Priority Queues

What operations does a queue support?

Priority: Number representing importance
- Convention lower is better priority
  
  *Bring back life form. Priority One. All other priorities rescinded.*
- Symmetric code if higher is better

Priority Queue (PQ): Supports 3 operations
- `void insert(T x, int p)`: Insert `x` with priority `p`
- `T findMin()`: Return the object with the best priority
- `void deleteMin()`: Remove the object with the best priority
Explicit Priority

\textbf{insert}(T \, x, \, \text{int} \, p)
- Priority is explicitly \text{int} \, p
- Separate from data

Implicit Priority

\textbf{insert}(\text{Comparable}<T> \, x)
- x "knows" its own priority
- Comparisons dictated by \( x\).compareTo(y) \)

Implicit is simpler for discussion: only one thing (x) to draw
Explicit usually uses a wrapper node of sorts

```
class PQNode<T> extends Comparable<PQNode> {
    int priority; T data;
    public int compareTo(PQNode that) {
        return this.priority - that.priority;
    }
}
```
Exercise: Design a PQ

Discuss

- How would you design PriorityQueue class?
- What underlying data structures would you use?
- Discuss with a neighbor
- Give rough idea of implementation
- Make it as efficient as possible in Big-O sense

Must Implement

- Constructor
- void insert(T x): Insert x, knows its own priority
- T findMin(): Return the object with the best priority
- void deleteMin(): Remove the object with the best priority
Most common way to build a PQ is using a new-ish data structure, the **Binary Heap**.

Looks similar to a Binary Search Tree but maintains a different property.
Heap and Not Heap

Which of these is a min-heap and which is not?

(a)  
(b)  

Which of these is a min-heap and which is not?
Trees and Heaps in Arrays

- Mostly we have used trees of linked Nodes
- Can also put trees/heaps in an array

Root is at 1 (discuss root at 0 later)
- left(i) = 2*i
- right(i) = 2*i + 1
Balanced v. Unbalanced in Arrays

Find the array layout of these two trees

- Root is at 1
- \( \text{left}(i) = 2 \times i \)
- \( \text{right}(i) = 2 \times i + 1 \)

Q: How big of array is required?
Balanced v. Unbalanced in Arrays

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>17</td>
<td>89</td>
<td>3</td>
<td>25</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>3</td>
<td>48</td>
<td>25</td>
<td>89</td>
<td>63</td>
<td>17</td>
<td>3</td>
<td>48</td>
<td>25</td>
<td>89</td>
<td>63</td>
<td>17</td>
<td>3</td>
<td>48</td>
</tr>
</tbody>
</table>

Jyh-Ming Lien
CS311 Data Structures Lecture 07—Priority Queue
Complete Trees

- Only "missing" nodes in their bottom row (level set)
- Nodes in bottom row are as far left as possible

Not Complete (Why?)

Complete

- Complete trees don’t waste space in arrays: no gaps
- Hard for general BSTs, easy for binary heaps...
PQ Ops with Binary Heaps

- Use an internal T array[] of queue contents
- Maintain min-heap order in array

**Define**

Tree-like ops for array[]

- root() => 1
- left(i) => i*2
- right(i) => i*2 + 1
- parent(i) => i / 2

**T findMin()**

Super easy

return array[root()];

**insert(T x)**

Ensure heap is a complete tree

- Insert at next array[size]
- Increment size
- Percolate new element up

**deleteMin()**

Ensure heap is a complete tree

- Decrement size
- Replace root with last data
- Percolate root down
Demos of Binary Heaps

Not allowed on exams, but good for studying

**Min Heap from David Galles @ Univ SanFran**
- Visualize both heap and array version
- All ops supported

**Max Heap from Steven Halim**
- Good visuals
- No array
- Slow to load
// Binary Heap, 1-indexed
public class BinaryHeapPQ<T>{
    private T [] array;
    private int size;

    // Helpers
    static int root(){
        return 1;
    }
    static int left(int i){
        return i*2;
    }
    static int right(int i){
        return i*2+1;
    }
    static int parent(int i){
        return i / 2;
    }

    // Insert a data
    public void insert(T x){
        size++;
        ensureCapacity(size+1);
        array[size] = x;
        percolateUp(size);
    }

    // Remove the minimum element
    public void deleteMin(){
        array[root()] = array[size];
        array[size] = null;
        size--;
        percolateDown(root());
    }
}

// Insert a data
public void insert(T x){
    size++;
    ensureCapacity(size+1);
    array[size] = x;
    percolateUp(size);
}

// Remove the minimum element
public void deleteMin(){
    array[root()] = array[size];
    array[size] = null;
    size--;
    percolateDown(root());
}
Percolate Up/Down

Up

void percolateUp(int xdx){
    while(xdx!=root()){
        T x = array[xdx];
        T p = array[parent(xdx)];
        if(doCompare(x, p) < 0){
            array[xdx] = p;
            array[parent(xdx)] = x;
            xdx = parent(xdx);
        }
        else{ break; }
    }
}

Down

void percolateDown(int xdx){
    while(true){
        T x = array[xdx];
        int cdx = left(xdx);
        // Determine which child
        // if any to swap with
        if(cdx > size){ break; } // No left, bottom
        if(right(xdx) < size && // Right valid
            doCompare(array[right(xdx)], array[cdx]) < 0){
            cdx = right(xdx); // Right smaller
        }
        T child = array[cdx];
        if(doCompare(child, x) < 0){ // child smaller
            array[cdx] = x; // swap
            array[xdx] = child;
            xdx = cdx; // reset index
        }
        else{ break; }
    }
}

Code distribution today contains working heap
*percolateUp()* and *percolateDown()* do most of the work
Uses ”root at index 1” convention

Weiss uses a different approach in percolate up/down
Move a ”hole” around rather than swapping
Probably saves 1 comparison per loop iteration
Have a look in *weiss/util/PriorityQueue.java*
T findMin();
void insert(T x); // x knows its priority
void deleteMin();

Give the complexity and justify for each
Efficiency of Binary Heap PQs

- `findMin()` clearly $O(1)$
- `deleteMin()` worst case height
- `insert(x)` worst case height

Height of a **Complete Binary Tree** wrt number of nodes $N$?

- Guesses?
Summary of Binary Heaps

<table>
<thead>
<tr>
<th>Op</th>
<th>Worst Case</th>
<th>Avg Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>findMin()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert(x)</td>
<td>$O(\log N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>deleteMin()</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

- Notice: No get(x) method or remove(x) methods
- These would involve searching the whole binary heap/priority queue if they did existed: $O(N)$