Lecture 9: Probability and Hashing

Sam McCauley

October 4, 2024

Williams College

Admin

- Apply to be a TA in the Spring (you should have gotten an email)
- Assignment 1 deadline Saturday
- Any lingering Assignment 1 questions?
- Homework 2 back
- Homework 3 out tonight
- Maybe short day today? We'll wrap up probability analysis and some other lingering topics

Useful formulas for probability ($e = 2.71\ldots$)

Two useful approximations for simplifying exponents (presented as inequalities, but really quite tight even for moderate n):

$$(1 + 1/n)^n \leq e \qquad\qquad (1 - 1/n)^n \leq 1/e$$

Example: $(1.1)^{10} = 2.593\ldots$

Useful formulas for probability ($e = 2.71\ldots$)

Two useful approximations for simplifying exponents (presented as inequalities, but really quite tight even for moderate n):

$$(1 + 1/n)^n \leq e \qquad\qquad (1 - 1/n)^n \leq 1/e$$

Example: $(1.1)^{10} = 2.593\ldots$

With probability we often use choose (a.k.a. binomial) notation, but it's unweidly. These inequalities can help approximate it:

$$\left(\frac{x}{y}\right)^y \leq \binom{x}{y} \leq \left(\frac{ex}{y}\right)^y$$

Example: $\binom{n}{10} = \Theta(n^{10})$

Expectation

Random Variable

- A variable whose values depend on the outcome of a random process

Random Variable

- A variable whose values depend on the outcome of a random process
- We’re using mostly for the sake of notation

Random Variable

- A variable whose values depend on the outcome of a random process
- We're using mostly for the sake of notation
- Let's say I draw four cards from a deck of cards. Let S be a random variable indicating the number of clubs I draw.

Random Variable

- A variable whose values depend on the outcome of a random process
- We're using mostly for the sake of notation
- Let's say I draw four cards from a deck of cards. Let S be a random variable indicating the number of clubs I draw.
- What can we say about S?

Random Variable

- A variable whose values depend on the outcome of a random process
- We're using mostly for the sake of notation
- Let's say I draw four cards from a deck of cards. Let S be a random variable indicating the number of clubs I draw.
- What can we say about S?
    - S is at least 0 and at most 4

Random Variable

- A variable whose values depend on the outcome of a random process
- We’re using mostly for the sake of notation
- Let’s say I draw four cards from a deck of cards. Let S be a random variable indicating the number of clubs I draw.
- What can we say about S?
  - S is at least 0 and at most 4
  - What is the probability that S is 0?

Random Variable

- A variable whose values depend on the outcome of a random process
- We're using mostly for the sake of notation
- Let's say I draw four cards from a deck of cards. Let S be a random variable indicating the number of clubs I draw.
- What can we say about S?
  - S is at least 0 and at most 4
  - What is the probability that S is 0?
  - $13/52 \cdot 13/51 \cdot 13/50 \cdot 13/49 \approx .0043$

Random Variable

- A variable whose values depend on the outcome of a random process
- We're using mostly for the sake of notation
- Let's say I draw four cards from a deck of cards. Let S be a random variable indicating the number of clubs I draw.
- What can we say about S?
    - S is at least 0 and at most 4
    - What is the probability that S is 0?
    - $13/52 \cdot 13/51 \cdot 13/50 \cdot 13/49 \approx .0043$
    - What is the probability that S is 4?

Random Variable

- A variable whose values depend on the outcome of a random process
- We’re using mostly for the sake of notation
- Let’s say I draw four cards from a deck of cards. Let S be a random variable indicating the number of clubs I draw.
- What can we say about S?
  - S is at least 0 and at most 4
  - What is the probability that S is 0?
  - $13/52 \cdot 13/51 \cdot 13/50 \cdot 13/49 \approx .0043$
  - What is the probability that S is 4?
  - $13/52 \cdot 12/51 \cdot 11/50 \cdot 10/49 \approx .00264$

Random Variable

- A variable whose values depend on the outcome of a random process
- We're using mostly for the sake of notation
- Let's say I draw four cards from a deck of cards. Let S be a random variable indicating the number of clubs I draw.
- What can we say about S?
    - S is at least 0 and at most 4
    - What is the probability that S is 0?
    - $13/52 \cdot 13/51 \cdot 13/50 \cdot 13/49 \approx .0043$
    - What is the probability that S is 4?
    - $13/52 \cdot 12/51 \cdot 11/50 \cdot 10/49 \approx .00264$
- Since each card is a club with probability (about) $1/4$, and we draw 4 cards, it seems like S should generally be around 1. Can we formalize this intuition?

Expectation

- When we make random decisions, we often care about the *average* outcome

Expectation

- When we make random decisions, we often care about the *average* outcome
- Example: let’s say I flip a fair coin until I get a heads. How long will it take me on average?

Expectation

- When we make random decisions, we often care about the *average* outcome
- Example: let's say I flip a fair coin until I get a heads. How long will it take me on average?
- 2 flips

Expectation

- When we make random decisions, we often care about the *average* outcome
- Example: let’s say I flip a fair coin until I get a heads. How long will it take me on average?
- 2 flips
- Another example: Consider a quicksort implementation that chooses each pivot at random.

Expectation

- When we make random decisions, we often care about the *average* outcome
- Example: let's say I flip a fair coin until I get a heads. How long will it take me on average?
- 2 flips
- Another example: Consider a quicksort implementation that chooses each pivot at random.
- This algorithm takes $O(n \log n)$ time in expectation.

Definition of Expectation

- Let's say a random variable X takes values $\{1, \ldots k\}$

Definition of Expectation

- Let's say a random variable X takes values $\{1, \ldots k\}$
- Then the expectation of X is

$$\mathrm{E}[X] = \sum_{i=1}^{k} i \cdot \Pr[X = i].$$

Definition of Expectation

- Let's say a random variable X takes values $\{1, \ldots k\}$

- Then the expectation of X is

$$\mathrm{E}[X] = \sum_{i=1}^{k} i \cdot \Pr[X = i].$$

- It is a weighted average of the outcomes

Expectation example

Let’s say I roll a 20-sided die, and I give you money equal to the number that shows up on top. I charge \$10 to play this game. Should you play it?

Let’s look at what you win on average

- Random variable X to represent how much you win

Expectation example

Let’s say I roll a 20-sided die, and I give you money equal to the number that shows up on top. I charge $10 to play this game. Should you play it?

Let’s look at what you win on average

- Random variable X to represent how much you win
- $\mathrm{E}[X] = \sum_{i=1}^{20} i/20$

Expectation example

Let's say I roll a 20-sided die, and I give you money equal to the number that shows up on top. I charge \$10 to play this game. Should you play it?

Let's look at what you win on average

- Random variable X to represent how much you win
- $\mathrm{E}[X] = \sum_{i=1}^{20} i/20$
- $\mathrm{E}[X] = \frac{20 \cdot 21}{2 \cdot 20} = 10.5$

Expectation example

Let’s say I roll a 20-sided die, and I give you money equal to the number that shows up on top. I charge \$10 to play this game. Should you play it?

Let’s look at what you win on average

- Random variable X to represent how much you win
- $\mathrm{E}[X] = \sum_{i=1}^{20} i/20$
- $\mathrm{E}[X] = \frac{20 \cdot 21}{2 \cdot 20} = 10.5$
- So you’ll win \$.50 on average; you should probably play the game

Independence of Random Variables

- Intuition: Two random variables are independent if the value of one does not depend on the value of the other.

Independence of Random Variables

- Intuition: Two random variables are independent if the value of one does not depend on the value of the other.
- More formally: X is independent of Y if for all i and j, $\Pr[X = j | Y = i] = \Pr[X = j]$. Let's look at intuitive examples

Independence of Random Variables

- Intuition: Two random variables are independent if the value of one does not depend on the value of the other.
- More formally: X is independent of Y if for all i and j, $\Pr[X = j | Y = i] = \Pr[X = j]$. Let's look at intuitive examples
- Example: let X_1 denote the number of heads on my first coin flip, and X_2 denote the number of heads on my second coin flip. These are independent.

