CS 321 Data Structures (Fall 2016)

Programming Assignment #3, Due on 10/18/2016, Tuesday (11PM)

- **Introduction:**

  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 $0 \leq \alpha \leq 1$. 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:**

HashObject:
<T> obj
int duplicates
int probes

  Set up the hash table to be an array of HashObject. A HashObject contains a generic object and a frequency count. The HashObject needs to override both the `equal` and the `toString` methods and should also have a `getKey` method.

public boolean equals(obj)
{
if (obj is a String)
compare objects directly
else
compare obj.HashCode()
}

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

public int getKey(obj)
{
return obj.HashCode();
}

  Choose 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. 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, $n = \alpha m$. Keep track of the average number of probes required for each value of α for linear probing and for double hashing.

HashTable(String tblType)
{
hashTblType = tblType;
set other instance vars ...
}

  For the 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.

Keep inserting elements until you reach 'n'.

  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 you have reached the desired load factor. Count the number of probes only for new insertions and not when you found a duplicate.

- **Experiment:**

  For the experiment we will consider three different sources of data as follows:

  - Generate random keys using `java.util.Random` class.
  - Generate keys by making calls to the method `System.currentTimeMillis()`.
  - Insert words from the file `word-list` in the directory
    `~lgundala/CS321/labs/lab3/files`
    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 (a method in Object class).

NOTE: 2 different strings may have the same HashCode() values. Uncommon, but possible.

Note that two different words may have the same key values, though the probability is small. Thus, we must compare the actual words to check if the word already exists in the table.

Don't copy the `word-list` file to your directory since its is large. Instead set up a symbolic link as follows:

`ln -s ~lgundala/CS321/labs/lab3/files/word-list`

- **Required file/class names and output:**

  The source code for the project. 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 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:

  Default debug level = 0

  debug = 0 ⟶ print summary of experiment
  debug = 1 ⟶ print out the hash table at the end ... after the summary
  debug = 2 ⟶ print the number of probes for each new insert ... in addition to output from level 1.

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

```
[lgundala@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 47896 elements, of which 0 duplicates
load factor = 0.5, Avg. no. of probes 1.5003966928344747

Using Double Hashing....
Inserted 47896 elements, of which 0 duplicates
load factor = 0.5, Avg. no. of probes 1.3893853348922667
```

  For debug value of 1, the table should be output in the following format.

```
table[3]: daie 0
table[5]: faience 2
table[6]: gaiety 1
table[15]: paienne 0
table[16]: amyntas 0
table[17]: gutzkow 0
table[44]: bingham 1
```

```
table[45]: bingham's 0
table[46]: ding 6
table[48]: finger 70
```

Note that null values in the table are omitted in the output.

- **Submission**

  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 below

  ~lgundala/CS321/labs/lab3/files/sample_result.txt

  Please do not submit executable since I'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 lgundala CS321 p3
  ```