LEC 12

CSE 373

Priority Queues, Heaps

BEFORE WE START

Rank the following code snippets: which would you expect to exhibit the most "spatial locality"?

1.
```
for (int i = 0; i < n; i++)
  arr[i] += 1;
```

2.
```
while (curr != null) {
  curr.data += 1;
  curr = curr.next;
}
```

3.
```
for (int i = 0; i < 30; i++)
  for (int j = 0; j < n; j += 30)
    arr[i + j] += 1;
```

pollev.com/uwcse373

Instructor Aaron Johnston

TAs Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

Announcements

- P2 due Wednesday 11:59pm PDT
- THANK YOU for filling out the mid-quarter feedback survey! 70% response rate – you rock!
  - We'll examine thoroughly as a staff and implement feedback where possible
  - Quick takeaways for now: PollEverywhere is ☺, long announcements are ☹
- Exam I
  - Additional Review Materials (including a Practice Exam) for Exam I will be released today
  - Exam I is open-book and done in groups, but we still recommend reviewing the material beforehand
    - TAs would love to help you with reviewing concepts beforehand! (during the test, only clarification questions)
    - Helps make group work more efficient if you're on the same page about the fundamentals
- Office Hours
  - We're exploring ways to make OH/Discord more effective for conceptual questions! Usually big group video calls. Don't be afraid to do the same with peers!

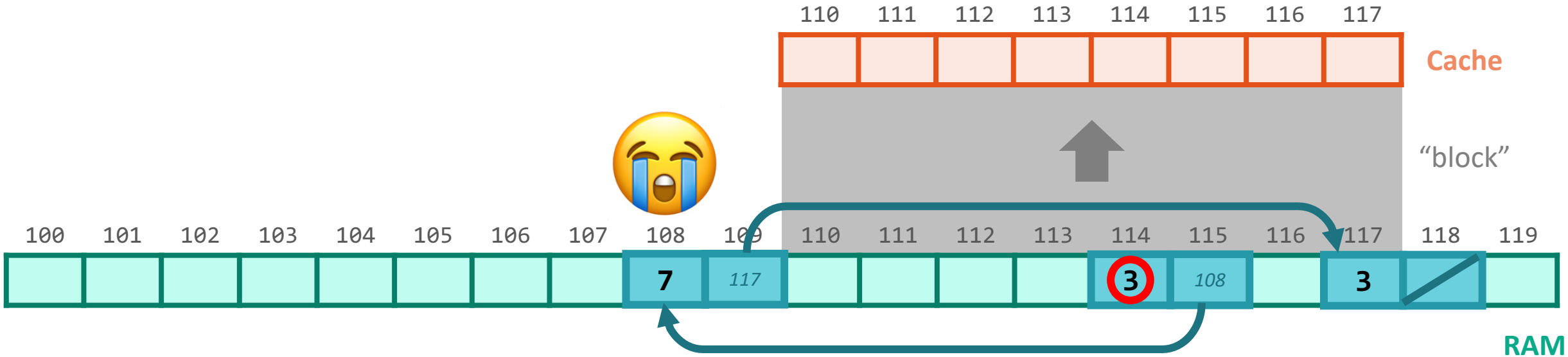
Learning Objectives

After this lecture, you should be able to...

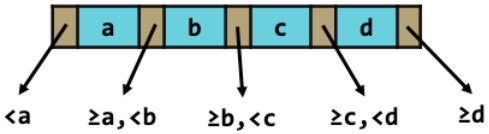
1. Identify use cases where the PriorityQueue ADT is appropriate
2. Describe the invariants that make up a heap and explain their relationship to the runtime of certain heap operations
3. Trace the removeMin(), and add() methods, including percolateDown() and percolateUp()
4. Describe how a heap can be stored using an array, and compare that implementation to one using linked nodes

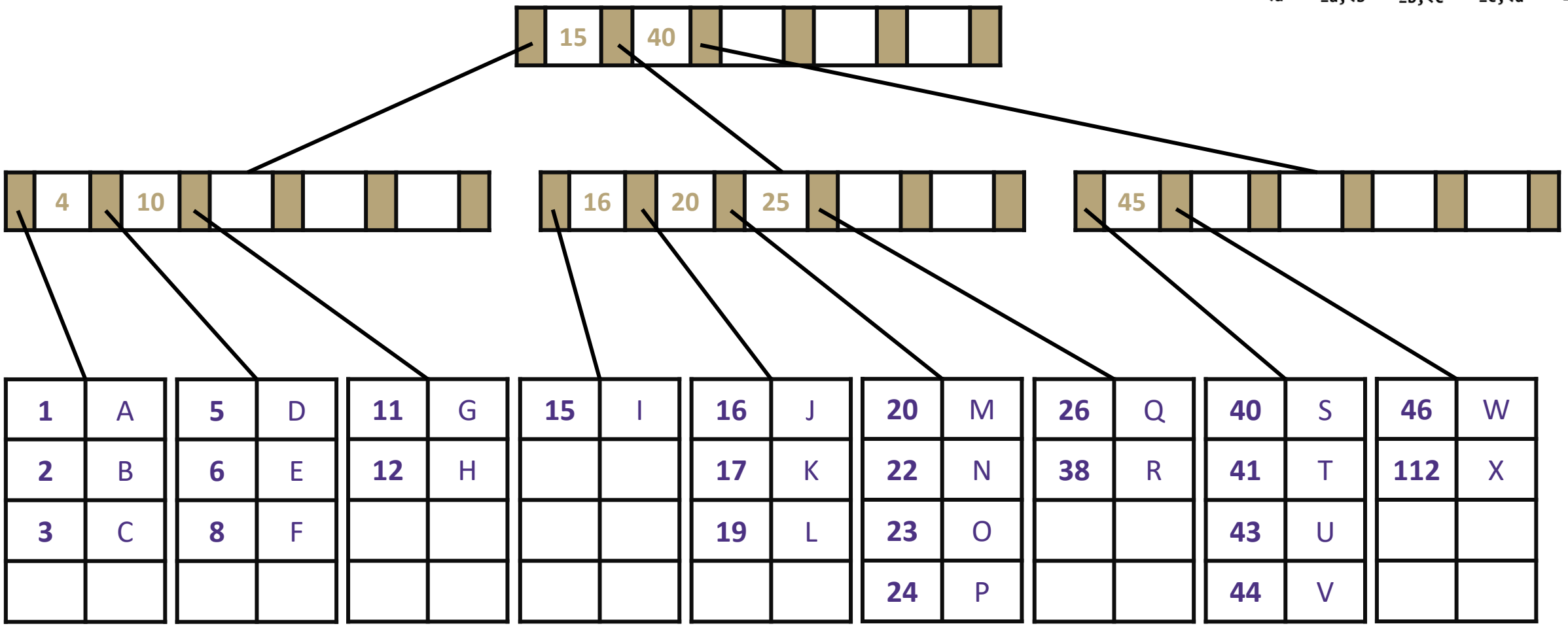
Review Cache Implications: Linked Lists

- Linked lists can be spread out all over the RAM array
  - Do not exhibit strong spatial locality!
- Don't get the same cache benefits – frequently the next list node is far enough away that it's not included on the same block



Review B+ Tree Example: `get(23)`





Review Why Are B+ Trees so Disk-Friendly?

1. We **minimized the height** of the tree by adding more keys/potential children at every node. Because the nodes are more spread out at a shallower place in the tree, it takes fewer nodes (disk-accesses) to traverse to a leaf.
2. All relevant information about a single node **fits in one page** (If it's an internal node: all the keys it needs to determine which branch it should go down next. If it's a leaf: the relevant K/V pairs).
3. We use **as much of the page as we can**: each node contains many keys that are all brought in at once with a single disk access, basically "for free".
4. The time needed to do a search within a node is **insignificant** compared to disk access time, so looking within a node is also "free".