Independence of Random Variables

- Intuition: Two random variables are independent if the value of one does not depend on the value of the other.
- More formally: X is independent of Y if for all i and j, $\Pr[X = j | Y = i] = \Pr[X = j]$. Let's look at intuitive examples
- Example: let X_1 denote the number of heads on my first coin flip, and X_2 denote the number of heads on my second coin flip. These are independent.
- But: let X_H denote the number of heads I flip over k coin flips, and X_T denote the number of tails. These are not independent.

Linearity of Expectation

- Consider a random variable that can be represented as the sum of other random variables (we’ll see an example in a moment)

Linearity of Expectation

- Consider a random variable that can be represented as the sum of other random variables (we'll see an example in a moment)
- $X = X_1 + X_2 + \ldots + X_n$

Linearity of Expectation

- Consider a random variable that can be represented as the sum of other random variables (we'll see an example in a moment)
- $X = X_1 + X_2 + \ldots + X_n$
- Then $\mathrm{E}[X] = \mathrm{E}[X_1] + \mathrm{E}[X_2] + \ldots + \mathrm{E}[X_n]$

Linearity of Expectation

- Consider a random variable that can be represented as the sum of other random variables (we'll see an example in a moment)
- $X = X_1 + X_2 + \ldots + X_n$
- Then $\mathrm{E}[X] = \mathrm{E}[X_1] + \mathrm{E}[X_2] + \ldots + \mathrm{E}[X_n]$
- True even if the X_i are not independent!!!

Using Linearity of Expectation

- Let’s say I flip a coin 100 times. How many heads will I see on average?

Using Linearity of Expectation

- Let’s say I flip a coin 100 times. How many heads will I see on average?
- Let’s figure this out on the board using linearity of expectation

Using Linearity of Expectation

- Let's say I flip a coin 100 times. How many heads will I see on average?
- Let's figure this out on the board using linearity of expectation
- $X =$ number of heads I see in 100 flips.
- $$X_i = \begin{cases} 1 & \text{if the } i\text{th flip is heads} \\ 0 & \text{otherwise} \end{cases}$$
- So then $X = X_1 + X_2 + \ldots X_{100}$
- We can see that $\mathrm{E}[X_i] = 1/2$.
- $\mathrm{E}[X] = 50$ by linearity of expectation

Linearity of Expectation Example 2

- Let’s say we want to Bubble Sort an array *A*, where *A* is randomly permuted

```
bubbleSort(A : list of sortable items)
    n = length(A)
    do
        swapped = false
        for i = 1 to n-1 do
            if A[i-1] > A[i] then
                swap(A[i-1], A[i])
                swapped = true
            end if
        end for
    while not swapped
```

Linearity of Expectation Example 2

- Let’s say we want to Bubble Sort an array A, where A is randomly permuted
- How many swaps does Bubble Sort perform?

Linearity of Expectation Example 2

- Let's say we want to Bubble Sort an array A, where A is randomly permuted
- How many swaps does Bubble Sort perform?
- Fact: number of swaps performed by Bubble Sort is exactly the number of *inversions* in A (pairs i,j such that $i < j$ but $A[i] > A[j]$)

Linearity of Expectation Example 2

- Let's say we want to Bubble Sort an array A, where A is randomly permuted
- How many swaps does Bubble Sort perform?
- Fact: number of swaps performed by Bubble Sort is exactly the number of *inversions* in A (pairs i,j such that $i < j$ but $A[i] > A[j]$)
    - Two line proof: each time bubble sort swaps, the number of inversions goes down by exactly one. Doesn't stop swapping until sorted (there are 0 remaining inversions)

Linearity of Expectation Example 2

- Let's say we want to Bubble Sort an array A, where A is randomly permuted
- How many swaps does Bubble Sort perform?
- Fact: number of swaps performed by Bubble Sort is exactly the number of *inversions* in A (pairs i,j such that $i < j$ but $A[i] > A[j]$)
  - Two line proof: each time bubble sort swaps, the number of inversions goes down by exactly one. Doesn't stop swapping until sorted (there are 0 remaining inversions)
- Rephrasing: how many inversions are there in a randomly-permuted array?

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?
- Let $X_{ij} = 1$ if i, j are an inversion; 0 otherwise

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?
- Let $X_{ij} = 1$ if i, j are an inversion; 0 otherwise
- Are the X_{ij} independent?

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is E[X]?
- Let $X_{ij} = 1$ if i, j are an inversion; 0 otherwise
- Are the X_{ij} independent?
    - No! If a, b is an inversion, and b, c is an inversion, then a, c is an inversion.

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?
- Let $X_{ij} = 1$ if i, j are an inversion; 0 otherwise
- Are the X_{ij} independent?
    - No! If a, b is an inversion, and b, c is an inversion, then a, c is an inversion.
    - Reason: know $a < b < c$, but $A[a] > A[b]$ and $A[b] > A[c]$, so $A[a] > A[c]$.

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?

- Let $X_{ij} = 1$ if i, j represent an inversion; 0 otherwise

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?
- Let $X_{ij} = 1$ if i, j represent an inversion; 0 otherwise
- $X = \sum_{i,j} X_{ij}$

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?
- Let $X_{ij} = 1$ if i, j represent an inversion; 0 otherwise
- $X = \sum_{i,j} X_{ij}$
- $\mathrm{E}[X] = \sum_{i,j} \mathrm{E}[X_{ij}]$

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?
- Let $X_{ij} = 1$ if i, j represent an inversion; 0 otherwise
- $X = \sum_{i,j} X_{ij}$
- $\mathrm{E}[X] = \sum_{i,j} \mathrm{E}[X_{ij}]$
- $\mathrm{E}[X_{ij}] = 1/2$

Linearity of Expectation Example 2

- $X =$ number of inversions in A. What is $\mathrm{E}[X]$?
- Let $X_{ij} = 1$ if i, j represent an inversion; 0 otherwise
- $X = \sum_{i,j} X_{ij}$
- $\mathrm{E}[X] = \sum_{i,j} \mathrm{E}[X_{ij}]$
- $\mathrm{E}[X_{ij}] = 1/2$
- $\mathrm{E}[X] = n(n-1)/4$

Cuckoo Hashing

A randomized algorithm

- Before we finish talking about probability, let’s look at an example of a randomized algorithm

A randomized algorithm

- Before we finish talking about probability, let’s look at an example of a randomized algorithm
- Hashing: way to implement a dictionary with constant-time insert, delete, lookup

A randomized algorithm

- Before we finish talking about probability, let's look at an example of a randomized algorithm
- Hashing: way to implement a dictionary with constant-time insert, delete, lookup
- Hashing is randomized, so performance can be bad sometimes

A randomized algorithm

- Before we finish talking about probability, let’s look at an example of a randomized algorithm
- Hashing: way to implement a dictionary with constant-time insert, delete, lookup
- Hashing is randomized, so performance can be bad sometimes
- Cuckoo *hashing*: $O(1)$ worst-case lookup. (Inserts are usually constant-time, but can be expensive sometimes.)

Cuckoo *filters* vs Cuckoo *hashing*

- Cuckoo hashing is used to store a hash table (an alternative to e.g. chaining)

Cuckoo *filters* vs Cuckoo *hashing*

- Cuckoo hashing is used to store a hash table (an alternative to e.g. chaining)
- Cuckoo filters store an approximate representation of a set

Cuckoo *filters* vs Cuckoo *hashing*

- Cuckoo hashing is used to store a hash table (an alternative to e.g. chaining)

- Cuckoo filters store an approximate representation of a set

- They use the same underlying ideas!

