Data Structures Succinctly Part 1

by Robert Horvick

Syncfusion | Technology Resource Portal

Data Structures Succinctly

Part 1

By
Robert Horvick

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Clay Burch, Ph.D., director of technical support, Syncfusion, Inc.

Copy Editor: Courtney Wright

Acquisitions Coordinator: Jessica Rightmer, senior marketing strategist, Syncfusion, Inc.

Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Robert Horvick is the founder and Principal Engineer at Raleigh-Durham, N.C.-based Devlightful Software where he focuses on delighting clients with custom .NET solutions and video-based training. He is an active Pluralsight author with courses on algorithms and data structures, SMS and VoIP integration, and data analysis using Tableau.

He previously worked for nearly ten years as a Software Engineer for Microsoft, as well as a Senior Engineer with 3 Birds Marketing LLC, and as Principal Software Engineer for Itron.

On the side, Horvick is married, has four children, is a brewer of reasonably tasty beer, and enjoys playing the guitar poorly.

Chapter 1 Algorithms and Data Structures

Why Do We Care?

I assume you are a computer programmer. Perhaps you are a new student of computer science or maybe you are an experienced software engineer. Regardless of where you are on that spectrum, algorithms and data structures matter. Not just as theoretical concepts, but as building blocks used to create solutions to business problems.

Sure, you may know how to use the C# **`List`** or **`Stack`** class, but do you understand what is going on under the covers? If not, are you really making the best decisions about which algorithms and data structures you are using?

Meaningful understanding of algorithms and data structures starts with having a way to express and compare their relative costs.

Asymptotic Analysis

When we talk about measuring the cost or complexity of an algorithm, what we are really talking about is performing an analysis of the algorithm when the input sets are very large. Analyzing what happens as the number of inputs becomes very large is referred to as asymptotic analysis. How does the complexity of the algorithm change when applied to ten, or one thousand, or ten million items? If an algorithm runs in 5 milliseconds with one thousand items, what can we say about what will happen when it runs with one million? Will it take 5 seconds or 5 years? Wouldn't you rather figure this out before your customer?

This stuff matters!

Rate of Growth

Rate of growth describes how an algorithm's complexity changes as the input size grows. This is commonly represented using Big-O notation. Big-O notation uses a capital O ("order") and a formula that expresses the complexity of the algorithm. The formula may have a variable, *n*, which represents the size of the input. The following are some common order functions we will see in this book but this list is by no means complete.

Constant – *O*(1)

An *O*(1) algorithm is one whose complexity is constant regardless of how large the input size is. The 1 does not mean that there is only one operation or that the operation takes a small amount of time. It might take 1 microsecond or it might take 1 hour. The point is that the size of the input does not influence the time the operation takes.

```
public int GetCount(int[] items)
{
    return items.Length;
}
```

Linear – *O*(*n*)

An *O*(*n*) algorithm is one whose complexity grows linearly with the size of the input. It is reasonable to expect that if an input size of 1 takes 5 milliseconds, an input with one thousand items will take 5 seconds.

You can often recognize an *O*(*n*) algorithm by looking for a looping mechanism that accesses each member.

```
public long GetSum(int[] items)
{
    long sum = 0;
    foreach (int i in items)
    {
        sum += i;
    }

    return sum;
}
```

Logarithmic – *O*(log *n*)

An *O*(log *n*) algorithm is one whose complexity is logarithmic to its size. Many divide and conquer algorithms fall into this bucket. The binary search tree **`Contains`** method implements an *O*(log *n*) algorithm.

Linearithmic – *O*(*n* log *n*)

A linearithmic algorithm, or loglinear, is an algorithm that has a complexity of O(n log n). Some divide and conquer algorithms fall into this bucket. We will see two examples when we look at merge sort and quick sort.

Quadratic – *O*(n^2)

An *O*(n^2) algorithm is one whose complexity is quadratic to its size. While not always avoidable, using a quadratic algorithm is a potential sign that you need to reconsider your algorithm or data structure choice. Quadratic algorithms do not scale well as the input size grows. For example, an array with 1000 integers would require 1,000,000 operations to complete. An input with one million items would take one trillion (1,000,000,000,000) operations. To put this into perspective, if each operation takes one millisecond to complete, an *O*(n^2) algorithm that receives an input of one million items will take nearly 32 years to complete. Making that algorithm 100 times faster would still take 84 days.

We will see an example of a quadratic algorithm when we look at bubble sort.

Best, Average, and Worst Case

When we say an algorithm is *O*(*n*), what are we really saying? Are we saying that the algorithm is *O*(*n*) on average? Or are we describing the best or worst case scenario?

We typically mean the worst case scenario unless the common case and worst case are vastly different. For example, we will see examples in this book where an algorithm is *O*(1) on average, but periodically becomes *O*(*n*) (see **`ArrayList.Add`**). In these cases I will describe the algorithm as *O*(1) on average and then explain when the complexity changes.

The key point is that saying *O*(*n*) does not mean that it is always *n* operations. It might be less, but it should not be more.

What Are We Measuring?

When we are measuring algorithms and data structures, we are usually talking about one of two things: the amount of time the operation takes to complete (operational complexity), or the amount of resources (memory) an algorithm uses (resource complexity).

An algorithm that runs ten times faster but uses ten times as much memory might be perfectly acceptable in a server environment with vast amounts of available memory, but may not be appropriate in an embedded environment where available memory is severely limited.

In this book I will focus primarily on operational complexity, but in the Sorting Algorithms chapter we will see some examples of resource complexity.

Some specific examples of things we might measure include:

- Comparison operations (greater than, less than, equal to).
- Assignments and data swapping.
- Memory allocations.

The context of the operation being performed will typically tell you what type of measurement is being made.

For example, when discussing the complexity of an algorithm that searches for an item within a data structure, we are almost certainly talking about comparison operations. Search is generally a read-only operation so there should not be any need to perform assignments or allocate memory.

However, when we are talking about data sorting it might be logical to assume that we could be talking about comparisons, assignments, or allocations. In cases where there may be ambiguity, I will indicate which type of measurement the complexity is actually referring to.

Code Samples

The code samples found in this book can be downloaded at https://bitbucket.org/syncfusion/data_structures_succinctly_part1/src.

Chapter 2 Linked List

Overview

The first data structure we will be looking at is the linked list, and with good reason. Besides being a nearly ubiquitous structure used in everything from operating systems to video games, it is also a building block with which many other data structures can be created.

In a very general sense, the purpose of a linked list is to provide a consistent mechanism to store and access an arbitrary amount of data. As its name implies, it does this by linking the data together into a list.

Before we dive into what this means, let's start by reviewing how data is stored in an array.

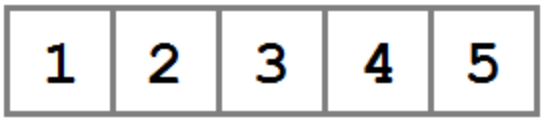

Figure 1: Integer data stored in an array

As the figure shows, array data is stored as a single contiguously allocated chunk of memory that is logically segmented. The data stored in the array is placed in one of these segments and referenced via its location, or index, in the array.

This is a good way to store data. Most programming languages make it very easy to allocate arrays and operate on their contents. Contiguous data storage provides performance benefits (namely data locality), iterating over the data is simple, and the data can be accessed directly by index (random access) in constant time.

There are times, however, when an array is not the ideal solution.

Consider a program with the following requirements:

1. Read an unknown number of integers from an input source (**`NextValue`** method) until the number 0xFFFF is encountered.
2. Pass all of the integers that have been read (in a single call) to the **`ProcessItems`** method.

Since the requirements indicate that multiple values need to be passed to the **`ProcessItems`** method in a single call, one obvious solution would involve using an array of integers. For example:

```
void LoadData()
{
```

```
    // Assume that 20 is enough to hold the values.
    int[] values = new int[20];
    for (int i = 0; i < values.Length; i++)
    {
        if (values[i] == 0xFFFF)
        {
            break;
        }

        values[i] = NextValue();
    }

    ProcessItems(values);
}

void ProcessItems(int[] values)
{
    // ... Process data.
}
```

This solution has several problems, but the most glaring is seen when more than 20 values are read. As the program is now, the values from 21 to *n* are simply ignored. This could be mitigated by allocating more than 20 values—perhaps 200 or 2000. Maybe the size could be configured by the user, or perhaps if the array became full a larger array could be allocated and all of the existing data copied into it. Ultimately these solutions create complexity and waste memory.

What we need is a collection that allows us to add an arbitrary number of integer values and then enumerate over those integers in the order that they were added. The collection should not have a fixed maximum size and random access indexing is not necessary. What we need is a linked list.

Before we go on and learn how the linked list data structure is designed and implemented, let's preview what our ultimate solution might look like.

```
static void LoadItems()
{
    LinkedList<int> list = new LinkedList<int>();
    while (true)
    {
        int value = NextValue();
        if (value != 0xFFFF)
        {
            list.Add(value);
        }
        else
        {
            break;
        }
    }

    ProcessItems(list);
}
```

```
static void ProcessItems(LinkedList<int> list)
{
        // ... Process data.
}
```

Notice that all of the problems with the array solution no longer exist. There are no longer any issues with the array not being large enough or allocating more than is necessary.

You should also notice that this solution informs some of the design decisions we will be making later, namely that the **`LinkedList`** class accepts a generic type argument and implements the **`IEnumerable`** interface.

Implementing a LinkedList Class

The Node

At the core of the linked list data structure is the **`Node`** class. A node is a container that provides the ability to both store data and connect to other nodes.

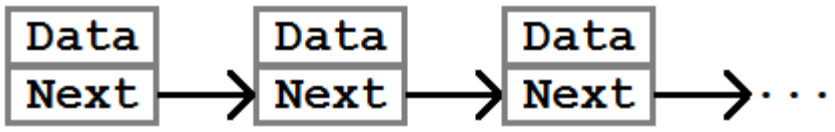

Figure 2: A linked list node contains data and a property pointing to the next node

In its simplest form, a **`Node`** class that contains integers could look like this:

```
public class Node
{
    public int Value { get; set; }
    public Node Next { get; set; }
}
```

With this we can now create a very primitive linked list. In the following example we will allocate three nodes (first, middle, and last) and then link them together into a list.

```
// +-----+------+
// |  3  | null +
// +-----+------+
Node first = new Node { Value = 3 };

// +-----+------+    +-----+------+
// |  3  | null +    |  5  | null +
// +-----+------+    +-----+------+
Node middle = new Node { Value = 5 };
```

```
// +-----+------+    +-----+------+
// |  3  |  *---+--->|  5  | null +
// +-----+------+    +-----+------+
first.Next = middle;

// +-----+------+    +-----+------+   +-----+------+
// |  3  |  *---+--->|  5  | null +   |  7  | null +
// +-----+------+    +-----+------+   +-----+------+
Node last = new Node { Value = 7 };

// +-----+------+    +-----+------+   +-----+------+
// |  3  |  *---+--->|  5  |  *---+-->|  7  | null +
// +-----+------+    +-----+------+   +-----+------+
middle.Next = last;
```

We now have a linked list that starts with the node **`first`** and ends with the node **`last`**. The **`Next`** property for the last node points to null which is the end-of-list indicator. Given this list, we can perform some basic operations. For example, the value of each node's **`Data`** property:

```
private static void PrintList(Node node)
{
    while (node != null)
    {
        Console.WriteLine(node.Value);
        node = node.Next;
    }
}
```

The **`PrintList`** method works by iterating over each node in the list, printing the value of the current node, and then moving on to the node pointed to by the **`Next`** property.

Now that we have an understanding of what a linked list node might look like, let's look at the actual **`LinkedListNode`** class.

```
public class LinkedListNode<T>
{
    /// <summary>
    /// Constructs a new node with the specified value.
    /// </summary>
    public LinkedListNode(T value)
    {
        Value = value;
    }

    /// <summary>
    /// The node value.
    /// </summary>
    public T Value { get;  internal set; }

    /// <summary>
    /// The next node in the linked list (null if last node).
```

```
    /// </summary>
    public LinkedListNode<T> Next { get;  internal set; }
}
```

The LinkedList Class

Before implementing our **`LinkedList`** class, we need to think about what we'd like to be able to do with the list.

Earlier we saw that the collection needs to support strongly typed data so we know we want to create a generic interface.

Since we're using the .NET framework to implement the list, it makes sense that we would want this class to be able to act like the other built-in collection types. The easiest way to do this is to implement the **`ICollection<T>`** interface. Notice I choose **`ICollection<T>`** and not **`IList<T>`**. This is because the **`IList<T>`** interface adds the ability to access values by index. While direct indexing is generally useful, it cannot be efficiently implemented in a linked list.

With these requirements in mind we can create a basic class stub, and then through the rest of the chapter we can fill in these methods.

```
public class LinkedList<T> :
    System.Collections.Generic.ICollection<T>
{
    public void Add(T item)
    {
        throw new System.NotImplementedException();
    }

    public void Clear()
    {
        throw new System.NotImplementedException();
    }

    public bool Contains(T item)
    {
        throw new System.NotImplementedException();
    }

    public void CopyTo(T[] array, int arrayIndex)
    {
        throw new System.NotImplementedException();
    }

    public int Count
    {
        get;
        private set;
    }

    public bool IsReadOnly
```

```
    {
        get { throw new System.NotImplementedException(); }
    }

    public bool Remove(T item)
    {
        throw new System.NotImplementedException();
    }

    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        throw new System.NotImplementedException();
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        throw new System.NotImplementedException();
    }
}
```

Add

Behavior	Adds the provided value to the end of the linked list.
Performance	*O*(1)

Adding an item to a linked list involves three steps:

1. Allocate the new **`LinkedListNode`** instance.
2. Find the last node of the existing list.
3. Point the **`Next`** property of the last node to the new node.

The key is to know which node is the last node in the list. There are two ways we can know this. The first way is to keep track of the first node (the “head” node) and walk the list until we have found the last node. This approach does not require that we keep track of the last node, which saves one reference worth of memory (whatever your platform pointer size is), but does require that we perform a traversal of the list every time a node is added. This would make **`Add`** an *O*(*n*) operation.

The second approach requires that we keep track of the last node (the “tail” node) in the list and when we add the new node we simply access our stored reference directly. This is an *O*(1) algorithm and therefore the preferred approach.

The first thing we need to do is add two private fields to the **`LinkedList`** class: references to the first (head) and last (tail) nodes.

```
private LinkedListNode<T> _head;
private LinkedListNode<T> _tail;
```

Next we need to add the method that performs the three steps.

```
public void Add(T value)
{
    LinkedListNode<T> node = new LinkedListNode<T>(value);

    if (_head == null)
    {
        _head = node;
        _tail = node;
    }
    else
    {
        _tail.Next = node;
        _tail = node;
    }

    Count++;
}
```

First, it allocates the new **`LinkedListNode`** instance. Next, it checks whether the list is empty. If the list is empty, the new node is added simply by assigning the **`_head`** and **`_tail`** references to the new node. The new node is now both the first and last node in the list. If the list is not empty, the node is added to the end of the list and the **`_tail`** reference is updated to point to the new end of the list.

The **`Count`** property is incremented when a node is added to ensure the **`ICollection<T>.Count`** property returns the accurate value.

Remove

Behavior	Removes the first node in the list whose value equals the provided value. The method returns **`true`** if a value was removed. Otherwise it returns **`false`**.
Performance	*O*(*n*)

Before talking about the **`Remove`** algorithm, let's take a look at what it is trying to accomplish. In the following figure, there are four nodes in a list. We want to remove the node with the value 3.

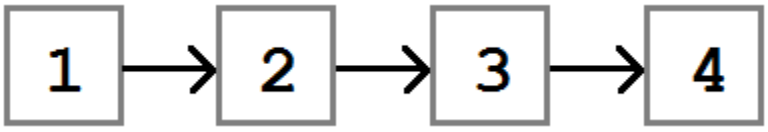

Figure 3: A linked list with four values

When the removal is done, the list will be modified such that the **`Next`** property on the node with the value 2 points to the node with the value 4.

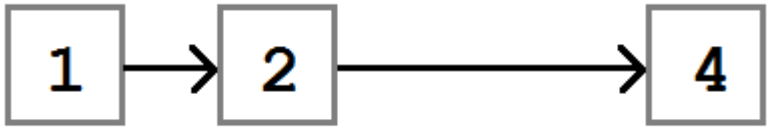

Figure 4: The linked list with the 3 node removed

The basic algorithm for node removal is:

1. Find the node to remove.
2. Update the **`Next`** property of the node that precedes the node being removed to point to the node that follows the node being removed.

