An Extensive Examination of Data Structures Using C# 2.0

Visual Studio 2005

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: This article, the second in a six-part series on data structures in the .NET Framework, examines three of the most commonly studied data structures: the Queue, the Stack, and the Hashtable. As we'll see, the Queue and Stack are specialized Lists, providing storage for a variable number of objects, but restricting the order in which the items may be accessed. The Hashtable provides an array-like abstraction with greater indexing flexibility. Whereas an array requires that its elements be indexed by an ordinal value, Hashtables allow items to be indexed by any type of object, such as a string. (19 printed pages)

> **Editor's note** This six-part article series originally appeared on MSDN Online starting in November 2003. In January 2005 it was updated to take advantage of the new data structures and features available with the .NET Framework version 2.0, and C# 2.0. The original articles are still available at http://msdn.microsoft.com/vcsharp/default.aspx?pull=/library/en-us/dv_vstechart/html/datastructures_guide.asp.

> **Note** This article assumes the reader is familiar with C#.

Contents

Introduction

In Part 1 of An Extensive Examination of Data Structures, we looked at what data structures are, how their performance can be evaluated, and how these performance considerations play into choosing which data structure to utilize for a particular algorithm. In addition to reviewing the basics of data structures and their analysis, we also looked at the most commonly used data structure, the array.

The array holds a set of homogeneous elements indexed by ordinal value. The actual contents of an array are laid out as a contiguous block, thereby making reading from or writing to a specific array element very fast. In addition to the standard array, the .NET Framework Base Class Library offers the **List** class. Like the array, the **List** is a collection of homogeneous data items. With a **List**, you don't need to worry about resizing or capacity limits, and there are numerous **List** methods for searching, sorting, and modifying the **List**'s data. As discussed in the previous article, the **List** class uses Generics to provide a type-safe, reusable collection data structure.

In this second installment of the article series, we'll continue our examination of array-like data structures by first examining the Queue and Stack. These two data structures are similar in some aspects to the **List**—they both are implemented using Generics to contain a type-safe collection of data items. The Queue and Stack differ from the **List** class in that there are limitations on how the Queue and Stack data can be accessed.

Following our look at the Queue and Stack, we'll spend the rest of this article digging into the Hashtable data structure. A *Hashtable*, which is sometimes referred to as an associative array, stores a collection of elements, but indexes these elements by an arbitrary object (such as a string), as opposed to an ordinal index.

Providing First Come, First Served Job Processing

If you are creating any kind of computer service—that is, a computer program that can receive multiple requests from multiple sources for some task to be completed—then part of the challenge of creating the service is deciding the order in which the incoming requests will be handled. The two most common approaches used are:

- First come, first served
- Priority-based processing

First come, first served is the job-scheduling task you'll find at your grocery store, the bank, and licensing departments. Those waiting for service stand in a line. The people in front of you will be served before you while the people behind you will be served after. Priority-based processing serves those with a higher priority before those with a lesser priority. For example, a hospital emergency room uses this strategy, opting to help someone with a potentially fatal wound before someone with a less threatening wound, regardless of who arrived first.

Imagine that you need to build a computer service and that you want to handle requests in the order in which they were received. Because the number of incoming requests might happen quicker than you can process them, you'll need to place the requests in some sort of buffer that can preserve the order in which they arrived.

One option is to use a List and an integer variable called `nextJobPos` to indicate the position of the next job to be completed. When each new job request comes in, simply use the List's `Add()` method to add it to the end of the List. Whenever you are ready to process a job in the buffer, grab the job at the `nextJobPos` position in the List and increment `nextJobPos`. The following simple program illustrates this algorithm:

```
public class JobProcessing
{
   private static List<string> jobs = new List<string>(16);
   private static int nextJobPos = 0;

   public static void AddJob(string jobName)
   {
      jobs.Add(jobName);
   }
```

```
   public static string GetNextJob()
   {
      if (nextJobPos > jobs.Count - 1)
         return "NO JOBS IN BUFFER";
      else
      {
         string jobName = jobs[nextJobPos];
         nextJobPos++;
         return jobName;
      }
   }

   public static void Main()
   {
      AddJob("1");
      AddJob("2");
      Console.WriteLine(GetNextJob());
      AddJob("3");
Console.WriteLine(GetNextJob());
      Console.WriteLine(GetNextJob());
      Console.WriteLine(GetNextJob());
      Console.WriteLine(GetNextJob());
      AddJob("4");
      AddJob("5");
      Console.WriteLine(GetNextJob());
   }
}
```

The output of this program is as follows:

```
1
2
3
NO JOBS IN BUFFER
NO JOBS IN BUFFER
4
```

While this approach is fairly simply and straightforward, it is horribly inefficient. For starters, the List will continue to grow unabated with each job that's added to the buffer, even if the jobs are processed immediately after being added to the buffer. Consider the case where every second a new job is added to the buffer and a job is removed from the buffer. This means that once a second the `AddJob()` method is called, which calls the List's `Add()` method. As the `Add()` method is continually called, the List's internal array's size is continually redoubled as needed. After five minutes (300 seconds) the List's internal array will be dimensioned for 512 elements, even though there has never been more than one job in the buffer at a time. This trend, of course, will continue so long as the program continues to run and the jobs continue to come in.

The reason the List grows in such ridiculous proportions is because the buffer locations used for old jobs are not reclaimed. That is, when the first job is added to the buffer, and then processed, clearly the first spot in the List is ready to be reused again. Consider the job schedule presented in the previous code sample. After the first two lines—`AddJob("1")` and `AddJob("2")`—the List will look like Figure 1.

