http://msdn.microsoft.com/en-us/library/ms379570

Visual Studio 2005 Technical Articles

An Extensive Examination of Data Structures Using C# 2.0

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: This article **kicks off** a six-part article series that focuses on important data structures and their use in application development. We'll examine both built-in data structures present in the .NET Framework, as well as essential data structures we'll build ourselves. This first part focuses on an **introduction to data structures**, defining what data structures are, how the efficiency of data structures are analyzed, and why this analysis is important. In this article, we'll also examine two of the most commonly used data structures present in the .NET Framework: the **Array and List**. (14 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

Welcome to the first in a six-part series on using data structures in .NET 2.0. This article series originally appeared on MSDN Online in October 2003, focusing on the .NET Framework version 1.x, and can be accessed at http://msdn.microsoft.com/vcsharp/default.aspx?pull=/library/en-us/dv_vstechart/html/datastructures_guide.asp. Version 2.0 of the .NET Framework adds new data structures to the Base Class Library, along with new features, such as Generics, that make creating type-safe data structures much easier than with version 1.x. This revised article series introduces these new .NET Framework data structures and examines using these new language features.

Throughout this article series, we will examine a variety of data structures, some of which are included in the .NET Framework's Base Class Library and others that we'll build ourselves. If you're unfamiliar with the term, *data structures* are classes that are used to organize data and provide various operations upon their data. Probably the most common and well-known data structure is the array, which contains a contiguous collection of data items that can be accessed by an ordinal index.

Before jumping into the content for this article, let's first take a quick peek at the roadmap for this six-part article series, so that you can see what lies ahead.

In this first part of the six-part series, we'll look at why data structures are important, and their effect on the performance of an algorithm. To determine a data structure's effect on performance, we'll need to examine how the various operations performed by a data structure can be rigorously analyzed. Finally, we'll turn our attention to two similar data structures present in the .NET Framework: the Array and the List. Chances are you've used these data structures in past projects. In this article, we'll examine what operations they provide and the efficiency of these operations.

In the Part 2, we'll explore the List's "cousins," the Queue and Stack. Like the List, both the Queue and Stack store a collection of data and are data structures available in the .NET Framework Base Class Library. Unlike a List, from which you can retrieve its elements in any order, Queues and Stacks only allow data to be accessed in a predetermined order. We'll examine some applications of Queues and Stacks, and see how these classes are implemented in the .NET Framework. After examining Queues and Stacks, we'll look at hashtables, which allow for direct access like an ArrayList, but store data indexed by a string key.

While arrays and Lists are ideal for directly accessing and storing contents, when working with large amounts of data, these data structures are often sub-optimal candidates when the data needs to be searched. In Part 3, we'll examine the binary search tree data structure, which is designed to improve the time needed to search a collection of items. Despite the improvement in search time with the binary tree, there are some shortcomings. In Part 4, we'll look at SkipLists, which are a mix between binary trees and linked lists, and address some of the issues inherent in binary trees.

In Part 5, we'll turn our attention to data structures that can be used to represent graphs. A graph is a collection of nodes, with a set of edges connecting the various nodes. For example, a map can be visualized as a graph, with cities as nodes and the highways between them as edged between the nodes. Many real-world problems can be abstractly defined in terms of graphs, thereby making graphs an often-used data structure.

Finally, in Part 6 we'll look at data structures to represent sets and disjoint sets. A set is an unordered collection of items. Disjoint sets are a collection of sets that have no elements in common with one another. Both sets and disjoint sets have many uses in everyday programs, which we'll examine in detail in this final part.

Analyzing the Performance of Data Structures

When thinking about a particular application or programming problem, many developers (myself included) find themselves most interested about writing the algorithm to tackle the problem at hand or adding cool features to the application to enhance the user's experience. Rarely, if ever, will you hear someone excited about what type of data structure they are using. However, the data structures used for a particular algorithm can greatly impact its performance. A very common example is finding an element in a data structure. With an unsorted array, this process takes time proportional to the number of elements in the array. With binary search trees or SkipLists, the time required is logarithmically proportional to the number of elements. When searching sufficiently large amounts of data, the data structure chosen can make a difference in the application's performance that can be visibly measured in seconds or even minutes.

Since the data structure used by an algorithm can greatly affect the algorithm's performance, it is important that there exists a rigorous method by which to compare the efficiency of various data structures. What we, as developers utilizing a data structure, are primarily interested in is how the data structures performance changes as the amount of data stored increases. That is, for each new element stored by the data structure, how are the running times of the data structure's operations effected?

Consider the following scenario: imagine that you are tasked with writing a program that will receive as input an array of strings that contain filenames. Your program's job is to determine whether that array of strings contains any filenames with a specific file extension. One approach to do this would be to scan through the array and set some flag once an XML file was encountered. The code might look like so:

```
public bool DoesExtensionExist(string [] fileNames, string extension)
{
      int i = 0;
      for (i = 0; i < fileNames.Length; i++)
         if (String.Compare(Path.GetExtension(fileNames[i]), extension, true) == 0)
            return true;

      return false;   // If we reach here, we didn't find the extension
   }
}
```

Here we see that, in the worst-case—when there is no file with a specified extension, or when there is such a file but it is the last file in the list—we have to search through each element of the array exactly once. To analyze the array's efficiency at sorting, we must ask ourselves the following: "Assume that I have an array with *n* elements. If I add another element, so the array has *n* + 1 elements, what is the new running time?" (The term "running time," despite its name, does not measure the absolute time it takes the program to run, but rather refers to the number of steps the program must perform to complete the given task at hand. When working with arrays, typically the steps considered are how many array accesses one needs to perform.) Since to search for a value in an array we need to visit, potentially, every array value, if we have *n* + 1 array elements, we might have to perform *n* + 1 checks. That is, the time it takes to search an array is linearly proportional to the number of elements in the array.

This sort of analysis described here is called *asymptotic analysis*, as it examines how the efficiency of a data structure changes as the data structure's size approaches infinity. The notation commonly used in asymptotic analysis is called *big-Oh notation*. The big-Oh notation to describe the performance of searching an unsorted array would be denoted as $O(n)$. The large script *O* is where the terminology big-Oh notation comes from, and the *n* indicates that the number of steps required to search an array grows linearly as the size of the array grows.

A more methodical way of computing the asymptotic running time of a block of code is to follow these simple steps:

1. Determine the steps that constitute the algorithm's running time. As aforementioned, with arrays, typically the steps considered are the read and write accesses to the array. For other data structures, the steps might differ. Typically, you want to concern yourself with steps that involve the data structure itself, and not simple, atomic operations performed by the computer. That is, with the block of code above, I analyzed its running time by only bothering to count how many times the array needs to be accessed, and did not bother worrying about the time for creating and initializing variables or the check to see if the two strings were equal.
2. Find the line(s) of code that perform the steps you are interested in counting. Put a 1 next to each of those lines.
3. For each line with a 1 next to it, see if it is in a loop. If so, change the 1 to 1 times the maximum number of repetitions the loop may perform. If you have two or more nested loops, continue the multiplication for each loop.
4. Find the largest single term you have written down. This is the running time.

Let's apply these steps to the block of code above. We've already identified that the steps we're interested in are the number of array accesses. Moving onto step 2 note that there is only one line on which the array, `fileNames`, is being accessed: as a parameter in the `String.Compare()` method, so mark a 1 next to that line. Now, applying step 3 notice that the access to `fileNames` in the `String.Compare()` method occurs within a loop that runs at most *n* times (where *n* is the size of the array). So, scratch out the 1 in the loop and replace it with *n*. , This is the largest value of *n*, so the running time is denoted as $O(n)$.

$O(n)$, or linear-time, represents just one of a myriad of possible asymptotic running times. Others include $O(\log_2 n)$, $O(n \log_2 n)$, $O(n^2)$, $O(2^n)$, and so on. Without getting into the gory mathematical details of big-Oh, the lower the term inside the parenthesis for large values of *n*, the better the data structure's operation's performance. For example, an operation that runs in $O(\log n)$ is more efficient than one that runs in $O(n)$ since $\log n < n$.

Note In case you need a quick mathematics refresher, $\log_a b = y$ is just another way to write $a^y = b$. So, $\log_2 4 = 2$, since $2^2 = 4$. Similarly, $\log_2 8 = 3$, since $2^3 = 8$. Clearly, $\log_2 n$ grows much slower than n alone, because when $n = 8$, $\log_2 n = 3$. In Part 3 we'll examine binary search trees whose search operation provides an $O(\log_2 n)$ running time.

Throughout this article series, each time we examine a new data structure and its operations, we'll be certain to compute its asymptotic running time and compare it to the running time for similar operations on other data structures.

Asymptotic Running Time and Real-World Algorithms

The asymptotic running time of an algorithm measures how the performance of the algorithm fares as the number of steps that the algorithm must perform approaches infinity. When the running time for one algorithm is said to be greater than another's, what this means mathematically is that there exists some number of steps such that once this number of steps is exceeded the algorithm with the greater running time will always take longer to execute than the one with the shorter running time. However, for instances with fewer steps, the algorithm with the asymptotically-greater running time may run faster than the one with the shorter running time.

For example, there are a myriad of algorithms for sorting an array that have differing running times. One of the simplest and most naïve sorting algorithms is bubble sort, which uses a pair of nested `for` loops to sort the elements of an array. Bubble sort exhibits a running time of $O(n^2)$ due to the two `for` loops. An alternative sorting algorithm is merge sort, which divides the array into halves and recursively sorts each half. The running time for merge sort is $O(n \log_2 n)$. Asymptotically, merge sort is much more efficient than bubble sort, but for small arrays, bubble sort may be more efficient. Merge sort must not only incur the expense of recursive function calls, but also of recombining the sorted array halves, whereas bubble sort simply loops through the array quadratically, swapping pairs of array values as needed. Overall, merge sort must perform fewer steps, but the steps merge sort has to perform are more expensive than the steps involved in bubble sort. For large arrays, this extra expense per step is negligible, but for smaller arrays, bubble sort may actually be more efficient.

Asymptotic analysis definitely has its place, as the asymptotic running time of two algorithms can show how one algorithm will outperform another when the algorithms are operating on sufficiently sized data. Using only asymptotic analysis to judge the performance of an algorithm, though, is foolhardy, as the actual execution times of different algorithms depends upon specific implementation factors, such as the amount of data being plugged into the algorithm. When deciding what data structure to employ in a real-world project, consider the asymptotic running time, but also carefully profile your application to ascertain the actual impact on performance your data structure choice bears.

Everyone's Favorite Linear, Direct Access, Homogeneous Data Structure: The Array

Arrays are one of the simplest and most widely used data structures in computer programs. Arrays in any programming language all share a few common properties:

- The contents of an array are stored in contiguous memory.
- All of the elements of an array must be of the same type or of a derived type; hence arrays are referred to as homogeneous data structures.
- Array elements can be directly accessed. With arrays if you know you want to access the i^{th} element, you can simply use one line of code: *arrayName*[*i*].

The common operations performed on arrays are:

- Allocation
- Accessing

In C#, when an array (or any reference type variable) is initially declared, it has a `null` value. That is, the following line of code simply creates a variable named `booleanArray` that equals `null`:

```
bool [] booleanArray;
```

Before we can begin to work with the array, we must create an array instance that can store a specific number of elements. This is accomplished using the following syntax:

```
booleanArray = new bool[10];
```

Or more generically:

```
arrayName = new arrayType[allocationSize];
```

This allocates a contiguous block of memory in the CLR-managed heap large enough to hold the *allocationSize* number of *arrayTypes*. If *arrayType* is a value type, then *allocationSize* number of unboxed *arrayType* values are created. If *arrayType* is a reference type, then *allocationSize* number of *arrayType* references are created. (If you are unfamiliar with the difference between reference and value types and the managed heap versus the stack, check out Understanding .NET's Common Type System.)

To help hammer home how the .NET Framework stores the internals of an array, consider the following example:

```
bool [] booleanArray;
FileInfo [] files;

booleanArray = new bool[10];
files = new FileInfo[10];
```

Here, the `booleanArray` is an array of the value type `System.Boolean`, while the `files` array is an array of a reference type, `System.IO.FileInfo`. Figure 1 shows a depiction of the CLR-managed heap after these four lines of code have executed.

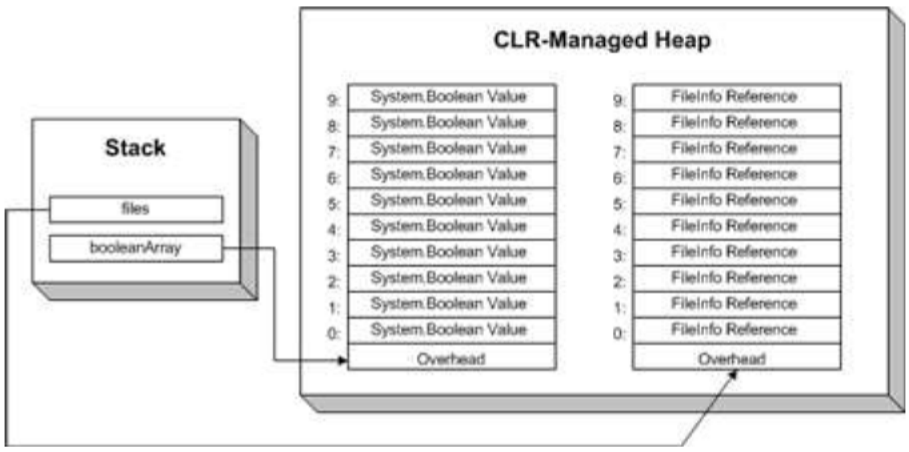


Figure 1. The contents of an array are laid out contiguously in the managed heap.

The thing to keep in mind is that the ten elements in the files array are *references* to `FileInfo` instances. Figure 2 hammers home this point, showing the memory layout if we assign some of the values in the `files` array to `FileInfo` instances.

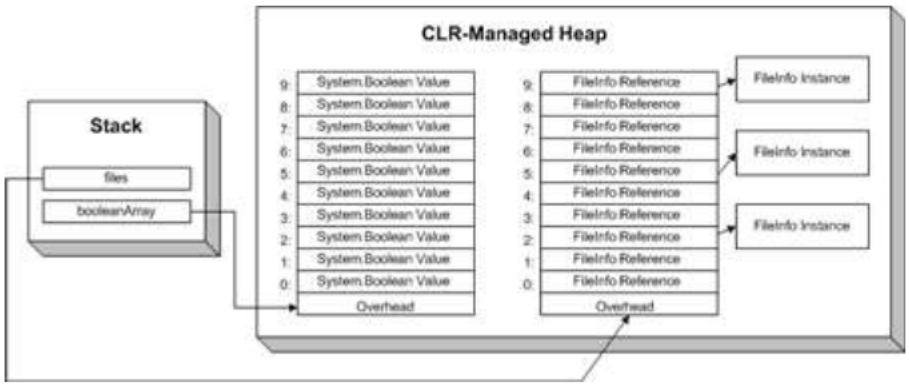


Figure 2. The contents of an array are laid out contiguously in the managed heap.

All arrays in .NET allow their elements to both be read and written to. The syntax for accessing an array element is:

```
// Read an array element
bool b = booleanArray[7];

// Write to an array element
booleanArray[0] = false;
```

The running time of an array access is denoted *O*(1) because it is constant. That is, regardless of how many elements are stored in the array, it takes the same amount of time to lookup an element. This constant running time is possible solely because an array's elements are stored contiguously, hence a lookup only requires knowledge of the array's starting location in memory, the size of each array element, and the element to be indexed.

Realize that in managed code, array lookups are a slight bit more involved than this because with each array access the CLR checks to ensure that the index being requested is within the array's bounds. If the array index specified is out of bounds, an `IndexOutOfRangeException` is thrown. This check help ensures that when stepping through an array we do not accidentally step past the last array index and into some other memory. This check, though, does not affect the asymptotic running time of an array access because the time to perform such checks does not increase as the size of the array increases.

Note This index-bounds check comes at a slight cost of performance for applications that make a large number of array accesses. With a bit of unmanaged code, though, this index out of bounds check can be bypassed. For more information, refer to Chapter 14 of *Applied Microsoft .NET Framework Programming* by Jeffrey Richter.

When working with an array, you might need to change the number of elements it holds. To do so, you'll need to create a new array instance of the specified size and copy the contents of the old array into the new, resized array. This process can be accomplished with the following code:

```
// Create an integer array with three elements
int [] fib = new int[3];
fib[0] = 1;
fib[1] = 1;
fib[2] = 2;

// Redimension message to a 10 element array
int [] temp = new int[10];

// Copy the fib array to temp
fib.CopyTo(temp, 0);

// Assign temp to fib
fib = temp;
```

After the last line of code, `fib` references a ten-element `Int32` array. The elements 3 through 9 in the `fib` array will have the default `Int32` value—0.

Arrays are excellent data structures to use when storing a collection of homogeneous types that you only need to access directly. Searching an unsorted array has linear running time. While this is acceptable when working with small arrays, or when performing very few searches, if your application is storing large arrays that are searched frequently, there are a number of other data structures better suited for the job. We'll look at some such data structures in upcoming pieces of this article series. Realize that if you are searching an array on some property and the array is *sorted* by that property, you can use an algorithm called binary search to search the array in $O(\log n)$ running time, which is on par with the search times for binary search trees. In fact, the `Array` class contains a static, `BinarySearch()` method. For more information on this method, check out an earlier article on mine, Efficiently Searching a Sorted Array.

Note The .NET Framework allows for multi-dimensional arrays as well. Multi-dimensional arrays, like single-dimensional arrays, offer a constant running time for accessing elements. Recall that the running time to search through a n-element single dimensional array was denoted $O(n)$. For an nxn two-dimensional array, the running time is denoted $O(n^2)$ because the search must check n^2 elements. More generally, a k-dimensional array has a search running time of $O(n^k)$. Keep in mind here than n is the number of elements in each dimension, not the total number of elements in the multi-dimensional array.

Creating Type-Safe, Performant, Reusable Data Structures

When creating a data structure for a particular problem, oftentimes the data structure's internals can be customized to the specifics of the problem. For example, imagine that you were working on a payroll application. One of the entities of this system would be an employee, so you might create an `Employee` class with applicable properties and methods. To represent a set of employees, you could use an array of type `Employee`, but perhaps you need some extra functionality not present in the array, or you simply don't want to have to concern yourself with writing code to watch the capacity of the array and resize it when necessary. One option would be to create a custom data structure that uses an internal array of `Employee` instances, and offered methods to extend the base functionality of an array, such as automatic resizing, searching of the array for a particular `Employee` object, and so on.

This data structure would likely prove very helpful in your application, so much so that you might want to reuse it in other applications. However, this data structure is not open to reuse because it is tightly-coupled to the payroll application, only being able to store elements of type `Employee` (or types derived from `Employee`). One option to make a more flexible data structure is to have the data structure maintain an internal array of `object` instances, as opposed to `Employee` instances. Because all types in the .NET Framework are derived from the `object` type, the data structure could store any type. This would make your collection data structure usable in other applications and scenarios.

Not surprisingly, the .NET Framework already contains a data structure that provides this functionality—the `System.Collections.ArrayList` class. The ArrayList maintains an internal `object` array and provides automatic resizing of the array as the number of elements added to the ArrayList grows. Because the ArrayList uses an object array, developers can add any type—strings, integers, `FileInfo` objects, `Form` instances, anything.

While the ArrayList provides added flexibility over the standard array, this flexibility comes at the cost of performance. Because the ArrayList stores an array of `objects`, when reading the value from an ArrayList you need to explicitly cast it to the data type being stored in the specified location. Recall that an array of a value type—such as a `System.Int32`, `System.Double`, `System.Boolean`, and so on—is stored contiguously in the managed heap in its unboxed form. The ArrayList's internal array, however, is an array of `object` references. Therefore, even if you have an ArrayList that stores nothing but value types, each ArrayList element is a reference to a boxed value type, as shown in Figure 3.

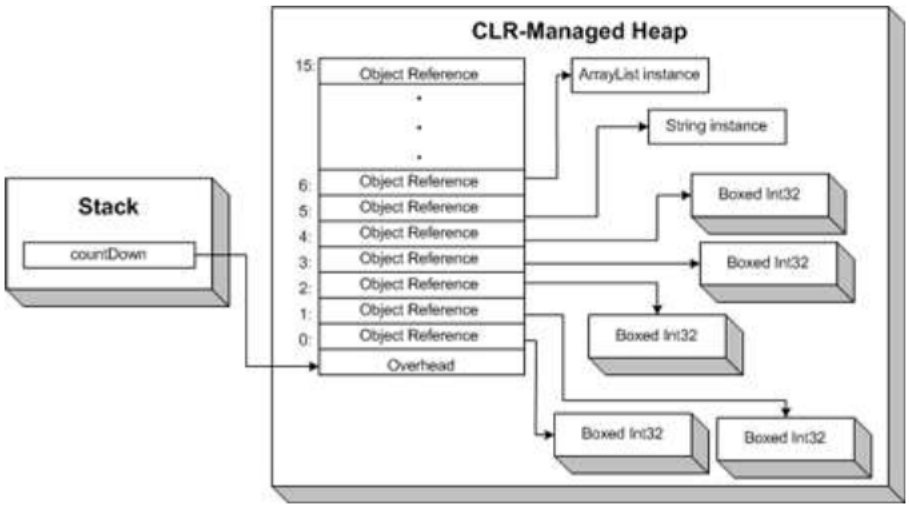


Figure 3. The ArrayList contains a contiguous block of object references

The boxing and unboxing, along with the extra level of indirection, that comes with using value types in an ArrayList can hamper the performance of your application when using large ArrayLists with many reads and writes. As Figure 3 illustrates, the same memory layout occurs for reference types in both ArrayLists and arrays.

Having an `object` array also introduces potential bugs that won't be noticed until run-time. A developer may intend to only add elements of a particular type to an ArrayList, but since the ArrayList allows any type to be added, adding an incorrect type won't be caught during compilation. Instead, such a mistake would not be apparent until run-time, meaning the bug would not be found until testing or, in the worse case, during actual use.

Generics to the Rescue

Fortunately, the typing and performance issues associated with the ArrayList have been remedied in the .NET Framework 2.0, thanks to *Generics*. Generics allow for a developer creating a data structure to defer type selection. The types associated with a data structure can, instead, be chosen by the developer utilizing the data structure. To better understand Generics, let's look at an example of creating a type-safe collection. Specifically, we'll create a class that maintains an internal array of a to-be specified type, with methods to read and add items from the internal array.

```
public class TypeSafeList<T>
{
    T[] innerArray = new T[0];
    int currentSize = 0;
    int capacity = 0;

    public void Add(T item)
    {
        // see if array needs to be resized
        if (currentSize == capacity)
        {
            // resize array
            capacity = capacity == 0 ? 4 : capacity * 2;  // double capacity
            T[] copy = new T[capacity];    // create newly sized array
            Array.Copy(innerArray, copy, currentSize);  // copy over the array
            innerArray = copy;    // assign innerArray to the new, larger array
        }

        innerArray[currentSize] = item;
        currentSize++;
    }
```

```
    public T this[int index]
    {
        get
        {
            if (index < 0 || index >= currentSize)
                throw new IndexOutOfRangeException();
            return innerArray[index];
        }
        set
        {
            if (index < 0 || index >= currentSize)
                throw new IndexOutOfRangeException();
            innerArray[index] = value;
        }
    }

    public override string ToString()
    {
        string output = string.Empty;
        for (int i = 0; i < currentSize - 1; i++)
            output += innerArray[i] + ", ";

        return output + innerArray[currentSize - 1];
    }
}
```

Notice that in the first line of code, in the class definition, a type identifier, `T`, is defined. What this syntax indicates is that the class will require the developer using it to specify a single type. This developer-specified type is aliased as `T`, although any other valid variable name could have been used. The type identifier is used within the class's properties and methods. For example, the inner array is of type `T`, and the `Add()` method accepts an input parameter of type `T`, which is then added to the array.

To declare a variable of this class, a developer would need to specify the type `T`, like so:

```
TypeSafeList<type> variableName;
```

The following code snippet demonstrates creating an instance of TypeSafeList that stores integers, and populating the list with the first 25 Fibonacci numbers.

```
TypeSafeList<int> fib = new TypeSafeList<int>();
fib.Add(1);
fib.Add(1);

for (int i = 2; i < 25; i++)
    fib.Add(fib[i - 2] + fib[i - 1]);

Console.WriteLine(fib.ToString());
```

The main advantages of Generics include:

- **Type-safety:** a developer using the TypeSafeList class can only add elements that are of the type or are derived from the type specified. For example, trying to add a string to the `fib` TypeSafeList in the example above would result in a compile-time error.
- **Performance:** Generics remove the need to type check at run-time, and eliminate the cost associated with boxing and unboxing.
- **Reusability:** Generics break the tight-coupling between a data structure and the application for which it was created. This provides a higher degree of reuse for data structures.

Many of the data structures we'll be examining throughout this series are data structures that utilize Generics, and when creating data structures—such as the binary tree data structure we'll build in Part 3—we'll be utilizing Generics ourselves.

The List: a Homogeneous, Self-Redimensioning Array

An array, as we saw, is designed to store a specific number of items of the same type in a contiguous fashion. Arrays, while simple to use, can quickly become a nuisance if you find yourself needing to regularly resize the array, or don't know how many elements you'll need when initializing the array. One option to avoid having to manually resize an array is to create a data structure that serves as a wrapper for an array, providing read/write access to the array and automatically resizing the array as needed. We started creating our own such data structure in the previous section—the TypeSafeList, but there's no need to implement this yourself as the .NET Framework provides such a class for you. This class, the `List` class, is found in the `System.Collections.Generics` namespace.

The `List` class contains an internal array and exposes methods and properties that, among other things, allow read and write access to the elements of the internal array. The `List` class, like an array, is a homogeneous data structure, meaning that you can only store items of the same type or from a derived type within a given `List`. The `List` utilizes Generics, a new feature in version 2.0 of the .NET Framework, in order to let the developer specify at development time the type of data a `List` will hold.

Therefore, when creating a `List` instance, you must specify the data type of the `List`'s contents using the Generics syntax:

```
// Create a List of integers
List<int> myFavoriteIntegers = new List<int>();

// Create a list of strings
List<string> friendsNames = new List<string>();
```

Note that the type of data the `List` can store is specified in the declaration and instantiation of the `List`. When creating a new `List`, you don't have to specify a `List` size, although you can specify a default starting size by passing in an integer into the constructor, or through the `List`'s `Capacity` property. To add an item to a `List`, simply use the `Add()` method. The `List`, like the array, can have its elements directly accessed via an ordinal index. The following code snippet shows creating a `List` of integers, populating the list with some initial values with the `Add()` method, and then reading and writing the `List`'s values through an ordinal index.

```
// Create a List of integers
List<int> powersOf2 = new List<int>();

// Add 6 integers to the List
powersOf2.Add(1);
powersOf2.Add(2);
powersOf2.Add(4);
powersOf2.Add(8);
powersOf2.Add(16);
powersOf2.Add(32);

// Change the 2nd List item to 10
powersOf2[1] = 10;

// Compute 2^3 + 2^4
int sum = powersOf2[2] + powersOf2[3];
```

The `List` takes the basic array and wraps it in a class that hides the implementation complexity. When creating a `List`, you don't need to explicitly specify an initial starting size. When adding items to the `List`, you don't need to concern yourself with resizing the data structure, as you do with an array. Furthermore, the `List` has a number of

other methods that take care of common array tasks. For example, to find an element in an array, you'd need to write a `for` loop to scan through the array (unless the array was sorted). With a `List`, you can simply use the `Contains()` method to determine if an element exists in an array, or `IndexOf()` to find the ordinal position of an element. The List class also contains a `BinarySearch()` method to efficiently search a sorted array, and methods like `Find()`, `FindAll()`, `Sort()`, and `ConvertAll()`, which can utilize delegates to perform operations that would require several lines of code using arrays.

The asymptotic running time of the `List`'s operations are the same as those of the standard array's. While the `List` does indeed have more overhead, the relationship between the number of elements in the `List` and the cost per operation is the same as the standard array.

Conclusion

This article, the first in a series of six, started our discussion on data structures by identifying why studying data structures was important, and by providing a means of how to analyze the performance of data structures. This material is important to understand, as being able to analyze the running times of various data structure operations is a major tool used when deciding what data structure to use for a particular programming problem.

After studying how to analyze data structures, we turned to examining two of the most common data structures in the .NET Framework Base Class Library: `System.Array` and `System.Collections.Generics.List`. Arrays allow for a contiguous block of homogeneous types and derived types. Their main benefit is that they provide lightning-fast access to reading and writing array elements. Their weak point lies in searching arrays, as each and every element must potentially be visited (in an unsorted array), and the fact that resizing the array requires writing a bit of code.

The `List` class wraps the functionality of an array with a number of helpful methods. For example, the `Add()` method adds an element to the List and automatically re-dimensions the array if needed. The `IndexOf()` method aids the developer by searching the `List`'s contents for a particular item. The functionality provided by a List is nothing that you couldn't implement using plain old arrays, but the List class saves you the trouble of having to write the code to perform these common tasks yourself.

In the next part of this article series we'll turn our attention first to two "cousins" of the **List**: the **Stack** and **Queue** classes. We'll also look at associative arrays, which are arrays indexed by a string key as opposed to an integer value. Associative arrays are provided in the .NET Framework Base Class Library using the **Hashtable** and **Dictionary** classes.

Happy Programming!

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.

http://msdn.microsoft.com/en-us/library/ms379571

Visual Studio 2005 Technical Articles

An Extensive Examination of Data Structures Using C# 2.0

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.

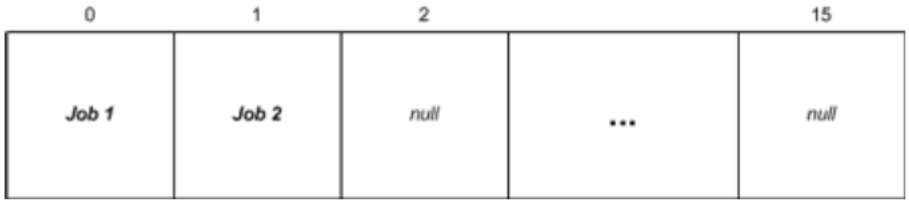


Figure 1. The ArrayList after the first two lines of code

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.

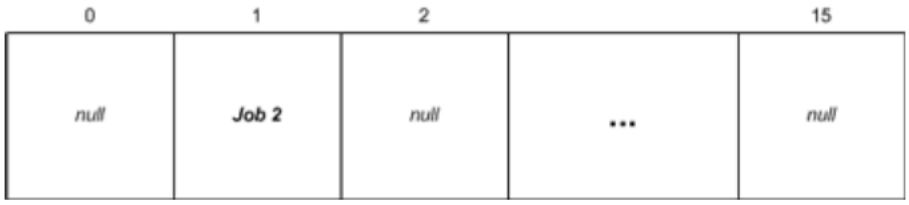


Figure 2. Program after the GetNextJob() method is invoked

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.

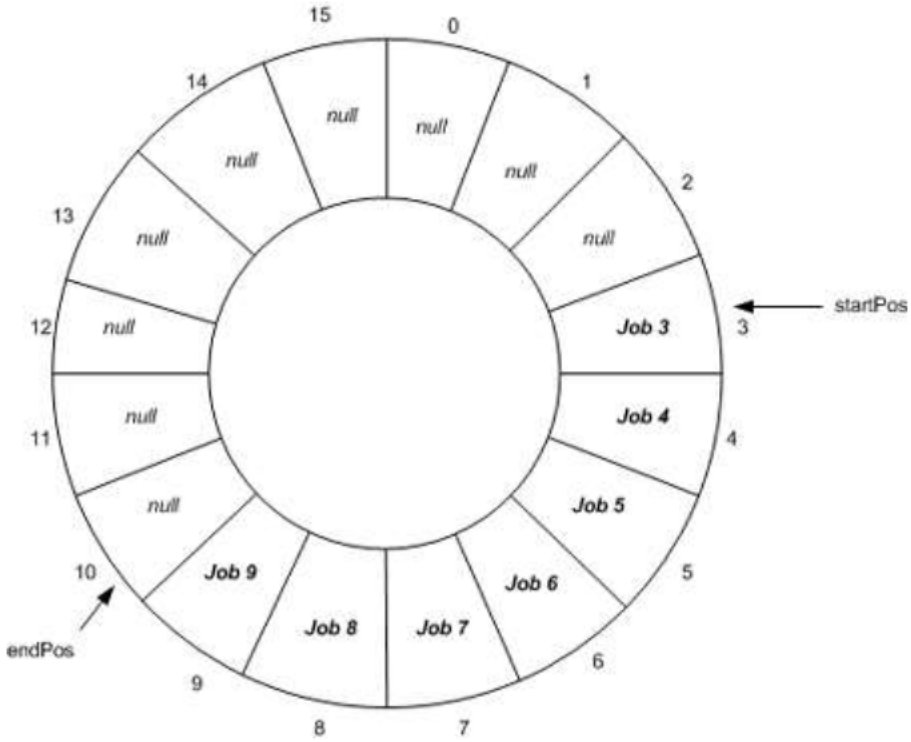


Figure 4. Example of a circular array

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.

