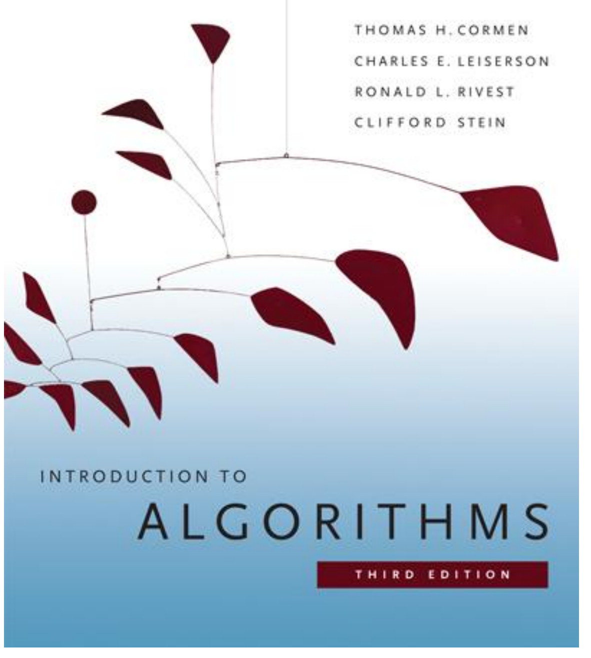
6.006- *Introduction to Algorithms*



Lecture 4

Menu

- Priority Queues
- Heaps
- Heapsort

Priority Queue

A data structure implementing a set S of elements, each associated with a key, supporting the following operations:

insert(S, x) : insert element x into set S

max(S) : return element of S with largest key

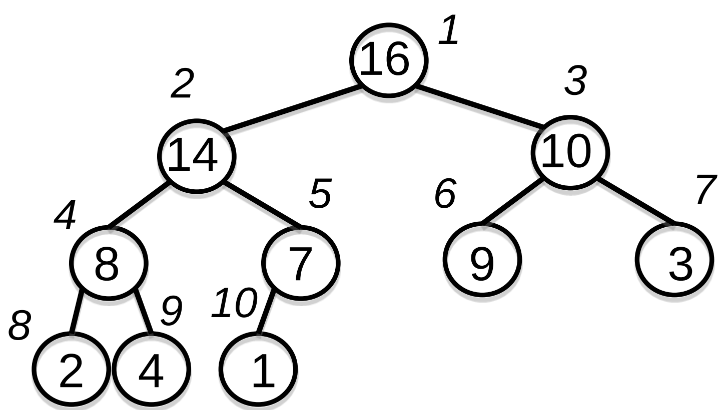
extract_max(S) : return element of S with largest key and remove it from S

increase_key(S, x, k) : increase the value of element x' s key to new value k (assumed to be as large as current value)

Heap

- Implementation of a priority queue
- An array, visualized as a nearly complete binary tree
- Max Heap Property: The key of a node is $\geq$ than the keys of its children

  (Min Heap defined analogously)



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

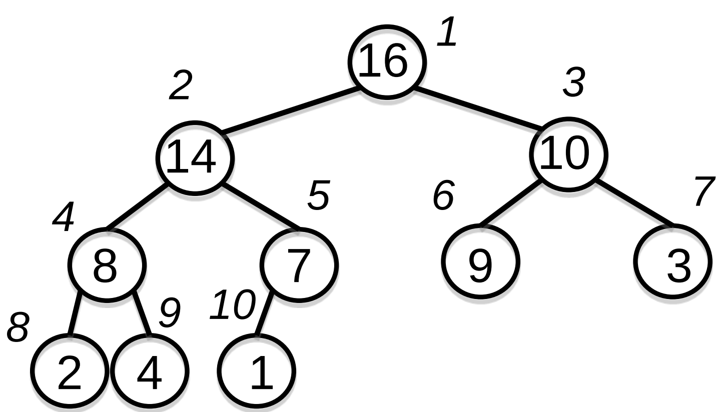
Heap as a Tree

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



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

No pointers required! Height of a binary heap is O(lg n)

Heap Operations

build_max_heap : produce a max-heap from an unordered array

max_heapify : correct a single violation of the heap property in a subtree at its root

