Assignment 3: Report

Author: Khulekani Jali
Course: CSC2

Introduction

The purpose of this experiment is to test the efficiency in search and insertion times for hash tables when using chaining, linear or quadratic resolution types for collisions. We test this on a Individual household electric power consumption data set. This data contains a time series of power usage for a suburban dwelling, with “Date/time”, “Global active power” and “Voltage”. Using the data time as the key to retrieve data within the table. This means we will be hashing strings(How this is done is discussed later).

Hypothesis

We expect to see higher search/insertion probe counts for the linear resolution when compared to the quadratic for low table sizes, with chaining having the lowest. For load factors we expect similar load factors for the Linear and Quadratic and lower fro the chaining.

OO Design

To do the experiment we created HashApp, LinkedList, aux and sort class. The HashApp class contains our Hash table and the driver class, the table is either created using a linear, quadratic or chaining resolution scheme. It also contains the hash function. The way it works is, since we have to hash strings, to avoid having hash values that are too big. We take the key e.g. “16/12/2006/17:37:00” and remove the redundant characters and only leave out the characters that make the key unique i.e. day, the hours and minutes, then for the above example we would have “161737”. We hash this and multiply by 10.

HashApp is ran with table size, Resolution, input data file and number of search keys, all are parsed as strings.
Example:

```
java HashApp “653” “Linear” “d.csv” “500”
```

returns(in this order):
Total search probe
Average search probe
Maximum search probe
Total insertion probes

Load factor

The type of scheme is chosen by the user, along with the table size making sure it is prime, if not the program ends and returns an appropriate message to the user. The code for the primality test was taken from geeksforgeeks.org. The user also supplies the number of searches to be done and the input data file. For the chaining resolution, the class inherits a linked list object from the linkedList class. This class has a linkedList super class and the Node subclass. Since the chaining hash table is made up of an array of linkedList objects, the HashApp class inherits the attributes from that class.

The aux helper class is only for printing out the prime numbers between 653 and 1009 as per the requirements for testing. Then a bash script was created to automate the process of the required tests. This script does 10 runs of the HashApp with table sizes ranging from 653 to 1009 per resolution scheme and stores the data into a .csv file “Sampledata.csv”. Since the data in the .csv file contains load factors, insertion and search probes for all resolution schemes we have to sort it. The sort.java file sorts the data into relevant .csv files using the FileWriter class to make it easier to graph the data into gnuplot.

Method

Notation:

searchLinear.csv - contains the total search probes for linear resolution
searchMaxLinear.csv - contains the maximum search probes for linear resolution
searchAvgLinear.csv - contains the average search probes for linear resolution
LFLinear.csv - load factors for the linear resolution scheme

To measure the time efficiency of the resolution types we use operations counts/probes. To do this every time there is a collision in the table, the index is incremented by 1(in the case of linear resolution). The for the resolution type with the highest probe either for insertion or search(which will be the same, see later in the discussion) that resolution type will have low efficiency.

Steps :

To achieve this we create a file with (d.csv in this case) with 500 data points, this is the file we append to the table. Then the table is created with a resolution scheme.

From this point we insert 500 data points into the table using the resolution schemes, while recording the insertion probes(increment 1 to the count) for every time there is

a collision and we need to change the index value. For chaining we used bit shift instead of squaring to optimize the algorithms performance.
Once created we do 10 searches in the table for each resolution type(*Creativity* I did 10 instead of 5 to get a much clearer picture of the trend). The 10 test values must be the same when testing for all the schemes.
Note: The first value = 653 for the quadratic resolution scheme kept returning an error since the number of probes eventually became grater than the table size so I opted for 677. But the rest are the same.

We record the load factor which is just the proportion of the table that is full.
Note: With chaining we would expect a load factor greater than 1 but since we will only be testing with table sizes greater than 500 i.e. table size > 653. The load load factor will be less than 1.

We also record the insertion probe and search probe, note that these will be the same for a given search, since the number of probes done to insert a data point will be the same it takes to find that same data point in the table. We also record the average search probe and the maximum number of probes done for a given number searches K.

