COM1009
Introduction to Algorithms and Data Structures

Topic 05: HeapSort

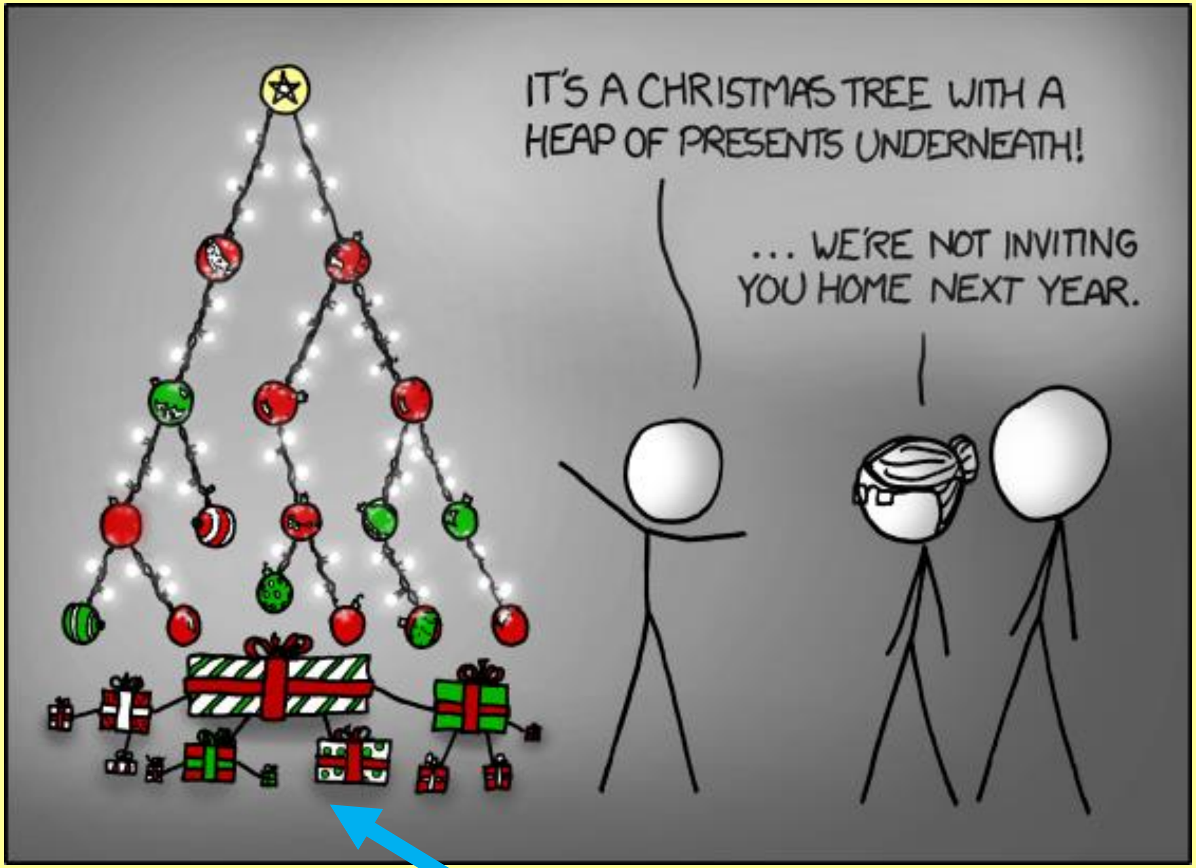
Essential Reading:
Sections 6.1-6.4

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

