LOGBOOK

Algorithms – Processes and data

J.Pritchard U1661665
U1661665@unimail.hud.ac.uk

Practical 4 (Week 7)

(Logbook) Question 1: Implement the List<T> interface, using singly linked lists.

Code Listing.

```
package linkedList.list;

import linkedList.node.SingleLinkNode;

/**
 * Created by u1661665(Joshua Pritchard) on 14/11/2018.
 * Version: 19/11/2018
 */
public class SingleLinkList<T> extends BasicList<SingleLinkNode<T>, T> implements List<T>
{
    /**
     * Used to record the size of the list.
     */
    private int size;

    /**
     * Create an empty SingleLinkList.
     */
    public SingleLinkList()
    {
        root = null;
        size = 0;
    }

    /**
     * Create a SingleLinkList with one node (the value passed in).
     *
     * @param value the value with which to create the root node of the SingleLinkList.
     */
    public SingleLinkList(T value)
    {
        root = new SingleLinkNode<T>(value);
        size = 1;
    }
```

```java
/**
 * Returns the size of the list.
 *
 * @return the size of the list.
 */
public int getSize()
{
    return this.size;
}
```

```java
    /**
     * Adds a new Value to the SingleLinkList at the index specified.
     *
     * @param index the index at which the new entry should be added.
     * @param value the value to be added.
     * @throws ListAccessError if the index is invalid in respect to the size of the list.
     */
    @Override
    public void add(int index, T value) throws ListAccessError
    {
        //Make sure the index trying to be accessed is not invalid.
        if(index < 0 || index > size) { throw new ListAccessError("Invalid index location " + index); }

        //Increase the recorded size of the list.
        size++;

        //Special case for adding to the root.
        if(index == 0)
        {
            setRoot(new SingleLinkNode<T>(value, getRoot()));
            return;
        }

        //Initial states for finding the previous and next elements (the case for adding at index 1)
        SingleLinkNode<T> previous = getRoot();

        /*
        If not adding at index 2:
        . Go through the list until the correct element prior to the element to be added is found.
        . Then find the element after this.
        . These two nodes will be used as the previous and next element for the element to be added.
         */
        for(int x = 2; x <= index; x++)
        {
            previous = previous.getNext();
        }
        SingleLinkNode<T> nextAfterNew = previous.getNext();

        //Create the new listNode and set up its next element.
        SingleLinkNode<T> newNode = new SingleLinkNode<>(value, nextAfterNew);

        //Correct the next element of the previousNode.
        previous.setNext(newNode);
    }
```

```java
/**
 * Remove the element at the specified index and return the value.
 *
 * @param index the index of the entry to be removed.
 * @return the value of the element removed.
 * @throws ListAccessError if the index is invalid in respect to the size of the list.
 */
@Override
public T remove(int index) throws ListAccessError {
    //Make sure the index trying to be accessed is not invalid.
    if(index < 0 || index >= size) { throw new ListAccessError("Invalid index location " + index); }

    //Initial states for finding the previous and the new index for the previous to point to
    //(the case for removing the root node).
    SingleLinkNode<T> previous = null;
    SingleLinkNode<T> previousNewNext = getRoot().getNext();

    //Get the value being removed before it becomes inaccessible via list modification.
    T removed = get(index);

    //The special case for removing the root node.
    if(index == 0)
    {
        setRoot(previousNewNext);
        size--;
        return removed;
    }

    //If the root node is not the one to be removed, set up basic conditions for removing an internal element.
    previous = getRoot();
    previousNewNext = previousNewNext.getNext();

    //Cycle through the list to find the correct values for previous and previousNewNext.
    for(int x = 1; x < index; x++)
    {
        previous = previous.getNext();
        previousNewNext = previousNewNext.getNext();
    }

    //Re-arrange the pointer for the previous element, and return the removed element.
    previous.setNext(previousNewNext);
    size--;
    return removed;
}
```

```java
    /**
     * Return the value of the element in the list at the index specified.
     *
     * @param index the index of the entry to be accessed.
     * @return the value of the element at the index specified.
     * @throws ListAccessError if the index is invalid in respect to the size of the list.
     */
    @Override
    public T get(int index) throws ListAccessError
    {
        //Make sure the index trying to be accessed is not invalid.
        if(index < 0 || index > size) { throw new ListAccessError("Invalid index location " + index); }

        //Set up a storage variable for the node being accessed.
        SingleLinkNode<T> get = getRoot();

        //Cycle through the list until the correct element has been found.
        for(int x = 1; x <= index; x++)
        {
            get = get.getNext();
        }

        //Return the element's value.
        return get.getValue();
    }
}
```

Test Class Code Listing.

```
package list;

/**
 * Created by u1661665(Joshua Pritchard) on 14/11/2018.
 * Version: 14/11/2018
 */

import linkedList.list.ListAccessError;
import linkedList.list.SingleLinkList;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

/**
 * A selection of test methods to test the SingleLinkList class and its methods.
 */
public class SingleLinkListTest
{
    @Test
    void testCreateSize0()
    {
        SingleLinkList<Integer> intList = new SingleLinkList<>();

        if(intList.getRoot() != null)
        {
            fail("Root is not null");
        }
        if(intList.getSize() != 0)
        {
            fail("Size is not 0");
        }
    }
```

```java
@Test
void testCreateSize1()
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(2);

    if(intList.getRoot() == null)
    {
        fail("Root is null for some reason");
    }
    if(intList.getSize() != 1)
    {
        fail("Size of list is not 1.");
    }
}

@Test
void testInitialNodeValue()
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);

    //List = {1}

    if(intList.getRoot().getValue() != 1)
    {
        fail("Root value is not 1 for some reason");
    }
}
```

```java
@Test
void testSizeWorks() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    intList.add(1, 2);
    intList.add(2, 3);
    intList.add(3, 4);

    //List = {1, 2, 3, 4}

    if(intList.getSize() != 4)
    {
        fail("Size is not 4");
    }

    intList.remove(0);
    //List = {2, 3, 4}
    if(intList.getSize() != 3)
    {
        fail("Size not correctly updated to 3");
    }

    intList.remove(1);
    //List = {2, 4}
    if(intList.getSize() != 2)
    {
        fail("Size not correctly updated to 2");
    }

    intList.remove(1);
    //List = {2}
    if(intList.getSize() != 1)
    {
        fail("Size not correctly updated to 1");
    }

    intList.remove(0);
    //List = {}
    if (intList.getSize() != 0)
    {
        fail("Size not correctly updated to 0");
    }
}
```

```java
@Test
void testAddIndexNegativeIndex() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    try
    {
        intList.add(-1, 1);
        fail("Index of -1 not caught.");
    }
    catch (ListAccessError e) {}
}

@Test
void testAddIndexGreaterThanSize() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    try
    {
        intList.add(2, 1);
        fail("Index of 2 not caught");
    }
    catch(ListAccessError e) {}
}

@Test
void testAddToEmptyListRootValue() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>();
    //List  = {}
    intList.add(0, 1);
    //List = {1}
    if(intList.getRoot().getValue() != 1)
    {
        fail("Root value is not 1 for some reason.");
    }
}
```

```java
@Test
void testAddToListSize1AtRoot() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    //List = {1}
    intList.add(0, 2);
    //List = {2, 1}
    if(intList.get(0) != 2)
    {
        fail("Root value is not 2");
    }
    if(intList.get(1) != 1)
    {
        fail("Root value of 1 was not moved up to index 1");
    }
}

@Test
void testAddToListSize1AfterRoot() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    //List = {1}
    intList.add(1, 2);
    //List = {1, 2}
    if(intList.get(0) != 1)
    {
        fail("Root value is no longer 1");
    }
    if(intList.get(1) != 2)
    {
        fail("Value of 2 not correctly added to index 1");
    }
}
```

```java
@Test
void testAddToGenericInternalIndex() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    intList.add(1, 2);
    intList.add(2, 3);
    //List = {1, 2, 3}
    if(intList.get(0) != 1 || intList.get(1) != 2 || intList.get(2) != 3)
    {
        fail("List not set up correctly for test.");
    }

    intList.add(1, 10);
    //List = {1, 10, 2, 3}
    if(intList.get(0) != 1)
    {
        fail("Root node is no longer 1");
    }

    if(intList.get(1) != 10)
    {
        fail("10 not added to index 1");
    }
    if(intList.get(2) != 2)
    {
        fail("2 not correctly moved to index 2");
    }
    if(intList.get(3) != 3)
    {
        fail("3 not correctly moved to index 3");
    }

    if(intList.getSize() != 4)
    {
        fail("Size not correctly updated to 4.");
    }
}
```

```java
@Test
void testAddToEndOfList() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    intList.add(1, 2);
    intList.add(2, 3);
    //List = {1, 2, 3}
    if(intList.get(0) != 1 || intList.get(1) != 2 || intList.get(2) != 3)
    {
        fail("List not set up correctly for test.");
    }

    intList.add(3, 10);
    //List = {1, 2, 3, 10}

    if(intList.get(2) != 3)
    {
        fail("3 no longer at correct index of 2");
    }
    if(intList.get(3) != 10)
    {
        fail("10 not added to end of list correctly.");
    }

    if(intList.getSize() != 4)
    {
        fail("Size not correctly updated to 4.");
    }
}

@Test
void testRemoveIndexNegativeIndex() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    try
    {
        intList.remove(-1);
        fail("Index of -1 not caught.");
    }
    catch (ListAccessError e) {}
}
```

```java
@Test
void testRemoveIndexEqualToSize() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    try
    {
        intList.remove(1);
        fail("Index of 1 not caught");
    }
    catch(ListAccessError e) {}
}

@Test
void testRemoveIndexGreaterThanSize() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    try
    {
        intList.remove(2);
        fail("Index of 2 not caught");
    }
    catch(ListAccessError e) {}
}

@Test
void testRemoveFromListSize1AtRoot() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    //List = {1}
    int removed = intList.remove(0);
    //List = {}
    if(removed != 1)
    {
        fail("Removed element value 1 not correctly returned.");
    }
    if(intList.getRoot() != null)
    {
        fail("root not removed correctly");
    }
}
```

```java
@Test
void testRemoveFromGenericInternalIndex() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    intList.add(1, 2);
    intList.add(2, 3);
    //List = {1, 2, 3}
    if(intList.get(0) != 1 || intList.get(1) != 2 || intList.get(2) != 3)
    {
        fail("List not set up correctly for test.");
    }

    int removed = intList.remove(1);
    //List = {1, 3}
    if(removed != 2)
    {
        fail("Removed element value 2 not correctly returned");
    }
    if(intList.get(0) != 1)
    {
        fail("Root node is no longer 1");
    }

    if(intList.get(1) != 3)
    {
        fail("3 not correctly moved down to index 1");
    }

    if(intList.getSize() != 2)
    {
        fail("Size not correctly updated to 2.");
    }
}
```

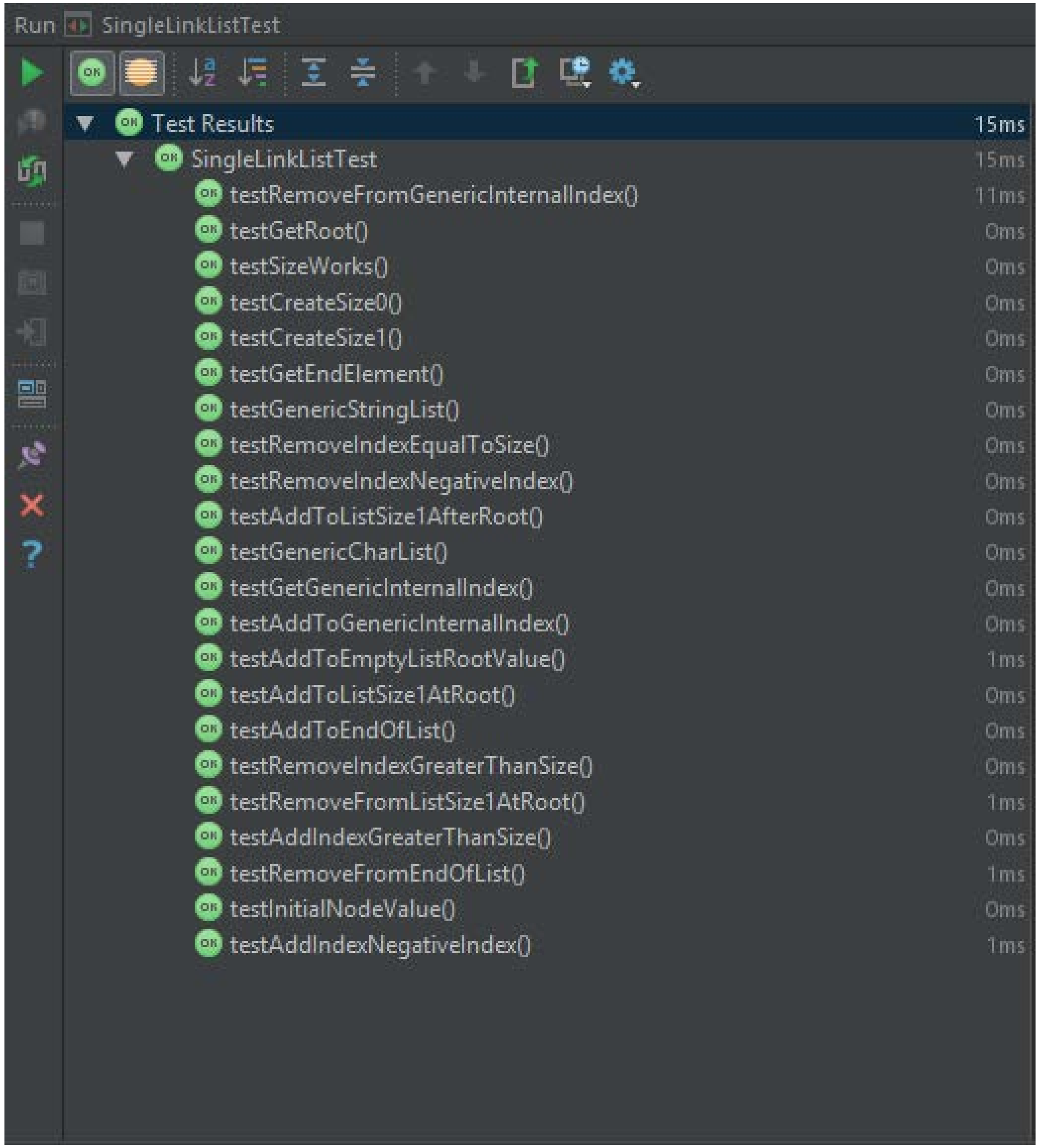
```java
@Test
void testRemoveFromEndOfList() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    intList.add(1, 2);
    intList.add(2, 3);
    //List = {1, 2, 3}
    if(intList.get(0) != 1 || intList.get(1) != 2 || intList.get(2) != 3)
    {
        fail("List not set up correctly for test.");
    }

    int removed = intList.remove(2);
    //List = {1, 2}

    if(removed != 3)
    {
        fail("Removed element value 3 not correctly returned.");
    }

    if(intList.get(1) != 2)
    {
        fail("2 no longer at correct index of 1");
    }

    if(intList.getSize() != 2)
    {
        fail("Size not correctly updated to 2.");
    }
}
```

```java
@Test
void testGetRoot() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    intList.add(1, 2);
    intList.add(2, 3);
    //List = {1, 2, 3}

    if(intList.get(0) != 1)
    {
        fail("Root element value 1 not correctly returned.");
    }
}

@Test
void testGetGenericInternalIndex() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    intList.add(1, 2);
    intList.add(2, 3);
    //List = {1, 2, 3}

    if(intList.get(1) != 2)
    {
        fail("Element value 2 not correctly returned");
    }
}

@Test
void testGetEndElement() throws ListAccessError
{
    SingleLinkList<Integer> intList = new SingleLinkList<>(1);
    intList.add(1, 2);
    intList.add(2, 3);
    //List = {1, 2, 3}

    if(intList.get(2) != 3)
    {
        fail("End element value 3 not correctly returned.");
    }
}
```

```java
@Test
void testGenericCharList() throws ListAccessError {
    SingleLinkList<Character> charList = new SingleLinkList<>('a');
    charList.add(1, 'b');
    charList.add(2, 'c');
    //List = {a, b, c}
    if(charList.getSize() != 3)
    {
        fail("Size of 3 not correctly returned.");
    }

    charList.add(0, 'z');
    //List = {z, a, b, c}
    if(charList.getSize() != 4)
    {
        fail("list size not correctly updated to 4");
    }
    if(charList.get(0) != 'z')
    {
        fail("Root element not correctly changed to z");
    }
    if(charList.get(1) != 'a')
    {
        fail("root element a not correctly moved to index 1");
    }
    char removed = charList.remove(2);
    //List = {z, a, c}
    if(charList.getSize() != 3)
    {
        fail("List size not correctly updated to 3");
    }
    if(charList.get(1) != 'a')
    {
        fail("Previous element no longer a");
    }
    if(charList.get(2) != 'c')
    {
        fail("Next element no longer c");
    }
    if(removed != 'b')
    {
        fail("Removed element value b not correctly returned.");
    }
}
```

```java
    @Test
    void testGenericStringList() throws ListAccessError {
        SingleLinkList<String> stringList = new SingleLinkList<>("aaa");
        stringList.add(1, "bbb");
        stringList.add(2, "ccc");
        //List = {aaa, bbb, ccc}
        if(stringList.getSize() != 3)
        {
            fail("Size of 3 not correctly returned.");
        }

        stringList.add(0, "zzz");
        //List = {zzz, aaa, bbb, ccc}
        if(stringList.getSize() != 4)
        {
            fail("list size not correctly updated to 4");
        }
        if(!(stringList.get(0).equals("zzz")))
        {
            fail("Root element not correctly changed to zzz");
        }
        if(!(stringList.get(1).equals("aaa")))
        {
            fail("root element aaa not correctly moved to index 1");
        }
        String removed = stringList.remove(2);
        //List = {zzz, aaa, ccc}
        if(stringList.getSize() != 3)
        {
            fail("List size not correctly updated to 3");
        }
        if(!(stringList.get(1).equals("aaa")))
        {
            fail("Previous element no longer aaa");
        }
        if(!(stringList.get(2).equals("ccc")))
        {
            fail("Next element no longer ccc");
        }
        if(!(removed.equals("bbb")))        {
            fail("Removed element value bbb not correctly returned.");
        }
    }
}
```

