CS4102 Algorithms
Spring 2019

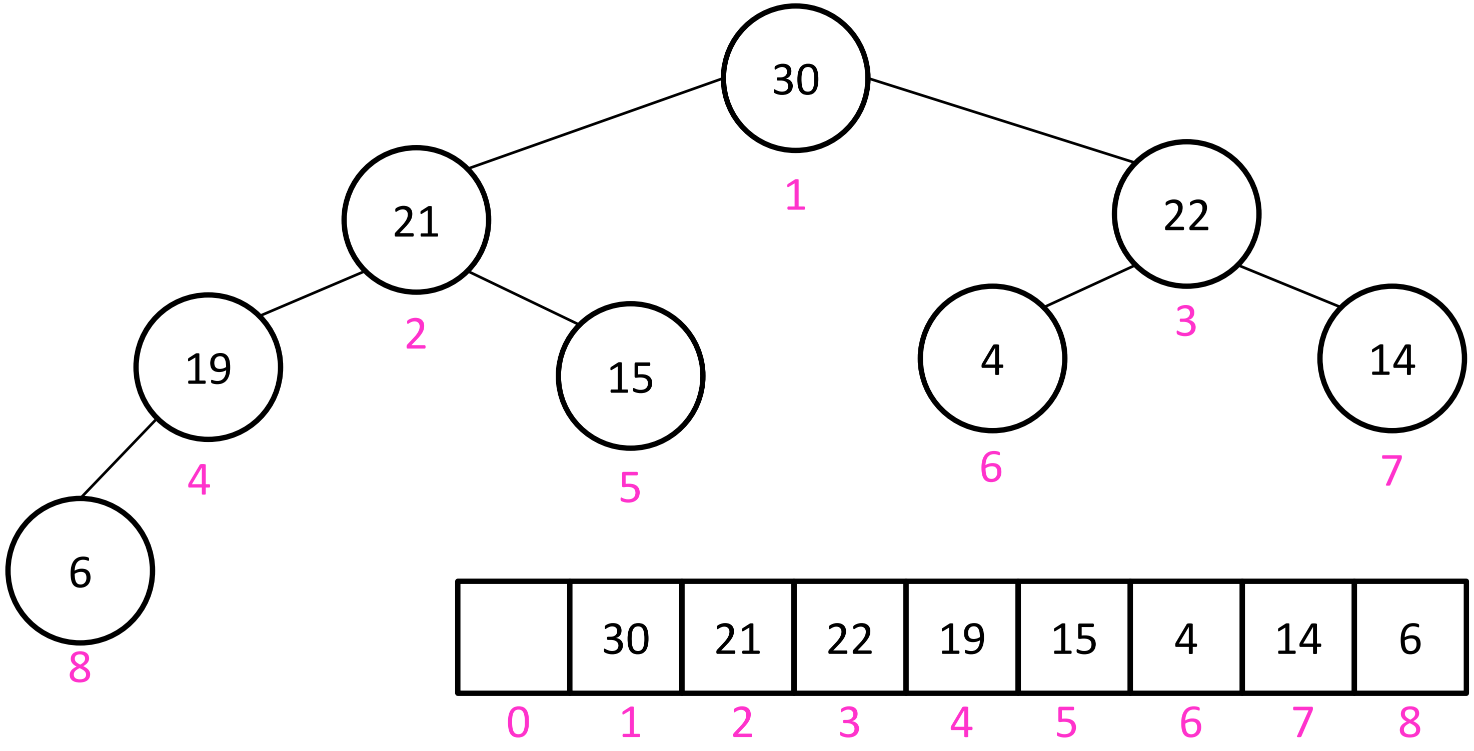
Warm up

Build a Max Heap from the following Elements:

4, 15, 22, 6, 18, 30, 14, 21

Heap

- Heap Property: Each node must be larger than its children



Today’s Keywords

- Sorting
- Quicksort
- Sorting Algorithm Characteristics
- Insertion Sort
- Bubble Sort
- Heap Sort
- Linear time Sorting
- Counting Sort
- Radix Sort

CLRS Readings

- Chapter 6
- Chapter 8

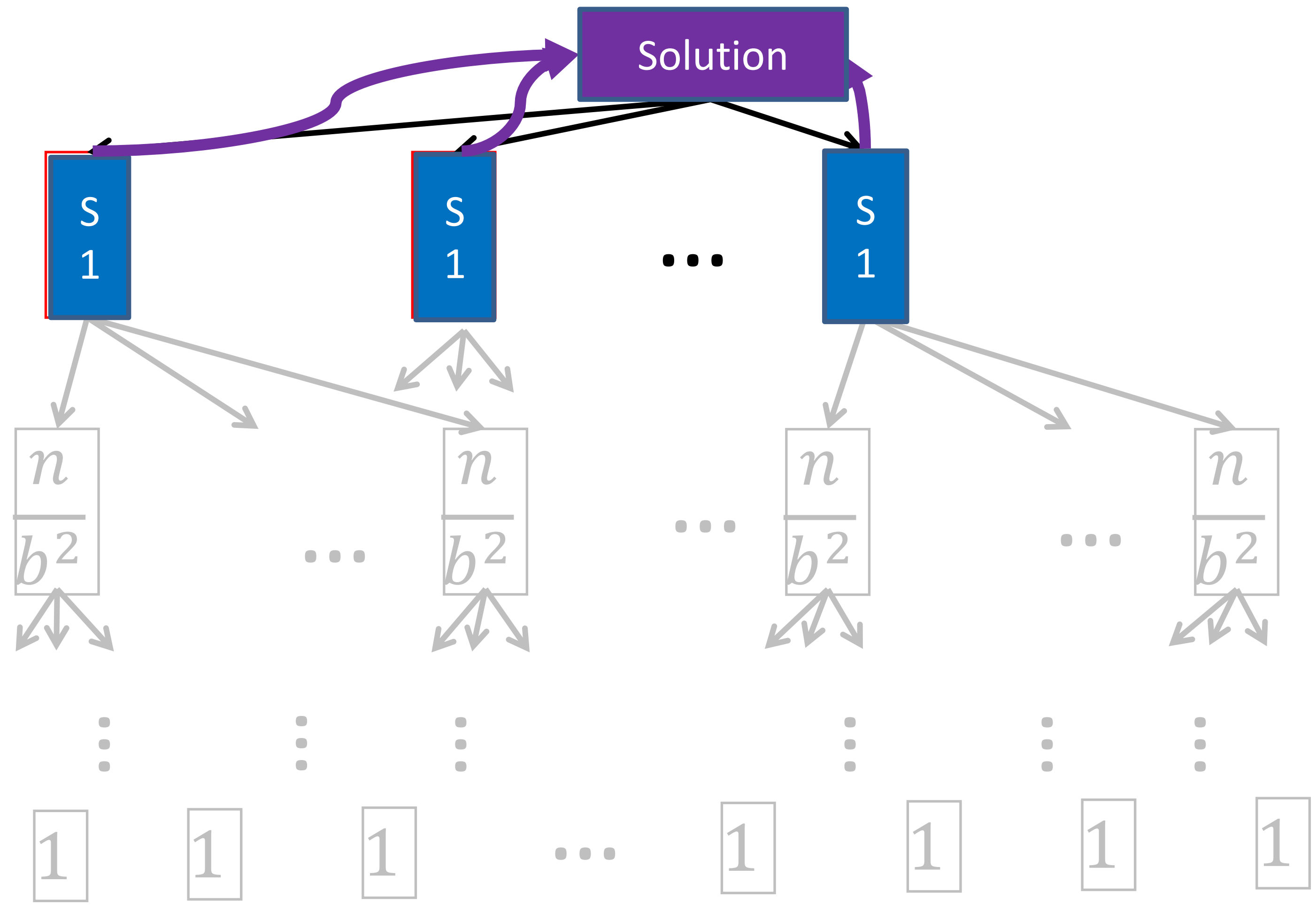
Homeworks

- HW3 due 11pm ~~Wednesday~~ Friday Feb. ~~20~~ 22
  - Divide and conquer
  - Written (use LaTeX!)
- HW4 coming on Wednesday
- Grading Notes
  - HW0 has been graded and released
  - HW1 grades (and solutions) released on Wednesday
  - HW2 is currently being graded (released tomorrow!)

