Lecture 13: Heaps

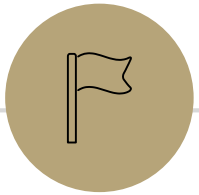
CSE 373 Data Structures and Algorithms

Administrivia

We'll post midterm review material tonight.

In the meantime, go to the bottom of spring's exam page.

https://courses.cs.washington.edu/courses/cse373/19sp/exams/



Heaps

Priority Queue ADT

Min Priority Queue ADT

state

Set of comparable values
- Ordered based on "priority"

behavior

insert(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 writing a patient management system for an ER. You need to make sure when a doctor becomes available the person who most urgently needs help is seen first.

Other uses:

- Operating System
- Well-designed printers
- Huffman Codes (in 143)
- Sorting (in Project 2)
- 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

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

insert(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

insert(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 insert and removeMin take with these data structures?

Implementation	Insert	removeMin	Peek
Unsorted Array			
Sorted Array (use "circular 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 insert and removeMin take with these data structures?

Implementation	Insert	removeMin	Peek
Unsorted Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sorted Array (use "circular array")	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
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 insert and removeMin take with these data structures?

Implementation	Insert	removeMin	Peek
Unsorted Array	$\Theta(1)$	$\Theta(n)$	~~$\Theta(n)$~~ $\Theta(1)$
Sorted Array (use "circular array")	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Linked List (sorted)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	~~$\Theta(\log n)$~~ $\Theta(1)$

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

Let's start with an AVL tree

AVLPriorityQueue<E>

state

`overallRoot`

behavior

`removeMin()` – traverse through tree all the way to the left, remove node, rebalance if necessary

`peekMin()` – traverse through tree all the way to the left

`insert()` – traverse through tree, insert node in open space, rebalance as necessary

There's a technical issue:

Priority Queues allow for repeated priorities, AVL trees don't

"easy" to fix

- Can add a custom `compareTo` that arbitrarily breaks ties.
- Or just weaken the BST invariant to allow for ties.

Implementing heaps with AVL trees isn't a crazy idea.

We're going to introduce another one, but keep this baseline in your mind.

Whatever we come up with, it has to be better than this.

Heaps

Idea:

In a BST, we organized the data to find anything quickly.

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.

The AVL condition was a bit complicated to maintain.

- Because we had to make sure when we inserted we could maintain the exact BST structure

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.

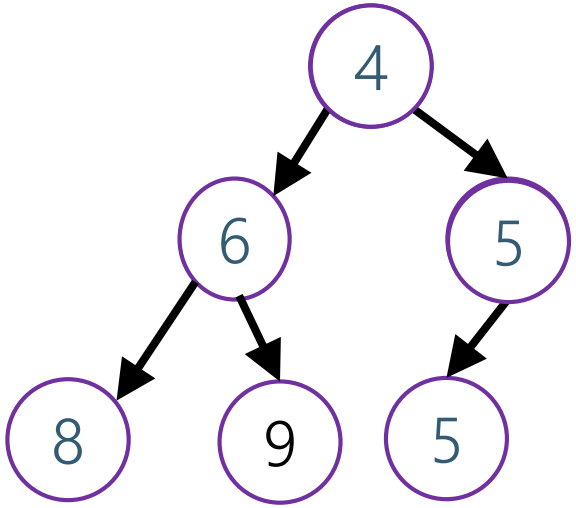
A tree is complete if:

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

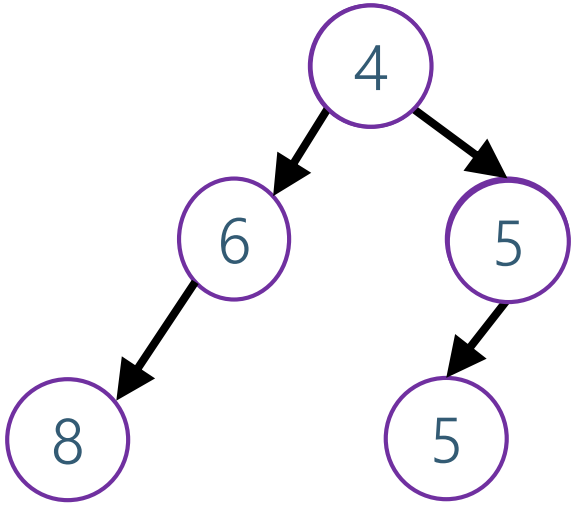
Tree Words

Complete – every row is completely filled, except possibly the last row, which is filled from left to right.

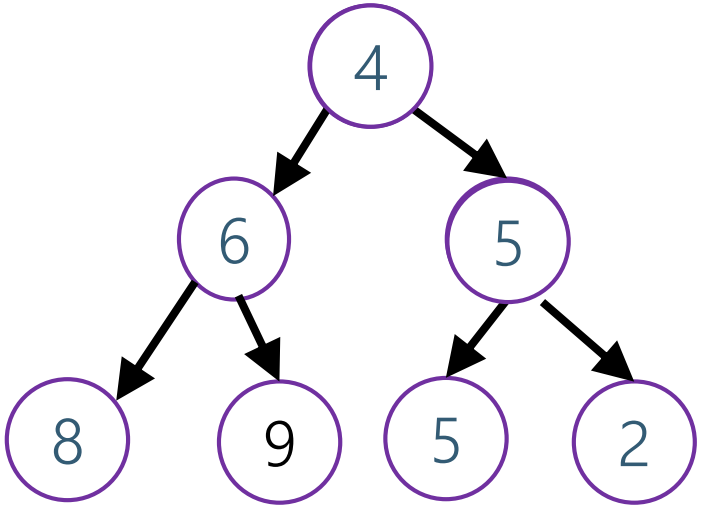
Perfect – every row is completely filled