Result of testing.



Self Evaluation.

The marking scheme lists 5 marks for a solution having Boundary checking and exceptions with the inclusion of a test suite.

My implementation implements boundary checking with exceptions at the start of all methods and includes a comprehensive test suite. Evidence for both can be found in the above three headings. My implementation further more includes boundary checking as part of its optimisation, reducing the number of checks and re-allocations required. This improves the efficiency of the implementation and allows it to scale up much further than other implementations.

(Additional) Question 1.5: Model answer comparison.

Model answer for Get method.

Code Listing.

```java
package linkedList.list;

import linkedList.node.ListNode;
import linkedList.node.SingleLinkNode;

/**
 * A partial implementation of the List interface.
 * This implementation only implements the T get(int index) method, and the class must, therefore
 * be declared abstract.
 *
 * @param <T> the type of object stored in the list.
 */
public abstract class SingleLinkListModel<T> extends BasicList<SingleLinkNode<T>,T> implements List<T> {

    /**
     * A helper method to access a node at a specified index.
     *
     * @param index the index of the node to be accessed.
     * @throws ListAccessError if there is no node with the given index.
     */
    ListNode<T> getNode(int index) throws ListAccessError {
        // Is the list empty?  If so, cannot access the node.
        if (isEmpty()) {
            throw new ListAccessError("Cannot get node.  List is empty.");
        }
        // Is the given index negative?  If so, this is an error.
        if (index < 0) {
            throw new ListAccessError("Cannot get node.  Negative index.");
        }
        /*
         * Try to find the specified node by "walking" through the list, following links to successor
         * nodes.  The index tells us how many links need to be followed to reach the required node,
         * so reduce the index by one each time a link is followed.  When the index reaches zero, the
         * required node has been found.  If the end of the list is reached (next node is null), before
         * the index reaches zero, there were not enough nodes in the list (the index was too high).
         */
        ListNode<T> currentNode = getRoot(); // start at the root
        while (index != 0 && currentNode != null) { // walk along the list (if haven't reached the end by hitting null node)
            currentNode = currentNode.getNext(); // by gettting next node in the list
```

```java
            index--; // and reducing index by one
        }
        // Reached the end of the list (by hitting null node)?  If so, cannot access the required node.
        if (currentNode == null) {
            throw new ListAccessError("Cannot get node.  Not enough nodes in the list.");
        }
        // Successfully found node by walking through until index was zero.
        return currentNode;
    }

    /**
     * Access the value at a given index.
     *
     * @param index the index of the value to be accessed.
     * @throws ListAccessError if there is no value with the given index.
     */
    public T get(int index) throws ListAccessError {
        return getNode(index).getValue();
    }
}
```

My Implementation for Get method.

Code Listing.

```
/**
 * Return the value of the element in the list at the index specified.
 *
 * @param index the index of the entry to be accessed.
 * @return the value of the element at the index specified.
 * @throws ListAccessError if the index is invalid in respect to the size of the list.
 */
@Override
public T get(int index) throws ListAccessError
{
    //Make sure the index trying to be accessed is not invalid.
    if(index < 0 || index > size) { throw new ListAccessError("Invalid index location " + index); }

    //Set up a storage variable for the node being accessed.
    SingleLinkNode<T> get = getRoot();

    //Cycle through the list until the correct element has been found.
    for(int x = 1; x <= index; x++)
    {
        get = get.getNext();
    }

    //Return the element's value.
    return get.getValue();
}
```

Comparison.

One thing that is noticeable straight off the bat is the larger exception handling that the model answer includes at the top of the function. Despite the extra lines, the model answer handles initial exceptions in a slightly more user friendly way, giving the user/programmer information of the exact problem they've encountered, rather than my method which simply returns the invalid index.

The model answer also does exception handling during the 'walk-through' to find the node requested, whereas as mine does not. This is because my implementation for the SingleLinkList adds a recorded size of the list which is used during the initial error checking to make sure the programmer/user is not trying to access a data location past the end of the list. The model answer does not have this 'size' recording, therefore cannot do this check at the start of the method, bulking out the code and making it slightly less readable.

The perfect combination of error checking would be the inclusion of my size recording and initial error checking handling everything along with the model implementations user friendly exception reporting.

The exclusion of this error handling at the start has impacts upon the efficiency of the model solution, as when finding the correct node, additional checks have to be done each time a new element is attempted to be accessed. This check makes sure that the next element is not null before attempting to access it. My implementation however, can simply walk through without any possibility of reaching a null node. This is objectively better than the model implementation's approach as it improves code readability & maintainability and improves efficiency via reducing the amount of brute force checks that have to be made during the walk-through.

In conclusion, I'd state that the model answer outweighs my solution in terms of user experience, allowing easier bugfixing and smoother usage. This could arguably be a key factor given the abstract implementation nature of these exercises. However, my implementation takes the lead where efficiency is concerned. Despite this efficiency lead being very small given the computational power of systems today, my implantation would scale far better into a system the likes of which are developed and used today.

(Additional) Question 2: Write some test code that uses array generators to create large random arrays. Use the values in these arrays to populate instances of your implementation of linked lists. Now attempt multiple accesses of the data both in the arrays and in the lists.

Code Listing.

```java
package Comparison;

import arrayGenerator.generator.IntegerArrayGenerator;
import arrayGenerator.scope.IntegerScope;
import linkedList.list.ListAccessError;
import linkedList.list.SingleLinkList;

import java.util.Random;

/**
 * Created by u1661665(Joshua Pritchard) on 19/11/2018.
 * Version : 20/11/2018
 */
public class SingleLinkListComparison
{
    private static final int NUM_TESTS = 100;
    private static final int SIZE_OF_ARRAY = 100;

    private static void timeComparison() throws ListAccessError
    {
        //Create a new random instance to obtain random indices.
        Random rand = new Random();

        //Set up an iterator to gradually increase the size of the array.
        for(int arraySize = SIZE_OF_ARRAY; arraySize <= 1000000; arraySize *= 10 )
        {
            //Create a new integer array based on the specified size of array.
            Integer[] ints = new IntegerArrayGenerator(new IntegerScope()).getArray(arraySize);

            //Use this array to populate a SingleLinkList.
            SingleLinkList<Integer> list = new SingleLinkList<>();
            for(int x = 0; x < arraySize; x++)
            {
                list.add(x, ints[x]);
            }

            System.out.println("Testing array of size: " + arraySize);

            //Repeat this however many times specified by num_tests.
            for (int x = 0; x < NUM_TESTS; x++)
            {
                //Obtain a random index within the array/list.
```

```java
            int i = rand.nextInt(arraySize);

            //Time how long it takes to access the array and print this value.
            double before = System.nanoTime();
            int arrayInt = ints[i];
            System.out.println("System took " + ((System.nanoTime()) - before) + " ns. to access the array");

            //Time how long it takes to access the list and print this value.
            before = System.nanoTime();
            int listInt = list.get(i);
            System.out.println("System took " + ((System.nanoTime()) - before) + " ns. to access the list.");
        }

        System.out.println();
    }
}

public static void main(String[] args)
{
    try
    {
        SingleLinkListComparison.timeComparison();
    }
    catch (ListAccessError listAccessError)
    {
        listAccessError.printStackTrace();
    }
}
}
```

Test data.

This is an example of the test data I acquired from running the code listing above. There are 100 rows of recordings for each array size, however I have abbreviated the table size in the interests of brevity.

100		1000		10000		100000		1000000	
Array	List	Array	List	Array	List	Array	List	Array	List
4277	8839	570	25945	1140	2566	856	6843	286	1841195
570	570	285	6843	285	88952	571	19957	285	1094225
285	855	1711	21668	285	22238	285	14255	285	305346
570	3421	285	3706	570	46757	570	88953	285	129722
285	1710	285	15110	285	7698	285	106628	285	1698643
570	1425	285	2851	0	3706	285	95224	285	396578
285	1996	285	10834	285	52459	285	14826	285	551105
570	2851	285	12830	285	2566	285	5702	285	1272414
286	2281	285	5702	285	13970	570	21953	285	1196577
570	1711	285	4562	1995	2851	285	124590	285	1747682
285	2851	285	5132	285	56735	570	127156	285	1223378
285	2566	0	3136	285	32217	570	68425	285	841910
570	2280	285	1711	285	855	285	43621	286	1139841
571	1996	285	1426	285	855	285	79259	0	1965500
285	2851	0	855	285	1711	285	71561	0	1987453
570	3136	285	856	285	12545	1140	9123	285	338703
285	3137	285	1710	285	52174	571	127441	285	420812
285	1782464	285	1711	571	2280	285	47327	0	2238059
570	3422	0	1996	285	570	570	143406	571	217249
285	2281	0	1711	285	9694	285	79828	570	1093940
285	1996	285	855	286	59302	571	92088	285	1349108
286	855	285	1140	285	8268	570	88382	285	1334852
285	2851	570	3136	285	48183	285	83821	570	1137847
285	2281	285	570	285	5702	285	5417	285	1338844
285	2566	285	570	1711	53884	285	144548	285	181611
285	3136	285	2281	285	8553	570	97220	286	1647611
285	2851	285	1426	285	4276	285	44761	285	23664
285	2851	285	1996	285	1711	285	9408	285	733285
285	2566	285	855	285	15681	2565	53884	285	1242479
571	1141	285	1711	285	55310	570	119173	285	285673
286	855	285	855	285	6842	286	17962	286	87811
571	2566	286	570	285	5417	570	56450	285	1030933
285	855	285	1140	285	48183	570	4847	3422	1630790
285	1140	285	2566	285	53885	285	24519	285	512900
570	2851	570	855	285	9408	285	15110	285	1211688
285	3706	285	2280	285	45047	285	112616	285	902636
285	3422	285	1425	570	11404	285	855	285	628652
285	1426	285	2566	285	40485	285	133999	286	331860
285	2281	285	570	285	5987	285	179615	285	220955
570	1711	285	1996	285	5132	570	36778	0	1823234
570	1711	285	2851	285	47042	285	121168	285	1881395
285	3137	0	1996	285	11974	571	62438	0	2005985
285	856	285	1425	285	2566	285	16251	0	2034780
571	3421	285	570	285	40485	285	102923	285	57876
571	3421	285	571	285	12829	285	70136	285	108624
285	1711	286	1996	285	21097	285	58161	285	80113
571	2566	285	1425	285	7697	285	52744	286	455024
285	1140	285	5702	1140	2566	285	6557	285	962508
285	1140	571	6843	285	11974	570	31931	285	1755665
285	3707	570	4276	571	9979	571	28796	285	2425087
285	2280	571	3706	571	14255	285	106628	0	396008
570	1996	570	3136	285	3706	285	856	285	755523
570	855	570	3422	285	5987	285	114611	0	944831
570	1996	855	2281	570	2281	571	98646	285	42765

Sheet1

Averages.

