Ben Diekhoff

Hashing

03/24/2020

**Introduction**

Hashing is the process of taking a piece of data, known as a key, and applying a function to it that generates a hash value. There are many reasons to hash data. It can be used to authenticate a password without actually knowing the actual password. If the hash you receive from the user typing their password matches the hash value you have on hand, they must have entered the correct password. It can also be used to verify the integrity of data. If Bob sends Jim a message that hashes to 3422adg but the message Jim receives hashes to daf2342, the contents of the message were changed in transit. For this project, I focus on hashing as a way to achieve near O(1) lookup time. This is done by storing our key in a hash table, where the hash value is the index, or location of the key. So if I have a huge hash table, and I want to search for a given number, I don’t have to look through the entire table and check each index. I only have to hash the number I want to find and check that index in the hash table.

There are several different hashing functions, but in this project, I used a very simple one. It accepts two parameters, the key, and the size of the hash table. Each key is mapped to the index returned from key % tablesize. However, if my table size was 10, 1 and 11 would map to the same location.

A collision occurs when two keys map to the same location. To avoid collisions, I implemented two collision resolution policies, or CRPs. The first was linear probing. In the even two keys map to the same slot, this function increments down the table one index at a time (modulo table size so I never go out of bounds of my hash table) until it finds an empty slot to insert the key. The second CRP is double hashing. When this function finds the original hash value slot isn’t empty, it increments down the table as well, but using a specified increment. In my case, it takes the last digit in the key and adds one. So keys ending in 2 increment 3 slots down the table, and keys ending in 3 increment 4 slots. This is helpful in reducing clustering.

The load factor is the percentage of the table that has data stored in it. A very low load factor decreases the likelihood of collisions, but means that memory isn’t being used as efficiently as possible. A very high load factor means that little memory is being wasted, but the likelihood of collisions increases, meaning more operations for the CPU to process. Ideally, a load factor of 80% is the best of both worlds. In this project, I calculate the average number of probes for a has table with a 66% load factor and a hash table with an 80% load factor.

In order to test my hash table in the most objective way possible, I randomly generated 250 numbers. Duplicates were filtered out and replaced. With non-duplicates. In order to avoid inadvertently generating two sets of the same random numbers, I used time(NULL) as my seed. This uses the number of seconds that have passed since 1970.

**Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Avg Probes #1** | **Avg Probes #2** | **Avg of 2 runs** |
| Linear LF = 66 | 2.020 | 1.473 | 1.747 |
| Linear LF = 80 | 2.816 | 2.068 | 2.442 |
| Double LF = 66 | 1.620 | 1.507 | 1.564 |
| Double LF = 80 | 1.940 | 1.884 | 1.912 |

As expected, for the lower load factor tests always had fewer avg probes than the higher load factor. This is due to fewer collisions. Also of note is the fact that on average, double hashing had fewer avg probes than linear hashing. This is because double hashing ensures that data is more evenly spread out across the table, avoiding clustering. The fact that all averages are under 3 shows that I am very close to O(1) access times.

Out of curiosity, I ran the experiments again with a table size of 104,729 (The prime closest to 100,000 that I could easily find) and load factors of 66% and 80%. I allowed numbers to be duplicated due to the rand() functions relatively small max. My results were as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Description** | **Avg Probes #1** | **Avg Probes #2** | **Avg of 2 runs** |
| Linear LF = 66 | 18,191.602 | 18,195.107 | 18,193.355 |
| Linear LF = 80 | 25,527.192 | 25,512.418 | 25,519.805 |
| Double LF = 66 | 5,020.838 | 5,053.919 | 5,037.379 |
| Double LF = 80 | 7,137.924 | 7,191.045 | 7,164.485 |

The difference between the linear hashing averages and the double hashing averages goes to show how important a good CRP is. Clustering caused the linear function to perform far worse than double hashing. However, it would be a stretch to say that the double hashing policy was near O(1) access time. I believe this is due to my extremely simple hashing function (key % table size). A better hashing function would spread data more evenly through the table and further reduce clustering.

**Conclusion**

In conclusion, I have discussed what hashing is, and how it’s useful for many purposes, including security, verification, and quick access times. I discussed my own hash function, and showed how its simplicity was a burden when dealing with large amounts of data. I described my CRPs and explained how CRPs that avoid clustering perform better than those that don’t. I showed the pros and cons of smaller and larger load factors for hash tables in terms of memory waste and access times. It seems that the single most important factor in reducing the amount of probes in a hash search is to reduce the load factor of the table. However, considering memory constraints, the most important factor is clustering. The more evenly data is spread across the table, the better access times will be.