Thomas Howe

U1660884

Algorithms Processes and Data Log Book

CIS2344

***Week 1 & 2 – Randomizer***

**package** intArrays;

**import** java.util.Arrays; // in order to be able to use the fill(...) method

/\*\*

\* An extension of RandomCount

\*

\* **@author** Hugh Osborne

\* **@version** September 2017

\*/

**public** **class** CleverRandomListing **extends** RandomListing

{

**public** CleverRandomListing(**int** size)

{

**super**(size);

}

**protected** **void** randomise()

{

**for**(**int** x = 0; x < getArray().length; x++)

{

**int** i = getRandomIndex();

**int** v = getArray() [x];

getArray() [x] = getArray() [i];

getArray() [i] = v;

}

}

**public** **static** **void** main(String[] args)

{

RandomListing count = **new** CleverRandomListing(1000000);

System.***out***.println(Arrays.*toString*(count.getArray()));

}

} // End of class CleverRandomCount

This is the test class for clever random listing.

**public** **class** CleverRandomListingTest **extends** ListingTest {

**private** **static** **long** *testStart*, *testEnd*;

@BeforeClass

**public** **static** **void** setUpBeforeClass() **throws** Exception {

}

@AfterClass

**public** **static** **void** tearDownAfterClass() **throws** Exception {

}

@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** CleverRandomListing(1));

}

@Test

**public** **void** testOneContents() {

testContents(1,**new** CleverRandomListing(1));

}

@Test

**public** **void** testTwoSize() {

testSize(2,**new** CleverRandomListing(2));

}

@Test

**public** **void** testTwoContents() {

testContents(2,**new** CleverRandomListing(2));

}

@Test

**public** **void** testFourSize() {

testSize(4,**new** CleverRandomListing(4));

}

@Test

**public** **void** testFourContents() {

testContents(4,**new** CleverRandomListing(4));

}

@Test

**public** **void** testHundredSize() {

testSize(100,**new** CleverRandomListing(100));

}

@Test

**public** **void** testHundredContents() {

testContents(100,**new** CleverRandomListing(100));

}

@Test

**public** **void** testThousandSize() {

testSize(1000,**new** CleverRandomListing(1000));

}

@Test

**public** **void** testThousandContents() {

testContents(1000,**new** CleverRandomListing(1000));

}

@Test

**public** **void** testMillionSize() {

testSize(1000000,**new** CleverRandomListing(1000000));

}

}

The output for the tests is as follows:

Test took 33 microseconds to complete.

Test "testHundredSize" took 70478 microseconds

Test took 1 microseconds to complete.

Test "testOneSize" took 39 microseconds

Test took 23 microseconds to complete.

Test "testHundredContents" took 331 microseconds

Test took 1 microseconds to complete.

Test "testTwoSize" took 35 microseconds

Test took 1 microseconds to complete.

Test "testOneContents" took 29 microseconds

Test took 1 microseconds to complete.

Test "testTwoContents" took 30 microseconds

Test took 212 microseconds to complete.

Test "testThousandContents" took 5309 microseconds

Test took 29327 microseconds to complete.

Test "testMillionSize" took 32491 microseconds

Test took 27 microseconds to complete.

Test "testFourSize" took 74 microseconds

Test took 52 microseconds to complete.

Test "testThousandSize" took 91 microseconds

Test took 5 microseconds to complete.

Test "testFourContents" took 38 microseconds

***Week 3 & 4 – Generic Swap***

Swap: Write a generic method to exchange two elements of an array. The method should take an array, and two integer indices into the array, and swap the two entries in the array at those indices.

