

# Maps and Hash Tables



# Outline

- Maps
- Hashing
- Multimaps
- Ordered Maps

# Learning Outcomes

- By understanding this lecture, you should be able to:
- Outline the ADT for a map and a multimap
- Identify applications for which maps, multimaps and ordered maps are appropriate
- Design and implement a map in java and verify that it satisfies the requirements of the map ADT
- Explain the purpose of hash tables
- Design and implement hashing methods using separate chaining or open addressing and a variety of hashing and double-hashing functions
- Specify worst-case and average-case asymptotic run-times for standard operations on maps based upon hash tables
- Specify (worst-case) asymptotic run-times for standard operations on sorted maps

# Outline

- Maps
- Hashing
- Multimaps
- Ordered Maps

# Maps



- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
- Applications:
  - address book
  - student-record database

# The Map ADT (in `java.util`)

## ➤ Map ADT methods:

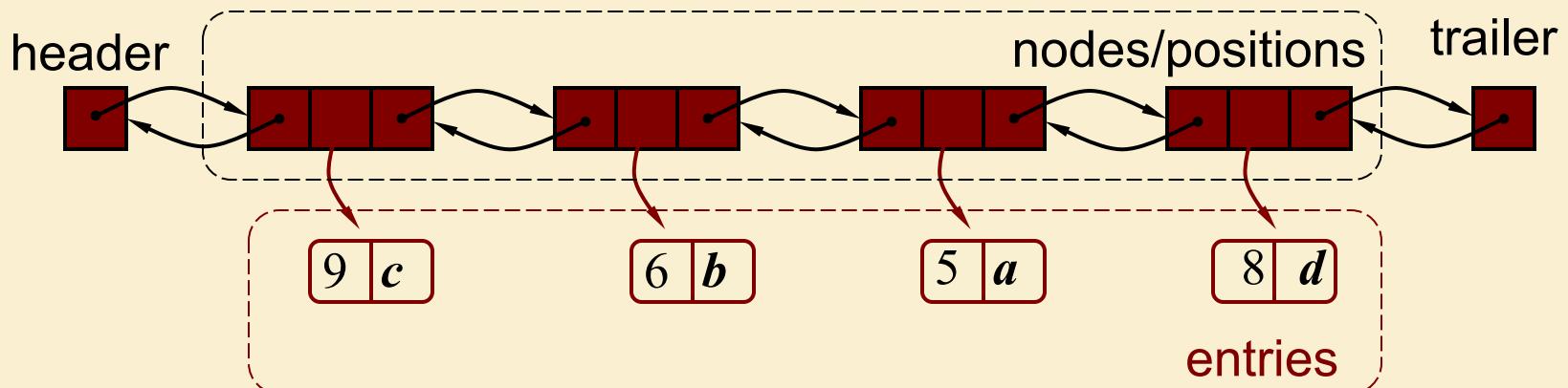
- ❑ `get(k)`: if the map  $M$  has an entry with key  $k$ , return its associated value; else, return null
- ❑ `put(k, v)`: insert entry  $(k, v)$  into the map  $M$ ; if key  $k$  is not already in  $M$ , then return null; else, return old value associated with  $k$
- ❑ `remove(k)`: if the map  $M$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

# Example

<i>Operation</i>	<i>Output</i>	<i>M</i>
isEmpty()	<b>true</b>	$\emptyset$
put(5, A)	<b>null</b>	(5, A)
put(7, B)	<b>null</b>	(5, A), (7, B)
put(2, C)	<b>null</b>	(5, A), (7, B), (2, C)
put(8, D)	<b>null</b>	(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)	<b>null</b>	(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)	<b>null</b>	(7, B), (8, D)
isEmpty()	<b>false</b>	(7, B), (8, D)

# A Simple List-Based Map

- We could implement a map using an unsorted list
  - We store the entries of the map in a doubly-linked list  $S$ , in arbitrary order



