CS 4102: Algorithms

Lecture 10: Linear Time Sorting

David Wu

Fall 2019

Warm Up

Show that finding the minimum of an unordered list requires $\Omega(n)$ comparisons

Lower Bound Proof for Finding the Minimum

Show that finding the minimum of an unordered list requires $\Omega(n)$ comparisons

Suppose (toward contradiction) that there is an algorithm for that does fewer than $n/2 = \Omega(n)$ comparisons.

This means there is at least <u>one</u> element that was not looked at

We have no <u>information</u> on whether this element is the minimum or not!

2	8	19	20	[redacted]	3	9	-4
0	1	2	3	4	5	6	7

Today’s Keywords

Sorting algorithms

Linear-time sorting algorithms

Counting sort

Radix sort

Maximum sum continuous subarray

CLRS Readings: Chapter 8

Homework

- **HW3** due **Tuesday, October 1, 11pm**
  - Divide and conquer algorithms
  - Written (use LaTeX!) – Submit both **zip** and **pdf**!

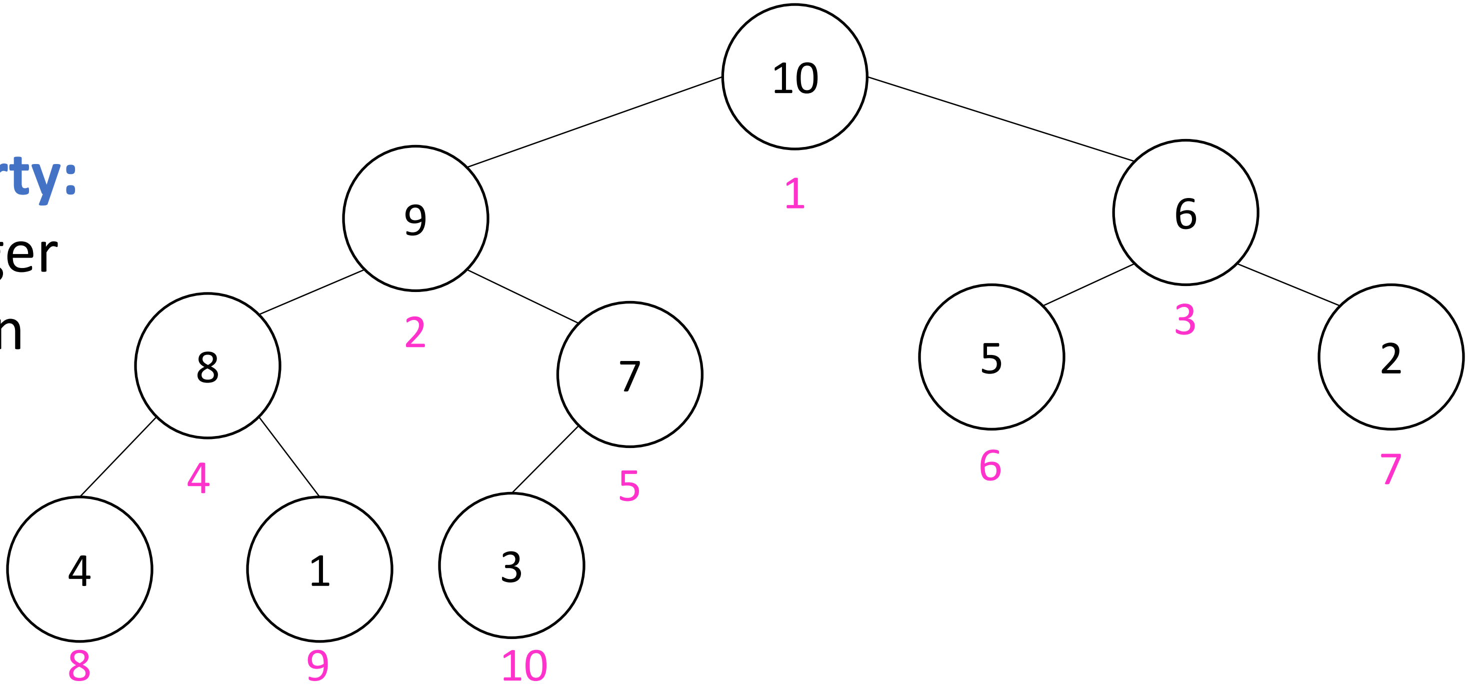
- **Regrade office hours:**
  - Thursday 11am-12pm (Rice 210)
  - Thursday 4pm-5pm (Rice 501)

Review: Heap Sort

Idea: Build a heap, repeatedly extract max element from the heap to build a sorted list (form right-to-left)

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

Max heap property:
Each node is larger than its children

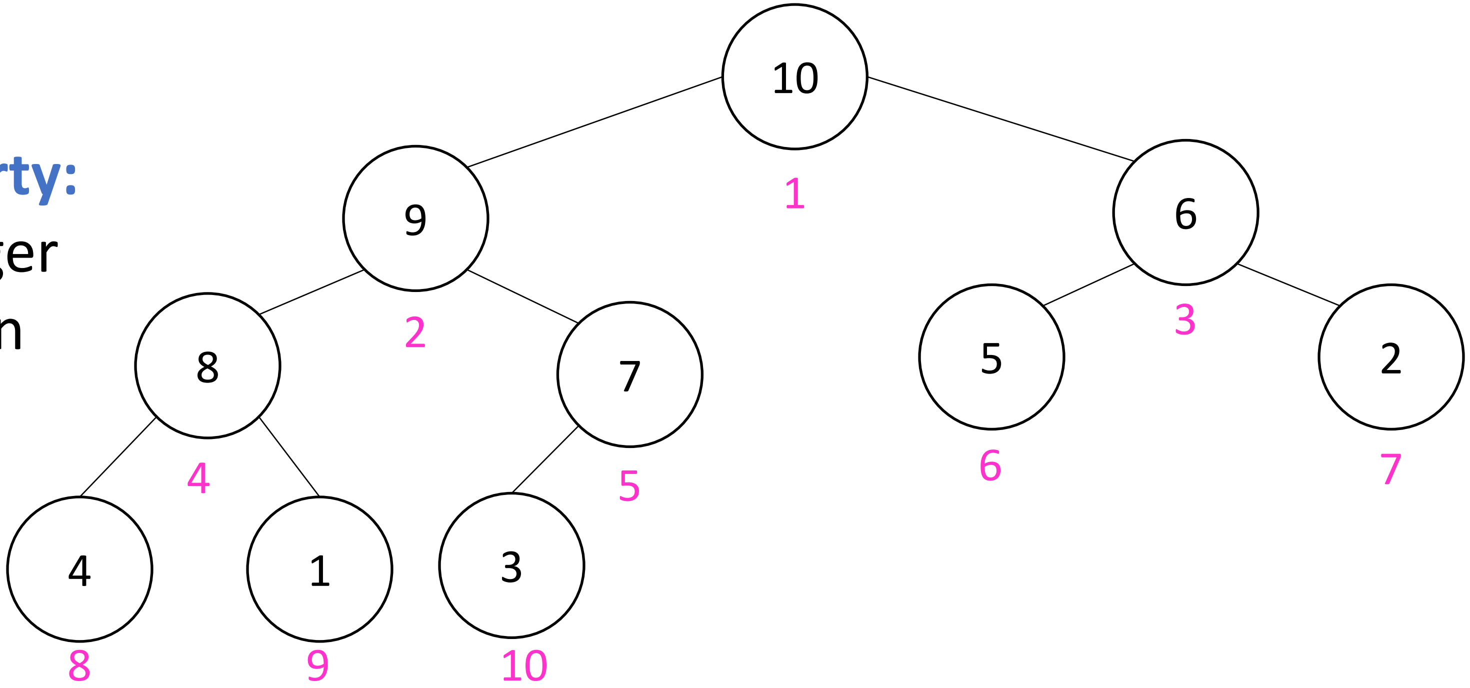


Review: Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)

