Chapter 27 Hashing

CS165
Original Slides by Liang from
Introduction to Java Programming
Modifications by Wim Bohm and Sudipto Ghosh

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Topics

- Why is hashing needed? (§27.3).
- How to obtain the hash code for an object and design the hash function to map a key to an index (§27.4).
- Handling collisions using open addressing (§27.5).
- Linear probing, quadratic probing, and double hashing (§27.5).
- Handling collisions using separate chaining (§27.6).
- Load factor and the need for rehashing (§27.7).
- Implementation of Hashmap (§27.8).

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Why Hashing?

- Motivation: Quickly search, insert, and delete an element in a container

- Well-balanced search trees: Find an element in O(logn) time.

- Can we do better? Yes!
  - Use a technique called *hashing*.
  - Implement a map or a set to search, insert, and delete an element in O(1) time.

Map

- Data structure that stores entries containing two parts:
    - *Key:* also called search key
        - Used to search for the corresponding value
    - *Value*
        - Data stored
- Example:
    - A Dictionary can be stored in a map
    - Keys: words
    - Values: definitions of the words
- A map is also called a *dictionary*, a *hash table*, or an associative array.
- The new trend is to use the term map.

All rights reserved.

What is Hashing?

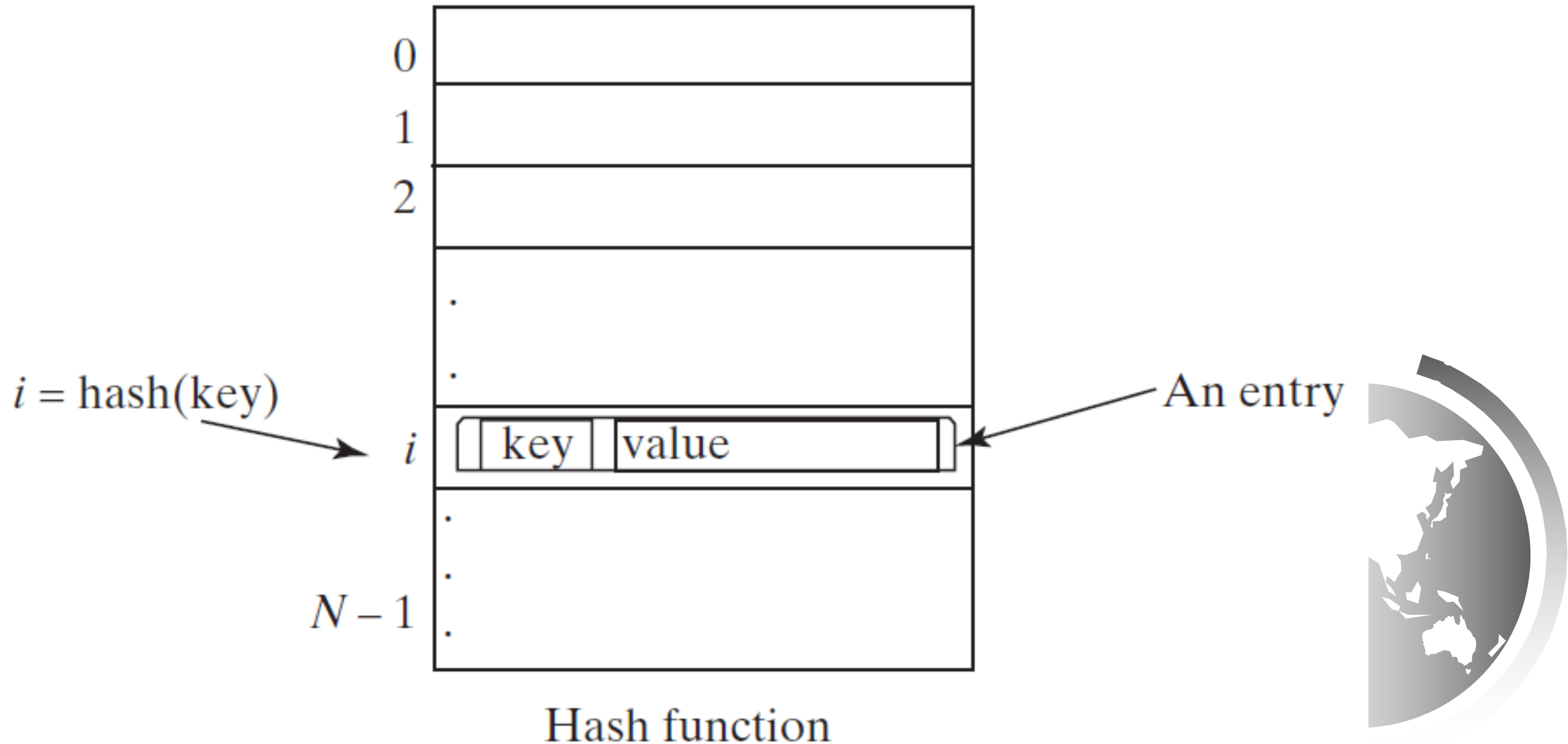
- Accessing an element in an array:
  - Retrieve the element using the index in O(1) time.
- Can we use an array as a map?
  - Key: array index
  - Value: array element
- Need to map a key to an array index.
- Hash table: array that stores the values
- Hash function: function that maps a key to an index in the table

Hashing is a technique that retrieves the value using the index obtained from key without performing a search.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Typical Hash Function

Step 1: Convert a search key to an integer value called a *hash code*.
Step 2: Compresses the hash code into an index to the hash table.

