Contents

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

```
brahan



=============================
=      File description     =
=============================
The Jar file contain 8 files including the README file you are reading right now :)
- SimpleSet.java - contain interface class that contain the basic method  of a Set
- SimpleHashSet.java - an abstract class that implement SimpleSet, contain the basic
                skeleton for hashTable
- ClosedHashSet.java - extend the SimpleHashSet class, contain a implementation
of a HashSet  based on closed-hashing with quadratic probing.
- OpenHashSet.java - extend the SimpleHashSet class, contain a implementation of a
    HashSet based on open-hashing with chaining.
- CollectionFacadeSet.java - contains a class  which wraps an underlying Collection and serves
    to both simplify its API and give it a common type with the implemented SimpleHashSets.
- SimpleSetPerformanceAnalyzer.java -has a main method that measures the run-times requested
                      in the "Performance Analysis" section.
- RESULTS - file that includes all the   measures the run-times requested in the
"Performance Analysis" section.


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

Most of the design has been defined by the exercise pdf.
I tried to keep the code as easy to understand as possible, while maintaining modularity and
encapsulation concept's we have learned in the lectures and tirgulim.
To maintain easy to read project, I used many helper methods, so that big methods like add and
delete are broken into mini methods that can be changed and read easily.
To maintain encapsulation i de-abstract some of the SimpleHashSet methods, and added setter to the
capacity a method that relvent to both of SimpleHashSet "childs" and mange the capacity changes
through the project. For the OpenHashSet I chose to use the Wrapper class solution that represented
in the pdf, I created the class as nested one, because as we learned in the lecture if we have a
class with a single propose, and we will use the object only at the OpenHashClass so it hold all
the conditions to be a nested class. For the Facade in my understanding all we needed to do is
maintain all the collections that pass through our Facade to be Set, so all we needed to do is to
check for duplicated items. For the ClosedHashSet I didnt had to use any "smart" design choices,
the class just extends the SimpleHashSet.


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

In you README, discuss the following:
• How you implemented OpenHashSet's table
I chose to implement the OpehHashTable with a Wrapper class of LinkedList which implemented
as a nested class of the class.
There were 2 main reasons i chose to do so.
When I started to write the exercise out of the 3 options given in the pdf, Wrapper was the easiest
to understand, and easiest to implement.
The second reason was sort of confirmation that I had choose the right option.
After thinking about extending CollectionFacade and learning about what Facade is
(done after i finished OpenHashSet), It made sense to use that option, but thinking about it more
have raised a problem I will check contain twice (the facade add need to maintain a set, so we have
to check if the item is alerdy in the set) but we alerdy check that when we adding to the openHashSet.
```

```
In my understanding running time are important concept in this exercise, by checking contain
twice we will get worst running time, then the wrapper. - this assumption I had in the begging
exercise have been proved in my opinion when I compared my running time into running
time of other people in the course.
• How you implemented the deletion mechanism in ClosedHashSet
I chose to define a string that will indicate that an object has been deleted.
I chose so for 2 main reasons, one was that it was the most straight-forward solution to the problem.
The second reason was more "Why I shouldnt use other solution".
By not defining another class that will solve this problem, I avoid "abuse" (In my opinion)
of objects, each class have to maintain encapsulation, by defining more and more class
it makes it harder to maintain so. A single private string solve both of those problems
without any complications. To assure that the solution wont be a restriction on the strings that
