CSE332: Data Abstractions

Lecture 8: Hashing

James Fogarty

Winter 2012

Conclusion of Balanced Trees

- Balanced trees make good dictionaries because they guarantee logarithmic-time **`find`**, **`insert`**, and **`delete`**
  - Essential and beautiful computer science
  - But only if you can maintain balance within the time bound

- AVL trees maintain balance by tracking height and allowing all children to differ in height by at most 1

- B trees maintain balance by keeping nodes at least half full and all leaves at same height

- Other great balanced trees (see text; worth knowing they exist)
  - Red-black trees: all leaves have depth within a factor of 2
  - Splay trees: self-adjusting; amortized guarantee; no extra space for height information

Simple Implementations

For dictionary with *n* key/value pairs

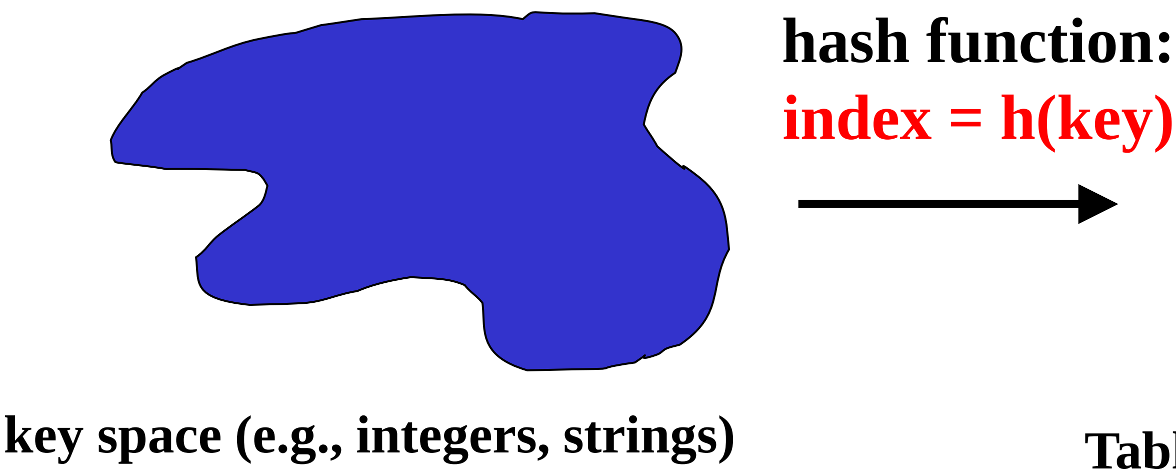
	`insert`	`find`	`delete`	
• Unsorted linked-list	$O(1)$	$O(n)$	$O(n)$	
• Unsorted array	$O(1)$	$O(n)$	$O(n)$	
• Sorted linked list	$O(n)$	$O(n)$	$O(n)$	
• Sorted array	$O(n)$	$O(\texttt{log}\ n)$	$O(n)$	
• Balanced tree	$O(\texttt{log}\ n)$	$O(\texttt{log}\ n)$	$O(\texttt{log}\ n)$	
• Magic array	$O(1)$	$O(1)$	$O(1)$	average case

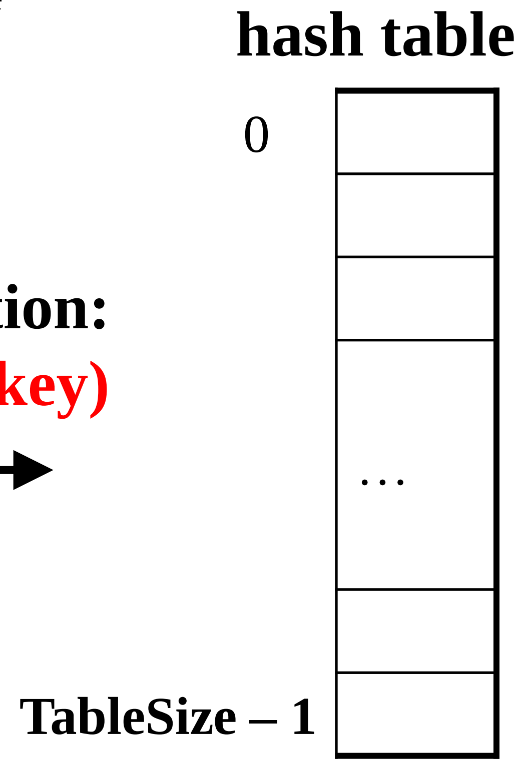
Hash Tables

- Aim for constant-time **`find`**, **`insert`**, and **`delete`**
  - "On average" under some reasonable assumptions

