L12/13 – Heap

7/12/24, 7/15/24

Announcements

- QZ04 tomorrow 7/16 (topics)
- EX07 due tonight
- EX08 releases after class
- This slide presentation in `.pptx` format has animations in slideshow mode (credit: Prof. Stotts)

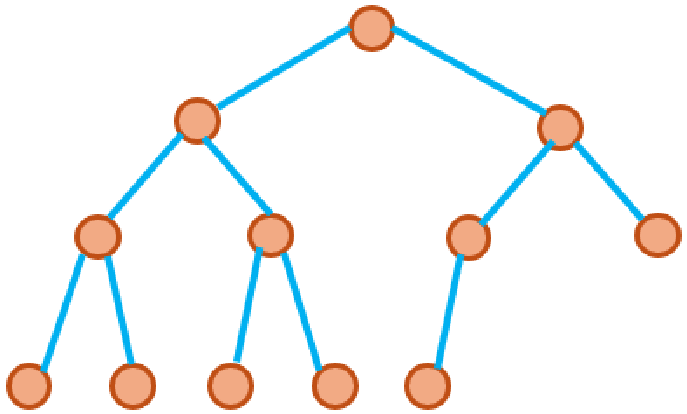
PrQueue slides from L11

- See PrQueue slides from L11
- PrQueue is an ADT we will implement with (binary) heap

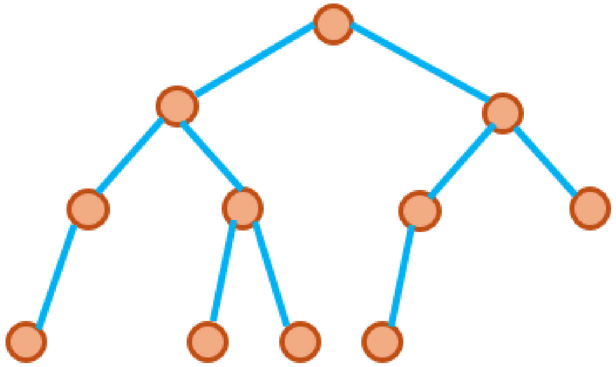
Min binary heap (min-heap)

- Structure property - Complete binary tree
  - All levels fully filled except possibly the last level, which is filled from left to right
- Heap-order property
  - Min element at root
  - Every child ≥ parent
  - Every path from root to leaf is an ordered (nondecreasing, small to large) list
  - Every subtree of heap is also a heap
- Visualization

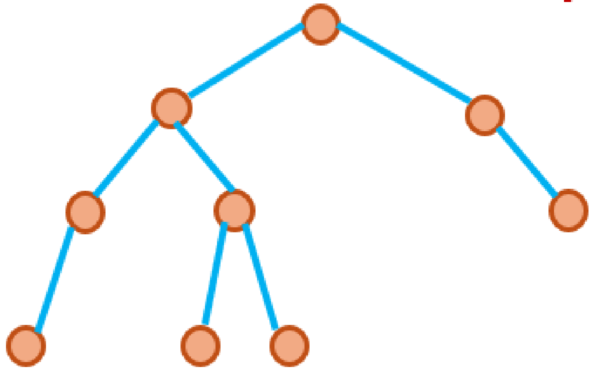
Structure property (complete BT)



