

SE2205b - Algorithms and Data Structures for Object-Oriented Design

Laboratory 4: Hash Tables

Due Date: March 18th, 2018

Student Name	
Student Number	
Laboratory Date/Time	

Note:

- **For this lab you will need to submit the full solution (PDF file and code) into both OWL and GitHub.**

1 Goal

In this lab two different collision resolution schemes will be implemented for a hash table and the resulting performance will be compared with that of linear hashing.

2 Resources

- Unit 6: Dictionaries

3 Java Project Files

- NetBeans project file: HashTable.zip

4 Create a new repository for Lab4

1. Download the lab4 file SE2205B-Lab4.zip from OWL into your course folder say "SE2205B".
2. Unzip this downloaded file to have a folder called "SE2205B-Lab4". (don't remove the hyphen)
3. Open GitHub Desktop.
4. Run File → Add local repository.
5. Click Choose and browse for the folder SE2205B-Lab4.
6. You will get a warning says "This directory does not appear to be a Git repository. Would you like to create a repository here instead?"
7. Click "create a repository" and then click Create repository.
8. In the GitHub Desktop tool bar, click Publish repository, check the option "keep this code private", choose your GitHub account, and then click Publish repository.
9. Now a new repository called "SE2205B-Lab4" should appear into your GitHub account.
10. It is a mandatory to add the instructor and the TAs to this repository.
 - a. In the GitHub account click the repository SE2205B-Lab4 and then click "Setting".
 - b. Click "Collaborators & teams" option. At the bottom of the page and in the Collaborators section, enter the account id of the GitHub user your want to add and then click "Add collaborator".
 - You need to add the following accounts to your repo: **aouda**, **kalhazmi2**, **rafadaguair**, **rbarboza**.

5 Introduction

One of the fastest dictionary implementations is the hash table. As long as the table does not become too full, the time for adding and finding an element will be $O(1)$. This performance does not come without some cost. The obvious penalty is that there will be space in the table that is wasted. Another penalty is that the items in the hash table are not in any particular order. Other dictionary implementations will keep items in key order, but it is an inherent property of the hash table that items are not ordered. In fact, as more items are added to the hash table, the size of the table may be increased to maintain the performance. In this case, the items will be rehashed and will no longer be in the same locations or order.

5.1 General Collision Resolution

To place an item in a hash table of size m , a hash function $H(k, m)$ is applied to the key k . An integer value between 0 and $m-1$ will be returned and will be the location of the object. If there is already an object in that location, a collision has occurred and must be resolved. In a hash table with open addressing, collisions are resolved by trying other locations until an empty slot is found. One way of viewing this process is that there is a series of hash functions $H_0(), H_1(), H_2(), H_3(), \dots, H_i(), \dots$, which are applied one at a time until a free slot is found.

5.2 Linear Hashing

For linear hashing (or linear probing), slots in the hash table are examined one after another. From the view of the general scheme, the hash functions are.

$$\begin{aligned} H_0(k, m) &= H(k, m) \\ H_1(k, m) &= (H(k, m) + 1) \bmod m \\ H_2(k, m) &= (H(k, m) + 2) \bmod m \\ &\dots \\ H_i(k, m) &= (H(k, m) + i) \bmod m \end{aligned}$$

The mod operation is required to keep the values in the range from 0 to $m-1$. While you could use these formulas to compute each of the hash locations, usually the previous value is used to compute the next one.

$$H_i(k, m) = (H_{i-1}(k, m) + 1) \bmod m$$

Linear hashing has the advantage of a simple computational formula that guarantees all the slots will be checked. The performance of linear hashing is affected by the creation of clusters of slots that are filled. Suppose that there are relatively few large clusters. If there is a collision with a slot inside a cluster, getting outside of the cluster will require a large number of probes. For the best performance, the free slots should be distributed evenly and large clusters avoided.

6 Pre-Lab Visualization

6.1 Double Hashing

Double hashing is scheme for resolving collisions that uses two hash functions $H(k, m)$ and $h(k, m)$. It is similar to linear hashing except that instead of changing the index by 1, the value of the second hash function is used.


