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 supports insert, search, and remove; 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 $x = s[0] + s[1] * C + s[2] * C^2 \ldots$ where x is the resulting integer, $s[i]$ is the ith character, and C is a positive integer constant. The strings only contain lower case English characters, which means 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. To prevent overflow, I used unsigned int for x, which means $x = x \bmod (2^{32} - 1)$. Although this method causes more collisions in the auxiliary hash table (because it is not modding x with the size of hash table), it gives the widest range of integers for my hash functions (described in next section) to work with and thus reducing the probability of false positives.

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

BloomFilter

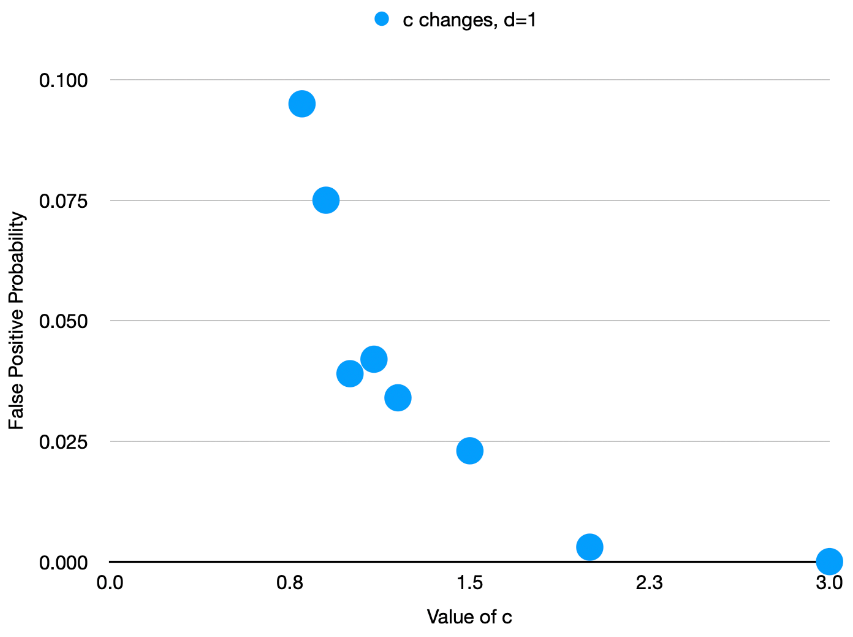
A bloom filter that supports insert, search, and remove; uses vector<bool> to record inserted elements; uses the auxiliary hash table described above to record deleted elements. Note that each boolean value in vector<bool> only takes 1 bit (reference 1).

The hash function uses multiplication method: $x = m * \text{frac}(kA)$ where m is the size of bloom filter, k is the key, A is a real number between 0 and 1 (exclusive), and $\text{frac}(kA)$ returns the fractional part of kA (for example frac(3.14)=0.14). I choose $A = 0.6180339887498949$ as recommended by Knuth (reference 2).

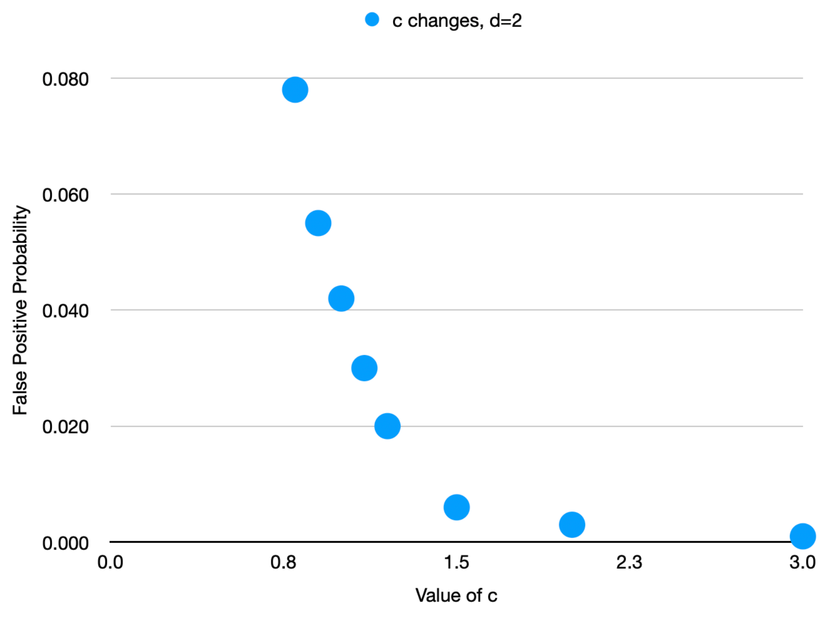
To generate a family of hash functions, simply multiply k by i. So the ith hash function looks like $x = m * \text{frac}(ikA)$. Since the key is multiplied by an irrational number, the fractional part of the result should be pretty random and therefore each hash function is independent and mostly uniformly distributed.

