Assignment 1: CAB301

Shridhar Thorat: n10817239

May 7, 2023

Contents

1 Introduction **1**

2 Algorithm Design and Analysis **1**
2.1 NoDVDS() . . . 1
2.1.1 Pseudocode . . . 1
2.1.2 Analysis environment . . . 1
2.1.3 Empirical Time Efficiency . . . 1
2.1.4 Limitations . . . 3

3 Test Plan **4**
3.1 Movie ADT . . . 4
3.1.1 CompareTo(IMovie another) . . . 4
3.1.2 ToString() . . . 4
3.2 MovieCollection ADT . . . 5
3.2.1 IsEmpty() . . . 5
3.2.2 Insert(IMovie movie) . . . 6
3.2.3 ToArray() . . . 6
3.2.4 Search(string title) . . . 7
3.2.5 Delete(IMovie movie) . . . 7
3.2.6 Clear() . . . 9
3.2.7 NoDVDs . . . 9

4 References **9**

5 Appendix **10**
5.1 Movie ADT . . . 10
5.1.1 CompareTo(IMovie another) . . . 10
5.1.2 ToString() . . . 13
5.2 MovieCollection ADT . . . 13
5.2.1 IsEmpty() . . . 13
5.2.2 Insert(IMovie movie) . . . 14
5.2.3 ToArray() . . . 16
5.2.4 Search(string title) . . . 17
5.2.5 Delete(IMovie movie) . . . 19
5.2.6 Clear() . . . 23
5.2.7 NoDVDs() . . . 23

1 Introduction

2 Algorithm Design and Analysis

The method `NoDVDS{()}` was designed recursively to visit each node in a movie collection and add the number of dvds at that node to the total.

2.1 NoDVDS()

2.1.1 Pseudocode

ALGORITHM NoDVDs()

Input: No input, however uses a recursive *MovieCollection* method called CountNoDVDS(BTreeNode).
Output: An integer that is the total amount of DVDs in the collection.

```
1: a ← CountNoDVDs(root)
2: return a
```

ALGORITHM CountNoDVDs(BTreeNode)

Input: a BTreeNode for a Binary Search Tree with the properties; *LChild RChild* and *Movie* where *Movie* has a property called *TotalCopies* that stores the total amount of DVDs for that movie.
Output: An integer that is the total amount of DVDs in the collection.

```
1: if node is null then
2:     return 0
3: a ← BTreeNode.Movie.TotalCopies
4: a ← a + CountNoDVDs(BTreeNode.LChild)
5: a ← a + CountNoDVDs(BTreeNode.RChild)
6: return a
```

2.1.2 Analysis environment

Analysis was done using the Visual Studio IDE using the .Net 7.0 framework. The operating system was MacOS Ventura 13.2.1 (22D68) and the computer was an M1 Mac-book Pro.

2.1.3 Empirical Time Efficiency

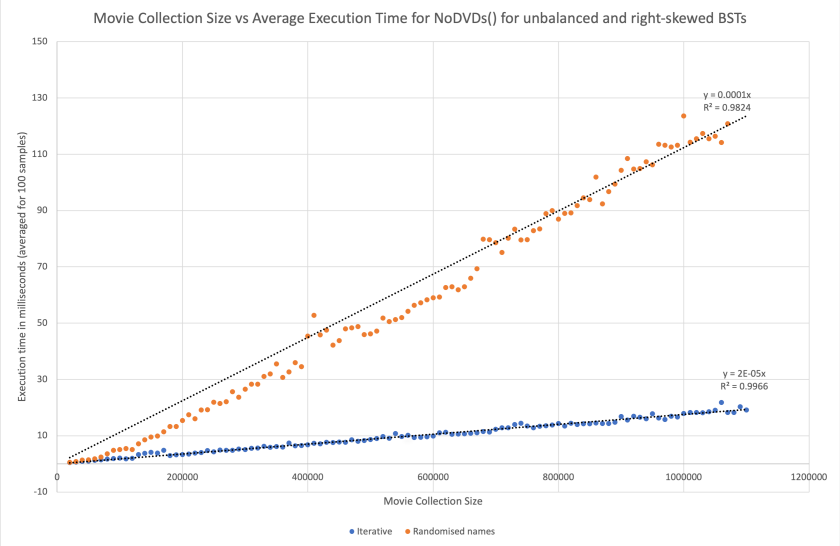
In order to analyse the time efficiency of `NoDVDs()`, a C# 'stopwatch' was used. The stopwatch was started before running `NoDVDs()` and stopped after it finished running and the elapsed milliseconds (`Elapsed.TotalMilliseconds`) was used to determine the time taken. Information was outputted into the console in the format below.

```
Average time for X nodes taking 100 samples:  Yms and has Z dvds.
```

In order to reduce variance, 100 iterations were done for each collection size X and the average time taken Y was used. These values were then plotted in a graph with collection size on the x-axis and time taken on the y-axis.

Tests were done for collection sizes from $20,000$ to $1,100,000$, varying by $10,000$. These large collections were designed using two techniques. The *Iterative* technique was by creating a right-skewed collection with titles starting at 01 going to X — the size of the collection. The other was called *Random* was done by first creating X random strings with 16 characters (arbitrarily chosen). Collections were then created by simply iterating over each string and inserting movies with those titles. Data can be found in the Microsoft Excel sheet provided with this report and the code used for analysis can be found in the `EmpiricalAnalysis.cs` file.

Before looking at ratios between $t(2n)$ and $t(n)$ for the two collection types, it is worth looking at the difference between execution time.

