Craig Lombardo

CS 150 Lab 4

Professor Liew

2/22/15

1. Introduction

The purpose, or goal, of this lab was to explore how to create a Hash Table in Java and how to store objects in the table. We also wanted to test the performance of insertion and find operations on Hash Tables. For this lab the problem included figuring out what load factors allowed for a fast and efficient program, and which had a large number of collisions, which slowed down computational time. For this lab I assumed that using LinkedLists was the best route to take in implementing HashTables. I also assumed that the hashing function we are using is the best for our particular lab, which I am a bit skeptical of. We also had to assume our random number generator created unique keys for all entries, which is rather unlikely but important to note.

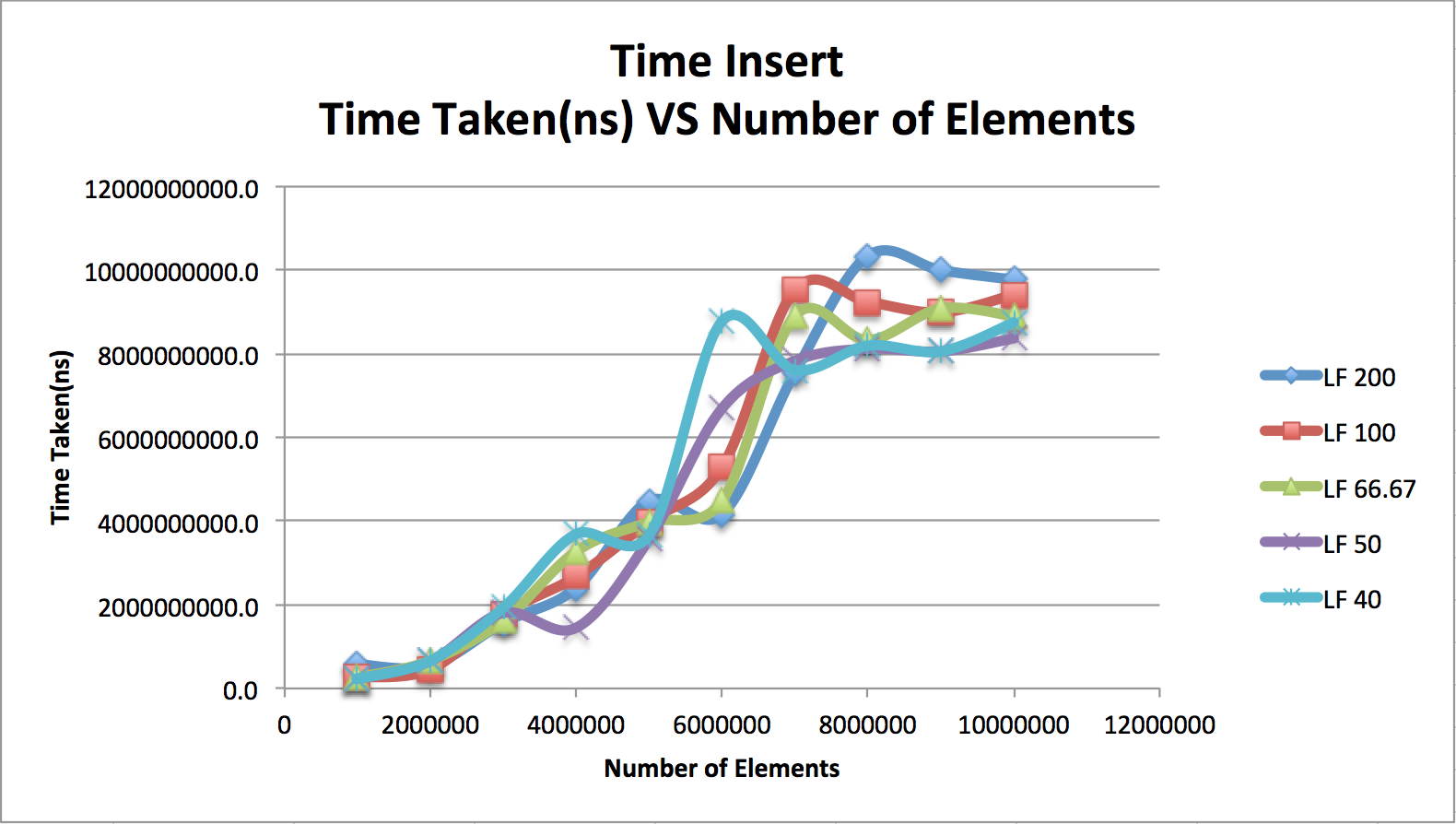
2. Approach:

For the purposes of this lab I have designed 4 classes for to conduct my HashTable testing. There is a class called Entry which will contain data passed into the HashTable. Each Entry has two values stored in it, a “key”, which helps for quick searches, and a “value”, which is the data stored within an Entry. The Entry class only has a method which returns the key associated with an Entry; the purpose of this is to get access to private data stored within an Entry. The second class is the beloved wheel class from all prior experiments; this class is used to generate pseudorandom numbers, which will be used for the keys and values. The third class, MyHashTable, creates a HashTable, which will store all instances of Entry passed into it. The HashTable will be created as an array of LinkedLists, which take in type Entry. In this class we will have methods to insert, find, and analyze the time and load factors of the HashTable. The final class will be an ExperimentController class, which will test my various classes and methods; This provides me with experimental data, which may be analyzed to draw conclusions on the effect of load factor and task to complete.

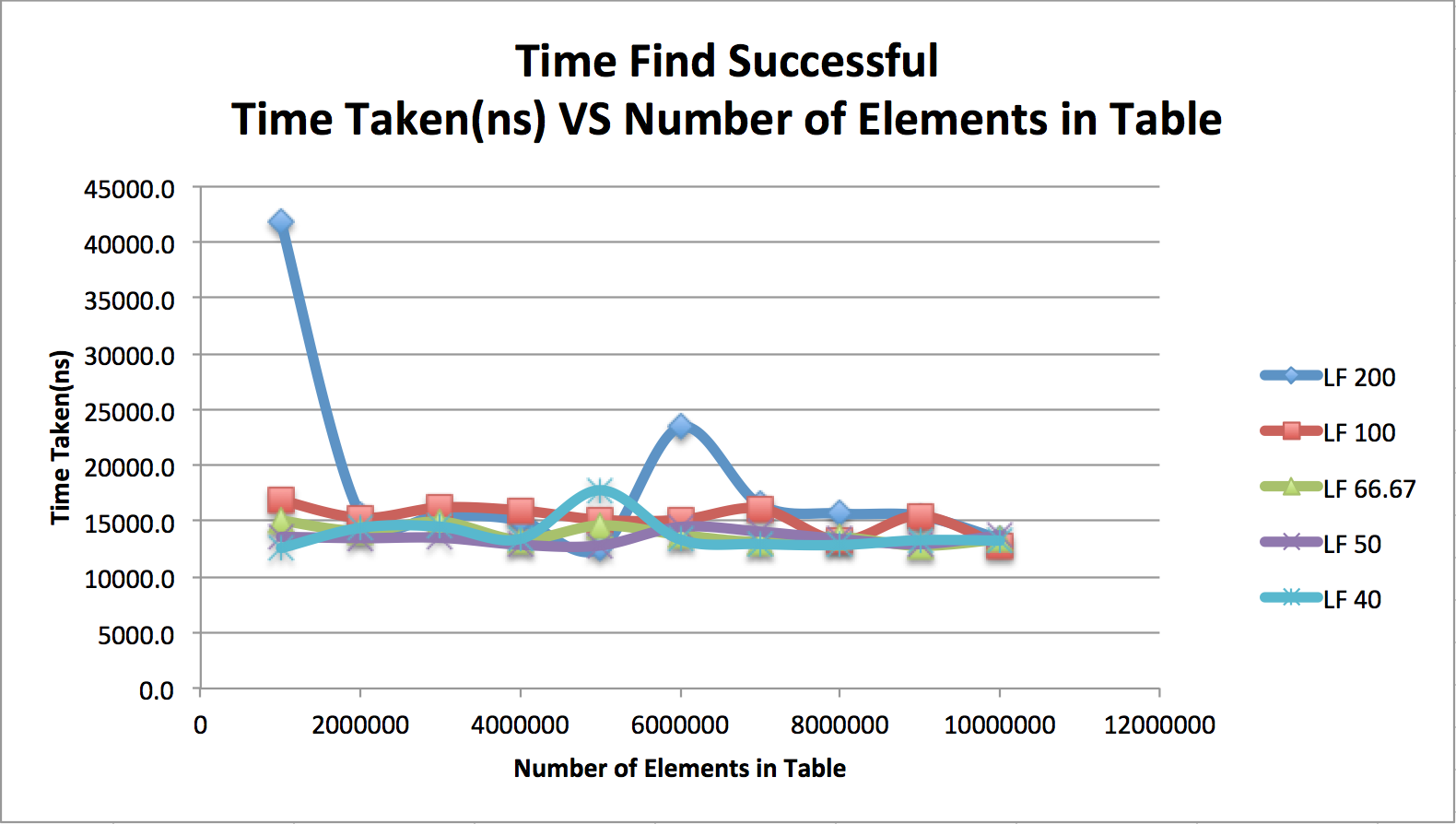
3. Methods

The setup of this program was broken down into different parts. A key aspect was the creation of methods that ran the experimental work. Each test was run in two stages, the first included a small test to ensure things worked as expected on a small scale, and then the second part was a more extensive test. The first test gave me a general sense of how long the program would take to run given the time taken for n elements. This small test also helped me get a general sense on what to expect for my more extensive tests. This time around, I knew I would not get a perfect fit on any of my data so I did not look for best-fit lines, but rather correlations between size and load factors. I decided to stick with the model I used from lab 2, in that my ExperimentController class had a main method which allows the user to specify what test to run, so not all tests must be run at the same time. This also gives the user, me in this case, the ability to easily change the number of trials executed and the seed. The parameters passed in, followed the format of the first argument, args[0], corresponding to the test I wanted to run, and the second argument, args[1], corresponding to the number of tests to conduct, the third argument, args[2] , represented the seed to be passed in for testing and args[3] was reserved for a true or false condition for the find test, if used for insert it was disregarded. The true or false condition was to tell the program whether or not we wanted the test to find a successful or unsuccessful test, respectively. It is important to note that I did not put in exceptions that will determine whether or not a valid string was passed in for the first argument, nor a check for non-integer values passed in for the second/third argument, nor a Boolean check for the last. If anything else is input other than this, then desired results may not occur.

