Algorithms & Data Structures

Lesson 8: Priority Queues

Marc Gaetano

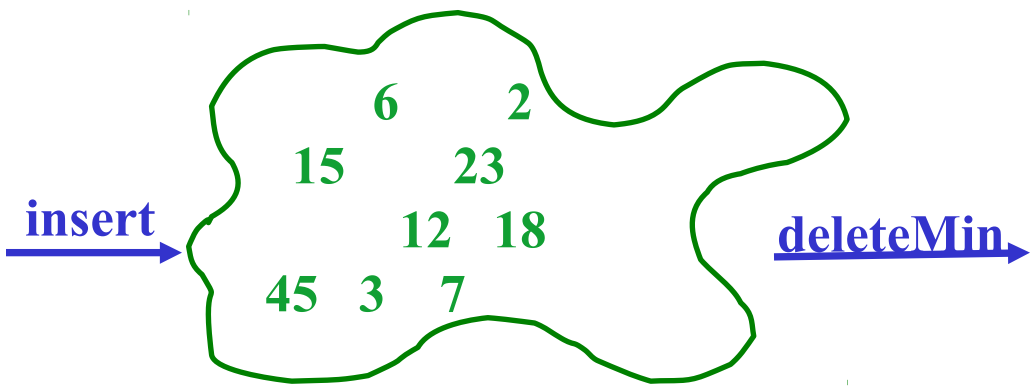
Edition 2017-2018

A new ADT: Priority Queue

- A **priority queue** holds *compare-able data*
  - Like dictionaries, we need to *compare items*
    - Given *x* and *y*, is *x* less than, equal to, or greater than *y*
    - Meaning of the ordering can depend on your data
  - Integers are comparable, so will use them in examples
    - But the priority queue ADT is much more general
    - Typically two fields, the *priority* and the *data*

Priorities

- Each item has a "priority"
  - In our examples, the *lesser* item is the one with the *greater* priority
  - So "priority 1" is more important than "priority 4"
  - (Just a convention, think "first is best")
- Operations:
  - **`insert`**
  - **`deleteMin`**
  - **`is_empty`**



- Key property: **`deleteMin`** *returns* and *deletes* the item with greatest priority (lowest priority value)

Example

`insert` *x1* with priority *5*

`insert` *x2* with priority *3*

`insert` *x3* with priority *4*

a = **`deleteMin // x2`**

b = **`deleteMin // x3`**

`insert` *x4* with priority *2*

`insert` *x5* with priority *6*

c = **`deleteMin // x4`**

d = **`deleteMin // x1`**

- 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

- Sometimes blatant, sometimes less obvious

- 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
- Sort (first **`insert`** all, then repeatedly **`deleteMin`**)

Finding 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; 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	$O(n)$
AVL tree	put in right place	$O(\log n)$	leftmost	$O(\log n)$

