6.006 Introduction to Algorithms
Spring 2008

6.006 Recitation

Build 2008.16

Coming up next...

- Sorting
  - Scenic Tour: Insertion Sort, Selection Sort, Merge Sort
  - New Kid on the Block: Merge Sort
- Priority Queues
  - Heap-Based Implementation

Sorting

- Input: array **a** of **N** keys
- Output: a permutation $\mathbf{a_s}$ of **a** such that $a_s[k] < a_s[k+1]$
- Stable sorting:

Sorting

- Maybe the oldest problem in CS
- Reflects our growing understanding of algorithm and data structures
- Who gives a damn?
  - All those database tools out there

Sorting Algorithms: Criteria

What	**Why**
Speed	That's what 6.006 is about
Auxiliary Memory	External sorting, memory isn't that cheap
Simple Method	You're learning / coding / debugging / analyzing it
# comparisons, data moving	Keys may be large (strings) or slow to move (flash memory)

Insertion Sort

- Base: a[0:1] has 1 element ⇒ is sorted
- Induction: a[0:k] is sorted, want to grow to a[0:k+1] sorted
    - find position of a[k+1] in a[0:k]
    - insert a[k+1] in a[0:k]

5	**8**	2	7	1	4	3	6
5	8	**2**	7	1	4	3	6
2	5	8	**7**	1	4	3	6
2	5	7	8	**1**	4	3	6
1	2	5	7	8	**4**	3	6
1	2	4	5	7	8	**3**	6
1	2	4	5	7	8	3	**6**
1	2	3	4	5	6	7	8

Insertion Sort: Costs

- Find position for a[k+1] in a[0:k] - O(log(k))
  - use binary search
- Insert a[k+1] in a[0:k]: O(k)
  - shift elements
- Total cost: $O(N \cdot \log(N)) + O(N^2) = O(N^2)$

- Pros:
  - Optimal number of comparisons
  - O(1) extra memory (no auxiliary arrays)
- Cons:
  - Moves elements around a lot

Selection Sort

- Base case: a[0:0] has the smallest 0 elements in a
- Induction: a[0:k] has the smallest k elements in a, sorted; want to expand to a[k+1]
    - find min(a[k+1:N])
    - swap it with a[k+1]

5	8	2	7	**1**	4	3	6
1	8	**2**	7	5	4	3	6
1	2	8	7	5	4	**3**	6
1	2	3	7	5	**4**	8	6
1	2	3	4	**5**	7	8	6
1	2	3	4	5	7	8	**6**
1	2	3	4	5	6	8	**7**
1	2	3	4	5	6	7	**8**

Selection Sort: Costs

- find minimum in a[k+1:N]) - O(N-k)
  - scan every element
- swap with a[k] - O(1)
  - need help for this?
- Total cost: $O(N^2) + O(N) = O(N^2)$

- Pros:
  - Optimal in terms of moving data around
  - O(1) extra memory (no auxiliary arrays)
- Cons:
  - Compares a lot

Merge-Sort

1. Divide
   - Break into 2 sublists
2. Conquer
   - 1-elements lists are sorted
3. Profit
   - Merge sorted sublists

5	8	2	7	1	4	3	6
5	8	2	7	1	4	3	6
5	8	2	7	1	4	3	6
2	5	7	8	1	3	4	6
1	2	3	4	5	6	7	8

There is no step 6

There is no step 7

There is no step 8

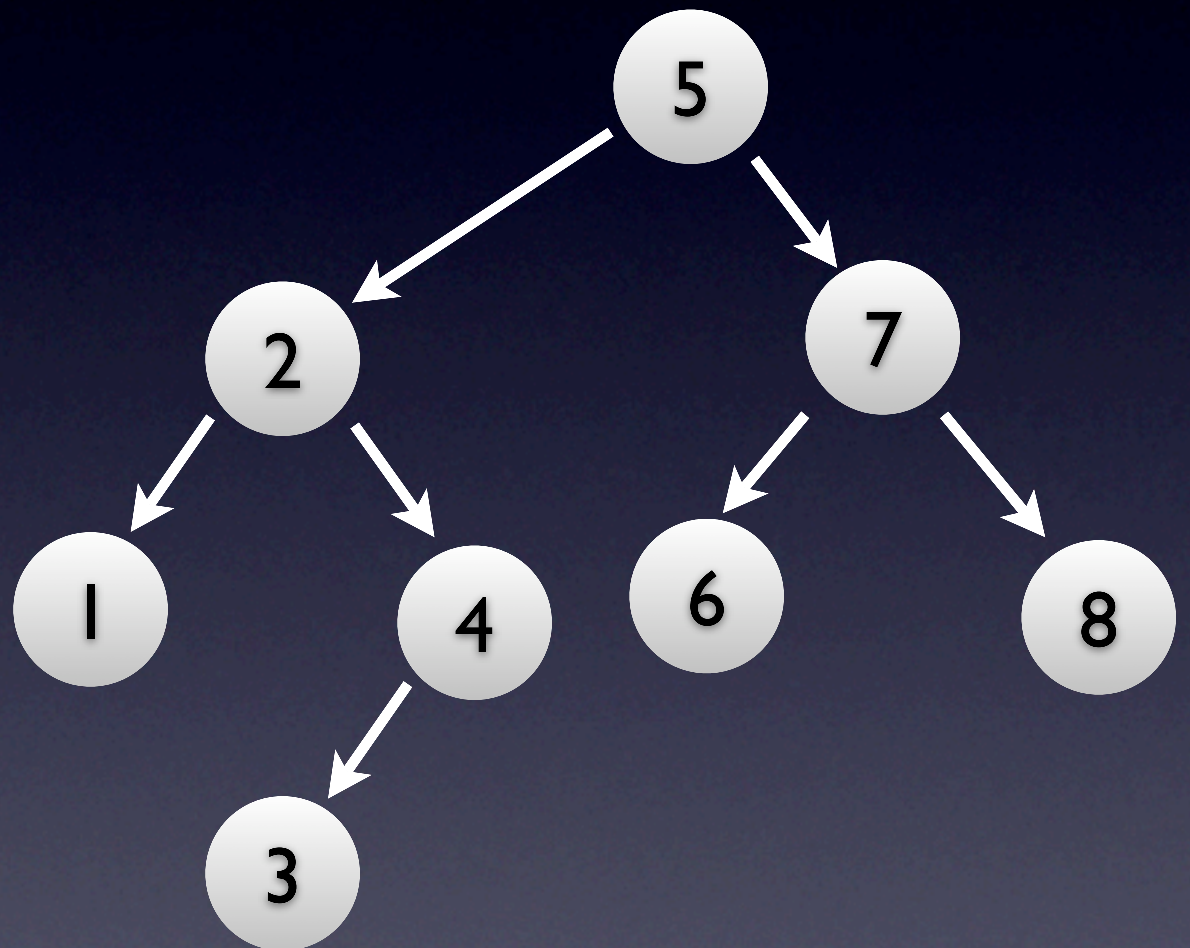
Merge-Sort: Cost

- You should be ashamed of if you don't know!
- $T(N) = 2T(N/2) + \Theta(N)$
- Recursion tree
  - $O(\log(N))$ levels, $O(N)$ work / level
- Total cost: $O(N \cdot \log(N))$

- Pros:
  - Optimal number of comparisons
  - Fast
- Cons:
  - $O(N)$ extra memory (for merging)

BST Sort

- Build a BST out of the keys
- Use inorder traversal to obtain the keys in sorted order
  - Or go to minimum(), then call successor() until it returns None



BST Sort: Cost

- Building the BST - $O(N \cdot \log(N))$
  - Use a balanced tree
- Traversing the BST - $O(N)$
  - Even if not balanced
- Total cost: $O(N \cdot \log(N))$

- Pros:
  - Fast (asymptotically)
- Cons:
  - Large constant
  - $O(N)$ extra memory (left/right pointers)
  - Complex code

Best of Breed Sorting

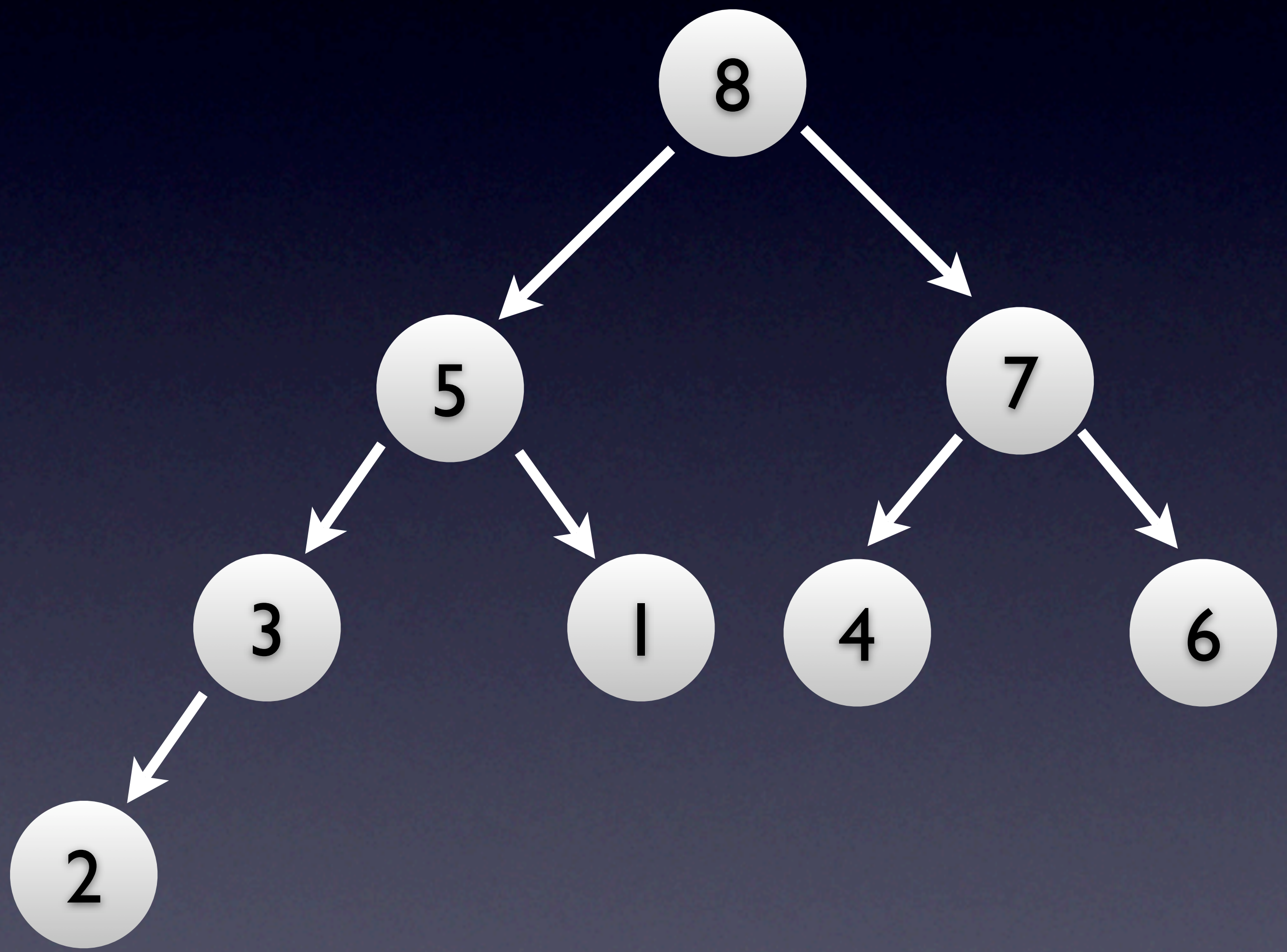
Speed	$O(N \cdot \log(N))$
Auxiliary Memory	$O(1)$
Code complexity	Simple
Comparisons	$O(N \cdot \log(N))$
Data movement	$O(N)$

Heap-Sort

Speed	$O(N \cdot \log(N))$	✓
Auxiliary Memory	$O(1)$	✓
Code complexity	Simple	✓
Comparisons	$O(N \cdot \log(N))$	✓
Data movement	$O(N)$	✗

