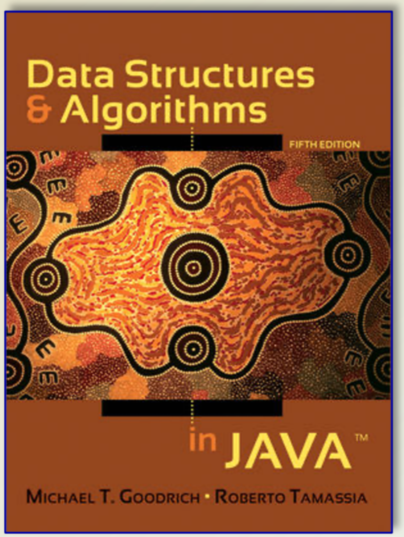
Data Structure & Algorithms in JAVA

5th edition

Michael T. Goodrich
Roberto Tamassia



Chapter 9: Hash Tables, Maps, and Skip Lists

CPSC 3200

Algorithm Analysis and Advanced Data Structure

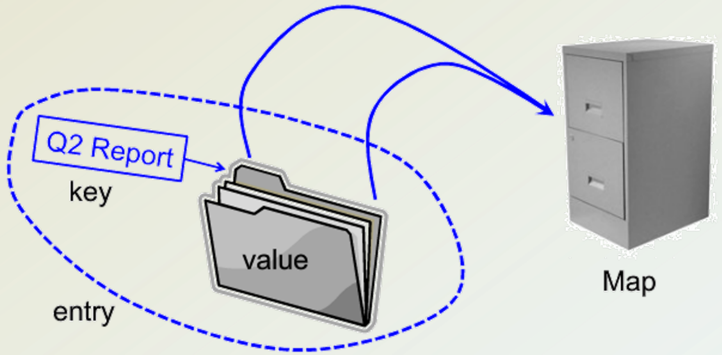
Chapter Topics

- Maps.
- Hash Tables.
- Dictionaries.

Maps

- A map models a searchable collection of key-value entries.
- A map stores **key-value** pairs (***k***, ***v***) which we call ***entries***.
- The main operations of a map are for **searching**, **inserting**, and **deleting** items.
- Multiple entries with the same **key** are not allowed (map ADT requires each key to be unique).

- **Applications:**
  - address book.
  - student-record database.



The Map ADT

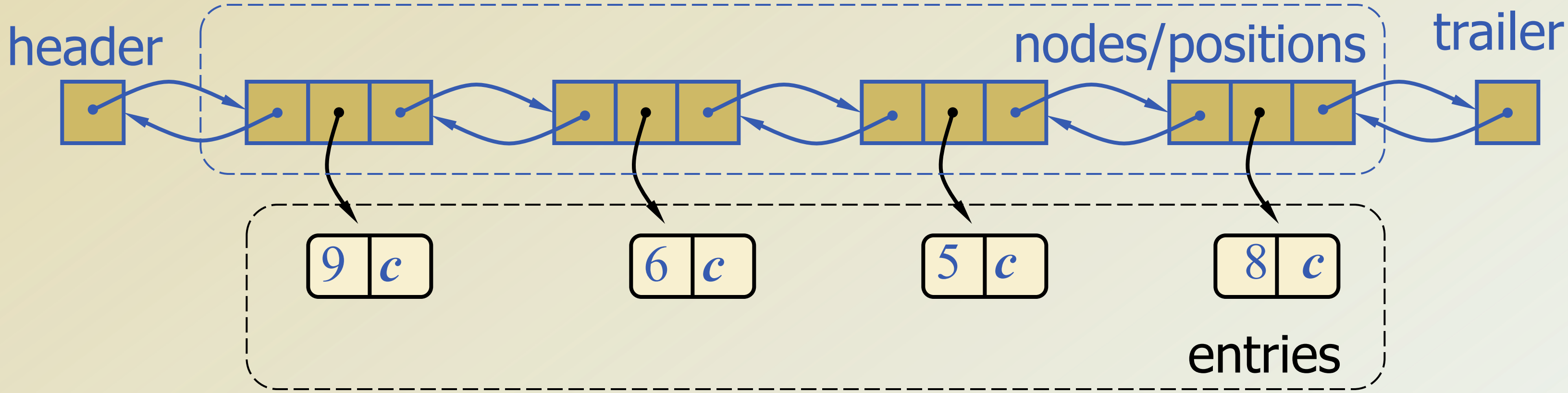
- **get(k):** if the map **M** has an entry with key ***k***, return its associated value; else, return null .
- **put(k, v):** insert entry (***k***, ***v***) into the map **M**; if key ***k*** is not already in **M**, then return null; else, return old value associated with ***k.***
- **remove(k):** if the map **M** has an entry with key ***k***, remove it from **M** and return its associated value; else, return null.
- **size( )**, **isEmpty( )**
- **entrySet( ):** return an iterable collection of the entries in **M**
- **keySet( ):** return an iterable collection of the keys in **M**
- **values( ):** return an iterator of the values in **M**

