CSE332: Data Abstractions

Lecture 4: Priority Queues

Tyler Robison

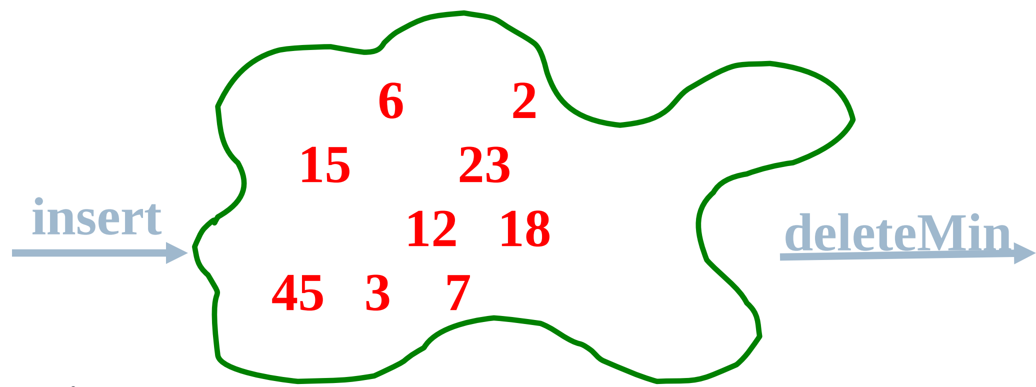
Summer 2010

A new ADT: Priority Queue

- Textbook Chapter 6: Priority Queues
  - Will go back to binary search trees (4) and hashtables (5) later
- A **priority queue** holds *compare-able data*
  - Unlike stacks and queues need to *compare items*
    - Given *x* and *y*, is *x* less than, equal to, or greater than *y*
    - What this means can depend on your data
      - Numbers: numeric ordering
      - Strings: lexicon ordering
      - Employee profile: lexicon ordering on name? Id?
  - Much of course will require comparable items:
    - Sorting
    - Binary Search Trees
  - Integers are comparable, so will use them in examples
    - But the priority queue ADT is much more general

Priority Queues

- Assume each item has a "priority"
  - The *lesser value* item is the one with the *greater* priority
  - So "priority 1" is more important than "priority 4"
  - (Just a convention)



- Operations:
  - `insert`
  - `deleteMin`
  - `create`, `is_empty`, `destroy`

- Key property: **`deleteMin`** returns and deletes from the queue the item with greatest priority (lowest priority value)
  - Can resolve ties arbitrarily

Focusing on the numbers

- For simplicity in lecture, we'll often suppose items are just `int`s and the `int` is the priority
  - So an operation sequence could be

```
insert 6
insert 5
x = deleteMin
```

  – `int` priorities are common, but really just need comparable
  - Not having "other data" is very rare
    - Example: print job is a priority *and* the file

Example

`insert` *5*
`insert` *3*
`insert` *4*
a = **`deleteMin`** **3**
b = **`deleteMin`** **4**
`insert` *2*
`insert` *6*
c = **`deleteMin`** **2**
d = **`deleteMin`** **5**

- Analogy: **`insert`** is like **`enqueue`**, **`deleteMin`** is like **`dequeue`**
  - But the whole point is to use priorities instead of FIFO

Applications

Like all good ADTs, the priority queue arises often

- Run multiple programs in the operating system
  - “critical” before “interactive” before “compute-intensive”
  - Maybe let users set priority level
- Treat hospital patients in order of severity (or triage)
- Select print jobs in order of decreasing length?
- Forward network packets in order of urgency
- Select most frequent symbols for data compression (cf. CSE143)
- Sort: **`insert`** all, then repeatedly **`deleteMin`**
  - Much like Project 1 uses a stack to implement reverse

More applications for Priority Queues

- "Greedy" algorithms
  - Perform the 'best-looking' choice at the moment
  - Will see an example when we study graphs in a few weeks
- Discrete event simulation (system modeling, virtual worlds, …)
  - Simulate how state changes when events fire
  - Each event *e* happens at some time t and generates new events *e1*, …, *en* at times *t+t1*, …, *t+tn*
  - Naïve approach: advance "clock" by 1 unit at a time and process any events that happen then
  - Better:
    - *Pending events* in a priority queue (priority = time happens)
    - Repeatedly: **`deleteMin`** and then **`insert`** new events
    - Effectively, "set clock ahead to next event"

Need a good data structure!

