

CH08-320201

Algorithms and Data Structures

ADS

Lecture 7

Dr. Kinga Lipskoch

Spring 2019

Max-Heapify(A, i) (5)

MAX-HEAPIFY(A, i)

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

Max-Heapify(A, i) (6)

Time complexity:

$$T(n) = O(h) = O(\lg n),$$

as in the worst case the element from position i has to go down all the way to the last level.

In the expression above h is the height of the element from position i .

Converting an Existing Array A to a Max-Heap (1)

Call MAX-HEAPIFY on which set of nodes?

- ▶ All inner nodes \Rightarrow move them down if necessary (*Max-Heapify*)
- ▶ Not necessary to call on leaf nodes

Leaves of a Heap of Size n

Let $n = A.\text{heapsize}$.

Where is the parent of the last element of a heap?

- ▶ At index $n/2$.

Therefore, the element at index $n/2 + 1$ does not have a child in the heap, and hence is a leaf.

In a heap, there are $n/2$ leaves:

- ▶ from index $n/2 + 1$ to n

Each leaf is the root of a valid max-heap of size 1.

Converting an Existing Array A to a Max-Heap (2)

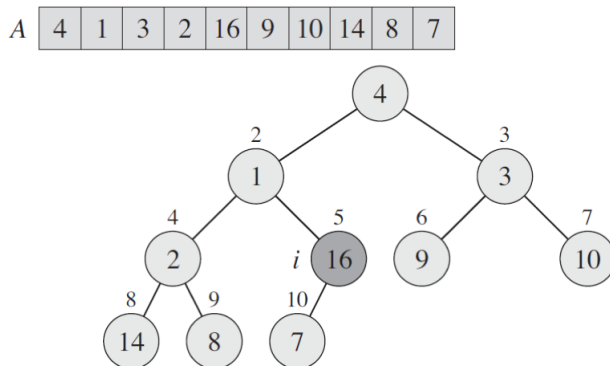
BUILD-MAX-HEAP(A)

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

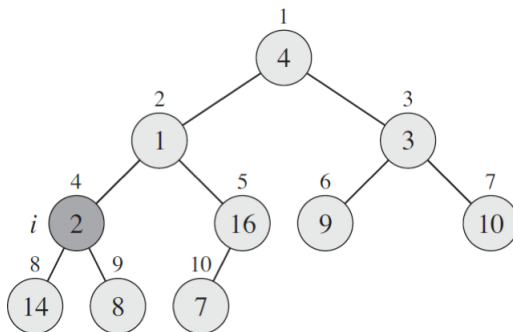
Loop invariant:

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

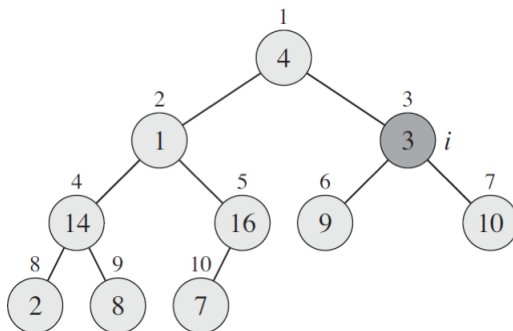
Build Max-Heap (2)



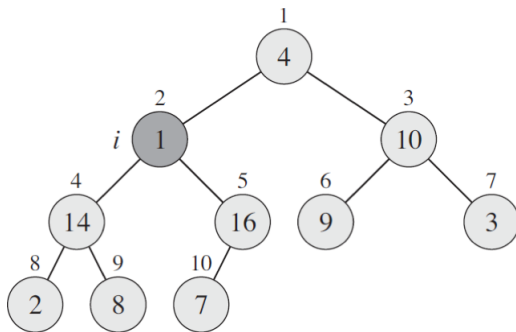
Build Max-Heap (3)



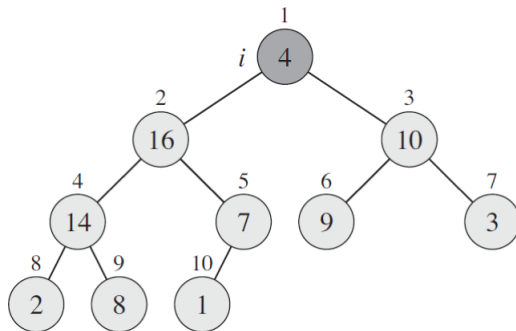
Build Max-Heap (4)



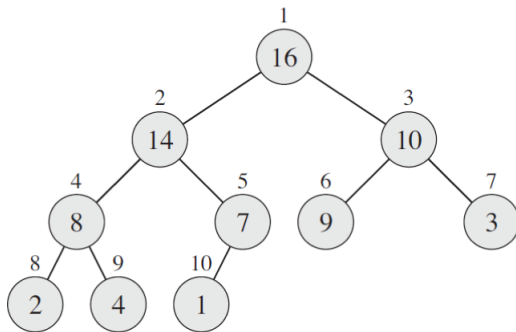
Build Max-Heap (5)



Build Max-Heap (6)



Build Max-Heap (7)



Build Max-Heap (8)

What is the time complexity of the algorithm?

Theorem:

Let m_h be the number of nodes of height h in any n element heap $A(n)$.

