CSCE 2110
Foundations of Data Structures

Priority Queues

Contents

- Priority Queues
- Heaps
- Heapsort

Priority Queue ADT

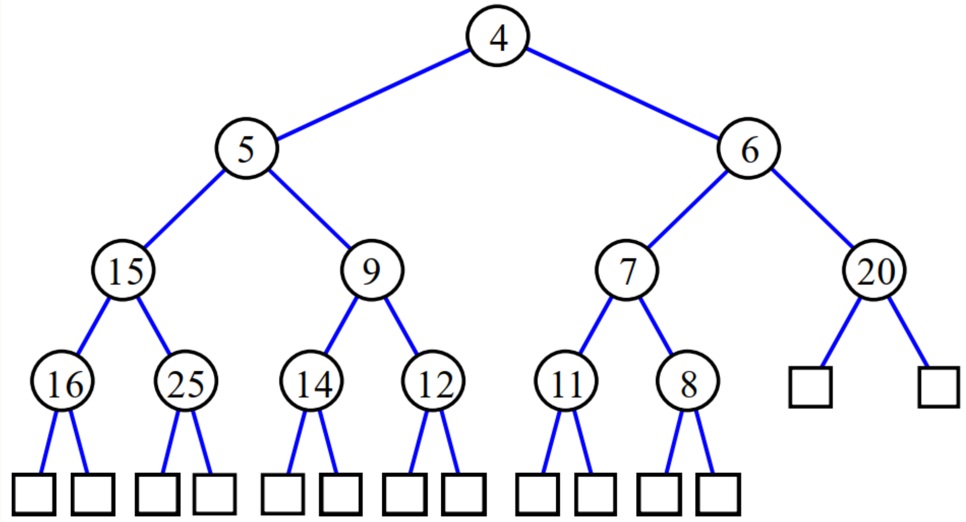
- Priority Queue is an extension of queue with following properties:
  - Entries consist of key (priority) and value.
  - Entries in priority queue are ordered by key
  - An entry with high key is dequeued before an element with low key.
  - If two entries have the same key, they are served according to their order in the queue.

Priority Queue ADT

- A typical priority queue supports following operations:
  - insert(key, value): Inserts an item with given key.
  - min/max(): Returns the smallest/largest key item.
  - removemin()/removemax(): Removes the smallest/largest key item.

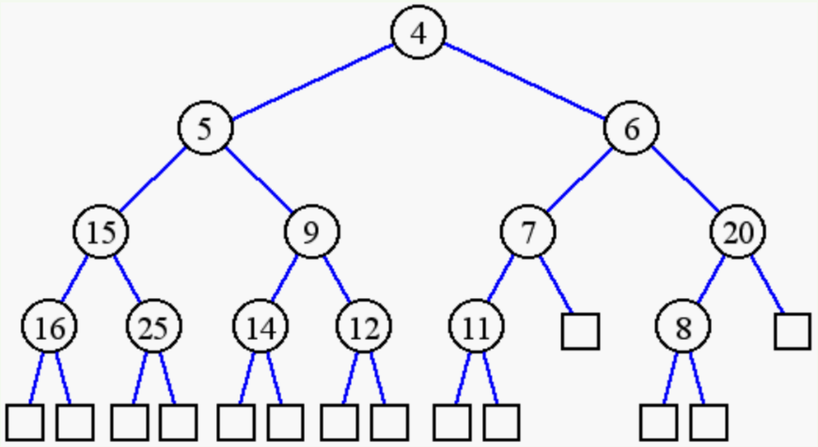
Heap

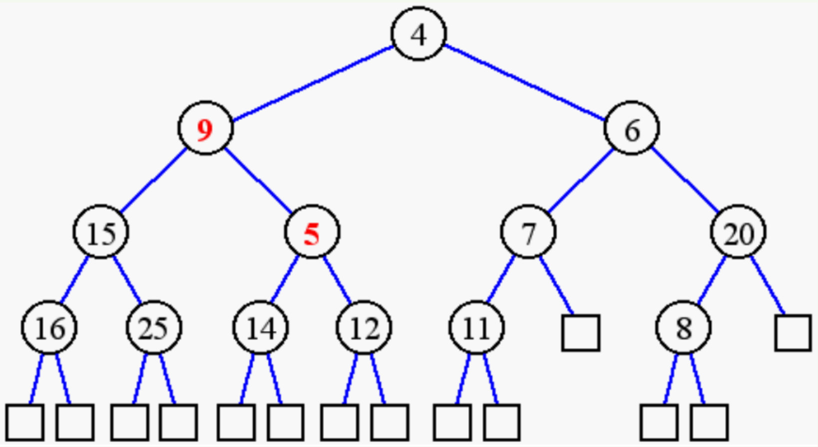
- Tree-based data structure
- A complete tree
  - every level, except possibly the last, is filled, and all nodes are as far left as possible
- Satisfies the heap property:
  - if P is a parent node of C, then the key of P is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) the key of C.



Heap

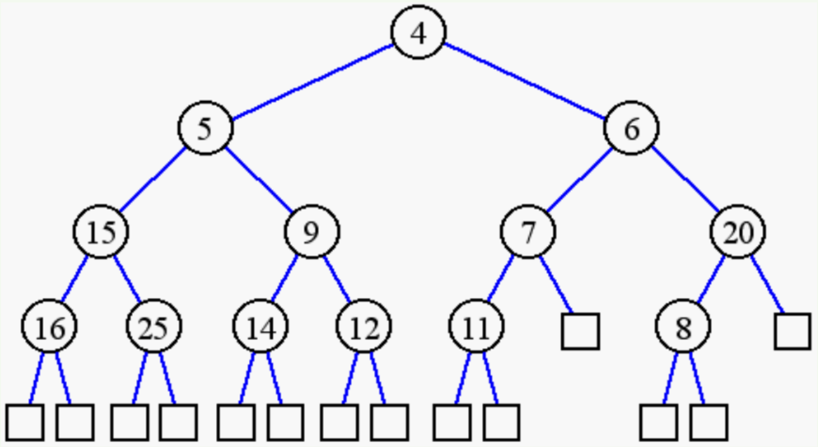
(min) Heap or Not a (min) Heap?