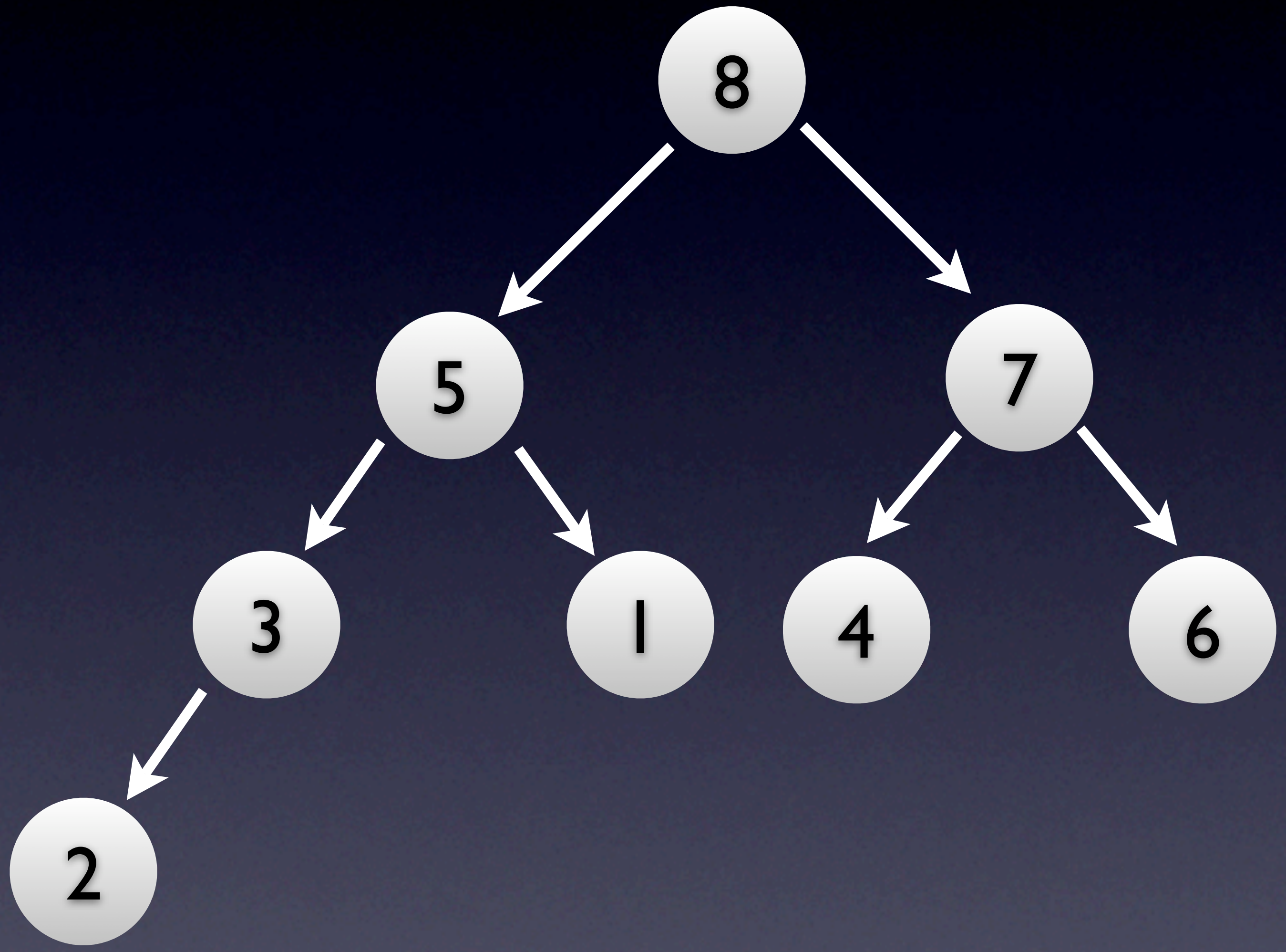
Heap-Sort uses a... Heap (creative, eh?)

- Max-Heap DT
  - Almost complete binary tree
  - Root node's key >= its children's keys
  - Subtrees rooted at children are Max-Heaps as well



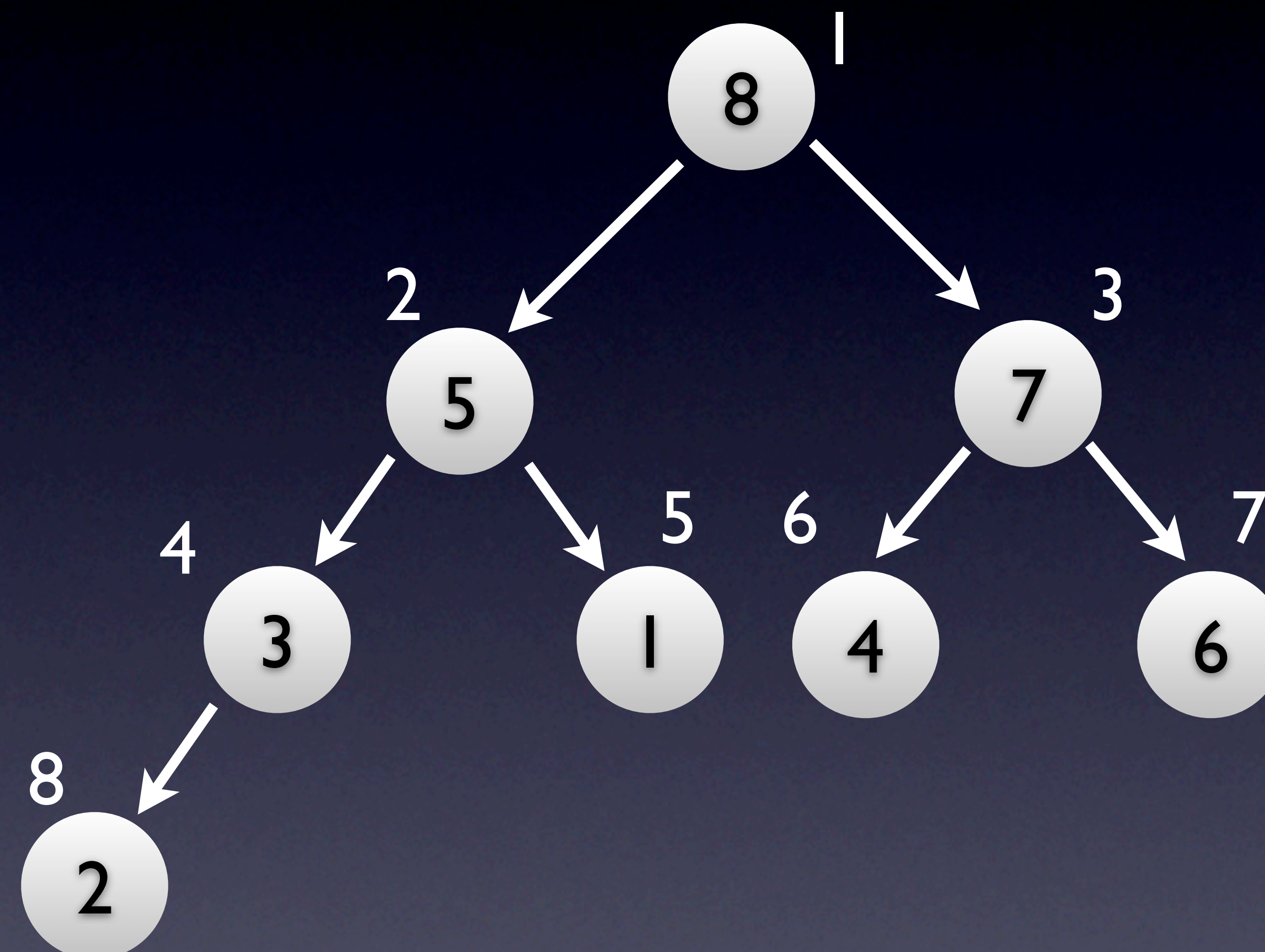
Max-Heap Properties

- Very easy to find max. value
  - look at root, doh
- Unlike BSTs, it's very hard to find any other value
  - 6 (3rd largest key) at same level as 1 (min. key)



Heaps Inside Arrays

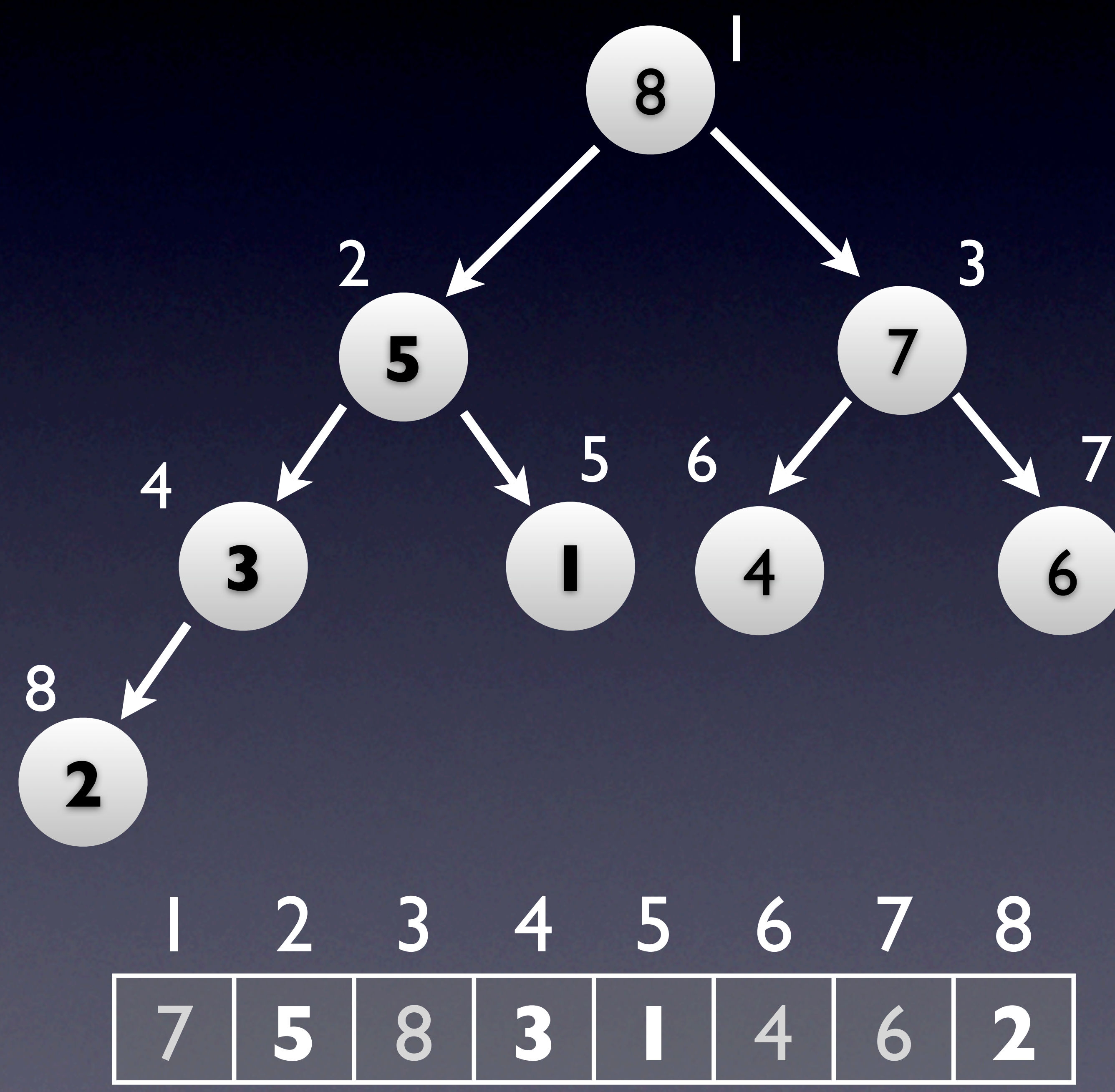
- THIS IS WHY HEAPS ROCK OVER BSTs
  - No need to store a heap as a binary tree (left, right, parent pointers)
- Store keys inside array, in level-order traversal



1	2	3	4	5	6	7	8
8	5	7	3	1	4	6	2

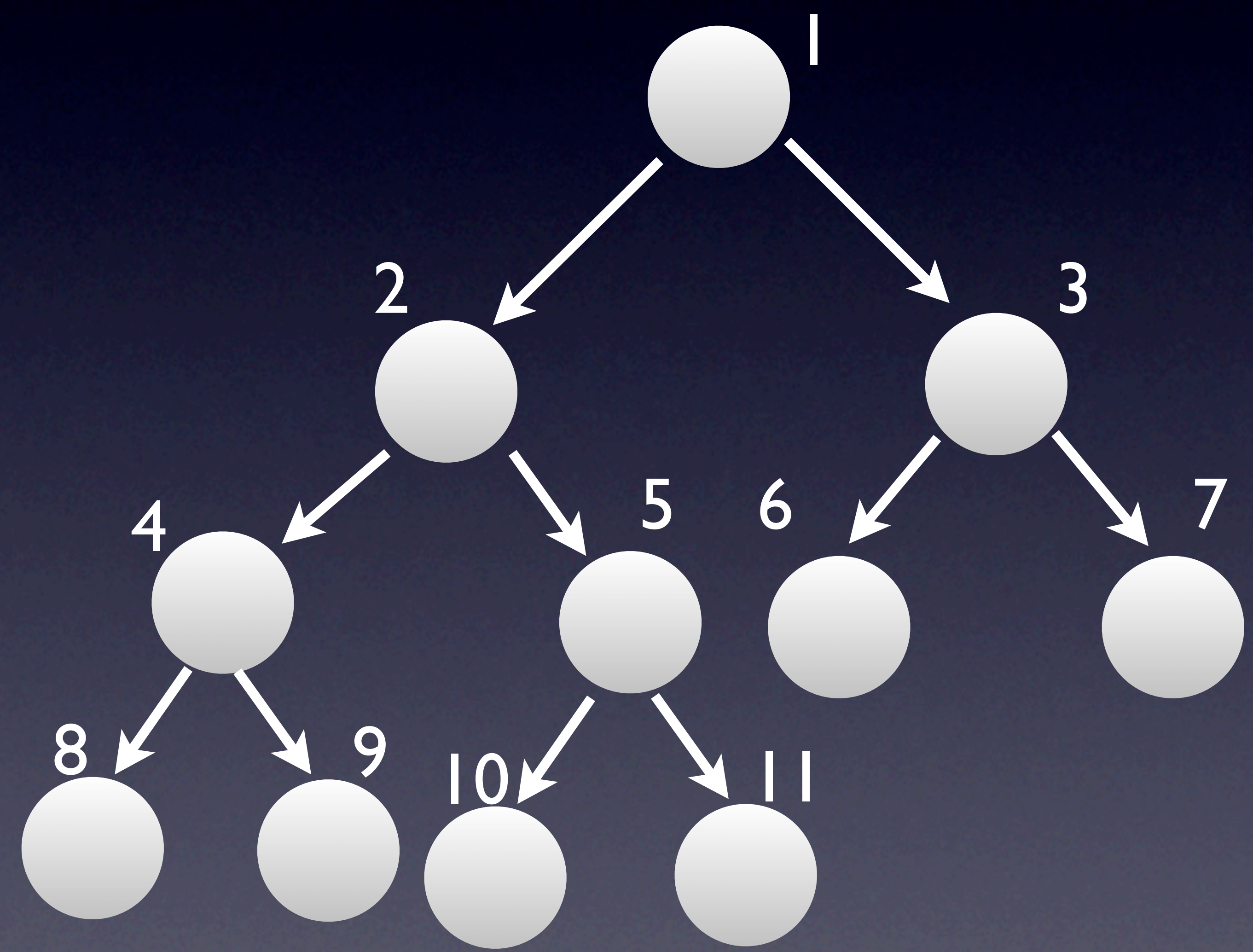
Heaps Inside Arrays

- Work with arrays, think in terms of trees
  - Left subtree of 8 is in bold... pretty mind-boggling, eh?
  - Prey that you don't have to debug this



