CSC2001F

Practical Assignment 3

Data Structure Comparison

Hash Tables

Niceta Nduku

NDKNIC001

5 April 2019

Introduction

To see instructions on how to run the program, please read the readme.txt file that is in the same directory as this document.

Object Oriented Programming Design Process

PowerData

This is an immutable class that creates the object that is going to be stored in the data structures. The instant variables of PowerData are Date/Time, Power and Voltage which are extracted and set when the CSV file is read. The class has methods, getDateTime and overridden toString from the object class. getDateTime returns the Date/Time of the object.

To create a new PowerData object,

```
PowerData dataItem = new PowerData (dateTime, power, voltage).
```

This class is the same as the one that was used in Assignment 1.

HashTable

HashTable is a class that creates a Hash Table. The constructor takes in an integer of the table size, and the collision resolution to be used eg

```
HashTable= new HashTable(701, "l")
```

There is an inner class called Entry that is the object to be stored in the hash tables. An entry object keeps a PowerData object. It had methods setNext(Entry next) and getNext() which are to be used when chaining.

The resolution schemes are "l" for linear, "q" for quadratic and "c" for chaining. There are instance variables for each collision resolution that are initially set to false and only set to true depending on the resolution specified by the user in the constructor parameters.

A table of the specified size is created, and everything is set to null.

`LinearProbe(int hashIndex)` is the method that will return an index using linear probing. `QuadraticProbe(int hashIndex)` does the same thing, however, it will search for the next index quadratically. The both take in an integer- that is the hashed value of the key- and the key that is either being used to search or insert. `Hash(String key)` is the hash function that gives the hash value of the key.

`Insert(PowerData d)` inserts the data item into the hash tables. It will first create a new entry class using the data item. It will then check to see which collision resolution is set to true and call on either `LinerProbe, quadraticProbe` or add the items using chaining.

Search(String key) searches for the data item that has a matching key. Just as above in insert, the collision resolution set to true will be checked and the key will be used to find the data item.

As each item is being inserted a count of each item is being incremented. getLFactor()returns the number of items added divided by the table size.

For every probe iteration, e.g. in the linearProbe loop, a count of the number of times a probe was done i.e. the total number of times the loop went over. `getInsertProbe()` and `getSearchProbe()` return the number of probes for insert and search respectively.

PowerHashApp

When PowerHashApp is run in the terminal eg `java PowerHashApp 700 c cleaned_data.csv 40`, the main method will check what the first position in the string argument is. If it is "test" it will run the experiment that will be described below. Otherwise, the first position in the string arguments is the tablesize, the next is the hashing scheme, the third is the file that contains the data and finally the number of keys to be searched for, k. The table size will be checked to see if it is a prime number using isPrime(n), and if false, nextPrime(n) will be used to get the next prime number. These two methods were from the text book Data Structures & Problem Solving Using Java [1]. getData(int size, String probe, String file) is then called and the file is read line by line, creating and adding PowerData items to a HashTable powerHash which is a global variable. As items are being added, a list of dateTime keys is being created.

output(k) will then display the load factor by calling on getInsertProbe. It also loops through k times searching in the powerHash table for a data item that matches the kth key in the key list. For each iteration, the number of search probes will be added into a table. The total number of probes, longest probe and average number of probes will then be displayed to the user.

Experiment

The aim of the experiment was to show which hashing scheme is the most efficient. This was done by comparing the number of probes during insertion and searching against the load factor. In linear and quadratic probing, the probe count was incriminated each time the mod operator was used in the functions LinearProbe and QuadraticProbe. For chaining, the probe count was incremented when traversing a list.

In the PowerHashApp, I created an array of 5 prime numbers from 653, choosing every tenth prime number as the table sizes. For each of the table sizes, generated hash tables using each of the collision resolutions and 400 search keys. The output was then used to generate the graphs below.