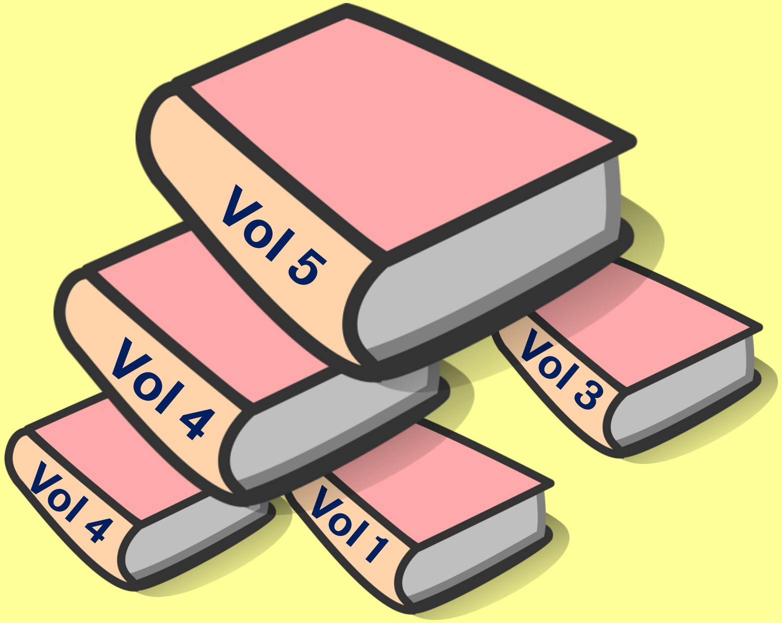


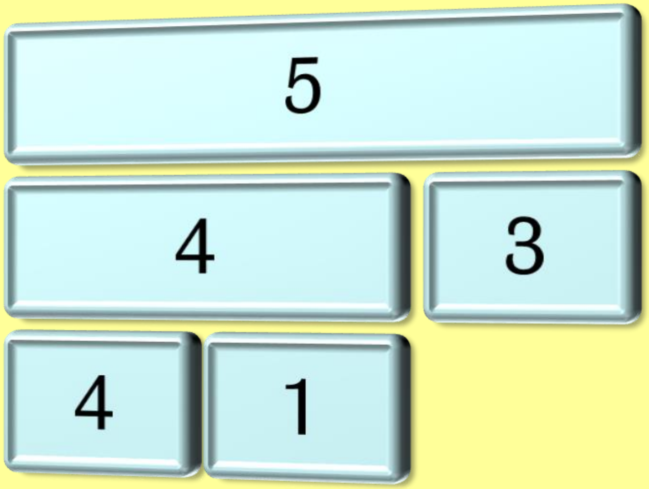
https://xkcd.com/835/

heap of presents

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

