Data Structures and Algorithms
Hashing 3

CS 225
Brad Solomon

November 15, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Learning Objectives

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

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

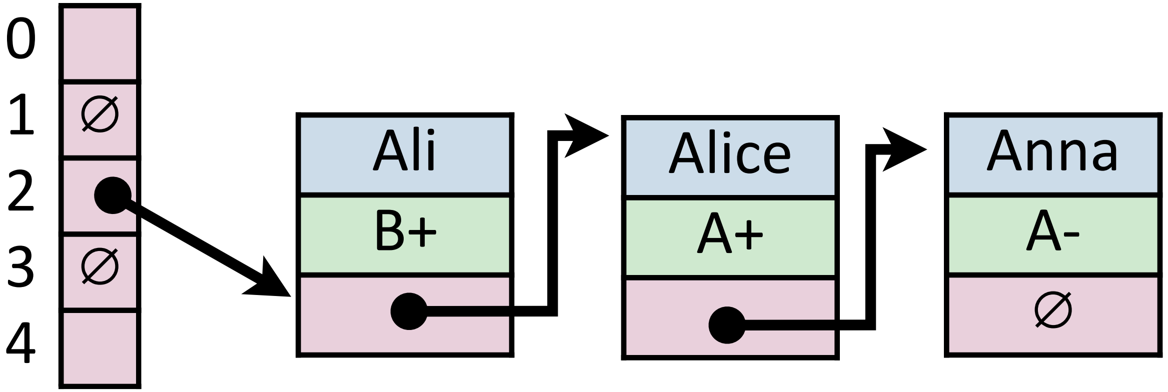
$$Pr(h[k_1]) = \frac{1}{m}$$

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

Open vs Closed Hashing

Addressing hash collisions depends on your storage structure.

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



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

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

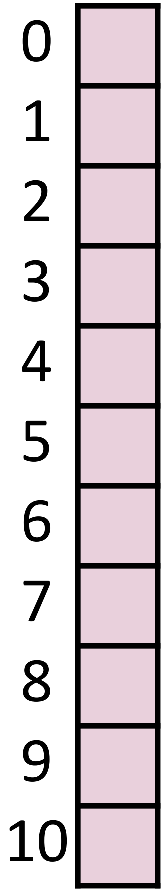
Separate Chaining Under SUHA

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

Find runs in: O(1+α).

Insert runs in: O(1).

Remove runs in: O(1+α).



Collision Handling: Linear Probing

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

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

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

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 …

(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”

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:

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

0	
1	**8**
2	**16**
3	
4	**4**
5	
6	**13**

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 ...

A Problem w/ Quadratic Probing

Secondary Clustering:

0	$\mathbf{0_1}$
1	$\mathbf{0_2}$
2	
3	
4	$\mathbf{0_3}$
5	
6	
7	
8	
9	$\mathbf{0_4}$

Description:

Remedy:

(Example of closed hashing)

Collision Handling: Double Hashing

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

$|S| = n$

$|Array| = m$

$h_1(k) = k \% 7$

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

0	
1	8
2	16
3	
4	4
5	
6	13

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

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 ...

Running Times

(Expectation under SUHA)

(Understand why we have these rough forms)

Open Hashing:

insert: ___________.

find/ remove: ___________.

Closed Hashing:

insert: ___________.

find/ remove: ___________.

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:

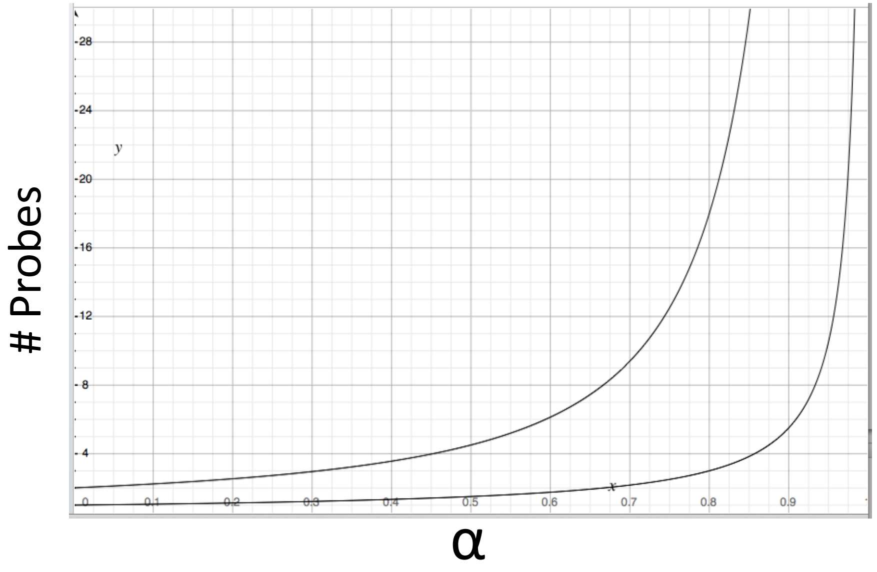
- If α is constant:

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

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

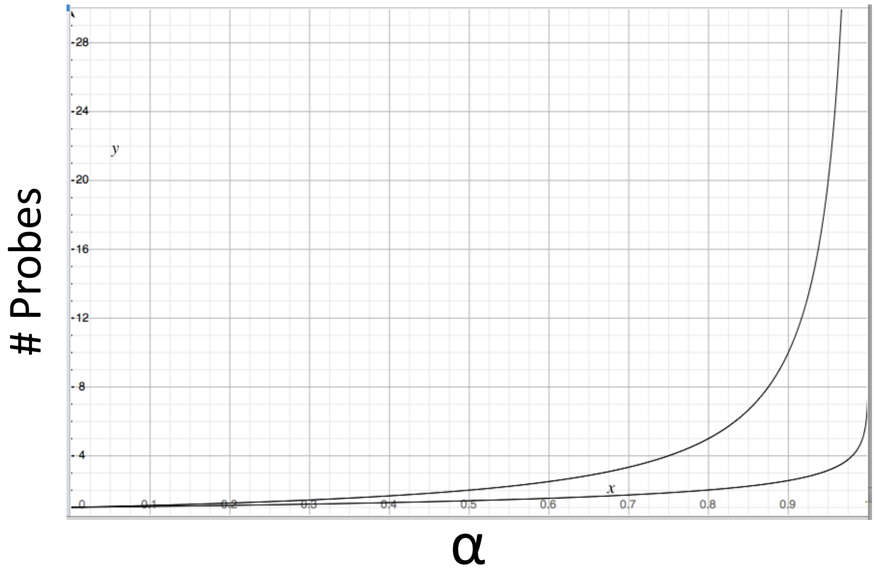
Linear Probing:

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



Double Hashing:

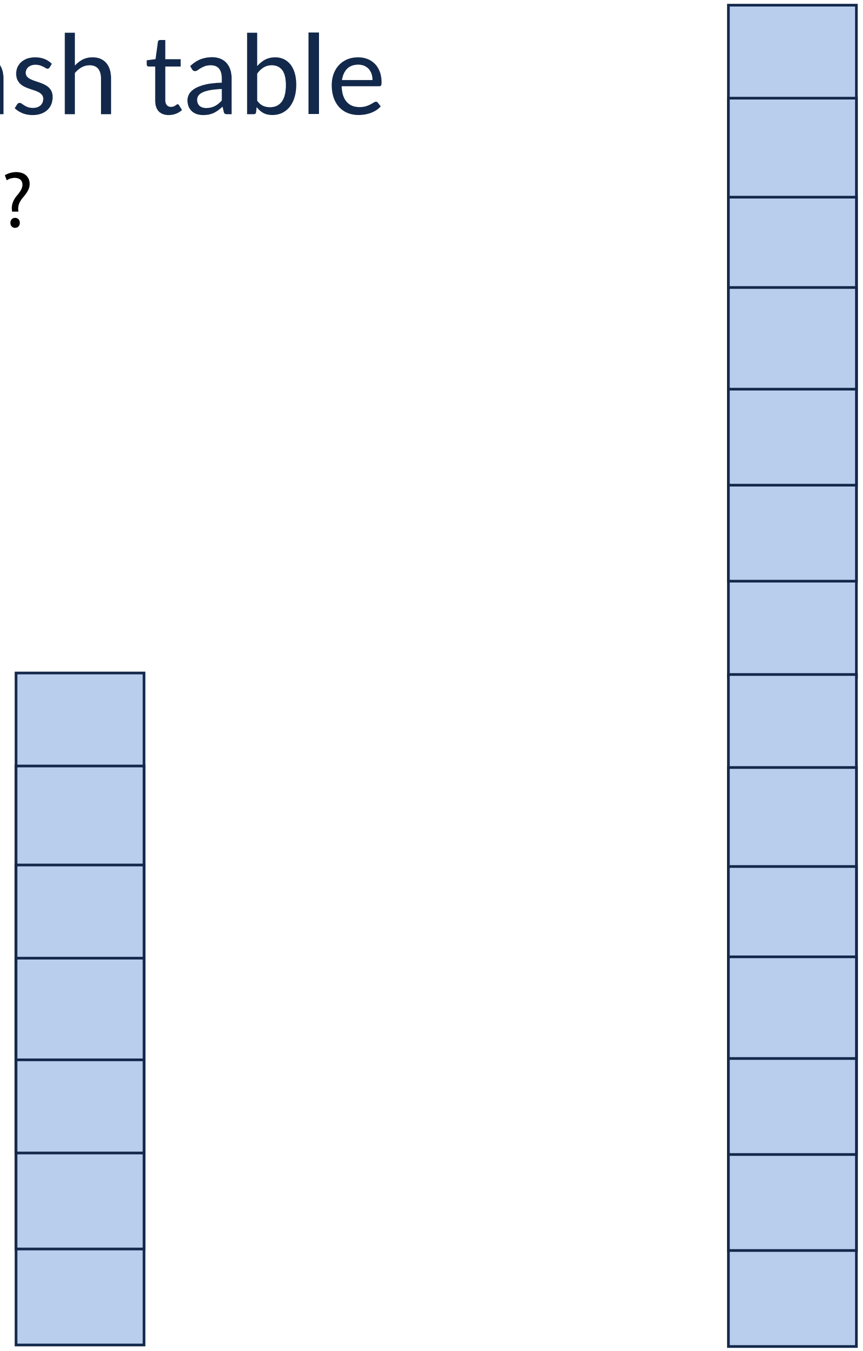
- Successful: **1/α * ln(1/(1-α))**
- Unsuccessful: **1/(1-α)**



When do we resize?

Resizing a hash table

How do you resize?



Which collision resolution strategy is better?

- Big Records:
- Structure Speed:

What structure do hash tables implement?

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

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			

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: