Many algorithms use Priority Queues

- **Dijkstra's algorithm**: finding a shortest path in a graph
- **Prim's algorithm**: constructing a minimum spanning tree of a graph
- **Huffman encoding**: constructing an optimum prefix-free encoding of a string
- **Heap sort:** sorting a given sequence

Using Heaps for Sorting
Heap Sort

Lecture 23

by Marina Barsky

We can sort using Heaps!

- After array elements are enqueued –
- Produce a sorted array by dequeuing them

Algorithm *HeapSort*

```
HeapSortNaive (array A of size n)

create an empty max-heap

for i from 0 to n-1:
    enqueue (A[i])

for i from n-1 downto 0:
    A[i] ← dequeue()
```

What is the running time of a naïve heap-based sorting algorithm?

A. O(1)

B. O(log n)

C. O(n)

D. O(n * log n)

E. None of the above

Heapsort: naive

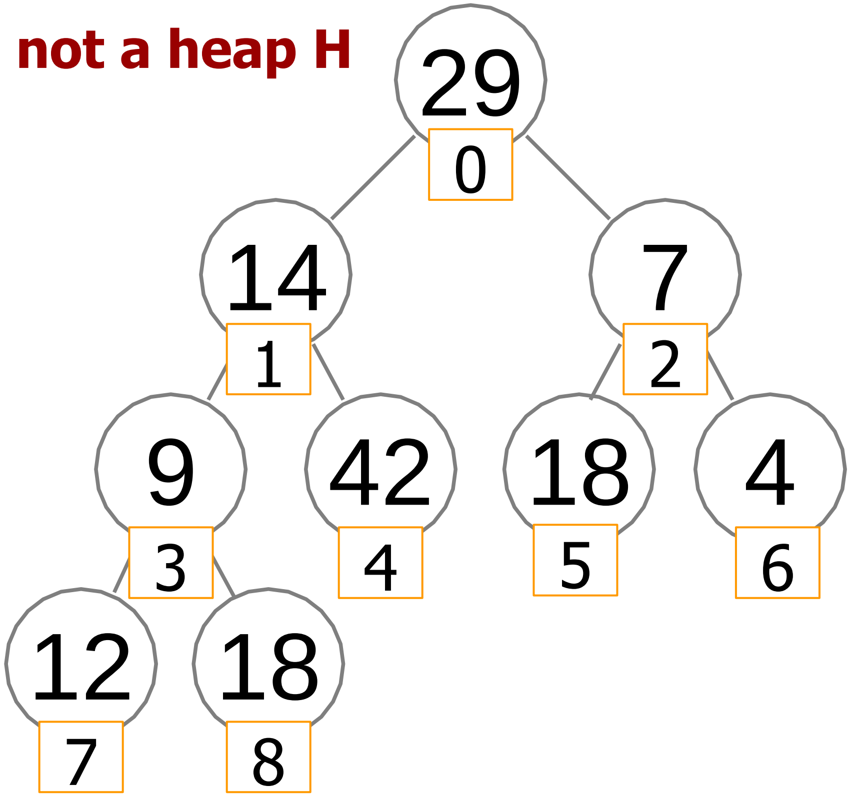
- The resulting algorithm has running time $O(n \log n)$
- Natural generalization of *selection sort* : instead of simply scanning the rest of the array to find the maximum value, use a smart data structure
- Uses additional space $O(n)$ to store the heap

In-place Heapsort: all is done inside the input array

- Turn input array A of size *n* into a heap of size *m*=n by rearranging its elements
- After this, extract max at A[0] and swap it with the element A[*m*-1]
- Decrement heap size *m*:= *m* -1
- Restore heap (*sift_down*)
- Continue until heap size m=1

How to Heapify an array

not a heap H

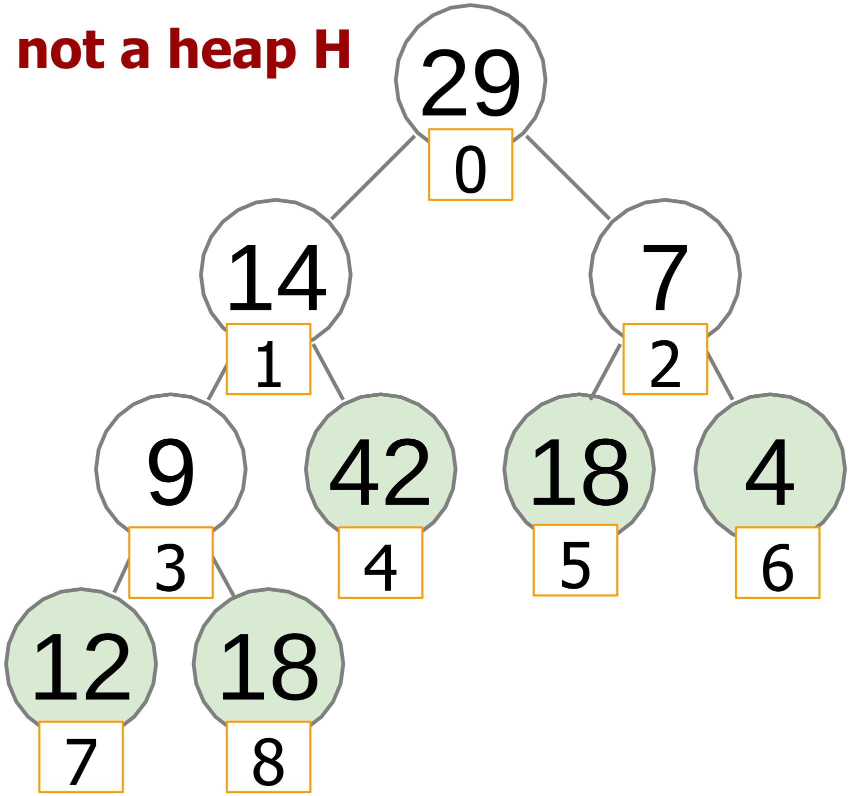


- ➔ Lets' go bottom up and repair heap property for all subtrees rooted at current node

29	14	7	9	42	18	4	12	18
0	1	2	3	4	5	6	7	8

How to Heapify an array

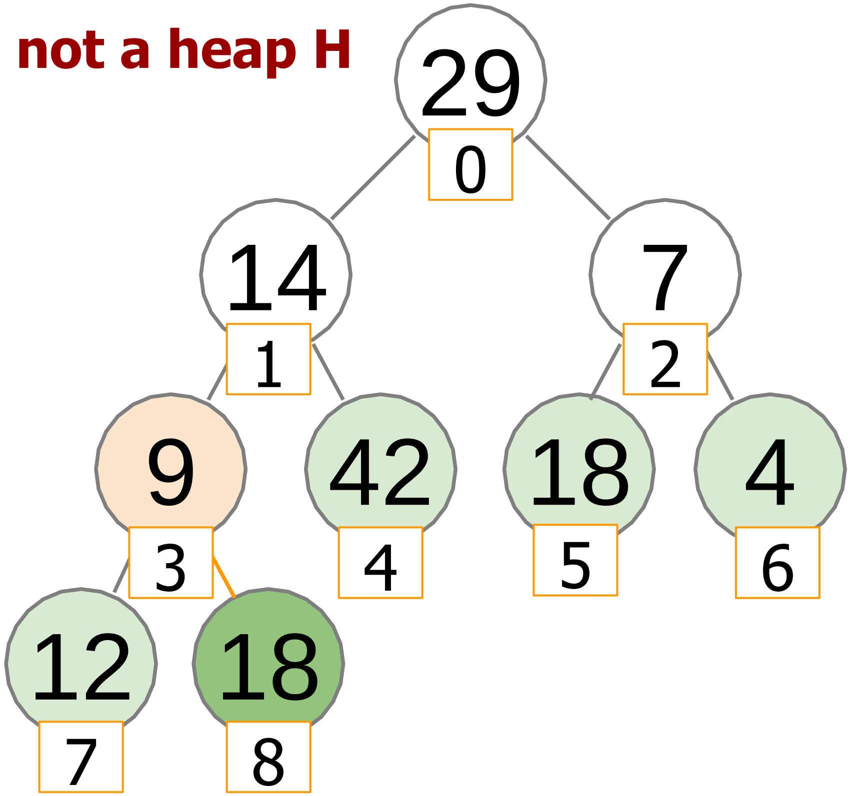
not a heap H



29	14	7	9	42	18	4	12	18
0	1	2	3	4	5	6	7	8

- → Lets’ go bottom up and repair heap property for all subtrees rooted at current node
- → If current node is a leaf, then it does not need to be repaired
- → How do we find the first from the end node that is not a leaf?

How to Heapify an array



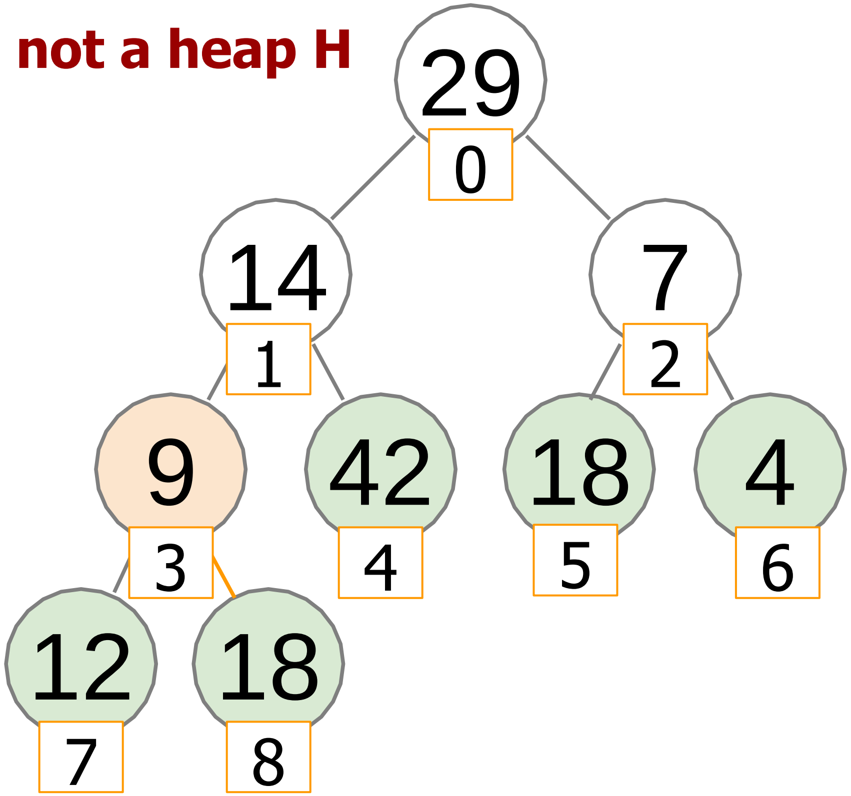
not a heap H

29	14	7	9	42	18	4	12	18
0	1	2	3	4	5	6	7	8

- ➔ Lets' go bottom up and repair heap property for all subtrees rooted at current node
- ➔ If current node is a leaf, then it does not need to be repaired
- ➔ How do we find the first from the end node that is not a leaf?
  - We find the parent of the last leaf H[n - 1]: $\text{parent}(i) = \lfloor (i - 1)/2 \rfloor$

