to show in a console output format >>

```
TestOne with Hash Set (100): 00:00:00.0118735
TestOne with Sorted Set (100): 00:00:00.0146180

TestOne with Hash Set (1000): 00:00:00.0088261
TestOne with Sorted Set (1000): 00:00:00.0335345

TestOne with Hash Set (10000): 00:00:00.0626907
TestOne with Sorted Set (10000): 00:00:00.2878814
```

TestOne with Hash Set (50000): 00:00:00.2740288
TestOne with Sorted Set (50000): 00:00:01.8826852

TestOne with Hash Set (100000): 00:00:00.5275904
TestOne with Sorted Set (100000): 00:00:03.8279774

Notice that for small sizes of data, the sorted set and the hash set have almost the same timings. Next, notice that the difference in the timings grows very large as the size of the collection grows. Once you have 100,000 elements the SortedSet takes six times as long to search for items.

Now let's move on to the second test, where I enumerate the collection in a sorted order. The test code requires more work on my part to create a valid test that doesn't do extra work. My goals are to create a test method that can be used with the HashSet<T> and SortedSet<T>, yet leverages the best parts of the SortedSet when possible.

Here's the test method.

```
static TimeSpan RunTestTwo(ISet<double> collection, int numItemsToTest)

{

   Stopwatch timer = new Stopwatch();

   timer.Start();

   // perform the test 50 times:

   for (int counter = 0; counter < 50; counter++)

   {

      collection.Clear();

      // add random items

      Random rnd = new Random();

      var sequence = from n in Enumerable.Range(1, numItemsToTest)

                  select rnd.NextDouble() * n;

      foreach (var item in sequence)

         collection.Add(item);
```

```
    // enumerate 1000 times
    double sum = 0;
    for (int inner = 0; inner < 1000; inner++)
    {

      IEnumerable<double> ordered = collection.IsSorted() ?
      collection as IEnumerable<double> :
      from item in collection
            orderby item
            select item;
      // must enumerate to make it work:
      sum += ordered.Sum();
    }
  }
  timer.Stop();
  return timer.Elapsed;
}
```

I've highlighted the portion of this test that is different from TestOne. Instead of searching for items, I enumerate the items in the list in sorted order. I am performing the sort each time through the loop on purpose, even though it does take more time. I want to measure the cost of the sort because in most applications where order matters, you would need to enumerate the set in different locations in the code, or after items have been added or removed. That requires a sort.

In particular, notice the call to IsSorted(). ISet<T> does not have this method, so I wrote an extension method that tests for a sorted sequence:

```
public static bool IsSorted<T>(this IEnumerable<T> sequence)
   where T : IComparable<T>
{
   // Simplified algorithm for this example:
   // Production code would look for other sorted collections
   // (SortedList, etc)
   if (sequence is SortedSet<T>)
      return true;
   IEnumerator<T> iter = sequence.GetEnumerator();
   if (!iter.MoveNext())
      return true;

   T value = iter.Current;
   while (iter.MoveNext())
   {
      if (value.CompareTo(iter.Current) > 0)
         return false;
      value = iter.Current;
   }
   return true;
}
```

There are two different tests inside this method. The first one performs a quick runtime check to see if the type is already sorted. Production code should check for other collections like SortedList and SortedDictionary. I left those checks out of the sample code for space. If the collection type is not always

sorted, this method performs a linear time check (remember, that would be O(n)) to see if the collection is sorted.

If the sequence is not sorted, and HashSet<T> is not sorted, RunTest() uses a LINQ query to sort the set. LINQ queries execute lazily, so the code must enumerate the results of the LINQ query in order to make the test valid.

Here's the results:

<< Also looking for a console like font here>>

TestTwo with Hash Set (50): 00:00:00.6778654
TestTwo with Sorted Set (50): 00:00:00.1415831

TestTwo with Hash Set (100): 00:00:01.6435130
TestTwo with Sorted Set (100): 00:00:00.2754725

TestTwo with Hash Set (500): 00:00:13.1820688
TestTwo with Sorted Set (500): 00:00:01.3200649

TestTwo with Hash Set (1000): 00:00:32.4777652
TestTwo with Sorted Set (1000): 00:00:02.6566001

TestTwo with Hash Set (5000): 00:03:34.4490727
TestTwo with Sorted Set (5000): 00:00:17.8972095

<<<< End sample output >>>>

Notice that for small sets, there isn't much difference. However, even with only 5000 items in the set, the cost of sorting overwhelms the increased cost of the adding elements. It's a 20 fold increase in time cost. If you are going to enumerate your set often, you will find a much better experience using the new SortedSet<T> class.

Conclusion and Disclaimers

Writing a sample like this is tricky: I want to highlight the differences between these collection styles, but I don't want to lead you to believe that the test code validates all assumptions in all cases. The sample shows the differences in the time cost of these two different types of collections, but that's no excuse to argue dogmatically instead of measuring. It's true that HashSet<T> will be faster than SortedSet<T> on Add and Search operations. It's also true that SortedSet<T> will be faster when you need to enumerate the set in a sorted order. What must be measured is which collection would be better and faster in your application. In some cases, you may find that a design that combines both collection classes provides the needed performance. The only way to know that is to measure typical scenarios in your application using both collection types. The only technique I want you to recognize is universal is to enable changing the collection types in your application. By writing your code using interfaces, hopefully only on the particular functionality you need to create your application. You may even need to write some utility methods to enable cheaper tests instead of blindly performing unneeded work.

There is a reason the .NET BCL team made so many collections that seem so similar. I've showed you how to evaluate the differences, and how to defer a decision until as late as possible in your development process.

- The source for this article can be found on the LinqFarm
- Here is a direct link to the source.

© 2014 Microsoft