Heaps Inside Arrays

- root index: 1
- left_child(node_index):
  - $node\_index \cdot 2$
- right_child(node_index):
  - $node\_index \cdot 2 + 1$
- parent(node_index):
  - $\lfloor node\_index / 2 \rfloor$



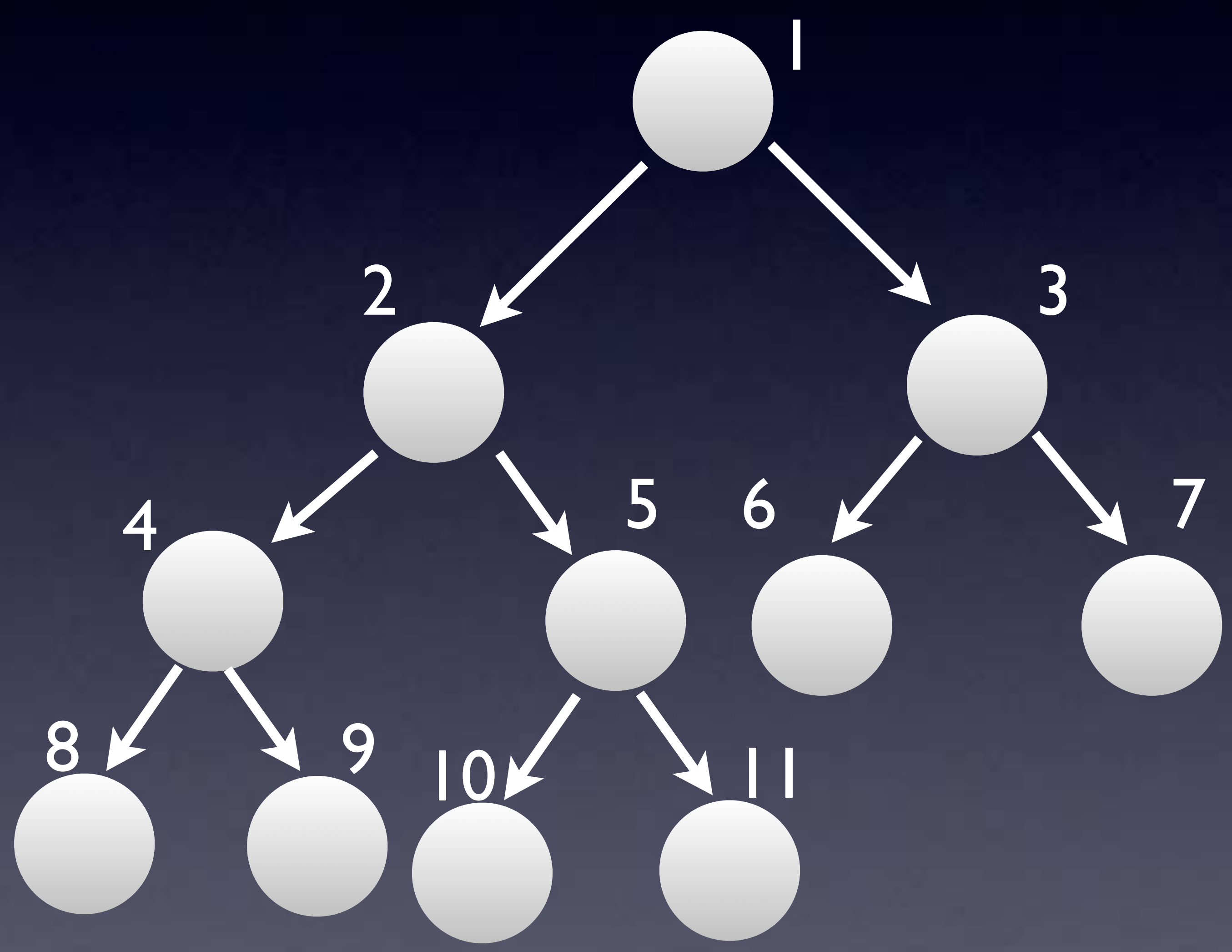
Heaps Inside Arrays

- How to recall this

  1. draw the damn heap (see right)

  2. remember the concept (divide / multiply by 2)

  3. work it out with the drawing



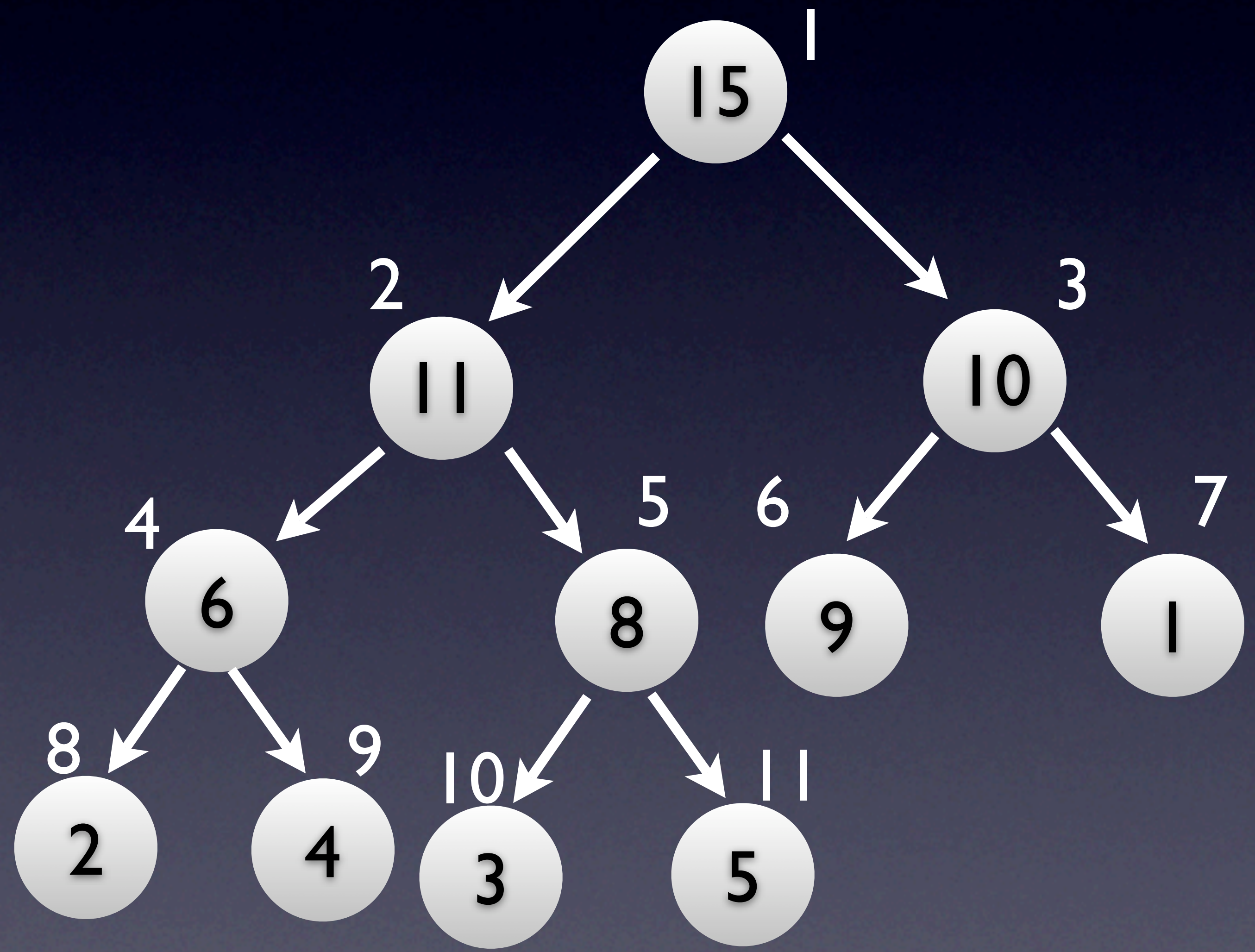
Heaps Inside Arrays: Python Perspective

- Lists are the closest thing to array
- Except they grow
  - Just like our growing hashes
  - Amortized O(1) per operation