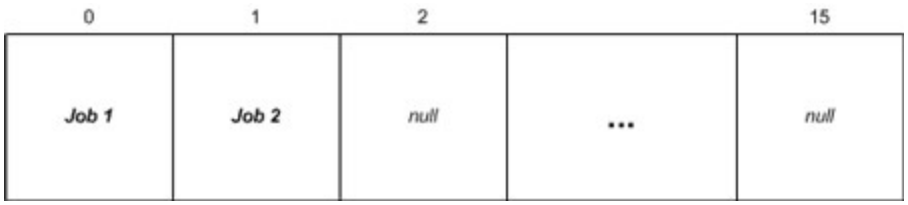

Figure 1. The ArrayList after the first two lines of code

Note that there are 16 elements in the List at this point because the List was initialized with a capacity of 16 in the code above. Next, the `GetNextJob()` method is invoked, which removes the first job, resulting in Fiure 2.

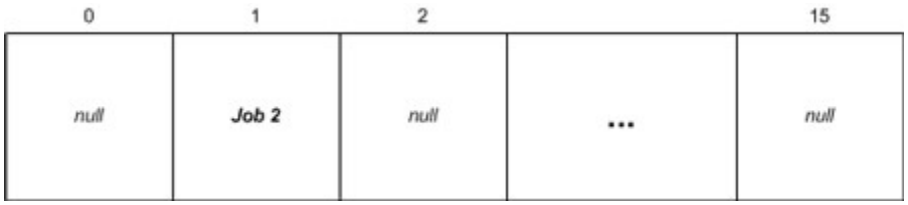

Figure 2. Program after the GetNextJob() method is invoked

When `AddJob("3")` executes, we need to add another job to the buffer. Clearly the first List element (index 0) is available for reuse. Initially it might make sense to put the third job in the 0 index. However, this approach can be eliminated by considering what would happen if after `AddJob("3")` we did `AddJob("4")`, followed by two calls to `GetNextJob()`. If we placed the third job in the 0 index and then the fourth job in the 2 index, we'd have something like the problem displayed in Figure 3.

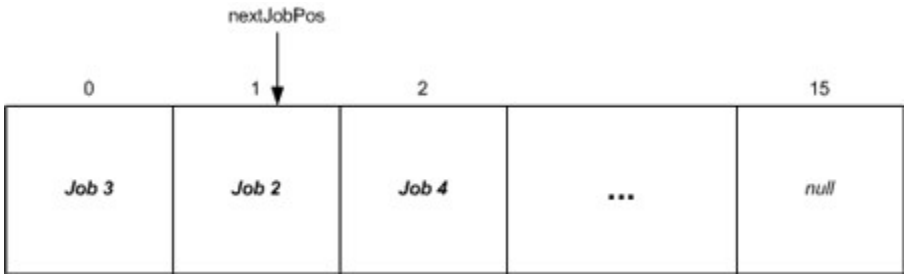

Figure 3.Issue created by placing jobs in the O index

Now, when `GetNextJob()` was called, the second job would be removed from the buffer, and `nextJobPos` would be incremented to point to index 2. Therefore, when `GetNextJob()` was called again, the *fourth* job would be removed and processed prior to the third job, thereby violating the first come, first served order we need to maintain.

The crux of this problem arises because the List represents the list of jobs in a linear ordering. That is, we need to keep adding the new jobs to the right of the old jobs to guarantee that the correct processing order is maintained. Whenever we hit the end of the List, the List is doubled, even if there are unused List elements due to calls to `GetNextJob()`.

To fix this problem, we need to make our List *circular*. A circular array is one that has no definite start or end. Rather, we have to use variables to remember the beginning and end positions of the array. A graphical representation of a circular array is shown in Figure 4.

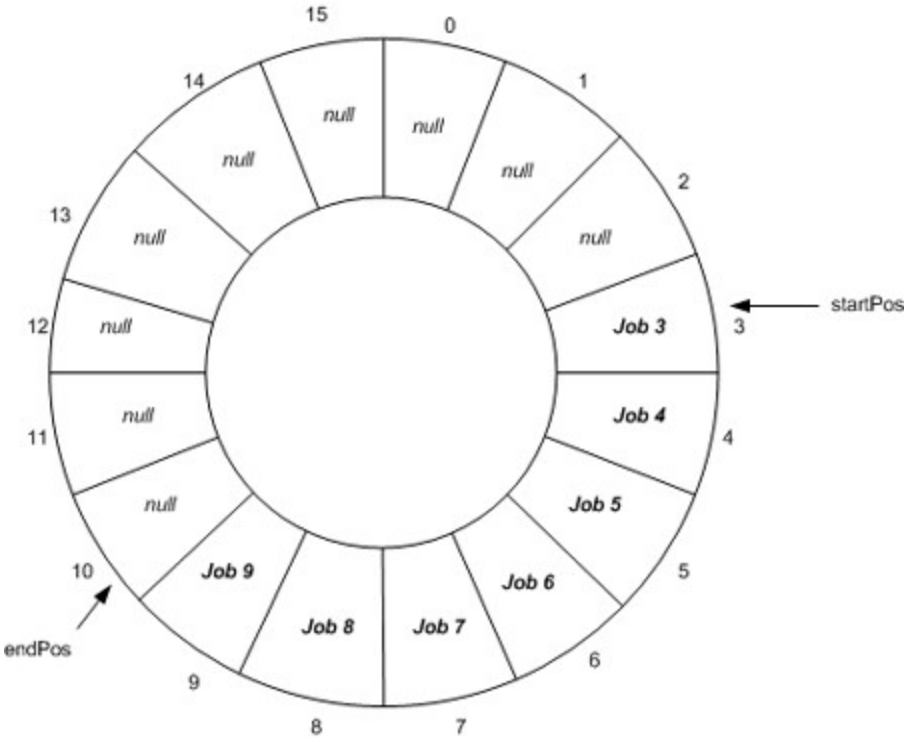

Figure 4. Example of a circular array

With a circular array, the `AddJob()` method adds the new job in index `endPos` and then "increments" `endPos`. The `GetNextJob()` method plucks the job from `startPos`, sets the element at the `startPos` index to `null`, and "increments" `startPos`. I put the word increments in quotation marks because here incrementing is a trifle more complex than simply adding one to the variable's current value. To see why we can't just add 1, consider the case when `endPos` equals 15. If we increment `endPos` by adding 1, `endPos` will equal 16. In the next `AddJob()` call, the index 16 will attempt to be accessed, which will result in an `IndexOutOfRangeException`.