**package** genericMethods;

**import** java.util.Arrays;

**public** **class** GenericMethods {

**public** **static** <T> **boolean** equals(T object1,T object2) {

**if** (object1==**null**) {

**return** object2==**null**;

} **else** {

**return** object1.equals(object2);

}

}

**public** **static** <T> **void**

swap(T[]array,**int** i, **int** j)

{

T obj1 = array[i];

T obj2 = array[j];

array[i] = obj2; //

array[j] =obj1;

}

**public** **static** **void** main(String[] args)

{

Object[] ints = {1, 2, 3, 4, 5}; //array list that is going to be swapped

*swap*(ints, 0, 3); // defining the index number in the array that are going to be swapped

System.***out***.println(Arrays.*toString*(ints)); // prints out the swapped array to be checked

}

}

***Week 5 – Sorting***

Quick Sort

**class** QuickSort

{

/\* This function takes last element as pivot,

places the pivot element at its correct

position in sorted array, and places all

smaller (smaller than pivot) to left of

pivot and all greater elements to right

of pivot \*/

**int** partition(**int** array[], **int** low, **int** high)

{

**int** pivot = array[high];

**int** i = (low-1); // index of smaller element

**for** (**int** j=low; j<high; j++)

{

// If current element is smaller than or equal to pivot number

**if** (array[j] <= pivot)

{

i++;

// this swaps array[i] and array[j]

**int** temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}

// swap array[i+1] and array[high] (or pivot)

**int** temp = array[i+1];

array[i+1] = array[high];

array[high] = temp;

**return** i+1;

}

**void** sort(**int** arr[], **int** low, **int** high)

{

**if** (low < high)

{

**int** pi = partition(arr, low, high);

sort(arr, low, pi-1);

sort(arr, pi+1, high);

}

}

**static** **void** printArray(**int** arr[])

{

**int** n = arr.length;

**for** (**int** i=0; i<n; ++i)

System.***out***.print(arr[i]+" ");

System.***out***.println();

}

**public** **static** **void** main(String args[])

{

//this is the array that needs sorting

**int** arr[] = {10, 7, 8, 9, 1, 5, 46, 60, 4, 3, 1002, 22, 123};

**int** n = arr.length;

QuickSort ob = **new** QuickSort();

ob.sort(arr, 0, n-1);

//this prints out the sorted a

System.***out***.println("Array is now sorted!!");

*printArray*(arr);

}

}

Selection sort

**class** SelectionSort

{

**void** sort(**int** arr[])

{

**int** n = arr.length;

**for** (**int** i = 0; i < n-1; i++)

{

// Find the minimum element in unsorted array

**int** min\_idx = i;

**for** (**int** j = i+1; j < n; j++)

**if** (arr[j] < arr[min\_idx])

min\_idx = j;

// Swap the found minimum element with the first

// element

**int** temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

}

// Prints the array

**void** printArray(**int** arr[])

{

**int** n = arr.length;

**for** (**int** i=0; i<n; ++i)

System.***out***.print(arr[i]+" ");

System.***out***.println();

}

// Driver code to test above

**public** **static** **void** main(String args[])

{

SelectionSort ob = **new** SelectionSort();

**int** arr[] = {64,25,12,22,11, 14, 10 , 5, 2, 88, 123};

ob.sort(arr);

System.***out***.println("Sorted array");

ob.printArray(arr);

}

}

My approximation for the time complexity of the 2 sorting algorithm that I used were (n log (n)) for quick sort, and for selection sort my approximation is (n^2).

***Week 6 – Lists***

**public** **class** Lists<T> **implements** List<T>

{

**private** Node<T> startNode=**null**;

**private** **int** NodeNo;

@Override

**public** **void** add(**int** index, T value) **throws** ListAccessError

{

Node<T> addNode = **new** Node<T>(value);

**if**(isEmpty())

{

startNode = addNode;

}

**else**

{

**if**(getNode(index-1).getNext()==**null**)

{

getNode(index-1).setNext(addNode);

}

**else**

{

addNode.setNext(getNode(index));

getNode(index).setNext(addNode);

}

NodeNo++;

}

}

**private** Node<T> getNode(**int** index) **throws** ListAccessError

{

Node<T> temp = startNode;

**if**(index>NodeNo | index<0 | isEmpty())

{

**throw** **new** ListAccessError("Index out of bounds");

}

**for**(**int** i =0;i<index;i++)

{

temp = temp.getNext();

