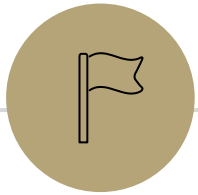


Lecture 11

Heaps

Department of Computer Science
Hofstra University



Priority Queue *ADT*

Binary Heap

Binary Heap Methods

Priority Queue ADT

Priority Queues are commonly used for sorting

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	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
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.

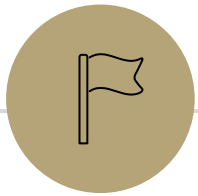
Implementing Priority Queues: Take I

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



Priority Queue ADT

Binary Heap

Binary Heap Methods

Heaps

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

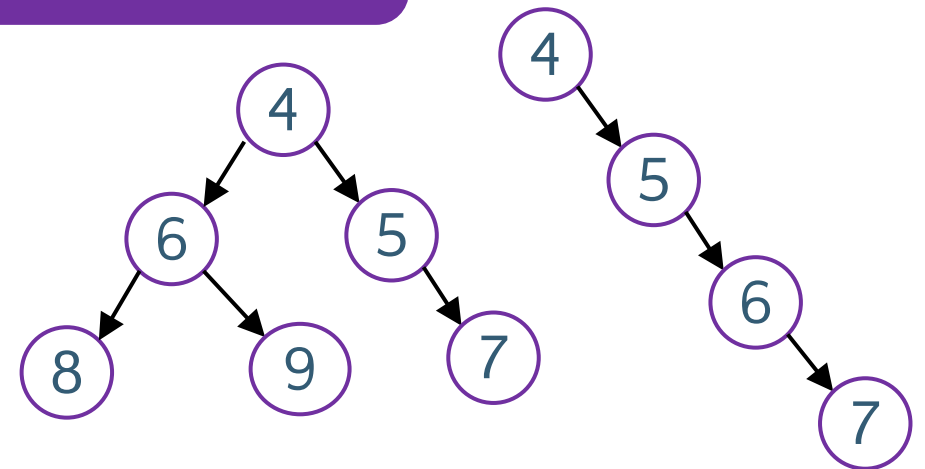
Now we just want to find the **smallest** item fast, so let's write a different invariant:

Heap invariant

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

In particular, the smallest node is at the root!

Do we need more invariants?



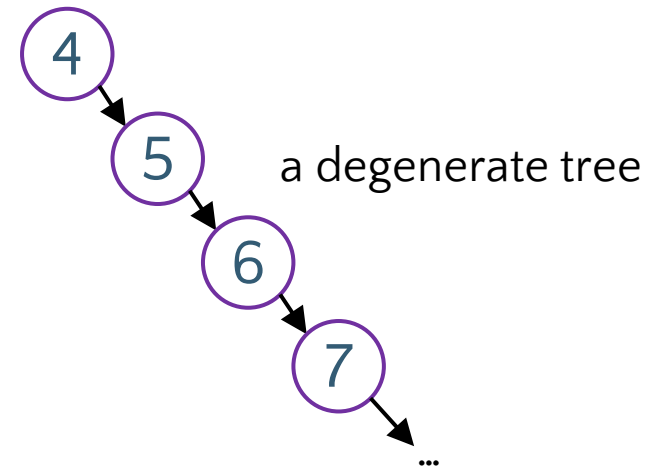
Heaps

We want to avoid degenerate trees (linear linked lists).

The heap invariant is less strict (looser) than the BST invariant, so we can impose stricter invariants on tree structure

- A BST is an ordered, or sorted, binary tree, with the following invariants:
- For every node with key k :
 - The left subtree has only keys smaller than k
 - The right subtree has only keys greater than k
 - This invariant applies recursively throughout tree

Recall: BST Invariant



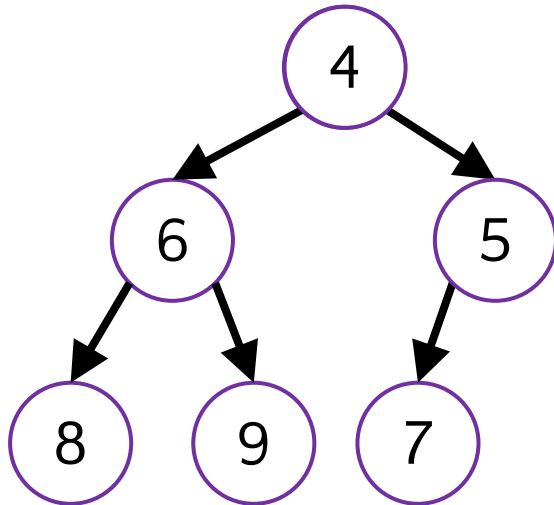
Heaps

Heap structure invariant:
A heap is always a **complete** tree.

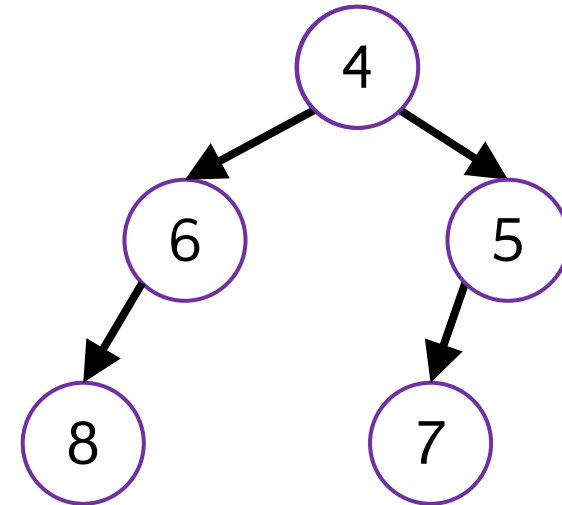
→ helps to avoid degenerate trees

A tree is complete if:

- Every row, except potentially the last, is completely full
- The last row is filled from left to right (no “gap”)

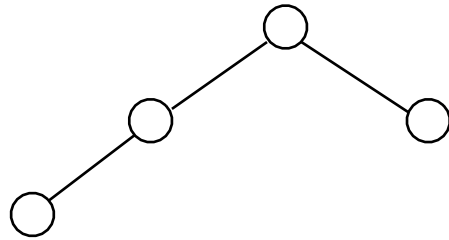
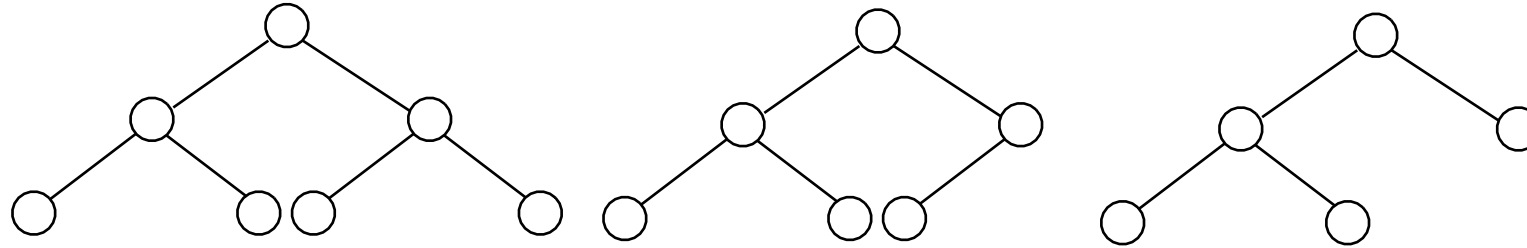


complete

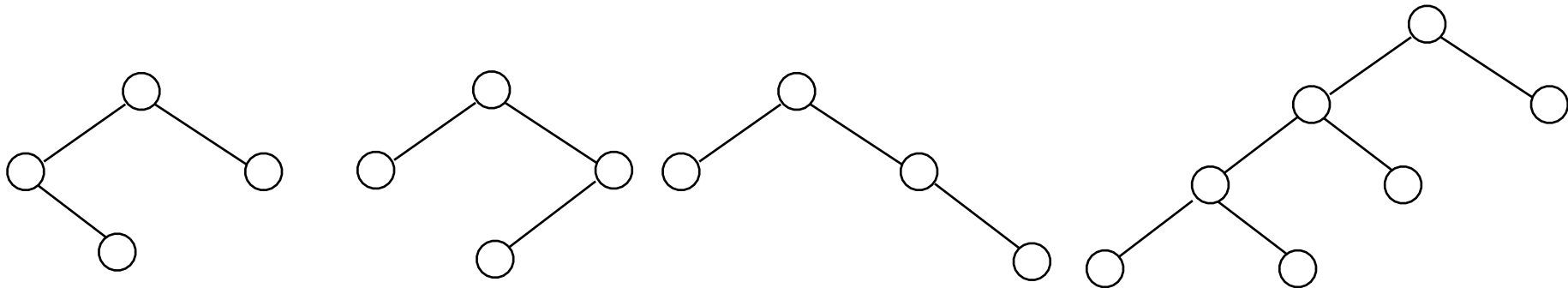


not complete

Complete Binary Tree or Not?