Rather, when `endPos` equals 15, we want to increment `endPos` by resetting it to 0. This can either be done by creating an `increment(`*`variable`*`)` function that checks to see if the passed-in variable equals the array's size and, if so, reset it to 0. Alternatively, the variable can have its value incremented by 1 and then `mod`-ed by the size of the array. In such a case, the code for `increment()` would look like:

```
int increment(int variable)
{
  return (variable + 1) % theArray.Length;
}
```

> **Note** The modulus operator, `%`, when used like `x % y`, calculates the remainder of `x` divided by `y`. The remainder will always be between 0 and `y - 1`.

This approach works well if our buffer will never have more than 16 elements, but what happens if we wish to add a new job to the buffer when there's already 16 jobs present? Like with the List's `Add()` method, we'll need to resize the circular array appropriately by, say, doubling the size of the array.

The System.Collections.Generic.Queue Class

The functionality we have just described—adding and removing items to a buffer in first come, first served order while maximizing space utilization—is provided in a standard data structure, the Queue. The .NET Framework Base Class Library provides the System.Collections.Generic.Queue class, which uses Generics to provide a type-safe Queue implementation. Whereas our earlier code provided `AddJob()` and `GetNextJob()` methods, the **Queue** class provides identical functionality with its `Enqueue(`*`item`*`)` and `Dequeue()` methods, respectively. Behind the scenes, the **Queue** class maintain an internal circular array and two variables that serve as markers for the beginning and ending of the circular array: `head` and `tail`.

The `Enqueue()` method starts by determining if there is sufficient capacity for adding the new item to the queue. If so, it merely adds the element to the circular array at the `tail` index, and then "increments" `tail` using the modulus operator to ensure that `tail` does not exceed the internal array's length. If, however, there is insufficient space, the array is increased by a specified growth factor. This growth factor has a default value of 2.0, thereby doubling the internal array's size, but you can optionally specify this factor in the Queue class's constructor.

The `Dequeue()` method returns the current element from the head index. It also sets the `head` index element to null and "increments" `head`. For those times where you may want to look at the head element, but not actually dequeue it, the **Queue** class also provides a `Peek()` method.

What is important to realize is that the Queue, unlike the List, does not allow random access. That is, you cannot look at the third item in the queue without dequeing the first two items. However, the **Queue** class does have a `Contains()` method, so you can determine whether or not a specific item exists in the Queue. There's also a `ToArray()` method that returns an array containing the Queue's elements. If you know you will need random access, though, the Queue is not the data structure to use—the List is. The Queue is, however, ideal for situations where you are only interested in processing items in the precise order with which they were received.

> **Note** You may hear Queues referred to as FIFO data structures. FIFO stands for First In, First Out, and is synonymous to the processing order of first come, first served.

A Look at the Stack Data Structure: First Come, Last Served

The Queue data structure provides first come, first served access by internally using a circular array of type `object`. The Queue provides such access by exposing an `Enqueue()` and `Dequque()` methods. First come, first serve processing has a number of real-world applications, especially in service programs like Web servers, print queues, and other programs that handle multiple incoming requests.

Another common processing scheme in computer programs is first come, *last* served. The data structure that provides this form of access is known as a Stack. The .NET Framework Base Class Library includes a **Stack** class in the `Sytem.Collections.Generic` namespace. Like the **Queue** class, the **Stack** class maintains its elements internally using a circular array. The **Stack** class exposes its data through two methods: `Push(`*`item`*`)`, which adds the passed-in item to the stack, and `Pop()`, which removes and returns the item at the top of the stack.

A Stack can be visualized graphically as a vertical collection of items. When an item is pushed onto the stack, it is placed on top of all other items. Popping an item removes the item from the top of the stack. The following two figures graphically represent a stack first after items 1, 2, and 3 have been pushed onto the stack in that order, and then after a pop.

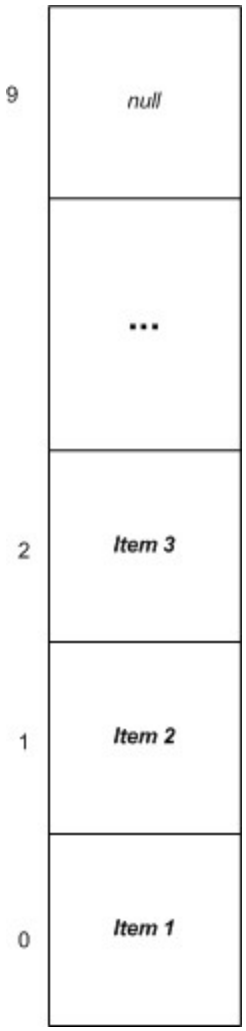

Figure 5. Graphical representation of a stack with three items

Figure 6. Graphical representation of a stack with three items after a pop

Like the List, when the Stack's internal array needs to be resized it is automatically increased by twice the initial size. (Recall that with the Queue this growth factor can be optionally specified through the constructor.)

