Priority Queues and Heaps

Data Structures and Algorithms

Warm Up

We have seen several data structures that can implement the Dictionary ADT so far: Arrays, Binary (AVL) Trees, and Hash Tables.

1. Discuss with your neighbors the relative merits of each approach?
2. Is there any reason to choose an AVL Tree Dictionary over a Hash Table?

BST vs Hash Table

BST ADVANTAGES

Extra Operations: Get keys in sorted order, ordering statistics (min-key, max-key, etc.), "closest" elements, range queries

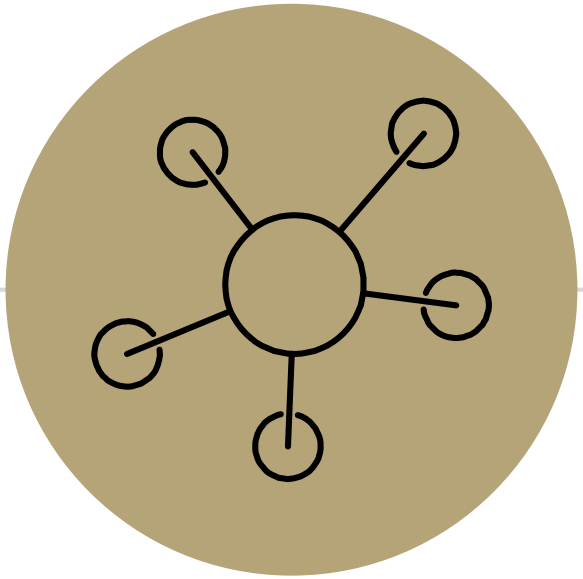
O(log n) **worst case** runtime guaranteed, better than O(n) for hash tables

Easier to implement

HASH TABLE ADVANTAGES

O(1) Average Case Operations

Array Lookups can be an order of magnitude faster than following references



How can we track the "biggest" thing?

Last class, a question was raised: couldn't we guarantee O(1) operations if we kept track of the largest cluster in a hash table?

ADT: Priority Queue

A **Priority Queue** models a collection that is *not* First-In-First-Out (FIFO), but instead **most-important-**in-first-out.

Example: Hospital emergency room. The patient who is most in danger is treated first.

Items in a priority queue are **comparable**, and the comparison determines priority. Often this is done by inserting items as a pair: (item, priority).

Operations:

insert(item, [priority]) – adds an item to the queue with a given priority

deleteMin() – removes and returns the most-important item in the priority queue

peekMin() – returns the most-important item in the priority queue *without* removal

(This is a min priority queue. You can also have a max priority queue by swapping min/max.)

Implementing Priority Queue

Idea	Description	removeMin() runtime	peekMin() runtime	insert() runtime
Unsorted ArrayList	Linear collection of values, stored in an Array, in order of insertion	O(n)	O(n)	O(1)
Unsorted LinkedList	Linear collection of values, stored in Nodes, in order of insertion	O(n)	O(n)	O(1)
Sorted ArrayList	Linear collection of values, stored in an Array, priority order maintained as items are added	O(1)	O(1)	O(n)
Sorted Linked List	Linear collection of values, stored in Nodes, priority order maintained as items are added	O(1)	O(1)	O(n)
Binary Search Tree	Hierarchical collection of values, stored in Nodes, priority order maintained as items are added	O(n)	O(n)	O(n)
AVL tree	Balanced hierarchical collection of values, stored in Nodes, priority order maintained as items are added	O(logn)	O(logn)	O(logn)