Lecture Outline

- PriorityQueues
  - PriorityQueue ADT
  - PriorityQueue Implementations
- Binary Heaps
  - Binary Heap Idea & Invariants
  - Binary Heap Implementation Details

An Important Use Case

- Imagine you're managing food orders at a restaurant, which are normally first-come-first-served.
  - You keep track of this using a Queue ADT (see LEC 02).

- Suddenly, Ana Mari Cauce walks into the restaurant!

- Of course, you realize that you want to serve her as soon as possible, and other celebrities (CSE 373 TAs) may not be far behind.
  - You realize your food management system should have a way to rank customers base on priority, to decide which food order to work on next (the most prioritized thing, not necessarily the first or last thing).

Priority Queue ADT

MIN PRIORITY QUEUE ADT

State

Set of comparable values (*ordered based on "priority"*)

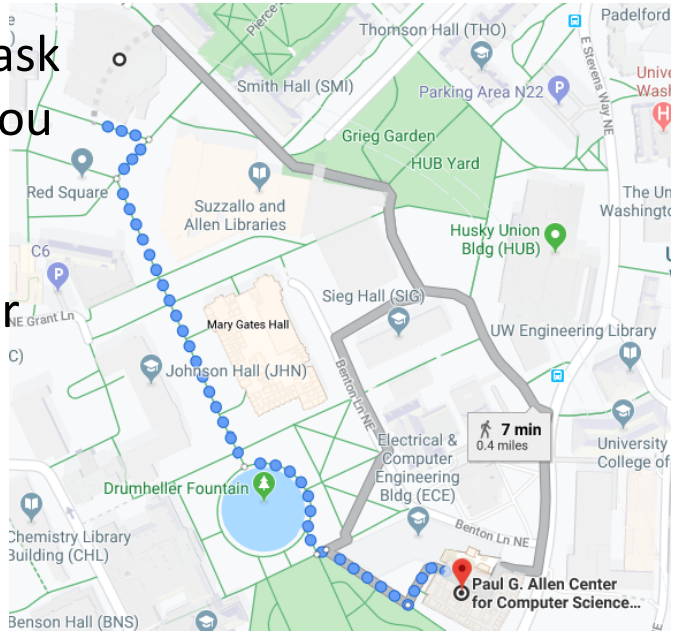
Behavior

add(value) – add a new element to the collection

removeMin() – remove and return the element with the smallest priority

peekMin() – find and return, but do ot remove, the element with the smallest priority

- An incredibly useful ADT
  - Well-designed printers
  - Huffman Coding (see last CSE 143 hw)
  - Sorting algorithms (coming up: week 8)
  - Graph algorithms (coming up: week 6)
- Example: route finding
  - Represent a map as a series of *segments*
  - At each intersection, ask which segment gets you closest to the destination (ie, has max priority or min distance)



Priority Queue ADT

- If a Queue is "First-In-First-Out" (FIFO), Priority Queues are "Most-Important-Out-First"
- Items in Priority Queue must be **comparable** – The data structure will maintain some amount of internal sorting, in a sort of similar way to BSTs/AVLs
- We'll talk about "Min Priority Queues" (lowest priority is most important), but "Max Priority Queues" are almost identical

MIN PRIORITY QUEUE ADT

State

Set of comparable values (*ordered based on "priority"*)

Behavior

add(value) – add a new element to the collection

removeMin() – returns the element with the smallest priority, removes it from the collection

peekMin() – find, but do not remove the element with the smallest priority

MAX PRIORITY QUEUE ADT

State

Set of comparable values (*ordered based on "priority"*)

Behavior

add(value) – add a new element to the collection

removeMin() – returns the element with the largest priority, removes it from the collection

peekMin() – find, but do not remove the element with the largest priority

Lecture Outline

- **PriorityQueues**
  - PriorityQueue ADT
  - **PriorityQueue Implementations**
- Binary Heaps
  - Binary Heap Idea & Invariants
  - Binary Heap Implementation Details

Implementing Priority Queues: Take 1

Maybe we already know how to implement a priority queue.
How long would removeMin and peek take with these data structures?

Implementation	add	removeMin	Peek
Unsorted Array			
Doubly Linked List (sorted)			
AVL Tree			

For Array implementations, assume you do not need to resize.
Other than this assumption, do **worst case** analysis.

Implementing Priority Queues: Take 1

Maybe we already know how to implement a priority queue.
How long would removeMin and peek take with these data structures?

Implementation	add	removeMin	Peek
Unsorted Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Doubly Linked List (sorted)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

For Array implementations, assume you do not need to resize.
Other than this assumption, do **worst case** analysis.

Implementing Priority Queues: Take 1.5

Maybe we already know how to implement a priority queue.
How long would removeMin and peek take with these data structures?

Simple improvement: add a field to keep track of the min.
Update on every insert or remove.

Implementation	add	removeMin	Peek
Unsorted Array	$\Theta(1)$	$\Theta(n)$	~~$\Theta(n)$~~ $\Theta(1)$
Doubly Linked List (sorted)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	~~$\Theta(\log n)$~~ $\Theta(1)$

Review Binary Trees vs. BSTs

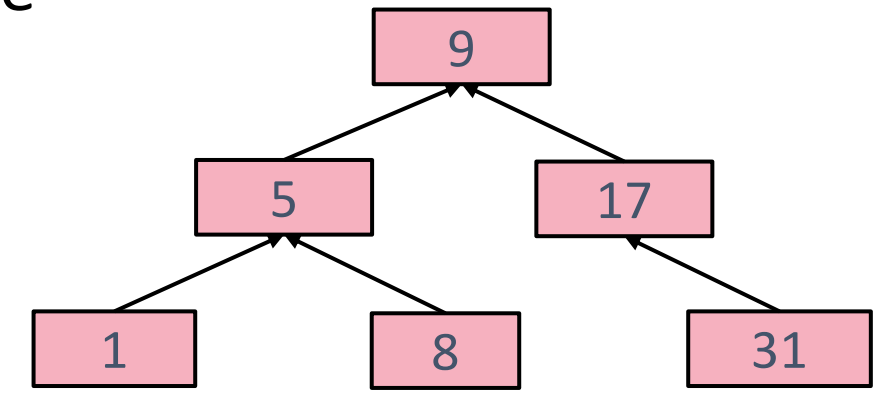
- A **Binary Tree** has at most 2 children per node
- A **Binary Search Tree** adds the BST invariant:

INVARIANT

Binary Search Tree Invariant:

For every node with key k:

- The left subtree has only keys smaller than k.
- The right subtree has only keys greater than k.




```
class BinaryNode<Value> {
  Value v;
  BinaryNode left;
  BinaryNode right;
}
```

Reminder: the BST ordering applies recursively to the entire subtree

Heaps

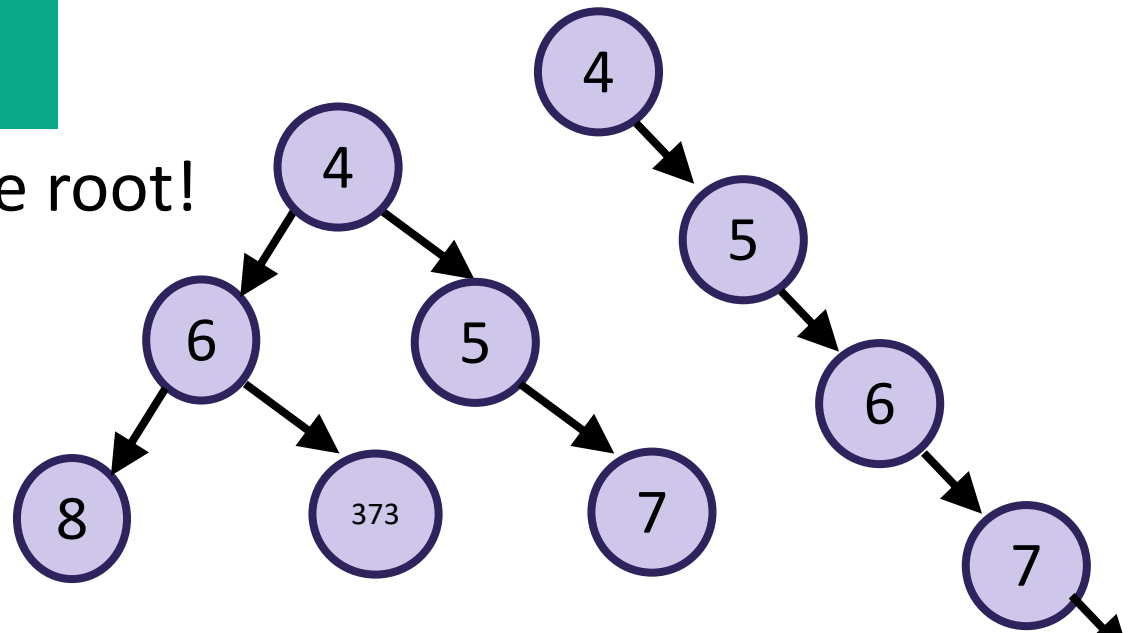
- Intuition:
  - In a BST, we organized the data to find **anything** quickly: go left or right to find a value deeper in the tree
  - Now we just want to find the **smallest things** fast, so let's write a different invariant (not on top of BST):