Then $m_h(A, n) \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$.

(Proof by induction over h)

Build Max-Heap (9)

Time complexity:

The time needed by *Max-Heapify* when called on a node of height h is $O(h)$. Therefore, the total cost of *Build-Max-Heap*(A) is upper bounded by

$$\begin{aligned} \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) &= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) \\ &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n). \end{aligned}$$

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} \text{ if } |x| < 1.$$

Conclusion:

We can convert an unordered array into a max-heap in linear time.

Heap Sort

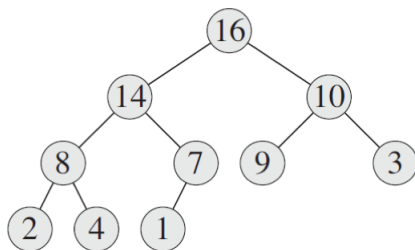
- ▶ Start by generating a max-heap.
- ▶ The maximum element of a max-heap is at the root.
- ▶ Put it in its right sorted place at $n = A.\text{heapsize}$ by swapping it with the last element $A[n]$, which now becomes $A[1]$.
- ▶ Decrement the heap size to create a smaller heap and thus implicitly remove the last element (the maximum) from the heap.
- ▶ The new $A[1]$ may not satisfy the max-heap property, so move it down.
- ▶ Iterate.

Heap Sort: Pseudocode

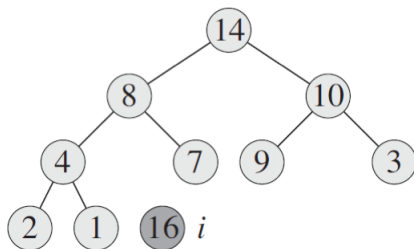
HEAPSORT(A)

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

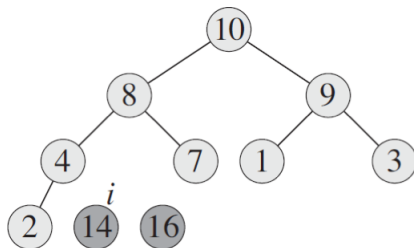

Heap Sort: Example (1)



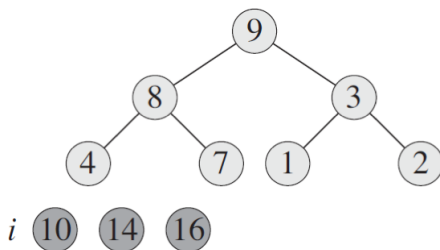
Heap Sort: Example (2)



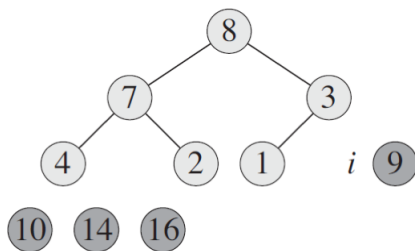
Heap Sort: Example (3)



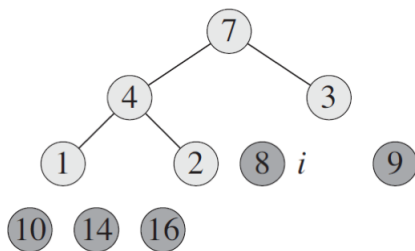
Heap Sort: Example (4)



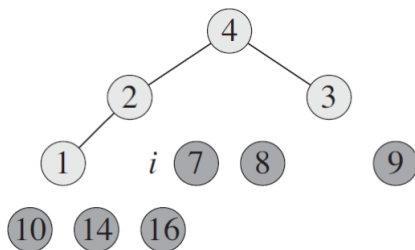
Heap Sort: Example (5)



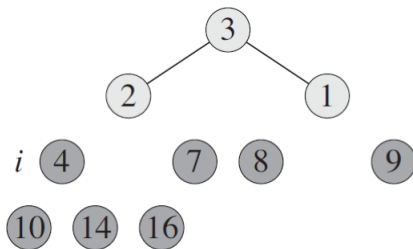
Heap Sort: Example (6)



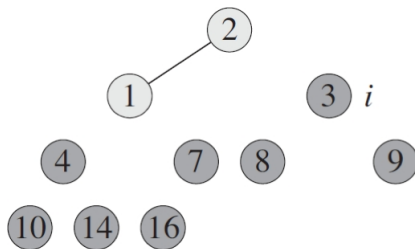
Heap Sort: Example (7)



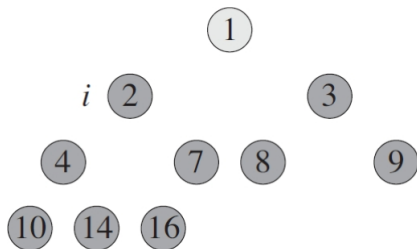
Heap Sort: Example (8)



Heap Sort: Example (9)



Heap Sort: Example (10)



Heap Sort: Runtime Analysis

HEAPSORT(A)

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

- ▶ Runtime costs:

$$O(n) + O(n \lg n) = O(n \lg n)$$

- ▶ Memory costs:

$O(1)$, i.e., in-situ sorting

- ▶ Visualization:

<http://www.sorting-algorithms.com/heap-sort>