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

```
=====================
===== EX4 TESTER =====
=====================

===== CHECKING JAR & FILES =====

===== ANALYZE README =====

===== COMPILE CODE =====

Code complied successfully
===== RUN TESTS =====

tests output :

OpenHashSet
===========
Perfect!

ClosedHashSet
=============
Perfect!


********************
Testing performance analysis results
********************
performance analysis results tests passed
```

2 README

```
muaz.abdeen



=============================
=      File description     =
=============================
- SimpleHashSet -
    A superclass for implementations of hash-sets implementing the SimpleSet interface.
- OpenHashSet -
    A hash-set based on open hashing (chaining). Extends SimpleHashSet.
- ClosedHashSet -
    A hash-set based on closed-hashing with quadratic probing. Extends SimpleHashSet.
- CollectionFacadeSet -
    Wraps an underlying Collection and serves to both simplify its API and give it
    a common type with the implemented SimpleHashSets.
- FacadeLinkedList -
    Wraps a java LinkedList to use in OpenHashSet.
- SimpleSetPerformanceAnalyzer -
    Simple class analyzes the performance of these five data structures:
    OpenHashSet, ClosedHashSet, LinkedList, HashSet & TreeSet.



=============================
=          Design           =
=============================

Most of the design is enforced by the given API.

In the abstract super class SimpleHashSet that implements the SimpleSet interface,
I implemented some methods to avoid code duplication, and made the remaining abstract.
I tried to keep the API minimal, I added two protected methods in the super class,
other additional helper methods, in super or sub-classes, are private.

There is two Facade classes, the first is given, which is, CollectionFacadeSet, I wrapped
the main functionalities of JAVA Collection class, which implements the methods in the
SimpleSet interface.

The second Facade class, FacadeLinkedList, wraps a java LinkedList to use in making the
array of linked lists in OpenHashSet class.

Finally, in the SimpleSetPerformanceAnalyzer class, I designed it in such a way, one could
enable and disable any test easily.


=============================
=  Implementation details   =
=============================

- OpenHashSet -
    I implemented a Facade class that wraps a LinkedList, called FacadeLinkedList.
    I declared an array of this FacadeLinkedList to be the hashtable, then initialized
    it with INITIAL_CAPACITY empty (null) cells.
    When adding an item, if it is mapped to empty cell, a new FacadeLinkedList is initialized
    with this item as its head node, else it is added at the first of the existing FacadeLinkedList.
    The size attribute is updated after each deletion or addition.
    I used the methods of the LinkedList to find, delete, and add to the linked lists in the array.
```

```
- ClosedHashSet -
    The most important issue here was: how to mange the deletion operation in such a way that
    puts no constraints on string values, and takes a constant time and space?
    I created a unique string called DUMMY by invoking the String constructor directly, and
    when deleting a value I put in its place a pointer to this DUMMY, and when searching for a
    value I compared between the addresses of the strings, not the contents of them, that is to
    say, I used the '=' operator, not equals() method. In this way I can add a string equals to
    'DUMMY' to the table, without causing any problem, because it will pointed to another
    memory address different from our unique DUMMY value.

- SimpleSetPerformanceAnalyzer -
    I defined two factory functions, one that creates a SimpleSet array with OpenHashSet
    and ClosedHashSet initialized with default constructor, and other data structures which
    wrapped in CollectionFacadeSet with there default constructor, too. In other words, each
    data structure here is empty. The second factory function initialize each data structure
    with a given array of strings.
    I created a separate test function for every functionality to test as stated in the exercise
    file, so that one could enable and disable any test easily, by comment or uncomment the
    function call in the main function.


============================
=    Answers to questions   =
============================
Q - Account, in separate, for OpenHashSet's and ClosedHashSet's bad results for data1.txt -
A - In data1.txt all the Strings have the same hashCode.
    in OpenHashSet (chaining), all of the strings will end in the same bucket, so every time we
    add an item, we first check if this list contains it, which means iterating over it, this
    takes a linear time, i.e O(n) for n = number of strings in data.txt.
    in ClosedHashSet, all of the string are mapped initially to the same bucket, which increases
    the secondary clustering, and creates a long run of filled slots away from the initial
    position, in other words, we must iterate along the probe sequence every time, this takes
    also linear time.

Q - Summarize the strengths and weaknesses of each of the data structures as reflected by the
    results. Which would you use for which purposes? -
A - (1) OpenHashSet and ClosedHashSet: behaves almost the same, one can notice that OpenHashSet
        behaves better than ClosedHashSet in the worst case, both takes linear time, but it
        seems that the coefficient of n is less in OpenHashSet.
        In the average and best case they behaves almost like the Java HashSet & TreeSet, there
        is difference of course, but not huge.
    (2) Java HashSet: achieves the best results in all cases, constant time.
    (3) Java TreeSet: almost the same as HashSet, but takes a bit longer, because it orders
        the elements in every bucket, this takes more time.
    (4) Java LinkedList: takes the longest time among all other data structures, because in every
        addition addition ot deletion we check the existence of the element, which needs to
        iterate over the list.
    In general, Java HashSet behaves the best, and I would use it in the cases I wanted an
    unordered set, otherwise, TreeSet would be my choice.

Q - How did your two implementations compare between themselves? -
A - behaves almost the same, one can notice that OpenHashSet behaves better than
    ClosedHashSet in the worst case, both takes linear time, but it seems that
    the coefficient of n is less in OpenHashSet.
    In the average and best case they behaves almost the same.

Q - How did your implementations compare to Java's built in HashSet? -
A - In the worst case, there is a huge difference, mine took linear time, while Java's
    took constant time.
    In the average and best case they behaved almost like the Java HashSet, there
    is difference of course, but not huge.

Q - Did you find java's HashSet performance on data1.txt surprising? Can you explain it?
A - Yes, I did. I think they used perfect hashing, or a long hash table !!
```