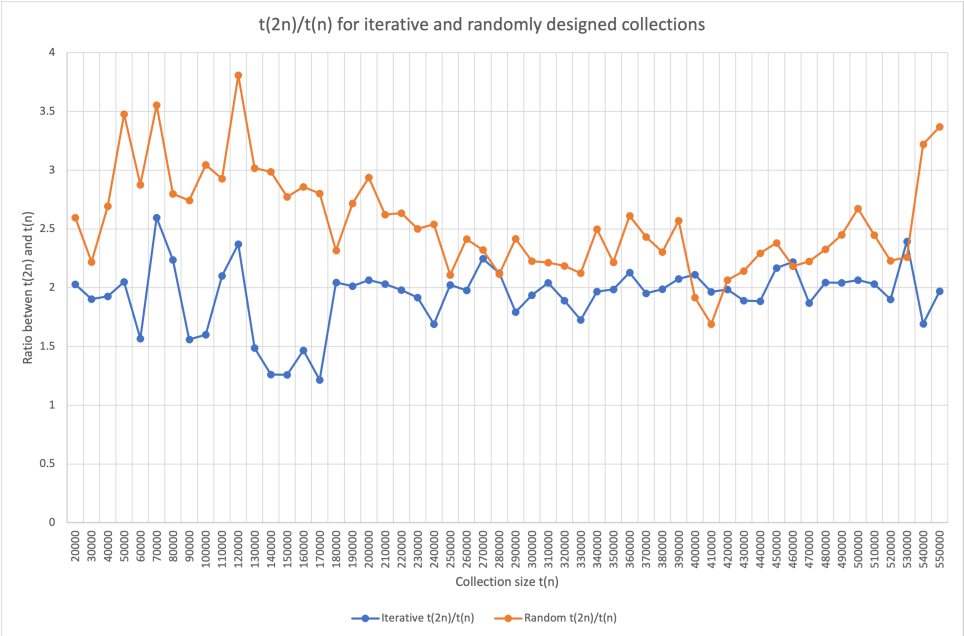


It is clearly evident that the randomised collections take much longer for larger collections. However it seems that for smaller collections; $\lessapprox 50,000$, the execution time is relatively the same as for iterative collections. It can be extrapolated that this applies to collection sizes smaller than $20,000$ that weren't tested.

The iterative collections have a much stronger indication to follow a linear trend with an R^2 value of 0.996 for a linear trend. Random collections seem to somewhat folow a linear trend but the indication is slightly weaker since it has an R^2 value of 0.982.

This graph indicates that `NoDVDs` has a time complexity of $O(n)$ for iterative collections and while the indication is slightly weaker, an $O(n)$ complexity for random names as well. However that the complexity could also be $O(nlogn)$ which is slightly worse than $O(n)$, but not as bad as $O(n^2)$.

In order to better undestand and hypothesize the likely efficiency classes for the *Iterative* and *Random* collections, the ratios between $\frac{t(2n)}{t(n)}$ for each collection size was calculated and graphed.



Theoretical Efficiencies

The variance between ratios with the high and low spikes, as well as the downward trend for the *Random* ratios may be due to the hardware (and memory) limitations of the computer. It was expected that the ratios would either stay consistent with the theoretical expectations. Theoretically the *Random* collections will either be unbalanced in the worst case and balanced in the best case. For a balanced cases, the worst efficiency class would be $O(nlogn)$ since each n node would be traversed once and each

visit may traverse the height ($logn$) of the tree. The best would In an unbalanced tree where left or right sub-trees have significant difference in height, or the entire tree is skewed, the worst-case efficiency is $O(n^2)$ since traversing nodes is proportional to the sum of the nodes; $\frac{n(n+1)}{2}$ (Goodrich et al., 2014).

Thus, it is expected that the efficiency of the *Random* tests would be $O(n^2)$ or $O(nlogn)$ at their worst and could be $O(n)$ at their best. For the *Iterative* tests, the best & worst case would be $O(n)$.

Empirical Efficiencies
It can be seen that the iteratively designed (right-skewed) collection has a ratio somewhere between 1.1 and 2.5. Ignoring the variance with large collection sizes, it seems that the time efficiency for a right-skewed tree is about $O(n)$ but in some worse seems to be $O(nlogn)$ and some best cases $O(logn)$.

The randomly designed collection has a higher range between 2 and about 2.9. This makes sense since in some of the 100 tests the tree's could have been relatively balanced, having an efficiency of close to, but above $O(n)$. In some of the worse cases at the left and right sides of the graph, the efficiency is greater than 2.5 but still less than 4. This could be due to the tree's being more unbalanced; having greater skews, resulting in the efficiency being much higher than the occurrences when they were closer to 2. Overall however, it seems that the efficiency class is $O(nlogn)$ when movies have random names.

Summary
Thus based on the empirical analysis, `NoDVDs` has an efficiency class of $O(nlogn)$. However in the case of randomised names in collections, is bounded by $O(nlogn)$ and $O(n)$, and in the case of right-skewed trees is likely bounded by $O(nlogn)$ and $O(n)$ as well but averages closer to $O(n)$.

2.1.4 Limitations

One limitation of this data is that it only provides information on the execution time of the function for a limited number of input sizes. Increasing the range of collection sizes can allow for a much better understanding of the difference between efficiency classes between randomised and right-skewed trees as well as the generalising the efficiency class for `NoDVDs`.

Additionally tests were conducted on a specific system and environment, and is likely not be representative of the performance on other systems or environments. To overcome this limitation, tests could be conducted on multiple computer systems and IDEs to get a better idea of the range of performance that can be expected.

Furthermore, the specific implementation of the function could also affect its performance. There may be ways to optimize the code or choose a different algorithm that could improve its performance. A non-recursive design using `Stack` objects could be explored and improvements to code that designs collections could be made.

Additionally, different types of binary search trees such as left-skewed and perfectly balanced trees could be used for the collection design.

Lastly, other techniques for analyzing performance, such as asymptotic analysis, could be used to provide a more theoretical understanding of the function's performance.

3 Test Plan

Testing was done on an M1 Macbook Pro (MacOS Ventura 13.3.1 [22E261]). The test suite was designed using an MSTest project provided by Microsoft in the Visual Studio IDE on the .Net 7.0 framework.

To ensure that all aspects of the software system cover functional, non-functional and boundary cases of methods, a comprehensive test plan was designed. The plan has been split into three sections that will delve deeper into what was tested each of the three ADTs and why.

3.1 Movie ADT

3.1.1 CompareTo(IMovie another)

The goal of `CompareTo` is to return -1 if this movie is less than another by dictionary order, 1 if it is greater, and 0 if it is the same.

Since the movies need to be in dictionary order; the `String.CompareOrdinal` method was used. In order to test that the method worked properly, an array of movies with titles for each ASCII character from `space` to `~` was created. Each movie in the array was in descending order of ASCII value.

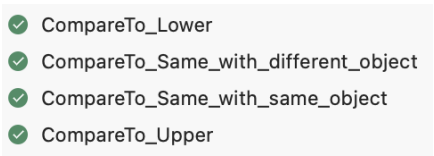
To test if -1 was outputted correctly, each `movie[i]` was *compared to* each movie after it (`movie[i+1]`) where `i` ranged from 0 to the length of the array of movies. It was expected that each comparison would output -1 since the movies were already arranged in descending order. This was called `CompareTo_Lower`.