Cuckoo *filters* vs Cuckoo *hashing*

- Cuckoo hashing is used to store a hash table (an alternative to e.g. chaining)
- Cuckoo filters store an approximate representation of a set
- They use the same underlying ideas!
- Cuckoo hashing is widely used and highly relevant to the course anyway

Reminder: Dictionary

- You’ve seen in 136 and/or 256

Reminder: Dictionary

- You’ve seen in 136 and/or 256
- Idea: want to store *n* key/value pairs

Reminder: Dictionary

- You’ve seen in 136 and/or 256
- Idea: want to store n key/value pairs
- Can insert new key/value pair

Reminder: Dictionary

- You’ve seen in 136 and/or 256
- Idea: want to store n key/value pairs
- Can insert new key/value pair
- Query: given a key, get the associated value stored in the dictionary

Reminder: Dictionary

- You’ve seen in 136 and/or 256
- Idea: want to store *n* key/value pairs
- Can insert new key/value pair
- Query: given a key, get the associated value stored in the dictionary
- How fast can we do inserts and queries? How much space do we need?

Reminder: Dictionary

- You’ve seen in 136 and/or 256
- Idea: want to store n key/value pairs
- Can insert new key/value pair
- Query: given a key, get the associated value stored in the dictionary
- How fast can we do inserts and queries? How much space do we need?
    - Can get $O(1)$ expected time for both operations using $O(n)$ space.

Assumptions on hash

- Let's assume we have access to a *uniform random* hash h that hashes any item to a value in $\{0, \ldots, M\}$.

Assumptions on hash

- Let's assume we have access to a *uniform random* hash h that hashes any item to a value in $\{0, \ldots, M\}$.
- For any x and any $i \in \{0, \ldots, M\}$, $\Pr(h(x) = i) = 1/M$

Assumptions on hash

- Let's assume we have access to a *uniform random* hash h that hashes any item to a value in $\{0, \ldots, M\}$.

- For any x and any $i \in \{0, \ldots, M\}$, $\Pr(h(x) = i) = 1/M$

- We assume h is independent: so even if we know that $h(y) = a$ and $h(z) = b$ (and so on), then we still have "For any x and any $i \in \{0, \ldots, M\}$, $\Pr(h(x) = i) = 1/M$"

Assumptions on hash

- Let's assume we have access to a *uniform random* hash h that hashes any item to a value in $\{0, \ldots, M\}$.
- For any x and any $i \in \{0, \ldots, M\}$, $\Pr(h(x) = i) = 1/M$
- We assume h is independent: so even if we know that $h(y) = a$ and $h(z) = b$ (and so on), then we still have "For any x and any $i \in \{0, \ldots, M\}$, $\Pr(h(x) = i) = 1/M$"
- Assume that M is much bigger than the number of items in our dataset. (Like $M = 2^{64}$)

Building a Dictionary (doesn't quite work yet)

To store n items:

- Allocate an array A with cn slots for some c. Each slot must be large enough to store a key/value pair
    - Hash tables have cn slots for the rest of today

Building a Dictionary (doesn't quite work yet)

To store n items:

- Allocate an array A with cn slots for some c. Each slot must be large enough to store a key/value pair
  - Hash tables have cn slots for the rest of today
- Insert x: store item x at position $h(x)$ % cn

Building a Dictionary (doesn't quite work yet)

To store n items:

- Allocate an array A with cn slots for some c. Each slot must be large enough to store a key/value pair
    - Hash tables have cn slots for the rest of today
- Insert x: store item x at position $h(x)$ % cn
- Query q: look at position $h(q)$ % cn and see if the key is stored there

Building a Dictionary (doesn’t quite work yet)

To store n items:

- Allocate an array A with cn slots for some c. Each slot must be large enough to store a key/value pair
    - Hash tables have cn slots for the rest of today
- Insert x: store item x at position $h(x)$ % cn
- Query q: look at position $h(q)$ % cn and see if the key is stored there
- If we can do this: $O(1)$ worst-case query, insert; always correct.

Building a Dictionary (doesn't quite work yet)

To store n items:

- Allocate an array A with cn slots for some c. Each slot must be large enough to store a key/value pair
  - Hash tables have cn slots for the rest of today
- Insert x: store item x at position $h(x)$ % cn
- Query q: look at position $h(q)$ % cn and see if the key is stored there
- If we can do this: $O(1)$ worst-case query, insert; always correct.
- What's the problem with this approach?

Collisions

- Several items might hash to the same location. How can we resolve this?

Collisions

- Several items might hash to the same location. How can we resolve this?
  - Chaining

Collisions

- Several items might hash to the same location. How can we resolve this?
  - Chaining
  - Linear Probing

Chaining

- Each entry in our array A is the head of a new data structure

Chaining

- Each entry in our array A is the head of a new data structure
- Often implemented as a singly-linked list (let’s draw this on the board)

Chaining

- Each entry in our array *A* is the head of a new data structure
- Often implemented as a singly-linked list (let’s draw this on the board)
- Insert: add item to linked list

Chaining

- Each entry in our array *A* is the head of a new data structure
- Often implemented as a singly-linked list (let's draw this on the board)
- Insert: add item to linked list
- Query: find item in linked list

Chaining

- Each entry in our array A is the head of a new data structure
- Often implemented as a singly-linked list (let’s draw this on the board)
- Insert: add item to linked list
- Query: find item in linked list
- Advantages?

Chaining

- Each entry in our array *A* is the head of a new data structure
- Often implemented as a singly-linked list (let’s draw this on the board)
- Insert: add item to linked list
- Query: find item in linked list
- Advantages?
    - Space-efficient (just need pointers for linked list)

Chaining

- Each entry in our array A is the head of a new data structure
- Often implemented as a singly-linked list (let’s draw this on the board)
- Insert: add item to linked list
- Query: find item in linked list
- Advantages?
  - Space-efficient (just need pointers for linked list)
  - Simple. What is the insert and query time?

Chaining

- Each entry in our array A is the head of a new data structure
- Often implemented as a singly-linked list (let's draw this on the board)
- Insert: add item to linked list
- Query: find item in linked list
- Advantages?
    - Space-efficient (just need pointers for linked list)
    - Simple. What is the insert and query time?
    - Good *worst-case* insert time of $O(1)$; we'll analyze query time in a moment

Chaining

- Each entry in our array A is the head of a new data structure
- Often implemented as a singly-linked list (let's draw this on the board)
- Insert: add item to linked list
- Query: find item in linked list
- Advantages?
  - Space-efficient (just need pointers for linked list)
  - Simple. What is the insert and query time?
  - Good *worst-case* insert time of $O(1)$; we'll analyze query time in a moment
- Disadvantages?

Chaining

- Each entry in our array A is the head of a new data structure
- Often implemented as a singly-linked list (let's draw this on the board)
- Insert: add item to linked list
- Query: find item in linked list
- Advantages?
  - Space-efficient (just need pointers for linked list)
  - Simple. What is the insert and query time?
  - Good *worst-case* insert time of $O(1)$; we'll analyze query time in a moment
- Disadvantages?
  - Cache inefficient

Linearity of Expectation Example 3: Chaining

- Chaining running time: hash in $O(1)$. May need to traverse other elements in the chain. Finally, need to compare the query if it exists in the bucket, in $O(1)$ time.

Linearity of Expectation Example 3: Chaining

- Chaining running time: hash in $O(1)$. May need to traverse other elements in the chain. Finally, need to compare the query if it exists in the bucket, in $O(1)$ time.
- What's the expected number of *non-query elements* in a given chain?

