

Part I – Structures and Concepts

I. Introduction

Data structures provide a way to organize the data for your program in a way that is efficient and easy to use. For example, in an air combat game, there would likely be a data structure keeping track of the thirty missiles your plane has fired, the six other planes in your squadron, and the fifty alien ships that you are trying to shoot down. There are many different data structures that might be used to keep track of these objects, each of which is suited to organizing the data differently. This article is the first in a series that is intended to be a guide to using data structures in games, and should help you decide which data structure is best suited for a task. The articles are designed to be at an introductory level. However, the series also contains examples of applications of data structures to games, so programmers who are strong in other areas, but with little game experience, may benefit as well.

At the most basic level, data structures allow you to perform three basic operations: adding data, accessing data, and modifying data. By analyzing the most frequent ways that your data is used, you can decide which data structure is appropriate. Each data structure is designed to be good at one kind of usage pattern, often at the expense of another. For example, a data structure might excel at accessing data in order, using a simple for loop. The same data structure might be much slower when asked to access a random element, however. Some data structures are geared towards easily adding new data, while others are fixed size. One of the arts in software development is learning to match data structures to how their data will be used. Poor data structure choice is often a significant factor in game performance problems.

The decision of which data structure to use, then, boils down to an analysis of how the data is likely to be used. To use the previous example of an air combat game, how often will you need to add new missiles? Will they be added every frame, or is the number of missiles a constant? How will you examine the data that makes up a missile? Will you always access missiles one after another, or will you need access to any missile at any time? Once you have a sense of how your data is likely to be used, you can decide which data structure is appropriate.

One quick disclaimer: any time this article discusses the internal workings of .NET Framework or XNA Framework classes, remember that private implementation details are always subject to change from version to version.

II. Big O

There are a couple of things to explain before we can begin discussing the data structures themselves. First, in order to talk about the performance of different data structures, we're going to need to have some kind of framework for analyzing performance. If I tried to write this whole article by calling everything fast, slow, slowish, or not-that-great-but-faster-than-the-other-guy, I'd go crazy.

To that end, I'm going to briefly explain something called *Big O notation*. Big O notation gives us a sense of how many steps it will take to complete a task or algorithm. Say, for example, I have the following function:

```
float Double(float x)
{
    return x * 2;
}
```

Examining this function, we can see that it will always take one step to complete, no matter what the input to the function is. This means the function will run in what's called "constant time," expressed in Big O notation as $O(1)$. Another function might be one that computes n factorial ($n!$):

```
int NFactorial(int n)
{
    int result = 1;
    for (int i = n; i > 0; i--)
    {
        result *= i;
    }
    return result;
}
```

If we assume that the code that handles the assignment, comparison, and return statements is trivial and can be ignored, we see that the performance of this function will be dominated by the line `result *= i;`. Since that line of code is in a loop that will be executed n times, this function is $O(n)$, "linear time."

Let's look at one more function.

```
int Badness(int n)
{
    int result = 1;
    for (int i = n; i > 0; i--)
    {
        for (int j = n; j > 0; j--)
        {
            result *= j;
        }
    }
    return result;
}
```

The function may seem to be a bit contrived—what's the point of it, after all? However, it's worth mentioning the double loop that this function uses. This is something that will pop up frequently in games programs, particularly in collision code, as we'll see later. So, what's so bad about this function? There's still only one multiplication happening. Again, the multiplication is inside of a loop, so it will happen n times, but the loop is also inside a loop, so it too will happen n times. All told, our one lonely multiplication will actually happen n^2 times, giving us a Big O of $O(n^2)$, which is called "quadratic time."

Watch out for functions like these: they don't look that bad from glancing at the code, and with small values of n they are not that bad. As n increases, though, you will quickly find you have a monster on your hands. Any time you're writing an algorithm, and you find yourself thinking "I'll just check everything against everything else," stop and ask yourself if there is a better way you could do it.

