

Hashing



Objectives

- To know what hashing is for (§27.3).
- To obtain the hash code for an object and design the hash function to map a key to an index (§27.4).
- To handle collisions using open addressing (§27.5).
- To know the differences among linear probing, quadratic probing, and double hashing (§27.5).
- To handle collisions using separate chaining (§27.6).
- To understand the load factor and the need for rehashing (§27.7).
- To implement MyHashMap using hashing (§27.8).



Why Hashing?

The preceding chapters introduced search trees. An element can be found in $O(\log n)$ time in a well-balanced search tree. Is there a more efficient way to search for an element in a container? This chapter introduces a technique called *hashing*. You can use hashing to implement a map or a set to search, insert, and delete an element in $O(1)$ time.



Map

A *map* is a data structure that stores entries. Each entry contains two parts: *key* and *value*. The key is also called a *search key*, which is used to search for the corresponding value. For example, a dictionary can be stored in a map, where the words are the keys and the definitions of the words are the values.

A map is also called a *dictionary*, a *hash table*, or an associative array. The new trend is to use the term map.



What is Hashing?

If you know the index of an element in the array, you can retrieve the element using the index in $O(1)$ time. So, can we store the values in an array and use the key as the index to find the value? The answer is yes if you can map a key to an index.

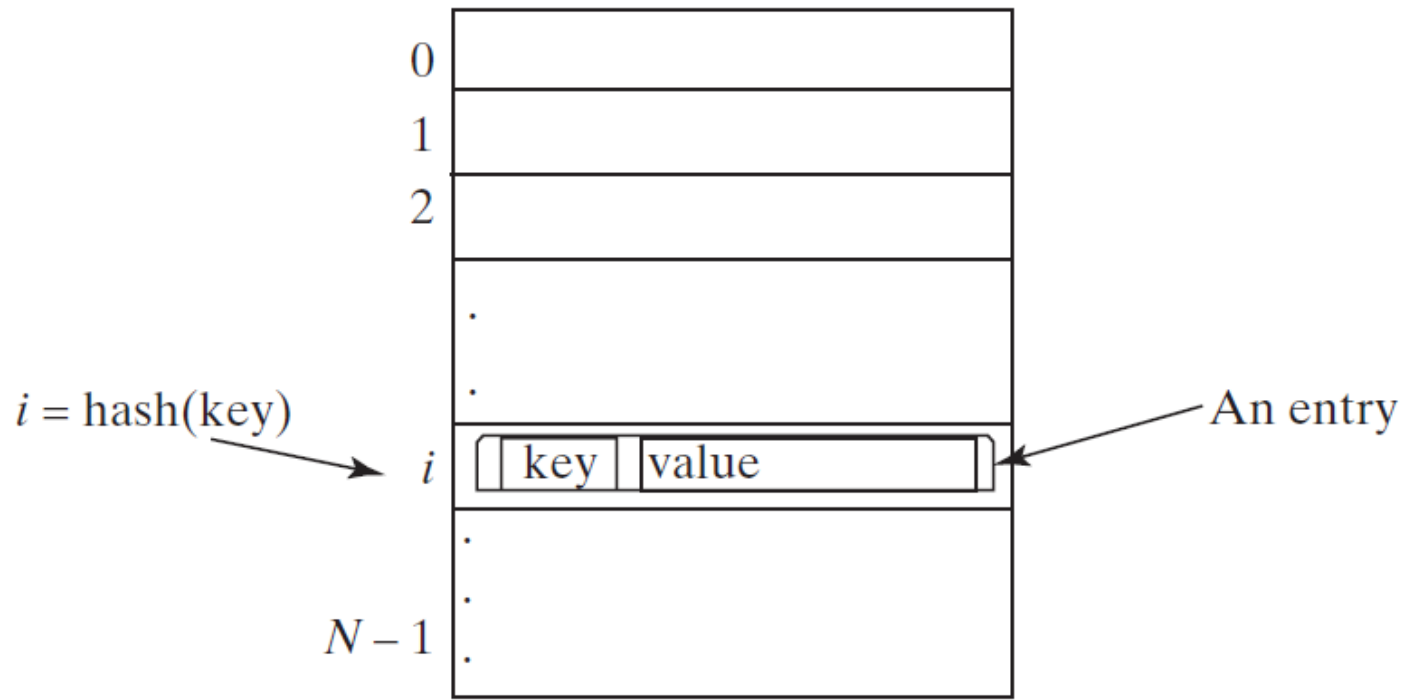
The array that stores the values is called a *hash table*. The function that maps a key to an index in the hash table is called a *hash function*.

