CSC2001F: Assignment 3

Leanne January

05 April 2019

Contents

1 Introduction **1**

1.1 Aim . . . 1

1.2 Execution . . . 1

2 Explanation of OO Design **2**

2.1 Power Usage Class . . . 2

2.2 Linear Hash Class . . . 2

2.3 Quadratic Hash Class . . . 2

2.4 Chaining Hash Class . . . 2

2.5 Power Hash Class . . . 3

3 Analysis of Experiment Results **4**

3.1 Insert Performance . . . 4

3.2 Search Performance . . . 5

4 Statement of Creativity **9**

5 Git Log **9**

1 Introduction

A Hash Table is a data structure that is more efficient in inserting, retrieving and deleting data. The main components of a Hash Table are (i) an array to store the data in, and (ii) a hash function which maps a data item to an index in the array using a key. A Hash Table is more efficient than other data structures as it supports constant time insertion, retrieval and deletion of data. This is due to the fact that the hash function can calculate, in most cases, the exact position of a key and insert, find or delete it in a small number of operations. The issue with this is that sometimes data items or keys are hashed to the same index position which results in what is called a collision in the hash table. Therefore, a Hash Table needs to have a method of dealing with collisions – this is called a collision resolution scheme. A collision scheme can use either an open addressing or a closed addressing method to resolve collisions. An open addressing method involves looking for (or probing) other empty indexes to find a new location to insert the data item where there will be no collisions. The linear probing method of open addressing looks for the next closest empty index after the index where the collision occurred – and, in this way, increases in steps of one. The quadratic probing method of open addressing is like the linear probing method except its steps increase quadratically. A closed addressing method stores all the values that are hashed to a particular index position at that index position. The separate chaining method does this by storing a linked list at each index that contains all the data items that were hashed to that particular index

1.1 Aim

The aim of this experiment is to determine how these different collision schemes behave in hash tables of varying fullness, and which scheme has the best performance.

1.2 Execution

I executed this experiment by implementing three different hash tables which each use one of these collision schemes. These hash tables all used the following hash function to map a string into an integer:

```
public int hashFunction(String key){
int hashVal = 0;

for( int i = 0; i < key.length(); i++ ) {
hashVal = (37 * hashVal) + key.charAt(i);
}
return Math.abs(hashVal % tableSize);
}
```

The code for each hash table was instrumented to measure how many probes it took to insert and retrieve data for tables of different sizes.

The load factor for each table was also recorded. The load factor is the probability that a cell is empty and is measured by dividing the number of items present in the table by the size of the table. The load factor is therefore directly proportional to the fullness of the table. The more items there are in a table, the higher the load factor. This means that a table with a high load factor is less likely to contain cells that are empty. More collisions are, therefore, expected for hash tables with a higher load factor.

2 Explanation of OO Design

2.1 Power Usage Class

The PowerUsage class stores the Date/Time, Power and Voltage values from the cleaned_data.csv files as 3 different Strings within one object.

2.2 Linear Hash Class

The linearHash class implements a Hash Table that uses linear probing as its collision scheme.

2.3 Quadratic Hash Class

The quadraticHash class implements a Hash Table that uses quadratic probing as its collision scheme.

2.4 Chaining Hash Class

The chainingHash class implements a Hash Table that uses seperate chaining as its collision scheme.