Complete, but not perfect



Neither



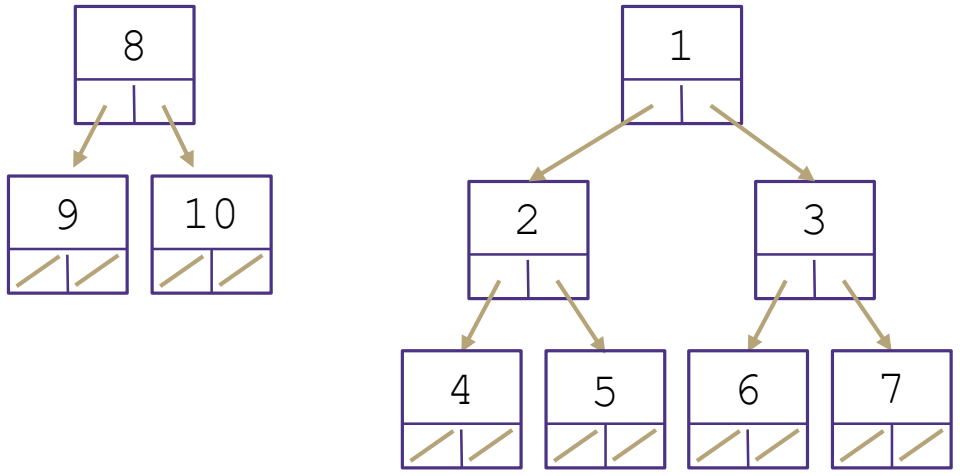
Both Perfect and Complete

Binary Heap

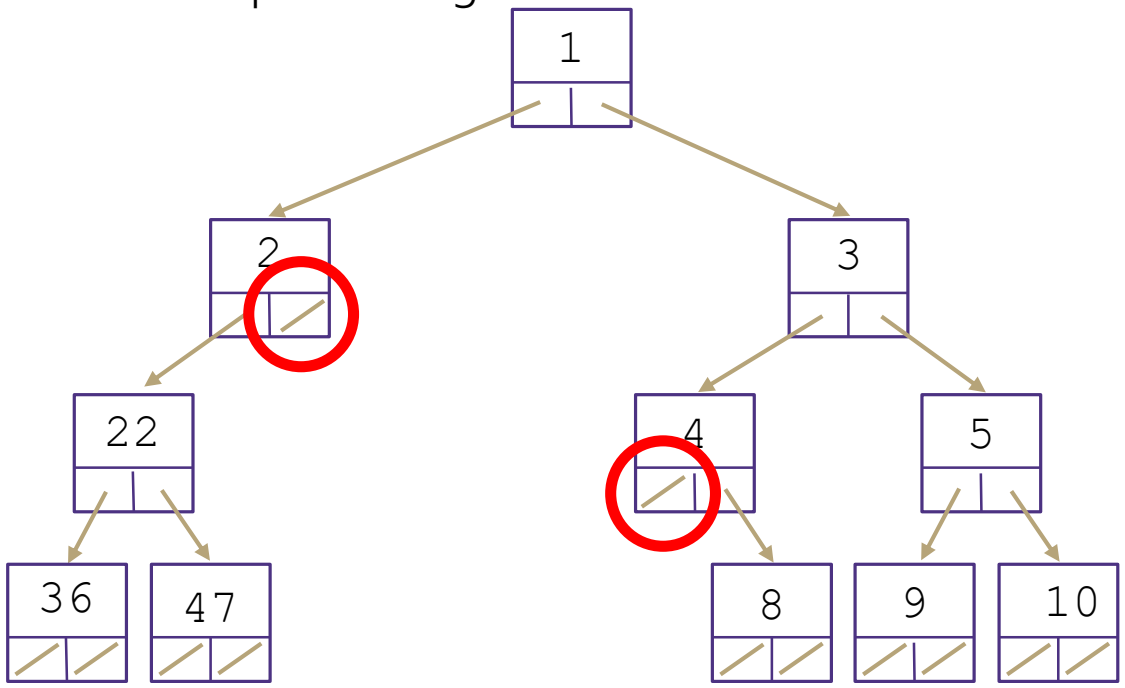
One flavor of heap is a **binary** heap.

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

2. **Heap**: every node is smaller than its child



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

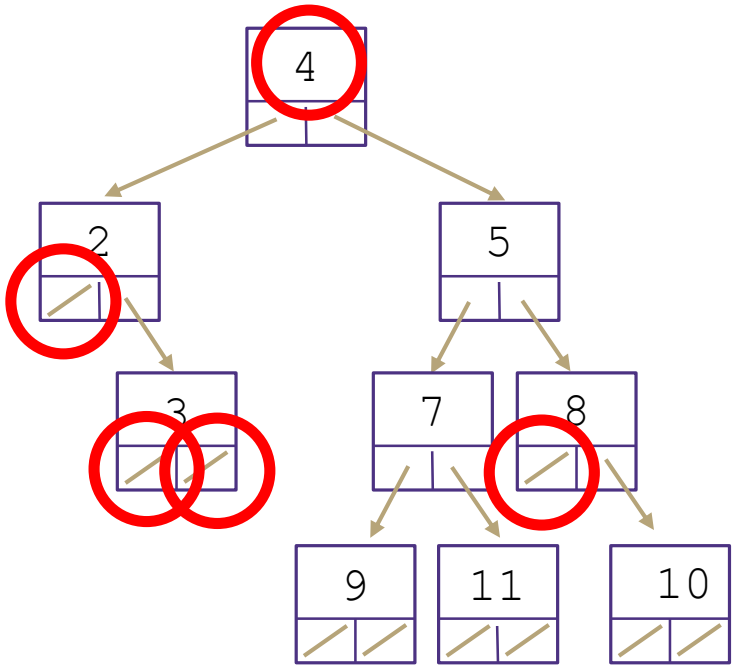


Self Check - Are these valid heaps?