Linearity of Expectation Example 3: Chaining

- Chaining running time: hash in $O(1)$. May need to traverse other elements in the chain. Finally, need to compare the query if it exists in the bucket, in $O(1)$ time.
- What's the expected number of *non-query elements* in a given chain?
- $X^j =$ number of non-query items in chain j

Linearity of Expectation Example 3: Chaining

- Chaining running time: hash in $O(1)$. May need to traverse other elements in the chain. Finally, need to compare the query if it exists in the bucket, in $O(1)$ time.
- What's the expected number of *non-query elements* in a given chain?
- $X^j =$ number of non-query items in chain j
- $X_i^j = 1$ if the ith item hashes to slot j

Linearity of Expectation Example 3: Chaining

- Chaining running time: hash in $O(1)$. May need to traverse other elements in the chain. Finally, need to compare the query if it exists in the bucket, in $O(1)$ time.
- What's the expected number of *non-query elements* in a given chain?
- X^j = number of non-query items in chain j
- $X_i^j = 1$ if the ith item hashes to slot j
- $\mathrm{E}[X_i^j] = 1/cn$

Linearity of Expectation Example 3: Chaining

- Chaining running time: hash in $O(1)$. May need to traverse other elements in the chain. Finally, need to compare the query if it exists in the bucket, in $O(1)$ time.
- What's the expected number of *non-query elements* in a given chain?
- X^j = number of non-query items in chain j
- $X_i^j = 1$ if the ith item hashes to slot j
- $\mathrm{E}[X_i^j] = 1/cn$
- $\mathrm{E}[X^j] = \sum_{i=1}^{n} \mathrm{E}[X_i^j] = 1/c$

Linearity of Expectation Example 3: Chaining

- Chaining running time: hash in $O(1)$. May need to traverse other elements in the chain. Finally, need to compare the query if it exists in the bucket, in $O(1)$ time.
- What's the expected number of *non-query elements* in a given chain?
- X^j = number of non-query items in chain j
- $X_i^j = 1$ if the ith item hashes to slot j
- $\mathrm{E}[X_i^j] = 1/cn$
- $\mathrm{E}[X^j] = \sum_{i=1}^{n} \mathrm{E}[X_i^j] = 1/c$
- So the expected length of the chain is $O(1 + 1/c) = O(1)$

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)
- Insert: attempt to insert x into $h(x)$ % cn. If slot is full, keep moving down the table to find the next empty slot.

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)
- Insert: attempt to insert x into $h(x)$ % cn. If slot is full, keep moving down the table to find the next empty slot.
- Query: start at $h(x)$ % cn. Need to keep checking until find the item, or find an empty slot

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)
- Insert: attempt to insert x into $h(x)$ % cn. If slot is full, keep moving down the table to find the next empty slot.
- Query: start at $h(x)$ % cn. Need to keep checking until find the item, or find an empty slot
- Let's look at this on the board

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)
- Insert: attempt to insert x into $h(x)$ % cn. If slot is full, keep moving down the table to find the next empty slot.
- Query: start at $h(x)$ % cn. Need to keep checking until find the item, or find an empty slot
- Let's look at this on the board
- Advantages?

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)
- Insert: attempt to insert x into $h(x)$ % cn. If slot is full, keep moving down the table to find the next empty slot.
- Query: start at $h(x)$ % cn. Need to keep checking until find the item, or find an empty slot
- Let's look at this on the board
- Advantages?
  - Somewhat space-efficient

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)
- Insert: attempt to insert x into $h(x) \% cn$. If slot is full, keep moving down the table to find the next empty slot.
- Query: start at $h(x) \% cn$. Need to keep checking until find the item, or find an empty slot
- Let's look at this on the board
- Advantages?
  - Somewhat space-efficient
  - Insert is $O(1 + \frac{1}{1-1/c}) = O(1)$ and query is $O(1 + \frac{1}{(1-1/c)}) = O(1)$ in expectation (Knuth's classic result; nontrivial)

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)
- Insert: attempt to insert x into $h(x)\ \%\ cn$. If slot is full, keep moving down the table to find the next empty slot.
- Query: start at $h(x)\ \%\ cn$. Need to keep checking until find the item, or find an empty slot
- Let's look at this on the board
- Advantages?
  - Somewhat space-efficient
  - Insert is $O(1 + \frac{1}{1-1/c}) = O(1)$ and query is $O(1 + \frac{1}{(1-1/c)}) = O(1)$ in expectation (Knuth's classic result; nontrivial)
  - Cache-efficient!

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)
- Insert: attempt to insert x into $h(x)$ % cn. If slot is full, keep moving down the table to find the next empty slot.
- Query: start at $h(x)$ % cn. Need to keep checking until find the item, or find an empty slot
- Let's look at this on the board
- Advantages?
    - Somewhat space-efficient
    - Insert is $O(1 + \frac{1}{1-1/c}) = O(1)$ and query is $O(1 + \frac{1}{(1-1/c)}) = O(1)$ in expectation (Knuth's classic result; nontrivial)
    - Cache-efficient!
- Disadvantages?

Linear Probing

- Set $c > 1$ (often have $1.5n$ or $2n$ slots in practice)
- Insert: attempt to insert x into $h(x)$ % cn. If slot is full, keep moving down the table to find the next empty slot.
- Query: start at $h(x)$ % cn. Need to keep checking until find the item, or find an empty slot
- Let's look at this on the board
- Advantages?
  - Somewhat space-efficient
  - Insert is $O(1 + \frac{1}{1-1/c}) = O(1)$ and query is $O(1 + \frac{1}{(1-1/c)}) = O(1)$ in expectation (Knuth's classic result; nontrivial)
  - Cache-efficient!
- Disadvantages?
  - Not that efficient; performance is terrible if table fills up

Cuckoo Hashing [Pagh, Rodler 2005]

- A third method of resolving collisions

Cuckoo Hashing [Pagh, Rodler 2005]

- A third method of resolving collisions
- Queries are $O(1)$ *worst case*

Cuckoo Hashing [Pagh, Rodler 2005]

- A third method of resolving collisions
- Queries are $O(1)$ *worst case*
- Insert will still be $O(1)$ in expectation

Cuckoo Hashing [Pagh, Rodler 2005]

- A third method of resolving collisions
- Queries are $O(1)$ *worst case*
- Insert will still be $O(1)$ in expectation
- Comparison to linear probing and chaining?

Cuckoo Hashing [Pagh, Rodler 2005]

- A third method of resolving collisions
- Queries are $O(1)$ *worst case*
- Insert will still be $O(1)$ in expectation
- Comparison to linear probing and chaining?
    - Chaining: $O(1)$ worst case inserts; $O(1)$ expected queries. Not as good for query-heavy workloads!

Cuckoo Hashing [Pagh, Rodler 2005]

- A third method of resolving collisions
- Queries are $O(1)$ *worst case*
- Insert will still be $O(1)$ in expectation
- Comparison to linear probing and chaining?
    - Chaining: $O(1)$ worst case inserts; $O(1)$ expected queries. Not as good for query-heavy workloads!
    - Linear probing: more cache-efficient, but both inserts and queries are only $O(1)$ on average

Cuckoo Hashing Invariant

- Have two hash functions h_1, h_2. (*No partial key/XOR stuff!* That's only for the filters. We'll come back to why.)

Cuckoo Hashing Invariant

- Have two hash functions h_1, h_2. (*No partial key/XOR stuff!* That's only for the filters. We'll come back to why.)
- Table of size cn with $c = 2$ (for now)

Cuckoo Hashing Invariant

- Have two hash functions h_1, h_2. (*No partial key/XOR stuff!* That's only for the filters. We'll come back to why.)
- Table of size cn with $c = 2$ (for now)
- Invariant: item x is either stored at $h_1(x)\ \%\ cn$, or at slot $h_2(x)\ \%\ cn$.

Cuckoo Hashing Invariant

- Have two hash functions h_1, h_2. (*No partial key/XOR stuff!* That's only for the filters. We'll come back to why.)
- Table of size cn with $c = 2$ (for now)
- Invariant: item x is either stored at $h_1(x)$ % cn, or at slot $h_2(x)$ % cn.
- We'll come back to inserts. But how can we query? How much time does a query take?

