Contents

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

```
bar246802



============================
=      File description    =
============================
SimpleHashSet.java - an abstract class implementing SimpleSet.
OpenHashSet.java - a hash-set based on chaining. Extends SimpleHashSet.
ClosedHashSet.java - a hash-set based on closed-hashing with quadratic probing. Extends SimpleHashSet.
CollectionFacadeSet.java - implements SimpleSet. Wrap an object implementing java's Collection
interface, such as LinkedList, TreeSet, or HashSet.
SimpleSetPerformanceAnalyzer.java - measures the run-times for Collections' methods.



============================
=          Design          =
============================
I implemented the OpenHashSet with a nested class - a wrapper class that has a LinkedList
and delegates methods to it.
This inner class is only used inside the OpenHashSet and it is indeed relatively small,
so it seemed fit to use the nested class concept.
I extended the SimpleHashSet class to have all the common methods and consts - that way in the
OpenHashSet & ClosedHashSet classes I didn't needed to copy-past code.



============================
=  Implementation details   =
============================
In the SimpleSetPerformanceAnalyzer class I also used a nested class in order to create an
array of the collections we wish to test.
I tried as much as I could to make the test of add and contain general (which was relatively
easy thanks to CollectionFacadeSet class)
that way if one wish to turn off any test he can comment-out the relevant test in the main function
and all the others will still work.


============================
=    Answers to questions   =
============================

1)Q: Open hashing-explain your choice in the implementmention of the the open-hashing set using linked lists.
A: As I explained before I implemented the OpenHashSet with a nested class -
a wrapper class that has a LinkedList and delegates methods to it.
This inner class is only used inside the OpenHashSet and it is indeed relatively small so it seemed fit to me
to use the nested class concept.
2)Q: Closed hashing - How you implemented the deletion mechanism in ClosedHashSet -
What will happen if when deleting a value, we simply put null in its place?
A: When deleteing a value we store a "deleted flag" in its place - meaning a const string that
marks for us a cell who is now free,
after we deleted the old value. That way we can compare addresses with the current value and the
value we wish to insert/search
If the addresses are the same as the delete flag we now we can enter a value in it's place / keep
on searching until we find the value or find a real null cell.
3) Q: Discuss the results of the analysis in depth:
- Account, in separate, for OpenHashSet's and ClosedHashSet's bad results for data1.txt
```

- Summarize the strengths and weaknesses of each of the data structures as reflected by
the results. Which would you use for which purposes?
- How did your two implementations compare between themselves?
- How did your implementations compare to Java's built in HashSet?
- Did you find java's HashSet performance on data1.txt surprising? Can you explain it?
A: After examine the time analysis tests I conclude the following conclusions:
* The LinkedList come last on almost every test, it's surprising considering how popular this
collection is with developers.
* HashSet performance were topnotch - even with data1.txt, this is thanks to the
inner implementation with the HashMap.
* TreeSet come 2 or 3 on the tests, only with data2 OpenHashSet's and ClosedHashSet's
had better result's then him. It seems the
* ClosedHashSet did bad on data1.txt as expected - because each value had
the same hash code.
OpenHashSet on the other hand did considerably better on the data1.txt related tests
which also make sense due to the fact we store the elements with the same hash in lists
and don't need to look for a new index every time.
It still did bad in:
"check if "-13170890158" is contained in the data structures initialized with data1"
in this test its result was very much the same as the LinkList's results
which also make sense because we basically search here with the same length of a list
(the inner list inside the cell who contained "-13170890158")
* on add data ClosedHashSet did better than OpenHashSet
that's because we insert in OpenHashSet to the same list over and over which
takes O(n).

==============================
= Changes for resubmission =
==============================
I only changed some chars in this README file that caused it to not be converted to pdf
in the first submission. I added to the resubmission the diff btw the 2 READMEs.