- Will show an efficient, non-obvious data structure for this ADT
  - But first let's analyze some "obvious" ideas for *n* data items
  - All times worst-case; but assume arrays "have room"

data	*insert algorithm / time*		*deleteMin algorithm / time*	
unsorted array	add at end	$O(1)$	search	$O(n)$
unsorted linked list	add at front	$O(1)$	search	$O(n)$
sorted circular array	search / shift	$O(n)$	move front	$O(1)$
sorted linked list	put in right place	$O(n)$	remove at front	O(1)
binary search tree	put in right place	$O(n)$	leftmost	$O(n)$

More on possibilities

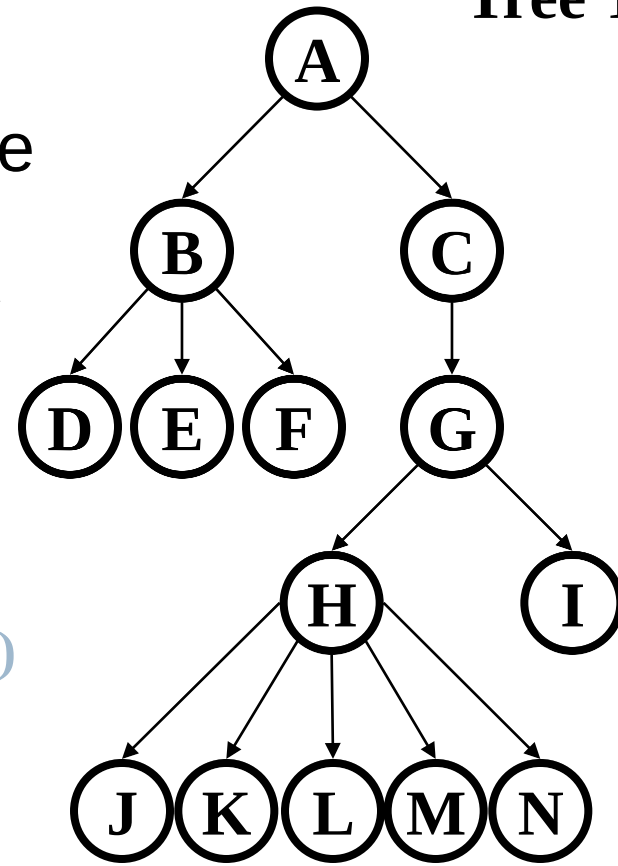
- If priorities are random, binary search tree will likely do better
  - $O($`log` $n)$ `insert` and $O($`log` $n)$ `deleteMin` on average

- One more idea: if priorities are 0, 1, …, *k* can use array of lists
  - `insert`: add to front of list at `arr[priority]`, $O(1)$
  - `deleteMin`: remove from lowest non-empty list $O(k)$
  - Only really feasible for small k

- But we are about to see a data structure called a "binary heap"
  - $O($`log` $n)$ `insert` and $O($`log` $n)$ `deleteMin` worst-case
  - Very good constant factors
  - If items arrive in random order, then `insert` is $O(1)$ on average!

Tree terms (review?)

Tree T



The binary heap data structure implementing the priority queue ADT will be a *tree*, so worth establishing some terminology

root(tree)

children(node)

parent(node)

leaves(tree)

siblings(node)

ancestors(node)

descendents(node)

subtree(node)

depth(node)

height(tree)

