ECS 32B - Priority Queues

Aaron Kaloti

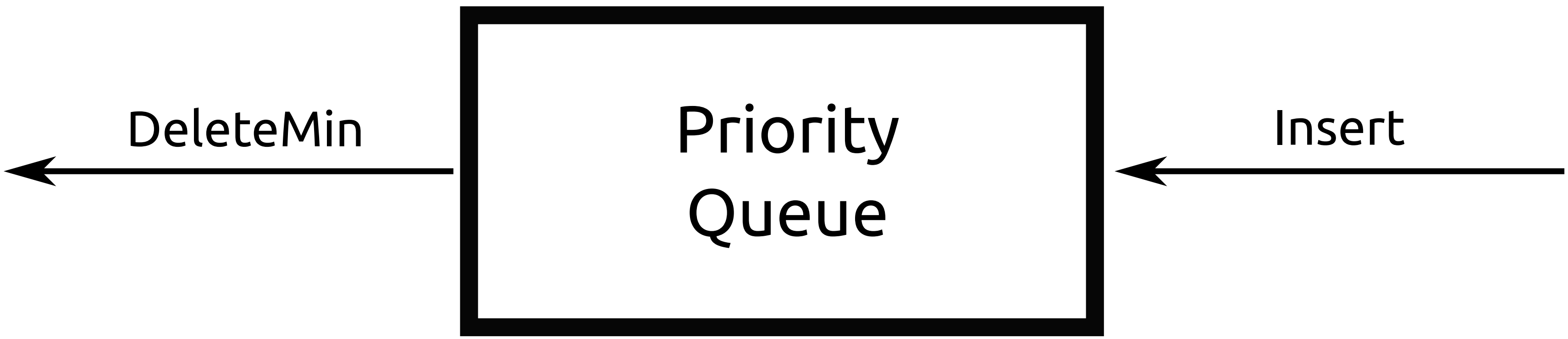
UC Davis - Summer Session #2 2020

Motivation

- Printer jobs.
- Operating system job scheduling.

Priority Queue

- **priority queue**: ADT requiring these two operations:
  - Insert.
  - DeleteMin: retrieve and delete element with smallest key.



Simple Implementations

Unordered Linked List

- Insert: constant time (at front).
- DeleteMin: linear time.

Ordered Linked List

- Insert: linear time.
- DeleteMin: constant time[1] (at front).

1. This *does* take constant time. During lecture, I accidentally talked as if it were an ordered Python list instead of an ordered linked list.

Other Implementations

Binary Search Tree

- Insert: linear time.
  - Average: logarithmic time.
- DeleteMin: linear time.
  - Average: logarithmic time.

Self-Balancing BST

- Insert: logarithmic time.
- DeleteMin: logarithmic time.
  - Peeking at min: also logarithmic time.

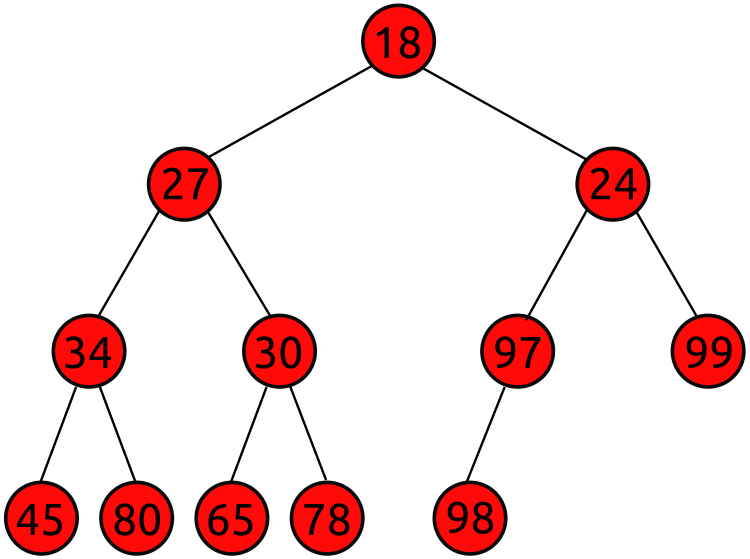
Binary Heap

- **Preferred implementation for priority queue.**
- Insert: logarithmic time.
  - Average: constant time.
- DeleteMin: logarithmic time.
  - Peeking at min: constant time.

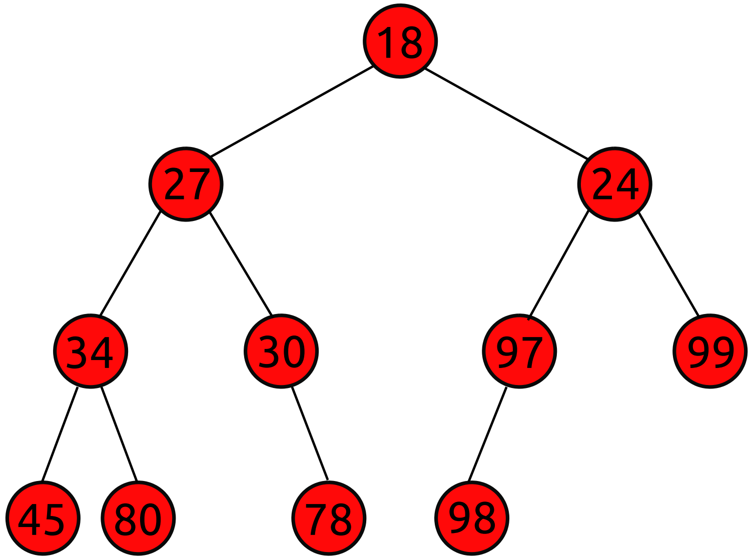
Binary Heap

- **binary heap**: binary tree with two properties:
  1. *structure property*: complete binary tree (completely filled, except possibly the last level, which is filled left to right).
  2. *heap-order property*: each node's key is less than its children's keys.
     - **min heap**: root is smallest element. (**max heap** reverses heap-order property.)
- Basic operations: insert, delete root (min if min heap).

Example



Nonexample



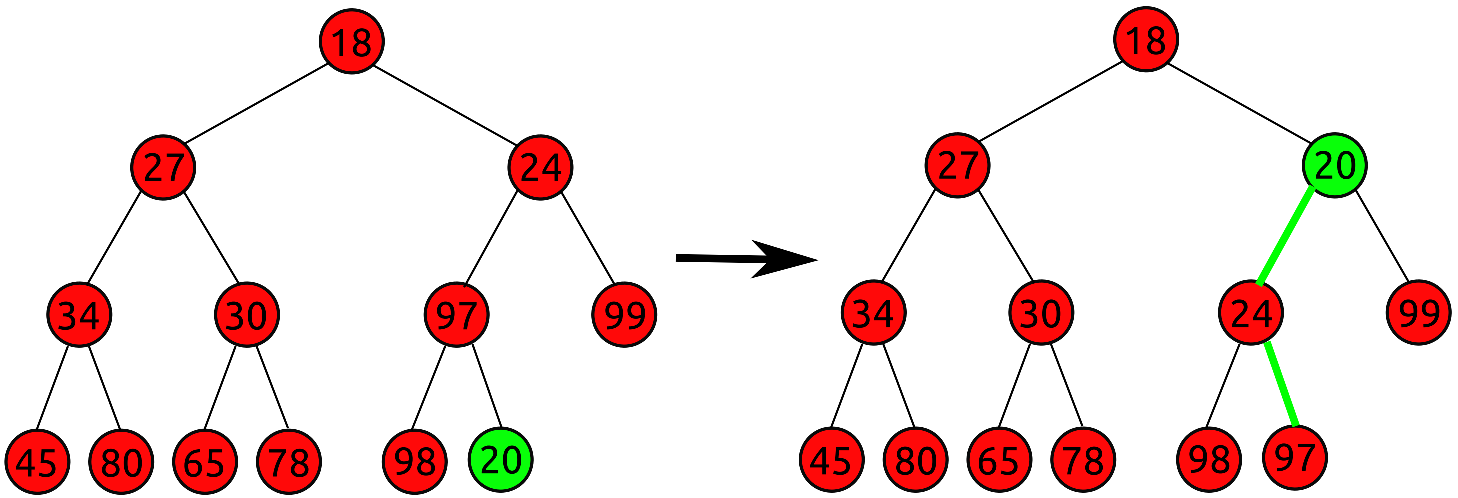
Binary Heap

Insert

1. Insert new element in correct spot to maintain *structure property*.
2. Move the element up until *heap-order property* restored (**percolate up**).

Example

- Insert 20.



Analysis

- $d + 1$ assignments, where inserted node is percolated up d times.
- Worst-case time complexity: $\Theta(\lg n)$ time (inserted node becomes root).
- Average-case time complexity: constant time[1].

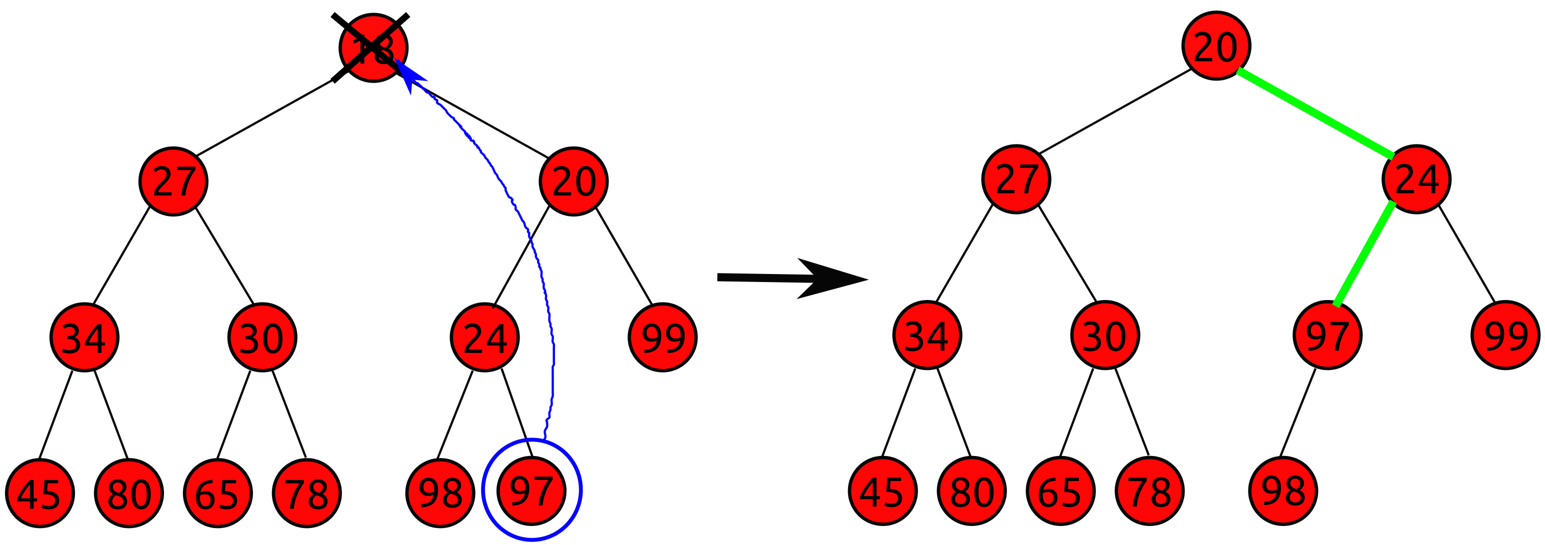
1. From p.251 of Weiss' book: "[I]t has been shown that 2.607 comparisons are required on average to perform an insert, so the average insert moves an element up 1.607 levels."

Binary Heap

Delete

1. Remove root.
2. Move rightmost leaf to be root, to maintain *structure property*.
3. Move this new root down (by swapping) until *heap-order property* restored (**percolate down**).

Example



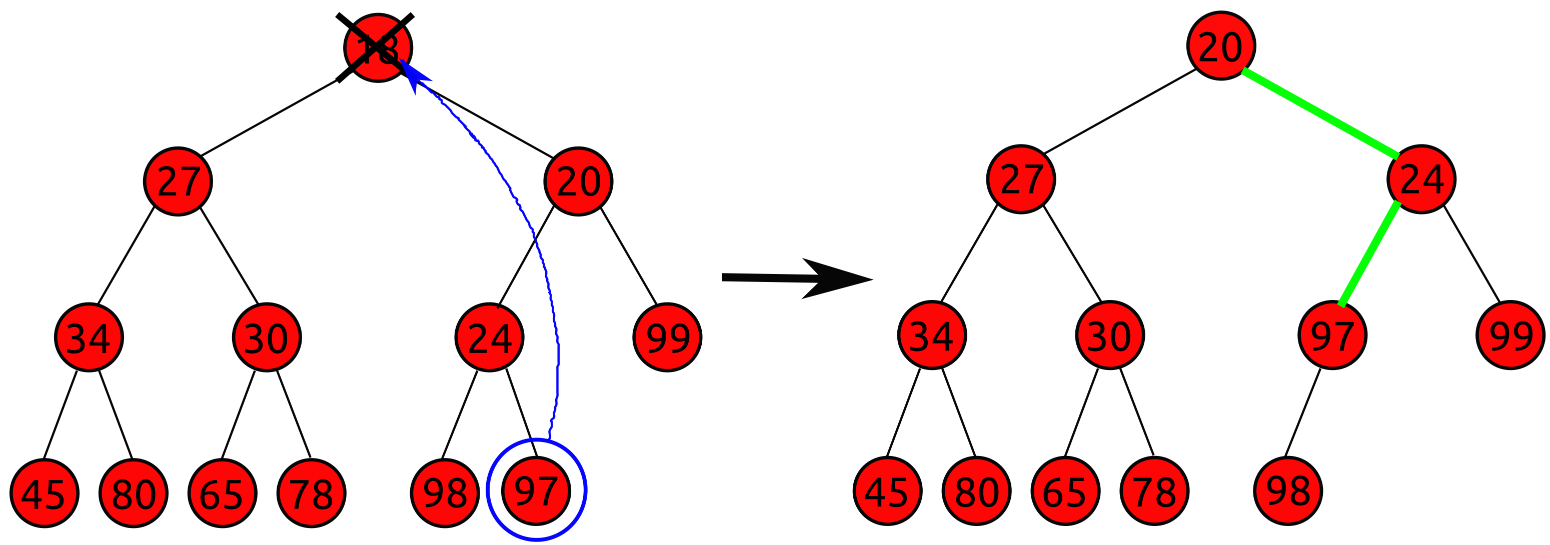
- For the first swap, could 97 be swapped with 27 instead of with 20?

