Tyler Scott

CS124 Project 5

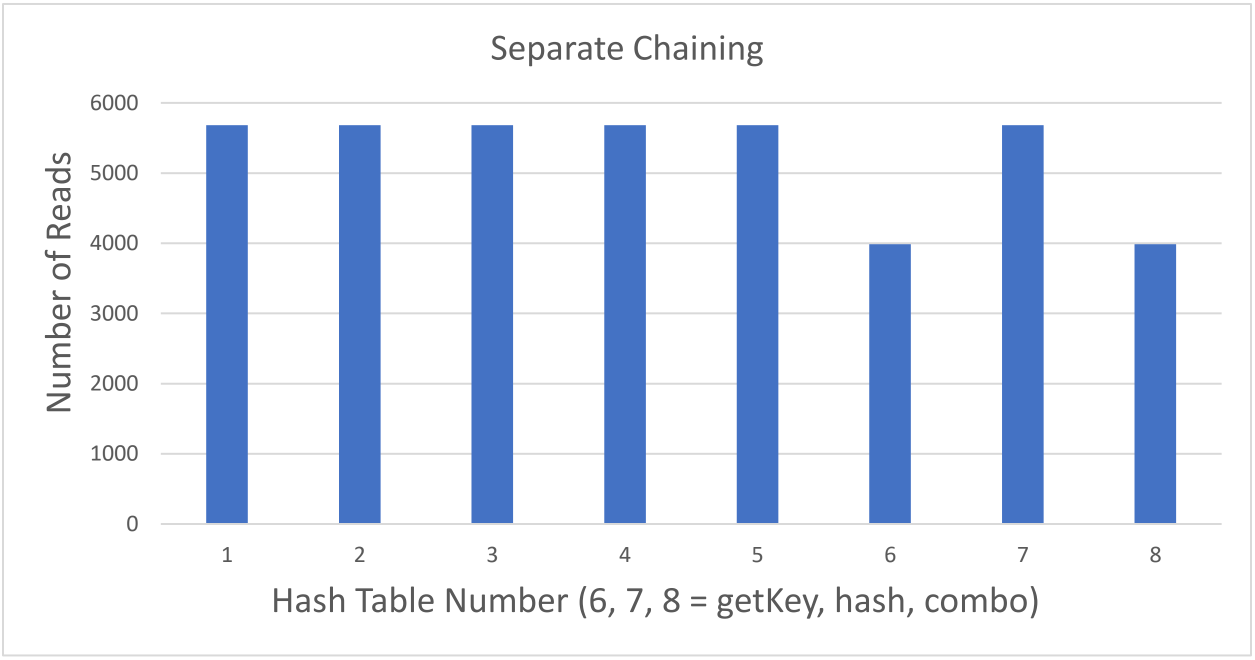
Professor Dion

The data set I have been using through the semester is the weather on Mars collected by the curiosity rover over a five-year period. I mainly use the Sol field in this project, which is the recorded day on Mars, though use the Pressure field (atmospheric pressure) in my alternative getKey() function. For the structure of the code I created one header file each for separate chaining and probing and included two classes in each; one using the hornerHash() and one for my alternative hash function. I decided to go with quadratic probing as my non-linear probing method. This was accomplished by changing the Linear Probing class and creating an unsigned long called *quad* that I initialize to zero. I then use that *quad* variable to iterate by 1 each loop and then multiply by itself (1 x 1 = 1, 2 x 2 = 4, 3 x 3 = 9, etc..) as opposed to using a power function. This is then added to the index and a remainder is given from modding by the table size.

To record the number of reads in the insertion method I created a global variable called *reads,* which was then set to 0. It then adds itself (the current total) plus 1 each time a hashable item is compared. Then I added a getReads() method at the end of each respective class to be able to call the total number of reads to be output to a file for further comparisons. For creating an alternative to the hornerHash() function I found a few possibilities online, but ended up settling on what is called the DJB2 algorithm. The logic was found in an article by York University in Toronto, which discussed several hashing algorithms, at www.cse.yorku.ca. Basically to accomplish this the initial hash value is set to 5381 and from what I have read this number is chosen due mainly to the time in which the algorithm was devised. At a time of 8-bit processors they wanted to use a prime number that would be larger than the possible maximum of unsigned integers the processor could handle. This may not apply quite as well with current 32 and 64 bit systems though. The next step is to use the *shift* operator in C++ (using “ << ” as we do with insertions such as with *cout).* In this context the operator shifts a physical bit over a set number of spaces, 5 spaces in my function, which is basically an equivalent of multiplying the initial value by 33 (1 = 00000001 to start and then shifted 5 spaces is 32 = 00100000; 32 + 1 = 33). The initial hash value as well as the ASCII value of the current letter is then added to that total and the entire new hash value is finally modded by the table size to get the remainder.