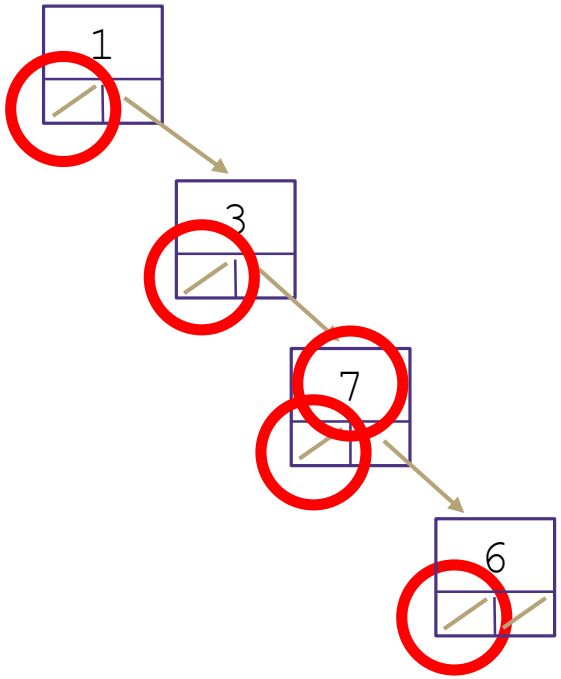
3 Minutes

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

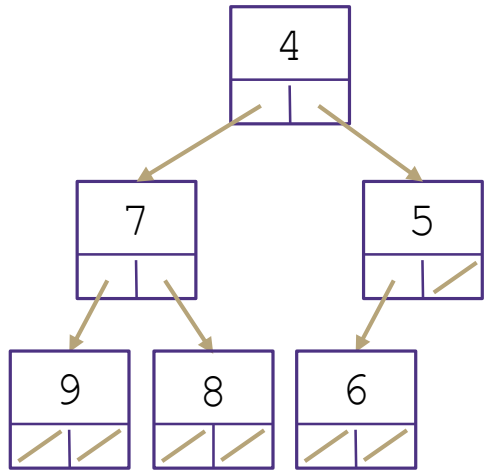
INVALID



INVALID

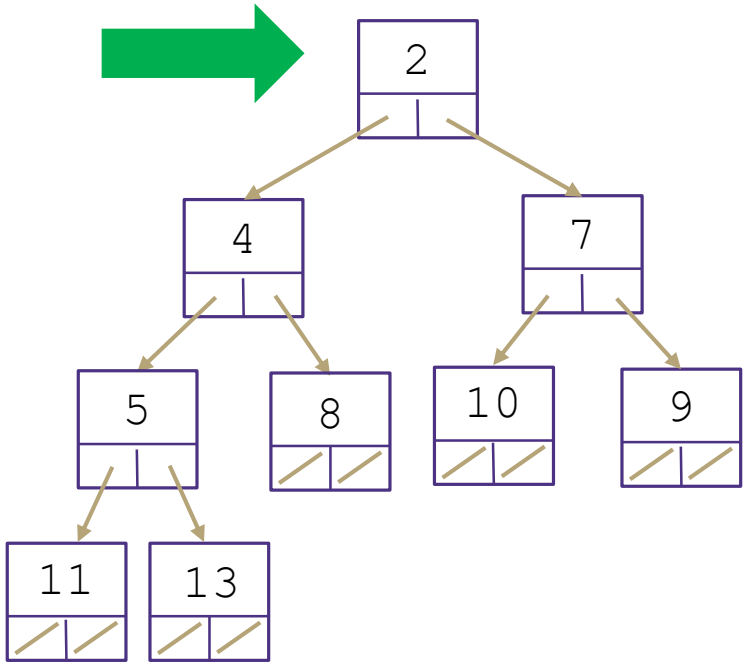


VALID



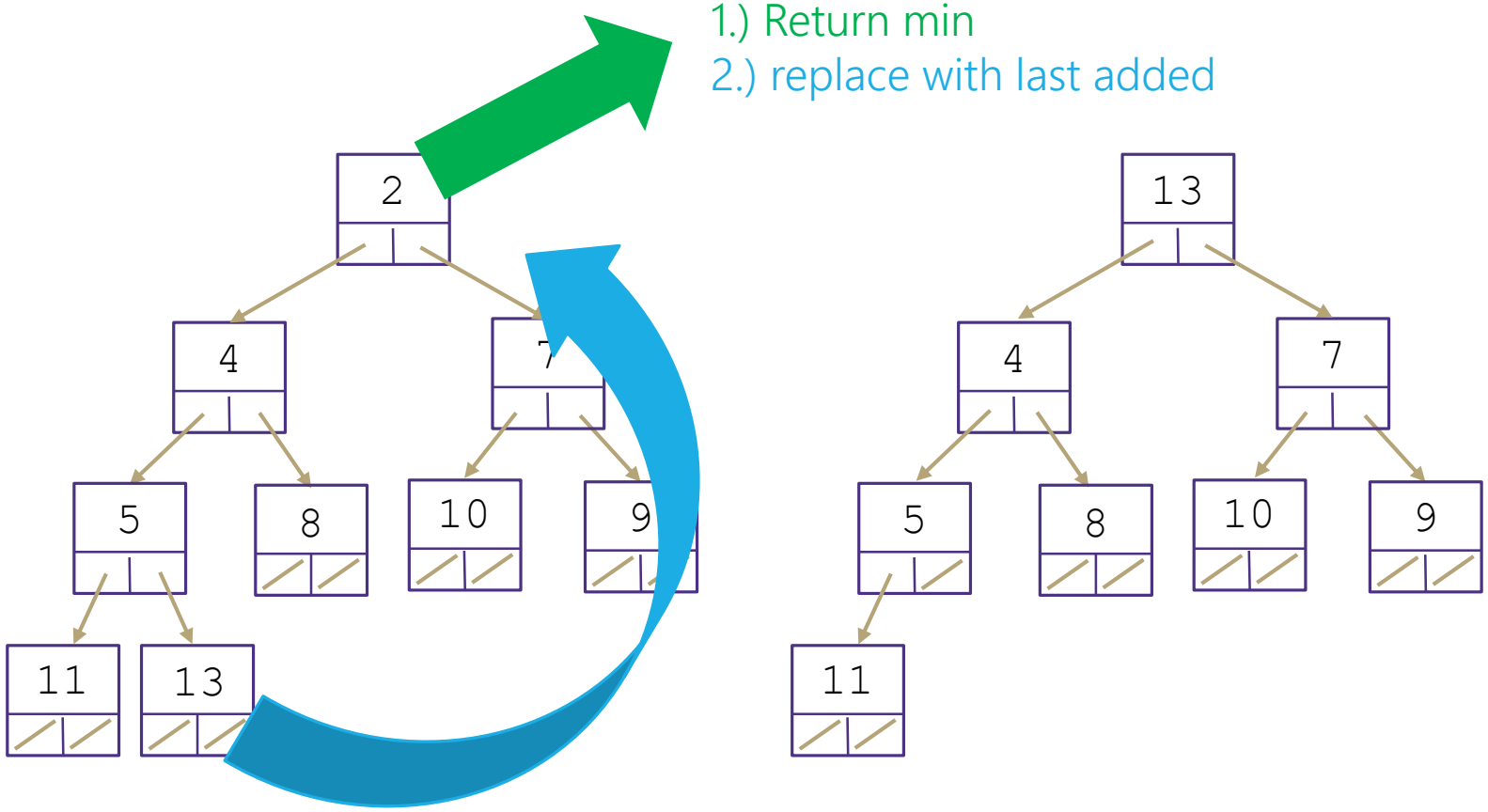
Implementing peekMin()

Runtime: $\Theta(1)$



Implementing removeMin()

1.) Return min
2.) replace with last added

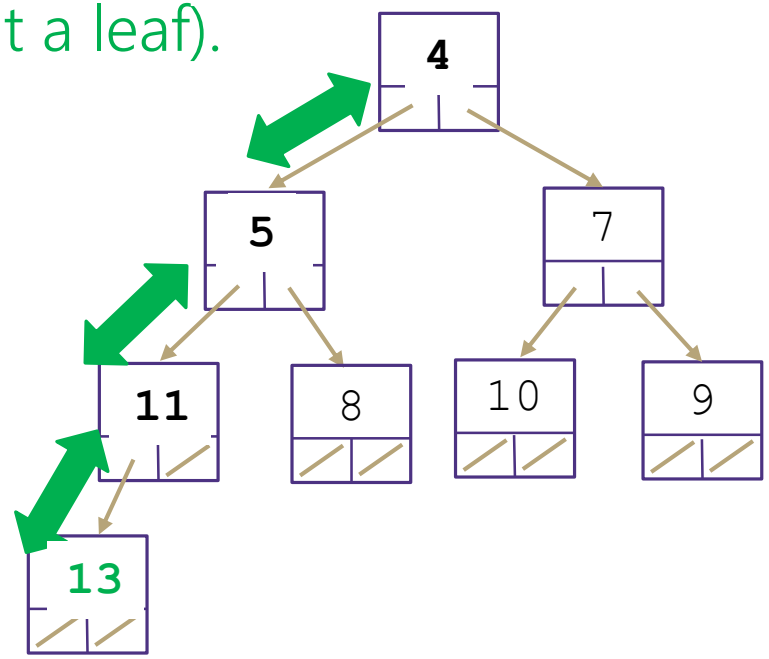


Structure invariant restored, heap invariant broken

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 running time?

Have to:

Find last element

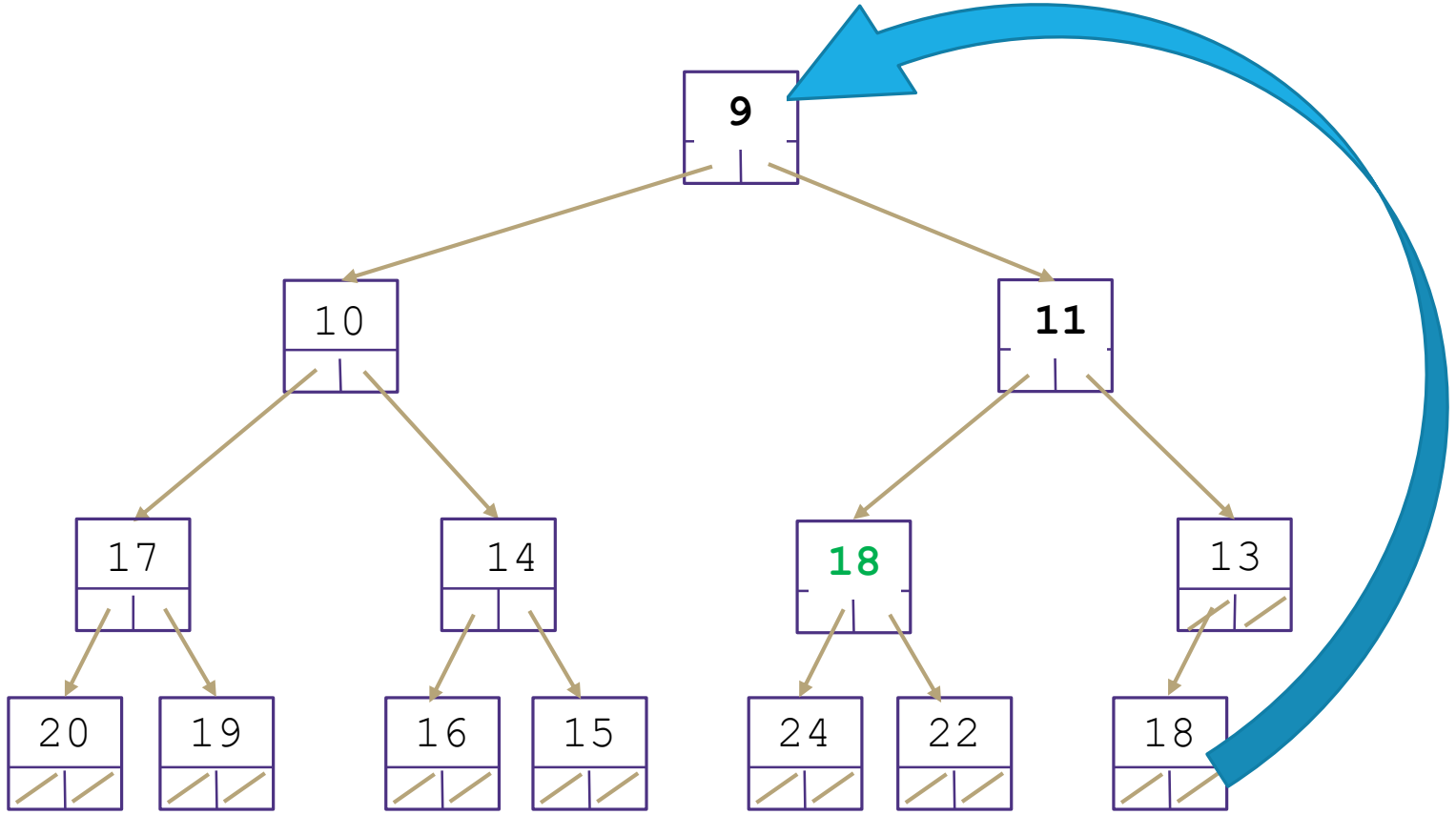
Move it to top spot

Swap until invariant restored

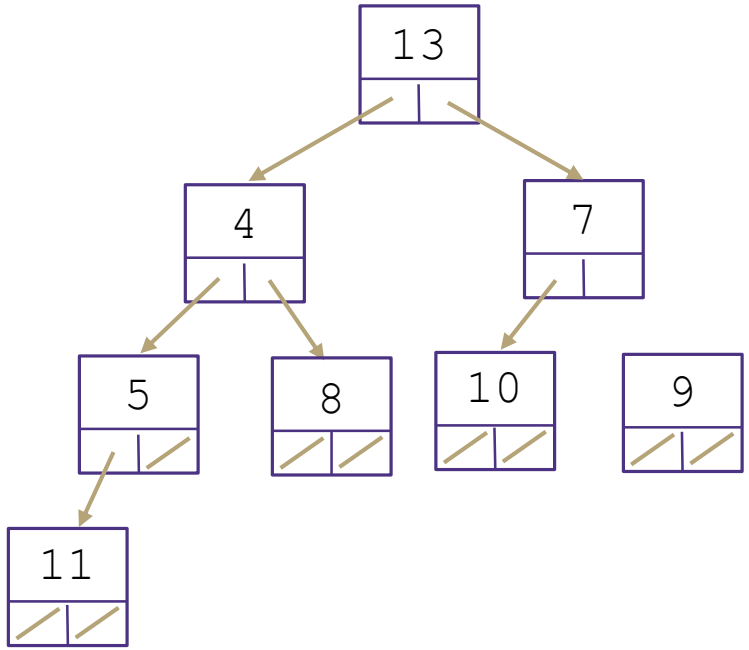
Structure invariant restored, heap invariant restored

3 Minutes

Practice: removeMin()



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 insert()

Algorithm:

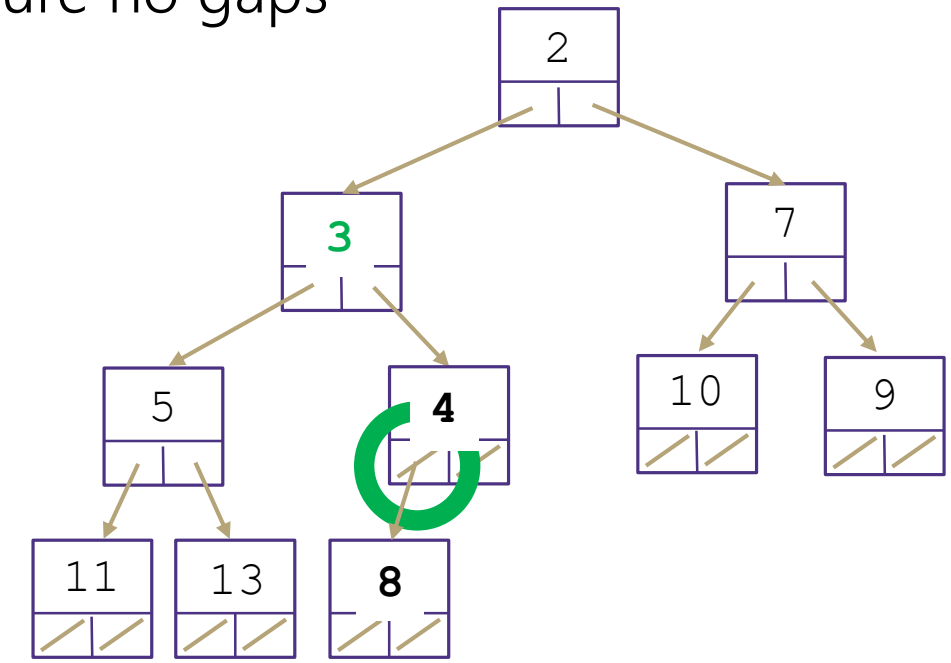
- Insert a node to ensure no gaps
- Fix heap invariant
- percolate **UP**

i.e. swap with parent,

until your parent is

smaller than you

(or you're the root).



3 Minutes

Practice: Building a minHeap

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

5, 10, 15, 20, 7, 2

Min Priority Queue ADT

state

Set of comparable values
- Ordered based on "priority"

behavior

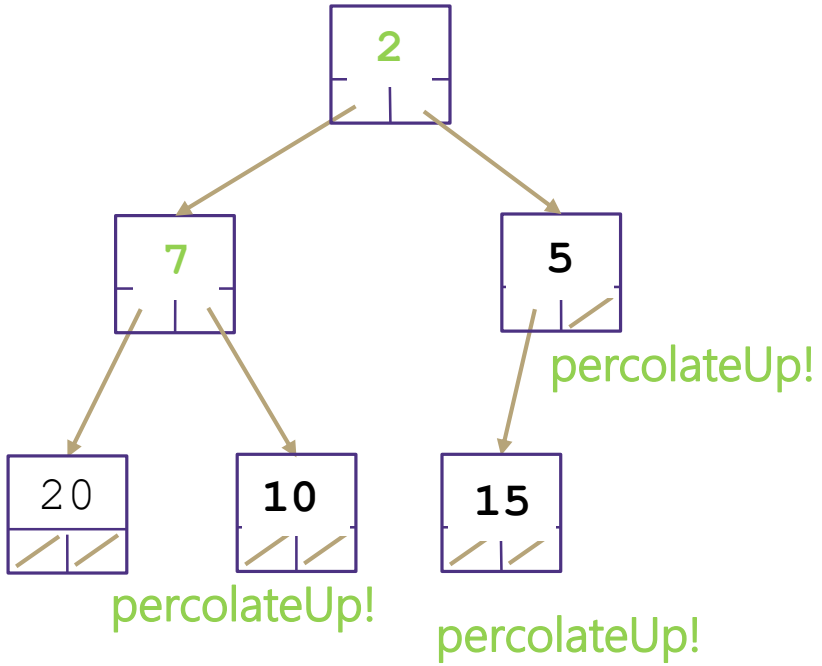
removeMin() – returns the element with the smallest priority, removes it from the collection

