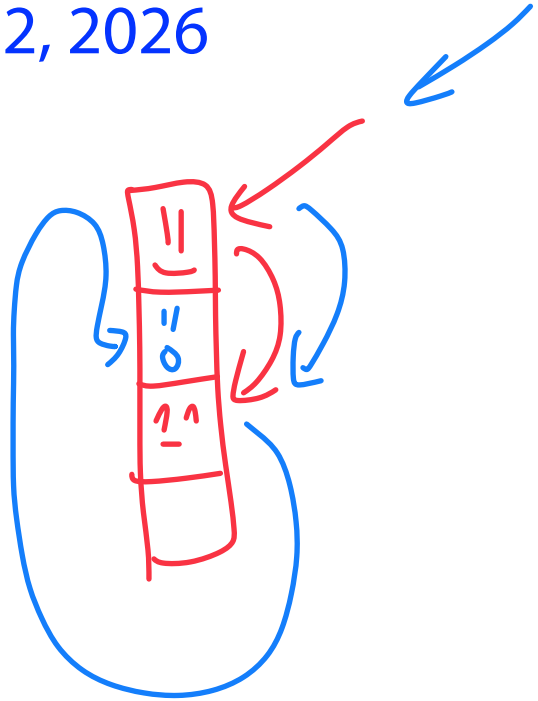
Data Structures and Algorithms

Hashing 3

CS 225

Brad Solomon

April 22, 2026



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

SIGECOM × SIGPOLICY

ecom

SIGPolicy

Predict. Trade. Win Cash.

A live prediction market experiment at UIUC. Trade on simulated events, compete for cash prizes, and contribute to economics & CS research.

NO MONEY REQUIRED **FOR RESEARCH**

$800+

In cash prizes across events & season

1st $100 | 2nd $50 | 3rd $25 | 4–5th $10

HOW IT WORKS

01 Sign up with your university email

02 Buy & sell shares on event outcomes

03 Attend live events

04 Top the leaderboard, win real cash

LIVE EVENTS

Wed, April 22 7 PM CST · ~1–2 hrs	EVENT 1
Wed, April 29 7 PM CST · ~1–2 hrs	EVENT 2
Wed, May 6 7 PM CST · ~1–2 hrs	EVENT 3

predictionslab.org

No experience needed
Free to join

Prediction Markets | Economics & CS Research | Open to All Students | No Experience Needed

Exam 5 and Retake Exam both next week

Exam 5 practice exam released this morning

If you have questions

Friday will review some core exam topics (and have a Q&A time)

Retake exam will allow you access to all retake exams. except exam 5

Only open the retake exam you wish to take!

one!

If you attempt to complete multiple exams, you may:

- Lose the opportunity for a retake exam
- Have the wrong exam counting (to your overall detriment)

- Not have time!

Learning Objectives

finish hash tables!

Review hash table implementations

Improve our closed hash implementation

Determine when and how to resize a hash table

Justify when to use different index approaches

Randomness in a hash table

Conceptually: Any single hash has a Big O of O(n) if U > m

universe

deterministic

general purpose hash function

Picking a **random** hash acts much like a **random** pivot in quicksort

Expectation!

O(1)*

Simple Uniform Hashing Assumption

Given table of size m, a simple uniform hash, h, implies

$$\forall k_1, k_2 \in U \text{ where } k_1 \neq k_2, \; Pr(h[k_1] = h[k_2]) = \frac{1}{m}$$

