Chapter 27 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(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?

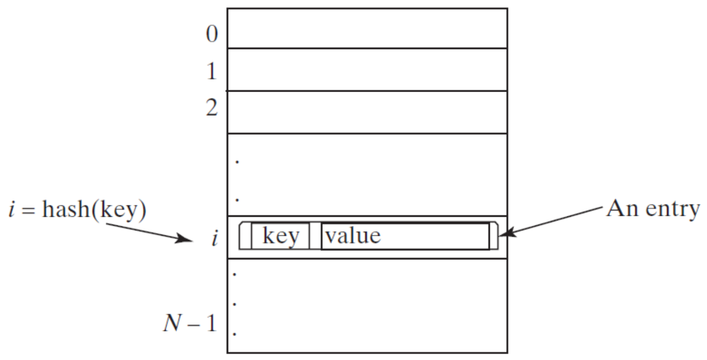
If you know the index of an element in the array, you can retrieve the element using the index in O(1) time. So, can we store the values in an array and use the key as the index to find the value? The answer is yes if you can map a key to an index.

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

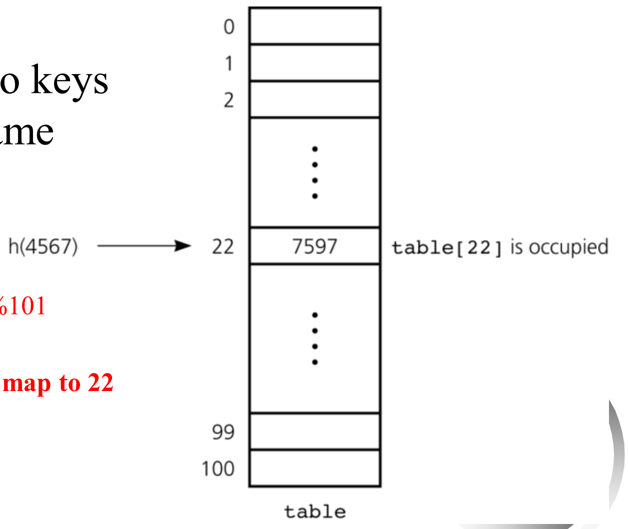
A typical hash function first converts a search key to an integer value called a *hash code*, and then compresses the hash code into an index to the hash table.



Hash function

Collisions

Collision: two keys map to the same index



Hash function: key%101

both 4567 and 7597 map to 22

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

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

The Birthday Problem: Probabilities

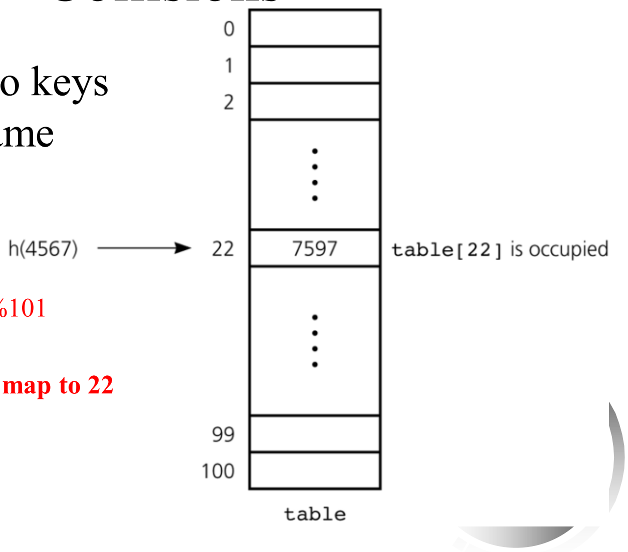
N: # of people	P(N): probability that at least two of the N people have the same birthday.
10	11.7 %
20	41.1 %
23	50.7 %
30	70.6 %
50	97. 0 %
57	99.0%
100	99.99997%
200	99.9999999999999999999999999998%
366	100%

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!

Collisions

Collision: two keys map to the same index



Hash function: key%101

both 4567 and 7597 map to 22

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** (e.g. ArrayList, or linked list), often called **separate chaining**
  – **Extendable dynamic hashing**

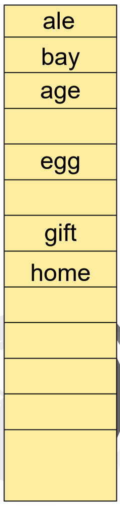
Open addressing

- 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.

