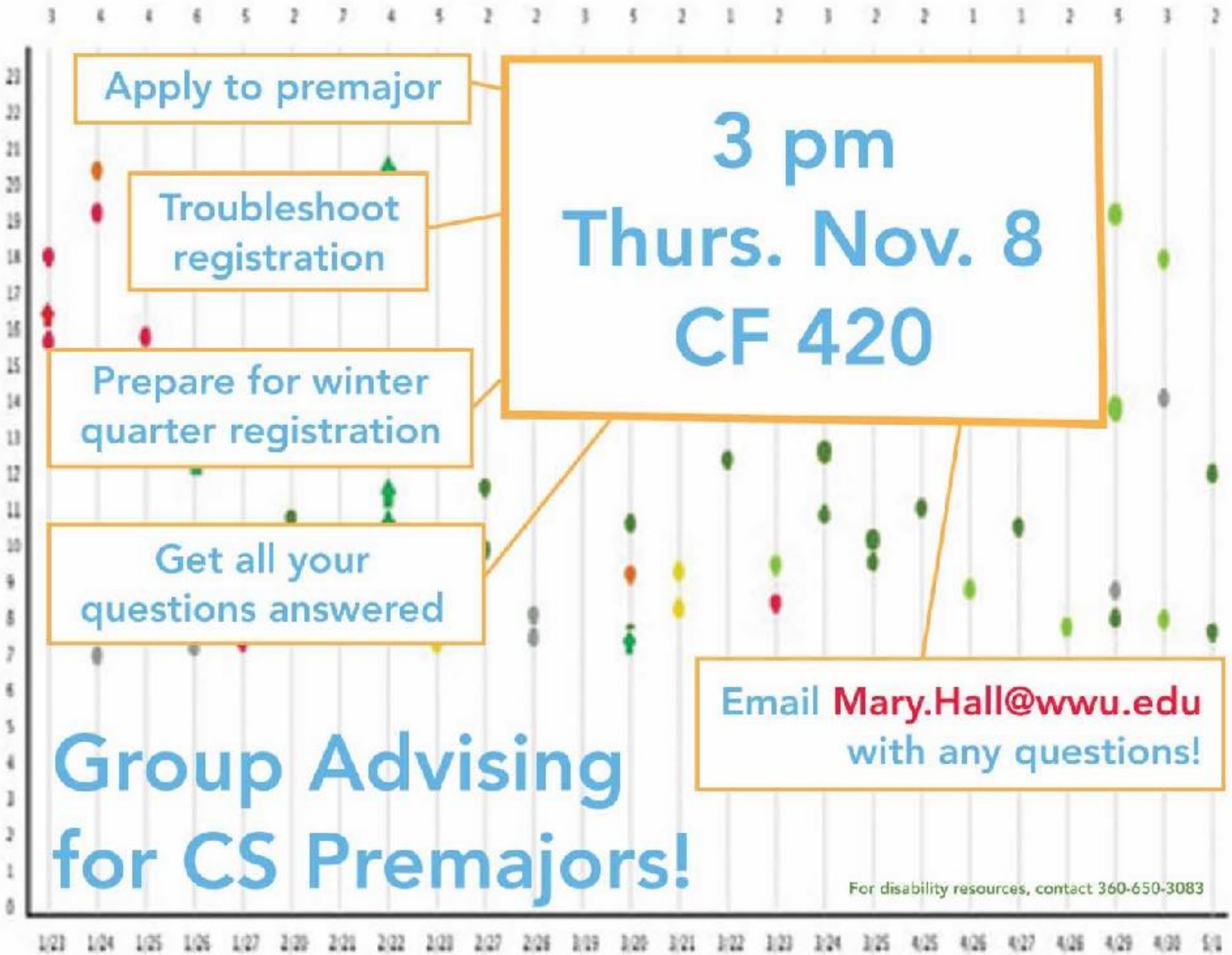
CSCI 241

Lecture 18

Map ADT, Rehashing, Open Addressing


Apply to premajor
Troubleshoot registration
Prepare for winter quarter registration
Get all your questions answered
3 pm
Thurs. Nov. 8
CF 420
Email Mary.Hall@wwu.edu with any questions!
Group Advising for CS Premajors!
For disability resources, contact 360-650-3083

MIX IT UP

INTERSECTIONS BETWEEN THE ARTS & SCIENCES

FEATURING

SAMMUS

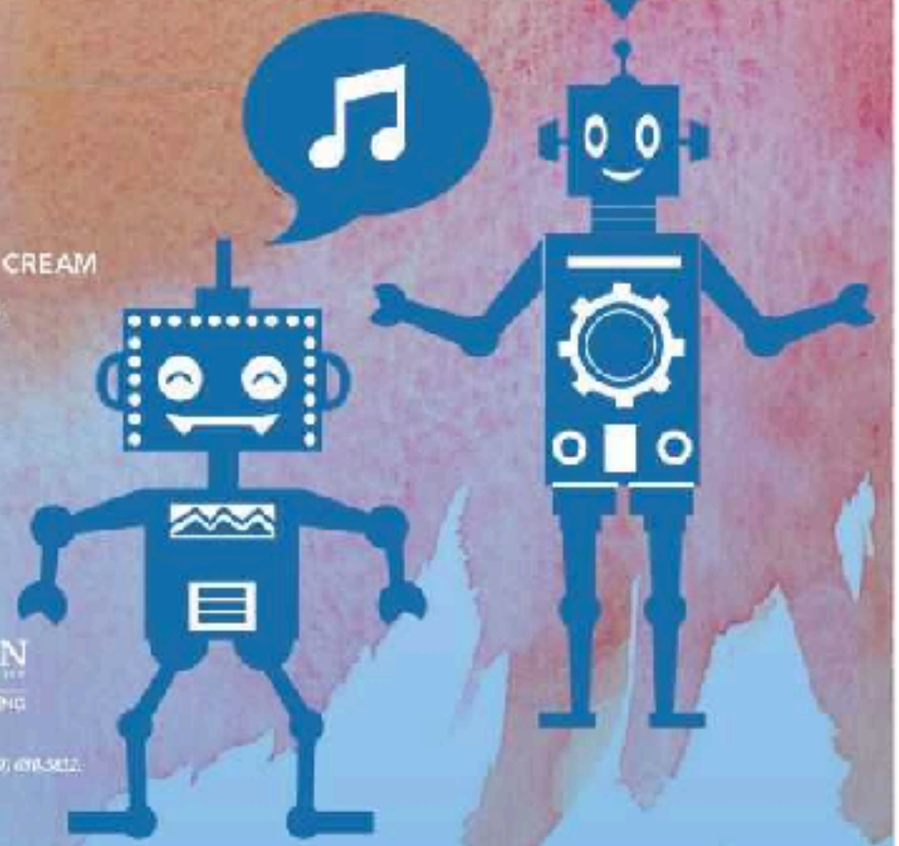
PHD SCIENCE AND TECHNOLOGY CANDIDATE AND RAPPER

MAC GYM

NOVEMBER 8

5PM

FREE FOOD
LIVE PERFORMANCE
RAFFLE PRIZES
LIQUID NITROGEN ICE CREAM
SCIENCE CLUB DEMOS
PHOTO BOOTH



WESTERN
COLLEGE of SCIENCE and ENGINEERING

Goals

- Understand the purpose and operations of the Map ADT.
- Know how to respond to large hash table load factors by resizing the array and **rehashing**.
- Know how to avoid using LinkedList buckets using **open addressing** with **linear** or **quadratic probing**.
- Understand the relationship between Java Object's **hashCode** and **equals** methods.