Heaps

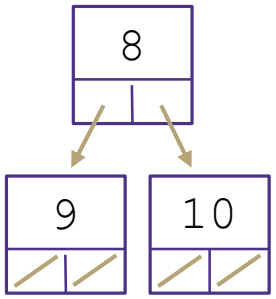
Priority Queue Data Structure

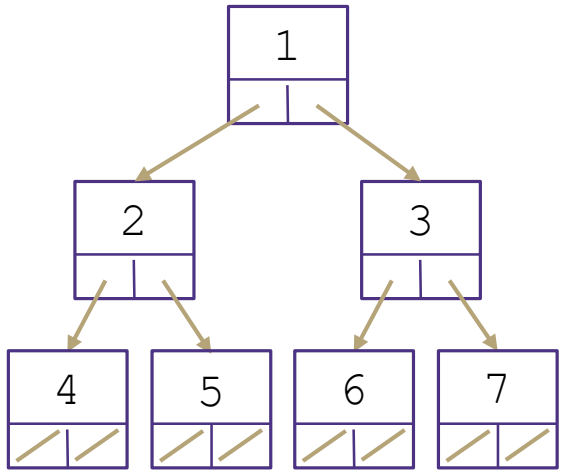
Binary Heap

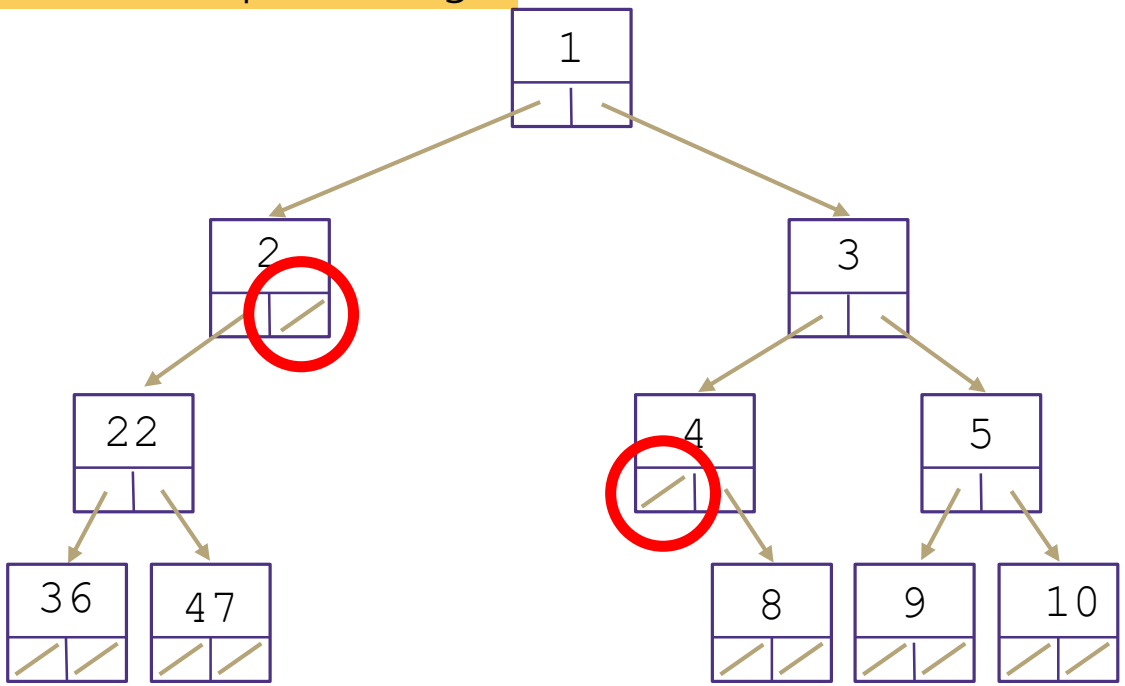
A type of tree with new set of invariants

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

If a node is missing a child, yet other nodes to the right of it have children on that level





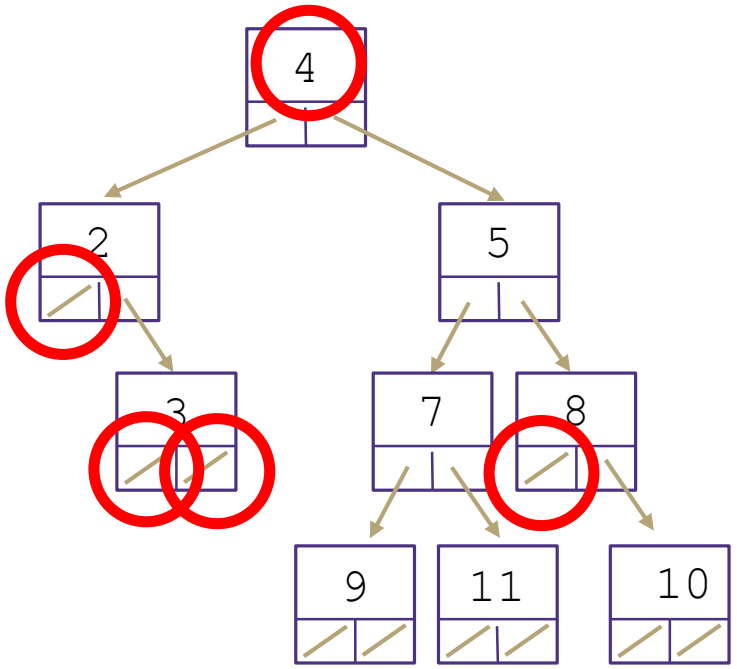


any complete tree is necessarily balanced
complete = all nodes in every level but the last have 2 children

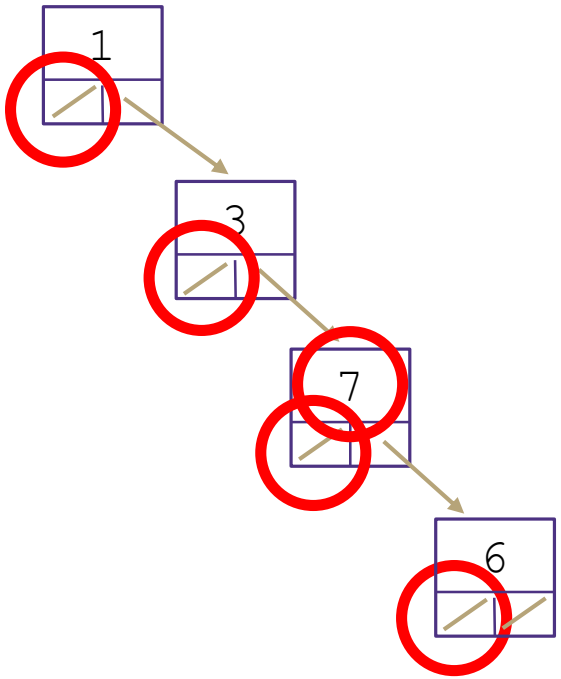
Self Check - Are these valid heaps?

