CIS 2344 - Algorithms Processes and Data

Log Book

A collection of exercises, notes, documentation and self-assessments.

Lewis Scrivens U1657778
U1657778@unimail.hu.ac.uk

Contents:

Randomisation

I decided that the best way of making a more efficient shuffle on a given array list would require a way without the while loop as that increases the time taken over more efficient ways. The method I used to decrease the time taken was to:

- Create a copy of the element in the array (Integer).
- Set that copy equal to the element at a random index using the provided method in RandomListing.
- Then to set the element at the random index equal to the copy created.
- Repeat the last three steps for each element in the array using a for loop.

CleverRandomListing.java

```
package intArrays;
import java.util.Arrays;

/**
 * An improvement of SimpleRandomListing used to randomly shuffle a given array.
 *
 * @author Lewis Scrivens
 * @version September 2017
 */

public class CleverRandomListing extends RandomListing
{
    // Variable for printing the time taken for the randomise function to run.
    public static long timeTaken;

    public CleverRandomListing(int size)
    {
        super(size);
    }

    protected void randomise()
    {
       startTimer();// Start Timer

       // Create local variables
       int copy;
       int randomIndex;
       int arraySize = getArray().length;

       // For every value in the array swap with a random value in the array.
       for (int i = 0; i < arraySize; i++)
       {
              randomIndex = getRandomIndex();

              // Create copy of array at i.
              copy = getArray()[i];
              // Set element of array at i to the element at the randomIndex.
              getArray()[i] = getArray()[randomIndex];
              // Set element of array at the randomIndex to the element at i.
              getArray()[randomIndex] = copy;
       }

       timeTaken = endTimer();// End Timer
    }
    public static void main(String[] args)
    {
       // Create a new clever random listing of size 10,000.
       RandomListing count = new CleverRandomListing(10000);
       System.out.println(Arrays.toString(count.getArray()));

       // Print the total time taken to the console.
       System.out.println("Time Taken: " + timeTaken);
    }

} // End of class CleverRandomListing
```

Testing

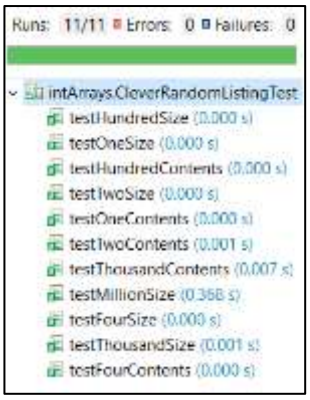
I have reused the test cases from the simple random listing test class as they are written in a way that it just works with the code I have wrote; I have also added comments to sections of the tests to show that I understand how they work.

CleverRandomListingTest.java

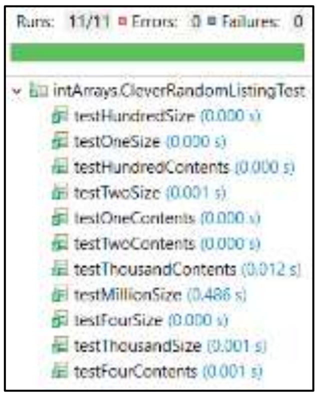
```
package intArrays;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TestName;

public class CleverRandomListingTest extends ListingTest
{
       private static long testStart, testEnd;
       @Rule public TestName testName = new TestName();

       @Before
       public void setUp() throws Exception
       {
              testStart = System.nanoTime();
       }
       @After
       public void tearDown() throws Exception
       {
              testEnd = System.nanoTime();
              System.out.println("Test \"" + testName.getMethodName() + "\" took " +
              (testEnd-testStart)/1000 + " microseconds");
       }
       @Test
       public void testOneSize()
       {
              testSize(1,new SimpleRandomListing(1));
       }
       @Test
       public void testOneContents()
       {
              testContents(1,new SimpleRandomListing(1));
       }
       @Test
       public void testTwoSize()
       {
              testSize(2,new SimpleRandomListing(2));
       }
```

```
	@Test
	public void testTwoContents()
	{
		testContents(2,new SimpleRandomListing(2));
	}
	@Test
	public void testFourSize()
	{
		testSize(4,new SimpleRandomListing(4));
	}

	@Test
	public void testFourContents()
	{
		testContents(4,new SimpleRandomListing(4));
	}
	@Test
	public void testHundredSize()
	{
		testSize(100,new SimpleRandomListing(100));
	}

	@Test
	public void testHundredContents()
	{
		testContents(100,new SimpleRandomListing(100));
	}

	@Test
	public void testThousandSize()
	{
		testSize(1000,new SimpleRandomListing(1000));
	}

	@Test
	public void testThousandContents()
	{
		testContents(1000,new SimpleRandomListing(1000));
	}

	@Test
	public void testMillionSize()
	{
		testSize(1000000,new SimpleRandomListing(1000000));
	}
}
```

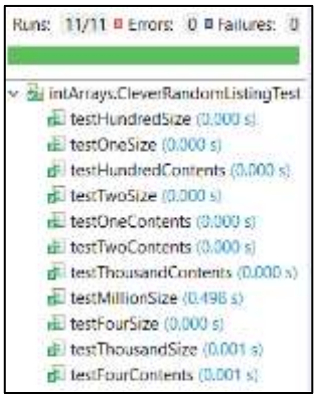
When running this test, it succeeds meaning that the code works correctly, and it gives me the time taken for each test. The three test results I ran to get an average time taken are shown below.



```
Runs: 11/11  Errors: 0  Failures: 0
intArrays.CleverRandomListingTest
    testHundredSize (0.000 s)
    testOneSize (0.000 s)
    testHundredContents (0.000 s)
    testTwoSize (0.000 s)
    testOneContents (0.000 s)
    testTwoContents (0.001 s)
    testThousandContents (0.007 s)
    testMillionSize (0.368 s)
    testFourSize (0.000 s)
    testThousandSize (0.001 s)
    testFourContents (0.000 s)
```



```
Runs: 11/11  Errors: 0  Failures: 0
intArrays.CleverRandomListingTest
    testHundredSize (0.000 s)
    testOneSize (0.000 s)
    testHundredContents (0.000 s)
    testTwoSize (0.001 s)
    testOneContents (0.000 s)
    testTwoContents (0.000 s)
    testThousandContents (0.012 s)
    testMillionSize (0.486 s)
    testFourSize (0.000 s)
    testThousandSize (0.001 s)
    testFourContents (0.001 s)
```



```
Runs: 11/11  Errors: 0  Failures: 0
intArrays.CleverRandomListingTest
    testHundredSize (0.000 s)
    testOneSize (0.000 s)
    testHundredContents (0.000 s)
    testTwoSize (0.000 s)
    testOneContents (0.000 s)
    testTwoContents (0.000 s)
    testThousandContents (0.000 s)
    testMillionSize (0.498 s)
    testFourSize (0.000 s)
    testThousandSize (0.001 s)
    testFourContents (0.001 s)
```

Comparison

I have also created a timing method to time how long it takes for the system to run the randomise method to compare the efficiency of SimpleRandomListing.java to CleverRandomListing.java.

Timer Methods in RandomListing.java and tests shown below:

```
public void startTimer()
{
   startTime = System.currentTimeMillis();// Start Timer, get current time.
}

public long endTimer()
{
   endTime = System.currentTimeMillis();// End Timer
   totalTime = endTime - startTime;// get total time taken.
   return totalTime;
}
```

List of size 1,000		
Test	SimpleRandomListing.java	CleverRandomListing.java
1	[600, 816, 265, Time Taken: 2	[115, 99, 736, Time Taken: 1
2	[146, 702, 736, Time Taken: 1	[108, 101, 127, Time Taken: 1
3	[8, 325, 852, Time Taken: 1	[200, 973, 51, : Time Taken: 1
Average Time Taken	1.3(1.d.p)(Milliseconds)	1 (Milliseconds)

List of size 10,000		
Test	SimpleRandomListing.java	CleverRandomListing.java
1	[2399, 9111, 9099, Time Taken: 5	[8302, 5711, 2533, 468, Time Taken: 2
2	[4503, 726, 3740, Time Taken: 5	[7565, 792, 5739, Time Taken: 2
3	[138, 9929, 329, Time Taken: 4	[6236, 1204, 181, Time Taken: 2
Average Time Taken	4.7(1.d.p) (Milliseconds)	2 (Milliseconds)

List of size 100,000		
Test	SimpleRandomListing.java	CleverRandomListing.java
1	[83755, 65601, 15317, Time Taken: 25	[24476, 10794, 64711, Time Taken: 7
2	[21457, 66764, 38676, Time Taken: 25	[73419, 27311, 81088, Time Taken: 8
3	[5917, 51947, 55999, Time Taken: 28	[63375, 3707, 22553, : Time Taken: 7
Average Time Taken	26 (Milliseconds)	7.3(1.d.p) (Milliseconds)

As you can see from the results above the CleverRandomListing was faster than the SimpleRandomListing, specifically on the higher volume array lists. Maybe an improvement I could have made would be to use nano-seconds to measure the time taken as it would be more accurate to see the difference in time taken between the two classes.

Generics

When creating my SwapMethod class I chose to create an array of names and swap between the indexes 0 and 3 as an example of the code working.

I did this by creating a method called swap, this method takes a String array and two integers which represent the two index's that will be swapped in the given array. I have also added some if statements that prevent the user from making errors such as trying to swap an item with itself or swapping index's that are not within the range of the array. I have explained the methods and code using comments seen bellow.

SwapMethod.java:

```
package genericMethods;

import java.util.Arrays;

/**
 * A generic method to swap two items in a given array.
 *
 * @author Lewis Scrivens
 * @version November 2017
 */

public class SwapMethod
{
	// Method takes a string and two integers which represent the array indices 
	to swap.
	public void swap(String[] array, int a, int b)
	{
		//If a or b is out of range. (Not an index within the given array)
		if (a > array.length - 1 || a < 0 || b > array.length - 1 || b < 0)
		{
			// Error printed to console.
			System.out.println("a or b is out of the array's range.");
		}
		else if (a == b)
		{
			// Error printed to console.
			System.out.println("Cannot swap the same item.");
		}
		else
		{
			// Variable to store a copy of a value while it is swapped.
			String copy;

			// Using the copy string to save the value of index a in array.
			copy = array[a];
			// Setting index, a to the value of index b in array.
			array[a] = array[b];
			// Setting index b in array to the copy string which was the 
			original index a in array.
			array[b] = copy;

			printArray(array);// Print the swapped array.
		}
	}

	public static void printArray(String[] array)
	{
		// Print swapped array to console.
		System.out.println(Arrays.toString(array));
	}
```

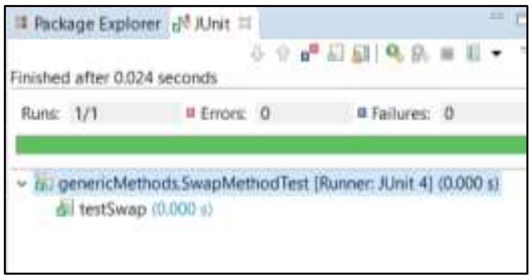
```
        /// Example for my SwapMethod, swapping elements 0 and 3 in an array.
        public void main(String[] args)
        {
                // Initialising variables
                int a, b;
                String[] names = new String[5];

                // Adding items to the names array.
                names[0] = "Lewis";
                names[1] = "Niamh";
                names[2] = "Sam";
                names[3] = "Tom";
                names[4] = "Ryan";

                int a = 0;// Index 1
                int b = 3;// Index 2

                // Method that swaps index a and b in any given array.
                swap(names, a, b);
        }
}
```

I have also created a Junit test class for swap method class to check if the swap works as intended. The code is shown below along with results from a test. The test could be automated by implementing a randomly generated array and performing random swaps on that array whilst checking for duplicates and successful swaps.

Testing

SwapMethodTest.java:

```
package genericMethods;

import static org.junit.Assert.*;
import org.junit.Test;

/**
 * A Test class for the SwapMethod class to ensure the methods work as intended.
 * @author Lewis Scrivens
 * @version November 2017
 */

public class SwapMethodTest
{
        @Test
        public void testSwap()
        {
                SwapMethod swap = new SwapMethod();
                String[] array = {"Lewis", "Tom", "Ryan", "Niamh", "Luke"};

                // a and b represent the indexes that will be swapped in the array.
                a = 1;
                b = 4;
                // swap items at index 1 and 4.
                swap.Swap(array, a, b);
                // Check if the values have been swapped.
                assertEquals(array[a], "Luke");
                assertEquals(array[b], "Tom");
        }
}
```



```
Finished after 0.024 seconds
Runs: 1/1    Errors: 0    Failures: 0
genericMethods.SwapMethodTest [Runner: JUnit 4] (0.000 s)
    testSwap (0.000 s)
```

Sorting Algorithms

I have implemented the quicksort and the selection sort. I used the following reference to help create the quick sorting algorithm and revised part of it to work with implementation of the ArraySort class.

QuickSort adaptation taken from http://www.java67.com/2014/07/quicksort-algorithm-in-java-in-place-example.html accessed on November 20th 2017.

The selection sort shown below takes each item in the array and compares it to every other value in that array. After doing this it returns a sorted array as everything is pushed to the correct place.

SelectionSort.java:

```
package arraySorter;

import RandomArray.RandomArray;
import RandomArray.RandomIntegerArray;
import java.util.Arrays;

/**
 * An extension of the ArraySortTool to implement the SelectionSort algorithm.
 *
 * @author Lewis Scrivens
 * @version November 2017
 */

public class SelectionSort<T extends Comparable<? super T>> implements ArraySort<T>
{
    public void sort(T[] array)
    {
         // Selection is an item in the array which is compared with every item
         after it using a for loop.
        for (int selection = 0; selection < array.length - 1; selection++)
        {
             // Smallest known value index is set to selection.
             int smallerValueIndex = selection;
             // Iterate through each index value after selection.
             for (int comparison = selection + 1; comparison < array.length;
            comparison++)
             {
                  // Check if the comparison is smaller than the current
                  smallerValueIndex.
                 if ((int)array[comparison] < (int)array[smallerValueIndex])
                  {
                       // Set the new smallerValueIndex to the comparison given its
                       smaller.
                       smallerValueIndex = comparison;
                  }
             }

             // Temporary copy of smaller number.
             T copy = array[smallerValueIndex];
             // Swap the smallest value with the current selection starting from the
            beginning of the array.
             array[smallerValueIndex] = array[selection];
             // Finish the swap by setting the new selection element to the copy of
            the smaller number.
             array[selection] = copy;
        }
    }
}
```

The QuickSort method was much more difficult to implement as I needed to add a method to be called recursively but after adapting the code from the original reference the quick sort I found this much easier. This sort method moves all the small values to the left and big values to the right and keeps doing this until the array is sorted.

QuickSort.java:

```
package arraySorter;

import RandomArray.RandomArray;
import RandomArray.RandomIntegerArray;
import java.util.Arrays;

/**
 * An extension of the ArraySortTool to implement the QuickSort algorithm.
 *
 * @author Lewis Scrivens
 * @version November 2017
 */

public class QuickSort<T extends Comparable<? super T>> implements ArraySort<T>
{
    public void sort(T[] array)
    {
       // Run the recursiveSort function on the whole array.
       recursiveSort(array, 0, array.length - 1);
    }

    public void recursiveSort(T[] array, int startIndex, int endIndex)
    {
       // Partition the array with the lower as the startIndex and the higher as
       the endIndex.
       int index = partition(array, startIndex, endIndex);

       // If the returned index - 1 is bigger than the current startIndex.
       if (startIndex < index - 1)
       {
             // Perform another recursiveSort on the left side.
             recursiveSort(array, startIndex, index - 1);
       }

       // If the returned index is smaller than the current endIndex.
       if (endIndex > index)
       {
             // Perform another recursiveSort on the right side.
             recursiveSort(array, index, endIndex);
       }
    }

    // Function for diving the array into two parts and comparing values from each
       half to swap.
    public int partition(T[] array, int left, int right)
    {
             // Set pivot (element to compare to) to element at the left index.
             int pivot = (int)array[left];

             // Run containing code while left is less than or equal to right.
             while (left <= right)
             {
                    // Increment left index by 1 when the element at the left index
                       is smaller than the pivot element.
                    while ((int)array[left] < pivot)
                    {
                           left++;
                    }
```

```
                // Decrement right index by 1 when the element at the right
                   index is larger than the pivot element.
                while ((int)array[right] > pivot)
                {
                      right--;
                }

                // If the left index is smaller or equal to the right index,
                   swap them.
                if (left <= right)
                {
                // Temporary copy of element at the left index.
                T copy = array[left];

                // Swap the left index element (smaller element) with the right
                   element.
                array[left] = array[right];

                // Finish the swap by setting the element at the right index to
                   the copy of the element at the left index.
                array[right] = copy;

                //Increment left and decrement right by 1 to continue the
                  partition.
                left++;
                right--;
                }
          }
          // Once the function as finished return the left index.
          return left;
     }
}
```

Testing

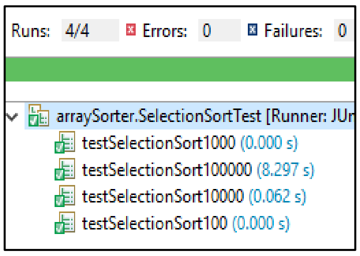
I have also created some Junit test cases for both the quick sort and the selection sort, I did this to ensure the array was sorted after the sort method was done. The following two tests classes were taken from the sort tester class written by Hugh.