In the sequel we use **step=1** for linear probing examples

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



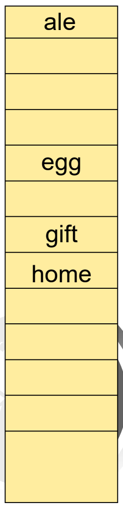
Question: During the process of linear probing, if there is an empty spot,
A. Item not found ?
or
B. There is still a chance to find the item ?

Open addressing: Linear Probing

- **Deletion:** The 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.

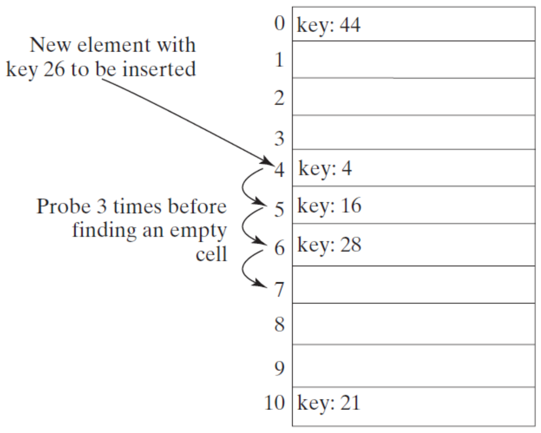
Linear Probing (cont.)

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



Question: Where does almond go now?

Linear Probing Animation

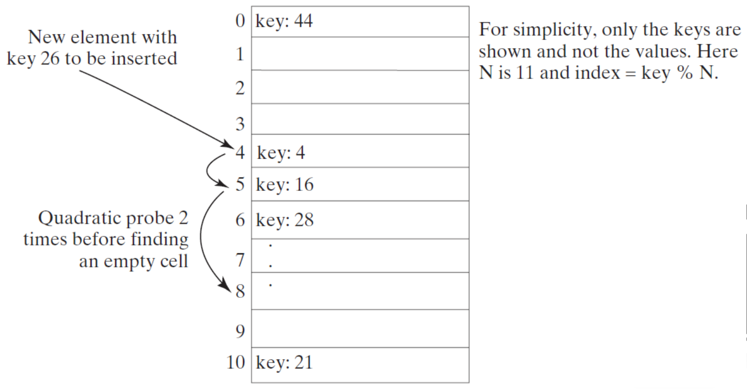

New element with key 26 to be inserted
Probe 3 times before finding an empty cell
0 key: 44
1
2
3
4 key: 4
5 key: 16
6 key: 28
7
8
9
10 key: 21

For simplicity, only the keys are shown and the values are not shown. Here N is 11 and index = key % N.

Quadratic Probing

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

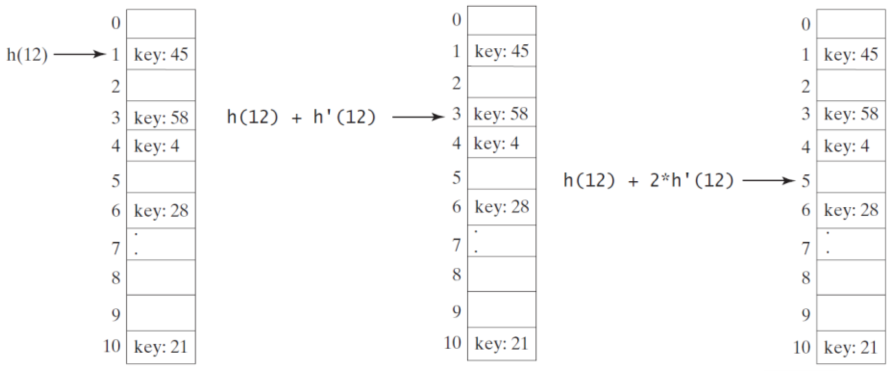
Quadratic probing can avoid the clustering problem in linear probing. Linear probing looks at the consecutive cells beginning at index k. Quadratic probing increases the index by j^2 for j = 1, 2, 3, ... The actual index searched are k, k + 1, k + 4, …


New element with key 26 to be inserted
Quadratic probe 2 times before finding an empty cell
0 key: 44
1
2
3
4 key: 4
5 key: 16
6 key: 28
7
8
9
10 key: 21
For simplicity, only the keys are shown and not the values. Here N is 11 and index = key % N.