}

**return** temp;

}

@Override

**public** T remove(**int** index) **throws** ListAccessError

{

**if**(index==0)

{

T returnVal = startNode.getValue();

startNode = startNode.getNext();

NodeNo--;

**return** returnVal;

}

**else**

{

Node<T> temp = getNode(index-1);

Node<T> overwrite = getNode(index);

T returnVal = (T) overwrite.getValue();

**if**(overwrite.getNext()!=**null**)

{

temp.setNext(overwrite.getNext());

}

**else**

{

temp.setNext(**null**);

}

NodeNo--;

**return** returnVal;

}

}

@Override

**public** T get(**int** index) **throws** ListAccessError

{

**return** (T) getNode(index).getValue();

}

@Override

**public** **boolean** isEmpty() {

**return** startNode==**null**;

}

}

***Week 8 – Binary Trees***

Binary tree class

**package** binaryTree;

**public** **class** BinaryTree<T **extends** Comparable<? **super** T>> **implements**

BTree<T> {

TreeNode<T> root = **null**;

@Override

**public** **void** insert(T value) {

**if** (root == **null**)

{

root = **new** TreeNode(value);

}

**else** **if**(value.compareTo(value()) < 0)

{

root.left().insert(value);

}

**else**

{

root.right().insert(value);

}

}

@Override

**public** T value() {

**return** root.value();

}

@Override

**public** BTree<T> left() {

**return** root.left();

}

@Override

**public** BTree<T> right() {

**return** root.right();

}

}

Btree class

**package** binaryTree;

**public** **interface** BTree<T **extends** Comparable<? **super** T>> {

**public** **void** insert(T value);

**public** T value();

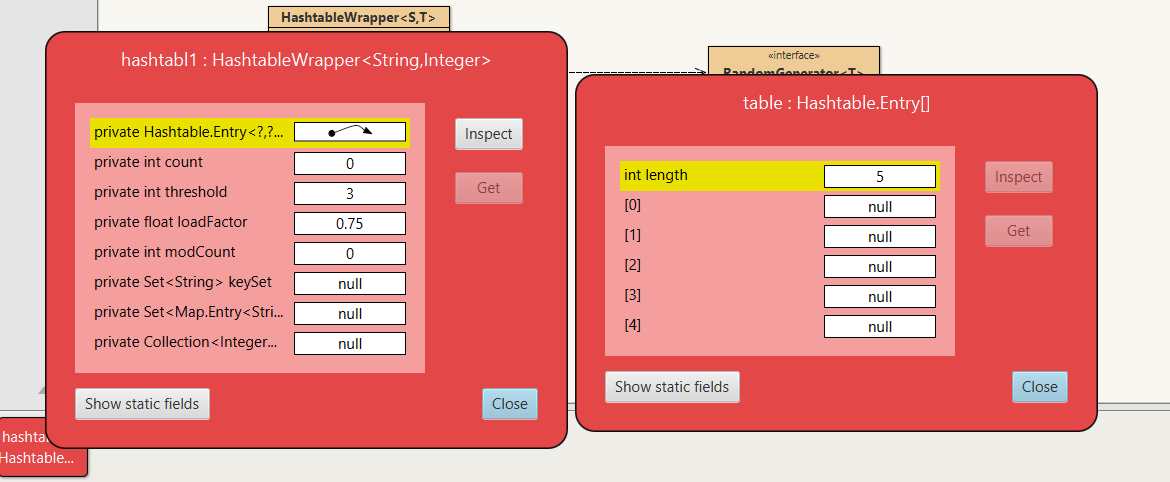
**public** BTree<T> left();

