CSCI 220 - Data Structures and Algorithms:

Project 3: Hashtable with Linking Analysis

Patrick Maes

November 15, 2022

Part A

In order to analyze the hash table and functions using timing, I timed each of the operations separately, for each file and hash function. The average time to do any operation is the total time of all the operations performed, divided by the number of operations done.
The way I computed the timings was using a vector and putting each word from the input file in the vector. The size is the number of insertions and we can simply insert each into the hash table using a loop.

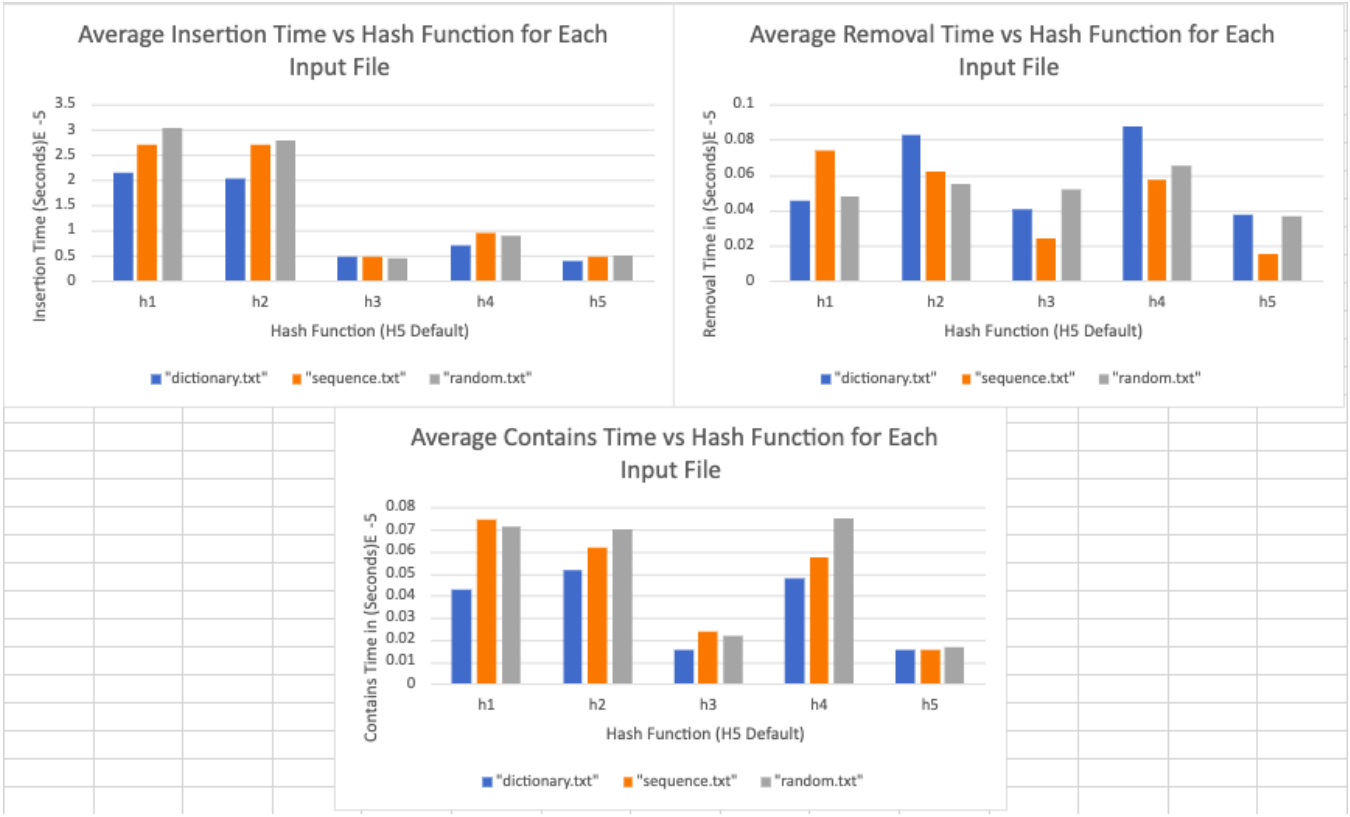

Figure 1: Plots of average time vs hash function for different input files.

