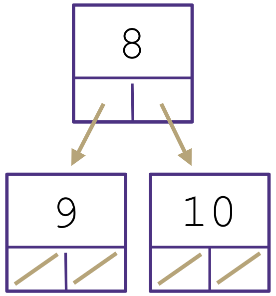
Lecture 14: Heap Percolations

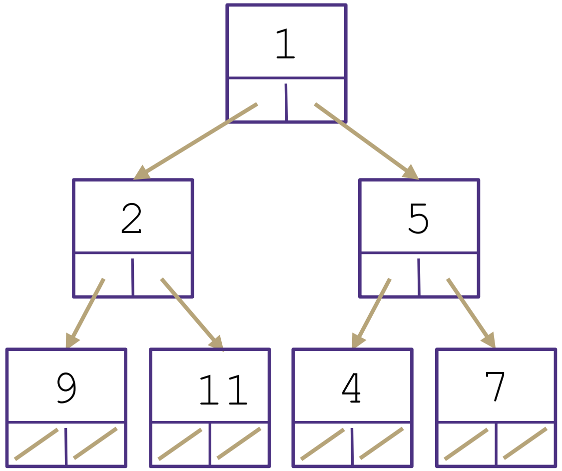
CSE 373 Data Structures and Algorithms

Warm Up

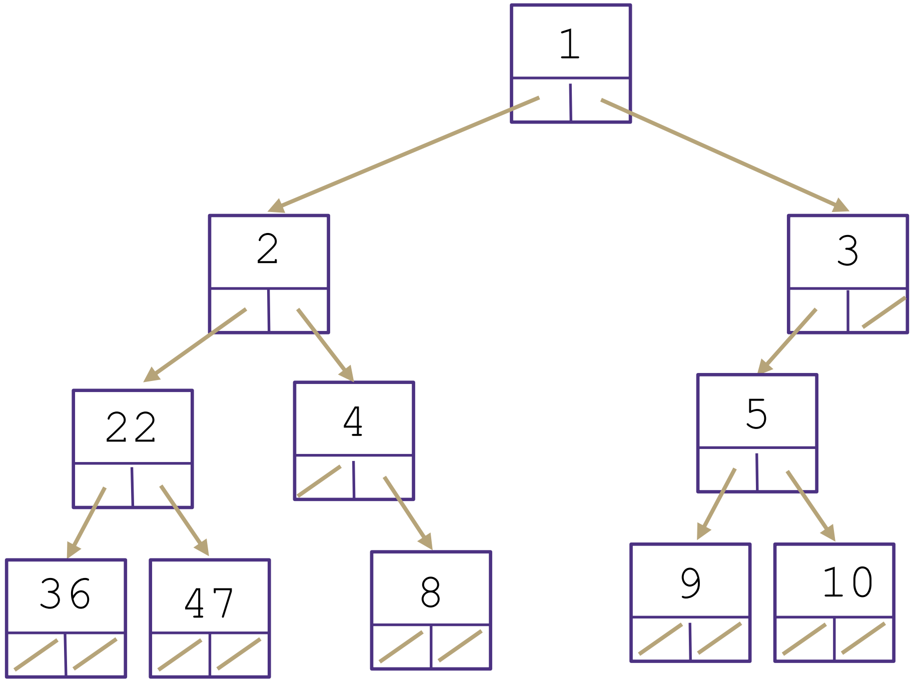
Are the following trees valid min heaps?



Valid



Invalid



Invalid

Announcements

P2 due today!

Midterm out this Friday – due 1 week later

NO LATE ASSIGNMENTS ACCEPTED

- Group assignment
- Open note/ open internet, <u>closed course staff</u>
- intended to take 1 person 1 hour
- Topics:
  - ADTs
  - Code Modeling
  - Big O, Big Theta, Big Omega
  - Case Analysis
  - Recurrences
  - Master Theorem & Tree Method
  - Hashing
  - BSTs & AVls
  - Heaps
  - Design Decisions

Sorry about OH – we doing out best!

What's NOT on the midterm:

- AVL Rotations
- Big O Proofs (C and N0 style)
- Summation Identities (Limited algebra)

Come to the Midterm Review!

- Thursday (tomorrow) evening 5:30-7:30 pm PST

Mid Quarter Surveys

- Lecture
- Section
- 90% response rate on all- 1 point EC for everyone!

Your toolbox so far...

- ADT
  - List – flexibility, easy movement of elements within structure
  - Stack – optimized for first in last out ordering
  - Queue – optimized for first in first out ordering
  - Dictionary (Map) – stores two pieces of data at each entry **<- It's all about data baby!**

SUPER common in comp sci
- Databases
- Network router tables
- Compilers and Interpreters

- Data Structure Implementation
  - Array – easy look up, hard to rearrange
  - Linked Nodes – hard to look up, easy to rearrange
  - Hash Table – constant time look up, no ordering of data
  - BST – efficient look up, possibility of bad worst case
  - AVL Tree – efficient look up, protects against bad worst case, hard to implement

