Collections

Part 2

Objectives

- Topics
  - Queue and Map interfaces and their concrete class implementations
  - Iterators and useful methods
- Goals: after this lecture, you will be able to
  - program applications using Queue and Map classes
  - create and use iterators
  - understand and use other built-in Queue and Map methods

Review

About how many nodes would be visited when searching for a particular item in an unsorted LinkedList with 1000 nodes?

a) 1

b) 10

c) 500

d) 1000

Review

About how many nodes would be visited when searching for a particular item in a sorted ArrayList with 1000 nodes using binary search?

a) 1

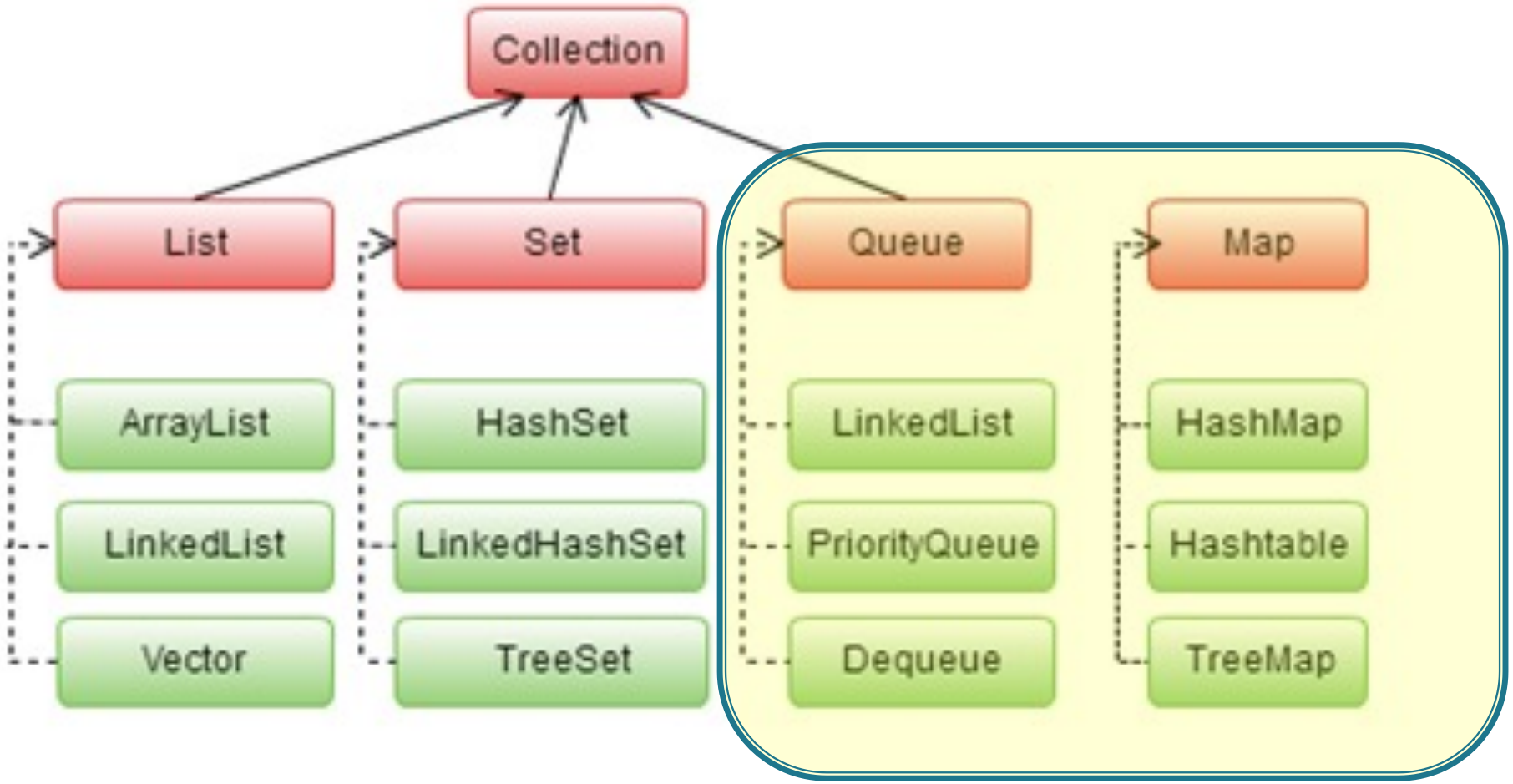
b) 10

c) 500

d) 1000

Queue and Map

- Recall that Queue< > is a child of Collection
  - ***Queue*** is used first-in, first-out collections: implemented by LinkedList, PriorityQueue, Deque
  - Queues are used for time-ordering or priority-ordering
- ***Map*** are used for fast searches