3 ClosedHashSet.java

```java
/**
 * a hash-set based on closed-hashing with quadratic probing.
 * It extends SimpleHashSet.
 * @author Bar Melinarskiy
 * @version 31/8/20
 */
public class ClosedHashSet extends SimpleHashSet
{
    // constants
    static final int HASHING_CONST = 2;

    // instance variables
    private String[] hashTable;

    /*----=  Constructor  =-----*/
    /**
     * A default constructor. Constructs a new,
     * empty table with default initial capacity (16),
     * upper load factor (0.75) and lower load factor (0.25).
     */
    public ClosedHashSet()
    {
         hashTable = new String[capacity];
    }
    /**
     * Constructs a new, empty table with the specified load factors,
     * and the default initial capacity (16).
     */
    public ClosedHashSet(float upperLoadFactor, float lowerLoadFactor)
    {
        super(upperLoadFactor, lowerLoadFactor);
        hashTable = new String[capacity];
    }
    /**
     * Data constructor - builds the hash set by adding the elements one by one.
     * Duplicate values should be ignored.
     * The new table has the default values of initial capacity (16),
     * upper load factor (0.75), and lower load factor (0.25).
     * @param data Values to add to the set
     */
    public ClosedHashSet(String[] data)
    {
        this(); //call the default constructor
        //insert the given value to the table
        add(data);
    }

    /*----=  Instance Methods  =-----*/
    /**
     * Add a specified element to the set if it's not already in it.
     * @param newValue New value to add to the set
     * @return False iff newValue already exists in the set
     */
    @Override
    public boolean add(String newValue)
    {
        //Check if the given value already exist inside this table
        if(contains(newValue))
        {
```

```
60              return false;
61          }
62          //If we reached this point than this value should be added
63          increaseSize();
64          //Get the right index for the new value
65          int i = MIN_INDEX;
66          int index = hashValue(newValue, i);
67          boolean cellIsOccupied = hashTable[index] != null && hashTable[index] != DELETED_CELL;
68          while(cellIsOccupied && i < capacity())
69          {
70              i++;
71              index = hashValue(newValue, i);
72              cellIsOccupied = hashTable[index] != null && hashTable[index] != DELETED_CELL;
73          }
74          //Insert the new value if we found a right spot for it
75          if(hashTable[index] == null || hashTable[index] == DELETED_CELL)
76          {
77              hashTable[index] = newValue;
78              return true;
79          }
80          return false;
81      }
82
83      /**
84       * Look for a specified value in the set.
85       * @param searchVal Value to search for
86       * @return True iff searchVal is found in the set
87       */
88      @Override
89      public boolean contains(String searchVal)
90      {
91          return getIndexByValue(searchVal) != NON_EXISTING;
92      }
93
94      /**
95       * Remove the input element from the set.
96       * @param toDelete Value to delete
97       * @return True iff toDelete is found and deleted
98       */
99      @Override
100     public boolean delete(String toDelete)
101     {
102         int indexToDelete = getIndexByValue(toDelete);
103         //Check if given value exist inside the table
104         if(indexToDelete != NON_EXISTING)
105         {
106             //Delete it from the table by flagging the cell with the deleted value
107             hashTable[indexToDelete] = DELETED_CELL;
108             decreaseSize();
109             //Check if we need to change the table's capacity after the removal
110 //            checkCapacity();
111             return true;
112         }
113         return false;
114     }
115     /** Adjust capacity of table after insert / remove.
116      */
117     @Override
118     protected void adjustToCapacity()
119     {
120         //create new hash table after the change to the capacity and rehash all valid cells
121         String[] tmpTable = hashTable.clone();
122         hashTable = new String[capacity];
123         for (String cell : tmpTable)
124         {
125             if (cell != DELETED_CELL && cell != null) {
126                 add(cell);
127             }
```

```
128             }
129         }
130 
131         /**
132          * Get the index of the cell in which the given value is stored.
