Lecture 13: Intro to Heaps

CSE 373 Data Structures and Algorithms

Warm Up: Class Gradebook

You have been asked to create a new system for organizing students in a course and their accompanying grades

What functionality is needed?

What operations need to be supported?

- Add students to course
- ➡ Add grade to student's record
- Update grade already in student's record
- Remove student from course
- Check if student is in course
- ➡ Find specific grade for student

Which operations should be prioritized?

What type of data will you have?

What are the relationships within the data?

Organize students by name, keep grades in time order…

How much data will you have?

A couple hundred students, < 20 grades per student

Will your data set grow? A lot at the beginning,

Will your data set shrink? Not much after that

How do you think things will play out?

How likely are best cases?

How likely are worst cases?

Lots of add and drops?

Lots of grade updates?

Students with similar identifiers?

Example: Class Gradebook

What data should we use to identify students? (keys)

- Student IDs – unique to each student, no confusion (or collisions)
- Names – easy to use, support easy to produce sorted by name

How should we store each student's grades? (values)

- Array List – easy to access, keeps order of assignments
- Hash Table – super efficient access, no order maintained

Which data structure is the best fit to store students and their grades?

- Hash Table – student IDs as keys will make access very efficient
- AVL Tree - student names as keys will maintain alphabetical order

Announcements

P2 due Wednesday

Midterm out this Friday – due 1 week later

- Group assignment
- Open note/ open internet, closed course staff
- 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

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

Review: Dictionaries

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

Why are we so obsessed with Dictionaries?

When dealing with data:

- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

Operation		ArrayList	LinkedList	HashTable	BST	AVLTree
put(key,value)	best	Θ(1)	Θ(1)	Θ(1)	Θ(1)	Θ(1)
	worst	Θ(n)	Θ(n)	Θ(n)	Θ(n)	Θ(logn)
get(key)	best	Θ(1)	Θ(1)	Θ(1)	Θ(1)	Θ(1)
	worst	Θ(n)	Θ(n)	Θ(n)	Θ(n)	Θ(logn)
remove(key)	best	Θ(1)	Θ(1)	Θ(1)	Θ(1)	Θ(logn)
	worst	Θ(n)	Θ(n)	Θ(n)	Θ(n)	Θ(logn)

Design Decisions

Before coding can begin engineers must carefully consider the design of their code will organize and manage data

Things to consider:

What functionality is needed?
- What operations need to be supported?
- Which operations should be prioritized?

What type of data will you have?
- What are the relationships within the data?
- How much data will you have?
- Will your data set grow?
- Will your data set shrink?

How do you think things will play out?
- How likely are best cases?
- How likely are worst cases?

Practice: Music Storage

You have been asked to create a new system for organizing songs in a music service. For each song you need to store the artist and how many plays that song has.

What functionality is needed?

- What operations need to be supported?
- Which operations should be prioritized?

Update number of plays for a song
Add a new song to an artist's collection
Add a new artist and their songs to the service
Find an artist's most popular song
Find service's most popular artist
more…

What type of data will you have?

- What are the relationships within the data?
- How much data will you have?
- Will your data set grow?
- Will your data set shrink?

Artists need to be associated with their songs,
songs need t be associated with their play counts
Play counts will get updated a lot
New songs will get added regularly

How do you think things will play out?

- How likely are best cases?
- How likely are worst cases?

Some artists and songs will need to be accessed a lot more than others
Artist and song names can be very similar

Practice: Music Storage

How should we store songs and their play counts?

Hash Table – song titles as keys, play count as values, quick access for updates

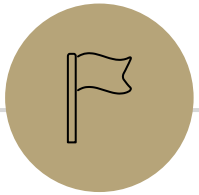
Array List – song titles as keys, play counts as values, maintain order of addition to system

How should we store artists with their associated songs?

Hash Table – artist as key,

- Hash Table of their (songs, play counts) as values
- AVL Tree of their songs as values

AVL Tree – artists as key, hash tables of songs and counts as values



Priority Queues

```
PriorityQueue<FoodOrder> pq = new PriorityQueue<>();
```

some motivation for today's lecture:

- PQs are a staple of Java's built-in data structures, commonly used for sorting needs
- Using PQs and knowing their implementations are common technical interview subjects
- You're implementing one in the next project – so everything you get out of today should be useful for that!

Priority Queue / heaps roadmap

- PriorityQueue ADT
- PriorityQueue implementations with current toolkit
- Binary Heap idea + invariants
- Binary Heap methods
- Binary Heap implementation details

A new ADT!

Imagine you're managing a queue of food orders at a restaurant, which normally takes food orders first-come-first-served.

Suddenly, Ana Mari Cauce walks into the restaurant!

You realize that you should serve her as soon as possible (to gain political influence or so that she leaves the restaurant as soon as possible), and realize other celebrities (CSE 373 staff) could also arrive soon. Your new food management system should rank customers and let us know which food order we should work on next (the most prioritized 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() – returns the element with the smallest priority, removes it from the collection

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

Imagine you're managing a queue of food orders at a restaurant, which normally takes food orders first-come-first-served. But suddenly, Ana Marie Cauce walks into the restaurant. You know that you should server her as soon as possible (to either suck up or kick her out of the restaurant), and realize other celebrities (CSE 373 staff) could also arrive soon. Your new food management system should rank customers and let us know which food order we should work on next (the most prioritized thing).

Other uses:

- Well-designed printers
- Huffman Coding (see in CSE 143 last hw)
- Sorting algorithms
- Graph algorithms

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

Min Priority Queue ADT

state

Set of comparable values - Ordered based on "priority"

behavior

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

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

Max Priority Queue ADT

state

Set of comparable values - Ordered based on "priority"

behavior

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

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

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

Implementing Priority Queues: Take I

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			
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 I

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	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
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 I

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	$\Theta(1)$	$\Theta(n)$	~~$\Theta(n)$~~ $\Theta(1)$
Linked List (sorted)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	~~$\Theta(\log n)$~~ $\Theta(1)$

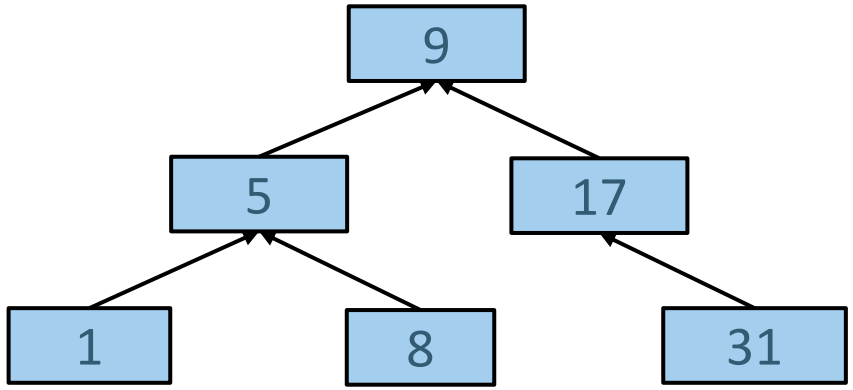
Add a field to keep track of the min.
Update on every insert or remove.

AVL Trees are our baseline – let's look at what computer scientists came up with as an alternative, analyze that, and then come back to AVL Tree as an option later

Review: Binary *Search* Trees

A Binary Search Tree is a binary tree with the following invariant: for every node with value k in the BST:

- The left subtree only contains values <k
- The right subtree only contains values >k




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

Reminder: the BST ordering applies recursively to the entire subtree

Heaps

Idea:

In a BST, we organized the data to find anything quickly. (go left or right to find a value deeper in the tree)

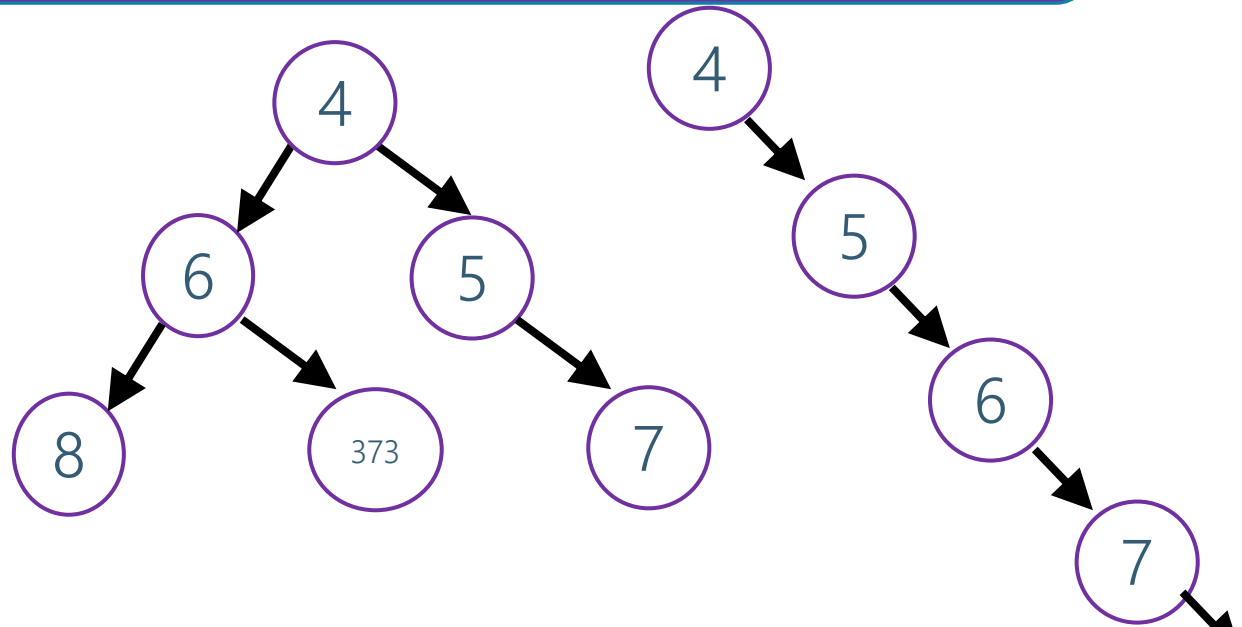
Now we just want to find the smallest things fast, so let's write a different invariant:

Heap invariant
Every node is less than or equal to both of its children.

