Exercise 3 – HashSet \ OOP 2015

1. Goals

1. Improve understanding of hash tables by getting your hands dirty: you will write two implementations of the HashSet data structure.
2. Experiment with real-life results by comparing performances of five data structures: the aforementioned two, java.util.LinedList, java.util.TreeSet, and java.util.HashSet.

2. General Notes

- Submission deadline: **7.5.15 , 23:55**
- Submitted files may use java.util.TreeSet, java.util.LinkedList, java.util.HashSet (the last is not to be used in your own implementation obviously, only in the comparison) and classes and interfaces from the java.lang package in standard java 1.7 distribution. Don't use any other classes that you didn't write yourself.

3. Background

Sets and hash-sets

A Set is an abstract mathematical data structure (not necessarily abstract in Java's sense, but in the sense that it is implementation-independent) which represents an unordered collection of elements with no duplicates. In Java, having no duplicate elements means that for each two elements *a,b* in the set it holds that *a.equals(b)* is *false* (where 'equals' may be the inherited method from Object, or an overriding version of this method implemented by the user).
A Set supports the following operations: *add()*, *contains()*, *delete()*, and *size()*.
A hash-set is a common implementation of a Set that provides constant-time ($O(1)$) implementations of these operations, **in the average case**, if a good hash function is used with regard to the inserted elements. We'll be using the objects' hashCode method (which again, is either inherited from Object or overrides the inherited method). The hashCode method must return a numeric value such that every two objects that are "equal" also have the same hashCode (note that the opposite is not necessarily true). Although this is hashCode's only formal requirement, it is also desirable that the hash code distributes evenly among the hash-set valid numeric values indices in the, in order to achieve the desired constant-time performance.
Internally, the hash-set uses – you've guessed it – a hash table. When an element is added to the hash-set, it is hashed to determine an index in the hash table in which to try and place the element (we'll soon discuss what happens if the cell is already taken). If, during the insertion, another element in the table is discovered which is 'equal' to the inserted one, the set remains the same (no duplicates allowed!).
In this exercise we will deal only with sets of Strings. How will you fit the hashCodes to a valid index in the range [0:tableSize-1]? Note that String's hashCode method may, and will, return negative values, or values that are greater than the table size. Modulo is a good start, but in java (-3)%7 is still -3. Use $Math.abs(hashCode(value))\%tableSize$ (and see appendix A) to fit a hash-code to the range of legal indices.

Performance considerations and re-hashing

The performance of a hash table is affected by two parameters (aside from its actual input elements): *upper load factor* and *lower load factor*. The *Load factor* is defined as $\frac{M}{capacity}$ where *M* is the current

number of elements and the capacity is the size of the table. The *upper / lower load factors* determine how full / empty the table is allowed to get before its capacity is increased / decreased respectively. After changing the capacity (i.e. allocating a new table), **re-hashing** is required. Re-hashing means inserting all elements from the "old" table in the new one.
Note1: Testing the load factor and deciding to increase is only done when add is called. Decreasing – only in the scope of a delete operation. The table capacity can decrease to as low as 1.
Note2: Do you need to check for duplicates when re-hashing elements into the new table?

Chained hashing

In this model, each cell in the hash table is a list ("bucket"), and an element with the hash k is added to the k'th bucket (after it's fitted to the legal index range, using modulo and abs or using bitwise-and as suggested in Appendix A).

Note: In Java, you'd probably like to declare an array of linked lists such as "LinkedList<String>[]", but for reasons we will not discuss until later in the course, such arrays are illegal in Java. Here are some possible solutions:

- Implement a linked list of strings yourself (unrecommended).
- Define a wrapper-class that has-a LinkedList<String> and delegates methods to it, and have an array of that class instead.
- Have an array of CollectionFacadeSet (see "the classes you should implement"), or of a subclass of CollectionFacadeSet, where each such facadewraps a LinkedList<String>.

You can also think of a solution of your own, so long as encapsulation and a reasonable design are held. Explain your choice in the README.
However, we ask that you **do not** use an ArrayList<LinkedList<String>> instead of a standard array, since an ArrayList doesn't provide as much control over its capacity, which is an important part of managing a hash table.

Open hashing

In this model each cell in the hash table is a reference to a String. When a new string is mapped to an occupied cell, there's need to probe further in the array to find an empty spot. In this exercise we'll be using quadratic probing: the i'th attempt to find an empty cell for $value$, or to simply find $value$, will use the index

$$hash(value) + c_1 \cdot i + c_2 \cdot i^2$$

hich is then fitted to the appropriate range [0:tableSize-1]. Note that for i=0, which is the first attempt, this simply means the hashCode of $value$.

