Many thanks to David Sun for some of the included slides!

**Important Dates**

- **Project 2 due**
  - DESIGN, DESIGN, DESIGN!!!
  - You may have 0 or 1 partner.
  - NO EXCEPTIONS!
  - Due Friday 7/24/2009 – 10pm
- **Midterm Review**
  - Saturday 7/26/2009 – 1-4pm in 306 Soda
- **Midterm 2**
  - Tuesday 7/28/2009 – 5-6pm in 10 Evans

**Queues (Review)**

```java
public interface Queue {
    public int size();
    public boolean isEmpty();
    public void enqueue(Object item);
    public Object dequeue() throws EmptyQueueException;
    public Object front() throws EmptyQueueException;
}
```

**What data structure would you use to store a queue?**

- Hash Table
- Binary Search Tree
- Linked List
- Array

**Using a Hash Table to implement a Queue (not a “good” idea)**

- Key is the # in line
- Value is the object in the queue
- Keep track of the current number

**Regular Queues with an Array**

- Insert 5

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 3 2 8 9 7</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 3 2 8 9 7 5</td>
<td></td>
</tr>
</tbody>
</table>
Regular Queues with an Array

• Remove First

Start
6 3 2 8 9 7 5
End

An Empty Queue

Start
End

A Queue Based upon Priority
PriorityQueue

public interface PriorityQueue {
  public int size();
  public boolean isEmpty();
  Entry insert(Object k, Object v);
  Entry max();
  Entry removeMax();
}

PriorityQueues Implemented with
Sorted Arrays

<table>
<thead>
<tr>
<th>Operation</th>
<th>Run Time</th>
<th>Other Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isEmpty()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert(...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>max()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeMax()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PriorityQueues Implemented with
Non-Sorted Arrays

<table>
<thead>
<tr>
<th>Operation</th>
<th>Run Time</th>
<th>Other Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isEmpty()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert(...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>max()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeMax()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Binary Heaps

• A Binary Heap is a binary tree, with two additional properties
  – Shape Property: It is a complete binary tree – a binary tree in which every row is full, except possibly the bottom row, which is filled from left to right.
  – Heap Property (or Heap Order Property): No child has a key greater than its parent’s key. This property is applied recursively: any subtree of a binary heap is also a binary heap.
• If we use the notion of smaller than in the Heap Property we get a min-heap. We’ll look at max-heap in this lecture.
Heaps Implement PriorityQueue

```java
public interface PriorityQueue {
    public int size();
    public boolean isEmpty();
    Entry insert(Object k, Object v);
    Entry max();
    Entry removeMax();
}
```

**max()**

- Trivial: The heap-order property ensures that the entry with the maximum key is always at the top of the heap. Hence, we simply return the entry at the root node.
  - If the heap is empty, return null or throw an exception.
- Runs in O(1) time.

**insert(8)**

Let \( x \) be the new entry \((k, v)\).

1. Place the new entry \( x \) in the bottom level of the tree, at the first free spot from the left. If the bottom level is full, start a new level with \( x \) at the far left.
2. If the new entry's key violates the heap-order property then compare \( x \)'s key with its parent's key; if \( x \)'s key is larger, we exchange \( x \) with its parent. Repeat the procedure with \( x \)'s new parent.

**removeMax()**

1. If the heap is empty, return null or throw an exception.
2. Otherwise, remove the entry at the root node. Replace the root with the last entry in the tree \( x \), so that the tree is still complete.
3. If the root violates the heap property then compare \( x \) with its children, swap \( x \) with the child with the larger key, repeat until \( x \) is greater than or equal to its children or reach a leaf.

---

**Representing Trees (Review)**

Array representations are common for Heaps

- **Left Child at** \( 2n + 1 \)
- **Right Child at** \( 2n + 2 \)

This is not a heap!
A Different Tree Implementation

(don't use the 0 index)

This is not a heap!

Left Child at 2n
Right Child at 2n + 1

Bottom-Up Heap Construction

• Suppose we are given a bunch of randomly ordered entries, and want to make a heap out of them.
• What’s the obvious way
  – Apply insert to each item in O(n log n) time.
• A better way: bottomUpHeap()
  1. Make a complete tree out of the entries, in any random order.
  2. Start from the last internal node (non-leaf node), in reverse order of the level order traversal, heapify down the heap as in removeMax().

Cost of Bottom Up Construction

• If each internal node bubbles all the way down, then the running time is proportional to the sum of the heights of all the nodes in the tree.
• Turns out this sum is less than n, where n is the number of entries being coalesced into a heap.
• Hence, the running time is in O(n), which is better than inserting n entries into a heap individually.

Running Times (Appendix)

• We could use a list or array, sorted or unsorted, to implement a priority queue. The following table shows running times for different implementations, with n entries in the queue.

<table>
<thead>
<tr>
<th></th>
<th>Binary Heap</th>
<th>Sorted List/Array</th>
<th>Unsorted List/Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>Theta(1)</td>
<td>Theta(1)</td>
<td>Theta(n)</td>
</tr>
<tr>
<td>insert (worst-case)</td>
<td>Theta(log n)*</td>
<td>Theta(n)</td>
<td>Theta(1)*</td>
</tr>
<tr>
<td>insert (best-case)</td>
<td>Theta(1)</td>
<td></td>
<td>It depends</td>
</tr>
<tr>
<td>removeMax (worst)</td>
<td>Theta(log n)</td>
<td>Theta(1)</td>
<td>Theta(1)*</td>
</tr>
<tr>
<td>removeMax (best)</td>
<td>Theta(1)</td>
<td>Theta(1)</td>
<td>Theta(n)</td>
</tr>
</tbody>
</table>

* If you are using an array-based data structure, these running times assume that you don't run out of room. If you do, it will take Omega(n) time to allocate a larger array and copy them into it.