Cuckoo Hashing Invariant

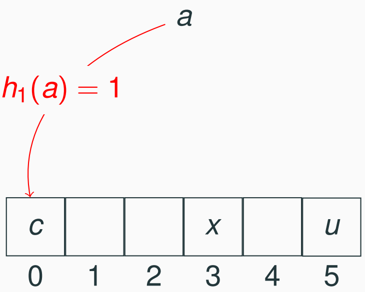
- Have two hash functions h_1, h_2. (*No partial key/XOR stuff!* That's only for the filters. We'll come back to why.)
- Table of size cn with $c = 2$ (for now)
- Invariant: item x is either stored at $h_1(x)\ \%\ cn$, or at slot $h_2(x)\ \%\ cn$.
- We'll come back to inserts. But how can we query? How much time does a query take?
  - Check both hash slots. Immediately get $O(1)$ time

Cuckoo Hashing Inserts

c			x		u
0	1	2	3	4	5

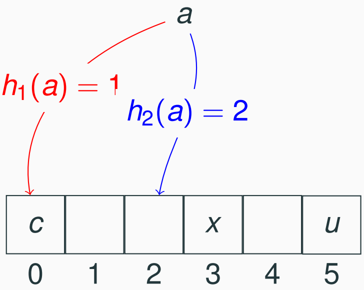
- Let’s say we want to insert a new item *a*. How can we do that?

Cuckoo Hashing Inserts



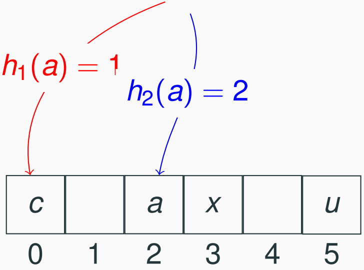
- Let's say we want to insert a new item a. How can we do that?

Cuckoo Hashing Inserts



- Let's say we want to insert a new item a. How can we do that?
- Easy case: if $h_1(a)$ % cn or $h_2(a)$ % cn is free, can just store a immediately.

Cuckoo Hashing Inserts



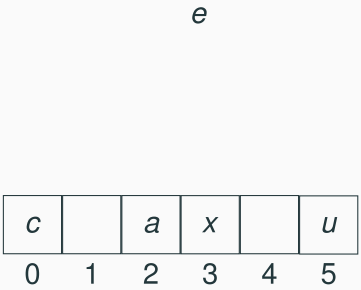
- Let's say we want to insert a new item a. How can we do that?
- Easy case: if $h_1(a)$ % cn or $h_2(a)$ % cn is free, can just store a immediately.

Cuckoo Hashing Inserts

c		a	x		u
0	1	2	3	4	5

- Let’s say we want to insert a new item a. How can we do that?
- Easy case: if $h_1(a)$ % cn or $h_2(a)$ % cn is free, can just store a immediately.

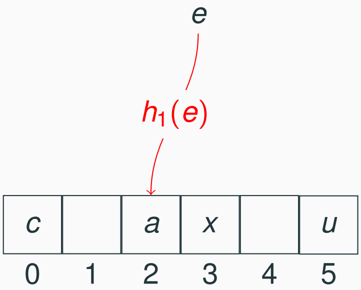
Cuckoo Hashing Inserts



0	1	2	3	4	5
c		a	x		u

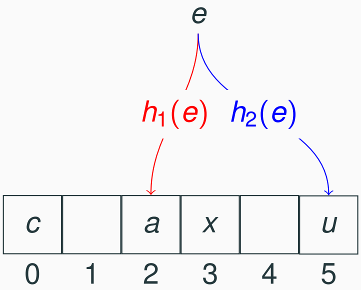
- Let’s say we want to insert a new item a. How can we do that?
- Easy case: if $h_1(a)$ % cn or $h_2(a)$ % cn is free, can just store a immediately.
- What do we do if both are full?

Cuckoo Hashing Inserts



- Let’s say we want to insert a new item a. How can we do that?
- Easy case: if $h_1(a)$ % cn or $h_2(a)$ % cn is free, can just store a immediately.
- What do we do if both are full?

Cuckoo Hashing Inserts

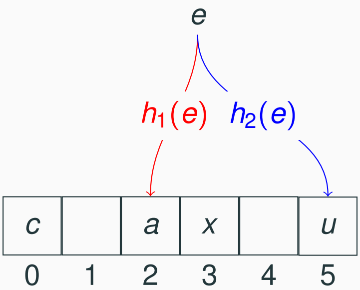


- Let's say we want to insert a new item *a*. How can we do that?
- Easy case: if $h_1(a)$ % cn or $h_2(a)$ % cn is free, can just store a immediately.
- What do we do if both are full?

Cuckooing!

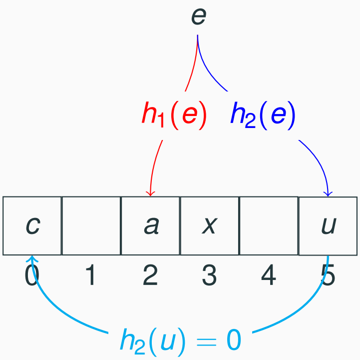
Cuckoos kick other birds' eggs out of the nest, replacing them with their own.

Cuckoo Hashing Inserts



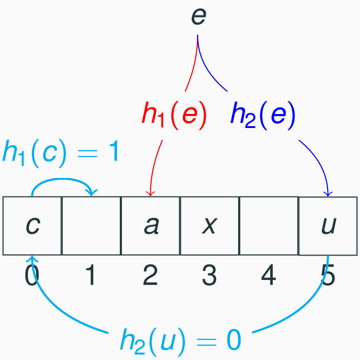
- Let's say we want to insert a new item *a*. How can we do that?
- Easy case: if $h_1(a)$ % cn or $h_2(a)$ % cn is free, can just store a immediately.
- What do we do if both are full?
- Move one of the items in the way to its other slot!
- If there's an item THERE, recurse

Cuckoo Hashing Inserts



- Let's say we want to insert a new item *a*. How can we do that?
- Easy case: if $h_1(a)$ % cn or $h_2(a)$ % cn is free, can just store *a* immediately.
- What do we do if both are full?
- Move one of the items in the way to its other slot!
- If there's an item THERE, recurse

Cuckoo Hashing Inserts



- Let's say we want to insert a new item a. How can we do that?
- Easy case: if $h_1(a)$ % cn or $h_2(a)$ % cn is free, can just store a immediately.
- What do we do if both are full?
- Move one of the items in the way to its other slot!
- If there's an item THERE, recurse

Why not partial-key?

- In Cuckoo Filters, we had to do this partial key cuckoo hashing trick where we used $h_2(x) = h_1(x) \wedge h(f(x))$

Why not partial-key?

- In Cuckoo Filters, we had to do this partial key cuckoo hashing trick where we used $h_2(x) = h_1(x) \wedge h(f(x))$
- Why don't we need this here?

Why not partial-key?

- In Cuckoo Filters, we had to do this partial key cuckoo hashing trick where we used $h_2(x) = h_1(x) \wedge h(f(x))$
- Why don't we need this here?
- Answer: since we are storing a dictionary, we have access to the item x *itself*
    - In the filter we only stored $f(x)$ rather than x
- So we can just rehash it!

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x)\ \%\ cn$ or $h_2(x)\ \%\ cn$

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x) \% cn$ or $h_2(x) \% cn$
- Is there a simple example where this is impossible?

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x)$ % cn or $h_2(x)$ % cn
- Is there a simple example where this is impossible?
- One option: three items x, y, and z all have the same two hashes

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x)$ % cn or $h_2(x)$ % cn
- Is there a simple example where this is impossible?
- One option: three items x, y, and z all have the same two hashes
    - What is the probability that this exact scenario happens?