- A hash table is an array of some fixed size

- Basic idea:





Hash Tables vs. Balanced Trees

- In terms of a Dictionary ADT for just **`insert`**, **`find`**, **`delete`**, hash tables and balanced trees are just different data structures
  - Hash tables $O(1)$ on average (*assuming* few collisions)
  - Balanced trees $O(\log n)$ worst-case

- Constant-time is better, right?
  - Yes, but you need “hashing to behave” (must avoid collisions)
  - Yes, but **`findMin`**, **`findMax`**, **`predecessor`**, **`successor`** go from $O(\log n)$ to $O(n)$, **`printSorted`** from $O(n)$ to $O(n \log n)$

- **Moral:** If you need to frequently use operations based on sort order, then you may prefer a balanced BST instead.

Hash Tables

- There are m possible keys (m typically large, even infinite)
- We expect our table to have only n items
- n is much less than m (often written $n << m$)

Many dictionaries have this property

  – Compiler: All possible identifiers allowed by the language vs. those used in some file of one program

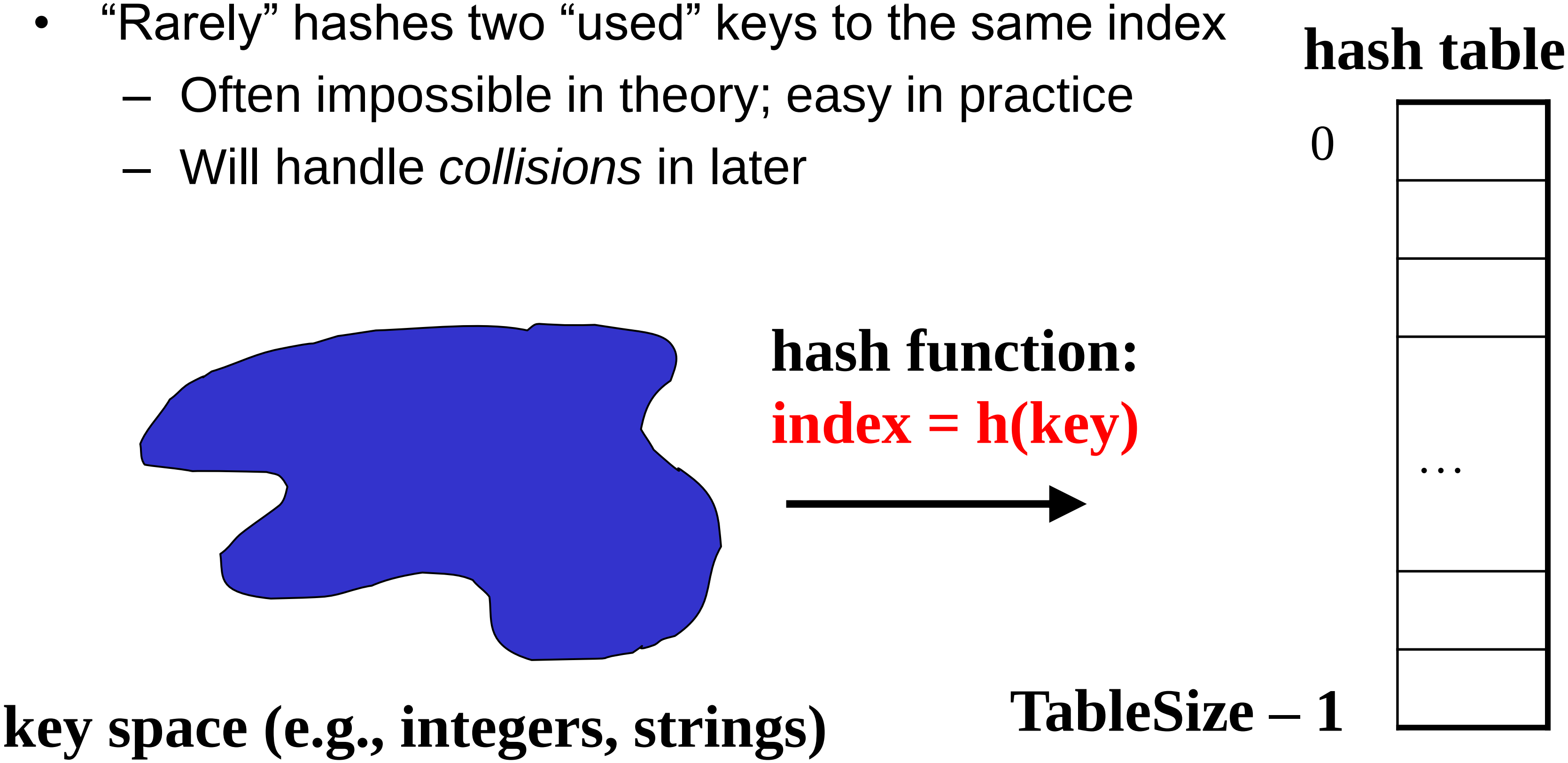
  – Database: All possible student names vs. students enrolled

