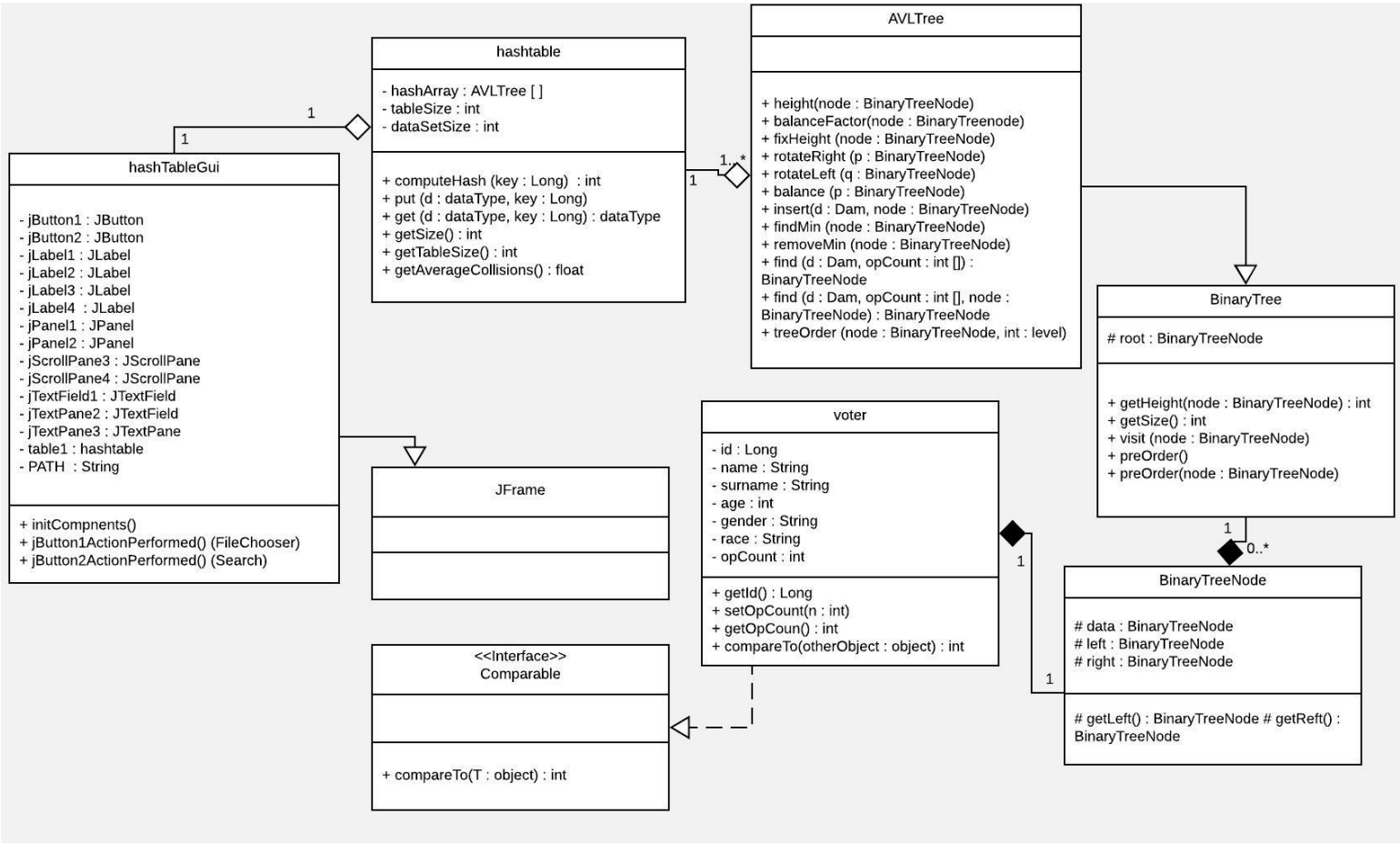
CSS2001F
Assignment 3: HashTables
DPLJER001
Report: Experiment Design and Results

a. Report Outline & Objectives

This report details my work on assignment 3. The primary objective of the assignment was to design a user-friendly, super efficient search application for voter information in memory. The first two parts of the report will detail my OO design, the type of hash table I chose to use and the functionality and layout of my GUI. The subsequent section will briefly outline the process and the asses the results of the complexity tests which compare the collision resolution mechanism of linear probing to AVL trees - the one used in the design of the application.

b. Object Orientated Design (UML)

The following is a UML diagram for the OO design of the Voter Search Application:



For security's sake, a separate set of classes were made to run complexity tests:

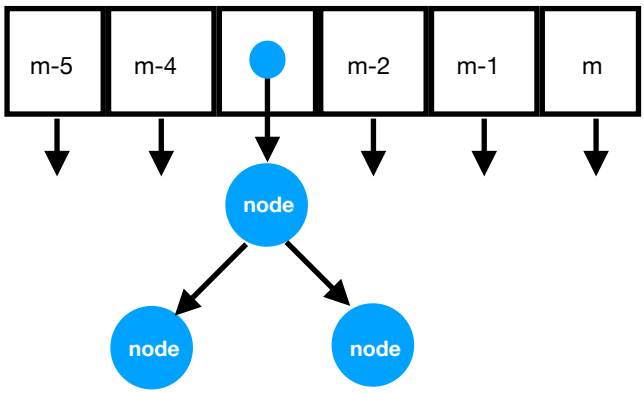
- A *hashTest* class to run the complexity tests and generate the data in .txt files
- A *hashTableLinearTest* class which implements a linear probing collision resolution mechanism
- A *hashTableAVLT*est class which is exactly the same as the *hashtable* class in the UML diagram above - it implements an AVL tree mechanism for dealing with collisions
- A *graph* class to plot the test results

These additional classes may be found in the src/ directory.

c. Voter Application & Hash Table Design

My chosen hash table design for the Voter Search Application was to implement a simple hashing algorithm, and always maintain a table size of 1.5 x the size of the data set (the number of voters in this case) ensuring a load factor of exactly 0.75 for each new dataset loaded into the app. The hashing algorithm I chose to use was to convert the 13 digit id number into a String, and use the String.hashCode() method to get an integer back, then I performed the bitwise AND operator (& 0xffffffff) in order introduce some randomness and hopefully help with the distribution, it also ensures the integer output is positive. I then took the modulus with the table size, and used the final output as an array index. I chose to handle collisions with AVL trees in each position of the array - the motivation behind this was that the balancing property would minimise comparisons during search.

Here is a sample of what the structure looks like (each array index points to an AVL tree "bucket"):

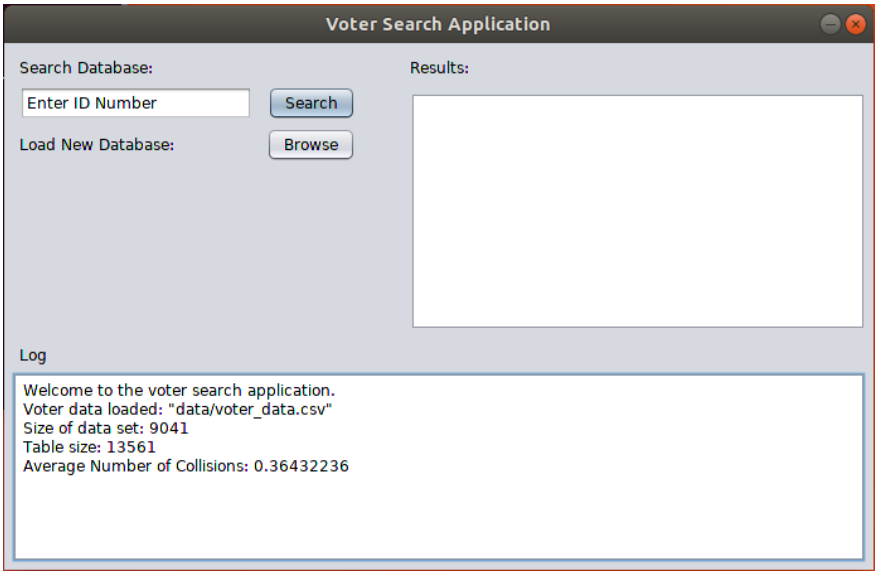


The combination of the table size, hash algorithm and the AVL tree collision handling results in an average collision rate of approximately 0.35 - in this case I defined the number of collision per index in the hashArray as the size of the number of nodes in the AVL tree over 1 (e.g. 2 nodes would be one collision, 3 nodes would be 2 collisions etc.).

