

Contents

1	Basic Test Results	2
2	README	3
3	ClosedHashSet.java	5
4	CollectionFacadeSet.java	9
5	FacadeLinkedList.java	11
6	OpenHashSet.java	13
7	RESULTS	16
8	SimpleHashSet.java	17
9	SimpleSetPerformanceAnalyzer.java	20

1 Basic Test Results

```
1  =====
2  ===== EX4 TESTER =====
3  =====
4
5  ===== CHECKING JAR & FILES =====
6
7  ===== ANALYZE README =====
8
9  ===== COMPILE CODE =====
10
11 Code complied successfully
12 ===== RUN TESTS =====
13
14 tests output :
15
16 OpenHashSet
17 =====
18 Perfect!
19
20 ClosedHashSet
21 =====
22 Perfect!
23
24
25 *****
26 Testing performance analysis results
27 *****
28 performance analysis results tests passed
```

2 README

```
1  muaz.abdeen
2
3
4
5  =====
6  =      File description      =
7  =====
8  - SimpleHashSet -
9      A superclass for implementations of hash-sets implementing the SimpleSet interface.
10 - OpenHashSet -
11     A hash-set based on open hashing (chaining). Extends SimpleHashSet.
12 - ClosedHashSet -
13     A hash-set based on closed-hashing with quadratic probing. Extends SimpleHashSet.
14 - CollectionFacadeSet -
15     Wraps an underlying Collection and serves to both simplify its API and give it
16     a common type with the implemented SimpleHashSets.
17 - FacadeLinkedList -
18     Wraps a java LinkedList to use in OpenHashSet.
19 - SimpleSetPerformanceAnalyzer -
20     Simple class analyzes the performance of these five data structures:
21     OpenHashSet, ClosedHashSet, LinkedList, HashSet & TreeSet.
22
23
24
25 =====
26 =      Design      =
27 =====
28
29 Most of the design is enforced by the given API.
30
31 In the abstract super class SimpleHashSet that implements the SimpleSet interface,
32 I implemented some methods to avoid code duplication, and made the remaining abstract.
33 I tried to keep the API minimal, I added two protected methods in the super class,
34 other additional helper methods, in super or sub-classes, are private.
35
36 There is two Facade classes, the first is given, which is, CollectionFacadeSet, I wrapped
37 the main functionalities of JAVA Collection class, which implements the methods in the
38 SimpleSet interface.
39
40 The second Facade class, FacadeLinkedList, wraps a java LinkedList to use in making the
41 array of linked lists in OpenHashSet class.
42
43 Finally, in the SimpleSetPerformanceAnalyzer class, I designed it in such a way, one could
44 enable and disable any test easily.
45
46
47 =====
48 =      Implementation details      =
49 =====
50
51 - OpenHashSet -
52     I implemented a Facade class that wraps a LinkedList, called FacadeLinkedList.
53     I declared an array of this FacadeLinkedList to be the hashtable, then initialized
54     it with INITIAL_CAPACITY empty (null) cells.
55     When adding an item, if it is mapped to empty cell, a new FacadeLinkedList is initialized
56     with this item as its head node, else it is added at the first of the existing FacadeLinkedList.
57     The size attribute is updated after each deletion or addition.
58     I used the methods of the LinkedList to find, delete, and add to the linked lists in the array.
59
```

```

60 - ClosedHashSet -
61     The most important issue here was: how to manage the deletion operation in such a way that
62     puts no constraints on string values, and takes a constant time and space?
63     I created a unique string called DUMMY by invoking the String constructor directly, and
64     when deleting a value I put in its place a pointer to this DUMMY, and when searching for a
65     value I compared between the addresses of the strings, not the contents of them, that is to
66     say, I used the '=' operator, not equals() method. In this way I can add a string equals to
67     'DUMMY' to the table, without causing any problem, because it will point to another
68     memory address different from our unique DUMMY value.
69
70 - SimpleSetPerformanceAnalyzer -
71     I defined two factory functions, one that creates a SimpleSet array with OpenHashSet
72     and ClosedHashSet initialized with default constructor, and other data structures which
73     wrapped in CollectionFacadeSet with their default constructor, too. In other words, each
74     data structure here is empty. The second factory function initializes each data structure
75     with a given array of strings.
76     I created a separate test function for every functionality to test as stated in the exercise
77     file, so that one could enable and disable any test easily, by comment or uncomment the
78     function call in the main function.
79
80
81 =====
82 =   Answers to questions   =
83 =====
84 Q - Account, in separate, for OpenHashSet's and ClosedHashSet's bad results for data1.txt -
85 A - In data1.txt all the Strings have the same hashCode.
86     in OpenHashSet (chaining), all of the strings will end in the same bucket, so every time we
87     add an item, we first check if this list contains it, which means iterating over it, this
88     takes a linear time, i.e.  $O(n)$  for  $n$  = number of strings in data.txt.
89     in ClosedHashSet, all of the strings are mapped initially to the same bucket, which increases
90     the secondary clustering, and creates a long run of filled slots away from the initial
91     position, in other words, we must iterate along the probe sequence every time, this takes
92     also linear time.
93
94 Q - Summarize the strengths and weaknesses of each of the data structures as reflected by the
95     results. Which would you use for which purposes? -
96 A - (1) OpenHashSet and ClosedHashSet: behaves almost the same, one can notice that OpenHashSet
97     behaves better than ClosedHashSet in the worst case, both take linear time, but it
98     seems that the coefficient of  $n$  is less in OpenHashSet.
99     In the average and best case they behave almost like the Java HashSet & TreeSet, there
100    is a difference of course, but not huge.
101    (2) Java HashSet: achieves the best results in all cases, constant time.
102    (3) Java TreeSet: almost the same as HashSet, but takes a bit longer, because it orders
103    the elements in every bucket, this takes more time.
104    (4) Java LinkedList: takes the longest time among all other data structures, because in every
105    addition or deletion we check the existence of the element, which needs to
106    iterate over the list.
107    In general, Java HashSet behaves the best, and I would use it in the cases I wanted an
108    unordered set, otherwise, TreeSet would be my choice.
109
110 Q - How did your two implementations compare between themselves? -
111 A - behaves almost the same, one can notice that OpenHashSet behaves better than
112    ClosedHashSet in the worst case, both take linear time, but it seems that
113    the coefficient of  $n$  is less in OpenHashSet.
114    In the average and best case they behave almost the same.
115
116 Q - How did your implementations compare to Java's built-in HashSet? -
117 A - In the worst case, there is a huge difference, mine took linear time, while Java's
118    took constant time.
119    In the average and best case they behaved almost like the Java HashSet, there
120    is a difference of course, but not huge.
121
122 Q - Did you find Java's HashSet performance on data1.txt surprising? Can you explain it?
123 A - Yes, I did. I think they used perfect hashing, or a long hash table !!

