CS 321: Data Structures
Programming Project #3: Experiments with Hashing
100 points

Introduction

In this project, we will study how the load factor affects the average number of probes required by open addressing while using linear probing and double hashing for different types of inputs. This will also improve our understanding of how hashing works. Note that we are not attempting to create a generic hash table that is usable by all applications.

Suppose we are inserting n keys into a hash table of size m. Then the load factor α is defined to be n/m. For open addressing $n \leq m$, which implies that $0 \leq \alpha \leq 1$.

Objectives

- To understand the concepts of hashing by experimenting with open addressing.
- To understand the `hashCode()` method in Java,
- To demonstrate knowledge of inheritance.
- To learn and demonstrate knowledge of software engineering practice of instrumenting code with levels of debugging output.

Setup

Navigate to the class Git repository that we cloned for the last project and update it as follows:

```
cd CS321-resources
git pull
```

The project starter files are in the subfolder `projects/p3`. Please pay close attention to the project rubric `p3-rubric.txt` in the project folder.

Design

Set up the hash table to be an array of `HashObject`. A `HashObject` contains a generic key object, a frequency count and a probe count. The `HashObject` needs to override both the `equals` and the `toString` methods and should also have a `getKey` method that returns an `Object` type.

We will try two alternatives: linear probing and double hashing. We will design the `HashTable` class as an abstract class with two subclasses, one for linear probing and one for double hashing. All of the common hash table functionality should be in the abstract parent class and the subclasses should only have the differing functionality.

Find a value of the table size m to be a prime in the range $[95500 \ldots 96000]$. A good value is to use a prime that is 2 away from another prime. That is, both m and $m-2$ are primes. Two primes (differ by two) are called "twin primes." Please find the table size using the **smallest twin primes** in the given grange $[95500 \ldots 96000]$. Write a separate class `TwinPrimeGenerator` for generating the twin primes.

We will vary the load factor α as 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.98, 0.99 by setting the value of n appropriately, that is, $n = \alpha m$. We will keep track of the average number of probes required for each value of α for linear probing and for double hashing.

For double hashing, the primary hash function is $h_1(k) = k \bmod m$ and the secondary hash function is $h_2(k) = 1 + (k \bmod (m-2))$.

There are three sources of data for this experiment as described in the next section.

Note that the data can contain duplicates. If a duplicate is detected, then update the frequency for the object rather than inserting it again. Keep inserting elements until we have reached the desired load factor. We will count the number of probes only for new insertions and not when we find a duplicate.

Experiment

For the experiment we will consider three different sources of data as follows. We will need to insert `HashObject`s until the pre-specified α is reached.

- Data Source 1: each `HashObject` contains an `Integer` object with a random int value generated by the method `nextInt()` in `java.util.Random` class. The key for each such `HashObject` is the Integer object inside.
- Data Source 2: each `HashObject` contains a `Long` object with a long value generated by the method `System.currentTimeMillis()`. The key for each such `HashObject` is the Long object inside.
- Data Source 3: each `HashObject` contains a word from the file `word-list` that provided with the starter files for this project. The file contains $3,037,802$ words (one per line) out of which 101,233 are unique. The key for each such `HashObject` is the word inside represented as a `String` type.

Note that, for fair comparison, the data inserted into both linear and double tables must be the same.

When we hash a `HashObject` into a table index, we will need a *key* value. It may not always be possible to have a key value, but Java provides an alternative that works well: a *hash code* for every object by calling the `hashCode()` method on any object. Here is the contract for the hash code from the Java API documentation:

> `public int hashCode()`
>
> Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap.
>
> The general contract of hashCode is:
>
> - Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
> - If two objects are equal according to the `equals(Object)` method, then calling the hashCode method on each of the two objects must produce the same integer result.
> - It is not required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.
>
> As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (The hashCode may or may not be implemented as some function of an object's memory address at some point in time.)

- Use the `hashCode()` method for the key object in the `HashObject` class as the key for the hash table. Java has a built-in `hashCode()` method in every object, so every class inherits it.