Here are several common running times and their Big O notation, from fastest to slowest:

Constant time	$O(1)$
Log n time	$O(\log(n))$
Linear time	$O(n)$
Quadratic time	$O(n^2)$
Factorial time	$O(n!)$

When expressing a function's running time in Big O, only the most significant term of the function is written. For example, say we've analyzed another function that computes factorials, and we've seen that it takes $n^2 + n + 5$ steps to complete. Once n grows very large, the n^2 term in $n^2 + n + 5$ will become the most significant term in the function: it will contribute the most to the result. For clarity when comparing different functions' run times, only the most significant term is written. Therefore, our function $n^2 + n + 5$ has a Big O of $O(n^2)$. Simplifying like this allows us to easily compare the running times of various functions, jumping right to the "meat" of the functions' performance. Constant time functions are always written as $O(1)$.

This "most significant term only" simplification obviously breaks down, however, when n is not large. For example, say we have two functions with running times of $n + 100$ and n^2 . The Big O of the first is $O(n)$, and the second is $O(n^2)$. If the two functions accomplish the same goal, a naïve approach would be to simply always use the first function, since $O(n)$ is faster than $O(n^2)$. However, notice that for all values of n less than 10, the second function, $O(n^2)$, is actually faster.

Situations like this are fairly common. In many cases, there are two ways to accomplish a certain task: the brute force approach, and the clever approach. The clever approach is often faster, but requires some setup work beforehand. With small data sets, it's often faster to just use the brute force algorithm, rather than pay the setup cost of the cleverer one. It is worthwhile to consider these "hidden costs" when determining which algorithm to use.

For a tabular view of Big O notation and the pros and cons of each algorithm, see [Cheat Sheet: Speed of Common Operations](#).

III. Generics

The last thing I need to cover before we can start discussing data structures themselves is a technology called *generics*. Generics were added to C# in version 2.0, and provide a way to reuse code while still having type safety. For example, let's say I have a class called a **Holder**. Its only purpose is to

hold on to one piece of data, so that different callers can share data between themselves without needing to know who else they might be sharing with. If I want the **Holder** to hold integers, it might look like this:

```
class IntHolder
{
    private int heldValue;
    public int HeldValue
    {
        get { return heldValue; }
        set { heldValue = value; }
    }
}
```

Now imagine I want a **Holder** that holds floats, or Booleans, or anything. I have to make a new class for every single kind of thing I might want to hold. This is a pretty trivial class, so it might not be that much code, but it clearly demonstrates the need for a better solution.

Alternatively, the **Holder** could hold objects, so that one class can hold anything, but this isn't ideal either. When someone accesses **Holder's HeldValue**, the first thing they'll do is cast it to what they expect it to be so they can use it. For example, suppose person A puts an int value into the **Holder**. Then person B comes along and tries to get at the value in the **Holder**. The problem is B thinks it's supposed to be a string, and the minute he tries to cast it that way, things will go downhill fast. Obviously, the type safety that we had with our separate **IntHolder** and **StringHolder** structures was quite useful.

C# 2.0 addressed this issue by adding generics, which are extremely powerful and useful. Their syntax is similar to that of C++'s templates (meaning that it's confusing). Changing our **Holder** to a generic would make it look like this:

```
class Holder<T>
{
    private T heldValue;
    public T HeldValue
    {
        get { return heldValue; }
        set { heldValue = value; }
    }
}
```

The **<T>** bit means our **Holder** is a generic class, specialized on some type that will be called **T**. Then, when someone comes along to use our **Holder**, they replace the **T** with the type they want, like so:

```
Holder<int> holder = new Holder<int>();
holder.HeldValue = 10;

Holder<string> stringHolder = new Holder<string>();
stringHolder.HeldValue = "Camelus Maximus";
```

Now we've got the best of both worlds: code reuse and type safety. Most of the data structures we're going to examine will be generics, so it's important to be comfortable working with them.

IV. Conclusion

Now that we've laid the groundwork for examining data structures, in Part II we can begin to discuss the actual data structures. I realize that this first part may be a bit of a tease (a data structures article that doesn't actually talk about any data structures?), but understanding Big O and generics is crucial to the rest of these articles, so it's important to discuss them early.