Uniform: All keys equally likely to hash to any position

$$Pr(h[k_1]) = \frac{1}{m}$$

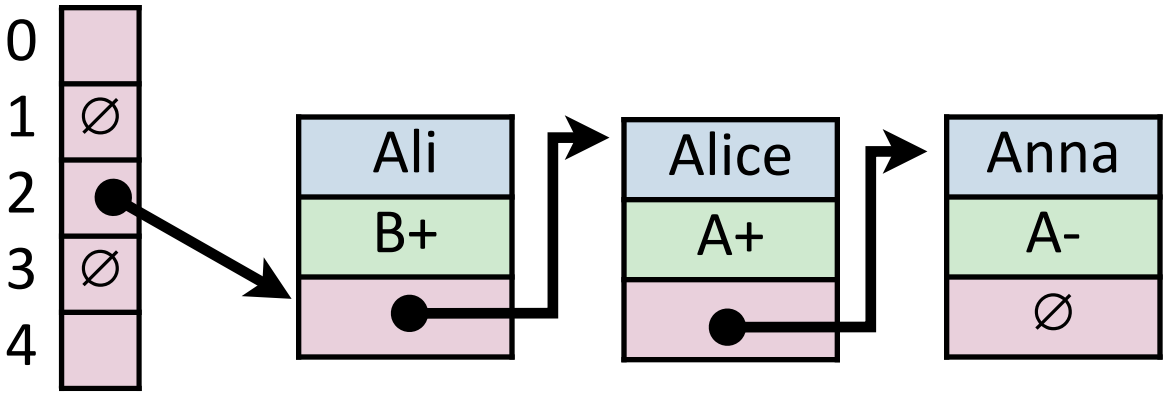
Independent: All key's hash values are independent of other keys

Today we see last * in $O(1)^{***}$

Open vs Closed Hashing

Addressing hash collisions depends on your storage structure.

- **Open Hashing:** store *k,v* pairs externally



I can store n items

$n =$ 10 million / billion / trillion

$m = n/10$ — $\alpha = \frac{n}{m}$ = my constant expected work

- **Closed Hashing:** store *k,v* pairs in the hash table

0	Anna, A-
1	
2	Ali, B+
3	Alice, A+

↖ Can store at most m items

↳ will never store m items

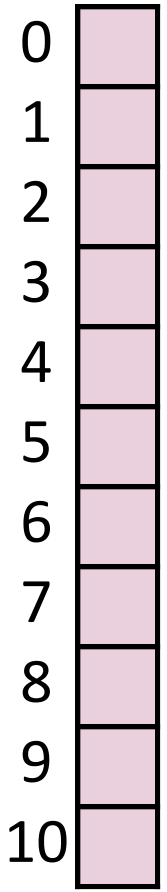
Separate Chaining Under SUHA

Under SUHA, a hash table of size *m* and *n* elements:

Find runs in: $O(1+\alpha)$.

Insert runs in: $O(1)$.

Remove runs in: $O(1+\alpha)$.



(Example of closed hashing)

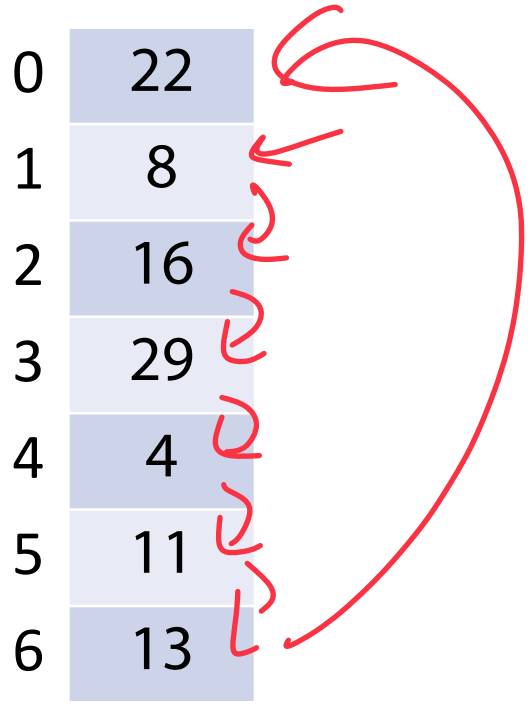
Collision Handling: Linear Probing

S = { 16, 8, 4, 13, 29, 11, 22 }

h(k) = k % 7

|S| = n

|Array| = m



h(k, i) = (k + i) % 7

Try h(k) = (k + 0) % 7, if full...

Try h(k) = (k + 1) % 7, if full...

Try h(k) = (k + 2) % 7, if full...

Try ...

Pick next available space.