# The get( $k$ ) Algorithm

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

$B = S.positions()$  *{ $B$  is an iterator of the positions in  $S$ }*

**while**  $B.hasNext()$  **do**

$p = B.next()$  *{the next position in  $B$ }*

**if**  $p.element().getKey() = k$  **then**

**return**  $p.element().getValue()$

**return null** *{there is no entry with key equal to  $k$ }*

# The put( $k, v$ ) Algorithm

**Algorithm** put( $k, v$ ):

$B = S.positions()$

**while**  $B.hasNext()$  **do**

$p = B.next()$

**if**  $p.element().getKey() = k$  **then**

$t = p.element().getValue()$

$S.set(p, (k, v))$

**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  $k$ }

# The remove( $k$ ) Algorithm

**Algorithm** remove( $k$ ):

$B = S.positions()$

**while**  $B.hasNext()$  **do**

$p = B.next()$

**if**  $p.element().getKey() = k$  **then**

$t = p.element().getValue()$

$S.remove(p)$

$n = n - 1$  {decrement number of entries}

**return**  $t$  {return the removed value}

**return null** {there is no entry with key equal to  $k$ }

# Performance of a List-Based Map

- Performance:
  - ❑ **put**, **get** and **remove** take  $O(n)$  time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for small maps

# Outline

- Maps
- Hashing
- Multimaps
- Ordered Maps

# Hash Tables

- A hash table is a data structure that can be used to make map operations faster.
- While worst-case is still  $O(n)$ , average case is typically  $O(1)$ .

# Applications of Hash Tables

- Databases
- Caches
- Programming languages

# Hash Functions and Hash Tables

- A **hash function**  $h$  maps keys of a given type to integers in a fixed interval  $[0, N - 1]$

- Example:

$$h(k) = k \bmod N$$

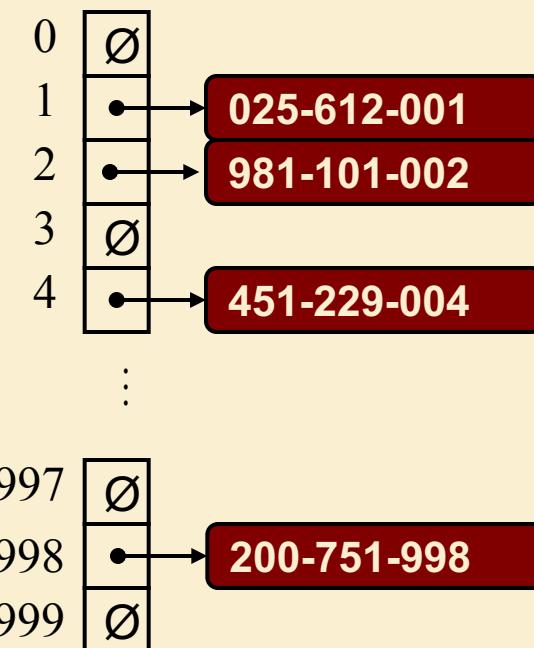
is a hash function for integer keys



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

# Example

- We design a hash table for a map storing entries as (SIN, Name), where SIN (social insurance number) is a nine-digit positive integer
- Our hash table uses an array of size  $N = 1,000$  and the hash function  
$$h(k) = \text{last three digits of SIN}$$
$$k$$
- **Problem:** what happens if two entries have the same last three digits?



# Hash Functions



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

**Hash code:**

$h_1$ : keys → integers

**Compression function:**

$h_2$ : integers ↳  $[0, N - 1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,  
$$h(k) = h_2(h_1(k))$$
- The goal of the hash function is to “disperse” the keys in an apparently random way

# Hash Codes



## ➤ Memory address:

- ❑ We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - ❑ Does not work well when copies of the same object may be stored at different locations.