From the view of the general scheme, the hash functions are.


$$\begin{aligned} H_0(k, m) &= H(k, m) \\ H_1(k, m) &= (H(k, m) + h(k, m)) \bmod m \\ H_2(k, m) &= (H(k, m) + 2 h(k, m)) \bmod m \\ &\dots \\ H_i(k, m) &= (H(k, m) + i h(k, m)) \bmod m \end{aligned}$$


As with linear hashing, the hash function can be defined in terms of the previous values.

$$H_i(k, m) = (H_{i-1}(k, m) + h(k, m)) \bmod m$$

You must be careful when defining the second hash function.

Suppose that $H(k,m)$ is 12, $h(k,m)=0$, and $m=15$. What are the locations that will be probed?																
$H0(k, m)$	$H1(k, m)$	$H2(k, m)$	$H3(k, m)$	$H4(k, m)$	$H5(k, m)$											

Suppose that $H(k,m)$ is 12, $h(k,m)=4$, and $m=15$. What are the locations that will be probed?																
$H0(k, m)$	$H1(k, m)$	$H2(k, m)$	$H3(k, m)$	$H4(k, m)$	$H5(k, m)$											

If $m=15$, which values of $h(k,m)$ will visit all of the locations in the table?																	
$h(k,m)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14		15
Visits all locations?																	

Since you really want to probe the entire table, the value returned by the second hash function has some limitations. The first condition is that it should not be 0. The second condition is that it should be relatively prime with respect to m . A common way to guarantee the second condition is to choose a table size that is a prime.

Suppose that you have access to an integer value c that is based on the key

$c = \text{HashCode}(k)$

The first hash function will be computed as

$H(k,m) = c \bmod m$

Under the assumption that m is a prime, give a formula for computing a second hash function using c . It should return values in the range of 1 to $m-1$



Double hashing can still be affected by clustering. Every key that has the same value for the second key will probe the table in the same pattern and can still be affected by clusters. Show how to modify the following code so that it computes the second hash value and then uses it in the search.

```
private int locate(int index, T key) {
    boolean found = false;
    while ( !found && (hashTable[index] != null) ) {
        if (hashTable[index].isIn() && key.equals(hashTable[index].getKey()))
            found = true; // key found
        else // follow probe sequence
            index = (index + 1) % hashTable.length; //linear probing
    } // end while
    // Assertion: either key is found or a null location is reached
    int result = -1;
    if (found) result = index;
    return result;
} // end locate
```

```
private int locate(int index, T key) {
```



6.2 Perfect Hashing

In perfect hashing, associated with each key is a unique random sequence of probe locations. Since each key has a unique “view” of the table, the locations of the free slots will be randomly spread out and clustering will be avoided. Even though an approximation to perfect hashing will be implemented in the lab, it is mainly of theoretical interest because perfect hashing is much easier to analyze than linear or double hashing.

Let $s_0(k)$, $s_1(k)$, $s_2(k)$, ... be a random sequence of values in the range 0 to $m-1$.

```

H0(k, m) = s0(k)
H1(k, m) = s1(k)
H2(k, m) = s2(k)
...
Hi(k, m) = si(k)

```

A truly random sequence is not possible, but you can approximate it using pseudo random numbers. Most algorithmic random number generators use the following simple formula to compute a sequence of numbers:

$$V_{n+1} = (aV_n + c) \bmod m$$

Given that $a = 3$, $c = 2$, and $m = 10$, what is the sequence of numbers computed?

V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7	



Is the preceding sequence random? No. It follows a prescribed sequence. If you know one value in the sequence, you know the next value. Further, notice that the preceding sequence misses some of the values between 0 and 9. This means it would not be suitable for a random number generator. In fact, most values for a , c , and m do not result in good pseudorandom number generators, so it is best not to choose your own values. Instead, either use a professionally designed random number generator or use one of the published sets of values that have passed a thorough battery of statistical tests.

Even though the sequence is not random, for good choices of values it can appear random which is sufficient for our algorithm. In addition, the fact that the values are actually not random is crucial for the implementation of the perfect hashing algorithm. Each time a value is searched for, you must follow exactly the same sequence. The first value in the sequence is called the seed. If you initialize the pseudorandom number generator with a given seed, it will always produce the same sequence of numbers. This will be the basis for the probe sequence used in perfect hashing.