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

Max heap property:
Each node is larger than its children

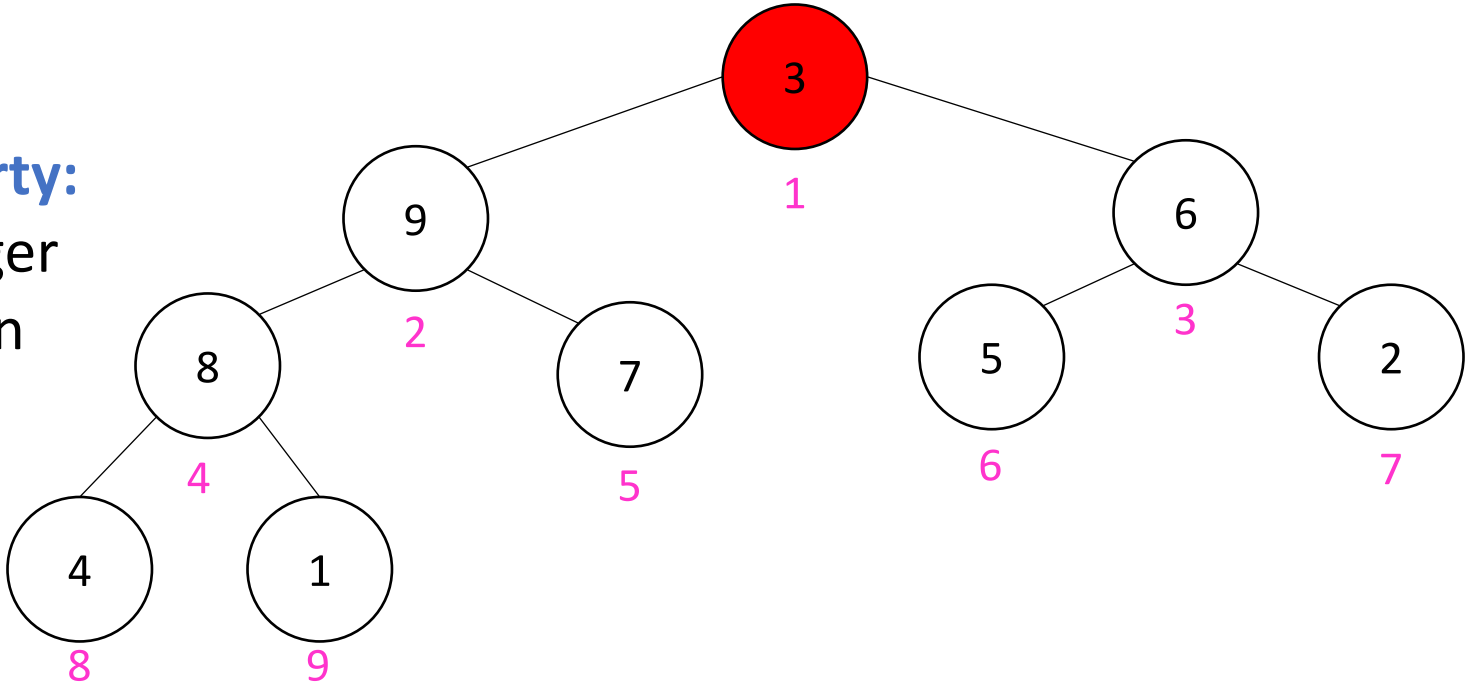


Review: Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)

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

Max heap property:
Each node is larger than its children

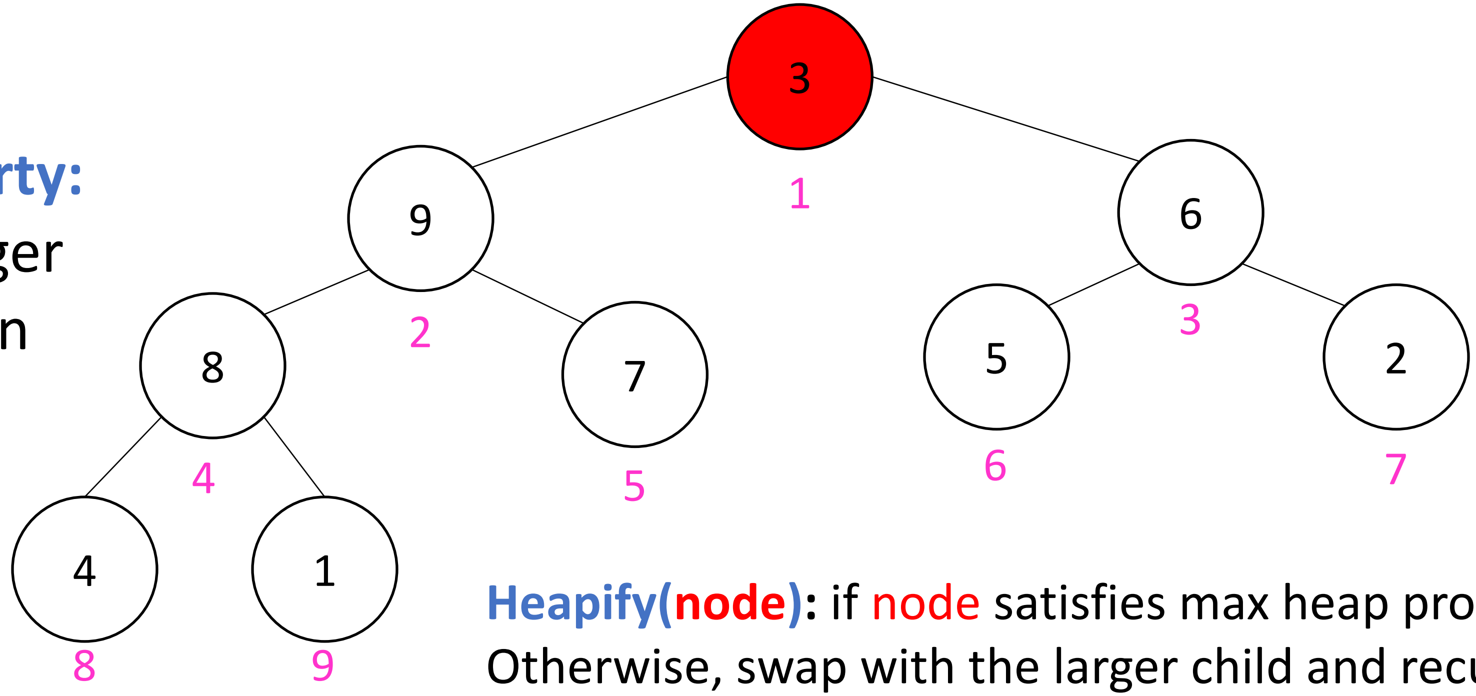


Review: Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)

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

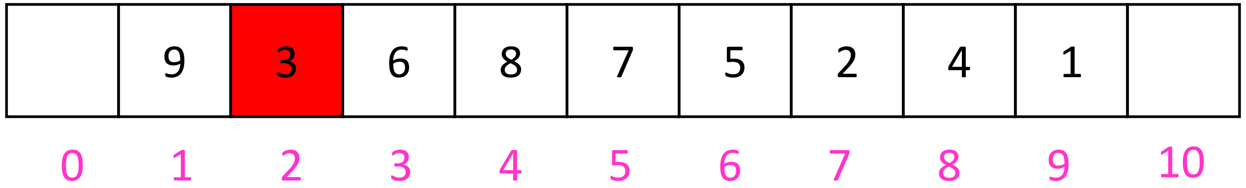
Max heap property:
Each node is larger than its children