**public** BTree<T> right();

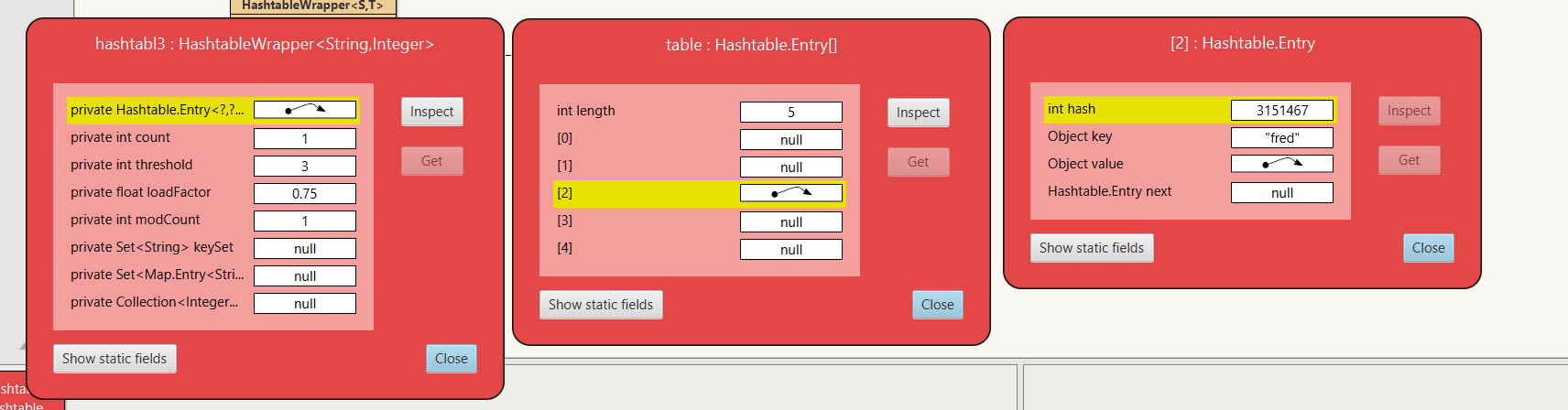
}

***Week 9 – Hash Tables***

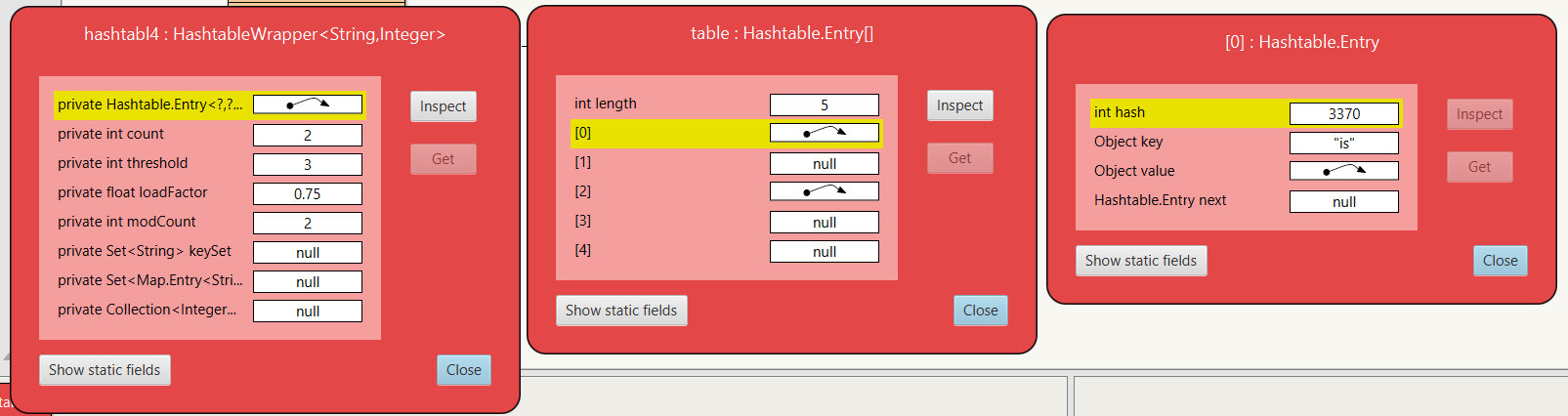
1) In this picture I have created a hashtable with string and a integer.