```

3 ClosedHashSet.java

```
1
2 /**
3  * A hash-set based on closed-hashing with quadratic probing
4  */
5 public class ClosedHashSet extends SimpleHashSet {
6
7     // ##### //
8     // #### CONSTANTS #### //
9     // ##### //
10
11     private static final int SMALL = -1;
12     private static final int BIG = 1;
13     private static final int NOT_FOUND = -1;
14
15     /* create DUMMY string by invoking the String constructor explicitly
16      * DUMMY now points to distinguish memory address */
17     private static final String DUMMY = new String("DUMMY");
18
19     // ##### //
20     // #### ATTRIBUTES #### //
21     // ##### //
22
23     /** A standard Java array represents the hash table */
24     private String[] _hashTable;
25
26     // ##### //
27     // #### CONSTRUCTORS #### //
28     // ##### //
29
30     /**
31      * A default constructor.
32      * Constructs a new, empty table with default initial capacity (16),
33      * upper load factor (0.75) and lower load factor (0.25).
34      */
35     public ClosedHashSet() {
36         super();
37         _hashTable = new String[INITIAL_CAPACITY];
38     }
39
40     /**
41      * Constructs a new, empty table with the specified load factors,
42      * and the default initial capacity (16).
43      * @param upperLoadFactor The upper load factor of the hash table.
44      * @param lowerLoadFactor The lower load factor of the hash table.
45      */
46     public ClosedHashSet(float upperLoadFactor, float lowerLoadFactor) {
47         super(upperLoadFactor, lowerLoadFactor);
48         _hashTable = new String[INITIAL_CAPACITY];
49     }
50
51     /**
52      * Data constructor - builds the hash set by adding the elements one by one.
53      * Duplicate values should be ignored.
54      * The new table has the default values of initial capacity (16),
55      * upper load factor (0.75), and lower load factor (0.25).
56      * @param data Values to add to the set.
57      */
58     public ClosedHashSet(String[] data) {
59         super();
```

```

60     _hashTable = new String[INITIAL_CAPACITY];
61     for (String key: data) {
62         add(key);
63     }
64 }
65
66 // #####
67 // #### METHODS ####
68 // #####
69 /**
70  * @return The current capacity (number of cells) of the table.
71  */
72 @Override
73 public int capacity() {
74     return _hashTable.length;
75 }
76
77 /**
78  * A quadratic probing function for collision resolution.
79  * @param hashCode Original hash code before probing.
80  * @param attempt Number of probing attempt.
81  * @return the hash code after probing
82  */
83 private int probe(int hashCode, int attempt) {
84     return hashCode + (attempt + attempt*attempt)/2;
85 }
86
87 /**
88  * Add a specified element to the set if it's not already in it.
89  * @param newValue New value to add to the set
90  * @return False iff newValue already exists in the set
91  */
92 @Override
93 public boolean add(String newValue) {
94     if (newValue != null && !contains(newValue)) {
95         map(newValue);
96         return true;
97     }
98     return false;
99 }
100
101 /**
102  * Maps the newValue to a certain cell in the hash table
103  * @param newValue the value to be mapped
104  */
105 private void map(String newValue) {
106     // resize and rehash if needed
107     int relativeSize = relativeSize((float)(_size+1)/capacity());
108     if (relativeSize == BIG) {
109         resizeTable(relativeSize);
110     }
111     // then add the new value
112     /**
113      * The way in which we choose c1, c2 ensures that as long as the table
114      * is not full, a place for a new value will be found during the first
115      * capacity attempts. (From exercise description)
116      */
117     int attempt = 0;
118     while (attempt < capacity()) {
119         int idx = clamp(probe(newValue.hashCode(), attempt));
120         // reference comparison (address comparison) of DUMMY
121         if (_hashTable[idx] == null || _hashTable[idx] == DUMMY) {
122             _hashTable[idx] = newValue;
123             ++_size;
124             return;
125         }
126         ++attempt;
127     }

