

# CS217 - Data Structures & Algorithm Analysis (DSAA)

## Lecture #4

### **HeapSort**

Prof. Pietro S. Oliveto

Department of Computer Science and Engineering

Southern University of Science and Technology (SUSTech)

`olivetop@sustech.edu.cn`

<https://faculty.sustech.edu.cn/olivetop>

Reading: Chapter 6

## ➤ Aims of this lecture

- To introduce the **HeapSort** algorithm.
- To show how a **clever data structure**, a **heap**, can lead to a **fast** and **in place** sorting algorithm
  - In place:  $O(1)$  additional space.
- To **practice the design and analysis of algorithms**.

## ➤ Idea behind HeapSort

- Idea:
  - Find the largest element.
  - Move it to the end of the array (put another one in its place).
  - Repeat with remaining elements.
- Like SelectionSort but ...
  - SelectionSort compares lots of elements to find the largest.
  - Can we store knowledge gained from these comparisons for the future?
  - Use this knowledge to make future iterations faster!

## ➤ Use your imagination...

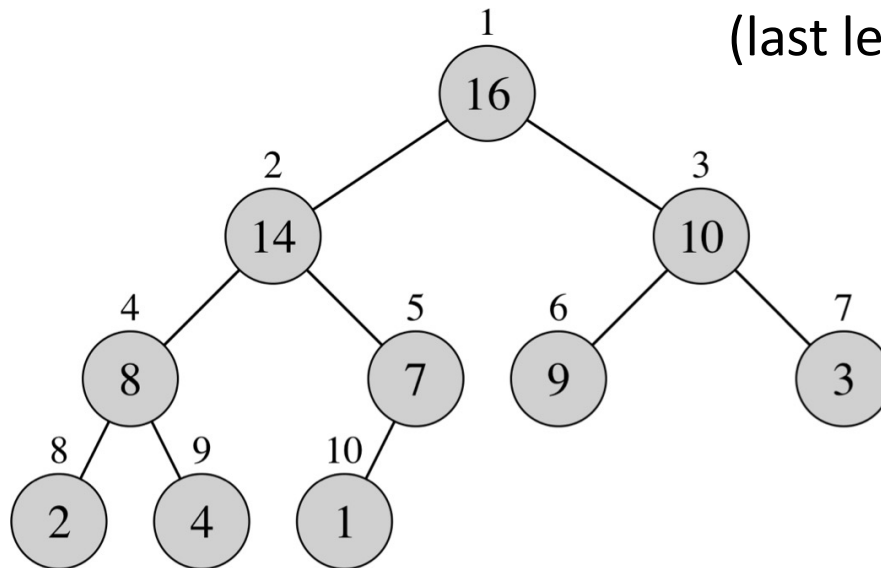


Photo : Thomas Bresson



## ➤ A Heap

- Essentially an array **imagined** as being a **binary tree**!
- Elements are arranged row by row from left to right.  
(last level may be incomplete)



(a)

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

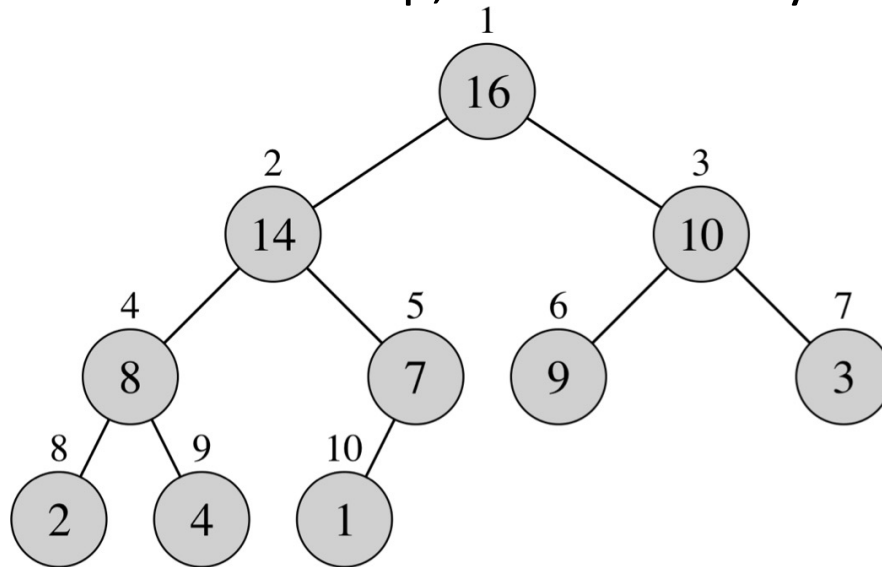
(b)

- Navigate through the array/imaginary tree using these operations:
- $\text{Parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$  ("floor of  $i/2$ "),  $\text{Left}(i) = 2i$ ,  $\text{Right}(i) = 2i + 1$

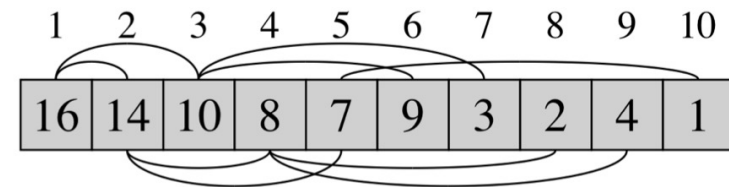
## ➤ Heap Properties

- **Max-heap property**: for every node other than the root, the parent is no smaller than the node,  $A[\textit{Parent}(i)] \geq A[i]$ .
- In a max-heap, the **root** always stores a **largest** element.

→ this is what we want!



(a)



(b)

- **Min-heap property**: for every node other than the root, the parent is no larger than the node,  $A[\textit{Parent}(i)] \leq A[i]$ .

## ➤ Procedures (what do we need)

1. **Build-Max-Heap**: produces a Max-Heap from an unordered array
  2. **Max-Heapify**: maintains the max-heap property once the maximum has been removed
  3. **HeapSort**: sorts an array in place
- New variable `A.heap-size` indicates how many elements of `A` are stored in a heap:  $0 \leq A.\text{heap-size} \leq A.\text{length}$ .
    - 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`.