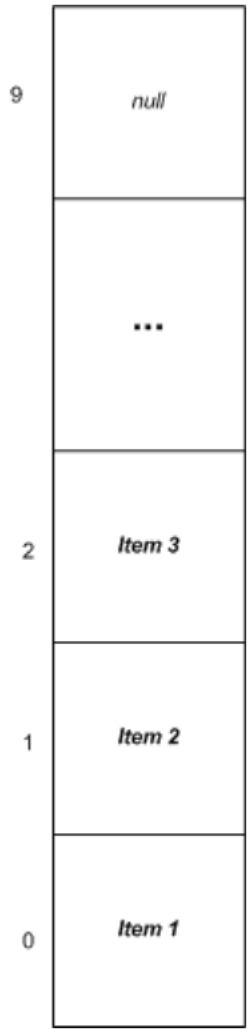


Figure 5. Graphical representation of a stack with three items

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.

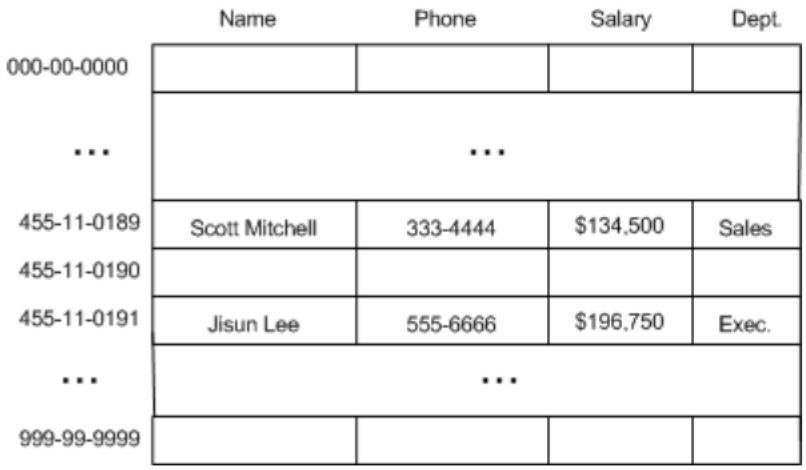


Figure 7. Array showing all possible elements for a 9-digit number

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.

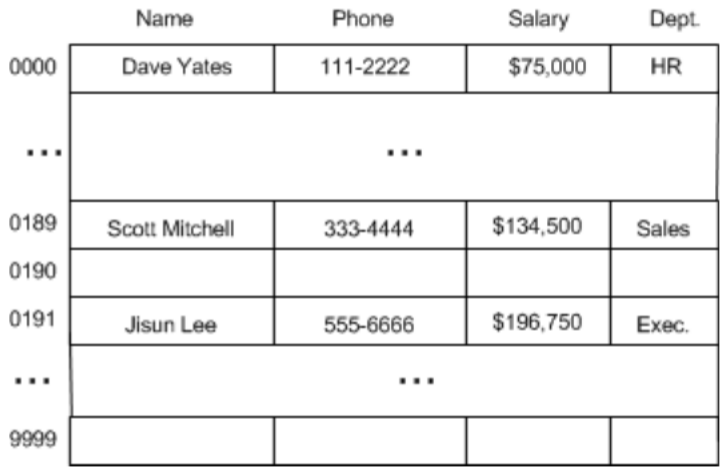


Figure 8. Trimmed down array

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.

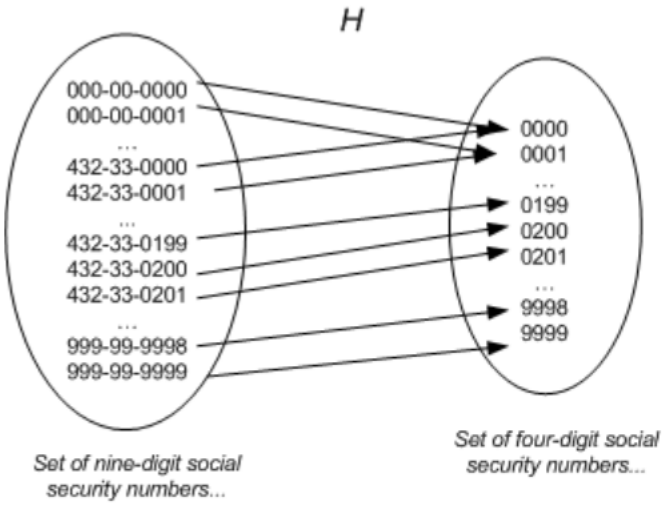


Figure 9. Graphical representation of a hash function

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 $f : A \rightarrow B$. 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:

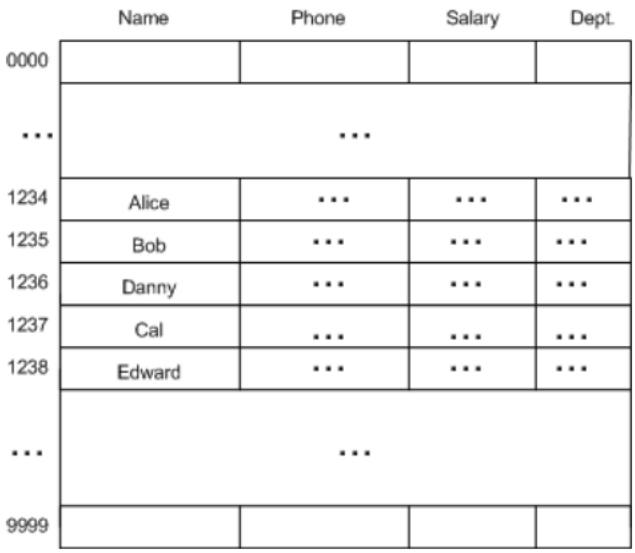


Figure 10. Hash table of four employees with similar numbers

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.

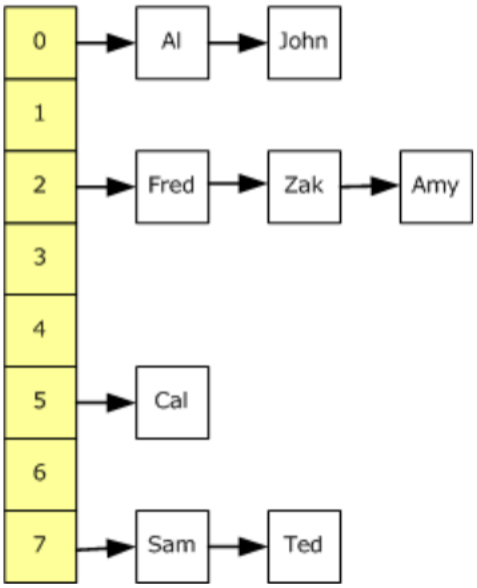


Figure 11. Chain created by a series of items hashed to the same bucket

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 `Employees`. 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.

http://msdn.microsoft.com/en-us/library/ms379572

Visual Studio 2005 Technical Articles

An Extensive Examination of Data Structures Using C# 2.0

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: This article, the **third** in a six-part series on data structures in the .NET Framework, looks at a common data structure that is *not* included in the .NET Framework Base Class Library—**binary trees**. Whereas arrays arrange data linearly, binary trees can be envisioned as storing data in two dimensions. A special kind of binary tree, called a **binary search tree**, or BST, allows for a much more optimized search time than with unsorted arrays. (30 printed pages)

Download the DataStructures20.msi sample file.

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, 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, and its relative, the List. In Part 2 we looked at the cousins of the List, the Stack and the Queue, which store their data like a List, but limit the means by which their contained data can be accessed. In Part 2 we also looked at the Hashtable and Dictionary classes, which are essentially arrays that are indexed by some arbitrary object as opposed to by an ordinal value.

The List, Stack, Queue, Hashtable, and Dictionary all use an underlying array as the means by which their data is stored. This means that, under the covers, these data structures are bound by the limitations imposed by an array. Recall from Part 1 that an array is stored linearly in memory, requires explicit resizing when the array's capacity is reached, and suffers from linear searching time.

In this third installment of the article series, we will examine a new data structure, the binary tree. As we'll see, binary trees store data in a non-linear fashion. After discussing the properties of binary trees, we'll look at a more

specific type of binary tree—the binary search tree, or BST. A BST imposes certain rules on how the items of the tree are arranged. These rules provide BSTs with a sub-linear search time.

Arranging Data in a Tree

If you've ever looked at a genealogy table, or at the chain of command in a corporation, you've seen data arranged in a *tree*. A tree is composed of a collection of *nodes*, where each node has some associated data and a set of *children*. A node's children are those nodes that appear immediately beneath the node itself. A node's *parent* is the node immediately above it. A tree's *root* is the single node that contains no parent.

Figure 1 shows an example of the chain of command in a fictional company.

Figure 1. Tree view of a chain of command in a fictitious company

In this example, the tree's root is Bob Smith, CEO. This node is the root because it has no parent. The Bob Smith node has one child, Tina Jones, President, whose parent is Bob Smith. The Tina Jones node has three children—Jisun Lee, CIO; Frank Mitchell, CFO; and Davis Johnson, VP of Sales. Each of these nodes' parent is the Tina Jones node. The Jisun Lee node has two children—Tony Yee and Sam Maher; the Frank Mitchell node has one child—Darren Kulton; and the Davis Johnson node has three children—Todd Brown, Jimmy Wong, and Sarah Yates.

All trees exhibit the following properties:

- There is precisely one root.
- All nodes except the root have precisely one parent.
- There are no *cycles*. That is, starting at any given node, there is not some path that can take you back to the starting node. The first two properties—that there exists one root and that all nodes save the root have one parent—guarantee that no cycles exist.

Trees are useful for arranging data in a hierarchy. As we will discuss later in this article, the time to search for an item can be drastically reduced by intelligently arranging the hierarchy. Before we can arrive at that topic, though, we need to first discuss a special kind of tree, the *binary tree*.

Note Throughout this article we will be looking at numerous new terms and definitions. These definitions, along with being introduced throughout the text, are listed in Appendix A.

Understanding Binary Trees

A binary tree is a special kind of tree, one in which all nodes have at most two children. For a given node in a binary tree, the first child is referred to as the *left* child, while the second child is referred to as the *right* child. Figure 2 depicts two binary trees.

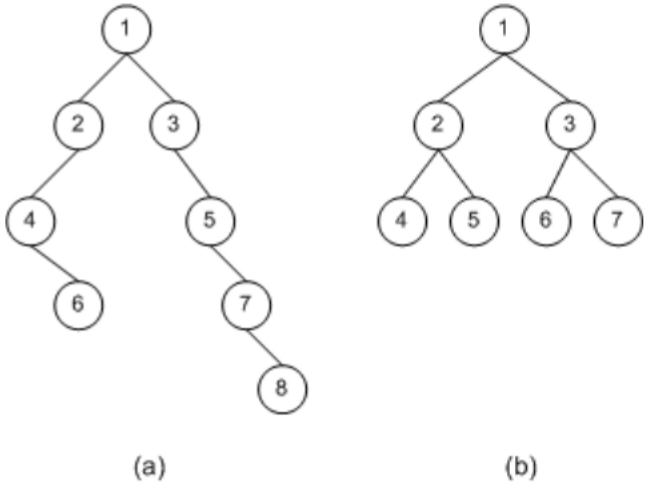


Figure 2. Illustration of two binary trees

Binary tree (a) has 8 nodes, with node 1 as its root. Node 1's left child is node 2; node 1's right child is node 3. Notice that a node doesn't need to have both a left child and right child. In binary tree (a), node 4, for example, has only a right child. Furthermore, a node can have no children. In binary tree (b), nodes 4, 5, 6, and 7 all have no children.

Nodes that have no children are referred to as *leaf nodes*. Nodes that have one or two children are referred to as *internal nodes*. Using these new definitions, the leaf nodes in binary tree (a) are nodes 6 and 8; the internal nodes are nodes 1, 2, 3, 4, 5, and 7.

Unfortunately, the .NET Framework does not contain a binary tree class, so in order to better understand binary trees, let's take a moment to create our own binary tree class.

The First Step: Creating a Base Node Class

The first step in designing our binary tree class is to create a class that represents the nodes of the binary tree. Rather than create a class specific to nodes in a binary tree, let's create a base `Node` class that can be extended to meet the needs of a binary tree node through inheritance. The base `Node` class represents a node in a general tree, one whose nodes can have an arbitrary number of children. To model this, we'll create not just a `Node` class, but a `NodeList` class as well. The Node class contains some data and a NodeList instance, which represents the node's children. The `Node` class affords a perfect time to utilize the power of Generics, which will allow us to let the developer using the class decide at develop-time what type of data to store in the node.

The following is the code for the `Node` class.

```
public class Node<T>
{
        // Private member-variables
        private T data;
        private NodeList<T> neighbors = null;

        public Node() {}
        public Node(T data) : this(data, null) {}
        public Node(T data, NodeList<T> neighbors)
        {
            this.data = data;
            this.neighbors = neighbors;
        }

        public T Value
        {
            get
            {
                return data;
            }
            set
            {
                data = value;
            }
        }

        protected NodeList<T> Neighbors
        {
            get
            {
                return neighbors;
            }
            set
            {
                neighbors = value;
            }
        }
    }
}
```

Note that the `Node` class has two private member variables:

- `data`, of type `T`. This member variable contains the data stored in the node of the type specified by the developer using this class.
- `neighbors`, of type `NodeList<T>`. This member variable represents the node's children.

The remainder of the class contains the constructors and the public properties, which provide access to the two member variables.

The `NodeList` class contains a strongly-typed collection of `Node<T>` instances. As the following code shows, the `NodeList` class is derived from the `Collection<T>` class in the `System.Collections.Generics` namespace. The `Collection<T>` class provides the base functionality for a strong-typed collection, with methods like `Add(T)`, `Remove(T)`, and `Clear()`, and properties like `Count` and a default indexer. In addition to the methods and properties inherited from `Collection<T>`, `NodeList` provides a constructor that creates a specified number of nodes in the collection, and a method that searches the collection for an element of a particular value.

```
public class NodeList<T> : Collection<Node<T>>
{
    public NodeList() : base() { }

    public NodeList(int initialSize)
    {
        // Add the specified number of items
        for (int i = 0; i < initialSize; i++)
            base.Items.Add(default(Node<T>));
    }

    public Node<T> FindByValue(T value)
    {
        // search the list for the value
        foreach (Node<T> node in Items)
            if (node.Value.Equals(value))
                return node;

        // if we reached here, we didn't find a matching node
        return null;
    }
}
```

Note The impetus behind creating a generic **Node** class is because later in this article, as well as in future parts, we'll be creating other classes that are made up of a set of nodes. Rather than have each class create its own specific node class, each class will borrow the functionality of the base `Node` class and extend the base class to meet its particular needs.

Extending the Base Node Class

While the `Node` class is adequate for any generic tree, a binary tree has tighter restrictions. As discussed earlier, a binary tree's nodes have at most two children, commonly referred to as left and right. To provide a binary tree-specific node class, we can extend the base Node class by creating a **BinaryTreeNode** class that exposes two properties—`Left` and `Right`—that operate on the base class's `Neighbors` property.

```
public class BinaryTreeNode<T> : Node<T>
{
    public BinaryTreeNode() : base() {}
    public BinaryTreeNode(T data) : base(data, null) {}
    public BinaryTreeNode(T data, BinaryTreeNode<T> left, BinaryTreeNode<T> right)
    {
        base.Value = data;
        NodeList<T> children = new NodeList<T>(2);
        children[0] = left;
        children[1] = right;

        base.Neighbors = children;
    }

    public BinaryTreeNode<T> Left
    {
        get
        {
            if (base.Neighbors == null)
                return null;
            else
                return (BinaryTreeNode<T>) base.Neighbors[0];
        }
```

```
        set
        {
            if (base.Neighbors == null)
                base.Neighbors = new NodeList<T>(2);

            base.Neighbors[0] = value;
        }
    }

    public BinaryTreeNode<T> Right
    {
        get
        {
            if (base.Neighbors == null)
                return null;
            else
                return (BinaryTreeNode<T>) base.Neighbors[1];
        }
        set
        {
            if (base.Neighbors == null)
                base.Neighbors = new NodeList<T>(2);

            base.Neighbors[1] = value;
        }
    }
}
```

The lion's share of the work of this extended class is in the `Right` and `Left` properties. In these properties we need to ensure that the base class's `NeighborsNodeList` has been created. If it hasn't, then in the get accessor we return `null`; in the set accessor we need to create a new `NodeList` with precisely two elements. As you can see in the code, the `Left` property refers to the first element in the `Neighbors` collection (`Neighbors[0]`), while `Right` refers to the second (`Neighbors[1]`).

Creating the BinaryTree Class

With the `BinaryTreeNode` class complete, the `BinaryTree` class is a cinch to develop. The `BinaryTree` class contains a single private member variable—`root`. `root` is of type `BinaryTreeNode` and represents the root of the binary tree. This private member variable is exposed as a public property. (The `BinaryTree` class uses Generics as well; the type specified for the `BinaryTree` class is the type used for the `BinaryTreeNode` root.)

The `BinaryTree` class has a single public method, `Clear()`, which clears out the contents of the tree. `Clear()` works by simply setting the `root` to `null`. Other than the root and a `Clear()` method, the **BinaryTree** class contains no other properties or methods. Crafting the contents of the binary tree is the responsibility of the developer using this data structure.

Below is the code for the `BinaryTree` class.

```
public class BinaryTree<T>
{
   private BinaryTreeNode<T> root;

   public BinaryTree()
   {
      root = null;
   }
```

```
   public virtual void Clear()
   {
      root = null;
   }

   public BinaryTreeNode<T> Root
   {
      get
      {
         return root;
      }
      set
      {
         root = value;
      }
   }
}
```

The following code illustrates how to use the `BinaryTree` class to generate a binary tree with the same data and structure as binary tree (a) shown in Figure 2.

```
BinaryTree<int> btree = new BinaryTree<int>();
btree.Root = new BinaryTreeNode<int>(1);
btree.Root.Left = new BinaryTreeNode<int>(2);
btree.Root.Right = new BinaryTreeNode<int>(3);

btree.Root.Left.Left = new BinaryTreeNode<int>(4);
btree.Root.Right.Right = new BinaryTreeNode<int>(5);

btree.Root.Left.Left.Right = new BinaryTreeNode<int>(6);
btree.Root.Right.Right.Right = new BinaryTreeNode<int>(7);

btree.Root.Right.Right.Right.Right = new BinaryTreeNode<int>(8);
```

Note that we start by creating a `BinaryTree` class instance, and then create its root. We then must manually add new `BinaryTreeNode` class instances to the appropriate left and right children. For example, to add node 4, which is the left child of the left child of the root, we use: `btree.Root.Left.Left = new BinaryTreeNode<int>(4);`

Recall from Part 1 of this article series that an array's elements are stored in a contiguous block of memory. By doing so, arrays exhibit constant-time lookups. That is, the time it takes to access a particular element of an array does not change as the number of elements in the array increases.

Binary trees, however, are not stored contiguously in memory, as Figure 3 illustrates. Rather, the `BinaryTree` class instance has a reference to the root `BinaryTreeNode` class instance. The root `BinaryTreeNode` class instance has references to its left and right child `BinaryTreeNode` instances; these child instances have references to their child instances, and so on. The point is, the various `BinaryTreeNode` instances that makeup a binary tree can be scattered throughout the CLR managed heap. They are not necessarily contiguous, as are the elements of an array.

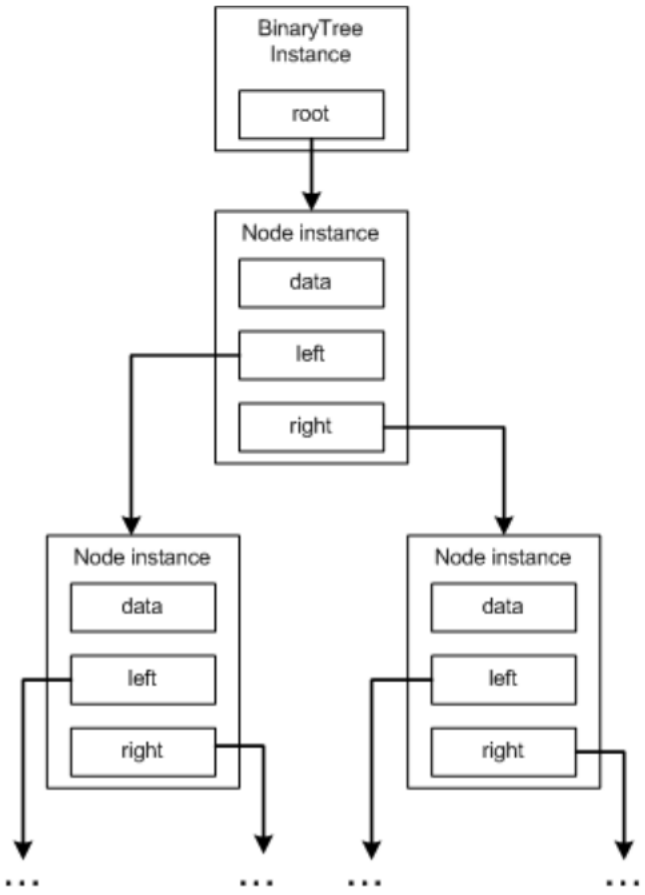


Figure 3. Binary trees stored in memory

Imagine that we wanted to access a particular node in a binary tree. To accomplish this we need to search the binary tree's set of nodes, looking for the particular node. There's no direct access to a given node as with an array. Searching a binary tree can take linear time, as potentially all nodes will need to be examined. That is, as the number of nodes in the binary tree increases, the number of steps to find an arbitrary node will increase as well.

So, if a binary tree's lookup time is linear, and its search time is linear, how is the binary tree any better than an array, whose search time is linear, but whose lookup time is constant? Well, a generic binary tree doesn't offer us any benefit over an array. However, by intelligently organizing the items in a binary tree, we can greatly improve the search time (and therefore the lookup time as well).

Improving the Search Time with Binary Search Trees (BSTs)

A *binary search tree* is a special kind of binary tree designed to improve the efficiency of searching through the contents of a binary tree. Binary search trees exhibit the following property: for any node *n*, every descendant node's value in the left *subtree* of *n* is less than the value of *n*, and every descendant node's value in the right *subtree* is greater than the value of *n*.

A subtree rooted at node *n* is the tree formed by imaging node *n* was a root. That is, the subtree's nodes are the descendants of *n* and the subtree's root is *n* itself. Figure 4 illustrates the concept of subtrees and the binary search tree property.

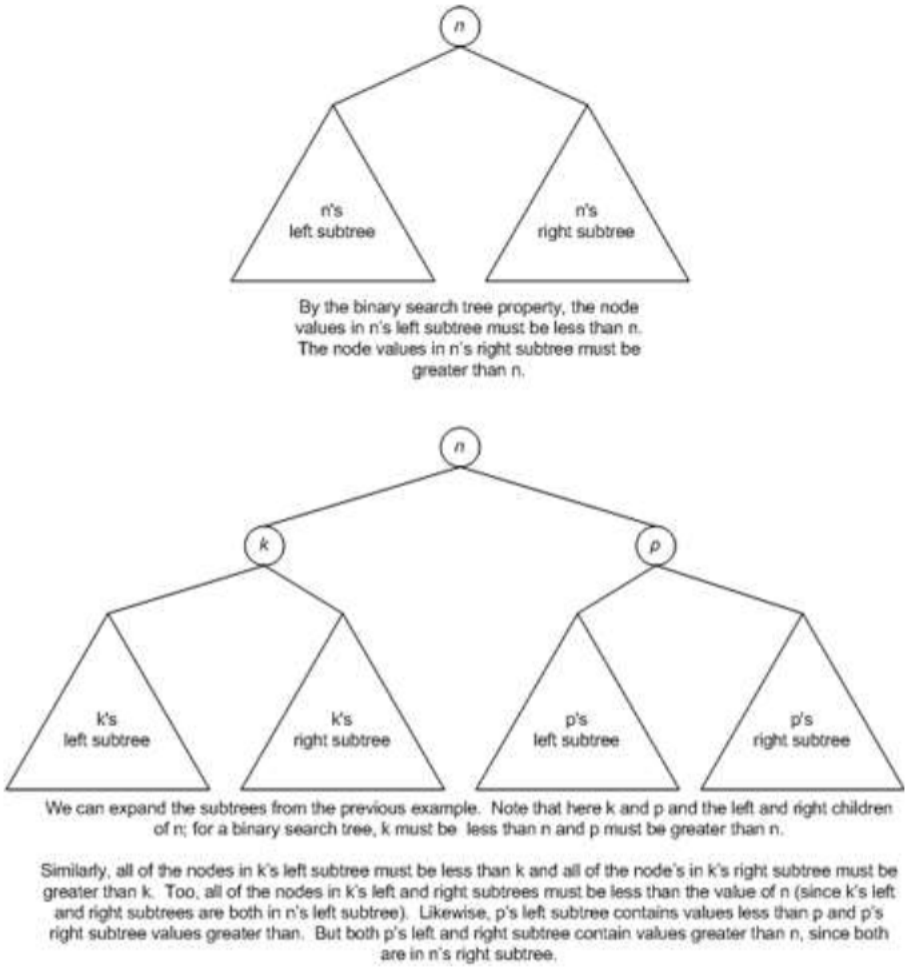


Figure 4. Subtrees and the binary search tree property

Figure 5 shows two examples of binary trees. The one on the right, binary tree (b), is a BST because it exhibits the binary search tree property. Binary tree (a), however, is not a BST because not all nodes of the tree exhibit the binary search tree property. Namely, node 10's right child, 8, is less than 10 but it appears in node 10's right subtree. Similarly, node 8's right child, node 4, is less than 8 but appears on node 8's right subtree. This property is violated in other locations, too. For example, node 9's right subtree contains values less than 9, namely 8 and 4.

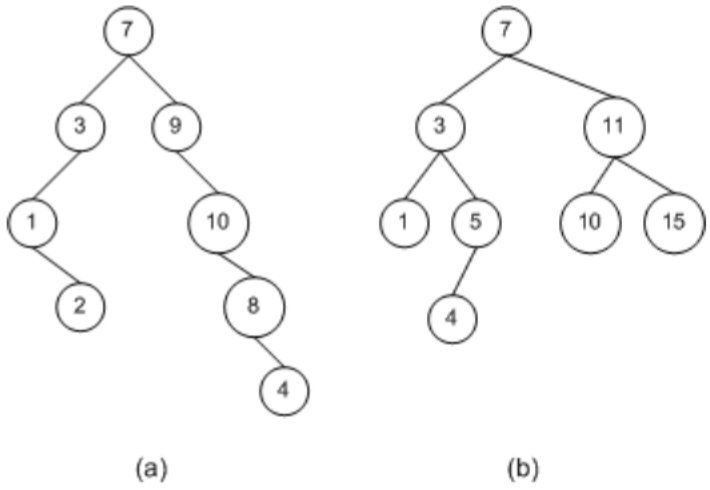


Figure 5. Comparison of non-BST binary tree (a) and a BST binary tree (b)

Note that for the binary search tree property to be upheld, the data stored in the nodes of a BST must be able to be compared to one another. Specifically, given two nodes, a BST must be able to determine if one is less than, greater than, or equal to the other.

Now, imagine that you want to search a BST for a particular node. For example, for the BST in Figure 5 (the binary tree (b)), imagine that we wanted to search for the node 10. A BST, like a regular binary tree, only has a direct reference to one node, its root. Can you think of an optimal way to search the tree to see if node 10 exists? There's a better way than searching through each node in the tree.

To see if 10 exists in the tree we can start with the root. We see that the root's value (7) is less than the value of the node we are looking for. Therefore, if 10 does exist in the BST, it must be in the root's right subtree. Therefore, we continue our search at node 11. Here we notice that 10 is less than 11, so if 10 exists in the BST it must exist in the left subtree of 11. Moving onto the left child of 11, we find node 10, and have located the node we are looking for.

What happens if we search for a node that does not exist in the tree? Imagine that we wanted to find node 9. We'd start by repeating the same steps above. Upon reaching node 10, we'd see that node 10 was greater than 9, so 9, if it exists, must be in 10's left subtree. However, we'd notice that 10 has no left child, therefore 9 must not exist in the tree.

More formally, our searching algorithm goes as follows. We have a node *n* we wish to find (or determine if it exists), and we have a reference to the BST's root. This algorithm performs a number of comparisons until a null reference is hit or until the node we are searching for is found. At each step we are dealing with two nodes: a node in the tree, call it *c*, that we are currently comparing with *n*, the node we are looking for. Initially, *c* is the root of the BST. We apply the following steps:

1. If *c* is a null reference, then exit the algorithm. *n* is not in the BST.
2. Compare *c*'s value and *n*'s value.
3. If the values are equal, then we found *n*.
4. If *n*'s value is less than *c*'s then *n*, if it exists, must be in the *c*'s left subtree. Therefore, return to step 1, letting *c* be *c*'s left child.
5. If *n*'s value is greater than *c*'s then *n*, if it exists, must be in the *c*'s right subtree. Therefore, return to step 1, letting *c* be *c*'s right child.

We applied these steps earlier when searching for node 10. We started with the root node and noted that 10 was greater than 7, so we repeated our comparison with the root's right child, 11. Here, we noted 10 was less than 11, so we repeated our comparison with 11's left child. At this point we had found node 10. When searching for node 9, which did not exist, we wound up performing our comparison with 10's left child, which was a null reference. Hence we deduced that 9 did not exist in the BST.

Analyzing the BST Search Algorithm

For finding a node in a BST, at each stage we ideally reduce the number of nodes we have to check by half. For example, consider the BST in Figure 6, which contains 15 nodes. When starting our search algorithm at the root, our first comparison will take us to either the root's left or right child. In either case, once this step is made the number of nodes that need to be considered has just halved, from 15 down to 7. Similarly, at the next step the number is halved again, from 7 down to 3, and so on.

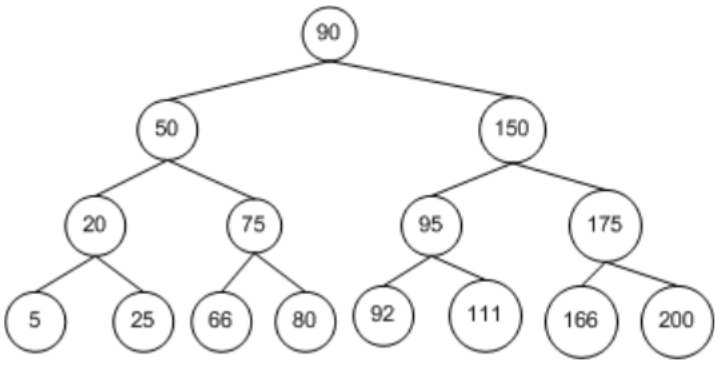


Figure 6. BST with 15 nodes

The important concept to understand here is that ideally at each step in the algorithm the number of nodes that have to be considered has been cut in half. Compare this to searching an array. When searching an array we have to search *all* elements, one element at a time. That is, when searching an array with *n* elements, after we check the first element, we still have *n* – 1 elements to check. With a BST of *n* nodes, however, after checking the root we have whittled the problem down to searching a BST with *n*/2 nodes.

Searching a binary tree is similar in analysis to searching a sorted array. For example, imagine you wanted to find if there is a John King in the phonebook. You could start by flipping to the middle of the phone book. Here, you'd likely find people with last names starting with the letter M. Because K comes before M alphabetically, you would then flip halfway between the start of the phonebook and the page you had reached in the Ms. Here, you might end up in the Hs. Since K comes after H, you'd flip half way between the Hs and the Ms. This time you might hit the Ks, where you could quickly see if James King was listed.

This is similar to searching a BST. With an ideally arranged BST the midpoint is the root. We then traverse down the tree, navigating to the left and right children as needed. These approaches cut the search space in half at each step. Such algorithms that exhibit this property have an asymptotic running time of $\log_2 n$, commonly abbreviated lg *n*. Recall from our mathematical discussions in Part 1 of this article series that $\log_2 n = y$ means that $2^y = n$. That is, as *n* grows, $\log_2 n$ grows very slowly. The growth rate of $\log_2 n$ compared to linear grown is shown in the graph in Figure 7. Due to $\log_2 n$'s slower growth than linear time, algorithms that run in asymptotic time $\log_2 n$ are said to be sublinear.

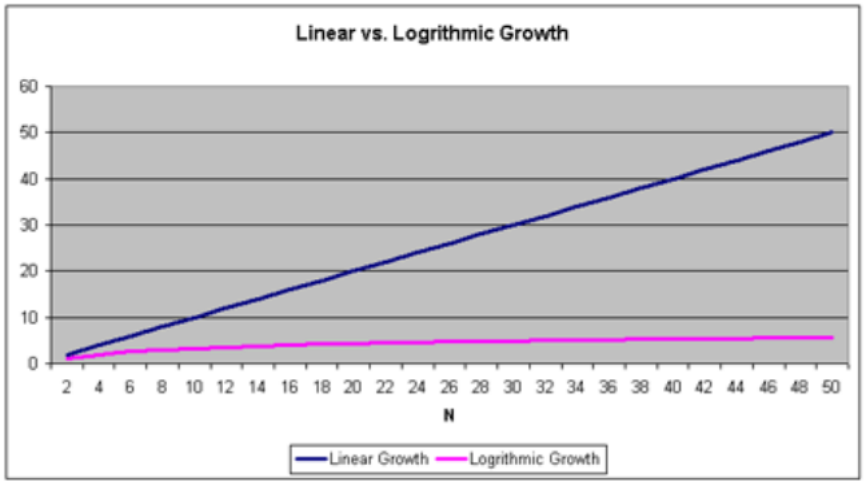


Figure 7. Comparison of linear growth rate vs. $\log_2 n$

While it may appear that the logarithmic curve is flat, it is increasing, albeit rather slowly. To appreciate the difference between linear and sublinear growth, consider searching an array with 1,000 elements versus searching a BST with 1,000 elements. For the array, we'll have to search up to 1,000 elements. For the BST, we'd ideally have to search no more than *ten* nodes! (Note that $\log_{10}$ 1024 equals 10.)

