Example Class 2 Report

CZ2001 Algorithms

Lab Group: SS1

Group 5

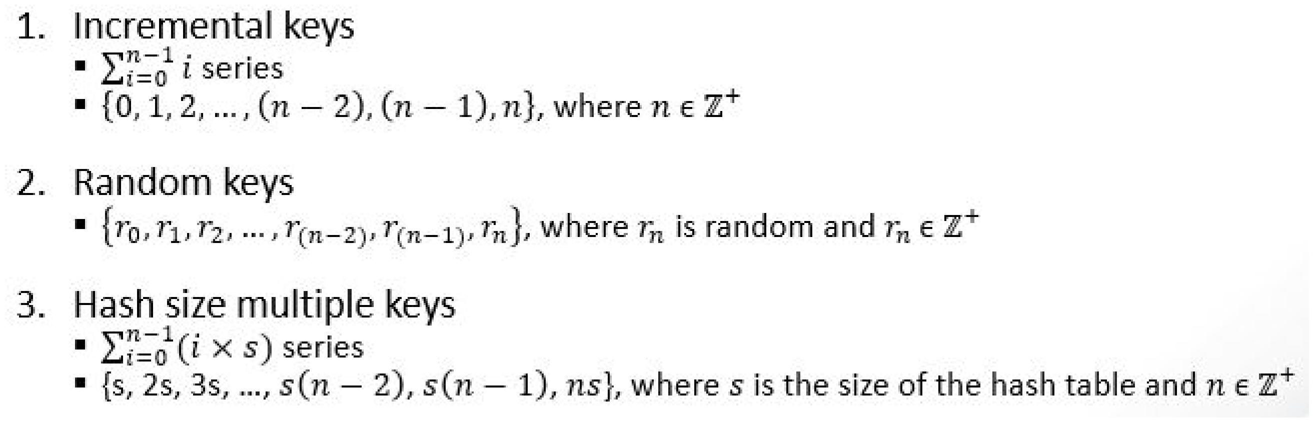
Chee Jun Yuan Glenn	U1621371J
Kevin Luvian	U1520135B
Lim Jun Ji Gerald	U1620748K
Ng Jun Hao	U1622597E
Tan Wee Seng	U1622264C
Wu Hao	U1622384A

Algorithm Lab 2 Report

Description of problem scenario and selected data set

In this scenario, we want to search through a list of employees for a certain employee and retrieve their particulars (name, …, etc.) given an id (employee's id).

For the selected data, we have generated 3 types of types of data used for searching:



1. Incremental keys
   - $\sum_{i=0}^{n-1} i$ series
   - $\{0, 1, 2, \ldots, (n-2), (n-1), n\}$, where $n \in \mathbb{Z}^+$
2. Random keys
   - $\{r_0, r_1, r_2, \ldots, r_{(n-2)}, r_{(n-1)}, r_n\}$, where r_n is random and $r_n \in \mathbb{Z}^+$
3. Hash size multiple keys
   - $\sum_{i=0}^{n-1} (i \times s)$ series
   - {s, 2s, 3s, …, $s(n-2)$, $s(n-1)$, ns}, where s is the size of the hash table and $n \in \mathbb{Z}^+$

Search type 1 for Incremental keys simulates the best case scenario (data1)
Search type 2 for Random keys simulates the average case scenario (data2)
Search type 3 for Hash size multiple simulates the worse case scenario (data2)

For our tests, our HashTable size would be fixed at 100k and we will be consolidating the data for the successful searches as well as unsuccessful cases with load factors of 0.25, 0.5 and 0.75 for each search to generate the test cases.

Description of hashing algorithms

1. Linear Probing

A key is obtained via i mod s where i is the input (employee's id) and s is the size of the HashTable.

key = input mod size

If the item to be searched is inside HashTable[key], then it has been found. Else, if the item to be search is not in *HashTable[key]*, then the key will be rehashed to key = (key + 1) mod size, **until the item has been found or it returns to the original position**; *HashTable[key mod size] = HashTable[input mod size].*

2. Double Hashing

   Similar to Linear probing, A key is obtained via i mod s where i is the input (employee's id) and s is The size of the HashTable.

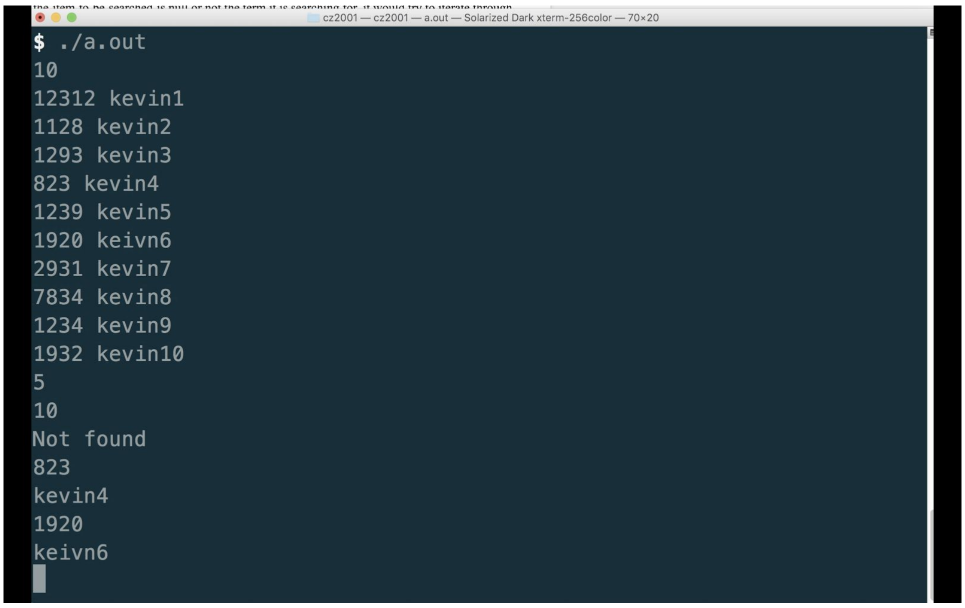
   key = input mod size

   However, before performing the search, the hash increment 'd' is computed as:

   $d = 1 + (floor((key/size)) \bmod (size - 1))$

   When the item to be searched is null or not the term it is searching for, it would try to rehash the key by changing key = (key + d) mod size until the item has been found or it returns to the original position.

Demonstration of implementation of hashing algorithms for both successful and unsuccessful cases:


cz2001 — cz2001 — a.out — Solarized Dark xterm-256color — 70×20

```
$ ./a.out
10
12312 kevin1
1128 kevin2
1293 kevin3
823 kevin4
1239 kevin5
1920 keivn6
2931 kevin7
7834 kevin8
1234 kevin9
1932 kevin10
5
10
Not found
823
kevin4
1920
keivn6
```

demo video: https://youtu.be/Klb2sCSB9Tc

Statistics on the average CPU time and number of key comparisons taken to search in data sets with load factors 0.25, 0.50 & 0.75.

Search 100K items with all items found with increasing key values

	Linear Probing		Double Hashing	
Load Factor	Average CPU Time/ns	Average Number of key comparison	Average CPU Time/ns	Average Number of key comparison
0.25	394	1	380	1
0.50	381	1	376	1
0.75	369	1	401	1

Search 100K items with no items found with increasing key values

	Linear Probing		Double Hashing	
Load Factor	Average CPU Time/ns	Average Number of key comparison	Average CPU Time/ns	Average Number of key comparison
0.25	8834	773	419	2.6
0.50	63553	6249	510	8.4
0.75	217735	21158	694	21.5

Search 100K items with all items found with random key values

	Linear Probing		Double Hashing	
Load Factor	Average CPU Time/ns	Average Number of key comparison	Average CPU Time/ns	Average Number of key comparison
0.25	452	1.03	382	1.03
0.50	412	1.16	406	1.15
0.75	403	1.66	450	1.47

Search 100K items with no items found with random key values

	Linear Probing		Double Hashing	
Load Factor	Average CPU Time/ns	Average Number of key comparison	Average CPU Time/ns	Average Number of key comparison
0.25	469	1.07	394	1.06
0.50	403	1.38	432	1.33
0.75	473	3.11	456	2.29

Search 100K items with all items found with key values multiple of hash table size

	Linear Probing		Double Hashing	
Load Factor	Average CPU Time/ns	Average Number of key comparison	Average CPU Time/ns	Average Number of key comparison
0.25	121116	12500	383	1
0.50	239939	25000	385	1
0.75	391787	37500	373	1

Search 100K items with no items found with key values multiple of hash table size

	Linear Probing		Double Hashing	
Load Factor	Average CPU Time/ns	Average Number of key comparison	Average CPU Time/ns	Average Number of key comparison
0.25	255691	25001	420	1
0.50	474094	50001	368	1
0.75	712009	75001	408	2.67

Explanation of results obtained and conclusion on time complexity comparison between chosen variants of hashing algorithms

We found that Double Hashing seems to outperforms Linear Probing for most cases from the results that we have obtained. This observation becomes more evident when the key value is an increasing sequence or it is a multiple of hash table size.

Although Linear Probing and Double hashing might seem to run at similar times at first when the key value is randomly created, Double Hashing still executes fewer key comparisons than Linear Probing on average; the average CPU time for Linear Probing increases substantially when the load factor approaches 1. This is primarily due to clustering, which means that there are long runs of occupied slots. This causes the rehashing function to be called many times.

Comparatively, Double Hashing has a relatively consistent performance for different values of load factor because there is a second hash function.

Time Complexity Comparison

For the worst case, both algorithms have a time complexity of $\in O(n)$ because the maximum number / worst case scenarios of comparisons is the hash table size itself. For example, for Linear Probing, it may search through the whole table to find a key which is just above the initial slot. For Double Hashing, the searching is done more randomly therefore reducing the chance of collision, but, the worst case is also searching through the whole table still. For Linear Probing, the worst case scenario is more likely to happen compared to the probability of the worst case scenario for Double Hashing, this shows that the second hash function is designed well.

For average cases, the time complexity for both algorithms is $\in O(1)$. Take Double Hashing as an instance, the probability for two key values to have the same hash function and rehash function based on our algorithm is extremely small. Thus it would take relatively constant time to search a key in the hash table for average case. However, even though they both have time complexity of $\in O(1)$, Linear Probing is worse than Double Hashing as its time complexity can become $\in O(n)$ easily.

Conclusion

Based on our observation of the average number of key comparisons and comparison of the time complexity, Double Hashing should be preferred compared to Linear Probing as it performs fewer number of key comparisons with the same load factor and it has a better performance than Linear Probing when the load factor approaches one.