LAB GROUP: SS4
GROUP: 2

CZ2001: ALGORITHMS

Example Class 2

by

Alvin Lee Yong Teck (U1620768F)

Htet Naing (U1620683D)

Lim Kian Hock, Bryan (U1620949L)

Liu Jihao, Bryan (U1621401C)

Lee Shao Wei, Luke (U1620850A)

Tay Qi En, Zacharias (U1620760H)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

Description of problem scenario

Each reservation ID is assigned to a unique movie ID number for identification. Each reservation ID is tagged to a seat identified by the movie ID. In this movie theatre, there is a maximum capacity of 1013 seats.

There may be cases where two movie ID may be assigned to the same reservation ID. In that case, we would need to come out with an algorithm that enable us to decide on where to assign the two movie IDs to the reservation ID in an orderly and efficient manner.

Chosen hashing algorithms

We will be mainly looking at Open Addressing hashing algorithms namely: Linear Probing & Double Hashing.

The fixed hash table size and the multiple for the data in the data set will be both prime numbers. The hashing algorithms are more efficient if both the hash table size and the elements in the data set are prime numbers as there will be lesser common factors between each other which will prevent the number of collisions during storing and searching. Additionally, since we will be using Open Addressing, the Load Factor should never exceed 1 as the size of hash table is fixed and cannot occupy any more keys for that particular slot which will lead to rehashing.

Our default hash function is chosen as follows:
f(k) = k mod 1013

Should the default hash function return a value which results in a collision, both Linear Probing and Double Hashing have their unique collision handling protocols, through rehashing.

Linear Probing

This algorithm uses a simple rehashing method to resolve collisions done with the default hash function. It sequentially searches through the table till the first empty space is found, through a default increment of 1.

Our chosen rehash function is as follows:
reHash (j) = (j + 1) mod 1013

Double Hashing

This algorithm employs a second hash function, with the first to get a new hash key. It is generally a more efficient algorithm, albeit a more complicated process. It requires a function to compute a new increment step, and a rehash function.

Our chosen rehash function is as follows:
d = hashInc(k) = [(k) mod (8)] + 1
reHash(j, d) = (j+d) mod 1013

Implementation

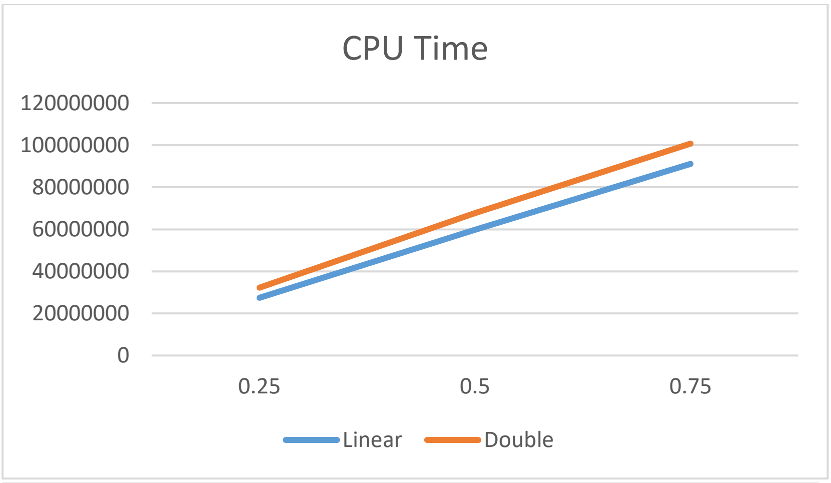
The code fragment below is our implementation for Linear Probing