2.5 Power Hash Class

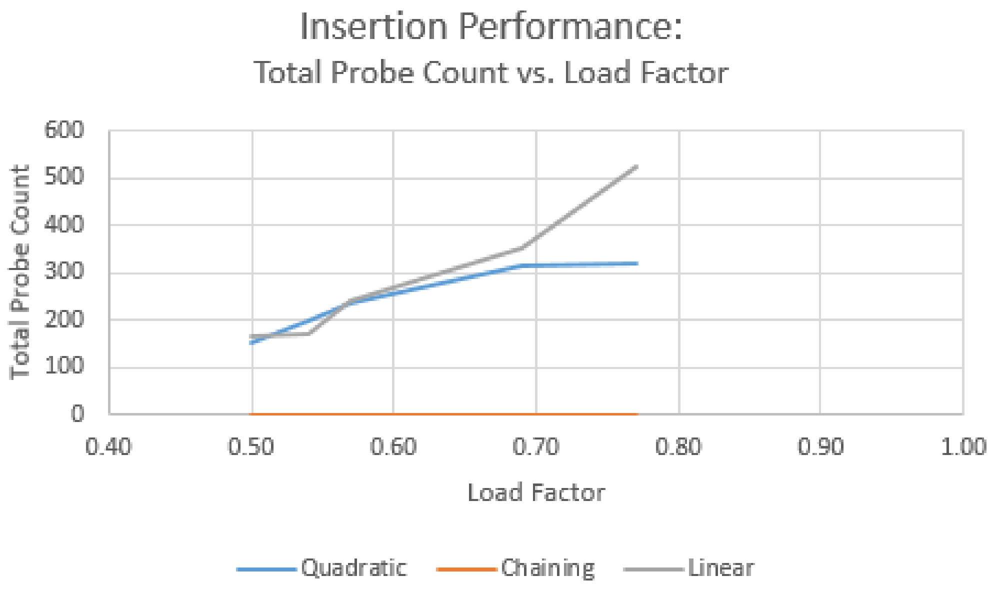
The PowerHash class accepts a table size, collision scheme, data file and a number of keys as its parameters. It generates an object of either linearHash, quadraticHash or a chainingHash type depending on the collision scheme entered by the user. The Hash Table is sized to the user specifications. The program reads in the data file and stores the values in the array. In this experiment I only used cleaned_data.csv file. The program records the number of insert probes. The program searches the hash table for whatever number of random keys the user requested and records the length of the probe sequence for each search.

3 Analysis of Experiment Results

3.1 Insert Performance

Table 1: Insert Performance: Total Probe Count vs. Load Factor

Table Size	Load Factor	Linear	Quadratic	Chaining
656	0.77	525	319	0
727	0.69	354	314	0
883	0.57	241	234	0
929	0.54	170	198	0
1009	0.5	167	153	0