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

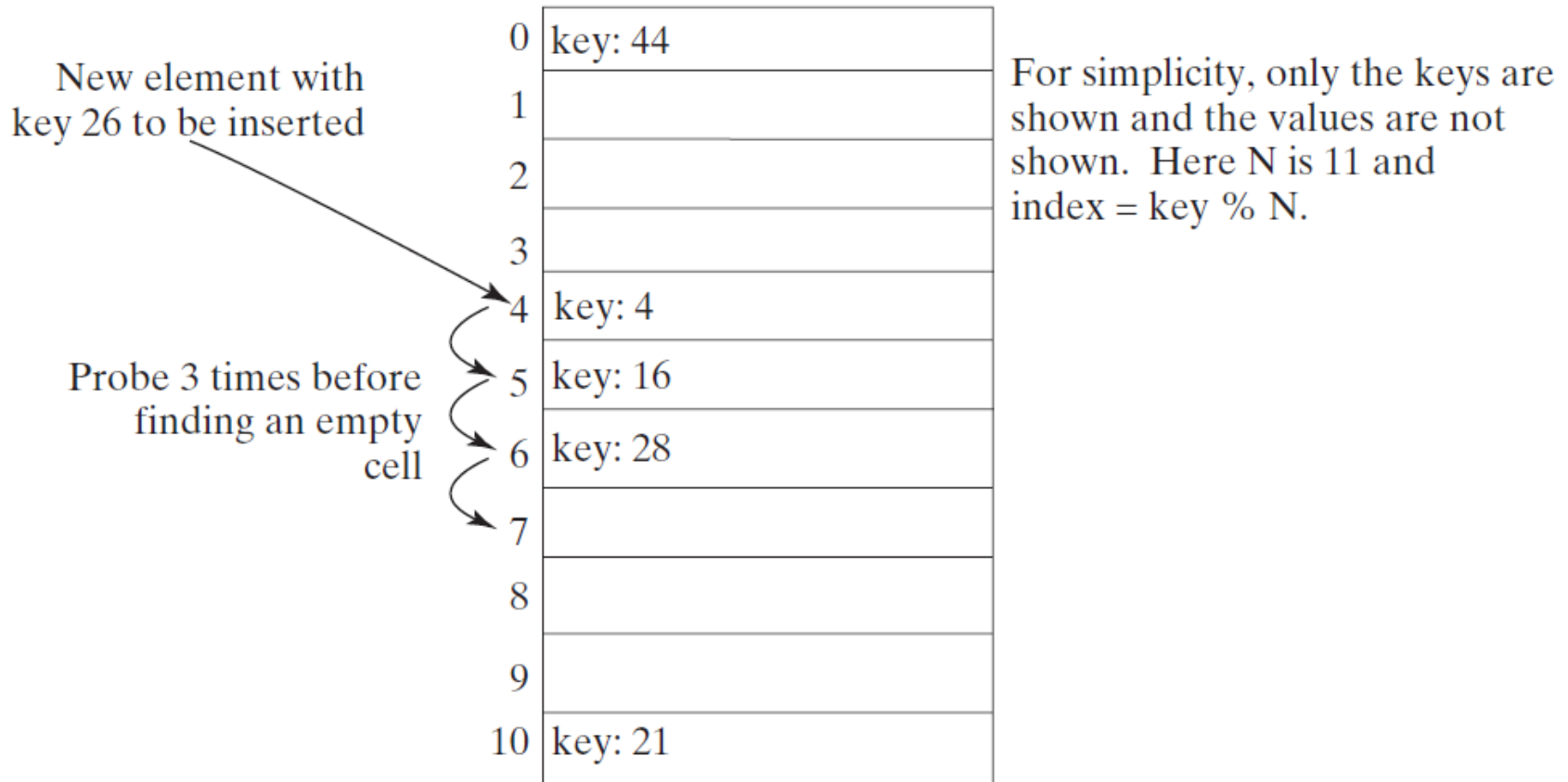


Hash Function and Hash Codes

A typical hash function first converts a search key to an integer value called a *hash code*, and then compresses the hash code into an index to the hash table.