SelectionSortTest.java:

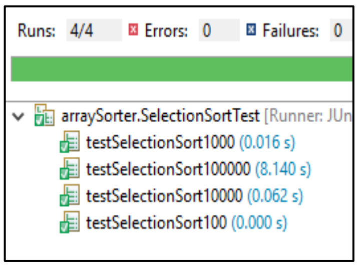
```
package arraySorter;
import static org.junit.Assert.*;
import org.junit.Test;
import RandomArray.RandomIntegerArray;

/**
 * A Test class for the SelectionSort to ensure all the methods work as intended.
 * Test cases were taken from the Sort Tester class which was written by Hugh.
 * @author Lewis Scrivens
 * @version November 2017
 */

public class SelectionSortTest
{
       @Test // Array with 100 elements
       public void testSelectionSort100()
       {
              SelectionSort sorter = new SelectionSort();
              RandomIntegerArray randomIntArray = new RandomIntegerArray(100);
              Integer[] array = randomIntArray.randomArray(100);
              sorter.sort(array);
              assert(sorter.isSorted(array));
       }
```

```
        @Test // Array with 1000 elements
        public void testSelectionSort1000()
        {
                SelectionSort sorter = new SelectionSort();
                RandomIntegerArray randomArray = new RandomIntegerArray(1000);
                Integer[] array = randomArray.randomArray(1000);

                sorter.sort(array);
                assert(sorter.isSorted(array));
        }
        @Test // Array with 10,000 elements
        public void testSelectionSort10000()
        {
                SelectionSort sorter = new SelectionSort();
                RandomIntegerArray randomArray = new RandomIntegerArray(10000);
                Integer[] array = randomArray.randomArray(10000);

                sorter.sort(array);
                assert(sorter.isSorted(array));
        }
        @Test // Array with 100,000 elements
        public void testSelectionSort100000()
        {
                SelectionSort sorter = new SelectionSort();
                RandomIntegerArray randomArray = new RandomIntegerArray(100000);
                Integer[] array = randomArray.randomArray(100000);

                sorter.sort(array);
                assert(sorter.isSorted(array));
        }
}
```

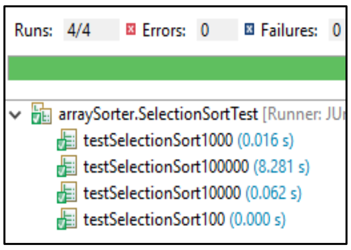
The results of this code over the three tests are shown below which prove that the code implemented works as intended.



```
Runs: 4/4   Errors: 0   Failures: 0

arraySorter.SelectionSortTest [Runner: JUn
    testSelectionSort1000 (0.000 s)
    testSelectionSort100000 (8.297 s)
    testSelectionSort10000 (0.062 s)
    testSelectionSort100 (0.000 s)
```



```
Runs: 4/4   Errors: 0   Failures: 0

arraySorter.SelectionSortTest [Runner: JUn
    testSelectionSort1000 (0.016 s)
    testSelectionSort100000 (8.140 s)
    testSelectionSort10000 (0.062 s)
    testSelectionSort100 (0.000 s)
```



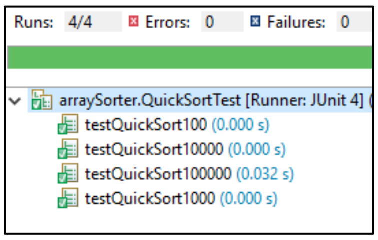
```
Runs: 4/4   Errors: 0   Failures: 0

arraySorter.SelectionSortTest [Runner: JU
    testSelectionSort1000 (0.016 s)
    testSelectionSort100000 (8.281 s)
    testSelectionSort10000 (0.062 s)
    testSelectionSort100 (0.000 s)
```

QuickSortTest.java:

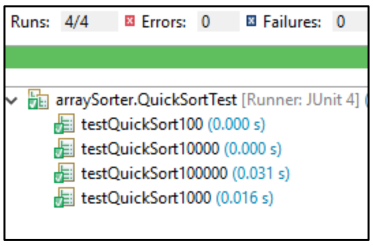
```
package arraySorter;
import static org.junit.Assert.*;
import org.junit.Test;
import RandomArray.RandomIntegerArray;

/**
 * A Test class for the QuickSort class to ensure all the methods work as intended.
 * Test cases were taken from the Sort Tester class which was written by Hugh.
 * @author Lewis Scrivens
 * @version November 2017
 */
```

```
public class QuickSortTest
{
	@Test // Array with 100 elements
	public void testQuickSort100()
	{
		QuickSort sorter = new QuickSort();
		RandomIntegerArray randomIntArray = new RandomIntegerArray(100);
		Integer[] array = randomIntArray.randomArray(100);
		sorter.sort(array);
		assert(sorter.isSorted(array));
	}
	@Test // Array with 1000 elements
	public void testQuickSort1000()
	{
		QuickSort sorter = new QuickSort();
		RandomIntegerArray randomArray = new RandomIntegerArray(1000);
		Integer[] array = randomArray.randomArray(1000);
		sorter.sort(array);
		assert(sorter.isSorted(array));
	}
	@Test // Array with 10,000 elements
	public void testQuickSort10000()
	{
		QuickSort sorter = new QuickSort();
		RandomIntegerArray randomArray = new RandomIntegerArray(10000);
		Integer[] array = randomArray.randomArray(10000);
		sorter.sort(array);
		assert(sorter.isSorted(array));
	}
	@Test // Array with 100,000 elements
	public void testQuickSort100000()
	{
		QuickSort sorter = new QuickSort();
		RandomIntegerArray randomArray = new RandomIntegerArray(100000);
		Integer[] array = randomArray.randomArray(100000);
		sorter.sort(array);
		assert(sorter.isSorted(array));
	}
}
```

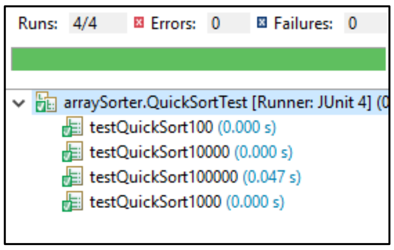
Below are the three results I got from this test and clearly the QuickSort is faster for larger values but for the smaller values the timing method used by the Junit test cases is not accurate enough, so I will next be using the timing method from the ArraySortTool which is in milliseconds.



```
Runs: 4/4    Errors: 0    Failures: 0
arraySorter.QuickSortTest [Runner: JUnit 4] (
    testQuickSort100 (0.000 s)
    testQuickSort10000 (0.000 s)
    testQuickSort100000 (0.032 s)
    testQuickSort1000 (0.000 s)
```



```
Runs: 4/4    Errors: 0    Failures: 0
arraySorter.QuickSortTest [Runner: JUnit 4] (
    testQuickSort100 (0.000 s)
    testQuickSort10000 (0.000 s)
    testQuickSort100000 (0.031 s)
    testQuickSort1000 (0.016 s)
```



```
Runs: 4/4    Errors: 0    Failures: 0
arraySorter.QuickSortTest [Runner: JUnit 4] (0
    testQuickSort100 (0.000 s)
    testQuickSort10000 (0.000 s)
    testQuickSort100000 (0.047 s)
    testQuickSort1000 (0.000 s)
```

(15.03.2018) **Note**: After writing this section a few months ago I did not realise that I could of just extended the sort tester class in selection sort test and quick sort test classes. This would have prevented repeating lines of code in the week 5 project. I notice this when going back over the code in the bubble sort test class.

Comparison

I also used the timing method implemented into the ArraySortTool class to test how long it would take for the three different algorithms (BubbleSort, SelectionSort and QuickSort) to sort varied sizes of random integer arrays. The four tables showing the results are shown below.

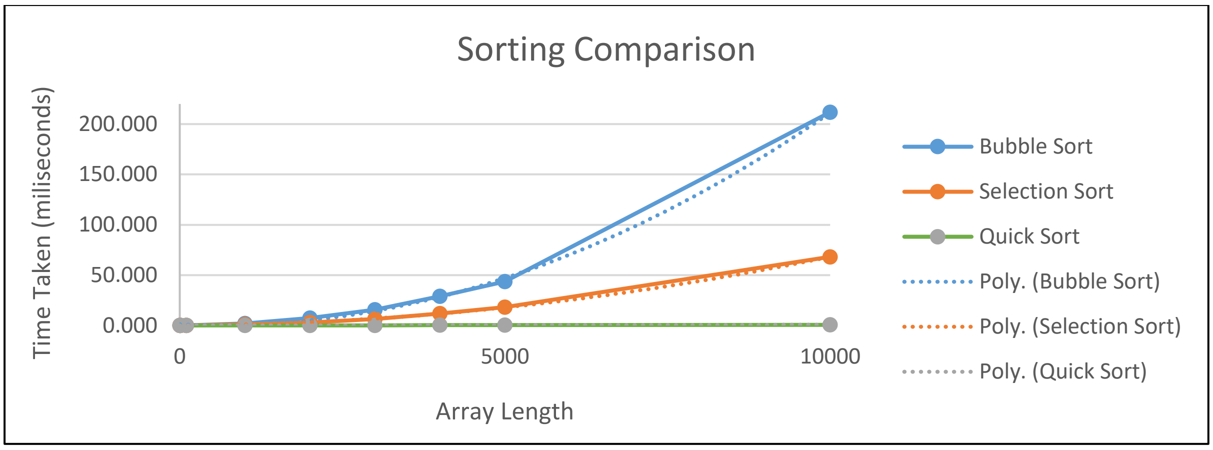
Test 1	Time Taken (3 d.p) (Milliseconds)		
Array Length	Bubble Sort	Selection Sort	Quick Sort
1	0	0.001	0.001
10	0.005	0.003	0.003
100	0.068	0.042	0.019
1,000	1.981	1.266	0.107
2,000	6.869	2.938	0.145
3,000	15.206	6.729	0.176
4,000	29.179	11.523	0.232
5,000	43.203	18.315	0.297
10,000	209.528	68.045	0.616

Test 2	Time Taken (3 d.p) (Milliseconds)		
Array Length	Bubble Sort	Selection Sort	Quick Sort
1	0	0	0.001
10	0.003	0.004	0.004
100	0.066	0.046	0.013
1,000	1.956	0.731	0.052
2,000	8.525	2.698	0.124
3,000	16.035	6.432	0.215
4,000	28.401	11.452	0.244
5,000	43.618	17.077	0.31
10,000	206.332	68.391	0.615

Test 3	Time Taken (3 d.p) (Milliseconds)		
Array Length	Bubble Sort	Selection Sort	Quick Sort
1	0.001	0	0.002
10	0.003	0.003	0.009
100	0.074	0.042	0.016
1,000	1.901	0.782	0.089
2,000	6.991	2.837	0.155
3,000	15.581	6.42	0.179
4,000	28.959	12.47	0.221
5,000	44.416	19.542	0.368
10,000	219.751	68.174	0.552

Average	Time Taken (3 d.p) (Milliseconds)		
Array Length	Bubble Sort	Selection Sort	Quick Sort
1	0.000	0.000	0.001
10	0.004	0.003	0.005
100	0.069	0.043	0.016
1,000	1.946	0.926	0.083
2,000	7.462	2.824	0.141
3,000	15.607	6.527	0.190
4,000	28.846	11.815	0.232
5,000	43.746	18.311	0.325
10,000	211.870	68.203	0.594

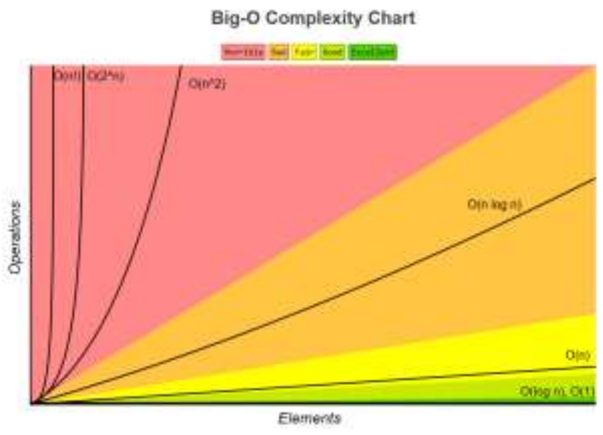
Using the results above I have created another table called average which is the average of each result for the three tests. I have done this so that I can get a reasonably accurate idea of the difference in time taken between the three different algorithms. On the next page I have also created a graph showing the difference between the efficiency of these algorithms.



Clearly the most efficient of the algorithms is the quick sort, followed by the selection sort then the bubble sort. The difference between the bubble sort and selection sort is that the selection sort is around 3 times more efficient whereas the quick sort is nearly 500 times more efficient in a list of size 10,000. The efficiency of the quick sort becomes more apparent the more elements in the array.

Big-0 Complexity equations:

By going to the link http://bigocheatsheet.com/ I found the following image which visually represents the Big 0 Complexity formulae for giving an indication to the efficiency of the sorts.



Looking at this chart I can see that the bubble sort's formulae is more towards O(n^2) which is 'horrible' by Big-O standards. The selection sort is O(n log(n)) which is 'bad' and then the quick sort which is 'excellent' in Big-O standards at O(log n) or O(1). These results also reflect my conclusion using the graph.

Polynomial Equations:

I have also used excel to determine an actual polynomial equation that best fits my current data, these equations can be seen below.

Selection Sort	Quick Sort
$y = (6 * 10^{-7})\ x^2 + 0.0004\ x - 0.0874$	$y = (-4 * 10^{-10})\ x^2 + (6 + 10^{-5})\ x + 0.0083$
Bubble Sort	
$y = (2 * 10^{-6})\ x^2 - 0.0026\ x + 0.9115$	

Lists (Singly Linked List)

I have implemented the list interface using singly linked lists into the java project. I have also added another method called get number of nodes which returns the current number of nodes in the list for testing purposes (to ensure that when I added nodes it was keeping track of those nodes correctly).

The code I have created to implement List<T> using singly linked lists is shown below including comments which explain important parts of the classes in more detail.

SinglyLinkedList.java:

```java
package linkedList;

/**
* An implementation of List<T> using singly linked lists. A singly linked list
* works by creating nodes that point to other nodes hence connecting into a singly
* linked list.
*
* @author Lewis Scrivens
* @version November 2017
*/

public class SinglyLinkedList<T> implements List<T>
{

	private Node<T> head = null;
	private Node<T> node = null;
	private int numberOfNodes = 0;

	// If the list is empty (null) then return true otherwise false.
	@Override
	public boolean isEmpty()
	{
		if (head == null)
		{
			return true;
		}
		else
		{
			return false;
		}
	}

	@Override
	public void add(int index, T value) throws ListAccessError
	{
		if (index == 0 && isEmpty())
		{
			// If index is 0 and the head is empty add the node value at
			the head (start of the list).
			head = new Node<T>(value);
		}
		else if (isEmpty())
		{
			// If head equals null and index is not 0.
			throw new ListAccessError("Index is out of bounds.");
		}
		else if (index < 0)
		{
			// If index is negative throw ListAccessError.
			throw new ListAccessError("Index is negative.");
		}
```

```
        else
        {
               // Set node to head (First in the list).
               node = head;

               // Find the node before the node to set to value.
               for(int i = 0; i < index - 1; i++)
               {
                      if (node.getNext() == null)
                      {
                             // If any node in-between the head and given index
                             is null then throw ListAccessError.
                             throw new ListAccessError("Index is out of
                             bounds.");
                      }
                      else
                      {
                             // Else set node to the next node.
                             node = node.getNext();
                      }
               }
               // Set the node at index to the value.
               node.setNext(new Node<T>(value));
        }
        // Increment the number of nodes after its added.
        numberOfNodes++;
 }

 @Override
 public T remove(int index) throws ListAccessError
 {
        // Set node to head (First in the list).
        node = head;

        if (head == null)
        {
               // If the head file is null the list must be empty.
               throw new ListAccessError("List is Empty.");
        }
        else if (index < 0)
        {
               // If the index is negative throw ListAccessError.
               throw new ListAccessError("Index is negative.");
        }
        else if (isEmpty())
        {
               // If the head file is null the list must be empty.
               throw new ListAccessError("List is Empty.");
        }
        else if (index == 0)
        {
               // If the first node is being removed set the head to the next
               index.
               head = node.getNext();
        }
        else
        {
               // For loop to get the next node until its in position to
               remove the index.
               for (int i = 0; i < index - 1; i++)
               {
                      node = node.getNext();
               }
```

```
                // Set the node at the index to the node in-front of it.
                node.setNext(node.getNext().getNext());
            }
            // Decrement numberOfNodes by 1 as one has been removed.
            numberOfNodes--;
            // Returns the new value of the node removed.
            return get(index);
        }

        @Override
        public T get(int index) throws ListAccessError
        {
            // Set node to head (First in the list).
            node = head;
            if (index < 0)
            {
                // If the given index is negative throw ListAccessError.
                throw new ListAccessError("Index is negative.");
            }
            else if (isEmpty())
            {
                // If the given index is negative throw ListAccessError.
                throw new ListAccessError("List is Empty.");
            }
            else if (index > numberOfNodes - 1)
            {
                // If the index is greater than the number of nodes throw
                ListAccessError.
                throw new ListAccessError("Index is out of bounds.");
            }
            else
            {
                // For loop will go through each node checking if it is empty.
                for(int i = 0; i < index; i++)
                {
                    if (node.getNext() == null)
                    {
                        // If the node at the given index is empty (null)
                        then throw ListAccessError.
                        throw new ListAccessError("Index is out of
                        bounds.");
                    }
                    else
                    {
                        // Else get the next node and repeat until node is
                        equal the node at the given index.
                        node = node.getNext();
                    }
                }
            }
            // Else return the node.
            return node.getValue();
        }
        // Method used for testing size of the list after removal of nodes.
        public int getNumberOfNodes() throws ListAccessError
        {
            // If the number of nodes is 0 the list must be empty.
            if (numberOfNodes == 0)
            {
                throw new ListAccessError("List is Empty.");
            }
            else
            {
                // Else return the number of nodes.
                return numberOfNodes;
            }
        }
}
```

Testing

I have created a Junit test class for the singly linked list class which ensures all the methods I have wrote work as intended. I have used exception expectations and assert equals (Comparison between what has happened to expect happens) to do these tests. I may have gone overboard with the number of test but currently it ensures that nearly if not everything in the code works as intended.

