Chapter 8: Map ADT

CS401

Michael Y. Choi, Ph.D.

Department of Computer Science
Illinois Institute of Technology

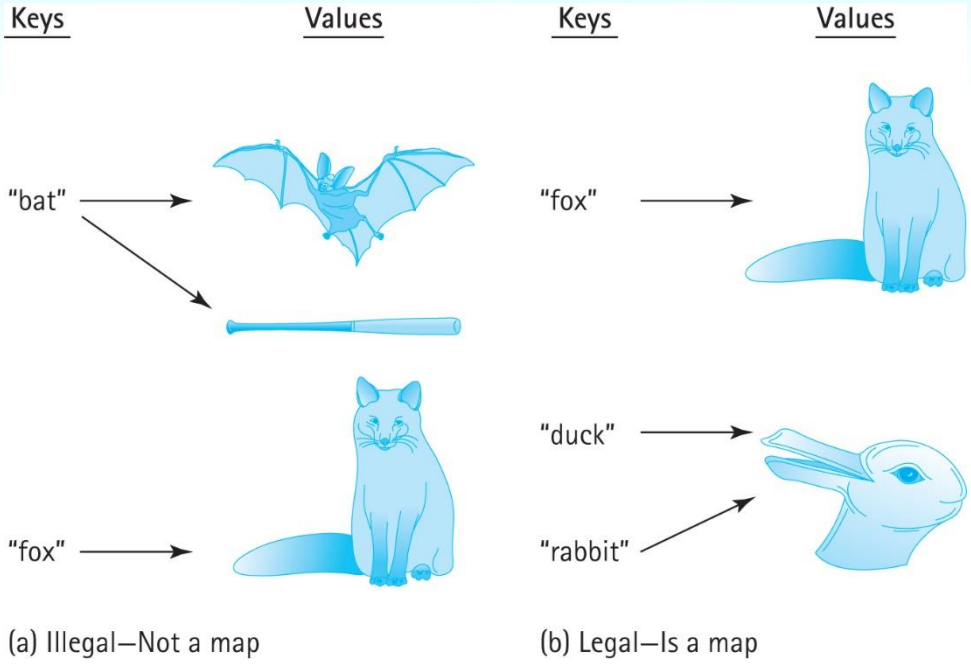
Revised Nell Dale Presentation

Chapter 8: The Map ADT

8.1 – The Map Interface
8.2 – Map Implementations
8.3 – Application: String-to-String Map
8.4 – Hashing
8.5 – Hash Functions
8.6 – A Hash-Based Map
8.7 – Map Variations