https://fresh2refresh.com/java-tutorial/java-collections-framework/

Queue

- The **Queue< >** interface is used for first-in, first-out list behavior: fair waiting
- The LinkedList class implements Queue in addition to List
  - so technically, you can use the List methods, but doing add( ) or remove( ) on a queue is un-queue-like

Queue Methods

- `size()` – number of elements on the queue
- `contains(element)` – returns true/false: need to override equals( )
- `offer()` – inserts at tail, returns false if it cannot
- `peek()` – returns element at head but doesn't remove it, returns `null` if empty
- `poll()` – removes and returns head element or `null` if empty

- `add()` – like offer(), but throws `IllegalArgumentException` if cannot insert (throws other exceptions, too)
- `element()` – like peek(), but throws `NoSuchElementException` if empty
- `remove()` – like poll(), but throws `NoSuchElementException` if empty

Queue Methods

```
Queue<Employee> myQueue = new LinkedList<>();
myQueue.offer( new Employee("Smith", 55, "Sales") );
myQueue.offer( new Employee("Jones", 42, "Accounting") );

Employee employee = myQueue.peek( );       // Smith
employee = myQueue.poll();                 // Also Smith
employee = myQueue.poll();                 // Jones
employee = myQueue.poll();                 // null
```

Queue Methods, cont.

```
Queue<Employee> myQueue = new LinkedList<>();
myQueue.offer( new Employee("Smith", 55, "Sales") );
myQueue.offer( new Employee("Jones", 42, "Accounting") );

Employee employee = null;
while (myQueue.peek( ) != null ){
      employee = myQueue.poll();
      System.out.println(employee.toString());
}
// Prints:
Name: Smith ID: 55 Department: Sales
Name: Jones ID: 42 Department: Accounting
```

Queue Methods, cont.

- But this still works – a LinkedList queue is also a List. This leaves the queue unchanged

```
Employee employee = null;
for (Employee e: myQueue) {
      System.out.println(e);
}
// Prints:
Name: Smith ID: 55 Department: Sales
Name: Jones ID: 42 Department: Accounting
```

Queue Methods, try-catch versions

```
Queue<Employee> myQueue = new LinkedList<>();
myQueue.add( new Employee("Smith", 55, "Sales") );
myQueue.add( new Employee("Jones", 42, "Accounting") );

Employee employee = null;
try {
      employee = myQueue.element( );          // Smith
} catch (NoSuchElementException e) {
      System.out.println("Queue error");
}
try {
      employee = myQueue.remove( );           // Smith
} catch (NoSuchElementException e) {
      System.out.println("Queue error");
}
```

Queue Methods, try-catch versions, cont.

```
Employee employee = null;
try {
  while (myQueue.element( ) != null ){
       employee = myQueue.remove();
       System.out.println(employee.toString());
  }
} catch (NoSuchElementException e) {
       System.out.println("Queue error");
}
```

PriorityQueue

- The **PriorityQueue< >** stores elements by priority as determined by the contained class' **`Comparable`** interface (so you need to implement that for your class) OR by providing a **`Comparator`** class when new-ing

```
public class Employee implements Comparable<Employee> {
        public int compareTo(Employee e) { return this.name.compareTo(e.name); }
        …
```

- String ordering probably doesn't make sense to order on, but …

PriorityQueue, cont.

```
Queue<Employee> myQueue = new PriorityQueue<>();
myQueue.offer( new Employee("Smith", 55, "Sales") );
myQueue.offer( new Employee("Jones", 42, "Accounting") );

Employee employee = null;
for (Employee e: myQueue) {
      System.out.println(e);
}
// Prints in sorted order this time:
Name: Jones ID: 42 Department: Accounting
Name: Smith ID: 55 Department: Sales
```