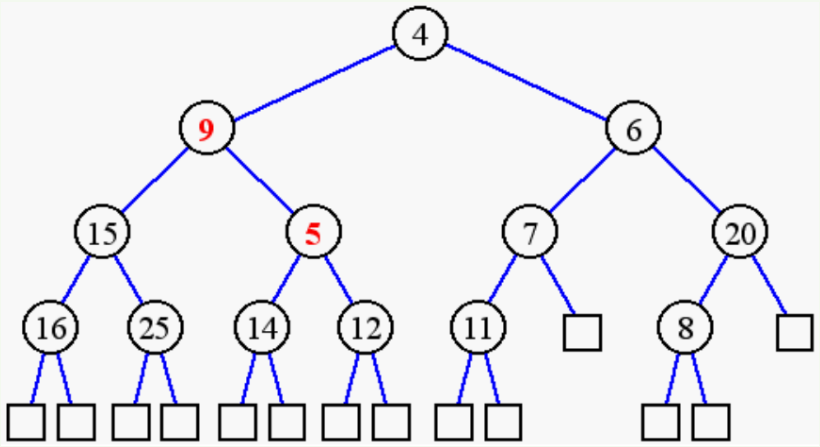


Heap

(min) Heap or Not a (min) Heap?



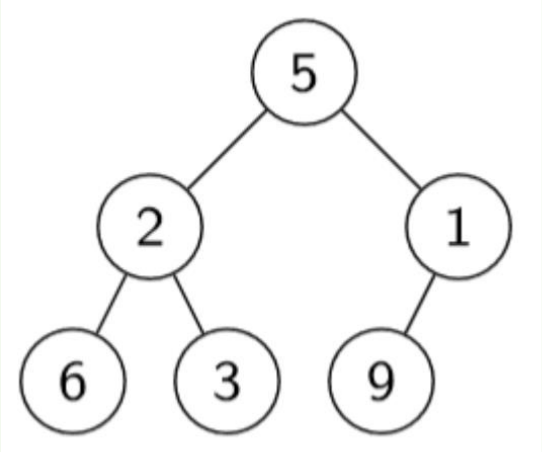
Min heap

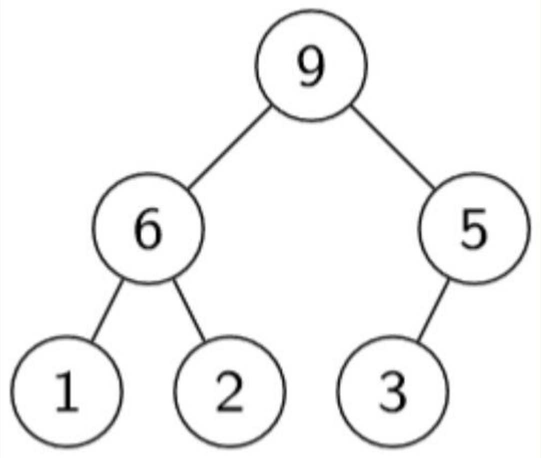


NOT a min heap

Heaps – Max Heap

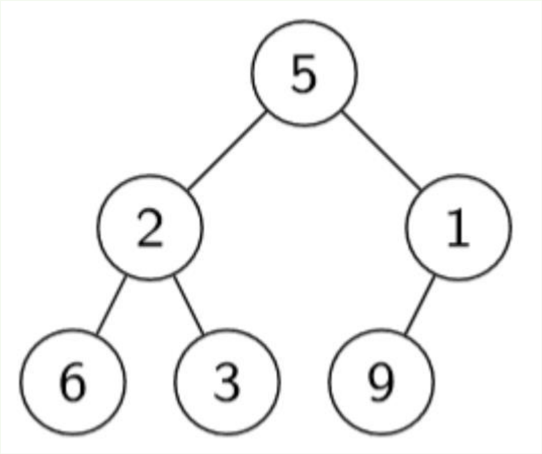
- A max heap is a heap such that for each node except the root, the parent of node i is greater than or equal to node i (max-heap property)



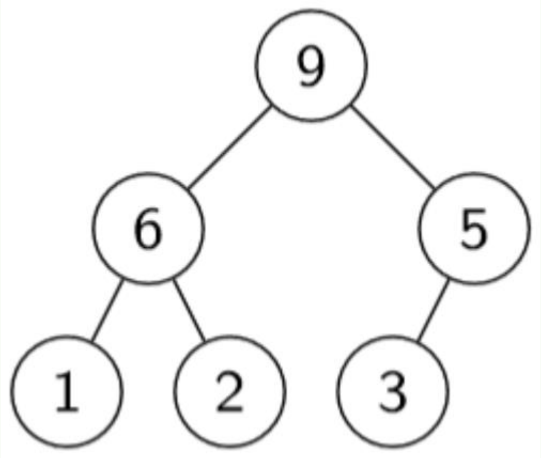


Heaps – Max Heap

- A max heap is a heap such that for each node except the root, the parent of node i is greater than or equal to node i (max-heap property)