Priority Queue / heaps roadmap

- PriorityQueue ADT

- PriorityQueue implementations with current toolkit

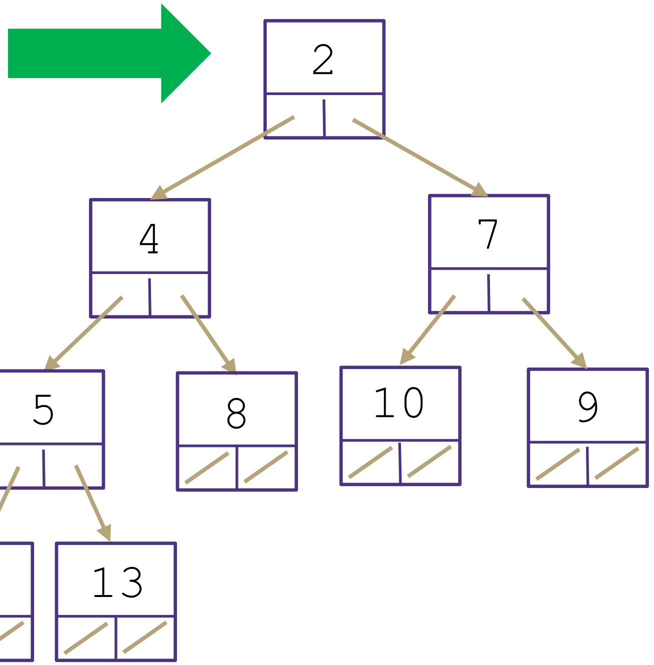
- Binary Heap idea + invariants

<u>- Binary Heap methods</u>

- Binary Heap implementation details

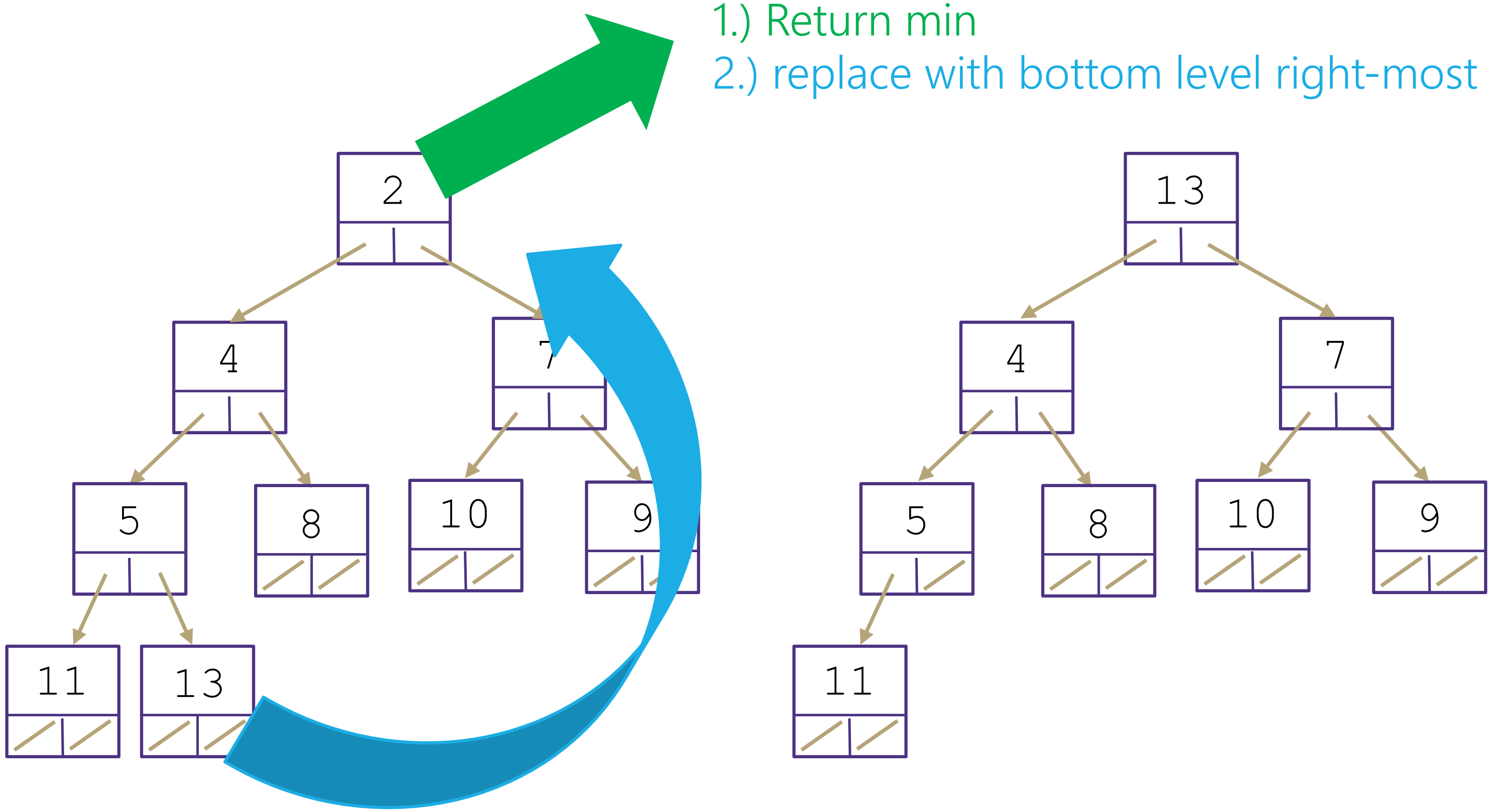
Implementing peekMin()

Runtime: $\Theta(1)$



Implementing removeMin()

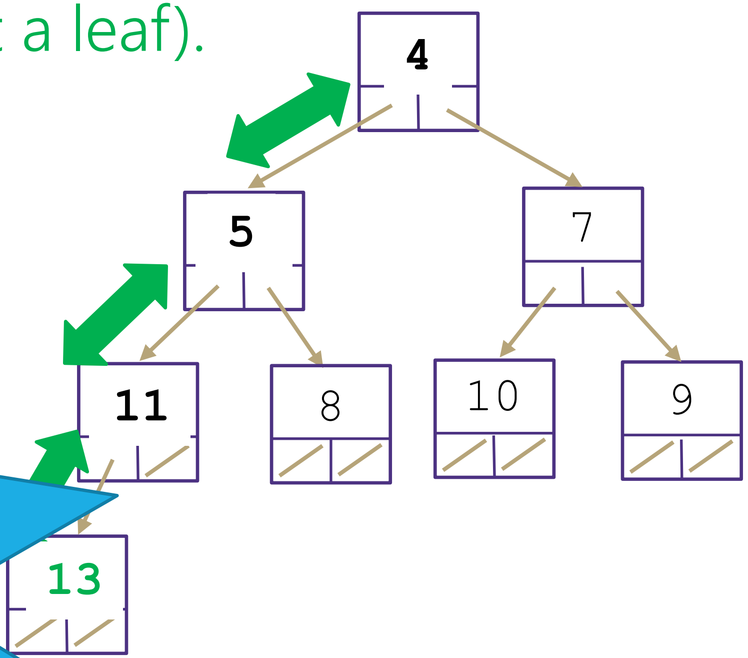
1.) Return min
2.) replace with bottom level right-most node



Structure invariant restored, heap invariant broken

Implementing removeMin() - percolateDown

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



This is a big idea! (height of all these tree DS correlates w worst case runtimes – we want to design our trees to have reasonably small height!)

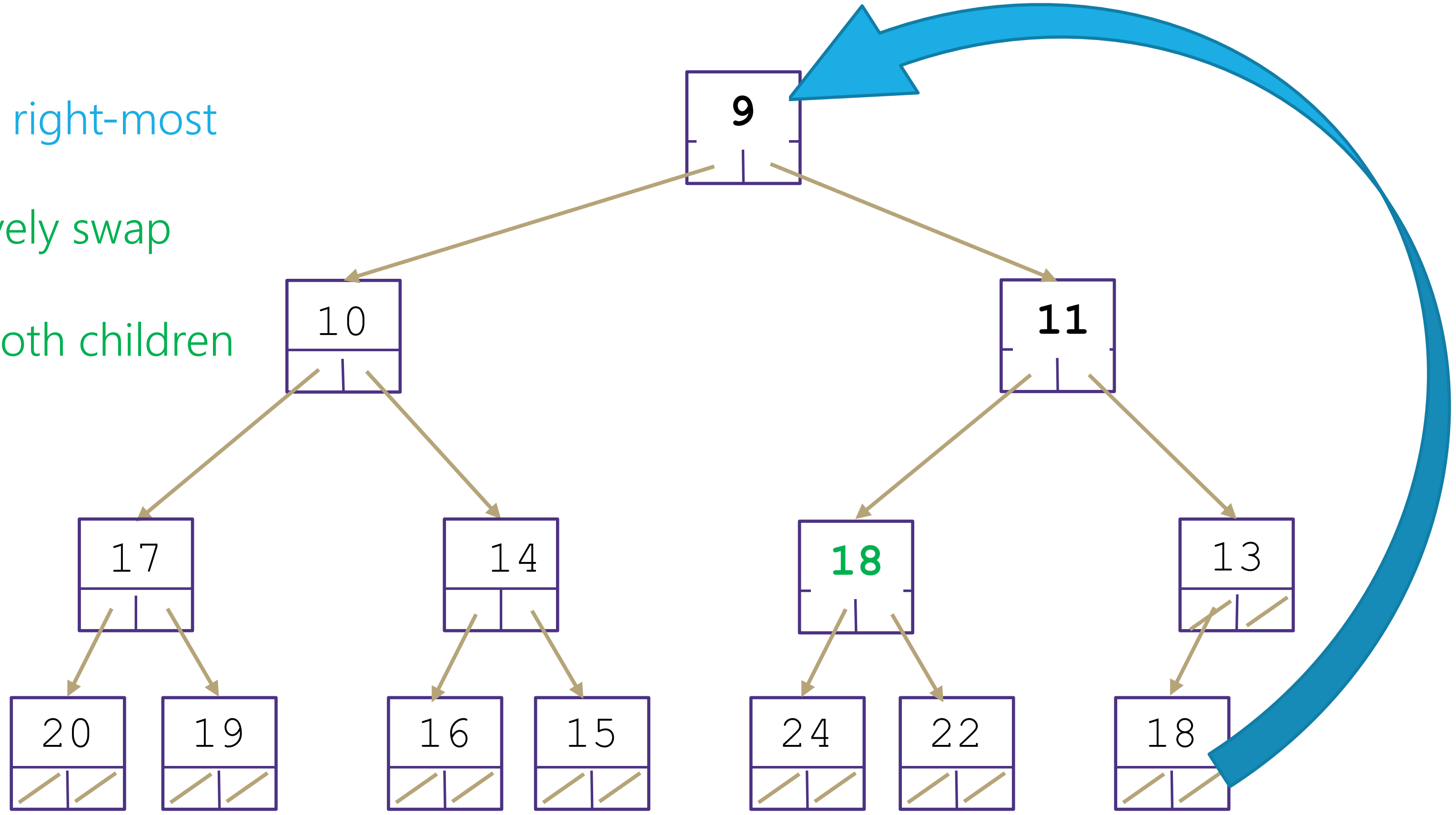
Structure invariant restored, heap invariant restored

What's the worst-case running time?
Have to:
Find last element
Move it to top spot
Swap until invariant restored
(how many times do we have to swap?)