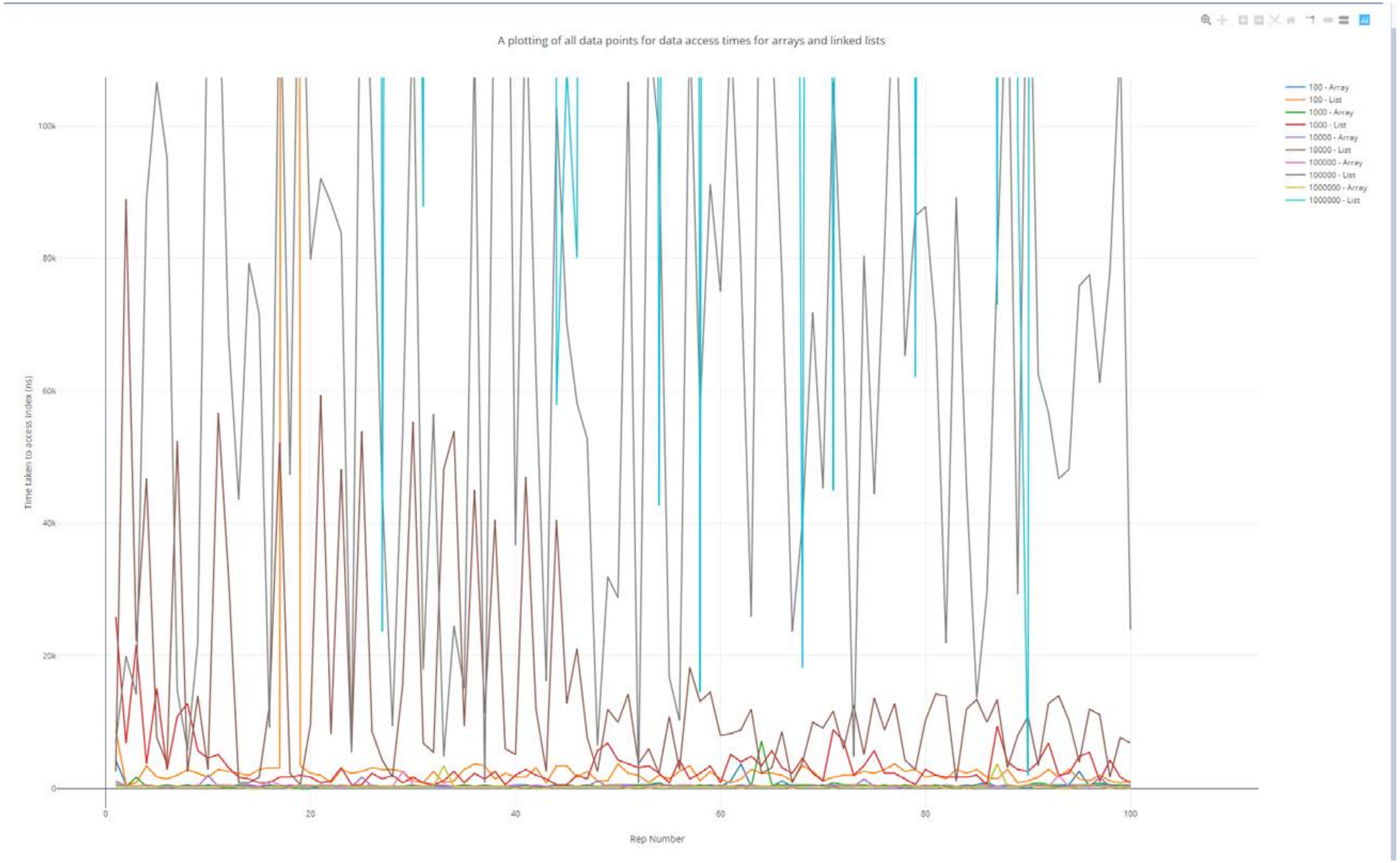
From this data I calculated the average access time for each data structure for each size of array tested.

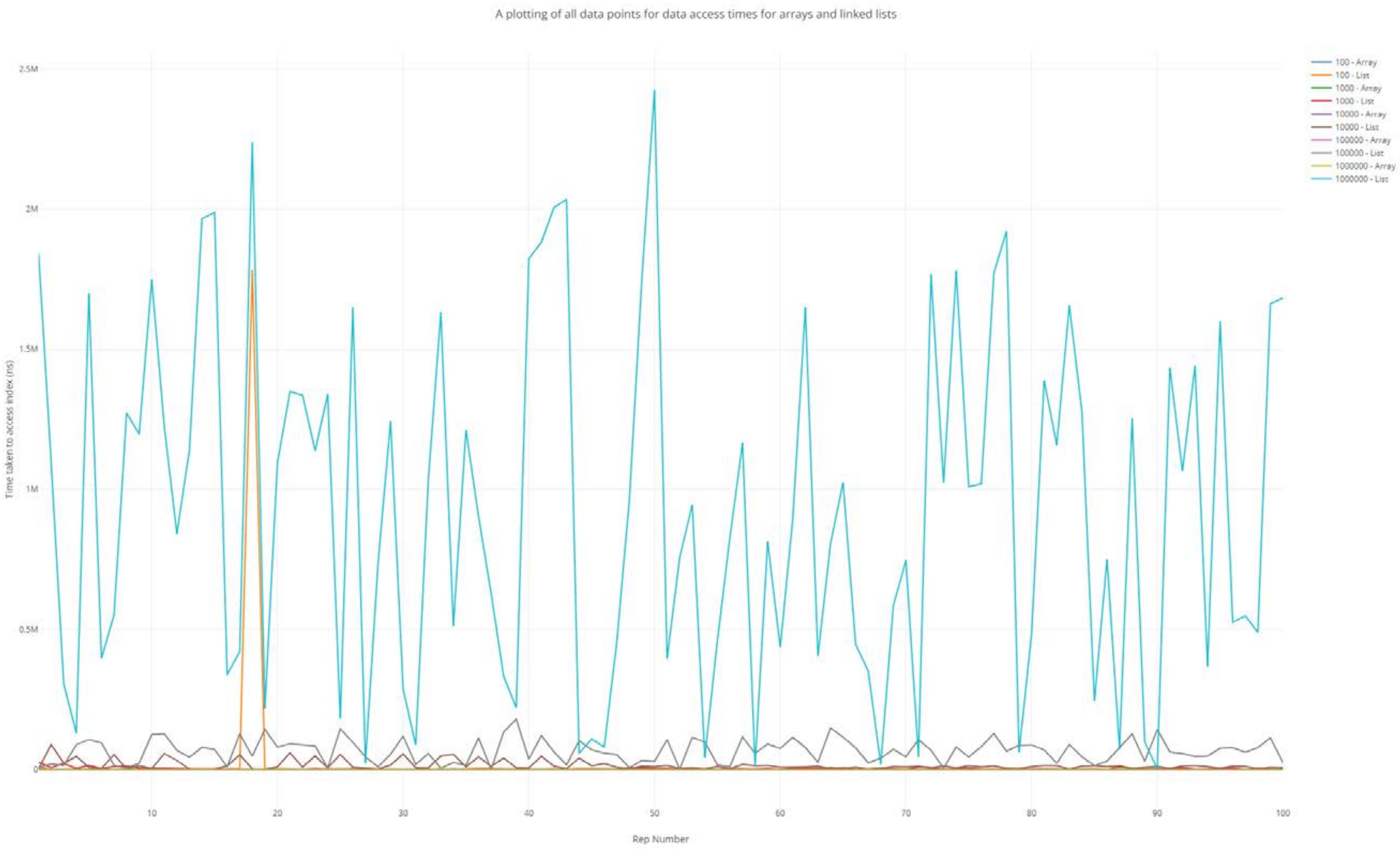
Averages	Array	List
100	484.7	19994.29
1000	467.55	3478.25
10000	379.14	15070.53
100000	484.7	66437.62
1000000	484.7	935018.2

Graph plotting.

All data points

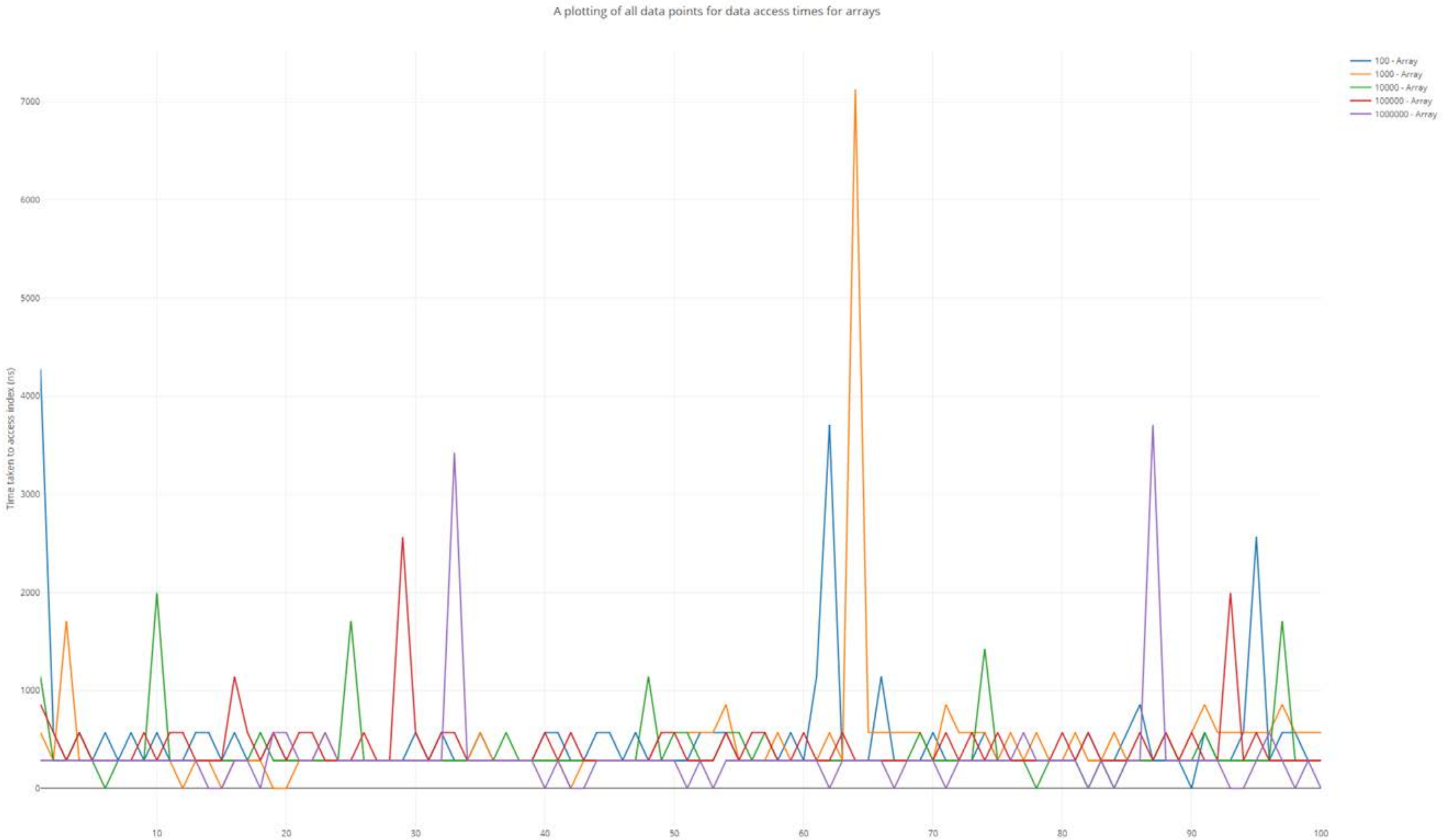
This graph shows every data point plotted. From this scale it is easy to see the constant access time of arrays from the traces near the bottom of the graph.





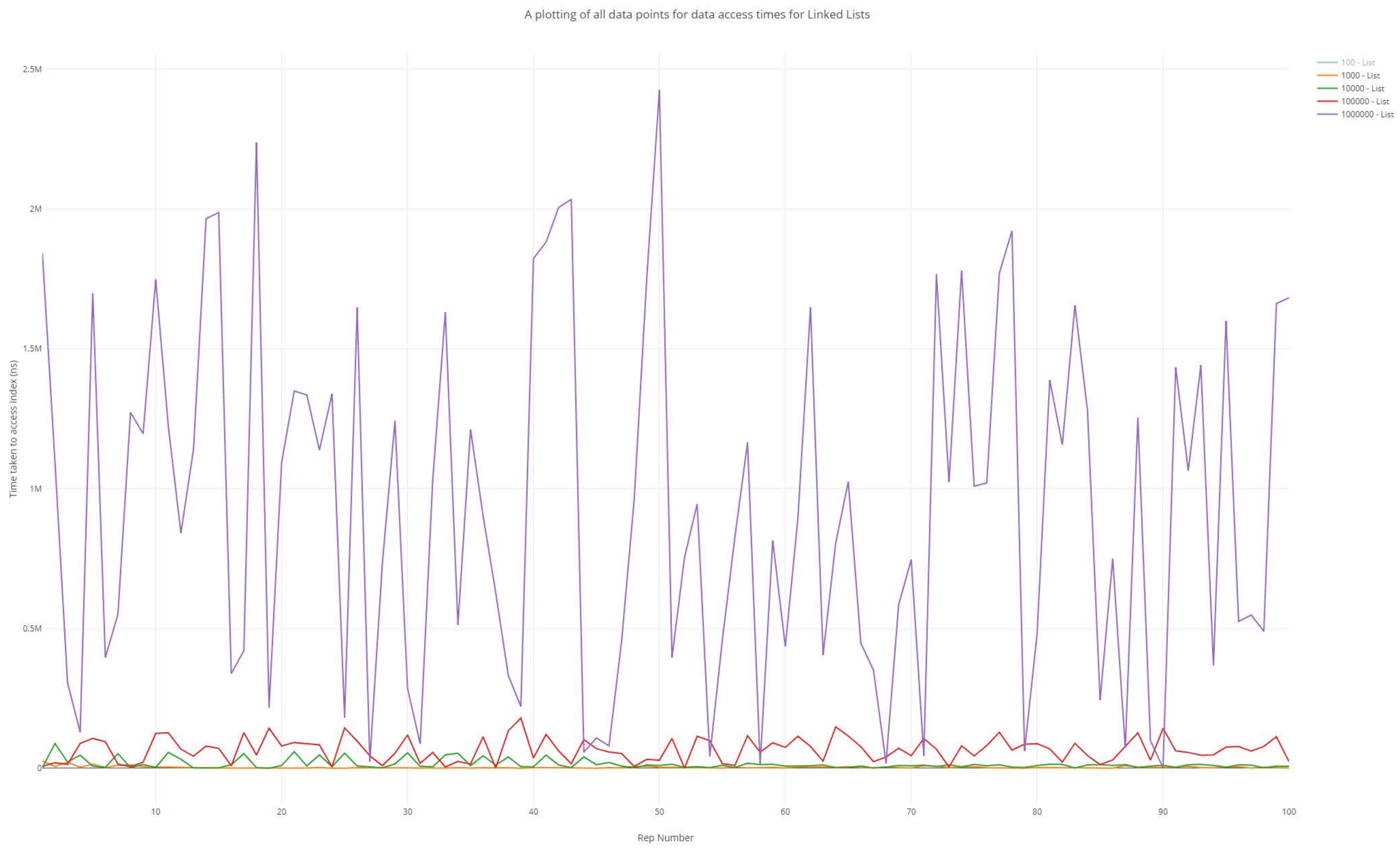
When the graph is scaled like this however, it becomes much easier to grasp just how badly the Linked List performs at higher element counts. The blue line on the graph plots the times for a Linked List of 1,000,000 elements and absolutely dwarfs everything beneath it. In comparison, the array trace for the same number of elements retains its constant access time.

Array time comparisons.