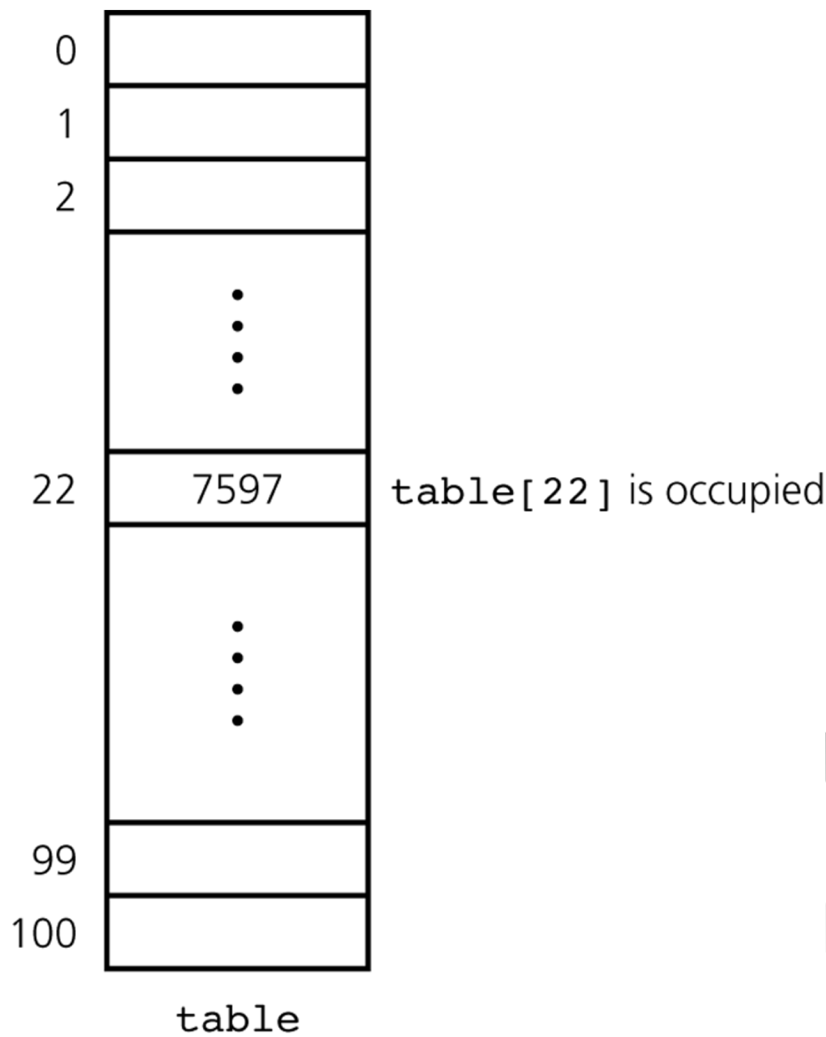


Hash function

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Collisions

Collision: two keys map to the same index



Hash function: key%101

Both 4567 and 7597 map to 22

All rights reserved.

The Birthday Problem

- What is the minimum number of people so that the probability that at least two of them have the same birthday is greater than ½?
- Assumptions:
  - Birthdays are independent
  - Each birthday is equally likely

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

The Birthday Problem

- What is the minimum number of people so that the probability that at least two of them have the same birthday is greater than ½?
- p_n – the probability that all people have different birthdays

$$p_n = 1 \frac{365}{366} \frac{364}{366} \cdots \frac{366-(n-1)}{366}$$

- at least two have same birthday:

$$n = 23 \rightarrow 1 - p_n \approx 0.506$$

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

The Birthday Problem: Probabilities

N: # of people	P(N): probability that at least two of the N people have the same birthday.
10	11.7 %
20	41.1 %
23	50.7 %
30	70.6 %
50	97. 0 %
57	99.0%
100	99.99997%
200	99.999999999999999999999999999998%
366	100%

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Probability of Collision

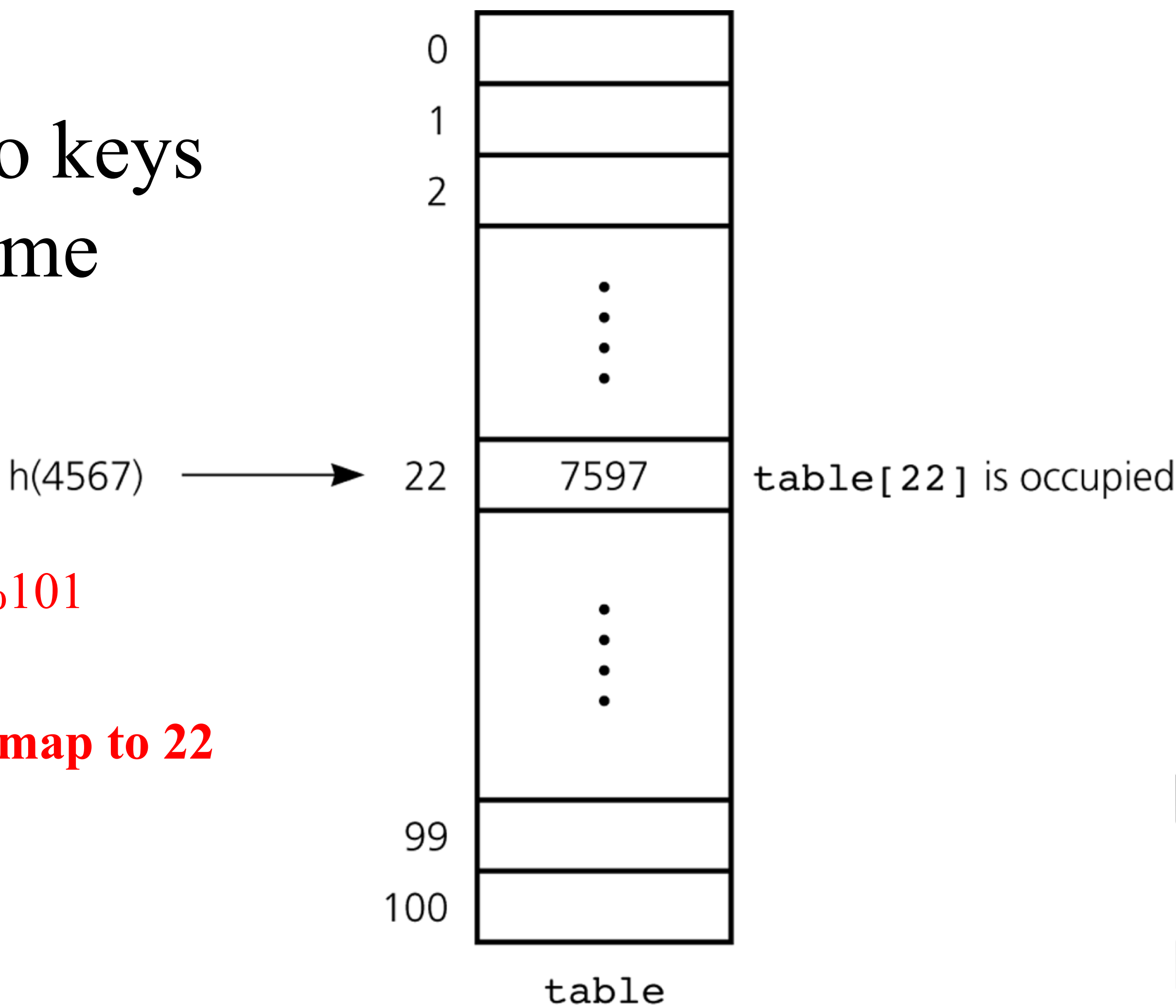
- How many items do you need to have in a hash table, so that the probability of collision is greater than ½?