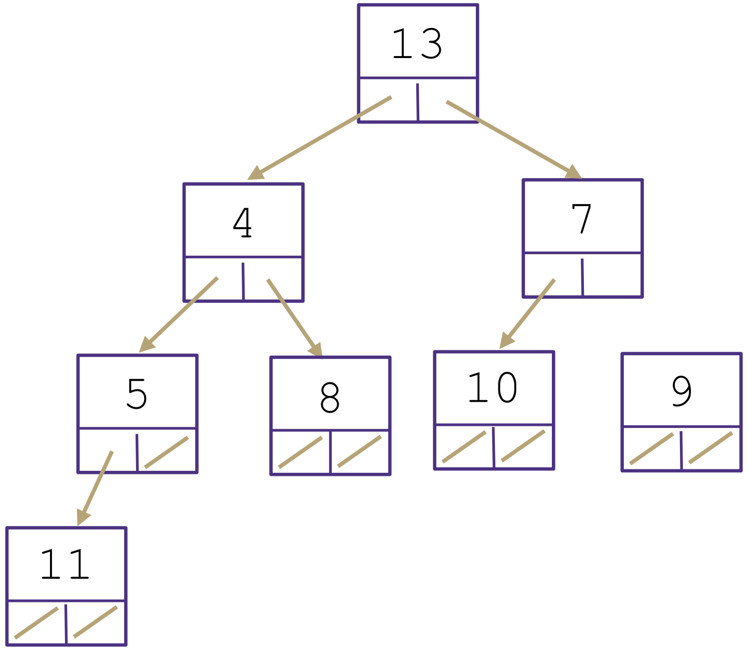
this is why we want to keep the height of the tree small! The height of these tree structures (BST, AVL, heaps) directly correlates with the worst case runtimes

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).



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:

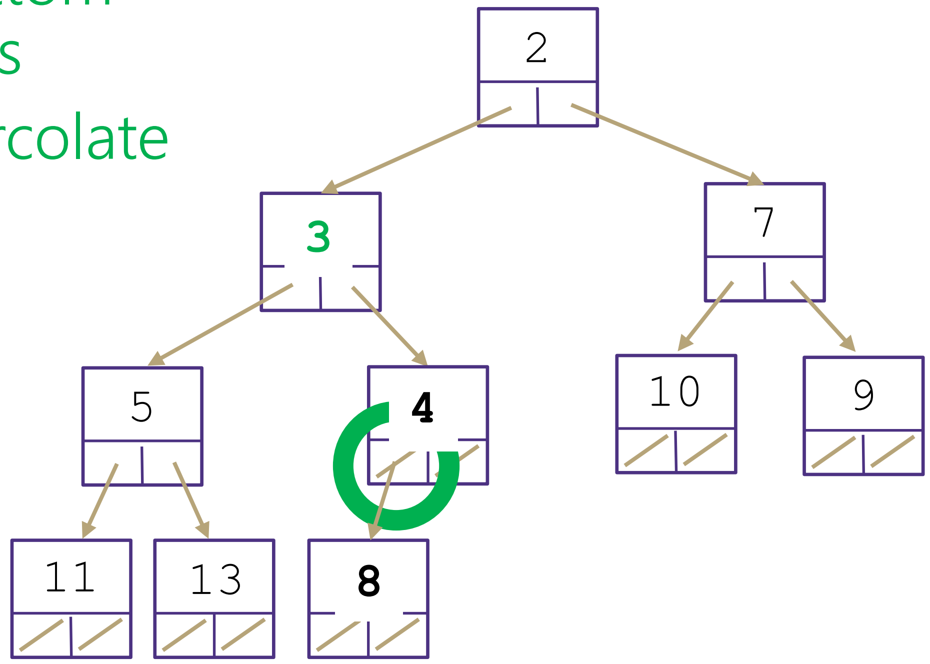
- Insert a node on the bottom level that ensure no gaps
- Fix heap invariant by percolate **UP**

i.e. 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))

Practice: Building a minHeap

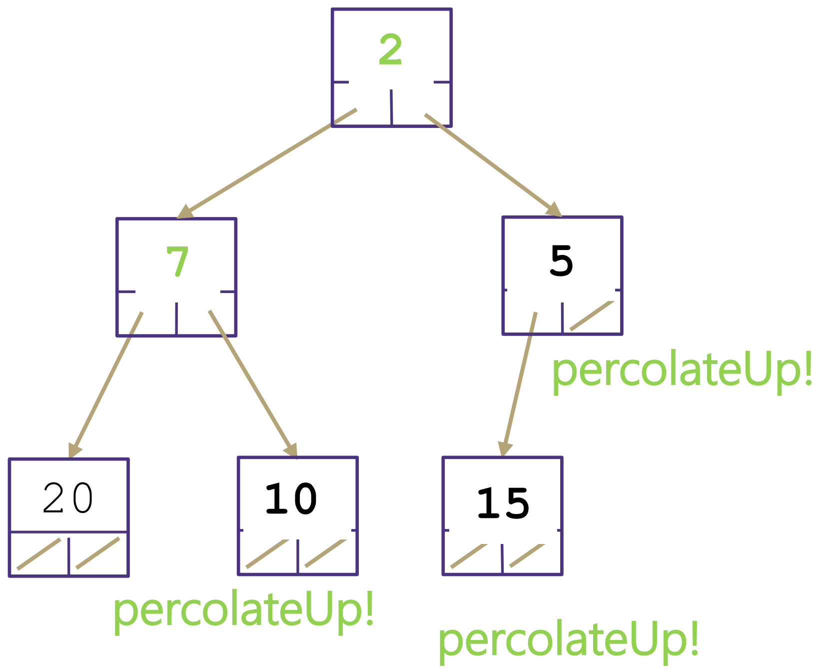
Construct a Min Binary Heap by adding the following values in this order:

5, 10, 15, 20, 7, 2

Add() Algorithm:
- 1.) Insert a node on the bottom level that ensures no gaps
- 2.)Fix heap invariant by percolate **UP**

i.e. swap with parent, until your parent is smaller than you (or you're the root).

Min Binary Heap Invariants
1. Binary Tree – each node has at most 2 children
2. Min Heap – each node's children are larger than itself
3. Level Complete - new nodes are added from left to right completely filling each level before creating a new one



minHeap runtimes

removeMin():

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

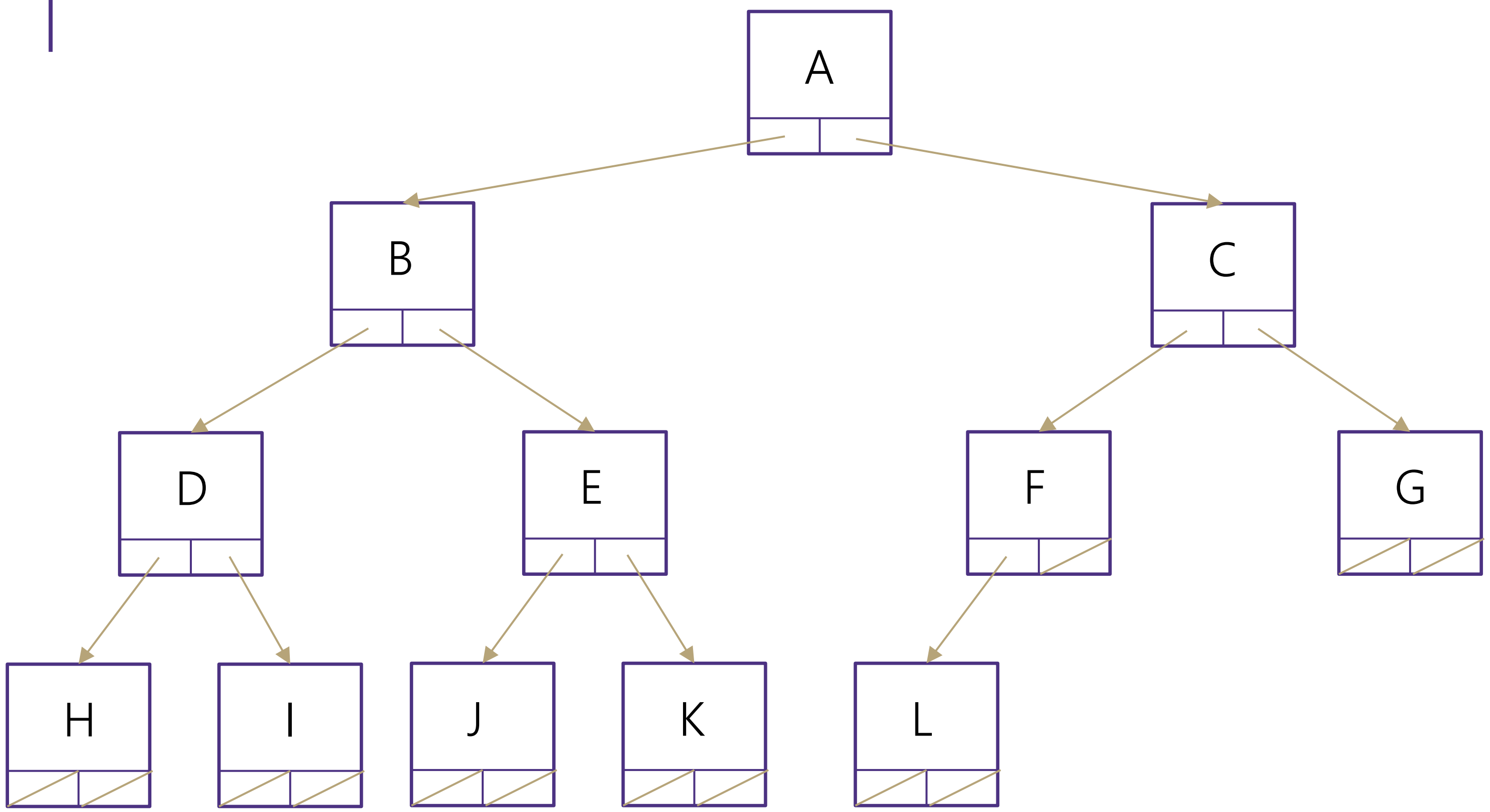
add():

