CSE373: Data Structures & Algorithms
Lecture 6: Priority Queues

Kevin Quinn
Fall 2015
A Quick Note:

• Homework 2 due tonight at 11pm!
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

insert e1 with priority 5
insert e2 with priority 3
insert e3 with priority 4
a = deleteMin // a = e2
b = deleteMin // b = e3
insert e4 with priority 2
insert e5 with priority 6
c = deleteMin // c = e4
d = deleteMin // d = e1

• 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 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?
More applications

• “Greedy” algorithms
  – May see an example when we study graphs in a few weeks
• Forward network packets in order of urgency
• Select most frequent symbols for data compression (cf. CSE143)
• Sorting (first insert all, then repeatedly deleteMin)
  – Much like Homework 1 uses a stack to implement reverse
Finding 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></td>
<td></td>
</tr>
<tr>
<td>unsorted linked list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sorted 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>
<tr>
<td>(our) hash table</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>unsorted linked list</td>
<td>add at front</td>
<td>O(1)</td>
</tr>
<tr>
<td>sorted array</td>
<td>search / shift</td>
<td>O(n)</td>
</tr>
<tr>
<td>sorted linked list</td>
<td>put in right place</td>
<td>O(n)</td>
</tr>
<tr>
<td>binary search tree</td>
<td>put in right place</td>
<td>O(n)</td>
</tr>
<tr>
<td>AVL tree</td>
<td>put in right place</td>
<td>$O(\log n)$ leftmost</td>
</tr>
<tr>
<td>(our) hash table</td>
<td>add</td>
<td>O(1)</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 \( \text{arr}[\text{priority}] \), \( O(1) \)
  - *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
      - *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
Structure Property: Completeness

- A Binary Heap is a complete binary tree:
  - A binary tree with all levels full, with a possible exception being the bottom level, which is filled left to right

Examples:

![Binary Heaps](image)

are these trees complete?
Heap Order Property

- The priority of every (non-root) node is greater than (or equal to) that of its parent.

Examples:

which of these are heaps?
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 (or equal to) the priority of its parent
  – *Not* a binary search tree
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 (or equal to) the priority of its parent
  - *Not* a binary search tree

![Binary Min-Heap Diagram]
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 (or equal to) 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
  - What happens if we swap with the larger child?
- Done if both children are ≥ item or reached a leaf node

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

- We will percolate down at most (height of heap) times
  - So 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 $= \lceil \log_2(n) \rceil$

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

- Add a value to the tree
- Afterwards, structure and heap properties must still be correct
- Where do we insert the new value?
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!
Array Representation of Binary Trees

From node i:

- left child: \( i \times 2 \)
- right child: \( i \times 2 + 1 \)
- parent: \( i / 2 \)

(wasting index 0 is convenient for the index arithmetic)

implicit (array) implementation:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
Judging the array implementation

Plusses:
- Less “wasted” space
  - Just index 0 and unused space on right
  - In conventional tree representation, one edge per node (except for root), so \( n-1 \) wasted space (like linked lists)
  - Array would waste more space if tree were not complete
- Multiplying and dividing by 2 is very fast (shift operations in hardware)
- Last used position is just index \texttt{size}

Minuses:
- Same might-be-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

Plusses outweigh minuses: “this is how people do it”