As always, the devil is in the details. There are a few cases we need to be thinking about when removing a node:

- The list might be empty, or the value we are trying to remove might not be in the list. In this case the list would remain unchanged.
- The node being removed might be the only node in the list. In this case we simply set the **`_head`** and **`_tail`** fields to **`null`**.
- The node to remove might be the first node. In this case there is no preceding node, so instead we need to update the **`_head`** field to point to the new head node.
- The node might be in the middle of the list. This is the case demonstrated in Figures 3 and 4.
- The node might be the last node in the list. In this case we update the **`_tail`** field to reference the penultimate node in the list and set its **`Next`** property to **`null`**.

```
public bool Remove(T item)
{
    LinkedListNode<T> previous = null;
    LinkedListNode<T> current = _head;

    // 1: Empty list: Do nothing.
    // 2: Single node: Previous is null.
    // 3: Many nodes:
    //    a: Node to remove is the first node.
    //    b: Node to remove is the middle or last.

    while (current != null)
    {
        if (current.Value.Equals(item))
        {
            // It's a node in the middle or end.
            if (previous != null)
            {
                // Case 3b.

                // Before: Head -> 3 -> 5 -> null
```

```
                // After:  Head -> 3 ------> null
                previous.Next = current.Next;

                // It was the end, so update _tail.
                if (current.Next == null)
                {
                    _tail = previous;
                }
            }
            else
            {
                // Case 2 or 3a.

                // Before: Head -> 3 -> 5
                // After:  Head ------> 5

                // Head -> 3 -> null
                // Head ------> null
                _head = _head.Next;

                // Is the list now empty?
                if (_head == null)
                {
                    _tail = null;
                }
            }

            Count--;

            return true;
        }

        previous = current;
        current = current.Next;
    }

    return false;
}
```

The **`Count`** property is decremented when a node is removed to ensure the **`ICollection<T>.Count`** property returns the accurate value.

Contains

Behavior	Returns a Boolean that indicates whether the provided value exists within the linked list.
Performance	*O*(*n*)

The **`Contains`** method is quite simple. It looks at every node in the list, from first to last, and returns true as soon as a node matching the parameter is found. If the end of the list is reached and the node is not found, the method returns **`false`**.

```
public bool Contains(T item)
{
    LinkedListNode<T> current = _head;
    while (current != null)
    {
        if (current.Value.Equals(item))
        {
            return true;
        }

        current = current.Next;
    }

    return false;
}
```

GetEnumerator

Behavior	Returns an **`IEnumerator<T>`** instance that allows enumerating the linked list values from first to last.
Performance	Returning the enumerator instance is an *O*(1) operation. Enumerating every item is an *O*(*n*) operation.

`GetEnumerator` is implemented by enumerating the list from the first to last node and uses the C# **`yield`** keyword to return the current node's value to the caller.

Notice that the **`LinkedList`** implements the iteration behavior in the **`IEnumerable<T>`** version of the **`GetEnumerator`** method and defers to this behavior in the **`IEnumerable`** version.

```
IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
    LinkedListNode<T> current = _head;
    while (current != null)
    {
        yield return current.Value;
        current = current.Next;
    }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return ((IEnumerable<T>)this).GetEnumerator();
}
```

Clear

Behavior	Removes all the items from the list.
Performance	*O*(1)

The **`Clear`** method simply sets the **`_head`** and **`_tail`** fields to **`null`** to clear the list. Because .NET is a garbage collected language, the nodes do not need to be explicitly removed. It is the responsibility of the caller, not the linked list, to ensure that if the nodes contain **`IDisposable`** references they are properly disposed of.

```
public void Clear()
{
    _head = null;
    _tail = null;
    Count = 0;
}
```

CopyTo

Behavior	Copies the contents of the linked list from start to finish into the provided array, starting at the specified array index.
Performance	*O*(*n*)

The **`CopyTo`** method simply iterates over the list items and uses simple assignment to copy the items to the array. It is the caller's responsibility to ensure that the target array contains the appropriate free space to accommodate all the items in the list.

```
public void CopyTo(T[] array, int arrayIndex)
{
    LinkedListNode<T> current = _head;
    while (current != null)
    {
        array[arrayIndex++] = current.Value;
        current = current.Next;
    }
}
```

Count

Behavior	Returns an integer indicating the number of items currently in the list. When the list is empty, the value returned is **`0`**.
Performance	*O*(1)

`Count` is simply an automatically implemented property with a public getter and private setter. The real behavior happens in the **`Add`**, **`Remove`**, and **`Clear`** methods.

```
public int Count
{
    get;
    private set;
}
```

IsReadOnly

Behavior	Returns **`false`** if the list is not read-only.
Performance	*O*(1)

```
public bool IsReadOnly
{
    get { return false; }
}
```

Doubly Linked List

The **`LinkedList`** class we just created is known as a singly linked list. This means that there exists only a single, unidirectional link between a node and the next node in the list. There is a common variation of the linked list which allows the caller to access the list from both ends. This variation is known as a doubly linked list.

To create a doubly linked list we will need to first modify our **`LinkedListNode`** class to have a new property named **`Previous`**. **`Previous`** will act like **`Next`**, only it will point to the previous node in the list.

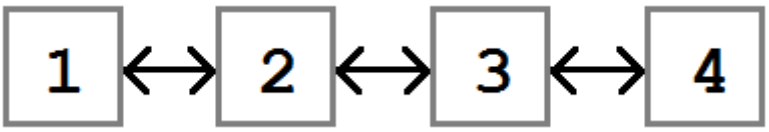

Figure 5: A doubly linked list using a Previous node property

The following sections will only describe the changes between the singly linked list and the new doubly linked list.

Node Class

The only change that will be made in the **`LinkedListNode`** class is the addition of a new property named **`Previous`** which points to the previous **`LinkedListNode`** in the linked list, or returns **`null`** if it is the first node in the list.

```
public class LinkedListNode<T>
{
    /// <summary>
    /// Constructs a new node with the specified value.
    /// </summary>
    /// <param name="value"></param>
    public LinkedListNode(T value)
    {
        Value = value;
    }

    /// <summary>
    /// The node value.
    /// </summary>
    public T Value { get;  internal set; }

    /// <summary>
    /// The next node in the linked list (null if last node).
    /// </summary>
    public LinkedListNode<T> Next { get; internal set; }

    /// <summary>
    /// The previous node in the linked list (null if first node).
    /// </summary>
    public LinkedListNode<T> Previous { get;  internal set; }
}
```

Add

While the singly linked list only added nodes to the end of the list, the doubly linked list will allow adding nodes to the start and end of the list using **`AddFirst`** and **`AddLast`**, respectively. The **`ICollection<T>.Add`** method will defer to the **`AddLast`** method to retain compatibility with the singly linked **`List`** class.

AddFirst

Behavior	Adds the provided value to the front of the list.
Performance	*O*(1)

When adding a node to the front of the list, the actions are very similar to adding to a singly linked list.

1. Set the **`Next`** property of the new node to the old head node.
2. Set the **`Previous`** property of the old head node to the new node.
3. Update the **`_tail`** field (if necessary) and increment **`Count`**.

```
public void AddFirst(T value)
{
    LinkedListNode<T> node = new LinkedListNode<T>(value);

    // Save off the head node so we don't lose it.
    LinkedListNode<T> temp = _head;

    // Point head to the new node.
    _head = node;

    // Insert the rest of the list behind head.
    _head.Next = temp;

    if (Count == 0)
    {
        // If the list was empty then head and tail should
        // both point to the new node.
        _tail = _head;
    }
    else
    {
        // Before: head -------> 5 <-> 7 -> null
        // After:  head  -> 3 <-> 5 <-> 7 -> null
        temp.Previous = _head;
    }

    Count++;
}
```

AddLast

Behavior	Adds the provided value to the end of the list.
Performance	*O*(1)

Adding a node to the end of the list is even easier than adding one to the start.

The new node is simply appended to the end of the list, updating the state of **`_tail`** and **`_head`** as appropriate, and **`Count`** is incremented.

```
public void AddLast(T value)
{
    LinkedListNode<T> node = new LinkedListNode<T>(value);

    if (Count == 0)
    {
        _head = node;
    }
    else
    {
        _tail.Next = node;

        // Before: Head -> 3 <-> 5 -> null
        // After: Head -> 3 <-> 5 <-> 7 -> null
        // 7.Previous = 5
        node.Previous = _tail;
    }

    _tail = node;
    Count++;
}
```

And as mentioned earlier, **`ICollection<T>.Add`** will now simply call **`AddLast`**.

```
public void Add(T value)
{
    AddLast(value);
}
```

Remove

Like **`Add`**, the **`Remove`** method will be extended to support removing nodes from the start or end of the list. The **`ICollection<T>.Remove`** method will continue to remove items from the start with the only change being to update the appropriate **`Previous`** property.

RemoveFirst

Behavior	Removes the first value from the list. If the list is empty, no action is taken.
Performance	*O*(1)

`RemoveFirst` updates the list by setting the linked list's **`head`** property to the second node in the list and updating its **`Previous`** property to **`null`**. This removes all references to the previous head node, removing it from the list. If the list contained only a singleton, or was empty, the list will be empty (the **`head`** and **`tail`** properties will be **`null`**).

```
public void RemoveFirst()
{
    if (Count != 0)
    {
        // Before: Head -> 3 <-> 5
        // After:  Head -------> 5

        // Head -> 3 -> null
        // Head ------> null
        _head = _head.Next;

        Count--;

        if (Count == 0)
        {
            _tail = null;
        }
        else
        {
            // 5.Previous was 3; now it is null.
            _head.Previous = null;
        }
    }
}
```

RemoveLast

Behavior	Removes the last node from the list. If the list is empty, no action is performed.
Performance	*O*(1)

`RemoveLast` works by setting the list's **`tail`** property to be the node preceding the current tail node. This removes the last node from the list. If the list was empty or had only one node, when the method returns the **`head`** and **`tail`** properties, they will both be **`null`**.

```
public void RemoveLast()
{
    if (Count != 0)
    {
        if (Count == 1)
        {
            _head = null;
            _tail = null;
        }
```

```
        else
        {
            // Before: Head --> 3 --> 5 --> 7
            //         Tail = 7
            // After:  Head --> 3 --> 5 --> null
            //         Tail = 5
            // Null out 5's Next property.
            _tail.Previous.Next = null;
            _tail = _tail.Previous;
        }

        Count--;
    }
}
```

Remove

Behavior	Removes the first node in the list whose value equals the provided value. The method returns **`true`** if a value was removed. Otherwise it returns **`false`**.
Performance	*O*(*n*)

The **`ICollection<T>.Remove`** method is nearly identical to the singly linked version except that the **`Previous`** property is now updated during the remove operation. To avoid repeated code, the method calls **`RemoveFirst`** when it is determined that the node being removed is the first node in the list.

```
public bool Remove(T item)
{
    LinkedListNode<T> previous = null;
    LinkedListNode<T> current = _head;

    // 1: Empty list: Do nothing.
    // 2: Single node: Previous is null.
    // 3: Many nodes:
    //    a: Node to remove is the first node.
    //    b: Node to remove is the middle or last.

    while (current != null)
    {
        // Head -> 3 -> 5 -> 7 -> null
        // Head -> 3 ------> 7 -> null
        if (current.Value.Equals(item))
        {
            // It's a node in the middle or end.
            if (previous != null)
            {
                // Case 3b.
```

```
                previous.Next = current.Next;

                // It was the end, so update _tail.
                if (current.Next == null)
                {
                    _tail = previous;
                }
                else
                {
                    // Before: Head -> 3 <-> 5 <-> 7 -> null
                    // After:  Head -> 3 <-------> 7 -> null

                    // previous = 3
                    // current = 5
                    // current.Next = 7
                    // So... 7.Previous = 3
                    current.Next.Previous = previous;
                }

                Count--;
            }
            else
            {
                // Case 2 or 3a.
                RemoveFirst();
            }

            return true;
        }

        previous = current;
        current = current.Next;
    }

    return false;
}
```

But Why?

We can add nodes to the front and end of the list—so what? Why do we care? As it stands right now, the doubly linked **`List`** class is no more powerful than the singly linked list. But with just one minor modification, we can open up all kinds of possible behaviors. By exposing the **`head`** and **`tail`** properties as read-only public properties, the linked list consumer will be able to implement all sorts of new behaviors.

```
public LinkedListNode<T> Head
{
    get
    {
        return _head;
    }
}

public LinkedListNode<T> Tail
```

```
{
    get
    {
        return _tail;
    }
}
```

With this simple change we can enumerate the list manually, which allows us to perform reverse (tail-to-head) enumeration and search.

For example, the following code sample shows how to use the list's **`Tail`** and **`Previous`** properties to enumerate the list in reverse and perform some processing on each node.

```
public void ProcessListBackwards()
{
    LinkedList<int> list = new LinkedList<int>();
    PopulateList(list);

    LinkedListNode<int> current = list.Tail;
    while (current != null)
    {
        ProcessNode(current);
        current = current.Previous;
    }
}
```

Additionally, the doubly linked **`List`** class allows us to easily create the Deque class, which is itself a building block for other classes. We will discuss this class later in Chapter 4.

Chapter 3 Array List

Overview

Sometimes you want the flexible sizing and ease of use of a linked list but need to have the direct (constant time) indexing of an array. In these cases, an **`ArrayList`** can provide a reasonable middle ground.

`ArrayList` is a collection that implements the **`IList<T>`** interface but is backed by an array rather than a linked list. Like a linked list, an arbitrary number of items can be added (limited only by available memory), but behave like an array in all other respects.

Class Definition

The **`ArrayList`** class implements the **`IList<T>`** interface. **`IList<T>`** provides all the methods and properties of **`ICollection<T>`** while also adding direct indexing and index-based insertion and removal. The following code sample features stubs generated by using Visual Studio 2010's **Implement Interface** command.

The following code sample also includes three additions to the generated stubs:

- An array of **`T (_items)`**. This array will hold the items in the collection.
- A default constructor initializing the array to size 0.
- A constructor accepting an integer length. This length will become the default capacity of the array. Remember that the capacity of the array and the collection **`Count`** are not the same thing. There may be scenarios when using the non-default constructor will allow the user to provide a sizing hint to the **`ArrayList`** class to minimize the number of times the internal array needs to be reallocated.

```
public class ArrayList<T> : System.Collections.Generic.IList<T>
{
    T[] _items;

    public ArrayList()
        : this(0)
    {
    }

    public ArrayList(int length)
    {
        if (length < 0)
        {
            throw new ArgumentException("length");
        }

        _items = new T[length];
```

```
    }

    public int IndexOf(T item)
    {
        throw new NotImplementedException();
    }

    public void Insert(int index, T item)
    {
        throw new NotImplementedException();
    }

    public void RemoveAt(int index)
    {
        throw new NotImplementedException();
    }

    public T this[int index]
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    public void Add(T item)
    {
        throw new NotImplementedException();
    }

    public void Clear()
    {
        throw new NotImplementedException();
    }

    public bool Contains(T item)
    {
        throw new NotImplementedException();
    }

    public void CopyTo(T[] array, int arrayIndex)
    {
        throw new NotImplementedException();
    }

    public int Count
    {
        get { throw new NotImplementedException(); }
    }

    public bool IsReadOnly
```

```
    {
        get { throw new NotImplementedException(); }
    }

    public bool Remove(T item)
    {
        throw new NotImplementedException();
    }

    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        throw new NotImplementedException();
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }
}
```

Insertion

Adding an item to an **`ArrayList`** is where the difference between the array and linked list really shows. There are two reasons for this. The first is that an **`ArrayList`** supports inserting values into the middle of the collection, whereas a linked list supports adding items to the start or end of the list. The second is that adding an item to a linked list is always an *O*(1) operation, but adding items to an **`ArrayList`** is either an *O*(1) or an *O*(*n*) operation.

Growing the Array

As items are added to the collection, eventually the internal array may become full. When this happens, the following needs to be done:

1. Allocate a larger array.
2. Copy the elements from the smaller to the larger array.
3. Update the internal array to be the larger array.

The only question we need to answer at this point is what size should the new array become? The answer to this question is defined by the **`ArrayList`** growth policy.

We'll look at two growth policies, and for each we'll look at how quickly the array grows and how it can impact performance.

