Hash Tables I

Data Structures and Parallelism

Announcements

Midterm Study Resources Online

Final Location Decided: CSE2 G01 on Thursday Aug 22, 9:40-10:40

Project 2
- Multiple dictionaries and sorting methods
- Writeup is extensive. It really is almost half of the assignment.

Exercise 4 due Wed
- Tree method recurrence solving
- Check your answer with the master theorem

Another Dictionary

Our guiding principle for designing AVL trees was optimizing for the **worst case.**

What if we want to optimize for the **average case**?

That goal will lead us to a totally different data structure: **hash tables**

A Simple Case

Suppose you were promised your keys would be distinct numbers in the range $\mathbf{0}$ to k.

How would you implement a dictionary. What are the running times for `insert`, `find`, and `delete`?

Just store the values in an array of size $k+1$.

Store the value associated with i at index i of the array.

$O(1)$ operations for everything!

Generalization (Step 1)

What if the keys are guaranteed to be integers,

But the upper limit is huge.

Why not just use the array from last time?

How could we still use the array of size k?

Generalization (Step 1)

What if the keys are guaranteed to be integers,

But the upper limit is huge.

Why not just use the array from last time?

WAY too much space

How could we still use the array of size k?

Map the keys into the range $\{0, \ldots, k-1\}$.

% table size

Map to index `key % TableSize`

indices	0	1	2	3	4	5	6	7	8	9
array	“:(” ’	“biz”				“bar”			“bop”	

```
put(0, “foo”);  0 % 10 = 0
put(5, “bar”);  5 % 10 = 5
put(11, “biz”) 11 % 10 = 1
put(18, “bop”);18 % 10 = 8
put(20, “:(”);  20 % 10 = 0
```

→ Collision!

Problem 1: What do we do when the keys collide?

Collision Resolution

Multiple Possible Strategies.

We’ll talk about “open addressing” strategies later.

First, the Strategy for P2 is “Separate Chaining”

Idea: If more than one thing goes to the same spot

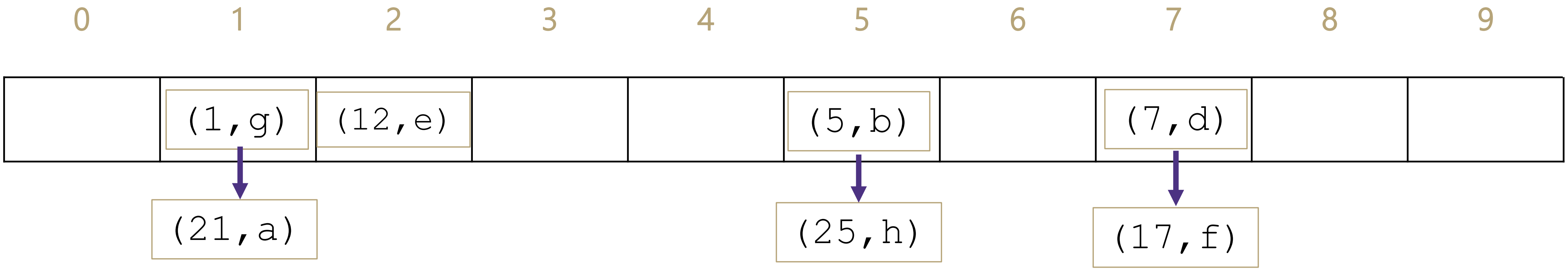
Just stuff them all in that one spot!

Separate Chaining

Instead of an array of values

Have an array of (say) LinkedLists of values.

Insert the following keys: (1, a) (5,b) (21,a) (7,d) (12,e) (17,f) (1,g) (25,h)



0	1	2	3	4	5	6	7	8	9
	(1,g)	(12,e)			(5,b)		(7,d)		
	(21,a)				(25,h)		(17,f)		

Running Times

What are the running times for:

`insert`

Best:

Worst:

`find`

Best:

Worst:

`delete`

Best:

Worst:

Running Times

What are the running times for:

`insert`

Best: $O(1)$

Worst: $O(n)$

`find`

Best: $O(1)$

Worst: $O(n)$

`delete`

Best: $O(1)$
Worst: $O(n)$

Average Case

What about on average?
Let's assume that the keys are randomly distributed

What is the average running time if the size of the table $TableSize$ and we've inserted n keys?