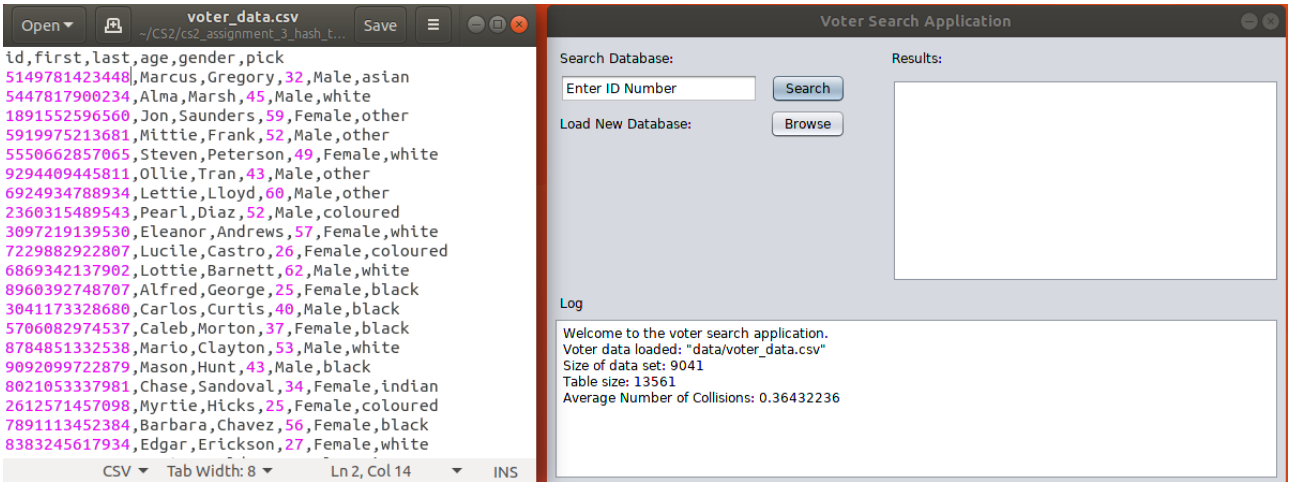
The design of the graphical user interface has the following core properties:

- Search functionality (with error handling for incorrect query)
- Comprehensive, clear display of results
- Functionality for loading new (.csv) data files into the app (with error handling for incorrect format and file type)
- A a scroll pane to log information about the data set (file path, size, average collisions) and about previous searches. It also reports on errors when attempting to load datasets with the incorrect format.

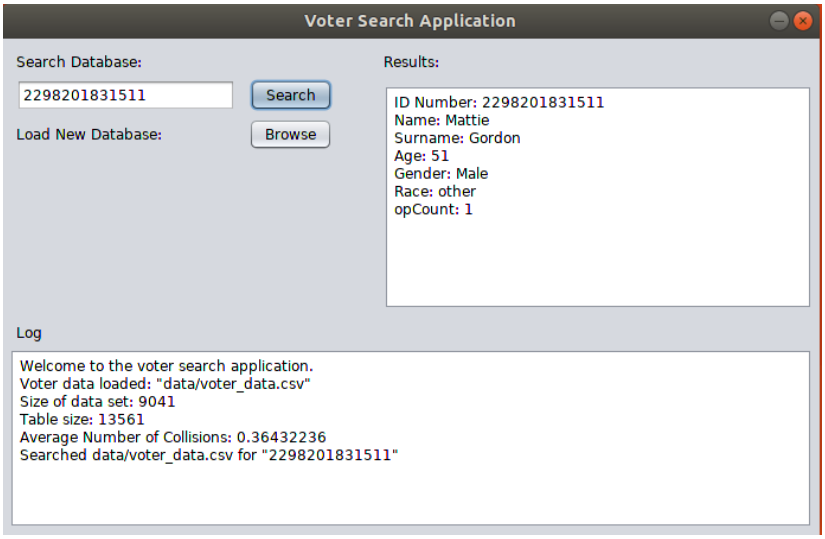
Below are screen shots of the core properties mention above:



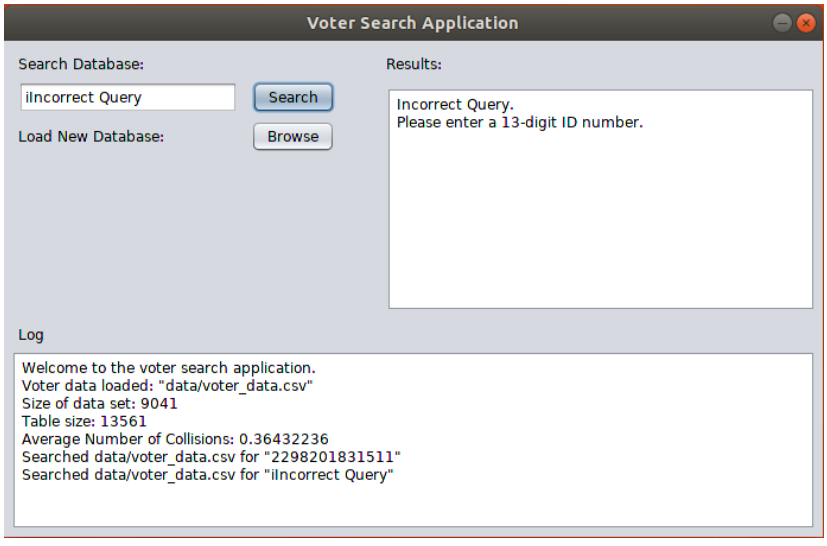
This is the home screen, upon opening the application loads the default data set from the path “data/voter_data.csv”. It size of the data set, table size, and average collisions in the hash table.