Heap examples

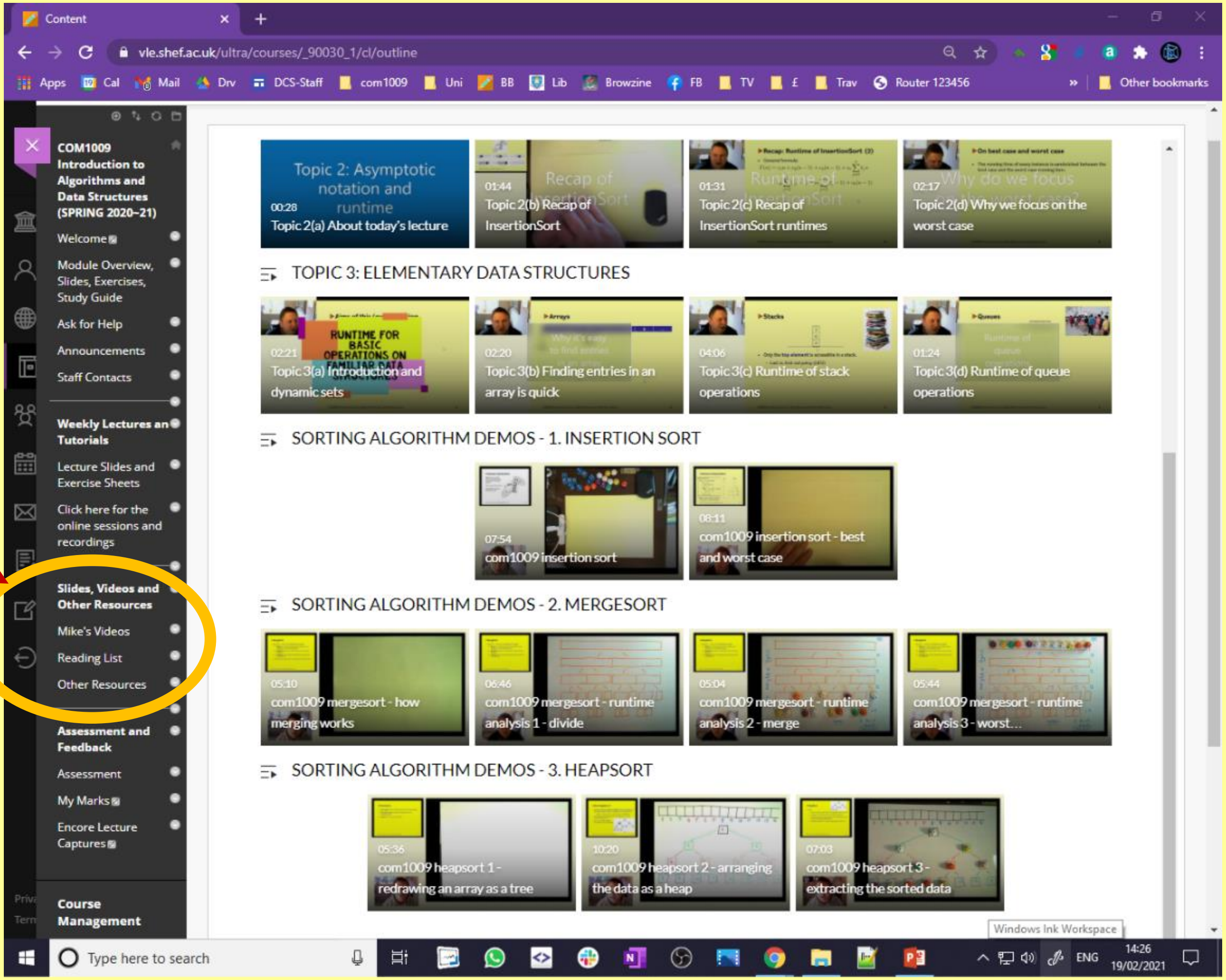




Binary tree: each object has at most 2 objects directly underneath it
Max-Heap property: No key is smaller than the keys underneath it

