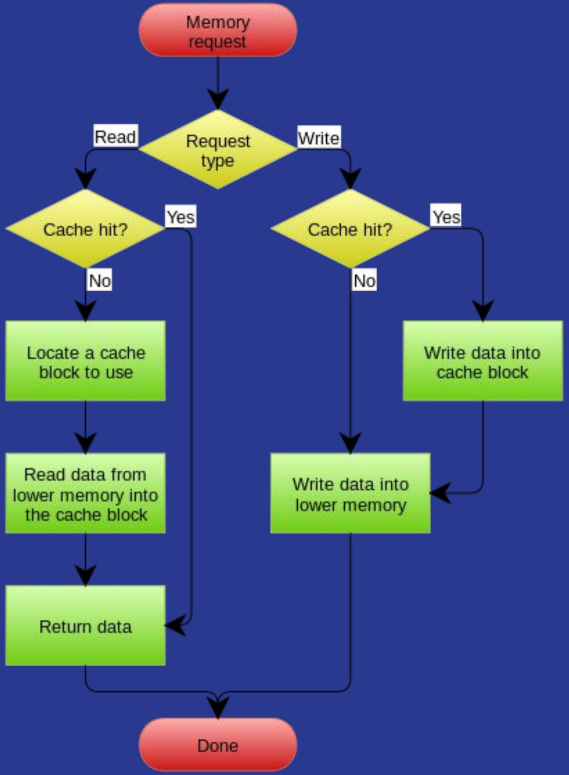
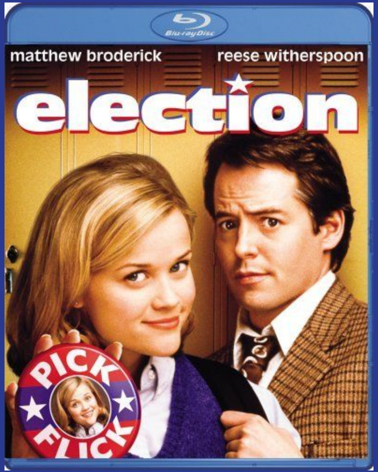
CS 9

Week 8 Problems

Andrew Benson
Ian Tullis


Memory request
Request type
Read
Write
Cache hit?
Yes
No
Cache hit?
Yes
No
Locate a cache block to use
Write data into cache block
Read data from lower memory into the cache block
Write data into lower memory
Return data
Done



Announcements

- Reminder: the university-wide withdrawal deadline is this Friday, 5 PM.
    - We want everyone to pass this class! If you think you are in danger of not passing, please reach out. Ian will also be emailing people tonight.
    - In general, we can offer options!

- If you ever didn't get a lecture password for some logistical reason, let Ian know. (We want you to get credit for the time you spent!)

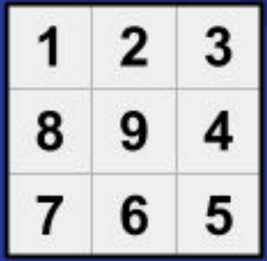
- Thanks for the great questions at this week's panel! We will be holding a second panel next Tuesday, with a different mix of companies/backgrounds represented.

⭐ Problem 8-1: Spiral

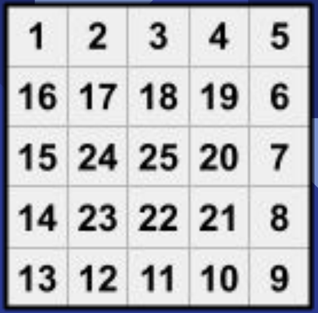
This is a pretty common question!

- You are given the dimensions *R* and *C* (each between 1 and 100) of a grid, i.e., the number of rows and columns.
- Your goal is to produce a path that starts at the upper left cell, heading to the right, and moves in a clockwise spiral around the grid, making a 90 degree right turn whenever the path encounters a grid boundary or a cell already used by the path.
- Represent the path as a list of coordinates. For example, for a grid with R = 3, C = 4, the answer is: `[(1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (3, 3), (3, 2), (3, 1), (2, 1), (2, 2), (2, 3)]`

⭐ Problem 8-1: Spiral



1	2	3
8	9	4
7	6	5



1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

"Ugh, isn't this just a busywork question?
There's no elegant algorithm to figure out!"

- The interviewer probably wants to see:
  - Do you come up with a reasonable design first?
  - Even though the idea/description is straightforward, how do you implement the idea?
  - Is your code clean and easy to read? How much redundancy is there?
  - Does it actually work, or are there off-by-one errors, etc.?
  - Does your code handle edge cases?

⭐⭐ Problem 8-2: LRU Cache

- Suppose you have a large but slow collection of *N* key-value pairs.
- You receive a sequence of requests where each request is a key, and in response you should send the value for that key.
- As an optimization, you add a cache. A Least Recently Used cache stores the *K* distinct most recently requested key-value pairs.
  - If a requested key is in the cache, its value can be returned.
  - Otherwise, the least recently requested key-value pair is evicted (hence the name "LRU"), and the newly requested key-value pair is added. Then the value can be returned.
- Design an **efficient** implementation of this data structure. (assume *N* can be quite large, and *K* is relatively smaller)
- (Follow-up: How would you design a *Most* Recently Used cache? What might that be good for?)