- For a table of size 1,000,000 you only need 1178 items for this to happen!

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Collisions

Collision: two keys map to the same index



Hash function: key%101

both 4567 and 7597 map to 22

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Methods for Handling Collisions

- Approach 1: Open addressing
  - Probe for an empty (open) slot in the hash table
- Approach 2: Restructuring the hash table
  - Change the structure of the array table:
    - make each hash table slot **a collection**
    - ArrayList, or linked list
  - often called **separate chaining**
  - **Extendable dynamic hashing**

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

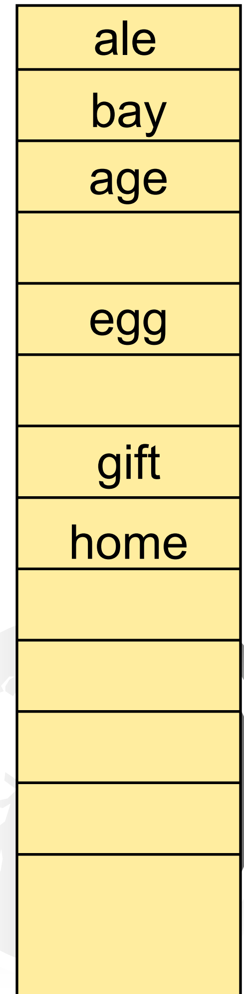
Open addressing

- When colliding with a location in the hash table that is already occupied
  - Probe for some other empty, open, location in which to place the item.
  - Probe sequence
    - The sequence of locations that you examine
    - **Linear probing** uses a constant step, and thus probes
      - Loc
      - (loc+step)%size
      - (loc+2*step)%size
      - etc.
- We use **step=1** for linear probing examples

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Linear Probing, step = 1

- Use first char. as hash function
  - Init: ale, bay, egg, home
- Where to search for
  - *egg* hash code 4
  - *ink* hash code 8
- Where to add
  - *gift* 6 empty
  - *age* 0 full, 1 full, 2 empty

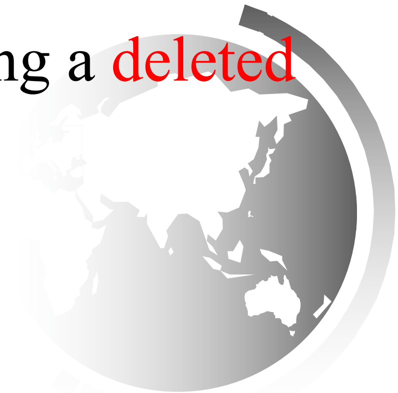


Question: During the process of linear probing, if there is an empty spot,
A. Item not found ?
or
B. There is still a chance to find the item ?

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

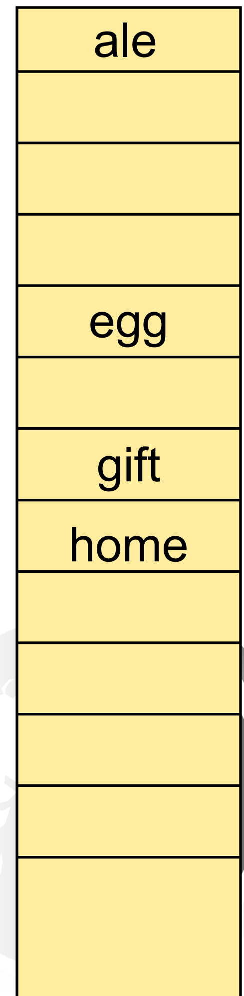
Open addressing: Linear Probing

- **Deletion:**
  - Empty positions created along a probe sequence could cause the retrieve method to stop, incorrectly indicating failure.
- **Resolution:**
  - Each position can be in one of three states **occupied, empty, or deleted**.
  - Retrieve then continues probing when encountering a deleted position.
  - Insert into empty or deleted positions.



Linear Probing (cont.)