Throughout our analysis of the BST search algorithm, I've repeatedly used the word "ideally." This is because the search time for a BST depends upon its *topology*, or how the nodes are laid out with respect to one another. For a binary tree like the one in Figure 6, each comparison stage in the search algorithm trims the search space in half. However, consider the BST shown in Figure 8, whose topology is synonymous to how an array is arranged.

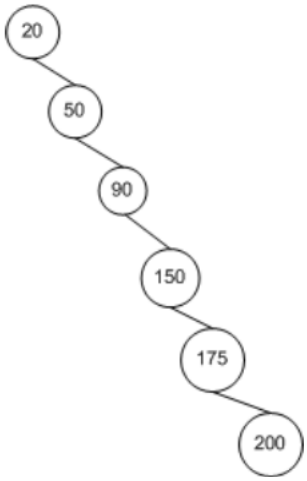


Figure 8. Example of BST that will be searched in linear time

Searching the BST in Figure 8 will take linear time because after each comparison the problem space is only reduced by 1 node, not by half of the current nodes, as with the BST in Figure 6.

Therefore, the time it takes to search a BST is dependent upon its topology. In the best case, the time is on the order of $\log_2 n$, but in the worst case it requires linear time. As we'll see in the next section, the topology of a BST is dependent upon the order with which the nodes are inserted. Therefore, the order with which the nodes are inserted affects the running time of the BST search algorithm.

Inserting Nodes into a BST

We've seen how to search a BST to determine if a particular node exists, but we've yet to look at how to add a new node. When adding a new node we can't arbitrarily add the new node; rather, we have to add the new node such that the binary search tree property is maintained.

When inserting a new node we will always insert the new node as a leaf node. The only challenge, then, is finding the node in the BST which will become this new node's parent. Like with the searching algorithm, we'll be making comparisons between a node *c* and the node to be inserted, *n*. We'll also need to keep track of *c*'s parent node. Initially, *c* is the BST root and *parent* is a null reference. Locating the new parent node is accomplished by using the following algorithm:

1. If *c* is a null reference, then *parent* will be the parent of *n*. If *n*'s value is less than *parent*'s value, then *n* will be *parent*'s new left child; otherwise *n* will be *parent*'s new right child.
2. Compare *c* and *n*'s values.
3. If *c*'s value equals *n*'s value, then the user is attempting to insert a duplicate node. Either simply discard the new node, or raise an exception. (Note that the nodes' values in a BST must be unique.)
4. If *n*'s value is less than *c*'s value, then *n* must end up in *c*'s left subtree. Let *parent* equal *c* and *c* equal *c*'s left child, and return to step 1.
5. If *n*'s value is greater than *c*'s value, then *n* must end up in *c*'s right subtree. Let *parent* equal *c* and *c* equal *c*'s right child, and return to step 1.

This algorithm terminates when the appropriate leaf is found, which attaches the new node to the BST by making the new node an appropriate child of *parent*. There's one special case you have to worry about with the insert algorithm: if the BST does not contain a root, then there *parent* will be null, so the step of adding the new node as a child of *parent* is bypassed; furthermore, in this case the BST's root must be assigned to the new node.

Figure 9 depicts the BST insert graphically.

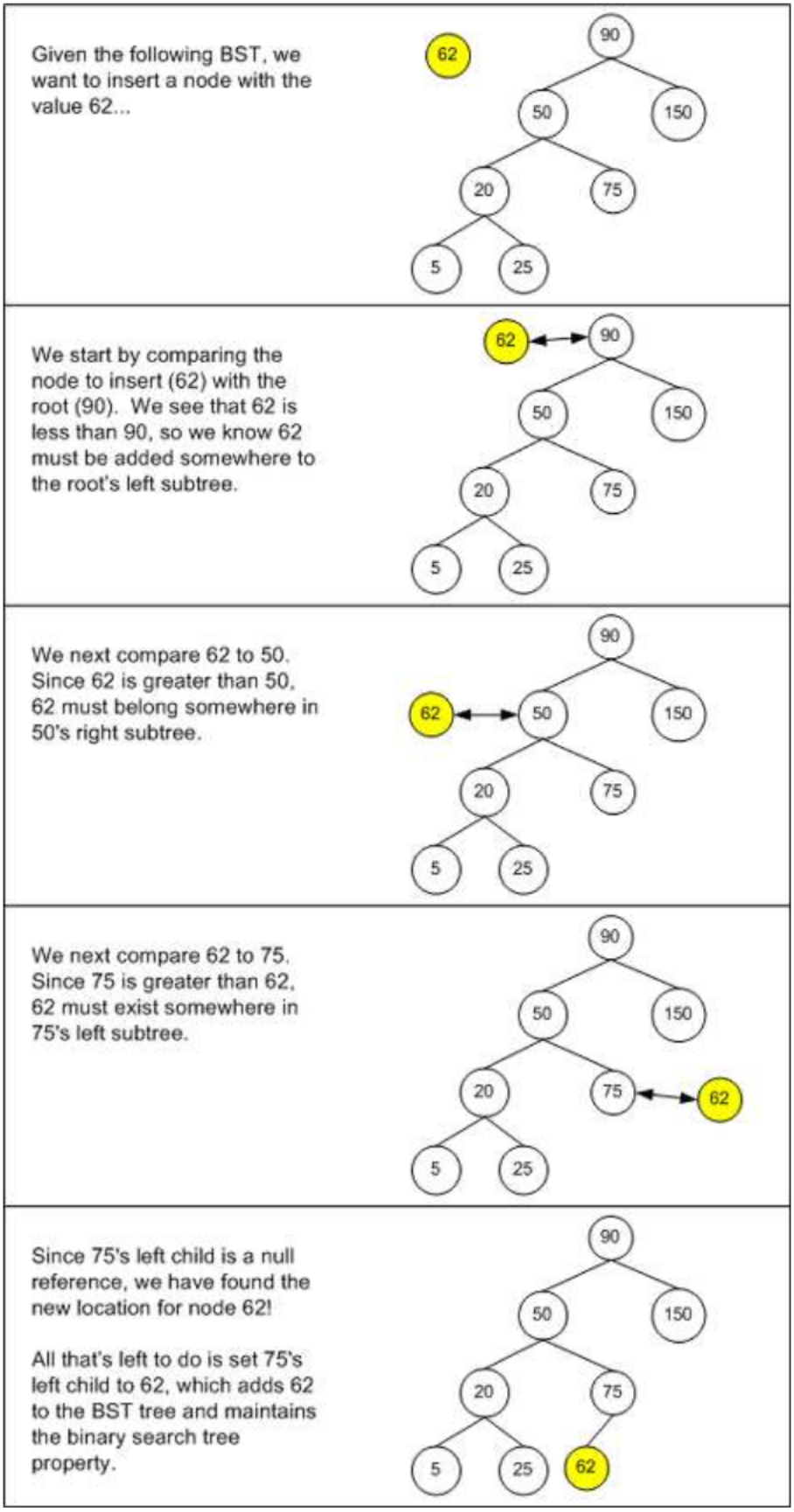


Figure 9. Insert into a BST

Both the BST search and insert algorithms share the same running time—$\log_2 n$ in the best case, and linear in the worst case. The insert algorithm's running time mimics the search's because it, essentially, uses the same tactics used by the search algorithm to find the location for the newly inserted node.

The Order of Insertion Determines the BST's Topology

Since newly inserted nodes into a BST are inserted as leaves, the order of insertion directly affects the topology of the BST itself. For example, imagine we insert the following nodes into a BST: 1, 2, 3, 4, 5, and 6. When 1 is inserted, it is insert as the root. Next, 2 is inserted as 1's right child. 3 is inserted as 2's right child, 4 as 3's right child, and so on. The resulting BST is one whose structure is precisely that of the BST in Figure 8.

If the values 1, 2, 3, 4, 5, and 6 had been inserted in a more intelligent manner, the BST would have had more breadth, and would have looked more like the tree in Figure 6. The ideal insertion order would be: 4, 2, 5, 1, 3, 6. This would put 4 at the root, 2 as 4's left child, 5 as 4's right child, 1 and 3 as 2's left and right children, and 6 as 5's right child.

Because the topology of a BST can greatly affect the running time of search, insert, and (as we will see in the next section) delete, inserting data in ascending or descending order (or in near order) can have devastating results on the efficiency of the BST. We'll discuss this topic in more detail at the article's end, in the "Binary Search Trees in the Real-World" section.

Deleting Nodes from a BST

Deleting nodes from a BST is slightly more difficult than inserting a node because deleting a node that has children requires that some other node be chosen to replace the hole created by the deleted node. If the node to replace this hole is not chosen with care, the binary search tree property may be violated. For example, consider the BST in Figure 6. If the node 150 is deleted, some node must be moved to the hole created by node 150's deletion. If we arbitrarily choose to move, say node 92 there, the BST property is deleted since 92's new left subtree will have nodes 95 and 111, both of which are greater than 92 and thereby violating the binary search tree property.

The first step in the algorithm to delete a node is to first locate the node to delete. This can be done using the searching algorithm discussed earlier, and therefore has a $\log_2 n$ running time. Next, a node from the BST must be selected to take the deleted node's position. There are three cases to consider when choosing the replacement node, all of which are illustrated in Figure 10.

- **Case 1:** If the node being deleted has no right child, then the node's left child can be used as the replacement. The binary search tree property is maintained because we know that the deleted node's left subtree itself maintains the binary search tree property, and that the values in the left subtree are all less than or all greater than the deleted node's parent, depending on whether the deleted node is a left or right child. Therefore, replacing the deleted node with its left subtree will maintain the binary search tree property.
- **Case 2:** If the deleted node's right child has no left child, then the deleted node's right child can replace the deleted node. The binary search tree property is maintained because the deleted node's right child is greater than all nodes in the deleted node's left subtree and is either greater than or less than the deleted node's parent, depending on whether the deleted node was a right or left child. Therefore, replacing the deleted node with its right child will maintain the binary search tree property.
- **Case 3:** Finally, if the deleted node's right child does have a left child, then the deleted node needs to be replaced by the deleted node's right child's left-most descendant. That is, we replace the deleted node with the deleted node's right subtree's *smallest* value.

  Note Realize that for any BST, the smallest value in the BST is the left-most node, while the largest value is the right-most node.

This replacement choice maintains the binary search tree property because it chooses the smallest node from the deleted node's right subtree, which is guaranteed to be larger than all node's in the deleted node's left subtree. Too, since it's the smallest node from the deleted node's right subtree, by placing it at the deleted node's position, all of the nodes in its right subtree will be greater.

Figure 10 illustrates the node to choose for replacement for each of the three cases.

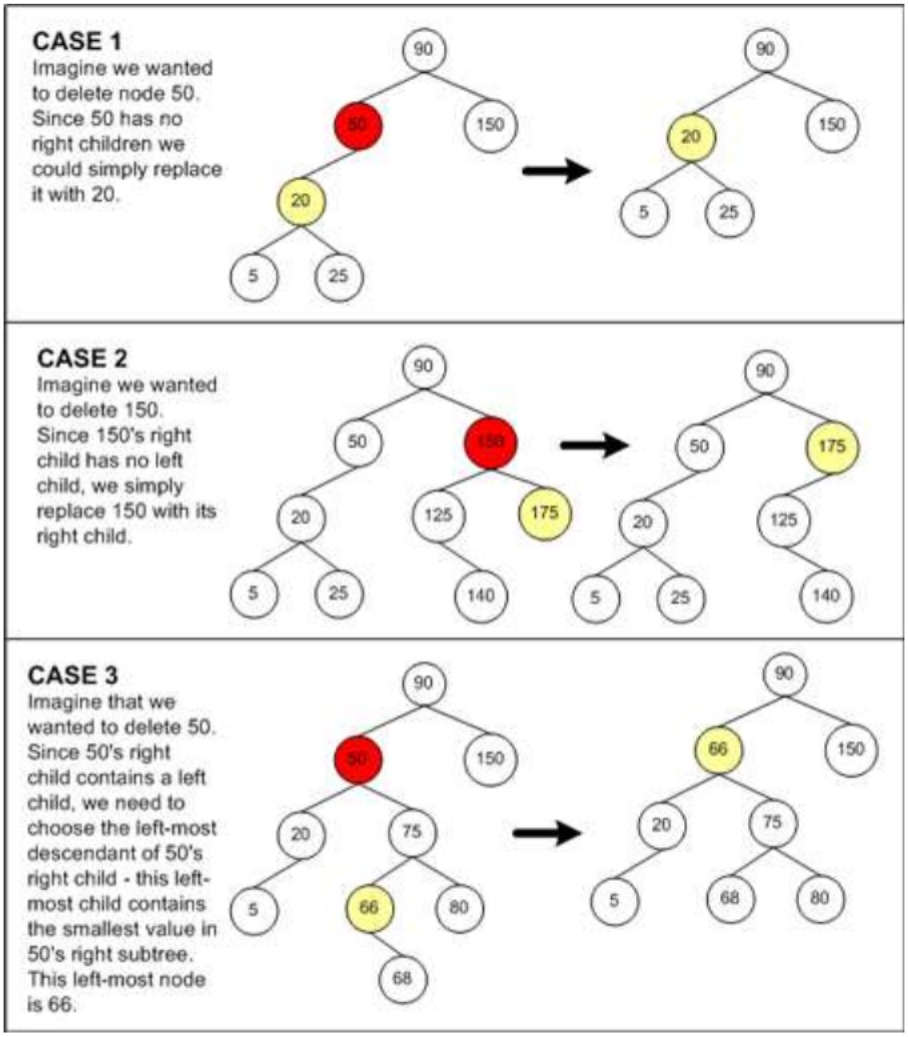


Figure 10. Cases to consider when deleting a node

As with the insert and searching algorithms, the asymptotic running time of delete is dependent upon the BST's topology. Ideally, the running time is on the order of $\log_2 n$. However, in the worst case, it takes linear time.

Traversing the Nodes of a BST

With the linear, contiguous ordering of an array's elements, iterating through an array is a straightforward manner: start at the first array element, and step through each element sequentially. For BSTs there are three different kinds of traversals that are commonly used:

- Preorder traversal
- Inorder traversal
- Postorder traversal

Essentially, all three traversals work in roughly the same manner. They start at the root and visit that node and its children. The difference among these three traversal methods is the order with which they visit the node itself

versus visiting its children. To help clarify this, consider the BST in Figure 11. (Note that the BST in Figure 11 is the same as the BST in Figure 6 and is repeated here for convenience.)

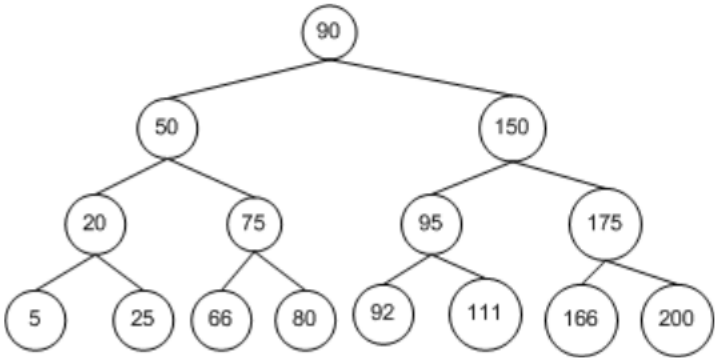


Figure 11. A sample Binary Search Tree

Preorder Traversal

Preorder traversal starts by visiting the current node—call it *c*—then its left child, and then its right child. Starting with the BST's root as *c*, this algorithm can be written out as:

1. Visit *c*. This might mean printing out the value of the node, adding the node to a List, or something else. It depends on what you want to accomplish by traversing the BST.
2. Repeat step 1 using *c*'s left child.
3. Repeat step 1 using *c*'s right child.

Imagine that in step 1 of the algorithm we were printing out the value of *c*. In this case, what would the output be for a preorder traversal of the BST in Figure 11? Well, starting with step 1 we would print the root's value. Step 2 would have us repeat step 1 with the root's left child, so we'd print 50. Step 2 would have us repeat step 1 with the root's left child's left child, so we'd print 20. This would repeat all the way down the left side of the tree. When 5 was reached, we'd first print out its value (step 1). Since there are no left or right children of 5, we'd return to node 20, and perform its step 3, which is repeating step 1 with 20's right child, or 25. Since 25 has no children, we'd return to 20, but we've done all three steps for 20, so we'd return to 50, and then take on step 3 for node 50, which is repeating step 1 for node 50's right child. This process would continue on until each node in the BST had been visited. The output for a preorder traversal of the BST in Figure 11 would be: 90, 50, 20, 5, 25, 75, 66, 80, 150, 95, 92, 111, 175, 166, 200.

Understandably, this may be a bit confusing. Perhaps looking at some code will help clarify things. The following code shows a method to iterate through the items of BST in a preorder traversal. Note that this method takes in a `BinaryTreeNode` class instance as one of its input parameters. This input node is the node *c* from the list of the algorithm's steps. Realize that a preorder traversal of the BST would begin by calling this method, passing in the BST's root.

```
void PreorderTraversal(Node current)
{
   if (current != null)
   {
            // Output the value of the current node
      Console.WriteLine(current.Value);

            // Recursively print the left and right children
      PreorderTraversal(current.Left);
      PreorderTraversal(current.Right);
   }
}
```

Inorder Traversal

Inorder traversal starts by visiting the current node's left child, then the current node, and then its right child. Starting with the BST's root as *c*, this algorithm can be written out as:

1. Repeat step 1 using c's left child.
2. Visit c. This might mean printing out the value of the node, adding the node to an ArrayList, or something else. It depends on what you want to accomplish by traversing the BST.
3. Repeat step 1 using c's right child.

The code for `InorderTraversal()` is just like `PreorderTraversal()` except that adding the `current` Node's data to the `StringBuilder` occurs *after* another call to `InorderTraversal()`, passing in `current`'s left child.

```
void InorderTraversal(Node current)
{
   if (current != null)
   {
            // Visit the left child...
      InorderTraversal(current.Left);

            // Output the value of the current node
      Console.WriteLine(current.Value);

            // Visit the right child...
      InorderTraversal(current.Right);
   }
}
```

Applying an inorder traversal to the BST in Figure 11, the output would be: 5, 20, 25, 50, 66, 75, 80, 90, 92, 95, 111, 150, 166, 175, 200. Note that the results returned are in ascending order.

Postorder Traversal

Finally, postorder traversal starts by visiting the current node's left child, then its right child, and finally the current node itself. Starting with the BST's root as *c*, this algorithm can be written out as:

1. Repeat step 1 using c's left child.
2. Repeat step 1 using c's right child.
3. Visit c. This might mean printing out the value of the node, adding the node to an ArrayList, or something else. It depends on what you want to accomplish by traversing the BST.

The output of a postorder traversal for the BST in Figure 11 would be: 5, 25, 20, 66, 80, 75, 50, 92, 111, 95, 166, 200, 175, 150, 90.

Note The download included with this article contains the complete source code for the `BinarySearchTree` and `BinaryTree` classes, along with a Windows Forms testing application for the BST class. Of particular interest, the Windows Forms application allows you to view the contents of the BST in either preorder, inorder, or postorder.

Realize that all three traversal times exhibit linear asymptotic running time. This is because each traversal option visits each and every node in the BST precisely once. So, if the number of nodes in the BST are doubled, the amount of work for a traversal doubles as well.

The Cost of Recursion

Recursive functions are often ideal for visualizing an algorithm, as they can often eloquently describe an algorithm in a few short lines of code. However, when iterating through a data structure's elements in practice, recursive

functions are usually sub-optimal. Iterating through a data structure's elements involves stepping through the items and returning them to the developer utilizing the data structure, one at a time. Recursion is not suited to stopping abruptly at each step of the process. For this reason, the enumerator for the `BinarySearchTree` class uses a non-recursive solution to iterate through the elements.

Implementing a BST Class

The .NET Framework Base Class Library does not include a binary search tree class, so let's create one ourselves. The `BinarySearchTree` class can reuse the `BinaryTreeNode` class we examined earlier. Over the next few sections we'll look at each of the major methods of the class. For the class's full source code, be sure to download the source code that accompanies this article, which also includes a Windows Forms application for testing the BST class.

Searching for a Node

The reason BSTs are important to study is that they offer sublinear search times. Therefore, it only makes sense to first examine the BSTs `Contains()` method. The `Contains()` method accepts a single input parameter and returns a Boolean indicating if that value exists in the BST.

`Contains()` starts at the root and iteratively percolates down the tree until it either reaches a `null` reference, in which case `false` is returned, or until it finds the node being searched for, in which case `true` is returned. In the `while` loop, `Contains()` compares the `Value` of the current `BinaryTreeNode` instance against the data being searched for, and snakes its way down the right or left subtree accordingly. The comparison is performed by a private member variable, `comparer`, which is of type `IComparer<T>` (where `T` is the type defined via the Generics syntax for the BST). By default, `comparer` is assigned the default `Comparer` class for the type T, although the `BST` class's constructor has an overload to specify a custom `Comparer` class instance.

```
public bool Contains(T data)

{
    // search the tree for a node that contains data
    BinaryTreeNode<T> current = root;
    int result;
    while (current != null)
    {
        result = comparer.Compare(current.Value, data);
        if (result == 0)
            // we found data
            return true;
        else if (result > 0)
            // current.Value > data, search current's left subtree
            current = current.Left;
        else if (result < 0)
            // current.Value < data, search current's right subtree
            current = current.Right;
    }

    return false;       // didn't find data
}
```

Adding a Node to the BST

Unlike the `BinaryTree` class we created earlier, the `BinarySearchTree` class does not provide direct access to its root. Rather, nodes are added to the BST through the BST's `Add()` method. `Add()` takes as input the item to add to the BST, which is then percolated down the tree, looking for its new parent. (Recall that the any new nodes added to a BST will be added as leaf nodes.) Once the new node's parent is found, the node is made either the left or right child of the parent, depending on if its value is less than or greater than the parent's value.

```
public virtual void Add(T data)
{
    // create a new Node instance
    BinaryTreeNode<T> n = new BinaryTreeNode<T>(data);
    int result;

    // now, insert n into the tree
    // trace down the tree until we hit a NULL
    BinaryTreeNode<T> current = root, parent = null;
    while (current != null)
    {
        result = comparer.Compare(current.Value, data);
        if (result == 0)
            // they are equal - attempting to enter a duplicate - do nothing
            return;
        else if (result > 0)
        {
            // current.Value > data, must add n to current's left subtree
            parent = current;
            current = current.Left;
        }
        else if (result < 0)
        {
            // current.Value < data, must add n to current's right subtree
            parent = current;
            current = current.Right;
        }
    }

    // We're ready to add the node!
    count++;
    if (parent == null)
        // the tree was empty, make n the root
        root = n;
    else
    {
        result = comparer.Compare(parent.Value, data);
        if (result > 0)
            // parent.Value > data, therefore n must be added to the left subtree
            parent.Left = n;
        else
            // parent.Value < data, therefore n must be added to the right subtree
            parent.Right = n;
    }
}
```

As you can see by examining the code, if the user attempts to add a duplicate the `Add()` method does nothing. That is, the BST cannot contain nodes with duplicate values. If you want, rather than simply returning, you could alter this code so inserting a duplicate raises an exception.

Deleting a Node from the BST

Recall that deleting a node from a BST is the trickiest of the BST operations. The trickiness is due to the fact that deleting a node from a BST necessitates that a replacement node be chosen to occupy the space once held by the deleted node. Care must be taken when selecting this replacement node so that the binary search tree property is maintained.

Earlier, in the "Deleting Nodes from a BST" section, we discussed how there were three different scenarios for deciding what node to choose to replace the deleted node. These cases were summarized in Figure 10. Below you can see how the cases are identified and handled in the `Remove()` method.

```
public bool Remove(T data)
{
    // first make sure there exist some items in this tree
    if (root == null)
        return false;       // no items to remove

    // Now, try to find data in the tree
    BinaryTreeNode<T> current = root, parent = null;
    int result = comparer.Compare(current.Value, data);
    while (result != 0)
    {
        if (result > 0)
        {
            // current.Value > data, if data exists it's in the left subtree
            parent = current;
            current = current.Left;
        }
        else if (result < 0)
        {
            // current.Value < data, if data exists it's in the right subtree
            parent = current;
            current = current.Right;
        }

        // If current == null, then we didn't find the item to remove
        if (current == null)
            return false;
        else
            result = comparer.Compare(current.Value, data);
    }

    // At this point, we've found the node to remove
    count--;

    // We now need to "rethread" the tree
    // CASE 1: If current has no right child, then current's left child becomes
    //         the node pointed to by the parent
    if (current.Right == null)
    {
        if (parent == null)
            root = current.Left;
        else
        {
            result = comparer.Compare(parent.Value, current.Value);
            if (result > 0)
                // parent.Value > current.Value, so make current's left child a
                //  left child of parent
                parent.Left = current.Left;
            else if (result < 0)
                // parent.Value < current.Value, so make current's left child a
                //  right child of parent
                parent.Right = current.Left;
        }
    }
```

```
    // CASE 2: If current's right child has no left child, then current's right
    //         child replaces current in the tree
    else if (current.Right.Left == null)
    {
        current.Right.Left = current.Left;

        if (parent == null)
            root = current.Right;
        else
        {
            result = comparer.Compare(parent.Value, current.Value);
            if (result > 0)
                // parent.Value > current.Value, so make current's right child a
                // left child of parent
                parent.Left = current.Right;
            else if (result < 0)
                // parent.Value < current.Value, so make current's right child a
                // right child of parent
                parent.Right = current.Right;
        }
    }

    // CASE 3: If current's right child has a left child, replace current with current's
    //          right child's left-most descendent
    else
    {
        // We first need to find the right node's left-most child
        BinaryTreeNode<T> leftmost = current.Right.Left, lmParent = current.Right;
        while (leftmost.Left != null)
        {
            lmParent = leftmost;
            leftmost = leftmost.Left;
        }

        // the parent's left subtree becomes the leftmost's right subtree

        lmParent.Left = leftmost.Right;

        // assign leftmost's left and right to current's left and right children
        leftmost.Left = current.Left;
        leftmost.Right = current.Right;

        if (parent == null)
            root = leftmost;
        else
        {
            result = comparer.Compare(parent.Value, current.Value);
            if (result > 0)
                // parent.Value > current.Value, so make leftmost a left child of parent
                parent.Left = leftmost;
            else if (result < 0)
                // parent.Value < current.Value, so make leftmost a right child of parent
                parent.Right = leftmost;
        }
    }

    return true;
}
```

The `Remove()` method returns a Boolean to indicate whether or not the item was removed successfully from the binary search tree. False is returned in the case where the item to be removed was not found within the tree.

The Remaining BST Methods and Properties

There are a few other BST methods and properties not examined here in this article. Be sure to download this article's accompanying code for a complete look at the BST class. The remaining methods and properties are:

- `Clear()`: removes all of the nodes from the BST.
- `CopyTo(Array, index[, TraversalMethods])`: copies the contents of the BST to a passed-in array. By default, an inorder traversal is used, although a specific traversal method can be specified. The `TraversalMethods` enumeration options are `Preorder`, `Inorder`, and `Postorder`.
- `GetEnumerator([TraversalMethods])`: provides iteration through the BST using inorder traversal, by default. Other traversal methods can be optionally specified.
- `Count`: a public, read-only property that returns the number of nodes in the BST.
- `Preorder`, `Inorder`, and `Postorder`: these three properties return `IEnumerable<T>` instances that can be used to enumerate the items in the BST in a specified traversal order.

The `BST` class implements the `ICollection`, `ICollection<T>`, `IEnumerable`, and `IEnumerable<T>` interfaces.

Binary Search Trees in the Real-World

While binary search trees ideally exhibit sublinear running times for insertion, searches, and deletes, the running time is dependent upon the BST's topology. The topology, as we discussed in the "Inserting Nodes into a BST" section, is dependent upon the order with which the data is added to the BST. Data being entered that is ordered or near-ordered will cause the BST's topology to resemble a long, skinny tree, rather than a short and fat one. In many real-world scenarios, data is naturally in an ordered or near-ordered state.

The problem with BSTs is that they can become easily *unbalanced*. A *balaced* binary tree is one that exhibits a good ratio of breadth to depth. As we will examine in the next part of this article series, there are a special class of BSTs that are self-balancing. That is, as new nodes are added or existing nodes are deleted, these BSTs automatically adjust their topology to maintain an optimal balance. With an ideal balance, the running time for inserts, searches, and deletes, even in the worst case, is $\log_2 n$.

In the next installment of this article series, we'll look at a couple self-balancing BST derivatives, including the red-black tree, and then focus on a data structure known as a SkipList, which exhibits the benefits of self-balancing binary trees, but without the need for restructuring the topology. Restructuring the topology, while efficient, requires a good deal of difficult code and is anything but readable.

Until then, happy programming!

Appendix A: Definitions

The following table is an alphabetized list of definitions for terms used throughout this article.

Binary tree	A *tree* where each node has at most two children.
Binary search tree	A *binary tree* that exhibits the following property: for any node *n*, every descendant node's value in the left *subtree* of *n* is less than the value of *n*, and every descendant node's value in the right *subtree* is greater than the value of *n*.
Cycle	A cycle exists if starting from some node *n* there exists a path that returns to *n*.
Internal node	A node that has one or more children.
Leaf node	A node that has no children.
Node	Nodes are the building blocks of *trees*. A node contains some piece of data, a parent, and a set of children. (Note: the *root* node does not contain a parent; *leafnodes* do not contain children.)
Root	The node that has no parent. All trees contain precisely one root.
Subtree	A subtree rooted at node *n* is the tree formed by imaging node *n* was a root. That is, the subtree's nodes are the descendants of *n* and the subtree's root is *n* itself.

Appendix B: Running Time Summaries

The following table lists the common operations that are performed on a BST and their associated running times.

Operation	**Best Case Running Time**	**Worst Case Running Time**
Search	$\log_2 n$	*n*
Insert	$\log_2 n$	*n*
Delete	$\log_2 n$	*n*
Preorder Traversal		*n*
Inorder Traversal		*n*
Postorder Traversal		*n*

References

- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.

http://msdn.microsoft.com/en-us/library/ms379573

Visual Studio 2005 Technical Articles

An Extensive Examination of Data Structures Using C# 2.0

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: This article, the **fourth** in the series, begins with a quick examination of **AVL trees and red-black trees**, which are two different self-balancing binary search tree data structures. The remainder of the article examines the skip list data structure. **Skip lists** are an ingenious data structure that turns a linked list into a data structure that offers the same running time as the more complex self-balancing tree data structures. As we'll see, the skip list works its magic by giving each element in the linked list a random "height." In the latter part of the article, we'll be building a skip list class in C#, which can be downloaded. (28 printed pages)

Download the DataStructures20.msi sample file.

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 3 of this article series we looked at the general *tree* data structure. A tree is a data structure that consists of nodes, where each node has some value and an arbitrary number of children nodes. Trees are common data structures because many real-world problems exhibit tree-like behavior. For example, any sort of hierarchical relationship among people, things, or objects can be modeled as a tree.

A *binary tree* is a special kind of tree, one that limits each node to no more than two children. A *binary search tree*, or BST, is a binary tree whose nodes are arranged such that for every node *n*, all of the nodes in *n*'s left subtree have a value less than *n*, and all nodes in *n*'s right subtree have a value greater than *n*. As we discussed, in the average case BSTs offer $\log_2 n$ asymptotic time for inserts, deletes, and searches. ($\log_2 n$ is often referred to as *sublinear* because it outperforms linear asymptotic times.)

The disadvantage of BSTs is that in the worst-case their asymptotic running time is reduced to linear time. This happens if the items inserted into the BST are inserted in order or in near-order. In such a case, a BST performs no better than an array. As we discussed at the end of Part 3, there exist self-balancing binary search trees, ones that ensure that, regardless of the order of the data inserted, the tree maintains a $\log_2 n$ running time. In this article, we'll briefly discuss two self-balancing binary search trees: AVL trees and red-black trees. Following that, we'll take an in-depth look at skip lists. Skip lists are a really neat data structure that is much easier to implement than AVL trees or red-black trees, yet still guarantees a running time of $\log_2 n$.

Note This article assumes you have read Part 3 of this article series. If you have not read Part 3, I strongly encourage you to read it thoroughly before continuing on with this fourth installment.

Self-Balancing Binary Search Trees

Recall that new nodes are inserted into a binary search tree at the leaves. That is, adding a node to a binary search tree involves tracing down a path of the binary search tree, taking left's and right's based on the comparison of the value of the current node and the node being inserted, until the path reaches a dead end. At this point, the newly inserted node is plugged into the tree at this reached dead end. Figure 1 illustrates the process of inserting a new node into a BST.

