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…

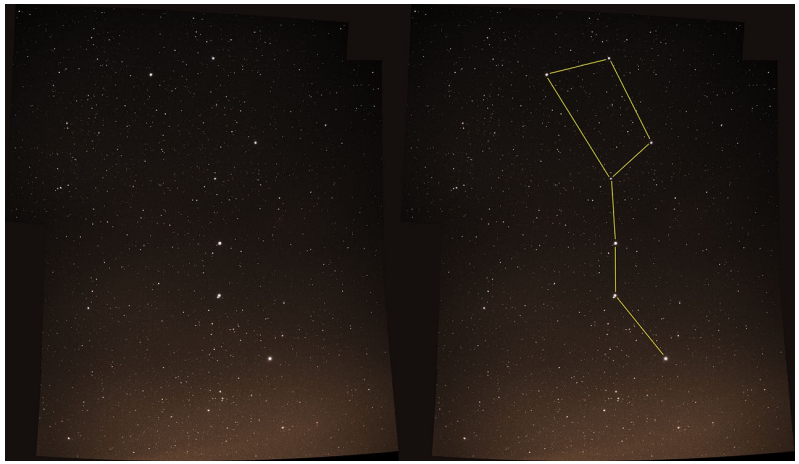
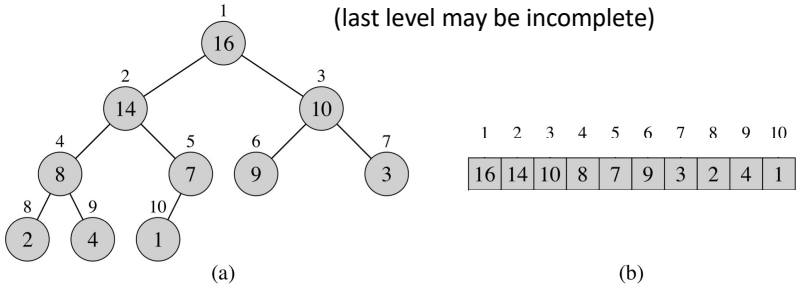


Photo : Thomas Bresson

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)

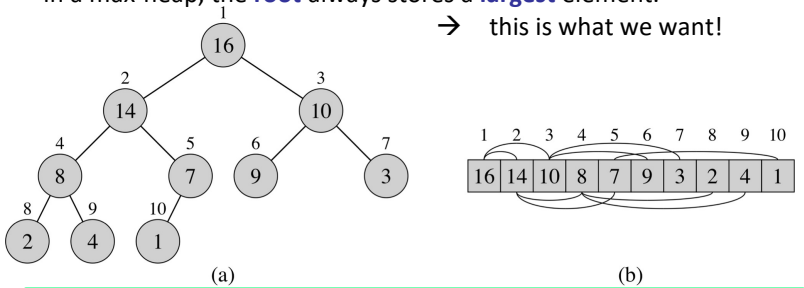


- Navigate through the array/imaginary tree using these operations:
- Parent$(i) = \lfloor \frac{i}{2} \rfloor$ ("floor of *i*/2"), Left$(i) = 2i$, Right$(i) = 2i + 1$

数组的索引顺序就是节点的“排队”顺序。一个节点的子节点，只有在它所有“前辈”节点的子节点都排完队之后，才能轮到它们自己排队。第i个节点的子节点：前面有i–1个节点作为父节点，生成2i–2个子节点，加上最前面的第一个根节点，一共2i–1个节点，后面就是第2i和2i+1个。

Heap Properties

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



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

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.heap-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.

Procedures (what do we need)

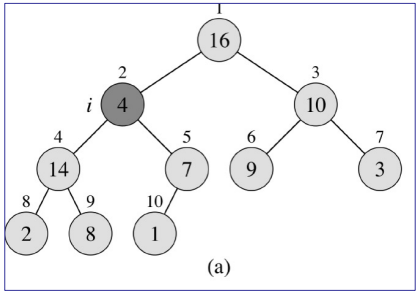
1. **Build-Max-Heap:** produces a Max-Heap from an unordered array
2. **Max-Heapify**: maintains the max-heap property once the maximum has been removed
3. **HeapSort:** sorts an array in place

- New variable A.heap-size indicates how many elements of A are stored in a heap: $0 \leq$ A.heap-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.