The code will run tests on the following areas of the singly linked list class:

- Getting an item from the list.
- Getting a negative index from the list. (Testing the list access error)
- Getting an out of bounds index from the list. (Testing the list access error)
- Getting an index from an empty list. (Testing the list access error)
- Adding an item to the list.
- Adding an item to the list at a negative index. (Testing the list access error)
- Adding an item to the list at an out of bounds index. (Testing the list access error)
- Test the number of nodes in the list is correct.
- Test the number of nodes in the list is correct after a removal.
- Test the number of nodes in the list is correct in an empty list. (Testing the list access error)
- Remove a node from the list.
- Remove the first node.
- Remove the last node.
- Remove a negative index from the list.
- Remove a node when the list is empty.

SinglyLinkedListTest.java:

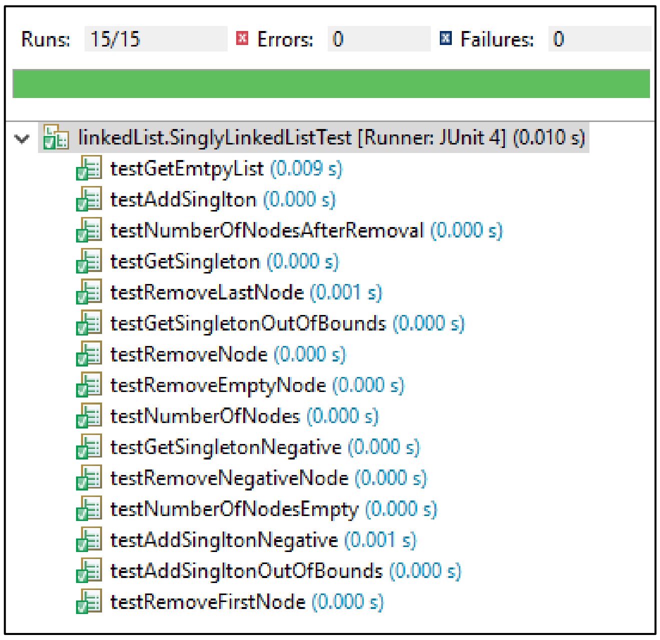
```java
package linkedList;

import static org.junit.Assert.*;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

/**
 * A Test class for the SinglyLinkedList class to ensure all the methods
 * work as intended, and the List created is in fact singly linked.
 *
 * @author Lewis Scrivens
 * @version December 2017
 */

public class SinglyLinkedListTest
{
	@Rule
	public ExpectedException exception = ExpectedException.none();

	@Test
	public void testGetSingleton() throws ListAccessError
	{
		SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
		// Add nodes to a singly linked list.
		list.add(0, 2); // Head node.
		list.add(1, 3); // Head nodes next node.
		list.add(2, 5);

		// Expected that the node at index 1 equals 3.
		assertEquals(new Integer(3),list.get(1));
	}
```

```
@Test
public void testGetSingletonNegative() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();

	list.add(0, 2);
	list.add(1, 3);
	list.add(2, 5);
	// Expect the following exception. Index is negative.
	exception.expect(ListAccessError.class);
	exception.expectMessage("Index is negative.");

	list.get(-1);
}
@Test
public void testGetSingletonOutOfBounds() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();

	list.add(0, 2);
	list.add(1, 3);
	list.add(2, 5);
	// Expect the following exception. Index is out of bounds.
	exception.expect(ListAccessError.class);
	exception.expectMessage("Index is out of bounds.");

	list.get(4);
}
@Test
public void testGetEmtpyList() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
	// Expect the following exception. List is Empty.
	exception.expect(ListAccessError.class);
	exception.expectMessage("List is Empty.");

	// Getting any number in an empty list.
	list.get(3);
}
@Test
public void testAddSinglton() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
	list.add(0, 2);
}
@Test
public void testAddSingltonNegative() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
	// Expect the following exception. Index is negative.
	exception.expect(ListAccessError.class);
	exception.expectMessage("Index is negative.");
	// Added to prevent out of bounds error due to the list not starting
	at index 0.
	list.add(0, 11);
	list.add(-1, 2);
}
@Test
public void testAddSingltonOutOfBounds() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
	// Expect the following exception. Index is out of bounds.
	exception.expect(ListAccessError.class);
	exception.expectMessage("Index is out of bounds.");

	list.add(4, 2);
}
```

```
@Test
public void testNumberOfNodes() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();

	list.add(0, 2);
	list.add(1, 3);
	list.add(2, 4);
	// Expected that the number of nodes will now be 3.
	assertEquals(list.getNumberOfNodes(), 3);
}
@Test
public void testNumberOfNodesAfterRemoval() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();

	list.add(0, 2);
	list.add(1, 3);
	list.add(2, 4);

	// Remove node from index 0 twice.
	list.remove(0);
	list.remove(0);
	// Expected that the number of nodes will now be 1.
	assertEquals(list.getNumberOfNodes(), 1);
}
@Test
public void testNumberOfNodesEmpty() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
	// Expect the following exception. List is Empty.
	exception.expect(ListAccessError.class);
	exception.expectMessage("List is Empty.");

	list.getNumberOfNodes();
}
@Test
public void testRemoveNode() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();

	list.add(0, 2);
	list.add(1, 3);
	list.add(2, 4);
	list.add(3, 3);
	// Expected that the new value at index 2 is 3 after removing the
	node.
	assertEquals(new Integer(3), list.remove(2));
}
@Test
public void testRemoveFirstNode() throws ListAccessError
{
	SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();

	list.add(0, 2);
	list.add(1, 3);
	list.add(2, 4);
	list.add(3, 3);
	// Expected that the new value at index 0 is 3 after removing the
	first node.
	assertEquals(new Integer(3), list.remove(0));
}
```

```
        @Test
        public void testRemoveLastNode() throws ListAccessError
        {
                SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();

                list.add(0, 2);
                list.add(1, 3);
                // Expect the following exception. Index is out of bounds.
                exception.expect(ListAccessError.class);
                exception.expectMessage("Index is out of bounds.");
                // Returned the node at the index removed, as it was the last node it
                equals null.
                list.remove(1);
        }
        @Test
        public void testRemoveNegativeNode() throws ListAccessError
        {
                SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();

                list.add(0, 2);
                list.add(1, 3);
                // Expect the following exception. Index is negative.
                exception.expect(ListAccessError.class);
                exception.expectMessage("Index is negative.");
                // Returned the node at the index removed, as it was the last node it
                equals null.
                list.remove(-2);
        }
        @Test
        public void testRemoveEmptyNode() throws ListAccessError
        {
                SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
                // Expect the following exception. List is Empty.
                exception.expect(ListAccessError.class);
                exception.expectMessage("List is Empty.");

                list.remove(1);
        }
}
```

I have run this test code and I feel that I have done a comprehensive set of tests for SinglyLinkedList.java. The results from running the test code is shown below. A way I could have improved the testing would be to fill the lists will randomly generated arrays and run the tests multiple times to ensure the methods are working as intended.



```
Runs: 15/15    Errors: 0    Failures: 0

linkedList.SinglyLinkedListTest [Runner: JUnit 4] (0.010 s)
    testGetEmtpyList (0.009 s)
    testAddSinglton (0.000 s)
    testNumberOfNodesAfterRemoval (0.000 s)
    testGetSingleton (0.000 s)
    testRemoveLastNode (0.001 s)
    testGetSingletonOutOfBounds (0.000 s)
    testRemoveNode (0.000 s)
    testRemoveEmptyNode (0.000 s)
    testNumberOfNodes (0.000 s)
    testGetSingletonNegative (0.000 s)
    testRemoveNegativeNode (0.000 s)
    testNumberOfNodesEmpty (0.000 s)
    testAddSingltonNegative (0.001 s)
    testAddSingltonOutOfBounds (0.000 s)
    testRemoveFirstNode (0.000 s)
```

Binary Trees and Traversals

To finish the implementation of binary trees in the provided project I completed each of the methods inside of the class “BinaryTree.java” such as insert, value, left and right. I also needed to add a tree node variable called root and a method named isEmpty to return a boolean on the root being null. Essentially what is happening to create this binary tree is that a tree node is being created at the root then every child etc.

The changes I have made to the BinaryTree class and the BTree class are shown below.

BTree.java:

```
package binaryTree;

public interface BTree<T extends Comparable<? super T>>
{
	public boolean isEmpty();
	public void insert(T value);
	public T value();
	public BTree<T> left();
	public BTree<T> right();
}
```

BinaryTree.java:

```
package binaryTree;

/**
* An implementation of BTree<T> to create a working binary tree with
* working methods. This binary tree creates a tree node class for each node
* on the binary tree.
* @author Lewis Scrivens
* @version December 2017
*/

public class BinaryTree<T extends Comparable<? super T>> implements BTree<T>
{
	TreeNode<T> root = null;

	// Method to return if the root is empty or not.
	@Override
	public boolean isEmpty()
	{
		if (root != null)
		{
			return false; // if the root is not empty then return false.
		}
		else
		{
			return true; // Otherwise return true.
		}
	}
	@Override
	public void insert(T value) // Insert method to add nodes to a binary tree.
	{
		// If the root is empty then initialise it with the given value.
		if (isEmpty())
		{
			root = new TreeNode<T>(value);
		}
```

```
            else
            {
                  // If the value compared to the roots value is less than 0 then
                     insert the value into left child of the root.
                  if (value.compareTo(value()) < 0)
                  {
                        root.left.insert(value);
                  }
                  // Otherwise insert it into the right child of the root.
                  else
                  {
                        root.right.insert(value);
                  }
            }
      }

      // Returns the roots current value.
      @Override
      public T value()
      {
            return root.value();
      }

      // Returns the left child binary tree of the root.
      @Override
      public BTree<T> left()
      {
            return root.left();
      }

      // Returns the right child binary tree of the root.
      @Override
      public BTree<T> right()
      {
            return root.right();
      }
}
```

The insert method inside of BinaryTree class was difficult to implement as it was complicated to turn an understanding from my first-year mathematics module into a java algorithm. The method currently checks if the root is empty on insertion of a value and if it is the root will be set to the inserted value.

If the root is not empty, then the value is compared to the current root value to check if it is bigger or smaller than it. If the inserted value is bigger than the root value, then it is inserted into the left child node of the root and if the value is smaller, it would be inserted into the right child node of the root. The other methods have comments which describe each of them.

Sorting Methods

After completing the implementation of the binary tree, I began creating another class which would hold all sorting methods for the binary tree named “BinaryTreeSortMethods.java”. The class contains three traversals on a given BinaryTree, these are in-order, post-order and pre-order.

BinaryTreeSortMethods.java:

```java
package binaryTree;
import java.util.ArrayList;
import java.util.List;

/**
* This class contains the sorting methods for any BTree starting from the given
* node. The methods in this class are recursive and will print the whole binary
* tree from the given node.
* @author Lewis Scrivens
* @version December 2017
*/

public class BinaryTreeSortMethods <T extends Comparable<? super T>>
{
	private List<T> traversal = new ArrayList<T>();

	public List<T> inOrder(BTree<T> node)
	{
		// If the node is not null then sort.
		if (node != null)
		{
			// If the left of node is empty then do not run.
			if (!node.left().isEmpty())
			{
				inOrder(node.left());
			}

			// Add the currently visited node to the traversal.
			traversal.add(node.value());

			// If the right of node is empty then do not run.
			if (!node.right().isEmpty())
			{
				inOrder(node.right());
			}
		}
		return traversal;
	}

	public List<T> postOrder(BTree<T> node)
	{
		if (node != null)
		{
			if (!node.left().isEmpty())
			{
				postOrder(node.left());
			}

			if (!node.right().isEmpty())
			{
				postOrder(node.right());
			}

			// Add the currently visited node to the traversal.
			traversal.add(node.value());
		}
		return traversal;
	}
	public List<T> preOrder(BTree<T> node)
	{
		if (node != null)
		{
			// Add the currently visited node to the traversal.
			traversal.add(node.value());

			if (!node.left().isEmpty())
			{
				preOrder(node.left());
			}

			if (!node.right().isEmpty())
			{
				preOrder(node.right());
			}
		}
		return traversal;
	}
}
```

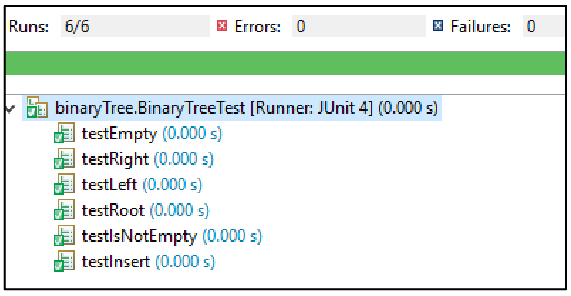
Testing

After finishing the Binary tree implementation, I added a test class to help test it to ensure all of the methods I had wrote using the lecture notes worked as intended. The code and individual test can be seen and are commented bellow.

BinaryTreeTest.java:

```
package binaryTree;

import static org.junit.Assert.*;
import org.junit.Test;

/**
* This class is used to show examples of how the sorting methods work and helps
* test the methods to ensure they are working as intended.
* @author Lewis Scrivens
* @version December 2017
*/

public class BinaryTreeTest
{
	@Test
	public void testEmpty()
	{
		BinaryTree tree = new BinaryTree();
		// Expected that the tree is empty as no values have been inserted.
		assertEquals(true, tree.isEmpty());
	}
	@Test
	public void testIsNotEmpty()
	{
		BinaryTree tree = new BinaryTree();
		// Insert something so the root is no longer null.
		tree.insert(" ");
		// Expected that the tree is not empty.
		assertEquals(false, tree.isEmpty());
	}
```

```
        @Test
        public void testLeft()
        {
                BinaryTree tree = new BinaryTree();
                // Insert the following into the BinaryTree tree.
                tree.insert("m");
                tree.insert("b");
                tree.insert("c");
                tree.insert("z");
                // Root->Left Node value.
                Comparable leftValue = tree.left().value();
                // B should be the left value as its smaller than the root M.
                assertEquals("b", leftValue);
        }
        @Test
        public void testRight()
        {
                BinaryTree tree = new BinaryTree();
                // Insert the following into the BinaryTree tree.
                tree.insert("m");
                tree.insert("b");
                tree.insert("c");
                tree.insert("z");
                // Root->Right Node value.
                Comparable rightValue = tree.right().value();
                // Z should be the left value as its bigger than the root M.
                assertEquals("z", rightValue);
        }

        @Test
        public void testInsert()
        {
                BinaryTree tree = new BinaryTree();
                // Insert the following into the Binary Tree tree.
                tree.insert("m");
                tree.insert("b");
                tree.insert("c");
                tree.insert("z");

                // Root->Left Node->Right Node value.
                Comparable value = tree.left().right().value();
                // I expect c to be in this position from the order I have inserted
                   the values.
                assertEquals("c", value);
        }

        @Test
        public void testRoot()
        {
                BinaryTree tree = new BinaryTree();

                // Insert the following to the BinaryTree tree.
                tree.insert("m");
                tree.insert("b");

                // Root value.
                Comparable root = tree.value();
                // I expect that the roots value is M as it was inserted first when
                   the root was null.
                assertEquals("m", root);
        }
}
```



```
Runs: 6/6        Errors: 0        Failures: 0

binaryTree.BinaryTreeTest [Runner: JUnit 4] (0.000 s)
    testEmpty (0.000 s)
    testRight (0.000 s)
    testLeft (0.000 s)
    testRoot (0.000 s)
    testIsNotEmpty (0.000 s)
    testInsert (0.000 s)
```

At first, I couldn't come up with many ways to test the binary tree sort methods code using Junit as the methods print to the console and I don't understand how to compare what has been output into the console to the expected outcome. After coming back to this a few weeks later I came up with a better test class shown on the next page.

OldBinaryTreeSortMethodsTest.java:

```
package binaryTree;

/**
* This class is used to show examples of how the sorting methods work and helps
* test the methods to ensure they are working as intended.
* @author Lewis Scrivens
* @version December 2017
*/

public class BinaryTreeTester
{
	public static void main(String[] args)
	{
		// The current example binary tree.
		BinaryTree tree = new BinaryTree();
		// To control the binary tree sort methods.
		BinaryTreeSortMethods sort = new BinaryTreeSortMethods();
		// true makes a binary tree of integers else strings.
		boolean integers = true;
		// 0 for inOrder, 1 for preOrder and 2 or anything else for postOrder
		int sortMethod = 1;

		// Adding nodes to the binary tree.
		if (integers)
		{
			tree.insert(new Integer(40));
			tree.insert(new Integer(23));
			tree.insert(new Integer(34));
			tree.insert(new Integer(1));
		}
		else
		{
			tree.insert("This");
			tree.insert("Message");
			tree.insert("Will");
			tree.insert("Print");
		}

		switch(sortMethod)
		{
		case 0:
			// Running an in order sort on the binary tree 'tree'.
			sort.inOrder(tree);
			break;
		case 1:
			// Running an pre order sort on the binary tree 'tree'.
			sort.preOrder(tree);
			break;
		case 2:
			// Running an post order sort on the binary tree 'tree'.
			sort.postOrder(tree);
			break;
		}
	}
}
```

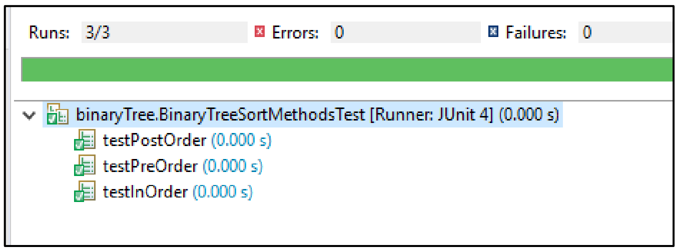
Using the old test class, I had to visually check if the console for the printed traversal and compare it to what it should be to find if it is working. Whereas with the new test required inclusion of a list in the binary tree sort methods class which is filled in the recursive methods and then returned to the test case.

