CSE 373: Data Structures & Algorithms

Hash Tables

Riley Porter

Winter 2017

Course Logistics

- HW2 clarification posted on spec: buildQueue replaces elements, not adds them.

- Weekly Summaries changed to Topic Summaries, first one out on amortized runtime, rest out soon.

Review + Motivating Hash Tables

	`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(logn)*	*O(n)*
BST	*O(n)*	*O(n)*	*O(n)*
AVL Tree	*O(logn)*	*O(logn)*	*O(logn)*
Magic Array?	*O(1)*	*O(1)*	*O(1)*

Sufficient "magic":

- Use key to compute array index for an item in *O*(1) time [doable]
- Have a different index for every item [magic]

Motivating Hash Tables

- Let's say you are tasked with counting the frequency of integers in a text file. You are guaranteed that only the integers 0 through 100 will occur:

  For example: 5, 7, 8, 9, 9, 5, 0, 0, 1, 12

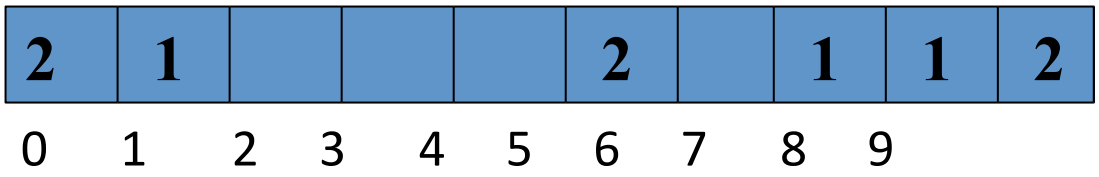
  Result: 0 → 2 1 → 1 5 → 2 7 → 1 8 → 1 9 → 2

  What structure is appropriate?

  Tree?

  List?

  Array?



Motivating Hash Tables

Now what if we want to associate name to phone number?

Suppose keys are first, last names

- how big is the key space?

What if we could map a large set of keys to a small amount of space? Like mapping all possible strings to the set of numbers from 1 to 100?

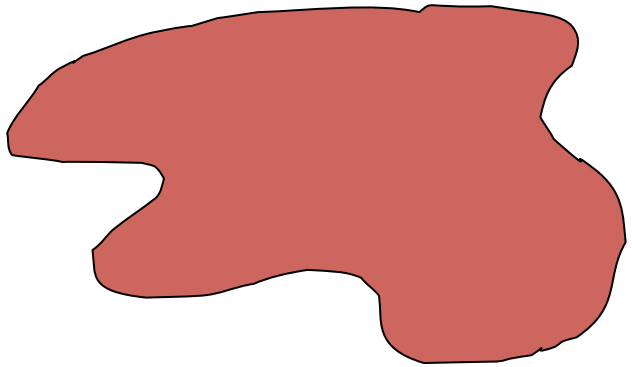
Hash Functions

- Maps any key to a number
  - result should be constrained to some range
  - passing in the same key should always give the same result
- Keys should be distributed over a range
  - very bad if everything hashes to 1!
  - should "look random"
- How would we write a hash function for String objects?

Hash Functions

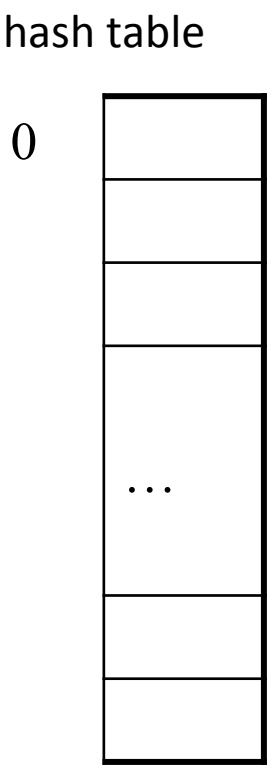
An ideal hash function:

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



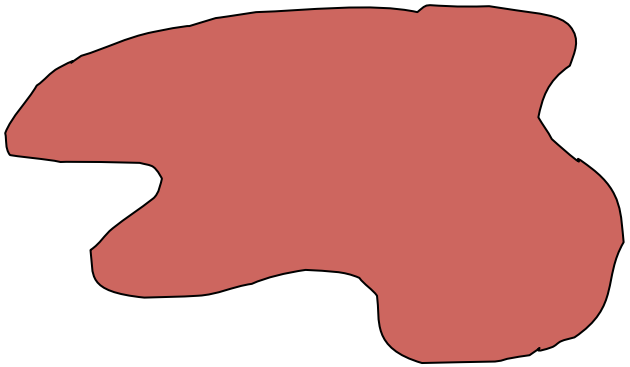
key space (e.g., integers, strings)

hash function:
index = h(key)



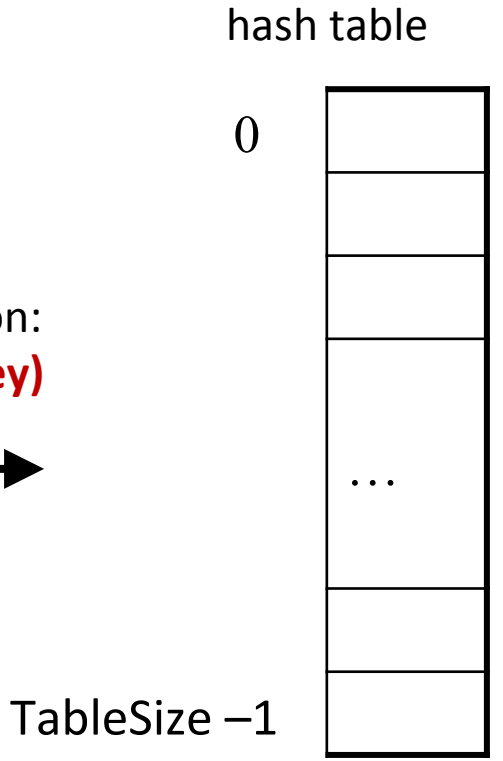
Using hash functions for Hash Tables

- Aim for constant-time (i.e., $O(1)$) **`find`**, **`insert`**, and **`delete`**
  - "On average" under some often-reasonable assumptions
- A hash table is an array of some fixed size
- Basic idea:

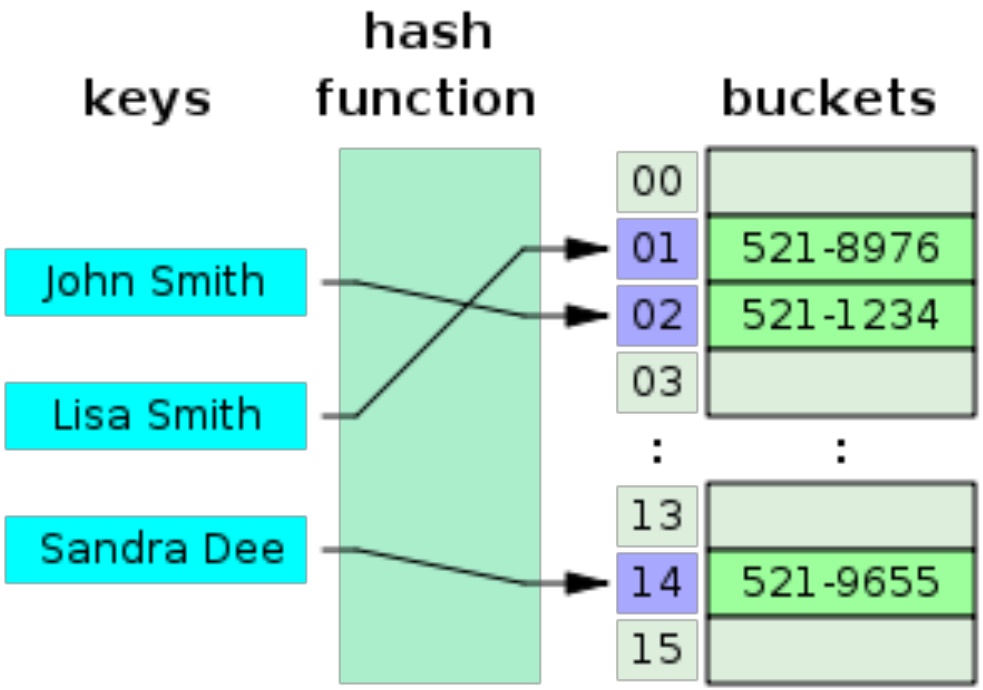


key space (e.g., integers, strings)

hash function:
index = h(key)



Example of Hash Table used for Dictionary of phone numbers



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* we follow good practices)
  - Balanced trees $O(\log n)$ worst-case

- Constant-time is better, right?
  - Yes, but you need "hashing to behave" (must avoid collisions)
  - Yes, but your data will not be stored in a sorted order
  - Yes, but **`findMin`**, **`findMax`**, **`predecessor`**, and **`successor`** go from $O(\log n)$ to $O(n)$, **`printSorted`** from $O(n)$ to $O(n \log n)$

Examples of Hash Tables

Really, any dictionary or collection that doesn't need to be sorted.

- 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

Review: Mod

Mod is the remainder function

To keep hashed values within the size of the table, we will generally do:

$$h(K) = \text{function}(K) \% \text{TableSize}$$

As a reference, useful properties of mod:

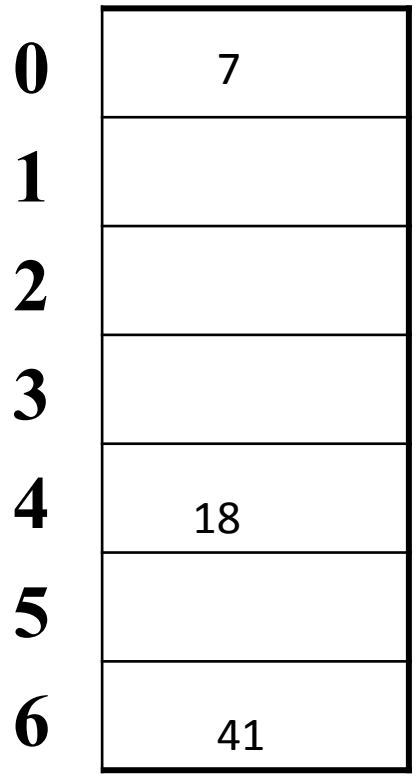
- $(a + b) \% c = [(a \% c) + (b \% c)] \% c$
- $(a\ b) \% c = [(a \% c)\ (b \% c)] \% c$
- $a \% c = b \% c \rightarrow (a - b) \% c = 0$

Example: Simple Integer Hash Function

- key space K = integers
- TableSize = 7

- h(K) = K % 7

- **Insert**: 7, 18, 41



Designing Hash Functions

Often based on **modular hashing**:

$$h(K) = f(K) \% P$$

P is typically the TableSize

P is often chosen to be prime:

- Reduces likelihood of collisions due to patterns in data
- Is useful for guarantees on certain hashing strategies (as we'll see)

Equivalent objects MUST hash to the same location

Hashing Objects in Java

- All Java objects contain the following method:

  ```
  public int hashCode()
  ```
  Returns an integer hash code for this object.

  – We can call `hashCode` on any object to find its preferred index.

- How is `hashCode` implemented?
  – Depends on the type of object and its state.
    - Example: a String's `hashCode` adds the ASCII values of its letters.
  – You can write your own `hashCode` methods in classes you write.
    - All classes come with a default version based on memory address.

Java String's hashCode

- The `hashCode` function inside `String` objects could look like this:

```
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < this.length(); i++) {
        hash = 31 * hash + this.charAt(i);
    }
    return hash;
}
```

  - As with any general hashing function, collisions are possible.
    - Example: "Ea" and "FB" have the same hash value.
  - Early versions of Java examined only the first 16 characters. For some common data this led to poor hash table performance.

Hashing your own Objects

For objects with several fields, usually best to have most of the "identifying fields" contribute to the hash to avoid collisions

Example:

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

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

- Bad idea(?): Use only first name
- Good idea(?): Use only middle initial? Combination of fields?
- Admittedly, what-to-hash-with is often unprincipled ☹

Note: Equal Objects MUST hash the same

- The Java library makes a very important assumption that clients must satisfy…

  If **a.equals(b)**, then we require

  `a.hashCode() == b.hashCode()`

- If you ever override equals
  - You need to override hashCode also in a consistent way
  - See CoreJava book, Chapter 5 for other "gotchas" with equals

Collisions

- **collision**: When hash function maps 2 values to same index.

```
h(k) = k % tableSize;
```

```
set.add(11);
```

index	0	1	2	3	4	5	6	7	8	9
value	0	0	0	0	0	0	0	0	0	0

Collisions

- **collision**: When hash function maps 2 values to same index.

```
h(k) = k % tableSize;
```

```
set.add(11);
set.add(49);
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	0	0	0	0	0	0

Collisions

- **collision**: When hash function maps 2 values to same index.

```
h(k) = k % tableSize;
```

```
set.add(11);
set.add(49);
set.add(24);
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	0	0	0	0	0	49

Collisions

- **collision**: When hash function maps 2 values to same index.

```
h(k) = k % tableSize;
```

```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	0	0	49

Collisions

- **collision**: When hash function maps 2 values to same index.

```
h(k) = k % tableSize;
```

```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	0	0	7	0	49

Collisions

- **collision**: When hash function maps 2 values to same index.

```
h(k) = k % tableSize;
```

```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);  // collides with 24!
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	54	0	0	7	0	49

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 should support collision resolution

– Ideas?

Probing

- **probing**: Resolving a collision by moving to another index.
  - **linear probing**: Moves to the next index.

```
set.add(11);
set.add(49);
set.add(24);
set.add(7);
set.add(54);  // collides with 24; must probe
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	**54**	0	7	0	49

  - Is this a good approach?
    - variation: **quadratic probing** moves increasingly far away

Linear Probing Example

- If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	/

Linear Probing Example

- If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Linear Probing Example

- If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	8
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Linear Probing Example

- If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	8
1	109
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Linear Probing Example

- If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	8
1	109
2	10
3	/
4	/
5	/
6	/
7	/
8	38
9	19

CSE 373: Data Structures & Algorithms

Hash Tables (Day 2)

Riley Porter

Winter 2017

Course logistics

- HW2 due tonight, HW1 grades out soon, but not in time for HW2
- HW3 out now, due in two weeks on the MIDTERM
- Midterm studying resources exist and are posted, but more (and what to expect) we'll get into next week

Where we left off on Wednesday

- hash tables:
  - a “magic” array that lets us do a find operation in O(1)
  - given a piece of data, use the hash function to find where the data goes
- hash function:
  - hashCode in Java, then map to the table size
  - usually involves primes and mod the table size
- collision resolution:
  - linear probing, just find the next empty spot
  - other options we’ll explore today

Open addressing

This is *one example* of open addressing

In general, open addressing means resolving collisions by trying a sequence of other positions in the table

Trying the next spot is called probing

- We just did linear probing
  - i^{th} probe was `(h(key) + i) % TableSize`
- In general have some probe function `f` and use `h(key) + f(i) % TableSize`

Open addressing does poorly with high load factor λ

- So want larger tables
- Too many probes means no more $O(1)$

Open Addressing Operations

`insert` finds an open table position using a probe function

What about **`find`**?

- Must use same probe function to "retrace the trail" for the data
- Unsuccessful search when reach empty position

What about **`delete`**?

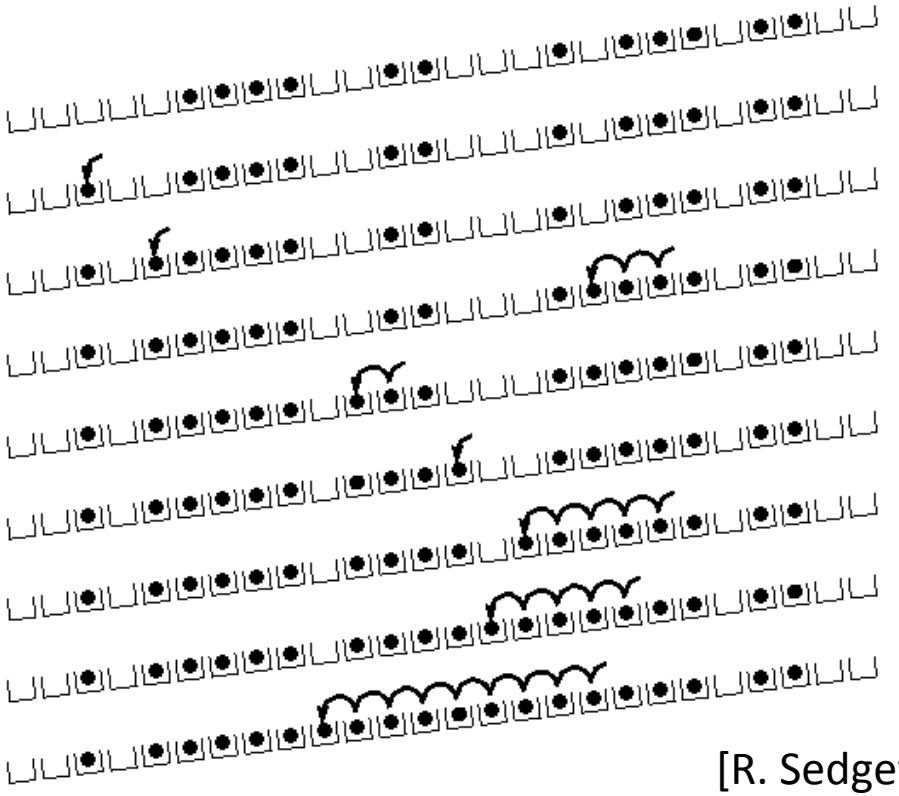
- ***Must*** use "lazy" deletion. Why?
  - Marker indicates "no data here, but don't stop probing"
- Note: **`delete`** with chaining is plain-old list-remove

(Primary) Clustering

It turns out linear probing is a *bad idea*, even though the probe function is quick to compute (which is a good thing)

Tends to produce *clusters*, which lead to long probing sequences

- Called primary clustering
- Saw this starting in our example



[R. Sedgewick]

Useful for Analysis: Load Factor

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

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$

← number of elements

Analysis of Linear Probing

- **Trivial fact**: For any $\lambda < 1$, linear probing will find an empty slot
  - It is "safe" in this sense: no infinite loop unless table is full

- **Non-trivial facts we won't prove**:
  Average # of probes given λ (in the limit as `TableSize` $\rightarrow \infty$)
  - Unsuccessful search:
  - Successful search:

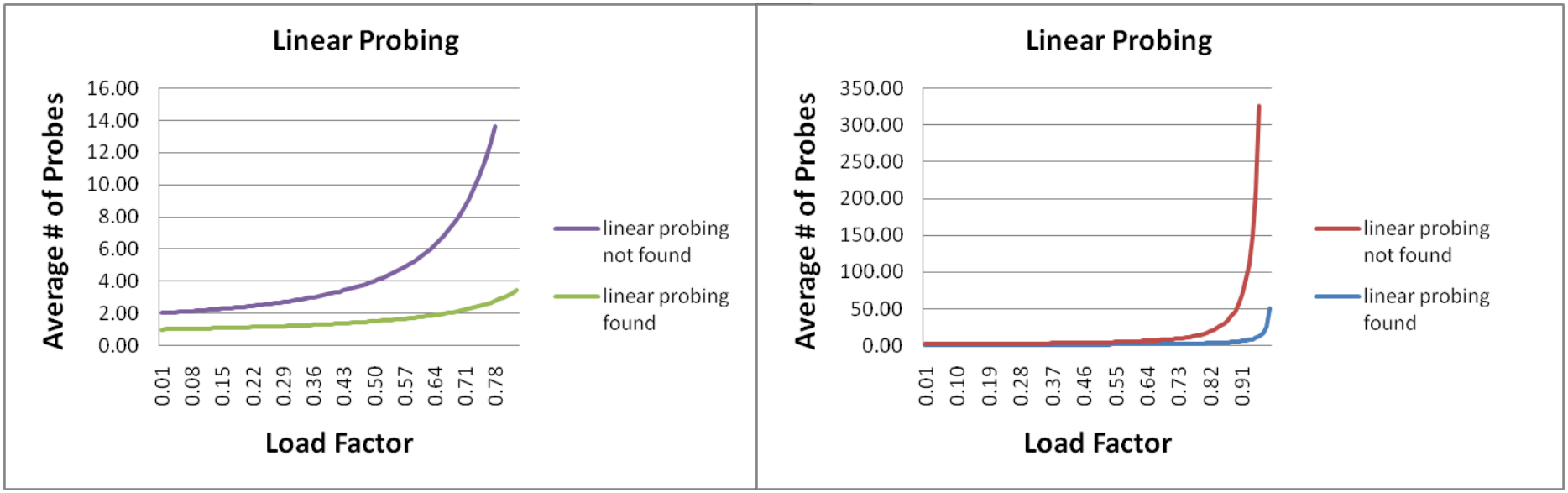
$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

- (Intuition) This is pretty bad: need to leave sufficient empty space in the table to get decent performance

In a chart

- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes “large table” but point remains)



Quadratic probing

- We can avoid primary clustering by changing the probe function

  `(h(key) + f(i)) % TableSize`

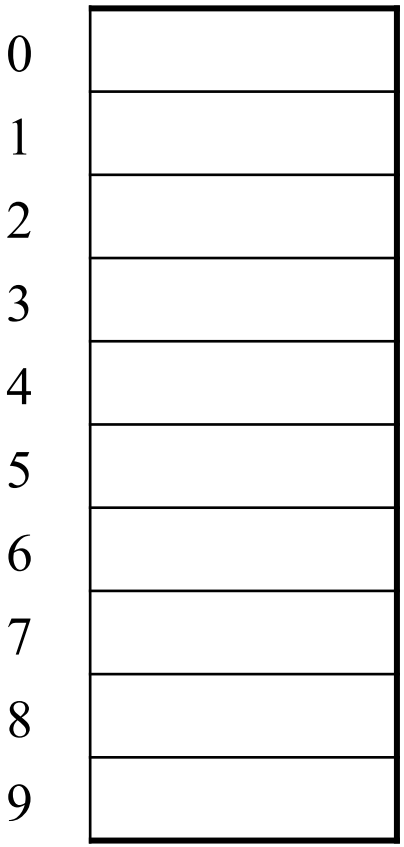
- A common technique is quadratic probing:

  `f(i) = i`^{2}

  – So probe sequence is:
  - 0th probe: **`h(key) % TableSize`**
  - 1st probe: **`(h(key) + 1) % TableSize`**
  - 2nd probe: **`(h(key) + 4) % TableSize`**
  - 3rd probe: **`(h(key) + 9) % TableSize`**
  - …
  - ith probe: **`(h(key) + i`**^{2}**`) % TableSize`**

- Intuition: Probes quickly "leave the neighborhood"

Quadratic Probing Example



TableSize=10

Insert:

89

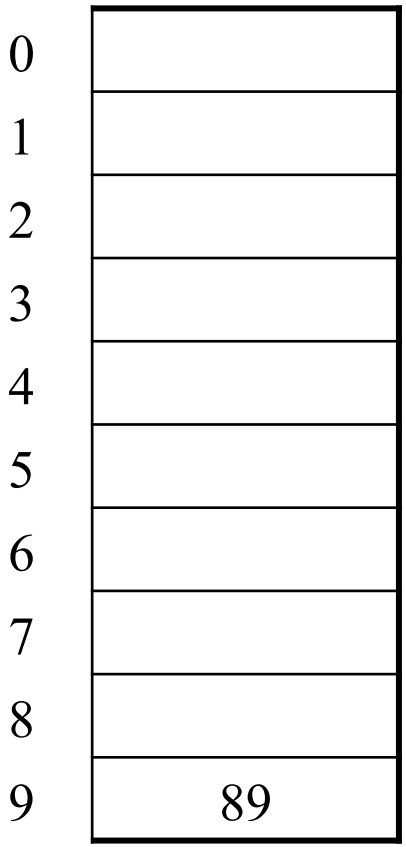
18

49

58

79

Quadratic Probing Example



TableSize=10

Insert:

89

18

49

58

79

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize=10

Insert:

89

18

49

58

79

49 % 10 = 9, 9 + i^2 % 10

49 % 10 = 9, 9 + (1 * 1) % 10

49 % 10 = 9 , 9 + (1 * 1) % 10 = 0

Quadratic Probing Example

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

TableSize=10
Insert:
89
18
49
58
79

58 % 10 = 8, 8 + i^2 % 10

58 % 10 = 8, 8 + (1 * 1) % 10

58 % 10 = 8 , 8 + (1 * 1) % 10 = 0

58 % 10 = 8, 8 + i^2 % 10

58 % 10 = 8, 8 + (2 * 2) % 10

58 % 10 = 8 , 8 + (2 * 2) % 10 = 2

Quadratic Probing Example

0	49
1	
2	58
3	79
4	
5	
6	
7	
8	18
9	89

TableSize=10
Insert:
89
18
49
58
79

79 % 10 = 9, 9 + i^2 % 10

79 % 10 = 9, 9 + (1 * 1) % 10

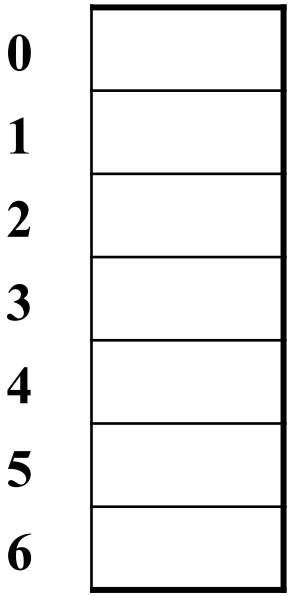
79 % 10 = 9 , 9 + (1 * 1) % 10 = 0

79 % 10 = 9, 9 + i^2 % 10

79 % 10 = 9, 9 + (2 * 2) % 10

79 % 10 = 9 , 9 + (2 * 2) % 10 = 3

Another Quadratic Probing Example

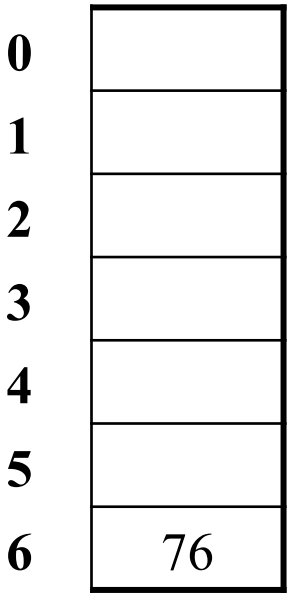


TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example

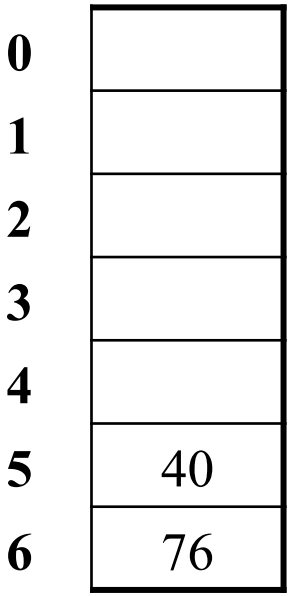


TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example

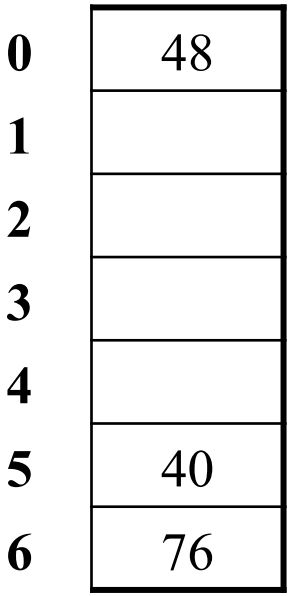


TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Another Quadratic Probing Example



TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

48 % 7 = 6, $6 + i^2 \% 7$

48 % 7 = 6, 6 + (1 * 1) % 7

48 % 7 = 6 , 6 + (1 * 1) % 7 = 0

Another Quadratic Probing Example

0	48
1	
2	5
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

5 % 7 = 5, $5 + i^2$ % 7

5 % 7 = 5, 5 + (1 * 1) % 7

5 % 7 = 5 , 5 + (1 * 1) % 7 = 0

5 % 7 = 5, $5 + i^2$ % 7

5 % 7 = 5, 5 + (2 * 2) % 7

5 % 7 = 5 , 5 + (2 * 2) % 7 = 2

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

55 % 7 = 6, 6 + i^2 % 7

55 % 7 = 6, 6 + (1 * 1) % 7

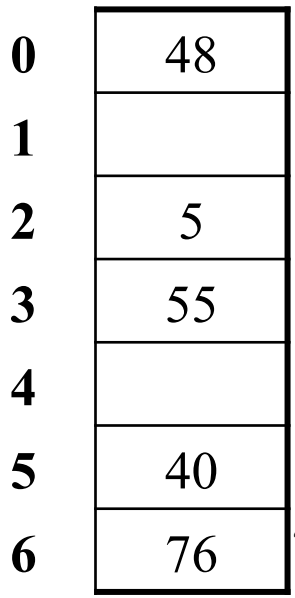
55 % 7 = 6 , 6 + (1 * 1) % 7 = 0

55 % 7 = 6, 6 + i^2 % 7

55 % 7 = 6, 6 + (2 * 2) % 7

55 % 7 = 6 , 6 + (2 * 2) % 7 = 3

Another Quadratic Probing Example



TableSize = 7

Insert:

76 (76 % 7 = 6)

40 (40 % 7 = 5)

48 (48 % 7 = 6)

5 (5 % 7 = 5)

55 (55 % 7 = 6)

47 (47 % 7 = 5)

47 % 7 = 5, 5 + i^2 % 7

47 % 7 = 5, 5 + (1 * 1) % 7

47 % 7 = 5 , 5 + (1 * 1) % 7 = 0

47 % 7 = 5, 5 + i^2 % 7

47 % 7 = 5, 5 + (2 * 2) % 7

47 % 7 = 5 , 5 + (2 * 2) % 7 = 2

47 % 7 = 5, 5 + i^2 % 7

47 % 7 = 5, 5 + (3 * 3) % 7

47 % 7 = 5 , 5 + (3 * 3) % 7 = 0

47 % 7 = 5, 5 + i^2 % 7

47 % 7 = 5, 5 + (4 * 4) % 7

47 % 7 = 5 , 5 + (4 * 4) % 7 = 0

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	(5 % 7 = 5)
55	(55 % 7 = 6)
47	(47 % 7 = 5)

Doh!: For all *n*, `((n*n) +5) % 7 is 0, 2, 5, or 6`

- Excel shows takes "at least" 50 probes and a pattern
- Proof uses induction and $(n^2+5) \% 7 = ((n-7)^2+5) \% 7$
  - In fact, for all *c* and *k*, $(n^2+c) \% k = ((n-k)^2+c) \% k$

From Bad News to Good News

- **Bad news**:
  - Quadratic probing can cycle through the same full indices, never terminating despite table not being full
- **Good news**:
  - If `TableSize` is *prime* and $\lambda < ½$, then quadratic probing will find an empty slot in at most `TableSize/2` probes
  - So: If you keep $\lambda < ½$ and `TableSize` is *prime*, no need to detect cycles
  - Optional
    - Key fact: For prime `T` and `0 < i,j < T/2` where `i ≠ j`, `(k + i²) % T ≠ (k + j²) % T` (i.e., no index repeat)

Clustering reconsidered

- Quadratic probing does not suffer from primary clustering: no problem with keys initially hashing to the same neighborhood

- But it's no help if keys initially hash to the same index
  - Called secondary clustering

- Can avoid secondary clustering with a probe function that depends on the key: double hashing…

Double hashing

Idea:

- Given two good hash functions *h* and *g*, it is very unlikely that for some *key*, `h(key) == g(key)`
- So make the probe function `f(i) = i*g(key)`

Probe sequence:

- 0th probe: `h(key) % TableSize`
- 1st probe: `(h(key) + g(key)) % TableSize`
- 2nd probe: `(h(key) + 2*g(key)) % TableSize`
- 3rd probe: `(h(key) + 3*g(key)) % TableSize`
- …
- ith probe: `(h(key) + i*g(key)) % TableSize`

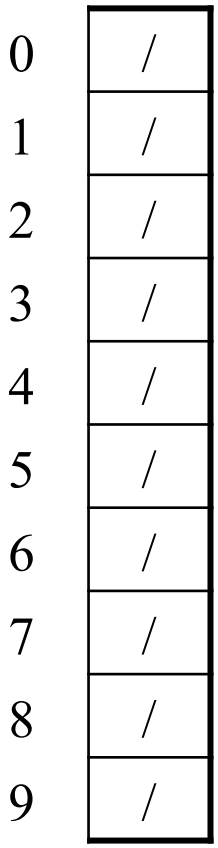
Detail: Make sure `g(key)` cannot be `0`

Double-hashing analysis

- Intuition: Because each probe is "jumping" by **`g(key)`** each time, we "leave the neighborhood" *and* "go different places from other initial collisions"

- But we could still have a problem like in quadratic probing where we are not "safe" (infinite loop despite room in table)
  - It is known that this cannot happen in at least one case:
    - **`h(key) = key % p`**
    - **`g(key) = q - (key % q)`**
    - **`2 < q < p`**
    - **`p`** and **`q`** are prime

Separate Chaining



Chaining:

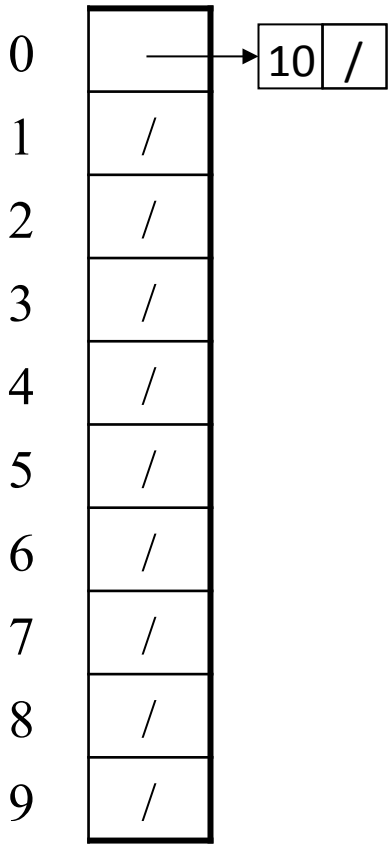
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

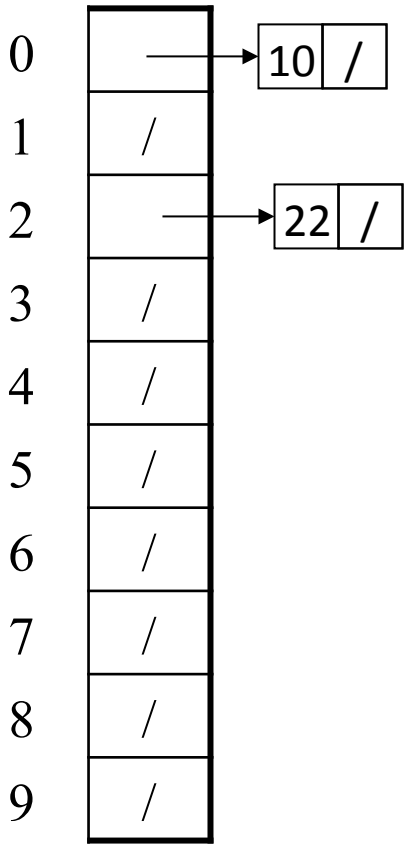
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

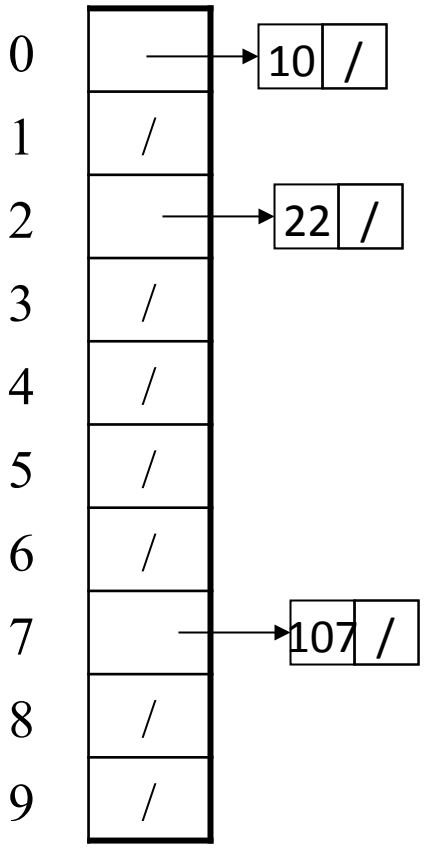
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

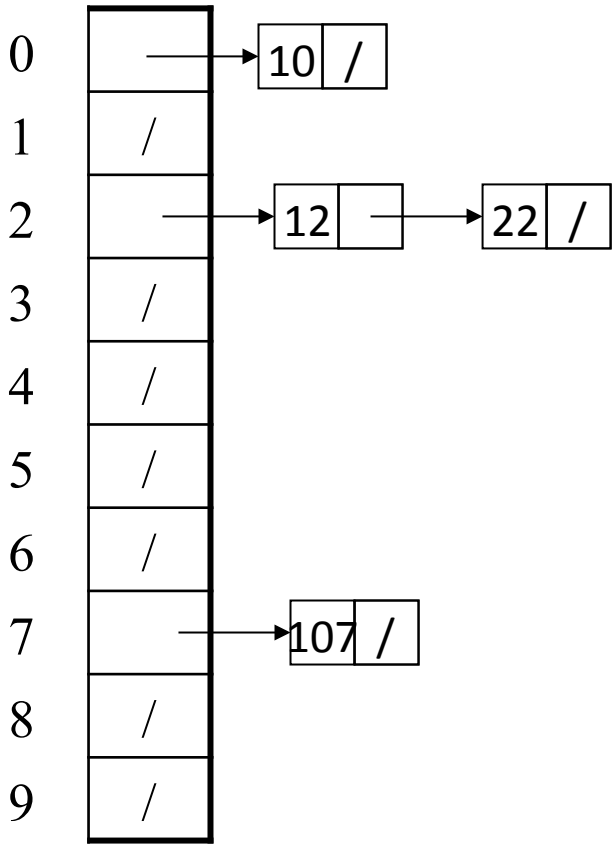
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

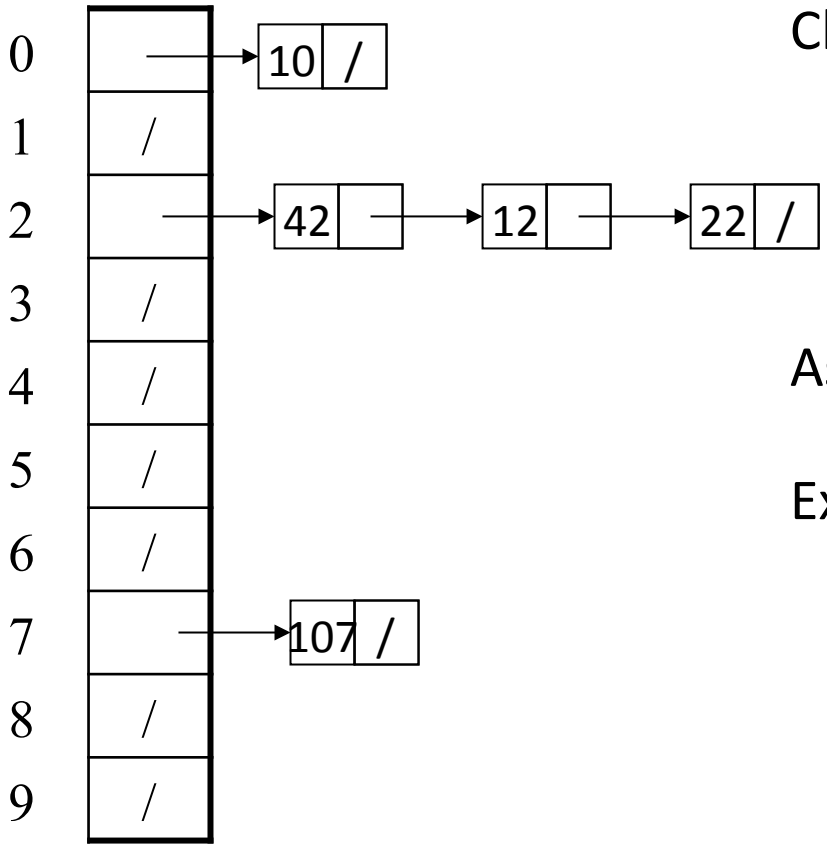
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

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 Analysis

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

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$

← 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

So we like to keep λ fairly low (e.g., 1 or 1.5 or 2) for chaining

Rehashing

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything
- With chaining, we get to decide what "too full" means
  - Keep load factor reasonable (e.g., < 1)?
  - Consider average or max size of non-empty chains?
- For open addressing, half-full is a good rule of thumb
- New table size
  - Twice-as-big is a good idea, except, uhm, that won't be prime!
  - So go *about* twice-as-big
  - Can have a list of prime numbers in your code since you won't grow more than 20-30 times

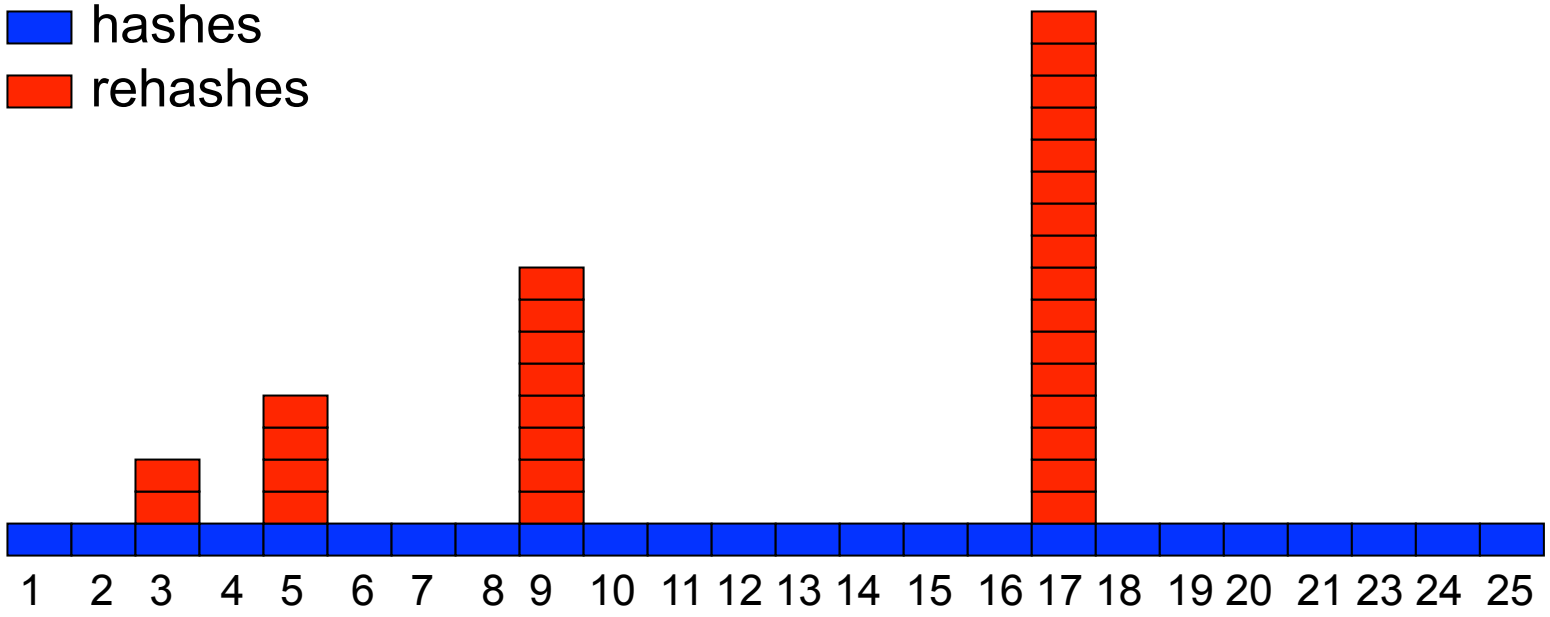
Rehashing

When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - Separate chaining: full ($\lambda = 1$)
  - Open addressing: half full ($\lambda = 0.5$)
  - When an insertion fails
  - Some other threshold
- Cost of a single rehashing?

Rehashing Picture

- Starting with table of size 2, double when load factor > 1.



Amortized Analysis of Rehashing

- Cost of inserting n keys is < 3n
- suppose $2^k + 1 \leq n \leq 2^{k+1}$
  - Hashes = n
  - Rehashes = $2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2$
  - Total = $n + 2^{k+1} - 2 < 3n$

- Example
  - n = 33, Total = 33 + 64 –2 = 95 < 99

Terminology

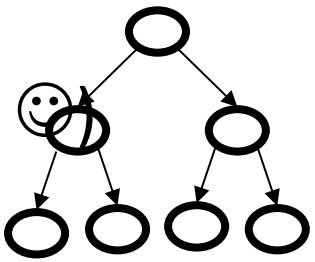
We and the book use the terms

- "chaining" or "separate chaining"
- "open addressing"

Very confusingly,

- "open hashing" is a synonym for "chaining"
- "closed hashing" is a synonym for "open addressing"

(If it makes you feel any better,
most trees in CS grow upside-down



Summary: Hashing

- Hashing is one of the most important tools.
- Hashing has many applications where operations are limited to find, insert, and delete.
  - But what is the cost of doing, e.g., findMin?
- Can use:
  - Separate chaining (easiest)
  - Open addressing (memory conservation, no linked list management)
- Rehashing has good amortized complexity.
- Also has a big data version to minimize disk accesses: extendible hashing. (See book.)

Summary: Hash Tables

- Aim for constant-time (i.e., $O(1)$) **`find`**, **`insert`**, and **`delete`**
  - "On average" under some reasonable assumptions
- A hash table is an array of some fixed size