Part II – Arrays, Lists, and Collections

I. Introduction

In Part I of this series of articles, I wrote about the importance of data structures in games (and all programming), and discussed Big O notation and generics. With that out of the way, in this part I can get to the meat of the article: the actual data structures.

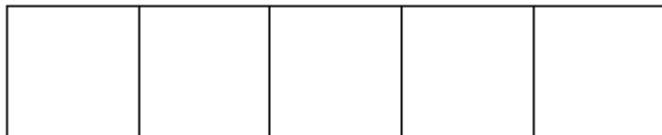
II. The Array

So, with that, let's look at the simplest data structure: the array. Arrays are built into almost every single programming language, and C# is no exception. You can think of an array as a group of multiple instances of some data type. Arrays are fixed size, and can hold only one kind of data. For example, an array of five integers can hold only integers, and it can hold only five of them. In C#, you create arrays by using the [] syntax:

```
int[] myArray = new int[5];
```

The **myArray** array is declared and is set to a five-integer array. You can then access and modify the elements of this array by using an index. The index is zero based, so to access the first element of **myArray**, use the index zero. For example,

```
int theFirst = myArray[0];  
int theLast = myArray[4];
```



myArray[1] myArray[3]
myArray[0] myArray[2] myArray[4]

Arrays are attractive mainly because of their simplicity. Whereas most other data structures incur additional processing overhead, arrays do not. They are also very memory efficient data structures. The five-integer array takes up the same amount of space in memory as five separate integers would, with a few extra bytes for CLR bookkeeping. Since the elements are accessed by index, accesses and modifications are fast, taking constant time, $O(1)$. This is true whether the accesses are in order or random order: in other words, it is just as fast to access **veryBigArray[0]** as it is **veryBigArray[1000]**. This will not be the case for some other data structures.

The major drawback to arrays is that they are fixed size. This causes problems if you want to add new objects to your array. For example, if you have an array of ten integers, and you want to add an eleventh, you cannot simply resize the array. You must instead create a new array of the proper size, and then copy all of the elements of the old array to the new one. This copy takes linear time, $O(n)$, so it can become very expensive, depending on the size of the old array. Other data structures can resize much more efficiently. Inserting items into the middle of an array is similarly slow: again, you must create a new array, and then copy the elements in the old array to the new one. Removing elements from the array presents the same problem—you must create a new array, and then copy the remaining elements, one by one.

Given these advantages and drawbacks, you can see that arrays are best used when storing data of a fixed size. A good example might be a tic-tac-toe grid, or a chess board: it is unlikely that a chess board would need to double in size halfway through the game.

III. Lists

The .NET Framework provides **System.Collections.Generic.List**, a data structure that has characteristics that are similar to that of the array and adds functionality for resizing.

System.Collections.Generic.List is a generic type, so to create one we use that crazy syntax described earlier in the article. For example, the following code can be used to create an empty list of **Color** types:

```
List<Color> colors = new List<Color>();
```

To add more colors to the list, simply use the **Add** function.

```
colors.Add(Color.AliceBlue);  
colors.Add(Color.AntiqueWhite);  
colors.Add(Color.BlanchedAlmond);
```

Notice that we didn't need to know how many colors were going to be in the list up front. This is the biggest difference between **List** and **Array**, and is a useful distinction. Elements in the list are still accessed using the array brackets, just like arrays:

```
Color first = colors[0];  
Color last = colors[2];
```

Internally, **List** is implemented by using a private array. The array is more than large enough to hold all of the contents of the list. If more and more elements are added to the list, and the internal array can no longer hold all of them, a new, bigger array is created, and the elements are copied to that one. Using this knowledge of the internal implementation of **List**, we can tell some things about its performance characteristics.

From the previous section, we know that arrays can quickly access and modify their elements, taking constant $O(1)$ time. **List** does nothing to modify this behavior, so in general, **List** will be equally fast. (There will be a small cost associated with the additional bounds checking that **List** requires, but this

should be negligible.) The big performance benefit that **List** has over **Array** is the dynamic sizing. Remember that since arrays cannot be resized, every time you need to add a new element, you must copy the entire contents of the collection to a new array. This copy will take linear time. However, since **List**'s internal array typically has some free space, when calling **Add** you can simply put the new element on the end, in the first free space. A copy is necessary only when the internal array is no longer large enough to hold all of the elements. This makes the general case for **Add** much faster, taking constant time. Removing from the end of the **List** is also a constant time operation, for similar reasons: the **List** can simply "forget" about the removed element, and leave the array alone.

