ALGORITHMS

(CSCI 2200)

Week 4

Heaps and Heapsort

Laura Toma
Bowdoin College

Week 4 Announcements

Week 4 Overview

- Two new sorting algorithms
- Heapsort
    - The heap (min-heaps and max-heaps)
    - Operations: Insert, Delete-Min, Heapify, Buildheap
- Quicksort / Randomized quicksort
    - Partition

The Priority Queue

- A container of objects that have keys (or: priorities)
- Supported operations on a Min-pqueue
  - **Insert**: insert a new object to the queue
  - **Delete-Min:** delete the object with a minimum key value
- Max-pqueues are symmetrical

PQueue Applications

- Sorting
    - Insert the objects into a priority queue; then call Delete-Min to put the elements in order
    - Run time: n x Insert + n x DeleteMin
- Event managers
    - objects = the events
    - key = time the event is scheduled to occur
    - DeleteMin: gives the next scheduled event
- Process scheduling
    - objects = processes waiting to be scheduled on the processor
    - key = priority of the process
    - DeleteMax: gives the next process to be scheduled

The binary heap

The heap

- The (binary) heap is standard implementation of a PQ

Min-heaps	Max-heaps	
Operations:	Operations:	
• Insert(A, element e)	• Insert(A, element e)	Run time: O(lg n)
• DeleteMin(A)	• DeleteMax(A)	Run time: O(lg n)
• Heapify(A, i)	• Heapify(A, i)	Run time: O(lg n)
• Buildheap(A)	• Buildheap(A)	O(n)

symmetrical

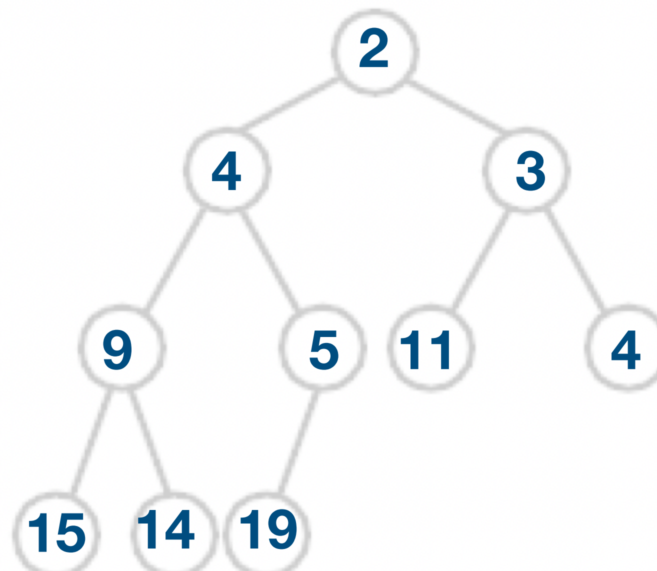
The min-heap

An array: viewed as corresponding to a complete binary tree (except last level, which is filled from left to right)

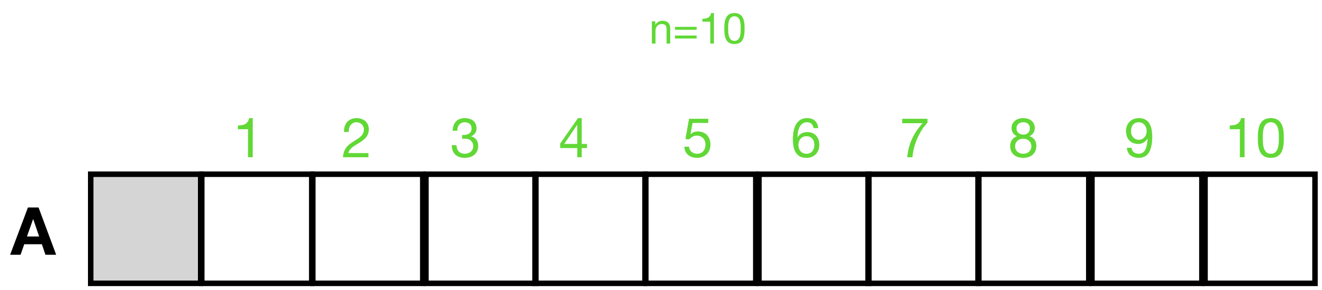
Heap property: for all nodes v, priority(v) <= priority of children(v)

A	2	4	3	9	5	11	4	15	14	19

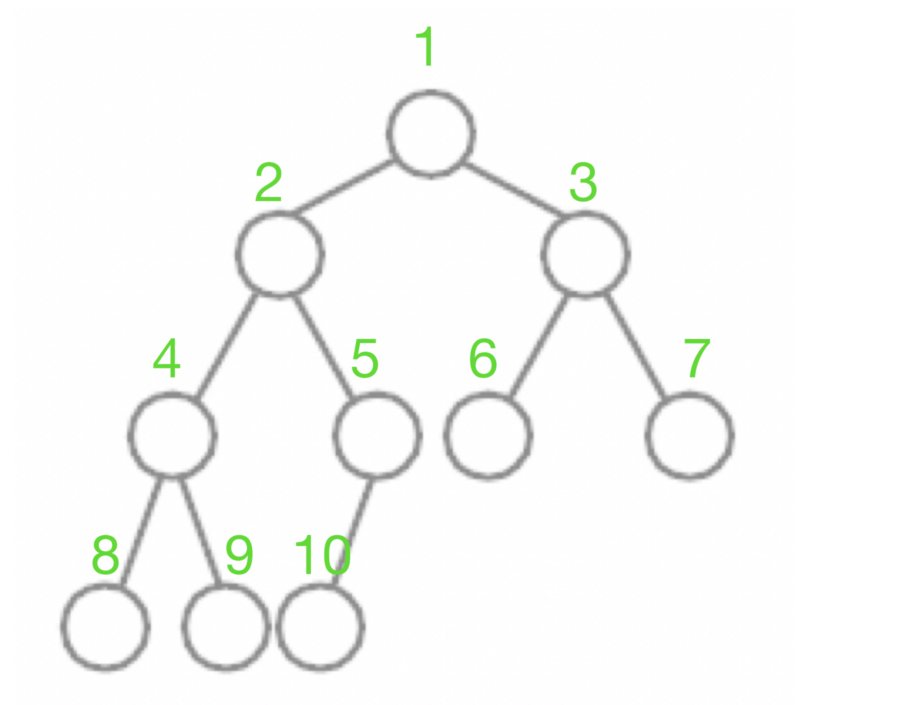
n=10



Properties



1. The smallest element is in the root
2. The height of a heap of n elements is $\Theta(\lg n)$
3. The indices of the children and parent of a node can be calculated (without storing pointers). For node at index i:
   - left(i) = 2 i
   - right(i) = 2i+1
   - parent(i) = i/2



Operations supported by a min-heap

- **insert(A, e):** A is a heap; Insert element e and maintain A as a heap.
- **deleteMin(A):** A is a heap; delete the min element in A and return it. Maintain A as a heap.
- **peak(A):** A is a heap; return the min element in A

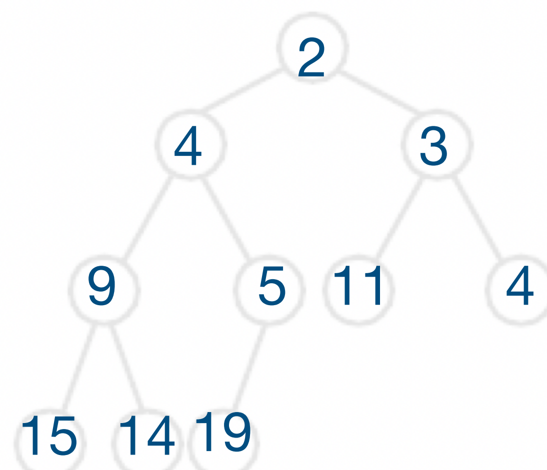
- Supporting operation
  - **heapify(A, i):** left(i) is a heap and right(i) is a heap. Make a heap under i.

peak(A)

n=10

A		2	4	3	9	5	11	4	15	14	19

this is the smallest element

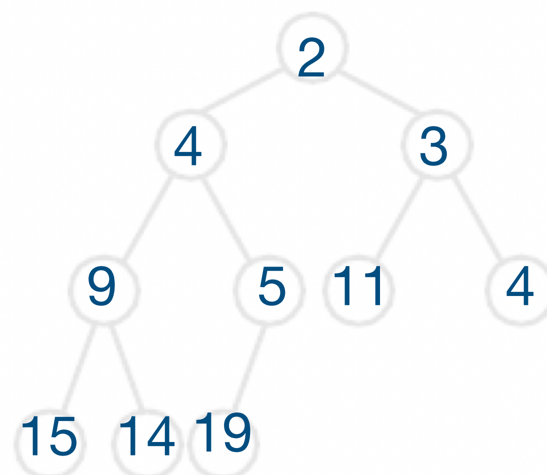


Inserting in a heap

n=10

A											
		2	4	3	9	5	11	4	15	14	19

Insert(A, 3)



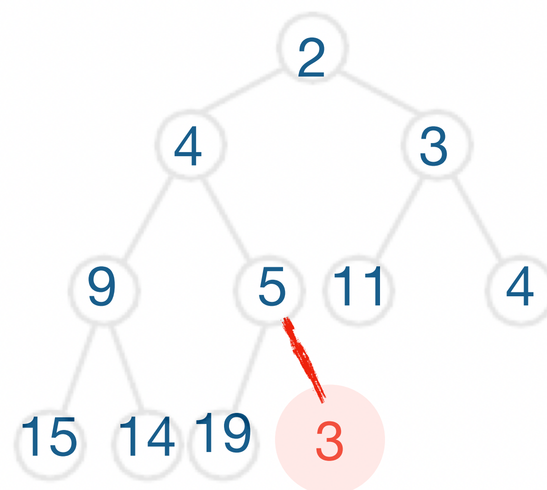
Inserting in a heap

n=11

~~n=10~~

A		2	4	3	9	5	11	4	15	14	19	3

Insert(A, 3)