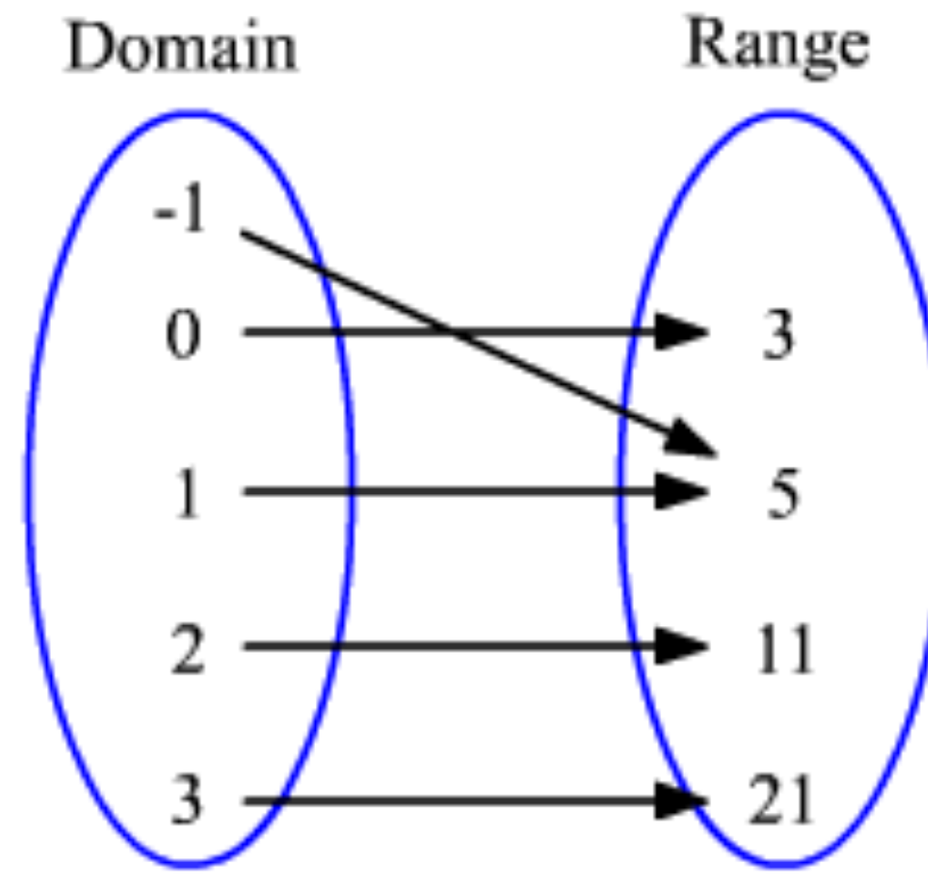
Origins of the term “hash”

Hans Peter Luhn (July 1, 1896 – August 19, 1964) was a researcher in the field of computer science, and, Library & Information Science for IBM

The Map ADT

- In math, a **map** is a function.
- What is a function, anyway?

The Map ADT

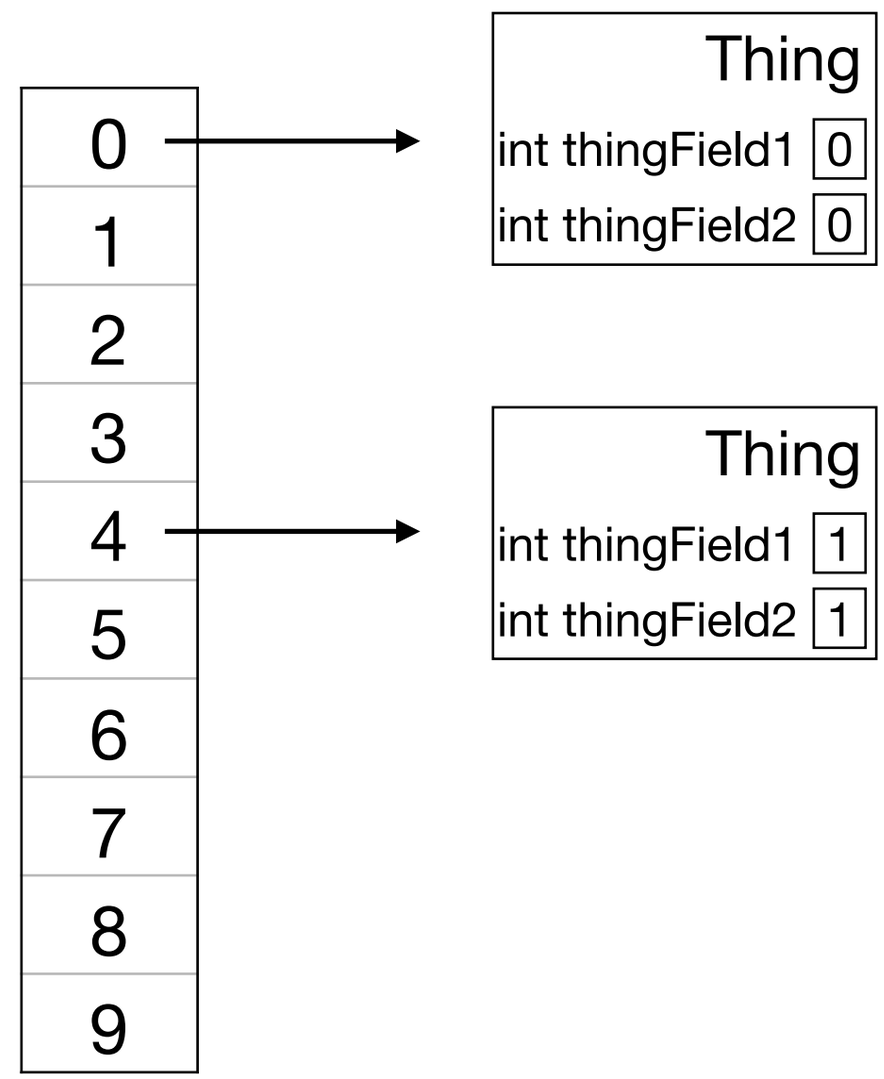


- In math, a **map** is a function.
- If F is a map then
  F(a) —> b
  means that a maps to b.
- F has a:
  - **domain** - the set of values F maps **from**
  - **range** - the set of values that F maps a domain element **to**
  - **codomain** - the set of **all** possible values in the range's type, regardless of whether any element in the domain maps to it

The Map ADT

- Arrays are great!
- Domain: 0..a.length
- Range: all elements stored in the array
- Codomain: the **type** of elements stored in the array.