`insert`

`find`

`delete`

Average Case

What about on average?
Let's assume that the keys are randomly distributed

What is the average running time if the size of the table $TableSize$ and we've inserted n keys?

`insert` $O(1)$

`find` $O\left(1+\frac{n}{TableSize}\right)$

`delete` $O\left(1+\frac{n}{TableSize}\right)$

Average Case

What about on average?
Let's assume that the keys are uniformly distributed

What is the average running time if the size of the table $TableSize$ and we've inserted n keys?

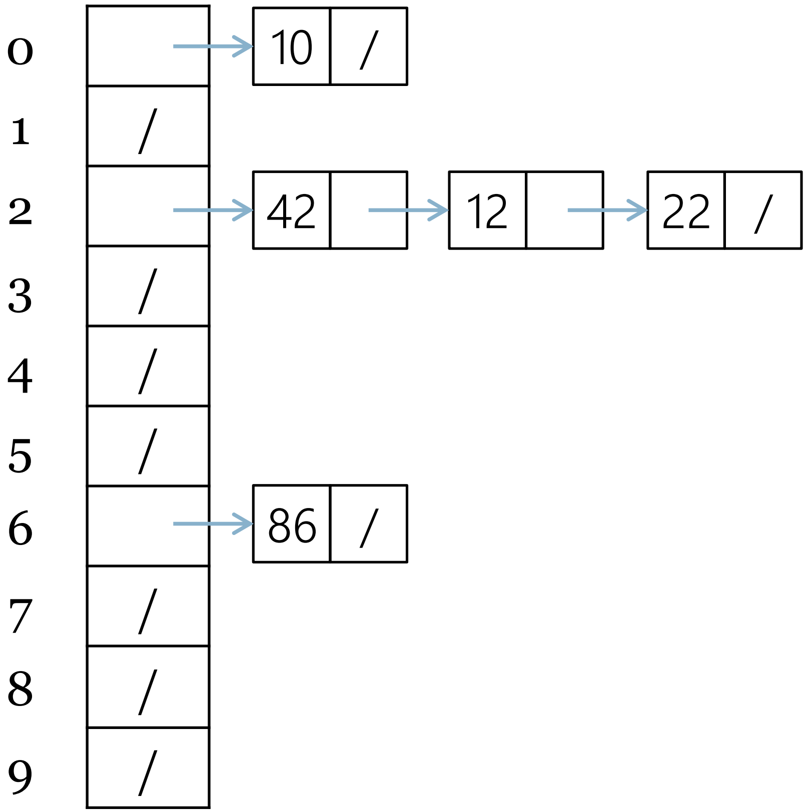
`insert` $O(1)$

`find` $O(1+\lambda)$

`delete` $O(1+\lambda)$

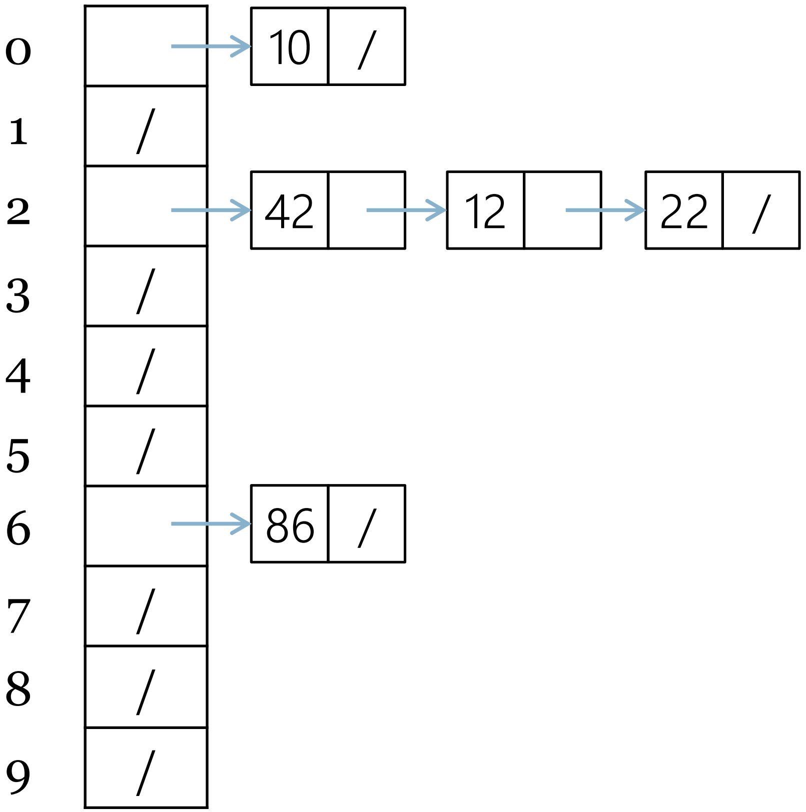
We'll denote $\frac{n}{TableSize}$ by λ.
Often called "load factor"

Load Factor?



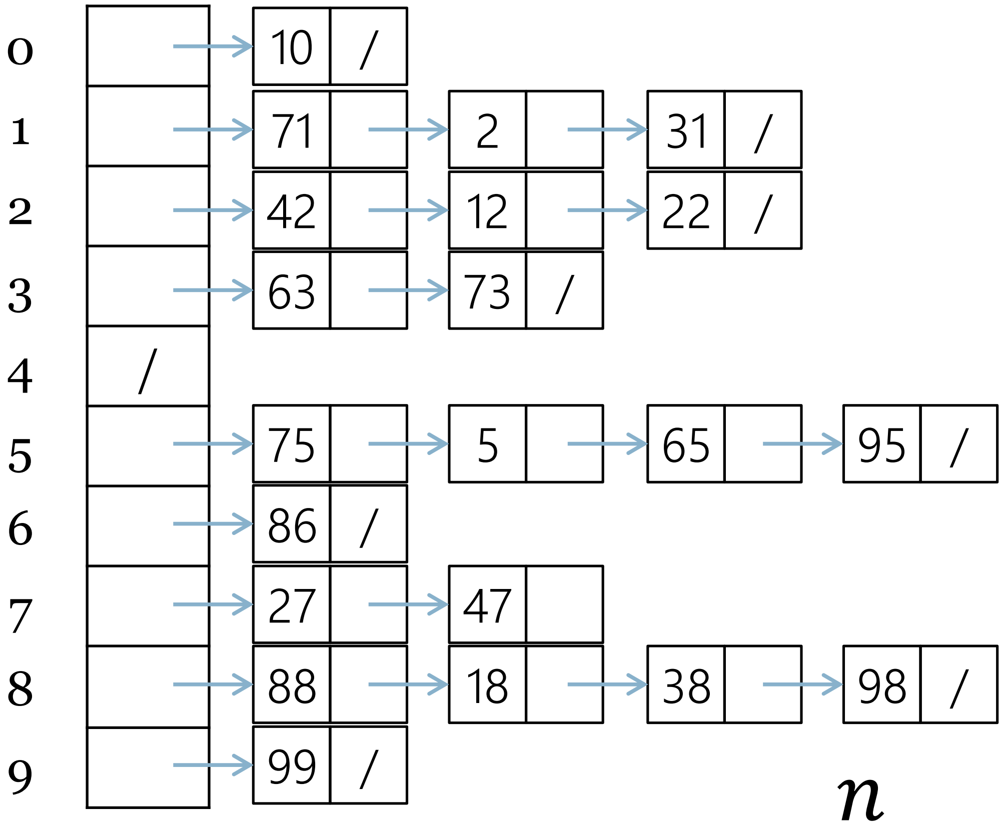
$$\lambda = \frac{n}{TableSize} = ?$$

Load Factor?