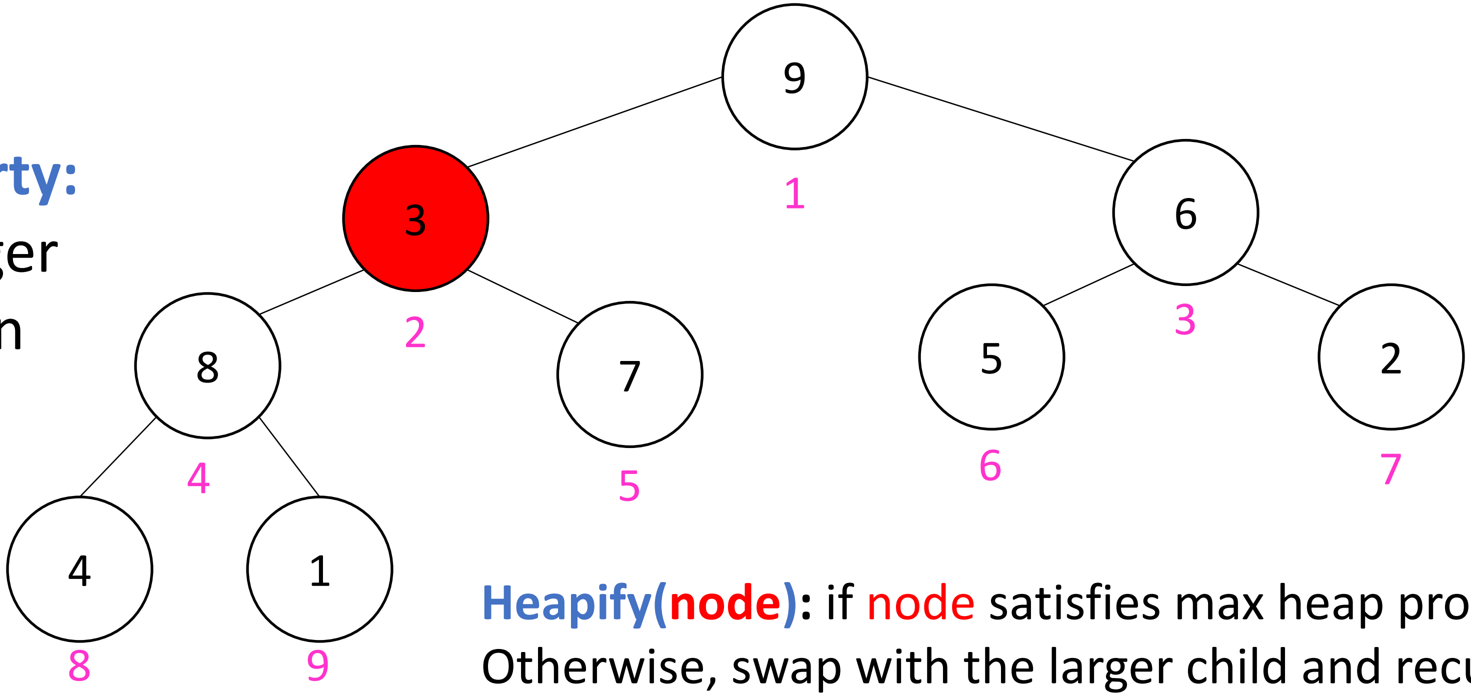
Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Review: Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)



Max heap property:
Each node is larger than its children



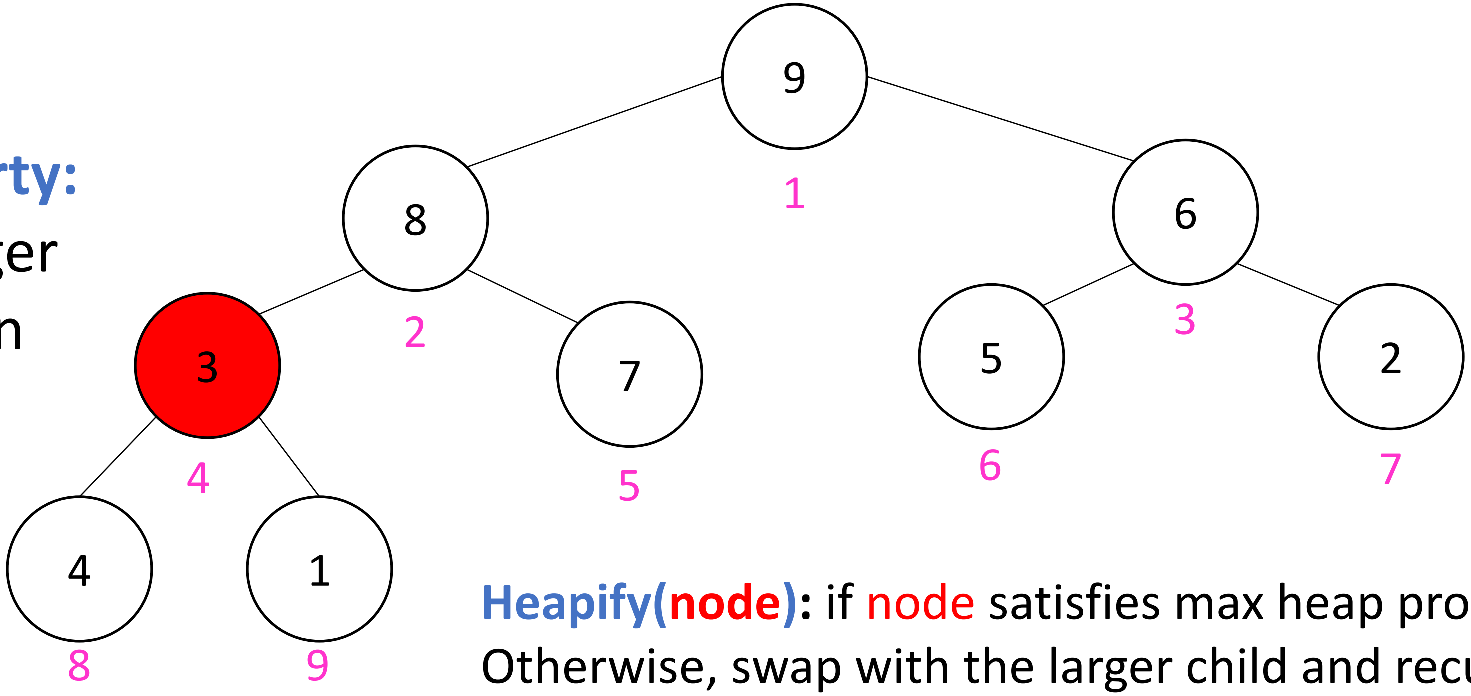
Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Review: Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)

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

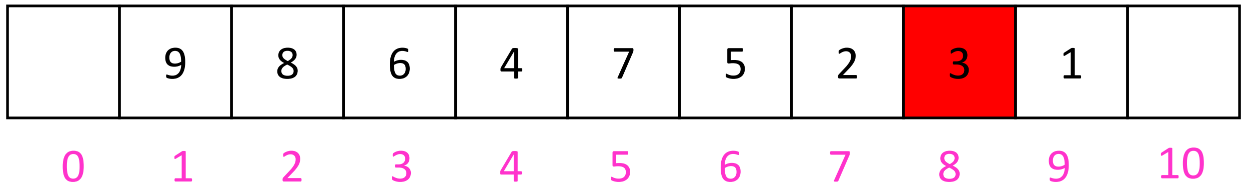
Max heap property:
Each node is larger than its children