the user can insert, we use == to compare references and not .equals("deleted object")
 which compare values.


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

In you README, discuss the following:
• Discuss the results of the analysis in depth:
- Account, in separate, for OpenHashSet's and ClosedHashSet's bad results for data1.txt
As we have been told data1 contains 100k of items with the same hashCode, if we look at the results of
add:
OpenHashSet_AddData1 = 35107
ClosedHashSet_AddData1 = 76347
TreeSet_AddData1 = 58
LinkedList_AddData1 = 29237
HashSet_AddData1 = 39
we see that open hash and LinkedList have similar running time. because all the items in the will
be hashed into the same cell in the Table we get a similar to a regular linkedlist dataStructure
(technically its just an array full of empty linked lists except of one cell that hold all the items).
As we cann learn from the results we see that when hashing to the same place closed hash is the least
efficient, although its expected to do so because of the way clamp work in ClosedHash. The clamp
need to iterate 100k timesfor each item because we have a collision each insert.
- 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?
If we compare only closed and open hash we can see that when we expected to have alot of collision
uwe should se openHash, because its handle it faster.
but if we look at the running time of add of data2 to the SimpleSets:
OpenHashSet_AddData2 = 31
ClosedHashSet_AddData2 = 10
TreeSet_AddData2 = 36
LinkedList_AddData2 = 16959
HashSet_AddData2 = 5
We can see that closedHashSet is better at inserting, the trivial explanation for this is that for items
with diffrenet hashCode we are adding items with no collision between them, meaning its act similar to
adding into an array. OpenHashSet also handle this pretty well with only 31.
Its no sunrise that LinkedList has the worst adding time, as we learned in dast adding into linkedList
while maintaining a Set structure will give us bad time and we shouldn't use it unless we have to.
So in Total if we want to add items between Open and Closed we will choose
open if we have alot of collision, closed if have less collision.
If we can use Java HashSet we will always use it - as We can see he have the BEST adding running time.
As for searching an item we can see:
data1:
OpenHashSet_Contains_hi1 = 21
ClosedHashSet_Contains_hi1 = 11
TreeSet_Contains_hi1 = 80
LinkedList_Contains_hi1 = 471834
HashSet_Contains_hi1 = 26
data2:
OpenHashSet_Contains_hi2 = 7
ClosedHashSet_Contains_hi2 = 20
TreeSet_Contains_hi2 = 68
LinkedList_Contains_hi2 = 391317
```

```
HashSet_Contains_hi2 = 10
Its obvius that the LLS got worst running time because the DataStracuse isnt built for fast search
like hashtable.
Between Open and Closed its surprising to see that we got that the running time are inverted.
Open was better at adding data 1 but got worst running time searching in it, same for closed and
data2. We can try to explain that with maybe Open handle adding better but to search an item is
similar to searching in linkedlist(but shorter one). and searching in closed is similar to searching
in array.
More interesting is to see the contain for the negative number, we have been told
that the number has the same hashcode as the items in data1:
OpenHashSet_Contains_negative = 559693
ClosedHashSet_Contains_negative =940124
TreeSet_Contains_negative = 119
LinkedList_Contains_negative = 540640
HashSet_Contains_negative =24
My worst results was given by this value, Its not very surprising because in closedHashSet we will
have to check all the item, without skipping on any (meaning O(n)), same for open that will
acts excatly like a linkedList as we can see their running time are very similar.
- Not mandatory: Did you find java's HashSet performance on data1.txt surprising? Can
you explain it?
java hashSet are surprising, as we can see we got a scalable "tens" in all the method that involved
it, We can just assume that java implement the DS with replacing the hashfunction from time time
(assumption after its passed some collision threshold). By doing so java assure uniform distribution
in the DS.

=============================
=        HashTable          =
=============================
:)
```

3 ClosedHashSet.java

```java
/**
 * a hash-set based on closed-hashing with quadratic probing. Extends SimpleHashSet
 */
public class ClosedHashSet extends SimpleHashSet {

    private static final String DELETED = "deleted object";
    private String[] closedHashTable;
    private static final int QUAD_PROBING_DENOMINATOR = 2;
    private static final int ITEM_IS_FOUND = 1;
    private static final int ITEM_ISNT_FOUND = 0;
    private static final int ITEM_DELETED_SUCCESS = 2;

    /**
     * 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();
        this.closedHashTable = 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);
        this.closedHashTable = 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) {
        this();
        for (String aData : data) {
            add(aData);
        }
    }

    /**
     * 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 loopHelper(searchVal, true) == ITEM_IS_FOUND;
    }

    /**
     * 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)) {// no duplicate allowed
            return false;
        } else {
            if (checkTableLoad(true) == BIGGER_REHASH) {// need to check rehash
                rehashBiggerTable();
            }
            int valueIndex = findHashIndex(newValue);
            this.closedHashTable[valueIndex] = newValue; // reg add
            this.size++;
            return true;
        }
    }

    /**
     * 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)) {
            return false;
        } else {
            loopHelper(toDelete, false);
            if (checkTableLoad(false) == SMALLER_REHASH) {
                rehashSmallerTable();
            }
            this.size--;
            return true;
        }
    }

    /**
     * Clamps hashing indices to fit within the current table capacity
     *
     * @param index the index before clamping
     * @return an index properly clamped
     */
    @Override
    protected int clamp(int index) {
        for (int i = 0; i < this.capacity; i++) {
            int quadraticProbing = (i + i * i) / QUAD_PROBING_DENOMINATOR;
            index = (index + quadraticProbing) & capacityMinusOne;
            if ((this.closedHashTable[index] == null) ||
                    (this.closedHashTable[index] == (DELETED))) { // empty or deleted
                return index;
            }
        }
        return index;
    }

    /**
     * in case current load factor is bigger then the upper load factor
     * we will need to rehash all the items into a new table
     */
    @Override
    protected void rehashBiggerTable() {
        int newCapacity = this.capacity * RESIZE_FACTOR;
        String[] temp = this.closedHashTable;
        this.closedHashTable = new String[newCapacity];
        rehash(temp, newCapacity);
    }

    /**
     * in case current load factor is smaller then the lower load factor
     * we will need to rehash all the items into a new table
     */
    @Override
    protected void rehashSmallerTable() {
        int newCapacity = this.capacity / RESIZE_FACTOR;
        String[] temp = this.closedHashTable;
        this.closedHashTable = new String[newCapacity];
        rehash(temp, newCapacity);
    }

    /**
     * the method responsible on resizing the table and rehashing all the items
     *
     * @param oldTable    the table before resizeing , we will rehash its item into a new Table
     * @param newCapacity our future capacity
     */
    private void rehash(String[] oldTable, int newCapacity) {
        setCapacity(newCapacity);
        for (String item : oldTable) {
            if (item != null && item != (DELETED)) {
                int itemIndex = findHashIndex(item);
                this.closedHashTable[itemIndex] = item;
            }
        }
    }

    /**
     * Helper method for delete and contain methods, loop over the items and return an
     * integer that represent
     * if the condition is held
     *
     * @param string    the string value we want to search for
     * @param isContain flag that indicate which method called this one, contain or delete
     * @return integer that indicate the situation
     */
    private int loopHelper(String string, boolean isContain) {
        int valPlace = string.hashCode();
        int quadraticProbing;
        for (int i = 0; i < this.capacity; i++) { // loop
            quadraticProbing = (i + i * i) / QUAD_PROBING_DENOMINATOR;
            valPlace = (valPlace + quadraticProbing) & capacityMinusOne;
            if (isContain) { // contain
                if (this.closedHashTable[valPlace] == null) {
                    return ITEM_ISNT_FOUND;
                } else if (this.closedHashTable[valPlace].equals(string)&&
                        this.closedHashTable[valPlace]!=DELETED) {
                    return ITEM_IS_FOUND;
                }
            } else { // delete
                if ((closedHashTable[valPlace] != null) &&
                        (closedHashTable[valPlace].equals(string)) &&
                        (closedHashTable[valPlace] != (DELETED))) {
                    this.closedHashTable[valPlace] = DELETED;
                    return ITEM_DELETED_SUCCESS;
                }
            }

        }
        return ITEM_ISNT_FOUND;
    }
}
```

4 CollectionFacadeSet.java

```java
import java.util.Collection; // LinkedList,TreeSet,HashSet
import java.util.TreeSet;

/**
 * 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 {

    private Collection<String> collection;

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

    /**
     * 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 boolean add(String newValue) {
        if (this.collection.contains(newValue)) {
            return false;
        } else {
            this.collection.add(newValue);
            return true;
        }
    }

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

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

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

}
```

5 OpenHashSet.java

```java
import java.util.*;

/**
 * a hash-set based on chaining. Extends SimpleHashSet.
 */
public class OpenHashSet extends SimpleHashSet {

    /**
     * wrapper-class that has a LinkedList<String> and delegates methods to it, and have
     * an array of that class instead
     */
    private WrappedLinkedList[] openHashTable;

    /**
     * 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();
        this.openHashTable = new WrappedLinkedList[INITIAL_CAPACITY];
        bucketCreator(openHashTable, 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);
        this.openHashTable = new WrappedLinkedList[INITIAL_CAPACITY];
        bucketCreator(openHashTable, 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) {
        this();
        for (String aData : data) {
            add(aData);
        }
    }

    /**
     * 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) || newValue == null) {
            return false;
        } else {
            if (checkTableLoad(true) == BIGGER_REHASH) {
                rehashBiggerTable();
            }
            int valIndex = findHashIndex(newValue);
            this.openHashTable[valIndex].getLinkedList().add(newValue);
            this.size++;
            return true;
        }
    }

    /**
     * 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) || toDelete == null) {
            return false;
        } else {
            int valToDeleteIndex = findHashIndex(toDelete);
            this.openHashTable[valToDeleteIndex].getLinkedList().remove(toDelete);
            if (checkTableLoad(false) == SMALLER_REHASH) {
                rehashSmallerTable();
            }
            this.size--;
            return true;
        }
    }

    /**
     * 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 (searchVal == null) {
            return false;
        } else {
            int valueIndex = findHashIndex(searchVal);
            return this.openHashTable[valueIndex].getLinkedList().contains(searchVal);
        }
    }

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

    /**
     * in case current load factor is bigger then the upper load factor
     * we will need to rehash all the items into a new table
     */
    @Override
    protected void rehashBiggerTable() {
        int newCapacity = this.capacity * RESIZE_FACTOR;
        WrappedLinkedList[] newTable = new WrappedLinkedList[newCapacity];
        bucketCreator(newTable, newCapacity);
        this.openHashTable = rehash(newTable, newCapacity);
    }

    /**
     * in case current load factor is smaller then the lower load factor
     * we will need to rehash all the items into a new table
     */
    @Override
    protected void rehashSmallerTable() {
        int newCapacity = this.capacity / RESIZE_FACTOR;
        WrappedLinkedList[] newTable = new WrappedLinkedList[newCapacity];
        bucketCreator(newTable, newCapacity);
        this.openHashTable = rehash(newTable, newCapacity);
    }

    /**
     * rehash all the item from the old table into a new table
     *
     * @param newTable    our new Table, we will be adding all the items from the old one
     *                    to this one
     * @param newCapacity our new capacity, defined by the resize factor
     * @return a new hashTable with the items as the old one but different size
     */
    private WrappedLinkedList[] rehash(WrappedLinkedList[] newTable, int newCapacity) {
        setCapacity(newCapacity);
        for (WrappedLinkedList bucket : this.openHashTable) {
            if (!bucket.getLinkedList().isEmpty()) {
                for (int j = 0; j < bucket.getLinkedList().size(); j++) {
                    String val = bucket.getLinkedList().get(j);
                    int valIndex = findHashIndex(val);
                    newTable[valIndex].getLinkedList().add(val);
                }
            }
        }
        return newTable;
    }

    /**
     * the method create bucket for each cell in the array
     *
     * @param table  the table we want to turn into a proper hashTable
     * @param hashTableLength the number of buckets we need to create
     */
    private void bucketCreator(WrappedLinkedList[] table, int hashTableLength) {
        for (int i = 0; i < hashTableLength; i++) {
            table[i] = new WrappedLinkedList();
        }
    }

    /**
     * nested class that represent a wrapped linked list, the "buckets" of the table
     */
    private class WrappedLinkedList {

        private LinkedList<String> linkedLst;

        /**
         * a constructor for the class, creates LinkedList
         */
        private WrappedLinkedList() {
            this.linkedLst = new LinkedList<String>();
        }

        /**
         * a getter method for the linked list object we created
         *
```

```
             * @return linked list
             */
            protected LinkedList<String> getLinkedList() {
                return this.linkedLst;
            }

        }
    }
```

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 take+s (in ms) to insert data1 to all data structures
OpenHashSet_AddData1 = 36967
ClosedHashSet_AddData1 = 74554
TreeSet_AddData1 = 63
LinkedList_AddData1 = 31949
HashSet_AddData1 = 39

#These values correspond to the time it takes (in ms) to insert data2 to all data structures
OpenHashSet_AddData2 = 25
ClosedHashSet_AddData2 = 10
TreeSet_AddData2 = 38
LinkedList_AddData2 = 17055
HashSet_AddData2 = 6

#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 = 11
TreeSet_Contains_hi1 = 93
LinkedList_Contains_hi1 = 455661
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 = 560342
ClosedHashSet_Contains_negative =974330
TreeSet_Contains_negative = 124
LinkedList_Contains_negative = 550916
HashSet_Contains_negative =25

#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 = 19
ClosedHashSet_Contains_23 = 18
TreeSet_Contains_23 = 41
LinkedList_Contains_23 = 131
HashSet_Contains_23 = 13


#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 = 10
ClosedHashSet_Contains_hi2 = 18
TreeSet_Contains_hi2 = 67
LinkedList_Contains_hi2 = 407893
HashSet_Contains_hi2 = 11
```

7 SimpleHashSet.java

```java
/**
 * an abstract superclass for implementations of hash-sets implementing the SimpleSet interface.
 */
public abstract class SimpleHashSet implements SimpleSet {
    private static final float DEFAULT_HIGHER_CAPACITY = 0.75f;
    private static final float DEFAULT_LOWER_CAPACITY = 0.25f;
    private static final int MIN_SIZE = 1; // minimum capacity size
    private static final int EMPTY_TABLE_SIZE = 0;
    protected static final int INITIAL_CAPACITY = 16;
    private float upperLoadFactor;
    private float lowerLoadFactor;
    protected int capacity;
    protected int capacityMinusOne;
    protected int size;
    protected static final int BIGGER_REHASH = 1;
    protected static final int SMALLER_REHASH = -1;
    protected static final int NO_REHASH = 0;
    protected static final int RESIZE_FACTOR = 2;


    /**
     * 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 SimpleHashSet() {
        baseValues();
        this.upperLoadFactor = DEFAULT_HIGHER_CAPACITY;
        this.lowerLoadFactor = DEFAULT_LOWER_CAPACITY;
        this.capacityMinusOne = this.capacity - 1;
    }

    /**
     * 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 SimpleHashSet(float upperLoadFactor, float lowerLoadFactor) {
        baseValues();
        this.upperLoadFactor = upperLoadFactor;
        this.lowerLoadFactor = lowerLoadFactor;
        this.capacityMinusOne = this.capacity - 1;
    }

    /**
     * @param newValue New value to add to the set
     * @return False iff newValue already exists in the set
     */
    @Override
    public abstract boolean add(String newValue);

    /**
     * @param searchVal Value to search for
     * @return True iff searchVal is found in the set
     */
    @Override
    public abstract boolean contains(String searchVal);

    /**
     * @param toDelete Value to delete
     * @return True iff toDelete is found and deleted
     */
    @Override
    public abstract boolean delete(String toDelete);

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

    /**
     * capacity in class SimpleHashSet
     *
     * @return The current capacity (number of cells) of the table.
     */
    public int capacity() {
        return this.capacity;
    }

    /**
     * getter for the upperLoadFactor
     *
     * @return The higher load factor of the table.
     */
    public float getUpperLoadFactor() {
        return this.upperLoadFactor;
    }

    /**
     * getter for the lowerLoadFactor
     *
     * @return The lower load factor of the table.
     */
    public float getLowerLoadFactor() {
        return this.lowerLoadFactor;
    }

    /**
     * a setter to the capacity field
     *
     * @param newCapacity the capacity which we want to set
     */
    protected void setCapacity(int newCapacity) {
        if (newCapacity >= MIN_SIZE) {
            this.capacity = newCapacity;
            this.capacityMinusOne = newCapacity - 1;
        } else {
            this.capacity = MIN_SIZE;
            this.capacityMinusOne = 0;
        }
    }

    /**
     * Clamps hashing indices to fit within the current table capacity
     *
     * @param index the index before clamping
     * @return an index properly clamped
     */
    protected abstract int clamp(int index);

    /**
     * get the index that our val should have in the hashTable and then clamp it
     * to our own hashtable size range
     *
     * @param val the value which we want to find its index in our hashtable
     * @return the index of the value after clamping
     */
    protected int findHashIndex(String val) {
        return clamp(val.hashCode());
    }

    /**
     * check if we need to rehash our table using the upper/lower load factors
     *
     * @return 1 if we need a bigger table, 0 if dont need to rehash, -1 if need smaller table
     */
    protected int checkTableLoad(boolean isAdd) {
        int checkUpper = this.size + 1;
        int checkLower = this.size - 1;
        float curLoadUpper = (float) (checkUpper) / (float) this.capacity;
        float curLoadLower = (float) (checkLower) / (float) this.capacity;
        if (isAdd && (curLoadUpper > this.upperLoadFactor || curLoadUpper > 1)) {
            return BIGGER_REHASH;
        }
        if (!isAdd && (curLoadLower < this.lowerLoadFactor && curLoadUpper <= 1)) {
            return SMALLER_REHASH;
        }
        return NO_REHASH;
    }