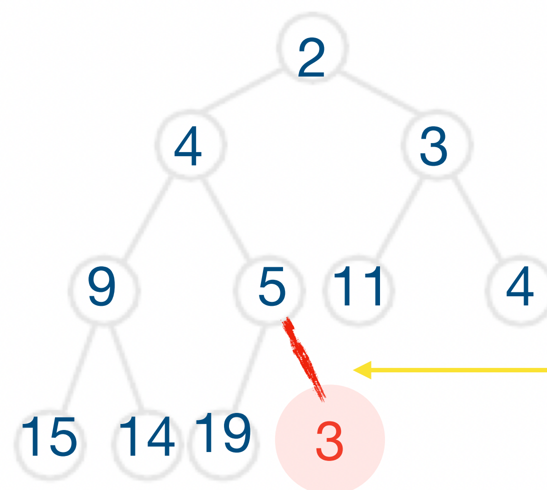


Inserting in a heap

n=11

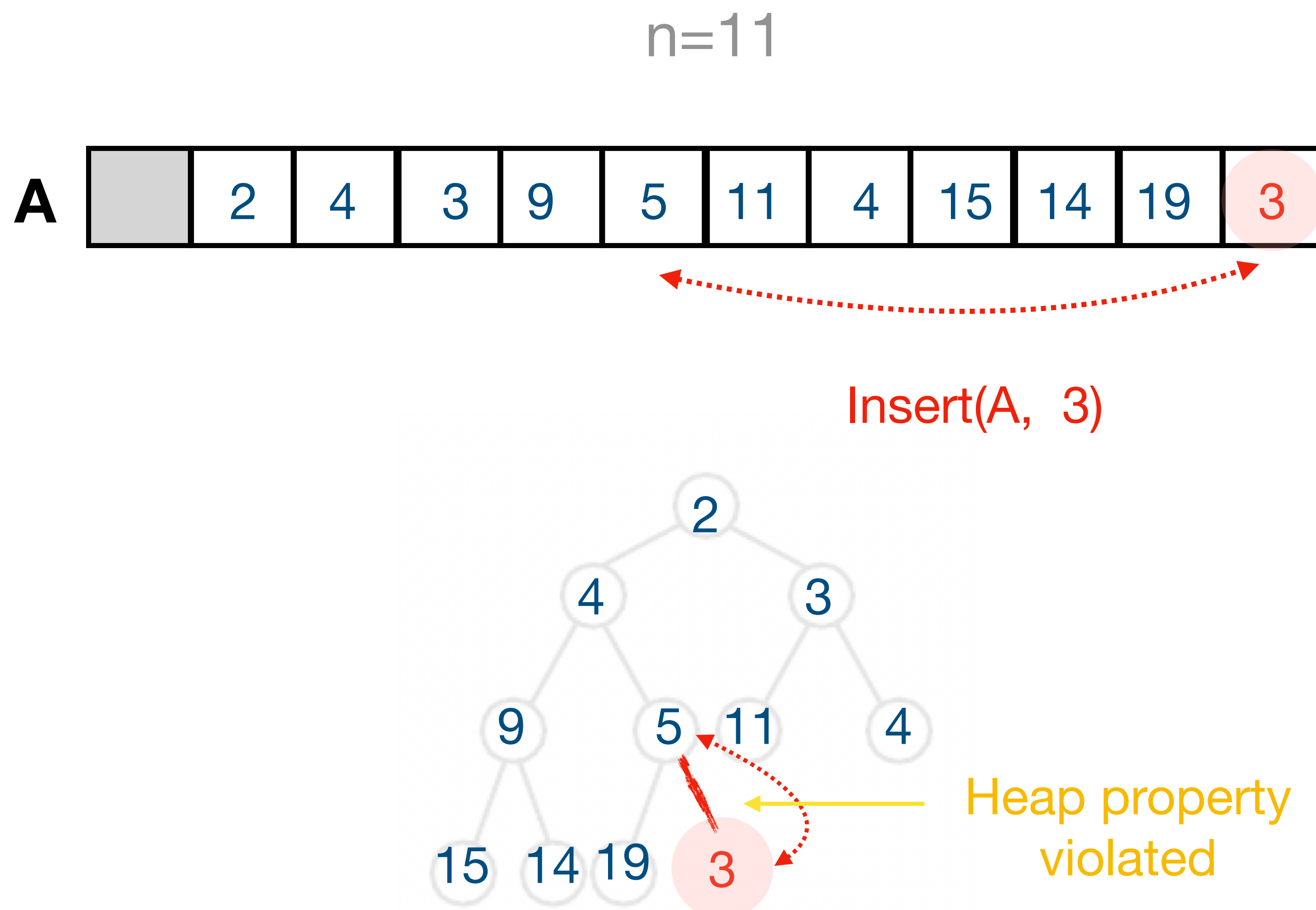
A												
		2	4	3	9	5	11	4	15	14	19	3

Insert(A, 3)



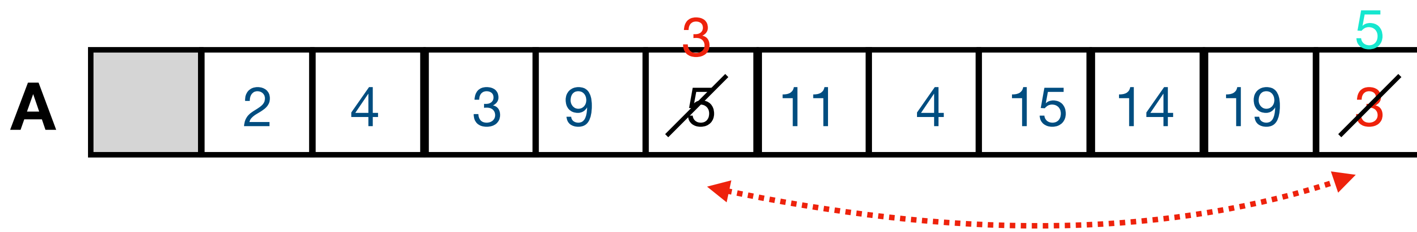
Heap property violated

Inserting in a heap

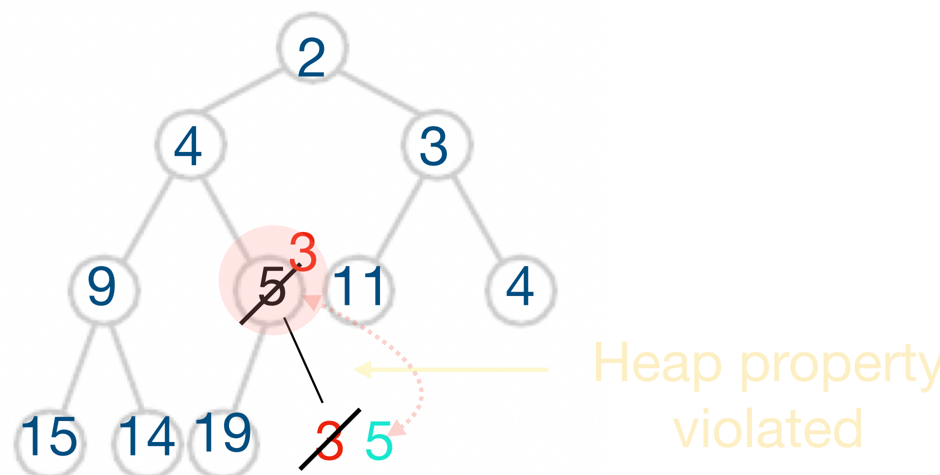


Inserting in a heap

n=11



Insert(A, 3)

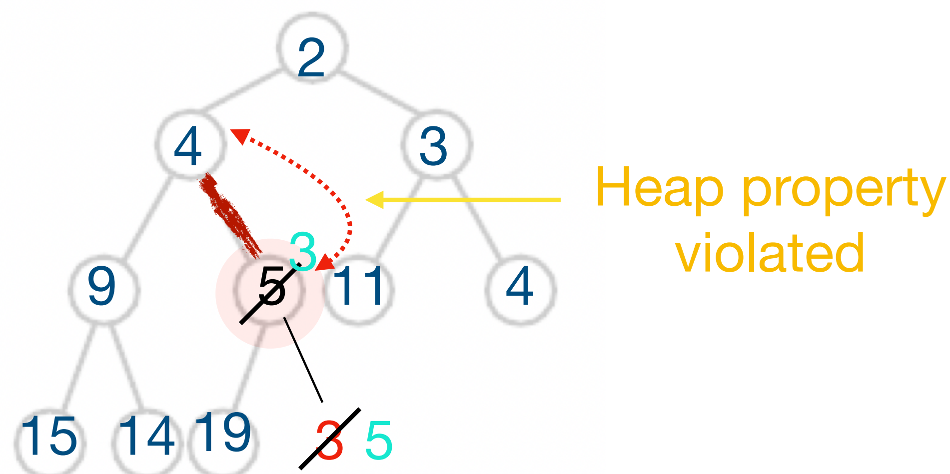


Inserting in a heap

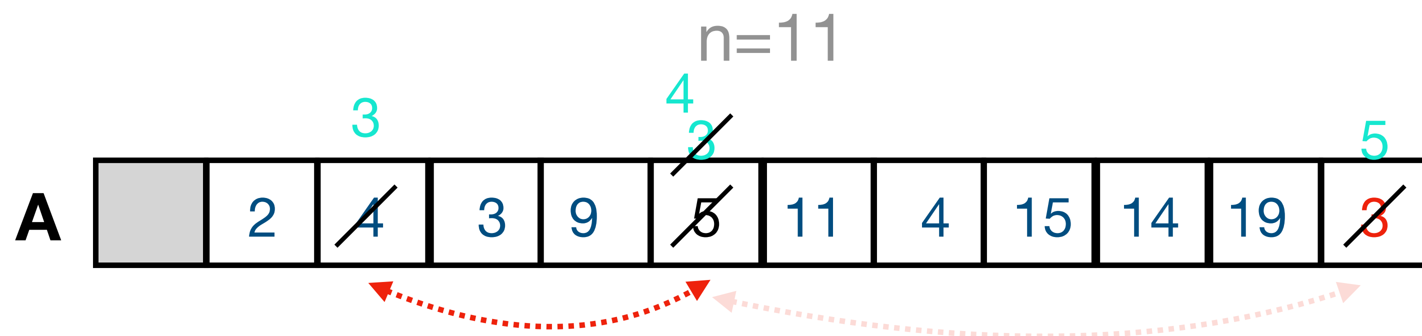
n=11

A		2	4	3	9	~~5~~ 3	11	4	15	14	19	~~3~~ 5

Insert(A, 3)



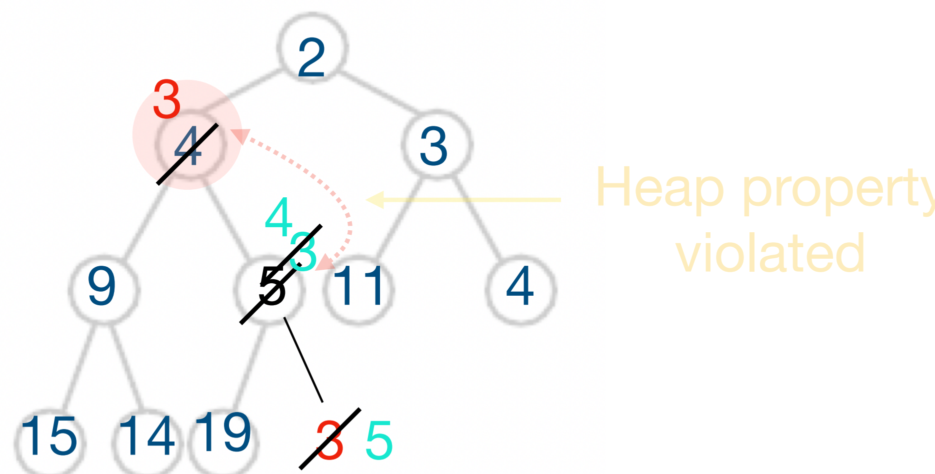
Inserting in a heap



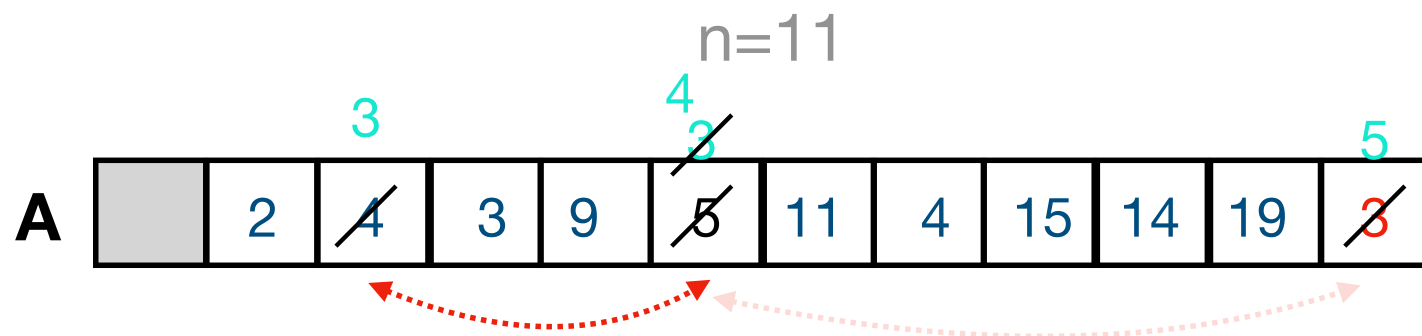
Insert(A, e)

1. Add e at the end of the heap
2. "Bubble-up" to restore heap property: swap e with its parent, and repeat

Insert(A, 3)



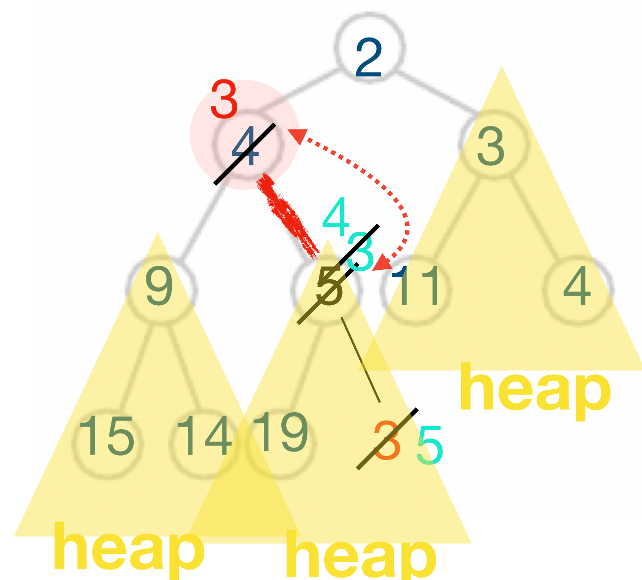
Inserting in a heap



Insert(A, e)

1. Add e at the end of the heap
2. “Bubble-up” to restore heap property: swap e with its parent, and repeat

Insert(A, 3)



Why is this correct?

