CS240 Algorithm Design and Analysis

Lecture 22

Randomized algorithms (Cont.)

Quan Li
Fall 2023
2023.12.26

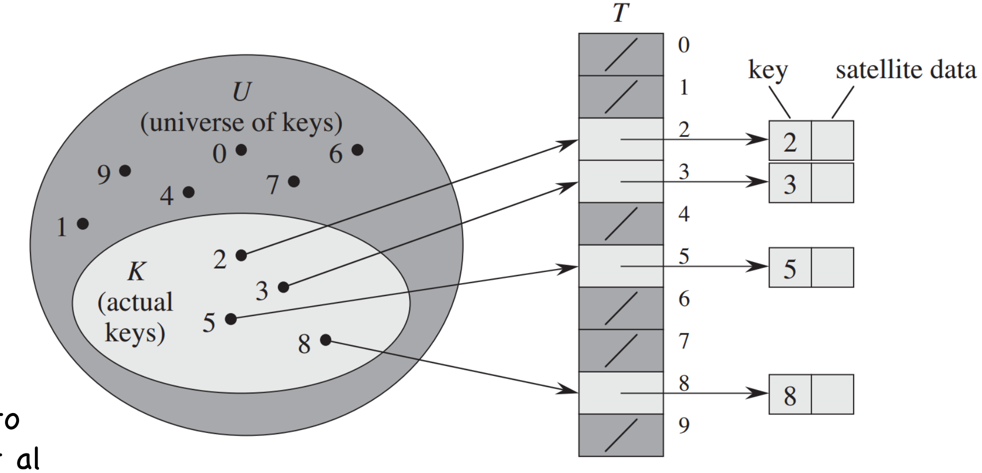
Hash Tables

Hash Tables

- A hash table is a randomized data structure to efficiently implement a dictionary.
- Supports find, insert, and delete operations all in expected O(1) time.
  - But in the worst case, all operations are O(n).
  - The worst case is provably very unlikely to occur.
- A hash table does not support efficient min / max or predecessor / successor functions.
  - All these take O(n) time on average.
- A practical, efficient alternative to binary search trees if only find, insert and delete needed.

Direct addressing

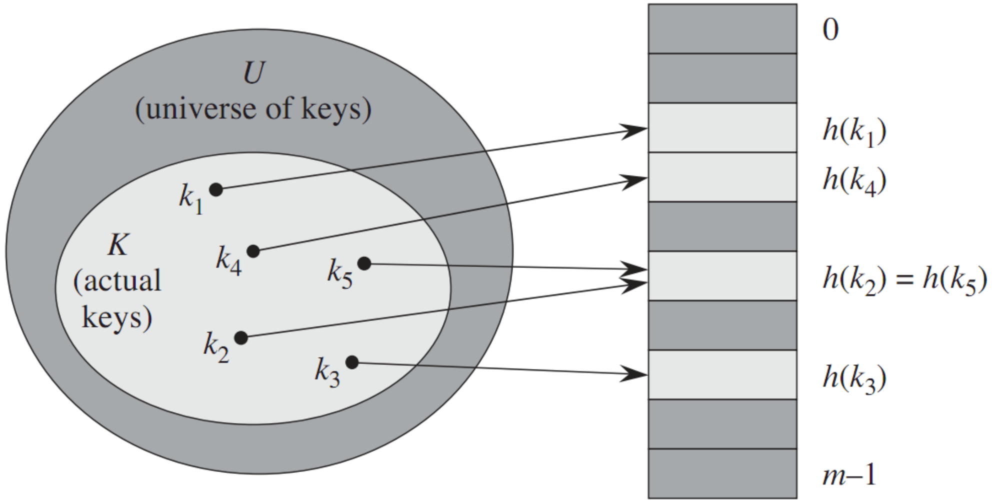
- Suppose we want to store (key, value) pairs, where keys come from a finite universe U = {0, 1, ..., m-1}.
- Use an array of size m.
  - insert(k, v) Store v in array position k.
  - find(k) Return the value in array position k.
  - delete(k) Clear the value in array position k.
- All operations take O(1) time.
- The problem is, if m is large, then we need to use a lot of memory.
  - Uses O(|U|) space.
  - Ex For 32 bit keys, need 4 GB memory. For 64 bit keys, more memory than in world.
- If only need to store few values, lots of space wasted.