1	2	3	4	5	6	7	8
7	**5**	8	**3**	**1**	4	6	**2**

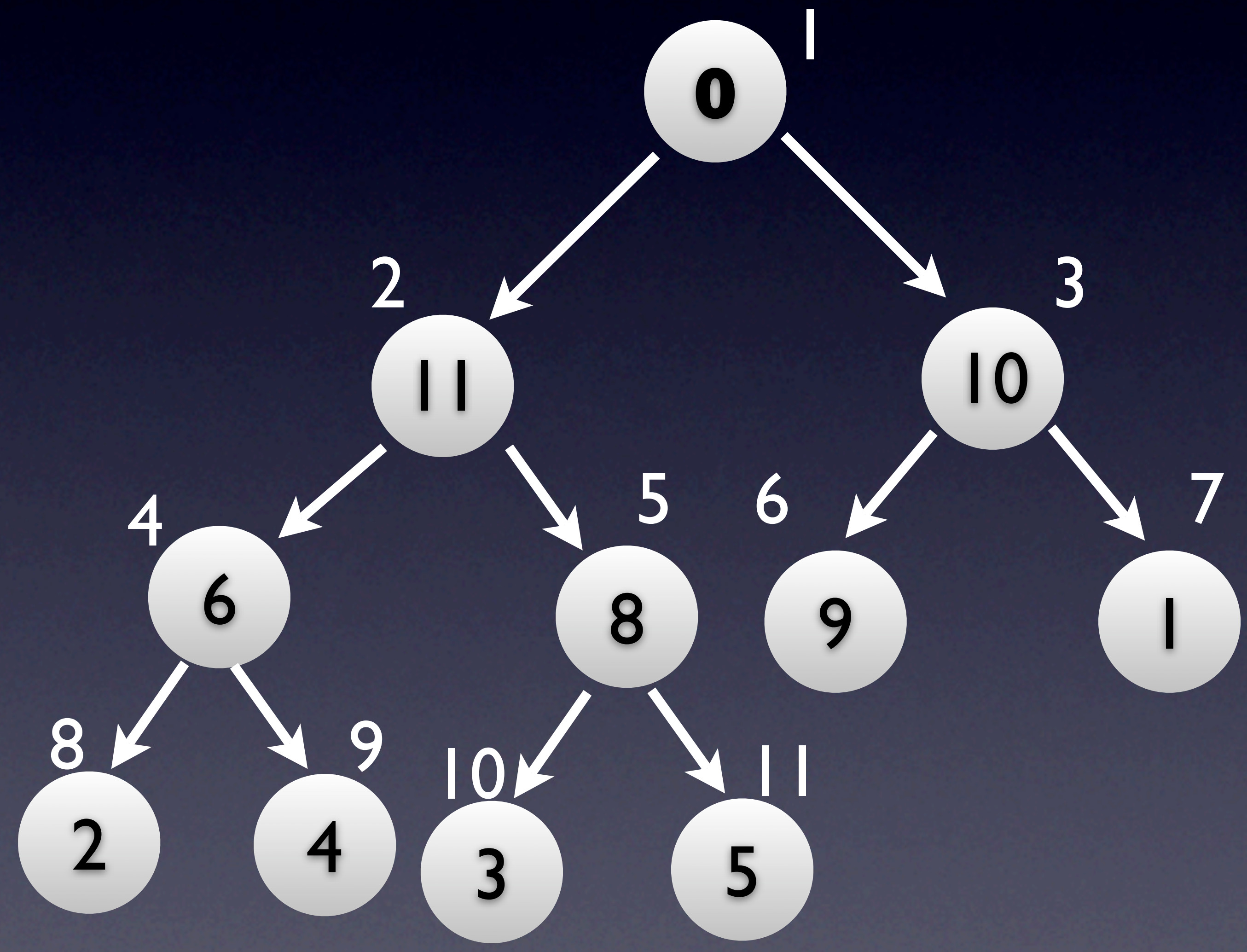
Messing with Heaps

- Goal:
  1. Change any key
  2. Restore Max-Heap invariants



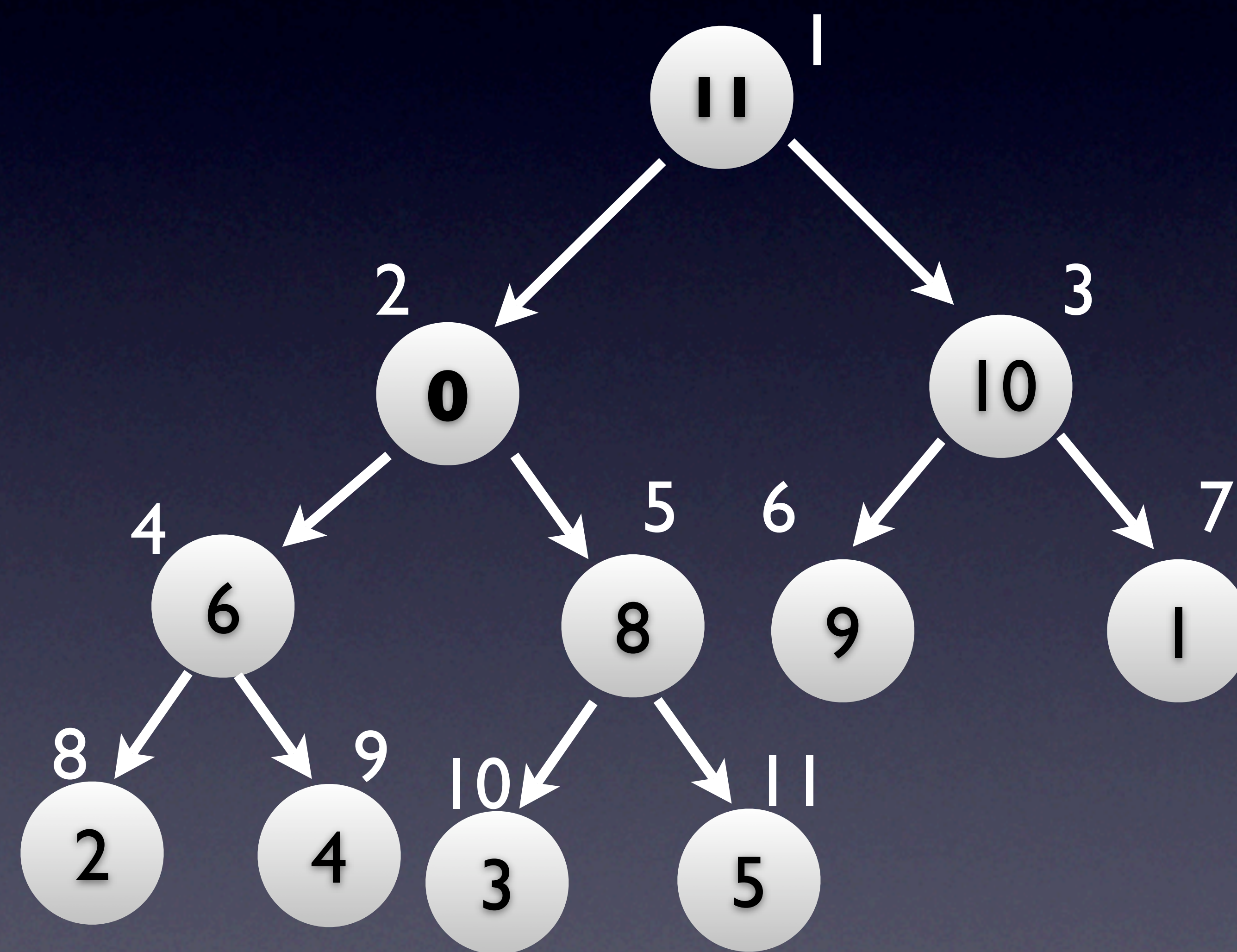
Messing with Heaps: Percolate

- Issue
  - key's node becomes smaller than children
  - only possible after decreasing a key
- Solution
  - percolate (huh??)



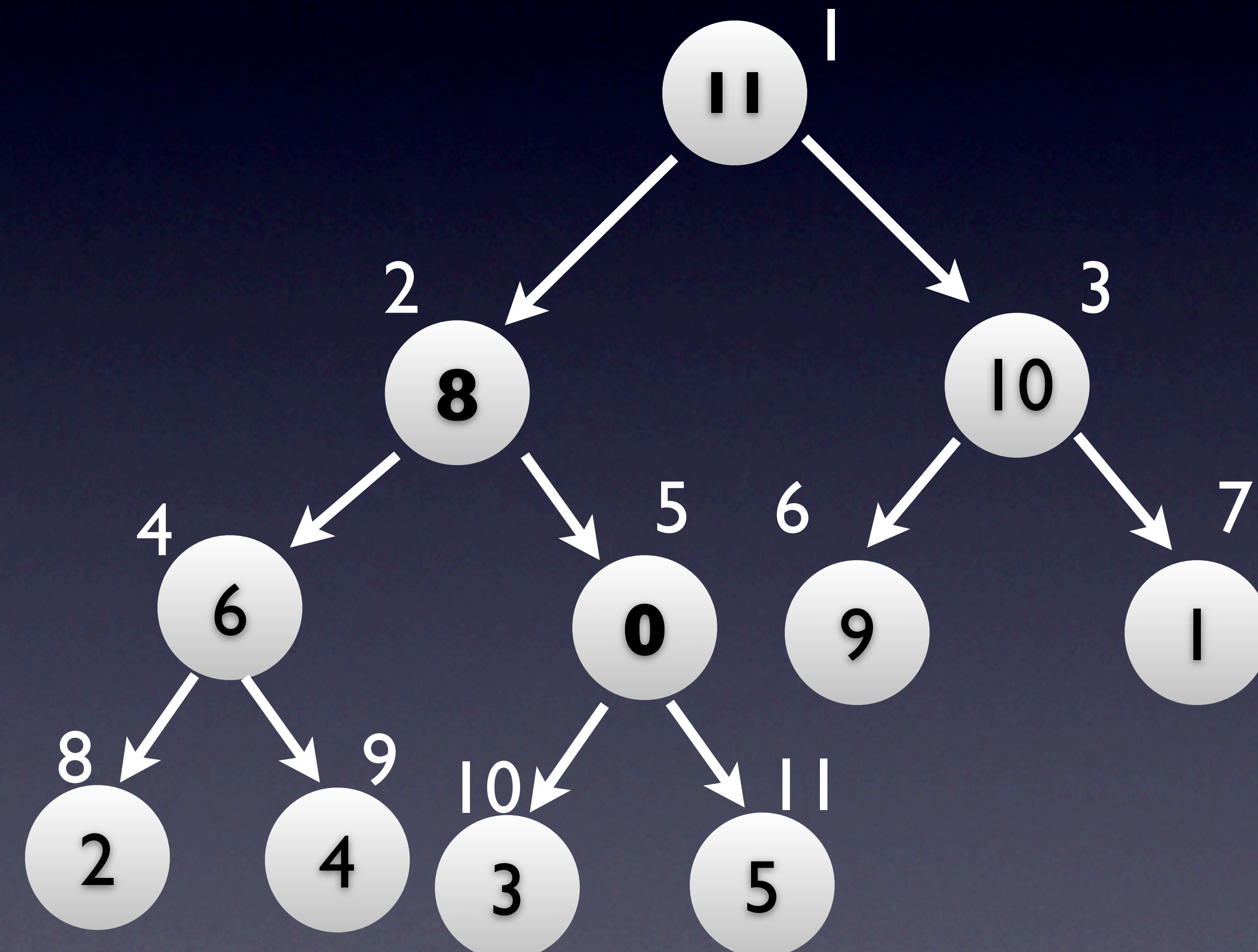
Messing with Heaps: Percolate

- Percolate:
  - swap node's key with max(left child key, right child key)
    - Max-Heap restored locally
    - the child we didn't touch still roots a Max-Heap



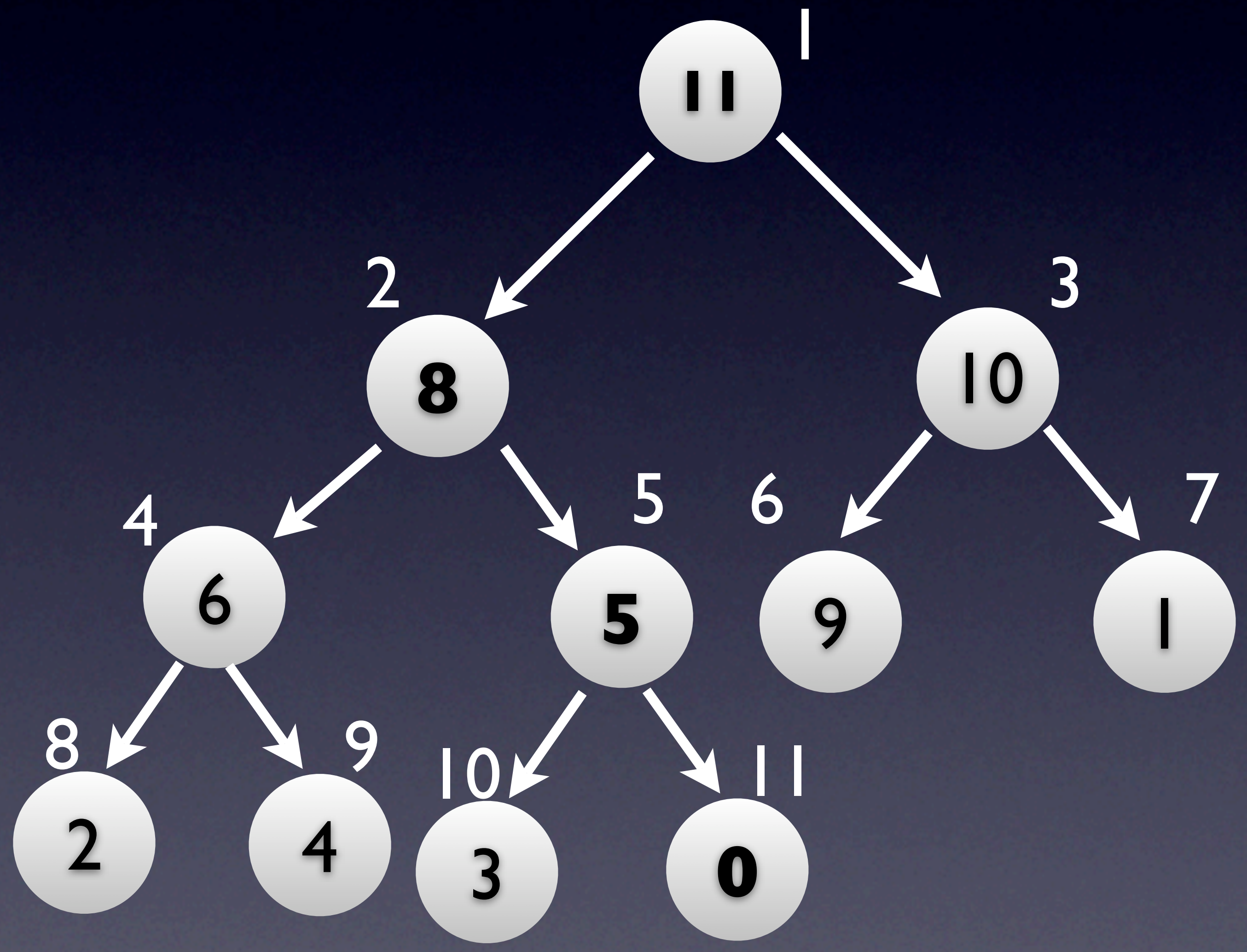
Messing with Heaps: Percolate

- Percolate
  - Issue: swapping decreased the key of the child touched
    - child might not root a Max-Heap
  - Solution: keep percolating