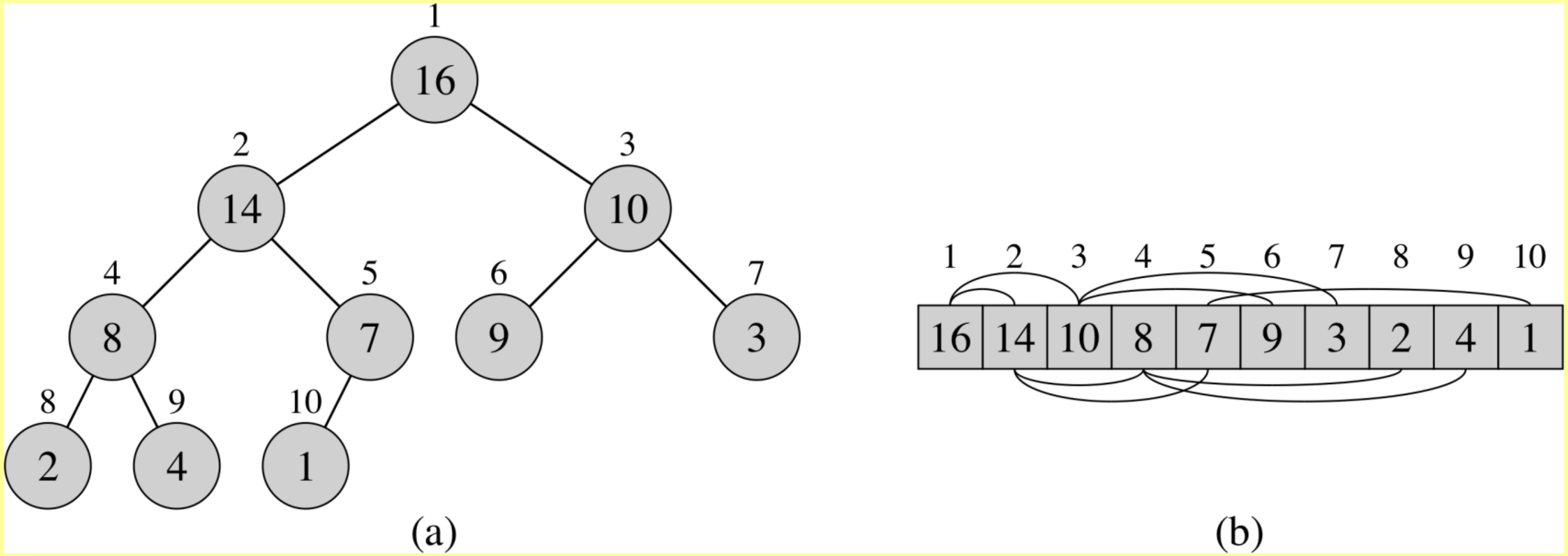
Watch the Sorting Algorithm Demos

Mike's Videos



▶Storing a binary tree in an array

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

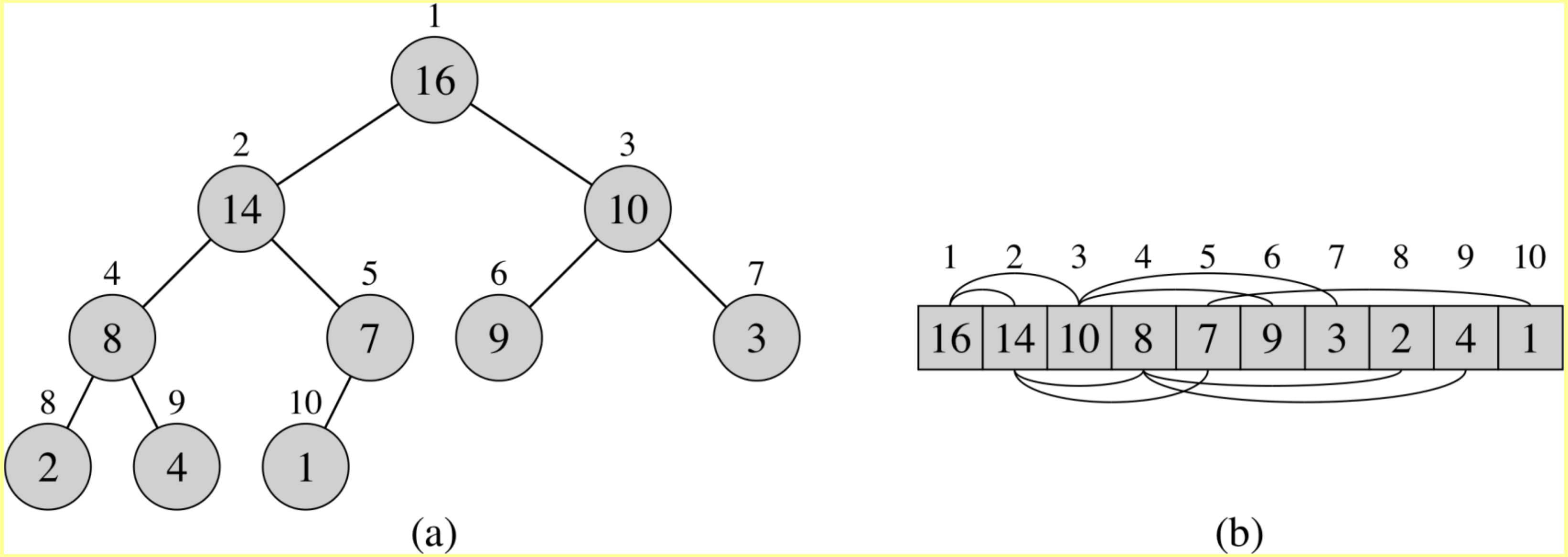


- Navigate through the "tree" using these operations:

$\mathrm{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$ ("floor of *i/2*"), $\mathrm{Left}(i) = 2i$, $\mathrm{Right}(i) = 2i + 1$

