CSCI 104L Lecture 28: Bloom Filters

Question 1. Could we use a hashtable to implement a set?

Question 2. What kind of issues would arise with such an implemenation?

Bloom Filters are a way to avoid requiring the storage of keys. They come with a tradeoff: Bloom Filters are Monte Carlo Randomized Algorithms. That is, there is a small chance you will get the wrong answer.

When implementing a set, we generally have three functions:

- add
- remove
- contains

When using a Bloom Filter, we will not implement remove (it's challenging, and not worth it). If you need a remove function, choose a different data structure.

Your set merely needs to save which items are inside. You can add an item, and you can check whether an item is contained within it.

All we really need is an array of bits. If we want to store item k inside our set, then pass k into our hash function h(k), and set a[h(k)] = 1.

Question 3. What's the problem with this proposed implementation?

A false positive occurs when we say something is in the set, but it actually isn't. We will try to minimize the probability of this occuring.

Question 4. Is a false negative possible?

To reduce the probability of a false positive, we can use multiple hash functions $h_1, h_2, ..., h_j$. When we add an item k to the array, we set $h_1(k), h_2(k), ..., h_j(k)$ to true.

When we check whether an item is in the array, we make sure ALL of the bits $h_1(k), h_2(k), ..., h_j(k)$ are set.

Question 5. Can a false negative occur in this new implementation?

The probability of a false positive should have vastly decreased, because we have j independent tests as to whether the item is being stored in the set. The tradeoff is that we must increase the size of our bloom filter to compensate for the additional number of inserted bits.

If you want a false positive rate of 1%, you would want 7 hash functions, and 9.2 bits in the bloom filter for each item you plan to have inside.

If you want a false positive rate of .1%, you would want 10 hash functions, and 14 bits per key.

Most Bloom Filters will have between 2-20 hash functions.

Pre-filter Bloom Filters have become quite popular for applications that expect the majority of queries to return "no". You do a very cheap Bloom Filter operation to check whether the item is in the set or not. If the answer is "yes", we then verify it via a more expensive operation.

Suppose we have the following hash functions:

- $h_1 = (7x + 4)\%10$
- $h_2 = (2x + 1)\%10$
- $h_3 = (5x + 3)\%10$

Question 6. What will the Bloom Filter look like after inserting $0, 1, 2, 8$?

Question 7. What will the Bloom Filter say if we lookup $2, 3, 4, 9$?

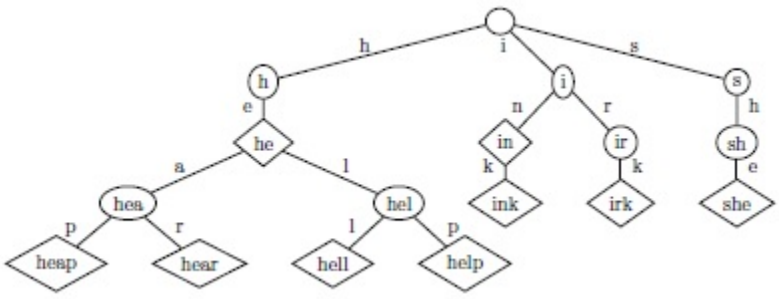
Question 8. Should we store our Bloom Filter as an array of bools?

We will use a concept called Masking to improve our space requirements.

Declare an array of integers. Each integer has 32 bits, so the first 32 bits will be stored in a[0], the second 32 bits will be stored in a[1], etc.

- To check if the jth bit is set, use the following formula: a[j/32] & (1 << (j % 32))
- (1 << j) will take the 1, and shift it left j bits. That is, it will insert j 0's to the right of the 1.
- x & j will do a bitwise AND between x and j.
- To set the jth bit, use the following formula: a[j/32] | (1 << (j % 32))

Tries



Suppose we have a binary search tree with n nodes, where each key is a string with k characters.

Question 9. For large k, would it be accurate to say that find takes $O(\log n)$ time? What would be a more accurate runtime analysis?

Trie, from re**trie**val are intended specifically for string keys which are very long.

This significantly cuts down on comparison on each level, as you're only looking at one character. Eventually you'll look at the whole string, but only once rather than $\log n$ times.

You can imagine there being a lot of wasted space in a Trie, such as when there are only two strings, each of 100 characters. You can improve on this by making a Compressed Trie.