(Example of closed hashing)

Collision Handling: Linear Probing

S = { 16, 8, 4, 13, 29, 11, 22 } **|S| = n**

h(k, i) = (k + i) % 7 **|Array| = m**

0	22
1	8
2	16
3	29
4	4
5	11
6	13

`_find(29)`

1) Hash the input key [h(29)=1]

2) Look at hash value (address) position

If present, return (k, v)

If not look at **next available space**

Stop when:

1) We find the object we are looking for
2) We have searched every position in the array
3) We find a blank space

(Example of closed hashing)

Collision Handling: Linear Probing

S = { 16, 8, 4, 13, 29, 11, 22 } **|S| = n**

h(k, i) = (k + i) % 7 **|Array| = m**

0	22
1	8
2	16
3	29
4	4
5	11
6	13

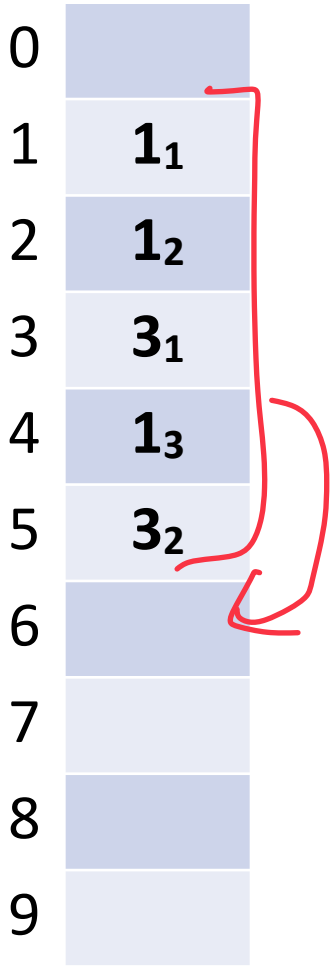
`_remove(16)`

1) Hash the input key [h(16)=2]
2) Find the actual location (if it exists)
3) Remove the (k,v) at hash value (address)

Don’t resize the array! Tombstone!

A Problem w/ Linear Probing

Primary Clustering: "Rich get richer"



Description:

Collisions create long runs of filled-in indices

Should have a **1/m** chance to hash anywhere

Instead have a **(size of cluster) / m** chance to hash at end

Remedy: Pick a better next available space!

A Problem w/ Linear Probing

Primary Clustering: "Rich get richer"

0	
1	1_1
2	1_2
3	3_1
4	1_3
5	3_2
6	
7	
8	
9	

Description:

Collisions create long runs of filled-in indices

Should have a **1/m** chance to hash anywhere

Instead have a **(size of cluster) / m** chance to hash at end

Remedy:

Pick a better "next available" position!

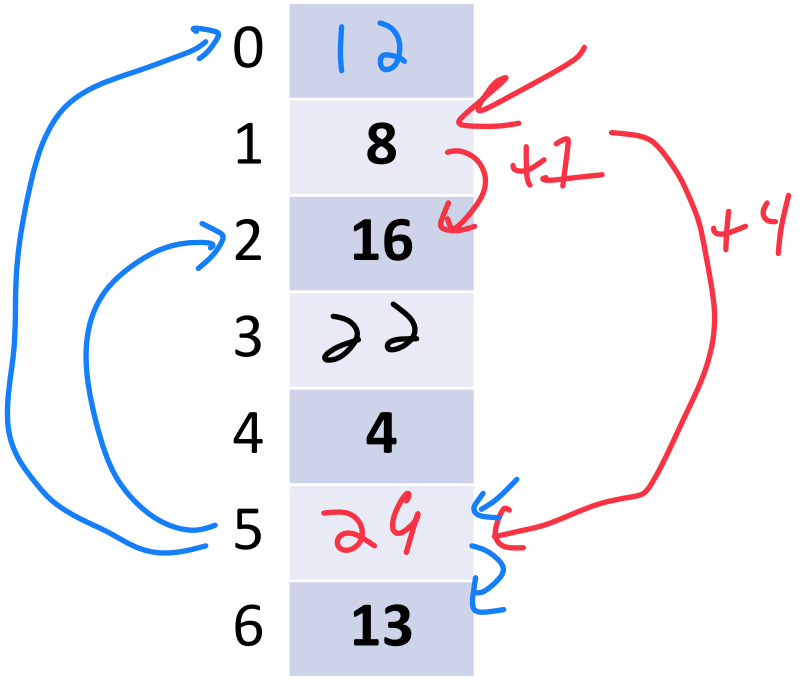
(Example of closed hashing)

Collision Handling: Quadratic Probing

S = { 16, 8, 4, 13, 29, 12, 22 } **|S| = n**

h(k) = k % 7 **|Array| = m**

29 % 7 = 1



h(k, i) = (k + i*i) % 7

Try h(k) = (k + 0) % 7, if full...

