LEC 13

CSE 373

Heaps II, Interviews

BEFORE WE START

Instructor Hunter Schafer

TAs
Ken Aragon
Khushi Chaudhari
Joyce Elauria
Santino Iannone
Leona Kazi
Nathan Lipiarski
Sam Long
Amanda Park
Paul Pham
Mitchell Szeto
Batina Shikhalieva
Ryan Siu
Elena Spasova
Alex Teng
Blarry Wang
Aileen Zeng

Announcements

- P2 due TONIGHT 11:59pm PDT
- P3 will come out today. Is also due in two weeks.
  - Very related to lectures this week!

Learning Objectives

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

1. (Continued) Trace the removeMin(), and add() methods, including percolateDown() and percolateUp()
2. Describe how a heap can be stored using an array, and compare that implementation to one using linked nodes
3. Recall the runtime to build a heap with Floyd's buildHeap algorithm, and describe how the algorithm proceeds

Lecture Outline

- Heaps II
  - Operations & Implementation
  - Building a Heap

- Technical Interviews

Review 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

★ We'll assume this one in 373!

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

Review 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

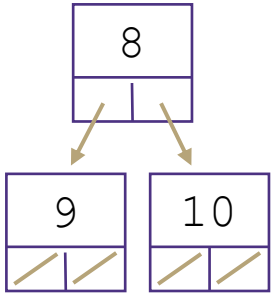
Heap Invariant

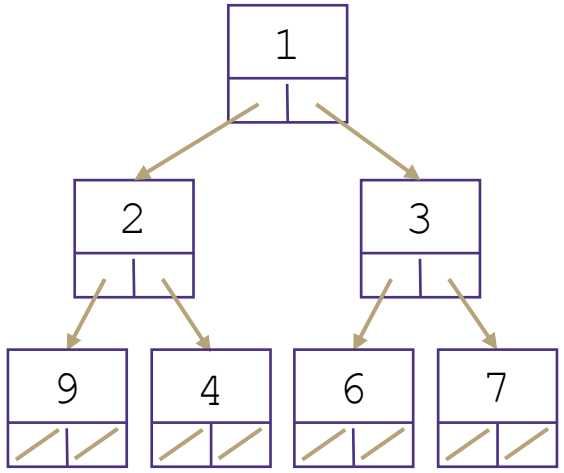
Every node is less than or equal to all of its children.

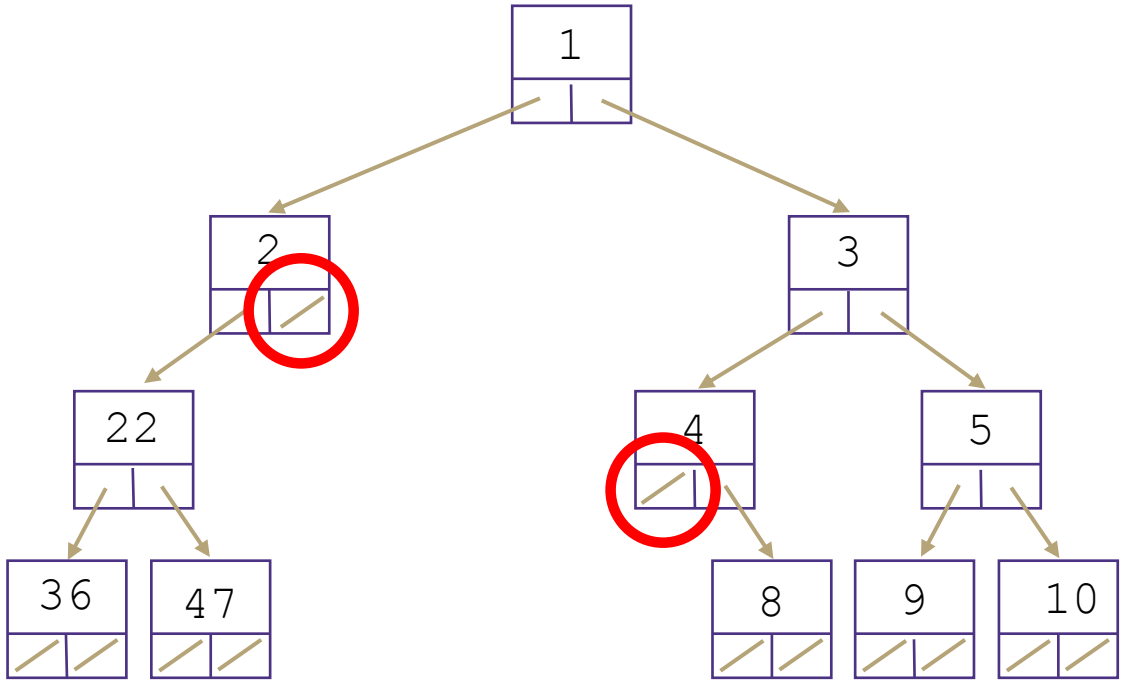
INVARIANT

Heap Structure Invariant

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

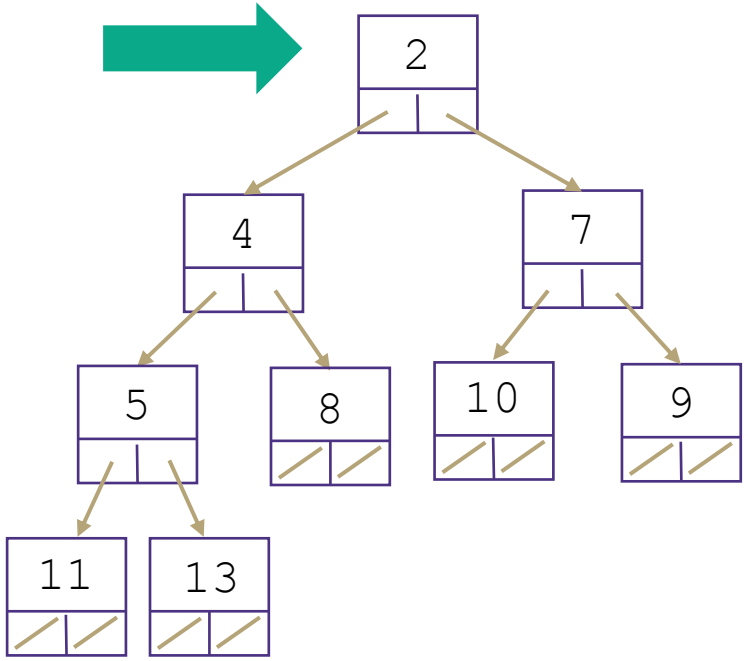






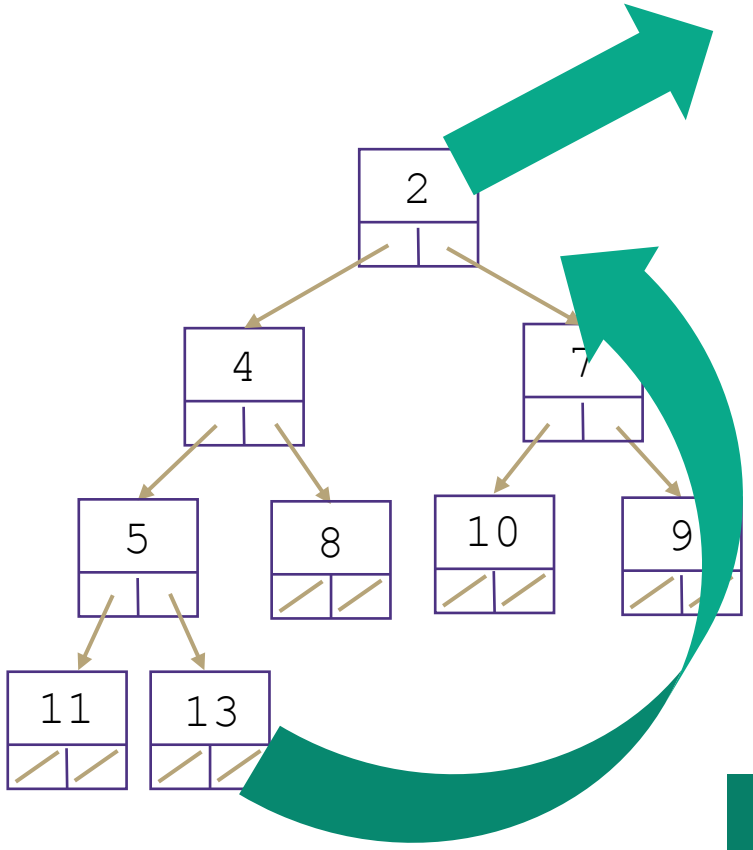
Review Implementing peekMin()

