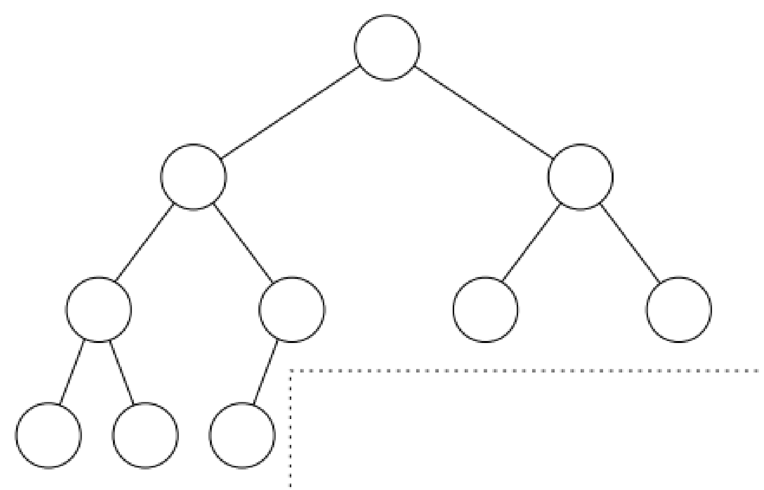
Heapsort

Introduction

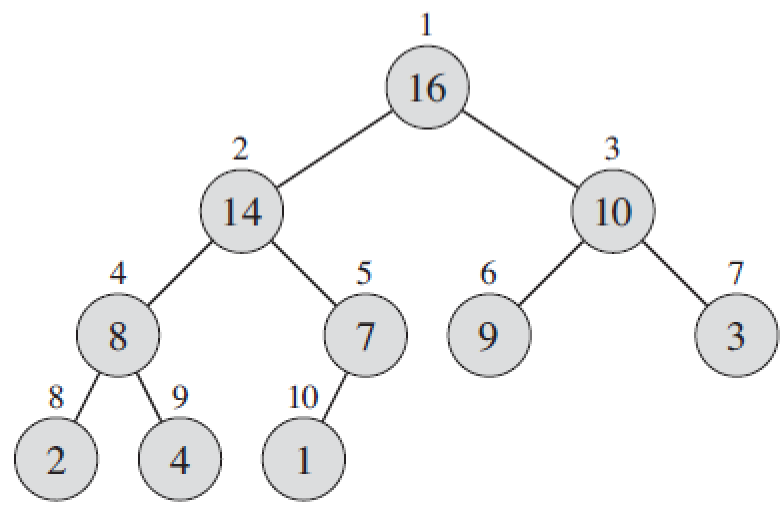
- Like insertion sort heapsort sorts in place.
- It introduces another algorithm design technique: using a data structure called “heap” to manage information.
- The term “heap” was originally coined in the context of heapsort, but it has since come to refer to “garbage-collected storage”.

Heaps

- A heap is represented as an left-complete binary tree.
  - all the levels of the tree are full except the bottommost level, which is filled from left to right.



Left−complete Binary Tree



Heaps

- An array A that represents a heap is an object with two attributes:
  - A.*length*, which gives the number of elements in the array.
  - A.*heap-size*, which represents how many elements in the heap are stored within array A.

  where $0 \leq A.\mathit{heap\text{-}size} \leq A.\mathit{length}$

Heap as a Tree

- root of tree: first element in the array, corresponding to i = 1
- parent(i) =i/2: returns index of node's parent
- left(i)=2i: returns index of node's left child
- right(i)=2i+1: returns index of node's right child

Height of a binary heap is O(lg n)