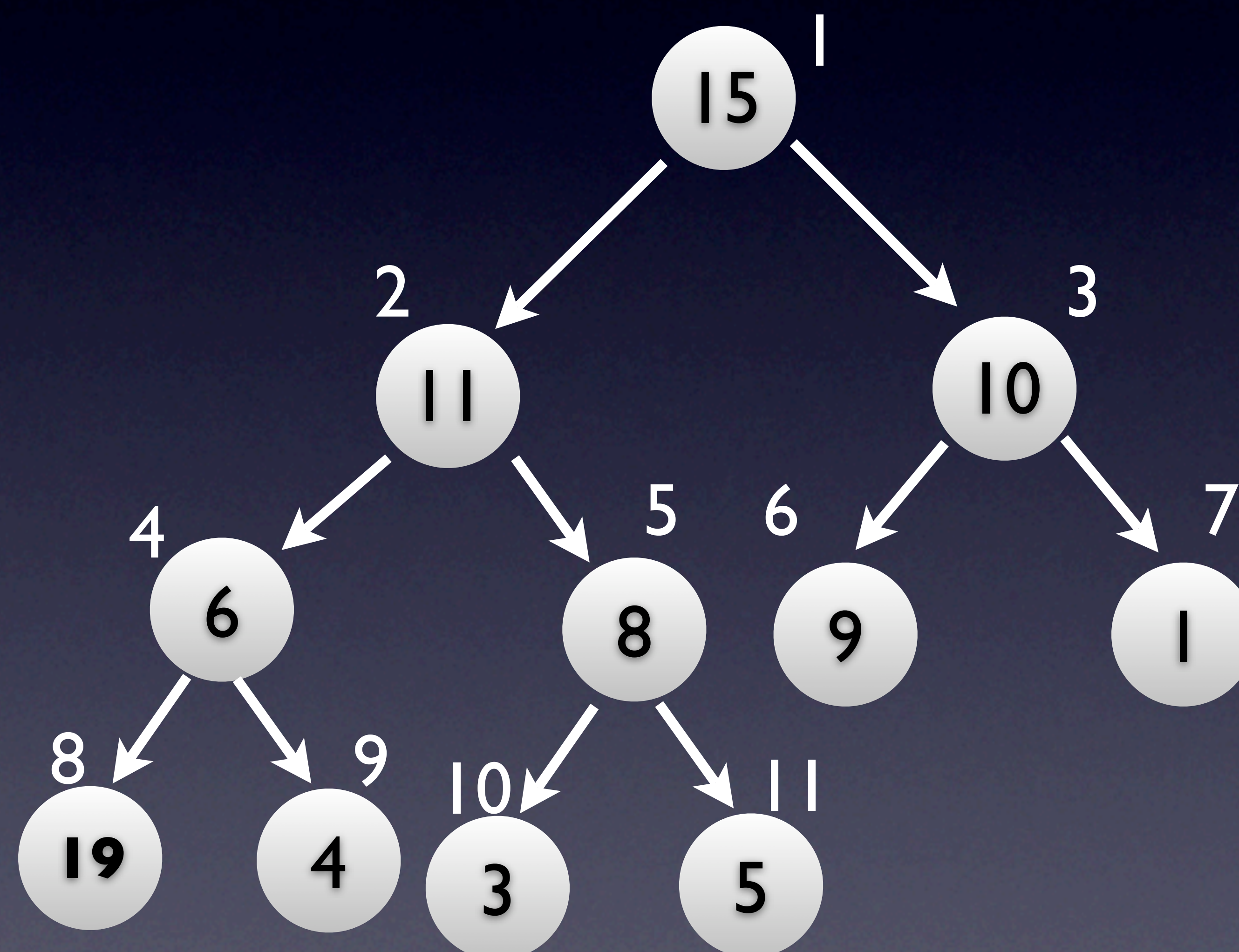
Messing with Heaps: Percolate

- Percolating is finite:
  - leaves are always Max-Heaps
- Percolate cost:
  - O(heap height - node's level)
  - O(log(N) - log(node))



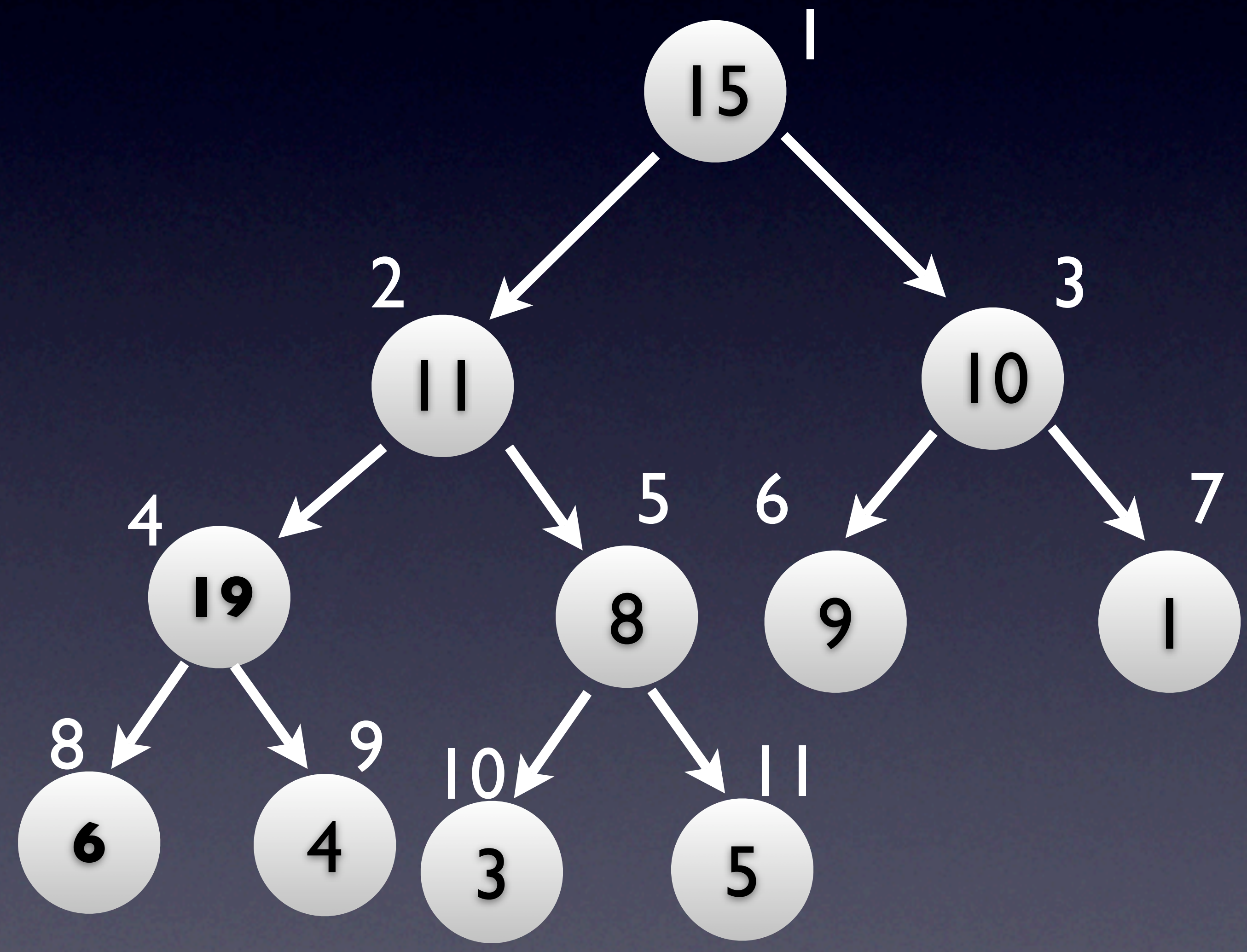
Messing with Heaps: Sift

- Issue
  - key's node becomes larger than parent
  - only possible after increasing a key
- Solution
  - sift (huh??)



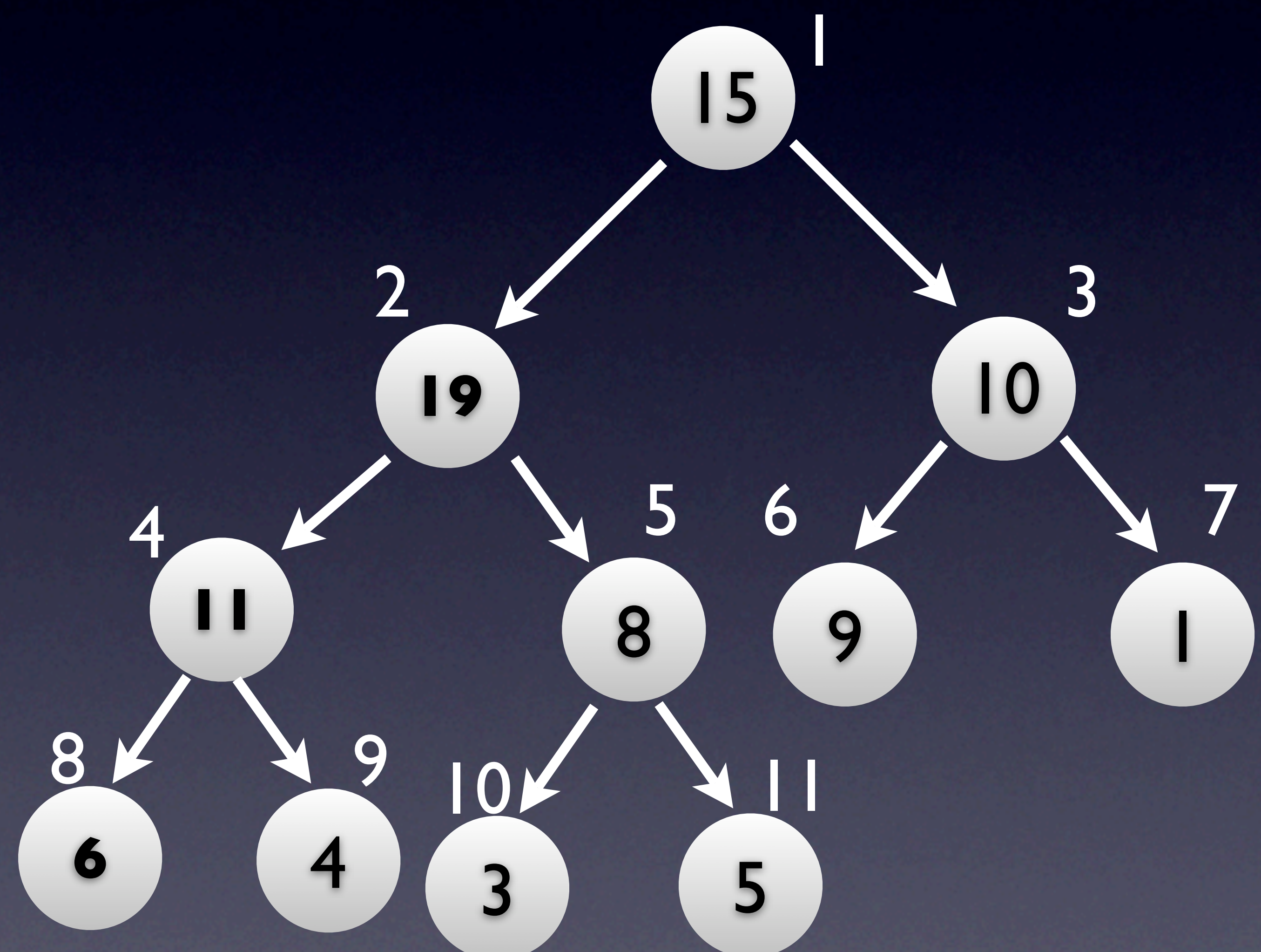
Messing with Heaps: Sift

- Sift
  - swap node's key with parent's key
  - parent's key was >= node's key, so must be >= children keys
  - Max-Heap restored for node's subtree



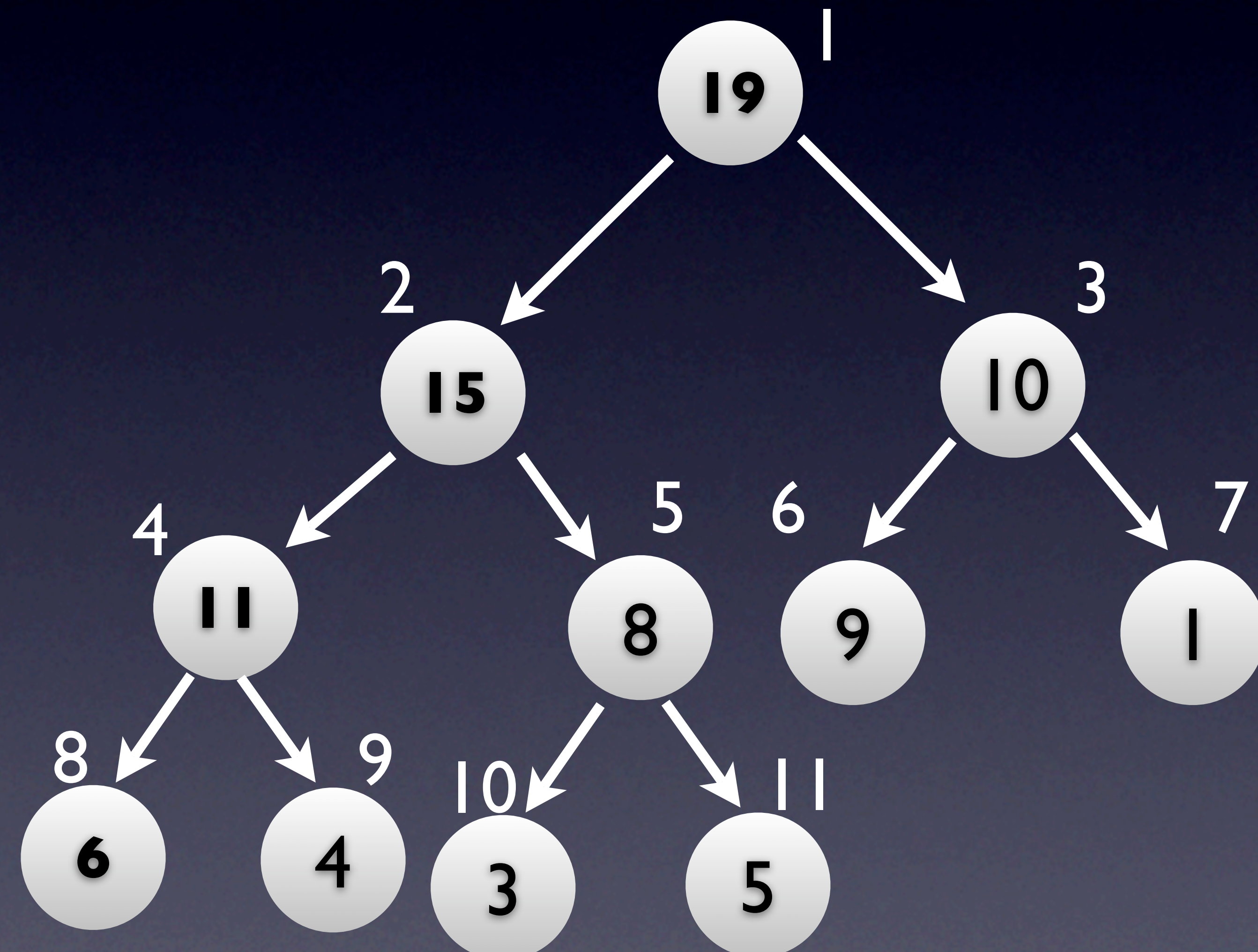
Messing with Heaps: Sift

- Sift
  - Issue: swapping increased the key of the parent
    - parent might not root a Max-Heap
  - Solution: keep sifting