6.3 Creating Random Search Keys

The average time to locate a value in a hash table is usually given based on whether the value is in the table or not. To insert a new value in a table, you must probe for a free slot. Thus the time to insert a new value is basically the same as the time to search for a value that is not in the table. To search for a value that is in the table, you must follow the same pattern of probes that was used when the value was inserted. The average will include values that were inserted early and thus require few probes to locate. This average will be less than the average number of probes required to determine that a value is not in the table.

To test both kinds of searches, an array of unique random words will be created. The first half of the array will be inserted into a hash table. The average of finding a value in the first half of the array will give the average for successful searches and the average over the second half will give the average for unsuccessful (failure) searches.

To make the test more interesting, random three-syllable pseudowords will be created. It is possible that a word will be generated twice. To avoid placing such words in the array, you will have to test to see if the word has been generated before. Using a hash table is a perfect way to do this. As a word is generated, check the hash table to see if it has been generated before. If not, add it to the array and the hash table.

Write an algorithm that creates the array of unique random words. You may assume that three arrays of syllables `firstSyl`, `secondSyl`, and `thirdSyl` have already been created. Further, assume that a random number generator of type `Random` from the package `java.util` has been created. It may be helpful to use the method `nextInt(int k)` which will return a random integer from 0 to $k-1$.



6.4 The Average Number of Words Generated to Get a Unique Word

One thing to consider is the number of words that will be generated before a unique value is found. As more unique words are generated, it becomes more and more likely that you will randomly generate a previously created word and have to discard it. This may become too much of a burden. Let's find out how much of a burden it will be.

To do so, the probability that more than one word will be generated will have to be computed. Probabilities are real values between 0 and 1. It tells you the likelihood that an event occurs. An event with a probability of 0.5 has a 50% chance of occurring. An event with a probability of 1 is certain.

Suppose that there are a total of $T=1000$ unique words that can be created. (In general, T will be the sizes of the three syllable arrays multiplied together.) Suppose further that 600 words have already been generated.

What is the probability that a randomly generated word will be one that has been generated before?



What is the probability that a randomly generated word will not be one that has been generated before?



To determine the average number of words that will need to be generated in order to get a unique word, one must consider all the possible events (number of words generated to get the unique word) along with their probabilities. Multiplying the number of words needed for each event by the corresponding probability and then adding all the products together gives the average. Let's create a table with that information (under the assumption that 600 of 1000 unique words have been found already).

Event	Probability	Value	Product
1 word generated			

2 words generated			
3 words generated			
4 words generated			
5 words generated			
6 words generated			
7 words generated			
8 words generated			
9 words generated			
...			

The probability that only one word is generated will be just the probability that the first word generated is not one that has been generated before. Write that probability in the table.

The probability that two words are generated will be the product of the probability that the first word had been generated before times the probability that the second word was not generated before. Write that probability in the table.

The probability that three words are generated will be the product of the probability that the first word had been generated before times the probability that the second word had been generated before times the probability that the third word was not generated before. Write that probability in the table.

There is a pattern here. Using that pattern, complete the probabilities column in the table.

The values will just be the number of words generated. Fill in that value column in the table.

For each row in the table multiply the probability by the value and record the result in the product column.

Add all the products together and record the sum here.



To get the exact answer, you must add up an infinite number of terms. The product is composed of two parts. One part is getting smaller exponentially and the other is getting larger linearly. Eventually the exponential part will dominate and the sum will converge. For the given situation, nine terms will give an answer that is reasonably close to the exact value of 2.5.

As long as no more than 60% of the possible words have been generated, the number of extra words that get generated will be less than 1.5 and will not be too much of a burden on our algorithm.

The syllable arrays given in the lab each have a size of 15. How many possible words are there?

What is 60% of this total?

To test hash tables with more data values than this, the size of the syllable arrays will need to be increased.



6.5 Counting the Number of Probes

Consider again the code that locates an item or a free slot in the hash table using linear probing.

```
private int locate(int index, T key) {
    boolean found = false;
    while ( !found && (hashTable[index] != null) ) {
        if (hashTable[index].isIn() && key.equals(hashTable[index].getKey()))
            found = true; // key found
        else // follow probe sequence
            index = (index + 1) % hashTable.length; //linear probing
    } // end while
    // Assertion: either key is found or a null location is reached
    int result = -1;
    if (found)
        result = index;
    return result;
} // end locate
```

Suppose that index is 2 and key is 57. Trace the code and circle the index of any location that is accessed.



