

COM1009

Introduction to Algorithms and Data Structures

Topic 05: HeapSort

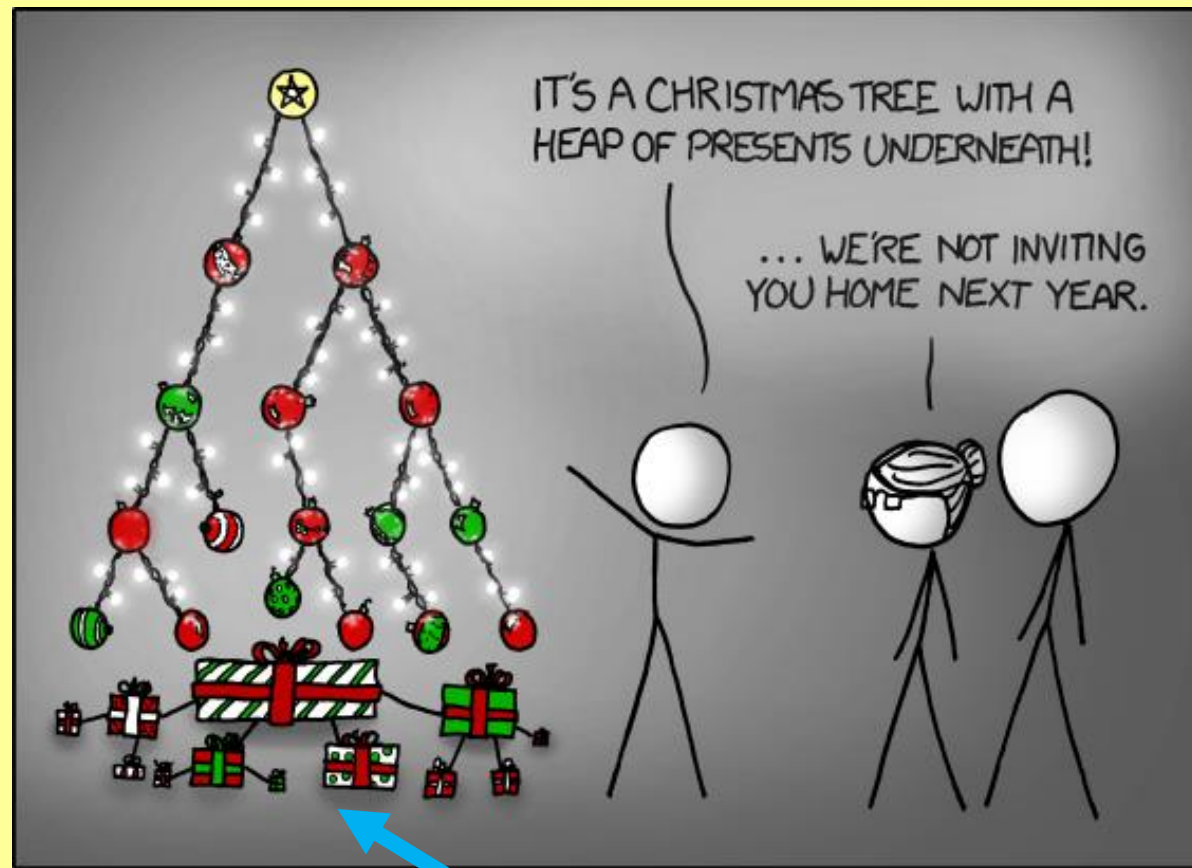
Essential Reading:
Sections 6.1-6.4

► Aims of this lecture

- To introduce **heaps**, a type of binary tree
 - How to re-arrange input data into a heap
 - How to remove the largest element easily
- Switching viewpoints: array or binary tree?
- To introduce the **HeapSort** algorithm (**fast** and **in-place**)
 - arrange the data as a heap, then keep removing the largest entry until everything's dealt with

► Max-Heaps

- A (max-)heap is a **binary tree** of objects with keys that can be compared
- **No parent is smaller than its children**