Binary Heap

Delete

1. Remove root.
2. Move rightmost leaf to be root, to maintain *structure property*.
3. Move this new root down (by swapping) until *heap-order property* restored (**percolate down**).

Example



- No; would violate *heap-order property*. Always swap with child that has smaller key.

Analysis

- Worst-case time complexity: $\Theta(\lg n)$.
- Average-case time complexity: $\Theta(\lg n)$.
    - Percolating down a node that was previously at bottom.

Binary Heap

Representation: Python List

- Works because of rigid structure of binary heap (complete binary tree).

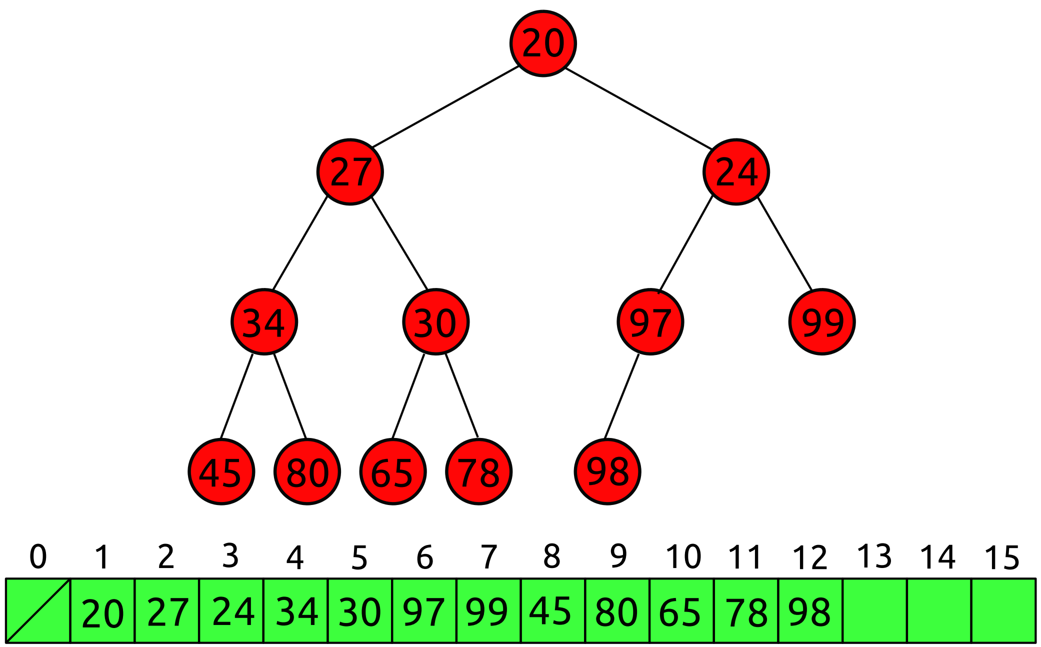
Pros/Cons

- Space-efficient (needn't waste space on the links as would with a typical tree implementation).
- Requires estimate of maximum heap size (can resize).

Parent/Children

- For node at index i:
  - Parent is at index $\frac{i}{2}$ (truncate).
  - Left child is at index $2i$.
  - Right child is at index $2i + 1$.

Example



Priority Queue ADT

Extended API (for Priority Queue P)

- If have additional data structure (e.g. hash table) to track position of each key/node in the heap (the underlying Python list), these extended operations take $\Theta(\lg N)$ time:
  - $DecreaseKey(k, change)$
  - $IncreaseKey(k, change)$
  - $Remove(k)$

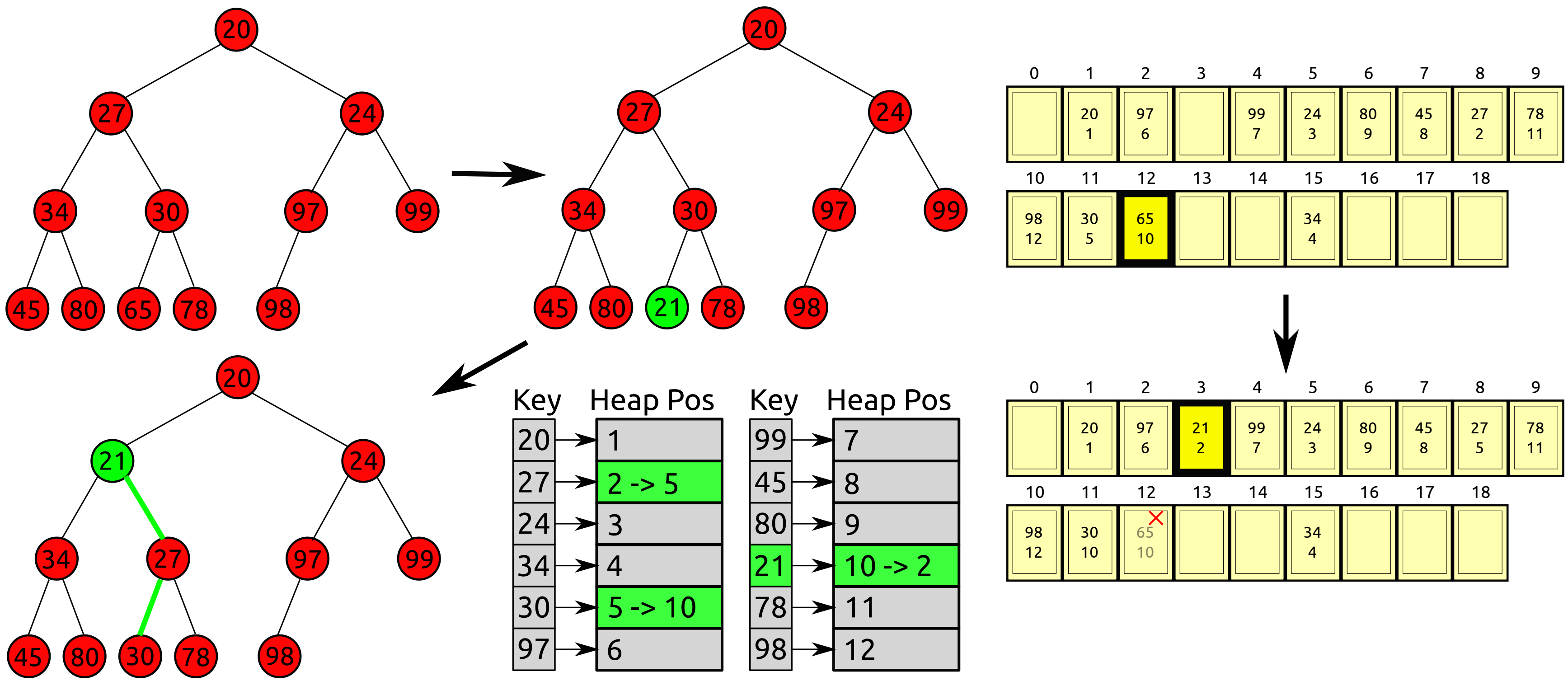
Priority Queue ADT

$DecreaseKey(k, change)$[1]

- Decreases the key of the target node by a specified amount.

Example[2]

1. Find the node using the additional data structure.
2. Decrease the key as desired.
3. Percolate up as long as necessary.



