CSE 332 Autumn 2026
Lecture 12: Sorting, pt. 1

Ruth Anderson

http://www.cs.uw.edu/332

Announcements

- “Meet the staff” activity – Come meet us in office hours!
- Participate in Section, earn a late day (see Ed Board post)
- Exercise 4 AVL Trees: Due Friday Oct 30
- Exercise 5 Hashing: Due Friday Nov 6
- Exercise 6 Sorting: coming later this week, due Nov 13
- Midterm: 5-6pm Monday November 9
  - Review during both lecture sections on Fri Nov 6

Sorting

- Rearrangement of items into some defined sequence
  - Usually: reordering a list from smallest to largest according to some metric
- Why sort things?
  - Enable things like binary search
    - It makes some algorithms faster
  - Nicer for human algorithms too

More Formal Definition

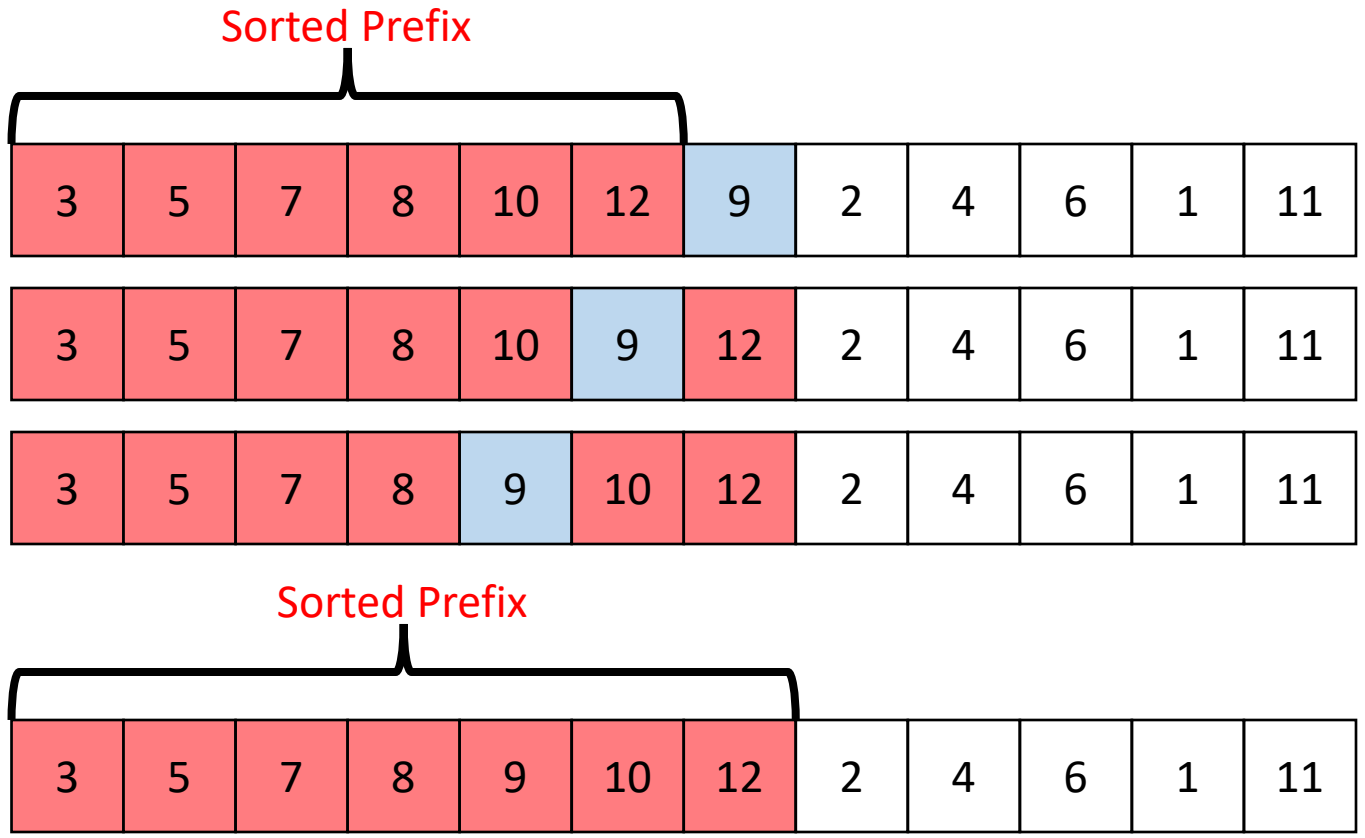
- Input:
  - An array A of items
  - A comparison function for these items
    - Given two items x and y, we can determine whether $x < y$, $x > y$, or $x = y$
- Output:
  - A permutation of A such that if $i \leq j$ then $A[i] \leq A[j]$
  - Permutation: a sequence of the same items but perhaps in a different order

Sorting “Landscape”

- There is no singular best algorithm for sorting
- Some are faster, some are slower
- Some use more memory, some use less
- Some are super extra fast if your data matches particular assumptions
- Some have other special properties that make them valuable
- No sorting algorithm can have only all the “best” attributes

Insertion Sort

- Idea: Maintain a sorted list prefix, extend that prefix by “inserting” the next element



Insertion Sort - Summary

- If the items at index 0 and 1 are out of order, swap them
- Keep swapping the item at index 2 with the thing to its left as long as the left thing is larger
- …
- Keep swapping the item at index i with the thing to its left as long as the left thing is larger

```
for (i = 1; i < a.length; i++) {
    prev = i - 1;
    while (a[i] < a[prev] && prev > -1) {
        temp = a[i];
        a[i] = a[prev];
        a[prev] = a[i];
        i--;
        prev--;
    }
}
```

Running Time:

Worst Case:

Best Case:

10	77	5	15	2	22	64	41	18	19	30	21	3	24	23	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Insertion Sort – (Answers)

- If the items at index 0 and 1 are out of order, swap them
- Keep swapping the item at index 2 with the thing to its left as long as the left thing is larger
- …
- Keep swapping the item at index i with the thing to its left as long as the left thing is larger

```
for (i = 1; i < a.length; i++) {
    prev = i - 1;
    while (a[i] < a[prev] && prev > -1) {
        temp = a[i];
        a[i] = a[prev];
        a[prev] = a[i];
        i--;
        prev--;
    }
}
```

Running Time:

Worst Case: $\Theta(n^2)$

Best Case: $\Theta(n)$

10	77	5	15	2	22	64	41	18	19	30	21	3	24	23	33
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Aside: Bubble Sort – we won’t cover it

"the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems” –Donald Knuth, The Art of Computer Programming



Properties To Consider

- Worst case running time
- In place:
  - We only need to use the pre-existing array to do sorting
  - Constant extra space (only some additional variables needed)
- Adaptive
  - The running time improves as the given list is closer to being sorted
  - It should be linear time for a pre-sorted list, and nearly linear time if the list is nearly sorted
- Online
  - We can start sorting before we have the entire list.
- Stable
  - “Tied” elements keep their original order