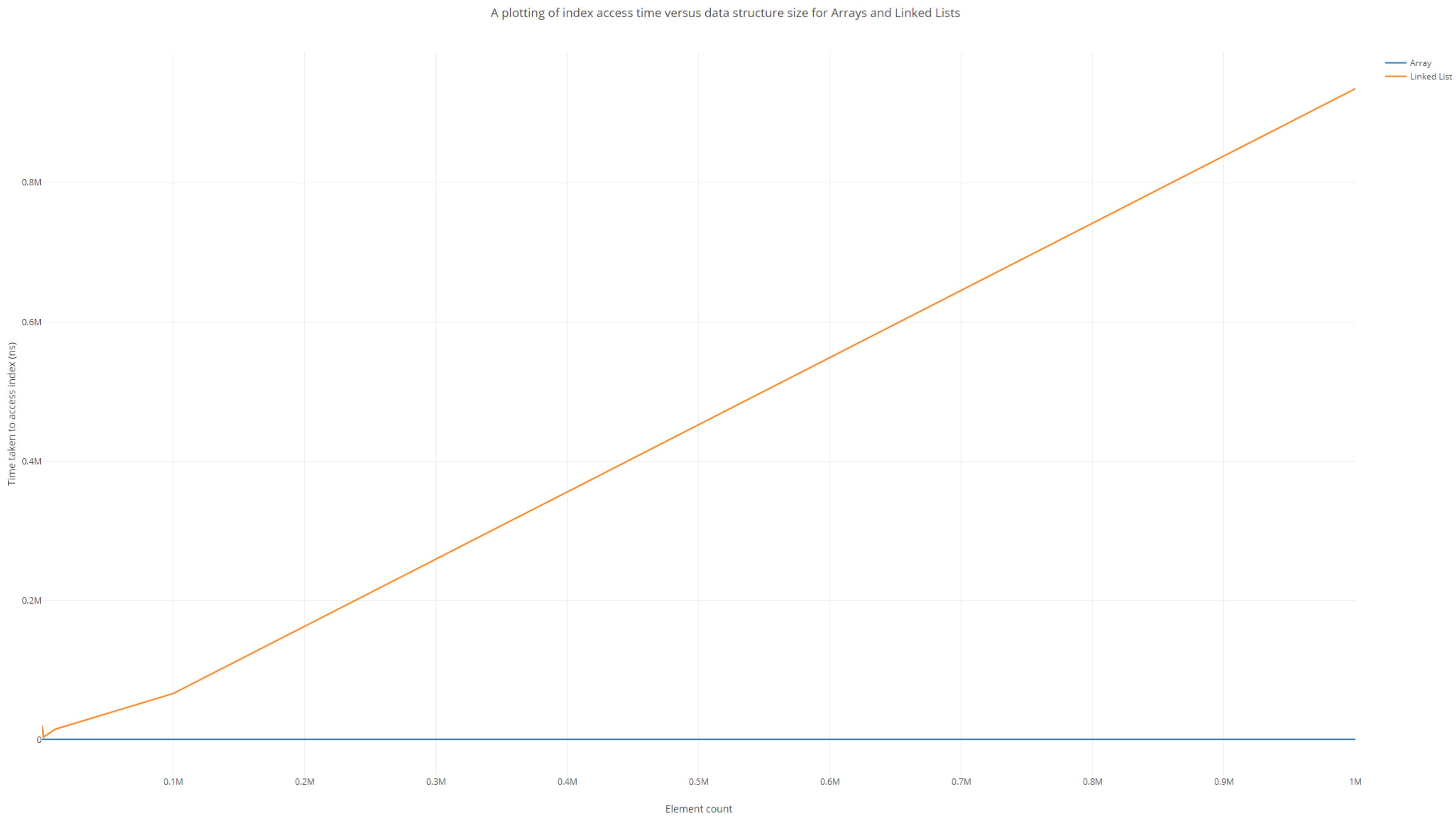
This graph shows the times taken by the array to access the same data as the Linked List. As can be seen, the highest time recorded to access an index was taken by an array of size 1000. This proves that arrays are not affected by size in the same way that Linked Lists are when it comes to access time. Interestingly, the graph's visibly discrete Y values display the constant access time quite well, as there are no analogue values.

Linked List time comparisons.



This graph shows the times taken by Linked Lists to access the same data as the arrays. Each trace visibly increases its average up the Y axis, proving the Linked List dependence on element count. This graph, in stark contrast to the graph plotted for the array, shows the continuous/analogue performance of the Linked List. This also suggests a dependence on computer performance that is much higher than the arrays. This has an upper limit and Linked Lists will always be slower than arrays however, as Linked Lists still have to do operations before accessing a memory address whereas arrays can almost directly access a memory address. This means that however fast a CPU gets at processing these instructions, a Linked List will **always** be slower than an array for the same tasks.

Average Plots.

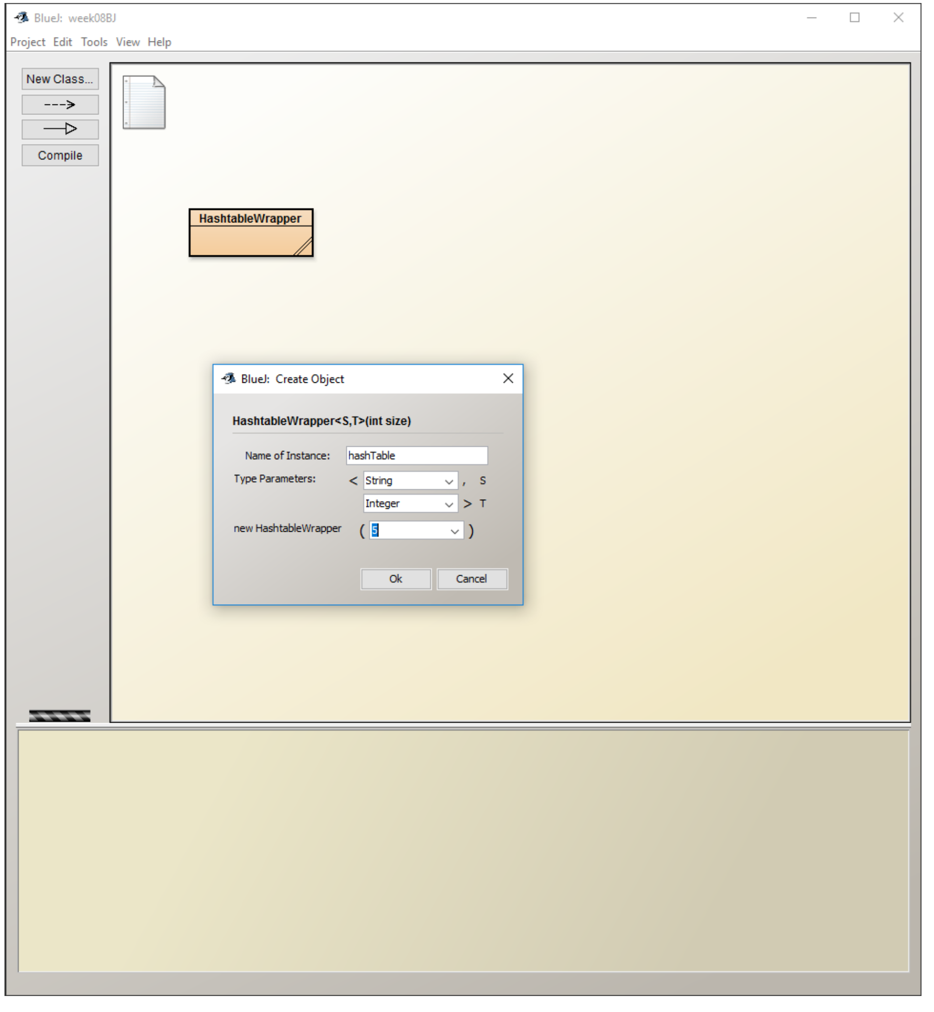


Finally, this graph (plotting the average access time per element count) shows the constant access time for Arrays very well. The graph may also suggest a linear relationship between data structure size and access time, however I am reluctant to outright state this as there are only 5 data point plotted on this graph, and the majority of the graph is taken up by the difference between 100,000 and 1,000,000 points.

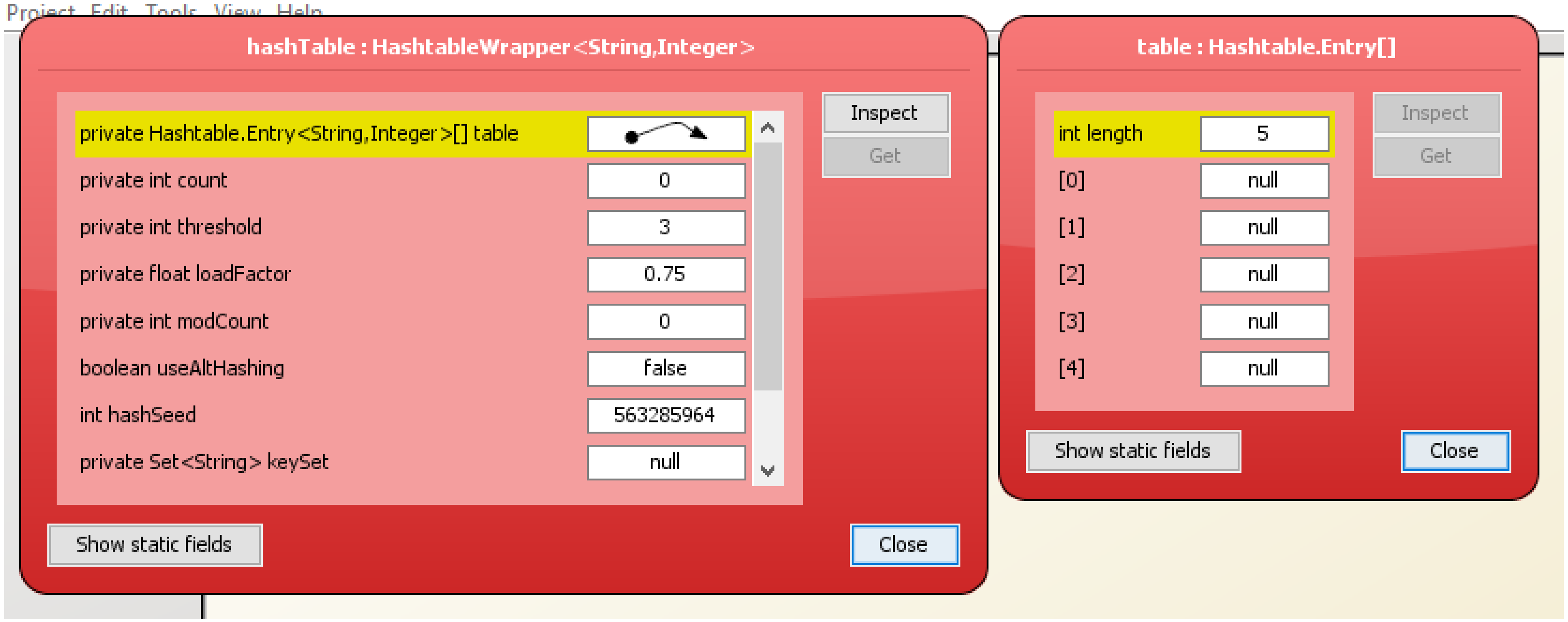
Practical 5 (Week 8)

(Logbook) Question1: Create an object instance of the HashtableWrapper(String, Integer) class.

Ensure this hash table uses Strings as keys, Integers as values, and has an initial size of 5.

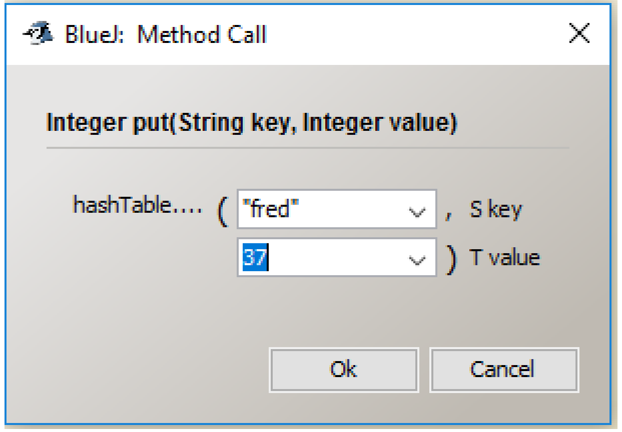


Inspect the object you have just created, paying particular attention to the object's internal array.



Some fields of note in this inspection might be the count of 0, (Stating the number of elements in the array) and each field in the internal array, showing that each entry is null and has nothing contained within.

Now, using the void put(String key, Integer data) method, add the key/value pair (“fred”, 37) to the hashtable.

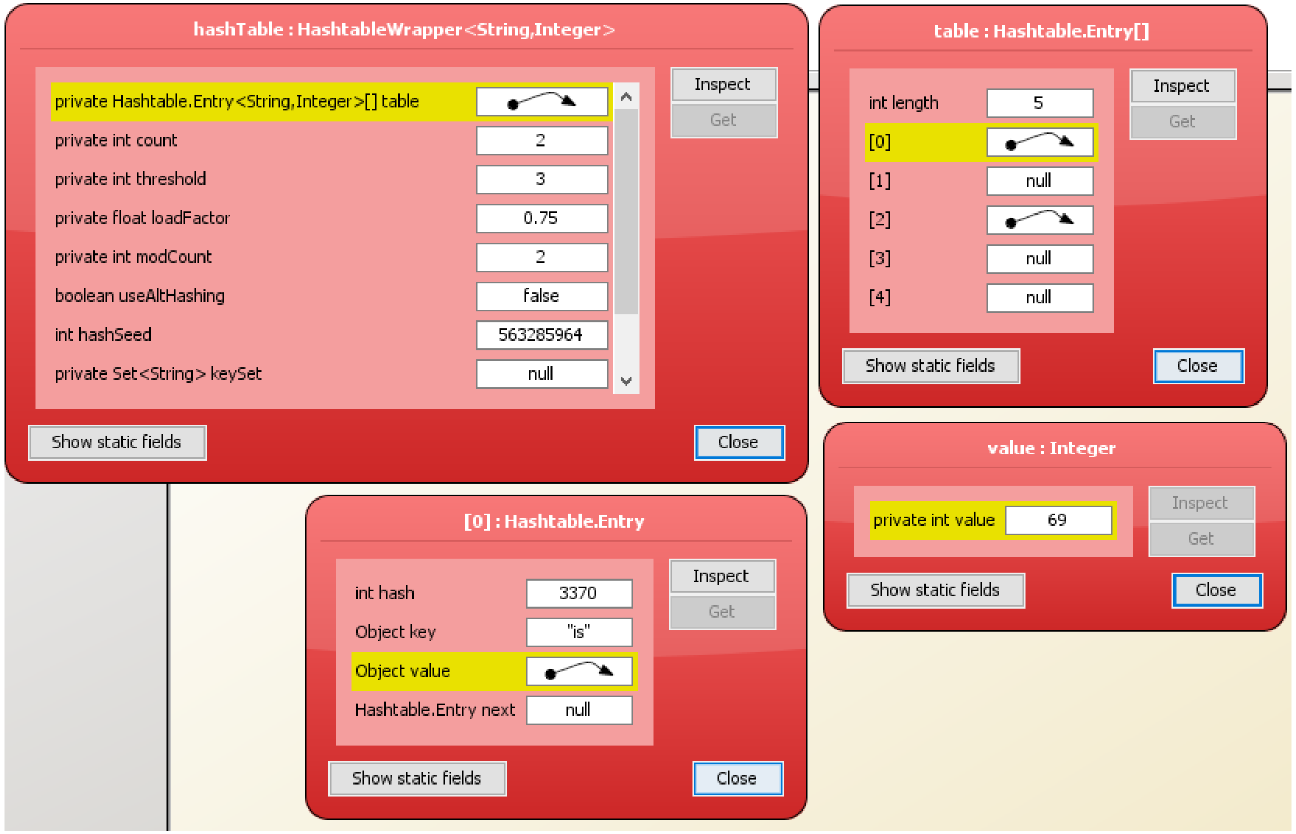


Inspect the object again.

Fields of note here: The count field has changed to 1 to signify there is one element within the internal array. The internal array at index 2 contains a pointer to a 'Hashtable Entry' which contains the hash, key, next entry (for collisions) and a pointer to the actual value held. The modcount field has increased to 1.