We store these results in relevant .csv file(as mentioned in OO Design) then plot them on gnuplot. We plot insertion probes of all the resolution schemes versus the load factors, then the total search probes versus the load factor, the average search probes versus the load factor and the maximum search probes versus load factor. Mind that we do this for all the resolution types on the same graph. We use the “plot” command to do achieve this.

We output some sample statistics to get a mathematical picture of what’s happening in the data. We use the “stats” command in gnuplot to do this.

We analyse the results and come to a conclusion.

Results

Sample Statistics :

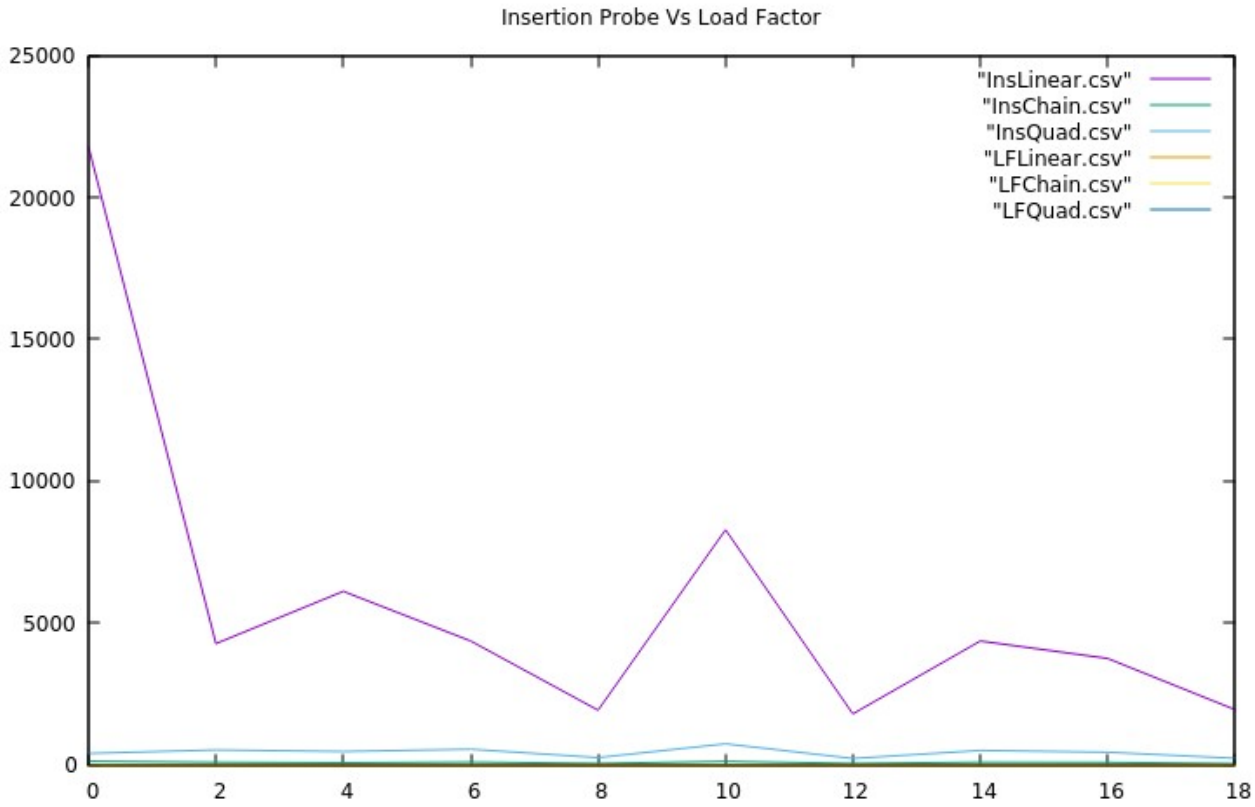
Insertion/Search:

Linear :	Quadratic:	Chaining:
mean = 5852.8	mean = 426.4	mean = 71.1
min = 1790	min = 218	min = 33
max = 21786	max = 732	max = 144

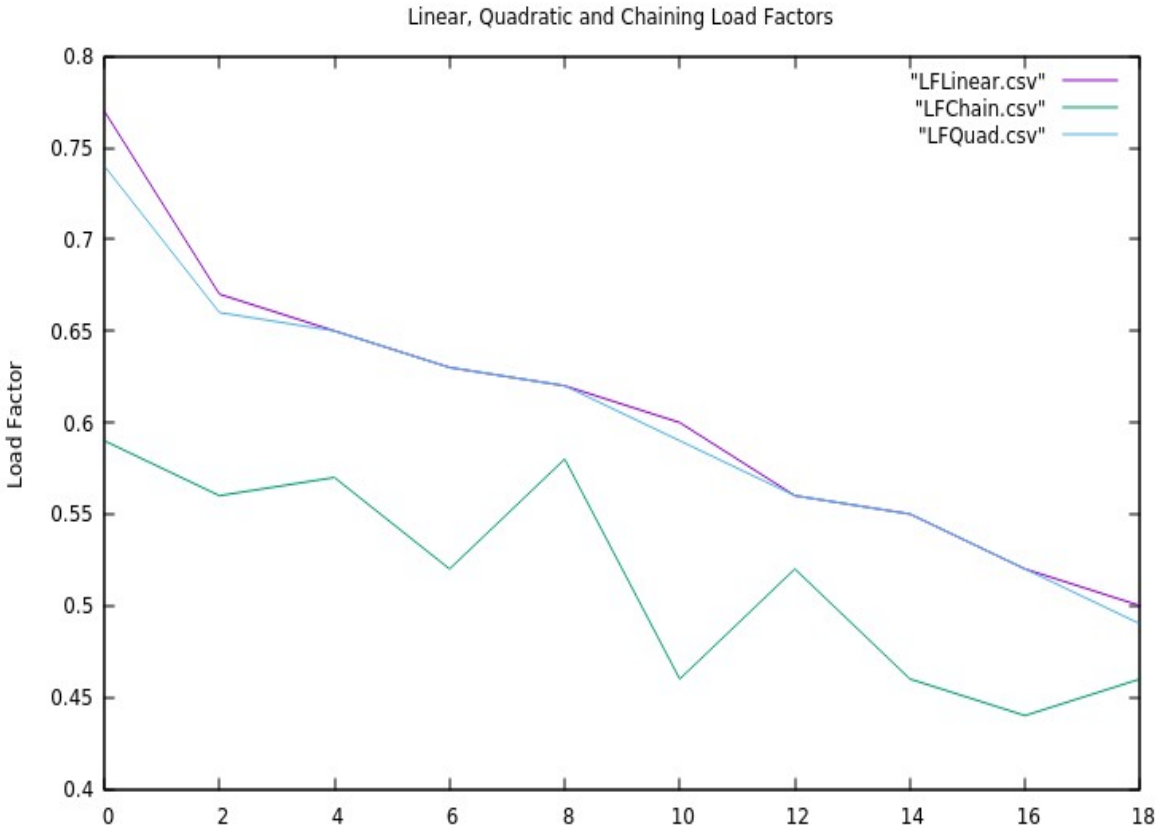
Search max:

Linear :	Quadratic:	Chaining:
min = 88	min = 9	min = 0
max = 342	max = 18	max = 1