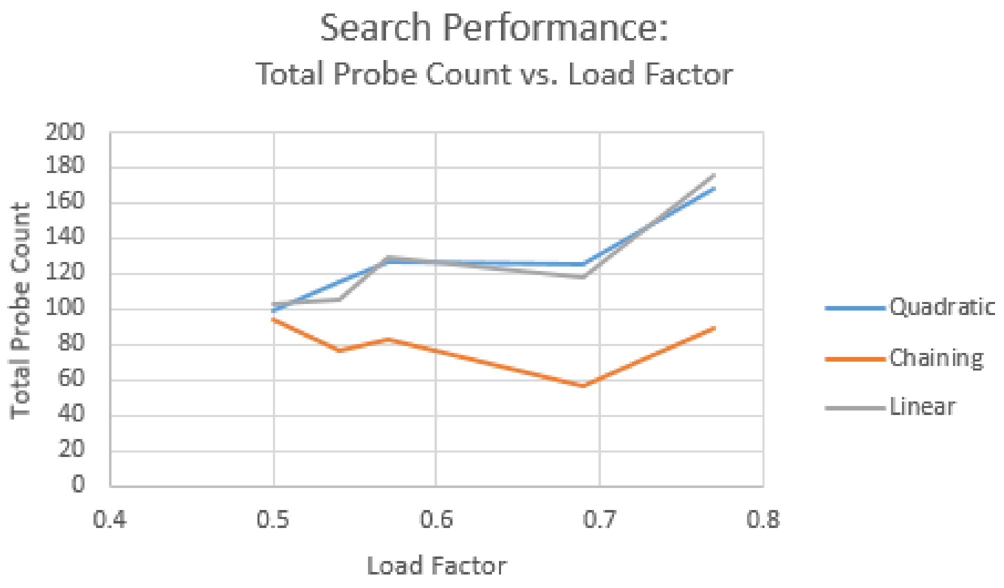


The above graph measures the load factor after insertion is complete against the total number of probes required to insert all 500 data items. The graph shows that the total number of probes for the linear and quadratic methods is directly proportional to the load factor. This means that as the load factor increases there are more collisions, which result in a greater number of probes for the open addressing methods (as more cells are full). For every different table used, the linear collision scheme results in the greatest amount of probes needed to insert data and gets significantly worse than other methods as the load factor increases and as a table fills up. This is due to a side effect of the linear method called primary clustering. Primary clustering refers to the grouping together of adjacent data items which increases the amount of probes needed to get to the next empty value in the table and slows performance. The quadratic method is the better than the linear method but worse than chaining. The total number of probes also increases as the load factor increases, but the gradient is not as high as that of the linear scheme. This is due to secondary clustering which still has the same issue of clusters forming and slowing down the performance, although it is less severe than primary clustering as the data items are widely separated. The separate chaining scheme has the best performance by far. The equation for the line for the chaining scheme is x = 0 which means that it has a gradient of 0 and the total number of probes is 0. This is because there are no collisions involved when adding a data item to a linked list and therefore no need for any probes.

3.2 Search Performance

Table 2: Search Performance: Total Probe Count vs. Load Factor

Table Size	Load Factor	Linear	Quadratic	Chaining
656	0.77	175	168	89
727	0.69	118	125	57
883	0.57	129	127	83
929	0.54	105	115	77
1009	0.5	103	99	94

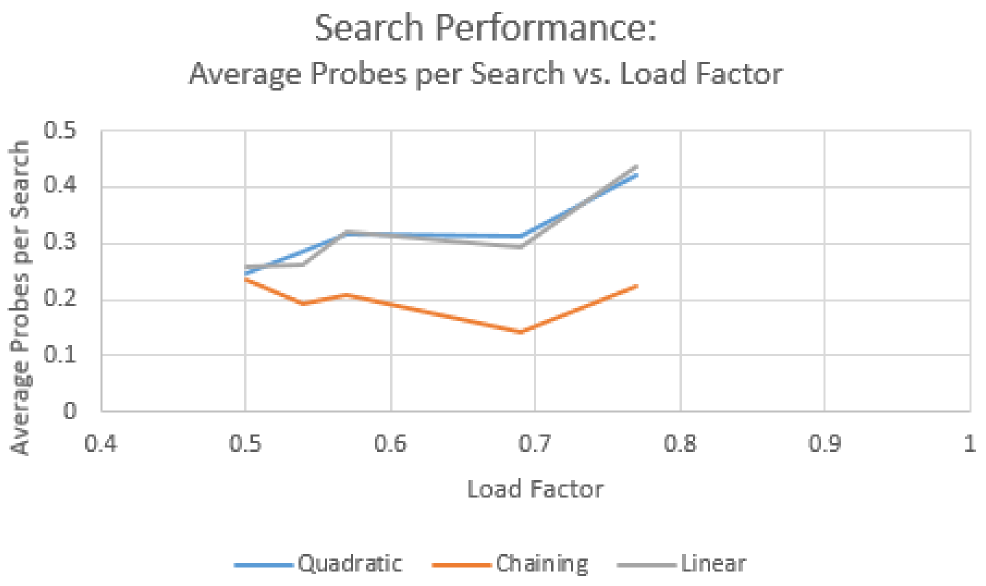


The above graphs compare the load factor with the total number of orobes needed to search for 400 data items, the average number of probes needed to search for a data item and the longest probe sequence that occured while searching for a data item.

The results for the above graph are, to an extent, unexpected. As expected, the separate chaining performed the best out of all the graphs as all its total probe counts remained under 100. However, I expected the graph to be a straight line with a small gradient as the total number of probes to search for a data item should increase as the load factor increases as there would be more collisions in a table that is smaller or fuller. However, it seems that the small difference of having different table sizes in each hash function has caused certain tables to have more collisions than expected and therefore longer probes to search. This explanation can also be used to explain the non-straight behaviour of the quadratic and linear methods. The graph also indicates that the quadratic method resulted in greater number of total probes to search for an item. This must mean that for some reason, in some cases, the secondary clustering turned out to be more severe than the primary clustering.