⭐⭐ Problem 8-2: LRU Cache

Example for SUNET-firstname pairs with *K* = 2:

- Cache is initially empty.
- Request for "adbenson": Cache fetches and serves "Andrew".
- Request for "itullis": Cache fetches and serves "Ian".
- Request for "adbenson": Cache serves "Andrew".
- Request for "tiffanyo": Cache evicts "itullis/Ian" (least recently used) and fetches and serves "Tiffany".
- Request for "itullis": Cache evicts "adbenson/Andrew" (least recently used) and fetches and serves "Ian".

⭐⭐⭐ Problem 8-3: Election Winner

Another one that is good to know (and that my manager used to ask), even though it is maybe a bit too "aha" for an interview question.

- *N* people (up to 100000) just voted in an election. Each voter gave the number (between 1 and *N*) of their preferred candidate.
- It is known that one candidate won a strict majority (i.e., over half of the votes), but no one remembers *who* won.
- You are presented with these votes in a streaming fashion – that is, you can only see one vote at a time, in some linear order.
- Determine the winner using as little *space* as possible.
  - Warm-up: The order is guaranteed to be random, and you only need to be 99.9% sure.
  - Harder: No guarantees on the order. **And you have to be right.**

Solutions to 8-2 (discussed in-class)

⭐⭐ Problem 8-2: LRU Cache Solution

What does “efficient” mean in this context?

- Time complexity: We want to return values for key requests quickly
  - Quickly check/update elements
  - Quickly find/remove the least recently used element
  - Quickly insert the new element
- Space complexity: We want to keep our memory footprint low

⭐⭐ Problem 8-2: LRU Cache Solution

- How about a hash table (key -> value, timestamp)?
  - O(1) check/update elements
  - O(k) find/remove LRU element
    - since we'd have to store some timestamp of most recent access, and then check all the timestamps…
  - O(1) insert new element
  - O(k) space. Nice!

⭐⭐ Problem 8-2: LRU Cache Solution

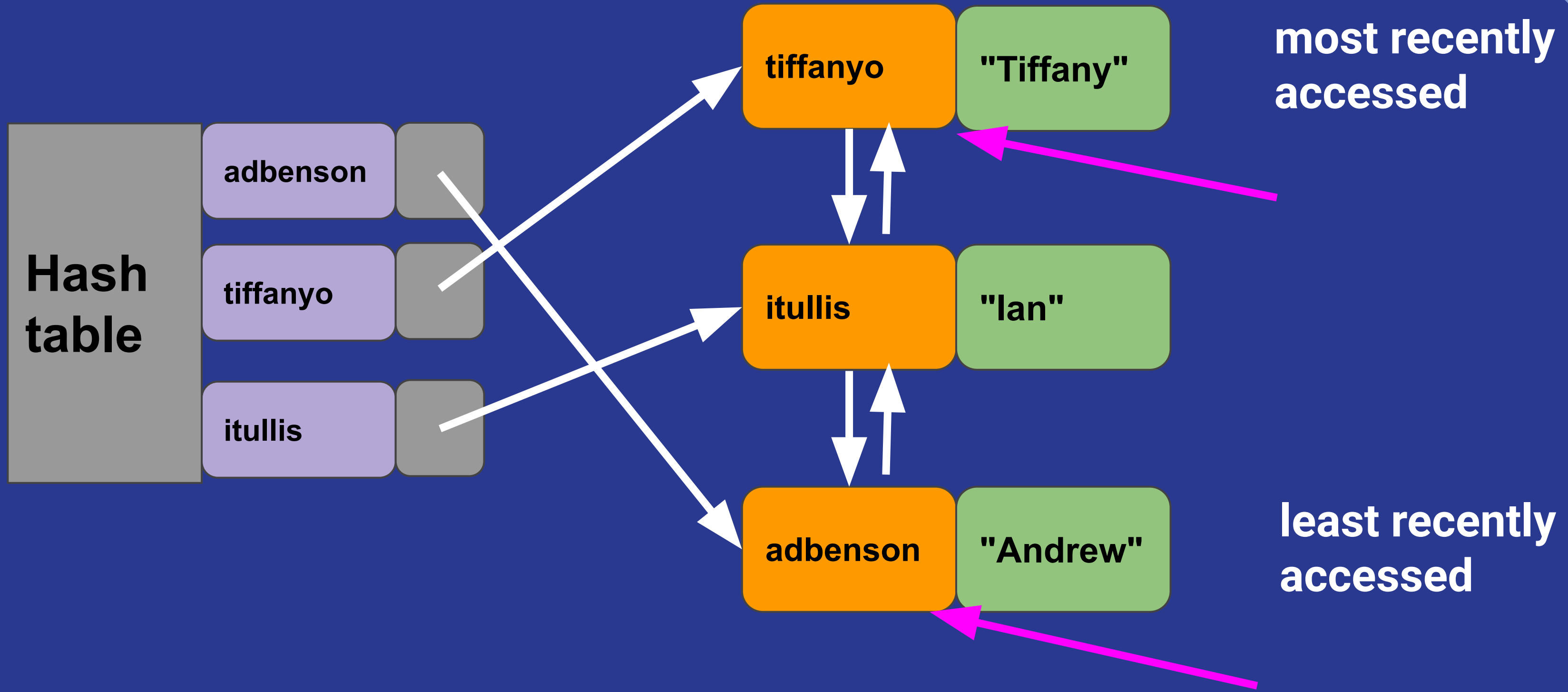
- How about a time-ordered doubly linked list (each node = key, value)?
  - O(k) check/update elements
    - since we have to potentially search the entire linked list to find the element we want
  - O(1) find/remove LRU element
  - O(1) insert new element
  - O(k) space. Nice!

