# Chapter 21 Hashing: Implementing Dictionaries and Sets

# Objectives

- To know what hashing is for (§21.2).
- To use the hash function to obtain a hash code (§21.2).
- To handle collisions using open addressing (§21.3).
- To know the differences among linear probing, quadratic probing, and double hashing (§21.3).
- To handle collisions using separate chaining (§21.4).
- To understand the load factor and the need for rehashing (§21.5).
- To implement Map using hashing (§21.6).
- To implement Set using hashing (§21.7).

# Why Hashing?

The preceding chapters introduced search trees. An element can be found in O(logn) 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*.

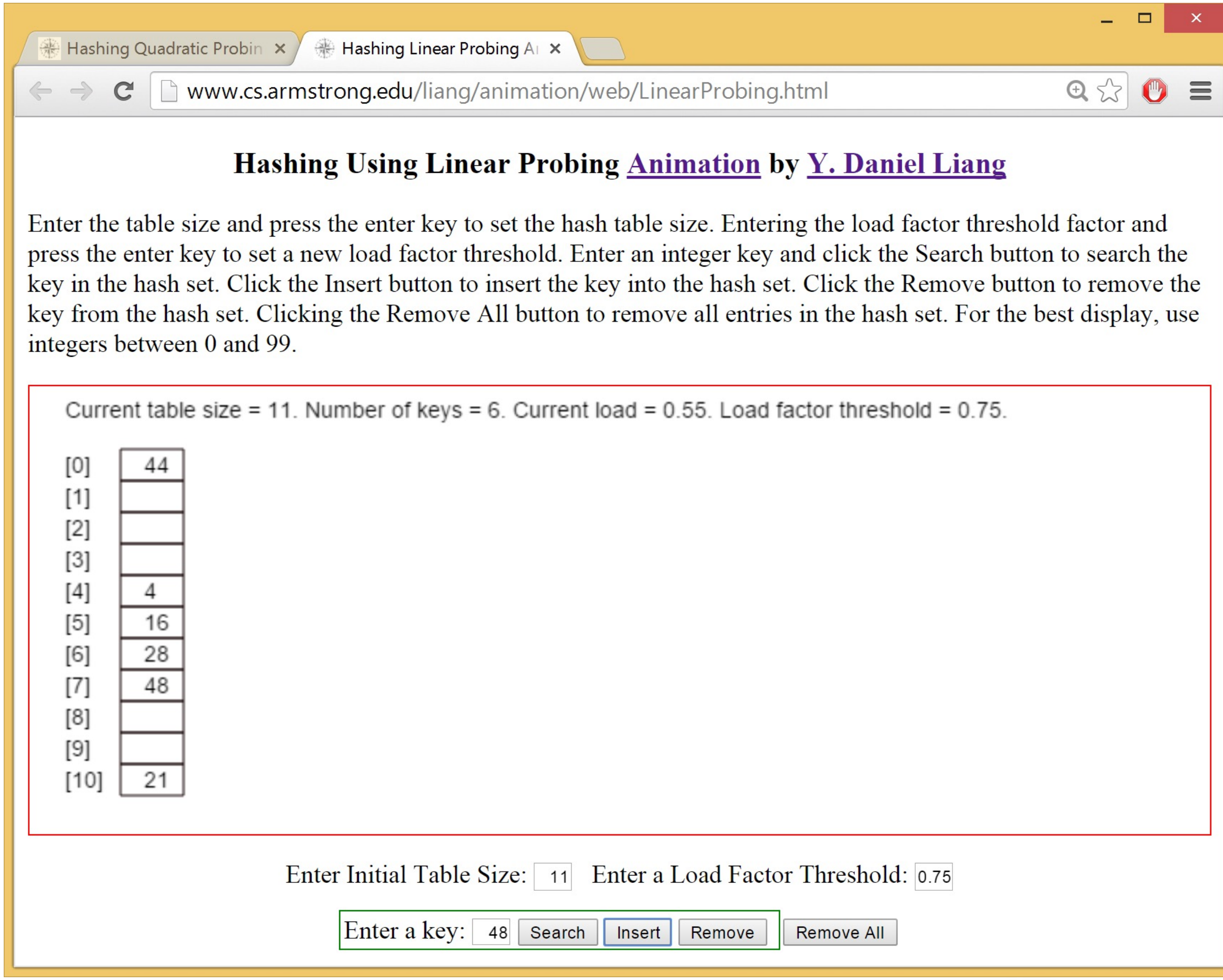
*Hashing* is a technique that retrieves the value using the index obtained from key without performing a search.