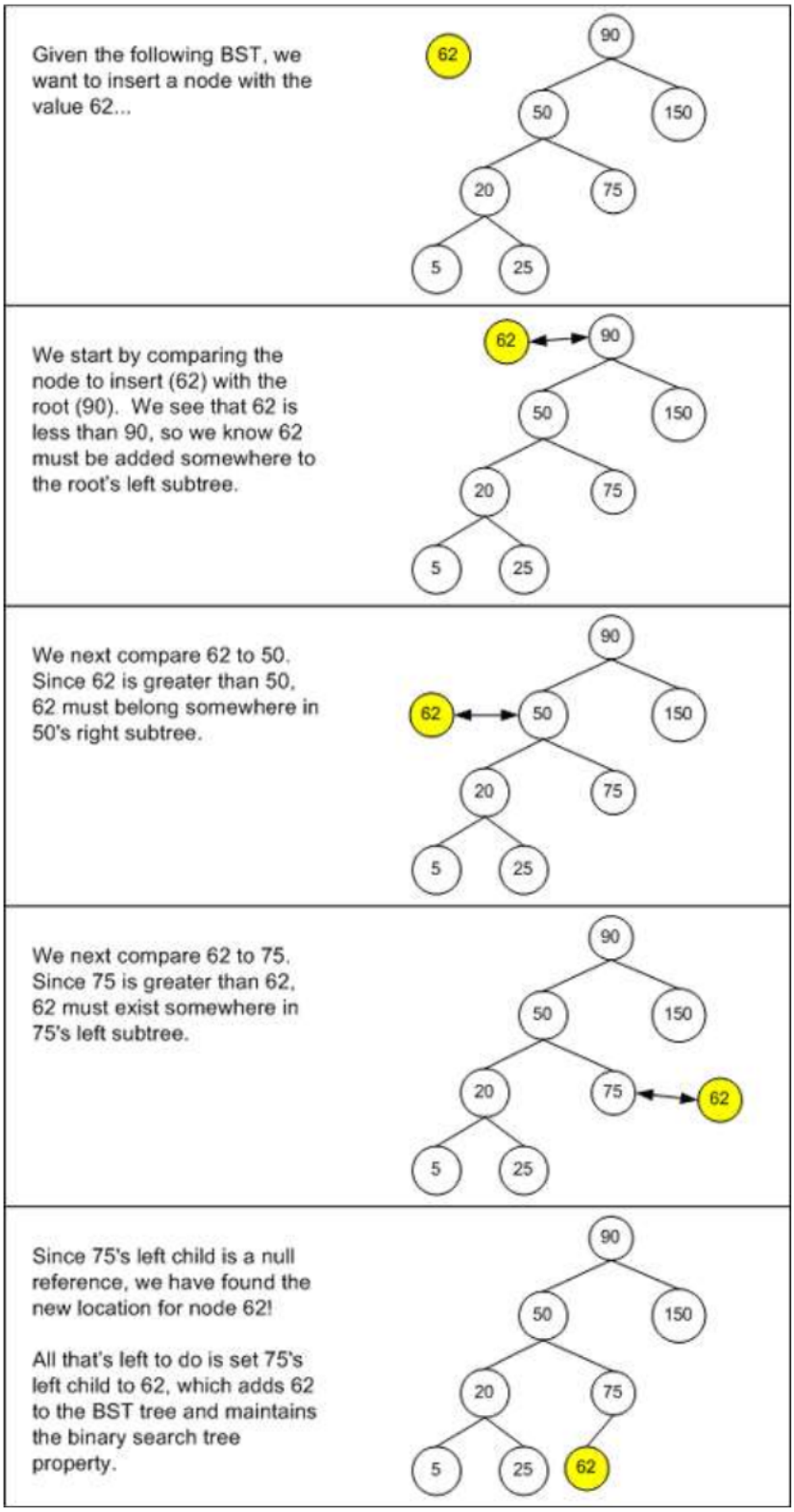


Figure 1. Inserting a new node into a BST

As Figure 1 shows, when making the comparison at the current node, the node to be inserted travels down the left path if its value is less than the current node, and down the right if its value is greater than the current's. Therefore, the structure of the BST is relative to the order with which the nodes are inserted. Figure 2 depicts a BST after nodes with values 20, 50, 90, 150, 175, and 200 have been added. Specifically, these nodes have been added in ascending order. The result is a BST with no breadth. That is, its topology consists of a single line of nodes rather than having the nodes fanned out.

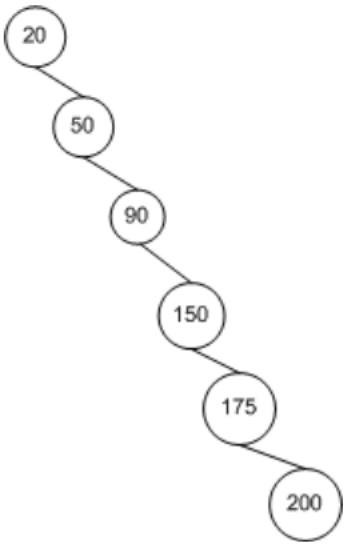


Figure 2. A BST after nodes with values of 20, 50, 90, 150, 175, and 200 have been added

BSTs—which offer sublinear running time for insertions, deletions, and searches—perform optimally when their nodes are arranged in a fanned out manner. This is because when searching for a node in a BST, each single step down the tree reduces the number of nodes that need to be potentially checked by one half. However, when a BST has a topology similar to the one in Figure 2, the running time for the BST's operations are much closer to linear time because each step down the tree only reduces the number of nodes that need to be searched by one. To see why, consider what must happen when searching for a particular value, such as 175. Starting at the root, 20, we must navigate down through each right child until we hit 175. That is, there is no savings in nodes that need to be checked at each step. Searching a BST like the one in Figure 2 is identical to searching an array—each element must be checked one at a time. Therefore, such a structured BST will exhibit a linear search time.

It is important to realize that the running time of a BST's operations is related to the BST's *height*. The height of a tree is defined as the length of the longest path starting at the root. The height of a tree can be defined recursively as follows:

- The height of a node with no children is 0.
- The height of a node with one child is the height of that child plus one.
- The height of a node with two children is one plus the greater height of the two children.

To compute the height of a tree, start at its leaf nodes and assign them a height of 0. Then move up the tree using the three rules outlined to compute the height of each leaf nodes' parent. Continue in this manner until every node of the tree has been labeled. The height of the tree, then, is the height of the root node. Figure 3 shows a number of binary trees with their height computed at each node. For practice, take a second to compute the heights of the trees yourself to make sure your numbers match up with the numbers presented in the figure below.

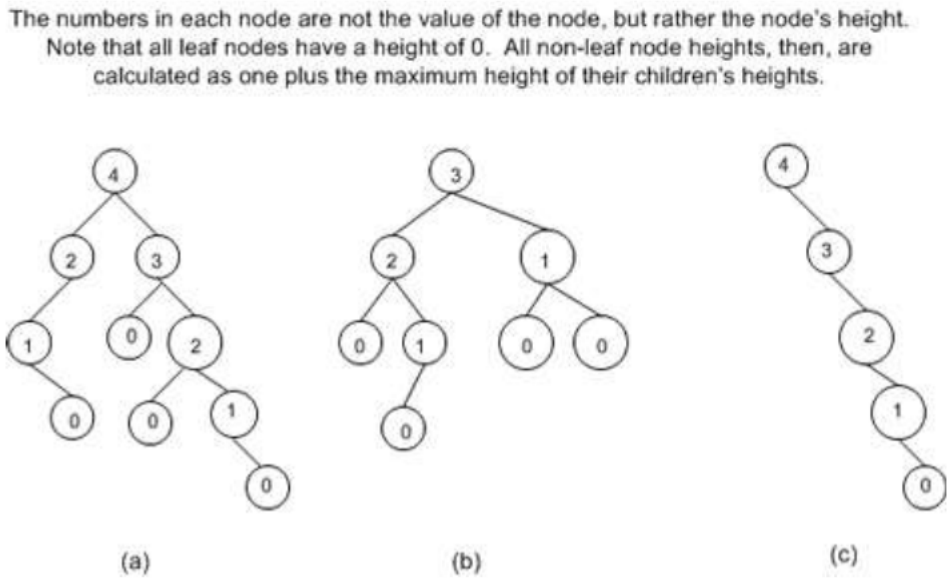


Figure 3. Example binary trees with their height computed at each node

A BST exhibits $\log_2 n$ running times when its height, when defined in terms of the number of nodes, *n*, in the tree, is near the floor of $\log_2 n$. (The floor of a number *x* is the greatest integer less than *x*. So, the floor of 5.38 would be 5 and the floor of 3.14159 would be 3. For positive numbers *x*, the floor of *x* can be found by simply truncating the decimal part of *x*, if any.) Of the three trees in Figure 3, tree (b) has the best height to number of nodes ratio, as the height is 3 and the number of nodes present in the tree is 8. As we discussed in Part 1 of this article series, $\log_a b = y$ is another way of writing $a^y = b$. $\log_2 8$, then, equals 3, because $2^3 = 8$. Tree (a) has 10 nodes and a height of 4. $\log_2 10$ equals 3.3219 and change, the floor of that being 3. So, 4 is not the ideal height. Notice that by rearranging the topology of tree (a)—by moving the far-bottom right node to the child of one of the non-leaf nodes with only one child—we could reduce the tree's height by one, thereby giving the tree an optimal height to node ratio. Finally, tree (c) has the worst height to node ratio. With its 5 nodes it could have an optimal height of 2, but due to its linear topology is has a height of 4.

The challenge we are faced with, then, is ensuring that the topology of the resulting BST exhibits an optimal ratio of height to the number of nodes. Because the topology of a BST is based upon the order in which the nodes are inserted, intuitively you might opt to solve this problem by ensuring that the data that's added to a BST is not added in near-sorted order. While this is possible if you know the data that will be added to the BST beforehand, it might not be practical. If you are not aware of the data that will be added—like if it's added based on user input, or is added as it's read from a sensor—then there is no hope of guaranteeing the data is not inserted in near-sorted order. The solution, then, is not to try to dictate the order with which the data is inserted, but to ensure that after each insertion the BST remains *balanced*. Data structures that are designed to maintain balance are referred to as *self-balancing binary search trees*.

A *balanced tree* is a tree that maintains some predefined ratio between its height and breadth. Different data structures define their own ratios for balance, but all have it close to $\log_2 n$. A self-balancing BST, then, exhibits $\log_2 n$ asymptotic running time. There are numerous self-balancing BST data structures in existence, such as AVL trees, red-black trees, 2-3 trees, 2-3-4 trees, splay trees, B-trees, and others. In the next two sections, we'll take a brief look at two of these self-balancing trees—AVL trees and red-black trees.

Examining AVL Trees

In 1962 Russian mathematicians G. M. **A**ndel'son-**V**el-skii and E. M. **L**andis invented the first self-balancing BST, called an AVL tree. AVL trees must maintain the following balance property—for every node *n*, the height of *n*'s left and right subtrees can differ by at most 1. The height of a node's left or right subtree is the height computed for its left or right node using the technique discussed in the previous section. If a node has only one child, then the height of childless subtree is defined to be -1.

Figure 4 shows, conceptually, the height-relationship each node in an AVL tree must maintain. Figure 5 provides three examples of BSTs. The numbers in the nodes represent the nodes' values; the numbers to the right and left of each node represent the height of the nodes' left and right subtrees. In Figure 5, trees (a) and (b) are valid AVL trees, but trees (c) and (d) are not, since not all nodes adhere to the AVL balance property.

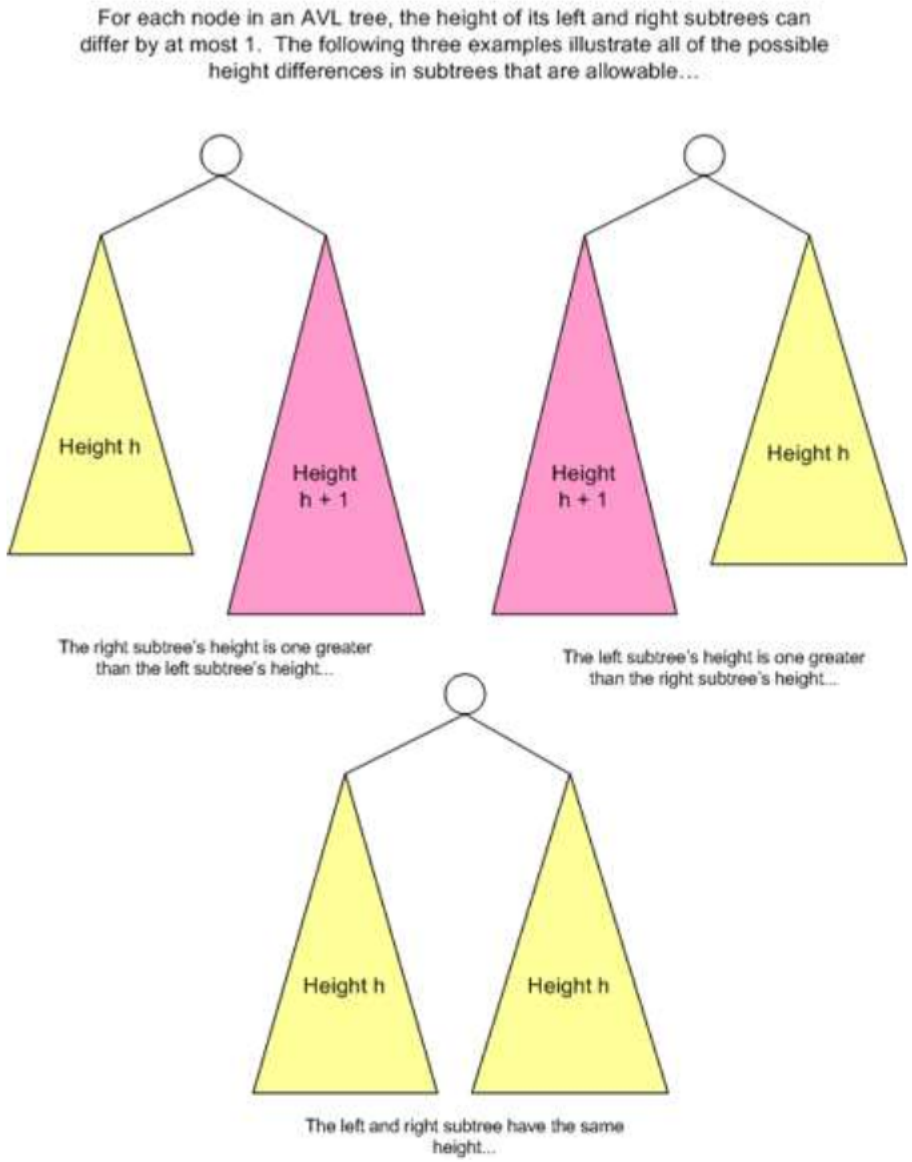


Figure 4. The height of left and right subtrees in an AVL tree cannot differ by more than one.

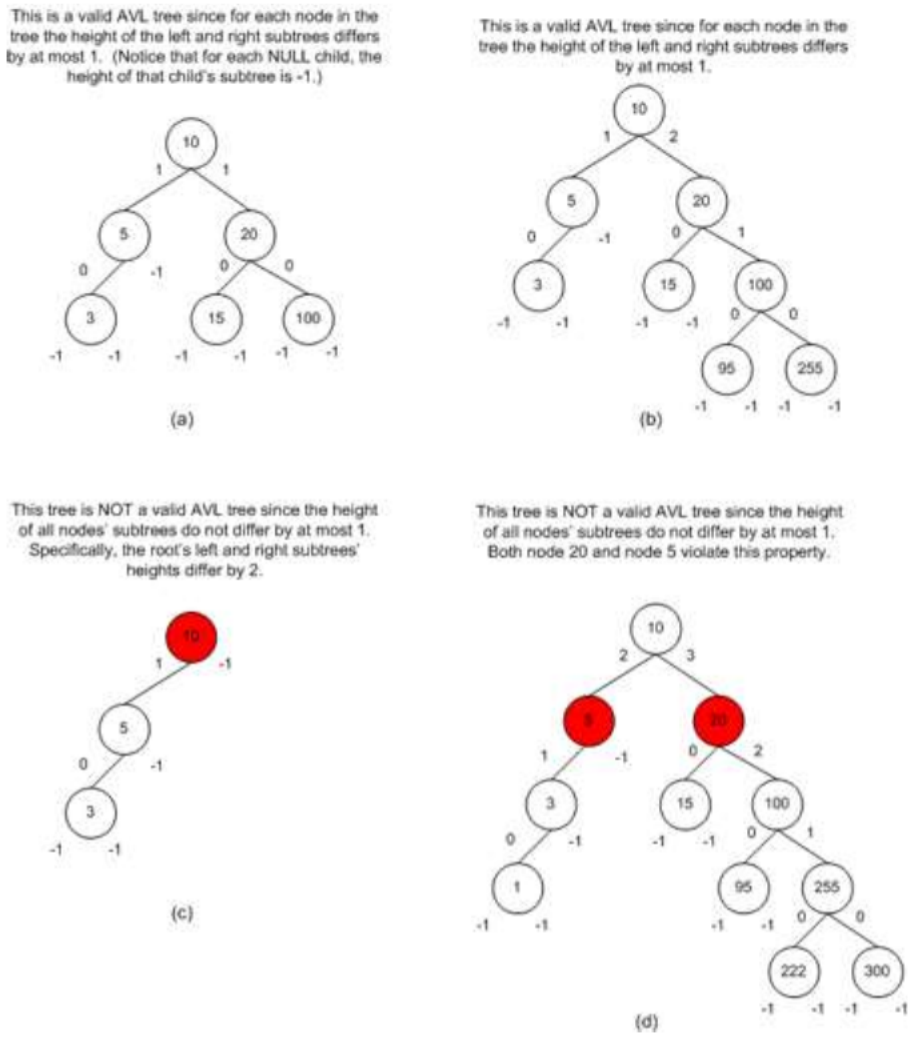


Figure 5. Example trees, where (a) and (b) are valid AVL trees, but (c) and d are not.

Note Realize that AVL trees are binary search trees, so in addition to maintaining a balance property, an AVL tree must also maintain the binary search tree property.

When creating an AVL tree data structure, the challenge is to ensure that the AVL balance remains regardless of the operations performed on the tree. That is, as nodes are added or deleted, it is vital that the balance property remains. AVL trees maintain the balance through *rotations*. A rotation slightly reshapes the tree's topology such that the AVL balance property is restored and, just as importantly, the binary search tree property is maintained.

Inserting a new node into an AVL tree is a two-stage process. First, the node is inserted into the tree using the same algorithm for adding a new node to a BST. That is, the new node is added as a leaf node in the appropriate location to maintain the BST property. After adding a new node, it might be the case that adding this new node caused the AVL balance property to be violated at some node along the path traveled down from the root to where the newly inserted node was added. To fix any violations, stage two involves traversing back up the access path, checking the height of the left and right subtrees for each node along this return path. If the heights of the subtrees differs by more than 1, a rotation is performed to fix the anomaly.

Figure 6 illustrates the steps for a rotation on node 3. Notice that after stage 1 of the insertion routine, the AVL tree property was violated at node 5, because node 5's left subtree's height was two greater than its right subtree's height. To remedy this, a rotation was performed on node 3, the root of node 5's left subtree. This rotation fixed the balance inconsistency and also maintained the BST property.

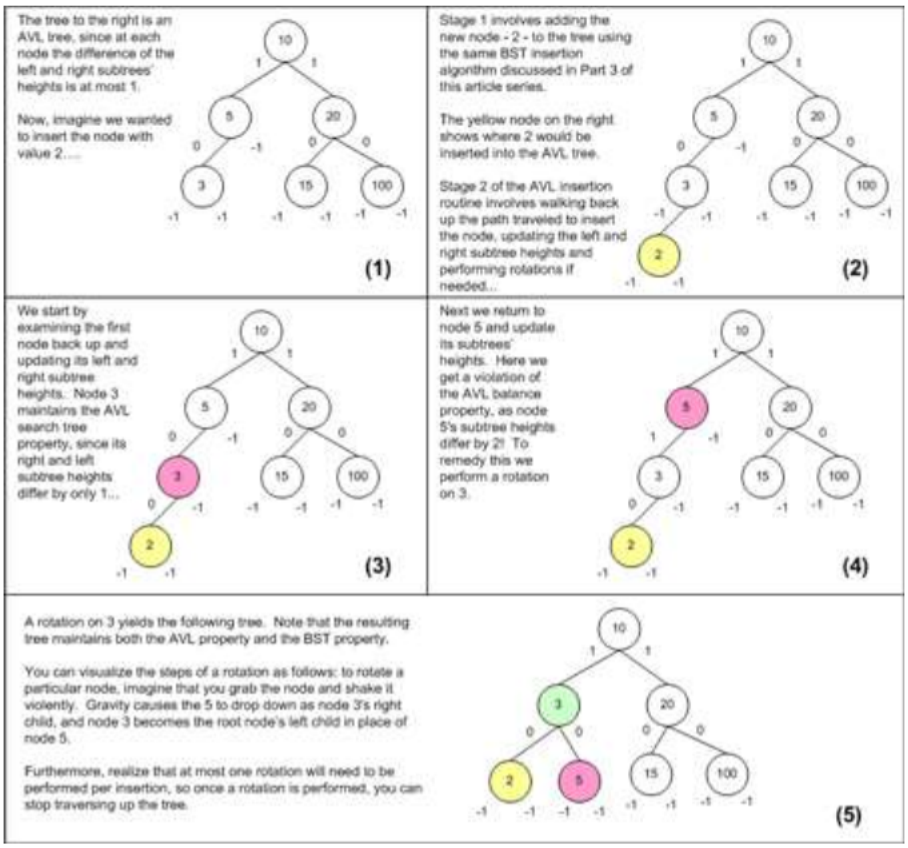


Figure 6. AVL trees stay balanced through rotations

In addition to the simple, single rotation shown in Figure 6, there are more involved rotations than are sometimes required. A thorough discussion of the set of rotations potentially needed by an AVL tree is beyond the scope of this article. What is important to realize is that both insertions and deletions can disturb the balance property to which that AVL trees must adhere. To fix any perturbations, rotations are used.

Note To familiarize yourself with insertions, deletions, and rotations from an AVL tree, check out the AVL tree applet at http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm. This Java applet illustrates how the topology of an AVL tree changes with additions and deletions.

By ensuring that all nodes' subtrees' heights differ by 1 at most, AVL trees guarantee that insertions, deletions, and searches will always have an asymptotic running time of $\log_2 n$, regardless of the order of insertions into the tree.

A Look at Red-Black Trees

The red-black tree data structure was invented in 1972 by Rudolf Bayer, a computer science professor at the Technical University of Munich. In addition to its data and left and right children, the nodes of a red-black tree contain an extra bit of information—a color, which can be either one of two colors, red or black. Red-black trees are complicated further by the concept of a specialized class of node referred to as NIL nodes. NIL nodes are pseudo-nodes that exist as the leaves of the red-black tree. That is, all regular nodes—those with some data associated with them—are internal nodes. Rather than having a NULL pointer for a childless regular node, the node is assumed to have a NIL node in place of that NULL value. This concept can be understandably confusing. Hopefully the diagram in Figure 7 will clear up any confusion.

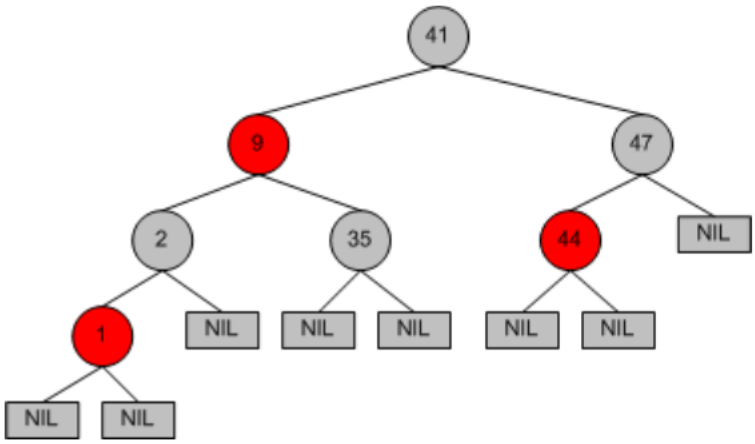


Figure 7. Red-black trees add the concept of a NIL node.

Red-black trees are trees that have the following four properties:

1. Every node is colored either red or black.
2. Every NIL node is black.
3. If a node is red, then both of its children are black.
4. Every path from a node to a descendant leaf contains the same number of black nodes.

The first three properties are pretty self-explanatory. The fourth property, which is the most important of the four, simply states that starting from any node in the tree, the number of black nodes from that node to any leaf (NIL), must be the same. In Figure 7, take the root node as an example. Starting from 41 and going to any NIL, you will encounter the same number of black nodes—3. For example, taking a path from 41 to the left-most NIL node, we start on 41, a black node. We then travel down to node 9, then node 2, which is also black, then node 1, and finally the left-most NIL node. In this journey we encountered three black nodes—41, 2, and the final NIL node. In fact, if we travel from 41 to *any* NIL node, we'll always encounter precisely three black nodes.

Like the AVL tree, red-black trees are another form of self-balancing binary search tree. Whereas the balance property of an AVL tree was explicitly stated as a relationship between the heights of each node's left and right subtrees, red-black trees guarantee their balance in a more conspicuous manner. It can be shown that a tree that implements the four red-black tree properties has a height that is always less than $2 * \log_2(n+1)$, where n is the total number of nodes in the tree. For this reason, red-black trees ensure that all operations can be performed within an asymptotic running time of $\log_2 n$.

Like AVL trees, any time a red-black tree has nodes inserted or deleted, it is important to verify that the red-black tree properties have not been violated. With AVL trees, the balance property was restored using rotations. With red-black trees, the red-black tree properties are restored through recoloring and rotations. Red-black trees are notoriously complex in their recoloring and rotation rules, requiring the nodes along the access path to make decisions based upon their color in contrast to the color of their parents and uncles. (An uncle of a node *n* is the node that is *n*'s parent's sibling node.) A thorough discussion of recoloring and rotation rules is far beyond the scope of this article.

To view the recoloring and rotations of a red-black tree as nodes are added and deleted, check out the red-black tree applet, which can also be accessed at http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm.

A Quick Primer on Linked Lists

One common data structure we've yet to discuss is the *linked list*. Because the skip list data structure we'll be examining next is the mutation of a linked list into a data structure with self-balanced binary tree running times, it is important that before diving into the specifics of skip lists we take a moment to discuss linked lists first.

Recall that with a binary tree, each node in the tree contains some bit of data and a reference to its left and right children. A linked list can be thought of as a unary tree. That is, each element in a linked list has some data associated with it, and a *single* reference to its neighbor. As Figure 8 illustrates, each element in a linked list forms a link in the chain. Each link is tied to its neighboring node, the node on its right.

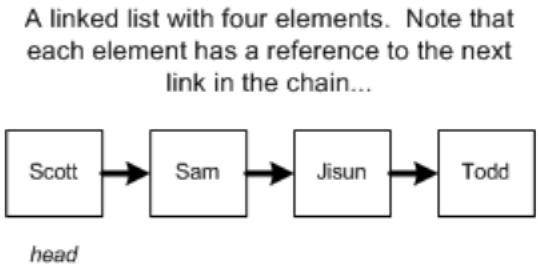


Figure 8. A four-element linked list

When we created a binary tree data structure in Part 3, the binary tree data structure only needed to contain a reference to the root of the tree. The root itself contained references to its children, and those children contained references to their children, and so on. Similarly, with the linked list data structure, when implementing a structure we only need to keep a reference to the *head* of the list because each element in the list maintains a reference to the next item in the list.

Linked lists have the same linear running time for searches as arrays. That is, to find if the element Sam is in the linked list in Figure 8, we have to start at the head and check each element one by one. There are no shortcuts as with binary trees or hashtables. Similarly, deleting from a linked list takes linear time because the linked list must first be searched for the item to be deleted. Once the item is found, removing it from the linked list involves reassigning the deleted item's left neightbor's neighbor reference to the deleted item's neighbor. Figure 9 illustrates the pointer reassignment that must occur when deleting an item from a linked list.

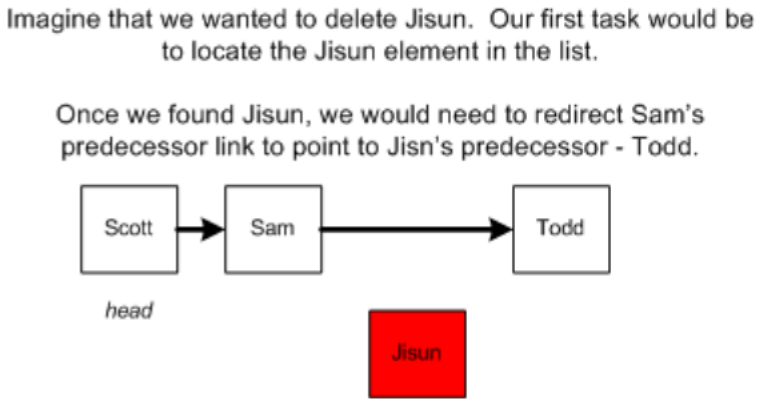


Figure 9. Deleting an element from a linked list

The asymptotic time required to insert a new element into a linked list depends on whether or not the linked list is a sorted list. If the list's elements need not be sorted, insertion can occur in constant time because we can add the element to the front of the list. This involves creating a new element, having its neighbor reference point to the current linked list head, and, finally, reassigning the linked list's head to the newly inserted element.

If the linked list elements need to be maintained in sorted order, then when adding a new element the first step is to locate where in the list it belongs. This is accomplished by exhaustively iterating from the beginning of the list to the element until the spot where the new element belongs. Let *e* be the element immediately before the location where the new element will be added. To insert the new element *e*'s processor reference must now point to the newly inserted element, and the new element's neighbor reference needs to be assigned to *e*'s old neighbor. Figure 10 illustrates this concept graphically.

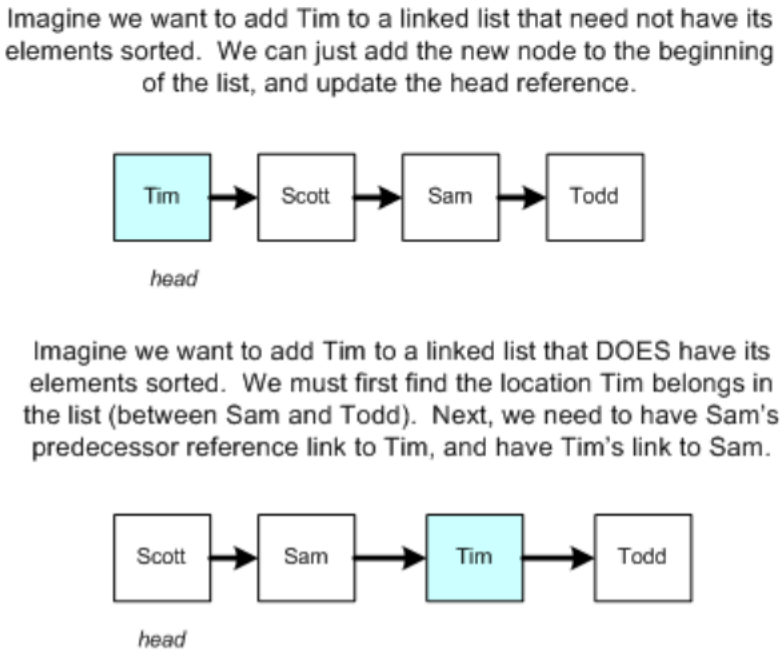


Figure 10. Inserting elements into a sorted linked list

Notice that linked lists do not provide direct access, like an array. That is, if you want to access the i^{th} element of a linked list, you have to start at the front of the list and walk through i links. With an array, though, you can jump straight to the i^{th} element. Given this, along with the fact that linked lists do not offer better search running times than arrays, you might wonder why anyone would want to use a linked list.

The primary benefit of linked lists is that adding or removing items does not involve messy and time-consuming resizing. Recall that array's have fixed size and therefore if an array needs to have more elements added to it than it has capacity, the array must be resized. Granted, the List class hides the code complexity of this, but resizing still carries with it a performance penalty. Furthermore, linked lists are ideal for interactively adding items in sorted order. With an array, inserting an item in the middle of the array requires that all remaining elements be shifted down one spot, and the array resized. In short, an array is usually a better choice if you have an idea on the upper bound of the amount of data that needs to be stored. If you have no conceivable notion as to how many elements will need to be stored, then a link list might be a better choice.

In closing, linked lists are fairly simple to implement. The main challenge comes with the threading or rethreading of the neighbor links with insertions or deletions, but the complexity of adding or removing an element from a linked list pales in comparison to the complexity of balancing an AVL or red-black tree.

With version 2.0 of the .NET Framework, a linked list class has been added to the Base Class Library—`System.Collections.Generic.LinkedList`. This class implements a *doubly-linked list*, which is a linked list whose nodes have a reference to both their next neighbor and their previous neighbor. A `LinkedList` is composed of a variable number of `LinkedListNode` instances, which, in addition to the next and previous references, contains a `Value` property whose type can be specified using Generics.