BinaryTreeSortMethodsTest.java:

```
package binaryTree;

import static org.junit.Assert.*;
import org.junit.Test;

/**
* This test class is used for proving that the binary tree sort methods
* (traversals) work as intended.
*
* Used the following website to find the inOrder, preOrder and postOrder
* of my binaryTree to compare to the codes output.
* http://btv.melezinek.cz/binary-search-tree.html
*
* @author Lewis Scrivens
* @version December 2017
*/

public class BinaryTreeSortMethodsTest
{
	@Test
	public void testInOrder()
	{
		// The current example binary tree.
		BinaryTree tree = new BinaryTree();
		// To control the binary tree sort methods.
		BinaryTreeSortMethods sort = new BinaryTreeSortMethods();

		// Fill Binary tree with nodes.
		tree.insert(new Integer(40));
		tree.insert(new Integer(23));
		tree.insert(new Integer(34));
		tree.insert(new Integer(1));
		tree.insert(new Integer(22));

		Integer[] expectedTraversal = {1, 22, 23, 34, 40};
		// Check that the expected traversal is equal to the actual.
		assertEquals(expectedTraversal, sort.inOrder(tree).toArray());
	}

	@Test
	public void testPreOrder()
	{
		// The current example binary tree.
		BinaryTree tree = new BinaryTree();
		// To control the binary tree sort methods.
		BinaryTreeSortMethods sort = new BinaryTreeSortMethods();

		// Fill Binary tree with nodes.
		tree.insert(new Integer(40));
		tree.insert(new Integer(23));
		tree.insert(new Integer(34));
		tree.insert(new Integer(1));
		tree.insert(new Integer(22));
		// Check that the expected traversal is equal to the actual.
		Integer[] expectedTraversal = {40, 23, 1, 22, 34};
		assertEquals(expectedTraversal, sort.preOrder(tree).toArray());
	}
```

```
        @Test
        public void testPostOrder()
        {
                // The current example binary tree.
                BinaryTree tree = new BinaryTree();
                // To control the binary tree sort methods.
                BinaryTreeSortMethods sort = new BinaryTreeSortMethods();

                // Fill Binary tree with nodes.
                tree.insert(new Integer(40));
                tree.insert(new Integer(23));
                tree.insert(new Integer(34));
                tree.insert(new Integer(1));
                tree.insert(new Integer(22));
                // Check that the expected traversal is equal to the actual.
                Integer[] expectedTraversal = {22, 1, 34, 23, 40};
                assertEquals(expectedTraversal, sort.postOrder(tree).toArray());
        }
}
```

The code can be seen working which is how I concluded that I have done the traversals correctly.



```
Runs: 3/3        Errors: 0        Failures: 0

binaryTree.BinaryTreeSortMethodsTest [Runner: JUnit 4] (0.000 s)
    testPostOrder (0.000 s)
    testPreOrder (0.000 s)
    testInOrder (0.000 s)
```

Hash Tables

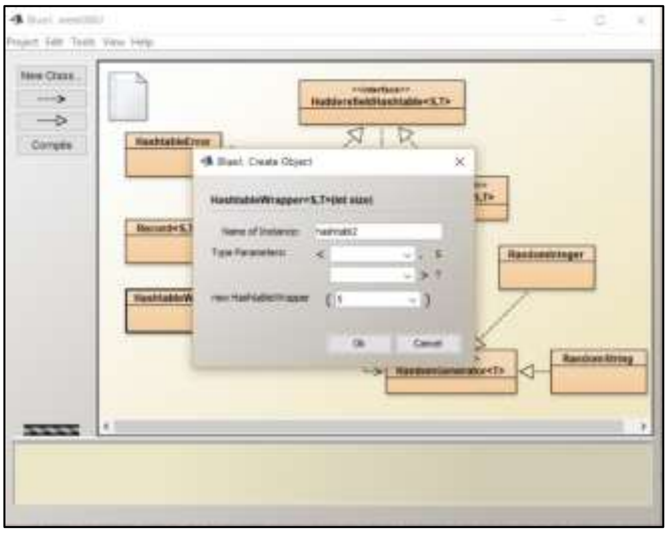
I completed this week's exercise for creating an instance of the HashtableWraper class in the BlueJ software. It made the inspection of the elements much easier and the ability to understand how the code works visually was also helpful.

Following the instructions of the logbook question (Screenshots of each step can be seen below) after inspecting the hash table object after it was first created, the internal array had a length of 5. However, after using the put method to insert all the values into the hash table object the array length had increased to 5. So, the array became bigger as more information was added to the hash table.

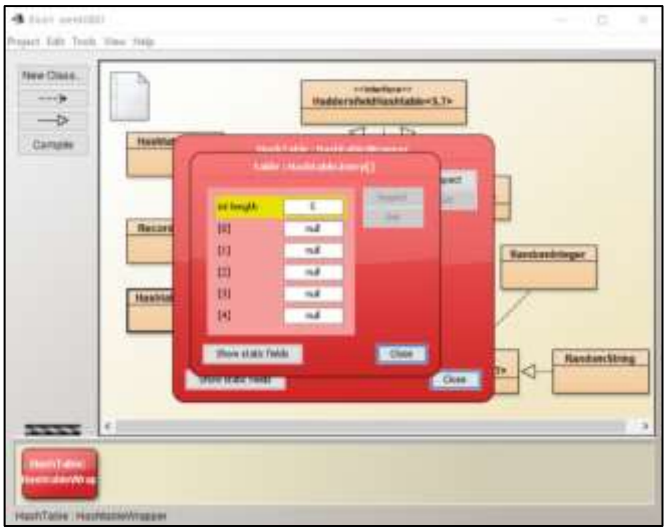
The way hash tables work is by hashing the key value where the hash given is linked to an index number in the array where the key and value is then placed. This is what's happening here and then when a record-B is placed into an index with an existing record-A, record-A is replaced and set as a next record pointer to record.

Instructions:

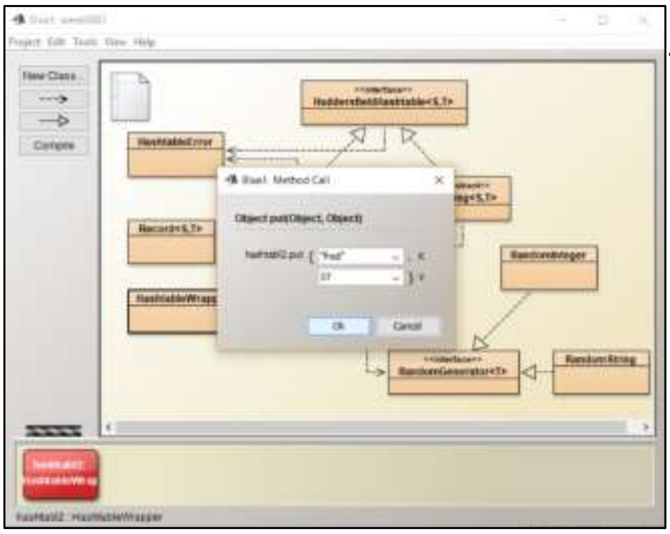
1



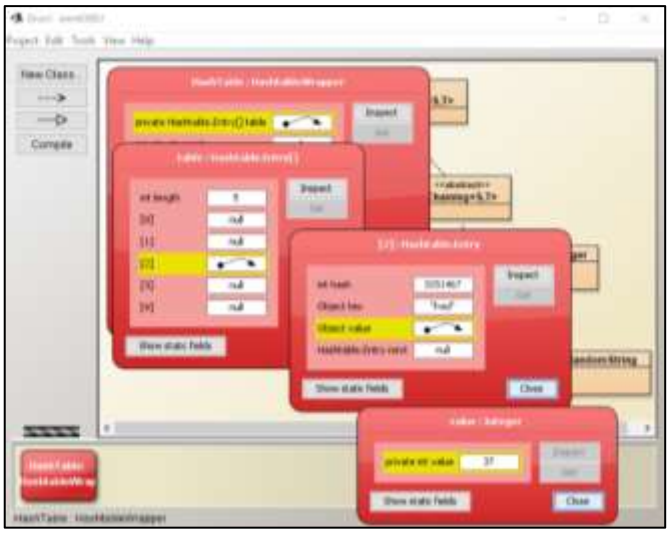
2



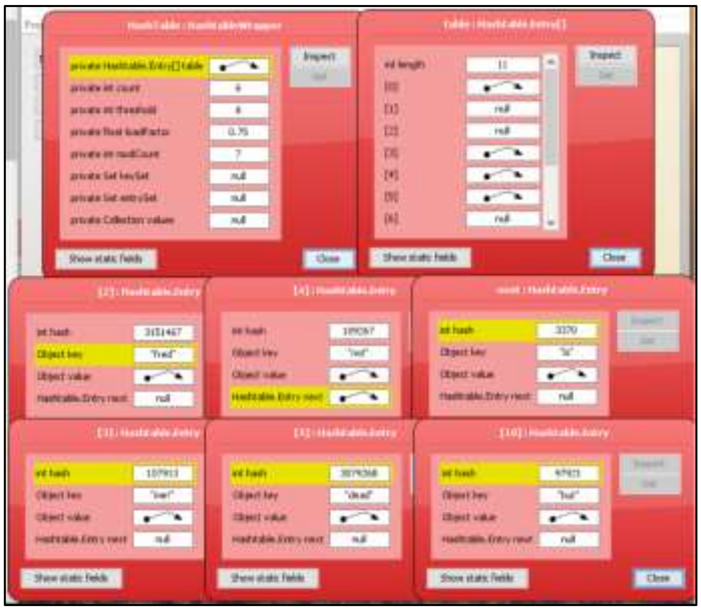
3



4



5



hashTable : HashtableWrapper

private Hashtable.Entry[] table	●→
private int count	6
private int threshold	8
private float loadFactor	0.75
private int modCount	7
private Set keySet	null
private Set entrySet	null
private Collection values	null

table : Hashtable.Entry[]

int length	11
[0]	●→
[1]	null
[2]	null
[3]	●→
[4]	●→
[5]	●→
[6]	null

[2] : Hashtable.Entry

int hash	3151467
Object key	"fred"
Object value	●→
Hashtable.Entry next	null

[4] : Hashtable.Entry

int hash	[illegible]
Object key	"red"
Object value	●→
Hashtable.Entry next	●→

next : Hashtable.Entry

int hash	3370
Object key	[illegible]
Object value	●→
Hashtable.Entry next	null

[3] : Hashtable.Entry

int hash	107913
Object key	[illegible]
Object value	●→
Hashtable.Entry next	null

[5] : Hashtable.Entry

int hash	[illegible]
Object key	"dead"
Object value	●→
Hashtable.Entry next	null

[10] : Hashtable.Entry

int hash	[illegible]
Object key	[illegible]
Object value	●→
Hashtable.Entry next	null

Also, from research I have done into hash tables and why they are so widely used is because the data's location is derived from a hash code of the key. So, by searching the key you would only have to hash the key again and the index that the key should be at is given. But if the key is not at the node at the given index, it will traverse through the singly linked list making it much faster than other methods of storing data. The way the hash tables allocate array slots to the data is by using the following function:

```
int index = (hash value of the key) % (array size)
```

This basically means that the chosen index for the data is the remainder given when dividing the hash value of the key by the array size. An example from the screenshot above is shown below.

- Fred in array index 2, has the hash value of 107913 and was the first to be input into the table when the array length was still 5. So initially the hash key for this was the following.

```
int index = 3151467 & 5  →  int index = 630269 remainder 2  → int index = 2
```

- Whereas because the array size has changed since the new index has been changed to.

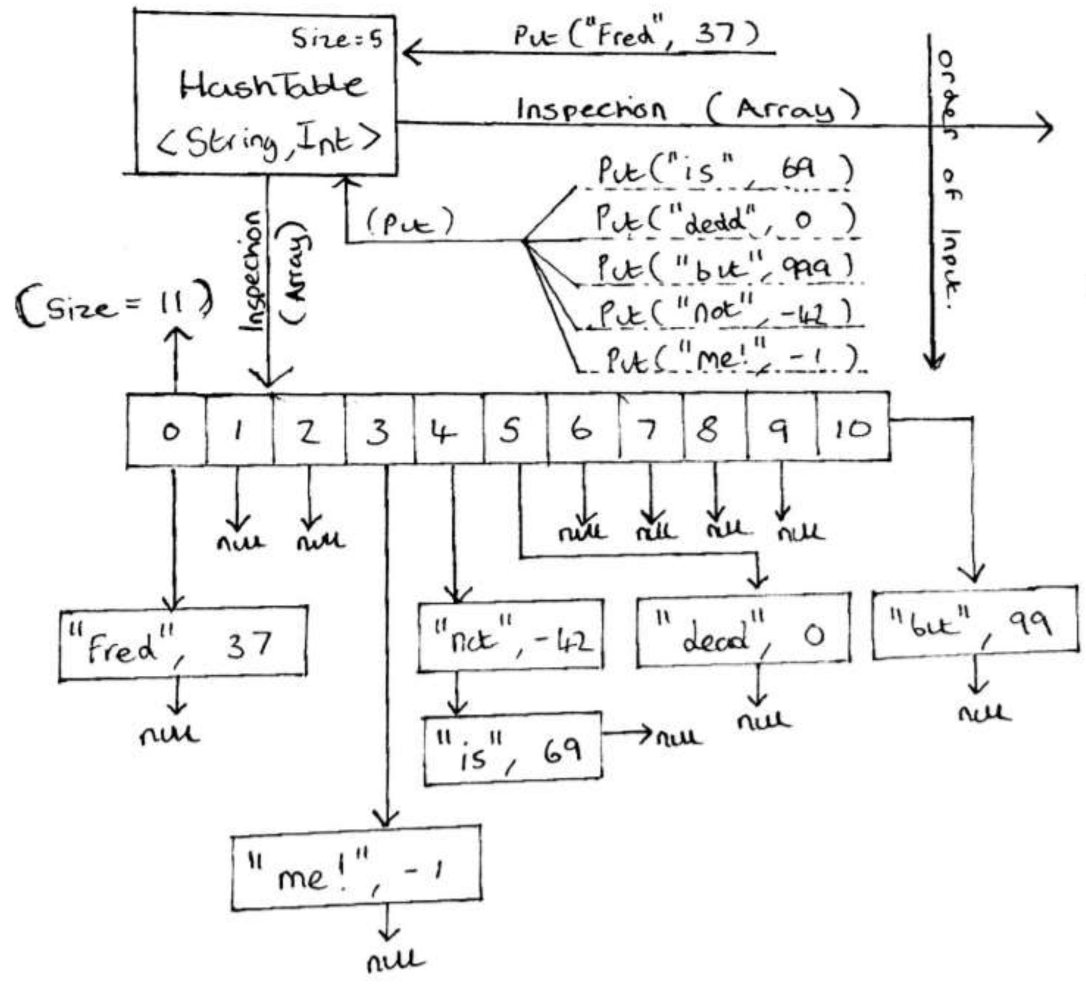
```
int index = 3151467 & 11  →  int index = 3151467 remainder 11  → int index = 11
```

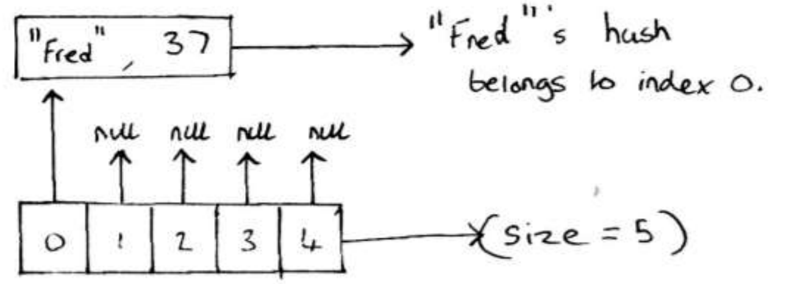
Regarding the memory management issues one of the obvious ones is that there is allot of empty space in a hash table, due to the way data is assigned to the array. Basically, there is no way to avoid wasted memory in a hash table as there will always be empty items in the array.

Looking at the Java API for hash tables I can see that there is a different version of a hash table called a hash map which can take null values unlike the hash table. One disadvantage of hash maps is that if thread synchronisation is needed for example to handle the arrays across multiple threads then hash tables would be the one to use whereas if individual threads handle arrays locally then hash maps would be better.

To help me better understand hash tables and how the code is working I have sketched a diagram of the instructions given in the logbook exercise I have included it on the next page with my written observations.

Hash Tables





note: From this diagram I have observed the following:

* If a record is put into an index that already contains a record it sets the old record as the new records nextRecord pointer. (Creating a singly linked list.)
* The size of the array has increased as more records have been added to it.

Graph Traversals

Before implementing the depth first traversal I used my understanding of the difference between it and the breadth first traversal in the lecture notes which is to recursively visit any neighbours of a node. Knowing this I constructed a DepthFirstTraversal class which is shown and explained with comments bellow.

DepthFirstTraversal.java:

```java
package graph;

import java.util.ArrayList;
import java.util.List;

/**
* An implementation of the depth first traversal on a graph, in this case the
* class extends the adjacency graph and implement the traversal methods.
*
* @author Lewis Scrivens
* @version January 2018
*/

public class DepthFirstTraversal<T> extends AdjacencyGraph<T> implements
Traversal<T>
{
	private List<T> traversal = new ArrayList<T>();

	/**
	 * Perform a depth first traversal of the adjacency graph and
	 * return the traversal as a list.
	 *
	 * @return a depth first traversal list of the graph
	 */
	@Override
	public List<T> traverse() throws GraphError
	{
		// Get the starting node for the traversal.
		T nextNode = getNextNode();

		if (nextNode == null)
		{
			throw new GraphError("The graph is empty, no nodes found to
			                      traverse.");
		}

		// While the node is not null visit the node then get the next node.
		while(nextNode != null)
		{
			visitNode(nextNode);
			nextNode = getNextNode();
		}

		// Once the traverse is complete return the traversal list.
		return traversal;
	}
	/**
	 * Check and return a boolean for if the traversal contains
	 * the node inputted.
	 *
	 * @return a boolean on if the traversal contains the given node.
	 */
	private boolean visited(T node)
	{
		// If the traversal contains the node return true else false.
		return traversal.contains(node);
	}

	/**
	 * Loops on the nodes in the graph until it a node is not
	 * contained in the traversal (hasn't been visited) then returns
	 * that node, otherwise null is returned.
	 *
	 * @return a node that has not yet been visited else null.
	 */
	private T getNextNode()
	{
		// For each node in the graph.
		for(T node: getNodes())
		{
			// If the traversal doesn't contain the node.
			if(!traversal.contains(node))
			{
				// Return the node.
				return node;
			}
		}

		// Return null if there are no unvisited nodes.
		return null;
	}

	/**
	 * Given node is added to the traversal then the method
	 * is ran for each of the nodes neighbours.
	 */
	public void visitNode(T node) throws GraphError
	{
		// If the node is in the traversal list then
		if(traversal.contains(node))
		{
			// exit method as it has been visited already.
			return;
		}

		// Add the node to the traversal list as its being visited.
		traversal.add(node);

		// For each neighbour node of "node".
		for(T neighbour: getNeighbours(node))
		{
			// If it has not been visited.
			if(!visited(neighbour))
			{
				// Visit the neighbour node.
				visitNode(neighbour);
			}
		}
	}
}
```

Testing

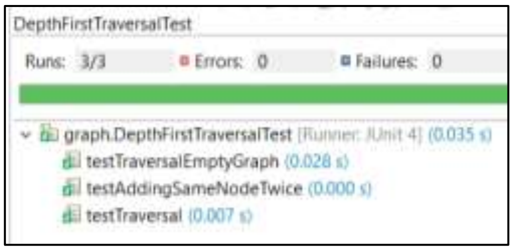
I have also implemented a Junit test case for the class depth first traversal class and the code can be seen bellow. I have also Included comments and method explanations.

