Review poll

In a Map backed by a balanced binary search tree, what is the worst-case time complexity of the get() method? Express your answer in terms of N, the number of entries in the Map.

a) O(1)

b) O(log N)

c) O(N)

d) $O(N^2)$

e) None of the above

PollEv.com/leahp

text **leahp** to **22333**

errata: this slide initially said “balanced binary tree” in error and has been updated to say “balanced binary search tree”

Lecture 20: Hashing

CS 2110, Matt Eichhorn and Leah Perlmutter

November 4, 2025

Announcements

- Prelim 2 on Thursday
- Today is an election day. Vote if you are able!
  - If voting in NY state, you can look up your polling place: voterlookup.elections.ny.gov/
  - Polls open 6am to 9pm in Tompkins County

Roadmap

Java, Complexity, OOP

- start– 9/30

ADTs I

- List, Stack, Queue, Iteration
  - 10/2 – 10/16

ADTs II

- Trees
  - Tues 10/21, Thurs 10/23, Thurs 10/28
- Set and Map
  - Thurs 10/30
- **Hash Tables**
  - **Tues 11/4**
- Graphs
  - Tues 11/6, Thurs 11/11, Tues 11/13

Beyond ADTs

- Graphical User Interfaces, Parallel Programming
  - 11/18 – end of semester

Motivation for Hashing

Review poll

In a Map backed by a balanced binary search tree, what is the worst-case time complexity of the get() method? Express your answer in terms of N, the number of entries in the Map.

a) O(1)

b) O(log N)

c) O(N)

d) $O(N^2)$

e) None of the above

PollEv.com/leahp

text **leahp** to **22333**

Answer: O(log N). Balanced BST guarantees the height will be at most log N. To get an element, we do at most one operation per level of the tree.

errata: this slide initially said “balanced binary tree” in error and has been updated to say “balanced binary search tree”

Review

Sets

- unordered, unique elements
- contains, add, remove, size

Maps

- associations between keys and values
- key/value pair is called an entry
- put, get, remove, containsKey, size

Set/Map Performance

Representation	Contains	Put	Get	Size
Unordered List	O(N)	O(N)	O(N)	O(1)
Ordered List	O(log N)	O(N)	O(N)	O(1)
Balanced BST	O(log N)	O(log N)	O(log N)	O(1)
???	O(1)*	O(1)*	O(1)*	O(1)

errata: this slide initially said “Balanced Tree” in error and has been updated to say “Balanced BST”

Hashing: The Big Idea

What if we could store our elements in an array? (Hash Table)

- With array indexing, we get O(1) access
- Allow the array to have gaps, so we don't need to move elements over when we add or remove
- Use an algorithm to calculate the index each element goes at

  If element X is in the array, we know it belongs at index K

  (hashing function)
- Caveat: What do we do if two elements end up at the same index? (collisions)
- Consideration: How big should the array be compared to the number of elements to minimize likelihood of collisions?

