Ben Diekhoff

Hashing

03/24/2020

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, the values I inserted were randomly generated values from 0 to 4999