Tavi Tenari

3/21/17

Program 3 Report

LINUX COMPILE INSTRUCTIONS:

g++ -std=c++11 HashTable.cpp Listing.cpp Lab3.cpp

then, ./a.out

# Assumptions

To begin, I should clarify my assumptions about the assignment.

According to the assignment instructions…

*“In general, hash functions are generated experimentally, based on the structure and statistics of the data to be stored.”*

This was actually a very important concept to the approach of my design. I wasn’t sure about whether I should develop my hash functions to accommodate *any* phone number listings, or solely the listings given in the text file. But the above sentence told me that I should first analyze the entries in the file to find patterns in the given data, as the hash functions should be shaped specifically for the given data. So, **my first assumption was that the hash functions were going to be generated solely for the provided phonebook.txt.**

Secondly, I want to address that the assignment description did not say that the hash function was going to be used to retrieve the listings back out of the hash table. Our instructions were only to develop hash functions to store the entries of phonebook.txt. So, I had this funny idea that the quickest way to store the entries would be to just hash every entry to the first index of the hash table, so that Index[0] would have a chain of 46,332 entries. Separate chaining is only concerned with hashing to the index, and collisions are allowed, so this hashing function would be 0(1) if we are only talking about *storing* the data. But obviously, this doesn’t seem reasonable, so **my second assumption was that the has functions would be designed to potentially retrieve the listings efficiently if desired.**

In addition to the previous assumption, the assignment description implied that we should store each *line* of the phonebook, as opposed to a *segment* of the line. Originally, I assumed that we would be storing the phone numbers of the line, and then using the associated names as the hash parameter, mimicking a phonebook. But the instructions did not say to store either the number, or the name, but to merely store the contents of phonebook.txt. So **my third assumption was that the characters of the entire line were free to use for hashing purposes, and that the data of the entire line would be stored in an element of the hash table.**

# General Design

This section will not talk about the hashing, but just how data will be taken from phonebook.txt and manipulated.

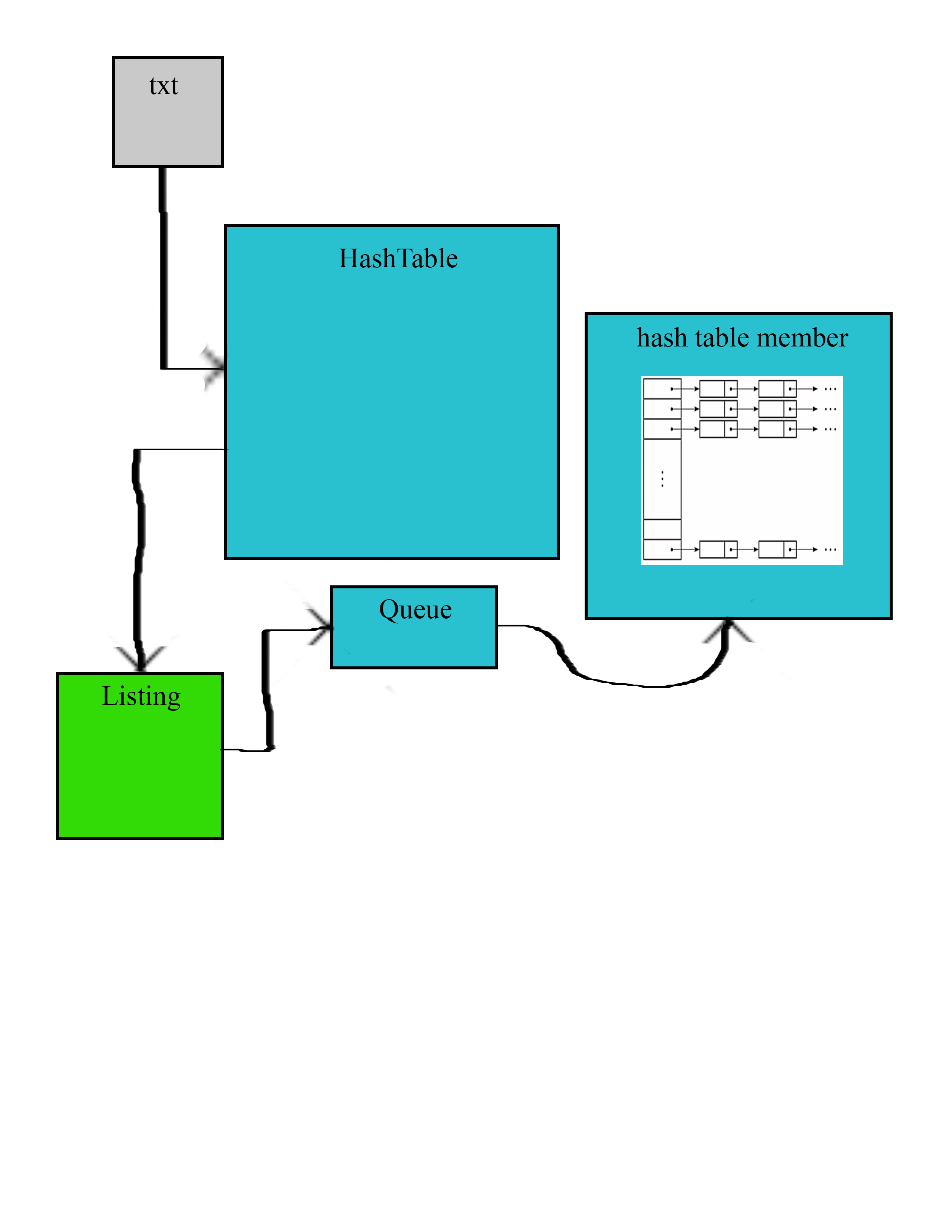
In CSS342, we were given multiple assignments which involved reading text from a file, and parsing it to pass as a parameter to somewhere else. Professor Dimpsey always told us never to pass strings around in such cases, but instead to construct a struct or object out of the data, and pass those around instead. So, my idea for this assignment was to create a Listing class, and use the information on each line to fill in the members of the Listing objects.