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

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

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

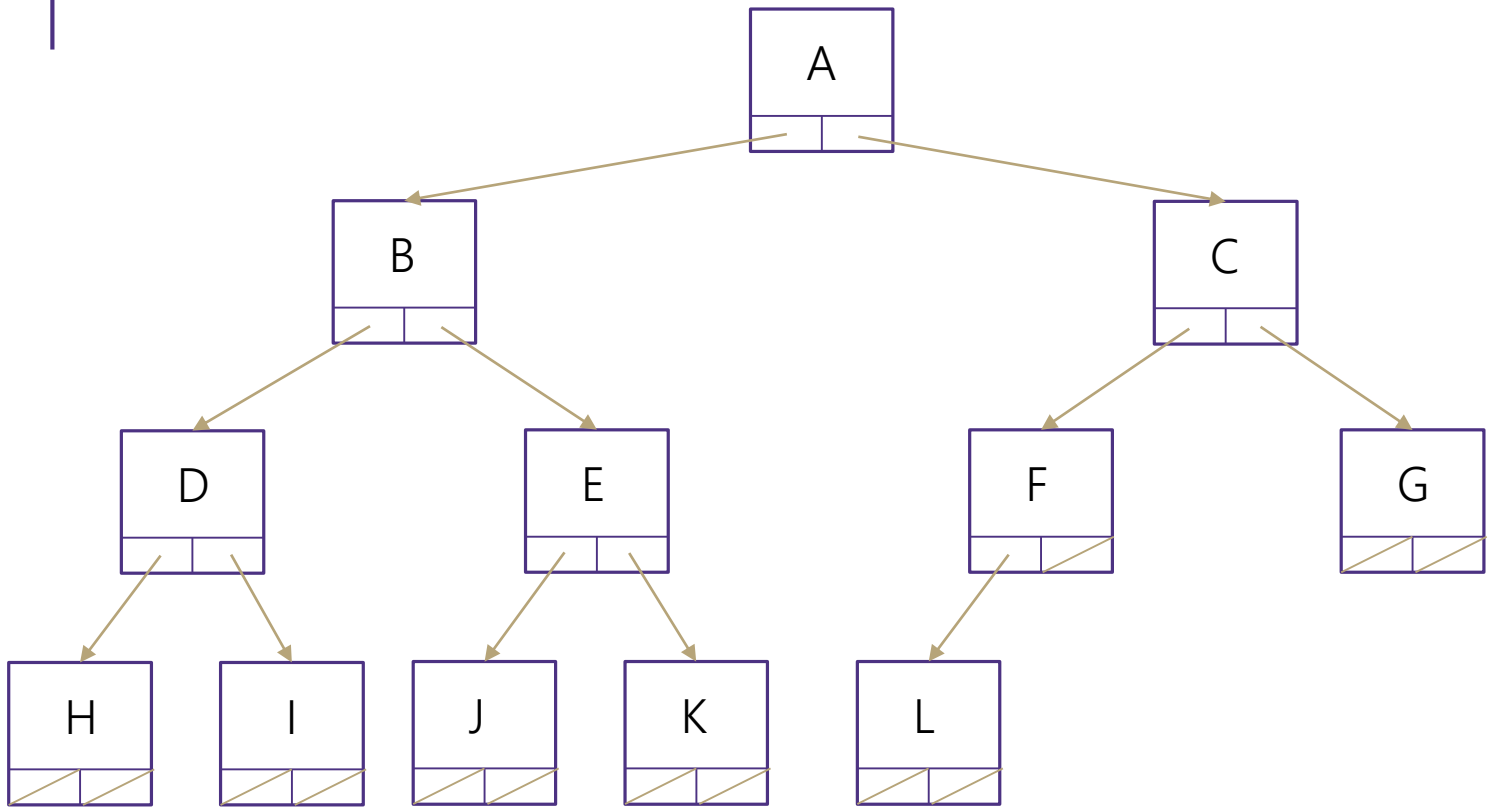
insert():

- 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!

Implementing Heaps



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?

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

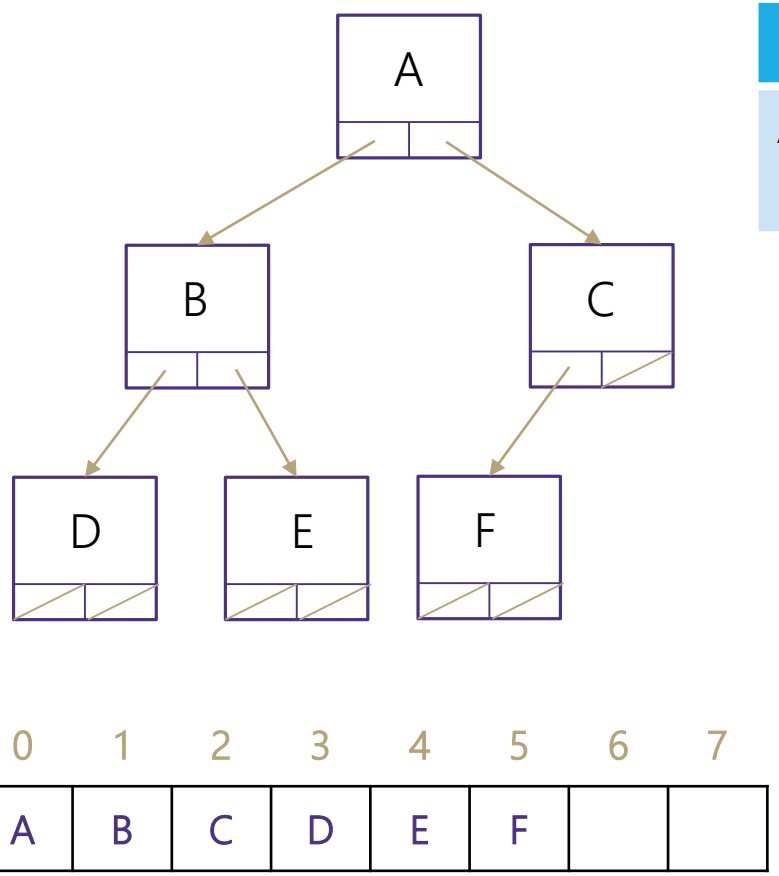
How do we find a node's right child?

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

How do we find a node's parent?

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

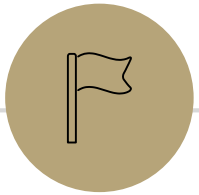
Heap Implementation Worst-Case Runtimes



Implementation	Insert	removeMin	Peek
Array-based heap	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$

We've matched the **asymptotic** 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.
A heap is MUCH simpler to implement.