Complete binary tree

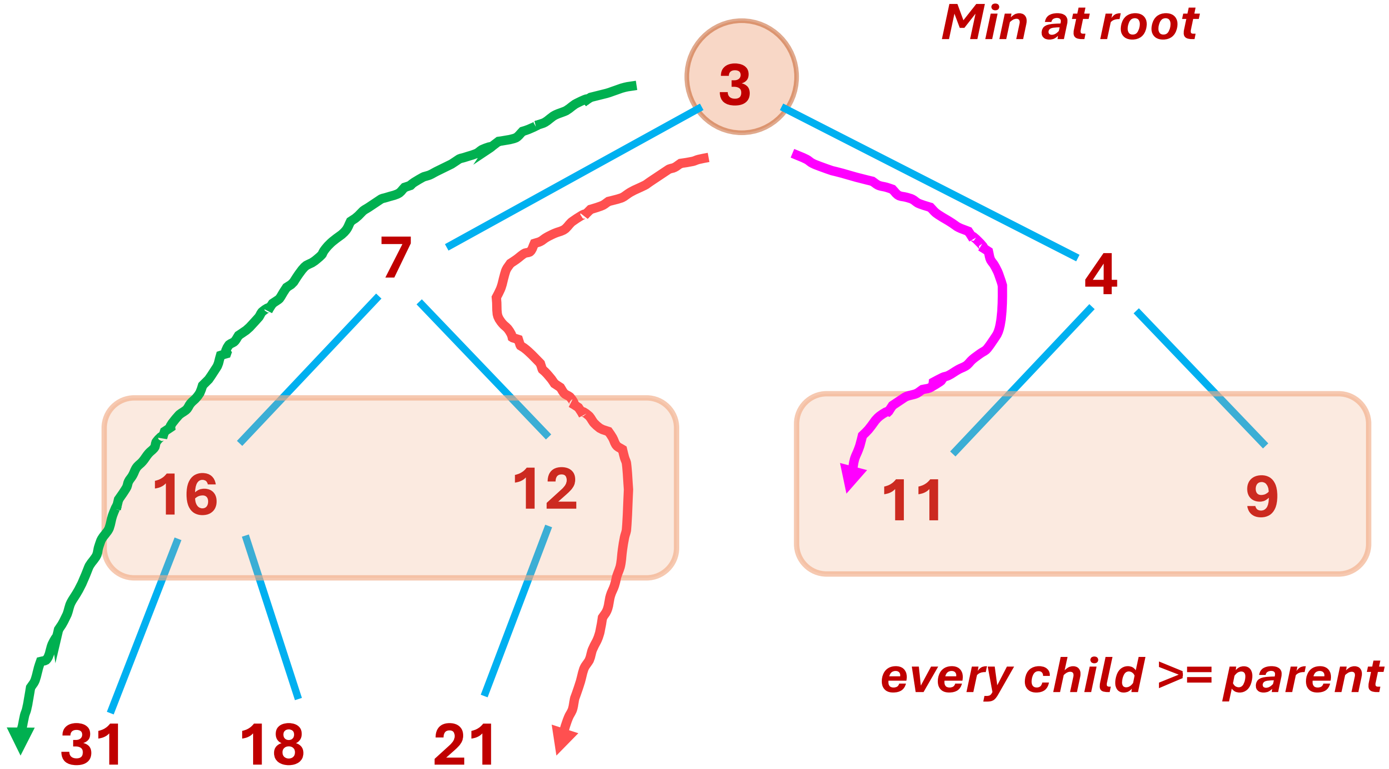


Not complete



Binary Heap

Heap-order property

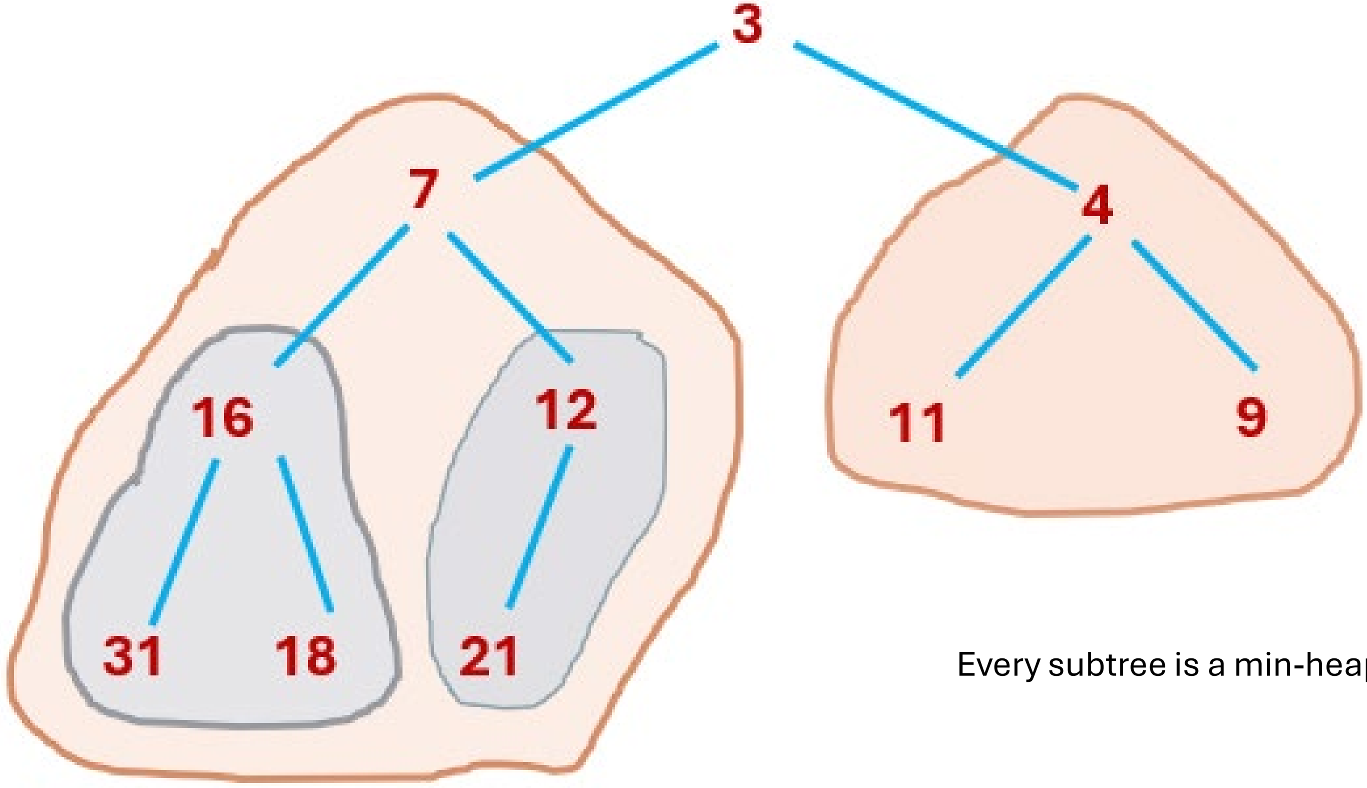


Min at root

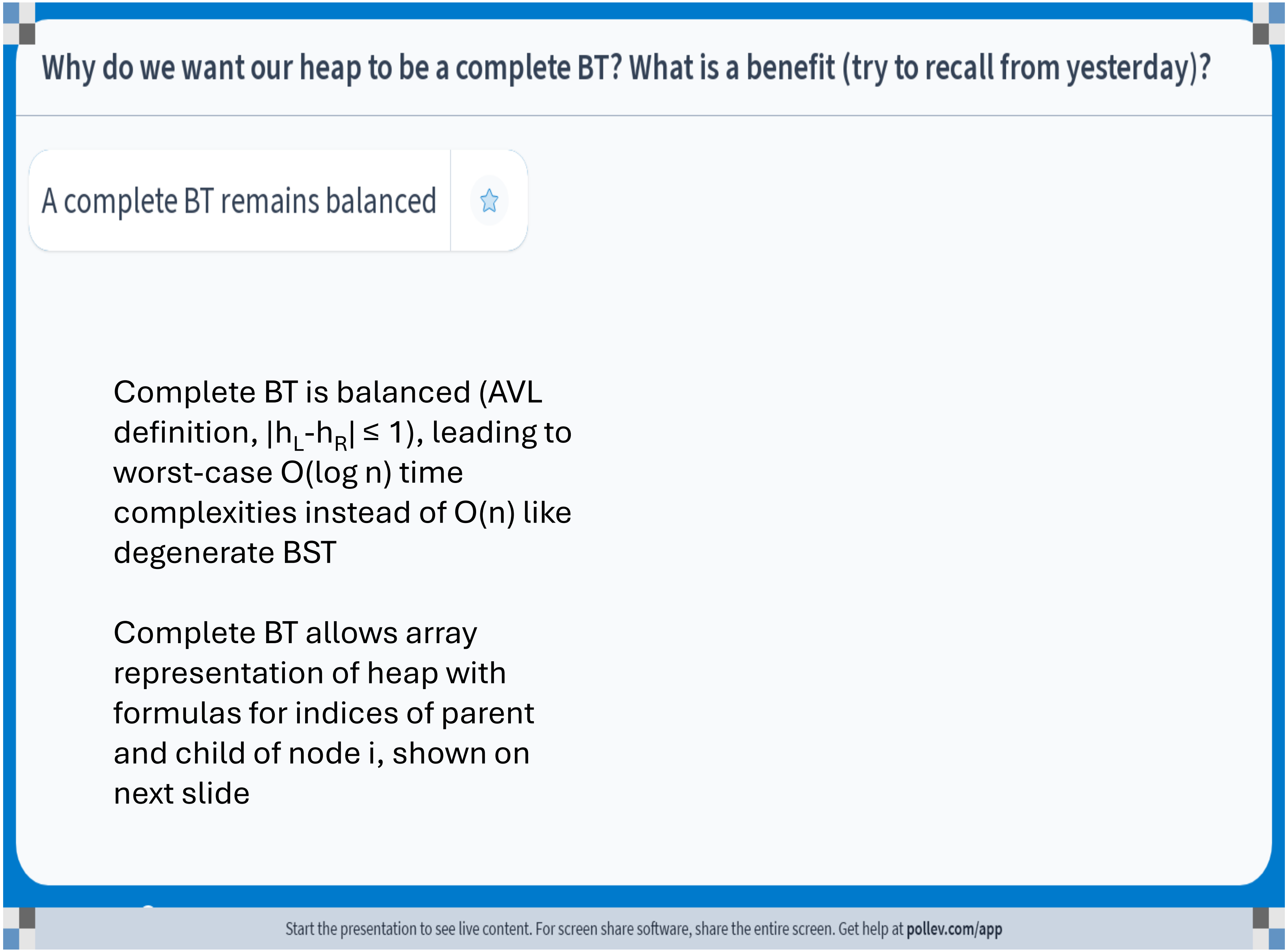
every child >= parent

every path root to leaf is an ordered sequence small to large

Heap-order property



Every subtree is a min-heap



Why do we want our heap to be a complete BT? What is a benefit (try to recall from yesterday)?

A complete BT remains balanced ☆

Complete BT is balanced (AVL definition, $|h_L - h_R| \leq 1$), leading to worst-case O(log n) time complexities instead of O(n) like degenerate BST

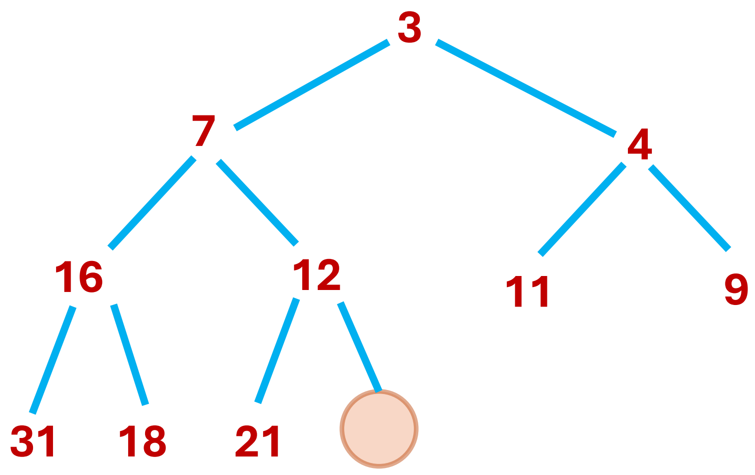
Complete BT allows array representation of heap with formulas for indices of parent and child of node i, shown on next slide

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Implementation

- Another benefit of structure property: allows very efficient representation with an array
  - Leave slot 0 unused (start at 1)
  - Store node value as array element
  - For node in slot i
    - Parent is at $\lfloor i / 2 \rfloor$ (floor, int division)
    - Left child at 2i
    - Right child at 2i+1
- Array representation => Node class not needed (save memory)

Example



To fill array, ***breadth-first*** *across tree*

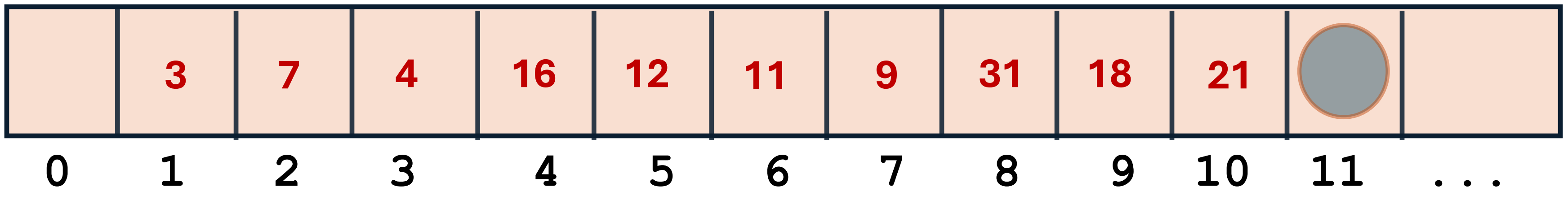
root in slot 1: **3**

then level 1 in slots 2,3: **7, 4**

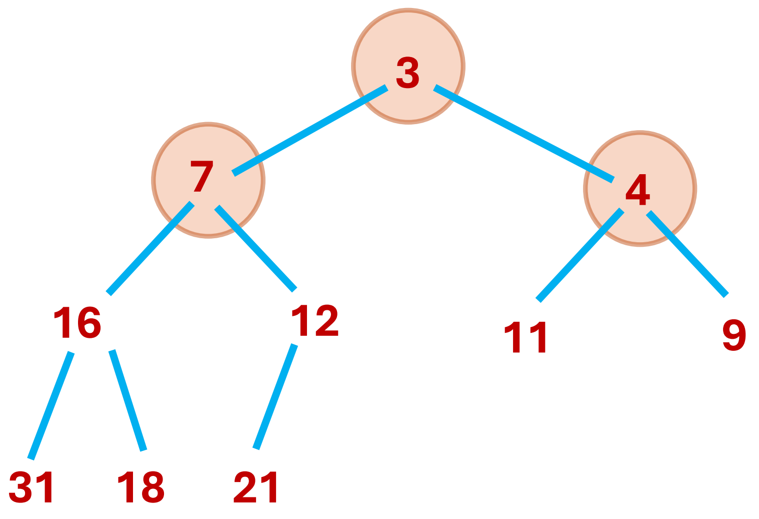
level 2 in slots 4,5,6,7: **16,12,11,9**

next level in slots 8, 9, 10 ...15: **31,18,21**

Next item causes slot 11 to fill



Example



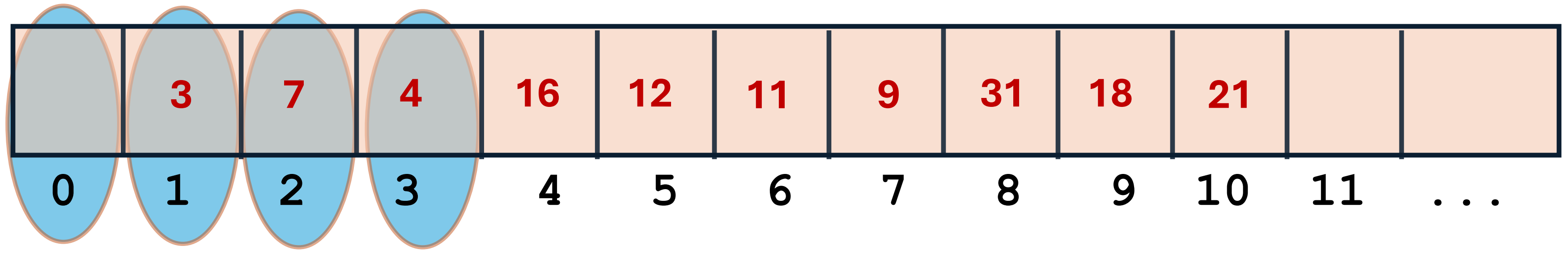
To infer tree structure from array

node in slot 1: **3**

Parent: floor(1/2) is 0 (*root, no parent*)

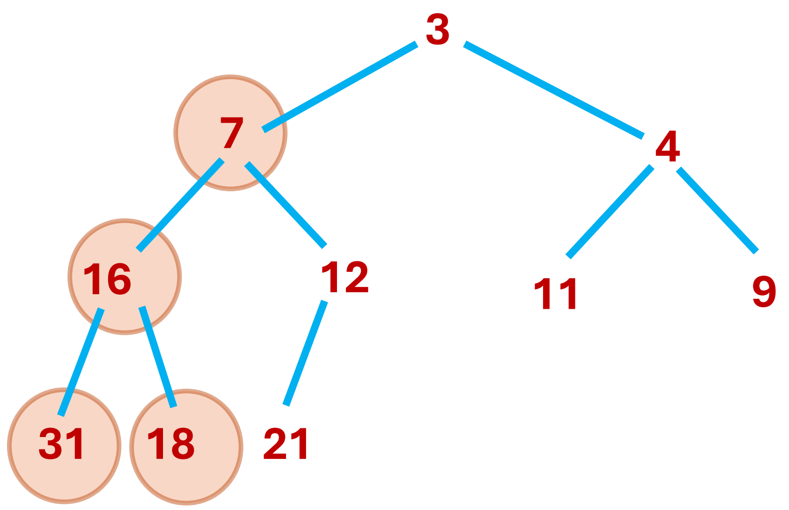
Lchild: 2*1 is 2, slot 2 has **7**

Rchild: (2*1)+1 is 3, slot 3 has **4**



	3	7	4	16	12	11	9	31	18	21		
0	1	2	3	4	5	6	7	8	9	10	11	. . .

Example



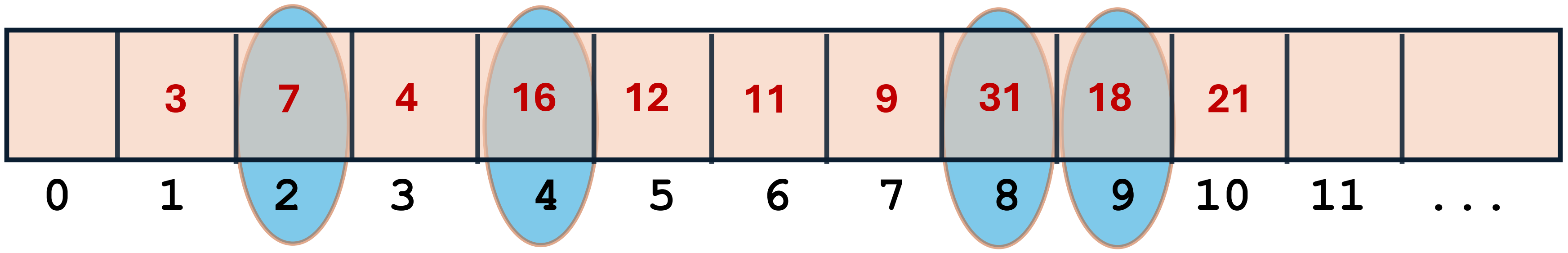
To infer tree structure from array

node in slot 4 is **16**

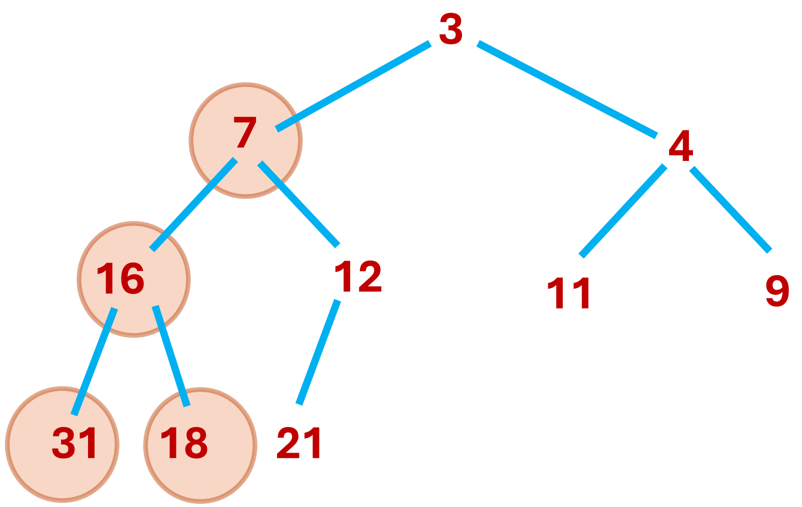
Parent: floor(4/2) is 2, slot 2 has **7**