Skip Lists: A Linked List with Self-Balancing BST-Like Properties

Back in 1989 William Pugh, a computer science professor at the University of Maryland, was looking at sorted linked lists one day thinking about their running time. Clearly a sorted linked list takes linear time to search since potentially each element must be visited, one right after the other. Pugh thought to himself that if half the elements in a sorted linked list had *two* neighbor references—one pointing to its immediate neighbor, and another pointing to the neighbor two elements ahead—while the other half just had one, then searching a sorted linked list could be done in half the time. Figure 11 illustrates a two-reference sorted linked list.

Figure 11. A skip list

The way such a linked list saves searching time is due in part to the fact that the elements are sorted, as well as the varying heights. To search for, say, Dave, we'd start at the *headelement*, which is a dummy element whose height is the same height as the maximum element height in the list. The head element does not contain any data, it merely serves as a place to start searching.

We start at the highest link because it lets us skip over lower elements. We begin by following the head element's top link to Bob. At this point we can ask ourselves, does Bob come before or after Dave? If it comes before Dave, then we know Dave, if he's in the list, must exist somewhere to the right of Bob. If Bob comes before Dave, then Bob must exist somewhere between where we're currently positioned and Bob. In this case, Dave comes after Bob alphabetically, so we can repeat our search again from the Bob element. Notice that by moving onto Bob, we are skipping over Alice. At Bob, we repeat the search at the same level. Following the top-most pointer we reach Dave, bypassing Cal. Because we have found what we are looking for, we can stop searching.

Now, imagine that we wanted to search for Cal. We'd begin by starting at the head element, and then moving onto Bob. At Bob, we'd start by following the top-most reference to Dave. Because Dave comes *after* Cal, we know that Cal must exist somewhere between Bob and Dave. Therefore, we move down to the next lower reference level and continue our comparison.

The efficiency of such a linked list arises because we are able to move two elements over every time instead of just one. This makes the running time on the order of $n/2$, which, while better than a regular sorted link list, is still an asymptotically linear running time. Realizing this, Pugh wondered what would happen if rather than limiting the height of an element to 2, it was instead allowed to go up to $\log_2 n$ for n elements. That is, if there were 8 elements in the linked list, there would be elements with height up to 3; if there were 16 elements, there would be elements with height up to 4. As Figure 12 shows, by intelligently choosing the heights of each of the elements, the search time would be reduced to $\log_2 n$.

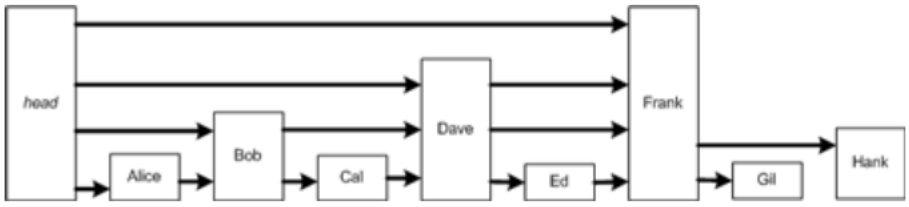


Figure 12. By increasing the height of each node in a skip list, better searching performance can be gained.

Notice that the nodes in Figure 12 every 2^{ith} node has references 2^i elements ahead. That is, the 2^0 element, Alice, has a reference to 2^0 elements ahead—Bob. The 2^{11} element, Bob, has a reference to a node 2^1 elements ahead—Dave. Dave, the 2^2 element, has a reference 2^2 elements ahead—Frank. Had there been more elements, Frank—the 2^3 element—would have a reference to the element 2^3 elements ahead.

The disadvantage of the approach illustrated in Figure 12 is that adding new elements or removing existing ones can wreck havoc on the precise structure. That is, if Dave is deleted, now Ed becomes the 2^2 element, and Gil the 2^3 element, and so on. This means *all* of the elements to the right of the deleted element will need to have their height and references readjusted. The same problem crops up with inserts. This redistribution of heights and references would not only complicate the code for this data structure, but would also reduce the insertion and deletion running times to linear.

Pugh noticed that this pattern created 50% of the elements at height 1, 25% at height 2, 12.5% at height 3, and so on. That is, $1/2^i$ percent of the elements were at height i. Rather than trying to ensure the correct heights for each element with respect to its ordinal index in the list, Pugh decided to just randomly pick a height using the ideal distribution—50% at height 1, 25% at height 2, and so on. What Pugh discovered was that such a randomized linked list was not only very easy to create in code, but that it also exhibited $\log_2 n$ running time for insertions, deletions, and lookups. Pugh named his randomized lists *skip lists* since iterating through the list skips over lower-height elements.

In the remaining sections, we'll examine the insertion, deletion, and lookup functions of the skip list, and implement them in a C# class. We'll finish off with an empirical look at the skip list's performance and discuss the tradeoffs between skip lists and self-balancing BSTs.

Creating the SkipListNode and SkipListNodeList Classes

A skip list, like a binary tree, is made up of a collection of elements. Each element in a skip list has some data associated with it, a height, and a collection of element references. For example, in Figure 12 the Bob element has the data Bob, a height of 2, and two element references: one to Dave and one to Cal. Before creating a skip list class, we first need to create a class that represents an element in the skip list. I created a class that extended the base `Node` class that we examined in Part 3 of this article series, and named this extended class `SkipListNode`. The germane code for `SkipListNode` is shown below. (The complete skip list code is available in this article as a code download.) Note that the `SkipListNode` is implemented using Generics, thereby allowing the developer creating a skip list to specify the type of data the skip list will contain at develop-time.

```
public class SkipListNode<T> : Node<T>
{
    private SkipListNode() {}   // no default constructor available, must supply
height
    public SkipListNode(int height)
    {
        base.Neighbors = new SkipListNodeList<T>(height);
    }

    public SkipListNode(T value, int height) : base(value)
    {
        base.Neighbors = new SkipListNodeList<T>(height);
    }

    public int Height
    {
        get { return base.Neighbors.Count; }
    }

    public SkipListNode<T> this[int index]
    {
        get { return (SkipListNode<T>) base.Neighbors[index]; }
        set { base.Neighbors[index] = value; }
    }
}
```

The `SkipListNode` class uses a `SkipListNodeList` class to store its collection of `SkipListNode` references; these `SkipListNode` references are the `SkipListNode`'s neighbors. (You can see that in the constructor the

base class's `Neighbors` property is assigned to a new `SkipListNodeList<T>` instance with the specified height.) The `SkipListNodeList` class extends the base NodeList class by adding two methods—`IncrementHeight()` and `DecrementHeight()`. As we'll see in the "Inserting Into a SkipList" section, the height of a SkipList might need to be incremented or decremented, depending on the height of the added or removed item.

```
public class SkipListNodeList<T> : NodeList<T>
{
    public SkipListNodeList(int height) : base(height) { }

    internal void IncrementHeight()
    {
        // add a dummy entry
        base.Items.Add(default(Node<T>));
    }

    internal void DecrementHeight()
    {
        // delete the last entry
        base.Items.RemoveAt(base.Items.Count - 1);
    }
}
```

The `SkipListNodeList` constructor accepts a height input parameter that indicates the number of neighbor references that the node will need. It allocates the specified number of elements in the base class's `Neighbors` property by calling the base class's constructor, passing in the height. The `IncrementHeight()` and `DecrementHeight()` methods add a new node to the collection and remove the top-most node, respectively. In a bit we'll see why these two helper methods are needed.

With the `SkipListNode` and `SkipListNodeList` classes created, we're ready to move on to creating the `SkipList` class. The `SkipList` class, as we'll see, contains a single reference to the head element. It also provides methods for searching the list, enumerating through the list's elements, adding elements to the list, and removing elements from the list.

Note For a graphical view of skip lists in action, be sure to check out the skip list applet at http://iamwww.unibe.ch/~wenger/DA/SkipList/. You can add and remove items from a skip list and visually see how the structure and height of the skip list is altered with each operation.

Creating the SkipList Class

The **SkipList** class provides an abstraction of a skip list. It contains public methods like:

- **Add(value):** adds a new item to the skip list.
- **Remove(value):** removes an existing item from the skip list.
- **Contains(value):** returns true if the item exists in the skip list, false otherwise.

And public properties such as:

- **Height:** the height of the tallest element in the skip list.
- **Count:** the total number of elements in the skip list.

The skeletal structure of the class is shown below. Over the next several sections we'll examine the skip list's operations and fill in the code for its methods.

```
public class SkipList<T> : IEnumerable<T>, ICollection<T>
{
   SkipListNode<T> _head;
   int _count;
   Random _rndNum;
   private IComparer<T> comparer = Comparer<T>.Default;

   protected readonly double _prob = 0.5;

   public int Height
   {
      get { return _head.Height; }
   }

   public int Count
   {
      get { return _count; }
   }

   public SkipList() : this(-1, null) {}
   public SkipList(int randomSeed) : this(randomSeed, null) {}
   public SkipList(IComparer<T> comparer) : this(-1, comparer) {}
   public SkipList(int randomSeed, IComparer<T> comparer)
   {
      _head = new SkipListNode<T>(1);
      _count = 0;
      if (randomSeed < 0)
         _rndNum = new Random();
      else
         _rndNum = new Random(randomSeed);

      if (comparer != null) this.comparer = comparer;
   }

   protected virtual int ChooseRandomHeight(int maxLevel)
   {
      ...
   }

   public bool Contains(T value)
   {
      ...
   }

   public void Add(T value)
   {
      ...
   }

   public bool Remove(T value)
   {
      ...
   }
}
```

We'll fill in the code for the methods in a bit, but for now pay close attention to the class's private member variables, public properties, and constructors. There are three relevant private member variables:

- `_head`, which is the list's head element. Remember that a skip list has a dummy head element (refer back to Figures 11 and 12 for a graphical depiction of the head element).
- `_count`, an integer value keeping track of how many elements are in the skip list.
- `_rndNum`, an instance of the `Random` class. Because we need to randomly determine the height when adding a new element to the list, we'll use this `Random` instance to generate the random numbers.

The **SkipList** class has two read-only public properties, `Height` and `Count`. `Height` returns the height of the tallest skip list element. Because the head is always equal to the tallest skip list element, we can simply return the head element's `Height` property. The `Count` property simply returns the current value of the private member variable `count`. (`count`, as we'll see, is incremented in the `Add()` method and decremented in the `Remove()` method.)

Notice there are four forms of the `SkipList` constructor, which provide all permutations for specifying a random seed and a custom comparer. The default constructor creates a skip list using the default comparer for the type `T`, allowing the `Random` class to choose the random seed. Regardless of what constructor form is used, a head for the skip list is created as a new `SkipListNode<T>` instance with height 1, and `count` is set equal to 0.

Note Computer random number generators, such as the `Random` class in the .NET Framework, are referred to as *pseudo-random number generators* because they don't really pick random numbers but instead use a function to generate the random numbers. The random number generating function works by starting with some value, called the *seed*. Based on the seed, a sequence of random numbers are computed. Slight changes in the seed value lead to seemingly random changes in the series of numbers returned.

If you use the `Random` class's default constructor, the system clock is used to generate a seed. You can optionally specify a specific seed value, however. The benefit of specifying a seed is that if you use the same seed value, you'll get the same sequence of random numbers. Being able to get the same results is beneficial when testing the correctness and efficiency of a randomized algorithm like the skip list.

Searching a Skip List

The algorithm for searching a skip list for a particular value is fairly straightforward. Non-formally, the search process can be described as follows: we start with the head element's top-most reference. Let *e* be the element referenced by the head's top-most reference. We check to see if the *e*'s value is less than, greater than, or equal to the value for which we are searching. If it equals the value, then we have found the item we're looking for. If it's greater than the value we're looking for then if the value exists in the list, it must be to the left of *e*, meaning it must have a lesser height than *e*. Therefore, we move down to the second level head node reference and repeat this process.

If, on the other hand, the value of *e* is less than the value we're looking for then the value, if it exists in the list, must be on the right hand side of *e*. Therefore, we repeat these steps for the top-most reference of *e*. This process continues until we either find the value we're searching for, or exhaust all the "levels" without finding the value.

More formally, the algorithm can be spelled out with the following pseudocode:

```
SkipListNode current = head
for i = skipList.Height downto 1
  while current[i].Value < valueSearchingFor
    current = current[i]  // move to the next node

if current[i].Value == valueSearchingFor then
   return true
else
return false
```

Take a moment to trace the algorithm over the skip list shown in Figure 13. The red arrows show the path of checks when searching the skip lists. Skip list (a) shows the results when searching for Ed; skip list (b) shows the results when searching for Cal; skip list (c) shows the results when searching for Gus, which does not exist in the skip list. Notice that throughout the algorithm we are moving in a right, downward direction. The algorithm never moves to a node to the left of the current node, and never moves to a higher reference level.

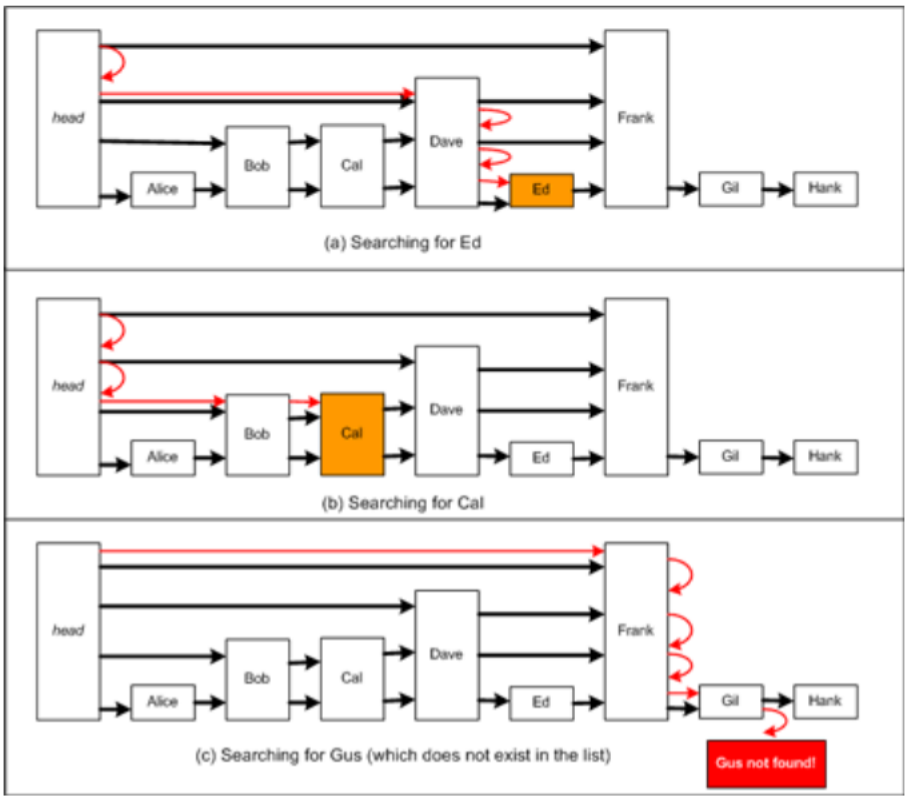


Figure 13. Searching over a skip list.

The code for the `Contains(value)` method is quite simple, involving just a `while` and a `for` loop. The `for` loop iterates down through the reference level layers; the `while` loop iterates across the skip list's elements.

```
public bool Contains(T value)
{
   SkipListNode<T> current = _head;

for (int i = _head.Height - 1; i >= 0; i--)
   {
      while (current[i] != null)
      {
         int results = comparer.Compare(current[i].Value, value);
         if (results == 0)
            return true;   // we found the element
         else if (results < 0)
            current = current[i];   // the element is to the left, so move down a
leve;
         else // results > 0
            break;  // exit while loop, because the element is to the right of this
node, at (or lower than) the current level
      }
   }

   // if we reach here, we searched to the end of the list without finding the
element
   return false;
}
```

Inserting into a Skip List

Inserting a new element into a skip list is akin to adding a new element in a sorted link list, and involves two steps. First, we must locate where in the skip list the new element belongs. This location is found by using the search

algorithm to find the location that comes immediately before the spot the new element will be added. Second, we have to thread the new element into the list by updating the necessary references.

Because skip list elements can has many levels and therefore many references, threading a new element into a skip list is not nearly as simple as threading in a new element into a simple linked list. Figure 14 shows a diagram of a skip list and the threading process that needs to be done to add the element Gus. For this example, imagine that the randomly determined height for the Gus element was 3. To successfully thread in the Gus element, we'd need to update Frank's level 3 and 2 references, as well as Gil's level 1 reference. Gus's level 1 reference would point to Hank. If there were additional nodes to the right of Hank, Gus's level 2 reference would point to the first element to the right of Hank with height 2 or greater, while Gus's level 3 reference would point to the first element right of Hank with height 3 or greater.

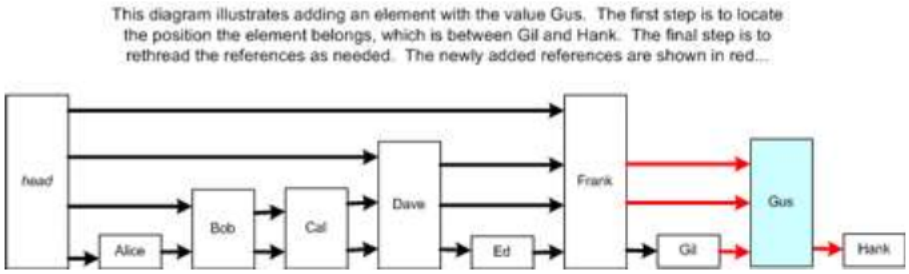


Figure 14. Inserting elements into a skip list

In order to properly rethread the skip list after inserting the new element, we need to keep track of the last element encountered for each height. In Figure 14, Frank was the last element encountered for references at levels 4, 3, and 2, while Gil was the last element encountered for reference level 1. In the insert algorithm below, this record of last elements for each level is maintained by the `updates` array. This array, which is populated as the search for the location for the new element is performed, is created in the `BuildUpdateTable()` method (also shown below).

Note Given a skip list of height *h*, in order to thread in a new node, *h* references will have to be updated in existing skip list nodes. The `updates` array maintains references to the *h* nodes in the skip list whose references will need to be updated. That is, `updates[i]` contains the node whose reference at level *i* will need to be rethreaded to point to the newly inserted node (assuming the newly inserted node's height is greater than or equal to *i*.)

```
public void Add(T value)
{
   SkipListNode<T>[] updates = BuildUpdateTable(value);
   SkipListNode<T> current = updates[0];

   // see if a duplicate is being inserted
   if (current[0] != null && current[0].Value.CompareTo(value) == 0)
      // cannot enter a duplicate, handle this case by either just returning or by
throwing an exception
      return;

   // create a new node
   SkipListNode<T> n = new SkipListNode<T>(value, ChooseRandomHeight(head.Height +
1));
   _count++;   // increment the count of elements in the skip list

   // if the node's level is greater than the head's level, increase the head's level
   if (n.Height > _head.Height)
   {
      _head.IncrementHeight();
      _head[_head.Height - 1] = n;
   }
```

```
   // splice the new node into the list
   for (int i = 0; i < n.Height; i++)
   {
      if (i < updates.Length)
      {
         n[i] = updates[i][i];
         updates[i][i] = n;
      }
   }
}

protected SkipListNode<T>[] BuildUpdateTable(T value)
{
    SkipListNode<T>[] updates = new SkipListNode<T>[_head.Height];
    SkipListNode<T> current = _head;

    // determine the nodes that need to be updated at each level
    for (int i = _head.Height - 1; i >= 0; i--)
    {
        while (current[i] != null && comparer.Compare(current[i].Value, value) < 0)
            current = current[i];

        updates[i] = current;
    }

    return updates;
}
```

There are a couple of key portions of the `Add(value)` and `BuildUpdateTable()` methods to pay close attention to. First, in the `BuildUpdateTable()` method, be certain to examine the `for` loop. In this loop, the `updates` array is fully populated. The `SkipListNode` immediately preceding where the new node will be inserted is represented by `updates[0]`.

After `BuildUpdateTable()`, a check is done to make sure that the data being entered is not a duplicate. I chose to implement my skip list such that duplicates are not allowed; however, skip lists can handle duplicate values just fine. If you want to allow for duplicates, simply remove this check.

Next, a new `SkipListNode` instance, *n*, is created. This represents the element to be added to the skip list. Note that the height of the newly created `SkipListNode` is determined by a call to the `ChooseRandomHeight()` method, passing in the current skip list height plus one. We'll examine this method shortly. Another thing to note is that after adding the `SkipListNode`, a check is made to see if the new `SkipListNode`'s height is greater than that of the skip list's head element's height. If it is, then the head element's height needs to be incremented, since the head element height should have the same height as the tallest element in the skip list.

The final `for` loop rethreads the references. It does this by iterating through the `updates` array, having the newly inserted `SkipListNode`'s references point to the `SkipListNode`s previously pointed to by the `SkipListNode` in the `updates` array, and then having the `updates` array `SkipListNode` update its reference to the newly inserted `SkipListNode`. To help clarify things, try running through the `Add(value)` method code using the skip list in Figure 14, where the added `SkipListNode`'s height happens to be 3.

Randomly Determining the Newly Inserted SkipListNode's Height

When inserting a new element into the skip list, we need to randomly select a height for the newly added `SkipListNode`. Recall from our earlier discussions of skip lists that when Pugh first envisioned multi-level linked list elements, he imagined a linked list where each 2^{ith} element had a reference to an element 2^i elements away. In such a list, precisely 50% of the nodes would have height 1, 25% with height 2, and so on.

The `ChooseRandomHeight()` method uses a simple technique to compute heights so that the distribution of values matches Pugh's initial vision. This distribution can be achieved by flipping a coin and setting the height to one greater than however many heads in a row were achieved. That is, if upon the first flip you get a tails, then the height of the new element will be one. If you get one heads and then a tails, the height will be 2. Two heads followed by a tails indicates a height of three, and so on. Because there is a 50% probability that you will get a tails, a 25% probability that you will get a heads and then a tails, a 12.5% probability that you will get two heads and then a tails, and so on, the distribution works out to be the same as the desired distribution.

The code to compute the random height is given by the following simple code snippet:

```
const double _prob = 0.5;
protected virtual int ChooseRandomHeight()
{
   int level = 1;
   while (_rndNum.NextDouble() < _prob)
      level++;

   return level;
}
```

One concern with the above method is that the value returned might be extraordinarily large. That is, imagine that we have a skip list with, say, two elements, both with height 1. When adding our third element, we randomly choose the height to be 10. This is an unlikely event because there is only roughly a 0.1% chance of selecting such a height, but it could conceivable happen. The downside of this, now, is that our skip list has an element with height 10, meaning there will be a number of superfluous levels in our skip list. To put it more bluntly, the references at levels 2 up to 10 would be unutilized. Even as additional elements were added to the list, there's still only a 3% chance of getting a node over a height of 5, so we'd likely have many wasted levels.

Pugh suggests a couple of solutions to this problem. One is to simply ignore it. Having superfluous levels doesn't require any change in the code of the data structure, nor does it affect the asymptotic running time. Another solution proposed by Pugh calls for using "fixed dice" when choosing the random level, which is the approach I chose to use for the `SkipList` class. With the "fixed dice" approach, you restrict the height of the new element to be a height of at most one greater than the tallest element currently in the skip list. The actual implementation of the `ChooseRandomHeight()` method is shown below, which implements this "fixed dice" approach. Notice that a `maxLevel` input parameter is passed in, and the `while` loop exits prematurely if level reaches this maximum. In the `Add(value)` method, note that the `maxLevel` value passed in is the height of the head element plus one. (Recall that the head element's height is the same as the height of the maximum element in the skip list.)

```
protected virtual int ChooseRandomHeight(int maxLevel)
{
   int level = 1;
   while (_rndNum.NextDouble() < _prob && level < maxLevel)
      level++;

   return level;
}
```

Because the head element should be the same height as the tallest element in the skip list, in the `Add(value)` method, if the newly added `SkipListNode`'s height is greater than the head element's height, I call the `IncrementHeight()` method:

```
/* - snippet from the Add() method... */
if (n.Height > _head.Height)
{
   _head.IncrementHeight();
   _head[_head.Height - 1] = n;
}
/**********************************/
```

The `IncrementHeight()` method simply adds a new `SkipListNode` to the head element. Because the height of the head node must be the height of the tallest element in the skip list, if we add an element taller than the head, we want to increment the head's height, which is what the code in the `if` statement above accomplishes.

Note In his paper, "Skip Lists: A Probabilistic Alternative to Balanced Trees," Pugh examines the effects of changing the value of `_prob` from 0.5 to other values, such as 0.25, 0.125, and others. Lower values of `_prob` decrease the average number of references per element, but increase the likelihood of the search taking substantially longer than expected. For more details, be sure to read Pugh's paper, which is mentioned in the References section at the end of this article.

Deleting an Element from a Skip List

Like adding an element to a skip list, removing an element involves a two-step process. First, the element to be deleted must be found. After that, the element needs to be snipped from the list and the references need to be rethreaded. Figure 15 shows the rethreading that must occur when Dave is removed from the skip list.

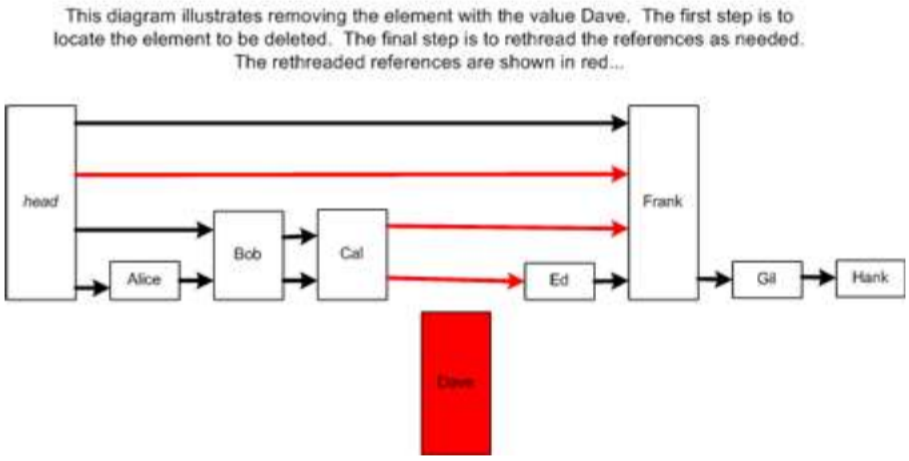


Figure 15. Deleting an element from a skip list

Like with the `Add(value)` method, `Remove(value)` maintains an `updates` array that keeps track of the elements at each level that appear immediately before the element to be deleted (as with `Add()`, this `updates` array is populated with a call to `BuildUpdateTable()`.). Once this updates array has been populated, the array is iterated through from the bottom up, and the elements in the array are rethreaded to point to the deleted element's references at the corresponding levels. The `Remove(value)` method code follows.

```
public virtual void Remove(IComparable value)
{
   SkipListNode<T>[] updates = BuildUpdateTable(value);
   SkipListNode<T> current = updates[0][0];

   if (current != null && comparer.Compare(current.Value, value) == 0)
   {
      _count--;

      // We found the data to delete
      for (int i = 0; i < _head.Height; i++)
      {
         if (updates[i][i] != current)
            break;
         else
            updates[i][i] = current[i];
      }
```

```
      // finally, see if we need to trim the height of the list
      if (_head[_head.Height - 1] == null)
         // we removed the single, tallest item... reduce the list height
         _head.DecrementHeight();

      return true;   // item removed, return true
   }
   else
      // the data to delete wasn't found - return false            return false;
}
```

The `Remove(value)` method starts like the `Add(value)` method, by populating the `updates` array with a call to `BuildUpdateTable()`. With the `updates` array populated, we next check to ensure that the element reached does indeed contain the value to be deleted. If not, the element to be deleted was not found in the skip list, so the `Remove()` method returns `false`. Assuming the element reached is the element to be deleted, the `_count` member variable is decremented and the references are rethreaded. Lastly, if we deleted the element with the greatest height, then we should decrement the height of the head element. This is accomplished via a call to the `DecrementHeight()` method of the `SkipListNode` class.

Analyzing the Skip List's Running Time

In "Skip Lists: A Probabilistic Alternative to Balanced Trees," Pugh provides a quick proof showing that the skip list's search, insertion, and deletion running times are asymptotically bounded by $\log_2 n$ in the average case. However, a skip list can exhibit linear time in the worst case, but the likelihood of the worst case happening is very, very, very, very slim.

Because the heights of the elements of a skip list are randomly chosen, there is a chance that all, or virtually all, elements in the skip list will end up with the same height. For example, imagine that we had a skip list with 100 elements, all that happened to have height 1 chosen for their randomly selected height. Such a skip list would be, essentially, a normal linked list, not unlike the one shown in Figure 8. As we discussed earlier, the running time for operations on a normal linked list is linear.

While such worst-case scenarios are possible, realize that they are highly improbable. To put things in perspective, the likelihood of having a skip list with 100 height 1 elements is the same likelihood of flipping a coin 100 times and having it come up tails all 100 times. The chances of this happening are precisely 1 in 1,267,650,600,228,229,401,496,703,205,376. Of course with more elements, the probability goes down even further. For more information be sure to read about Pugh's probabilistic analysis of skip lists in his paper.

Examining Some Empirical Results

Included in the article's download is the `SkipList` class along with a testing Windows Forms application. With this testing application, you can manually add, remove, and inspect the list, and can see the nodes of the list displayed. Also, this testing application includes a "stress tester," where you can indicate how many operations to perform and an optional random seed value. The stress tester then creates a skip list, adds at least half as many elements as operations requested, and then, with the remaining operations, does a mix of inserts, deletes, and queries. At the end you can see review a log of the operations performed and their result, along with the skip list height, the number of comparisons needed for the operation, and the number of elements in the list.

The graph in Figure 16 shows the average number of comparisons per operation for increasing skip list sizes. Note that as the skip list doubles in size, the average number of comparisons needed per operation only increases by a small amount (one or two more comparisons). To fully understand the utility of logarithmic growth, consider how the time for searching an array would fare on this graph. For a 256 element array, on average 128 comparisons would be needed to find an element. For a 512 element array, on average 256 comparisons would be needed. Compare that to the skip list, which for skip lists with 256 and 512 elements require only 9 and 10 comparisons on average.

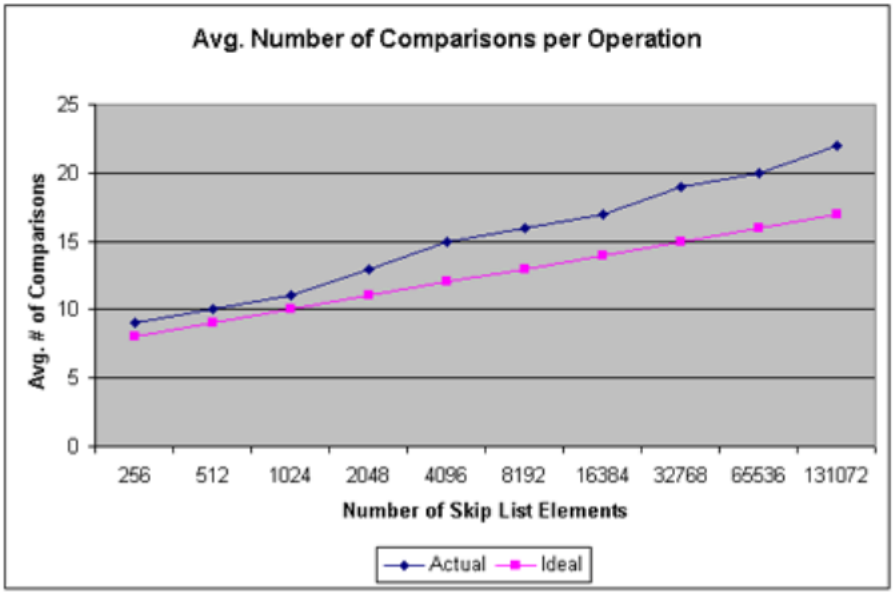


Figure 16. Viewing the logarithmic growth of comparisons required for an increasing number of skip list elements.

Conclusion

In Part 3 of this article series we looked at binary trees and binary search trees. BSTs provide an efficient $\log_2 n$ running time in the average case. However, the running time is sensitive to the topology of the tree, and a tree with an suboptimal ratio of breadth to height can reduce the running time of a BST's operations to linear time.

To remedy this worst-case running time of BSTs, which could happen quite easily since the topology of a BST is directly dependent on the order with which items are added, computer scientists have been inventing a myriad of self-balancing BSTs, starting with the AVL tree created in the 1960s. While data structures such as the AVL tree, the red-black tree, and numerous other specialized BSTs offer $\log_2 n$ running time in both the average and worst case, they require especially complex code that can be difficult to correctly create.

An alternative data structure that offers the same asymptotic running time as a self-balanced BST, is William Pugh's skip list. The skip list is a specialized, sorted link list, one whose elements have a height associated with them. In this article, we constructed a `SkipList` class and saw just how straightforward the skip list's operations were, and how easy it was to implement them in code.