Insertion Sort Properties

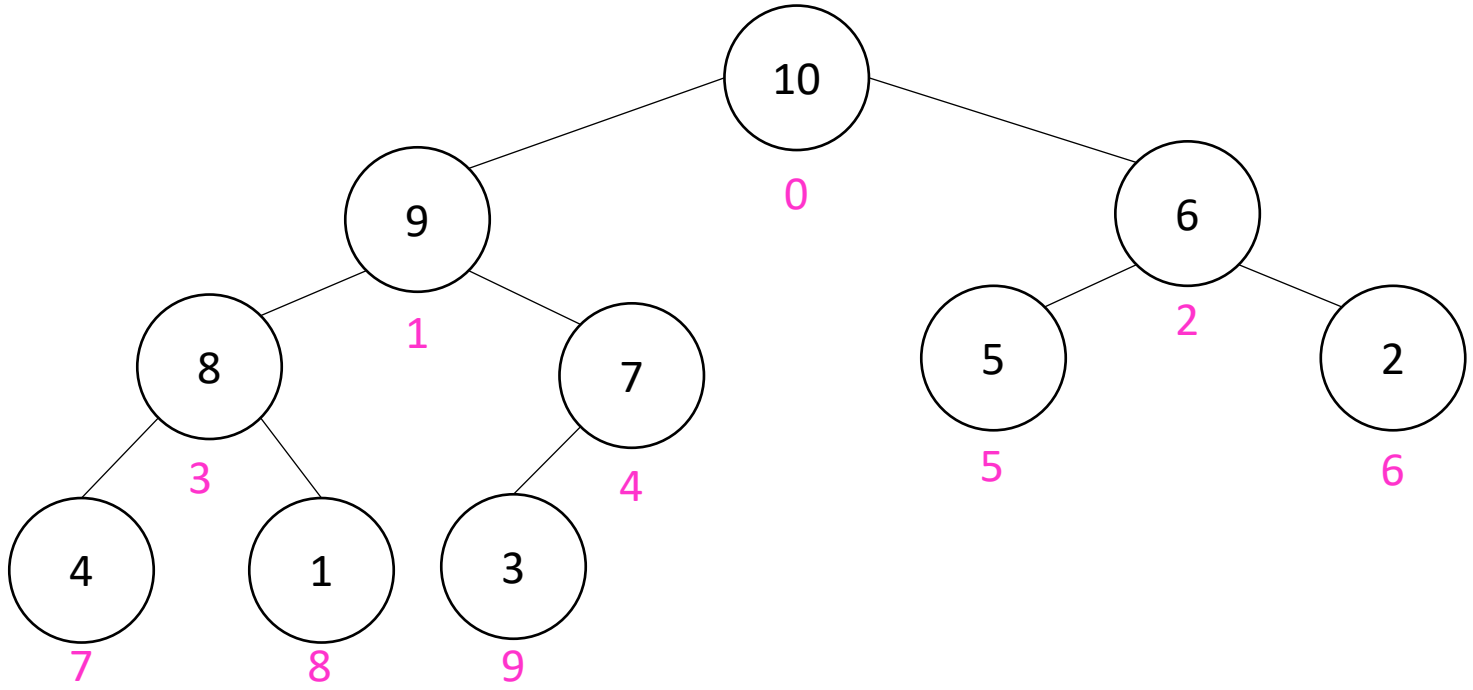
Summary: except for its asymptotic running time, it has everything you might want!

- Worst case running time
  - $\Theta(n^2)$
  - With VERY small constants! (No faster way of sorting a list of ≤ 50ish elements!)
- In place:
  - YES!
  - We only swap items within the given array
- Adaptive
  - YES!
  - The only elements that move are the elements that are out of position
- Online
  - Yes!
  - Each time an item arrives, “insert” it into the sorted prefix
- Stable
  - YES!
  - If the item we’re inserting has a tie with its left neighbor, don’t swap

Heap Sort

- Idea: 1) Build a maxHeap, then 2) repeatedly extract the max element from the heap to build sorted list Right-to-Left

Max Heap Property: Each node is larger than its children



10	9	6	8	7	5	2	4	1	3
0	1	2	3	4	5	6	7	8	9

Reminder: Floyd's buildHeap method

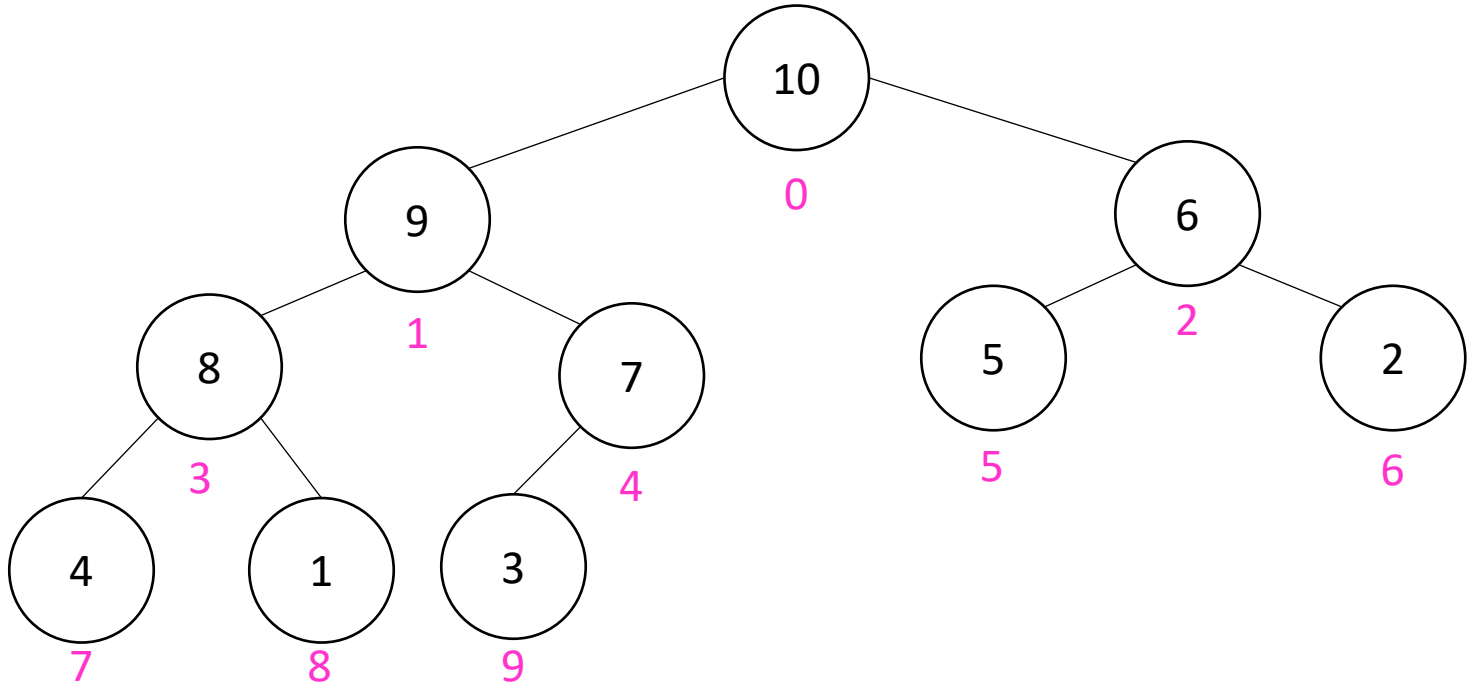
- Working towards the root, one row at a time, percolate down

```
buildHeap() {
    for (int i = size; i > 0; i--) {
        percolateDown(i);
    }
}
```