# 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

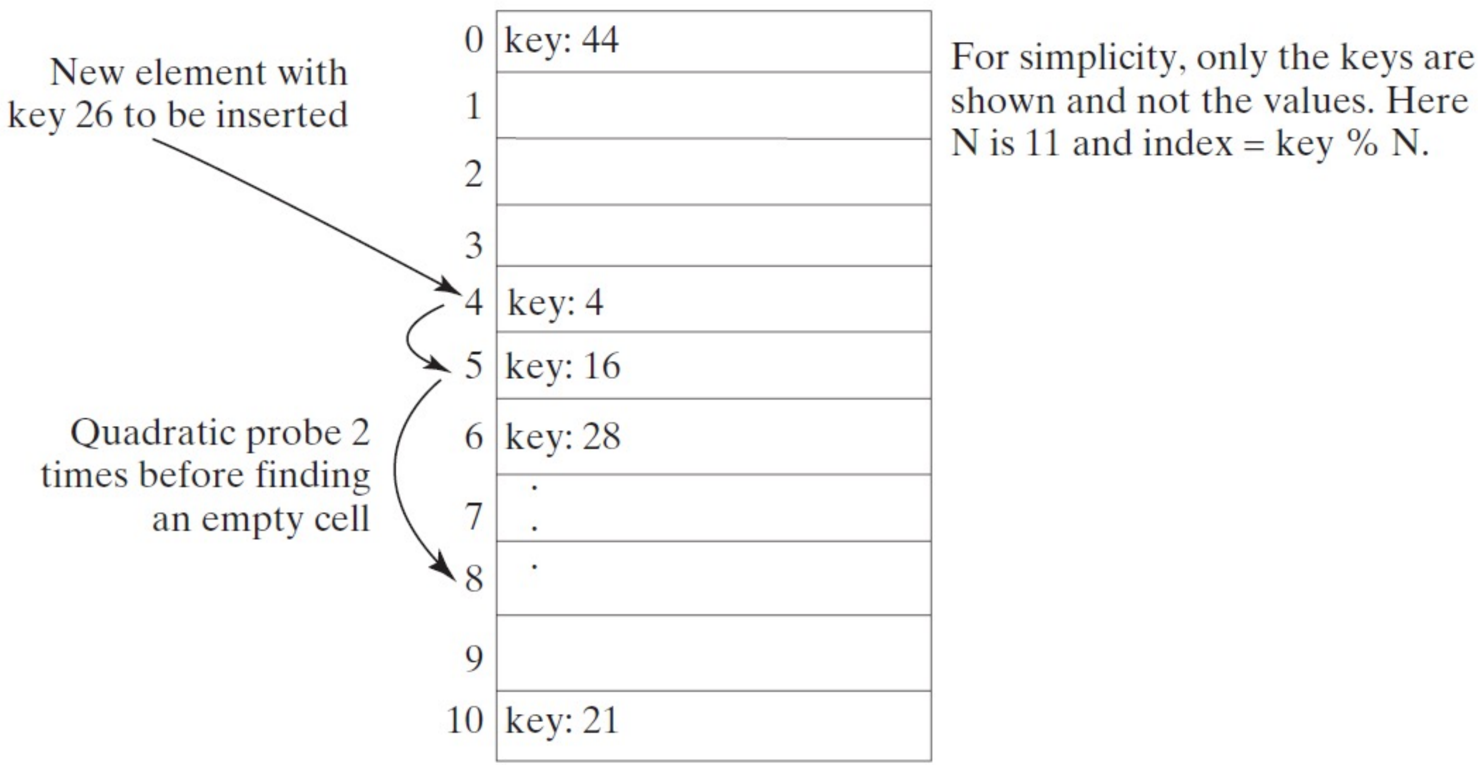
https://liveexample.pearsoncmg.com/dsanimation/LinearProbingeBook.html



# Quadratic Probing

https://liveexample.pearsoncmg.com/dsanimation/QuadraticProbingeBook.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, …

