CSC2001F: Assignment 3
Mahmoodah Jaffer – JFFMAH001
3 April 2019

Object Oriented Design

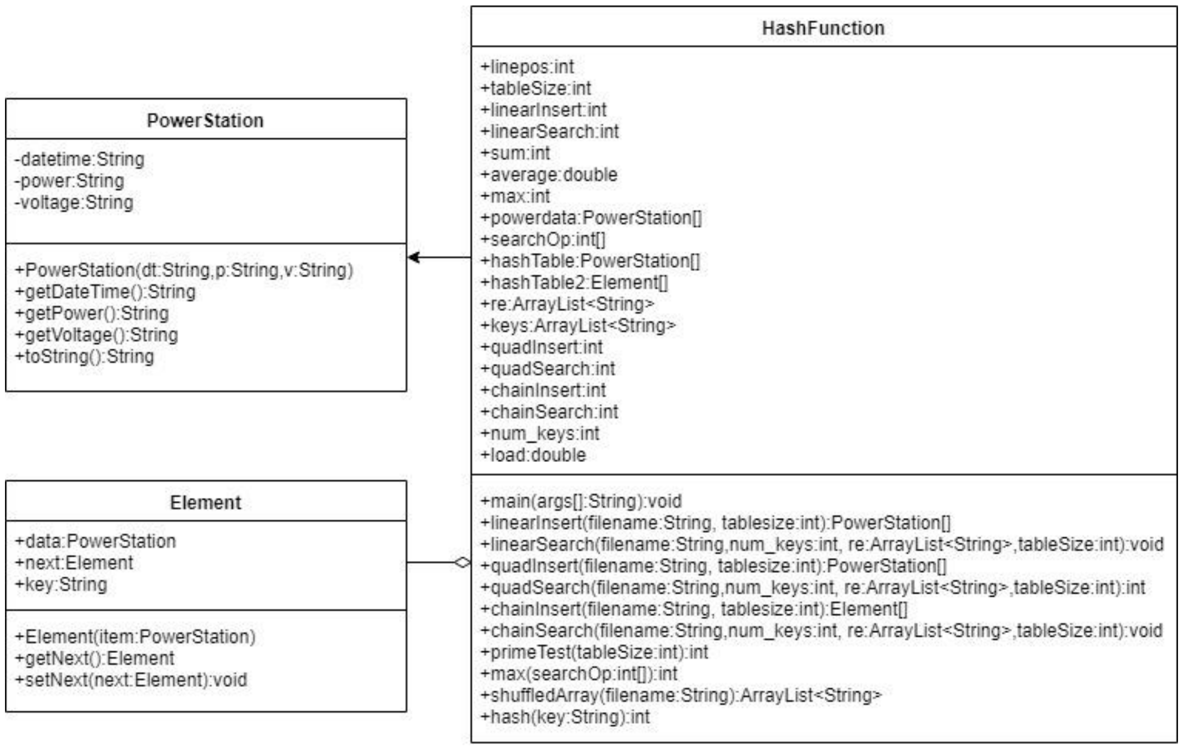

HashFunction
+linepos:int
+tableSize:int
+linearInsert:int
+linearSearch:int
+sum:int
+average:double
+max:int
+powerdata:PowerStation[]
+searchOp:int[]
+hashTable:PowerStation[]
+hashTable2:Element[]
+re:ArrayList<String>
+keys:ArrayList<String>
+quadInsert:int
+quadSearch:int
+chainInsert:int
+chainSearch:int
+num_keys:int
+load:double
+main(args[]:String):void
+linearInsert(filename:String, tablesize:int):PowerStation[]
+linearSearch(filename:String,num_keys:int, re:ArrayList<String>,tableSize:int):void
+quadInsert(filename:String, tablesize:int):PowerStation[]
+quadSearch(filename:String,num_keys:int, re:ArrayList<String>,tableSize:int):int
+chainInsert(filename:String, tablesize:int):Element[]
+chainSearch(filename:String,num_keys:int, re:ArrayList<String>,tableSize:int):void
+primeTest(tableSize:int):int
+max(searchOp:int[]):int
+shuffledArray(filename:String):ArrayList<String>
+hash(key:String):int
PowerStation
-datetime:String
-power:String
-voltage:String
+PowerStation(dt:String,p:String,v:String)
+getDateTime():String
+getPower():String
+getVoltage():String
+toString():String
Element
+data:PowerStation
+next:Element
+key:String
+Element(item:PowerStation)
+getNext():Element
+setNext(next:Element):void