Source: Introduction to Algorithms, Cormen et al

Hash Table

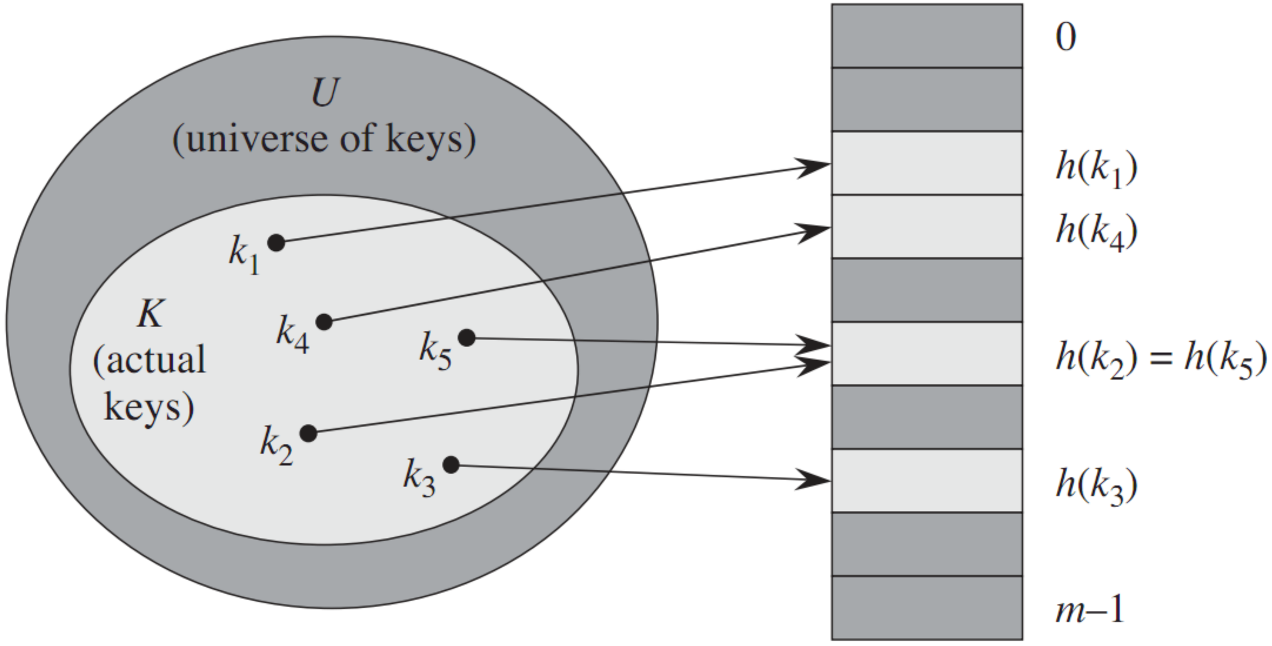
- Similar to direct addressing but uses much less space.
- Idea Instead of storing directly at key's location, convert key to much smaller value, and store at this location.
- A hash table consists of the following.
  - A universe U of keys.
  - An array of T of size m.
  - A hashing function h:U→{0,1,…,m-1}.



- We'll talk later about how to pick good hash functions.
- insert(k, v) Hash key to h(k). Store v in T[h(k)].
- find(k) Return the value in T[h(k)]
- delete(k) Delete the value in T[h(k)]
- Assuming h(k) takes O(1) time to compute, all ops still take O(1) time. Uses O(m) space.
- If $m \ll |U|$, then hashing uses much less space than direct addressing.
- However, our current scheme doesn't quite work, due to collisions.