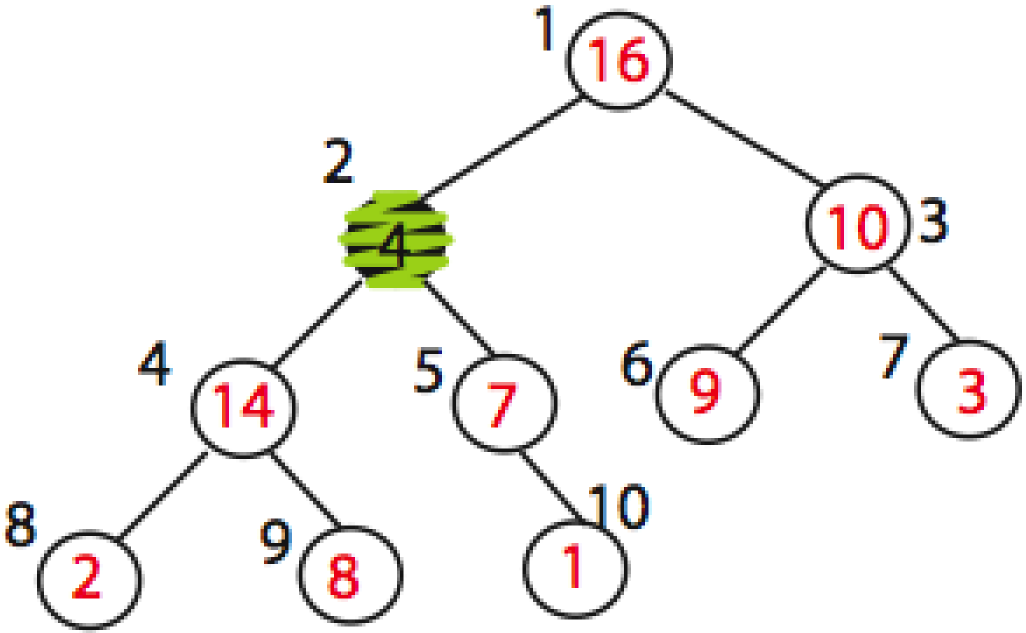
insert, extract_max, heapsort

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

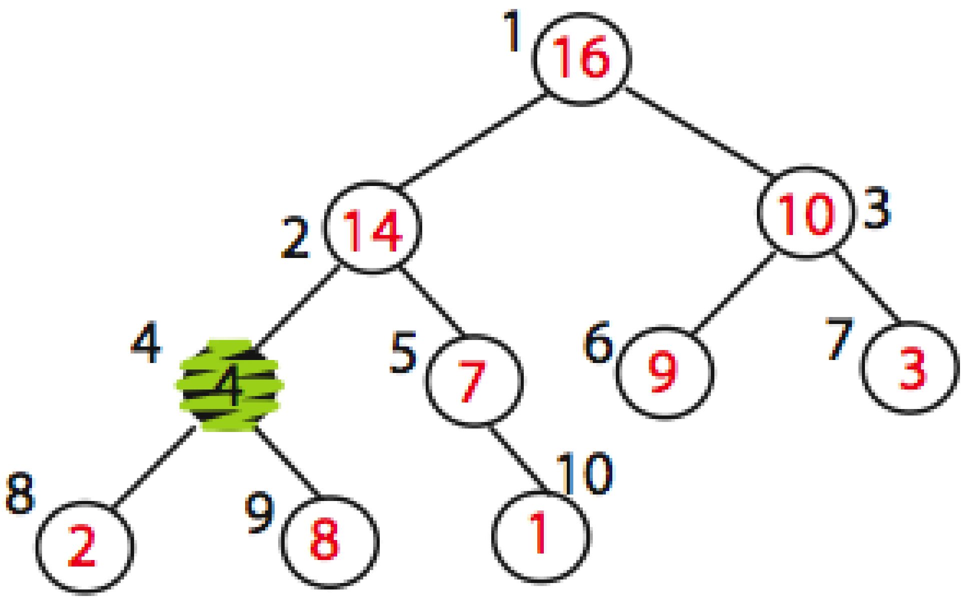
Max_heapify (Example)



MAX_HEAPIFY (A,2)
heap_size[A] = 10

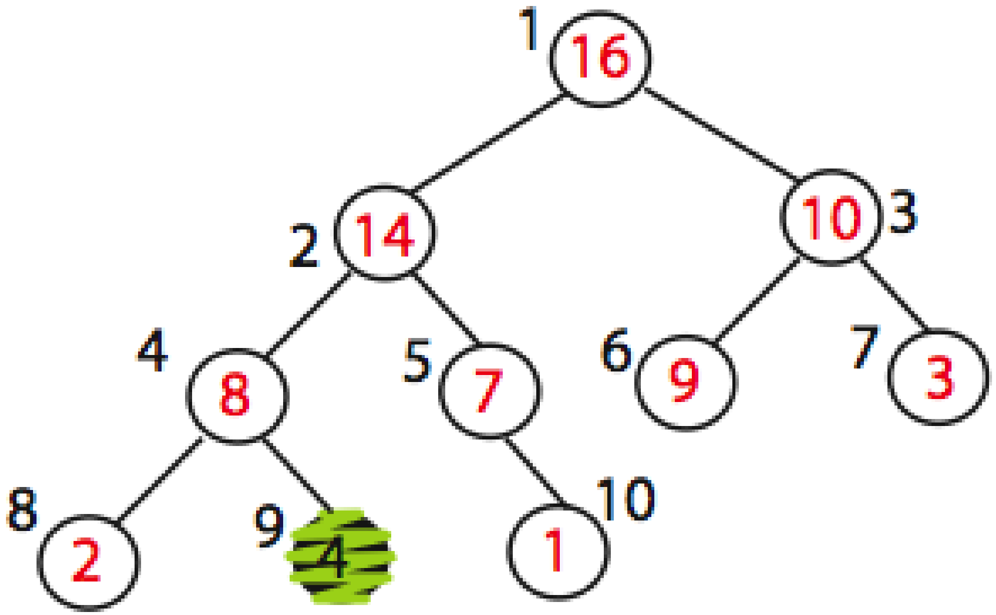
Node 10 is the left child of node 5 but is drawn to the right for convenience