Above: complete binary trees
Below: not complete binary trees



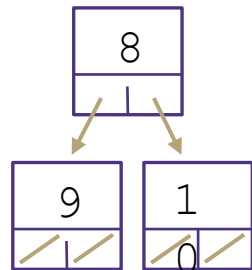
Leaf level is not filled from left to right.

Non-leaf level is not completely filled.

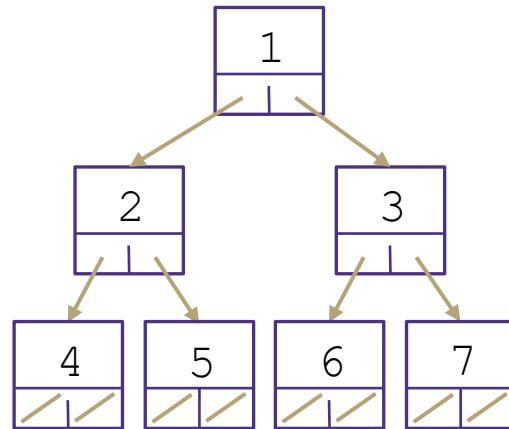
Binary Heap invariants

A **binary** heap satisfies the following invariants:

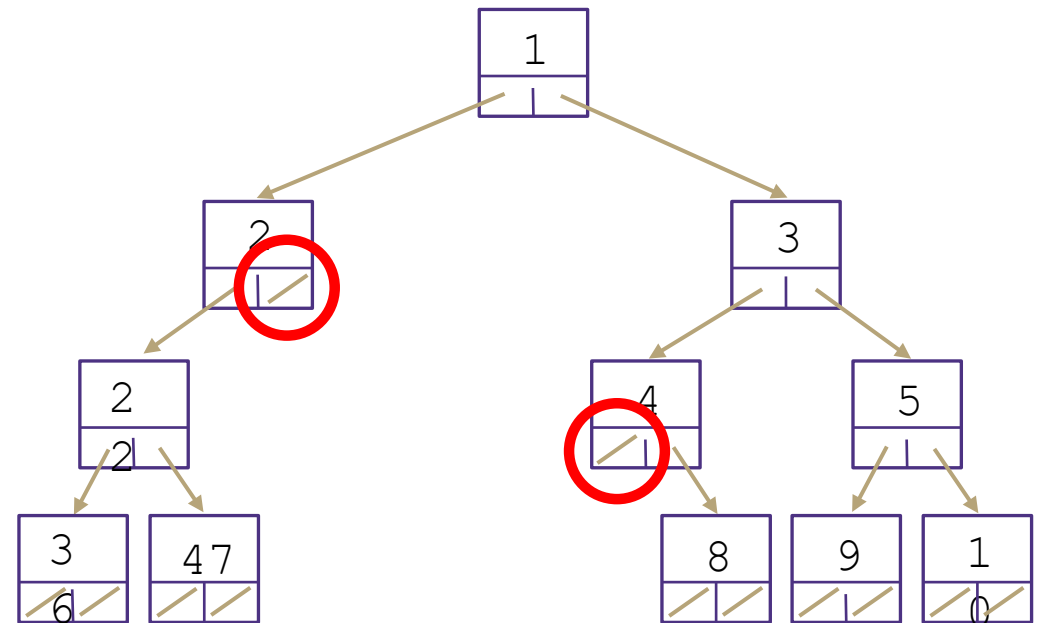
1. **Binary Tree**: every node has at most 2 children
2. **Heap invariant**: every node is smaller than (or equal to) its children
3. **Heap structure invariant**: each level is “complete” meaning it has no “gaps”
 - a. Heaps are filled up left to right



Valid heap



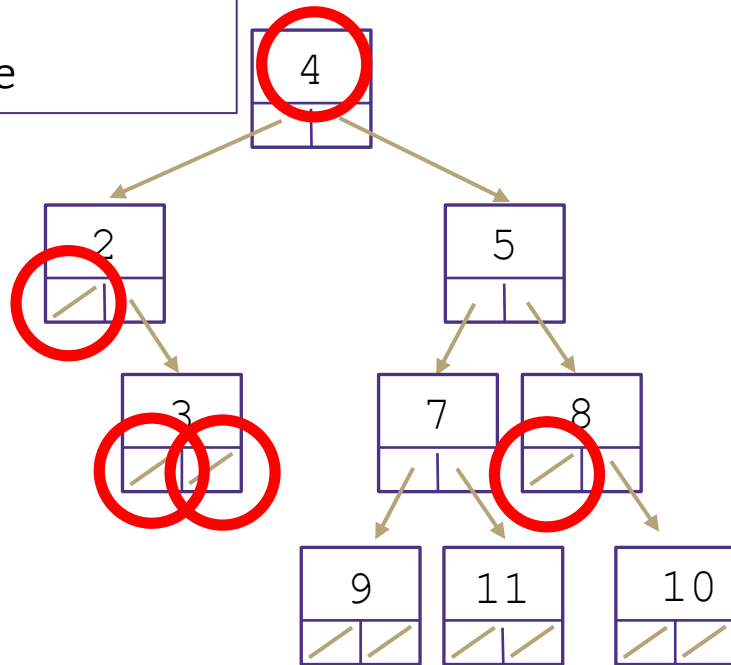
Valid heap



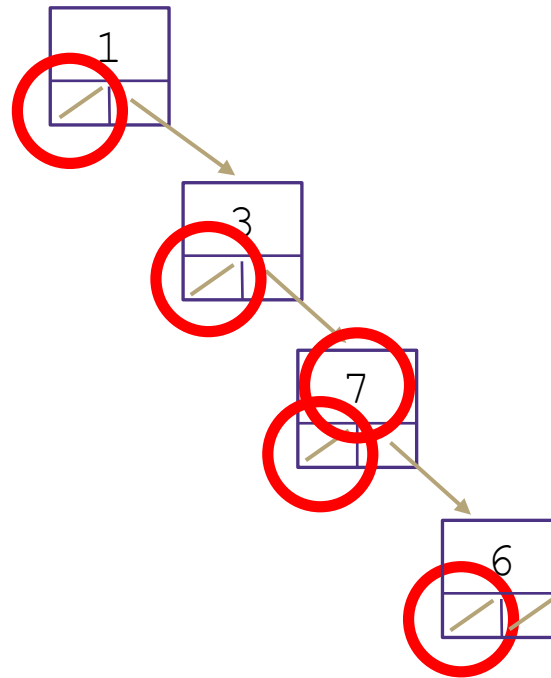
Invalid heap

Quiz - Are these valid heaps?

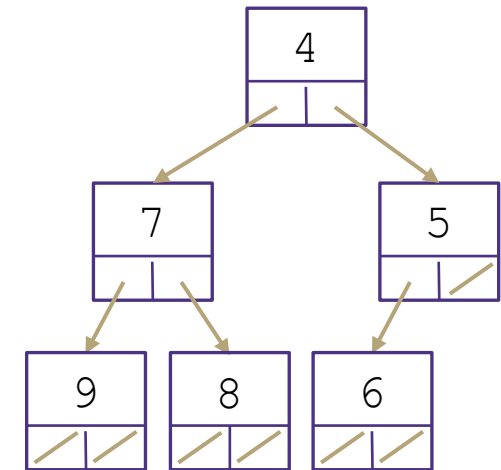
Binary Heap
Invariants:
1. Binary Tree
2. Heap
3. Complete



Invalid



Invalid



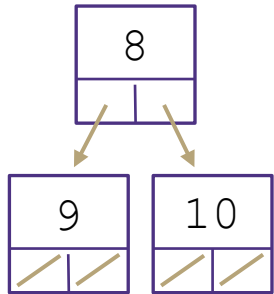
Valid

Quiz – Are these valid heaps?

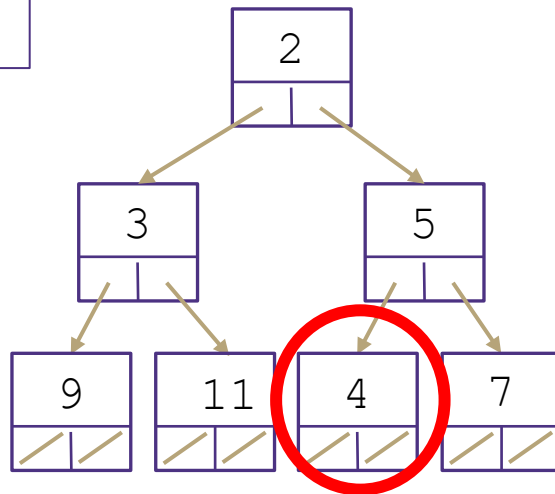
Binary Heap

Invariants:

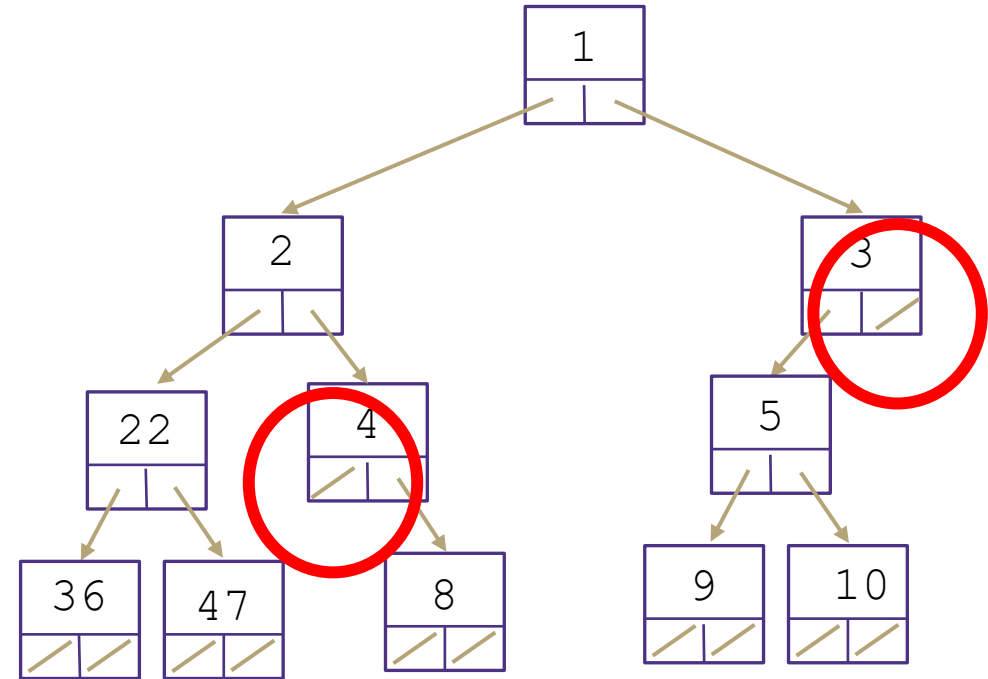
1. Binary Tree
2. Heap
3. Complete



Valid



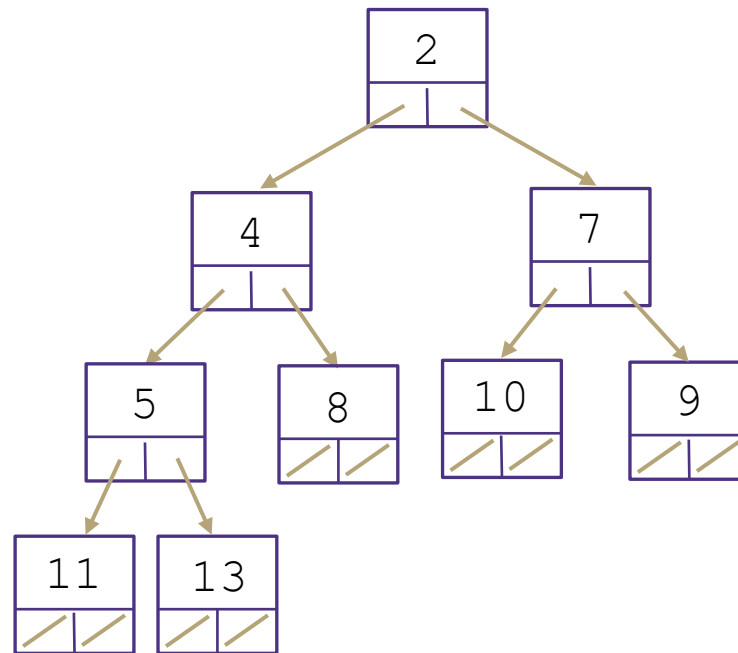
Invalid



Invalid

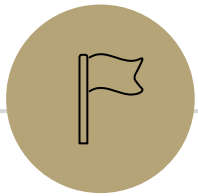
Heap heights

A binary heap bounds our height at $O(\log(n))$ because it's complete. This means the runtime to traverse from root to leaf or leaf to root will be $\log(n)$ time.



Priority Queue ADT

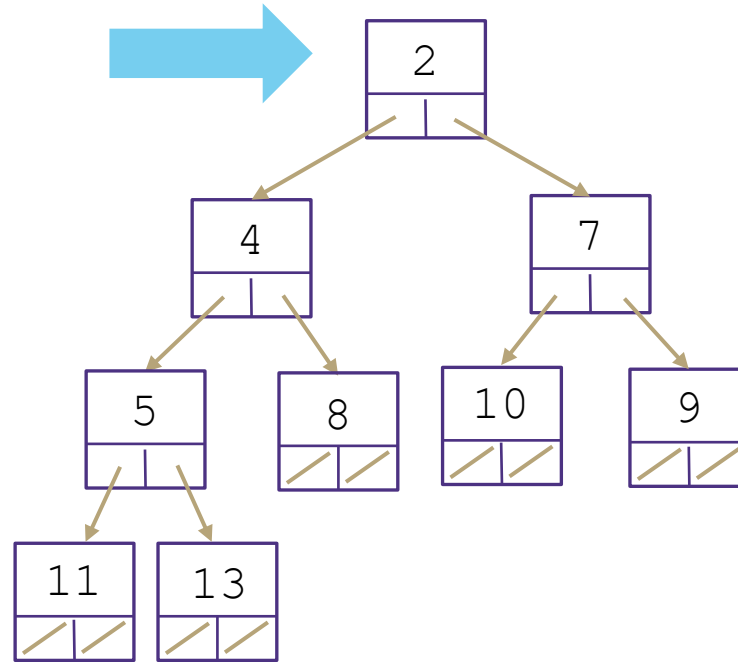
Binary Heap



Binary Heap Methods

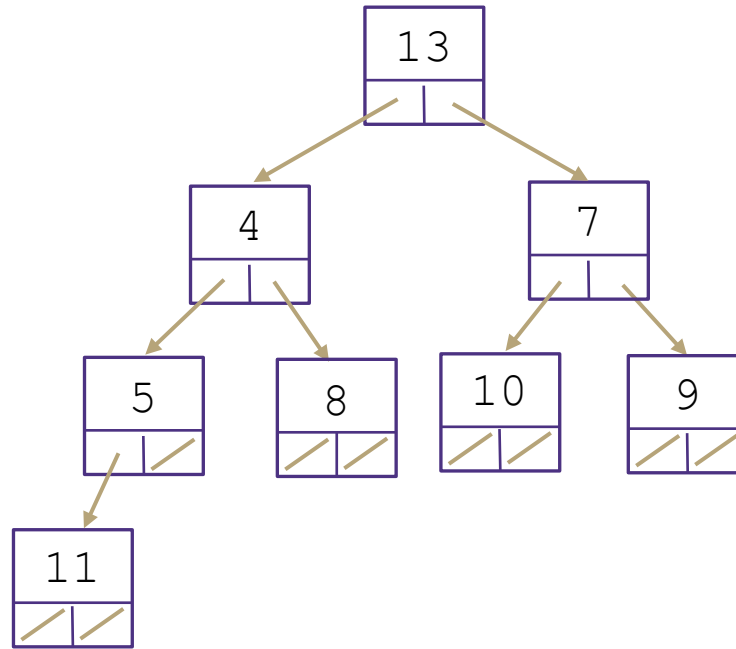
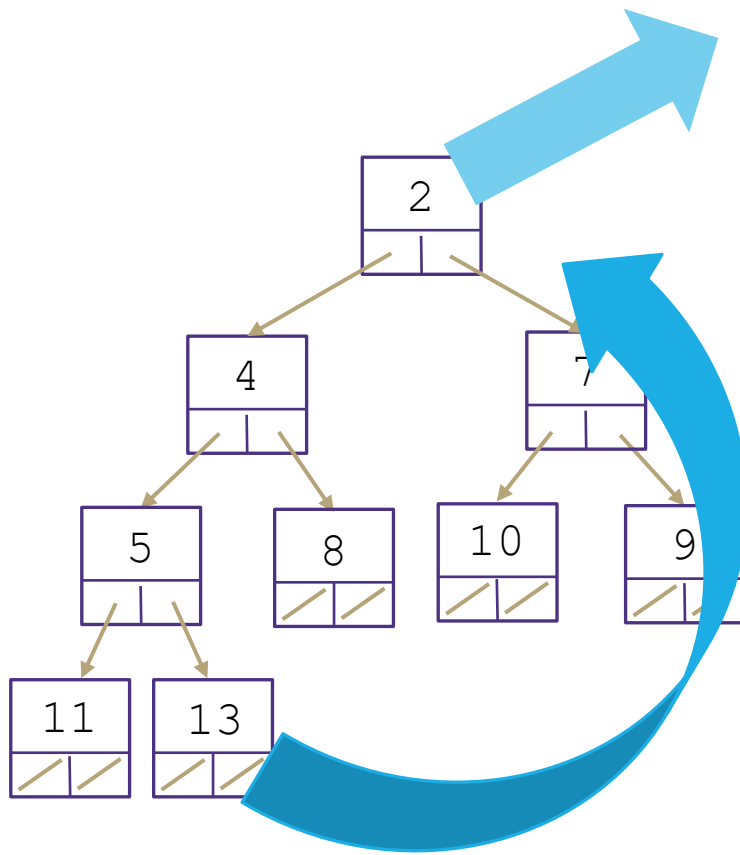
Implementing peekMin()