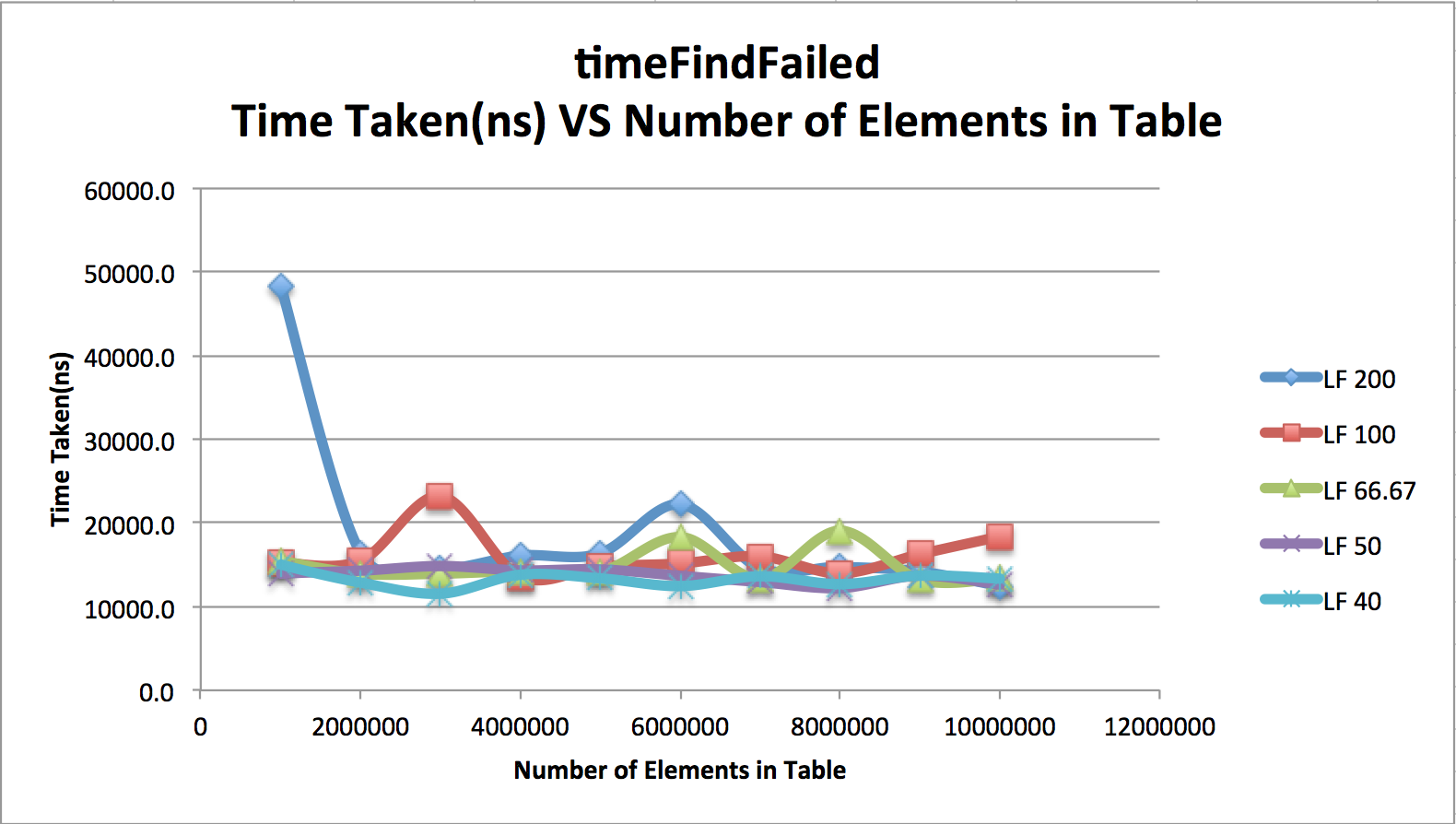
4. Data and Analysis



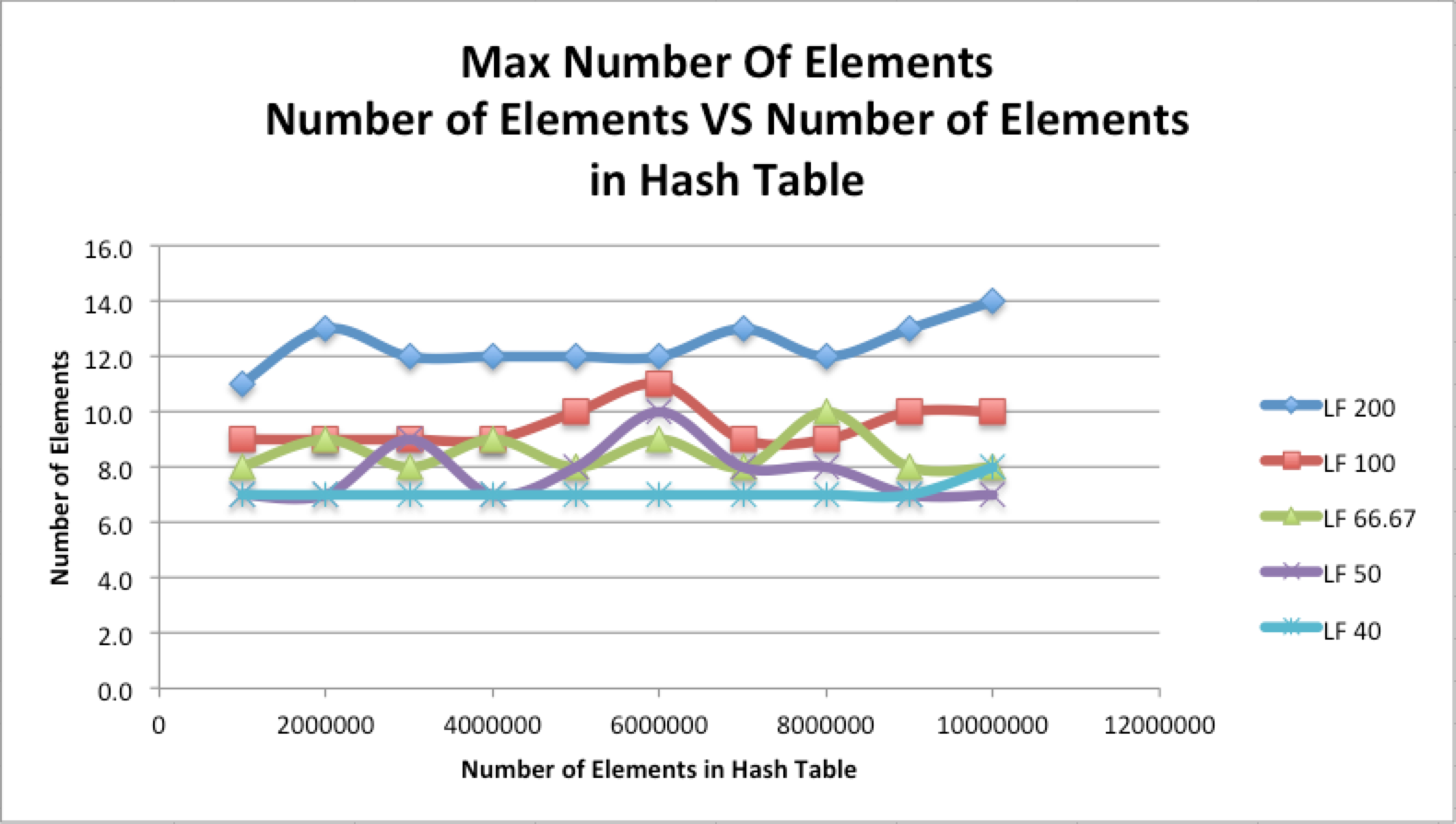
This graph represents the relationship between the time in ns to the number of elements inserted into a hash table. There are 5 sets of data on this one plot, corresponding to load factors of 200.0%, 100.0%, 66.67%, 50.0% and 40.0%. It was predetermined that the complexity for inserting elements should be the same complexity, as regardless of the load factor, the insert method should not take longer. Each element to insert has a key value, which is hashed giving me a corresponding index value, that index is used to access a LinkedList which adds the new element to the front of the LinkedList, which is O(1). Although it can be a little challenging to see if you look at all traces at once, it can be seen that each individual plot is roughly linear. This trend matches the complexity as the plots are used for n elements, thus making the complexity O(n).