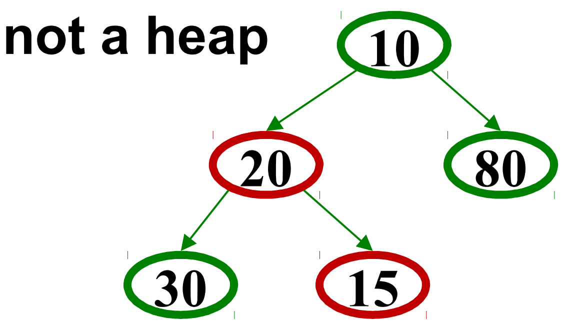
More on possibilities

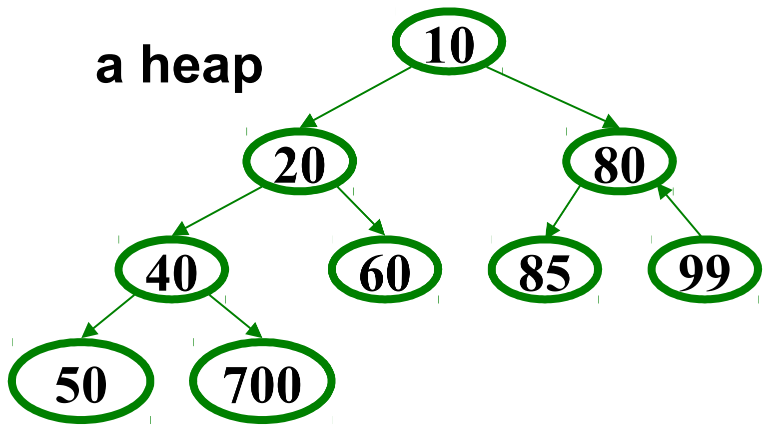
- One more idea: if priorities are 0, 1, …, *k* can use an array of k lists
  - **`insert`**: add to front of list at **`arr[priority]`**, *O*(1)
  - **`deleteMin`**: remove from lowest non-empty list *O(k)*
- We are about to see a data structure called a "binary heap"
  - Another binary tree structure with specific properties
  - *O*(**`log`** *n*) **`insert`** and *O*(**`log`** *n*) **`deleteMin`** *worst-case*
    - Possible because we don't support unneeded operations; no need to maintain a full sort
  - *Very* good constant factors
  - *If* items arrive in random order, then **`insert`** is *O*(1) on *average*
    - Because 75% of nodes in bottom two rows

Our data structure

A *binary min-heap* (or just *binary heap* or just *heap*) has:

- Structure property: A *complete* binary tree
- Heap property: The priority of every (non-root) node is less important than the priority of its parent
  - ***Not** a binary search tree*



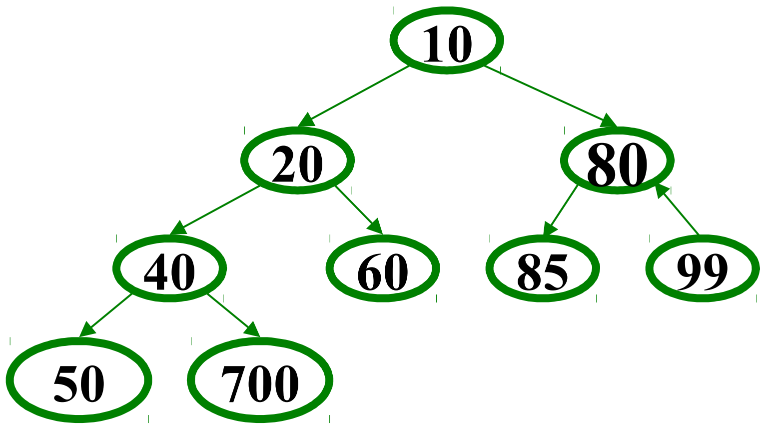


So:

- Where is the highest-priority item?
- What is the height of a heap with *n* items?

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

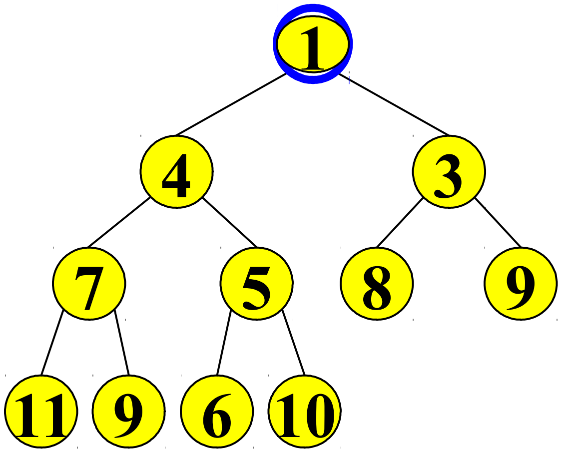


Overall strategy:

- *Preserve structure property*
- *Break and restore heap property*

DeleteMin

Delete (and later return) value at root node

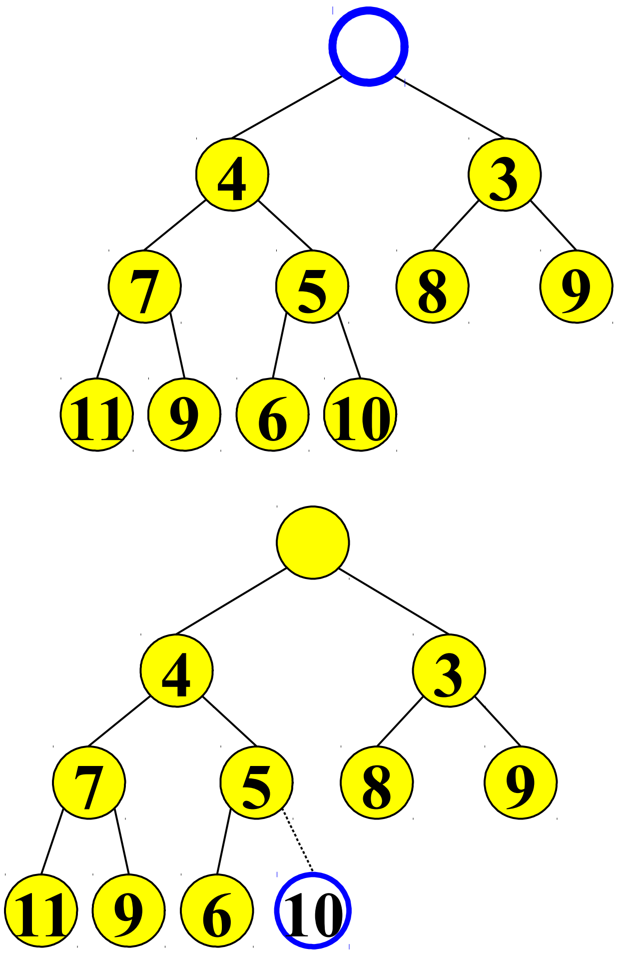


DeleteMin: Keep the Structure Property

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

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

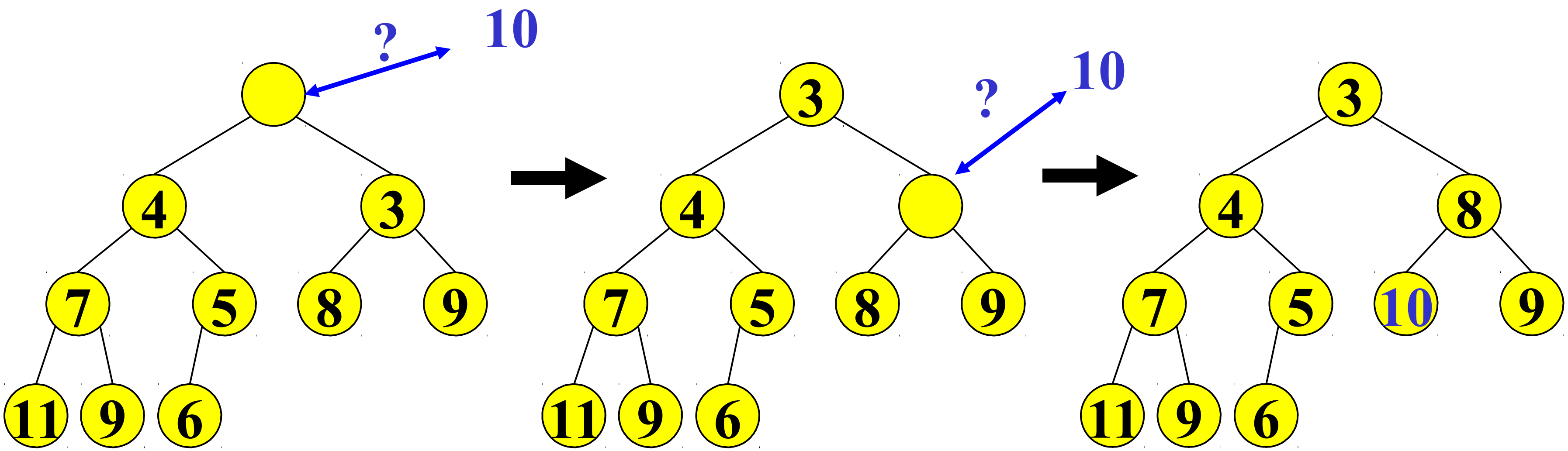
- Pick the last node on the bottom row of the tree and move it to the "hole"