8.1 The Map Interface

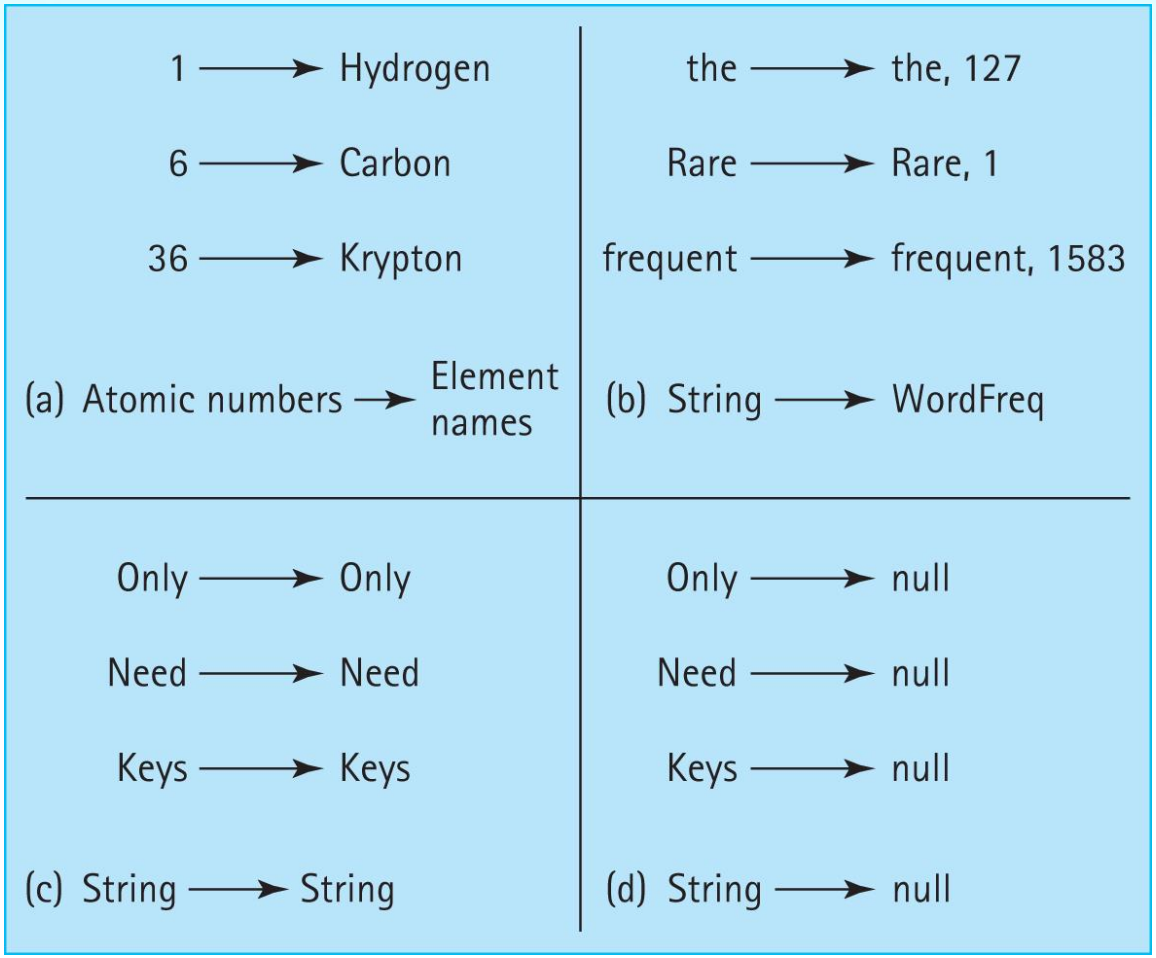
- Maps associate a key with exactly one value
- In other words
  - a map structure does not permit duplicate keys
  - but two distinct keys can map onto the same value



(a) Illegal—Not a map

(b) Legal—Is a map

Legal mapping variations



MapInterface

```
//-------------------------------------------------------------------
// MapInterface.java           by Dale/Joyce/Weems                Chapter 8
//
// A map provides (K = key, V = value) pairs, mapping the key onto
// the value.
// Keys are unique. Keys cannot be null.
//
// Methods throw IllegalArgumentException if passed a null key argument.
//
// Values can be null, so a null value returned by put, get, or remove does
// not necessarily mean that an entry did not exist.
//-------------------------------------------------------------------
//
// . . . continued on next slide
```

```
package ch08.maps;
import java.util.Iterator;

public interface MapInterface<K, V> extends Iterable<MapEntry<K,V>>
{
  V put(K k, V v);
  // If an entry in this map with key k already exists then the value
  // associated with that entry is replaced by value v and the original
  // value is returned; otherwise, adds the (k, v) pair to the map and
  // returns null.

  V get(K k);
  // If an entry in this map with a key k exists then the value associated
  // with that entry is returned; otherwise null is returned.

  V remove(K k);
  // If an entry in this map with key k exists then the entry is removed
  // from the map and the value associated with that entry is returned;
  // otherwise null is returned.
  //
  // Optional. Throws UnsupportedOperationException if not supported.

  // Also requires contains(K key), isFull(), isEmpty() and size()
}
```

Iteration

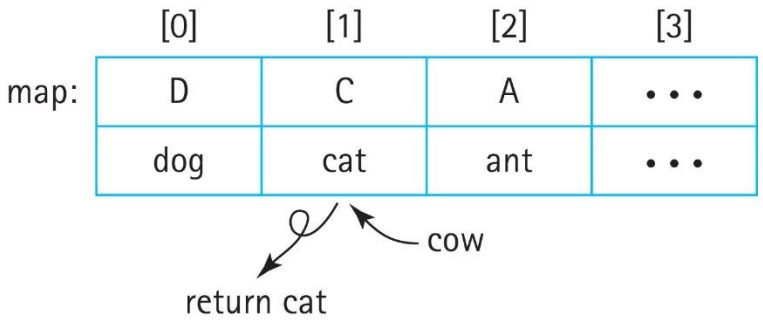
- We require an iteration that returns key-value pairs
- The class `MapEntry` represents the key-value pairs
  - requires the key and value to be passed as constructor arguments
  - provides getter operations for both key and value
  - provides a setter operation for the value
  - provides a `toString`