- 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

And there's a much better way!

Implement Heaps with an array



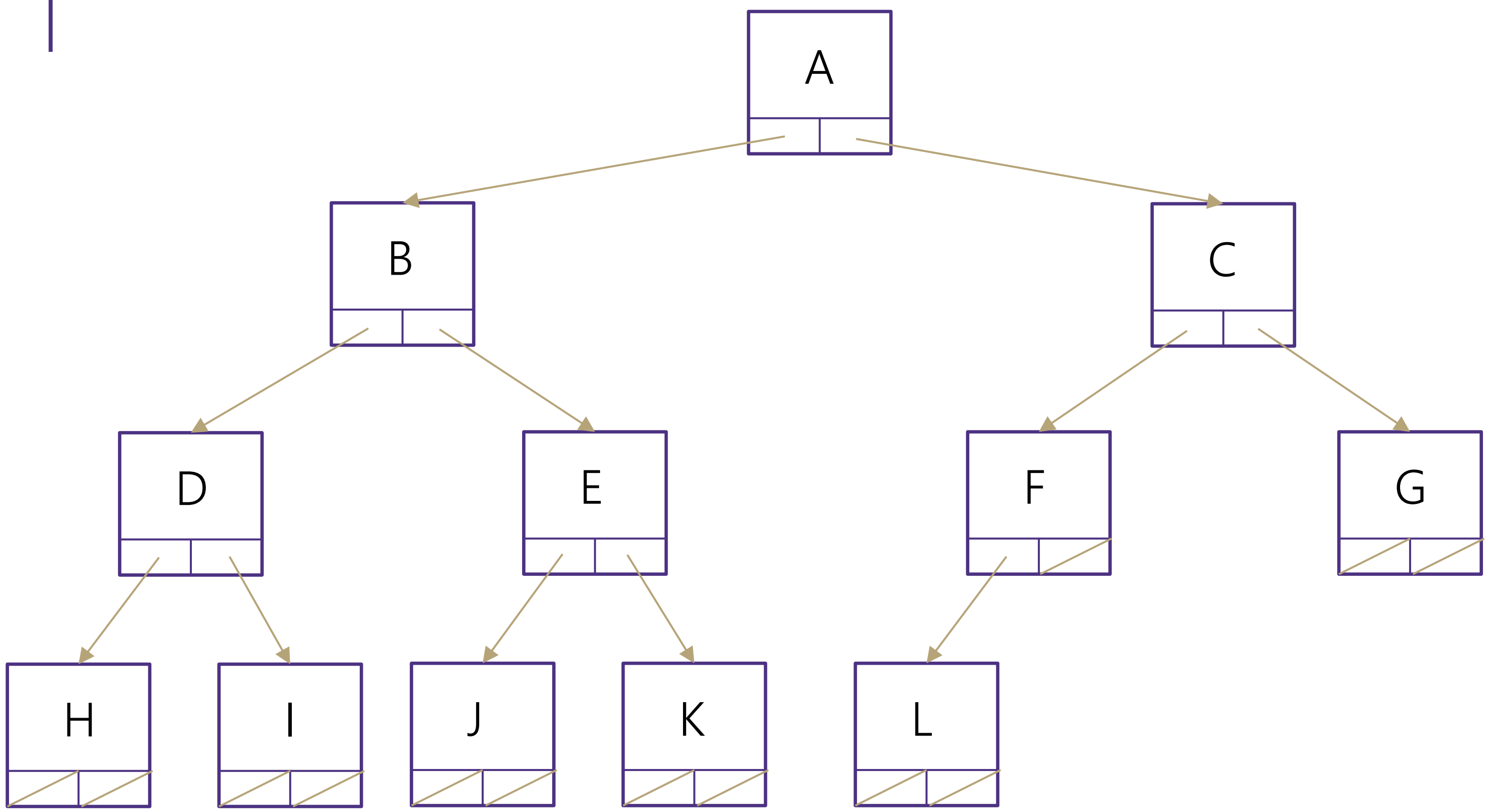
We map our binary-tree representation of a heap into an array implementation where you fill in the array in level-order from left to right.

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 still is true!

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		

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?

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

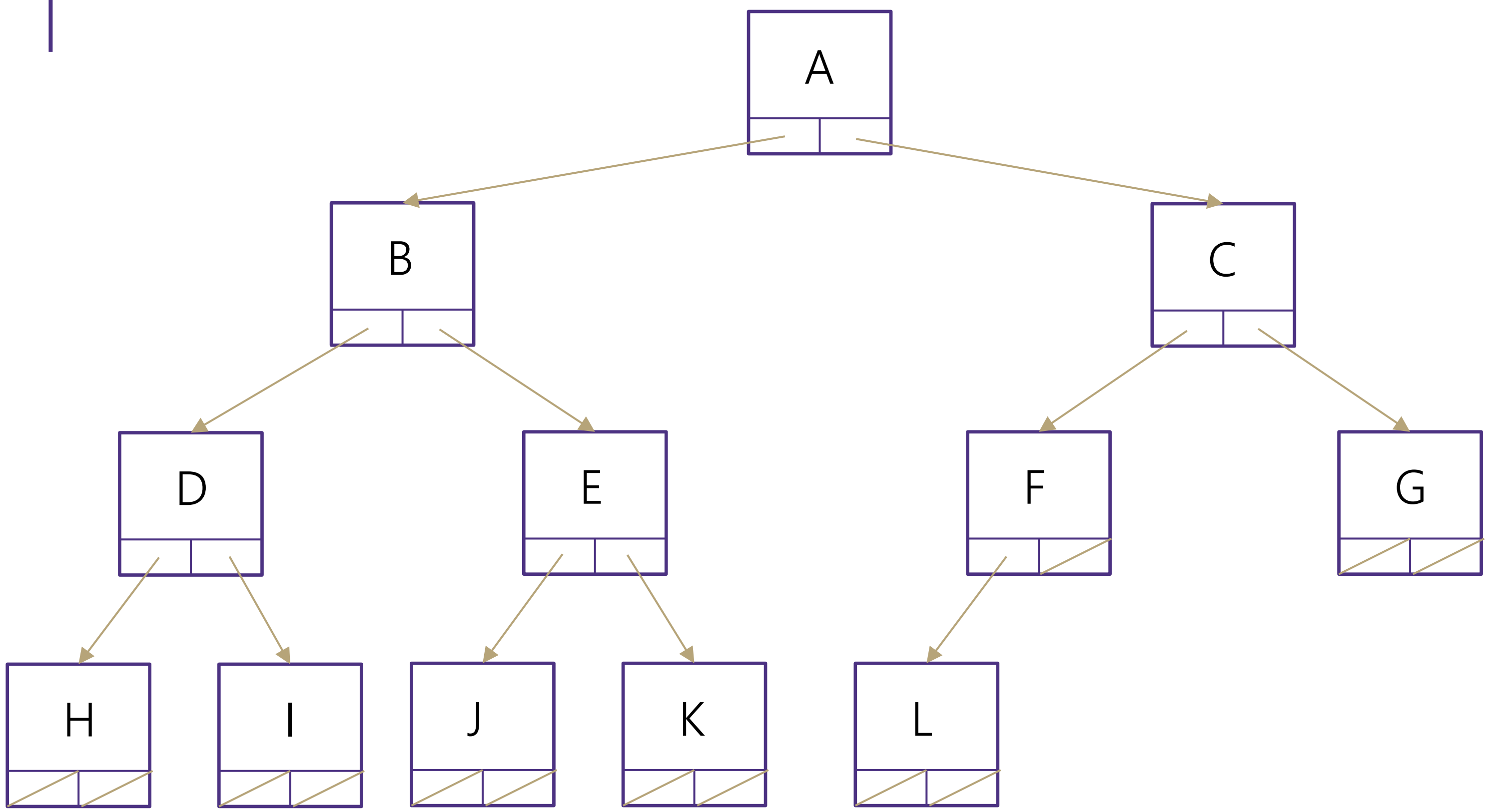
How do we find a node's right child?