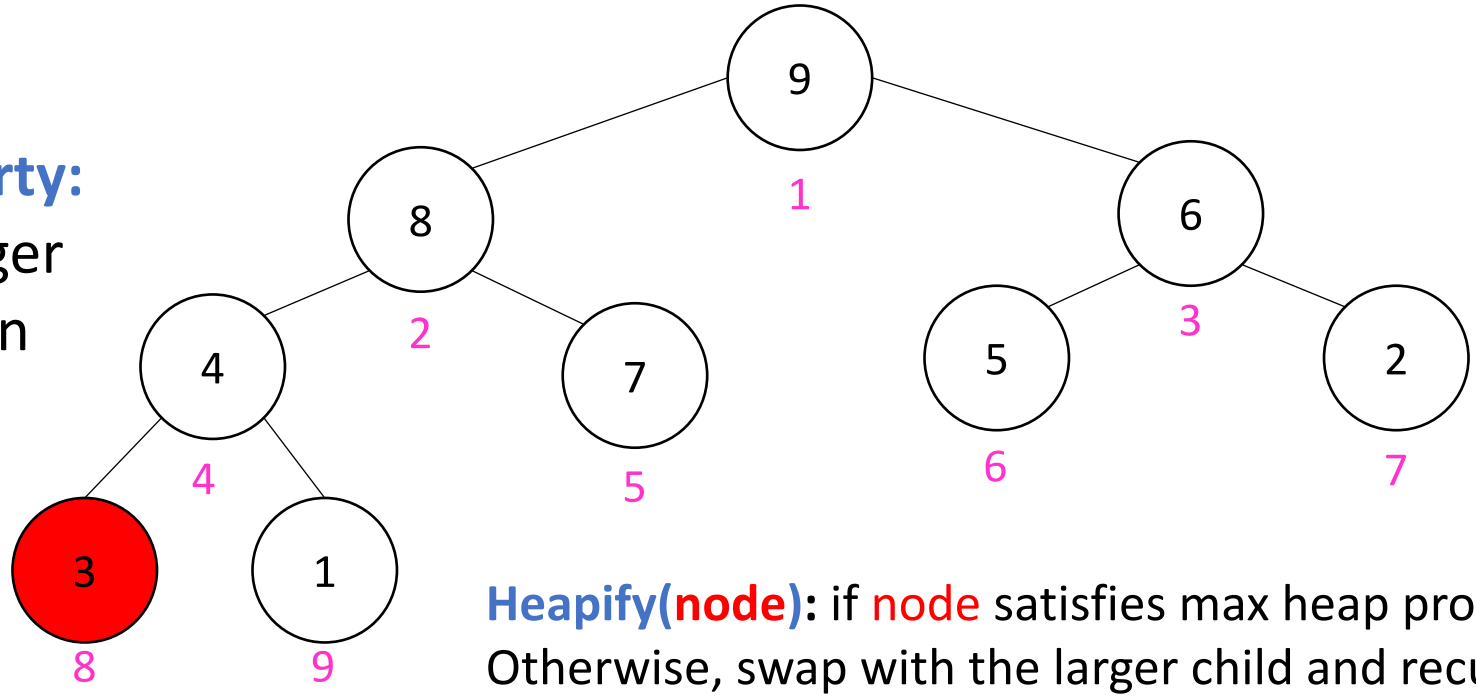
Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Review: Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)



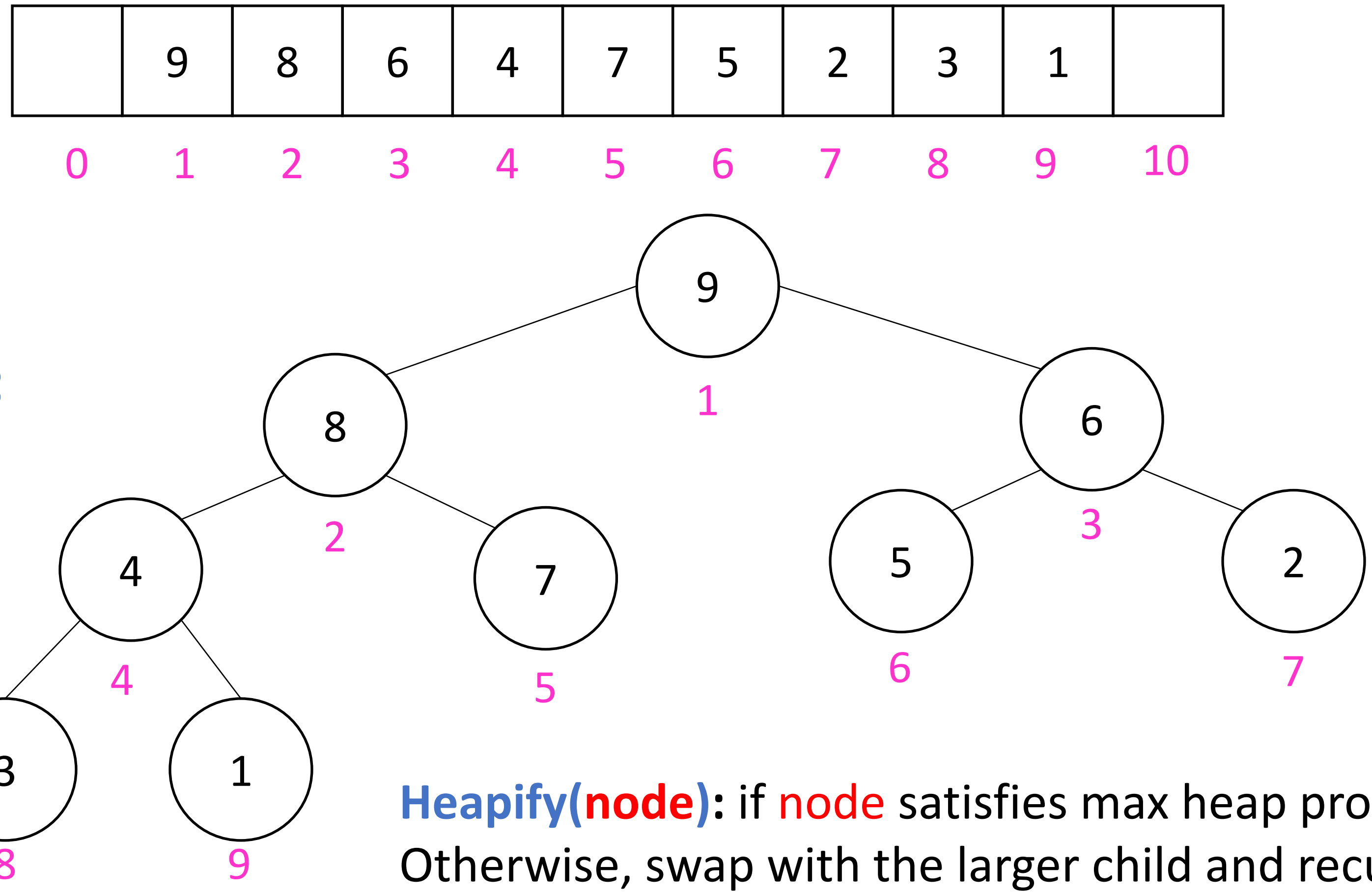
Max heap property:
Each node is larger than its children



Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Review: Heap Sort

Remove the max element (i.e. the root) from the heap, and the root with the last element, restore heap property by calling Heapify(root)



Max heap property:
Each node is larger than its children

Running time:
$O(\log n)$

Heapify(node): if node satisfies max heap property, then we are done. Otherwise, swap with the larger child and recurse on that subtree

Review: Heap Sort

Idea: Build a heap, repeatedly extract max element from the heap to build sorted list (from right to left)

Run Time?

$O(n \log n)$

(constants worse than quicksort)

Running time:

- Constructing heap by calling Heapify on each node in tree (bottom up): $O(n \log n)$
- Extracting maximum element to sort list: $O(n \log n)$

Review: Heap Sort

Idea: Build a heap, repeatedly extract max element from the heap to build sorted list (from right to left)

Run Time?

$O(n \log n)$

(constants worse than quicksort)

In Place?