Generic Divide and Conquer Solution

```
def myDCalgo(problem):
    if baseCase(problem):
        solution = solve(problem) #brute force if necessary
        return solution
    subproblems = Divide(problem)
    for subproblem of problem:
        subsolutions.append(myDCalgo(subproblem))
    solution = Combine(subsolutions)
    return solution
```

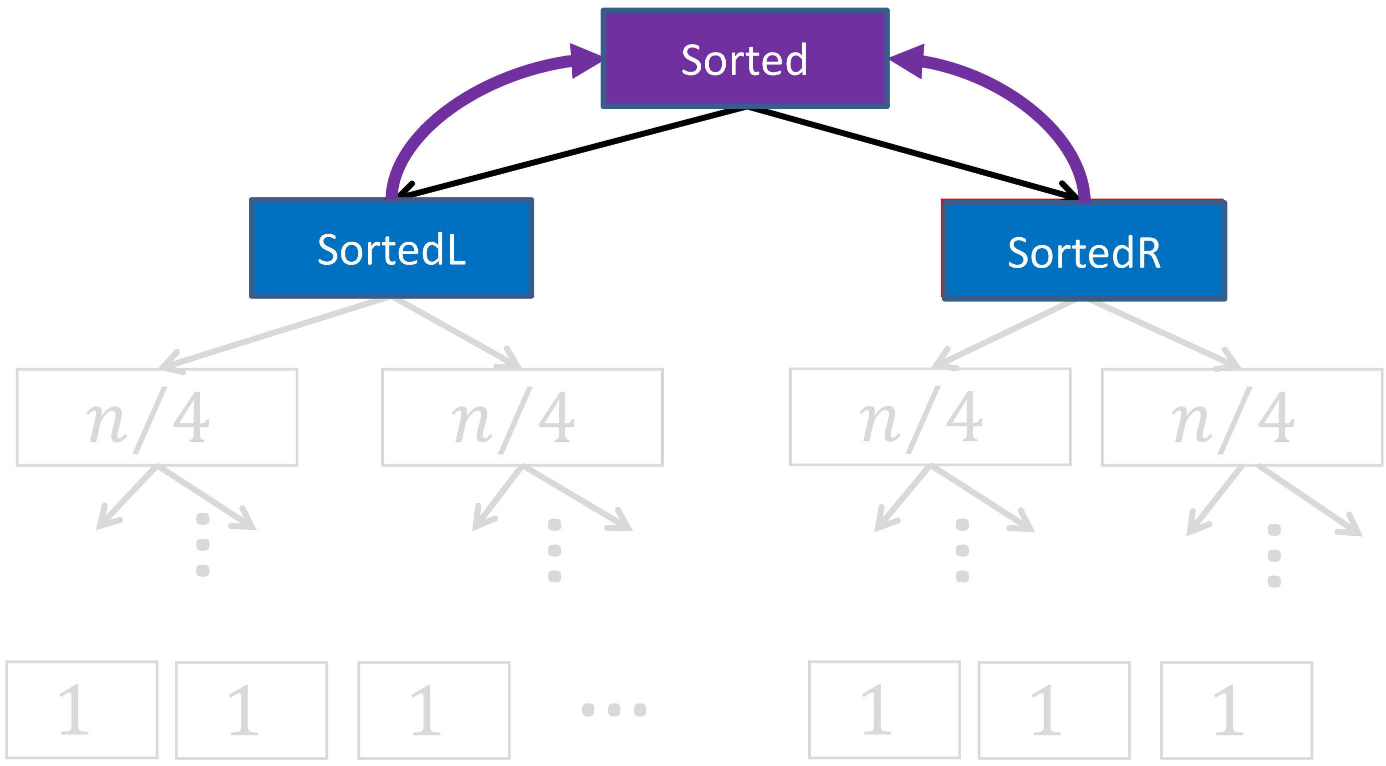
Generic Divide and Conquer Solution



MergeSort Divide and Conquer Solution

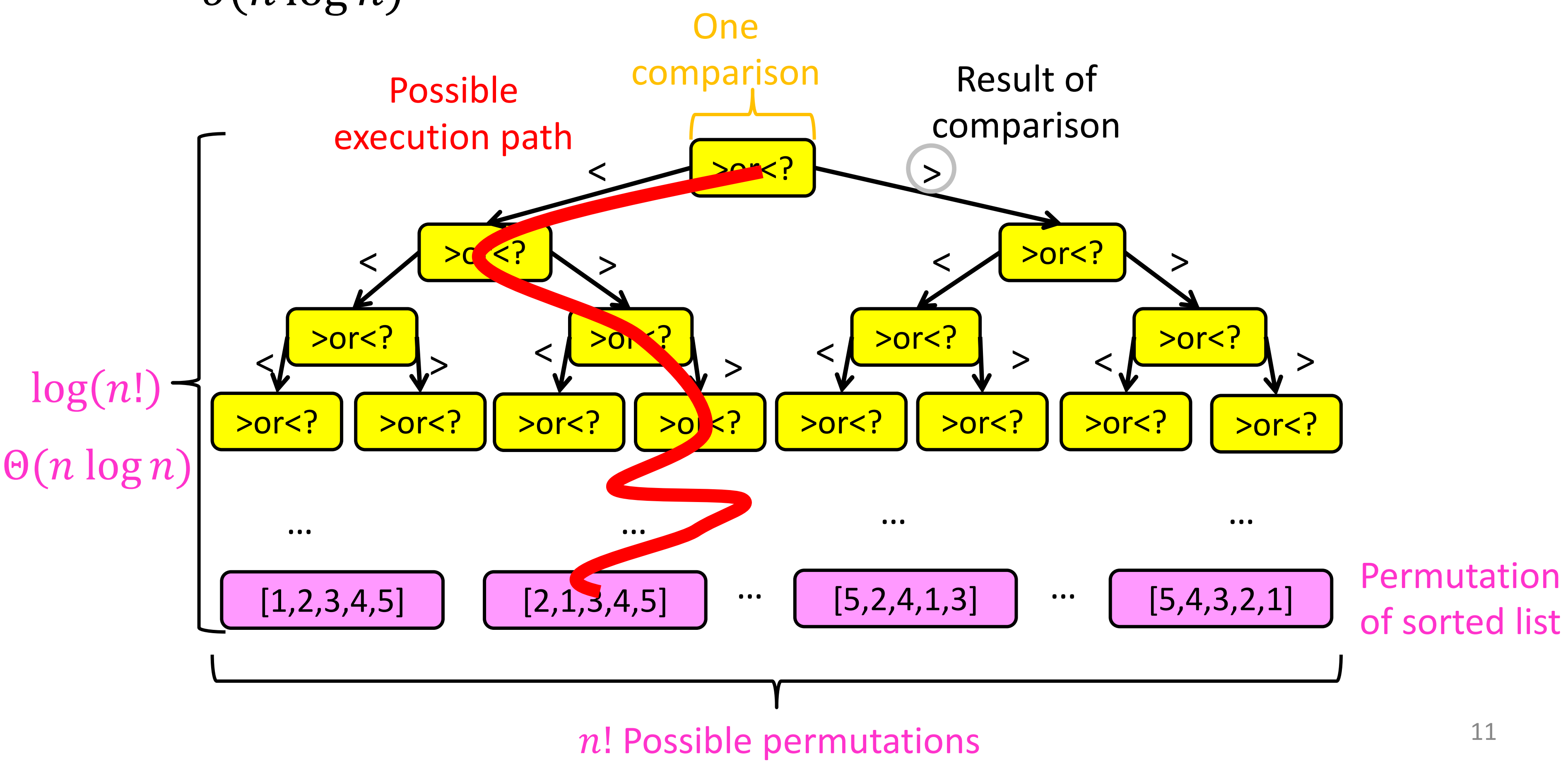
```
def mergesort(list):
    if list.length < 2:
        return list  #list of size 1 is sorted!
    {listL, listR} = Divide_by_median(list)
    for list in {listL, listR}:
        sortedSubLists.append(mergesort(list))
    solution = merge(sortedL, sortedR)
    return solution
```

MergeSort Divide and Conquer Solution



Strategy: Decision Tree

- Conclusion: Worst Case Optimal run time of sorting is $\Theta(n \log n)$
  - There is no (comparison-based) sorting algorithm with run time $o(n \log n)$



Sorting, so far

- Sorting algorithms we have discussed:
  - Mergesort $O(n \log n)$ Optimal!
  - Quicksort $O(n \log n)$ Optimal!
- Other sorting algorithms
  - Bubblesort $O(n^2)$
  - Insertionsort $O(n^2)$
  - Heapsort $O(n \log n)$ Optimal!

Speed Isn't Everything

- Important properties of sorting algorithms:
- Run Time
  - Asymptotic Complexity
  - Constants
- In Place (or In-Situ)
  - Done with only constant additional space
- Adaptive
  - Faster if list is nearly sorted
- Stable
  - Equal elements remain in original order
- Parallelizable
  - Runs faster with multiple computers

Mergesort

- **Divide**:
  - Break n-element list into two lists of $^n/_2$ elements
- **Conquer**:
  - If $n > 1$: Sort each sublist recursively
  - If $n = 1$: List is already sorted (base case)
- **Combine**:
  - Merge together sorted sublists into one sorted list

Run Time?

$\Theta(n \log n)$

Optimal!

In Place?	Adaptive?	Stable?
No	No	Yes! (usually)

Merge

- **Combine:** Merge sorted sublists into one sorted list
- We have:
  - 2 sorted lists (L_1, L_2)
  - 1 output list (L_{out})

While (L_1 and L_2 not empty):

    If $L_1[0] \leq L_2[0]$:

        L_{out}.append(L_1.pop())

    Else:

        L_{out}.append(L_2.pop())

L_{out}.append(L_1)

L_{out}.append(L_2)

Stable:
If elements are equal, leftmost comes first

Mergesort

- **Divide**:
  - Break n-element list into two lists of $^n/_2$ elements
- **Conquer**:
  - If $n > 1$: Sort each sublist recursively
  - If $n = 1$: List is already sorted (base case)
- **Combine**:
  - Merge together sorted sublists into one sorted list

Run Time?

$\Theta(n \log n)$

Optimal!

In Place?	Adaptive?	Stable?	Parallelizable?
No	No	Yes! (usually)	Yes!