3 ClosedHashSet.java

```java

/**
 * A hash-set based on closed-hashing with quadratic probing
 */
public class ClosedHashSet extends SimpleHashSet {

    // ################## //
    // #### CONSTANTS #### //
    // ################## //

    private static final int SMALL = -1;
    private static final int BIG = 1;
    private static final int NOT_FOUND = -1;

    /* create DUMMY string by invoking the String constructor explicitly
     * DUMMY now points to distinguish memory address */
    private static final String DUMMY = new String("DUMMY");

    // ################### //
    // #### ATTRIBUTES #### //
    // ################### //

    /** A standard Java array represents the hash table */
    private String[] _hashTable;

    // ##################### //
    // #### CONSTRUCTORS #### //
    // ##################### //

    /**
     * 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() {
        super();
        _hashTable = new String[INITIAL_CAPACITY];
    }

    /**
     * Constructs a new, empty table with the specified load factors,
     * and the default initial capacity (16).
     * @param upperLoadFactor The upper load factor of the hash table.
     * @param lowerLoadFactor The lower load factor of the hash table.
     */
    public ClosedHashSet(float upperLoadFactor, float lowerLoadFactor) {
        super(upperLoadFactor, lowerLoadFactor);
        _hashTable = new String[INITIAL_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) {
        super();
```

```java
        _hashTable = new String[INITIAL_CAPACITY];
        for (String key: data) {
            add(key);
        }
    }

    // ################# //
    // #### METHODS #### //
    // ################# //
    /**
     * @return The current capacity (number of cells) of the table.
     */
    @Override
    public int capacity() {
        return _hashTable.length;
    }

    /**
     * A quadratic probing function for collision resolution.
     * @param hashCode Original hash code before probing.
     * @param attempt Number of probing attempt.
     * @return the hash code after probing
     */
    private int probe(int hashCode, int attempt) {
        return hashCode + (attempt + attempt*attempt)/2;
    }

    /**
     * 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 (newValue != null && !contains(newValue)) {
            map(newValue);
            return true;
        }
        return false;
    }

    /**
     * Maps the newValue to a certain cell in the hash table
     * @param newValue the value to be mapped
     */
    private void map(String newValue) {
        // resize and rehash if needed
        int relativeSize = relativeSize((float)(_size+1)/capacity());
        if (relativeSize == BIG) {
            resizeTable(relativeSize);
        }
        // then add the new value
        /*
         * The way in which we choose c1, c2 ensures that as long as the table
         * is not full, a place for a new value will be found during the first
         * capacity attempts.  (From exercise description)
         */
        int attempt = 0;
        while (attempt < capacity()) {
            int idx = clamp(probe(newValue.hashCode(), attempt));
            // reference comparison (address comparison) of DUMMY
            if (_hashTable[idx] == null || _hashTable[idx] == DUMMY) {
                _hashTable[idx] = newValue;
                ++_size;
                return;
            }
            ++attempt;
        }
```

```
    }

    /**
     * Resizes the current hash table
     * @param relativeSize relative size of the current table to the demanded one
     */
    private void resizeTable(int relativeSize) {
        // resize
        String[] oldTable = _hashTable;
        _size = 0;
        if (relativeSize == BIG) {
            _hashTable = new String[capacity() * 2];
        } else if (relativeSize == SMALL && capacity() > 1) {
            _hashTable = new String[capacity() / 2];
        } else {
            return;
        }
        // rehash
        for (String element: oldTable) {
            if (element != null && element != DUMMY)
                map(element);
        }
    }

    /**
     * Look for a specified value in the table.
     * @param searchVal Value to search for
     * @return True iff searchVal is found in the set
     */
    @Override
    public boolean contains(String searchVal) {
        return find(searchVal) != NOT_FOUND;
    }

    /**
     * Look for a specified value in the table.
     * @param value Value to search for
     * @return the index if found, else NOT_FOUND(= -1)
     */
    private int find(String value) {
        // Quadratic probing generates at most 'capacity' probing sequences
        int attempt = 0;
        int idx = clamp(probe(value.hashCode(), attempt));
        while (_hashTable[idx] != null && attempt <= capacity()) {
            if (_hashTable[idx].equals(value) && _hashTable[idx] != DUMMY) {
                return idx;
            }
            ++attempt;
            idx = clamp(probe(value.hashCode(), attempt));
        }
        return NOT_FOUND;
    }

    /**
     * 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) {
        int idx = find(toDelete);
        if (idx != NOT_FOUND) {
            // remove the value
            _hashTable[idx] = DUMMY;
            --_size;
            // resize and rehash if needed
            int relativeSize = relativeSize((float)(_size)/capacity());
            if (relativeSize == SMALL)
```

```
                resizeTable(relativeSize);
            return true;
        }
        return false;
    }

}
```