```

```

128     }
129
130     /**
131      * Resizes the current hash table
132      * @param relativeSize relative size of the current table to the demanded one
133      */
134     private void resizeTable(int relativeSize) {
135         // resize
136         String[] oldTable = _hashTable;
137         _size = 0;
138         if (relativeSize == BIG) {
139             _hashTable = new String[capacity() * 2];
140         } else if (relativeSize == SMALL && capacity() > 1) {
141             _hashTable = new String[capacity() / 2];
142         } else {
143             return;
144         }
145         // rehash
146         for (String element: oldTable) {
147             if (element != null && element != DUMMY)
148                 map(element);
149         }
150     }
151
152     /**
153      * Look for a specified value in the table.
154      * @param searchVal Value to search for
155      * @return True iff searchVal is found in the set
156      */
157     @Override
158     public boolean contains(String searchVal) {
159         return find(searchVal) != NOT_FOUND;
160     }
161
162     /**
163      * Look for a specified value in the table.
164      * @param value Value to search for
165      * @return the index if found, else NOT_FOUND(= -1)
166      */
167     private int find(String value) {
168         // Quadratic probing generates at most 'capacity' probing sequences
169         int attempt = 0;
170         int idx = clamp(probe(value.hashCode(), attempt));
171         while (_hashTable[idx] != null && attempt <= capacity()) {
172             if (_hashTable[idx].equals(value) && _hashTable[idx] != DUMMY) {
173                 return idx;
174             }
175             ++attempt;
176             idx = clamp(probe(value.hashCode(), attempt));
177         }
178         return NOT_FOUND;
179     }
180
181     /**
182      * Remove the input element from the set.
183      * @param toDelete Value to delete
184      * @return True iff toDelete is found and deleted
185      */
186     @Override
187     public boolean delete(String toDelete) {
188         int idx = find(toDelete);
189         if (idx != NOT_FOUND) {
190             // remove the value
191             _hashTable[idx] = DUMMY;
192             --_size;
193             // resize and rehash if needed
194             int relativeSize = relativeSize((float)(_size)/capacity());
195             if (relativeSize == SMALL)

```

```
196         resizeTable(relativeSize);
197         return true;
198     }
199     return false;
200 }
201
202 }
```


4 CollectionFacadeSet.java

```
1  import java.util.Collection;
2
3  /**
4   * Wraps an underlying Collection and serves to both simplify its API
5   * and give it a common type with the implemented SimpleHashSets.
6   */
7  public class CollectionFacadeSet implements SimpleSet {
8
9      // #####
10     // ##### ATTRIBUTES #####
11     // #####
12
13     protected Collection<String> _collection;
14
15     // #####
16     // ##### CONSTRUCTOR #####
17     // #####
18
19     /**
20      * Creates a new facade wrapping the specified collection.
21      * @param collection The Collection to wrap.
22      */
23     public CollectionFacadeSet(Collection<String> collection) {
24         _collection = collection;
25     }
26
27     // #####
28     // ##### METHODS #####
29     // #####
30
31     /**
32      * Add a specified element to the set if it's not already in it.
33      * @param newValue New value to add to the set
34      * @return False iff newValue already exists in the set
35      */
36     @Override
37     public boolean add(String newValue) {
38         if (!_collection.contains(newValue)) {
39             return _collection.add(newValue);
40         }
41         return false;
42     }
43
44     /**
45      * Look for a specified value in the set.
46      * @param searchVal Value to search for
47      * @return True iff searchVal is found in the set
48      */
49     @Override
50     public boolean contains(String searchVal) {
51         return _collection.contains(searchVal);
52     }
53
54     /**
55      * Remove the input element from the set.
56      * @param toDelete Value to delete
57      * @return True iff toDelete is found and deleted
58      */
59     @Override
```

```

60     public boolean delete(String toDelete) {
61         return _collection.remove(toDelete);
62     }
63
64     /**
65      * @return The number of elements currently in the set
66      */
67     @Override
68     public int size() {
69         return _collection.size();
70     }
71 }