Runtime: $\Theta(1)$



Implementing removeMin()

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

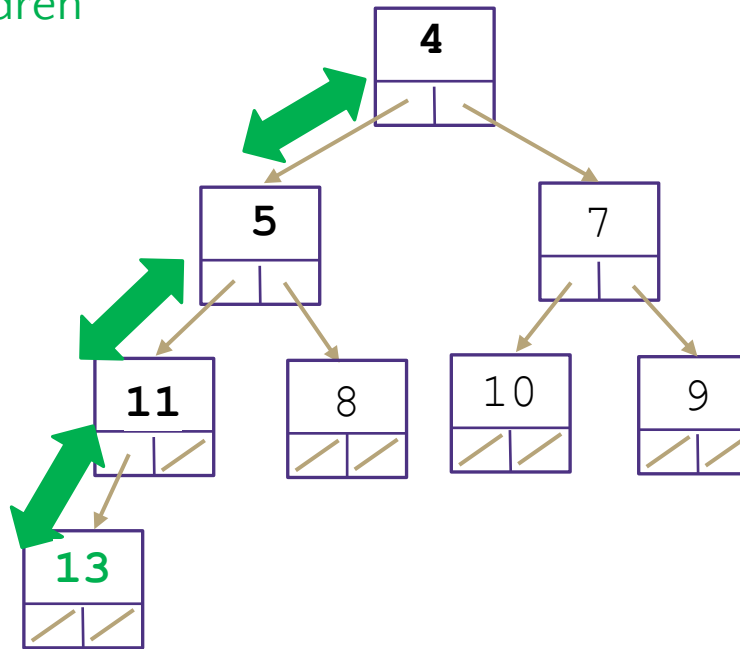


Structure invariant restored
Heap invariant broken

Implementing removeMin() – percolateDown

1. Return min
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).



Structure invariant restored
Heap invariant restored

What's the worst-case running time?

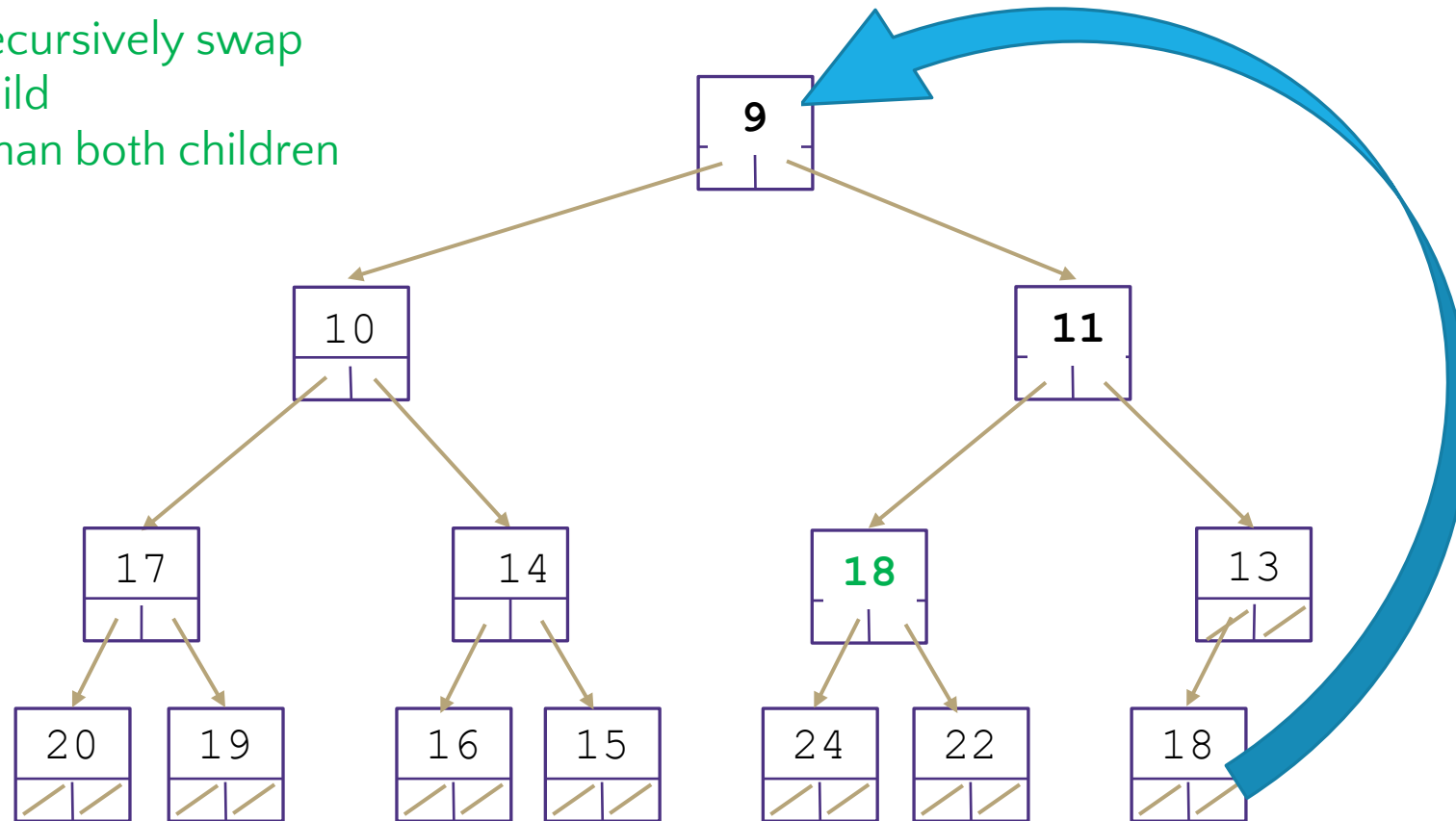
Have to:

- Find last element
- Move it to top spot
- Swap until invariant restored
- Number of swaps is $O(\text{TreeHeight})$

Hence we want to keep tree height small, as tree height (BST, AVL, heaps) directly correlates with worst-case runtimes

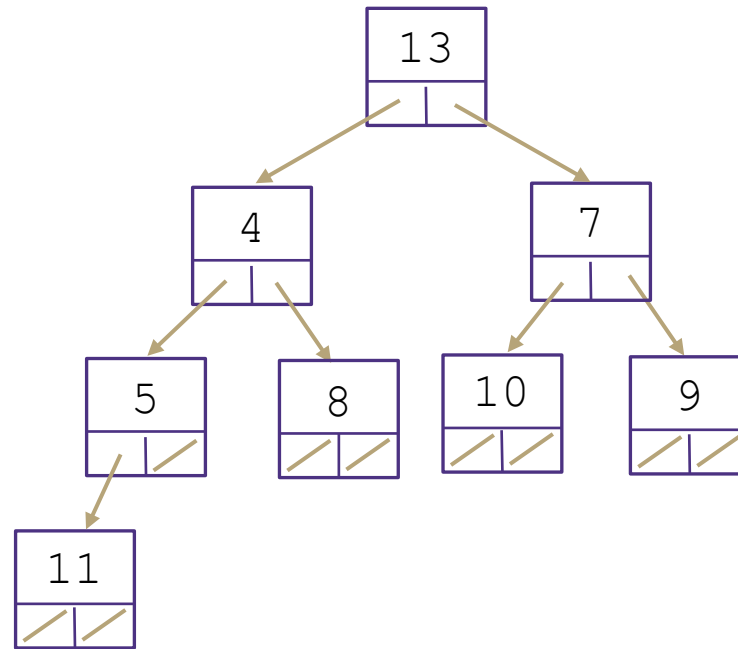
Practice: removeMin()

- 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).



percolateDown()

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!

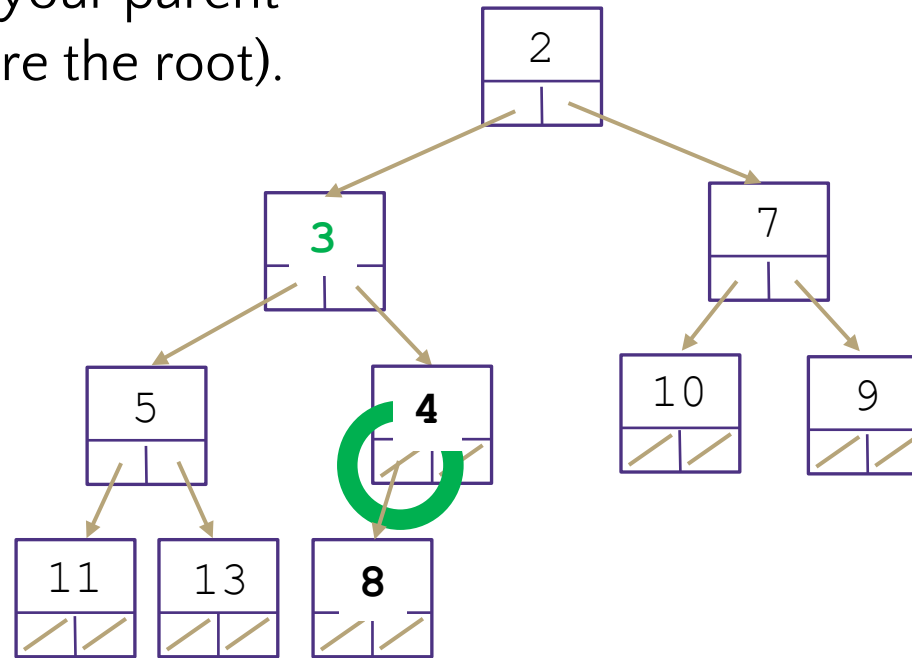
7 is greater than 4 (it's not the smallest child!) so it will violate the invariant.