4 CollectionFacadeSet.java

```java
import java.util.Collection;

/**
 * Wraps an underlying Collection and serves to both simplify its API
 * and give it a common type with the implemented SimpleHashSets.
 */
public class CollectionFacadeSet implements SimpleSet {

    // #################### //
    // #### ATTRIBUTES #### //
    // #################### //

    protected Collection<String> _collection;

    // ##################### //
    // #### CONSTRUCTOR #### //
    // ##################### //

    /**
     * Creates a new facade wrapping the specified collection.
     * @param collection The Collection to wrap.
     */
    public CollectionFacadeSet(Collection<String> collection) {
        _collection = collection;
    }

    // ################# //
    // #### 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 (!_collection.contains(newValue)) {
            return _collection.add(newValue);
        }
        return false;
    }

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

    /**
     * Remove the input element from the set.
     * @param toDelete Value to delete
     * @return True iff toDelete is found and deleted
     */
    @Override
```

```java
    public boolean delete(String toDelete) {
        return _collection.remove(toDelete);
    }

    /**
     * @return The number of elements currently in the set
     */
    @Override
    public int size() {
        return _collection.size();
    }
}
```

5 FacadeLinkedList.java

```java
import java.util.Iterator;
import java.util.LinkedList;

/**
 * Wraps a java LinkedList to use in OpenHashSet.
 */
public class FacadeLinkedList {

    // #################### //
    // #### ATTRIBUTES #### //
    // #################### //

    /** The wrapped linked list */
    private LinkedList<String> _linkedList;

    // ##################### //
    // #### CONSTRUCTOR #### //
    // ##################### //

    /**
     * Creates a new facade wrapping a Java LinkedList.
     */
    public FacadeLinkedList() {
        _linkedList = new LinkedList<String>();
    }

    // ################# //
    // #### METHODS #### //
    // ################# //

    /**
     * @return Returns an iterator over the elements in this list
     */
    public Iterator<String> iterator() {
        return _linkedList.iterator();
    }

    /**
     * Insert a new node at the head of the list.
     * @param newValue New value to add to the list.
     */
    public void add(String newValue) {
        // no need to check if contains because we did that in hash set add method
        _linkedList.addFirst(newValue);
        assert _linkedList.peek() != null && _linkedList.peek().equals(newValue);
    }

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

    /**
     * Removes the first occurrence of the specified element from this list, if it is present.
     * @return true if this list contained the specified element.
```

```java
     */
    public boolean delete(String toDelete) {
        return _linkedList.remove(toDelete);
    }

    /**
     * @return The number of elements currently in the list
     */
    public int size() {
        return _linkedList.size();
    }

    // End of FacadeLinkedList class
}
```

6 OpenHashSet.java

```java
import java.util.Iterator;

/**
 * A hash-set based on open hashing (chaining)
 */
public class OpenHashSet extends SimpleHashSet {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    private static final int SMALL = -1;
    private static final int BIG = 1;

    // #################### //
    // #### ATTRIBUTES #### //
    // #################### //

    /** The hash table that contains a linked list in every cell */
    private FacadeLinkedList[] _hashTable;

    // ###################### //
    // #### CONSTRUCTORS #### //