INVARIANT

Heap Invariant

Every node is less than or equal to all of its children.

- In particular, the smallest node is at the root!
  - Super easy to `peek` now!
- Do we need more invariants?



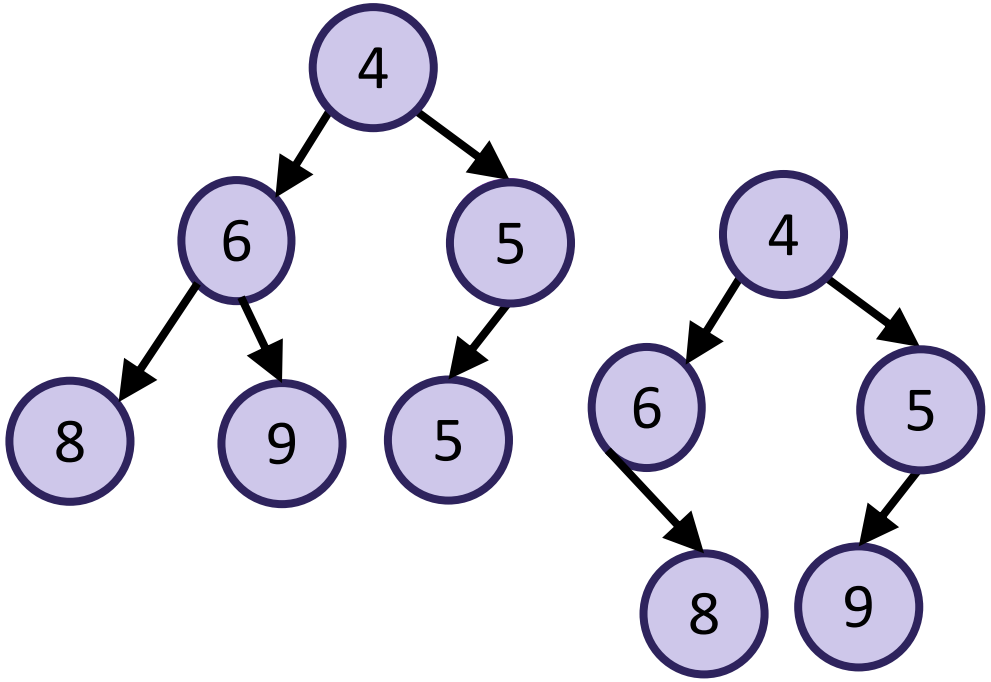
Heaps

- We can still get degenerate trees. From our AVL exploration, we know limiting height is important when we *do* have to search down through the tree
- In AVL, we saw that BST invariant was so strict we couldn't also enforce a strict structure invariant (e.g. exactly balanced)
  - But heap invariant is looser!
  - So we can do a stronger structure invariant

INVARIANT

Heap Structure Invariant

A heap is always a ***complete*** tree.



- A tree is **complete** if:
  - Every row, except possibly the last, is completely full.
  - The last row is filled from left to right (no "gap")

Lecture Outline

- PriorityQueues
  - PriorityQueue ADT
  - PriorityQueue Implementations
- **Binary Heaps**
  - **Binary Heap Idea & Invariants**
  - Binary Heap Implementation Details

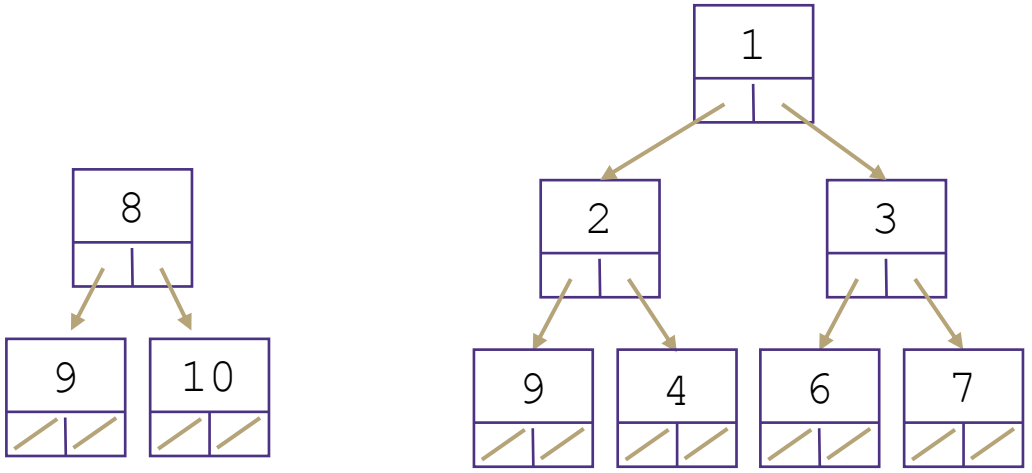
Binary Heap Invariants Summary

- One flavor of heap is a **binary heap**, which is a Binary Tree with the heap invariants (*NOT* a Binary Search Tree)

INVARIANT

Binary Tree

Every node has at most 2 children



INVARIANT

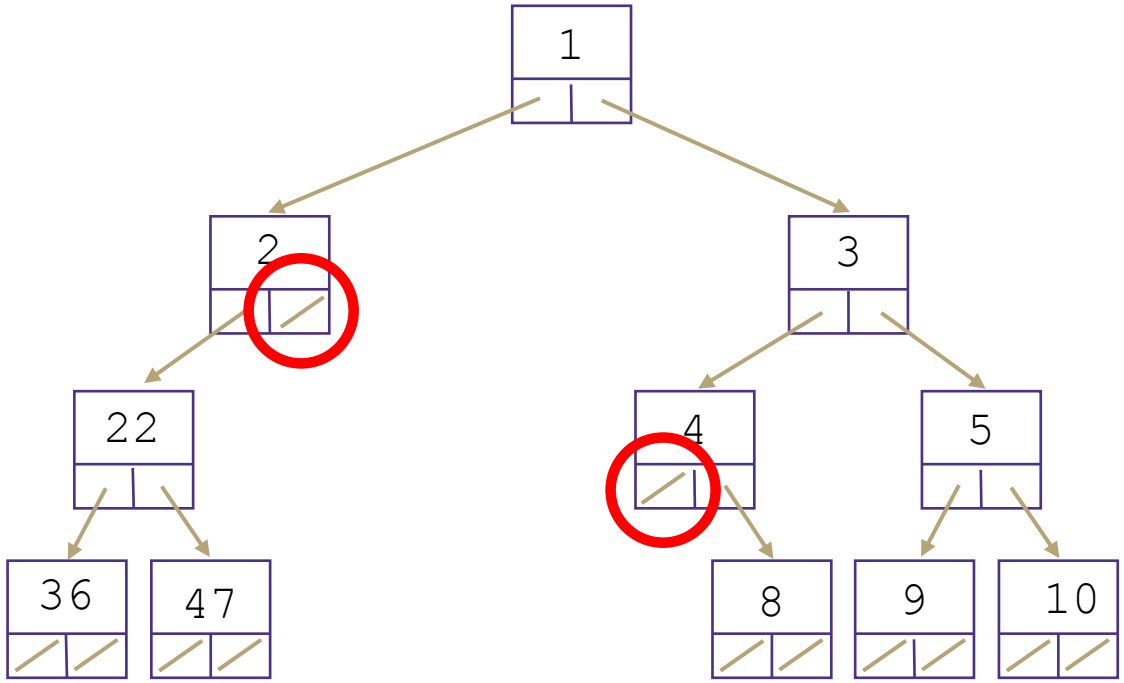
Heap Invariant

Every node is less than or equal to all of its children.