How to Heapify an array

not a heap H

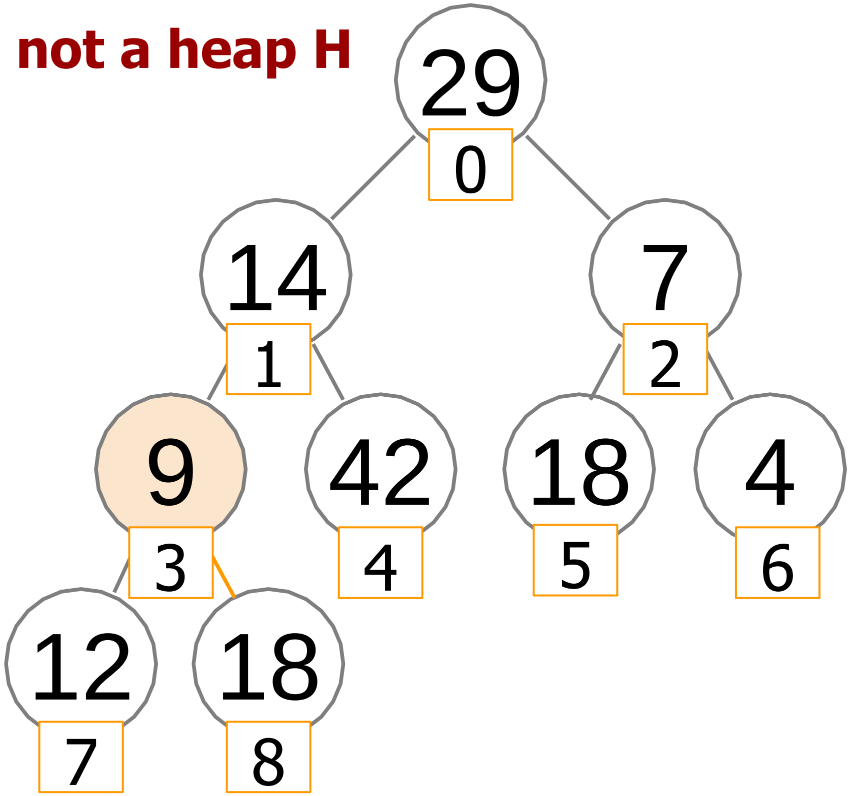


- → We need to process all elements starting from position $i=\lfloor(8-1)/2\rfloor = 3$ until position 0 and repair heap violations by calling sift_down(i)