NOT a max heap

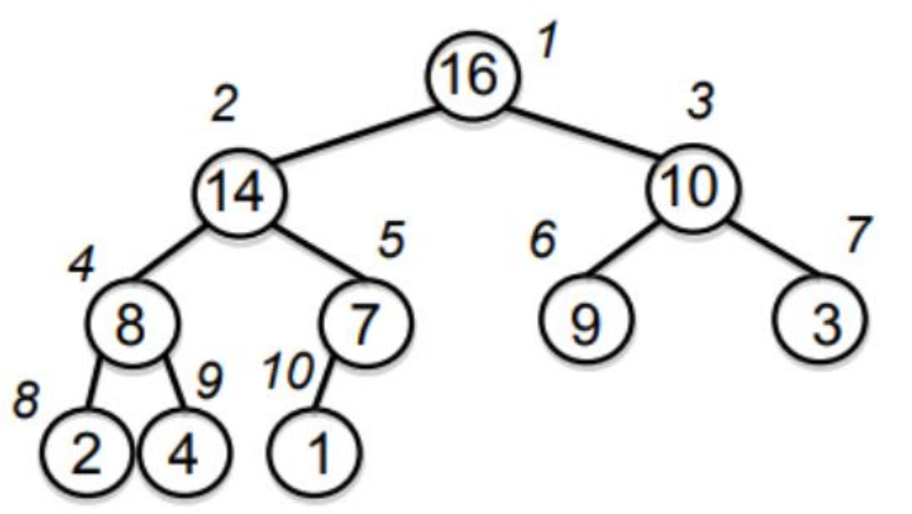


Max heap

For max heap, where is the largest element?
where is the smallest element?

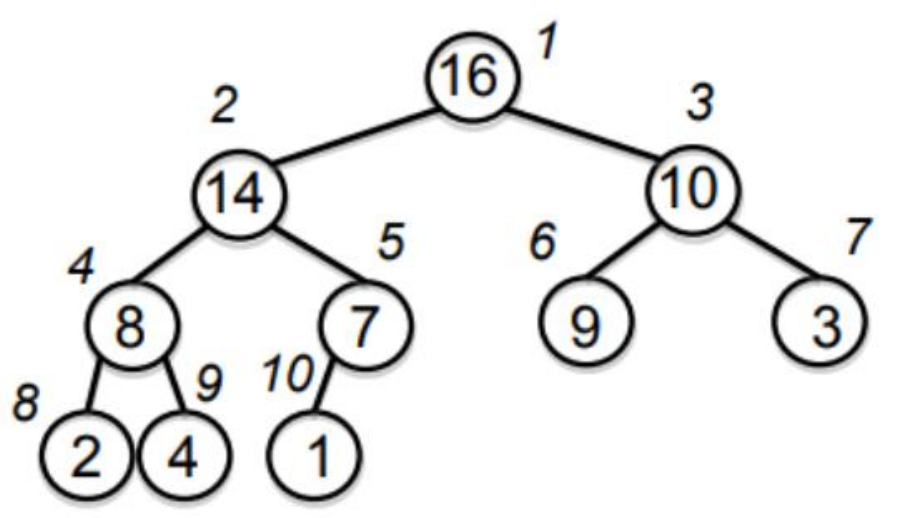
Binary Heap

- An array, visualized as a complete binary tree
- Often refer as heap
- Height of a binary heap is $O(\lg n)$



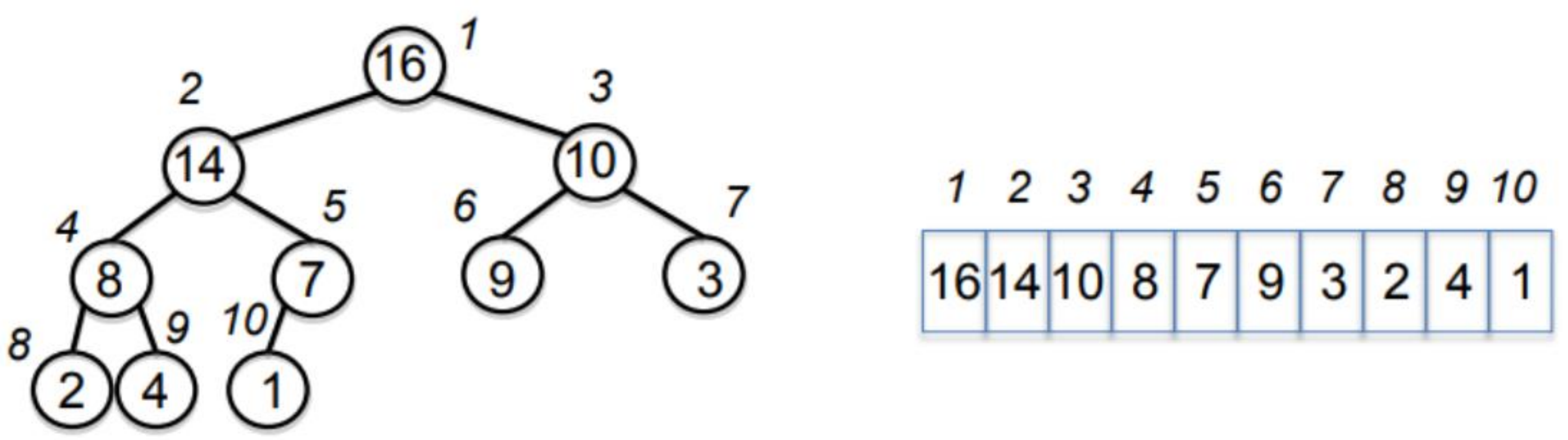
Heap as a Tree

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



Heap as a Tree

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



Heap Operations

For max heap:

- max: return the maximum item
- extract_max: return and remove the maximum item

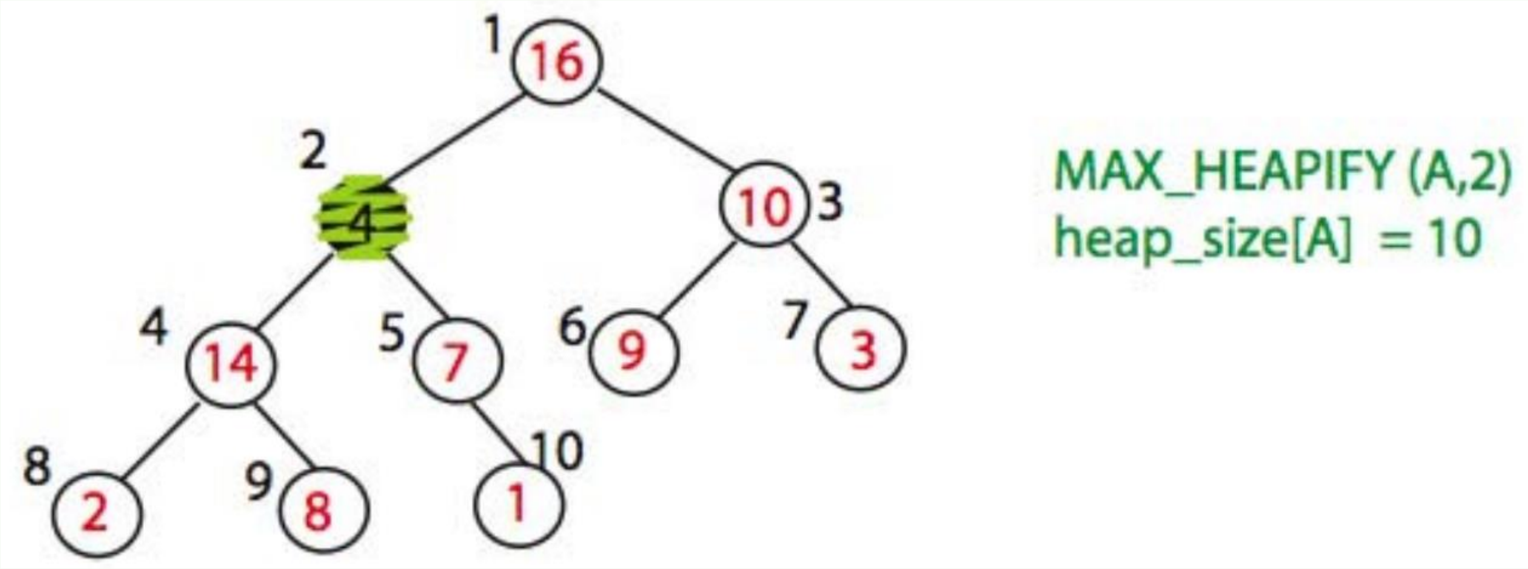
- build_max_heap: produce a max-heap from an unordered array
- max_heapify: correct a single violation of the heap property in a subtree at its root
- insert
- heapsort

max_heapify

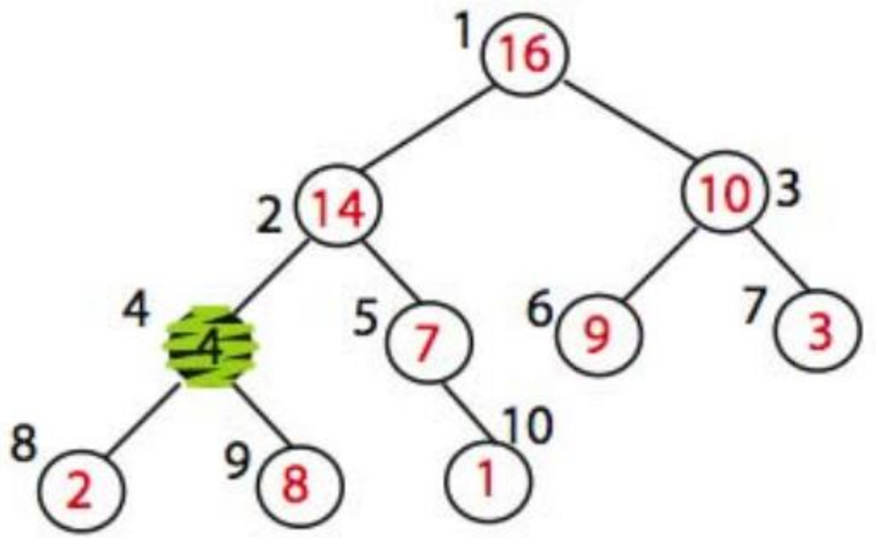
- Assume that the trees/subtrees rooted at left(i) and right(i) are max-heaps

- 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

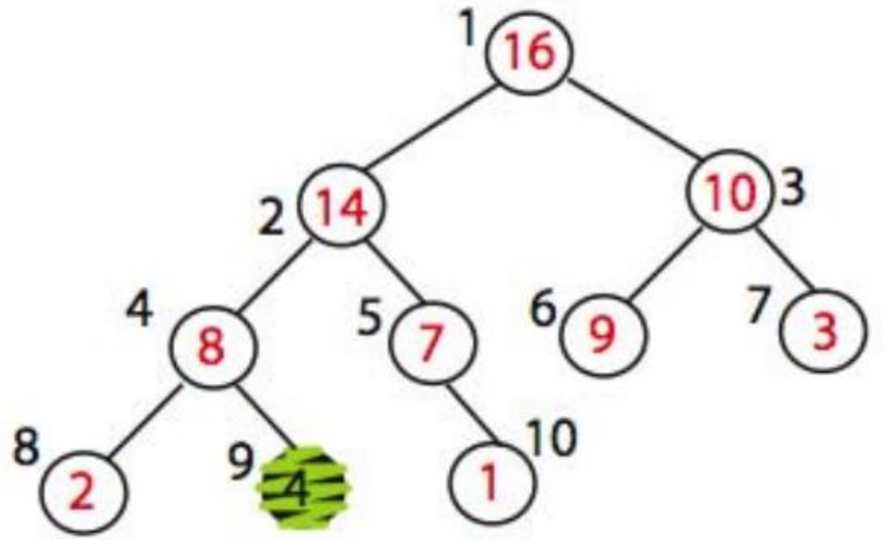


max_heapify: example



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

max_heapify: pseudocode

```
max_heapify(A, i):
    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 != i
       then exchange A[i] and A[largest]
            max_heapify(A, largest)
```

build_max_heap(A)