In particular, the smallest node is at the root!

- Super easy to `peek` now!

Do we need more invariants?



Heaps

With the current definition we could still have degenerate trees. From our BST / AVL intuition, we know that degenerate trees take a long time to traverse from root → leaf, so we want to avoid these tree structures.

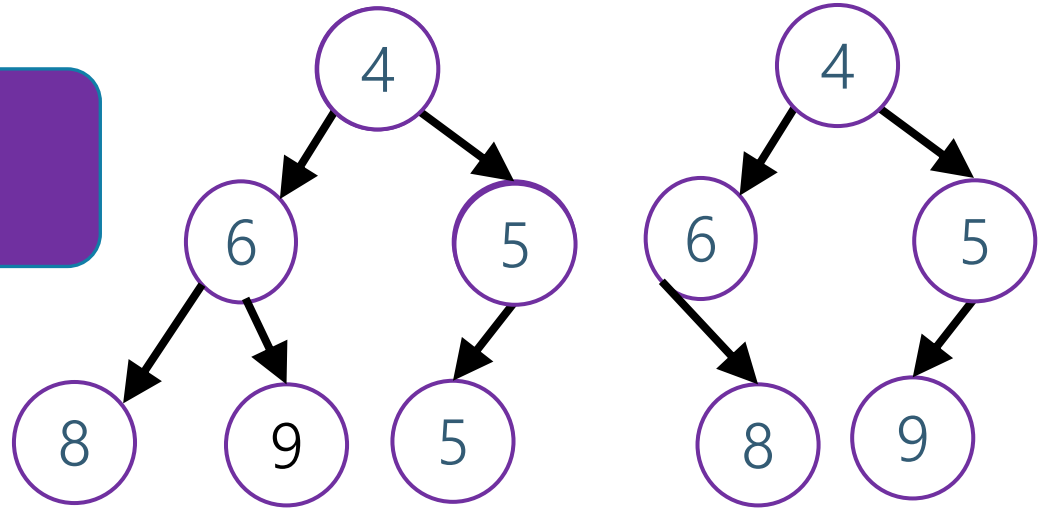
The BST invariant was a bit complicated to maintain.

- Because we had to make sure when we inserted we could maintain the exact BST structure where nodes to the left are less than, nodes to the right are greater than…
- The heap invariant is looser than the BST invariant.
- Which means we can make our structure invariant stricter.

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")



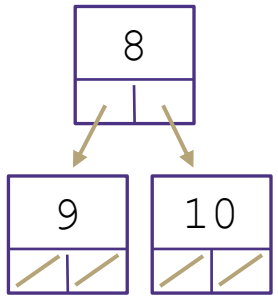
Binary Heap invariants summary

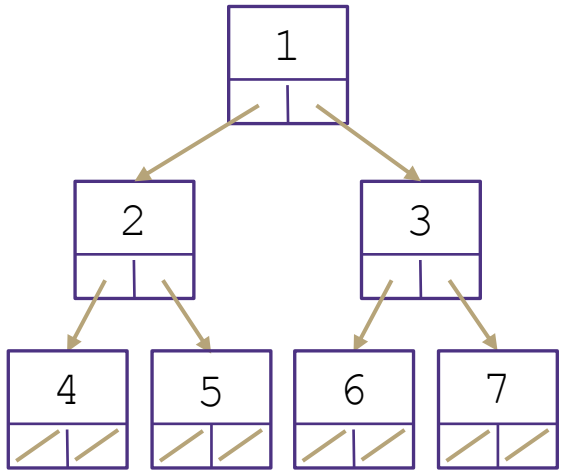
This is a big idea!
(heap invariants!)

One flavor of heap is a **binary** heap.

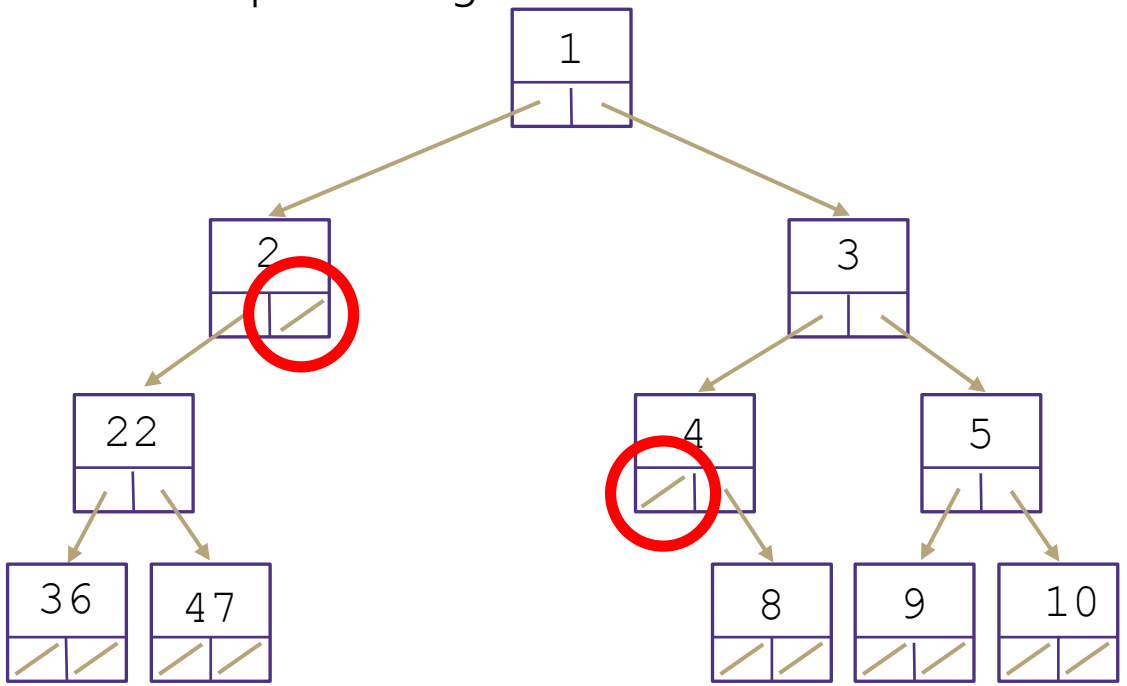
1. Binary Tree: every node has at most 2 children

2. Heap invariant: every node is smaller than (or equal to) its children





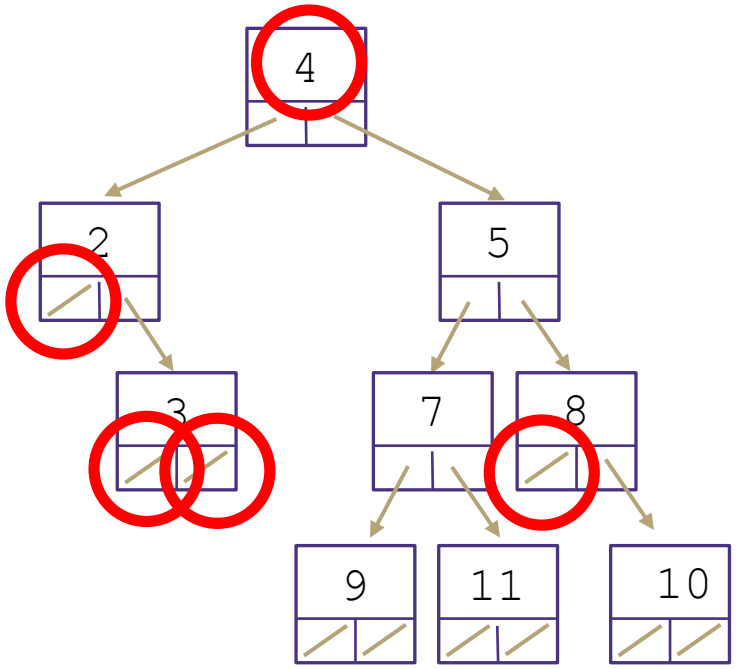
3. Heap structure invariant: Each level is "complete" meaning it has no "gaps"
- Heaps are filled up left to right



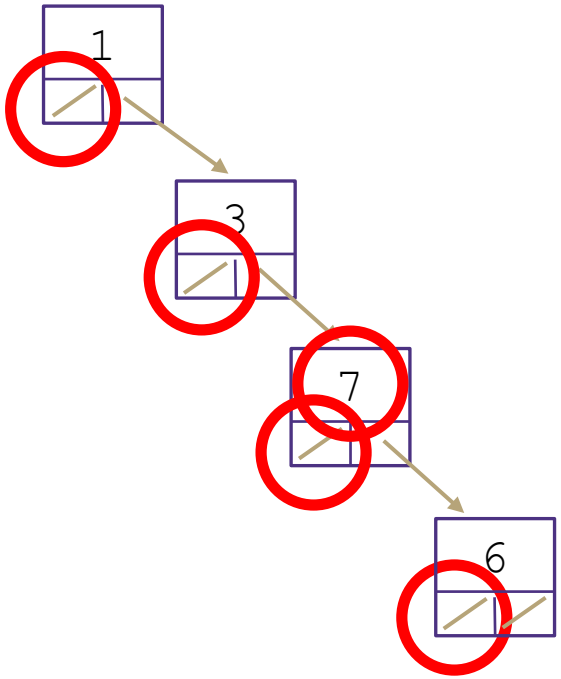
Self Check - Are these valid heaps?

Binary Heap Invariants:
1. Binary Tree
2. Heap
3. Complete

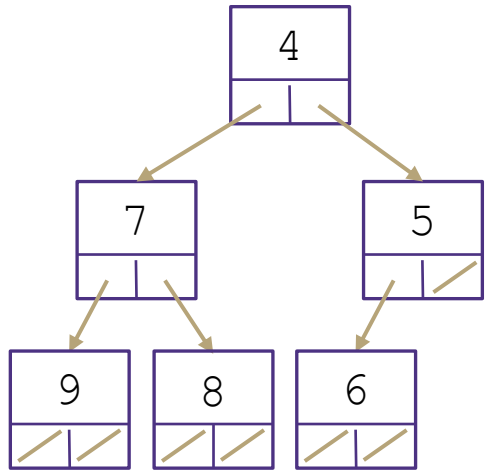
INVALID



INVALID



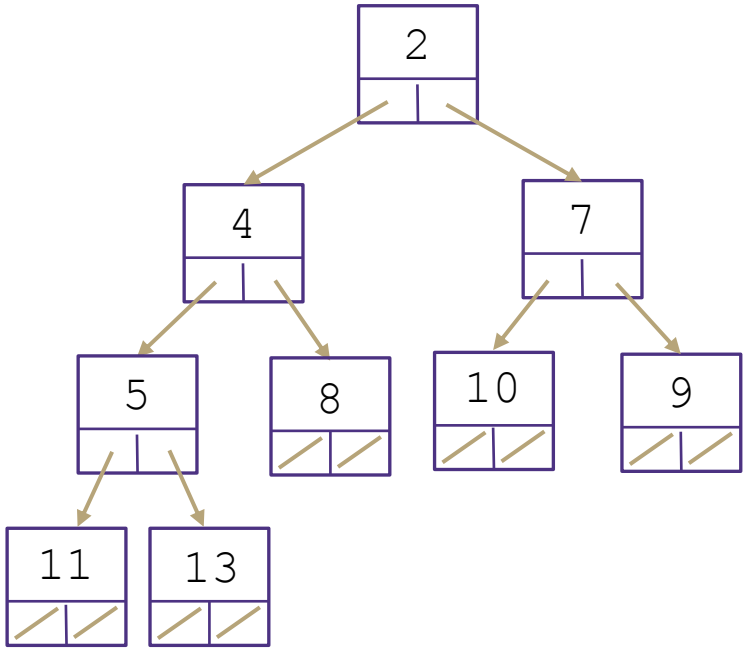
VALID

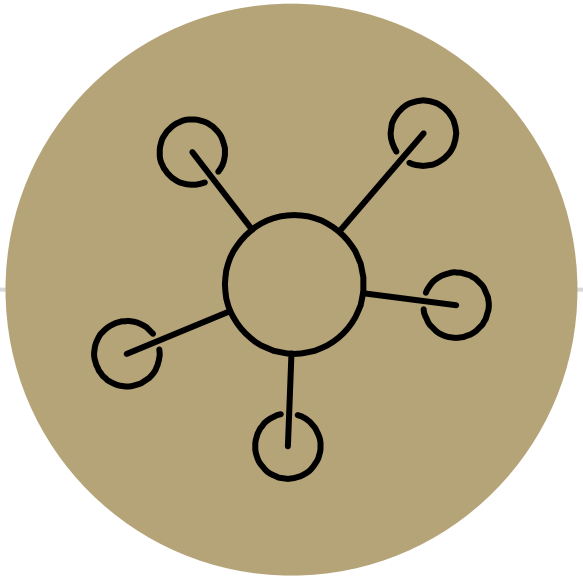


Heap heights

A binary heap bounds our height at Theta(log(n)) because it's complete – and it's actually a little stricter and better than AVL.

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





Questions?

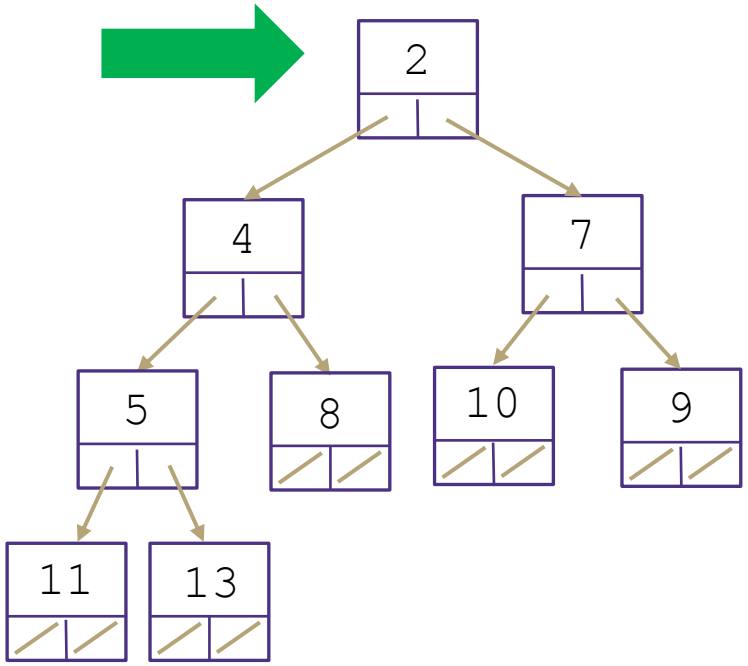
Priority Queue ADT
Priority Queue possible implementations
Heap invariants
Heap height

Priority Queue / heaps roadmap

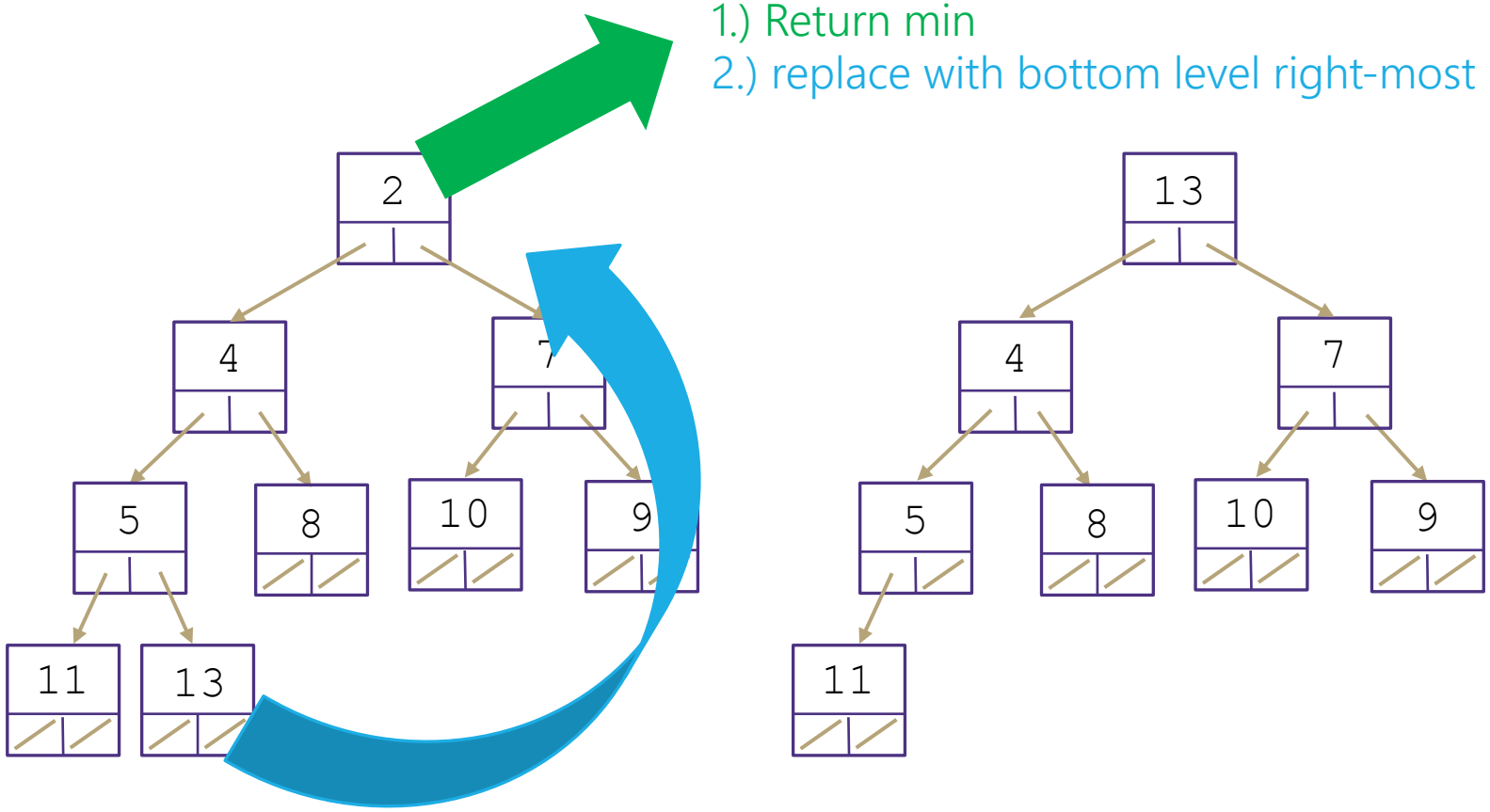
- PriorityQueue ADT
- PriorityQueue implementations with current toolkit
- Binary Heap idea + invariants
- **Binary Heap methods**
- Binary Heap implementation details

Implementing peekMin()

Runtime: $\Theta(1)$



Implementing removeMin()



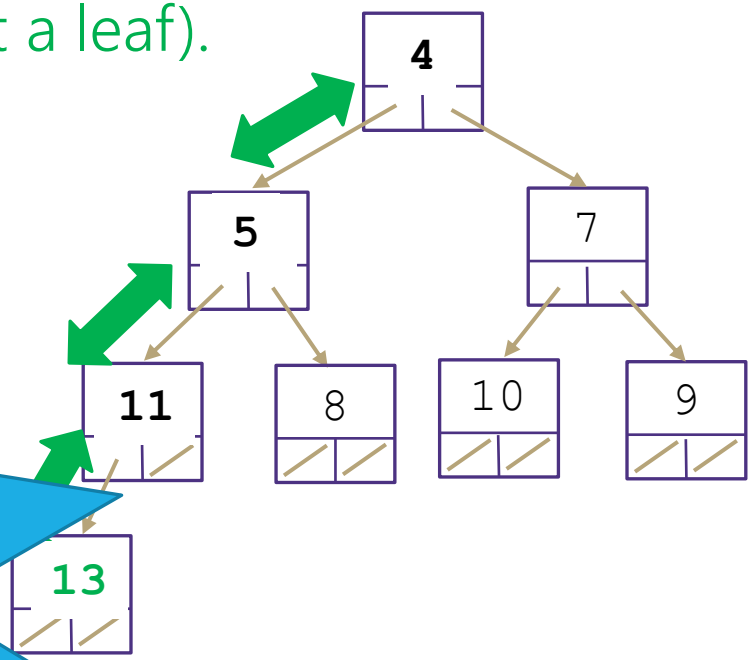
1.) Return min
2.) replace with bottom level right-most node

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).



