CSE 2123:
Collections: Priority Queues

Jeremy Morris
Recall: A *queue* is a specific type of *collection*

- Keeps elements in a particular order
- We’ve seen two examples
  - FIFO queues
  - Stacks (LIFO queues)

What if we want to keep things in a particular order that *isn’t* based on when they entered the queue?

- Keep elements in sorted order using a *priority*
A priority queue uses queue operations like a FIFO queue or a stack

- But when remove is called, the element with the “most urgent” priority is returned
  - Instead of the element put first into the queue or last into the queue

- For the Java implementation of a priority queue, lower values have a “more urgent” priority
  - Something with a priority of 1 will be removed before something with a priority of 10
  - This of this as a “ranked list” of items – the #1 football team has a higher priority than the #10 team
Collections – Priority Queue

- Priority queue processing relies on a few standard methods (that should be familiar):
  - Add an object to the queue:
    ```java
    void add(E obj)
    ```
  - Retrieve (and remove) the item with the lowest score from the queue
    ```java
    E remove()
    ```
  - Examine the next item in the queue without removing it:
    ```java
    E peek()
    ```
  - Test to see if the queue is empty:
    ```java
    boolean isEmpty()
    ```
Java uses the *PriorityQueue* class to implement a priority queue

- A priority queue of Integers:
  ```java
  PriorityQueue<Integer> intQueue =
  new PriorityQueue<Integer>();
  ```

- A priority queue of Strings:
  ```java
  PriorityQueue<String> strQueue =
  new PriorityQueue<String>();
  ```
What do we use for “rankings” (priorities)?

- Numeric types (Integer, Double) are in the obvious order (ascending by value)
- For boolean values, false comes before true
- Characters are ordered by their numeric (Unicode) value
- Strings are ordered lexicographically
  - Similar to how words are ordered in the dictionary
  - See pages 92-93 in the text or the Java API documentation for the compareTo method of String for details
What do we use for “rankings” (priorities) for other types?

- Java’s priority queue implementation requires that we implement the Comparable interface for anything we want to put into it
  - We must write a compareTo method for any classes we want to use with a PriorityQueue
  - As with TreeSets and TreeHash
public static void main(String[] args) {
    // TODO Auto-generated method stub
    PriorityQueue<Integer> myQueue =
        new PriorityQueue<Integer>();
    myQueue.add(10);
    myQueue.add(5);
    myQueue.add(22);
    System.out.println("QUEUE: "+myQueue);

    while (!myQueue.isEmpty()) {
        int head = myQueue.remove();
        System.out.print("HEAD: "+head+" ");
        System.out.println("QUEUE: "+myQueue);
    }
}
Collections – Priority Queue (Heaps)

- Priority queues are often implemented using a data structure known as a **heap**
  - This is how the PriorityQueue class is implemented in Java

- A **heap** is another form of binary tree
  - It is NOT a binary search tree
  - Instead we want to keep our element with the “most urgent” priority (smallest value) as the root node of the heap
Heaps must satisfy a **heap ordering property**
- Heaps can either be *min heaps* or *max heaps*, determined by their ordering property
- For Java’s `PriorityQueue`, the *min heap* property is used (and that will be the one we discuss here)

The *min heap* property states that the value of each node is always *greater than or equal to* the value of its parent
- That ensures that the node with the minimum value will be at the root
Suppose we have the elements {1, 3, 7, 10, 19, 22} stored in a heap

- The heap constructed might look like the tree to the right
- The heap creates a complete binary tree – filling in nodes across each level
Collections – Priority Queue (Heap)

- Suppose we have the elements {1, 3, 7, 10, 19, 22} stored in a heap
  - The heap constructed might look like the tree to the right
  - The heap creates a complete binary tree – filling in nodes across each level
- Now suppose we remove our first element (our “most urgent” value in the min node)
Now suppose we remove our first element (our “most urgent” value in the min node)

- We need a new root node
- How can we preserve our min heap property?

When the root node is removed, we automatically “swap in” the “bottom” node of the priority queue
Collections – Priority Queue (Heap)

- When the root node is removed, we automatically “swap in” the “bottom” node of the priority queue
  - The min heap property is violated, so our “remove” is not finished yet
- Next we need fix our tree so that the heap property is maintained
  - The root is swapped with its child that has the smallest value
Collections – Priority Queue (Heap)

- The root is swapped with its child that has the smallest value
- We’re still not done – our heap property is still violated
  - So our node is again swapped with its child that has the smallest value
Collections – Priority Queue (Heap)

- We’re still not done – our heap property is still violated
  - So our node is again swapped with its child that has the smallest value
- At this point we’re finished
  - We have our value taken off the priority queue
  - The min heap property is no longer violated
  - The root node is our minimum value and is in position to be removed next
What about if we need to add something to our heap?

- This operates like remove in reverse
- Instead of moving a bad root node down the heap, we’ll move a good node up the heap
- We stop when we reach the spot in the heap where our node “should” be

Let’s add the value of 2 to our heap
Let’s add the value of 2 to our heap
- The 2 is added to the bottom of the heap
- We’re in violation of our min heap property
- So we swap the child with its parent
Collections – Priority Queue (Heap)

- We’re in violation of our min heap property
- So we swap the child with its parent

We’re still in violation of our min heap property, so we keep going
- We swap our inserted node with its new parent
Collections – Priority Queue (Heap)

- We swap our inserted node with its new parent
- And now we’re done
  - Our min heap property is satisfied
  - No more swaps are needed

```
2
/  \
/    \
10   3
/ \
/   \
22 19 7
```
Collections – Priority Queue (Heap)

- Notice that our min heap property is maintained just by applying a few simple rules
  - **Always insert at the bottom of the heap**
    - When we insert, check to see if the parent of our new node is larger
    - If it is, swap them and repeat until the parent is smaller or our new node is the root node
  - **Remove nodes from the top of the heap**
    - When we remove, move the bottom node up to the root
    - Check to see if our new root node is larger than its children
      - If it is, swap it with the smallest of its children and repeat until all children are larger or we hit the bottom of the heap
Using these rules we can build a heap

- Suppose we have the elements {7, 10, 3, 22, 1, 19} to store in a heap
- We’ll add them in that order
- First we add the 7
  - Nothing in the heap yet
  - It has no parents
  - The heap property is satisfied
Using these rules we can build a heap

- Suppose we have the elements {7, 10, 3, 22, 1, 19} to store in a heap
- We’ll add them in that order
- First we add the 7
- Next the 10
  - Add it to the bottom
  - Compare to its parent
  - The heap property is satisfied
Collections – Priority Queue (Heap)

- {7, 10, 3, 22, 1, 19}
  - Next the 3
    - Add it to the bottom
    - Compare to its parent
    - The heap property is NOT satisfied
    - Swap it with its parent
Collections – Priority Queue (Heap)

- \{7, 10, 3, 22, 1, 19\}
  - Next the 3
    - Add it to the bottom
    - Compare to its parent
    - The heap property is NOT satisfied
    - Swap it with its parent
    - Now the heap property is satisfied
Collections – Priority Queue (Heap)

- {7, 10, 3, 22, 1, 19}
  - Next the 22
    - Add it to the bottom
    - Compare to its parent
    - The heap property is satisfied
Collections – Priority Queue (Heap)

- {7, 10, 3, 22, 1, 19}
  - Next the 1
    - Add it to the bottom
    - Compare to its parent
    - The heap property is NOT satisfied
    - Swap it with its parent
Collections – Priority Queue (Heap)

- {7, 10, 3, 22, 1, 19}
  - Next the 1
    - Add it to the bottom
    - Compare to its parent
    - The heap property is NOT satisfied
    - Swap it with its parent
    - The heap property is STILL NOT satisfied
    - Swap it with its new parent
Collections – Priority Queue (Heap)

- \{7, 10, 3, 22, 1, 19\}
  - Next the 1
    - Add it to the bottom
    - Compare to its parent
    - The heap property is NOT satisfied
    - Swap it with its parent
    - The heap property is STILL NOT satisfied
    - Swap it with its new parent
    - Now the heap property is satisfied
Collections – Priority Queue (Heap)

- \{7, 10, 3, 22, 1, 19\}
  - Next the 19
    - Add it to the bottom
    - Compare to its parent
    - The heap property is satisfied
Your Turn - Heaps

- Provide the heap that would result if the following integers were added to the tree in the given order:
  
  \[ 8, 3, 23, 15, 1, 5, 42, 64, 36 \]

- Provide heap that would result if the following characters were added in the given order:
  
  ['r', 'q', 'z', 'a', 'c', 'l', 'x', 'w', 'b']
So when is it good to use a PriorityQueue in Java?

- With the heap implementation, finding the smallest value is constant time
  - It’s always at the top of the heap
- Removing the minimum element or adding a new element is at worst log n time
  - We may need to make up to log n swaps to maintain our heap property
- Finding an arbitrary element in the heap is at worst linear time
  - Worst case we have to examine every node!
Collections – Priority Queue

- So when is it good to use a PriorityQueue in Java?
  - When we restrict ourselves to queue operations
  - Working only in *priority order*
    - Don’t need to access elements in the “middle” of the pack
public static void main(String[] args) {
    // TODO Auto-generated method stub
    PriorityQueue<String> myQueue =
        new PriorityQueue<String>();

    myQueue.add("Marley, Bob");
    myQueue.add("Cool, Joe");
    myQueue.add("Brown, Charles");
    System.out.println("QUEUE: "+myQueue);

    while (!myQueue.isEmpty()) {
        String head = myQueue.remove();
        System.out.print("HEAD: "+head+" ");
        System.out.println("QUEUE: "+myQueue);
    }
}