CSE 332: Data Structures & Parallelism

Lecture 3: Priority Queues

Ruth Anderson
Autumn 2016
Today

- Finish up Asymptotic Analysis
- New ADT! Priority Queues
Scenario

What is the difference between waiting for service at a pharmacy versus an ER?

Pharmacies usually follow the rule First Come, First Served

Emergency Rooms assign priorities based on each individual's need
A new ADT: Priority Queue

• Textbook Chapter 6
  – We will go back to binary search trees (ch4) and hash tables (ch5) later
  – Nice to see a new and surprising data structure first
• A priority queue holds compare-able data
  – Unlike stacks and queues need to compare items
    • Given $x$ and $y$, is $x$ less than, equal to, or greater than $y$
    • What this means can depend on your data
    • Much of course will require comparable data: e.g. 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
Priority Queue ADT

- Assume 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, could also do a maximum priority

- Main Operations:
  - insert
  - deleteMin

- Key property: deleteMin returns and deletes from the queue the item with greatest priority (lowest priority value)
  - Can resolve ties arbitrarily
Aside: We will use ints as data and priority

For simplicity in lecture, we’ll often suppose items are just ints and the int is also the priority

• So an operation sequence could be
  
  ```
  insert 6
  insert 5
  x = deleteMin  // Now x = 5.
  ```

– int priorities are common, but really just need comparable

• Not having “other data” is very rare
  
  – Example: print job has a priority and the file to print is the data
To simplify our examples, we will just use the priority values from now on.

**Priority Queue Example**

*After execution:*

- Insert *a* with priority 5
- Insert *b* with priority 3
- Insert *c* with priority 4
- $w = \text{deleteMin}$
- $x = \text{deleteMin}$
- Insert *d* with priority 2
- Insert *e* with priority 6
- $y = \text{deleteMin}$
- $z = \text{deleteMin}$

Analogy: **insert** is like **enqueue**, **deleteMin** is like **dequeue**. But the whole point is to use priorities instead of FIFO.
Applications

Like all good ADTs, the priority queue arises often
   – Sometimes “directly”, 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: **insert** all, then repeatedly **deleteMin**
More applications

• “Greedy” algorithms
  – Select the ‘best-looking’ choice at the moment
  – Will see an example when we study graphs in a few weeks
• Discrete event simulation (system modeling, virtual worlds, …)
  – Simulate how state changes when events fire
  – Each event $e$ happens at some time $t$ and generates 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 = time happens)
    • Repeatedly: $deleteMin$ and then $insert$ new events
    • Effectively, “set clock ahead to next event”
# Preliminary Implementations of Priority Queue ADT

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>deleteMin</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>
</tbody>
</table>

**Notes:** Worst case, Assume arrays have enough space

10/03/2016
Aside: More on possibilities

- Note: If priorities are inserted in random order, binary search tree will likely do better than $O(n)$
  - $O(\log n)$ insert and $O(\log n)$ deleteMin on average
  - Could get same performance from a balanced binary search tree (e.g. AVL tree we will study later)

- 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 $O(k)$
Our Data Structure: The Heap

The Heap:
• Worst case: $O(\log n)$ for insert
• Worst case: $O(\log n)$ for deleteMin
• If items arrive in random order, then the average-case of insert is $O(1)$
• Very good constant factors

Key idea: Only pay for functionality needed
• We need something better than scanning unsorted items
• But we do not need to maintain a full sorted list

• We will visualize our heap as a tree, so we need to review some tree terminology
Q: Reviewing Some Tree Terminology

root(T):

leaves(T):

children(B):

parent(H):

siblings(E):

ancestors(F):

descendents(G):

subtree(G):
Q: Some More Tree Terminology

**depth** (B):

**height** (G):

**height** (T):

**degree** (B):

**branching factor** (T):
Types of Trees

Binary tree: Every node has \( \leq 2 \) children

n-ary tree: Every node has \( \leq n \) children

Perfect tree: Every row is completely full

Complete tree: All rows except possibly the bottom are completely full, and it is filled from left to right
Some Basic Tree Properties

Nodes in a perfect binary tree of height $h$?

Leaf nodes in a perfect binary tree of height $h$?

Height of a perfect binary tree with $n$ nodes?

Height of a complete binary tree with $n$ nodes?
Properties of a Binary Min-Heap

More commonly known as a binary heap or simply a heap
  – **Structure Property:**
    A complete [binary] tree
  – **Heap Property:**
    The priority of every non-root node is greater than (or possibly equal to) the priority of its parent

How is this different from a binary search tree?
Properties of a Binary Min-Heap

More commonly known as a binary heap or simply a heap

- **Structure Property:**
  A complete [binary] tree

- **Heap Order Property:**
  The priority of every non-root node is greater than the priority of its parent

![A Heap](image1)

![Not a Heap](image2)
Properties of a Binary Min-Heap

- Where is the minimum priority item?

- What is the height of a heap with n items?
Heap Operations

- findMin:
- deleteMin: percolate down.
- insert(val): percolate up.
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 order property

- **insert:**
  1. Put new node in next position on bottom row to restore structure property
  2. “Percolate up” to restore heap order property

Overall strategy:

- Preserve complete tree structure property
- This may break heap order property
- Percolate to restore heap order property
DeleteMin Implementation

1. Delete value at root node (and store it for later return)

2. There is now a "hole" at the root. We must "fill" the hole with another value, must have a tree with one less node, and it must still be a complete tree.

3. The "last" node is the is obvious choice, but now the heap order property is violated.

4. We percolate down to fix the heap order:
   While greater than either child
   Swap with smaller child
Percolate Down

Percolate down:
- Keep comparing with both children
- Move smaller child up and go down one level
- Done if both children are $\geq$ item or reached a leaf node
- Why does this work? What is the run time?
DeleteMin: Run Time Analysis

- Run time is $O$ (height of heap)

- A heap is a complete binary tree

- Height of a complete binary tree of $n$ nodes?
  - height $= \lfloor \log_2(n) \rfloor$

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

- Add a value to the tree

- Structure and heap order properties must still be correct afterwards
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 order property
Maintain the heap order 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 does this work? What is the run time?
A Clever Trick for Storing the Heap…

Clearly, insert and deleteMin are worst-case $O(\log n)$
• But we promised average-case $O(1)$ insert (how??)

Insert requires access to the “next to use” position in the tree
• Walking the tree from root to leaf requires $O(\log n)$ steps
• Insert and Deletemin would have to update the “next to use” reference each time: $O(\log n)$

We should only pay for the functionality we need!!
• Why have we insisted the tree be complete? 😊

All complete trees of size $n$ contain the same edges
• So why are we even representing the edges?

Here comes the really clever bit about implementing heaps!!!
Array Representation of a Binary Heap

From node i:
- left child:
- right child:
- parent:

- We skip index 0 to make the math simpler
- Actually, it can be a good place to store the current size of the heap
Array Representation of a Binary Heap

From node i:
- left child: 2i
- right child: 2i+1
- parent: i / 2

We skip index 0 to make the math simpler
Actually, it can be a good place to store the current size of the heap