More Priority Queue Operations

More Operations

Min Priority Queue ADT

state

Set of comparable values
- Ordered based on "priority"

behavior

insert(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**?

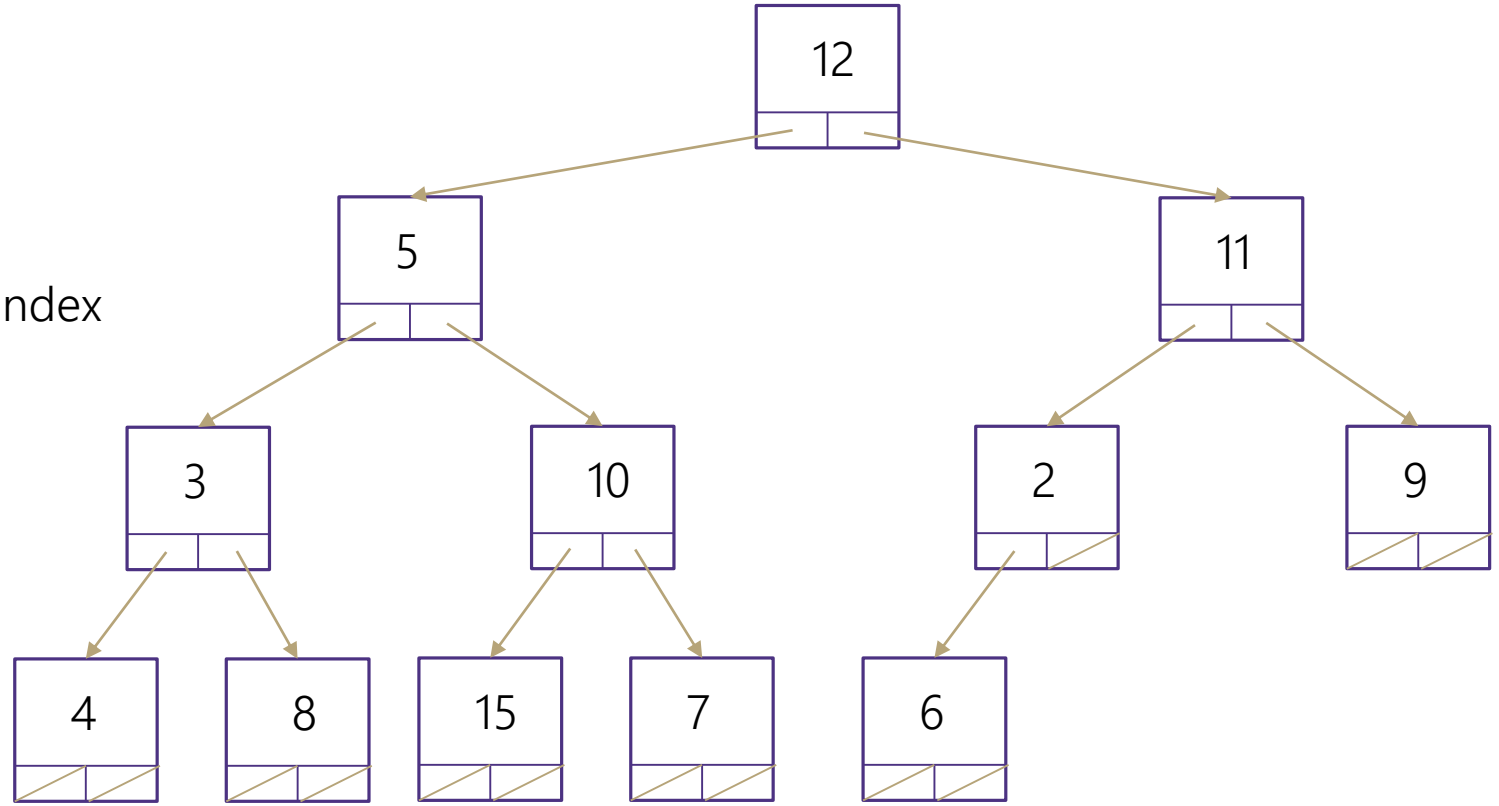
It's not clear what order to call the `percolateDown`'s in.