Similarly, to test if 1 was outputted correctly, each `movie[i+1]` was *compared to* each movie before it (`movie[i]`). Since movies were already arranged in descending order, a movie with a greater array index would have a greater ASCII value. Thus it was expected that each comparison would output 1. This was called `CompareTo_Upper`

To test if a movie titled compared to itself is 0, each movie in the array was compared to itself. To ensure that the method didn't return 0 because the movie had the same reference, a different array with the the same movies (but different objects) was also used. It was expected that each movie compared to a different movie instance, but with the same titles, would also output 0. This was called `CompareTo_Same_with_same_object` and `CompareTo_Same_with_different_object` respectively.

From the below summary it can be seen that all tests passed and the method performend as required. These 4 tests can be found in the appendix for CompareTo.

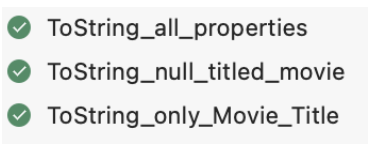
Unit testing summarisation



3.1.2 ToString()

ToString simply needed to output some of the properties of a movie. Hence there were only three tests; testing a movie will all properties defined, with only the movie title defined, and a movie with a null title (as this is the only property that can be set to null). The test data, results and testing code can be found in the appendix for ToString. From the below summary it can be seen that all tests passed and the method performend as required.

Unit testing summarisation



3.2 MovieCollection ADT

Since the methods implementated in the `MovieCollection` ADT all involve Movies and collecitons of Movies, they all used the same data; ofcourse, not all tests used all of the data. As the genre, classification, duration and available copies/total copies aren't relevant to the methods tested, they are kept constant for each different movie object. Additionally, *single collection* refers to a collection with one movie and a *large collection* is a collection with more than one movie.

Common Test Data

```
MovieCollection coll1 = new MovieCollection();
Movie mov0 = new Movie("Ar", MovieGenre.Action, MovieClassification.M, 301, 1);
Movie mov1 = new Movie("Av", MovieGenre.Action, MovieClassification.M, 301, 1);
Movie mov2 = new Movie("B",  MovieGenre.Action, MovieClassification.M, 301, 1);
Movie mov3 = new Movie("C",  MovieGenre.Action, MovieClassification.M, 301, 1);
Movie mov4 = new Movie("D",  MovieGenre.Action, MovieClassification.M, 301, 1);
Movie mov5 = new Movie("Ca", MovieGenre.Action, MovieClassification.M, 301, 1);
Movie mov6 = new Movie("E",  MovieGenre.Action, MovieClassification.M, 301, 1);
```

3.2.1 IsEmpty()

`IsEmpty` was required to simply return `true` if a colection had no movies and `false` otherwise.

1. `IsEmpty_True_empty_collection`:

   Where the method returns `true` for an empty collection

2. `IsEmpty_True_after_deleting_single_collection`:

   Where the method returns `true` after deleting the one movie in the collection.

3. `IsEmpty_True_after_deleting_large_collection`:

   Where the method returns `true` after deleting all the movies in the collection.

4. `IsEmpty_False_single_collection`:

   Where the method returns `false` for a collection with one movie.

5. `IsEmpty_False_large_collection`:

   Where the method returns `false` for a collection with lots of movie.

Technically testing the output to be `false` for a non-empty collection after all of its movie were deleted is not required since it uses another method; `Clear`. `Clear` was already tested to work correctly, hence it was used in the `IsEmpty` testing. Additionally, testing the output to be `true` when inserting multiple movies was also unnecessary since the number of movies will be greater than zero whether one or a million movies exist in a collection. However due to their trivial nature (and to prevent losing marks), they were still included.

Additionally, for each of these tests, the post-condition that the the `count` property (read outside of the *MovieCollection* class using the `Number` field) doesn't change before and after `IsEmpty` is called, was also checked.

From the below summary it can be seen that all tests passed and the method performend as required. The test data, results and testing code can be found in the appendix for IsEmpty.

Unit testing summarisation

3.2.2 Insert(IMovie movie)

The first test involved inserting a `root` to an empty collection and was done to ensure that the method correctly sets the `root` variable to the new movie. This is important as in this assignment a tree can only be traversed if the root is defined. This test was expected to return `true`;

The next two tests involed testing whether a `LChild` and `RChild` were inserted correctly and that the method returns `true`. Where a `RChild` must be greater than the root and the `LChild` is less than it in dictionary order. This is important to determine whether the collection is created correctly. It is worth mentioning, this test was done by checking whether the `Insert` method writes `'movie is a RChild'` or `'movie is a LChild'` into a line in the Visual Studio 'console' for a given `movie` parameter that is inserted successfully.

Next, it was tested whether trying to insert a movie that already exists in a single collection returns `false` and the movie is not inserted (by checking if `Number` is invariant).

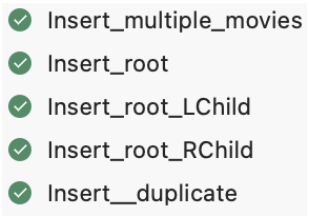
Lastly, as a sanity check, it was tested whether inserting multiple movies all outputted true. It is worth noting that testing insertion of multiple movies is not required. This is because inserting a movie into a Binary Search Tree is never done between existing movie but only as leaves. For example, if collection has a `root` as 'Batman' and it's `RChild` was 'Dungeons and Dragons'. Then inserting 'Cars' wouldn't make it the new `RChild` of 'Batman' but the `LChild` of 'Dungeons and Dragons'.

Additionally, for each of test, the post-condition was tested — that the `Number` of movies of increments by 1 if `Insert` is successful (returns `true`) and doesn't change if it is unsuccessfull (returns `false`).

From the below summary it can be seen that all tests passed and the method performend as required.

These tests were names intuitively are summarised below. Test data, results and testing code can be found in the appendix for Insert.

Unit testing summarisation



3.2.3 ToArray()

The `ToArray` method simply returns an `IMovie` array containing `Movie` objects, sorted in dictionary order. Thus, testing involved checking if the sorting was correct for a *single* collection and *large* collection, as well as for an empty collection; in which case the output was expected to be an empty `IMovie` array (i.e. `new IMovie[0]`).

The below summary shows these three all passed. Test data, results and code can be found in the appendix for ToArray.

Unit testing summarisation

3.2.4 Search(string title)

The `Search` method was designed to return a reference to an `IMovie` object if the movie is in '`this`' movie collection and `null` otherwise. A number of tests were designed to ensure these conditions were met.