INVARIANT

Heap Structure Invariant

A heap is always a ***complete*** tree.



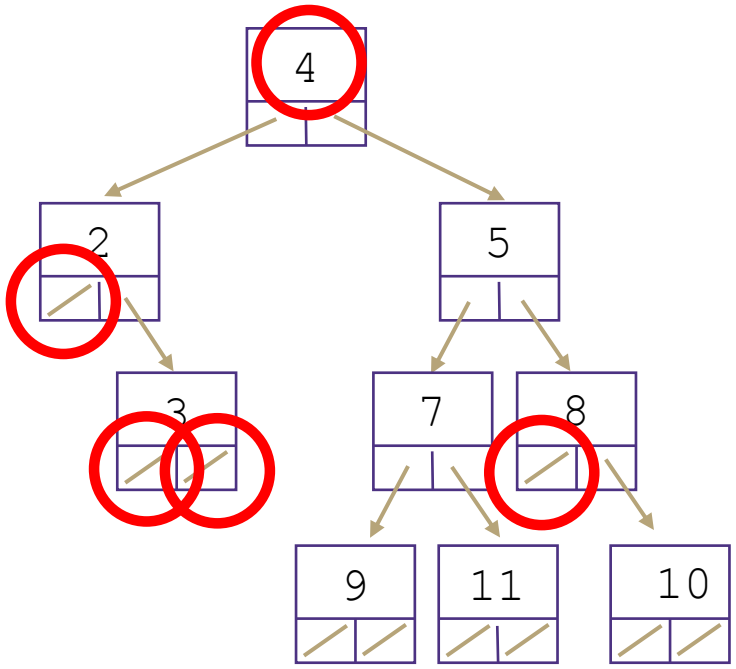
Poll Everywhere

pollev.com/uwcse373

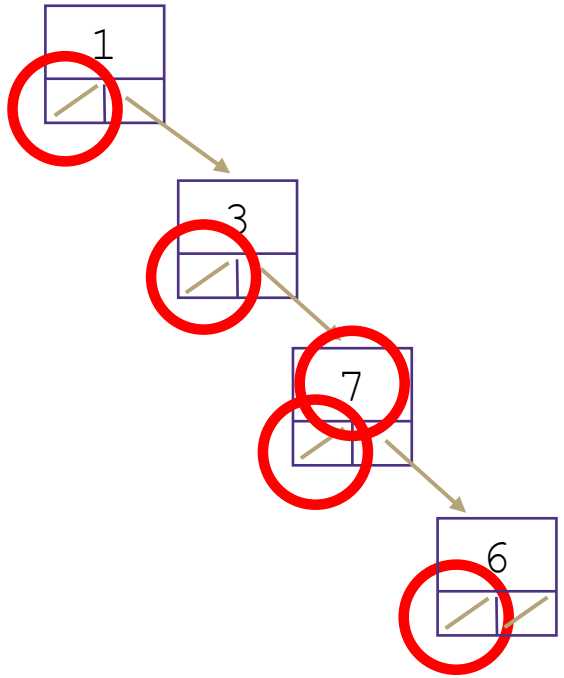
Self-Check: Are these valid heaps?

Binary Heap Invariants:
1. Binary Tree
2. Heap
3. Structure (Complete)

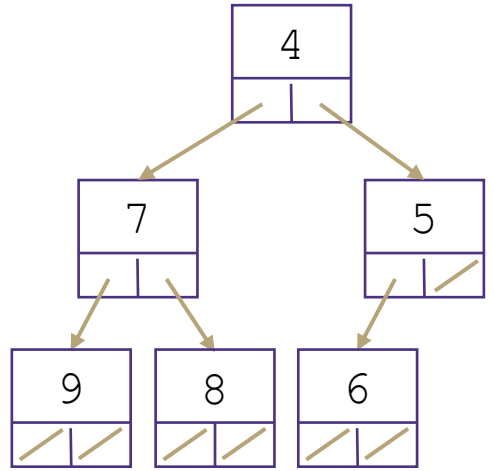
INVALID



INVALID



VALID

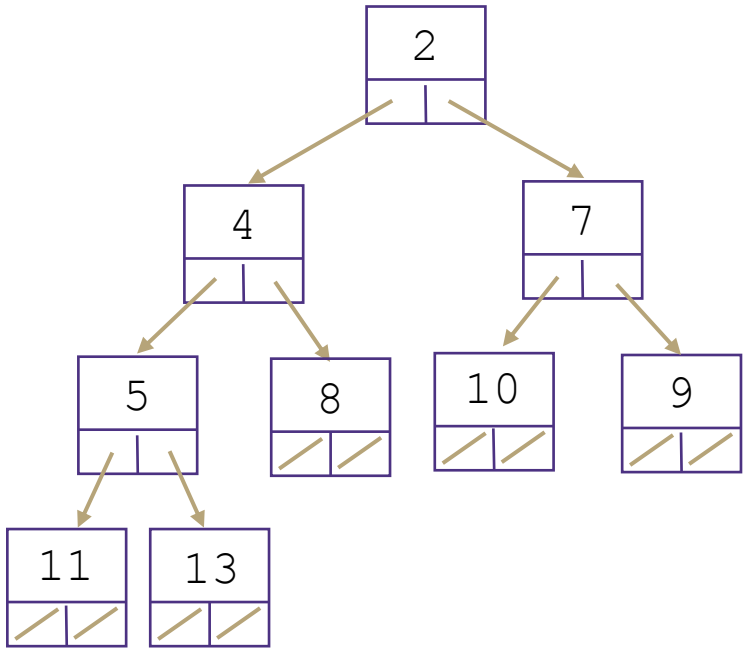


Heap Height

- **A binary heap bounds our height at $\theta(\log n)$ because it's *complete***
  - Although asymptotically the same, a little stricter/better than AVL because no leniency!

This means the runtime to traverse from root to leaf or leaf to root will be log(n) time.

Coming up, we'll see why we might need to do that.