Lchild: 4*2 is 8, slot 8 has **31**

Rchild: (4*2)+1 is 9, slot 9 has **18**



Example

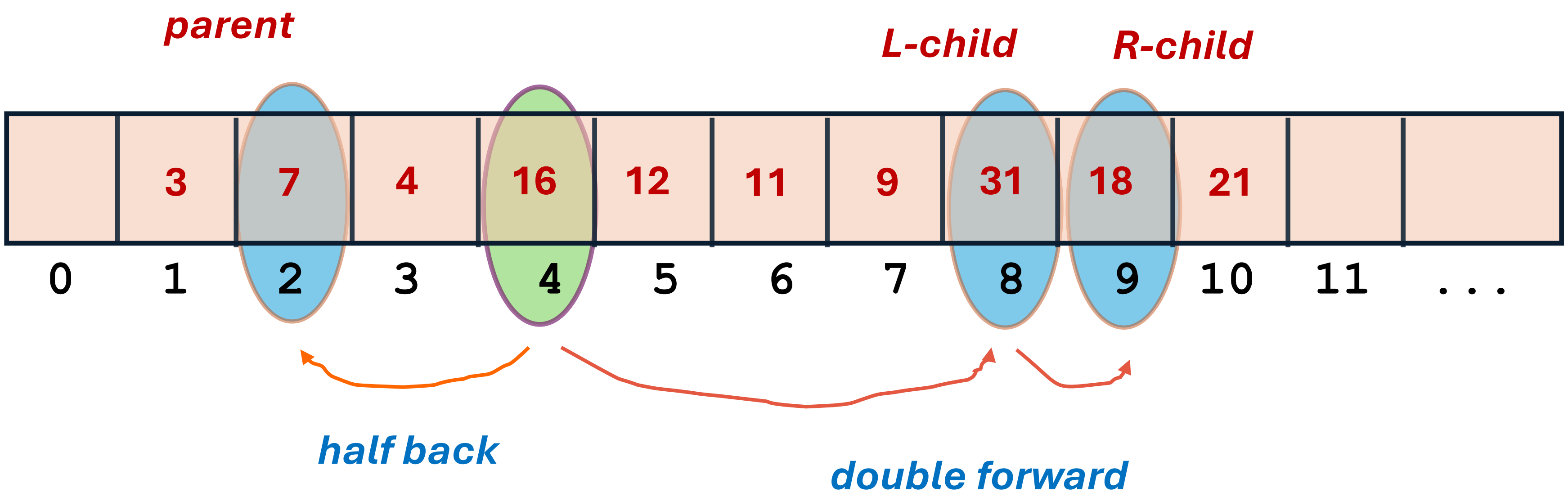


node in slot 4 is **16**

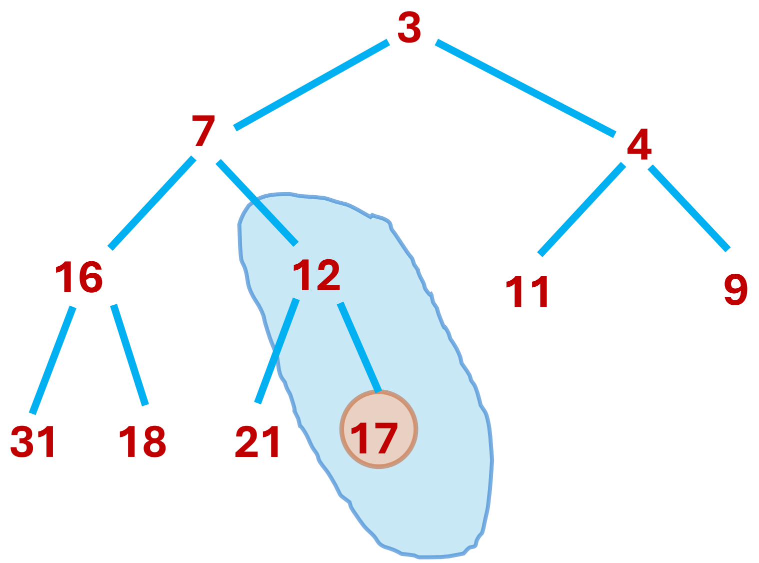
Parent: floor(4/2) is 2, slot 2 has **7**

Lchild: 4*2 is 8, slot 8 has **31**

Rchild: (4*2)+1 is 9, slot 9 has **18**



Insert



Next item causes slot 11 to fill

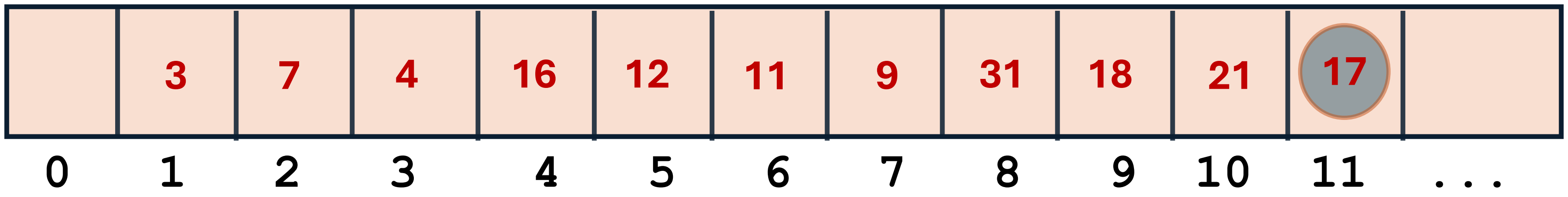
this maintains structure property

insert (17)

still has heap-order property ?

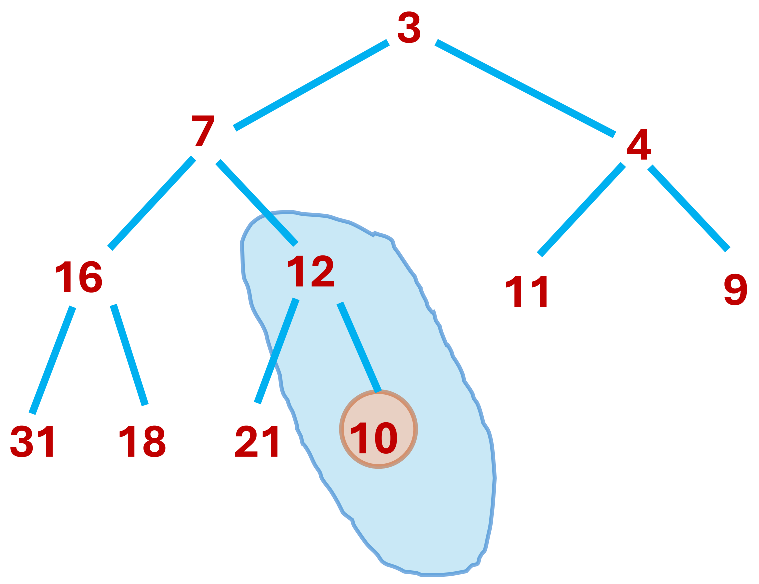
We are good

(by luck of it being 17, > parent 12)



0	1	2	3	4	5	6	7	8	9	10	11	. . .
	3	7	4	16	12	11	9	31	18	21	17	

Insert… what if ?



insert (10)

still has heap-order property ?

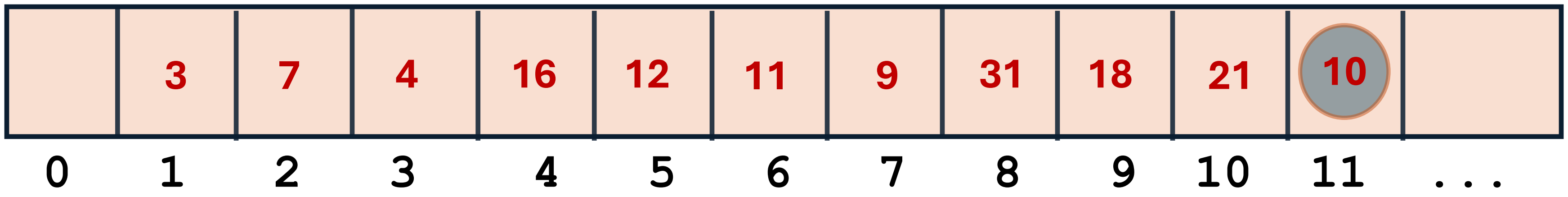
We are not good

(since 10 < parent 12)

swap 10 with parent 12

Good here

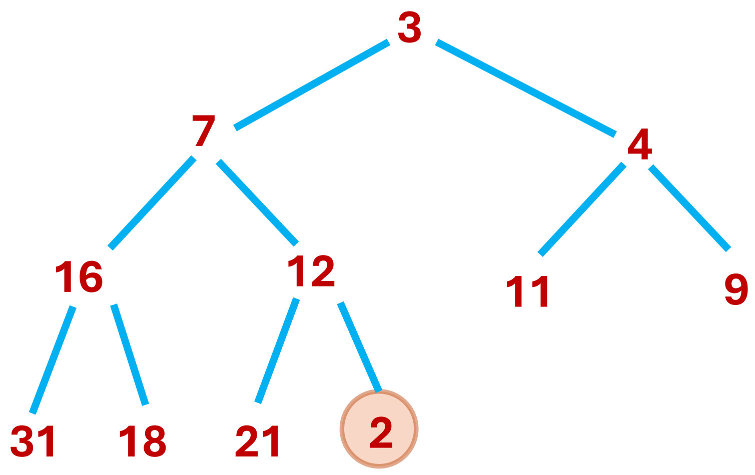
And good here



	3	7	4	16	12	11	9	31	18	21	10	
0	1	2	3	4	5	6	7	8	9	10	11	. . .

