Preliminaries: queues vs. stacks

Stack: last-in-first-out (LIFO)
Our IList classes work like this.

Queue: first-in-first-out (FIFO)
Useful for all kinds of things, especially in parallel processing, networking, etc.

Functional vs. mutating?
edu.rice.tree.IQueue has a “functional” interface
Often done with mutation (see, e.g., java.util.LinkedList add/remove)
How do you implement a mutating queue?

Option 1: doubly-linked list (java.util.LinkedList)
Every node has pointers to next and previous nodes
“Sentinel” node to deal with empty-list case
O(1) insert and removal
Technically possible to add and remove from either end (= Deque or Dequeue)

Option 2: array (java.util.ArrayDeque)
Tricky: requires growing the array, tracking start/end, wraparound
O(1) insert and removal (amortized cost: sometimes requires an O(n) copy)
How about a **functional** queue?

**Option 1: use two lists inside**

“Inbox” and “outbox”

- New inputs go to the front of the inbox - $O(1)$
- If the outbox is empty, then reverse the inbox - $O(n)$
- Subsequent reads are constant time - $O(1)$
  
  - Amortized cost: still constant time even with reversing!
  - You did this in your lab this week.

**Option 2: use a tree (not recommended)**

- Associate a counter with each value, tree splits on counter
- “GetMin” to return the earliest
- All operations are $O(\log n)$
So what's a *priority* queue?

Priority queues are typically implemented via mutation. A `getMin()` function that returns the smallest value.

```java
class PriorityQueue<T extends Comparable<T>> {
    T getMin();
    void insert(T val);
    int size();
    boolean empty();
}
```
Binary heaps: efficient priority queues

Storage happens in an array (or java.util.ArrayList)

Map a tree into the array. No pointers, just math on array indices.

lchild(i) = 2i + 1
rchild(i) = 2i + 2
parent(i) = (i−1)/2

Note: integer arithmetic

A tree like this is called “complete”.

<table>
<thead>
<tr>
<th>Heap values</th>
<th>2</th>
<th>4</th>
<th>3</th>
<th>7</th>
<th>9</th>
<th>6</th>
<th>5</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Binary heaps: data definition

“The heap property”:
A parent’s value is less than (or equal to) its children

Ergo:
The minimum value is in the root

This is the same “heap property” we used in treaps.
Binary heaps: inserting a new value

First, go to the next available slot, put the value in
Work upward, swapping to create the heap property

insert(3)
Binary heaps: inserting a new value

First, go to the next available slot, put the value in
Work upward, swapping to create the heap property

insert(3)
Binary heaps: inserting a new value

First, go to the next available slot, put the value in
Work upward, swapping to create the heap property

insert(3)
Binary heaps: inserting a new value

First, go to the next available slot, put the value in
Work upward, swapping to create the heap property

insert(3)
Binary heaps: inserting a new value

First, go to the next available slot, put the value in
Work upward, swapping to create the heap property

insert(3)
Binary heaps: inserting a new value

First, go to the next available slot, put the value in
Work upward, swapping to create the heap property

insert(3)
Binary heaps: inserting a new value

First, go to the next available slot, put the value in
Work upward, swapping to create the heap property

insert(3)
Binary heaps: inserting a new value

First, go to the next available slot, put the value in
Work upward, swapping to create the heap property

insert(3)

Insertion cost: $O(\log n)$
Binary heaps: removing the min value

The answer is right on top, so that’s easy
But what about the children?

First, the last child *must* move
Binary heaps: removing the min value

The answer is right on top, so that’s easy
But what about the children?

First, the last child must move
So let’s start it off on top!
Binary heaps: removing the min value

The answer is right on top, so that's easy
But what about the children?

First, the last child must move
So let's start it off on top!

Now, recursively work down
Reestablish “heap property”
  Use smallest value (either child!)
  If values are equal, either is fine.
Binary heaps: removing the min value

The answer is right on top, so that’s easy
But what about the children?

First, the last child must move
So let’s start it off on top!

Now, recursively work down
Reestablish “heap property”
Use smallest value (either child!)
If values are equal, either is fine.
Binary heaps: removing the min value

The answer is right on top, so that’s easy
But what about the children?

First, the last child must move
So let’s start it off on top!

Now, recursively work down
Reestablish “heap property”
Use smallest value (either child!)
If values are equal, either is fine.
Binary heaps: removing the min value

The answer is right on top, so that’s easy
But what about the children?

First, the last child must move
So let’s start it off on top!

Now, recursively work down
Reestablish “heap property”
  Use smallest value (either child!)
  If values are equal, either is fine.
The answer is right on top, so that’s easy
But what about the children?

First, the last child **must** move
So let’s start it off on top!

Now, recursively work down
Reestablish “heap property”
Use smallest value (either child!)
If values are equal, either is fine.
Binary heaps: removing the min value

The answer is right on top, so that’s easy
But what about the children?

First, the last child **must** move
So let’s start it off on top!

Now, recursively work down
Reestablish “heap property”
Use smallest value (either child!)
If values are equal, either is fine.
Binary heaps: removing the min value

The answer is right on top, so that’s easy
But what about the children?

First, the last child must move
So let’s start it off on top!

Now, recursively work down
Reestablish “heap property”
Use smallest value (either child!)
If values are equal, either is fine.

Total cost: $O(\log n)$
Mapping onto arrays is a serious win

Less space (no need for all those left/right pointers)

Navigating up or down is easy

$O(\log n)$ operations aren't just “expected case”
A “complete” tree cannot degenerate into a $O(n)$ list-like structure
Performance numbers

**FIFO queues (functional)**

- **ListQueue**: 1000000 inserts, 10000 fetches: 0.170 μs per insert
- **TreapQueue**: 1000000 inserts, 10000 fetches: 0.387 μs per insert

**FIFO queues (mutating)**

- **LinkedList**: 1000000 inserts, 10000 fetches: 0.020 μs per insert
- **ArrayDeque**: 1000000 inserts, 10000 fetches: 0.018 μs per insert

**Priority queues (mutating)**

- **BinaryHeap**: 1000000 inserts, 10000 fetches: 0.099 μs per insert
- **PriorityQueue**: 1000000 inserts, 10000 fetches: 0.061 μs per insert
## Performance numbers

### FIFO queues (functional)

<table>
<thead>
<tr>
<th>Queue</th>
<th>Inserts</th>
<th>Fetches</th>
<th>μs per insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListQueue</td>
<td>1000000</td>
<td>10000</td>
<td>0.170</td>
</tr>
<tr>
<td>TreapQueue</td>
<td>1000000</td>
<td>10000</td>
<td>0.387</td>
</tr>
</tbody>
</table>

### FIFO queues (mutating)

<table>
<thead>
<tr>
<th>Queue</th>
<th>Inserts</th>
<th>Fetches</th>
<th>μs per insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>1000000</td>
<td>10000</td>
<td>0.020</td>
</tr>
<tr>
<td>ArrayDeque</td>
<td>1000000</td>
<td>10000</td>
<td>0.018</td>
</tr>
</tbody>
</table>

### Priority queues (mutating)

<table>
<thead>
<tr>
<th>Queue</th>
<th>Inserts</th>
<th>Fetches</th>
<th>μs per insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinaryHeap</td>
<td>1000000</td>
<td>10000</td>
<td>0.099</td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>1000000</td>
<td>10000</td>
<td>0.061</td>
</tr>
</tbody>
</table>

*Mutation is winning big, this time...*
Live coding
Q&A for project 4