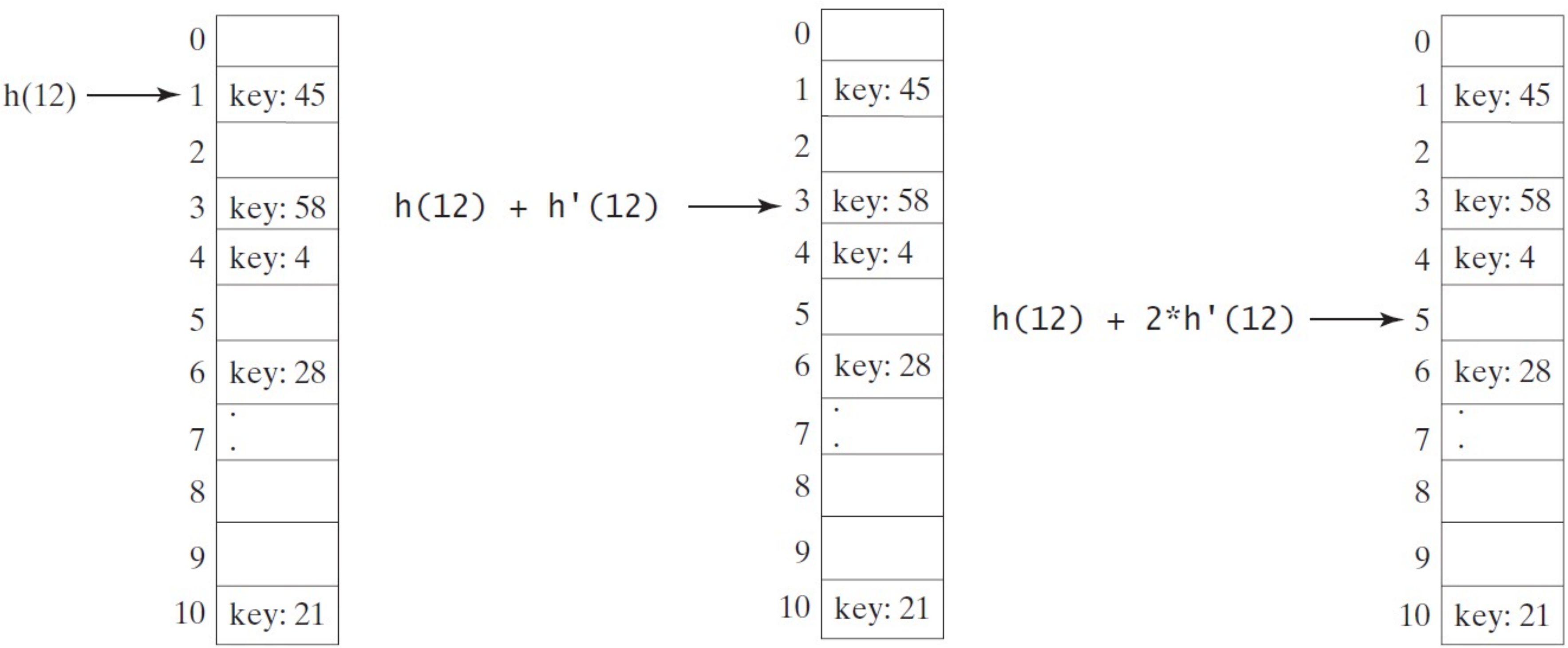


# Double Hashing

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

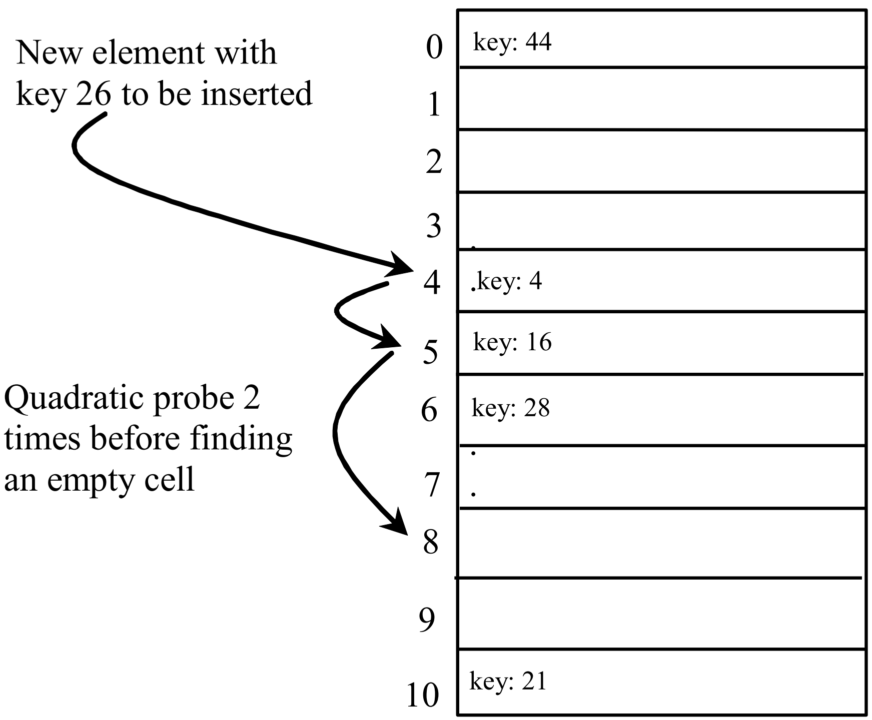
Double hashing uses a secondary hash function on the keys to determine the increments to avoid the clustering problem.