Slot 11 parent is floor(11/2) = 5

Swap-Up the Value



insert (2)

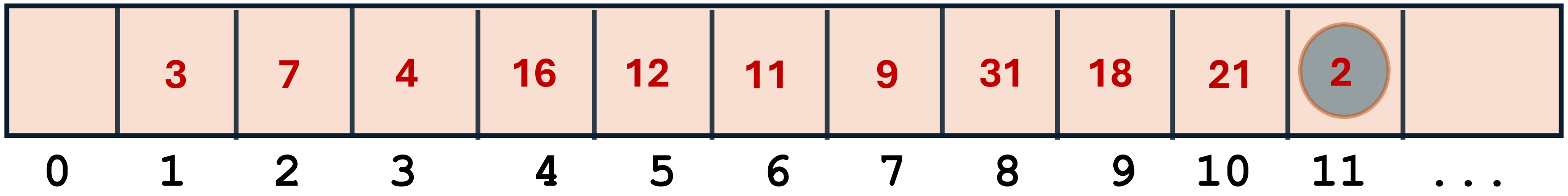
Check for heap-order, swap upwards
repeat until we get it

In the array representation…

Slot 11 parent is floor(11/2) = 5

Slot 5 parent is floor(5/2) = 2

Slot 2 parent is floor(2/2) = 1



0	1	2	3	4	5	6	7	8	9	10	11	. . .
	3	7	4	16	12	11	9	31	18	21	2	

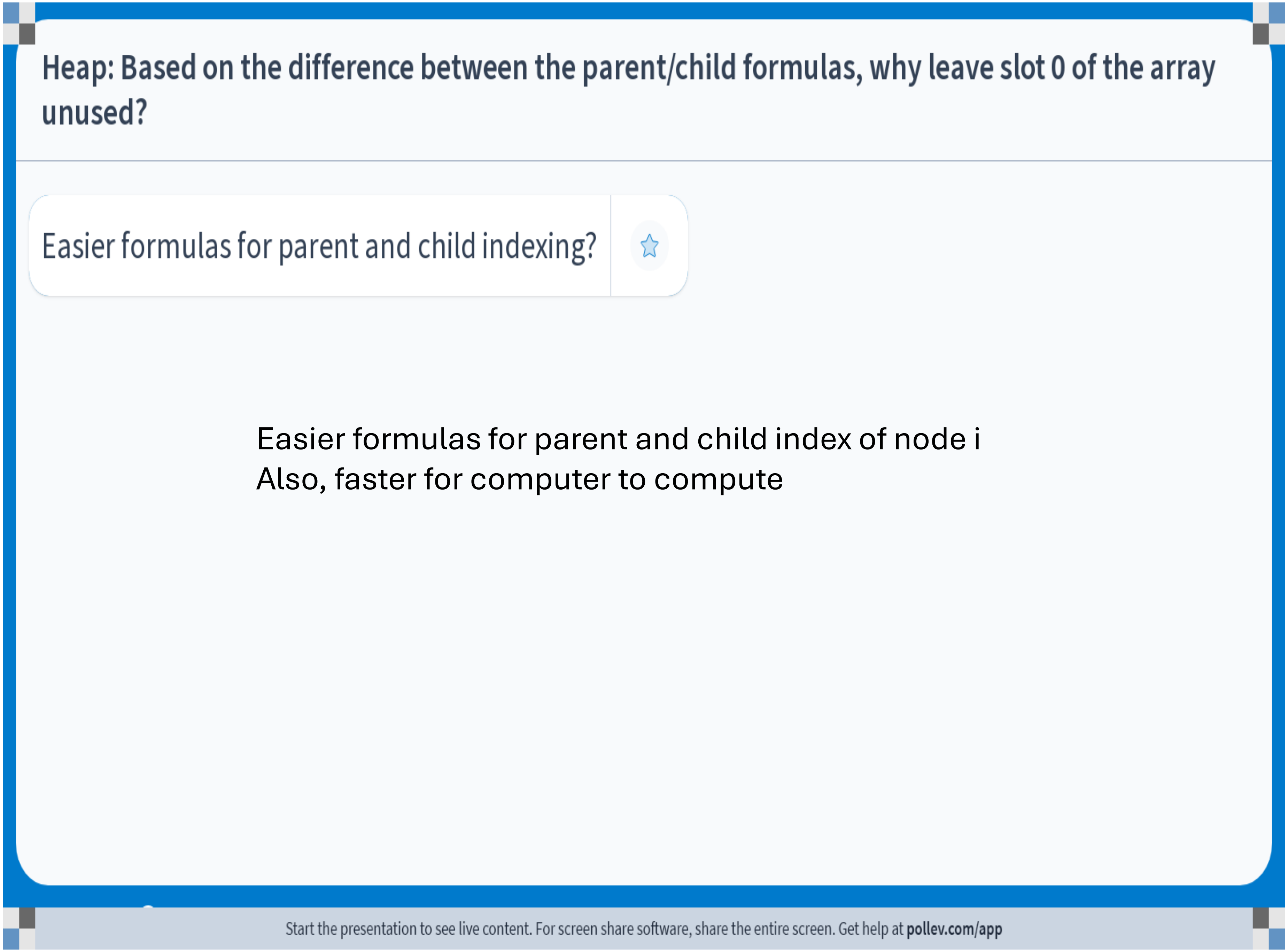
Slot 11 parent is floor(11/2) = 5

Start with an empty minimum binary heap and insert the following values in this order (separate inserts): 25, 6, 17, 8, 5, 12, 7, 22, 3. Which of the following is correct for the order of elements in the heap array (smallest index to largest)?

- 3, 5, 6, 7, 8, 12, 17, 22, 25 — 0%
- ✓ 3, 5, 7, 6, 8, 17, 12, 25, 22 — 0%
- 3, 7, 5, 12, 17, 8, 6, 22, 25 — 0%
- 3, 5, 6, 25, 22, 8, 7, 17, 12 — 0%
- None of the above — 0%

Alternate array representation – use index 0

- Why leave slot 0 unused? Wastes space?
- If we used slot 0,
  - For node in slot i
    - Parent is at $\lfloor (i-1) / 2 \rfloor$ (floor, int division)
    - Left child at $2i + 1$
    - Right child at $2i+2$
- Our previous formulas (leaving 0 unused) were
  - For node in slot i
    - Parent is at $\lfloor i / 2 \rfloor$
    - Left child at $2i$
    - Right child at $2i+1$



Heap: Based on the difference between the parent/child formulas, why leave slot 0 of the array unused?

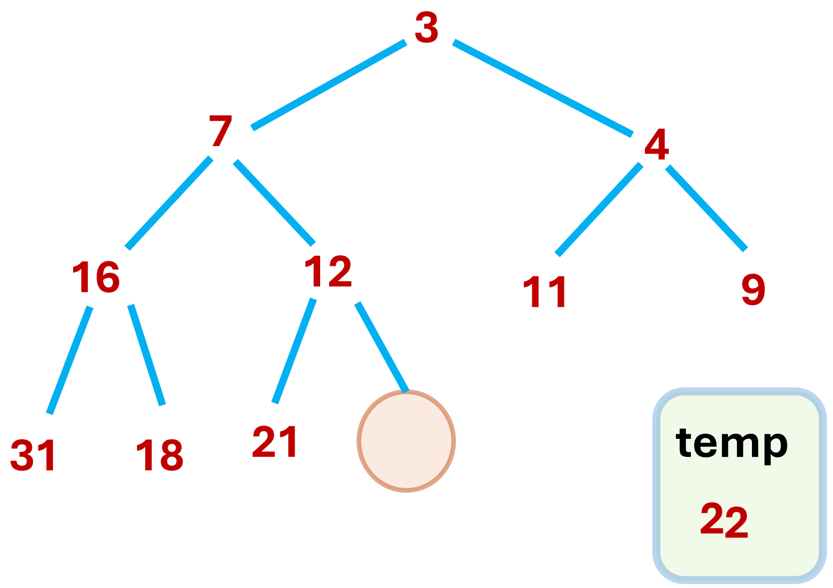
Easier formulas for parent and child indexing? ☆

Easier formulas for parent and child index of node i
Also, faster for computer to compute

Insert code details

- Bubble up element
  - Each swap takes 3 assignments (`temp` variable)
  - O(log n) swaps
  - O(3 log N) = O(log n)
- Alternatively, “bubble-up the hole” to avoid swaps, as shown on next slides

Bubble-up the hole

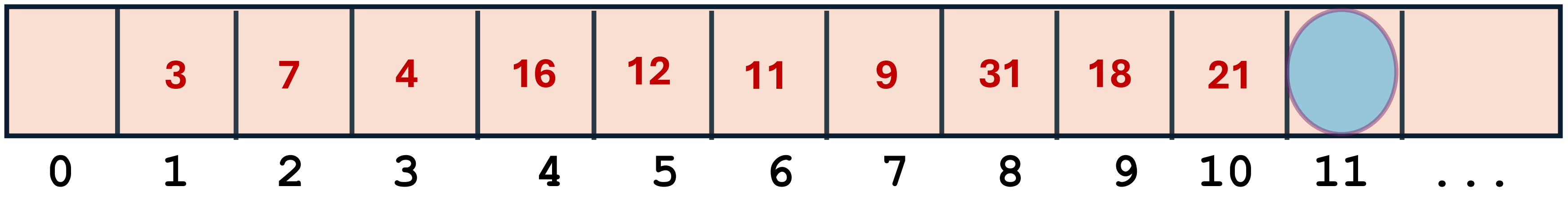


insert (2)

Check for heap-order, if 2 were put in the hole

Move parent down into hole, check 2 again, and again

In the array representation Move values into the hole, hole moves toward slot 1



Insert time complexity

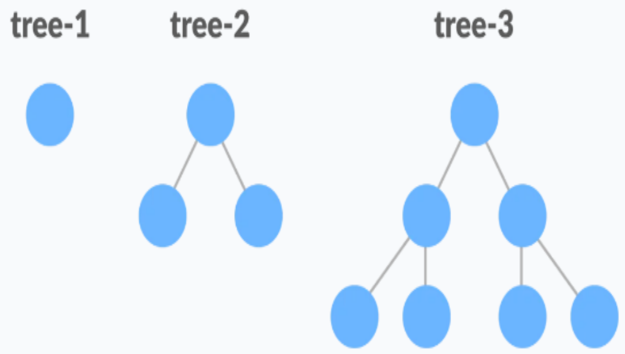
- Put new value in next open slot in array/tree
- Swap/bubble up towards root until heap-order is achieved
- Worst-case **O(log N)** as it follows path to root (height)

Insert average case O(1)!

- `insert` average-case **O(1)**!
  - In short, ½ to ¾ of nodes are in the bottom 2 layers
    - New elements will probably end up at bottom 2 layers or near
    - We consider bottom 2 layers, not bottom layer only, because bottom layer of complete tree may have only one element
  - To make the rest of this analysis easier, assume the tree is perfect so that ½ of nodes are leaves
  - Leaves tend to be larger due to heap-order
    - If we assume (for now) the leaves are the largest elements in the heap, then the probability that the new element will be a leaf (upper half of values) is 0.5
    - Otherwise, suppose the new element will be in the second-to-last layer, with probability 0.25
      - Full second to last-layer has ½ the nodes of full last layer
    - Compute expected value
      - Expected number of comparisons is 1(0.5) + 2(0.25) + 3(0.125) + … = 2
    - Previous assumption that the new element will be a leaf is 0.5 is false (slightly less), but still bounded by a constant, so the reasoning holds. Also may not be the case that the tree is perfect
  - Average number of comparisons for `insert` is 2.67, O(1)

Show that for a perfect binary tree, when height is large, the proportion of nodes in the bottom 2 layers is roughly 3/4.

- Perfect BT of height h has 2^h nodes in bottom layer
- Perfect BT of height h has 2^{h+1}-1 nodes
- Ignoring the -1 (negligible difference for large h), ½ of a perfect BT's nodes are in the bottom layer
- Thus, ½ (½) = ¼ of a perfect BT's nodes are in the layer above
- ½ + ¼ = ¾



Nobody has responded yet.