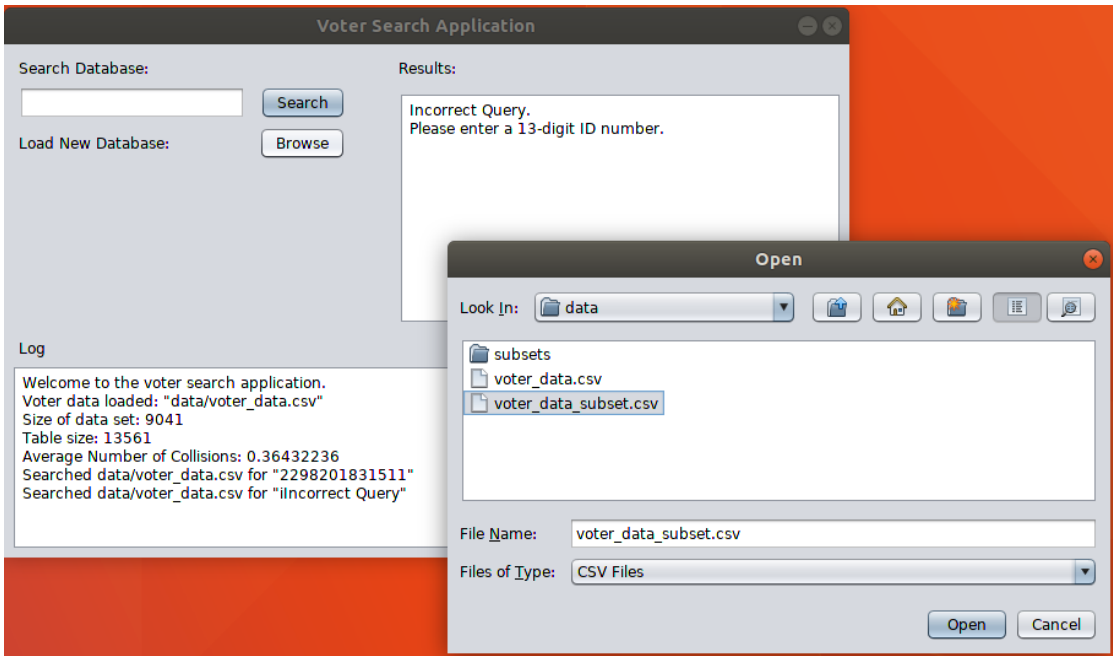
This is a screen shot of the text file containing the voter data next to the GUI, you should use the id numbers the id numbers to search.



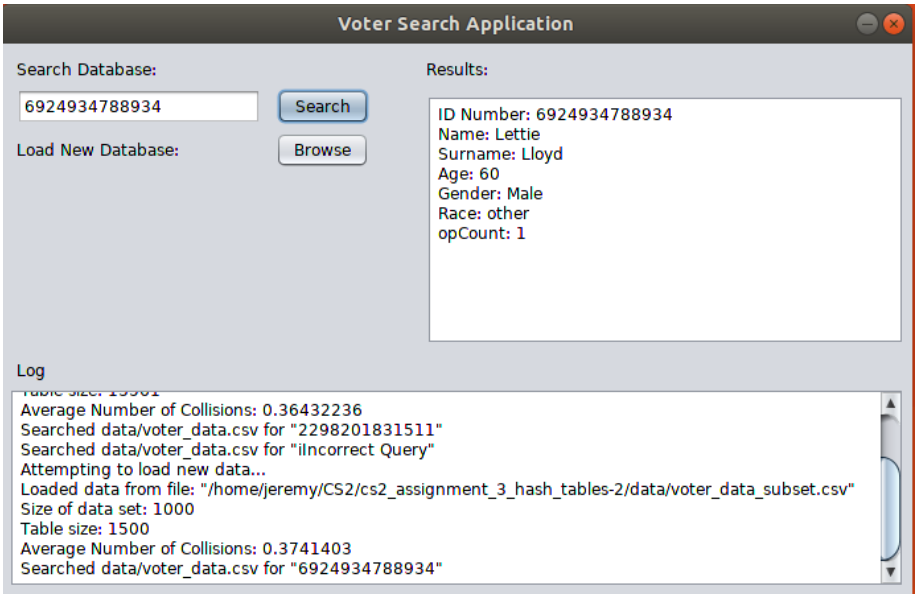
This is an example of a successful search for a voter contained in the dataset. The results show all the details of the voter including the opCount, which is the number of comparisons made when the search was executed. The logs records the search.