Yes

When removing an element from the heap, move it to the (now unoccupied) end of the list

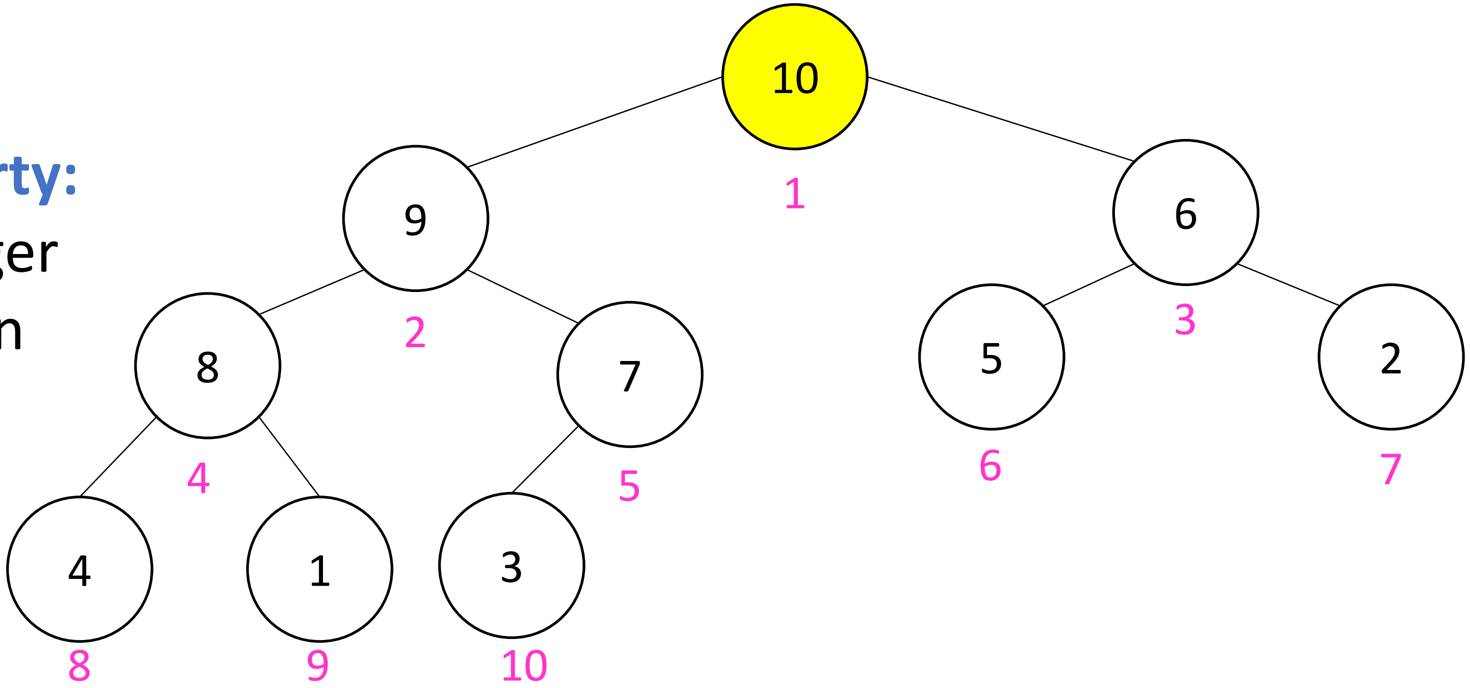
Constructing heap is also in-place (just requires calling Heapify)

In-Place Heap Sort

Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

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

Max heap property:
Each node is larger than its children

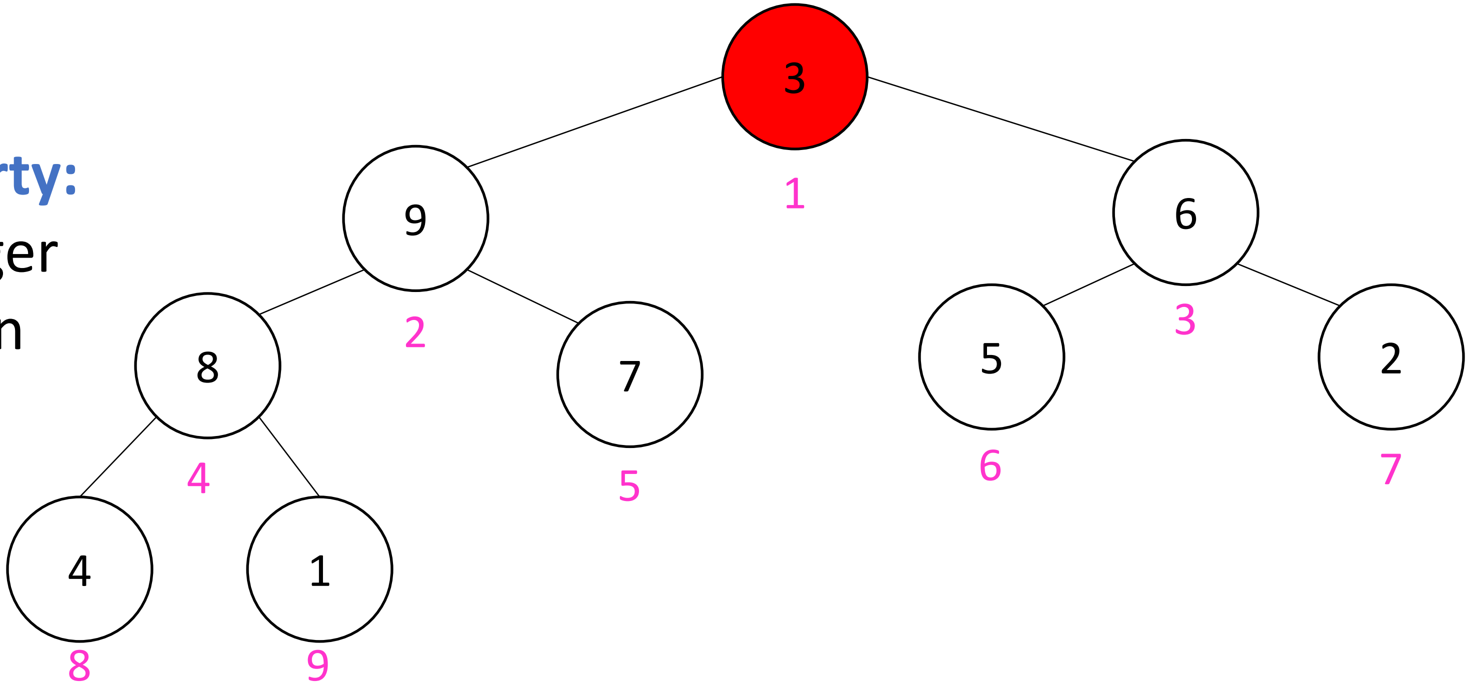


In-Place Heap Sort

Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

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

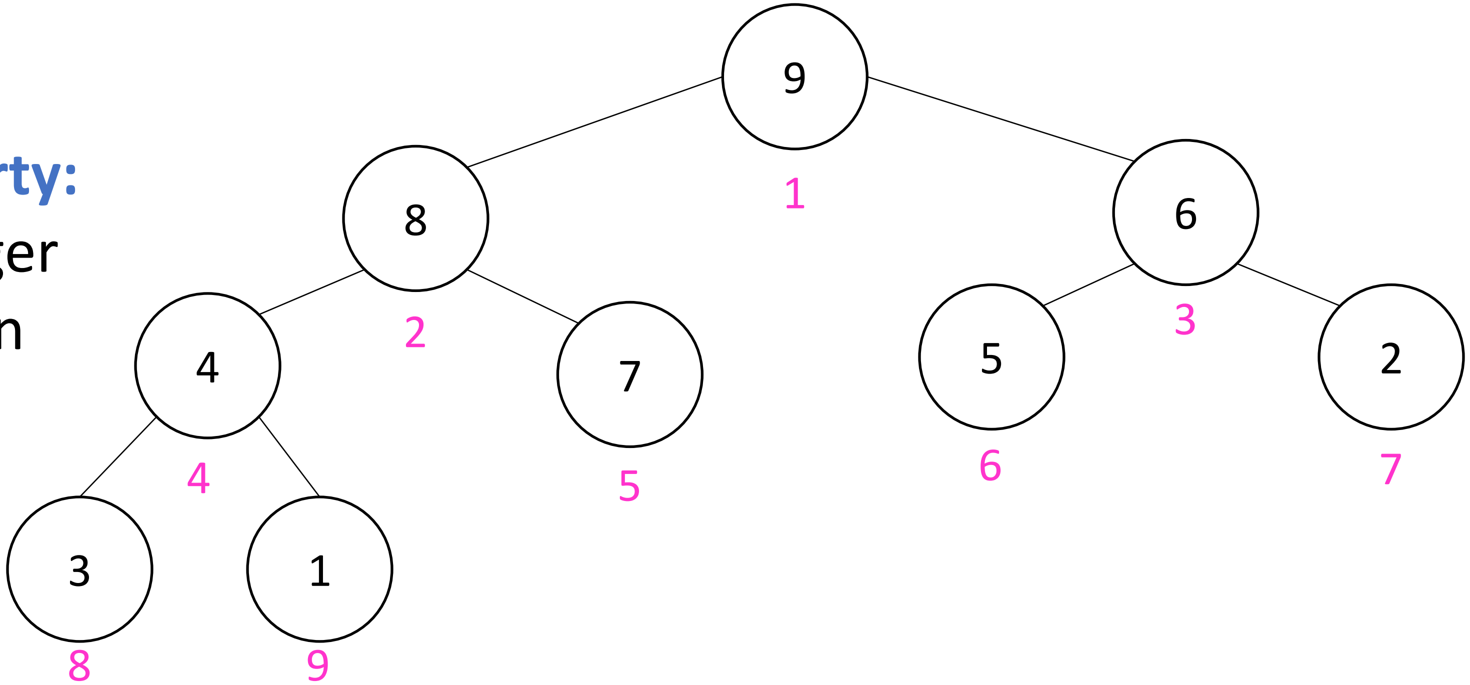
Max heap property:
Each node is larger than its children



