Queues, Linked Lists, and Recursion

CS187 Data Structures
March 3, 2010
Chris Vigorito
Queues

• First-In-First-Out (FIFO) data structure
• Commonly used in producer/consumer scenarios, buffers, etc.
• Can be standard or prioritized by some key
• Many ways to implement
Queue

- Interface that provides standard queue functionality
- Defines two sets of methods for insertion, removal, and examination
- One set throws exception if operation fails
- Other set returns a special value (null or false)
Queue

• **Insertion**: `add(e) / offer(e)` methods
  • Inserts element `e` (type defined through generics) at tail of queue if queue is not full

• **Deletion**: `remove() / poll()`
  • Pulls off and returns element at head of queue if queue is not empty

• **Examination**: `element() / peek()`
  • Returns element at head of queue if queue is not empty but leaves element in queue
<table>
<thead>
<tr>
<th></th>
<th>Throws exception</th>
<th>Returns special value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert</strong></td>
<td><code>add(e)</code></td>
<td><code>offer(e) [false]</code></td>
</tr>
<tr>
<td><strong>Remove</strong></td>
<td><code>remove()</code></td>
<td><code>poll() [null]</code></td>
</tr>
<tr>
<td><strong>Examine</strong></td>
<td><code>element()</code></td>
<td><code>peek() [null]</code></td>
</tr>
</tbody>
</table>
Implementing Classes

• Multiple classes depending on functionality you want
• For simplest, standard queue, *LinkedList* seems most straightforward
• *PriorityQueue* implements prioritized queue
  • Uses either natural ordering or a specified *Comparator* to prioritize
Iterating with Queues

• For standard queues (e.g., LinkedList), a ListIterator can be used to iterate through list in correct order

  • Retrieved by calling listIterator() method on list object

• For priority queues, standard Iterator can be used (retrieved via iterator() method)

  • Does not iterate through elements in any particular order
Example

```java
LinkedList<Integer> queue = new LinkedList<Integer>();
queue.add(4);
queue.offer(8);
queue.add(15);
System.out.println(queue.remove());  // 4
System.out.println(queue.poll());   // 8
System.out.println(queue.peek());   // 15
System.out.println(queue.element()); // 15
queue.add(16);
queue.add(23);
queue.add(42);
ListIterator<Integer> it = queue.listIterator();
while (it.hasNext())
    System.out.print(it.next() + " ");
System.out.println();
System.out.println(queue.remove());  // 15
```

Wednesday, March 10, 2010
LinkedList

• Linked list implementation of List interface
• Contains methods sufficient for functionality as queue, deque, or stack
• Your job to be consistent about list behavior
• Methods for insertion at front or back of list
• Performance is as would be for a doubly-linked list

Wednesday, March 10, 2010
**LinkedList**

- Important methods:
  - `addFirst()`/`addLast()` - add element to beginning/end of list
  - `removeFirst()`/`removeLast()` - remove element from beginning/end of list
  - methods from `Queue` and `Stack` interfaces
ListIterator

- Special type of iterator
- Allows traversal of list both forwards and backwards
- Allows you to add and remove safely while iterating
ListIterator

• Iterator keeps track of “cursor position”

• Always sits in between two elements (next and previous)

• `next() / previous()` - returns the next/previous element in the list

• `hasNext() / hasPrevious()` - returns boolean based on whether there are more elements in forward/backward direction
ListIterator

- **add()** inserts element before element that would be returned by **next()** and after element that would be returned by **previous()**
- Subsequent **next()** call is unaffected
- Subsequent **previous()** call would return new element
- **remove()** deletes last element returned by either **next()** or **previous()**
- Only one call per **next**/*previous* call allowed
Recursion

- Useful to think about the sequence of methods calls using a stack
- This is how recursive calls are actually kept track of in memory
- Can be less efficient but more simple/elegant