```

5 FacadeLinkedList.java

```
1  import java.util.Iterator;
2  import java.util.LinkedList;
3
4  /**
5   * Wraps a java LinkedList to use in OpenHashSet.
6   */
7  public class FacadeLinkedList {
8
9      // #####
10     // ##### ATTRIBUTES #####
11     // #####
12
13     /** The wrapped linked list */
14     private LinkedList<String> _linkedList;
15
16     // #####
17     // ##### CONSTRUCTOR #####
18     // #####
19
20     /**
21      * Creates a new facade wrapping a Java LinkedList.
22      */
23     public FacadeLinkedList() {
24         _linkedList = new LinkedList<String>();
25     }
26
27     // #####
28     // ##### METHODS #####
29     // #####
30
31     /**
32      * @return Returns an iterator over the elements in this list
33      */
34     public Iterator<String> iterator() {
35         return _linkedList.iterator();
36     }
37
38     /**
39      * Insert a new node at the head of the list.
40      * @param newValue New value to add to the list.
41      */
42     public void add(String newValue) {
43         // no need to check if contains because we did that in hash set add method
44         _linkedList.addFirst(newValue);
45         assert _linkedList.peek() != null && _linkedList.peek().equals(newValue);
46     }
47
48     /**
49      * Look for a specified value in the list.
50      * @param searchVal Value to search for
51      * @return True iff searchVal is found in the list
52      */
53     public boolean contains(String searchVal) {
54         return _linkedList.contains(searchVal);
55     }
56
57     /**
58      * Removes the first occurrence of the specified element from this list, if it is present.
59      * @return true if this list contained the specified element.
```

```

60     */
61     public boolean delete(String toDelete) {
62         return _linkedList.remove(toDelete);
63     }
64
65     /**
66      * @return The number of elements currently in the list
67      */
68     public int size() {
69         return _linkedList.size();
70     }
71
72     // End of FacadeLinkedList class
73 }

```

6 OpenHashSet.java

```
1  import java.util.Iterator;
2
3  /**
4   * A hash-set based on open hashing (chaining)
5   */
6  public class OpenHashSet extends SimpleHashSet {
7
8      // #####
9      // #### CONSTANTS ####
10     // #####
11
12     private static final int SMALL = -1;
13     private static final int BIG = 1;
14
15     // #####
16     // #### ATTRIBUTES ####
17     // #####
18
19     /** The hash table that contains a linked list in every cell */
20     private FacadeLinkedList[] _hashTable;
21
22     // #####
23     // #### CONSTRUCTORS ####
24     // #####
25
26     /**
27      * A default constructor.
28      * Constructs a new, empty table with default initial capacity (16),
29      * upper load factor (0.75) and lower load factor (0.25).
30      */
31     public OpenHashSet() {
32         super();
33         _hashTable = new FacadeLinkedList[INITIAL_CAPACITY];
34     }
35
36     /**
37      * Constructs a new, empty table with the specified load factors,
38      * and the default initial capacity (16).
39      * @param upperLoadFactor The upper load factor of the hash table.
40      * @param lowerLoadFactor The lower load factor of the hash table.
41      */
42     public OpenHashSet(float upperLoadFactor, float lowerLoadFactor) {
43         super(upperLoadFactor, lowerLoadFactor);
44         _hashTable = new FacadeLinkedList[INITIAL_CAPACITY];
45     }
46
47     /**
48      * Data constructor - builds the hash set by adding the elements one by one.
49      * Duplicate values should be ignored.
50      * The new table has the default values of initial capacity (16),
51      * upper load factor (0.75), and lower load factor (0.25).
52      * @param data Values to add to the set.
53      */
54     public OpenHashSet(String[] data) {
55         super();
56         _hashTable = new FacadeLinkedList[INITIAL_CAPACITY];
57         for (String key: data) {
58             add(key);
59         }
60     }
61 }
```

```

60     }
61
62     // #####
63     // ### METHODS ###
64     // #####
65
66     /**
67      * @return The current capacity (number of cells) of the table.
68      */
69     @Override
70     public int capacity() {
71         return _hashTable.length;
72     }
73
74     /**
75      * Add a specified element to the set if it's not already in it.
76      * @param newValue New value to add to the set
77      * @return False iff newValue already exists in the set
78      */
79     @Override
80     public boolean add(String newValue) {
81         if (!contains(newValue)) {
82             // resize and rehash if needed
83             int relativeSize = relativeSize((float)(_size+1)/capacity());
84             if (relativeSize == BIG) {
85                 resizeTable(relativeSize);
86             }
87             // then add the new value
88             if (_hashTable[clamp(newValue.hashCode())] == null) {
89                 _hashTable[clamp(newValue.hashCode())] = new FacadeLinkedList();
90             }
91             _hashTable[clamp(newValue.hashCode())].add(newValue);
92             ++_size;
93             return true;
94         }
95         return false;
96     }
97
98
99     /**
100      * Resizes the current hash table
101      * @param relativeSize relative size of the current table to the demanded one
102      */
103     private void resizeTable(int relativeSize) {
104         // resize
105         FacadeLinkedList[] oldTable = _hashTable;
106         _size = 0;
107         if (relativeSize == BIG) {
108             _hashTable = new FacadeLinkedList[capacity() * 2];
109         } else if (relativeSize == SMALL && capacity() > 1) {
110             _hashTable = new FacadeLinkedList[capacity() / 2];
111         } else {
112             return;
113         }
114         // rehash
115         rehash(oldTable);
116     }
117
118     /**
119      * Rehashes the elements of the old table to the new one
120      * @param oldTable the old hash table
121      */
122     private void rehash(FacadeLinkedList[] oldTable) {
123         for (FacadeLinkedList list : oldTable) {
124             if (list != null) {
125                 Iterator<String> it = list.iterator();
126                 while (it.hasNext()) {
127                     add(it.next());