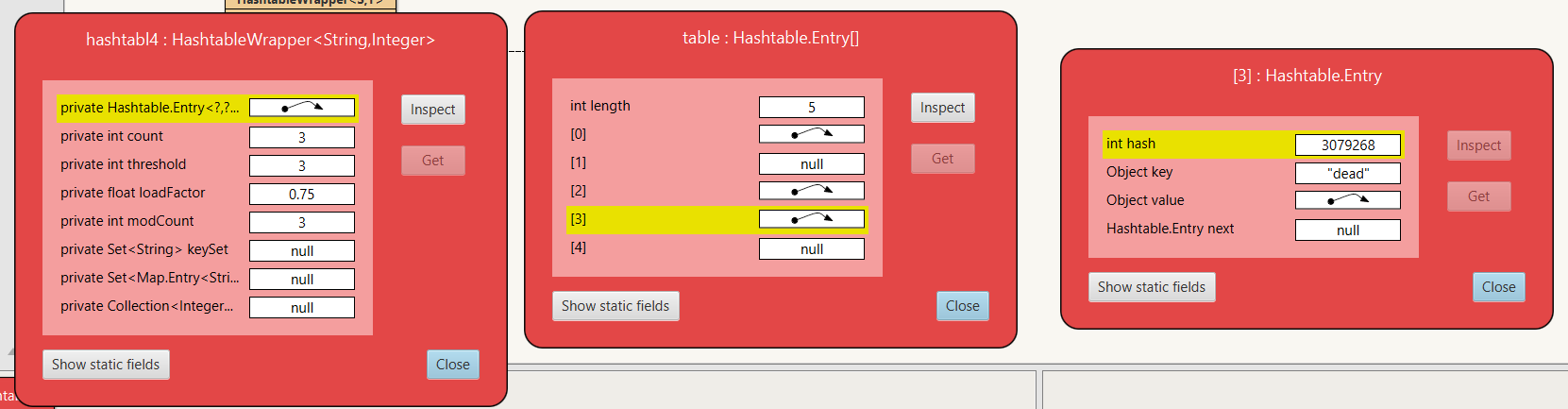
2) Here I have added “fred” as the string and 37 as the integer, the hash number is 3151467. This item went into position 2 in the hashtable.



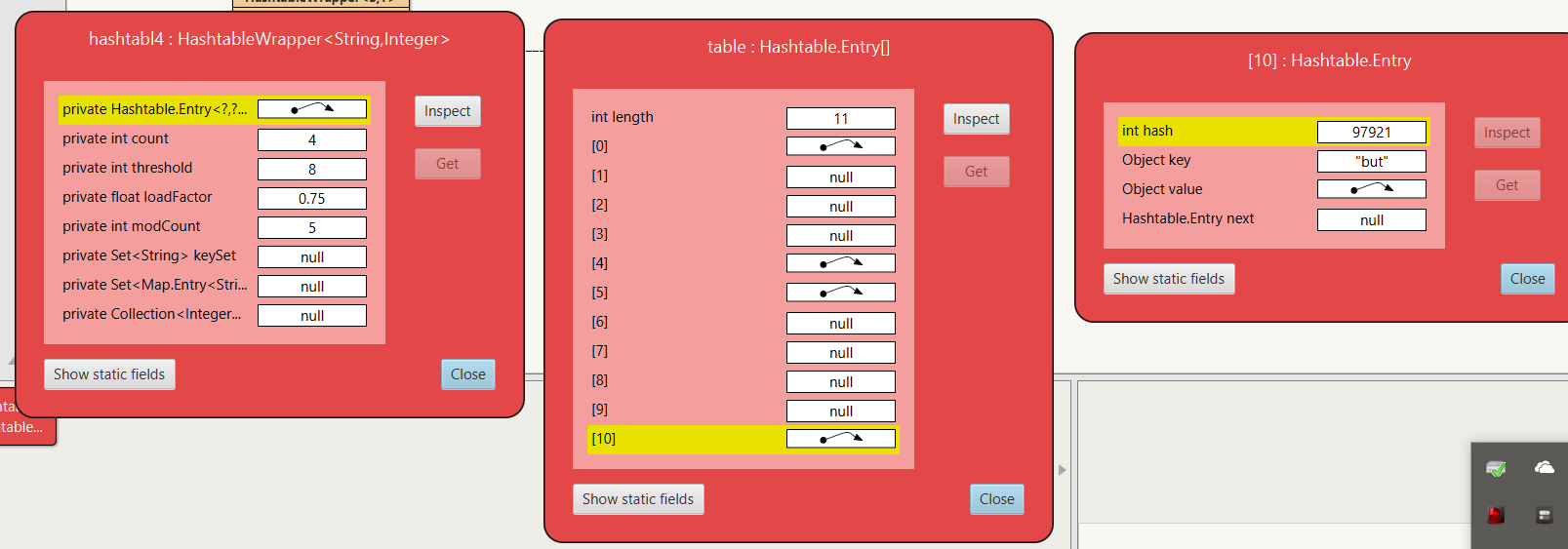
3) The 2nd entry I added is “is” with an integer of 69 and the hash number is 3370. This item went into position 0 in the hashtable.