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

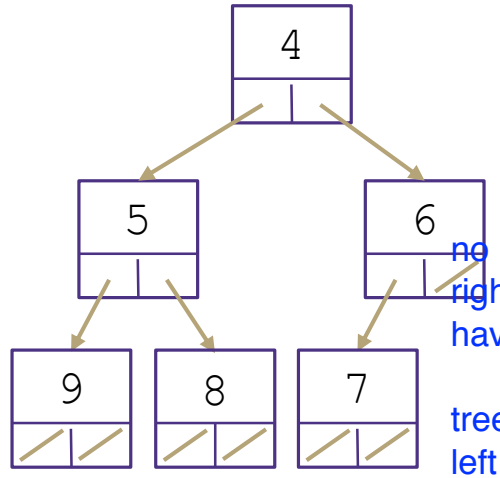
INVALID



INVALID



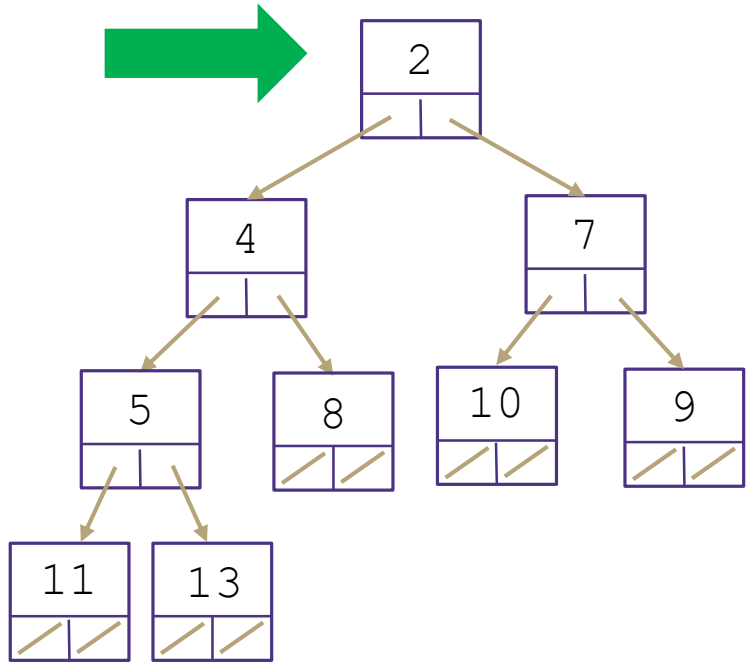
VALID



no nodes to the right of thus have a child

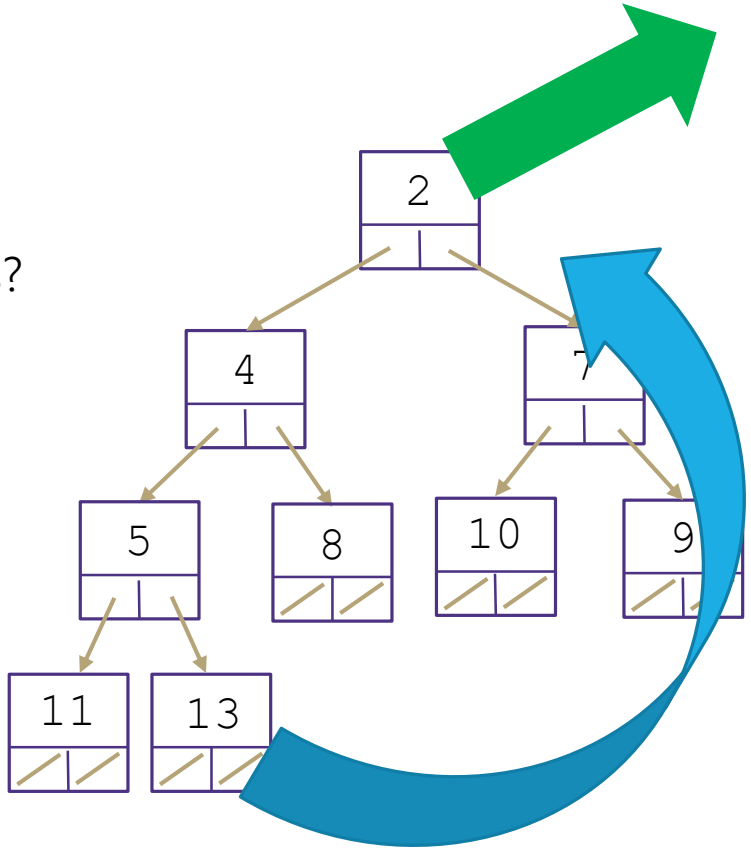
tree was filled left to right NOT a gap

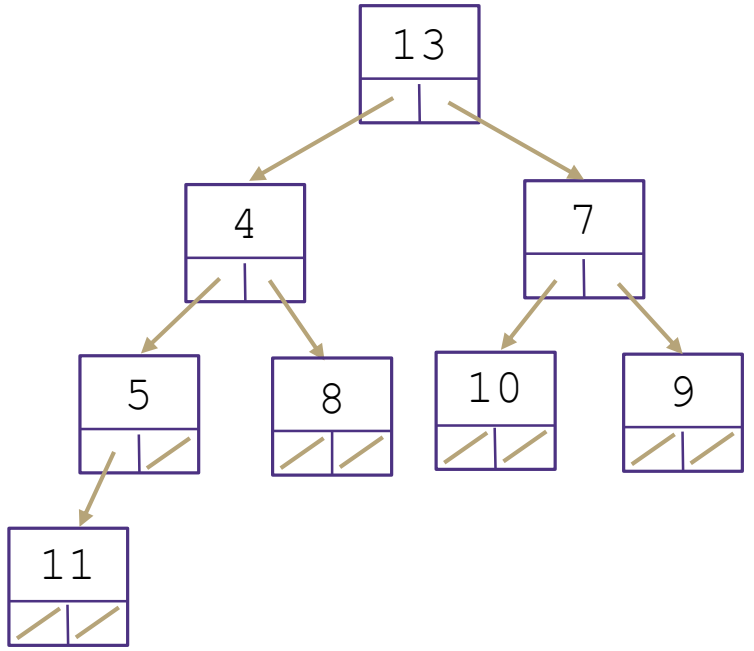
Implementing peekMin()



Implementing removeMin()

Removing overallRoot creates a gap
Replacing with one of its children causes lots of gaps
What node can we replace with overallRoot that wont cause any gaps?





