Exercise 4 – HashSet \ OOP 2014

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: **7.5.14 , 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 data structure (not necessarily abstract in Java's sense, but in the sense that it is implementation-independent) which represents an unordered collection 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) == false* (where 'equals' may be the inherited method from Object, or better, 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 Set implementation that provides constant-time ($O(1)$) implementations for 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 "equal" objects have the same value. Although this is its only requirement, it is also desirable that this value is distributed evenly among the possible numeric values if the hash-set is to have its 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 (more later about what happens if the cell is taken). If, during the insertion, another element in the table is discovered which is 'equal' to the inserted one, the set remains the same.
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. Modulo is a good start, but in java (-3)%7 is -3. Use $Math.abs(hashCode(value))\%tableSize$ (and see appendix A).

Performance considerations and re-hashing

The performance of a hash table instance 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

simply means to insert all elements from the "old" table in the new one.
Note: Testing the load factor and deciding to increase is only done when add is called. Decreasing – only in the scope of a delete operation.

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 fitting to the legal index range in the array using modulo and abs).
Note: In Java, you'd probably want to declare an array of linked list such as "LinkedList<String>[]", but for reasons we will not discuss here, constructing such an array will not compile. Here are some possible solutions:

- Implement a linked list of strings yourself (highly unrecommended).
- Use an ArrayList<LinkedList<String>> instead of a standard array
- Wrap LinkedList<String> in an inner class, and have an array of that class instead.
- Have an array of CollectionFacadeSet (see "the classes you should implement"), or of an extending class, wrapping 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.

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

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

(which is then fitted to the appropriate range [0:tableSize-1]). 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 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 troubles: use $hash(value) + (i + i^2)/2$ which is always a whole number (why?).
- 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. Implement this as you wish and explain your choice in the README.

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.
- Ex4Utils.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.

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.

- **CollectionFacadeSet** – a façade[1] 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 implementation 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 pass 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 it).
  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.

[1] French for 'front', or 'face'. Pronounced like 'fasad'.

6. Performance Analysis

You'll compare the performances of the following data structures:

- ChainedHashSet
- OpenHashSet
- Java's TreeSet[2]
- 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, in miliseconds, required for performing the following (instructions will follow):

1. Adding all the words in data1.txt, one by one, to each of the data structures (with default initialization) in separate.
2. The same for data2.txt.
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.
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).

Organize the results in the following manners:

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.
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".

Some of the results are astonishing.

[2] This is actually a "red-black tree": a self-balancing binary tree

Notes:

- You can use *Date.getTime()* to compute time differences:

```
long timeBefore = new Date().getTime();

//  … some operations that we wish to time

long timeAfter = new Date().getTime();

// time difference in miliseconds

long difference = timeAfter – timeBefore;
```

- Use the supplied *Ex4Utils.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. Submission

7.1 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 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?
    - -1 points if you didn't list java's HashSet performance on data1.txt as 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? And how awesome is that?

7.2 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 implementations in separate. As in previous exercises, you're only given some of the tests.

To run the tests when in CS computer labs, put all the required files (next section), including the README, in a jar and type:

```
~oop/bin/ex4  <jar file>
```

The online-tests this command runs are available here.

Their format is described in appendix B.

7.3 What to submit

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

Don't forget to Javadoc your code.

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' operation with k ones will give a number's k last digits.
Therefore,

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

Though they are effectively the same when dealing with non-negatives, the bitwise operation is typically much faster than the arithmetic equivalent, resulting in better performances when one repeatedly performs this operation. While this kind of optimizations is commonly taken care of by the compiler in some cases, in this case the compiler has no way of knowing in compile-time he's dealing with powers of two, and it has to allow for a general modulo operation.
Maybe even more importantly, using & allows us to lose the Math.abs since it will always return a valid index.

You are therefore not required, but are very encouraged, to use bitwise-and in place of modulo of a power of 2 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 operators $\ll= 1$ and $\gg= 1$ 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 be automatically performed by the compiler in any case.

B. 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 ex4*) 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.