h'(k) = 7 – k % 7;



# Quadratic Probing

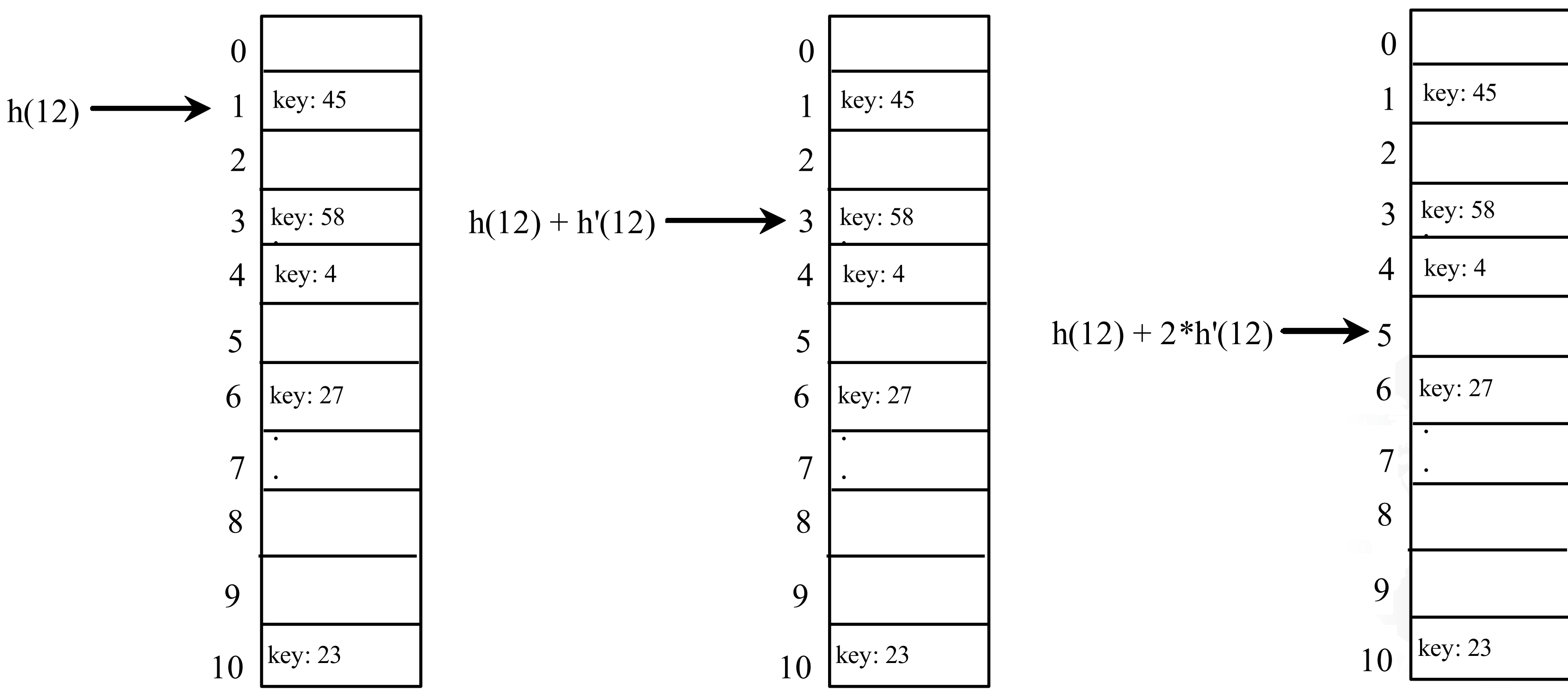
Quadratic probing can avoid the clustering problem in linear probing. Linear probing looks at the consecutive cells beginning at index k.




