CZ2001 Algorithms Example Class 2

Report

SSP4 Team 3

Tan Wei Xuan (U1721260J)

Datta Anusha (U1822948G)

Prashant Mohit (U1823630E)

Han Jun (U1820665L)

Kumar Mehul (U1822146E)

Ho Shu Peng Xavier (U1822670E)

Wang Yifan (U1821102B)

SCHOOL OF COMPUTER ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

Contents

1. INTRODUCTION 3
    1.1 Problem Domain 3
    1.2 Data Set 3
2. IMPLEMENTATION 4
    2.1 Linear Probing 4
    2.2 HASHING FUNCTIONS 6
        2.2.1 Division Method 6
        2.2.2 Shift Folding 6
        2.2.3 Mid-Square Method 8
3. STATISTICS 9
    3.1 Division Method 9
    3.2 Shift folding 9
    3.3 Mid-Square Method 10
4. CONCLUSION 13

1. INTRODUCTION

Description of the problem domain and data sets

1.1 Problem Domain

There is a myriad of real world problems that requires searching through a vast amount of data. Using an example in today's society, our team decided to look into how storing and searching of books are being carried.

A book can be uniquely identified by an **I**nternational **S**tandard **B**ook **N**umber (ISBN). An ISBN is a 13-digit numeric commercial book identifier that identifies a specific book's title, edition, format and its registrant. Every ISBN begins with a prefix that can only be either 978 or 979, and the prefix is always 3 digits in length.

Searching and matching an ISBN to their respective book title can be a very tedious and inefficient process if we were to search through the data sequentially. To improve the efficiency of this process, linear probing hashing can be used for storing and searching of these ISBN numbers. Our team will be analyzing the efficiency of the linear-probing hashing method with three different hashing functions - Division method, Shift Folding and Mid-Square Method.

1.2 Data Set

For this experiment, since real world data sets are not available, synthetic data sets are being generated using a pseudo random algorithm. Since the first 3 digits of an ISBN can be either '978' or '979', the ISBNs generated will range from 9780000000000 to 9799999999999. Fig. 1 below illustrates how this will be done.

```
//ISBN numbers are 13-digits long and will always start with a prefix of 978 or 979
private final BigInteger upperLimit = new BigInteger("9799999999999"); // Upper Limit
private final BigInteger lowerLimit = new BigInteger("9780000000000"); // Lower Limit
private final BigInteger diff;
private final int maxNumBitLength;
public void generateData(HashMap hashMap,int hashFuncChoice,double loadFactor) throws IOException {

    int dataCounter = 0;
    BigInteger key;

    //Generate the amount of data according to dataSize;
    while(dataCounter < data_size) {

        key = new BigInteger(maxNumBitLength, rand);

        if (key.compareTo(lowerLimit) < 0)
              key = key.add(lowerLimit);
        if (key.compareTo(upperLimit) >= 0)
              key = key.mod(diff).add(lowerLimit);

        System.out.println(key);
        String name = br.readLine();
        System.out.println("\t : " + name);

        // Add the data into the hash table
        if(hashMap.addItem(key, name, hashFuncChoice) == 1) {
            //List of all added Keys
            addedKeys.add(key);
            dataCounter++;
        }
    }
```

Fig 1. Generating Data set

The data size will be calculated based on the chosen load factor *(For the purpose of this assignment, we will be using load factors of 0.10, 0.30, 0.50, 0.70 and 0.90).* For each data set, 10 repetitions of randomly generated test cases *(equivalent to the data size)* will be carried out for each of the hashing function, where the number of key comparisons and CPU computational time for both successful and unsuccessful searches will be considered. The average number of key comparison and the respective average CPU computational time for each of the hashing function will be analyzed and a conclusion on which hashing function 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. If collision occurs, linear probing searches the table for the closest following free location and inserts the new key there. Fig 2 below will illustrate how this will be done.

```
public int addItem(BigInteger key, String value, int hashFuncChoice) {

    // Calculate a Hash Index for the key

    int hashIndex = calculateHashFunction(key, hashFuncChoice,0);

    //If the slot is not empty
    while (hash_table[hashIndex] != null) {
        // If the item in the slot is not equal to the key we want to insert
        if (hash_table[hashIndex].getKey() != key) {
            //rehash by moving to the next slot. Linear Probing adds item sequentially and thus we have to +1 to the key
            //hashIndex will increase by from i = 1 to HASHTABLE_SIZE - 1
            hashIndex = (hashIndex + 1) % HASHTABLE_SIZE;
        }
        else { // The item in the slot is same as the key we want to insert, we do not have to add the key again
            return -1;
        }
    }
```

Fig 2: Linear Probing: Inserting an item

From Fig 2, if *hash_table[hashIndex]* is filled, the next ISBN number will be stored directly below that filled slot.