Messing with Heaps: Sift

- Sifting is finite:
  - root has no parent, so it can be increased at will
- Sift cost:
  - O(height)
  - O(log(node))

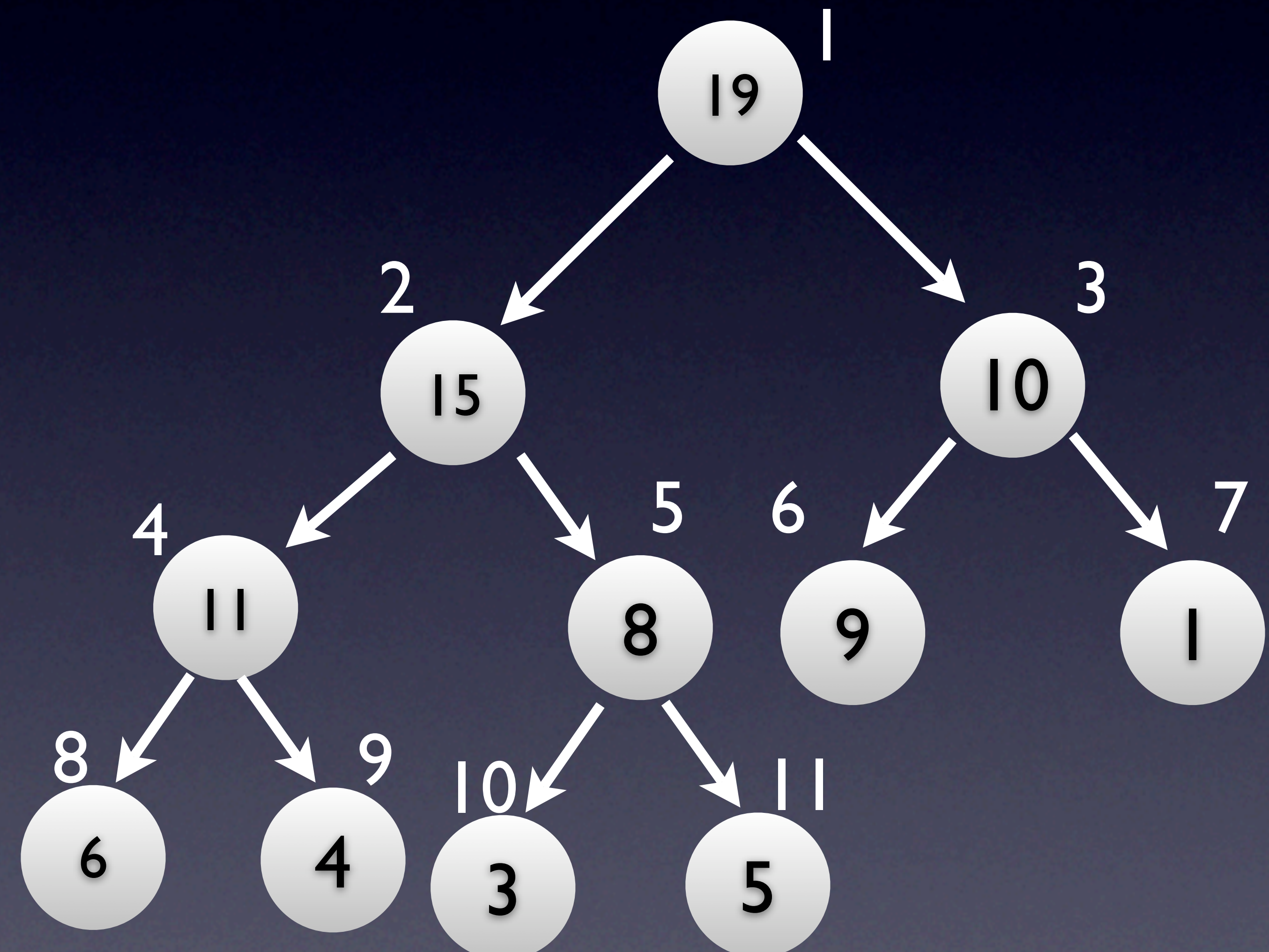


Messing with Heaps

- Update(node, new_key)
  - old_key ← heap[node].key
  - heap[node].key ← new_key
  - if new_key < old_key: sift(node)
  - else: percolate(node)

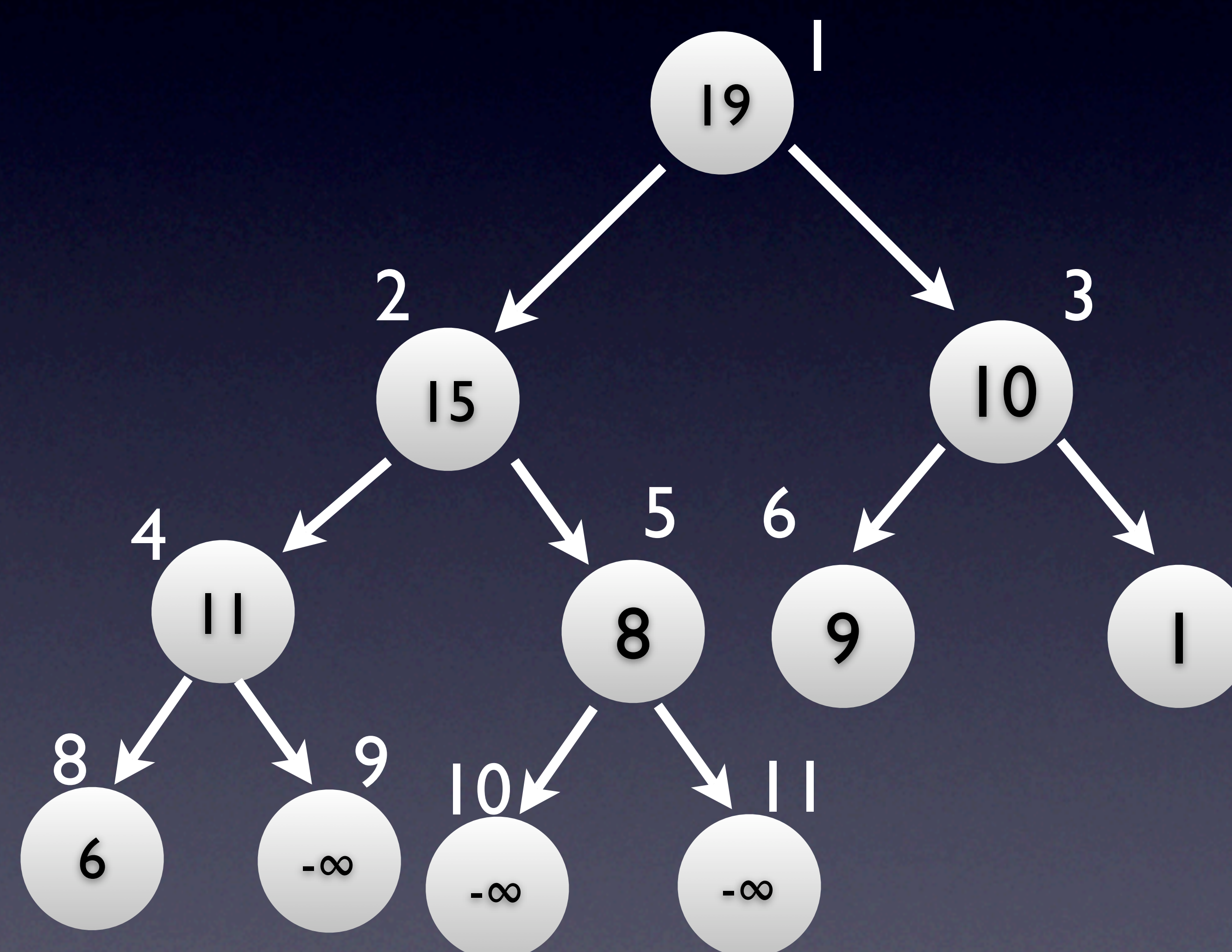
Messing with Heaps II

- Goal
  - Want to shrink or grow the heap
- Growing:
  - inserting keys
- Shrinking:
  - deleting keys



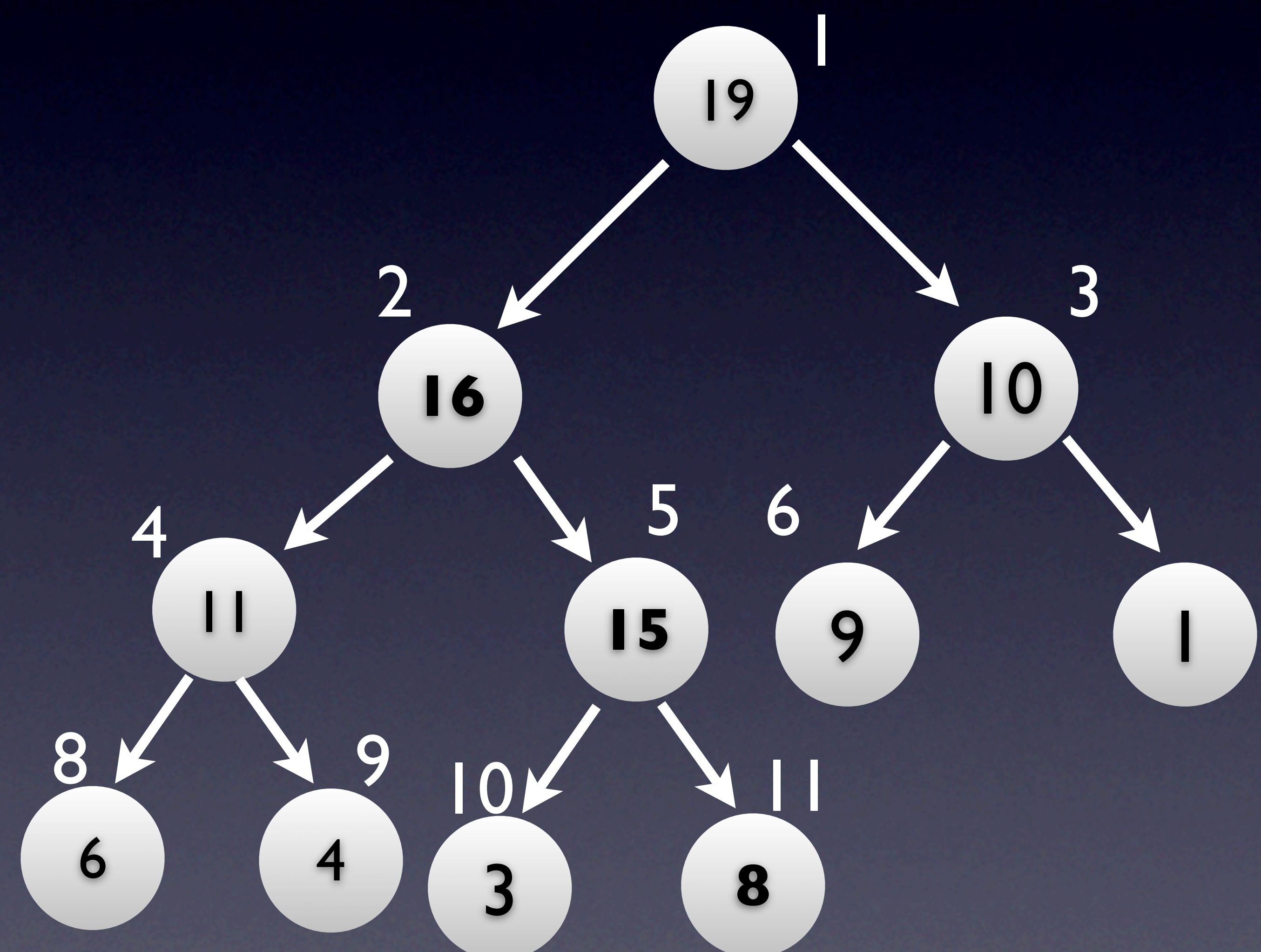
Messing with Heaps II: One More Node

- Can always insert -∞ at the end of the heap
- Max-Heap will not be violated
  - Can only add to the end, otherwise we wouldn't get an (almost) complete binary tree