In-Place Heap Sort

Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

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



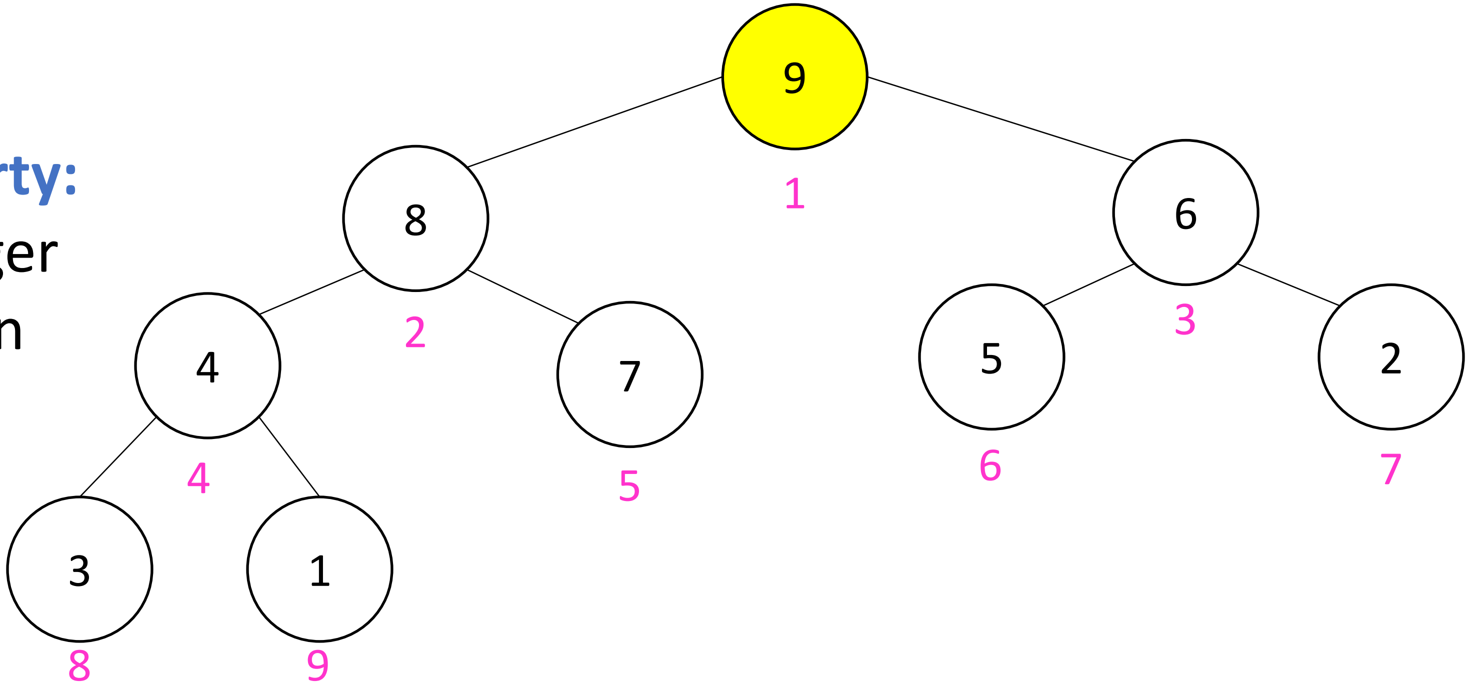
Max heap property:
Each node is larger than its children