- Converts $A[1 \dots n]$ to a max heap

```
build_max_heap(A):
    for i=n/2 down to 1
         do max_heapify(A, i)
```

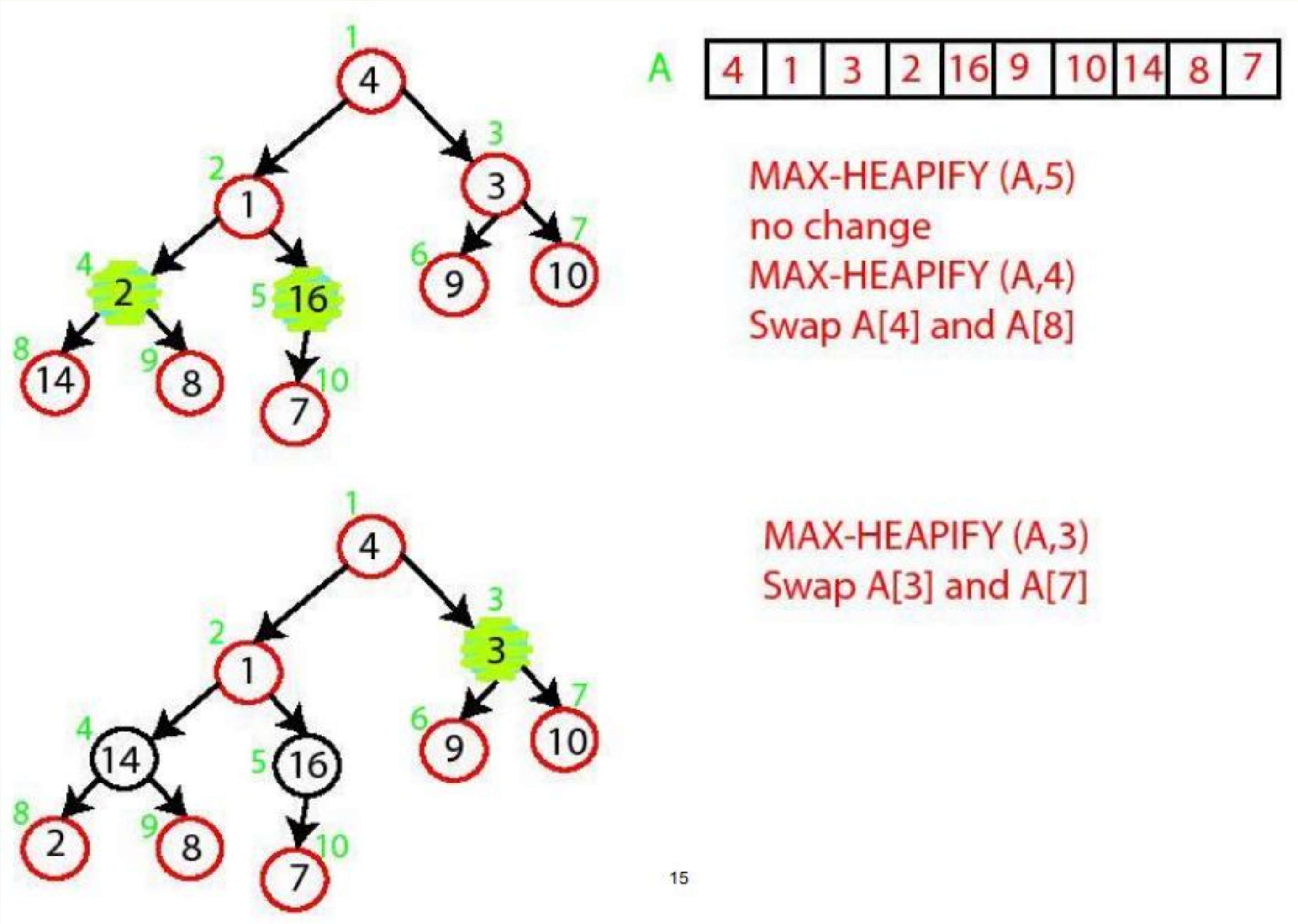
build_max_heap(A)