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

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?

$$leftChild(i) = 2i$$

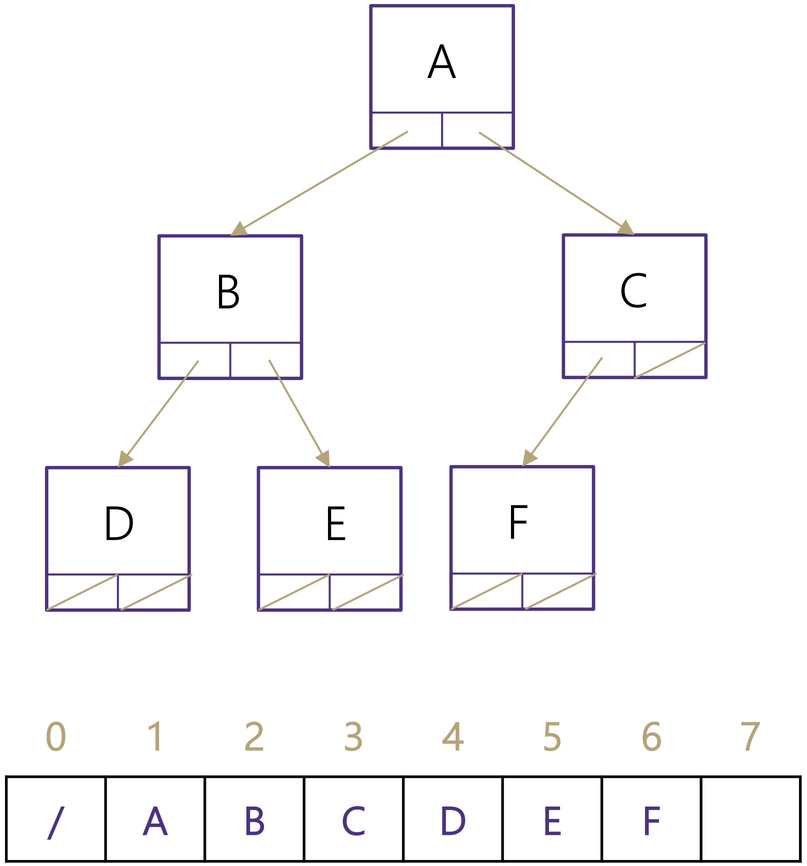
How do we find a node's right child?

$$rightChild(i) = 2i$$

How do we find a node's parent?

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

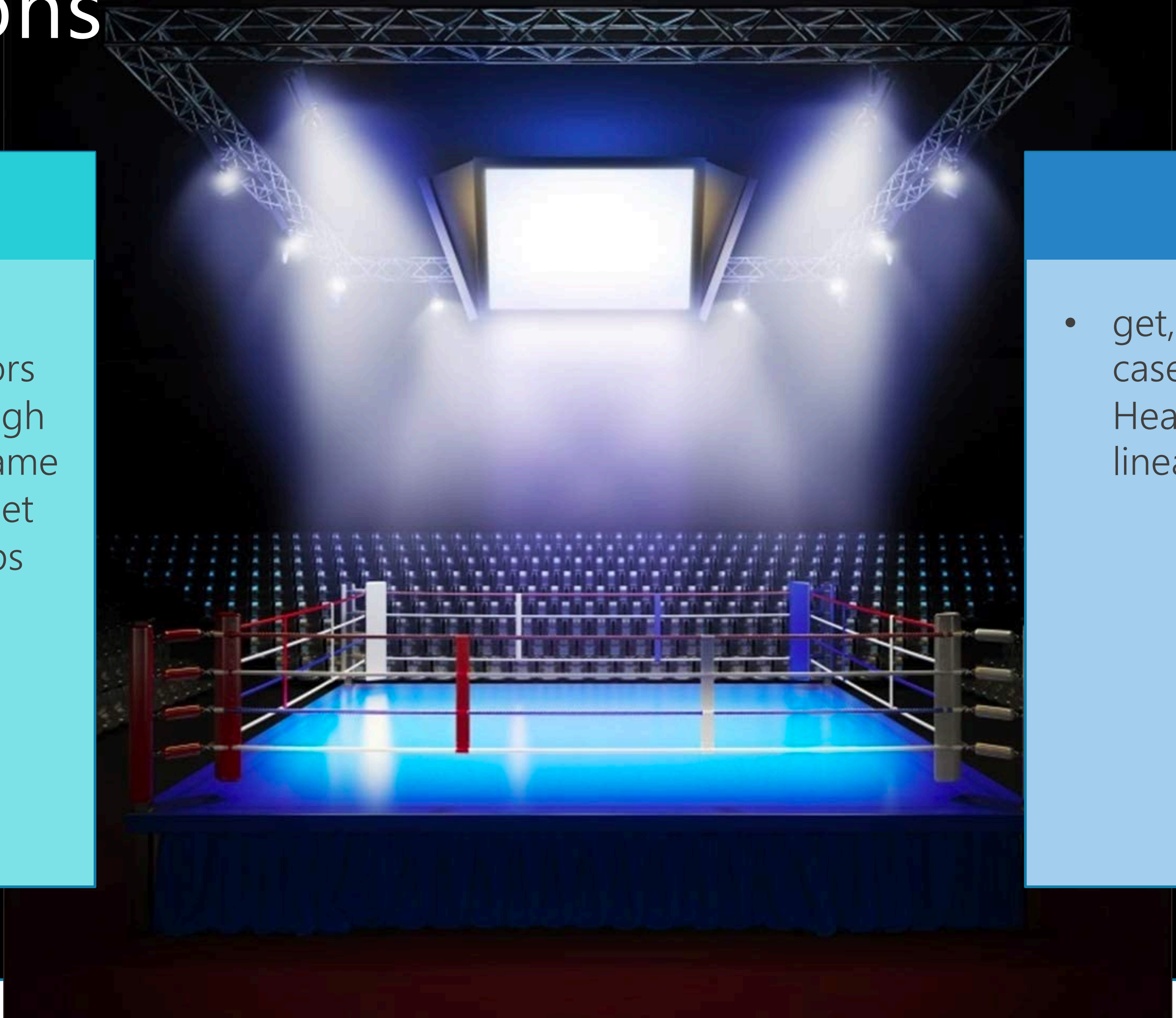
Array-Implemented MinHeap Runtimes



Operation	Case	Runtime
`removeMin()`	best	Θ(1)
	worst	Θ(log n)
	in practice	Θ(log n)
`add(key)`	best	Θ(1)
	worst	Θ(log n)
	in practice	Θ(1)
`peekMin()`	all cases	Θ(1)

- With array implementation, heaps match runtime of finding min in AVL trees
- But better in many ways!
  - Constant factors: array accesses give contiguous memory/spatial locality, tree constant factor shorter due to stricter height invariant
  - In practice, add doesn’t require many swaps
  - WAY simpler to implement!

AVL vs Heaps: Good For Different Situations



HEAPS

- removeMin: much better constant factors than AVL Trees, though asymptotically the same
- add: in-practice, sweet sweet $\Theta(1)$ (few swaps usually required)

↓

PriorityQueue

AVL TREES

- get, containsKey: worst-case (log n) time (unlike Heap, which has to do a linear scan of the array)

↓

Map/Set

Project 3

Build a heap! Alongside hash maps, heaps are one of the most useful data structures to know – and pop up many more times this quarter!