Lookups are performed in the same way, by searching the table sequentially starting at the position given by the hash function, until finding a cell with a matching key or an empty cell. There are two outcomes of these lookups: Success or Fail. Linear Probing will go on until the algorithm either reaches the starting point of the search or reaches an empty slot. Fig 3 on the next page illustrates how this will be done.

```java
//Search for a value associated with the given key
public String get(BigInteger searchKey, int hashFuncChoice) {

    // Calculate a Hash Index for the key
    int hashIndex = calculateHashFunction(searchKey, hashFuncChoice,1);

    //Start location of the search
    int startLocation = hashIndex;

    //Counter to track the total number of Comparison
    counter = 1;

    //If the slot is not empty
    while (hash_table[hashIndex] != null) {
        // If the item in the slot is equal to the search key
        if ((hash_table[hashIndex].getKey()).intValue() == searchKey.intValue()) {
            //Return the value associated with the key
            return hash_table[hashIndex].getValue();
        }
        // If the item in the slot is not equal to the search key
        else {
            //Rehash by moving the search to the next slot.
            // Linear Probing searches for item sequentially and thus we have to +1 to the key
            hashIndex = (hashIndex + 1) % HASHTABLE_SIZE;
            //Increase the Comparison Counter by 1
            counter++;

            // If the search is unsuccessful (Hash Index becomes the same as the index that we started our search at)
            if (hashIndex == startLocation) {
                break;
            }
        }
    }
    return null;
}
```

Fig 3: Linear Probing: Search for an Item

As seen in Fig 3, if *table[hash_index]* is the same as the key (ISBN) that we are searching for, the search will be successful. Else, the search will continue by comparing the key with the key in the next slot. The search will go on till the *hash_index* goes back to the starting position of the search or till an empty slot in the table is being encountered.

2.2 HASHING FUNCTIONS

We will be comparing the efficiency of the Linear Probing hashing with the use of 3 different Hash Functions. The 3 Hashing Functions that we will be using will be the Division Method, Shift Folding and Mid-Square Method.

2.2.1 Division Method

The division method is a hash function that divides an integer key by the hash table size and the remainder will be taken as the hash value. The algorithm for the Division Method is as follows:

$$H(k) = k \ mod \ m$$

Where: m is some predetermined divisor integer (ie. the table size), k is the preconditioned key and mod stands for modulo

Fig 4 below illustrates how this is done.

```
case 1:

    hashIndex = (key.mod(hashtable_size_big)).intValue();

    break;
```

Figure 4: Division Method

From Fig 4, We divide our search key (ISBN) with the size of our Hash Table and set the hash value as the remainder.

2.2.2 Shift Folding

The shift folding method breaks up a key into precise segments, with the exclusion of the last segment being able to be of a different size. These segments are then added together to form a hash value. The division method is applied after and the hash key will be the calculated remainder value. The algorithm for the Shift Folding method is as follows:

$$H(k) = (a + b + \cdots + n) mod \ m$$

$$Where: a, b \ldots n \ represents \ the \ preconditioned \ key \ broken \ down \ into \ n \ parts, 'm' \ is \ the \ hash \ table \ size,$$

$$and \ mod \ stands \ for \ modulo.$$

$$Example: With \ a \ \ table \ size \ of \ 11, k = 3205 \ and \ number \ of \ segments = 3$$

$$H(3205) = (32 + 05 + 07) \ mod \ 11 = 37 \ mod \ 11 = 4$$

Fig. 5 below illustrates how the Shift Folding function is being implemented.

From Fig 5, we first split up the key into 5 segments *(keyPiecesStr.split("(?<=\\G.{3}"))*, with 3 elements within each segment. As the ISBN is a 13-digit numerical value, we will have 4 equal segments comprising of 3 elements and 1 unequal segment comprising of 1 element. We then sum up these segments (*int keyPiecesSum = IntStream.of(keyPiecesIntArray).sum();)* to obtain a hash value. Lastly, we divide the calculated hash value with the size of our Hash Table and set the hash key as the remainder.

```
//Folding Method
case 2:

    String keyPiecesStr = key.toString();

    //Split Key Pieces into 4 Equal Size Pieces and 1 Unequal Size Piece (Last Piece)
    String[] keyPiecesStrArray = keyPiecesStr.split("(?<=\\G.{3})");

    int[] keyPiecesIntArray = new int[keyPiecesStrArray.length];

    for (int i=0; i < (keyPiecesStrArray.length); i++) {
        keyPiecesIntArray[i] = Integer.parseInt(keyPiecesStrArray[i]);
    }



    int keyPiecesSum = IntStream.of(keyPiecesIntArray).sum();
    if(choice == 0) {
        System.out.println("Key Pieces:" + java.util.Arrays.toString(keyPiecesStrArray));
        System.out.println("Sum of Key Pieces: " + keyPiecesSum);
    }


    hashIndex = keyPiecesSum % HASHTABLE_SIZE;

    break;
```