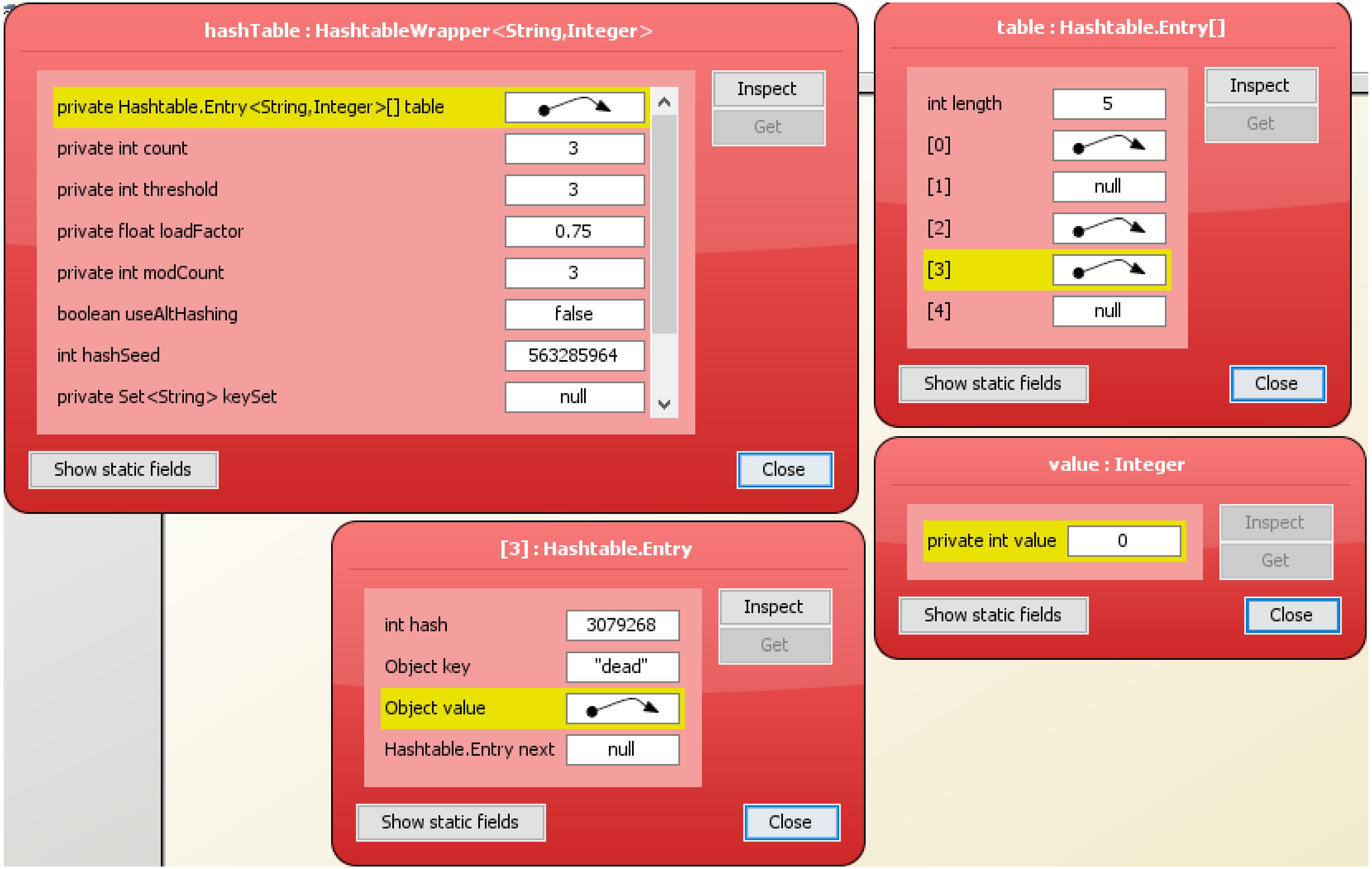
Now add the following key/value pairs, again inspecting the hashtable object after each new pair is entered.

("is", 69)



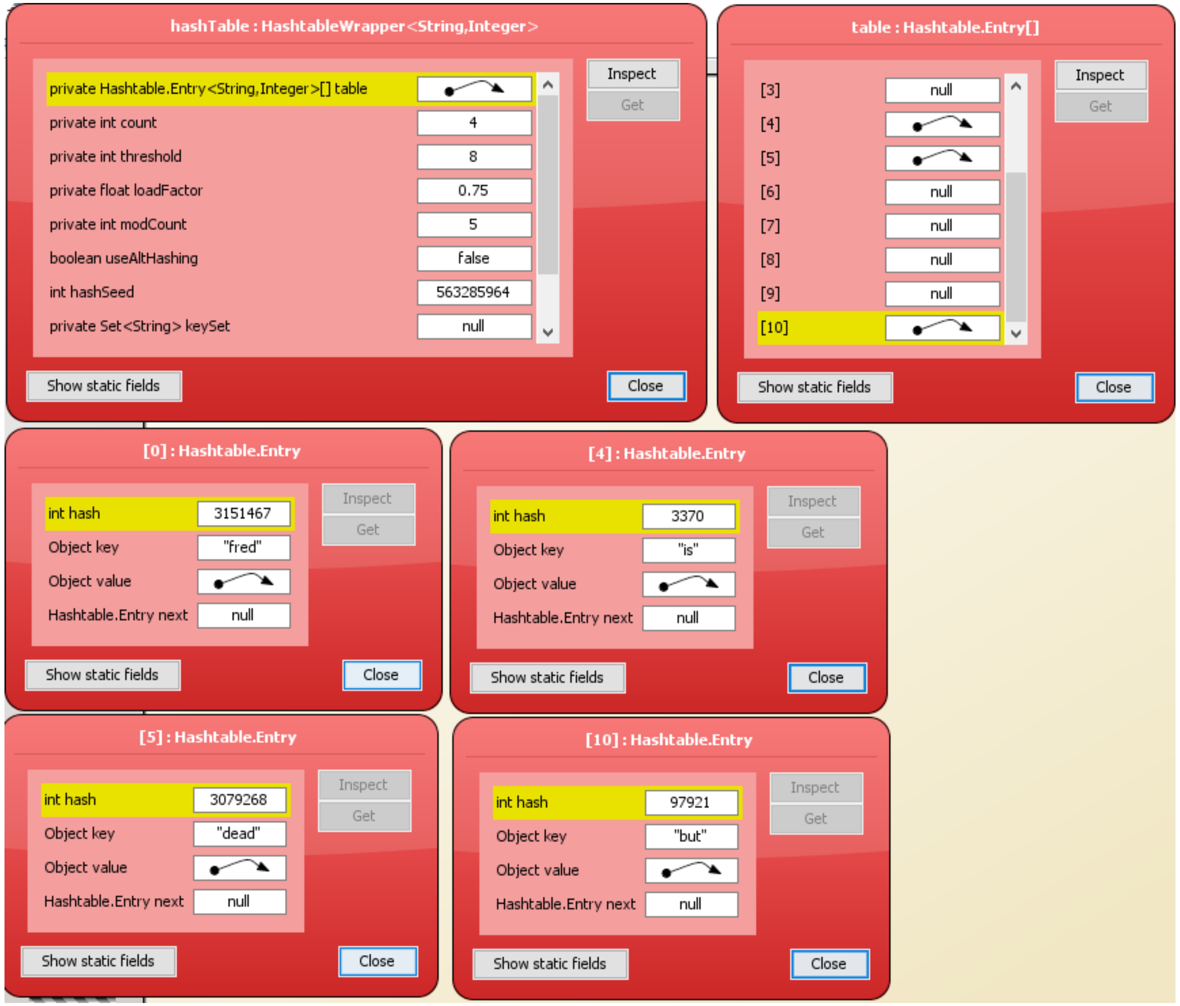
The count field has changed to 2. The modcount field has increased to 2. Index 0 in the internal array contains a hashtable entry for ("is", 69).

("dead", 0)



The count field has increased to 3, the modcount field has increased to 3. Index 3 contains a pointer to a hashtable entry for ("dead", 0).

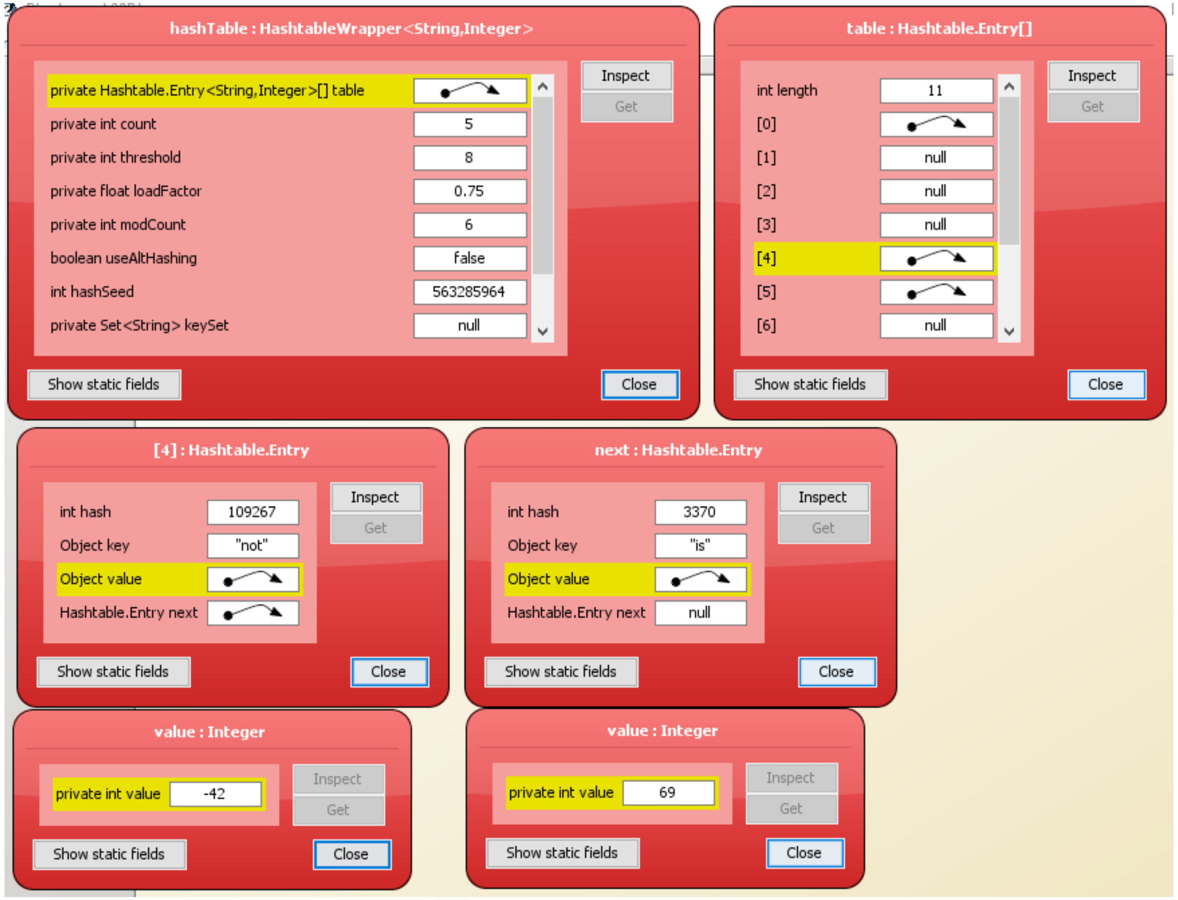
("but", 999)



Value	Old Index	New Index
("fred", 37)	2	0
("is", 69)	0	4
("dead", 0)	3	5

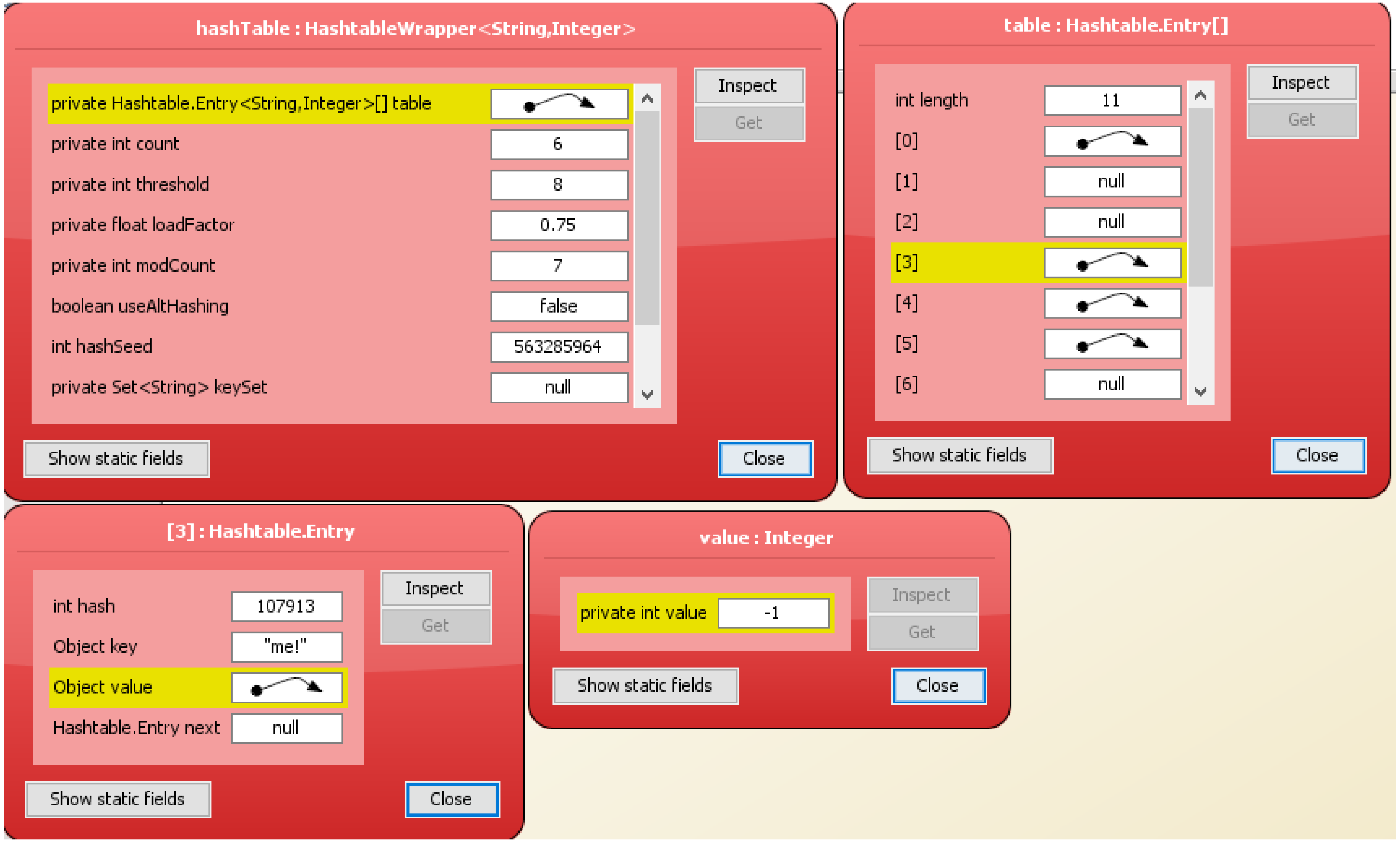
The count field has increased to 4. The threshold field has increased to 8. This is perhaps due to count reaching the value of threshold, upon which the hashtable realises it needs to increase the size of its internal array. The value of modCount has increased to 5. The hashtable entries have also been moved around, however I'm unsure exactly as to how the Hashtable class has determined this.

("not", -42)



Count has increased to 5. Modcount has increased to 6. The new hashtable entry has fallen on the index of ("is", 69), so the new hashtable entry has taken its place, and kicked ("is", 69) down a place so that ("not", -42) is the first visible entry in the internal array and ("is", 69) resides at the second layer.