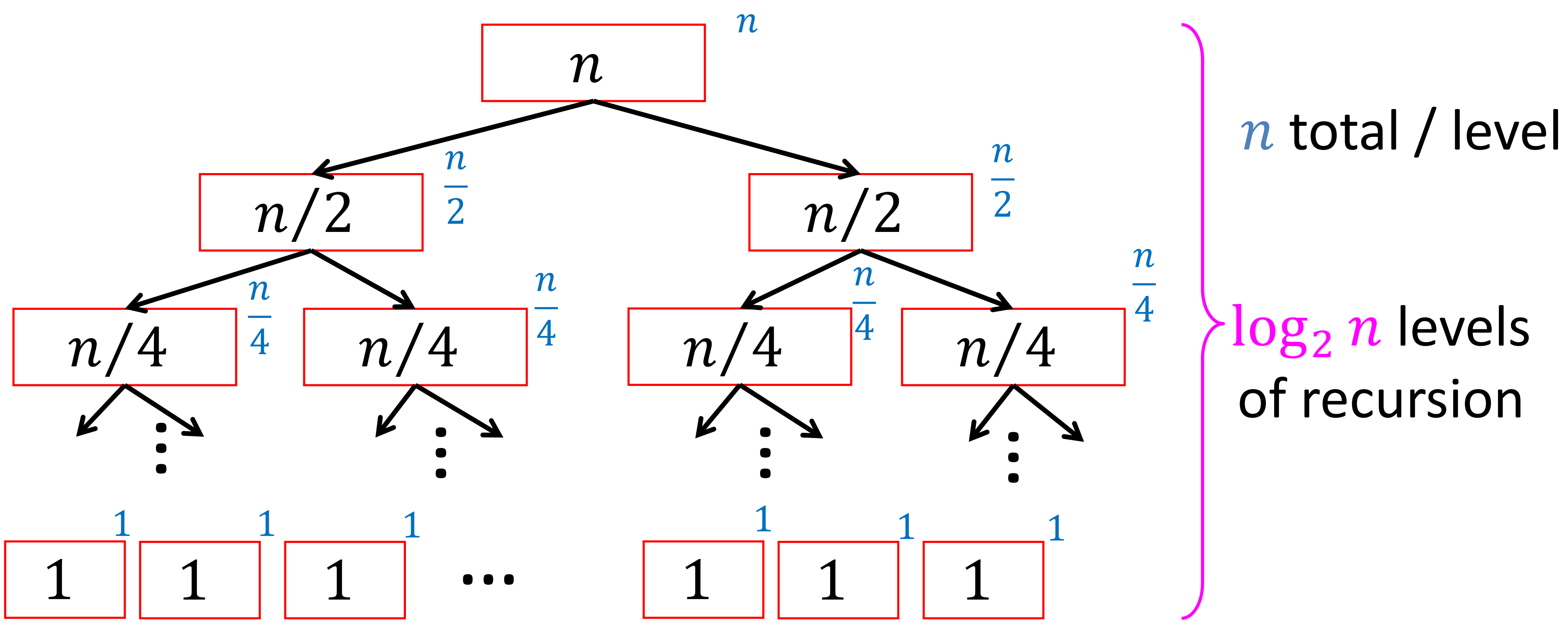
Mergesort

- **Divide**:
  - Break n-element list into two lists of $^n/_2$ elements
- **Conquer**:
  - If $n > 1$:
    - Sort each sublist recursively
  - If $n = 1$:
    - List is already sorted (base case)
- **Combine**:
  - Merge together sorted sublists into one sorted list

Parallelizable: Allow different machines to work on each sublist

Mergesort (Sequential)

$$T(n) = 2T(\frac{n}{2}) + n$$

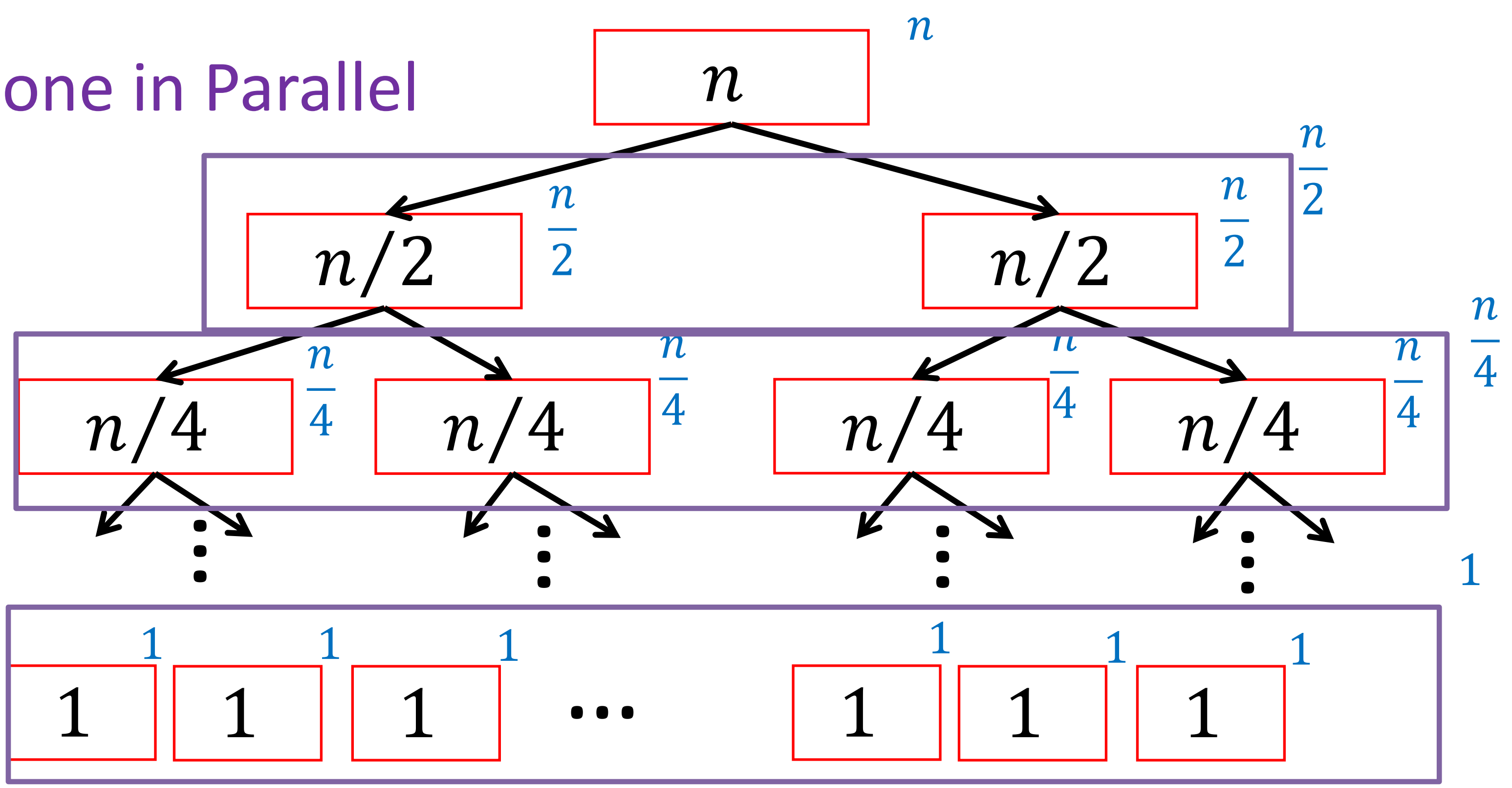


Run Time: $\Theta(n \log n)$

Mergesort (Parallel)

$$T(n) = T(\frac{n}{2}) + n$$

Done in Parallel



Run Time: $\Theta(n)$

Quicksort

- Idea: pick a partition element, recursively sort two sublists around that element
- Divide: select an element p, Partition(p)
- Conquer: recursively sort left and right sublists
- Combine: Nothing!

Run Time?

$\Theta(n \log n)$

(almost always)

Better constants than Mergesort

In Place?

kinda

Uses stack for recursive calls

Adaptive?

No!

Stable?

No

Parallelizable?

Yes!

Bubble Sort

- Idea: March through list, swapping adjacent elements if out of order, repeat until sorted

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

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

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

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

Bubble Sort

- Idea: March through list, swapping adjacent elements if out of order, repeat until sorted

Run Time?

$\Theta(n^2)$

Constants worse than Insertion Sort

In Place?

Yes

Adaptive?

Kinda

"Compared to straight insertion […], bubble sorting requires a more complicated program and takes about twice as long!" –Donald Knuth

Bubble Sort is "almost" Adaptive

- Idea: March through list, swapping adjacent elements if out of order

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

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

Only makes one "pass"

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

After one "pass"

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

Requires n passes, thus is $O(n^2)$

Bubble Sort

- Idea: March through list, swapping adjacent elements if out of order, repeat until sorted

Run Time?