Based on this data, the default c++ hash looks like it is the best in a lot of tests. The only time it was not the best was for testing insertions with the random.txt file against hash three. However, hash three performed comparably well in many other tests.

Otherwise, three of the hash functions have at least one file where they take a long time relative to the others. The default hash and hash 3 have less variability from file to file as well.
However, there does not look like there is a particular hash function that performs the worst overall. Each may be decent at one operation, but it seems most of the other hashes have a operation where they are relatively bad. You can easily see this by just looking at removal and insertion for 1,2 and 4, where they all appear to be weak hashes for at least one operation.

As far as explaining the timings of the bad hashes for the different input files, I noticed that the sequence file is only strings of length 4 and they are all unique. As for dictionary.txt and random.txt, the string lengths are variable and for dictionary.txt many of the words share a common prefix/suffix.
This could explain why the operations for hash1 took longer. Since hash1 only uses the first 4 characters as the hash value, there could have been many collisions where the string lengths were longer than 4 and the strings shared a prefix. In this case, we can not guarantee that the value we are hashing will be unique. When removing in this case, we might have to iterate over a longer chain list.

I would conclude the same for hash2. This hash adds all the characters in the string together to get the hash value. This is a bad hash since any two strings containing the same characters will hash to the same location. Order should matter when comparing strings, so a hash function should take into account the position of the character as well. If we have two strings and say one is the reverse of the other, they will have the same hash. This is bad especially for dictionary and random especially, not to mention two strings can have the same sum like "cccb" and "ccda". Even though all the characters are not the same, d+a = 197 and c + b = 197 where a=97, b = 98, c= 99, and d=100 (ASCII Characters).

Part B

In order to attempt to verify my results from part A, I wrote a function that stored the bin size frequencies of the hash table after all the elements were inserted. Then I wrote the data to a .csv file in order to input it to excel. My initial plots are bar charts because they are show the shape of the distribution well and are readable even with multiple input files.

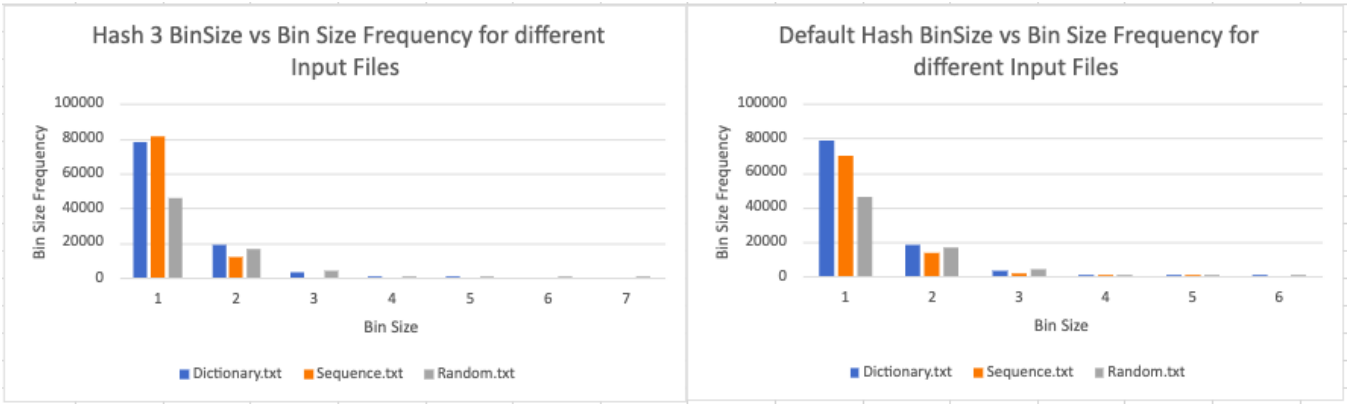

Figure 2: Bin sizes of java string hash (3) and default c++ hash. Examples of "good" hash functions.