This is an example of an incorrect query due to an invalid search. A search must be a 13 digit numerical String for it to be considered legitimate. The Results field displays the error and the log records the search.



This is an example of how a new .csv file can be loaded, parsed and used as a dataset for searching. Clicking on the browse button takes the user to the home directory. There is a file in “data/voter_data_subset.csv” which will load correctly. A file of the incorrect format (without the specified columns and correct data types in each column) will fail to load and an error will be displayed.



This is the state of the application after the new dataset has been loaded, and a successful search has been executed. All activity is registered in the log panel.

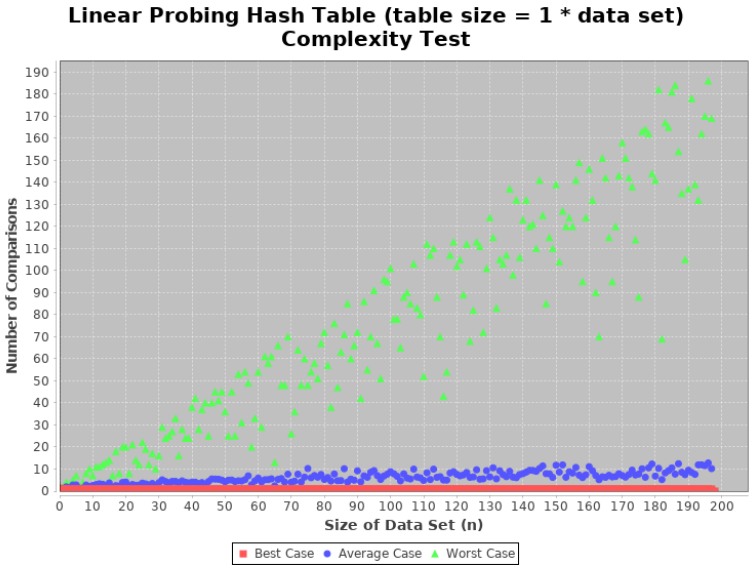
The guy can be run with the command *$make run_gui* from the command line in the project directory.

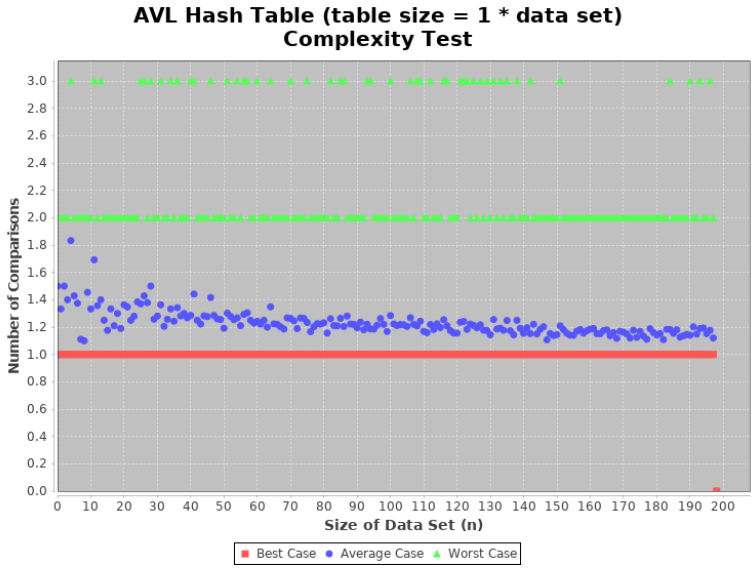
d. Complexity Tests: Linear Probing vs. AVL Tree Collision Resolution

In order to analyse the efficiency of the search operation executed by the hash table with the AVL collision mechanism, I ran efficiency tests against a different hash table which uses linear probing to resolve clashes. The process was an follows:

- Create 200 subsets of the voter data (from size n = 1 —> n = 200)
- For each subset:
  - The data is loaded into a hashTableAVLTest object and a hashTableLinearTest object
  - Choose the table size M = 1 * the size of the dataset N (a load factor of 1)
  - The average collisions for each are recorded upon insert
  - Search each array for every name contained in the array once, recording the number of comparisons (the opCount) each time
  - Calculate the best, average and worst case for comparisons
  - Write the results to file
- Repeat the process for a table size M = 1.5 * the size of the data set N (a load factor of 0.75)