Figure 1: UML Diagram for Hash Function

The class **PowerStation** was used to store the date/time, active power and voltage of each line in the CSV into one object. The **HashFunction** was used to insert **PowerStation** objects into a hash table and search for K number of values using a specified collision scheme. The **hash()** method was used to get the index value of each element in the hash table. **linearInsert** was used to insert the PowerStation objects into the hash table using the linear collision scheme. **linearSearch** was used to search for K number for keys in a shuffled array. A similar method was used to insert and search for both quadratic probing. For chaining, the hash table storing the elements were stored almost like a linked list in the way that if many date/time values had the same index in the hash table then each node at the same index pointed at the next element until the last node was pointing to a null address. **primeTest()** was used to determine if the table size that was input by the user was a prime number since collision schemes work optimally when the table size of the hash table is a prime number [1]. **max()** determines what the maximum search probe was for an entire search sequence for any collision scheme. **shuffledArray()** uses the **Collections.shuffle()** method to shuffle the array of keys. **Element** is an inner class inside the **HashFunction** class that defines the nodes of the linked list in the hash table for chaining.

Goal and Execution of Experiment

The goal of the experiment is to determine which hashing scheme has the best insert/search cost in terms of probing. We are also going to determine what the total, average and maximum number of search probes are for each hashing scheme. The load factor of the hash table should be proportional to the number of probes, so this will be examined as well.

The experiment was executed by creating a java program called **HashFunction**. This had a method called hash that was used as the hash function for all three hashing schemes. **HashFunction** also contained methods for the insertion and searching of elements in the hash table for each scheme. Four parameters were taken in from the user: 1) Table Size 2) Collision scheme 3) Input file 4) Number of keys. The table size inputted by the user was tested for being a prime number, as hashing schemes are optimised if the table size is a prime number. The file "cleaned_data.csv" was taken in to obtain the data for the hash table. The number of insertion probes and search probes were counted for the chosen hashing scheme. These were used to determine the total, average and maximum number of probes. Each collision scheme was tested for 5 different table sizes so that the data could all be compared. All testing was done using the program **HashTesting** which contained the same code as **HashFunction** with a few alterations. The data obtained was then used to generate various graphs, that will be shown and analysed further on in the report. These graphs can help us make an informed decision about which hashing scheme is best and why it is the best.

Results

Insertion

As can be seen in Figure 2 below, as the load factor of the hash table increases the total number of insertion probes increases a lot for both linear and quadratic probing. This result makes sense because if the load factor of the table increases it means that the hash table is fuller. If the hash table is fuller, it means that more collisions are going to take place thereby increasing the number of insertion probes required to insert the data into the hash tables. The number of insertion probes for linear probing increases almost exponentially. We can see that while quadratic probing also increases at an almost exponential rate, the rate at which it increases is lower than that of linear probing. The number of insertion probes for chaining remains relatively low as the load factor increases when compared to linear and quadratic probing. The number of insertion probes for chaining at a load factor of 0.765 seems a bit lower than it should be. This could be because of the way that the values were inserted – maybe less linked lists were formed when inserting into the hash table thereby decreasing the number of traversals required, which decreases the probe count.

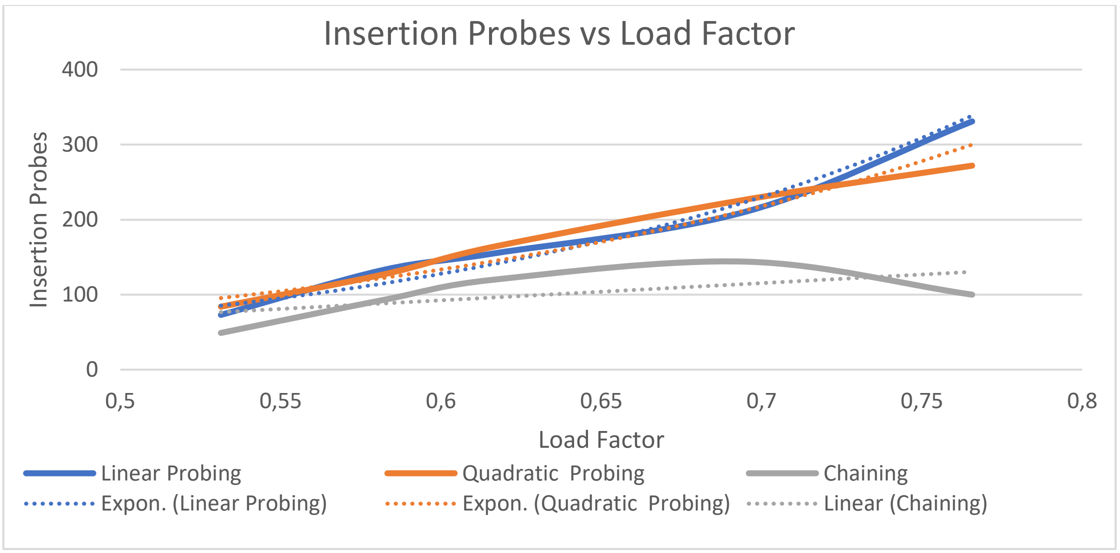


Figure 2: Graph showing the total number of insertion probes versus the load factor for each hashing scheme