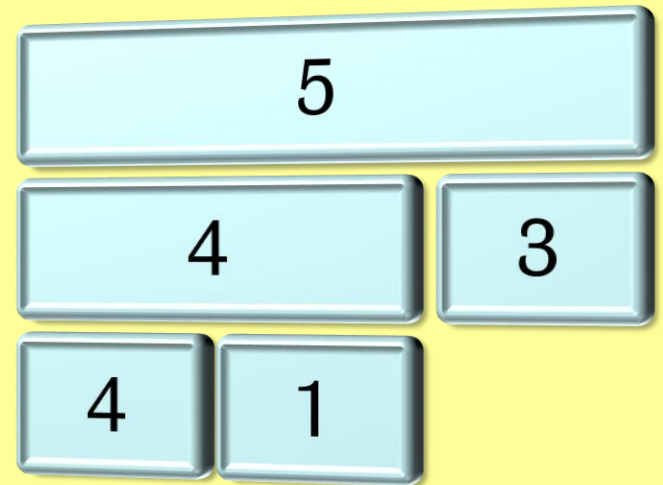
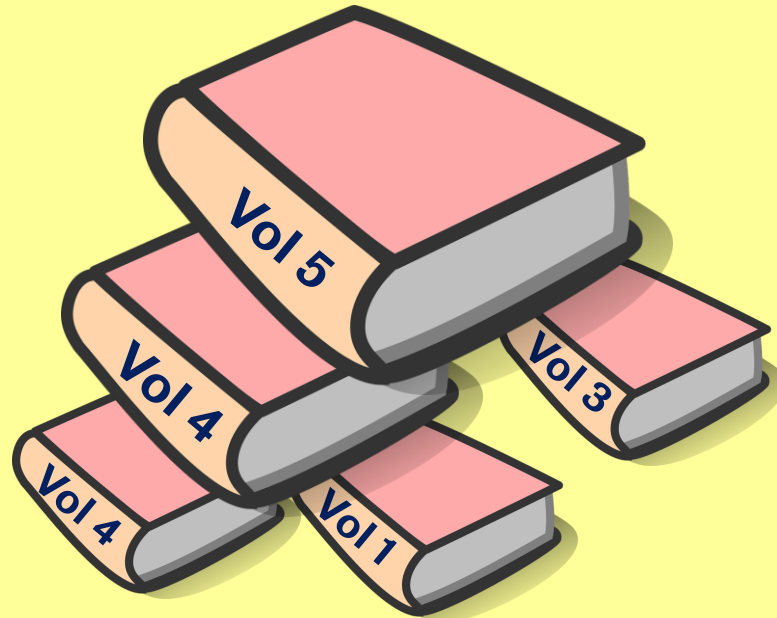


<https://xkcd.com/835/>

heap of presents

[**Min-heaps** are similar, except **no parent is bigger** than its children.]

