|  |
| --- |
| 5243 Adv. Algorithm Analysis, Spring 2020 |
| Hashing Experiment |
| Prof: Dr. Halverson |

|  |
| --- |
| Corbin Matamoros  3-26-2020 |

1. Introduction
   1. **Hashing** – Hash tables are data structures capable of constant time data access. Easily implemented and treated as an array / list, a hash table supplies programmers with the basic operations of insertion, deletion, and searching in, on average, near-constant time. The ideal hash table is of prime-number size (where that value is reasonably far from a power of 2), maps all data values to unique spots (i.e. no collisions), offers constant time insertions, deletions, and searches, and lastly is no more than 70% full [1]. Now, a hash table can be broken down into these parts: the hash function, and the collision resolution policy. Data is mapped to a specific spot / location in the table with a hash function, a mathematical function of the data itself. For example, hash\_spot = data\_value (mod table\_size) would be a very simplistic hash function for integer data values. Now, to be a bit more specific, the term “data\_value” will be replaced with “key”. The perfect hash function will map every key to a unique spot in the hash table; however, this is nearly impossible. Some keys will map to the same location, and a hash table’s collision resolution policy (CRP) establishes collision handling, either by chaining or open addressing. If chaining is used, the linked list must be doubly linked to maintain constant time data access. If open addressing is used, the probe sequence – discussed later – must adequately mimic randomness to ensure a uniform distribution of keys in the table.
   2. **Program hash function –** In our program, we use the easy hash function, the division function

* is the key
* is the table size.

By mod-ing the key by the table size, we enforce uniform hashing, which means all spots in the table are equally likely to be hashed to with any given set of keys. This hash function is by no means adequate nor secure in more realistic situations; there are other more sophisticated methods such as universal hashing that are more durable.

* 1. **Program CRP and associate probing techniques** – Our program implements a hash table as a dynamically-allocated array in C++11 with a table size of 311, a prime number and reasonably far from 256, a power of 2. We pit two open addressing techniques to determine probe sequence against each other – linear probing and double hashing.
  2. Linear probing –linear probing has an increment of one. This means if a new key collides with an old key, the adjacent spot following the collision spot is checked. If no key resides in that spot, the new key is placed there. However, if that spot is also occupied, the following spots are checked one by one until an empty spot is found or the entire hash table has been checked / probed. The equation is:
* is the result of the equation from part B above
* is the increment
* is the table size

Linear probing, although simple to implement, promotes primary clustering. This occurs because placing keys in adjacent spots leads to long “lines” of data collecting in one section in the table. This leads to longer search times in the end, so more complicated probing techniques are preferred in real-world situations.

* 1. Double hashing – one of the best methods for open addressing. Instead of checking the next spot in the table, the key is placed into another hash function. The equation is
* where is the result from the first hash function in part B
* is the second hash function, which is:
* is the increment
* is the table size

Double hashing acts like a random spot selector, which is why it’s one of the best CRP methods. Now, the second hash function needs to be relatively prime to the table size for the whole table to be searched (since we want to promote uniform hashing).

* 1. **Program load factors** **–** we employ two load factors in our experiment, 0.66 and 0.8. In brief, this means we will stop inserting keys into the hash table once it is 66% and 80% full, respectively. Now, the point behind testing both is to compare the number of probes required on average per key as a function of the table size. For each key insert, we will count the number of probes – or number of spots checked if empty before placing the key in the spot. For the tables with load factors of 0.66, we can expect the average number of probes to be smaller than tables with load factors of 0.8, since those tables will be less full.
  2. **Program random number generation** – For our experiment, 1000 keys will be generated randomly from the set of all positive integers between zero and 10,000, and dumped into a file. The file will be checked for duplicates, all of which will be removed. Duplicates are unlikely, though, since the set range should be large enough to prevent duplicates. Data generation is performed in an auxiliary program not called in this experiment. This must be done separately.

1. Actual results
   1. **Results**
      1. Table 1 – Hash Experiment Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Description** | **Avg. Probes #1** | **Avg. Probes #2** | **Avg. of 2 runs** |
| **0.66 Load Factor** | Linear Probing | 1.834 | 1.464 | 1.649 |
| Double Hashing | 1.488 | 1.384 | 1.436 |
| **0.80 Load Factor** | Linear Probing | 3.917 | 2.764 | 3.341 |
| Double Hashing | 2.756 | 1.972 | 2.364 |

* + 1. Figure 1 – Hash Experiment Results
  1. **Discussion –** As expected, the number of probes per key increases as the load factor gets larger, which is intuitive. The double hashing probing technique works significantly better – fewer average probes – than linear probing because its probe sequence mimics randomness. In detail, since the second hash function is a function of the key (which was randomly generated), the probe sequence is also random. Although the number of average probes exceeds one in every test, hash table access time remains constant given the probes count is much smaller than our table size of 311. Probes counts were consistently smaller for run 2, and this could not be explained; the cause is likely program-related, as there were no changes between runs to the data file from which keys were drawn.

1. **Conclusion –** Hash tables allow data storage insertion, deletion, and searching in constant time. However, keys will map to the same location, so a hash table’s collision resolution policy (CRP) must be robust enough to maintain the constant time data accesses; depending on the context, one could choose chaining, which appends same-spot mapped data to form a linked list, or open addressing, which searches subsequent spots for openings. This experiment supports double hashing as the better open addressing probe sequence technique compared to linear probing. Its probe sequence closely resembles random spot picking, which keeps probe counts low with hash table load factors of 0.66 and 0.8. In the future, one could test the chaining CRP and compare its results to this experiment’s. One could also add deletion functionality to both CRPs to see which performs better when there are a large number of key deletes.
2. **Sources**

[1] User “Hello World” on Stackoverflow: <https://stackoverflow.com/a/31401836/11456744>

[2] Cormen, Thomas H. et al, “Introduction to Algorithms”, 3rd ed. 2009. Massachusetts Institute of Technology, Cambridge, Massachusetts. Print.