Overview of today

Hashing

- Motivation ✅
- Hash Table Basics
- Chaining
- Linear Probing

Hash Table Basics

Example Hash Set

Hash function h for `String` keys

- h(“Turing”) -> 3
- h(“von Neumann”) -> 7
- h(“Liskov”) -> 5

Index	Element
0	null
1	null
2	null
3	Turing
4	null
5	Liskov
6	null
7	von Neumann

Example Hash Map

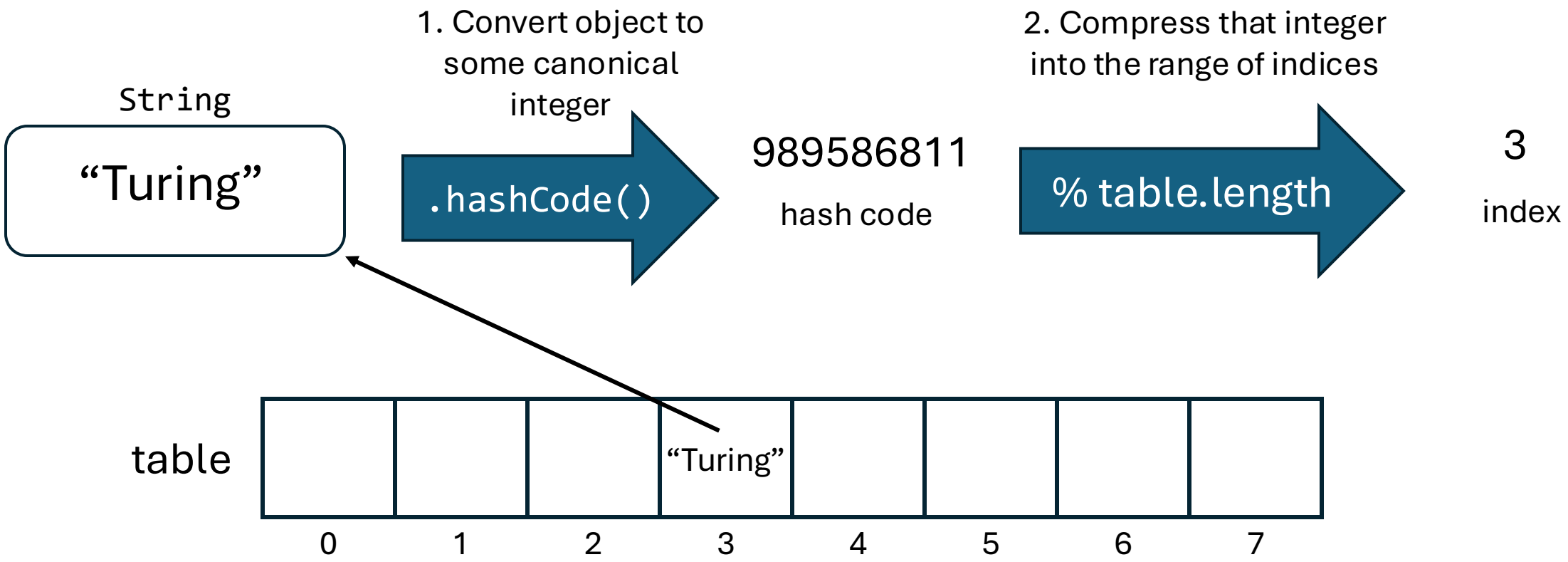
Hash function h for `String` keys

- h(“Turing”) -> 3
- h(“von Neumann”) -> 7
- h(“Liskov”) -> 5

Index	Element
0	null
1	null
2	null
3	(Turing, 1912-06-23)
4	null
5	(Liskov, 1939-11-07)
6	null
7	(von Neumann, 1903-12-28)

Hash codes and indices

"Hashing" an object to an index is a two-step process:



The `hashCode()` method

`Object` defines a `hashCode()` method that returns an `int`

- Any Java object can be used as a key
- (Though, many of them probably shouldn't be…)

Hash code must be consistent with equality

- If `x.equals(y)`, then `x.hashCode() == y.hashCode()`
- `Override equals()` and `hashCode()` as a pair
- `Object` defaults to using memory address for both, which is consistent

Good `hashCode()`s

- Goal: two non-equal objects should be *unlikely* to share a hash code
  - Should depend on *all* of an object's state
  - Should depend on *ordering* of any sequential state (e.g. arrays)
  - Outputs should span whole range of valid java integers

- When an object has multiple fields used to determine equality with other objects, its hash code should be based on all of them
  - `Objects.hash()`, `Arrays.hashCode()` can help

- When analyzing performance, we will assume `hashCode()` is O(1)
  - Long strings, data tables would not make performant keys

Good hash function

A good `hashCode()` function returns values distributed over the entire range of ints, even for inputs that look very similar.

Which of the following approaches is mostly likely to yield a good hash code function?

PollEv.com/leahp
text **`leahp`** to **`22333`**

A. Given any `Object`, generate a random number between -2^-31 and 2^31-1

B. Given a `String`, return its length

C. Given a `String`, return the Unicode codepoint of its last letter

D. Given an `int`, return its remainder when divided by 43 (a prime)

E. Given an `int`, add it to an arbitrary large constant, then multiply by a large prime number

Good hash function

A good `hashCode()` function returns values distributed over the entire range of ints, even for inputs that look very similar.