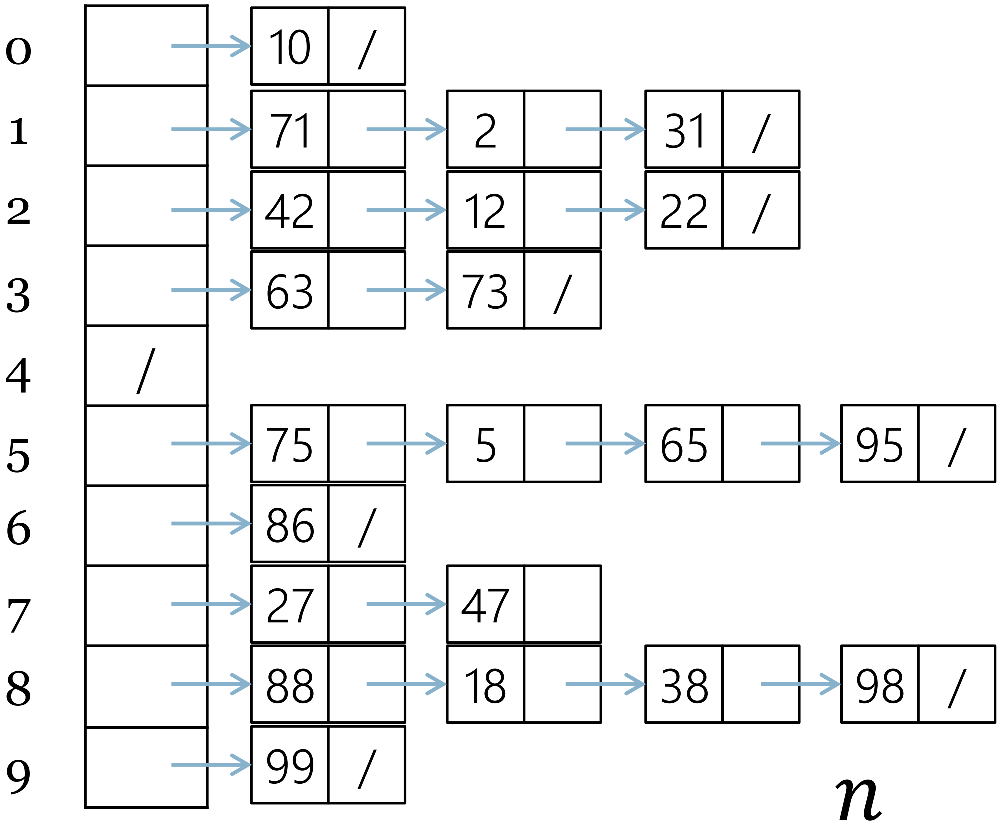
$$\lambda = \frac{n}{TableSize} = \frac{5}{10} = 0.5$$

Load Factor?



$$\lambda = \frac{n}{TableSize} = ?$$

Load Factor?



$$\lambda = \frac{n}{TableSize} = \frac{21}{10} = 2.1$$

When λ Grows

If we keep inserting things into the array, λ will keep increasing.

We'll never *really* run out of room.

When should we resize?

When it slows us down, i.e. when λ is a constant.

Heuristic: for separate chaining λ between 1 and 3 is a good time to resize.

Resizing

How long does it take to resize?