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



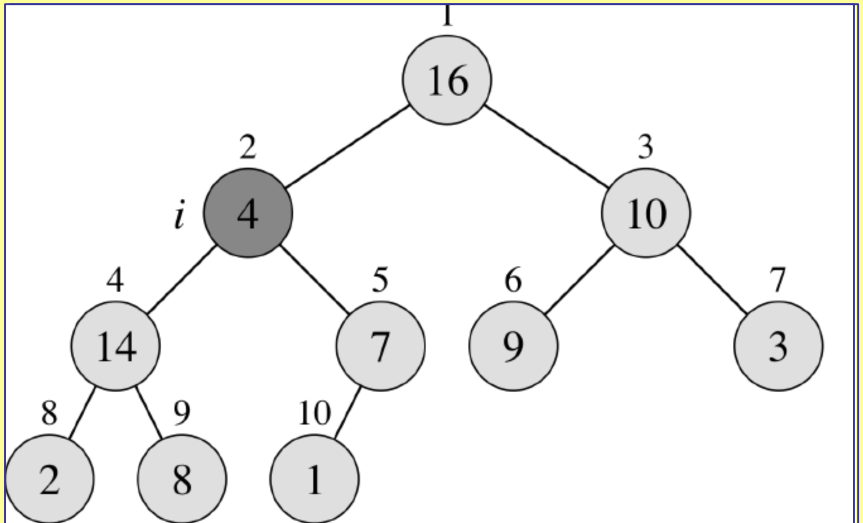
Procedures

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: $0 \leq A.heap\text{-}size \leq A.length$.
  - 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*)

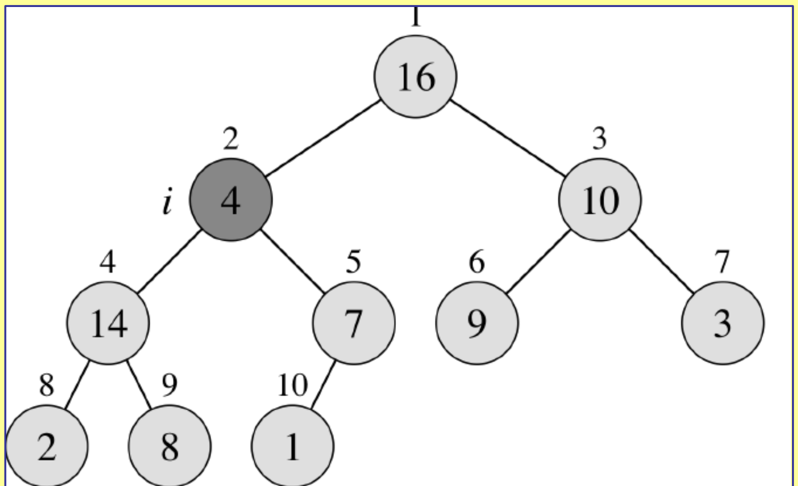
- **Assumes subtrees Left(*i*) and Right(*i*) are max-heaps**, but max-heap property might be violated in root of subtree at *i*.
  - "Subtree x": the part of the tree including x and everything below.
- Lets the value at *A[i]* "float down" if necessary, to restore max-heap property at *i*
- At the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



(a)