Which of the following approaches is mostly likely to yield a good hash code function?

PollEv.com/leahp
text **`leahp`** to **`22333`**

A. Given any `Object`, generate a random number between -2^-31 and 2^31-1

B. Given a `String`, return its length

C. Given a `String`, return the Unicode codepoint of its last letter

D. Given an `int`, return its remainder when divided by 43 (a prime)

E. Given an `int`, add it to an arbitrary large constant, then multiply by a large prime number

Simple Uniform Hashing Assumption

The *Simple Uniform Hashing Assumption* (SUHA) asserts that for a given hash function $h_C : \mathsf{T} \to \{0, 1, \ldots, C-1\}$, each object inserted in the hash table is equally likely to be placed in any of its buckets, and its placement is not affected by the objects already in the hash table.

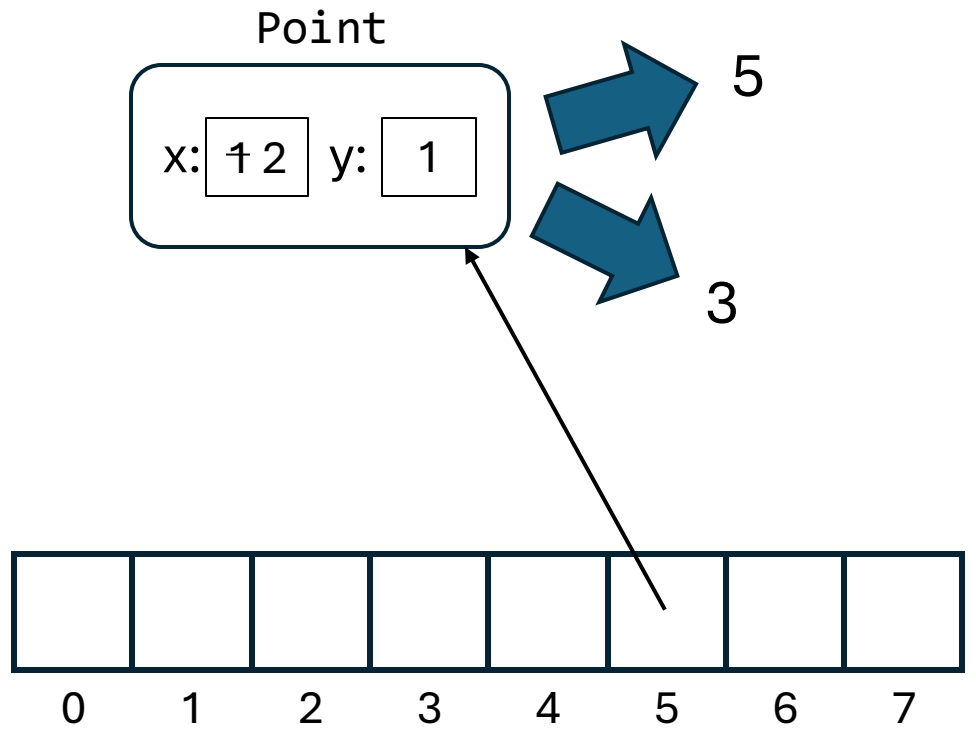
More concretely, for a given object x of type T, $\Pr\left(h_C(x) = i\right) = \frac{1}{C}$ for all $i \in \{0, 1, \ldots, C-1\}$, and these probabilities are *mutually independent* from the hash values of all other objects.