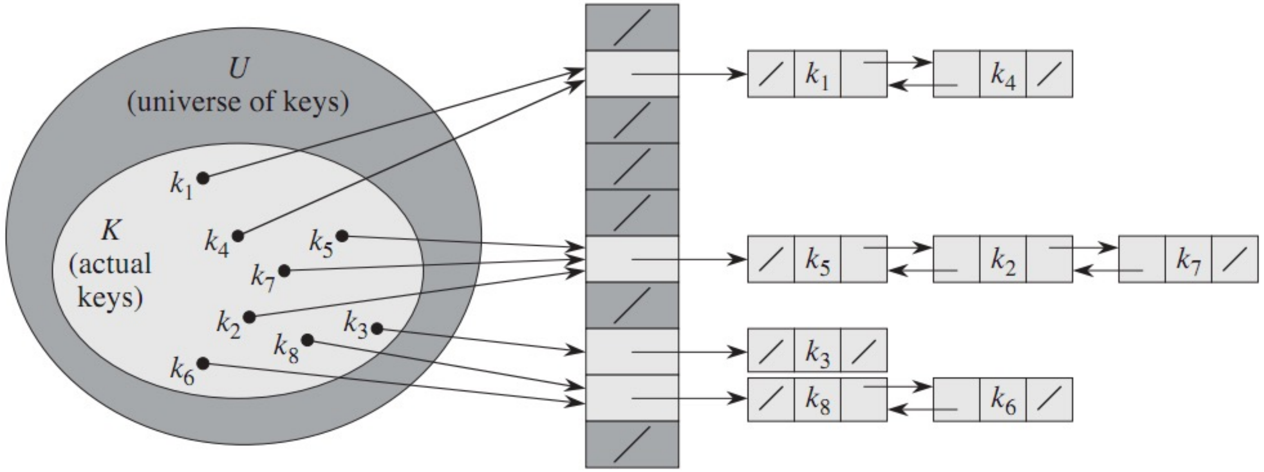
Collisions

- We store a key at array position h(k).
- But what if two keys hash to the same location, i.e., $k_1 \neq k_2$, but $h(k_1) = h(k_2)$?
  - This is called a collision.
- Collisions are unavoidable when $|U| > m$.
  - By Pigeonhole Principle, must exist at least two different keys in U that hash to same value.



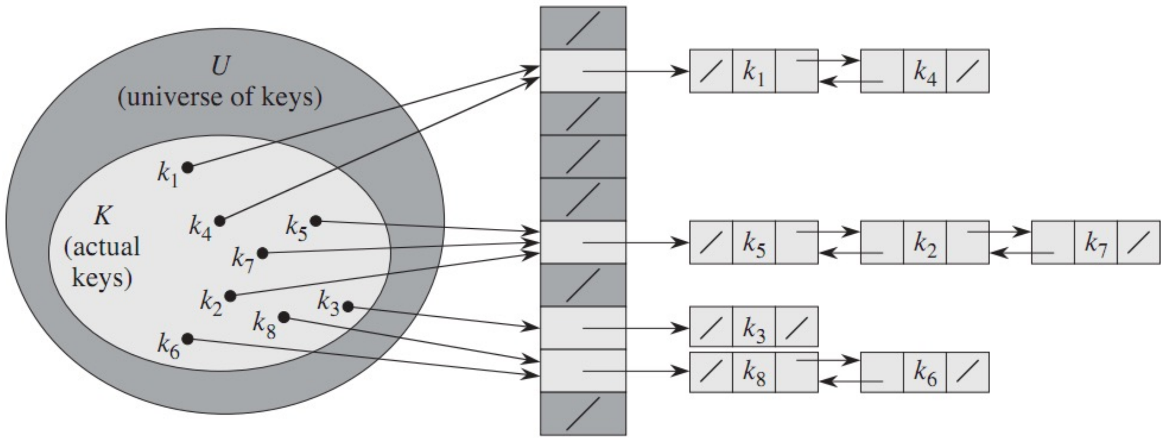
Closed Addressing

- In closed addressing, every entry in hash table points to a linked list.
  - Keys that hash to the same location get added to the linked list.
  - For simplicity, we'll ignore values from now on and only focus on keys.
- insert(k) Add k to the linked list in T[h(k)].
- find(k) Search the linked list in T[h(k)] for k.
- delete(k) Delete k from the linked list in T[h(k)].
- Suppose the longest list has length $\hat{n}$, and average length list is $\bar{n}$.
  - Each operation takes worst case $O(\hat{n})$ time.
  - An operation on a random key takes $O(\bar{n})$ time.