Inserting in a heap

n=11

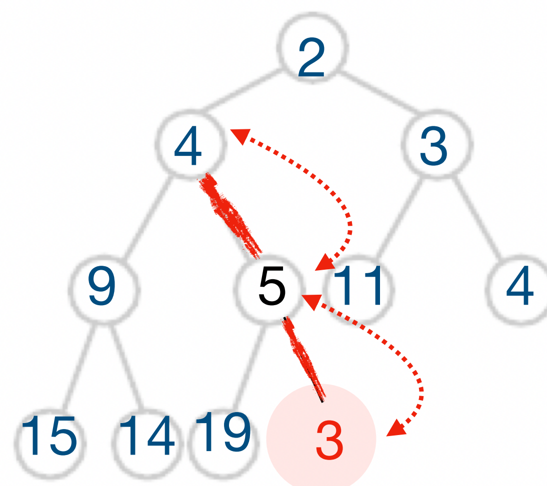
A												
		2	4	3	9	5	11	4	15	14	19	3

Insert(A, e)

1. Add e at the end of the heap
2. “Bubble-up” to restore heap property: swap e with its parent, and repeat

Analysis: O(height) = O(lg n)

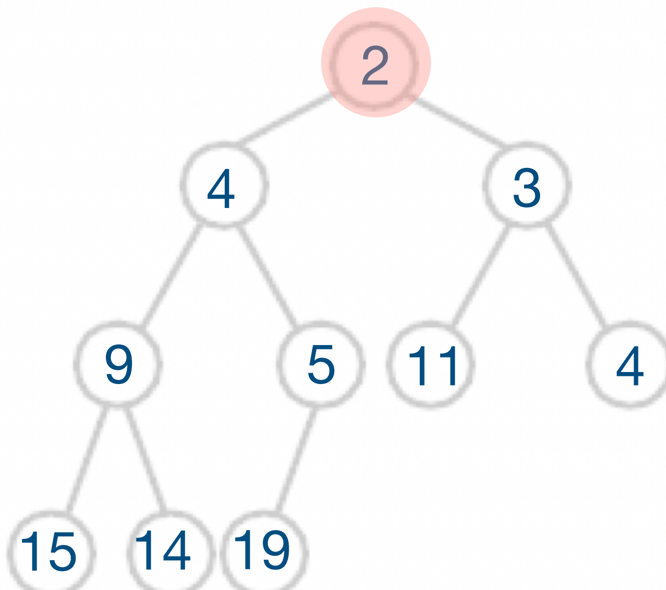
Insert(A, 3)



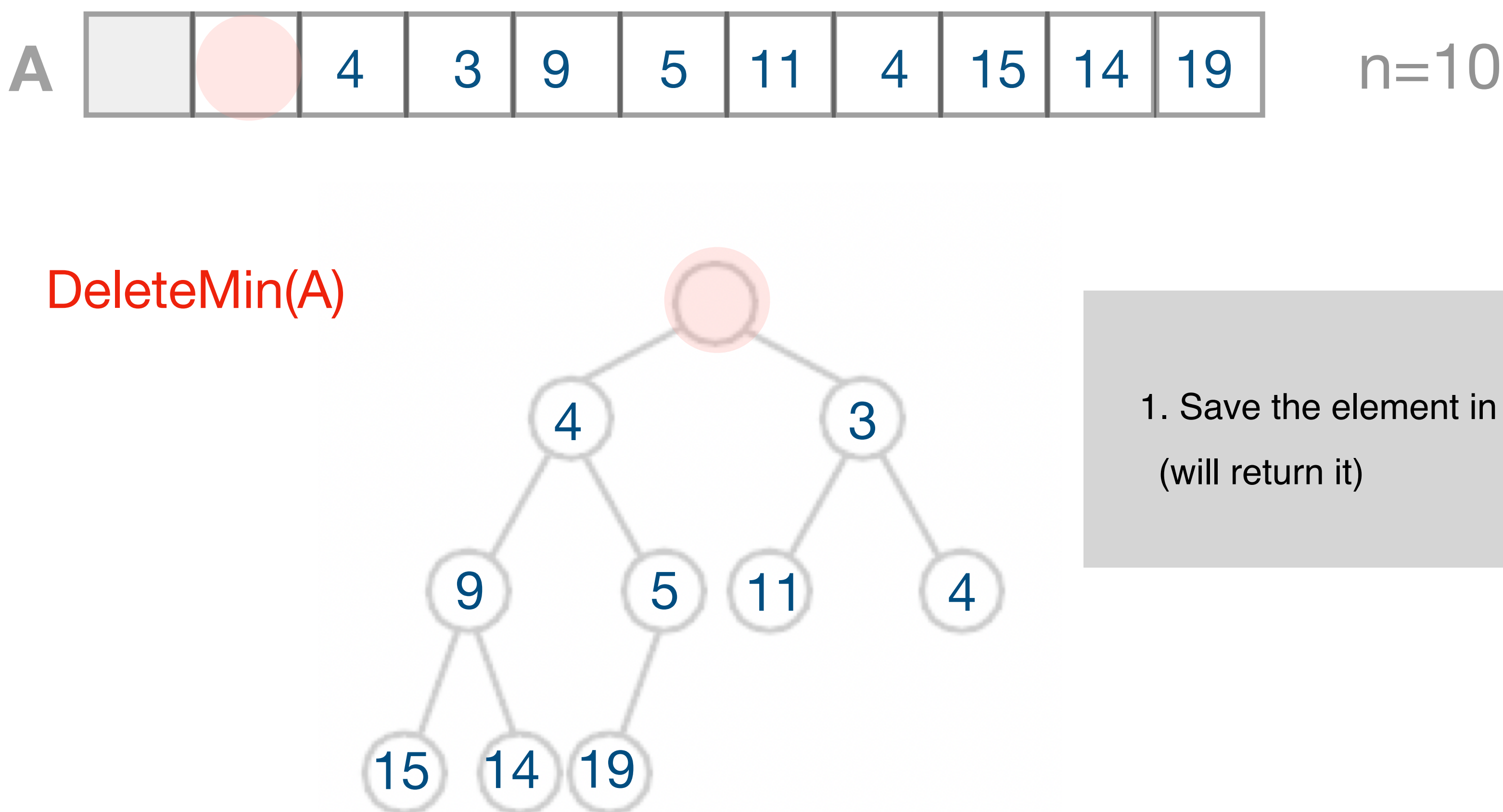
DeleteMin in a heap

n=10

DeleteMin(A)



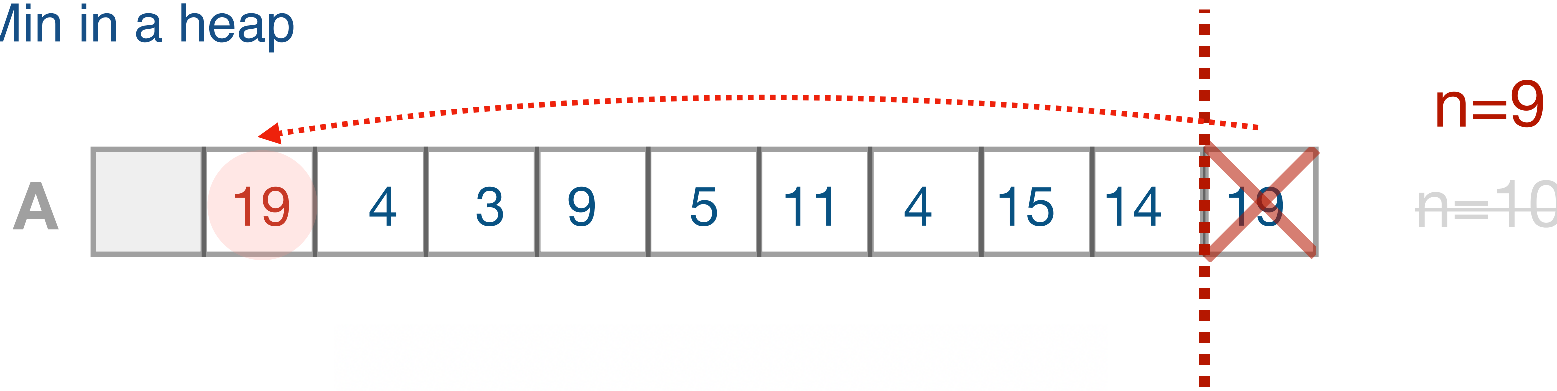
DeleteMin in a heap



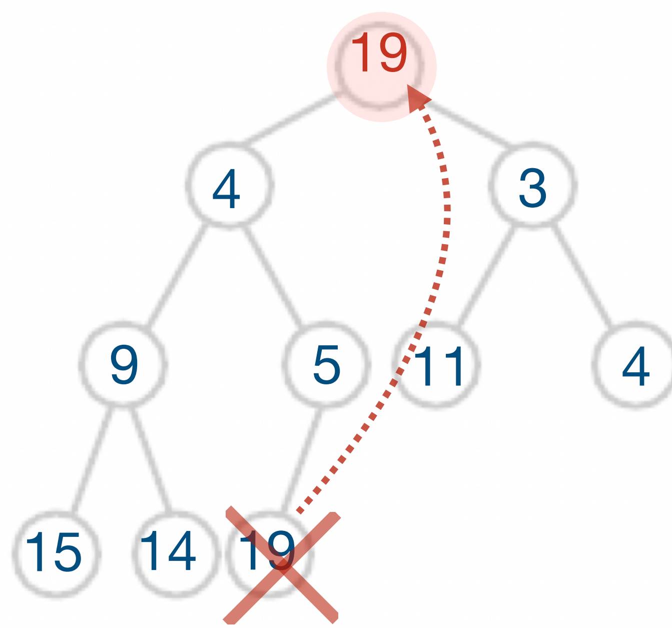
DeleteMin(A)