MapExample

- Instructors can now discuss and demonstrate the `MapExample` application found in the `ch08.apps` package … it uses the `ArrayListMap` class presented in the next section

8.2 Map Implementations

- Unsorted Array
  - The `put` operation creates a new `MapEntry` object and performs a brute force search (O(N)) of all the current keys in the array to prevent key duplication
  - If a duplicate key is found, then the associated `MapEntry` object is replaced by the new object and its `value` attribute is returned, for example `map.put(cow)`

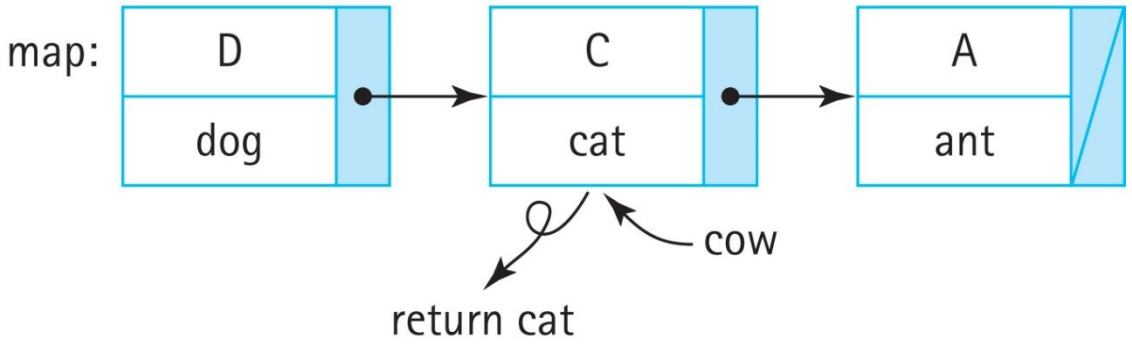


Map Implementations

- Unsorted Array
  - Like `put`, the `get`, `remove`, and `contains` operations would all require brute force searches of the current array contents, so they are all O(N)
- Sorted Array
  - The binary search algorithm can be used, greatly improving the efficiency of the important `get` and `contains` operations.
  - Although it is not a requirement, in general it is expected that a map will provide fast implementation of these two operations.

Map Implementations

- Unsorted Linked List
  - Similar to an unsorted array, most operations require brute force search
  - In terms of space, a linked list grows and shrinks as needed so it is possible that some advantage can be found in terms of memory management, as compared to an array.