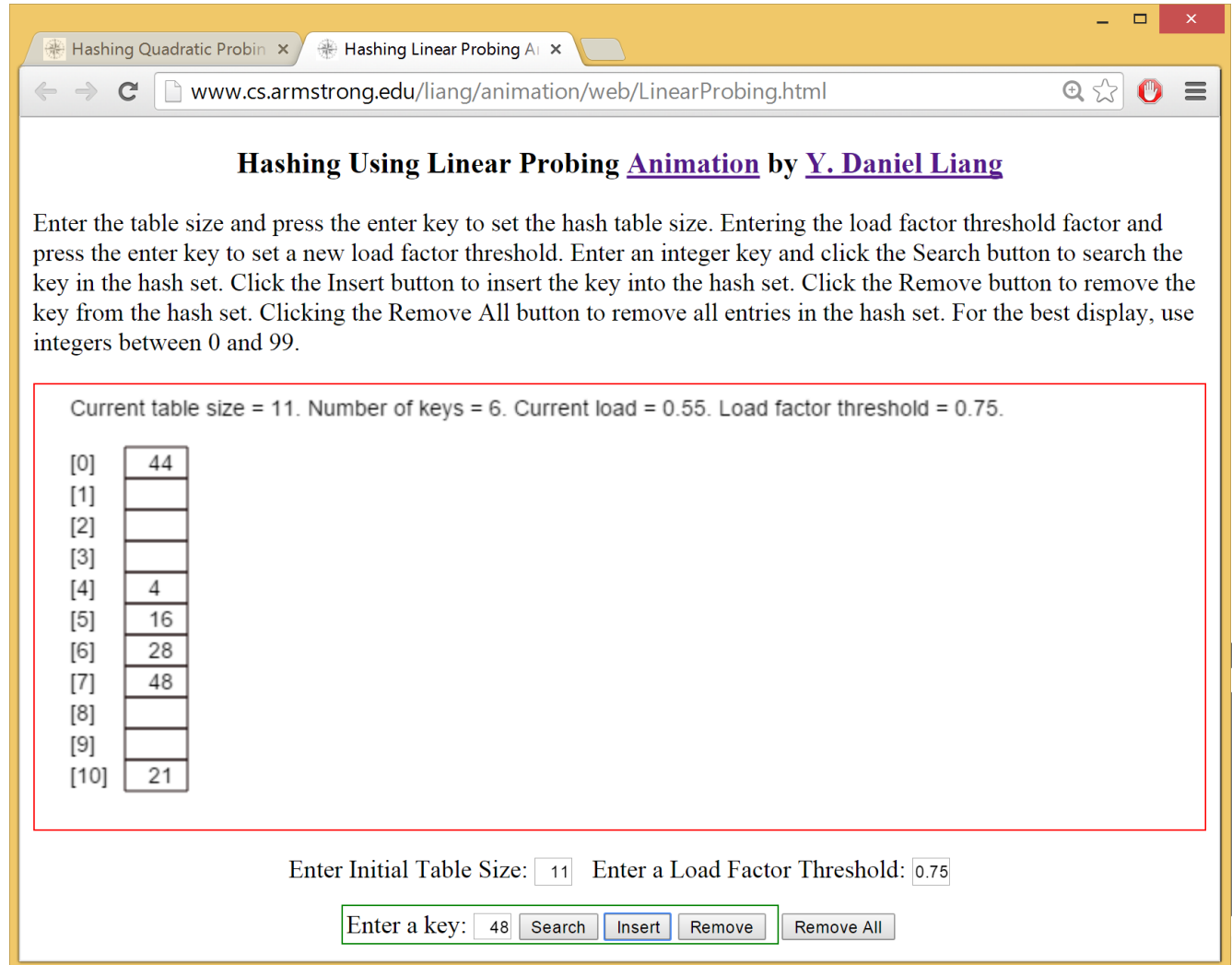


Linear Probing Animation



Linear Probing Animation

<https://liveexample.pearsoncmg.com/dsanimation/LinearProbingBook.html>



The screenshot shows a web browser window with two tabs: "Hashing Quadratic Probin" and "Hashing Linear Probing Ai". The address bar shows the URL www.cs.armstrong.edu/liang/animation/web/LinearProbing.html. The page title is "Hashing Using Linear Probing Animation by Y. Daniel Liang".