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x)$ % cn or $h_2(x)$ % cn
- Is there a simple example where this is impossible?
- One option: three items x, y, and z all have the same two hashes
  - What is the probability that this exact scenario happens?
  - There exist slots s_1 and s_2 such that all of x, y, and z all hash to one of these two slots

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x)\ \%\ cn$ or $h_2(x)\ \%\ cn$
- Is there a simple example where this is impossible?
- One option: three items x, y, and z all have the same two hashes
    - What is the probability that this exact scenario happens?
    - There exist slots s_1 and s_2 such that all of x, y, and z all hash to one of these two slots
    - For a given x, y, z, s_1, and s_2, how often does $h_1(x) = h_1(y) = h_1(z) = s_1$ and $h_2(x) = h_2(y) = h_2(z) = s_2$?

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x) \% cn$ or $h_2(x) \% cn$
- Is there a simple example where this is impossible?
- One option: three items x, y, and z all have the same two hashes
  - What is the probability that this exact scenario happens?
  - There exist slots s_1 and s_2 such that all of x, y, and z all hash to one of these two slots
  - For a given x, y, z, s_1, and s_2, how often does $h_1(x) = h_1(y) = h_1(z) = s_1$ and $h_2(x) = h_2(y) = h_2(z) = s_2$?
  - $(1/m)^6$

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x)$ % cn or $h_2(x)$ % cn
- Is there a simple example where this is impossible?
- One option: three items x, y, and z all have the same two hashes
    - What is the probability that this exact scenario happens?
    - There exist slots s_1 and s_2 such that all of x, y, and z all hash to one of these two slots
    - For a given x, y, z, s_1, and s_2, how often does $h_1(x) = h_1(y) = h_1(z) = s_1$ and $h_2(x) = h_2(y) = h_2(z) = s_2$?
    - $(1/m)^6$
    - There are $\binom{n}{3}\binom{m}{2}$ choices of x, y, z, s_1, and s_2

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x)$ % cn or $h_2(x)$ % cn
- Is there a simple example where this is impossible?
- One option: three items x, y, and z all have the same two hashes
  - What is the probability that this exact scenario happens?
  - There exist slots s_1 and s_2 such that all of x, y, and z all hash to one of these two slots
  - For a given x, y, z, s_1, and s_2, how often does $h_1(x) = h_1(y) = h_1(z) = s_1$ and $h_2(x) = h_2(y) = h_2(z) = s_2$?
  - $(1/m)^6$
  - There are $\binom{n}{3}\binom{m}{2}$ choices of x, y, z, s_1, and s_2
  - So this happens with probability $\Theta(n^3 m^2 / m^6) = \Theta(1/n)$.

Does this always work?

- Recall our invariant: every item x is stored at $h_1(x)$ % cn or $h_2(x)$ % cn
- Is there a simple example where this is impossible?
- One option: three items x, y, and z all have the same two hashes
    - What is the probability that this exact scenario happens?
    - There exist slots s_1 and s_2 such that all of x, y, and z all hash to one of these two slots
    - For a given x, y, z, s_1, and s_2, how often does $h_1(x) = h_1(y) = h_1(z) = s_1$ and $h_2(x) = h_2(y) = h_2(z) = s_2$?
    - $(1/m)^6$
    - There are $\binom{n}{3}\binom{m}{2}$ choices of x, y, z, s_1, and s_2
    - So this happens with probability $\Theta(n^3 m^2 / m^6) = \Theta(1/n)$.
    - Probability of *any* impossible configuration is $O(1/n)$ (outside the scope of the course)

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?
  - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
    - Two cache misses per query. Is that good?
    - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line
- What is the Insert performance?

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?
  - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line
- What is the Insert performance?
  - $O(1)$ in expectation if we do not encounter an infinite loop

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
    - Two cache misses per query. Is that good?
    - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line
- What is the Insert performance?
    - $O(1)$ in expectation if we do not encounter an infinite loop
    - Idea: half the slots are empty, so each time we go to a new slot, we should have a $\approx 1/2$ probability of being done

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?
  - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line
- What is the Insert performance?
  - $O(1)$ in expectation if we do not encounter an infinite loop
  - Idea: half the slots are empty, so each time we go to a new slot, we should have a $\approx 1/2$ probability of being done
  - (Analysis is nontrivial since we need to carefully avoid cases with an infinite loop)

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
    - Two cache misses per query. Is that good?

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?
  - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?
  - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line
- What is the Insert performance?

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?
  - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line
- What is the Insert performance?
- Cache performance?

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?
  - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line
- What is the Insert performance?
- Cache performance?
  - One cache miss per “cuckoo”–OK but not great

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?
  - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line
- What is the Insert performance?
- Cache performance?
  - One cache miss per “cuckoo”–OK but not great
  - In practice, inserts are really pretty bad for cuckoo hashing due to poor constants

Cuckoo Hashing Performance

- Queries: $O(1)$ worst case
- Cache performance?
  - Two cache misses per query. Is that good?
  - Kind of! Probably better than chaining. But linear probing has only $\approx$one cache miss on any query, so long as $\log n$ items fit in a cache line
- What is the Insert performance?
- Cache performance?
  - One cache miss per “cuckoo”–OK but not great
  - In practice, inserts are really pretty bad for cuckoo hashing due to poor constants
- Idea: cuckoo hashing does great on queries (though with potentially worse cache efficiency than linear probing), but pays for it with expensive inserts

Limits of Expectation

Limits of Expectation

- Let’s say I charge you $1000 to play a game. With probability 1 in 1 million, I give you $10 billion. Otherwise, I give you $0.

Limits of Expectation

- Let’s say I charge you $1000 to play a game. With probability 1 in 1 million, I give you $10 billion. Otherwise, I give you $0.

- Would you play this game?

Limits of Expectation

- Let’s say I charge you $1000 to play a game. With probability 1 in 1 million, I give you $10 billion. Otherwise, I give you $0.
- Would you play this game?
- Answer: maybe, but probably not. You’re just going to lose $1000.
- But expectation is good! You expect to win $9000.

Concentration bounds

- Rather than giving the average performance, bound the probability of *bad* performance.

Concentration bounds

- Rather than giving the average performance, bound the probability of *bad* performance.
- Let's say I flip a coin k times. On average, I see $k/2$ heads. But what is the probability I *never* see a heads?

Concentration bounds

- Rather than giving the average performance, bound the probability of *bad* performance.
- Let's say I flip a coin k times. On average, I see $k/2$ heads. But what is the probability I *never* see a heads?
- Answer: $1/2^k$

Concentration bounds

- Rather than giving the average performance, bound the probability of *bad* performance.
- Let's say I flip a coin k times. On average, I see $k/2$ heads. But what is the probability I *never* see a heads?
- Answer: $1/2^k$
- Quicksort has expected runtime $O(n \log n)$. What is the probability that the running time is more than $O(n \log n)$?

Concentration bounds

- Rather than giving the average performance, bound the probability of *bad* performance.
- Let's say I flip a coin k times. On average, I see $k/2$ heads. But what is the probability I *never* see a heads?
- Answer: $1/2^k$
- Quicksort has expected runtime $O(n \log n)$. What is the probability that the running time is more than $O(n \log n)$?
- Answer: $O(1/n)$ (this is why quicksort is not worse than merge sort: you'll never see the worst case in your life if n is at all large)

With High Probability

- An event happens *with high probability* (with respect to n) if it happens with probability $1 - O(1/n)$

With High Probability

- An event happens *with high probability* (with respect to n) if it happens with probability $1 - O(1/n)$
- So: quicksort is $O(n \log n)$ with high probability

With High Probability

- An event happens *with high probability* (with respect to n) if it happens with probability $1 - O(1/n)$
- So: quicksort is $O(n \log n)$ with high probability
- Cuckoo hashing can maintain its invariant with high probability

With High Probability

- An event happens *with high probability* (with respect to n) if it happens with probability $1 - O(1/n)$
- So: quicksort is $O(n \log n)$ with high probability
- Cuckoo hashing can maintain its invariant with high probability
- Cuckoo hashing inserts require $O(\log n)$ swaps with high probability

With High Probability

- An event happens *with high probability* (with respect to n) if it happens with probability $1 - O(1/n)$
- So: quicksort is $O(n \log n)$ with high probability
- Cuckoo hashing can maintain its invariant with high probability
- Cuckoo hashing inserts require $O(\log n)$ swaps with high probability
- Linear probing queries require $O(\log n)$ time with high probability. (Contrast to $O(1)$ in expectation!)

