Implementation overview

I created three data structures for this project, they are described below.

LinkedList

A singly linked list that supports prepend, search, and remove. It can also print all elements inserted in the list.

HashTable

A hash table that resolves collisions by using separate chaining.

I expect at most 1000 insertions and want the load factor to be around 1.5 which gives a fast lookup time without wasting too much memory. $1000 \div 1.5 \approx 667$, so 673 (a prime) is a good size for the hash table.

To convert strings to integers, I used the method described in the assignment, which is $x = s[0] + s[1] * C + s[2] * C^2 \ldots$ The inputs are lower case English letters, so each character has 26 possibilities so 29 (a prime) is a good choice for C. Using this method, the location of characters matters a lot because they are multiplied by C to different powers. So words like "abc" and "acb" are mapped to considerably different values.

Each bucket of the hash table is a linked list object described earlier.

BloomFilter

A bloom filter that uses Vector<bool> to keep track of inserted elements; uses the hash table described above to keep track of deleted elements. Note that each boolean value in Vector<bool> only takes 1 bit (reference 1).

The hash function uses multiplication method: $x = m * \text{frac}(k A)$ where m is the size of bloom filter, k is the key, A is a real number between 0 and 1 (exclusive), and $\text{frac}(k A)$ returns the

fractional part of kA (for example frac(3.14)=0.14). I choose $A = 0.6180339887498949$ as recommended by Knuth (reference 2).

To generate a family of hash functions, simply multiply k by i. So each the ith hash function looks like $x = m * \text{frac}(ikA)$

Analysis of c and d

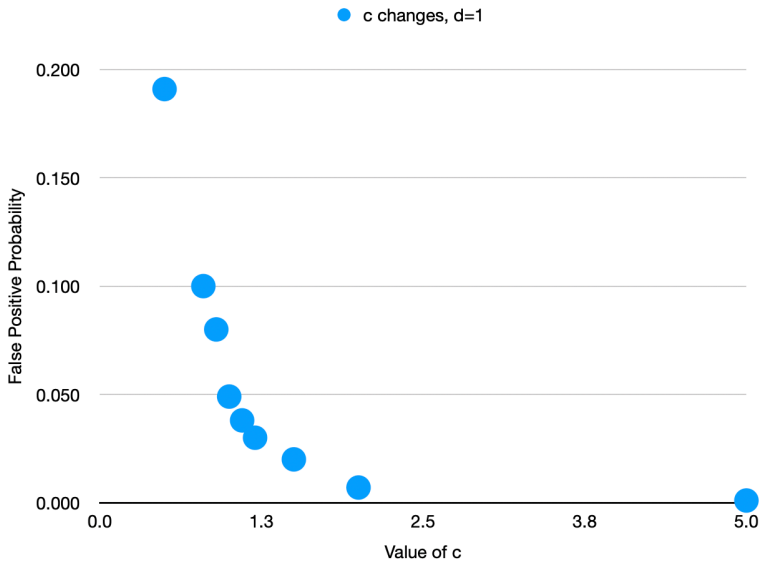
The directory named "Output" contains the output files for different values of c and d. A spread sheet named "False Positive Probability" summarizes the false positive probability for different choices of c and d. I included a screen shot of the spread sheet for your convenience.

False Positive Probability for Different Values of c and d

For all of the numbers below: p=0.1, and m=10000. When c=1, and d=1, the performance is 0.049; settings with better performance are highlighted in green.

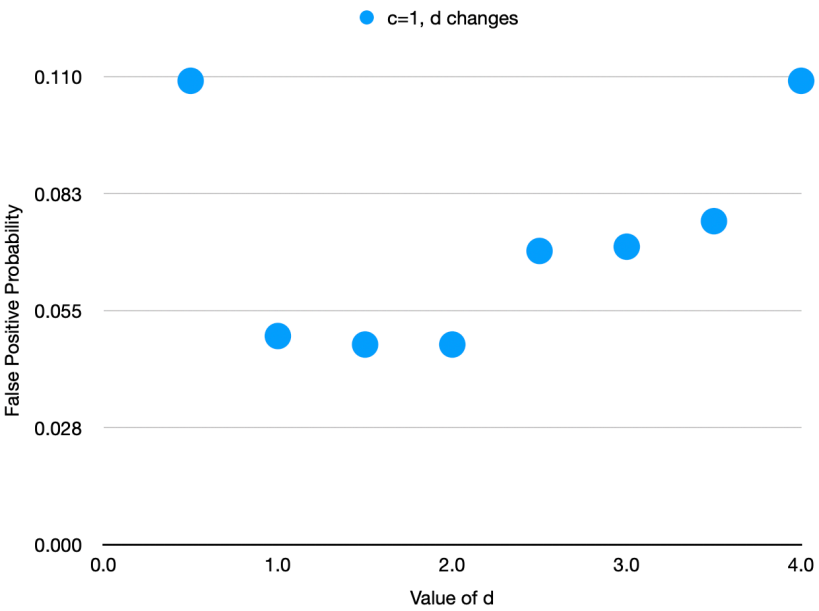
		Value of c								
		0.5	0.8	0.9	1.0	1.1	1.2	1.5	2.0	5.0
Value of d	0.5	1.000	0.143	0.111	0.109	0.103	0.095	0.042	0.011	0.002
	1.0	0.191	0.100	0.080	0.049	0.038	0.030	0.020	0.007	0.001
	1.5	0.173	0.074	0.056	0.047	0.037	0.034	0.021	0.006	0.001
	2.0	0.151	0.067	0.055	0.047	0.031	0.031	0.016	0.007	0.001
	2.5	0.180	0.077	0.065	0.069	0.047	0.038	0.026	0.011	0.001
	3.0	0.180	0.084	0.079	0.070	0.049	0.037	0.032	0.013	0.001
	3.5	0.201	0.125	0.102	0.076	0.067	0.053	0.045	0.017	0.003
	4.0	0.216	0.150	0.107	0.109	0.089	0.077	0.058	0.028	0.004

If we keep d the same and changes c, we get a plot like:



Ideally, we want c to be as high as possible to minimize false positive probability. Practically, a value like $c \approx 1.2$ significantly reduces false positive probability without using too much memory.

If we keep c the same and changes d, we get a plot like:



The false positive probability appears to be the smallest when *d* is between 1 and 2. So using too many hash functions doesn't reduces false positive probability and uses more computation.

In summary, higher *c* always reduces false positive probability whereas the choices of *d* should be between 1 and 2 to minimize false positive probability. This trend makes sense because higher *c* means more bits to record the elements, which makes collisions less likely. Having too little hash functions makes elements hard to distinguish from each other; having too many hash functions fills up the buckets too quickly; a proper choice of *d* creates just the right amount of hash functions to make each element unique and does not fill up the buckets too quickly.