Instructions: Enter the table size and press the enter key to set the hash table size. Entering the load factor threshold factor and press the enter key to set a new load factor threshold. Enter an integer key and click the Search button to search the key in the hash set. Click the Insert button to insert the key into the hash set. Click the Remove button to remove the key from the hash set. Clicking the Remove All button to remove all entries in the hash set. For the best display, use integers between 0 and 99.

Current table size = 11. Number of keys = 6. Current load = 0.55. Load factor threshold = 0.75.

[0]	44
[1]	
[2]	
[3]	
[4]	4
[5]	16
[6]	28
[7]	48
[8]	
[9]	
[10]	21

Enter Initial Table Size: Enter a Load Factor Threshold:

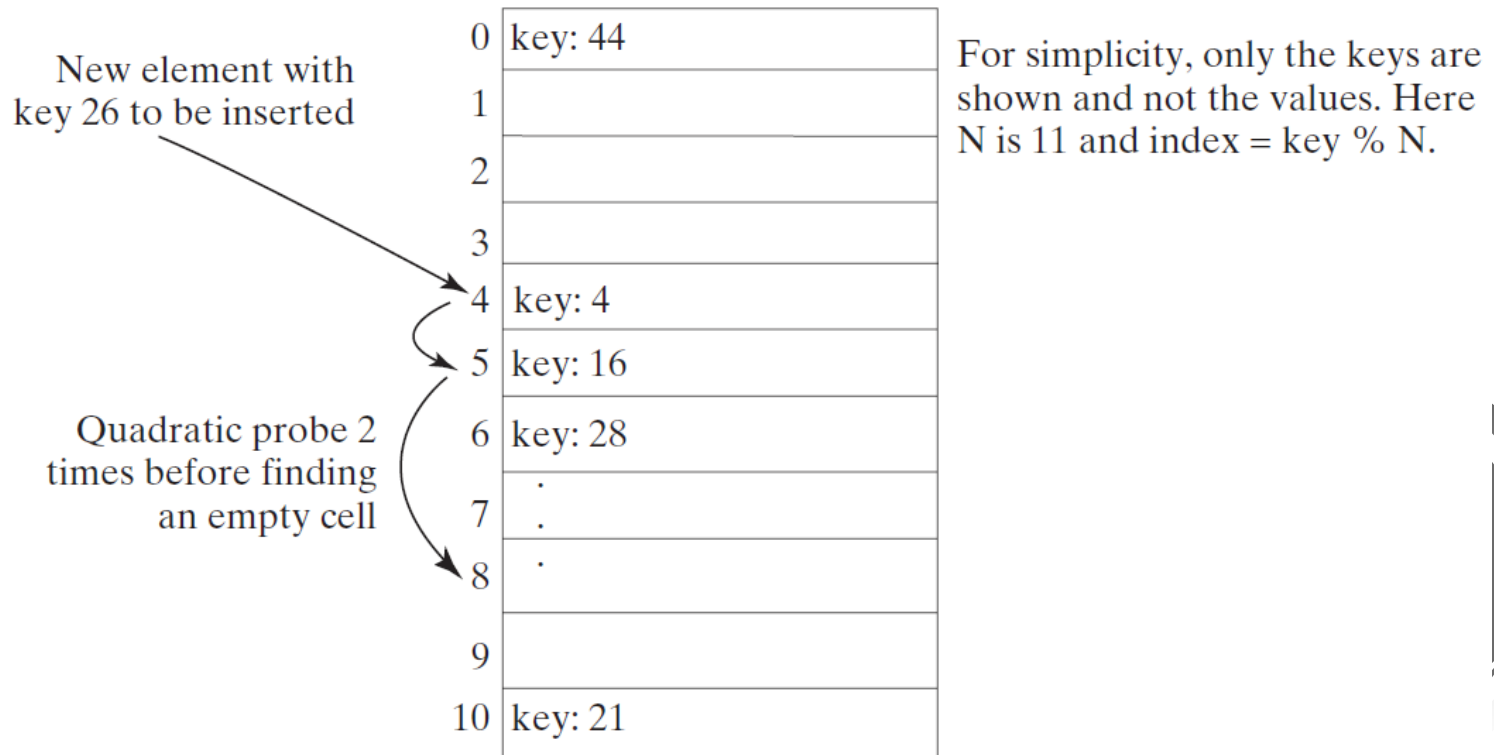
Enter a key:

Quadratic Probing



<https://liveexample.pearsoncmg.com/dsanimation/QuadraticProbingBook.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 $j = 1, 2, 3, \dots$. The actual index searched are $k, k + 1, k + 4, \dots$



Double Hashing

Double hashing uses a secondary hash function on the keys to determine the increments to avoid the clustering problem.

$$h'(k) = 7 - k \% 7;$$

$h(12) \longrightarrow$


0	
1	key: 45
2	
3	key: 58
4	key: 4
5	
6	key: 28
7	.
8	
9	
10	key: 21

$$h(12) + h'(12) \longrightarrow$$

0	
1	key: 45
2	
3	key: 58
4	key: 4
5	
6	key: 28
7	.
8	
9	
10	key: 21

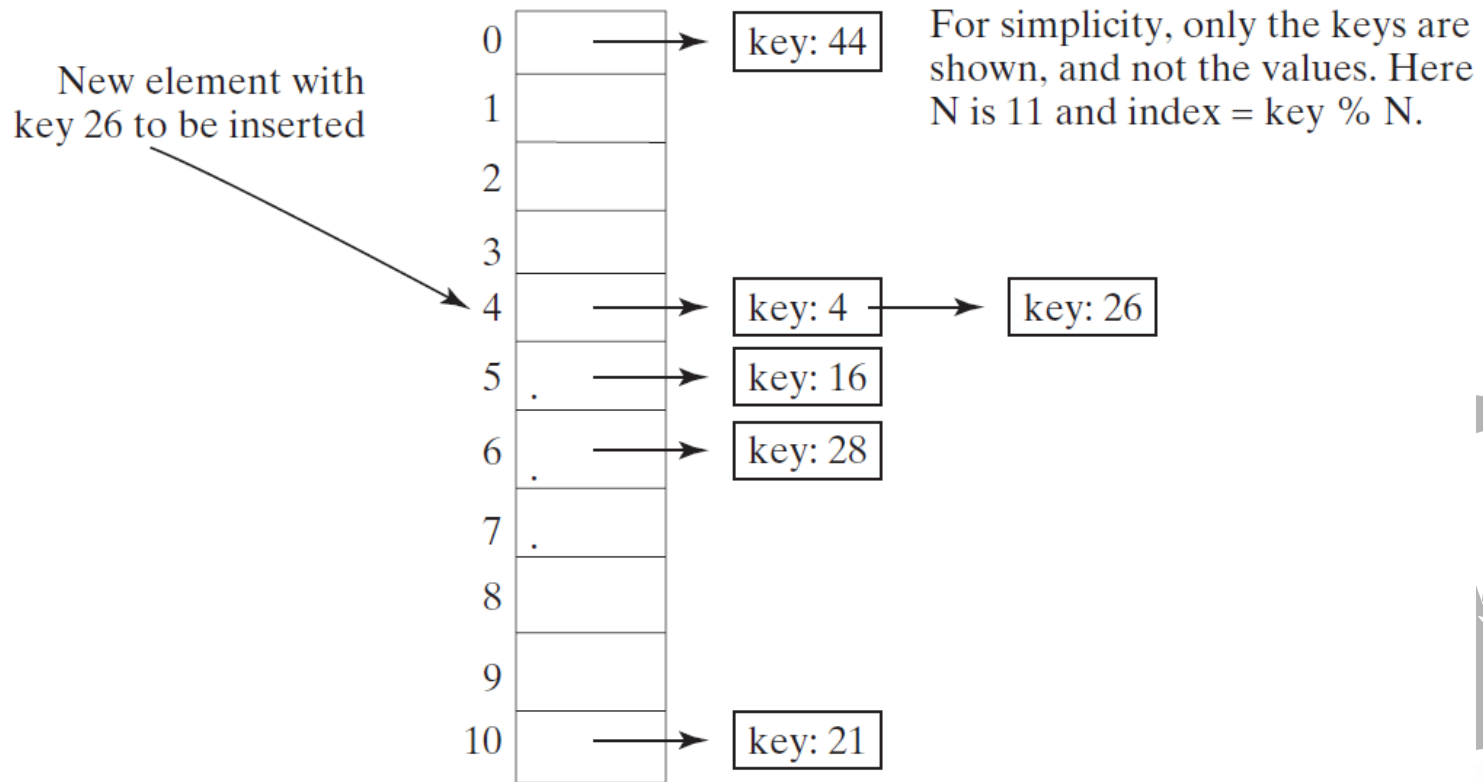
$$h(12) + 2 * h'(12) \longrightarrow$$

0	
1	key: 45
2	
3	key: 58
4	key: 4
5	
6	key: 28
7	.
8	
9	
10	key: 21



Handling Collisions Using Separate Chaining

The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a container that holds multiple entries.