```

```

128         }
129     }
130 }
131 }
132
133 /**
134  * Look for a specified value in the set.
135  * @param searchVal Value to search for
136  * @return True iff searchVal is found in the set
137  */
138 @Override
139 public boolean contains(String searchVal) {
140     if (_hashTable[clamp(searchVal.hashCode())] != null) {
141         return _hashTable[clamp(searchVal.hashCode())].contains(searchVal);
142     }
143     return false;
144 }
145
146 /**
147  * Remove the input element from the set.
148  * @param toDelete Value to delete
149  * @return True iff toDelete is found and deleted
150  */
151 @Override
152 public boolean delete(String toDelete) {
153     if (contains(toDelete)) {
154         // delete the value
155         if (_hashTable[clamp(toDelete.hashCode())].delete(toDelete)) {
156             --_size;
157             // resize and rehash if needed
158             int relativeSize = relativeSize((float)(_size)/capacity());
159             if (relativeSize == SMALL)
160                 resizeTable(relativeSize);
161             return true;
162         }
163     }
164     return false;
165 }
166
167 }

```

7 RESULTS

```
1  #Fill in your runtime results in this file
2  #You should replace each X with the corresponding value
3
4  #These values correspond to the time it takes (in ms) to insert data1 to all data structures
5  OpenHashSet_AddData1 = 132021
6  ClosedHashSet_AddData1 = 186513
7  TreeSet_AddData1 = 85
8  LinkedList_AddData1 = 58644
9  HashSet_AddData1 = 70
10
11 #These values correspond to the time it takes (in ms) to insert data2 to all data structures
12 OpenHashSet_AddData2 = 84
13 ClosedHashSet_AddData2 = 58
14 TreeSet_AddData2 = 80
15 LinkedList_AddData2 = 42721
16 HashSet_AddData2 = 35
17
18 #These values correspond to the time it takes (in ns) to check if "hi" is contained in
19 #the data structures initialized with data1
20 OpenHashSet_Contains_hi1 = 47
21 ClosedHashSet_Contains_hi1 = 38
22 TreeSet_Contains_hi1 = 204
23 LinkedList_Contains_hi1 = 910915
24 HashSet_Contains_hi1 = 27
25
26 #These values correspond to the time it takes (in ns) to check if "-13170890158" is contained in
27 #the data structures initialized with data1
28 OpenHashSet_Contains_negative = 1019833
29 ClosedHashSet_Contains_negative = 3687211
30 TreeSet_Contains_negative = 267
31 LinkedList_Contains_negative = 955169
32 HashSet_Contains_negative = 88
33
34 #These values correspond to the time it takes (in ns) to check if "23" is contained in
35 #the data structures initialized with data2
36 OpenHashSet_Contains_23 = 86
37 ClosedHashSet_Contains_23 = 73
38 TreeSet_Contains_23 = 72
39 LinkedList_Contains_23 = 189
40 HashSet_Contains_23 = 33
41
42 #These values correspond to the time it takes (in ns) to check if "hi" is contained in
43 #the data structures initialized with data2
44 OpenHashSet_Contains_hi2 = 79
45 ClosedHashSet_Contains_hi2 = 59
46 TreeSet_Contains_hi2 = 149
47 LinkedList_Contains_hi2 = 795988
48 HashSet_Contains_hi2 = 28
49
```