This graph represents the relationship between the time taken to find an element and the number of elements in the table. Like in the prior plot there are 5 sets of data on this one plot, corresponding to load factors of 200.0%, 100.0%, 66.67%, 50.0% and 40.0%. It was predetermined that for a “good” HashTable, looking for an element should be constant time regardless of the number of elements. A “good” HashTable is determined based on whether or not the load factor is low or not. As shown by this graph, higher load factors are generally slower, although this is not always the case. It is important to notice the spike at the very beginning of the test for a load factor of 200%, this spike is very high; however, all test runs of this type gave me data relating to this so it can not be too strange. As shown by the graphical representation, the data points fluctuate; however, they all follow a generally linear model (with the notable exception of the first point for load factor of 200%). This should be linear as finding should be O(1) and we are looking for a single element.



This graph represents the relationship between the time taken to find an element and the number of elements in the table, this plot however is for an unsuccessful search. Like in the prior plot there are 5 sets of data on this one plot, corresponding to load factors of 200.0%, 100.0%, 66.67%, 50.0% and 40.0%. Since a “good” HashTable should have constant find time. As shown by this graph, higher load factors are generally slower, although this is not always the case. It is important to notice the spike at the very beginning of the test for a load factor of 200%, this spike is very high; however, all test runs of this type, just like the last test did, gave me data relating to this so it can not be too strange. As shown by the graphical representation, the data points fluctuate; however, they all follow a generally linear model (with the notable exception of the first point for load factor of 200%). This should be linear as finding should be O(1) and we are looking for a single element.



This graph represents the relationship between the maximum number of elements at any given index (contained in a LinkedList) and the number of elements total in the Hash Table. Like in the prior plot there are 5 sets of data on this one plot, corresponding to load factors of 200.0%, 100.0%, 66.67%, 50.0% and 40.0%. This data may not be the greatest in a sense of efficiency; however, it demonstrates the consistency of a Hash Table. There are minor fluctuations for each respective load factor, demonstrating that on average there are consistent maxima for a given load factor. This shows us that by decreasing our load factor we can in turn decrease the time taken to find an element, as there is more efficient indexing taking place which in turn means less to search through. Assuming all data is correctly constructed, the plots should be linear/horizontal in nature as shown because all load factors should have roughly the same maximum numbers of elements out of any index.

5. Conclusion

Through my experimental data, I have determined that if you have a low load factor then the computational time decreases. That is of course unless you make your table so big that the memory allocation is vast thus slowing down the program a bit. I have also determined that both finding an element successfully will have the essentially the same complexity and will require the same time to find. If we look at the graphs we can see this point represented graphically, the reason they are near identical is because in the worst case of the find methods successful run it will have to search through every Entry at it’s hashed index to find the correct one, and the unsuccessful test will also have to search through every Entry at it’s hashed index just the same; the only difference is that the successful test will return an Entry while the unsuccessful one clearly will not. Overall I would say hashing seems to be an amazing tool in speeding up a program and is definitely the most powerful thus far, in my opinion.

6. References

<https://moodle.lafayette.edu/pluginfile.php/151579/mod_resource/content/1/writeup-guidelines.pdf>

<http://www.cs.lafayette.edu/~liew/courses/cs150/lab/labs/lab04h/>

Data Structures and Problem Solving Using Java by Mark Allen Weiss