CSE373: Data Structures & Algorithms
Lecture 6: Priority Queues

Dan Grossman
Fall 2013
A new ADT: Priority Queue

• Textbook Chapter 6
  – Nice to see a new and surprising data structure

• A priority queue holds compare-able data
  – Like dictionaries and unlike stacks and queues, need to compare items
    • Given $x$ and $y$, is $x$ less than, equal to, or greater than $y$
    • Meaning of the ordering can depend on your data
    • Many data structures require this: dictionaries, sorting
  – Integers are comparable, so will use them in examples
    • But the priority queue ADT is much more general
    • Typically two fields, the priority and the data
Priorities

- Each item has a “priority”
  - The lesser item is the one with the greater priority
  - So “priority 1” is more important than “priority 4”
  - (Just a convention, think “first is best”)

- Operations:
  - `insert`
  - `deleteMin`
  - `is_empty`

- Key property: `deleteMin` returns and deletes the item with greatest priority (lowest priority value)
  - Can resolve ties arbitrarily
Example

\begin{align*}
\text{insert } x_1 \text{ with priority } 5 \\
\text{insert } x_2 \text{ with priority } 3 \\
\text{insert } x_3 \text{ with priority } 4 \\
&= \text{deleteMin } // x_2 \\
&= \text{deleteMin } // x_3 \\
\text{insert } x_4 \text{ with priority } 2 \\
\text{insert } x_5 \text{ with priority } 6 \\
&= \text{deleteMin } // x_4 \\
&= \text{deleteMin } // x_1
\end{align*}

- Analogy: \text{insert} is like \text{enqueue}, \text{deleteMin} is like \text{dequeue} \\
  - But the whole point is to use priorities instead of FIFO
Applications

Like all good ADTs, the priority queue arises often
  – Sometimes blatant, sometimes less obvious

• Run multiple programs in the operating system
  – “critical” before “interactive” before “compute-intensive”
  – Maybe let users set priority level

• Treat hospital patients in order of severity (or triage)
• Select print jobs in order of decreasing length?
• Forward network packets in order of urgency
• Select most frequent symbols for data compression (cf. CSE143)
• Sort (first \texttt{insert} all, then repeatedly \texttt{deleteMin})
  – Much like Homework 1 uses a stack to implement reverse
More applications

• “Greedy” algorithms
  – May see an example when we study graphs in a few weeks

• Discrete event simulation (system simulation, virtual worlds, …)
  – Each event $e$ happens at some time $t$, updating system state and generating new events $e_1, \ldots, e_n$ at times $t+t_1, \ldots, t+tn$
  – Naïve approach: advance “clock” by 1 unit at a time and process any events that happen then
  – Better:
    • $Pending$ events in a priority queue (priority = event time)
    • Repeatedly: deleteMin and then insert new events
    • Effectively “set clock ahead to next event”
Finding a good data structure

- Will show an efficient, non-obvious data structure
  - But first let’s analyze some “obvious” ideas for $n$ data items
  - All times worst-case; assume arrays “have room”

<table>
<thead>
<tr>
<th>data</th>
<th>insert algorithm / time</th>
<th>deleteMin algorithm / time</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted array</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsorted linked list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sorted circular array</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sorted linked list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVL tree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Need a good data structure!

- Will show an efficient, non-obvious data structure for this ADT
  - But first let’s analyze some “obvious” ideas for $n$ data items
  - All times worst-case; assume arrays “have room”

<table>
<thead>
<tr>
<th>data</th>
<th>insert algorithm / time</th>
<th>deleteMin algorithm / time</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted array</td>
<td>add at end</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>unsorted linked list</td>
<td>add at front</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>search</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>sorted circular array</td>
<td>search / shift</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>move front</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>sorted linked list</td>
<td>put in right place</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>remove at front</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>binary search tree</td>
<td>put in right place</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>leftmost</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>AVL tree</td>
<td>put in right place</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td></td>
<td>leftmost</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
More on possibilities

- If priorities are random, binary search tree will likely do better
  - $O(\log n)$ insert and $O(\log n)$ deleteMin on average

- One more idea: if priorities are 0, 1, ..., $k$ can use array of lists
  - insert: add to front of list at $arr[priority]$, $O(1)$
  - deleteMin: remove from lowest non-empty list $O(k)$

- We are about to see a data structure called a “binary heap”
  - $O(\log n)$ insert and $O(\log n)$ deleteMin worst-case
    - Possible because we don’t support unneeded operations; no need to maintain a full sort
      - Very good constant factors
    - If items arrive in random order, then insert is $O(1)$ on average
Our data structure

A binary min-heap (or just binary heap or just heap) is:

- **Structure property**: A *complete* binary tree
- **Heap property**: The priority of every (non-root) node is greater than the priority of its parent
  - *Not* a binary search tree

So:

- Where is the highest-priority item?
- What is the height of a heap with \( n \) items?
**Operations: basic idea**

- **findMin**: return root.data
- **deleteMin**:
  1. answer = root.data
  2. Move right-most node in last row to root to restore structure property
  3. “Percolate down” to restore heap property
- **insert**:
  1. Put new node in next position on bottom row to restore structure property
  2. “Percolate up” to restore heap property

**Overall strategy:**
- Preserve structure property
- Break and restore heap property
DeleteMin

1. Delete (and later return) value at root node
2. Restore the Structure Property

• We now have a “hole” at the root
  – Need to fill the hole with another value

• When we are done, the tree will have one less node and must still be complete
3. Restore the Heap Property

Percolate down:
- Keep comparing with both children
- Swap with lesser child and go down one level
- Done if both children are \( \geq \) item or reached a leaf node

Why is this correct? What is the run time?
DeleteMin: Run Time Analysis

• Run time is $O(\text{height of heap})$

• A heap is a complete binary tree

• Height of a complete binary tree of $n$ nodes?
  – height $= \left\lfloor \log_2(n) \right\rfloor$

• Run time of deleteMin is $O(\log n)$
Insert

- Add a value to the tree
- Afterwards, structure and heap properties must still be correct
Insert: Maintain the Structure Property

- There is only one valid tree shape after we add one more node
- So put our new data there and then focus on restoring the heap property
Maintain the heap property

Percolate up:
• Put new data in new location
• If parent larger, swap with parent, and continue
• Done if parent ≤ item or reached root

Why is this correct? What is the run time?
Insert: Run Time Analysis

- Like `deleteMin`, worst-case time proportional to tree height
  - $O(\log n)$

- But... `deleteMin` needs the “last used” complete-tree position and `insert` needs the “next to use” complete-tree position
  - If “keep a reference to there” then `insert` and `deleteMin` have to adjust that reference: $O(\log n)$ in worst case
  - Could calculate how to find it in $O(\log n)$ from the root given the size of the heap
    - But it’s not easy
    - And then `insert` is always $O(\log n)$, promised $O(1)$ on average (assuming random arrival of items)

- There’s a “trick”: don’t represent complete trees with explicit edges!