> **Note** Stacks are often referred to as LIFO data structures, or Last In, First Out.

Stacks: A Common Metaphor in Computer Science

When talking about queues it's easy to conjure up many real-world parallels like lines at the bakery, printer job processing, and so on. But real-world examples of stacks in action are harder to come up with. Despite this, stacks are a prominent data structure in a variety of computer applications.

For example, consider any imperative computer programming language, like C#. When a C# program is executed, the CLR maintains a *call stack* which, among other things, keeps track of the function invocations. Each time a function is called, its information is added to the call stack. Upon the function's completion, the associated information is popped from the stack. The information at the top of the call stack represents the current function being executed. (For a visual demonstration of the function call stack, create a project in Visual Studio .NET, set a breakpoint and go to Debug/Start. When the breakpoint hits, display the Call Stack window from Debug/Windows/Call Stack.)

Stacks are also commonly used in parsing grammars (from simple algebraic statements to computer programming languages), as a means to simulate recursion, and even as an instruction execution model.

The Limitations of Ordinal Indexing

Recall from Part 1 of this article series that the hallmark of the array is that it offers a homogeneous collection of items *indexed by an ordinal value*. That is, the i^{th} element of an array can be accessed in constant time for reading or writing. (Recall that constant-time was denoted as *O*(1).)

Rarely do we know the ordinal position of the data we are interested in, though. For example, consider an employee database. Employees might be uniquely identified by their social security number, which has the form DDD-DD-DDDD, where D is a digit (0-9). If we had an array of all employees that were randomly ordered, finding employee 111-22-3333 would require, potentially, searching through *all* of the elements in the employee array, a *O*(*n*) operation. A somewhat better approach would be to sort the employees by their social security numbers, which would reduce the asymptotic search time down to *O*(log *n*).

Ideally, we'd like to be able to do is access an employee's records in *O*(1) time. One way to accomplish this would to build a *huge* array, with an entry for each possible social security number value. That is, our array would start at element 000-00-0000 and go to element 999-99-9999, as shown in Figure 7.

	Name	Phone	Salary	Dept.
000-00-0000				
...	...			
455-11-0189	Scott Mitchell	333-4444	$134,500	Sales
455-11-0190				
455-11-0191	Jisun Lee	555-6666	$196,750	Exec.
...	...			
999-99-9999				

Figure 7. Array showing all possible elements for a 9-digit number

As this figure shows, each employee record contains information like Name, Phone, Salary, and so on, and is indexed by the employee's social security number. With such a scheme, any employee's information could be accessed in constant time. The disadvantage of this approach is its extreme waste: there are a total of 10^9—that's one *billion* (1,000,000,000)—different social security numbers. For a company with 1,000 employees, only 0.0001% of this array would be utilized. (To put things in perspective, your company would have to employ about one-sixth of the world's population in order to make this array near fully utilized.)

Compressing Ordinal Indexing with a Hash Function

Creating a one billion element array to store information about 1,000 employees is clearly unacceptable in terms of space. However, the performance of being able to access an employee's information in constant time is highly desirable. One option would be to reduce the social security number span by only using the last four digits of an employee's social security number. That is, rather than having an array spanning from 000-00-0000 to 999-99-9999, the array would only span from 0000 to 9999. Figure 8 below shows a graphical representation of this trimmed-down array.

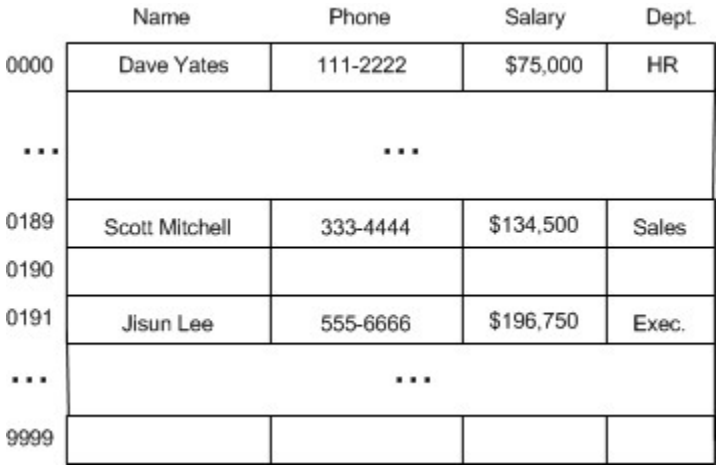

Figure 8. Trimmed down array

This approach provides both the constant time lookup cost, as well as much better space utilization. Choosing to use the last four digits of the social security number was an arbitrary choice. We could have used the middle four digits, or the first, third, eighth, and ninth.

The mathematical transformation of the nine-digit social security number to a four-digit number is called *hashing*. An array that uses hashing to compress its indexers space is referred to as a *hash table*.

A *hash function* is a function that performs this hashing. For the social security number example, our hash function, *H*, can be described as follows:

```
H(x) = last four digits of x
```

The inputs to *H* can be any nine-digit social security number, whereas the result of *H* is a four-digit number, which is merely the last four digits of the nine-digit social security number. In mathematical terms, *H* maps elements from the set of nine-digit social security numbers to elements from the set of four-digit social security numbers, as shown graphically in the Figure 9.