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(log n)

Max_Heapify Pseudocode

l = left(i)

r = right(i)

if (l <= heap-size(A) and A[l] > A[i])

then largest = l else largest = i

if (r <= heap-size(A) and A[r] > A[largest])

then largest = r

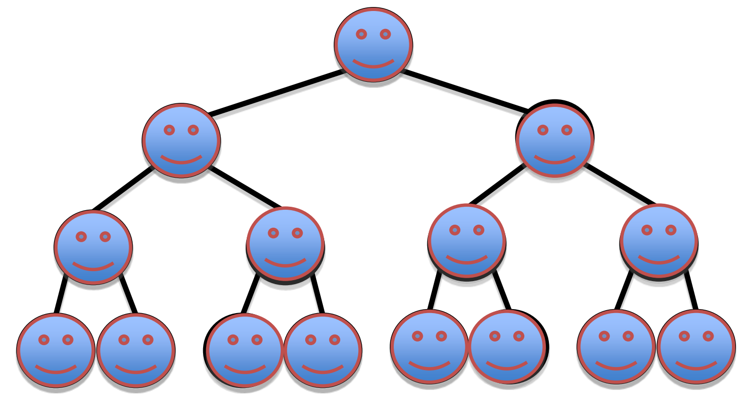
if largest $\neq$ i

then exchange A[i] and A[largest]

Max_Heapify(A, largest)

Build_Max_Heap(A)

Converts A[1…n] to a max heap



Build_Max_Heap(A):
    for i=n/2 downto 1
        do Max_Heapify(A, i)

Why start at n/2?

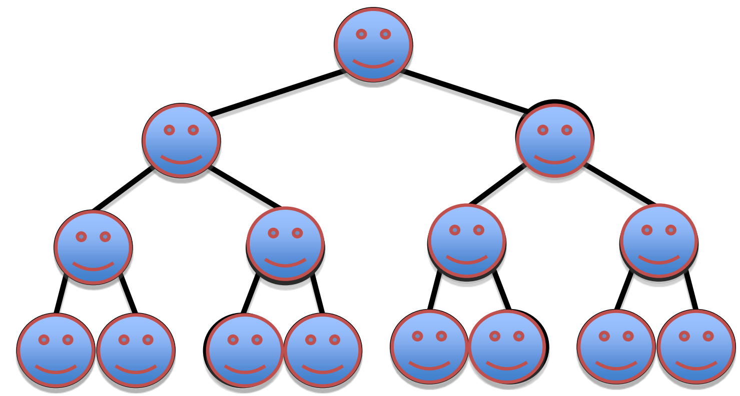
Because elements A[n/2 + 1 … n] are all leaves of the tree
2i > n, for i > n/2 + 1

Time=? O(n log n) via simple analysis

Build_Max_Heap(A) Analysis

Converts A[1…n] to a max heap

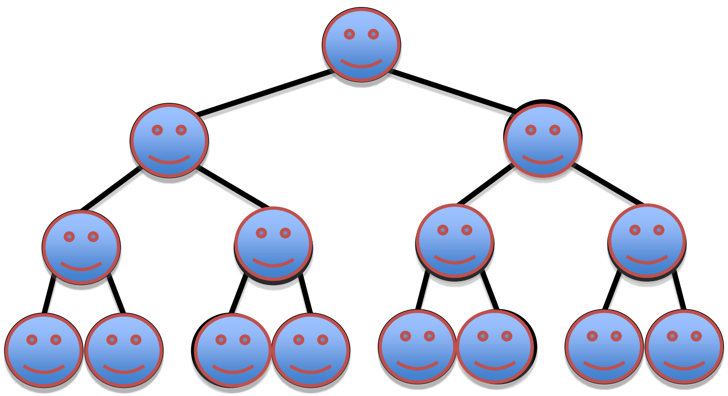
Build_Max_Heap(A):
    for i=n/2 downto 1
        do Max_Heapify(A, i)



Observe however that Max_Heapify takes O(1) for time for nodes that are one level above the leaves, and in general, O(l) for the nodes that are l levels above the leaves. We have n/4 nodes with level 1, n/8 with level 2, and so on till we have one root node that is lg n levels above the leaves.