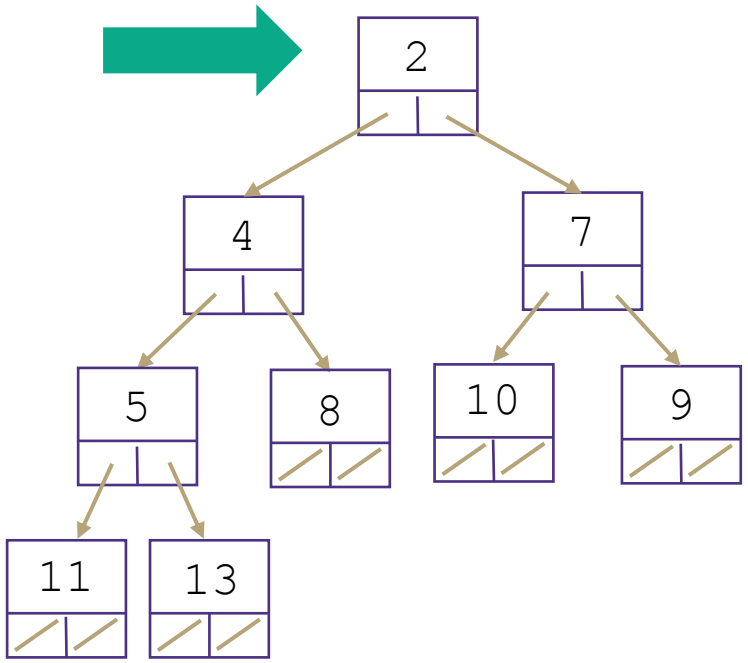


Lecture Outline

- PriorityQueues
  - PriorityQueue ADT
  - PriorityQueue Implementations
- **Binary Heaps**
  - Binary Heap Idea & Invariants
  - **Binary Heap Implementation Details**

Implementing peekMin()

Runtime: Θ(1)



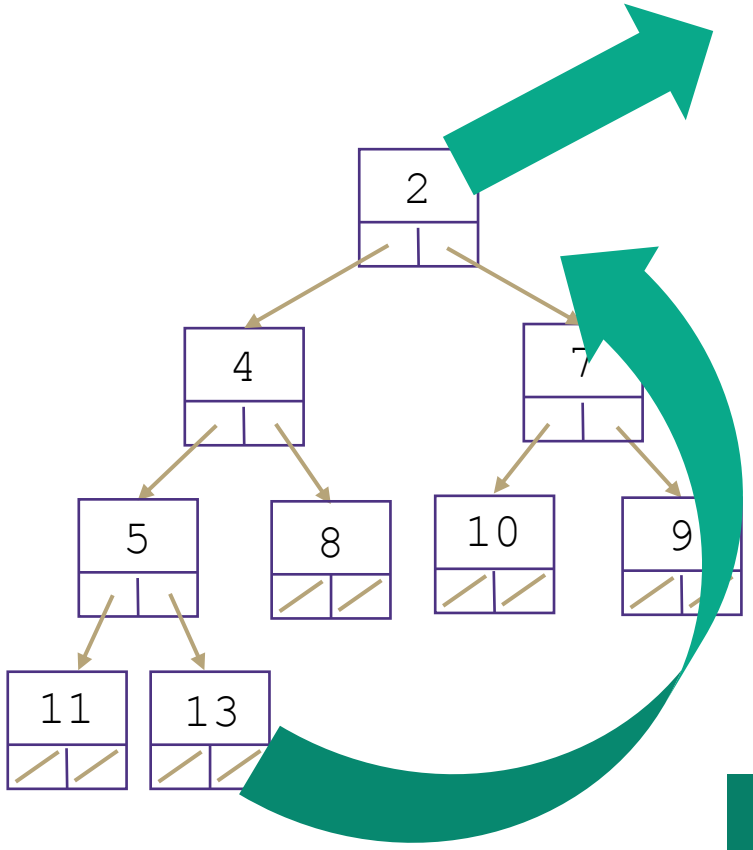
Simply return the value at the root! That's a constant-time operation if we've ever seen one ☺

Implement removeMin()

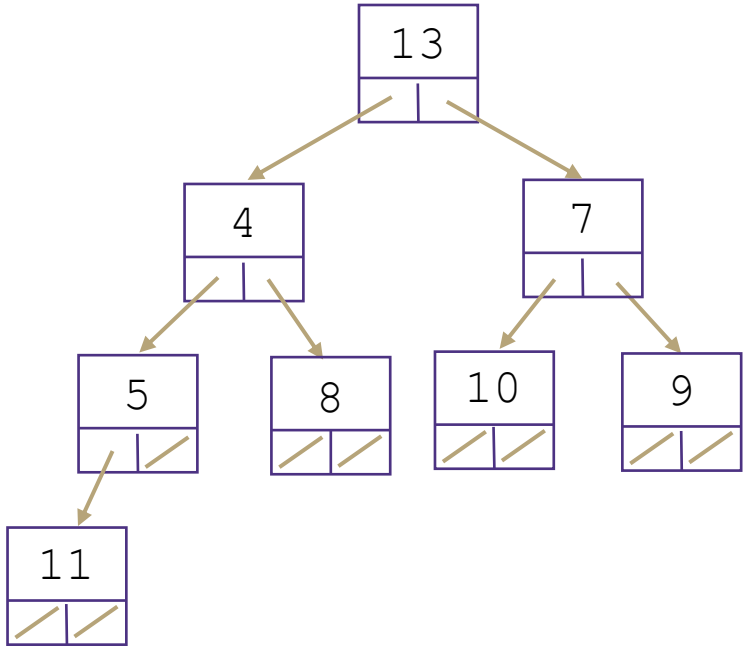
INVARIANT

Heap Structure Invariant

A heap is always a ***complete*** tree.



1) Remove min to return
2) Structure Invariant broken: replace with bottom level right-most node (the only one that can be moved)



INVARIANT

Heap Invariant

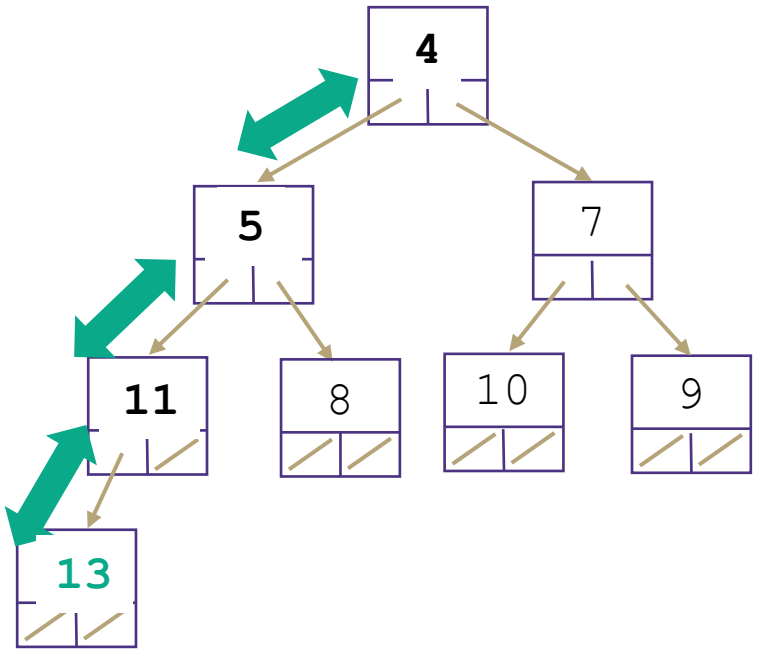
Every node is less than or equal to all of its children.

Structure Invariant restored, but Heap Invariant now broken

Implement removeMin(): percolateDown

3) percolateDown()

Recursively swap parent with **smallest** child until parent is smaller than both children (or at a leaf).



What's the worst-case running time?