1. Save the element in the root
(will return it)

DeleteMin in a heap

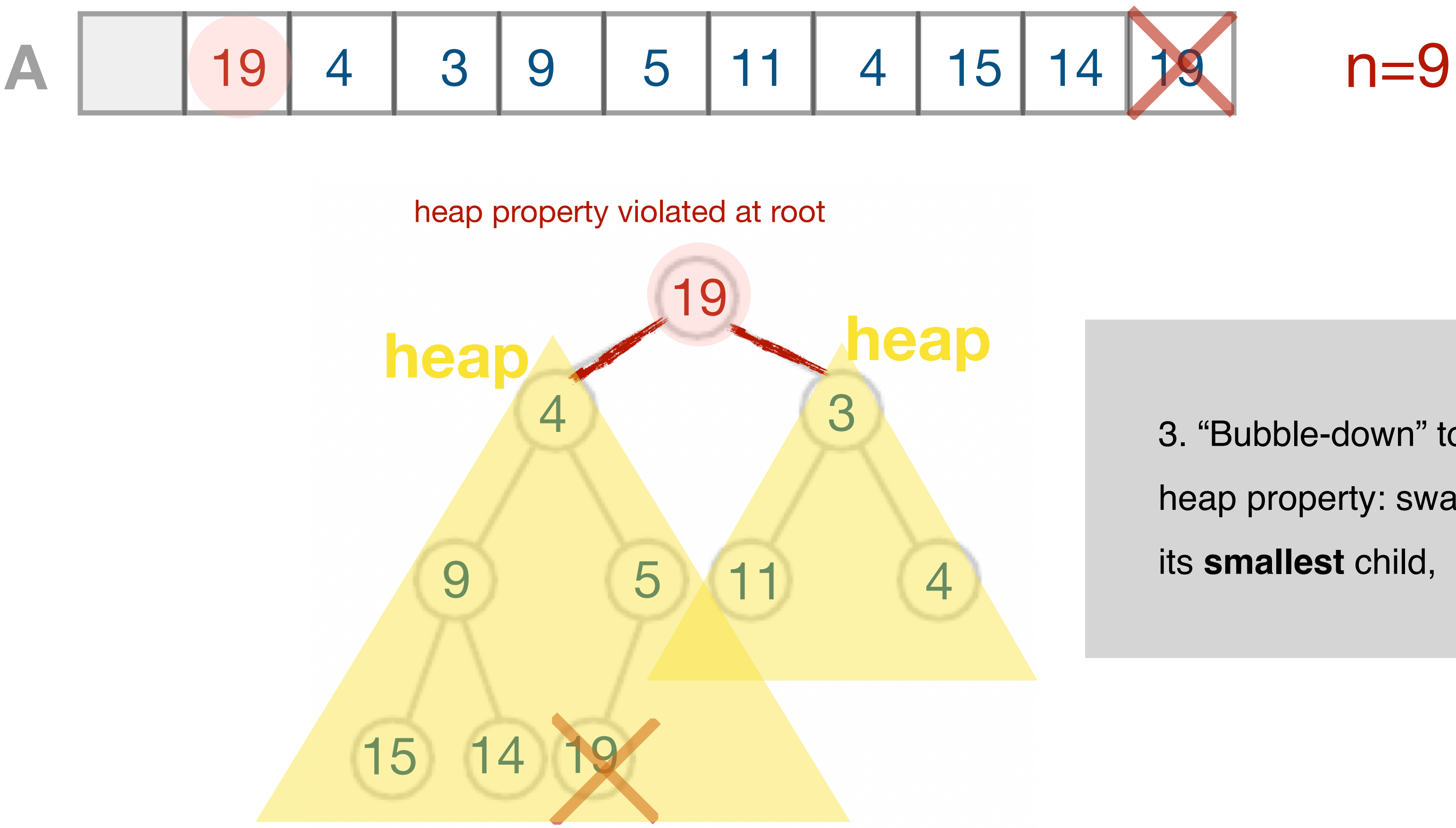


DeleteMin(A)



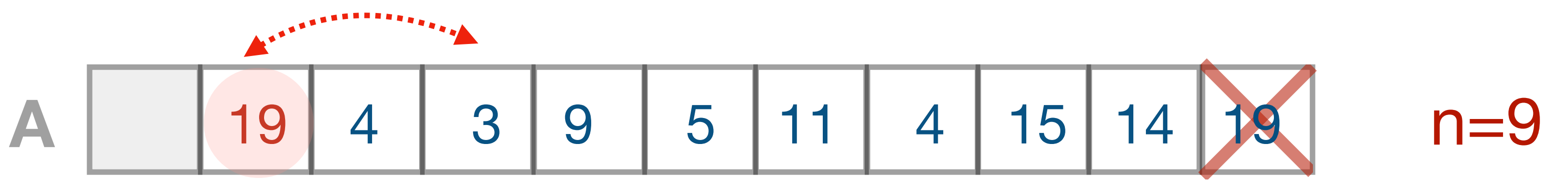
2. Take the last element and put it in the root

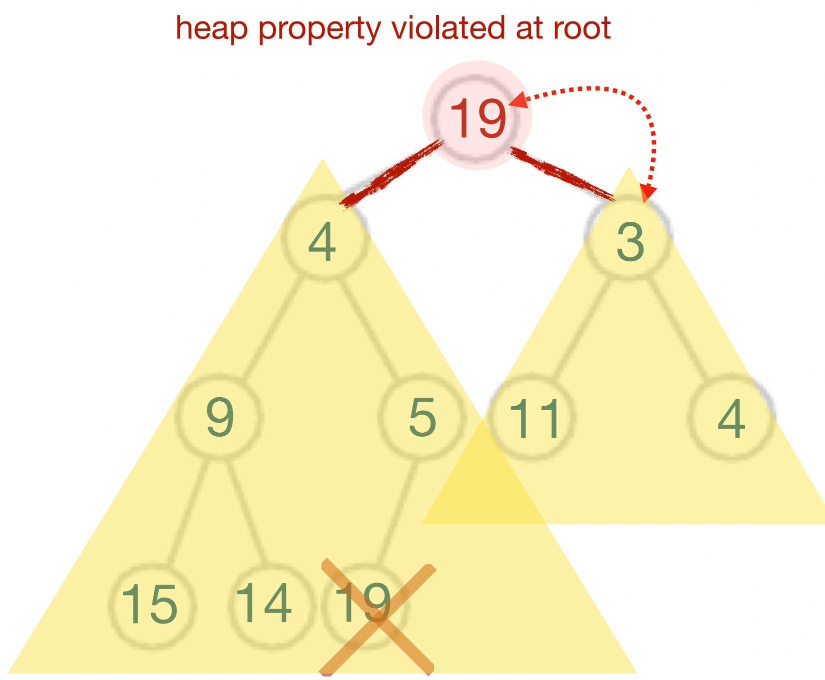
DeleteMin in a heap



3. “Bubble-down” to restore heap property: swap root with its **smallest** child, and repeat

DeleteMin in a heap



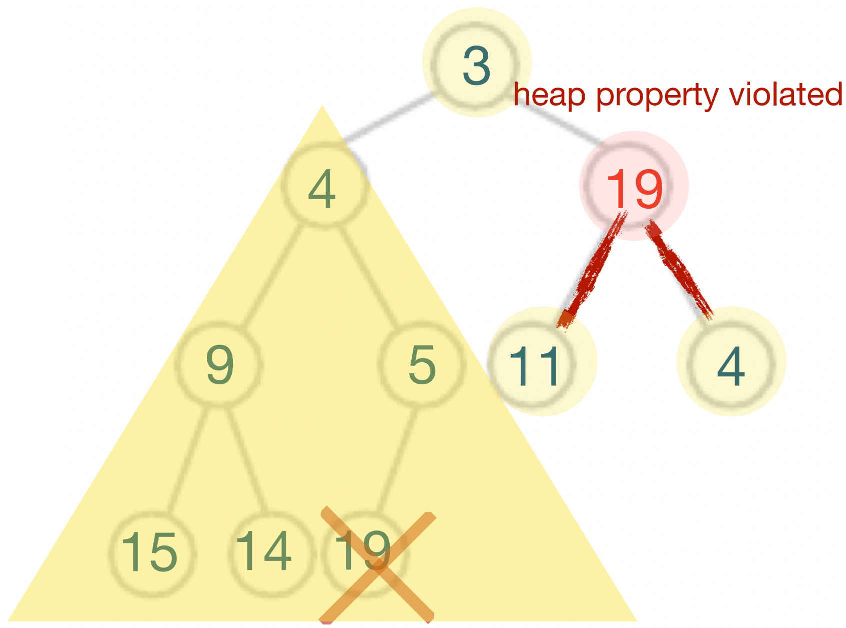


3. “Bubble-down” to restore heap property: swap root with its **smallest** child, and repeat

DeleteMin in a heap

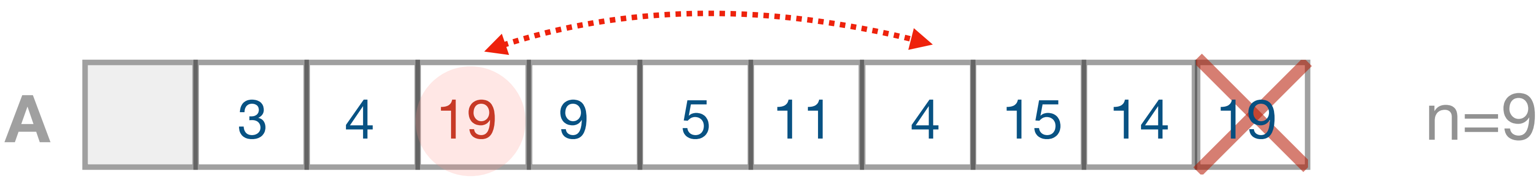
A		3	4	19	9	5	11	4	15	14	~~19~~

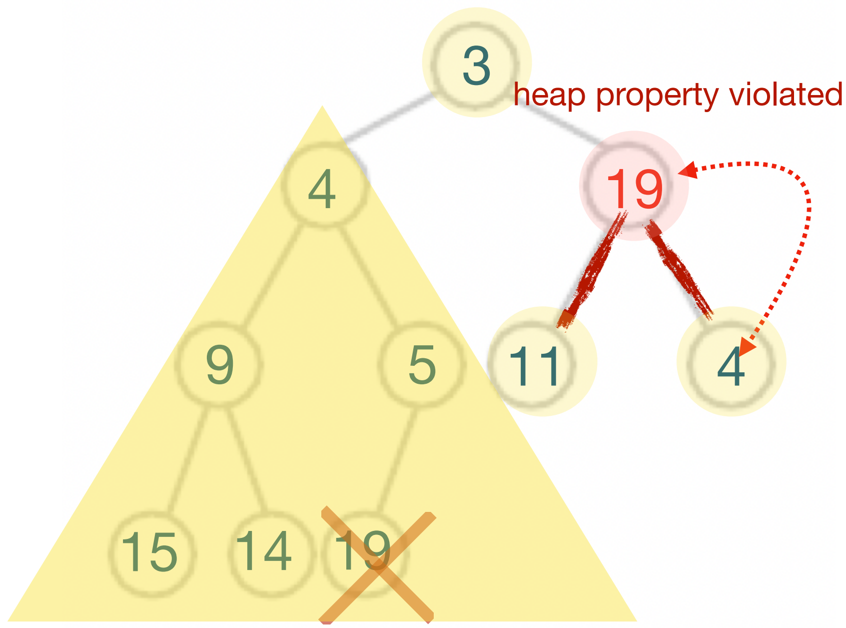
n=9



3. "Bubble-down" to restore heap property: swap root with its **smallest** child, and repeat

DeleteMin in a heap

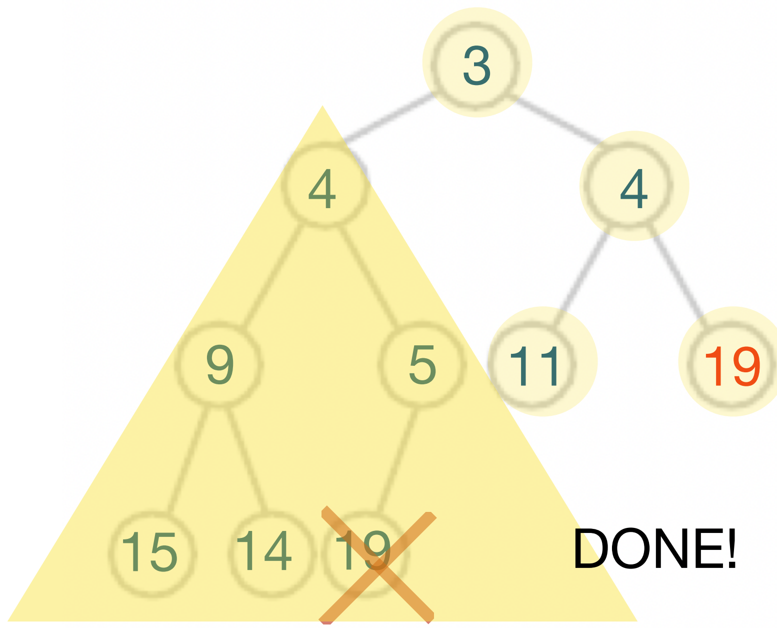




3. “Bubble-down” to restore heap property: swap root with its **smallest** child, and repeat

DeleteMin in a heap

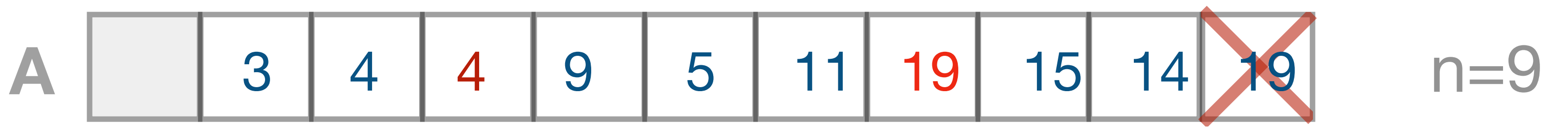
A		3	4	4	9	5	11	19	15	14	~~19~~	n=9



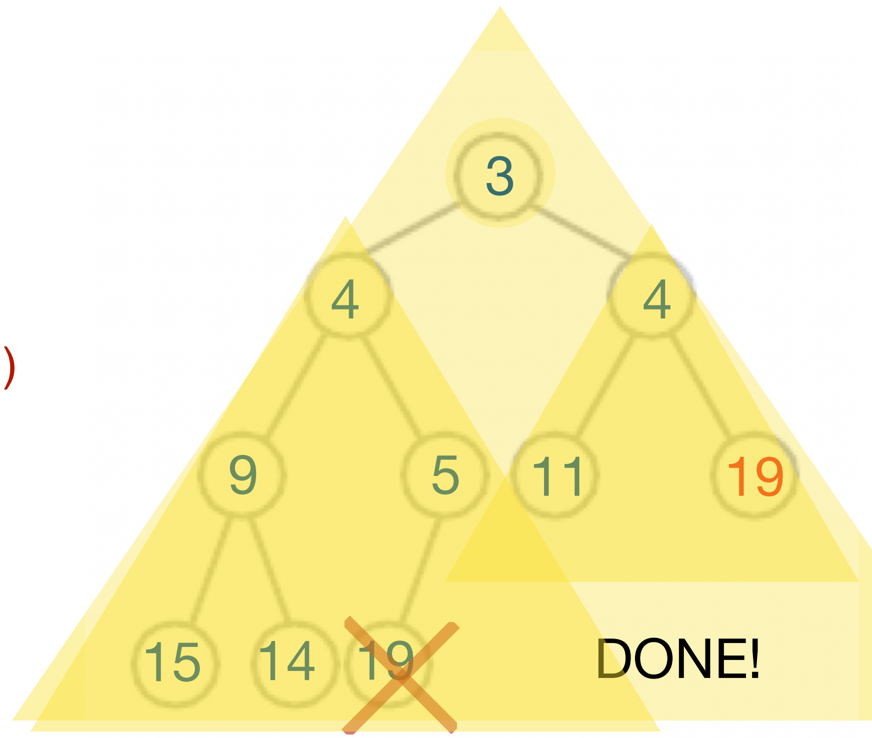
DONE!

3. “Bubble-down” to restore heap property: swap root with its **smallest** child, and repeat

DeleteMin in a heap



Analysis O(lg n)



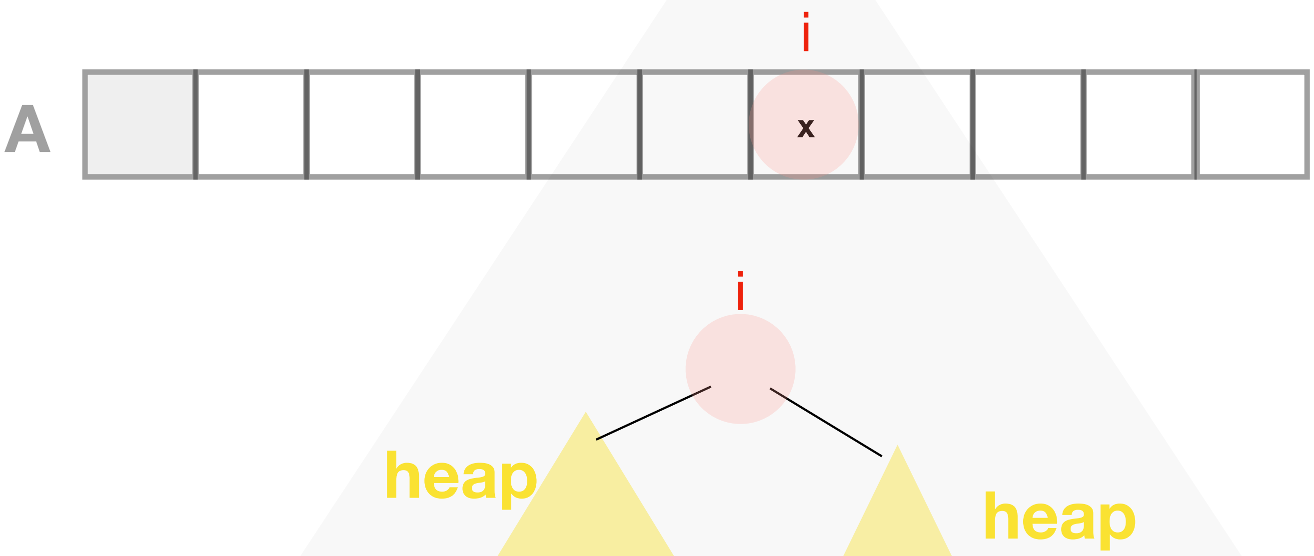
DeleteMin(A)

1. Save the element in the root (will return it)

2. Take the last element and put it in the root

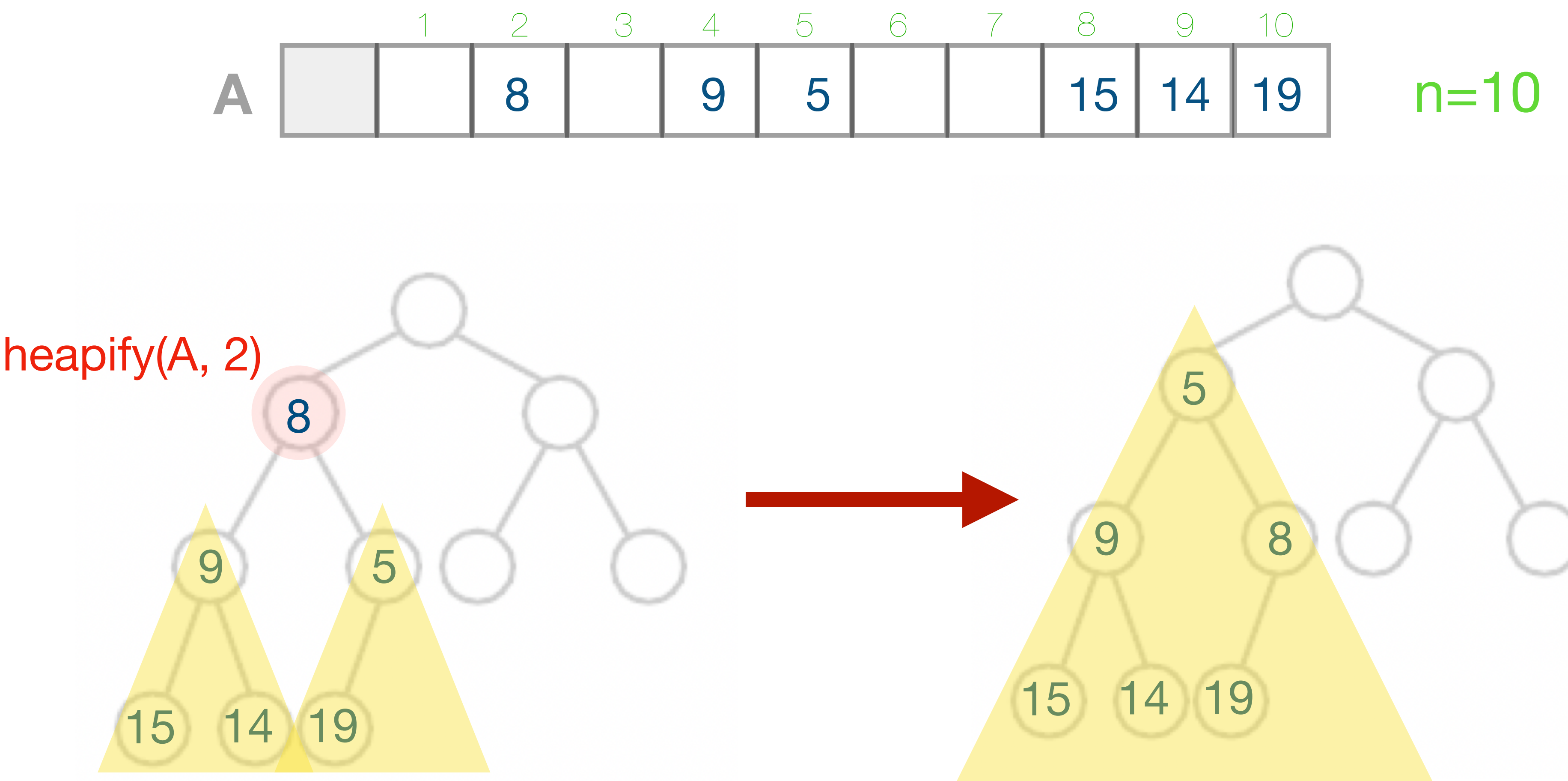
3. “Bubble-down” to restore heap property: swap root with its **smallest** child, and repeat

Heapify(A, i): makes a heap under i



- **Input**:
    - i is an index in A, $1 \leq i \leq n$
    - the subtrees rooted at left(i) and right(i) both satisfy heap property, but heap property is violated at node i
- **Output**: the subtree rooted at i satisfies heap property

Example



A
1 2 3 4 5 6 7 8 9 10
8 9 5 15 14 19
n=10
heapify(A, 2)
8
9 5
15 14 19
5
9 8
15 14 19

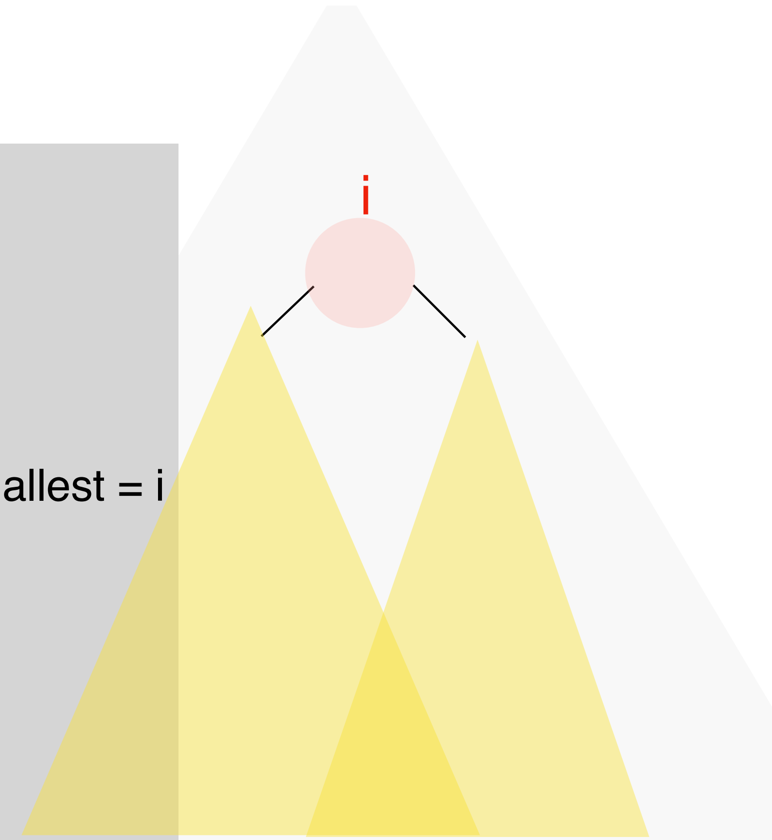
Heapify(A, i)