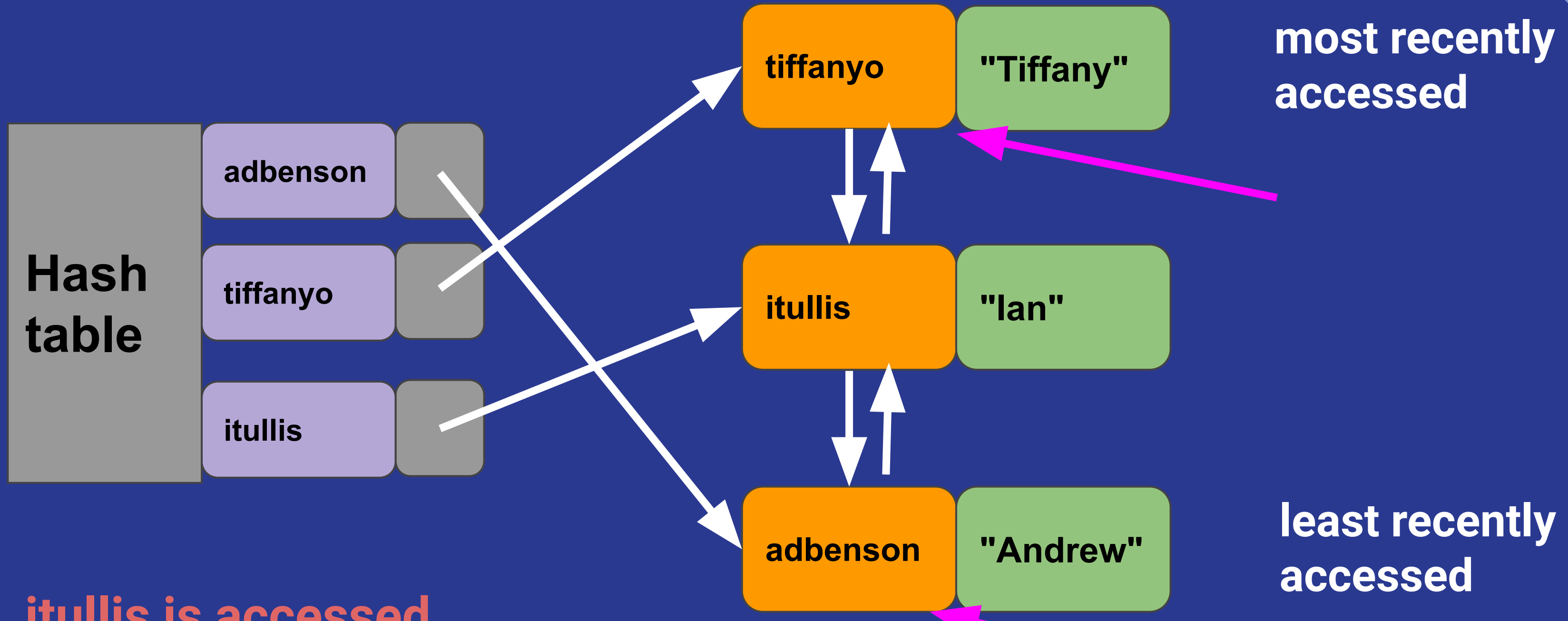
⭐⭐ Problem 8-2: LRU Cache Solution

- Some sort of balanced tree so that we can get O(logn) operations? Probably possible with a treap (node = key, value, timestamp)…?
  - O(log *k*) check/update elements
  - O(log *k*) find/remove LRU element (at head)
  - O(log *k*) insert new element
  - O(*k*) space. Nice!