With High Probability

- An event happens *with high probability* (with respect to n) if it happens with probability $1 - O(1/n)$
- So: quicksort is $O(n \log n)$ with high probability
- Cuckoo hashing can maintain its invariant with high probability
- Cuckoo hashing inserts require $O(\log n)$ swaps with high probability
- Linear probing queries require $O(\log n)$ time with high probability. (Contrast to $O(1)$ in expectation!)
- With high probability is always with respect to a variable. Assume that it's with respect to n unless stated otherwise.

WHP example

- How many coins do I need to flip before I see a heads with high probability? (With respect to some variable n)

WHP example

- How many coins do I need to flip before I see a heads with high probability? (With respect to some variable n)
- If I flip k times, I see a heads with probability $1 - 1/2^k$.

WHP example

- How many coins do I need to flip before I see a heads with high probability? (With respect to some variable n)
- If I flip k times, I see a heads with probability $1 - 1/2^k$.
- So I need $1/2^k = O(1/n)$. Solving, $k = \Theta(\log n)$.

Expectation vs Concentration (WHP)

- We’ll usually use “with high probability” for concentration bounds

Expectation vs Concentration (WHP)

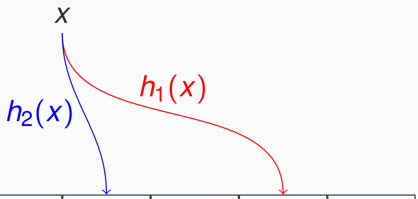
- We’ll usually use “with high probability” for concentration bounds
- Expectation states how well the algorithm does on average. Could be much better or worse sometimes!

Expectation vs Concentration (WHP)

- We’ll usually use “with high probability” for concentration bounds
- Expectation states how well the algorithm does on average. Could be much better or worse sometimes!
- With high probability gives a guarantee that will almost always be met: if n is large it becomes vanishingly unlikely that the bound will be violated.

Quick Cuckoo Filters Review

Cuckoo Filter

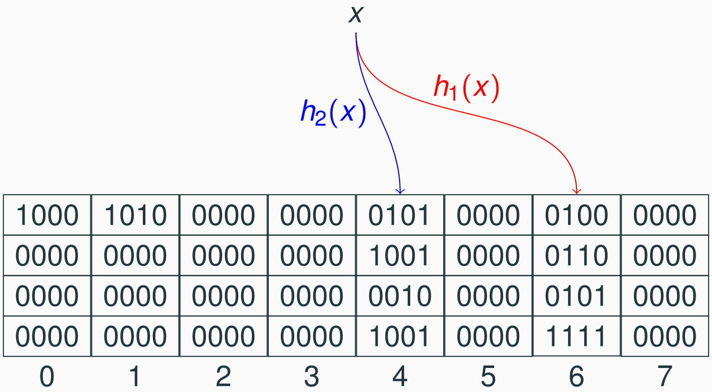


1000	1010	0000	0000	0101	0000	0100	0000
0000	0000	0000	0000	1001	0000	0110	0000
0000	0000	0000	0000	0010	0000	0101	0000
0000	0000	0000	0000	1001	0000	1111	0000
0	1	2	3	4	5	6	7

A cuckoo filter with fingerprints of length 4, $k = 2$, and 4 slots per bin.

Reminder: how do we calculate $h_2(x)$?

Cuckoo Filter

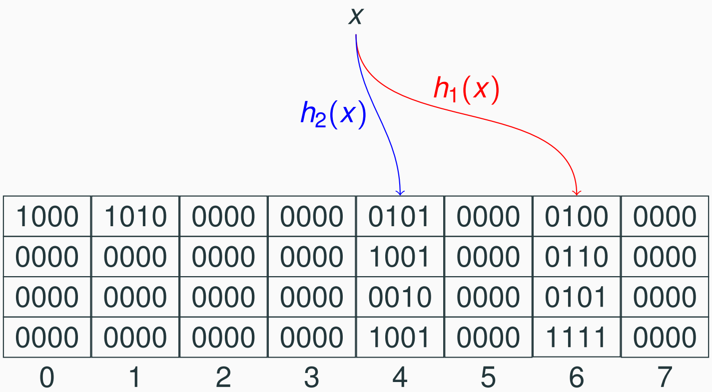


1000	1010	0000	0000	0101	0000	0100	0000
0000	0000	0000	0000	1001	0000	0110	0000
0000	0000	0000	0000	0010	0000	0101	0000
0000	0000	0000	0000	1001	0000	1111	0000
0	1	2	3	4	5	6	7

A cuckoo filter with fingerprints of length 4, $k = 2$, and 4 slots per bin.

Reminder: what happens if all slots store a value > 0?

Cuckoo Filter

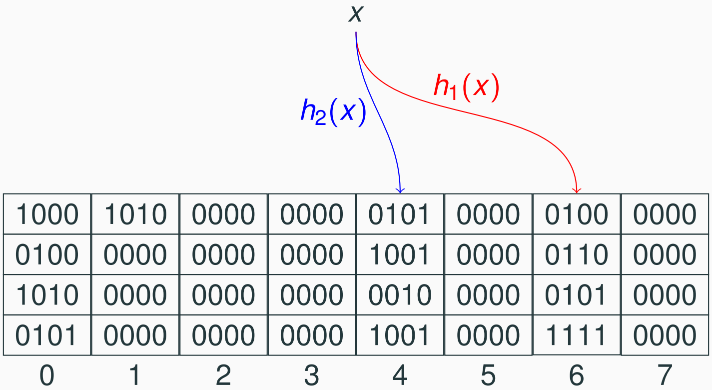


1000	1010	0000	0000	0101	0000	0100	0000
0000	0000	0000	0000	1001	0000	0110	0000
0000	0000	0000	0000	0010	0000	0101	0000
0000	0000	0000	0000	1001	0000	1111	0000
0	1	2	3	4	5	6	7

A cuckoo filter with fingerprints of length 4, $k = 2$, and 4 slots per bin.

Please note: when choosing what slot to cuckoo out of, you cannot always use the same slot! Easy solution: increment every time.

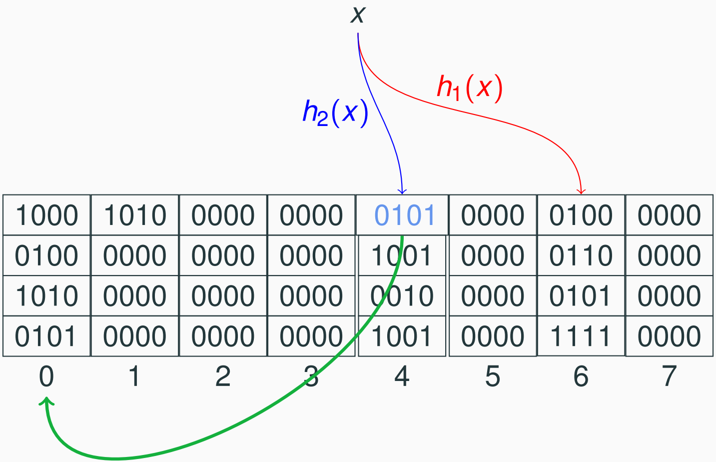
Cuckoo Filter



0	1	2	3	4	5	6	7
1000	1010	0000	0000	0101	0000	0100	0000
0100	0000	0000	0000	1001	0000	0110	0000
1010	0000	0000	0000	0010	0000	0101	0000
0101	0000	0000	0000	1001	0000	1111	0000

Remember to rotate what slot in a bit you cuckoo out of.