8 SimpleHashSet.java

```
1  /**
2   * A superclass for implementations of hash-sets implementing the SimpleSet interface.
3   */
4  public abstract class SimpleHashSet implements SimpleSet {
5
6      // #####
7      // #### CONSTANTS ####
8      // #####
9
10     private static final int SMALL = -1;
11     private static final int GOOD = 0;
12     private static final int BIG = 1;
13
14     /** Describes the higher load factor of a newly created hash set. */
15     protected static float DEFAULT_HIGHER_CAPACITY = 0.75f;
16
17     /** Describes the lower load factor of a newly created hash set. */
18     protected static float DEFAULT_LOWER_CAPACITY = 0.25f;
19
20     /** Describes the capacity of a newly created hash set. */
21     protected static int INITIAL_CAPACITY = 16;
22
23     // #####
24     // #### ATTRIBUTES ####
25     // #####
26
27     /** The current capacity (number of cells) of the table. */
28     protected int _capacity;
29
30     /** The current size (number of occupied cells) of the table. */
31     protected int _size;
32
33     /** The current higher load factor of the hash set. */
34     private float _upperLoadFactor;
35
36     /** The current lower load factor of the hash set. */
37     private float _lowerLoadFactor;
38
39     // #####
40     // #### CONSTRUCTORS ####
41     // #####
42
43     /**
44      * Constructs a new hash set with the default capacities given in DEFAULT_LOWER_CAPACITY
45      * and DEFAULT_HIGHER_CAPACITY.
46      */
47     protected SimpleHashSet() {
48         _upperLoadFactor = DEFAULT_HIGHER_CAPACITY;
49         _lowerLoadFactor = DEFAULT_LOWER_CAPACITY;
50     }
51     //
52     _capacity = INITIAL_CAPACITY;
53     _size = 0;
54
55     /**
56      * Constructs a new hash set with capacity INITIAL_CAPACITY.
57      * @param upperLoadFactor the upper load factor before rehashing.
58      * @param lowerLoadFactor the lower load factor before rehashing
59      */
60     protected SimpleHashSet(float upperLoadFactor, float lowerLoadFactor) {
```

```

60         _upperLoadFactor = upperLoadFactor;
61         _lowerLoadFactor = lowerLoadFactor;
62         //         _capacity = INITIAL_CAPACITY;
63         _size = 0;
64     }
65
66     // #####
67     // ##### METHODS #####
68     // #####
69
70     /**
71      * @return The number of elements currently in the set.
72      */
73     public int size() {
74         return _size;
75     }
76
77     /**
78      * @return The current capacity (number of cells) of the table.
79      */
80     public abstract int capacity();
81
82     /**
83      * @return The lower load factor of the table.
84      */
85     protected float getLowerLoadFactor() {
86         return _lowerLoadFactor;
87     }
88
89     /**
90      * @return The higher load factor of the table.
91      */
92     protected float getUpperLoadFactor() {
93         return _upperLoadFactor;
94     }
95
96     /**
97      * Clamps hashing indices to fit within the current table capacity.
98      * @param index the index before clamping.
99      * @return an index properly clamped.
100     */
101     protected int clamp(int index) {
102         // index % tableSize-1
103         return index & (capacity() - 1);
104     }
105
106     /**
107      * Checks if the current load factor of the table is within the bounds
108      * @param loadFactor The load factor of the table (size/capacity)
109      * @return -1 if less, 1 if greater, 0 if within.
110     */
111     protected int relativeSize(float loadFactor) {
112         if (loadFactor < this.getLowerLoadFactor() && capacity() > 1) {
113             return SMALL;
114         }
115         if (loadFactor > this.getUpperLoadFactor()) {
116             return BIG;
117         }
118         return GOOD;
119     }
120
121     /**
122      * Add a specified element to the set if it's not already in it.
123      * @param newValue New value to add to the set
124      * @return False iff newValue already exists in the set
125     */
126     public abstract boolean add(String newValue);
127

```

```

128     /**
129      * Look for a specified value in the set.
130      * @param searchVal Value to search for
131      * @return True iff searchVal is found in the set
132      */
133     public abstract boolean contains(String searchVal);
134
135     /**
136      * Remove the input element from the set.
137      * @param toDelete Value to delete
138      * @return True iff toDelete is found and deleted
139      */
140     public abstract boolean delete(String toDelete);
141
142 }