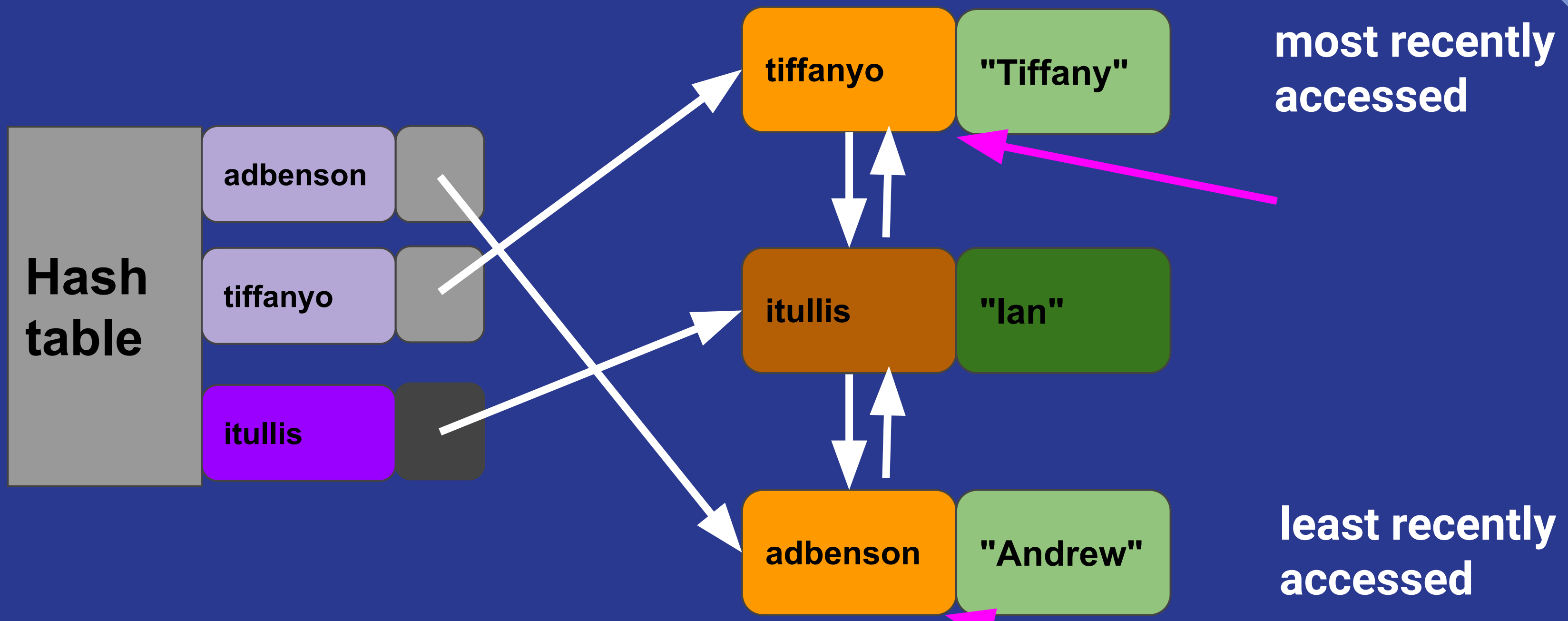
⭐⭐ Problem 8-2: LRU Cache Solution

- A balanced tree works, but if we want to beat that, we pretty much need to get O(1) time complexity
  - Hash table gets us O(1) lookup and update but not finding the LRU element
  - Time-ordered linked list gets us O(1) finding the LRU element, but not lookup/update
  - So…some sort of unholy combination of the two?

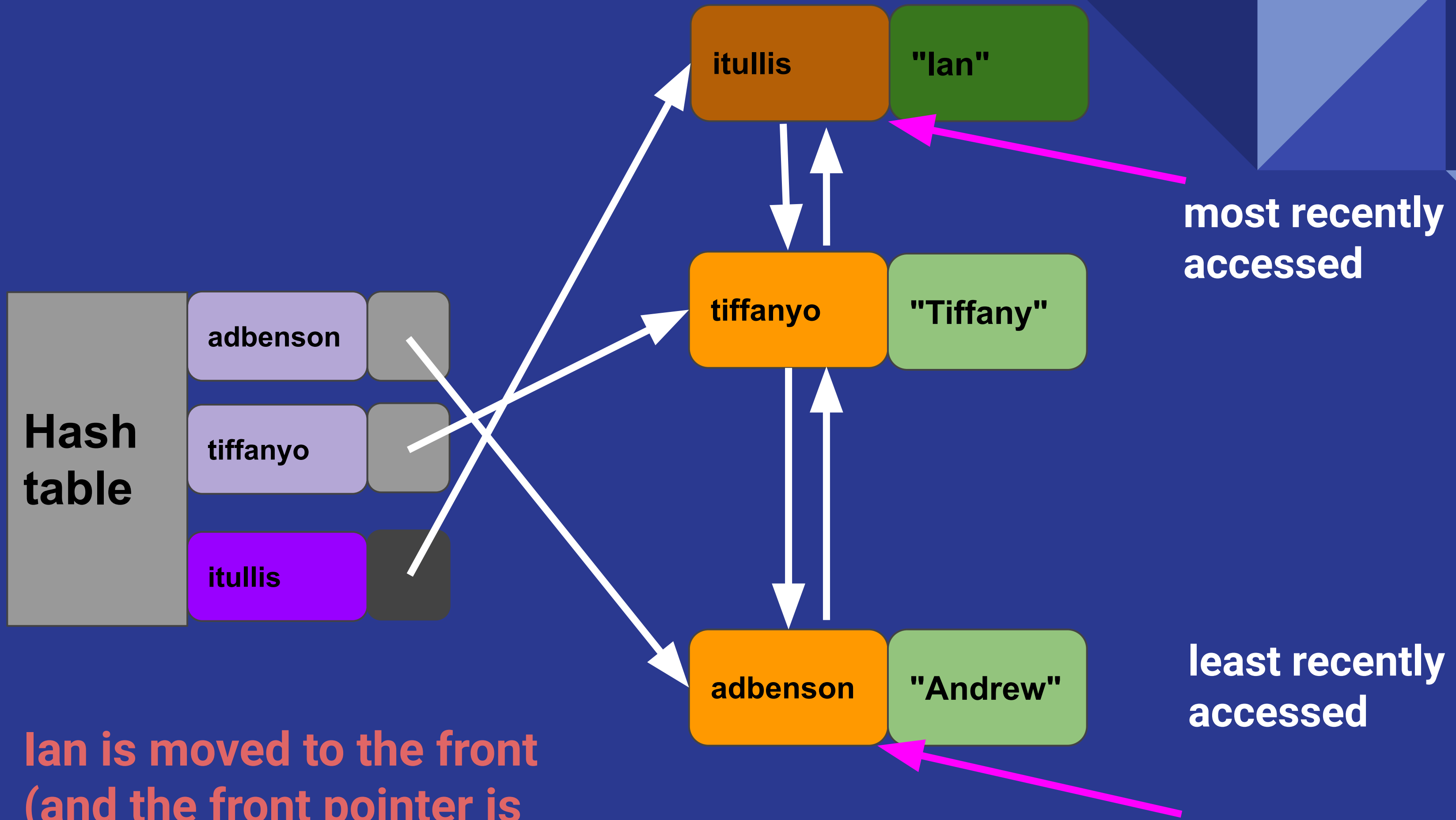

Hash table
adbenson
tiffanyo
itullis
tiffanyo
"Tiffany"
itullis
"Ian"
adbenson
"Andrew"
most recently accessed
least recently accessed