    // ###################### //

    /**
     * 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() {
        super();
        _hashTable = new FacadeLinkedList[INITIAL_CAPACITY];
    }

    /**
     * Constructs a new, empty table with the specified load factors,
     * and the default initial capacity (16).
     * @param upperLoadFactor The upper load factor of the hash table.
     * @param lowerLoadFactor The lower load factor of the hash table.
     */
    public OpenHashSet(float upperLoadFactor, float lowerLoadFactor) {
        super(upperLoadFactor, lowerLoadFactor);
        _hashTable = new FacadeLinkedList[INITIAL_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 OpenHashSet(String[] data) {
        super();
        _hashTable = new FacadeLinkedList[INITIAL_CAPACITY];
        for (String key: data) {
            add(key);
        }
```

```java
    }

    // ################# //
    // #### METHODS #### //
    // ################# //

    /**
     * @return The current capacity (number of cells) of the table.
     */
    @Override
    public int capacity() {
        return _hashTable.length;
    }

    /**
     * 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 (!contains(newValue)) {
            // resize and rehash if needed
            int relativeSize = relativeSize((float)(_size+1)/capacity());
            if (relativeSize == BIG) {
                resizeTable(relativeSize);
            }
            // then add the new value
            if (_hashTable[clamp(newValue.hashCode())] == null) {
                _hashTable[clamp(newValue.hashCode())] = new FacadeLinkedList();
            }
            _hashTable[clamp(newValue.hashCode())].add(newValue);
            ++_size;
            return true;
        }
        return false;
    }


    /**
     * Resizes the current hash table
     * @param relativeSize relative size of the current table to the demanded one
     */
    private void resizeTable(int relativeSize) {
        // resize
        FacadeLinkedList[] oldTable = _hashTable;
        _size = 0;
        if (relativeSize == BIG) {
            _hashTable = new FacadeLinkedList[capacity() * 2];
        } else if (relativeSize == SMALL && capacity() > 1) {
            _hashTable = new FacadeLinkedList[capacity() / 2];
        } else {
            return;
        }
        // rehash
        rehash(oldTable);
    }

    /**
     * Rehashes the elements of the old table to the new one
     * @param oldTable the old hash table
     */
    private void rehash(FacadeLinkedList[] oldTable) {
        for (FacadeLinkedList list : oldTable) {
            if (list != null) {
                Iterator<String> it = list.iterator();
                while (it.hasNext()) {
                    add(it.next());
```

```
                }
            }
        }
    }

    /**
     * Look for a specified value in the set.
     * @param searchVal Value to search for
     * @return True iff searchVal is found in the set
     */
    @Override
    public boolean contains(String searchVal) {
        if (_hashTable[clamp(searchVal.hashCode())] != null) {
            return _hashTable[clamp(searchVal.hashCode())].contains(searchVal);
        }
        return false;
    }

    /**
     * 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 (contains(toDelete)) {
            // delete the value
            if (_hashTable[clamp(toDelete.hashCode())].delete(toDelete)) {
                --_size;
                // resize and rehash if needed
                int relativeSize = relativeSize((float)(_size)/capacity());;
                if (relativeSize == SMALL)
                    resizeTable(relativeSize);
                return true;
            }
        }
        return false;
    }

}
```