- insert
  - *bay*
  - *age*
  - *acre*
- remove
  - *bay*
  - *age*
- retrieve
  - *acre*

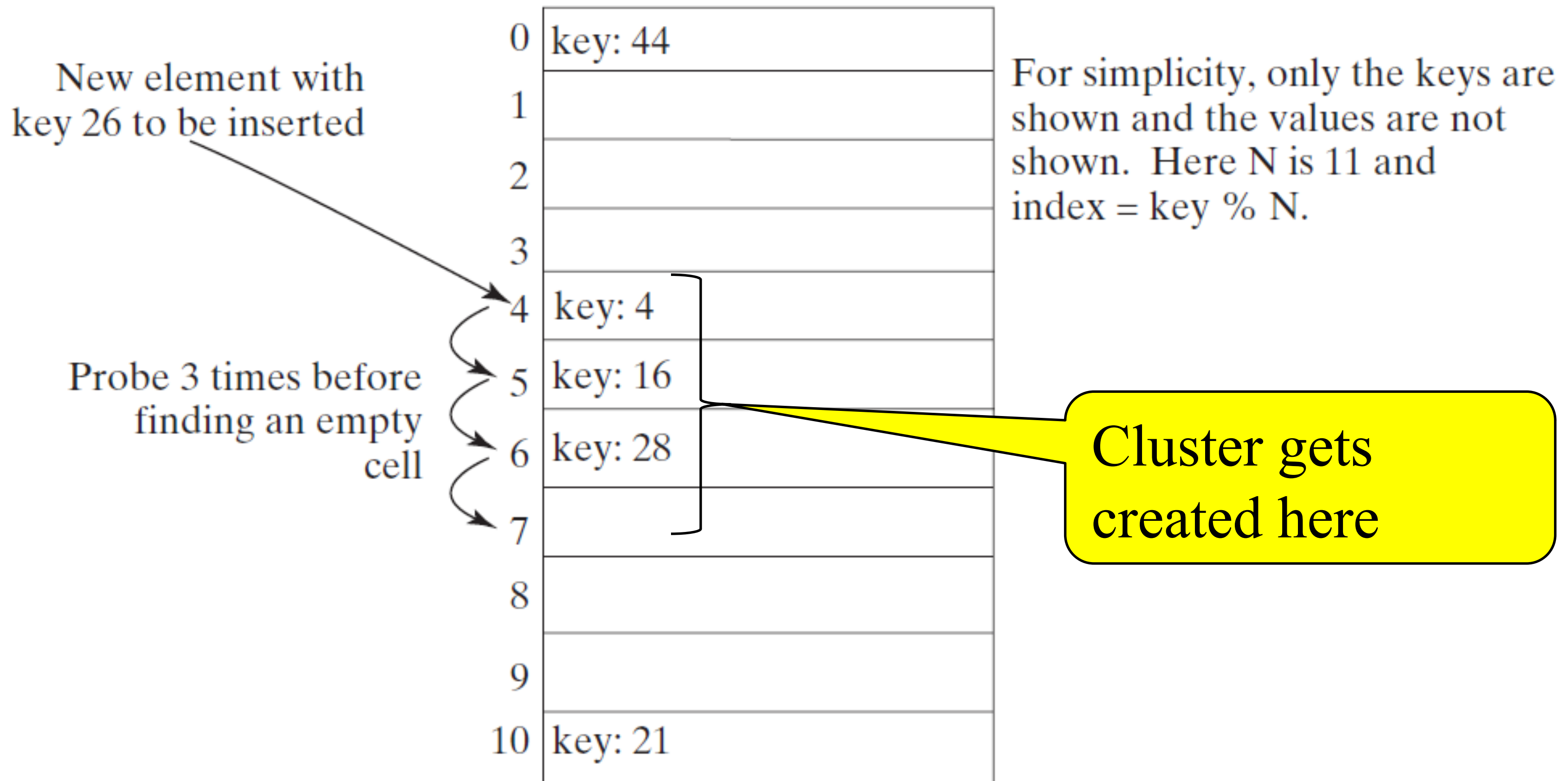


Question: Where does almond go now?