- We will use the `hashCode()` methods to perform the linear probing or double hashing calculation. Note that `hashCode()` can return negative integers. We need to ensure the mod operation in the probing calculation always returns positive integers. We will define a helper function `positiveMod` that will be used both in linear probing and in double hashing. For example, the computation of the primary hash value (i.e., $h_1(key)$) and the secondary hash value (i.e., $h_2(key)$) for double hashing should be:

```
h1(key) = positiveMod (key.hashCode(), tablesize);
h2(key) = 1 + positiveMod (key.hashCode(), tablesize - 2);
```

where the `positiveMod` function is defined as follows:

```
protected int positiveMod (int dividend, int divisor)
{
   int value = dividend % divisor;
   if (value < 0)
      value += divisor;
   return value;
}
```

Note that two different objects (key objects) may have the same `hashCode()` value, though the probability is small. Thus, we must compare the actual key objects to check if the `HashObject` to be inserted is a duplicate.

Required file/class names and output

The driver program should be named as `HashtableTest`, and it should have three (the third one is optional) command-line arguments with the following usage:

```
Usage: java HashtableTest <input type> <load factor> [<debug level>]
         input type = 1 for random numbers, 2 for system time, 3 for word list
         debug = 0 ==> print summary of experiment
         debug = 1 ==> save the two hash tables to a file at the end
         debug = 2 ==> print debugging output for each insert
```

The `<input type>` should be 1, 2, or 3 depending on whether the data is generated using `java.util.Random, System.currentTimeMillis()` or from the file `word-list`.

The program should print out the input source type, total number of keys inserted into the hash table and the average number of probes required for *linear probing* and *double hashing*. The optional argument specifies a debug level with the following meaning:

- debug = 0 ⟶ print summary of experiment on the console
- debug = 1 ⟶ print summary of experiment on the console and also save the hash tables with number of duplicates and number of probes into two files `linear-dump.txt` and `double-dump.txt`. The two sample dump files generated by executing

  java HashtableTest 3 0.5 1

are given in the starter files, named `sample-linear-dump.txt` and `sample-double-dump.txt`.

Please make sure the dump files have the same format as the provided sample dump files so that we can use Linux command `diff` *to compare them for correctness checking.*

- debug $= 2 \longrightarrow$ print element by element detailed output to help us debug and trace the behavior of the code.

For debug level of 0, the output to the console is a summary. An example is shown below.

```
java HashtableTest 3 0.5

HashtableTest: Twin prime table size found in the range [95500..96000]: 95791

HashtableTest: Data source type --> word-list

HashtableTest: Using Linear Hashing....
HashtableTest: Input 1305930 elements, of which 1258034 duplicates
HashtableTest: load factor = 0.5, Avg. no. of probes 1.5969183230332387

HashtableTest: Using Double Hashing....
HashtableTest: Input 1305930 elements, of which 1258034 duplicates
HashtableTest: load factor = 0.5, Avg. no. of probes 1.3904918991147486
```

Note that empty entries of the table are omitted in the output. You can compare your output with the sample output on onyx (or any Linux or Mac system) with the following commands:

```
diff linear-dump.txt sample-linear-dump.txt
diff double-dump.txt sample-double-dump.txt
```

Note that the dumps from the random and system time inputs will not be comparable. That's why we are providing dumps for the word list for a specific load factor.

Submission

A `README.md` file, following the usual template, should contain tables showing the average number of probes versus load factors in the appropriate section. There should be three tables for the three different sources of data. Each table should have eight rows (for different α) and two columns (for linear probing and double hashing). A sample result containing three tables can be seen the file `sample_result.txt` in the starter files for the project.

Please remove `*.class` files and other unnecessary files (such as word-list and sample files provided by the instructors)before submitting.

Before submission, you need to make sure that your program can be compiled and run in `onyx`. Submit your program(s) from `onyx` by copying all of your files to an empty directory and typing the following FROM WITHIN this directory:

```
submit amit CS321 p3
```