CSE332: Data Abstractions

Lecture 4: Priority Queues; Heaps

James Fogarty

Winter 2012

Administrative

- Eclipse Resources
- HW 1 Due Friday
  - Discussion board post regarding HW 1 Problem 2
- Project 1A Milestone and Grading
  - Inquiry about due date timing
  - Use of private nested classes
  - Private helper for array resize
- Testing Script Posted in Forum
  - By Atanas w/ correction by Jackson
  - If you use this, be sure you understand and acknowledge it

Administrative

- Office Hours
  - Will keep calendar updated

- Readings
  - Will keep calendar updated
  - Weiss Chapter 6 to 6.5

New ADT: Priority Queue

- A priority queue holds compare-able data

- Unlike LIFO stacks and FIFO queues, needs 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
  - Many data structures will require this: dictionaries, sorting

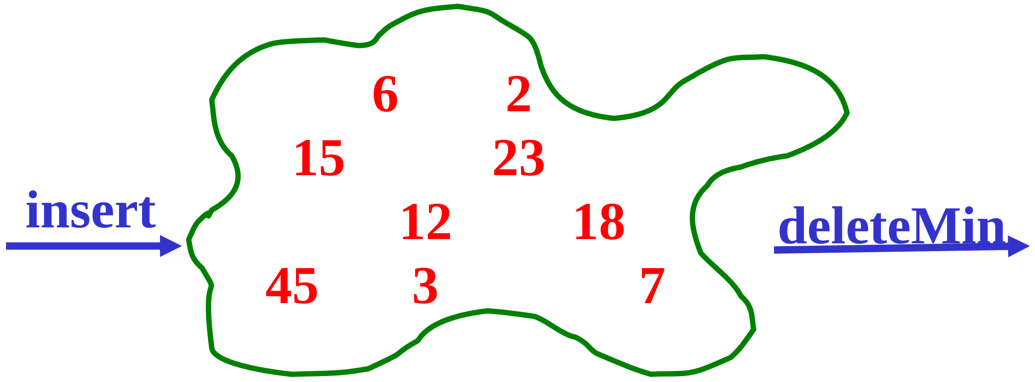
- Integers are comparable, so will use them in examples

- The priority queue ADT is much more general
  - Typically two fields, the *priority* and the *data*

New ADT: Priority Queue

- Each item has a “priority”
  - The *next* or *best* item is the one with the *lowest* priority
  - So “priority 1” should come before “priority 4”
  - Simply by convention, could also do maximum priority

- Operations:
  - **`insert`**
  - **`deleteMin`**



- **`deleteMin`** *returns* and *deletes* item with lowest priority
  - Can resolve ties arbitrarily

Priority Queue

`insert` *a* with priority *5*
`insert` *b* with priority *3*
`insert` *c* with priority *4*
w = **`deleteMin`**
x = **`deleteMin`**
`insert` *d* with priority *2*
`insert` *e* with priority *6*
y = **`deleteMin`**
z = **`deleteMin`**

`after execution:`

w = *b*
x = *c*
y = *d*
z = *a*

Applications

- Priority queue is a major and common ADT
  – Sometimes blatant, sometimes less obvious
- Forward network packets in order of urgency
- Execute work tasks in order of priority
  – “critical” before “interactive” before “compute-intensive” tasks
  – allocating idle tasks in cloud hosting environments
- Sort (first *insert* all items, then *deleteMin* all items)
  – Similar to Project 1’s use of a stack to implement reverse

Advanced Applications

- “Greedy” algorithms
  - Efficiently track what is “best” to try next
- Discrete event simulation (e.g., virtual worlds, system simulation)
  - Every 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, exhaustively checking for events
  - Better:
    - Pending events in a priority queue (priority = event time)
    - Repeatedly: **`deleteMin`** and then **`insert`** new events
    - Effectively “set clock ahead to next event”

Finding a Good Data Structure

- We will examine an efficient, non-obvious data structure
  - But let's first 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	$O(n)$

Our Data Structure: Heap

- We are about to see a data structure called a “heap”
  - Worst-case $O($**`log`** $n)$ **`insert`** and $O($**`log`** $n)$ **`deleteMin`**
  - Average-case $O(1)$ **`insert`** (if items arrive in random order)
  - Very good constant factors
- Possible because we only pay for the functionality we need
  - Need something better than scanning unsorted items
  - But do not need to maintain a full sort
- The heap is a tree, so we need to review some terminology

Tree Terminology

***root*(T):**

***leaves*(T):**

***children*(B):**

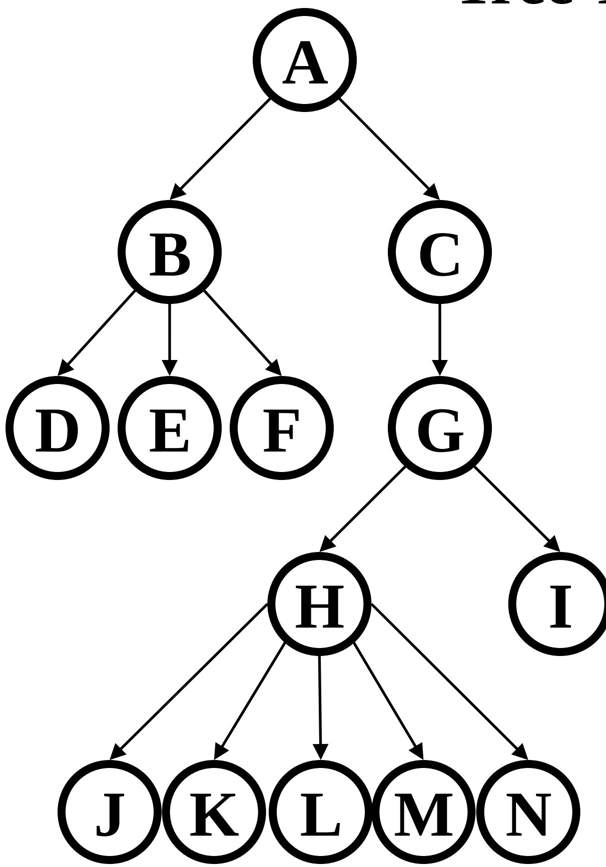
***parent*(H):**

***siblings*(E):**

***ancestors*(F):**

***descendents*(G):**

***subtree*(C):**



Tree Terminology

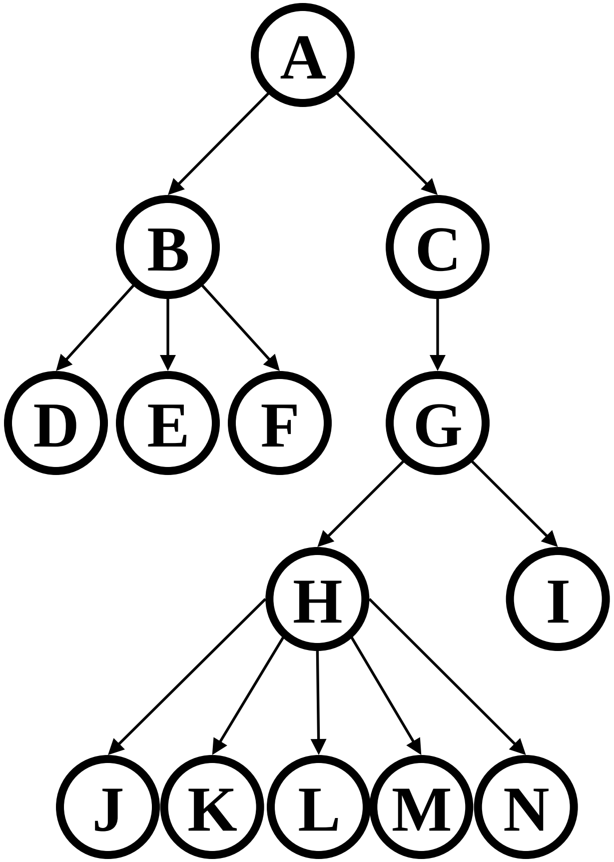
***depth*(B):**

***height*(G):**

***height*(T):**

***degree*(B):**

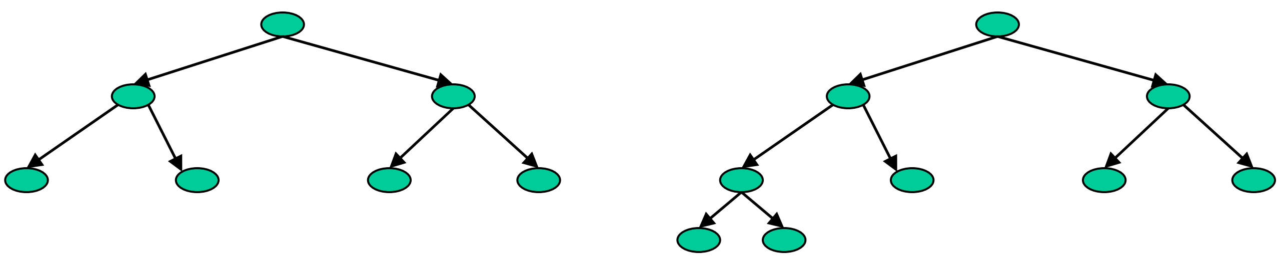
***branching factor*(T):**



Types of Trees

Certain terms define trees with specific structures

- Binary tree: Every node has at most 2 children
- *n*-ary tree: Every node as at most *n* children
- Perfect tree: Every row is completely full
- Complete tree: All rows except the bottom are completely full, and it is filled from left to right



What is the height of a perfect tree with n nodes? A complete tree?

Properties of a Binary Min-Heap

More commonly known as a binary heap or simply a heap

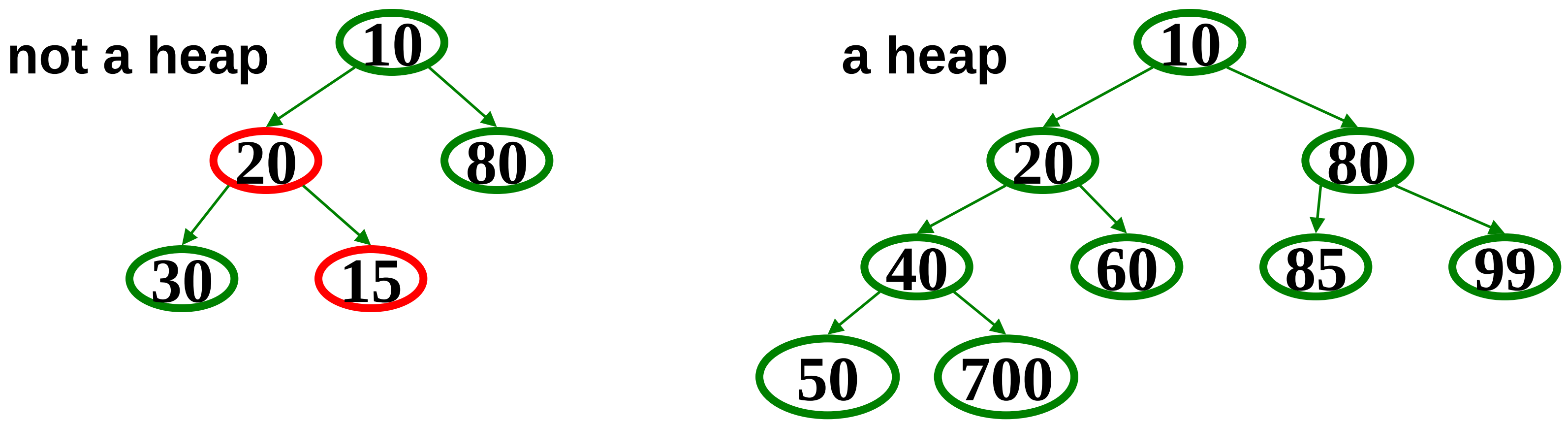
- Structure Property: A complete tree

- Heap Property: The priority of every non-root node is greater than the priority of its parent

How is this different from a binary search tree?

Properties of a Binary Min-Heap

Requires both structure property and the heap property



Where is the minimum priority item?

What is the height of a heap with *n* items?

Basics of Heap Operations

`findMin`:

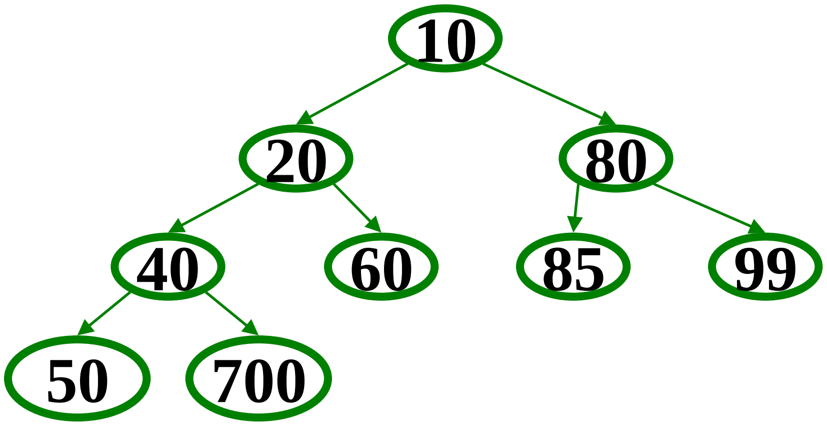
- return **`root.data`**

`deleteMin`:

- Move last node up to root
- Violates heap property, "Percolate Down" to restore

`insert`:

- Add node after last position
- Violate heap property, "Percolate Up" to restore

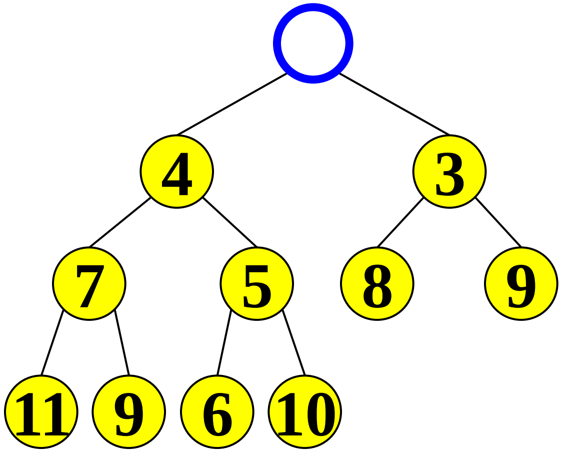


Overall, the strategy is:

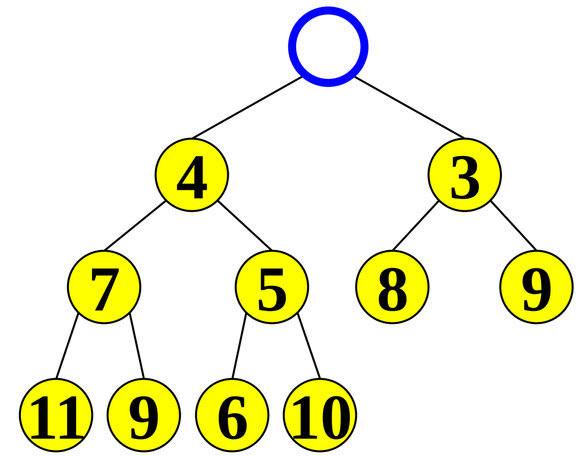
- Preserve structure property
- Break and restore heap property

DeleteMin Implementation

1. Delete value at root node
   (and store it for later return)



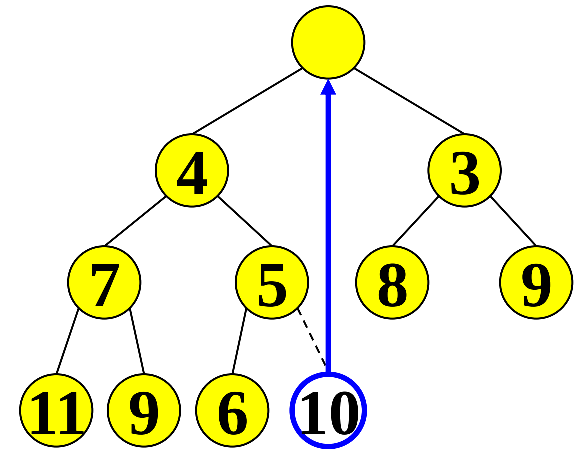
Restoring the Structure Property



2. We now have a "hole" at the root

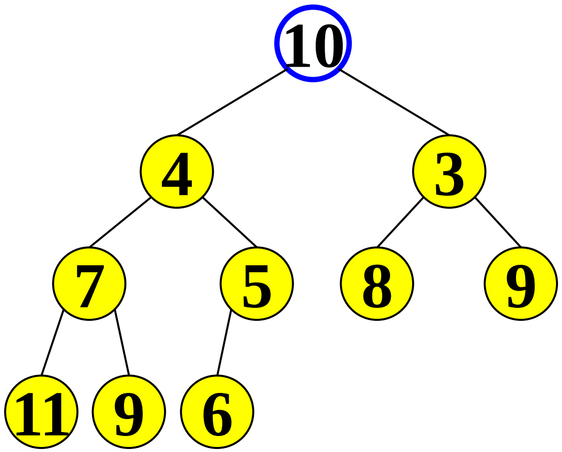
3. We must "fill" the hole with another value, must have a tree with one less node, and it must still be a complete tree

4. The "last" node is the is obvious choice



Restoring the Heap Property

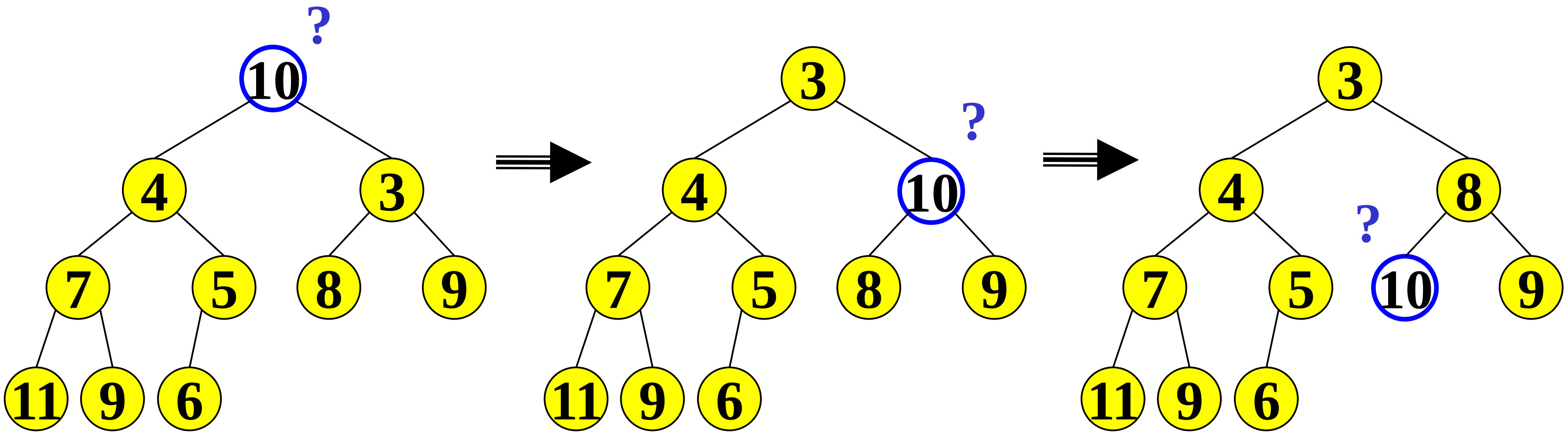
5. Not a heap, it violates the heap property



6. We percolate down to fix the heap

```
While greater than either child
   Swap with smaller child
```

Percolate Down



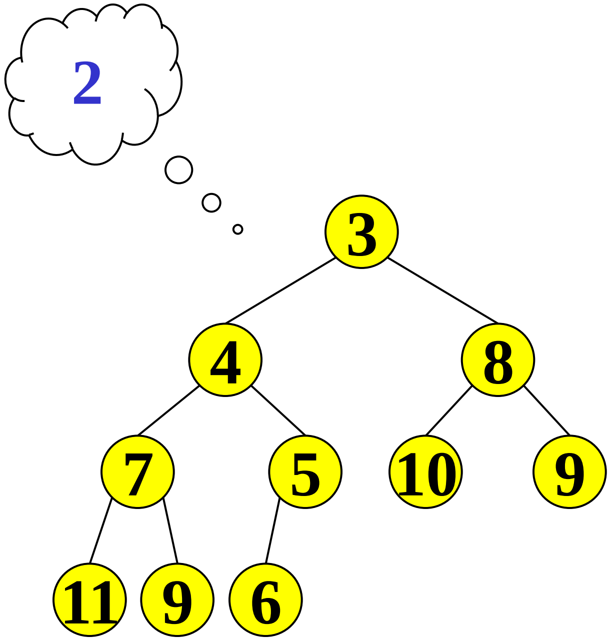

```
While greater than either child
    Swap with smaller child
```

What is the runtime?
O(log n)

Why does this work?
Both children are heaps

Insert Implementation

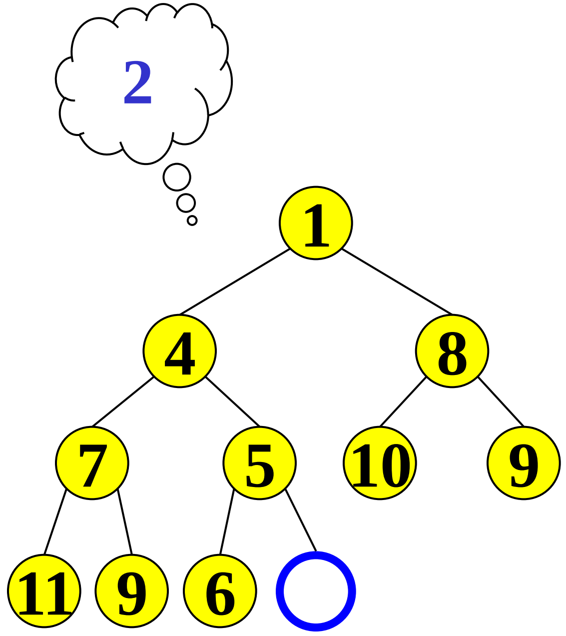
- Add a value to the tree
- Afterwards, structure and heap properties must still be correct



Maintaining the Structure Property

1. There is only one valid shape for our tree after addition of one more node

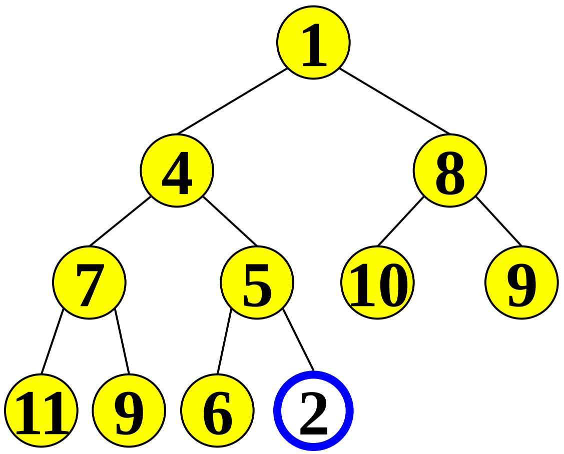
2. Put our new data there



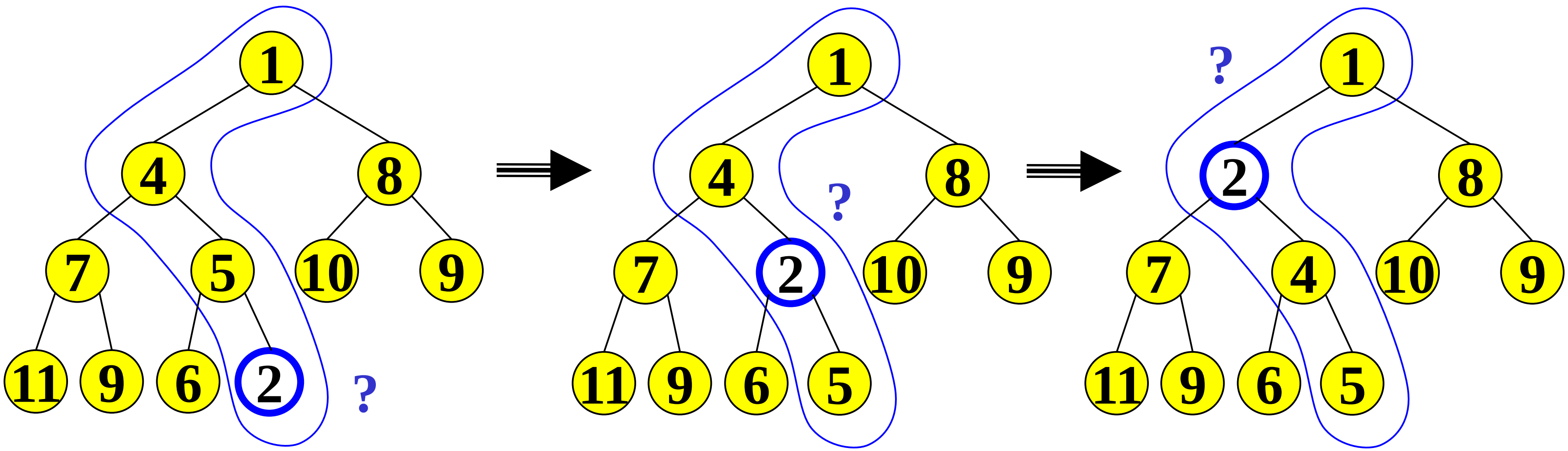
Restoring the Heap Property

3. Then percolate up to fix heap property

```
While less than parent
   Swap with parent
```



Percolate Up




```
While less than parent
    Swap with parent
```

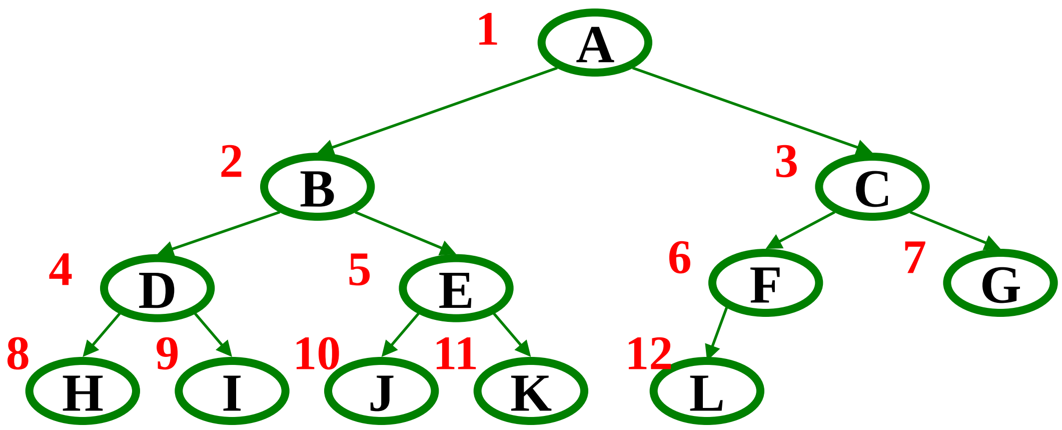
What is the runtime?
O(log n)

Why does this work?
Both children are heaps

A Clever and Important Trick

- We have seen worst-case O(log n) insert and deleteMin
  - But we promised average-case O(1) insert
- Insert requires access to the “next to use” position in the tree
  - Walking the tree requires O(log n) steps
- Remember to only pay for the functionality we need
  - We have said the tree is complete, but have not said why
- All complete trees of size n contain the same edges
  - So why are we even representing the edges?

Array Representation of a Binary Heap



From node **`i`**:

left child: **`i*2`**
right child: **`i*2+1`**
parent: **`i/2`**

wasting index 0 is convenient for the math

Array implementation:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Tradeoffs of the Array Implementation

Advantages:

- Non-data space: only index 0 and any unused space on right
  - Contrast to link representation using one edge per node (except root), a total of n-1 wasted space (like linked lists)
  - Array would waste more space if tree were not complete
- Multiplying and dividing by 2 is extremely fast
- The major one: Last used position is at index **`size`**, O(1) access

Disadvantages:

- Same might-be-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

Advantages outweigh disadvantages: “this is how people do it”