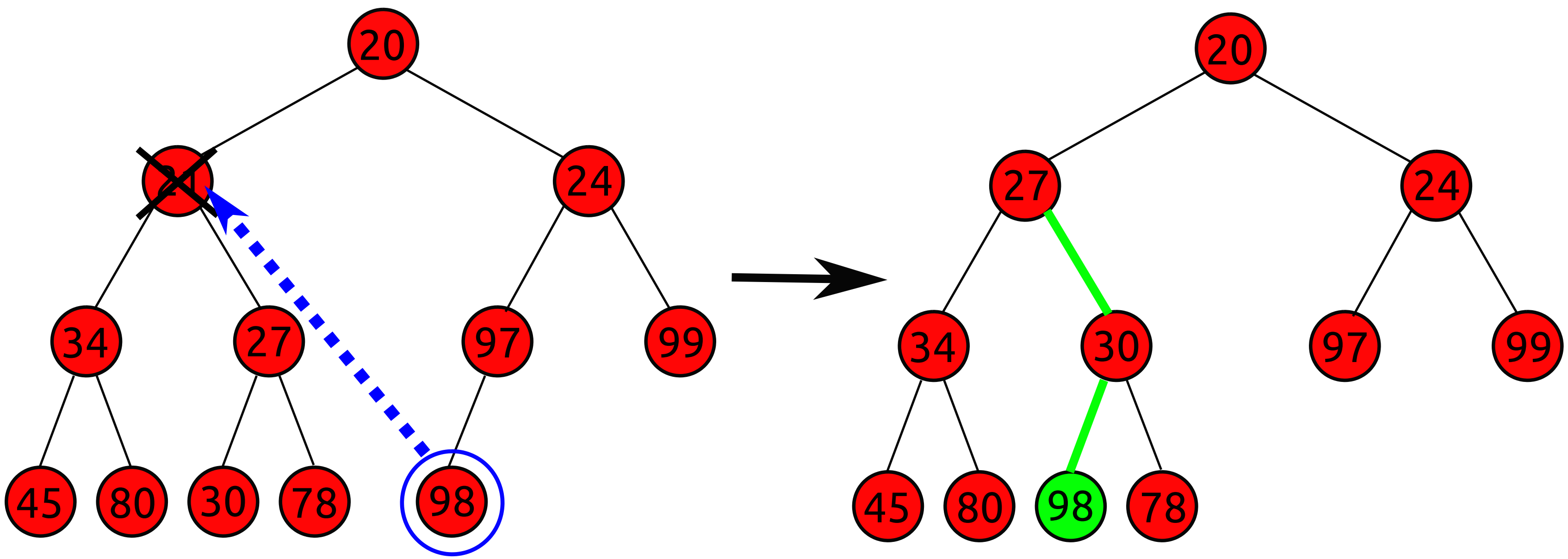
1. IncreaseKey is very similar.
2. The order in which the keys appear in the bottom-right of the image is unimportant (it should be a hash table anyways, not a map).

Priority Queue ADT

Remove(k) (any element, not necessarily minimum)[1]

1. Find the node using the additional data structure.
2. Remove the node.
3. Replace the node with the rightmost leaf, to maintain *structure property*.
4. Percolate up or down.

Example



1. In the book by Weiss, they implement $Remove(P, k)$ by doing $DecreaseKey(P, k, \infty)$ (i.e. forcing k to become the root) and then doing $ExtractMin(P)$. This needlessly takes twice the amount of time but doesn't change the big-O.

Heap Construction

- **Goal**: Create a heap out of n given items (given in a Python list).

Naive Method

- Insert each item.
- Worst-case time complexity: $\Theta(n \lg n)$.
- Average-case time complexity: $\Theta(n)$.

Build Heap

- Worst-case time complexity: $\Theta(n)$ time.
  - We won't prove here.
- Steps:
  1. Treat list as a heap.
  2. For each level upwards (starting at second-to-lowest level), percolate each node down.

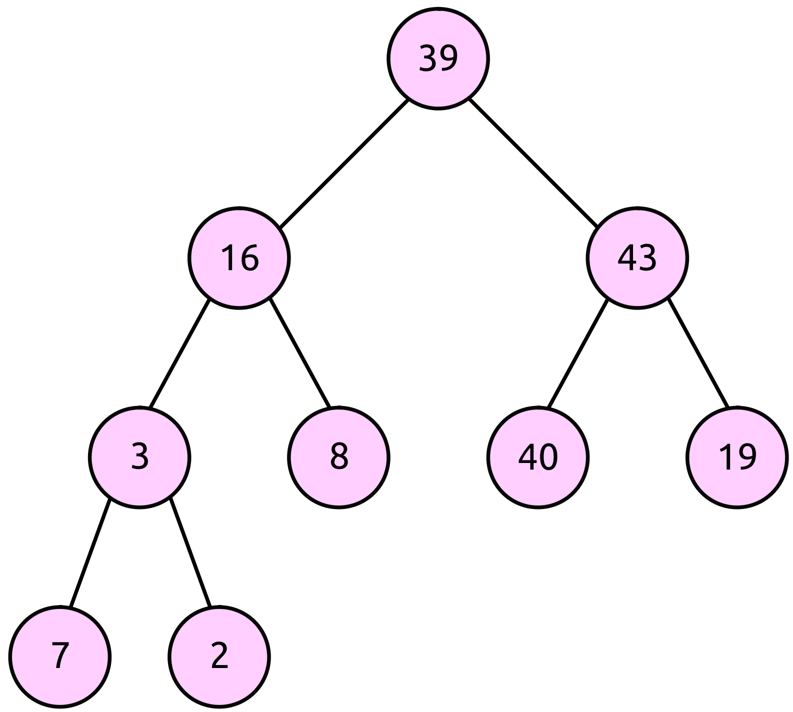
Heap Construction

Build Heap

Example

0	1	2	3	4	5	6	7	8
39	16	43	3	8	40	19	7	2

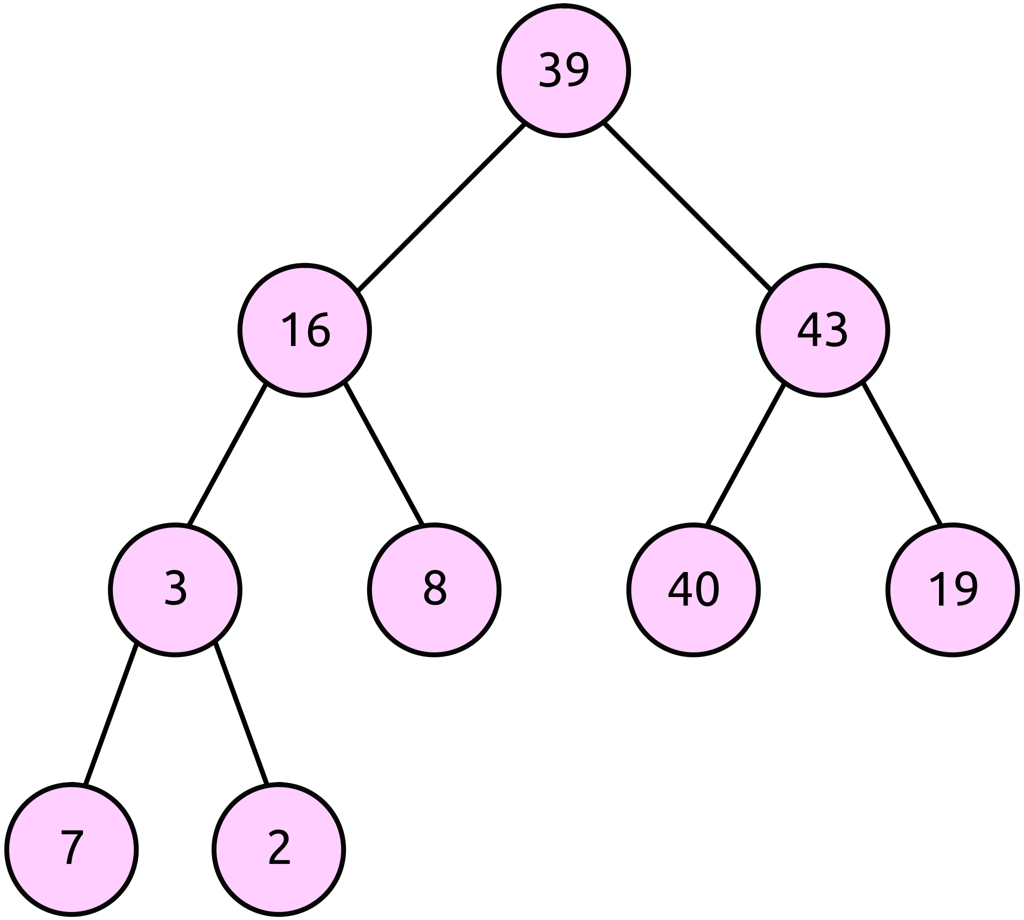
0	1	2	3	4	5	6	7	8	9
	39	16	43	3	8	40	19	7	2



