CS 362, Randomized Data Structures : Skip Lists, Bloom Filters, Count-Min sketch

Jared Saia
University of New Mexico

Outline

- Skip Lists
- Bloom Filters
- Count-Min Sketch

Dictionary ADT

A dictionary ADT implements the following operations

- *Insert(x)*: puts the item x into the dictionary
- *Delete(x)*: deletes the item x from the dictionary
- *IsIn(x)*: returns true iff the item x is in the dictionary

Skip List

- Enables insertions and searches for ordered keys in $O(\log n)$ expected time
- Very elegant randomized data structure, simple to code but analysis is subtle
- They guarantee that, with high probability, all the major operations take $O(\log n)$ time (e.g. Find-Max, Predecessor/ Sucessor)
- Can even enable "find-i-th value" if store with each edge the number of elements that edge skips

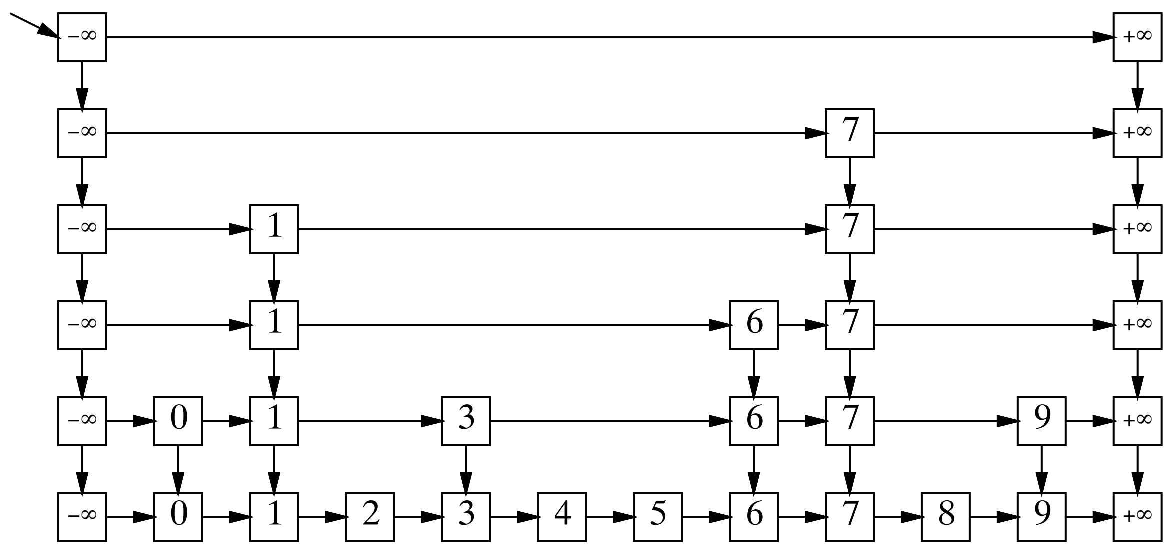
Skip List

- A skip list is basically a collection of doubly-linked lists, $L_1, L_2, \ldots, L_x$, for some integer x
- Each list has a special head and tail node, the keys of these nodes are assumed to be $-$MAXNUM and $+$MAXNUM respectively
- The keys in each list are in sorted order (non-decreasing)

Skip List

- Every node is stored in the bottom list
- For each node in the bottom list, we flip a coin over and over until we get tails. For each heads, we make a duplicate of the node.
- The duplicates are stacked up in levels and the nodes on each level are strung together in sorted linked lists
- Each node v stores a search key (key(v)), a pointer to its next lower copy (down(v)), and a pointer to the next node in its level (right(v)).