## ➤ Integer cast:

- ❑ We reinterpret the bits of the key as an integer
  - ❑ Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)

## ➤ Component sum:

- ❑ We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - ❑ Suitable for keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

# Problems with Component Sum Hash Codes

- Hashing works when
  - the number of commonly-occurring keys is small relative to the hashing space (e.g.,  $2^{32}$  for a 32-bit hash code).
  - the hash codes for commonly-occurring keys are well-distributed (do not collide) in this space.
- Component Sum codes ignore the ordering of the components.
  - e.g., using 8-bit ASCII components, ‘stop’ and ‘pots’ yields the same code.
- If commonly-occurring keys are anagrams of each other, this is a bad idea!

# Polynomial Hash Codes

## ➤ Polynomial accumulation:

- We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

$$a_0 \ a_1 \ \dots \ a_{n-1}$$

- We evaluate the polynomial

$p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$  at a fixed value  $z$ , ignoring overflows

- Especially suitable for strings

- Polynomial  $p(z)$  can be evaluated in  $O(n)$  time using Horner's rule:

- ❖ The following polynomials are successively computed, each from the previous one in  $O(1)$  time

$$p_0(z) = a_{n-1}$$

$$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \dots, n-1)$$

- We have  $p(z) = p_{n-1}(z)$

# Compression Functions

## ➤ Division:

- $h_2(y) = y \bmod N$
- The size  $N$  of the hash table is usually chosen to be a prime (on the assumption that the differences between hash keys  $y$  are less likely to be multiples of primes).

## ➤ Multiply, Add and Divide (MAD):

- $h_2(y) = [(ay + b) \bmod p] \bmod N$ , where
  - ❖  $p$  is a prime number greater than  $N$
  - ❖  $a$  and  $b$  are integers chosen at random from the interval  $[0, p - 1]$ , with  $a > 0$ .

# Collisions

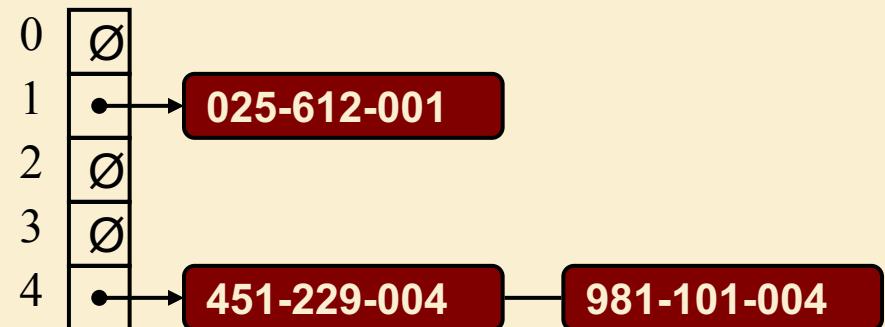


- Collisions occur when different elements are mapped to the same cell
- **Example:**
  - Suppose  $h_2(y) = y \bmod N$ , where  $N = 13$ .
  - $y = 17$  and  $y = 30$  will hash to the same location (4).

# 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 Methods with Separate Chaining

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

**Algorithm**  $\text{get}(k)$ :

**Output:** The value associated with the key  $k$  in the map, or **null** if there is no entry with key equal to  $k$  in the map

```
return  $A[h(k)].\text{get}(k)$       {delegate the get to the list-based map at  $A[h(k)]$ }
```

# Map Methods with Separate Chaining

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

**Algorithm**  $\text{put}(k, v)$ :

**Output:** Store the new (key, value) pair. If there is an existing entry with key equal to  $k$ , return the old value; otherwise, return **null**

```
 $t = A[h(k)].\text{put}(k, v)$            {delegate the put to the list-based map at  $A[h(k)]$ }

if  $t = \text{null}$  then            $\{k$  is a new key $\}$ 

     $n = n + 1$ 

return  $t$ 
```

# Map Methods with Separate Chaining

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

**Algorithm** `remove(k)`:

**Output:** The (removed) value associated with key  $k$  in the map, or **null** if there is no entry with key equal to  $k$  in the map

`t = A[h(k)].remove(k)` {delegate the remove to the list-based map at  $A[h(k)]$ }