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



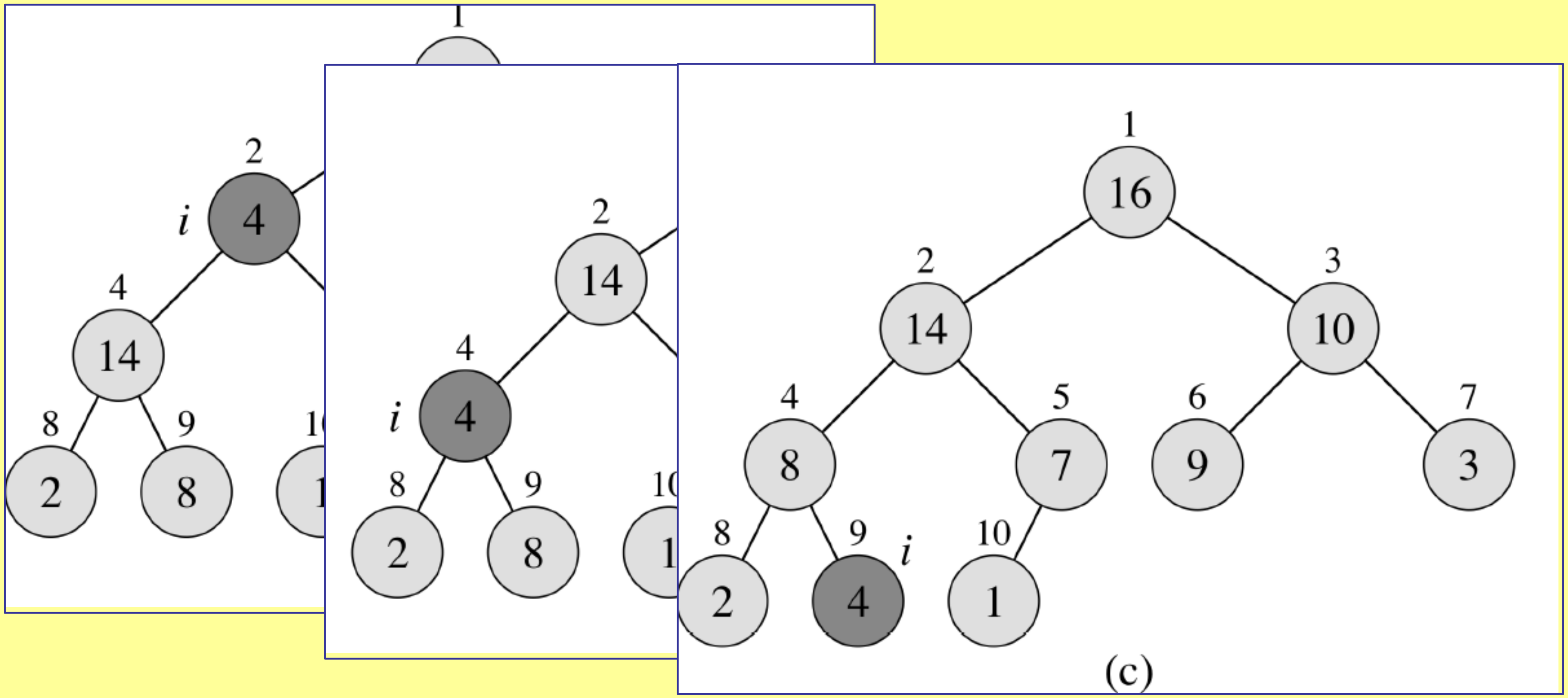
(a)

MAX-HEAPIFY(A, i)

```
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i] then
    largest = l
else
    largest = i
if r ≤ A.heap-size and A[r] > A[largest] then
    largest = r
if largest ≠ i then
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

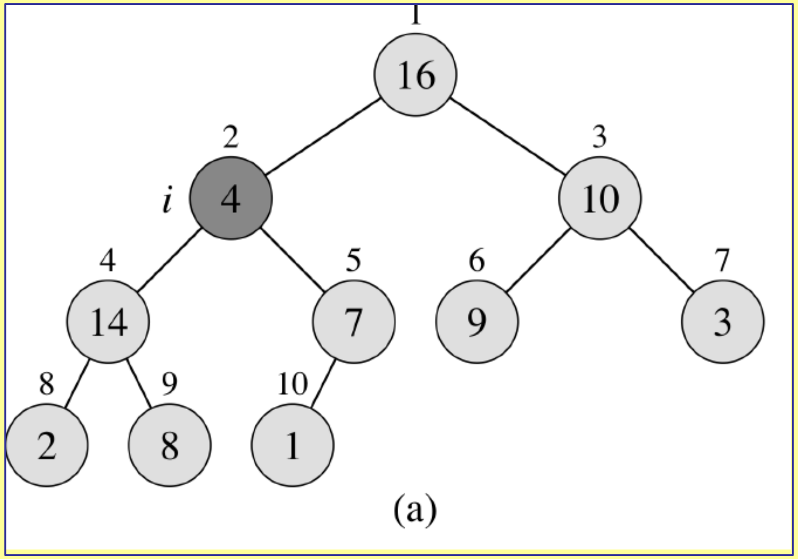
Max-Heapify: Example

- 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 in $A\left[\left(\left\lfloor\frac{n}{2}\right\rfloor + 1\right), \ldots, n\right]$ – are leaves, so they're already max-heaps.