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

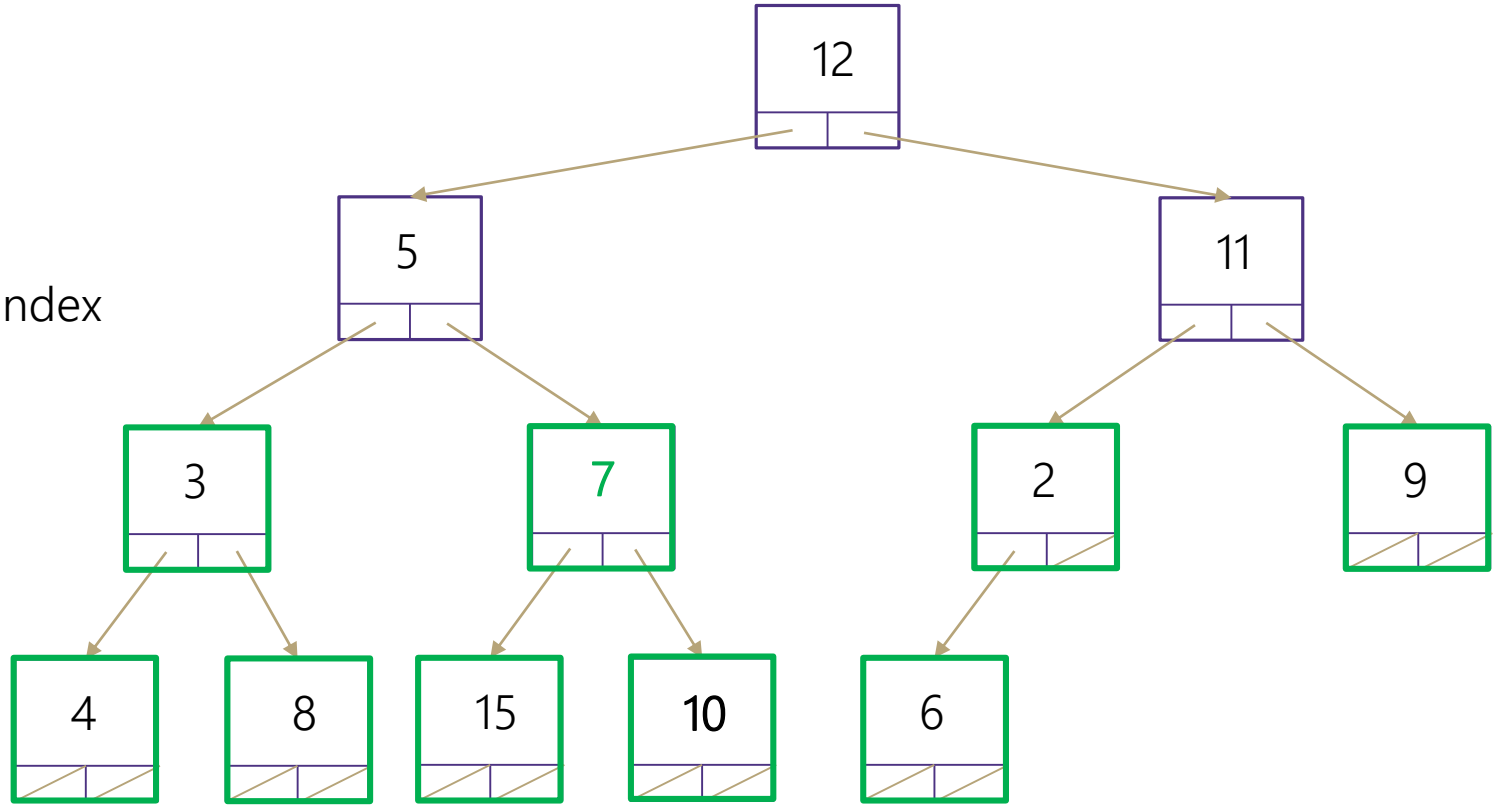


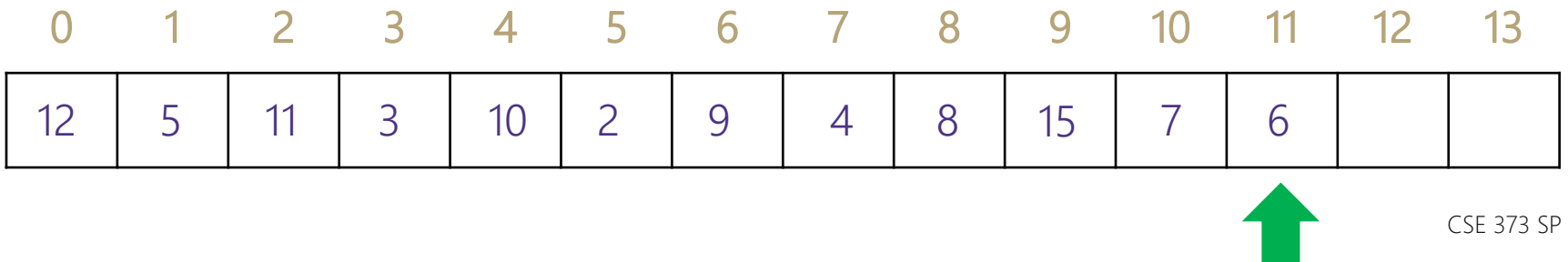
0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	15	7	6		