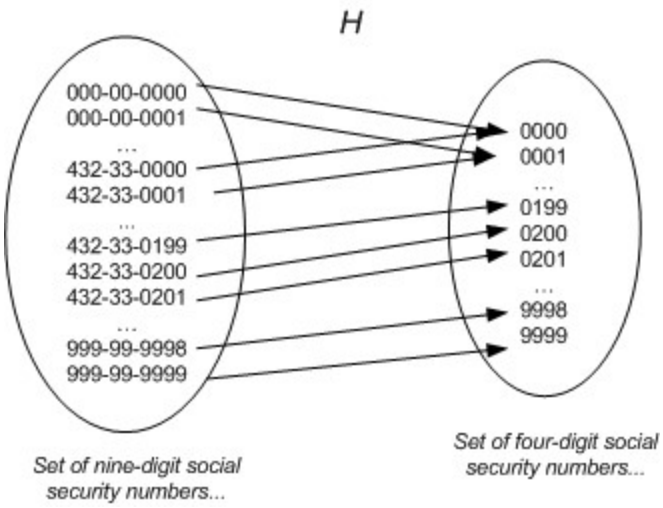

Figure 9. Graphical representation of a hash function

The above figure illustrates a behavior exhibited by hashing functions called *collisions*. In general, with hashing functions you will be able to find two elements in the larger set that map to the same value in the smaller set. With our social security number hashing function, all social security numbers ending in 0000 will map to 0000. That is, the hash value for 000-00-0000, 113-14-0000, 933-66-0000 and many others will all be 0000. (In fact, there will be precisely 10^5, or 100,000, social security numbers that end in 0000.)

To put it back into the context of our earlier example, consider what would happen if a new employee was added with social security number 123-00-0191. Attempting to add this employee to the array would cause a problem because there already exists an employee at array location 0191 (Jisun Lee).

> **Mathematical Note** A hashing function can be described in more mathematically precise terms as a function f : A -> B. Because |A| > |B| it must be the case that f is not one-to-one; therefore, there will be collisions.

Clearly the occurrence of collisions can cause problems. In the next section, we'll look at the correlation between the hash function and the occurrence of collisions and then briefly examine some strategies for handling collisions. In the section after that, we'll turn our attention to the `System.Collections.Hashtable` class, which provides an implementation of a hash table. We'll look at the Hashtable class's hash function, collision resolution strategy, and some examples of using the **Hashtable** class in practice. Following a look at the **Hashtable** class, we'll study the **Dictionary** class, which was added to the .NET Framework 2.0 Base Class Library. The **Dictionary** class is identical to the Hashtable, save for two differences:

- It uses Generics and is therefore strongly-typed.
- It employs an alternate collision resolution strategy.

Collision Avoidance and Resolution

When adding data to a hash table, a collision throws a monkey wrench into the entire operation. Without a collision, we can add the inserted item into the hashed location; with a collision, however, we must decide upon some corrective course of action. Due to the increased cost associated with collisions, our goal should be to have as few collisions as possible.

The frequency of collisions is directly correlated to the hash function used and the distribution of the data being passed into the hash function. In our social security number example, using the last four digits of an employee's social security number is an ideal hash function assuming that social security numbers are randomly assigned. However, if social security numbers are assigned such that those born in a particular year or location are more likely to have the same last four digits, then using the last four digits might cause a large number of collisions if your employees' birth dates and birth locations are not uniformly distributed.

> **Note** A thorough analysis of a hash functions value requires a bit of experience with statistics, which is beyond the scope of this article. Essentially, we want to ensure that for a hash table with k slots, the probability that a random value from the hash function's domain will map to any particular element in the range is 1/k.

Choosing an appropriate hash function is referred to as *collision avoidance*. Much study has gone into this field, as the hash function used can greatly impact the overall performance of the hash table. In the upcoming sections, we'll look the hash function used by the **Hashtable** and **Dictionary** classes in the .NET Framework.

In the case of a collision, there are a number of strategies that can be employed. The task at hand, *collision resolution*, is to find some other place to put the object that is being inserted into the hash table because the actual location was already taken. One of the simplest approaches is called *linear probing* and works as follows:

1. When a new item is inserted into the hash table, use the hash function to determine where in the table it belongs.
2. Check to see if an element already exists in that spot in the table. If the spot is empty, place the element there and return, otherwise go to step 3.
3. If the location the hash function pointed to was location *i*, simply check location *i* + 1 to see if that is available. If it is also taken, check *i* + 2, and so on, until an open spot is found.

Consider the case where the following four employees were inserted into the hash table: Alice (333-33-1234), Bob (444-44-1234), Cal (555-55-1237), Danny (000-00-1235), and Edward (111-00-1235). After these inserts the hash table will look like:

	Name	Phone	Salary	Dept.
0000				
...	...			
1234	Alice	...	...	...
1235	Bob	...	...	...
1236	Danny	...	...	...
1237	Cal	...	...	...
1238	Edward	...	...	...
...	...			
9999				

Figure 10. Hash table of four employees with similar numbers

Alice's social security number is hashed to 1234, and she is inserted at spot 1234. Next, Bob's social security number is hashed to 1234, but Alice is already at spot 1234, so Bob takes the next available spot, which is 1235. After Bob, Cal is inserted, his value hashing to 1237. Because no one is currently occupying 1237, Cal is inserted there. Danny is next, and his social security number is hashed to 1235. 1235 is taken, 1236 is checked, and because 1236 is open, Danny is placed there. Finally, Edward is inserted, his social security number also hashing to 1235. 1235 is taken, so 1236 is checked. That's taken too, so 1237 is checked. That's occupied by Cal, so 1238 is checked, which is open, so Edward is placed there.

In addition to gumming up the insertion process, collisions also present a problem when searching a hash table. For example, given the hash table in Figure 10, imagine we wanted to access information about Edward. Therefore, we take Edward's social security number, 111-00-1235, hash it to 1235, and start our search there. However, at spot 1235 we find Bob, not Edward. So we have to check 1236, but Danny's there. Our linear search continues until we either find Edward or hit an empty slot. If we reach an empty spot, we know that Edward is not in our hashtable.