注意这边，Max-Heapify是一个用来递归构造一个Max-Heap的方法，也就是from bottor to up建立。上面的1、2、3步并不是步骤，2 Max-Heapify也被包含在1中的build过程中！

Max-Heapify(*A, i*)

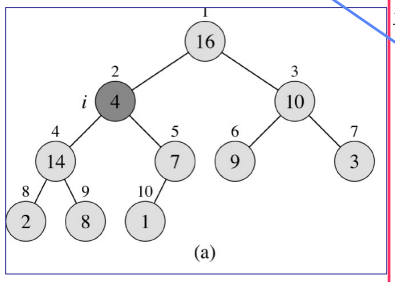
- Assumes subtrees Left(*i*) and Right(*i*) are max-heaps, but max-heap property might be violated in root of subtree at *i*.
  - "Subtree x": the part of the tree including x and everything below.
- Lets the value at *A[i]* "float down" if necessary, to restore max-heap property at *i*
- At the end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



1. i是一个叶子节点：l=2i会超过堆的边界，也就是没有子节点。
2. 在堆排序过程中，堆的大小动态缩小时，比如l = LEFT(5) = 10 ⩽ 9，会显示False，这时候子节点是上一回合中最大的那个数，已不在当前回合的范围。

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



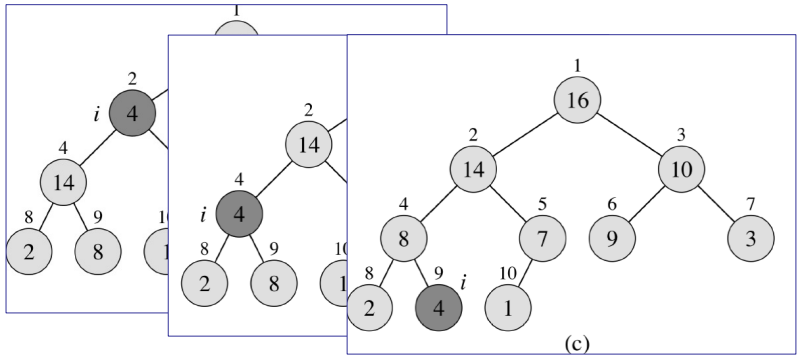
MAX-HEAPIFY(A, i)

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


现在的largest这个位置还是前面的那个位置，现在这个位置上的数字变成了之前i上的数字。我们对这个子树再重复验证它的root是不是还有上面的violation。