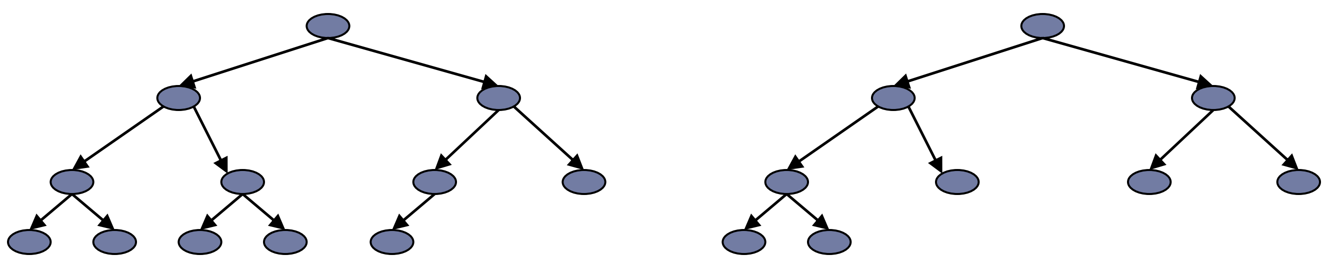
degree(node)

branching factor(tree)

Kinds of trees

Certain terms define trees with specific structure

- Binary tree: Each node has at most 2 children
- *n*-ary tree: Each node as at most *n* children
- Complete tree: Each row is completely full except maybe the bottom row, which is filled from left to right



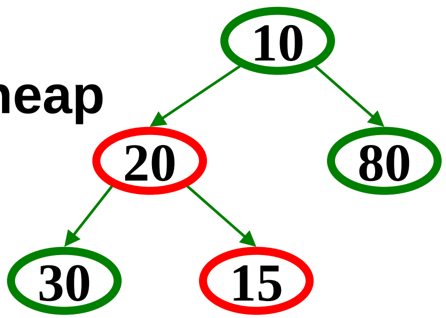
Later we'll learn a tree is a kind of directed graph with specific structure

Binary Heap: Priority Queue DS

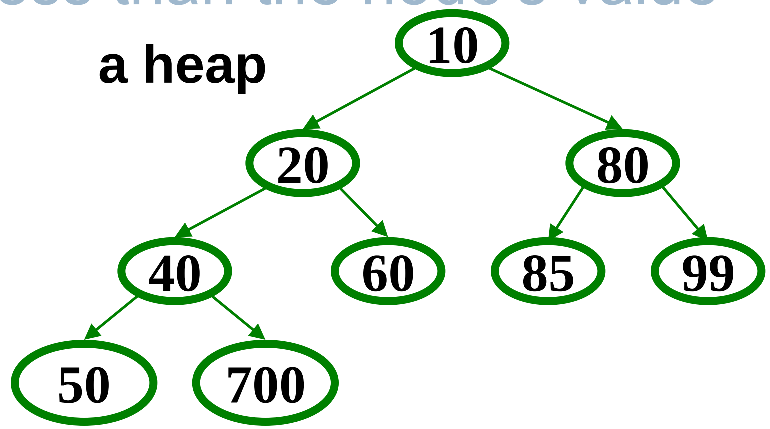
Finally, then, a *binary min-heap* (aka *binary heap* or just *heap*) has the following 2 properties:

- Structure property : A complete tree
- Heap ordering property: For every (non-root) node the parent node's value is less than the node's value

not a heap



a heap

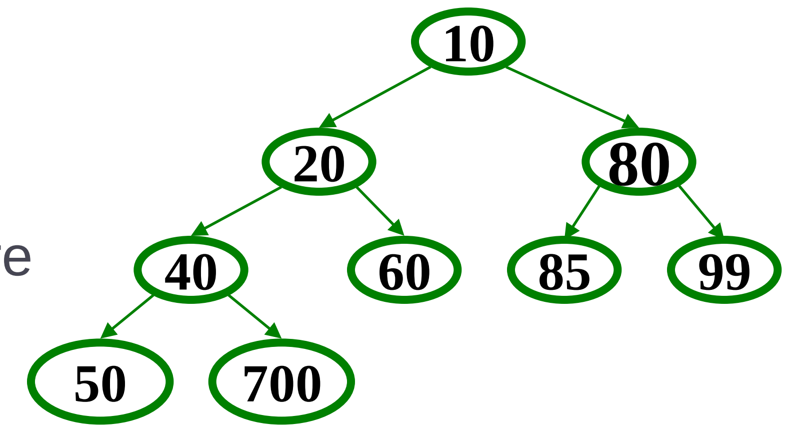


So:

- Where is the highest-priority item? **root**
- What is the height of a heap with *n* items? **O(logn)**

Operations: basic idea

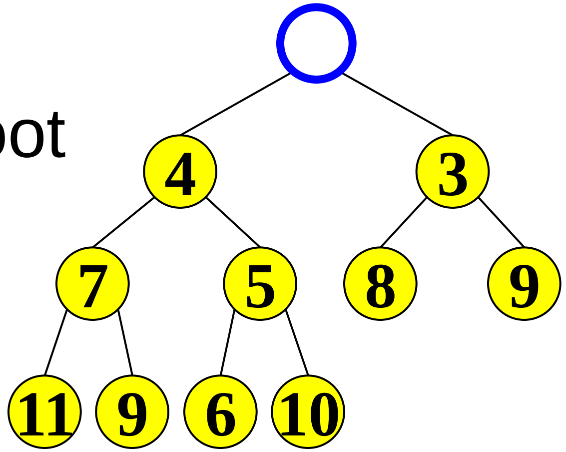
- **`findMin`**: return **`root.data`**
- **`deleteMin`**:
  1. **`answer = root.data`**
  2. Move right-most node in last row to root to restore structure property
  3. "Percolate down" to restore heap property



- **`insert:`**
  1. Put new node in next position on bottom row to restore structure property
  2. "Percolate up" to restore heap property

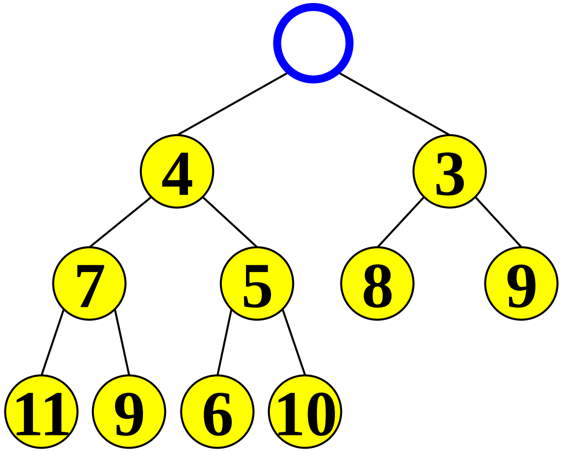
DeleteMin

1. Delete (and return) value at root node

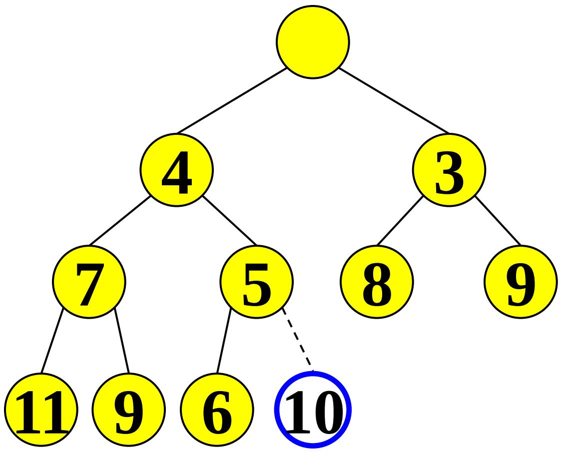


2. Restore the Structure Property

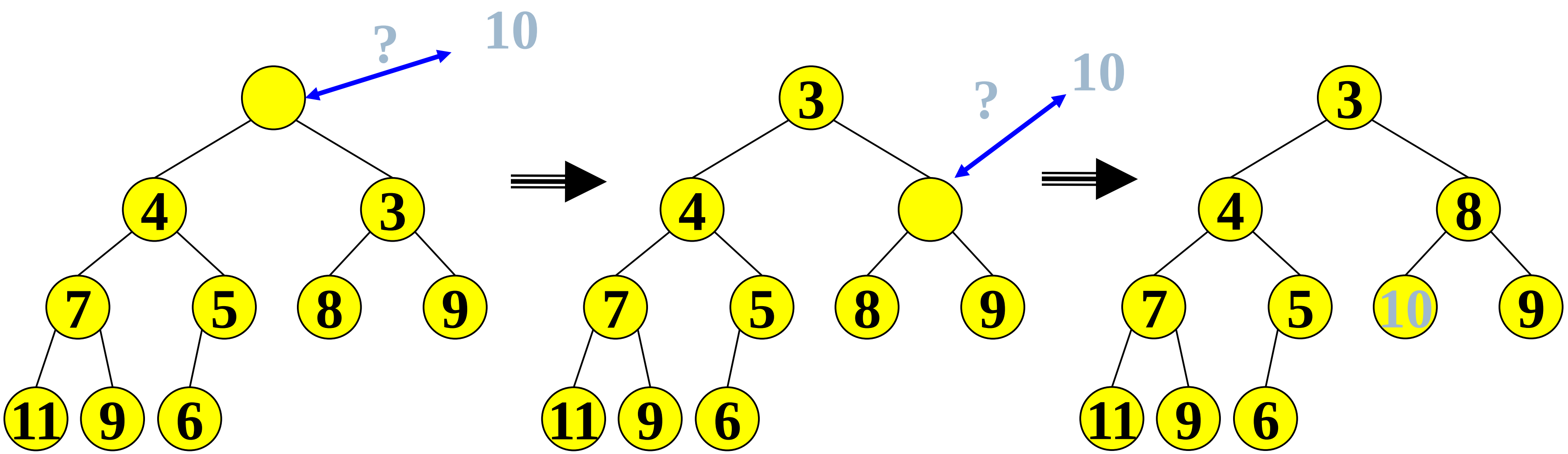
- We now have a "hole" at the root
  - Need to fill the hole with another value



- When we are done, the tree will have one less node and must still be complete



3. Restore the Heap Property



Percolate down:

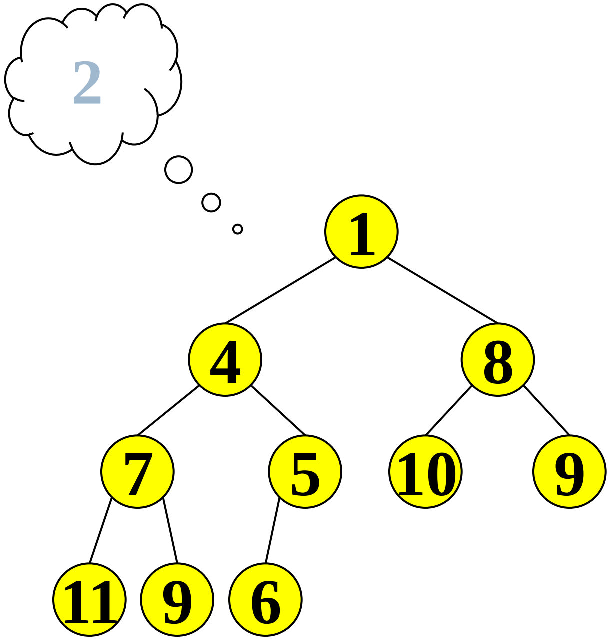
- Keep comparing with both children
- Move smaller child up and go down one level
- Done if both children are ≥ item or reached a leaf node
- What is the run time?

O(logn)

•Why not swap with larger of children, if it's smaller than both?

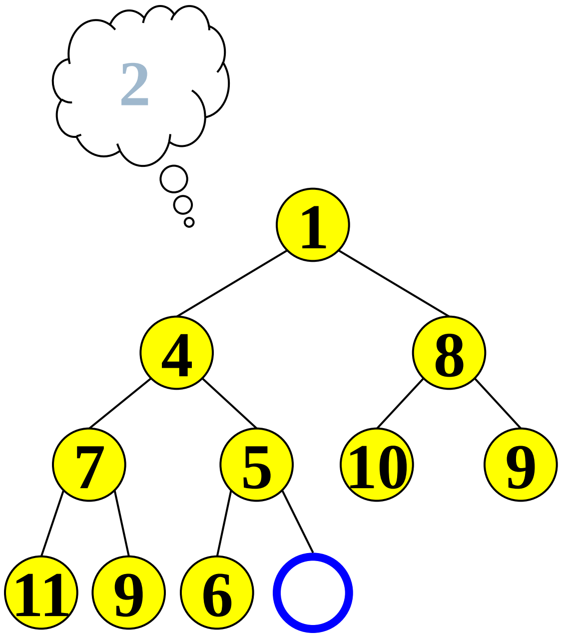
Insert

- Add a value to the tree
- Structure and heap order properties must still be correct afterwards