My idea for a Listing object was to have it store as separate members: first name, last name, area code, 7-digit number. This way, if I needed exclusive access to any of these strings, it would be more convenient than dealing with just one long string. And by the end of the assignment, this proved essential.

Also, since each line would need to be hashed separately, I decided to store the listings in a queue. Queues are efficient data structures because they do not involve indexing, and do not take up much memory, and I knew that I would not need to access anything in the middle. FIFO would be just fine for processing each listing.

I decided to have the HashTable class have the queue as a data member, and the actual hash table as a data member. And for convenience, I opted to use vectors as the “chain” structure, as opposed to list. It did not seem to be that important of a decision for this assignment, since we are just storing the entries, but I find vectors generally easier to work with, and in case of any strange testing situations, I would be able to easily access the middle of the chains as vectors include indexing.

Additionally, I designed the HashTable class to store all the txt reading functions and the hashing functions, so the driver merely creates the HashTable object. All functionality comes out of the HashTable class.



# Initial Hash Assumptions

Since phonebook.txt contains 46332 lines of entries, and our table is only size 4177, the calculated loading factor is 46332/4177 = ~11.09. This is not good, according to the lecture slides, as with separate chaining, it’s good to aim for a loading factor of 1 or less.

So, the most efficient hash will be the one with the least collisions. And in this case, this means that the ideal hash function will produce a hash table with 4177 chains of ~11.09 length. **Since this number isn’t whole, the best function will have chains between lengths 11 and 12.** So the efficiency of this table will ideally be O(1) + O(11), as it should be constant time to access the index, and then O(n) to iterate through the chain, where n is the length of the chain.

# First Hash Function

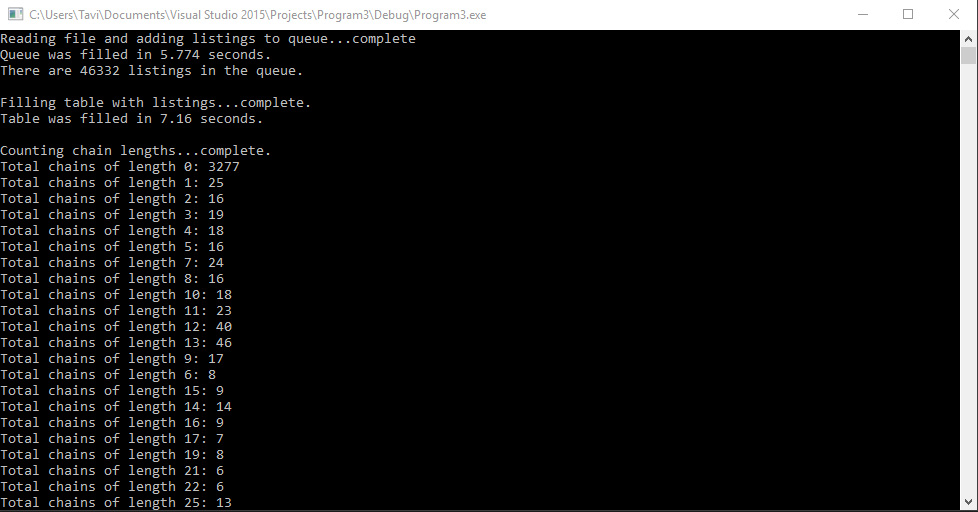
Upon looking at the entries in the phonebook, I noticed that multiple phone numbers were associated with the same name. It also appeared that difference names had the same number. So, I figured that if I were to use either only the name, or only the phone number, the hash would produce many collisions.