```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1 do
    MAX-HEAPIFY(A, i)
```

Correctness of Build-Max-Heap

Build-Max-Heap(A)

```
1: A.heap-size = A.length
2: for i = ⌊A.length/2⌋ downto 1 do
3:     Max-Heapify(A, i)
```

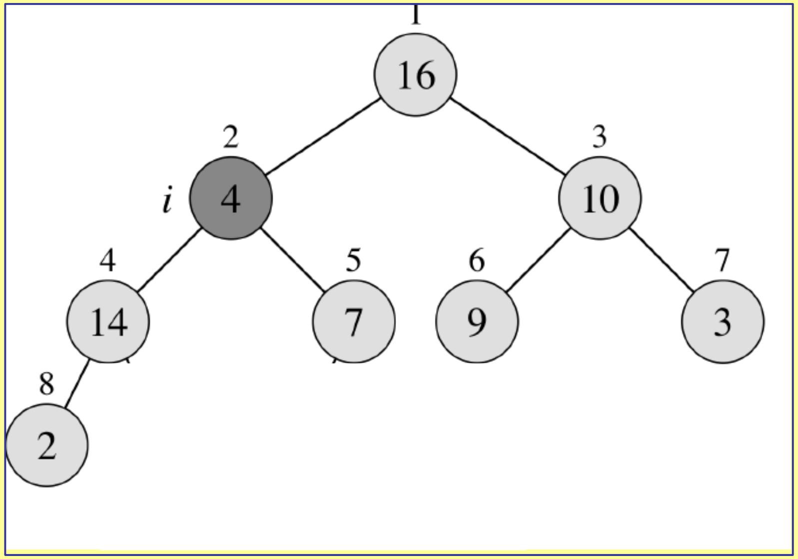
- **Loop invariant:** At the start of each iteration of the for loop, each node $i+1, i+2, \dots, n$ is the root of a max-heap.
- **Initialisation:** true for leaves $\left\lfloor \frac{n}{2} \right\rfloor + 1, \dots, n$.
- **Maintenance:** by loop invariant, all children of i are roots of max-heaps (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

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

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

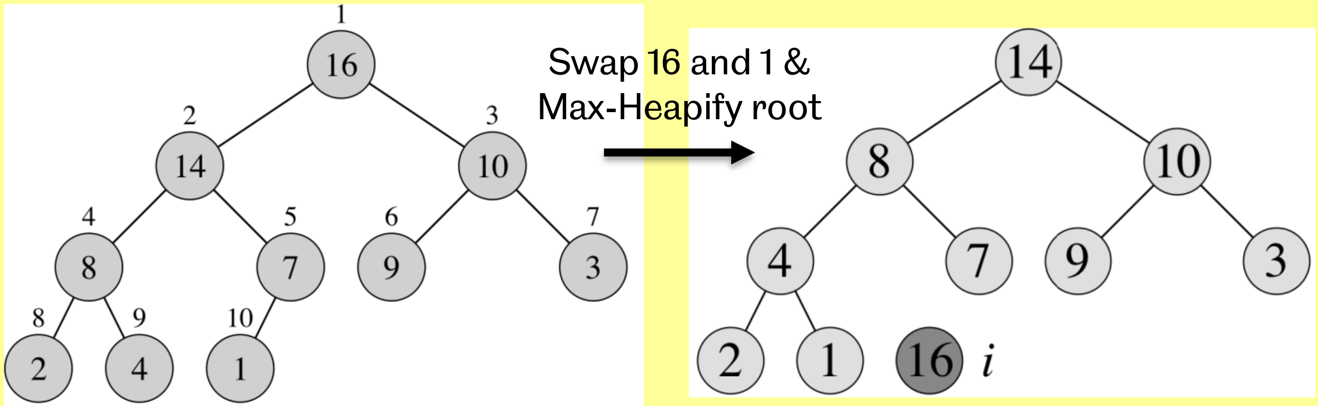


- So size and height are related as $n \geq 2^h \Leftrightarrow \log n \geq h$

▶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 Max-Heapify(*A, 1*) to restore heap property at the root.

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2 do
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

Runtime: O(n.log n) + O(n)O(log n) = O(n.log n)

O(n log n)

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**.