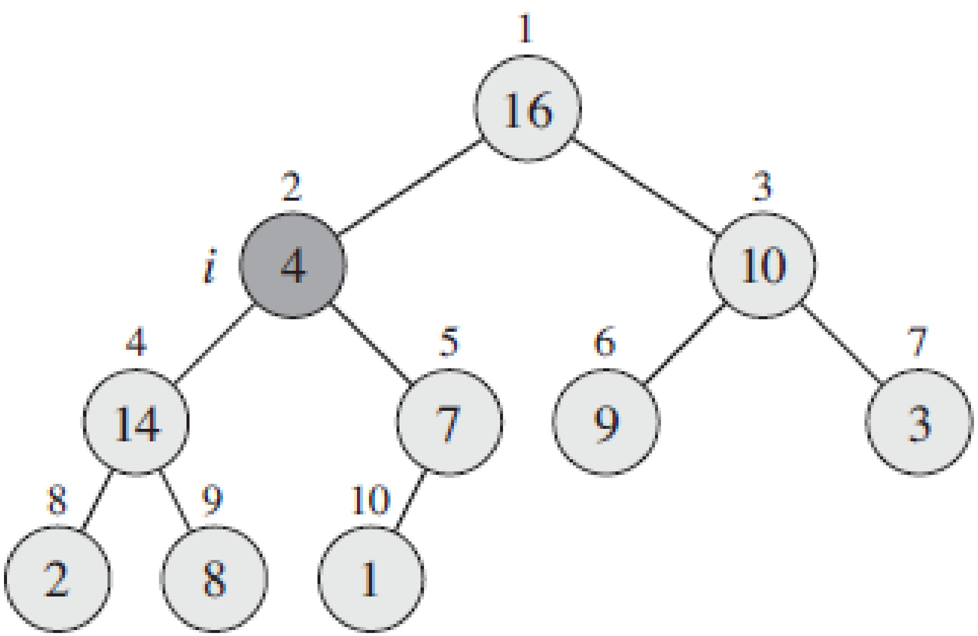
Heap Operations

- MAX-HEAPIFY : correct a single violation of the heap property in a subtree at its root.
- BUILD-MAX-HEAP : produces a maxheap from an unordered input array.
- HEAPSORT : sorts an array in place.
- MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY and HEAP-MAXIMUM

Max-Heapify

- Assume that the trees rooted at left(i) and right(i) are max-heaps.
- If element A[i] violates the max-heap property, correct violation by “trickling” element A[i] down the tree, making the subtree rooted at index *i* a max-heap

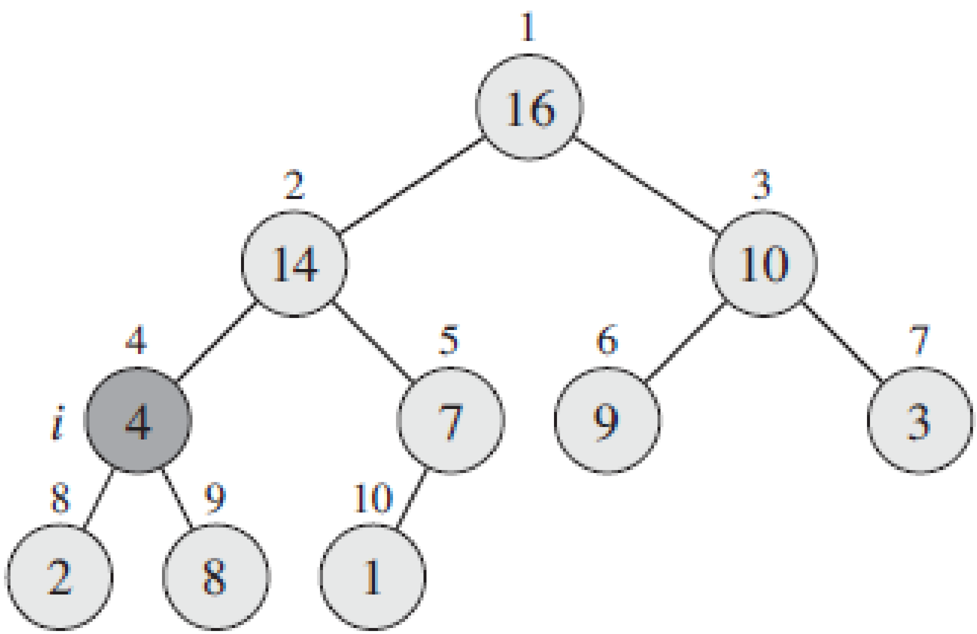
Max-Heapify (Example)



MAX-HEAPIFY(A, 2)

A.*heap_size = 10*

Max-Heapify (Example)