Double Hashing

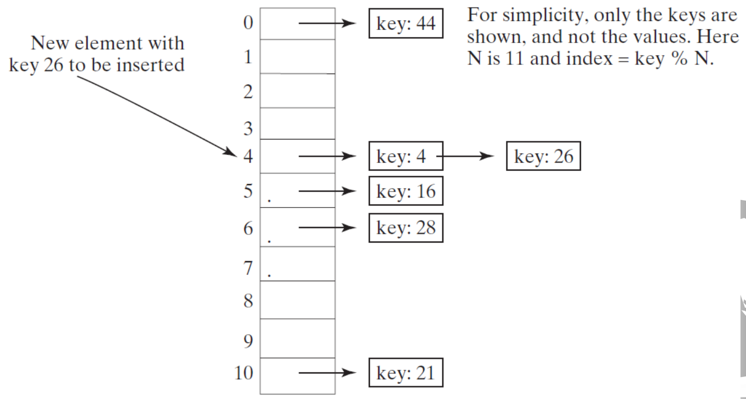
Double hashing uses a secondary hash function on the keys to determine the increments to avoid the clustering problem.

h’(k) = 7 – k % 7;

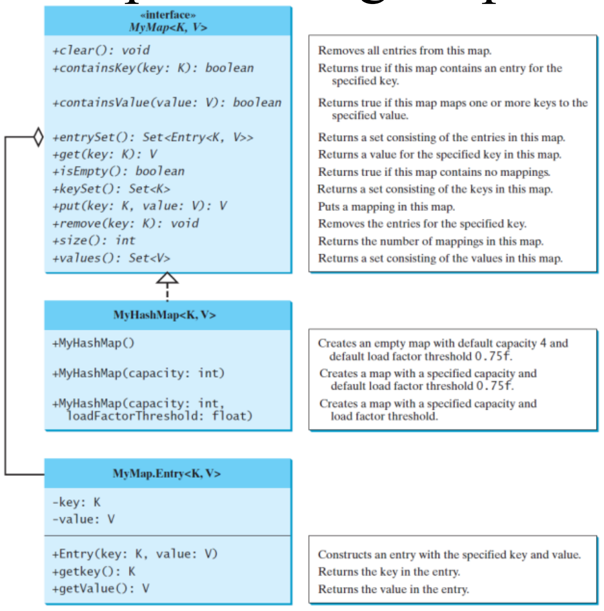


Handling Collisions Using Separate Chaining

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.


New element with key 26 to be inserted
0
1
2
3
4
5
6
7
8
9
10
key: 44
key: 4
key: 26
key: 16
key: 28
key: 21
For simplicity, only the keys are shown, and not the values. Here N is 11 and index = key % N.

Implementing Map Using Hashing



«interface»
MyMap<K, V>

+clear(): void	Removes all entries from this map.
+containsKey(key: K): boolean	Returns true if this map contains an entry for the specified key.
+containsValue(value: V): boolean	Returns true if this map maps one or more keys to the specified value.
+entrySet(): Set<Entry<K, V>>	Returns a set consisting of the entries in this map.
+get(key: K): V	Returns a value for the specified key in this map.
+isEmpty(): boolean	Returns true if this map contains no mappings.
+keySet(): Set<K>	Returns a set consisting of the keys in this map.
+put(key: K, value: V): V	Puts a mapping in this map.
+remove(key: K): void	Removes the entries for the specified key.
+size(): int	Returns the number of mappings in this map.
+values(): Set<V>	Returns a set consisting of the values in this map.

MyHashMap<K, V>

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

MyMap.Entry<K, V>

-key: K
-value: V

+Entry(key: K, value: V)	Constructs an entry with the specified key and value.
+getkey(): K	Returns the key in the entry.
+getValue(): V	Returns the value in the entry.

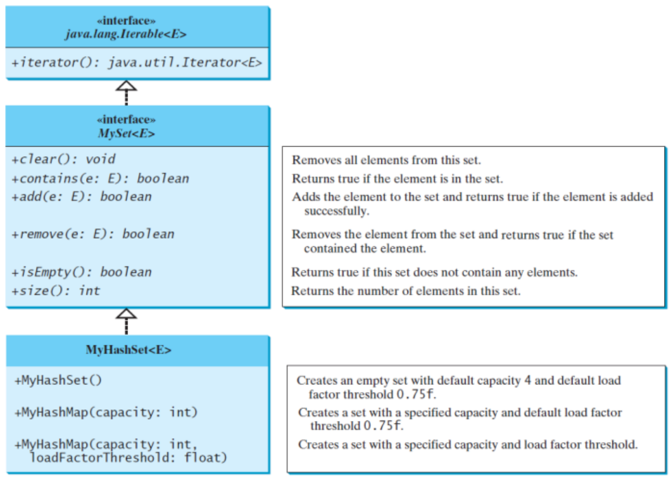
MyMap

MyHashMap

TestMyHashMap

Run

Implementing Set Using Hashing



+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()	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.

MySet

MyHashSet

TestMyHashSet

Run