Build_Max_Heap(A) Analysis

Converts A[1…*n*] to a max heap



Build_Max_Heap(A):
    for i=n/2 downto 1
        do Max_Heapify(A, i)

Total amount of work in the for loop can be summed as:

$$n/4\ (1\ c) + n/8\ (2\ c) + n/16\ (3\ c) + \ldots + 1\ (\lg n\ c)$$

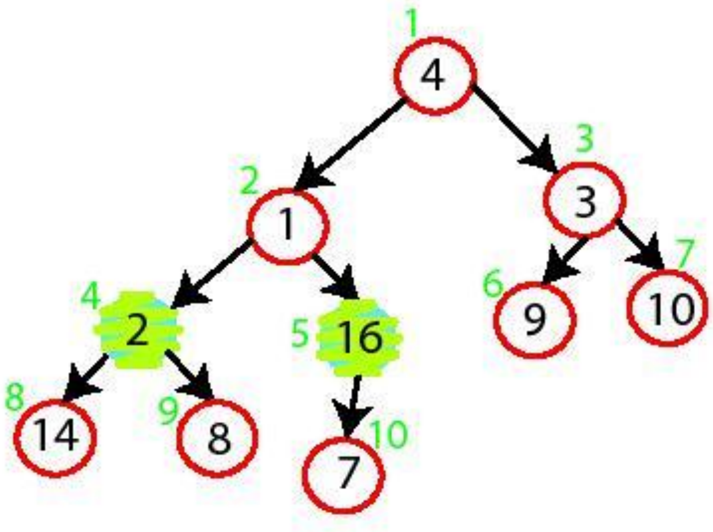
Setting $n/4 = 2^k$ and simplifying we get:

$$c\ 2^k(1/2^0 + 2/2^1 + 3/2^2 + \ldots\ (k+1)/2^k)$$

The term is brackets is bounded by a constant!

This means that Build_Max_Heap is $O(n)$

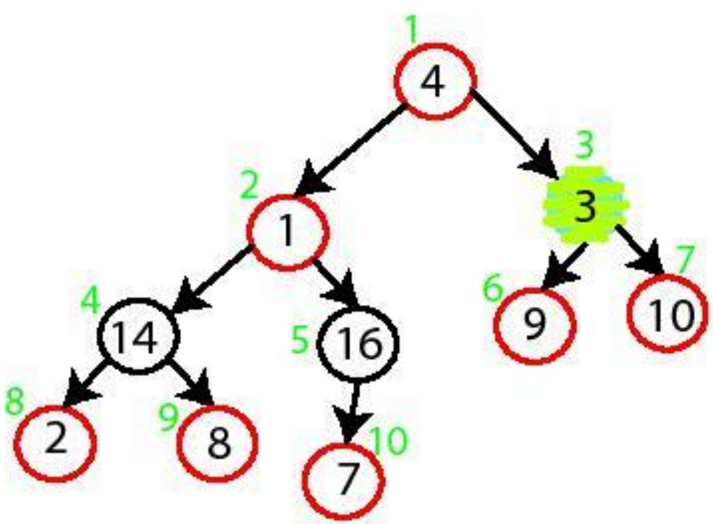
Build-Max-Heap Demo



A

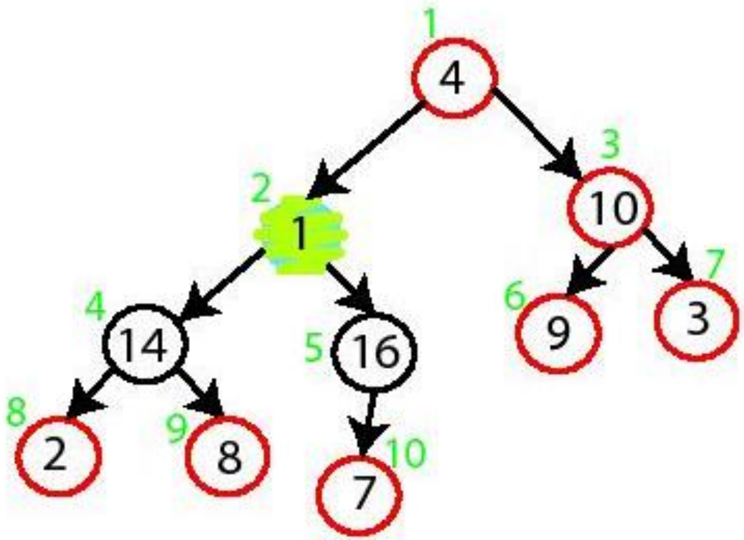
4	1	3	2	16	9	10	14	8	7

MAX-HEAPIFY (A,5)
no change
MAX-HEAPIFY (A,4)
Swap A[4] and A[8]

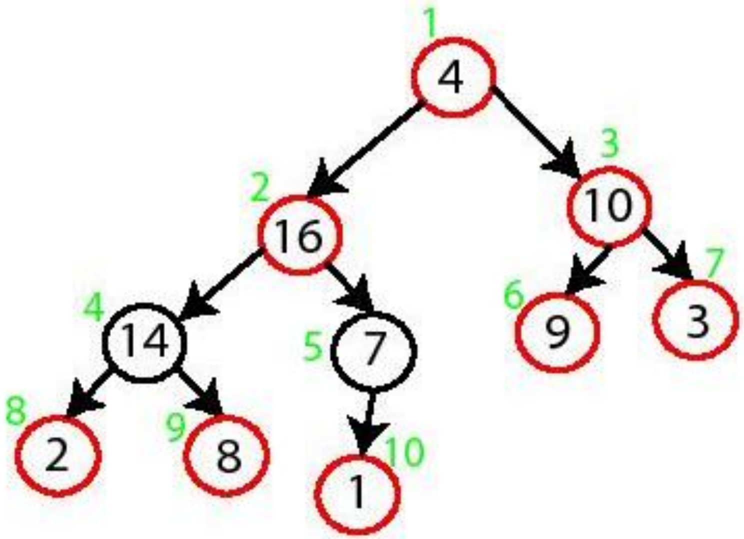


MAX-HEAPIFY (A,3)
Swap A[3] and A[7]