Results and Analysis

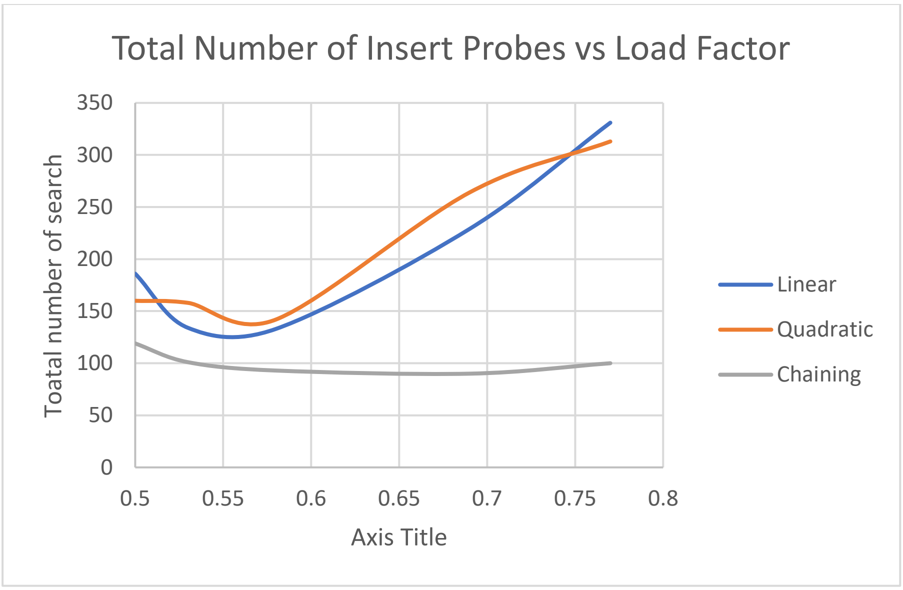

Figure 1: Total number of insert probes against load factor.

For linear and quadratic hashing schemes, it can be seen that as the load factor increases the number of probes increase. Chaining has the least because it only needs to move down a list to add. Assuming that the hash function gave multiple unique keys, then there wouldn't be lengthy lists and therefore require very few probes.

For low load factors, quadratic and linear probing nearly behave the same. Due to the large table sizes compared to input data.

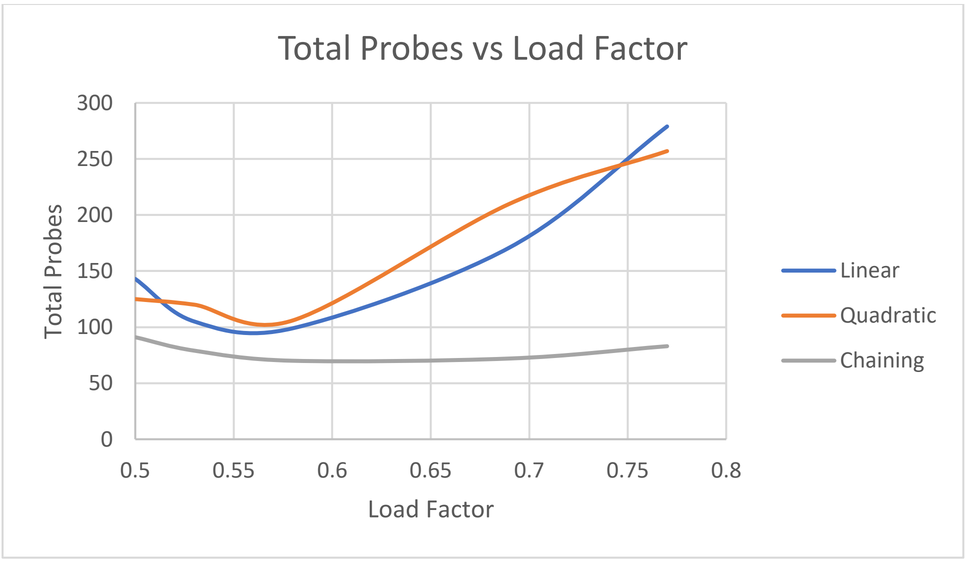

Figure 2: Total number of search probes.

The total number of probes in searching will nearly be the same as the insert probe graph. This is because they keys being searched for are the same being inserted- minus the 100 not in the keys. Adding all the search probes will give the same results as adding all the insert probes.