Linear probing, while simple, is not a very good collision resolution strategy because it leads to *clustering*. That is, imagine that the first 10 employees we insert all have the social security hash to the same value, say 3344. Then 10 consecutive spots will be taken, from 3344 through 3353. This cluster requires linear probing any time any one of these 10 employees is accessed. Furthermore, any employees with hash values from 3345 through 3353 will add to this cluster's size. For speedy lookups, we want the data in the hash table uniformly distributed, not clustered around certain points.

A more involved probing technique is *quadratic probing*, which starts checking spots a quadratic distance away. That is, if slot *s* is taken, rather than checking slot $s + 1$, then $s + 2$, and so on as in linear probing, quadratic probing checks slot $s + 1^2$ first, then $s - 1^2$, then $s + 2^2$, then $s - 2^2$, then $s + 3^2$, and so on. However, even quadratic hashing can lead to clustering.

In the next section, we'll look at a third collision resolution technique called *rehasing*, which is the technique used by the .NET Framework's **Hashtable** class. In the final section, we'll look at the **Dictionary** class, which uses a collision resolution technique knows as *chaining*.

The System.Collections.Hashtable Class

The .NET Framework Base Class Library includes an implementation of a hash table in the **Hashtable** class. When adding an item to the Hashtable, you must provide not only the item, but the unique key by which the item is accessed. Both the key and item can be of any type. In our employee example, the key would be the employee's social security number. Items are added to the Hashtable using the `Add()` method.

To retrieve an item from the Hashtable, you can index the Hashtable by the key, just like you would index an array by an ordinal value. The following short C# program demonstrates this concept. It adds a number of items to a Hashtable, associating a string key with each item. Then, the particular item can be accessed using its string key.

```
using System;
using System.Collections;

public class HashtableDemo
{
    private static Hashtable employees = new Hashtable();

    public static void Main()
    {
```

```
        // Add some values to the Hashtable, indexed by a string key
        employees.Add("111-22-3333", "Scott");
        employees.Add("222-33-4444", "Sam");
        employees.Add("333-44-55555", "Jisun");

        // Access a particular key
        if (employees.ContainsKey("111-22-3333"))
        {
            string empName = (string) employees["111-22-3333"];
            Console.WriteLine("Employee 111-22-3333's name is: " + empName);
        }
        else
            Console.WriteLine("Employee 111-22-3333 is not in the hash table...");
    }
}
```

This code also demonstrates the `ContainsKey()` method, which returns a Boolean indicating whether or not a specified key was found in the Hashtable. The Hashtable class contains a `Keys` property that returns a collection of the keys used in the Hashtable. This property can be used to enumerate the items in a Hashtable, as shown below:

```
// Step through all items in the Hashtable
foreach(string key in employees.Keys)
    Console.WriteLine("Value at employees[\"" + key + "\"] = " + employees[key].ToString());
```

Realize that the order with which the items are inserted and the order of the keys in the `Keys` collection are not necessarily the same. The ordering of the `Keys` collection is based on the slot the key's item was stored. The slot an item is stored depends on the key's hash value and collision resolution strategy. If you run the above code you can see that the order the items are enumerated doesn't necessarily match with the order with which the items were added to the Hashtable. Running the above code outputs:

```
Value at employees["333-44-5555"] = Jisun
Value at employees["111-22-3333"] = Scott
Value at employees["222-33-4444"] = Sam
```

Even though the data was inserted into the Hashtable in the order "Scott," "Sam," "Jisun."

The Hashtable Class's Hash Function

The hash function of the Hashtable class is a bit more complex than the social security number hash code we examined earlier. First, keep in mind that the hash function must return an ordinal value. This was easy to do with the social security number example since the social security number is already a number itself. To get an appropriate hash value, we merely chopped off all but the final four digits. But realize that the Hashtable class can accept a key of *any* type. As we saw in a previous example, the key could be a string, like "Scott" or "Sam." In such a case, it is only natural to wonder how a hash function can turn a string into a number.

This magical transformation can occur thanks to the `GetHashCode()`, which is defined in the `System.Object` class. The `Object` class's default implementation of `GetHashCode()` returns a unique integer that is guaranteed not to change during the lifetime of the object. Because every type is derived, either directly or indirectly, from `Object`, all objects have access to this method. Therefore, a string, or any other type, can be represented as a unique number. Of course, this method can be overridden to provide a hash function more suitable to a specific class. (The

`Point` class in the `System.Drawing` namespace, for example, overrides `GetHashCode()`, returning the XOR of its `x` and `y` member variables.)

The Hashtable class's hash function is defined as follows:

$$H(key) = [GetHash(key) + 1 + (((GetHash(key) >> 5) + 1) \% (hashsize - 1))] \% hashsize$$

Here, GetHash(*key*) is, by default, the result returned by *key*'s call to `GetHashCode()` (although when using the Hashtable you can specify a custom `GetHash()` function). GetHash(*key*) >> 5 computes the hash for *key* and then shifts the result 5 bits to the right – this has the effect of dividing the hash result by 32. As discussed earlier in this article, the `%` operator performs modular arithmetic. `hashsize` is the number of total slots in the hash table. (Recall that $x \% y$ returns the remainder of x / y, and that this result is always between 0 and $y - 1$.) Due to these mod operations, the end result is that *H*(*key*) will be a value between 0 and *hashsize* – 1. Since *hashsize* is the total number of slots in the hash table, the resulting hash will always point to within the acceptable range of values.