Build-Max-Heap Demo



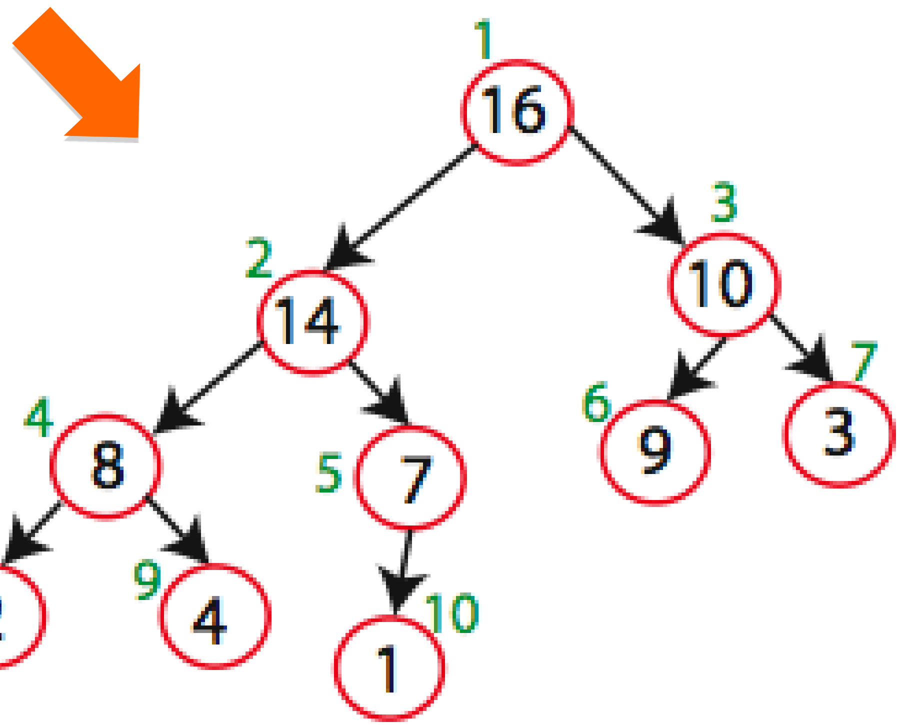
MAX-HEAPIFY (A,2)
Swap A[2] and A[5]
Swap A[5] and A[10]



MAX-HEAPIFY (A,1)
Swap A[1] with A[2]
Swap A[2] with A[4]
Swap A[4] with A[9]

Build-Max-Heap

A	4	1	3	2	16	9	10	14	8	7



Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

3. Swap elements A[n] and A[1]:
   now max element is at the end of the array!

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

3. Swap elements A[*n*] and A[1]:
   now max element is at the end of the array!

4. Discard node *n* from heap
   (by decrementing heap-size variable)

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

3. Swap elements A[*n*] and A[1]:
   now max element is at the end of the array!

4. Discard node *n* from heap
   (by decrementing heap-size variable)

5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.

Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

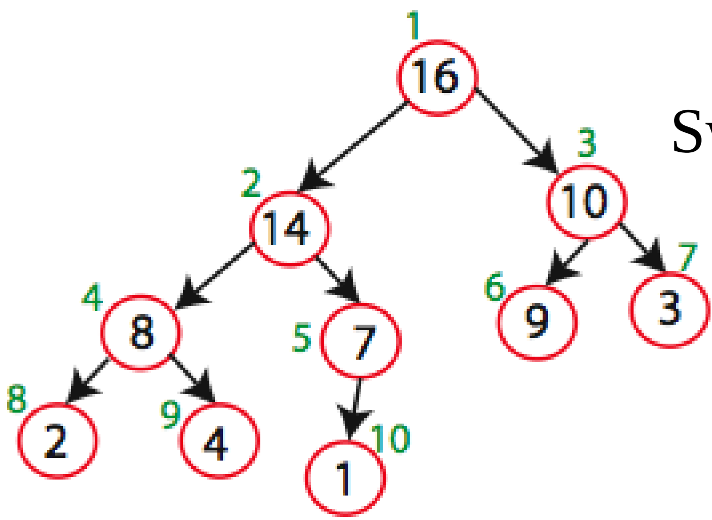
3. Swap elements A[*n*] and A[1]:
   now max element is at the end of the array!