//find smallest of its children

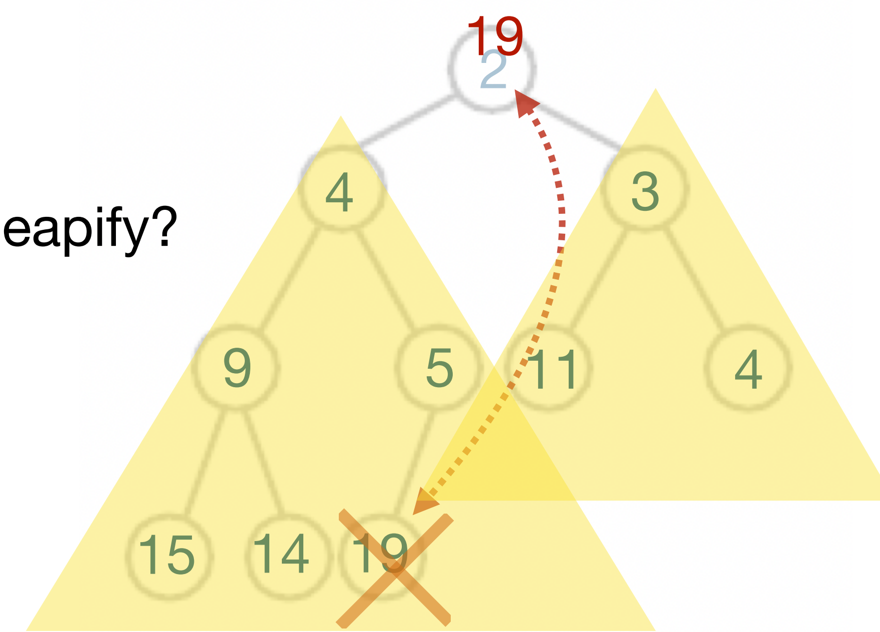
- l = left(i),   r = right(i)
- if l <= heapsize(A) and A[l] < A[i]: smallest = l, else smallest = i
- if (r <= heapsize(A) and A[r] < A[smallest] : smallest = r

//swap and recurse

- if smallest ! = i:
    - exchange A[i] with A[smallest]
    - Heapify(A, smallest)



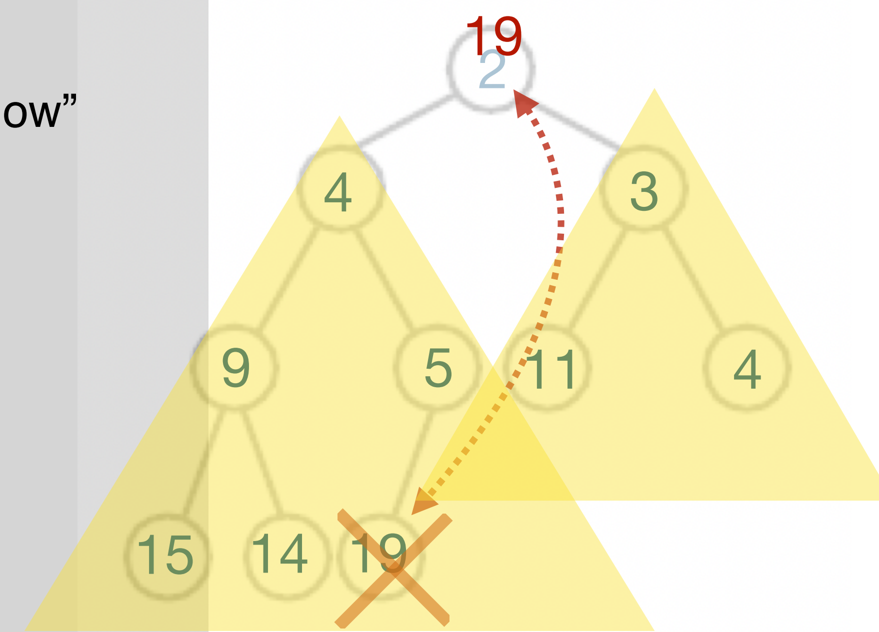
Can we implement deleteMin in terms of heapify?



A		2	4	3	9	5	11	4	15	14	19	n=10

deleteMin(A)

- if heapsize(A) < 1: error "heap underflow"
- min = A[1]
- A[1] = A[heapsize(A)]
- heapsize(A) - -
- Heapify(A, 1)
- return min



Sorting with a heap

Sorting with a heap

- **The** p**roblem**: A is an array. Sort A using a heap.
- How?

Sorting with a heap

- **The problem**: A is an array. Sort A using a heap.
- We could traverse the elements in A and insert them in a heap, then deleteMin one at a time.

```
sort-with-a-heap(A)

  • H = empty heap

  • for i=0 to n-1:    insert(H, A[i])

  • for i=0 to n-1:    A[i] = deleteMin(H)
```

Sorting with a heap

- **The problem**: A is an array. Sort A using a heap.
- We could traverse the elements in A and insert them in a heap, then deleteMin one at a time.

> sort-with-a-heap(A)
>
> - H = empty heap
> - for i=0 to n-1: insert(H, A[i])
> - for i=0 to n-1: A[i] = deleteMin(H)

Analysis: n x insert + n x deleteMin = O(n lg n)

- This is great, but it's not in place.
- Can we sort with a heap in place?

Sorting with a heap **in place**

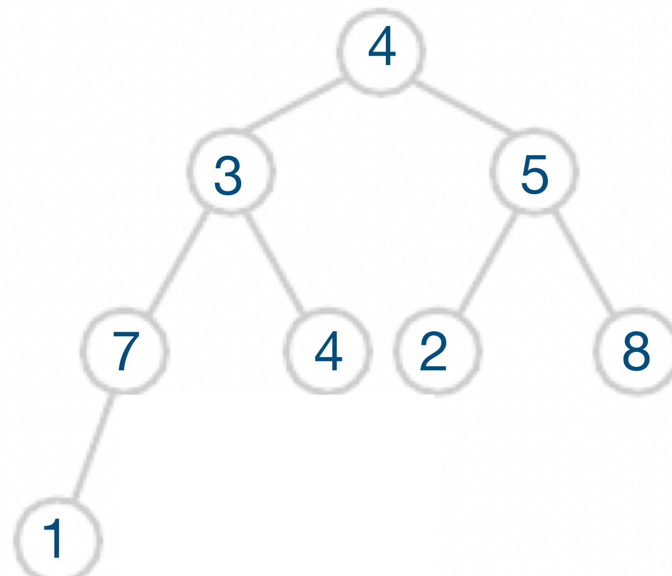
- Ingredient 1: making an array into a heap in place

buildheap(A)

Input: A is an array

Output: A is a heap

A								
	4	3	5	7	4	2	8	1



Sorting with a heap **in place**

- Ingredient 1: making an array into a heap in place

buildheap(A)

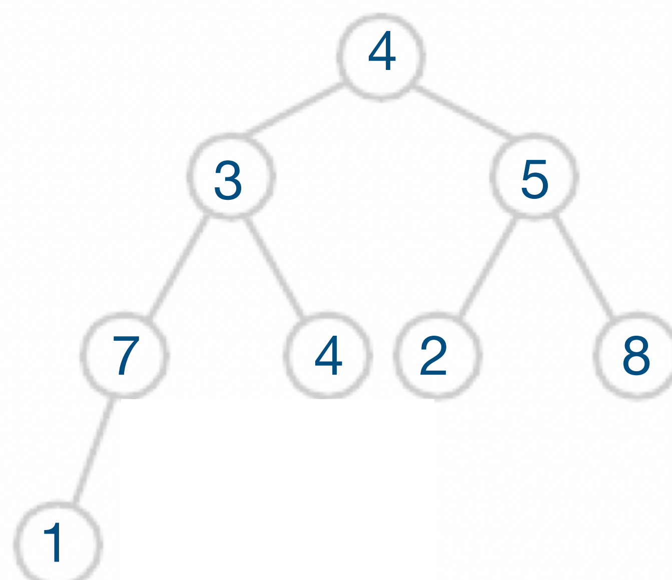
Input: A is an array

Output: A is a heap

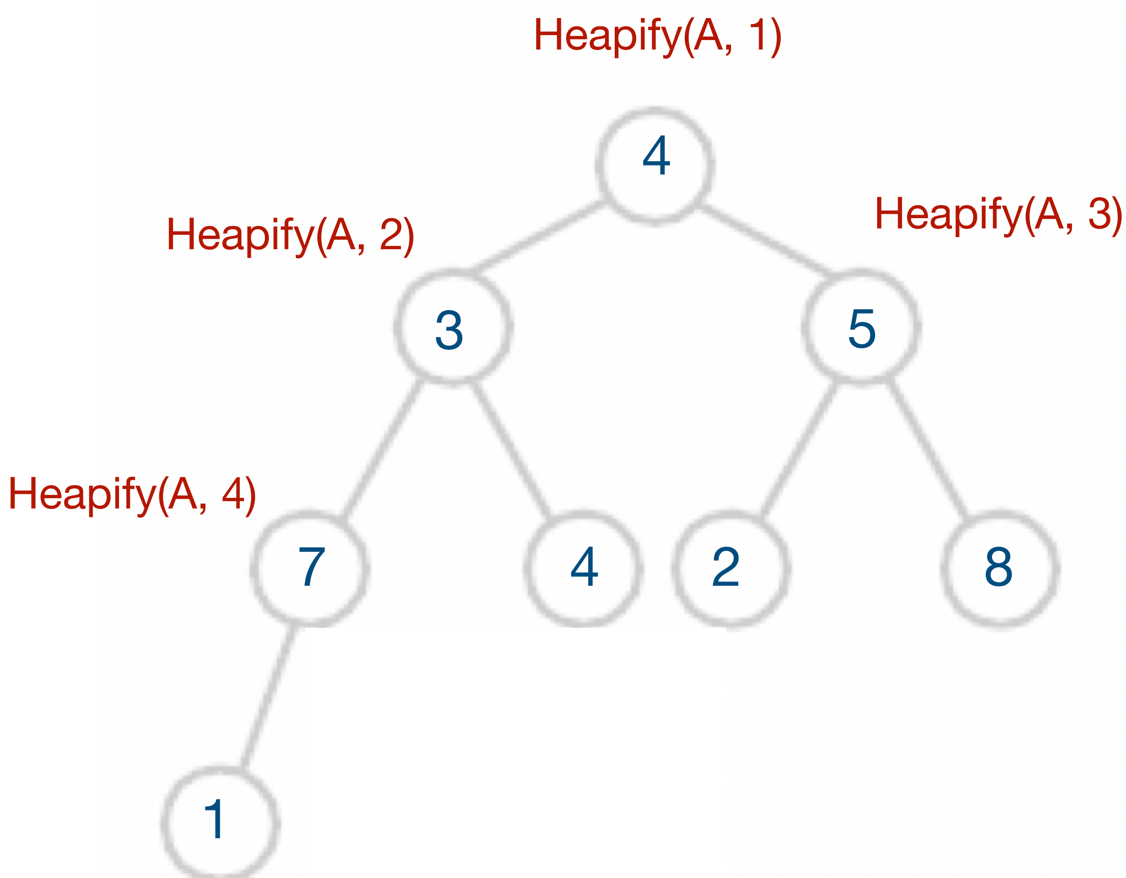
//build a heap gradually, bottom up

- for i = n/2 down to 1: Heapify (A, i)