0	1	2	3	4	5	6	7	8	9
	15	25	13	57	19			16	11

How many locations were circled? (How many probes were made?)

How many times did the body of the loop execute?

Now suppose index is 2 and key is 99. Trace the code again and circle the index of any location that is accessed.

0	1	2	3	4	5	6	7	8	9
	15	25	13	57	19			16	11

How many locations were circled? (How many probes were made?)

How many times did the body of the loop execute?

It should be the case that only one of the traces had the same number of loop executions as probes.

Show how to modify the locate () method given previously so that it will add the number of probes made to a static variable named total Probes.

Use **GitHub desktop** to commit your work.

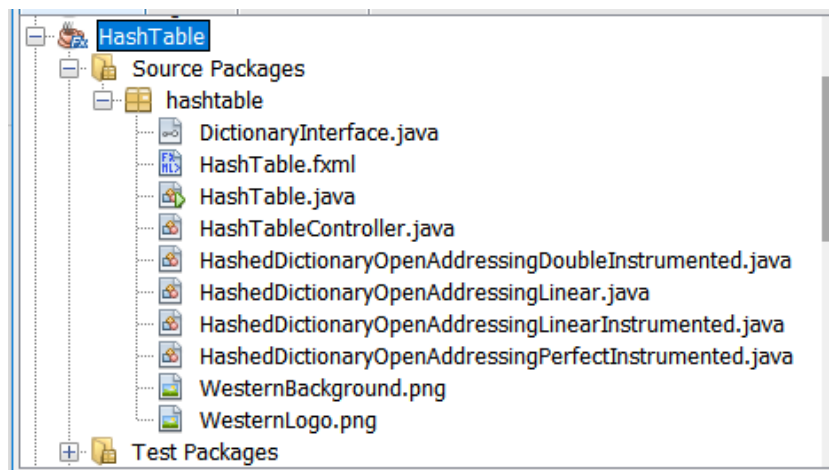
1. Click the changes tab in the left sidebar to see a list of the files that have been changed or added since the last commit.
2. Use the checkboxes to indicate which files should be part of the commit. In this activity, you'll select the "SE2205B Lab4 – HashTables.pdf" file.
3. Type your commit message (Pre-lab checkpoint) in the Summary field.
4. You will notice that GitHub Desktop has already populated the commit button with the current branch. Simply click the button to commit your changes
5. Click Push origin in the menu bar to push this change to your GitHub account.

Checkpoint Once you completed the pre-lab visualizations, commit and push this workbook to your GitHub account.

7 Directed Lab Work

7.1 Import Source code to IDE

1. Open NetBeans IDE and import the project "HashTable.zip". When you import it successfully you will have all files shown in the figure.
2. All required class are given with some empty method that you need to complete as directed in the following sections.
3. At this point, the given code should compile and run without error but with incomplete methods.
4. Do not proceed further until you complete this step successfully.



7.2 Startup

A hash table class with linear collision resolution has already been implemented in the `HashedDictionaryOpenAddressingLinearInstrumented` class.

You will complete/implement two new classes

`HashedDictionaryOpenAddressingDoubleInstrumented`, and

`HashedDictionaryOpenAddressingPerfectInstrumented` based on that class.

They will allow you to gather statistics about the number of probes made to insert values. The `HashPerformance` class will generate random arrays of keys, insert the keys in the various kinds of hash tables, and then display the averages.

7.3 Implementing Double Hashing

- Step 1.** If you have not done so, look at the implementation of a hash table with linear probing in `HashedDictionaryOpenAddressingLinear.java`. Compile and run the application, from the main menu click Linear Hashing → Testing Linear Hashing. The application will generate an array of

1000 random values between 0 and 1000. The first 500 of those values will be inserted into a hash table. The code will check that searches work correctly. The first 250 values in the array will then be removed from the hash table. Again, searches will be checked. Finally, the last 500 values in the array will be added into the hash table. Again, searches will be checked.

Verify that the code passed each of the three tests.

The first goal is to create the class for double hashing and verify that it works.