DepthFirstTraversalTest.java:

```java
package graph;

import static org.junit.Assert.*;
import java.util.Arrays;
import java.util.List;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

/**
* A JUnit test case that tests if the depth first traversal class works
* as intended to. Specifically, the traversal method and some GraphErrors.
*
* @author Lewis Scrivens
* @version January 2018
*/

public class DepthFirstTraversalTest<T>
{
	DepthFirstTraversal<Integer> graph;

	@Rule
	public final ExpectedException exception = ExpectedException.none();

	/**
	 *     This graph is filled with values and the connections are made
	 *  so that it is identical to the graph in the lecture notes for
	 *  a depth first traversal.
	 *
	 * @throws GraphError if the same node is added twice to the graph.
	 */
	@Before
	public void buildGraph() throws GraphError
	{
		graph = new DepthFirstTraversal<Integer>();

		graph.add(0);
		graph.add(1);
		graph.add(2);
		graph.add(3);
		graph.add(4);
		graph.add(5);

		graph.add(0, 1);
		graph.add(0, 3);
		graph.add(1, 2);
		graph.add(2, 4);
		graph.add(2, 5);
		graph.add(4, 5);
	}
```

```
        /**
         *     Tests the class by checking the outcome of the depth first
         *  traversal on graph. It should output. "0 1 2 4 5 3".
         *
         * @throws GraphError.
         */
        @Test
        public void testTraversal() throws GraphError
        {
                List<Integer> traversal = graph.traverse();
                // The traversal outcome should be this as seen from the lecture
                   notes.
                List<Integer> outcome = Arrays.asList(0, 1, 2, 4, 5, 3);

                assertEquals(traversal, outcome);
        }

        /**
         *     Test method on adding the same node twice as it should throw a
         *  graph error.
         *
         * @throws GraphError with the message for adding two of the same nodes.
         */
        @Test
        public void testAddingSameNodeTwice() throws GraphError
        {
                Integer node = 0;
                // Expected graph error message.
                exception.expectMessage("Cannot add " + node + " to the graph.  This
                       node is " + "already in the graph already contains " + node);

                // Add repeated node to the graph.
                graph.add(0);
        }

        /**
         *     Test method to traverse an empty graph as it should throw
         *     a graph error
         *
         *     stating that the graph is empty.
         *
         * @throws GraphError message for an empty graph traversal.
         */
        @Test
        public void testTraversalEmptyGraph() throws GraphError
        {
                DepthFirstTraversal<Integer> emptyGraph = new
                                                  DepthFirstTraversal<Integer>();
                // Expect the following graph error.
                exception.expectMessage("The graph is empty, no nodes found to
                                         traverse.");

                // Run traversal method on empty graph.
                emptyGraph.traverse();
        }
}
```

Bellow you can see the Junit test class results, this test case could be improved by implementing a random graph generator etc.



DepthFirstTraversalTest

Runs: 3/3 Errors: 0 Failures: 0

graph.DepthFirstTraversalTest [Runner: JUnit 4] (0.035 s)
testTraversalEmptyGraph (0.028 s)
testAddingSameNodeTwice (0.000 s)
testTraversal (0.007 s)

Topological Sorts

I have implemented the Topological Sort interface using a reference counting topological sort. The getSort() method from the topological sort interface will return a list containing the reference counting topological sort. The sort will find the first node in order that has zero references and add it to the sort then remove it from the graph, it then breaks the for loop and restarts. This is done until there are no more nodes left in the graph.

ReferenceCountingSort.java:

```
package graph;

import java.util.ArrayList;
import java.util.List;

/**
* An extension of AdjacencyGraph and implementation of Topological Sort to
* create a reference counting topological sort on an adjacency graph.
*
* This sort will go to the nodes with 0 references in order.
* @author Lewis Scrivens
* @version January 2018
*/

public class ReferenceCountingSort<T> extends AdjacencyGraph<T> implements
TopologicalSort<T>
{
       private List<T> sort = new ArrayList<T>();

       /**
        * Perform a reference counting sort on the graph. While there are nodes in
        * the graph, find the first node that has 0 references and visit that node
       *  then break the for loop.
        *
        * @throws GraphError.
        */
       @Override
       public List<T> getSort() throws GraphError
       {
              // If the graph is empty then a sort cannot be done.
              if (getNoOfNodes() <= 0)
              {
                     throw new GraphError("The graph is empty, no nodes found to
                                                         sort.");
              }
              // While there are nodes in the graph.
              while (getNoOfNodes() > 0)
              {
                     // For each node in the graph.
                     for (T n: getNodes())
                     {
                            // If the reference count of that node is 0.
                            if (getReferenceCount(n) == 0)
                            {
                                   // Visit the node n.
                                   visitNode(n);

                                   // If a node is visited restart the for loop.
                                   break;
                            }
                     }
              }
              // Return the sorted list.
              return sort;
       }
```

```
        /**
         * Visit the given node by adding it to the sort then removing it
         * from the graph.
         *
         * @throws GraphError.
         */
        public void visitNode(T node) throws GraphError
        {
                // if the node has been visited do not continue and return.
                if (sort.contains(node)) return;

                // Add the node to the sorted list.
                sort.add(node);

                // Remove the visited node from the graph.
                this.remove(node);
        }
}
```

Also, I have implemented a method named getReferenceCount which will return the number of references on a given node to be used by the getSort method.

```
    /**
     * Find reference count for any given node.
     *
     * @throws GraphError.
     */
    public Integer getReferenceCount(T node) throws GraphError
    {
       Integer count = 0; // Start Integer count at 0.

       // For each node in the graph.
       for (T n: getNodes())
       {
               // If it contains the given node.
               if (contains(n, node))
               {
                       count++; // Increase the count by 1.
               }
       }

       // Return the reference count of the given node.
       return count;
    }
```

Testing

I have also created a Junit test case for the reference counting sort class, the test case will test the following:

- Reference Counting Sort on lecture graph.
- Reference Counting Sort on empty graph.
- Get reference count for node 4.
- Get reference count for node 9.
- Add same node to the graph.

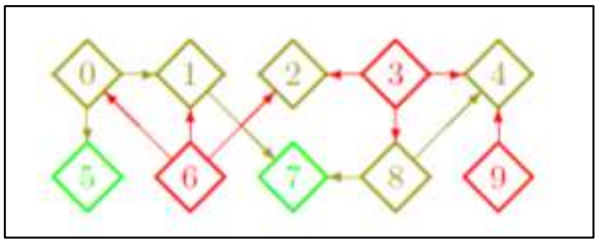
The test class and the results can be seen across the next few pages.

```java
package graph;

import static org.junit.Assert.*;
import java.util.Arrays;
import java.util.List;
import org.junit.Before;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

/**
* A Junit test case for the class ReferenceCountingSort. This is used to ensure
* all the methods work as intended.
*
* @author Lewis Scrivens
* @version January 2018
*/

public class ReferenceCountingSortTest
{
	ReferenceCountingSort<Integer> graph;

	@Rule
	public final ExpectedException exception = ExpectedException.none();

	/**
	 *  This graph is filled with values and the connections are made
	 *  so that it is identical to the graph in the lecture notes for
	 *  a reference counting sort.
	 *
	 * @throws GraphError.
	 */
	@Before
	// Construct graph from lecture notes to test reference counting sort.
	public void buildGraph() throws GraphError
	{
		graph = new ReferenceCountingSort<Integer>();

		// Adding nodes to the graph.
		graph.add(0);
		graph.add(1);
		graph.add(2);
		graph.add(3);
		graph.add(4);
		graph.add(5);
		graph.add(6);
		graph.add(7);
		graph.add(8);
		graph.add(9);

		// Adding connections between the nodes to complete the graph.
		graph.add(0, 1);
		graph.add(0, 5);
		graph.add(1, 7);
		graph.add(3, 2);
		graph.add(3, 4);
		graph.add(3, 8);
		graph.add(6, 0);
		graph.add(6, 1);
		graph.add(6, 2);
		graph.add(8, 4);
		graph.add(8, 7);
		graph.add(9, 4);
	}
```

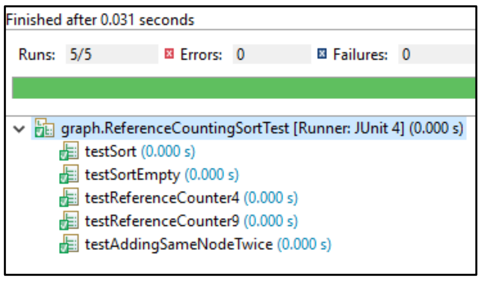
```
/**
 *  Tests the class by checking the outcome of the reference counting sort
 *  on graph. It should output "3 6 0 1 2 5 8 7 9 4".
 *
 * @throws GraphError.
 */
@Test
public void testSort() throws GraphError
{
       // Get the reference counting sort from graph.
       List<Integer> actual = graph.getSort();

       // The expected sort from the lecture notes used to compare to
       // the actual sort.
       List<Integer> expected = Arrays.asList(3, 6, 0, 1, 2, 5, 8, 7, 9, 4);

       // The sort should match the expected list as they are both
       // intended to be reference counting sorts of the constructed graph.
       assertEquals(expected, actual);
}

/**
 *  Test method on adding the same node twice as it should throw a
 *  graph error.
 *
 * @throws GraphError with the message for adding two of the same nodes.
 */
@Test
public void testAddingSameNodeTwice() throws GraphError
{
       Integer node = 0;

       // Expected graph error message.
       exception.expectMessage("Cannot add " + node + " to the graph.  This
              node is " + "already in the graph already contains " + node);

       // Add repeated node to the graph.
       graph.add(0);
}

/**
 *  Test method to sort an empty graph as it should throw a graph error
 *  stating that the graph is empty.
 *
 * @throws GraphError message for an empty graph sort.
 */
@Test
public void testSortEmpty() throws GraphError
{
       // Create an empty graph.
       ReferenceCountingSort<Integer> emptyGraph = new
       ReferenceCountingSort<Integer>();

       // Expect the following graph error.
       exception.expectMessage("The graph is empty, no nodes found to
                                   sort.");

       emptyGraph.getSort();
}
```

```
	/**
	 *	Test method to check that the reference counting sort is giving
	 *  the correct number of references for the node 4.
	 *
	 * @throws GraphError.
	 */
	@Test
	public void testReferenceCounter4() throws GraphError
	{
		// Get the actual reference count for node 4.
		Integer actual = graph.getReferenceCount(4);

		// The node 4 is expected to have 3 references.
		Integer expected = 3;

		// The expected and actual values should be identical
		if the code works.
		assertEquals(expected, actual);
	}

	/**
	 *	Test method to check that the reference counting sort is giving
	 *  the correct number of references for the node 9.
	 *
	 * @throws GraphError.
	 */
	@Test
	public void testReferenceCounter9() throws GraphError
	{
		// Get the actual reference count for node 9.
		Integer actual = graph.getReferenceCount(9);

		// The node 9 is expected to have 0 references.
		Integer expected = 0;

		// The expected and actual values should be identical
		if the code works.
		assertEquals(expected, actual);
	}
}
```

The graph that I have input is the graph from the lecture notes which I used to help me get a much better understanding of what the code was doing as well as having a visual graph to compare to the code results as I can see the reference count in the notes when the sorts are done step by step visually.



The results of this code can be seen below:



```
Finished after 0.031 seconds
Runs: 5/5    Errors: 0    Failures: 0

graph.ReferenceCountingSortTest [Runner: JUnit 4] (0.000 s)
    testSort (0.000 s)
    testSortEmpty (0.000 s)
    testReferenceCounter4 (0.000 s)
    testReferenceCounter9 (0.000 s)
    testAddingSameNodeTwice (0.000 s)
```

Concurrent Systems

The exercise to implement two counters in the count test class, one which counts from 0 to 10 and another that counts from 10 to 0. Decrementing by 1 each time. Before doing this, I was asked to implement the run method in the counter class that will start the counter and while it has not finished step the count.

The code implemented into the counter class can be seen bellow.

Counter.java run method:

```
/**
 * Method to start the counter and step while the
 * counter is not finished.
 *
 * @author Lewis Scrivens
 **/
public void run()
{
   // Start the counter.
   startCount();

   // While the count has not finished step every delay.
   while (!isFinished())
   {
          // Step the counter (and print step to console)
          stepCount();
   }
}
```

I was also asked to implement the run set method in the thread hash set class which I have done by creating a for loop that will go through each counter in the set and run it, the current counter will then wait until the thread is no longer running then continue the for loop.

Synchronous runSet method:

```
/**
 * Method that will run each thread then wait for itself to finish
 * before running the next thread in the for loop. This is done as
 * there is only one counter, so the threads must take turns making
 * use of it.
 *
 * @author Lewis Scrivens
 **/
@Override
public void runSet() throws InterruptedException
{
       // For each thread in this thread set start the thread.
       for (T thread: this)
       {
              // Run the current thread.
              thread.run();
              // While the thread is running wait here.
              thread.join();
       }
}
```

Asynchronous runSet method:

```
    /**
     * Asynchronous method that will run each thread waiting without waiting
     * for them to finish.
     *
     * @author Lewis Scrivens
     **/
      @Override
      public void runSet() throws InterruptedException
      {
             // For each thread in this thread set start the thread.
             for (T thread: this)
             {
                    // Run the current thread.
                    thread.start();
             }
             for (T thread: this)
             {
                    // While the thread is running wait here.
                    thread.join();
             }
      }
```

Testing

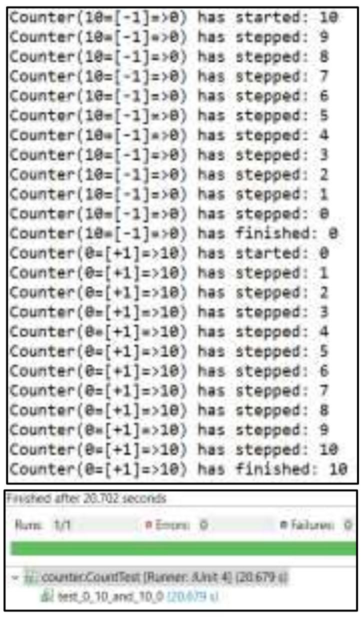
Synchronous runSet method:

I have implemented a new test into the code that will count from 0 to 10 and then it will count from 10 to 0.

```
    /**
     * Method to test running a counter from 0 to 10 and 10 to 0 at
     * the same time so it can be observed in the console.
     * This test method was adapted from Hugh's test method named
     * test_5_0_and_0_5().
     *
     * @author Lewis Scrivens
     **/
      @Test
      public void test_0_10_and_10_0() throws InterruptedException
      {
             countSet.add(new Counter(0,10));
             countSet.add(new Counter(10,0));

             countSet.runSet();
      }
```

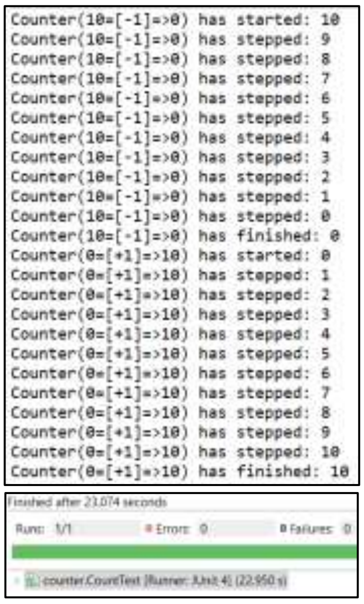
After running this test, I gathered some console output that indicates the tests are synchronous as the second counter appears to wait for the second counter. This is because I have the thread.join() line in the same for loop as the thread.start() line. This means that when the counter is run, thread.join() will cause each thread to wait for the current thread to finish the count. I was unsure if this is what Hugh wanted so I also created the Asynchronous runSet method.

Console output and test results for the three tests using the synchronous runSet method is on the next page.