$\Theta(n^2)$

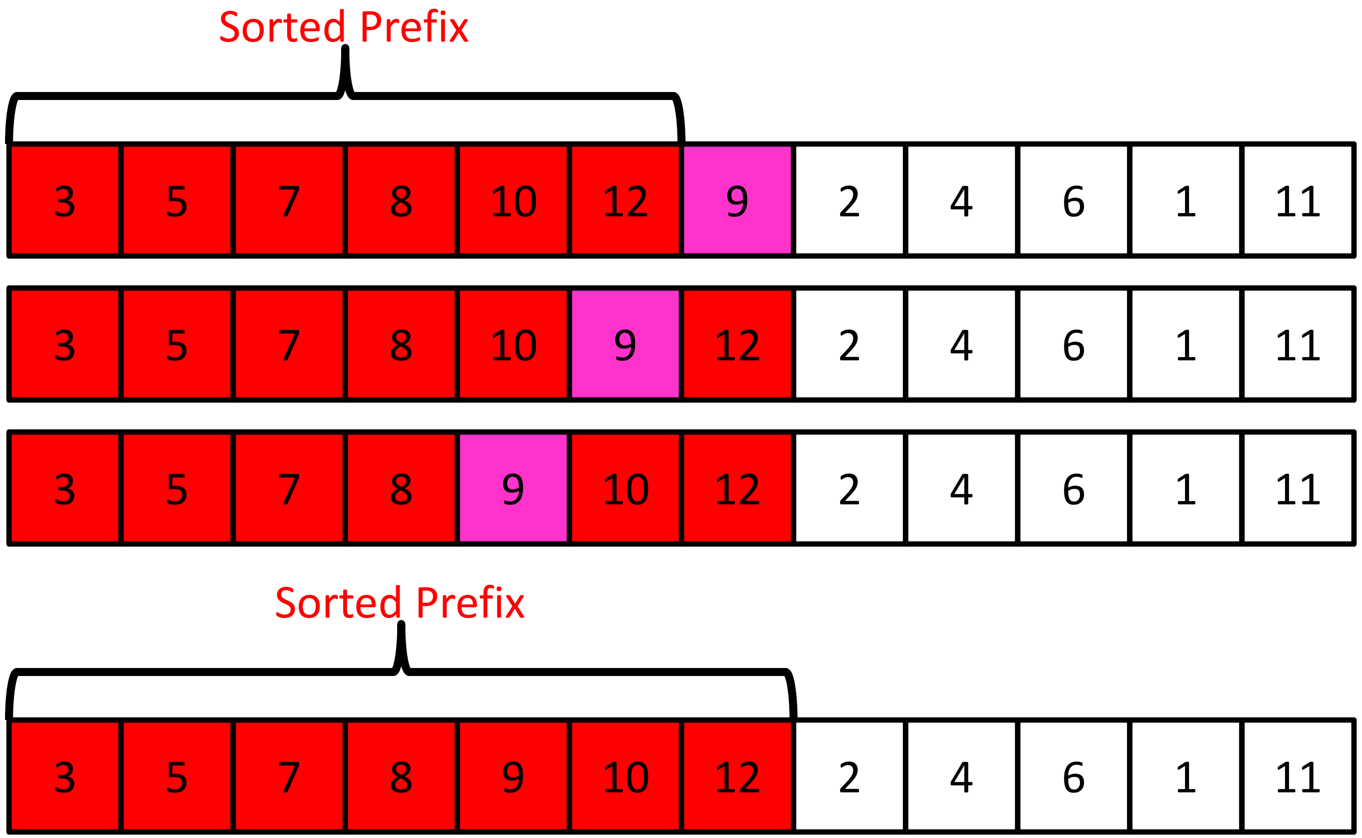
Constants worse than Insertion Sort

In Place?	Adaptive?	Stable?	Parallelizable?
Yes!	~~Kinda~~ Not really	Yes	No

"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

Insertion Sort

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



Insertion Sort

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

Run Time?

$\Theta(n^2)$

(but with very small constants)

Great for short lists!

In Place?

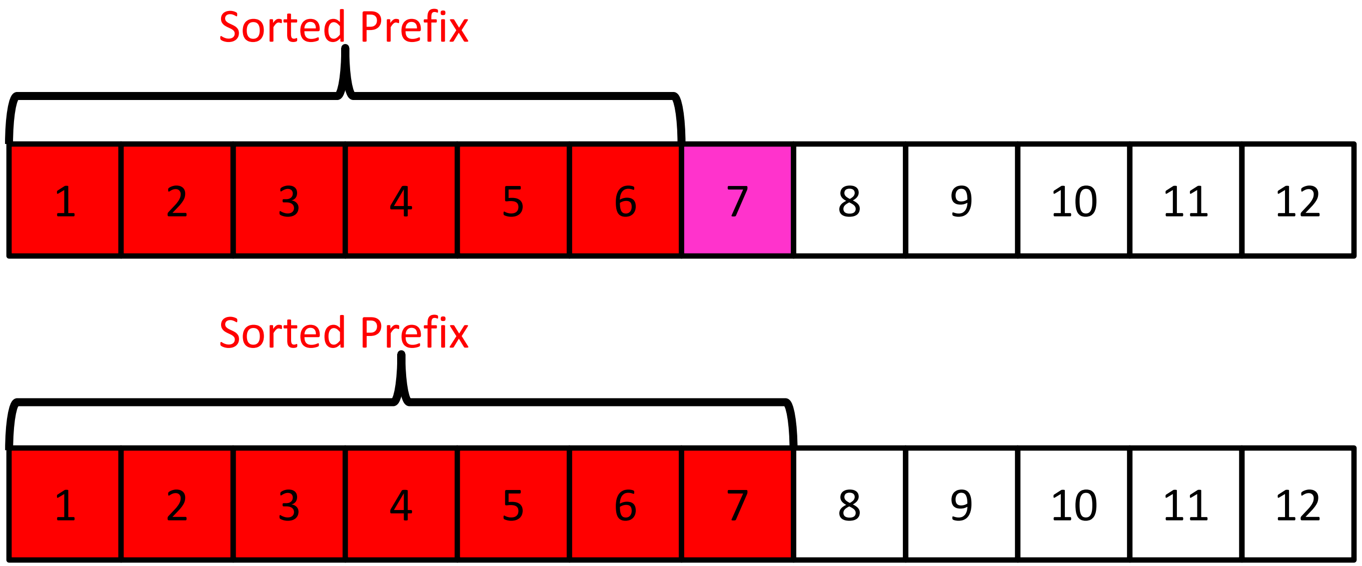
Yes!

Adaptive?

Yes

Insertion Sort is Adaptive

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



Only one comparison needed per element! Runtime: $O(n)$

Insertion Sort

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

In Place?

Yes!

Adaptive?

Yes

Stable?

Yes

Run Time?

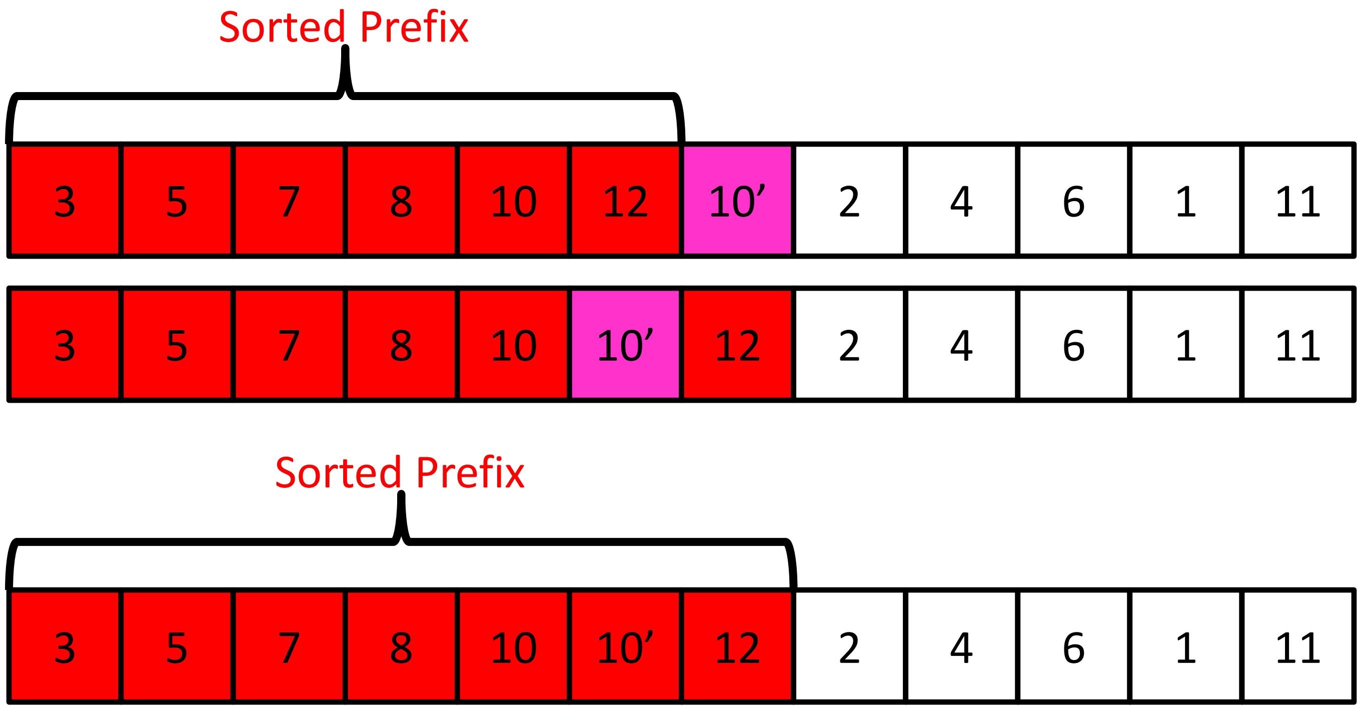
$$\Theta(n^2)$$

(but with very small constants)

Great for short lists!