In-Place Heap Sort

Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

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

Max heap property:
Each node is larger than its children

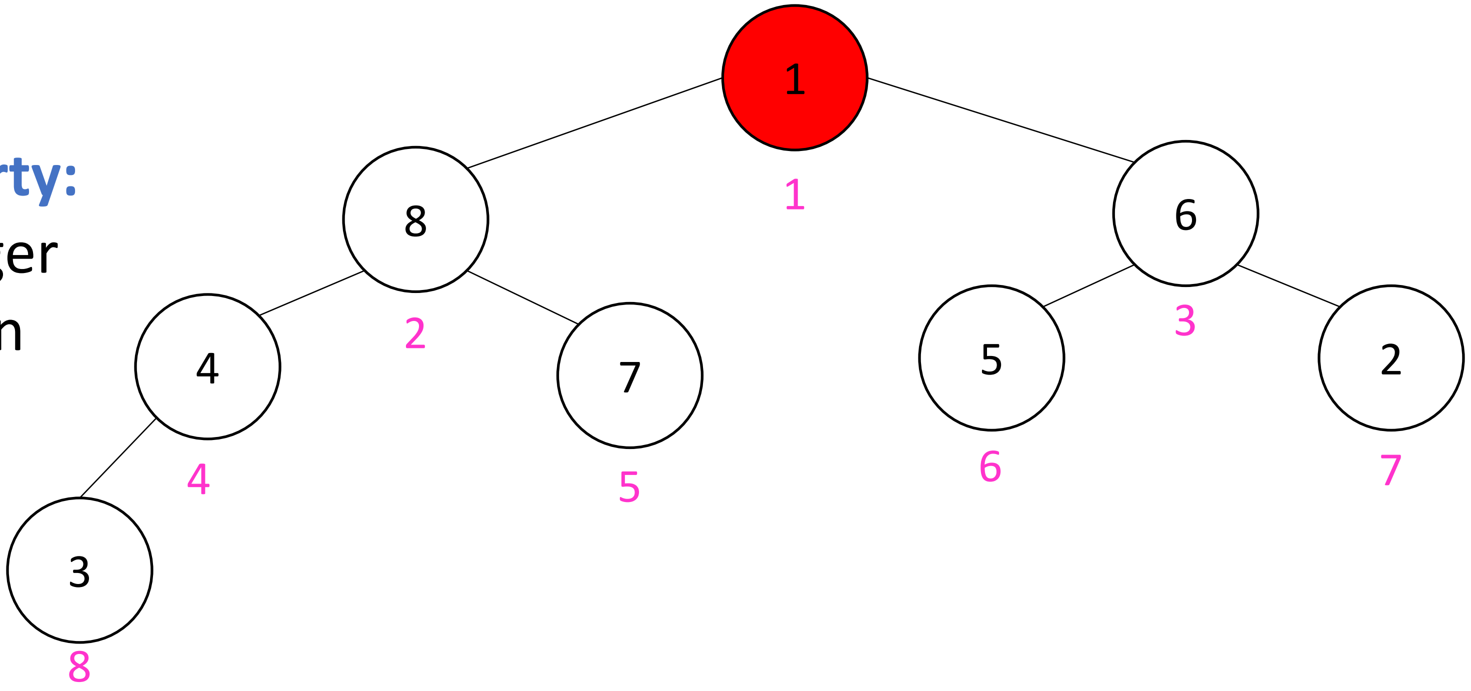


In-Place Heap Sort

Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

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

Max heap property:
Each node is larger than its children

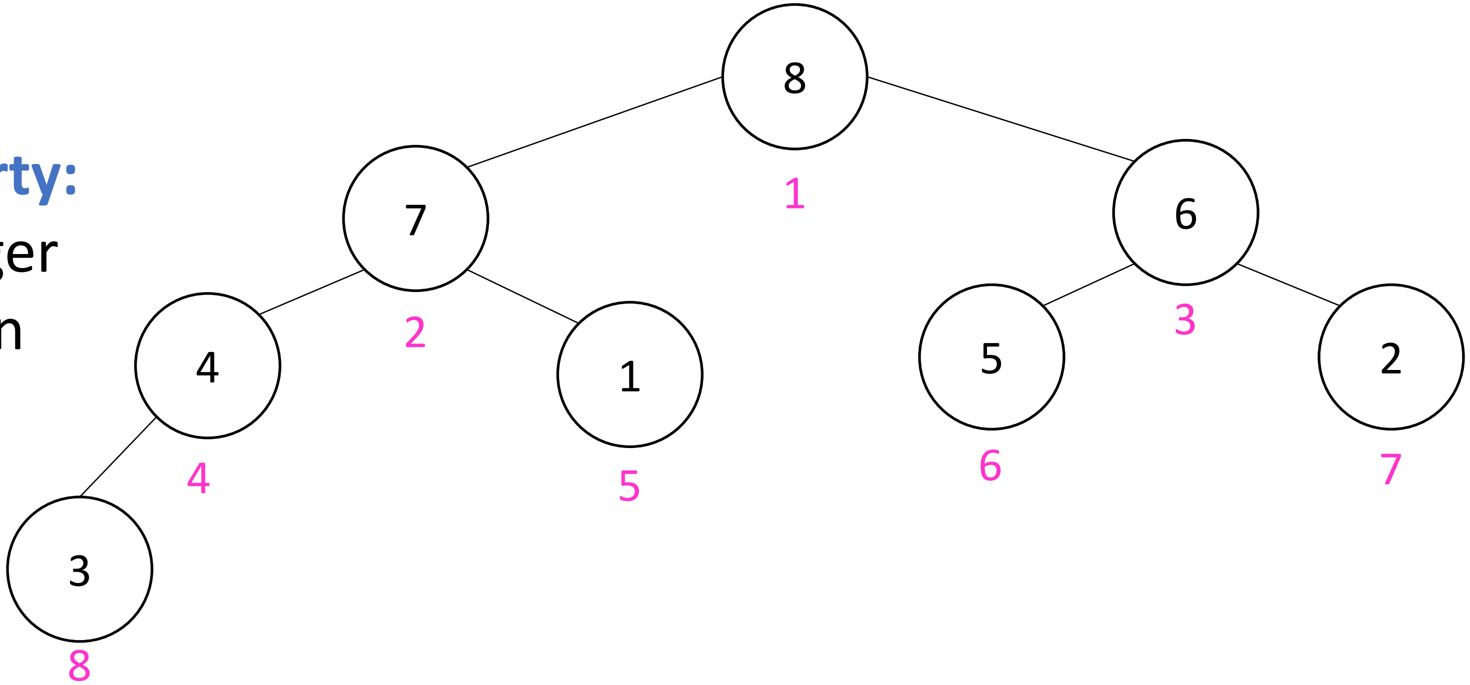


In-Place Heap Sort

Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

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

Max heap property:
Each node is larger than its children

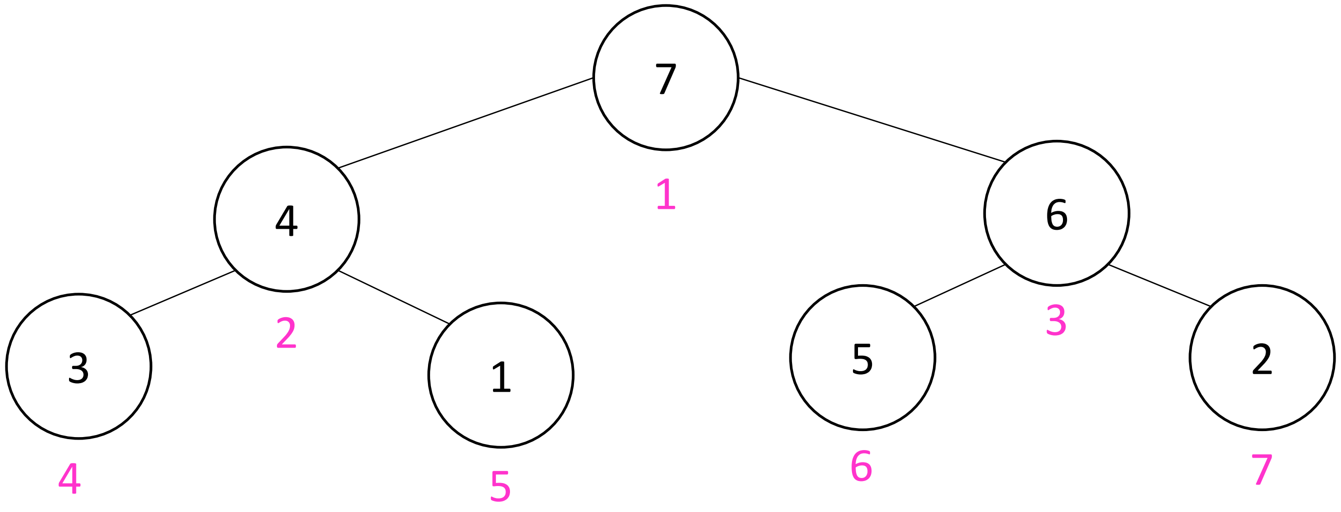


In-Place Heap Sort

Idea: When removing an element from the heap, move it to the (now unoccupied) end of the list

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

Max heap property:
Each node is larger than its children



Heap Sort

Idea: Build a heap, repeatedly extract max element from the heap to build sorted list (from right to left)

Run Time?

$O(n \log n)$

(constants worse than quicksort)

In Place?	Adaptive?	Stable?	Parallelizable?
Yes	No	No	No

Sorting Algorithms

Sorting algorithms we have discussed:

- Mergesort $O(n \log n)$
- Quicksort $O(n \log n)$

Other sorting algorithms (will discuss):

- Bubble sort $O(n^2)$
- Insertion sort $O(n^2)$
- Heapsort $O(n \log n)$

Can we do better than $O(n \log n)$?

Sorting in Linear Time

Cannot be a comparison sort

Implication: Need to make additional assumption about list contents

- Small number of unique values
- Small range of values

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

- Range is $[1, k]$ (here, $k = 6$)
- Initialize an array C of size k
- Count number of times each value occurs

$C =$

2	0	2	1	0	3
1	2	3	4	5	6

Value 1 appears 2 times

Value 4 appears 1 time

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

- Range is $[1, k]$ (here, $k = 6$)
- Initialize an array C of size k
- Count number of times each value occurs

$C =$

2	0	2	1	0	3
1	2	3	4	5	6

- Compute "running sum" of the number of values less than each value

$C =$

2					
1	2	3	4	5	6

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

- Range is $[1, k]$ (here, $k = 6$)
- Initialize an array C of size k
- Count number of times each value occurs

$C =$

2	0	2	1	0	3
1	2	3	4	5	6

- Compute "running sum" of the number of values less than each value

$C =$

2	2				
1	2	3	4	5	6

Counting Sort

Assumption: <u>Small</u> number of unique values

Idea: Count how many values are less than each element

$L =$

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

- Range is $[1, k]$ (here, $k = 6$)
- Initialize an array C of size k
- Count number of times each value occurs

$C =$

2	0	2	1	0	3
1	2	3	4	5	6

- Compute "running sum" of the number of values less than each value

$C =$

2	2	4			
1	2	3	4	5	6

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

- Range is $[1, k]$ (here, $k = 6$)
- Initialize an array C of size k
- Count number of times each value occurs

