**YAQ, YAQ, haha! (Yet Another Queue)**

- What is the dequeue policy for a Queue?
  - Why do we implement Queue with LinkedList
    - Interface and class in java.util
  - Can we remove an element other than first?

- How does queue help word-ladder/shortest path?
  - First item enqueued/added is the one we want
  - What if different element is “best”?

- PriorityQueue has a different dequeue policy
  - Best item is dequeued, queue manages itself to ensure operations are efficient

**PriorityQueue Apps to raison d'être**

- Algorithms Using PQ for efficiency
  - Shortest Path: from Mapquest to Internet Routing
    - How is this like word-ladder? How different?
  - Connecting all outlets in a house with minimal wiring
    - Minimal spanning tree in graph
  - Optimal A* search, game-playing, AI
    - Can’t explore entire search space, can estimate good move

- Data compression facilitated by priority queue
  - Alltime best assignment in a Compsci 100 course?
    - Subject to debate, of course
  - From A-Z, soup-to-nuts, bits to abstractions

**PQ Application: Data Compression**

- Compression is a high-profile application
  - .zip, .mp3, .jpg, .gif, .gz, ...
  - What property of MP3 was a significant factor in what made Napster work (why did Napster ultimately fail?)
  - Who invented Napster, how old, when?

- Why do we care?
  - Secondary storage capacity doubles every year
  - Disk space fills up quickly on every computer system
  - More data to compress than ever before
  - Will we ever need to stop worrying about storage?

**More on Compression**

- Different compression techniques
  - .mp3 files and .zip files?
  - .gif and .jpg?
  - Lossless and lossy

- Impossible to compress/lossless everything: Why?

- Lossy methods
  - Good for pictures, video, and audio (JPEG, MPEG, etc.)

- Lossless methods
  - Run-length encoding, Huffman, LZW, ...
Priority Queue

- Compression motivates ADT priority queue
  - Supports two basic operations
    - add/insert -- an element into the priority queue
    - remove/delete -- the minimal element from the priority queue
  - Implementations allow getmin/peek as well as delete
    - Analogous to top/pop, peek/dequeue in stacks, queues

- Think about implementing the ADT, choices?
  - Add compared to min/remove
  - Balanced search tree is ok, but can we do better?

Priority Queue sorting

- See PQDemo.java,
  - code below sorts, complexity?

  ```java
  String[] array = { ... }; // array filled with data
  PriorityQueue<String> pq = new PriorityQueue<String>();
  for (String s : array) pq.add(s);
  for (int k = 0; k < array.length; k++) {
    array[k] = pq.remove();
  }
  ```

- Bottlenecks, operations in code above
  - Can we improve looking at all words?
  - What about add and remove?

Priority Queue top-M sorting

- What if we have lots and lots and lots of data
  - code below sorts top-M elements, complexity?

  ```java
  Scanner s = ... // initialize;
  PriorityQueue<String> pq = new PriorityQueue<String>();
  while (s.hasNext()) {
    pq.add(s.next());
    if (pq.size() > M) pq.remove();
  }
  ```

- What’s advantageous about this code?
  - Store everything and sort everything?
  - Store everything, sort first M?
  - What is complexity of sort: O(n log n)

Priority Queue implementations

- Implementing priority queues: average and worst case

<table>
<thead>
<tr>
<th></th>
<th>Insert average</th>
<th>Getmin (delete)</th>
<th>Insert worst</th>
<th>Getmin (delete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted list</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Search tree</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
</tr>
<tr>
<td>Balanced tree</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
</tr>
<tr>
<td>Heap</td>
<td>O(1)</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
</tr>
</tbody>
</table>

- Heap has O(n) build heap from n elements
PriorityQueue.java (Java 5+)

- What about objects inserted into pq?
  - Comparable, e.g., essentially sortable
  - How can we change what minimal means?
  - Implementation uses heap, tree stored in an array

- Use a Comparator for comparing entries we can make a min-heap act like a max-heap, see PQDemo
  - Where is class Comparator declaration? How used?
  - What if we didn't know about Collections.reverseOrder?
    - How do we make this ourselves?

Sorting w/o Collections.sort(…)

```java
public static void sort(ArrayList<String> a) {
    PriorityQueue<String> pq = new PriorityQueue<String>();
    pq.addAll(a);
    for (int k = 0; k < a.size(); k++) a.set(k, pq.remove());
}
```

- This works, regardless of pq implementation?
- What is the complexity of this method?
  - add O(1), remove O(log n)? If add O(log n)?
  - heapsort uses array as the priority queue rather than separate pq object, so no extra storage
  - From a big-Oh perspective no difference: Θ(n log n)
    - Is there a difference? What’s hidden with O notation?