Load Factor

- The key to making closed addressing hashing fast is to make sure list lengths aren't too long.
- For this, we want the hash function to appear random.
  - Assume that any key is uniformly likely to be hashed to any table location.
- Suppose the hash table contains n items, and has size m.
- Then under the uniform hashing assumption, each table location has on average n/m keys.
  - Call $\alpha = n/m$ the load factor.
- So the average time for each operation is $O(\alpha)$.
- However, even with uniform hashing, in the worst case, all keys can hash to the same location. So, the worst-case performance is O(n).



Picking a hash function

- We saw that we want hash functions to hash keys to "random" locations.
  - However, note that each hash function is itself a deterministic function, i.e. h(k) always has the same value.
    - If h(k) can produce different values, we can't find key k in the hash table anymore.
- It's hard to find such random hash functions, since we don't assume anything about the distribution of input keys.
- In practice, we use a number of heuristic functions.

Heuristic hash functions

- Assume the keys are natural numbers.
  - Convert other data types to numbers.
  - Ex To convert ASCII string to natural number, treat the string as a radix 128 number. E.g. "pt" → (112*128)+116 = 14452.
- Division method h(k) = k mod m
  - Often choose m a prime number not too close to a power of 2.
- Multiplication method $h(k) = \lfloor m\,(k\,A \bmod 1)\rfloor$, where A is some constant.
  - Knuth's suggestion is $A = \frac{\sqrt{5}-1}{2} \approx 0.618034\ldots$

Universal hashing

- As we said, regardless of the hash function, an adversary can choose a set of n inputs to make all operations O(n) time.
- Universal hashing overcomes this using randomization.
  - No matter what the n input keys are, every operation takes O(n/m) time in expectation, for a size m hash table.
  - Note O(n/m) time is optimal.
- Instead of using a fixed hash function, universal hashing uses a random hash function, chosen from some set of functions H.
- Say H is a universal hash family if for any keys $x \neq y$

$$\Pr_{h \in H}[h(x) = h(y)] = 1/m$$

- So if we randomly choose a hash function from H and use it to hash any keys x, y, they have 1/m probability of colliding.
- Note the hash functions in H are not random. However, we choose which function to use from H randomly.

Universal hashing

- Thm Let H be a universal hash family. Let S be a set of n keys, and let $x \in S$. If $h \in H$ is chosen at random, then the expected number of $y \in S$ s.t. $h(x) = h(y)$ is n/m.
- Proof Say $S = \{x_1, \dots, x_n\}$.
  - Let X be a random variable equal to the number of $y \in S$ s.t. $h(x) = h(y)$.
  - Let $X_i = 1$ if $h(x_i) = h(x)$ and 0 otherwise.
  - $E[X_i] = \Pr_{h \in H}[h(x_i) = h(x)] \times 1 + \Pr_{h \in H}[h(x_i) \neq h(x)] \times 0 = 1/m.$
    - First equality follows by universal hashing property.
  - $E[X] = E[X_1] + \cdots + E[X_n] = n/m.$