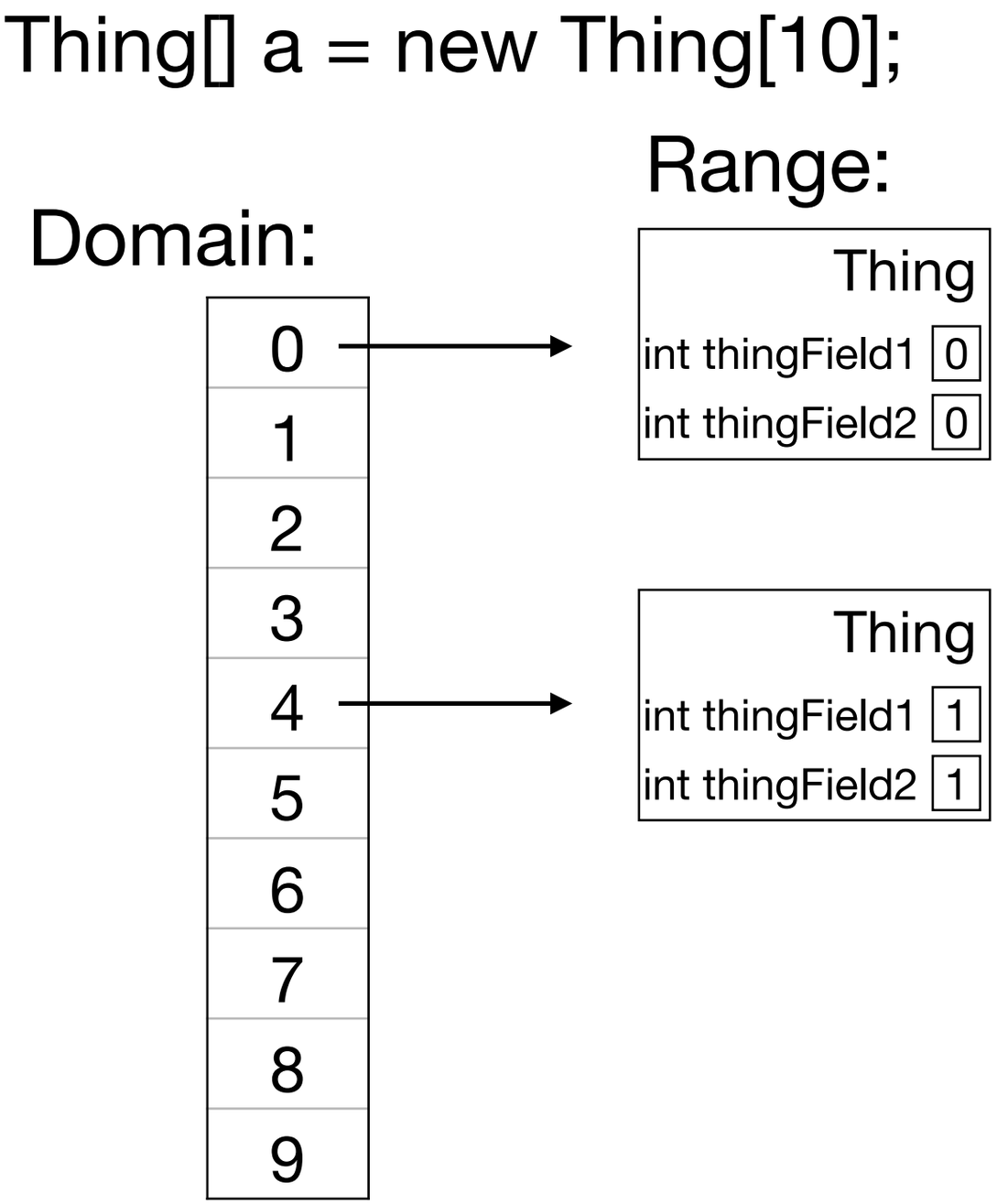
Thing[] a = new Thing[10];



The Map ADT

- Arrays are great!
- Domain: 0..a.length
- Range: all elements stored in the array
- Codomain: the **type** of elements stored in the array.

Thing[] a = new Thing[10];



We get to choose the **codomain**.

Codomain: Thing objects.