Implementing add()

add() Algorithm:

1. Insert a node on the bottom level that ensure 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).

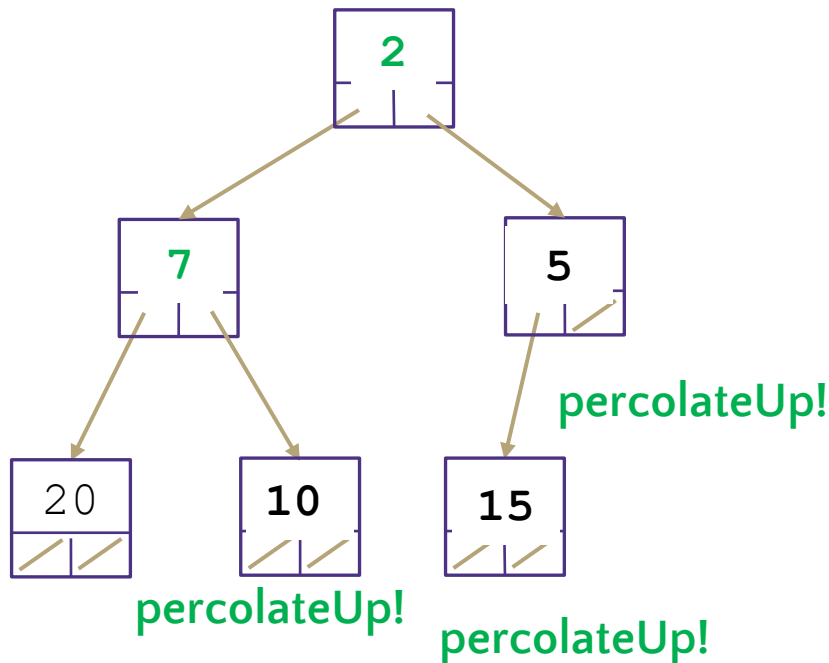


Worst case runtime is similar to `removeMin` and `percolateDown` – might have to do $\log(n)$ swaps, so the worst-case runtime is $O(\log(n))$

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 ensure 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.

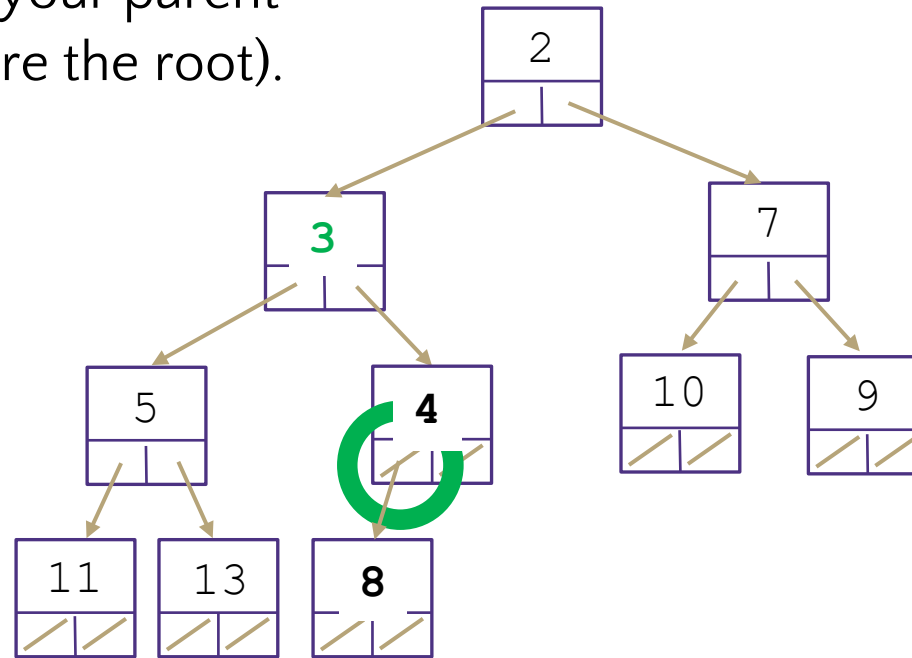
You can do it in $O(\log n)$ time on complete trees, with some extra class variants

Implementing add()

add() Algorithm:

1. Insert a node on the bottom level that ensure 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).

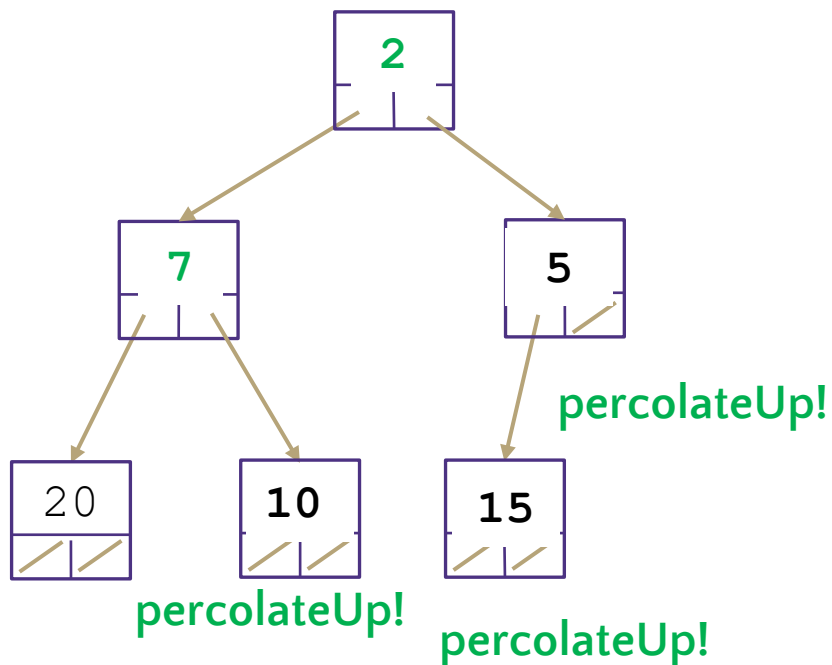


Worst case runtime is similar to `removeMin` and `percolateDown` – might have to do $\log(n)$ swaps, so the worst-case runtime is $O(\log(n))$

Quiz: 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 ensure 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():

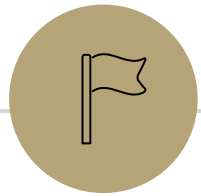
- remove root node
- find last node in tree and swap to top level
- percolate down to fix heap invariant

add()

- insert new node into next available spot
- percolate up to fix heap invariant

Finding the last node/next available spot takes $O(\log n)$ time on complete trees