Runtime: Θ(1)

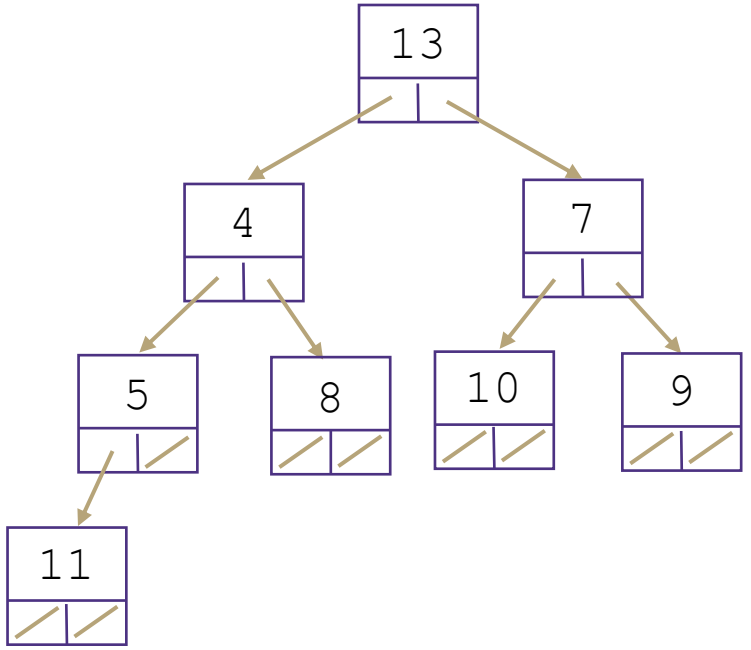


Simply return the value at the root! That's a constant-time operation if we've ever seen one ☺

Review Implement removeMin()



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 Structure Invariant

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

INVARIANT

Heap Invariant

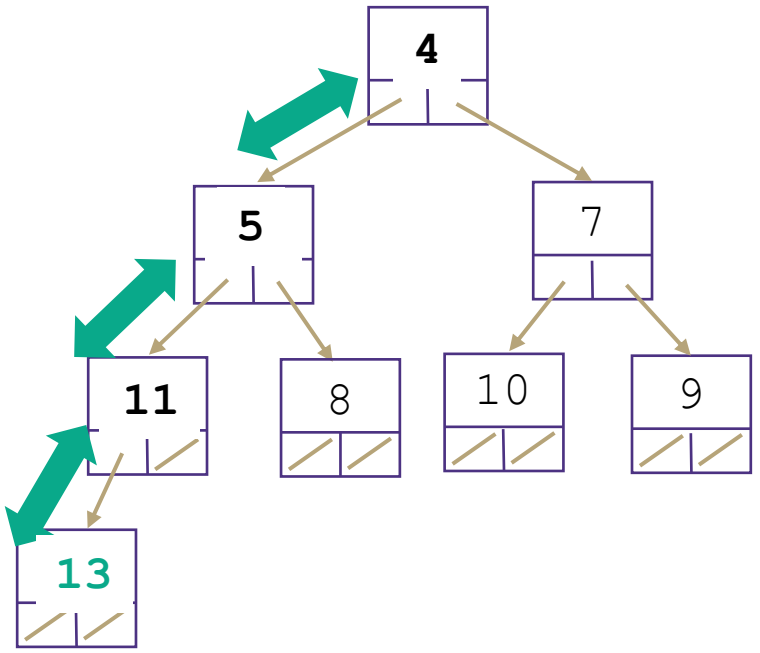
Every node is less than or equal to all of its children.

Structure Invariant restored, but Heap Invariant now broken

Review 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?

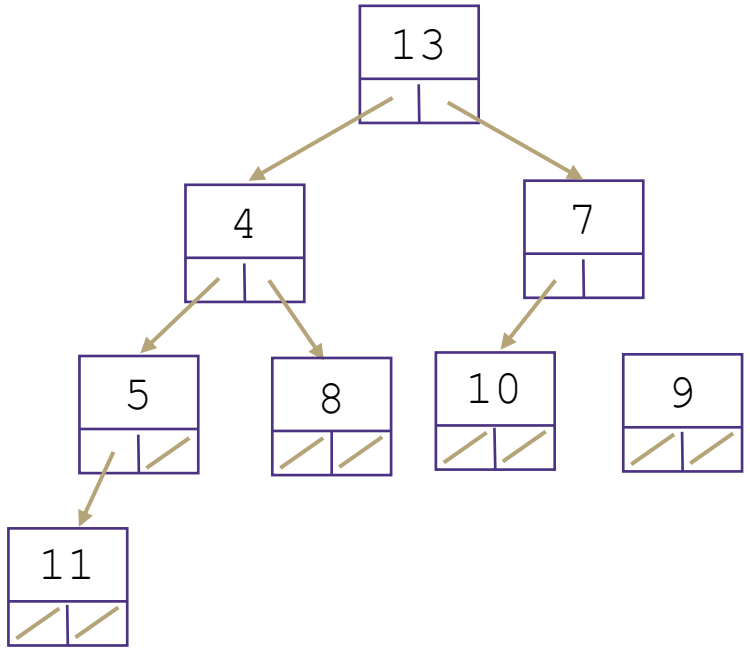
- Find last element
- Move it to top spot
- Swap until invariant restored

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

Review percolateDown: Why Smallest Child?