► Heap examples



Binary tree: each object has at most 2 objects directly underneath it

Max-Heap property: No key is smaller than the keys underneath it

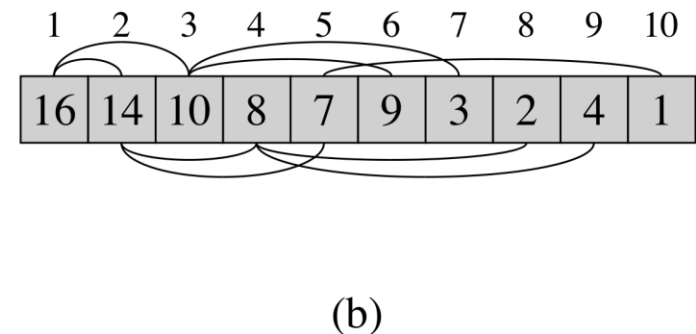
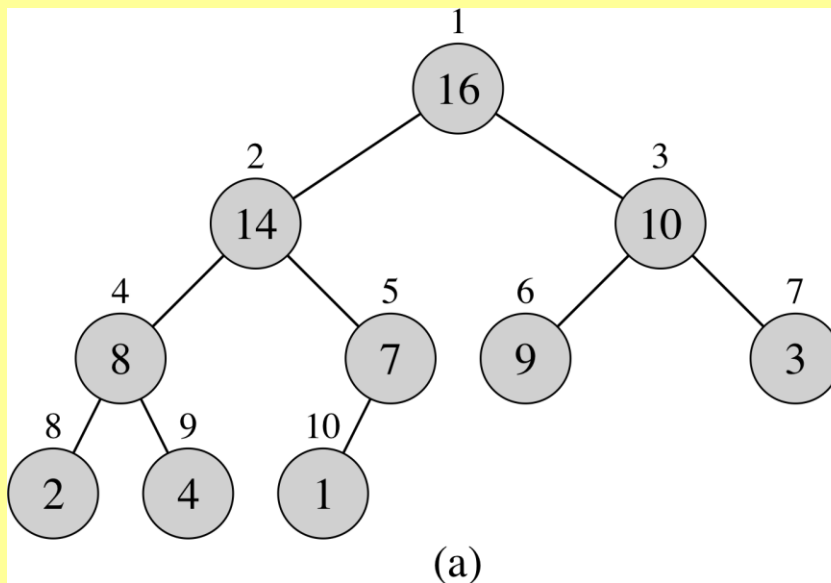
► Watch the Sorting Algorithm Demos

Mike's Videos

The screenshot shows a web browser window with the URL vle.shef.ac.uk/ultra/courses/_90030_1/c/outline. The left sidebar contains a navigation menu for the course 'COM1009 Introduction to Algorithms and Data Structures (SPRING 2020-21)'. The menu items include: Welcome, Module Overview, Slides, Exercises, Study Guide, Ask for Help, Announcements, Staff Contacts, Weekly Lectures and Tutorials, Lecture Slides and Exercise Sheets, Click here for the online sessions and recordings, **Slides, Videos and Other Resources** (highlighted with a yellow circle), Mike's Videos, Reading List, Other Resources, Assessment and Feedback, Assessment, My Marks, Encore Lecture Captures, and Course Management. A red arrow points from the text 'Mike's Videos' to the 'Mike's Videos' link in the sidebar. The main content area displays a grid of video thumbnails for various topics, including: Topic 2: Asymptotic notation and runtime, Topic 2(a) About today's lecture, Topic 2(b) Recap of InsertionSort, Topic 2(c) Recap of InsertionSort runtimes, Topic 2(d) Why we focus on the worst case, TOPIC 3: ELEMENTARY DATA STRUCTURES, Topic 3(a) Introduction and dynamic sets, Topic 3(b) Finding entries in an array is quick, Topic 3(c) Runtime of stack operations, Topic 3(d) Runtime of queue operations, SORTING ALGORITHM DEMOS - 1. INSERTION SORT, com1009 insertion sort, com1009 insertion sort - best and worst case, SORTING ALGORITHM DEMOS - 2. MERGESORT, com1009 mergesort - how merging works, com1009 mergesort - runtime analysis 1 - divide, com1009 mergesort - runtime analysis 2 - merge, com1009 mergesort - runtime analysis 3 - worst..., and SORTING ALGORITHM DEMOS - 3. HEAPSORT, com1009 heapsort 1 - redrawing an array as a tree, com1009 heapsort 2 - arranging the data as a heap, and com1009 heapsort 3 - extracting the sorted data.

