Object Oriented Programming

Exercise 3: HashSet

Contents

1. Goals

1) Improve understanding of hash tables by getting your hands dirty: you will write two implementations of the 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

- Submission deadline: **3.5.17 , 23:55**
- You can use the following classes, and **only them**: java.util.TreeSet, java.util.LinkedList, java.util.HashSet (the last can't be used in your own implementation obviously, only in the comparison) and classes and interfaces from the java.lang package in standard java 1.8 distributions.
- In this exercise you should Javadoc your code and it should compile correctly.

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.
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 ($\Theta(1)$) 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. 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's valid indices, 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, hence the 'new' element is not inserted!).
In this exercise we will only deal with sets of Strings.
Note that String's hashCode method may, and will, return negative values, or values that are greater than the table size (more details about the calculation here). We will later discuss how to tune this value to the desired range.

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 into the new one.

Notes:

- Decide if there is a need to check for duplicates when re-hashing elements into the new table.
- 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.
- 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 (a reminder follows) to surpass it. Therefore when **adding** elements you should always make sure you never go beyond the upper load factor, even momentarily.
- However, a change in the table's size (and a re-hashing) should occur at most once per add/remove. You might therefore encounter situations where this means that during an element **removal** the actual load factor, after re-hashing, will still be below its lower limit. That's fine.

Open 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 being fitted to the legal index range, which will be discussed shortly).

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 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 (see "the classes you should implement"), and have an array of this subclass where each such façade 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.

Closed 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 search for $value$, will use the index

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

where c_1 and c_2 are constants. This index 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 (see appendix C) that as long as the table is not full, a place for a new value will be found during the first *capacity* attempts, **so those are the values we'll use**.

Implementation issues:

- Rarely does using floating-point variables doesn't entail numeric issues and bugs – avoid them when possible. Use $hash(value) + (i + i^2)/2$ which is always a whole number (can you see why?) instead of $hash(value) + 0.5 \cdot i + 0.5 \cdot i^2$. Will $i/2 \ + i^2/2$ also work as an alternative?

- 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 $\Theta(1)$ additional space and 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 String for example). **Explain your solution in the README.**

Comparing Strings

In Java, there are two meanings for String equality:

1) 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 comparison may even prove useful in this exercise.
2) The Strings may or may not reference the same object in memory, but their contents are logically identical, which can be checked using $str1.equals(str2)$.
3) Objects.equals(str1, str2) this function compare two strings, but verify 'null' value first.

Needless to say, when we say that your data structure should not allow duplicates, we mean it in the third 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 for or delete.
Unfortunately, it is extremely common to mean the second or third senses but accidently write the first one. See appendix B for situations in which reference equality will still unfortunately work in your tests and how to try and 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.
On the one hand, prime numbers tend to have much more uniform distributions of elements. However when sticking with 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:

1) Are convenient for rehashing.
2) Support a load factor of up to 1 when using quadratic probing and the aforementioned values of c_1, c_2 (see appendix C).
3) Can be more efficient when mapping an object to an index: see appendix A.

We'll be using powers of two.

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 $hash(value)$, and in the case of closed hashing, it'll be $hash(value) + c_1 \cdot i + c_2 \cdot i^2$ for some i. In either case, the expression might be

either negative or larger than *tableSize*-1.
Modulo is a nice start, but in java $(-3)\%7$ is still -3 (while mathematically, $(-3)\ mod\ 7 = 4$). This is easily solvable using the hashCode's absolute value, so we could use: $Math.abs(expression)\%tableSize$.
Note: There's an edge case when *expression* equals $Integer.MIN\_VALUE$, which is -2^{31}. Since there's no corresponding positive value as $Integer.MAX\_VALUE$ is only $2^{31}-1$, this is the only case when Math.abs returns a negative value unchanged. Luckily, -2^{31} is a multiply of *tableSize* which means $Integer.MIN\_VALUE\%tableSize == 0$ so there's no necessity to address this case (yet).

Is that it? Not quite. While this will do for the expression in the open hashing case, in the case of closed hashing we rely on the fact that if $N = 2^k$ and $c_1 = c_2 = \frac{1}{2}$, we'll find an empty slot in the table (if one exists) during the first N attempts (proof in appendix C). If you'll revise the proof, you'll see it proves nothing about clamping using $Math.abs(hash(value) + c_1 \cdot i + c_2 \cdot i^2)\%tableSize$. It does, however, work for:

$$(Math.abs(hash(value)) + c_1 \cdot i + c_2 \cdot i^2)\%tableSize$$

Unfortunately if *value* is $Integer.MIN\_VALUE$ and *i* isn't large enough, the expression $Math.abs(hash(value)) + c_1 \cdot i + c_2 \cdot i^2$ will be negative, and one which is not a multiply of *tableSize*. This would cause '%' to return a negative value, which in turn will crash the program. To solve this, we can cast $hash(value)$ to a *long* variable (which is 8 bytes and not 4, resulting in a bigger range which does include 2^{31}). Subsequently, the *long* overload of Math.abs will be called instead of the *int* overload, and since there's a corresponding positive value for the negative value -2^{31} when dealing with *long*s, Math.abs will return a non-negative.

Lastly, what if i is very big and i^2 is bigger than $Integer.MAX\_VALUE$ (i.e. causing the *int* holding the expression to *overflow*)?
We can again cast *i* to a *long* variable, and that would be enough range in this exercise. Now that the expression $Math.abs((long)hash(value)) + c_1 \cdot i + c_2 \cdot (long)i \cdot i$ is a valid non-negative *long*, we can apply '$\%tableSize$' which will reduce it back to a value containable in an *int*.

But there's also a much simpler solution to all this. For a *tableSize* which is a power of two, it holds that:

$$expression\ \&\ (tableSize - 1) == expression\ mod\ tableSize$$

where "mod" is the mathematical operator, not Java's '%'. It holds even if *expression* is the result of an overflowing operation, and as opposed to "$expression\ \%\ tableSize$" - even if *expression* is negative. This bypasses the need for Math.abs, the need for '%', and the need for a *long variable*. The equivalence to true arithmetic modulo also ensures the proof in appendix C holds, and as an extra bonus, this method can also potentially improve performance significantly. See **appendix A** for an explanation of what '&' is and why it works.

Bottom line

In the case of open hashing, either use

$$int\ index = (int)(Math.abs(hash(value))\%tableSize)$$

Or:

$$int\ index = hash(value)\&(tableSize - 1)$$

In the case of closed hashing, either use

$$int\ index = (int)((Math.abs((long)hash(value)) + (i + (long)i * i)/2)\%tableSize)$$

Or:

$$int\ index = (hash(value) + (i + i * i)/2)\&(tableSize - 1)$$

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: 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.
- Automatic test files (90% of them, as always). More details in section 7.3.
- Pay attention that the school solution is not given to you this time.

5. The classes You Should Implement

1) **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.
2) **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).
3) **ClosedHashSet** – a hash-set based on closed-hashing with quadratic probing. Extends SimpleHashSet.

Note: 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 (link here). 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. The documentation also specifies behaviors for invalid parameters.

4) **CollectionFacadeSet** – as learned - a façade is a neat, or a compact API, wrapping a more complex one, less suitable to the task at hand. You also learned it can additionally be used to give an existing class a new "front", which complies with another API. 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 (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, here. 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 naïve 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.

Note: See the API of all these classes. Excluding the case of SimpleHashSet, **you may not change the API**.

5) **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.

6) Additional helper classes, if you like.

Additional notes:

1) 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 probes might tempt you to write almost-duplicate code segments. Don't – this will also be checked. Note that two code segments that are about the same can often be merged into one **parameterized** segment.
2) When you write "0.65" in Java the expression's type is *double*. To make its type *float* where one is expected, Java allows us to append an '*f* ': " *float number = 0.65f;* " .

6. Performance Analysis

You will compare the performances of the following data structures:

1) OpenHashSet
2) ClosedHashSet
3) Java's TreeSet[1]
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

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

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) separately. This time should be stored 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 stored 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 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.