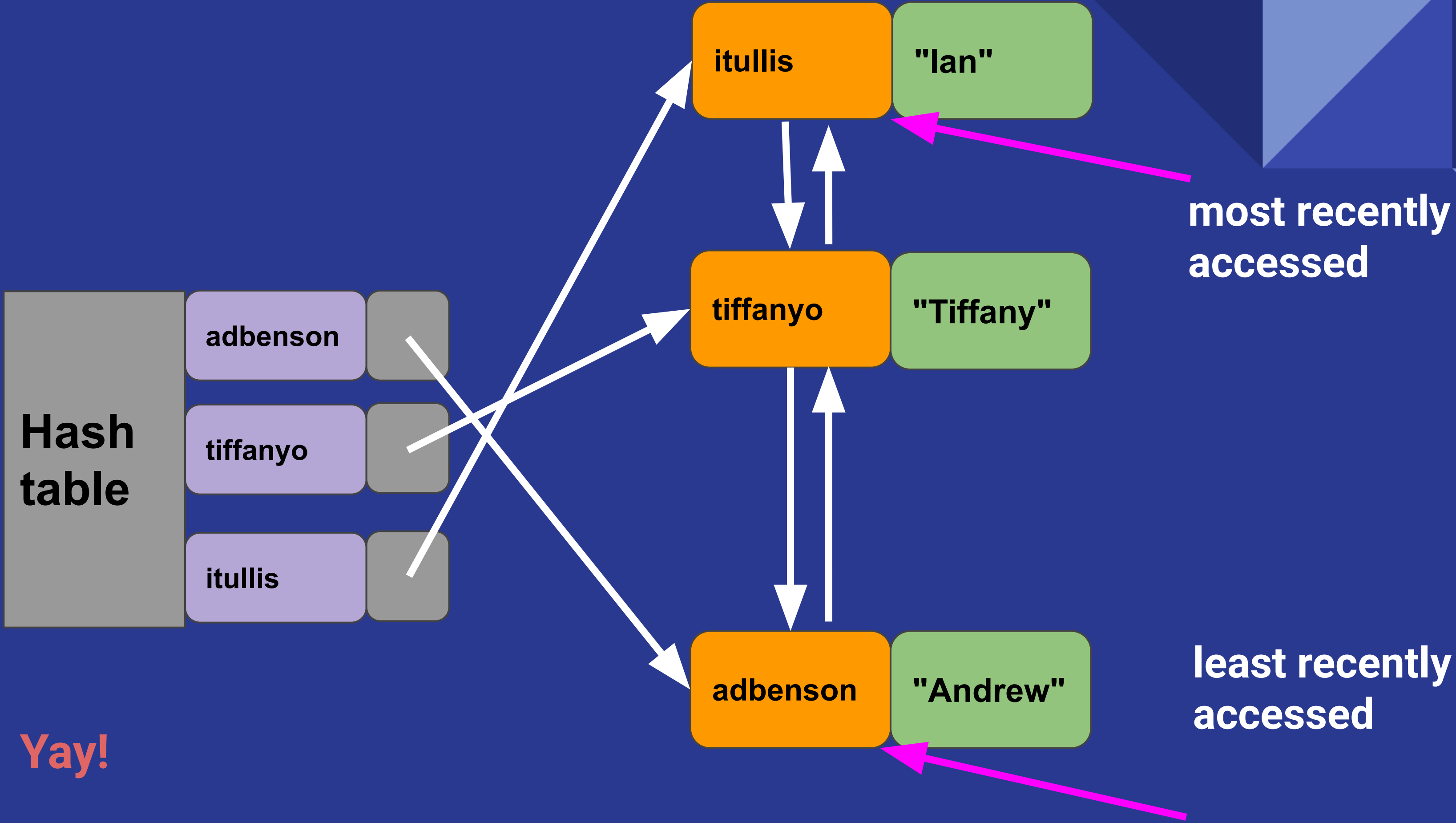
itullis is accessed



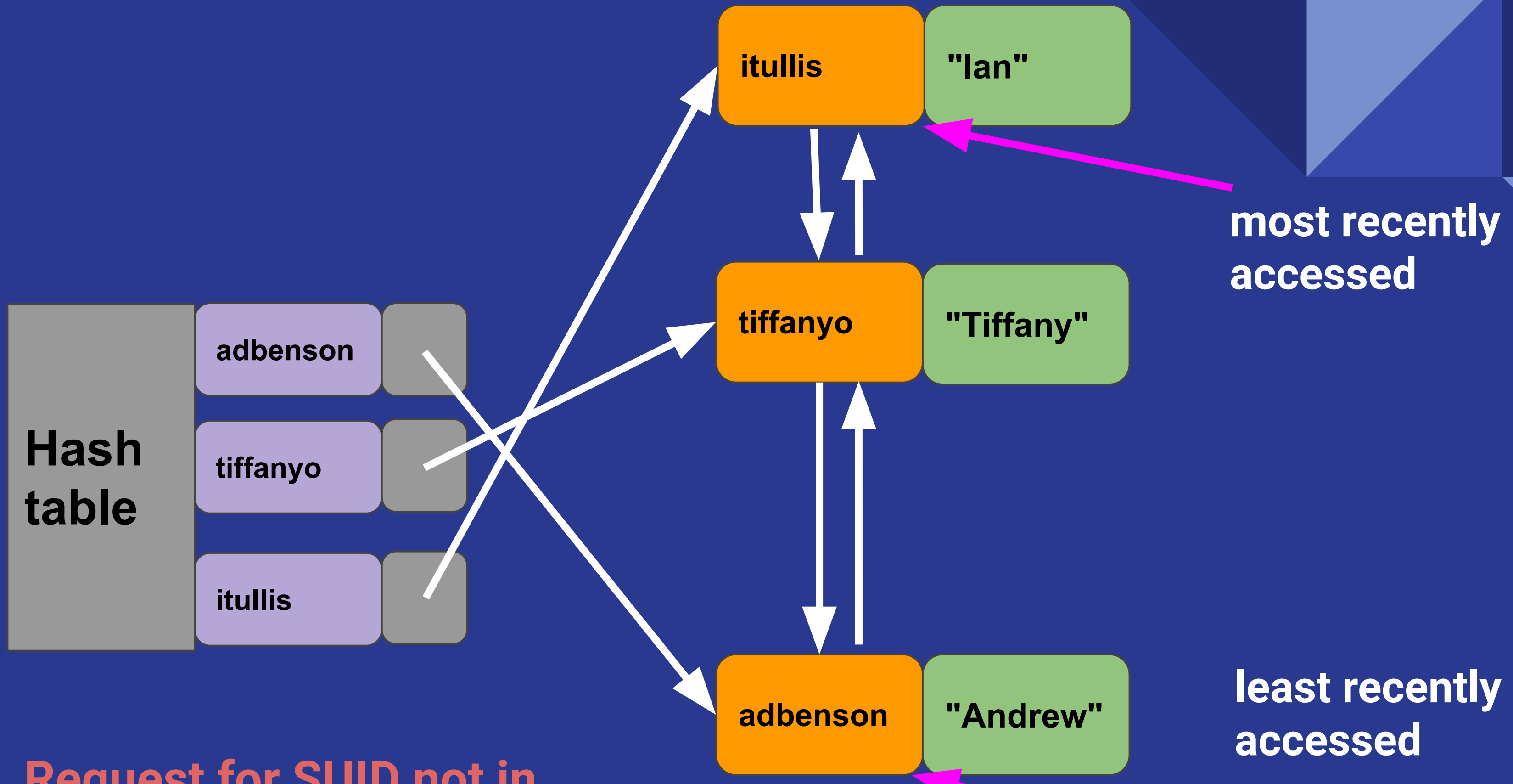
itullis is accessed
"Ian" is returned