The Map ADT

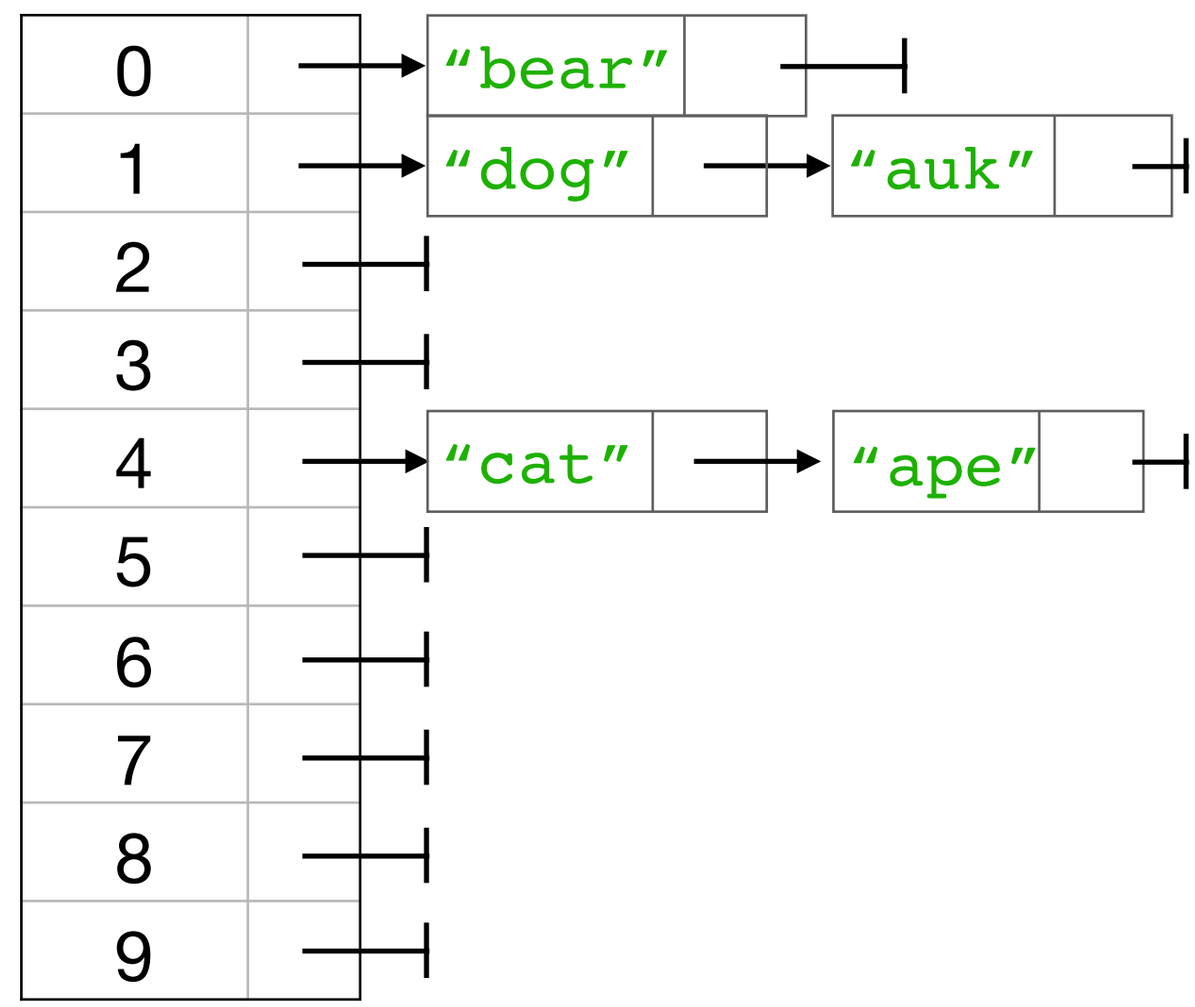
- Arrays are great!
- We get to choose the codomain - type of the array.
- Wouldn’t it be nice to choose the domain as well?
- The Map ADT represents a mapping from **keys** to **values**.
  - we get to choose the type of the **keys** (domain) AND the **values** (codomain)

The Map Interface

```
public interface Map<K,V> {
    /** Returns the value to which the specified key
      * is mapped, or null if this map contains no
      * mapping for the key. */
    V get(Object key);

    /** Associates the specified value with the
      * specified key in this map */
    V put(K key, V value);

    /** Removes the mapping for a key from this map
      * if it is present */
    V remove(Object key);

    // more methods
}
```

Implementing Map<K,V>

- Use a HashTable!
- Hash the key to determine array index
- Store values in array

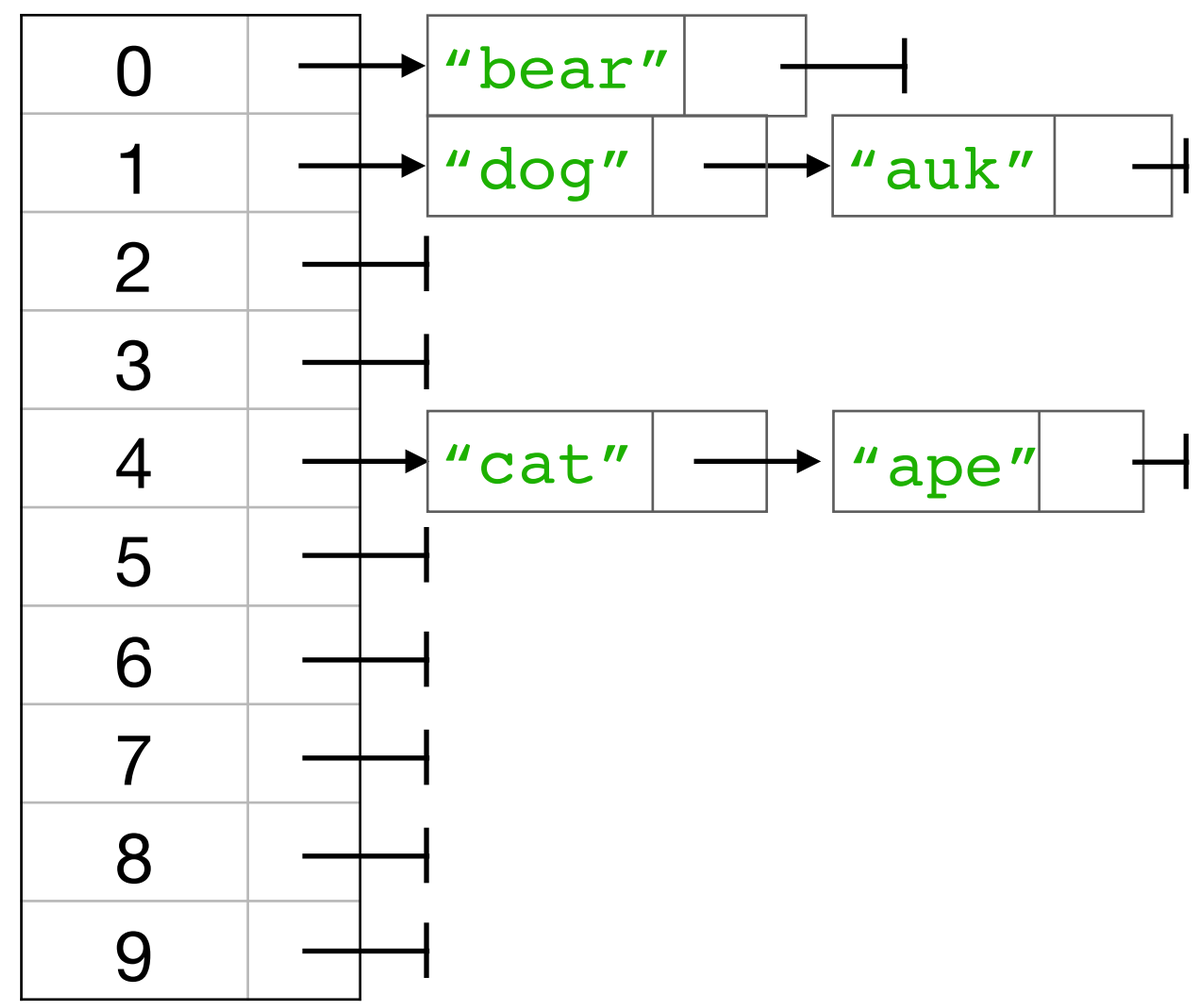


h(k) = k % A.length

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

Implementing Map<K,V>

- Use a HashTable!
- Hash the key to determine array index
- Store values in array