To test if the correct ouput was `null`, the method was tested against an empty collection with a `non-null` **title**, a *single* and *large* collection with a `null` **title** as well as a **title**; that is not in either collection.

To test if the correct output was a reference, the method was tested using a movie **title** known to exist in a *single* and a *large* collection.

Of course, the post-condition that the `Number` of movies remains unchanged and the object passed is a reference rather tha independent copy, was also checked for each of the 7 tests. As summarised below, each of these tests passed. Test data, results and code can be found in the appendix for Search.

Unit testing summarisation

3.2.5 Delete(IMovie movie)

If the parameter `movie` is in the collection, the `Delete` method would remove it and return `true` and decrement the `Number` value by 1 and return `false` otherwise while also leaving the `Number` parameter unchanged.

Additionally, it was assumed that when deleting a node (movie) that isn't a leaf and has two children, it would be replaced by the *right most node in the left sub-tree* of that node; similar to lecture material. It is worth mentioning that `ToArray` was used to test if the structure of the collection was preserved.

For the tests where the method returned `false` the post-condition that the `Number` of movies is unchanged ($oldNumber = newNumber$) and that the collection remains unchanged (`ToArray` is the same before and after false deletion) was checked.

For the tests where `Delete` was `true`, the post-condition that the deleted movie could no longer be found in the collection (using `Search`); signifying that it was correctly set to null and no longer exists was checked. The other post-condition that `Number` decrements (i.e $oldNumber = newNumber + 1$) was checked.

The 13 tests can be summarised below. Note that for **9** to **13**, the deleted `node` is not a leaf or a root for the collection.

1. `Delete_node_not_in_single_collection`

   Where the method returns `false` when the movie is not in a collection with one movie.

2. `Delete_node_not_in_large_collection`

   Where the method returns `false` when the movie is not in a collection with multiple movies

3. `Delete_node_not_in_empty_collection.`

   Where the method returns `false` when the movie is not in an empty collection.

4. `Delete_null_node_in_single_collection`

   Where the method returns `false` when the movie is `null` for a collection with only one movie.

5. `Delete_root_single_collection`

   Where the method returns `true` when the movie is deleted from a collection with only one movie.

6. `Delete_root_has_only_LChild`

   Where the method returns `true` when the movie is deleted from a collection with only a `root` and its `LChild`. The new root is the deleted node's `LChild`; confirmed by checking if `Search`-ing for `LChild` is `true` since `Search` traverses the collection starting at the `root`.

7. `Delete_root_has_only_RChild`

   Where the method returns `true` when the movie is deleted from a collection with only a `root` and its `RChild`. The new root is the deleted node's `RChild`; confirmed by checking if `Search`-ing for `LChild` is `true` since `Search` traverses the collection starting at the `root`.

8. `Delete_root_has_LChild_and_RChild_as_leaves`

   Where the method returns `true` when the `root` is deleted from a collection with a `root` having **both** a `LChild` **and** `RChild`. The new root becomes the deleted node's `LChild`; confirmed by checking if `coll1.ToArray` is confirmed to be in dictionary order. Since `ToArray` uses an $In-Order\ traversal\ method$, if the node was replaced by its `RChild` the order would then be *D, B* rather than *B, D* which would result from `LChild` becoming the root.

9. `Delete_node_has_only_LChild_leaf`

   Where the method returns `true` when a node that only has a `LChild` leaf, is deleted. `LChild` is confirmed to replace it by checking if the `coll1.ToArray` method doesn't contains a `null` value, since if the non-existent `RChild` replaces the node, the collection will have a movie with a `null` title.

10. `Delete_node_has_only_RChild_leaf`

    Where the method returns `true` when a node that only has a `RChild` leaf, is deleted. `RChild` is confirmed to replace it by checking if the `coll1.ToArray` method doesn't contains a `null` value, since if the non-existent `LChild` replaces the node, the collection will have a movie with a `null` title.

11. `Delete_node_has_LChild_and_RChild_as_leaves`

    Where the method returns `true` when a node that has a `RChild` leaf and `LChild` leaf, is deleted. `LChild` is confirmed to replace it by checking if the `coll1.ToArray` method is in dictionary order. Since `ToArray` uses a $In-Order\ traversal\ method$, if the node was replaced by its `RChild` the order would then be *Av, D, B* which is incorrect. And if it was replaced by `LChild` the order would be *Av, B, D*, which is correct.

12. `Delete_node_has_LChild_with_LeftSkewedtree_and_RChild_as_leaf`

    Where the method returns `true` when a node that has a `RChild` leaf and and a `LChild` with a left-skewed sub-tree, is deleted, and replaced by the immediate `LChild`. The replacement is checked to be correct by confirming that `coll1.ToArray` is in dictionary order. Since `ToArray` uses an $In-Order\ traversal\ method$, if the node was replaced by its non-immediate `LChild`; `Av`, the order would then be *Ar, B, Av, Ca*— which is incorrect, rather than *Ar, Av, B, Ca* which would result from the immediate `LChild` being the replacement.

13. `Delete_node_has_LChild_with_RightSkewedtree_and_RChild_as_leaf`

    Where the method returns `true` when a node that has a `RChild` leaf and and a `LChild` with a right-skewed sub-tree, is deleted, and replaced by the *right most node in its left sub-tree*. The replacement is checked to be correct by confirming that `coll1.ToArray` is in dictionary order. Since `ToArray` uses an $In-Order\ traversal\ method$, if the node was replaced by its immediate `LChild`; `Ar`, the order would then be *B, Av, Ar, D, E*— which is incorrect, rather than *Av, Ar, B, D, E* which would result from the *right most node in its left sub-tree*; `B`, being the replacement.

As summarised below, each of these tests passed and the method was confirmed to perform as required. Test data, results and code can be found in the appendix for Delete.

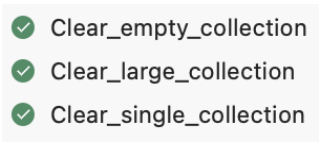
Unit testing summarisation

3.2.6 Clear()

The `Clear` method simply sets the root of a BST as null and the garbage collector for C# in Visual Studio erases all other nodes from memory. Thus three tests were only needed. Testing involved clearing an empty array, clearing a *single* collection and clearing a *large* collection while checking if the `Number` property becomes 0 for each of them.

From the below summary it can be seen that all tests passed and the method performend as required. Test data, results and testing code can be found in the appendix for Clear.