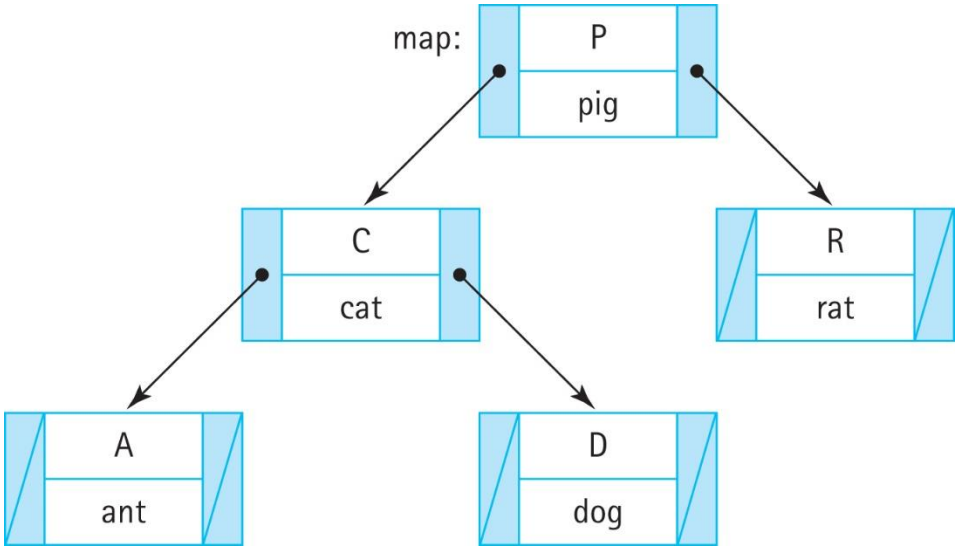


Map Implementations

- Sorted Linked List
  - Even though a linked list is kept sorted, it does not permit use of the binary search algorithm as there is no efficient way to inspect the “middle” element.
  - So there is not much advantage to using a sorted linked list to implement a map, as compared to an unsorted linked list

Map Implementations

- Binary Search Tree
  - If a map can be implemented as a *balanced* binary search tree, then all of the primary operations (put, get, remove, and contains) can exhibit efficiency $O(\log_2 N)$.



`ArrayListMap`

- Instructors can now discuss the `ArrayListMap` class found in the `ch08.maps` package and review the associated notes found on pages 511 and 512

8.3 Application: String-to-String Map

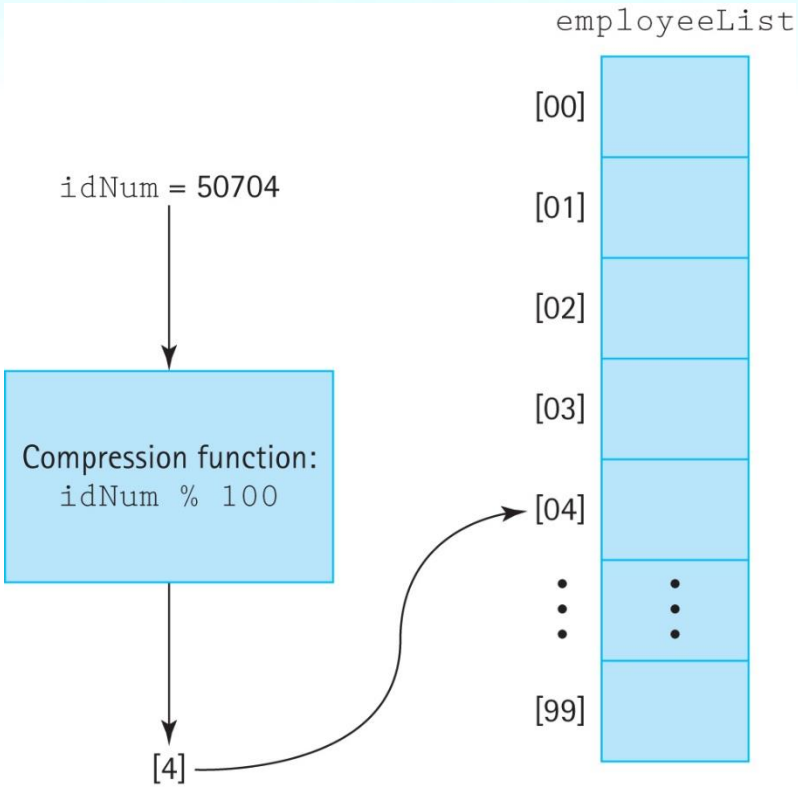
- The `StringPairApp` found in the `ch08.apps` package reads # separated pairs of strings (key # value) from specified input file and then allows the user to enter keys and reports back to them the associated value if there is one.
- It is a short yet versatile application that demonstrates the use of our Map ADT

8.4 Hashing

- An efficient approach to implementing a Map
- Typically provides O(1) operation implementation
- Uses an array (typically called a hash table) to hold the key/value pairs
- Hashing involves determining array indices directly from the key of the entry being stored/accessed.