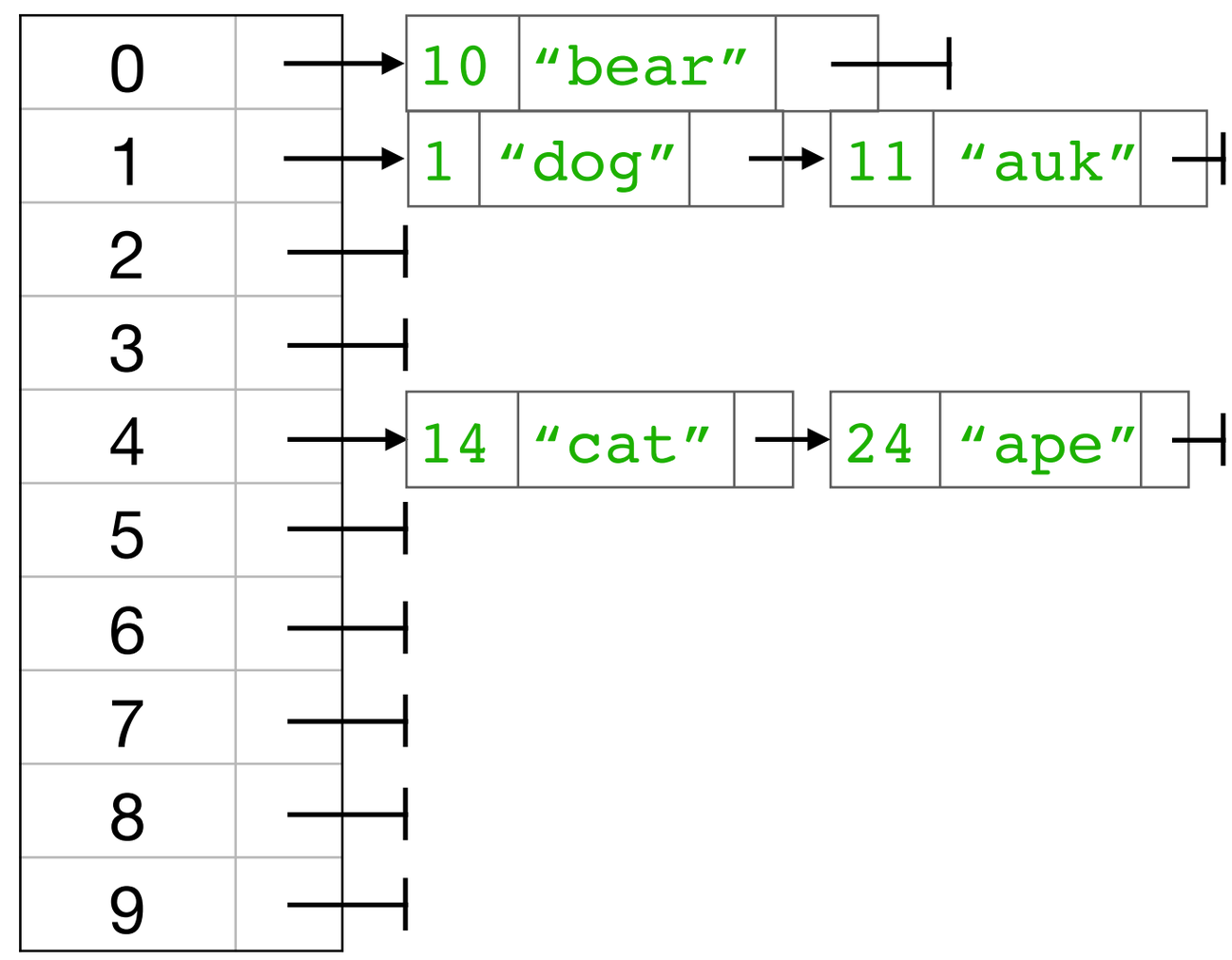
h(k) = k % A.length

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

Implementing Map<K,V>

- Use a HashTable (or a HashSet of Key-Value pairs)
- Hash the key to determine array index
- ~~Store values in array~~
- Store (K,V) pairs in the array.

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);



Hash Tables: Load Factor

How full is your hash table?

$$\text{Load factor } \lambda = \frac{\text{\# entries in table}}{\text{size of the array}}$$

The average bucket size is λ.

Average-case runtime is $O(\lambda)$.

Hash Tables: Load Factor

$$\text{Load factor } \lambda = \frac{\text{\# entries in table}}{\text{size of the array}}$$

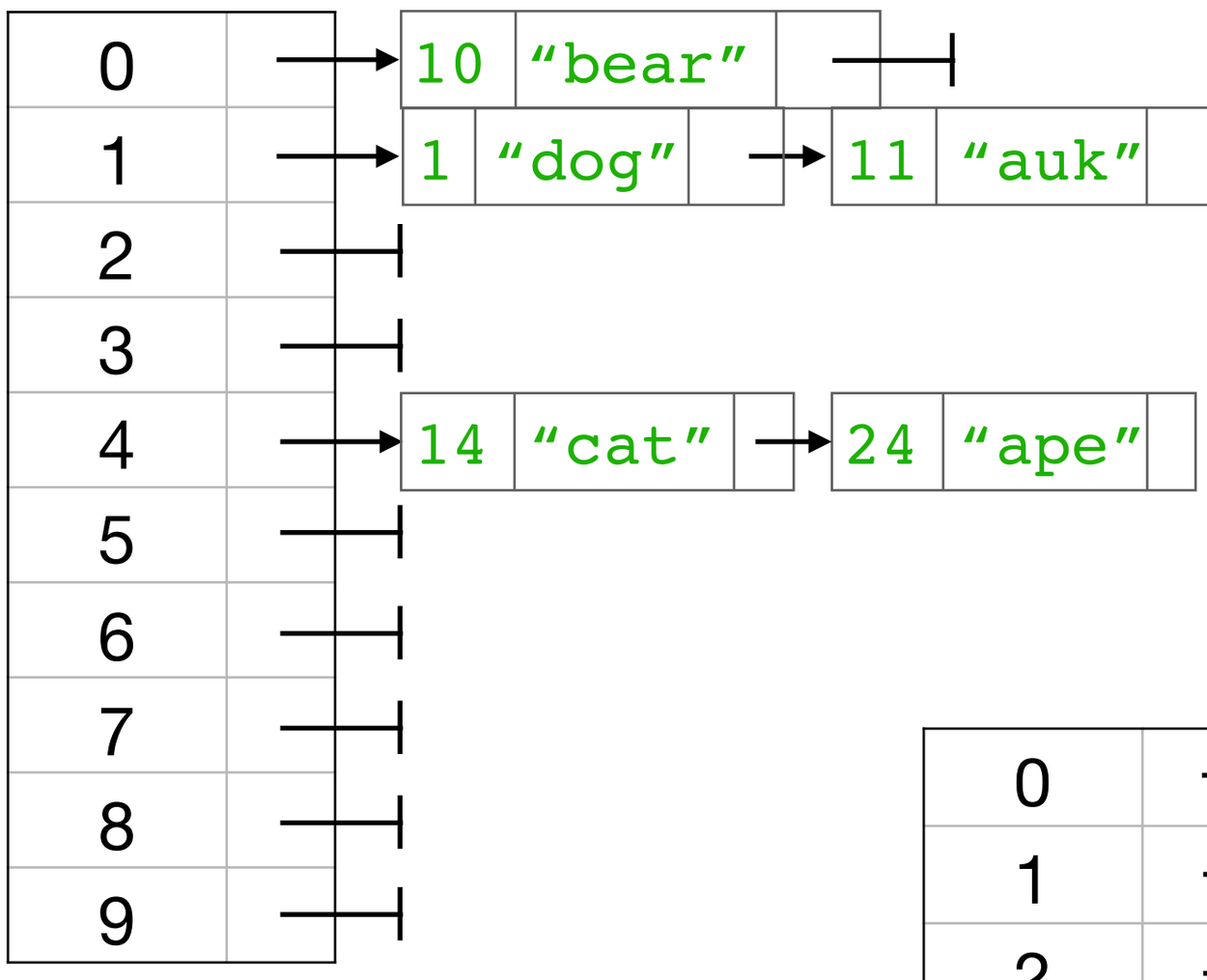
Average-case runtime is $O(\lambda)$.