A special property of tables whose size is a power of two, is that $c_1 = c_2 = \frac{1}{2}$ ensures that a place will be found,at some attempt, for every new value as long as the table is not full, so those are the values we'll use.
Implementation issues:

- Using floating points will cause trouble: use $hash(value) + (i + i^2)/2$ which is always a whole number (why?) instead of $hash(value) + 0.5 \cdot i + 0.5 \cdot i^2$.
- What will happen if when deleting a value, we simply put null in its place? Consider the scenario in which words *a* and *b* have the same hash, we insert *a* and then *b*, and then delete *a*. While searching for b, we will encounter null and assume b is not in the table.

- Therefore, we need a way to flag a cell as deleted, so that when searching we know to continue our search. Solve this issue in a way which does not use additional space and uses constant time (both when deleting and later when searching). Note that in any case your solution should not restrict the class's functionality (by prohibiting insertion of a certain word for example). Explain your solution in the README.

Comparing Strings

As you know by now, there are two meanings for string equality:

- The two strings are two references to the exact same object in memory. This can be checked using $str1 == str2$. This is **usually** not what we want. Yet it is still legal to compare strings in this manner, and it can be used in situations where we're interested in knowing whether or not a given string reference is referencing a **specific** string object we have in mind, and we want to keep the actual contents out of the equation (literally). This comparision may even prove useful in this exercise.
- The strings may or may not reference to the same object in memory, but their contents are logically identical, which can be checked using $str1.equals(str2)$.

Needless to say, when we say that your data structure should not allow duplicates, we mean it in the **second** sense. The same goes for searching and deleting – the input string may be only logically identical to a string in the table that we wish to search/delete.
Unfortunately, it is **extremely** common to mean the second sense but accidently write the first one. If you're lucky, and have written your tests correctly, this will cause your program to malfunction. If you're unlucky (and/or written your tests incorrectly), the program will still behave as expected for your test cases. See appendix B for situations in which reference equality will still unfortunately work for your program and how to avoid these in order to comply with our testers.

Hash-table sizes

There are two popular choices for the sizes of hash tables: prime numbers and powers of two.
Prime numbers have an advantage as they tend to have much more uniform distributions of elements. However when using prime numbers and having to resize, one cannot simply multiply the size by 2, so a pre-calculated list of prime numbers in ascending order must help choose the new size. Moreover, a prime-sized table will actually not support a load factor of more than 0.5 when using quadratic probing.
Tables whose size is a power of two tend to have a less uniform distribution of random values, but:

- Are convenient for rehashing.
- Support a load factor of up to 1 when using quadratic probing and the aforementioned values of c_1, c_2.
- Can be more efficient in the hash function computation: see appendix A.

We'll be using powers of two.

4. Supplied Material

- SimpleSet.java: an interface consisting of the *add(), delete()*, *contains(),* and *size()* methods. See the API.
- Ex3Utils.java: contains a single static helper method: *file2array,* which returns the lines in a specified file as an array. See the API.
- data1.txt, data2.txt: will help you with the performance analysis.

- A skeleton for the RESULTS file which you'll use in the analysis section.
- A skeleton for the QUESTIONS file which you'll use in the theoretical questions section.

Download all these files here.

5. The classes You Should Implement

- **SimpleHashSet** – an abstract class implementing SimpleSet. **You may expand its API** (link at end of section) if you wish, keeping in mind the minimal API principal. You may implement methods from SimpleSet or keep them abstract as you see fit.
- **ChainedHashSet** – a hash-set based on chaining. Extends SimpleHashSet.
  Note regarding the API: the capacity of a chaining based hash-set is simply the number of buckets (the length of the array of lists).
- **OpenHashSet** – a hash-set based on open-hashing with quadratic probing. Extends SimpleHashSet.

In addition to implementing the methods in SimpleHashSet, these last two classes must have 3 constructors of a specified form, as seen in their API. Note that the full description of the empty constructors (that take no parameters) in the API specifies exact default values that should be used – not using them will cause your code to fail our tests.

- **CollectionFacadeSet** – as learned, a façade is a neat, or a compact API, wrapping a more complex API, less suitable to the task at hand. In this exercise, we'd like our set implementations to have a common type with java's sets, but without having to implement all of java's Set<String> interface which contains much more methods than we actually need. That's where SimpleSet comes in.
  The job of this class, which implements SimpleSet, is to wrap an object implementing java's Collection<String> interface, such as LinkedList<String>, TreeSet<String>, or HashSet<String>, with a class that has a common type with your own implementations for sets. This means the façade should contain a reference to some Collection<String>, and delegate calls to add/delete/contains/size to the Collection's respective methods. For example, calling the façade's 'add' will internally simply add the specified element to the Collection (if it's not already in it). In this manner java's collections are effectively interchangeable with your own sets whenever a SimpleSet is expected - this will make comparing their performances easier and more elegant.
  In addition to the methods from SimpleSet, it has a single constructor which receives the Collection to wrap (no need to clone the received collection, you can simply keep the reference).
  The API will help you to better understand the class's purpose.