Hang tight! Responses are coming in.

getMin time complexity

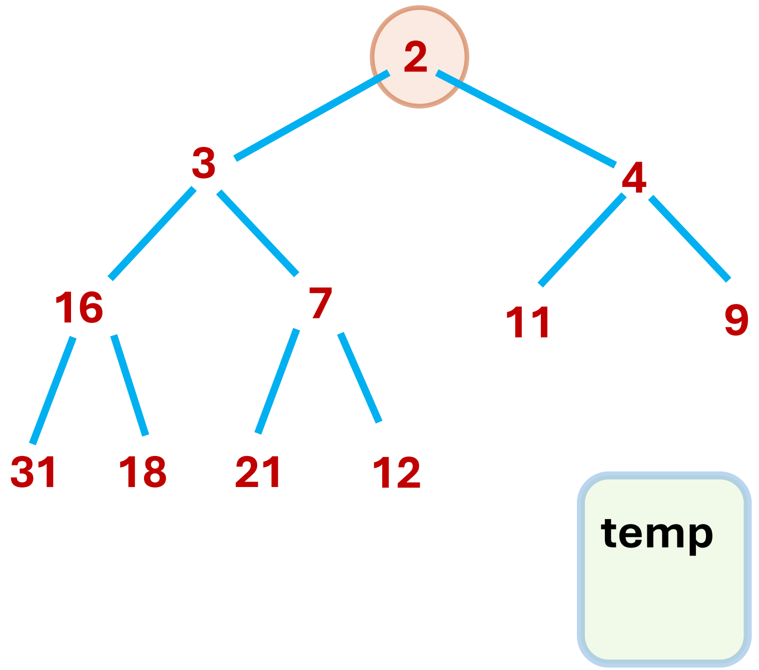
- **O(1)**
    - Read the value at root

delMin

- Delete the minimum (root node) of the min heap and ensure that the new minimum bubbles to the root

- Remove root node val (it is min val)
  - Leaves a “hole” at root node
- Pull out val in last leaf (eliminates a node)
- See if it fits in root hole
  - If so, put leaf val into root hole
  - If not, move hole down to **smaller** child
  - Repeat

delMin Example



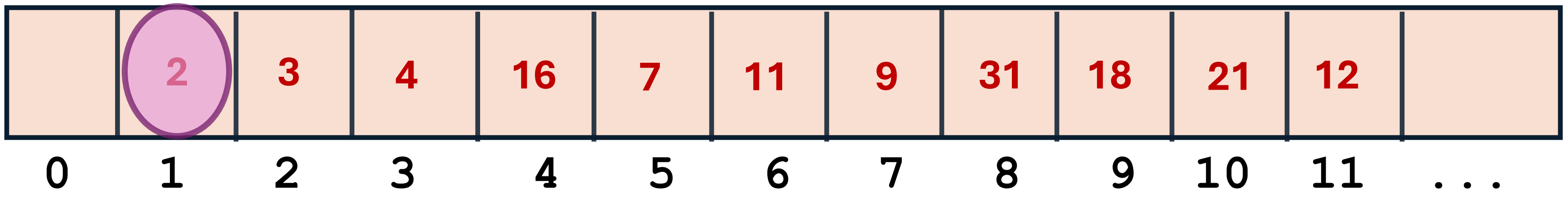
delMin ( )

Remove root value

Save out last element

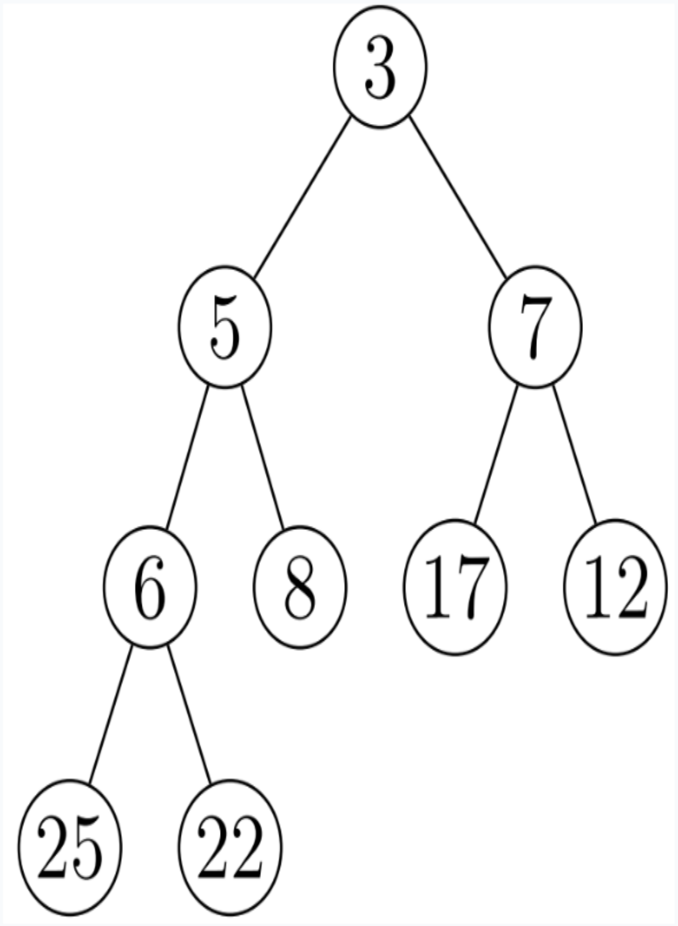
Bubble hole down from root, swapping with smaller child
Stop when last element causes heap-order in the hole

Array representation



0	1	2	3	4	5	6	7	8	9	10	11	. . .
	2	3	4	16	7	11	9	31	18	21	12	

Do a single deleteMin on this min-heap. Which of the following is correct for the order of elements in the heap array (smallest index to largest left to right) after this operation?



5, 6, 7, 8, 12, 17, 22, 25

50%

5, 7, 6, 12, 17, 8, 22, 25

0%

5, 6, 22, 25, 8, 7, 17, 12

0%

✓ 5, 6, 7, 22, 8, 17, 12, 25

50%

None of the above

0%

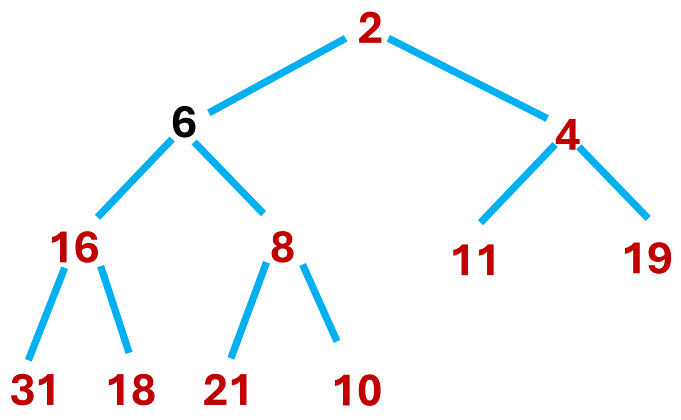
delMin time complexity

- Worst-case **O(log n)**
  - Remove root node: **O(1)**
  - Save out last element: **O(1)**
  - Bubble down the hole: **O(log N)**
    - One copy per bubble move
    - +1 at end for copying temp value into hole
  - Swap down method: **O(log N)**
    - 3 assigns per swap, O(log N) swaps
    - O(3 log N) is O(log N)

increaseKey/decreaseKey

- Find the tree element with the key
- Change priority by adding some increment
- Swap/bubble down (with smaller child) toward leaves until heap-order is achieved
    - Worst-case **O(log N)** as it follows path root to leaf (height)
- For `decreaseKey`, bubble up toward root

increaseKey Example

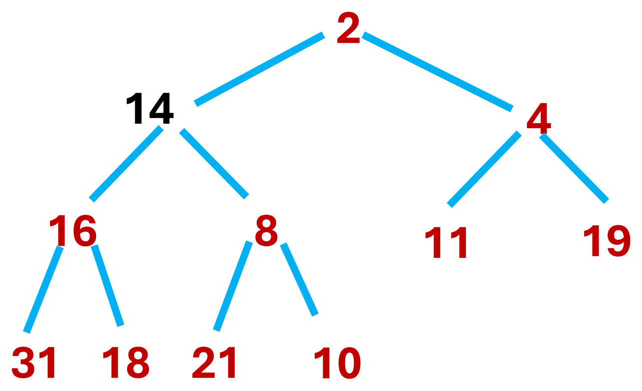


increaseKey(curPriority, delta)

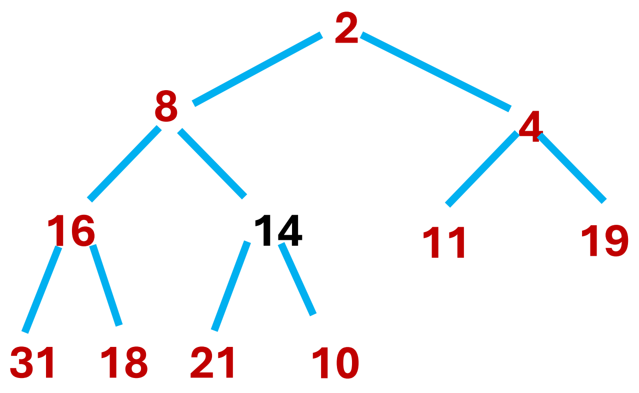
increaseKey(6, 8)

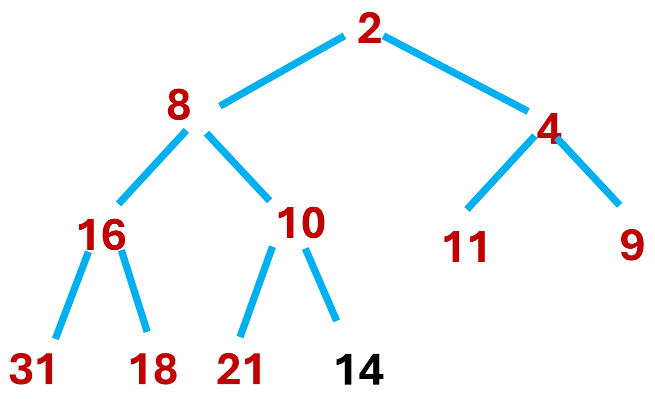
Find element with priority 6

Bump up by delta 8 to 14



Bubble hole down from elt
Stop when heap-order is restored





increaseKey (unoptimized) worst-case time complexity? n is the number of leaves in the tree.

$O(1)$ 0%

$O(\log n)$ 0%

✓ $O(n)$ 0%

$O(n \log n)$ 0%

$O(n^2)$ 0%

increaseKey issues

- As we just saw, can be worst-case **O(n)** to find element with some priority
- To speed up `find` to O(1), can use a `HashMap` along with the heap
    - `HashMap` is similar to Python `dict`
    - Attach unique data field (id) to each node
    - Find node by id
    - Key is id, value is array index
    - Now finding is **O(1)**, but still **O(log n)** cost for bubbling

Heap time complexities

	Average	Worst-case
insert	O(1)	O(log n)
delMin	O(log n)	O(log n)
getMin	O(1)	O(1)
find(priority)	O(n)	O(n)
incKey(curPriority, delta)	O(n) or O(log n) (optimized)	O(n) or O(log n) (optimized)

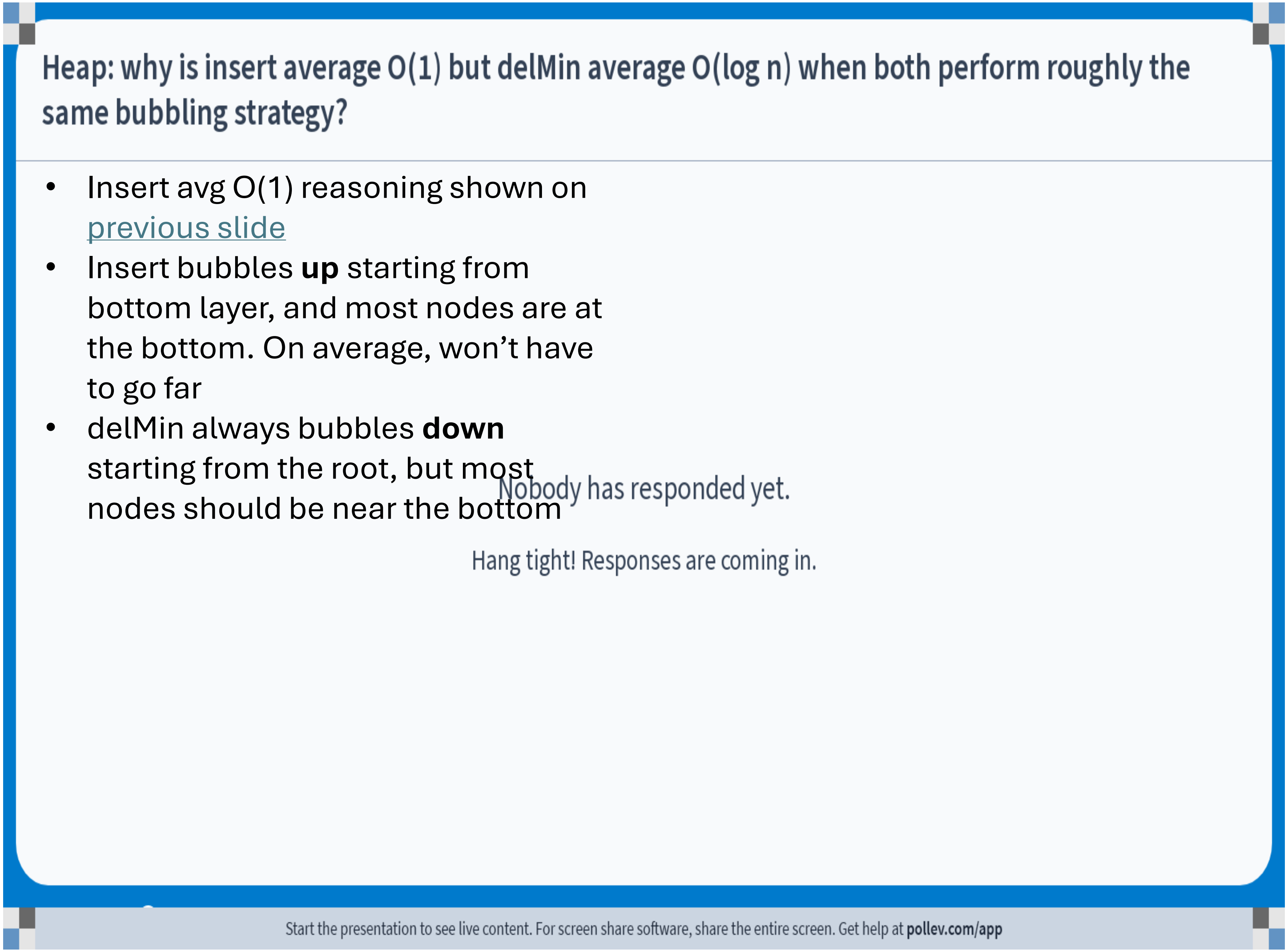
n is the number of nodes

Why is it worst-case O(log n) to insert and remove from a heap but O(n) worst-case to do the same on a BST?

- Degenerate BST has O(n) insert and remove because height is n – 1, not log(n). Insert and remove thus function similarly to linked list insert/remove
- Heap is complete, thus balanced (AVL)
    - No degenerate case
    - Height is always $\lfloor \log_2(n) \rfloor$
    - Insert and remove worst-case O(log n)

Nobody has responded yet.

Hang tight! Responses are coming in.



Heap: why is insert average O(1) but delMin average O(log n) when both perform roughly the same bubbling strategy?

- Insert avg O(1) reasoning shown on previous slide
- Insert bubbles **up** starting from bottom layer, and most nodes are at the bottom. On average, won't have to go far
- delMin always bubbles **down** starting from the root, but most nodes should be near the bottom

Nobody has responded yet.

Hang tight! Responses are coming in.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

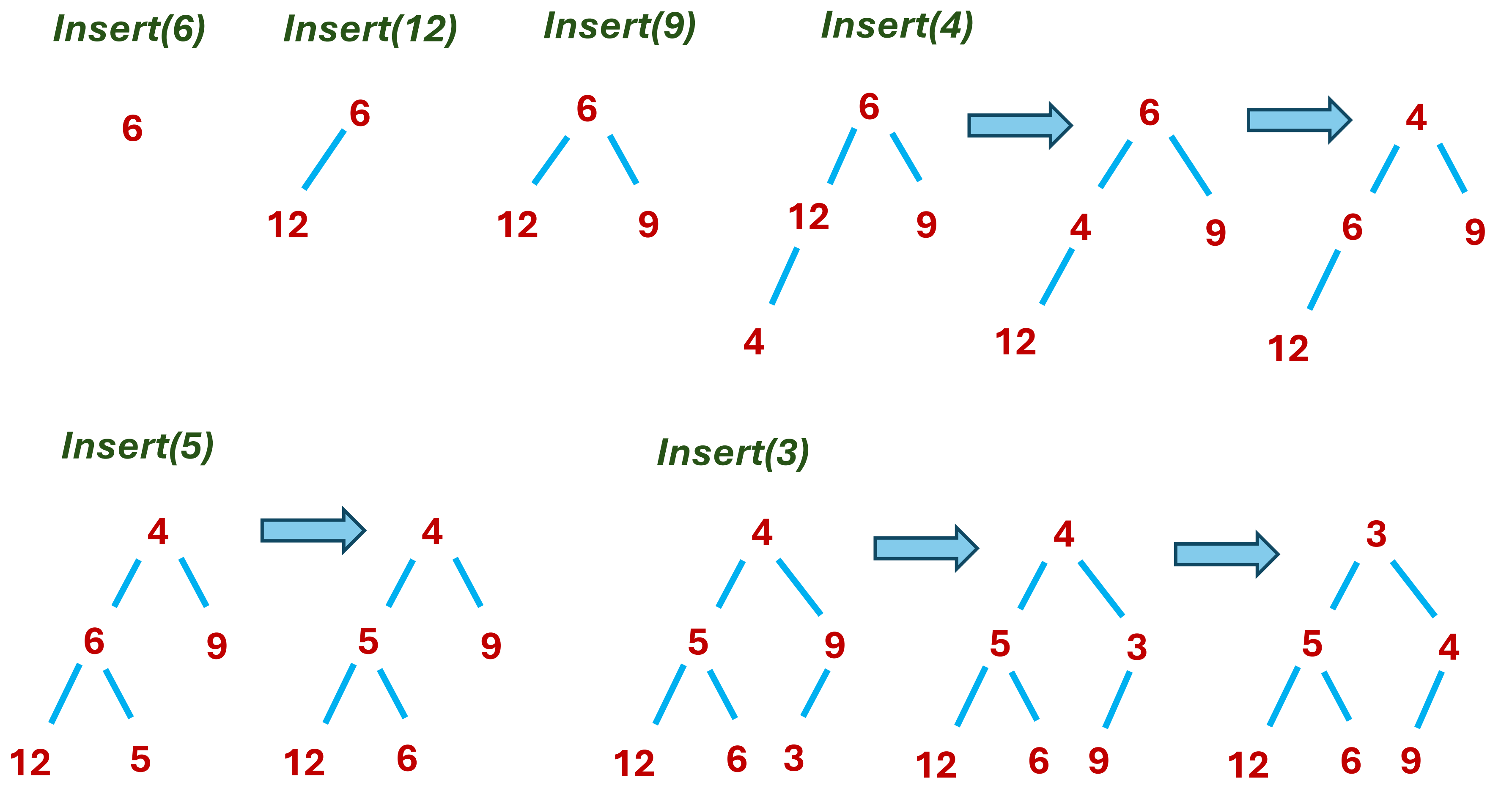
Heap limitations

- Although heap has much faster `getMin` than BST, it has less sort information
- Can't get full sort out of heap directly like a BST with in-order traversal
- How to sort with heap?
    - Recall, quicksort and mergesort average case O(n log n)
    - BST sort average O(n log n), worst O(n^2)

Sorting with heap (naïve)

- Worst-case
    - `insert` O(log n)
    - Insert n items O(n log n)
    - To get sorted sequence, `delMin` n times
    - `delMin` is O(log n)
    - Retrieve sequence is O(n log n)
    - O(n log n) + O(n log n) = **O(n log n)** is not practically as good as BST sort's average case
        - BST sort average-case is O(n log n) + O(n)

Make BHEAP by N Inserts



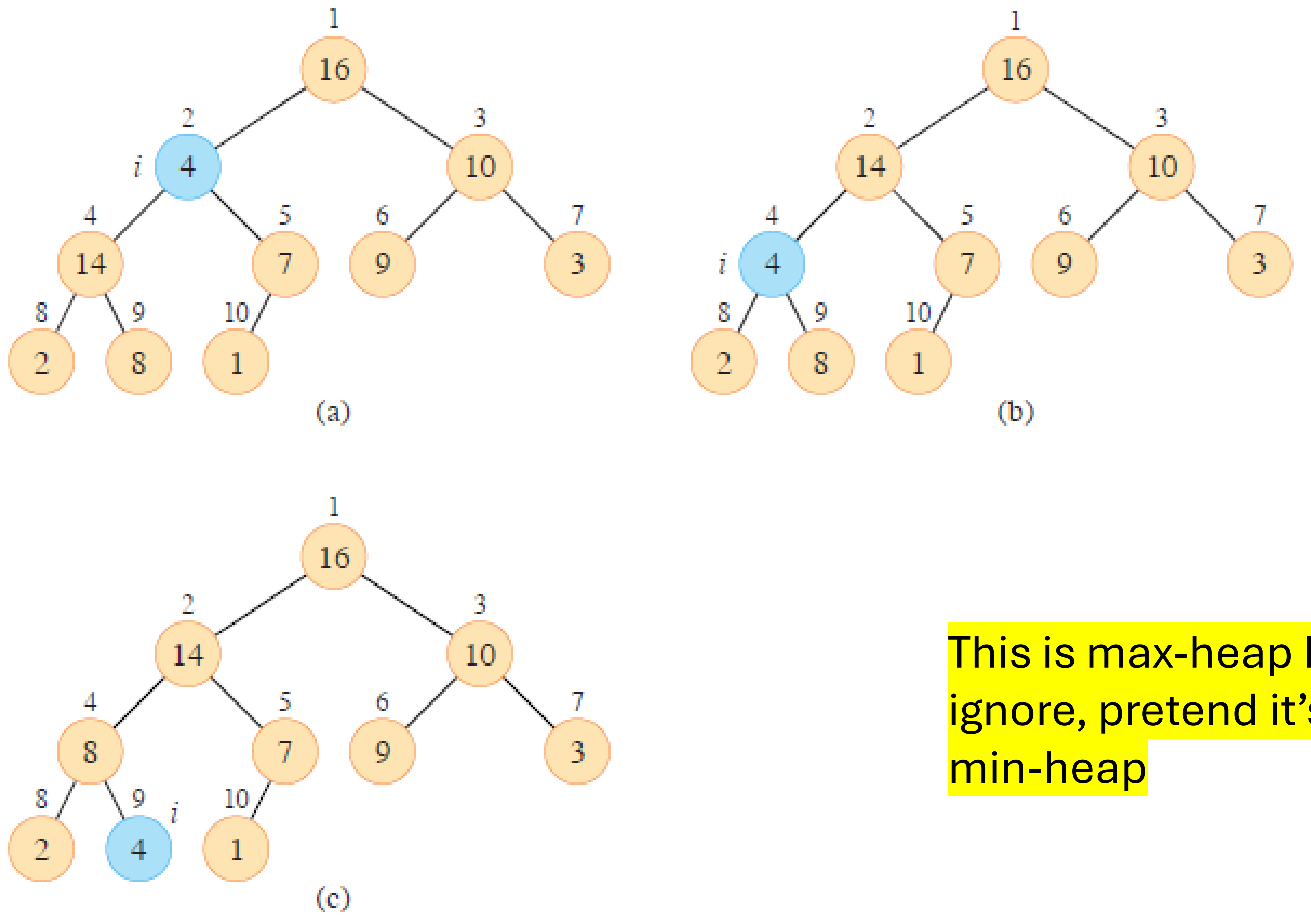
O(N log N)

Efficient build heap

- There is a way to convert a list of unsorted n elements to a heap in **O(n)** time
- Structural property
    - Load n elements into array from slot 1 to last (order unimportant)
- Heap-order
    - Call `min-heapify` $\lfloor n / 2 \rfloor$ times
        - Shown on next slide

Min-heapify(A, i)

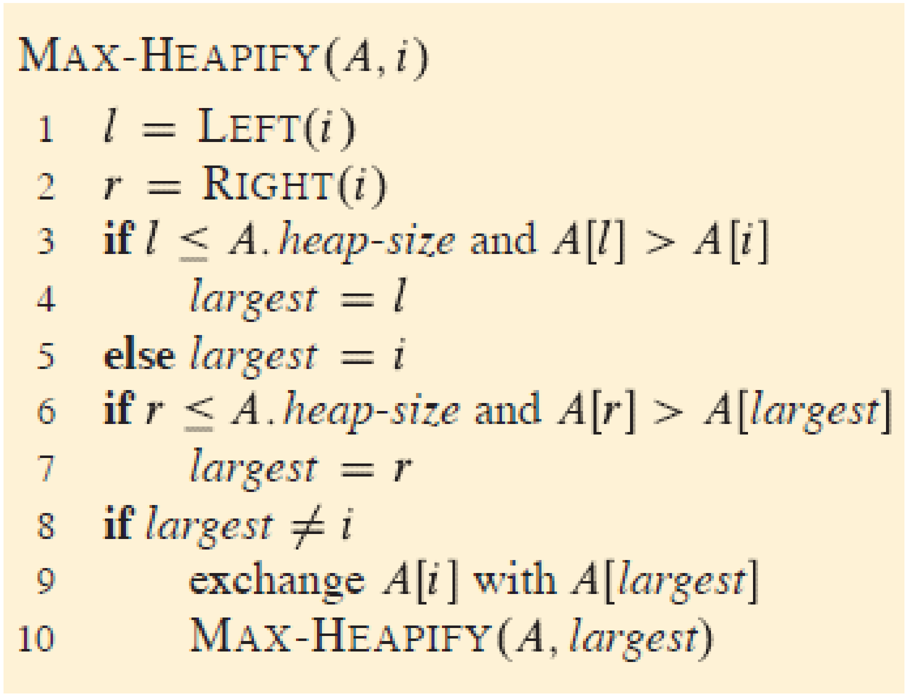
- `i` is the index of the node that we “min-heapify”
- `Min-heapify` assumes that the children of `i` (left and right) are both min heaps, but `i` may violate heap order
- Bubble-down node at index `i` until heap-order is satisfied
    - Similar to `deleteMin`, must swap with smaller element



This is max-heap but ignore, pretend it's a min-heap

Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.\mathit{heap\text{-}size} = 10$. The node that potentially violates the max-heap property is shown in blue. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After $A[4]$ and $A[9]$ are swapped, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

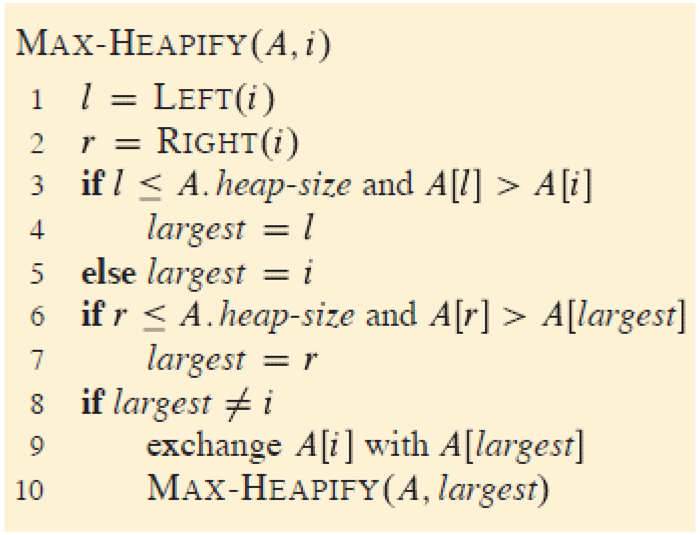
Min-heapify



```
MAX-HEAPIFY$(A, i)$
$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq A.\mathit{heap\text{-}size}$ and $A[l] > A[i]$
        $\mathit{largest} = l$
else $\mathit{largest} = i$
if $r \leq A.\mathit{heap\text{-}size}$ and $A[r] > A[\mathit{largest}]$
        $\mathit{largest} = r$
if $\mathit{largest} \neq i$
        exchange $A[i]$ with $A[\mathit{largest}]$
        MAX-HEAPIFY$(A, \mathit{largest})$
```

This is max-heapify, where we swap (possibly) with the larger child, then recur. For min-heapify, simply find the smaller child and swap (possibly) with that

Min-heapify analysis

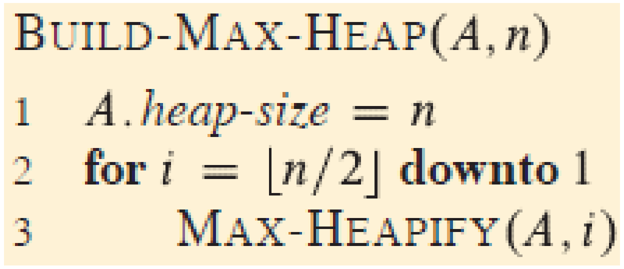


```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4       largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8  if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

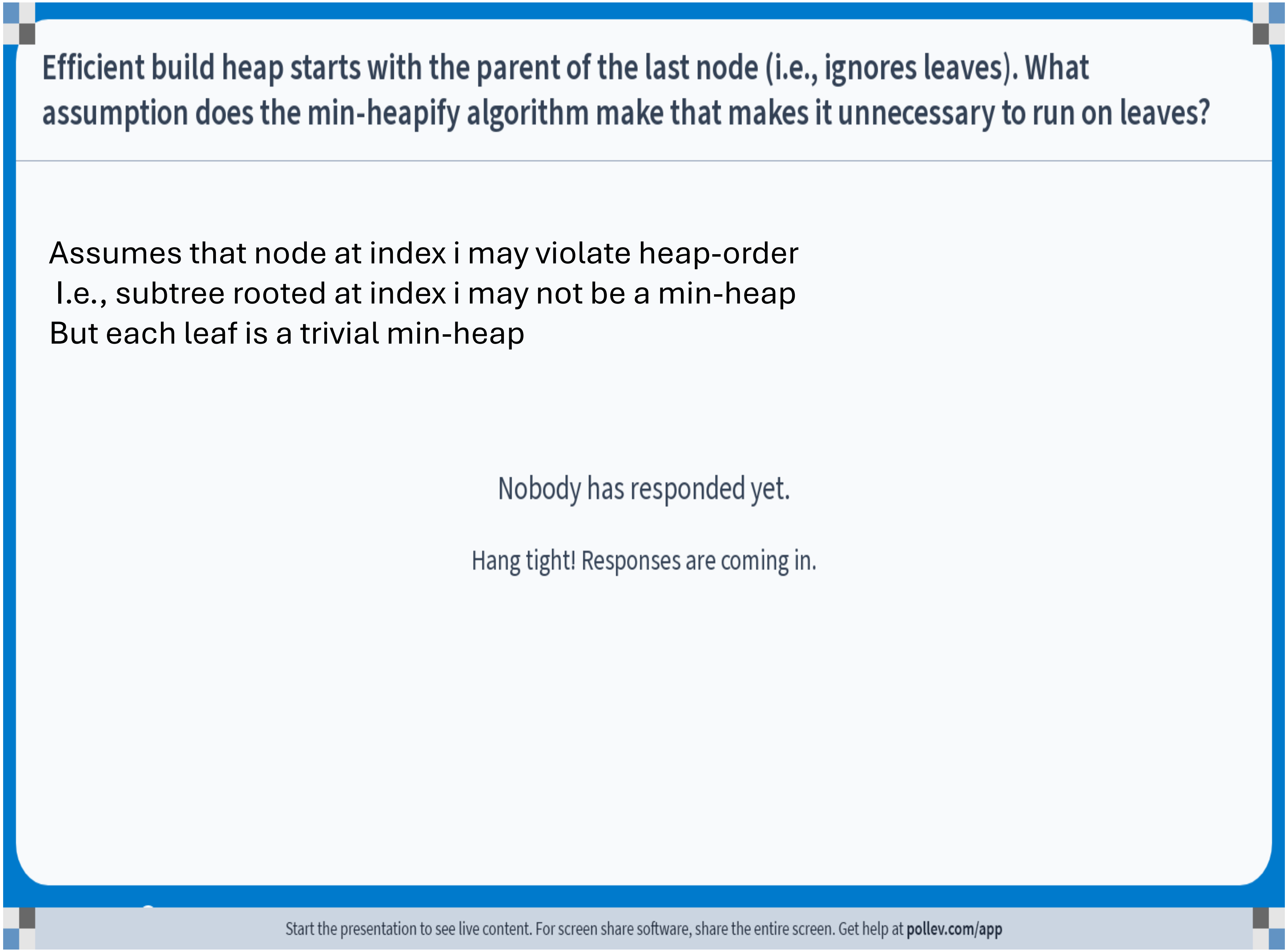
- Lines 1-9 are Θ(1)
- Recursive call on line 10 will operate on a child subtree of size at most 2n/3
  - Proof that children subtrees have size at most 2n/3 is omitted
- Thus, `min-heapify` worst-case recurrence is $T(n) \leq T(2n/3) + \Theta(1)$
- Solution by master theorem (not covered in COMP 210) is **O(log n)**
  - Or use recursion tree

Efficient build heap

- Start with **parent of last node**, bubble down (call `min-heapify`) as needed
- Go breadth-order backward to root



```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



Efficient build heap starts with the parent of the last node (i.e., ignores leaves). What assumption does the min-heapify algorithm make that makes it unnecessary to run on leaves?

Assumes that node at index i may violate heap-order
I.e., subtree rooted at index i may not be a min-heap
But each leaf is a trivial min-heap

Nobody has responded yet.

Hang tight! Responses are coming in.

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

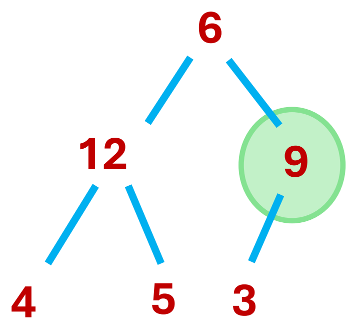
Make BHEAP by Build

Insert 6, 12, 9, 4, 5, 3

We have all values at the start, no need to do individual insert ops

0	1	2	3	4	5	6	7	8	9	10	11	…
	6	12	9	4	5	3						

Initial tree form



Now bubble down

Start with first non-leaf that has a child

What array slot is this?