Unit testing summarisation

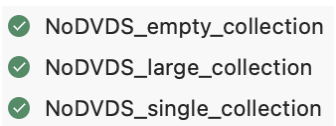


3.2.7 NoDVDs

This method simply outputted the sum of `TotalCopies` for each movie in the collection. Since `Insert` and `Delete` were already tested to perform as specified, there was no need to test `NoDVDs` before and after inserting or deleting. Hence it was simply tested for 3 static collections; an empty, *single* and *large* collection.

From the below summary it can be seen that all tests passed and the method performend as required. Test data, results and testing code can be found in the appendix for Clear.

Unit testing summarisation



4 References

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in Java (6th ed.). John Wiley & Sons.http://bedford-computing.co.uk/learning/wp-content/uploads/2016/08/Data-Structures-and-Algorithms-in-Java-6th-Edition.pdf

5 Appendix

5.1 Movie ADT

5.1.1 CompareTo(IMovie another)

Back to test plan for CompareTo.

```
public class ObjectGenerator
{
    /// <summary>
    /// Returns an IMovie array with the same length as the inputString
    /// and Movie objects with titles for each character in inputString
    /// </summary>
    /// <param name="inputString">A string with more than 0 characteres</param>
    /// <returns>An IMovie[] array of length: inputString, with Movie objects</returns>
    public static IMovie[] MovieGenerator(String inputString)
    {
        IMovie[] moviesArray = new Movie[inputString.Length];
        int i = 0;
        foreach (char c in inputString)
        {
            moviesArray[i] = new Movie(c.ToString());
            i++;
        }
        return moviesArray;
    }

    /// <summary>
    /// Returns an integer array with some length and each value being the
    /// expectation
    /// </summary>
    /// <param name="length">The length of an int[] array</param>
    /// <param name="expectation">The value for each item in the array</param>
    /// <returns></returns>
    public static int[] expectedArrayGenerator(int length, int expectation)
    {
        // Expected array with all -1 values
        int[] expectedComparisons = new int[length];
        for (int i = 0; i < length; i++)
        {
            expectedComparisons[i] = expectation;
        }
        return expectedComparisons;
    }
}
```

Figure 1: Helper class for CompareTo testing

```
[TestMethod]
public void CompareTo_Lower()
{
    // Create a list of movies with titles as characters in string str
    IMovie[] moviesArray = MovieGenerator(str);

    // Test Movie i against every other movie
    // The total number of comparisons made
    int numComparisons = moviesArray.Length * (moviesArray.Length - 1) / 2;
    int[] allComparisons = new int[numComparisons];

    // Compare every movie with the movie after it
    // i.e. every character vs every character after it
    int k = 0; // increment for index of allComparisons
    for (int i = 0; i < moviesArray.Length; i++)
    {
        for (int j = i + 1; j < moviesArray.Length; j++)
        {
            int comparisonResult = moviesArray[i].CompareTo(moviesArray[j]);
            if (comparisonResult != -1)
            {
                Console.WriteLine($"{moviesArray[i]} CompareTo {moviesArray[j]} is: {comparisonResult}");
            }
            allComparisons[k] = comparisonResult;
            k++;
        }
    }


    // Expected array with all -1 values
    int[] expectedComparisons = expectedArrayGenerator(numComparisons,-1);

    // Assert
    bool assertion = Enumerable.SequenceEqual(expectedComparisons, allComparisons);
    Assert.IsTrue(assertion);
}
```

Figure 2: Test for testing if -1 is outputted correctly for all possible comparisons

```
[TestMethod]
public void CompareTo_Upper()
{
    // Create a list of movies with titles as characters in string str
    IMovie[] moviesArray = MovieGenerator(str);

    // Test Movie i against every other movie
    // The total number of comparisons made
    int numComparisons = moviesArray.Length * (moviesArray.Length - 1) / 2;
    int[] allComparisons = new int[numComparisons];

    int k = 0; // increment for index of allComparisons
    for (int i = 0; i < moviesArray.Length; i++)
    {
        for (int j = i + 1; j < moviesArray.Length; j++)
        {
            int comparisonResult = moviesArray[j].CompareTo(moviesArray[i]);
            if (comparisonResult != 1) Console.WriteLine($"{moviesArray[j]} CompareTo {moviesArray[i]} is: {comparisonResult}")
            allComparisons[k] = comparisonResult;
            k++;
        }
    }

    // Expected array with all -1 values
    int[] expectedComparisons = expectedArrayGenerator(numComparisons, 1);

    // Assert
    bool assertion = Enumerable.SequenceEqual(expectedComparisons, allComparisons);
    Assert.IsTrue(assertion);
}
```

Figure 3: Test for testing if 1 is outputted correctly for all possible comparisons

```
[TestMethod]
public void CompareTo_Same_with_same_object()
{
    // Create a list of movies with titles as characters in string str
    IMovie[] moviesArray = MovieGenerator(str);

    int numComparisons = moviesArray.Length;
    int[] allComparisons = new int[numComparisons];

    int k = 0; // increment for index of allComparisons
    for (int i = 0; i < moviesArray.Length; i++)
    {
        int comparisonResult = moviesArray[i].CompareTo(moviesArray[i]);
        if (comparisonResult != 0) Console.WriteLine($"{moviesArray[i]} CompareTo {moviesArray[i]} is: {comparisonResult}");
        allComparisons[k] = comparisonResult;
        k++;
    }

    // Expected array with all -1 values
    int[] expectedComparisons = expectedArrayGenerator(numComparisons, 0);

    // Assert
    bool assertion = Enumerable.SequenceEqual(expectedComparisons, allComparisons);
    Assert.IsTrue(assertion);

}
```

Figure 4: Test for testing if 0 is outputted correctly for all possible comparisons

```
[TestMethod]
public void CompareTo_Same_with_different_object()
{
    // Create a list of movies with titles as characters in string str
    IMovie[] moviesArray = MovieGenerator(str);
    IMovie[] moviesArray2 = MovieGenerator(str);

    int numComparisons = moviesArray.Length;
    int[] allComparisons = new int[numComparisons];

    for (int i = 0; i < moviesArray.Length; i++)
    {
        for (int j = 0; j < moviesArray.Length; j++)
        {
            int comparisonResult = moviesArray[j].CompareTo(moviesArray2[j]);
            if (comparisonResult != 0) Console.WriteLine($"{moviesArray[i]} CompareTo {moviesArray[j]} is: {comparisonResult}");
            allComparisons[i] = comparisonResult;
        }
    }

    // Expected array with all -1 values
    int[] expectedComparisons = expectedArrayGenerator(numComparisons, 0);

    // Assert
    bool assertion = Enumerable.SequenceEqual(expectedComparisons, allComparisons);
    Assert.IsTrue(assertion);

}
```