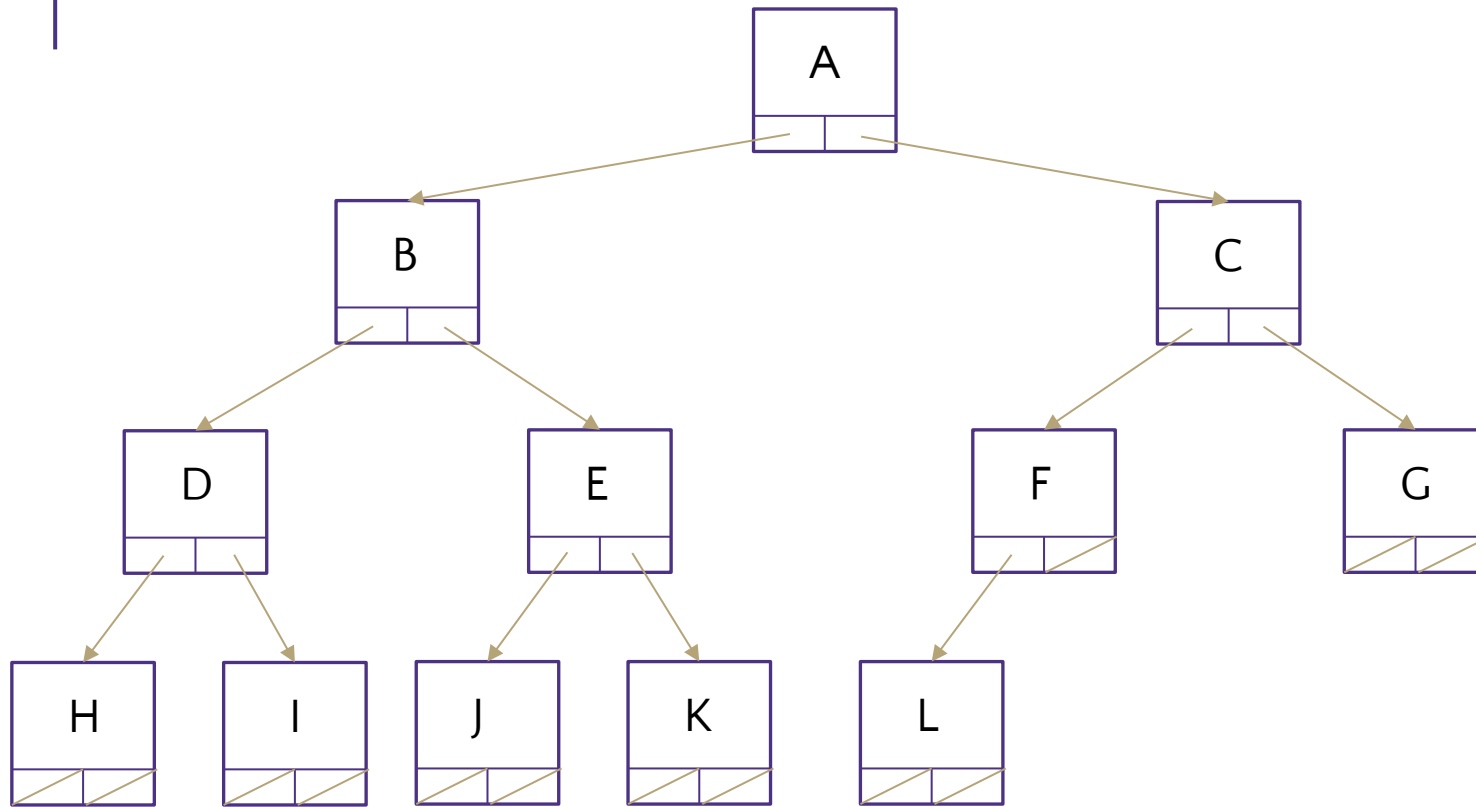
But there's a better way



Heap Array Implementation

More Priority Queue Operations

Implement Heaps with an array



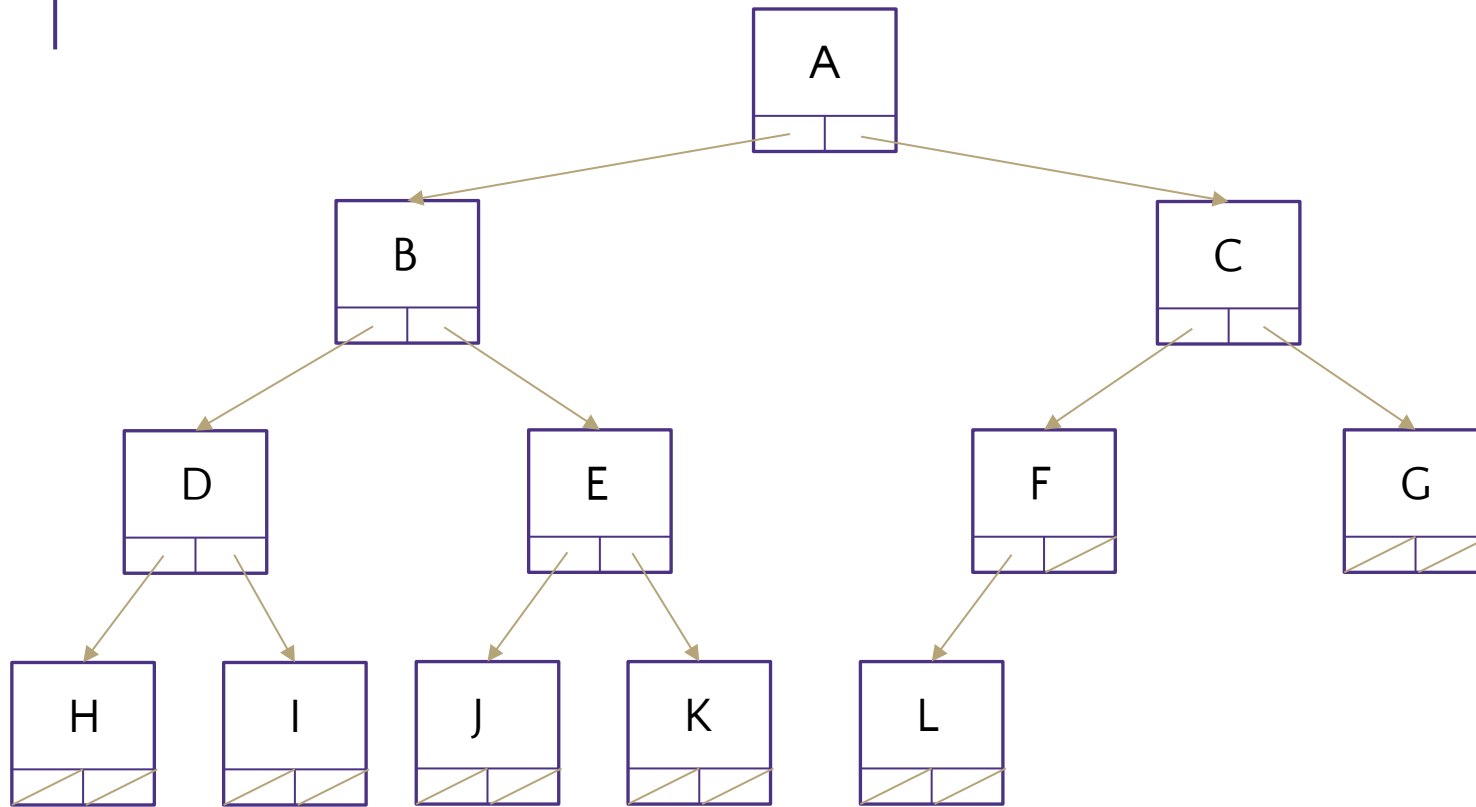
Fill array in **level-order** from left to right

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 implementation of a heap is an array, but the tree drawing is how to think of it conceptually.

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?

$$\text{peekMin}() = \text{arr}[0]$$

How do we find the last node?

$$\text{lastNode}() = \text{arr}[\text{size} - 1]$$

How do we find the next open space?

$$\text{openSpace}() = \text{arr}[\text{size}]$$

How do we find a node's left child?

$$\text{leftChild}(i) = 2i + 1$$

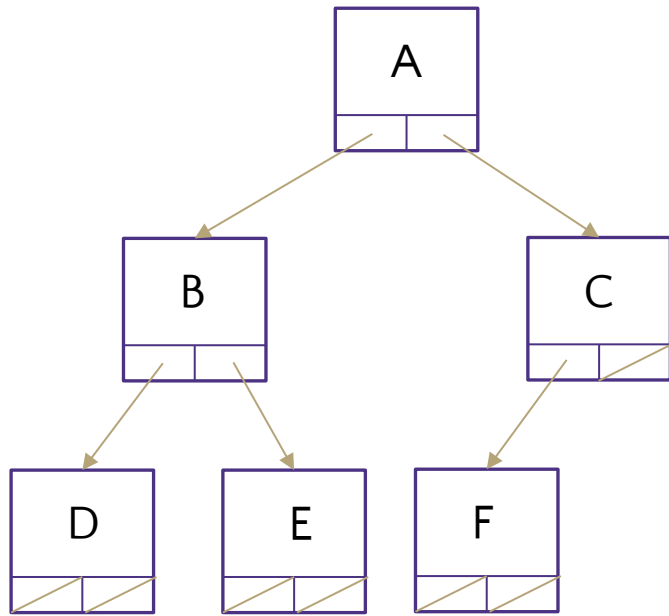
How do we find a node's right child?

$$\text{rightChild}(i) = 2i + 2$$

How do we find a node's parent?

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

Heap Implementation Runtimes

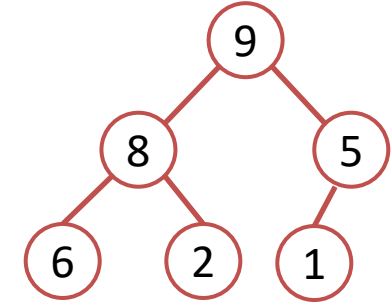


Implementation	add	removeMin	Peek
Array-based heap	worst-case: $O(\log n)$	Worst-case: $O(\log n)$	$O(1)$

0	1	2	3	4	5	6	7
A	B	C	D	E	F		

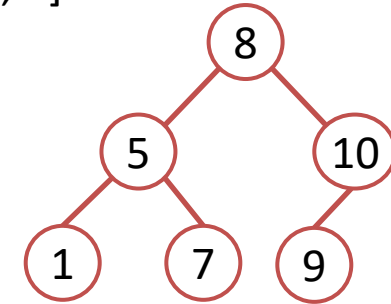
Binary Heap vs. Binary Search Tree

- Binary Heap: the min-heap property
 - Value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.
 - A heap is not a sorted data structure and can be regarded as partially ordered.
- BST: Ordered, or sorted, binary trees
 - Items to the left of a given node are smaller.
 - Items to the right of a given node are larger.
- Both structures offer $O(\log n)$ time complexity for certain operations, they are used in different scenarios.
 - Heapsort is used for efficient sorting and simple priority queue implementations
 - BST can also be used for sorting, by insertions followed by in-order traversal, with $O(n \log(n))$ average-case complexity



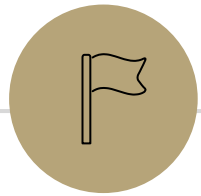
Binary Heap

In-order traversal does not give sorted list [6, 8, 2, 9, 5, 1]



Binary Search Tree

In-order traversal gives sorted list [1, 5, 7, 8, 9, 10]



Heap Array Implementation

More Priority Queue Operations

BuildHeap

BuildHeap(elements e_1, \dots, e_n)

Given n elements, create a heap containing exactly those n elements.