- You'll also get practice using multiple data structures together to implement an ADT!
- Directly apply the invariants we've talked so much about in lecture! Even has an invariant checker to verify this (a *great* defensive programming technique!)

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

changePriority(item, priority) *– update the priority of an element*

contains(item) *– check if an element exists in the priority queue*

Project 3 Tips

Project 3 adds `changePriority` and `contains` to the PriorityQueue ADT, which aren't efficient on a heap alone

You should utilize an extra data structure for `changePriority`!

- Doesn't affect *correctness* of PQ, just runtime. Please use a built-in Java collection instead of implementing your own (although you could in theory).

`changePriority` Implementation Strategy:

- implement without regards to efficiency (without the extra data structure) at first
- analyze your code's runtime and figure out which parts are inefficient
- reflect on the data structures we've learned and see how any of them could be useful in improving the slow parts in your code

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

changePriority(item, priority) *– update the priority of an element*

contains(item) *– check if an element exists in the priority queue*

More Priority Queue Operations

More Operations

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

We'll use priority queues for lots of things later in the quarter.

Let's add them to our ADT now.

Some of these will be **asymptotically** faster for a heap than an AVL tree!

BuildHeap(elements $e_1, \ldots, e_n$)
Given n elements, create a heap containing exactly those n elements.

Even More Operations

BuildHeap(elements $e_1, \ldots, e_n$ **)** – Given n elements, create a heap containing exactly those n elements.

Try 1: Just call insert n times.

Worst case running time?

n calls, each worst case $\Theta(\log n)$. So it's $\Theta(n \log n)$ right?

That proof isn't valid. There's no guarantee that we're getting the worst case every time!

Proof is right if we just want an $O()$ bound

- But it's not clear if it's tight.

BuildHeap Running Time

Let's try again for a Theta bound.

The problem last time was making sure we always hit the worst case.

If we insert the elements in decreasing order **we will!**

- Every node will have to percolate all the way up to the root.

So we really have $n\ \Theta(\log n)$ operations. QED.

There's still a bug with this proof!

BuildHeap Running Time (again)

Let's try once more.

Saying the worst case was decreasing order was a good start.

What are the actual running times?

It's $\Theta(h)$, where h is the **current** height.

- The tree isn't height $\log n$ at the beginning.

But most nodes are inserted in the last two levels of the tree.

- For most nodes, h is $\Theta(\log n)$.

The number of operations is at least

$\frac{n}{2} \cdot \Omega(\log n) = \Omega(n \log n)$.

Can We Do Better?

What's causing the n **add** strategy to take so long?

- Most nodes are near the bottom, and might need to percolate all the way up.

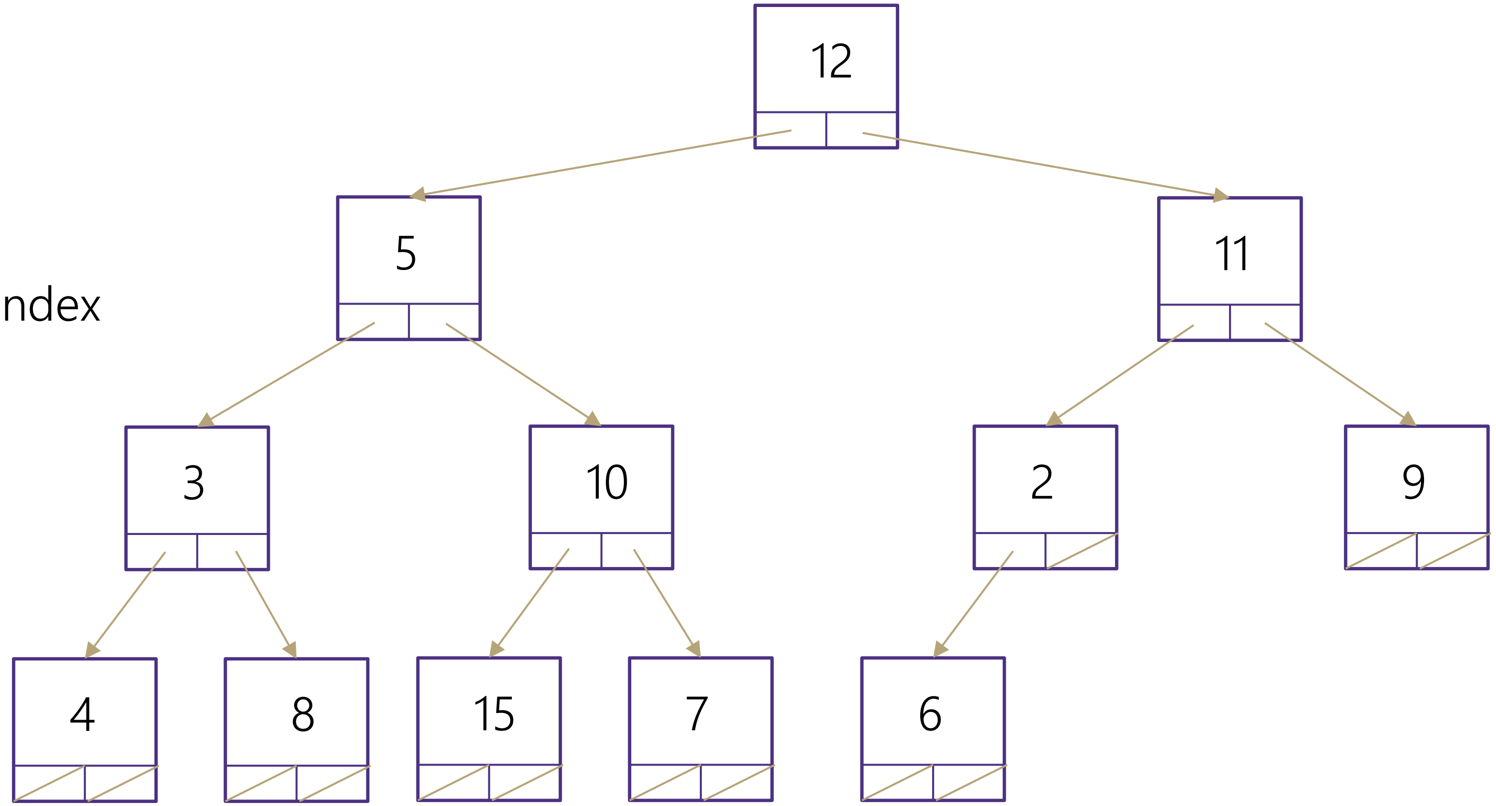
Idea 2: Dump everything in the array, and percolate things down until the heap invariant is satisfied

- Intuition: this could be faster!
- The bottom two levels of the tree have $\Omega(n)$ nodes, the top two have 3 nodes
- Maybe we can make "most of the nodes" go only a constant distance