Separate Chaining Animation

<https://liveexample.pearsoncmg.com/dsanimation/SeparateChainingBook.html>

Hashing Separate Chainin x

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

Hashing Using Separate Chaining Animation by Y. Daniel Liang

Enter the table size and press the enter key to set the hash table size. Entering the load factor threshold factor and press the enter key to set a new load factor threshold. Enter an integer key and click the Search button to search the key in the hash set. Click the Insert button to insert the key into the hash set. Click the Remove button to remove the key from the hash set. Clicking the Remove All button to remove all entries in the hash set. For the best display, use integers between 0 and 99.

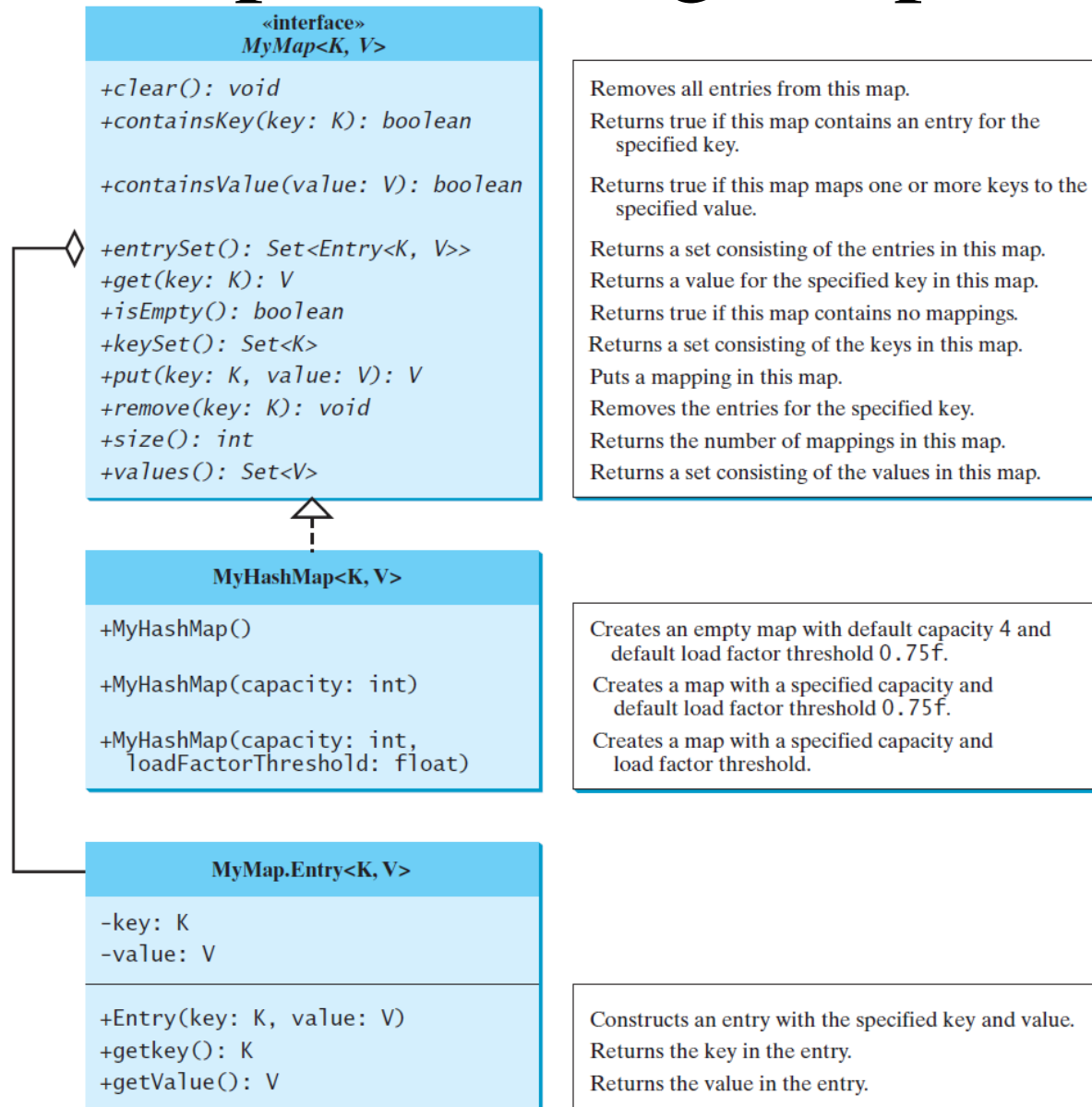
Current table size = 11. Number of keys = 4. Current load = 0.36. Load factor threshold = 0.5.