  – AI: All possible chess-board configurations vs. those considered by the current player

Hash Functions

An ideal hash function:

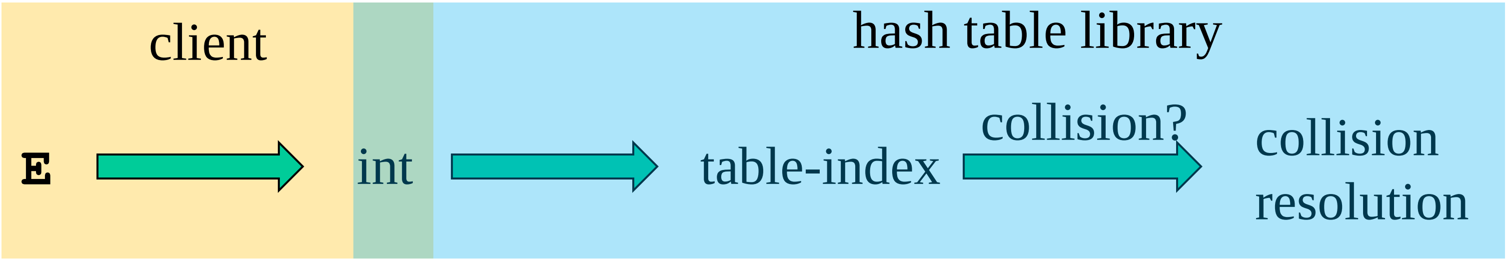
- Is fast to compute
- "Rarely" hashes two "used" keys to the same index
  - Often impossible in theory; easy in practice
  - Will handle *collisions* in later



Who Hashes What

- Hash tables can be generic
  - To store elements of type **`E`**, we just need **`E`** to be:
    1. Comparable: order any two **`E`** (as with all dictionaries)
    2. Hashable: convert any **`E`** to an **`int`**

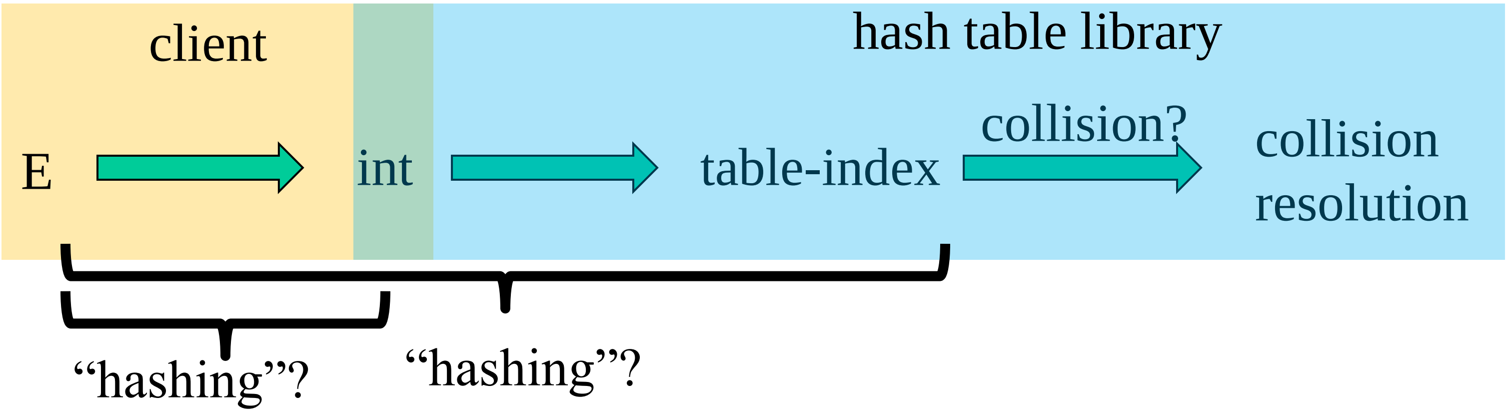
- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:



- We will learn both roles, but most programmers “in the real world” spend more time as clients while understanding the library

More on Roles

Some ambiguity in terminology on which parts are “hashing”



Two roles must both contribute to minimizing collisions (heuristically)

- Client should aim for different ints for expected items
  - Avoid “wasting” any part of `E` or the 32 bits of the `int`
- Library should aim for putting “similar” `int`s in different indices
  - conversion to index is almost always “mod table-size”
  - using prime numbers for table-size is common

What to Hash?

We will focus on two most common things to hash: ints and strings

- If you have objects with several fields, it is usually best to hash most of the “identifying fields” to avoid collisions
- Example:

```
class Person {
   String first; String middle; String last;
   Date birthdate;
}
```

- An inherent trade-off: hashing-time vs. collision-avoidance

Hashing Integers

- key space = integers
- Simple hash function:

  `h(key) = key % TableSize`

  – Client: **`f(x) = x`**

  – Library **`g(x) = f(x) % TableSize`**

  – Fairly fast and natural
- Example:

  – TableSize = 10

  – Insert 7, 18, 41, 34, 10

  – (As usual, ignoring corresponding data)

0	10
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Collision Avoidance

- With "`x % TableSize`" the number of collisions depends on
  - the ints inserted
  - `TableSize`
- Larger table-size tends to help, but not always
  - Example: 70, 24, 56, 43, 10
    with `TableSize` = 10 and `TableSize` = 60
- Technique: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern,
  - "Multiples of 61" are probably less likely than "multiples of 60"
  - We will see some collision strategies do better with prime size

More Arguments for a Prime Size

If **`TableSize`** is 60 and…

- Lots of data items are multiples of 2, wasting 50% of table
- Lots of data items are multiples of 5, wasting 80% of table
- Lots of data items are multiples of 10, wasting 90% of table

If **`TableSize`** is 61…

- Collisions can still happen but 2, 4, 6, 8, … will fill table
- Collisions can still happen, but 5, 10, 15, 20, … will fill table
- Collisions can still happen but 10, 20, 30, 40, … will fill table

In general, if **`x`** and **`y`** are “co-prime” (means **`gcd(x,y)==1`**),
then **`(a * x) % y == (b * x) % y`** if and only if **`a % y == b % y`**

- Good to have a **`TableSize`** that has
  no common factors with any “likely pattern” of **`x`**

What if `key` is not an `int`?

- If keys are not `int`s, the client must convert to an `int`
  - Trade-off: speed and distinct keys hashing to distinct `int`s

- Common and important example: Strings
  - Key space K = $s_0 s_1 s_2 \ldots s_{m-1}$
    - where s_i are chars: $s_i \in [0,256]$
  - Some choices: Which best avoid collisions?

1. $h(K) = s_0$ % TableSize

2. $h(K) = \left(\sum_{i=0}^{m-1} s_i \right)$ % TableSize

3. $h(K) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right)$ % TableSize

Combining Hash Functions

A few rules of thumb / tricks:

1. Use all 32 bits (careful, that includes negative numbers)
2. Use different overlapping bits for different parts of the hash
   - This is why a factor of 37^i works better than 256^i
   - Example: “abcde” and “ebcda”
3. When smashing two hashes into one hash, use bitwise-xor
   - bitwise-and produces too many 0 bits
   - bitwise-or produces too many 1 bits
4. Rely on expertise of others; consult books and other resources
5. Advanced: If keys are known ahead of time, a *perfect hash*

Collision Resolution

Collision:

When two keys map to the same location in the hash table

We try to avoid it, but number-of-keys exceeds table size

So hash tables generally need to support collision resolution

Separate Chaining

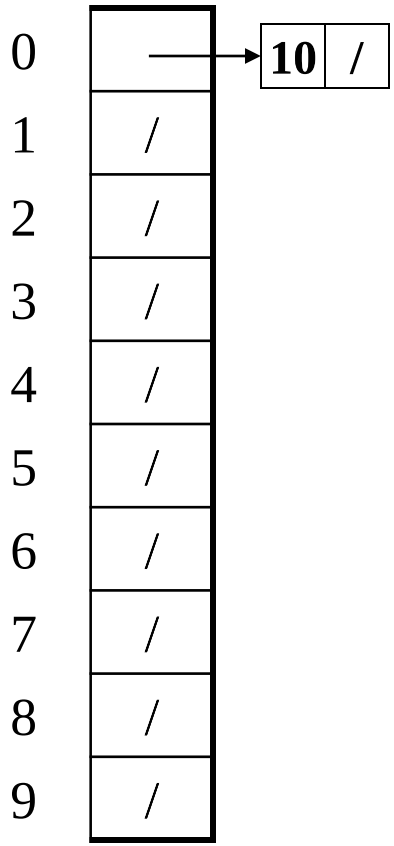
0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

Chaining:
  All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
  insert 10, 22, 107, 12, 42 with mod hashing and **`TableSize`** = 10

Separate Chaining

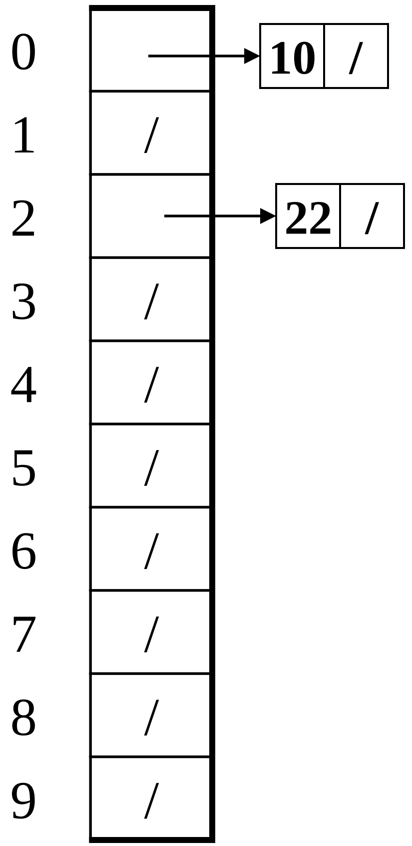


Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

Separate Chaining



Chaining:

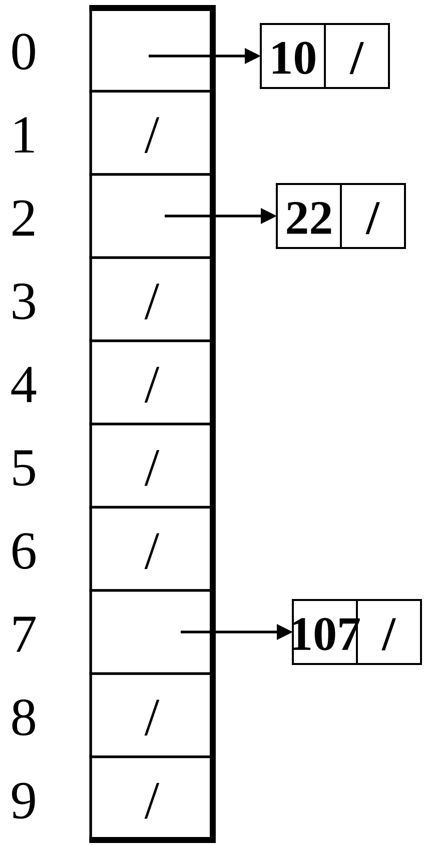
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

Separate Chaining



Chaining:

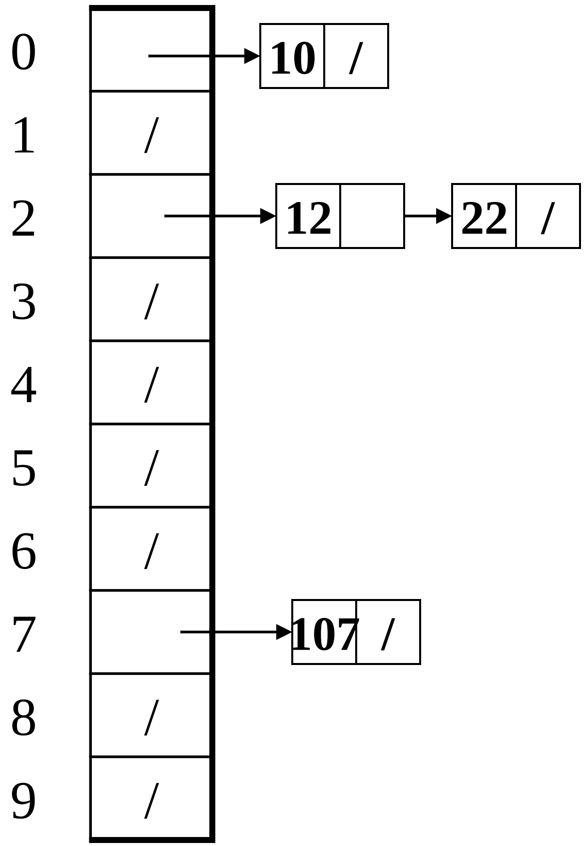
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

Separate Chaining



Chaining:

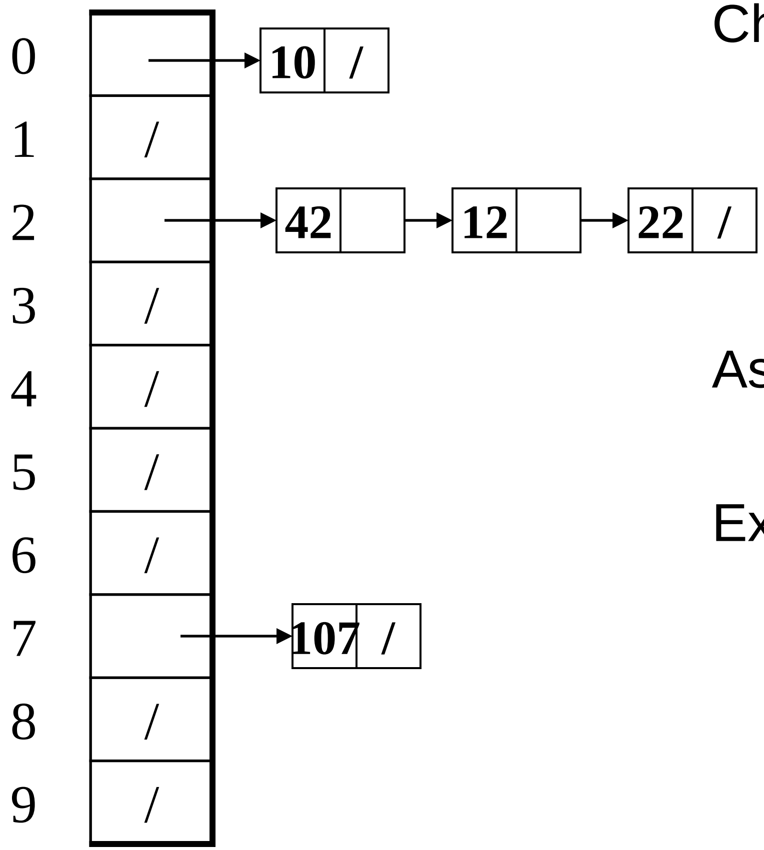
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42 with mod hashing and **`TableSize`** = 10

Separate Chaining



Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

Thoughts on Separate Chaining

- Worst-case time for **`find`**?
  - Linear
  - But only with really bad luck or bad hash function
  - So not worth avoiding (e.g., with balanced trees at each bucket)
    - Keep small number of items in each bucket
    - Overhead of tree balancing not worthwhile for small n

- Beyond asymptotic complexity, some “data-structure engineering”
  - Linked list, array, or a hybrid
  - Move-to-front list (as in Project 2)
  - Leave one element in the table itself,
    to optimize constant factors for the common case

More Rigorous Separate Chaining Analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{\mathrm{N}}{\mathrm{TableSize}} \quad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is ___

So if some inserts are followed by *random* finds, then on average:

- Each unsuccessful **`find`** compares against ____ items
- Each successful **`find`** compares against _____ items

- How big should TableSize be??

More Rigorous Separate Chaining Analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{\mathrm{N}}{\mathrm{TableSize}} \quad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is λ

So if some inserts are followed by *random* finds, then on average:

- Each unsuccessful **`find`** compares against λ items
- Each successful **`find`** compares against $\lambda / 2$ items
- If λ is low, find & insert likely to be O(1)
- We like to keep λ around 1 for separate chaining

Separate Chaining Deletion?

Separate Chaining Deletion

- Not too bad
  - Find in table
  - Delete from bucket

- Delete 12

- Similar run-time as insert