- Step 2.** We have copied `HashedDictionaryOpenAddressingLinearInstrumented.java` into a new file `HashedDictionaryOpenAddressingDoubleInstrumented.java`.
- Step 3.** After the comment line “ADD IN CODE FOR THE SECOND HASH FUNCTION”, create a new private method `getSecondHashIndex(Object key)`, which computes a second hash function. Refer to the formula created in the Pre-Lab exercises.
- Step 4.** Refer to the Pre-Lab exercises and modify the `locate()` and `probe()` methods to use double hashing instead of linear hashing.
- Step 5.** Compile and run the application, from the main menu click Double Hashing → Testing Double Hashing. All tests should pass.

The next goal is to create the class for perfect hashing and verify that it works

Notice Make sure that you have the following values for the input fields: 1, 1000, and 123 for the number of trials, the number of data values, and a seed respectively.

7.4 Implementing Perfect Hashing

- Step 1.** We have copied `HashedDictionaryOpenAddressingLinearInstrumented.java` into a new file `HashedDictionaryOpenAddressingPerfectInstrumented.java`
- Step 2.** Create a new private method `getHashGenerator(Object key)` which will create the random number generator used to generate the sequence of probes. Refer to your answer from the Pre-Lab exercise.
- Step 3.** Again, refer to the Pre-Lab exercises and modify the `locate()` and `probe()` methods to use perfect hashing instead of linear hashing. (Remember to change the first argument to be a random number generator instead of an integer.)
- Step 4.** Find all places where the `locate()` and `probe()` methods are called and change it so that `getHashGenerator` is called instead of `getHashIndex`. Once you are finished, there should no longer be any calls to `getHashIndex`. Remove the `getHashIndex` method.
- Step 5.** Compile and run the application, from the main menu click Perfect Hashing → Testing Perfect Hashing. All tests should pass, if not, debug the code and retest.

Notice Make sure that you have the following values for the input fields: 1, 1000, and 123 for the number of trials, the number of data values, and a seed respectively.

7.5 Count the number of probes

- Step 1.** Now you will work on the file `HashedDictionaryOpenAddressingLinearInstrumented.java`.
- Step 2.** Refer to the Pre-Lab exercises and add in code to the `locate()` and `probe()` methods that will count the number of probes.
- Step 3.** Compile and run the application, from the main menu click Linear Hashing → Testing Linear Hashing. All tests should pass, if not, debug the code and retest.

Notice Make sure that you have the following values for the input fields: 1, 1000, and 123 for the number of trials, the number of data values, and a seed respectively.

- Step 4.** Make similar changes in `HashedDictionaryOpenAddressingDoubleInstrumented` and `HashedDictionaryOpenAddressingPerfectInstrumented`.
- Step 5.** Compile and run the application, from the main menu click Double Hashing → Testing Double Hashing and Perfect Hashing → Testing Perfect Hashing. All tests should pass, if not, debug the code and retest.

7.6 Generating Random Keys

- Step 1.** Finish the method `generateRandomData()` in the class `HashTableController`. Refer to the algorithm in the Pre-Lab exercises.
- Step 2.** Compile and run the application, Enter 1 for the number of items to insert, 10 for the number of trials, and 0.9 for the maximum load factor for the hash table.
- Step 3.** Click Hash Performance → Check Performance. If all has gone well, the total number of probes will be 10 and the average will be 1.
- Step 4.** Enter 10 for the number of items to insert, 10 for the number of trials, and 0.9 for the maximum load factor for the hash table.
- Step 5.** Click Hash Performance → Check Performance. If all has gone well, the total number of probes will be approximately 105 and the average will be 1.05. Check that the strings in each array are all different.
- Step 6.** Enter 80 for the number of items to insert, 10 for the number of trials, and 0.9 for the maximum load factor for the hash table.
- Step 7.** Click Hash Performance → Check Performance. If all has gone well, the total number of probes for linear hashing will be approximately 2200. The total number of probes for double and perfect hashing should be about 1550. In general, perfect hashing is expected to take slightly fewer probes than double hashing.

7.7 Insert Performance

- Step 1.** Run HashPerformance for different numbers of items to be inserted into the hash table and record the results in the following two tables. In each case, enter 10 for the number of trials and 0.5 for the maximum load for the hash table.