One thing you can do to improve the performance of **Add** is to pre-size the internal array. A common pattern with a **List** is to initialize it with some number of default objects. For example, the first thing you might do after creating a new list of enemies is to immediately enter a **for** loop and fill that list with 100 bad guys. There's obviously room for optimization here: If you know at creation time how big your list will need to be, why not use that information? If you know how many enemies you have, you can automatically size the list's internal array to the appropriate size, avoiding all the unnecessary array creations and copies. To that end, **List** has a constructor that takes an integer argument, n , which will be the initial size of the internal array.

So, **List** does a lot to improve the speed of adding and removing items at the end of the list. It is important to note that this benefit is true only at the end of the list. Inserting and removing at the middle is not much improved. Consider the best case scenario when inserting into the middle of a list. There is enough space to hold the new element, which you want to be at the index i . To insert the new item, you must make room for it by taking all of the elements at indices greater than and equal to i and shifting them to the right. If i is near the end of the list, there are not many items that must be copied, but as i approaches 0, performance deteriorates. In general, inserting into a **List**, like **Array**, takes linear time, $O(n)$. Again, removing from a **List** has similar performance characteristics to inserting, because in both cases you must shift the items, which takes linear time. So in general, lists might be slightly better suited to insert operations than arrays are, but they are still not the ideal data structure for situations that will require a lot of inserting or removing.

IV. Collections

The **Collection** class is another data structure provided by the .NET Framework. Internally, it's implemented as a **List**, so its performance characteristics will be similar. From a performance and functionality point of view, it does not differ significantly from a **List**.

Why does **Collection** exist, then? If you take a close look at the functions exposed by **List**, you'll see that there are no virtual functions. There is a performance penalty associated with virtual functions, and since **List** is designed to be as fast as possible, the designers decided that none of **List**'s functions should be virtual. This has the effect of making it impossible to derive from **List** in a useful way. The **Collection** class exists to fill this gap: it is designed to be easy to derive from. This makes **Collection** a good data structure to expose in APIs or libraries.

Say, for example, you've written a 2D sprite engine that you want to share with your friends. One of your functions, called **GetCollisions**, returns a **List** of sprites that are colliding with one another. If, for some reason, you want to change the way that **List** works, you're stuck. Since a **List** is not usefully derivable, you would have to change **GetCollisions** to return something other than a **List**. When you distribute the next version of your library, your friends' code wouldn't compile anymore. Since the **Collection** class is easy to derive from, this would have been a better choice. If **GetCollisions** had returned a **Collection** instead, you could have simply written a new class, **MyCollection**, that inherits from **Collection**. You could then change **GetCollisions** internally to use **MyCollection**, and your friends could continue to deal with it as if it were a **Collection**, oblivious to the tweaks you had made under the hood.

In some cases, this flexibility could be very useful. In your internal code, however, **List**'s speed will probably make it a better choice.

V. Conclusion

The key things to take away from this part of the article is that arrays are lightweight and simple, and that lists are more flexible, but still fast. Collections aren't optimized for speed, but are still useful as base classes to your own complex data structures. In Part III of this article, we continue the discussion of different data structures, and cover linked lists, stacks, and queues.

Part III – Linked Lists, Stacks, and Queues

I. Introduction

This is the third in a series of articles dealing with the use of data structures in game programming. Part I explained why this is an interesting topic, and explained Big O and generics, which are important to understand in order to get the most from this article. Part II began discussing actual data structures, covering arrays, lists, and collections. This article continues that discussion, and covers linked lists, stacks, and queues.

II. Linked Lists

All of the data structures that we have examined so far have not been ideally suited to insert or add operations. Lists were slightly better than arrays, but they still required linear time copies in some cases. What we need is a data structure that lets us insert in constant time, all the time. The **LinkedList** structure provides this functionality.