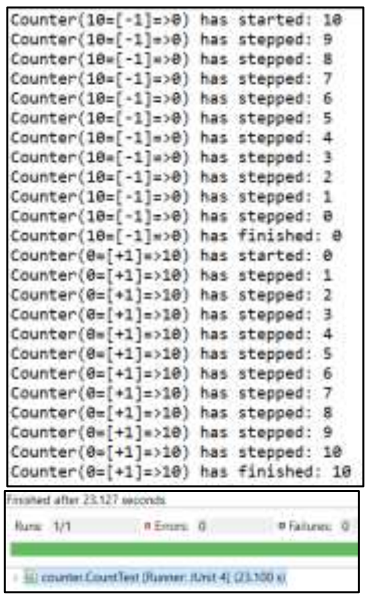
```
Counter(10=[-1]=>0) has started: 10
Counter(10=[-1]=>0) has stepped: 9
Counter(10=[-1]=>0) has stepped: 8
Counter(10=[-1]=>0) has stepped: 7
Counter(10=[-1]=>0) has stepped: 6
Counter(10=[-1]=>0) has stepped: 5
Counter(10=[-1]=>0) has stepped: 4
Counter(10=[-1]=>0) has stepped: 3
Counter(10=[-1]=>0) has stepped: 2
Counter(10=[-1]=>0) has stepped: 1
Counter(10=[-1]=>0) has stepped: 0
Counter(10=[-1]=>0) has finished: 0
Counter(0=[+1]=>10) has started: 0
Counter(0=[+1]=>10) has stepped: 1
Counter(0=[+1]=>10) has stepped: 2
Counter(0=[+1]=>10) has stepped: 3
Counter(0=[+1]=>10) has stepped: 4
Counter(0=[+1]=>10) has stepped: 5
Counter(0=[+1]=>10) has stepped: 6
Counter(0=[+1]=>10) has stepped: 7
Counter(0=[+1]=>10) has stepped: 8
Counter(0=[+1]=>10) has stepped: 9
Counter(0=[+1]=>10) has stepped: 10
Counter(0=[+1]=>10) has finished: 10
```

```
Finished after 20.702 seconds
Runs: 1/1    Errors: 0    Failures: 0
counter.CountTest [Runner: JUnit 4] (20.679 s)
    test_0_10_and_10_0 (20.679 s)
```



```
Counter(10=[-1]=>0) has started: 10
Counter(10=[-1]=>0) has stepped: 9
Counter(10=[-1]=>0) has stepped: 8
Counter(10=[-1]=>0) has stepped: 7
Counter(10=[-1]=>0) has stepped: 6
Counter(10=[-1]=>0) has stepped: 5
Counter(10=[-1]=>0) has stepped: 4
Counter(10=[-1]=>0) has stepped: 3
Counter(10=[-1]=>0) has stepped: 2
Counter(10=[-1]=>0) has stepped: 1
Counter(10=[-1]=>0) has stepped: 0
Counter(10=[-1]=>0) has finished: 0
Counter(0=[+1]=>10) has started: 0
Counter(0=[+1]=>10) has stepped: 1
Counter(0=[+1]=>10) has stepped: 2
Counter(0=[+1]=>10) has stepped: 3
Counter(0=[+1]=>10) has stepped: 4
Counter(0=[+1]=>10) has stepped: 5
Counter(0=[+1]=>10) has stepped: 6
Counter(0=[+1]=>10) has stepped: 7
Counter(0=[+1]=>10) has stepped: 8
Counter(0=[+1]=>10) has stepped: 9
Counter(0=[+1]=>10) has stepped: 10
Counter(0=[+1]=>10) has finished: 10
```

```
Finished after 23.074 seconds
Runs: 1/1    Errors: 0    Failures: 0
counter.CountTest [Runner: JUnit 4] (22.950 s)
```



```
Counter(10=[-1]=>0) has started: 10
Counter(10=[-1]=>0) has stepped: 9
Counter(10=[-1]=>0) has stepped: 8
Counter(10=[-1]=>0) has stepped: 7
Counter(10=[-1]=>0) has stepped: 6
Counter(10=[-1]=>0) has stepped: 5
Counter(10=[-1]=>0) has stepped: 4
Counter(10=[-1]=>0) has stepped: 3
Counter(10=[-1]=>0) has stepped: 2
Counter(10=[-1]=>0) has stepped: 1
Counter(10=[-1]=>0) has stepped: 0
Counter(10=[-1]=>0) has finished: 0
Counter(0=[+1]=>10) has started: 0
Counter(0=[+1]=>10) has stepped: 1
Counter(0=[+1]=>10) has stepped: 2
Counter(0=[+1]=>10) has stepped: 3
Counter(0=[+1]=>10) has stepped: 4
Counter(0=[+1]=>10) has stepped: 5
Counter(0=[+1]=>10) has stepped: 6
Counter(0=[+1]=>10) has stepped: 7
Counter(0=[+1]=>10) has stepped: 8
Counter(0=[+1]=>10) has stepped: 9
Counter(0=[+1]=>10) has stepped: 10
Counter(0=[+1]=>10) has finished: 10
```

```
Finished after 23.127 seconds
Runs: 1/1    Errors: 0    Failures: 0
counter.CountTest [Runner: JUnit 4] (23.100 s)
```

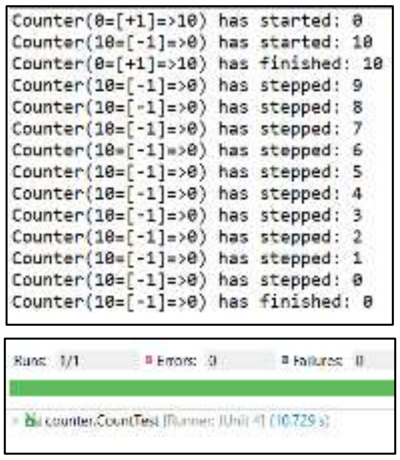
Asynchronous runSet method:

Note: The code is the same as the synchronous test just using the asynchronous runSet method in the 'countSet' variable.

After running this test, I gathered some console output that indicates the tests are asynchronous as the counters appear start in order without waiting for the current counter to finish. This is because I have the thread.join() line in a separate for loop that is ran after the counters are all started.

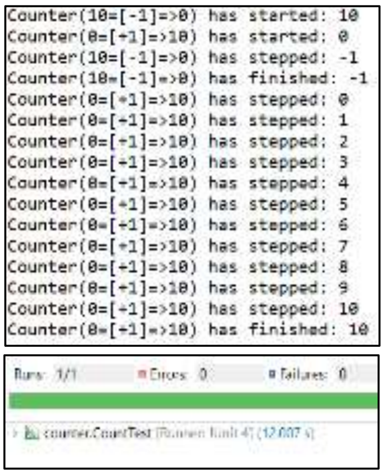
I'm sure this method is the one Hugh was asking for as there was nowhere in the question asking to synchronize the threads.

The console output for the three tests using the Asynchronous runSet method is shown bellow along with test results.



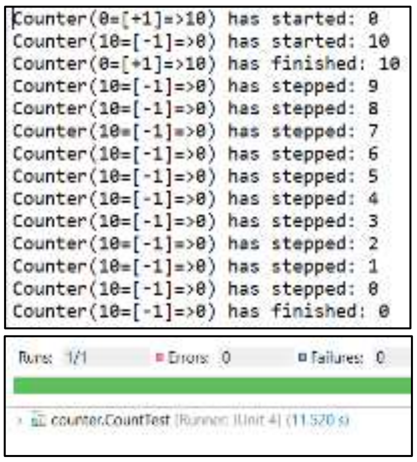
```
Counter(0=[+1]=>10) has started: 0
Counter(10=[-1]=>0) has started: 10
Counter(0=[+1]=>10) has finished: 10
Counter(10=[-1]=>0) has stepped: 9
Counter(10=[-1]=>0) has stepped: 8
Counter(10=[-1]=>0) has stepped: 7
Counter(10=[-1]=>0) has stepped: 6
Counter(10=[-1]=>0) has stepped: 5
Counter(10=[-1]=>0) has stepped: 4
Counter(10=[-1]=>0) has stepped: 3
Counter(10=[-1]=>0) has stepped: 2
Counter(10=[-1]=>0) has stepped: 1
Counter(10=[-1]=>0) has stepped: 0
Counter(10=[-1]=>0) has finished: 0
```

```
Runs: 1/1    Errors: 0    Failures: 0
counter.CountTest [Runner: JUnit 4] (10.729 s)
```



```
Counter(10=[-1]=>0) has started: 10
Counter(0=[+1]=>10) has started: 0
Counter(10=[-1]=>0) has stepped: -1
Counter(10=[-1]=>0) has finished: -1
Counter(0=[+1]=>10) has stepped: 0
Counter(0=[+1]=>10) has stepped: 1
Counter(0=[+1]=>10) has stepped: 2
Counter(0=[+1]=>10) has stepped: 3
Counter(0=[+1]=>10) has stepped: 4
Counter(0=[+1]=>10) has stepped: 5
Counter(0=[+1]=>10) has stepped: 6
Counter(0=[+1]=>10) has stepped: 7
Counter(0=[+1]=>10) has stepped: 8
Counter(0=[+1]=>10) has stepped: 9
Counter(0=[+1]=>10) has stepped: 10
Counter(0=[+1]=>10) has finished: 10
```

```
Runs: 1/1    Errors: 0    Failures: 0
counter.CountTest [Runner: JUnit 4] (12.007 s)
```



```
Counter(0=[+1]=>10) has started: 0
Counter(10=[-1]=>0) has started: 10
Counter(0=[+1]=>10) has finished: 10
Counter(10=[-1]=>0) has stepped: 9
Counter(10=[-1]=>0) has stepped: 8
Counter(10=[-1]=>0) has stepped: 7
Counter(10=[-1]=>0) has stepped: 6
Counter(10=[-1]=>0) has stepped: 5
Counter(10=[-1]=>0) has stepped: 4
Counter(10=[-1]=>0) has stepped: 3
Counter(10=[-1]=>0) has stepped: 2
Counter(10=[-1]=>0) has stepped: 1
Counter(10=[-1]=>0) has stepped: 0
Counter(10=[-1]=>0) has finished: 0
```

```
Runs: 1/1    Errors: 0    Failures: 0
counter.CountTest [Runner: JUnit 4] (11.520 s)
```

Looking at the output for the asynchronous runSet method, I can see that both the counters will start and then they will fight to increment/decrement the counter until one of the end points is reached for example in the middle console output image counter 10 → 0 became -1 which is less than the end point for that counter, so it was terminated/finished.

Questions

1. Will the test always terminates? I.e. is it certain that no matter how often you were to run the test it would always end in a finite length of time?

Synchronous runSet method:

After running the tests multiple times it can be seen visually that it has the exact same output so in these terms yes, but also looking at the run set code from the thread hash set class shows that the thread waits in the for loop for itself to finish so two counters will never be ran at the same time meaning that there is a single counter going one direction that will always reach its end point.

Asynchronous runSet method:

Again, after running the tests three times to get an average idea if it will terminate, I concluded that it will always end with the delay of 1, but if the delay becomes small enough the counters may get stuck in a loop of incrementing and decrementing the counter.

2. What is the shortest possible output for the test, in terms of the number of lines output?

Synchronous runSet method:

I have run the test three times to get an average amount of outputs which is 24 as all three tests have 24 outputs. Also, I can find that the minimum lines of output will always be 24 anyway without running the code by checking where and when the system prints to the console. Which is when:

- The counter is started.
- The counter finished.
- Every counter step.

If the first counter is going to count from 0 to 10 then this will be print once when the counter starts, then again 10 times for every step and finally once more for the count finish method. This totals up to 12 lines of output. The second count 10 to 0 is also 12 as its still counting the same amount.

Asynchronous runSet method:

Again, I have rn the test three times to get an average amount of outputs and this number comes to 14 (0.d.p). I also know that the maximum number of lines of output will be 14 as seen from the first output log on the previous page.

If both counters start in a certain way, for example the 10 → 0 counter starts (1 line of output) followed by the 0 → 10 counter (1 line of output) the current position is 10. This means that the first counter started is now at its end point, so it will finish/terminate (1 line of output). The counter will then count from 10 to 0 (10 lines of output). Finally, the last counter will end as it has reached its end point 0 (1 line of output). All of this results in 14 lines of output and no possibility of it being any smaller due to the way the counters print lines and work.

3. What is the largest possible value that the count can reach when the test is run?

Synchronous runSet method:

The largest possible value that the count can reach when this test is ran is 10 as shown from the three-test run on the previous page. The count increases once every second, when it reaches 10 the while loop dependant on the "!isFinished()" in run inside the counter class will exit causing the count to end that the value will never become larger than 10 in this specific case.

Asynchronous runSet method:

With this method I ran into a test where the output in the console showed that the count had become less than 0 while both counters are running. This means there is a possibility it could count over 10 to 11. The count is terminated/finished when this happens, but it indicates that in the largest possible value that the count can reach is 11.

4. What is the lowest possible value that the count can reach when the test is run?

Synchronous runSet method:

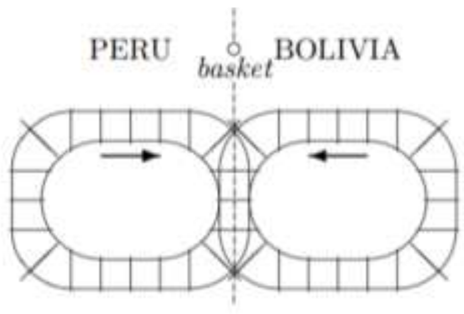
The lowest possible value that the count will ever reach is 0 for the exact same reason that the largest possible value is 10. The count will only ever go between 0 and 10 either way in the current implementation.

Asynchronous runSet method:

Again, in question three I explained that the smallest possible value in the Asynchronous case is -1 as I could not find any problems with the code I had wrote.

Dekkers Trains

After reading the lecture notes carefully I feel that I have successfully implemented her solution. I have done this by translating the pseudo code from the lecture notes depicting Dekkers algorithm.



I used the image shown above from the lecture notes to get a good understanding of how this algorithm works from my implementation. As the current train approaches the crossing (the point where railways connect) it will send a request to cross although while the next train is requesting to cross, and it is the next trains turn the current train will resign the request to cross and wait for the next train to cross. Afterwards the current train will cross. Also, after a train has crossed it will resign the request to cross and increment the shared 'trainsTurn' variable.

I have also included comments describing sections of the run train algorithm in the following code.

The algorithm I currently have working is slightly different from the pseudo code in the lecture notes because it has an else statement that waits while the requests are processed. This had to be added as one of the trains was getting stuck in the while loop for some unknown reason.

Peru.java:

```java
package railways;

import errors.RailwaySystemError;
import errors.SetUpError;
import tools.Clock;
import tools.Delay;

public class Peru extends Railway
{
	/**
	 * @throws RailwaySystemError if there is an error in constructing the delay
	 * Change the parameters of the Delay constructor in the call of the
	 * superconstructor to change the behaviour of this railway.
	 *
	 */
	public Peru() throws SetUpError
	{
		super("Peru",new Delay(0.1,0.3));
	}
```

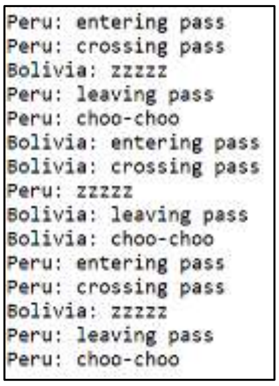
```
/**
* Run the train on the railway.
*
* The code was transformed from pseudo code from the lecture notes. Uses a
* static trainsTurn integer and boolean array to assign turns to a railway
* crossing.
*
* @author Lewis Scrivens
*/
public void runTrain() throws RailwaySystemError
{
   Clock clock = getRailwaySystem().getClock();

   // While the clock is running.
   while (!clock.timeOut())
   {
          choochoo();// complete first section as there are crossings. (No
                         danger)

          // Set the process request to true in the railway system.
          RailwaySystem.setProcessRequest(0, true);

          // While the next railway/train's request is being processed.
             (Prevention of danger)
          while (RailwaySystem.getProcessRequest(1))
          {
                 // If its not this trains turn.
                 if (RailwaySystem.getTurn() != 0)
                 {
                        // Resign request to pass and allow priority (next
                           train).
                        RailwaySystem.setProcessRequest(0, false);

                        // Wait for this trains turn until then sleep.
                        while (RailwaySystem.getTurn() != 0)
                        {
                               siesta();// Sleep
                        }

                        // Once it’s this trains turn request to pass again.
                        RailwaySystem.setProcessRequest(0, true);
                 }
                 else
                 {
                        siesta();// Sleep
                 }
          }
          // When this railway/train's process request is accepted, cross the
             pass. (Danger avoided)
          crossPass();
          // Set the trains turn to the next train in the railway system.
          RailwaySystem.setTurn(1);
          // Resign the request to cross as the train has now crossed.
          RailwaySystem.setProcessRequest(0, false);
   }
}
```

Bolivia.java:

```
package railways;

import errors.RailwaySystemError;
import errors.SetUpError;
import tools.Clock;
import tools.Delay;

public class Bolivia extends Railway
{
       /**
        * @throws RailwaySystemError if there is an error in constructing the delay
        * Change the parameters of the Delay constructor in the call of the
        * superconstructor to change the behaviour of this railway.
        */
       public Bolivia() throws SetUpError{
              super("Bolivia",new Delay(0.1,0.3), 1);// Id is 1.
       }

    /**
    * Run the train on the railway.
    *
    * The code was transformed from pseudo code from the lecture notes. Uses a
    * static trainsTurn integer and boolean array to assign turns to a railway
    * crossing.
    *
    * @author Lewis Scrivens
    */
    public void runTrain() throws RailwaySystemError
    {
       Clock clock = getRailwaySystem().getClock();

       // While the clock is running.
       while (!clock.timeOut())
       {
              choochoo();// complete first section as there are crossings. (No
                         danger)

              // Set the process request to true in the railway system.
              RailwaySystem.setProcessRequest(1, true);

              // While the next railway/train's request is being processed.
                (Prevention of danger)
              while (RailwaySystem.getProcessRequest(0))
              {
                     // If its not this trains turn.
                     if (RailwaySystem.getTurn() != 1)
                     {
                            // Resign request to pass and allow priority (next
                               train).
                            RailwaySystem.setProcessRequest(1, false);

                            // Wait for this trains turn until then sleep.
                            while (RailwaySystem.getTurn() != 1)
                            {
                                   siesta();// Sleep
                            }

                            // Once its this trains turn request to pass again.
                            RailwaySystem.setProcessRequest(1, true);
                     }
                     else
                     {
                            siesta();// Sleep (Wait while requests are processed).
                     }
              }
```

```
                // When this railway/train's process request is accepted, cross the
                   pass. (Danger avoided)
                crossPass();
                // Set the trains turn to the next train in the railway system.
                RailwaySystem.setTurn(0);
                // Resign the request to cross as the train has now crossed.
                RailwaySystem.setProcessRequest(1, false);
        }
    }
}
```

When running the railway system class, the trains take turns in crossing the pass until the time limit of the system is exceeded. The turn is initially set to 0 in the railway system class which is Peru's id, from there the algorithm handles the rest ensuring fair turn taking with no deadlock or starvation.



```
Peru: entering pass
Peru: crossing pass
Bolivia: zzzzz
Peru: leaving pass
Peru: choo-choo
Bolivia: entering pass
Bolivia: crossing pass
Peru: zzzzz
Bolivia: leaving pass
Bolivia: choo-choo
Peru: entering pass
Peru: crossing pass
Bolivia: zzzzz
Peru: leaving pass
Peru: choo-choo
```

I also have not used (id + 1) % 2 as I don't think it is needed in the current system as only 2 trains can be handled now. So, considering that the run train methods are not inherited from railway means that I can just tell the algorithm what the current and next trains id is.