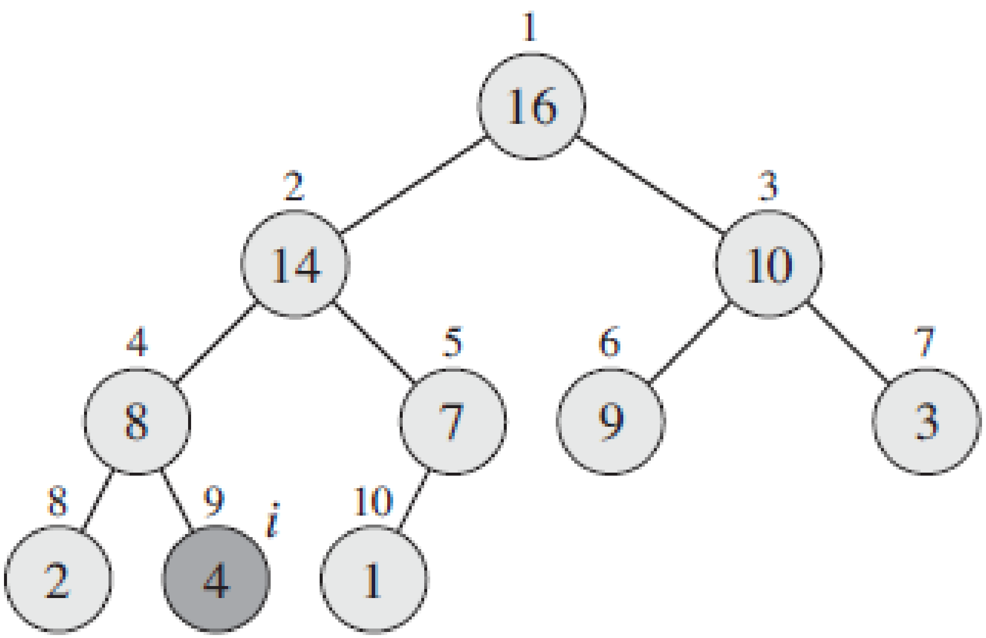


Exchange A[2] with A[4]

Call MAX-HEAPIFY(A, 4)

Because max_heap property is violated

Max-Heapify (Example)



Exchange A[4] with A[9]

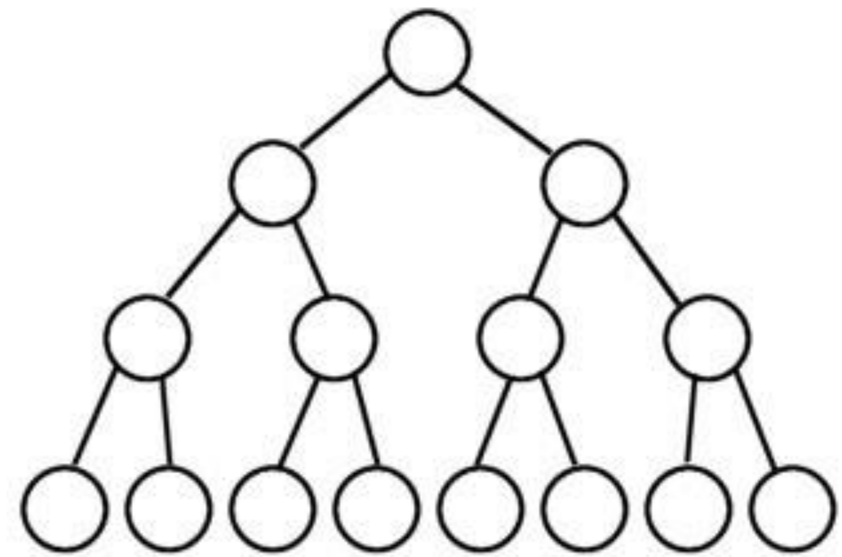
No more calls

Time = ?   O(logn)

Max-Heapify Pseudocode

l = left(*i*)
r = right(*i*)
if (*l* <= A.heap-size and A[*l*] > A[*i*])
        largest = *l*
else largest = *i*
if (*r* <= A.heap-size and A[*r*] > A[largest])
        largest = *r*
if largest ≠ *i*
        exchange A[*i*] and A[largest]
Max_Heapify(A, largest)

Build-Max-Heap



- Converts A[1…*n*] to a max heap

Build_Max_Heap(A):