Warning: Keys should be Immutable

- If an object's `hashCode()` depends on its mutable state, don't use it as the key type in a `HashMap` or element type in a `HashSet`.

```
Map<Point,String> map = new HashMap<>();
Point p = new Point(1,1);
map.put(p,"a");
p.shiftRight(); // mutate x-coordinate
System.out.println(map.containsKey(p));
```

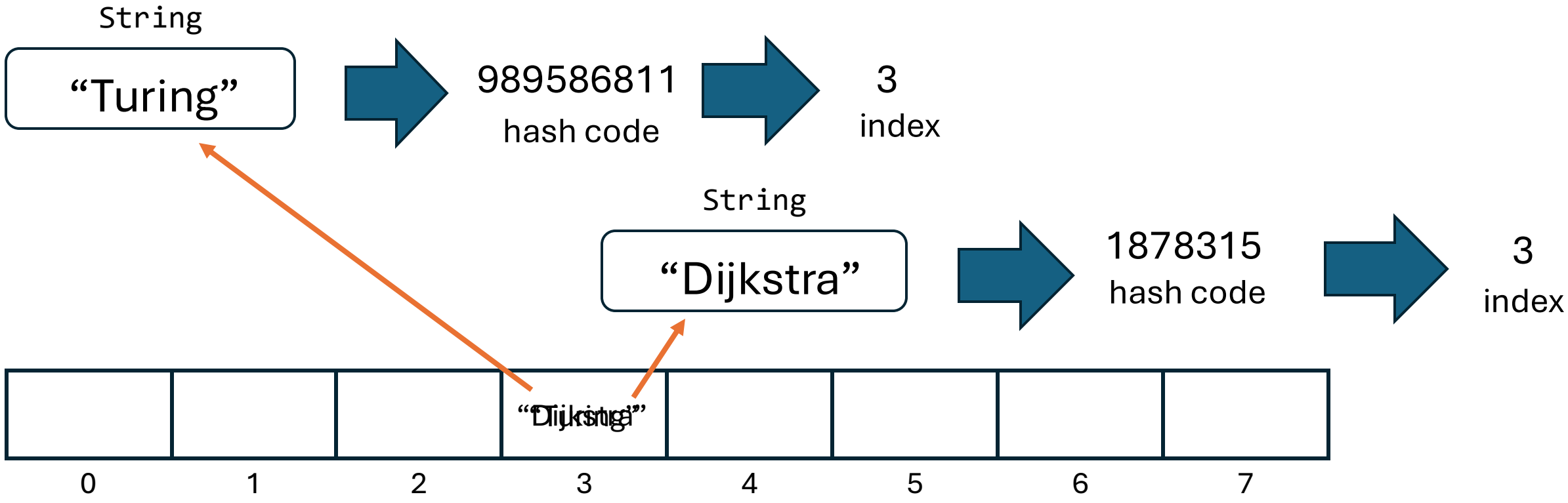
```
>  false
```



Collisions

How can we resolve collisions?

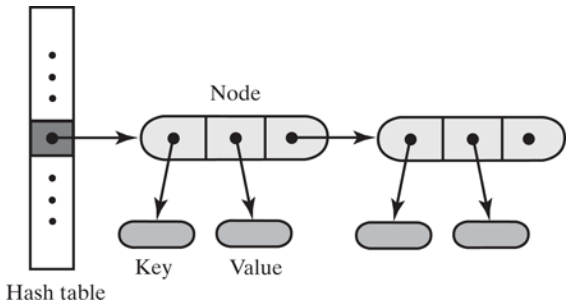
- Different objects might be “hashed” to the same index.
  - Could occur if hashCodes ( )are the same, or have the same remainder (so compress to the same index)



Collision resolution approaches

Chaining

- Treat array elements as "buckets" storing a *collection* of entries (e.g. a linked list)
- Finding the right bucket is O(1), but searching it will be slower



Probing

- Array elements point directly to entries
- If desired index is occupied, pick the next index to try according to a probing sequence