Try 1: Just call insert n times.

- n calls, each with worst-case complexity $O(\log n)$, so overall worst-case complexity is $O(n \log n)$
- Worst-case input: if we insert elements in decreasing order, every node will have to percolate all the way up to the root.
- Can we do better?

Can We Do Better?

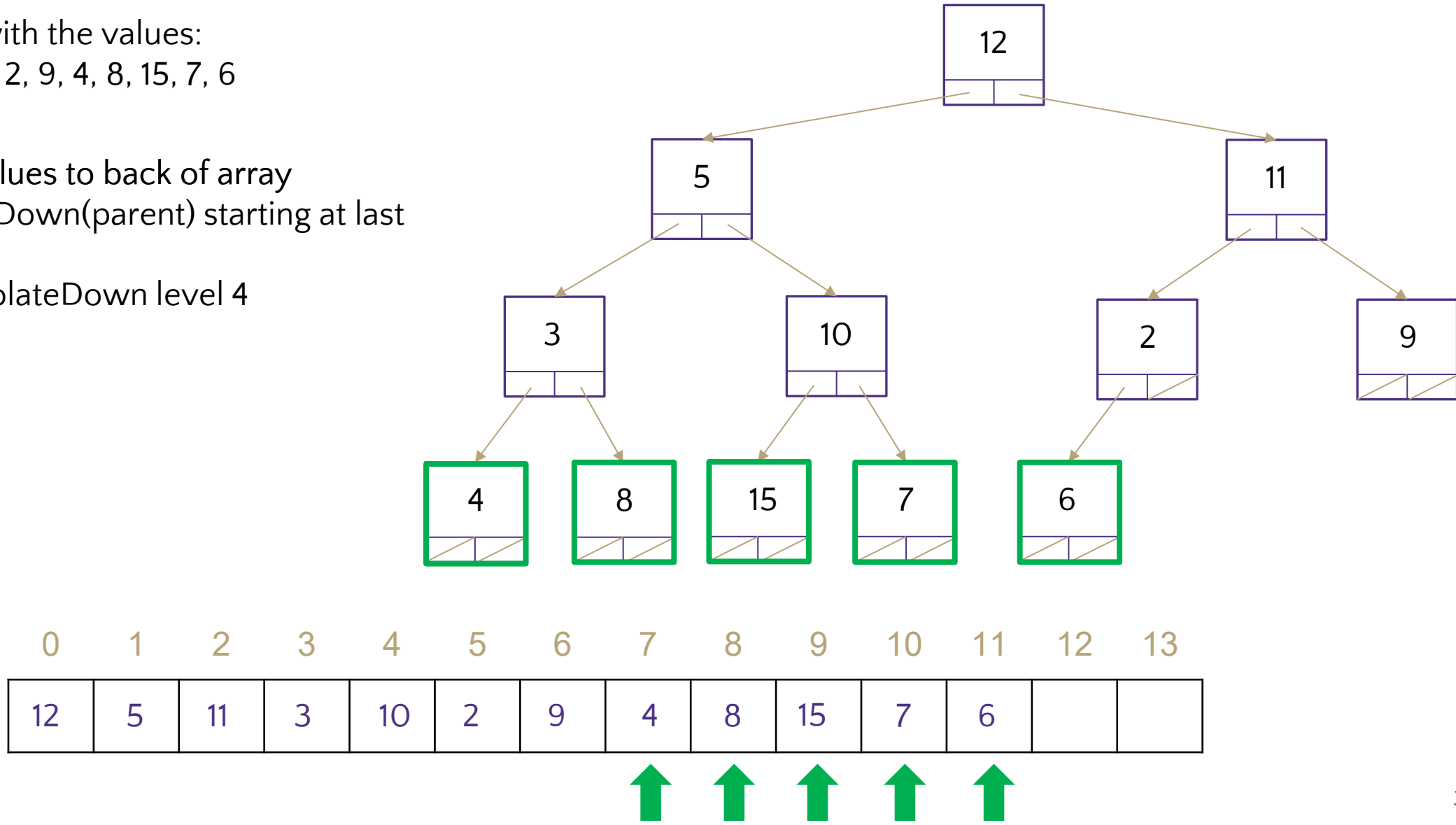
- What's causing the n add strategy to take so long?
 - Most nodes are near the bottom, and might need to percolate **all the way up**
- Idea 2: Dump everything in the array, and percolate things **down** until the heap invariant is satisfied
 - The bottom two levels of the tree have $O(n)$ nodes, the top two have 3 nodes
 - Maybe we can make “most of the nodes” at the bottom 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
 1. percolateDown level 4