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?)

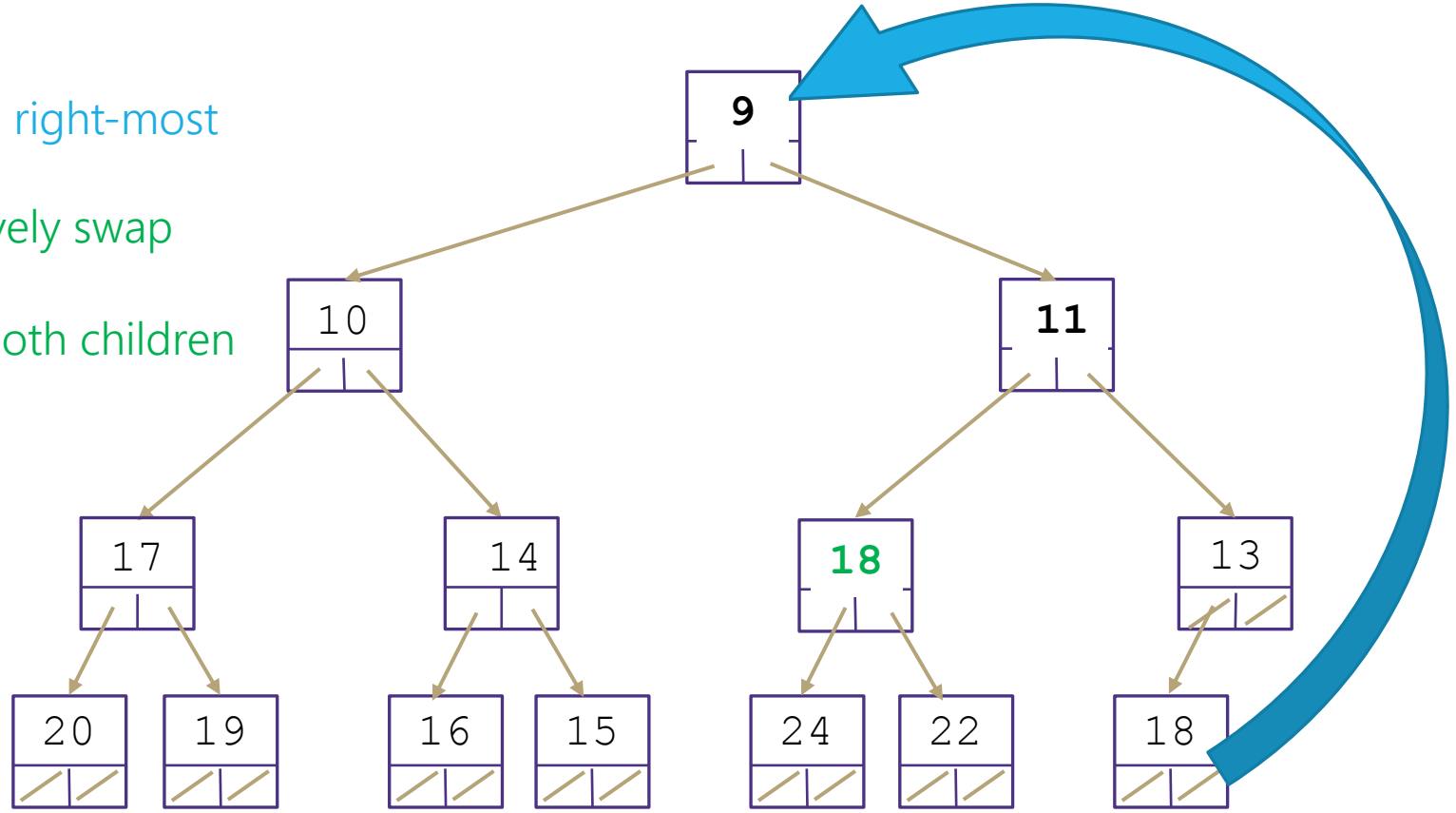
this is why we want to keep the height of the tree small! The height of these tree structures (BST, AVL, heaps) directly correlates with the worst case runtimes

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!)

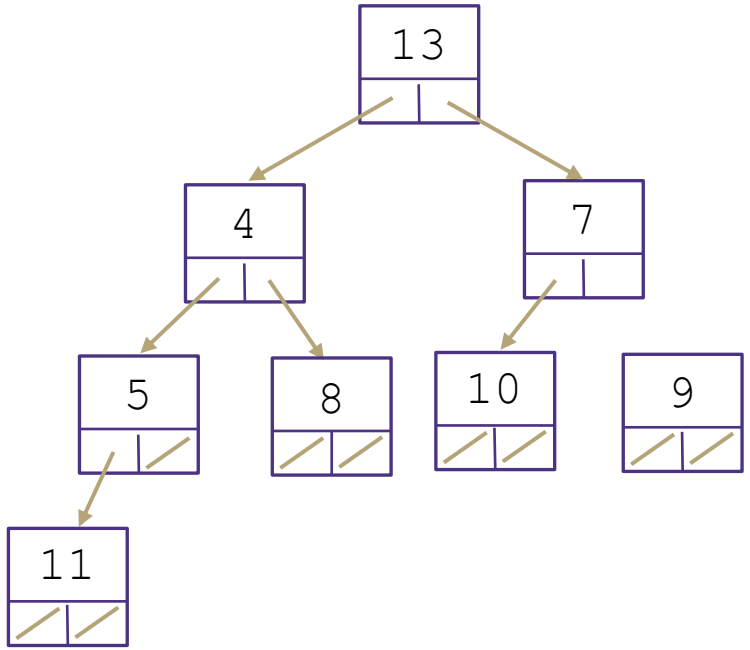
Structure invariant restored, heap invariant restored

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:

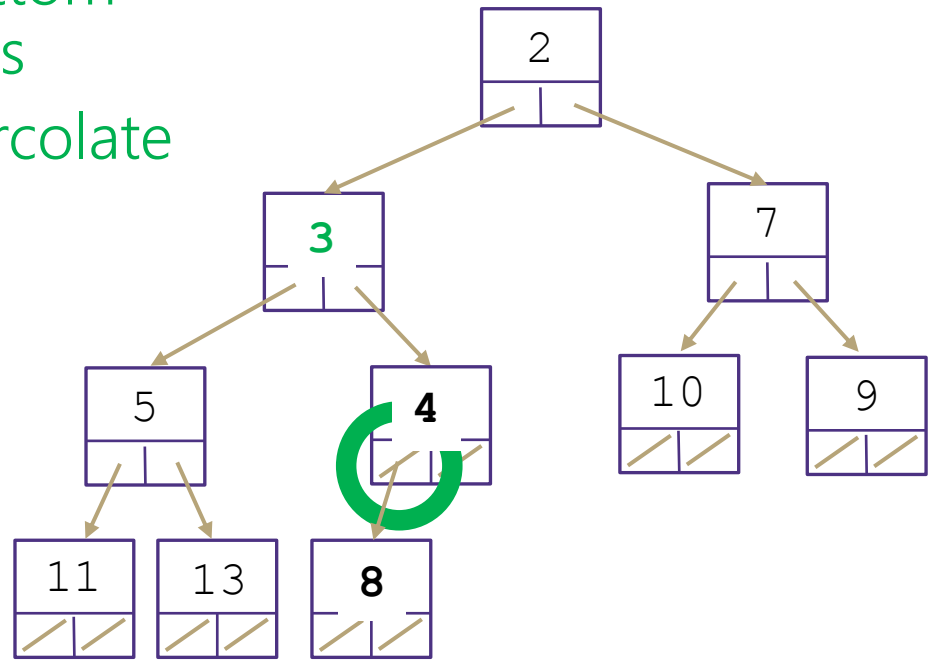
- Insert a node on the bottom level that ensure no gaps
- Fix heap invariant by percolate **UP**

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

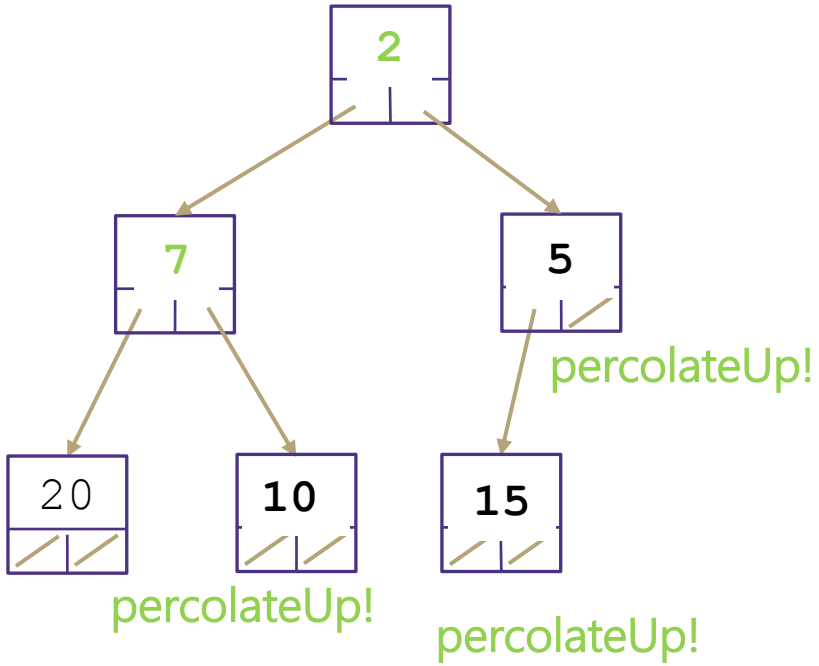
Add() Algorithm:

- 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

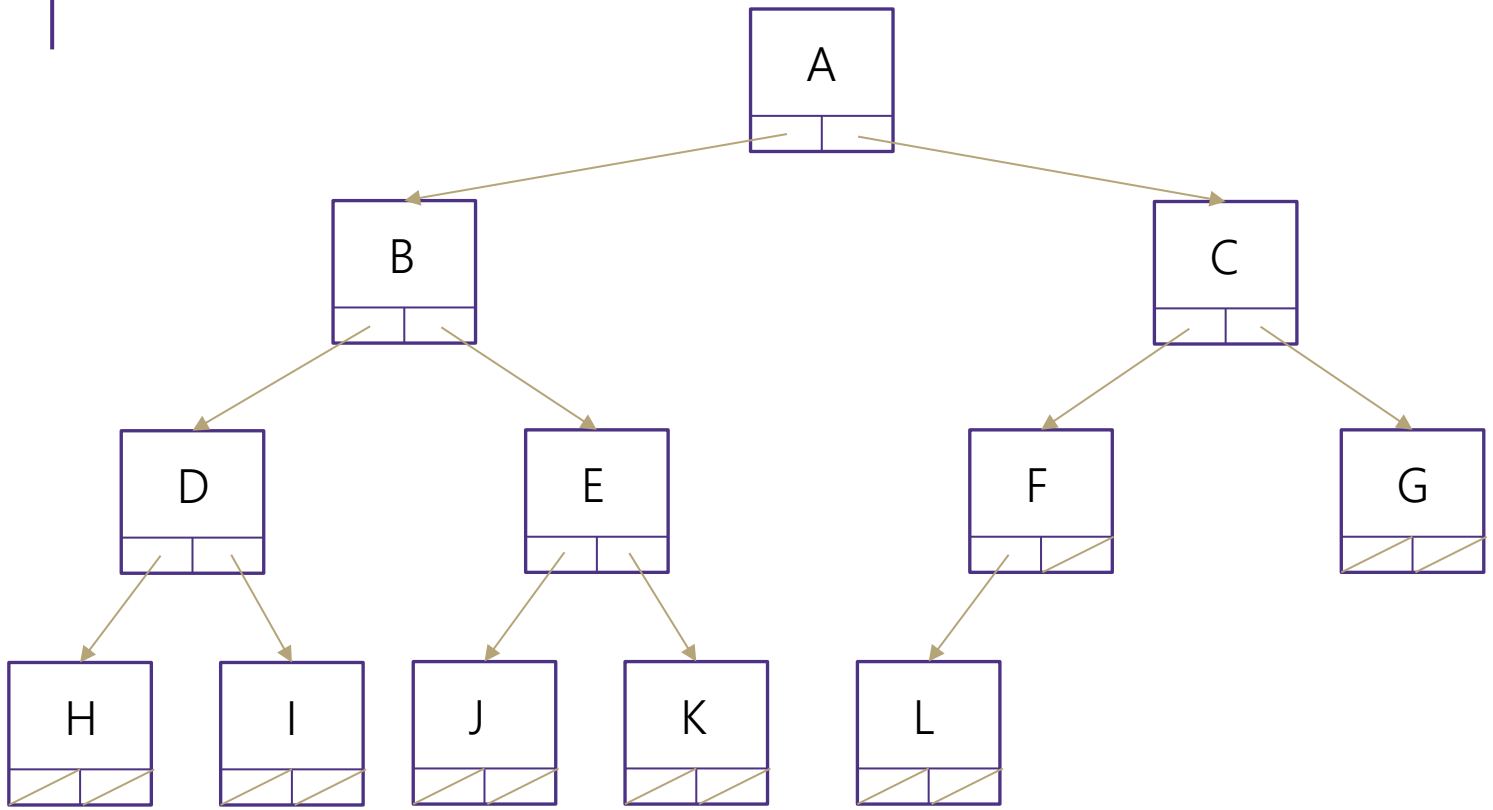
add():

- 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



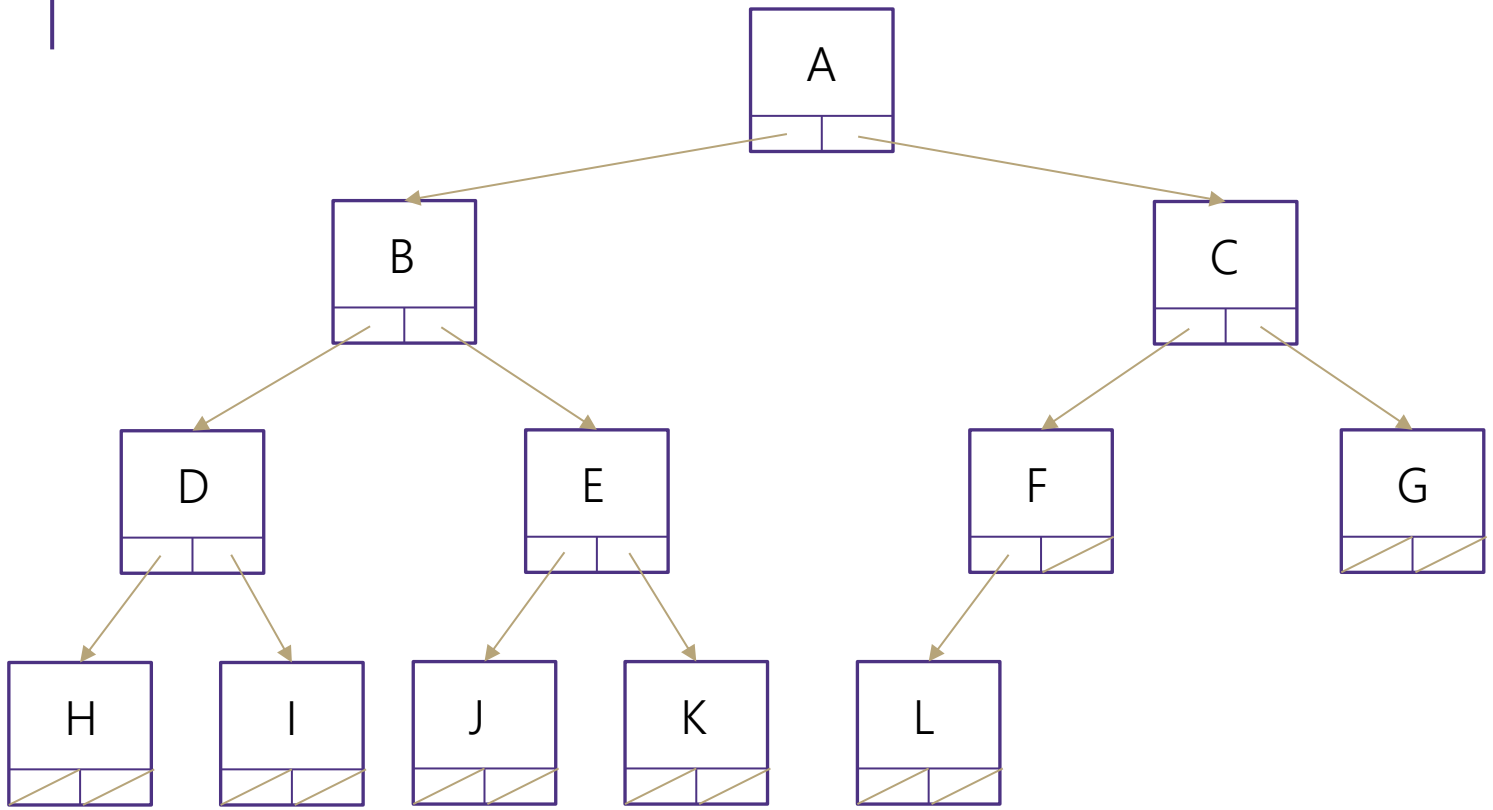
Fill array in level-order from left to right

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 still is 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?

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

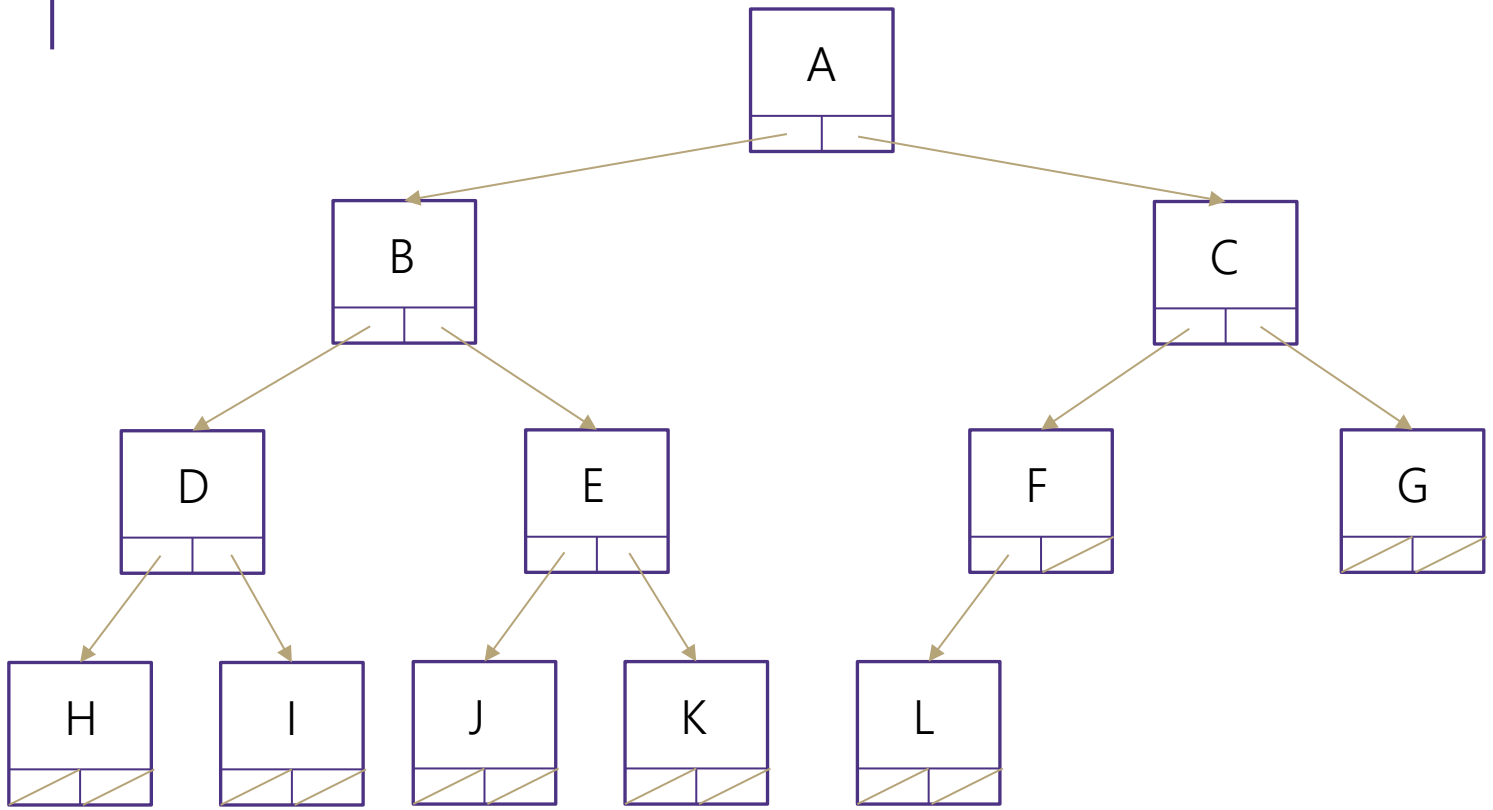
How do we find a node's right child?