Try h(k) = (k + 1*1) % 7, if full...

Try h(k) = (k + 2*2) % 7, if full...

Try ...

9

16

25

22 % 7 = 1

1 + 1

1 + 4 = 5

1 + 9 = 10 % 7 = 3

12 % 7 = 5

5 + 1

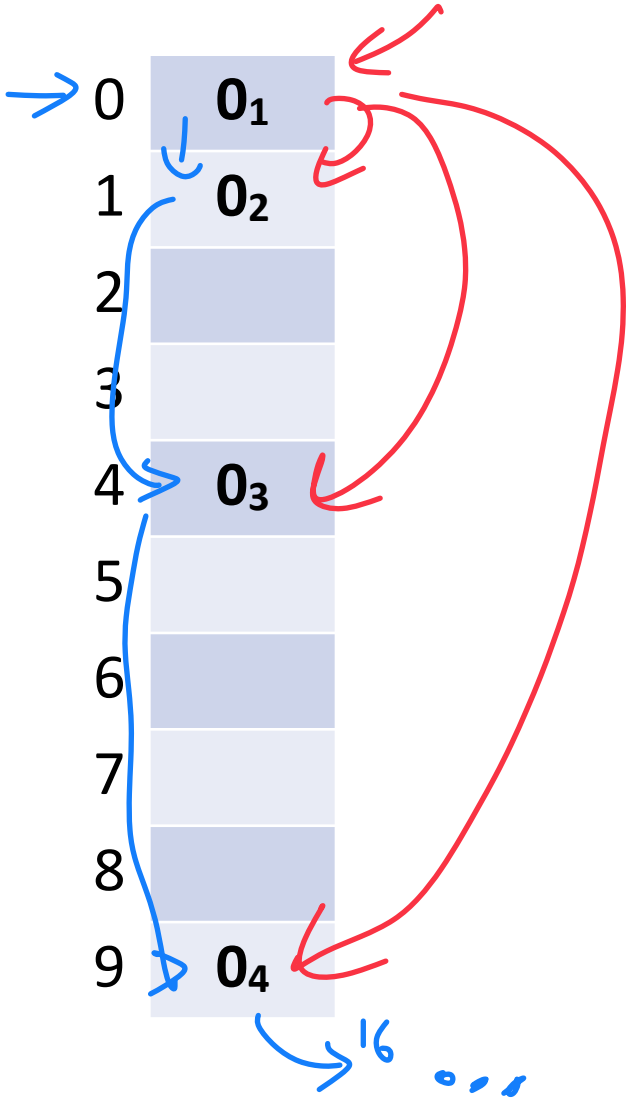
5 + 4 = 9 % 7 = 2

5 + 9 = 14 % 7 = 0

Hash functions which are "good" can be problematic (might take more than n steps to find available)

A Problem w/ Quadratic Probing

Secondary Clustering:



Description: Individual collisions still cause long "chains"

Tradeoff

Needs to be deterministic ⟷ I want to collide less often

Remedy:

↳ Add a second hash function!

A Problem w/ Quadratic Probing

Secondary Clustering:

0	0_1
1	0_2
2	
3	
4	0_3
5	
6	
7	
8	
9	0_4

Description:

Individual collisions still yield long chains

Remedy:

Be less consistent (but still deterministic)