The most obvious thing about the entries was that no combination of name and number were the same. For instance, there was only 1 Zoe Zacks with the number (425) 427-2267. Therefore, an obvious hash idea was to add up the alpha-numeric values of the entire line of data (A=0, B=1, etc.), and mod that sum to fit into the indexes of the table.

The pseudocode for this algorithm:

int(first name + last name + area code + phone number) % 4177

Upon testing this code, I found the following results:



So this hash function turned out to be terrible. 3277 indexes were empty, and the indexes that were filled got a wide range of chain lengths from 1 to over 250 listings long. The hash took an average of 7.05 seconds to store the listings in the hash table, so it will be interesting to compare this run time to the other hash functions, as this algorithm seems very simple.

# Second Hash Function

Since the first algorithm performed so poorly, **I decided to analyze the data file to find much more unique aspects of each of the entries. This should reduce so many collisions to one area.** Upon analysis, I discovered interesting things about the file:

Out of 46,332 listings, there were 234 unique names. Of those 234 names, there were 26 unique first names and 9 unique last names. The first names started with letters A-Z, and the last names started with A, B, E, G, J, L, S, T, Z. The file also contained 198 unique phone numbers. Out of those 198 unique numbers, only 2 different area codes existed, along with 99 unique 7-digit numbers.

This information proved to be very useful. It turned out that the 198 unique numbers repeated for each unique name, so every unique name is associated with every unique number.

And since 99 unique numbers have the (206) area code and the (425) area code, and I wanted chain-lengths of roughly 11, I had the idea to try and split the listings into 9 blocks of 11 per each area code. The 3rd digit of the 7-digit number iterates after blocks of 11 from 3, 4, 5, 6, 7, 8, 9, 0, 1. So 99 blocks of numbers can be split into 9 blocks of 11. So with this logic, I was able to develop a hash function that filled the hash table horizontally in alphabetical order, in chains of 11. The hash fills the table from 0-4177 in 9 blocks of 11 in the order (last name initial, first name initial) AA, AB, AC, AD,…,AZ, BA, BB, BC, BD,…,ZZ.

Pseudocode:

[(18 \* the first initial of the first name) +

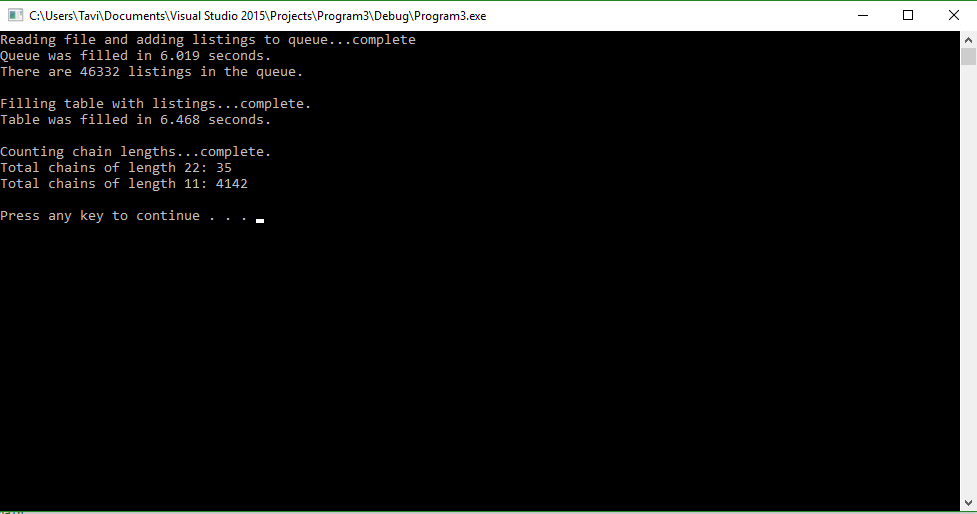
(468 \* the sequence position of the first initial of the last name) +

(the sequence position of the third digit of the phone number) +

(the sequence position of the first digit of the area code)] % 4177

This special algorithm is based specifically on the way that the numbers repeated for each name, and that the names descended in alphabetical order. And of course, this algorithm is entirely deterministic, and will always index ZZ to the same index, regardless of what order it appears in the file.