Average Number of Probes for the Three Kinds of Hash Tables

NUMBER OF ITEMS INSERTED	AVERAGE PROBES FOR LINEAR	AVERAGE PROBES FOR DOUBLE	AVERAGE PROBES FOR PERFECT
-----------------------------	------------------------------	------------------------------	-------------------------------

	HASHING	HASHING	HASHING
10			
20			
30			
40			
50			
60			
70			
80			
90			
100			
110			

You should notice a sudden jump in the number of probes needed. The hash table resizing itself and then rehashing all the items cause this. The average cost for the insertions after a resize will show a decrease as the cost of the resizing is spread out over the insertions that follow. At approximately what values did a resizing occur?



Average Number of Probes for the Three Kinds of Hash Tables

NUMBER OF ITEMS INSERTED	AVERAGE PROBES FOR LINEAR HASHING	AVERAGE PROBES FOR DOUBLE HASHING	AVERAGE PROBES FOR PERFECT HASHING
100			
200			
300			
400			
500			
600			
700			
800			
900			
1000			

7.8 Insertion Performance versus Initial Table Size

The cost of resizing the table is a hidden cost that gets spread out over all of the insertions. If you can accurately predict the number of data values to be inserted, this hidden cost can be avoided by setting the initial size of the table to be larger.

Step 1. Run HashPerformance for different initial sizes of the hash table and record the results. In each case, enter 1000 for the number of items to insert, 10 for the number of trials, and 0.5 for the maximum load.



Average Number of Probes with Respect to the Initial Table Size

NUMBER OF ITEMS INSERTED	AVERAGE PROBES FOR LINEAR HASHING	AVERAGE PROBES FOR DOUBLE HASHING	AVERAGE PROBES FOR PERFECT HASHING
50			
100			
250			
500			
1000			
2000			

In each of the cases, the final size of the hash table will be about 2000.

7.9 Search Performance versus Load Factor

The cost of searching for an item in a hash table is not affected by resizing. Code will be added to distinguish between the number of probes required to search for items in the table (successful search) and items that are not in the table (failure or unsuccessful search).

- Step 1.** In the class HashTableController, add a method insertHalfData() that is based on insertAllData(). It will insert just the first half of the array into the hash table.
- Step 2.** In the class HashTableController, add the methods searchFirstHalf() and searchSecondHalf() that search for each of the keys in the first and second half of the array, respectively. Use the method contains() to determine if the value is in the hash table.
- Step 3.** Change the call to generateRandomData() so that it uses 2*insertCount instead of insertCount.
- Step 4.** Change the code in HashTableController class so that it calls insertHalfData() instead of insertAllData().
- Step 5.** After the code that records the number of probes for the insertions, add code that calls resetTotalProbes, performs searchFirstHalf, and then finally records the number of probes needed for the successful searches. Do this for each of the three kinds of hash tables.
- Step 6.** After that code, add code that calls resetTotalProbes, performs searchSecondHalf, and finally records the number of probes needed for the unsuccessful searches. Do this for each of the three kinds of hash tables.
- Step 7.** Enter 100 for the number of values to insert 1000 for the number of trials, 0.175 for the maximum load factor, and 75 for the initial table size and run HashPerformance.

The values should be close (typically a difference between -0.1 to 0.1) to the ones listed in the following table. If the values are close but not within the desired range, run the code again with the same values and recheck the results. If the values are still not close, carefully examine the code for errors.

	Average Number of Probes to Insert the Data	Average Number of Probes for Successful Searches	Average Number of Probes for Unsuccessful Searches
Linear Hashing	3.1	1.7	3.6
Double Hashing	2.6	1.5	2.6
Perfect Hashing	2.6	1.5	2.6

8 Hand in

For this lab you need to submit the full solution (PDF file and code) into both OWL and GitHub.

1. Using NetBeans IDE, export the Dictionary project to ZIP file and name it yourUwoId_SE2205B_Lab4.zip. Use File → Export Project → to ZIP, then enter the zip file name.
2. Submit at the due date mentioned above
 - a. This zip file along with your solution workbook (this PDF file, make sure your answers are saved), using OWL assignment link
 - b. Commit and Push your full solution folder along with your solved workbook into your GitHub account.**