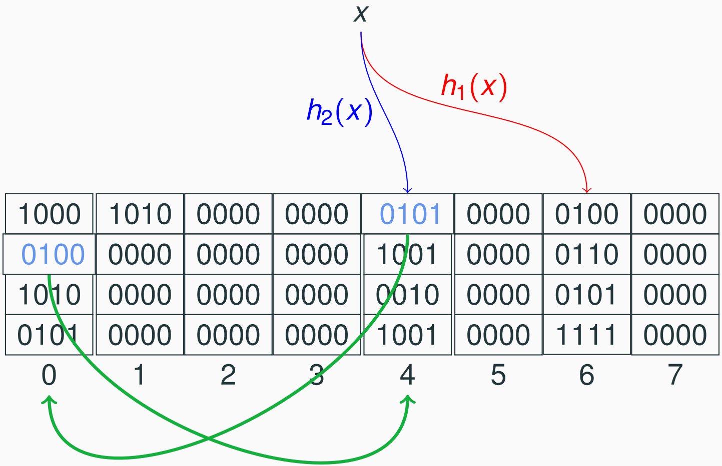
Cuckoo Filter



0	1	2	3	4	5	6	7
1000	1010	0000	0000	0101	0000	0100	0000
0100	0000	0000	0000	1001	0000	0110	0000
1010	0000	0000	0000	0010	0000	0101	0000
0101	0000	0000	0000	1001	0000	1111	0000

Remember to rotate what slot in a bit you cuckoo out of.

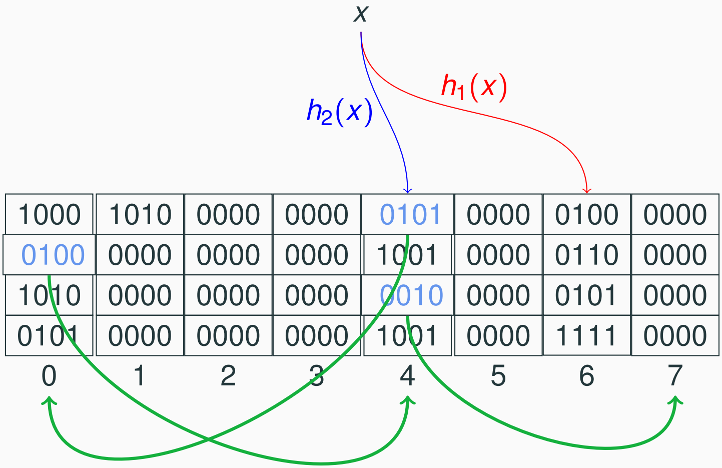
Cuckoo Filter



1000	1010	0000	0000	0101	0000	0100	0000
0100	0000	0000	0000	1001	0000	0110	0000
1010	0000	0000	0000	0010	0000	0101	0000
0101	0000	0000	0000	1001	0000	1111	0000
0	1	2	3	4	5	6	7

Remember to rotate what slot in a bit you cuckoo out of.

Cuckoo Filter

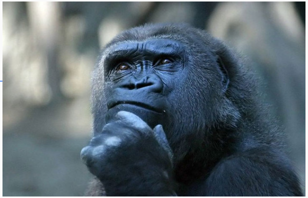


0	1	2	3	4	5	6	7
1000	1010	0000	0000	0101	0000	0100	0000
0100	0000	0000	0000	1001	0000	0110	0000
1010	0000	0000	0000	0010	0000	0101	0000
0101	0000	0000	0000	1001	0000	1111	0000

Remember to rotate what slot in a bit you cuckoo out of.

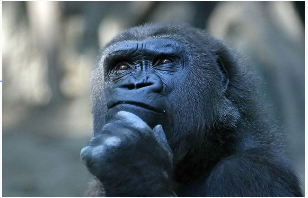
Choosing Hash Functions in Practice

Choosing a hash function



What do we want out of a hash function?

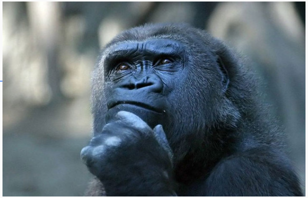
Choosing a hash function



What do we want out of a hash function?

- Quick to compute

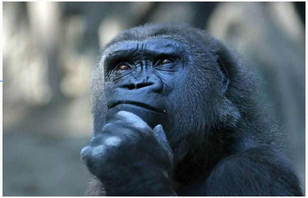
Choosing a hash function



What do we want out of a hash function?

- Quick to compute
- Small space to store the function

Choosing a hash function



What do we want out of a hash function?

- Quick to compute
- Small space to store the function
- Random enough that different elements usually don’t hash together.

Hashing in Java

- `Java.hashCode()` hashes a 64 bit `long` to a 32 bit `int`. Anyone have any ideas how?

Hashing in Java

- `Java.hashCode()` hashes a 64 bit `long` to a 32 bit `int`. Anyone have any ideas how?

- $h(x) = x \wedge (x >> 32)$ (the $\wedge$ means XOR)

Hashing in Java

- `Java.hashCode()` hashes a 64 bit `long` to a 32 bit `int`. Anyone have any ideas how?
- $h(x) = x \wedge (x >> 32)$ (the $\wedge$ means XOR)
- Is this going to work well for a filter or dictionary?

Hashing in Java

- `Java.hashCode()` hashes a 64 bit `long` to a 32 bit `int`. Anyone have any ideas how?
- $h(x) = x \wedge (x >> 32)$ (the $\wedge$ means XOR)
- Is this going to work well for a filter or dictionary?
- No: if $x < 2^{32}$ then $h(x) = x$!

Multiply-Shift Hashing

- The hash you used on Assignment 1

Multiply-Shift Hashing

- The hash you used on Assignment 1
- Is it fast?

Multiply-Shift Hashing

- The hash you used on Assignment 1
- Is it fast?
- Does it spread items out well?

Multiply-Shift Hashing

- The hash you used on Assignment 1
- Is it fast?
- Does it spread items out well?
  - Answer: works well in expectation, but bad concentration bounds!

Multiply-Shift Hashing

- The hash you used on Assignment 1
- Is it fast?
- Does it spread items out well?
    - Answer: works well in expectation, but bad concentration bounds!
    - Can show that if you hash n items to n bucket, expect 1 item per bucket

Multiply-Shift Hashing

- The hash you used on Assignment 1
- Is it fast?
- Does it spread items out well?
    - Answer: works well in expectation, but bad concentration bounds!
    - Can show that if you hash n items to n bucket, expect 1 item per bucket
    - But...good probability of getting some bucket of size $\Theta(\sqrt{n})$!

Multiply-Shift Hashing

- The hash you used on Assignment 1
- Is it fast?
- Does it spread items out well?
  - Answer: works well in expectation, but bad concentration bounds!
  - Can show that if you hash n items to n bucket, expect 1 item per bucket
  - But...good probability of getting some bucket of size $\Theta(\sqrt{n})$!
  - Big problem for cuckoo hashing/cuckoo filters/etc.

Murmurhash

- Popular hash that does a bunch of random-looking operations

Murmurhash

- Popular hash that does a bunch of random-looking operations

- No theory bounds!

Murmurhash

- Popular hash that does a bunch of random-looking operations
- No theory bounds!
- But GREAT practical performance, which is why we'll use it on Homework 3

Murmurhash

- Popular hash that does a bunch of random-looking operations
- No theory bounds!
- But GREAT practical performance, which is why we'll use it on Homework 3
- We'll look at this in detail the lecture after next.

Have a great weekend!