4. Discard node *n* from heap
   (by decrementing heap-size variable)

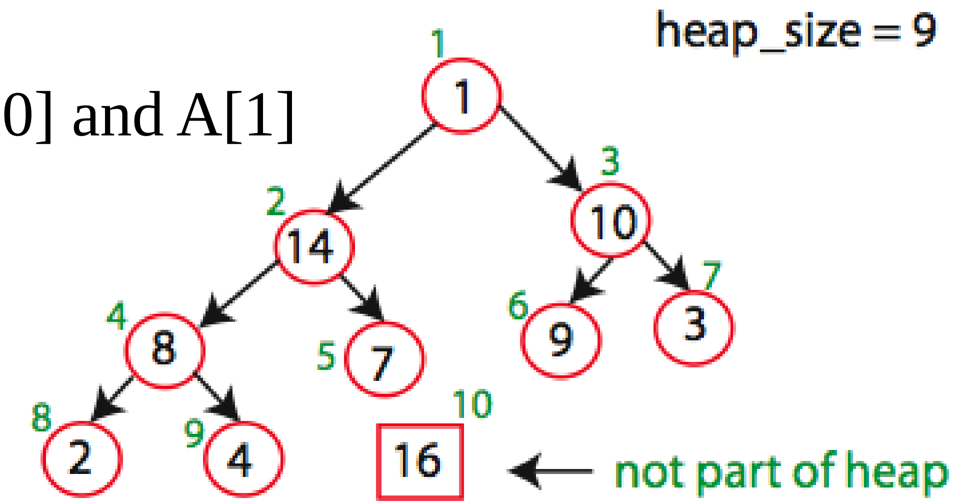
5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.

6. Go to Step 2 unless heap is empty.

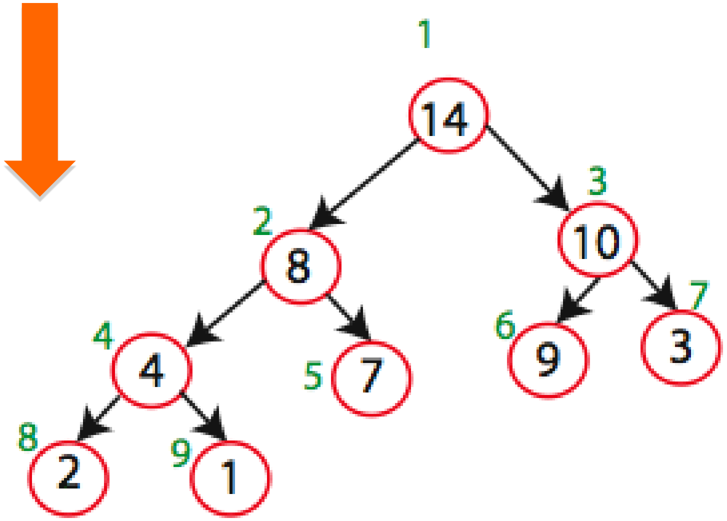
Heap-Sort Demo



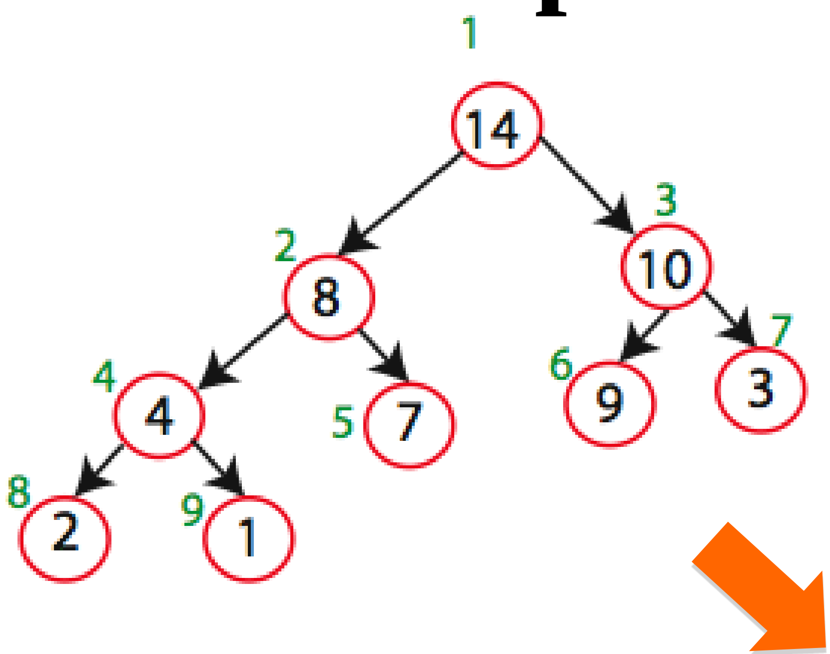
Swap A[10] and A[1]



