Object Oriented Programming - Exercise 4: Hashset

Contents

1 Objectives

1. Improve understanding of hash tables by getting your hands dirty: you will write two implementations of a HashSet data structure

2. Measuring actual running times by comparing performances of five data structures: the aforementioned two, java.util.LinkedList, java.util.TreeSet, and java.util.HashSet

3. Get familiarized with real-life technical and theoretical considerations that should be taken into account when going from theory to practice

2 General Notes

- You are encouraged to go over the class and TA material regarding hash tables again

- Please read the entire document before starting to work on the exercise. Specifically, there is a section with important implementation details

- You are allowed to, and should, use the following classes:
java.util.TreeSet, java.util.LinkedList, java.util.HashSet (the last can't be used in your own implementation obviously, only in the comparison part). You can also use classes and interfaces from the java.lang package

- Writing tests to your code is not mandatory in this exercise. However we highly recommend you to do so

3 Supplied Material

- **SimpleSet.java**: an interface consisting of the add(), delete(), contains(), and size() methods

- **Ex4Utils.java**: contains a single static helper method: file2array, which returns the lines in a specified file as an array

- **data1.txt**, **data2.txt**: files that will help you with the performance analysis (more details in the corresponding section)

- A skeleton for the RESULTS file which you'll use in the analysis section

- Pay attention that the school solution is not given to you this time

4 The Classes You Should Implement

Note: the API for all classes you should implement is available here. Please read it thoroughly.

- **SimpleHashSet** – an abstract class implementing SimpleSet. You may expand its API if you wish, keeping in mind the minimal API principal. You may implement methods from SimpleSet or keep them abstract as you see fit

- **OpenHashSet*** – a hash-set based on chaining. Extends SimpleHashSet. Note: the capacity of a chaining based hash-set is simply the number of buckets (the length of the array of lists)

- **ClosedHashSet*** – a hash-set based on closed-hashing with quadratic probing. Extends SimpleHashSet

  *Note: In addition to implementing the methods in SimpleHashSet, the classes OpenHashSet and ClosedHashSet must have additional constructors of a specified form, as seen in their API. The full description of the empty constructors in the API specifies exact default values that should be used – not using them will cause your code to fail our tests. The documentation also specifies behaviors for invalid parameters.

- **CollectionFacadeSet** – Recall, in the Facade design pattern, a facade is an object that provides a simplified interface to a more complex class or even a set of classes. 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 facade should contain a reference to some Collection<String>, and delegate calls to add/delete/-contains/size of the Collection's respective methods. For example, calling the facade's 'add' will internally simply add the specified element to the Collection (only if you're certain 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). More details in the class' API. Be sure to read the constructor's documentation.

  Note: This class represents a set. As such, it must appear to always behave like one (e.g. adding an element twice and then removing it once leaves none in the set) - but this says nothing about the internal state of the collection. We do, however, for the sake of memory and performance, recommend you do keep the inner collection without duplicates. There are various ways of doing this, from the most naive to more sophisticated ones. You are encouraged to use original and efficient mechanisms, but whatever you do you **must not** use reflections or instanceof, or in any other way specify in this class the name of a concrete kind of collection.

- **SimpleSetPerformanceAnalyzer** – has a main method that measures the run-times requested in the "Performance Analysis" section. This class must be written in a way which permits to easily disable or enable any subset of the tests described in section 6. This will greatly help you test it, and will also be checked by us.

- Any additional helper classes you may need.

5 Implementation details

You are going to implement two variants of hash sets. Both will use hash tables, as we've seen in class, and each will use a different approach to handle collisions: open- and closed-hashing.

5.1 Elements and Hash function

In this exercise we are only going to deal with String elements and the default hash function for Strings implemented at String.hashCode(). Remember that when comparing elements (for example, when looking for an item) we want to compare by **value**, and not by reference!

Note that String's hashCode method may, and will, return negative values or values that are greater than our table size (look here for implementation details of String.hashCode). Therefore we should wrap our cell value to the valid range. More details about that down below.

5.2 Resizing the table

From time to time, you will have to resize your table, depending of the load factor. Don't forget, when resizing the table we have to insert all existing elements by **re-hashing** them (remember why). Another few points you should think of when resizing the table:

- Do we need to check for duplicates when re-hashing elements into the new table? Should we care?
- Testing the load factor and deciding to increase the table size is only done when add is called. Decreasing – only when delete is called. The table capacity can decrease to as low as 1
- The upper and lower load factors are inclusive – they both represent a valid state. However, consider an upper load factor of 1 – while valid, it is impossible for a closed-hashing hash-table to surpass it. Therefore when adding elements you should always make sure you never go beyond the upper load factor, even momentarily
- Resize occurs at most **once** per add/delete operation. It implies that there could be a scenario in which we **remove** an element, decrease the table size, but the load factor would still be smaller than the lower bound – this is fine (try to think of such a case)

5.3 Open hashing

You are required to implement the open-hashing set using linked lists. You might be tempted to create an array of linked lists of Strings like that:

```
LinkedList<String>[] arr = new LinkedList<String>[16];
```

However, such arrays are illegal in Java (for reasons we won't discuss here). Here are some possible alternatives:

- 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

- Extend CollectionFacadeSet and have an array of this subclass where each such facade wraps a LinkedList<String>

You can also think of a solution of your own, as long as encapsulation and a reasonable design are held. Explain your choice in the README. However, we ask that for the purpose of this exercise 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.