- If λ is large, runtime is slow.
- If λ is small, memory is wasted.

Strategy: grow or shrink array when λ gets too large or small.

Shrinking the array

Requires **rehashing**: put each element where in belongs in the new array.



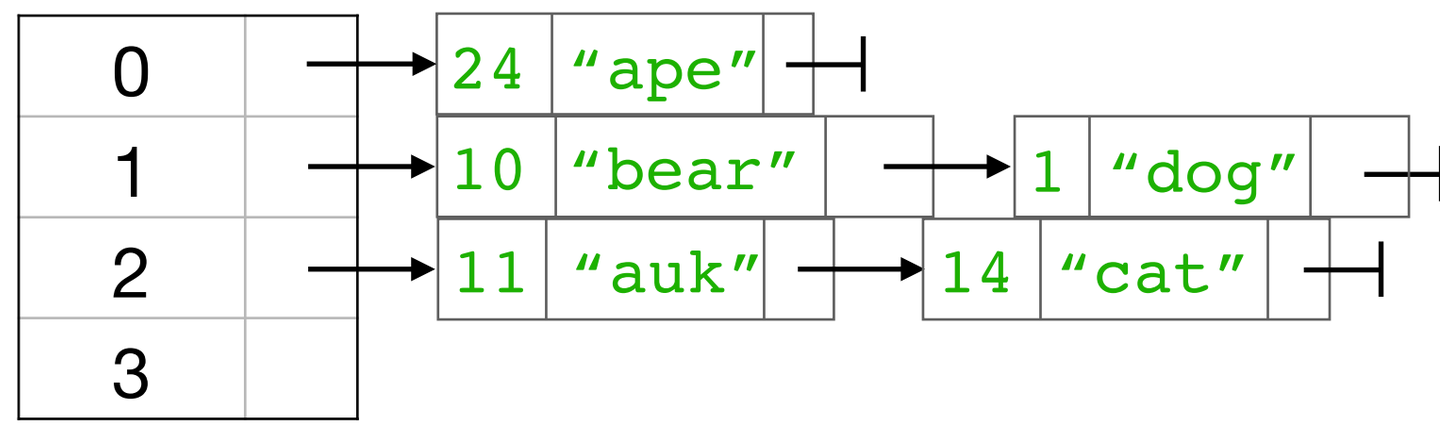
(10 % 3) -> 1

(1 % 3) -> 1

(11 % 3) -> 2

(14 % 3) -> 2

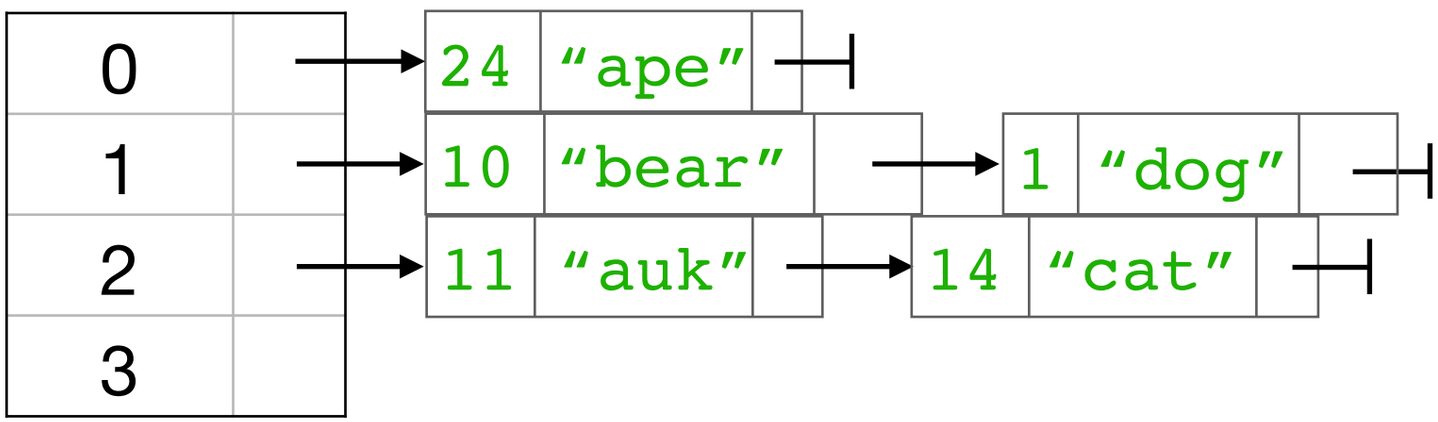
(24 % 3) -> 0

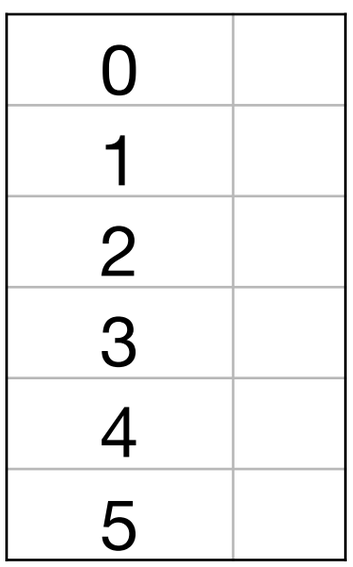


Growing the array

Also requires **rehashing**: put each element where in belongs in the new array.