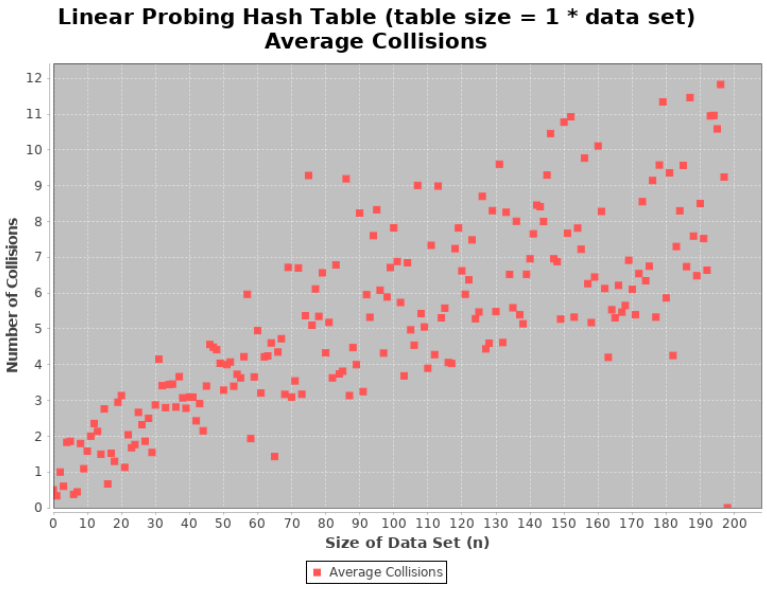
Here are the results of the tests for the table size **M = 1 * N**

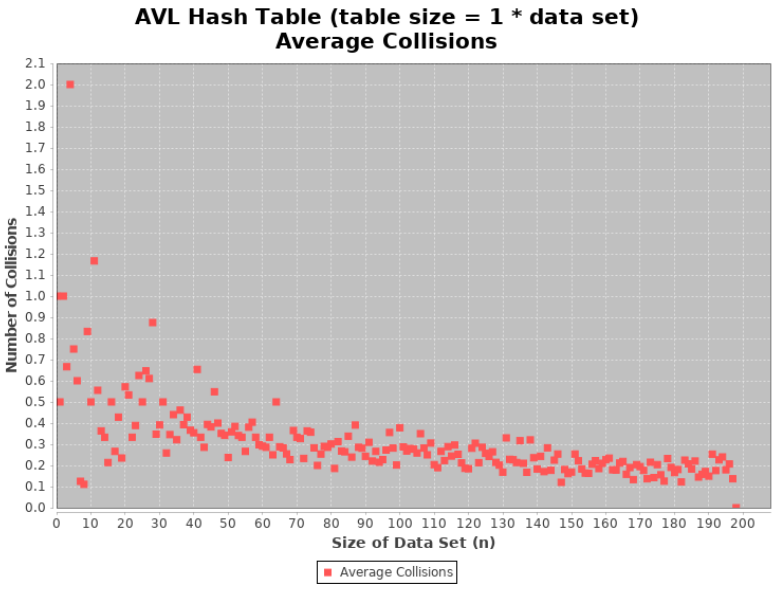




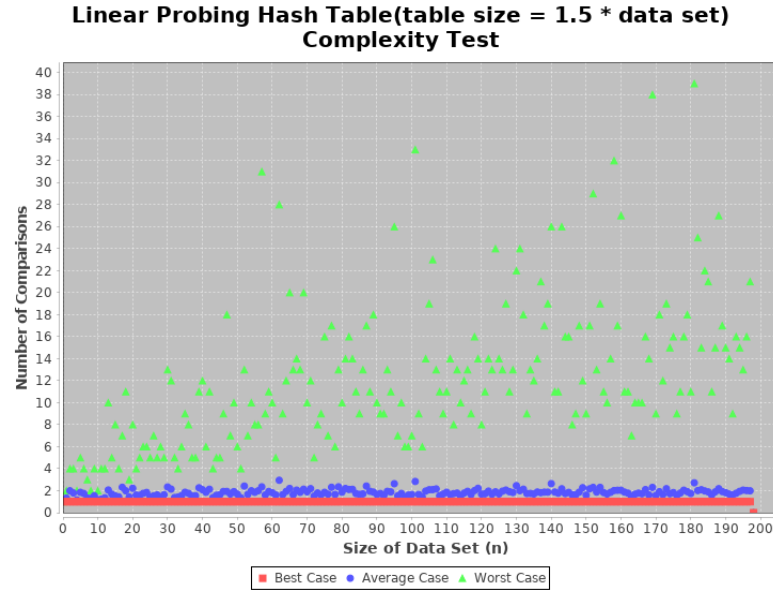
These results indicate that, with a load factor of 1, the linear probing algorithm has a search efficiency O(n) in the worst case - this is the case where the last element is inserted and the table is full. The average case is has a slightly positive correlation to the size of the data set, but is fairly consistent. The efficiency of the hash table with the AVL collision mechanism is incredibly efficient by comparison , despite the load factor of 1, due to the balancing algorithm of the AVL tree. The data shows the number of comparisons in the worst case is never higher than 3, which would still be O(1). The average collisions actually decreased with the size of the dataset and approach 1 asymptotically.