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

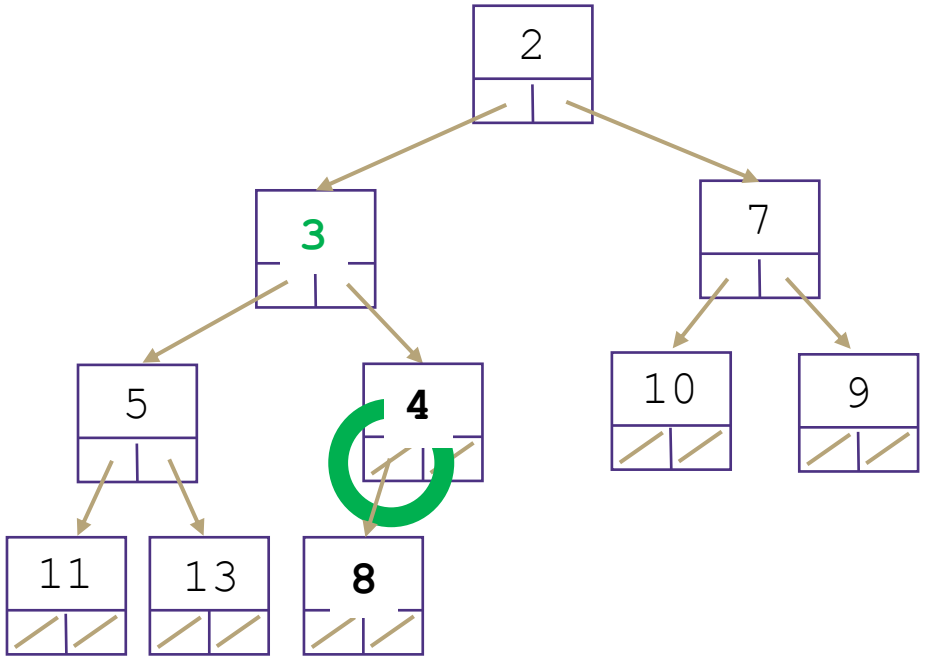


- If we swap 13 and 7, the heap invariant isn't restored!
- 7 is greater than 4 (it's not the smallest child!) so it will violate the invariant.

Review Implement add(key): percolateUp!

ADD ALGORITHM

- Insert node on the bottom level to ensure no gaps (Heap Structure Invariant)
- Fix Heap Invariant with new technique: `percolateUp`
  - Swap with parent, until your parent is smaller than you (or you're the root).



Worst case runtime similar to removeMin and percolateDown

- might have to do log(n) swaps, so the worst-case runtime is Θ(log(n))

MinHeap Runtimes

`removeMin()`:

1 — 1. Remove root node

→ log n — 2. Find last node in tree and swap to top level

log n — 3. Percolate down to fix heap invariant

`add(key)`:

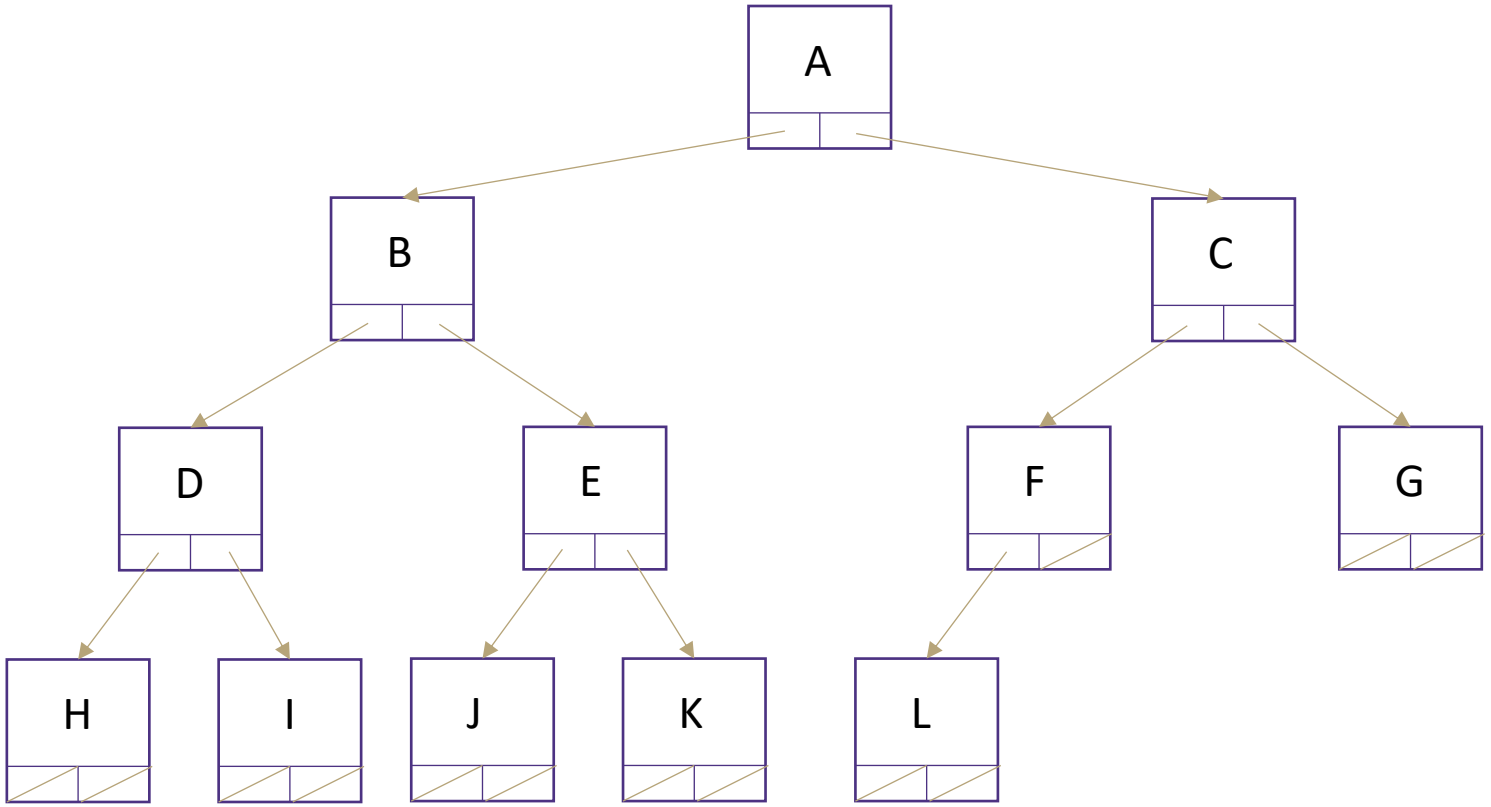
→ log n — 1. Find next available spot and insert new node

log n — 2. Percolate up to fix heap invariant

Operation	Case	Runtime
`removeMin()`	best	$\Theta(1)$
	worst	$\Theta(\log n)$
`add(key)`	best	$\Theta(1)$
	worst	$\Theta(\log n)$

- **Finding the "end" of the heap is hard!**
  - Can do it in $\Theta(\log n)$ time on complete trees with extra class variables by walking down
  - Fortunately, there's a better way ☺

Implementing Heaps with an Array

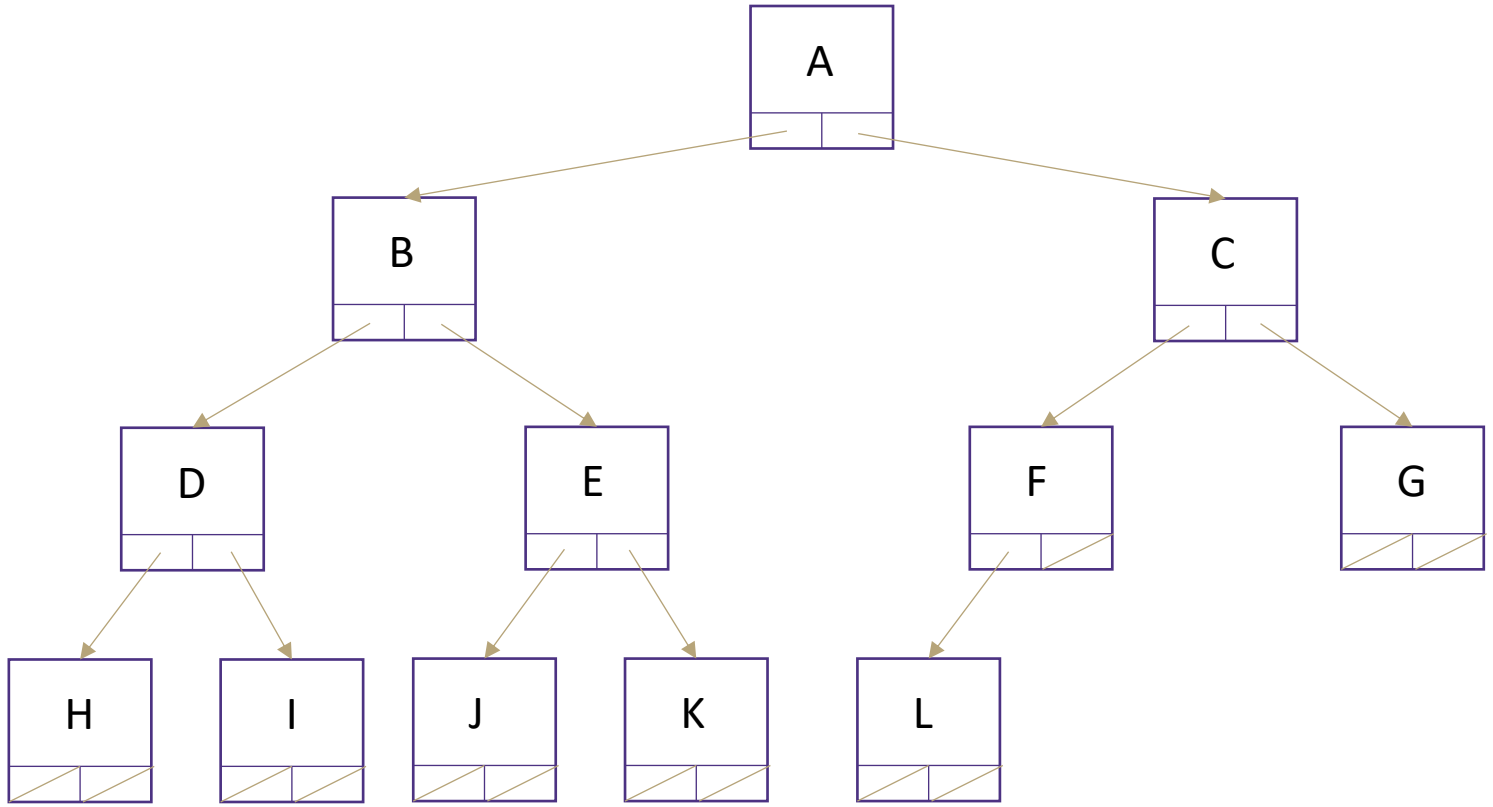


Fill array in **level-order** from left to right

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		

- Map our binary tree heap representation into an array
  - Fill in the array in level order from left to right
  - Remember, heaps are complete trees – very predictable number of nodes on each level!
- Note: array implementation is how people almost always implement a heap
  - But tree drawing is good way to think of it conceptually!
  - **Everything we've discussed about the tree is still true** – these are just different ways of looking at the same thing

Implementing 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		

Calculations to navigate array:

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?

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

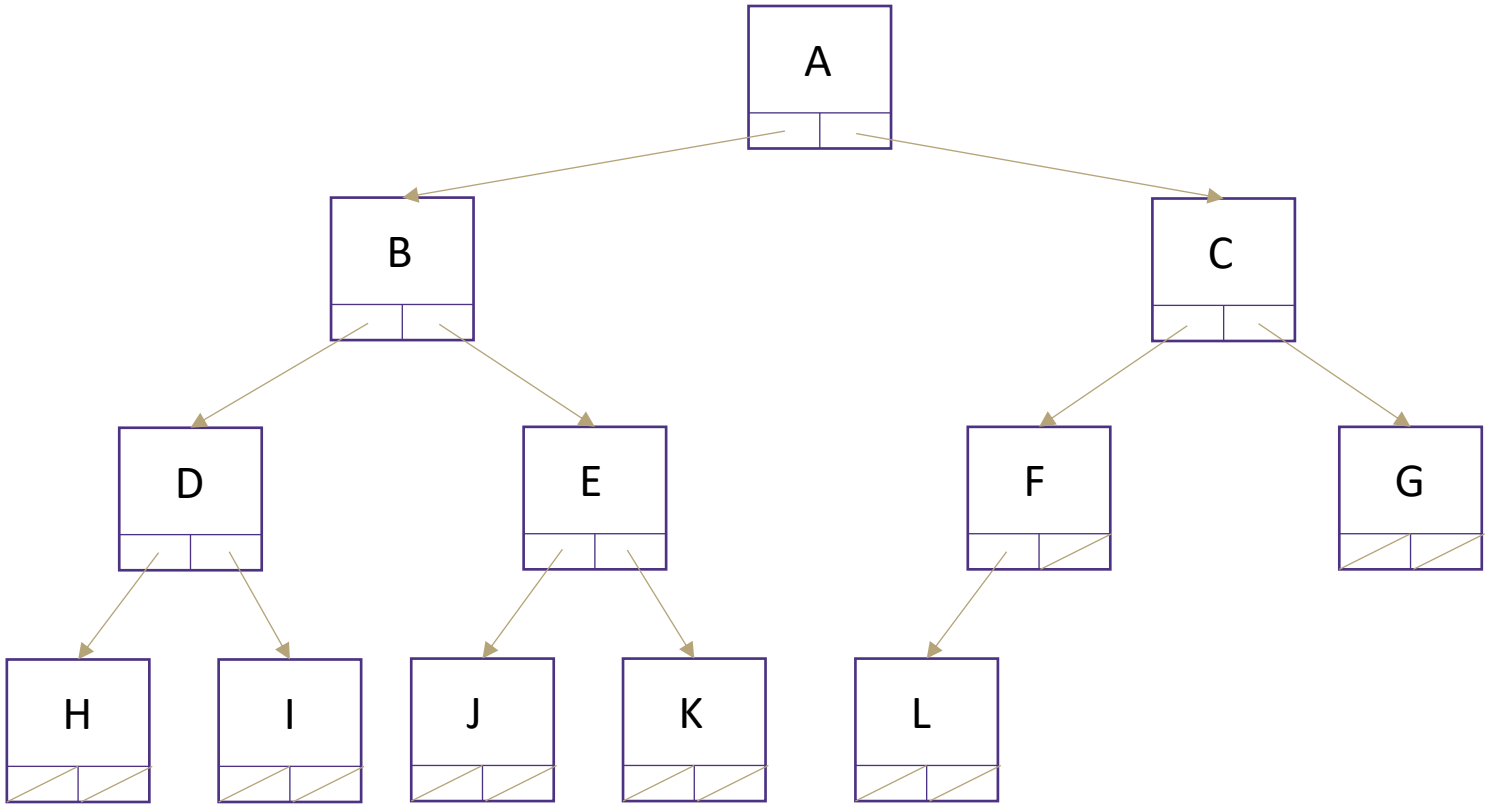
How do we find a node's right child?

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

How do we find a node's parent?

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

Implementing 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	

***Simplified* calculations to navigate array, if we skip index 0:**

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?

$$leftChild(i) = 2i$$

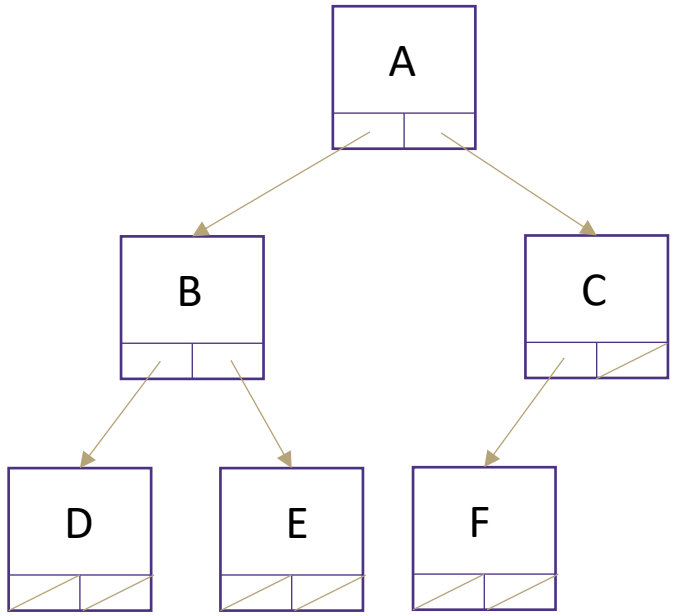
How do we find a node's right child?

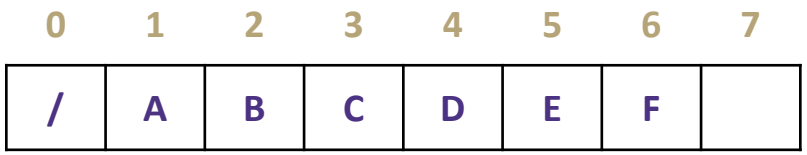
$$rightChild(i) = 2i + 1$$

How do we find a node's parent?

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

Array-Implemented MinHeap Runtimes

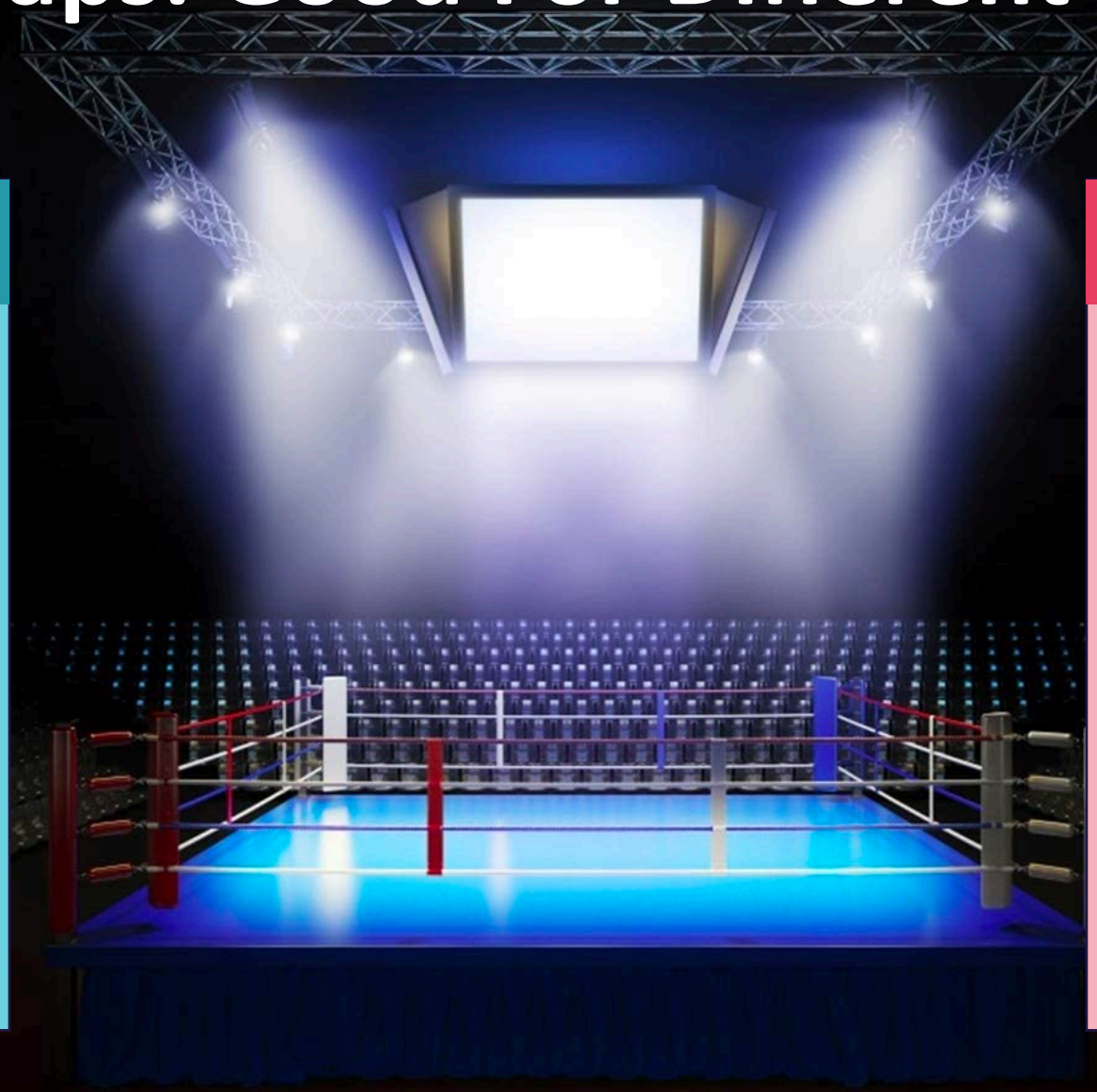




Operation	Case	Runtime
`removeMin()`	best	Θ(1)
	worst	Θ(log n)
	in practice	Θ(log n)
`add(key)`	best	Θ(1)
	worst	Θ(log n)
	in practice	Θ(1)
`peekMin()`	all cases	Θ(1)

- 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 Θ(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

Lecture Outline

- **Heaps II**
  - Operations & Implementation
  - **Building a Heap**
- Design Decisions
- Technical Interviews

Building a Heap

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

- Idea 1: Call add n times.
  - Worst case runtime?
    - Each call takes logarithmic time, and there are n calls
    - Θ(n log n)
    - (Technically, the worst case is not this simple – you're not always going to hit logarithmic runtime because many insertions happen in a pretty empty tree – but this intuition is good enough)
  - Could we do better?

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)

Can We Do Better?

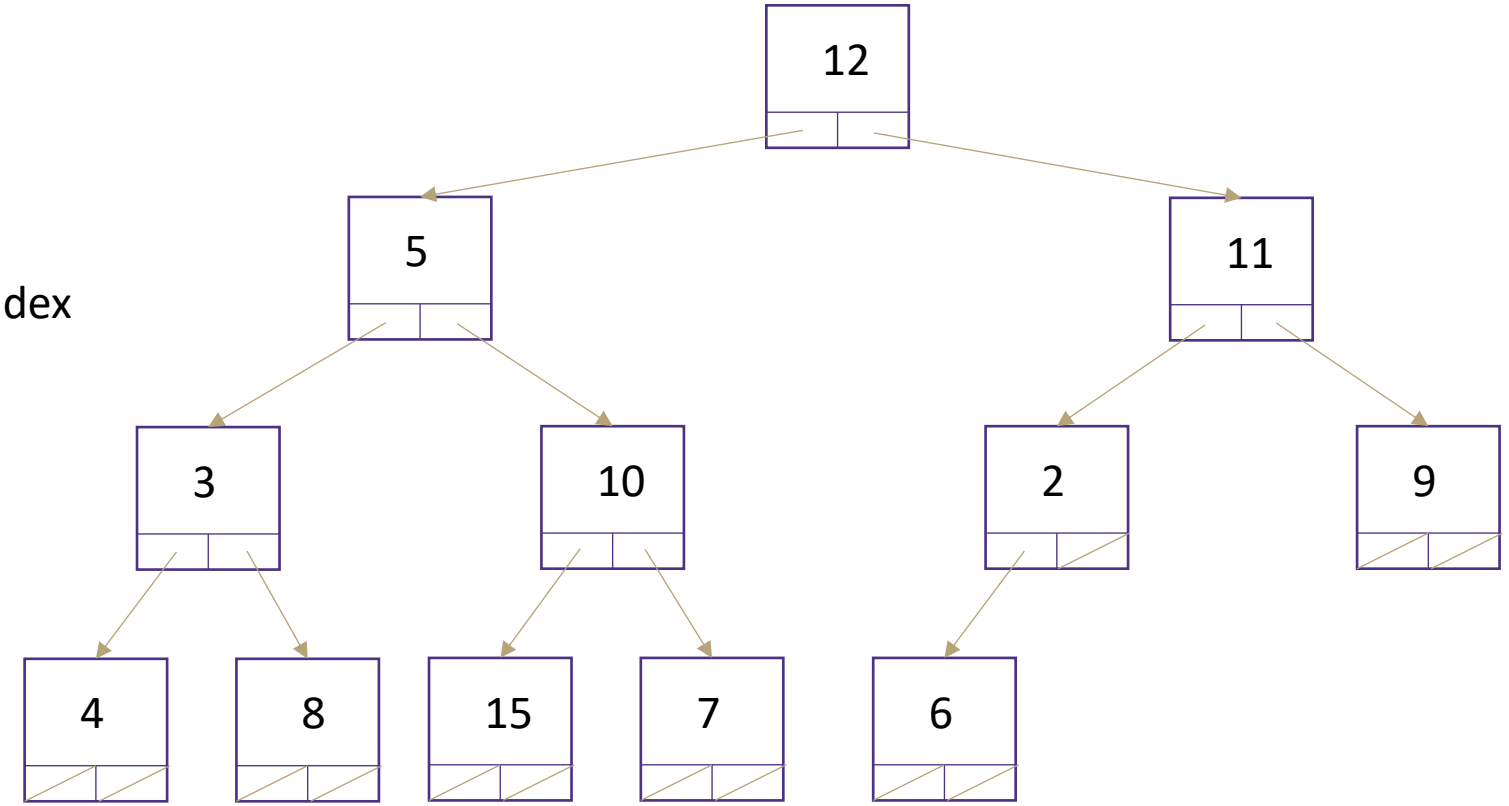
- What's causing the n add strategy to take so long?
  - Most nodes are near the bottom, and might need to percolate all the way up.

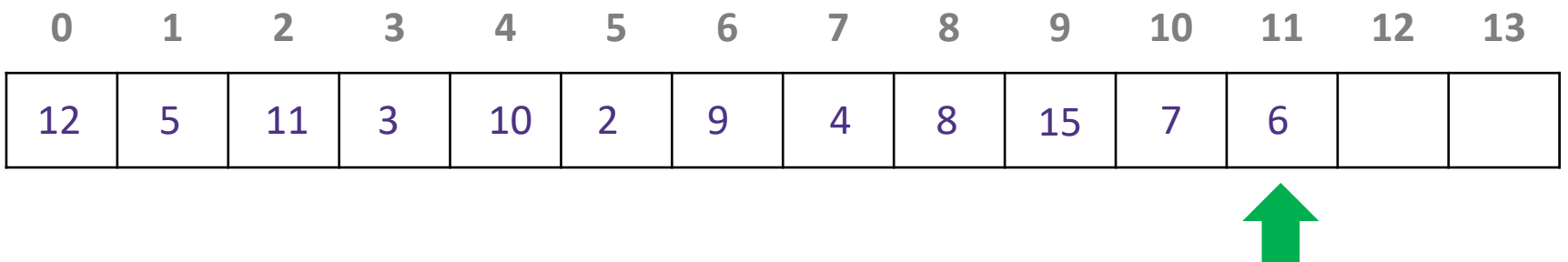
- Idea 2: Dump everything in the array, and percolate things down until the heap invariant is satisfied
  - Intuition: this could be faster!
  - The bottom two levels of the tree have $\Omega(n)$ nodes, the top two have 3 nodes
  - Maybe we can make "most of the nodes" go only a constant distance

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

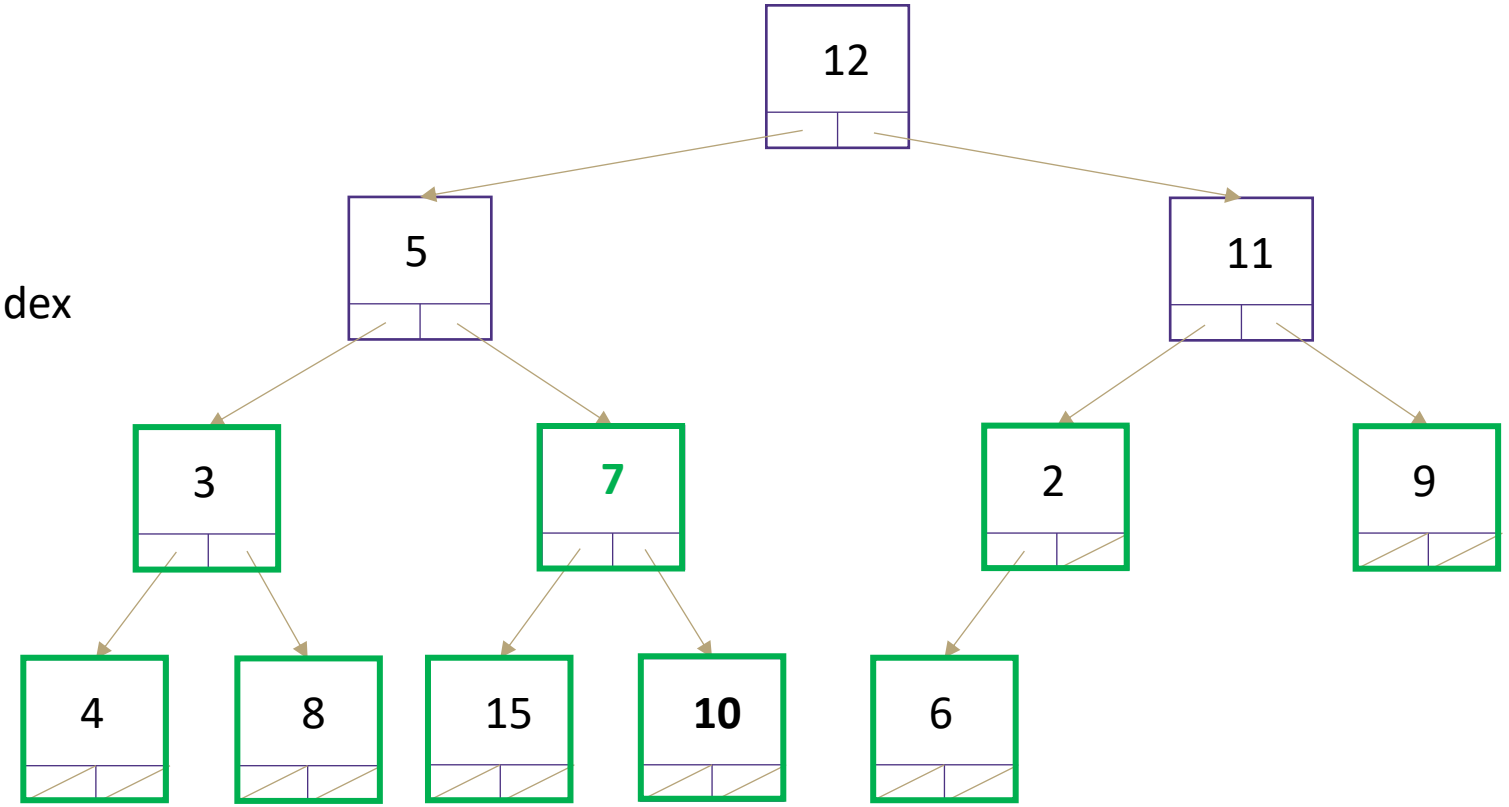




Floyd's buildHeap algorithm

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



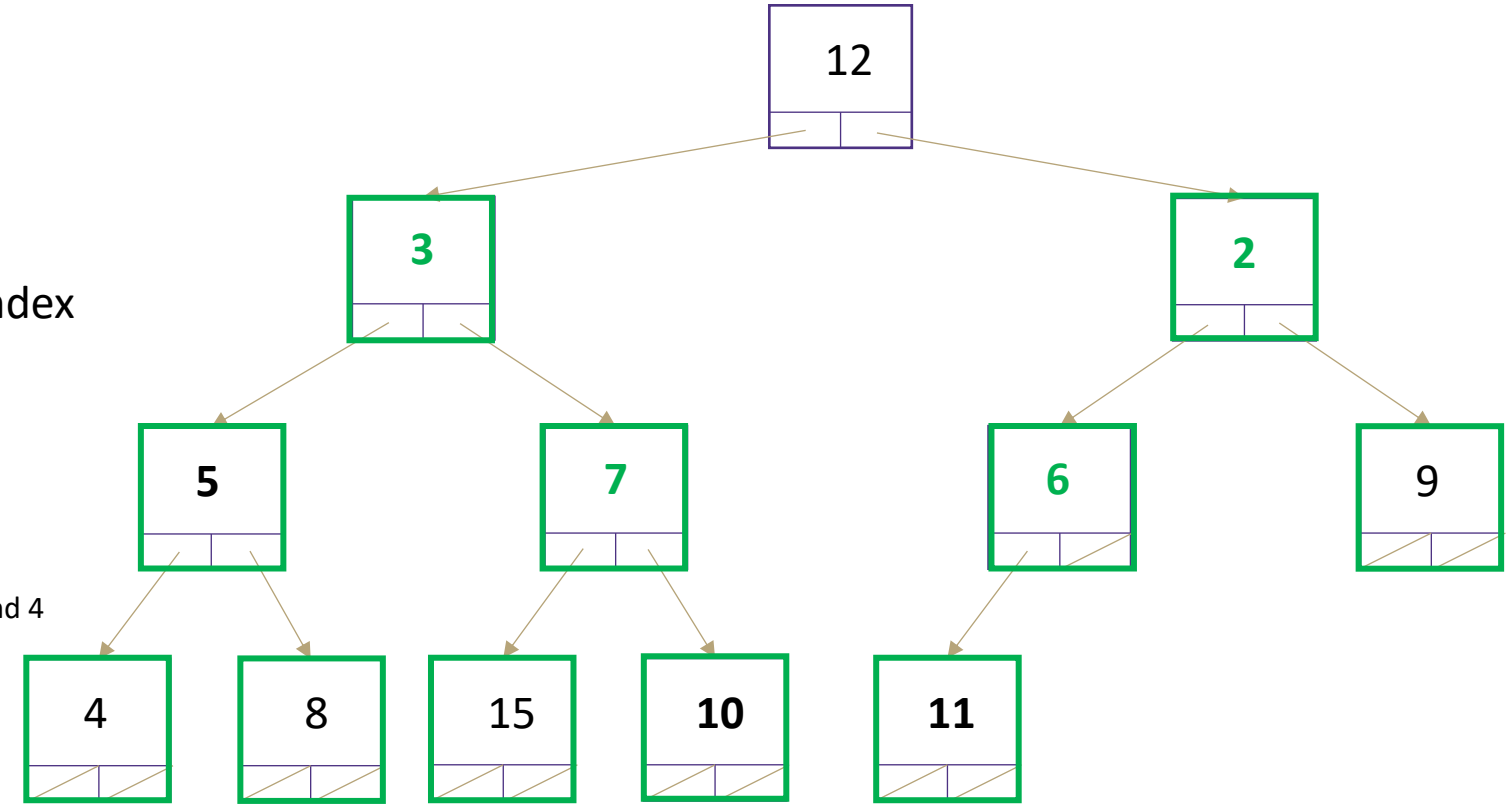
0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	15	7	6		