- Converts $A[1 \ldots n]$ to a max heap

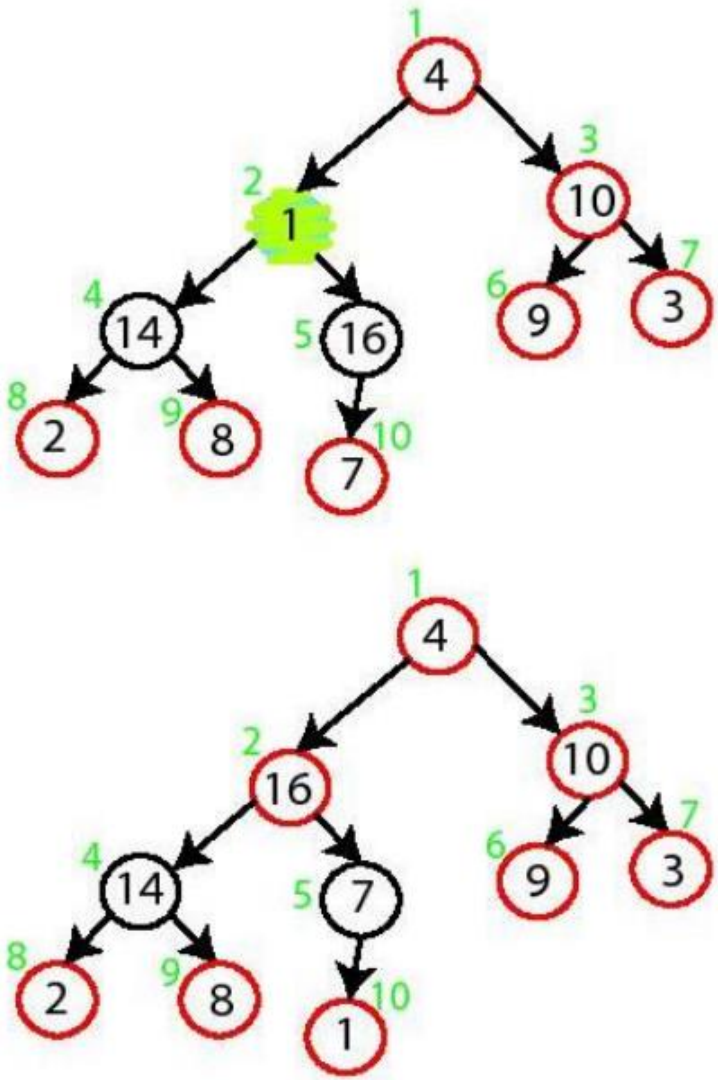
```
build_max_heap(A):
    for i=n/2 down to 1
        do max_heapify(A, i)
```

- Why start at $n/2$?

build_max_heap Demo



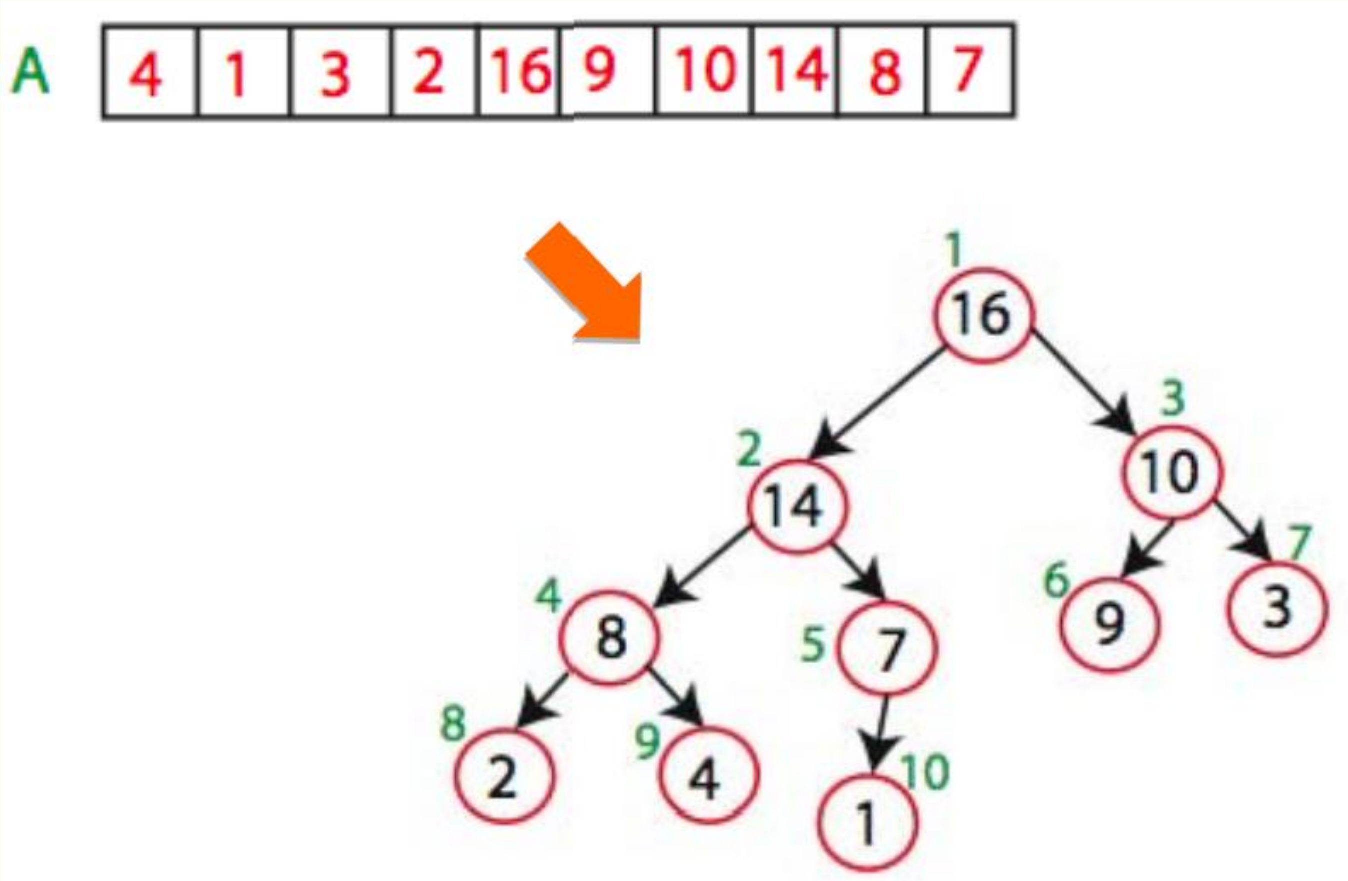
build_max_heap Demo



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]