Collision Handling: Double Hashing

S = { 16, 8, 4, 13, 29, 11, 22 }

$h_1(k) = k \% 7$

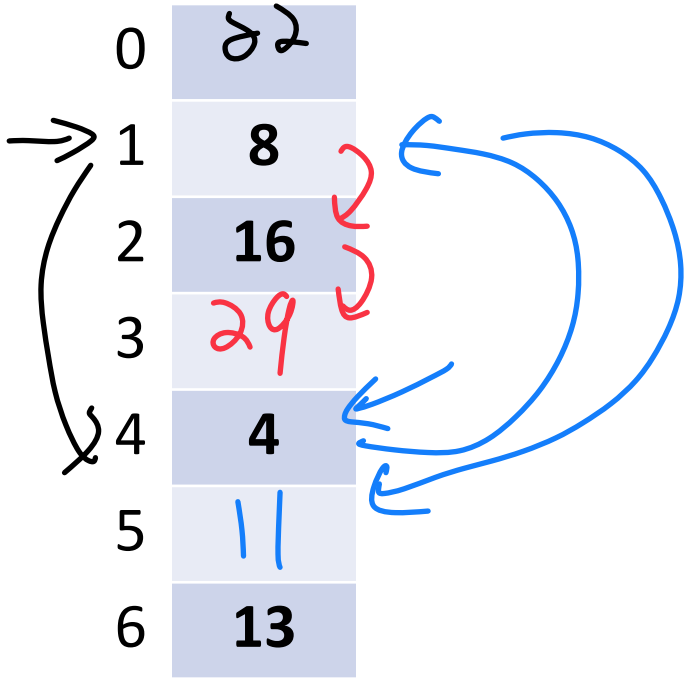
$h_2(k) = 5 - (k \% 5)$

2 4 2 ← starting pos

2 4 3 ← hash collision strategy

|S| = n

|Array| = m



$h(k, i) = (h_1(k) + i*h_2(k)) \% 7$

Try $h(k) = (k + 0*h_2(k)) \% 7$, if full...

Try $h(k) = (k + 1*h_2(k)) \% 7$, if full...

Try $h(k) = (k + 2*h_2(k)) \% 7$, if full...

Try ...

different collision resolution approaches

11 / 4

4+4 = 8 % 7 = 1

4+8 = 12 % 7 = 5

22 / 1

1+3 = 4

1+6 = 7 % 7 = 0

(Example of closed hashing)

Collision Handling: Double Hashing

S = { 16, 8, 4, 13, 29, 11, 22 }

$h_1(k) = k \% 7$

$h_2(k) = 5 - (k \% 5)$

|S| = n

|Array| = m

this is a part of list * in all***

If not careful can infinite loop!

↳ size of table is a prime #

↳ $h_2(k)$ cannot be 0

↳ h_1 & h_2 should "play nicely"

↳ be independent of each other

Running Times

(Expectation under SUHA)

(Understand why we have these rough forms)

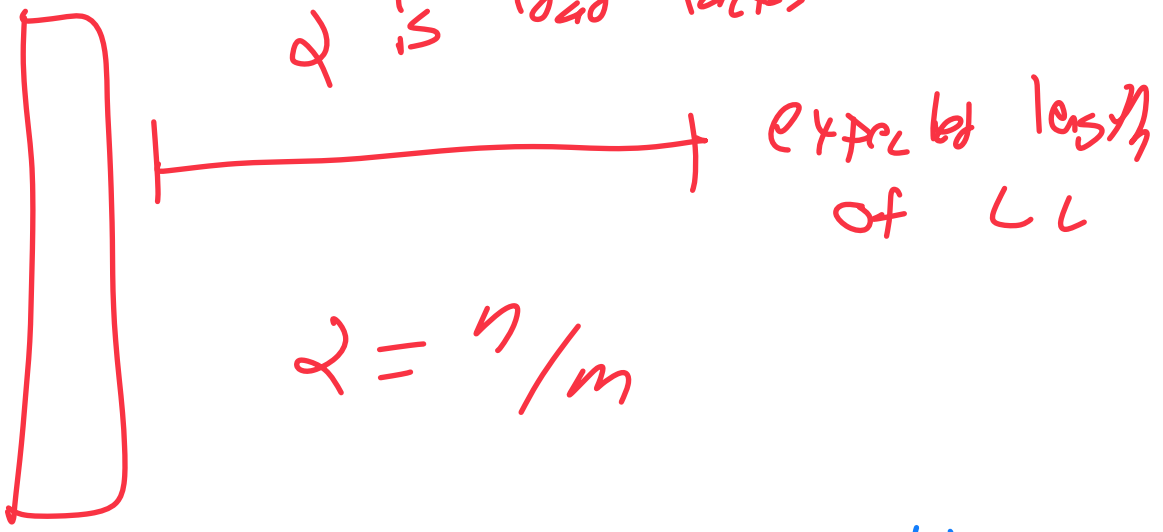
$0 \le \alpha \le \infty$

Open Hashing:

insert: 1.

find/ remove: $1 + \alpha$.