Compression function

- If we have a positive integral key, such as an ID number, we can just use the key as the index into the array
- If the range of key values is larger than the array we must "compress" the key into a usable index

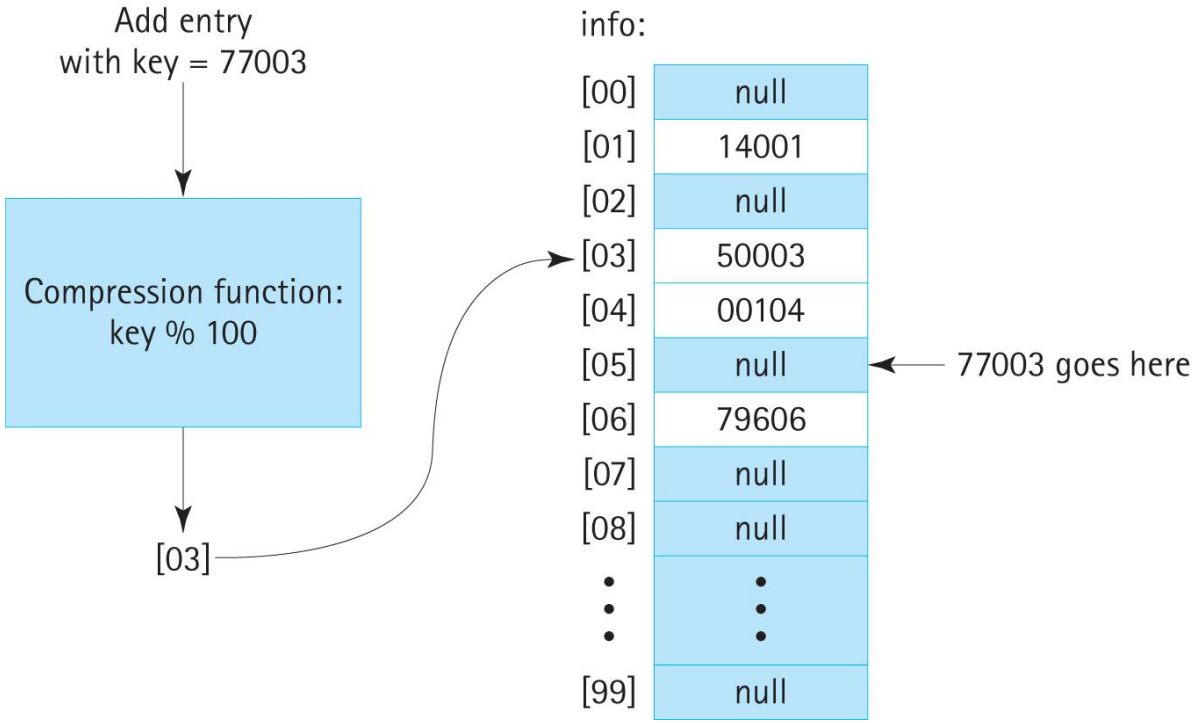


Collisions

- If two keys compress to the same array location we call it a “collision”
- minimizing such collisions is the biggest challenge in designing a good hashing system
  - we cover this in Section 8.5, “Hash Functions.”
- In our discussion of collision resolution policies, we will assume
  - use of an array `info` to hold the information
  - the int variable `location` to indicate an array/hash-table slot.

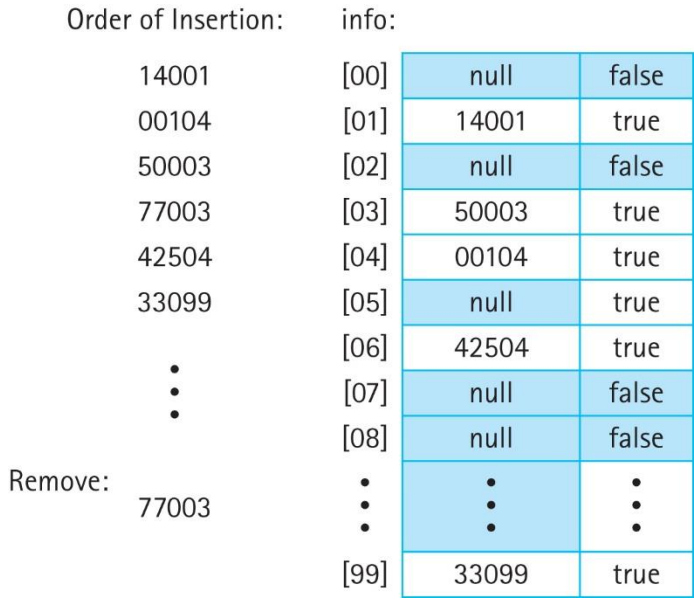
Collision resolution policies

- Linear probing: store the colliding entry into the next available space:



Item Removal

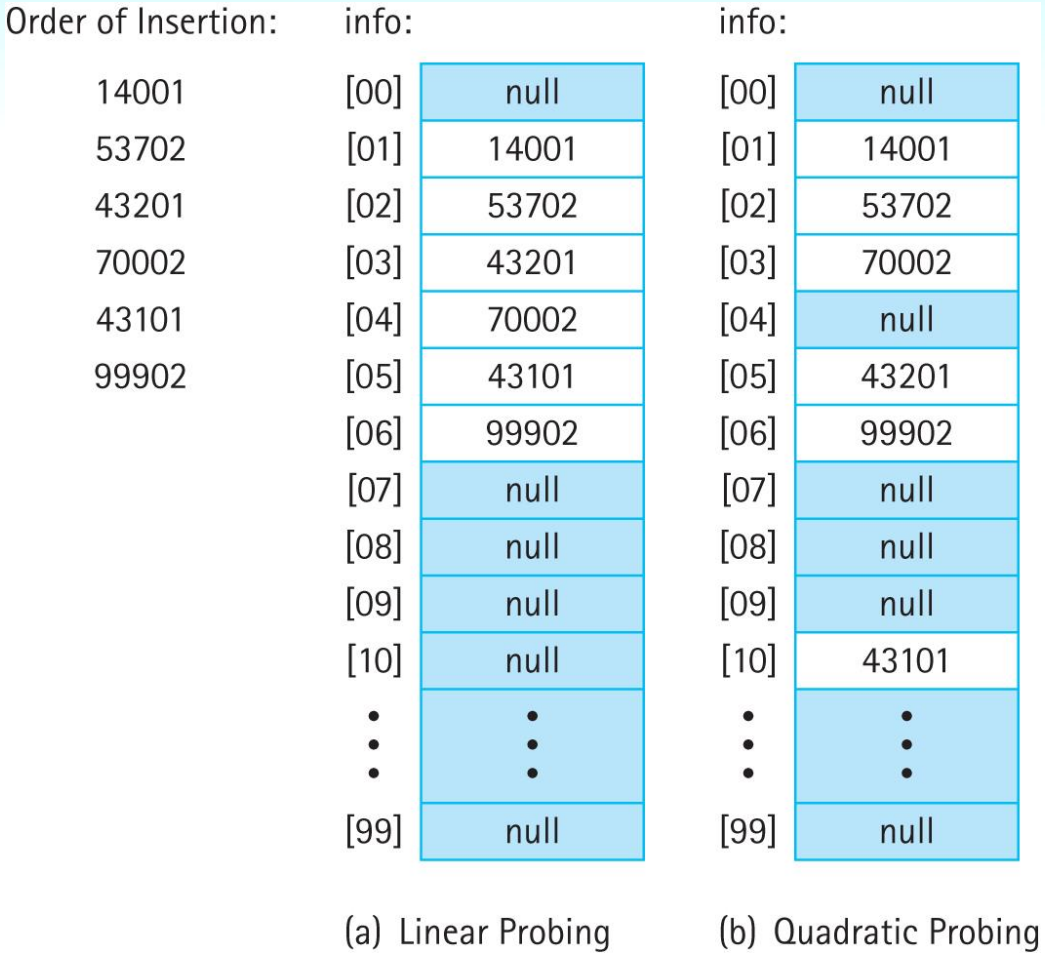
- Complicates searching: can not terminate search upon finding a null entry therefore:
  - use a special value for a removed entry
  - use a boolean value associated with each hash table slot:
  - disallow removal