Overview of today

Hashing

- Motivation ✅
- Hash Table Basics ✅
- Chaining
- Linear Probing

Chaining

HashSet

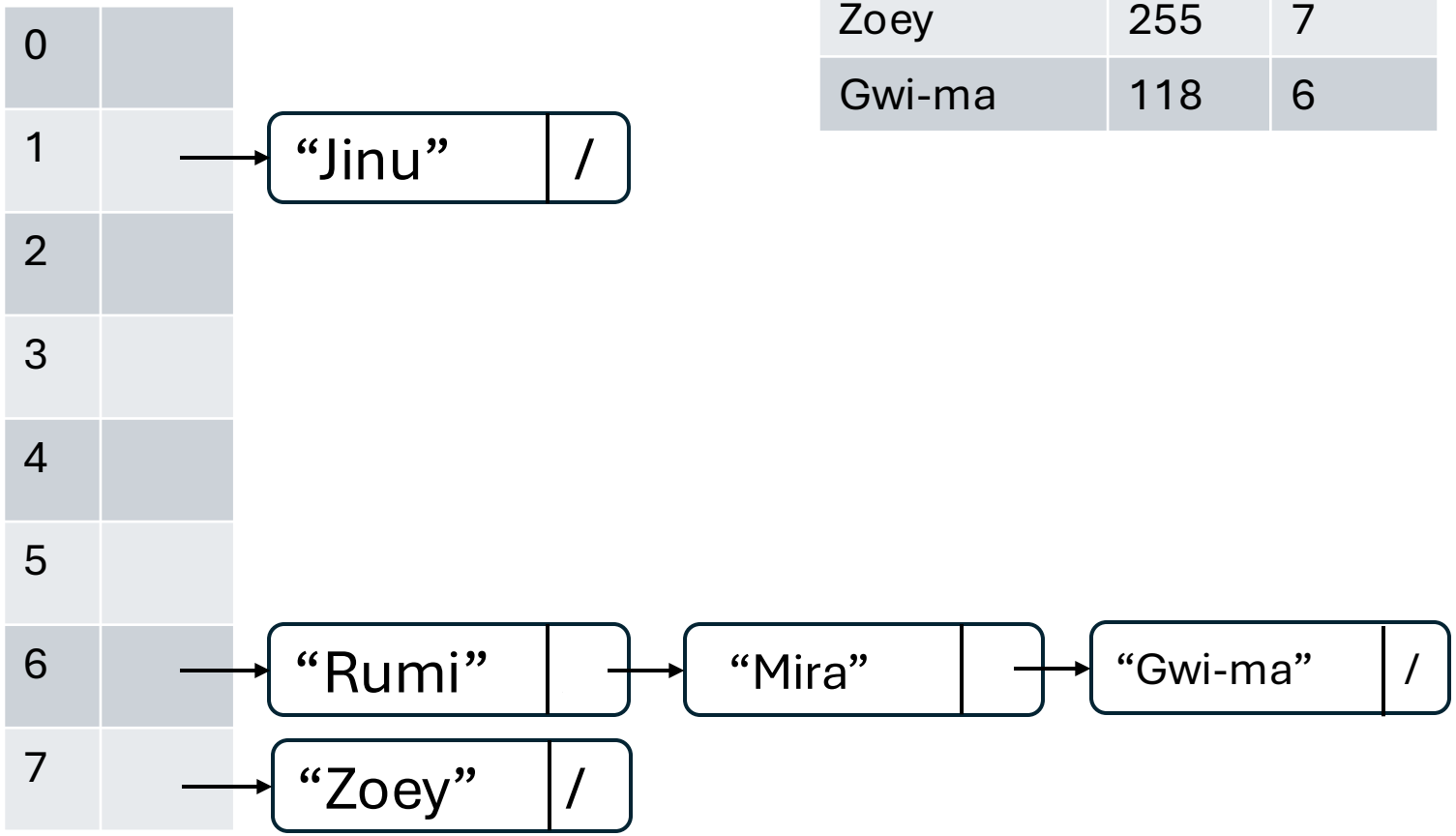
Chaining:

Key	Hash code	Index (%8)
Rume	126	6
Jinu	97	1
Mira	86	6
Zoey	255	7
Gwi-ma	118	6

`put(k,v)`

1. Compute `h = k.hashCode()`
2. Compute `i = h % table.length`
3. Search bucket i for entry with key k
   - Update value v if present
   - Add entry (k,v) if not present

Similar for `get(k)` and `remove(k)`



Chaining:

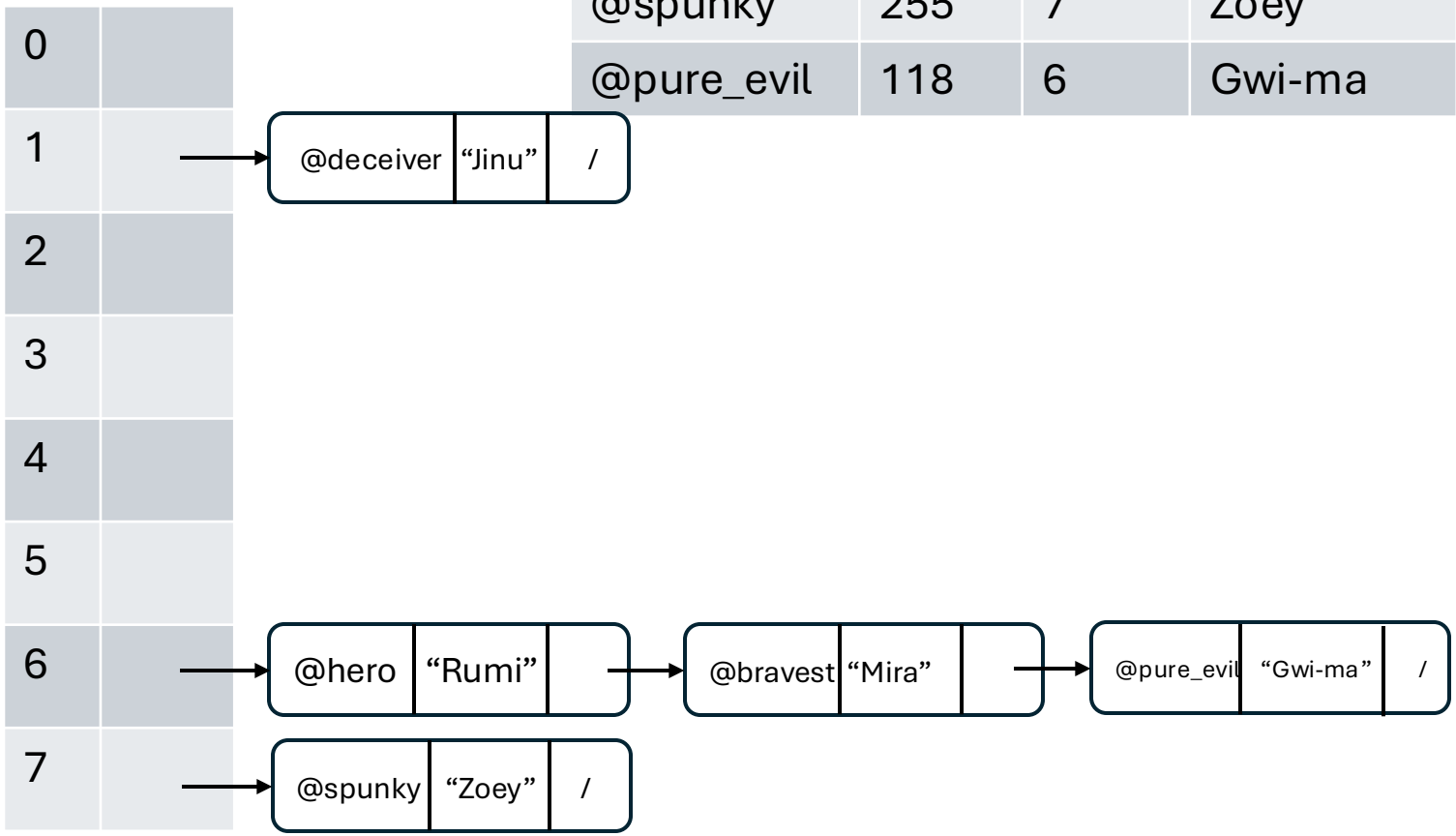
HashMap

Key (handle)	Hash code	Index (%8)	Value (name)
@hero	126	6	Rumi
@deceiver	97	1	Jinu
@bravest	86	6	Mira
@spunky	255	7	Zoey
@pure_evil	118	6	Gwi-ma