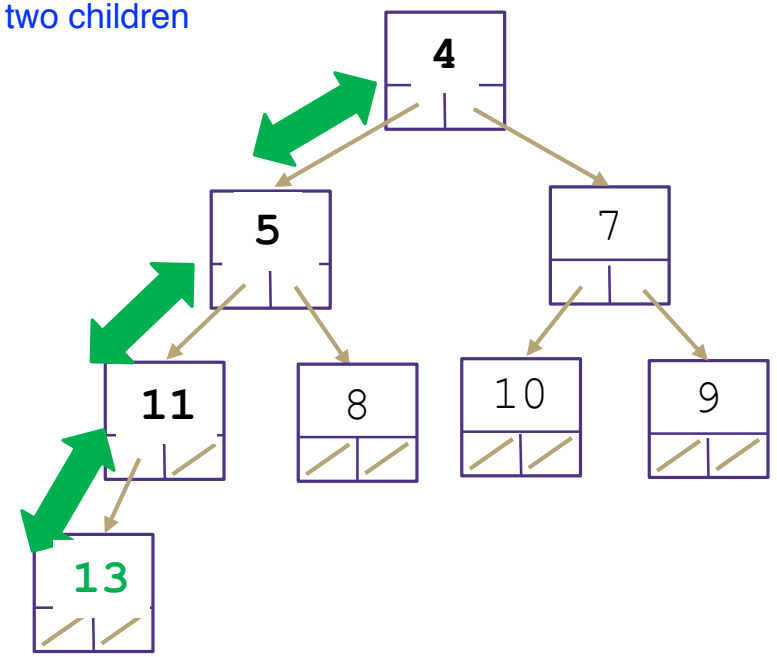
Structure maintained, heap broken

prevents formation of gaps

Fixing Heap – percolate down

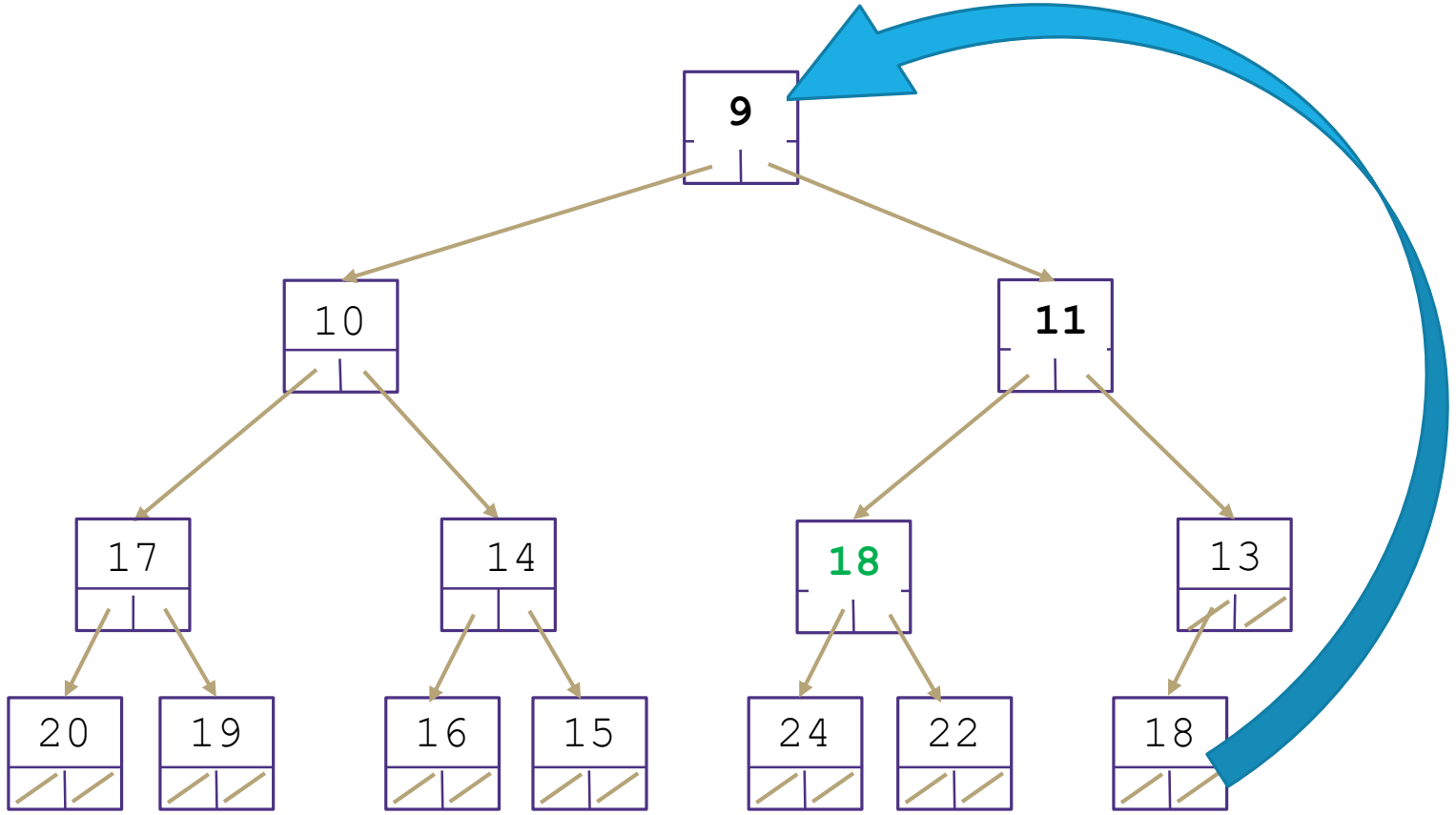
Recursively swap parent with smallest child

Once 13 is at the top, swap its spot with the smaller of two children
repeat until 13 is no longer larger than its two children




```
percolateDown(node) {
    while (node.data is bigger than its children) {
        swap data with smaller child
    }
}
```

Self Check – removeMin() on this tree



Implementing insert()