Need to:

Remake the table

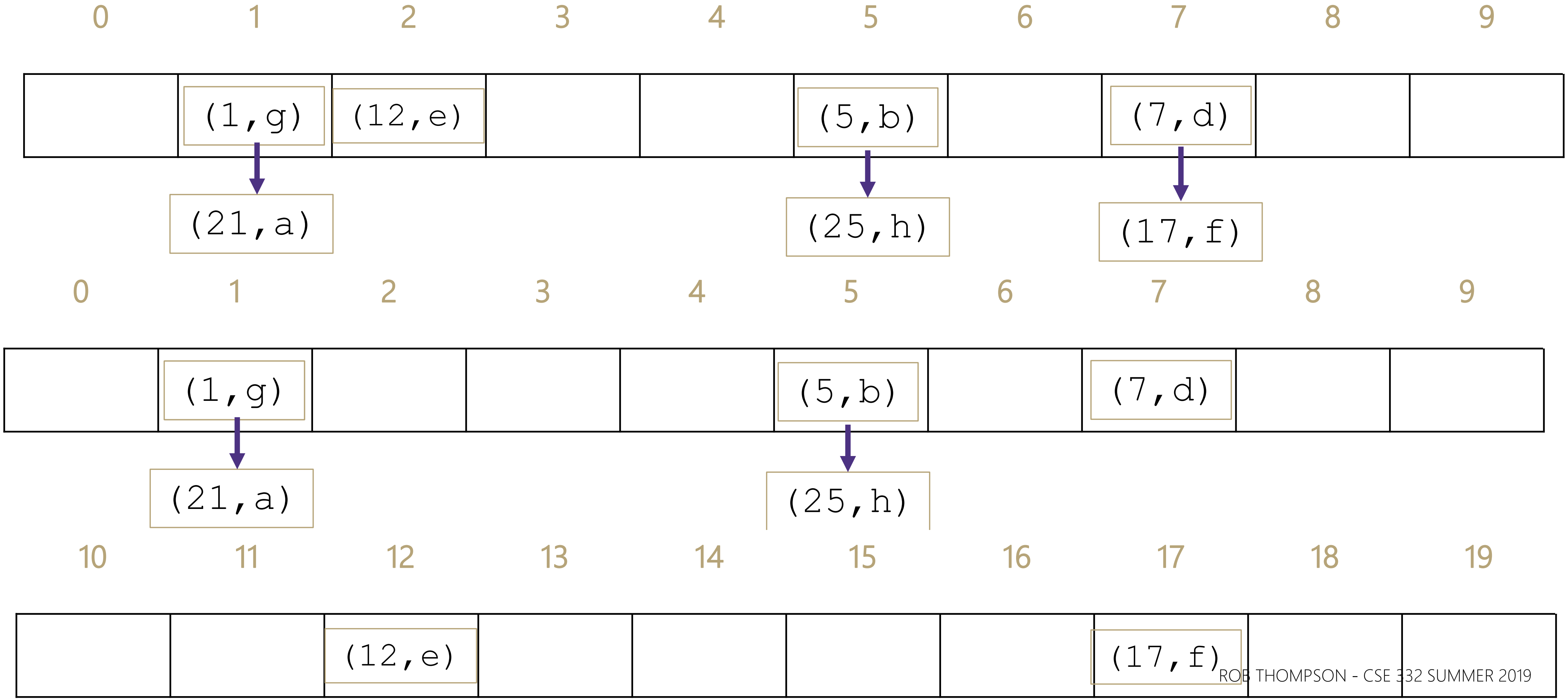
Evaluate the hash function over again.

Re-insert.

Total time: $O(n + TableSize) = O(n)$ if λ is a constant.

Resizing Redux

Let’s resize by doubling the size of the array.



0	1	2	3	4	5	6	7	8	9
	(1,g) (21,a)	(12,e)			(5,b) (25,h)		(7,d) (17,f)		

0	1	2	3	4	5	6	7	8	9
	(1,g) (21,a)				(5,b) (25,h)		(7,d)		

10	11	12	13	14	15	16	17	18	19
		(12,e)					(17,f)		

Resizing Redux

That didn’t work very well!

It turned out that most of the keys that were equal mod 10 were also equal mod 20.

This is likely with real data.

