Priority Queues!

With Janette

except all the credit goes to Dan Grossman,

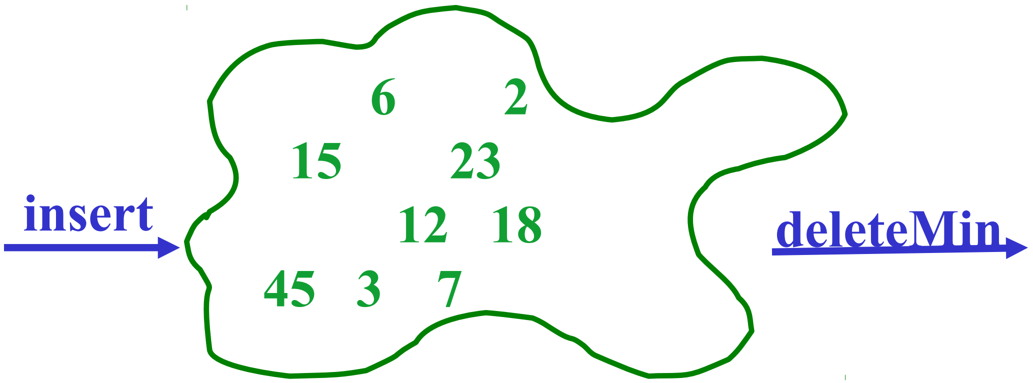
from whom I straight up stole these slides

A new data structure: Priority Queue

- A **priority queue** holds *compare-able data*
  - Unlike stacks and queues, 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
    - Many data structures require this: dictionaries, sorting
  - Integers are comparable, so will use them in examples
    - But the priority queue is much more general
    - Typically two fields, the *priority* and the *data*

Priorities

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



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

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

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

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 (cf. CSE143)
- Sort (first **`insert`** all, then repeatedly **`deleteMin`**)

More applications

- "Greedy" algorithms
- Discrete event simulation (system simulation, virtual worlds, …)
  - Each event *e* happens at some time *t*, updating system state and generating 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 = event time)
    - Repeatedly: **`deleteMin`** and then **`insert`** new events
    - Effectively "set clock ahead to next event"

Finding a good data structure

- Will show an efficient, non-obvious data structure
  - 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		
unsorted linked list		
sorted circular array		
sorted linked list		
binary search tree		

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; 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)*
- We are about to see a data structure called a "binary heap"
  - *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*

Tree terms (review?)

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

***root*(tree)**

***leaves*(tree)**

***children*(node)**

***parent*(node)**

***siblings*(node)**

***ancestors*(node)**

***descendents*(node)**

***subtree*(node)**

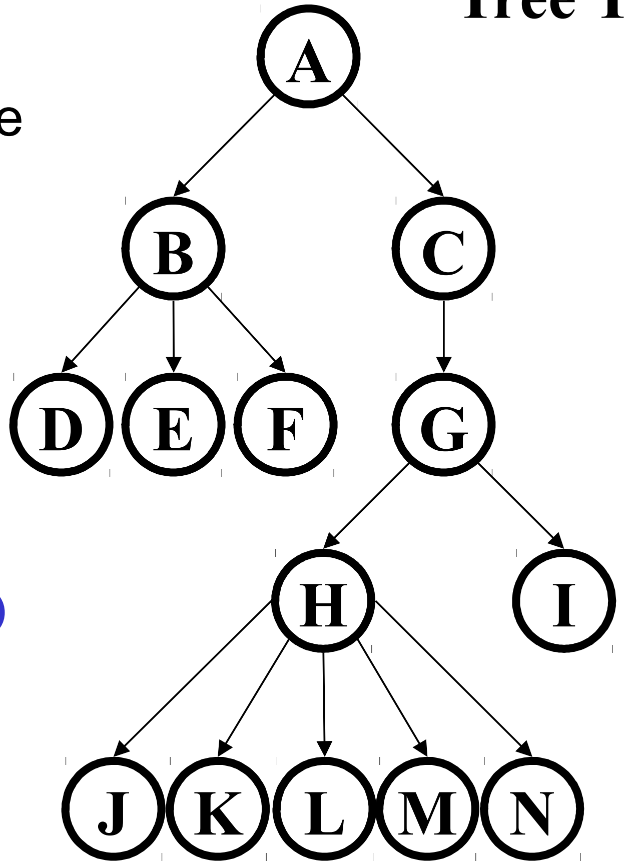
***depth*(node)**

***height*(tree)**

***degree*(node)**

***branching factor*(tree)**

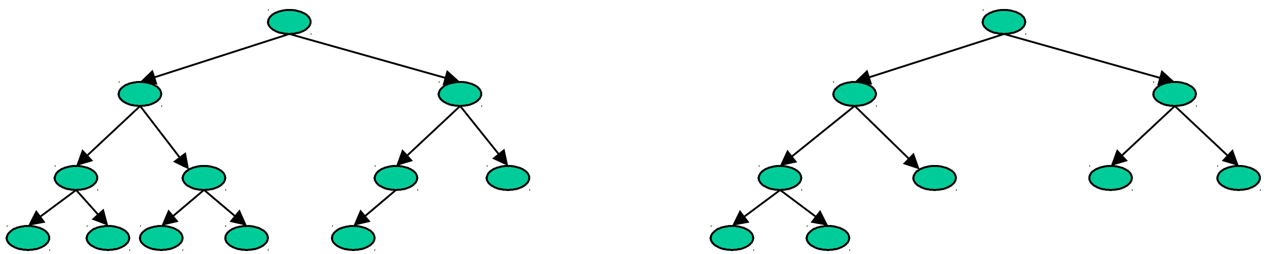
Tree T



Kinds of trees

Certain terms define trees with specific structure

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



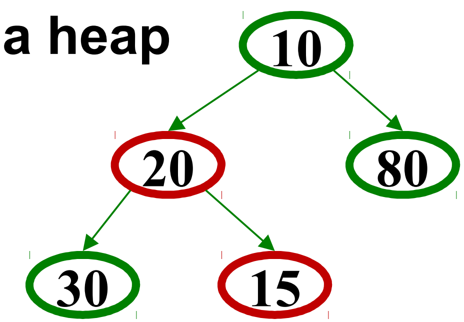
What is the height of a perfect tree with n nodes? A complete tree?

Our data structure

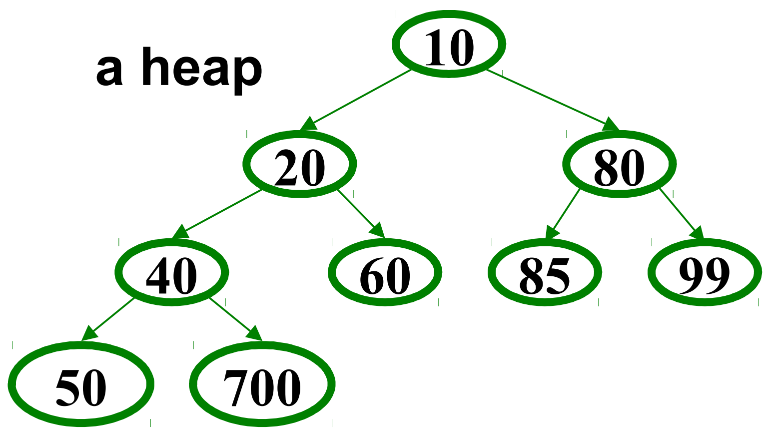
Finally, then, a *binary min-heap* (or just *binary heap* or just *heap*) is:

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

not a heap



a heap

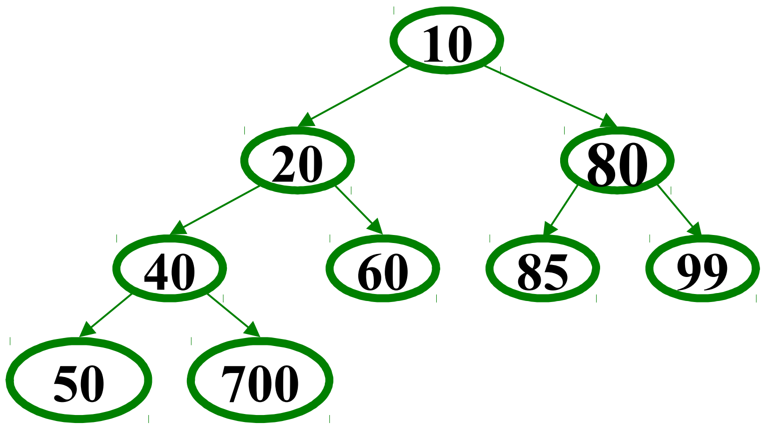


So:

- 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

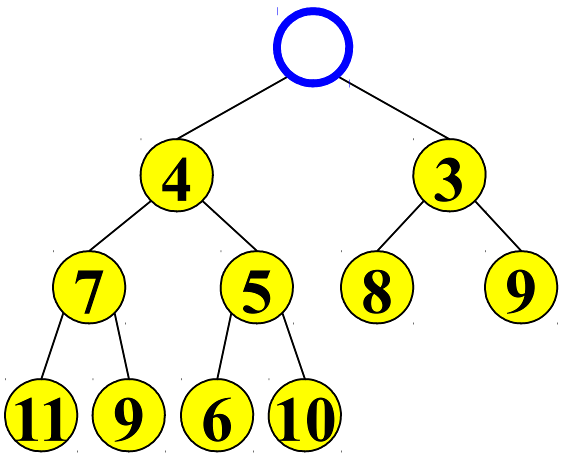


Overall strategy:

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

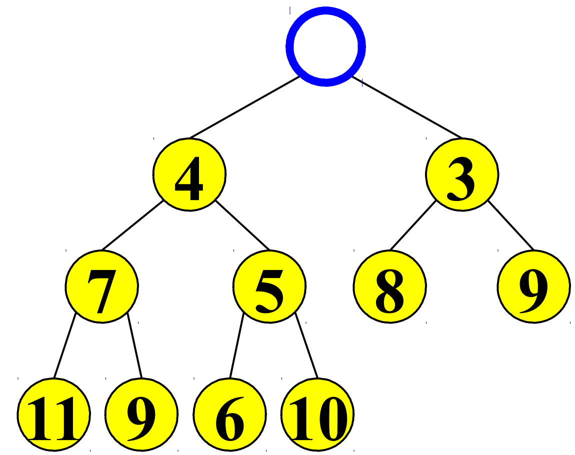
DeleteMin

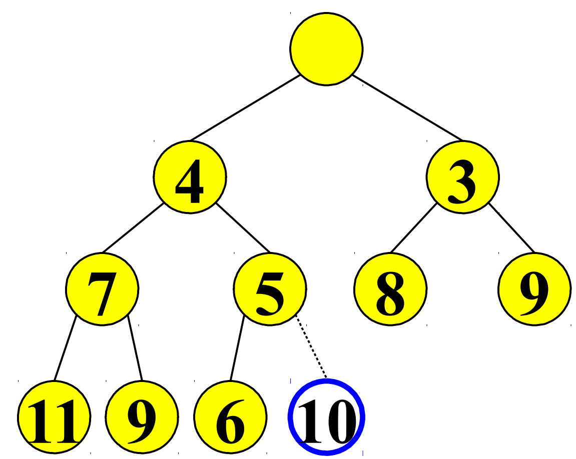
1. Delete (and later return) value at root node



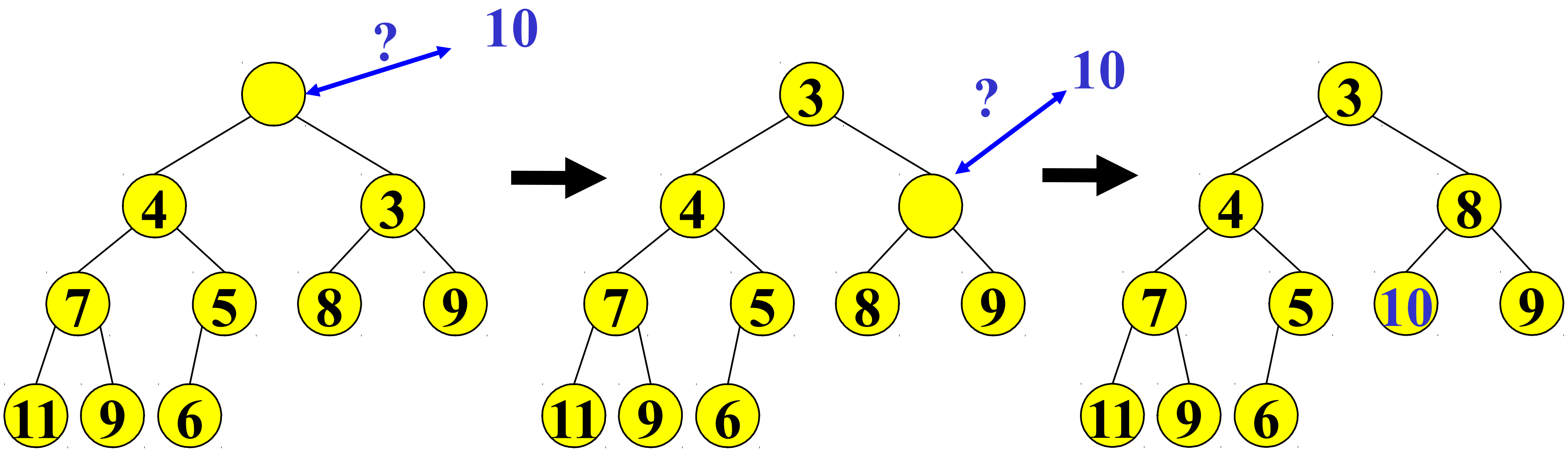
2. Restore the Structure Property

- 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
- Swap with lesser child and go down one level
- Done if both children are ≥ item or reached a leaf node

Why is this correct? What is the run time?

DeleteMin: Run Time Analysis

- Run time is *O*(height of heap)

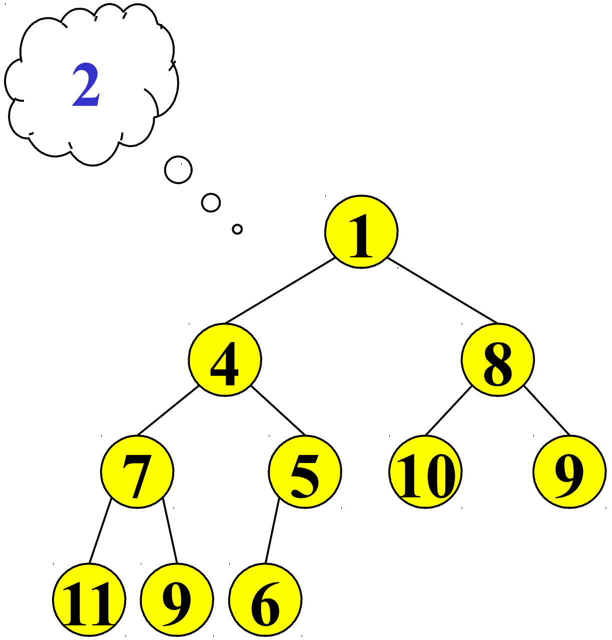
- A heap is a complete binary tree

- Height of a complete binary tree of *n* nodes?
  - height = $\lfloor \log_2(n) \rfloor$

- Run time of **`deleteMin`** is $O(\log n)$

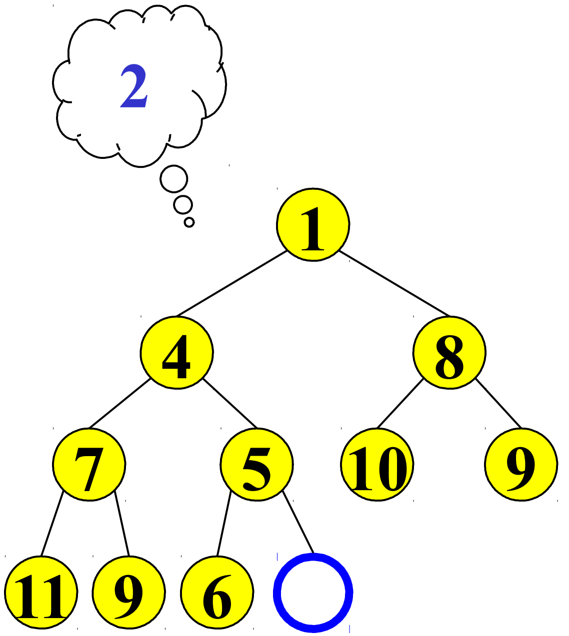
Insert

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

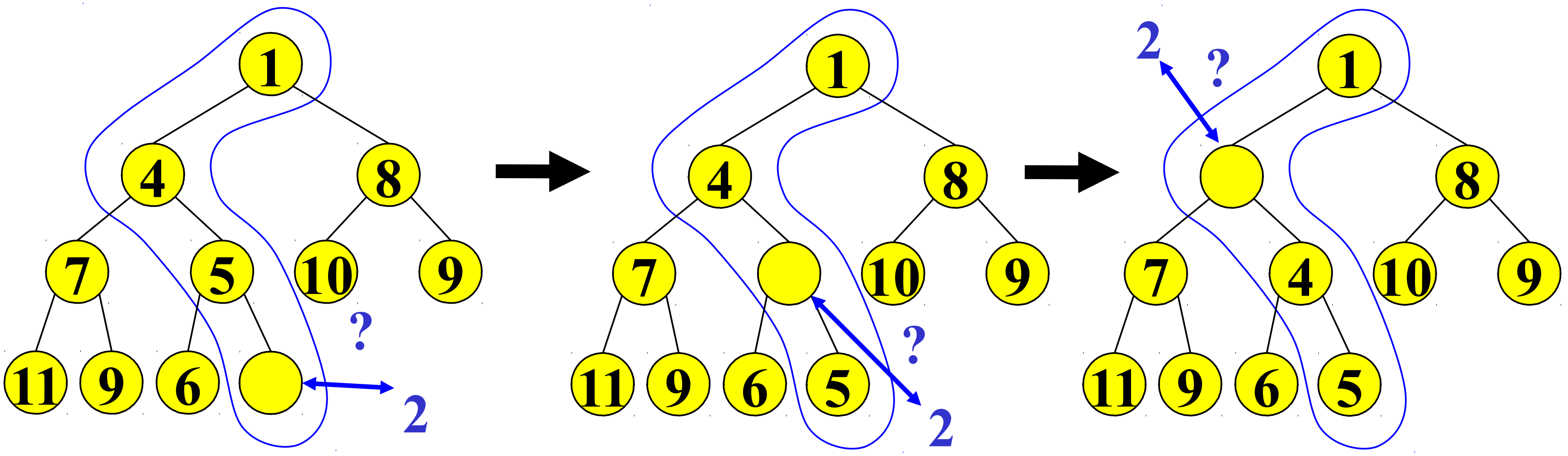


Insert: Maintain the Structure Property

- 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



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

Why is this correct? What is the run time?

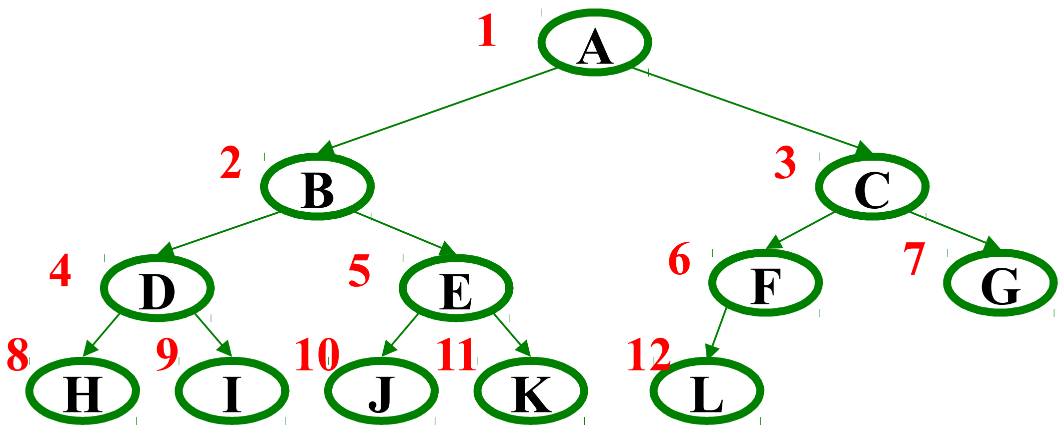
Insert: Run Time Analysis

- 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*), promised *O*(1) on average (assuming random arrival of items)

- There's a "trick": don't represent complete trees with explicit edges!

Array Representation of Binary Trees



From node **`i`**:

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

(wasting index 0 is convenient for the index arithmetic)

implicit (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

Judging the array implementation

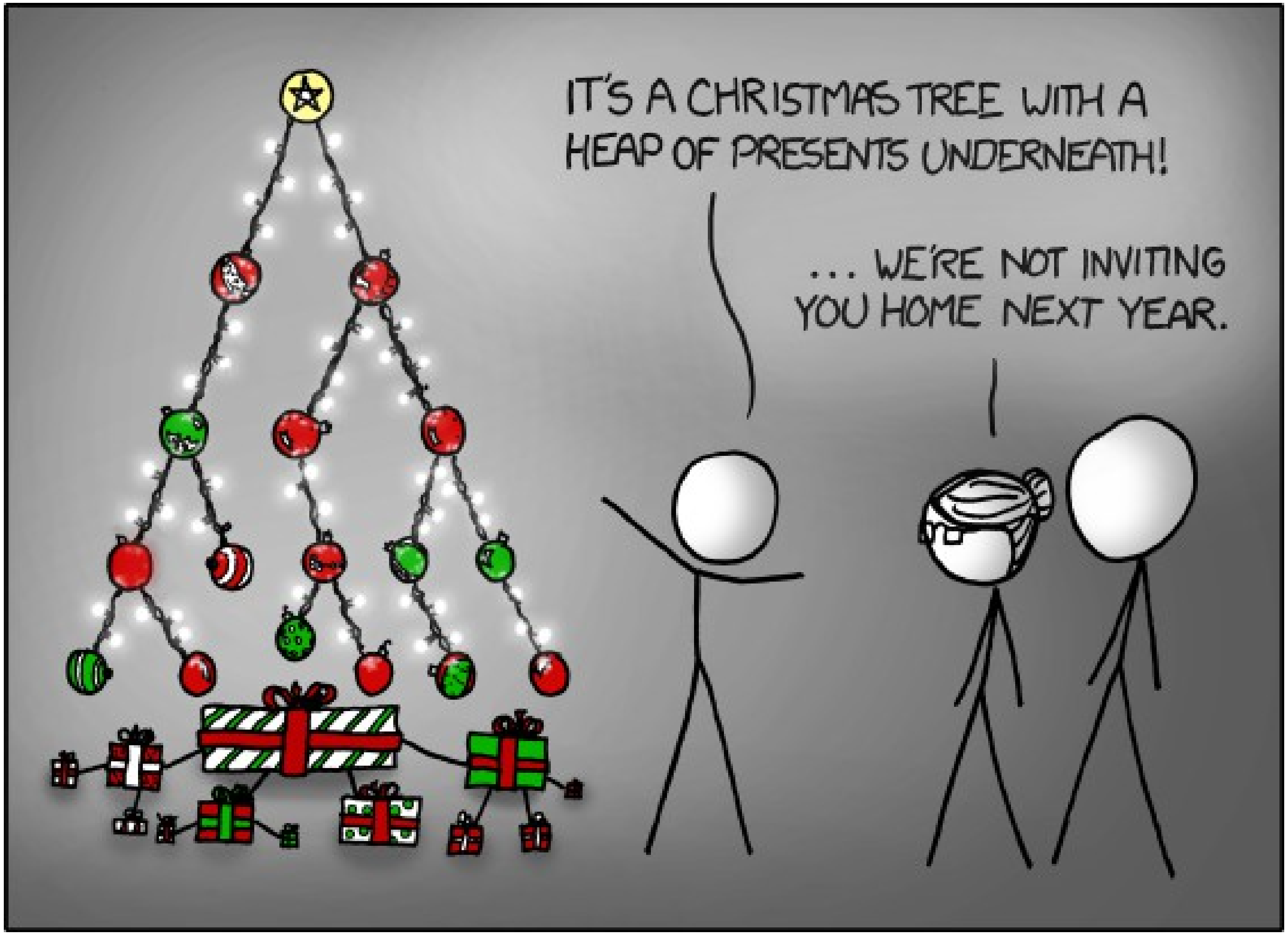
Plusses:

- Non-data space: just index 0 and unused space on right
  - In conventional tree representation, one edge per node (except for root), so *n*-1 wasted space (like linked lists)
  - Array would waste more space if tree were not complete
- For reasons you learn in CSE351, multiplying and dividing by 2 is very fast
- Last used position is just index **`size`**

Minuses:

- Same might-be-empty (wasted space) or might-get-full (have to resize) problems we saw with ArrayLists

Plusses outweigh minuses: “this is how people do it”



Not only is that terrible in general, but you just KNOW Billy's going to open the root present first, and then everyone will have to wait while the heap is rebuilt.