Figure 5: Shift Folding

2.2.3 Mid-Square Method

In the Mid-Square method, the algorithm is as follows:

1. Square the Key (Seed Value)
2. Extract some digits from the middle (Same position must be used for all the keys)
3. Extracted digits form a new seed
4. Repeat from step 1 as many times as a key is required
5. Final seed will be the hash value

Example:

$$seed: 3205 \qquad seed^2: 10272025 \qquad seed': 72 \qquad seed'^2: 5184 \qquad seed'' : 18$$

Fig 6 below illustrates the implementation of the Mid-Square Method.

```
//Mid-Square Method
case 3:

    BigInteger seed = key;

    seed = seed.pow(2);
    BigInteger dividend = new BigInteger("10000000000");
    BigInteger modulo = new BigInteger("1000");

    //Extract a seed ( Middle 3 Digits)
    seed = seed.divide(dividend);
    seed = seed.mod(modulo);

    BigInteger loopDividend = new BigInteger("100");
    BigInteger loopModulo = new BigInteger("1000");

    //Repeat the process for N times. We set it as 3 for now.
    for(int i = 0;  i < 2 ; i ++) {
        seed = seed.pow(2);

        seed = seed.divide(loopDividend);
        seed = seed.mod(loopModulo);

    }

    hashIndex = seed.intValue();

    break;
```

Figure 6: Mid-Square Method

We square our seed value (ISBN) and extract the middle 3 digits from the result. We use a different set of dividends and modulo to ensure that the extracted seed will always be from the same position. The process is repeated for 3 times.

3. STATISTICS

3.1 Division Method

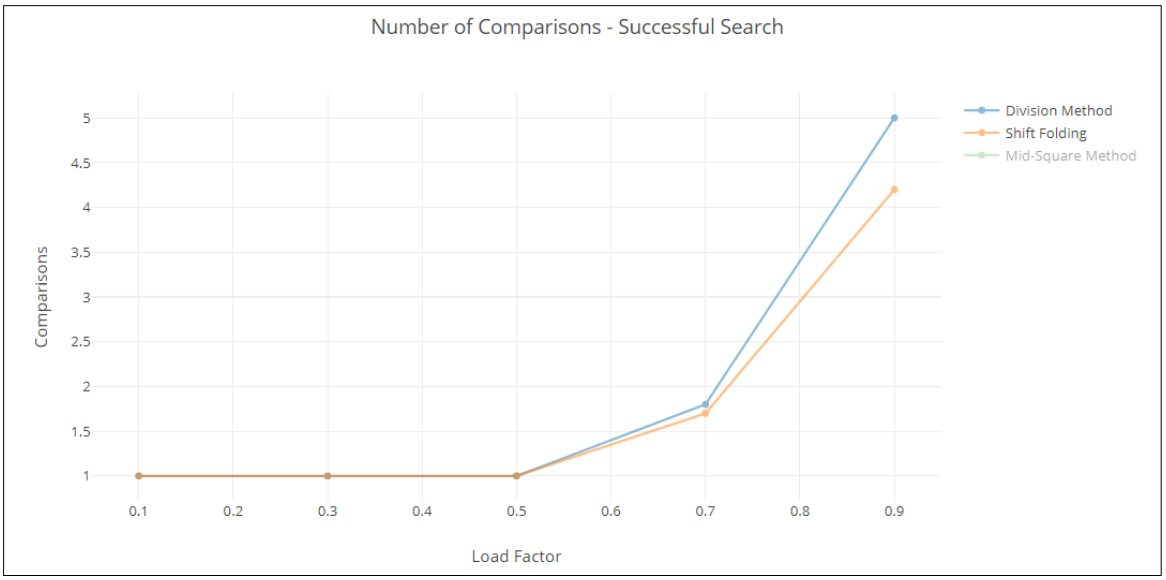
Load Factor	Data Size	Successful		Unsuccessful	
		Average Number of Comparisons	Average CPU time (nanoseconds)	Average Number of Comparisons	Average CPU time (nanoseconds)
0.1	100	1	8621.3	1	1111.4
0.3	302	1	4014	1	1083.7
0.5	504	1	648.8	2.1	867.5
0.7	706	1.8	711.4	4.9	1451.9
0.9	908	5	665.8	44.8	2630.3