This fourth part of the article series completes our discussion of trees. In the fifth installment, we'll look at *graphs*. A graph is a collection of vertexes with an arbitrary number of edges connecting each vertex to one another. As we'll see in Part 5, trees are a special form of graphs. Graphs have an extraordinary number of applications in real-world problems.

Happy Programming!

References

- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.
- Pugh, William. "Skip Lists: A Probabilistic Alternative to Balanced Trees." Available online at ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf

http://msdn.microsoft.com/en-us/library/ms379574

Visual Studio 2005 Technical Articles

An Extensive Examination of Data Structures Using C# 2.0

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: A graph, like a tree, is a collection of nodes and edges, but has no rules dictating the connection among the nodes. In this **fifth** part of the article series, we'll learn all about graphs, one of the most versatile data structures.(22 printed pages)

Download the DataStructures20.msi sample file.

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

Part 1 and Part 2 of this article series focused on linear data structures—the array, the List, the Queue, the Stack, the Hashtable, and the Dictionary. In Part 3 we began our investigation of trees. Recall that trees consist of a set of *nodes*, where all of the nodes share some connection to other nodes. These connections are referred to as *edges*. As we discussed, there are numerous rules spelling out how these connections can occur. For example, all nodes in a tree except for one—the root—must have precisely one *parent* node, while all nodes can have an arbitrary number of children. These simple rules ensure that, for any tree, the following three statements will hold:

1. Starting from any node, any other node in the tree can be reached. That is, there exists no node that can't be reached through some simple path.
2. There are no *cycles*. A cycle exists when, starting from some node v, there is some path that travels through some set of nodes $v_1, v_2, ..., v_k$ that then arrives back at v.
3. The number of edges in a tree is precisely one less than the number of nodes.

In Part 3 we focused on *binary trees*, which are a special form of trees. Binary trees are trees whose nodes have at most two children.

In this fifth installment of the article series, we're going to examine *graphs*. Graphs are composed of a set of nodes and edges, just like trees, but with graphs there are no rules for the connections between nodes. With graphs there

is no concept of a root node, nor is there a concept of parents and children. Rather, a graph is just a collection of interconnected nodes.

Note Realize that all trees are graphs. A tree is a special case of a graph, one whose nodes are all reachable from some starting node and one that has no cycles.

Figure 1 shows three examples of graphs. Notice that graphs, unlike trees, can have sets of nodes that are disconnected from other sets of nodes. For example, graph (a) has two distinct, unconnected set of nodes. Graphs can also contain cycles. Graph (b) has several cycles. One such is the path from v_1 to v_2 to v_4 and back to v_1. Another one is from v_1 to v_2 to v_3 to v_5 to v_4 and back to v_1. (There are also cycles in graph (a).) Graph (c) does not have any cycles, as one less edge than it does number of nodes, and all nodes are reachable. Therefore, it is a tree.

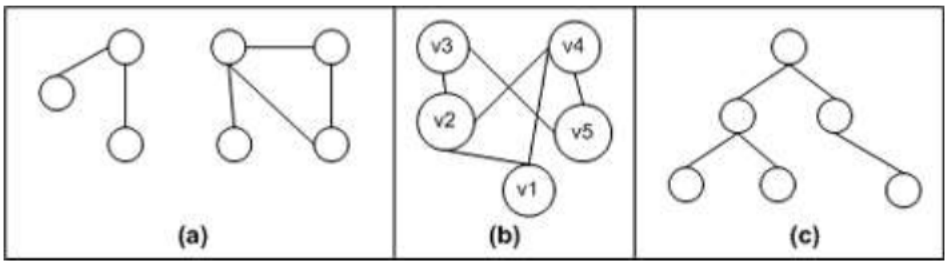


Figure 1. Three examples of graphs

Many real-world problems can be modeled using graphs. For example, search engines model the Internet as a graph, where Web pages are the nodes in the graph and the links among Web pages are the edges. Programs like Microsoft MapPoint that can generate driving directions from one city to another use graphs, modeling cities as nodes in a graph and the roads connecting the cities as edges.

Examining the Different Classes of Edges

Graphs, in their simplest terms, are a collection of nodes and edges, but there are different kinds of edges:

1. Directed versus undirected edges
2. Weighted versus unweighted edges

When talking about using graphs to model a problem, it is usually important to indicate what class of graph you are working with. Is it a graph whose edges are directed and weighted, or one whose edges are undirected and weighted? In the next two sections we'll discuss the differences between directed and undirected edges and weighted and unweighted edges.

Directed and Undirected Edges

The edges of a graph provide the connections between one node and another. By default, an edge is assumed to be bidirectional. That is, if there exists an edge between nodes *v* and *u*, it is assumed that one can travel from *v* to *u* and from *u* to *v*. Graphs with bidirectional edges are said to be *undirected graphs*, because there is no implicit direction in their edges.

For some problems, though, an edge might infer a one-way connection from one node to another. For example, when modeling the Internet as a graph, a hyperlink from Web page *v* linking to Web page *u* would imply that the edge between *v* to *u* would be unidirectional. That is, that one could navigate from *v* to *u*, but not from *u* to *v*. Graphs that use unidirectional edges are said to be *directed graphs*.

When drawing a graph, bidirectional edges are drawn as a straight line, as shown in Figure 1. Unidirectional edges are drawn as an arrow, showing the direction of the edge. Figure 2 shows a directed graph where the nodes are

Web pages for a particular Web site and a directed edge from *u* to *v* indicates that there is a hyperlink from Web page *u* to Web page *v*. Notice that both *u* links to *v* and *v* links to *u*, two arrows are used—one from *v* to *u* and another from *u* to *v*.

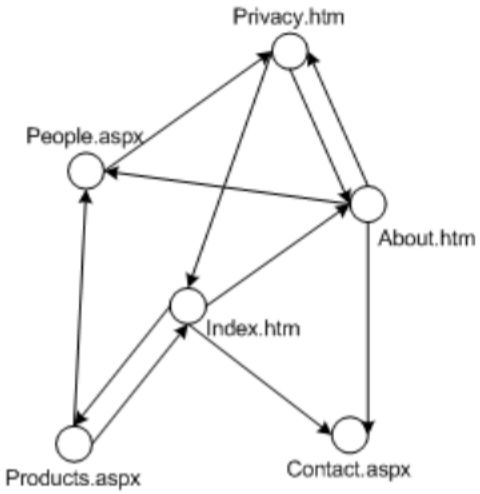


Figure 2. Model of pages making up a website

Weighted and Unweighted Edges

Typically graphs are used to model a collection of "things" and their relationship among these "things." For example, the graph in Figure 2 modeled the pages in a Web site and their hyperlinks. Sometimes, though, it is important to associate some cost with the connection from one node to another.

A map can be easily modeled as a graph, with the cities as nodes and the roads connecting the cities as edges. If we wanted to determine the shortest distance and route from one city to another, we first need to assign a cost from traveling from one city to another. The logical solution would be to give each edge a *weight*, such as how many miles it is from one city to another.

Figure 3 shows a graph that represents several cities in southern California. The cost of any particular path from one city to another is the sum of the costs of the edges along the path. The shortest path, then, would be the path with the least cost. In Figure 3, for example, a trip from San Diego to Santa Barbara is 210 miles if driving through Riverside, then to Barstow, and then back to Santa Barbara. The shortest trip, however, is to drive 100 miles to Los Angeles, and then another 30 up to Santa Barabara.

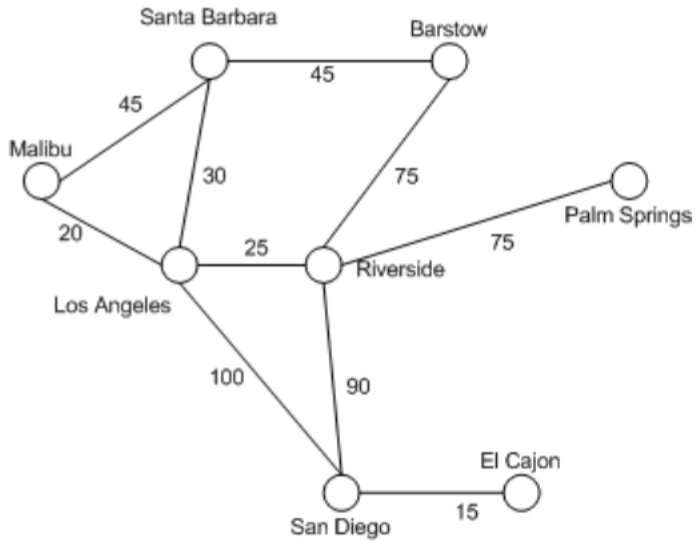


Figure 3. Graph of California cities with edges valued as miles

Realize that directionality and weightedness of edges are orthogonal. That is, a graph can have one of four arrangements of edges:

- Directed, weighted edges
- Directed, unweighted edges
- Undirected, weighted edges
- Undirected, unweighted edges

The graph's in Figure 1 had undirected, unweighted edges. Figure 2 had directed, unweighted edges, and Figure 3 used undirected, weighted edges.

Sparse Graphs and Dense Graphs

While a graph could have zero or a handful of edges, typically a graph will have more edges than it has nodes. What's the maximum number of edges a graph could have, given n nodes? It depends on whether the graph is directed or undirected. If the graph is directed, then each node could have at an edge to every other node. That is, all n nodes could have $n - 1$ edges, giving a total of $n * (n - 1)$ edges, which is nearly n^2.

Note For this article, I am assuming nodes are not allowed to have edges to themselves. In general, though, graphs allow for an edge to exist from a node v back to node v. If self-edges are allowed, the total number of edges for a directed graph would be n^2.

If the graph is undirected, then one node, call it v_1, could have an edge to each and every other node, or $n - 1$ edges. The next node, call it v_2, could have at most $n - 2$ edges, because there already exists an edge from v_2 to v_1. The third node, v_3, could have at most $n - 3$ edges, and so forth. Therefore, for n nodes, there would be at most $(n - 1) + (n - 2) + ... + 1$ edges. Summed up this comes to $[n *(n-1)] / 2$, or, as you might have already guessed, exactly half as many edges as a directed graph.

If a graph has significantly less than n^2 edges, the graph is said to be *sparse*. For example, a graph with n nodes and n edges, or even $2n$ edges would be said to be sparse. A graph with close to the maximum number of edges is said to be *dense*.

When using graphs in an algorithm it is important to know the ratio between nodes and edges. As we'll see later on in this article, the asymptotic running time operations performed on a graph is typically expressed in terms of the number of nodes and edges in the graph.

Creating a Graph Class

While graphs are a very common data structure used in a wide array of different problems, there is no built-in graph data structure in the .NET Framework. Part of the reason is because an efficient implementation of a `Graph` class depends on a number of factors specific to the problem at hand. For example, graphs are typically modeled in either one of two ways:

- As an adjacency list
- As an adjacency matrix

These two techniques differ in how the nodes and edges of the graph are maintained internally by the `Graph` class. Let's examine both of these approaches and weigh the pros and cons of each approach.

Representing a Graph Using an Adjacency List

In Part 3 we created a base class to represent nodes, the `Node` class. This base class was extended to provide specialized node classes for the `BinaryTree`, `BST`, and `SkipList` classes. Because a each node in a graph has an

arbitrary number of neighbors, it might seem plausible that we can simply use the base `Node` class to represent a node in the graph, because the `Node` class consists of a value and an arbitrary number of neighboring `Node` instances. However, while this base class is a step in the right direction, it still lacks needed features, such as a way to associate a cost between neighbors. One option, then, is to create a `GraphNode` class that derives from the base `Node` class and extends it to include the required additional capabilities. Each `GraphNode` class, then, will keep track of its neighboring `GraphNodes` in the base class's `Neighbors` property.

The `Graph` class contains a `NodeList` holding the set of `GraphNodes` that constitute the nodes in the graph. That is, a graph is represented by a set of nodes, and each node maintains a list of its neighbors. Such a representation is called an *adjacency list*, and is depicted graphically in Figure 4.

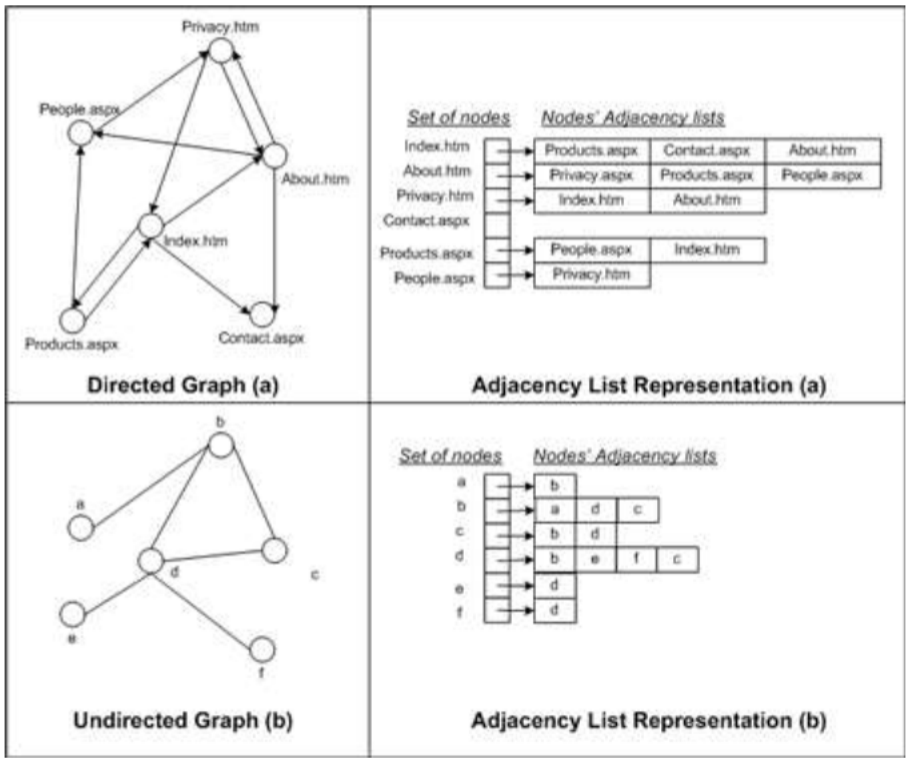


Figure 4. Adjacency list representation in graphical form

Notice that with an undirected graph, an adjacency list representation duplicated the edge information. For example, in adjacency list representation (b) in Figure 4, the node *a* has *b* in its adjacency list, and node *b* also has node *a* in its adjacency list.

Each node has precisely as many `GraphNodes` in its adjacency list as it has neighbors. Therefore, an adjacency list is a very space-efficient representation of a graph—you never store more data than needed. Specifically, for a graph with *V* nodes and *E* edges, a graph using an adjacency list representation will require *V* + *E*`GraphNode` instances for a directed graph and *V* + 2*E*`Node` instances for an undirected graph.

While Figure 4 does not show it, adjacency lists can also be used to represent weighted graphs. The only addition is that for each `GraphNode`*n*'s adjacency list, each `GraphNode` instance in the adjacency list needs to store the cost of the edge from *n*.

The one downside of an adjacency list is that determining if there is an edge from some node *u* to *v* requires that *u*'s adjacency list be searched. For dense graphs, *u* will likely have many `GraphNodes` in its adjacency list. Determining if there is an edge between two nodes, then, takes linear time for dense adjacency list graphs. Fortunately, when using graphs we'll likely not need to determine if there exists an edge between two particular nodes. More often than not, we'll want to simply enumerate *all* the edges of a particular node.

Representing a Graph Using an Adjacency Matrix

An alternative method for representing a graph is to use an *adjacency matrix*. For a graph with *n* nodes, an adjacency matrix is an *n* x *n* two-dimensional array. For weighted graphs the array element (*u*, *v*) would give the cost of the edge between *u* and *v* (or, perhaps -1 if no such edge existed between *u* and *v*. For an unweighted graph, the array could be an array of Booleans, where a True at array element (*u*, *v*) denotes an edge from *u* to *v* and a False denotes a lack of an edge.

Figure 5 depicts how an adjacency matrix representation in graphical form.

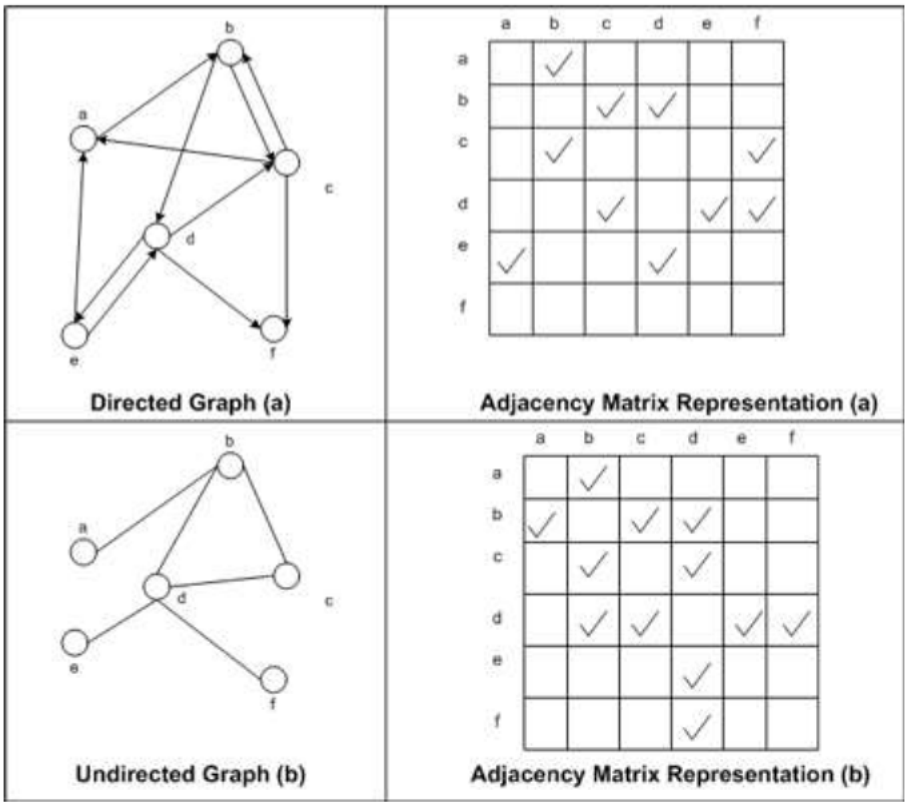


Figure 5. Adjacency matrix representation in graphical form

Note that undirected graphs display symmetry along the adjacency matrix's diagonal. That is, if there is an edge from *u* to *v* in an undirected graph then there will be two corresponding array entries in the adjacency matrix, (*u*, *v*) and (*v*, *u*).

Because determining if an edge exists between two nodes is simply an array lookup, this can be determined in constant time. The downside of adjacency matrices is that they are space inefficient. An adjacency matrix requires an n^2 element array, so for sparse graphs much of the adjacency matrix will be empty. Also, for undirected graphs half of the graph is just repeated information.

While either an adjacency matrix or adjacency list would suffice as an underlying representation of a graph for our `Graph` class, let's move forward using the adjacency list model. I chose this approach primarily because it is a logical extension from the `BinaryTreeNode` and `BinaryTree` classes that we've already created together, and can be implemented by extending the `Node` class used as a base class for the data structures we've examined previously.

Creating the GraphNode Class

The `GraphNode` class represents a single node in the graph, and is derived from the base `Node` class we examined in Part 3 of this article series. The `GraphNode` class extends its base class by providing public access to the

`Neighbors` property, as well as providing a `Cost` property. The `Cost` property is of type `List<int>`; for weighted graphs `Cost[i]` it can be used to specify the cost associated with traveling from the `GraphNode` to `Neighbors[i]`.

```
public class GraphNode<T> : Node<T>
{
    private List<int> costs;

    public GraphNode() : base() { }
    public GraphNode(T value) : base(value) { }
    public GraphNode(T value, NodeList<T> neighbors) : base(value, neighbors) { }

    new public NodeList<T> Neighbors
    {
        get
        {
            if (base.Neighbors == null)
                base.Neighbors = new NodeList<T>();

            return base.Neighbors;
        }
    }

    public List<int> Costs
    {
        get
        {
            if (costs == null)
                costs = new List<int>();

            return costs;
        }
    }
}
```

As the code for the `GraphNode` class shows, the class exposes two properties:

- **Neighbors:** this just provides a public property to the protected base class's `Neighbors` property. Recall that Neighbors is of type `NodeList<T>`.
- **Costs:** a `List<int>` mapping a weight from the GraphNode to a specific neighbor.

Building the Graph Class

Recall that with the adjacency list technique, the graph maintains a list of its nodes. Each node, then, maintains a list of adjacent nodes. So, in creating the `Graph` class we need to have a list of `GraphNodes`. This set of nodes is maintained using a `NodeList` instance. (We examined the `NodeList` class in Part 3; this class was used by the `BinaryTree` and `BST` classes, and was extended for the `SkipList` class.) The `Graph` class exposes its set of nodes through the public property `Nodes`.

Additionally, the `Graph` class has a number of methods for adding nodes and directed or undirected and weighted or unweighted edges between nodes. The `AddNode()` method adds a node to the graph, `while AddDirectedEdge()` and `AddUndirectedEdge()` allow a weighted or unweighted edge to be associated between two nodes.

In addition to its methods for adding edges, the `Graph` class has a `Contains()` method that returns a Boolean indicating if a particular value exists in the graph or not. There is also a `Remove()` method that deletes a `GraphNode` and all edges to and from it. The germane code for the `Graph` class is shown below (some of the overloaded methods for adding edges and nodes have been removed for brevity):

```
public class Graph<T> : IEnumerable<T>
{
    private NodeList<T> nodeSet;
    public Graph() : this(null) {}
    public Graph(NodeList<T> nodeSet)
    {
        if (nodeSet == null)
            this.nodeSet = new NodeList<T>();
        else
            this.nodeSet = nodeSet;
    }

    public void AddNode(GraphNode<T> node)
    {
        // adds a node to the graph
        nodeSet.Add(node);
    }

    public void AddNode(T value)
    {
        // adds a node to the graph
        nodeSet.Add(new GraphNode<T>(value));
    }

    public void AddDirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost)
    {
        from.Neighbors.Add(to);
        from.Costs.Add(cost);
    }

    public void AddUndirectedEdge(GraphNode<T> from, GraphNode<T> to, int cost)
    {
        from.Neighbors.Add(to);
        from.Costs.Add(cost);

        to.Neighbors.Add(from);
        to.Costs.Add(cost);
    }

    public bool Contains(T value)
    {
        return nodeSet.FindByValue(value) != null;
    }

    public bool Remove(T value)
    {
        // first remove the node from the nodeset
        GraphNode<T> nodeToRemove = (GraphNode<T>) nodeSet.FindByValue(value);
        if (nodeToRemove == null)
            // node wasn't found
            return false;

        // otherwise, the node was found
        nodeSet.Remove(nodeToRemove);

        // enumerate through each node in the nodeSet, removing edges to this node
        foreach (GraphNode<T> gnode in nodeSet)
        {
            int index = gnode.Neighbors.IndexOf(nodeToRemove);
            if (index != -1)
            {
                // remove the reference to the node and associated cost
                gnode.Neighbors.RemoveAt(index);
                gnode.Costs.RemoveAt(index);
            }
        }

        return true;
    }
```

```
    public NodeList<T> Nodes
    {
        get  {  return nodeSet;  }
    }

    public int Count
    {
        get { return nodeSet.Count; }
    }
}
```

Using the Graph Class

At this point, we have created all of the classes needed for our graph data structure. We'll soon turn our attention to some of the more common graph algorithms, such as constructing a minimum spanning tree and finding the shortest path from a single node to all other nodes, but before we do let's examine how to use the `Graph` class in a C# application.

Once we create an instance of the `Graph` class, the next task is to add the `Nodes` to the graph. This involves calling the `Graph` class's `AddNode()` method for each node to add to the graph. Let's recreate the graph from Figure 2. We'll need to start by adding six nodes. For each of these nodes let's have the `Key` be the Web page's filename; we'll leave the `Data` as `null`, although this might conceivably contain the contents of the file, or a collection of keywords describing the Web page content.

```
Graph<string> web = new Graph<string>();
web.AddNode("Privacy.htm");
web.AddNode("People.aspx");
web.AddNode("About.htm");
web.AddNode("Index.htm");
web.AddNode("Products.aspx");
web.AddNode("Contact.aspx");
```

Next we need to add the edges. Because this is a directed, unweighted graph, we'll use the `Graph` class's `AddDirectedEdge(u, v)` method to add an edge from *u* to *v*.

```
web.AddDirectedEdge("People.aspx", "Privacy.htm");  // People -> Privacy

web.AddDirectedEdge("Privacy.htm", "Index.htm");    // Privacy -> Index
web.AddDirectedEdge("Privacy.htm", "About.htm");    // Privacy -> About

web.AddDirectedEdge("About.htm", "Privacy.htm");    // About -> Privacy
web.AddDirectedEdge("About.htm", "People.aspx");    // About -> People
web.AddDirectedEdge("About.htm", "Contact.aspx");   // About -> Contact

web.AddDirectedEdge("Index.htm", "About.htm");      // Index -> About
web.AddDirectedEdge("Index.htm", "Contact.aspx");   // Index -> Contacts
web.AddDirectedEdge("Index.htm", "Products.aspx");  // Index -> Products

web.AddDirectedEdge("Products.aspx", "Index.htm");  // Products -> Index
web.AddDirectedEdge("Products.aspx", "People.aspx");// Products -> People
```

After these commands, `web` represents the graph shown in Figure 2. Once we have a constructed a graph we'll typically want to answer some questions. For example, for the graph we just created we might want to answer, "What's the least number of links a user must click to reach any Web page when starting from the homepage (`Index.htm`)?" To answer such questions we can usually fall back on using existing graph algorithms. In the next section we'll examine two common algorithms for weighted graphs: constructing a minimum spanning tree and finding the shortest path from one node to all others.

A Look at Some Common Graph Algorithms

Because graphs are a data structure that can be used to model a bevy of real-world problems, there are innumerous algorithms designed to find solutions for common problems. To further our understanding of graphs, let's take a look at two of the most studied applications of graphs: finding a minimum spanning tree and computing the shortest path from a source node to all other nodes.

The Minimum Spanning Tree Problem

Imagine that you work for the phone company and your task is to provide phone lines to a village with 10 houses, each labeled H1 through H10. Specifically this involves running a single cable that connects each home. That is, the cable must run through houses H1, H2, and so forth, up through H10. Due to geographic obstacles—hills, trees, rivers, and so on—it is not feasible to necessarily run the cable from one house to another.

Figure 6 shows this problem depicted as a graph. Each node is a house, and the edges are the means by which one house can be wired up to another. The weights of the edges dictate the distance between the homes. Your task is to wire up all ten houses using the least amount of telephone wiring possible.

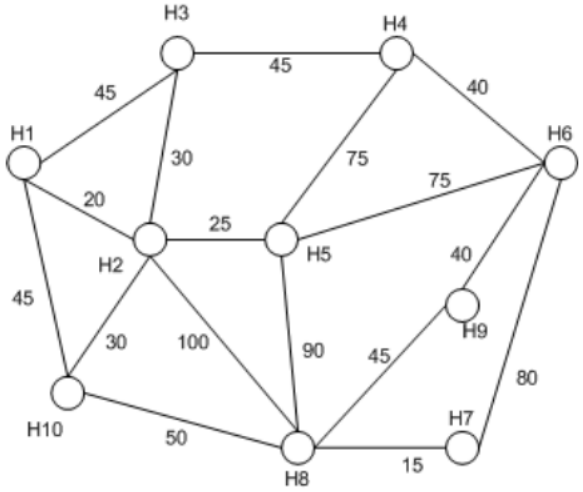


Figure 6. Graphical representation of hooking up a 10-home village with phone lines

For a connected, undirected graph, there exists some subset of the edges that connect all the nodes and does not introduce a cycle. Such a subset of edges would form a tree (because it would comprise one less edge than vertices and is acyclic), and is called a *spanning tree*. There are typically many spanning trees for a given graph. Figure 7 shows two valid spanning trees from the Figure 6 graph. (The edges forming the spanning tree are bolded.)

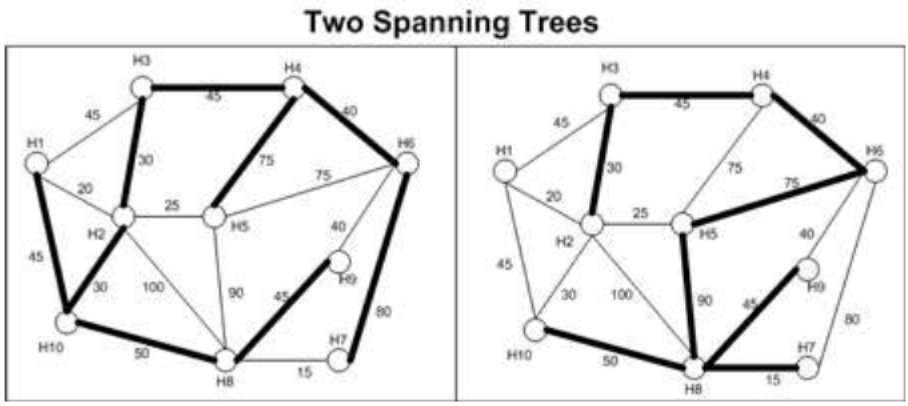


Figure 7.Spanning tree subsets based on Figure 6

For graphs with weighted edges, different spanning trees have different associated costs, where the cost is the sum of the weights of the edges that comprise the spanning tree. A *minimum spanning tree*, then, is the spanning tree with a minimum cost.

There are two basic approaches to solving the minimum spanning tree problem. One approach is build up a spanning tree by choosing the edges with the minimum weight, so long as adding that edge does not create a cycle among the edges chosen thus far. This approach is shown in Figure 8.

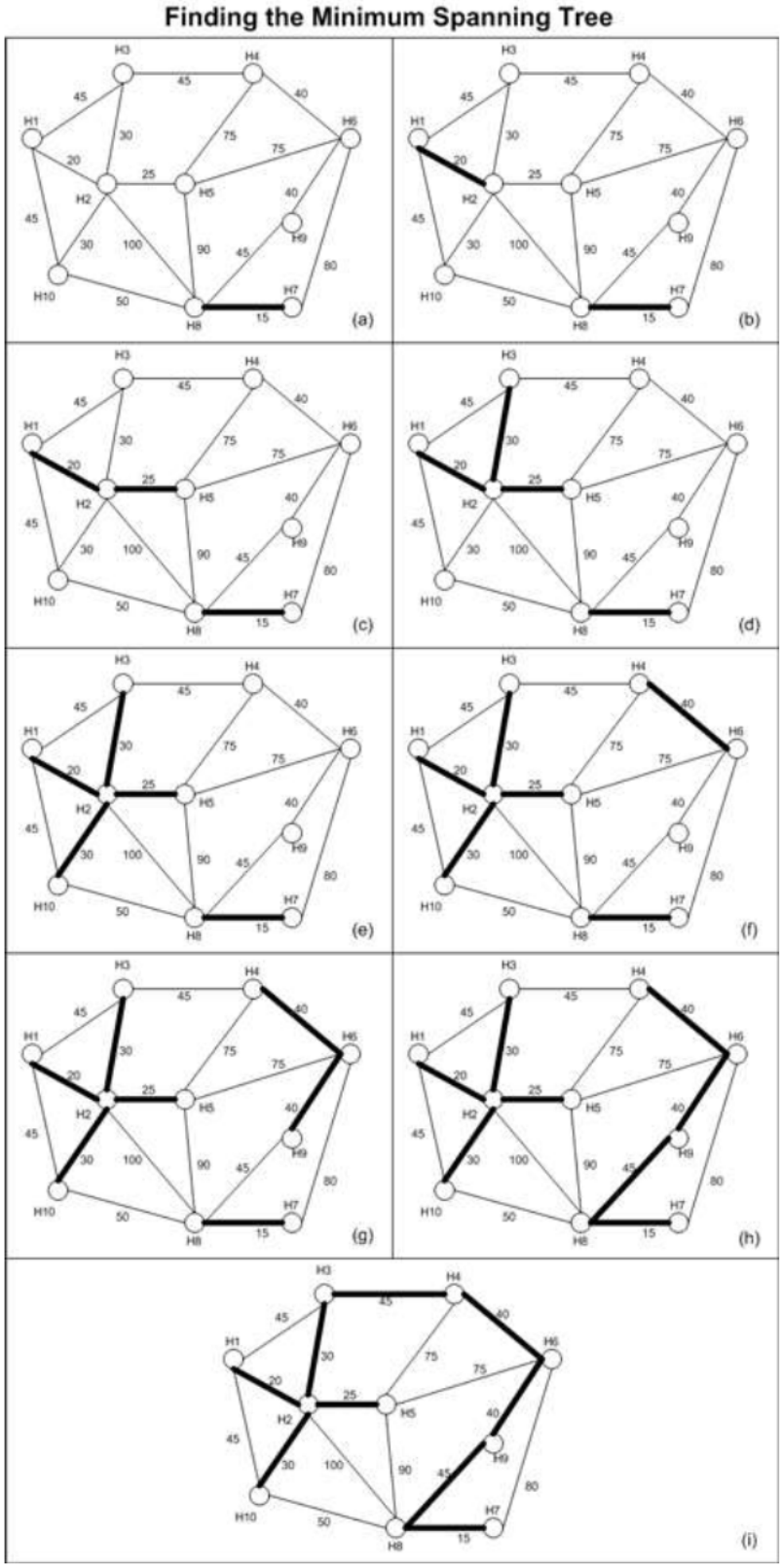


Figure 8. Minimum spanning tree that uses the edges with the minimum weight

The other approach builds up the spanning tree by dividing the nodes of the graph into two disjoint sets: the nodes currently in the spanning tree and those nodes not yet added. At each iteration, the least weighted edge that connects the spanning tree nodes to a node not in the spanning tree is added to the spanning tree. To start off the algorithm, some random start node must be selected. Figure 9 illustrates this approach in action, using H1 as the starting node. (In Figure 9 those nodes that are in the set of nodes in the spanning tree are shaded light yellow.)