Constructing universal hash family 1

上海科技大学 ShanghaiTech University

- Choose a prime number p such that p > m, and p > all keys.
- Let $h_{ab}(k) = \big((ak+b) \bmod p\big) \bmod m$.
- Let $H_{pm} = \{h_{ab} \mid a \in \{1,2,\dots,p-1\}, b \in \{0,1,\dots,p-1\}\}$.
- Thm H_{pm} is a universal hash family.
- Proof Let $x, y < p$ be two different keys. For a given h_{ab} let
$$r = (ax+b) \bmod p, \qquad s = (ay+b) \bmod p$$
- We have $r \neq s$, because $r - s \equiv a(x-y) \bmod p \neq 0$, since neither a nor $x-y$ divide p.
- Also, each pair (a,b) leads to a different pair (r,s), since
$$a = \big((r-s)(x-y)^{-1} \bmod p\big), \qquad b = (r - ax) \bmod p$$
  - Here, $(x-y)^{-1} \bmod p$ is the unique multiplicative inverse of $x-y$ in $\mathbb{Z}_p^*$.

Constructing universal hash family 2

- Since there are $p(p-1)$ pairs (a,b) and $p(p-1)$ pairs (r,s) with $r \neq s$, then a random (a,b) produces a random (r,s).
- The probability x and y collide equals the probability $r \equiv s \bmod m$.
- For fixed r, number of $s \neq r$ s.t. $r \equiv s \bmod m$ is $(p-1)/m$.
- So for each r and random $s \neq r$, probability that $r \equiv s \bmod m$ is $((p-1)/m))/(p-1) = 1/m$.
- So $\Pr_{h_{ab} \in H_{pm}}[h_{ab}(x) = h_{ab}(y)] = 1/m$ and H_{pm} is universal.

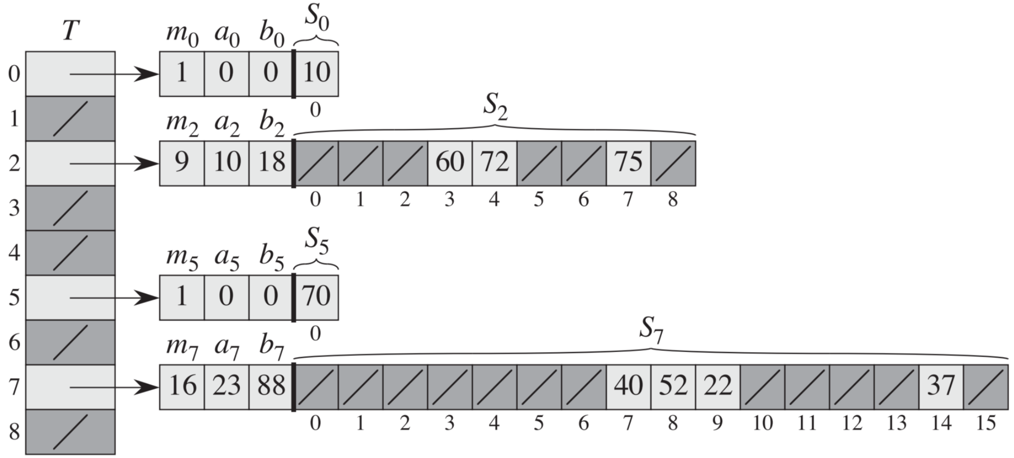
Perfect hashing

- The hashing methods we've seen can ensure $O(1)$ expected performance but are $O(n)$ in the worst case due to collisions.
- However, if we have a fixed set of keys, perfect hashing can ensure no collisions at all.
  - Perfect hashing maintains a static set and allows find(k) and delete(k) in $O(1)$ time.
  - It doesn't support insert(k).
- Ex The fixed set of keys may represent the file names on a non-writable DVD.

Perfect hashing

- Suppose we want to store n items with no collisions.
- Perfect hashing uses two levels of universal hashing.
  - The first layer hash table has size m = n.
  - Use first layer hash function h to hash key to a location in T.
  - Each location j in T points to a hash table S_j with hash function h_j.
  - If n_j keys hash to location j, the size of S_j is $m_j = n_j^2$.
- We'll ensure there are no collisions in the secondary hash tables $S_1, \dots, S_m$.
  - So all operations take worst case O(1) time.
- Overall the space use is $O(m + \sum_{j=1}^{m} n_j^2)$.
  - We'll show this is $O(n) = O(m)$.
  - So perfect hashing uses same amount of space as normal hashing.