$0 \le \alpha < 1$

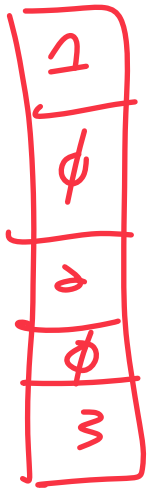


compute total # of expected collisions

Closed Hashing:

insert: $\frac{1}{1-\alpha}$.

find/ remove: $\frac{1}{1-\alpha}$.



$1 + \alpha + \alpha^2 + \alpha^3 + \alpha^4 + \dots\dots$

100% time we hash once

roll'd once

coll'd twice

B/c $\alpha < 1$, sum $= \frac{1}{1-\alpha}$ Taylor Series

Running Times (Expectation under SUHA)

Open Hashing: $0 \leq \alpha \leq \infty$

insert: 1.

find/ remove: $1 + \alpha$.

Closed Hashing: $0 \leq \alpha < 1$

insert: $\frac{1}{1-\alpha}$.

find/ remove: $\frac{1}{1-\alpha}$.

Observe:

- As α increases: runtime approaches ∞

OH: $\alpha \to \infty$, $1+\alpha \to \infty$

CH: $\alpha \to 1$, $\frac{1}{1-\alpha} \to \infty$

- If α is constant:

OH: $\alpha = C$, $1+C$ is constant

CH: $\alpha = C < 1$ $= 0.5$, $\frac{1}{1-C}$ is constant

Running Times *(Don't memorize these equations, no need.)*

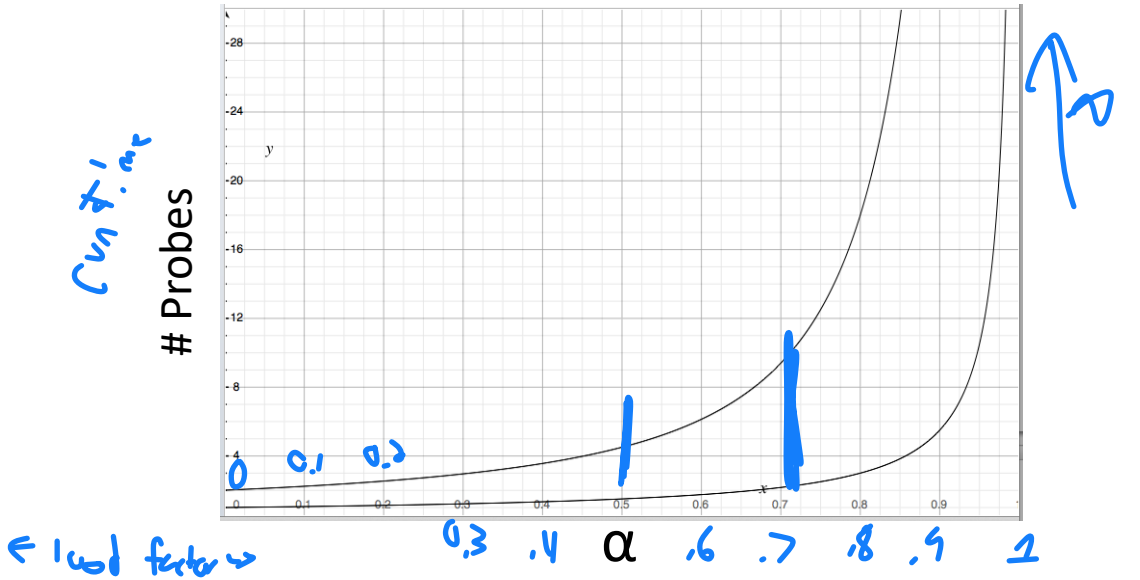
The expected number of probes for find(key) under SUHA