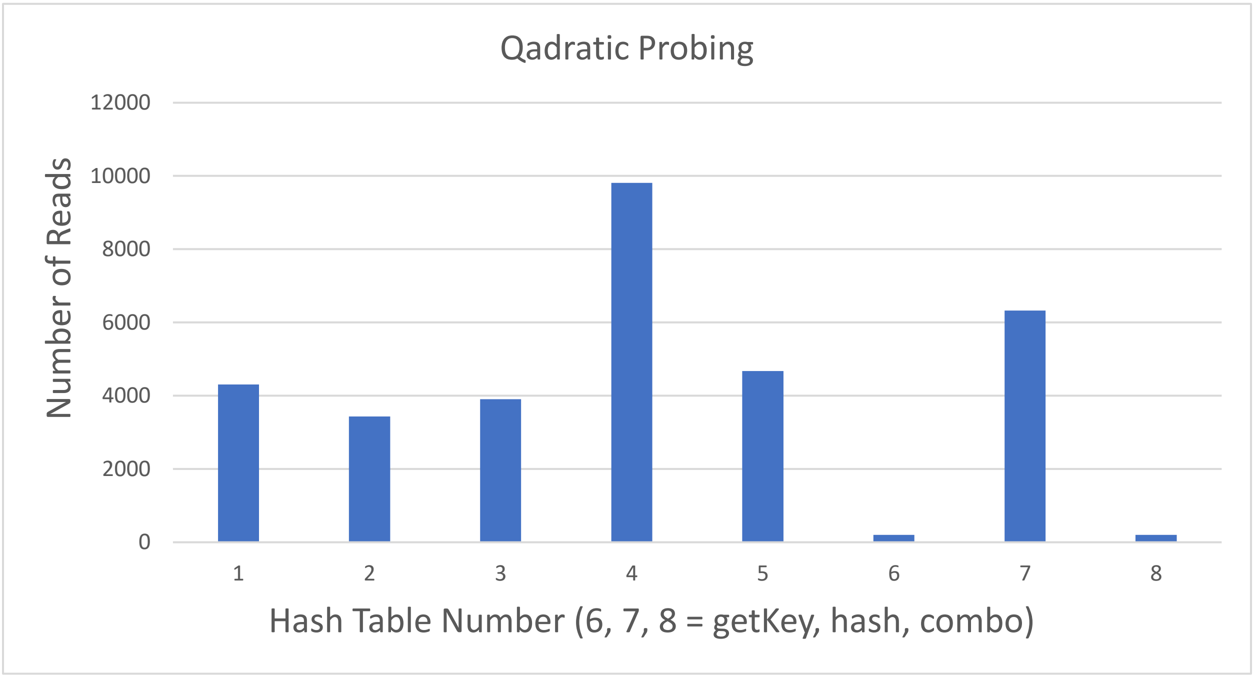
Separate Chaining

Separate chaining was certainly interesting to get started with as my number of reads actually all came out to be the same, given that the field I’m inserting with is the same. This does make sense in a way as when each element is inserted in the hash table, separate chaining would just create a linked list each time a collision occurs causing multiple items to be added to the same index as opposed to filling unused indexes regardless of how large the table is. So it is likely that each increment of the table size doesn't change the remainder when using modulus (%) in relation to the data set attributes because they all scale relative to one another. The only decrease in reads I saw was when changing the getKey() function to use the pressure attribute. Using this attribute, which only ranges roughly between about 700 and 1000, allowed for separate chaining to really do what it's best at, adding to linked lists when there are collisions. The are many collisions happening with the pressure field because there are so many repeated numbers contained within it, causing many of them to have the same key after hashing. By using this field for organizing the number of reads dropped from 5,682 down to 3,988. This can be seen in the chart below.



Quadratic Probing

Looking at the quadratic probing chart and it does appear the there is a small jump in potential efficiency over separate chaining. The least number of reads I had for my original getKey() was 3,430, the maximum was 9,813, with an average of 5,226 (rounded). The lowest read count was achieved with a table size of 2,250 against my roughly 1,800 elements from my data set, or just about 125% larger than the actual data size. It appears at first glance that even multiples may lead to a higher read count as the elements are dispersed in much wider variations on the fourth table (4^2), with both 3^2 and 5^2 producing reads at less than half the amount. Changing the hash function and using the same table size as table 3 resulted in almost double the amount of reads so is likely not as efficient as the hornerHash() method we have used. My number of reads dropped drastically to only 200 by changing the getKey() function to retrieve the pressure field instead of sol. This looks great at a first glance, but some concerns regarding this are discussed in the conclusion. The graph below shows the quadratic probing reads.



Conclusion

Quadratic probing with a table size roughly 125% of the data set size and organization by the pressure field was certainly the best approach to obtain the lowest number of reads possible. Though it forces me to wonder how this could negatively affect searching and removing as there are now dozens of elements all stored at the same hashed keys due to the pressure attribute having so many repeated values. This would undoubtedly cause some mistakes to be made the larger the data becomes. The number of reads would certainly be less for searching and removing, but the problem is whether the correct data would actually be returned once completed. With my data set being one that will not change, the only operations that are likely to be performed during use is searching. Due to that I would likely choose to go by the original organization of the Sol (current day), as opposed to the pressure, because it provides a unique ID to each element. That would make searching for items more efficient once it gets to the proper key because there shouldn't be multiple values at different indexes caused by the quadratic probing pushing items to an open index when a collision happens. By having unique values at each index we can know we are accessing the correct field and not 1 of 20 that all share the same value at different keys that the pressure field would give. It also (circumstantially) gives a slightly smaller read count than separate chaining. Though I could certainly see an argument being made in favor of separate chaining if the organization is kept by the pressure field as it will allow for multiple pieces of data to be stored at the same index. I would be most curious to see how this holds up as we scale the data sets to larger and larger quantities.