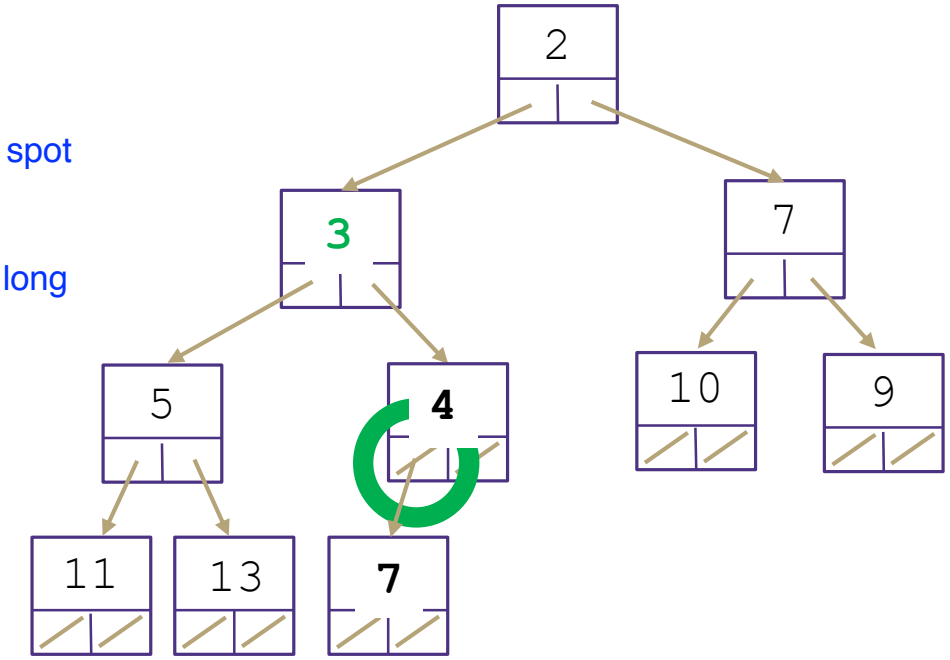
Insert a node to ensure no gaps

Fix heap invariant

percolate **UP**

add node in next spot (only 1 spot wont create a gap)

swap node up with parent as long as node is less than parent.



How long does percolating take?

Up and Down both go through all levels

- O(h)

How tall is the tree?

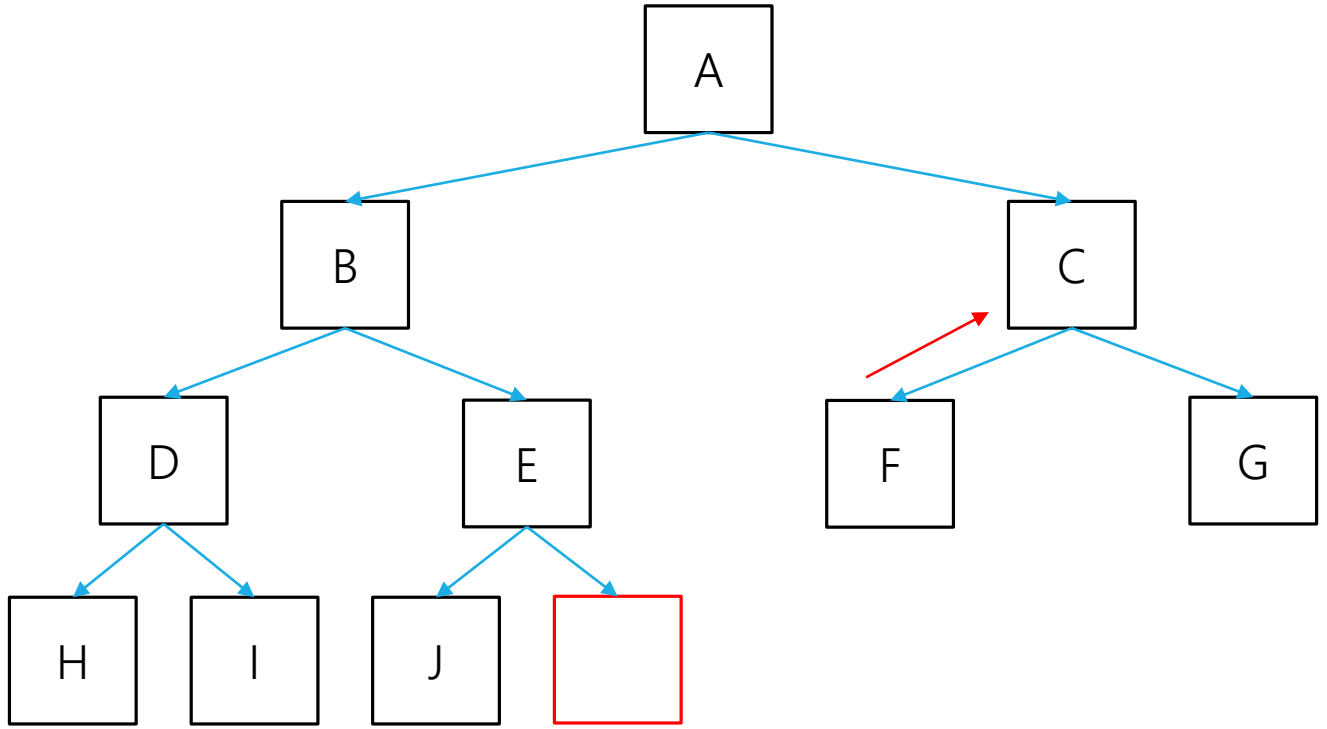
- Complete trees are always balanced (all leaves in level h or h+1)
- h = log n

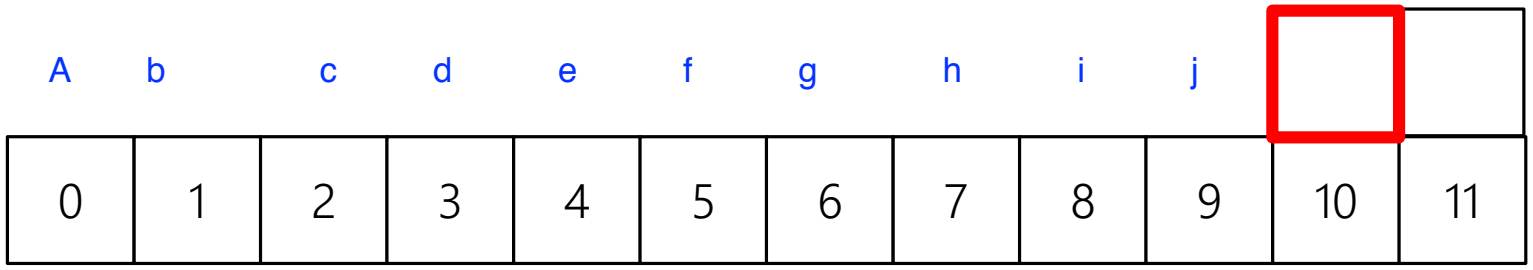
So, percolation is O(log n)