Insertion Sort is Stable

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



The “second” 10 will stay to the right

Insertion Sort

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

Run Time?

$\Theta(n^2)$

(but with very small constants)

Great for short lists!

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

Can sort a list as it is received, i.e., don't need the entire list to begin sorting

Online?

Yes

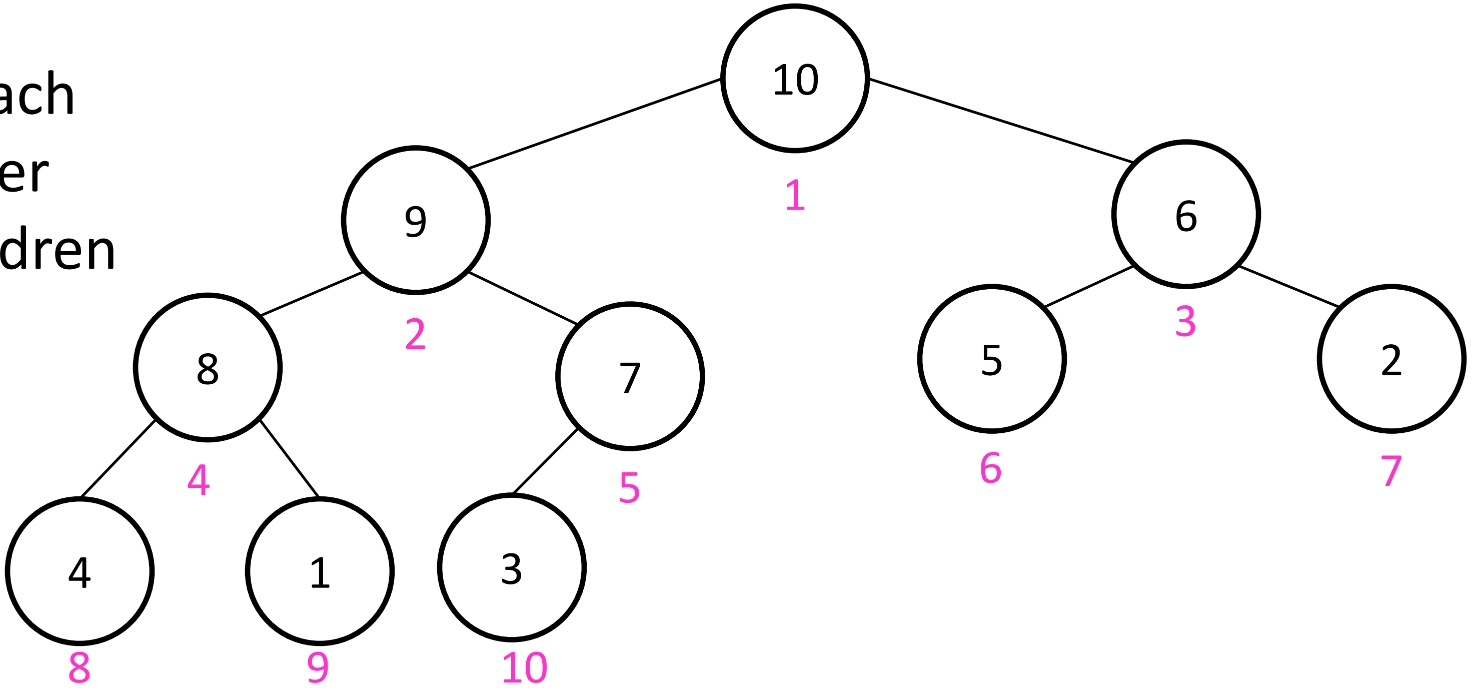
"All things considered, it's actually a pretty good sorting algorithm!" –Nate Brunelle

Heap Sort

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

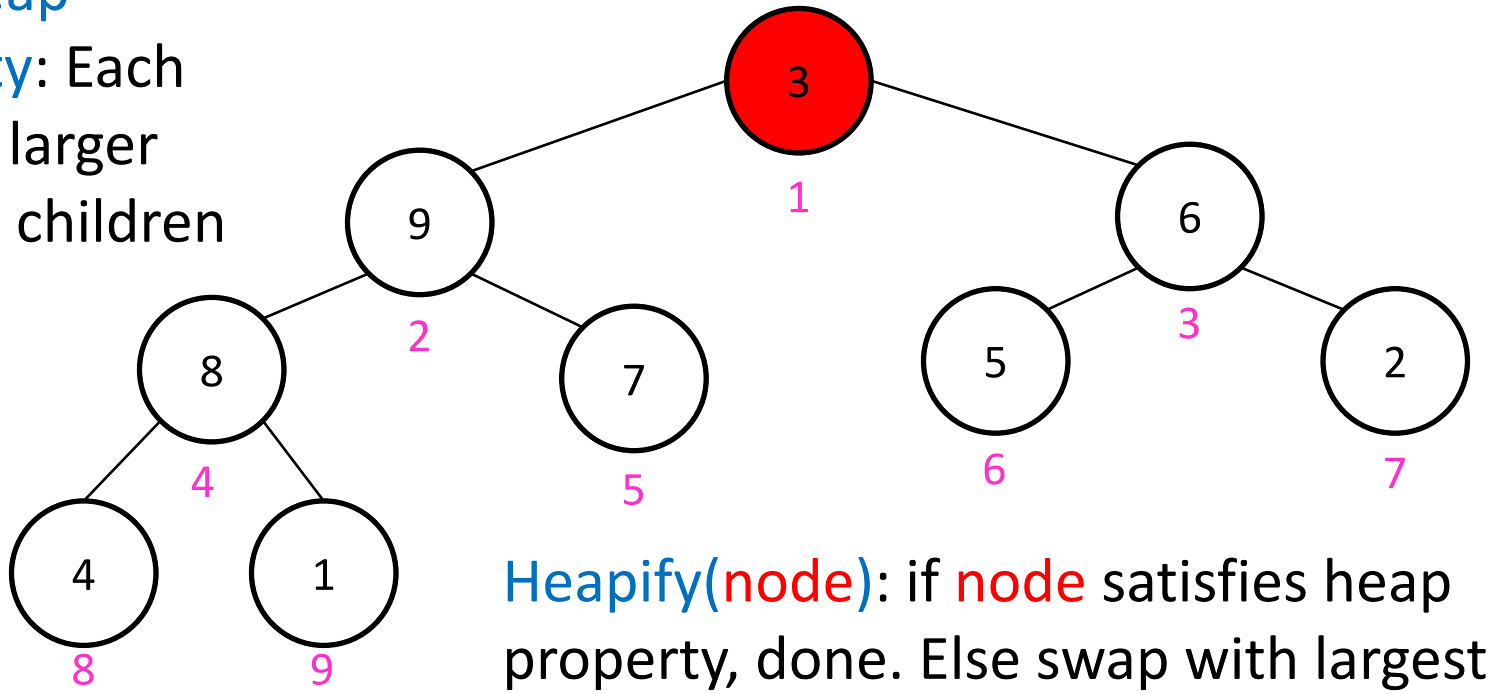


Heap Sort

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

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

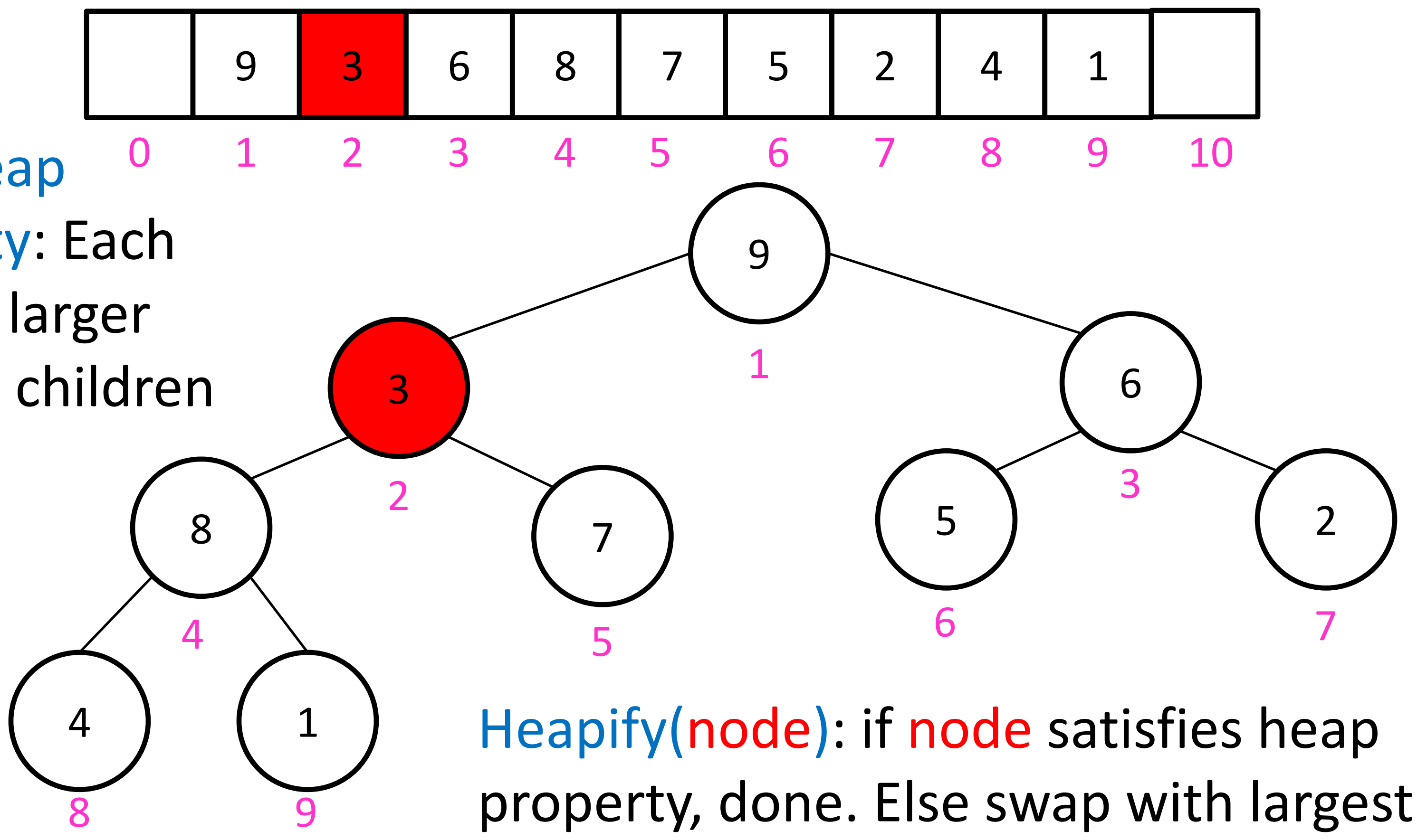
Max Heap Property: Each node is larger than its children