A.*heap_size* = A.*length*

for i=A.*length*/2 **downto** 1

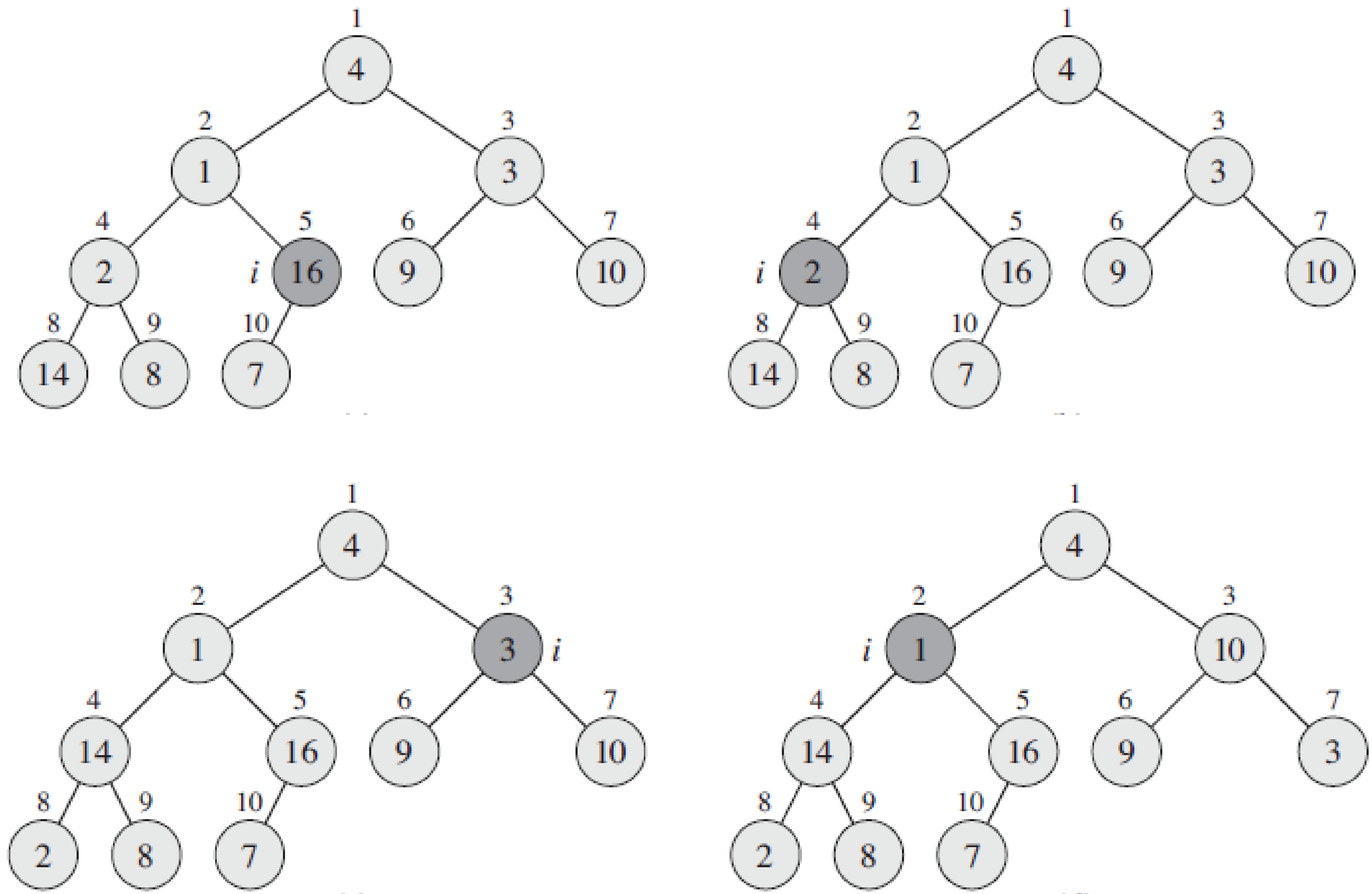
    Max-Heapify(A, i)

Why start at A.*length*/2?

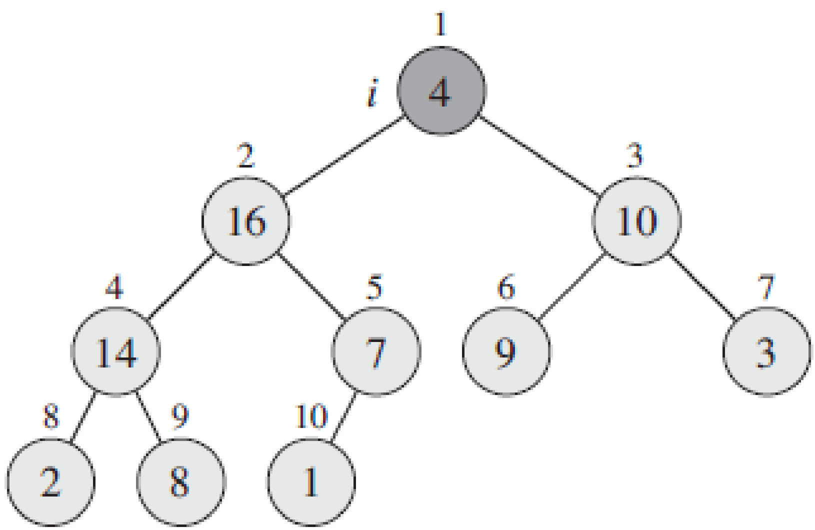
Because elements A[n/2 + 1 … n] are all leaves of the tree 2i > n, for i > n/2 + 1