**if**  $t \neq \text{null}$  **then** { $k$  was found}

$n = n - 1$

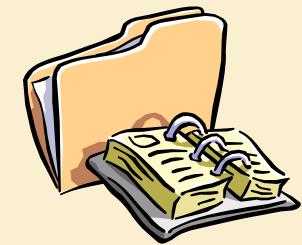
**return** *t*

# Open Addressing: 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, so that future collisions cause a longer sequence of probes
- Example:
- $h(k) = k \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	41		4	18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

# Get with Linear Probing



➤ Consider a hash table **A** of length **N** that uses linear probing

➤ **get(*k*)**

- We start at cell  $h(k)$
- We probe consecutive locations until one of the following occurs
  - ❖ An item with key  $k$  is found, or
  - ❖ An empty cell is found, or
  - ❖  $N$  cells have been unsuccessfully probed

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

```
i  $\leftarrow h(k)$ 
p  $\leftarrow 0$ 
repeat
  c  $\leftarrow A[i]$ 
  if c =  $\emptyset$ 
    return null
  else if c.key() = k
    return c.element()
  else
    i  $\leftarrow (i + 1) \bmod N$ 
    p  $\leftarrow p + 1$ 
until p = N
return null
```

# End of Lecture

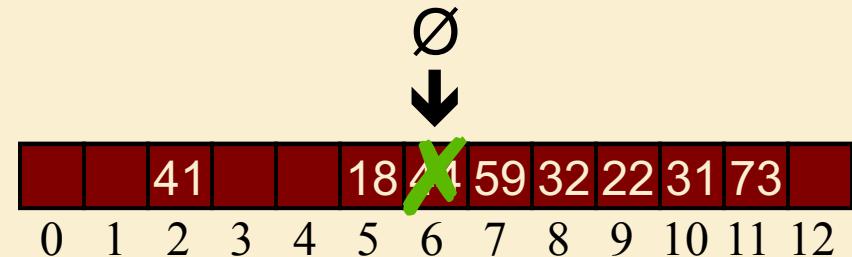
Feb 13, 2018

# Remove with Linear Probing

- Suppose we receive a **remove(44)** message.
- What problem arises if we simply remove the key = 44 entry?

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

$k$	$h(k)$	$i$
18	5	5
41	2	2
22	9	9
44	5	6
59	7	7
32	6	8
31	5	10
73	8	11



What happens now if we do a get(31)?

# Removal with Linear Probing

- To address this problem, we introduce a special object, called **AVAILABLE**, which replaces deleted elements
- **AVAILABLE** has a null key
- No changes to **get(k)** are required.

## Algorithm **get(k)**

```
i  $\leftarrow h(k)$ 
p  $\leftarrow 0$ 
repeat
  c  $\leftarrow A[i]$ 
  if c =  $\emptyset$ 
    return null
  else if c.key() = k
    return c.element()
  else
    i  $\leftarrow (i + 1) \bmod N$ 
    p  $\leftarrow p + 1$ 
  until p = N
return null
```

# Updates with Linear Probing

## ➤ **remove( $k$ )**

- We search for an entry with key  $k$
- If such an entry  $(k, v)$  is found, we replace it with the special item *AVAILABLE* and we return value  $v$
- Else, we return *null*

## ➤ **put( $k, v$ )**

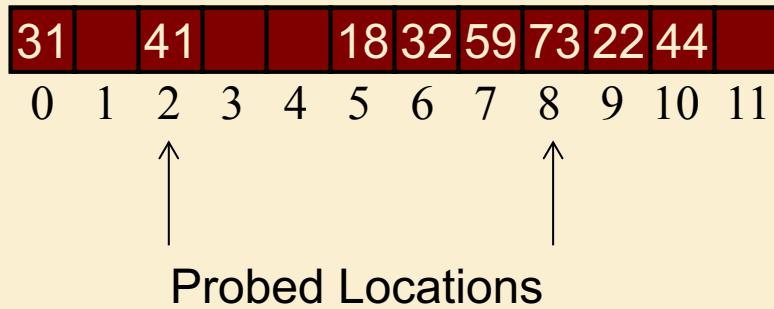
- We start at cell  $h(k)$  and then probe consecutive cells
- If we encounter a cell labeled AVAILABLE we note its index
- We continue probing until either
  - ❖ A cell  $i$  is found that is empty or has key matching  $k$
  - ❖  $N$  cells have been unsuccessfully probed
- If a cell has key matching  $k$ , we update the value for this cell to  $v$
- Else if a cell labeled AVAILABLE was encountered we store entry  $(k, v)$  there.
- Else if an empty cell was encountered we store entry  $(k, v)$  there
- Else if  $N$  cells were probed unsuccessfully we throw an exception.