29	14	7	9	42	18	4	12	18
0	1	2	3	4	5	6	7	8

How to Heapify an array

not a heap H

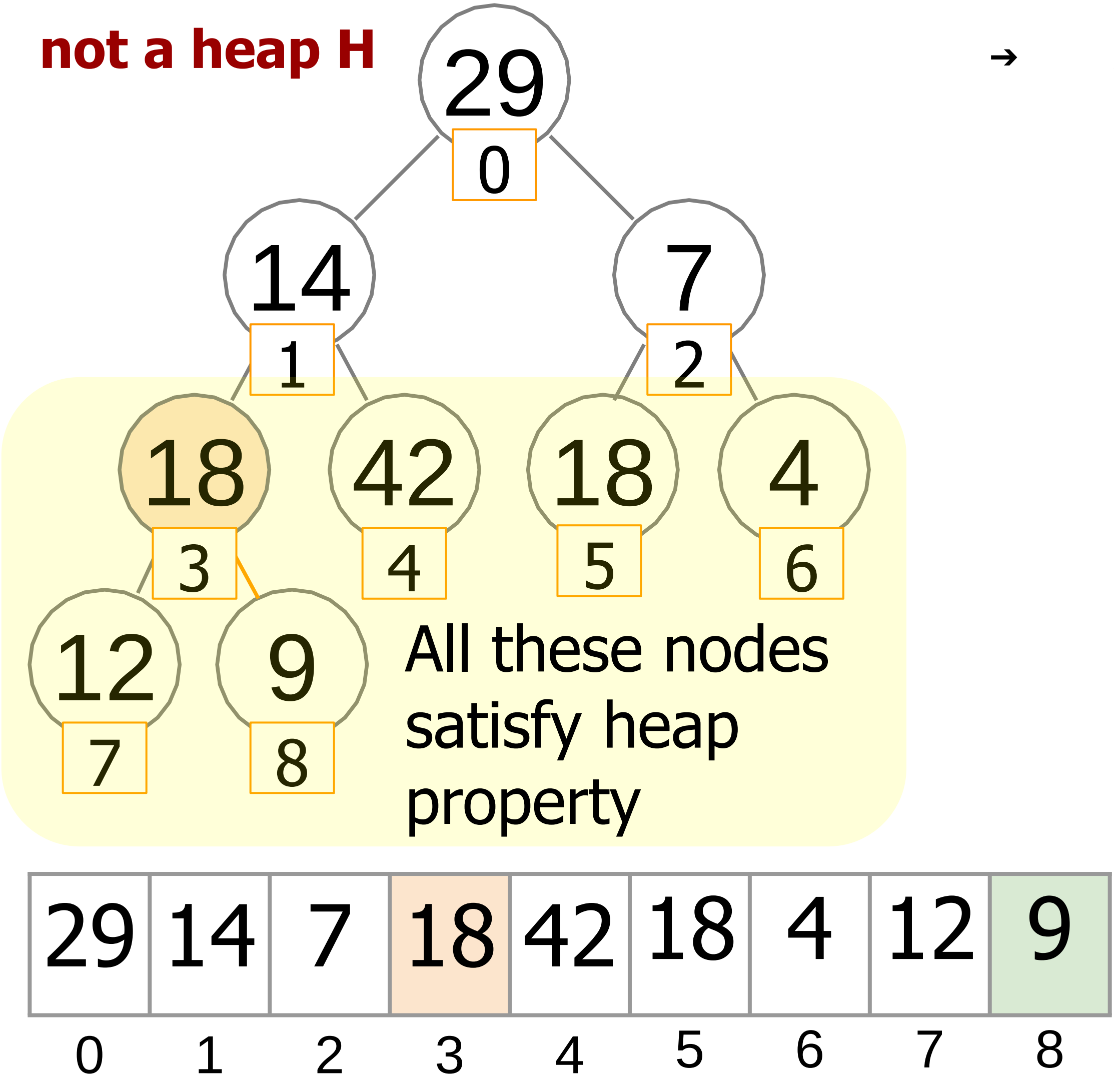


→ We need to process all elements starting from position $i=\lfloor(8-1)/2\rfloor = 3$ until position 0 and repair heap violations by calling sift_down(i)

sift_down(3)