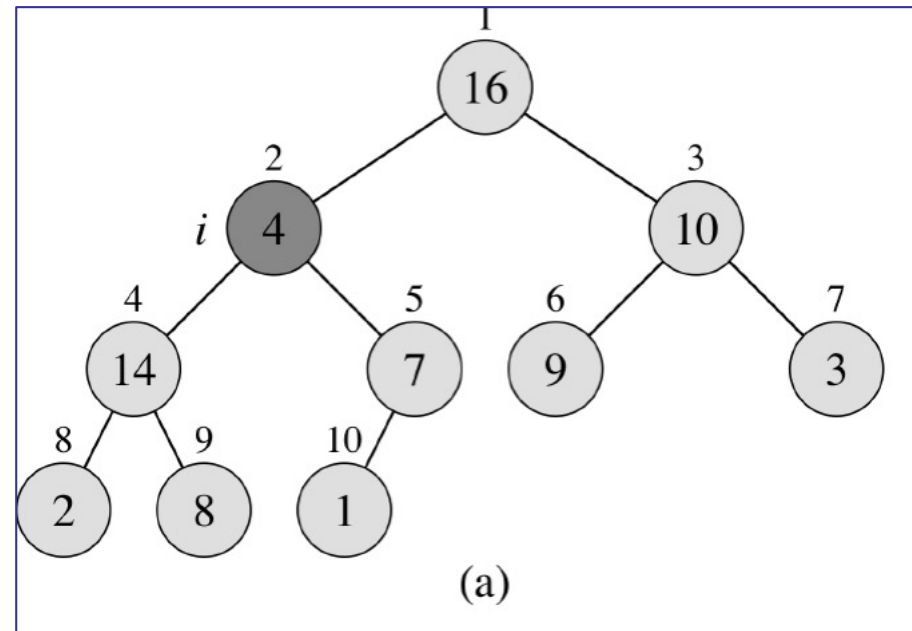
## ➤ Procedures (what do we need)

1. **Build-Max-Heap**: produces a Max-Heap from an unordered array
  2. **Max-Heapify**: maintains the max-heap property once the maximum has been removed
  3. **HeapSort**: sorts an array in place
- New variable `A.heap-size` indicates how many elements of `A` are stored in a heap:  $0 \leq A.\text{heap-size} \leq A.\text{length}$ .
    - 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`.



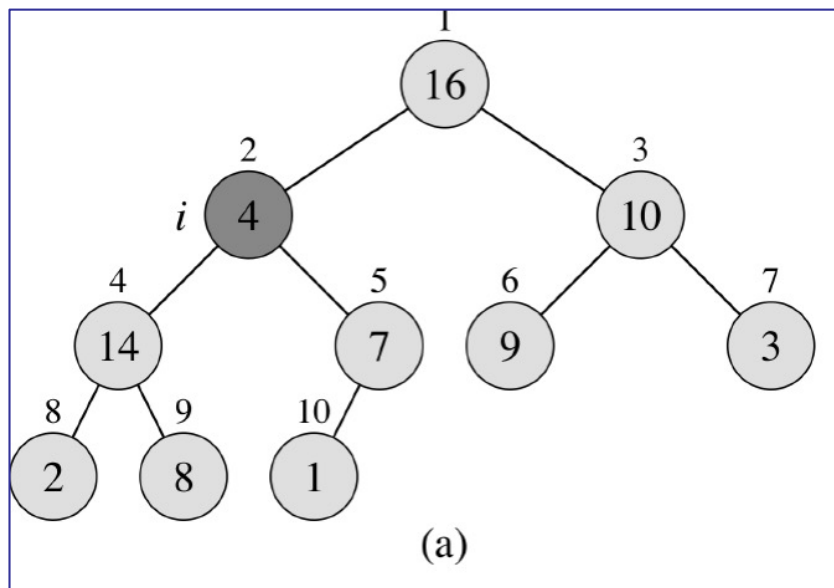
## ➤ Max-Heapify( $A, i$ )

- 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( $A, i$ )

