ITCS6114/8114
Homework 4
Hashing and Bloom Filters

Note: for all hash tables, open hashing+linked list is sufficient and recommended.

1.

Problem A: Write your own open hash table that includes functions insert and look up. Allow yourself to specify size of the hash table, *m*. You can use a data structure such as a linked list to store your pointers to your data associated with a given hash value in the table.

Problem B: Using a large range of integers generated from a random integer function, apply multiple divisive and multiplicative hash functions to your dataset. While choosing hash functions, consider their distribution over the size of the hash table.

Problem C: A hash table handles *n* values that are input and has *m* possible unique hashes. The load factor, α, is defined as $\alpha \equiv n/m$. Experiment with different values of α from $\alpha=0.1$ to $\alpha=10$. Study the collision rate, runtime, and total memory occupation.

2. A *bloom filter* is a tool that leverages hashing. If a user wants to check for the presence of an element in a data structure and the lookup can be potentially expensive, a bloom filter can help. A bloom filter can inform the user that an element either might be in the data structure or is definitely not in a data structure behind the bloom filter. If the bloom filter detects that an element is definitely not in the data structure behind the bloom filter, the user is spared from performing a potentially expensive lookup on the data structure.

A bloom filter works by leveraging an array of *m* bits and *k* different hash functions that map the *n* inserted values to the indices of these bits. A bloom filter's array of *m* bits are initially all zero. When inserting an element into the data structure behind the bloom filter, the $h_1, h_2, \ldots h_k$ hash functions are each applied to the element independently. The output of each hash function applied to a given element will correspond to an index in the bloom filter's bit array. The bit at that index is then flipped to 1. Because multiple hash functions are applied to a single value, a single element passing through the bloom filter can set up to k bits from 0 to 1 (this is the case where all the hashes of a given element are unique). If a bit has already been set to 1 from a previous element being inserted into the data structure behind the bloom filter, no action is taken.

The bloom filter becomes useful when we are interested in looking to see if a value is in the data structure behind the bloom filter. If we have a value we would like to look up, we can apply the bloom filter's hashes to it to see if the resulting bits in the bit array are set to 1. If any bit at an index corresponding to a hash of the element is 0, then the element has not passed through the bloom filter before, and therefore, is not in the data structure behind it.

For an interactive visualization of a bloom filter, I recommend this website. Note that this is hashing strings, not integers: http://billmill.org/bloomfilter-tutorial/ .

Problem A: Implement a simple bloom filter with a specifiable size of *m* bits*.* Choose *k* inexpensive hash functions for integers. While choosing hash functions, consider their output distribution over the size of the bit array, and also consider the uniqueness of their hashes when compared to other chosen hash functions.

Problem B: explore the relationship between *m, k*, and the *n* random integer values to pass through the bloom filter. First, choose a bit array size, *m. m* should not exceed 1.1MB in size in order to fit in cache. Then, choose your *k* inexpensive hash functions. For a given *m* and *k,* pass *n* integer values through the bloom filter where *n* goes from 0 to 32M values. As *n* elements are passed through the bloom filter, plot the percentage occupancy (number of bits which are 1/total number of bits in bloom filter) as a function of *n*. Repeat and vary combinations of the number of hash functions, *k,* and the size of the bloom filter, *m,* until you have an idea of how *m, k,* and *n* are related. Plot your results. Note that it is not necessary to plot every point as *n* varies for a given *k* and *m*; be sure to plot so that the features of the graph are adequately preserved.

3. Using knowledge gained from problems 1 and 2, store 4 million random integer values in an unsorted array by passing them through a bloom filter. Perform several thousand lookups on additional random integer values each while varying m_{bloom}'s size and the number of hash functions, *k.*