Programming Assignment 1 – Sander VanWilligen

Specification List:

For each of the four bloom filter classes (FNV, Murmur, Ran, and Dynamic) I created the methods below (The functions are nearly identical unless otherwise noted):

Constructor (IE. BloomFilterFNV):

> The constructor took the set size and the bits per element, and used those to initialize a BitSet that acted as the filter, as well as initialize the filter size and the number of hash function. For the DynamicFilter class, the constructor only took an elements per element argument, and instead of a single instance of the number of hash functions and size, it stored arraylists of each.

add(String s):

> This method added the specified string to the bloom filter, provided it was not already detected. In the case of the dynamic filter, it also checked to see if it needed to create a new filter.

appears(String s):

> This method checked to see if the string s appeared in the filter. For the dynamic filter, it had to check each individual filter.

filterSize, dataSize, and NumHashes:

> These methods are pretty straitforward. They return the size of the filter, the number of elements added (not counting duplicates and false positives), and the number of Hash Functions.

hashFNV, hashMurmur, hashRan, and hashDynamic (String s):

> These methods are private, and are the individual functions used to hash the given string s. hashDynamic is the same as hashRan, but took an additional argument, I, which was the index of the filter used (So that it used the correct values for p, a, and b).

hash(int index, BigInteger h1, BigInteger h2):

> This is the method used to simulate k hash functions and bound the result by the size of the filter. It takes an index, which is the index of the hash function used. It also takes two hashed values. H1 is the initial hash of the string, and h2 is a hash of h1. IT multiplies the two hash values together, then multiplies them by the index, then does a modulo of the filter size to bound the data by that. In the case of the dynamic filter, it also takes a filter index to know which filter size to bound the hash by.

getPrime(int range):

this is a simple method to get the first prime greater than range. It is used by the Ran bloom filter and the Dynamic bloom filter.

For the FalsePositives class, I did not create any methods (since none were listed and I wanted to be able to do everything in the constructor in case that was the desired functionality). The constructor takes stringSize, numberOfStrings, and bitsPerElement as arguments.

For BloomJoin, I created the constructor and join method as specified. I noticed that the dynamic filter had a poor false positive rate (see reasoning below), so I used a murmur filter instead.

Hash Value Generation:

I found an article that mentioned it was possible to generate an infinite number of hash functions using only two. I get the second hash function by simply hashing the result of the first function again (which is also very efficient). Then, to generate an infinite number, I add the two hash values and multiply by the hash function number (1 … k), before bounding it by the filter size. This process is the same for FNV, Murmur, and Ran.

False Positives:

False Positives creates a list of numberOfStrings strings, each of size stringSize, with no duplicates. It then adds these elements to the four different types of filters. If an element is added but the size of the filter does not increment (because the strings are unique), that means there was a false positive. To get the false positive rate, simply take (numberOfStrings – dataSize) / numberOfStrings for each filter. All this information gets printed by the constructor.

Filter Comparison:

Sample FP values for 50000 strings of size 20 are listed below:

	4 bits	8 bits	16 bits
FNV	0.14046	0.07226	0.03474
Murmur	0.14032	0.09244	0.03626
Ran	0.17918	0.09830	0.04878
Expected	0.14586	0.02127	0.00045

From the data above, it appears that FNV and Murmur are better than Ran. In general they all seem to perform fairly well compared to the expected value, though as the number of bits increases they tend differ more. The time estimates are below for adding 50000 strings of size 20 at 8 bits with each filter.

	Runtime for add 50000	Runtime for search 50000
FNV	1002 milliseconds	463 milliseconds
Ran	189 milliseconds	100 milliseconds
Murmur	290 milliseconds	137 milliseconds

Ran Vs Dynamic:

I got the following false positive estimates for Ran and Dynamic with 500000 strings at length 20 each with no duplicates and 8 bits:

	False Positives
Ran	37111 (.074222)
Dynamic	396758 (.793516)

I'm not positive I did the DynamicFilter exactly right. For the filter, each time the number of elements goes over the bounds, I create a new filter of double the size, with a new set of hash functions for that filter based on the new size. The problem arises because the code has to check for an element in each of the filters. So, when you have several filters, and each one has a chance of having a false positive, the chance of getting one increases as the number of filters increases. I could not see any way around this, for the following reasons:

1. We cannot add the old elements to the new filter, since bloom filters do not allow retrieval, so we must keep track of all of the filters.
2. We cannot search a specific filter, since a given string could be in any filter. We must search every single filter.

The only way I could see of reducing the false positive rate was to limit the number of elements to less than 1000, or adjust the number of elements per element. I admit that I might not be seeing the proper way to implement this though.

As a final heads up, the code that generates unique strings for testing purposes in the FalsePositives class is not fast. Amounts of string greater than 50,000 will take a while to generate.