7 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 = 132021
ClosedHashSet_AddData1 = 186513
TreeSet_AddData1 = 85
LinkedList_AddData1 = 58644
HashSet_AddData1 = 70

#These values correspond to the time it takes (in ms) to insert data2 to all data structures
OpenHashSet_AddData2 = 84
ClosedHashSet_AddData2 = 58
TreeSet_AddData2 = 80
LinkedList_AddData2 = 42721
HashSet_AddData2 = 35

#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 = 47
ClosedHashSet_Contains_hi1 = 38
TreeSet_Contains_hi1 = 204
LinkedList_Contains_hi1 = 910915
HashSet_Contains_hi1 = 27

#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 = 1019833
ClosedHashSet_Contains_negative = 3687211
TreeSet_Contains_negative = 267
LinkedList_Contains_negative = 955169
HashSet_Contains_negative = 88

#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 = 86
ClosedHashSet_Contains_23 = 73
TreeSet_Contains_23 = 72
LinkedList_Contains_23 = 189
HashSet_Contains_23 = 33

#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 = 79
ClosedHashSet_Contains_hi2 = 59
TreeSet_Contains_hi2 = 149
LinkedList_Contains_hi2 = 795988
HashSet_Contains_hi2 = 28
```

8 SimpleHashSet.java

```java
/**
 * A superclass for implementations of hash-sets implementing the SimpleSet interface.
 */
public abstract class SimpleHashSet implements SimpleSet {

    // ################### //
    // #### CONSTANTS #### //
    // ################### //

    private static final int SMALL = -1;
    private static final int GOOD = 0;
    private static final int BIG = 1;

    /** Describes the higher load factor of a newly created hash set. */
    protected static float DEFAULT_HIGHER_CAPACITY = 0.75f;

    /** Describes the lower load factor of a newly created hash set. */
    protected static float DEFAULT_LOWER_CAPACITY = 0.25f;

    /** Describes the capacity of a newly created hash set. */
    protected static int INITIAL_CAPACITY = 16;

    // #################### //
    // #### ATTRIBUTES #### //
    // #################### //

//    /** The current capacity (number of cells) of the table. */
//    protected int _capacity;

    /** The current size (number of occupied cells) of the table. */
    protected int _size;