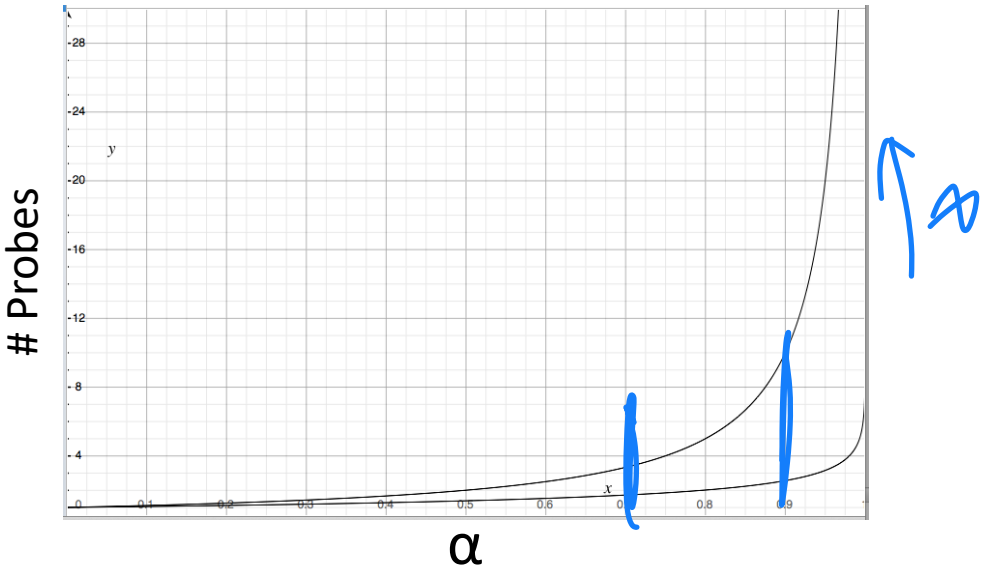
Linear Probing:

- Successful: **½(1 + 1/(1-α))**
- Unsuccessful: **½(1 + 1/(1-α))²**

Double Hashing:

- Successful: **1/α * ln(1/(1-α))**
- Unsuccessful: **1/(1-α)**





When do we resize?

LP: 0.5 – 0.7

DH: 0.7 – 0.9

Resizing a hash table

How do you resize?

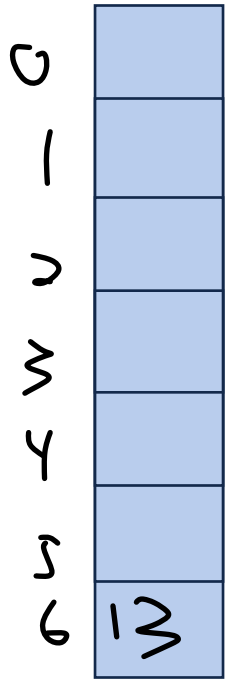
1) Double size of array + change compression
   - to nearest prime larger than double

~~a) copy all items~~

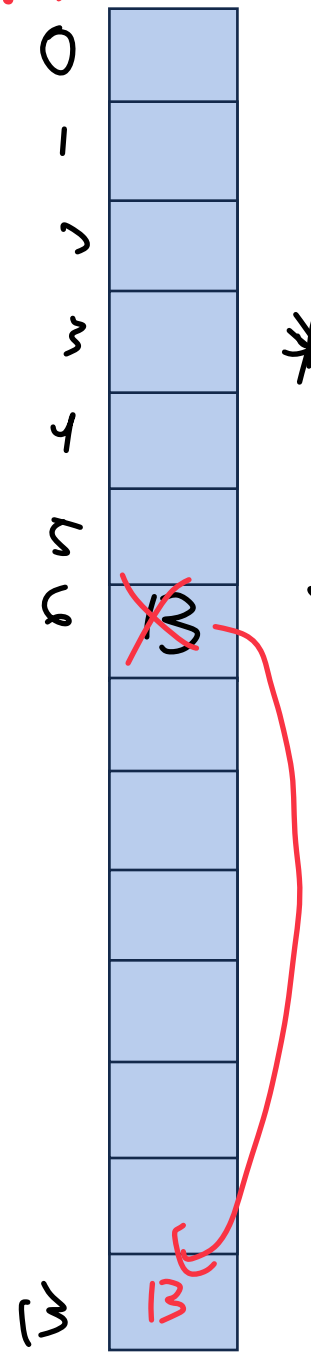
2) Rehash all items

$O(n)$ to resize n items

$h(k) = k\%7$



$h(k) = k\%13$



$O(1)^{*}$ **

* = expectation (random hash)

* = SUHA

* = Pseudo-amortized

↳ double to prime #

↳ resize when $\alpha = 0.7$-0.9

Which collision resolution strategy is better?

- Big Records: Open hashing (anything which is passed by ref)
- Structure Speed: arrays are nice! closed hashing

What structure do hash tables implement?