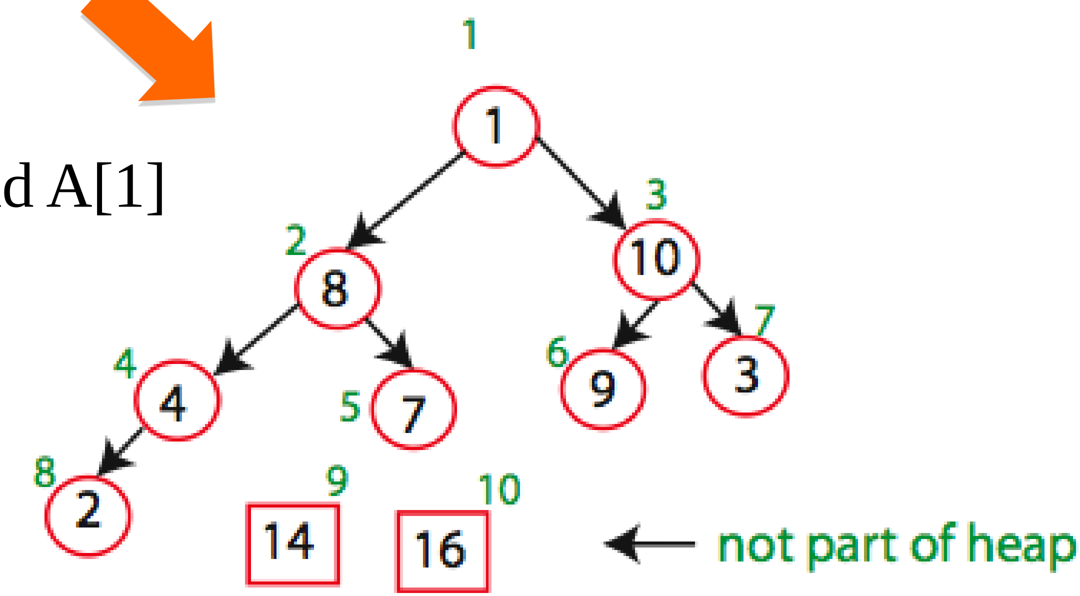
Max_heapify(A,1)



Heap-Sort Demo

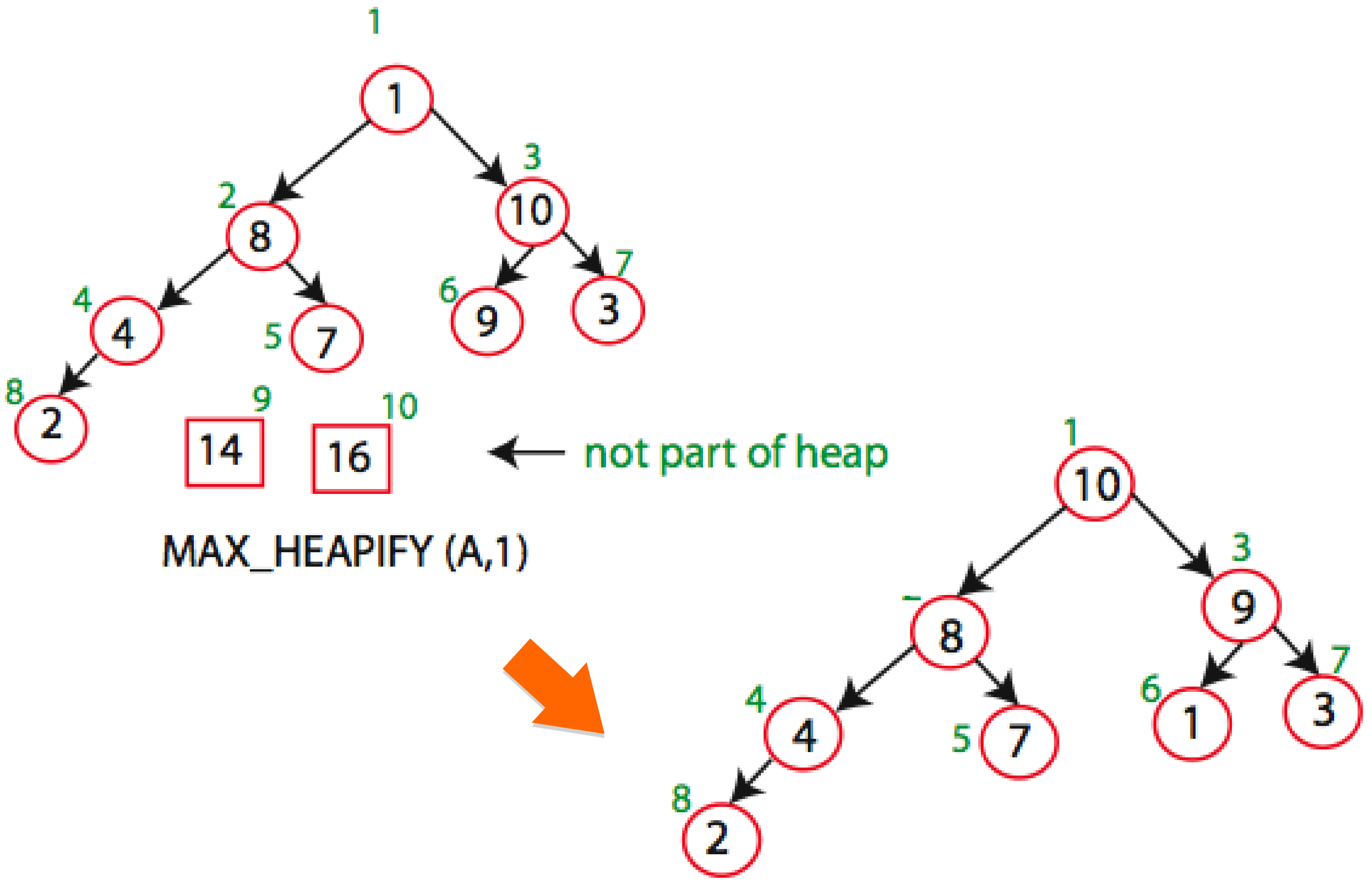


Swap A[9] and A[1]