("me!", -1)



Count has increased to 6. Modcount has increased to 7. Otherwise, ("me!", -1) has been treated very much like a standard hashtable entry, taking the until now empty index of 3.

Additional explanation.

For each key-value pair, the key is passed through the hashing function of the hashtable, to gain an integer hash value. This is used as an index (Wrapped around using a modulus) to find the position of the hashtable entry.

The java Hashtable class is an implementation of an *open* hashtable, which means that when collisions occur, the hashtable uses *buckets* to store multiple entries (searched seququentially) under one index value. An example of this behaviour is when (“not”, -42) is added.

The loadFactor variable is a measure of how full the hash table is allowed to get before its capacity is automatically increased. 0.75 (used here) is the default and offers ‘a good tradeoff between time and space costs’. It’s obvious to see that a lower load factor means that the array will be expanded more often, therefore there is less chance of a collision. However, the array is larger and takes up more space in memory. On the flipside, having a larger loadFactor means that the array will be expanded less often, resulting in less space in memory used, but larger chance for collisions to occur. This results in more buckets and sequential searches, diminishing the advantages the hashtable provides.

When the size of the array is dynamically increased, the entire array is rehashed. This explains the behaviour of the hashtable entries moving around when (“but”, 999) is added. Rehashing is a relatively time consuming operation however. If a lot of entries are going to be made into the hashtable, then it can sometimes be better to create the hashtable with a larger initial capacity so that rehashing is less likely to occur.

The Java Hashtable documentation states that **no** rehashing will ever occur **if** the **initial capacity** is **greater than** the (**maximum entries to contain** / **load factor**)

The only danger of this is setting the initial capacity too high, which can waste space if a lot of duplicate entries are added.

Self Evaluation

I believe I provide more than just a sequence of screenshots, providing a step by step description then an extended analysis at the end of the exercise. I believe my analysis explains how internal array slots are allocated and provides a look at the differences in choice where loadFactor and initialCapacity are concerned. I believe my analysis is full and takes everything into consideration. Therefore, I would self-evaluate this week's exercises as a 5/5.

Practical 6 (Week 9)

(Logbook) Question 1: Complete the implementation of the binary tree class.

Code Listing

```java
package binaryTree;

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

public class BinaryTree<T extends Comparable<? super T>> implements BTree<T> {

    private TreeNode<T> root; // the root node

    /**
     * Construct an empty tree.
     */
    public BinaryTree() {
        root = null;
    }

    /**
     * Construct a singleton tree.
     * A singleton tree contains a value in the root, but the left and right subtrees are
     * empty.
     * @param value the value to be stored in the tree.
     */
    public BinaryTree(T value) {
        root = new TreeNode(value);
    }

    /**
     * Construct a tree with a root value, and left and right subtrees.
     * @param value the value to be stored in the root of the tree.
     * @param left the tree's left subtree.
     * @param right the tree's right subtree.
     */
    public BinaryTree(T value,BinaryTree<T> left,BinaryTree<T> right) {
        root = new TreeNode(value,left,right);
    }
```

```java
/**
 * Check if the tree is empty.
 * @return true iff the tree is empty.
 */
@Override
public boolean isEmpty() {
    return root == null;
}

/**
 * Insert a new value in the binary tree at a position determined by the current contents
 * of the tree, and by the ordering on the type T.
 * @param value the value to be inserted into the tree.
 */
@Override
public void insert(T value) {
    if(isEmpty())
    {
        root = new TreeNode<T>(value, new BinaryTree<>(), new BinaryTree<>());
        return;
    }

    if(value.compareTo(this.getValue()) < 0)
    {
        root.getLeft().insert(value);
    }
    else
    {
        root.getRight().insert(value);
    }
    // implement insert(T value) here
}
```

```java
    /**
     * Get the value stored at the root of the tree.
     * @return the value stored at the root of the tree.
     */
    @Override
    public T getValue() throws NullPointerException {
        // Note: it might make sense to define getValue() to throw a (custom) exception if an attempt
        // is made to access a value from an empty tree.
        // However, since a tree is empty iff it's root node is null, it is also acceptable to rely
        // on Java's NullPointerException.
        // This comment also applies to the other get and set methods defined in this interface.

        // placeholder return value below - replace with implementation of getValue()

        if(isEmpty())
        {
            throw new NullPointerException("Tree at current node is empty.");
        }
        else
        {
            return root.value;
        }
    }

    /**
     * Change the value stored at the root of the tree.
     * @param value the new value to be stored at the root of the tree.
     */
    @Override
    public void setValue(T value) {
        // implement setValue(T value) here
        if(isEmpty())
            root = new TreeNode<T>(value, new BinaryTree<>(), new BinaryTree<>());
        else
            root.value = value;
    }
```

```java
    /**
     * Get the left subtree of this tree.
     * @return  This tree's left subtree.
     */
    @Override
    public BTree<T> getLeft() throws NullPointerException {
        // placeholder return value below - replace with implementation of getLeft()
        if(isEmpty())
        {
            throw new NullPointerException("Current node is empty.");
        }
        return root.left;
    }

    /**
     * Change the left subtree of this tree.
     * @param tree the new left subtree.
     */
    @Override
    public void setLeft(BTree<T> tree) {
        // implement setLeft(BTree<T> tree) here
        root.left = tree;
    }

    /**
     * Get the right subtree of this tree.
     * @return this tree's right subtree.
     */
    @Override
    public BTree<T> getRight() throws NullPointerException {
        // placeholder return value below - replace with implementation of getRight()
        if(isEmpty())
        {
            throw new NullPointerException("Current node is empty.");
        }
        return root.right;
    }
```

```java
    /**
     * Change the right subtree of this tree.
     * @param tree the new right subtree.
     */
    @Override
    public void setRight(BTree<T> tree) {
        // implement setRight(BTree<T> tree) here
        root.right = tree;
    }

    /**
     * Check if the tree contains a given value.
     * @param value the value to be checked.
     * @return true iff the value is in the tree.
     */
    @Override
    public boolean contains(T value) {
        // placeholder return value below - replace with implementation of contains(T value)
        //Terminate this branch of the recursion if the current node is empty.
        if(!isEmpty())
        {
            //If the value is found, return it.
            if (value.equals(getValue()))
            {
                return true;
            }
            //Else search the left subtree if less than the current node.
            else if (value.compareTo(root.getValue()) < 0)
            {
                return root.getLeft().contains(value);
            }
            //Or search the right subtree if greater than the current node.
            else if (value.compareTo(root.getValue()) > 0)
            {
                return root.getRight().contains(value);
            }
        }
        return false;
    }
```

```java
    /**
     * Traverse the tree, producing a list of all the values contained in the tree.
     * This is an implementation of an inOrder traversal.
     * @return a list of all the values in the tree.
     */
    @Override
    public List<T> traverse() {
        // placeholder return value below - replace with implementation of traverse()

        //Create a new arrayList to store the values.
        ArrayList<T> list = new ArrayList<>();

        //Call inorder traverse with this tree and the created list.
        inOrderTraverse(this, list);

        //Return the populated list.
        return list;
    }

    private void inOrderTraverse(BTree<T> tree, List<T> list)
    {
        //Terminate this branch of the recursion if the subtree is empty.
        if(tree.isEmpty())
        {
            return;
        }

        //recursively search the left subtree
        inOrderTraverse(tree.getLeft(), list);

        //add the current value.
        list.add(tree.getValue());

        //And recursively search the right subtree.
        inOrderTraverse(tree.getRight(), list);
    }
}
```

Testing class.

```java
package binaryTreeTest;

import ...;


import static org.junit.jupiter.api.Assertions.*;

/**
 * Created by u1661665(Joshua Pritchard) on 30/11/2018.
 * Version: 30/11/2018
 */
public class BinaryTreeTest
{
    /**
     * Test that a null tree is created correctly by using the isEmpty() and getValue() methods.
     */
    @Test
    void testNullTreeConstructor()
    {
        BinaryTree<Integer> intTree = new BinaryTree<>();

        if(!intTree.isEmpty())
            fail("Binary tree has not been created empty.");
        try
        {
            int x = intTree.getValue();
            fail("Exception for root being null not caught.");
        }
        catch(NullPointerException e){}
        try
        {
            BTree<Integer> leftTree = intTree.getLeft();
            fail("Exception for trying to access left tree from null node not caught.");
        }
        catch(NullPointerException e){}
        try
        {
            BTree<Integer> rightTree = intTree.getRight();
            fail("Exception for trying to access right tree from null node not caught.");
        }
        catch(NullPointerException e){}
    }
```

```java
/**
 * Test that a single node tree is created correctly by using the isEmpty() and getValue() methods.
 */
@Test
void testRootTreeConstructor()
{
    BinaryTree<Integer> intTree = new BinaryTree<>(1);

    if(intTree.isEmpty())
        fail("Binary tree created empty.");

    try
    {
        BTree<Integer> leftTree = intTree.getLeft();
    }
    catch (NullPointerException e)
    {
        fail("Exception thrown trying to access available left null subtree.");
    }

    try
    {
        BTree<Integer> rightTree = intTree.getRight();
    }
    catch(NullPointerException e)
    {
        fail("Exception thrown trying to access available right null subtree.");
    }

    try
    {
        assertTrue(1 == intTree.getValue(), "Root value not 1 as expected.");
    }
    catch (NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown against node holding a value.");
    }
}
```

```java
/**
 * Test that a tree created with a left subtree and a null right subtree is created correctly using the
 * isEmpty(), getValue(), getLeft() and getRight() methods.
 */
@Test
void testRootAndLeftTreeConstructor()
{
    BinaryTree<Integer> intTree = new BinaryTree<>(2, new BinaryTree<>(1), new BinaryTree<>());
    /*
     *         2
     *        / \
     *       1   null
     */

    if(intTree.isEmpty())
        fail("Binary tree created empty.");

    try
    {
        int x = intTree.getValue();
        if(x != 2)
            fail("Root value not 2 as expected.");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown against node holding a value");
    }

    try
    {
        BTree<Integer> lTree = intTree.getLeft();
        int x = lTree.getValue();
        if(x != 1)
            fail("left subtree value not 1 as expected.");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown trying to access available subtree/its value.");
    }

    try
    {
```

```java
            BTree<Integer> rTree = intTree.getRight();
            try
            {
                int x = rTree.getValue();
                fail("Exception for right root being null not caught.");
            }
            catch(NullPointerException e) {}
        }
        catch(NullPointerException e)
        {
            e.printStackTrace();
            fail("Exception thrown trying to access available right null subtree.");
        }
    }
```

```
/**
 * Test that a tree created with a right subtree and a null left subtree is created correctly using the
 * isEmpty(), getValue(), getLeft() and getRight() methods.
 */
@Test
void testRootAndRightTreeConstructor()
{
    BinaryTree<Integer> intTree = new BinaryTree<>(2, new BinaryTree<>(), new BinaryTree<>(3));
    /*
     *        2
     *       / \
     *   null   3
     */

    if(intTree.isEmpty())
        fail("Binary tree created empty.");

    try
    {
        int x = intTree.getValue();
        if(x != 2)
            fail("Root value not 2 as expected.");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown against node holding a value");
    }

    try
    {
        BTree<Integer> rTree = intTree.getRight();
        int x = rTree.getValue();
        if(x != 3)
            fail("Right subtree value not 3 as expected.");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown trying to access available subtree/its value.");
    }

    try
    {
```

```java
            BTree<Integer> lTree = intTree.getLeft();
            try
            {
                int x = lTree.getValue();
                fail("Exception for left root being null not caught.");
            }
            catch(NullPointerException e) {}
        }
        catch(NullPointerException e)
        {
            e.printStackTrace();
            fail("Exception thrown trying to access available left null subtree.");
        }
    }
```

```java
    /**
     * Test that a tree created with a left subtree and a right subtree is created correctly using the
     * isEmpty(), getValue(), getLeft() and getRight() methods.
     */
    @Test
    void testRootAndBothTreeConstructor()
    {
        BinaryTree<Integer> intTree = new BinaryTree<>(2, new BinaryTree<>(1), new BinaryTree<>(3));
        /*
         *      2
         *     / \
         *    1   3
         */

        if(intTree.isEmpty())
            fail("Binary tree created empty.");

        try
        {
            int x = intTree.getValue();
            if(x != 2)
                fail("Root value not 2 as expected.");
        }
        catch(NullPointerException e)
        {
            e.printStackTrace();
            fail("Exception thrown against node holding a value");
        }

        try
        {
            BTree<Integer> rTree = intTree.getRight();
            int x = rTree.getValue();
            if(x != 3)
                fail("Right subtree value not 3 as expected.");
        }
        catch(NullPointerException e)
        {
            e.printStackTrace();
            fail("Exception thrown trying to access available right subtree/its value.");
        }

        try
        {
```

```
            BTree<Integer> lTree = intTree.getLeft();
            int x = lTree.getValue();
            if(x != 1)
                fail("Left subtree value not 1 as expected.");
        }
        catch(NullPointerException e)
        {
            e.printStackTrace();
            fail("Exception thrown trying to access available left subtree/its value.");
        }
    }
```

```java
    @Test
    void testInsertFullLeft()
    {
        BinaryTree<Integer> intTree = new BinaryTree<>(5, new BinaryTree<>(4), new BinaryTree<>(6));
        /*
         *      5
         *     / \
         *    4   6
         */

        intTree.insert(3);
        /*
         *      5
         *     / \
         *    4   6
         *   /
         *  3
         */

        try
        {
            if (intTree.getLeft().getLeft().getValue() != 3)
                fail("Left Left subtree of intTree not correctly set as 3.");
        }
        catch(NullPointerException e)
        {
            e.printStackTrace();
            fail("Exception thrown for inserted accessible value.");
        }
    }
```

```java
@Test
void testInsertLeftThenRight()
{
    BinaryTree<Integer> intTree = new BinaryTree<>(10, new BinaryTree<>(5), new BinaryTree<>(15));
    /*
     *      10
     *     /  \
     *    5    15
     */

    intTree.insert(6);
    /*
     *      10
     *     /  \
     *    5    15
     *     \
     *      6
     */

    try
    {
        if (intTree.getLeft().getRight().getValue() != 6)
            fail("Left Right subtree of intTree not correctly set as 6.");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown for inserted accessible value.");
    }
}
```

```java
@Test
void testInsertRightThenLeft()
{
    BinaryTree<Integer> intTree = new BinaryTree<>(10, new BinaryTree<>(5), new BinaryTree<>(15));
    /*
     *      10
     *     /  \
     *    5    15
     */

    intTree.insert(14);
    /*
     *      10
     *     /  \
     *    5    15
     *        /
     *       14
     */
    try
    {
        if (intTree.getRight().getLeft().getValue() != 14)
            fail("Right Left subtree of intTree not correctly set as 14.");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown for inserted accessible value.");
    }
}
```

```java
@Test
void testInsertFullRight()
{
    BinaryTree<Integer> intTree = new BinaryTree<>(5, new BinaryTree<>(4), new BinaryTree<>(6));
    /*
     *      5
     *     / \
     *    4   6
     */

    intTree.insert(7);
    /*
     *      5
     *     / \
     *    4   6
     *         \
     *          7
     */
    try
    {
        if (intTree.getRight().getRight().getValue() != 7)
            fail("Right Right subtree of intTree not correctly set as 7.");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown for inserted accessible value.");
    }
}
```