The core of the **LinkedList** can be found in the **LinkedListNode**, the class that makes up **LinkedList**. Every **LinkedListNode** is made up of two things: a bit of data, and references to the node's neighbors in the list.

```
public sealed class LinkedListNode<T>
{
    public LinkedListNode<T> Next { get; }
    public LinkedListNode<T> Previous { get; }
    public T Value { get; set; }
}
```

Every operation performed on **LinkedList** is based around “walking the links,” moving from one **LinkedListNode** to the next. **LinkedList** itself keeps a reference to the first node in the list, so the operations know where to begin. Inserting items into a **LinkedList**, then, will be much faster than doing so with an array or list. The implementation probably looks something like this:

```
public void InsertAfter( LinkedListNode<T> after,
    LinkedListNode<T> newNode )
{
    newNode.Previous = after;
    newNode.Next = after.Next;
    after.Next.Previous = newNode;
    after.Next = newNode;
}
```

Notice that the insertion does not depend in any way on how many elements are in the **LinkedList**: we’ve got a constant-time insertion. Removing is a similar operation, and will also take constant time. Both of these operations are much faster than with arrays or lists, which take linear time.

There is of course, a tradeoff: unlike an array, a **LinkedList** does not have quick constant-time access to a random element in the structure. Whereas with arrays, you can simply use [] to jump to any element, a **LinkedList** must walk the links, starting from the beginning until it reaches the desired index. The code probably looks something like this:

```
public LinkedListNode<T> Get( LinkedListNode<T> head, int n )
{
    LinkedListNode<T> toReturn = head;
    for (int i = 0; i < n; i++)
    {
        toReturn = toReturn.Next;
    }
    return toReturn;
}
```

Notice that **for** loop? We're back in the slow realm of linear time, $O(n)$.

Although random order access is slower than lists and arrays, it's important to note that the speed of in-order access is still fast. To see why, look at the following code, which calculates the sum of all if the members of an array and a linked list:

```
// Add every element of an int array together: this takes linear time
int[] someArray = new int[10];

// Insert array filling code here

int sum = 0;
for (int i = 0; i < someArray.Length; i++)
{
    sum += someArray[i];
}

// Add every element of a LinkedList together:
// This also takes linear time
LinkedList<int> someLinkedList = new LinkedList<int>();

// Insert list filling code here

sum = 0;
LinkedListNode<int> node = someLinkedList.First;
while (node != null)
{
    sum += node.Value;
    node = node.Next;
}
```

The contents of both loops are executed n times, meaning both loops take $O(n)$. In other words, if all your algorithm needs to do is operate on your entire data structure from one end to another, it'll be no different whether your structure is an array or a **LinkedList**—both will take linear time.

So, at a high level, in-order access is just as fast as using arrays and lists. However, it may be worthwhile to consider that in-order access may be slowed down in a **LinkedList** that has had many items inserted

or deleted. Despite the fact that insertions and removals operations themselves are fast, a **LinkedList** that has had many items removed and inserted in different locations typically has its data spread all over in memory. This means that as you move through your **LinkedList**, there will be many more cache misses as data is fetched from more "spread out" locations in memory. A full discussion of cache misses is out of scope for this document, but for more information, a quick internet search will yield plenty of information. It's a very important topic, but to callously gloss over the whole subject in a few words: cache misses are slow.

III. Stacks

Stacks are common data structures, and are commonly used in many different algorithms. They are interesting primarily for their functionality, not necessarily because of performance benefits. You can visualize a stack, just as you might guess from the name, as a stack of data. Just like a real world stack, when you add a new item, it goes on the top of a stack, and when you remove an item, you remove it from the top. This behavior is called "last in, first out," or LIFO, and makes many algorithms and data collections much easier to work with.

The .NET Framework **Stack** implementation provides three main functions: **Push**, which puts a new item on the top of the stack; **Pop**, which removes an item from the top of the stack; and **Peek**, which examines the element on the top of the stack. Like **List**, **Stack** is implemented using a dynamically growing array, so we know that adding new elements via **Push** will be $O(1)$ if there is enough space in the array, and $O(n)$ if not. The implementations of **Pop** and **Peek** are trivial, and take $O(1)$.