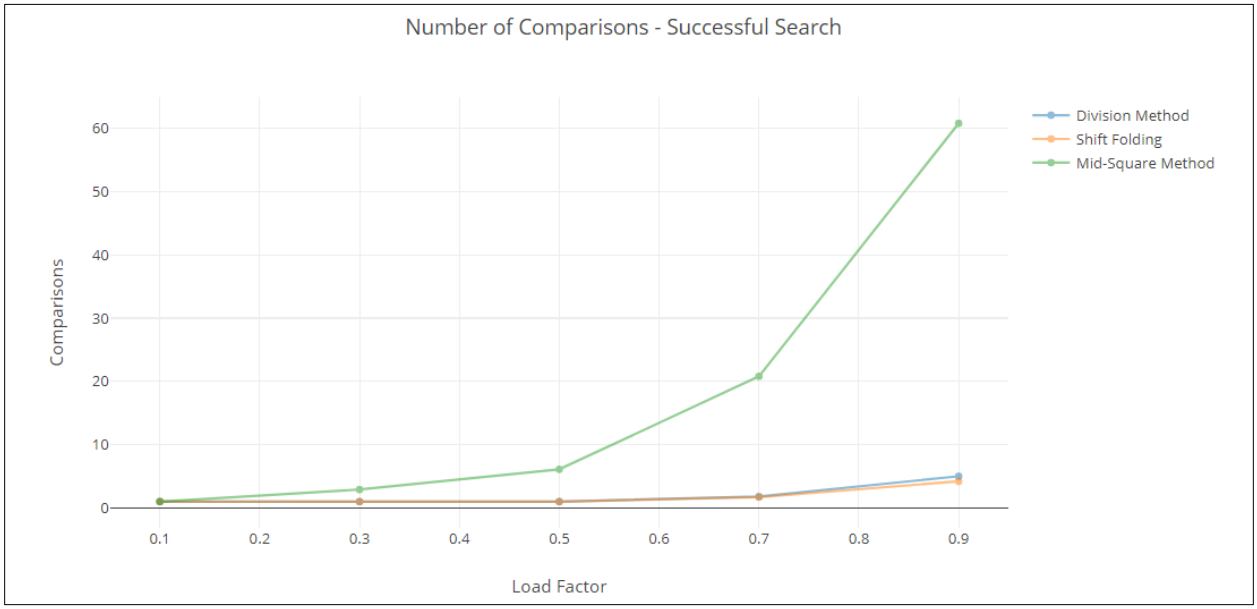
3.2 Shift folding

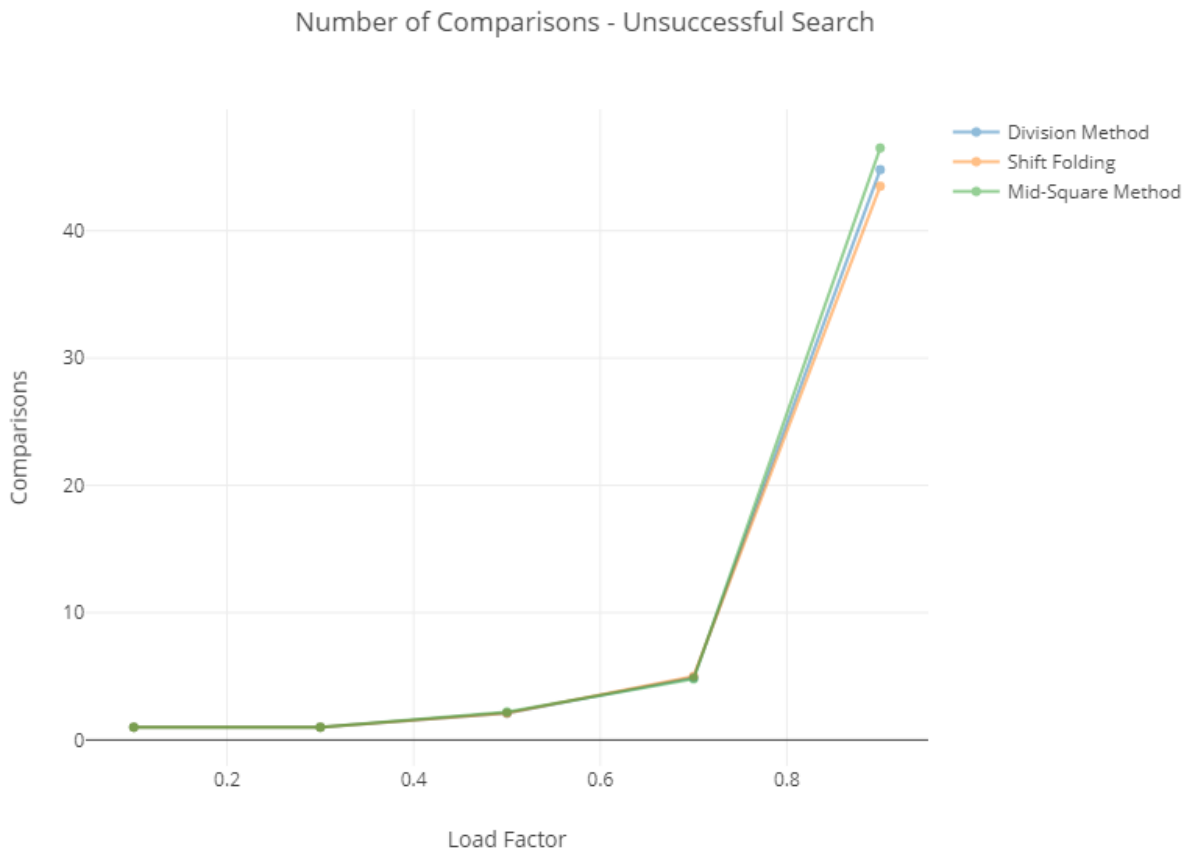
Load Factor	Data Size	Successful		Unsuccessful	
		Average Number of Comparisons	Average CPU time (nanoseconds)	Average Number of Comparisons	Average CPU time (nanoseconds)
0.1	100	1	32288.6	1	27967.6
0.3	302	1	25285.3	1	17186.5
0.5	504	1	9613.9	2.1	9834.5
0.7	706	1.7	6606.1	5	5479.4
0.9	908	4.2	7742.9	43.5	7362.4

3.3 Mid-Square Method

Load Factor	Data Size	Successful		Unsuccessful	
		Average Number of Comparisons	Average CPU time (nanoseconds)	Average Number of Comparisons	Average CPU time (nanoseconds)
0.1	100	1	38029.1	1	17952.6
0.3	302	2.9	8353.9	1	6075.6
0.5	504	6.1	5334.8	2.2	4086.2
0.7	706	20.8	6138.9	4.8	3473.3
0.9	908	60.8	8783.3	46.5	5074


Number of Comparisons - Successful Search
Comparisons
5
4.5
4
3.5
3
2.5
2
1.5
1
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Load Factor
Division Method
Shift Folding
Mid-Square Method


Number of Comparisons - Successful Search
Comparisons
60
50
40
30
20
10
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Load Factor
Division Method
Shift Folding
Mid-Square Method



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. Suppose the hash table has a load factor of 0.5, that is h = 2n. $\frac{n}{2}$ keys will require 1 comparison.

$$\frac{1}{n}\sum_{i=1}^{\frac{n}{2}} (1) = \frac{1}{n} \times \frac{n}{2} = 0.5 = O(1)$$

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

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

$$\frac{1}{2n}(\sum_{i=1}^{n} i + 0) = \frac{1}{2n} \times \frac{n(n+1)}{2} \approx \frac{n}{4} = O(n)$$

Through the statistics and time complexities given earlier, we can see that the Shift Folding hashing functions performs the best in the aspect of having the least number comparisons as the load factor tends to 1.

However, as both Shift Folding and Mid-Square methods perform poorly in terms of CPU execution time as compared the Division Method. This is likely due to the fact that the Division Method only comprises of a single divisor in its hashing function while both the Shift Folding and Mid-Square hashing methods comprises of loops within their respective hashing functions. However, although there are 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.

The number of comparisons using the Mid-Square Method increases the most substantially with an increase in the data size (n). This is due to the fact with the Mid-Square Method, the seed has a tendency to converge, most likely to zero. This, along with how linear probing matches data to the next available slot, leads to a higher change of clustering.

We can conclude that between three hashing functions analyzed, the Division Method is more efficient if we require faster computational rate while the Shift Folding method is better for avoiding collisions.