Doubling (Mono and Rotor)

There are two implementations of the **`ArrayList`** class we can look at online: Mono and Rotor. Both of them use a simple algorithm that doubles the size of the array each time an allocation is needed. If the array has a size of 0, the default capacity is 16. The algorithm is:

```
size = size == 0 ? 1 : size * 2;
```

This algorithm has fewer allocations and array copies, but wastes more space on average than the Java approach. In other words, it is biased toward having more *O*(1) inserts, which should reduce the number of times the collection performs the time consuming allocation-and-copy operation. This comes at the cost of a larger average memory footprint, and, on average, more empty array slots.

Slower Growth (Java)

Java uses a similar approach but grows the array a little more slowly. The algorithm it uses to grow the array is:

```
size = (size * 3) / 2 + 1;
```

This algorithm has a slower growth curve, which means it is biased toward less memory overhead at the cost of more allocations. Let's look at the growth curve for these two algorithms for an **`ArrayList`** with more than 200,000 items added.

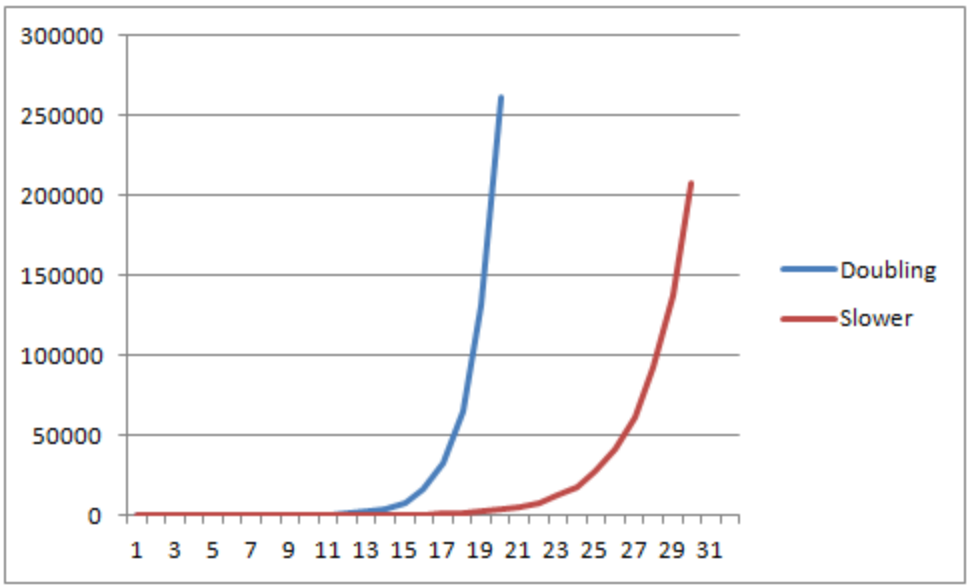

Figure 6: The growth curve for Mono/Rotor versus Java for 200,000+ items

You can see in this graph that it took 19 allocations for the doubling algorithm to cross the 200,000 boundary, whereas it took the slower (Java) algorithm 30 allocations to get to the same point.

So which one is correct? There is no right or wrong answer. Doubling performs fewer *O*(*n*) operations, but has more memory overhead on average. The slower growth algorithm performs more *O*(*n*) operations but has less memory overhead. For a general purpose collection, either approach is acceptable. Your problem domain may have specific requirements that make one more attractive, or it may require you to create another approach altogether. Regardless of the approach you take, the collection's fundamental behaviors will remain unchanged.

Our **`ArrayList`** class will be using the doubling (Mono/Rotor) approach.

```
private void GrowArray()
{
    int newLength = _items.Length == 0 ? 16 : _items.Length << 1;

    T[] newArray = new T[newLength];

    _items.CopyTo(newArray, 0);

    _items = newArray;
}
```

Insert

Behavior	Adds the provided value at the specified index in the collection. If the specified index is equal to or larger than **`Count`**, an exception is thrown
Performance	*O*(*n*)

Inserting at a specific index requires shifting all of the items after the insertion point to the right by one. If the backing array is full, it will need to be grown before the shifting can be done.

In the following example, there is an array with a capacity of five items, four of which are in use. The value “3” will be inserted as the third item in the array (index 2).

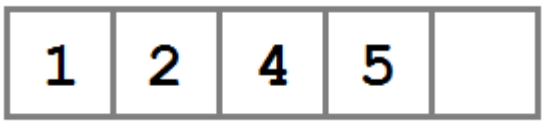

Figure 7: The array before the insert (one open slot at the end)

Figure 8: The array after shifting to the right

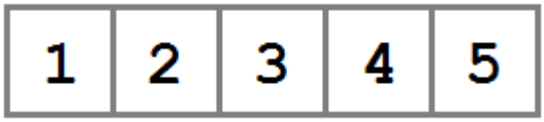

Figure 9: The array with the new item added at the open slot

```
public void Insert(int index, T item)
{
    if (index >= Count)
    {
        throw new IndexOutOfRangeException();
    }

    if (_items.Length == this.Count)
    {
        this.GrowArray();
    }

    // Shift all the items following index one slot to the right.
    Array.Copy(_items, index, _items, index + 1, Count - index);

    _items[index] = item;

    Count++;
}
```

Add

Behavior	Appends the provided value to the end of the collection.
Performance	*O*(1) when the array capacity is greater than **`Count`**; *O*(*n*) when growth is necessary.

```
public void Add(T item)
{
    if (_items.Length == Count)
    {
        GrowArray();
    }

    _items[Count++] = item;
}
```

Deletion

RemoveAt

Behavior	Removes the value at the specified index.
Performance	*O*(*n*)

Removing at an index is essentially the reverse of the **`Insert`** operation. The item is removed from the array and the array is shifted to the left.

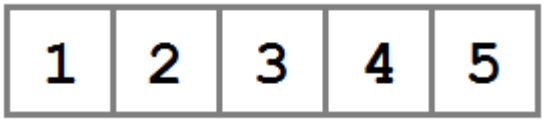

Figure 10: The array before the value 3 is removed

Figure 11: The array with the value 3 removed

Figure 12: The array shifted to the left, freeing the last slot

```
public void RemoveAt(int index)
{
    if (index >= Count)
    {
        throw new IndexOutOfRangeException();
    }

    int shiftStart = index + 1;
    if (shiftStart < Count)
    {
        // Shift all the items following index one slot to the left.
        Array.Copy(_items, shiftStart, _items, index, Count - shiftStart);
    }

    Count--;
}
```

Remove

Behavior	Removes the first item in the collection whose value matches the provided value. Returns **`true`** if a value was removed. Otherwise it returns **`false`**.
Performance	*O*(*n*)

```
public bool Remove(T item)
{
    for (int i = 0; i < Count; i++)
    {
        if (_items[i].Equals(item))
        {
            RemoveAt(i);
            return true;
        }
    }

    return false;
}
```

Indexing

IndexOf

Behavior	Returns the first index in the collection whose value equals the provided value. Returns **`-1`** if no matching value is found.
Performance	*O*(*n*)

```
public int IndexOf(T item)
{
    for (int i = 0; i < Count; i++)
    {
        if (_items[i].Equals(item))
        {
            return i;
        }
    }

    return -1;
}
```

Item

Behavior	Gets or sets the value at the specified index.
Performance	*O*(1)

```
public T this[int index]
{
    get
    {
        if(index < Count)
        {
            return _items[index];
        }

        throw new IndexOutOfRangeException();
    }
    set
    {
        if (index < Count)
        {
            _items[index] = value;
        }
        else
        {
            throw new IndexOutOfRangeException();
        }
    }
}
```

Contains

Behavior	Returns **`true`** if the provided value exists in the collection. Otherwise it returns **`false`**.
Performance	*O*(*n*)

```
public bool Contains(T item)
{
    return IndexOf(item) != -1;
}
```

Enumeration

GetEnumerator

Behavior	Returns an **`IEnumerator<T>`** instance that allows enumerating the array list values in order from first to last.
Performance	Returning the enumerator instance is an *O*(1) operation. Enumerating every item is an *O*(*n*) operation.

Note that we cannot simply defer to the **`_items`** array's **`GetEnumerator`** because that would also return the items that are not currently filled with data.

```
public System.Collections.Generic.IEnumerator<T> GetEnumerator()
{
    for (int i = 0; i < Count; i++)
    {
        yield return _items[i];
    }
}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
```

Remaining IList<T> Methods

Clear

Behavior	Removes all the items from the array list.
Performance	*O*(1)

There are two options when implementing **`Clear`**. The array can be left alone or it can be reallocated as a 0-length array. This implementation reallocates a new array with a length of 0. A larger array will be allocated when an item is added to the array using the **`Add`** or **`Insert`** methods.

```
Public void Clear()
{
    _items = new T[0];
```

```
    Count = 0;
}
```

CopyTo

Behavior	Copies the contents of the internal array from start to finish into the provided array starting at the specified array index.
Performance	*O*(*n*)

Note that the method does not simply defer to the **`_items`** array's **`CopyTo`** method. This is because we only want to copy the range from index **`0`** to **`Count`**, not the entire array capacity. Using **`Array.Copy`** allows us to specify the number of items to copy.

```
public void CopyTo(T[] array, int arrayIndex)
{
    Array.Copy(_items, 0, array, arrayIndex, Count);
}
```

Count

Behavior	Returns an integer that indicates the number of items currently in the collection. When the list is empty, the value is **`0`**.
Performance	*O*(1)

`Count` is simply an automatically implemented property with a public getter and private setter. The real behavior happens in the functions that manipulate the collection contents.

```
public int Count
{
    get;
    private set;
}
```

IsReadOnly

Behavior	Returns **`false`** because the collection is not read-only.
Performance	*O*(1)

```
public bool IsReadOnly
{
    get { return false; }
}
```

Chapter 4 Stack and Queue

Overview

So far we've looked at collections that provide very basic data storage—essentially abstractions over an array. In this chapter we're going to look at what happens when we add a few very basic behaviors that entirely change the utility of the collections.

Stack

A stack is a collection that returns objects to the caller in a Last-In-First-Out (LIFO) pattern. What this means is that the last object added to the collection will be the first object returned.

Stacks differ from list and array-like collections. They cannot be indexed directly, objects are added and removed using different methods, and their contents are more opaque than lists and arrays. What I mean by this is that while a list-based collection provides a **`Contains`** method, a stack does not. Additionally, a stack is not enumerable. To understand why this is, let's look at what a stack is and how the usage of a stack drives these differences.

One of the most common analogies for a stack is the restaurant plate stack. This is a simple spring-loaded device onto which clean plates are stacked. The spring ensures that regardless of how many plates are in the stack, the top plate can be easily accessed. Clean plates are added to the top of the stack, and when a customer removes a plate, he or she is removing the top-most plate (the most recently added plate).

We start with an empty plate rack.

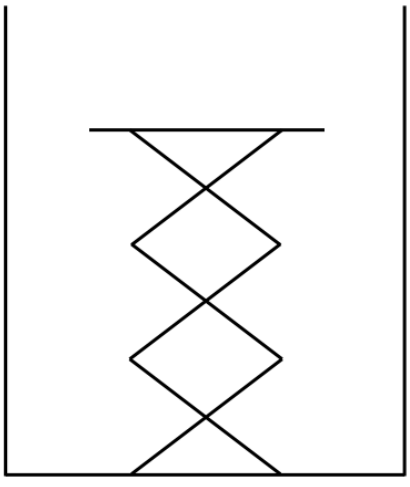

Figure 13: An empty plate stack (the spring is holding no plates)

And then we add a red, a blue, and a green plate to the rack in that order.

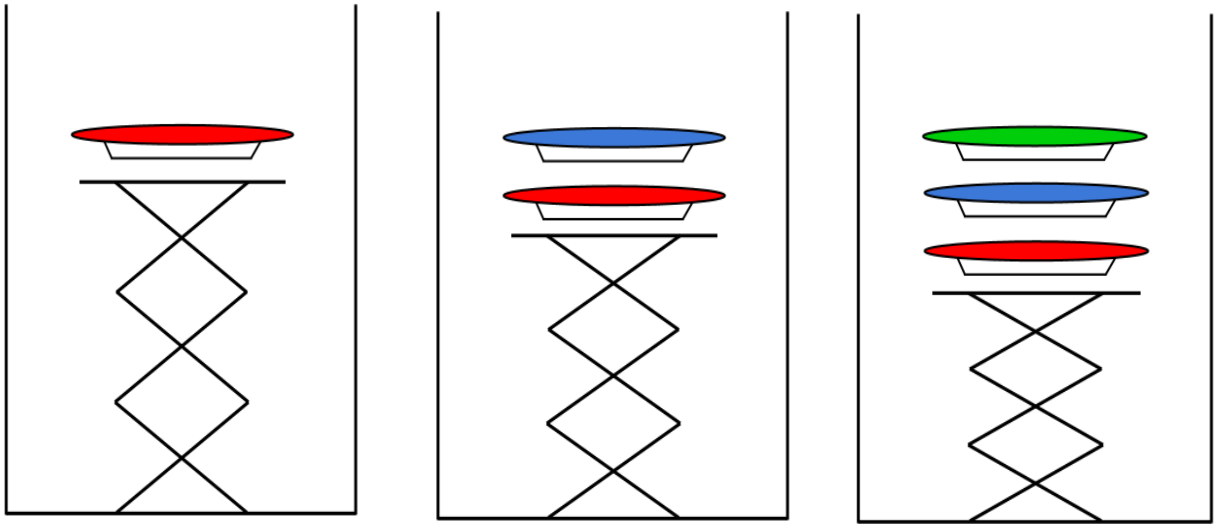

Figure 14: A red, blue, and green plate are added to the plate rack

The key point to understand here is that as new plates are added, they are added to the top of the stack. If a customer retrieves a plate, he or she will get the most recently added plate (the green plate in Figure 14). The next customer would get the blue plate, and finally the red plate would be removed.

Now that we understand how a stack works, let's define a few new terms.

When an item is added to the stack, it is "pushed" on using the **`Push`** method. When an item is removed from the stack, it is "popped" off using the **`Pop`** method. The top item in the stack, the most recently added, can be "peeked" at using the **`Peek`** method. Peeking allows you to view the item without removing it from the stack (just like the customer at the plate rack would be able to see the color of the top plate). With these terms in mind, let's look at the implementation of a **`Stack`** class.

Class Definition

The **`Stack`** class defines **`Push`**, **`Pop`**, and **`Peek`** methods, a **`Count`** property, and uses the **`LinkedList<T>`** class to store the values contained in the stack.

```
public class Stack<T>
{
    LinkedList<T> _items = new LinkedList<T>();

    public void Push(T value)
    {
        throw new NotImplementedException();
    }

    public T Pop()
    {
```

```
        throw new NotImplementedException();
    }

    public T Peek()
    {
        throw new NotImplementedException();
    }

    public int Count
    {
        get;
    }
}
```

Push

Behavior	Adds an item to the top of the stack.
Performance	*O*(1)

Since we're using a linked list as our backing store, all we need to do is add the new item to the end of the list.

```
public void Push(T value)
{
    _items.AddLast(value);
}
```

Pop

Behavior	Removes and returns the last item added to the stack. If the stack is empty, an **`InvalidOperationException`** is thrown.
Performance	*O*(1)

`Push` adds items to the back of the list, so we will "pop" them from the back. If the list is empty, an exception is thrown.

```
public T Pop()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The stack is empty");
    }
```

```
    T result = _items.Tail.Value;

    _items.RemoveLast();

    return result;
}
```

Peek

Behavior	Returns the last item added to the stack but leaves the item on the stack. If the stack is empty, an **`InvalidOperationException`** is thrown.
Performance	*O*(1)

```
public T Peek()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The stack is empty");
    }

    return _items.Tail.Value;
}
```

Count

Behavior	Returns the number of items in the stack.
Performance	*O*(1)

Since the stack is supposed to be an opaque data structure, why do we have a **`Count`** property? Knowing whether a stack is empty (Count == 0) is very useful, especially since **`Pop`** throws an exception when the stack is empty.

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

Example: RPN Calculator

The classic stack example is the Reverse Polish Notation (RPN) calculator.

RPN syntax is quite simple. It uses

`<operand> <operand> <operator>`

rather than the traditional

`<operand> <operator> <operand>.`

In other words, instead of saying “4 + 2,” we would say “4 2 +.” If you want to understand the historical significance of RPN syntax, I encourage you to head to Wikipedia or your favorite search engine.

The way RPN is evaluated, and the reason that a stack is so useful when implementing an RPN calculator, can be seen in the following algorithm (stack operations are bold):

```
for each input value

    if the value is an integer

        push the value on to the operand stack

    else if the value is an operator

        pop the left and right values from the stack

        evaluate the operator

        push the result on to the stack

pop answer from stack.
```

So given the input string “**4 2 +**,” the operations would be:

```
push (4)

push (2)

push (pop() + pop())
```

Now the stack contains a single value: **6** (the answer).

The following is a complete implementation of a simple calculator that reads an equation (e.g., “4 2 +”) from console input, splits the input at every space (e.g., [“4”, “2”, and “+”]), and performs the RPN algorithm on the input. The loop continues until the input is the word “quit”.

```
void RpnLoop()
{
    while (true)
    {
        Console.Write("> ");
        string input = Console.ReadLine();
        if (input.Trim().ToLower() == "quit")
        {
            break;
        }
        // The stack of integers not yet operated on.
        Stack<int> values = new Stack<int>();

        foreach (string token in input.Split(new char[] { ' ' }))
        {
            // If the value is an integer...
            int value;
            if (int.TryParse(token, out value))
            {
                // ... push it to the stack.
                values.Push(value);
            }
            else
            {
                // Otherwise evaluate the expression...
                int rhs = values.Pop();
                int lhs = values.Pop();

                // ... and pop the result back to the stack.
                switch (token)
                {
                    case "+":
                        values.Push(lhs + rhs);
                        break;
                    case "-":
                        values.Push(lhs - rhs);
                        break;
                    case "*":
                        values.Push(lhs * rhs);
                        break;
                    case "/":
                        values.Push(lhs / rhs);
                        break;
                    case "%":
                        values.Push(lhs % rhs);
                        break;
                    default:
                        throw new ArgumentException(
                            string.Format("Unrecognized token: {0}", token));
                }
            }
        }

        // The last item on the stack is the result.
```

```
        Console.WriteLine(values.Pop());
    }
}
```

Queue

Queues are very similar to stacks—they provide an opaque collection from which objects can be added (enqueued) or removed (dequeued) in a manner that adds value over a list-based collection.

Queues are a First-In-First-Out (FIFO) collection. This means that items are removed from the queue in the same order that they were added. You can think of a queue like a line at a store checkout counter—people enter the line and are serviced in the order they arrive.

Queues are commonly used in applications to provide a buffer to add items for future processing or to provide orderly access to a shared resource. For example, if a database is capable of handling only one connection, a queue might be used to allow threads to wait their turn (in order) to access the database.

Class Definition

The **`Queue`**, like the **`Stack`**, is backed by a **`LinkedList`**. Additionally, it provides the methods **`Enqueue`** (to add items), **`Dequeue`** (to remove items), **`Peek`**, and **`Count`**. Like **`Stack`**, it will not be treated as a general purpose collection, meaning it will not implement **`ICollection<T>`**.

```
public class Queue<T>
{
    LinkedList<T> _items = new LinkedList<T>();

    public void Enqueue(T value)
    {
        throw new NotImplementedException();
    }

    public T Dequeue()
    {
        throw new NotImplementedException();
    }

    public T Peek()
    {
        throw new NotImplementedException();
    }

    public int Count
    {
        get;
    }
}
```

Enqueue

Behavior	Adds an item to the end of the queue.
Performance	*O*(1)

This implementation adds the item to the start of the linked list. The item could just as easily be added to the end of the list. All that really matters is that items are enqueued to one end of the list and dequeued from the other (FIFO). Notice that this is the opposite of the **`Stack`** class where items are added and removed from the same end (LIFO).

```
Public void Enqueue(T value)
{
    _items.AddFirst(value);
}
```

Dequeue

Behavior	Removes and returns the oldest item from the queue. An **`InvalidOperationException`** is thrown if the queue is empty.
Performance	*O*(1)

Since **`Enqueue`** added the item to the start of the list, **`Dequeue`** must remove the item at the end of the list. If the queue contains no items, an exception is thrown.

```
public T Dequeue()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The queue is empty");
    }

    T last = _items.Tail.Value;

    _items.RemoveLast();

    return last;
}
```

Peek

Behavior	Returns the next item that would be returned if **`Dequeue`** were called. The queue is left unchanged. An **`InvalidOperationException`** is thrown if the queue is empty.
Performance	*O*(1)

```
public T Peek()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The queue is empty");
    }

    return _items.Tail.Value;
}
```

Count

Behavior	Returns the number of items currently in the queue. Returns **`0`** if the queue is empty.
Performance	*O*(1)

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

Deque (Double-Ended Queue)

A double-ended queue, or deque, extends the queue behavior by allowing items to be added or removed from both sides of the queue. This new behavior is useful in several problem domains, specifically task and thread scheduling. It is also generally useful for implementing other data structures. We'll see an example of using a deque to implement another data structure later.

Class Definition

The **Deque** class is backed by a doubly linked list. This allows us to add and remove items from the front or back of the list and access the **First** and **Last** properties. The main changes between the **Queue** class and the **Deque** class are that the **Enqueue**, **Dequeue**, and **Peek** methods have been doubled into **First** and **Last** variants.

```
public class Deque<T>
{
    LinkedList<T> _items = new LinkedList<T>();

    public void EnqueueFirst(T value)
    {
        throw new NotImplementedException();
    }

    public void EnqueueLast(T value)
    {
        throw new NotImplementedException();
    }

    public T DequeueFirst()
    {
        throw new NotImplementedException();
    }

    public T DequeueLast()
    {
        throw new NotImplementedException();
    }

    public T PeekFirst()
    {
        throw new NotImplementedException();
    }

    public T PeekLast()
    {
        throw new NotImplementedException();
    }

    public int Count
    {
        get;
    }
}
```

Enqueue

EnqueueFirst

Behavior	Adds the provided value to the head of the queue. This will be the next item dequeued by **`DequeueFirst`**.
Performance	*O*(1)

```
public void EnqueueFirst(T value)
{
    _items.AddFirst(value);
}
```

EnqueueLast

Behavior	Adds the provided value to the tail of the queue. This will be the next item dequeued by **`DequeueLast`**.
Performance	*O*(1)

```
public void EnqueueLast(T value)
{
    _items.AddLast(value);
}
```

Dequeue

DequeueFirst

Behavior	Removes and returns the first item in the deque. An **`InvalidOperationException`** is thrown if the deque is empty.
Performance	*O*(1)

```
public T DequeueFirst()
{
    if (_items.Count == 0)
    {
```

```
        throw new InvalidOperationException("DequeueFirst called when deque is empty");
    }

    T temp = _items.Head.Value;

    _items.RemoveFirst();

    return temp;
}
```

DequeueLast

Behavior	Removes and returns the last item in the deque. An **`InvalidOperationException`** is thrown if the deque is empty.
Performance	*O*(1)

```
public T DequeueLast()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("DequeueLast called when deque is empty");
    }

    T temp = _items.Tail.Value;

    _items.RemoveLast();

    return temp;
}
```

PeekFirst

Behavior	Returns the first item in the deque but leaves the collection unchanged. An **`InvalidOperationException`** is thrown if the deque is empty.
Performance	*O*(1)

```
public T PeekFirst()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("PeekFirst called when deque is empty");
```

```
    }

    return _items.Head.Value;
}
```

PeekLast

Behavior	Returns the last item in the deque but leaves the collection unchanged. An **`InvalidOperationException`** is thrown if the deque is empty.
Performance	*O*(1)

```
public T PeekLast()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("PeekLast called when deque is empty");
    }

    return _items.Tail.Value;
}
```

Count

Behavior	Returns the number of items currently in the deque, or **`0`** if the deque is empty.
Performance	*O*(1)

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

Example: Implementing a Stack

Deques are often used to implement other data structures.

We've seen a stack implemented using a **`LinkedList`**, so now let's look at one implemented using a **`Deque`**.

You might wonder why I would choose to implement a **Stack** using a **Deque** rather than a **LinkedList**. The reason is one of performance and code reusability. A linked list has the cost of per-node overhead and reduced data locality—the items are allocated in the heap and the memory locations may not be near each other, causing a larger number of cache misses and page faults at the CPU and memory hardware levels. A better performing implementation of a queue might use an array as the backing store rather than a list. This would allow for less per-node overhead and could improve performance by addressing some locality issues.

Implementing a **Stack** or **Queue** as an array is a more complex implementation, however. By implementing the **Deque** in this more complex manner and using it as the basis for other data structures, we can realize the performance benefits for all structures while only having to write the code once. This accelerates development time and reduces maintenance costs.

We will look at an example of a **Deque** as an array later in this chapter, but first let's look at an example of a **Stack** implemented using a **Deque**.

```
public class Stack<T>
{
    Deque<T> _items = new Deque<T>();

    public void Push(T value)
    {
        _items.EnqueueFirst(value);
    }

    public T Pop()
    {
        return _items.DequeueFirst();
    }

    public T Peek()
    {
        return _items.PeekFirst();
    }

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

Notice that all of the error checking is now deferred to the **Deque** and any optimization or bug fix made to the **Deque** will automatically apply to the **Stack** class. Implementing a **Queue** is just as easy and as such is left as an exercise to the reader.

Array Backing Store

As mentioned previously, there are benefits to using an array rather than a linked list as the backing store for the **`Deque<int>`** (a deque of integers). Conceptually this seems simple, but there are actually several issues that need to be addressed for this to work.

Let's look at some of these issues graphically and then see how we might deal with them. Along the way, keep in mind the growth policy issues discussed in the ArrayList chapter and that those same issues apply here.

When the collection is created, it is a 0-length array. Let's look at how some actions affect the internal array. As we go through this, notice that the green "h" and red "t" in the figures refer to "head" and "tail," respectively. The head and tail are the array indexes that indicate the first and last items in the queue. As we add and remove items, the interaction between head and tail will become clearer.

```
Deque<int> deq = new Deque<int>();
deq.EnqueueFirst(1);
```

Figure 15: Adding a value to the front of the deque

```
deq.EnqueueLast(2);
```

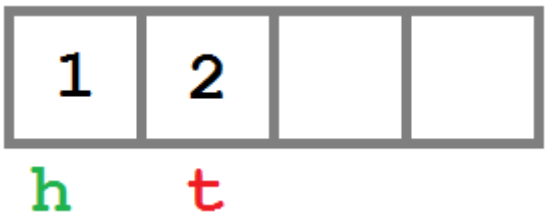

Figure 16: Adding a value to the end of the deque

```
deq.EnqueueFirst(0);
```

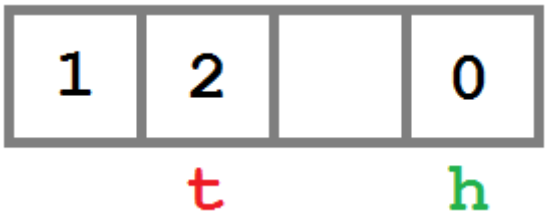

Figure 17: Adding another value to the front of the deque; the head index wraps around

Notice what has happened at this point. The head index has wrapped around to the end of the array. Now the first item in the deque, what would be returned by **`DequeueFirst`**, is the value at array index 3 (0).

```
deq.EnqueueLast(3);
```

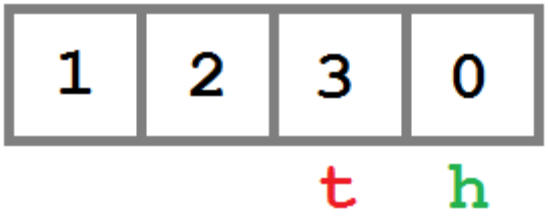

Figure 18: Adding a value to the end of the deque

At this point, the array is filled. When another item is added, the following will occur:

1. The growth policy will define the size of the new array.
2. The items will be copied from head to tail into the new array.
3. The new item will be added.
    a. **`EnqueueFirst`** – The item is added at index 0 (the copy operation leaves this open).
    b. **`EnqueueLast`** – The item is added to the end of the array.

```
deq.EnqueueLast(4);
```

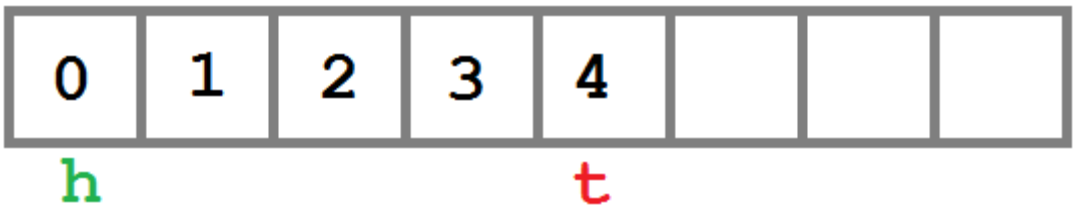

Figure 19: Adding a value to the end of the expanded deque

Now let's see what happens as items are removed from the **Deque**.

```
deq.DequeueFirst();
```

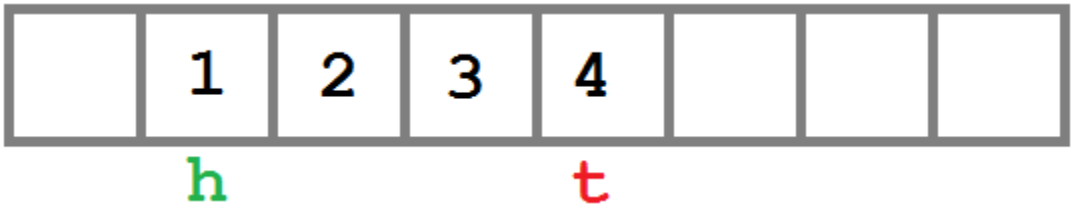

Figure 20: Removing the first item from the expanded deque

```
deq.DequeueLast();
```

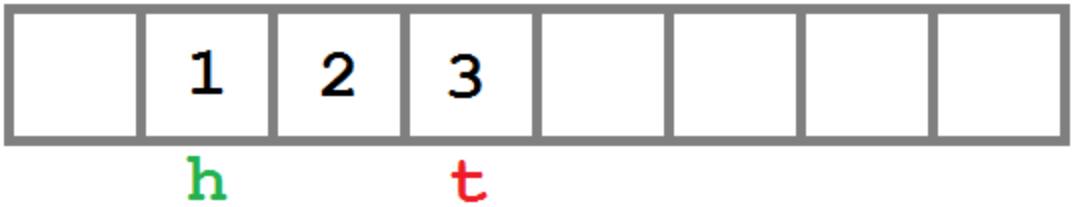

Figure 21: Removing the last item from the expanded deque

The critical point to note is that regardless of the capacity of the internal array, the logical contents of the **Deque** are the items from the head index to the tail index, taking into account the need to wrap around at the end of the array. An array that provides the behavior of wrapping around from the head to the tail is often known as a circular buffer.

With this understanding of how the array logic works, let's dive right into the code.

Class Definition

The array-based **Deque** methods and properties are the same as the list-based, so they will not be repeated here. However, the list has been replaced with an array and there are now three properties to contain the size, head, and tail information.

```
public class Deque<T>
{
    T[] _items = new T[0];

    // The number of items in the queue.
    int _size = 0;

    // The index of the first (oldest) item in the queue.
    int _head = 0;
```

```
    // The index of the last (newest) item in the queue.
    int _tail = -1;
...
}
```

Enqueue

Growth Policy

When the internal array needs to grow, the algorithm to increase the size of the array, copy the array contents, and update the internal index values needs to run. The **`Enqueue`** method performs that operation and is called by both **`EnqueueFirst`** and **`EnqueueLast`**. The **`startingIndex`** parameter is used to determine whether to leave the array slot at index 0 open (in the case of **`EnqueueFirst`**).

Pay specific attention to how the data is unwrapped in cases where the walk from head to tail requires going around the end of the array back to 0.

```
private void allocateNewArray(int startingIndex)
{
    int newLength = (_size == 0) ? 4 : _size * 2;

    T[] newArray = new T[newLength];

    if (_size > 0)
    {
        int targetIndex = startingIndex;

        // Copy the contents...
        // If the array has no wrapping, just copy the valid range.
        // Else, copy from head to end of the array and then from 0 to the tail.

        // If tail is less than head, we've wrapped.
        if (_tail < _head)
        {
            // Copy the _items[head].._items[end] -> newArray[0]..newArray[N].
            for (int index = _head; index < _items.Length; index++)
            {
                newArray[targetIndex] = _items[index];
                targetIndex++;
            }

            // Copy _items[0].._items[tail] -> newArray[N+1]..
            for (int index = 0; index <= _tail; index++)
            {
                newArray[targetIndex] = _items[index];
                targetIndex++;
            }
        }
        else
```

```
        {
            // Copy the _items[head].._items[tail] -> newArray[0]..newArray[N]
            for (int index = _head; index <= _tail; index++)
            {
                newArray[targetIndex] = _items[index];
                targetIndex++;
            }
        }

        _head = startingIndex;
        _tail = targetIndex - 1; // Compensate for the extra bump.
    }
    else
    {
        // Nothing in the array.
        _head = 0;
        _tail = -1;
    }

    _items = newArray;
}
```

EnqueueFirst

Behavior	Adds the provided value to the head of the queue. This will be the next item dequeued by **`DequeueFirst`**.
Performance	*O*(1) in most cases; *O*(*n*) when growth is necessary.

```
public void EnqueueFirst(T item)
{
    // If the array needs to grow.
    if (_items.Length == _size)
    {
        allocateNewArray(1);
    }

    // Since we know the array isn't full and _head is greater than 0,
    // we know the slot in front of head is open.
    if (_head > 0)
    {
        _head--;
    }
    else
    {
        // Otherwise we need to wrap around to the end of the array.
        _head = _items.Length - 1;
    }

    _items[_head] = item;
```

```
    _size++;
}
```

EnqueueLast

Behavior	Adds the provided value to the tail of the queue. This will be the next item dequeued by **`DequeueLast`**.
Performance	*O*(1) in most cases; *O*(*n*) when growth is necessary.

```
public void EnqueueLast(T item)
{
    // If the array needs to grow.
    if (_items.Length == _size)
    {
        allocateNewArray(0);
    }

    // Now we have a properly sized array and can focus on wrapping issues.
    // If _tail is at the end of the array we need to wrap around.
    if (_tail == _items.Length - 1)
    {
        _tail = 0;
    }
    else
    {
        _tail++;
    }

    _items[_tail] = item;
    _size++;
}
```

Dequeue

DequeueFirst

Behavior	Removes and returns the first item in the deque. An **`InvalidOperationException`** is thrown if the deque is empty.
Performance	*O*(1)

```
public T DequeueFirst()
{
    if (_size == 0)
    {
        throw new InvalidOperationException("The deque is empty");
    }

    T value = _items[_head];

    if (_head == _items.Length - 1)
    {
        // If the head is at the last index in the array, wrap it around.
        _head = 0;
    }
    else
    {
        // Move to the next slot.
        _head++;
    }

    _size--;

    return value;
}
```

DequeueLast

Behavior	Removes and returns the last item in the deque. An **`InvalidOperationException`** is thrown if the deque is empty.
Performance	*O*(1)

```
public T DequeueLast()
{
    if (_size == 0)
    {
        throw new InvalidOperationException("The deque is empty");
    }

    T value = _items[_tail];

    if (_tail == 0)
    {
        // If the tail is at the first index in the array, wrap it around.
        _tail = _items.Length - 1;
    }
    else
    {
        // Move to the previous slot.
        _tail--;
```

```
    }

    _size--;

    return value;
}
```

PeekFirst

Behavior	Returns the first item in the deque but leaves the collection unchanged. An **`InvalidOperationException`** is thrown if the deque is empty.
Performance	*O*(1)

```
public T PeekFirst()
{
    if (_size == 0)
    {
        throw new InvalidOperationException("The deque is empty");
    }

    return _items[_head];
}
```

PeekLast

Behavior	Returns the last item in the deque but leaves the collection unchanged. An **`InvalidOperationException`** is thrown if the deque is empty.
Performance	*O*(1)

```
public T PeekLast()
{
    if (_size == 0)
    {
        throw new InvalidOperationException("The deque is empty");
    }

    return _items[_tail];
}
```

Count

Behavior	Returns the number of items currently in the deque or **`0`** if the deque is empty.
Performance	*O*(1)

```
public int Count
{
    get
    {
        return _size;
    }
}
```

Chapter 5 Binary Search Tree

So far we've looked at data structures that organize data in a linear fashion. Linked lists contain data from a single starting node to a single terminating node. Arrays hold data in contiguous, one-dimensional blocks.

In this chapter, we will see how adding one more dimension will allow us to introduce a new data structure: the tree. Specifically, we will be looking at a type of tree known as a binary search tree. Binary search trees take the general tree structure and apply a set of simple rules that define the tree's structure.

Before we learn about those rules, let's learn what a tree is.

Tree Overview

A tree is a data structure where each node has 0 or more children. For example, we might have a tree like this:

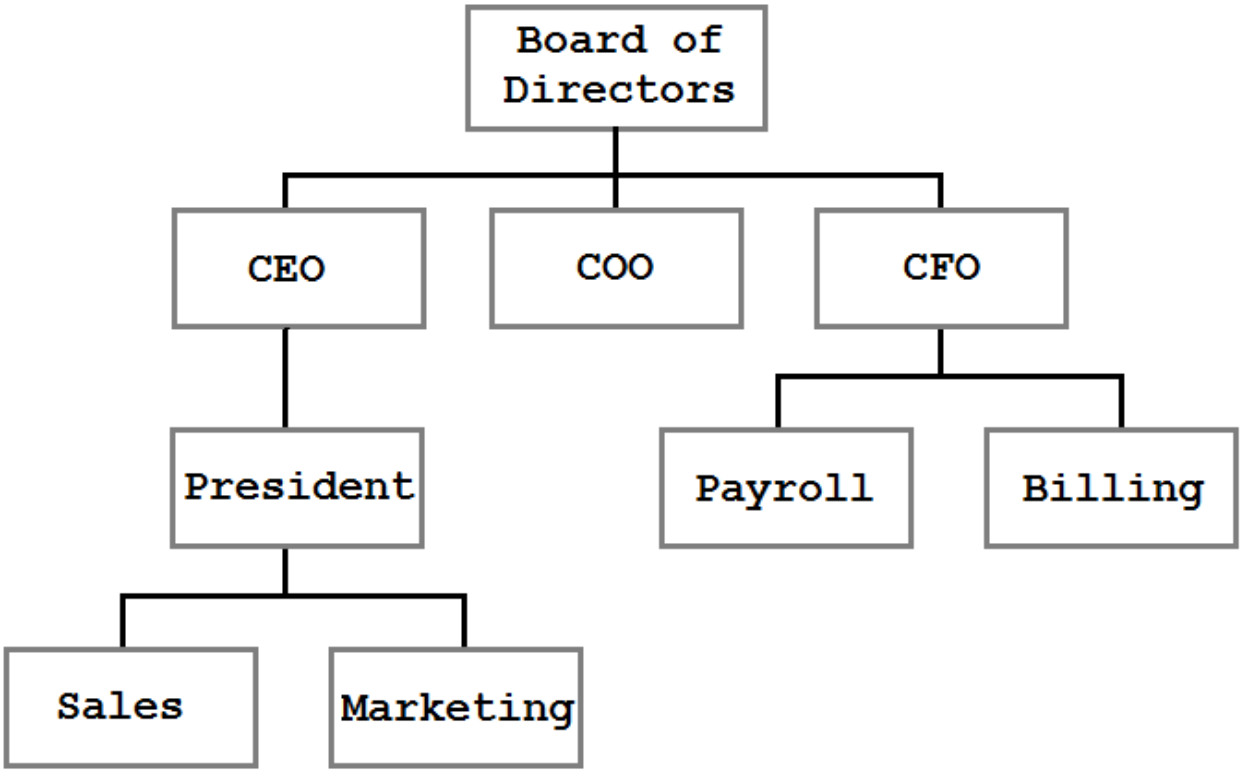

Figure 22: An organizational tree structure

In this tree, we can see the organizational structure of a business. The blocks represent people or divisions within the company, and the lines represent reporting relationships. A tree is a very efficient, logical way to present and store this information.

The tree shown in the previous figure is a general tree. It represents parent/child relationships, but there are no rules for the structure. The CEO has one direct report but could just as easily have none or twenty. In the figure, **Sales** is shown to the left of **Marketing**, but that ordering has no meaning. In fact, the only observable constraint is that each node has at most one parent (and the top-most node, the **Board of Directors**, has no parent).

Binary Search Tree Overview

A binary search tree uses the same basic structure as the general tree shown in the last figure but with the addition of a few rules. These rules are:

1. Each node can have 0, 1, or 2 children.
2. Any value less than the node's value goes to the left child (or a child of the left child).
3. Any value greater than, or equal to, the node's value goes to the right child (or a child thereof).

Let's look at a tree that is built using these rules:

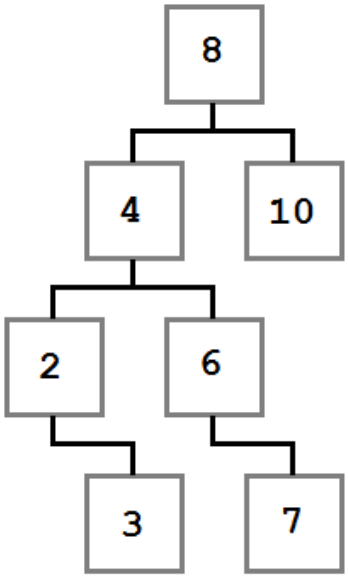

Figure 23: Binary Search Tree

Notice how the constraints we specified are enforced in the diagram. Every value to the left of the root node (8) has a value less than 8, and every value to the right is greater than or equal to the root node. This rule applies recursively at every node along the way.

With this tree in mind, let's think about the steps that went into building it. When the process started, the tree was empty and then a value, 8, was added. Because it was the first value added, it was put into the root (ultimate parent) position.

We don't know the exact order that the rest of the nodes were added, but I'll present one possible path. Values will be added using a method named **Add** that accepts the value.

```
BinaryTree<int> tree = new BinaryTree<int>();

tree.Add(8);

tree.Add(4);

tree.Add(2);

tree.Add(3);

tree.Add(10);

tree.Add(6);

tree.Add(7);
```

Let's walk through the first few items.

8 was added first and became the root. Next, 4 was added. Since 4 is less than 8, it needs to go to the left of 8 as per rule #2. Since 8 has no child on its left, 4 becomes the immediate left child of 8.

2 is added next. 2 is less than 8, so it goes to the left. There is already a node to the left of 8, so the comparison logic is performed again. 2 is less than 4, and 4 has no left child, so 2 becomes the left child of 4.

3 is added next and goes to the left of 8 and 4. When compared to the 2 node, it is larger, so 3 is added to the right of 2 as per rule #3.

This cycle of comparing values at each node and then checking each child over and over until the proper slot is found is repeated for each value until the final tree structure is created.

The Node Class

The **BinaryTreeNode** represents a single node in the tree. It contains references to the left and right children (null if there are none), the node's value, and the **IComparable.CompareTo** method which allows comparing the node values to determine if the value should go to the left or right of the current node. This is the entire **BinaryTreeNode** class—as you can see, it is very simple.

```
class BinaryTreeNode<TNode> : IComparable<TNode>
    where TNode : IComparable<TNode>
```

```
{
    public BinaryTreeNode(TNode value)
    {
        Value = value;
    }

    public BinaryTreeNode<TNode> Left { get; set; }
    public BinaryTreeNode<TNode> Right { get; set; }
    public TNode Value { get; private set; }

    /// <summary>
    /// Compares the current node to the provided value.
    /// </summary>
    /// <param name="other">The node value to compare to</param>
    /// <returns>1 if the instance value is greater than
    /// the provided value, -1 if less, or 0 if equal.</returns>
    public int CompareTo(TNode other)
    {
        return Value.CompareTo(other);
    }
}
```

The Binary Search Tree Class

The **`BinaryTree`** class provides the basic methods you need to manipulate the tree: **`Add`**, **`Remove`**, a **`Contains`** method to determine if an item exists in the tree, several traversal and enumeration methods (these are methods that allow us to enumerate the nodes in the tree in various well-defined orders), and the normal **`Count`** and **`Clear`** methods.

To initialize the tree, there is a **`BinaryTreeNode`** reference that represents the head (root) node of the tree, and there is an integer that keeps track of how many items are in the tree.

```
public class BinaryTree<T> : IEnumerable<T>
    where T : IComparable<T>
{
    private  BinaryTreeNode<T> _head;
    private int _count;

    public void Add(T value)
    {
        throw new NotImplementedException();
    }

    public bool Contains(T value)
    {
        throw new NotImplementedException();
    }

    public bool Remove(T value)
    {
        throw new NotImplementedException();
```

```
    }

    public void PreOrderTraversal(Action<T> action)
    {
        throw new NotImplementedException();
    }

    public void PostOrderTraversal(Action<T> action)
    {
        throw new NotImplementedException();
    }

    public void InOrderTraversal(Action<T> action)
    {
        throw new NotImplementedException();
    }

    public IEnumerator<T> GetEnumerator()
    {
        throw new NotImplementedException();
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }

    public void Clear()
    {
        throw new NotImplementedException();
    }

    public int Count
    {
        get;
    }
}
```

Add

Behavior	Adds the provided value to the correct location within the tree.
Performance	*O*(log *n*) on average; *O*(*n*) in the worst case.

Adding a node to the tree isn't terribly complex and is made even easier when the problem is simplified into a recursive algorithm. There are two cases that need to be considered:

- The tree is empty.
- The tree is not empty.

In the first case, we simply allocate the new node and add it to the tree. In the second case, we compare the value to the node's value. If the value we are trying to add is less than the node's value, the algorithm is repeated for the node's left child. Otherwise, it is repeated for the node's right child.

```
public void Add(T value)
{
    // Case 1: The tree is empty. Allocate the head.
    if (_head == null)
    {
        _head = new BinaryTreeNode<T>(value);
    }
    // Case 2: The tree is not empty, so recursively
    // find the right location to insert the node.
    else
    {
        AddTo(_head, value);
    }

    _count++;
}

// Recursive add algorithm.
private void AddTo(BinaryTreeNode<T> node, T value)
{
    // Case 1: Value is less than the current node value
    if (value.CompareTo(node.Value) < 0)
    {
        // If there is no left child, make this the new left,
        if (node.Left == null)
        {
            node.Left = new BinaryTreeNode<T>(value);
        }
        else
        {
            // else add it to the left node.
            AddTo(node.Left, value);
        }
    }
    // Case 2: Value is equal to or greater than the current value.
    else
    {
        // If there is no right, add it to the right,
        if (node.Right == null)
        {
            node.Right = new BinaryTreeNode<T>(value);
        }
        else
        {
            // else add it to the right node.
            AddTo(node.Right, value);
        }
    }
}
```

Remove

Behavior	Removes the first node found with the indicated value.
Performance	*O*(log *n*) on average; *O*(*n*) in the worst case.

Removing a value from the tree is a conceptually simple operation that becomes surprisingly complex in practice.

At a high level, the operation is simple:

1. Find the node to remove.
2. Remove it.

The first step is simple, and as we'll see, is accomplished using the same mechanism that the `Contains` method uses. Once the node to be removed is identified, however, the operation can take one of three paths dictated by the state of the tree around the node to be removed. The three states are described in the following three cases.

Case 1: The node to be removed has no right child.

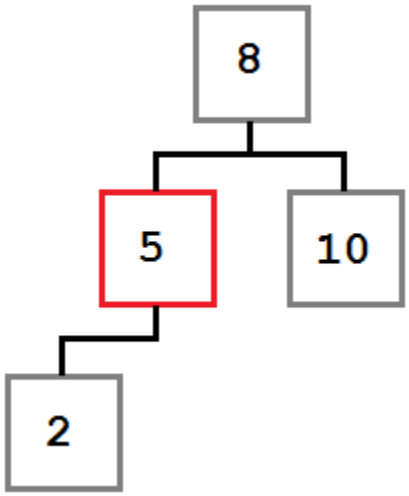

Figure 24: Case 1—The node to be removed has no right child

In this case, the removal operation can simply move the left child, if there is one, into the place of the removed node. The resulting tree would look like this:

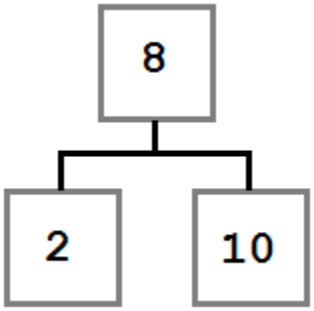

Figure 25: Case 1—Tree state after removal

Case 2: The node to be removed has a right child which, in turn, has no left child.

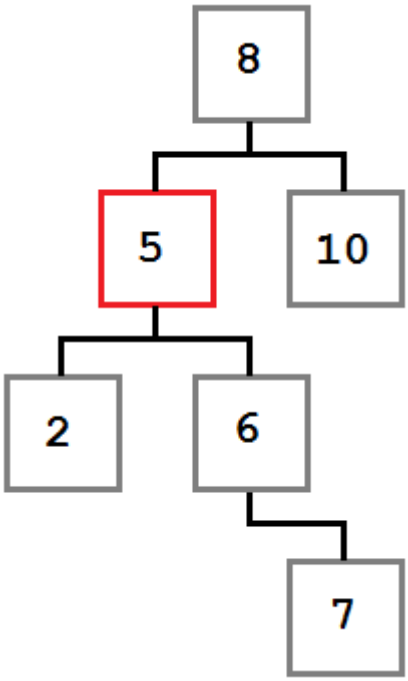

Figure 26: Case 2—The node to be removed has a right child which has no left child

In this case, we want to move the removed node's right child (6) into the place of the removed node. The resulting tree will look like this:

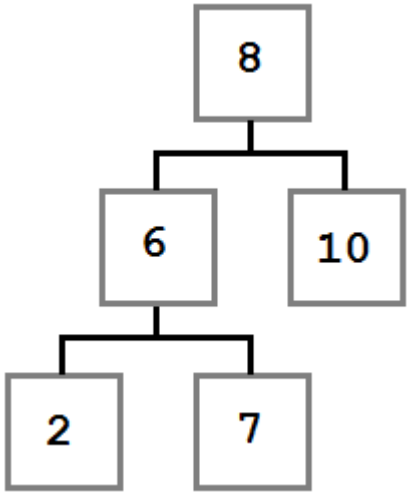

Figure 27: Case 2—Tree state after removal

Case 3: The node to be removed has a right child which, in turn, has a left child.

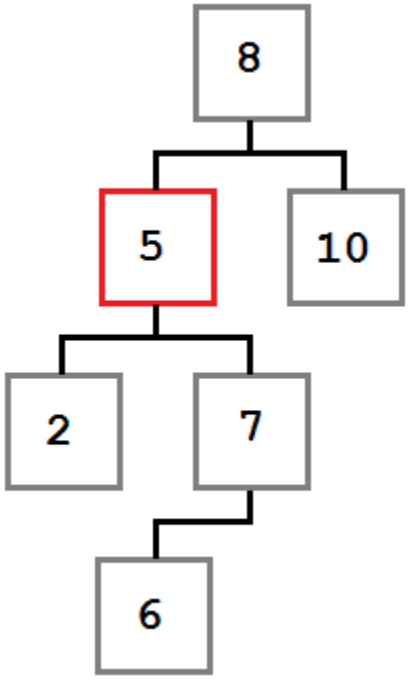

Figure 28: Case 3—The node to be removed has a right child which has a left child

In this case, the left-most child of the removed node's right child must be placed into the removed node's slot.

Let's take a minute to think about why this is true. There are two facts that we know about the sub-tree starting with the node being removed (i.e., the sub-tree whose root is the node with the value 5).

- Every value to the right of the node is greater than or equal to 5.
- The smallest value in the right sub-tree is the left-most node.

We need to place a value into the removed node's slot which is smaller than, or equal to, every node to its right. To do that, we need to get the smallest value on the right side. Therefore we need the right child's left-most node.

After the node removal, the tree will look like this:

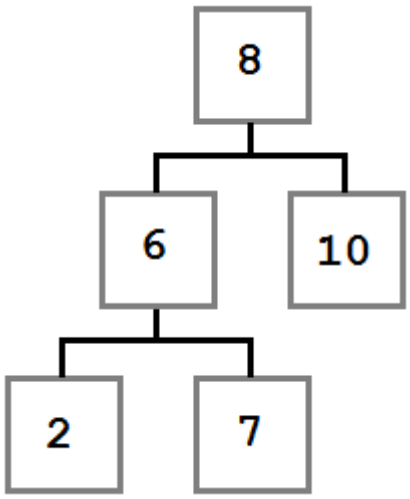

Figure 29: Case 3—Tree after node removal

Now that we understand the three remove scenarios, let's look at the code to make it happen.

One thing to note: The **`FindWithParent`** method (see the Contains section) returns the node to remove as well as the parent of the node being removed. This is done because when the node is removed, we need to update the parent's **`Left`** or **`Right`** property to point to the new node.

We could avoid doing this if all nodes kept a reference to their parent, but that would introduce per-node memory overhead and bookkeeping costs that are only needed in this one case.

```
public bool Remove(T value)
{
    BinaryTreeNode<T> current, parent;

    // Find the node to remove.
    current = FindWithParent(value, out parent);

    if (current == null)
    {
        return false;
    }
```

```
    _count--;

    // Case 1: If current has no right child, current's left replaces current.
    if (current.Right == null)
    {
        if (parent == null)
        {
            _head = current.Left;
        }
        else
        {
            int result = parent.CompareTo(current.Value);
            if (result > 0)
            {
                // If parent value is greater than current value,
                // make the current left child a left child of parent.
                parent.Left = current.Left;
            }
            else if (result < 0)
            {
                // If parent value is less than current value,
                // make the current left child a right child of parent.
                parent.Right = current.Left;
            }
        }
    }
    // Case 2: If current's right child has no left child, current's right child
    //         replaces current.
    else if (current.Right.Left == null)
    {
        current.Right.Left = current.Left;

        if (parent == null)
        {
            _head = current.Right;
        }
        else
        {
            int result = parent.CompareTo(current.Value);
            if (result > 0)
            {
                // If parent value is greater than current value,
                // make the current right child a left child of parent.
                parent.Left = current.Right;
            }
            else if (result < 0)
            {
                // If parent value is less than current value,
                // make the current 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 child.
    else
    {
        // Find the right's left-most child.
        BinaryTreeNode<T> leftmost = current.Right.Left;
        BinaryTreeNode<T> leftmostParent = current.Right;

        while (leftmost.Left != null)
        {
            leftmostParent = leftmost;
            leftmost = leftmost.Left;
        }

        // The parent's left subtree becomes the leftmost's right subtree.
        leftmostParent.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)
        {
            _head = leftmost;
        }
        else
        {
            int result = parent.CompareTo(current.Value);
            if (result > 0)
            {
                // If parent value is greater than current value,
                // make leftmost the parent's left child.
                parent.Left = leftmost;
            }
            else if (result < 0)
            {
                // If parent value is less than current value,
                // make leftmost the parent's right child.
                parent.Right = leftmost;
            }
        }
    }

    return true;
}
```

Contains

Behavior	Returns **`true`** if the tree contains the provided value. Otherwise it returns **`false`**.
Performance	*O*(log *n*) on average; *O*(*n*) in the worst case.

Contains defers to **FindWithParent**, which performs a simple tree-walking algorithm that performs the following steps, starting at the head node:

1. If the current node is **null**, return **null**.
2. If the current node value equals the sought value, return the current node.
3. If the sought value is less than the current value, set the current node to left child and go to step #1.
4. Set current node to right child and go to step #1.

Since **Contains** returns a **Boolean**, the returned value is determined by whether **FindWithParent** returns a non-null **BinaryTreeNode** (true) or a null one (false).

The **FindWithParent** method is used by the **Remove** method as well. The **out** parameter, parent, is not used by **Contains**.

```
public bool Contains(T value)
{
    // Defer to the node search helper function.
    BinaryTreeNode<T> parent;
    return FindWithParent(value, out parent) != null;
}

/// <summary>
/// Finds and returns the first node containing the specified value. If the value
/// is not found, it returns null. Also returns the parent of the found node (or null)
/// which is used in Remove.
/// </summary>
private BinaryTreeNode<T> FindWithParent(T value, out BinaryTreeNode<T> parent)
{
    // Now, try to find data in the tree.
    BinaryTreeNode<T> current = _head;
    parent = null;

    // While we don't have a match...
    while (current != null)
    {
        int result = current.CompareTo(value);

        if (result > 0)
        {
            // If the value is less than current, go left.
            parent = current;
            current = current.Left;
        }
        else if (result < 0)
        {
            // If the value is more than current, go right.
            parent = current;
            current = current.Right;
        }
        else
        {
```

```
            // We have a match!
            break;
        }
    }

    return current;
}
```

Count

Behavior	Returns the number of values in the tree (0 if empty).
Performance	*O*(1)

The **`count`** field is incremented by the **`Add`** method and decremented by the **`Remove`** method.

```
public int Count
{
    get
    {
        return _count;
    }
}
```

Clear

Behavior	Removes all the nodes from the tree.
Performance	*O*(1)

```
public void Clear()
{
    _head = null;
    _count = 0;
}
```

Traversals

Tree traversals are algorithms that allow processing each value in the tree in a well-defined order. For each of the algorithms discussed, the following tree will be used as the sample input.

The examples that follow all accept an **`Action<T>`** parameter. This parameter defines the action that will be applied to each node as it is processed by the traversal.

The **`Order`** section for each traversal will indicate the order in which the following tree would traverse.

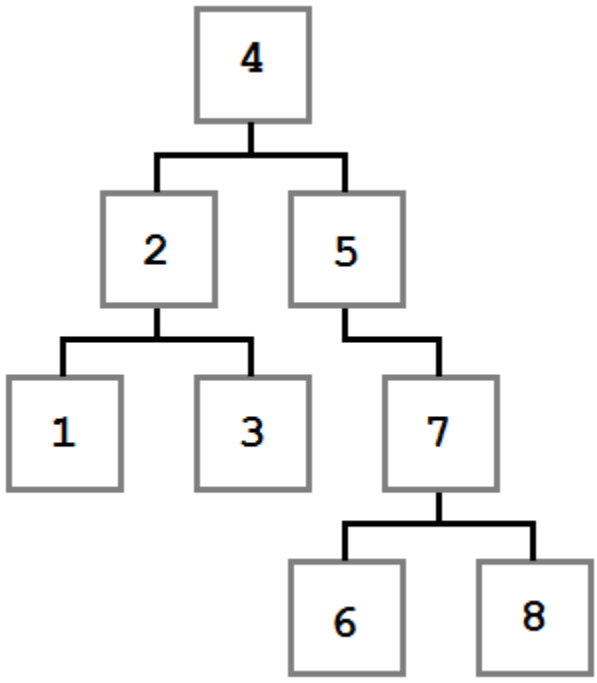

Figure 30: The sample tree for traversal ordering results

Preorder

Behavior	Performs the provided action on each value in preorder (see the description that follows).
Performance	*O*(*n*)
Order	4, 2, 1, 3, 5, 7, 6, 8

The preorder traversal processes the current node before moving to the left and then right children. Starting at the root node, 4, the action is executed with the value 4. Then the left node and all of its children are processed, followed by the right node and all of its children.

A common usage of the preorder traversal would be to create a copy of the tree that contained not just the same node values, but also the same hierarchy.

```
       public void PreOrderTraversal(Action<T> action)
{
    PreOrderTraversal(action, _head);
}

private void PreOrderTraversal(Action<T> action, BinaryTreeNode<T> node)
{
    if (node != null)
    {
        action(node.Value);
        PreOrderTraversal(action, node.Left);
        PreOrderTraversal(action, node.Right);
    }
}
```

Postorder

Behavior	Performs the provided action on each value in postorder (see the description that follows).
Performance	*O*(*n*)
Order	1, 3, 2, 6, 8, 7, 5, 4

The postorder traversal visits the left and right child of the node recursively, and then performs the action on the current node after the children are complete.

Postorder traversals are often used to delete an entire tree, such as in programming languages where each node must be freed, or to delete subtrees. This is the case because the root node is processed (deleted) last and its children are processed in a way that will minimize the amount of work the **`Remove`** algorithm needs to perform.

```
       public void PostOrderTraversal(Action<T> action)
{
    PostOrderTraversal(action, _head);
}

private void PostOrderTraversal(Action<T> action, BinaryTreeNode<T> node)
{
    if (node != null)
    {
        PostOrderTraversal(action, node.Left);
        PostOrderTraversal(action, node.Right);
        action(node.Value);
    }
}
```

Inorder

Behavior	Performs the provided action on each value in inorder (see the description that follows).
Performance	*O*(*n*)
Order	1, 2, 3, 4, 5, 6, 7, 8

Inorder traversal processes the nodes in the sort order—in the previous example, the nodes would be sorted in numerical order from smallest to largest. It does this by finding the smallest (left-most) node and then processing it before processing the larger (right) nodes.

Inorder traversals are used anytime the nodes must be processed in sort-order.

The example that follows shows two different methods of performing an inorder traversal. The first implements a recursive approach that performs a callback for each traversed node. The second removes the recursion through the use of the **`Stack`** data structure and returns an **`IEnumerator`** to allow direct enumeration.

```
Public void InOrderTraversal(Action<T> action)
{
    InOrderTraversal(action, _head);
}

private void InOrderTraversal(Action<T> action, BinaryTreeNode<T> node)
{
    if (node != null)
    {
        InOrderTraversal(action, node.Left);

        action(node.Value);

        InOrderTraversal(action, node.Right);
    }
}

public IEnumerator<T> InOrderTraversal()
{
    // This is a non-recursive algorithm using a stack to demonstrate removing
    // recursion.
    if (_head != null)
    {
        // Store the nodes we've skipped in this stack (avoids recursion).
        Stack<BinaryTreeNode<T>> stack = new Stack<BinaryTreeNode<T>>();

        BinaryTreeNode<T> current = _head;
```

```
        // When removing recursion, we need to keep track of whether
        // we should be going to the left node or the right nodes next.
        bool goLeftNext = true;

        // Start by pushing Head onto the stack.
        stack.Push(current);

        while (stack.Count > 0)
        {
            // If we're heading left...
            if (goLeftNext)
            {
                // Push everything but the left-most node to the stack.
                // We'll yield the left-most after this block.
                while (current.Left != null)
                {
                    stack.Push(current);
                    current = current.Left;
                }
            }

            // Inorder is left->yield->right.
            yield return current.Value;

            // If we can go right, do so.
            if (current.Right != null)
            {
                current = current.Right;

                // Once we've gone right once, we need to start
                // going left again.
                goLeftNext = true;
            }
            else
            {
                // If we can't go right, then we need to pop off the parent node
                // so we can process it and then go to its right node.
                current = stack.Pop();
                goLeftNext = false;
            }
        }
    }
}
```

GetEnumerator

Behavior	Returns an enumerator that enumerates using the **`InOrder`** traversal algorithm.
Performance	*O*(1) to return the enumerator. Enumerating all the items is *O*(*n*).

```
public IEnumerator<T> GetEnumerator()
{
    return InOrderTraversal();
}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
```

Chapter 6 Set

The **Set** is a collection type that implements the basic algebraic set algorithms including union, intersection, difference, and symmetric difference. Each of these algorithms will be explained in detail in their respective sections.

Conceptually, sets are collections of objects that often have some commonality. For example, we might have a set that contains positive even integers:

```
[2, 4, 6, 8, 10, ...]
```

And a set that contains positive odd integers:

```
[1, 3, 5, 7, 9, ...]
```

These two sets do not have any values in common. Now consider a third set that is all the factors of the number 100:

```
[1, 2, 4, 5, 10, 20, 25, 50, 100]
```

Given these sets, we can now answer the question, “Which factors of 100 are odd?” by looking at the set of odd integers and the set of factors of 100 and seeing which values exist in both sets. But we could also answer questions such as, “Which odd numbers are not factors of 100?” or, “Which positive numbers, even or odd, are not factors of 100?”

This may not seem very useful, but that’s because the example is somewhat contrived. Imagine if the sets were every employee at a company and every employee who had completed the mandatory annual training.

```
[All Employees]

[Employees Who Are Trained]
```

We could easily answer the question, “Which employees have not completed the mandatory training?”

We can continue to add additional sets and start to answer very complex questions such as, “Which full-time employees on the sales team who have been issued a corporate credit card have not attended the mandatory training on the new expense reporting process?”

Set Class

The **Set** class implements the **IEnumerable** interface and accepts a generic argument which should be an **IComparable** type (testing for equality is necessary for the set algorithms to function).

The members of the set will be contained in a .NET **`List`** class, but in practice, sets are often contained in tree structures such as a binary search tree. This choice of underlying container affects the complexity of the various algorithms. For example, using the **`List`**, **`Contains`** has a complexity of *O*(*n*), whereas using a tree would result in *O*(log *n*) on average.

In addition to the methods we will be implementing, the **`Set`** includes a default constructor and one that accepts an **`IEnumerable`** of items to populate the set with.