133          * @param value Value to search for
134          * @return The index of the given value if it exist, -1 otherwise.
135          */
136         private int getIndexByValue(String value)
137         {
138             int i = MIN_INDEX;
139             int index = hashValue(value, i);
140             while(hashTable[index] != null)
141             {
142                 if(hashTable[index].equals(value))
143                 {
144                     return index;
145                 }
146                 i++;
147                 index = hashValue(value, i);
148             }
149             return NON_EXISTING;
150         }
151 
152         /**
153          * Get the index of the cell in which the given value is stored.
154          * @param value Value to search for
155          * @param index number of try
156          * @return The index of the given value if it exist, -1 otherwise.
157          */
158         private int hashValue(String value, int index)
159         {
160             return clamp(hash(value) + (index + (index * index)) / HASHING_CONST);
161         }
162     }
```

4 CollectionFacadeSet.java

```java
/**
 * implements SimpleSet.
 * Wrap an object implementing java's Collection<String>.
 * @author Bar Melinarskiy
 * @version 31/8/20
 */
public class CollectionFacadeSet implements SimpleSet
{
    // instance variables
    private java.util.Collection<java.lang.String> currentCollection;

    /*-----=  Constructor  =-----*/
    public CollectionFacadeSet(java.util.Collection<java.lang.String> collection)
    {
        currentCollection = collection;
    }
    /*-----=  Instance Methods  =-----*/
    /**
     * Add a specified element to the set if it's not already in it.
     * @param newValue New value to add to the set
     * @return False iff newValue already exists in the set
     */
    @Override
    public boolean add(String newValue)
    {
        if(!currentCollection.contains(newValue))
        {
            return currentCollection.add(newValue);
        }
        return false;
    }

    /**
     * Look for a specified value in the set.
     * @param searchVal Value to search for
     * @return True if searchVal is found in the set
     */
    @Override
    public boolean contains(String searchVal)
    {
        return currentCollection.contains(searchVal);
    }

    /**
     * Remove the input element from the set.
     * @param toDelete Value to delete
     * @return True iff toDelete is found and deleted
     */
    @Override
    public boolean delete(String toDelete)
    {
        if(currentCollection.contains(toDelete))
        {
            return currentCollection.remove(toDelete);
        }
        return false;
    }

    /**
```

```
60          * @return The number of elements currently in the set
61          */
62         @Override
63         public int size()
64         {
65             return currentCollection.size();
66         }
67     }
```

5 OpenHashSet.java

```java
import java.util.LinkedList;
import java.util.ListIterator;

/**
 *  a hash-set based on chaining.
 * It extends SimpleHashSet.
 * @author Bar Melinarskiy
 * @version 31/8/20
 */
public class OpenHashSet extends SimpleHashSet
{
    // instance variables
    /** The hash table, each cell is a LinkedList of strings.
     */
    private TableCellLinkedList[] hashTable;

    /*----=  Constructor  =-----*/
    /**
     * A default constructor. Constructs a new,
     * empty table with default initial capacity (16),
     * upper load factor (0.75) and lower load factor (0.25).
     */
    public OpenHashSet()
    {
        hashTable = new TableCellLinkedList[capacity];
        initializeTable();
    }
    /**
     * Constructs a new, empty table with the specified load factors,
     * and the default initial capacity (16).
     */
    public OpenHashSet(float upperLoadFactor, float lowerLoadFactor)
    {
        super(upperLoadFactor, lowerLoadFactor);
        hashTable = new TableCellLinkedList[capacity];
        initializeTable();
    }
    /**
     * Data constructor - builds the hash set by adding the elements one by one.
     * Duplicate values should be ignored.
     * The new table has the default values of initial capacity (16),
     * upper load factor (0.75), and lower load factor (0.25).
     * @param data Values to add to the set
     */
    public OpenHashSet(String[] data)
    {
        this(); //call the default constructor
        //insert the given value to the table
        add(data);
    }

    /**
     * Add a specified element to the set if it's not already in it.
