****

**CZ2001 Algorithms Example Class 2**

**Report**

**SS3 Team 2**

**Joshua Martin Tjahjadi**

**Kwek Ai Hui**

**Jason Lee**

**Shantanu Kamath**

**Jordan Kwong**

**Suyash Lakhotia**

SCHOOL OF COMPUTER ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

Contents

[1. INTRODUCTION 3](#_Toc429996573)

[1.1 Problem Domain 3](#_Toc429996574)

[1.2 Data Set 3](#_Toc429996575)

[2. IMPLEMENTATION 4](#_Toc429996576)

[2.1 Linear Probing 4](#_Toc429996577)

[2.2 Double Hashing 6](#_Toc429996578)

[3. STATISTICS 8](#_Toc429996579)

[3.1 Linear Probing 8](#_Toc429996580)

[3.2 Double Hashing 8](#_Toc429996581)

[4. CONCLUSION 9](#_Toc429996582)

[5. REFERENCE 10](#_Toc429996583)

# 1. INTRODUCTION

*Description of the problem domain and data sets*

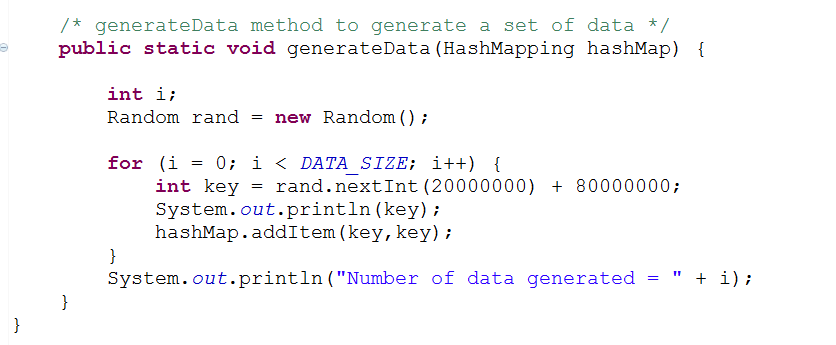
## 1.1 Problem Domain

There are many real world problems that requires searching through a huge amount of data. Using an example that is common in today’s society, we decided to look into how storing and searching mobile phone numbers are carried out.

A mobile phone number has eight digits. Searching and matching the mobile phone number to a student can be very tedious and inefficient if we were to search through the data in ascending order. To improve the efficiency of this process, linear-probing hashing and double hashing are good ways to store the mobile phone numbers, as well as using binary search to search for a mobile phone number.

## 1.2 Data Set

For this experiment, since real world data sets are not available, synthetic data sets are generated using pseudo random algorithm. Since Singapore’s mobile phone numbers start from the digit ‘8’ or the digit ‘9’, the mobile phone numbers generated will range from 80000000 to 99999999. Fig 1 below illustrates how this will be done.



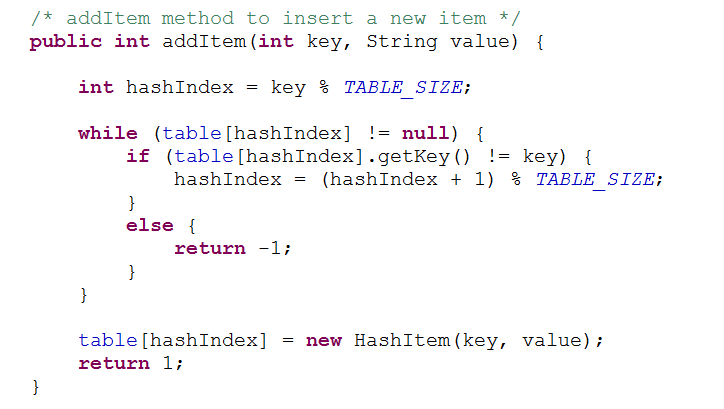
**Fig 1: Generating data sets**

The data size will range from 100, 300, 500, 700 and 900. For each data set, 10 test cases will be carried out where the key comparisons will be considered. The average CPU computational time will then be calculated. Last but not least, the time complexity of Linear-probing Hashing and Double Hashing is compared and a conclusion of which hashing algorithm is better will be made.