Searching

Total Number of Search Probes

When searching the hash table for data elements the number of keys that were being searched for was 400. From Figure 3 below, we can see that the total number of searches for linear probing increases exponentially as the load factor of the hash table increases. The total number of searches also increases exponentially for quadratic probing but at a slower rate than that of linear probing. The total number of search probes for chaining remains relatively low. As can be seen from Figure 3, for linear probing the total number of search probes is higher than that of quadratic probing. This is to be expected since the collision resolution for quadratic probing is more efficient than the collision resolution for linear probing. The collision resolution for quadratic probing is more efficient than the collision resolution for linear probing since the hash values for quadratic probing are more spread out therefore resulting in fewer collisions. The total number of search probes for chaining is much lower than that of linear and quadratic probing. This is because for chaining, when a date/time key has the same index as another date/time it only searches the linked list at that index, whereas for linear and quadratic probing if a date/time has the same index as a date/time that does not match the date/time being searched, then the collision resolution is applied until an empty index for that date/time is found.

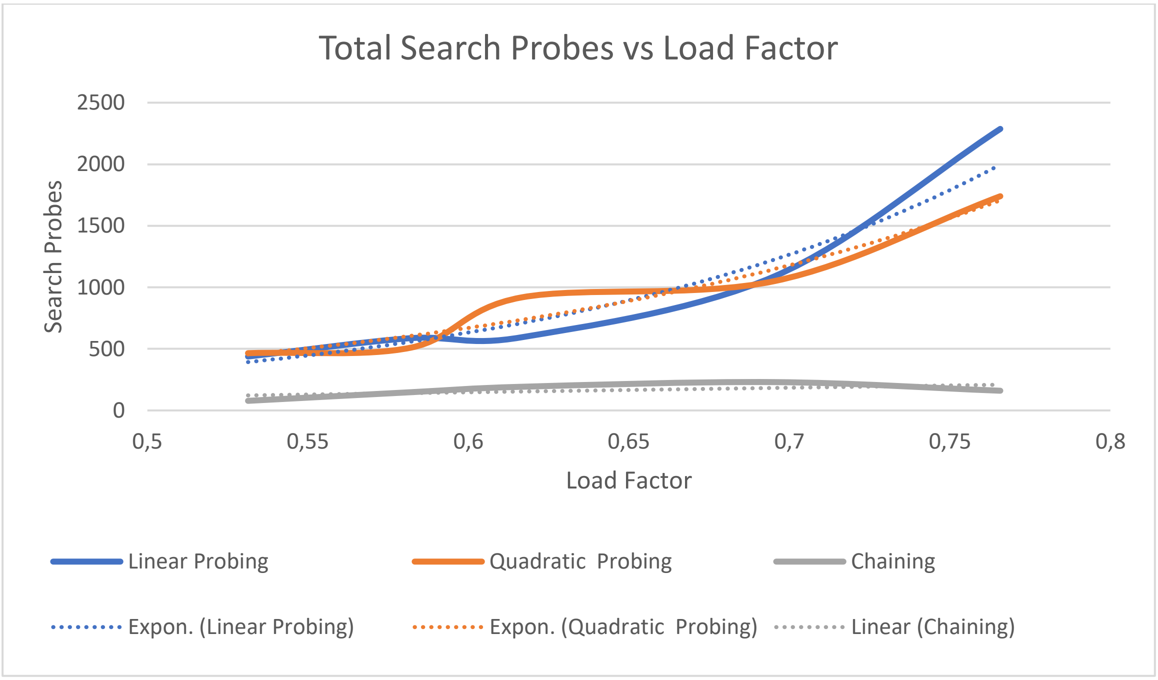


Figure 3: Graph showing the total number of search probes versus load factor for each hashing scheme