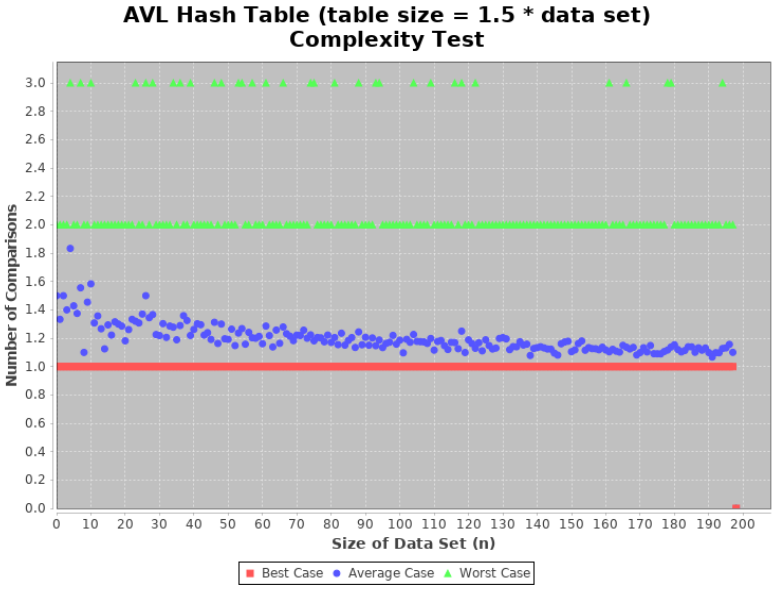
The data for the average collisions per dataset size (shown below) show similar trends. In the case of linear probing a collision was defined as the number of times (after the initial calculation) a new hash needed to be calculated before the item could be inserted into the array. The data on the linear probing shows a gradual upward trend as the size of the dataset is increased (again, with a load factor of 1), which corresponds with the above data on the search complexity. Likewise the average collisions for the AVL mechanism corresponds to the average case for search comparisons above - as the table size increases the modulus function allows for a more evenly distributed set of hash codes.