Don’t just double the table size

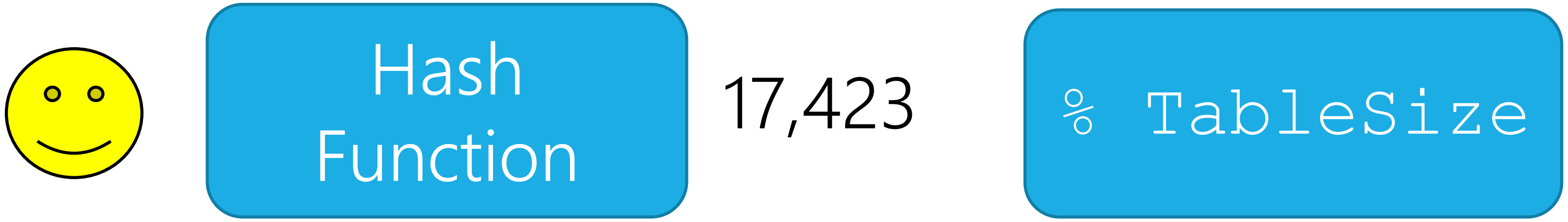
Instead make the table size some new prime number.

Collisions can still happen, but patterns with multiple prime numbers are rarer in real data than patterns with powers of 2.

Reaching the Average Case

In general our keys might not be integers.

Given an arbitrary object type E, how do we get an array index?



Usually Object writer's responsibility

Usually HashTable writer's responsibility

How do we make our assumption (keys are uniformly distributed) true?
Or at least true-ish?

Designing a Hash Function

For simplicity, let’s start with Strings.

Question: How many Strings are there compared to ints?

WAY more strings.

Can we always avoid collisions

- NO!

We can try to minimize them though.

Some Possible Hash Functions

For each of these hash functions, think about

- what Strings will cause collisions
- how long it will take to evaluate

Keys: strings of form $s_0 s_1 \ldots s_{k-1}$ (s_i are `chars` in range [0,256])

$h(K) = s_0$

$h(K) = \sum_{i=0}^{k-1} s_i$

A Better Hash Function

$h(K) = \sum_{i=0}^{k-1} s_i \cdot 31^i$

Can we do this fast? Avoid calculating 31^i directly

```
for(i=k-1; i>=0;i--){
    h = 31*h + s[i];
 }
```

Other Classes

Should we use that same hash function if the strings are all URLs?

Other Classes

Should we use that same hash function if the strings are all URLs?

No! “https://www.” Is worthless, use the rest of the string

Other Classes

Should we use that same hash function if the strings are all URLs?

No! “https://www.” Is worthless, use the rest of the string

Person Class

String firstname; String middlename; String lastname; Date birthdate;

Tradeoff between speed and collision avoidance.

What to hash is often just an unprincipled guess.

General Principles

You have 32 bits, use them.

Use different overlapping bits for different parts of the hash

-This is why a factor of 37^i works better than 256^i

Bitwise xor if you have to combine

If keys are known in advance (not likely), choose a perfect hash

DON'T DO THIS IF YOU DON'T HAVE TO

Rely on others to get this right if you can.

Thoughts on separate chaining

Worst-case time for `find` is linear

- But only with really bad luck or bad hash function
- So not worth avoiding (e.g., with balanced trees at each bucket)
  - Keep # of items in each bucket small
  - Overhead of AVL tree, etc. not worth it for small n

Beyond asymptotic complexity, some "data-structure engineering" can improve constant factors

- Linked list vs. array or a hybrid of the two
- Move-to-front (part of Project 2)
- Leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
  - A time-space trade-off…

Java Specific Notes

Every object in Java implements the `hashCode` method.

If you define a new Object, and want to use a hash table, you might want to override `hashCode`.

But if you do, you also need to override `equals`

Such that

If `a.equals(b)` then `a.hashCode() == b.hashCode()`

This is part of the contract. Other code makes this assumption!

What about the converse?

Can’t require it, but you should try to make it true as often as possible.