Figure 5: Test for testing if 0 is outputted correctly for all possible comparisons

5.1.2 ToString()

Back to test plan for ToString.

```
[TestMethod]
public void ToString_all_properties()
{
    Movie mov1 = new Movie("Batman", MovieGenre.Action, MovieClassification.M, 100, 2);
    string expected = "Movie(Title: Batman, Genre: Action, Classification: M, Duration: 100)";
    string actual = mov1.ToString();

    Console.WriteLine(expected);
    Console.WriteLine(actual);

    StringAssert.Equals(expected, actual);
}

[TestMethod]
public void ToString_only_Movie_Title()
{
    Movie mov1 = new Movie("Batman");
    string expected = "Movie(Title: Batman, Genre: 0, Classification: 0, Duration: 0)";
    string actual = mov1.ToString();

    Console.WriteLine(expected);
    Console.WriteLine(actual);

    StringAssert.Equals(expected, actual);
}
[TestMethod]
public void ToString_null_titled_movie()
{
    Assert.IsTrue((new Movie(null)).ToString() == null);
}
```

Figure 6: Test for testing if 0 is outputted correctly for all possible comparisons

5.2 MovieCollection ADT

5.2.1 IsEmpty()

Back to test plan for IsEmpty.

```
[TestMethod]
public void IsEmpty_True_empty_collection()
{
    int oldNumber = coll1.Number;
    bool result = coll1.IsEmpty();
    int newNumber = coll1.Number;
    Assert.IsTrue((result == true) && (oldNumber == newNumber));
}
```

Figure 7: Testing when collection is empty

```
[TestMethod]
public void IsEmpty_True_after_deleting_single_collection()
{
    coll1.Insert(mov0); // Insert a Movie so it isn't empty
    coll1.Delete(mov0); // Delete the movie so it is empty again
    int oldNumber = coll1.Number;
    bool result = coll1.IsEmpty();
    int newNumber = coll1.Number;
    Assert.IsTrue((result == true) && (oldNumber == newNumber));
}
[TestMethod]
public void IsEmpty_True_after_deleting_large_collection()
{
    // Insert lots of movies
    coll1.Insert(mov1); coll1.Insert(mov3); coll1.Insert(mov6);
    coll1.Insert(mov5); coll1.Insert(mov4);

    // Delete all the movies so it is empty again
    coll1.Delete(mov1); coll1.Delete(mov3); coll1.Delete(mov6);
    coll1.Delete(mov5); coll1.Delete(mov4);
    int oldNumber = coll1.Number;
    bool result = coll1.IsEmpty();
    int newNumber = coll1.Number;
    Assert.IsTrue((result == true) && (oldNumber == newNumber));
}
```

Figure 8: Testing when a *single* or *large* collection are emptied

```
[TestMethod]
public void IsEmpty_False_single_collection()
{
    coll1.Insert(mov0); // Insert a Movie so it isn't empty
    int oldNumber = coll1.Number;
    bool result = coll1.IsEmpty();
    int newNumber = coll1.Number;
    Assert.IsTrue((result == false) && (oldNumber == newNumber));
}
[TestMethod]
public void IsEmpty_False_large_collection()
{
    // Insert lots of movies
    coll1.Insert(mov1); coll1.Insert(mov3); coll1.Insert(mov6);
    coll1.Insert(mov5); coll1.Insert(mov4);
    int oldNumber = coll1.Number;
    bool result = coll1.IsEmpty();
    int newNumber = coll1.Number;
    Assert.IsTrue((result == false) && (oldNumber == newNumber));
}
```

Figure 9: Testing for non-empty *single* or *large* collections

5.2.2 Insert(IMovie movie)

Back to test plan for Insert.

```
[TestMethod]
public void Insert_root()
{
    int oldNumber = coll1.Number;
    bool result = coll1.Insert(mov1);
    int newNumber = coll1.Number;

    Assert.IsTrue((result == true) && ((oldNumber+1) == newNumber));
}
```

Figure 10: Inserting root into empty collection

```
[TestMethod]
public void Insert_root_LChild()
{
    // Get the console output stream
    var consoleOut = new StringWriter();
    Console.SetOut(consoleOut); // Start reading

    int oldNumber = coll1.Number;
    coll1.Insert(mov2); // root
    coll1.Insert(mov1); // LChild of mov2
    int newNumber = coll1.Number;

    //Read the last line from the console output stream
    var consoleLines = consoleOut.ToString().Split(new[] { Environment.NewLine }, StringSplitOptions.RemoveEmptyEntries);

    var actual = consoleLines[consoleLines.Length - 1];
    var expected = "movie is a LChild"; // From insert method

    coll1.PrettyPrint();
    Assert.IsTrue(actual == expected && ((oldNumber+2) == newNumber));

}
```

Figure 11: Inserting an LChild of a root

```
[TestMethod]
public void Insert_root_RChild()
{
    // Get the console output stream
    var consoleOut = new StringWriter();
    Console.SetOut(consoleOut); // Start reading

    int oldNumber = coll1.Number;
    coll1.Insert(mov2); // root
    coll1.Insert(mov3); // RChild of mov2
    int newNumber = coll1.Number;

    // Read the last line from the console output stream
    var consoleLines = consoleOut.ToString().Split(new[] { Environment.NewLine }, StringSplitOptions.RemoveEmptyEntries);

    var actual = consoleLines[consoleLines.Length - 1];
    var expected = "movie is a RChild"; // From insert method

    coll1.PrettyPrint();
    Assert.IsTrue(actual == expected && ((oldNumber + 2) == newNumber));
}
```

Figure 12: Inserting an RChild of a root

```
[TestMethod]
public void Insert__duplicate()
{

    coll1.Insert(mov0);
    int oldNumber = coll1.Number;
    bool result = coll1.Insert(mov0);
    int newNumber = coll1.Number;
    Assert.IsTrue( (result == false) && (oldNumber == newNumber));
}
```

Figure 13: Inserting a duplicate movie into a collection

```
[TestMethod]
public void Insert_multiple_movies()
{
    int oldNumber = coll1.Number;
    bool result = (coll1.Insert(mov3) && coll1.Insert(mov1)
        && coll1.Insert(mov0) && coll1.Insert(mov2));
    int newNumber = coll1.Number;
    Assert.IsTrue((result == true) && ((oldNumber+4) == newNumber));
}
```