Max-Heapify: Example

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



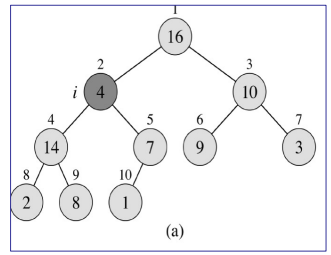
注意：h实际上是与n有关的数，所以并不能O(h)=O(1)。

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 = Left(i)
2:  r = Right(i)
3:  if l ≤ A.heap-size and A[l] > A[i] then
4:      largest = l
5:  else
6:      largest = i
7:  if r ≤ A.heap-size and A[r] > A[largest] then
8:      largest = r
9:  if largest ≠ i then
10:     exchange A[i] with A[largest]
11:     MAX-HEAPIFY(A, largest)
```

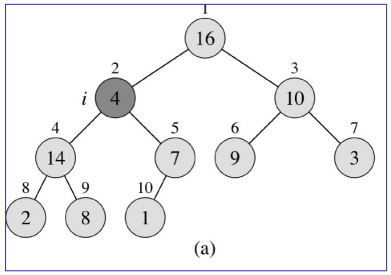


Bounding the height of a heap

- **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

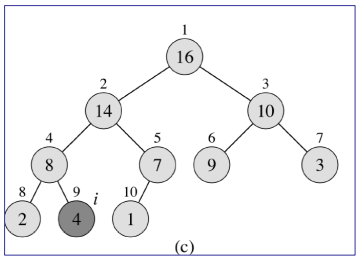
$$1 + 2 + 4 + \cdots + 2^{h-1} + 1 = 2^h$$

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



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

Max-Heapify: Correctness

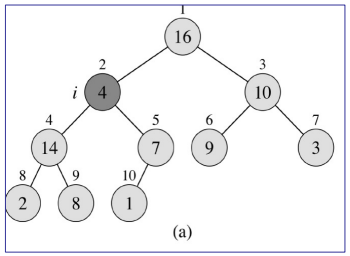


MAX-HEAPIFY(A, i)

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

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

Max-Heapify: Correctness



MAX-HEAPIFY(A, i)

```
1: l = Left(i)
2: r = Right(i)
3: if l ≤ A.heap-size and A[l] > A[i] then
4:       largest = l
5: else
6:       largest = i
7: if r ≤ A.heap-size and A[r] > A[largest] then
8:       largest = r
9: if largest ≠ i then
10:      exchange A[i] with A[largest]
11:      MAX-HEAPIFY(A, 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 Left(i) and Right(i) (if any) and one subtree was already a heap and the other will be by inductive hypothesis.

Procedures (what do we need)

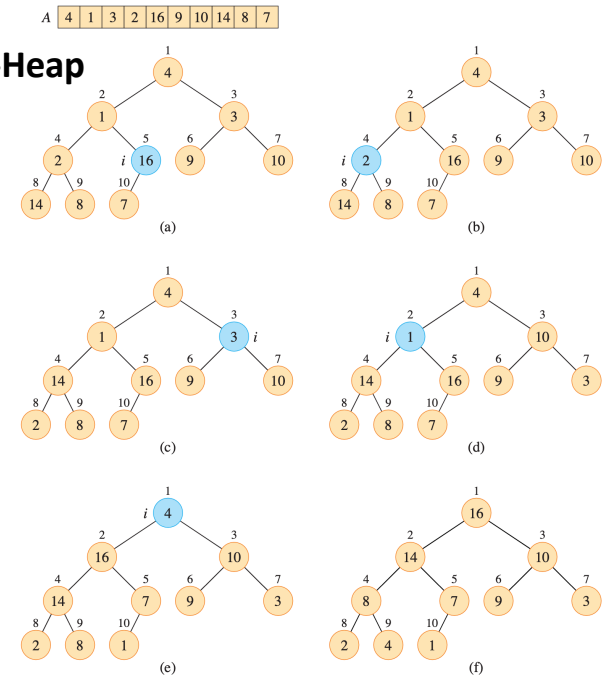
1. **Build-Max-Heap:** produces a Max-Heap from an unordered array
2. **Max-Heapify**: maintains the max-heap property once the maximum has been removed ✓
3. **HeapSort:** sorts an array in place

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 Left(i) and Right(i) are heaps. Top-down wouldn't work, bottom-up does.
- Note: nodes in $A\left[\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right), \ldots, n\right]$ are all leaves. Leaves are max-heaps, so no work required.

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

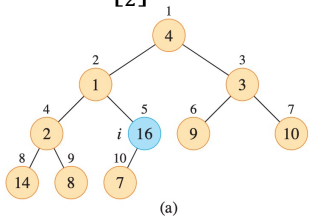

Build-Max-Heap



Correctness of Build-Max-Heap

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

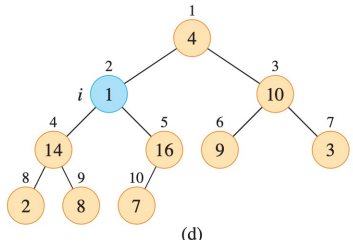
- **Loop invariant:** At the start of each iteration i of the for loop, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.
- **Initialisation:** true for leaves $\left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n$.



