Lab 4: Hash Functions

A Option, Part 1

Nathan Robinson

1. Part A
   1. Tasks:
      1. Create a hash table in main memory and fill it 50% full. Use the linear probe technique
      2. Look up the first 30 entries placed in the table. Print the min, max, and avg number of probes required to locate the first 30 keys placed in the table
      3. Now search for the last 30 keys placed in the table. Print the min, max, and avg number of probes required to locate the last 30 keys placed in the table
      4. Print contents of hash table indicating open entries (see primary/secondary clustering effect)
      5. Calculate and print theoretical expected number of probes to locate a random item in the table.
      6. Explain your empirical results in light of the theoretical results
   2. Results:
      1. Primary clustering is expected when using the linear probe, and the poor hashing function compounds on this clustering. The result is one large primary cluster near the middle of the table. The average probe count is 15.5 for the first 30 keys and the average probe count is 48.5 for the final 30 keys, which are both much worse than the expected results of 1.50 probes per key. This confirms the use of a poor hashing function, as the theoretical results hold for the ideal or near ideal hash algorithm.
      2. This large discrepancy exhibits just how poorly the function performs at producing unique values. Inspection of the resulting hash table shows that most keys are hashing to the same value, which is explained in part E.
2. Part B:
   1. Tasks:
      1. Create a hash table in main memory and fill it 90% full. Use the linear probe technique
      2. Search for the first 30 entries placed in the table. Print the min, max, and avg number of probes required to locate the first 30 keys placed in the table
      3. Physically search for the last 30 keys in the table to calculate the statistics. Print the min, max, and avg number of probes required to locate the last 30 keys in the table
      4. Fill the table with the same keys in the same order as used when only filling the table 50% full
      5. Print the contents of the table clearly indicating open entries (see the primary clustering effect)
      6. Calculate and print the theoretical expected number of probes to locate a random item in the table
      7. Discuss your empirical results in light of the theoretical results.
   2. Results:
      1. Following from the previously discussed results, the hashing function shows how poorly it performs as the load level increases (as the percent full increases). Even good hashing functions are only efficient up to about 75% load, so a poor function at 87% load will result in major issue. The data appears in the hash table as a very large primary cluster, wrapping around to the beginning of the table as in this case the table size and mod value are the same. While some entries are not hashing to location 32, 63 searches max are still performed given 64 keys as the damage has already been done.
3. Part C:
   1. Tasks:
      1. Repeat A and B above but use the random probe for handling collisions as developed in class.
      2. When you print the contents of the table, the secondary clustering affect should be visible.
      3. Calculate and print the theoretical expected number of probes to locate a random item.
      4. Discuss your empirical results in light of the theoretical results.
   2. Results:
      1. While still hashing almost all keys to the same value, the keys are more spread out due to random probe generation. This is shown by secondary clustering in the hash table. The random probe is not beneficial because the keys are mostly hashing to the same value for the first 64 inputs, and one of the necessities for the random number generator is that the results are repeatable so that we may search the table efficiently every time.
      2. The same string of random offsets will be generated in the same order, unless the seed is updated, and add that same sequence to every hash address on every search. This means, just like the linear probe, any 2 keys that hash to the same value will search through the same locations in the same order before reaching an empty location. Locations accessed will just be random instead of sequential.
      3. Once more varied keys are introduced into the hash table, different hash values will be available, improving the average probes when combined with the random generator. At the 90% load level, this is exhibited as entries in the input file where the 13th character is not a space are encountered, which interrupt the search cycle and follow a different probe route. This will typically result in fewer average probes based on the expected output of random probe vs linear probe. The random probe only improves the results slightly more than the linear probe, because the “Burris” hash function is so inefficient.
4. Part D:
   1. Tasks:
      1. Present results in tabular format (typed naturally). Compare your results to the theoretical results.
      2. If there are differences, please explain why.
