Queues, Priority Queues and Heaps

7/23/2009

Many thanks to David Sun for some of the included slides!

Important Dates

- Project 2 due
  - DESIGN, DESIGN, DESIGN!!!
  - You may have 0 or 1 partner.
    - NO EXCEPTIONS!
  - Due Friday 7/24/2009 – 10pm
- Midterm Review
  - Saturday 7/26/2009 – 1-4pm in 306 Soda
- Midterm 2
  - Tuesday 7/28/2009 – 5-6pm in 10 Evans

Queues (Review)

```
public interface Queue {
  public int size();
  public boolean isEmpty();
  public void enqueue(Object item);
  public Object dequeue() throws EmptyQueueException;
  public Object front() throws EmptyQueueException;
}
```

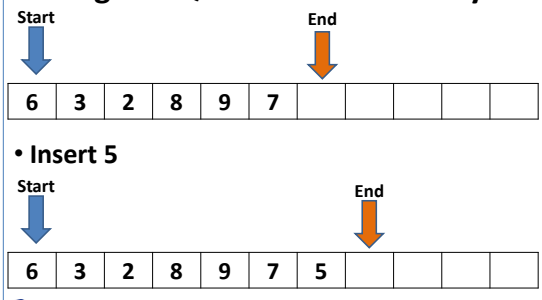
What data structure would you use to store a queue?

- Hash Table
- Binary Search Tree
- Linked List
- Array

Using a Hash Table to implement a Queue (not a "good" idea)

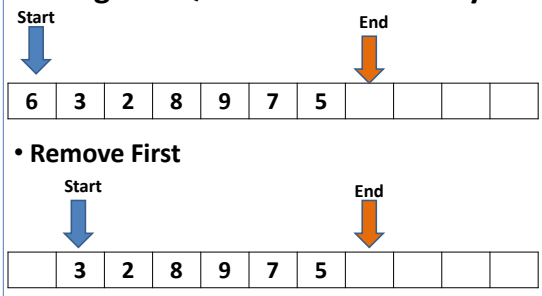
- Key is the # in line
- Value is the object in the queue
- Keep track of the current number

Regular Queues with an Array



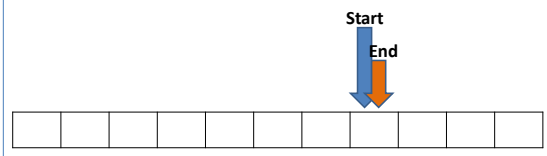
- Insert 5

Regular Queues with an Array



- Remove First

An Empty Queue



A Queue Based upon Priority PriorityQueue

```
public interface PriorityQueue {
    public int size();
    public boolean isEmpty();
    Entry insert(Object k, Object v);
    Entry max();
    Entry removeMax();
    }
```

PriorityQueues Implemented with Sorted Arrays

Operation	Run Time	Other Requirements
`size()`		
`isEmpty()`		
`insert(…)`		
`max()`		
`removeMax()`		

PriorityQueues Implemented with Non-Sorted Arrays

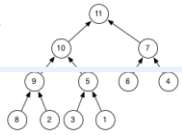
Operation	Run Time	Other Requirements
`size()`		
`isEmpty()`		
`insert(…)`		
`max()`		
`removeMax()`		

Binary Heaps

- A Binary Heap is a binary tree, with two additional properties
  - **Shape Property:** It is a complete binary tree – a binary tree in which every row is full, except possibly the bottom row, which is filled from left to right.
  - **Heap Property (or Heap Order Property):** *No child has a key greater than its parent's key.* This property is applied recursively: any subtree of a binary heap is also a binary heap.
- If we use the notion of *smaller than* in the Heap Property we get a *min-heap.* We'll look at *max-heap* in this lecture.

Heaps Implement PriorityQueue

```
public interface PriorityQueue {
    public int size();
    public boolean isEmpty();
    Entry insert(Object k, Object v);
    Entry max();
    Entry removeMax();
    }
```



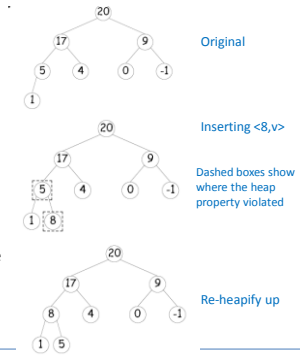
max()

- Trivial: The heap-order property ensures that the entry with the maximum key is always at the *top* of the heap. Hence, we simply return the entry at the root node.
  - If the heap is empty, return null or throw an exception.
- Runs in O(1) time.

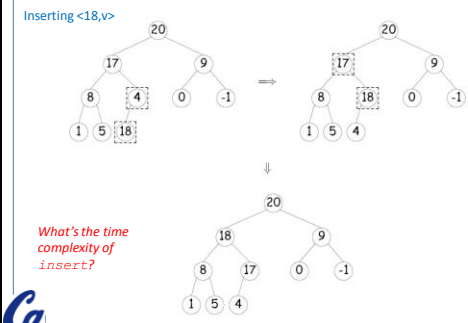
insert(8)

Let *x* be the new entry *(k, v)*.

1. Place the new entry *x* in the bottom level of the tree, at the first free spot from the left. If the bottom level is full, start a new level with *x* at the far left.
2. If the new entry's key violates the heap-order property then compare *x*'s key with its parent's key; if *x*'s key is larger, we exchange *x* with its paren. Repeat the procedure with *x*'s new parent.



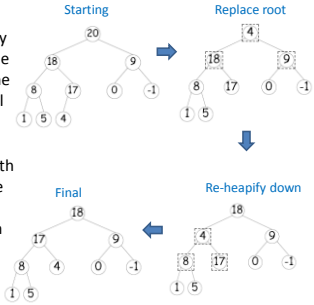
insert(18)



What's the time complexity of `insert`*?*

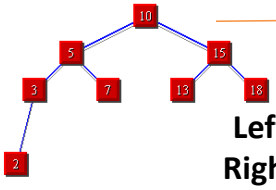
removeMax()

1. If the heap is empty, return null or throw an exception.
2. Otherwise, remove the entry at the root node. Replace the root with the last entry in the tree x, so that the tree is still complete.
3. If the root violates the heap property then compare *x* with it's children, swap *x* with the child with the **larger** key, repeat until *x* is greater than or equal to its children or reach a leaf.



Representing Trees (Review)

Array representations are common for Heaps



This is not a heap!

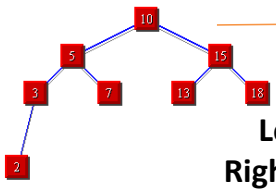
Left Child at 2n + 1

Right Child at 2n + 2

0	1	2	3	4	5	6	7	8	9
10	5	15	3	7	13	18	2		

A Different Tree Implementation

(don't use the 0 index)



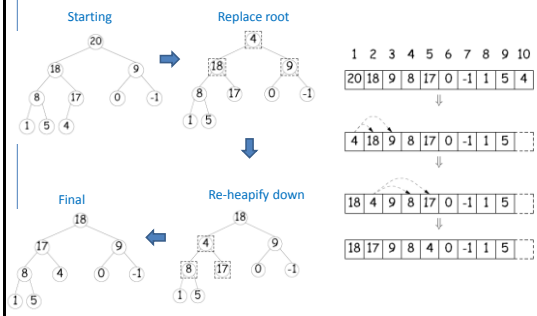
This is not a heap!

Left Child at 2n

Right Child at 2n + 1

	10	5	15	3	7		18			
0	1	2	3	4	5	6	7	8	9	10

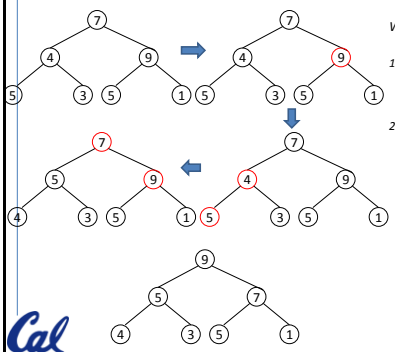
`removeMax()`



Bottom-Up Heap Construction

- Suppose we are given a bunch of randomly ordered entries, and want to make a heap out of them.
- What's the obvious way
  - Apply `insert` to each item in *O(n log n)* time.
- A better way: `bottomUpHeap()`
  1. Make a complete tree out of the entries, in any *random* order.
  2. Start from the last internal node (non-leaf node), in reverse order of the level order traversal, *heapify down* the heap as in `removeMax`().

Example



Why this works? We can argue inductively:

1. *The leaf nodes satisfy the heap order property vacuously (they have no children).*
2. *Before we bubble an entry down, we know that its two child subtrees* ***must be*** *heaps. Hence, by bubbling the entry down, we create a larger heap rooted at the node where that entry started.*

Cost of Bottom Up Construction

- If *each* internal node bubbles *all the way* down, then the running time is proportional to the sum of the heights of all the nodes in the tree.
- Turns out this sum is less than n, where n is the number of entries being coalesced into a heap.
- Hence, the running time is in *O(n)*, which is better than inserting n entries into a heap individually.

Running Times (Appendix)

- We could use a list or array, sorted or unsorted, to implement a priority queue. The following table shows running times for different implementations, with *n* entries in the queue.

	Binary Heap	Sorted List/Array	Unsorted List/Array
max	Theta(1)	Theta(1)	Theta(n)
insert (worst-cast)	Theta(log n)*	Theta(n)	Theta(1)*
insert(best-case)	Theta(1)*	it depends	Theta(1)*
removeMax (worst)	Theta(log n)	Theta(1)	Theta(n)
removeMax (best)	Theta(1)	Theta(1)	Theta(n)

* If you are using an array-based data structure, these running times assume that you don't run of room. If you do, it will take Omega(n) time to allocate a larger array and copy them into it.