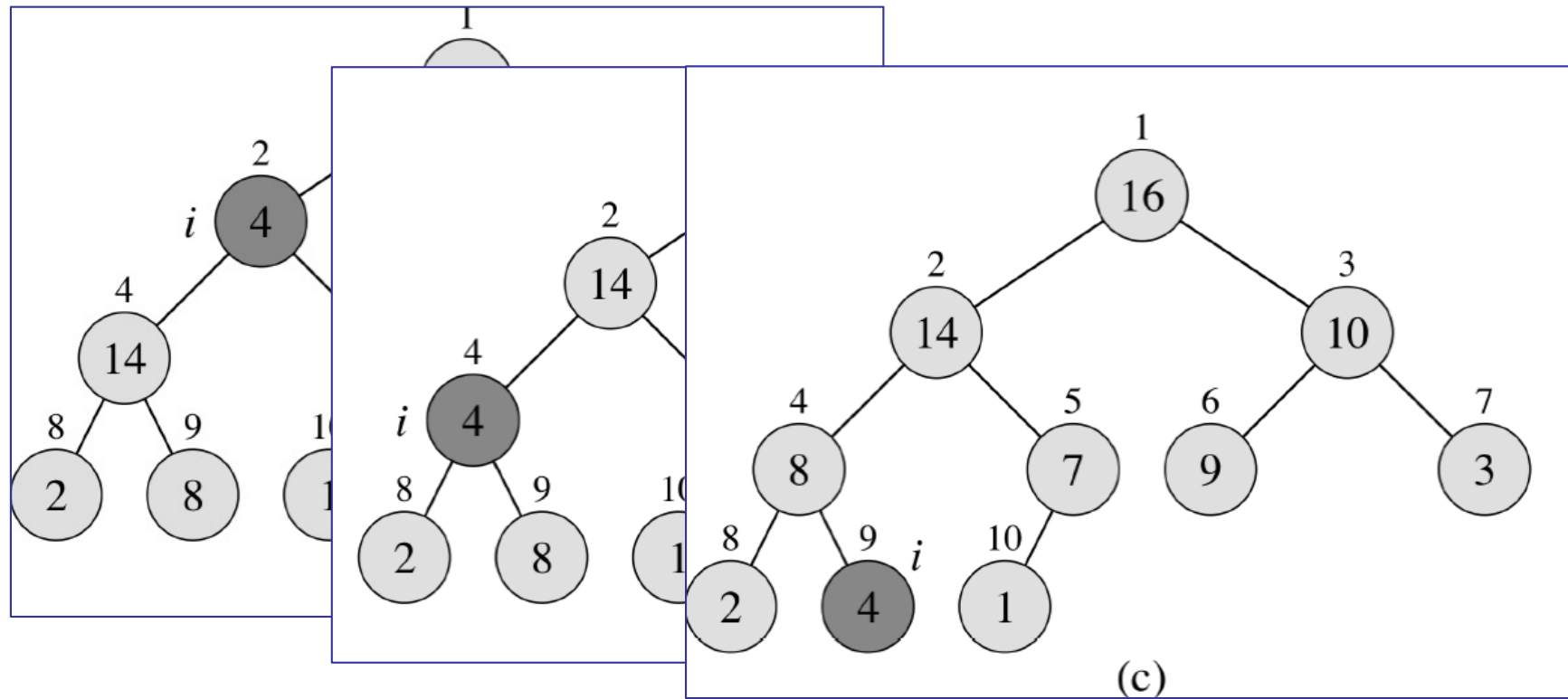
---

```
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.
- Running time 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  $O(1)$ .

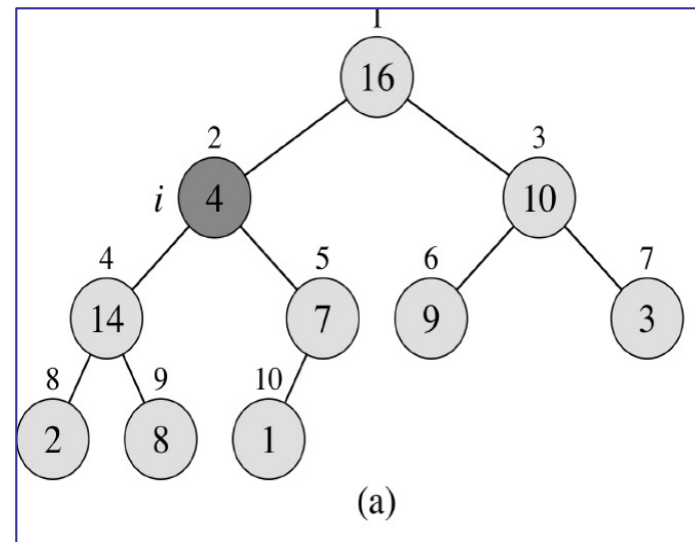
---

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( $A, \text{largest}$ )
```

---



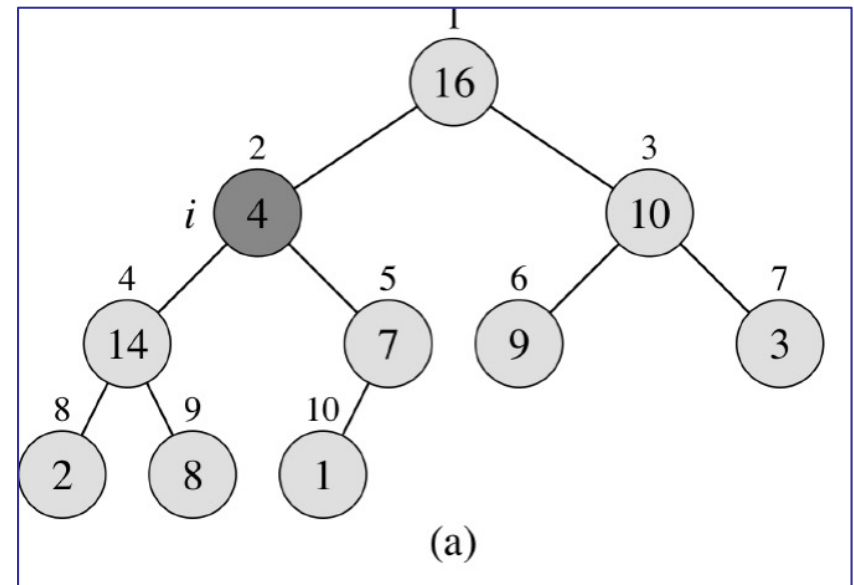
## ➤ Bounding the height of a heap

- **Claim:** the **height of a 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

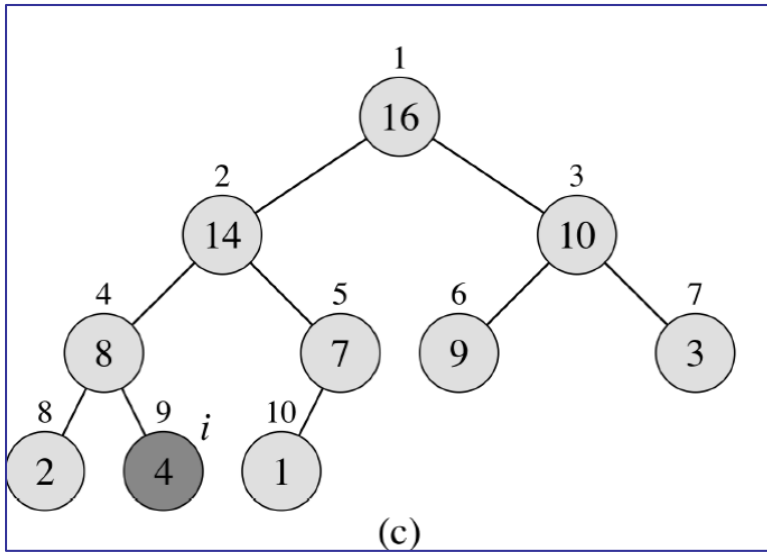
$$1 + 2 + 4 + \dots + 2^{h-1} + 1 = 2^h$$

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



- So size and height are related as  $n \geq 2^h \Leftrightarrow \log n \geq h$
- “the height of the root is at most  $\log n$ ”
- So the runtime of Max-Heapify is  $O(\log n)$

## ➤ Max-Heapify: Correctness



---

MAX-HEAPIFY( $A, i$ )