- Find last element
- Move it to top spot
- Swap until invariant restored

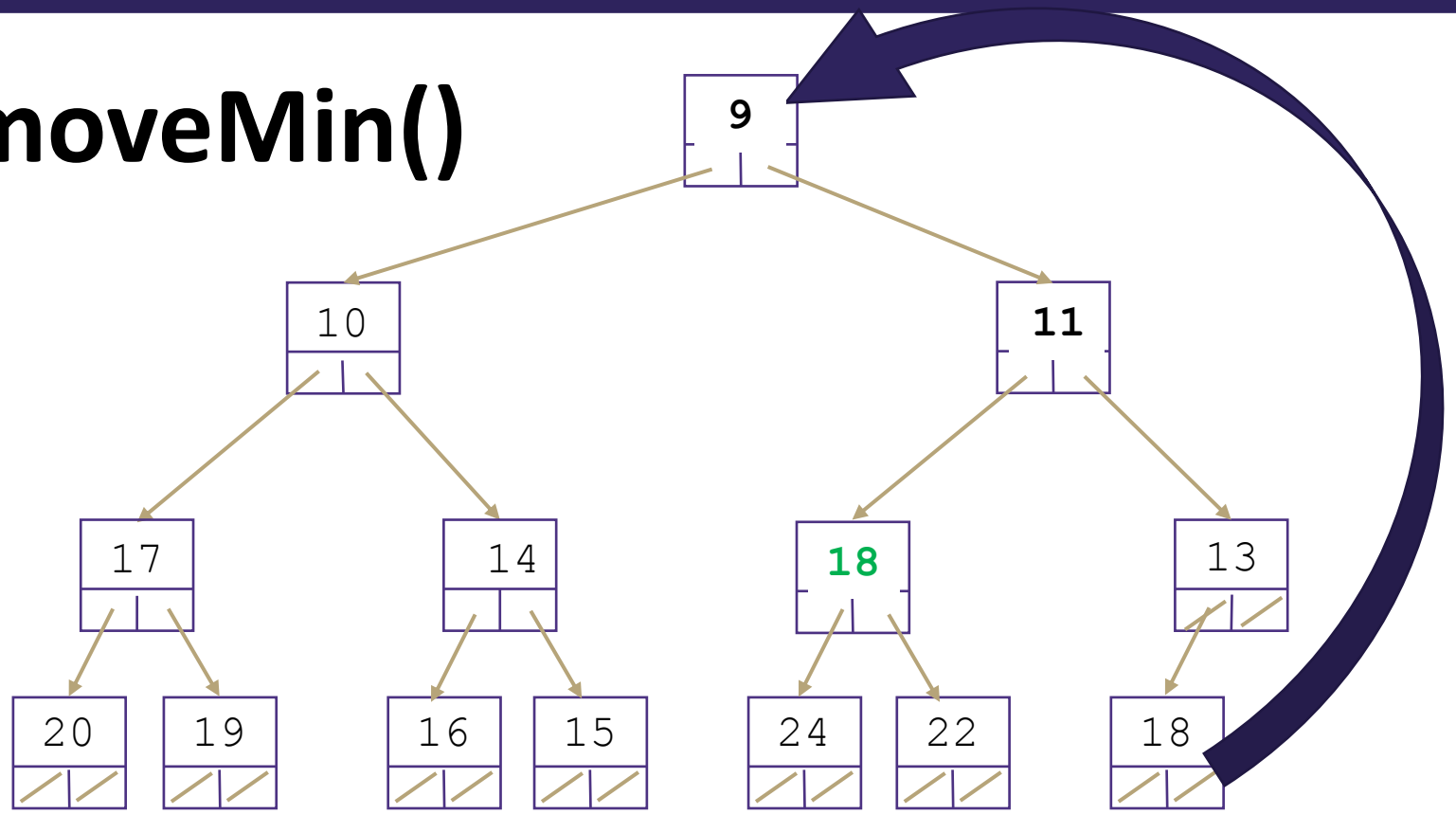
This is why we want to keep the height of the tree small! The height of these tree structures (BST, AVL, heaps) directly correlates with the worst case runtimes

Structure invariant restored, heap invariant restored

Poll Everywhere

pollev.com/uwcse373

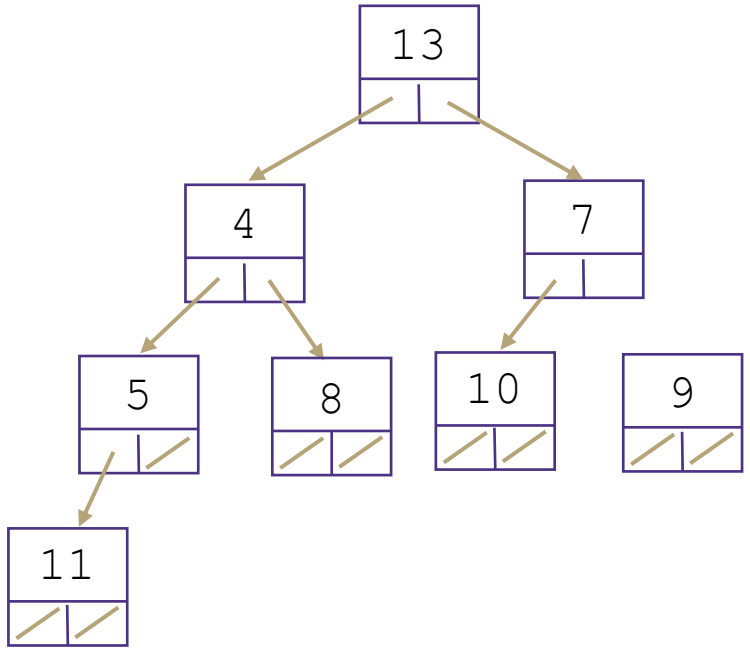
Practice removeMin()



1) Remove min node
2) Replace with bottom level right-most node
3) percolateDown - Recursively swap parent with **smallest** child until parent is smaller than both children (or we're at a leaf).

percolateDown(): Why Smallest Child?

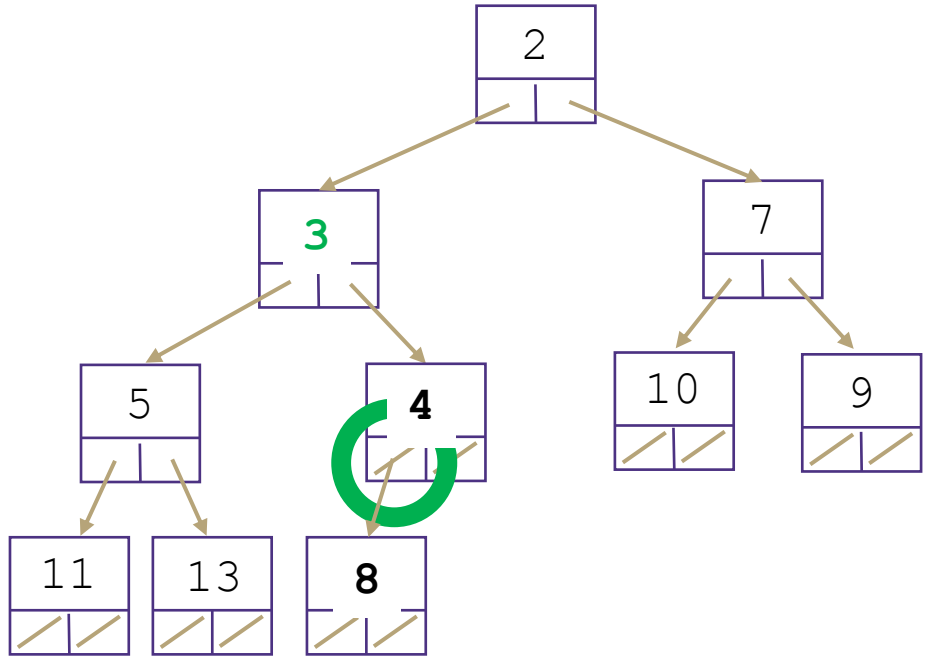
- Why does `percolateDown` swap with the smallest child instead of just any child?



- If we swap 13 and 7, the heap invariant isn't restored!
- 7 is greater than 4 (it's not the smallest child!) so it will violate the invariant.

Implementing add()

- add() Algorithm:
  - Insert a node on the bottom level that ensure no gaps
  - Fix heap invariant by new method: `percolateUp()`
    - Swap with parent, until your parent is smaller than you (or you're the root).