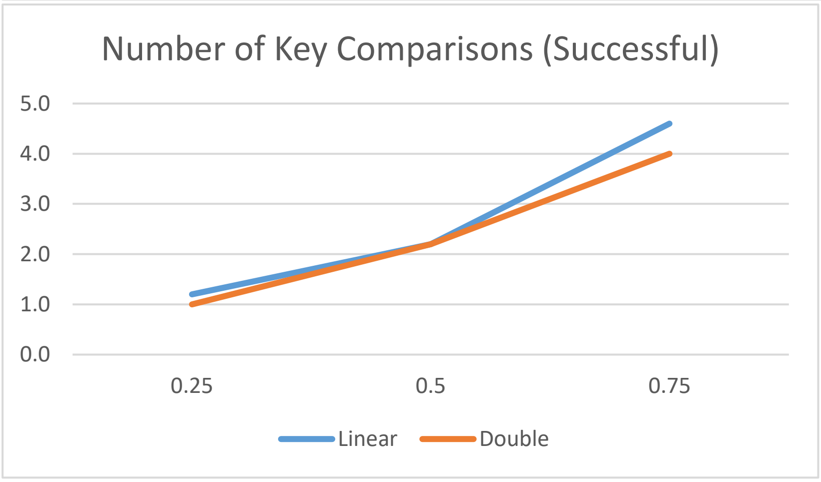
```
public static int[] linearHash(int[] key, int[] hashTable, int dataSize, int size) {
    for (int i = 1; i <= dataSize; i++) {
        int slot = key[i - 1] % size;
        int n;
        // keep increasing the hash index by 1
        for (n = slot; hashTable[n] != -1; n = (n + 1) % size) {

        }
        hashTable[n] = key[i - 1]; // insert the key into hash table

        System.out.print(key[i - 1] + "-" + n + "\t");

        // print 7 results in a row
        if (i % 7 == 0) System.out.println();
    }
    return hashTable;
}
```

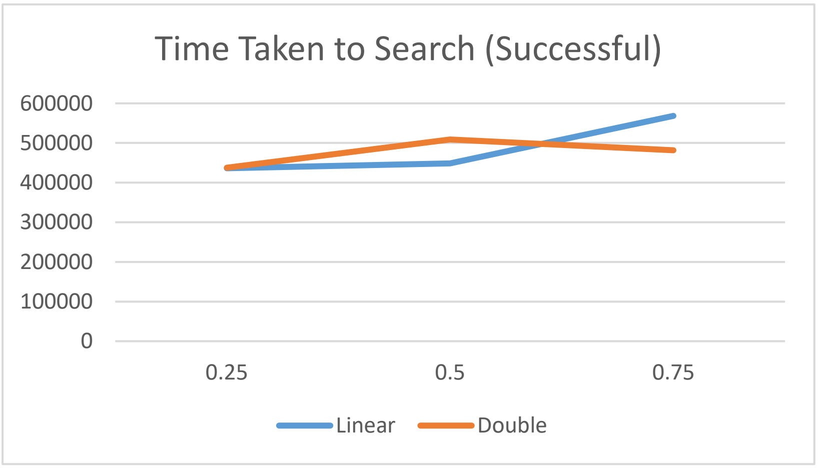
The code fragment below is our implementation for Double Hashing

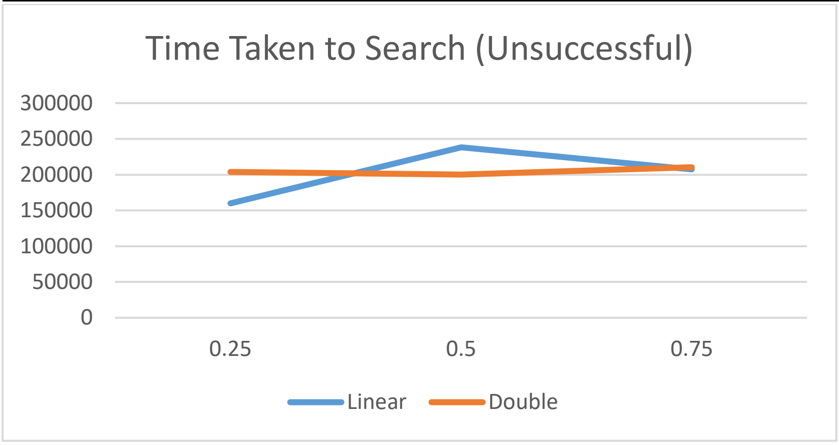
```
public static int[] doubleHash(int[] key, int[] hashTable, int dataSize, int size) {
    for (int i = 1; i <= dataSize; i++) {
        int slot = key[i - 1] % size;
        int d = (key[i - 1] % 8) + 1;
        int n;
        for (n = slot; hashTable[n] != -1; n = (n + d) % size) {
        }
        hashTable[n] = key[i - 1]; // insert the key into hash table

        System.out.print(key[i - 1] + "-" + n + "\t");

        // print 7 results in a row
        if (i % 7 == 0) System.out.println();
    }
    return hashTable;
}
```

