CSE 373

Java Collection Framework, Part 2: Priority Queue, Map

slides created by Marty Stepp
http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.
Priority queue ADT

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element
  - usually implemented using a tree structure called a *heap*

- priority queue operations:
  - **add**: adds in order; $O(\log N)$ worst
  - **peek**: returns *minimum* value; $O(1)$ always
  - **remove**: removes/returns *minimum* value; $O(\log N)$ worst
  - **isEmpty, clear, size, iterator**: $O(1)$ always
public class PriorityQueue<E> implements Queue<E>

<table>
<thead>
<tr>
<th>Method/Constructor</th>
<th>Description</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityQueue&lt;E&gt;()</td>
<td>constructs new empty queue</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(E value)</td>
<td>adds value in sorted order</td>
<td>O(log N )</td>
</tr>
<tr>
<td>clear()</td>
<td>removes all elements</td>
<td>O(1)</td>
</tr>
<tr>
<td>iterator()</td>
<td>returns iterator over elements</td>
<td>O(1)</td>
</tr>
<tr>
<td>peek()</td>
<td>returns minimum element</td>
<td>O(1)</td>
</tr>
<tr>
<td>remove()</td>
<td>removes/returns min element</td>
<td>O(log N )</td>
</tr>
</tbody>
</table>

Queue<String> pq = new PriorityQueue<String>();
pq.add("Stuart");
pq.add("Marty");
...
Priority queue ordering

• For a priority queue to work, elements must have an ordering
  ▪ in Java, this means implementing the `Comparable` interface
    • many existing types (Integer, String, etc.) already implement this
    • if you store objects of your own types in a PQ, you must implement it
  ▪ `TreeSet` and `TreeMap` also require Comparable types

```java
public class Foo implements Comparable<Foo> {
    ...
    public int compareTo(Foo other) {
        // Return > 0 if this object is > other
        // Return < 0 if this object is < other
        // Return    0 if this object == other
    }
}
```
The Map ADT

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"

- **basic map operations:**
  - **put**(key, value): Adds a mapping from a key to a value.
  - **get**(key): Retrieves the value mapped to the key.
  - **remove**(key): Removes the given key and its mapped value.

```
myMap.get("Juliet") returns "Capulet"
```
Map concepts

- A map can be thought of as a generalization of a tallying array.
  - The "index" (key) doesn't have to be an int.

- Count digits: 22092310907
  
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

// (R)epublican, (D)emocrat, (I)ndependent

- Count votes: "RDDDDRRRRRRDDDDDDDRDRRIRDRRIRDDLIRID"

<table>
<thead>
<tr>
<th>key</th>
<th>&quot;R&quot;</th>
<th>&quot;D&quot;</th>
<th>&quot;I&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>15</td>
<td>14</td>
<td>3</td>
</tr>
</tbody>
</table>

- Diagram:
  - Keys: "R", "D", "I"
  - Values: 15, 14, 3
Map implementation

- In Java, maps are represented by `Map` interface in `java.util`.

- `Map` is implemented by the `HashMap` and `TreeMap` classes:
  - `HashMap`: implemented using an array called a "hash table"; extremely fast: $O(1)$; keys are stored in unpredictable order.
  - `TreeMap`: implemented as a linked "binary tree" structure; very fast: $O(\log N)$; keys are stored in sorted order.

- A map requires 2 type parameters: one for keys, one for values.

```java
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```
## Map methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put(key, value)</code></td>
<td>Adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one.</td>
</tr>
<tr>
<td><code>get(key)</code></td>
<td>Returns the value mapped to the given key (<code>null</code> if not found).</td>
</tr>
<tr>
<td><code>containsKey(key)</code></td>
<td>Returns <code>true</code> if the map contains a mapping for the given key.</td>
</tr>
<tr>
<td><code>remove(key)</code></td>
<td>Removes any existing mapping for the given key.</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Removes all key/value pairs from the map.</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>Returns the number of key/value pairs in the map.</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>Returns <code>true</code> if the map's size is 0.</td>
</tr>
<tr>
<td><code>toString()</code></td>
<td>Returns a string such as <code>{a=90, d=60, c=70}</code>.</td>
</tr>
<tr>
<td><code>keySet()</code></td>
<td>Returns a set of all keys in the map.</td>
</tr>
<tr>
<td><code>values()</code></td>
<td>Returns a collection of all values in the map.</td>
</tr>
<tr>
<td><code>putAll(map)</code></td>
<td>Adds all key/value pairs from the given map to this map.</td>
</tr>
<tr>
<td><code>equals(map)</code></td>
<td>Returns <code>true</code> if given map has the same mappings as this one.</td>
</tr>
</tbody>
</table>
Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).

    ```java
    // key      value
    put("Joe", "206-685-2181")
    ```

- Later, we can supply only the key and get back the related value:
  Allows us to ask: *What is Joe's phone number?*

    ```java
    get("Joe")
    ```
Maps vs. sets

- A set is like a map from elements to boolean values.
  - **Set:** Is Joe found in the set? (true/false)
  - **Map:** What is Joe's phone number?