PriorityQueue, cont.

size required

Anonymous inner class

```
Queue<Employee> myQueue = new PriorityQueue<>(100, new Comparator<Employee>() {
        public int compare(Employee e1, Employee e2){
                return e1.getID() - e2.getID() });

myQueue.offer( new Employee("Smith", 55, "Sales") );
myQueue.offer( new Employee("Jones", 42, "Accounting") );

Employee employee = null;
for (Employee e: myQueue) {
        System.out.println(e);
}
// Prints in sorted order by ID:
Name: Jones ID: 42 Department: Accounting
Name: Smith ID: 55 Department: Sales
```

Map

- The `Map<Key, Value>` interface associates one thing (key) with another thing (value) for fast searches
- Two common implementations are `HashMap` and `TreeMap`
- `HashMap` returns values in hash order: it uses an object's inherited or overridden `hash( )` method to decide where to insert a pair
- `TreeMap` returns values in sorted order based on `Comparable` or `Comparator`

Hash Tables

- A hash table or hash map uses a mathematical function, the hash function, to decide where to store a key's value in the table
- Simple example: store just positive int keys in an array of length 10 using the hash function h(key) = key % 10

Insert 12: h(12) = 2, so store it at index 2

Insert 47: h(47) = 7

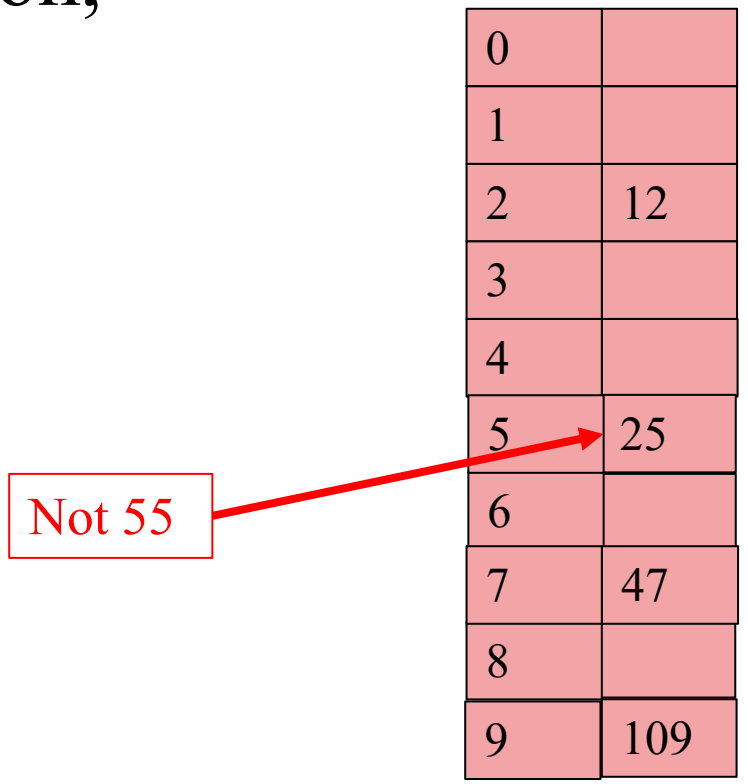
Insert 109: h(109) = 9

Insert 25: h(25) = 5

0	
1	
2	12
3	
4	
5	25
6	
7	47
8	
9	109

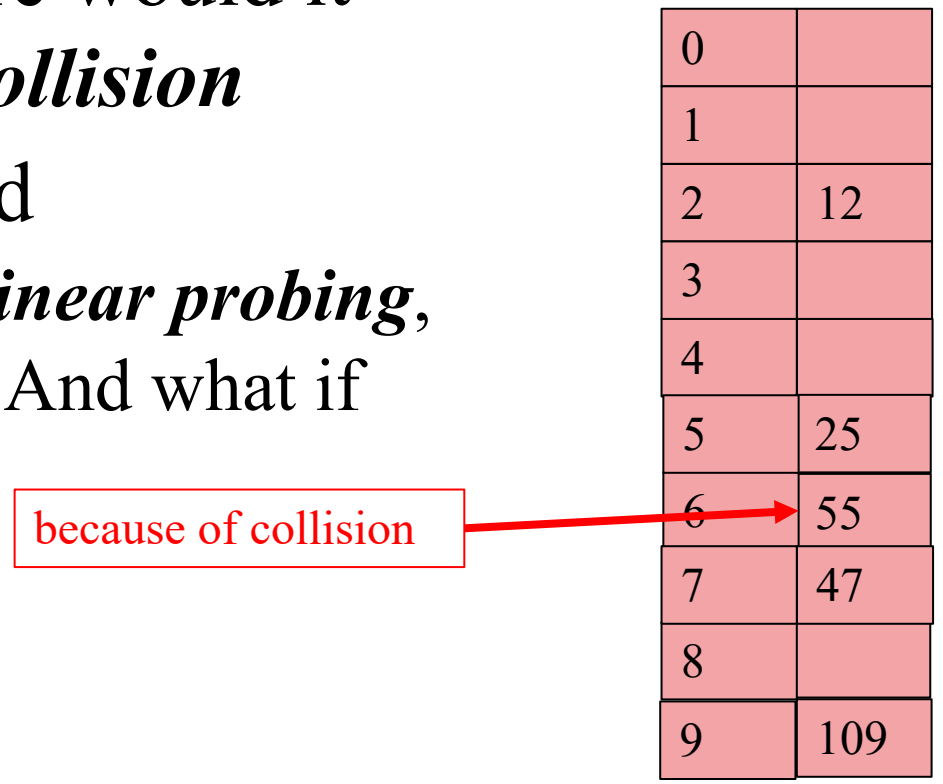
Hash Tables, cont.

- Search for a key is now a 1-step operation, plus a check.
  - Search(47): h(47) = 7, is list[7] == 47? Yes
  - Search(34): h(34) = 4, is list[4] == 34? No
  - Search(55): h(55) = 5, is list[5] == 55? No



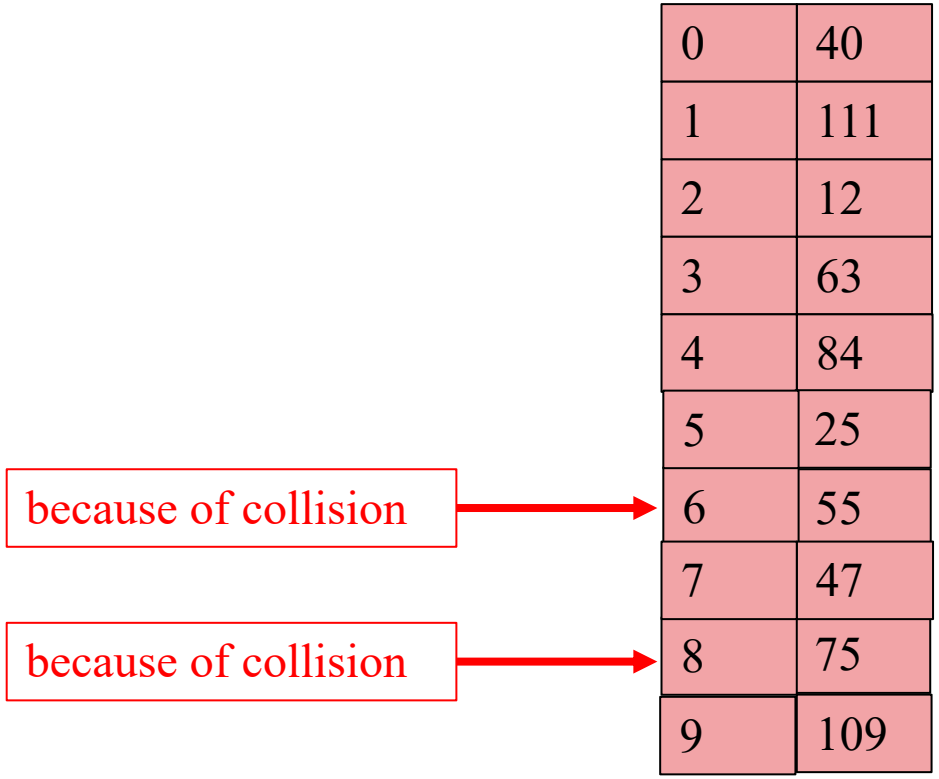
Hash Tables, cont.