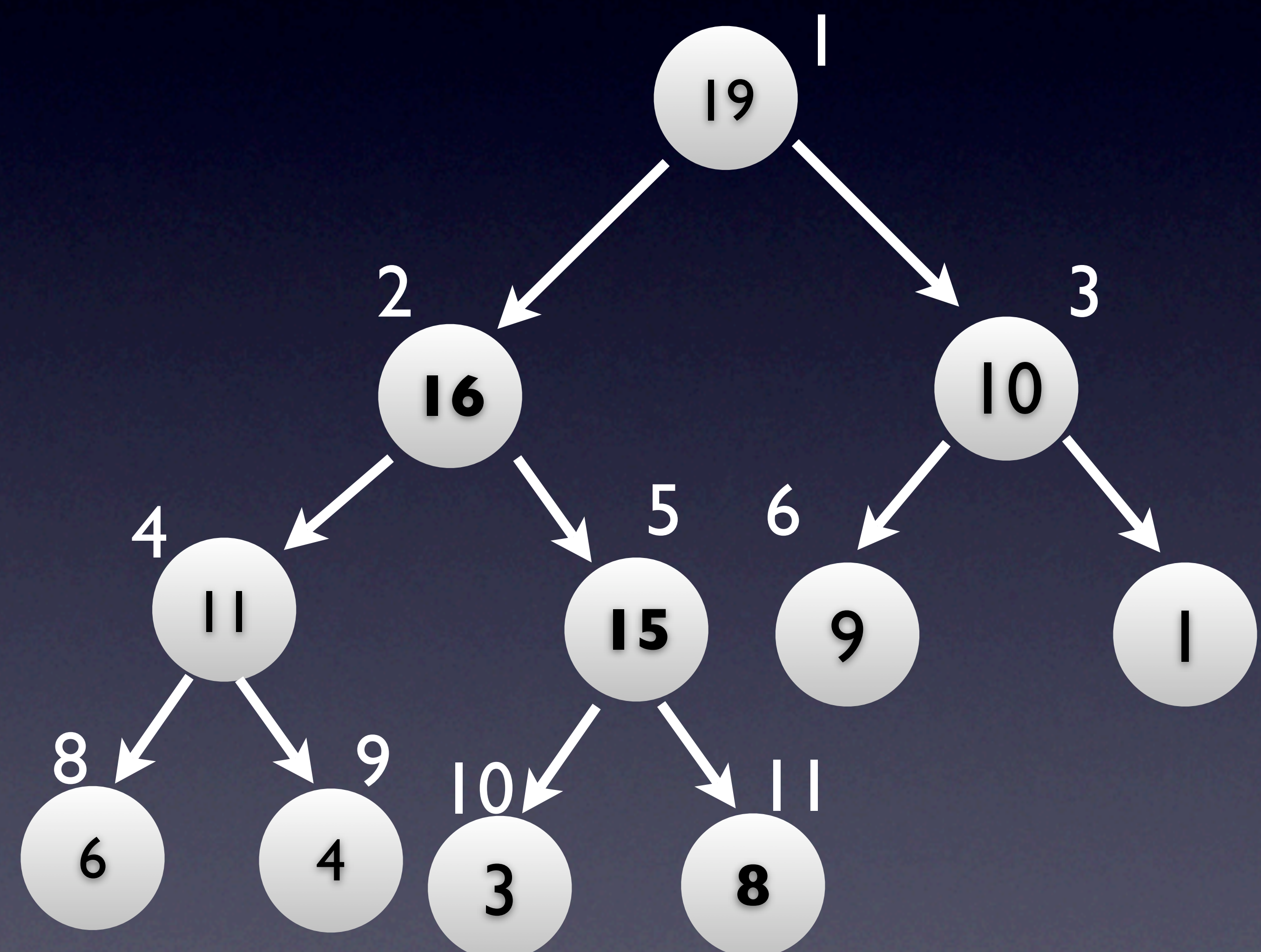
Messing with Heaps II: One More Node

- Insert any key
  - insert -∞ at the end of the heap
  - change node's key to desired key
  - sift



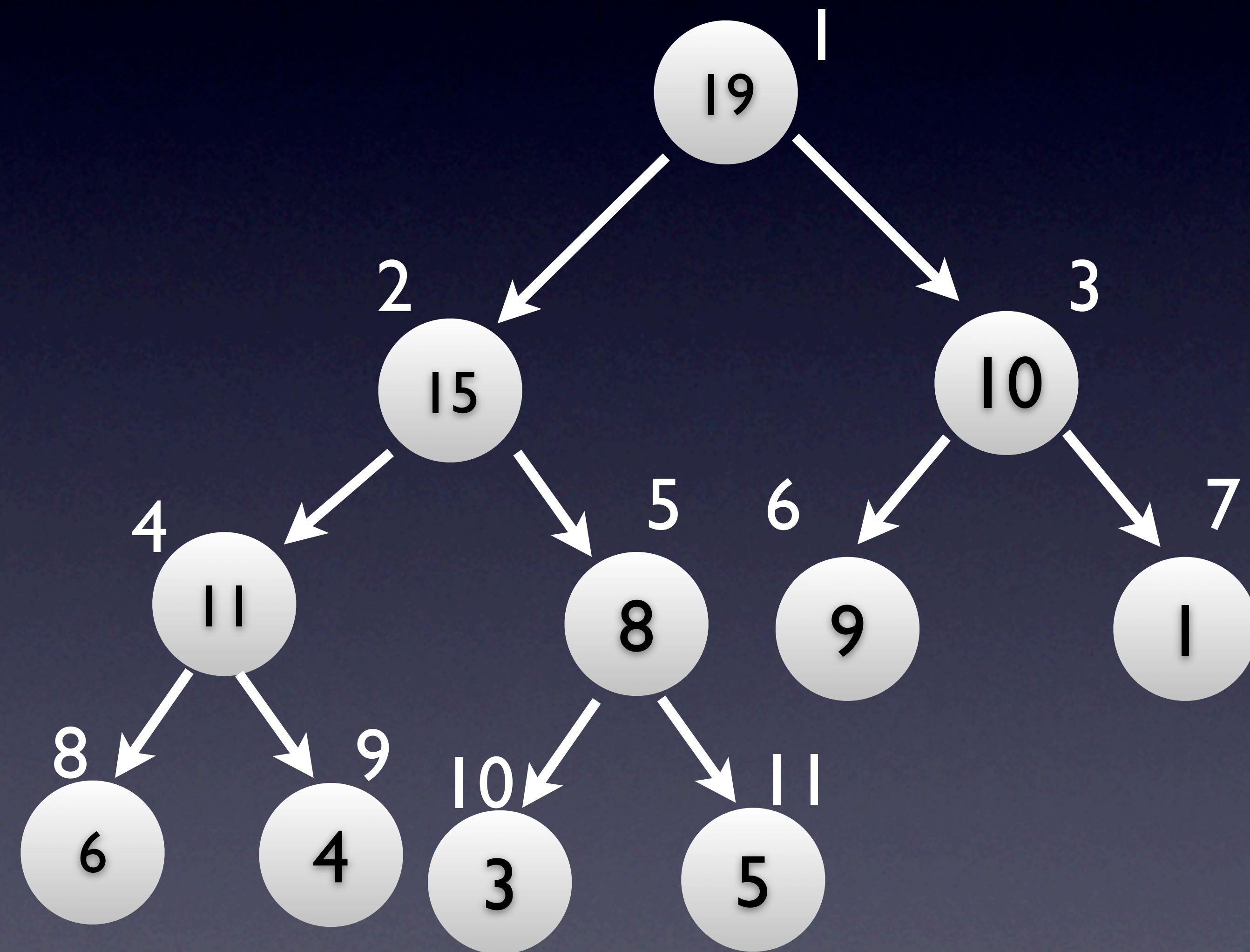
Messing with Heaps II: One More Node

- Insertion cost
  - insert -∞ at the end of the heap - O(1)
  - change node's key to new key - O(1)
  - sift - O(log(N))
- Total cost: O(log(N))



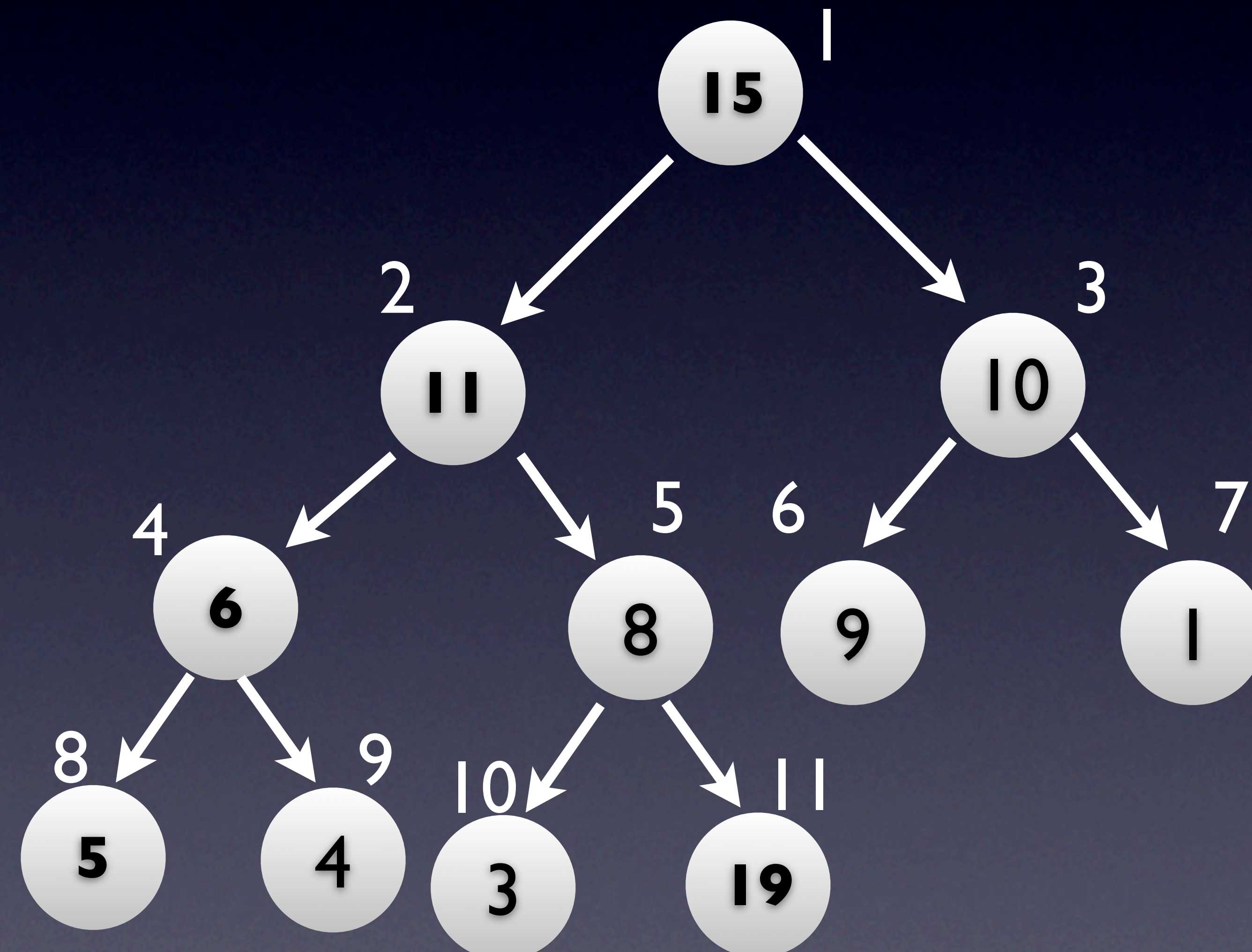
Messing with Heaps II: One ~~More~~ Less Node

- Can always delete last node
- Max-Heap will not be violated
  - It must be the last node, otherwise the binary tree won't be (almost) complete