The performance characteristics of a **Stack**, then, are very similar to those of a **List**. For this reason, don't choose a **Stack** for performance reasons. Rather, choose a **Stack** when the algorithm lends itself to LIFO behavior.

IV. Queues

Queues exist for the same reason as stacks: not for performance reasons, but because the functionality they offer is well suited to different types of algorithms. The difference is that stacks provide last-in, first-out data storage, whereas queues are first in, first out (FIFO).

The .NET Framework implementation uses the functions **Enqueue**, **Dequeue**, and **Peek**. **Enqueue** adds an element to the end of the queue, **Dequeue** removes an element from the front of the queue, and **Peek** examines the next element to be de-queued. The internal implementation of **Queue** is similar to that of **Stack**, using an array to store data, which is resized as necessary as you add more elements. Again, adding new elements will be $O(1)$ if there is enough space, $O(n)$ if not. **Dequeue** and **Peek** are $O(1)$.

V. Conclusion

That concludes Part III. Part IV finishes our discussion of data structures, and covers dictionaries and trees, both of which are extremely useful in game programming.

Part IV – Dictionaries and Trees

I. Introduction

This is the last in a series of four articles discussing the use of data structures in games. If you missed the first three (or skipped them), Part I explained why data structures are important, and covered Big O and generics, which will be important in order to understand this article. Parts II and III discussed different data structures and their strengths and weaknesses. Part II covered arrays, lists, and collections, and Part III covered linked lists, stacks, and queues. This article will explain dictionaries and trees, albeit briefly – both of these data structures are complex, and a more in-depth discussion is unfortunately out of scope for this article.

II. Dictionaries

One thing that's important to note about all of the data structures that have been discussed so far is that none of them are well suited to searching their contents. For example, say we have a collection of players in our game. We want to find the player named "Bob" in that collection, because we have just aimed a gigantic missile attack at him. Every one of the collections so far will take $O(n)$ time to find Bob: they all have to go through their contents from beginning to end, checking each player to see if he has the desired name. In many cases, it can be desirable to quickly search through an entire collection to see if an element exists, or to jump straight to an element you know exists. The .NET Framework provides this with the **Dictionary**.

Dictionaries store elements as pairs of keys and values. You can search for a specific value by looking up its key. To use the previous example, the key would be the player's name, and the value stored would be the player itself.

Since every dictionary will have different types of keys and values, **Dictionary** has two generic parameters. The other collections discussed in this paper all have one generic parameter: just the type they are storing. Because of this, the syntax for creating dictionaries is different than the other collections:

```
List<string> list = new List<string>();  
Dictionary<string, Player> dictionary = new Dictionary<string, Player>();
```

The first line of code declares and instantiates a new list of strings. The second one declares and instantiates a **dictionary** that is keyed on strings, and stores a custom type called a **Player**.

A full discussion of how **Dictionary** is implemented in .NET is unfortunately out of scope for this article. To learn more about the implementation details, [An Extensive Examination of Data Structures – Part 2](#) contains great information. In fact, the whole series of articles is definitely a worthwhile read. In brief, the **Dictionary** maintains an internal array, which is dynamically resized. When a new element is added to the **Dictionary**, a number called a hash code is computed from the key. That hash code is used to

determine where the element will be placed in the internal array. Adding values can be done in two ways:

```
Dictionary<string, Player> dictionary = new Dictionary<string,
Player>();

dictionary["Bob"] = new Player("Bob", PlayerIndex.One);
dictionary.Add("Joe", new Player("Joe", PlayerIndex.Two));
```

In both of these cases, a hash code is computed from the string key, "Bob" or "Joe". The hash code is then used to store the **Player** object in the dictionary. Remember that in every other collection we've discussed, to find a specific element, we had to do a linear search through the collection, examining every one until the desired one is found. With a **Dictionary**, this is unnecessary; if we want to get the **Player** named Bob, we can simply compute the hash code of "Bob" to figure out where Bob would go, and then return the **Player** at that location.