     * @param newValue New value to add to the set
     * @return False iff newValue already exists in the set
     */
    @Override
    public boolean add(String newValue)
    {
```

```
60          //Check if the given value already exist inside this table
61          if(contains(newValue))
62          {
63              return false;
64          }
65          //If we reached this point than this value should be added
66          increaseSize();
67  //          checkCapacity();
68          //Add the value to the right index
69          int index = hashValue(newValue);
70          if(hashTable[index] != null)
71          {
72              hashTable[index].add(newValue);
73              return true;
74          }
75          return false;
76      }
77
78      /**
79       * Look for a specified value in the set.
80       * @param searchVal Value to search for
81       * @return True iff searchVal is found in the set
82       */
83      @Override
84      public boolean contains(String searchVal)
85      {
86          int index = hashValue(searchVal);
87          if(hashTable[index] != null)
88          {
89              return hashTable[index].contains(searchVal);
90          }
91          return false;
92      }
93
94      /**
95       * Remove the input element from the set.
96       * @param toDelete Value to delete
97       * @return True iff toDelete is found and deleted
98       */
99      @Override
100     public boolean delete(String toDelete)
101     {
102         if(contains(toDelete))
103         {
104             int indexToDelete = hashValue(toDelete);
105             //Delete it from the table by flagging the cell with the deleted value
106             hashTable[indexToDelete].delete(toDelete);
107             decreaseSize();
108             //Check if we need to change the table's capacity after the removal
109 //              checkCapacity();
110             return true;
111         }
112
113         return false;
114     }
115
116     /**
117      * Adjust capacity of table after insert / remove.
118      */
119     @Override
120     protected void adjustToCapacity()
121     {
122         //create new hash table after the change to the capacity and rehash all valid cells
123         TableCellLinkedList[] tmpTable = hashTable.clone();
124         hashTable = new TableCellLinkedList[capacity];
125         initializeTable();
126             //rehash the table
127             for(TableCellLinkedList cell : tmpTable)
```

```
            {
                if(cell != null)
                {
                    ListIterator<String> it = cell.iterator();
                    //rehash the current list's values
                    while(it.hasNext())
                    {
                        add(it.next());
                    }
                }
            }
    }

    /**
     * initialize the hash table
     */
    private void initializeTable()
    {
        for(int i = 0; i < hashTable.length; i++)
        {
            hashTable[i] = new TableCellLinkedList();
        }
    }

    /**
     * Get the index of the cell in which the given value is stored.
     * @param value Value to search for
     * @return The index of the given value
     */
    private int hashValue(String value)
    {
        return clamp(hash(value));
    }

    /*----=  Nested Class  =-----*/
    /**
     * a wrapper class for linkedList of strings.
     * @author Bar Melinarskiy
     * @version 31/8/20
     */
    private class TableCellLinkedList
    {
        // instance variables
        private LinkedList<String> list;
        /*----=  Constructor  =-----*/
        /**
         * Default constructor.
         */
        public TableCellLinkedList()
        {
            list = new LinkedList<String>();
        }

        /**
         * Add a specified element to the list if it's not already in it.
         * @param newValue New value to add to the set
         * @return False iff newValue already exists in the list
         */
        private boolean add(String newValue) {
            return list.add(newValue);
        }

        /**
         * Look for a specified value in the list.
         * @param searchVal Value to search for
         * @return True iff searchVal is found in the list
         */
        private boolean contains(String searchVal) {
```

```
196             return list.contains(searchVal);
197         }
198 
199         /**
200          * Remove the input element from the list.
201          * @param toDelete Value to delete
202          * @return True iff toDelete is found and deleted
203          */
204         private boolean delete(String toDelete) {
205             return list.remove(toDelete);
206         }
207 
208         /**
209          * Returns an iterator over the elements in this list (in proper
210          * sequence).<p>
211          *
212          * This implementation merely returns a list iterator over the list.