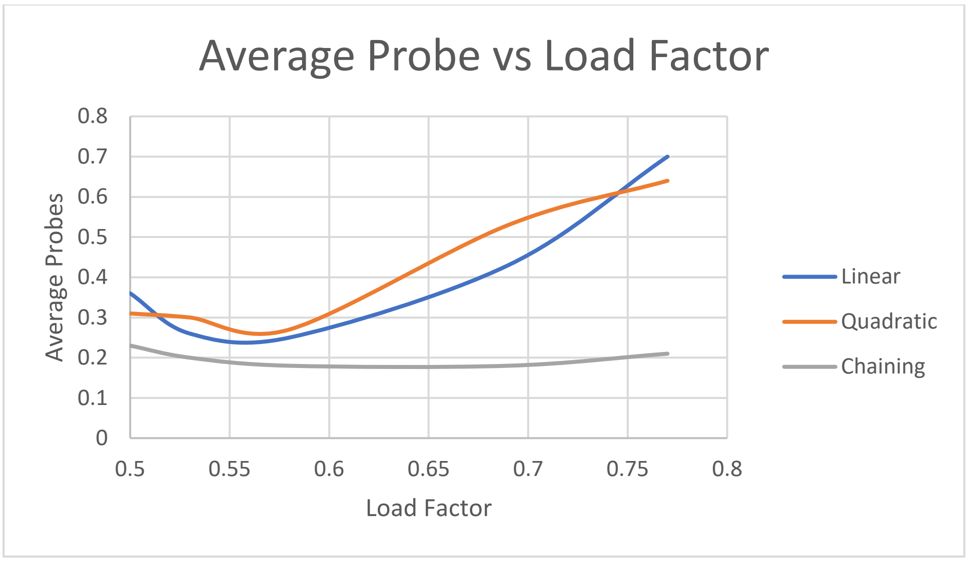

Figure 3: Average number of search probes against load factor.

It can be assumed from the graph above that the hash function almost always gives unique indexes because the average probe count does not exceed 1. This means that for majority of the keys, there was no collision and were therefore found in the index produced by the hash function. Obviously, as the load factor increased, the average number increased because the probability of a collision is higher.

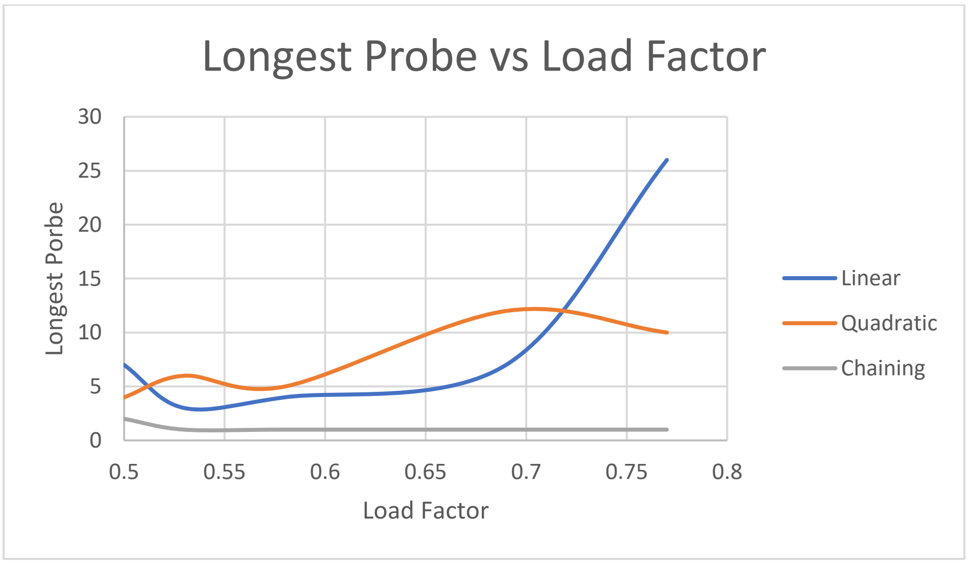

Figure 4: Longest search probe against load factor.

Similar to all the trends above, as the load factor increased so did the probes. Linear probing shows the worst change in the longest probe after a load factor of 0.7, which is expected because primary clustering increased with increase in load factor. Therefore, the probability of having very long probes increases.

Creativity

I created 3 classes, two object classes and one main application. This was to make the program run more efficiently. Hash table simply created a hash table and PowerData was the object being stored. PowerHash run everything from the user's commands.

I also included my own node class in HashTable, Entry, to use for chaining instead of java's linked list by simply adding a setNext and getNext method to the object being added.

Instead of using a bash script, I included my testing into the main program. In the terminal, running `java PowerHashApp test cleaned_data.csv` will automatically run the experiment for the five table sizes and produce the output as required.

Appendix

Git log

```
commit 05c23342e43494cda7b7735b91485fc52e777eb1
Author: NICETA NDUKU <NDKNIC001@myuct.ac.za>
Date: Thu Apr 4 23:06:52 2019 +0200

Complete working practical

commit 857042df79f7ae1f804c8886415bce2db301edc4
Author: NICETA NDUKU <NDKNIC001@myuct.ac.za>
Date: Mon Apr 1 16:53:53 2019 +0200

...
Author: NICETA NDUKU <NDKNIC001@myuct.ac.za>
Date: Sat Mar 30 15:09:27 2019 +0200

Modication on table insert

commit 31f86f59c6eddbb5465e89abc62840aba68d2df7
Author: NICETA NDUKU <NDKNIC001@myuct.ac.za>
Date: Sat Mar 30 11:16:18 2019 +0200

Practical 3 Code
```