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index

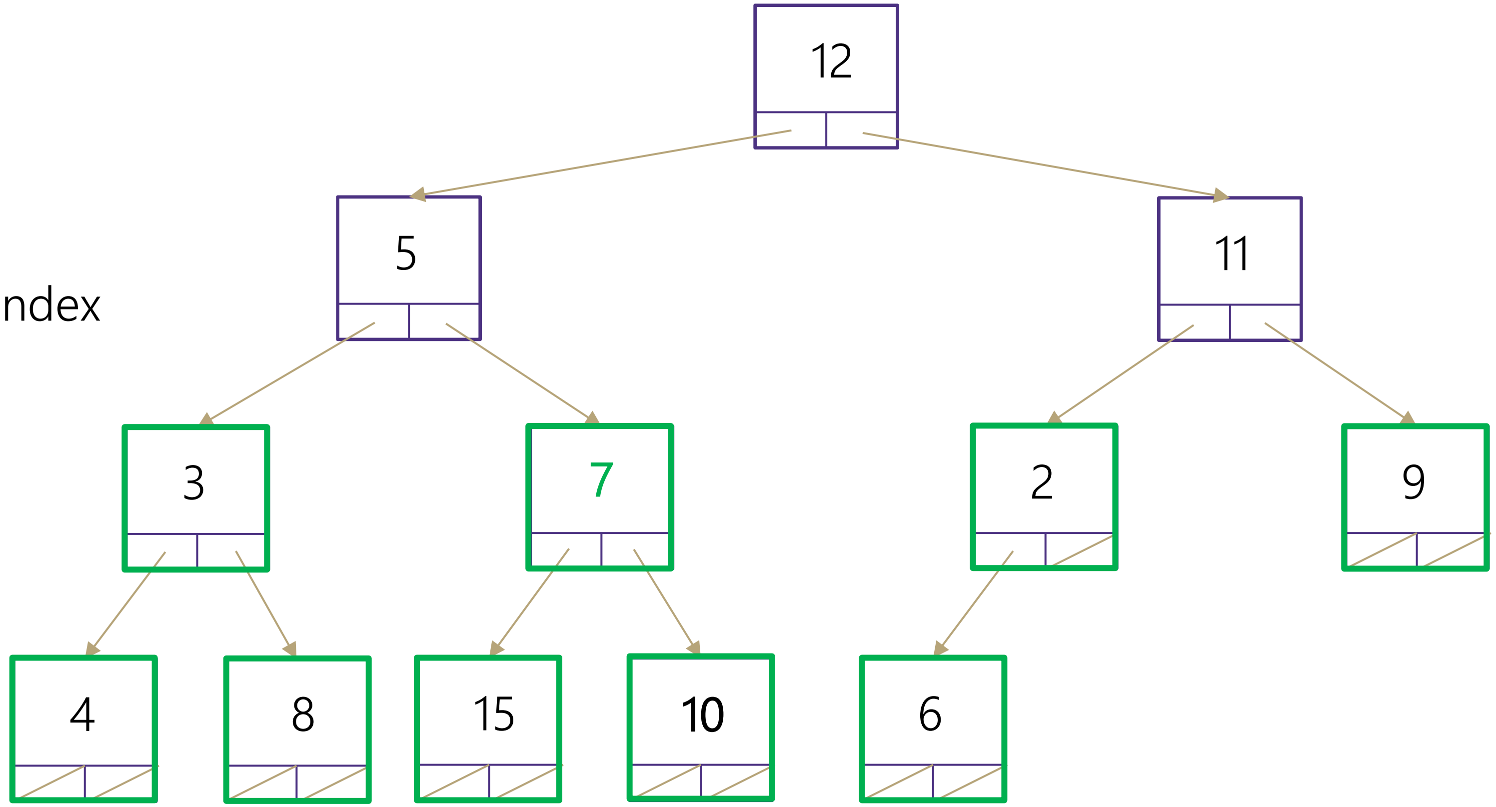


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	15	7	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3



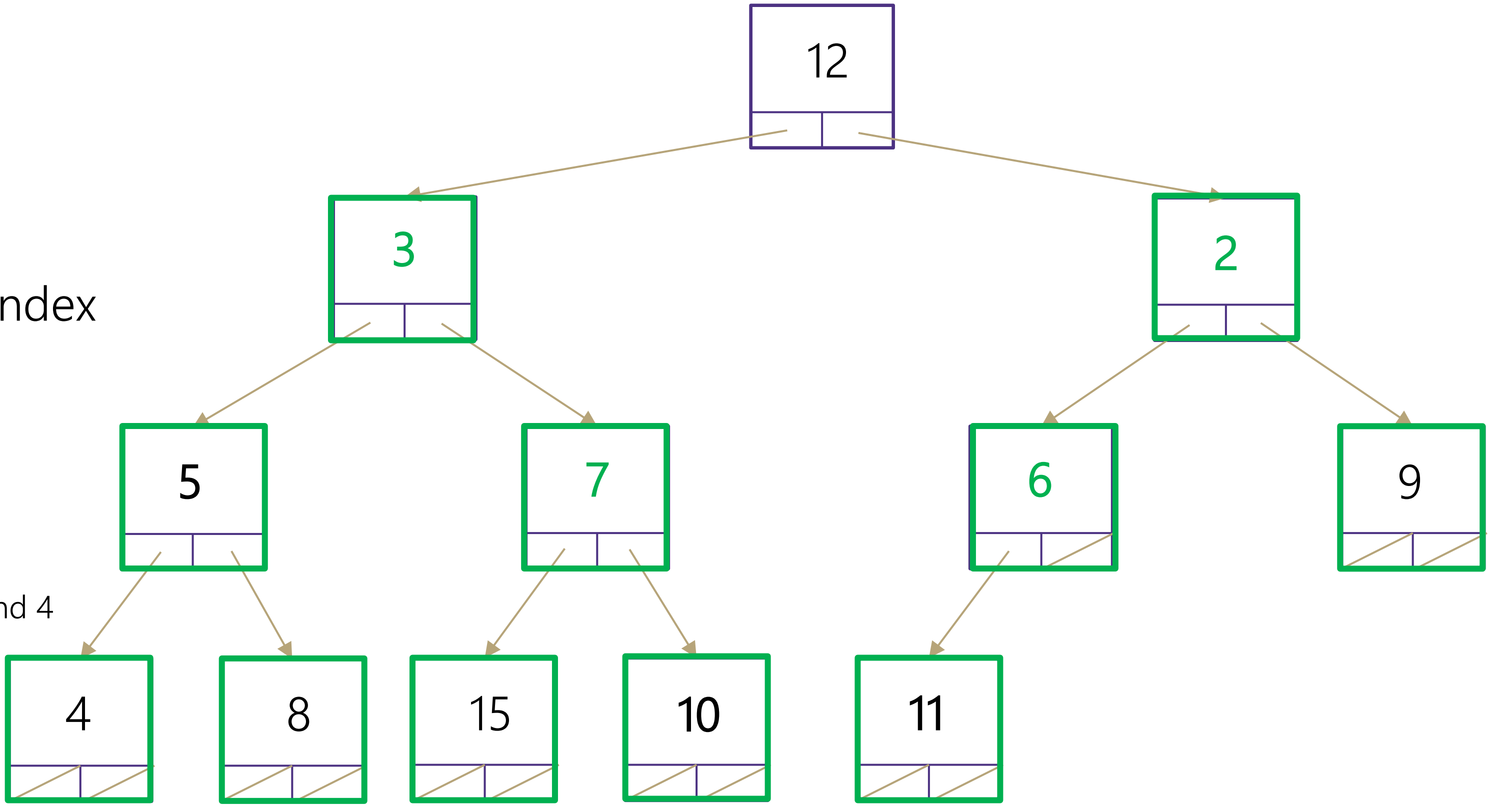
0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	15	7	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2