# Open Addressing: Double Hashing

- Double hashing is an alternative open addressing method that uses a **secondary hash function  $h'(k)$**  in addition to the primary hash function  $h(x)$ .
- Suppose that the primary hashing  $i=h(k)$  leads to a collision.
- We then iteratively probe the locations
$$(i + jh'(k)) \bmod N \text{ for } j = 0, 1, \dots, N - 1$$
- The secondary hash function  $h'(k)$  cannot have zero values
- Choose  $N$  to be prime.
- Common choice of secondary hash function  $h'(k)$ :
  - $h'(k) = q - k \bmod q$ , where
    - ❖  $q < N$
    - ❖  $q$  is a prime
- The possible values for  $h'(k)$  are
$$1, 2, \dots, q$$

# Open Addressing: Double Hashing

- **Potential problem:** depending upon the relative values of  $N$  and  $h'(k)$ , not all locations may be probed.
- Example:  $N = 12$ ,  $h'(k) = 6$ . Suppose primary hash  $i = h(k) = 2$ .



- To avoid this problem, we need to ensure that none of the  $N$  probed locations repeat, i.e., we require:

$$(i + j h'(k)) \bmod N \neq (i + j' h'(k)) \bmod N, \text{ where } 0 \leq j < j' < N$$

# Open Addressing: Double Hashing

$$(i + jh'(k)) \bmod N \neq (i + j'h'(k)) \bmod N, \text{ where } 0 \leq j < j' < N$$

- In other words, we require

$$mh'(k) \neq nN, \forall m, n \in 1, 2, \dots, N-1, m \neq n$$

- This will be true if  $h'(k)$  and  $N$  are relatively prime, i.e., share no common factors.
- We have ensured this by making  $N$  prime. However this has downsides:
  - ❑ Modulo operations tend to be less efficient.
  - ❑ Have to search for prime numbers
- More common to make  $N$  a power of 2.
  - ❑ Makes modulo operations very efficient.
  - ❑ No searching for special numbers required.
- **Q:** If  $N$  is a power of 2, how can we ensure that  $h'(k)$  and  $N$  are relatively prime?
- **A:** Make  $h'(k)$  odd!

# Open Addressing: Double Hashing (Revised)

- Select  $N$  to be a power of 2.
- Let  $h'(k)$  be a random odd integer,  $0 < h'(k) < N$ , e.g.,
  - $h'(k) = 2 * \text{Random.nextInt}(N/2) + 1$ ,  
where  $\text{Rand.nextInt}(n)$  generates a pseudo-random integer in the range  $0 \dots n-1$ .

# Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $h'(k) = 7 - k \bmod 7$

- Insert keys 18, 41, 22, 44, 59, 32, 31, 73

$k$	$h(k)$	$h'(k)$	Probes
18	3	6	
41	1	5	
22	2	4	
44	4	3	
59	5	2	
32	6	1	
31	7	0	
73	0	6	

31	41		18	32	59	73	22	44			
0	1	2	3	4	5	6	7	8	9	10	11

# Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $h'(k) = 7 - k \bmod 7$

$k$	$h(k)$	$h'(k)$	Probes	
18	5	3	5	
41	2	1	2	
22	9	6	9	
44	5	5	5	10
59	7	4	7	
32	6	3	6	
31	5	4	5	9 0
73	8	4	8	