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

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?

$$leftChild(i) = 2i$$

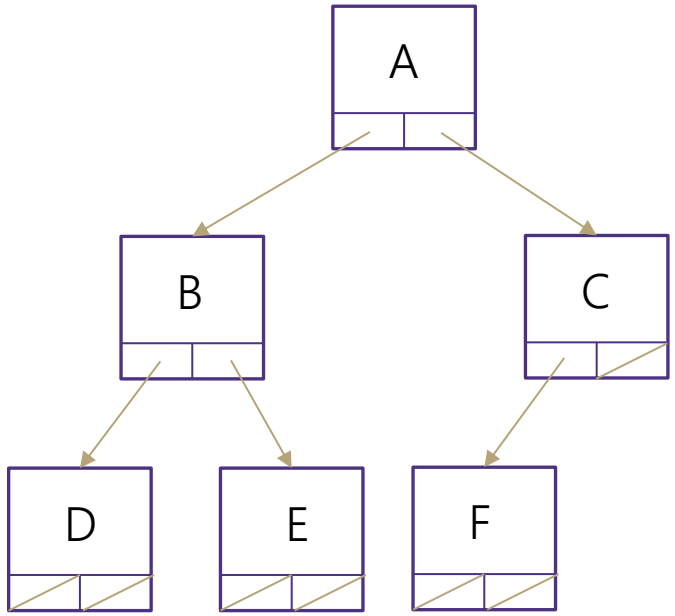
How do we find a node's right child?

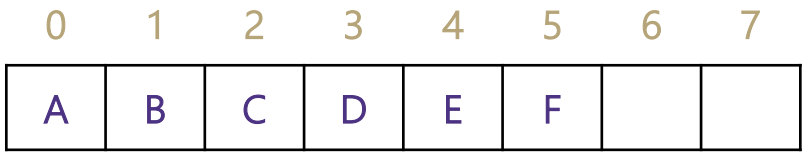
$$rightChild(i) = 2i$$

How do we find a node's parent?

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

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.

Are heaps always better? AVL vs Heaps

- 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.

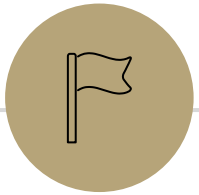
Heaps are stuck at Theta(n) runtime and we can't do anything more clever…. aha, just kidding.. unless…?

Relevant hint for project 3:

- When coming up with data structures, we can actually combine them with existing tools to improve our algorithms and runtimes. We can improve the worst-case runtime of get/contains to be a lot better than Theta(n) time depending on how we have our heap utilize an extra data-structure.

- For project 3, you should use an additional data structure to improve the runtime for changePriority(). It does not affect the correctness of your PQ at all (i.e. you can implement it correctly without the additional data structure). Please use a built-in Java collection instead of implementing your own (although you could in-theory).

- For project 3, feel free to try the following development strategy for the changePriority method
  - implement changePriority without regards to efficiency (without the extra data structure) at first
  - then, 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



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

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

Where Were We?

We were trying to design an algorithm for:

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

Just inserting leads to a $\Theta(n \log n)$ algorithm in the worst case.

Can we do better?

Can We Do Better?

What's causing the n `insert` strategy to take so long?

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

What if instead we dumped everything in the array and then

tried to percolate things down to fix the invariant?

Seems like it might be faster

- The bottom two levels of the tree have $\Omega(n)$ nodes, the top two have 3 nodes.
- Maybe we can make "most nodes" go a constant distance.

Is It Really Faster?

Assume the tree is **perfect**
- the proof for complete trees just gives a different constant factor.

percolateDown() doesn't take $\log n$ steps each time!

Half the nodes of the tree are leaves
- Leaves run percolate down in constant time

1/4 of the nodes have at most 1 level to travel

1/8 the nodes have at most 2 levels to travel

etc…

work(n) $\approx \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 1 \cdot (\log n)$

Closed form Floyd's buildHeap

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

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

$$work(n) \leq n \sum_{i=1}^{\log n} \frac{\left(\frac{3}{2}\right)^i}{2^i} \qquad if\ -1 < x < 1\ then \sum_{i=0}^{\infty} x^i = \frac{1}{1-x} = x \qquad work(n) \approx n \sum_{i=1}^{\log n} \frac{i}{2^i} \leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = n * 4$$

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

Floyd's BuildHeap

Ok, it's really faster.
But can we make it **work**?

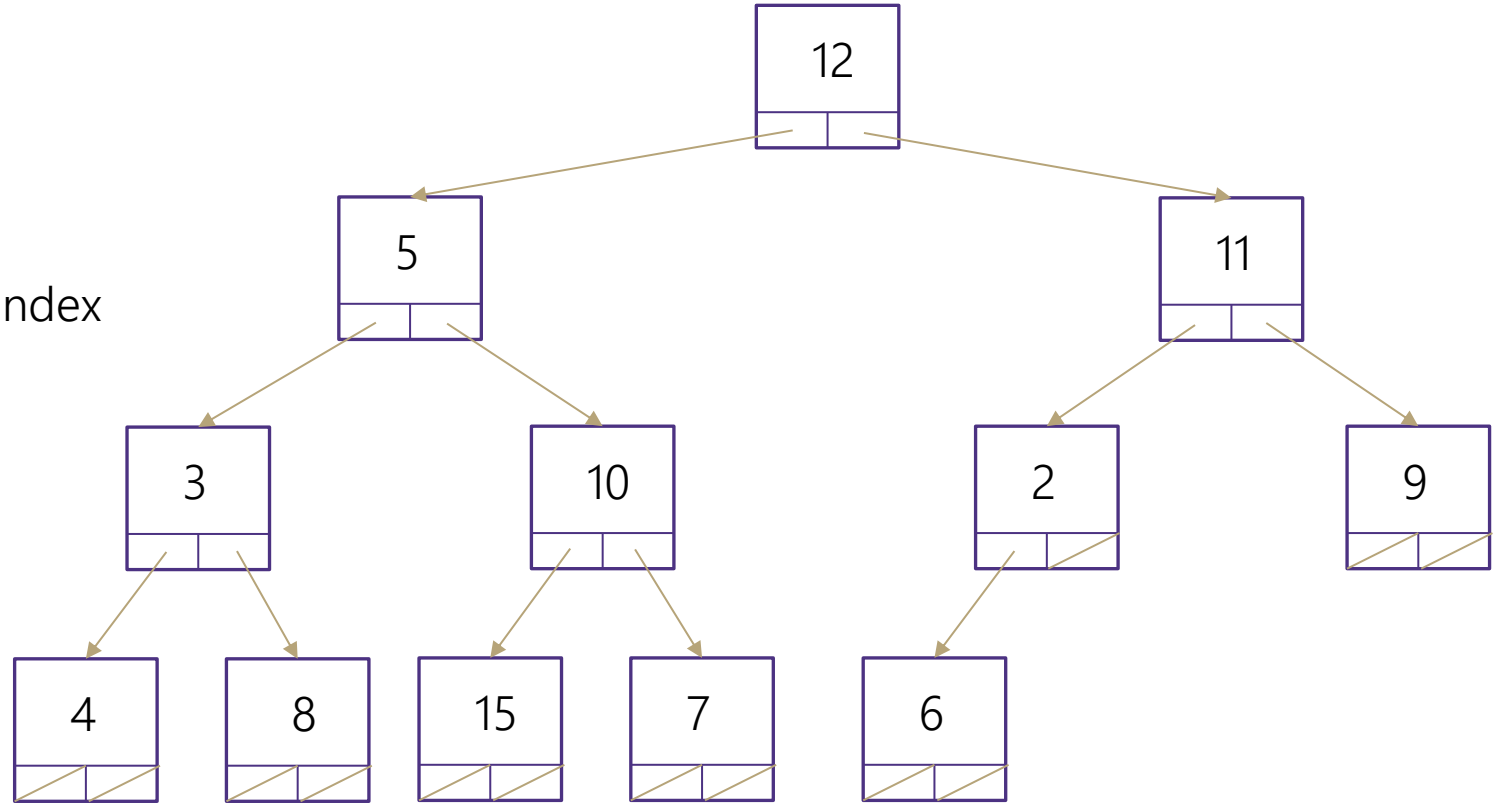
It's not clear what order to call the `percolateDown`'s in.