Big-Oh and a tighter look at inserts

- What is \(\log(1) + \log(2) + \log(3) + \ldots + \log(n)\)
  - Property of logs, \(\log(a) + \log(b) = \log(a \cdot b)\)
  - \(\log(1+2+3+\ldots+n) = \log(n!}\)

- We can show using Sterling's formula:
  \(n! \approx \sqrt{2\pi n} n^n e^{-n}\)

- \(\log(n!) = c_1 \log(n) + n \log(n) - c_2 n\)

- We can get \(O(n \log n)\) easily, this goes tight, lower, \(\Omega(n \log n)\) as well

Priority Queue implementation

- Heap data structure is fast and reasonably simple
  - Why not use inheritance hierarchy as was used with Map?
  - Trade-offs when using HashMap and TreeMap:
    - Time, space, ordering properties, e.g., what does TreeMap support?

- Changing comparison when calculating priority?
  - Create object to replace, or in lieu of compareTo
  - Comparable interface compares this to passed object
  - Comparator interface compares two passed objects

- Both comparison methods: compareTo() and compare()
  - Compare two objects (parameters or self and parameter)
  - Returns -1, 0, +1 depending on `<`, `==`, `>`
Creating Heaps

- Heap is an array-based implementation of a binary tree used for implementing priority queues, supports:
  - add/insert, peek/getmin, remove/deletemin, O(???)
- Using array minimizes storage (no explicit pointers), faster too — children are located by index/position in array
- Heap is a binary tree with shape property, heap/value property
  - shape: tree filled at all levels (except perhaps last) and filled left-to-right (complete binary tree)
  - each node has value smaller than both children

Array-based heap

- store “node values” in array beginning at index 1
- for node with index k
  - left child: index $2k$
  - right child: index $2k + 1$
- why is this conducive for maintaining heap shape?
- what about heap property?
- is the heap a search tree?
- where is minimal node?
- where are nodes added? deleted?

Thinking about heaps

- Where is minimal element?
  - Root, why?
- Where is maximal element?
  - Leaves, why?
- How many leaves are there in an N-node heap (big-Oh)?
  - O(n), but exact?
- What is complexity of find max in a minheap? Why?
  - O(n), but $\frac{1}{2}N$?
- Where is second smallest element? Why?
  - Near root?

Adding values to heap

- to maintain heap shape, must add new value in left-to-right order of last level
  - could violate heap property
  - move value “up” if too small
- change places with parent if heap property violated
  - stop when parent is smaller
  - stop when root is reached
- pull parent down, swapping isn’t necessary (optimization)
Adding values, details (pseudocode)

```java
void add(Object elt) {
    // add elt to heap in myList
    myList.add(elt);
    int loc = myList.size()-1;
    while (1 < loc &&
          elt < myList.get(loc/2)) {
        myList.set(loc,myList.get(loc/2));
        loc = loc/2; // go to parent
    }
    // what's true here?
    myList.set(loc,elt);
}
```

Removing minimal element

- Where is minimal element?
  - If we remove it, what changes, shape/property?
- How can we maintain shape?
  - “last” element moves to root
  - What property is violated?
- After moving last element, subtrees of root are heaps, why?
  - Move root down (pull child up) does it matter where?
- When can we stop “re-heaping”?
  - Less than both children
  - Reach a leaf

Anita Borg 1949-2003

- “Dr. Anita Borg tenaciously envisioned and set about to change the world for women and for technology… she fought tirelessly for the development technology with positive social and human impact.”
- “Anita Borg sought to revolutionize the world and the way we think about technology and its impact on our lives.”
- [http://www.youtube.com/watch?v=1yPxd5jgq_Q](http://www.youtube.com/watch?v=1yPxd5jgq_Q)

Digression: word ladders

- How many ladders from cart to dire as shown?
  - Enqueue dare more than once?
  - Downside? Alternative?
- We want to know number of ladders that end at W.
  - What do we know initially?
  - When we put something on the queue, what do we know?
  - How do we keep track?
- Similar to tree-counting?
Word Ladder: more details

- # ladders that end at dare
  - At each word W
- Ladder length to W
  - Calculable from??
- Two maps

Dequeue s
- foreach W one-away
  - if not-seen ???
  - else ???

Digression: Bogglescore APT

- How many ways to find Aha?!
  - Each one scores points
  - Related to genomics problem

- Number of ways that AHA ends at (2,1)?
  - What do we know initially?
  - When we extend search what do we know?
  - How do we keep track?

- Why do we need to avoid revisits?
  - Caching, memoizing needed!

Digression: Bogglescore APT

# times we can find: AAAAA...AAA
- # 50-step paths in grid, why? Big? Small?
- Calculate # 49 step paths at each spot
  - Foreach cell, sum neighbors
  - If recursive, what happens?

- Memoize on row, column, word-length
  - For each word, clear cache/memo
  - For every row, column
    - If [r][c] == first char, recurse

- String shorter by 1
  - Toward length 1, stop and return 1
  - If seen [r][c][length] return value
  - When do we return zero?

Bogglescore alternative

- Last solution was lexicon-first with memoizing
  - Visit every cell, but don’t revisit with identical substring

- What about board-first solution?
  - When we visit a cell, we’re working with word-so-far
  - Avoid revisiting cell with same word-so-far
    - Memoize on word/row/col, make this Map-able
    - Hash on word, row, and col
    - Compare based on word, row, and col
  - Don’t recurse if not a prefix
  - Store score if word-so-far is a word, still recurse