Heap Construction

Build Heap

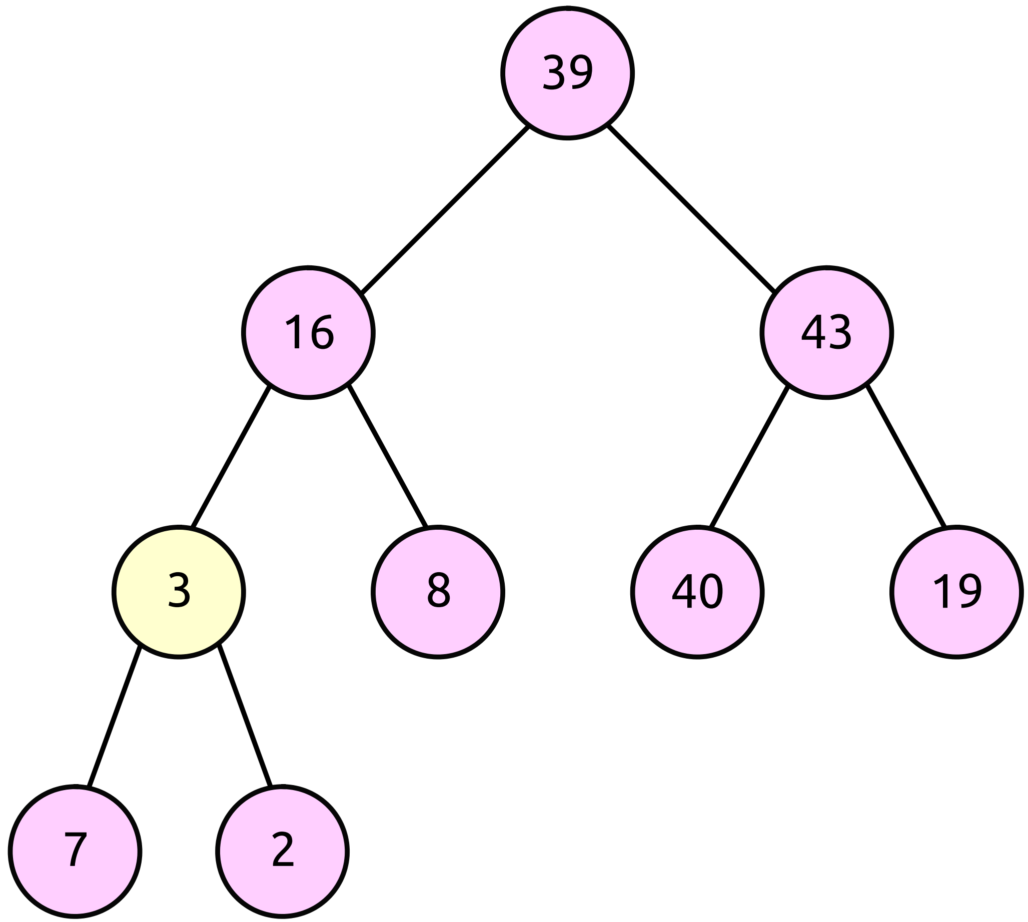
Example



Heap Construction

Build Heap

Example

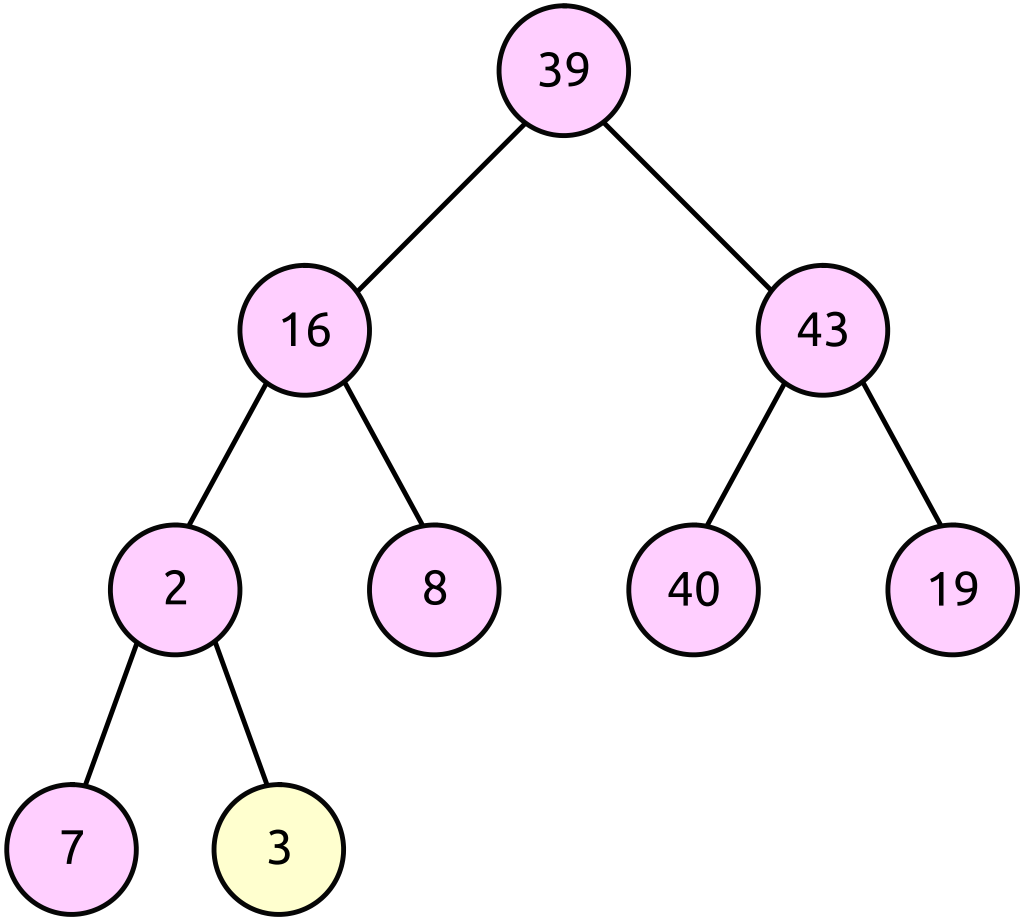


- Need to percolate 3 down.