Correctness of Build-Max-Heap

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

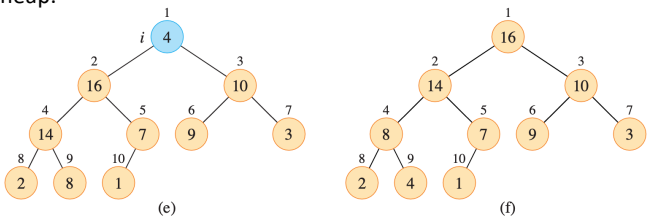
- **Loop invariant:** At the start of each iteration i of the for loop, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.
- **Maintenance:** by loop invariant, all children of i are roots of max-heaps (as their numbers are larger than i).
  Then Max-Heapify(A, i) turns the subtree at i into a max-heap.



Correctness of Build-Max-Heap

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

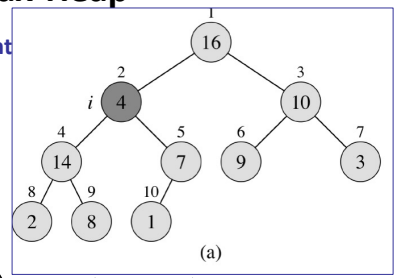
- The **height of a heap** = height of the root is at most *log n*.
- So all nodes have height at most *log n*.
- Every call to Max-Heapify takes time $O(\log n)$.
- Build-Max-Heap calls Max-Heapify $O(n)$ times.
- Total time is at most $O(n) \cdot O(\log n) = O(n \log n)$.
  - The time can be improved 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.



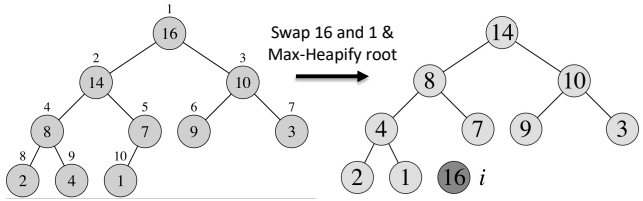
- 1st equality, we used that: $\lceil x \rceil \leq 2x$ for $x \geq 1/2$

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

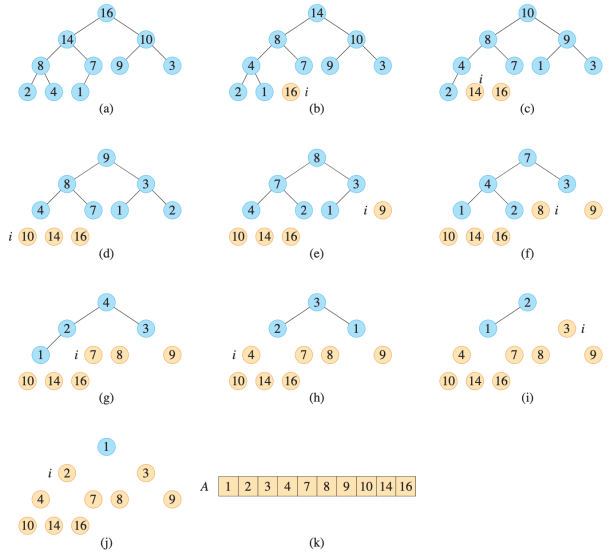
HeapSort



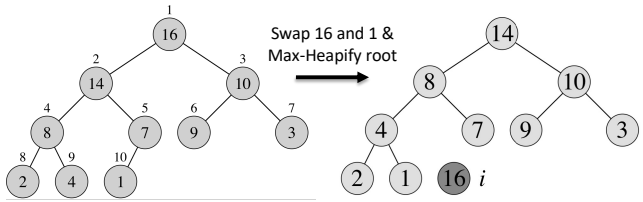
- Ideas:
  1. Build a max-heap, such that the root contains largest element.
  2. Swap the root with the last element of the heap/array.
  3. Discard the last element from the heap by reducing heap.size. (We simply imagine a smaller heap.)
  4. Call Max-Heapify(*A, 1*) to restore heap property at the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

```
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)
```