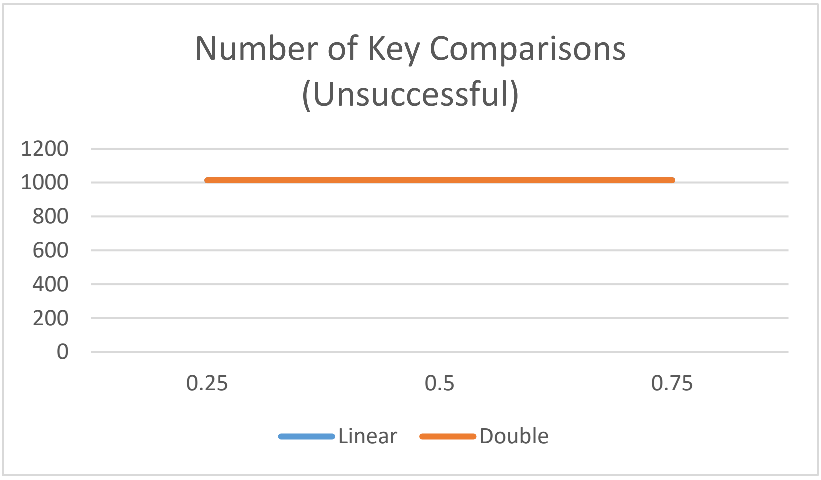
Statistics


CPU Time
120000000
100000000
80000000
60000000
40000000
20000000
0
0.25
0.5
0.75
Linear
Double


Number of Key Comparisons (Successful)
5.0
4.0
3.0
2.0
1.0
0.0
0.25
0.5
0.75
Linear
Double


Time Taken to Search (Successful)
600000
500000
400000
300000
200000
100000
0
0.25
0.5
0.75
Linear
Double





Explanation of findings

Time complexity

Linear probing and Double Hashing are both under Open Address Hashing

Assuming h = 2n, load factor is n/h

Best case: Keys evenly spreaded, average no. of probes for an unsuccessful search is 0.5

When load factor = 0.5, i.e. hash table is half-full, both methods have nearly equal efficiency

Worst case: Keys on one half of table, other half empty, average no. of probes for an unsuccessful search is

$$\frac{1}{2n}\left(\sum_{i=1}^{n} i+0\right)=\frac{1}{2n}\left(\frac{n(n+1)}{2}\right) \approx n/4 = \mathbf{O(n)}$$

All open addressing schemes have linear O(n) for worst case as load factor tends to 1.

However, since clusters are formed in a random and dynamic process, it is rather difficult to determine a consistent average case.

Conclusion

Double hashing performs better than Linear Probing on average, but as the table size increases, its performance also suffers.

The items stored under Double Hashing are more evenly distributed since the step size changes based on the secondary hash function.

On the contrary, items stored under Linear Probing tend to cluster together since the step size is always 1.

From our findings, we can see that double hashing is a much more effective way of storing hash values in the table as compared to linear probing as it has lesser collision rate and a lesser time complexity value.