Collision Resolution in the Hashtable Class

Recall that when inserting an item into or retrieving an item from a hash table, a collision can occur. When inserting an item, an open slot must be found. When retrieving an item, the actual item must be found if it is not in the expected location. Earlier we briefly examined two collusion resolution strategies:

- Linear probing
- Quardratic probing

The **Hashtable** class uses a different technique referred to as *rehasing*. (Some sources refer to rehashing as *double hashing*.)

Rehasing works as follows: there is a set of hash different functions, H_1 ... H_n, and when inserting or retrieving an item from the hash table, initially the H_1 hash function is used. If this leads to a collision, H_2 is tried instead, and onwards up to H_n if needed. The previous section showed only one hash function, which is the initial hash function (H_1). The other hash functions are very similar to this function, only differentiating by a multiplicative factor. In general, the hash function H_k is defined as:

$$H_k(key) = [GetHash(key) + k * (1 + (((GetHash(key) >> 5) + 1) \% (hashsize - 1)))] \% hashsize$$

> **Mathematical Note** With rehasing it is important that each slot in the hash table is visited exactly once when `hashsize` number of probes are made. That is, for a given key you don't want H_i and H_j to hash to the same slot in the hash table. With the rehashing formula used by the **Hashtable** class, this property is maintained if the result of (1 + (((GetHash(key) >> 5) + 1) % (hashsize – 1)) and `hashsize` are relatively prime. (Two numbers are relatively prime if they share no common factors.) These two numbers are guaranteed to be relatively prime if `hashsize` is a prime number.

Rehasing provides better collision avoidance than either linear or quadratic probing.

Load Factors and Expanding the Hashtable

The **Hashtable** class contains a private member variable called `loadFactor` that specifies the maximum ratio of items in the Hashtable to the total number of slots. A `loadFactor` of, say, 0.5, indicates that at most the Hashtable can only have half of its slots filled with items and the other half must remain empty.

In an overloaded form of the Hashtable's constructor, you can specify a `loadFactor` value between 0.1 and 1.0. Realize, however, that whatever value you provide, it is scaled down 72%, so even if you pass in a value of 1.0 the Hashtable class's actual `loadFactor` will be 0.72. The 0.72 was found by Microsoft to be the optimal load factor, so consider using the default 1.0 load factor value (which gets scaled automatically to 0.72). Therefore, you would be encouraged to use the default of 1.0 (which is really 0.72).

> **Note** I spent a few days asking various listservs and folks at Microsoft why this automatic scaling was applied. I wondered why, if they wanted to values to be between 0.072 and 0.72, why not make that the legal range? I ended up talking to the Microsoft team that worked on the Hashtable class and they shared their reason for this decision. Specifically, the team found through empirical testing that values greater than 0.72 seriously degraded the performance. They decided that the developer using the Hashtable would be better off if they didn't have to remember a seeming arbitrary value in 0.72, but instead just had to remember that a value of 1.0 gave the best results. So this decision, essentially, sacrifices functionality a bit, but makes the data structure easier to use and will cause fewer headaches in the developer community.

Whenever a new item is added to the Hashtable class, a check occurs to make sure adding the new item won't push the ratio of items to slots past the specified maximum ratio. If it will, then the Hashtable is *expanded*. Expansion occurs in two steps:

1. The number of slots in the Hashtable is approximately doubled. More precisely, the number of slots is increased from the current prime number value to the next largest prime number value in an internal table. Recall that for rehashing to work properly, the number of hash table slots needs to be a prime number.
2. Because the hash value of each item in the hash table is dependent on the number of total slots in the hash table, all of the values in the hash table need to be rehashed (because the number of slots increased in step 1).

Fortunately the **Hashtable** class hides all this complexity in the `Add()` method, so you don't need to be concerned with the details.

The load factor influences the overall size of the hash table and the expected number of probes needed on a collision. A high load factor, which allows for a relatively dense hash table, requires less space but more probes on collisions than a sparsely dense hash table. Without getting into the rigors of the analysis, the expected number of probes needed when a collision occurs is at most 1 / (1 – *lf*), where *lf* is the load factor.

As aforementioned, Microsoft has tuned the Hashtable to use a default load factor of 0.72. Therefore, for you can expect on average 3.5 probes per collision. Because this estimate does not vary based on the number of items in the Hashtable, the asymptotic access time for a Hashtable is *O*(1), which beats the pants off of the *O*(*n*) search time for an array.

Finally, realize that expanding the Hashtable is not an inexpensive operation. Therefore, if you have an estimate as to how many items you're Hashtable will end up containing, you should set the Hashtable's initial capacity accordingly in the constructor so as to avoid unnecessary expansions.

The System.Collections.Generic.Dictionary Class

The Hastable is a loosely-typed data structure, because a developer can add keys and values to the Hashtable of any type. As we've seen with the **List** class, as well as variants on the **Queue** and **Stack** classes, with the introduction of Generics in the .NET Framework 2.0, many of the built-in data structures have been updated to provide type-safe versions using Generics. The **Dictionary** class is a type-safe Hashtable implementation, and strongly types both the

keys and values. When creating a Dictionary instance, you must specify the data types for both the key and value, using the following syntax:

```
Dictionary<keyType, valueType> variableName = new Dictionary<keyType, valueType>();
```

Returning to our earlier example of storing employee information using the last four digits of the social security as a hash, we might create a Dictionary instance whose key was of type integer (the nine digits of an employee's social security number), and whose value was of type `Employee` (assuming there exists some class `Employee`):

```
Dictionary<int, Employee> employeeData = new Dictionary<int, Employee>();
```

Once you have created an instance of the **Dictionary** object, you can add and remove items from it just like with the **Hashtable** class.

```
// Add some employees
employeeData.Add(455110189) = new Employee("Scott Mitchell");
employeeData.Add(455110191) = new Employee("Jisun Lee");
...
// See if employee with SSN 123-45-6789 works here
if (employeeData.ContainsKey(123456789))
    ...
```

Collision Resolution in the Dictionary Class

The Dictionary class differs from the Hashtable class in more ways than one. In addition to being strongly-typed, the **Dictionary** also employs a different collision resolution strategy than the Hashtable class, using a technique referred to as *chaining*. Recall that with probing, in the event of a collision another slot in the list of buckets is tried. (With rehashing, the hash is recomputed, and that new slot is tried.) With chaining, however, a secondary data structure is utilized to hold any collisions. Specifically, each slot in the **Dictionary** has an array of elements that map to that bucket. In the event of a collision, the colliding element is prepended to the bucket's list.

