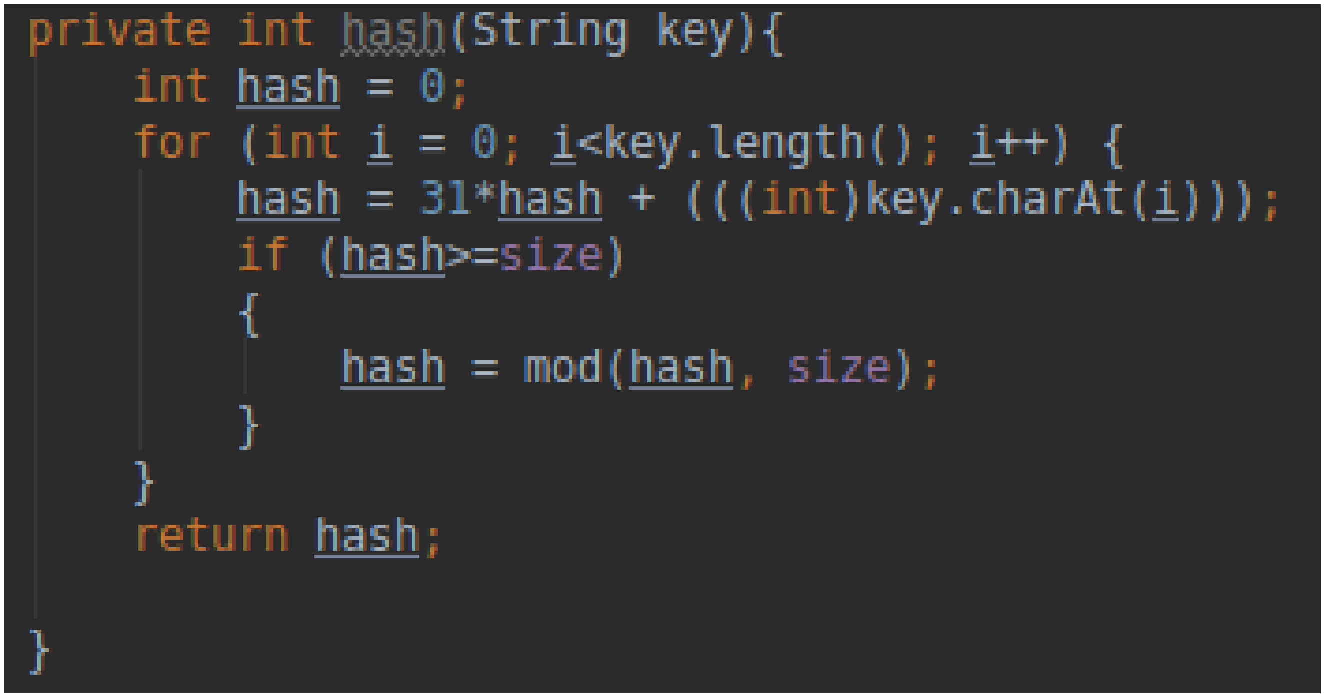


```
private int hash(String key){
    int hash = 0;
    for (int i = 0; i<key.length(); i++) {
        hash = 31*hash + (((int)key.charAt(i)));
        if (hash>=size)
        {
            hash = mod(hash, size);
        }
    }
    return hash;

}
```

A Comparison of Collision Resolution Schemes

MSTRAH001

Rahul Mistri | CSC2001F | Assignment 3

The Experiment

Hash tables is a data structure that provides constant time for searching, inserting and deleting elements in the table. This is achieved with the use of a hash function and a collision resolution scheme in the case that the hash function maps two keys to one value. In this experiment, three hash collision resolution schemes, namely linear probing, quadratic probing and chaining, were compared using their respective insert and search costs (in terms of the number of probes required). Note that the initial hash index computation was not included in the probe count. The three schemes were tested on five different table sizes, and with each table size, the number of probes required for inserting 500 keys, searching the same 400 keys as well as the maximum and average number of probes after 400 searches were recorded. These results were then plotted and carefully analyzed.

Class Design

In this experiment, 6 classes were created. This section will describe the primary fields and methods. For a description of all fields and methods, please consult the Javadocs.

Class: HashTable.java – this class is an abstract class used to declare the common fields and methods for the three classes.