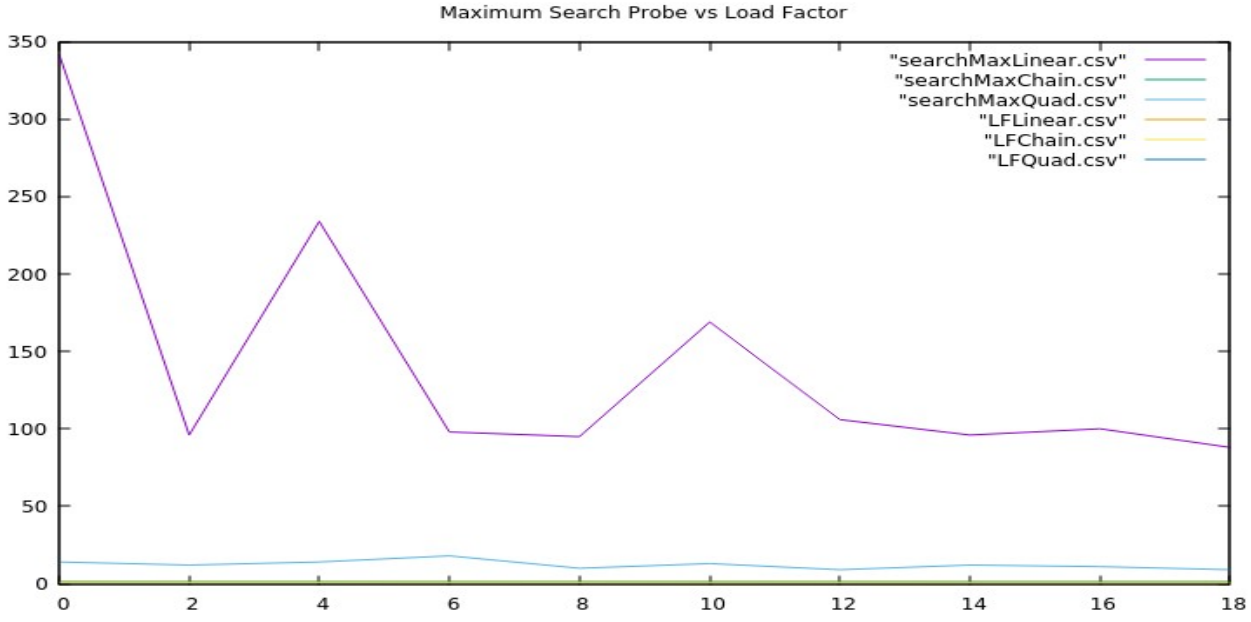
Graph 1: Insertion for Linear, Quadratic and Chaining Vs Load Factor



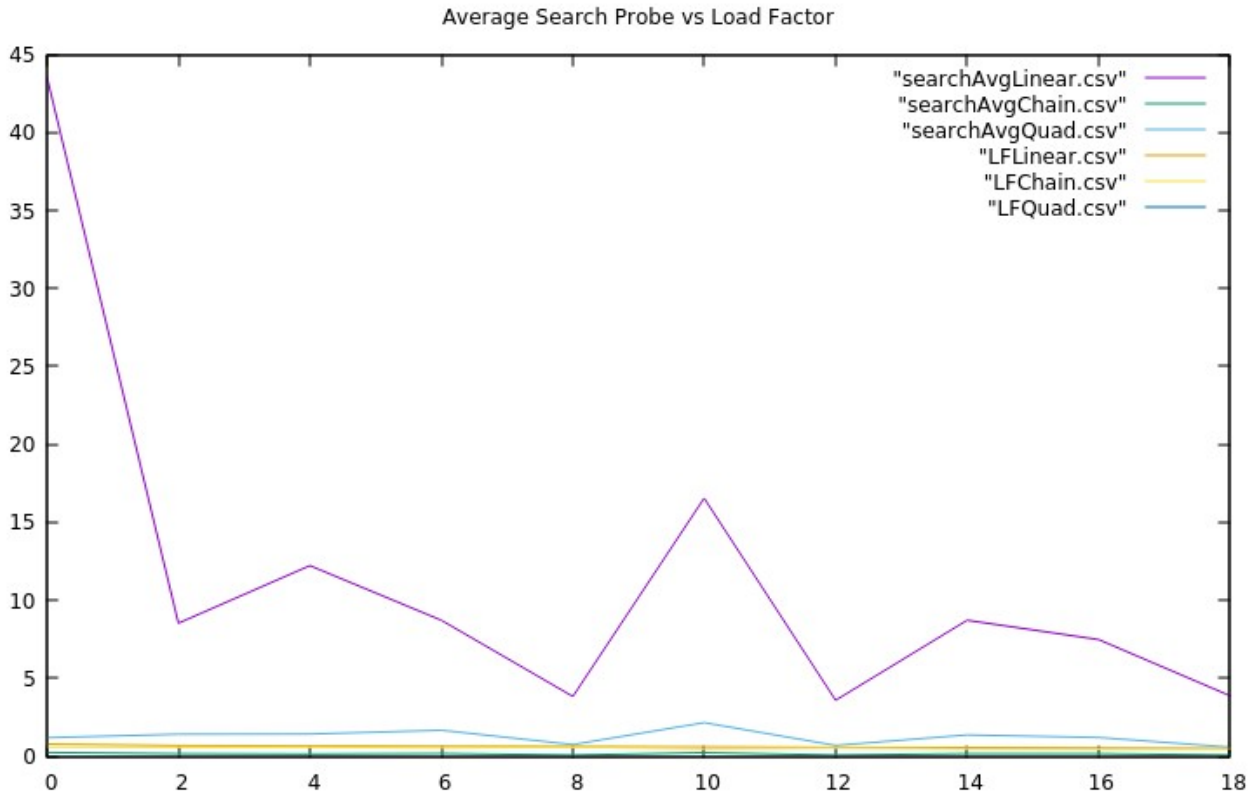
Extra Graph: Linear, Quadratic and Chaining Load Factors