[0]		
[1]	→	1
[2]		
[3]		
[4]	→	48
[5]		
[6]		
[7]		
[8]		
[9]	→	31
[10]	→	21

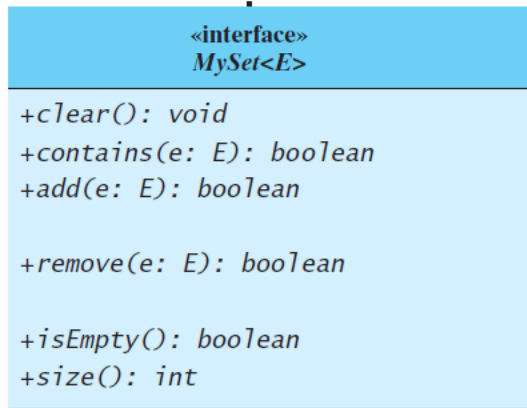
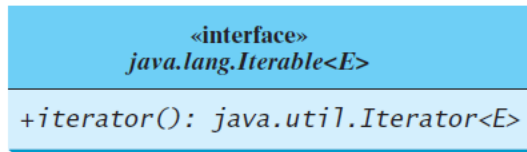
Enter Initial Table Size: 11 Enter a Load Factor Threshold: 0.5

Enter a key: 48 Search Insert Remove Remove All

Implementing Map Using Hashing



Implementing Set Using Hashing



Removes all elements from this set.

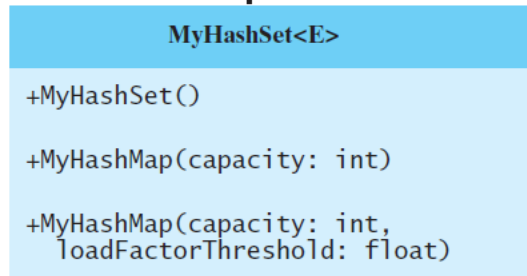
Returns true if the element is in the set.

Adds the element to the set and returns true if the element is added successfully.

Removes the element from the set and returns true if the set contained the element.

Returns true if this set does not contain any elements.

Returns the number of elements in this set.



Creates an empty set with default capacity 4 and default load factor threshold 0.75f.

Creates a set with a specified capacity and default load factor threshold 0.75f.

Creates a set with a specified capacity and load factor threshold.

MySet

MyHashSet

TestMyHashSet

Run