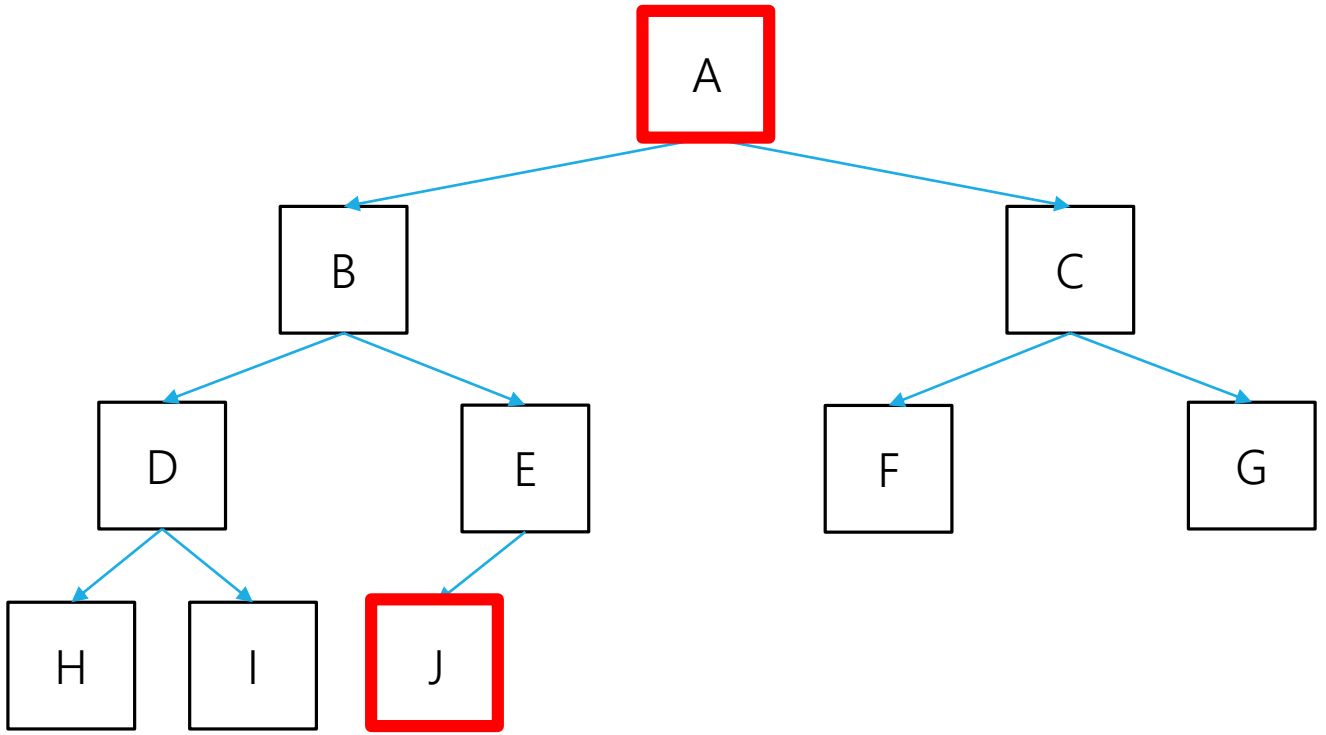
Problems

Finding the last child in a complete tree is O(n)

Finding parents is difficult without back-links (O(n)), so percolate-up is hard







because the tree is complete, we can think of each position in the tree as an index in an array; label indices move from top down and left right

Indices: min = 0 last = n - 1 insert location = n

left child of i = $2i + 1$ right child of i = $2i + 2$ parent of i = $(i - 1) / 2$

Runtimes

Calculating Indices is $\Theta(1)$

peekMin = find min index = $\Theta(1)$

removeMin = find min index + find last index + swap + percolate down

= $\Theta(1)$ + $\Theta(1)$ + $\Theta(1)$ + $\Theta(\log n)$

= $\Theta(\log n)$

insert = find insert location + add to array + percolate up

= $\Theta(1)$ + $\Theta(1)$ + $\Theta(\log n)$

= $\Theta(\log n)$

How Long to build a heap from scratch?

n inserts at Θ(log n) gives Θ(n log n)

But early inserts don't need to percolate as far, is it actually better?

$$\sum_{i=0}^{\log n} 2^i \cdot i = \theta(n \log n)$$

# of levels

how many items at lvl i

percolation dist

level

No.

The basic strategy here was to always percolate up from the bottom. This means the largest layer needs to percolate the farthest.

Can we do better?

Floyd's Build Heap

BUILDS HEAP LINEARLY : O(n)
takes the biggest layer (bottom) and designs algorithm so it has to percolate/travel the shortest distance (fewest operations)

Main Idea: Start from the bottom (end of array) and percolate **down**.

This ensures that the largest layer has the least distance to percolate.

We start with the *structural* invariants (complete binary tree) fulfilled, and then fix the *ordering* invariant (the heap property – parents smaller than children)