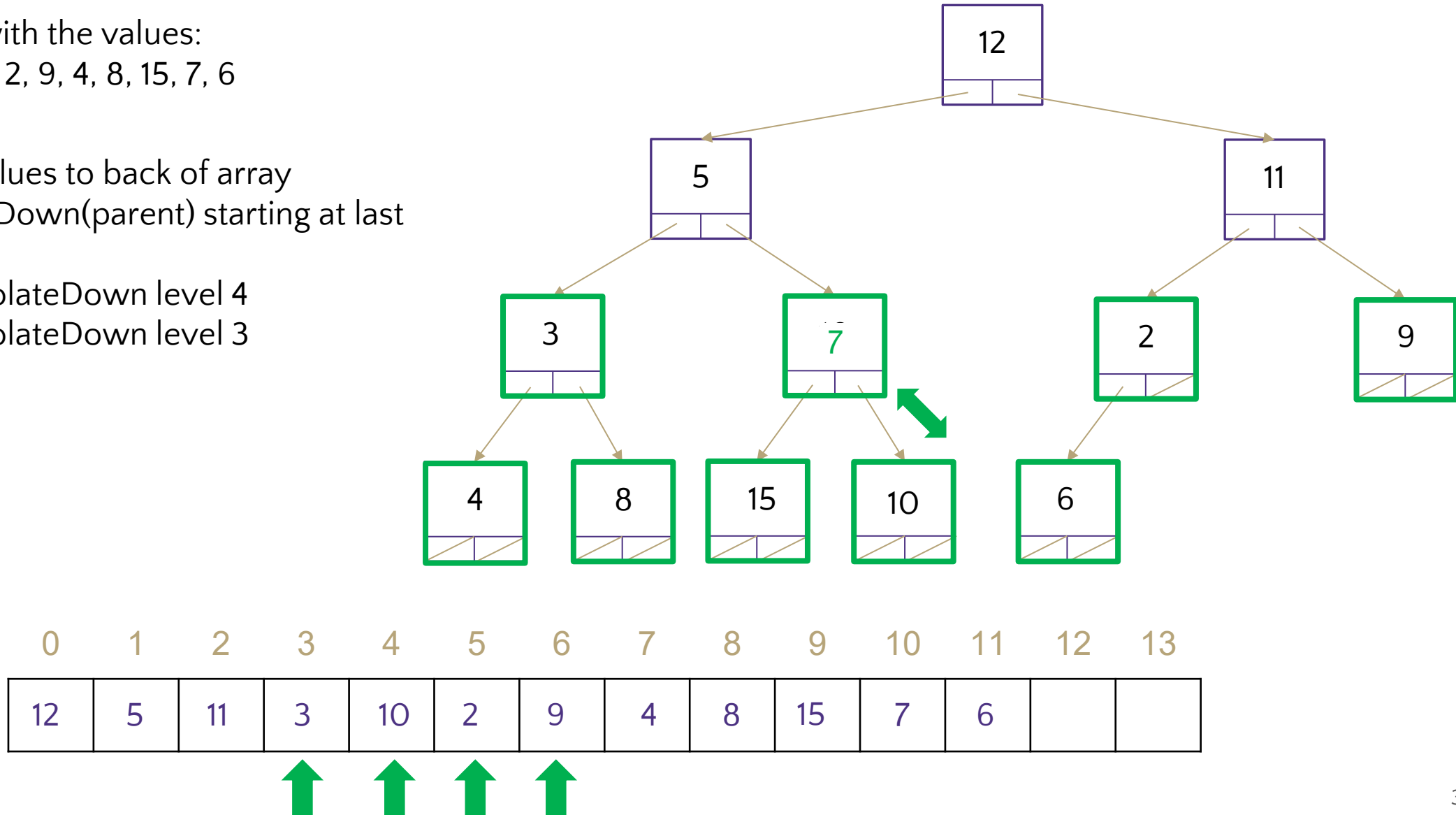


Floyd's buildHeap algorithm

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

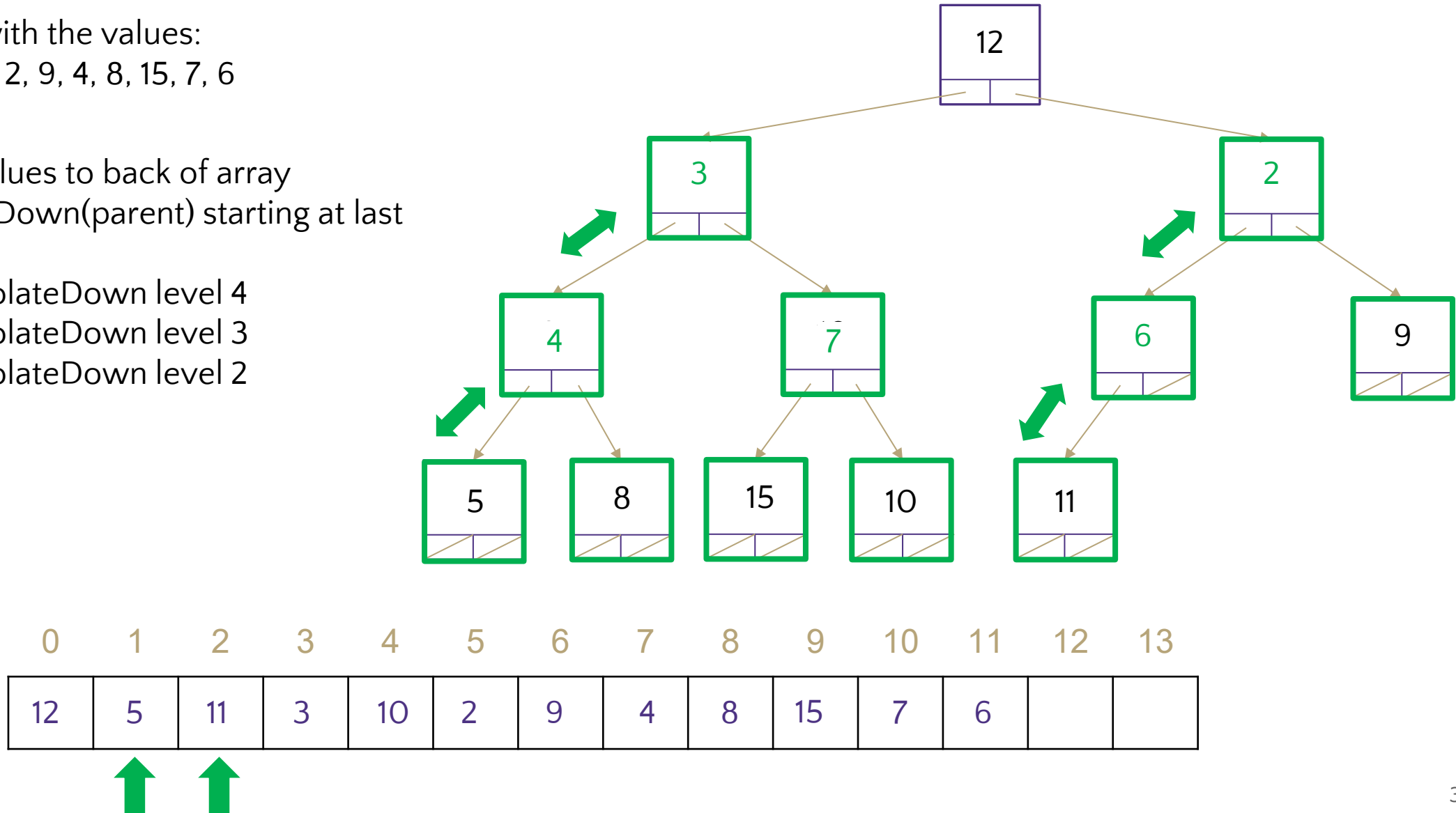


Floyd's buildHeap algorithm

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

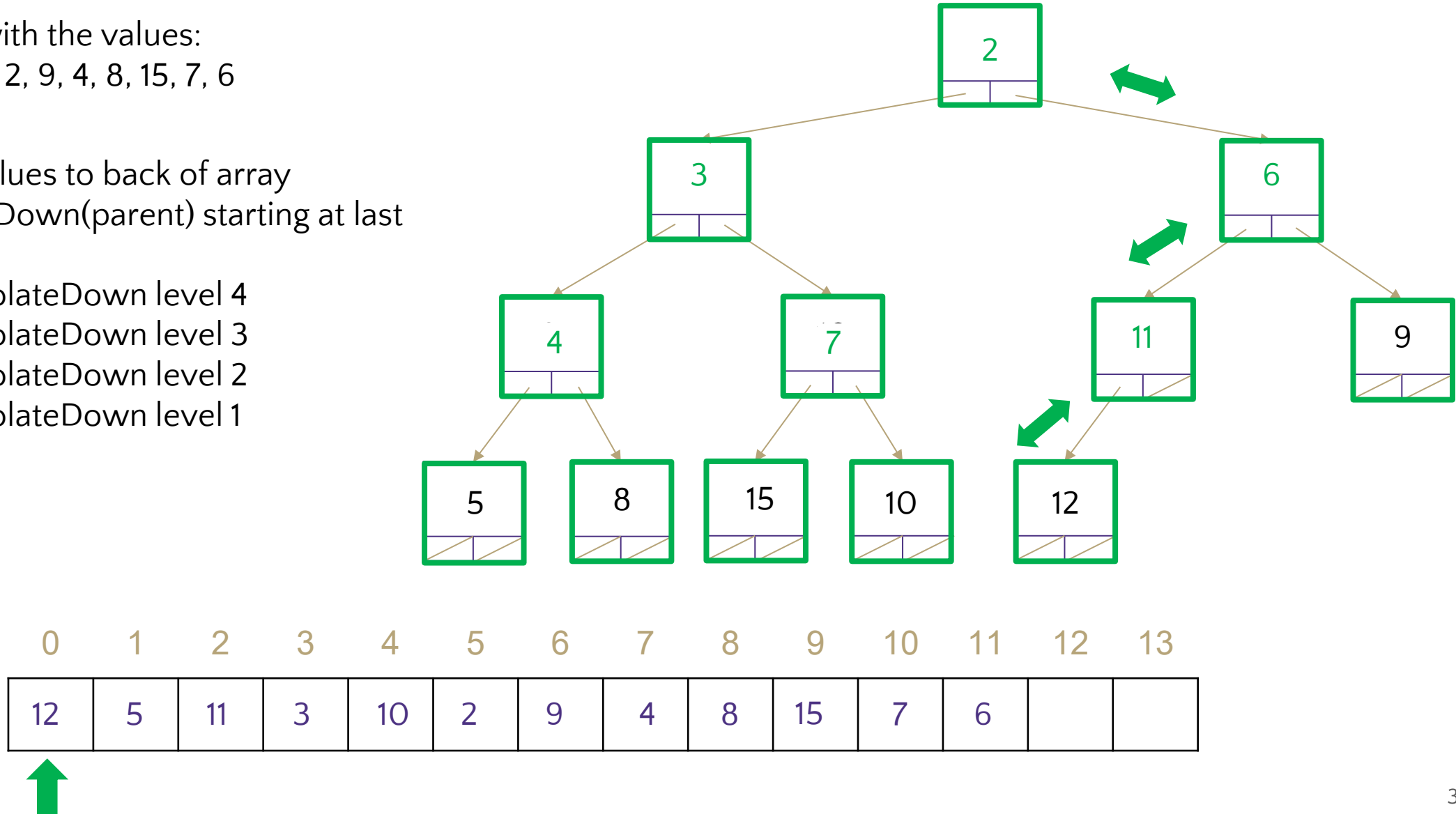


Floyd's buildHeap algorithm

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



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 $1/2^i$) we converge to a constant factor of n .

Optional Slide Floyd's buildHeap Summation

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

factor out n

$$\text{work}(n) \approx n \left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \dots + \frac{\log n}{n} \right) \text{ find a pattern } \rightarrow \text{powers of 2} \quad \text{work}(n) \approx n \left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \dots + \frac{\log n}{2^{\log n}} \right) \text{ Summation!}$$

$$\text{work}(n) \approx n \sum_{i=1}^? \frac{i}{2^i} \quad ? = \text{upper limit should give last term}$$

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

Infinite geometric series

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

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

References

- What Is a Binary Heap? By Spanning Tree
 - <https://www.youtube.com/watch?v=AE5IOxACpZs>
- Can we represent a tree with an array? – Inside code
 - <https://www.youtube.com/watch?v=EitnYxinKkw>
- Min Heap Animations | Data Structure | Visual How
 - <https://www.youtube.com/watch?v=AFPzC2RJOMk>