Explanation

## Franklin Leung - COSC 3319 - Lab 5 - Fall 2012

# C-A

Primary clustering is readily apparent, though it may be argued that the weakness of the hash function in combination with this data set is causing it to unnaturally cluster at the front of the table. Most values will hash to zero, resulting in most entries to grow from the top due to collisions.

If every entry will collide with the first entry, then the number of collisions should be the sum of 1 to N, or N(N+1) / 2.

For 30 keys, if they all collided, that would be 30\*31 / 2, which equals 465 collisions. One less, at 29\*30 / 2, equals 435. One less again, at 28\*29 / 2, equals 406, which searching for the first 30 keys resulted in. Overall, there were only 2 non-colliding probes.

For the last 30 keys, since the entries were placed in the table at spaces 124 - 154, but most of the hashes computed to zero. If they were all collisions, then the expected number is . My results show 3895 collisions, very close to the maximum possible number.

# C-B

The explanation for the first 30 keys remain the same as the hash function is identical, as is the data.

The explanation for the last 30 keys is conceptually the same, only the input of numbers is different as the last 30 keys in a 90% filled table are spaces 201 - 231. So the expected number is , which is close to the 6405 I obtained.

# C-C

The secondary clustering is also readily apparent - most are clusters sized around 4 items, compared to the huge blob of primary clustering.

Theoretically, I should be getting about 1.53 probes for a 60% full table and 2.58 probes for a 90% full table. For the first 30 keys, my average of two appears to be very close to it, but for the last 30, the probe count goes much higher. I believe it's operating similarly to a linear probe as the first value compared will always be at HA 0, but since the offset is uniquely non-repeating, the list of keys-to-be-compared is continually shrinking with no redundancies. This explains the lower average of probes compared to searching linearly.

# C-D

Linear probing for both the first and last 30 keys are close to expected values (considering how the default hash function works). In random probing, the average number of probes did decrease compared to linear probing but collisions didn't *necessarily* lower. It seems that depending on the load factor (and this specific hash function), it would be generally better but not always.

# C-E

The default hash function relies on the integer values of characters 13 through 16 in determining a hash address. The problem is that most of the values (226 if I'm counting correctly) have identical hash addresses as the characters of those 226 are identical (all spaces, integer value 32). Obviously, collisions are common and makes the hash function extremely inefficient.

However, assuming the inability to write a better hash function, you could minimize the inefficiency by sorting your data by usage with the most used/searched item at the lowest index. Not an ideal solution, but it is Less Bad™.

Old hash:

My hash:

I have created a hash function that depend on the first four characters of the string, as it is much more likely that those four characters will have both a different character and not be spaces. Additionally, I've multiplied each character by a different prime number - character placement within the string is important. "aba" will hash differently from "aab" and "baa".

Judging from the results on the table, it is overall much better, cutting collisions down by about 40% in the worst case (4747 to 2988) to 99.3% (406 to 3). The average number of probes also improved, especially in linear probing, but random probing does not see as great of an improvement. I credit that to the existing improvement of random probing over linear probing in general - a better hash function has less of an effect.