    /**
     * in case current load factor is bigger then the upper load factor
     * we will need to rehash all the items into a new table
     */
    protected abstract void rehashBiggerTable();

    /**
     * in case current load factor is smaller then the lower load factor
     * we will need to rehash all the items into a new table
     */
    protected abstract void rehashSmallerTable();

    /**
     * define the hashtable fields into the default values as defined in the pdf
     */
    private void baseValues() {
        this.capacity = INITIAL_CAPACITY;
        this.size = EMPTY_TABLE_SIZE;
    }
}
```

8 SimpleSet.java

```java
/**
 * an interface consisting of the add(), delete(), contains(), and size()  methods
 */
public interface SimpleSet {

    /**
     * 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
     */
    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
     */
    boolean contains(String searchVal);

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

    /**
     * @return The number of elements currently in the set
     */
    int size();
}
```

9 SimpleSetPerformanceAnalyzer.java

```java
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Scanner;
import java.util.TreeSet;

/**
 * has a main method that measures the run-times requested
 * in the "Performance Analysis" section.
 */
public class SimpleSetPerformanceAnalyzer {

    private static final String DATA1 = "data1.txt";
    private static final String DATA2 = "data2.txt";
    private static final int NS_TO_MS_FACTOR = 1000000;
    private static final int WARM_UP_SETS = 70000;
    private static final int WARM_UP_LLS = 7000;
    private static final String SEARCH_VAL_HI = "hi";
    private static final String SEARCH_VAL_NUM = "23";
    private static final String SEARCH_VAL_NEGATIVE_NUM = "-13170890158";
    private static final int TEST_ADD_ALL_DATA1 = 1;
    private static final int TEST_ADD_ALL_DATA2 = 2;
    private static final int TEST_SEARCH_HI_DATA1 = 3;
    private static final int TEST_SEARCH_NEGATIVE_NUM_DATA1 = 4;
    private static final int TEST_SEARCH_NUM_DATA2 = 5;
    private static final int TEST_SEARCH_HI_DATA2 = 6;
    private static final int TEST_ALL = 7;
    private static SimpleSet[] simpleSets;
    private static final String SET_ONE_OPEN_HASH = "OpenHashSet";
    private static final String SET_TWO_CLOSED_HASH = "ClosedHashSet";
    private static final String SET_THREE_TREE = "TreeSet";
    private static final String SET_FOUR_LLS = "LinkedList";
    private static final String SET_FIVE_HASH = "HashSet";

    private static final String[] simpleSetsTypes = {SET_ONE_OPEN_HASH, SET_TWO_CLOSED_HASH,
            SET_THREE_TREE, SET_FOUR_LLS, SET_FIVE_HASH};
    private static final String[] dataOneArray = Ex4Utils.file2array(DATA1);
    private static final String[] dataTwoArray = Ex4Utils.file2array(DATA2);

    /**
     * Init an array of SimpleSets, with all the dataSets requested in the "Performance Analysis"
     */
    private static void init() {
        simpleSets = new SimpleSet[5];
        simpleSets[0] = new OpenHashSet();
        simpleSets[1] = new ClosedHashSet();
        simpleSets[2] = new CollectionFacadeSet(new TreeSet<String>());
        simpleSets[3] = new CollectionFacadeSet(new LinkedList<String>());
        simpleSets[4] = new CollectionFacadeSet(new HashSet<String>());

    }

    /**
     * helper method for the contain Tests, add all the data into the SimpleSet
     *
     * @param data the data we want our SimpleSet will hold
     */
    private static void addData(String[] data) {
        for (SimpleSet set : simpleSets) {
            for (String item : data) {
                set.add(item);
            }
        }
    }

    /**
     * test how much time it takes to add all the data into the SimpleSet
     *
     * @param data the data we want our SimpleSet will hold
     */
    private static void checkAddAllData(String[] data) { // in ms
        init();
        for (int i = 0; i < simpleSetsTypes.length; i++) {
            System.out.println("Start adding to " + simpleSetsTypes[i]);
            long startTime = System.nanoTime();
            for (String item : data) {
                simpleSets[i].add(item);
            }
            long endTime = System.nanoTime();
            long total = (endTime - startTime) / NS_TO_MS_FACTOR;
            System.out.println("Adding all the given data to " + simpleSetsTypes[i] + " took " + total);
        }