Analysis of c and d

Let's keep $d = 1$ and see how change of c affects false positive probability:



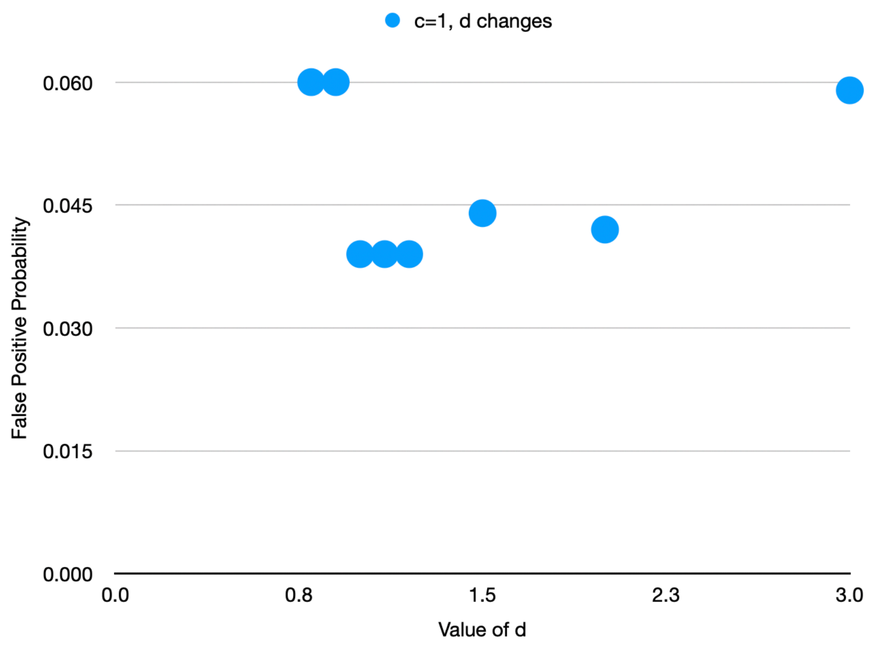
Here is another plot of changing c while keeping $d = 2$:



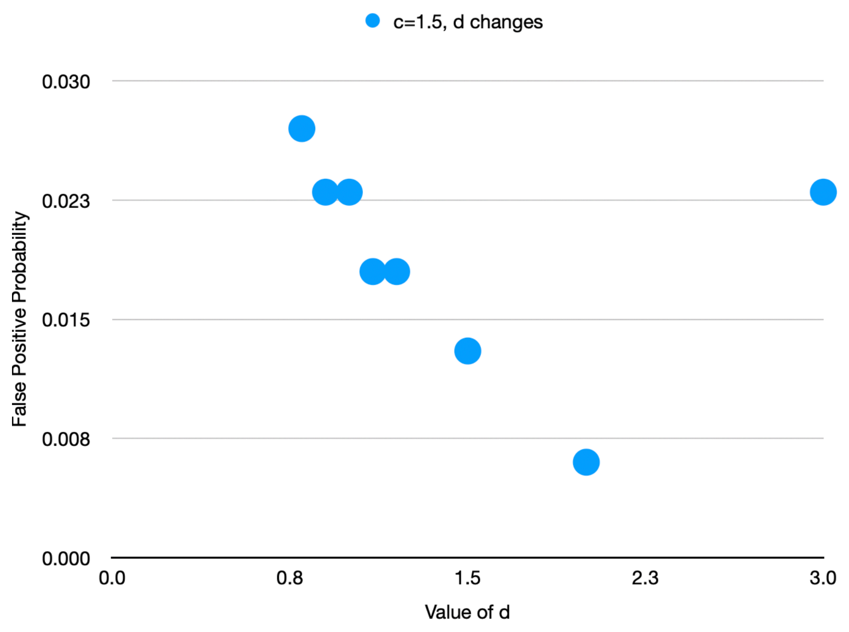
From the above two plots, it is pretty obvious that increase in c reduces false positive probability.

The ideal value of c should be as high as possible, but the cost of memory would be too high.

If we keep $c = 1$ and changes d, we get a plot like:



Here is another plot of changing d while keeping $c = 1.5$:



Unlike c, the ideal value of d is not as high as possible. It appears that the ideal value of d should be slightly higher than the value of c.

Here is a spread sheet that summarizes all of the false positive probability:

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 0.039; settings with better performance are highlighted in green.

		Value of c							
		0.8	0.9	1.0	1.1	1.2	1.5	2.0	3.0
Value of d	0.8	0.095	0.075	0.060	0.058	0.034	0.027	0.006	0.002
	0.9	0.095	0.075	0.060	0.042	0.034	0.023	0.006	0.001
	1.0	0.095	0.075	0.039	0.042	0.034	0.023	0.003	0.000
	1.1	0.095	0.051	0.039	0.035	0.028	0.018	0.005	0.000
	1.2	0.063	0.051	0.039	0.035	0.028	0.018	0.005	0.001
	1.5	0.063	0.051	0.044	0.037	0.021	0.013	0.004	0.000
	2.0	0.078	0.055	0.042	0.030	0.020	0.006	0.003	0.001
	3.0	0.109	0.075	0.059	0.046	0.035	0.023	0.008	0.001

In summary, higher c almost always reduces false positive probability (at a cost of using more memory) whereas d should be slightly higher than c (which means $1 < d < 2$ for reasonable setups of c) 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 (small d) makes elements hard to distinguish from each other; having too many hash functions (large d) 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.