Worst case runtime is similar to removeMin and percolateDown – might have to do log(n) swaps, so the worst-case runtime is Theta(log(n))

MinHeap Runtimes

removeMin():

- remove root node
- Find last node in tree and swap to top level
- Percolate down to fix heap invariant

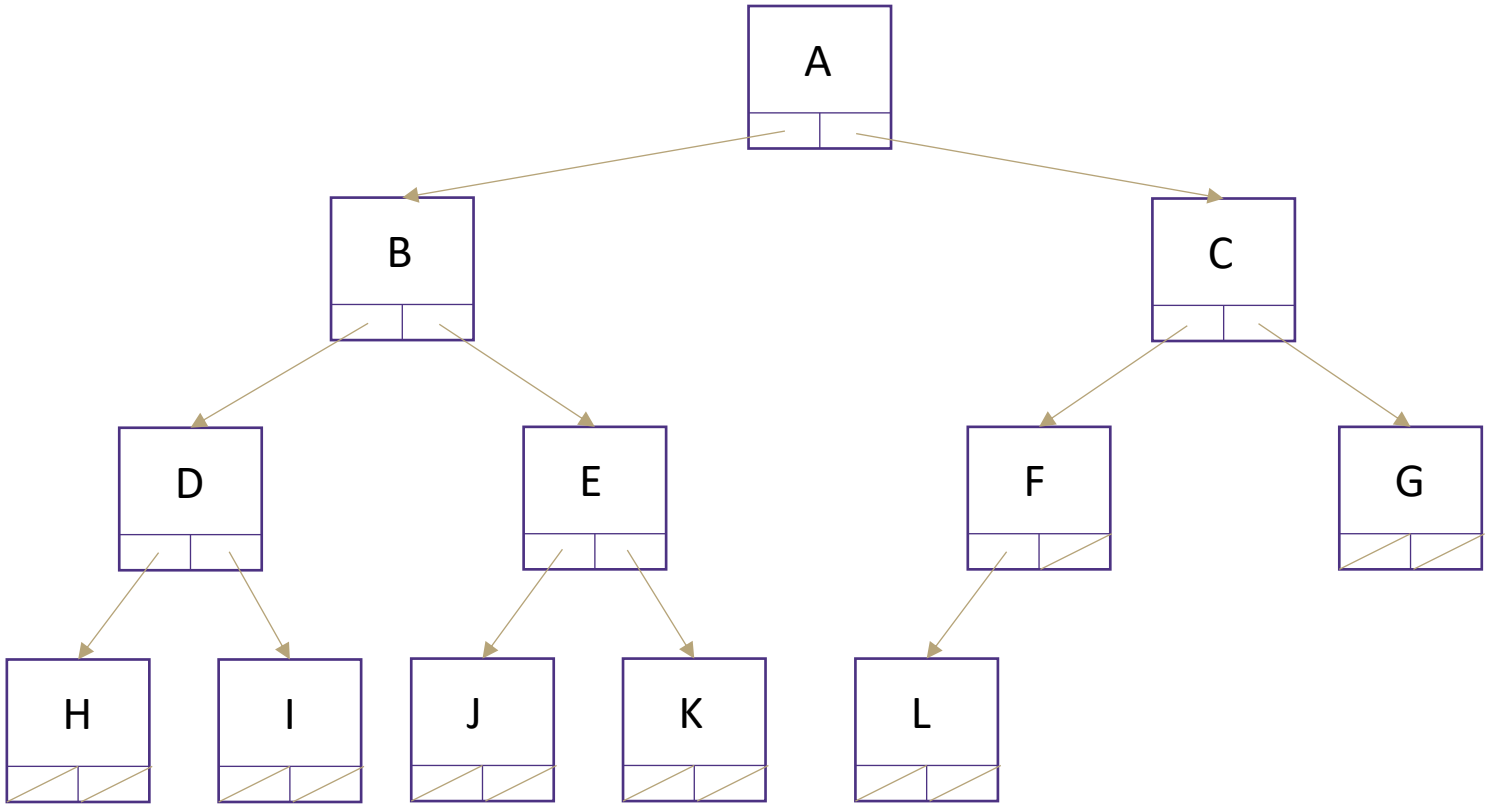
add():

- Insert new node into next available spot
- Percolate up to fix heap invariant

- Finding the last node/next available spot is the hard part.
- You can do it in $\Theta(\log n)$ time on complete trees, with some extra class variables, but it's not fun

- Fortunately, there's a much better way!

Implement Heaps with an Array



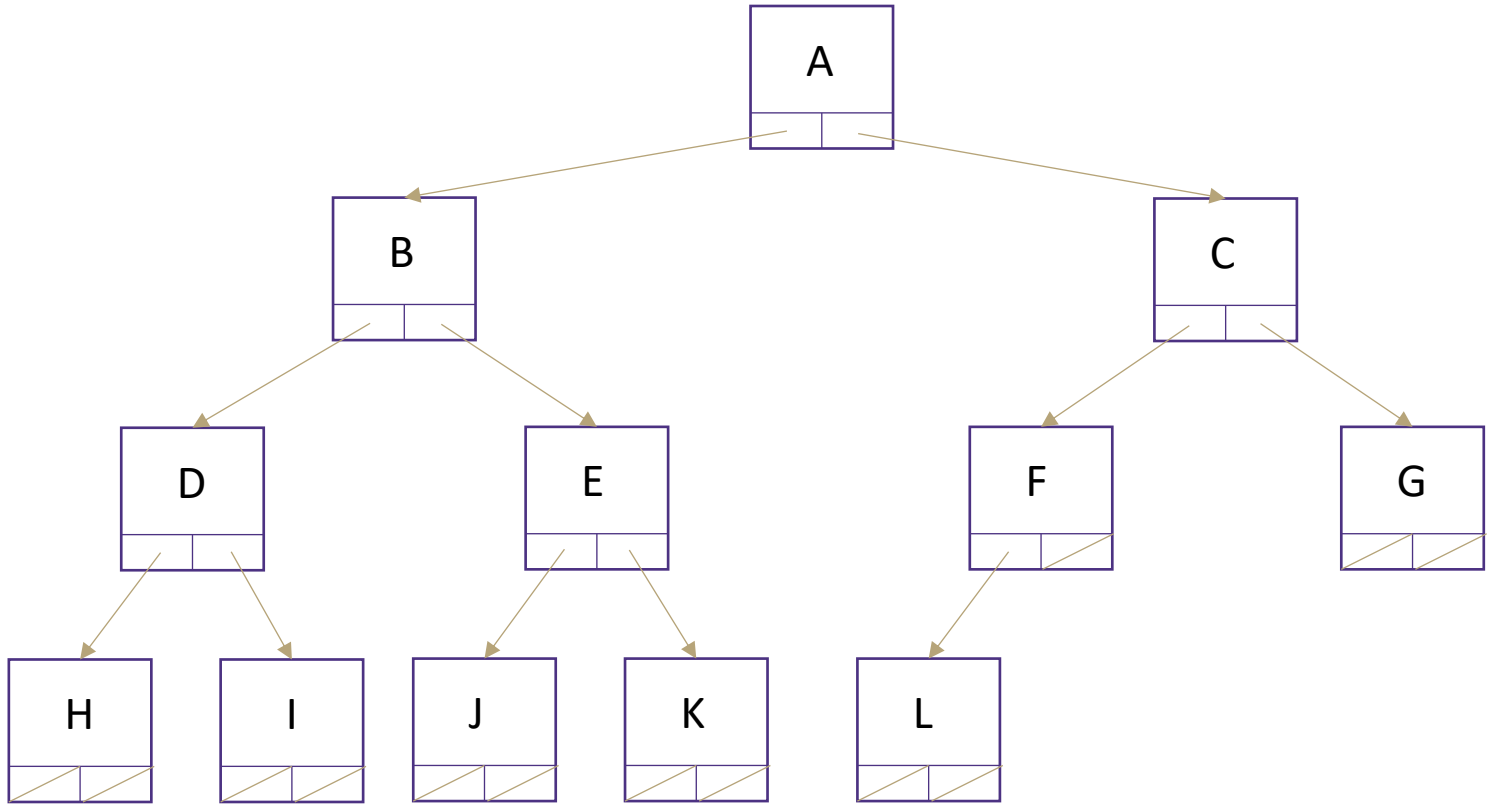
Fill array in **level-order** from left to right

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

We map our binary-tree representation of a heap into an array implementation where you fill in the array in level-order from left to right.

The array implementation of a heap is what people actually implement, but the tree drawing is how to think of it conceptually. **Everything we've discussed about the tree representation is still true**!