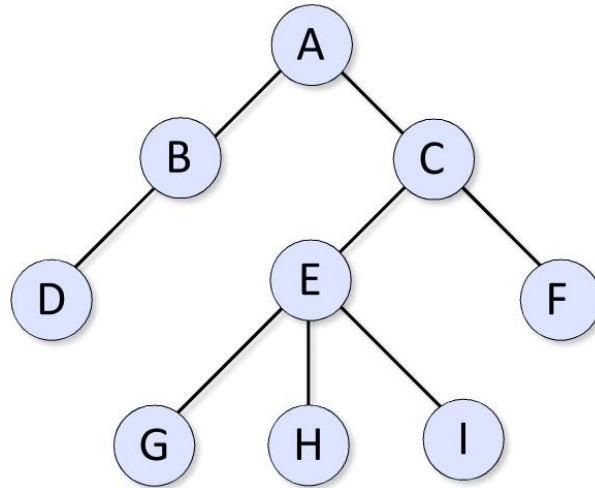
Like the other collections that are implemented with a dynamically resizing array, adding new elements is a best case $O(1)$, worst case $O(n)$. Removing is $O(1)$. In-order access and out of order accessing both take constant time; but it is important to note that this is not what dictionaries are optimized for. This is a good example of the problems with Big O that were discussed earlier. Although in-order access and out-of-order access both take constant time, the constant is a large number. There are a lot of steps associated with getting elements out of a **Dictionary**. If the primary reason you need a data structure is to store data that will simply be looped over, a **Dictionary** is not the ideal solution.

One common use for dictionaries in games is asset management. This can be seen in the XNA Framework **ContentManager** class. **ContentManager** has a **Load** method, which takes in a string parameter for the name of the asset to load. To provide this functionality, the **ContentManager** has an internal dictionary that maps from strings to loaded assets. When a user requests the asset "myTexture", the **ContentManager** checks the internal dictionary. If an asset by that name has been loaded already, the **ContentManager** returns it. Otherwise, it will load the asset, add it to the dictionary for later use, and then return it. This means that even if fifteen different classes in your game want to use the same texture, that texture is loaded into memory only once.

III. Trees

Trees are not one specific kind of data structure, but rather a class of data structures. There are many different kinds of trees, each designed to solve a different problem. Since there isn't one standard kind of tree, one isn't implemented in the .NET Framework, but trees are mentioned in this paper primarily because they are such a commonly used data structure in game programming.

Trees are similar to linked lists in some ways: the real core of the functionality is not found in the collection itself, but in its individual nodes. Remember that **LinkedListNode** has a **.Next** reference that points to the next **LinkedListNode** in the collection. A **TreeNode** is similar, but each **TreeNode** will have not just one reference, but multiple **.Children** references. Just as the name implies, they can be best visualized as a tree: the trunk splits into multiple branches, and each branch can split again, and so on.