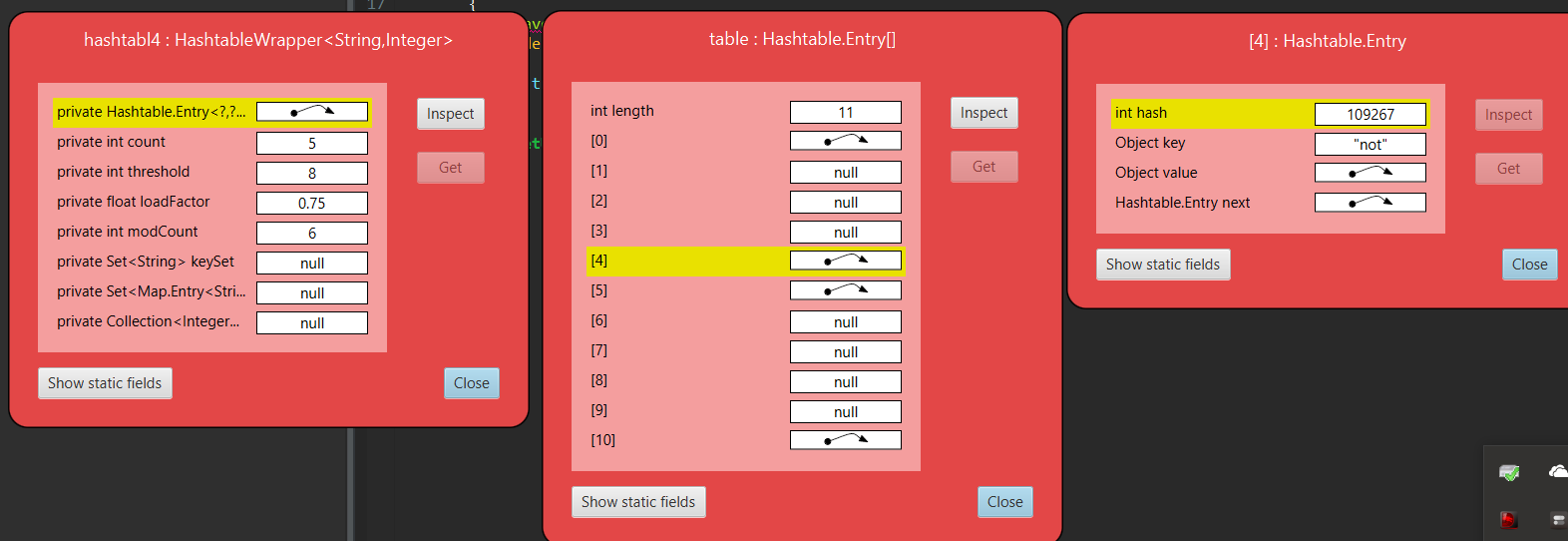
4) The 4th entry into the hashtable is “dead” with it the integer was 0 and the has number was 3079268. The 3rd position in this hashtable was taken up by this entry.