Average Number of Search Probes

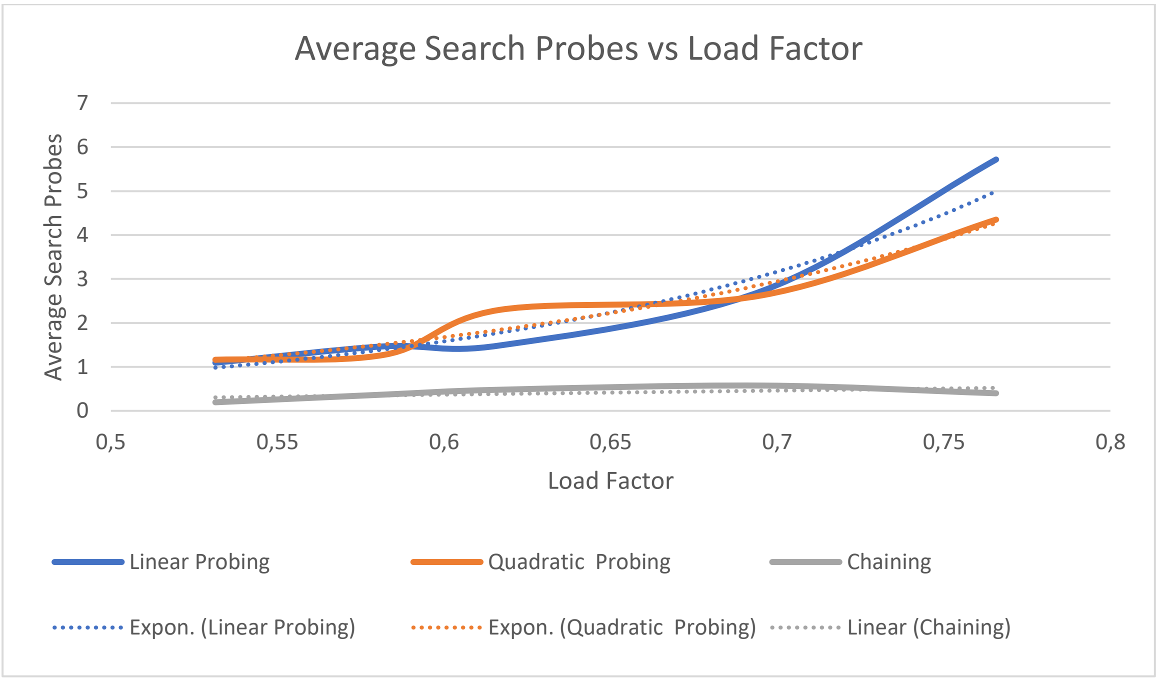


Figure 4: Graph showing the average number of search probes versus load factor for each hashing scheme

As can be seen from Figure 4 above, the number of average probes for chaining remains relatively linear and we can see that the average number of search probes is under 1 for each of the various load factors. This is to be expected since the total number of search probes for chaining is also quite low. The relationship for the average number of search probes versus the load factor for linear probing is still exponential. This is because the fuller that the hash table is the more collisions that will occur. As we can see, the relationship for quadratic probing closely resembles that of an exponential relationship. The exponential rate for quadratic probing is slower than the rate for linear probing. The average number of search probes for linear and quadratic probing both exceed one. This is because the total number for search probes for each table size is more than the number of elements being searched for in the hash table. The graph shown in Figure 4 is nearly identical to the graph shown in Figure 3 – the only difference being the scale against which the graph is being plotted.

Maximum Number of Search Probes

The maximum number of search probes for chaining is quite low and the relationship between number of search probes and load factor is relatively linear. We can see that the maximum number of search probes for linear probing is quite high – this tells us that using linear probing as a hashing scheme causes many collisions in the hash table. The number of search probes for quadratic probing between the table size of 0.6 and 0.65 looks like an outlier when compared to the rest of the maximum search probes. This is probably due to the order of the data elements inside the shuffled array used for that table size since a different random array of keys was used for each table size. The maximum number of search probes for quadratic probing for most table sizes are lower than that of linear probing therefore indicating that quadratic probing has less collisions than linear probing.