Fields

- size – the table size (int)
- loadfactor – proportion of the table filled (double)
- table – array of Power objects used for linear and quadratic probing(Power[])
- tableC – LinkedList<Power> array used for chaining (LinkedList<Power>[])
- file – the file name to read data from (String)
- numkeys – the number of keys to be searched (int)
- IPC – Counter to keep track of total probes used to insert (int)
- SPC – Counter to keep track of total probes used to search (int)
- maxSPC – Counter to keep track of the max probes used to insert (int)
- chaining – boolean used to store if scheme is chaining or not (boolean)

Methods

- *HashTable(size: int, file: String, numkeys: int, chaining: boolean)* – declares a HashTable given the above parameters and instantiates it by calling populate(). Throws an exception if size is not prime.
- Insert(Power p) – abstract method defined in subclasses
- Search(String [] keys) – calls the search method for each key in the array
- Search(String key) – abstract method defined in the subclasses
- Hash(String key) – returns an integer value in [0, size-1] corresponding to the key

- Mod(int k, int i) – returns the equivalent of k%i using subtraction
- isPrime(int num) – checks if num is a prime number
- printDataToFile(String filename, String val) – prints val to a file filename
- populate() – calls the insert method for each power object created from the file given in the constructor.

Class: LinearHash.java – this class is a subclass of HashTable and defines the insertion and search method used for a linear probing scheme

Fields
- numinsertions – the number of insertions(int)

Methods

- *LinearHash(size: int, file: String, numkeys: int)* – declares a LinearHash object using the above parameters.
- Insert(Power p) – inserts the given power object into the hash table
- Search(String key) – searches for a power object in the hash table given a key

Class: QuadraticHash.java – this class is a subclass of HashTable and defines the insertion and search method used for a quadratic probing scheme

Fields
- numinsertions – the number of insertions(int)

Methods

- *QuadraticHash(size: int, file: String, numkeys: int)* – declares a QuadraticHash object using the above parameters.
- Insert(Power p) – inserts the given power object into the hash table
- Search(String key) – searches for a power object in the hash table given a key
- Update(int val, int i) – increases val by (2i-1) using bitshifting

Class: ChainingHash.java – this class is a subclass of HashTable and defines the insertion and search method used for a chaining scheme

Fields
- numinsertions – the number of insertions (int)

Methods

- *ChainingHash(size: int, file: String, numkeys: int)* – declares a ChainingHash object using the above parameters.
- Insert(Power p) – inserts the given power object into the hash table
- Search(String key) – searches for a power object in the hash table given a key

Class: Power.java – this class is defines the objects that are stored in the hash table

Fields
- date (String)

- power (String)
- voltage (String)

Methods

- *Power(date: String, power: String, voltage: String)* – declares a Power object using the above parameters.
- toString() – returns a summary of the fields
- compareTo(Power p) – returns 0 if both recordings were on the same day, -1 if the calling object was recorded prior to p or 1 if the calling object was recorded after p
- equals(Object o) – returns whether the dates are equal since each power object is uniquely identified by the date
- copy() – returns a Power object with the same fields as the calling power object

Class: MainTesting.java – this class is used to create and test hash tables from the command line interface

Fields

- arr – used to store the search keys (static String[])

Methods

- main(String[] args) – directs how the testing is done from the command line interface and performs a search of the keys stored in arr on a HashTable
- testLinearInsert() – specifically tests the linear insertion
- testQuadraticInsert() – specifically tests the quadratic insertion
- testChainingInsert() – specifically tests the chaining insertion
- getKeys() – populates arr with keys to search for