and we didn't have to search the linked list!



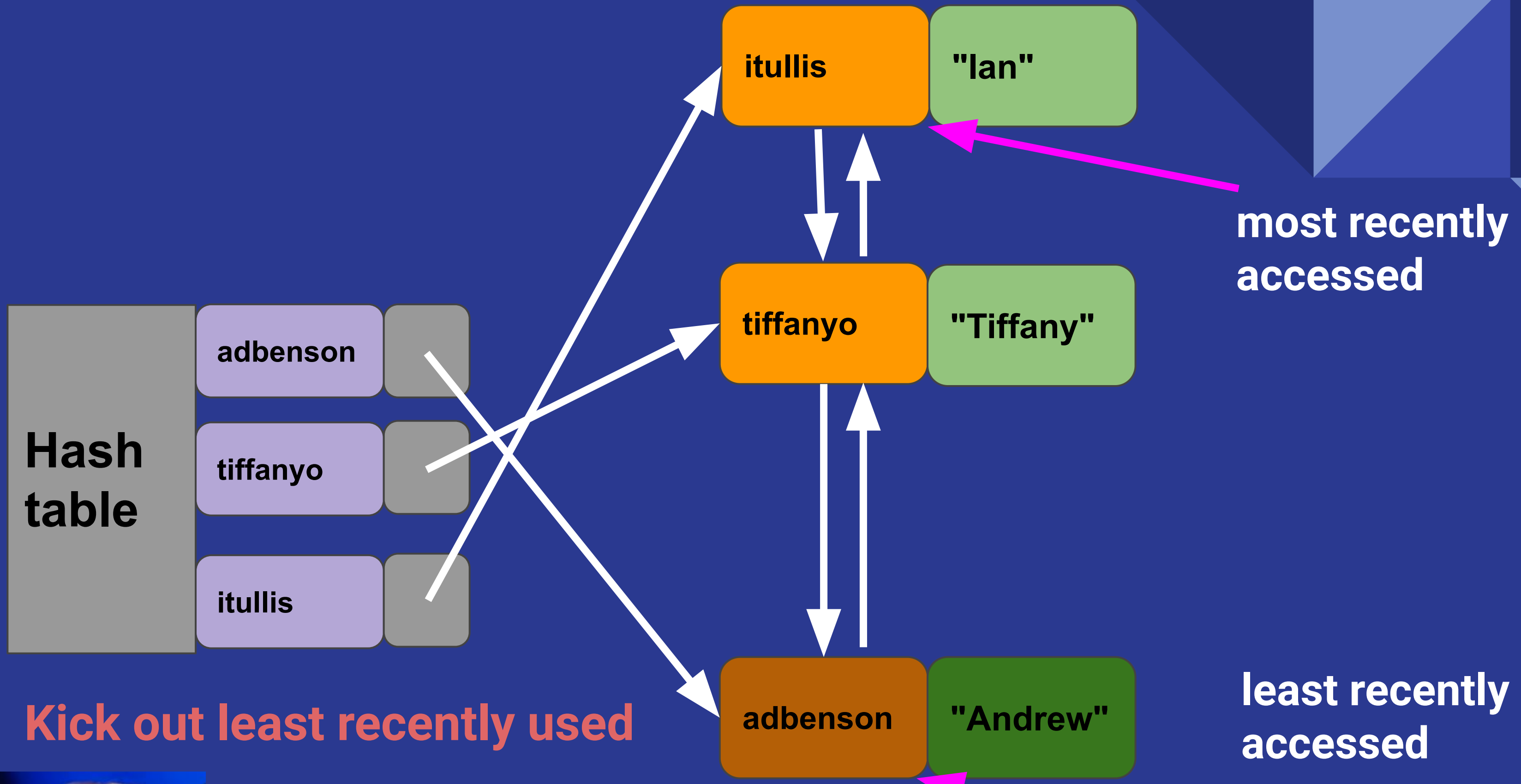
Ian is moved to the front (and the front pointer is updated)