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

Heap Sort

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

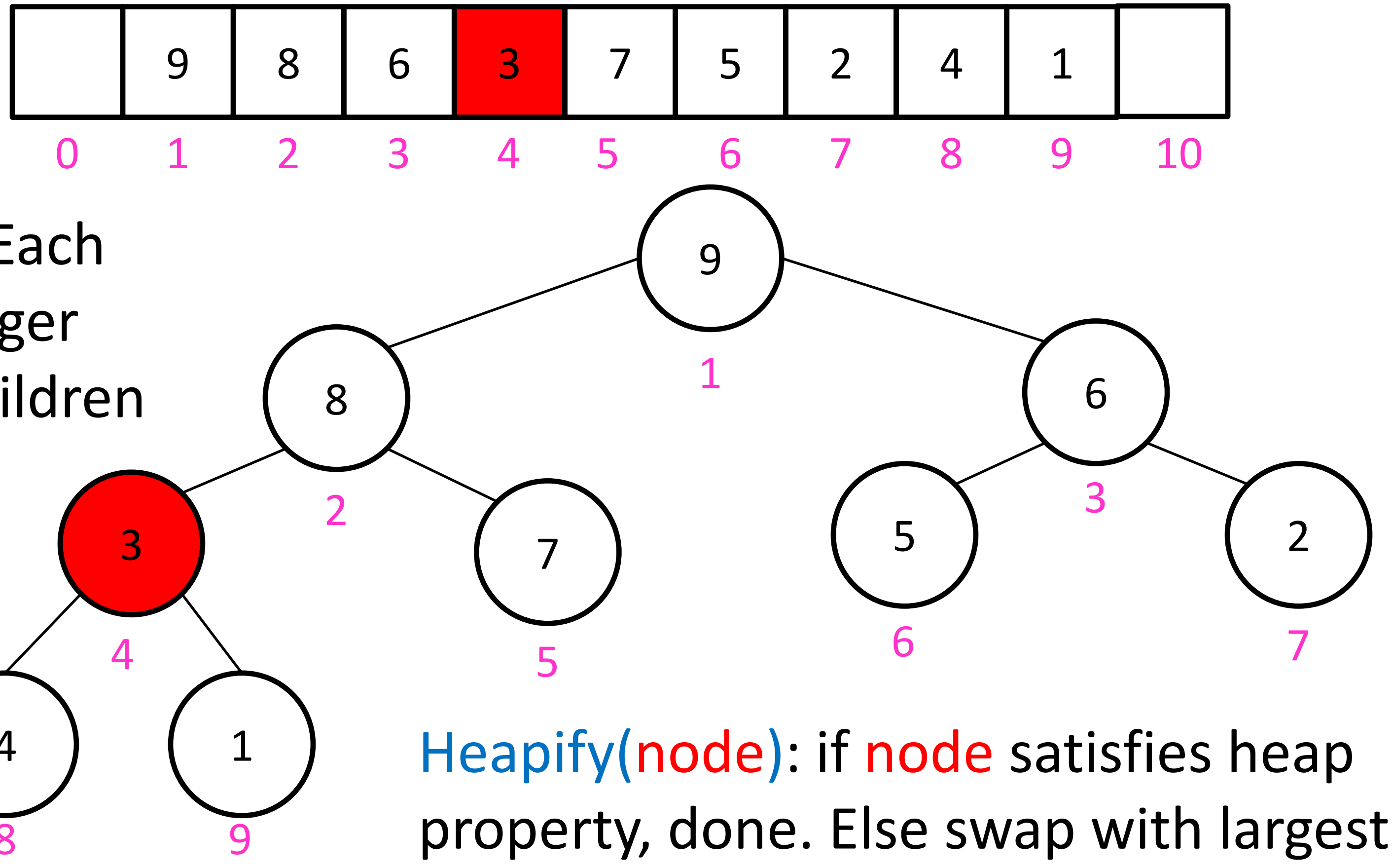


Max Heap Property: Each node is larger than its children

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

Heap Sort

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

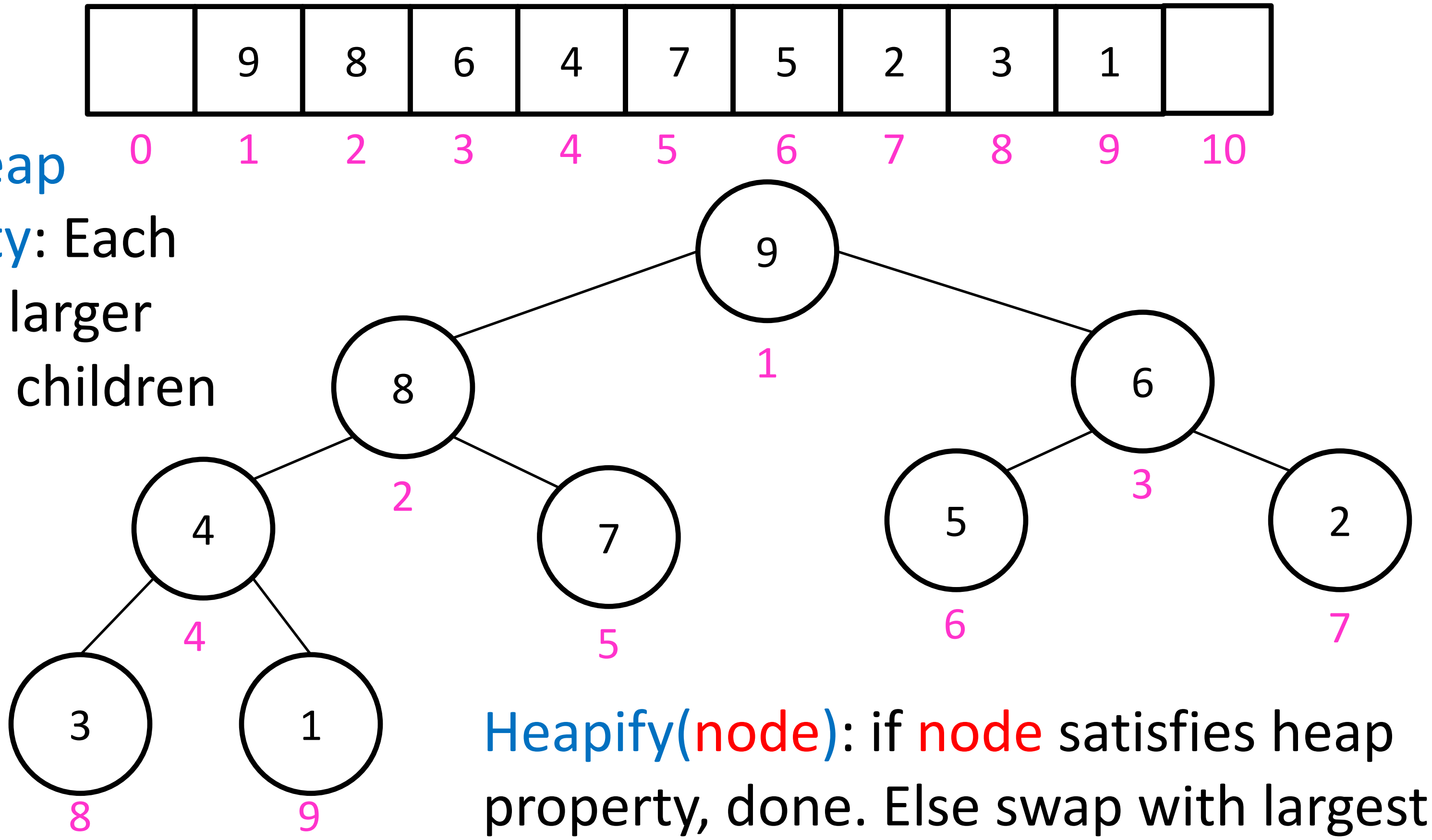


Max Heap Property: Each node is larger than its children

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

Heap Sort

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



Max Heap Property: Each node is larger than its children

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

Heap Sort

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

Run Time?