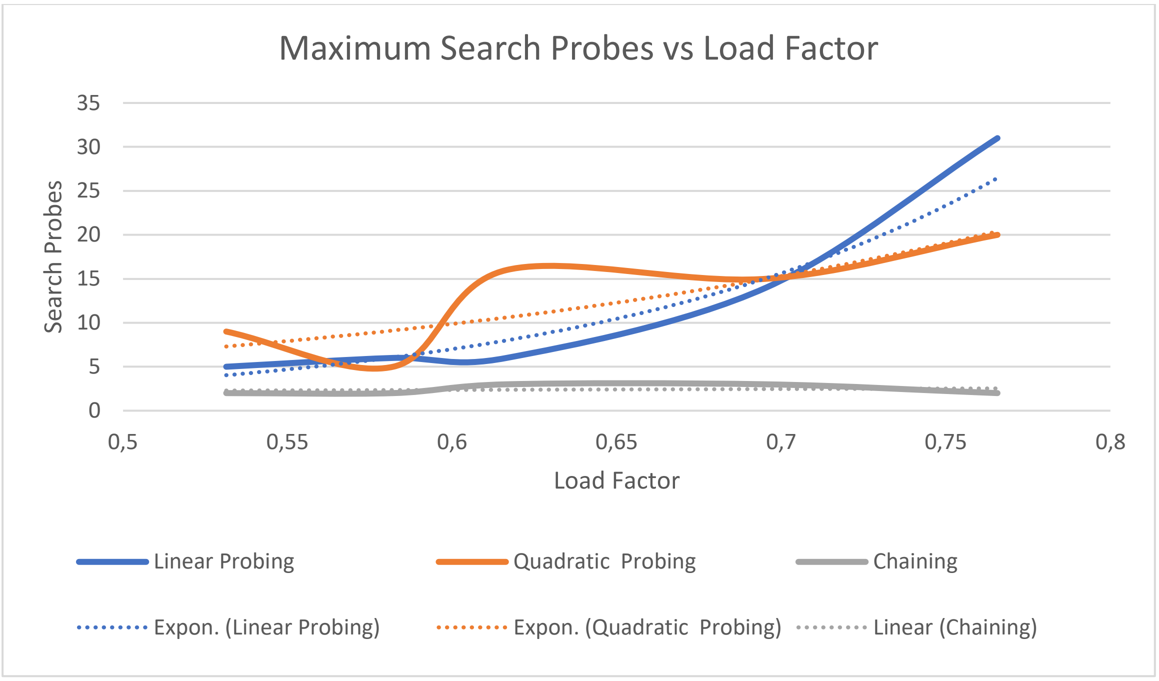


Figure 5: Graph showing the maximum number of search probes versus load factor for each hashing scheme

Other Comments

- As can be seen from the above graphs, the load factor for all hashing schemes are the same. However, this is not always the case. The equation used to calculate load factor is:
$$\lambda = \frac{Number\ of\ Elements}{Table\ Size}$$
The optimal load factor for linear probing is 0.75. The optimal load factor for quadratic probing is 0.5 because if the hash table is half full and the table size is a prime then the same cell in the hash table will never be checked twice and we can always insert a new element. The load factor for chaining is generally more than one and this is because there are usually multiple elements stored in a single hash table cell. However, in the case of this assignment we were constrained to choose table sizes in the range of 653-1009 which are all more than the number of elements we are inserting into the hash table. It is because of this that the load factor for chaining is not more than 1.
- Although less collisions occur for quadratic probing when compared to linear probing, the cost of the operations is higher when using ***** and **%** operators. These costs can be offset by using bitwise shifting and subtraction.

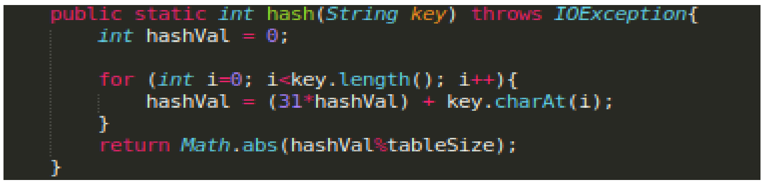
Final Analysis

From the above results, we can see that the best hashing scheme in terms of insertion/search cost is chaining. Although chaining requires more storage than linear and quadratic probing, in terms of time and computational efficiency it is better than linear and quadratic probing. Chaining has a lower cost of operations than linear and quadratic probing since it uses a linked list instead of operations such as +, -, * and %. The reason why the search and insertion cost of chaining is better than that of linear and quadratic probing is because when the index created for a certain date/time key is the same as the index created for another date/time key then only the linked list at that index needs to be traversed, whereas with linear and quadratic probing an empty cell is looked for in the hash table every time it encounters a collision. Between linear and quadratic probing, quadratic probing is better than linear probing because the number of probes for quadratic probing increases at a lower exponential rate than that of linear probing.

Creativity

I created a UML Diagram to assist with the understanding of the way the classes I created were used. I ensured that if a file had duplicate entries and it was used as the input file for the **HashFunction** that each duplicate would only be put into the hash table once (This is different to checking for duplicate keys because 2 different entries could still have the same key). I also used another program **HashTesting** as mentioned above to test my program.

The main point of this assignment was to compare the insert/search cost in terms of probes for each hashing scheme. While doing this assignment it became apparent that the insert/search cost does not only depend on which hashing scheme you use but also on how efficient your hash function is. The hashing function that I used for the assignment was as follows:

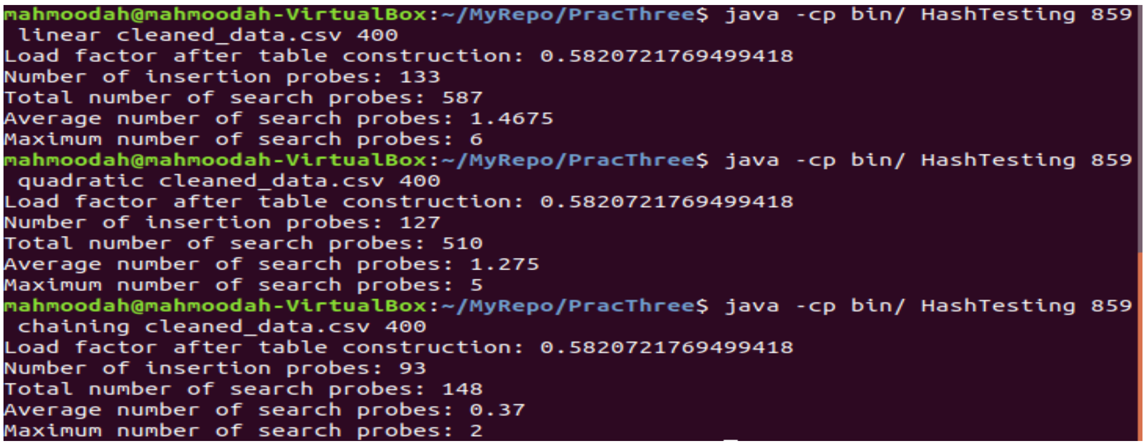


```java
public static int hash(String key) throws IOException{
    int hashVal = 0;

    for (int i=0; i<key.length(); i++){
        hashVal = (31*hashVal) + key.charAt(i);
    }
    return Math.abs(hashVal%tableSize);
}
```

Figure 6: Efficient hash function

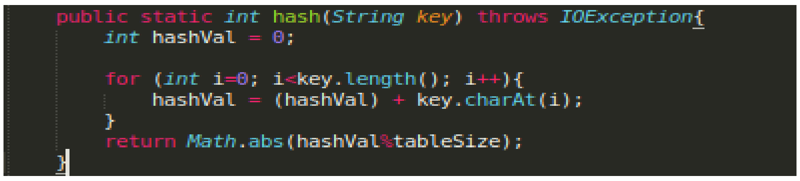
The hashing function that I used multiplies the sum of the Unicode values of each character of the date/time string by a prime number 31. This increases the hash values as well as dealing with the possibility of having the same Unicode values for different date/time values – it deals with this by shifting each character by 31, therefore producing more unique key values which in turn causes a more spread out hash function. The output produced using the above hash function was as follows:



```
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ java -cp bin/ HashTesting 859
 linear cleaned_data.csv 400
Load factor after table construction: 0.5820721769499418
Number of insertion probes: 133
Total number of search probes: 587
Average number of search probes: 1.4675
Maximum number of search probes: 6
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ java -cp bin/ HashTesting 859
 quadratic cleaned_data.csv 400
Load factor after table construction: 0.5820721769499418
Number of insertion probes: 127
Total number of search probes: 510
Average number of search probes: 1.275
Maximum number of search probes: 5
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ java -cp bin/ HashTesting 859
 chaining cleaned_data.csv 400
Load factor after table construction: 0.5820721769499418
Number of insertion probes: 93
Total number of search probes: 148
Average number of search probes: 0.37
Maximum number of search probes: 2
```

Figure 7: Output when using the efficient hash function

I edited the function shown in Figure 6, so that the hash value was not multiplied by any number, as seen below:



```java
public static int hash(String key) throws IOException{
    int hashVal = 0;

    for (int i=0; i<key.length(); i++){
        hashVal = (hashVal) + key.charAt(i);
    }
    return Math.abs(hashVal%tableSize);
}
```

Figure 8: Inefficient hash function

The output for the inefficient hash function in Figure 8 can be seen below:

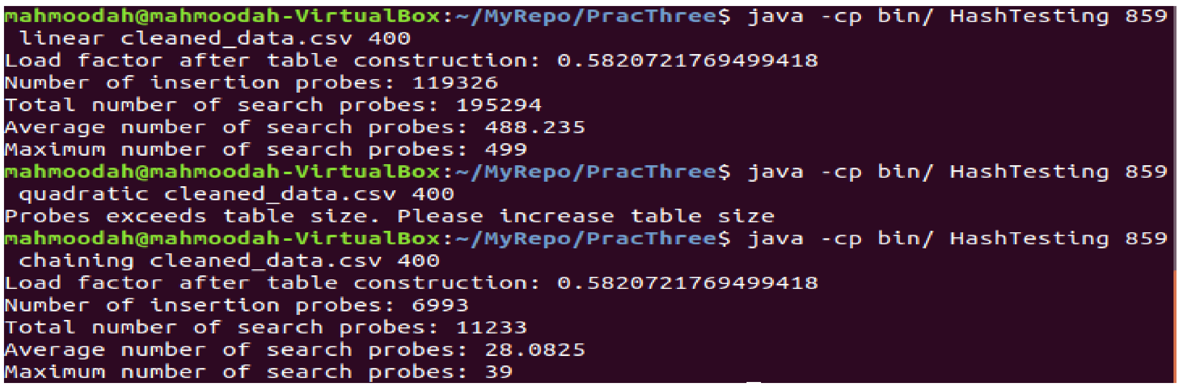
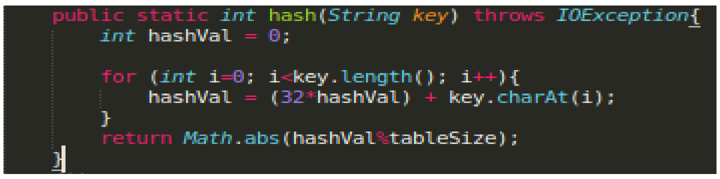

```
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ java -cp bin/ HashTesting 859
 linear cleaned_data.csv 400
Load factor after table construction: 0.5820721769499418
Number of insertion probes: 119326
Total number of search probes: 195294
Average number of search probes: 488.235
Maximum number of search probes: 499
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ java -cp bin/ HashTesting 859
 quadratic cleaned_data.csv 400
Probes exceeds table size. Please increase table size
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ java -cp bin/ HashTesting 859
 chaining cleaned_data.csv 400
Load factor after table construction: 0.5820721769499418
Number of insertion probes: 6993
Total number of search probes: 11233
Average number of search probes: 28.0825
Maximum number of search probes: 39
```