Results and Discussion of Results

Figure 1

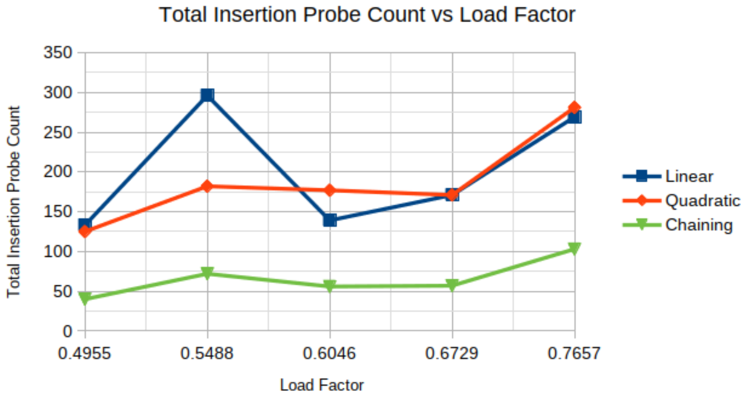


Figure 2

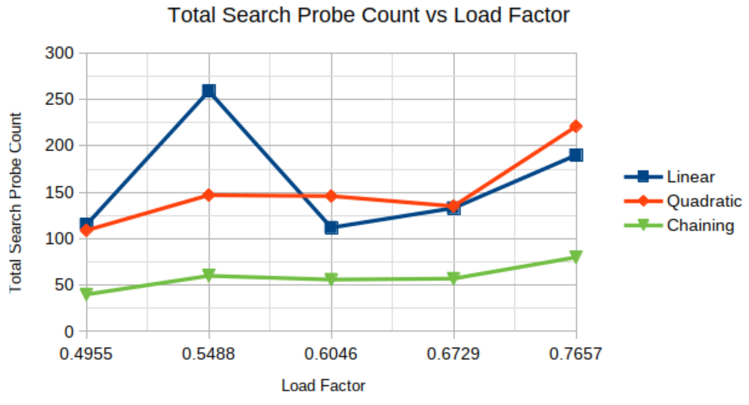


Figure 3

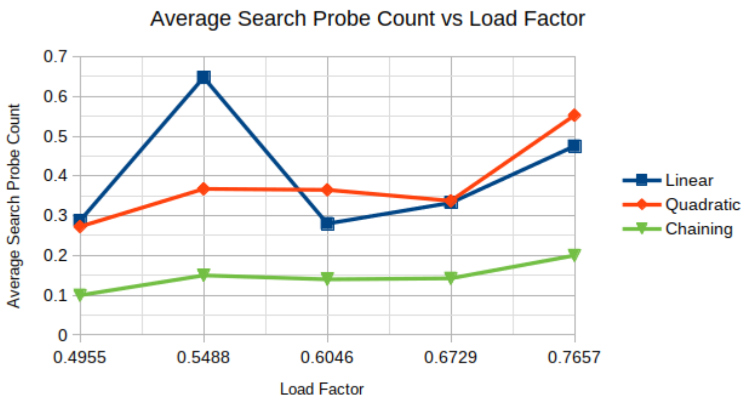
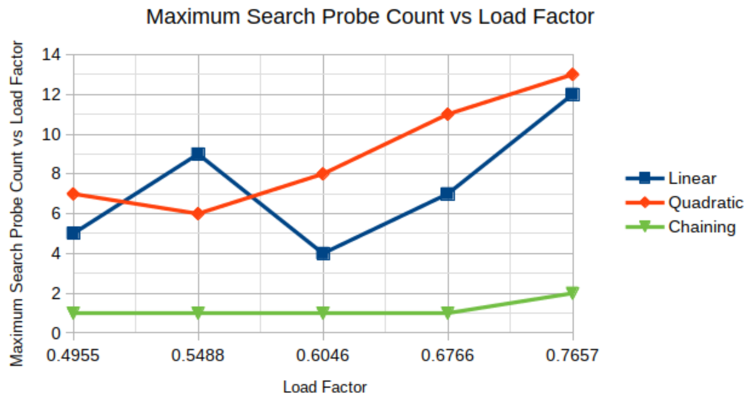


Figure 4



Note: the load factor is inversely proportionate to table size

In the above figures, the load factor for each table was calculated as the number of elements inserted divided by the table size. Because 500 elements were inserted into each of the five sets' three tables, with each set increasing in table size, the load factor is the same for each of the schemes in each of the five sets.