Example



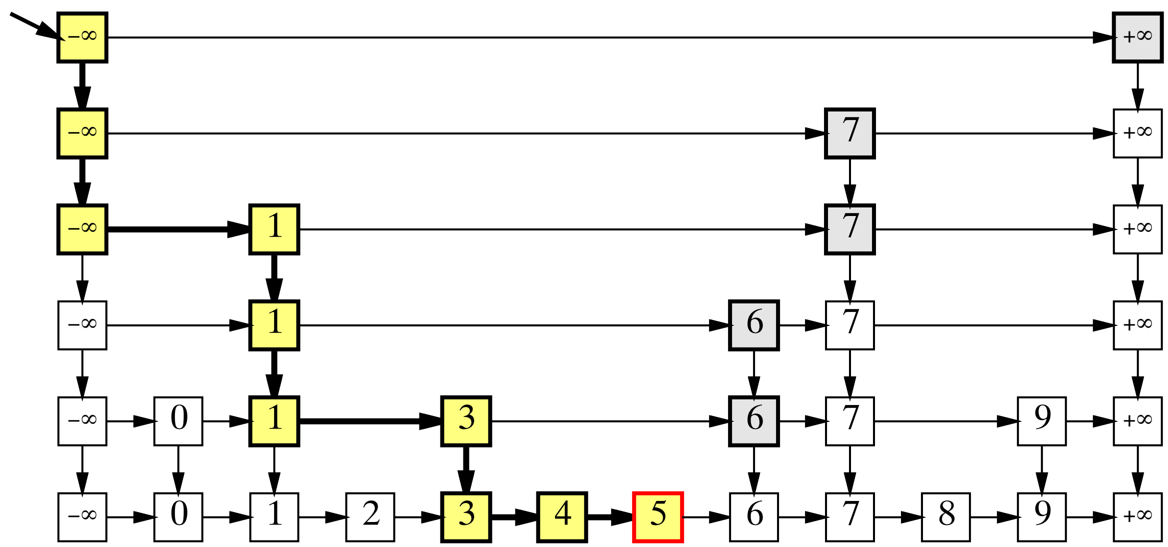
Search

- To do a search for a key, x, we start at the leftmost node L in the highest level
- We then scan through each level as far as we can without passing the target value x and then proceed down to the next level
- The search ends either when we find the key x or fail to find x on the lowest level

Search

```
SkipListFind(x, L){
  v = L;
  while (v != NULL) and (Key(v) != x){
    if (Key(Right(v)) > x)
      v = Down(v);
    else
      v = Right(v);
  }
return v;
}
```

Search Example



Insert

coin() returns "heads" with probability 1/2

```
Insert(k){
First call Search(k), let pLeft be the leftmost elem <= k in L_1
Insert k in List 0, to the right of pLeft
i = 1;
while (coin() = "heads"){
  insert k in List i;
  i++;
}
```

Deletion

- Deletion is very simple
- First do a search for the key to be deleted
- Then delete that key from all the lists it appears in from the bottom up, making sure to “zip up” the lists after the deletion

Analysis

- Intuitively, each level of the skip list has about half the number of nodes of the previous level, so we expect the total number of levels to be about $O(\log n)$
- Similarly, each time we add another level, we cut the search time in half except for a constant overhead
- So after $O(\log n)$ levels, we would expect a search time of $O(\log n)$
- We will now formalize these two intuitive observations

Height of Skip List

- For some key, k, let X_k be the maximum height of k in the skip list.
- Q: What is the probability that $X_k \geq 2\log n$?
- A: If $p = 1/2$, we have:

$$
\begin{aligned}
P(X_k \geq 2\log n) &= \left(\frac{1}{2}\right)^{2\log n} \\
&= \frac{1}{(2^{\log n})^2} \\
&= \frac{1}{n^2}
\end{aligned}
$$

- Thus the probability that a particular key k achieves height $2\log n$ is $\frac{1}{n^2}$

New Tool: Union Bound

FACT: Given two events E_1 and E_2,

$$Pr(E_1 \cup E_2) \leq Pr(E_1) + Pr(E_2)$$

Proof:

$$\begin{aligned} Pr(E_1 \cup E_2) &= Pr(E_1) + Pr(E_2) - Pr(E_1 \cap E_2) \\ &\leq Pr(E_1) + Pr(E_2) \end{aligned}$$

Generalizing to n events, we have that:

$$Pr(\cup_{i=1}^{n} E_i) \leq \sum_{i=1}^{n} Pr(E_i)$$

Height of Skip List

- Q: What is the probability that *any* key achieves height $2\log n$?
- A: We want

$$P(X_1 \geq 2\log n \text{ or } X_2 \geq 2\log n \text{ or } \ldots \text{ or } X_n \geq 2\log n)$$

- By a Union Bound, this probability is no more than

$$P(X_1 \geq 2\log n) + P(X_2 \geq 2\log n) + \cdots + P(X_n \geq 2\log n)$$

- Which equals:

$$\sum_{i=1}^{n} \frac{1}{n^2} = \frac{n}{n^2} = 1/n$$

Height of Skip List

- This probability gets small as n gets large
- In particular, the probability of having a skip list of height exceeding $2\log n$ is $o(1)$
- If an event occurs with probability $1 - o(1)$, we say that it occurs *with high probability*
- *Key Point:* The height of a skip list is $O(\log n)$ with high probability.

In-Class Exercise Trick

A trick for computing expectations of discrete positive random variables:

- Let X be a discrete r.v., that takes on values from 1 to n

$$E(X) = \sum_{i=1}^{n} P(X \geq i)$$

Why?

$$
\begin{aligned}
\sum_{i=1}^{n} P(X \geq i) &= P(X=1) + P(X=2) + P(X=3) + \ldots \\
&+ P(X=2) + P(X=3) + P(X=4) + \ldots \\
&+ P(X=3) + P(X=4) + P(X=5) + \ldots \\
&+ \ldots \\
&= 1Pr(X=1) + 2Pr(X=2) + 3Pr(X=3) + \ldots \\
&= E(X)
\end{aligned}
$$

In-Class Exercise

Q: How much memory do we expect a skip list to use up?