```

9 SimpleSetPerformanceAnalyzer.java

```
1  import java.util.*;
2
3  /**
4   * Simple class analyzes the performance of these five data structures:
5   * OpenHashSet, ClosedHashSet, LinkedList, HashSet & TreeSet.
6   */
7  public class SimpleSetPerformanceAnalyzer {
8
9      private static final int DASTs_NUM = 5;
10     private static final String[] DASTsNames= new String[]{"OpenHashSet", "ClosedHashSet",
11                                                             "HashSet", "TreeSet", "LinkedList"};
12
13     private static final int NANO_TO_MILL_FACTOR = 1000000;
14     private static final int WARMUP_TIME = 70000;
15
16     private static final String[] data1 = Ex4Utils.file2array("./src/data1.txt");
17     private static final String[] data2 = Ex4Utils.file2array("./src/data2.txt");
18
19     /** Array of SimpleSet data structures, each initialized with the words in data1.txt.*/
20     private static SimpleSet[] DASTs_data1;
21     /** Array of SimpleSet data structures, each initialized with the words in data2.txt.*/
22     private static SimpleSet[] DASTs_data2;
23
24     /**
25      * Sets up the test resources
26      */
27     private static void setUp() {
28         System.out.println(" ... Preparing test sources ... \n" +
29                             " .. THIS WILL TAKE A FEW MINUTES .. ");
30         System.out.println(" Building Array of SimpleSet data structures," +
31                             " each initialized with the words in data1.txt.");
32         long timeBefore1 = System.nanoTime();
33         DASTs_data1 = DASTsFromListFactory(data1);
34         long difference1 = (System.nanoTime() - timeBefore1) / NANO_TO_MILL_FACTOR;
35         System.out.println(" Time: " + difference1 + " milliseconds.");
36
37         System.out.println(" Building Array of SimpleSet data structures," +
38                             " each initialized with the words in data2.txt.");
39         long timeBefore2 = System.nanoTime();
40         DASTs_data2 = DASTsFromListFactory(data2);
41         long difference2 = (System.nanoTime() - timeBefore2) / NANO_TO_MILL_FACTOR;
42         System.out.println(" Time: " + difference2 + " milliseconds.\n");
43     }
44
45     /**
46      * Creates a SimpleSet array and initialize it with the 5 aforementioned DASTs
47      * @return a SimpleSet array
48      */
49     private static SimpleSet[] DASTsDefaultFactory() {
50         SimpleSet[] DASTsList = new SimpleSet[DASTs_NUM];
51         DASTsList[0] = new OpenHashSet();
52         DASTsList[1] = new ClosedHashSet();
53         DASTsList[2] = new CollectionFacadeSet(new HashSet<String>());
54         DASTsList[3] = new CollectionFacadeSet(new TreeSet<String>());
55         DASTsList[4] = new CollectionFacadeSet(new LinkedList<String>());
56         return DASTsList;
57     }
58
59     /**
60      * Creates a SimpleSet array and initialize it with the 5 aforementioned DASTs
61      */
62 }
```

```

60     * @return a SimpleSet array
61     */
62     private static SimpleSet[] DASTsFromListFactory(String[] dataSet) {
63         SimpleSet[] DASTsList = new SimpleSet[DASTs_NUM];
64         DASTsList[0] = new OpenHashSet(dataSet);
65         DASTsList[1] = new ClosedHashSet(dataSet);
66         DASTsList[2] = new CollectionFacadeSet(new HashSet<String>(Arrays.asList(dataSet)));
67         DASTsList[3] = new CollectionFacadeSet(new TreeSet<String>(Arrays.asList(dataSet)));
68         DASTsList[4] = new CollectionFacadeSet(new LinkedList<String>(Arrays.asList(dataSet)));
69         return DASTsList;
70     }
71
72     /**
73      * Adding all the words in data1.txt, one by one, to each of the data structures.
74      */
75     private static void addTest1() {
76         System.out.println(" === add(String value) TEST - data1.txt === ");
77         SimpleSet[] DASTs = DASTsDefaultFactory();
78         for (int i=0; i<DASTs_NUM; ++i) {
79             System.out.printf("(%d) %s ... ", i+1, DASTsNames[i]);
80             assert data1 != null;
81             long timeBefore = System.nanoTime();
82             for (String value: data1) {
83                 DASTs[i].add(value);
84             }
85             long difference = (System.nanoTime() - timeBefore) / NANO_TO_MILL_FACTOR;
86             System.out.println("Time: " + difference + " milliseconds.");
87         }
88         System.out.println();
89     }
90
91     /**
92      * Adding all the words in data2.txt, one by one, to each of the data structures.
93      */
94     private static void addTest2() {
95         System.out.println(" === add(String value) TEST - data2.txt === ");
96         SimpleSet[] DASTs = DASTsDefaultFactory();
97         for (int i=0; i<DASTs_NUM; ++i) {
98             System.out.printf("(%d) %s ... ", i+1, DASTsNames[i]);
99             assert data2 != null;
100             long timeBefore = System.nanoTime();
101             for (String value: data2) {
102                 DASTs[i].add(value);
103             }
104             long difference = (System.nanoTime() - timeBefore) / NANO_TO_MILL_FACTOR;
105             System.out.println("Time: " + difference + " milliseconds.");
106         }
107         System.out.println();
108     }
109
110     /**
111      * Searching the string "hi" in data1.txt, for each data structure.
112      */
113     private static void containsHiTest1() {
114         System.out.println(" === contains(hi) TEST - data1.txt === ");
115         SimpleSet[] DASTs = DASTs_data1;
116         for (int i=0; i<DASTs_NUM; ++i) {
117             System.out.printf("(%d) %s ... ", i+1, DASTsNames[i]);
118             if (i<DASTs_NUM-1) {
119                 // WARM UP
120                 for (int j=0; j<WARMUP_TIME; ++j)
121                     DASTs[i].contains("hi");
122             }
123             long timeBefore = System.nanoTime();
124             for (int j=0; j<WARMUP_TIME; ++j)
125                 DASTs[i].contains("hi");
126             long difference = (System.nanoTime() - timeBefore) / WARMUP_TIME;
127             System.out.println("Time: " + difference + " nanoseconds.");