itullis
"Ian"
tiffanyo
"Tiffany"
adbenson
"Andrew"
most recently accessed
least recently accessed
Hash table
adbenson
tiffanyo
itullis

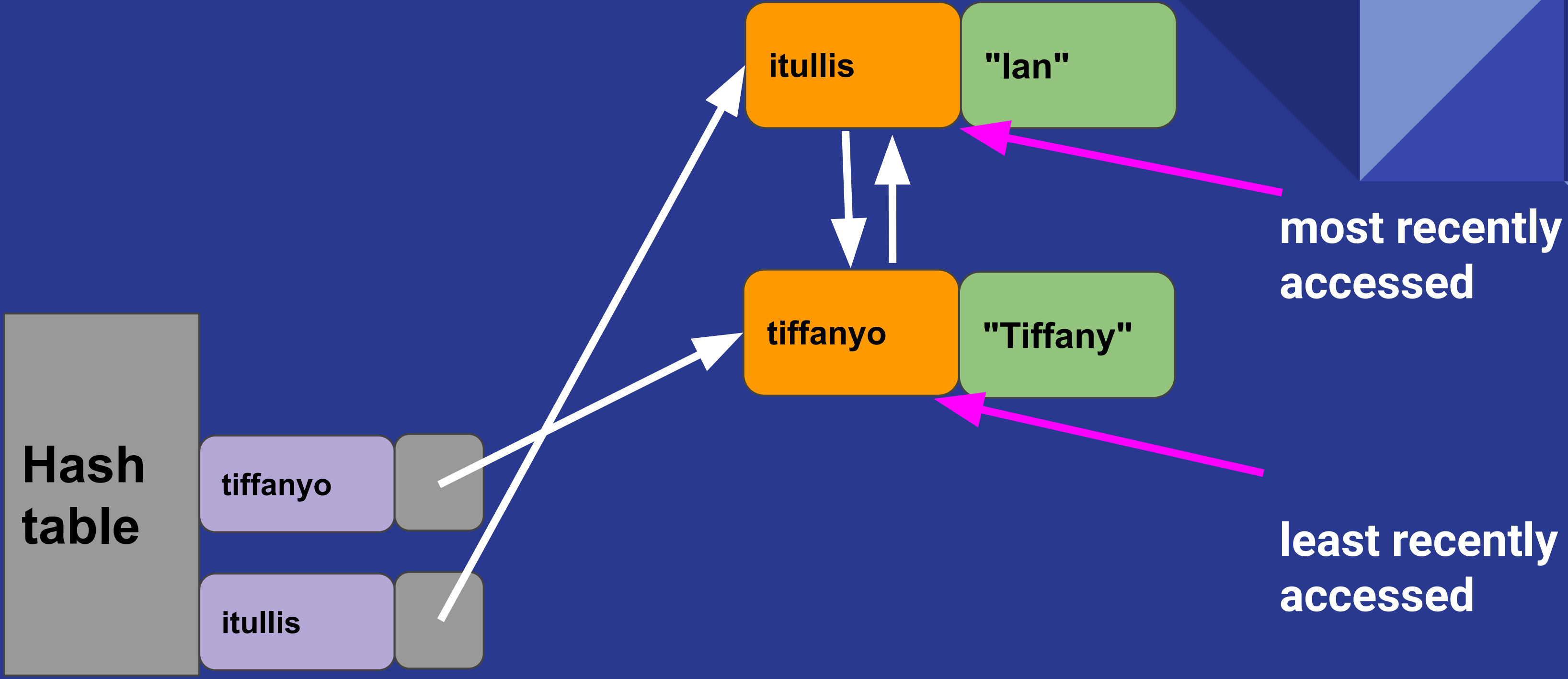
Yay!