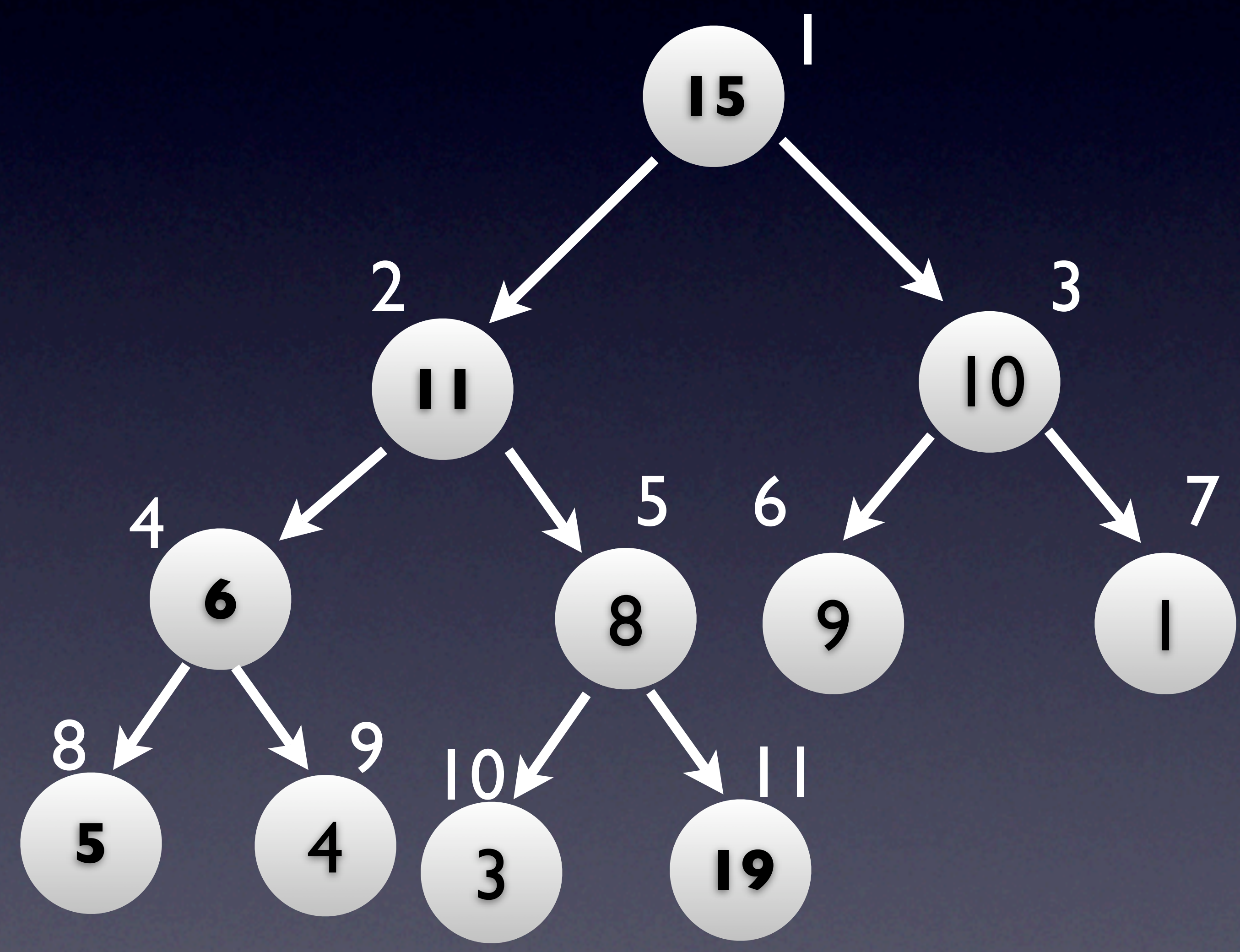
Messing with Heaps II: One ~~More~~ Less Node

- Deleting root
    - Replace root key with last key
    - Delete last node
    - Percolate



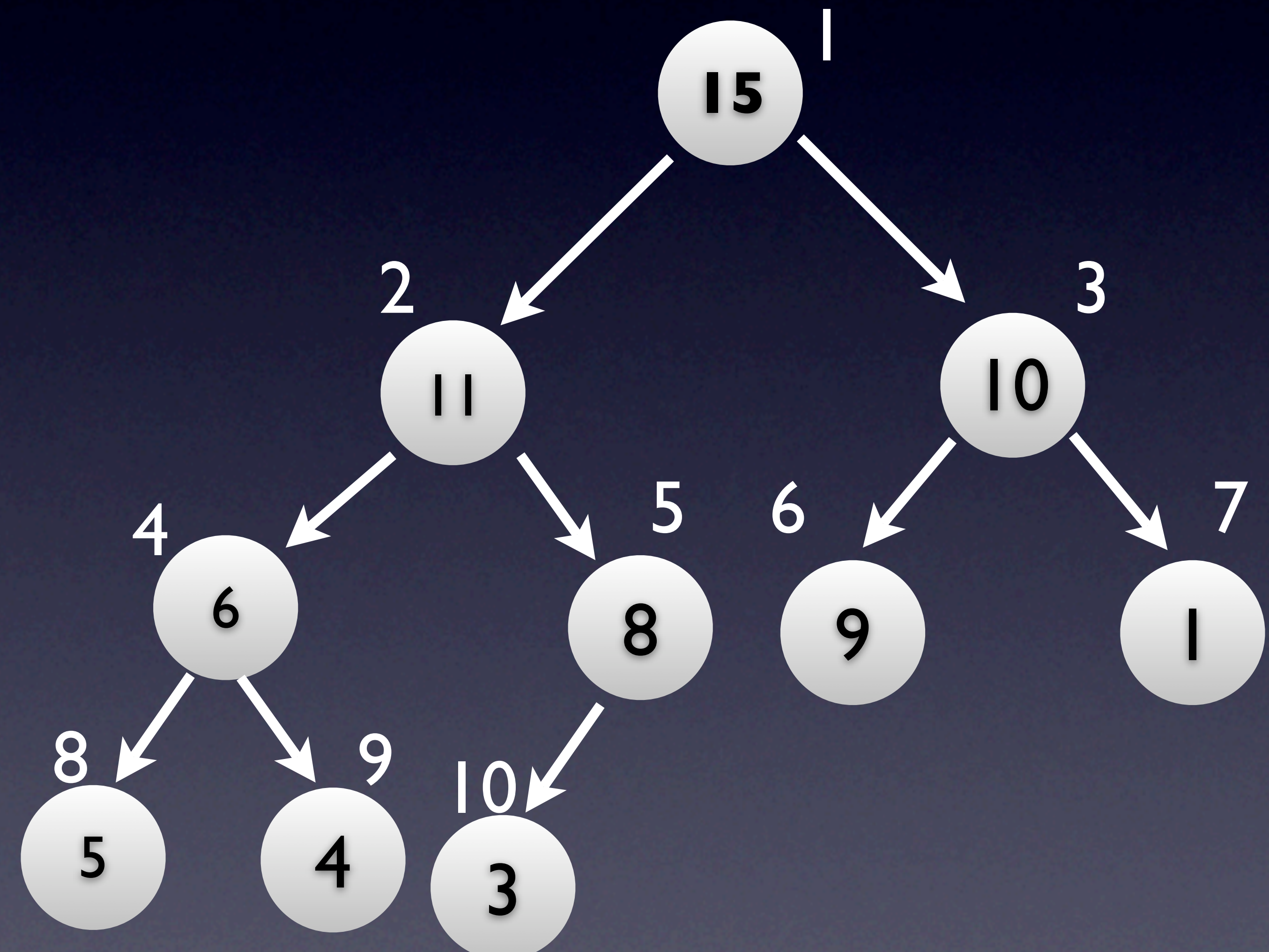
Messing with Heaps II: One ~~More~~ Less Node

- Deleting root cost
  - Replace root key with last key - O(1)
  - Delete last - O(1)
  - Percolate - O(log(N))
- Total cost: O(log(N))



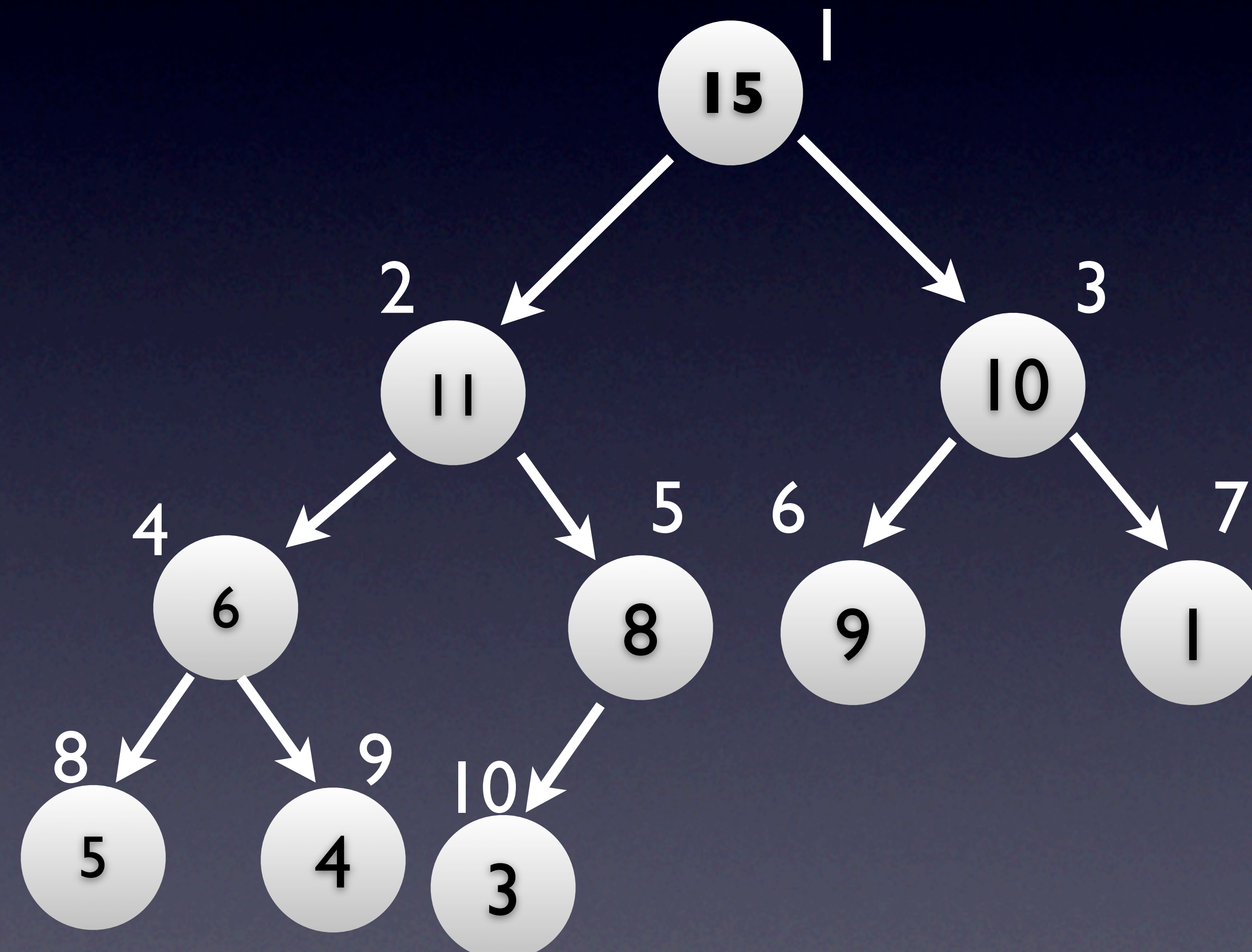
Messing with Heaps II: One ~~More~~ Less Node

- Deleting any node
  - Change key to +∞
  - Sift
  - Delete root



Messing with Heaps II: One ~~More~~ Less Node

- Deletion cost
  - Change key to +∞ - O(1)
  - Sift - O(log(N))
  - Remove root - O(log(N))
- Total cost: O(log(N))



Heap-Sort: Everything Falls Into Place

- Start with empty heap
- Build the heap: insert a[0] ... a[N-1]
- Build the result: delete root until heap is empty, gets keys sorted in reverse order
- Use a to store both the array and the heap (explained in lecture)

Heap-Sort: Slightly Faster

- Build the heap faster: Max-Heapify
  - Explained in lecture
  - $O(N)$ instead of $O(N \cdot \log(N))$
- Total time for Heap-Sort stays $O(N \cdot \log(N))$ because of N deletions
- Max-Heapify is very useful later

Priority Queues

- Data Structure
  - **insert**(key) : adds to the queue
  - **max**() : returns the maximum key
  - **delete-max**() : deletes the max key
  - **delete**(key) : deletes the given key
    - optional (only needed in some apps)

Priority Queues with Max-Heaps

- Doh? (assuming you paid attention so far)
- Costs (see above line for explanations)
  - insert: O(log(N))
  - max: O(1)
  - delete-max: O(log(N))
  - delete: O(log(N)) - only if given the index of the node containing the key

Cool / Smart Problem

- Given an array **a** of numbers, extract the **k** largest numbers
- Want good running time for any **k**

Cool / Smart Problem

- Small cases:
  - k = 1: scan through the array, find N
  - k small
    - try to scale the scan
    - getting to O(kN), not good

Cool / Smart Problem

- Solution: Heaps!
  - build heap with Max-Heapify
  - delete root k times
  - $O(k \cdot \log(N))$
- Bonus Solution: Selection Trees (we'll come back to this if we have time)

Discussion: Priority Queue Algorithms

- BSTs
  - store keys in a BST
- Regular Arrays
  - store keys in an array
- Arrays of Buckets
  - a[k] stores a list of keys with value k