Figure 9: Output when using the inefficient hash function

As can be seen from Figure 9 above, when we don't optimise the hash function the number of insertion and search probes increases tremendously. This is because the characters are no longer shifted by any number and therefore more date/time keys are similar therefor resulting in more collisions. Multiplying the hash value by a non-prime number affects the number of insertion/search probes as well – this hash function can be seen in Figure 10 below:


```java
public static int hash(String key) throws IOException{
    int hashVal = 0;

    for (int i=0; i<key.length(); i++){
        hashVal = (32*hashVal) + key.charAt(i);
    }
    return Math.abs(hashVal%tableSize);
}
```

Figure 10: Possible hash function

The output produced by this hash function is as follows:

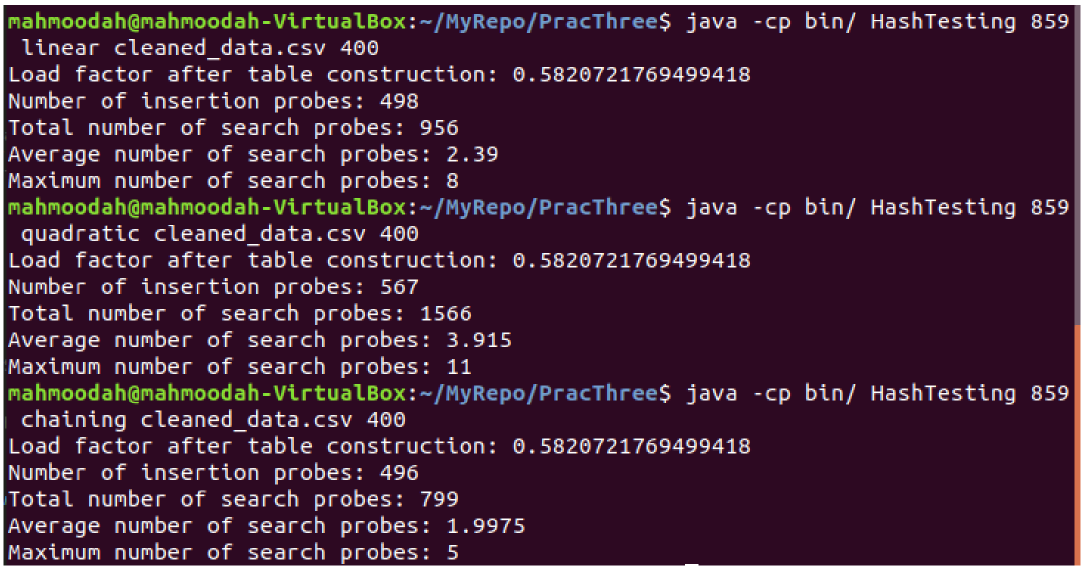
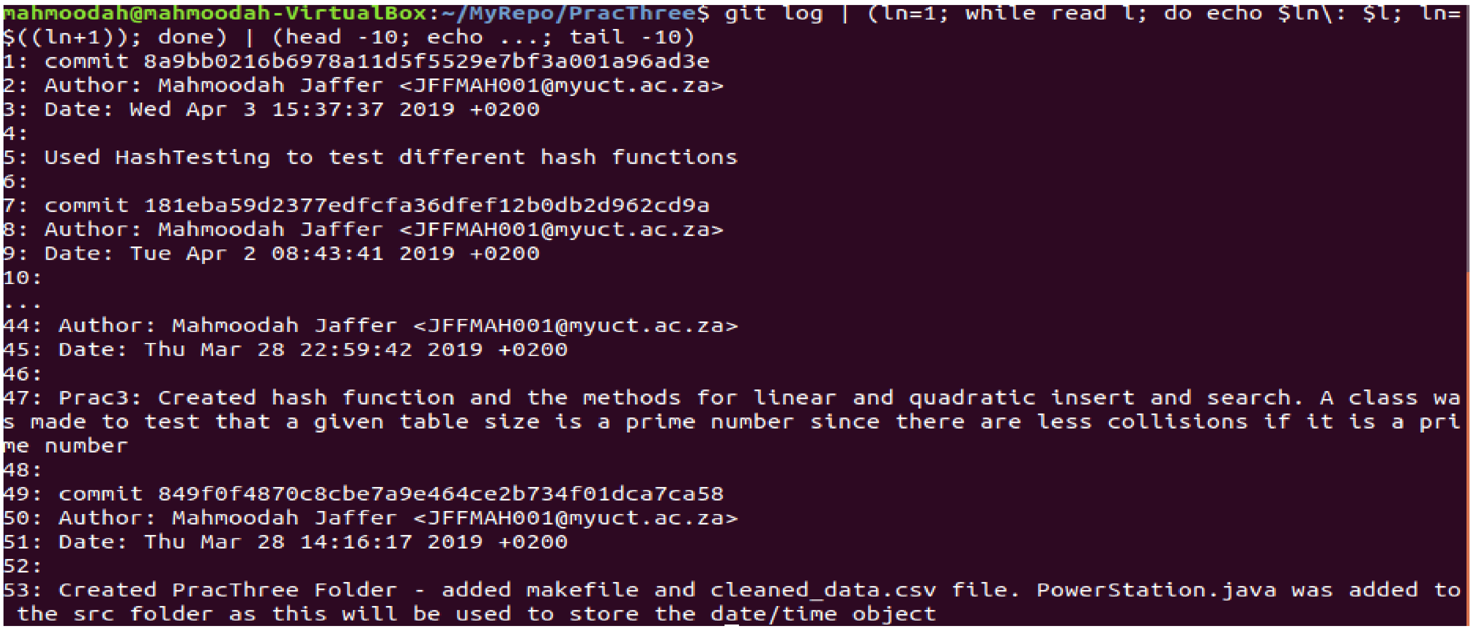

```
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ java -cp bin/ HashTesting 859
 linear cleaned_data.csv 400
Load factor after table construction: 0.5820721769499418
Number of insertion probes: 498
Total number of search probes: 956
Average number of search probes: 2.39
Maximum number of search probes: 8
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ java -cp bin/ HashTesting 859
 quadratic cleaned_data.csv 400
Load factor after table construction: 0.5820721769499418
Number of insertion probes: 567
Total number of search probes: 1566
Average number of search probes: 3.915
Maximum number of search probes: 11
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ java -cp bin/ HashTesting 859
 chaining cleaned_data.csv 400
Load factor after table construction: 0.5820721769499418
Number of insertion probes: 496
Total number of search probes: 799
Average number of search probes: 1.9975
Maximum number of search probes: 5
```