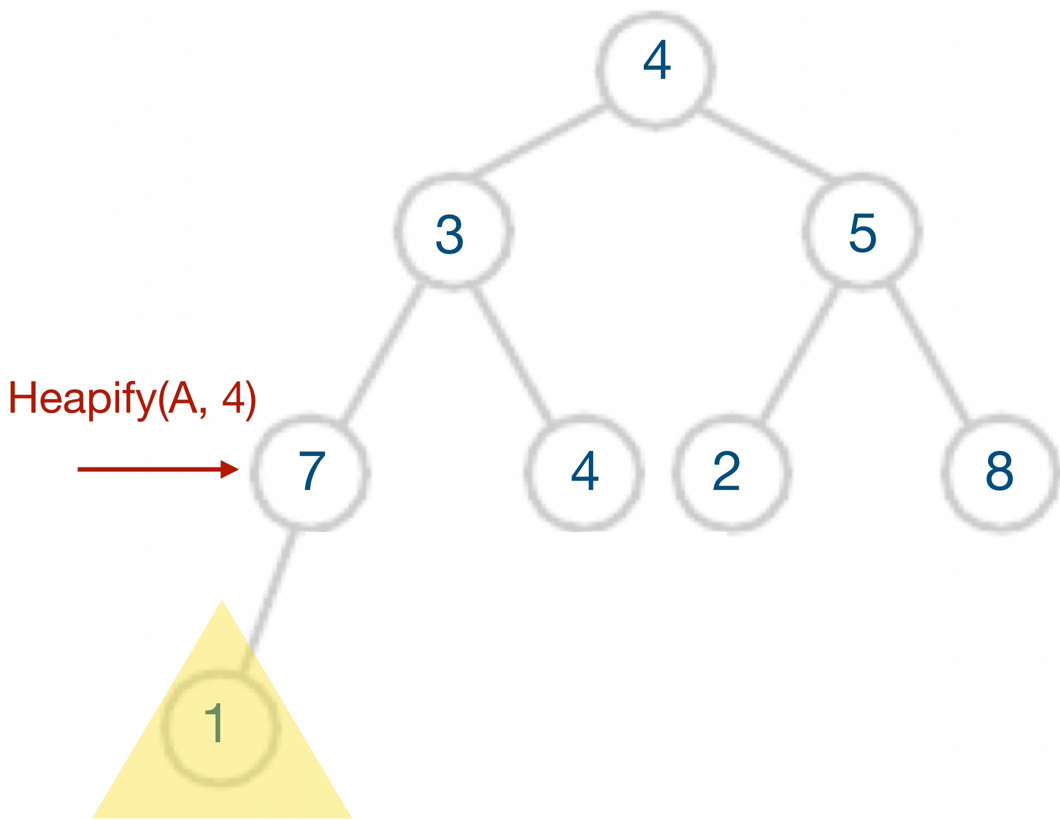
A	4	3	5	7	4	2	8	1



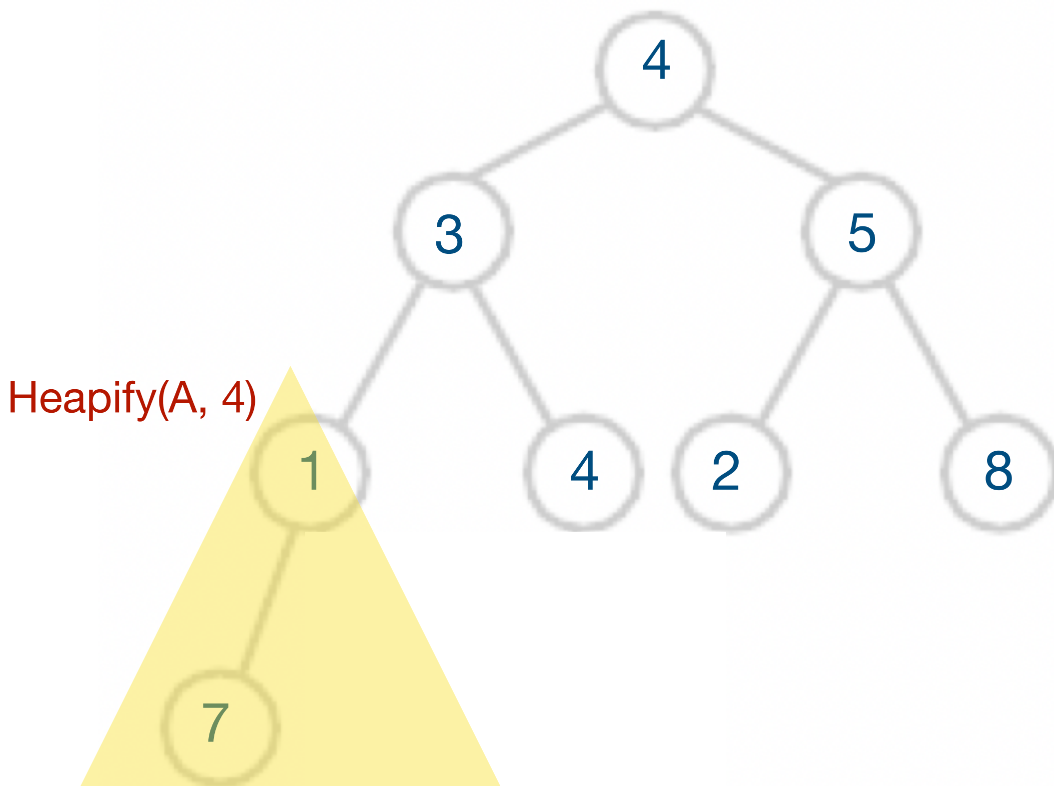
Buildheap(A)



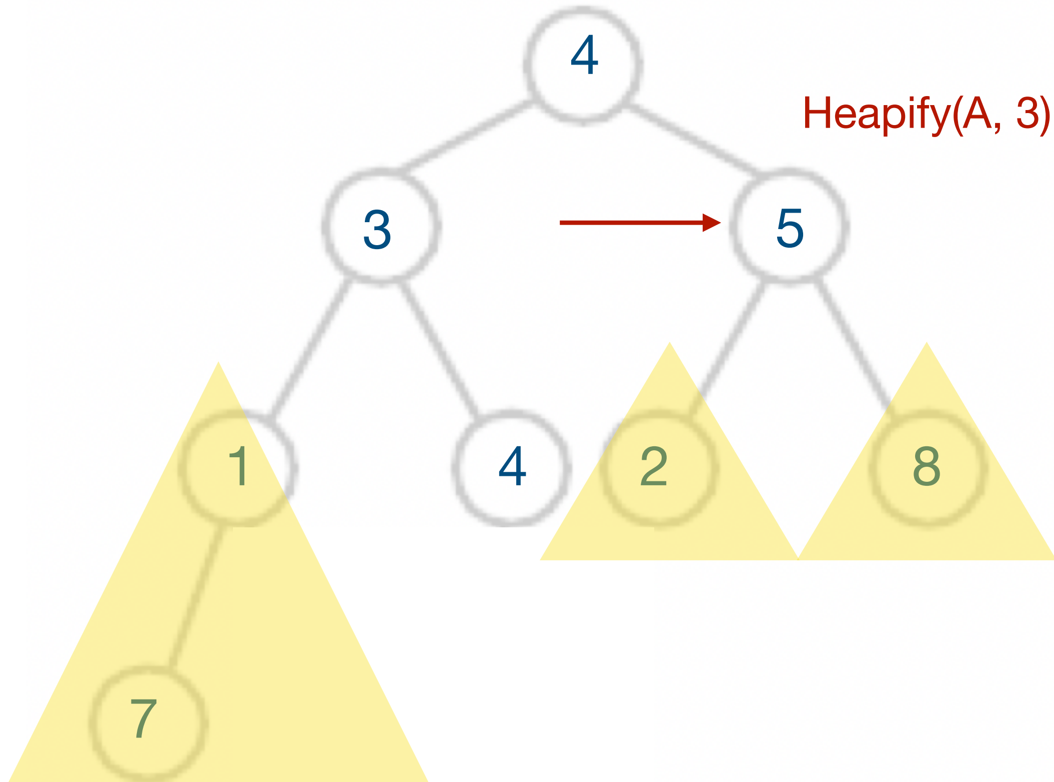
Buildheap(A)



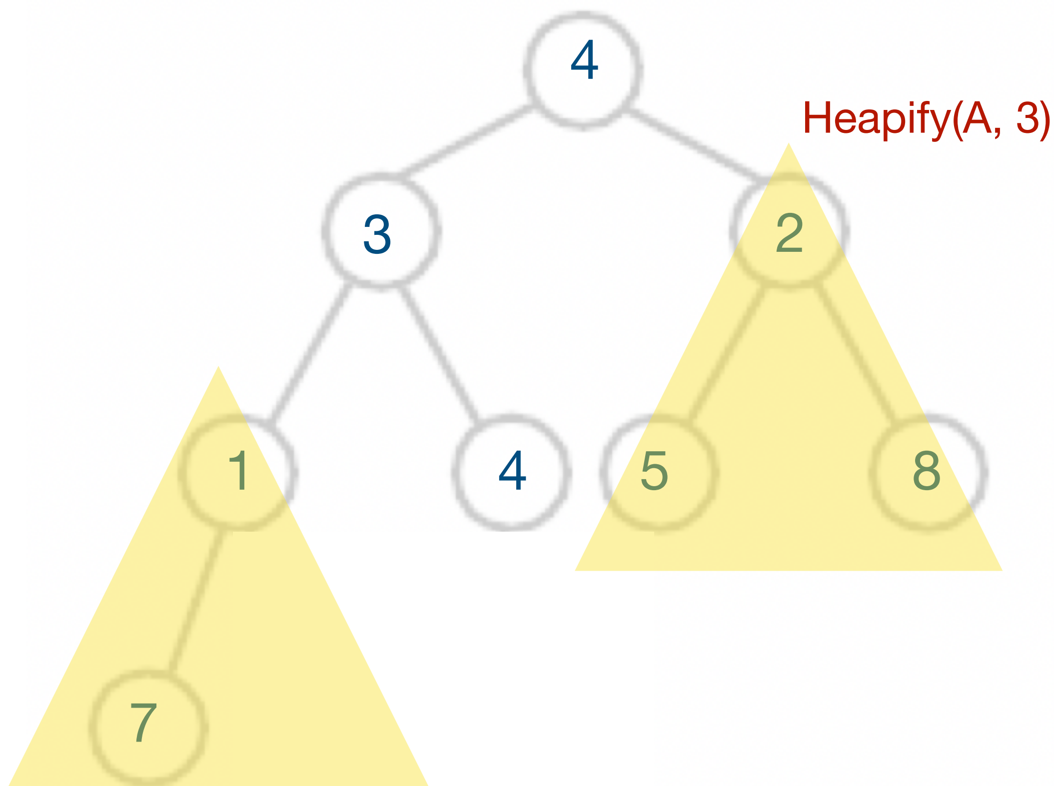
Buildheap(A)