```
public class Set<T> : IEnumerable<T>
    where T: IComparable<T>
{
    private readonly List<T> _items = new List<T>();

    public Set()
    {
    }

    public Set(IEnumerable<T> items)
    {
        AddRange(items);
    }

    public void Add(T item);

    public void AddRange(IEnumerable<T> items);

    public bool Remove(T item);

    public bool Contains(T item);

    public int Count
    {
        get;
    }

    public Set<T> Union(Set<T> other);

    public Set<T> Intersection(Set<T> other);

    public Set<T> Difference(Set<T> other);

    public Set<T> SymmetricDifference(Set<T> other);

    public IEnumerator<T> GetEnumerator();

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator();
}
```

Insertion

Add

Behavior	Adds the item to the set. If the item already exists in the set, an **`InvalidOperationException`** is thrown.
Performance	*O*(*n*)

When implementing the **`Add`** algorithm, a decision needs to be made: will the set allow duplicate items or not? For example, given the following set:

```
[1, 2, 3, 4]
```

If the caller attempts to add the value 3, will the set become:

```
[1, 2, 3, 3, 4]
```

While this might be acceptable in some contexts, it's not the behavior we are going to implement. Imagine a set that contains all the students at a local college. It would not be logical to allow the same student to be added to the set twice. In fact, attempting to do so is likely an error (and will be treated as such in this implementation).

Note:** Add **uses the `Contains` method.

```
public void Add(T item)
{
    if (Contains(item))
    {
        throw new InvalidOperationException("Item already exists in Set");
    }

    _items.Add(item);
}
```

AddRange

Behavior	Adds multiple items to the set. If any member of the input enumerator exists in the set, or if there are duplicate items in the input enumerator, an **`InvalidOperationException`** will be thrown.
Performance	*O*(*mn*), where *m* is the number of items in the input enumeration and *n* is the number of items currently in the set.

```
public void AddRange(IEnumerable<T> items)
{
    foreach (T item in items)
    {
        Add(item);
    }
}
```

Remove

Behavior	Removes the specified value from the set if found, returning **`true`**. If the set does not contain the specified value, **`false`** is returned.
Performance	*O*(*n*)

```
public bool Remove(T item)
{
    return _items.Remove(item);
}
```

Contains

Behavior	Returns **`true`** if the set contains the specified value. Otherwise it returns **`false`**.
Performance	*O*(*n*)

```
public bool Contains(T item)
{
    return _items.Contains(item);
}
```

Count

Behavior	Returns the number of items in the set or **`0`** if the set is empty.
Performance	*O*(1)

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

GetEnumerator

Behavior	Returns an enumerator for all the items in the set.
Performance	*O*(1) to return the enumerator. Enumerating all the items has a complexity of *O*(*n*).

```
public IEnumerator<T> GetEnumerator()
{
    return _items.GetEnumerator();
}

System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
{
    return _items.GetEnumerator();
}
```

Algorithms

Union

Behavior	Returns a new set that is the result of the union operation of the current and input set.
Performance	*O*(*mn*), where *m* and *n* are the number of items in the provided and current sets, respectively.

The union of two sets is a set that contains all of the distinct items that exist in both sets.

For example, given two sets (each represented in red):

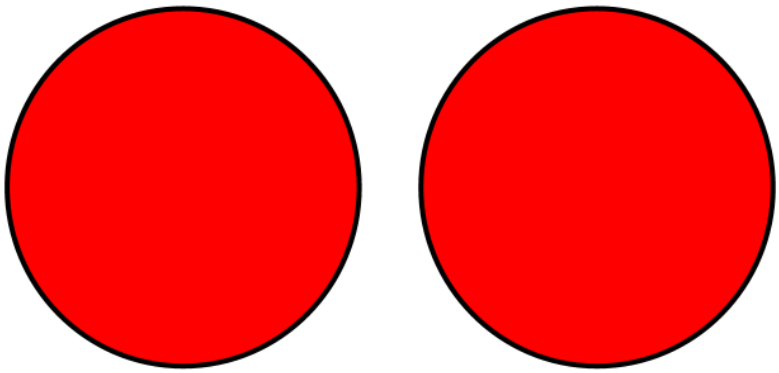

Figure 31: Two input sets before the union operation

When the union operation is performed, the output set contains all of the items in both sets. If there are any items that exist in both sets, only a single copy of each item is added to the output set.

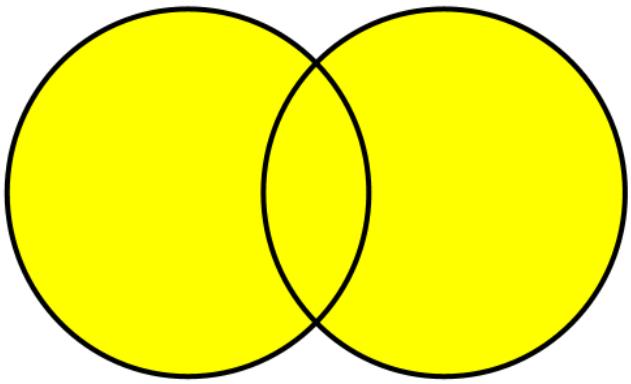

Figure 32: The output set after the union operation (returned items are yellow)

A more concrete example can be seen when we union together multiple sets of integers:

```
[1, 2, 3, 4] union [3, 4, 5, 6] = [1, 2, 3, 4, 5, 6]
```

```
public Set<T> Union(Set<T> other)
{
    Set<T> result = new Set<T>(_items);

    foreach (T item in other._items)
    {
        if (!Contains(item))
        {
            result.Add(item);
        }
    }

    return result;
}
```

Intersection

Behavior	Returns a new set that is the result of the intersection operation of the current and input sets.
Performance	*O*(*mn*), where *m* and *n* are the number of items in the provided and current sets, respectively.

Intersection is the point at which two sets “intersect”—i.e., their common members. Using the Venn diagram from the union example, the intersection of the two sets is shown here:

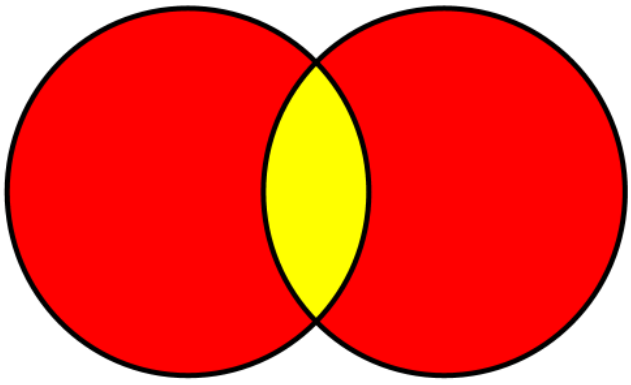

Figure 33: The intersection of the two input sets is shown in yellow

Or, using sets of integers:

```
[1, 2, 3, 4] intersect [3, 4, 5, 6] = [3, 4]
```

```
public Set<T> Intersection(Set<T> other)
{
    Set<T> result = new Set<T>();

    foreach (T item in _items)
    {
        if (other._items.Contains(item))
        {
            result.Add(item);
        }
    }

    return result;
}
```

Difference

Behavior	Returns a new set that is the result of the difference operation of the current and input sets.
Performance	*O*(*mn*), where *m* and *n* are the number of items in the provided and current sets, respectively.

The difference, or set difference, between two sets is the items that exist in the first set (the set whose **`Difference`** method is being called), but do not exist in the second set (the method's parameter). The Venn diagram for this method is shown here with the returned set in yellow:

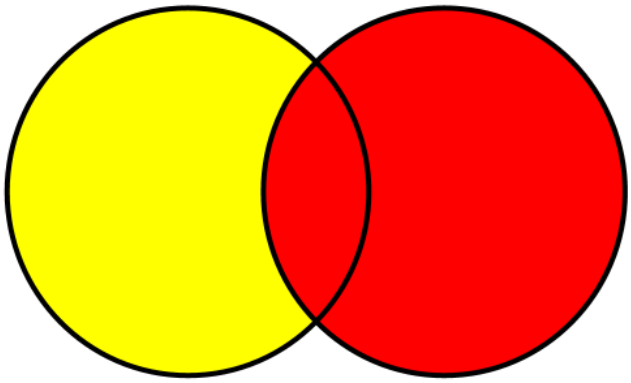

Figure 34: The set difference between two sets

Or, using sets of integers:

```
[1, 2, 3, 4] difference [3, 4, 5, 6] = [1, 2]
```

```
public Set<T> Difference(Set<T> other)
```

```
{
    Set<T> result = new Set<T>(_items);

    foreach (T item in other._items)
    {
        result.Remove(item);
    }

    return result;
}
```

Symmetric Difference

Behavior	Returns a new set that is the result of the symmetric difference operation of the current and input sets.
Performance	*O*(*mn*), where *m* and *n* are the number of items in the provided and current sets, respectively.

The symmetric difference of two sets is a set whose members are those items which exist in only one or the other set. The Venn diagram for this method is shown here with the returned set in yellow:

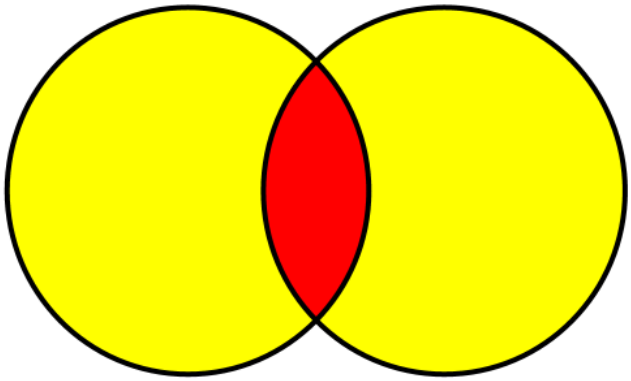

Figure 35: The symmetric difference of two sets

Or, using integer sets:

```
[1, 2, 3, 4] symmetric difference [3, 4, 5, 6] = [1, 2, 5, 6]
```

You may have noticed that this is the exact opposite of the intersection operation. With this in mind, let's see what it would take to find the symmetric difference using only the set algorithms we've already looked at.

Let's walk through what we want.

We want a set that contains all of the items from both sets except for those that exist in both. Or said another way, we want the union of both sets except for the intersection of both sets. We want the set difference between the union and the intersection of both sets.

Step by step, it looks like this:

`[1, 2, 3, 4] union [3, 4, 5, 6] = **[1, 2, 3, 4, 5, 6]**`

`[1, 2, 3, 4] intersection [3, 4, 5, 6] = **[3, 4]**`

`**[1, 2, 3, 4, 5, 6]** set difference **[3, 4]** = [1, 2, 5, 6]`

Which yields the resulting set we wanted: (`[1, 2, 5, 6]`).

```
public Set<T> SymmetricDifference(Set<T> other)
{
    Set<T> union = Union(other);
    Set<T> intersection = Intersection(other);

    return union.Difference(intersection);
}
```

IsSubset

You might be wondering why I did not add an **`IsSubset`** method. This type of method is commonly used to determine if one set is entirely contained in another set. For example, we might want to know if:

`[1, 2, 3] is subset [0, 1, 2, 3, 4, 5] = true`

Whereas

`[1, 2, 3] is subset [0, 1, 2] = false`

The reason I haven't detailed an **`IsSubset`** method is that it can be performed using existing means. For example:

`[1, 2, 3] difference [0, 1, 2, 3, 4, 5] = []`

An empty result set shows that the entire first set was contained in the second set, so we know the first set is a complete subset of the second.

Another example, using intersection:

`[1, 2, 3] intersection [0, 1, 2, 3, 4, 5] = [1, 2, 3]`

If the output set has the same number of elements as the input set, we know the input set is a subset of the second set.

In a general purpose **`Set`** class, having an **`IsSubset`** method might be useful (and could be implemented more optimally); however, I wanted to make the point that this is not necessarily a new behavior, but rather just another way of thinking about existing operations.

Chapter 7 Sorting Algorithms

In this chapter we are going to look at five algorithms used to sort data in an array. We will start with a naïve algorithm, bubble sort, and end with the most common general purpose sorting algorithm, quick sort.

With each algorithm I will explain how the sorting is done and also provide information on the best, average, and worst case complexity for both performance and memory usage.

Swap

To keep the sorting algorithm code a little easier to read, a common **`Swap`** method will be used by any sorting algorithm that needs to swap values in an array by index.

```
void Swap(T[] items, int left, int right)
{
    if (left != right)
    {
        T temp = items[left];
        items[left] = items[right];
        items[right] = temp;
    }
}
```

Bubble Sort

Behavior	Sorts the input array using the bubble sort algorithm.		
Complexity	**Best Case**	**Average Case**	**Worst Case**
Time	$O(n)$	$O(n^2)$	$O(n^2)$
Space	$O(1)$	$O(1)$	$O(1)$

Bubble sort is a naïve sorting algorithm that operates by making multiple passes through the array, each time moving the largest unsorted value to the right (end) of the array.

Consider the following unsorted array of integers:

Figure 36: Unsorted array of integers

On the first pass through the array, the values 3 and 7 are compared. Since 7 is larger than 3, no swap is performed. Next, 7 and 4 are compared. 7 is greater than 4 so the values are swapped, thus moving the 7 one step closer to the end of the array. The array now looks like this:

Figure 37: The 4 and 7 have swapped positions

This process is repeated, and the 7 eventually ends up being compared to the 8, which is greater, so no swapping can be performed, and the pass ends at the end of the array. At the end of pass 1, the array looks like this:

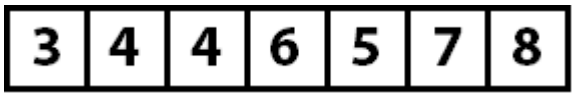

Figure 38: The array at the end of pass 1

Because at least one swap was performed, another pass will be performed. After the second pass, the 6 has moved into position.

Figure 39: The array at the end of pass 2

Again, because at least one swap was performed, another pass is performed.

The next pass, however, finds that no swaps were necessary because all of the items are in sort-order. Since no swaps were performed, the array is known to be sorted and the sorting algorithm is complete.

```
public void Sort(T[] items)
{
    bool swapped;

    do
    {
        swapped = false;
        for (int i = 1; i < items.Length; i++)
        {
            if (items[i - 1].CompareTo(items[i]) > 0)
```

```
            {
                Swap(items, i - 1, i);
                swapped = true;
            }
        }
    } while (swapped != false);
}
```

Insertion Sort

Behavior	Sorts the input array using the insertion sort algorithm.		
Complexity	**Best Case**	**Average Case**	**Worst Case**
Time	$O(n)$	$O(n^2)$	$O(n^2)$
Space	$O(1)$	$O(1)$	$O(1)$

Insertion sort works by making a single pass through the array and inserting the current value into the already sorted (beginning) portion of the array. After each index is processed, it is known that everything encountered so far is sorted and everything that follows is unknown.

Wait, what?

The important concept is that insertion sort works by sorting items as they are encountered. Since it processes the array from left to right, we know that everything to the left of the current index is sorted. This graphic demonstrates how the array becomes sorted as each index is encountered:

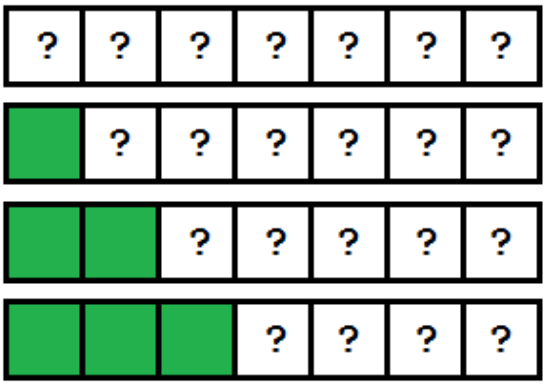

Figure 40: An array being processed by insertion sort.

As the processing continues, the array becomes more and more sorted until it is completely sorted.

Let's look at a concrete example. The following is an unsorted array that will be sorted using insertion sort.

Figure 41: Unsorted array of integers

When the sorting process begins, the sorting algorithm starts at index 0 with the value 3. Since there are no values that precede this, the array up to and including index 0 is known to be sorted.

The algorithm then moves on to the value 7. Since 7 is greater than everything in the known sorted range (which currently only includes 3), the values up to and including 7 are known to be in sort-order.

At this point the array indexes 0–1 are known to be sorted, and 2–*n* are in an unknown state.

The value at index 2 (4) is checked next. Since 4 is less than 7, it is known that 4 needs to be moved into its proper place in the sorted array area. The question now is to which index in the sorted array should the value be inserted. The method to do this is the **`FindInsertionIndex`** shown in the code sample following Figure 43. This method compares the value to be inserted (4) against the values in the sorted range, starting at index 0, until it finds the point at which the value should be inserted.

This method determines that index 1 (between 3 and 7) is the appropriate insertion point. The insertion algorithm (the **`Insert`** method in the code sample following Figure 43) then performs the insertion by removing the value to be inserted from the array and shifting all the values from the insertion point to the removed item to the right. The array now looks like this:

Figure 42: Array after first insertion algorithm

The array from index 0 to 2 is now known to be sorted, and everything from index 3 to the end is unknown. The process now starts again at index 3, which has the value 4. As the algorithm continues, the following insertions occur until the array is sorted.

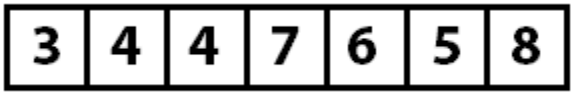

Figure 43: Array after further insertion algorithms

When there are no further insertions to be performed, or when the sorted portion of the array is the entire array, the algorithm is finished.