Figure 14: Inserting multiple movies into a collection

5.2.3 ToArray()

Back to test plan for ToArray.

```
[TestMethod]
public void ToArray_single_collection()
{
    coll1.Insert(mov1);
    IMovie[] expected = new IMovie[1] { mov1 };
    IMovie[] actual = coll1.ToArray();

    Console.WriteLine("Expected");
    foreach (IMovie movie in expected) Console.WriteLine(movie);

    Console.WriteLine("\nActual");
    foreach (IMovie movie in actual) Console.WriteLine(movie);

    bool assertion = Enumerable.SequenceEqual(expected, actual);
    Assert.IsTrue(assertion);
}
[TestMethod]
public void ToArray_large_collection()
{
    coll1.Insert(mov1); coll1.Insert(mov0); coll1.Insert(mov2);
    IMovie[] expected = new IMovie[3] { mov0, mov1, mov2 };
    IMovie[] actual = coll1.ToArray();

    Console.WriteLine("Expected");
    foreach (IMovie movie in expected) Console.WriteLine(movie);

    Console.WriteLine("\nActual");
    foreach (IMovie movie in actual) Console.WriteLine(movie);

    bool assertion = Enumerable.SequenceEqual(expected, actual);
    Assert.IsTrue(assertion);
}
```

Figure 15: ToArray for a single and large collection

```
[TestMethod]
public void ToArray_empty_collection()
{
    IMovie[] expected = Array.Empty<IMovie>();
    IMovie[] actual = coll1.ToArray();
    Console.WriteLine("Expected");
    foreach (IMovie movie in expected) Console.WriteLine(movie);

    Console.WriteLine("\nActual");
    foreach (IMovie movie in actual) Console.WriteLine(movie);

    bool assertion = Enumerable.SequenceEqual(expected, actual);
    Assert.IsTrue(assertion);
}
```

Figure 16: ToArray for an empty collection

5.2.4 Search(string title)

Back to test plan for Search.

```
[TestMethod]
public void Search_doesnt_exist_large_collection()
{
    coll1.Insert(mov1); coll1.Insert(mov3); coll1.Insert(mov6);
    coll1.Insert(mov5); coll1.Insert(mov4);
    int oldNumber = coll1.Number;
    IMovie? result = coll1.Search(mov1.Title);
    int newNumber = coll1.Number;
    Assert.IsTrue((result == null) && (oldNumber == newNumber));
}
```

Figure 17: Searching for mov1 in a large collection that doesn't have it

```
[TestMethod]
public void Search_doesnt_exist_single_collection()
{
    coll1.Insert(mov0);
    int oldNumber = coll1.Number;
    IMovie? result = coll1.Search(mov1.Title);
    int newNumber = coll1.Number;
    Assert.IsTrue((result == null) && (oldNumber == newNumber));

}
```

Figure 18: Searching for mov1 in a single collection that doesn't have it

```
[TestMethod]
public void Search_empty_collection()
{
    int oldNumber = coll1.Number;
    IMovie? result = coll1.Search(mov0.Title);
    int newNumber = coll1.Number;
    Assert.IsTrue((result == null) && (oldNumber==newNumber));
}
```

Figure 19: Searching for mov0 in an empty collection

```
[TestMethod]
public void Search_exists_large_collection()
{
    coll1.Insert(mov1); coll1.Insert(mov0); coll1.Insert(mov2);
    int oldNumber = coll1.Number;
    IMovie? result = coll1.Search(mov1.Title);
    int newNumber = coll1.Number;
    Assert.IsTrue((object.ReferenceEquals(mov1, result))
        && (oldNumber == newNumber));
}
```

Figure 20: Searching for mov1 in a large collection that has it

```
[TestMethod]
public void Search_exists_single_collection()
{
    coll1.Insert(mov0);
    int oldNumber = coll1.Number;
    IMovie? result = coll1.Search(mov0.Title);
    int newNumber = coll1.Number;
    Assert.IsTrue((object.ReferenceEquals(mov0,result))
        && (oldNumber == newNumber));
}
```

Figure 21: Searching for mov0 in a large collection that has it

```
[TestMethod]
public void Search_null_in_large_collection()
{
    coll1.Insert(mov1); coll1.Insert(mov3); coll1.Insert(mov6);
    coll1.Insert(mov5); coll1.Insert(mov4);
    int oldNumber = coll1.Number;
    IMovie? result = coll1.Search(null);
    int newNumber = coll1.Number;
    Assert.IsTrue((result == null) && (oldNumber == newNumber));
}
```

Figure 22: Searching for null in a large collection

```
[TestMethod]
public void Search_null_in_single_collection()
{
    coll1.Insert(mov0);
    int oldNumber = coll1.Number;
    IMovie? result = coll1.Search(null);
    int newNumber = coll1.Number;
    Assert.IsTrue((result == null) && (oldNumber == newNumber));
}
```

Figure 23: Searching for null in a single collection

5.2.5 Delete(IMovie movie)

Back to test plan for Delete.

```
[TestMethod]
public void Delete_node_not_in_single_collection()
{
    coll1.Insert(mov0);

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov2); // "B"  doesn't exist
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[1] { mov0 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == false) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber));
}
```

Figure 24: Deleting a mov0 not in a *single* collection. Note that the assertion captures all post-conditions

```
[TestMethod]
public void Delete_node_not_in_large_collection()
{
    coll1.Insert(mov1); coll1.Insert(mov3); coll1.Insert(mov6);
    coll1.Insert(mov5); coll1.Insert(mov4);

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov2); // "B"  doesn't exist
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[5] { mov1, mov3, mov5, mov4, mov6 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == false) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber));
}
```

Figure 25: Deleting a mov2 not in a *large* collection.

```
[TestMethod]
public void Delete_node_from_empty_collection()
{
    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov0); // "Ar"  doesn't exist
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[0];
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == false) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber));
}
```

Figure 26: Deleting a mov0 not in an empty collection.

```
[TestMethod]
public void Delete_null_node_in_single_collection()
{
    coll1.Insert(mov0);

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(null); // null doesn't exist'
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[1] { mov0 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == false) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber));
}
```

Figure 27: Deleting a null node not in a *single* collection.

```
[TestMethod]
public void Delete_root_single_collection()
{
    coll1.Insert(mov0); // "Ar"  root

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov0); // "Ar"  root
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[0];
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == true) && (coll1.Search(mov0.Title) == null) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber - 1));
}
```