► Storing a binary tree in an array

- Elements are arranged row by row from left to right (the last level may be incomplete)

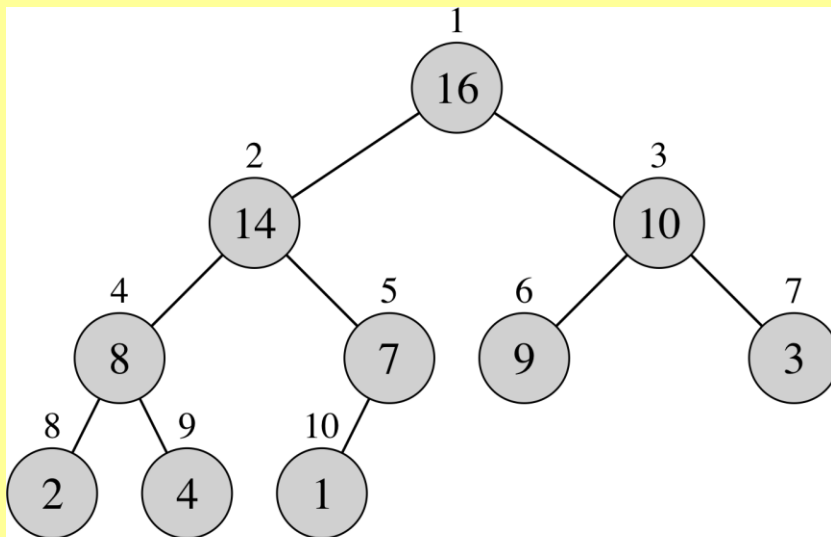


- Navigate through the “tree” using these operations:

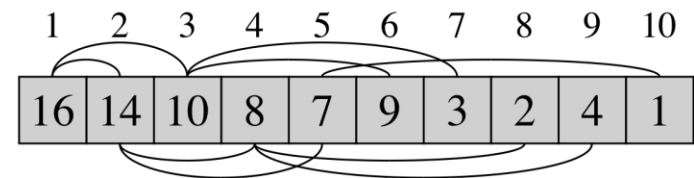
$$\text{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor \text{ (“floor of } i/2\text{”), } \text{Left}(i) = 2i, \text{ Right}(i) = 2i + 1$$

► Heap Properties

- **Max-heap property:** parents are never smaller than their children: $A[\text{Parent}(i)] \geq A[i]$
- In a max-heap, the **root** always stores a **largest** element.
 - The root is **the first entry in the array** version of the heap



(a)



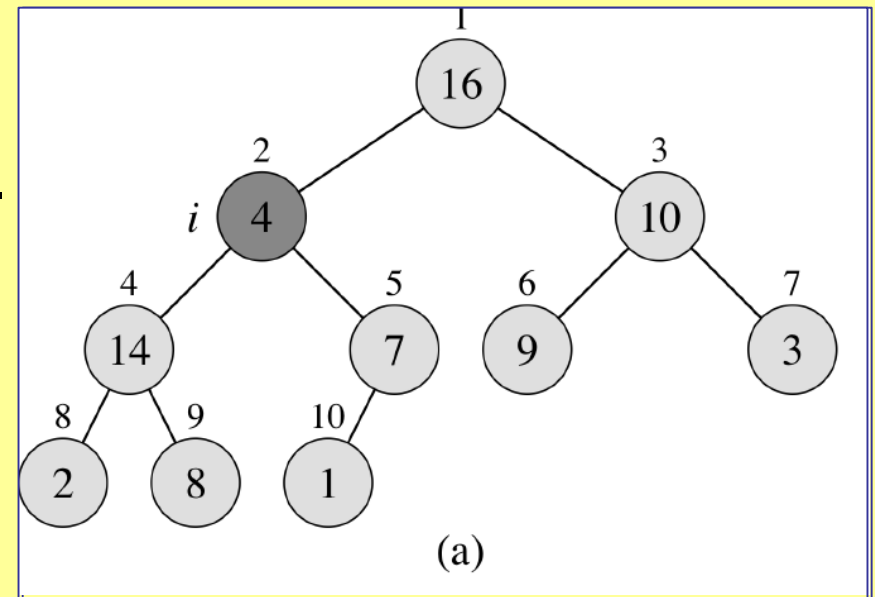
(b)

► Procedures

1. **Max-Heapify**: key to maintaining the max-heap property.
 2. **Build-Max-Heap**: produces a max-heap from an unordered array.
 3. **Heapsort**: sorts an array in place.
- New variable **A.heap-size** indicates how many elements of A are stored in a heap: $0 \leq \text{A.heap-size} \leq \text{A.length}$.
 - Decreasing A.heap-size by 1 effectively removes the last element from the heap (we imagine a heap without it)
 - There are analogous operations for min-heaps: Min-Heapify and Build-Min-Heap.