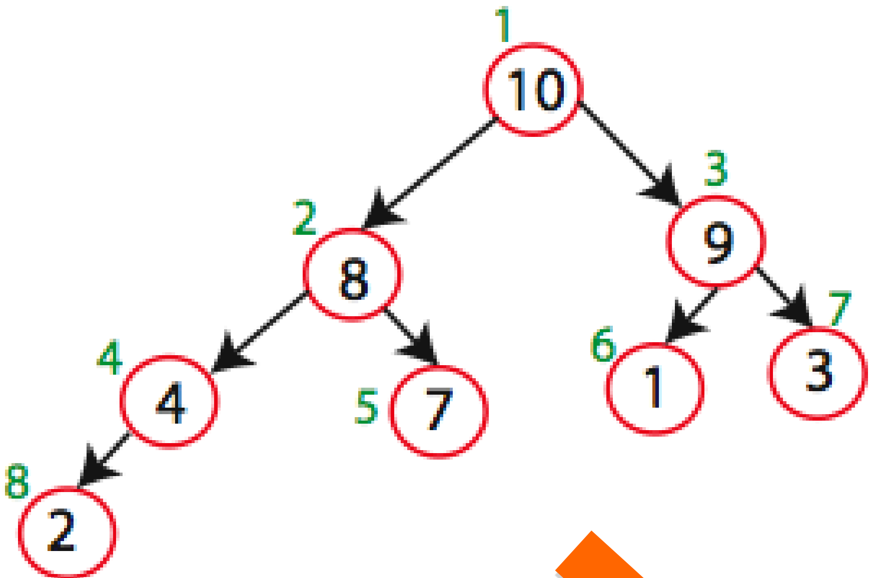


MAX_HEAPIFY (A,1)

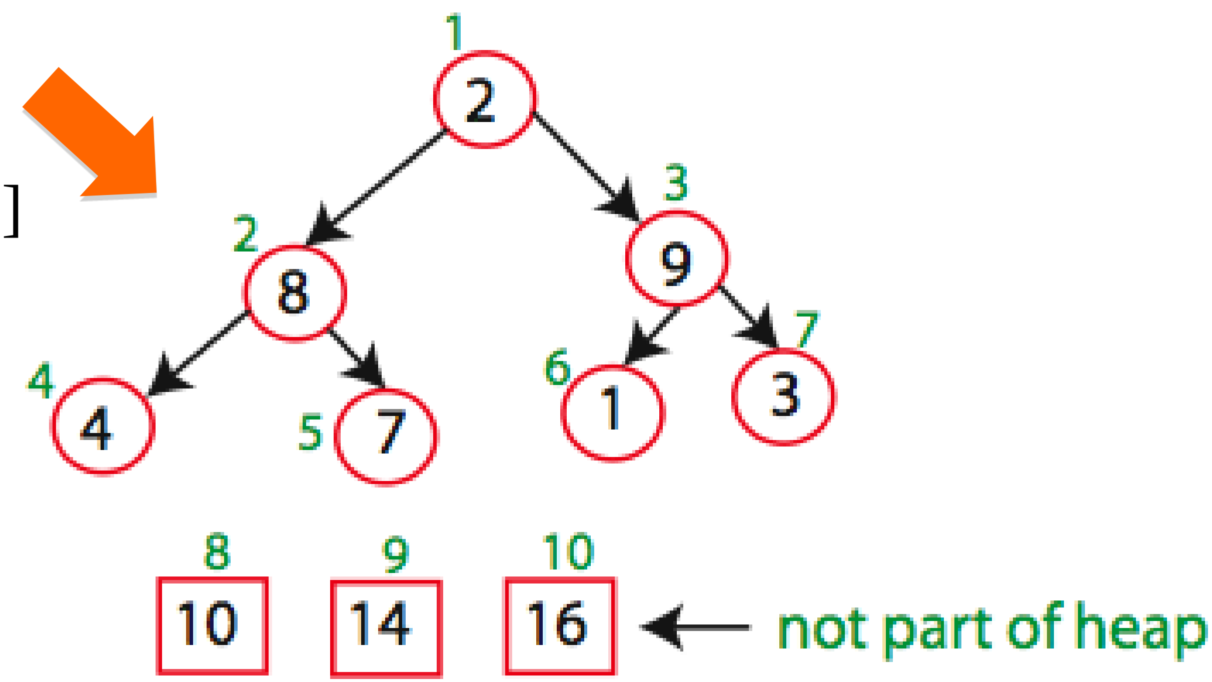
Heap-Sort Demo



Heap-Sort Demo



Swap A[8] and A[1]



Heap-Sort

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 $O(n \log n)$