- Insert keys 18, 41, 22, 44, 59, 32, 31, 73



# Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take  $O(n)$  time
- The worst case occurs when all the keys inserted into the map collide
- The load factor  $\lambda = n/N$  affects the performance of a hash table
  - For separate chaining, performance is typically good for  $\lambda < 0.9$ .
  - For open addressing , performance is typically good for  $\lambda < 0.5$ .
  - `java.util.HashMap` maintains  $\lambda < 0.75$
- Open addressing can be more memory efficient than separate chaining, as we do not require a separate data structure.
- However, separate chaining is typically as fast or faster than open addressing.

# Rehashing

- When the load factor  $\lambda$  exceeds threshold, the table must be **rehashed**.
  - ❑ A larger table is allocated (typically at least double the size).
  - ❑ A new hash function is defined.
  - ❑ All existing entries are copied to this new table using the new hash function.

# Outline

- Maps
- Hashing
- **Multimaps**
- Ordered Maps

# Multimap ADT

- The Multimap ADT is identical to the Map ADT, but **allows entries with the same key**.
- Multimaps are also sometimes known as **dictionaries**.
- As for maps, the main operations are searching, inserting, and deleting items
- Applications:
  - ❑ word-definition pairs
- Dictionary ADT methods:
  - ❑ **get(k)**: if the dictionary has at least one entry with key k, returns one of them, else, returns null
  - ❑ **getAll(k)**: returns an iterable collection of all entries with key k
  - ❑ **put(k, v)**: inserts and returns the entry (k, v)
  - ❑ **remove(e)**: removes and returns the entry e. Throws an exception if the entry is not in the dictionary.
  - ❑ **entrySet()**: returns an iterable collection of the entries in the dictionary
  - ❑ **size(), isEmpty()**

# Multimaps and Java

- Note: The `java.util.Dictionary` class actually implements a map ADT.
- There is no multimap data structure in the Java Collections Framework that supports multiple entries with equal keys.
- The textbook (Ch. 10.5.3) sketches an implementation of a multimap based upon a map of keys, each entry of which supports a List of entries with the same key.
  - **Note:** There are some errors in the textbook java implementation of the multimap – please ignore Code Fragment 10.17.

# Example

<i>Operation</i>	<i>Output</i>	<i>Dictionary</i>
put(5,A)	(5,A)	(5,A)
put(7,B)	(7,B)	(5,A),(7,B)
put(2,C)	(2,C)	(5,A),(7,B),(2,C)
put(8,D)	(8,D)	(5,A),(7,B),(2,C),(8,D)
put(2,E)	(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(4)	<b>null</b>	(5,A),(7,B),(2,C),(8,D),(2,E)
get(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
getAll(2)	(2,C),(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
size()	5	(5,A),(7,B),(2,C),(8,D),(2,E)
remove(get(5))	(5,A)	(7,B),(2,C),(8,D),(2,E)
get(5)	<b>null</b>	(7,B),(2,C),(8,D),(2,E)

# Subtleties of remove(e)

- **remove(e)** will remove an entry that matches **e** (i.e., has the same (key, value) pair).
- If the dictionary contains more than one entry with identical (key, value) pairs, **remove(e)** will only remove one.
- Example:

<i>Operation</i>	<i>Output</i>	<i>Dictionary</i>
$e1 = \text{put}(2,C)$	$(2,C)$	$(5,A),(7,B),(2,C)$
$e2 = \text{put}(8,D)$	$(8,D)$	$(5,A),(7,B),(2,C),(8,D)$
$e3 = \text{put}(2,E)$	$(2,E)$	$(5,A),(7,B),(2,C),(8,D),(2,E)$
$\text{remove}(\text{get}(5))$	$(5,A)$	$(7,B),(2,C),(8,D),(2,E)$
$\text{remove}(e3)$	$(2,E)$	$(7,B),(2,C),(8,D)$
$\text{remove}(e1)$	$(2,C)$	$(7,B),(8,D)$