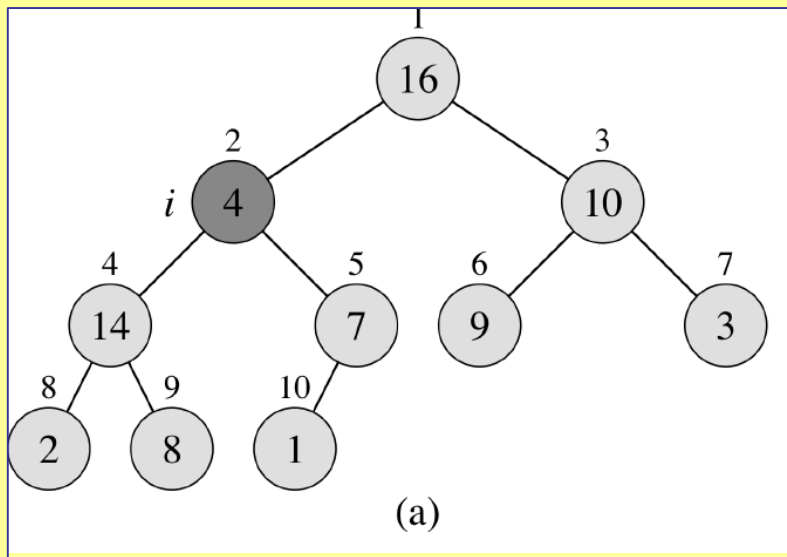
► Max-Heapify(A, i)

- Assumes subtrees $\text{Left}(i)$ and $\text{Right}(i)$ are max-heaps, but max-heap property might be violated in root of subtree at i .
 - “Subtree x ”: the part of the tree including x and everything below.
- Lets the value at $A[i]$ “float down” if necessary, to restore max-heap property at i
- At the end of Max-Heapify the subtree at i is a max-heap.



► Max-Heapify: informal and in pseudocode

- Compare $A[i]$ with all existing children
- If **largest child** is larger than $A[i]$, swap and recurse on child