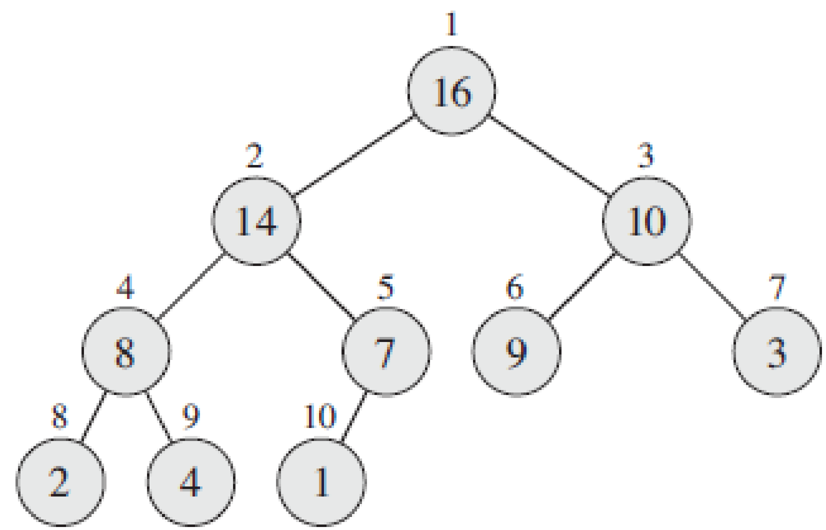
Time=? O(nlogn) via simple analysis

Build-Max-Heap



Build-Max-Heap





Build-Max-Heap Analysis

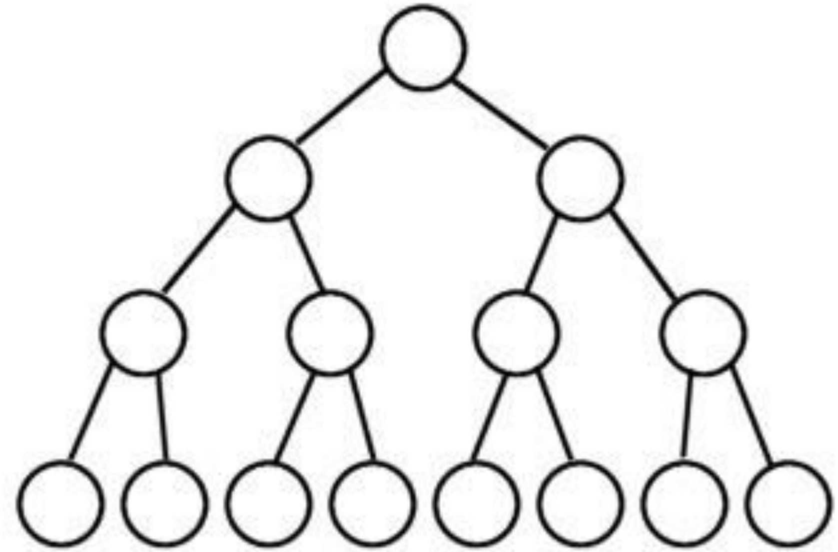
- Converts A[1…*n*] to a max heap

Build_Max_Heap(A):

A.*heap_size* = A.*length*

for i=A.*length*/2 **downto** 1

    Max-Heapify(A, i)



Observe however that Max_Heapify takes O(1) for time for nodes that are one level above the leaves, and in general, O(h) for the nodes that are h levels above the leaves. We have:

- n/4 nodes with level 1
- n/8 with level 2 and so on

That is at most $\lceil n/2^{h+1} \rceil$ nodes of any height h

Build-Max-Heap Analysis

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

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

$$O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(\frac{n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

Therefore the running time of Build-Max-Heap can be bound as O(n).

Heapsort

Sorting Strategy:

- Build Max Heap from unordered array;
- Find maximum element A[1];
- Swap elements A[*n*] and A[1]:

  now max element is at the end of the array!
- Discard node *n* from heap (by decrementing heap_size variable)
- New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.
- Go to Step 2 unless heap is empty.