29	14	7	9	42	18	4	12	18
0	1	2	3	4	5	6	7	8

How to Heapify an array

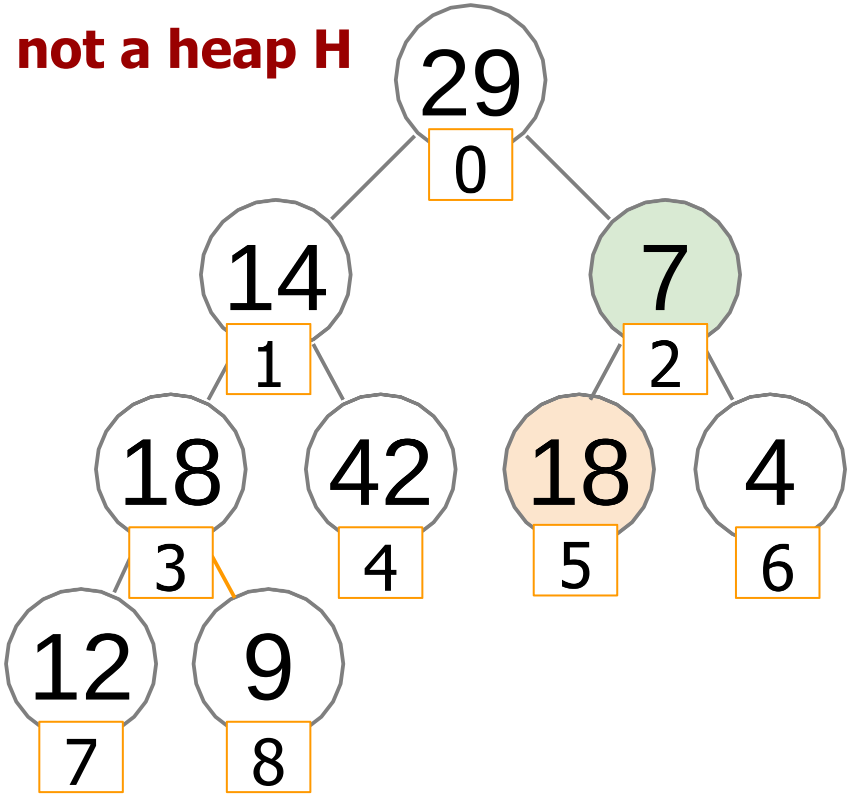


All these nodes satisfy heap property

→ All the nodes H[3...8] are now repaired

How to Heapify an array

not a heap H



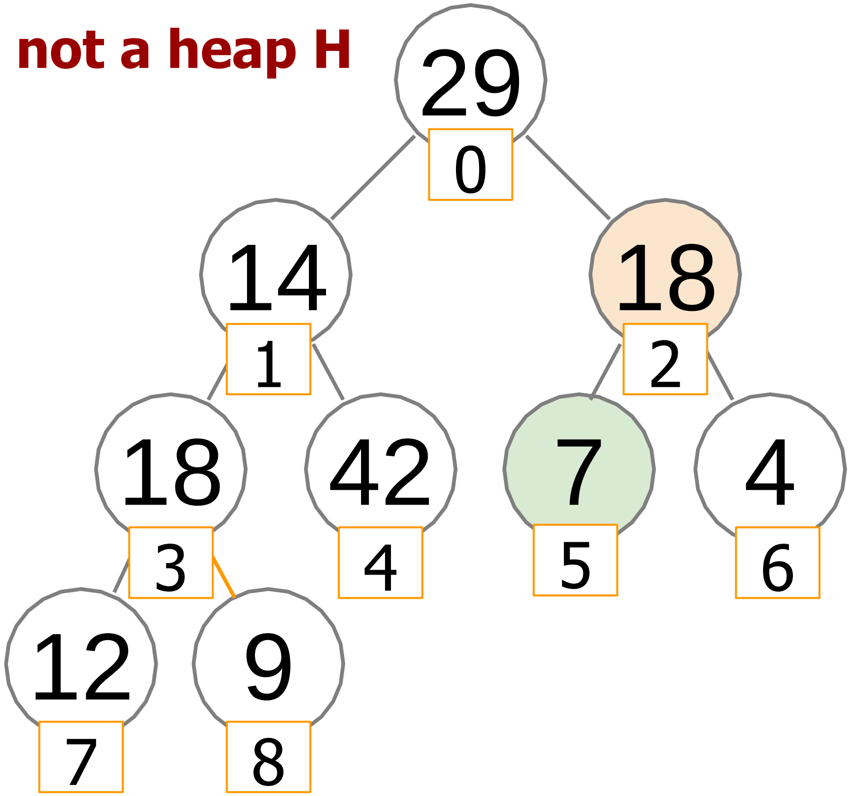
→ the next node we need to fix is at position 2 of the array

sift_down(2)

29	14	7	18	42	18	4	12	9
0	1	2	3	4	5	6	7	8

How to Heapify an array

not a heap H



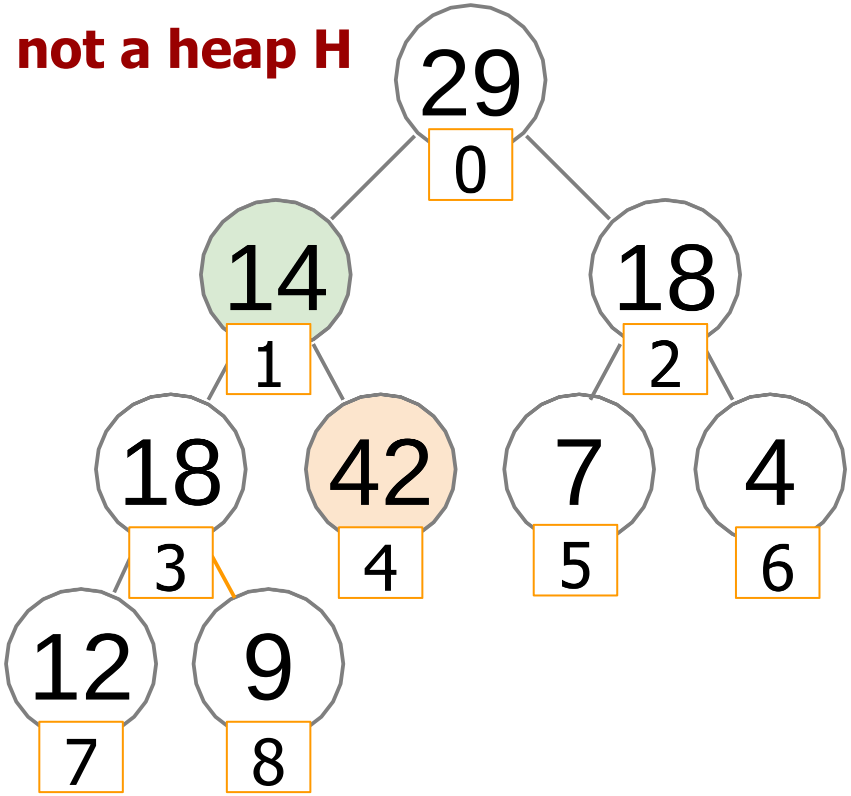
➔ the next node we need to fix is at position 2 of the array

sift_down(2)

29	14	18	18	42	7	4	12	9
0	1	2	3	4	5	6	7	8

How to Heapify an array

not a heap H



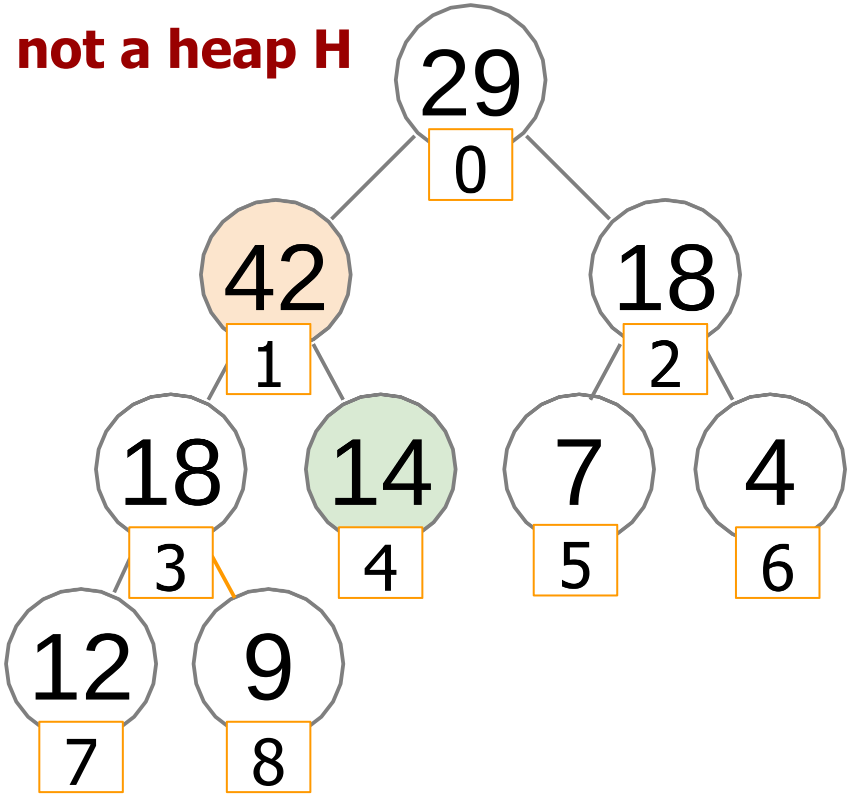
➔ the next node we need to fix is at position 1 of the array

sift_down(1)

29	14	18	18	42	7	4	12	9
0	1	2	3	4	5	6	7	8

How to Heapify an array

not a heap H

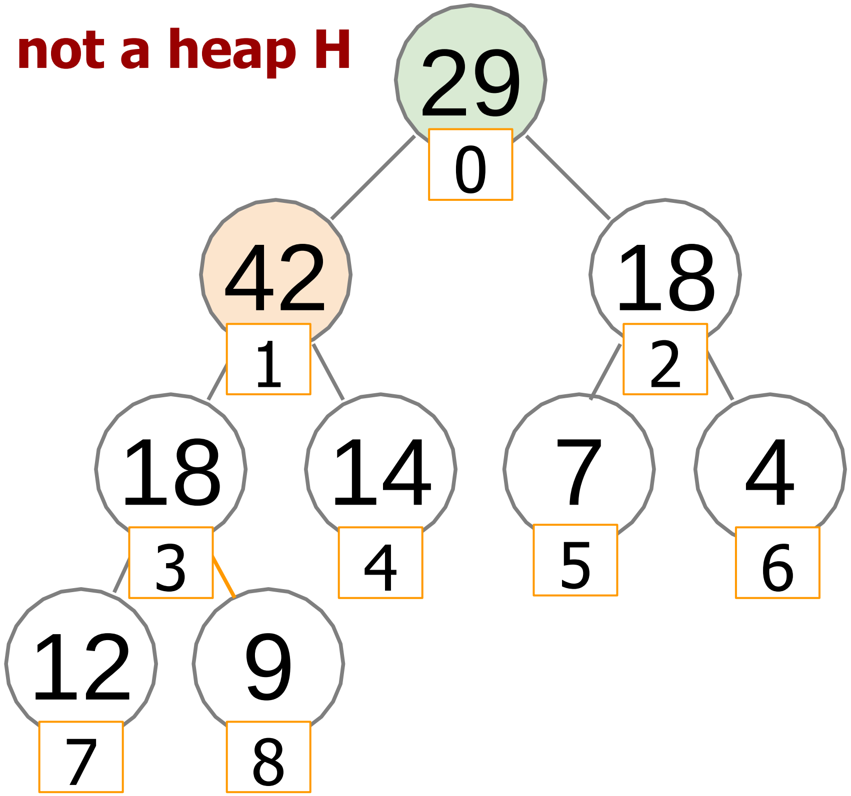


- ➔ the next node we need to fix is at position 1 of the array

sift_down(1)

29	42	18	18	14	7	4	12	9
0	1	2	3	4	5	6	7	8

How to Heapify an array



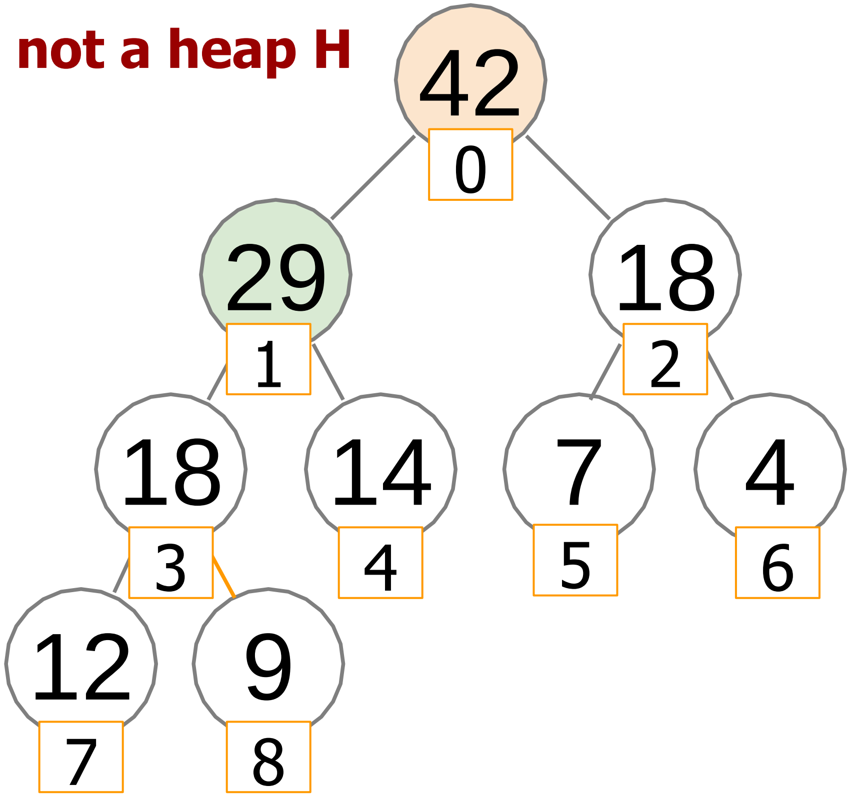
→ Finally, we fix the root at position 0

sift_down(0)