DeleteMin: Restore the Heap Property

Percolate down:

- Keep comparing priority of item with both children
- If priority is less important, swap with the most important child and go down one level
- Done if both children are less important than the item or we've reached a leaf node



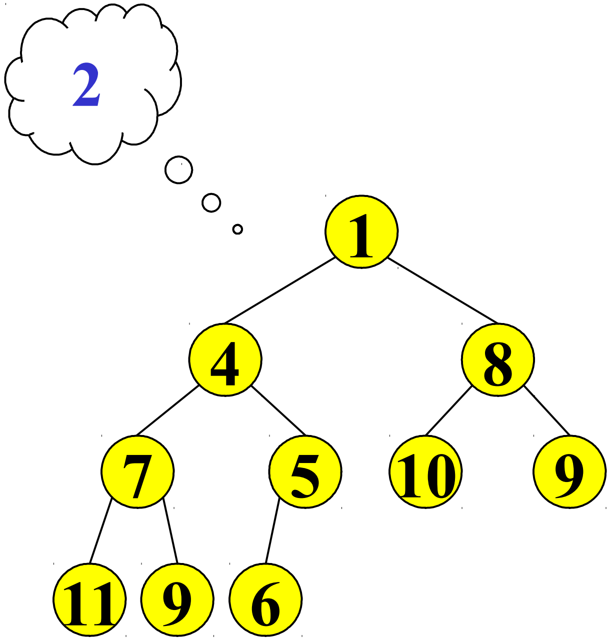
Why is this correct?
What is the run time?

DeleteMin: Run Time Analysis

- Run time is *O*(height of heap)
- A heap is a complete binary tree
- Height of a complete binary tree of *n* nodes?
  - height = $\lfloor \mathtt{log}_2(n) \rfloor$
- Run time of **`deleteMin`** is $O(\mathtt{log}\ n)$

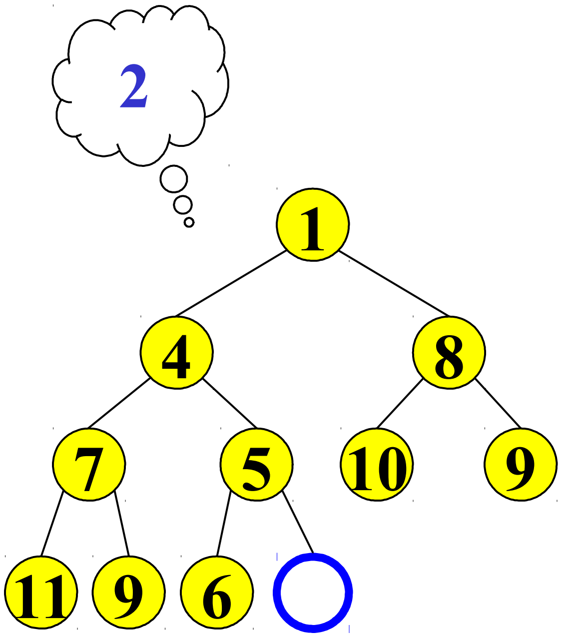
Insert

- Add a value to the tree
- Afterwards, structure and heap properties must still be correct