      3. Physically calculate the theoretical values for comparison to empirical results for both the linear and random probes
   2. Results:
      1. Figure 1
      2. As previously discussed, the table exhibits the fact that given a function outputting the same hash for multiple inputs, the random probe does not provide any benefit. Once introduced to unique values, we can see that the average probes do improve a bit over the average linear probe, but not much. Expected improvement should be about 55% better. The difference between the theoretical expected average probes and the actual average probes for a given load level again showcase that the hash is not optimal at all for the given data set and not anywhere close to providing unique hash values.
5. Part E:
   1. Tasks:
      1. Criticize the Burris hash function on a technical basis.
      2. Clearly and explicitly state why my hash function should fail!
      3. Based on your criticism, write a better hash function.
      4. Explain why your hash function should be better from both a theoretical and empirical standpoint.
      5. Implement the hash function and generate the results for parts A thru D presented in tabular format for comparison. Results for all parts of the lab should appear in a single table
      6. Formally evaluate the results as part of your lab (typed evaluation)
   2. Results:
      1. The given “Burris” hash function uses folding by adding the 1st, 2nd, to the 6th, and 7th characters and then multiples the result by 2^8, which shifts the result left by 8 bits. Then the 13th character is added to the result, which is an 8 bit ASCII character. This occupies the first 8 bits that are now empty after the left shift. Finally, the result is mod 128, which extracts the first 7 bits. This demonstrates that the hash function will yield the value of the 13th character or less. The keys are left-justified strings, so the 13th character is almost always a space, ASCII value 32. Thus, most input characters will hash to location 32.
      2. This function did not account for the type or characteristics of the expected keys. It could be improved using different value for the modulus, such as 127.
      3. The comparison for my hashing function, which uses the *square and extract N bits* method is below. The differences in my function’s performance using main memory versus secondary storage are included also.
      4. Figure 2
      5. Unsigned 64 bit mod type was used to preserve as much data as possible because intermediate tests show the resulting hash before extracting bits is usually around 59 - 60 bits in size using my method. This mod type is also used for the initial conversion from string to integer to reduce the need for coercion within operations. My hash improves on the supplied hash’s performance by taking the product of the first 8 characters of the key and last 8 characters of the key after they have been converted to 64 bit integers. Performing the square in this way is done to attempt to take the full range of possible values into account for a given input string. I shift the resulting number right 4 decimal places (divide by 10000) to dispose of the lower order digits which is where we *typically* find repetition, and then extract exactly as many bits as I can use, based on table size, for the hash value. In summary, this function considers the full range of possible input characteristics by squaring the two halves together, shifts the resulting decimal number right 4 places to discard repetitious lower order values, and then extracts the remaining bits necessary to produce a result for the given table size. While I did know the properties for the majority of the keys in advance, I chose this exact method to give a more flexible algorithm to allow for better performance with more varied data sets, i.e. words with an average length over 10 characters or even 16 digit numbers, addresses, or any data type.
6. A Option, Part 1
   1. Tasks:
      1. Do both the "C" and "B" options.
      2. **Compare the results. Main mem vs relative file**
   2. Results:
      1. The resulting minimum, maximum, and average probe counts are the same regardless of the storage type, clearly seen in Figures 1 and 2 and by examining the hash table printouts. This is also evident by simple inspection of the theoretical value equations in that they do not need any input other than load level to determine the expected probe count. While the average complexity remains the same no matter the storage medium, it is important to note that the actual *time* the function takes will differ! Main memory will take the shortest amount of actual measured time to complete any of these operations, completely obliterating secondary storage access times. Were that to be a measured output of the lab, we would see that it holds true here as well. Discounting storage access times, in the case of huge amounts of data we can then see that utilizing secondary storage is perfectly acceptable and will have no effect on the actual hashing function, the whole process will just be slower.

Comparison Table for Clustering: 50% Full



Comparison Table for Clustering: 90% Full