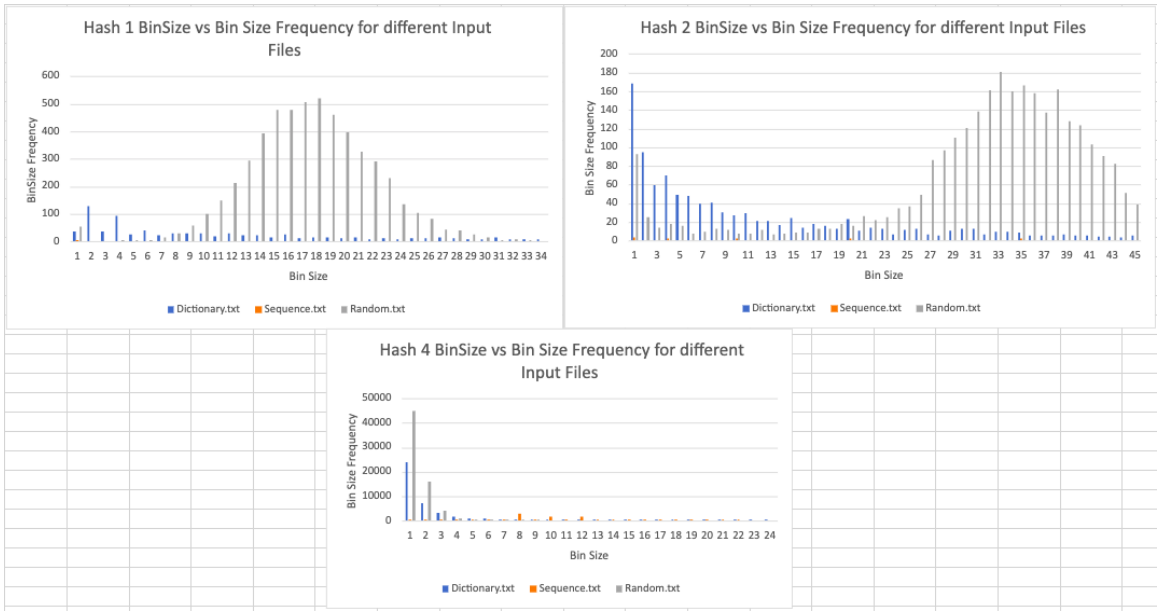

Figure 3: Bin sizes of hash 1, 2, and 4. Examples of worse hash functions.

The above graphs represent the outcomes of the bin sizes for different hash functions. The top graph clearly represents better hash functions, since most of the bin sizes are 1. The bottom shows the "bad" hash functions, where bin sizes might be more likely to be larger. Hash 4 is not a horrible hash function compared to the top two, but it still has a few bin sizes larger than 10-20 for some of the input files.

However, the plots of the bin sizes are not as clear for the bottom plots as they are for the "good" hash functions.

Hash 1 looks fine for Dictionary and Random in the graph, but we arent able to see data for sequence.txt. The bin sizes when testing this file were all 73 or 91, and there were 648 bins with each of these sizes. This is quite bad for this file in specific. This was better represented with a dot plot since there were only a few bin sizes, and they were spread far apart:

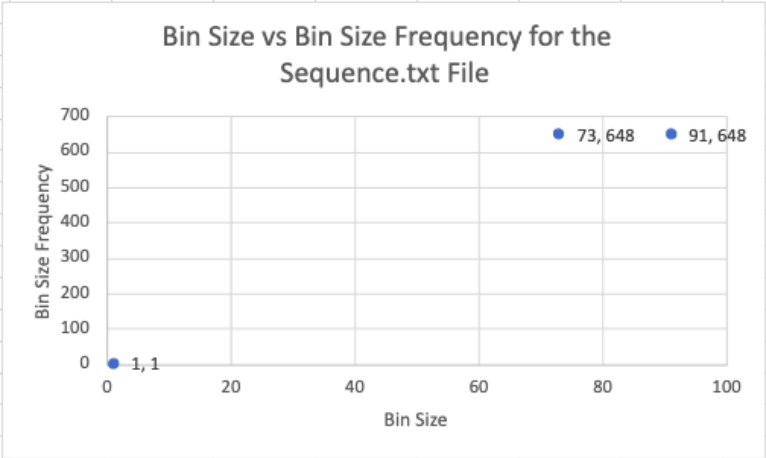

Figure 4: Bin sizes of hash 1 for the Sequence.txt file, nearly all elements are in a chain of size 73 or 91.