    /** The current higher load factor of the hash set. */
    private float _upperLoadFactor;

    /** The current lower load factor of the hash set. */
    private float _lowerLoadFactor;

    // ###################### //
    // #### CONSTRUCTORS #### //
    // ###################### //

    /**
     * Constructs a new hash set with the default capacities given in DEFAULT_LOWER_CAPACITY
     * and DEFAULT_HIGHER_CAPACITY.
     */
    protected SimpleHashSet() {
        _upperLoadFactor = DEFAULT_HIGHER_CAPACITY;
        _lowerLoadFactor = DEFAULT_LOWER_CAPACITY;
//        _capacity = INITIAL_CAPACITY;
        _size = 0;
    }

    /**
     * Constructs a new hash set with capacity INITIAL_CAPACITY.
     * @param upperLoadFactor the upper load factor before rehashing.
     * @param lowerLoadFactor the lower load factor before rehashing
     */
    protected SimpleHashSet(float upperLoadFactor, float lowerLoadFactor) {
```

```java
        _upperLoadFactor = upperLoadFactor;
        _lowerLoadFactor = lowerLoadFactor;
//        _capacity = INITIAL_CAPACITY;
        _size = 0;
    }

    // ################# //
    // #### METHODS #### //
    // ################# //

    /**
     * @return The number of elements currently in the set.
     */
    public int size() {
        return _size;
    }

    /**
     * @return The current capacity (number of cells) of the table.
     */
    public abstract int capacity();

    /**
     * @return The lower load factor of the table.
     */
    protected float getLowerLoadFactor() {
        return _lowerLoadFactor;
    }

    /**
     * @return The higher load factor of the table.
     */
    protected float getUpperLoadFactor() {
        return _upperLoadFactor;
    }

    /**
     * Clamps hashing indices to fit within the current table capacity.
     * @param index the index before clamping.
     * @return an index properly clamped.
     */
    protected int clamp(int index) {
        // index % tableSize-1
        return index & (capacity() - 1);
    }

    /**
     * Checks if the current load factor of the table is within the bounds
     * @param loadFactor The load factor of the table (size/capacity)
     * @return -1 if less, 1 if greater, 0 if within.
     */
    protected int relativeSize(float loadFactor) {
        if (loadFactor < this.getLowerLoadFactor() && capacity() > 1) {
            return SMALL;
        }
        if (loadFactor > this.getUpperLoadFactor()) {
            return BIG;
        }
        return GOOD;
    }

    /**
     * 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
     */
    public abstract boolean add(String newValue);

```

```
    /**
     * Look for a specified value in the set.
     * @param searchVal Value to search for
     * @return True iff searchVal is found in the set
     */
    public abstract boolean contains(String searchVal);

    /**
     * Remove the input element from the set.
     * @param toDelete Value to delete
     * @return True iff toDelete is found and deleted
     */
    public abstract boolean delete(String toDelete);

}
```

9 SimpleSetPerformanceAnalyzer.java

```java
import java.util.*;

/**
 * Simple class analyzes the performance of these five data structures:
 * OpenHashSet, ClosedHashSet, LinkedList, HashSet & TreeSet.
 */
public class SimpleSetPerformanceAnalyzer {

    private static final int DASTs_NUM = 5;
    private static final String[] DASTsNames= new String[]{"OpenHashSet", "ClosedHashSet",
                                                           "HashSet", "TreeSet", "LinkedList"};
    private static final int NANO_TO_MILL_FACTOR = 1000000;
    private static final int WARMUP_TIME = 70000;

    private static final String[] data1 = Ex4Utils.file2array("./src/data1.txt");
    private static final String[] data2 = Ex4Utils.file2array("./src/data2.txt");

    /** Array of SimpleSet data structures, each initialized with the words in data1.txt.*/
    private static SimpleSet[] DASTs_data1;
    /** Array of SimpleSet data structures, each initialized with the words in data2.txt.*/
    private static SimpleSet[] DASTs_data2;