213          *
214          * @return an iterator over the elements in this list (in proper sequence)
215          */
216         private ListIterator<String> iterator()
217         {
218             return list.listIterator();
219         }
220     }
221 }
```

6 RESULTS

```
#Fill in your runtime results in this file
#You should replace each X with the corresponding value

#These values correspond to the time it takes (in ms) to insert data1 to all data structures
OpenHashSet_AddData1 = 89483
ClosedHashSet_AddData1 = 268611
TreeSet_AddData1 = 73
LinkedList_AddData1 = 39538
HashSet_AddData1 = 46

#These values correspond to the time it takes (in ms) to insert data2 to all data structures
OpenHashSet_AddData2 = 108
ClosedHashSet_AddData2 = 38
TreeSet_AddData2 = 60
LinkedList_AddData2 = 20109
HashSet_AddData2 = 9

#These values correspond to the time it takes (in ns) to check if "hi" is contained in
#the data structures initialized with data1
OpenHashSet_Contains_hi1 = 22
ClosedHashSet_Contains_hi1 = 21
TreeSet_Contains_hi1 = 82
LinkedList_Contains_hi1 = 603367
HashSet_Contains_hi1 = 19

#These values correspond to the time it takes (in ns) to check if "-13170890158" is contained in
#the data structures initialized with data1
OpenHashSet_Contains_negative = 663219
ClosedHashSet_Contains_negative = 2150305
TreeSet_Contains_negative = 186
LinkedList_Contains_negative = 686059
HashSet_Contains_negative = 34

#These values correspond to the time it takes (in ns) to check if "23" is contained in
#the data structures initialized with data2
OpenHashSet_Contains_23 = 27
ClosedHashSet_Contains_23 = 21
TreeSet_Contains_23 = 55
LinkedList_Contains_23 = 189
HashSet_Contains_23 = 15

#These values correspond to the time it takes (in ns) to check if "hi" is contained in
#the data structures initialized with data2
OpenHashSet_Contains_hi2 = 9
ClosedHashSet_Contains_hi2 = 44
TreeSet_Contains_hi2 = 91
LinkedList_Contains_hi2 = 459004
HashSet_Contains_hi2 = 10

```

7 SimpleHashSet.java

```java
import java.util.Arrays;

/**
 * an abstract class implementing SimpleSet
 * It extends the abstract class SpaceShip
 * @author Bar Melinarskiy
 * @version 31/8/20
 */
public abstract class SimpleHashSet implements SimpleSet
{
    // constants
    /** Describes the higher load factor of a newly created hash set.
     */
    protected final static float DEFAULT_HIGHER_CAPACITY = 0.75f;
    /** Describes the lower load factor of a newly created hash set.
     */
    protected final static float DEFAULT_LOWER_CAPACITY = 0.25f;
    /** The first valid index of the table.
     */
    protected final static int MIN_INDEX = 0;
    /** Flag for non-existing value
     */
    protected final static int NON_EXISTING = -1;
    /** The const we use to keep the table size a power of 2.
     */
    protected final static int CAPACITY_ADJUST_CONST = 2;
    /** Describes the capacity of a newly created hash set.
     */
    protected final static int MIN_CAPACITY = 1;
    /** Flag for a cell in the table who has been removed
     */
    protected final static String DELETED_CELL = new String();
    /** Describes the capacity of a newly created hash set.
     */
    protected final static int INITIAL_CAPACITY = 16;
    /** Describes the size of a newly created hash set.
     */
    protected final static int INITIAL_SIZE = 0;
    /** Describes the size of a newly created hash set.
     */
    protected final static int INITIAL_SIZE_AFTER_ADJUST = 1;
    // instance variables
    /** Describes the current capacity (number of cells) of the table.
     */
    protected int capacity = INITIAL_CAPACITY;
    /** Describes the number of elements currently in the set.