Heap Sort (Example)

- Idea: Build a maxHeap, repeatedly extract the max element from the heap to build sorted list Right-to-Left

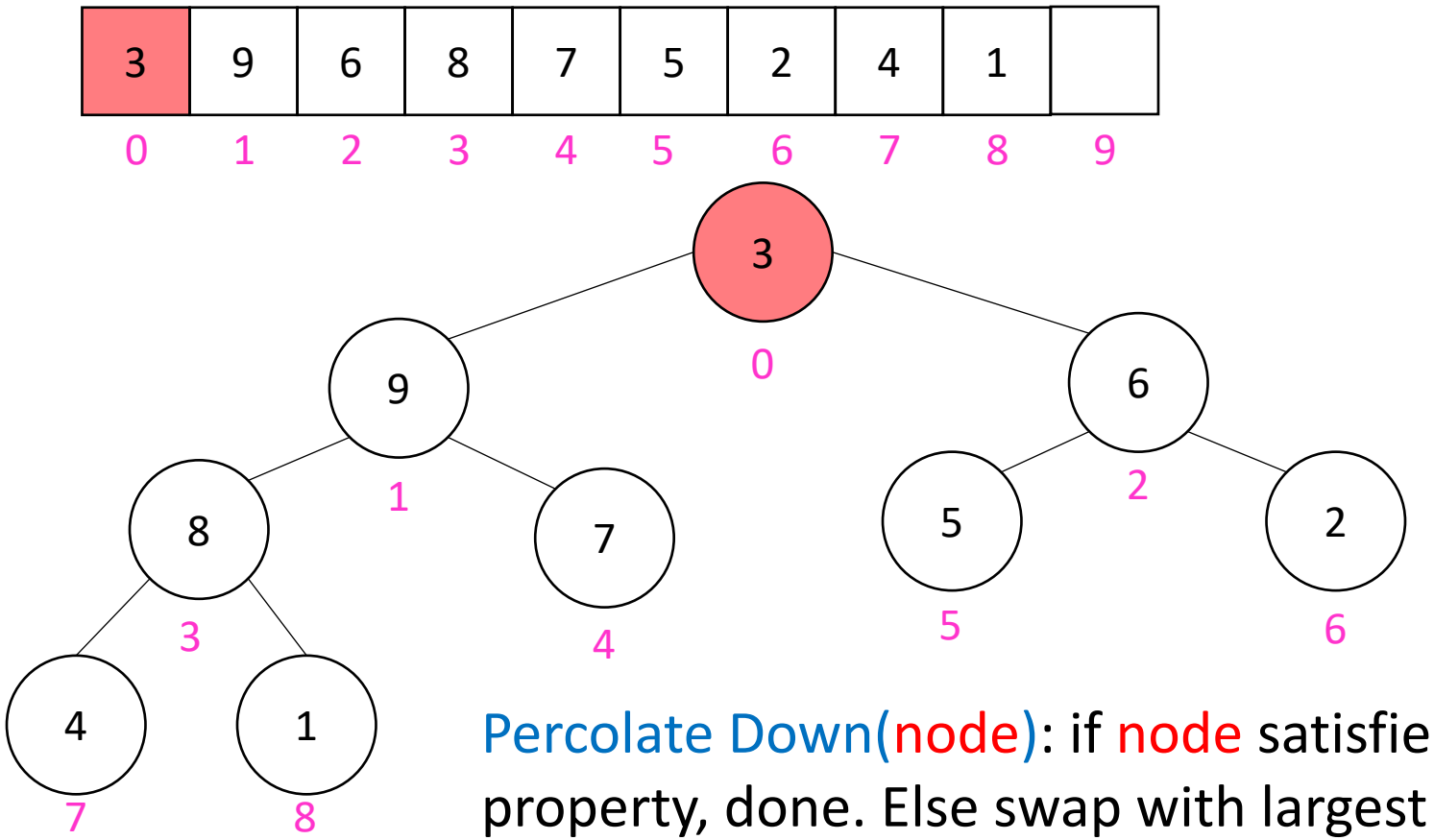
Max Heap Property: Each node is larger than its children



10	9	6	8	7	5	2	4	1	3
0	1	2	3	4	5	6	7	8	9

Heap Sort (First Extract)

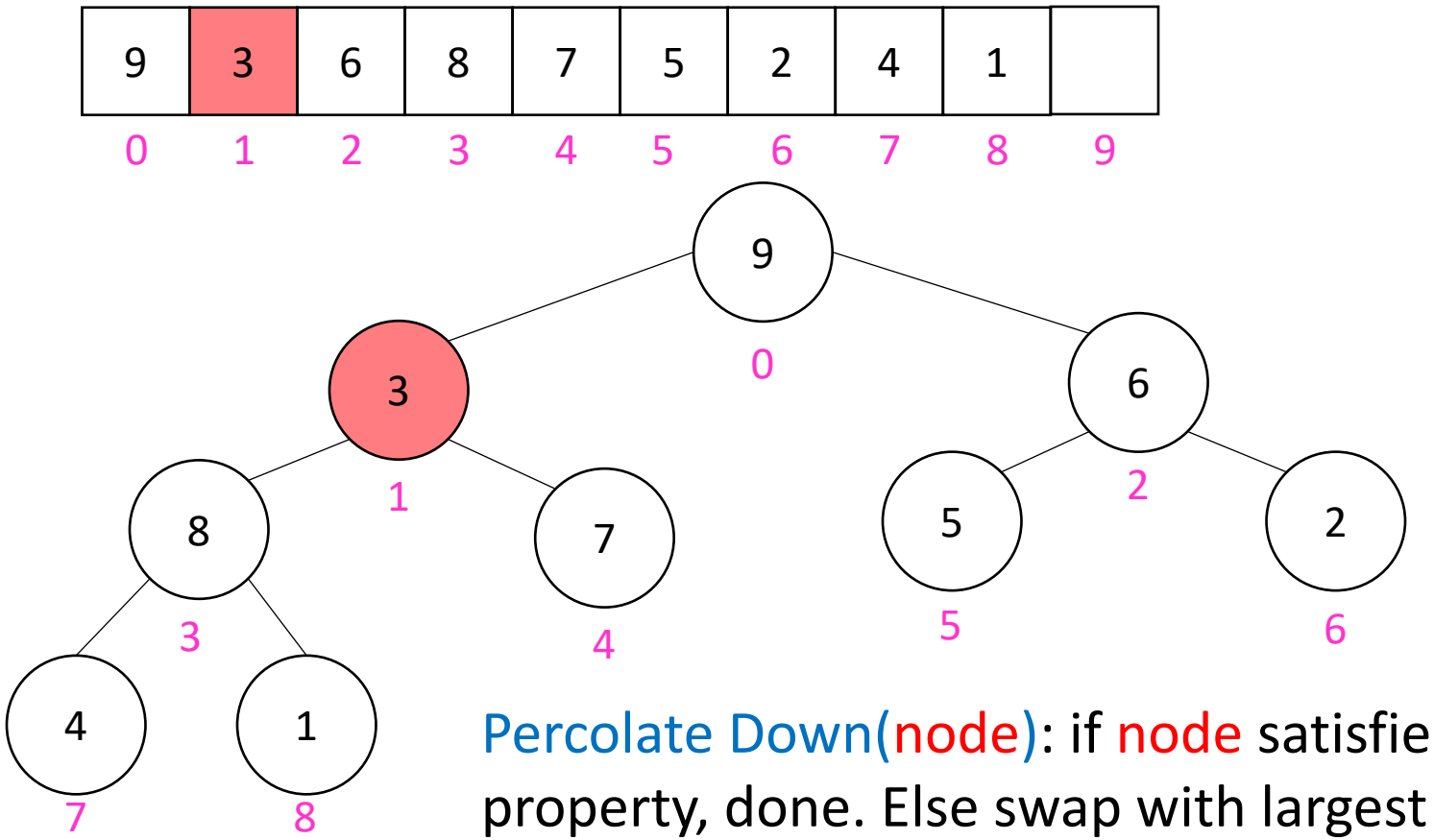
- Remove the Max element (i.e. the root) from the Heap:
  replace with last element, call percolateDown(root)



Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort (Percolate Down, Swap 1)

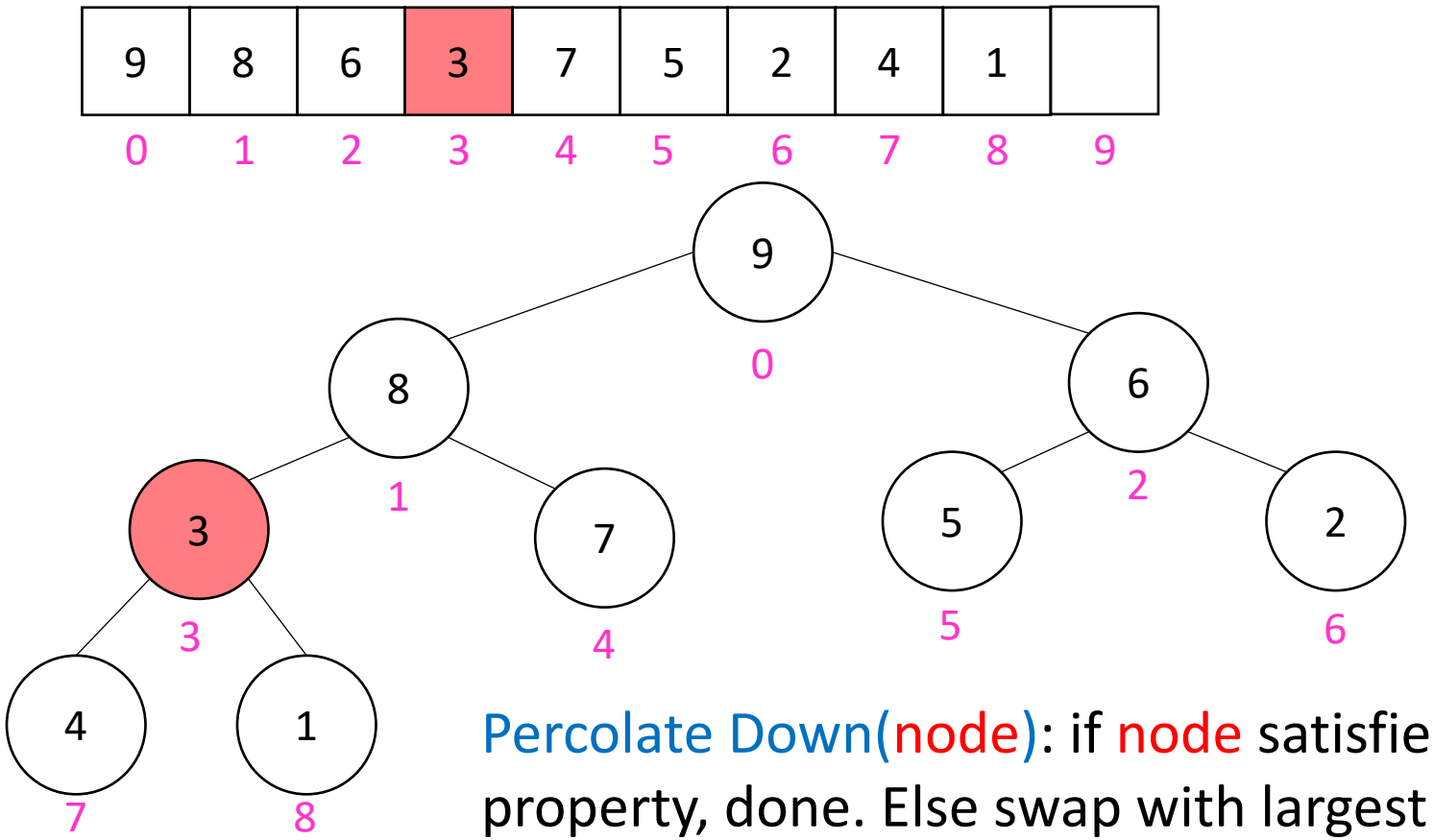
- Remove the Max element (i.e. the root) from the Heap:
  replace with last element, call percolateDown(root)



Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort (Percolate Down, Swap 2)

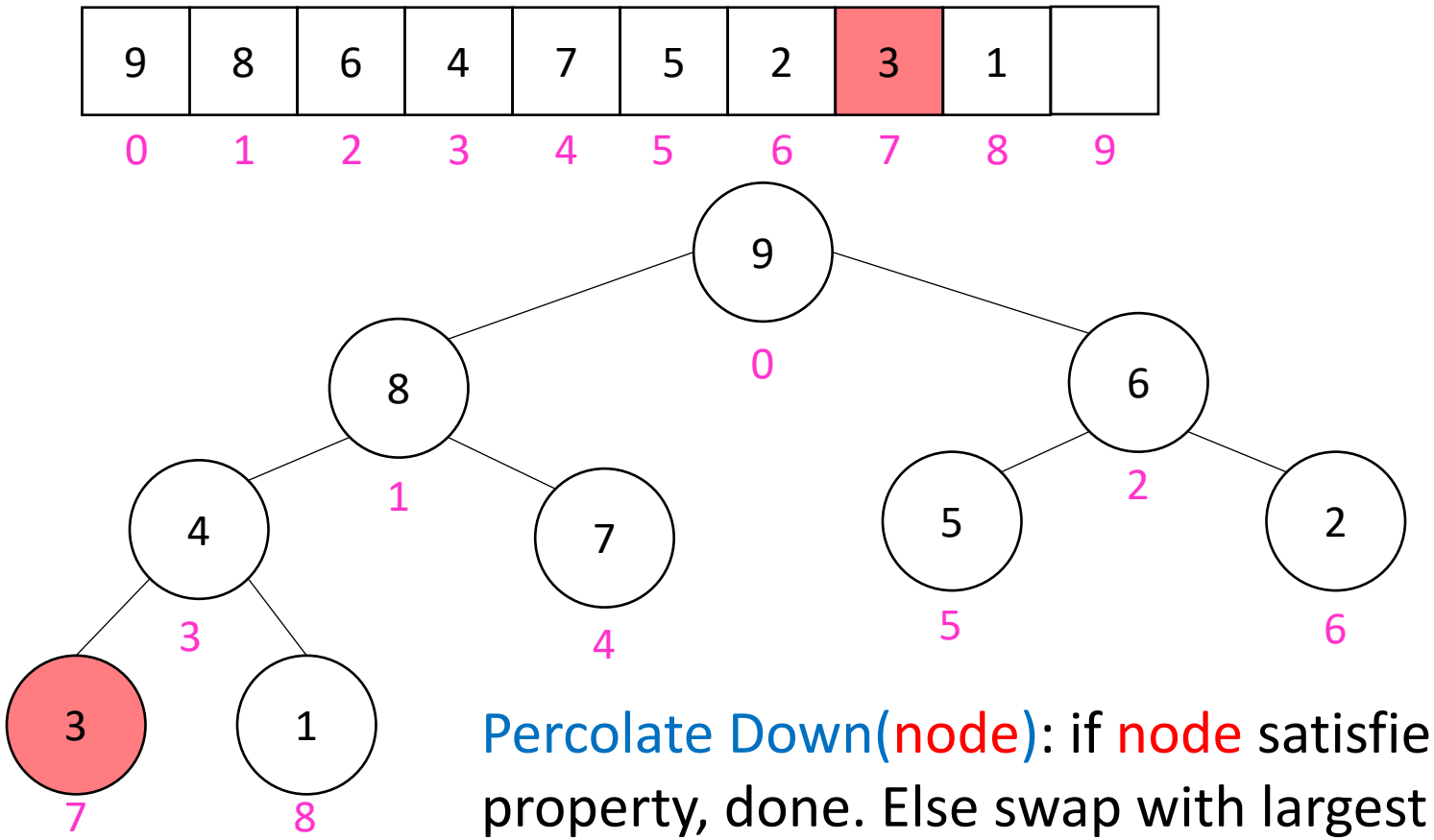
- Remove the Max element (i.e. the root) from the Heap:
  replace with last element, call percolateDown(root)



Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort (Percolate Down, Swap 3)

- Remove the Max element (i.e. the root) from the Heap:
  replace with last element, call percolateDown(root)



Percolate Down(node): if node satisfies heap property, done. Else swap with largest child and repeat on that subtree

Heap Sort - Summary

- Build a max heap
- Call extract
- Put that at the end of the array

```
myHeap = buildMaxHeap(a);
for (int i = a.length - 1; i >= 0; i--) {
        item = myHeap.extract();
        a[i] = item;
}
```

Running Time:

Worst Case:

Best Case:

Heap Sort - Answers

- Build a max heap
- Call extract
- Put that at the end of the array

```
myHeap = buildMaxHeap(a);
for (int i = a.length - 1; i >= 0; i--) {
	item = myHeap.extract();
	a[i] = item;
}
```

Running Time:

Worst Case: $\Theta(n \log n)$

Best Case: $\Theta(n \log n)$

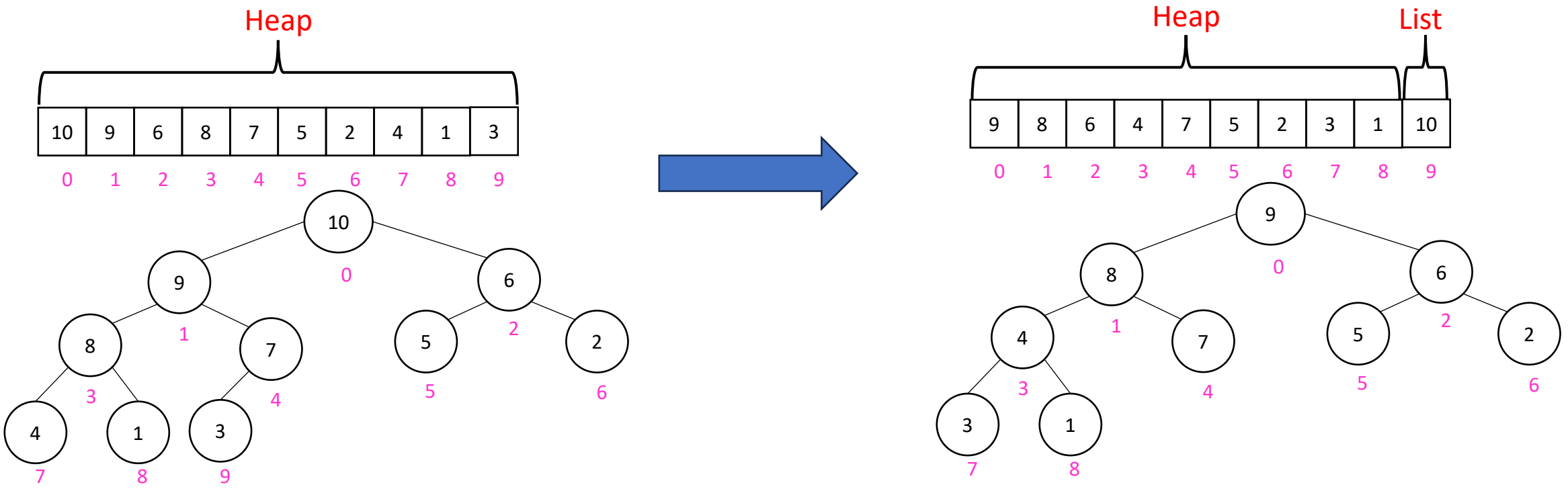
“In Place” Sorting Algorithm

- A sorting algorithm which requires no extra data structures
- Idea: It sorts items just by swapping things in the same array given
- Definition: it only uses $\Theta(1)$ extra space

- Insertion sort: In Place!
- Heap sort: Not In Place!
  - But we can fix that!