Results:



These results were very close to ideal! Not only did this algorithm beat the simple algorithm of the first hash function by nearly a whole second, but 4142 chains were of length 11!

The only issue here is that 4177 entries aren’t enough to hold AA-ZZ of blocks of 11, so after index 4176, the algorithm started back at the beginning of the table again, and started indexing the remaining ZY and ZZ numbers at indexes 0-34. This caused the first 35 indexes to have chains of length 22, double the ideal amount. This seems to be a negligible hick-up, since it’s only 35/4177 indexes of length 22, which isn’t terribly long.

But I realized that if I developed an algorithm which filled the table vertically, then it would keep the chains minimally short as it loops around after index 4176 is filled.

# Third Hash Function

Analyzing the data again, I developed this pseudocode:

[(the sequence position of the first initial of the last name \* 5148) +

(the first initial of the first name \* 198) +

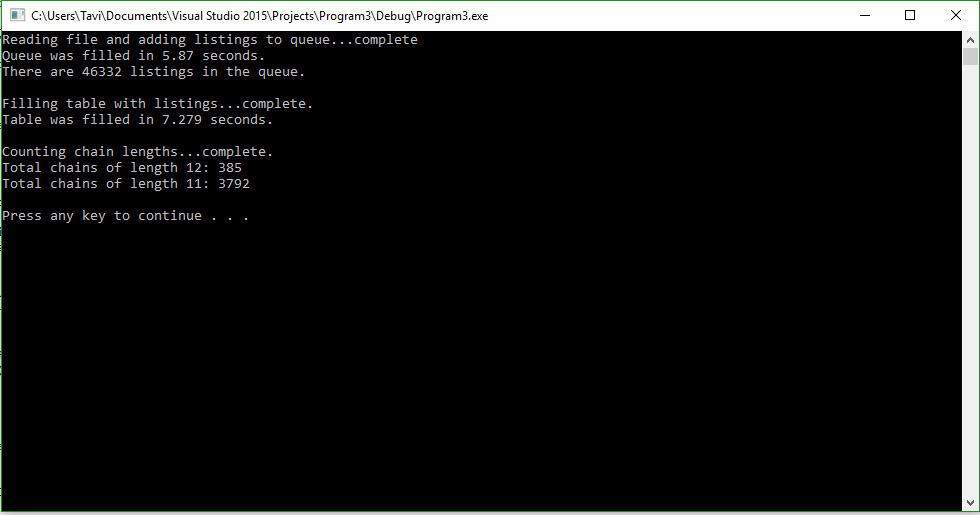
(the sequence position of the first digit of the area code \* 99) +

(the sequence position of the third digit of the phone number \* 11) +

(the sequence position of the sum of the last two digits of the phone number)] % 4177

The logic here is that there are 11 numbers per 3rd digit of unique 7-digit numbers, 99 numbers per area code, 198 numbers per unique first name, and 5148 numbers per unique full name. The key was to make sure that I associated all last names that begin with A with the first 5148 indexes (4177 + 971), and fill the table vertically in order.

Results:



This hash function produces a very very ideal situation. All chain lengths are between 11 and 12, with 3792 chain lengths of length 11, and only 385 of length 12. However, the algorithm requires an additional operation than the preceding operation, which may be the reason that this took a second longer to store all the values than the second hash function. But with a guaranteed O(constant) + either O(11) or O(12) to iterate through a chain when searching the table, this was my pick for the most attractive hash function.

# Testing Strategy

To test my hash functions, I implemented a display function which displays the contents of the table to the console, along with the associated chain lengths, so I could see exactly which name was being assigned to which index. This implementation is called in the driver, so you can see how I verified that the hash functions worked correctly.

\*For some reason, in linux, the time counting in the code produces inaccurate numbers of seconds. The times reported in this report were gotten from running the program on windows.