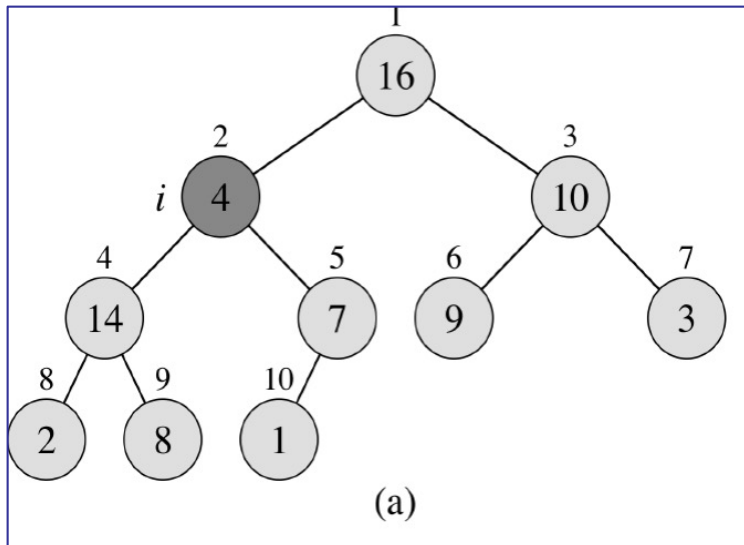
---

```
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( $A, \text{largest}$ )
```

---

- By induction (on the height):
- **Base case:** height = 0 ( $i$  is a leaf)
- Then left( $i$ ) and right( $i$ ) are larger than  $A.\text{heap-size}$  and the algorithm returns a heap!

## ➤ Max-Heapify: Correctness



---

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( $A, \text{largest}$ )
```

---

- By induction (on the height):
- **Inductive case:** assume it works for height  $h = i - 1$  and show it works for  $h = i$
- Then the algorithm swaps  $A[i]$  with the larger between  $\text{Left}(i)$  and  $\text{Right}(i)$  (if any) and one subtree was already a heap and the other will be by inductive hypothesis.

## ➤ Procedures (what do we need)

1. **Build-Max-Heap**: produces a Max-Heap from an unordered array
2. **Max-Heapify**: maintains the max-heap property once the maximum has been removed ✓
3. **HeapSort**: sorts an array in place



## ➤ Building a Heap

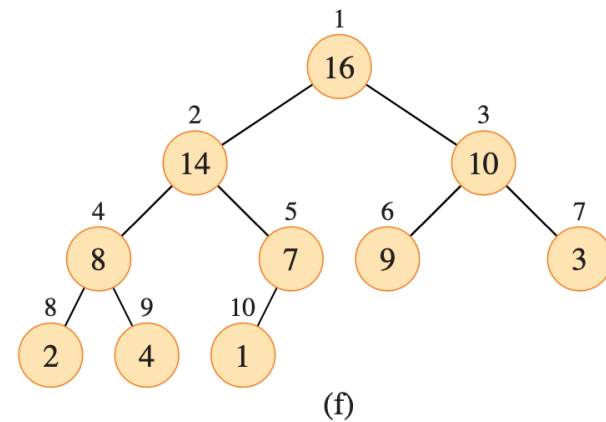
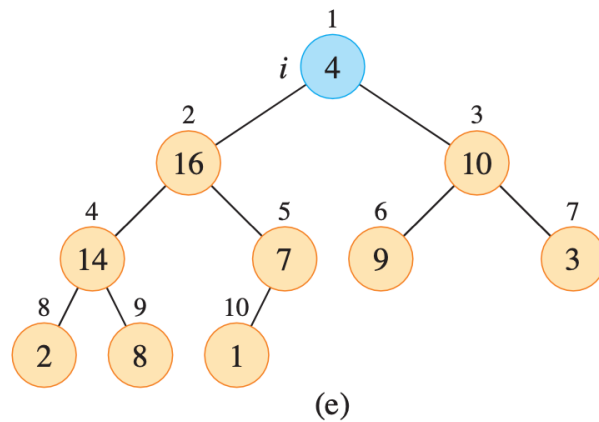
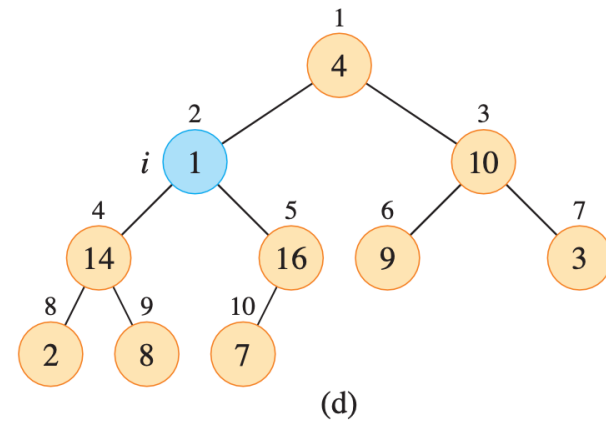
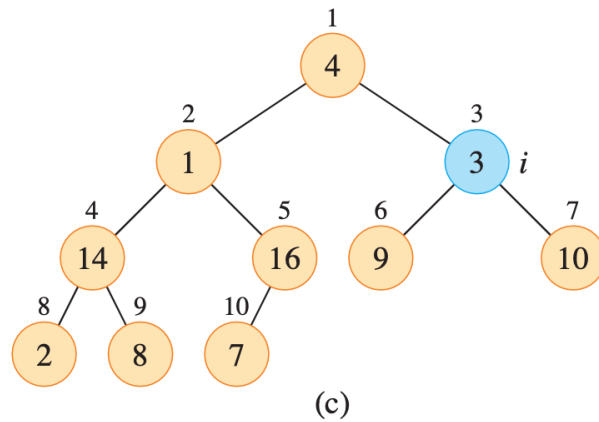
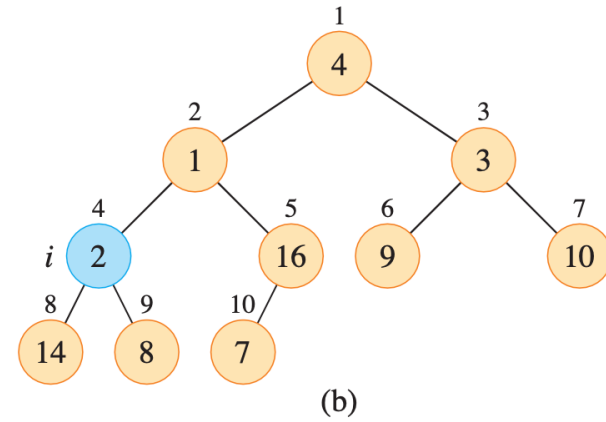
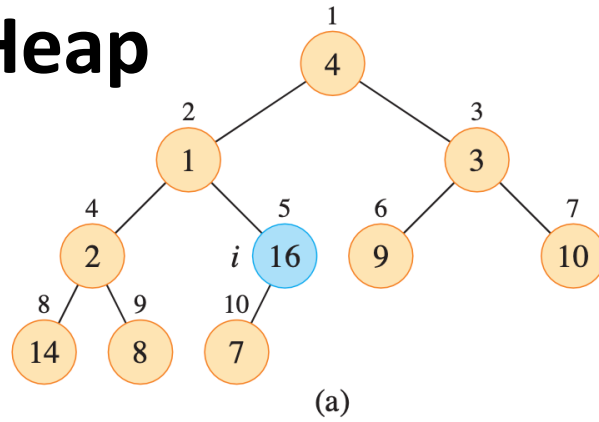
- Idea: use Max-Heapify repeatedly to create a heap.
- Which order of nodes: top-down or bottom-up?
- Answer: **bottom-up** – Max-Heapify assumes  $\text{Left}(i)$  and  $\text{Right}(i)$  are heaps. Top-down wouldn't work, bottom-up does.
- Note: nodes in  $A \left[ \left( \left\lfloor \frac{n}{2} \right\rfloor + 1 \right), \dots, n \right]$  are all leaves. Leaves are max-heaps, so no work required.

```
BUILD-MAX-HEAP( $A, n$ )  
1   $A.\text{heap-size} = n$   
2  for  $i = \lfloor n/2 \rfloor$  downto 1  
3      MAX-HEAPIFY( $A, i$ )
```

A 

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

## ➤ Build-Max-Heap

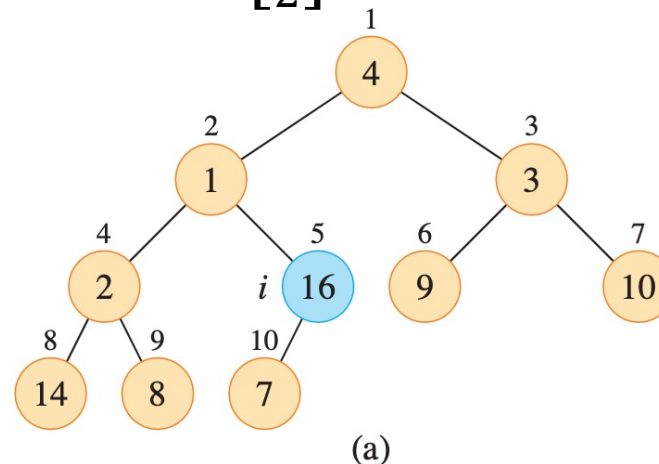


## ➤ Correctness of Build-Max-Heap

BUILD-MAX-HEAP( $A, n$ )

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

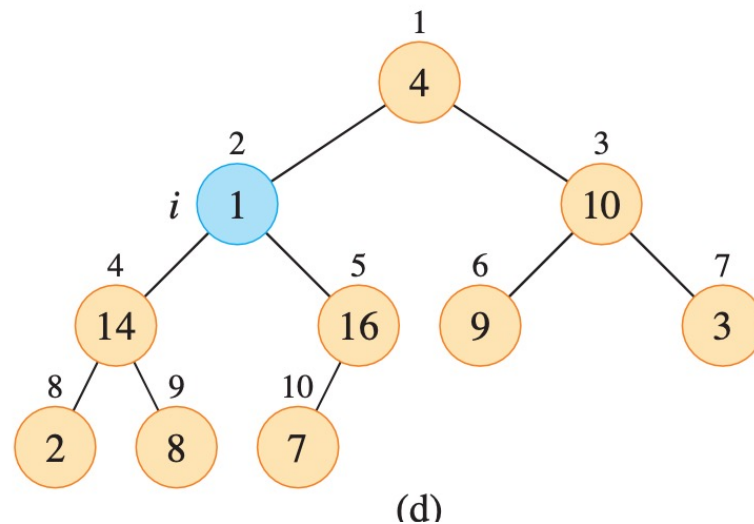
- **Loop invariant:** At the start of each iteration  $i$  of the for loop, each node  $i + 1, i + 2, \dots, n$  is the root of a max-heap.
- **Initialisation:** true for leaves  $\lfloor \frac{n}{2} \rfloor + 1, \dots, n$ .



## ➤ Correctness of Build-Max-Heap