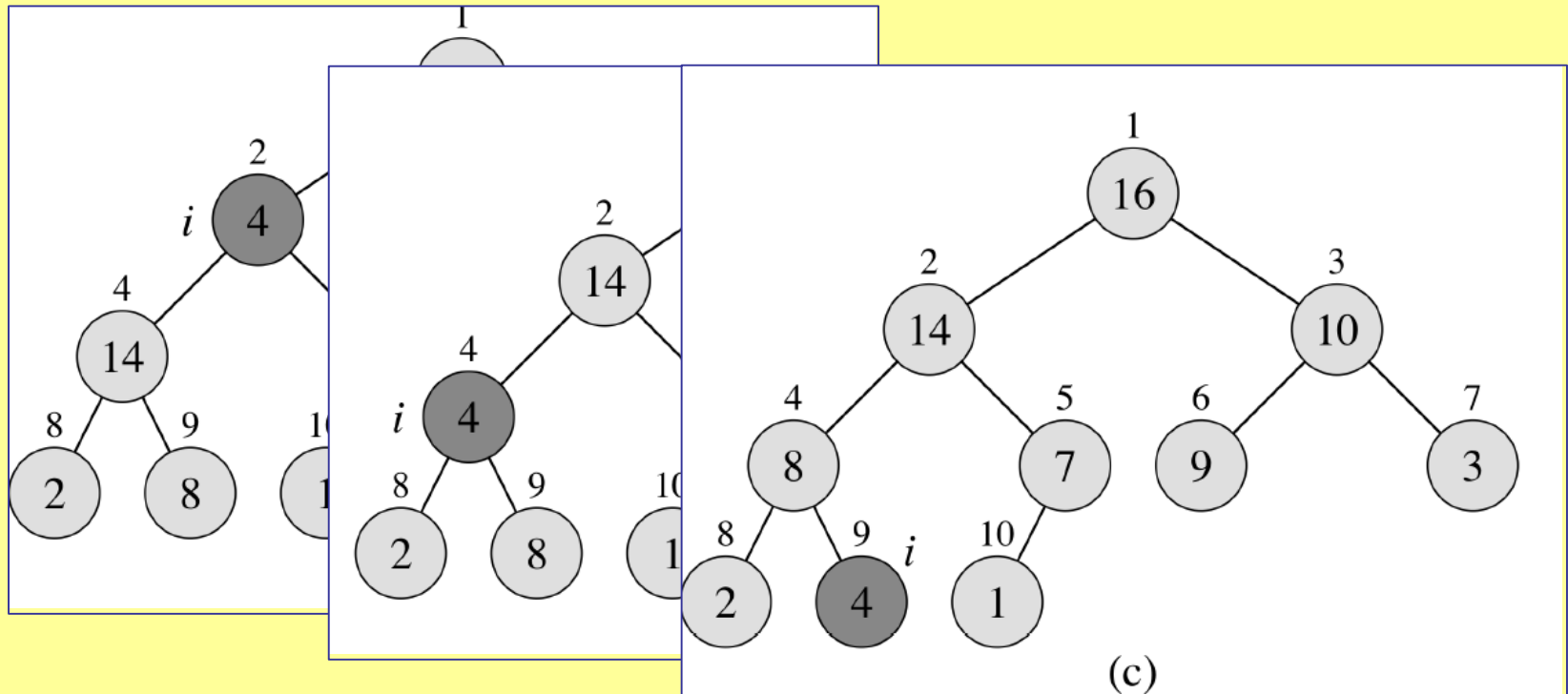


MAX-HEAPIFY(A, i)

```
1:  $l = \text{Left}(i)$ 
2:  $r = \text{Right}(i)$ 
3: if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$  then
4:      $\text{largest} = l$ 
5: else
6:      $\text{largest} = i$ 
7: if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$  then
8:      $\text{largest} = r$ 
9: if  $\text{largest} \neq i$  then
10:    exchange  $A[i]$  with  $A[\text{largest}]$ 
11:    MAX-HEAPIFY( $A, \text{largest}$ )
```

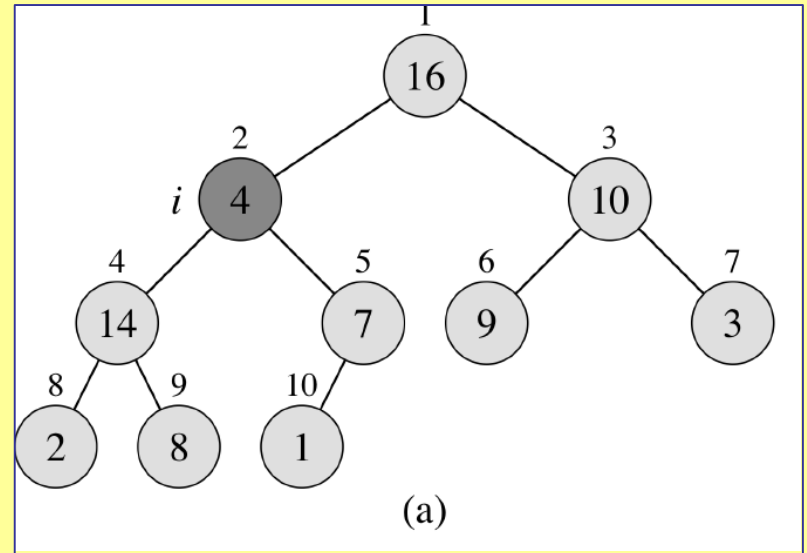
► Max-Heapify: Example

- Compare $A[i]$ with all existing children
- If **largest child** is larger than $A[i]$, swap and recurse on child