```

```

128     }
129     System.out.println();
130 }
131
132 /**
133  * Searching the string "-13170890158" in data1.txt, for each data structure.
134  */
135 private static void containsTest1() {
136     System.out.println(" === contains(-13170890158) TEST - data1.txt === ");
137     SimpleSet[] DASTs = DASTs_data1;
138     for (int i=0; i<DASTs_NUM; ++i) {
139         System.out.printf("(%d) %s", i+1, DASTsNames[i]);
140         if (i<DASTs_NUM-1) {
141             // WARM UP
142             for (int j=0; j<WARMUP_TIME; ++j)
143                 DASTs[i].contains("-13170890158");
144         }
145         long timeBefore = System.nanoTime();
146         for (int j=0; j<WARMUP_TIME; ++j)
147             DASTs[i].contains("-13170890158");
148         long difference = (System.nanoTime() - timeBefore) / WARMUP_TIME;
149         System.out.println("Time: " + difference + " nanoseconds.");
150     }
151     System.out.println();
152 }
153
154 /**
155  * Searching the string "23" in data1.txt, for each data structure.
156  */
157 private static void containsTest2() {
158     System.out.println(" === contains(23) TEST - data2.txt === ");
159     SimpleSet[] DASTs = DASTs_data2;
160     for (int i=0; i<DASTs_NUM; ++i) {
161         System.out.printf("(%d) %s", i+1, DASTsNames[i]);
162         if (i<DASTs_NUM-1) {
163             // WARM UP
164             for (int j=0; j<WARMUP_TIME; ++j)
165                 DASTs[i].contains("23");
166         }
167         long timeBefore = System.nanoTime();
168         for (int j=0; j<WARMUP_TIME; ++j)
169             DASTs[i].contains("23");
170         long difference = (System.nanoTime() - timeBefore) / WARMUP_TIME;
171         System.out.println("Time: " + difference + " nanoseconds.");
172     }
173     System.out.println();
174 }
175
176 /**
177  * Searching the string "hi" in data2.txt, for each data structure.
178  */
179 private static void containsHiTest2() {
180     System.out.println(" === contains(hi) TEST - data2.txt === ");
181     SimpleSet[] DASTs = DASTs_data2;
182     for (int i=0; i<DASTs_NUM; ++i) {
183         System.out.printf("(%d) %s", i+1, DASTsNames[i]);
184         if (i<DASTs_NUM-1) {
185             // WARM UP
186             for (int j=0; j<WARMUP_TIME; ++j)
187                 DASTs[i].contains("hi");
188         }
189         long timeBefore = System.nanoTime();
190         for (int j=0; j<WARMUP_TIME; ++j)
191             DASTs[i].contains("hi");
192         long difference = (System.nanoTime() - timeBefore) / WARMUP_TIME;
193         System.out.println("Time: " + difference + " nanoseconds.");
194     }
195     System.out.println();

```

```
196     }
197
198     public static void main(String[] args) {
199         setUp();
200         addTest1();
201         addTest2();
202         containsHiTest1();
203         containsTest1();
204         containsTest2();
205         containsHiTest2();
206     }
207
208
209 }
```