See the API of all these classes.

- **SimpleSetPerformanceAnalyzer** – has a main method that measures the run-times requested in the "Performance Analysis" section. Implement it as you wish.
- Additional helper classes, if you like.

Note1: in this exercise we will be checking the performances of your implementations both automatically and manually. Avoid redundant operations such as needlessly probing your table. Avoiding redundant probings might tempt you to write almost-duplicate code segments. Don't – this will also be checked. Mind you, in many cases two code segments that are about the same can be merged into one **parameterized** segment.

Note2: Java recognizes floating numbers when the letter "f" is attached to them. For example, to

initialize a floating number with the value 0.65 we use: “float number = 0.65f;”.

6. Performance Analysis

You will compare the performances of the following data structures:

- ChainedHashSet
- OpenHashSet
- Java’s TreeSet[1]
- Java’s LinkedList
- Java’s HashSet

Since LinkedList<String>, TreeSet<String> and HashSet<String> can be wrapped by your ContainterFacadeSet class, these five data structures are effectively all implementing the SimpleSet interface, and can therefore be kept in a single array. After setting up the array, your code can be oblivious to the actual set implementation it's currently dealing with.

The file data1.txt (see ‘supplied material’) contains a list of 99,999 different words **with the same hash**. The file data2.txt contains a more natural mixture of different 99,999 words (that should be distributed more or less uniformly in your hash table).

Measure the time required to perform the following (instructions on how to measure will follow):

1. Adding all the words in data1.txt, one by one, to each of the data structures (with default initialization) in separate. This time should be displayed in milliseconds ($1ms = 10^{-3}s$).
2. The same for data2.txt. Again – in milliseconds.
3. For each data structure, perform *contains*(“hi”) when it’s initialized with data1.txt. Note that “hi” has a **different** hashCode than the words in data1.txt. **All ‘contains’ operations should be displayed in nanoseconds (** $1ns = 10^{-9}s$ **)**.
4. For each data structure, perform *contains*(“-13170890158”) when it’s initialized with data1.txt. “-13170890158” has the same hashCode as all the words in data1.txt.
5. For each data structure, perform *contains*(“23”) when it’s initialized with data2.txt. Note that “23” appears in data2.txt.
6. For each data structure, perform *contains*(“hi”) when it’s initialized with data2.txt. “hi” does not appear in data2.txt.

Some of these will take some time to compute. Do yourself a favor by printing some progress information (percentage etc.), but note that very intensive printing will slow the program down by an order of magnitude and ruin the comparison (most of the time might be spent on printing).
Fill in the results in the supplied RESULTS file. You will find instructions in the file itself.

In addition, organize the results in the README file in the following manner:

1. For data1.txt: the time it took to initialize each data structure with its words. *Mark* the fastest.
2. Same for data2.txt.
3. For each data structure: the time it took to initialize with the contents of data1.txt compared to the time it took to initialize with data2.txt.
4. Compare the different data structures for *contains*(”hi”) after data1.txt initialization. *Mark* the fastest.

[1] This is actually a “red-black tree”: a self-balancing binary tree

5. Compare the data structures for *contains*("-13170890158") after data1.txt initialization. *Mark* the fastest.
6. For each data structure initialized with data1.txt, compare the time it took for the query *contains*("hi") as opposed to "-13170890158".
7. Repeat 4-6 for data2.txt and "hi" vs "23".

Notes:

- You can use *System.nanoTime()* to compute time differences:

```
long timeBefore = System.nanoTime();

/*  ... some operations we wish to time */

// time difference in nanoseconds:

long difference = System.nanoTime() – timeBefore;
```

- In order to display time in milliseconds, divide the difference by 1,000,000.
- System.nanoTime() is one of the most precise methods in Java to measure time. Even so – extremely short operations are never measured reliably with any measuring technique. A popular approximation workaround is to perform the operation iteratively at least some longer period of time (a second, for instance), and then divide the exact elapsed time (which is not necessarily the minimal time threshold you set) by the number of iterations.
- Use the supplied *Ex3Utils.file2array(fileName)* to get an array containing the words in a file. The method expects either the full path of the input file or a path relative to the project's directory.