```
BUILD-MAX-HEAP( $A, n$ )  
1   $A.heap-size = n$   
2  for  $i = \lfloor n/2 \rfloor$  downto 1  
3      MAX-HEAPIFY( $A, i$ )
```

- **Loop invariant:** At the start of each iteration  $i$  of the for loop, each node  $i + 1, i + 2, \dots, n$  is the root of a max-heap.
- **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.

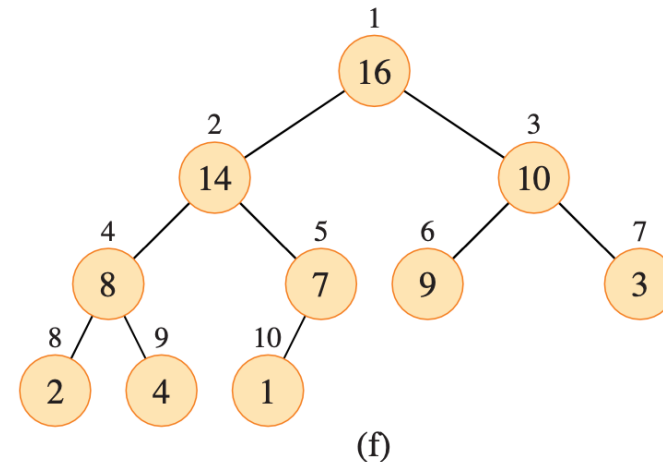
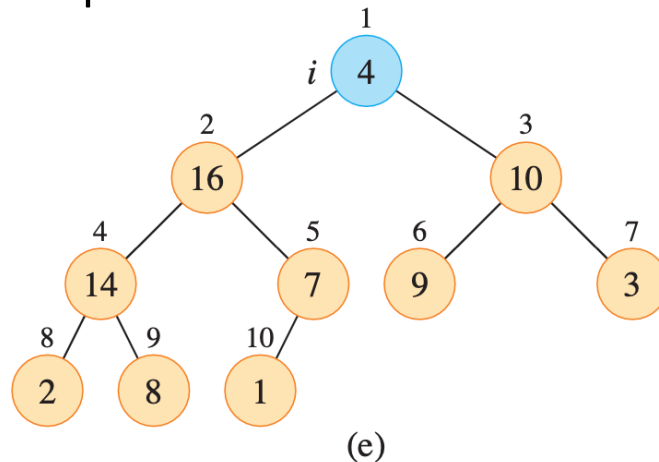


## ➤ Correctness of Build-Max-Heap

BUILD-MAX-HEAP( $A, n$ )

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

- **Loop invariant:** At the start of each iteration  $i$  of the for loop, each node  $i + 1, i + 2, \dots, n$  is the root of a max-heap.
- **Termination:** the loop terminates at  $i = 0$ , hence node 1 is the root of a max-heap.



## ➤ Runtime of Build-Max-Heap

- The **height of a heap** = height of the root is at most  $\log n$ .
- So all nodes have height at most  $\log n$ .
- 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.
  - $O(n \log n)$  is sufficient for us, though.

# ➤ Refined Analysis of Build-Max-Heap

- **Observation: most nodes have small height**

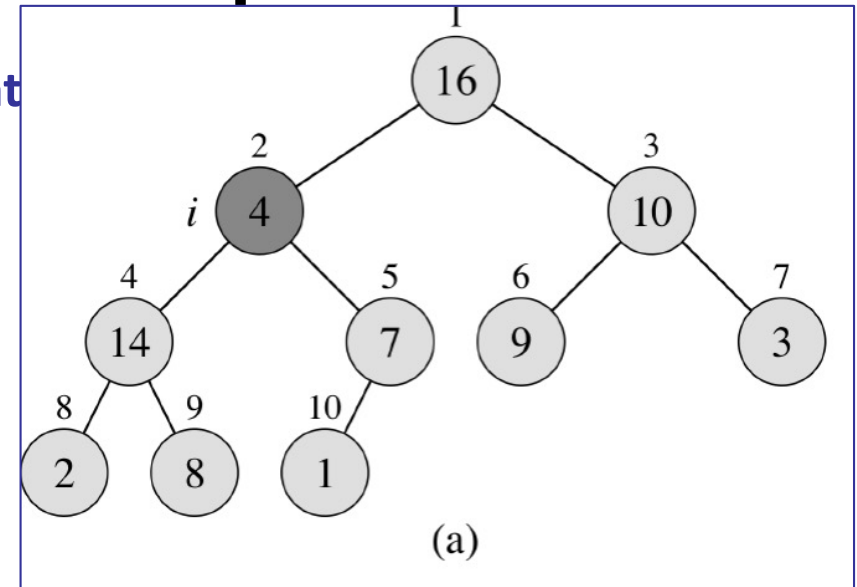
- One can show: there are at most  $\left\lceil \frac{n}{2^{h+1}} \right\rceil$  nodes of height  $h$ .
- $O(\log n)$  time bound is correct, but crude for most nodes.

- A better bound:

$$\sum_{h=1}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O \left( n \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O \left( n \sum_{h=1}^{\infty} \frac{h}{2^h} \right) = O(n)$$

as the infinite series of  $\frac{h}{2^h}$  is 2.

- 1<sup>st</sup> equality, we used that:  $\lceil x \rceil \leq 2x$  for  $x \geq 1/2$   
 $\Rightarrow$  for  $h \leq \log n$ ,  $\frac{n}{2^{h+1}} \geq 1/2$  because  $n \geq 2^h$  (see slide 13)
- 2<sup>nd</sup> equality, we used that  $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$  for  $|x| < 1$



## ➤ Procedures (what do we need)

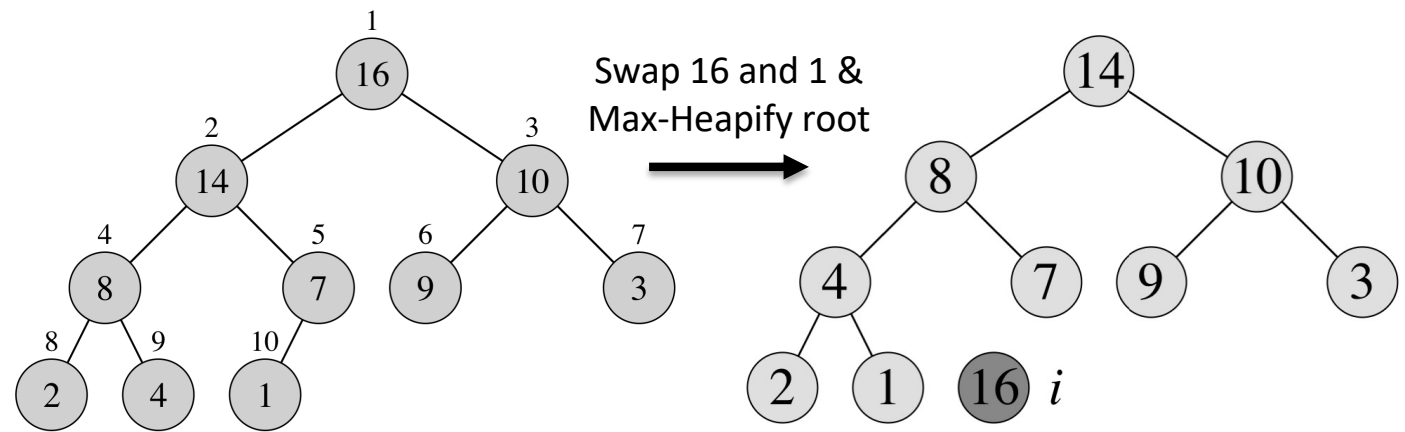
1. **Build-Max-Heap:** produces a Max-Heap from an unordered array ✓
2. **Max-Heapify:** maintains the max-heap property once the maximum has been removed ✓
3. **HeapSort:** sorts an array in place



## ➤ 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.




---

$\text{HEAPSORT}(A)$

---

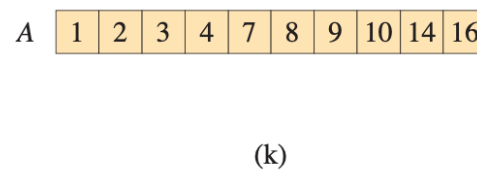
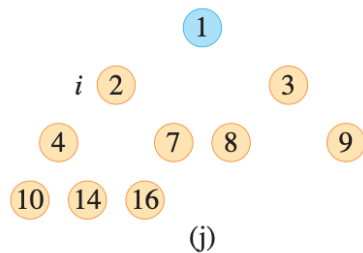
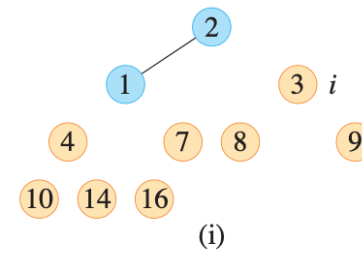
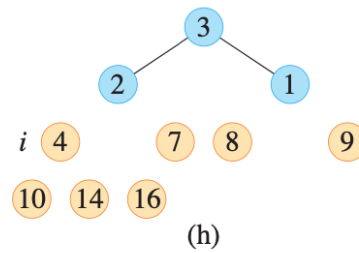
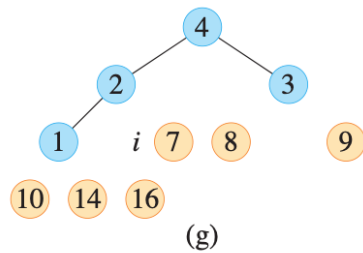
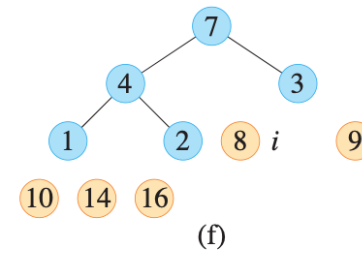
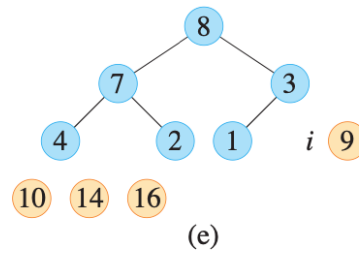
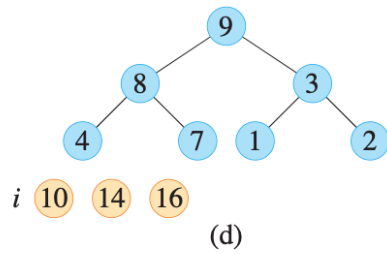
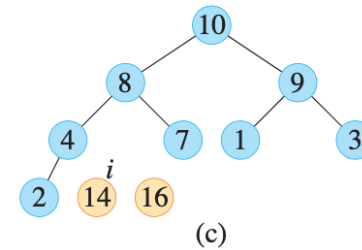
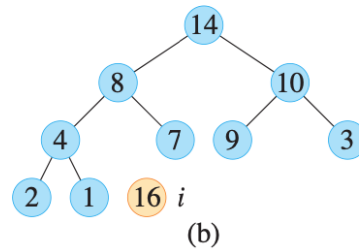
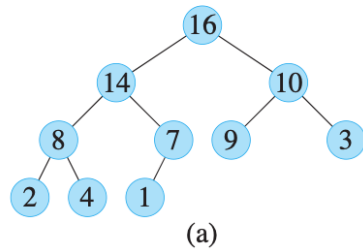
```