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



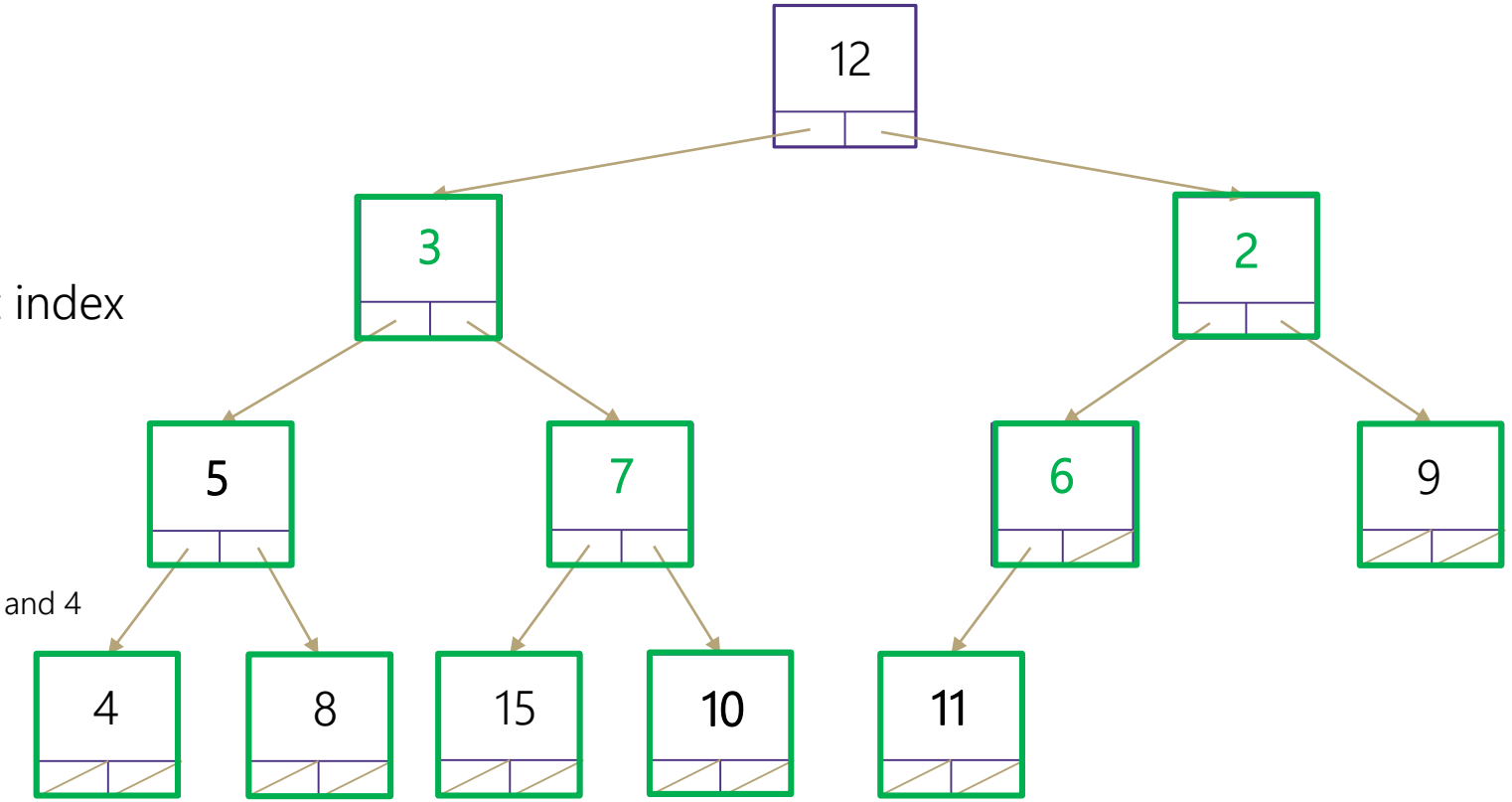


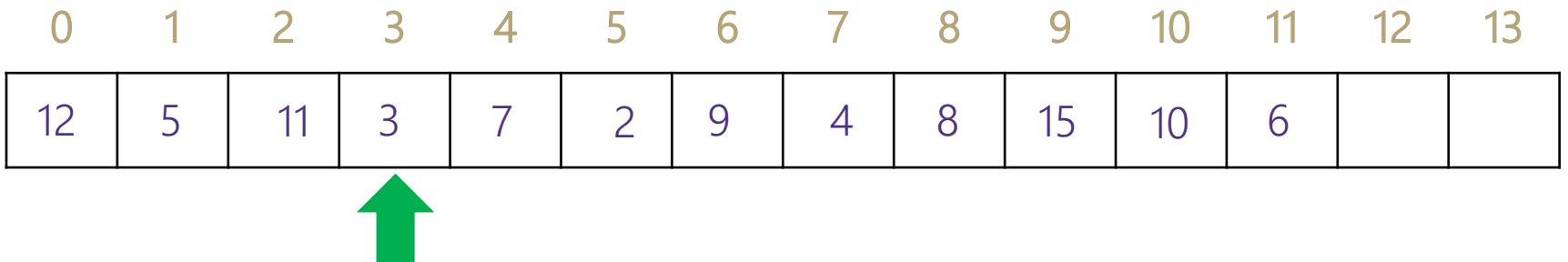
Floyd's buildHeap algorithm

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

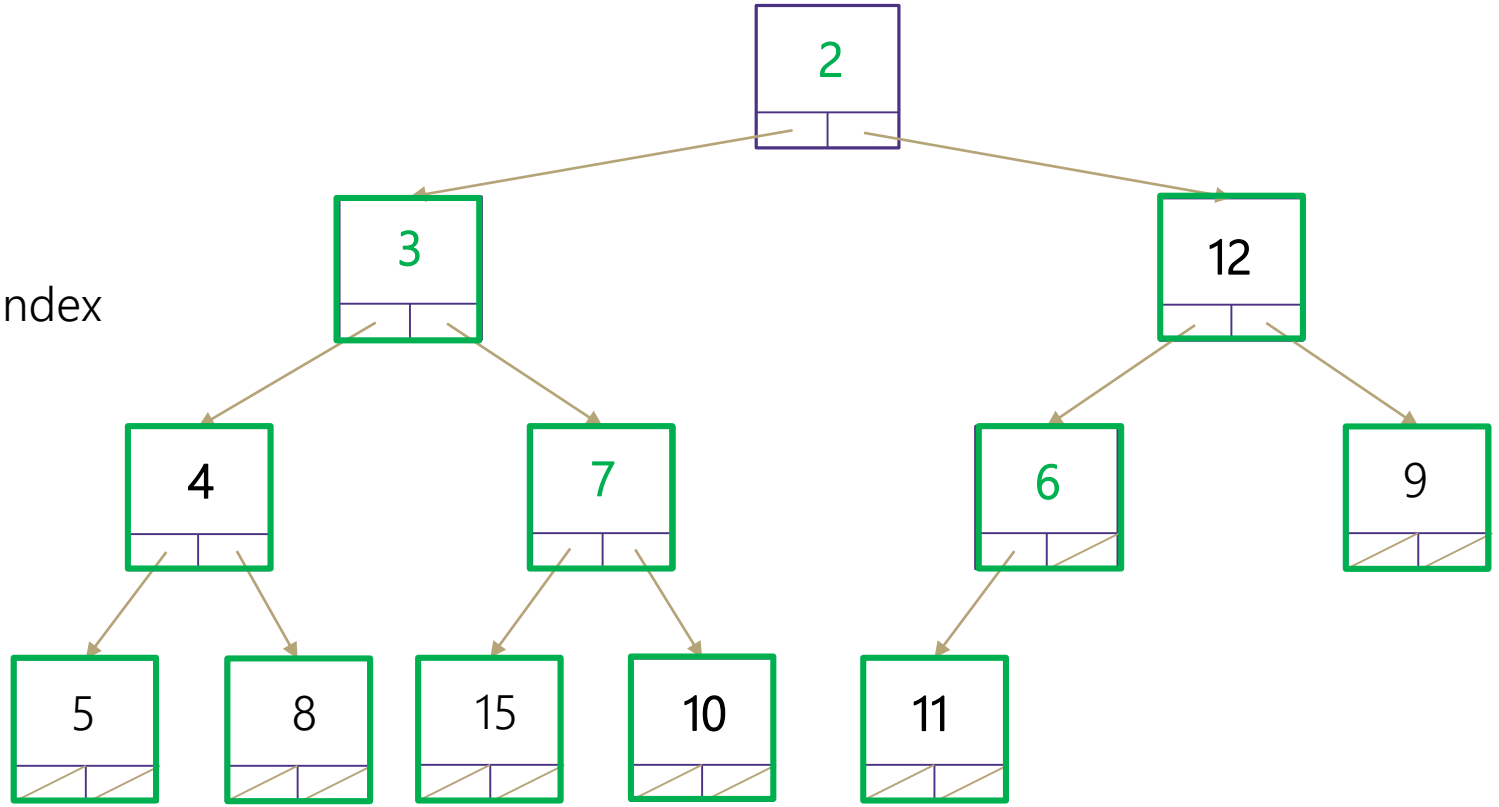




Floyd's buildHeap algorithm

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

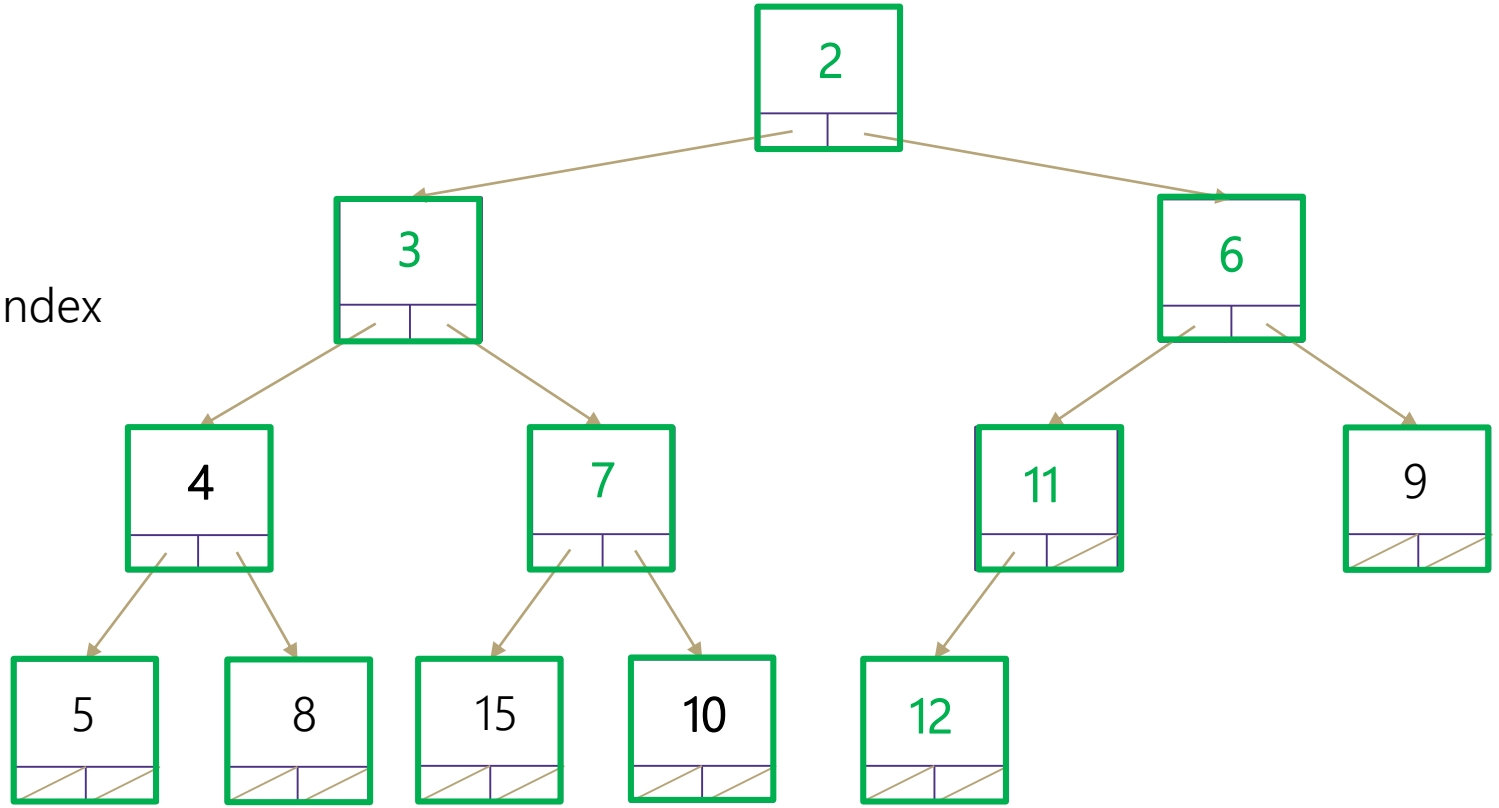


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	3	2	4	7	6	9	5	8	15	10	11		

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.