```
"Joe"  true/false
```

```
"Joe"  "206-685-2181"
```
keySet and values

- **keySet method** returns a `Set` of all keys in map
  - can loop over the keys in a foreach loop
  - can get each key's associated value by calling `get` on the map

```java
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Joe", 57);
ages.put("Geneva", 2);  // ages.keySet() returns Set<String>
ages.put("Vicki", 19);
for (String name : ages.keySet()) {
    int age = ages.get(name);
    System.out.println(name + " -> " + age);
}  
```

- **values method** returns a `Collection` of all values in map
  - `ages.values()` above returns `[2, 57, 19]`
  - can loop over the values with a for-each loop
  - no easy way to get from a value back to its associated key(s)
### Collections summary

<table>
<thead>
<tr>
<th>collection</th>
<th>ordering</th>
<th>benefits</th>
<th>weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>by index</td>
<td>fast; simple</td>
<td>little functionality; cannot resize</td>
</tr>
<tr>
<td>ArrayList</td>
<td>by insertion, by index</td>
<td>random access; fast to modify at end</td>
<td>slow to modify in middle/front</td>
</tr>
<tr>
<td>LinkedList</td>
<td>by insertion, by index</td>
<td>fast to modify at both ends</td>
<td>poor random access</td>
</tr>
<tr>
<td>TreeSet</td>
<td>sorted order</td>
<td>sorted; O(log N)</td>
<td>must be comparable</td>
</tr>
<tr>
<td>HashSet</td>
<td>unpredictable</td>
<td>very fast; O(1)</td>
<td>unordered</td>
</tr>
<tr>
<td>LinkedHashSet</td>
<td>order of insertion</td>
<td>very fast; O(1)</td>
<td>uses extra memory</td>
</tr>
<tr>
<td>TreeMap</td>
<td>sorted order</td>
<td>sorted; O(log N)</td>
<td>must be comparable</td>
</tr>
<tr>
<td>HashMap</td>
<td>unpredictable</td>
<td>very fast; O(1)</td>
<td>unordered</td>
</tr>
<tr>
<td>LinkedHashMap</td>
<td>order of insertion</td>
<td>very fast; O(1)</td>
<td>uses extra memory</td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>natural/comparable</td>
<td>fast ordered access</td>
<td>must be comparable</td>
</tr>
</tbody>
</table>

- It is important to be able to choose a collection properly based on the capabilities needed and constraints of the problem to solve.
Choosing a collection

Compound collections

- You will often find that you want a collection of collections:
  - a list of lists; a map of strings to lists; a queue of sets; ...

- Example: how would you store people's friends?
  - i.e., I need to quickly look up the names of all of Jimmy's buddies, or test whether a given person is a friend of Jimmy's or not.

```java
// don't forget to initialize each Set of friends
Map<String, Set<String>> pals =
    new HashMap<String, Set<String>>();
pals.put("Jimmy", new HashSet<String>());
pals.get("Jimmy").add("Bill");
pals.get("Jimmy").add("Katherine");
pals.get("Jimmy").add("Stuart");
```
Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of any collection.
  - Remembers a position, and lets you:
    - get the element at that position
    - advance to the next position
    - remove the element at that position
Iterator methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext()</td>
<td>returns true if there are more elements to examine</td>
</tr>
<tr>
<td>next()</td>
<td>returns the next element from the collection (throws a NoSuchElementException if there are none left to examine)</td>
</tr>
<tr>
<td>remove()</td>
<td>removes the last value returned by next() (throws an IllegalStateException if you haven't called next() yet)</td>
</tr>
</tbody>
</table>

- Iterator interface in java.util
  - every collection has an iterator() method that returns an iterator over its elements

```java
Set<String> set = new HashSet<String>();
...
Iterator<String> itr = set.iterator();
...```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38); // Jenny
scores.add(87);
scores.add(43); // Marty
scores.add(72);
...

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();
    System.out.println("The score is " + score);
    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
System.out.println(scores); // [72, 87, 94]
A surprising example

• What's bad about this code?

```java
List<Integer> list = new LinkedList<Integer>();
... (add lots of elements) ...
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

![Diagram showing linked list elements](image)
Iterators and linked lists

- Iterators are particularly useful with linked lists.
  - The previous code is $O(N^2)$ because each call on `get` must start from the beginning of the list and walk to index $i$.
  - Using an iterator, the same code is $O(N)$. The iterator remembers its position and doesn't start over each time.

![Diagram of linked list with iterator at index 1, current element 17]
ListIterator

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(value)</td>
<td>inserts an element just after the iterator's position</td>
</tr>
<tr>
<td>hasPrevious()</td>
<td>true if there are more elements before the iterator</td>
</tr>
<tr>
<td>nextIndex()</td>
<td>the index of the element that would be returned the next time next is called on the iterator</td>
</tr>
<tr>
<td>previousIndex()</td>
<td>the index of the element that would be returned the next time previous is called on the iterator</td>
</tr>
<tr>
<td>previous()</td>
<td>returns the element before the iterator (throws a NoSuchElementException if there are none)</td>
</tr>
<tr>
<td>set(value)</td>
<td>replaces the element last returned by next or previous with the given value</td>
</tr>
</tbody>
</table>

ListIterator<String> li = myList.ListIterator();

- lists have a more powerful ListIterator with more methods
  - can iterate forwards or backwards
  - can add/set element values (efficient for linked lists)