- $h(k) = \big((3k + 42) \bmod 101\big) \bmod 9$
- $h_j(k) = \Big(\big(a_j k + b_j\big) \bmod 101\Big) \bmod m_j$



Avoiding collisions

- Lemma Suppose we store n keys in a hash table of size m = n^2 using universal hashing. Then with probability $\geq 1/2$ there are no collision.
- Proof There are $\binom{n}{2}$ pairs of keys that can collide.
  - Each collision occurs with probability 1/m = $1/n^2$, by universal hashing.
  - So the expected number of collisions is $\frac{\binom{n}{2}}{n^2} \leq \frac{1}{2}$.
  - By Markov's inequality the Pr[# collisions ≥ 1] $\leq$ E[# collisions] $\leq 1/2$.
- When building each hash table S_j, there's < 1/2 probability of having any collisions.
  - If collisions occur, pick another random hash function from the universal family and try again.
  - In expectation, we try twice before finding a hash function causing no collisions.

Space Complexity

- Lemma Suppose we store n keys in a hash table of size m=n. Then the secondary hash tables use space $E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$, where n_j is the number of keys hashing to location j.
- Proof $E\left[\sum_{j=0}^{m-1} n_j^2\right] = E[\sum_{j=0}^{m-1}(n_j + 2\binom{n_j}{2})] = E[\sum_{j=0}^{m-1} n_j] + 2\,E\left[\sum_{j=0}^{m-1}\binom{n_j}{2}\right]$
- $\sum_{j=0}^{m-1}\binom{n_j}{2}$ is the total number of pairs of hash keys which collide in the first level hash table.
  - By universal hashing, this equals $\binom{n}{2}\frac{1}{m} = \frac{n-1}{2}$.
- $E[\sum_{j=0}^{m-1} n_j] = n$.
- So $E\left[\sum_{j=0}^{m-1} n_j^2\right] = n + \frac{2(n-1)}{2} < 2n$.

Bloom Filters

Approximate Sets

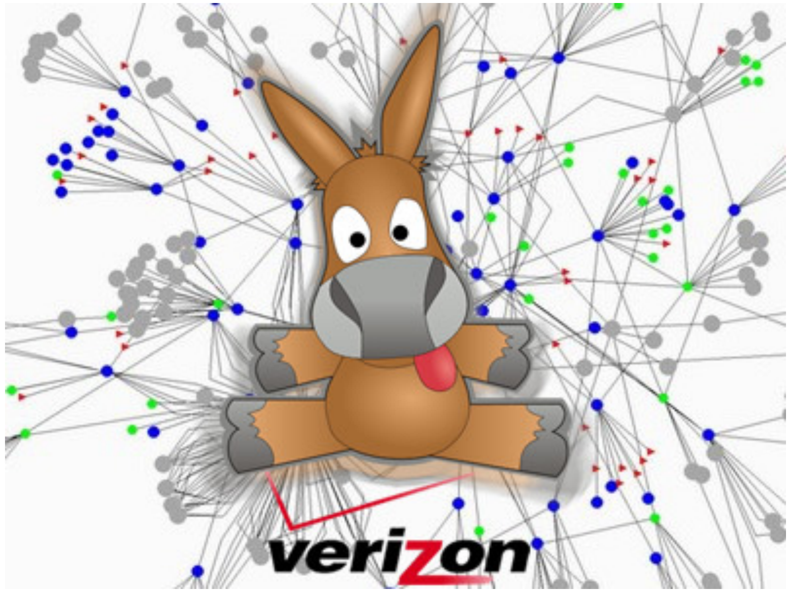
- A Bloom filter is a data structure that can implement a set.
  - It only keeps track of which keys are present, not any values associated to keys.
  - It supports insert and find operations.
  - It doesn't support delete operations.
- Bloom filters use less memory than hash tables or other ways of implementing sets.
- However, Bloom filters are approximate.
  - It can produce false positives: it says an element is present even though it's not.
    - We can bound the probability of false positives.
  - But it doesn't produce false negatives: if it says an element isn't present, then it's not.