Collision resolution policies

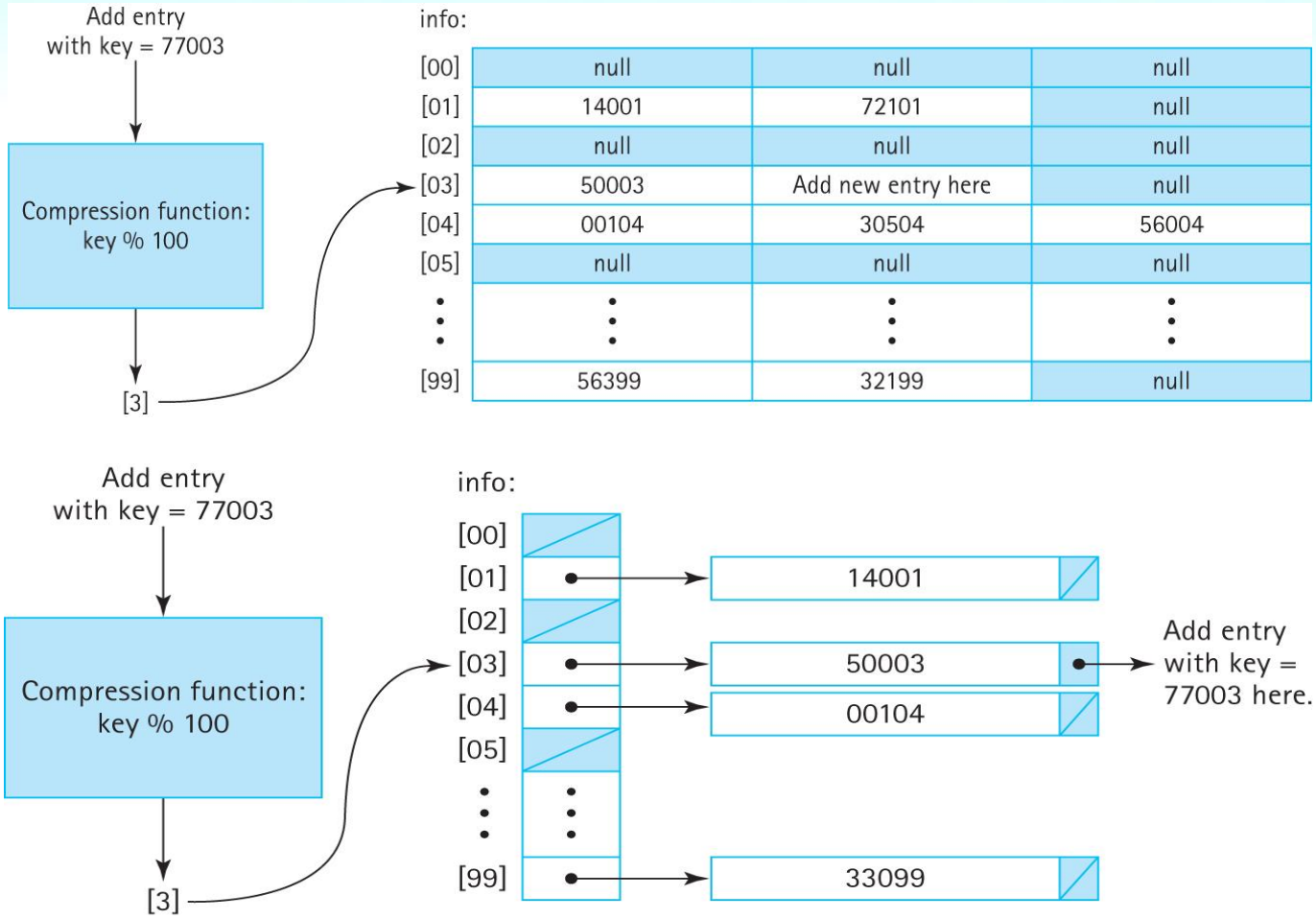
- Linear probing approach can lead to inefficient clusters of entries
- Quadratic probing: the value added at each step is dependent on how many locations have already been inspected.
  - The first time it looks for a new location it adds 1 to the original location
  - the second time it adds 4 to the original location
  - the third time it adds 9 to the original location
  - and so on—the i^{th} time it adds i^2:

Comparison


Order of Insertion:
14001
53702
43201
70002
43101
99902
info:
[00] null
[01] 14001
[02] 53702
[03] 43201
[04] 70002
[05] 43101
[06] 99902
[07] null
[08] null
[09] null
[10] null
[99] null
(a) Linear Probing
info:
[00] null
[01] 14001
[02] 53702
[03] 70002
[04] null
[05] 43201
[06] 99902
[07] null
[08] null
[09] null
[10] 43101
[99] null
(b) Quadratic Probing