Heap Construction

Build Heap

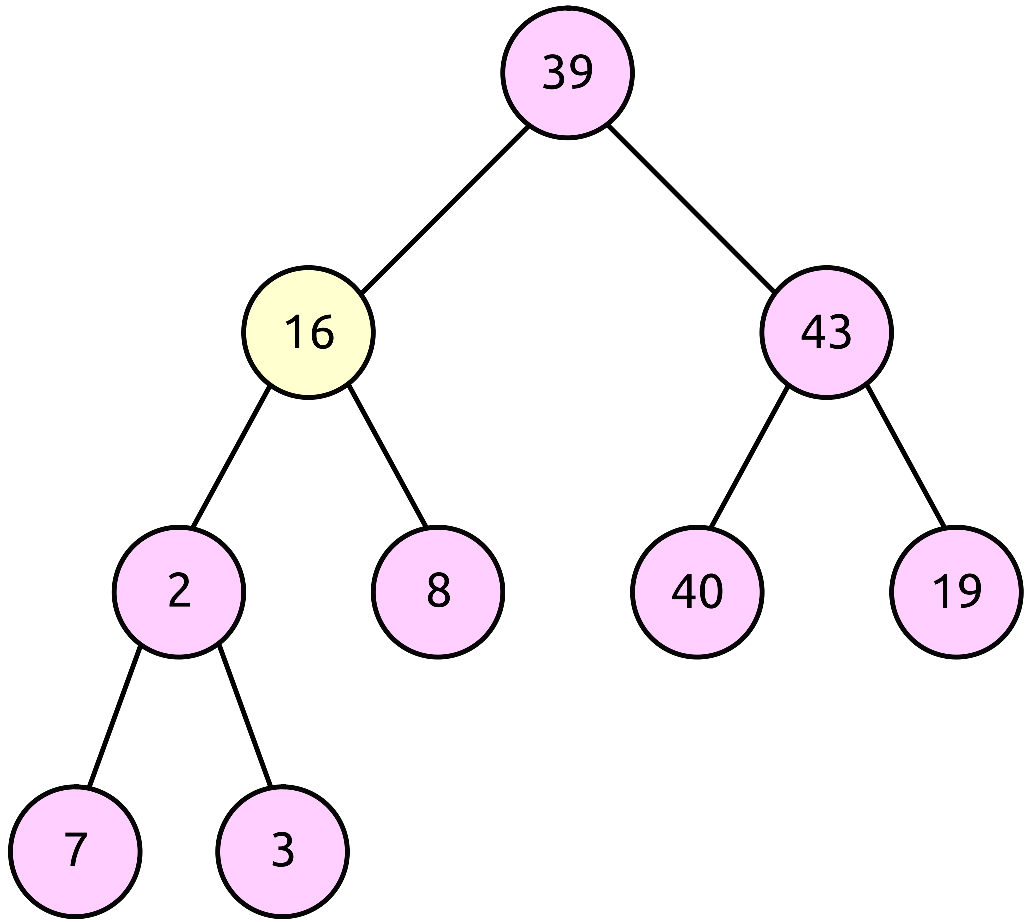
Example



Heap Construction

Build Heap

Example

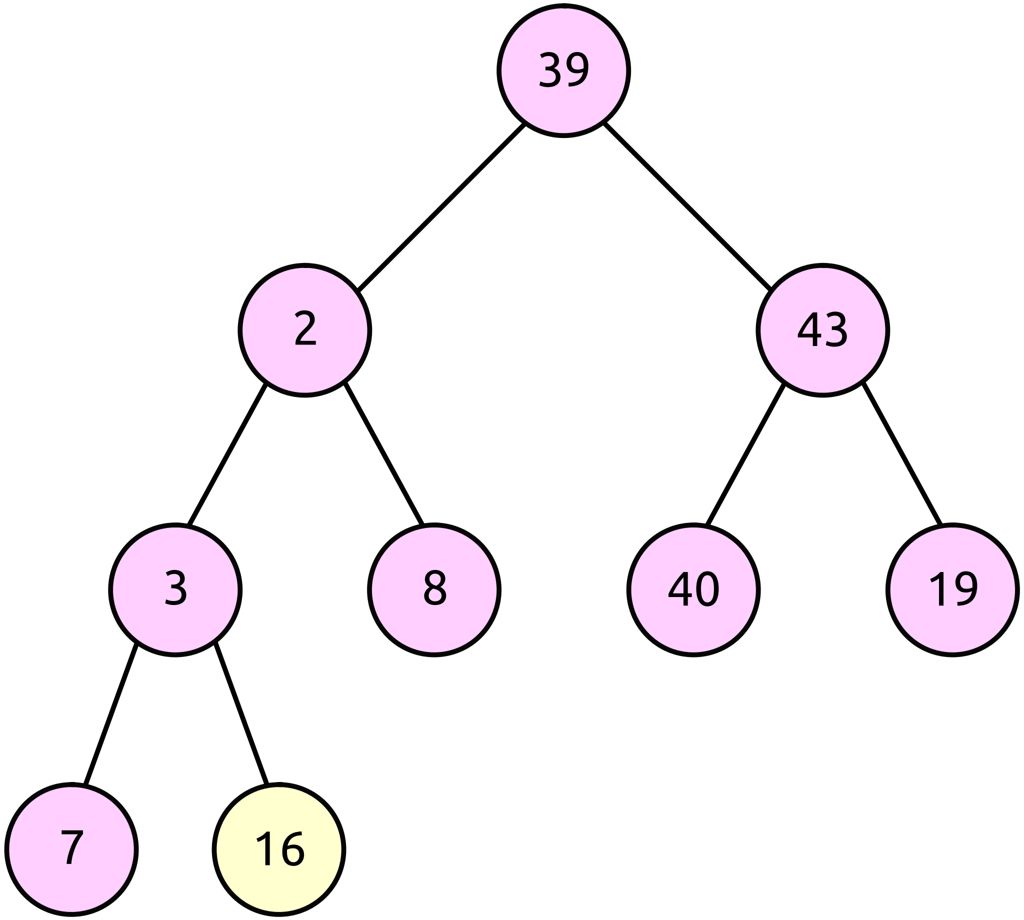


- 8, 40, and 19 are leaves, so skip them.
- Percolate 16 down.