Hash 2 was also extremely bad with the Sequence.txt file. It had evenly distributed long chains, each with a frequency of around 2. I made a simmilar plot, but not all of the data could be displayed:

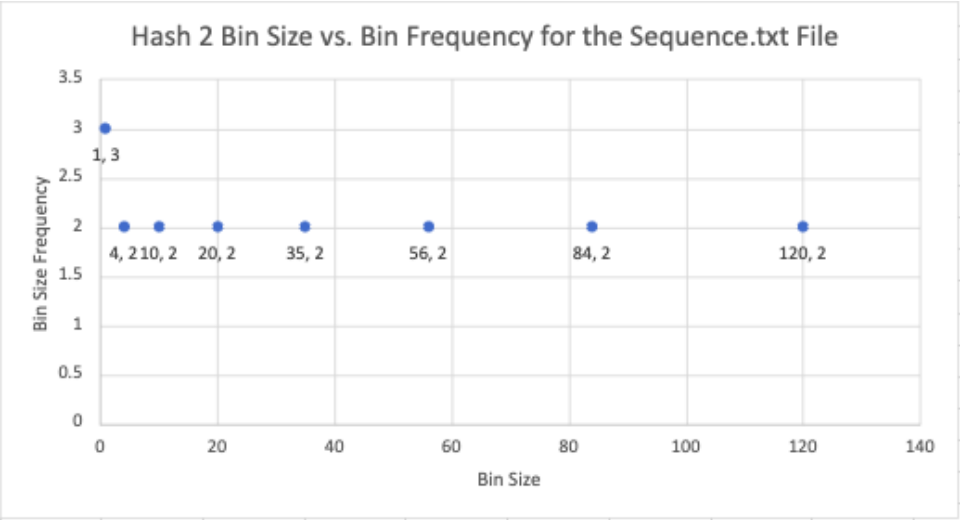

Figure 5: Bin sizes of hash 2 for the Sequence.txt file, nearly all bin sizes have frequency of 2 and are distributed widely.

This is not all of the data for this hash function and file. The bin sizes follow this pattern all the way past 3,000. That means that many of the elements are placed in chains that are longer than 100 or even 1000.

In the graph for hash 4, almost all the data is visible, but it is not easy to read. Hash 4 performed decent for the random.txt file, but had many more long chains for the dictionary.txt file especially. However, you can tell that it is not quite as bad as hash 1 and 2 for most of the input files. These results align with our timings, especially for the insertions. We can see that hash 4 is slightly worse than hash 3 and 5 in this plot. Hash 1 and 2 are consistently bad across all the timings.

Some of the results do not align exactly with the bin size results. We can see that hash 4 still took a long time for removal and contains. This could be due to run time discrepancies, such as CPU usage at runtime. However, we can see that hash 4 performed badly with the dictionary.txt file as we stated previously. Even though the timings between the hash functions do not match exactly, we can still identify the differences between individual hash functions, file to file.

I still agree with my hypothesis in part A as to why each hash function is bad or good, except for hash 4. I am not completely sure why a prime base makes the hashes more likely to be unique. I would guess that it is due to the number of factors the hashed value has. If we use 8 as the base, our final hash will likely have more factors than if we use a prime number. Then when we mod by the table size, we are probably more likely to get repeated remainders when there is common factors between the table size and hashed value.

Last, I made a data table that includes the mean, median, and max size of the buckets for each input file, per hash function. I computed the average as all the bucket sizes added together, divided by the total number of buckets. This is essentially the load factor of the hash table since the sum of all the buckets is the number of elements in the table, divided by the number of buckets which is the table size. This is why the mean is the same across each input file, since there are always the same number of elements inserted. Even in the best hash functions, I found that the mean bucket size was always zero, since more than half of the buckets were always empty.

Hash Function	Input File	Max Size	Average Size	Median Size
Hash 1	"dictionary.txt"	1611	0.485	0
Hash 2	"dictionary.txt"	599	0.485	0
Hash 3	"dictionary.txt"	5	0.485	0
Hash 4	"dictionary.txt"	1240	0.485	0
Default	"dictionary.txt"	6	0.485	0
Hash 1	"sequence.txt"	90	0.4	0
Hash 2	"sequence.txt"	3894	0.4	0
Hash 3	"sequence.txt"	2	0.4	0
Hash 4	"sequence.txt"	22	0.4	0
Default	"sequence.txt"	5	0.4	0
Hash 1	"random.txt"	33	0.728	0
Hash 2	"random.txt"	60	0.728	0
Hash 3	"random.txt"	7	0.728	0
Hash 4	"random.txt"	9	0.728	0
Default	"random.txt"	6	0.728	0

Figure 6: Mean, median, and max bucket sizes per input file, per hash file.