To better understand how chaining works, it helps to visualize the **Dictionary** as a hashtable whose buckets each contain a linked list of items that hash to that particular bucket. Figure 11 illustrates how series of items that hash to the same bucket will form a chain on that bucket.

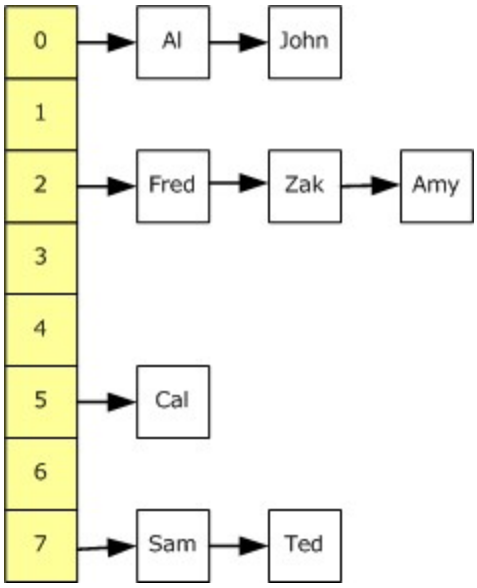

Figure 11. Chain created by a series of items hashed to the same bucket

The **Dictionary** in Figure 11 has eight buckets, drawn in yellow and running from the top down. A number of `Employee` objects have been added to the Dictionary. When an `Employee` object is added to the Dictionary, it is added to the bucket that its key hashes to and, if there's already an `Employee` instance there, it's prepended to the list of `Employee`s. That is, employees Al and John hash to the same bucket, as do Fred, Zak, and Amy, and Sam and Ted. Rather than reprobing in the event of a collision, as is done with the Hashtable class, the Dictionary simply chains any collisions onto the bucket's list.

> **Note** In our discussions on the **Hashtable** class, I mentioned that the average asymptotic running time for adding, removing, and searching the hashtable using probing is constant time, O(1). Adding an item to a hashtable that uses chaining takes constant time as well because it involves only computing the item's hash and prepending it to the appropriate bucket's list. Searching and removing items from a chained hashtable, however, take, on average, time proportional to the total number of items in the hashtable and the number of buckets. Specifically, the running time is O(n/m), where n is the total number of elements in the hashtable and m is the number of buckets. The Dictionary class is implemented such that n = m at all times. That is, the sum of all chained elements can never exceed the number of buckets. Because n never exceeds m, searching and removing run in constant time as well.

Conclusion

In this article we examined four data structures with inherent class support in the .NET Framework Base Class Library:

- The Queue
- The Stack
- The Hashtable
- The Dictionary

The Queue and Stack provide List -like capabilities in that they can store an arbitrary number of elements. The Queue and Stack differ from the List in the sense that while the List allows direct, random access to its elements, both the Queue and Stack limit how elements can be accessed.

The Queue uses a FIFO strategy, or first in, first out. That is, the order with which items can be removed from the Queue is precisely the order with which they were added to the Queue. To provide these semantics, the Queue offers two methods: `Enqueue()` and `Dequeue()`. Queues are useful data structures for job processing or other tasks where the order with which the items are processed is based by the order in which they were received.

The Stack, on the other hand, offers LIFO access, which stands for last in, first out. Stacks provide this access scheme through its `Push()` and `Pop()` methods. Stacks are used in a number of areas in computer science, from code execution to parsing.

The final two data structure examined were the Hashtable and Dictionary. The Hashtable extends the ArrayList by allowing items to be indexed by an arbitrary key, as opposed to indexed by an ordinal value. If you plan on searching the array by a specific unique key, it is much more efficient to use a Hashtable instead, as the lookups by key value occur in constant time as opposed to linear time. The Dictionary class provides a type-safe Hashtable, with an alternate collision resolution strategy.

This completes the second installment of this article series. In the third part we'll look at binary search trees, a data structure that provides *O*(log *n*) search time. Like Hashtables, binary search trees are an ideal choice over arrays if you know you will be searching the data frequently.

Until next time, Happy Programming!

References

- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.
- Headington, Mark R. and David D. Riley. "Data Abstraction and Structures Using C++." D.C. Heath and Company. 1994.
- Richter, Jeffrey. "Applied Microsoft .NET Framework Programming." Microsoft Press. 2002.
- Shared Source Common Language Infrastructure 1.0 Release. Microsoft - http://www.microsoft.com/downloads/details.aspx?FamilyId=3A1C93FA-7462-47D0-8E56-8DD34C6292F0&displaylang=en. Made available: November, 2002.

Scott Mitchell, author of six books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies since January 1998. Scott works as an independent consultant, trainer, and writer, and holds a Masters degree in Computer Science from the University of California – San Diego. He can be reached at mitchell@4guysfromrolla.com, or via his blog at http://ScottOnWriting.NET.