    }

    /**
     * test how much time it takes to find a value in the SimpleSet
     *
     * @param data      the data we want our SimpleSet to hold
     * @param searchVal the value we want to check how much time it will take to find in the
     *                  SimpleSet
     */
    private static void checkContain(String[] data, String searchVal, boolean isTestAll) {
        if (isTestAll) {
            checkContainSetHelper(searchVal);
        } else {
            System.out.println("Start Initialization");
            init();
            addData(data);
            System.out.println("Finished Initialization");
            checkContainSetHelper(searchVal);
        }
    }

    /**
     * the method reposnsible to mange the warm stage for the contains
     *
     * @param value the string we are looking for in teh
     */
    private static void checkContainSetHelper(String value) {
        for (int i = 0; i < 5; i++) {
            if (simpleSets[i] instanceof LinkedList) {
                checkContainLoopHelper(value, WARM_UP_LLS, i);
            } else {
                checkContainLoopHelper(value, WARM_UP_SETS, i);
            }
        }
    }

    /**
     * a helper method for the checkContain method,
     *
     * @param value  the value we want to search for
     * @param warmUp how much warmUp rounds is needed for the data set
     * @param set    the SimpleSet we to check
     */
    private static void checkContainLoopHelper(String value, int warmUp, int set) {
        if (!(simpleSets[set] instanceof LinkedList)) {
            for (int j = 0; j < warmUp; j++) {
                simpleSets[set].contains(value);
            }
        }
        long startTime = System.nanoTime();
        for (int j = 0; j < warmUp; j++) {
            simpleSets[set].contains(value);
        }
        long endTime = System.nanoTime();
        long total = (endTime - startTime) / warmUp;
        System.out.println("For the item " + value +
                " total time of contain for " + simpleSetsTypes[set] + " " + total);
    }

    /**
     * manage the testing process,activate the needed test according userInput he will get in the
     * main method
     *
     * @param testNumber an integer that indicate which test to activate
     */
    private static void chooseTest(int testNumber) {
        if (testNumber == TEST_ADD_ALL_DATA1) { // test how much it takes to add all data1
            checkAddAllData(dataOneArray);
        } else if (testNumber == TEST_ADD_ALL_DATA2) { // test how much it takes to add all data2
            checkAddAllData(dataTwoArray);
        } else if (testNumber == TEST_SEARCH_HI_DATA1) { // data 1 contain "hi"
            checkContain(dataOneArray, SEARCH_VAL_HI, false);
        } else if (testNumber == TEST_SEARCH_NEGATIVE_NUM_DATA1) { // data 1 contain "-13170890158"
            checkContain(dataOneArray, SEARCH_VAL_NEGATIVE_NUM, false);
        } else if (testNumber == TEST_SEARCH_NUM_DATA2) { // data 2 contain "23"
            checkContain(dataTwoArray, SEARCH_VAL_NUM, false);
        } else if (testNumber == TEST_SEARCH_HI_DATA2) { // data 2 contain "hi"
            checkContain(dataTwoArray, SEARCH_VAL_HI, false);
        } else if (testNumber == TEST_ALL) { // run all tests
            checkAddAllData(dataOneArray);
            checkContain(dataOneArray, SEARCH_VAL_HI, true);
            checkContain(dataOneArray, SEARCH_VAL_NEGATIVE_NUM, true);
            checkAddAllData(dataTwoArray);
            checkContain(dataTwoArray, SEARCH_VAL_NUM, true);
            checkContain(dataTwoArray, SEARCH_VAL_HI, true);
        } else {
            System.out.println("Not A Valid Input");
        }
    }

    /**
     * the method responsible to mange the main method
     */
    private static void mangeMain() {
        Scanner input = new Scanner(System.in);
        System.out.println("Which Test would you like to run: \n" +
                "Press 1 to check running time of add data1 \n" +
                "Press 2 to check running time of add data2 \n" +
                "Press 3 to check running time of contain 'hi' in data1 \n" +
                "Press 4 to check running time of contain '-13170890158' in data1 \n" +
                "Press 5 to check running time of contain '23' in data2 \n" +
                "Press 6 to check running time of contain 'hi' in data2 \n" +
                "Press 7 to check all the running time at once");
        int userInput = input.nextInt();
        chooseTest(userInput);
    }


    public static void main(String[] args) {
        mangeMain();
    }
}
```