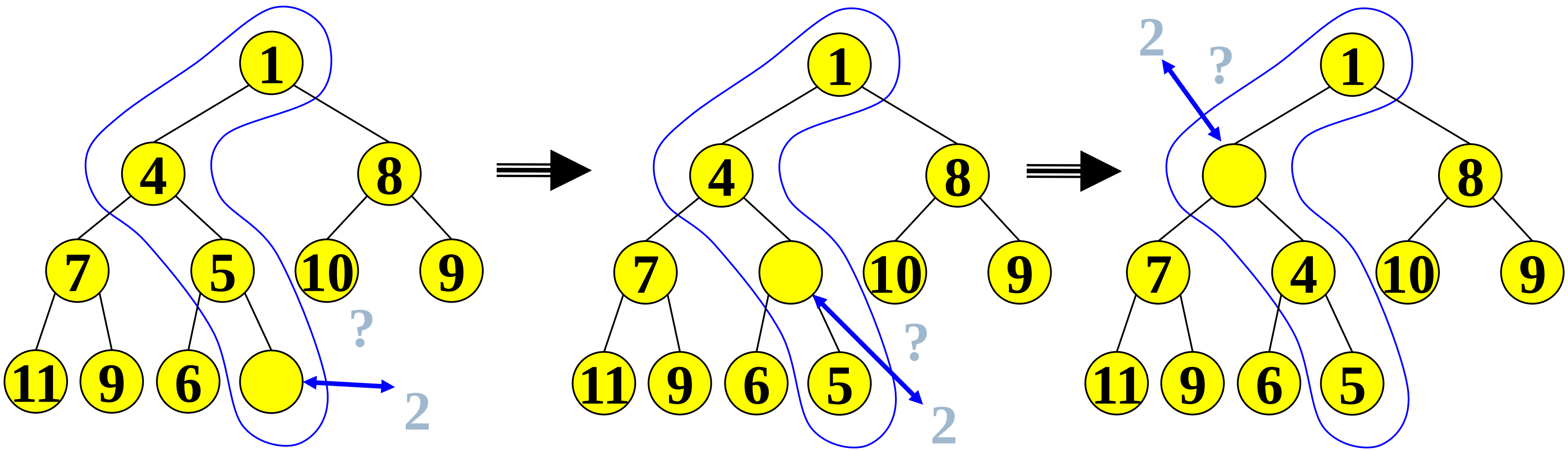


Insert: Maintain the Structure Property

- There is only one valid tree shape after we add one more node
- So put our new data there and then focus on restoring the heap property



Maintain the heap property



Percolate up:

- Put new data in new location
- If parent larger, swap with parent, and continue
- Done if parent ≤ item or reached root
- Run time?

At the end, how do we know 2 is going to be less than its left child (here, 7) which it wasn't compared against?

Insert: Run Time Analysis

- Like **`deleteMin`**, worst-case time proportional to tree height
  - *O*(**`log`** *n*)
- But… **`deleteMin`** needs the "last used" complete-tree position and **`insert`** needs the "next to use" complete-tree position
  - If "keep a reference to there" then **`insert`** and **`deleteMin`** have to adjust that reference: *O*(**`log`** *n*) in worst case
  - Could calculate how to find it in *O*(**`log`** *n*) from the root given the size of the heap
    - But it's not easy
    - And then **`insert`** is always *O*(**`log`** *n*); what about the promised *O*(1) on average (assuming random arrival of items)?
- There's a "trick": don't represent complete trees as nodes with pointers to children