Figure 28: Deleting the root; mov0

```
[TestMethod]
public void Delete_root_has_only_LChild_leaf()
{
    coll1.Insert(mov2); // "B"  root
    coll1.Insert(mov1); // "Av" root.LChild

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov2); // "B"  root
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[1] { mov1 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == true) && (coll1.Search(mov2.Title) == null) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber - 1));
}
```

Figure 29: Deleting the root; mov2

```
[TestMethod]
public void Delete_root_has_only_RChild_leaf()
{
    coll1.Insert(mov2); // "B" root
    coll1.Insert(mov3); // "C" root.RChild

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov2); // "B" root
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[1] { mov3 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == true) && (coll1.Search(mov2.Title) == null) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber - 1));
}
```

Figure 30: Deleting the root; mov2

```
[TestMethod]
public void Delete_node_has_only_LChild_leaf()
{
    coll1.Insert(mov2); // "B"  root
    coll1.Insert(mov3); // "C" root.RChild
    coll1.Insert(mov1); // "Av" root.LChild
    coll1.Insert(mov0); // "Ar" root.LChild.LChild

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov1); // "Av" root.LChild
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[3] { mov0, mov2, mov3 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == true) && (coll1.Search(mov1.Title) == null) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber - 1));
}
```

Figure 31: Deleting the node; mov1

```
[TestMethod]
public void Delete_node_has_only_RChild_leaf()
{
    coll1.Insert(mov0); // "Ar" root
    coll1.Insert(mov1); // "Av" root.RChild
    coll1.Insert(mov2); // "B"  root.RChild.RChild

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov2); // "Ar" root.RChild.RChild
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[2] { mov0, mov1 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == true) && (coll1.Search(mov2.Title) == null) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber - 1));
}
```

Figure 32: Deleting the node; mov2

```
[TestMethod]
public void Delete_node_has_LChild_and_RChild_as_leaves()
{
    coll1.Insert(mov1); // "Av" root
    coll1.Insert(mov3); // "C"  root.RChild
    coll1.Insert(mov2); // "B"  root.RChild.LChild
    coll1.Insert(mov4); // "D"  root.RChild.RChild

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov3); // "C" root.LChild
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[3] { mov1, mov2, mov4 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == true) && (coll1.Search(mov3.Title) == null) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber - 1));
}
```

Figure 33: Deleting the node; mov3

```
[TestMethod]
public void Delete_node_has_LChild_with_LeftSkewedtree__and_RChild_as_leaf()
{
    coll1.Insert(mov0); // "Ar"       root
    coll1.Insert(mov3); // "C"        root.RChild
    coll1.Insert(mov2); // "B"        root.RChild.LChild
    coll1.Insert(mov5); // "Ca"       root.RChild.RChild
    coll1.Insert(mov1); // "Av"       root.RChild.LChild.RChild

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov3); // "C" root.LChild
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[4] { mov0, mov1, mov2, mov5 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == true) && (coll1.Search(mov3.Title) == null) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber - 1));
}
```

Figure 34: Deleting the node; mov3

```
[TestMethod]
public void Delete_node_has_LChild_with_RightSkewedtree_and_RChild_as_leaf()
{
    coll1.Insert(mov6); // "E"  root
    coll1.Insert(mov3); // "C"  root.LChild
    coll1.Insert(mov4); // "D"  root.LChild.RChild
    coll1.Insert(mov0); // "Ar" root.LChild.LChild
    coll1.Insert(mov1); // "Av" root.LChild.LChild.RChild
    coll1.Insert(mov2); // "B"  root.LChild.LChild.RChild.RChild

    int oldNumber = coll1.Number;
    bool deletedResult = coll1.Delete(mov3); // "C" root.LChild
    int newNumber = coll1.Number;

    IMovie[] expectedArray = new IMovie[5] { mov0, mov1, mov2, mov4, mov6 };
    IMovie[] actualArray = coll1.ToArray();

    bool arrayAssertion = Enumerable.SequenceEqual(expectedArray, actualArray);
    Assert.IsTrue((deletedResult == true) && (coll1.Search(mov3.Title) == null) &&
                  (arrayAssertion == true) &&
                  (newNumber == oldNumber - 1));
}
```

Figure 35: Deleting the node; mov3

5.2.6 Clear()

Back to test plan for Clear.

```
// Clear
[TestMethod]
public void Clear_single_collection()
{
    coll1.Insert(mov1);
    int prevNumber = coll1.Number; // 1
    coll1.Clear();
    int newNumber = coll1.Number; // 0
    Assert.IsTrue(newNumber == 0);
}
[TestMethod]
public void Clear_empty_collection()
{
    int prevNumber = coll1.Number; // 0
    coll1.Clear();
    int newNumber = coll1.Number; // 0

    Assert.IsTrue(newNumber == 0);
}
[TestMethod]
public void Clear_large_collection()
{
    coll1.Insert(mov1); coll1.Insert(mov3);
    coll1.Insert(mov6); coll1.Insert(mov5);
    coll1.Insert(mov4);
    int prevNumber = coll1.Number; // 5
    coll1.Clear();
    int newNumber = coll1.Number; // 0

    Assert.IsTrue(newNumber == 0);
}
```

Figure 36: Clear works in all three scenarios

5.2.7 NoDVDs()

Back to test plan for NoDVDs.

```
[TestMethod]
public void NoDVDS_empty_collection()
{
    int oldNumber = coll1.Number; // 0
    int dvds = coll1.NoDVDs();    // 0
    int newNumber = coll1.Number; // 0
    Assert.IsTrue( (dvds == 0) && (oldNumber == newNumber) );
}
[TestMethod]
public void NoDVDS_single_collection()
{
    coll1.Insert(mov0);
    int oldNumber = coll1.Number; // 5
    int dvds = coll1.NoDVDs();    // 1
    int newNumber = coll1.Number; // 5
    Assert.IsTrue((dvds == 1) && (oldNumber == newNumber));
}
[TestMethod]
public void NoDVDS_large_collection()
{
    coll1.Insert(mov1); coll1.Insert(mov3); coll1.Insert(mov6);
    coll1.Insert(mov5); coll1.Insert(mov4);

    int oldNumber = coll1.Number; // 5
    int dvds = coll1.NoDVDs();
    int newNumber = coll1.Number; // 5
    Assert.IsTrue((dvds == 5) && (oldNumber == newNumber));
}
```

Figure 37: NoDVDs and post-condtion is correct for all 3 tests