$\Theta(n \log n)$

Constants worse than Quick Sort

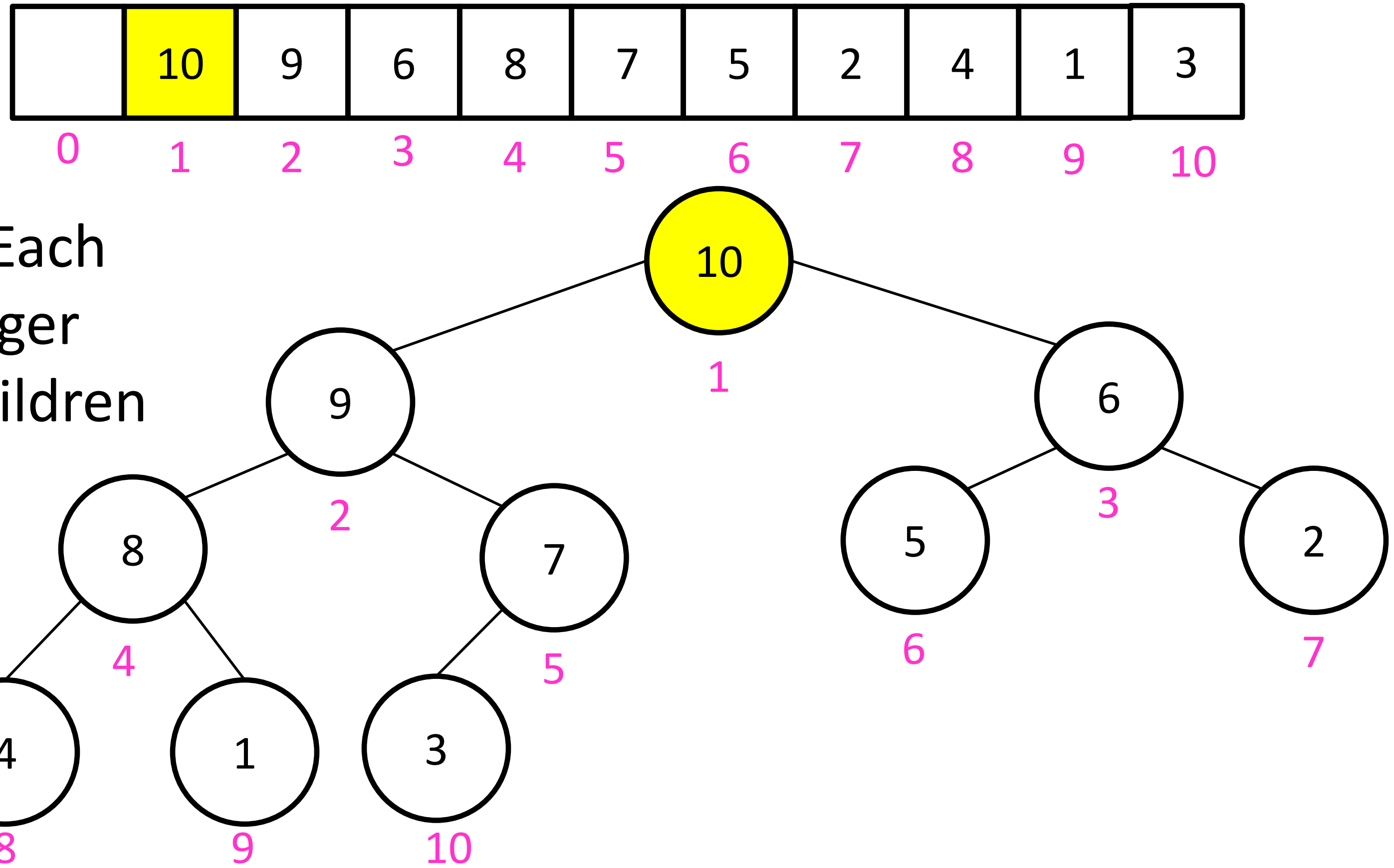
In Place?

Yes!

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

In Place Heap Sort

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



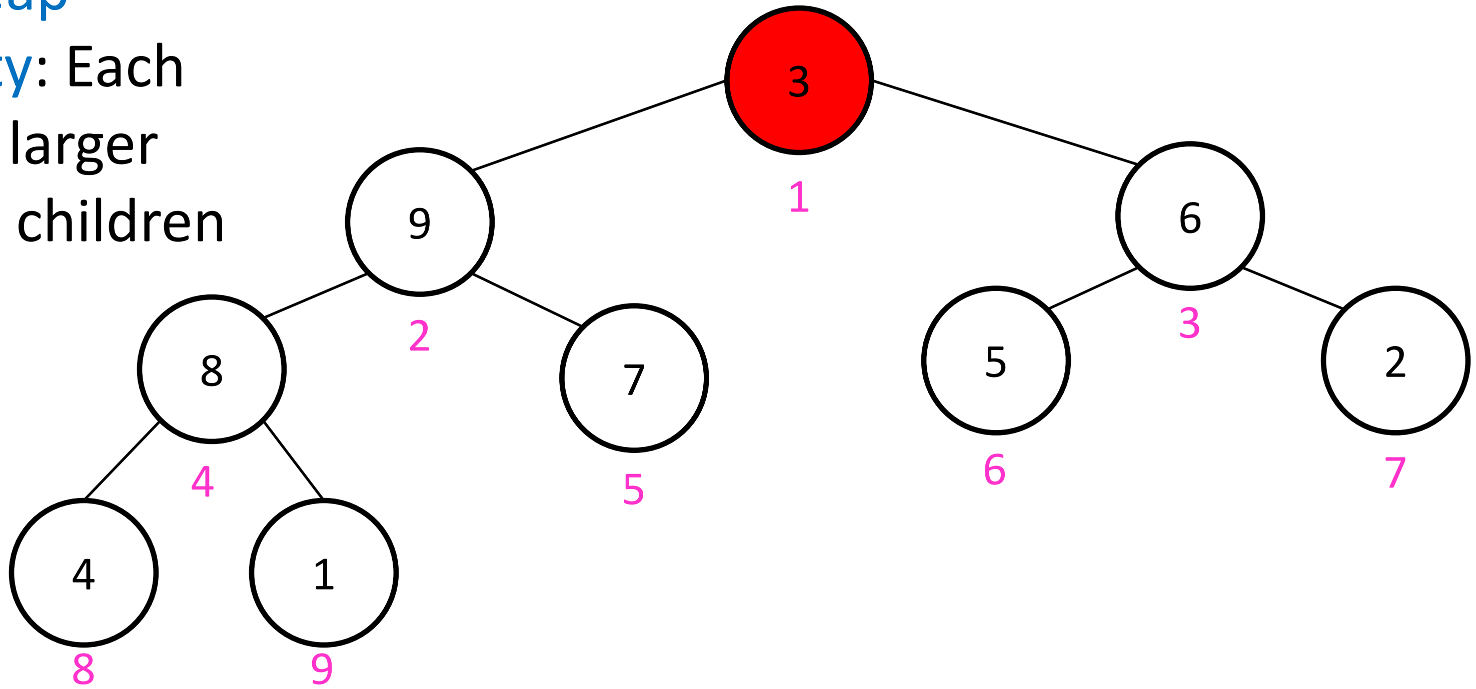
Max Heap Property: Each node is larger than its children