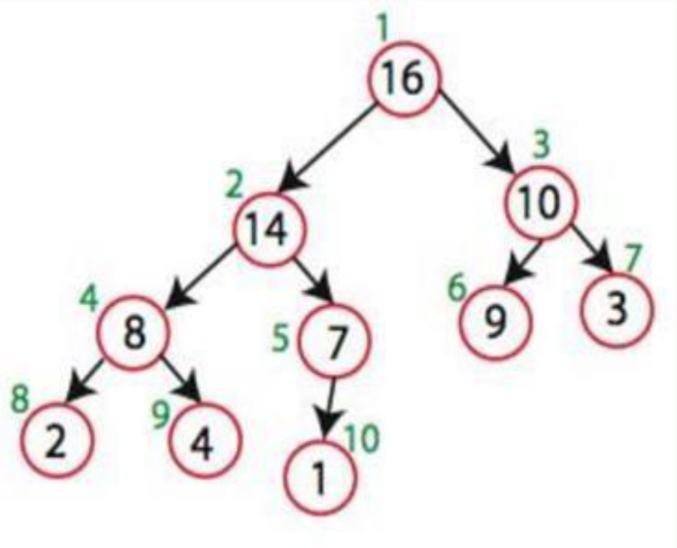
build_max_heap Demo



Insert

$insert(k)$

- Let X be the new entry k
- Place X at the bottom level of the tree, at first free spot from left; i.e., first free location in array
- Bubbles up tree until heap property is satisfied (max-heapify)
  - Repeat:
    - Compare X's key with its parent's key
    - If X's key is larger, exchange



True or False

- A max heap forms, if keys 2^{k-1} to 1 are inserted in order into an initially empty array.

max

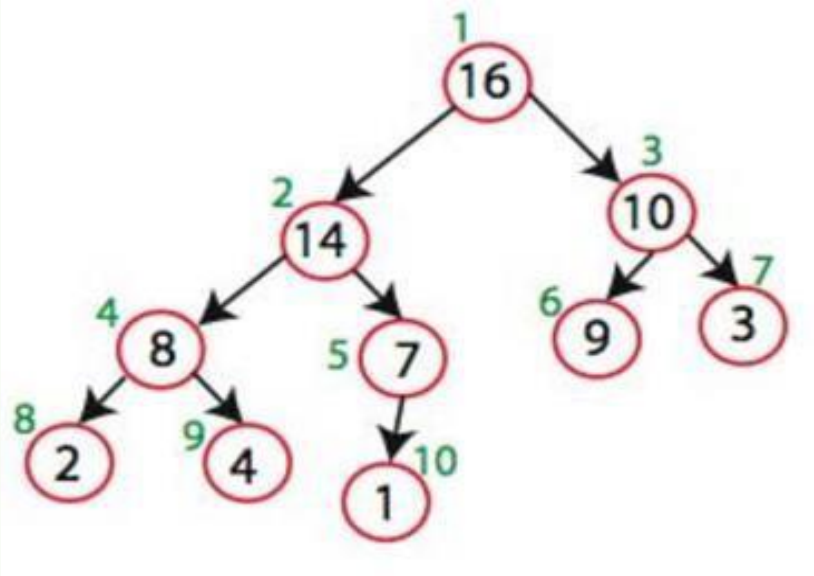
- ?

max

- Return entry at root

extract_max

- Return and remove entry at root
- Save item at root for return value
- Fill root with last item "X" in tree
- Bubble "X" down the heap (max-heapify)
  - Repeat: If X < one or both of its children, swap X with its maximum child



Heapsort

How does knowing the maximum element of an array A help in sorting A?

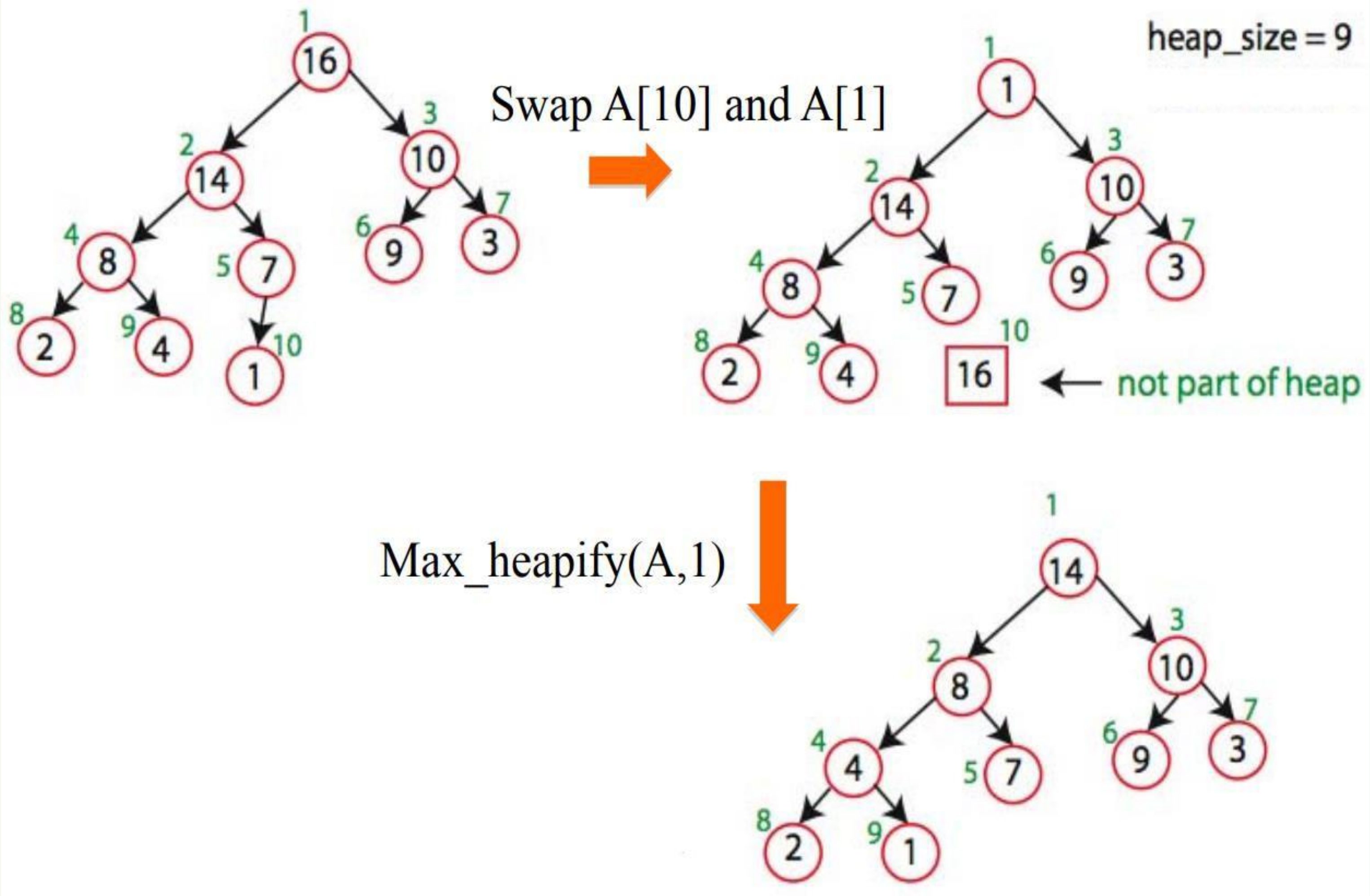
- Build a heap for A
- Get the maximum
- Put it in place (exchange with the last item)
- Update the heap accordingly, reduce size, max-heapify
- Get the new maximum
- Put it in place
- Update the heap accordingly, reduce size, max-heapify
- ...