What constraint exists on hashing that doesn't exist with BSTs?

on Friday

Why talk about BSTs at all?

std::map in C++

```
T& map<K, V>::operator[]
pair<iterator, bool> map<K, V>::insert()
iterator map<K, V>::erase()


iterator map<K, V>::lower_bound( const K & );
iterator map<K, V>::upper_bound( const K & );
```

std::unordered_map in C++

```
T& unordered_map<K, V>::operator[]
pair<iterator, bool> unordered_map<K, V>::insert()
iterator unordered_map<K, V>::erase()
```

~~`iterator map<K, V>::lower_bound( const K & );`~~

~~`iterator map<K, V>::upper_bound( const K & );`~~

```
float unordered_map<K, V>::load_factor();
void unordered_map<K, V>::max_load_factor(float m);
```

Running Times

	Hash Table	AVL	Linked List
Find	Expectation*: Worst Case:		
Insert	Expectation*: Worst Case:		
Storage Space			

Next Class: Probabilistic Accuracy in Data Structures

1. Assume input data is random to estimate average-case performance

2. Use randomness inside algorithm to estimate expected running time

3. Use randomness inside algorithm to approximate solution in fixed time

Probabilistic Accuracy: Fermat primality test

Pick a random a in the range $[2,\ p-2]$

If p is prime and a is not divisible by p, then $a^{p-1} \equiv 1 \pmod{p}$

But… ***sometimes*** if n is composite and $a^{n-1} \equiv 1 \pmod{n}$

Probabilistic Accuracy: Fermat primality test

Let's assume $\alpha = .5$

First trial: $a = a_0$ and prime test returns 'prime!'

Second trial: $a = a_1$ and prime test returns 'prime!'

Third trial: $a = a_2$ and prime test returns 'not prime!'

Is our number prime?

What is our **false positive** probability? Our **false negative** probability?

Probabilistic Accuracy: Fermat primality test

	$a^{p-1} \equiv 1 \pmod{p}$	$a^{p-1} \not\equiv 1 \pmod{p}$
p is prime		
p is not prime		

Probabilistic Accuracy: Fermat primality test

Summary: Randomized algorithms can also have fixed (or bounded) runtimes at the cost of probabilistic accuracy.

Randomness:

Assumptions:

Probabilistic Accuracy: Fermat primality test

Summary: Randomized algorithms can also have fixed (or bounded) runtimes at the cost of probabilistic accuracy.

Randomness: The choice of α.

We can even pick more than one α if we want!

Assumptions: Only that random numbers are actually random

While strictly not true, generally an acceptable assumption in practice

Types of randomized algorithms

A **Las Vegas** algorithm is a randomized algorithm which will always give correct answer if run enough times but has no fixed runtime.

A **Monte Carlo** algorithm is a randomized algorithm which will run a fixed number of iterations and may give the correct answer.

What type of algorithm is Fermat's primality test?

What type of algorithm is randomized quick sort?

Bonus Slides

Hash Table

Worst-Case behavior is bad — but what about randomness?

1) **Fix *h***, our hash, and assume it is good for ***all keys***:

Simple Uniform Hashing Assumption

(Assume our dataset hashes optimally)

2) Create a ***universal hash function family:***

Given a collection of hash functions, pick one randomly

Like **random quicksort** if pick of hash is random, good expectation!

Hash Function (Division Method)

Hash of form: $h(k) = k \,\%\, m$

Pro:

Con:

Hash Function (Mid-Square Method)

Hash of form: $h(k) = (k * k)$ and take b bits from middle ($m = 2^b$)

Hash Function (Mid-Square Method)

Hash of form: $h(k) = (k * k)$ and take b bits from middle ($m = 2^b$)

Hash Function (Multiplication Method)

Hash of form: $h(k) = \lfloor m(kA \% 1) \rfloor,\ 0 \leq A \leq 1$

Pro:

Con:

Hash Function (Universal Hash Family)

Hash of form: $h_{ab}(k) = \big((ak + b)\,\%\,p\big)\,\%\,m,\ a, b \in Z_p^*, Z_p$

$$\forall k_1 \neq k_2,\ Pr_{a,b}(h_{ab}[k_1] = h_{ab}[k_2]) \leq \frac{1}{m}$$

Pro:

Con: