CS206

Priority Queues
**Performance of Trees**

<table>
<thead>
<tr>
<th></th>
<th>Complete Tree</th>
<th>Worst Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Create a dataset to make the worst possible tree
Priority Queue

• A queue that maintains order of elements according to some priority
  • Removal order, not general order
    • the rest may or may not be sorted

• Types of PQs
  • min PQ — the element with smallest key is removed first
  • max PQ — the largest is removed first

• Consider a PQ in which priority is based on insertion time
  • min PQ == ??
  • max PQ== ??
Key

• Priority queues are ordered by some key, which may be:
  • derived from the data element
    • one field
    • combination of fields
  • independent of data element
    • for example: insertion time
• best practice is to define relation between keys using `compareTo` 
• Changing `compareTo` allows changing the priority queue ordering while changing nothing else
Key-Value Pair

- Typically think of PQ as containing a pair
  - (Key, Value)
    - Key defines priority
    - Value is data the objects store
- KV pairs are frequently used
- Ideally keys are unique
  - how to handle duplicate keys?
- Ideally keys have a natural ordering.
  - Using `compareTo` allows arbitrary comparisons
- Values need not be numerical or unique
## Example - minPQ

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>Priority Queue Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(5,A)</code></td>
<td></td>
<td><code>{ (5,A) }</code></td>
</tr>
<tr>
<td><code>insert(9,C)</code></td>
<td></td>
<td><code>{ (5,A), (9,C) }</code></td>
</tr>
<tr>
<td><code>insert(3,B)</code></td>
<td></td>
<td><code>{ (3,B), (5,A), (9,C) }</code></td>
</tr>
<tr>
<td><code>min()</code></td>
<td>(3,B)</td>
<td><code>{ (3,B), (5,A), (9,C) }</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>{ (5,A), (9,C) }</code></td>
</tr>
<tr>
<td><code>removeMin()</code></td>
<td>(5,A)</td>
<td><code>{ (5,A), (7,D), (9,C) }</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>{ (7,D), (9,C) }</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>{ (9,C) }</code></td>
</tr>
<tr>
<td><code>removeMin()</code></td>
<td>(7,D)</td>
<td><code>{ }</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>{ }</code></td>
</tr>
<tr>
<td><code>removeMin()</code></td>
<td>(9,C)</td>
<td><code>{ }</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>{ }</code></td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>true</td>
<td><code>{ }</code></td>
</tr>
</tbody>
</table>
public interface PriorityQueueInterface<E extends Comparable<E>> extends BinaryTreeInterface<E> {
    E getRootElement();
    int size();
    boolean isEmpty();
    boolean contains(E element);
    void insert(E element);
    boolean remove(E element);
    E peek(); // look at min/max; do not remove
    E poll(); // removeMin/removeMax;
}
How do we implement it?

• Efficiency depends on implementation

<table>
<thead>
<tr>
<th></th>
<th>Unsorted array</th>
<th>Unsorted list</th>
<th>Sorted array</th>
<th>Sorted list</th>
</tr>
</thead>
<tbody>
<tr>
<td>peek</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>poll</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Remove may apply to any element, poll just to the “first”
Priority Queue Sort

- Sorting using a priority queue
  1. Insert with a series of `insert` operations
  2. Remove in sorted order with a series of `poll` operations
- Efficiency depends on implementation and runtime of `insert` and `poll`
Selection Sort

- **Selection-sort:**
  - select the min/max and swap with 0
- priority queue is implemented with an unsorted sequence
- $O(n^2)$
**Example**

<table>
<thead>
<tr>
<th>Input:</th>
<th>Sequence S</th>
<th>Priority Queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Phase 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>(4,8,2,5,3,9)</td>
<td>(7)</td>
</tr>
<tr>
<td>(b)</td>
<td>(8,2,5,3,9)</td>
<td>(7,4)</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>(g)</td>
<td>()</td>
<td>(7,4,8,2,5,3,9)</td>
</tr>
<tr>
<td><strong>Phase 2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>(2)</td>
<td>(7,4,8,5,3,9)</td>
</tr>
<tr>
<td>(b)</td>
<td>(2,3)</td>
<td>(7,4,8,5,9)</td>
</tr>
<tr>
<td>(c)</td>
<td>(2,3,4)</td>
<td>(7,8,5,9)</td>
</tr>
<tr>
<td>(d)</td>
<td>(2,3,4,5)</td>
<td>(7,8,9)</td>
</tr>
<tr>
<td>(e)</td>
<td>(2,3,4,5,7)</td>
<td>(8,9)</td>
</tr>
<tr>
<td>(f)</td>
<td>(2,3,4,5,7,8)</td>
<td>(9)</td>
</tr>
<tr>
<td>(g)</td>
<td>(2,3,4,5,7,8,9)</td>
<td>()</td>
</tr>
</tbody>
</table>
Insertion Sort

- Insertion-sort:
  - Insert/swap the element into the correct sorted position
- Priority queue is implemented with a sorted sequence
- $O(n^2)$
## Example

<table>
<thead>
<tr>
<th>Input:</th>
<th>Sequence S</th>
<th>Priority queue P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>(4,8,2,5,3,9)</td>
<td>(7)</td>
</tr>
<tr>
<td>(b)</td>
<td>(8,2,5,3,9)</td>
<td>(4,7)</td>
</tr>
<tr>
<td>(c)</td>
<td>(2,5,3,9)</td>
<td>(4,7,8)</td>
</tr>
<tr>
<td>(d)</td>
<td>(5,3,9)</td>
<td>(2,4,7,8)</td>
</tr>
<tr>
<td>(e)</td>
<td>(3,9)</td>
<td>(2,4,5,7,8)</td>
</tr>
<tr>
<td>(f)</td>
<td>(9)</td>
<td>(2,3,4,5,7,8)</td>
</tr>
<tr>
<td>(g)</td>
<td>()</td>
<td>(2,3,4,5,7,8,9)</td>
</tr>
<tr>
<td>Phase 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>(2)</td>
<td>(3,4,5,7,8,9)</td>
</tr>
<tr>
<td>(b)</td>
<td>(2,3)</td>
<td>(4,5,7,8,9)</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>(g)</td>
<td>(2,3,4,5,7,8,9)</td>
<td>()</td>
</tr>
</tbody>
</table>
Binary Heap

- A heap is a binary tree storing keys at its nodes and satisfying:
  - heap-order: for every internal node \( v \) other than root, \( \text{key}(v) \geq \text{key}(\text{parent}(v)) \)
  - complete binary tree: let \( h \) be the height of the heap
    - there are \( 2^i \) nodes of depth \( i, 0 \leq i \leq h - 1 \)
    - at depth \( h \), the leaf nodes are in the leftmost positions
    - last node of a heap is the rightmost node of max depth
Height of a Heap

- A heap storing $n$ keys has a height of $O(\log n)$
Insertion into a Heap

- Insert as new last node
- Need to restore heap order
Upheap

- Restore heap order
  - swap upwards
  - stop when finding a smaller parent
  - or reach root
- $O(\log n)$
Poll

- Removing the root of the heap
  - Replace root with last node
  - Remove last node $w$
  - Restore heap order
Downheap

• Restore heap order
  □ swap downwards
  □ swap with smaller child
  □ stop when finding larger children
  □ or reach a leaf

• $O(\log n)$
Heap Sort

- A PQ-sort implemented with a heap
- Space $O(n)$
- insert/poll (each) $O(\log n)$
- total time $O(n \log n)$
General Removal

- swap with last node
- delete last node
- may need to upheap or downheap

Heap:

```
5
 / 
9   11
 /   \
9    8  15
      /   \
      17    17
      /     / \
     21    19 11
```

```
1
 / 
6   9
 /   \
8    17
 /     \
21     19
```

```
22
 / 
33  27
```

Delete this node

Delete this node