Collision resolution policies

Buckets and Chaining



8.5 Hash Functions

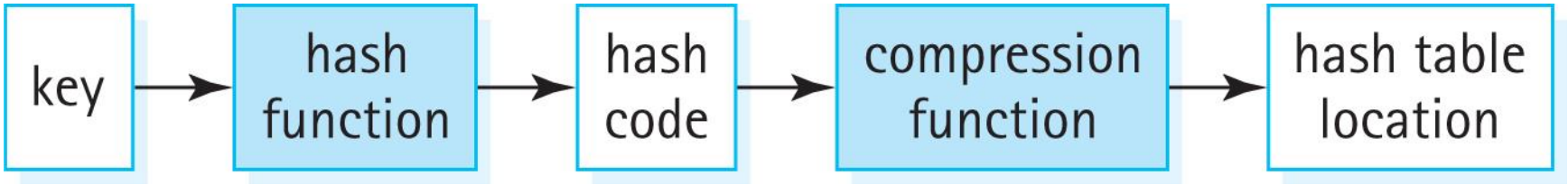
- To get the most benefit from a hashing system we need for the eventual locations used in the underlying array to be as spread out as possible.
- Two factors affect this spread
  - the size of the underlying array
  - the set of integral values presented to the compression function

Array Size

- space versus time trade-off
- hash systems will often monitor their own load—the percentage of array indices being used
- Once the load reaches a certain level, for example 75%, the system is rebuilt using a larger array
- This approach is often called “rehashing” because after the array is enlarged all of the previous entries have to be reinserted into the new array

The Hash Function

- Keys might not be integral
- Even if integral, keys might not provide a good "spread"
- So, we add another step, the hash function:



- Synonyms for "hash code" include "hash value," and sometimes we just use the word "hash."

Creating a hash function

- Selecting: Identify selected parts of the key – try to use parts that will provide a good variety of results
- Digitizing: the selected parts must be transformed to integers
- Combining: combine the resultant integers using a mathematical function

Considerations

- A hash code is not unique. Do not use a hash code as a key.
- If two entries are considered to be equal, then they should hash to the same value.
- When defining a hash function, consider the work required to calculate it.
- A precise analysis of the complexity of hashing depends on the domain and distribution of keys, the hash function, the size of the table, and the collision resolution policy. In practice it is usually not difficult to achieve close to O(1) efficiency using hashing.

Java's Support for Hashing

- The Java Library includes a `HashMap` class (discussed in Section 8.7) and a `HashSet` class that use hash techniques to support storing objects
- The Java `Object` class exports a `hashCode` method that returns an `int` hash code.
  - The standard Java hash code for an object is a function of the object's memory location.
- For most applications, hash codes based on memory locations are not usable. Many of the Java classes that define commonly used objects (such as `String` and `Integer`), override the `Object` class's `hashCode` method.
- If you plan to use hash tables in your programs, you should do likewise.

8.6 A Hash-Based Map

`Hmap.java`

- is implemented with an internal hash table that uses the `hashCode` method of the key class
- is unbounded
- has a default capacity of 1,000 and a default load factor of 75%
- does not support the `remove` operation
- is located in the `ch08.maps` package
- is used by the `VocDensMeasureHMap` application located in the `ch08.apps` package

8.7 Map Variations

- Some programming languages, (e.g., Awk, Haskell, JavaScript, Lisp, MUMPS, Perl, PHP, Python, and Ruby), directly support
- Many other languages, including Java, C++, Objective-C, and Smalltalk provide map functionality through their standard code libraries

Maps are known by many names:

- Symbol table - one of the first carefully studied and designed data structures, and were related to compiler design
- Dictionary - the idea of looking up a word (the key) in a dictionary to find its definition (the value) makes the concept of a dictionary a good fit for maps
- Hashes - because a hash system is a very efficient and common way to implement a map, you will sometimes see the two terms used interchangeably
- Associative Arrays - You can view a map as an array—one that associates keys with values rather than indices with values.