Here are the results of the tests for the table size **M = 1.5 * N**

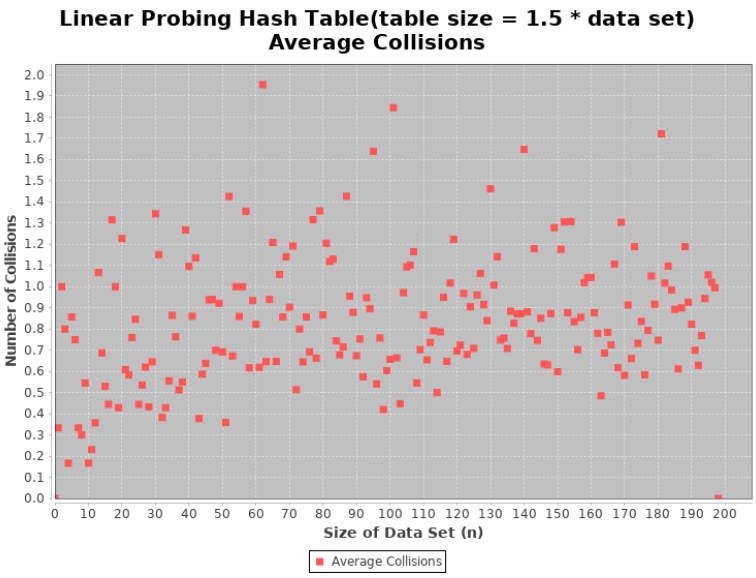


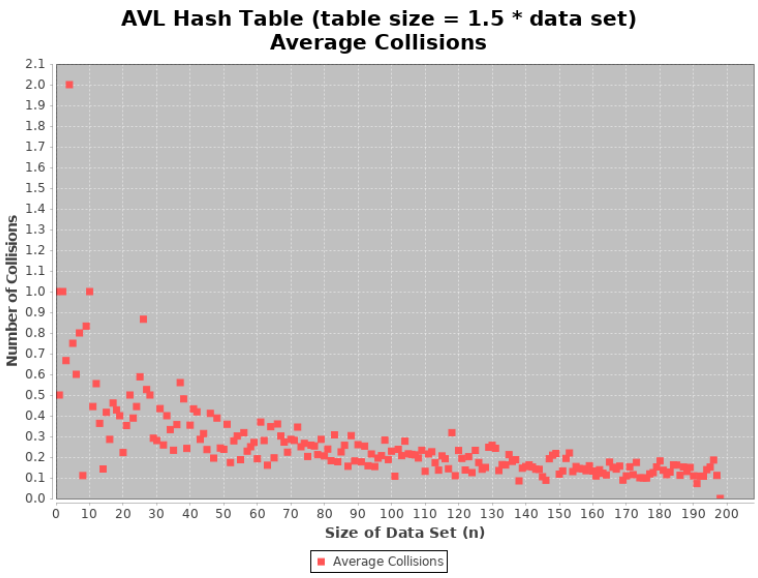


The results show that an increase in table size has dramatic effects on the time complexity of the search algorithm for the linear probing hash table. In the worst case the line of best fit (though the green points) would almost make a Log(n) shape. This makes sense as the number of empty spaces int he array would be 0.5 * N (the size of the data set), resulting in fewer collisions, and therefore fewer search comparisons. The increase in table size has a slightly positive effect on the efficiency of the search algorithm for the AVL hash table. In the worst has there are significantly fewer cases where 3 comparisons are made, and the average cases appear to have reduced, if only slightly, by around 0.1.

The results of the average collisions test below show exactly what we were expecting: the linear probing hash table has an average of less than on collision per data item due to the empty space, and the data shows no aggressive upward trend with an increase in the size of the data set as before. The results from the AVL Tree hash table are similar to the previous case, but with a slight decrease in the average number of collisions (approximately 0.1). The data also shows a downward trend as the size of the data set is increased, but levels out asymptotically to around 0.1.

The results of the complexity tests show that the linear probing mechanism works well on average when the load factor is around 0.75 or less, but its efficiency suffers greatly as the ratio (N/M) is





decreased. The AVL Tree collision mechanism proves a highly effective, dynamic strategy which appears to maintain the order of it's efficiency in all cases, regardless of table size.

e. Creative Components

I feel I went beyond the requirements of the assignment and demonstrated creativity in the following ways:

Extensive work designing GUI:

- Search function and error handling
- Comprehensive display of results
- Activity log (scroll pane) displaying information about the underlying hash table and datasets, as well as historical info about searches
- Functionality to load new data sets (csv files) into the application (and error handling for wrong files chosen)

Comparison of Different Hash Table Designs:

- Various complexity tests (opCounts during search and Average collisions) on different types of collision resolution mechanisms.
- Varying the size of data set and table size to ascertain interesting results.

f. Git log (from most recent)

I had some trouble with my git objects - they became corrupt just as I was finishing the assignment. I had pushed all my commits to but bucket so I went and copied the records from the online repo:

From 68c1c976965c951ca8d43d2b8baaec67c7bf6c3f
From: jeremy <dpljer001@myuct.ac.za>
Date: Tue, 17 Apr 2018 09:02:25 +0200
Subject: [PATCH] All results correctly and graphs accurate

From a43758b1b144e3817d4835f8f0a4d51d3c4bff01
From: jeremy <dpljer001@myuct.ac.za>
Date: Tue, 17 Apr 2018 00:55:42 +0200
Subject: [PATCH] graphs working

From 7a66440b330ef70f8de310ffad31d130a60ef6e4
From: jeremy <dpljer001@myuct.ac.za>
Date: Mon, 16 Apr 2018 23:59:08 +0200

Subject: [PATCH] tests working

From df81d2d076e5a094d0b01b33a0a7a10688b5d0a1
From: jeremy <dpljer001@myuct.ac.za>
Date: Mon, 16 Apr 2018 11:30:25 +0200
Subject: [PATCH] GUI working

From 910a03d51d158b1354071b2f3c804b270d2a9bd5
From: jeremy <dpljer001@myuct.ac.za>
Date: Wed, 11 Apr 2018 18:06:45 +0200
Subject: [PATCH] Hashtable test application working, started work on GUI

From de408cd7626e96183bb990836785f0723f49c93c
From: jeremy <dpljer001@myuct.ac.za>
Date: Tue, 10 Apr 2018 18:41:21 +0200
Subject: [PATCH] All basic classes written