Bloom Filter Applications

- Suppose we have a big database and querying it to check if an item is present is expensive.
- We store the set of items in the database using a Bloom filter.
  - This tells us whether an item is in database or not.
- If filter says an item's not present, it's definitely not in the database.
  - So, no need to do an expensive query.
- If filter says an item is present, then either item is present, or there's false positive.
  - When we query the database, there's a small probability we waste time querying for a nonexistent item.
- Overall, we save time by checking Bloom filter first before querying database.

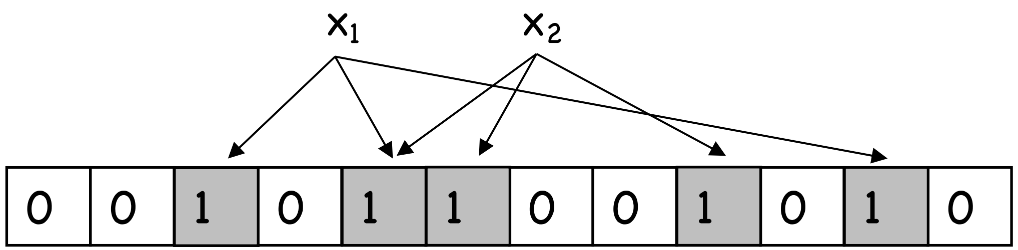
Bloom Filter Applications

- Consider a P2P network, where each node stores some files.
- If you want to get a file, you need to know which nodes have it.
- Keeping a list of all items stored at each node is too expensive.
- Instead, for every other node, keep a Bloom filter of its files.
- If filter says no for a node, it definitely doesn't have the file.
- If filter says yes, then either node has the file, or there's false positive and we make a useless request.
- Overall, we save space, and also won't waste much communication because we rarely make useless requests.



Bloom Filters

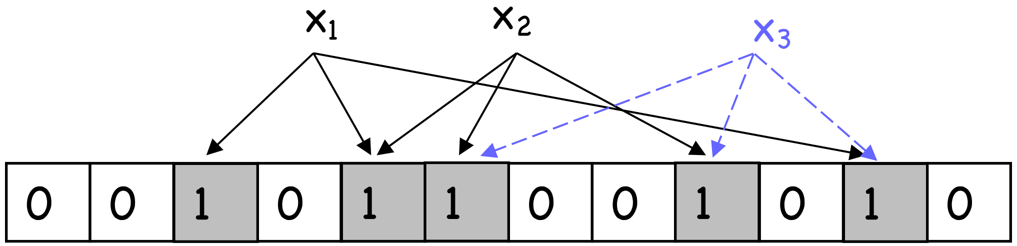
- A Bloom filter consists of
  - An array A of size m, initially all 0's.
  - k independent hash functions $h_1,\dots,h_k$, each mapping from keys to $\{1,\dots,m\}$.
- To store key x
  - Set $A[h_1(x)]$, $A[h_2(x)]$, …, $A[h_k(x)]$ all to 1.
  - Some locations can get set to 1 multiple times; that's fine.
- To check if key x is in the set
  - Read array locations $A[h_1(x)]$, $A[h_2(x)]$, …, $A[h_k(x)]$.
  - If all the values are 1, output "x is in set".
  - Otherwise, output "x is not in set".



A Bloom filter with k=3 hash functions storing 2 items.

Correctness

- Let's look at the correctness of the search function.
- If search for x returns no, then at least one of $A[h_1(x)],\ldots, A[h_k(x)]$ equals 0.
  - So x cannot be in the set, because if x had been inserted into the set, then we would have $A[h_1(x)]=\ldots=A[h_k(x)]=1$.
  - So there are no false negatives.
- If search for x returns yes, then $A[h_1(x)]=\ldots=A[h_k(x)]=1$.
  - So either x was inserted into the set.
  - Or we inserted some keys that hashed to the same k locations as x.
    - So it looks as if x was inserted, even though it wasn't.
    - This is a false positive. We'll bound the probability this happens.