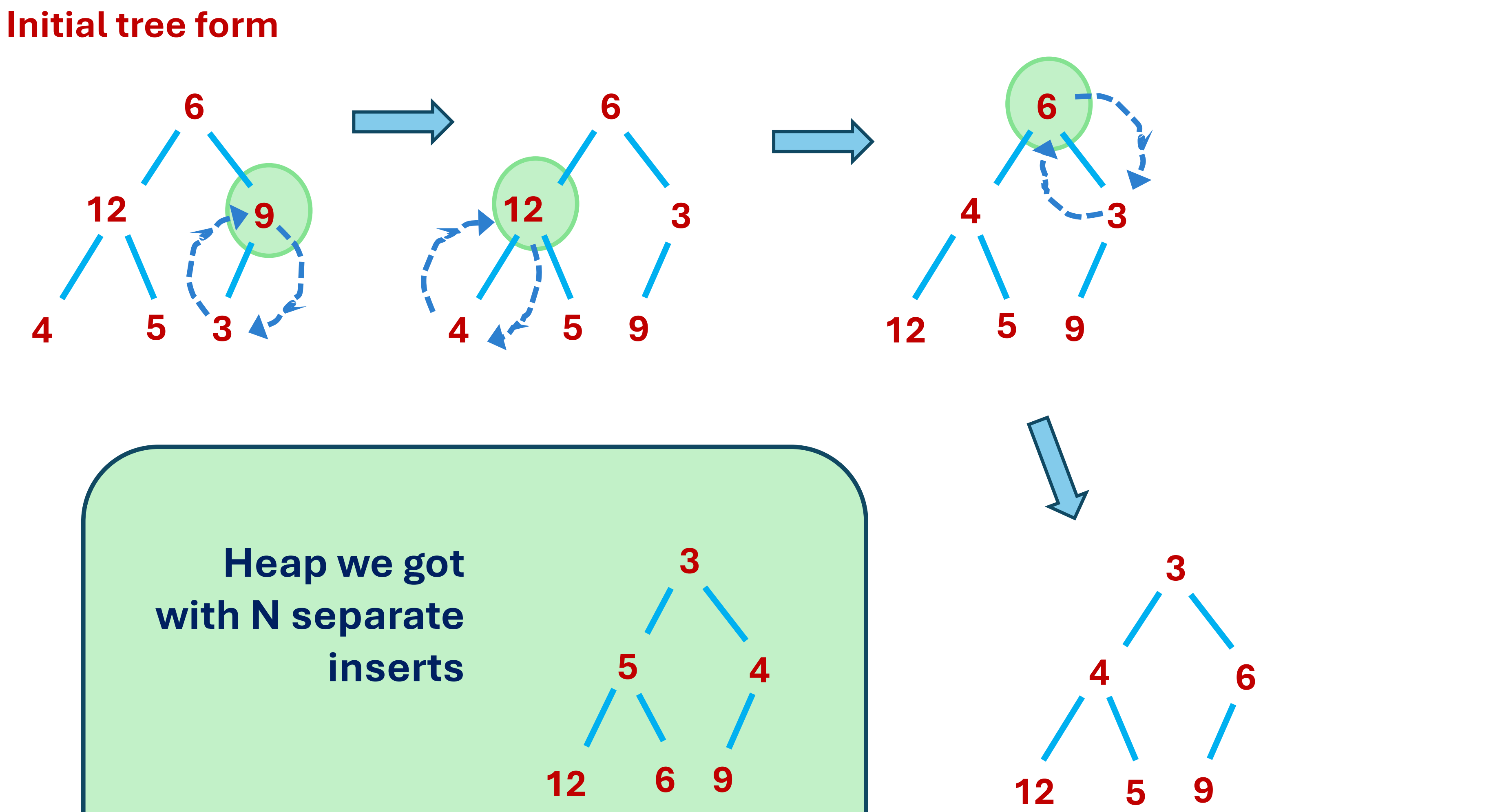
parent of the last array item

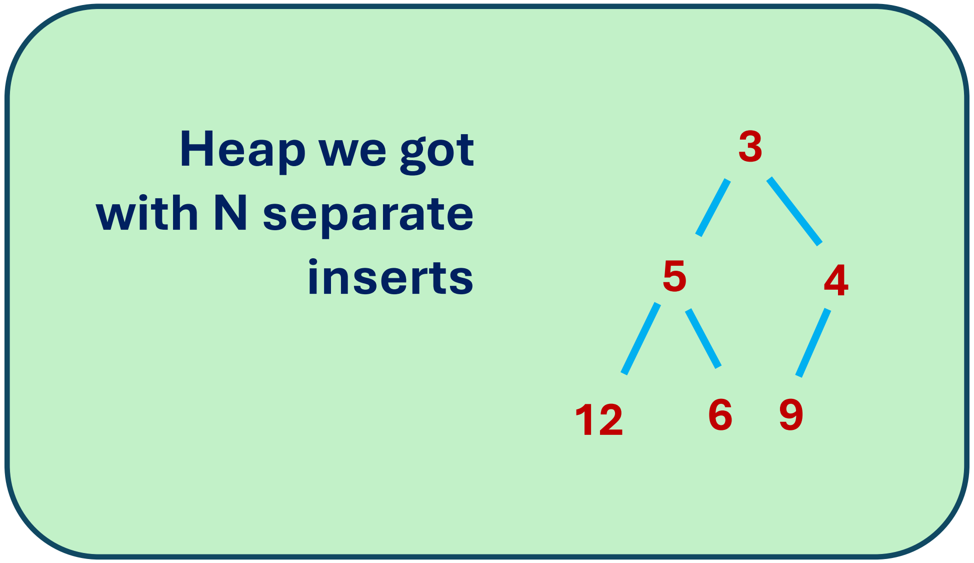
Last item (3) is in slot 6

Parent of 3 is floor(6/2) which is 9, in slot 3

So start bubble down at item in slot 3.

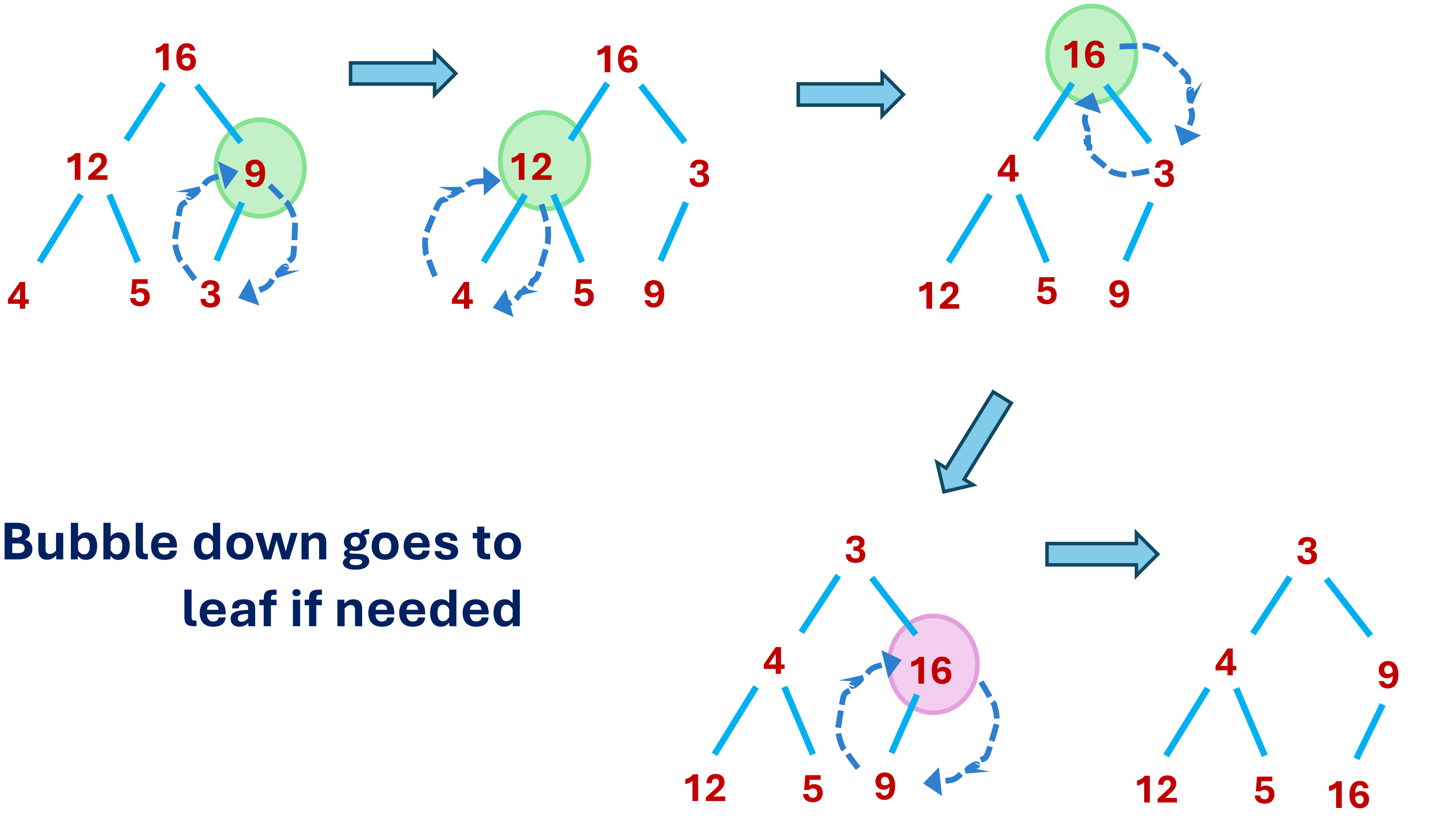
Bubble Down in Build




Heap we got with N separate inserts

Bubble Down in Build

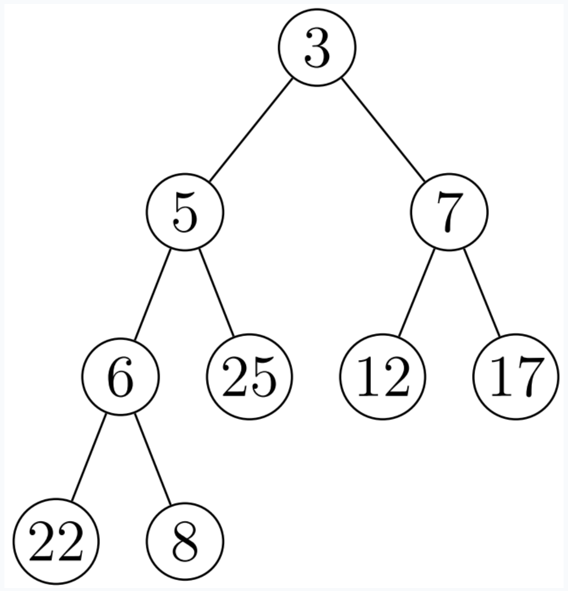
Initial tree form



Bubble down goes to leaf if needed

Perform the buildMinHeap (efficient build heap algorithm) on the array [25, 6, 17, 8, 5, 12, 7, 22, 3] (in that order). Then list the numbers in the array in order from index 1 to 9.

0



3, 5, 7, 6, 25, 12, 17, 22, 8

Nobody has responded yet.

Hang tight! Responses are coming in.

buildMinHeap analysis (informal)

- Naïve upper bound **O(n log n)**
  - Each `min-heapify` call is worst-case O(log n)
  - Build makes O(n) calls
  - O(n log n)
- Tighter upper bound **O(n)**
  - `min-heapify`’s runtime depends on the height of the node
  - The height of most nodes are small (e.g., ½ to ¾ of nodes are in the bottom 2 layers)
    - Similar to `insert` average case O(1) analysis
  - Full proof on CLRS 4 ed. pg. 169

Heapsort time complexity

- Build heap (efficient) - O(n)
- Retrieve sequence by performing `delMin` n times - O(n log n)
- O(n log n) + O(n) = **O(n log n)**
  - Theoretically same as BST sort

Real heapsort

- Heapsort can be done in-place (no extra array memory needed to store results of `delMin`)
- `delMin` removes an element from heap array at end, leaving an array slot open at end
- Simply put the removed element at array end (heap last + 1)
- Repeat until heap empty
- Array will contain all elements in **reverse** order

In-Place Heap Sort

Elements: **7, 2, 19, 8, 11, 4, 16**

Initial heap array:	x	7	2	19	8	11	4	16
	0	1	2	3	4	5	6	7
Complete *Build*:	x	2	7	4	8	11	19	16
	0	1	2	3	4	5	6	7
First *delMin*:	x	4	7	16	8	11	19	2
	0	1	2	3	4	5	6	7
Next *delMin*:	x	7	8	16	19	11	4	2
	0	1	2	3	4	5	6	7
Final array:	x	19	16	11	8	7	4	2
	0	1	2	3	4	5	6	7

In-place heapsort details

- Final sort ends up in reversed order in array
    - Previous slide used min-heap, but sorted sequence is in nonincreasing (large to small) order
- For sort in nondecreasing order,
    - Use max-heap
    - Or reverse the array
        - Swap first and last, then second and next-to-last
            - n / 2 swaps, each swap 3 assignments
            - **O(n)**
        - Use a stack
            - 2n pushes and pops
            - **O(n)**
            - But needs twice as much memory