     */
    protected int size = INITIAL_SIZE;
    /** Describes the higher load factor of the table.
     */
    protected float upperLoadFactor = DEFAULT_HIGHER_CAPACITY;
    /** Describes the lower load factor of the table.
     */
    protected float lowerLoadFactor = DEFAULT_LOWER_CAPACITY;

    /*----=  Constructor  =-----*/

    /** Constructs a new hash set with the default capacities given
     *  in DEFAULT_LOWER_CAPACITY and DEFAULT_HIGHER_CAPACITY.
```

```
60          */
61         protected SimpleHashSet()
62         {
63             setCapacity(INITIAL_CAPACITY);
64             setLowerLoadFactor(DEFAULT_LOWER_CAPACITY);
65             setUpperLoadFactor(DEFAULT_HIGHER_CAPACITY);
66         }
67 
68         /** Constructs a new hash set with capacity INITIAL_CAPACITY.
69          * @param upperLoadFactor The upper load factor of the hash table.
70          * @param lowerLoadFactor The lower load factor of the hash table.
71          */
72         protected SimpleHashSet(float upperLoadFactor, float lowerLoadFactor)
73         {
74             setCapacity(INITIAL_CAPACITY);
75             setLowerLoadFactor(lowerLoadFactor);
76             setUpperLoadFactor(upperLoadFactor);
77         }
78 
79         /*----=  Abstract Methods  =-----*/
80         /** Adjust capacity of table after insert / remove.
81          */
82         protected abstract void adjustToCapacity();
83 
84         /*----=  Instance Methods  =-----*/
85         /** Get the number of elements currently in the set
86          * @return The number of elements currently in the set
87          */
88         public int size()
89         {
90             return size;
91         }
92         /** Get the current capacity (number of cells) of the table.
93          * @return The current capacity (number of cells) of the table.
94          */
95         int    capacity()
96         {
97             return capacity;
98         }
99         /** Get the lower load factor of the table.
100          * @return the lower load factor of the table.
101          */
102         protected float    getLowerLoadFactor()
103         {
104             return lowerLoadFactor;
105         }
106         /** get the higher load factor of the table.
107          * @return the higher load factor of the table.
108          */
109         protected float    getUpperLoadFactor()
110         {
111             return upperLoadFactor;
112         }
113         /** Add to table from the given values.
114          * Duplicate values should be ignored.
115          * @param data Values to add to the set
116          */
117         protected void add(String[] data)
118         {
119             //get only unique values from the given table
120             String[] uniqueData = Arrays.stream(data).distinct().toArray(String[]::new);
121             for(String cell : uniqueData)
122             {
123                 if(cell != DELETED_CELL && cell != null)
124                 {
125                     add(cell);
126                 }
127             }
```

```
128     }
129     /** Calc the current load factor of the table.
130      * @return the current load factor of the table.
131      */
132     protected float    getLoadFactor()
133     {
134         return size / (float)capacity;
135     }
136     /** check capacity of table after insert / remove.
137      * If needed then calling adjustCapacity to fix the table.
138      * @param afterInsert true if the method was called after insert,
139      * false if after delete
140      */
141     protected void checkCapacity(boolean afterInsert)
142     {
143         int newCapacity = capacity;
144         float currentLoadFactor = getLoadFactor();
145         if(afterInsert && currentLoadFactor > getUpperLoadFactor())
146         {
147             newCapacity = capacity * CAPACITY_ADJUST_CONST;
148             size = INITIAL_SIZE_AFTER_ADJUST;
149 
150         }
151         else if(!afterInsert && currentLoadFactor < getLowerLoadFactor())
152         {
153             newCapacity = Math.max(MIN_CAPACITY, (capacity / CAPACITY_ADJUST_CONST));
154             size = INITIAL_SIZE;
155         }
156 
157         if(capacity != newCapacity)
158         {
159             setCapacity(newCapacity);
160             adjustToCapacity();
161         }
162     }
163     /** set the current capacity (number of cells) of the table.