From the table, I would say that the best measure of hash strength would be the maximum bin size. However, it is also important to consider the number of buckets with size one and size zero from the bin size tests. The more buckets with size 1, the more constant time access the hash table has for different elements. Also, less buckets with size zero means that the hash function was able to distribute the elements across more buckets. While most of the buckets are of size zero in the table, our goal is to have the least number of zero sized buckets possible.

Part C

To find the worst case complexity of the naive string search algorithm, we need to know the complexity of comparing two strings and the number of comparisons needed. In my example, n is the size of the string we are searching in and m will be the size of the pattern.

The complexity of comparing the sub-string to the pattern will always be O(m) in the worst case, since each string will be of size m. We will have to do n+1-m comparisons in the worst case. I found this by using arbitrary pattern lengths. For the algorithm case, a pattern length of 2 needs 7 comparisons, length of 3 needs 6, all the way until m = n, where we only need one comparison. The cost of each comparison is always a constant O(m) operation, so we can multiply these costs. The worst case cost of the naive approach is O(m(n-m+1)) which is the same as O($mn - m^2 + 1$) which simplifies to O($mn - m^2$).

For the hashing algorithm, there would be a huge improvement if we can hash the string in constant time, then we would be able to compare the hashes in constant time as well. However if the hashes are equal, we still need to compare the pattern and sub-string to ensure that they are in fact equal. This is because we cannot guarantee the strings are equal from the hash, since two strings are able to share the same hash value.
In order to hash in constant time, the hash value cannot be dependant on the string length. Since the java string hash uses all the characters and their positions, we need a way to calculate the hash without all the characters. If j was the start of the string and the string length was 4, then the hash would roughly be defined as:

$$(((s[j] * 31 + s[j+1]) * 31 + s[j+2]) * 31 + s[j+3])$$

The biggest thing to notice is that for each extra character, we multiply by another factor of 31. This happens all the way down but excludes the last letter. That means that the first letter is multiplied by 31^{m-1}. If we want to calculate the next hash, we can subtract off the first value and then multiply by 31. Then we simply add on the next character in the string. The way I would express this would be:

$h_{j+1} = 31(h_j - 31^{m-1}s[j-1]) + s[j+m]$

The hash function above is not dependant on the entire sub-string being hashed, instead only the last character and the character immediately before the sub-string. I represented this as s[j-1] and multiplied this character by 31^{m-1} since it is no longer included and it was the highest order term. It is equivalent to the java hash since all the characters would be multiplied by an extra factor of 31, except for the last term added. While we do need to access character array elements, it is at most 2 for every hash. The java string needed m characters from the array, where n is the size of the pattern/sub-string.

To analyze the complexity of the hash algorithm, I attempted to look at the possible cases as the rolling hashes were calculated. Under the assumption of a good hash function, we still need n+1-m comparisons of hash values. Each of these hashes is computed in O(1) time, since the hash is not dependant on the string length. This means that when the string is not found, and no duplicate hashes are made, the complexity is O(n) since n is always greater than m. When the string is found in the worst case with the last comparison, we would likely have n-m O(1) comparisons and 1 O(m) comparison. Again, m is the length of the pattern and this assumes that strings that don't match hash to a different value.
However, if the hash was worse we could end up needing string comparisons for every sub-string. For example, the strings never match but we always get the same hash. This case would be exactly the same as the worst case for the naive string comparison, but would be unlikely with a good hash function. So for the average case, I will assume that non-matching strings will not hash to the same value. However, an unlucky case can have a time complexity of $O(mn - m^2)$

This means that if the string is found, we will need $\frac{n+1-m}{2}$ comparisons on average, with one being an O(m) comparison. So the complexity of the inccorect searches would be O(n-m) since n $\geq$ m. Then we would add on one O(m) comparison for the correct sub-string. The addition of these simplifies down to O(n). Therefore, the average case of the hash algorithm comparisons is O(n) with a good hash function, which makes sense if hashes are computed in O(1) time, and we need n comparisons through the string.