For simplicity, only the keys are shown and the values are not shown.

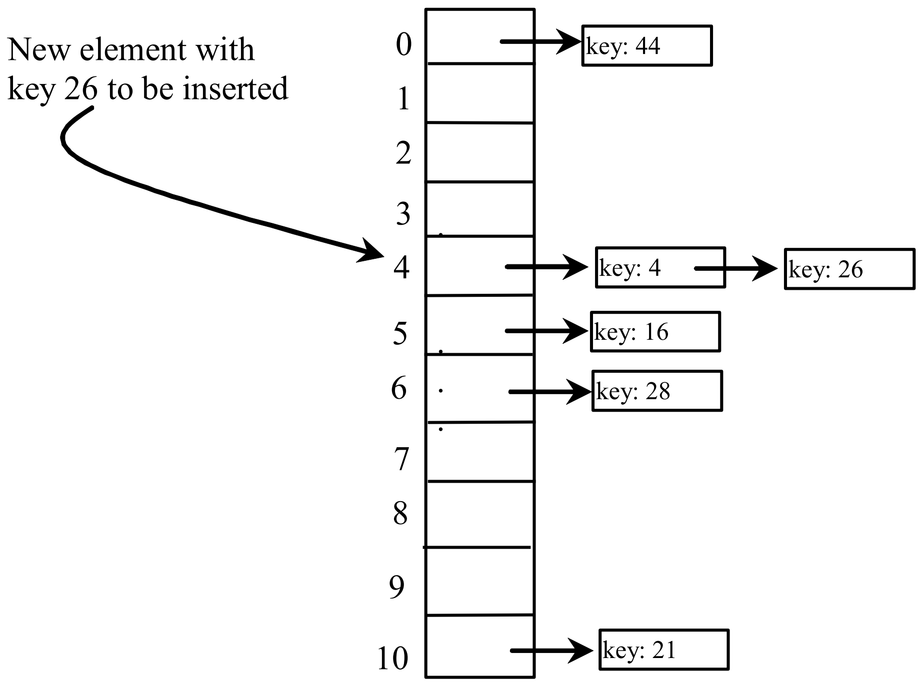
# Double Hashing

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



# Handling Collisions Using Separate Chaining

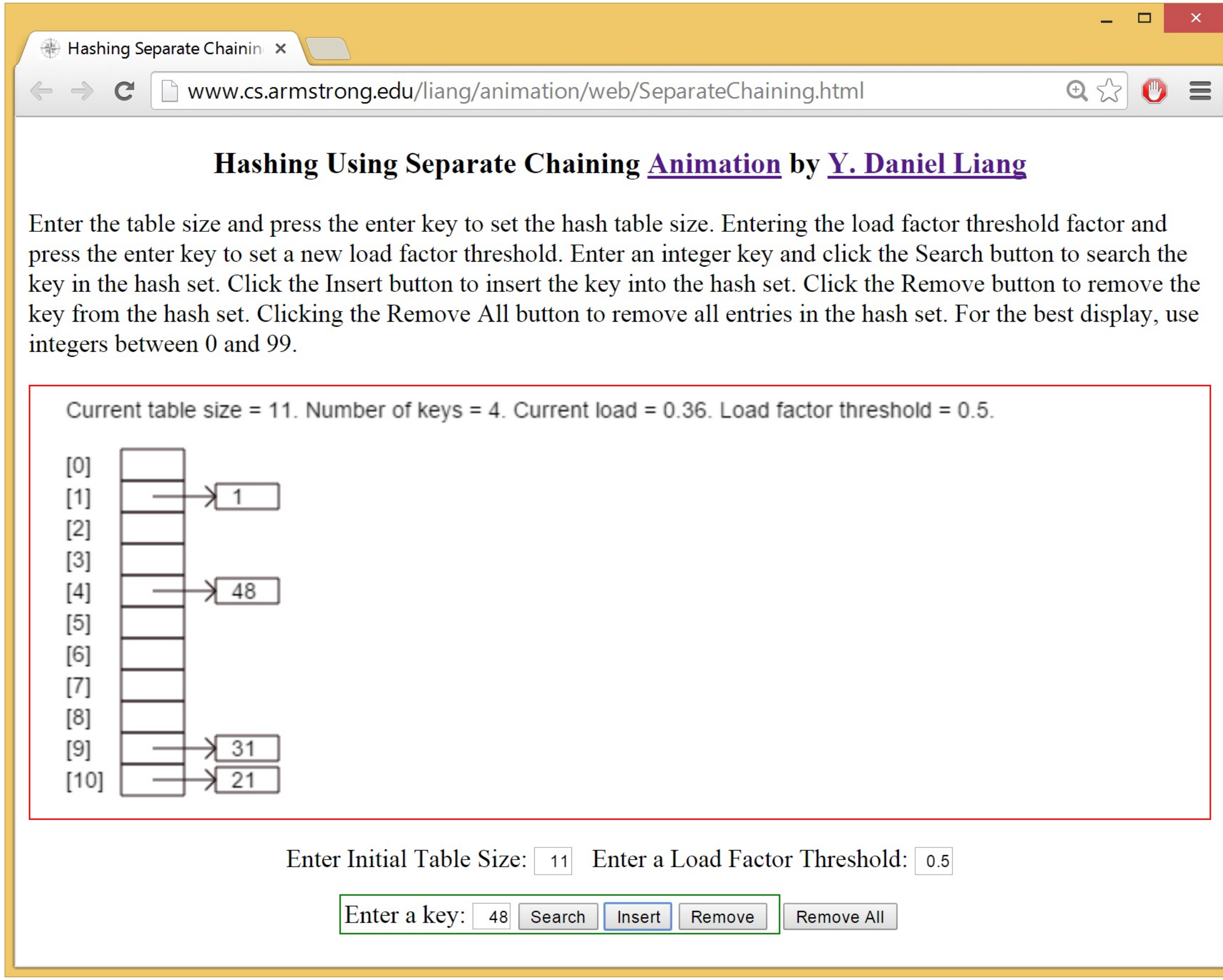
The separate chaining scheme places all entries with the same hash index into the same location, rather than finding new locations. Each location in the separate chaining scheme is called a *bucket*. A bucket is a container that holds multiple entries.




For simplicity, only the keys are shown and the values are not shown.

# Separate Chaining Animation

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



# Implementing Map Using Hashing

| Map | |
|---|---|
| size: int | The number of entries in the map. |
| table: list | Each element in the list is a bucket to hold a list of entries. |
| Map() | Constructs an empty map. |
| put(key: object, value: object): None | Adds an entry to this map. |
| remove(key: object): None | Removes an entry for the specified key. |
| containsKey(key: object): bool | Returns true if this map contains an entry for the specified key. |
| containsValue(value: object): bool | Returns true if this map maps one or more keys to the specified value. |
| items(): list | Returns a list consisting of the entries in this map. |
| get(key: object): V | Returns a value for the specified key in this map. |
| getAll(key: object): list | Returns all values for the specified key in this map. |
| keys(): list | Returns a list consisting of the keys in this map. |