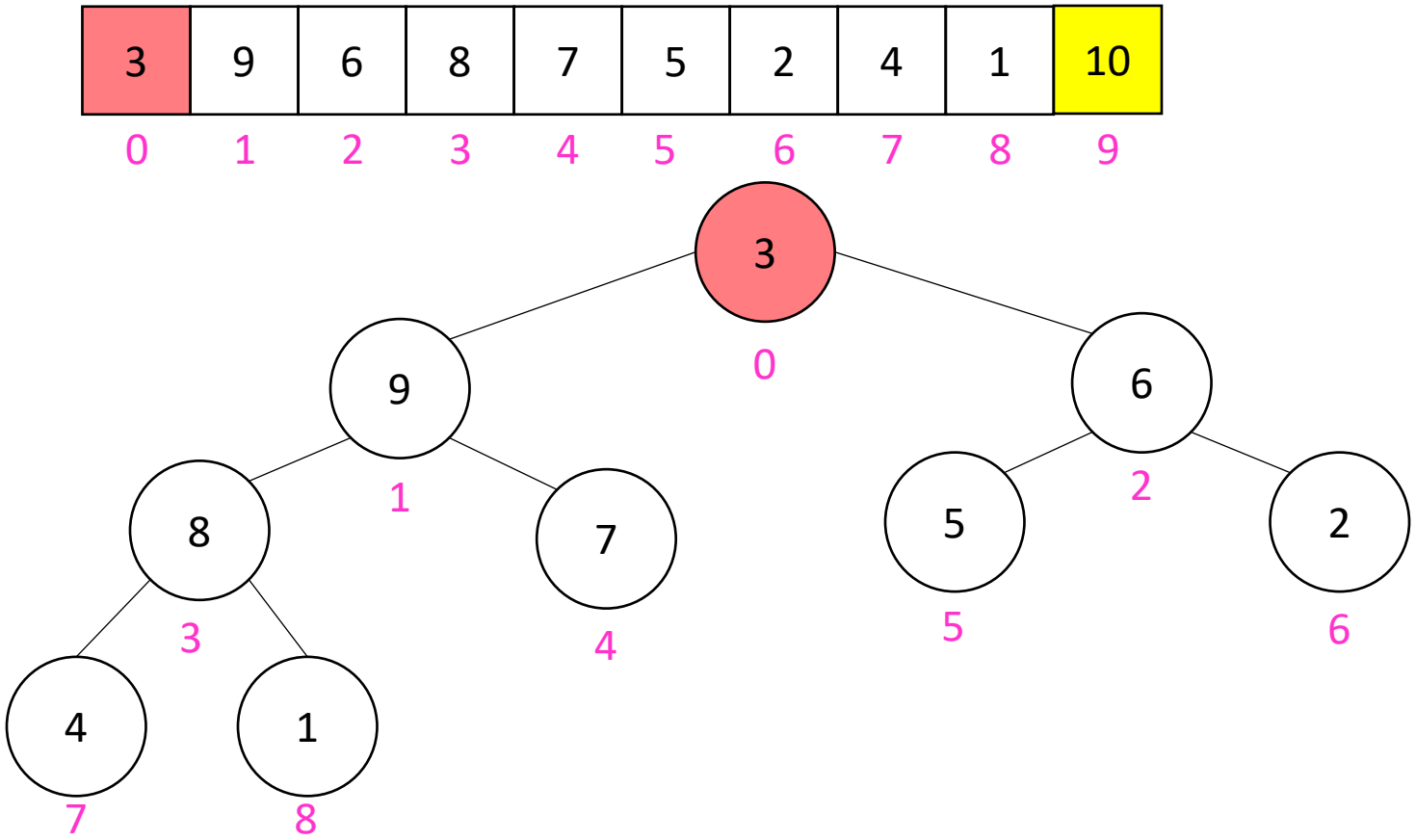
In Place Heap Sort

- Idea: Insert all items into a max heap, extract the largest one-by-one to fill sorted list from end to beginning, all the while the heap and sorted list share the same array



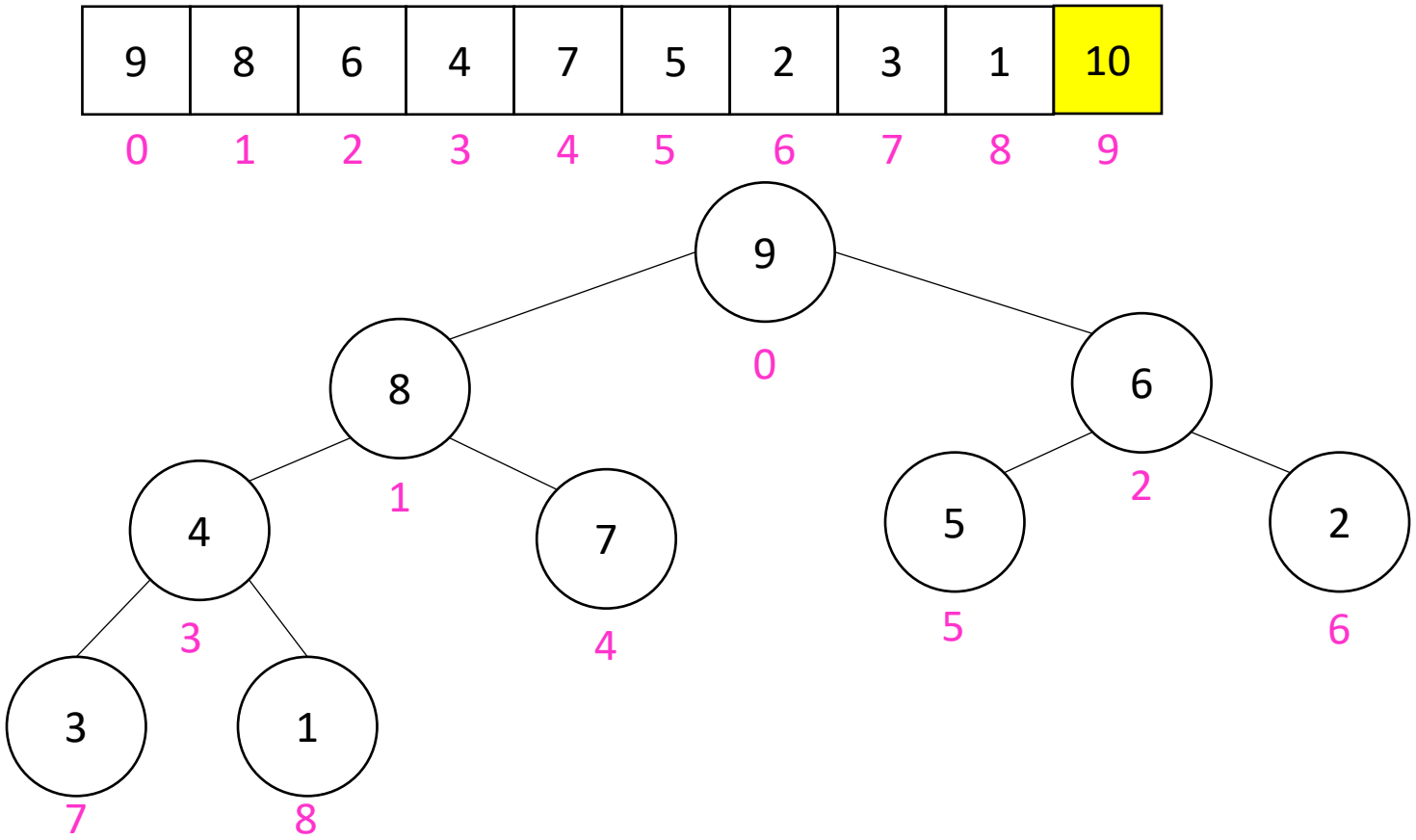
In Place Heap Sort (First Extract)

- Idea: When extracting an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter



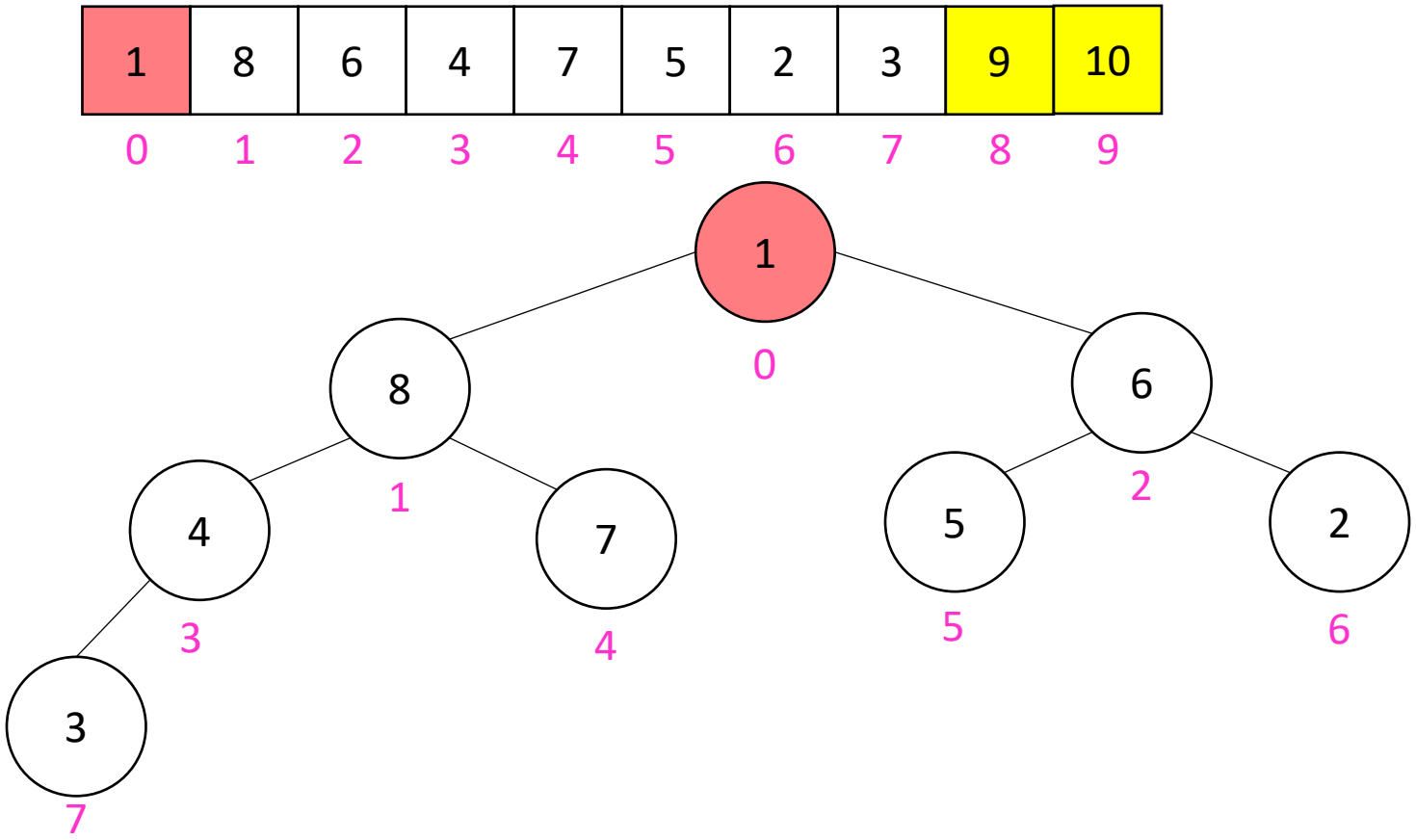
In Place Heap Sort (First Percolate Down)

- Idea: When extracting an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter



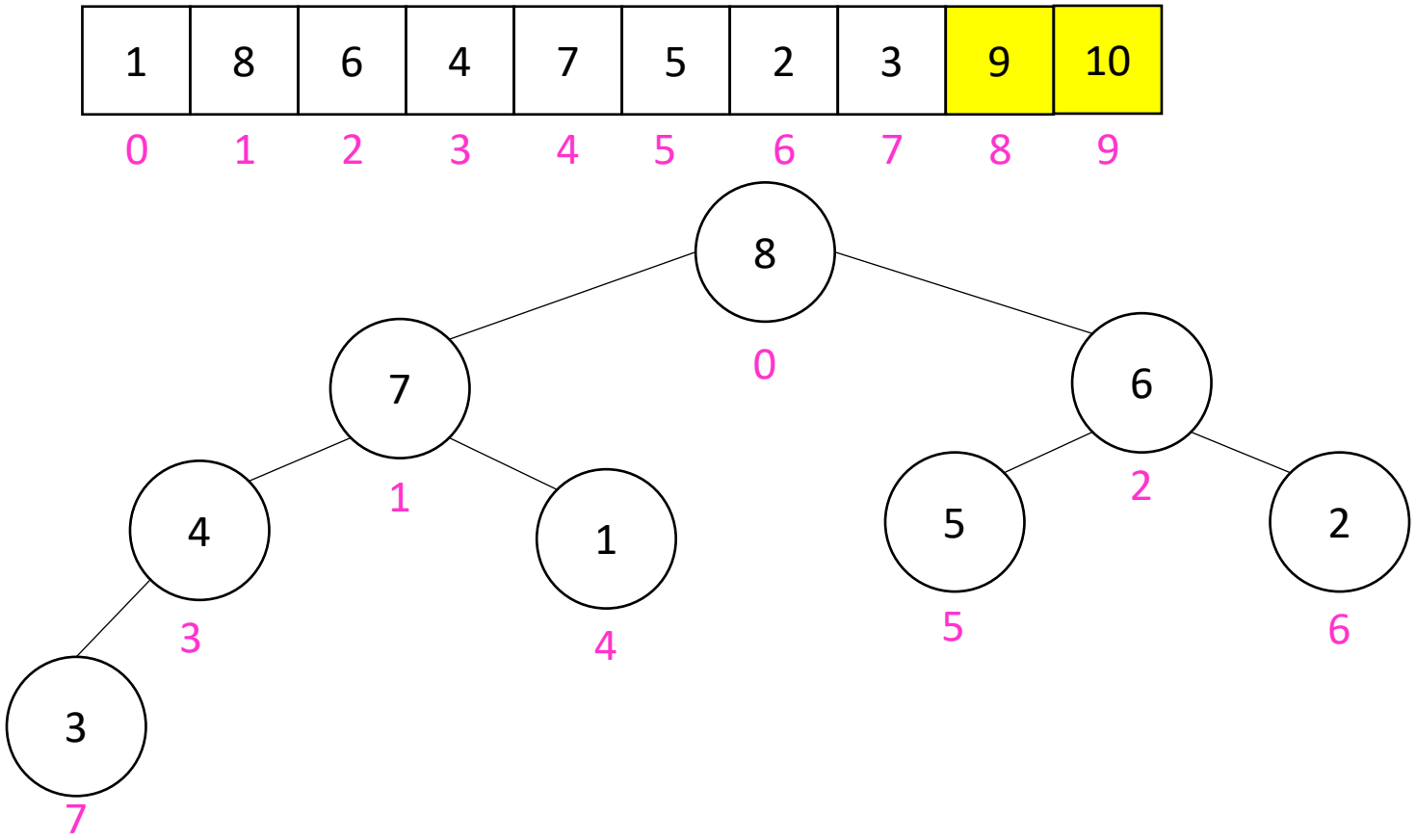
In Place Heap Sort (Second Extract)

- Idea: When extracting an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter



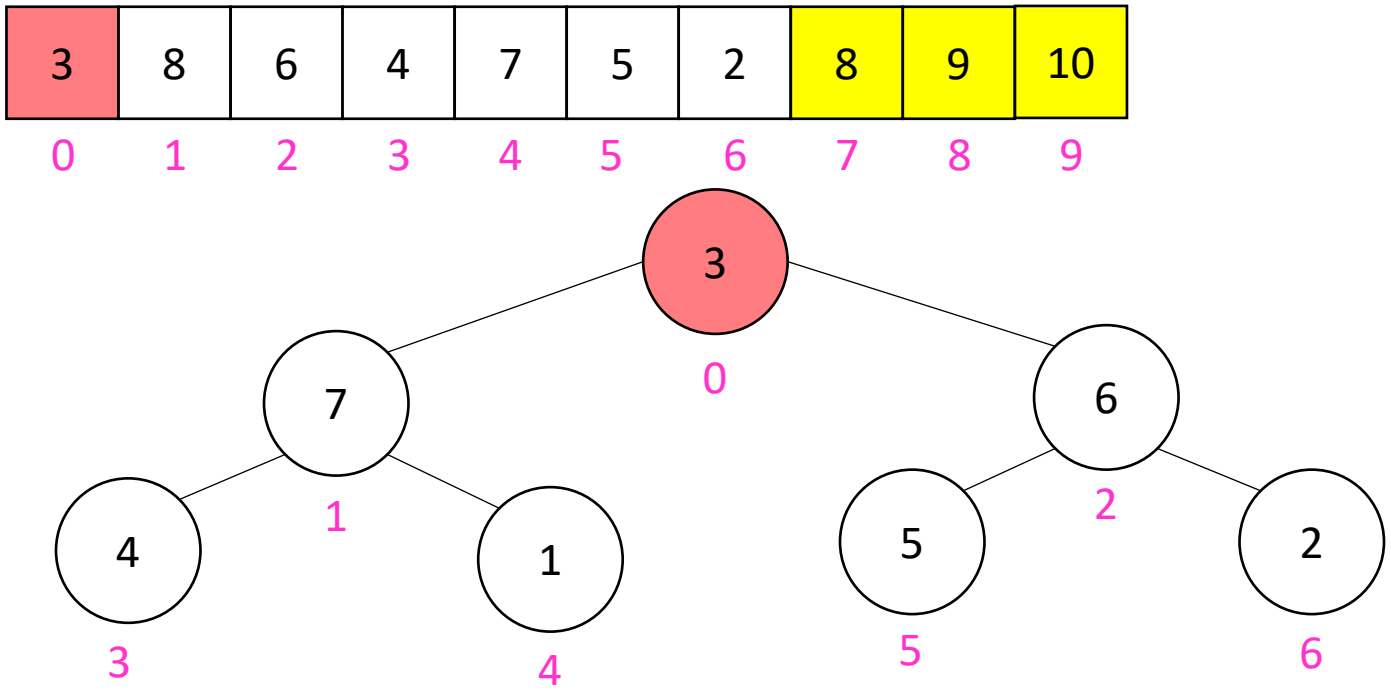
In Place Heap Sort (Second Percolate Down)

- Idea: When extracting an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter



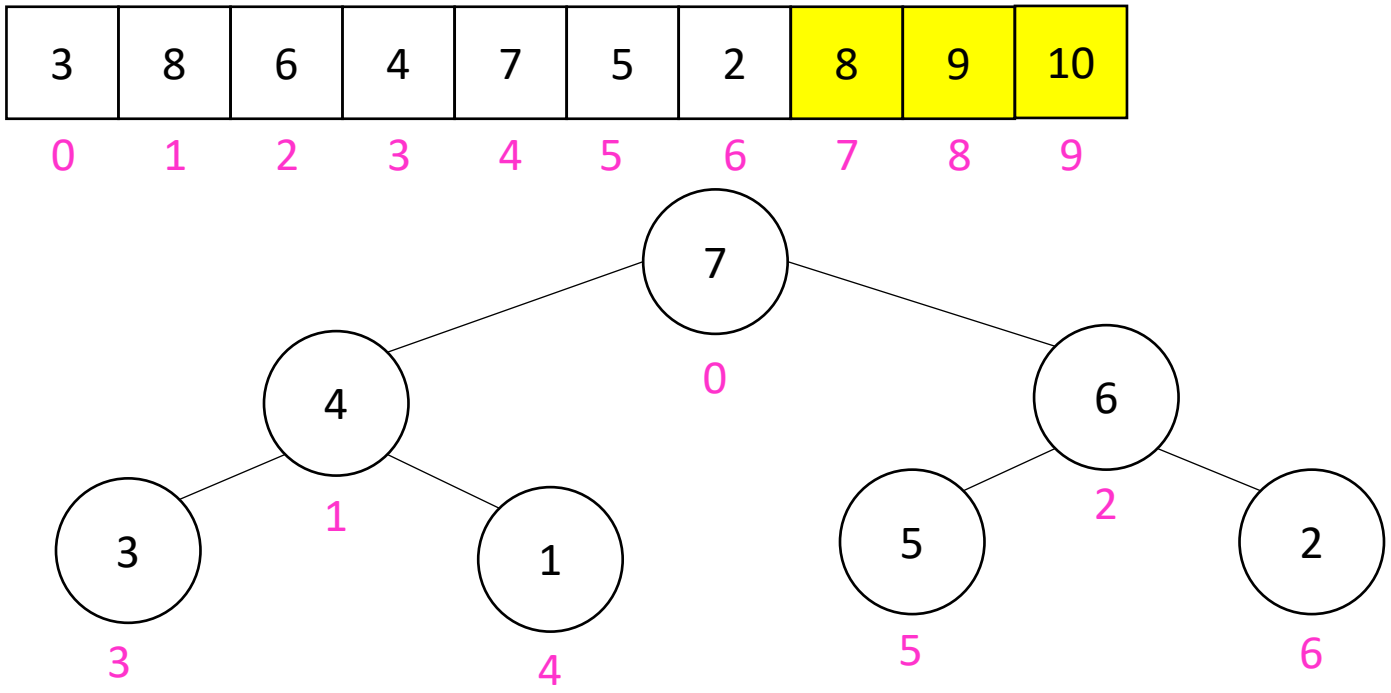
In Place Heap Sort (Third Extract)

- Idea: When extracting an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter



In Place Heap Sort (Third Percolate Down)

- Idea: When extracting an element from the heap, swap it with the last item of the heap then “pretend” the heap is one item shorter



In Place Heap Sort - Summary

- Build a heap using the same array (Floyd's build heap algorithm works)
- Call extract
- Put that at the end of the array

```
buildHeap(a);
for (int i = a.length - 1; i >= 0; i--) {
        temp = a[i]
        a[i] = a[0];
        a[0] = temp;
        percolateDown(0);
}
```

Running Time:

Worst Case: $\Theta(n \log n)$

Best Case: $\Theta(n \log n)$

Heap Sort Properties

- Worst case running time
  - $\Theta(n \log n)$
- In place:
  - YES!
  - We only swap items within the given array
- Adaptive
  - NO!
  - Running time is the same regardless of original list's order
- Online
  - NO!
  - We need all elements in the heap before we can start extracting
- Stable
  - NO!
  - Question on Exercise 6

Divide And Conquer Sorting

- Divide and Conquer:
  - Recursive algorithm design technique
  - Solve a large problem by breaking it up into smaller versions of the same problem

Divide and Conquer

- **Base Case**:
  - If the problem is “small” then solve directly and return
- **Divide**:
  - Break the problem into subproblem(s), each smaller instances
- **Conquer**:
  - Solve subproblem(s) recursively
- **Combine**:
  - Use solutions to subproblems to solve original problem