- So if we wanted to insert 55, where would it go? Position 5 is already in use: ***collision***
- Need a collision resolution method
  - Put it in the next available position: ***linear probing***, #6, but what if we now search for it? And what if we want to insert(36)?



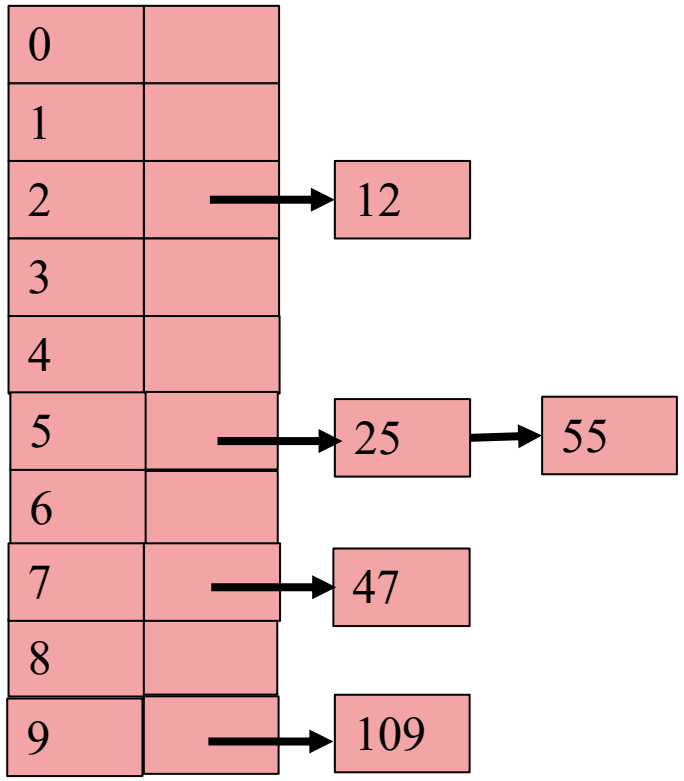
Hash Tables, cont.

- And what if the table fills up?
  - Expand and re-hash to the new table size: costly, because of the re-insertions
  - When table get more than 90% full, collision searching slows it down
  - Above 95%, approaches linear search



Hash Tables, cont.

- Alternative: each entry is the start of a linked list
- This doesn't solve the eventual linearity problem, but delays it:
  - Requires a linear search for every search request



Map methods

- Map has `put( ), get( ), containsKey( ), putIfAbsent( ),` and `remove( )`

String key, Employee value

```
Map<String, Employee> empmap = new HashMap<>();
Employee e1 = new Employee("Jones", 45, "Sales");
Employee e2 = new Employee("Ng", 27, "Marketing");
empmap.put(e1.getName(), e1);
empmap.put(e2.getName(), e2);
Employee result = empmap.get("Jones");
```

Map methods, cont.

- Map also has these, but try to resist them
  - If you're using these for lookups, you're mis-using the Map

  - `keySet( ) – Set of all keys`
  - `values( ) – Collection of all values`
  - `entrySet( ) – Set of all <key, value> pairs`

```
// Why? Sometimes you want to see all the keys
for (String s: empmap.keySet() ) {
    System.out.println(s);
}
```

hashcode() and equals()

- These are inherited from Object
- Override `hashcode( )` carefully: bad hashes cause too many collisions
- Here is `String.hashcode( )` simplified

```
public int hashcode() {
      int hash=0;
      for (int i = 0; i < length(); i++ ) {
            hash = 31 * hash + charAt(i);
      }
      return hash;
}
```

hashcode() and equals()

- `equals( )` is reasonably easy to override
  - If you override equals(), *always* override hashcode(): equal objects must have the same hashcode, and if you don't override hashcode, what's used?

```
public class Employee {
        . . .
        public boolean equals(Object o) {
            if (o == null) return false;
            if (o == this) return true;
            if ( !(o instanceof Employee) ) return false;
            Employee e = (Employee)o;
            return this.name.equals(e.name) && this.id == e.id
                        && this.dept.equals(e.dept);
        }
```