False Positive Probability 1

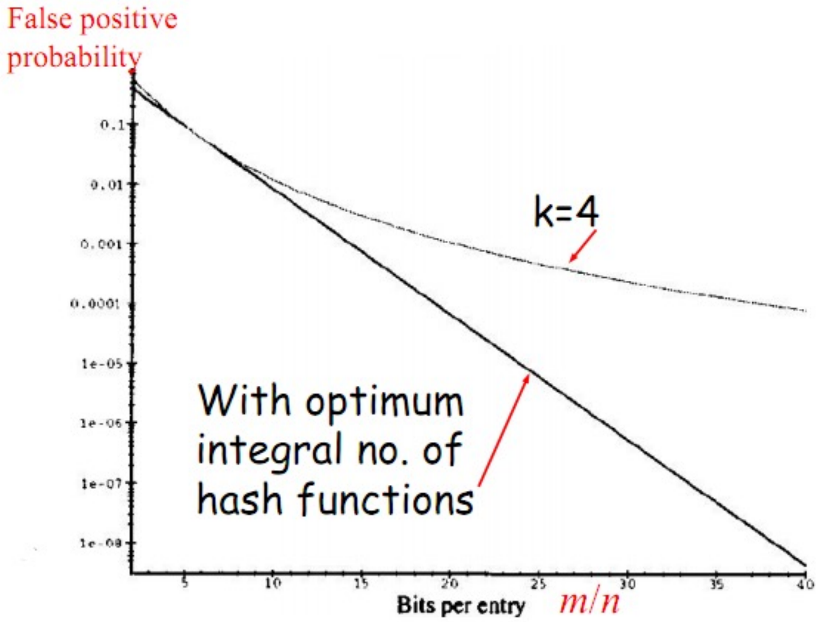
- False positive probability depends on k (number of hash functions), m (size of table) and n (number of keys inserted).
- Assume hash functions hash keys to random locations.
- When inserting one key, we set k random locations to 1.
- Fix any position i. Probability i is set to 1 by a hash function is 1/m, so probability i stays 0 is 1-1/m.
  - After k hashes, probability i still 0 is $(1-1/m)^k$.
  - To insert n items, we used nk hashes. So, probability i still 0 after all these is $p=(1-1/m)^{nk}$.
- We now use an approximation $\left(1-\frac{1}{m}\right)^{nk} \approx e^{-\frac{nk}{m}}$.

False Positive Probability 2

- So, probability any position i is 1 after n keys inserted is $1 - p \approx 1 - e^{-\frac{nk}{m}}$.
- Since there are m positions in the array, assume there are (1-p)m positions that are 1.
  - This isn't quite correct. The actual number of 1's in the array is a random variable, whose expectation is (1-p)m.
  - However, we can make the argument rigorous by showing that the actual number of 1's is $(1-p)m \pm \sqrt{m \log m}$ with high probability.
- We only get a false positive if when we check k random locations, they're all 1.
  - Probability is $f = (1-p)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k$.

False Positive Probability 3

- Notice the false prob. $\left(1 - e^{-\frac{nk}{m}}\right)^k$ is a function of k, the number of hash functions we use.
- We find k to minimize the false positive prob. by differentiating f wrt k and solving.
- The optimum k is $\frac{m\ln(2)}{n}$, which leads to $f = \left(\frac{1}{2}\right)^k \approx 0.6185^{\frac{m}{n}}$.
  - Notice that m/n is the average number of bits per item. So error rate decreases exponentially in space usage.



Improvements

- Right now, Bloom filters can't handle deletes.
  - Say keys k_1, k_2 hash to two overlapping sets of locations. If you delete k_1 by setting some of its locations to 0, you could also delete k_2.
- Deletes can be done by storing a count of how many keys hashed to that location, and inc / dec the counts when inserting or deleting.
  - But this uses more memory.
  - Also, what if the counts overflow?

Next Time:
Randomized algorithms (Cont.)