In figure 1 we see across all three graphs that as the load factor increases, a general increase in the total insertion probe count is observed. This implies that the smaller the table size, the higher the required number of probes are needed for insertion. Note the spike in all three graphs when the load factor is approximately 0.5488. This is due to a higher amount of collisions for that specific hash function at that specific table size. The linear probing scheme experiences a high probe count because it checks the next index value linearly, ie if there is a collision at k, it will sequentially check k+1,k+2,…,k+n until a free slot is found. On the other hand, the quadratic probing scheme checks $k+1^2, k+2^2, \ldots, k+n^2$ until a free slot is found. Because the linear scheme checks and inserts adjacent slots, it has a high density of records around some value of k, in other words it experiences clustering at a high level. Since quadratic checks every $(k+n^2)^{th}$ slot, clustering is reduced. This explains why the spike in the linear scheme is so much higher. In chaining, the load factor is also the expected length of each list at any given index. This is why the chaining scheme's line graph stays consistently low since the expected length is below 1, and so the hash function rarely maps two keys to one index, and if it does, instead of checking the following indices, it just adds it to the end of the list. Hence chaining is the consistently least expensive for insertion in terms of total probe count. Note we are assuming that the hash function used is a decent one in that it spreads values evenly. If, as an extreme case it mapped all keys to one index, under the linear and quadratic schemes, insertion would become O(n) and the chaining scheme would degenerate into just a LinkedList and follow the same big Oh notation.

In figures 2 and 3, we also notice that as the load factor increases, there is a general increase in the total and average search probe count for all three schemes. In figures 2, 3 and 4, the spike seen in the linear scheme's line graph can be related due to clustering. As seen above, chaining remains to be the least costly since it's expected length of any given list at an index is less than 1. Notice that as the load factors increase, the total, average and maximum search probe counts for linear and quadratic probing schemes come closer together. This is because of the increase in clustering. In figure 4 it is interesting to note that for the chaining scheme, the maximum probe count is mainly 1, meaning that the hash function maps at most two elements to an index. Since the average search probe cost (in figure 3) for chaining is well below one, we can see that this rarely occurs. In figure 4, as the load factor increases (and hence the smaller the table size), the greater the number of collisions occur for the linear probing and especially the quadratic probing scheme.

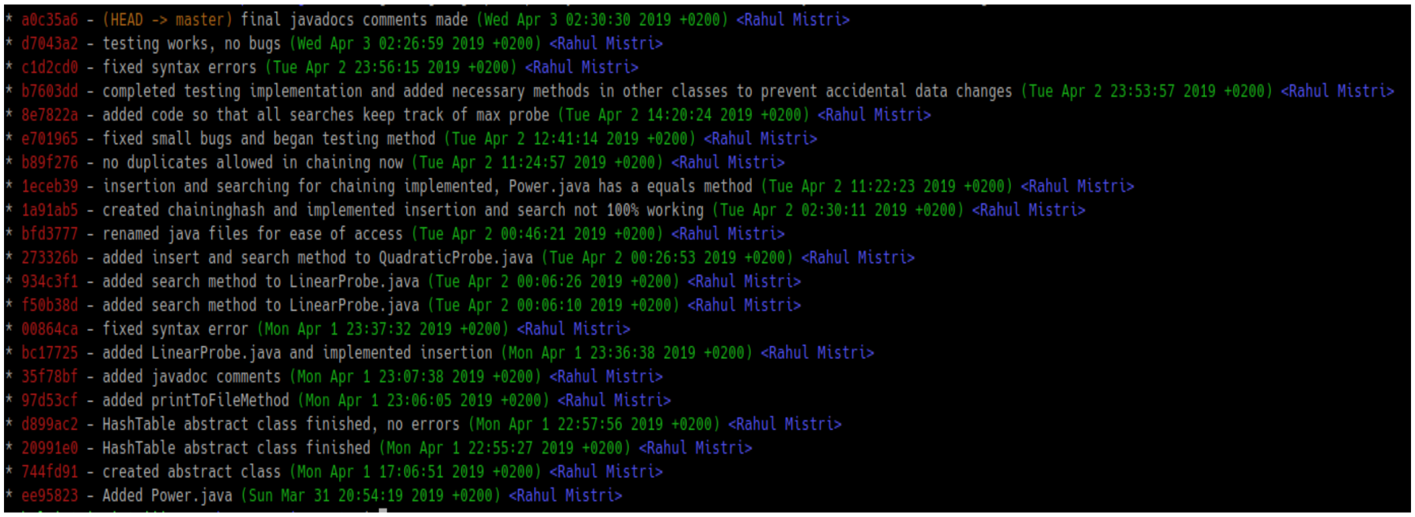
These results align with the theory behind the different collision resolution schemes. Linear probing should have a table with a load factor of at most between 0.5 and 0.7. Quadratic probing should have a table with a load factor of at most 0.5, as we can see a sharp increase in all figures in the probe cost, especially in figure 4 due to the high number of collisions. Chaining should have a load factor of at most 1 and the results in the figures above confirm this, as all costs are fairly low in comparison to linear and quadratic probing schemes when the load factor is below 1.