The variables from the railway system class with get and set methods are shown below which the algorithm uses to work correctly.

```
        // ProcessRequest holder for both Bolivia and Peru trains.
        private static boolean[] processRequest = {false, false};

        // Initialise turn for all railways/trains to 0.
        private static int trainsTurn = 0;



        // return the trainsTurn static variable.
        public static int getTurn()
        {
                return trainsTurn;
        }
        // Set the trainsTurn static variable to the int input.
        public static void setTurn(int turn)
        {
                RailwaySystem.trainsTurn = turn;
        }
        // return the process request boolean at index id.
        public static boolean getProcessRequest(int id)
        {
                return processRequest[id];
        }
        // Set the process request at index id to the boolean processRequest.
        public static void setProcessRequest(int id, boolean processRequest)
        {
                RailwaySystem.processRequest[id] = processRequest;
        }
```

Semaphores

I have found the variables denoted in the logbook question as ‘critSec.P()’ and ‘noElts.P()’ which are said to have essential positioning meaning that the system will not work with them in reversed positions. The location of these variables is in the buffer classes get method. They are denoted as the following.

```
        noOfElements.poll();
        criticalSection.poll();
```

Flipping the positioning of these methods in the get method causes an error. The error messages are: “Buffer error: Buffer has closed – cannot get data item from it” and “Buffer error: Buffer has closed – cannot add 0 to it” and then the system terminates.

What’s happening here is that the put and get methods are being ran while the buffer is still open as it has not yet been closed due the critical section being polled before the number of elements. To start with the number of elements in the buffer is 0 and the poll of critical section before the number of elements will reduce the value below 0 causing the system to break and enter deadlock.

When the critical section acquires the semaphore before the number of elements it has not yet made sure there is one or more data items in the buffer and without this it runs into the error as the buffer is being accessed while it is still open.

The put method in the buffer class contains a similar set of variables.

```
        noOfSpaces.poll();
        criticalSection.poll();
```

When switching these two method calls around there are no errors thrown across the system so there is no specific order as stated in section 1 of the logbook questions.

I have run the code multiple times with the two method calls switched around and there are no errors showing up bellow is console output from running the code with the variables in reversed positioning. This console output proves that there is no essential positioning.

Console Output		Continuation of Console Output
`Buffer size was missing.`		`Buffer: 7 added, 1 item in the buffer`
`Using a buffer size of 10`		`Producer: Added 7 to the buffer`
`Buffer run time was missing.`		`Buffer: 7 removed, 0 items in the`
`Using a buffer run time of`		`buffer`
`10.0second(s).`		`Consumer: Retrieved 7 from the buffer`
`First delay argument was missing.`		`Consumer: Retrieving data item`
`Using a delay of 1.0second(s).`		`Producer: adding 8`
`Second delay argument was missing.`		`Buffer: 8 added, 1 item in the buffer`
`Using a delay of 1.0second(s).`		`Producer: Added 8 to the buffer`
`Buffer can hold up to 10 elements`		`Buffer: 8 removed, 0 items in the`
`System will run for 10.0s`		`buffer`
`System will start with producer taking`		`Consumer: Retrieved 8 from the buffer`
`up to 1.0s for each buffer access`		`Producer: adding 9`
`and consumer taking up to 1.0s`		`Buffer: 9 added, 1 item in the buffer`
`System will then change to producer`		`Producer: Added 9 to the buffer`
`taking up to 1.0s`		`Consumer: Retrieving data item`
`and consumer taking up to 1.0s for`		`Buffer: 9 removed, 0 items in the`
`each buffer access`		`buffer`
`Consumer: Retrieving data item`		`Consumer: Retrieved 9 from the buffer`
`Producer: adding 0`		`Producer: adding 10`
`Buffer: 0 added, 1 item in the buffer`		`Buffer: 10 added, 1 item in the buffer`

```
Producer: Added 0 to the buffer
Buffer: 0 removed, 0 items in the
buffer
Consumer: Retrieved 0 from the buffer
Producer: adding 1
Buffer: 1 added, 1 item in the buffer
Producer: Added 1 to the buffer
Consumer: Retrieving data item
Buffer: 1 removed, 0 items in the
buffer
Consumer: Retrieved 1 from the buffer
Producer: adding 2
Buffer: 2 added, 1 item in the buffer
Producer: Added 2 to the buffer
Consumer: Retrieving data item
Buffer: 2 removed, 0 items in the
buffer
Consumer: Retrieved 2 from the buffer
Producer: adding 3
Buffer: 3 added, 1 item in the buffer
Producer: Added 3 to the buffer
Consumer: Retrieving data item
Buffer: 3 removed, 0 items in the
buffer
Consumer: Retrieved 3 from the buffer
Producer: adding 4
Buffer: 4 added, 1 item in the buffer
Producer: Added 4 to the buffer
Consumer: Retrieving data item
Buffer: 4 removed, 0 items in the
buffer
Consumer: Retrieved 4 from the buffer
Consumer: Retrieving data item
Producer: adding 5
Buffer: 5 added, 1 item in the buffer
Producer: Added 5 to the buffer
Buffer: 5 removed, 0 items in the
buffer
Consumer: Retrieved 5 from the buffer
Consumer: Retrieving data item
Producer: adding 6
Buffer: 6 added, 1 item in the buffer
Producer: Added 6 to the buffer
Buffer: 6 removed, 0 items in the
buffer
Consumer: Retrieved 6 from the buffer
Consumer: Retrieving data item
Producer: adding 7
```

```
Producer: Added 10 to the buffer
Consumer: Retrieving data item
Buffer: 10 removed, 0 items in the
buffer
Consumer: Retrieved 10 from the buffer
Producer: adding 11
Buffer: 11 added, 1 item in the buffer
Producer: Added 11 to the buffer
Consumer: Retrieving data item
Buffer: 11 removed, 0 items in the
buffer
Consumer: Retrieved 11 from the buffer
Consumer: Retrieving data item
Producer: adding 12
Buffer: 12 added, 1 item in the buffer
Producer: Added 12 to the buffer
Buffer: 12 removed, 0 items in the
buffer
Consumer: Retrieved 12 from the buffer
Consumer: Retrieving data item
Producer: adding 13
Buffer: 13 added, 1 item in the buffer
Producer: Added 13 to the buffer
Buffer: 13 removed, 0 items in the
buffer
Consumer: Retrieved 13 from the buffer
Producer: adding 14
Buffer: 14 added, 1 item in the buffer
Producer: Added 14 to the buffer
Consumer: Retrieving data item
Buffer: 14 removed, 0 items in the
buffer
Consumer: Retrieved 14 from the buffer
Producer: adding 15
Buffer: 15 added, 1 item in the buffer
Producer: Added 15 to the buffer
Consumer: Retrieving data item
Buffer: 15 removed, 0 items in the
buffer
Consumer: Retrieved 15 from the buffer
Producer: adding 16
Buffer: 16 added, 1 item in the buffer
Producer: Added 16 to the buffer
Consumer has finished
Producer has finished
System terminated
```

The put method in the buffer class seems to work by checking first if there is a space, if not then wait/sleep until told otherwise (when 'noOfSpaces' is released), then continue to check the critical section if the buffer is available for access otherwise wait/sleep again until told otherwise. Finally put an item into the buffer and then any further errors will be caught there. Afterwards the critical section is released and made available followed by the number of elements stating that there are now one or more elements in the buffer.

Monitors

I have implemented a lock resource manager using code example from the lecture notes and the hint to assign a condition object to each priority. I have included comments throughout the code which explain the key areas. The code can be seen bellow.

From my understanding the conditions array created to hold each priority is initialised into the lock as new conditions, then when a user tries to access the resource the shared lock is locked to prevent multiple accesses at once, and the individual conditions are put to sleep until the resource is released. On release a new priority is found, specifically one that is waiting to access the resource. The new priority is then waking up and repeats the process.

LockResourceManager.java:

```
package resourceManager;

// Use of conditions and locks.
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * A resource manager that makes use of locks and conditions, I used
 * some of the code examples for Hugh's lecture to help me implement
 * this class. The lock resource manager initialises a condition linked
 * to a shared lock for each priority. Locks are used to ensure that
 * only one process can request or release at once.
 *
 * @author Lewis Scrivens
 * @version February 2018
 *
 */

public class LockResourceManager extends BasicResourceManager
{

       // Initialising conditions and lock.
       final Lock lock = new ReentrantLock();
       // Create an array of conditions for each priority.
       final Condition[] conditions = new Condition[NO_OF_PRIORITIES];
       // Boolean to prevent multiple accesses at once.
       boolean resourceLocked;

       public LockResourceManager(Resource resource, int maxUses) throws
                                                                  ResourceError
       {
              // Calling constructor for the basic resource manager.
           super(resource, maxUses);

           // Initialise resource to unlocked.
           resourceLocked = false;

              // Initialise the conditions array creating new conditions for each
                 priority.
              for (int priority = 0; priority < NO_OF_PRIORITIES; priority++)
              {
                     // For each of the conditions initialise them.
                     conditions[priority] = lock.newCondition();
              }
       }
```

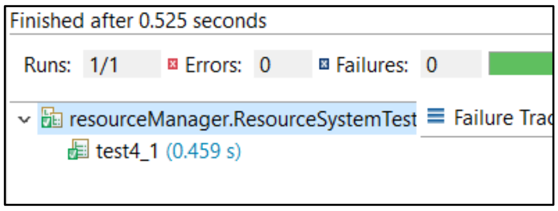
```
        @Override
        public void requestResource(int priority) throws ResourceError
        {
                lock.lock();// Lock the method.

                while (resourceLocked)
                {
                        // Increase the number waiting as this process will now wait.
                        increaseNumberWaiting(priority);

                        try
                        {
                                // Sleep while resource is locked.
                                conditions[priority].await();
                        }
                        catch (InterruptedException e)
                        {
                                throw new ResourceError("The proccess was interupted
                                                        while waiting.");
                        }
                }
                // Lock the resource for other priorities.
                resourceLocked = true;

                lock.unlock();// Unlock the method.
        }

        @Override
        public int releaseResource()
        {
                lock.lock();// Lock the method.

                // Initialise priority to non-waiting as it will not be changed if no
                   processes are waiting.
                int highestPriority = NONE_WAITING;

                // Find a priority with the highest number waiting.
                for (int priority = 0; priority < NO_OF_PRIORITIES; priority++)
                {
                        if (getNumberWaiting(priority) > 0)
                        {
                                highestPriority = priority;// Set new highest priority.
                        }
                }

                // If a priority has waiting processes.
                if (highestPriority != NONE_WAITING)
                {
                        // Decrease number waiting as a priority is being woken.
                        decreaseNumberWaiting(highestPriority);

                        // Wake up any conditions that are waiting.
                        conditions[highestPriority].signal();
                }

                // The resource is no longer locked.
                resourceLocked = false;

                // Unlock the method.
                lock.unlock();

                // Once done return the priority or none_waiting.
                return highestPriority;
        }
}
```

Testing

After implementing the lock resource manager, I used the pre-made test to check if the new resource manager was working as intended. Looking at the console I can see that each user is accessing the resource one by one. The first process to start and then request a resource will be the first granted access, this will then loop until the resource is exhausted (Max uses is reached). The console output can be seen bellow when running the following test from the resource system test class.

```
    /**
     * Four users sharing one resource.
     * @author Hugh.
     */
      public void test4_1() throws ResourceError
      {
             ResourceSystem resource1 = new ResourceSystem();
             // The resource - may be used up to 10 times
             resource1.addResource("A", 10);
             // User 1 uses the resource up to 1/10 second each time.
             resource1.addUser("1",0.1);
             // User 2 uses the resource up to 1/10 second each time.
             resource1.addUser("2",0.1);
             // User 3 uses the resource up to 1/5 second each time.
             resource1.addUser("3",0.2);
             // User 4 uses the resource up to 1/5 second each time.
             resource1.addUser("4",0.2);
             // Start the resource system.
             resource1.run();
      }
```

Result:


Finished after 0.525 seconds

Runs: 1/1 Errors: 0 Failures: 0

resourceManager.ResourceSystemTest
test4_1 (0.459 s)

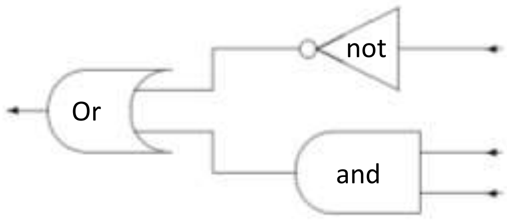
Console output:

```
Starting Process "1" (priority: 0)
Starting Process "4" (priority: 0)
Starting Process "2" (priority: 0)
Starting Process "3" (priority: 0)
Process "4" (priority: 3) is requesting resource "A"
Process "1" (priority: 10) is requesting resource "A"
Process "4" (priority: 3) gained access to resource "A"
Process "3" (priority: 1) is requesting resource "A"
4 is using resource "A"
Process "2" (priority: 1) is requesting resource "A"
4 has finished using resource "A"
resource "A" has 9 uses left
Process "4" (priority: 3) released resource "A", to a process with priority 10
Process "1" (priority: 10) gained access to resource "A"
1 is using resource "A"
1 has finished using resource "A"
resource "A" has 8 uses left
Process "1" (priority: 10) released resource "A", to a process with priority 1
Process "3" (priority: 1) gained access to resource "A"
3 is using resource "A"
3 has finished using resource "A"
resource "A" has 7 uses left
Process "3" (priority: 1) released resource "A", to a process with priority 1
Process "2" (priority: 1) gained access to resource "A"
2 is using resource "A"
2 has finished using resource "A"
resource "A" has 6 uses left
Process "2" (priority: 1) released resource "A", there were no waiting processes
Process "1" (priority: 7) is requesting resource "A"
Process "1" (priority: 7) gained access to resource "A"
1 is using resource "A"
1 has finished using resource "A"
resource "A" has 5 uses left
Process "1" (priority: 7) released resource "A", there were no waiting processes
Process "2" (priority: 4) is requesting resource "A"
Process "2" (priority: 4) gained access to resource "A"
2 is using resource "A"
2 has finished using resource "A"
resource "A" has 4 uses left
Process "2" (priority: 4) released resource "A", there were no waiting processes
Process "1" (priority: 5) is requesting resource "A"
Process "1" (priority: 5) gained access to resource "A"
1 is using resource "A"
Process "4" (priority: 9) is requesting resource "A"
Process "2" (priority: 6) is requesting resource "A"
1 has finished using resource "A"
resource "A" has 3 uses left
Process "4" (priority: 9) gained access to resource "A"
4 is using resource "A"
Process "1" (priority: 5) released resource "A", to a process with priority 9
Process "3" (priority: 5) is requesting resource "A"
Process "1" (priority: 4) is requesting resource "A"
4 has finished using resource "A"
resource "A" has 2 uses left
Process "2" (priority: 6) gained access to resource "A"
2 is using resource "A"
Process "4" (priority: 9) released resource "A", to a process with priority 6
2 has finished using resource "A"
resource "A" has 1 uses left
Process "2" (priority: 6) released resource "A", to a process with priority 5
Process "3" (priority: 5) gained access to resource "A"
3 is using resource "A"
Process "4" (priority: 10) is requesting resource "A"
Process "2" (priority: 8) is requesting resource "A"
3 has finished using resource "A"
resource "A" has 0 uses left
Process "3" (priority: 5) released resource "A", to a process with priority 10
Process "4" (priority: 10) gained access to resource "A"
Process "4" (priority: 10) cannot use resource "A" as the resource is exhausted
resource "A" has 0 uses left
Process "4" (priority: 10) released resource "A", to a process with priority 8
Process "2" (priority: 8) gained access to resource "A"
Process "2" (priority: 8) cannot use resource "A" as the resource is exhausted
resource "A" has 0 uses left
Process "2" (priority: 8) released resource "A", to a process with priority 4
Process "1" (priority: 4) gained access to resource "A"
Process "1" (priority: 4) cannot use resource "A" as the resource is exhausted
resource "A" has 0 uses left
Process "1" (priority: 4) released resource "A", there were no waiting processes
Process "1" (priority: 4) has finished
Process "4" (priority: 10) has finished
Process "2" (priority: 8) has finished
Process "3" (priority: 5) has finished
All processes finished
```

As you can see from the console output the first process to gain access was the first one to request then on release it selected the next resource waiting with the highest amount of priorities. This resource is then unlocked and relocked by the new user. This continues to happen until there are no uses left and the processes will finish, and the system ends.

Quantum Computing - Circuits and Matrices

I have been asked to represent the following quantum circuit as a matrix. The circuit bellow is constructed from an "and" gate, "not" gate and a "or" gate.



A: Not matrix = $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ **B: And** matrix = $\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ **C: Or** matrix = $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$

Note: I modelled the or matrix by doing the practical exercise to do.

The Quantum circuit in matrix form will be modelled using the following formulae. C * (A ⊗ B). This equation for the matrix was built using the following rules from the lecture:

- Parallel circuits are modelled into a matrix by finding the tensor product of the two. (⊗)
- Circuit connections are modelled by multiplication between matrixes.