| values(): list | Returns a list consisting of the values in this map. |
| clear(): None | Removes all entries from this map. |
| getSize(): int | Returns the number of mappings in this map. |
| isEmpty(): bool | Returns true if this map contains no mappings. |
| toString(): string | Returns the hash table as a string. |
| setLoadFactorThreshold(threshold: int): None | Sets a new load factor threshold. |
| toString(): str | Returns the entries in the map as a string. |
| getTable(): str | Returns the internal hash table as a string. |

Map

TestMap

# Implementing Set Using Hashing

| Set | |
|---|---|
| size: int | The number of elements in the set. |
| table: list | The hash table for storing set elements. |
| Set() | Creates an empty set. |
| add(e: object): bool | Adds the element to the set and returns true if the element is added successfully. |
| remove(e: object): bool | Removes the element from the set and returns true if the set contained the element. |
| clear(): void | Removes all elements from this set. |
| contains(e: object): bool | Returns true if the element is in the set. |
| isEmpty(): bool | Returns true if this set contains no elements. |
| getSize(): int | Returns the number of elements in this set. |
| union(s: Set): Set | Set union. |
| difference(s: Set): Set | Set difference. |
| intersect(s: Set): Set | Set intersection. |
| toString(): str | Returns a string representation for the set. |
| getTable(): str | Returns the internal hash table as a string. |

Set

TestSet