Exercise: **Grow the array to size 6 and rehash:**



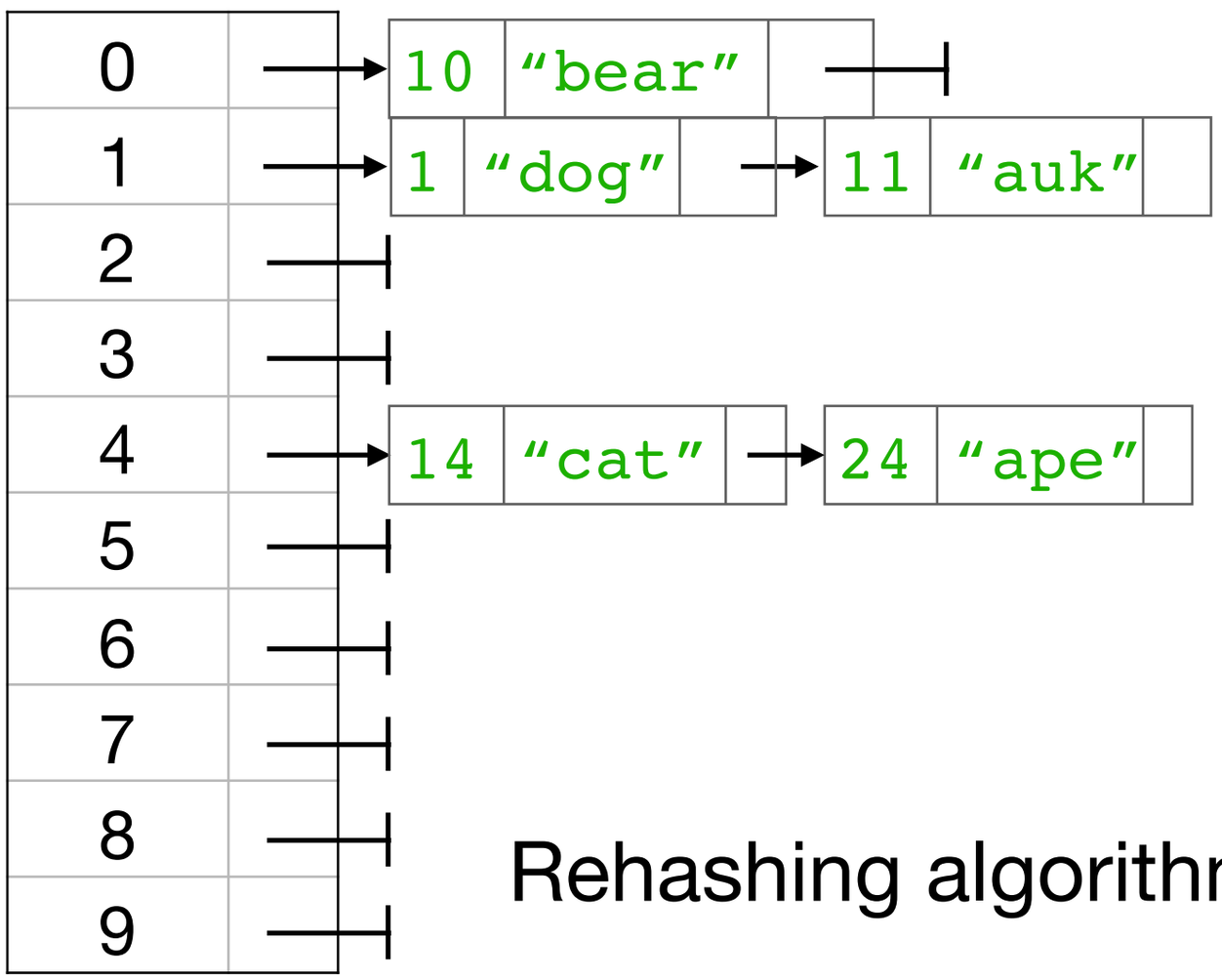


ABCD:

How many elements are in the most full bucket?

A. 1
B. 2
C. 3
D. 4

Rehashing: Runtime



Let N = array size

Let n = number of entries

Rehashing algorithm:

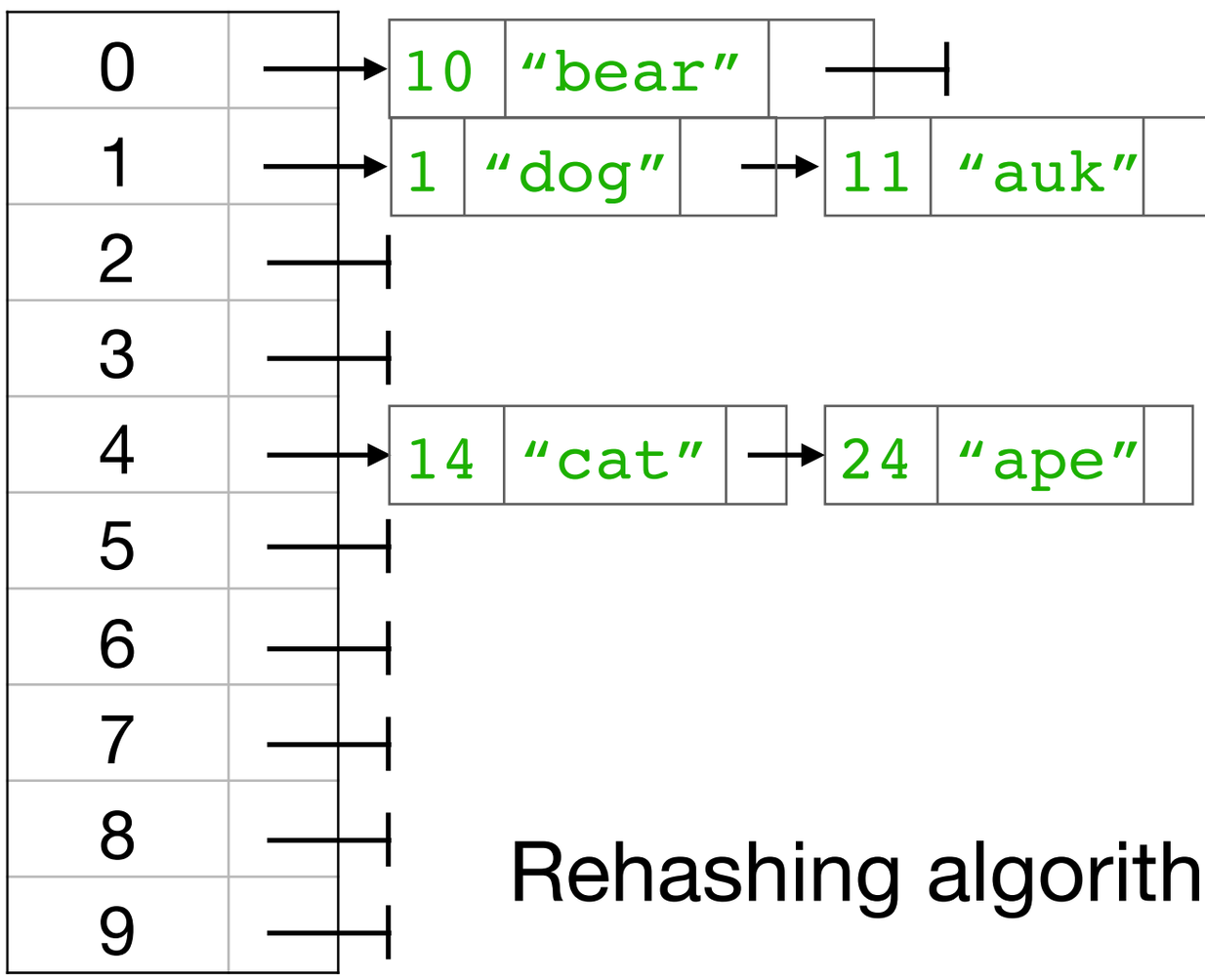
visits N buckets

visits n entries (total)

could be O(n) =(

```
for each bucket b:
  for each element e in b:
    put e into the new array
```

Rehashing: Runtime



Let N = array size

Let n = number of entries

Overall runtime is O(N + n^2)

Rehashing algorithm:

visits N buckets

visits n entries (total)

could be O(n) =(

```
for each bucket b:
  for each element e in b:
    put e into the new array
```

Hashing Multiple Integers

- Various heuristic methods:
  - (a + b + c + d) % N
  - (ak^1 + bk^2 + ck^3 + dk^4) % N

Hashing Strings

- Interpret ASCII (or unicode) representation as an integer.
- Java String uses:
  `s[0]*31^(n-1) + s[1]*31^(n-2)+ … +s[n-1]`

Collision Resolution

- **Chaining** - use a LinkedList to store multiple elements per bucket.
- **Open Addressing** - use empty buckets to store things that belong in other buckets.
  - Need some scheme for deciding which buckets to look in.

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	
1	
2	
3	
4	

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	
1	(1, dog)
2	
3	
4	

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	
1	(1, dog)
2	(11, auk)
3	
4	

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	
4	

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	
4	(14, cat)

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	(24, ape)
4	(14, cat)

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Linear Probing

- Problem with linear probing:
  - Hashing clustered values (e.g., 1, 1, 3, 2, 3, 4, 6, 4, 5) will result in a lot of searching.

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	(24, ape)
4	(14, cat)

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

Open Addressing with Quadratic Probing

- **Quadratic Probing:** Jump further ahead to avoid clustering of full buckets.

Linear probing looks at H, H+1, H+2, H+3, H+4, …

Quadratic probing looks at H, H+1, H+4, H+9, H+16, …

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	(24, ape)
4	(14, cat)

```
put(key):
    H = hash(key);
    i = 0;
    while A[h] is full:
        h = (H + i²) % N
        i++;
    A[h] = value
```

Open Addressing with Quadratic Probing

- **Quadratic Probing:** Jump further ahead to avoid clustering of full buckets.

Linear probing looks at H, H+1, H+2, H+3, H+4, …

Quadratic probing looks at H, H+1, H+4, H+9, H+16, …

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);

0	(10, bear)
1	(1, dog)
2	(11, auk)
3	(24, ape)
4	(14, cat)

```
put(key):
    H = hash(key);
    i = 0;
    while A[h] is full:
        h = (H + i²) % N
        i++;
    A[h] = value
```

Open Addressing with Quadratic Probing

- **Quadratic Probing:** Jump further ahead to avoid clustering of full buckets.

Exercise: Which buckets are full after the following insertions into an array size of 10 using quadratic probing?

put(0, “ape”);
put(1, “dog”);
put(20, “elf”);
put(21, “auk”);
put(40, “bear”);
put(41, “cat”);
put(60, “elk”);
put(61, “imp”);

```
put(key):
    H = hash(key);
    i = 0;
    while A[h] is full:
        h = (H + i²) % N
        i++;
    A[h] = value
```

Open Addressing with Quadratic Probing

- **Quadratic Probing:** Jump further ahead to avoid clustering of full buckets.

Exercise: Which buckets are full after the following insertions into an array size of 10 using quadratic probing?

put(0, “ape”);	0
put(1, “dog”);	1
put(20, “elf”);	0, 1, 4
put(21, “auk”);	1, 2
put(40, “bear”);	0, 1, 4, 9
put(41, “cat”);	1, 2, 5
put(60, “elk”);	0, 1, 4, 9, 6
put(61, “imp”);	1, 2, 5, 10, 7

```
put(key):
    H = hash(key);
    i = 0;
    while A[h] is full:
        h = (H + i²) % N
        i++;
    A[h] = value
```

Hashing in Java

- Object has a <u>hashCode</u> method.

  By default, this returns the object's address in memory.

- It needs to have the properties of a hash function!

  1. Deterministic: always returns the same value for the same object.
  2. **Equal** objects have equal hash codes.

In Java, "equal" means whatever the `equals` method says.

Consequence: if you override `equals`, you have to override `hashCode` to match.

Hashing in Java

Consequence: if you override `equals`, you have to override `hashCode` to match.

```
class Person {
  String firstName;
  String lastName;

  public boolean equals(Person p) {
    return firstName.equals(p.firstName)
        && lastName.equals(p.lastName);
  }

  public int hashCode() {
    return auxHash(firstName)
         + auxHash(lastName);
  }
}
```

Open Addressing: Runtime

- May be faster, but may not be. Depends on keys.
- There’s no free lunch: worst-case is always O(n).
- In practice, average-case is O(1) if you make good design decisions and insertions are not done by an adversary.

Further Reading

- CLRS 11.5: Perfect Hashing
  - You can guarantee O(1) lookups and insertions if the set of keys is fixed
- C++ <u>implementations</u> from Google:
  - sparse_hash_map - optimized for memory overhead
  - dense_hash_map - optimized for speed