LULEÅ UNIVERSITY OF TECHNOLOGY

Algorithms and Data Structures - Laboration 2

Course: D0012E

Marcus Lund (amuulo-4)

Edvin Åkerfeldt (edvker-4)

Samuel Karlsson (samkar-4)

December 7, 2015

First submission.

Contents

1 Introduction

The Lab is to implement two variations of linear probing. Linear probing is a way to store data by filling a array, slot by slot. It can be imagined as a parkinglot where you arrive with a car and start to search for a empty slot from any random position. You continue to search for a empty slot till you find one or discover the parking lot is full.

The first variant in the lab is a search in both directions from the initial spot. In the second variant you only search c (ware c is a content of users chose) steps in both directions, in comparison to the the first variant where you search the entire array (linear). If no slot is available if an interval, the array is re-hashed and we try again with the new array.

1.1 Language

This lab we choose tried to implement the algorithms in C++, but discovered that our tests were wrong and shwitched to Java since we have more experience with Java compared to C++. In the previous lab we used Python and during the lab we discovered that Python is really bad optimised language. Therefore we decided to switch language, any other than Python. Another contributing factor to switch language between labs is to get a wider knowledge of diffrent languages. Also to reftesh old skills.

1.2 Implementation

The implementation of the two variants is a bit tricky because there is a lot of small details to keep track of. For example, a variable that count the number of an element in the array or a variable for the array size. Both are easy to kip track of but it needs to be done.

Our first implementation takes an element and hashes it. The program then tries to place the element in an array at the position of the hash value. If the spot is not empty we look for the closest spot to that location.If we find an enmpty slot, we place the element at that slot. Otherwise the array is full.

The secound implementation is simlair to the first one. It hashes the element and tries to place the element in an array at the hashed value. Then if the slot is taken, we check up and down to hopefully find the closest slot that´s empty. Once we know the distance between the original hashed value and the free slot, we check if the distance is grater than a predetermined constatnt. If there is a free space, than place the element at that psoition. If there is no available slot within the range of x and $x + c$, then find an element in the interval x and $x + c$ with the condition that y (the possible position) has an empty slot in the intervall y and $y + c$. If sutch an element exists, move the element at y to the empty slot in the intervall y and $y + c$. Then place the new element at the y, witch is now empty. Otherwise, rehash the element and try again.

Table 3.1: Result from Implementation 1

Probes	Hashtable size	max_collision chain	insertion_collisions	runtime
10	10	5	6	0 ms
100	100	92	42	0 ms
1000	1000	797	530	15 ms
10 000	10 000	8744	5596	0 ms
10 000	100 000	6	2417	16 ms
10 000	1 000 000	3	2227	16 ms

2 Test procedure

The tests were conducted using different hashtable sizes: 10,100,1000,10000,100000 and 1000000. 10000 keys were randomly generated between 0 and 1000000000, we chose to stop at 10000 because we thought it wouldn't contribute anything to the end result to go higher. The keys were then hashed to find a home address to insert it into. Also the testing function stops trying to inserting new values if the hashingtable is full.

2.1 Implementation 1

When it comes to implementation 1 no other values then the hashtable size was to impact the end result, so therefore tests were only conducted with the different hashtable sizes.

2.2 Implementation 2

Although when it comes to implementation 2 a c value was thrown into the mix and therefore tests were conducted with that aswell. The c value was changed with a constant hashtable size.

3 Result

3.1 Implementation 1

Two things to take away from this result is firstly that runtime can be seen to be unconsistent and might not be completely reliable, secondly is that we can see a big drop in $max_collision_chain$ when the keys no longer fills the hashtable, this is because we no longer get so much overlap.

3.2 Implementation 2

First test were preformed with a constant c of 500.

Second test is preformed with varying c and constant hashtable size and probes of 10000.

In the last result table we can see that the c has big impact on both runningtime and insertion fails.

Table 3.2: My caption

Probes	Hashtable size	rehashes made	insertions failed	runtime
10	10	0	0	15 ms
100	100	0	0	0 ms
1000	1000	41	41	79 ms
10000	10 000	83	83	218 ms
10000	100 000	0	0	0 ms
10000	1 000 000	0	0	0 ms

Table 3.3: Result from implementation 2

C	insertions_faild	rehashes made	runtime
2	2571	2571	2407 ms
4	1514	1514	1813 ms
8	917	917	1281 ms
16	599	599	1000 ms
32	415	415	782 ms
64	243	243	531 ms
128	183	183	453 ms

4 Discussion

We had some problems with the code witch gave as a ridicules amount of collisions. We discovered that it was because of ours hashing method that gave a really bad variation of integers. Thiess result in a bad load factor in the second variation. When we fixed the hashing method the number of collisions went down to resemble values.

When the hash table size is big and the elements to insert small is the number of collisions really small. this was expected, to image it on a big parking with fuel cars is it easy to find an empty slot but on a small parking with many cars is it hard to find an empty slot to park in.

While the hash table has a low load factor is liner probing a good way to store data. When the load factor is hay is the many collisions and many insertion fails (second variation). The extra works collision cause is really bad for the runtime. The insertion fails are wary bad for the run time especially if you don't give up after one rehashing. We did only one rehashing before giving up on that element. Our rehashing has a low rat of fixing the insertion fail, if the hash table size increases wen a rehashing happens is the rehashing more likely to fix the insertion fail.