Heap Construction

Build Heap

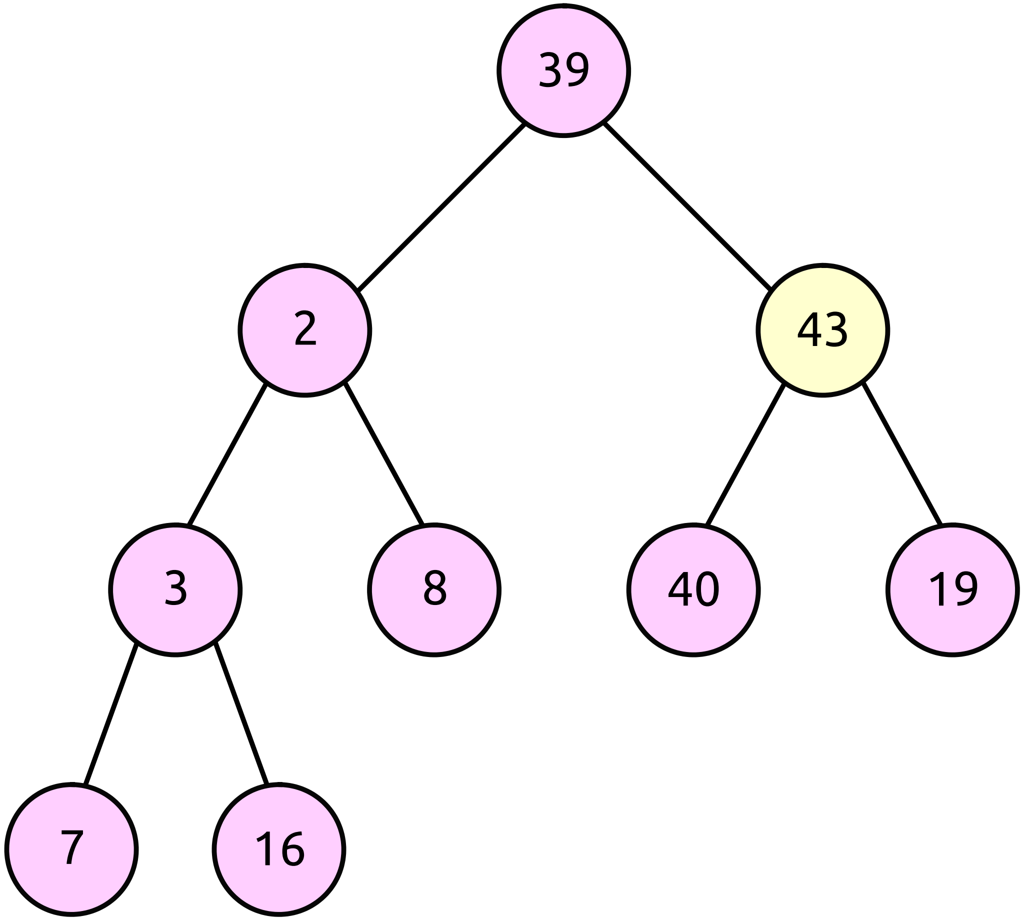
Example



Heap Construction

Build Heap

Example

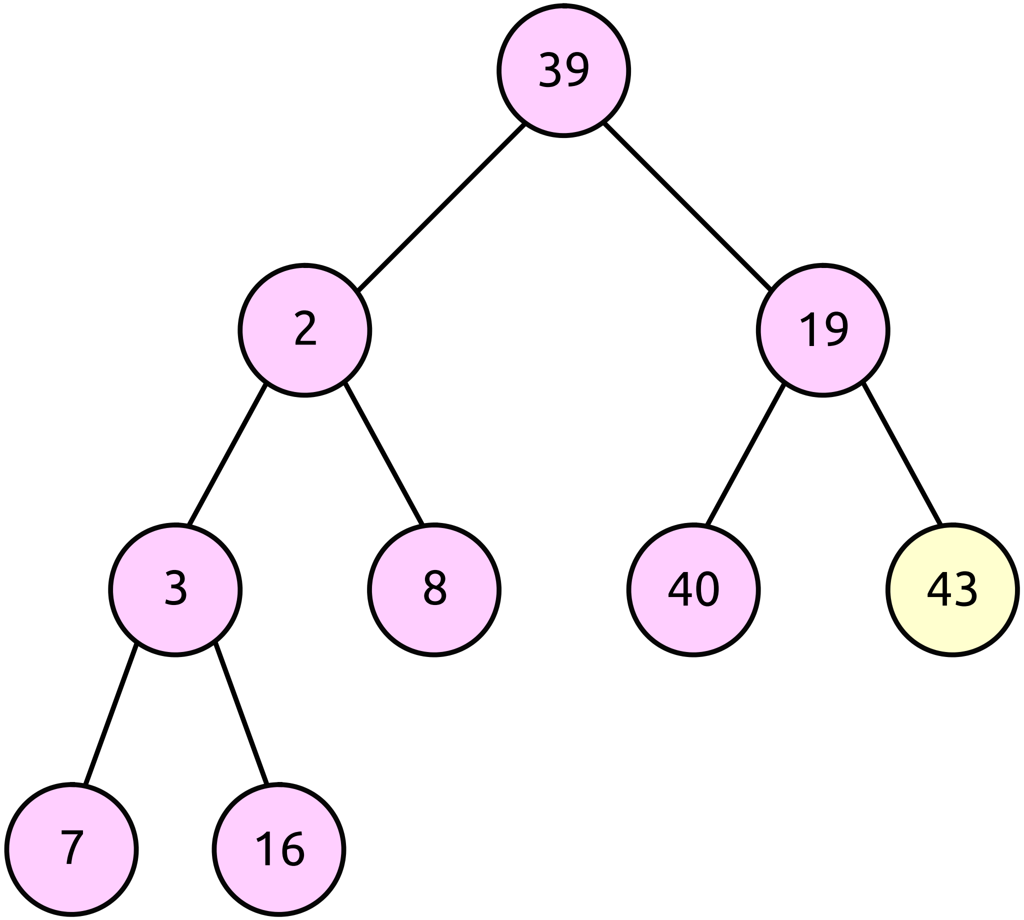


- Percolate 43 down.