29	42	18	18	14	7	4	12	9
0	1	2	3	4	5	6	7	8

How to Heapify an array

not a heap H

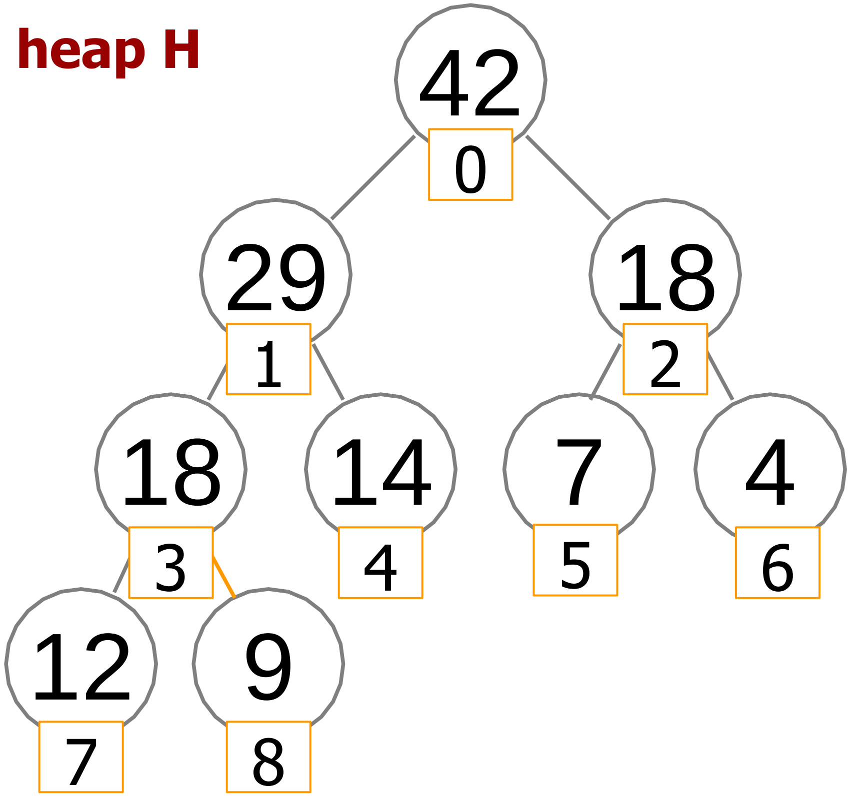


- Finally, we fix the root at position 0

sift_down(0)

42	29	18	18	14	7	4	12	9
0	1	2	3	4	5	6	7	8

How to Heapify an array



42	29	18	18	14	7	4	12	9
0	1	2	3	4	5	6	7	8

- → We rearranged the elements of the input array such that it is now a heap
- → Next, we can use *dequeue* inside the array itself to sort it in-place

First - turn Array into a Heap

```
Heapify (array $A$ of size $n$)
$last \leftarrow n - 1$
for $i$ from $\lfloor (last - 1)/2 \rfloor$ down to 0:
        sift_down ($i$)
```

Group Work

heapify the following array:

10, 85, 15, 70, 20, 60, 30, 50, 65, 40

What is the state of array X after the first *sift_down* in heapify(X)?

Converting X into **max**-heap

X	10	85	15	70	20	60	30	50	65	40
	0	1	2	3	4	5	6	7	8	9

A.

85	10	15	70	20	60	30	50	65	40
0	1	2	3	4	5	6	7	8	9

B.