# 2. IMPLEMENTATION

## 2.1 Linear Probing

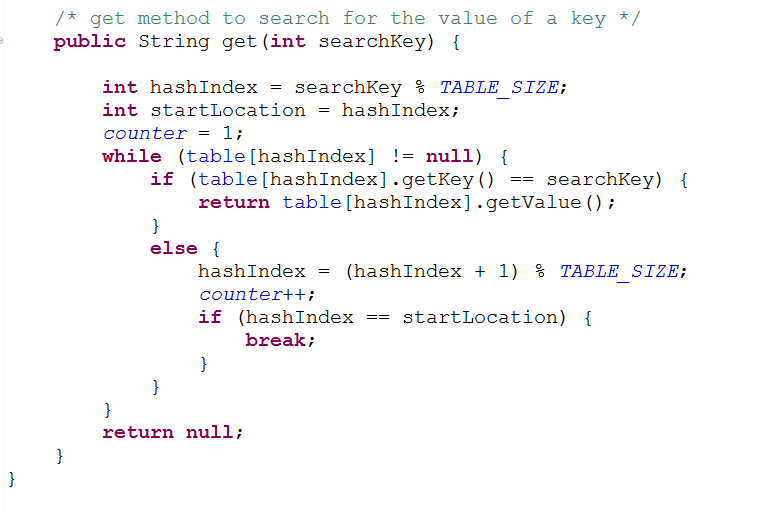
Linear probing is the simplest rehashing method. As the load factor n/h is never greater than 1, collision is handled by rehashing, a process to look for an alternative slot. Fig 2 below illustrates how this will be done.



**Fig 2: Linear Probing - Store**

As seen in Fig 2, if table[hashIndex] is filled, the next mobile phone number stored will be directly below that filled slot.

After storing the mobile phone numbers into the hash table, we will also need to retrieve the phone number by searching with the same algorithm. There are two outcomes of this search: Success or fail. Linear probing will go on until the algorithm reaches the starting point of the search. Fig 3 illustrates how this will be done.

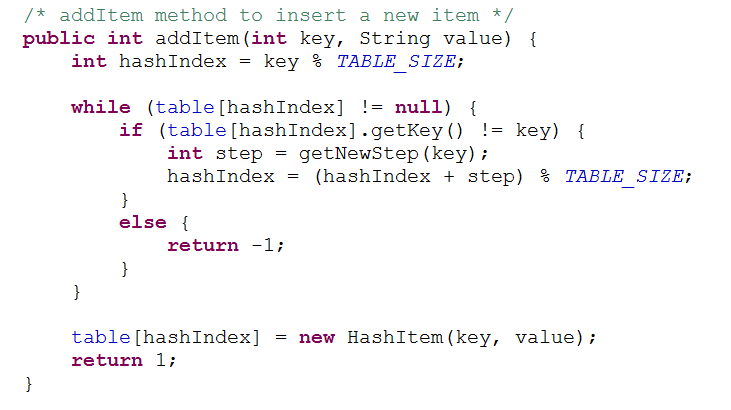


**Fig 3: Linear Probing - Search**

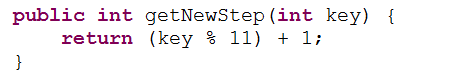
As seen in Fig 3, if table[hashIndex] is the same as the key (mobile phone number), the search will be successful. Else, the search will go on by comparing the search key with the key in the next slot. This search will go on until the comparing goes back to the starting position of the search, which it will then be considered as a fail search.

## 2.2 Double Hashing

Double Hashing solves the problem of primary clustering caused by linear probing – Long runs of occupied slots. This makes the searching expensive as the load factor approaches 1. As a result, a different algorithm is needed to make sure that the keys are more widely spread. This is illustrated in Fig 4 below.