Heap Construction

Build Heap

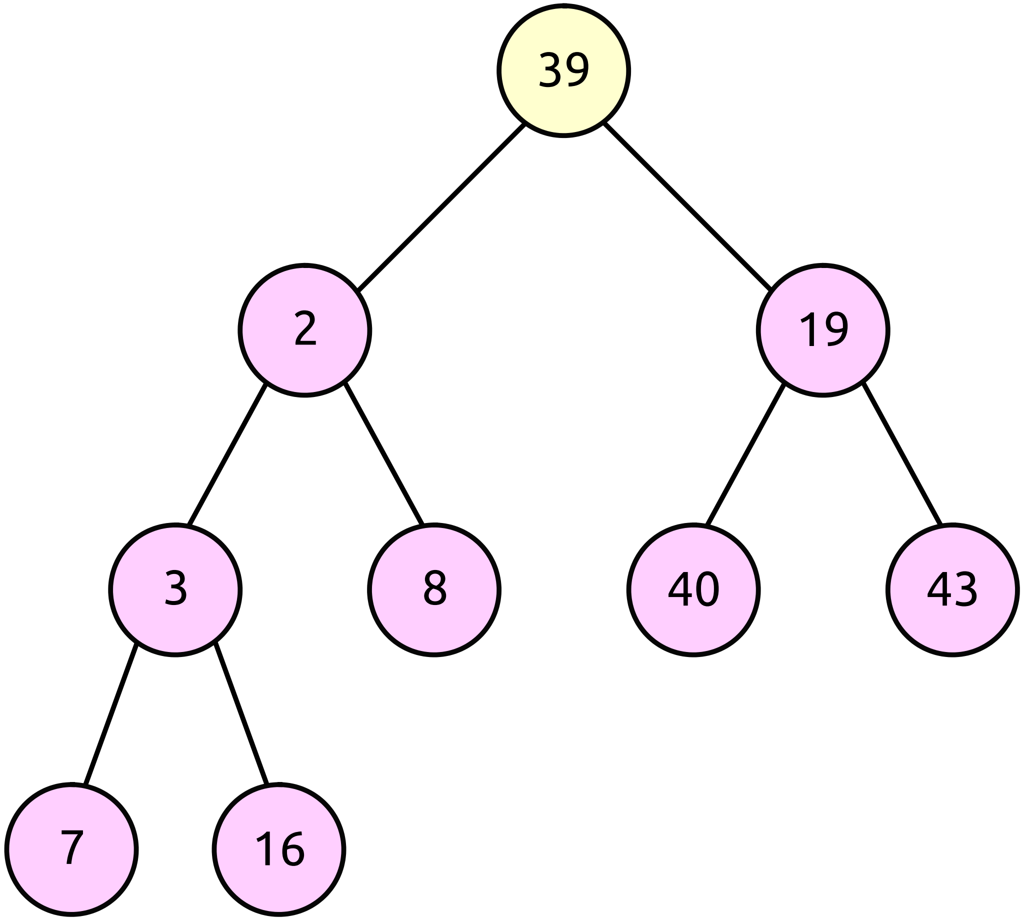
Example



Heap Construction

Build Heap

Example

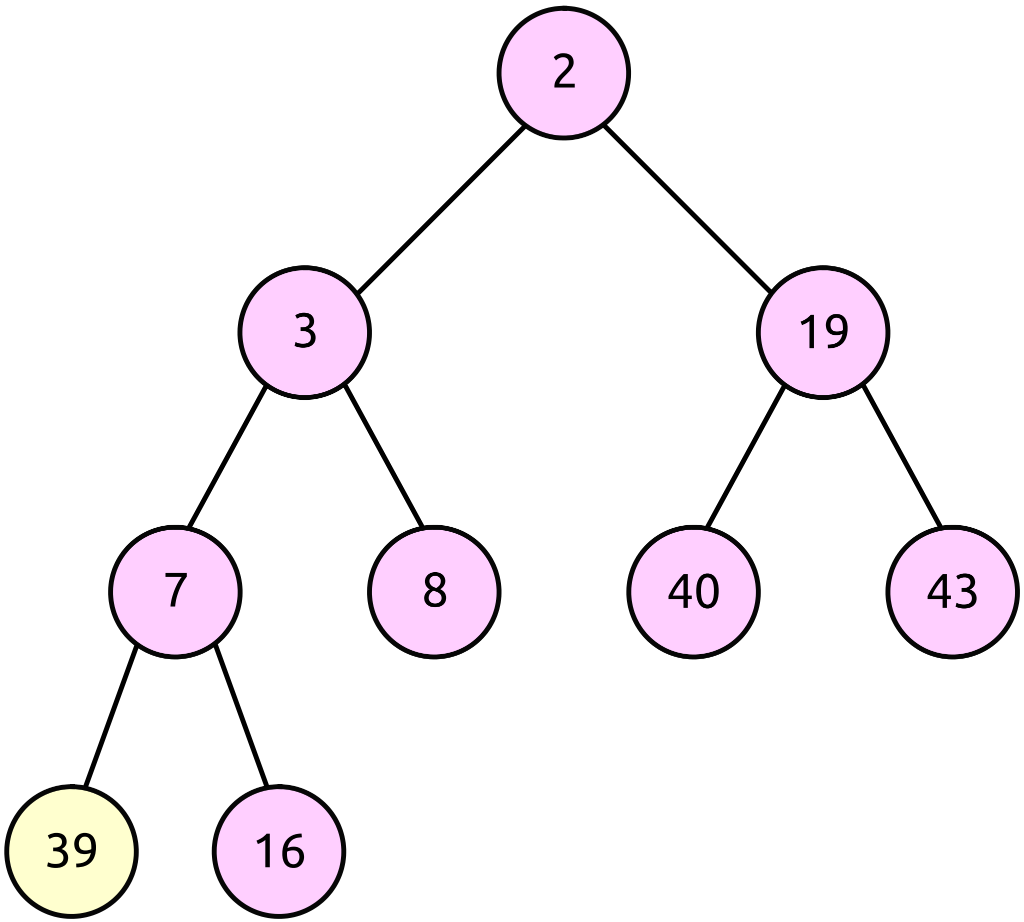


- Percolate 39 down.

Heap Construction

Build Heap

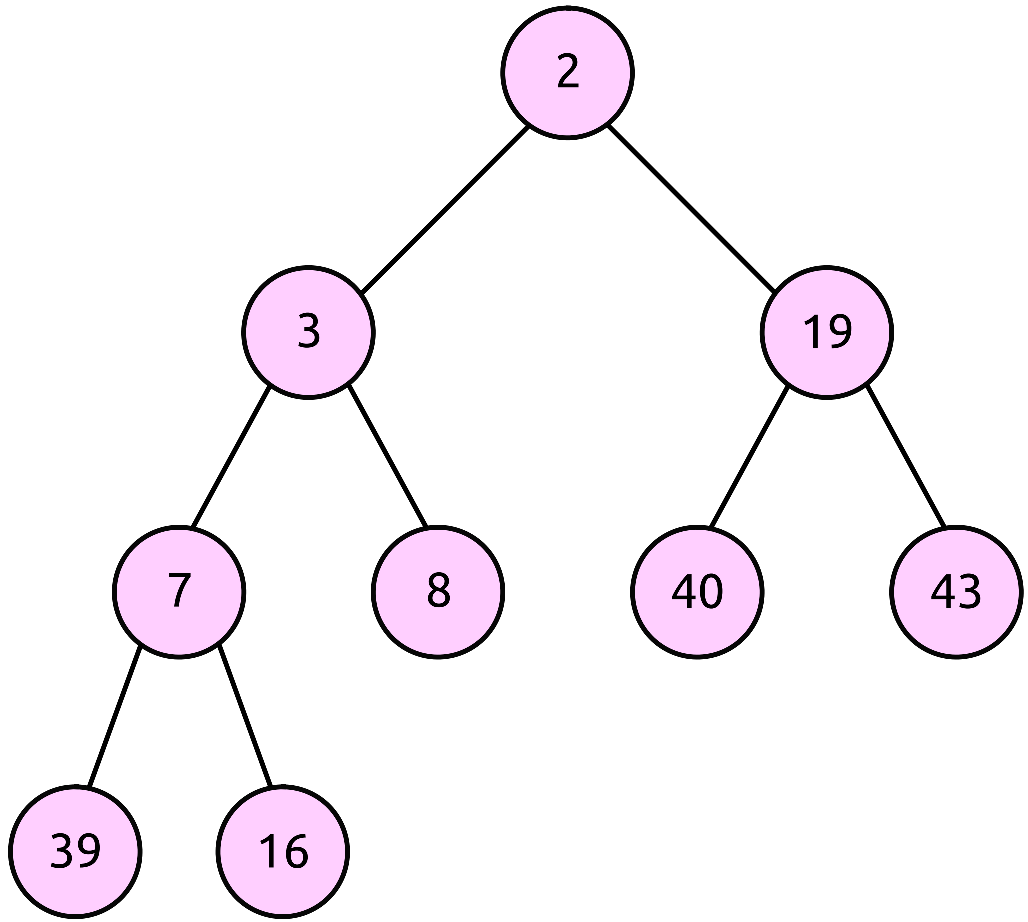
Example



Heap Construction

Build Heap

Example



- Done.

References / Further Reading

- Chapter 6 of *Data Structures and Algorithm Analysis in C*++ by Mark Allen Weiss (Fourth Edition).
- Sections 7.8-7.10 of *Problem Solving with Algorithms and Data Structures using Python* by Brad Miller and David Ranum.