Justin Jones  
COSC 3319.01 Spring 2017  
Lab 4  
'A' Option 1

Part B/C Responses:

1. Utilizing linear probe, we expect to see primary clustering. Combined with the poor results of the hashing function, the result is one large primary cluster. Contrasting with the expected results of 1.33 probes per key, for the first 30 keys the average probe count is 15 and for the final 30 keys the average probe count is 35. This confirms the use of a poor hashing function, as the theoretical results hold for the ideal, perfect hash algorithm. Inspection of the resulting hash table shows that mostly all keys are hashing to the same value, which I’ll explain in part E.
2. Following from the previously discussed results, the hashing function truly shows how poorly it performs as the load level increases. Even good hashing functions are only efficient up to about 75% load, so a poor function at 87% truly exhibits problems. The data appears in the hash table as a very large primary cluster, wrapping around to the beginning of the table. While we encounter some entries not hashing to location 32, in the end we still perform max 50 searches given 51 keys as the damage has already been done.
3. While still hashing almost all keys to the same value, the keys are more spread out due to random probe generation which is also evidenced by secondary clustering in the hash table. Because the keys are all mostly hashing to the same value for the first 51 inputs, the random probe doesn’t provide a benefit to us because 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. In other words, we will generate the same string of random offsets in the same order (unless we change the seed) and add that same sequence to every hash address each time we search. This means that just like 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 – the locations just won’t be n, n + 1, etc. Once more varied keys are introduced into the hash table we may get different hash values which will improve our average probes when combines with the random generator. Moving up to 87% load level, we see this exhibited as we encounter entries in the input file where the 13th character is not a space, thus interrupting the search cycle and following a different probe route which will typically result in fewer average probes based on the expected output of random probe vs linear probe. Since the hash function is still impressively bad, it only improves slightly over linear probe.



Figure 1

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 on the order expected (we should see ≈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.

1. The given hashing function implements folding by adding the combination of the 1st and 2nd characters to the 6th and 7th characters, then shifts the result left 8 bits by multiplying by 256 (2^8). Following this, the 13th character is added to the previous result. As an ASCII character is 8 bits, it replaces the first 8 bits that were just shifted left and now empty. The value is then mod 128, thus extracting the first 7 bits. *It can then be seen from inspection then that this hashing function will always yield ≤ value of the 13th character!* Because the keys are primarily short strings that are left-justified, the 13th character is almost always a space, ASCII value 32, so the **majority of the characters in the input file will hash to location 32**. The function could easily be improved simply by using different value for the modulus, such as 127. Were this a real programming environment, I would say it likely that the creator of this function did not take into account the type or characteristics of the expected keys.

Utilizing the *square and extract N bits* method, I have created a hashing algorithm with the results seen below for comparison. Included the (nonexistent) differences between main memory and secondary storage pertaining to the function’s performance:  


Figure 2

I utilized an unsigned 64 bit mod type within Ada 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, etc.

Option A comparison:

The resulting minimum, maximum, and average probe counts are the same no matter the storage medium, as evidenced by the statistics shown in Figures 1 and 2 and/or 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.