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

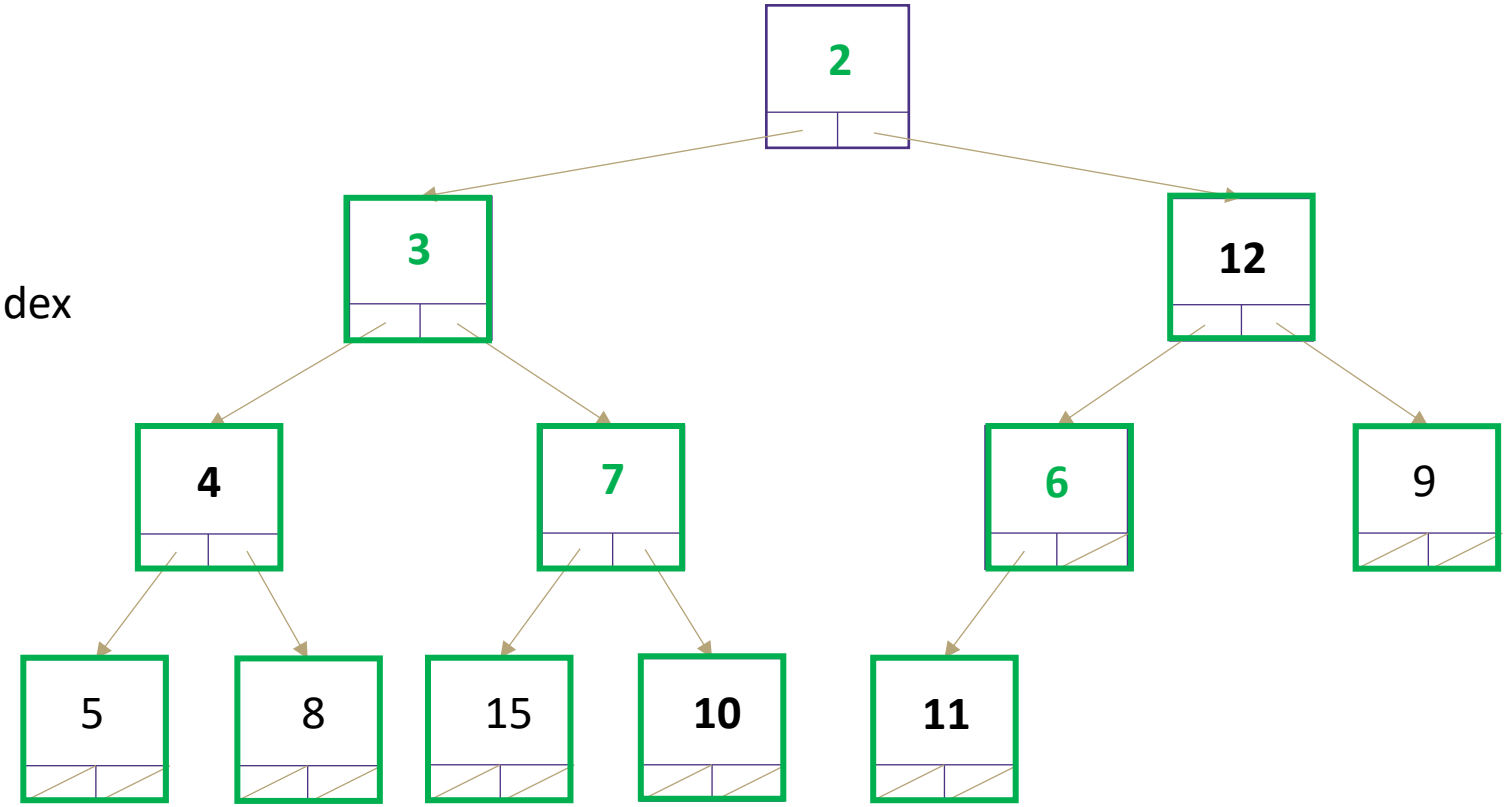


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	7	2	9	4	8	15	10	6		

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

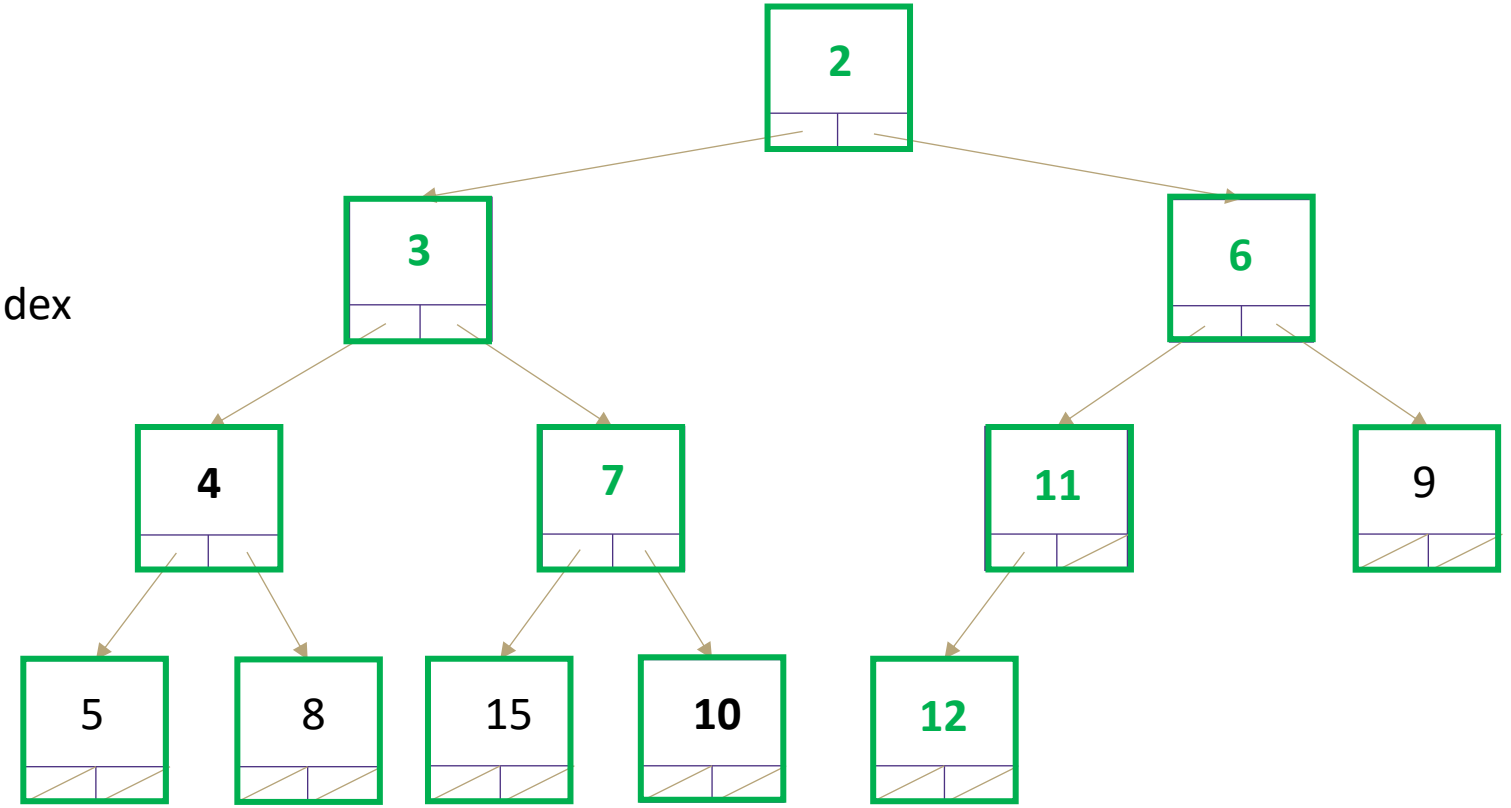


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	3	2	4	7	6	9	5	8	15	10	11		

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…
- 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

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

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

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*

Lecture Outline

- Heaps II
  - Operations & Implementation
  - Building a Heap

- **Technical Interviews**

Beyond CSE 373: Industry

- Many people take CSE 373 because they're interested in a career related to software engineering or more broadly CS
  - If not, that's totally okay too! There are so many good reasons to take this class and ways to apply it; you don't have to be interested in software engineering specifically
  - Perhaps you've become more interested over the course of the quarter

- If this sounds like you, it's never too early to start thinking about preparing for a job/internship hunt!
  - We'll talk about this a few more times throughout the course
  - But a few highlights now to get you thinking!

The Technical Interview Process

PREPARATION

- Putting together your resume
- Personal projects/other experience to help you stand out
- Identifying where to apply, and when

CSE 373 MATERIAL

- Practicing with interview problems & design decisions
- Identifying common patterns in interview questions

PROCESS

1 *Applying*

Online Apps

Career Fairs

2 *Interviewing*

Technical

Behavioral

Job/Internship Offer

1 *Applying*

Start Here:

http://bit.ly/cseresumeguide

An incredibly useful guide to fleshing out your resume, highlighting your experience to make you stand out, Do's and Don'ts of what to include

- http://bit.ly/csestorycrafting – how to turn your experience into a cohesive story that stands out to recruiters
- UW Career & Internship Center – tons of helpful articles, *especially* for online job hunting!
- College of Engineering Career Center – schedule an appointment to talk to a career counselor

2 *Interviewing*

This is where CSE 373 comes in!

- Technical interviews *love* to ask about maps, trees, heaps, recursive algorithms, and **especially algorithmic analysis!**
- Making design decisions and determining tradeoffs between data structures is a crucial skill! Fortunately, you've been practicing all quarter ☺

- Leetcode and Hackerrank – *tons* of practice interview questions. If you're feeling nervous, it always helps to practice ☺
  - Check out #career-prep on Discord! Your amazing TA Joyce has been highlighting example interview problems, and more and more options available as we learn more this quarter!
- The Secrets No One Told You About Technical Interviews – fantastic article *(& by previous 373 instructor Kasey Champion!)*
- Approaching Technical Interview Questions – when you get asked a question that stumps you, what should you do?

A+ Advice for Getting a Software Job

- A guest lecture by the amazing Kim Nguyen, former Career Coach for UW CSE, **specifically for 373 students!**
  - Back when a "lecture" was something that happened in person…
  - If you're overwhelmed with all these resources, **highly recommend you start here!** Kim is the absolute best!