CS 310: Priority Queues and Binary Heaps

Chris Kauffman

Week 14-2
## Logistics

### The end is Nigh

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 11/30</td>
<td>Balanced Trees</td>
</tr>
<tr>
<td>Wed 12/2</td>
<td>Priority Queues</td>
</tr>
<tr>
<td>Mon 12/7</td>
<td>Heap Sort, Evals</td>
</tr>
<tr>
<td>Wed 12/9</td>
<td>Java Jeopardy</td>
</tr>
<tr>
<td>Fri 12/11</td>
<td>Code Fest (!?)</td>
</tr>
<tr>
<td>Sat 12/12</td>
<td>HW 4 Due</td>
</tr>
<tr>
<td>Mon 12/14</td>
<td>Final Exam</td>
</tr>
<tr>
<td></td>
<td>Sec 1: 10:30am</td>
</tr>
<tr>
<td></td>
<td>Sec 3: 1:30pm</td>
</tr>
</tbody>
</table>

### HW 4
- Overview today
- Due in 10 days

### Reading: Weiss
- 21: Priority Queue/Binary Heap
- 6.9: Priority Queue Interface

### Goals Today
- Priority queues
- Binary Heaps
TripleStore database: 3 columns Entity, Relation, Property
Each "row" is a Record, is unique
Basic add() and remove() functionality
Support for query() including wild cards
All operations are logarithmic
  \( N \): number of records in the TripleStore
  add() is \( O(\log N) \) (add single records)
  query() is \( O(\log N + K) \), \( K \) is number of matches
  remove() is \( O(K \times \log N) \), \( K \) number of items to remove

Use three trees sorted in different ways
ERP, RPE, PER, corresponding Comparators
In query(), select one tree to enable fast lookup
Use java TreeSet<Record>
Explore TreeSet.tailSet(x) which gives a sorted subset of tree in \( O(\log N) \) time and provides an iterator
// No wild fields
Record r = Record.makeRecord("Alf","EATS","cat")

// No wild fields
Record q = Record.makeQuery("*", "Alf","EATS","cat")

// Property wild in both, same query
Record q1 = Record.makeQuery("*", "Alf","EATS","*")
Record q2 = Record.makeQuery("??", "Alf","EATS","??")
TripleStore: Which Tree for Query?

\[ f = \text{fixed string} \]
\[ * = \text{wild card} \]

<table>
<thead>
<tr>
<th>E</th>
<th>R</th>
<th>P</th>
<th>Tree</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>f</td>
<td>*</td>
<td>ERP</td>
<td>Entity/Relation fixed, scan through Property</td>
</tr>
<tr>
<td>f</td>
<td>*</td>
<td>*</td>
<td>ERP</td>
<td>Entity fixed, scan through Relation/Property</td>
</tr>
<tr>
<td>f</td>
<td>*</td>
<td>f</td>
<td>PER</td>
<td>Property/Entity fixed, scan through Relations</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>f</td>
<td>PER</td>
<td>Property fixed, scan through Entities/Relations</td>
</tr>
</tbody>
</table>

How many rows total?
Priority Queues

Queue
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
Priority

Explicit Priority

insert(T x, int p)
  ▶ Priority is explicitly int p
  ▶ Separate from data

Implicit Priority

insert(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 the object with the best priority
Binary Heap: Sort of Sorted

- 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

**BST Property**
A Node must be bigger than its left children and smaller than its right children

**Binary Min-Heap Property**
A Node must be smaller than its children
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
- left(i) = 2*i
- right(i) = 2*i + 1

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

<table>
<thead>
<tr>
<th></th>
<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></td>
<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></th>
<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></td>
<td>17</td>
<td>3</td>
<td>48</td>
<td>25</td>
<td>89</td>
<td>63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 trees don’t waste space in arrays: no gaps
Hard for general BSTs, easy for binary heaps...
Trees/Heaps in Array: Keep them Complete

- Storing in arrays: can cost space overhead
- If the tree is **complete** or nearly so, little wasted space

### BSTs in arrays

- Hard to keep tree complete
- BST + balancing property makes it hard
- Rotations may not be constant time anymore
- Trees not usually laid out in arrays: linked nodes much more common

### Binary Heaps in arrays

- **Very easy** to keep tree complete
- Heap Property is more loose, easier to maintain
- No rotations, no worries..
- Binary heaps almost always laid out in arrays
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
PQ/Binary Heap Code

BinaryHeapPQ.java

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

Text Book Binary Heap

- 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`
Complexity of Binary Heap PQ methods?

T findMin();
void insert(T x); // x knows its priority
void deleteMin();

Give the complexity and justify for each
Height Again...

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 $N$?
Look it up for next time

Reading

- 21: Priority Queue/Binary Heap
- 6.9: Priority Queue Interface