Implement Heaps with an Array



Fill array in **level-order** from left to right

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

How do we find the minimum node?

$$peekMin() = arr[0]$$

How do we find the last node?

$$lastNode() = arr[size - 1]$$

How do we find the next open space?

$$openSpace() = arr[size]$$

How do we find a node's left child?

$$leftChild(i) = 2i + 1$$

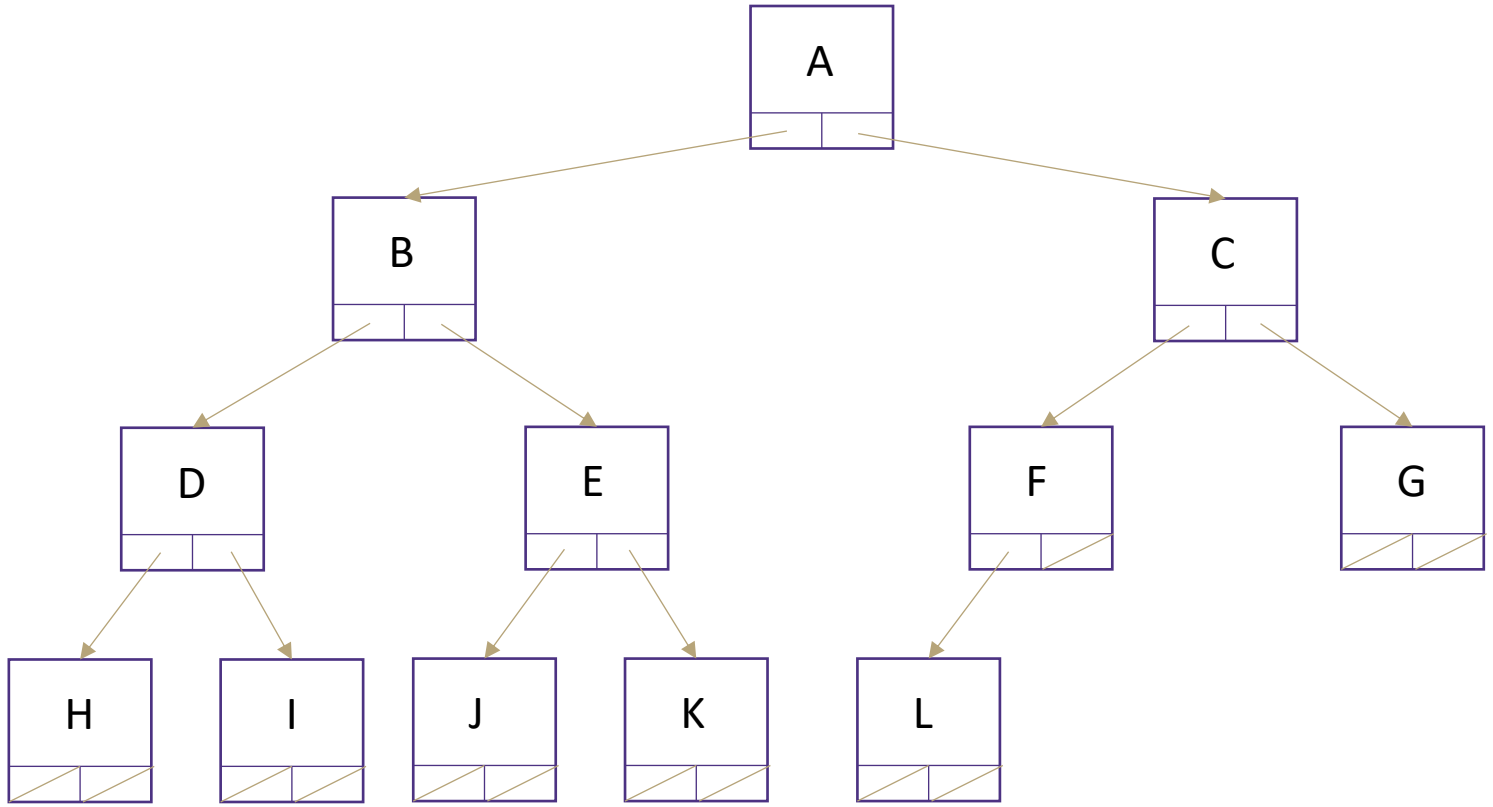
How do we find a node's right child?

$$rightChild(i) = 2i + 2$$

How do we find a node's parent?

$$parent(i) = \frac{(i-1)}{2}$$

Implement Heaps with an Array



Fill array in **level-order** from left to right

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

How do we find the minimum node?

$$peekMin() = arr[1]$$

How do we find the last node?

$$lastNode() = arr[size]$$

How do we find the next open space?

$$openSpace() = arr[size + 1]$$

How do we find a node's left child?

$$leftChild(i) = 2i$$

How do we find a node's right child?

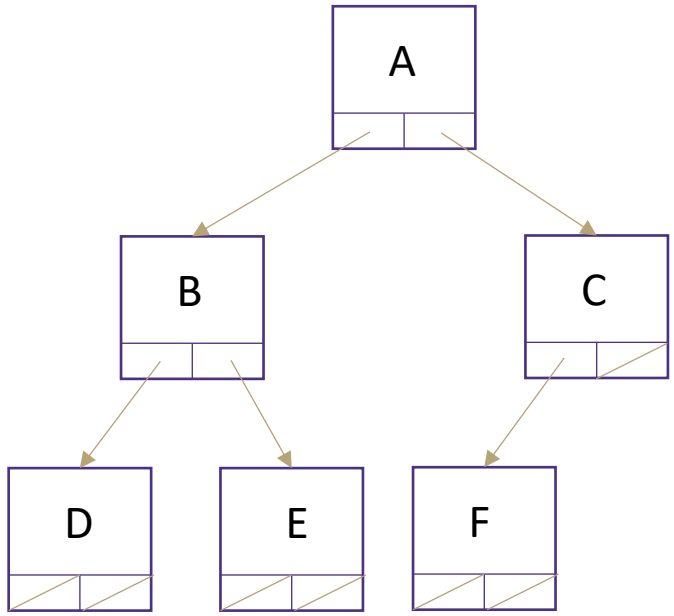
$$rightChild(i) = 2i + 1$$

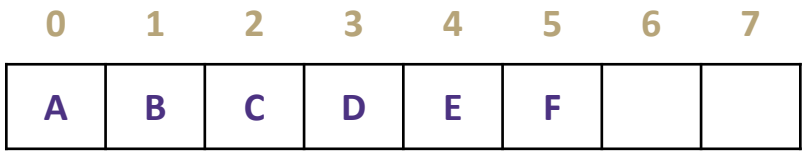
How do we find a node's parent?

$$parent(i) = \frac{i}{2}$$

CORRECTED AFTER LECTURE

Heap Implementation Runtimes





Implementation	add	removeMin	Peek
Array-based heap	worst: $\Theta(\log n)$ in-practice: $\Theta(1)$	worst: $\Theta(\log n)$ in–practice: $\Theta(\log n)$	$\Theta(1)$

We've matched the **asymptotic worst-case** behavior of AVL trees.

But we're actually doing better!

- The constant factors for array accesses are better.
- The tree can be a constant factor shorter because of stricter height invariants.
- In-practice case for add is really good.
- A heap is MUCH simpler to implement.

AVL vs Heaps: Good For Different Situations

- - The really amazing things about heaps over AVL implementations are the constant factors (e.g. 1.2n instead of 2n) and the sweet sweet Theta(1) in-practice `add` time.

- - The really amazing things about AVL implementations over heaps is that AVL trees are absolutely sorted, and they guarantee worst-case be able to find (contains/get) in Theta(log(n)) time.

- If heaps have to implement methods like contains/get/ (more generally: finding a particular value inside the data structure) – it pretty much just has to loop through the array and incur a worst case Theta(n) runtime.
- Heaps are stuck at Theta(n) runtime and we can't do anything more clever…. aha, just kidding.. unless…?