10	85	15	70	65	60	30	50	20	40
0	1	2	3	4	5	6	7	8	9

C. None of the above

In-place Heap Sort

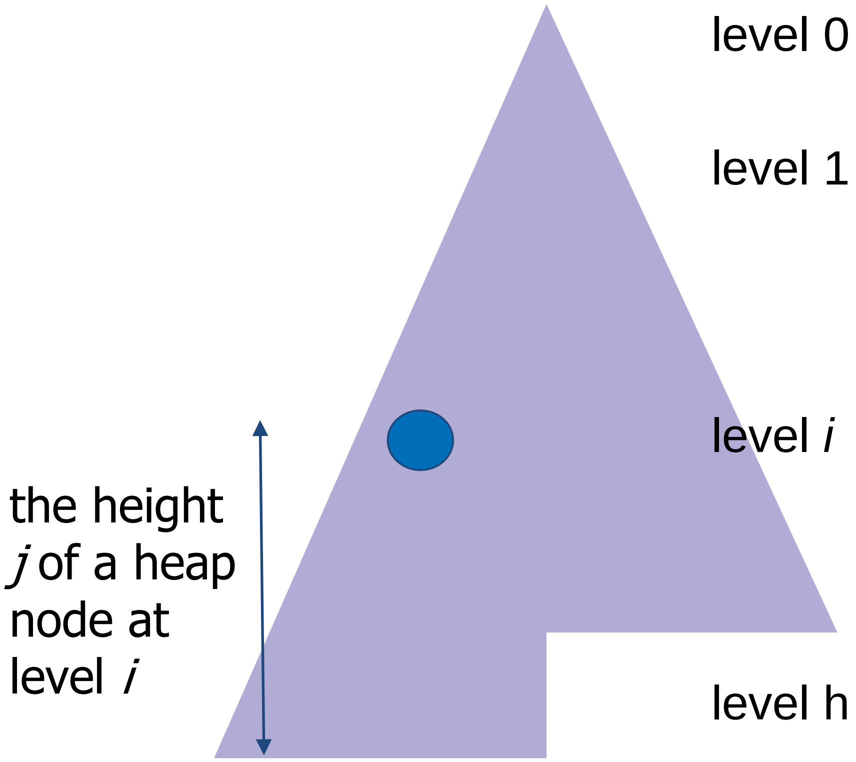
```
HeapSort (array $A$ of size $n$)

Heapify ($A$)
m ← n
repeat $(n - 1)$ times:
    swap $A[0]$ and $A[m-1]$
    $m \leftarrow m - 1$
    sift_down (heap of size $m$, 0)
```

No additional space (in-place)

Run-time of *Heapify*

- The running time of *Heapify* is $O(n \log n)$ since we call *sift_down* for $O(n)$ nodes
- If a node is a leaf then we do not call *sift_down* on it
- If a node is close to the leaves, then sifting it down does not take $\log n$
- We have many such nodes!
- Is our estimate of the running time of *Heapify* too pessimistic?

The height of nodes at level *i*

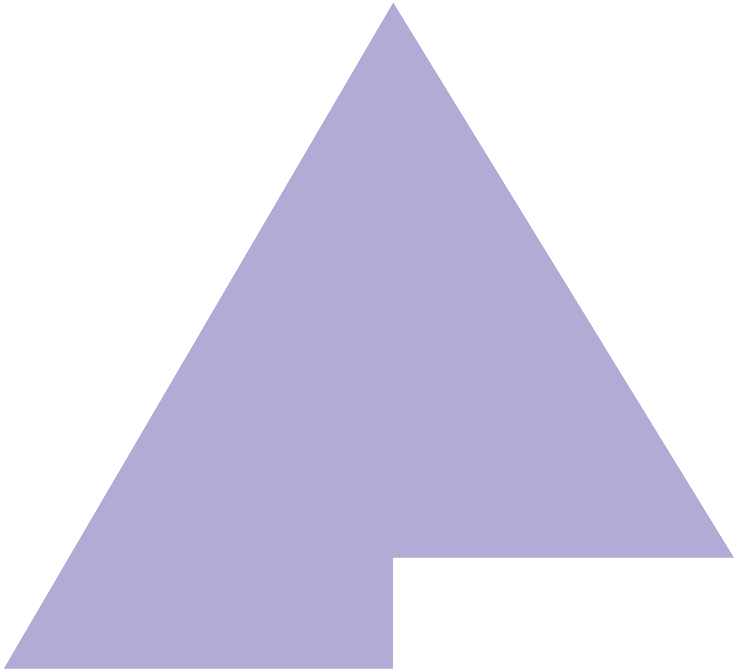


Definition

If we count levels of the heap from top to bottom, then the ***height*** of a heap node at level *i* is defined to be $j = h - i$, where *h* is the total height of the heap

When we are repairing the heap, for each node at level *i* we need to swap at most *j* values

Run-time of Heapify



level	# nodes	node height
h-h	2^{h-h}	h
	…	…
h-2	2^{h-2}	2
h-1	$\leq 2^{h-1}$	1
h-0	$\leq 2^{h-0}$	0

Total work:

$$\sum_{j=0}^{h} j * 2^{h-j} = 2^h \sum_{j=0}^{h} j * \frac{1}{2^j}$$

, where *j* represents the height of the nodes at each of 0…*h* tree levels

This expression evaluates to O(*n*)

$$\boxed{2^h \sum_{j=0}^{h} j * \frac{1}{2^j}} \le 2^h * 2 = O(2^h) = O(2^{\log n}) = O(n)$$

The running time of Heapify is ***O(n)***

To convert an arbitrary array into a heap takes
linear time and no additional space!

In-place Heap Sort

`HeapSort` (array *A* of size *n*)

```
Heapify (A)
m ← n
repeat (n – 1) times:
    swap A[0] and A[m-1]
    m ← m – 1
    sift_down (heap of size m, 0)
```

No additional space (in-place)

What is the running time of an improved heapsort?

A. O(1)

B. O(log n)

C. O(n)

D. O(n * log n)

E. None of the above

Group Work

Sort the heapified array using the last step of in-place HeapSort

Max-heap	85	70	60	65	40	15	30	50	10	20
	0	1	2	3	4	5	6	7	8	9

Application: Top-k Problem

Input: An array A of size n, an integer $1 \le k \le n$.

Output: ***k largest*** elements of A (top-k).

Can be solved in time: $O(n) + O(k \log n)$