Figure 11: Output when using the hash function with an even number

As we can see that while the number of probes for the hash function in Figure 10 is not as much as the number of probes for the hash function in Figure 9, it is still not as efficient as the hash function I used shown in Figure 8.

Git Log



```
mahmoodah@mahmoodah-VirtualBox:~/MyRepo/PracThree$ git log | (ln=1; while read l; do echo $ln\: $l; ln=
$((ln+1)); done) | (head -10; echo ...; tail -10)
1: commit 8a9bb0216b6978a11d5f5529e7bf3a001a96ad3e
2: Author: Mahmoodah Jaffer <JFFMAH001@myuct.ac.za>
3: Date: Wed Apr 3 15:37:37 2019 +0200
4:
5: Used HashTesting to test different hash functions
6:
7: commit 181eba59d2377edfcfa36dfef12b0db2d962cd9a
8: Author: Mahmoodah Jaffer <JFFMAH001@myuct.ac.za>
9: Date: Tue Apr 2 08:43:41 2019 +0200
10:
...
44: Author: Mahmoodah Jaffer <JFFMAH001@myuct.ac.za>
45: Date: Thu Mar 28 22:59:42 2019 +0200
46:
47: Prac3: Created hash function and the methods for linear and quadratic insert and search. A class wa
s made to test that a given table size is a prime number since there are less collisions if it is a pri
me number
48:
49: commit 849f0f4870c8cbe7a9e464ce2b734f01dca7ca58
50: Author: Mahmoodah Jaffer <JFFMAH001@myuct.ac.za>
51: Date: Thu Mar 28 14:16:17 2019 +0200
52:
53: Created PracThree Folder - added makefile and cleaned_data.csv file. PowerStation.java was added to
 the src folder as this will be used to store the date/time object
```

Figure 11: Summary statistics for git log usage

References

[1] https://www.javatpoint.com/prime-number-program-in-java