- Let X_k be the number of lists that key k is inserted in.
- Q: What is $P(X_k \geq 1)$, $P(X_k \geq 2)$, $P(X_k \geq 3)$?
- Q: What is $P(X_k \geq i)$ for $i \geq 1$?
- Q: What is $E(X_k)$?
- Q: Let $X = \sum_{k=1}^{n} X_k$. What is $E(X)$?

Search Time

- Its easier to analyze the search time if we imagine running the search backwards
- Imagine that we start at the found node v in the bottommost list and we trace the path backwards to the top leftmost senitel, L
- This will give us the length of the search path from L to v which is the time required to do the search

Backwards Search

```
SLFback(v){
  while (v != L){
    if (Up(v)!=NIL)
      v = Up(v);
    else
      v = Left(v);
}}
```

Backward Search

- For every node v in the skip list Up(v) exists with probability $1/2$. So for purposes of analysis, SLFBack is the same as the following algorithm:

```
FlipWalk(v){
  while (v != L){
    if (COINFLIP == HEADS)
      v = Up(v);
    else
      v = Left(v);
}}
```

Analysis

- For this algorithm, the expected number of heads is exactly the same as the expected number of tails
- Thus the expected run time of the algorithm is twice the expected number of upward jumps
- Since we already know that the number of upward jumps is $O(\log n)$ with high probability, we can conclude that the expected search time is $O(\log n)$

Bloom Filters

- Randomized data structure for representing a set. Implements:
- Insert(x) :
- IsMember(x) :
- Allow false positives but require very little space
- Used frequently in: Databases, networking problems, p2p networks, packet routing

Bloom Filters

- Have m slots, k hash functions, n elements; assume hash functions are all independent
- Each slot stores 1 bit, initially all bits are 0
- Insert(x) : Set the bit in slots $h_1(x), h_2(x), ..., h_k(x)$ to 1
- IsMember(x) : Return yes iff the bits in $h_1(x), h_2(x), ..., h_k(x)$ are all 1

Analysis Sketch

- m slots, k hash functions, n elements; assume hash functions are all independent
- Then $P(\text{fixed slot is still 0}) = (1 - 1/m)^{kn}$
- Useful fact from Taylor expansion of e^{-x}:
  $e^{-x} - x^2/2 \le 1 - x \le e^{-x}$ for $x < 1$
- Then if $x \le 1$

$$e^{-x}(1 - x^2) \le 1 - x \le e^{-x}$$

Analysis

- Thus we have the following to good approximation.

$$\begin{aligned} Pr(\text{fixed slot is still 0}) &= (1 - 1/m)^{kn} \\ &\approx e^{-kn/m} \end{aligned}$$

- Let $p = e^{-kn/m}$ and let ρ be the fraction of 0 bits after n elements inserted then

$$Pr(\text{false positive}) = (1 - \rho)^k \approx (1 - p)^k$$

- Where this last approximation holds because ρ is very close to p (by a Martingale argument beyond the scope of this class)

Analysis

- Want to minimize $(1-p)^k$, which is equivalent to minimizing $g(p) = k \ln(1-p)$
- Trick: Note that $g(p) = -(m/n)\ln(p)\ln(1-p)$
- By symmetry, this is minimized when $p = 1/2$ or equivalently $k = (m/n)\ln 2$
- False positive rate is then $(1/2)^k \approx (.6185)^{m/n}$

Tricks

- Can get the union of two sets by just taking the bitwise-or of the bit-vectors for the corresponding Bloom filters
- Can easily half the size of a bloom filter - assume size is power of 2 then just bitwise-or the first and second halves together
- Can approximate the size of the intersection of two sets - inner product of the bit vectors associated with the Bloom filters is a good approximation to this.

Extensions

- Counting Bloom filters handle deletions: instead of storing bits, store integers in the slots. Insertion increments, deletion decrements.
- Bloomier Filters: Also allow for data to be inserted in the filter - similar functionality to hash tables but less space, and the possibility of false positives.

Data Streams

- A router forwards packets through a network
- A natural question for an administrator to ask is: what is the list of substrings of a fixed length that have passed through the router more than a predetermined threshold number of times
- This would be a natural way to try to, for example, identify worms and spam
- Problem: the number of packets passing through the router is *much* too high to be able to store counts for every substring that is seen!