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.
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:

- 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.
- 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;
```

- *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 (*contains*, for example) 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.
- Bonus (1pt.): For what *contains* operations and for which data structure did the warm-up make the most difference in your implementation? How much time did its warm-up take, and how much did the actual measurement? Answer in your README under the title BONUS.

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 OpenHashSet's table.
3. How you implemented the deletion mechanism in ClosedHashSet.
4. Bonus answers, if you chose to answer them
5. The results of the analysis.
6. 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?
- 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 don't Google it and leave this empty)
- If you tried clamping expressions to valid indices in more than one way, what were they and how significant was the speed-up?

7.2 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). These warnings are also presented in IDEA in the messages tool window.

In this exercise, we will check that your code does not produce specific warnings. 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.**

Most of the warnings we will look for are also reported automatically by IDEA. In order to have all of the warnings displayed when you compile your code in IDEA, go to File -> Settings -> Build, Execution, Deployment -> Java Compiler and enter the following line under "Additional command line parameters":

```
-Xlint:rawtypes –Xlint:static –Xlint:empty –Xlint:divzero –Xlint:deprecation
```

If you want to check for them manually when you compile your code using the shell, 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.

7.3 Automatic Testing

Our 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 separately. 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 and the RESULTS file, 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 D.

7.4 What to submit

1) SimpleHashSet.java
2) OpenHashSet.java
3) ClosedHashSet.java
4) CollectionFacadeSet.java
5) SimpleSetPerformanceAnalyzer.java
6) Any additional javas needed to compile your program
7) README
8) RESULTS

Good luck!

Appendices

Appendix 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 (modulo) by a power of two can be done very easily and efficiently.
Note that in decimal, the rightmost k digits of a (non-negative) number *X* equal $X \; mod \; 10^k$.
Similarly, in binary representation, the rightmost k digits of *X* (the rightmost k **bits**), are the binary representation of $X \; mod \; 2^k$. When a number overflows, its leftmost bits are discarded, but its rightmost bits remain intact – so overflowing or not, as long as it's non-negative (we'll show the same for negatives in a moment), we're simply interested in an expression's k rightmost bits in order to get its " $mod \; 2^k$ " and clamp it to a table of size 2^k .

The bitwise 'and' (&) operator works on two operands as follows: **a bit in the result will be 1 if and only if both of the corresponding bits in the operands are 1**. Therefore, in order to get the 3 rightmost bits of *X*, we could perform a bitwise-and with the number whose binary representation is 111 – which is, in decimal, the number 7. Let's take 35&7 for example. 35 in binary is 100011, and 7 is 000111, so the result in binary is 011, meaning 35&7==3 (3 is also, you'll notice, $35 \; mod \; 8$).
All that's left is getting the number whose binary is k subsequent 1-s (רצף של k אחדות). Note that this is the largest number with k bits. In decimal, the largest number with k digits is k subsequent 9-s (רצף של K תשיעיות), which is one less than 10^k . Similarly, we can get the number we want by subtracting 1 from 2^k .
In other words, for every non-negative number *num*:

$$(*) \qquad num \; mod \; 2^k == num\&(2^k - 1)$$

What about negative values? Because of the way they are represented in memory (you can read about *Two's Complement*), the k rightmost bits of a negative int num are also the k rightmost bits of $2^{32} + num$, which is necessarily **positive**. Therefore, for any $k < 32$, the k rightmost bits of num are:

$$num\&(2^k - 1) = (2^{32} + num)\&(2^k - 1) = (2^{32} + num) mod \; 2^k = num \; mod \; 2^k$$

So $(*)$ holds for any $num \in Z$.
Mathematical mod is also what the proof in appendix C uses so by using the k rightmost bits we guarantee the correctness of our quadratic probing.

Additionally, the bitwise operation '&' is typically much faster than the arithmetic operation '%', resulting in an increase in performance when one repeatedly performs it. While this kind of optimizations is commonly taken care of by the Java compiler or the JIT compiler, we can't hope to assume it'll know it's dealing with powers of two, so when encountering '%' it'll have to allow for a general modulo operation.
More importantly, using '&' allows us to lose the Math.abs and the castings to *long.*

Two final notes while we're in the realm of nano-optimizations:
Note 1: $num \; \& \; (tableSize - 1)$ is more than double the time of $num \; \& \; tableSizeMinusOne$ (why?).
Note 2: the binary operators $\ll = 1$ and $\gg = 1$ which multiply and divide by two respectively (more details here), are similarly more efficient than their arithmetic equivalents $*= 2$ and $/ = 2$, but by a

much smaller margin - these arithmetic operations are much faster than modulo to begin with. More importantly, they are **much** rarer than modulo in the implementation of a hash table. Lastly, if 2 is a literal or saved in a final variable, the compiler has a chance of making this optimization in any case.

Appendix B: Testing for Correct String Comparison

1) As mentioned, it is legitimate to compare strings using $str1 == str2$ if that is what we actually mean. If we mean 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 bugless:

```
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: it first adds an object to the list, and then checks (via isInWordList) if this same exact object indeed exists in the list.

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

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

Seemingly, both these testers don't use the same exact reference when calling $isInWordList$ so they should fail (which is good since the method they check is incorrect). Depending on the compiler and its optimizations, however, both of these tests might still pass.
The compiler and in Java's case the JVM as well, which are unaware of our intentions, can see that these testers use the same string contents twice. They 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 try 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 most likely fail for the given method (which is good):

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

Is this guaranteed to work, always? No. Eventually, a compiler/JVM which is smart enough to see we're trying to fool it (but not smart enough to understand why) might still optimize this tester by making it equivalent to the first, incorrect version. The only way to truly make sure this doesn't happen is to turn off optimizations altogether both in the compiler and in the JVM. Other considerations then come into play (which parts of your code would you want non-optimized?), but that's beyond the scope of this course.

Appendix C: Quadratic Probing with c1=c2= $\frac{1}{2}$ and Capacity= 2^k

In this appendix we prove that you can rest assured: using a table with a power-of-two size with at least one empty/deleted cell, and the values $c_1 = c_2 = \frac{1}{2}$ in the quadratic probing, you will always hit an available cell during the first 2^k attempts where 2^k is the size of the table. That is, if you use arithmetic modulo to clamp the probing's result. We will prove this by contradiction:
Assume this is not the case for a certain hashCode. Then according to the Pigeonhole Principle, two of the first 2^k attempts must hit the same cell.
So we assume that $i_1 \neq i_2 \in [0 : 2^k - 1]$ exist s.t.

$$hash(value) + \tfrac{i_1}{2} + \tfrac{i_1^2}{2} \equiv hash(value) + \tfrac{i_2}{2} + \tfrac{i_2^2}{2} \quad (mod\ 2^k)$$

$$\Rightarrow \exists\, m \in Z\ s.t.\ \ hash(value) + \tfrac{i_1}{2} + \tfrac{i_1^2}{2} - hash(value) - \tfrac{i_2}{2} - \tfrac{i_2^2}{2} = m2^k$$

$$\Rightarrow i_1 + i_1^2 - i_2 - i_2^2 = 2m2^k = m2^{k+1}$$

$$\Rightarrow \underbrace{(i_1 - i_2)}_{a}\underbrace{(i_1 + i_2 + 1)}_{b} = m2^{k+1}$$

Note that $b - a = 2i_2 + 1$ which is an odd number. Therefore either a is even and b is odd, or the opposite is true. Whichever the case, the even one **must** divide by 2^{k+1} (why?).
a cannot possibly divide by 2^{k+1} since it's in the group $\{-2^k + 1 \leq x \leq 2^k - 1,\ x \neq 0\}$ which doesn't contain any number that divides by 2^{k+1}.
b, on the other hand, is contained in the range $\{2 \leq x \leq (2^k - 2) + (2^k - 1) + 1 = 2^{k+1} - 2$ which also doesn't contain any number that divides by 2^{k+1}. Therefore our assumption is false - **Q.E.D.**

Appendix D: 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.

1) The first line starts with a '#' sign, and is a comment describing the test (i.e., this line is ignored).
2) The second line creates a hash-set object using one of the three constructors:
    a. An empty line calls the default constructor.

b. A line starting with "*N:*" (N for "numbers") followed by two integers (e.g., *N: 0.8 0.1*) calls the constructor with an upper and lower load factors respectively.

c. 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.

3) *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 values 0.8 and 0.1, 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.