    /**
     * Sets up the test resources
     */
    private static void setUP() {
        System.out.println(" ... Preparing test sources ... \n" +
                           " .. THIS WILL TAKE A FEW MINUTES .. ");
        System.out.println(" Building Array of SimpleSet data structures," +
                           " each initialized with the words in data1.txt.");
        long timeBefore1 = System.nanoTime();
        DASTs_data1 = DASTsFromListFactory(data1);
        long difference1 = (System.nanoTime() - timeBefore1) / NANO_TO_MILL_FACTOR;
        System.out.println(" Time: " + difference1 + " milliseconds.");

        System.out.println(" Building Array of SimpleSet data structures," +
                           " each initialized with the words in data2.txt.");
        long timeBefore2 = System.nanoTime();
        DASTs_data2 = DASTsFromListFactory(data2);
        long difference2 = (System.nanoTime() - timeBefore2) / NANO_TO_MILL_FACTOR;
        System.out.println(" Time: " + difference2 + " milliseconds.\n");
    }

    /**
     * Creates a SimpleSet array and initialize it with the 5 aforementioned DASTs
     * @return a SimpleSet array
     */
    private static SimpleSet[] DASTsDefaultFactory() {
        SimpleSet[] DASTsList = new SimpleSet[DASTs_NUM];
        DASTsList[0] = new OpenHashSet();
        DASTsList[1] = new ClosedHashSet();
        DASTsList[2] = new CollectionFacadeSet(new HashSet<String>());
        DASTsList[3] = new CollectionFacadeSet(new TreeSet<String>());
        DASTsList[4] = new CollectionFacadeSet(new LinkedList<String>());
        return DASTsList;
    }

    /**
     * Creates a SimpleSet array and initialize it with the 5 aforementioned DASTs
```

```java
     * @return a SimpleSet array
     */
    private static SimpleSet[] DASTsFromListFactory(String[] dataSet) {
        SimpleSet[] DASTsList = new SimpleSet[DASTs_NUM];
        DASTsList[0] = new OpenHashSet(dataSet);
        DASTsList[1] = new ClosedHashSet(dataSet);
        DASTsList[2] = new CollectionFacadeSet(new HashSet<String>(Arrays.asList(dataSet)));
        DASTsList[3] = new CollectionFacadeSet(new TreeSet<String>(Arrays.asList(dataSet)));
        DASTsList[4] = new CollectionFacadeSet(new LinkedList<String>(Arrays.asList(dataSet)));
        return DASTsList;
    }

    /**
     * Adding all the words in data1.txt, one by one, to each of the data structures.
     */
    private static void addTest1() {
        System.out.println(" === add(String value) TEST - data1.txt === ");
        SimpleSet[] DASTs = DASTsDefaultFactory();
        for (int i=0; i<DASTs_NUM; ++i) {
            System.out.printf("(%d) %s ... ", i+1, DASTsNames[i]);
            assert data1 != null;
            long timeBefore = System.nanoTime();
            for (String value: data1) {
                DASTs[i].add(value);
            }
            long difference = (System.nanoTime() - timeBefore) / NANO_TO_MILL_FACTOR;
            System.out.println("Time: " + difference + " milliseconds.");
        }
        System.out.println();
    }

    /**
     * Adding all the words in data2.txt, one by one, to each of the data structures.
     */
    private static void addTest2() {
        System.out.println(" === add(String value) TEST - data2.txt === ");
        SimpleSet[] DASTs = DASTsDefaultFactory();
        for (int i=0; i<DASTs_NUM; ++i) {
            System.out.printf("(%d) %s ... ", i+1, DASTsNames[i]);
            assert data2 != null;
            long timeBefore = System.nanoTime();
            for (String value: data2) {
                DASTs[i].add(value);
            }
            long difference = (System.nanoTime() - timeBefore) / NANO_TO_MILL_FACTOR;
            System.out.println("Time: " + difference + " milliseconds.");
        }
        System.out.println();
    }