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Linear Probing Animation

http://www.cs.armstrong.edu/liang/animation/web/LinearProbing.html



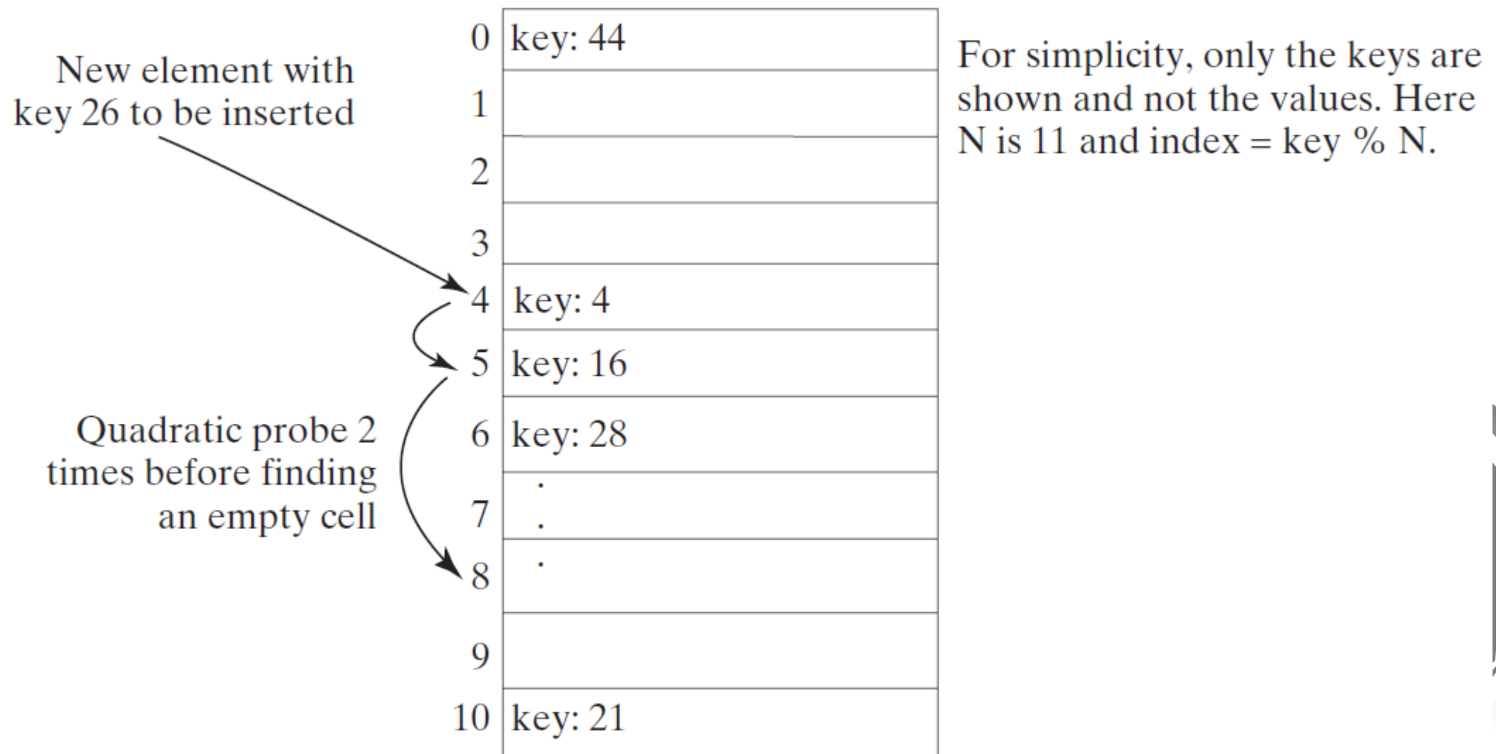
- Clusters can grow and merge into large clusters.
- Affects search, adding, removal.

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Quadratic Probing

www.cs.armstrong.edu/liang/animation/web/QuadraticProbing.html

- Quadratic probing can avoid the clustering problem in linear probing.
- Linear probing looks at the consecutive cells beginning at index k.
- Quadratic probing increases the index by j^2 for $j = 1, 2, 3, ...$
- The actual index searched are k, $k + 1$, $k + 4$, …



Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Summary of Linear and Quadratic Probing

- Start at index k = *hash(key)*
- Increments are <u>independent of the keys</u>
- Incr = step for linear, j^2 for quadratic
- New index
  - Linear probing with step=1: $(k + 1)\%N$, $(k + 2)\%N$, …
  - Quadratic probing j=1: $(k + 1)\%N$, $(k + 4)\%N$, …
- Both can cause clustering.
  - Linear probing is worse
  - Quadratic probing can also cause entries to collide in the same sequence (just quadratic instead of linear)

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Double Hashing

- Use a secondary hash function on the keys to determine the increments to avoid the clustering problem.
- Initial index k is calculated by hash function $h(key)$.
- Use second hash function $h'(key)$ to calculate increments
- New index = $(k + j * h'(key)) \% N$
  - $(k + h'(key))\%N$, $(k + 2*h'(key))\%N$, $(k + 3*h'(key))\%N$, …

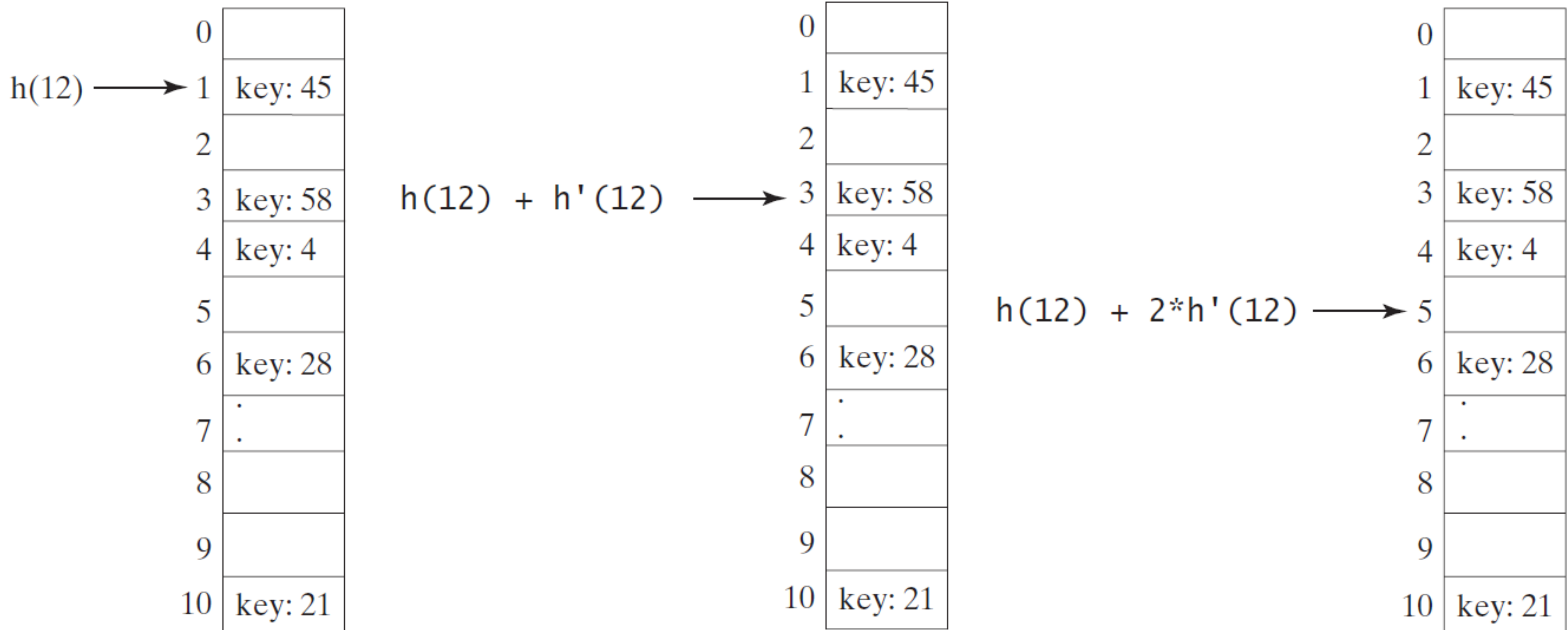
Example:
$h(key) = key \% 11;$
$h'(key) = 7 - key \% 7;$