$C =$

1	2	3	4	5	6
2	0	2	1	0	3

- Compute "running sum" of the number of values less than each value

$C =$

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

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

- Range is $[1, k]$ (here, $k = 6$)
- Initialize an array C of size k
- Count number of times each value occurs
- Compute "running sum" of the number of values less than each value

Observation: Value at index i is index of the last value of i (if there is one)

Indices 1-2 has value 1

Index 5 has value 4

Indices 6-8 has value 6

$C =$

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

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

$B =$

1	2	3	4	5	6	7	8

For each element of L:
Use C to find its proper place in B
Decrement that position of C

$C =$

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

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

$B =$

1	2	3	4	5	6	7	8

For each element of L:
Use C to find its proper place in B
Decrement that position of C

$C =$

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

Counting Sort

Assumption: <u>Small</u> number of unique values

Idea: Count how many values are less than each element

$L =$

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

$B =$

1	2	3	4	5	6	7	8
			3				

Last index of value 3 is 4

$C =$

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

For each element of L:
Use C to find its proper place in B
Decrement that position of C

Counting Sort

Assumption: <u>Small</u> number of unique values

Idea: Count how many values are less than each element

$L =$

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

$B =$

			3				
1	2	3	4	5	6	7	8

Last index of value 3 is 4

$C =$

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

For each element of L:
Use C to find its proper place in B
Decrement that position of C

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

$B =$

			3				6
1	2	3	4	5	6	7	8

For each element of L:
Use C to find its proper place in B
Decrement that position of C

$C =$

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

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

$B =$

1	2	3	4	5	6	7	8
			3				6

For each element of L:
Use C to find its proper place in B
Decrement that position of C

$C =$

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

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

$B =$

1	2	3	4	5	6	7	8
			3			6	6

For each element of L:
Use C to find its proper place in B
Decrement that position of C

$C =$

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

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

$B =$

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

For each element of L:
Use C to find its proper place in B
Decrement that position of C

$C =$

0	2	2	4	5	5
1	2	3	4	5	6

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

- Range is $[1, k]$ (here, $k = 6$)
- Initialize an array C of size k
- Count number of times each value occurs

$\Theta(n + k)$

- Compute "running sum" of the number of values less than each value

$\Theta(k)$

For each element of L:
Use C to find its proper place in B
Decrement that position of C

$\Theta(n)$

Counting Sort

Assumption: Small number of unique values

Idea: Count how many values are less than each element

$L =$

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

Runtime: $\Theta(n+k)$

Space: $\Theta(n+k)$

- Range is $[1,k]$ (here, $k=6$)
- Initialize an array C of size k
- Count number of times each value occurs

$\Theta(n+k)$

- Compute "running sum" of the number of values less than each value

$\Theta(k)$

For each element of L:
Use C to find its proper place in B
Decrement that position of C

$\Theta(n)$

Counting Sort

Why not always use counting sort?

For 64-bit numbers, requires an array of length $2^{64} > 10^{19}$

- 5 GHz CPU will require > 116 years to initialize the array
- 18 Exabytes of data
  - Total amount of data that Google has

Somewhere Between 3 and 12 Exabytes

Radix Sort

Assumption: Values are numeric

Idea: Stable sort each digit, from least significant to most significant

103	801	401	323	255	823	999	101	113	901	555	512	245	800	018	121
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sort each element based on their 1's place

800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999
0	1	2	3	4	5	6	7	8	9

800	801	401	101	901	121	512	103	323	823	113	255	555	245	018	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Radix Sort

Assumption: Values are numeric

Idea: Stable sort each digit, from least significant to most significant

800	801	401	101	901	121	512	103	323	823	113	255	555	245	018	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sort each element based on their 10's place

0	1	2	3	4	5	6	7	8	9
800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999

Observe: digits in the 1's place are correctly sorted (because we are using a stable sort)!

Radix Sort

Assumption: Values are numeric

Idea: Stable sort each digit, from least significant to most significant

800	801	401	101	901	121	512	103	323	823	113	255	555	245	018	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sort each element based on their 10's place

0	1	2	3	4	5	6	7	8	9
800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999

800	801	401	101	901	103	512	113	018	121	323	823	245	255	555	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Radix Sort

Assumption: Values are <u>numeric</u>

Idea: <u>Stable</u> sort each digit, from least significant to most significant

800	801	401	101	901	103	512	113	018	121	323	823	245	255	555	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sort each element based on their 100's place

018	101 103 113 121	245 255	323	401	512 555			800 801 823	901 999
0	1	2	3	4	5	6	7	8	9

Observe: digits in the 1's and 10's places are correctly sorted (because we are using a <u>stable</u> sort)!

Radix Sort

Assumption: Values are numeric

Idea: Stable sort each digit, from least significant to most significant

800	801	401	101	901	103	512	113	018	121	323	823	245	255	555	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sort each element based on their 100's place

018	101 103 113 121	245 255	323	401	512 555			800 801 823	901 999
0	1	2	3	4	5	6	7	8	9

018	101	103	113	121	245	255	323	401	512	555	800	801	823	901	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Radix Sort

Assumption: Values are numeric

Idea: Stable sort each digit, from least significant to most significant

018	101	103	113	121	245	255	323	401	512	555	800	801	823	901	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Runtime: $\Theta\left(d(n+b)\right)$

Space: $\Theta(n+b)$

d: number of digits

b: base ("radix")

n: number of values

Maximum Sum Subarray Problem

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Maximum sum contiguous subarray (MSCS) problem:

find the largest <u>contiguous</u> subarray that maximizes the sum of the values

Maximum Sum Subarray Problem

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Maximum sum contiguous subarray (MSCS) problem:

find the largest contiguous subarray that maximizes the sum of the values

Divide and Conquer $\Theta(n \log n)$

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively solve on left

Divide in half

Recursively solve on right

Divide and Conquer $\Theta(n \log n)$

5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively solve on left

19

Divide in half

Recursively solve on right

25

Divide and Conquer $\Theta(n \log n)$

Largest sum that ends here + Largest sum that starts here

6	1	-7	-3	-6	-13	2	8	-12	5	13	-37	-42	-20
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively solve on left

19

Divide in half

Recursively solve on right

25

Combine: Find largest sum that spans the cut

Divide and Conquer $\Theta(n \log n)$

Largest sum that ends here + Largest sum that starts here

6	1	-7	-3	-6	-13	2	8	-12	5	**13**	-37	-42	-20
5	8	-4	3	7	-15	2	8	-20	17	8	-50	-5	22
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Recursively solve on left

19

Divide in half

Recursively solve on right

25

Combine: Find largest sum that spans the cut

19

$$T(n) = 2T(n/2) + \Theta(n) \in \Theta(n \log n)$$

Divide and Conquer Summary

Typically multiple subproblems
Typically all roughly the same size

Divide

- Break the list in half

Conquer

- Find the best subarrays on the left and right

Combine

- Find the best subarray that “spans the divide”
- Output best subarray among the three possible subarrays

Generic Divide and Conquer Template

```
def myDCalgo(problem):
    if baseCase(problem):
        solution = solve(problem) # brute force if necessary
        return solution
    subproblems = divide(problem)
    for sub in subproblems:
        subsolutions.append(myDCalgo(sub))
    solution = combine(subsolutions)
    return solution
```

MSCS Divide and Conquer $\Theta(n \log n)$

```
def MSCS(list):
        if list.length < 2:
                return list[0] # list of size 1 the sum is maximal
        {listL, listR} = divide(list)
        for list in {listL, listR}:
                subsolutions.append(MSCS(list))
        solution = max(solnL, solnR, span(listL, listR))
        return solution
```

Types of "Divide and Conquer"

Divide and Conquer

- Break the problem up into multiple subproblems of similar size and recursively solve
- **Examples:** Karatsuba, closest pair of points, Mergesort, Quicksort

Decrease and Conquer

- Break the problem into a single smaller subproblem and recursively solve
- **Examples:** Mission Impossible, Quickselect, binary search

Pattern So Far

Typically looking to divide the problem by some fraction (½, ¼ the size)

Not necessarily always the best!

- Sometimes, we can write faster algorithms by finding unbalanced splits