Table 3: Search Performance: Average Probe Count vs. Load Factor

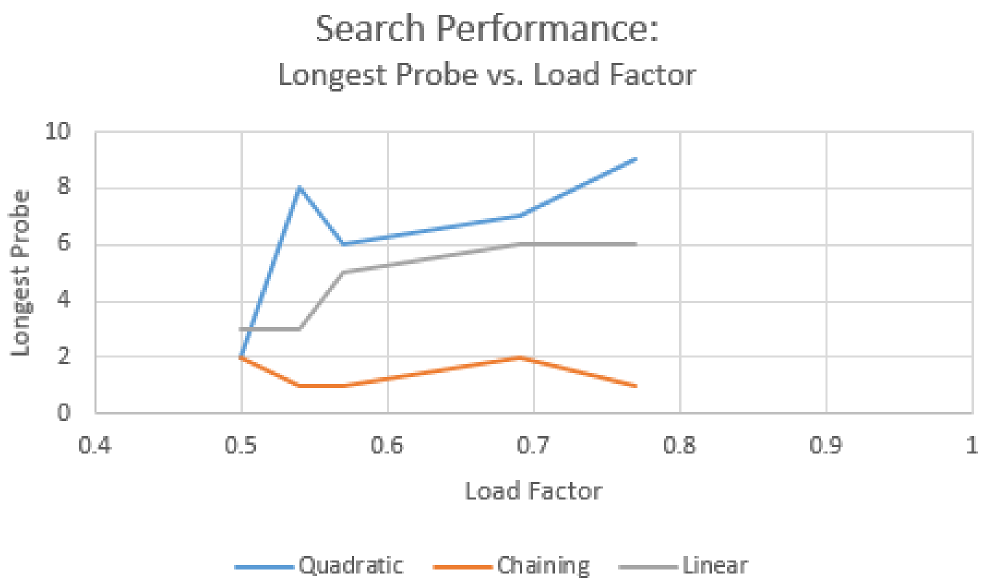
Table Size	Load Factor	Linear	Quadratic	Chaining
656	0.77	0.436	0.419	0.222
727	0.69	0.294	0.312	0.142
883	0.57	0.322	0.317	0.207
929	0.54	0.262	0.287	0.192
1009	0.5	0.257	0.247	0.234



In this graph, the data seems to follow the same pattern of the secondary clustering being more severe than the primary clustering as the line for the quadratic method indicates a higher average than the line for the linear method, although the linear method does have the overall worst average of 0.436 probes to search in a table with a load factor of 0.77. This shows that as the load factor increases, the quadratic method eventually outperforms the linear method.

Table 4: Search Performance: Longest Probe Count vs. Load Factor

Table Size	Load Factor	Linear	Quadratic	Chaining
656	0.77	6	9	1
727	0.69	6	7	2
883	0.57	5	6	1
929	0.54	3	8	1
1009	0.5	3	2	2



The final graph also shows that the quadratic method has the worst performance and that secondary clustering is in some cases worse than primary clustering as the longest quadratic probe is 9, which is more than the longest linear probe of 6.

In conclusion, both open addressing methods result in long probe sequences that hinder performance and, therefore, seperate chaining is a much more efficient method to use to store, retrieve and delete data.

4 Statement of Creativity

1. I implemented a function in all Hash Table classes that displays the Hash Table. This can be found in the output files.
2. I coded this report using TeXstudio and added tables with the data for each graph.

5 Git Log

```
1: commit 271bf907ed1a46c85c732902d8a4e8d7db93b536
2: Author: Leanne January ¡JNRLEA001@myuct.ac.za¿
3: Date: Thu Apr 4 20:41:34 2019 +0200
4:
5: 6. Fixed errors in code and edited Javadoc comments.
6:
7: commit 8580258ed16d3be214077455b405013006ee5d6b
8: Author: Leanne January ¡JNRLEA001@myuct.ac.za¿
9: Date: Tue Apr 2 10:30:01 2019 +0200 10:
...
26: Author: Leanne January ¡JNRLEA001@myuct.ac.za¿
27: Date: Fri Mar 29 16:12:00 2019 +0200
28:
29: 2. Implemented Hash Table that uses quadratic probing to deal with collisions. Edited the Makefile. Also added the PowerUsage class which will be used to store data objects.
30:
31: commit 5071938057b1bc7383e2f996a675657506b1e6ef
32: Author: Leanne January ¡JNRLEA001@myuct.ac.za¿
33: Date: Fri Mar 29 16:01:16 2019 +0200
34:
35: 1. Implemented Hash Table that deals with collisions using linear probing. Made a Makefile for it.
```