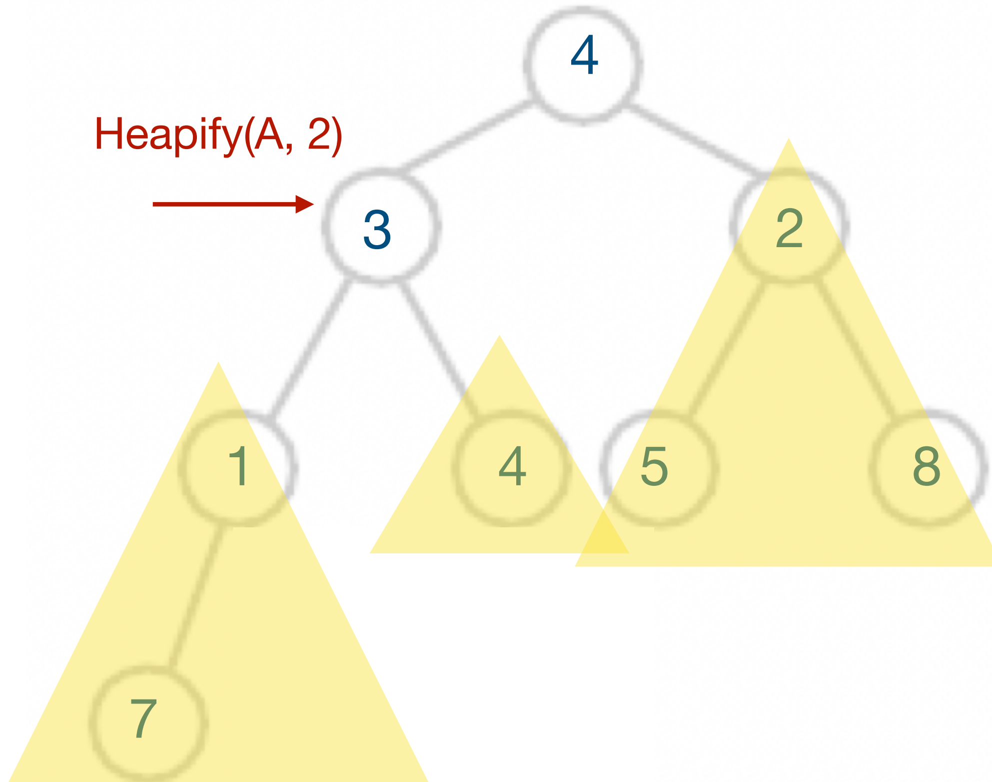
Buildheap(A)



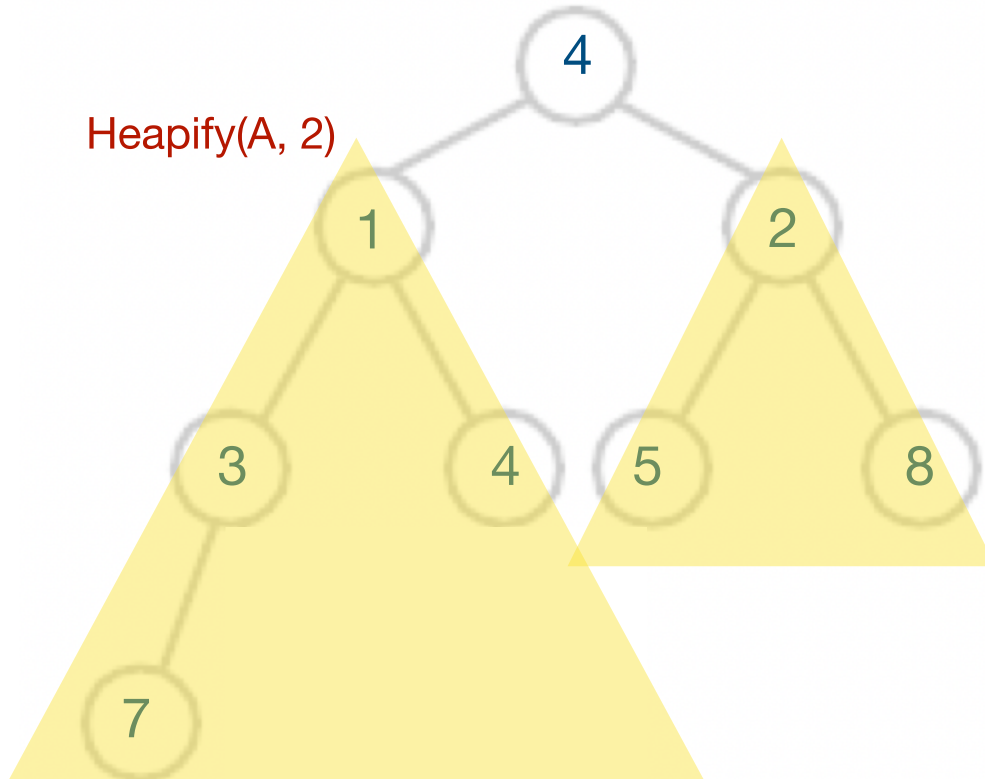
Buildheap(A)



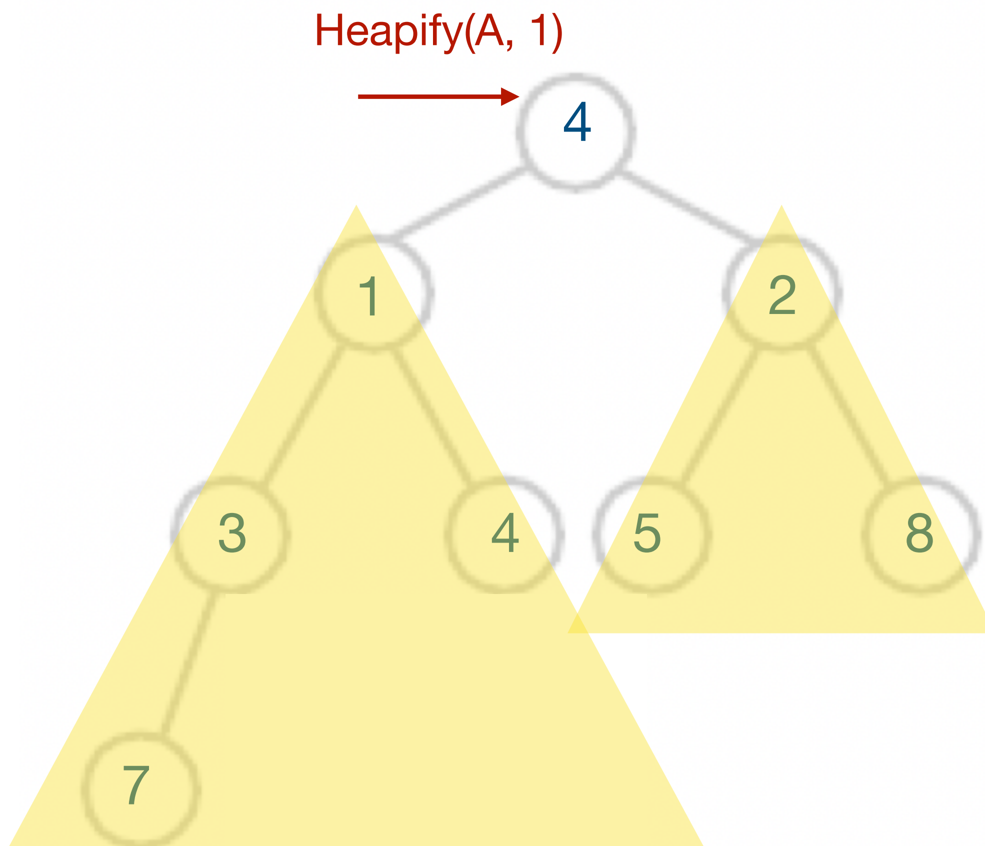
Buildheap(A)



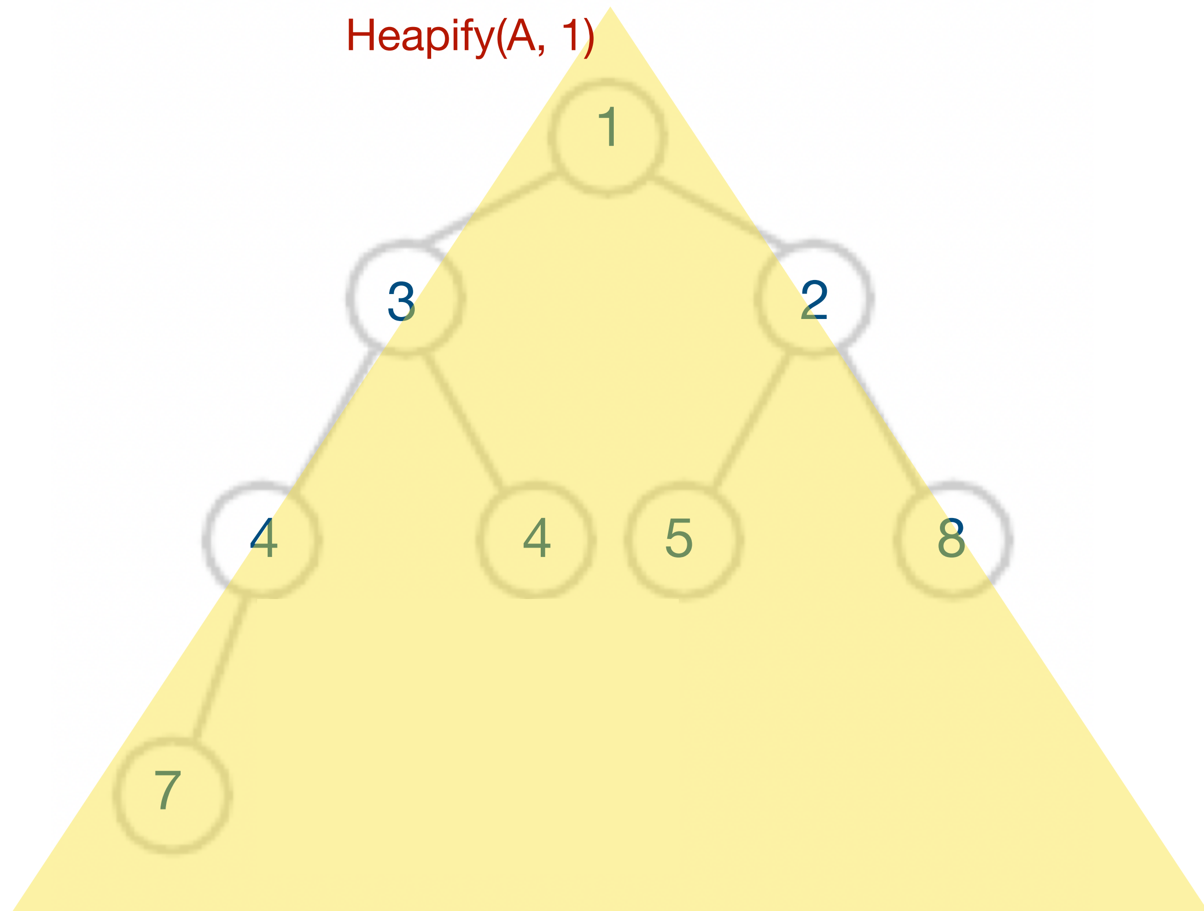
Buildheap(A)



Buildheap(A)



Buildheap(A)



buildheap(A)

- for i = n/2 down to 1: Heapify (A, i)

Analysis: It can be shown that this runs in overall O(n)

A	4	3	5	7	4	2	8	1

$\longrightarrow$

A	1	3	2	4	4	5	8	7

Heapsort(A) in place

Idea: use a max pqueue

```
Heapsort(A)
    Convert A into a max-heap
    //Repeatedly Delete-Max and put it at the end of the array
    for i=0 to n-1: A[n-i] = DELETE-MAX(A)
```

Run time: Buildheap + n x Delete-Max ==> O(n lg n)

Heaps: summary

Heaps are arrays + heap property

- Insert(A, e)
- Delete-Min() O(lg n)
- Heapify(A, i)
- Buildheap(A) O(n)
- Heapsort (A) O(n lg n), in place

- Note that cannot **Search** efficiently in a heap
- Generalize to 3-heaps, …. d-heaps