► Runtime of Max-Heapify

- Define the **height** of a node as the longest number of simple downward edges from the node to a **leaf**.
- Leaf**: a node without children.
- Max-Heapify takes constant time, $\Theta(1)$, on each level.
- Runtime of Max-Heapify on a node of height h is $O(h)$.
- It's not $\Omega(h)$ as Max-Heapify may stop early, e.g. if heap-property holds at i .
- For leaves $h = 0$ and the time is $O(1)$.



► Turning the initial data into a heap

- Think of the initial array as a binary tree
- Use Max-Heapify repeatedly to make the tree into a heap.
 - Start at the last entry in the tree and work leftwards
- Note: roughly half the nodes – those in $A \left[\left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \right), \dots, n \right]$ – are leaves, so they're already max-heaps.

BUILD-MAX-HEAP(A)

```
1: A.heap-size = A.length
2: for  $i = \lfloor A.length/2 \rfloor$  downto 1 do
3:     MAX-HEAPIFY( $A, i$ )
```

► Correctness of Build-Max-Heap

BUILD-MAX-HEAP(A)

1: $A.\text{heap-size} = A.\text{length}$
2: **for** $i = \lfloor A.\text{length}/2 \rfloor$ **downto** 1 **do**
3: MAX-HEAPIFY(A, i)

- **Loop invariant:** At the start of each iteration of the for loop, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap.
- **Initialisation:** true for leaves $\lfloor \frac{n}{2} \rfloor + 1, \dots, n$.
- **Maintenance:** by loop invariant, all children of i are roots of max-heaps (as their numbers are larger than i).
Then Max-Heapify(A, i) turns the subtree at i into a max-heap.
- **Termination:** the loop terminates at $i = 0$, hence node 1 is the root of a max-heap.

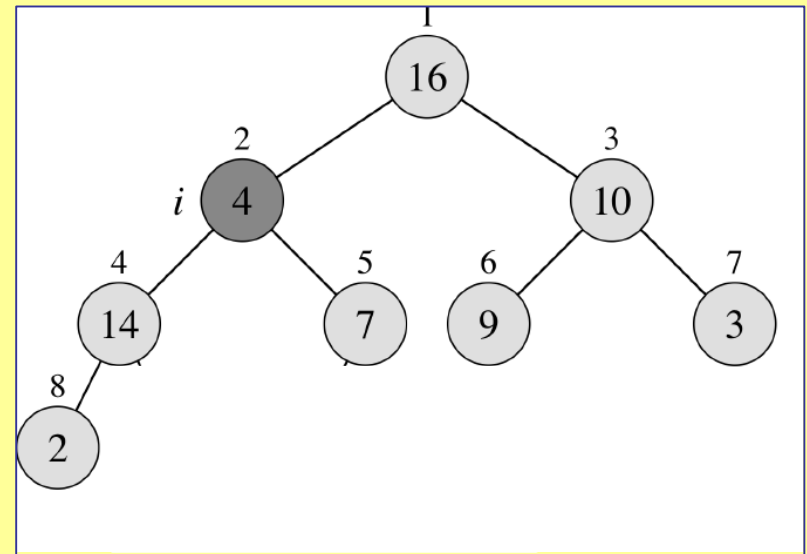
► Bounding the height of a heap

- The height of the heap (= height of the root) is at most $\log n$.
- **Proof:** the number n of elements in a heap of height h is

- Doubling on each level
- At least 1 node on the last level
- Hence in total at least

$$1 + 2 + 4 + \dots + 2^{h-1} + 1 = 2^h$$

(we used $\sum_{i=0}^{k-1} 2^i = 2^k - 1$)