Insert: Maintain the Structure Property

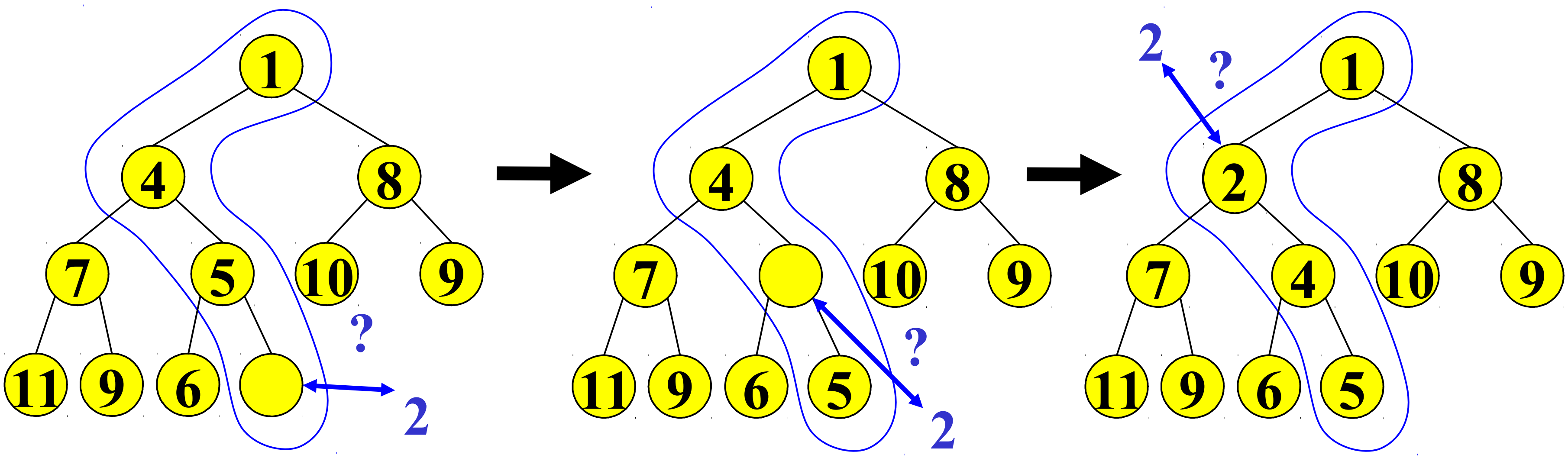
- There is only one valid tree shape after we add one more node
- So put our new data there and then focus on restoring the heap property



Insert: Restore the heap property

Percolate up:

- Put new data in new location
- If parent is less important, swap with parent, and continue
- Done if parent is more important than item or reached root

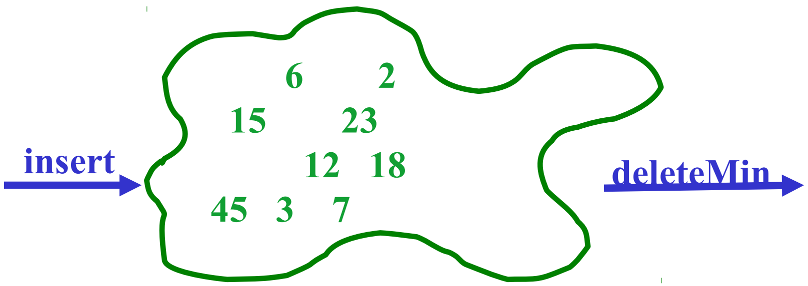


What is the running time?

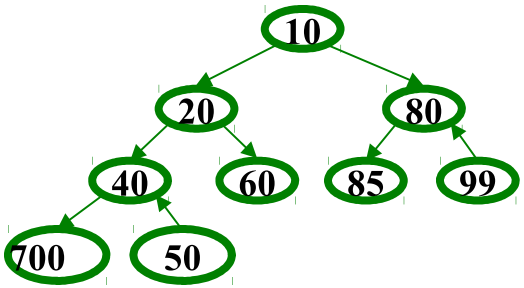
Like `deleteMin`, worst-case time proportional to tree height: $O(\log n)$

Summary

- Priority Queue ADT:
  - **`insert`** comparable object,
  - **`deleteMin`**



- Binary heap data structure:
  - Complete binary tree
  - Each node has less important priority value than its parent



- **`insert`** and **`deleteMin`** operations = O(height-of-tree)=O(**`log`** n)
  - **`insert`**: put at new last position in tree and percolate-up
  - **`deleteMin`**: remove root, put last element at root and percolate-down