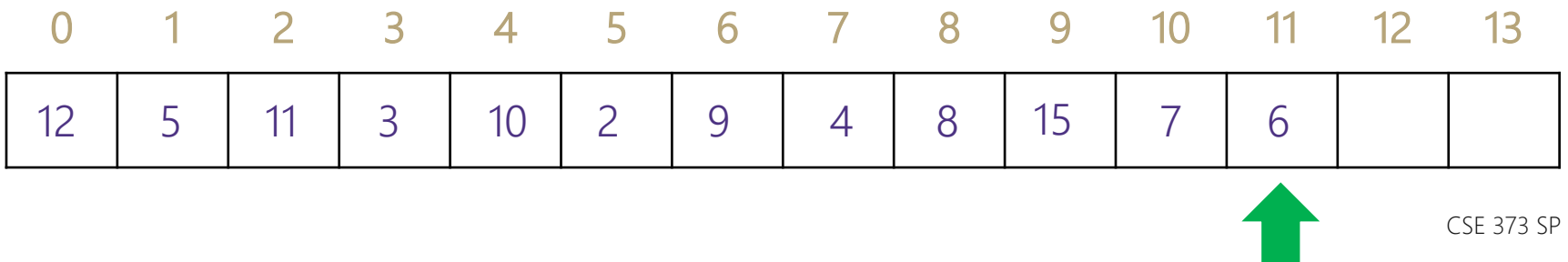
Should we start at the top or bottom? Will one `percolateDown` on each element be enough?