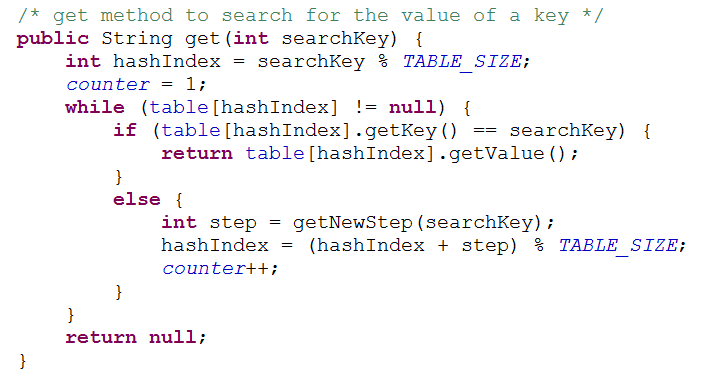
**Fig 4a: Double Hashing – Store**



**Fig 4b: getNewStep method**

As seen in Fig 4b, an additional step of ‘(key%11) + 1;’ is added so that if the slot is filled, a different hash index is obtained to store the key somewhere else in the hash table.

Similarly, after storing the mobile phone numbers into the hash table, we will also need to retrieve the phone number by searching with the same algorithm. Fig 5 illustrates how this will be done.



**Fig 5: Double Hashing - Search**

As seen in Fig 3, if table[hashIndex] is the same as the key (mobile phone number), the search will be successful. Else, the search will go on by comparing the search key with the key in the next slot. This search will go on until the comparing goes back to the starting position of the search, which it will then be considered as a fail search.

# 3. STATISTICS

## 3.1 Linear Probing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Size | Unsuccessful | | Successful | |
| Number of Comparison | Average CPU time (nanoseconds) | Number of Comparison | Average CPU time (nanoseconds) |
| 100 | 1 | 13388 | 1 | 5801 |
| 300 | 2 | 23204 | 2 | 4909 |
| 500 | 2 | 18628 | 4 | 17829 |
| 700 | 3 | 24098 | 6 | 17140 |
| 900 | 4 | 21866 | 10 | 19371 |

## 3.2 Double Hashing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data Size | Unsuccessful | | Successful | |
| Number of Comparison | Average CPU time (nanoseconds) | Number of Comparison | Average CPU time (nanoseconds) |
| 100 | 1 | 12941 | 2 | 8479 |
| 300 | 1 | 16958 | 1 | 6247 |
| 500 | 5 | 27667 | 3 | 17850 |
| 700 | 2 | 16065 | 3 | 15823 |
| 900 | 2 | 23651 | 7 | 18923 |

# 4. CONCLUSION

Since both the methods belong to Open Address Hashing, we can say that:

**Best Case: All the keys in odd slots and even slots are empty;**

The average number of probes for an unsuccessful search is constant when n is proportional to h

i.e. h=2n then the average is 0.5 (constant ) which is the load factor, so it has a constant complexity, namely O(1)

**Worst Case: All the keys in one half of the table, another half is empty.**

For an unsuccessful search the average number of probes is given by:



Through the statistics and the time complexities given earlier we can see that between the two open address hashing methods, the linear probing method is faster for a smaller data set while double hashing is faster for a bigger data set.

We further conclude that due to the fact that linear probing matches the data to the next available slot, there is a high chance of clustering that increases the probability of the worst-case scenario. This also increases the length of the linked lists so as to let the load factor increase substantially. We find that double hashing does not have this problem and spreads its data in sprinkled order and provides a better range hence making the time complexity better.

Our statistics show that double hashing is more suited to handle collisions and increase efficiency than the linear probing method. We note that with an increase in the data size (n) the number of comparisons and time grow at slower rate as compared to linear probing. But we would also like highlight the fact that though we see changes in the execution time that supports our conclusion, CPU time isn’t a good measure of efficiency or complexity as it varies in each computer. Some computers are faster and stronger than others.

Hence we conclude that though **double hashing** may look complex and inefficient in its implementation, **it is better than linear probing** as it can be seen when the data size gets bigger and bigger.

# 5. REFERENCE

1. Lecture materials on NTULearn.