`put(k,v)`

1. Compute `h = k.hashCode()`
2. Compute `i = h % table.length`
3. Search bucket i for entry with key k
   - Update value v if present
   - Add entry (k,v) if not present

Similar for `get(k)` and `remove(k)`



Chaining code for HashSet

```
/** A Set implementation backed by a chaining hash table. */
public class HashSet<T> implements Set<T> {

  /** The backing storage of this HashSet. */
  private LinkedList<T>[] buckets;
  ...

  /** Returns the hash value of the given `elem` */
  private int index(T elem) {
    return Math.abs(elem.hashCode() % buckets.length);
  }

  @Override
  public boolean contains(T elem) {
    assert elem != null;
    return buckets[index(elem)].contains(elem);
  }
  ...
}
```

What is the worst-case time complexity of contains() ?

PollEv.com/leahp
text **leahp** to **22333**

Chaining code for HashSet

What is the worst-case time complexity of contains() ?

A. O(1)
B. O(L) where L is the maximum chain length
C. O(C) where C is the number of buckets
D. O(N) where N is the number of elements in the set
E. Something else

Chaining code for HashSet

What is the worst-case time complexity of contains() ?

A. O(1)

B. O(L) where L is the maximum chain length

C. O(C) where C is the number of buckets

D. O(N) where N is the number of elements in the set

E. Something else

B is correct because it takes O(1) time to find the correct array index, then we have to traverse the chain at that index, which is at worst O(L) nodes long. D is also correct because in the degenerate case, where all elements are stored at the same array index, O(N) is equal to O(L).

Load factor and performance

- Load Factor is the average chain length

$$\lambda = \frac{N}{C} = \frac{\text{number of elements}}{\text{number of buckets}}$$

- Load Factor λ is the *expected* runtime of hash table operations that need to traverse chains.
- When λ <= 1, expected runtime for hash table operations is O(1)
- If N >> C, then λ approaches N and runtime approaches O(N)

Set/Map Performance

Representation	Contains	Put	Get	Size
Unordered List	O(N)	O(N)	O(N)	O(1)
Ordered List	O(log N)	O(N)	O(N)	O(1)
Balanced BST	O(log N)	O(log N)	O(log N)	O(1)
???	O(1)*	O(1)*	O(1)*	O(1)

errata: this slide initially said “Balanced Tree” in error and has been updated to say “Balanced BST”

Set/Map Performance

Representation	Contains	Put	Get	Size
Unordered List	O(N)	O(N)	O(N)	O(1)
Ordered List	O(log N)	O(N)	O(N)	O(1)
Balanced BST	O(log N)	O(log N)	O(log N)	O(1)
Hash Table with λ close to 1	O(1)*	O(1)*	O(1)*	O(1)

* = expected runtime

errata: this slide initially said “Balanced Tree” in error and has been updated to say “Balanced BST”

Load factor and resizing

- When load factor gets too big, we expand our array
- Recall that in our hashing function, we did modular arithmetic with the *length of the array*
- When array length changes, we need to recompute indices for all elements

Overview of today

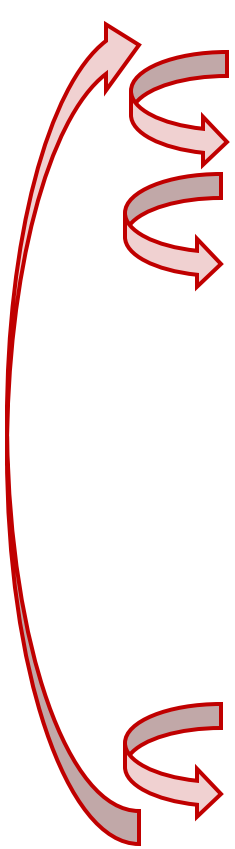
Hashing

- Motivation ✅
- Hash Table Basics ✅
- Chaining ✅
- Linear Probing

Linear Probing

HashSet

Linear probing: add(k)