Double Hashing

https://liveexample.pearsoncmg.com/dsanimation/DoubleHashingeBook.html

Example: Insert element with search key = 12

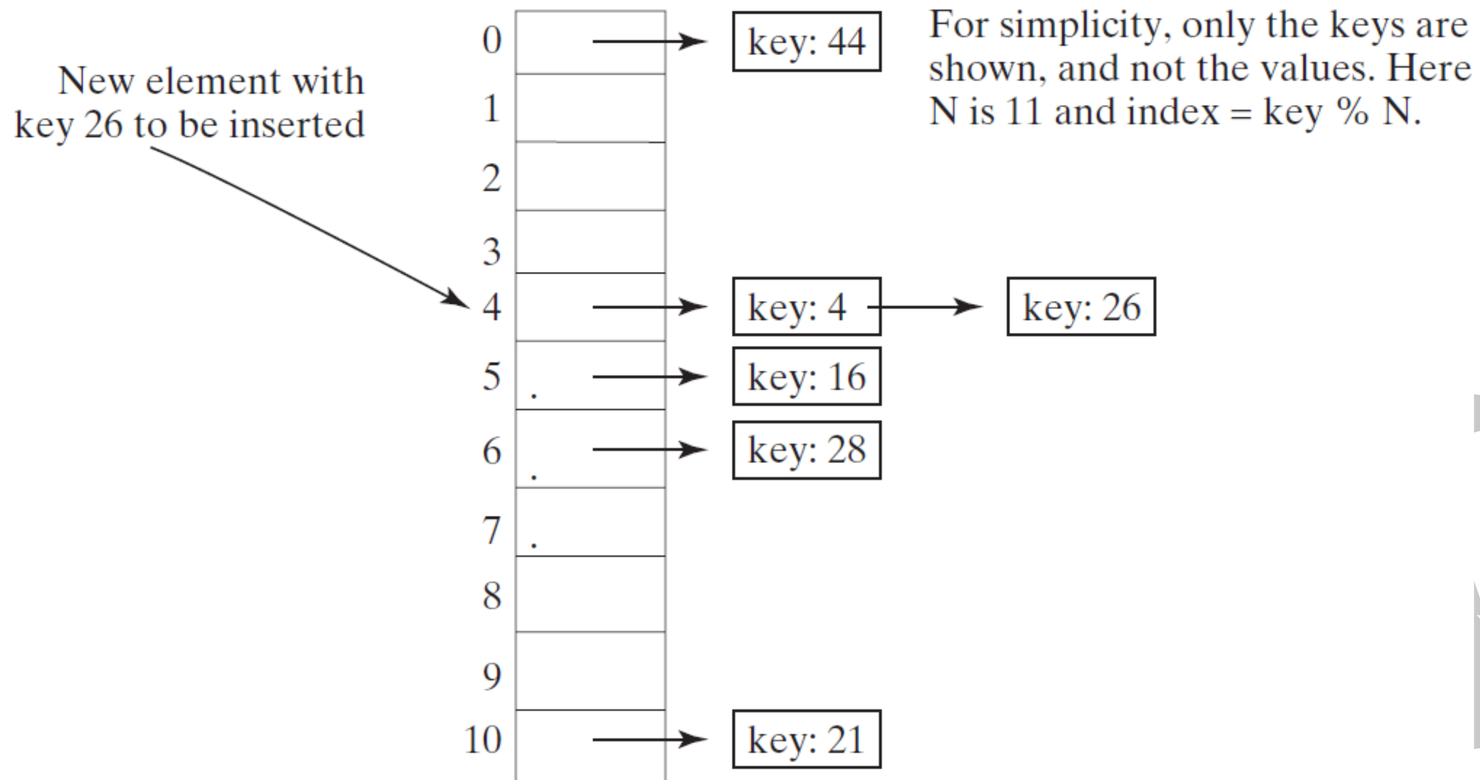
- $h(12) = 12 \% 11 = 1$
- $h'(12) = 7 - 12 \% 7 = 7 - 5 = 2;$



Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Handling Collisions Using Separate Chaining

- Don't try to find new locations.
- Place all entries with the same hash index into the same location,
- Each location in the separate chaining scheme is called a *bucket*.
- A bucket is a container that holds multiple entries.



Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Load Factor

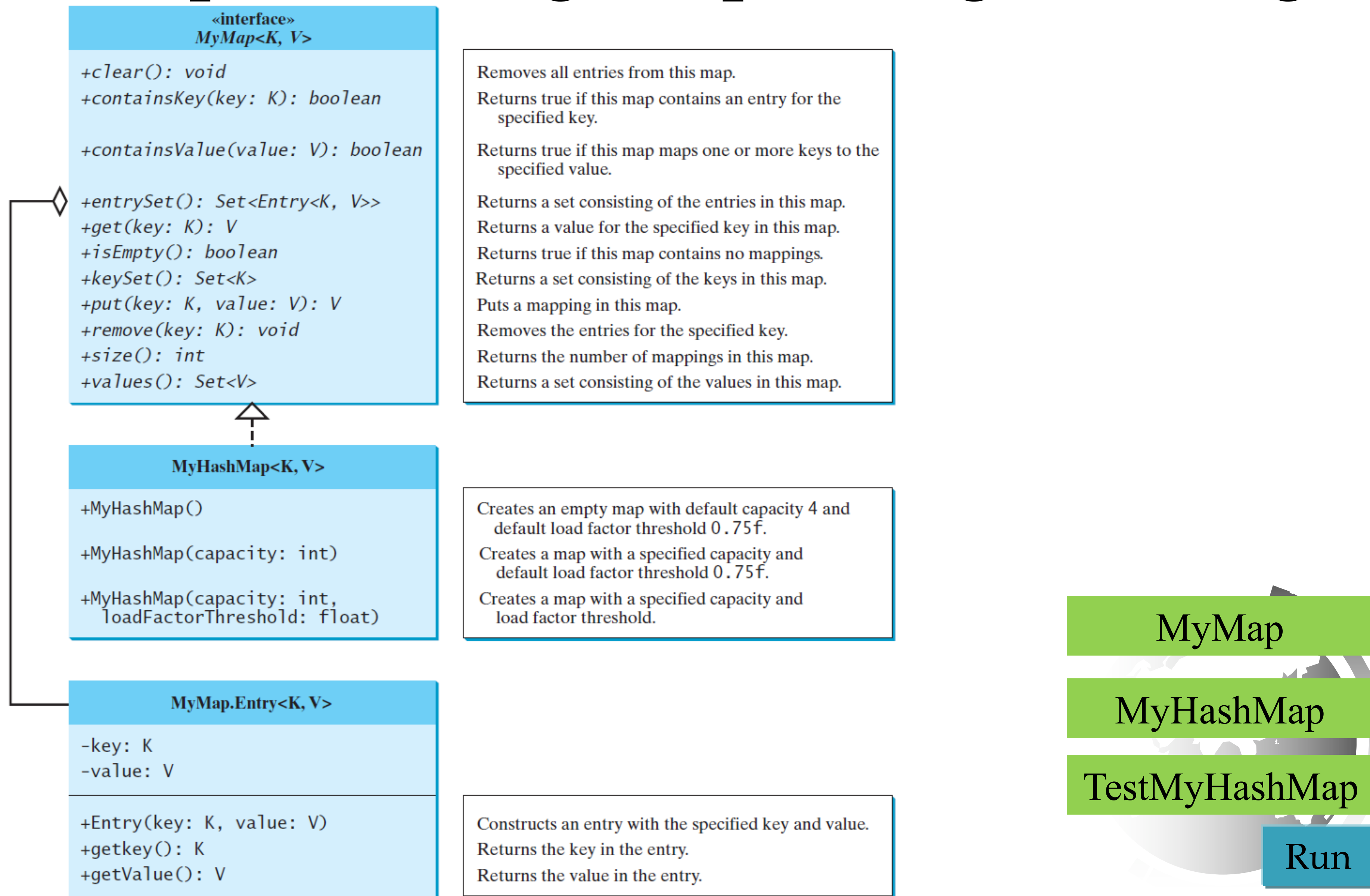
- Measures how full a hash table is
- $\lambda = \dfrac{number\ of\ elements\ in\ the\ hash\ table}{number\ of\ locations\ in\ the\ hash\ table}$
- Collisions can increase with higher value of λ
- For open addressing schemes:
  - λ lies between 0 (empty) and 1 (full)
  - Ideal value = 0.5
- For separate chaining scheme:
  - λ can have any value
  - Ideal value = 0.9

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Rehashing

- To avoid collisions, when λ reaches a threshold
  - Create a new larger hash table
  - Rehash all the map entries into the new hash table
- Rehashing is costly and can prevent other operations on the hash table from happening
- Generally size is doubled upon rehashing
- java.util.HashMap uses a threshold of 0.75

Implementing Map Using Hashing

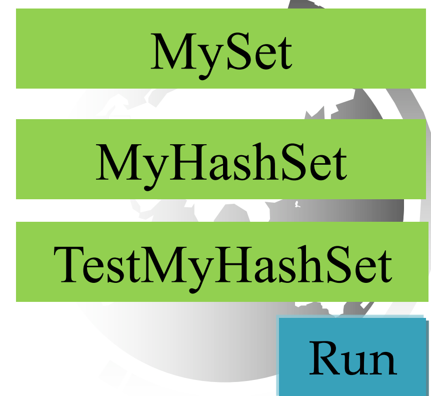


Implementing Set Using Hashing

«interface» *java.lang.Iterable<E>*	
+iterator(): java.util.Iterator<E>	

«interface» *MySet<E>*	
+clear(): void	Removes all elements from this set.
+contains(e: E): boolean	Returns true if the element is in the set.
+add(e: E): boolean	Adds the element to the set and returns true if the element is added successfully.
+remove(e: E): boolean	Removes the element from the set and returns true if the set contained the element.
+isEmpty(): boolean	Returns true if this set does not contain any elements.
+size(): int	Returns the number of elements in this set.

MyHashSet<E>	
+MyHashSet()	Creates an empty set with default capacity 4 and default load factor threshold 0.75f.
+MyHashMap(capacity: int)	Creates a set with a specified capacity and default load factor threshold 0.75f.
+MyHashMap(capacity: int, loadFactorThreshold: float)	Creates a set with a specified capacity and load factor threshold.



Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Explanation of MyHashMap and MyHashSet

Sudipto Ghosh and Wim Bohm

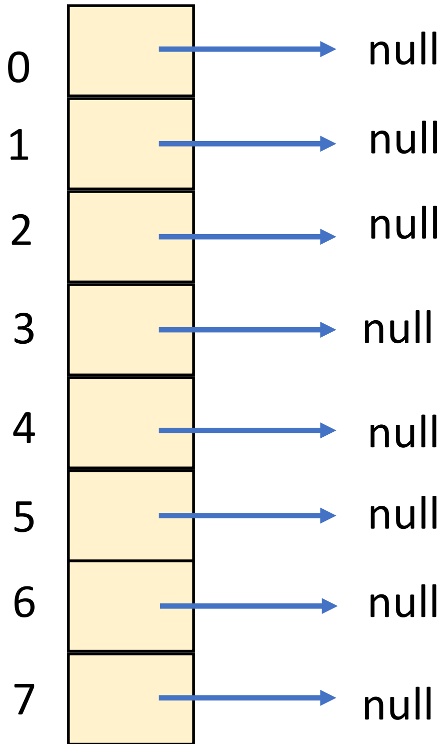
CS165

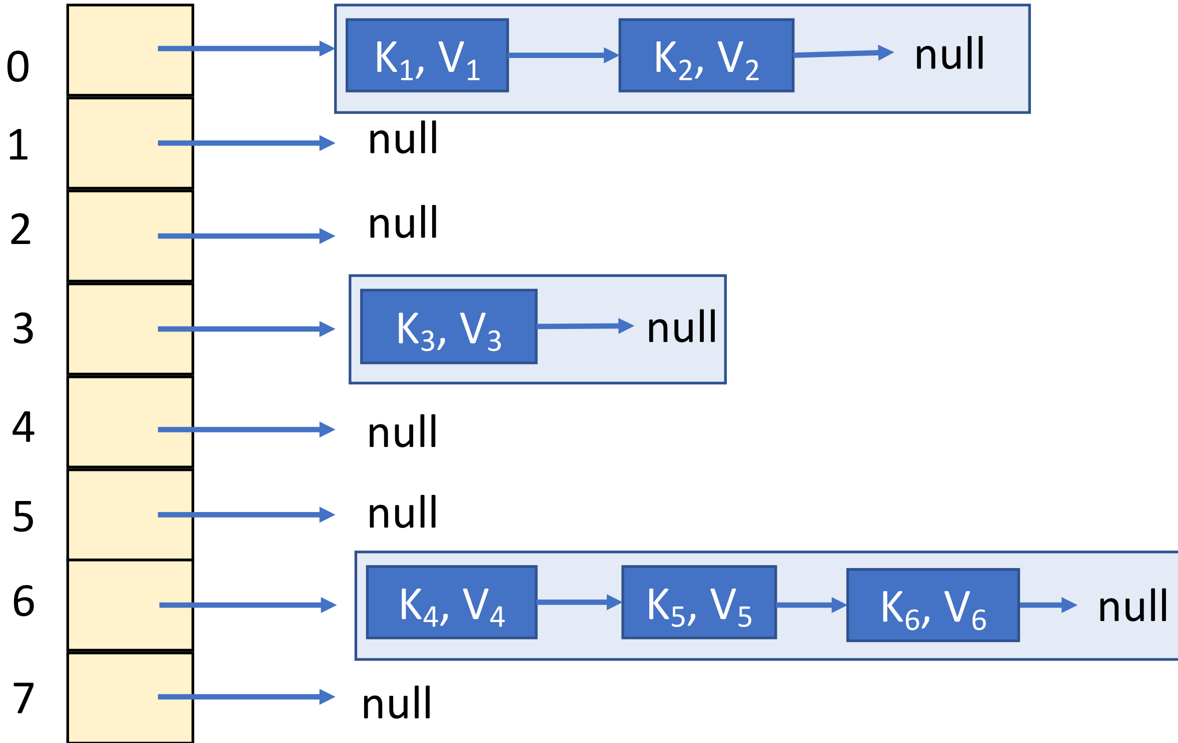
Based on the code in Liang Chapter 27

MyHashMap structure

Entry is a <key, value> pair

Number of slots is a power of 2 for convenience with hashing.





Initially each entry points to null.
There are no buckets.

At some later point in the execution.

Simple hash function

hash(key) = key & (N-1)

This uses bit-wise operators.

Faster execution than multiplication, division, etc.

Why do we choose this type of hash function?

If N is a power of 2,

then this hash will always produce a number between 0 and N-1.

Let’s take N = 8, so N-1 = 7.

Key = 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 0 1 0 1 0 1 0 1 1 1 0 0

N-1 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1

Key & (N-1) = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 = 4

Problem with simple hash function and solution

In the last example, if the last three bits are the same, then the keys will produce the same hash value.

Need a better distribution.

Use the notion of folding.

Use bitwise right shift operator and bitwise exclusive-or operator

Key = 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 0 1 0 1 0 1 0 1 1 1 0 0

Key >> 16 = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1

Key^(Key >> 16)= 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 0 0 0 1 1 1 1 0 1 1 1 1 0 1 1 1

Hash function used in the code

```
/** Ensure the hashing is evenly distributed */
private static int supplementalHash(int h) {
        h ^= (h >>> 20) ^ (h >>> 12);
        return h ^ (h >>> 7) ^ (h >>> 4);
}

/** Hash function */
private int hash(int hashCode) {
        return supplementalHash(hashCode) & (capacity - 1);
}
```

>>> unsigned right-shift operator