5.4 Closed hashing

Remember that in closed hashing we need to use a probing function for collision resolution. In this exercise we'll use quadratic probing: the i'th attempt to find an empty cell for element e, or to simply search for e, will use the value

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

where c_1 and c_2 are constants. This value 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 e. A special property of tables whose size is a power of two, is that values of $c_1 = \frac{1}{2}, c_2 = \frac{1}{2}$ ensure that as long as the table is not full, a place for a new value will be found during the first *capacity* attempts. Therefore we'll use these values, and always keep the table size a power of 2.
Implementation issues:

- Floating point arithmetic can cause numeric issues that lead to bugs. Therefore, instead of computing the probing value with $hash(e) + \frac{1}{2}i + \frac{1}{2}i^2$, use

  $$hash(e) + (i + i^2)/2$$

  which is always a whole number

- What will happen if when deleting a value, we simply put null in its place? Consider the scenario in which the Strings 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. We therefore 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 uses only a constant addition in space and time (both when deleting and later when searching). Note that in any case your solution should not restrict the class' functionality (by prohibiting insertion of a certain String for example). Explain your solution in the README

5.5 Clamping an expression to a valid index

Whatever kind of table we use, at some stage we'd like to clamp an expression to the valid range of table indices. In the case of open hashing, this expression will simply be $hash(e)$, and in the case of closed hashing, it'll be $hash(e) + (i + i^2)/2$ for some i. In either case, the expression might be either negative or larger than $tableSize - 1$. The modulo operator in Java will output negative results for negative numbers, e.g. in Java $(-3 \ \% \ 7) == -3$. To overcome this we could use Math.abs(), but this introduces more numerical issues we should address (and won't discuss here). Instead, we will use a much simpler solution to compute the mathematical *modulo*

operator, and not Java's %, using bitwise-AND (you can read more about Bitwise operations and bitwise in Java). For $tableSize$ that is a power of 2, we have the special property that $(n \,\&\, (tableSize - 1)) == (n \mod tableSize)$, Where & is the bitwise-AND operator in Java.
Bottom line:
For open-hashing use the index

$$hash(e) \,\&\, (tableSize - 1)$$

For closed-hashing use the index

$$\big(hash(e) + (i + i^2)/2\big) \,\&\, (tableSize - 1)$$

6 Performance Analysis

You will compare the performances of the following data structures:

1. OpenHashSet
2. ClosedHashSet
3. Java's TreeSet
4. Java's LinkedList
5. Java's HashSet

Since LinkedList<String>, TreeSet<String> and HashSet<String> can be wrapped by your CollectionFacadeSet 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 measured 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 measured in nanoseconds** ($1ns = 10^{-9}s$)
4. For each data structure, perform contains("-13170890158") when it is initialized with the words in data1.txt. Note that "-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 become a performance bottleneck (the CPU will spend most of its time printing) and ruin the comparison. Fill in the results in the supplied RESULTS file. You will find instructions in the file itself.

Notes:

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

```
long timeBefore = System.nanoTime();
/* ... some operations we wish to time */
long difference = System.nanoTime() - timeBefore;
```

- difference is now in nanoseconds. Divide it by 1,000,000 to translate it to milliseconds
- System.nanoTime() is one of the most precise methods in Java to measure time. Even so – extremely short operations (e.g. contains) are never measured reliably with any measuring technique. A popular approximation workaround is to measure the time it takes to perform the operation iteratively a large number of times, and then divide that measurement by the number of iterations
- When the JVM (Java Virtual Machine - which interprets the bytecode at runtime) identifies a piece of bytecode as performance-critical (for instance, when it's executed repeatedly), it gradually replaces pieces of bytecode with their compiled and optimized machine-code. This compilation takes a little time, but is assumed to pay off. Measuring the performance of a short action x should reflect its performance once all compilations and optimizations have already been applied. In order to do so you are required to "warm-up" the JVM - execute your target operation enough times so that the JVM will have had time to run and replace the bytecode with compiled machine-code. How much warm-up is enough? The answer depends on the bytecode. Basically, when your measured time on some action x, as a function of the number of iterations in the warm-up preceding the measurement, hits a plateau - it means the JVM had time to optimize x as best it could prior to the measurement phase
- For simplicity, we will use a uniform warm-up time. Use 70000 iterations as the warm-up before any contains operation, for every data structure except LinkedList (explanation ahead). In other words, for each data structure, excluding LinkedList, and for every contains operation, call the relevant contains method 70000 times without measuring, then measure the time it takes to do another 70000 iterations, and divide that by 70000. This will be your approximation for that operation
- As for LinkedList - you'll find that its contains operation takes much longer, and a warm-up makes a less of a difference for long operations. It also requires less iterations to measure

accurately, so simply measure the time it takes to perform 7000 contains, and divide by the number of iterations

7 Submission

- Submission deadline: **May 16, 2019**
- Submit a single JAR file named ex4.jar with the following files: SimpleHashSet.java, OpenHashSet.java, ClosedHashSet.java, CollectionFacadeSet.java, SimpleSetPerformanceAnalyzer.java, README, RESULTS.
  In addition, include in your JAR any source files needed to compile your program.

In you README, discuss the following:

- How you implemented OpenHashSet's table
- How you implemented the deletion mechanism in ClosedHashSet
- Discuss the results of the analysis in depth:
  - Account, in separate, for OpenHashSet's and ClosedHashSet's bad results for data1.txt
  - Summarize the strengths and weaknesses of each of the data structures as reflected by the results. Which would you use for which purposes?
  - How did your two implementations compare between themselves?
  - How did your implementations compare to Java's built in HashSet?
  - Not mandatory: Did you find java's HashSet performance on data1.txt surprising? Can you explain it?
- Any additional information about your design you'd like to specify

Good Luck !