Example

Operation	*Output*	*Map*
isEmpty()	**true**	$\emptyset$
put(5, A)	**null**	$\{(5,A)\}$
put(7, B)	**null**	$\{(5,A),(7,B)\}$
put(2, C)	**null**	$\{(5,A),(7,B),(2,C)\}$
put(8, D)	**null**	$\{(5,A),(7,B),(2,C),(8,D)\}$
put(2, E)	C	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(7)	B	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(4)	**null**	$\{(5,A),(7,B),(2,E),(8,D)\}$
get(2)	E	$\{(5,A),(7,B),(2,E),(8,D)\}$
size()	4	$\{(5,A),(7,B),(2,E),(8,D)\}$
remove(5)	A	$\{(7,B),(2,E),(8,D)\}$
remove(2)	E	$\{(7,B),(8,D)\}$
get(2)	**null**	$\{(7,B),(8,D)\}$
isEmpty()	**false**	$\{(7,B),(8,D)\}$
entrySet()	$\{(7,B),(8,D)\}$	$\{(7,B),(8,D)\}$
keySet()	$\{7,8\}$	$\{(7,B),(8,D)\}$
values()	$\{B,D\}$	$\{(7,B),(8,D)\}$

A Simple List-Based Map

- We can efficiently implement a map using an unsorted list
- We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order.



- The unsorted list implementation is effective only for maps of **small size** (e.g., historical record of logins to a workstation)

The get(k) Algorithm

Algorithm get(***k***):

Input: A key ***k***

Output: The value for key k in **M**, or null if there is no key ***k*** in **M**

```
for each position p in S.positions( ) do
  if p.element( ).getKey( ) = k then
      return p.element( ).getValue( )
return null  {there is no entry with key equal to k}
```

Time complexity ?

The put(k,v) Algorithm

Algorithm put(k,v):
Input: A key-value pair (k,v)
Output: The old value associated with key k in M, or null if k is new

```
for each position p in S.positions( ) do
  if p.element( ).getKey( ) = k then
      t ← p.element( ).getValue( )
      B.set(p,(k,v))
      return t  {return the old value}
S.addLast((k,v))
n ← n+1  {increment variable storing number of entries}
return null  {there was no previous entry with key equal to k}
```

Time complexity ?

The remove(k) Algorithm

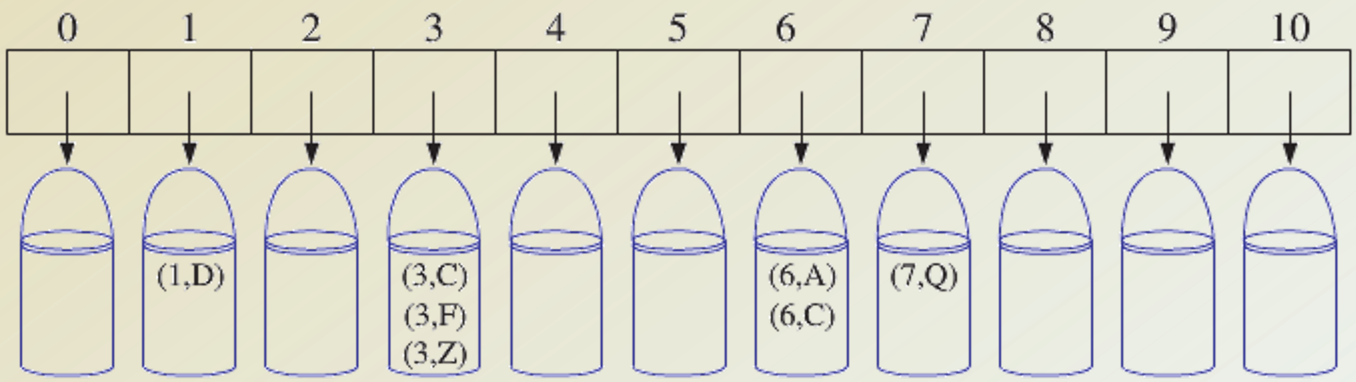
Algorithm remove(k):
Input: A key k
Output: The (removed) value for key k in M, or null if k is not in M

```
for each position p in S.positions( ) do
if p.element().getKey() = k then
  t ← p.element().getValue()
  S.remove(p)
  n ← n−1  {decrement variable storing number of entries}
  return t  {return the removed value}
return null  {there is no entry with key equal to k}
```

Time complexity ?