In Place Heap Sort

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

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

Max Heap Property: Each node is larger than its children

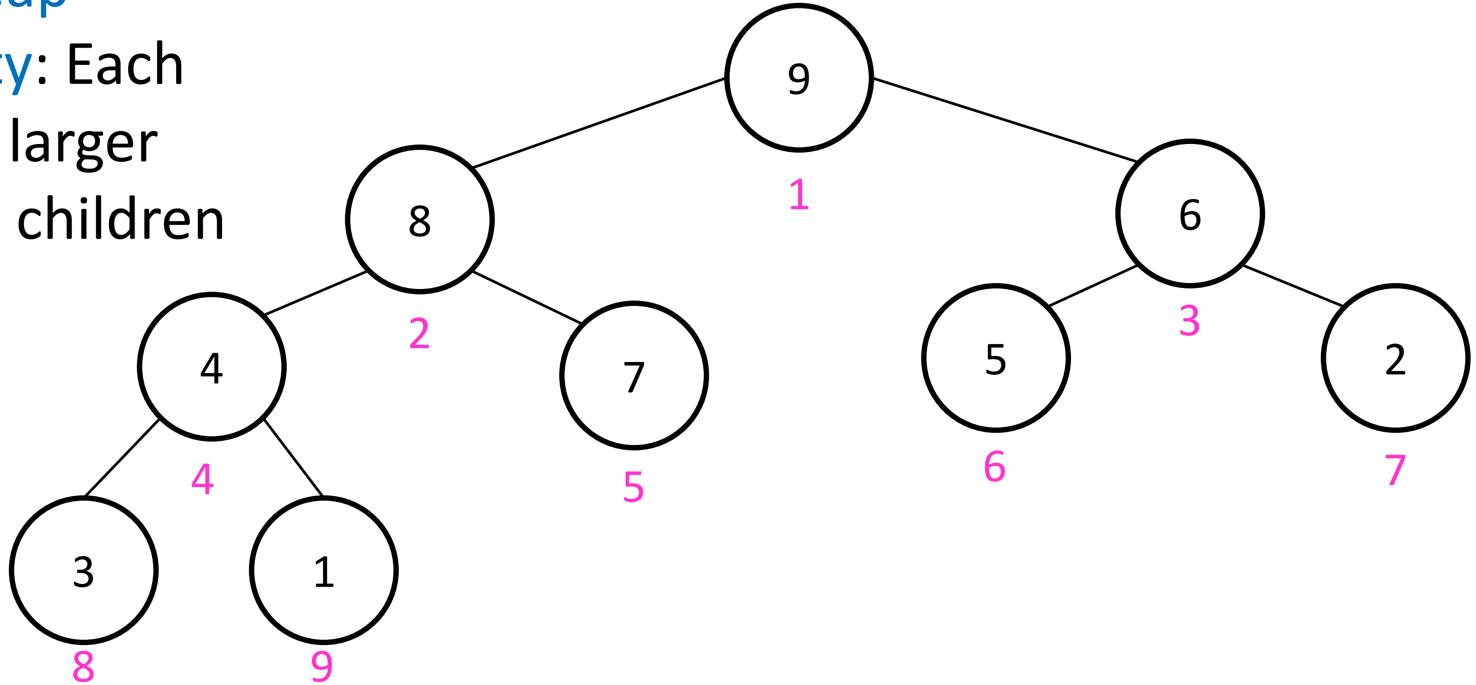


In Place Heap Sort

- 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
	9	8	6	4	7	5	2	3	1	10

Max Heap Property: Each node is larger than its children

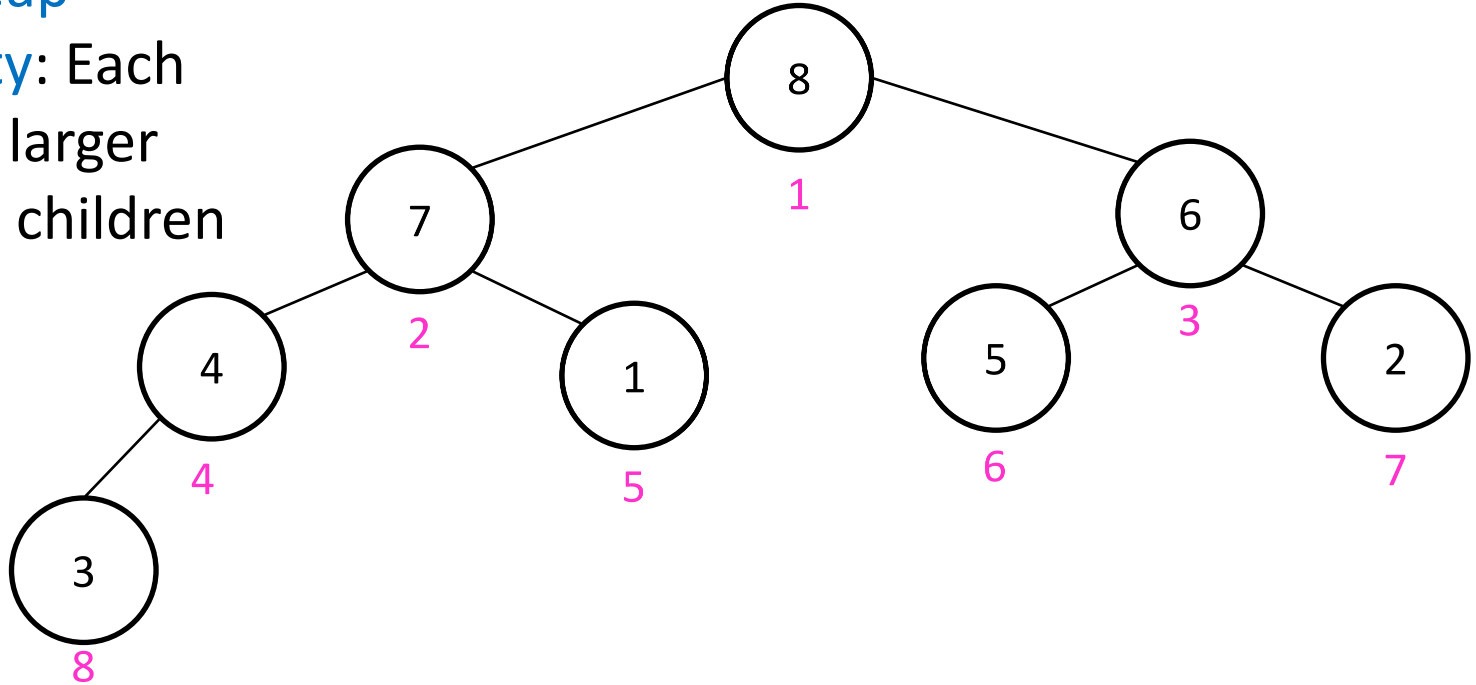


In Place Heap Sort

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

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

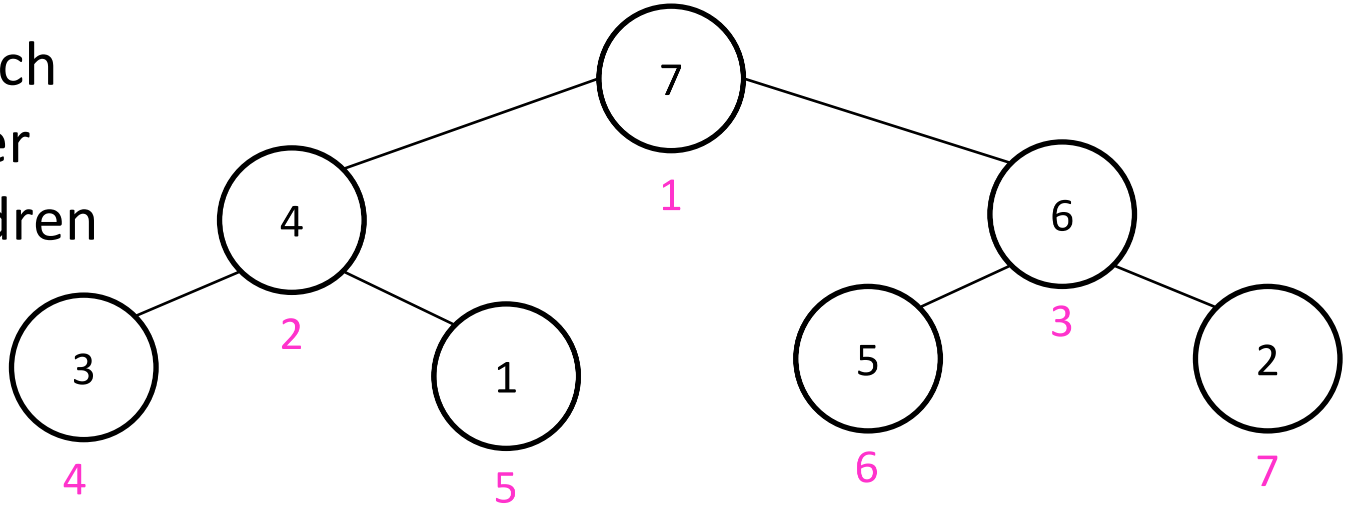


In Place Heap Sort

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

	7	4	6	3	1	5	2	8	9	10
0	1	2	3	4	5	6	7	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 Right-to-Left

Run Time?

$\Theta(n \log n)$

Constants worse than Quick Sort

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

Sorting in Linear Time

- Cannot be comparison-based
- Need to make some sort of assumption about the contents of the list
  - Small number of unique values
  - Small range of values
  - Etc.

Counting Sort

- Idea: Count how many things are less than each element

$L =$

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

1. Range is $[1, k]$ (here $[1,6]$)
   make an array C of size k
   populate with counts of each value

For i in L:
    $C[L[i]]{+}{+}$

$C =$

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

running sum

2. Take "running sum" of C
   to count things less than each value

For $i = 1$ to $\text{len}(C)$:
    $C[i] = C[i-1] + C[i]$

$C =$

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

To sort: last item of value 3 goes at index 4

Counting Sort

- Idea: Count how many things are less than each element

$L =$

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

$C =$

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

Last item of value 6 goes at index 8

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

For $i = \mathrm{len}(L)$ downto 1:

$$B\left[C\left[L[i]\right]\right] = L[i]$$

$$C\left[L[i]\right] = C\left[L[i]\right] - 1$$

$B =$

							6
1	2	3	4	5	6	7	8

Counting Sort

- Idea: Count how many things are less than each element

$L =$

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

$C =$

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

Last item of value 1 goes at index 2

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

For $i = \text{len}(L)$ downto 1:

$$B\left[C\left[L[i]\right]\right] = L[i]$$

$$C\left[L[i]\right] = C\left[L[i]\right] - 1$$

$B =$

	1						6
1	2	3	4	5	6	7	8

Run Time: $O(n+k)$

Memory: $O(n+k)$

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

12 Exabytes

Radix Sort

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

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

Place each element into a "bucket" according to its 1's place

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

Radix Sort

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

Place each element into a “bucket” according to its 10’s place

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
800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999

Radix Sort

- Idea: Stable sort on 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

Place each element into a “bucket” according to its 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

Run Time: $O\big(d(n+b)\big)$

d = digits in largest value

b = base of representation