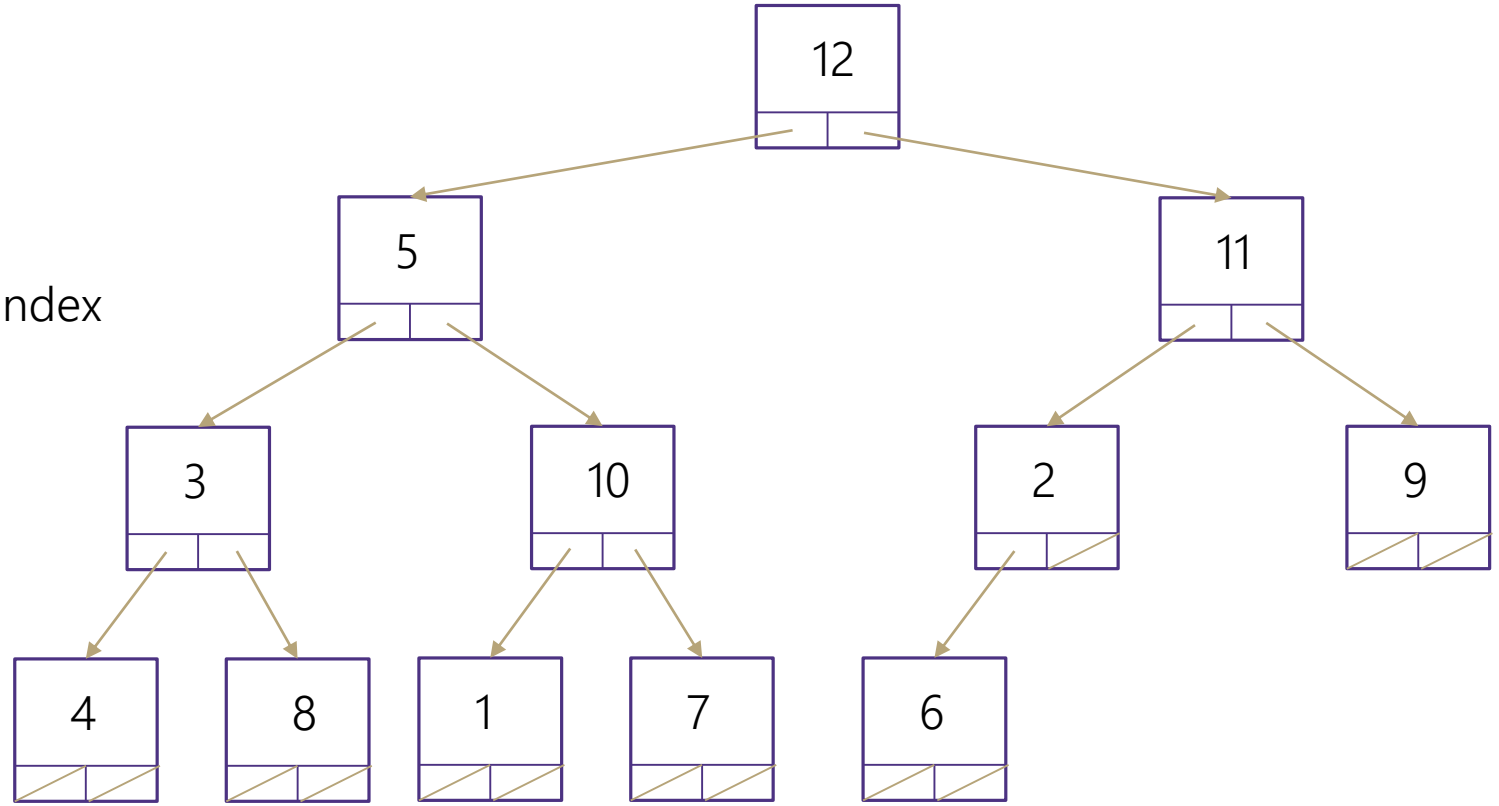
After each percolation, the subtree rooted at the percolated node is a valid min-heap!