- So size and height are related as $n \geq 2^h \Leftrightarrow \log n \geq h$

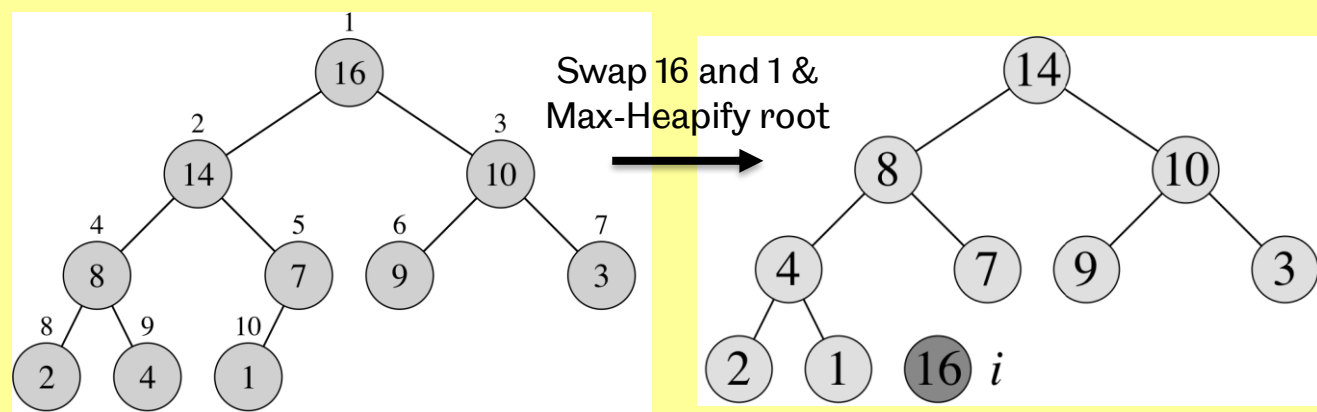
► Runtime of Build-Max-Heap

- Just seen: the height of the root is at most $\log n$.
 - So all nodes have height at most $\log n$.
 - Max-Heapify does $O(h)$ work at height h (and less at lower heights)
 - So every call to Max-Heapify takes time $O(\log n)$.
- Build-Max-Heap calls Max-Heapify $O(n)$ times.
- Total time is at most $O(n) \cdot O(\log n) = O(n \log n)$.
 - The time can be improved to $O(n)$ since most nodes have small height. See the book!
 - $O(n \log n)$ is sufficient for us, though.

► HeapSort

- Ideas:

1. Build a max-heap, such that the root contains largest element.
2. Swap the root with the last element of the heap/array.
3. Discard the last element from the heap by reducing heap.size.
(We simply imagine a smaller heap.)
4. Call $\text{Max-Heapify}(A, 1)$ to restore heap property at the root.



$\text{HEAPSORT}(A)$ **Runtime: $O(n \log n) + O(n)O(\log n) = O(n \log n)$**

```

1: BUILD-MAX-HEAP( $A$ )
2: for  $i = A.\text{length}$  downto 2 do
3:     exchange  $A[1]$  with  $A[i]$ 
4:      $A.\text{heap-size} = A.\text{heap-size} - 1$ 
5:     MAX-HEAPIFY( $A, 1$ )
    
```

Do the following $O(n)$ times:

$O(1)$

$O(1)$

$O(\log n)$

► Summary

- **HeapSort** sorts **in place** in time $O(n \log n)$.
 - Building a Heap in time $O(n)$.
 - Extracting the largest element and restoring the heap-property in total time $O(n \log n)$.
- The use of appropriate **data structures** can speed up computation (in contrast to SelectionSort).
 - The heap “memorises” information about comparisons of elements.
 - The heap is imaginary, no objects/pointers required!
- Outlook for later: heaps also play a role in **Priority Queues**.