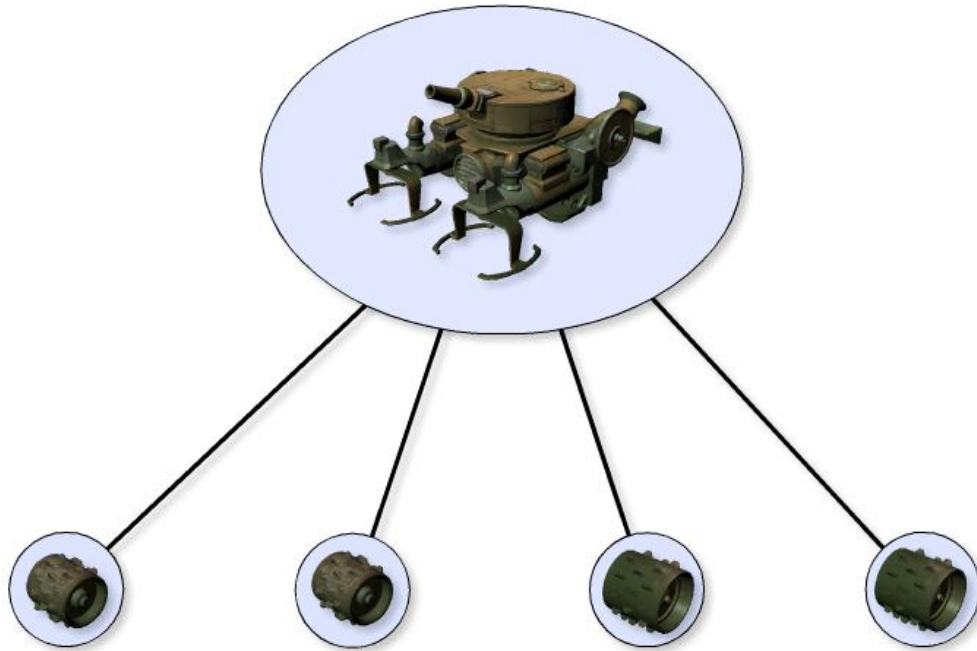


In this example, the root of the tree is node A. It has two children, B and C. There are many different restrictions you can put on different trees: some require each node to have a specific number of children, others have a maximum depth to which the tree can grow. (The depth of the tree is how many levels of hierarchy there are. In this example, the depth is 4: the first level is A, the second is B and C, and so on.)

Functions such as adding and removing elements, searching, and accessing elements will be very different from tree to tree. Therefore, it is not possible to give performance metrics for basic data structure operations on trees.

Tree structures and hierarchical structures appear in many different contexts in game programming. One of the most common is that of a transformation hierarchy. The XNA Framework **Model** class uses a transformation hierarchy, so it's possible that you may have already used a tree structure in your game code. To see what a transformation hierarchy is, and why it is useful, let's look at an example.

Let's say you're writing a 3D game with a tank in it. Among other things, the tank has four wheels. The wheels need to spin, obviously, so they've been created as separate 3D pieces; that way you can draw them independently of the rest of the parts of the tank. Call each one of those 3D pieces a mesh. Now, no matter how the tank moves or turns, the wheel meshes should stay attached, in the same position relative to the tank. The most common way to accomplish this is with a transformation hierarchy. Imagine that every mesh in your game is a node in a tree, and can have child nodes, just like any other tree. Each node also has information about the mesh's current position and orientation. Here's the cool part: nodes also "inherit" their position and orientation from their parent node. In more mathematical terms, the position and orientation of nodes are relative to their parent node. So, in the tank example, the tree structure would look like this:



The main tank is the parent, and there are four children, one for each of the wheels. Each wheel node has different position information offsetting it from the main tank. One wheel node's position puts the wheel on the front of the tank on the left; one is the front right, and so on. Since these wheel nodes also "inherit" position and orientation from parent node, the tank can move, turn, and go wherever it wants and the wheel nodes will stay in the correct place.

At first glance, that may not see like that big of a benefit: it is certainly possible to write code in your **Tank** class that computes the proper location for the wheels every time the tank moves. However, this is a very common pattern, and transformation hierarchies can get fairly deep. Imagine a human character, with a body, upper arm, lower arm, hand, and fingers: it quickly becomes apparent that hard-coding fixups in every one of your classes is not appropriate, and that a solution like this one is necessary.

More complicated tree-based structures, such as QuadTrees, Octrees, and KD-trees are also extremely common in games. They are used to divide up the game world into more manageable chunks for efficient collision detection and visibility checking. Each one of those data structures could easily be a whole article, so unfortunately it's not possible to go into detail about them here.

IV. Conclusion

Each one of the data structures described in this article warrants a thorough article of its own. Trees and dictionaries, in particular, deserve far more attention than they were given here. Both can be useful for game programming when used correctly.

In any case, I hope this article was at least useful as a starting point. Before you leap back into your code, however, let me give you one last rule of thumb: although it is always a good idea to keep performance in mind when writing your game, don't worry about performance too soon. Make your

code clean and easy to use first. When you do get performance problems, you'll know where to spend your time optimizing. Good luck and happy coding.

V. Further Reading

[An Extensive Examination of Data Structures](#) – A six part article on MSDN regarding data structures, written by Scott Mitchell of 4GuysFromRolla.com.

[Selecting a Collection Class](#) – An MSDN article with rules of thumb for data structure selection.