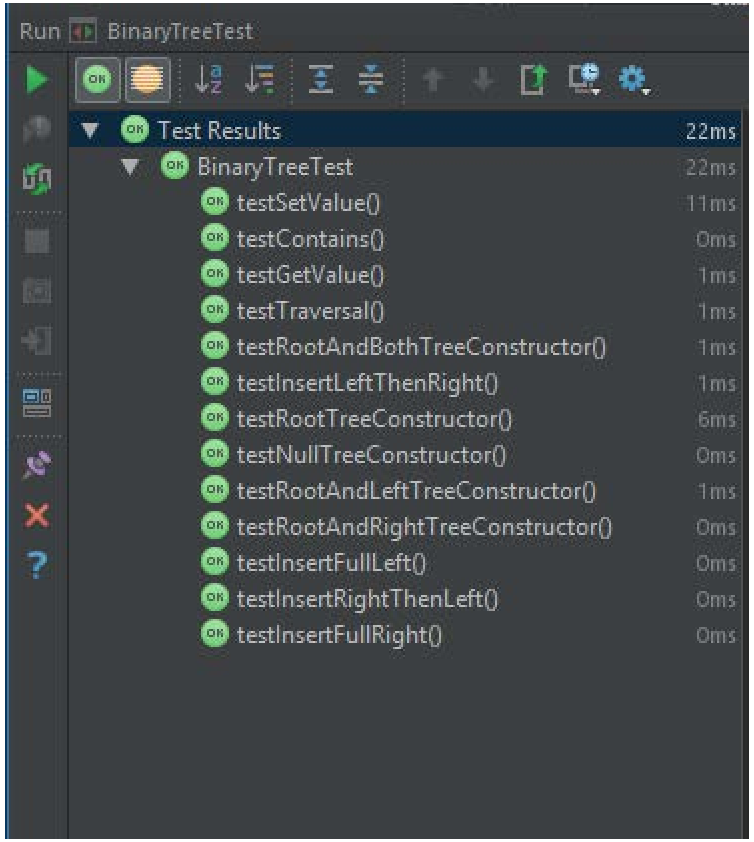
```java
@Test
void testGetValue()
{
    BinaryTree<Integer> intTree = new BinaryTree<>(2, new BinaryTree<>(1), new BinaryTree<>());
    /*
     *      2
     *     / \
     *    1   null
     */
    try
    {
        if (intTree.getValue() != 2)
            fail("getValue() not correctly returned 2.");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown for accessible value.");
    }

    try
    {
        if (intTree.getLeft().getValue() != 1)
            fail("getValue() not correctly returned 1.");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown for accessible value.");
    }
}
```

```java
@Test
void testSetValue()
{
    BinaryTree<Integer> intTree = new BinaryTree<>();
    /*
     *    null
     */

    intTree.setValue(2);
    /*
     *      2
     *     / \
     * null   null
     */

    try
    {
        if (intTree.getValue() != 2)
            fail("2 not correctly set as intTree value");
    }
    catch(NullPointerException e)
    {
        e.printStackTrace();
        fail("Exception thrown for accessible value.");
    }
}
```

```java
@Test
void testContains()
{
    BinaryTree<Integer> intTree = new BinaryTree<>(10);
    intTree.insert(5);
    intTree.insert(2);
    intTree.insert(7);
    intTree.insert(15);
    intTree.insert(12);
    intTree.insert(17);

    /*
     *             10
     *           /    \
     *         5        15
     *       /  \      /   \
     *     2     7    12    17
     */

    if(!intTree.contains(2))
        fail("2 not found within tree.");
    if(!intTree.contains(7))
        fail("7 not found within tree.");
    if(!intTree.contains(5))
        fail("5 not found within tree.");
    if(!intTree.contains(10))
        fail("10 not found within tree.");
    if(!intTree.contains(15))
        fail("15 not found within tree.");
    if(!intTree.contains(12))
        fail("12 not found within tree.");
    if(!intTree.contains(17))
        fail("17 not found within tree.");

    if(intTree.contains(1))
        fail("1 found within tree");
}
```

```java
    /**
     * Output is known seeing as a predefined traversal method is being used.
     */
    @Test
    void testTraversal()
    {
        BinaryTree<Integer> intTree = new BinaryTree<>(10);
        intTree.insert(5);
        intTree.insert(2);
        intTree.insert(7);
        intTree.insert(15);
        intTree.insert(12);
        intTree.insert(17);

        /*
         *             10
         *           /    \
         *         5        15
         *       /  \      /   \
         *     2     7    12    17
         */

        ArrayList<Integer> expectedList = new ArrayList<>();
        expectedList.add(2);
        expectedList.add(5);
        expectedList.add(7);
        expectedList.add(10);
        expectedList.add(12);
        expectedList.add(15);
        expectedList.add(17);

        ArrayList<Integer> returnedList = (ArrayList<Integer>)intTree.traverse();

        for (int x = 0; x < 7; x++)
        {
            if(!Objects.equals(expectedList.get(x), returnedList.get(x)))
                fail("Discrepancy between expected and returned results at index: " + x);
        }
    }
}
```

Test results.



Self Evaluation.

For 3 marks, I was asked to implement all methods, which I have done so. This qualifies me for 3 marks at the least. For 4 and 5 marks I was asked for good documentation and good testing. I don't think my documentation or testing are absolutely perfect, but given that the descriptor used is good, I would say my inclusion of testing and documentation is enough to qualify my work this week being 5/5

Practical 7 (Week 10)

(Logbook) Question 1: Implement the Traversal interface using depth-first traversal.

Code listing.

```
package graph;

import java.util.*;

/**
 * Created by u1661665(Joshua Pritchard) on 30/11/2018.
 * Version: 30/11/2018
 */
public class DepthFirstTraversal<T> extends AdjacencyGraph<T> implements Traversal<T>
{
    /**
     * Used to hold the list in which the nodes of the graph are traversed.
     */
    ArrayList<T> traversal;

    /**
     * A constructor for a depth first traversal that initializes the data structure used for the traversal list.
     */
    public DepthFirstTraversal()
    {
        traversal = new ArrayList<>();
    }
```

```java
/**
     * An implementation of the traverse() method defined in Traversal.
     *
     * @return Returns a list of nodes in the order of which they were found during traversal.
     * @throws GraphError if the graph given to the traversal is empty.
     */
    @Override
    public List<T> traverse() throws GraphError
    {
        //This is used to ensure the existence of at least one node.
        if(getNoOfNodes() == 0)
            throw new GraphError("Graph is empty.");


        //At least one node exists so this can be used safely.
        T node = getUnvisitedNode();
        do
        {
            //Begin the population of the traversal list from the chosen node.
            populateTraversal(getUnvisitedNode());

            //Check for any more unvisited nodes.
            node = getUnvisitedNode();

        //Repeat if another unvisited node was found after the first pass.
        }while(node != null);

        //Return the now populated traversal list.
        return traversal;
    }
```

```java
/**
     * A recursive method to find and populate the traversal list for graph based on the initial T node.
     *
     * @param node the node to begin the traversal and search from.
     * @throws GraphError
     */
    private void populateTraversal(T node) throws GraphError
    {
        //If the node has not already been visited.
        if(!traversal.contains(node))
        {
            //Record the node as visited and add it to the traversal list.
            traversal.add(node);

            //For each of its neighbours, run this method again.
            for(T neighbour:getNeighbours(node))
            {
                populateTraversal(neighbour);
            }
        }
    }
```

```java
/**
     * Get an unvisited node from the graphs set of nodes.
     *
     * @return a T node within the graph that has been unvisited according to the traversal list in this class. OR null
     * if no unvisited node has been found./
     * if no unvisited node has been found.
     */
    private T getUnvisitedNode()
    {
        //For each node in the set of nodes.
        for(T node : getNodes())
        {
            //If it is unvisited, return this node.
            if(!traversal.contains(node))
                return node;
        }

        //Default case for when no node is found.
        return null;
    }
}
```

Testing class.

```
import graph.AdjacencyGraph;
import graph.DepthFirstTraversal;
import graph.GraphError;
import org.junit.Test;

import static org.junit.Assert.fail;


import java.util.ArrayList;

/**
 * Created by u1661665(Joshua Pritchard) on 30/11/2018.
 * Version: 30/11/2018
 */
public class DepthFirstTest<T> extends AdjacencyGraph<T>
{

    //Specifies the number of repetitions of circularGraphTest()
    private final int NUMBER_CIRCULAR_GRAPH_TESTS = 10;
```

```java
/**
     * Creates a determined circular integer graph with 3 elements and tests a traversal of this graph for correctness.
     *
     * @throws GraphError if any attempt to access an invalid element is made.
     */
    @Test
    public void test() throws GraphError
    {
        DepthFirstTraversal<Integer> intGraph = new DepthFirstTraversal<>();

        intGraph.add(1);
        intGraph.add(2);
        intGraph.add(3);

        intGraph.add(1, 2);
        intGraph.add(2, 3);
        intGraph.add(3, 1);

        ArrayList<Integer> returnedList = (ArrayList<Integer>) intGraph.traverse();

        if (returnedList.get(0) == 1 && returnedList.get(1) == 2 && returnedList.get(2) == 3)
        {
            return;
        }
        else if (returnedList.get(1) == 1 && returnedList.get(2) == 2 && returnedList.get(0) == 3)
        {
            return;
        }
        else if(returnedList.get(2) == 1 && returnedList.get(0) == 2 && returnedList.get(1) == 3)
        {
            return;
        }
        else
        {
            fail("Correct path not returned.");
        }
    }
```

```java
/**
     * Creates a determined circular graph out of 5 elements and tests a traversal of this graph for correctness.
     *
     * @throws GraphError if any attempt to access an invalid element of the graph is made.
     */
    @Test
    public void circularGraphTest() throws GraphError
    {
        //Create a circular graphh
        DepthFirstTraversal<Integer> circularIntGraph = new DepthFirstTraversal<>();

        //Add nodes.
        circularIntGraph.add(1);
        circularIntGraph.add(2);
        circularIntGraph.add(3);
        circularIntGraph.add(4);
        circularIntGraph.add(5);

        /*
        1 2 3 4 5
         */

        //Add edges.
        circularIntGraph.add(1, 2);
        circularIntGraph.add(2, 3);
        circularIntGraph.add(3, 4);
        circularIntGraph.add(4, 5);
        circularIntGraph.add(5, 1);

        /*
            1 -> 2 -> 3 -> 4 -> 5
            ^                   |
            |___________________|
         */

        //---All test cases---
        //A search from 1 should return {1, 2, 3, 4, 5}
        //A search from 2 should return {2, 3, 4, 5, 1}
        //A search from 3 should return {3, 4, 5, 1, 2}
        //A search from 4 should return {4, 5, 1, 2, 3}
        //A search from 5 should return {5, 1, 2, 3, 4}

        //Do a number of tests to make sure all test cases are covered.
        for(int x = 0; x < NUMBER_CIRCULAR_GRAPH_TESTS; x++)
```

```
        {
            //Cast the returned set to an array list.
            ArrayList<Integer> returnedList = (ArrayList<Integer>) circularIntGraph.traverse();

            if(returnedList.get(0) == 1 && returnedList.get(1) == 2 && returnedList.get(2) == 3 && returnedList.get(3) == 4 &&
returnedList.get(4) == 5)
            {break;}
            else if(returnedList.get(0) == 2 && returnedList.get(1) == 3 && returnedList.get(2) == 4 && returnedList.get(3) == 5
&& returnedList.get(4) == 1)
            {break;}
            else if(returnedList.get(0) == 3 && returnedList.get(1) == 4 && returnedList.get(2) == 5 && returnedList.get(3) == 1
&& returnedList.get(4) == 2)
            {break;}
            else if(returnedList.get(0) == 4 && returnedList.get(1) == 5 && returnedList.get(2) == 1 && returnedList.get(3) == 2
&& returnedList.get(4) == 3)
            {break;}
            else if(returnedList.get(0) == 5 && returnedList.get(1) == 1 && returnedList.get(2) == 2 && returnedList.get(3) == 3
&& returnedList.get(4) == 4)
            {break;}
            else
            {
                fail("Returned list did not match any possible test case.");
            }
        }
    }
```

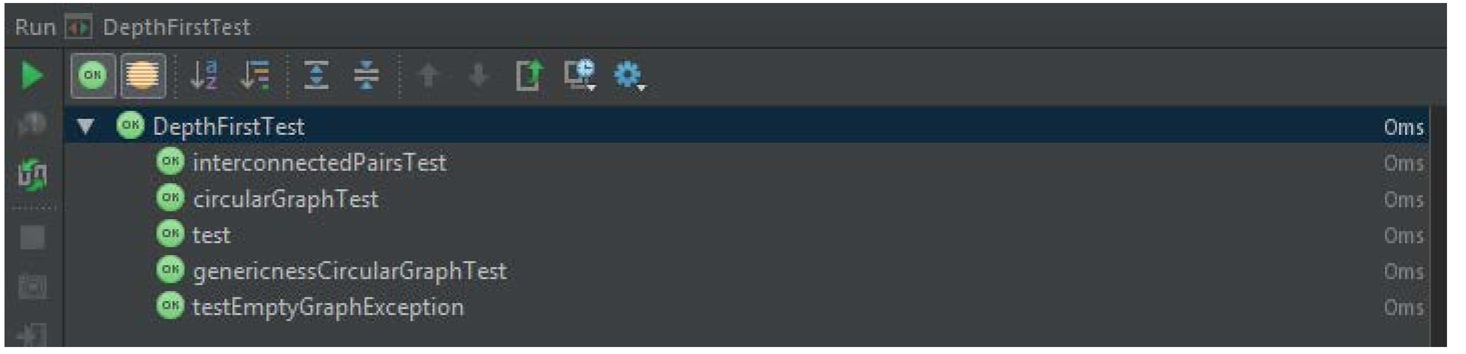
```java
/**
     * Creates a determined graph with 6 elements, then interconnects these to make 3 pairs. Then tests a traversal
     * of this graph for correctness based on the guaranteed difference between each pair.
     *
     * @throws GraphError if any attempt to access and invalid element of the graph is made.
     */
    @Test
    public void interconnectedPairsTest() throws GraphError
    {
        //Create a graph.
        DepthFirstTraversal<Integer> interconnectedPairsGraph = new DepthFirstTraversal<>();

        //Create 6 nodes.
        interconnectedPairsGraph.add(1);
        interconnectedPairsGraph.add(2);
        interconnectedPairsGraph.add(3);
        interconnectedPairsGraph.add(4);
        interconnectedPairsGraph.add(5);
        interconnectedPairsGraph.add(6);

        //Link 1&2, 3&4 and 5&6
        interconnectedPairsGraph.add(1, 2);
        interconnectedPairsGraph.add(2, 1);
        interconnectedPairsGraph.add(3, 4);
        interconnectedPairsGraph.add(4, 3);
        interconnectedPairsGraph.add(5, 6);
        interconnectedPairsGraph.add(6, 5);

        /*
            1<->2 3<->4 5<->6
         */


        //Traverse the graph.
        ArrayList<Integer> returnedList = (ArrayList<Integer>) interconnectedPairsGraph.traverse();

        //each pair will be adjacent in the list.
        if(Math.abs(returnedList.get(0) - returnedList.get(1)) != 1)
        {
            fail("first pair not adjacent");
        }
        else if(Math.abs((returnedList.get(2) - returnedList.get(3))) != 1)
        {
            fail("Second pair not adjacent.");
```

```
        }
        else if(Math.abs(returnedList.get(4) - returnedList.get(5)) != 1)
        {
            fail("Third pair not adjacent.");
        }
    }
```

```java
/**
     * Creates a determined circular graph with 3 string elements and tests a traversal of this graph for correctness.
     *
     * @throws GraphError if any attempt to access an invalid element of the graph is made.
     */
    @Test
    public void genericnessCircularGraphTest() throws GraphError
    {
        DepthFirstTraversal<String> stringGraph = new DepthFirstTraversal<>();

        stringGraph.add("first");
        stringGraph.add("second");
        stringGraph.add("third");

        stringGraph.add("first", "second");
        stringGraph.add("second", "third");
        stringGraph.add("third", "first");

        ArrayList<String> returnedList = (ArrayList<String>) stringGraph.traverse();

        if (returnedList.get(0).equals("first") && returnedList.get(1).equals("second") && returnedList.get(2).equals("third"))
        {
            return;
        }
        else if (returnedList.get(1).equals("first") && returnedList.get(2).equals("second") &&
returnedList.get(0).equals("third"))
        {
            return;
        }
        else if(returnedList.get(2).equals("first") && returnedList.get(0).equals("second") &&
returnedList.get(1).equals("third"))
        {
            return;
        }
        else
        {
            fail("Correct path not returned with a string graph");
        }
    }
}
```

```java
/**
 * Creates an empty graph with no nodes or edges then attempts to traverse it. Should throw a graphError.
 *
 * @throws GraphError if any invalid attempt is made upon the graph, aside from trying to traverse it whilst
 *                       empty.
 */
@Test
public void testEmptyGraphException() throws GraphError
{
    //Create a new graph.
    DepthFirstTraversal<Integer> emptyIntGraph = new DepthFirstTraversal<>();

    //Add no nodes or edges.

    //Attempt to traverse the graph.
    try
    {
        emptyIntGraph.traverse();
        fail("Empty graph traversal allowed.");
    }
    catch(GraphError e)
    {

    }
}
```

Test results.