Using the equation C * (A ⊗ B) I will now show the steps of how I modelled the matrix.

$$(A \otimes B) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & 1\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ 1\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & 0\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Note: Tensor product is found by multiplying each item in a matrix by the second matrix.

$$C \text{ x } (A \otimes B) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

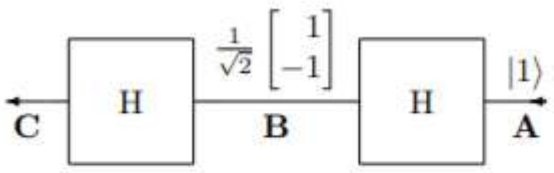
Note: Tensor product is found by multiplying each item in a matrix by the second matrix.

Final Matrix representation = $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$

Using the following link: https://www.dcode.fr/tools-list#matrix accessed on 15.03.2018, I was able to test the matrix I obtained ensuring that it was the correct representation. This was done after doing all of the calculations above as I wanted to work it out manually.

Quantum Computing – Hadamard Gate

In this section of the logbook I will be talking about what happens to a matrix when passing it through two Hadamard gates, also I will discuss the state of the qubit at the different points in a sequence and matrix superpositions.



Hadamard gate matrix: $\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$

Note: Every time a matrix passes through a Hadamard gate it is charged to the tensor product of itself and the Hadamard gate matrix shown above.

If '|ψi' is a pure state, for example when |ψi = |0i it will manipulate matrices at the points A -> B -> C. The starting matrix before entering the sequence of Hadamard gates is $\begin{bmatrix}1\\0\end{bmatrix}$ in this case. Bellow I will show the ways this matrix is manipulated when passing through the sequence:

When: |ψi = |0i

Starting point: A → **Middle point**: B → **End point**: C

$\begin{bmatrix}1\\0\end{bmatrix}$

$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix} * \begin{bmatrix}1\\0\end{bmatrix}$

$= \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}0.707106\ldots\\0.707106\ldots\end{bmatrix}$

$= \begin{bmatrix}0.7071\\0.7071\end{bmatrix}$ 4.d.p

$= \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix} * \begin{bmatrix}0.7071\\0.7071\end{bmatrix}$

$= \frac{1}{\sqrt{2}}\begin{bmatrix}1.4142\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$

When: |ψi = |1i

Starting point: A → **Middle point**: B → **End point**: C

$\begin{bmatrix}0\\1\end{bmatrix}$

$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix} * \begin{bmatrix}0\\1\end{bmatrix}$

$= \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix} = \begin{bmatrix}0.707106\ldots\\-0.707106\ldots\end{bmatrix}$

$= \begin{bmatrix}0.7071\\-0.7071\end{bmatrix}$ 4.d.p

$= \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix} * \begin{bmatrix}0.7071\\-0.7071\end{bmatrix}$

$= \frac{1}{\sqrt{2}}\begin{bmatrix}1.4142\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$

In both cases after an input matrix passes through the gate, the matrix appears become the product of itself and the Hadamard's gate's matrix. After passing through a second time the input matrix returns to its initial value. It appears that the qubits enter a superposition where they equal both 1 and 0 after passing through the first gate, then passing through the second gate will force the qubit chooses a final state.

Self-Assessment: Week 1 - 8

Week	Overall	Documentation	Structure	Names	Test	Functionality
1 - 2: Search Timer	A-	B+	A	A	B	A
3 - 4: Generic Swap	B+	A	A	A	B-	B
5: Sorting	A	A-	A	A	B	A-
6: Linked Lists	A	A-	A-	A	A	A
8: Binary Trees	A	A	A	A	A	A

Evidence and Justification:

Any evidence this week can be seen by observing the sections that are being self-assessed and all of my justification for the grades I have given to myself can be seen bellow.

Search Timer:

In these weeks I have given myself an A- overall, the test grade pulls the rest down. I feel that my documentation in the search timer classes of my creation does not have many problems. Being honest with myself I feel like I could have written the comments in more detail during these weeks as well as the descriptions of the methods at the top of each class.

The structure of the clever random listing class was good in my opinion, although across this whole document I have opened methods on the next line like the following.

```
void method()
{
}
```

I did this as I find it much easier to read the methods when they are more fleshed out. Names I have given the methods, variables and classes all fit will the java naming conventions for the current week, so I could see no reason not to give myself a high mark in this section.

As for testing my clever random listing class I feel like I could have done much better than just making use of Hugh's simple random listing test class to help build my own. Although the test method I used was not my own, I feel like I did an excellent job of using these tests to construct tables of data used to compare Hugh's simple random listing to my clever random listing. This data made it very easy to find a pattern to the speed of the algorithm.

As for the functionality of these tests again I feel like they functioned perfectly for my purpose in finding the difference in speed between the original and new class. I gave myself an A- overall as the only thing really pulling me down in this week was the test class as it is a copy of the simple random listing test class.

Generic swap:

In these weeks I have given myself a B+, I would say the low point of what I was asked for during this week would be the test cases I have written. The documentation during these two weeks was very good as I had written entire class documentation at the top as well as explaining all the important lines in both the swap method and the test case for that method.

The structure of my code again was perfectly fine in my opinion. The visual structure is neat as I went over all the code to ensure it was easy to follow and didn't have unnecessary variables. Names I have given the methods, variables and classes all fit will the java naming conventions for the current week, so I could see no reason not to give myself a high mark in this section.

Again, testing for this week is where I fell short, I feel that I could have implemented more testing methods this week other than just the one. Although I still think I made effective use of the testing suite to ensure the swap was happening which is also the reason that the functionality is graded equally as the code performs very well but only in the single test that I have done.

Sorting:

This week I have given myself an A, I believe I showed a wide range of different skills including the examination of data from the different algorithms. The documentation in the classes I have written this week was good as I have given detailed explanations across all the algorithms I have implemented along with class explanations at the top of each class.

The structure of my code was perfectly fine in my opinion, the visual structure is laid out the same way I have laid out every other week's code, this week I cannot spot any unnecessary variables. Names I have use for variables, methods and classes during this week's exercises have been clear what there intended use is. Because of this I could see no reason not to give myself a high mark.

Testing on the sorting algorithms this week could have been much better as not only could have I made my own test methods, but I could have extended the sort tester class instead of creating duplicate code in the system, I still think it was an effective use for gathering data from the two implemented algorithms. Finally, the functionality of the test cases this week was very good for what I needed, they gave me data that I could then interpret in a graph and gather equations from for prediction of time taken. Because of this I feel like I got an A-.

Linked Lists:

This week I have given myself an A, I have done this because I believe I did very well in understanding and presenting my work with correct documentation, naming, structure and a wide range of testing to ensure my code worked as intended. The documentation during this week was very good as I believe I have walked through each of the key areas in the singly linked list methods, I have also provided good documentation on my test methods this week. Class descriptions can also be seen at the top of all the required classes for this week's exercises.

The structure of this week's code was good as everything was laid out to be visually easy to read through, some spaces and empty lines can be seen but this was to give it a more spaced out look for the logbook. Naming for the methods, variables and classes created for this exercise was also very good and comply with the java naming conventions, because of this I could see no reason not to give myself an A.

Testing done during this week was also very good as I made effective use of testing suites and presented a wide range of test methods for everything coded for the logbook. These tests have been executed and previewed in a screenshot, so I believe that I have made very good use of the test cases I have written. Functionality this week was also very good as all the test cases passed and helped me in the process of making this work as intended, evidence of this can be seen in the single linked list testing section.

Binary Trees:

This week I have again given myself an A, I believe I have a very good understanding of binary trees and how each section of code works, including the traversals of a tree. I believe I have show this in the binary tree and traversals section, I also believe I have put allot of time making effective use of test suites. The documentation during this week was very good as I have shown throughout the exercises shown in the binary tree section of the logbook, there are comments going through all the key areas of the binary tree implementation although I could have explained the traversal of a binary tree better.

The structure during this week was very well done which is the reason for my high grade, I feel that I have structured the code in a way it can be interpreted easier. Also, there are no noticeable unnecessary variables. The naming of the methods, variables and classes are all complying with the java naming conventions so again like previous weeks I see no reason not to assign myself a higher mark.

The testing during this week was also very good as I have shown effective use of test suites within java. I have implemented tests and returned to improve on them, I did this for the traversal testing class as stated previously in the logbook. I made use of the results from the traversal tests to ensure that they are correct, comparisons were made using an online resource of what the traversals should be. Finally, the binary tree test class also shows in-depth testing of different areas of the code, this was also done to ensure the binary tree was behaving the way it should.

Functionality of the testing done for this week's exercises was also very good as it helped me complete the implementation of the 'BTree' class. Test classes ensured me that the binary tree's insert method works, and the code passes all these test cases (evidence in the binary tree testing section).

Self-Assessment: Week 9 - 11

Week	Overall	Documentation	Structure	Names	Test	Functionality
9: Hashtables	A	▨	▨	▨	▨	▨
10: Graph Traversals	A	A	A	A	A-	A
11: Topological Sorts	A	A	A	A	A-	A

Evidence and Justification:

Overall this term I believe I have done very well and improved my methods of taking my code and presenting it in logbook format. Any evidence this week can be seen by observing the sections that are being self-assessed and all my justification for the grades I have given to myself can be seen bellow.

Hashtables:

During this week I have given myself an A overall. I believe this because I have given a good explanation using step by step images to explain how Hashtables work including the way they are inserted into the array, why the length increases and how values can move around depending on the size of the array. Also, I have come up with imaginative ways to present my understanding in this document.

Graph Traversals:

This week I have graded myself with an A, I have fully implemented the graph traversal class with very good documentation, this includes method and class descriptions, also I have commented any key areas in the code.

The structure of the code made during this week's exercises was very good and is complaint to the format submitted into the logbook up to this point, I believe I have shown good consistency throughout and presentation in a way that the code is easily interpreted. Naming methods, variables and classes in the graph project was also very good as all names are compliant with the java naming conventions.

Testing for the traversal class was also good this week as I have shows effective use of testing suites, as well as evidence that the traversal throughout the graph is working in the correct way. I have also provided very good documentation on these test cases. Finally, the functionality of this week also very good as the evidence given from passing the test shows that my code performs in the correct way.

Topological Sorts:

This week I have graded myself at an A again, I feel that I have fully implemented the topological sorting algorithms as required in the marking scheme. I have also graded myself high as I believe I have given a very good explanation of how the reference counting sort works as well as provided a range of test cases that ensure this sort works correctly. I also had the lecture notes to compare my sort to get it working correctly.

My documentation in the classes created for this week's exercise was very good, I included comments that described sections of the algorithm as well as test classes. Also, I have described the nature of each class I have wrote at the top of the code. The structure I have given the code this week is the same as every week inside of this logbook, I did this for consistency throughout. Also, I believe I have obtained a good grade for structure as there are no noticeable variables that are unnecessary.

Names for methods, variables and classes in this week's coding exercises were very good and clearly related to their individual uses, these names are also compliant with the java naming conventions, so I see no reason not to give myself a higher grade.

Testing the classes during this week has also been done to a good standard as there are test cases for all the critical parts for the reference counting sort class to work. These test classes and interpretation of test results can be seen in the topological sorts section of the logbook. Functionality between my code and these test is very good also, the way my code performs in this test is perfect, the results show the tests are passed which is also why testing has helped me improve this class.

Self-Assessment: Week 13 - 16

Week	Overall	Documentation	Structure	Names	Test	Functionality
13: Counter Behaviour	A	▨	▨	▨	▨	▨
14: Dekker Trains	A-	A	B+	A	B	A
15: Semaphore Behaviour	B-	▨	▨	▨	▨	▨
16: Locks and Conditions	B	B+	B+	A	B	B

Evidence and Justification:

Any evidence this week can be seen by observing the sections that are being self-assessed and all my justification for the grades I have given to myself can be seen bellow.

Counter Behaviour:

I have given myself an A overall during this week, I have given myself an A as I feel that I have given a very good explanation on why I think that the counter acts in a certain way in both synchronized and asynchronized run set methods. Evidence is given for my strong understanding of the behaviour of the counter and what's happening internally in the counter section of the logbook.

Dekker Trains:

This week I have graded myself at an A-, I have done this because I feel that I have implemented the correct run train method as requested by the logbook question. I used the pseudo code in the lecture notes to create my run train method, I then used my understanding again from the lecture to model a method that used both turns and process requests keeping this method in terms of trains which was also a request.

The documentation for this week's classes was very good as I have walked through each line of the methods I have wrote describing them using comments, I have also included class descriptions at the top of them. Due to these being included in this section of the logbook I could not see many areas of improvement, therefore I have graded myself with an A-, as maybe I could give more details on the way the siesta method works. The structure of my code this week was the same as usual, very good. This is because I keep a consistent format throughout.

Naming the methods, variable and classes during this week's exercises was done very well. I believe this as the names are all compliant with the java naming conventions. Testing during this week was not done by myself, the code at least. The test method I used was running the railway system and observing the output. I did this as there was no requests for test class and I was and still am stuck on ways to test this runTrain method other than observations.

Functionality of using this method to check that the method was working as intended was good, although I have given myself a lower grade as I believe with visual observations there is always room for human error, so I could have made a mistake.

Semaphore Behaviour:

I have given myself a B- overall during this week of the assignment. I really struggled through this week as I did not make it to the lecture and could not find it when looking for the lecture capture on blackboard. I feel like a did a decent job of looking, location and discussing an error in the system by swapping the elements. I concluded that I had caused deadlock by switching them, but this could be incorrect as I could have done more testing to check for this. I feel that I have obtained the incorrect error as it has something to do with the buffer closing which we were told to ignore. This is another reason why I have given myself a lower grade.

Locks and Conditions:

This week have given myself a B, I have done this as I believe I have only a good understanding of how these methods work, I think my documentation could use some more work and understanding in the code comments and descriptions. I also have reused Hugh's testing code for any resource manager. I also believe that some of the areas of code I have written could contain unnecessary variables which needs further testing that I am unsure how to do.

The documentation for this week's class was good as I walked through the code as well as I could understand it as well as described how the class worked in the locks and conditions section of the logbook. I feel like I could improve this documentation with further testing and understanding of how locks and conditions work.

The structure of my code throughout the week has been very good and it was in this exercises code although it could have been cleaned up before inserting into the logbook. This is the reason I gave myself a lower grad than other weeks. This includes the fact that there may be some unnecessary variables but like I said, it would require further testing that I am unsure how to do.

Naming methods, variables and classes was done to the best of my understanding, but I do believe that all names relate to the use of the variable/method. The names are also complaint with the java naming conventions. Testing in this class was also difficult so I ended up reusing Hugh's test method for the resource manager which is the reason for the lower grade, although it is not lower as I used these test methods to ensure that the lock resource manager was working, I ensured it was working by checking it was acting similarly to other resource managers.

Finally, the functionality of these tests when using them on my lock resource manager class was good, it works for my purpose and helped me prove that my resource manager was working the way it was intended to, Although I came to that conclusion from comparisons to the already implemented resource managers which is another reason for the overall low grade in this section.

Self-Assessment: Week 17 - 20

Week	Overall	Criteria
17: Modelling Circuits	B+	Have you derived a matrix for a half-adder? Have you correctly applied the matrix methods for constructing sequential and parallel circuits? Have you explained/justified/proved your derivation? Have you tested it on (matrix representations of) various inputs?
20: Quantum Computing	B+	Have you fully analysed the values that will appear at points A, B, and C in the circuit? Have you discussed the relationship between the values appearing at A and C? Have you considered what the implications of a purely probabilistic model would be for maintaining this relationship?

Evidence and Justification:

Any evidence this week can be seen by observing the sections that are being self-assessed and all my justification for the grades I have given to myself can be seen bellow.

Modelling Circuits:

For this week I have graded myself a B+, I have done this as I feel that I have passed most of the criteria provided. I feel that I have derived the correct matrix from the circuit by using methods to derive parallel circuits then sequential ones. I have proved my justification by including all my calculations and methods for deriving the matrix. Although I do not believe I have tested it on matrix representation of various inputs being the reason for the lower grade.

Quantum Computing:

For this week I have graded myself with a B+, I have done this as I believe I could have a much better understanding of this section with more research. I have also done this because I believe I have included all the criteria in the week 20 section, except for considering the implications of a purely probabilistic model as I do not clearly understand the implications. More research would have to be done and I plan to investigate this in my spare time.

General-Assessment:

Assessment Criterion	Grade
Answers to flagged logbook questions	A-
Answers to other practical questions	C
Other practical work (additional exercises you have undertaken of your own initiative)	▨
Understanding of the module material to date	B+
Level of self-reflection & evaluation	A-
Participation in timetabled activities	C-
Time spent outside timetabled classes (guideline is two hours self-study for each hour of timetabled study)	A

Evidence and Justification:

Any evidence across the entire logbook can be seen throughout the logbook exercises to date.

I believe that I have given acceptable grades in this section. I have answered all logbook questions with few exceptions, so I believe that my grade for this is an A- as there are some areas in the logbook where questions or my understanding of the topic could be lacking. I have answered most logbook questions some of which are included in certain sections of the logbook for example the or matrix in week 17 needed to be done in the other practical questions so there is evidence that I have done this. I have not completed all the practical questions maybe the majority, so I believe I have obtained a C.

Other practical work is not applicable as I have not completed any as my focus has been on the logbook questions and my understanding of them. My understanding of the module to date is a B+ as there are exceptions as stated in previous assessments, looking across my logbook evidence can be seen I have a clear understanding of most of the content. I believe my level of self-reflection has been very good throughout this logbooks development although, I am limited to the level of knowledge of each section, but I still believe I have been honest with myself.

My participation in timetables activities was very poor at the beginning of the year as discussed with the university I could not attend on Fridays due to work and due to the fact that I live in Manchester and commute meaning that I had to make acceptations, towards the end of the year my attendance to the lectures has grown but tutorials was still very low again because of the cost to commute. As for my time spent outside of timetabled classes for self-study can be seen on my GitHub commit logs, my access logs to the blackboard resources and lecture capture and this logbook.