keep percolating down
like normal here and swap 5 and 4

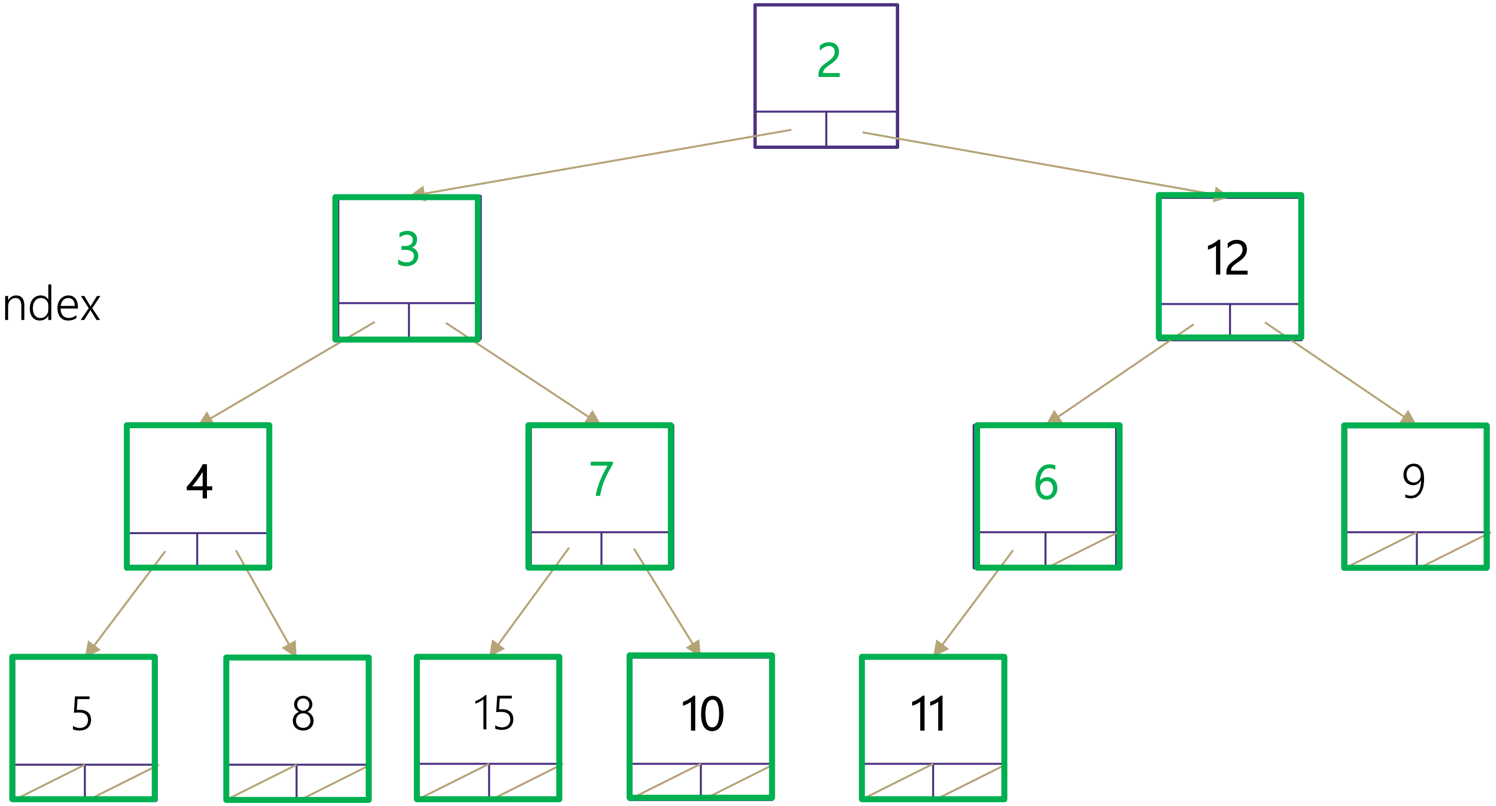


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	7	2	9	4	8	15	10	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
    1. percolateDown level 4
    2. percolateDown level 3
    3. percolateDown level 2
    4. percolateDown level 1

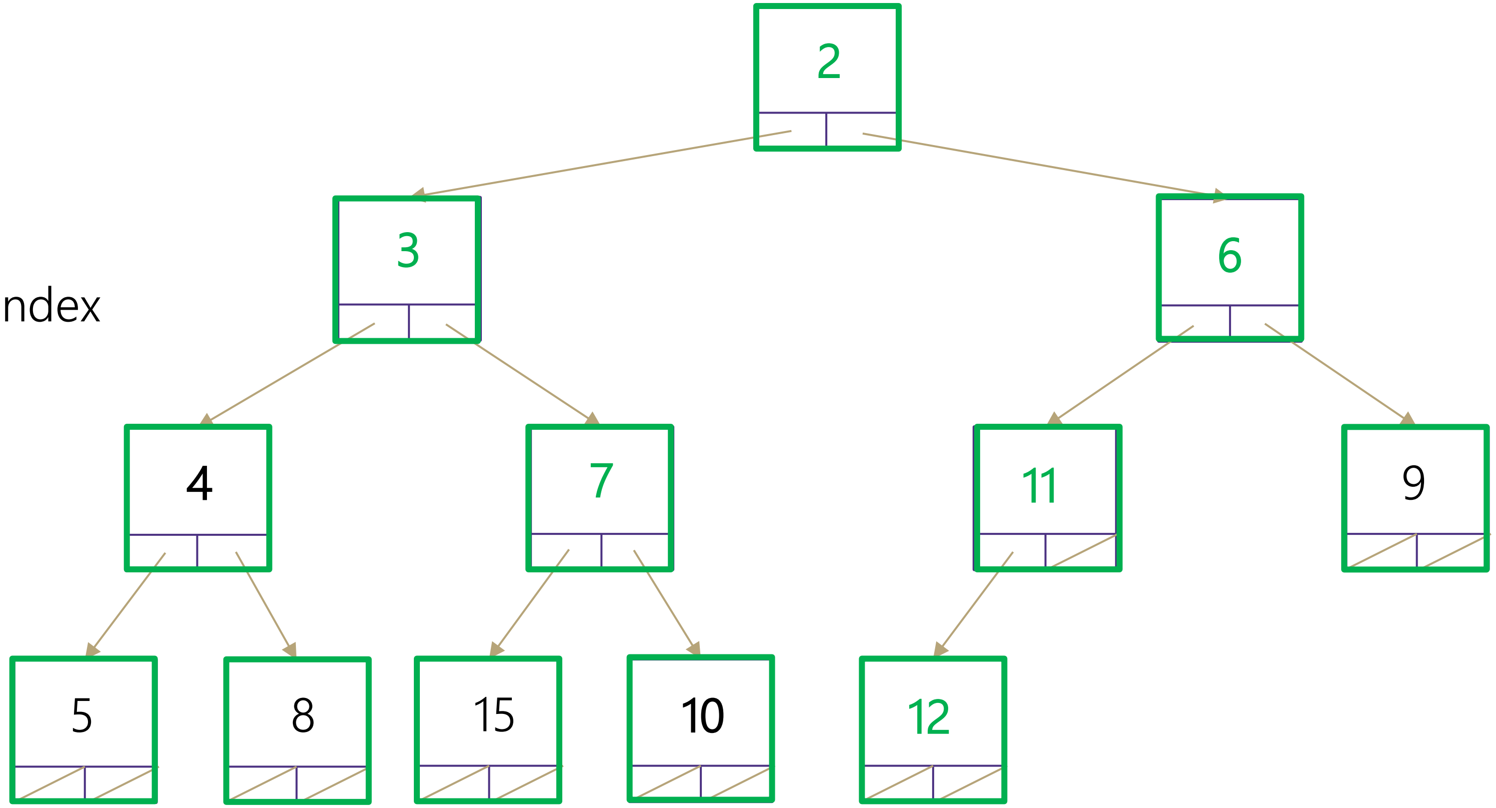


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	3	2	4	7	6	9	5	8	15	10	11		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 15, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
    1. percolateDown level 4
    2. percolateDown level 3
    3. percolateDown level 2
    4. percolateDown level 1



0	1	2	3	4	5	6	7	8	9	10	11	12	13
2	3	6	4	7	11	9	5	8	15	10	12		

Is It Really Faster? Floyd's buildHeap runs in O(n) time!

percolateDown() has worst case log n in general, but for most of these nodes, it has a much smaller worst case!

- n/2 nodes in the tree are leaves, have 0 levels to travel
- n/4 nodes have at most 1 level to travel
- n/8 nodes have at most 2 levels to travel
- etc…

$$\text{worst-case-work(n)} \approx \underbrace{\frac{n}{2}\cdot 1}_{\text{much of the work}} + \underbrace{\frac{n}{4}\cdot 2}_{\text{a little less}} + \underbrace{\frac{n}{8}\cdot 3}_{\text{a little less}} + \cdots + \underbrace{1\cdot(\log n)}_{\text{barely anything}}$$

Intuition: Even though there are log n levels, each level does a smaller and smaller amount of work. Even with infinite levels, as we sum smaller and smaller values (think $\frac{1}{2^i}$) we converge to a constant factor of n.

Optional Slide Floyd's buildHeap Summation

$n/2 \cdot 1 + n/4 \cdot 2 + n/8 \cdot 3 + \cdots + 1 \cdot (\log n)$

factor out n

work(n) $\approx n\left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{\log n}{n}\right)$ find a pattern -> powers of 2 work(n) $\approx n\left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{\log n}{2^{\log n}}\right)$ Summation!

$$work(n) \approx n \sum_{i=1}^{?} \frac{i}{2^i}$$

? = upper limit should give last term

We don't have a summation for this! Let's make it look more like a summation we do know.

Infinite geometric series

$$work(n) \leq n \sum_{i=1}^{\text{logn}} \frac{\left(\frac{3}{2}\right)^i}{2^i}$$

$$if -1 < x < 1 \; then \sum_{i=0}^{\infty} x^i = \frac{1}{1-x} = x$$

$$work(n) \approx n \sum_{i=1}^{\text{logn}} \frac{i}{2^i} \leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = n * 4$$

Floyd's buildHeap runs in O(n) time!

Even More Operations

These operations will be useful in a few weeks…

IncreaseKey(element,priority) Given an element of the heap and a new, larger priority, update that object's priority.

DecreaseKey(element,priority) Given an element of the heap and a new, smaller priority, update that object's priority.

Delete(element) Given an element of the heap, remove that element.

Should just be going to the right spot and percolating…

Going to the right spot is the tricky part.

In the programming projects, you'll use a dictionary to find an element quickly.