Data Streams

- This problem motivates the data stream model
- Informally: there is a stream of data given as input to the algorithm
- The algorithm can take at most one pass over this data and must process it sequentially
- The memory available to the algorithm is much less than the size of the stream
- In general, we won't be able to solve problems exactly in this model, only approximate

Our Problem

- We are presented with a stream of items i
- Want good approximation to the value Count(i,T), which is the number of times we have seen item i up to time T

Count-Min Sketch

- Our solution will be to use a data structure called a *Count-Min Sketch*
- This is a randomized data structure that will keep approximate values of Count(i,T)
- It is implemented using k hash functions and m counters

Count-Min Sketch

- Think of our m counters as being in a 2-dimensional array, with m/k counters per row and k rows
- Let $C_{a,b}$ be the counter in row a and column b
- Our hash functions map items from the universe into counters
- In particular, hash function h_a maps item i to counter $C_{a,h_a(i)}$

Updates

- Initially all counters are set to 0
- When we see item i in the data stream we do the following
- For each $1 \leq a \leq k$, increment $C_{a,h_a(i)}$

Count Approximations

- Let $C_{a,b}(T)$ be the value of the counter $C_{a,b}$ after processing T tuples
- We approximate Count(i,T) by returning the value of the *smallest* counter associated with i
- Let $m(i, T)$ be this value

Analysis

Theorem 1. For any $\epsilon > 0$, with probability at least $1 - e^{-m\epsilon/e}$, our Count-Min Sketch ensures for every time step T and every item i:

$$\text{Count}(i, T) \leq m(i, T) \leq \text{Count}(i, T) + \epsilon T$$

Proof

- Easy to see that $m(i, T) \geq$ Count(i,T), since each counter $C_{a,h_a(i)}$ incremented by c_t every time pair (i, c_t) is seen
- Hard Part: Showing $m(i, T) \leq$ Count(i,T) $+\epsilon T$.
- To see this, we will first consider the specific counter $C_{1,h_1(i)}$ and then use symmetry.

Proof

- Let Z_1 be a random variable giving the amount the counter is incremented by items other than i
- Let X_t be an indicator r.v. that is 1 if j is the t-th item, and $j \neq i$ and $h_1(i) = h_1(j)$
- Then $Z_1 = \sum_{t=1}^{T} X_t$
- But if the hash functions are "good", then if $i \neq j$, $Pr\left(h_1(i) = h_1(j)\right) = k/m$ (specifically, we need the hash functions to come from a 2-universal family, but we won't get into that in this class)
- Hence, $E(X_t) = k/m$

Proof

- Thus, by linearity of expectation, we have that:

$$\begin{aligned} E(Z_1) &\leq \sum_{t=1}^{T} (k/m) \\ &= Tk/m \end{aligned}$$

- We now need to make use of a very important inequality: Markov's inequality

Markov's Inequality

Markov's Inequality. Let X be a random variable that only takes on non-negative values. Then for any $\lambda > 0$:

$$Pr(X \geq \lambda) \leq E(X)/\lambda$$

- Proof: Assume by contradiction that there exists a λ such that $Pr(X \geq \lambda)$ was actually larger than $E(X)/\lambda$
- But then the expected value of X would be at least $\lambda Pr(X \geq \lambda) > E(X)$, which is a contradiction!!!

Proof (Using Markov's)

- Now, by Markov's inequality,

$$Pr(Z_1 \geq \epsilon T) \leq \frac{Tk/m}{\epsilon T} = \frac{k}{m\epsilon}$$

- This is the event where Z_1 is "bad" for item i, in the sense that it gives more than a ϵT overestimate of how often item i has been seen.

Proof (Using Independence)

- Since our k hash functions are "good" in the sense that they are independent, we have

$$\prod_{i=1}^{k} Pr(Z_j \geq \epsilon T) \leq \left(\frac{k}{m\epsilon}\right)^k$$

Proof (Choosing k)

- Finally, we want to choose a k that minimizes $f(k) = \left(\frac{k}{m\epsilon}\right)^k$
- Note that $\frac{\partial f}{\partial k} = \left(\frac{k}{m\epsilon}\right)^k \left(\ln \frac{k}{m\epsilon} + 1\right)$
- From this, we can see that the probability is minimized when $k = m\epsilon/e$, in which case:

$$\left(\frac{k}{m\epsilon}\right)^k = e^{-m\epsilon/e}$$

Recap

- Our Count-Min Sketch is very good at giving estimating counts of items with very little external space
- Tradeoff is that it only provides approximate counts, but we can bound the approximation!
- Note: Can use the Count-Min Sketch to keep track of all the items in the stream that occur more than a given threshold ("heavy hitters")
- Basic idea is to store an item in a list of "heavy hitters" if its count estimate ever exceeds some given threshold