0	Zoey
1	Jinu
2	Gwi-ma
3	
4	
5	
6	Rumi
7	Mira

Key	Hash code	Index (%8)
Rume	126	6
Jinu	97	1
Mira	86	6
Zoey	255	7
Gwi-ma	118	6

HashSet

Linear probing: `contains(k)`

0	Zoey
1	Jinu
2	Gwi-ma
3	
4	
5	
6	Rumi
7	Mira

Key	Hash code	Index (%8)
Rume	126	6
Jinu	97	1
Mira	86	6
Zoey	255	7
Gwi-ma	118	6
Derpy	105	1

```
contains("Jinu")

contains("Zoey")

contains("Derpy")
```

HashSet

Linear probing: `remove(k)`

Key	Hash code	Index (%8)
Rume	126	6
Jinu	97	1
Mira	86	6
Zoey	255	7
Gwi-ma	118	6

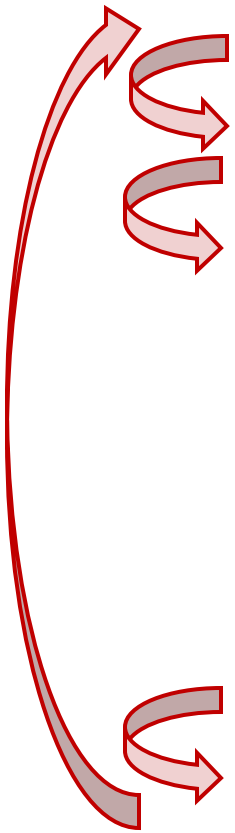
0	Zoey
1	~~Jinu~~ 
2	Gwi-ma
3	
4	
5	
6	Rumi
7	Mira

```
remove("Jinu")

contains("Gwi-ma")
```

A tombstone indicates something has been removed and is used to prevent breaking probing sequences.

Linear probing: `put(k,v)`



0	(@spunky, Zoey)
1	(@deceiver, Jinu)
2	(@pure_evil, Gwi-ma)
3	
4	
5	
6	(@hero, Rumi)
7	(@bravest, Mira)

Key (handle)	Hash code	Index (%8)	Value (name)
@hero	126	6	Rumi
@deceiver	97	1	Jinu
@bravest	86	6	Mira
@spunky	255	7	Zoey
@pure_evil	118	6	Gwi-ma

Linear probing: `contains(k)`

0	(@spunky, Zoey)
1	(@deceiver, Jinu)
2	(@pure_evil, Gwi-ma)
3	
4	
5	
6	(@hero, Rumi)
7	(@bravest, Mira)

Key (handle)	Hash code	Index (%8)	Value (name)
@hero	126	6	Rumi
@deceiver	97	1	Jinu
@bravest	86	6	Mira
@spunky	255	7	Zoey
@pure_evil	118	6	Gwi-ma
@kitty	105	1	Derpy

```
contains(@deceiver)

contains(@spunky)

contains(@kitty)
```

Linear probing: `remove(k)`

0	(@spunky, Zoey)
1	~~(@deceiver, Jinu)~~ ?
2	(@pure_evil, Gwi-ma)
3	
4	
5	
6	(@hero, Rumi)
7	(@bravest, Mira)

Key (handle)	Hash code	Index (%8)	Value (name)
@hero	126	6	Rumi
@deceiver	97	1	Jinu
@bravest	86	6	Mira
@spunky	255	7	Zoey
@pure_evil	118	6	Gwi-ma
@kitty	105	1	Derpy

```
remove(@deceiver)

contains(@pure_evil)
```

A tombstone indicates something has been removed and is used to prevent breaking probing sequences.

Chaining vs. probing

Chaining

- …

Probing

- …

Chaining vs. probing

Chaining

- Less sensitive to load factor
- Accommodates deletions better

- Requires more memory (at same load factor)

Probing

- More sensitive to load factor; vulnerable to “clustering”
- Deletions deteriorate performance

- Requires less memory (at same load factor)

Overview of today

Hashing

- Motivation ✅
- Hash Table Basics ✅
- Chaining ✅
- Linear Probing ✅

Metacognition

- Take 1 minute to write down a brief summary of what you have learned today

closing announcements to follow…

Announcements

- Prelim 2 on Thursday
- Today is an election day. Vote if you are able!