164      * @param newCapacity - The new capacity (number of cells) of the table.
165      */
166     protected void setCapacity(int newCapacity)
167     {
168         capacity = newCapacity;
169     }
170     /** Set the lower load factor of the table.
171      * @param newFactor the new lower load factor of the table.
172      */
173     protected void setLowerLoadFactor(float newFactor)
174     {
175         lowerLoadFactor = newFactor;
176     }
177     /** Set the higher load factor of the table.
178      * @param newFactor the new higher load factor of the table.
179      */
180     protected void setUpperLoadFactor(float newFactor)
181     {
182         upperLoadFactor = newFactor;
183     }
184     /**
185      * Clamps hashing indices to fit within the current table capacity.
186      * @param index the index before clamping.
187      * @return an index properly clamped.
188      */
189     protected int clamp(int index)
190     {
191         return index & (capacity() - 1);
192     }
193     /**
194      * Computes key.hashCode()
195      * @param value value to calc hash for
```

```
196         * @return the hase code of the given value
197         */
198        protected int hash(String value)
199        {
200            return (value == null) ? 0 : value.hashCode() ;
201        }
202        /** Increase the table size by 1.
203         */
204        protected void increaseSize()
205        {
206            size++;
207            checkCapacity(true);
208        }
209        /** Decrease the table size by 1.
210         */
211        protected void decreaseSize()
212        {
213            size--;
214            checkCapacity(false);
215        }
216    }
217
```

8 SimpleSetPerformanceAnalyzer.java

```java
import java.util.HashSet;
import java.util.LinkedList;
import java.util.TreeSet;
import java.text.MessageFormat;

/**
 * measures the run-times for Collections' methods.
 * @author Bar Melinarskiy
 * @version 31/8/20
 */
public class SimpleSetPerformanceAnalyzer
{
    // constants
    private static final int  CAST_TO_MILLISECONDS = 1000000;
    private static final int  NUM_OF_ITERATIONS = 70000;
    private static final int  NUM_OF_ITERATIONS_LINKED_LIST = 7000;
    private static final int  NUM_OF_COLLECTIONS = 5;
    private static final int  INIT_SIZE = 0;
    private static final int  OPEN_HASH_SET = 0;
    private static final int  CLOSED_HASH_SET = 1;
    private static final int  TREE_SET = 2;
    private static final int  LINKED_LIST = 3;
    private static final int  HASHSET = 4;
    private static final String  OPEN_HASH_SET_S = "OpenHashSet";
    private static final String  CLOSED_HASH_SET_S = "ClosedHashSet";
    private static final String  TREE_SET_S = "TreeSet";
    private static final String  LINKED_LIST_S = "LinkedList";
    private static final String  HASHSET_S = "HashSet";

    /*----=  Nested Class  =-----*/
    private static class TestSet
    {
        // instance variables
        SimpleSet setObject;
        String name;
        int numOfIterations = NUM_OF_ITERATIONS;
    }

    /*----=  Test methods  =-----*/
    /**
     * Create an array of all the sets we wish to test.
     * @param collections the array of sets we wish to populate
     */
    private static void initializeCollections(TestSet[] collections)
    {
        for(int i = 0 ; i < collections.length; i++)
        {
            collections[i] = new TestSet();

            switch (i)
            {
            case OPEN_HASH_SET:
                collections[i].setObject = new OpenHashSet();
                collections[i].name = OPEN_HASH_SET_S;
                break;
            case CLOSED_HASH_SET:
                collections[i].setObject = new ClosedHashSet();
                collections[i].name = CLOSED_HASH_SET_S;
                break;
```

```
60              case TREE_SET:
61                  collections[i].setObject = new CollectionFacadeSet(new TreeSet<String>());
62                  collections[i].name = TREE_SET_S;
63                  break;
64              case LINKED_LIST:
65                  collections[i].setObject = new CollectionFacadeSet(new LinkedList<String>());
66                  collections[i].name = LINKED_LIST_S;
67                  collections[i].numOfIterations = NUM_OF_ITERATIONS_LINKED_LIST;
68                  break;
69              case HASHSET:
70                  collections[i].setObject = new CollectionFacadeSet(new HashSet<String>());
71                  collections[i].name = HASHSET_S;
72                  break;
73              default:
74                  break;
75              }
76          }
77      }
78
79      /**
80       * Test adding time to the set.