7. Theoretical Questions

The supplied QUESTIONS file contains questions that are meant to check your basic understanding of the corresponding video lectures. Answer them in-file and include the QUESTIONS file in your submission. The main purpose of this is to serve as an indication for students who must revise the material. The questions are basic and few - answering them successfully is by no means a guarantee that your understanding is thorough.

- You are encouraged to leave comments in the file (using '#') that explain your answers: In case of a wrong answer, you'll be able to appeal, refer the grader to the corresponding textual explanation, and gain points back.
- The pre-submission script will tell you if the QUESTIONS file is missing or if its format is incorrect, but will not correct any mistakes.

8. Submission

8.1 README

Include the following in your README:

1. Description of any java files not mentioned in this document. The description should include the purpose of each class and its main methods.
2. How you implemented ChainedHashSet's table.
3. How you implemented the deletion mechanism in OpenHashSet.
4. The results of the analysis.

5. Discuss the results of the analysis in depth.
    - Account, in separate, for ChainedHashSet's and OpenHashSet'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?
    - What results surprised you and which did you expect?
    - Did you find java's HashSet performance on data1.txt surprising? Can you explain it? Can google? (no penalty if you leave this empty)
    - If you implemented the modulo efficiently (appendix A), how significant was the speed-up?

8.2 Compiling without warnings

We have seen before that compiling Java code with the "javac" command might return errors that we need to fix before we can run our program. Javac can also return warnings, which indicate that our code is probably problematic for a host of various reasons (but the code is still runnable).

Eclipse also presents warnings to the user (yellow lines under problematic code). The warnings given by eclipse and by javac are not the same since Eclipse does not use javac for compilation.

As of this exercise, we will check that your code does not produce specific warnings (When compiled with javac). In case that your code issues a warning, it will appear in the pre-submission script with an explanation. You have to fix the code that produces a warning. Any submission that produces a warning will get an automatic -5 points reduce.

All the warnings we will look for are also reported by Eclipse. If you want to check for them manually, you can use the following customized compilation command:

javac –Xlint: rawtypes –Xlint:static –Xlint:empty –Xlint:divzero –Xlint:deprecation file1.java file2.java …

"-Xlint:static" for example is a flag that indicates that the "static" warning should be issued. There are 5 different flags we used, each enables a different warning. You can read about these 5 different warnings in the following page:
http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html
Note: Copy-pasting the command line above to the shell might result in some invalid characters.

8.3 Automatic Testing

The tester is a small interpreter, while the different tests are its scripts: each test creates a hash-set and performs some operations on it. All tests are performed on each of the two implementations in separate. As in previous exercises, you're only given some of the tests.
To run the tests in the CS computer labs, put all the required files (next section), including the README RESULTS and QUESTIONS files, in a jar and type:

```
~oop/bin/ex3  <jar file>
```

The online-tests this command runs are available here.
Their format is described in appendix B.

8.4 What to submit

- SimpleHashSet.java
- ChainedHashSet.java
- OpenHashSet.java
- CollectionFacadeSet.java
- SimpleSetPerformanceAnalyzer.java
- Any additional javas needed to compile your program
- README
- RESULTS
- QUESTIONS

Don't forget to Javadoc your code. **Javadoc should compile correctly**.

Enjoy!

Appendices

A. Efficient modulo operations

One of the benefits of using hash tables of sizes which are powers of 2, is that computing the remainder of a division by a power of two can be done much more efficiently than by a number which is not a power of two, since numbers are represented in memory in binary.
A number modulo 2^k is actually the number's last k bits, since dividing by 2^k is simply discarding these bits (similar to the way we integer-divide by ten by discarding the last digit when dealing with decimal numbers).
Therefore, a bitwise operation can be performed in this case in place of an arithmetic modulo. Specifically, an 'and'[2] operation with k ones will give a number's k last digits. K ones is the largest binary number with k digits, which means it's one less than the smallest binary number with k+1 digits which is 2^k (it's equivalent to k nines in decimal, which is one less than 10^k).
We get:

$$num\%(2^k) == num\&(2^k - 1)$$

Though these expressions are effectively the same when dealing with non-negatives, the bitwise operation is typically much faster than the arithmetic equivalent, resulting in non-negligible increase in performance when one repeatedly performs this operation. While this kind of optimizations is commonly taken care of by the compiler in many cases, in this case the compiler has no way of knowing in compile-time it's dealing with powers of two, and it has to allow for a general modulo operation.
Perhaps even more importantly, using & allows us to lose the Math.abs since the bitwise-and will always return a value in the range $0 \ldots 2^k - 1$ (a valid index).