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:      $\text{MAX-HEAPIFY}(A, 1)$ 

```

---

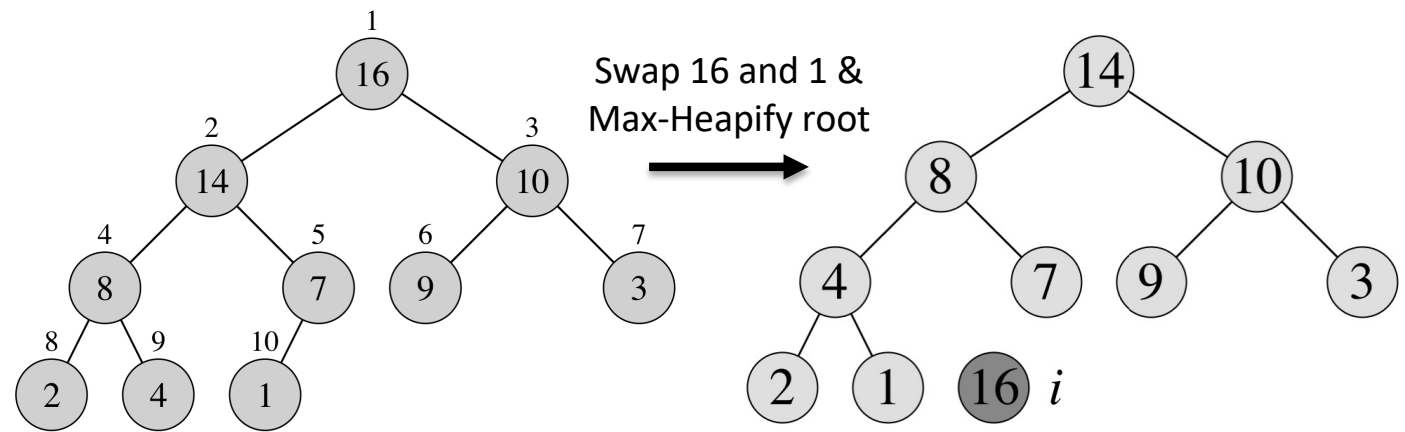
# ➤ HeapSort: Example



## ➤ 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.




---

$\text{HEAPSORT}(A)$

---