buildHeap(array) – Modifies an array in-place to be a heap

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 1, 7, 6

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

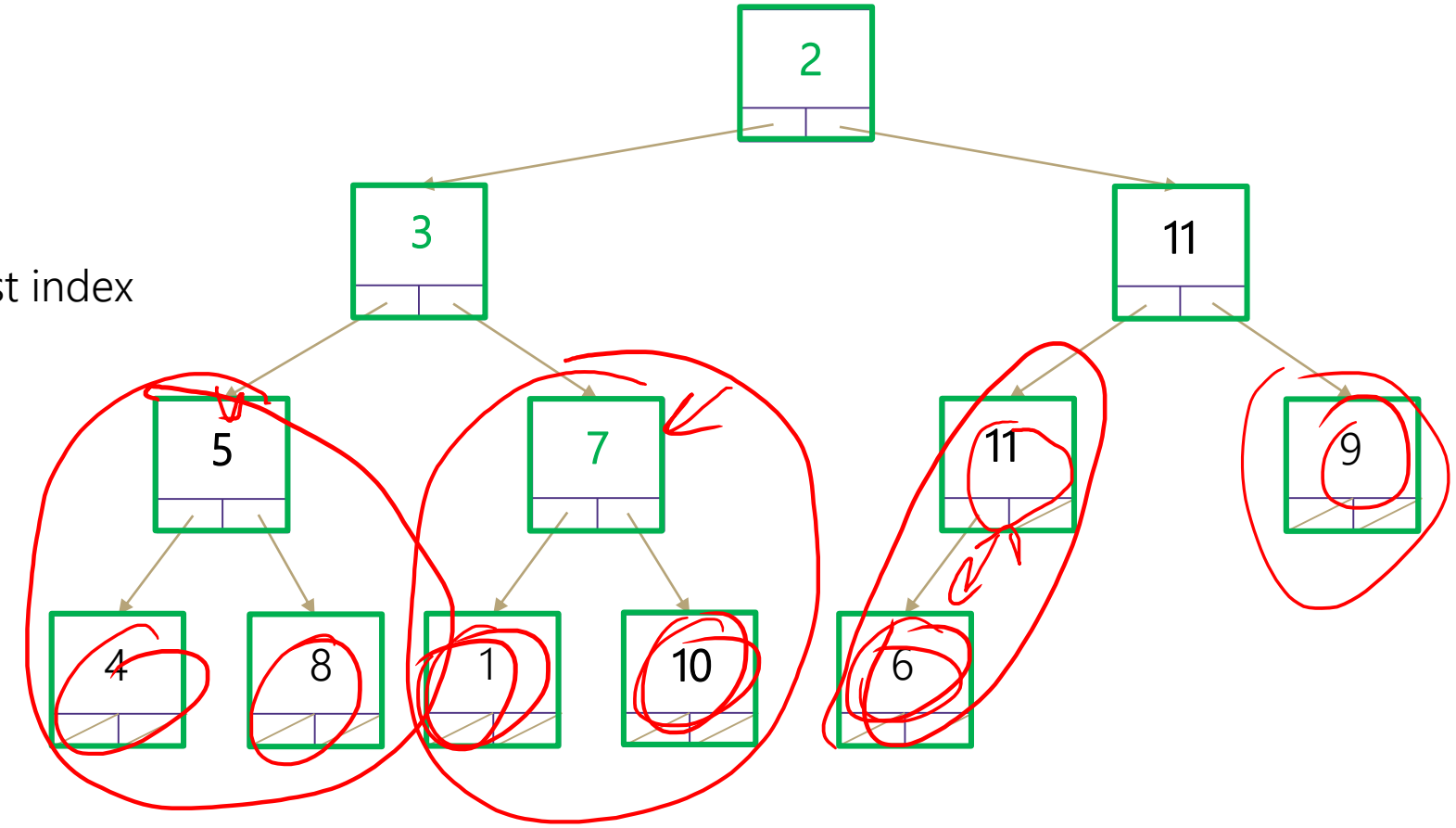


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	1	7	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 1, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
    1. percolateDown level 3
    2. percolateDown level 2
    3. percolateDown level 1
    4. percolateDown level 0



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

Floy'd buildHeap Runtime

For repeated inserts we had $\sum_{i=0}^{\log n} 2^i \cdot i$

Work per node at the i-th level is now (log n – i) since we percolate down

$$S = \sum_{i=0}^{\log n} 2^i \cdot (\log n - i) = \theta(n)$$

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r}$$

$$2S = 2\sum_{i=0}^{h} 2^i (h-i)$$

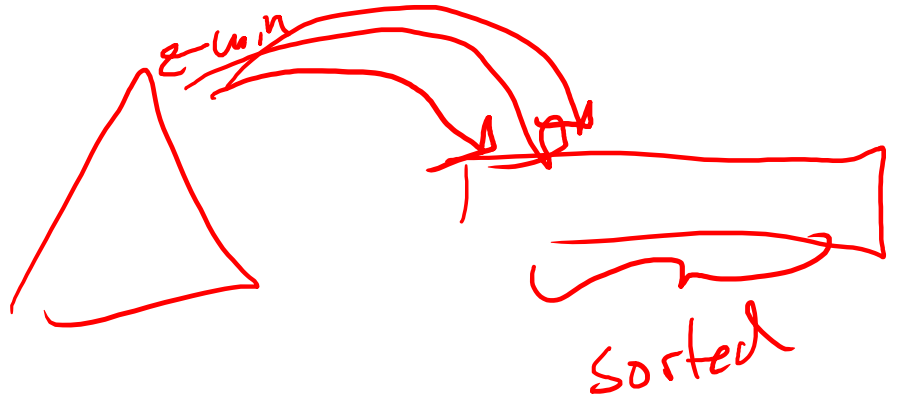
$$S = 2S - S = \sum_{i=0}^{h} 2^{i+1}(h-i) - \sum_{i=0}^{h} 2^i (h-i)$$

$$= \sum_{i=1}^{h+1} 2^i (h-i+1) - \sum_{i=0}^{h} 2^i (h-i)$$

$$= -h + \sum_{i=1}^{h} 2^i = -h + \frac{1-2^{h+1}}{1-2} + 1 \rightarrow 2^{h+1} - h$$

$$\simeq 2^{\log n + 1} - h \simeq \theta(h)$$

Heap Sort



Can we use a heap to sort data?

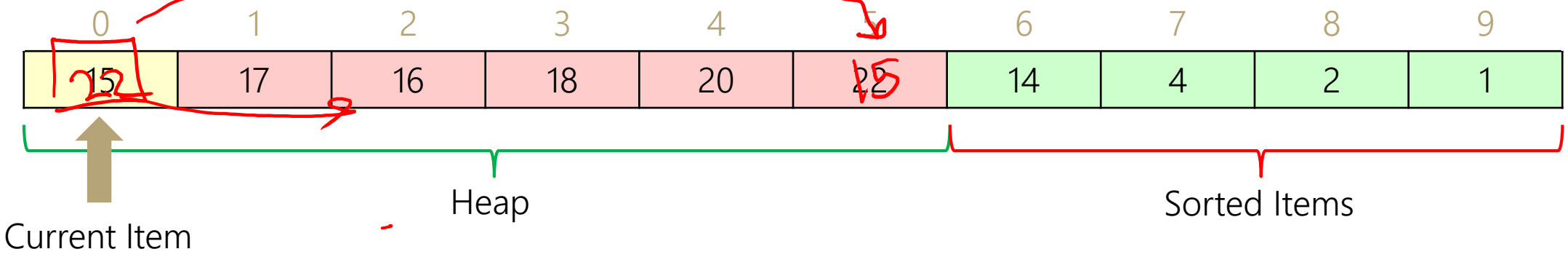
Yes! Heap Sort: repeatedly removeMin, and add this to an output array.

This takes an extra array of space. Can we do better?

Yes! Re-use the space vacated at the end of the array after each remove.

This reverses the sorting order – either reverse the list (O(n)), or use a max-heap!

In Place Heap Sort




```
public void inPlaceHeapSort(collection) {
   E[] heap = buildHeap(collection)
   for (n)
      output[n - i - 1] = removeMin(heap)
}
```

Worst case runtime? O(nlogn)

Best case runtime? O(nlogn)

Average runtime? O(nlogn)

Complication: final array is reversed!

- Run reverse afterwards (O(n))
- Use a max heap
- Reverse compare function to emulate max heap

Announcements

Midterm is next Friday in class

- All material up to and including next Monday's lecture is fair game

HW3 (an individual written assignment) will be posted soon

- Excellent midterm review (similar questions)

- Due after midterm, but ***HIGHLY SUGGESTED*** you at least solve, if not write up, all of the questions before midterm as studying.

Project 1 is due Friday by midnight