5) The 5th entry into the hashtable was the string “but” and the integer was 999 and the hash number was 97921. This item created new entries into the hashtable, thus moving this entry into the 10th position. Also some of the items have moved around in the hashtable “fred” has moved from position 2 to position 0, and “is” has now moved to position 4, “dead” has moved from 3 to 5



6) in this step I added “not” and the int -42 this went into the 4th position, with the hash number of 109267. Adding this entry removed entry “is” from the hashtable.



7) in this entry I added “me!” and the integer of -1 this went into the 3rd position in the hashtable with a hash number 107913 nothing else in the array has changed.

***Week 10 – Graph Traversal***

**public** **class** DepthFirstTraversal<T> **extends** AdjacencyGraph<T> **implements** Traversal<T>

{

**private** List<T> nodes, traversal = **new** ArrayList<T>();

**private** **int** nodeSize = getNodes().size();

@Override

**public** List<T> traverse() **throws** GraphError

{

**for** (**int** i=0; i<nodeSize; i++)

{

**if**(traversal.size() < nodeSize)

{

T startNode = (T) getNodes().toArray()[i];

**if**(!traversal.contains(startNode))

{

Traversal(startNode);

}

}

}

**return** traversal;

}

**private** **void** Traversal(T node) **throws** GraphError

{

traversal.add(node);

nodes.add(node);

T[] neighbours = (T[])getNeighbours(node).toArray();

**for**(**int** i=0;i<neighbours.length;i++)

{

**if**(neighbours[i]!=**null**&&!traversal.contains(neighbours[i]))

{

Traversal(neighbours[i]);

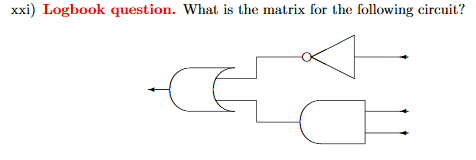
}

}

}

}

***Week 17 – Modelling Circuits***



The NOT gate has the representation matrix as

The AND gate has the representation of

Finally, the OR gate is represented as

Which means that the single matrix of all of them would be

***Week 20 – Quantum Computing***

For the first test I made 0 the pure state then passed it through a Hadamard gate to see what would happen to it. After the first pass through the Hadamard gate I put the answer through another one to see how that would affect the answer.

Hadamard \* ZERO

ans =

0.7071

0.7071

Hadamard \* ans

ans =

1.0000

0

Hadamard \* ONE

ans =

0.7071

-0.7071

Hadamard \* ans

ans =

0

1.0000

***Self-Assessment***

Week 7

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| week | Overall | Docs | Struct | Names | Tests | Funct |
| 1&2 Search Timer | B | C | C | B | C | C |
| 3 & 4 Generic Swap | B | C | C | B | C | C |
| 5  Sorting | B | C | C | B | B | B |
| 6  Linked lists | G | G | G | G | G | G |
| 8 Binary trees | C | C | C | C | C | C |

Consolidation Week

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Week | Overall | Docs | Struct | Names | Tests | Funct |
| 9  Hash tables | C | N/A | | | | |
| 10  Depth First Traversal | C | D | C | C | D | C |
| 11  Reference  counting  topological  sort | G | G | G | G | G | G |

Week 17

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Week | Overall | Docs | Struct | Names | | Tests | | Funct |
| 13 Counter Behaviour | G | N/A | | | | | | |
| 14  Dekker Trains | G | G | G | G | | G | | G |
| 15 Semaphore Behaviour | G | N/A | | | | | | |
| 16  Locks And Conditions | G | G | G | G | G | | G | |

|  |  |  |
| --- | --- | --- |
| Week | Overall | Criteria |
| 17  Modelling Circuits | C | Have you derived a matrix for a half-adder?  Have you correctly applied the matrix meth-  ods 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 | C | Have you fully analysed the values that will ap-  pear at points A, B, and C in the circuit? Have  you discussed the relationship between the val-  ues appearing at A and C ? Have you considered  what the implications of a purely probabilistic  model would be for maintaining this relation-  ship? |

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