    /**
     * Searching the string "hi" in data1.txt, for each data structure.
     */
    private static void containsHiTest1() {
        System.out.println(" === contains(hi) TEST - data1.txt === ");
        SimpleSet[] DASTs = DASTs_data1;
        for (int i=0; i<DASTs_NUM; ++i) {
            System.out.printf("(%d) %s ... ", i+1, DASTsNames[i]);
            if (i<DASTs_NUM-1) {
                // WARM UP
                for (int j=0; j<WARMUP_TIME; ++j)
                    DASTs[i].contains("hi");
            }
            long timeBefore = System.nanoTime();
            for (int j=0; j<WARMUP_TIME; ++j)
                DASTs[i].contains("hi");
            long difference = (System.nanoTime() - timeBefore) / WARMUP_TIME;
            System.out.println("Time: " + difference + " nanoseconds.");
```

```
        }
        System.out.println();
    }

    /**
     * Searching the string "-13170890158" in data1.txt, for each data structure.
     */
    private static void containsTest1() {
        System.out.println(" === contains(-13170890158) TEST - data1.txt === ");
        SimpleSet[] DASTs = DASTs_data1;
        for (int i=0; i<DASTs_NUM; ++i) {
            System.out.printf("(%d) %s", i+1, DASTsNames[i]);
            if (i<DASTs_NUM-1) {
                // WARM UP
                for (int j=0; j<WARMUP_TIME; ++j)
                    DASTs[i].contains("-13170890158");
            }
            long timeBefore = System.nanoTime();
            for (int j=0; j<WARMUP_TIME; ++j)
                DASTs[i].contains("-13170890158");
            long difference = (System.nanoTime() - timeBefore) / WARMUP_TIME;
            System.out.println("Time: " + difference + " nanoseconds.");
        }
        System.out.println();
    }

    /**
     * Searching the string "23" in data1.txt, for each data structure.
     */
    private static void containsTest2() {
        System.out.println(" === contains(23) TEST - data2.txt === ");
        SimpleSet[] DASTs = DASTs_data2;
        for (int i=0; i<DASTs_NUM; ++i) {
            System.out.printf("(%d) %s", i+1, DASTsNames[i]);
            if (i<DASTs_NUM-1) {
                // WARM UP
                for (int j=0; j<WARMUP_TIME; ++j)
                    DASTs[i].contains("23");
            }
            long timeBefore = System.nanoTime();
            for (int j=0; j<WARMUP_TIME; ++j)
                DASTs[i].contains("23");
            long difference = (System.nanoTime() - timeBefore) / WARMUP_TIME;
            System.out.println("Time: " + difference + " nanoseconds.");
        }
        System.out.println();
    }

    /**
     * Searching the string "hi" in data2.txt, for each data structure.
     */
    private static void containsHiTest2() {
        System.out.println(" === contains(hi) TEST - data2.txt === ");
        SimpleSet[] DASTs = DASTs_data2;
        for (int i=0; i<DASTs_NUM; ++i) {
            System.out.printf("(%d) %s", i+1, DASTsNames[i]);
            if (i<DASTs_NUM-1) {
                // WARM UP
                for (int j=0; j<WARMUP_TIME; ++j)
                    DASTs[i].contains("hi");
            }
            long timeBefore = System.nanoTime();
            for (int j=0; j<WARMUP_TIME; ++j)
                DASTs[i].contains("hi");
            long difference = (System.nanoTime() - timeBefore) / WARMUP_TIME;
            System.out.println("Time: " + difference + " nanoseconds.");
        }
        System.out.println();
```

```java
    }

    public static void main(String[] args) {
        setUP();
        addTest1();
        addTest2();
        containsHiTest1();
        containsTest1();
        containsTest2();
        containsHiTest2();
    }


}
```