```

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( $A, 1$ )

```

---

**Runtime:**

$$\begin{aligned}
 &O(n \log n) \\
 &+ (n - 1) \cdot O(\log n) \\
 &= O(n \log n)
 \end{aligned}$$

## ➤ Correctness of HeapSort

**Loop Invariant:** “At the start of each iteration of the for loop of lines 2-5, the subarray  $A[1..i]$  is a max-heap containing the  $i$  smallest elements of  $A[1..n]$ , and the subarray  $A[i+1..n]$  contains the  $n-i$  largest elements of  $A[1..n]$ , sorted.”

- **Initialization:** The subarray  $A[i+1..n]$  is empty, thus the invariant holds.

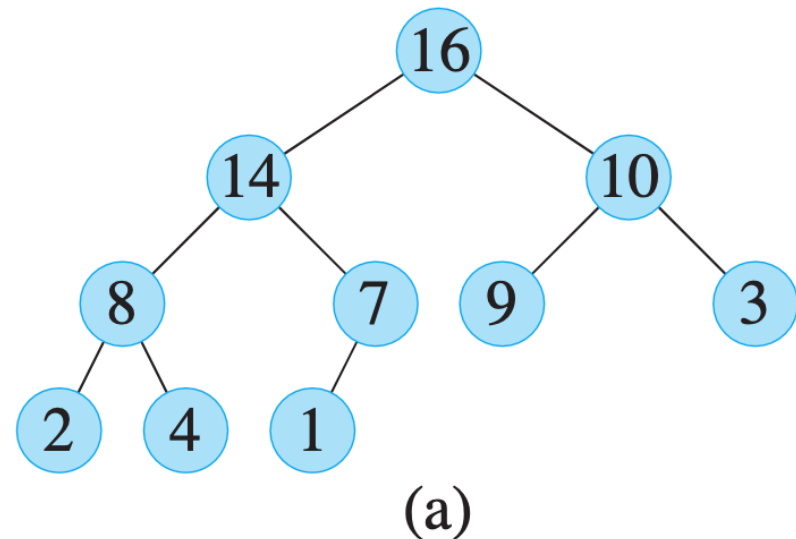
---

HEAPSORT( $A$ )

---

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

---



## ➤ Correctness of HeapSort

**Loop Invariant:** “At the start of each iteration of the for loop of lines 2-5, the subarray  $A[1..i]$  is a max-heap containing the  $i$  smallest elements of  $A[1..n]$ , and the subarray  $A[i+1..n]$  contains the  $n-i$  largest elements of  $A[1..n]$ , sorted.”

**Maintenance:**  $A[1]$  is the largest element in  $A[1..i]$  and it is smaller than the elements in  $A[i+1..n]$ . When we put it in the  $i$ th position, then  $A[i..n]$  contains the largest elements, sorted. Decreasing the heap size and calling Max-Heapify turns  $A[1..i-1]$  into a max-heap. Decrementing  $i$  sets up the invariant for the next iteration.

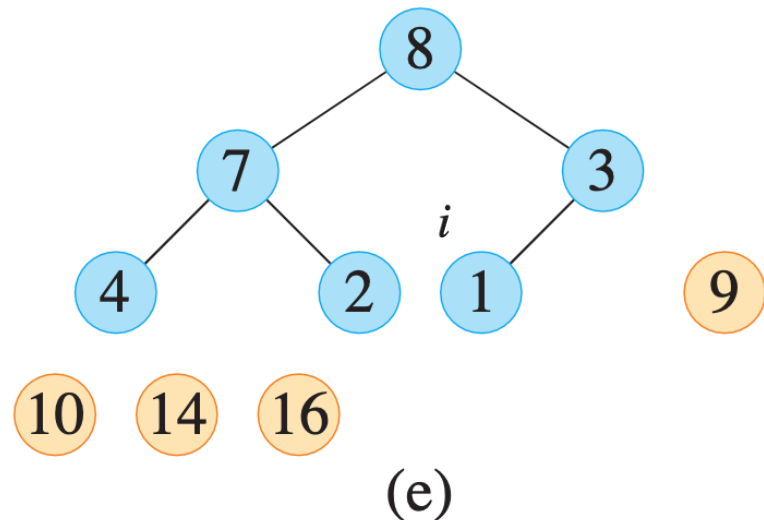
---

HEAPSORT( $A$ )

---

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

---



## ➤ Correctness of HeapSort

**Loop Invariant:** “At the start of each iteration of the for loop of lines 2-5, the subarray  $A[1..i]$  is a max-heap containing the  $i$  smallest elements of  $A[1..n]$ , and the subarray  $A[i+1..n]$  contains the  $n-i$  largest elements of  $A[1..n]$ , sorted.”

- **Termination:** After the loop  $i=1$ . This means that  $A[2..n]$  is sorted and  $A[1]$  is the smallest element in the array, which makes the array sorted.

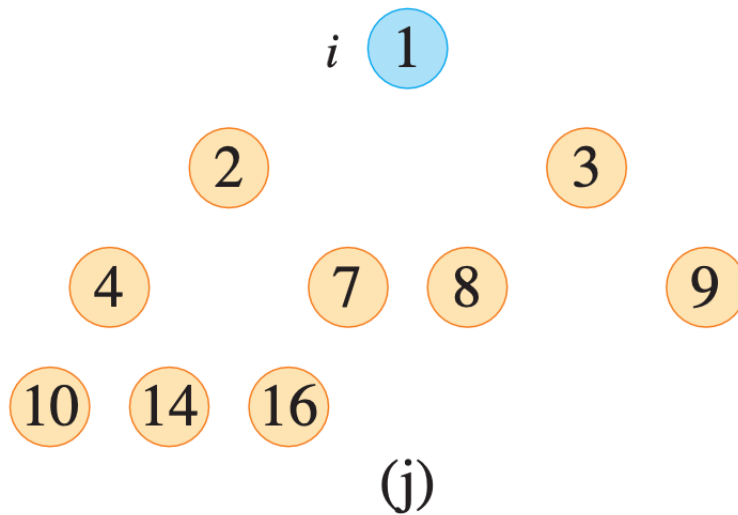
---

HEAPSORT( $A$ )

---

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

---



## ➤ 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!
- Heaps also play a role in **Priority Queues**.