Graph 2: Maximum Search Probe for Linear Chaining and Quadratic Versus Versus Load factor



Graph 3: Average Search Probe for Linear, Quadratic and Chaining Versus Load Factor



Discussion

Since the search and insertion probes are the same for all resolution types we will not print out the search against load factor graph, but it can be found in the submitted file.

From graph 1: We see that the all the load factors are between 0 and 1, this is to be expected since the the load factor is a calculation of the sum of the number of elements in the table divided by the table size. We observe this for all the graphs. The insertion probes are quite high for the linear probe. This is caused by Primary Clustering, once there is a collision the hash value is only incremented by 1 which gets worse as there are more insertions. We also note that the probes get smaller as the table size increases, this is because data points are getting more and more spread out within in the table and search becomes easier. We also note that the insertion probes for quadratic resolution are slightly higher than those from chaining, this is caused by the use of linked lists, this means that every time there is a collision we only increment by 1 and add the new value to the beginning of the linkedlist. This results in lower insertion counts.

From graph 2 : From the sample stats we found that the maximum are 342, 18 and 1 for linear, quadratic and chaining respectively. Which is quite clear from the graph. Obviously linear resolution has the worst search probes with chaining having the best, meaning at worst chaining is only doing 1 probe to find matching keys. If the test table sizes were smaller the maximum would increase.

From graph 3: The third graph is just the first graph squished in, since we are only taking averages of the searches, the results are what we would expect.

(*Creativity)*
From the extra graph which is all the load factors we can see a distinct pattern where as with the other graphs for large table size the load factor decreases, this is simply because for larger table size the amount of data within the table stays the same and the load factor decreases. One more thing to note is that the load factor for chaining is consistently lower than that for the linear and quadratic. This is because values are inserted into linked list these list(as mentioned above) are at most of size 2, meaning the number of elements(Linked lists) is much smaller for chaining. The load factors for linear and quadratic are similar because the number of elements in the tables still has to be 500.

Sample Statistics:

From the sample statistics we see similar results as from the graphs

Conclusion

From the results it is easy to tell that the chaining resolution has much better performance for searching and insertion since it doesn’t use mod(which is computationally expensive) when probing but uses linked lists it is much better, followed by the quadratic probing which has acceptable results but when compared with the linear which has the worst results of all, but easier to code. These results are exactly what we hypothesised.

Git log can be found in log.txt.