You are therefore not required, but are very encouraged, to use bitwise-and in place of abs+modulo and see how it affects your running time. Mind you: while we're in the realm of nano-optimizations, $num \,\&\, (tableSize - 1)$ takes more than twice the time of $num \,\&\, tableSizeMinusOne$ (why?).

Note: the binary operators $\ll = 1$ and $\gg = 1$ which multiply and divide by two respectively, are similarly more efficient than their arithmetic equivalents $*= 2$ and $/ = 2$, but by a much smaller margin, since the arithmetic multiplication and division operations are much faster than modulo to begin with. Moreover, they are **much** rarer than modulo in the implementation of a hash table, and lastly, if 2 is a literal or saved in a final variable, this optimization will probably be automatically performed by the compiler in any case.

[2] A bitwise-and between two operands yields a number whose i'th digit (from the right), in binary, is 1 iff both of the operands have 1 in the i'th digit of their binary form. For example, when writing in binary form: 11011 and 111 should yield 11 since they both have 1's only in the last two digits. 11011 in decimal is 27, 111 is 7, and 11 is 3. In java, bitwise-and can be performed with '&'. This means that 27&7==3.

B. Testing for Correct String Comparison

1) As mentioned, it is completely legitimate to compare strings using *str*1 == *str*2 if that is what we actually mean. If we actually meant to compare contents, it will obviously be a bug. However, there are two cases in which the application will seem to work correctly even though strings are compared in this manner. Carefully consider the following code which contains a simple method, and an extremely simple tester that checks part of its functionality and is intended to return true if the method is correct:

```
private boolean isInWordList(String inputStr) {
	for(String word : wordsList) {
		if(inputStr == word)
			return true;
	}
	return false;
}

public boolean testIsInWordList() {
	String word = "someWord";
	wordsList.add(word);
	return isInWordList(word);
}
```

Note that the first method contains the aforementioned bug. The tester however will succeed because we add and look for the same string object.

2) Consider the same buggy *isInWordList* method, with one of the following testers:

```
public boolean testIsInWordList() {
	wordsList.add("someWord");
	return isInWordList("someWord");
}
public boolean testIsInWordList() {
	String word = "someWord";
	wordsList.add(word);
	return isInWordList("someWord");
}
```

Seemingly, these testers don't use the same exact reference so they should fail (which is good since the method they check is incorrect). Depending on the compiler and its optimizations, however, this test might still pass.
The compiler, which is unaware of our intentions, can see that these testers use the same string contents twice. It **might** therefore only create one string object, and have two lines reference to it – this would make these testers effectively the same as the first one.

In order to enforce two different string objects in your tester (that would correctly check that your code is comparing using equals) you can explicitly create separate Strings. The following tester will always fail for the given method (which is good):

```
public boolean testIsInWordList() {
	wordsList.add(new String("someWord"));
	return isInWordList(new String("someWord"));
}
```

C. Tests Format

In this appendix we describe the format of the automated tests, in order to help you in your debugging.

As mentioned, each test is actually a simple script of operations to perform on your hash-sets.

- The first line starts with a '#' sign, and is a comment describing the test (i.e., this line is ignored).
- The second line creates a hash-set object using one of the three constructors:
  - An empty line calls the default constructor.
  - A line starting with "*N:*" (N for "numbers") followed by three integers (e.g., *N: 0.8 0.1*) calls the constructor with an upper and lower load factors.
  - A line starting with "A*:*" (A for "array") followed by a sequence of words (e.g., *A: Welcome to ex3*) calls the data constructor that initializes a new instance with the given data.

- The following lines call methods of the created set. They can be one of the following:
- *add some_string return_value*
  Verifies that add's return value for some_string matches *return_value* (either *true* or *false*).
- *delete somes_string return_value*
  likewise.
- *contains some_string return_value*
  likewise.
- *size return_value*
  Verifies that the size's return value is *return_value* (a non-negative integer).
- *capacity return_value*
  Verifies the capacity.

For example:

```
# calling the constructor with the value 20, adding "Hi", adding "Hello", deleting "Hi", and then searching for "Bye".
N: 0.8 0.1
add Hi true
add Hello true
delete Hi true
contains Bye false
```

The first line is a comment describing the test. The second line starts with "N:" which means calling the constructor that gets the upper and lower load factors (0.8 and 0.1 respectively in this case). The third line calls *myHash.add("Hi"),* and then verifies that the command succeeded. The fifth line calls

myHash.delete("Hi"), and verifies that it succeeded. The last line calls *myHash.contains("Bye")* and verifies that "Bye" was not found.