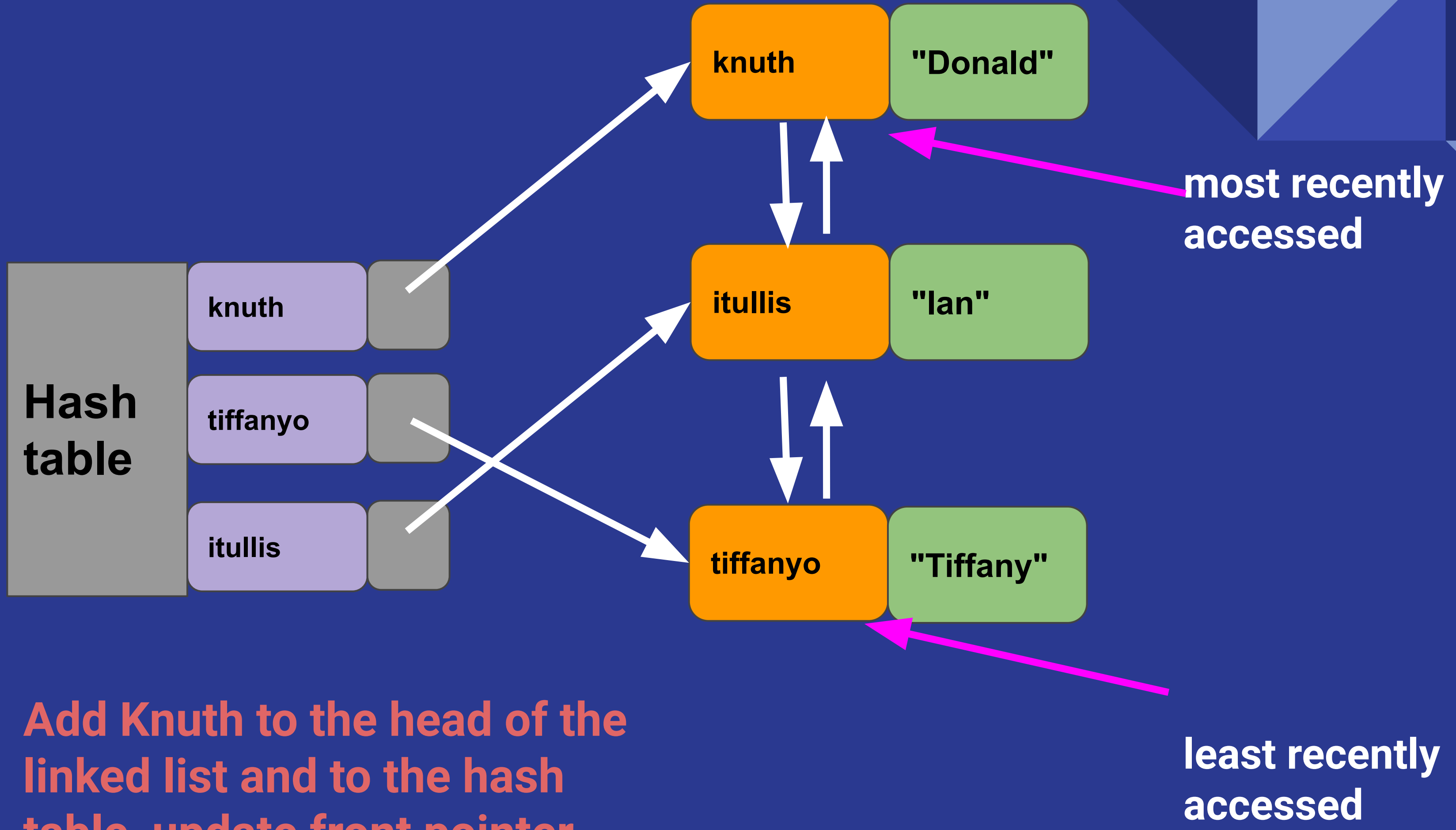
Request for SUID not in cache: dknuth



Kick out least recently used



back pointer
is updated too



Add Knuth to the head of the linked list and to the hash table, update front pointer

And so on!

- Remember that the linked list elements are not actually moving around in memory, which would be costly. We're just rewiring constant numbers of pointers, which is not costly.
- Notice that it's important that the linked list is doubly linked, specifically because we need to be able to remove an element from the middle of the list and then reconnect its neighbors.
- See, despite their non-searchability and slowness, linked lists really are more useful than arrays sometimes!

Don't let Ian forget to say the password!!!!!!

Solutions to 8-3

The random case

- Even if the votes are in a random order (for the warm-up version), what are some worst-case scenarios we might have to deal with?
  - almost exactly tied election, with N odd, (N-1)/2 votes for winner, ((N-1)/2) + 1 votes for runner-up
  - or, all votes not for the winner are all for distinct candidates
  - or, all votes not for the winner are distributed among some relatively small number of candidates…

- If the order is random, we can, e.g., assume that the first 10 elements contain the majority with very high probability (around 1023/1024), and then just keep a count of each of those.

The other case is hard!

- We can maintain a hash table and keep track of the votes for every distinct candidate, but there could be N/2 total candidates in the worst case!
    - and the first N/2 votes could all be for different candidates, so it wouldn't be obvious who, if anyone, to throw out

- A reasonable idea is to try to only hold onto counts for the top X candidates, for some X, but how do we decide when to kick someone out of our list? What if we do, and that info later turns out to have been crucial?

A surprising solution

Amazingly, there is a super simple (but very difficult to just come up with) solution to this problem:

- Maintain two values: a vote count V (initially 0) and a candidate label C (initially empty)
- For each vote in the stream:
    - If V = 0, set V to 1 and C to that candidate.
    - If V is not 0:
        - If the vote is for our candidate, V += 1.
        - If the vote is not for our candidate, V -= 1.
- At the end, return C.

How can this possibly work?

- Let's turn it around: how could this possibly *not* work?
- Every one of the winner's votes would have to be cancelled out by someone else.
- But that's not possible, since the winner has strictly more votes.
- So even though the winner is not necessarily our stored candidate the whole time, they will be at the end.

Thanks to Tim Roughgarden and Greg Valiant for introducing this problem in CS168! If you're interested in these streaming problems, consider CS168, 261, or 368.