CS 321 Data Structures (Fall 2017)
Programming Assignment #3, Due on 10/20/2017, Friday (11PM)

Introduction:

Suppose we are inserting n keys into a hashtable of size m. Then the load factor α is defined to be $\alpha = n/m$. For open addressing $n \leq m$, which implies that $0 \leq \alpha \leq 1$. In this assignment we will study how the load factor affects the average number of probes required by open addressing while using linear probing and double hashing.

Design:

Set up the hash table to be an array of HashObject<T>. A HashObject<T> contains a generic object T, a duplicate count, and a probe count. The HashObject<T> needs to override both the *equals* and the *toString* methods and should also have a *getKey* method.

Also we will use linear probing as well as double hashing. So design the HashTable<T> class by passing in an indicator via constructor so that the appropriate kind of probing will be performed.

Implement a method that chooses a value of the table size *m* to be a prime in the range [95500 ...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 that differ by are called “Twin Primes”. Please find the table size using the smallest twin primes in the given range [95500...96000].

Vary the load factor as $\alpha \in$ { 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, $n = \alpha m$. Keep track of the average number of probes required for each value of α for linear probing and for double hashing.

For the double hashing, the primary hash function is $h1(k) = k \bmod m$ and the secondary hash function is $h2(k) = 1 + (k \bmod (m-2))$.

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 number of duplicates for the object rather than inserting it again. Keep inserting elements until you have reached the desired load factor (do not count duplicates). Count the number of probes only for new insertions and not when you found a duplicate.*

Experiment 1:

For the experiment we will consider three different sources of data as follows. You will need to insert HashObjects until the pre-specified α is reached (without counting duplicates), where

- **Data Source 1:** each HashObject<T> contains an Integer object (i.e. T is an Integer) randomly generated by the method *nextInt()* method in *java.util.Random* class. Initialize the Random object with the seed 1234. The key for each such HashObject is obtained by calling the *hashCode()* method (a method in *Object* class) on the Integer object inside.
- **Data Source 2:** each HashObject<T> contains a Long object (i.e. T is a Long) generated by the method *System.currentTimeMillis()*. The key for each such HashObject is obtained by calling the *hashCode()* method on the Long object inside.
- **Data Source 3:** each HashObject contains a word from the file **word-list** contained in the zip file. The file contains 3,037,798 words (one per line) out of which 101,233 are unique. Note that after

you read in a word, you will have to convert it to a number as its key value by calling the *hashCode()* method on the String object inside.

NOTE: When implementing the *getKey* method, apply *Math.abs()* to the value returned by *hashCode()* to ensure that key values are positive.

Note that two different elements from the same data source may have the same key *hashCode()* values, though the probability is small. Thus, we must compare the actual element to check if it already exists in the table.

Required file/class names and output for Experiment 1:

The source code for Experiment 1. The driver program should be named as *HashTest*, it should have three (the third one is optional) command-line arguments as follows:

java HashTest <input type> <load factor> [<debug level>]

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 (1) input source type, (2) total number of keys inserted into the hash table, (3) 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 the experiment on the console
- debug = 1 → print summary of the experiment on the console and also print the hash tables into two files linear-dump and double-dump.

For debug value of 0, the output is a summary. An example is shown below.

```
[fspezzano@onyx sol]$ java HashTest 1 0.5 0
A good table size is found: 95791
Data source type: random number generator

Using Linear Hashing....
Inserted 47895 elements, of which 0 duplicates
load factor = 0.5, Avg. no. of probes 1.503
Using Double Hashing....
Inserted 47895 elements, of which 0 duplicates
load factor = 0.5, Avg. no. of probes 1.380
```

For debug value of 1, the dump file should have the following format. The first number after the word is the number of duplicates and the second is the number of probes. Note that empty entries in the table are omitted in the output.

```
Table[0]:  3 1
Table[1]: enfetter'd 0 1
Table[2]: atherton 1 1
Table[3]: yorick's 0 1
Table[4]: whate'er 25 1
Table[5]: frank'd 1 2
Table[8]: cried 89 1
```

Table[11]: mansfield 4 1
Table[13]: proserpina's 0 1
Table[17]: lin'd 4 1
Table[18]: dropsy 1 1

Submit the JAVA source classes developed for Experiment 1 and a readme file that contains tables showing the average number of probes versus load factors. 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 in the file *sample_result.txt* contained in the zip file.

Experiment 2: Using Java HashMap

In this experiment you will make practice in using Java's implementation of HashTables. Modify the HashTest.java to add another experiment using JAVA provided *Hashtable<K,V>* class. Details about this class can be found at https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html
The *Hashtable<K,V>* class is actually a hashing-by-chaining HashTable. The *Hashtable<K,V>* class is in the *java.util* package. The methods *get* and *put* defined in the *Hashtable* class can be used to conduct the experiment. A *Hashtable<K,V>* can store a set of <key, value> pairs. In this assignment, the key is the data object generated from the data sources, i.e. the key can be an *Integer*, a *Long*, or a *String*, and the corresponding value is an *Integer* object recording the duplicate count.

Required file/class names and output for Experiment 2:

The driver program for this second experiment should be named as *HashTestWithJava*, it should have three (the third one is optional) command-line arguments as follows:

java HashTestWithJava <input type> <load factor> <tableSize> [<debug level>]

The <input type> should be 1, 2, or 3 depending on whether the data (i.e., key in the <key, value> pair) is generated using *java.util.Random*, *System.currentTimeMillis()* or from the file word-list. Create the *Hashtable* by using the constructor taking in input the tableSize.

The program should print out (1) the input source type and (2) total number of keys inserted into the hash. NOTE that this time is not required to print the average number of probes because we do not have this information for the JAVA implementation of HashTables.

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 print the hashmap-dump. To print the hashmap-dump, the *toString()* method in *HashMap* class can be used to print the <key, value> pairs stored in the map.

For debug value of 0, the output is a summary. An example is shown below.

[fspezzano@onyx sol]$ java HashTestWithJava 3 0.5 95791
Table size: 95791
Data source type: word-list
Using JAVA Hashtable....

Inserted 47895 elements, of which 1258020 duplicates
load factor = 0.5

Submit the JAVA source class developed for Experiment 2.

Final Notes

Please do not submit executable since we'll be recompiling your programs.
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 (with no subdirectories) and typing the following FROM WITHIN this directory:

submit *instructorAccount* cs321 hashtable