# A List-Based Multi-Map (Dictionary)

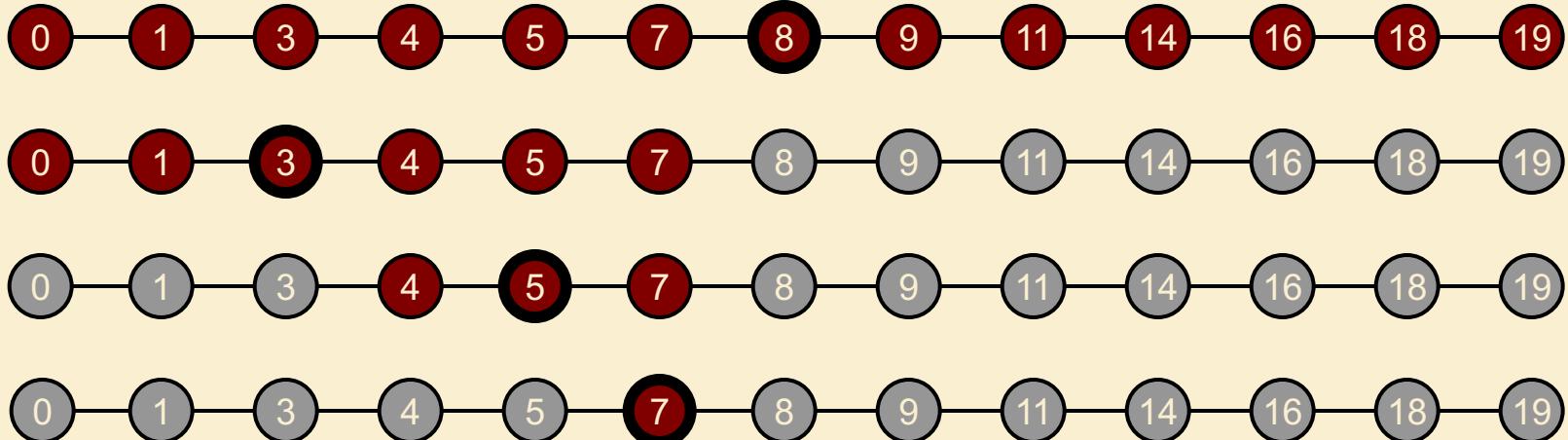
- A log file or audit trail is a dictionary implemented by means of an unsorted sequence
  - We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- Performance:
  - **insert** takes  $O(1)$  time since we can insert the new item at the beginning or at the end of the sequence
  - **find** and **remove** take  $O(n)$  time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

# Outline

- Maps
- Hashing
- Multimaps
- **Ordered Maps**

# Ordered Maps and Dictionaries

- If keys obey a total order relation, can represent a map or dictionary as an ordered search table stored in an array.
- Can then support a fast **find(k)** using **binary search**.
  - at each step, the number of candidate items is halved
  - terminates after a logarithmic number of steps
  - Example: **find(7)**



# Ordered Search Tables

- Performance:
  - ❑ **find** takes  $O(\log n)$  time, using binary search
  - ❑ **insert** takes  $O(n)$  time since in the worst case we have to shift  $n$  items to make room for the new item
  - ❑ **remove** takes  $O(n)$  time since in the worst case we have to shift  $n$  items to compact the items after the removal
- A search table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

# Outline

- Maps
- Hashing
- Multimaps
- Ordered Maps

# Learning Outcomes

**By understanding this lecture, you should be able to:**

- Outline the ADT for a map and a multimap
- Identify applications for which maps, multimaps and ordered maps are appropriate
- Design and implement a map in java and verify that it satisfies the requirements of the map ADT
- Explain the purpose of hash tables
- Design and implement hashing methods using separate chaining or open addressing and a variety of hashing and double-hashing functions
- Specify worst-case and average-case asymptotic run-times for standard operations on maps based upon hash tables
- Specify (worst-case) asymptotic run-times for standard operations on sorted maps