Creativity

In this assignment, I made use of numerous techniques which I believe constitutes as creativity. Firstly, I made an abstract class called HashTable.java which enabled me to code the general properties that apply across all collision resolution schemes. This improved the readability of the code across the three schemes as well as the abstract class. Unnecessary repetition of code was avoided as well. This also helped with regards to debugging of code. If the same error occurred across all three schemes, it pointed out that the bug most likely was in the abstract class, whereas if it occurred in just one scheme, it could be narrowed down to being in either the insert or search method, which is easy to check in which method the bug is in. The abstract class can also be used to implement other collision resolution schemes and hence the code is very reusable.

Secondly, multiple methods were made that replaced computationally expensive operations. When writing the update method in the quadratic probing scheme (which updates the index if a collision occurs), instead of using the computationally expensive squaring of a number, the number was bitshifted and then 1 was subtracted since the difference between $(i)^2$ and $(i+1)^2$ is 2i-1. The computationally expensive % operator was replaced with a method called mod which performed the same operation but using subtraction.

Git Log



```
* a0c35a6 - (HEAD -> master) final javadocs comments made (Wed Apr 3 02:30:30 2019 +0200) <Rahul Mistri>
* d7043a2 - testing works, no bugs (Wed Apr 3 02:26:59 2019 +0200) <Rahul Mistri>
* c1d2cd0 - fixed syntax errors (Tue Apr 2 23:56:15 2019 +0200) <Rahul Mistri>
* b7603dd - completed testing implementation and added necessary methods in other classes to prevent accidental data changes (Tue Apr 2 23:53:57 2019 +0200) <Rahul Mistri>
* 8e7822a - added code so that all searches keep track of max probe (Tue Apr 2 14:20:24 2019 +0200) <Rahul Mistri>
* e701965 - fixed small bugs and began testing method (Tue Apr 2 12:41:14 2019 +0200) <Rahul Mistri>
* b89f276 - no duplicates allowed in chaining now (Tue Apr 2 11:24:57 2019 +0200) <Rahul Mistri>
* 1eceb39 - insertion and searching for chaining implemented, Power.java has a equals method (Tue Apr 2 11:22:23 2019 +0200) <Rahul Mistri>
* 1a91ab5 - created chaininghash and implemented insertion and search not 100% working (Tue Apr 2 02:30:11 2019 +0200) <Rahul Mistri>
* bfd3777 - renamed java files for ease of access (Tue Apr 2 00:46:21 2019 +0200) <Rahul Mistri>
* 273326b - added insert and search method to QuadraticProbe.java (Tue Apr 2 00:26:53 2019 +0200) <Rahul Mistri>
* 934c3f1 - added search method to LinearProbe.java (Tue Apr 2 00:06:26 2019 +0200) <Rahul Mistri>
* f50b38d - added search method to LinearProbe.java (Tue Apr 2 00:06:10 2019 +0200) <Rahul Mistri>
* 00864ca - fixed syntax error (Mon Apr 1 23:37:32 2019 +0200) <Rahul Mistri>
* bc17725 - added LinearProbe.java and implemented insertion (Mon Apr 1 23:36:38 2019 +0200) <Rahul Mistri>
* 35f78bf - added javadoc comments (Mon Apr 1 23:07:38 2019 +0200) <Rahul Mistri>
* 97d53cf - added printToFileMethod (Mon Apr 1 23:06:05 2019 +0200) <Rahul Mistri>
* d899ac2 - HashTable abstract class finished, no errors (Mon Apr 1 22:57:56 2019 +0200) <Rahul Mistri>
* 20991e0 - HashTable abstract class finished (Mon Apr 1 22:55:27 2019 +0200) <Rahul Mistri>
* 744fd91 - created abstract class (Mon Apr 1 17:06:51 2019 +0200) <Rahul Mistri>
* ee95823 - Added Power.java (Sun Mar 31 20:54:19 2019 +0200) <Rahul Mistri>
```