```
public void Sort(T[] items)
{
    int sortedRangeEndIndex = 1;

    while (sortedRangeEndIndex < items.Length)
    {
        if (items[sortedRangeEndIndex].CompareTo(items[sortedRangeEndIndex - 1]) < 0)
        {
            int insertIndex = FindInsertionIndex(items, items[sortedRangeEndIndex]);
            Insert(items, insertIndex, sortedRangeEndIndex);
        }

        sortedRangeEndIndex++;
    }
}

private int FindInsertionIndex(T[] items, T valueToInsert)
{
    for (int index = 0; index < items.Length; index++)
    {
        if (items[index].CompareTo(valueToInsert) > 0)
        {
            return index;
        }
    }

    throw new InvalidOperationException("The insertion index was not found");
}

private void Insert(T[] itemArray, int indexInsertingAt, int indexInsertingFrom)
{
    // itemArray =       0 1 2 4 5 6 3 7
    // insertingAt =     3
    // insertingFrom =   6
    // actions
    //  1: Store index at in temp     temp = 4
    //  2: Set index at to index from  -> 0 1 2 3 5 6 3 7   temp = 4
    //  3: Walking backward from index from to index at + 1.
    //     Shift values from left to right once.
    //     0 1 2 3 5 6 6 7   temp = 4
    //     0 1 2 3 5 5 6 7   temp = 4
    //  4: Write temp value to index at + 1.
    //     0 1 2 3 4 5 6 7   temp = 4

    // Step 1.
    T temp = itemArray[indexInsertingAt];

    // Step 2.

    itemArray[indexInsertingAt] = itemArray[indexInsertingFrom];

    // Step 3.
    for (int current = indexInsertingFrom; current > indexInsertingAt; current--)
```

```
    {
        itemArray[current] = itemArray[current - 1];
    }

    // Step 4.
    itemArray[indexInsertingAt + 1] = temp;
}
```

Selection Sort

Behavior	Sorts the input array using the selection sort algorithm.		
Complexity	**Best Case**	**Average Case**	**Worst Case**
Time	$O(n)$	$O(n^2)$	$O(n^2)$
Space	$O(1)$	$O(1)$	$O(1)$

Selection sort is a kind of hybrid between bubble sort and insertion sort. Like bubble sort, it processes the array by iterating from the start to the end over and over, picking one value and moving it to the right location. However, unlike bubble sort, it picks the smallest unsorted value rather than the largest. Like insertion sort, the sorted portion of the array is the start of the array, whereas with bubble sort the sorted portion is at the end.

Let's see how this works using the same unsorted array we've been using.

Figure 44: Unsorted array of integers

On the first pass, the algorithm is going to attempt to find the smallest value in the array and place it in the first index. This is performed by the **`FindIndexOfSmallestFromIndex`**, which finds the index of the smallest unsorted value starting at the provided index.

With such a small array, we can tell that the first value, 3, is the smallest value so it is already in the correct place. At this point we know that the value in array index 0 is the smallest value, and therefore is in the proper sort order. So now we can begin pass two—this time only looking at the array entries 1 to *n*-1.

The second pass will determine that 4 is the smallest value in the unsorted range, and will swap the value in the second slot with the value in the slot that 4 was held in (swapping the 4 and 7). After pass two completes, the value 4 will be inserted into its sorted position.

Figure 45: Array after second pass

The sorted range is now from index 0 to index 1, and the unsorted range is from index 2 to *n*-1. As each subsequent pass finishes, the sorted portion of the array grows larger and the unsorted portion becomes smaller. If at any point along the way no insertions are performed, the array is known to be sorted. Otherwise the process continues until the entire array is known to be sorted.

After two more passes the array is sorted:

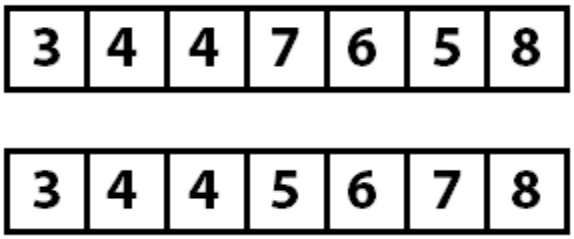

Figure 46: Sorted array

```
public void Sort(T[] items)
{
    int sortedRangeEnd = 0;

    while (sortedRangeEnd < items.Length)
    {
        int nextIndex = FindIndexOfSmallestFromIndex(items, sortedRangeEnd);
        Swap(items, sortedRangeEnd, nextIndex);

        sortedRangeEnd++;
    }
}

private int FindIndexOfSmallestFromIndex(T[] items, int sortedRangeEnd)
{
    T currentSmallest = items[sortedRangeEnd];
    int currentSmallestIndex = sortedRangeEnd;

    for (int i = sortedRangeEnd + 1; i < items.Length; i++)
    {
        if (currentSmallest.CompareTo(items[i]) > 0)
        {
            currentSmallest = items[i];
            currentSmallestIndex = i;
        }
```

```
    }

    return currentSmallestIndex;
}
```

Merge Sort

Behavior	Sorts the input array using the merge sort algorithm.		
Complexity	**Best Case**	**Average Case**	**Worst Case**
Time	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Space	$O(n)$	$O(n)$	$O(n)$

Divide and Conquer

So far we've seen algorithms that operate by linearly processing the array. These algorithms have the upside of operating with very little memory overhead but at the cost of quadratic runtime complexity. With merge sort, we are going to see our first divide and conquer algorithm.

Divide and conquer algorithms operate by breaking down large problems into smaller, more easily solvable problems. We see these types of algorithms in everyday life. For example, we use a divide and conquer algorithm when searching a phone book.

If you wanted to find the name Erin Johnson in a phone book, you would not start at *A* and flip forward page by page. Rather, you would likely open the phone book to the middle. If you opened to the M's, you would flip back a few pages, maybe a bit too far—the H's, perhaps. Then you would flip forward. And you would keep flipping back and forth in ever smaller increments until eventually you found the page you wanted (or were so close that flipping forward made sense).

How efficient are divide and conquer algorithms?

Say the phone book is 1000 pages long. When you open to the middle, you have cut the problem into two 500-page problems. Assuming you are not on the right page, you can now pick the appropriate side to search and cut the problem in half again. Now your problem space is 250 pages. As the problem is cut in half further and further, we can see that a 1000-page phone book can be searched in only 10 page turns. This is 1% of the total number of page turns that could be necessary when performing a linear search.

Merge Sort

Merge sort operates by cutting the array in half over and over again until each piece is only 1 item long. Then those items are put back together (merged) in sort-order.

Let's start with the following array:

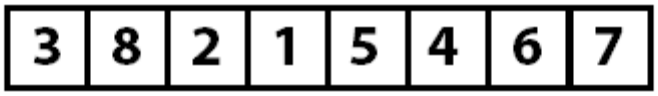

Figure 47: Unsorted array of integers

And now we cut the array in half:

Figure 48: Unsorted array cut in half

Now both of these arrays are cut in half repeatedly until each item is on its own:

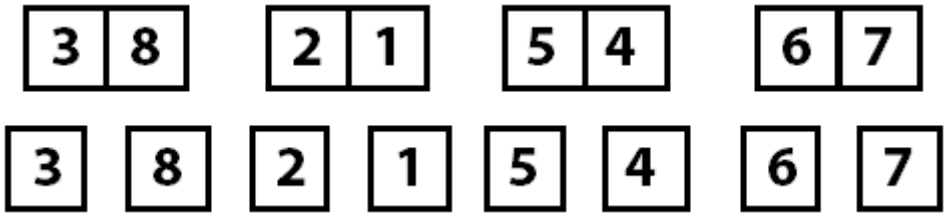

Figure 49: Unsorted array cut in half until each index is on its own

With the array now divided into the smallest possible parts, the process of merging those parts back together in sort-order occurs.

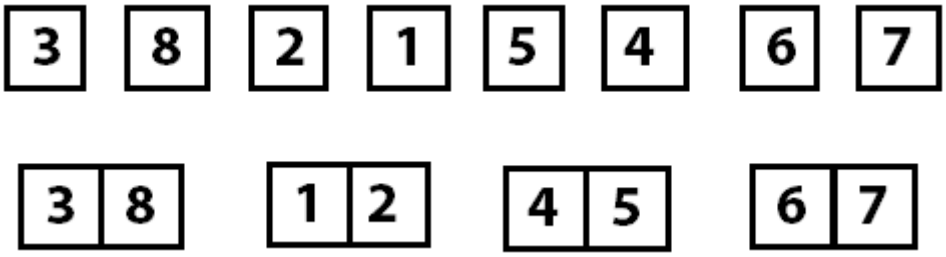

Figure 50: Array sorted into groups of two

The individual items become sorted groups of two, those groups of two merge together into sorted groups of four, and then they finally all merge back together as a final sorted array.

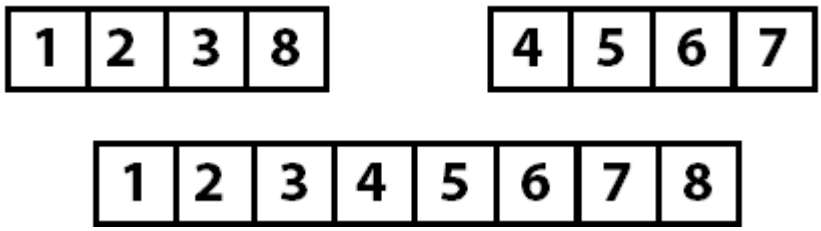

Figure 51: Array sorted into groups of four (top) and the completed sort (bottom)

Let's take a moment to think about the individual operations that we need to implement:

1. A way to split the arrays recursively. The **`Sort`** method does this.
2. A way to merge the items together in sort-order. The **`Merge`** method does this.

One performance consideration of the merge sort is that unlike the linear sorting algorithms, merge sort is going to perform its entire split and merge logic, including any memory allocations, even if the array is already in sorted order. While it has better worst-case performance than the linear sorting algorithms, its best-case performance will always be worse. This means it is not an ideal candidate when sorting data that is known to be nearly sorted; for example, when inserting data into an already sorted array.

```
public void Sort(T[] items)
{
    if (items.Length <= 1)
    {
        return;
    }

    int leftSize = items.Length / 2;
    int rightSize = items.Length - leftSize;

    T[] left = new T[leftSize];
    T[] right = new T[rightSize];

    Array.Copy(items, 0, left, 0, leftSize);
    Array.Copy(items, leftSize, right, 0, rightSize);

    Sort(left);
    Sort(right);
    Merge(items, left, right);
}

private void Merge(T[] items, T[] left, T[] right)
{
    int leftIndex = 0;
    int rightIndex = 0;
    int targetIndex = 0;

    int remaining = left.Length + right.Length;

    while(remaining > 0)
```

```
    {
        if (leftIndex >= left.Length)
        {
            items[targetIndex] = right[rightIndex++];
        }
        else if (rightIndex >= right.Length)
        {
            items[targetIndex] = left[leftIndex++];
        }
        else if (left[leftIndex].CompareTo(right[rightIndex]) < 0)
        {
            items[targetIndex] = left[leftIndex++];
        }
        else
        {
            items[targetIndex] = right[rightIndex++];
        }

        targetIndex++;
        remaining--;
    }
}
```

Quick Sort

Behavior	Sorts the input array using the quick sort algorithm.		
Complexity	**Best Case**	**Average Case**	**Worst Case**
Time	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Space	$O(1)$	$O(1)$	$O(1)$

Quick sort is another divide and conquer sorting algorithm. This one works by recursively performing the following algorithm:

1. Pick a pivot index and partition the array into two arrays. This is done using a random number in the sample code. While there are other strategies, I favored a simple approach for this sample.
2. Put all values less than the pivot value to the left of the pivot point and the values above the pivot value to the right. The pivot point is now sorted—everything to the right is larger; everything to the left is smaller. The value at the pivot point is in its correct sorted location.
3. Repeat the pivot and partition algorithm on the unsorted left and right partitions until every item is in its known sorted position.

Let's perform a quick sort on the following array:

Figure 52: Unsorted array of integers

Step one says we pick the partition point using a random index. In the sample code, this is done at this line:

```
int pivotIndex = _pivotRng.Next(left, right);
```

Figure 53: Picking a random partition index

Now that we know the partition index (4), we look at the value at that point (6) and move the values in the array so that everything less than the value is on the left side of the array and everything else (values greater than or equal) is moved to the right side of the array. Keep in mind that moving the values around might change the index the partition value is stored at (we will see that shortly).

Swapping the values is done by the **`partition`** method in the sample code following Figure 57.

Figure 54: Moving values to the left and right of the partition value

At this point, we know that 6 is in the correct spot in the array. We know this because every value to the left is less than the partition value, and everything to the right is greater than or equal to the partition value. Now we repeat this process on the two unsorted partitions of the array.

The repetition is done in the sample code by recursively calling the **`quicksort`** method with each of the array partitions. Notice that this time the left array is partitioned at index 1 with the value 5. The process of moving the values to their appropriate positions moves the value 5 to another index. I point this out to reinforce the point that you are selecting a partition value, not a partition index.

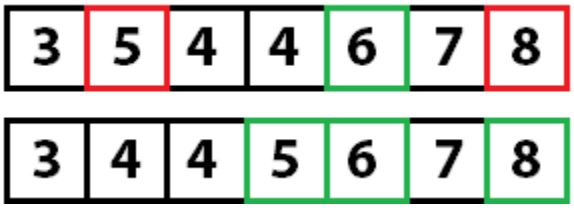

Figure 55: Repeating the pivot and partition

Quick sorting again:

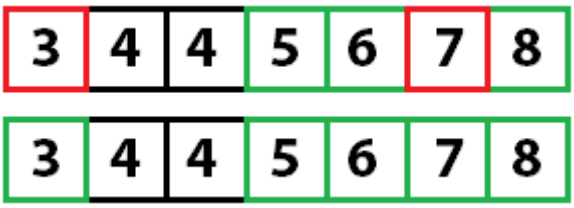

Figure 56: Repeating the pivot and partition again

And quick sorting one last time:

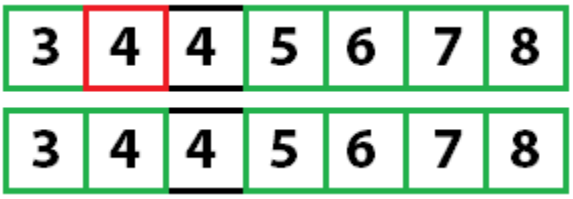

Figure 57: Repeating the pivot and partition again

With only one unsorted value left, and since we know that every other value is sorted, the array is fully sorted.

```
Random _pivotRng = new Random();

public void Sort(T[] items)
{
    quicksort(items, 0, items.Length - 1);
}

private void quicksort(T[] items, int left, int right)
{
    if (left < right)
    {
        int pivotIndex = _pivotRng.Next(left, right);
        int newPivot = partition(items, left, right, pivotIndex);

        quicksort(items, left, newPivot - 1);
        quicksort(items, newPivot + 1, right);
    }
}

private int partition(T[] items, int left, int right, int pivotIndex)
{
    T pivotValue = items[pivotIndex];

    Swap(items, pivotIndex, right);

    int storeIndex = left;

    for (int i = left; i < right; i++)
    {
```

```
        if (items[i].CompareTo(pivotValue) < 0)
        {
            Swap(items, i, storeIndex);
            storeIndex += 1;
        }
    }

    Swap(items, storeIndex, right);
    return storeIndex;
}
```