Assignment 1: CAB301

Shridhar Thorat: n10817239

May 5, 2023

1 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. appendix for CompareTo.

1.1 Movie ADT

1.1.1 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.

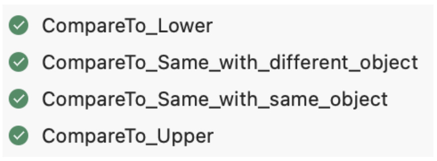
To test if -1 was outputted correctly, each `movie[i]` was *compared to* each movie after it (`movie[i+1]`) where `i` ranged from 0 to the length of the array of movies. It was expected that each comparison would output -1 since the movies were already arranged in descending order. This was called `CompareTo_Lower`.

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.

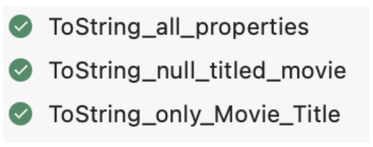
Unit testing summarisation


✓ CompareTo_Lower
✓ CompareTo_Same_with_different_object
✓ CompareTo_Same_with_same_object
✓ CompareTo_Upper

1.1.2 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


✓ ToString_all_properties
✓ ToString_null_titled_movie
✓ ToString_only_Movie_Title

1.2 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);
```

1.2.1 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`:

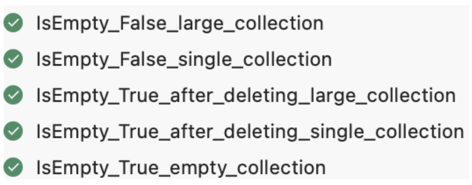
   Where the method returns `false` for a collection with lots of movie.

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


✓ IsEmpty_False_large_collection
✓ IsEmpty_False_single_collection
✓ IsEmpty_True_after_deleting_large_collection
✓ IsEmpty_True_after_deleting_single_collection
✓ IsEmpty_True_empty_collection

1.2.2 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).

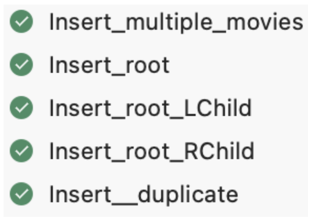
Lastly, as a sanity check, it was tested whether inserting multiple movies all outputted true. It is worth noting that testing insertion of multiple movies is not required. This is because inserting a movie into a Binary Search Tree is never done between existing movie but only as leaves. For example, 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

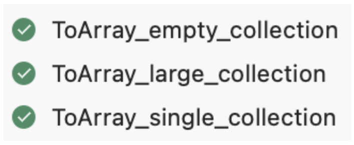

✓ Insert_multiple_movies
✓ Insert_root
✓ Insert_root_LChild
✓ Insert_root_RChild
✓ Insert__duplicate

1.2.3 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


✓ ToArray_empty_collection
✓ ToArray_large_collection
✓ ToArray_single_collection

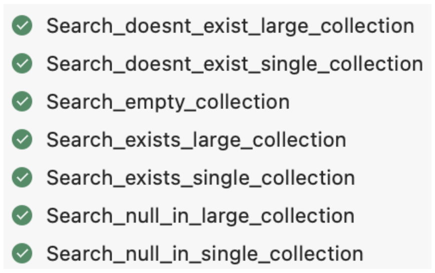
1.2.4 Search(string title)

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



✓ Search_doesnt_exist_large_collection
✓ Search_doesnt_exist_single_collection
✓ Search_empty_collection
✓ Search_exists_large_collection
✓ Search_exists_single_collection
✓ Search_null_in_large_collection
✓ Search_null_in_single_collection

1.2.5 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`

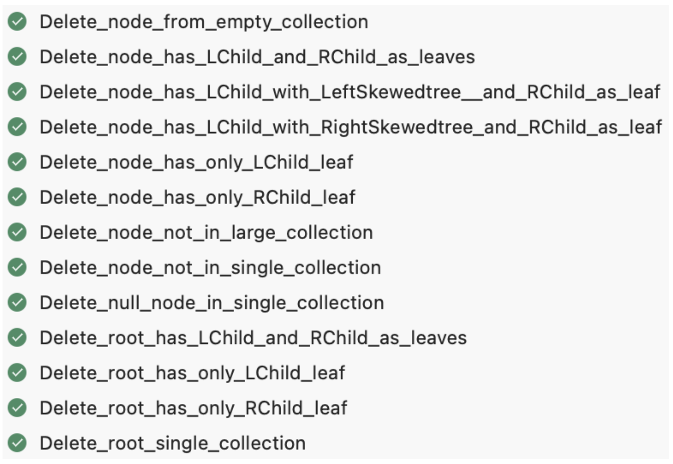
    Where the method returns `true` when a node that has a `RChild` leaf and and a `LChild` with a left-skewed sub-tree, is deleted, and replaced by the immediate `LChild`. The replacement is checked to be correct by confirming that `coll1.ToArray` is in dictionary order. Since `ToArray` uses an $In-Order\ traversal\ method$, 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



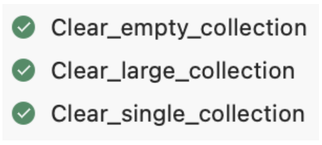
- ✓ Delete_node_from_empty_collection
- ✓ Delete_node_has_LChild_and_RChild_as_leaves
- ✓ Delete_node_has_LChild_with_LeftSkewedtree__and_RChild_as_leaf
- ✓ Delete_node_has_LChild_with_RightSkewedtree_and_RChild_as_leaf
- ✓ Delete_node_has_only_LChild_leaf
- ✓ Delete_node_has_only_RChild_leaf
- ✓ Delete_node_not_in_large_collection
- ✓ Delete_node_not_in_single_collection
- ✓ Delete_null_node_in_single_collection
- ✓ Delete_root_has_LChild_and_RChild_as_leaves
- ✓ Delete_root_has_only_LChild_leaf
- ✓ Delete_root_has_only_RChild_leaf
- ✓ Delete_root_single_collection

1.2.6 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



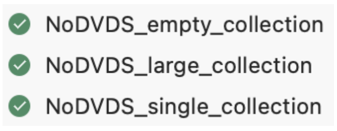
- ✓ Clear_empty_collection
- ✓ Clear_large_collection
- ✓ Clear_single_collection

1.2.7 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



- ✓ NoDVDS_empty_collection
- ✓ NoDVDS_large_collection
- ✓ NoDVDS_single_collection

2 References

3 Appendix

3.1 Trial Data