Self Evaluation.

The self evaluation criteria for 3 marks asks for a full implementation of the Depth-First traversal algorithm. I believe my implementation is full and works correctly returning a traversal list with no duplicated elements. Furthermore I believe that my re-use of the traversal list as a visited list removes space complexity and makes my implementation better.

For 4 and 5 marks I must have Javadoc documentation, which I have included, and full testing. Testing was difficult to develop for this due to the inherent difficulty in testing of graphs. This is because of their use of sets, which are not guaranteed to return the same element every time one is requested. This means complex graphs are very hard to test unless they have testable *properties* consistent across the entire graph. If none of these are present, testing a large, complex graph, would require hardcoding a test against every single possible result of the traversal.

This is long winded, repetitive and pointless considering that smaller graphs with intrinsic properties can be tested in the same way, meaning the more complex graphs will test fine as well.

I believe that my testing is full, adequate and covers all bases of the traversal method that may come into question.

Practical 8 (Week 11)

(Logbook) Question 1: Implement the TopologicalSort interface, using a depth first topological sort. The getSort() method should return a List(T), containing a topological sort of the nodes in the graph.

Code Listing.

```java
package graph;

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

/**
 * Created by u1661665(Joshua Pritchard) on 10/12/2018.
 * Version: 14/01/2019
 */
public class ReferenceCountingTopologicalSort<T> extends AdjacencyGraph<T> implements TopologicalSort<T>
{
    private HashMap<T, Integer> nodes;
    private ArrayList<T> sorted;

    /**
     * Creates a default topological sort object and initializes the required data structures.
     */
    public ReferenceCountingTopologicalSort()
    {
        this.nodes = new HashMap<>();
        this.sorted = new ArrayList<>();
    }
```

```java
/**
     * Gets a topological sort of the graph held within this topological sort object.
     *
     * @return A List(T) containing a topological sort of the graph described by this topological sort object.
     * @throws GraphError if any illegal attempts to access nodes or edges of the graph are made.
     */
    @Override
    public List<T> getSort() throws GraphError
    {
        populateNodesMap();

        //Get an initial node to add.
        T noPredecessors = getNoPredecessorNode();

        //If there are still nodes unsorted.
        while(noPredecessors != null)
        {
            //Add it to the topological sort.
            sorted.add(noPredecessors);

            //Remove the node from the graph.
            removeFromGraph(noPredecessors);

            //Get another node with no predecessors
            noPredecessors = getNoPredecessorNode();
        }

        //Return the topological sort.
        return sorted;
    }
```

```java
/**
     * Used by the sorter to create a modifiable list of object value pairs concerning nodes and their reference count.
     *
     * @throws GraphError if any illegal attempts to access nodes or edges within this graph are made.
     */
    private void populateNodesMap() throws GraphError
    {
        /*
        //---OLD IMPLEMENTATION---
        //Used for keeping track of the reference count of the current node being calculated.
        int referenceCount;

        //Go through each node in the graph.
        for(T node: getNodes())
        {
            //Reset the reference count tracker.
            referenceCount = 0;

            //Check the node against all other nodes in the graph.
            for(T otherNode : getNodes())
            {
                //...Ignoring itself.
                //And if the other node contains an edge leading to the node being calculated.
                if(otherNode != node && getNeighbours(otherNode).contains(node))
                {
                    //Increase the reference count tracker for the node being calculated.
                    referenceCount++;
                }
            }

            //Add the calculated node along with its reference count to the hashmap.
            nodes.put(node, referenceCount);
        }
        //---END OF OLD IMPLEMENTATION---
        */


        //Two stage operation.

        //Add each node to the hashmap.
        for(T node: getNodes())
        {
            nodes.put(node, 0);
        }
```

```
        //For each node.
        for(T node: getNodes())
        {
            //For each neighbour
            for(T neighbour: getNeighbours(node))
            {
                //Increase the reference count stored against it by 1.
                nodes.replace(neighbour, nodes.get(neighbour) + 1);
            }
        }
    }
```

```java
/**
     * Used by the sorter to return a node within the graph that has no predecessors i.e. it has a reference count of 0.
     *
     * @return an element T contained within the graph that has no predecessors.
     * @throws GraphError if any illegal attempts to access the nodes or edges of this graph are made.
     */
    private T getNoPredecessorNode() throws GraphError
    {
        //TODO: Look at this algorithm reaching a point where no ref 0 node can be found.

        //For each node in the graph.
        for(T node : getNodes())
        {
            //If it is unsorted, and has a reference count of 0...
            if(nodes.get(node) == 0 && !sorted.contains(node))
            {
                //Return it
                return node;
            }
        }

        //What if there are no nodes with a zero reference count but there are still some left to be added.
        if(sorted.size() != getNodes().size())
        {
            throw new GraphError("Graph not acyclic");
        }

        //Base return null (Should never occur if this method is used in accordance with a correct
        // reference counting topological sort algorithm).
        return null;
    }
```

```java
/**
     * Used by the sorter to remove a node from consideration i.e. prevent it being added to the graph twice and
     * decrease the reference count of all appropriate nodes.
     *
     * @param node a T element specifying the node within the graph to remove.
     * @throws GraphError if any illegal attempts to access nodes or edges within the graph are made.
     */
    private void removeFromGraph(T node) throws GraphError
    {
        //For each successor of the passed in node.
        for(T successor: getNeighbours(node))
        {
            //Decrement the reference Count held against it in the hashmap by 1.
            nodes.replace(successor, nodes.get(successor) - 1);
        }
    }
}
```

Note – I have chosen to leave the original implementation of populateNodesMap() in the code as a comment to show the progression from a less efficient version of the algorithm to a more efficient one.

Testing class.

```
import graph.Graph;
import graph.GraphError;
import graph.ReferenceCountingTopologicalSort;
import org.junit.Test;

import java.util.List;

import static org.junit.Assert.fail;

/**
 * Created by u1661665(Joshua Pritchard) on 14/01/2019.
 * Version: 15.01.2019
 */
public class RefCountTopoSortTest
{
    /**
     * Create an acyclic graph with 4 elements and make sure that the returned list contains all the same elements
     * as are present in the graph.
     *
     * @throws GraphError if any attempts to make illegal accesses are made.
     */
    @Test
    public void testContents() throws GraphError
    {
        //Create a graph.
        ReferenceCountingTopologicalSort<Integer> intSort = new ReferenceCountingTopologicalSort<>();

        //Populate it with nodes.
        intSort.add(1);
        intSort.add(2);
        intSort.add(3);
        intSort.add(4);

        //Create some edges.
        intSort.add(1, 2);
        intSort.add(1, 3);
        intSort.add(2, 4);

        /*
              __2---4
            1 __
                3
```

```
     */

    //Get the sort.
    List<Integer> returned = intSort.getSort();

    //Print the sort.
    System.out.println(returned.toString());

    //If any of the nodes are not contained within the sort, it's a fail.
    if(!returned.contains(1))
    {
        fail("1 not found.");
    }
    if(!returned.contains(2))
    {
        fail("2 not found.");
    }
    if(!returned.contains(3))
    {
        fail("3 not found.");
    }
    if(!returned.contains(4))
    {
        fail("4 not found.");
    }
}
```

```java
/**
     * Create an acyclic graph with 4 elements and test that the size of the list returned matches the number of
     * nodes given to the graph.
     *
     * @throws GraphError if any illegal attempts to access the nodes or edges of this graph are made.
     */
    @Test
    public void testSizeList() throws GraphError
    {
        //Create a graph.
        ReferenceCountingTopologicalSort<Integer> intSort = new ReferenceCountingTopologicalSort<>();

        //Populate it with nodes.
        intSort.add(1);
        intSort.add(2);
        intSort.add(3);
        intSort.add(4);

        //Create some edges.
        intSort.add(1, 2);
        intSort.add(1, 3);
        intSort.add(2, 4);

        /*
              __2---4
            1 __
                3
         */

        //Get the sort.
        List<Integer> returned = intSort.getSort();

        //If the size of the list is not 4, then something has gone wrong.
        if(returned.size() != 4)
        {
            fail("Size not 4 elements.");
        }
    }
```

```java
/**
     * Create an acyclic graph with 4 elements and test that the topological properties of the sort are present.
     *
     * @throws GraphError if any illegal attempts to access the nodes or edges of this graph are made.
     */
    @Test
    public void testSimpleGraphTopological() throws GraphError
    {
        //Create a graph.
        ReferenceCountingTopologicalSort<Integer> intSort = new ReferenceCountingTopologicalSort<>();

        //Populate it with nodes.
        intSort.add(1);
        intSort.add(2);
        intSort.add(3);
        intSort.add(4);

        //Create some edges.
        intSort.add(1, 2);
        intSort.add(1, 3);
        intSort.add(2, 4);

        /*
              __2---4
            1 __
                3
         */

        //Get the sort.
        List<Integer> returned = intSort.getSort();

        //1 must be the first element, 2,3 and 4 must be after 1, 4 must be after 2.

        //1 must be the first element.
        if(returned.get(0) != 1)
        {
            fail("First element is not 1.");
        }

        //2, 3 and 4 are now inherently after 1.

        //4 must be after 2.
        if(returned.indexOf(4) < returned.indexOf(2))
        {
```

```
            fail("Index of 4 is before index of 2.");
        }
    }
```

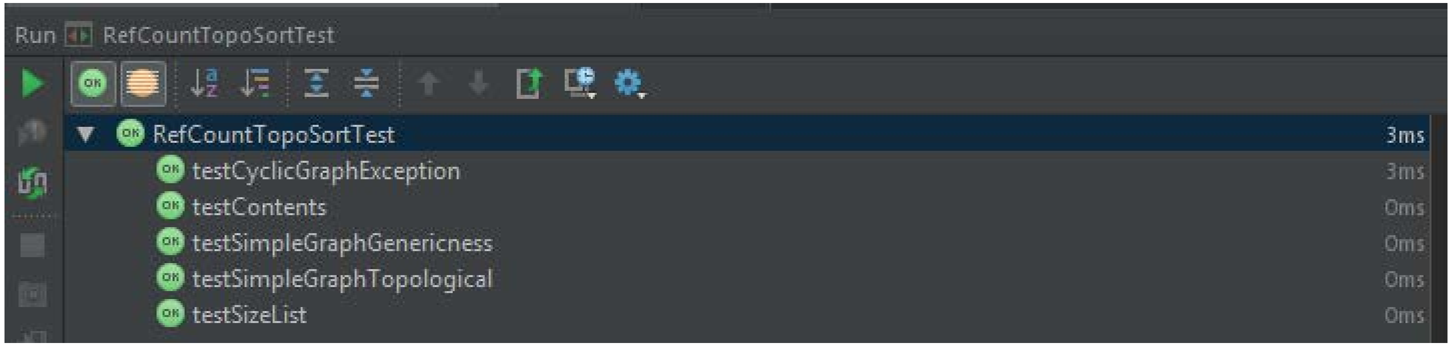
```java
/**
     * Create an acyclic graph of 4 string elements and test that the topological properties are retained for
     * applications of this generic method to other data types.
     *
     * @throws GraphError if any illegal attempts to access the nodes or edges of this graph are made.
     */
    @Test
    public void testSimpleGraphGenericness() throws GraphError
    {
        //Create a graph.
        ReferenceCountingTopologicalSort<String> stringSort = new ReferenceCountingTopologicalSort<>();

        //Populate it with nodes.
        stringSort.add("First");
        stringSort.add("Second");
        stringSort.add("Third");
        stringSort.add("Fourth");

        //Create some edges.
        stringSort.add("First", "Second");
        stringSort.add("First", "Third");
        stringSort.add("Second", "Fourth");

        /*
                  __Second---Fourth
            First __
                     Third
         */

        //Get the sort.
        List<String> returned = stringSort.getSort();

        //First must be the first element, Second, Third and Fourth must be after First, Fourth must be after Second.

        //First must be the first element.
        if(!returned.get(0).equals("First"))
        {
            fail("First element is not 1.");
        }

        //Second, Third and Fourth are now inherently after First.

        //Fourth must be after Second.
        if(returned.indexOf("Fourth") < returned.indexOf("Second"))
```

```
        {
            fail("Index of Fourth is before index of Second.");
        }
    }
```

```java
/**
     * Create a cyclic graph with 4 elements and test the sorters ability to detect the cyclic nature and throw
     * an exception detailing this.
     *
     * @throws GraphError if any illegal attempts are made to access the nodes or edges of this graph
     *                    ASIDE from the testing exception which is expected.
     */
    @Test
    public void testCyclicGraphException() throws GraphError
    {
        //Create a graph.
        ReferenceCountingTopologicalSort<Integer> intSort = new ReferenceCountingTopologicalSort<>();

        //Populate it with nodes.
        intSort.add(1);
        intSort.add(2);
        intSort.add(3);
        intSort.add(4);

        //Create some edges.
        intSort.add(1, 2);
        intSort.add(1, 3);
        intSort.add(2, 4);

        //Create a cyclic edge.
        intSort.add(4, 2);

        /*
              __2<->4
            1 __
                3
         */

        //Try to return a sort, an error should be thrown.
        try
        {
            //Get the sort.
            List<Integer> returned = intSort.getSort();

            //If this point was reached then the required exception was not thrown.
            fail("Exception not thrown.");
        }
        catch(GraphError e)
        {
```

```
        }
    }
}
```

Test results.



Self Evaluation

For 3 marks, a full implementation of the reference counting topological sort is expected, I have done this and further attempted to decrease the overall complexity of my implementation by refactoring code to be more efficient. This refactor also makes the code more readable and easier to maintain.

For 4/5 marks, full Javadoc documentation is expected, which I have included as usual, and full testing. I have done both of these, as evidenced by my code snippets above.