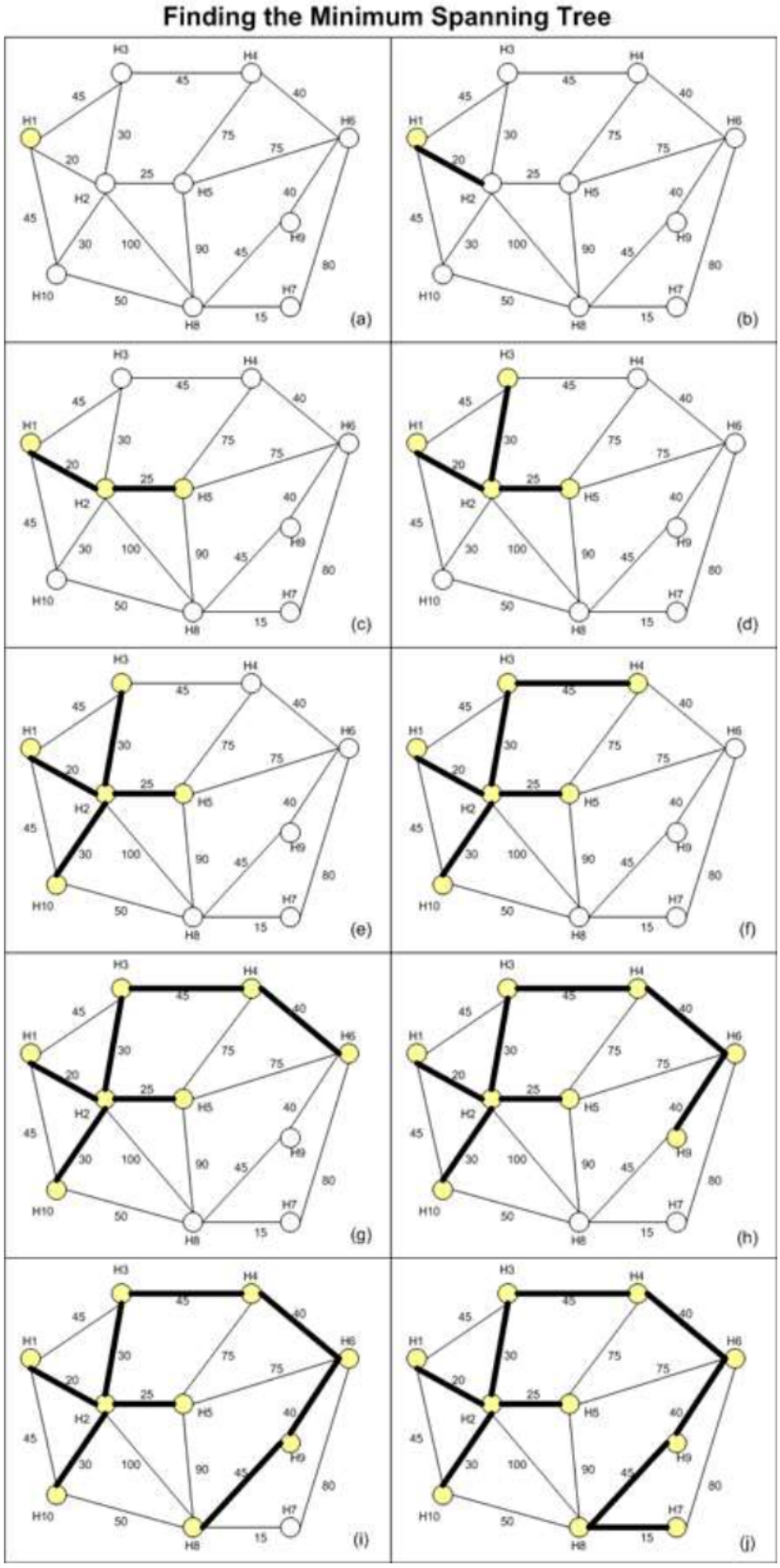


Figure 9. Prim method of finding the minimum spanning tree

Notice that the techniques illustrated in both Figures 8 and 9 arrived at the same minimum spanning tree. If there is only one minimum spanning tree for the graph, then both of these approaches will reach the same conclusion. If, however, there are multiple minimum spanning trees, these two approaches might arrive with different results (both results will be correct, naturally).

Note The first approach we examined was discovered by Joseph Kruskal in 1956 at Bell Labs. The second technique was discovered in 1957 by Robert Prim, also a researcher at Bell Labs. There is a plethora of information on these two algorithms on the Web, including Java applets showing the algorithms in progress graphically (Kruskal's Algorithm | Prim's Algorithm), as well as source code in a variety of languages.

Computing the Shortest Path from a Single Source

When flying from one city to another, part of the headache is finding a route that requires the fewest number of connections—who likes their flight from New York to L.A. to first go from New York to Chicago, then Chicago to Denver, and finally Denver to L.A? Rather, most people would rather have a direct flight straight from New York to L.A.

Imagine, however, that you are not one of those people. Instead, you are someone who values his money much more than his time, and are most interested in finding the *cheapest* route, regardless of the number of connections. This might mean flying from New York to Miami, then Miami to Dallas, then Dallas to Phoenix, Phoenix to San Diego, and finally San Diego to L.A.

We can solve this problem by modeling the available flights and their costs as a directed, weighted graph. Figure 10 shows such a graph.

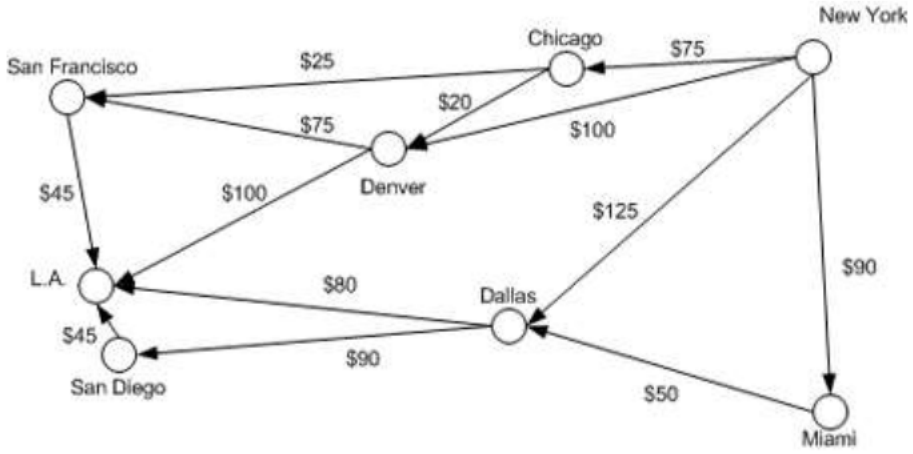


Figure 10. Modeling of available flights based on cost

What we are interested in knowing is what is the least expensive path from New York to L.A. By inspecting the graph, we can quickly determine that it's from New York to Chicago to San Francisco and finally down to L.A., but in order to have a computer accomplish this task we need to formulate an algorithm to solve the problem at hand.

The late Edsger Dijkstra, one of the most noted computer scientists of all time, invented the most commonly used algorithm for finding the shortest path from a source node to all other nodes in a weighted, directed graph. This algorithm, dubbed Dijkstra's Algorithm, works by maintaining two tables, each of which have a record for each node. These two tables are:

- A distance table, which keeps an up-to-date "best distance" from the source node to every other node.
- A route table, which, for each node *n*, indicates what node was used to reach *n* to get the best distance.

Initially, the distance table has each record set to some high value (like positive infinity) except for the start node, which has a distance to itself of 0. The route table's rows are all set to `null`. Also, a collection of nodes, *Q*, that need to be examined is maintained; initially, this collection contains all of the nodes in the graph.

The algorithm proceeds by selecting (and removing) the node from *Q* that has the lowest value in the distance table. Let this selected node be called *n* and the value in the distance table for *n* be *d*. For each of the *n*'s edges, a check is made to see if *d* plus the cost to get from *n* to that particular neighbor is less than the value for that neighbor in the distance table. If it is, then we've found a better way to reach that neighbor, and the distance and route tables are updated accordingly.

To help clarify this algorithm, let's begin applying it to the graph from Figure 10. Because we want to know the cheapest route from New York to L.A. we use New York as our source node. Our initial distance table, then, contains a value of infinity for each of the other cities, and a value of 0 for New York. The route table contains `null`s for all entries, and *Q* contains all nodes (see Figure 11).

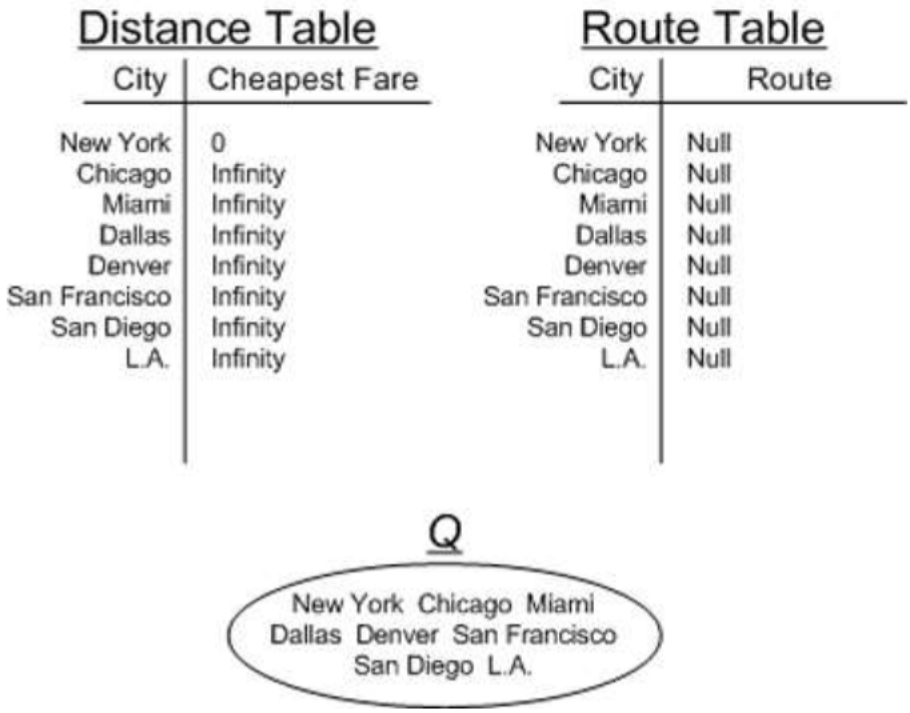


Figure 11. Distance table and route table for determining cheapest fare

We start by extracting the city from *Q* that has the lowest value in the distance table—New York. We then examine each of New York's neighbors and check to see if the cost to fly from New York to that neighbor is less than the best cost we know of, namely the cost in the distance table. After this first check, we'd have removed New York from *Q* and updated the distance and route tables for Chicago, Denver, Miami, and Dallas.

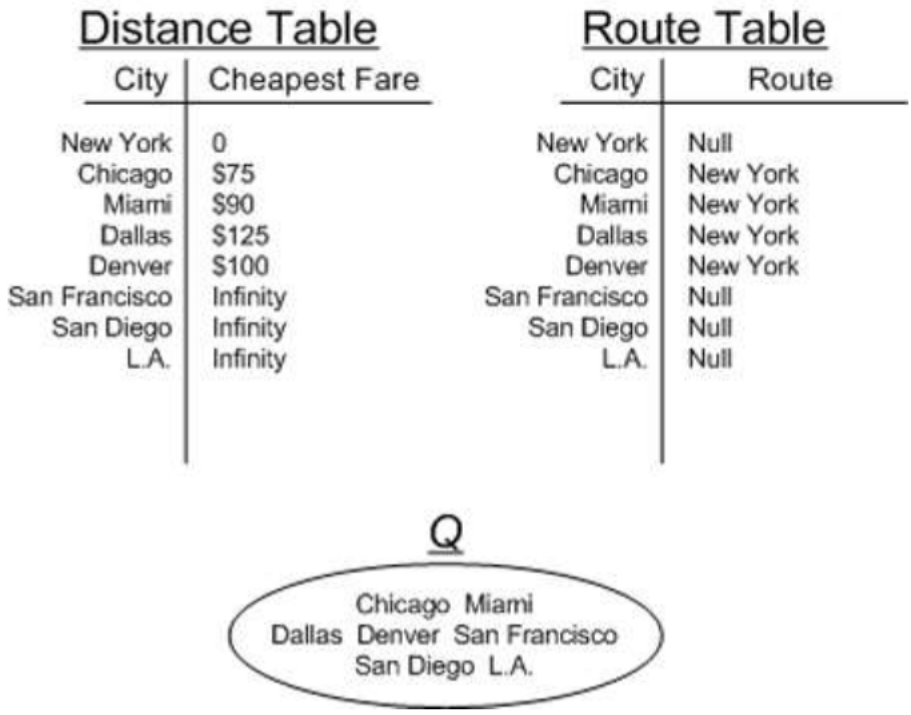


Figure 12. Step 2 in the process of determining the cheapest fare

The next iteration gets the cheapest city out of *Q*, Chicago, and then checks its neighbors to see if there is a better cost. Specifically, we'll check to see if there's a better route for getting to San Francisco or Denver. Clearly the cost to get to San Francisco from Chicago—$75 + $25—is less than Infinity, so San Francisco's records are updated. Also, note that it is cheaper to fly from Chicago to Denver than from New York to Denver ($75 + $20 < $100), so Denver is updated as well. Figure 13 shows the values of the tables and *Q* after Chicago has been processed.

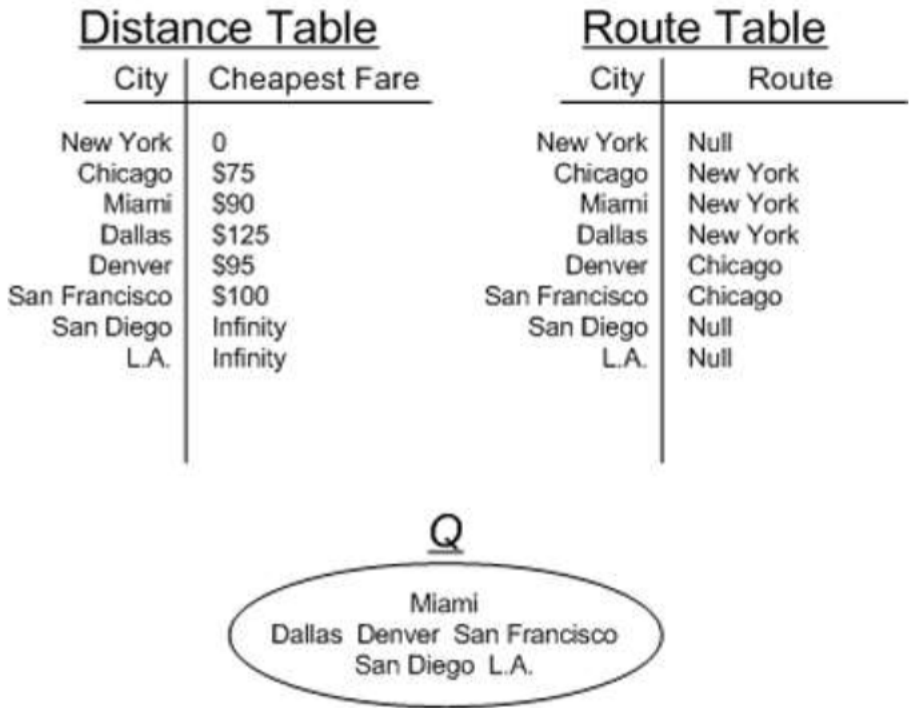
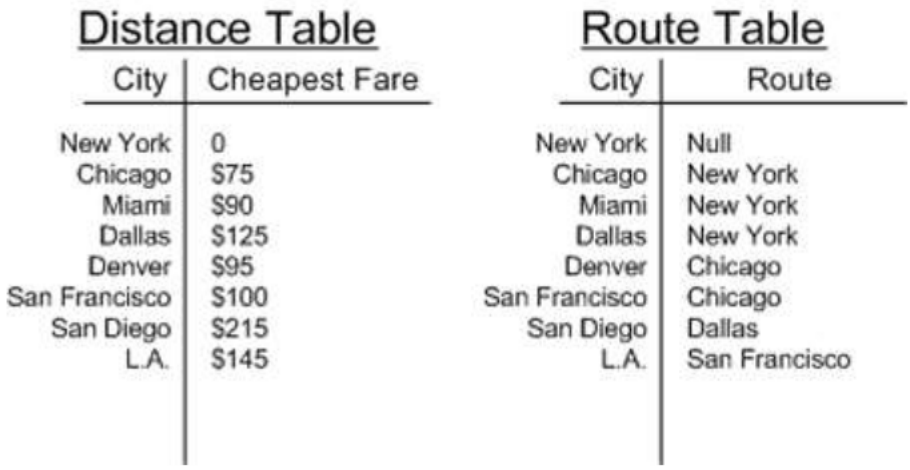


Figure 13. Table status after the third leg of the process is finished

This process continues until there are no more nodes in *Q*. Figure 14 shows the final values of the tables when *Q* has been exhausted.



Distance Table

City	Cheapest Fare
New York	0
Chicago	$75
Miami	$90
Dallas	$125
Denver	$95
San Francisco	$100
San Diego	$215
L.A.	$145

Route Table

City	Route
New York	Null
Chicago	New York
Miami	New York
Dallas	New York
Denver	Chicago
San Francisco	Chicago
San Diego	Dallas
L.A.	San Francisco

Figure 14. Final results of determining the cheapest fare

At the point of exhausting *Q*, the distance table will contain the lowest cost from New York to each city. To determine the flight path to arrive at L.A., start by examining the L.A. entry in the route table and work back up to New York. That is, the route table entry for L.A. is San Francisco, meaning the last leg of the flight to L.A. leaves from San Francisco. The route table entry for San Francisco is Chicago, meaning you'll get to San Francisco via Chicago. Finally, Chicago's route table entry is New York. Putting this together we see that the cheapest flight path is from New York to Chicago to San Francisco to L.A, and costs $145.

Note To see a working implementation of Dijkstra's Algorithm check out the download for this article, which includes a testing application for the `Graph` class that determines the shortest distance from one city to another using Dijkstra's Algorithm.

Conclusion

Graphs are a commonly used data structure because they can be used to model many real-world problems. A graph consists of a set of nodes with an arbitrary number of connections, or edges, between the nodes. These edges can be either directed or undirected and weighted or unweighted.

In this article we examined the basics of graphs and created a `Graph` class. This class was similar to the `BinaryTree` class created in Part 3, the difference being that instead of only have a reference for at most two edges, the `Graph` class's `GraphNodes` could have an arbitrary number of references. This similarity is not surprising because trees are a special case of graphs.

In addition to creating a `Graph` class, we also looked at two common graph algorithms, the minimum spanning tree problem and computing the shortest path from some source node to all other nodes in a weighted, directed graph. While we did not examine source code to implement these algorithms, there are plenty source code examples available on the Internet. Too, the download included with this article contains a testing application for the `Graph` class that uses Dijkstra's Algorithm to compute the shortest route between two cities.

In the next installment, Part 6, we'll look at efficiently maintaining disjoint sets. Disjoint sets are a collection of two or more sets that do not share any elements in common. For example, with Prim's Algorithm for finding the minimum spanning tree, the nodes of the graph can be divided into two disjoint sets: the set of nodes that currently constitute the spanning tree and the set of nodes that are not yet in the spanning tree.

References

- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.

http://msdn.microsoft.com/en-us/library/ms379575

Visual Studio 2005 Technical Articles

An Extensive Examination of Data Structures Using C# 2.0

Scott Mitchell
4GuysFromRolla.com

Update January 2005

Summary: This **sixth installment** of the article series examines how to implement a common mathematical construct, the **set**. A set is an unordered collection of unique items that can be enumerated and compared to other sets in a variety of ways. In this article we'll look at data structures for implementing general sets as well as disjoint sets. (20 printed pages)

Download the DataStructures20.msi sample file.

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

One of the most basic mathematical constructs is a *set*, which is an unordered collection of unique objects. The objects contained within a set are referred to as the set's *elements*. Formally, a set is denoted as a capital, italic letter, with its elements appearing within curly braces ({ . . . }). Examples of this notation can be seen below:

```
S = { 1, 3, 5, 7, 9 }
T = { Scott, Jisun, Sam }
U = { -4, 3.14159, Todd, x }
```

In mathematics, typically sets are comprised strictly of numbers, such as set *S* above, which contains the odd positive integers less than 10. But notice that the elements of a set can be anything—numbers, people, strings, letters, variables, and so on. Set *T*, for example, contains peoples' names; set *U* contains a mix of numbers, names, and variables.

In this article we'll start with a basic introduction of sets, including common notation and the operations that can be performed on sets. Following that, we'll examine how to efficiently implement a set data structure with a defined universe. The article concludes with an examination of disjoint sets, and the best data structures to use.

The Fundamentals of Sets

Recall that a set is simple a collection of elements. The "element of" operator, denoted $x \in S$, implies that x is an element in the set S. For example, if set S contains the odd positive integers less than 10, then $1 \in S$. When reading such notation, you'd say, "1 is an element of S." In addition to 1 being an element of S, we have $3 \in S$, $5 \in S$, $7 \in S$, and $9 \in S$. The "not and element of" operator, denoted $x \notin S$, means that x is not an element of set S.

The number of unique elements in a set is the set's *cardinality*. The set {1, 2, 3} has cardinality 3, just as does the set {1, 1, 1, 1, 1, 1, 1, 2, 3} (because it only has three unique elements). A set may have no elements in it at all. Such a set is called the *empty set*, and is denoted as {} or ∅, and has a cardinality of 0.

When first learning about sets, many developers assume they are tantamount to collections, like a List. However, there are some subtle differences. A List is an *ordered* collection of elements. Each element in a List has an associated ordinal index, which implies order. Too, there can be duplicate elements in a List.

A set, on the other hand, is *unordered* and contains *unique* items. Because sets are unordered, the elements of a set may be listed in any order. That is, the sets {1, 2, 3} and {3, 1, 2} are considered equivalent. Also, any duplicates in a set are considered redundant. The set {1, 1, 1, 2, 3} and the set {1, 2, 3} are equivalent. Two sets are equivalent if they have the same elements. (Equivalence is denoted with the = sign; if S and T are equivalent they are written as $S = T$.)

Note In mathematics, an *ordered* collection of elements that allows duplicates is referred to as a *list*. Two lists, L_1 and L_2 are considered equal if and only if for i ranging from 1 to the number of elements in the list the ith element in L_1 equals the ith element in L_2.

Typically the elements that can appear in a set are restricted to some *universe*. The universe is the set of all possible values that can appear in a set. For example, we might only be interested in working with sets whose universe are integers. By restricting the universe to integers, we can't have a set that has a non-integer element, like 8.125, or Sam. (The universe is denoted as the set U.)

Relational Operators of Sets

There are a bevy of relational operators that are commonly used with numbers. Some of the more often used ones, especially in programming languages, include <, <=, =, !=, >, and >=. A relational operator determines if the operand on the left hand side is related to the operand on the right hand side based on criteria defined by the relational operator. Relational operators return a "true" or "false" value, indicating whether or not the relationship holds between the operands. For example, $x < y$ returns true if x is less than y, and false otherwise. (Of course the meaning of "less than" depends on the data type of x and y.)

Relational operators like <, <=, =, !=, >, and >= are typically used with numbers. Sets, as we've seen, use the = relational operator to indicate that two sets are equivalent (and can likewise use != to denote that two sets are not equivalent), but relational operators <, <=, >, and >= are not defined for sets. After all, how is one to determine if the set {1, 2, 3} is less than the set {Scott, 3.14159}?

Instead of notions of < and <=, sets use the relational operators *subset* and *proper subset*, denoted $\subseteq$ and $\subset$, respectively. (Some older texts will use $\subset$ for subset and $\subseteq$ for proper subset.) S is a subset of T—denoted $S \subseteq T$—if every element in S is in T. That is, S is a subset of T if it is *contained* within T. If $S = \{1, 2, 3\}$, and $T = \{0, 1, 2, 3, 4, 5\}$, then $S \subseteq T$ because every element in S – 1, 2 and 3—is an element in T. S is a proper subset of T—denoted $S \subset T$—if $S \subseteq T$ and $S \neq T$. That is, if $S = \{1, 2, 3\}$ and $T = \{1, 2, 3\}$, then $S \subseteq T$ because every element in S is an element in T, but $S \not\subset T$ because $S = T$. (Notice that there is a similarity between the relational operators < and <= for numbers and the relational operators $\subset$ and $\subseteq$ for sets.)

Using the new subset operator, we can more formally define set equality. Given sets S and T, $S = T$ if and only if $S \subseteq T$ and $T \subseteq S$. In English, S and T are equivalent if and only if every element in S is in T, and every element in T is in S.

Note Because $\subseteq$ is analogous to <=, it would make sense that there exists a set relational operator analogous to >=. This relational operator is called *superset*, and is denoted $\supseteq$; a *proper superset* is denoted $\supset$. Like with <= and >=, $S \supseteq T$ if and only if $T \subseteq S$.

Set Operations

As with the relational operators, many operations defined for numbers don't translate well to sets. Common operations on numbers include addition, multiplication, subtraction, exponentiation, and so on. For sets, there are four basic operations:

1. **Union:** the union of two sets, denoted $S \cup T$, is akin to addition for numbers. The union operator returns a set that contains all of the elements in S and all of the elements in T. For example, {1, 2, 3} $\cup$ {2, 4, 6} equals {1, 2, 3, 2, 4, 6}. (The duplicate 2 can be removed to provide a more concise answer, yielding {1, 2, 3, 4, 6}.) Formally, $S \cup T = \{x : x \in S \text{ or } x \in T\}$. In English, this translates to S union T results in the set that contains an element x if x is in S or in T.
2. **Intersection:**the intersection of two sets, denoted $S \cap T$, is the set of elements that S and T have in common. For example, {1, 2, 3} $\cap$ {2, 4, 6} equals {2}, because that's the only element both {1, 2, 3} and {2, 4, 6} share in common. Formally, $S \cap T = \{x : x \in S \text{ and } x \in T\}$. In English, this translates to S intersect T results in the set that contains an element x if x is both in S**and** in T.
3. **Difference:** the difference of two sets, denoted $S - T$, are all of the elements in S that are **not** in T. For example, {1, 2, 3} – {2, 4, 6} equals {1, 3}, because 1 and 3 are the elements in S that are not in T. Formally, $S - T = \{x : x \in S \text{ and } x \notin T\}$. In English, S set difference T results in the set that contains an element x if x is in S and **not** in T.
4. **Complement:** Earlier we discussed how typically sets are limited to a known universe of possible values, such as the integers. The complement of a set, denoted S', is $U - S$. (Recall that U is the universe set.) If our universe is the integers 1 through 10, and S = {1, 4, 9, 10}, then S' = {2, 3, 5, 6, 7, 8}. Complementing a set is akin to negating a number. Just like negating a number twice will give you the original number back. That is, $--x = x$ – complementing a set twice will give you the original set back – $S'' = S$.

When examining new operations, it is always important to get a solid grasp on the nature of the operations. Some questions to ask yourself when learning about any operation, be it one defined for numbers or one defined for sets, are:

- **Is the operation commutative?** An operator *op* is commutative if *xopy* is equivalent to *yop x*. In the realm of numbers, addition is an example of a commutative operator, while division is not commutative.
- **Is the operation associative?** That is, does the order of operations matter. If an operator *op* is associative, then *x op* (*y op z*) is equivalent to (*x op y*) *op z*. Again, in the realm of numbers addition is associative, but division is not.

For sets, the union and intersection operations are both commutative and associative. $S \cup T$ is equivalent to $T \cup S$, and $S \cup (T \cup V)$ is equivalent to $(S \cup T) \cup V$. Set difference, however, is neither commutative nor associative. (To see that set difference is not commutative, consider that {1, 2, 3} – {3, 4, 5} = {1, 2}, but {3, 4, 5} – {1, 2, 3} = {4, 5}.)

Finite Sets and Infinite Sets

All of the set examples we've looked at thus far have dealt with finite sets. A finite set is a set that has a finite number of elements. While it may seem counterintuitive at first, a set can contain an infinite number of elements. The set of positive integers, for example, is an infinite set because there is no bounds to the number of elements in the set.

In mathematics, there are a couple infinite sets that are used so often that they are given a special symbol to represent them. These include:

$\mathbb{N} = \{0, 1, 2, ...\}$

$\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$

$\mathbb{Q} = \{a/b: a \in \mathbb{Z}, b \in \mathbb{Z}, \text{and } b \neq 0\}$

$\mathbb{R} =$ *set of real numbers*

$\mathbb{N}$ is the set of *natural numbers*, or positive integers greater than or equal to 0. $\mathbb{Z}$ is the set of *integers*. $\mathbb{N}$ is the set of *rational numbers*, which are numbers that can be expressed as a fraction of two integers. Finally, $\mathbb{R}$ is the set of *real numbers*, which are all rational numbers, plus irrational numbers as well (numbers that cannot be expressed as a fraction of two integers, such as *pi*, and the square root of 2).

Infinite sets, of course, can't be written down in their entirety, as you'd never finish jotting down the elements, but instead are expressed more tersely using mathematical notation like so:

$$S = \{x : x \in \mathbb{N} \text{ and } x > 100\}$$

Here *S* would be the set of all natural numbers greater than 100.

In this article we will be looking at data structures for representing finite sets. While infinite sets definitely have their place in mathematics, rarely will we need to work with infinite sets in a computer program. Too, there are unique challenges with representing and operating upon infinite sets, because an infinite set's contents cannot be completely stored in a data structure or enumerated.

Note Computing the cardinality of finite sets is simple. Just count up the number of elements in the set. But how does one compute the cardinality of an infinite set? This discussion is far beyond the scope of this article, but realize that there's different types of cardinality for infinite sets. For instance, the set of positive integers has the same cardinality as the set of **all** integers, but the set of real numbers has a larger cardinality than the set of all integers.

Sets in Programming Languages

C++, C#, Visual Basic .NET, and Java don't provide inherent language features for working with sets. If you want to use sets, you need to create your own set class with the appropriate methods, properties, and logic. (We'll do precisely this in the next section.) There have been programming languages in the past, though, that have offered sets as a fundamental building block in the language. Pascal, for example, provides a set construct that can be used to create sets with an explicitly defined universe. To work with sets, Pascal provides the `in` operator to determine if an element is in a particular set. The operators +, *, and – are used for union, intersection, and set difference, respectively. The following Pascal code illustrates the syntax used to work with sets:

```
/* declares a variable named possibleNumbers, a set whose universe is the
  set of integers between 1 and 100... */
var
  possibleNumbers = set of 1..100;

...

/* Assigns the set {1, 45, 23, 87, 14} to possibleNumbers */
possibleNumbers := [1, 45, 23, 87, 14];
```

```
/* Sets possibleNumbers to the union of possibleNumbers and {3, 98} */
possibleNumbers := possibleNumbers + [3, 98];

/* Checks to see if 4 is an element of possible numbers... */
if 4 in possibleNumbers then write("4 is in the set!");
```

Other previous languages have allowed for more powerful set semantics and syntax. A language called SETL (an acronym for SET Language) was created in the 70s and offered sets as a first-class citizen. Unlike Pascal, when using sets in SETL you are not restricted to specifying the set's universe.

Implementing an Efficient Set Data Structure

In this section we'll look at creating a class that provides the functionality and features of a set. When creating such a data structure, one of the first things we need to decide is how to store the elements of the set. This decision can greatly affect the asymptotic efficiency of the operations performed on the set data structure. (Keep in mind that the operations we'll need to perform on the set data structure include: union, intersection, set difference, subset, and element of.)

To illustrate how storing the set's elements can affect the run time, imagine that we created a set class that used an underlying ArrayList to hold the elements of the set. If we had two sets, S_1 and S_2 that we wanted to union (where S_1 had m elements and S_2 had n elements), we'd have to perform the following steps:

1. Create a new set class, T, that holds the union of S_1 and S_2.
2. Iterate through the elements of S_1, adding it to T.
3. Iterate through the elements of S_2. If the element does not already exist in T, then add it to T.

How many steps would performing the union take? Step (2) would require m steps through S_1's m elements. Step (3) would take n steps, and for each element in S_2, we'd have to determine if the element was in T. To determine if an element is in an unsorted List the entire List must be enumerated linerarly. So, for each of the n elements in S_2 we might have to search through the m elements in T. This would lead to a quadratic running time for union of $O(m * n)$.