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

```
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)
```

Runtime:

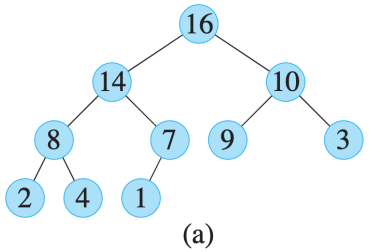
$$O(n \log n) + (n-1) \cdot O(\log n) = O(n \log n)$$

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-size = A.heap-size −1
5:     Max-Heapify(A, 1)
```

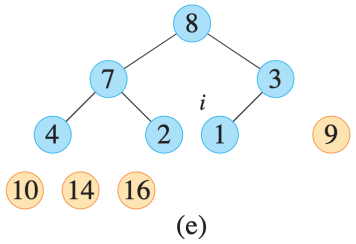


Correctness of HeapSort

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."

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 ith 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)
```



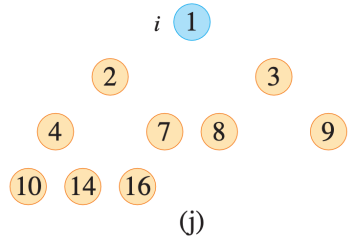
(e)

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)
```



(j)

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**.

关键区别：重建 (Rebuild) vs. 修复 (Repair)
让我们来对比一下这两种思路：

1. 浪费时间的思路 (重建整个堆)
我们有一个 n 个元素的最大堆。取出最大值 A[1]。
现在我们剩下 n−1 个元素，它们是 A[2], A[3], ..., A[n]。
(浪费时间的步骤) 把这 n-1 个元素当成一个全新的、无序的数组，然后调用 Build-Max-Heap 算法，花费 O(n) 的时间重新把它们组织成一个堆。
重复这个过程...
如果按照这个思路，每次取出一个元素都要花费 O(n) 的时间来重建堆，总时间复杂度会是 O(n^2)，这和低效的选择排序、冒泡排序就没什么区别了。

2. 高效的思路 (Heapsort的实际做法：仅修复)
Heapsort的实际做法是：在取出最大值后，堆的结构几乎是完好的，只有一个地方出了问题。我们只需要修复那个小问题就行了。
发生了什么？交换：我们将堆顶 A[1] (最大值) 和堆的最后一个元素 A[heap.size] 交换。缩小：heap.size−−。
现在的局面是什么？根节点 (Root): A[1] 现在存放的是原来那个很小的叶子节点的值。这导致最大堆的性质在根部被破坏了。A[1] 很可能比它的子节点 A[2] 和 A[3] 要小。其他部分 (The Rest of the Tree): 根节点的两个子树（以 A[2] 和 A[3] 为根的子树）本身仍然是完美的最大堆！ 因为我们根本没有动它们内部的结构。
我们要做的工作：我们不需要重建一切。我们只需要把那个被安插在CEO位置上的“实习生”（原来那个小的值）一路“下沉”或“筛选”到它合适的位置就行了。这个操作就是 Max−Heapify。

Max-Heapify(A, 1) 的工作效率：
它从根节点 i=1 开始。
它将 A[1] 的值和它的子节点比较，然后和较大的那个子节点交换。
现在，那个小的值下沉了一层。它可能又破坏了下一层的堆性质。
所以 Max−Heapify 会递归地沿着一条路径继续往下走，直到这个元素不再小于它的子节点，或者它自己变成了叶子节点。

这个过程有多快?
这个“下沉”的路径最长能有多长？就是这个堆的高度。
一个包含 k 个元素的完全二叉树，它的高度大约是 $\log_2(k)$。
所以，每一次“修复”操作的时间复杂度仅仅是 O(log n)。