**Applications and Empirical Analysis of Searching Algorithms**

**1.** **Scenario:**

We have an NTU student night which certain number of students have registered to. On the spot, we need to scan the coming student Matriculation ID and search whether it is registered. As brute force search is time consuming, we store all the registered IDs in Hash table and search.

Our implementation is written in Java with hash table size being 1000, data size (registered number of students) varying from 100, 300, 500, 700 to 900.

**2.** **Data Set:**

Matriculation number has 9 characters with the first and last being capitalized English character(A-Z) and the middle ones being digits (0-9). E.g. U1520636J. In this experiment, “random\_matricNo” and “random\_application” classes utilize “java.util.Random” class to provide random matriculation number of the size specified.

**3.** **Hashing Algorithms:**

We have implemented three Hashing Algorithms, namely closed address hashing, open address hashing (linear probing) and open address hashing (double hashing). In this following part, we will introduce the common hashing function across the three algorithms, and introduce each algorithm in detail.

**3.1**  **Hashing function**

**public** **int** hash\_func(String matricNo){

**long** hash = 3;

**for** (**int** i = 0; i < matricNo.length(); i++) {

      hash = hash\*11 + matricNo.charAt(i);

      }

**int** hash\_value = (**int**)(hash % 997);

**return** hash\_value;

 }

As shown above, our hashing function uses 3 prime number in initialization, addition and mod operation, to ensure the randomness of the resulting index, given any matriculation data set. In the addition process (for loop), it gives different weightage to different digits, so that it ensures randomness even when the matriculation numbers given are in the same combination of characters.

**3.2**  **Closed Addressing Hashing Algorithm**

In Closed Addressing Hashing, the hash table is a reference to an array of table\_size, with each element in the array being a LinkedList. An element is inserted into the end of corresponding linkedlist, if the linkedlist is not empty. Otherwise it will be the first element of that linkedlist. An element is searched throughout the corresponding linkedlist, in searching process.

Followings are the results in terms of average CPU time and number of comparisons in searching one matriculation number. The number of test search cases are 30000 to ensure the average performance.

Unsuccessful search example:

datasize: 100

 CPU time: 108 ns

 comparisons: 0.1

datasize: 300

 CPU time: 70 ns

 comparisons: 0.3

datasize: 500

 CPU time: 72 ns

 comparisons: 0.5

datasize: 700

 CPU time: 76 ns

 comparisons: 0.7

datasize: 900

 CPU time: 75 ns

 comparisons: 0.9

successful search example:

datasize: 100

 CPU time: 119 ns

 comparisons: 1.1

datasize: 300

 CPU time: 67 ns

 comparisons: 1.1

datasize: 500

 CPU time: 73 ns

 comparisons: 1.2

datasize: 700

 CPU time: 72 ns

 comparisons: 1.4

datasize: 900

 CPU time: 95 ns

 comparisons: 1.4

**3.3**  **Open Addressing Linear Probing**

In open addressing, the hash table is an array of strings. Initially all strings are set to be “empty” until they are filled with explicit matriculation number. At collision, the new address is just one more than the previous address or from the last address to first address. Followings are the results in terms of average CPU time and number of comparisons in searching one matriculation number. The number of test search cases are 30000 to ensure the average performance.

Unsuccessful search example:

datasize: 100

 CPU time: 59 ns

 comparisons: 1.1

datasize: 300

 CPU time: 63 ns

 comparisons: 1.5

datasize: 500

 CPU time: 74 ns

 comparisons: 2.6

datasize: 700

 CPU time: 88 ns

 comparisons: 5.4

datasize: 900

 CPU time: 214 ns

 comparisons: 33.8

Successful search example:

datasize: 100

 CPU time: 107 ns

 comparisons: 2.2

datasize: 300

 CPU time: 76 ns

 comparisons: 2.7

datasize: 500

 CPU time: 79 ns

 comparisons: 3.6

datasize: 700

 CPU time: 106 ns

 comparisons: 8.2

datasize: 900

 CPU time: 288 ns

 comparisons: 42.3

**3.4**  **Open Addressing Double Hashing**

In open addressing, the hash table is an array of strings. Initially all strings are set to be “empty” until they are filled with explicit matriculation number. At collision, the new address is given by:

hash\_value = (hash\_value + inc\_hash) % 997;

with the inc\_hash value being:

**private** **int** double\_hashing(String matricNo){

**long** hash = 7;

**for** (**int** i = 0; i < matricNo.length(); i++) {

      hash = hash\*17 + matricNo.charAt(i);

      }

**int** inc\_hash = (**int**)(hash % 991) + 3;

**return** inc\_hash;

  }

Followings are the results in terms of average CPU time and number of comparisons in searching one matriculation number. The number of test search cases are 30000 to ensure the average performance.

Unsuccessful search:

datasize: 100

 CPU time: 72 ns

 comparisons: 1.1

datasize: 300

 CPU time: 68 ns

 comparisons: 1.4

datasize: 500

 CPU time: 77 ns

 comparisons: 2.0

datasize: 700

 CPU time: 83 ns

 comparisons: 3.4

datasize: 900

 CPU time: 133 ns

 comparisons: 10.2

Successful search:

datasize: 100

 CPU time: 94 ns

 comparisons: 2.1

datasize: 300

 CPU time: 83 ns

 comparisons: 2.5

datasize: 500

 CPU time: 83 ns

 comparisons: 2.9

datasize: 700

 CPU time: 86 ns

 comparisons: 3.8

datasize: 900

 CPU time: 116 ns

 comparisons: 8.0

**4.** **Comparisons and Conclusion:**

Showing below is the consolidated result. Comparing across successful & unsuccessful searches with implementation of Closed Addressing, Open Addressing (Linear Probing) and Open Addressing (Double Hashing). We find the Closed Addressing has better performance compared to two open addressing method on both successful and unsuccessful searching data sets. In terms of two open addressing hashing algorithms, Double hashing outperforms Linear probing in all data sets.

Another observation is Closed addressing needs less comparisons on unsuccessful data sets compared with successful cases. However, Linear probing & Double hashing has the opposite condition, they both finish searching faster on successful cases.

So our conclusion is that in terms of this scenario, Closed Addressing is the best, followed by Double Hashing and the worst is Linear Probing.