Hash Functions and Hash Tables

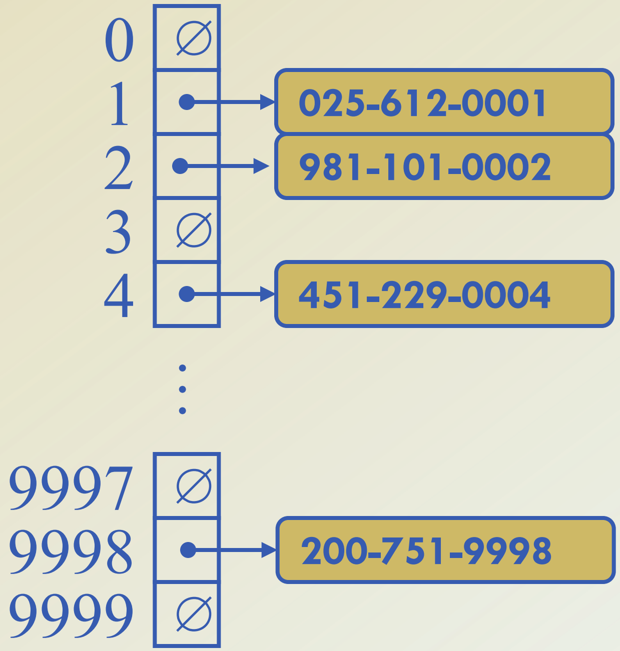
- A hash table for a given key type consists of
  - Hash function ***h***
  - Array (called table) of size ***N***
- When implementing a map with a hash table, the goal is to store item (***k***, ***v***) at index ***i*** = ***h***(***k***)
- A hash function ***h*** maps keys of a given type to integers in a fixed interval [0, ***N*** - 1]



- The integer ***h***(***k***) is called the hash value of key ***k***

Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer.
- Our hash table uses an array of size N = 10,000 and the hash function $h(x)$ = last four digits of x



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:

  mapping the key $\boldsymbol{k}$ to integer
  h_1: keys $\rightarrow$ integers

  Compression function:

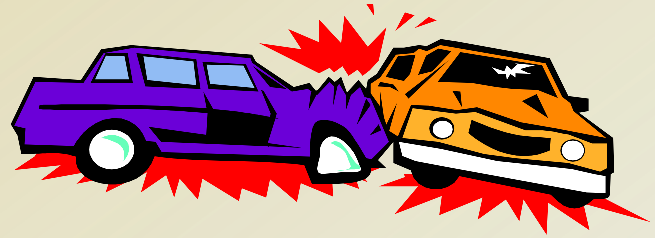
  mapping the hash code to an integer in range of indices [0, N-1]
  h_2: integers $\rightarrow$ $[0, N - 1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

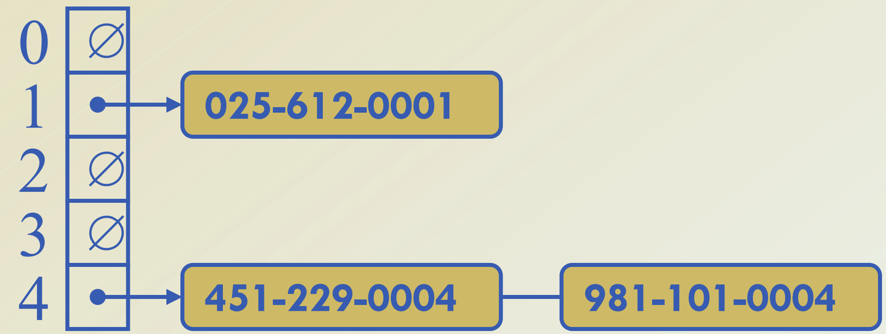
$$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to "disperse" the keys in an apparently random way.

Collision Handling



- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table point to a linked list of entries that map there



- Separate chaining is simple, but requires additional memory outside the table

Map with Separate Chaining

Delegate operations to a list-based map at each cell:

Algorithm get(k):
return A[h(k)].get(k)

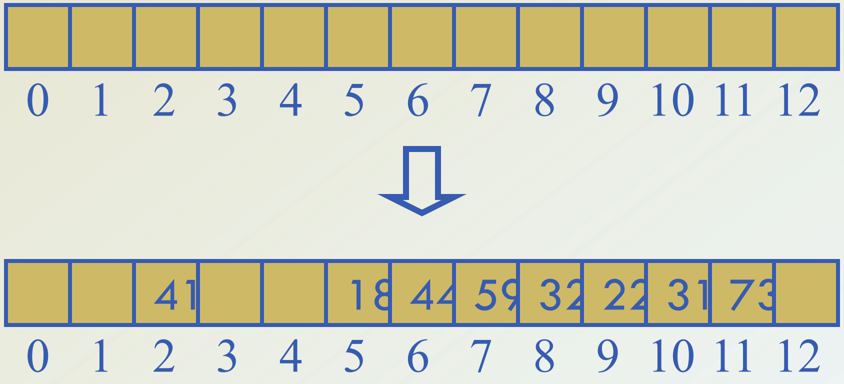
Algorithm put(k,v):
t = A[h(k)].put(k,v)
if t = **null then** {k is a new key}
    n = n + 1
return t

Algorithm remove(k):
t = A[h(k)].remove(k)
if t ≠ **null then** {k was found}
    n = n - 1
return t

Linear Probing

- Open addressing: the colliding item is placed in a different cell of the table
- Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



End of Chapter 9