The reason a union with a List takes quadratic time is because determining if an element exists within a set takes linear time. That is, to determine if an element exists in a set, the set's List must be exhaustively searched. If we could reduce the running time for the "element of" operation to a constant, we could improve the union's running time to a linear $O(m + n)$. Recall from Part 2 of this article series that Hashtables and Dictionaries provides constant running time to determine if an item resides within the data structure. Hence, a Hashtable or Dictionary would be a better choice for storing the set's elements than a List.

If we require that the set's universe be known, we can implement an even more efficient set data structure using a bit array. Assume that the universe consists of elements e_1, e_2, ..., e_k. Then we can denote a set with a k-element bit array; if the ith bit is 1, then the element e_i is in the set; if, on the other hand, the ith bit is 0, then the element e_i is not in the set. Representing sets as a bit array not only provides tremendous space savings, but also enables efficient set operations, as these set-based operations can be performed using simple bit-wise instructions. For example, determining if element e_i exists in a set takes constant time because only the ith bit in the bit array needs to be checked. The union of two sets is simply the bit-wise OR of the sets' bit arrays; the intersection of two sets is the bit-wise AND of the sets' bit arrays. Set difference and subset can be reduced down to bit-wise operations as well.

Note A bit array is a compact array composed of 1s and 0s, typically implemented as an integer array. Because an integer in the .NET Framework has 32 bits, a bit array can store 32 bit values in one element of an integer array (rather than requiring 32 array elements).

Bit-wise operations are ones that are performed on the individual bits of an integer. There are both binary bit-wise operators and unary bit-wise operators. The bit-wise AND and bit-wise OR operators are binary, taking in two bits each, and returning a single bit. Bit-wise AND returns 1 only if both inputs are 1, otherwise it returns 0. Bit-wise OR returns 0 only if both inputs are 0, otherwise it returns 1.

For a more in-depth look at bit-wise operations in C# be sure to read Bit-Wise Operators in C#.

Let's look at how to implement a set class that uses C#'s bit-wise operations.

Creating the PascalSet Class

Understand that to implement a set class that uses the efficient bit-wise operators the set's universe must be known. This is akin to the way Pascal uses sets, so in honor of the Pascal programming language I have decided to name this set class the `PascalSet` class. `PascalSet` restricts the universe to a range of integers or characters (just like the Pascal programming language). This range can be specified in the `PascalSet`'s constructor.

```
public class PascalSet : ICloneable, ICollection, IEnumerable
{
   // Private member variables
   private int lowerBound, upperBound;
   private BitArray data;

   public PascalSet(int lowerBound, int upperBound)
   {
      // make sure lowerbound is less than or equal to upperbound
      if (lowerBound > upperBound)
         throw new ArgumentException("The set's lower bound cannot be
            greater than its upper bound.");

      this.lowerBound = lowerBound;
      this.upperBound = upperBound;

      // Create the BitArray
      data = new BitArray(upperBound - lowerBound + 1);
   }

   ...
}
```

So, to create a `PascalSet` whose universe is the set of integers between -100 and 250, the following syntax could be used:

```
PascalSet mySet = new PascalSet(-100, 250);
```

Implementing the Set Operations

`PascalSet` implements the standard set operations—union, intersection, and set difference—as well as the standard relational operators—subset, proper subset, superset, and proper superset. The set operations union, intersection, and set difference, all return a new `PascalSet` instance, which contains the result of unioning, intersecting, or set differencing. The following code for the `Union(`*`PascalSet`*`)` method illustrates this behavior:

```
public virtual PascalSet Union(PascalSet s)
{
    if (!AreSimilar(s))
        throw new ArgumentException("Attempting to union two dissimilar
           sets.  Union can only occur between two sets with the same universe.");

    // do a bit-wise OR to union together this.data and s.data
```

```
    PascalSet result = (PascalSet)Clone();
    result.data.Or(s.data);

    return result;
}

public static PascalSet operator +(PascalSet s, PascalSet t)
{
   return s.Union(t);
}
```

The `AreSimilar(PascalSet)` method determines if the `PascalSet` passed has the same lower and upper bounds as the `PascalSet` instance. Therefore, union (and intersection and set difference) can only be applied to two sets with the same universe. You could make a modification to the code here to have the returned `PascalSet`'s universe be the union of the two universe sets, thereby allowing sets with non-disjoint universes to be unioned. If the two `PascalSets` have the same universe, then a new `PascalSet`—`result`—is created, which is an exact duplicate of the `PascalSet` instance whose `Union()` method was called. This cloned `PascalSet`'s contents are modified using the bit-wise OR method, passing in the set whose contents are to be unioned. Notice that the `PascalSet` class also overloads the + operator for union (just like the Pascal programming language).

Similarly, the **PascalSet** class provides methods for intersection and set difference, along with the overloaded operators * and –, respectively.

Enumerating the PascalSet's Members

Because sets are an *unordered* collection of elements, it would not make sense to have `PascalSet` implement `IList`, as collections that implement `IList` imply that the list has some ordinal order. Because `PascalSet` is a collection of elements, though, it makes sense to have it implement `ICollection` and `IEnumerable`. Because `PascalSet` implements `IEnumerable`, it needs to provide a `GetEnumerator()` method that returns an `IEnumerator` instance allowing a developer to iterate through the set's elements. This method simply iterates through the set's underlying `BitArray`, returning the corresponding value for each bit with a value of 1.

```
public IEnumerator GetEnumerator()
{
    int totalElements = Count;
    int itemsReturned = 0;
    for (int i = 0; i < this.data.Length; i++)
    {
        if (itemsReturned >= totalElements)
            break;
        else if (this.data.Get(i))
            yield return i + this.lowerBound;
    }
}
```

To enumerate the PascalSet, you can simply use a `foreach` statement. The following code snippet demonstrates creating two PascalSet instances and then enumerating the elements of their union:

```
PascalSet a = new PascalSet(0, 255, new int[] {1, 2, 4, 8});
PascalSet b = new PascalSet(0, 255, new int[] { 3, 4, 5, 6 });

foreach(int i in a + b)
   MessageBox.Show(i);
```

This code would display seven messageboxes, one after the other, displaying values 1, 2, 3, 4, 5, 6, and 8.

The complete code for the `PascalSet` class is included as a download with this article. Along with the class, there is an interactive Windows Forms testing application, `SetTester`, from which you can create a `PascalSet` instance and perform various set operations, viewing the resulting set.

Maintaining a Collection of Disjoint Sets

Next time you do a search at Google notice that with each result there's a link titled "Similar Pages." If you click this link, Google displays a list of URLs that are related to the item whose "Similar Pages" link you clicked. While I don't know how Google particularly determines how pages are related, one approach would be the following:

- Let *x* be the Web page we are interested in finding related pages for.
- Let S_1 be the set of Web pages that *x* links to.
- Let S_2 be the set of Web pages that the Web pages in S_1 link to.
- Let S_3 be the set of Web pages that the Web pages in S_2 link to.
- ...
- Let S_k be the set of Web pages that the Web pages in S_{k-1} link to.

All of the Web pages in S_1, S_1, up to S_k are the related pages for *x*. Rather than compute the related Web pages on demand, we might opt to create the set of related pages for *all* Web pages once, and to store this relation in a database or some other permanent store. Then, when a user clicks on the "Similar Pages" link for a search term, we simply query the display to get the links related to this page.

Google has some sort of database with all of the Web pages it knows about. Each of these Web pages has a set of links. We can compute the set of related Web pages using the following algorithm:

1. For each Web page in the database create a set, placing the single Web page in the set. After this step completes, if we have *n* Web pages in the database, we'll have *n* one-element sets.
2. For a Web page *x* in the database, find all of those Web pages it directly links to. Call these linked-to pages *S*. For each element *p* in *S*, union the set containing *p* with *x*'s set.
3. Repeat step 2 for all Web pages in the database.

After step 3 completes, the Web pages in the database will be partitioned out into related groups. To view a graphical representation of this algorithm in action, consult Figure 1.

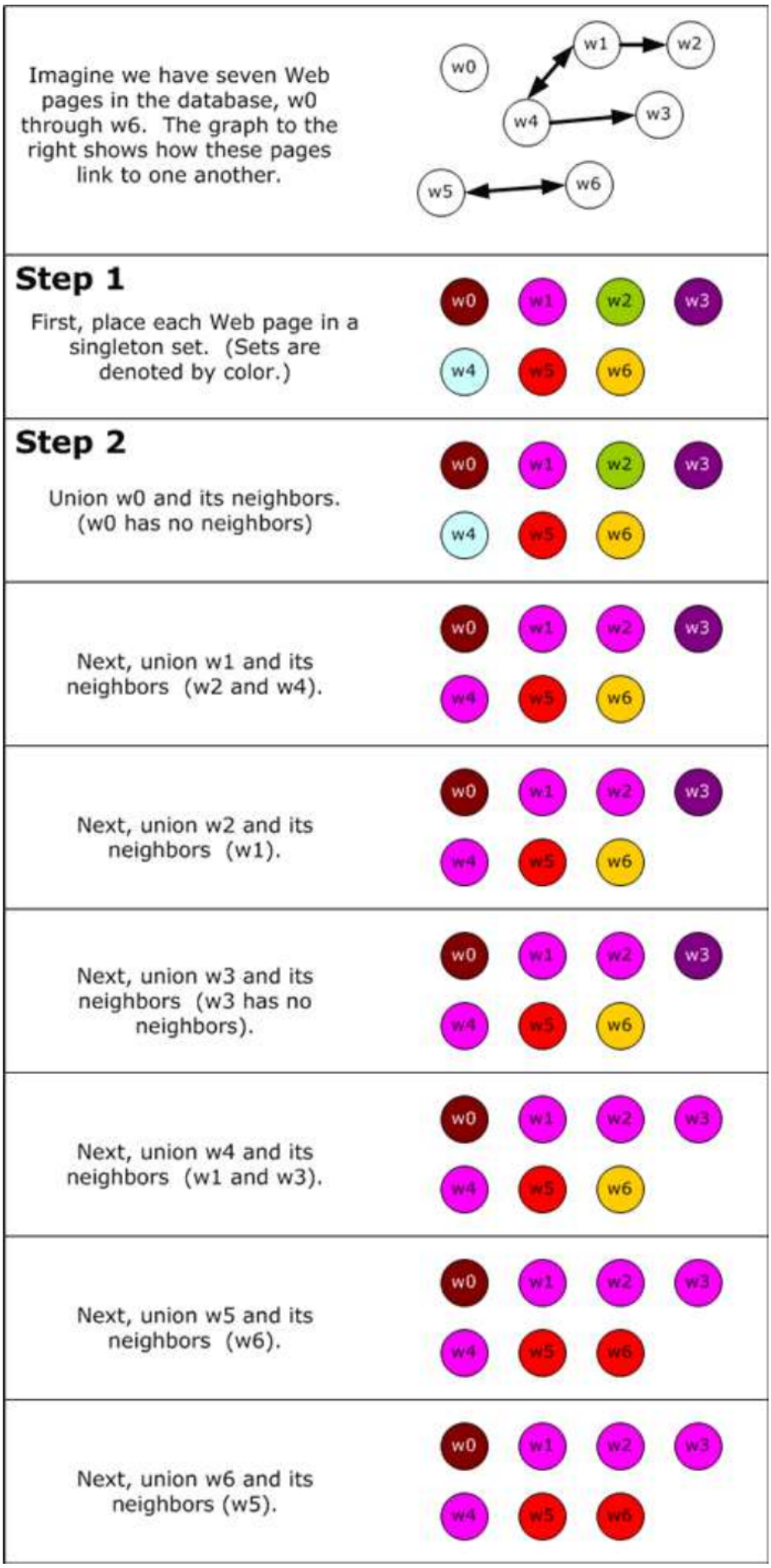


Figure 1. A graphical representation of an algorithm for grouping linked web pages.

Examining Figure 1, notice that in the end, there are three related partitions:

- w0
- w1, w2, w3, and w4
- w5 and w6

So, when a user clicks the "Similar Pages" link for w2, they would see links to w1, w3, and w4; clicking the "Similar Pages" link for w6 would show only a link to w5.

Notice that with this particular problem only one set operation is being performed—union. Furthermore, all of the Web pages fall into *disjoint sets*. Given an arbitrary number of sets, these sets are said to be *disjoint* if they share no elements in common. {1,2,3} and {4,5,6} are disjoint, for example, while {1,2,3} and {2,4,6} are not, because they share the common element 2. In all stages shown in Figure 1, each of the sets containing Web pages are disjoint. That is, it's never the case that one Web page exists in more than one set at a time.

When working with disjoint sets in this manner, we often need to know what particular disjoint set a given element belongs to. To identify each set we arbitrarily pick a *representative*. A representative is an element from the disjoint set that uniquely identifies that entire disjoint set. With the notion of a representative, I can determine of two given elements are in the same set by checking to see if they have the same representative.

A disjoint set data structure needs to provide two methods:

- **GetRepresentative(element):** this method accepts an element as an input parameter and returns the element's representative element.
- **Union(element, element):** this method takes in two elements. If the elements are from the same disjoint set, then `Union()` does nothing. If, however, the two elements are from different disjoint sets, then `Union()` combines the two disjoint sets into one set.

The challenge that faces us now is how to efficiently maintain a number of disjoint sets, where these disjoint sets are often merged from two sets into one. There are two basic data structures that can be used to tackle this problem—one uses a series of linked lists, the other collection of trees.

Maintaining Disjoint Sets with Linked Lists

In Part 4 of this article series we took a moment to look at a Quick Primer on Linked Lists. Recall that linked lists are a set of nodes that typically have a single reference to their next neighbor. Figure 2 shows a linked list with four elements.

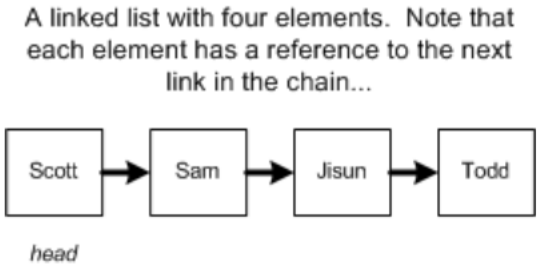


Figure 2. A linked list with four elements

For the disjoint set data structure, a set is represented using a modified linked list. Rather than just having a reference to its neighbor, each node in the disjoint set linked list has a reference to the set's representative. As Figure 3 illustrates, *all* nodes in the linked list point to the *same* node as their representative, which is, by convention, the head of the linked list. (Figure 3 shows the linked list representation of the disjoint sets from the

final stage of the algorithm dissected in Figure 1. Notice that for each disjoint set there exists a linked list, and that the nodes of the linked list contain the elements of that particular disjoint set.)

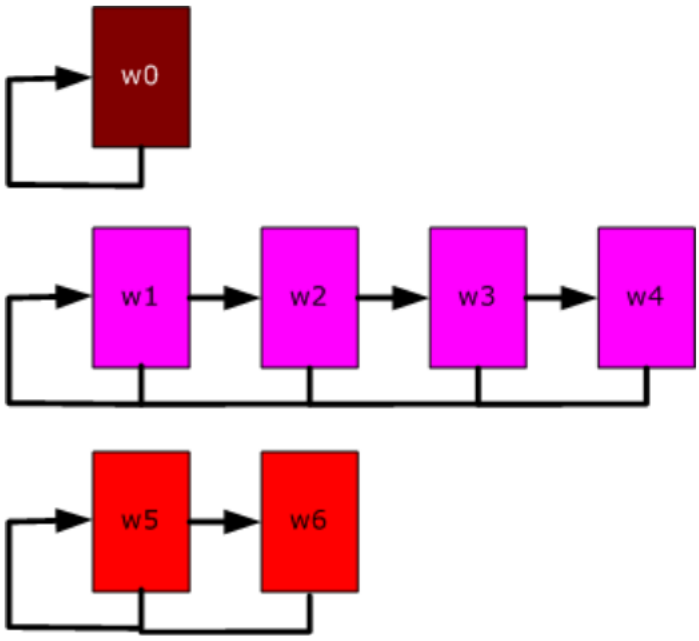


Figure 3. A linked list representation of the disjoint sets from the final stage of the algorithm dissected in Figure 1.

Because each element in a set has a direct reference back to the set's representative, the `GetRepresentative(element)` method takes constant time. (To understand why, consider that regardless of how many elements a set has, it will always take one operation to find a given element's representative, because it involves just checking the element's representative reference.)

Using the linked list approach, combining two disjoint sets into one involves adding one linked list to the end of another, and updating the representative reference in each of the appended nodes. The process of joining two disjoint sets is depicted in Figure 4.

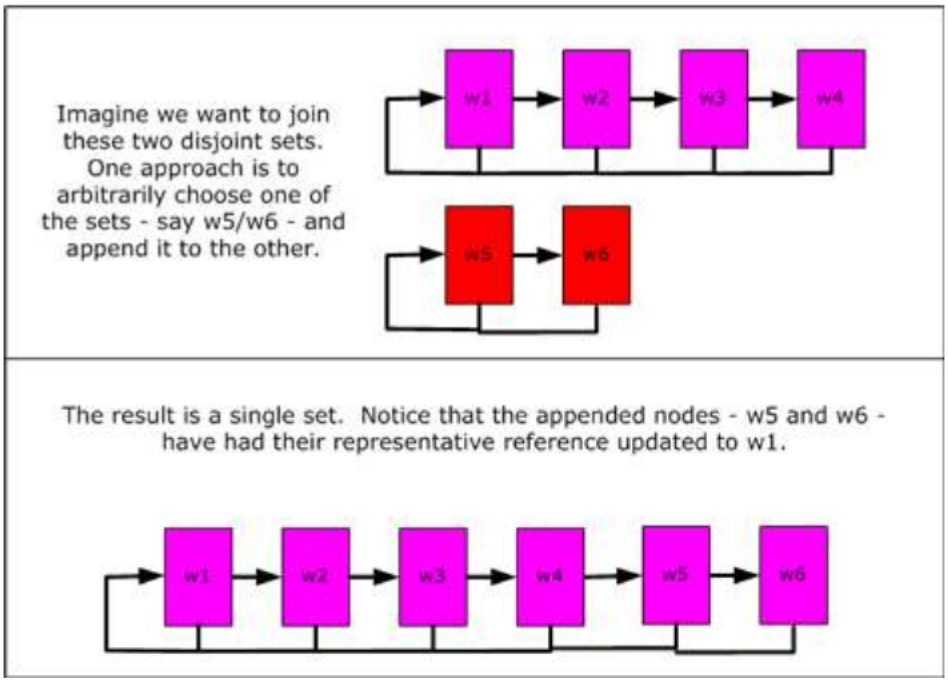


Figure 4. The process of joining two disjoint sets

When unioning together two disjoint sets, the correctness of the algorithm is not affected by which of the two sets is appended to the other. However, the running time can be. Imagine that our union algorithm randomly chose one of the two linked lists to be appended to the other. By a stroke of bad luck, imagine that we always chose the longer of the two linked lists to append. This can negatively impact the running time of the union operation since we have to enumerate all of the nodes in the appended linked list to update their representative reference. That is, imagine we make n disjoint sets, S_1 to S_n. Each set would have one element. We could then do n - 1 unions, joining all n sets into one big set with n elements. Imagine the first union joined S_1 and S_2, having S_1 be the representative for this two element unioned set. Since S_2 only has one element, only one representative reference would need to be updated. Now, imagine S_1—which has two elements—is unioned with S_3, and S_3 is made the representative. This time two representative references—S_1's and S_2's—will need to be updated. Similarly, when joining S_3 with S_4, if S_4 is made the representative of the new set, three representative references will need to be updated (S_1, S_2, and S_3). In the (n-1)th union, n-2 representative references will need to be updated.

Summing up the number of operations that must be done for each step, we find that the entire sequence of steps—n make set operations and n-1 unions—takes quadratic time—$O(n^2)$.

This worst-case running time can transpire because it is possible that union will choose the longer set to append to the shorter set. Appending the longer set requires that more nodes' representative references need to be updated. A better approach is to keep track of the size of each set, and then, when joining two sets, to append the smaller of the two linked lists. The running time when using this improved approach is reduced to $O(n \log_2 n)$. A thorough time analysis is a bit beyond the scope of this article, and is omitted for brevity. Refer to the readings in the References section for a formal proof of the time analysis.

To appreciate the improvement of $O(n \log_2 n)$ from $O(n^2)$, observe Fifvcgure 5, which shows the growth rate of n^2 in blue, and the grown rate of $n \log_2 n$ in pink. For small values of n, these two are comparable, but as n exceeds 32, the $n \log_2 n$ grows much slower than n^2. For example, performing 64 unions would require over 4,000 operations using the naive linked list implementation, while it would take only 384 operations for the optimized linked list implementation. These differences become even more profound as n gets larger.

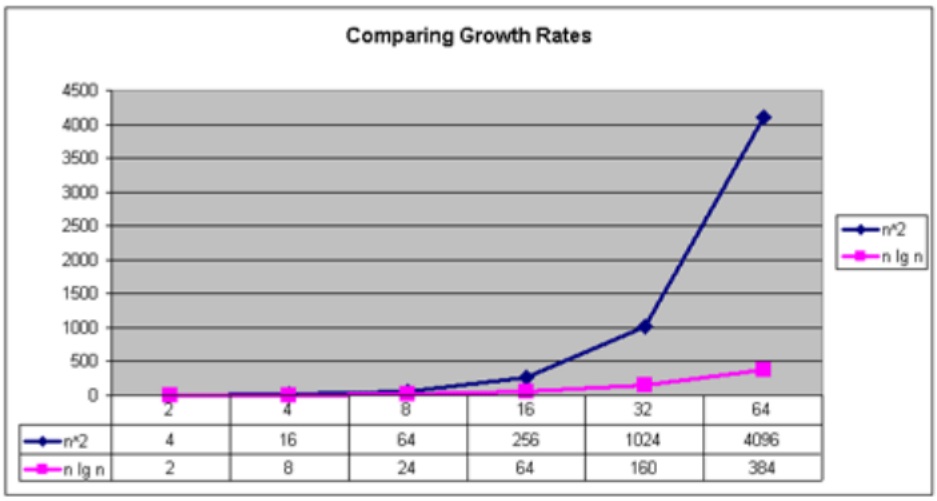


Figure 5. Growth rates of n^2 and $n \log_2$

Maintaining Disjoint Sets with a Forest

Disjoint sets can also be maintained using a *forest*. A forest is a set of trees (get it?). Recall that with the linked list implementation, the set's representative was the head of the list. With the forest implementation, each set is implemented as a tree, and the set's representative is the root of the tree. (If you are unfamiliar with what trees are, consider reading Part 3 of this article series, where we discussed trees, binary trees, and binary search trees.)

With the linked list approach, given an element, finding its set's representative was fast because each node had a direct reference to its representative. However, with the linked list approach unioning took longer because it involved appending one linked list to another, which required that the appended nodes' representative references be updated. The forest approach aims at making unions fast, at the expense finding a set's representative given an element in the set.

The forest approach implements each disjoint set as a tree, with the root as the representative. To union together two sets, one tree is appended as a child of the other. Figure 6 illustrates this concept graphically.

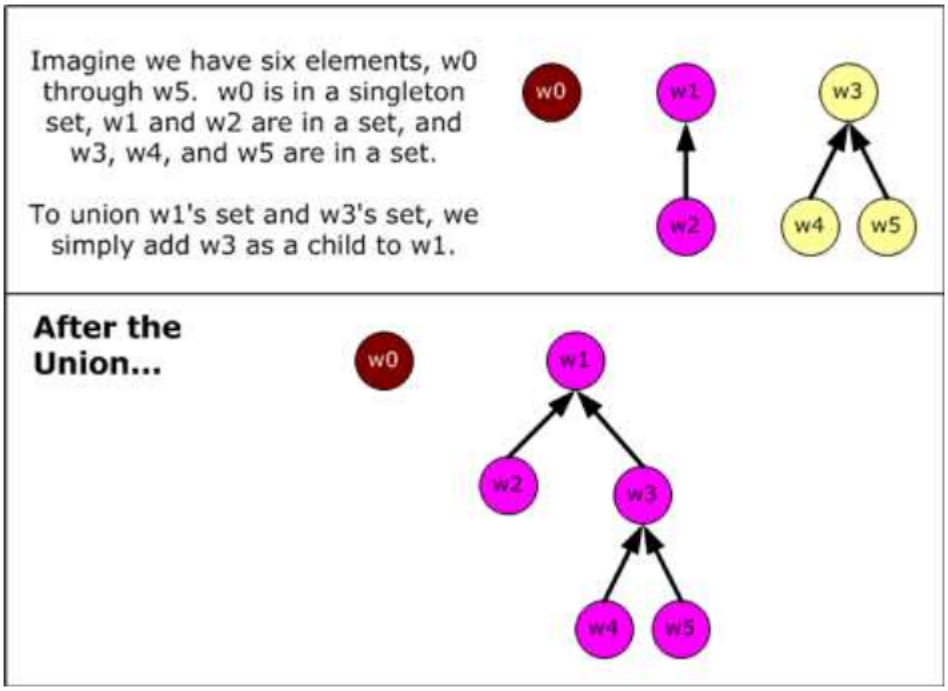


Figure 6. The union of two sets

To union two sets together requires constant time, as only one node needs to have its representative reference updated. (In Figure 6, to union together the w1 and w3 sets, all we had to do was have w3 update its reference to w1—nodes w4 and w5 didn't need any modification.)

Compared to the linked list implementation, the forest approach has improved the time required for unioning two disjoint sets, but has worsened the time for finding the representative for a set. The only way we can determine a set's representative, given an element, is to walk up the set's tree until we find the root. Imagine that we wanted to find the representative for w5 (after sets w1 and w3 had been unioned). We'd walk up the tree until we reached the root—first to w3, and then to w1. Hence, finding the set's representative takes time relative to the depth of the tree, and not constant time as it does with the linked list representation.

The forest approach offers two optimizations that, when both employed, yield a linear running time for performing *n* disjoint set operations, meaning that each single operation has an average constant running time. These two optimizations are called union by rank and path compression. What we are trying to avoid with these two optimizations is having a sequence of unions generate a tall, skinny tree. As discussed in Part 3 of this article series, the ratio of a tree's height to breadth typically impacts its running time. Ideally, a tree is fanned out as much as possible, rather than being tall and narrow.

The Union by Rank Optimization

Union by rank is akin to the linked list's optimization of appending the shorter list to the longer one. Specifically, union by rank maintains a rank for each sets' root, which provides an upperbound on the height of the tree. When unioning two sets, the set with the smaller rank is appended as a child of the root with the larger rank. Union by

rank helps ensure that our trees will be broad. However, even with union by rank we might still end up with tall, albeit wide, trees. Figure 7 shows a picture of a tree that might be formed by a series of unions that adhere only to the union by rank optimization. The problem is that leaf nodes on the right hand side still must perform a number of operations to find their set's representative.

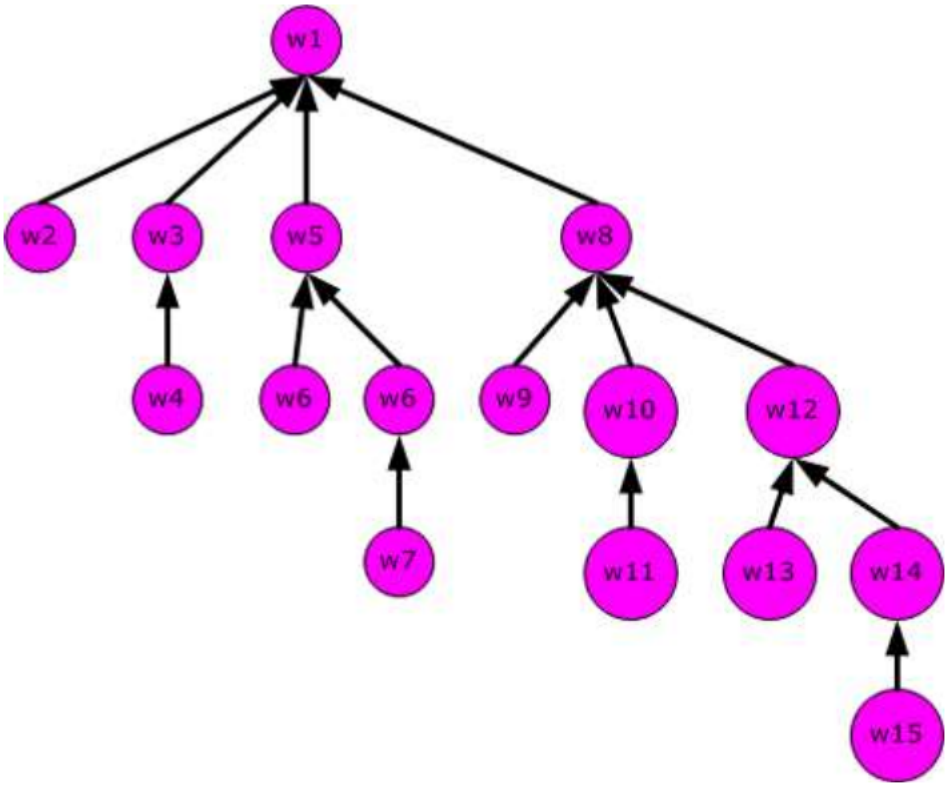


Figure 7. A tree that might be formed by a series of unions that adhere only to the union by rank optimization

Note The forest approach, when implementing just the union by rank optimization, has the same running time as the optimized link list implementation.

The Path Compression Optimization

Because a tall tree makes finding a set's representative expensive, ideally we'd like our trees to be broad and flat. The path compression optimization works to flatten out a tree. As we discussed earlier, whenever an element is queries for its set's representative, the algorithm walk up the tree to the root. The way the path compression optimization works is in this algorithm, the nodes that are visited in the walk up to the root have their parent reference updated to the root.

To understand how this flattening works, consider the tree in Figure 7. Now, imagine that we need to find the set representative for w13. The algorithm will start at w13, walk up to w12, then to w8, and finally to w1, returning w1 as the representative. Using path compression, this algorithm will also have the side effect of updating w13 and w12's parents to the root—w1. Figure 8 shows a screenshot of the tree after this path compression has occurred.

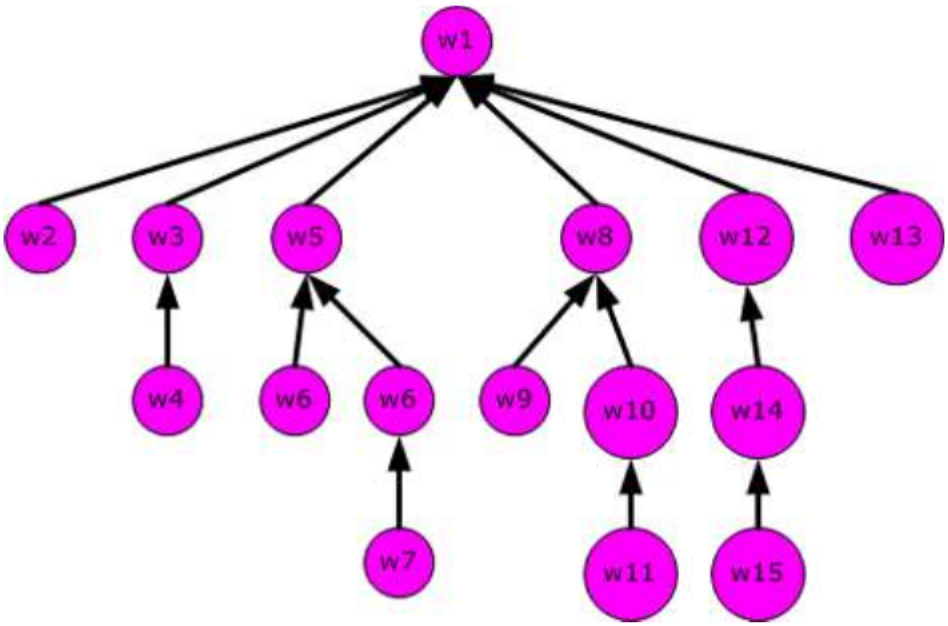


Figure 8. A tree after path compression

Path compression pays a slight overhead the first time when finding a representative, but benefits future representative lookups. That is, after this path compression has occurred, finding the set representative for w13 takes one step, because w13 is a child of the root. In Figure 7, prior to path compression, finding the representative for w13 would have taken three steps. The idea here is that you pay for the improvement once, and then benefit from the improvement each time the check is performed in the future.

When employing both the union by rank and path compression algorithms, the time is takes to perform *n* operations on disjoint sets is linear. That is, the forest approach, utilizing both optimizations, has a running time of *O*(*n*). You'll have to take my word on this, as the formal proof for the time complexity is quite lengthy and involved, and could easily fill several printed pages. If you are interested, though, in reading this multi-page time analysis, refer to the "Introduction to Algorithms" text listed in the references.

References

- Alur, Rajeev. "Disjoint Sets." Available online at: http://www.cis.upenn.edu/~cse220/h29.pdf.
- Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.
- Devroye, Luc. "Disjoint Set Structures." Available online at: http://www.cs.mcgill.ca/~cs251/OldCourses/1997/topic24/.

Scott Mitchell, author of six books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies because 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.