81       * @param collections the array of sets we wish to populate
82       * @param fileName the values source file name
83       */
84      private static void testAdding(TestSet[] collections, String fileName)
85      {
86          initializeCollections(collections);
87          String[] data = Ex4Utils.file2array(fileName);
88          if(data != null)
89          {
90              final String msgFormat = "For collection {0} It took {1} milliseconds" +
91                                       " to add words from file {2}";
92              for (TestSet collection : collections)
93              {
94                  long timeBefore = System.nanoTime();
95
96                  for(String value: data)
97                  {
98                      if(value != null)
99                      {
100                         collection.setObject.add(value);
101                     }
102                 }
103
104                 long difference = (System.nanoTime() - timeBefore) / CAST_TO_MILLISECONDS;
105                 System.out.println(MessageFormat.format(msgFormat, collection.name, difference, fileName));
106             }
107         }
108     }
109
110     /**
111      * Test contains method run time to the set.
112      * @param collections the array of sets we wish to test
113      * @param searchVal Value to search for
114      * @param fileName the values source file name
115      */
116     private static void testContains(TestSet[] collections, String searchVal, String fileName)
117     {
118         final String msgFormat = "For collection {0} It took {1} nanoseconds " +
119                                  "to perform contains(\"{2}\")" +
120                                  "when it's initialized with data from file {3}";
121
122         for (TestSet collection : collections)
123         {
124             //allow this method to be called even if we cancelled testAdding
125             if(collection == null || collection.setObject.size() == INIT_SIZE)
126             {
127                 testAdding(collections, fileName);
```

```
128             }
129
130             if(collection != null && collection.setObject.size() != INIT_SIZE)
131             {
132                 //perform contains without measuring time to warmup the JVM
133                 performContains(collection, searchVal);
134                 long timeBefore = System.nanoTime();
135                 //perform contains with measuring time
136                 performContains(collection, searchVal);
137                 long difference = ((System.nanoTime() - timeBefore) / collection.numOfIterations);
138                 String msg = MessageFormat.format(msgFormat, collection.name,
139                                                   difference, searchVal, fileName);
140                 System.out.println(msg);
141             }
142         }
143     }
144
145     public static void performContains(TestSet collection, String searchVal)
146     {
147         if(collection != null)
148         {
149             for(int i = 0; i < collection.numOfIterations; i++)
150             {
151                 collection.setObject.contains(searchVal);
152             }
153         }
154     }
155
156     public static void main(String[] args)
157     {
158         final String file1 = "src/data1.txt";
159         final String file2 = "src/data2.txt";
160         TestSet[] collections = new TestSet[NUM_OF_COLLECTIONS];
161         //Adding all the words in data1.txt, one by one, to each of the data structures
162         testAdding(collections, file1);
163         //For each data structure, perform contains("hi") when it's initialized with data1.txt
164         testContains(collections, "hi", file1);
165         //For each data structure, perform contains("-13170890158") with data1.txt
166         testContains(collections, "-13170890158", file1);
167         //Adding all the words in data2.txt, one by one, to each of the data structures
168         testAdding(collections, file2);
169         //For each data structure, perform contains("23") when it's initialized with data2.txt
170         testContains(collections, "23", file2);
171         //For each data structure, perform contains("hi") when it's initialized with data2.txt
172         testContains(collections, "hi", file2);
173     }
174 }
```