Pseudocode

BUILD-MAX-HEAP(A)

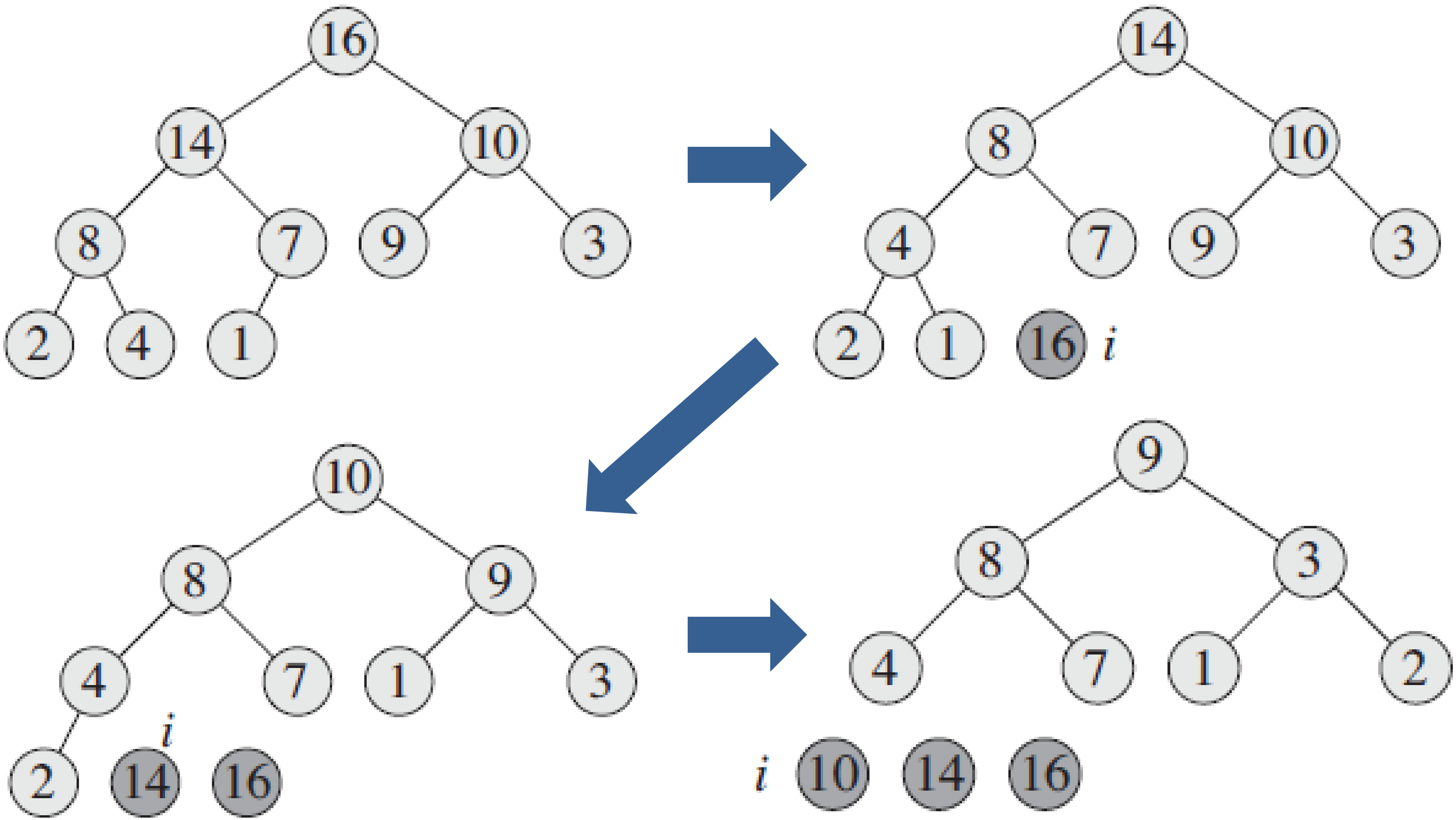
for i = A.*length* **downto** 2

    exchange A[1] with A[i]

    A.*heap_size* = A.*heap_size* – 1

    MAX-HEAPIFY(A, 1)

Heapsort Example



HeapSort Analysis

Running time:

after n iterations the Heap is empty; every iteration involves a swap and a max_heapify operation; hence it takes $O(\log n)$ time

Overall $O(n \log n)$