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

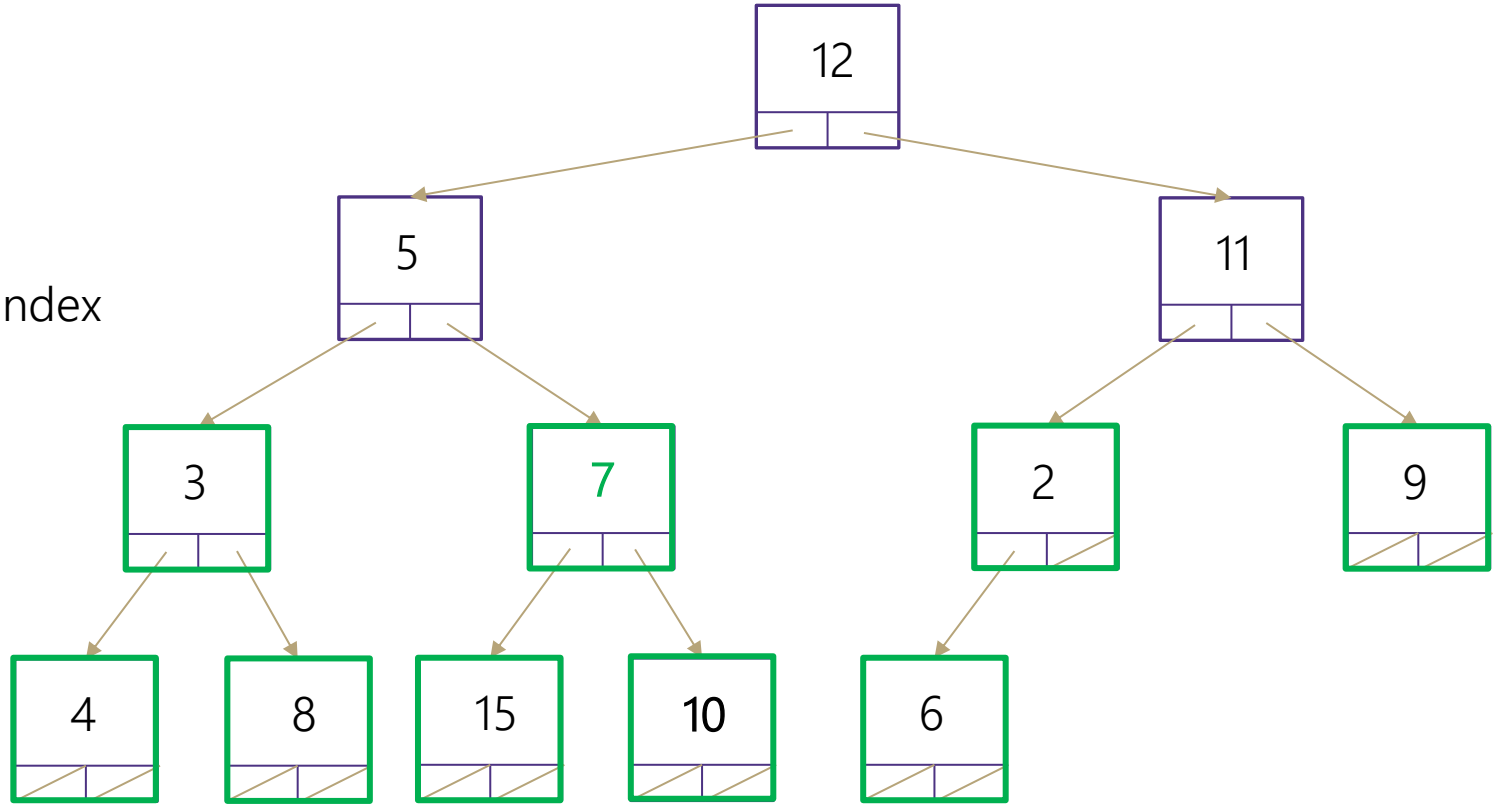


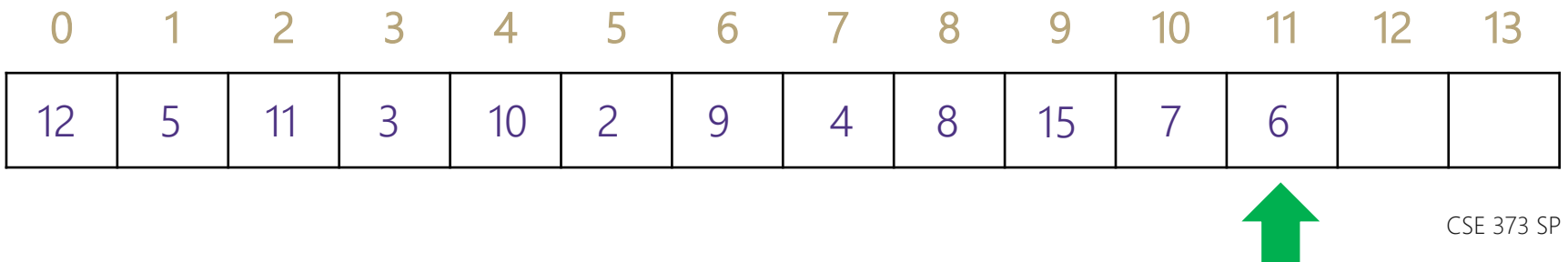


Floyd's buildHeap algorithm

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



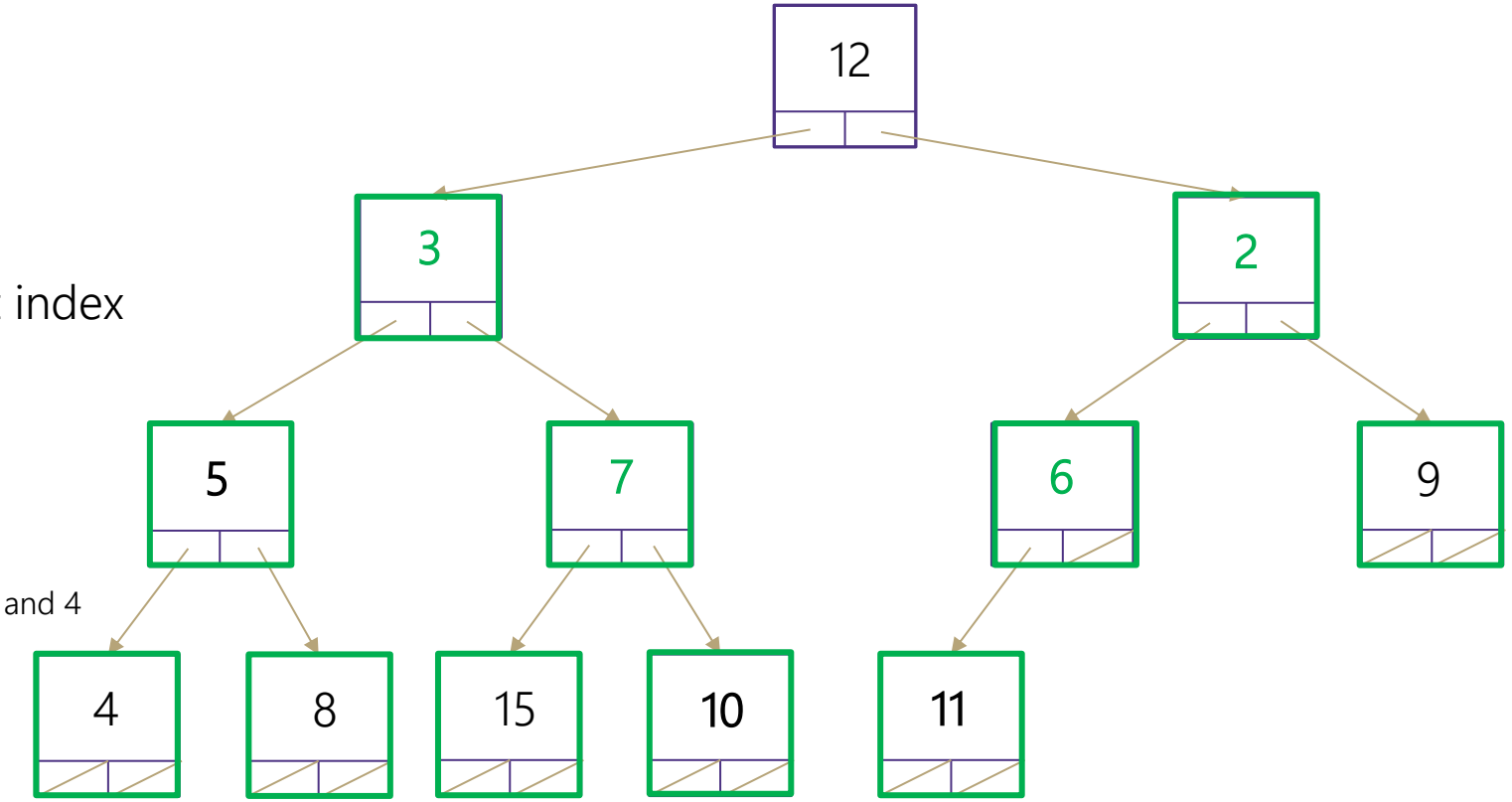


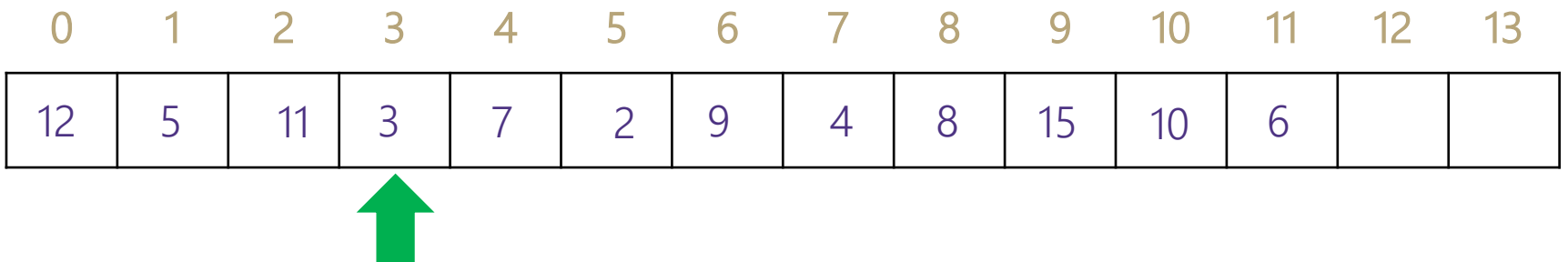
Floyd's buildHeap algorithm

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

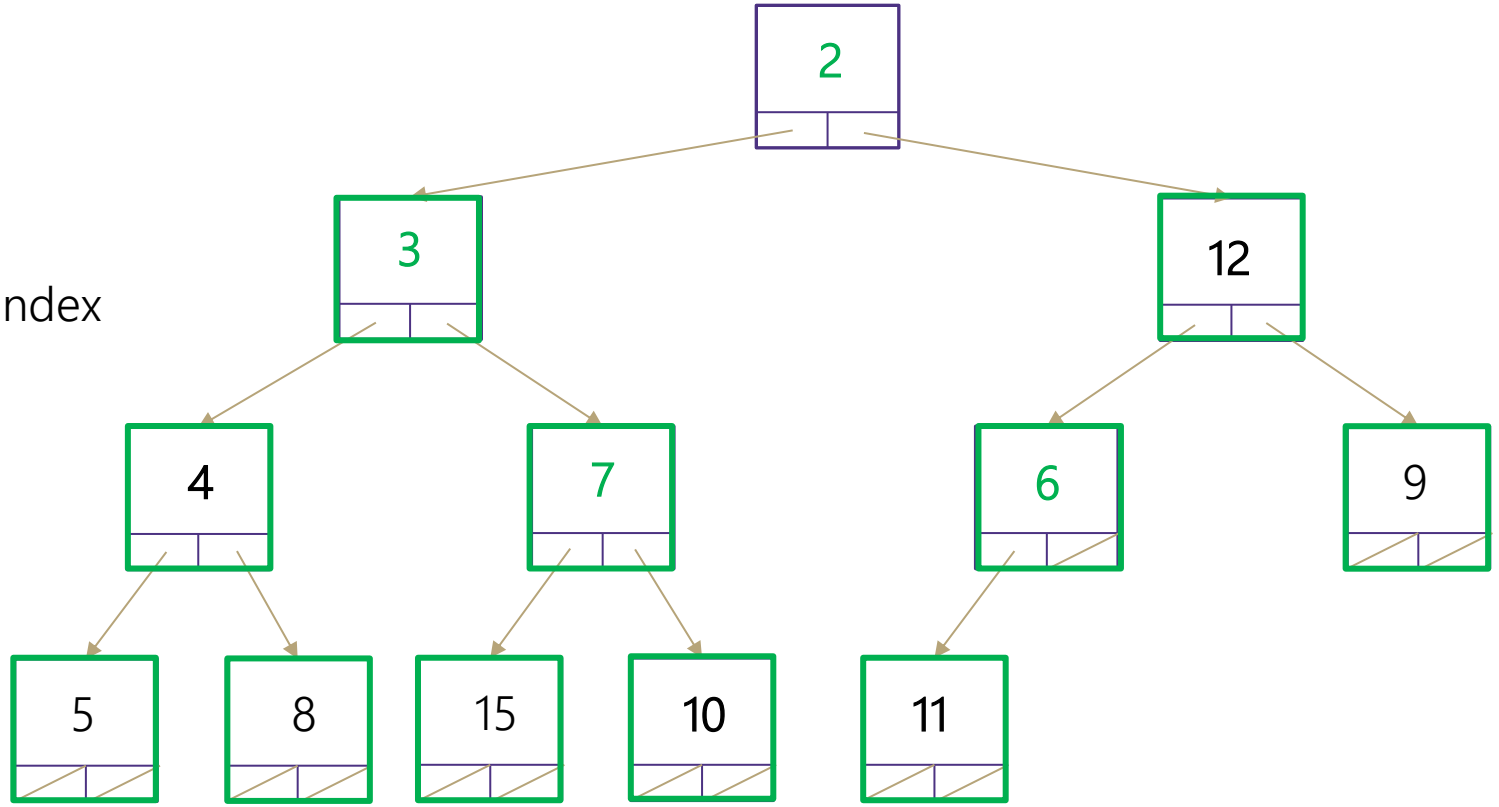




Floyd's buildHeap algorithm

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

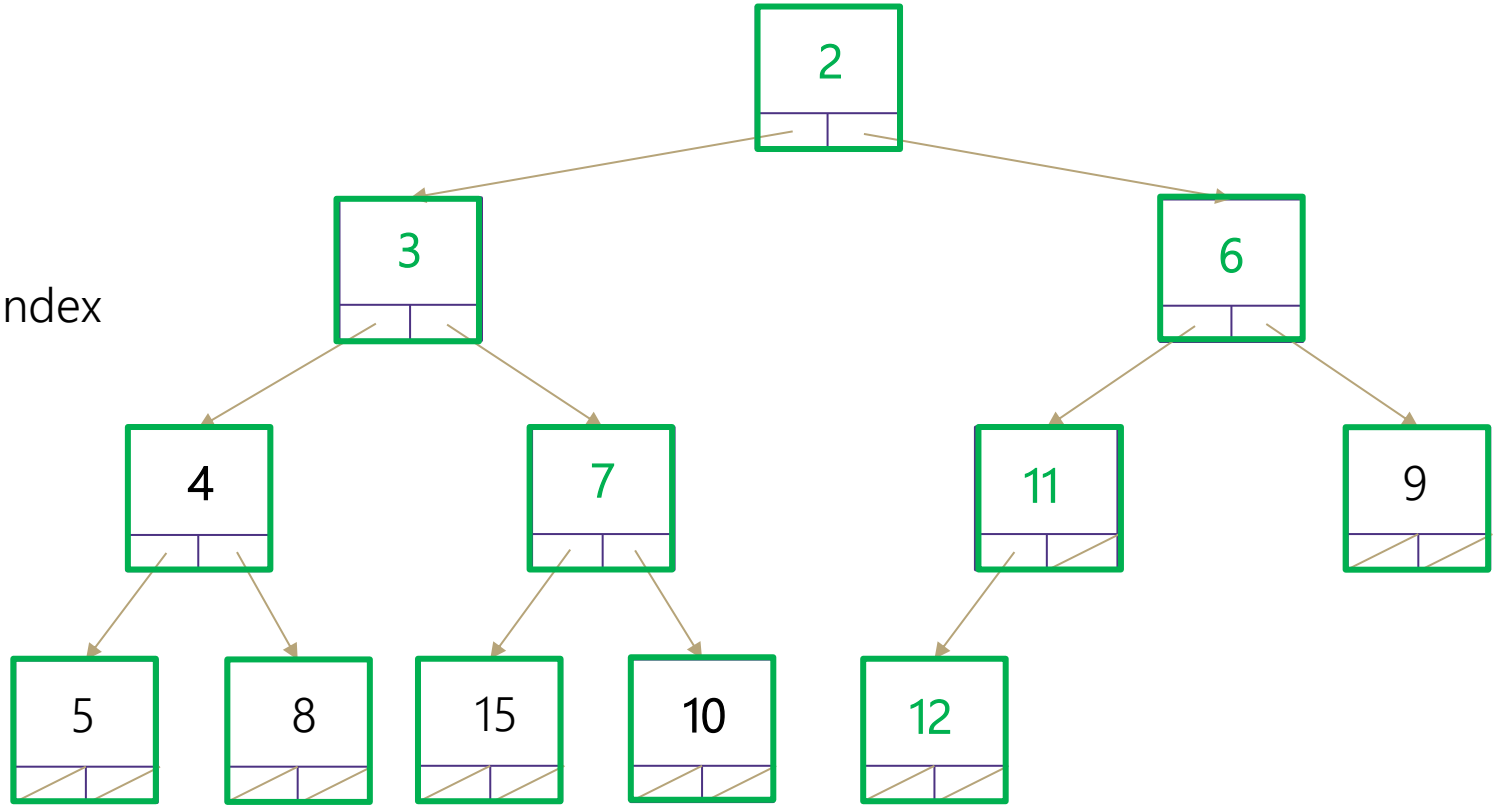


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	3	2	4	7	6	9	5	8	15	10	11		

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		

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.