Heapsort

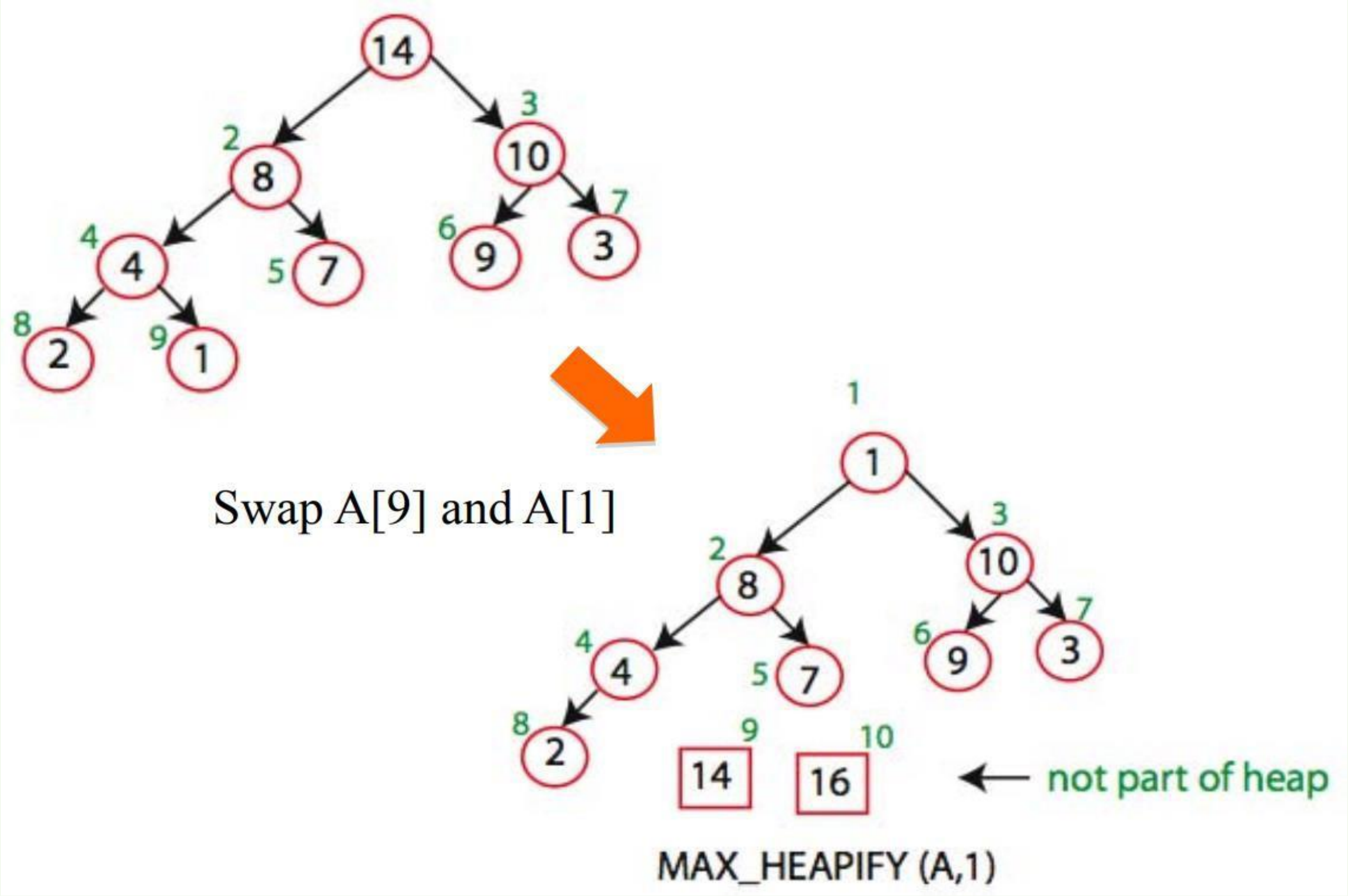
Sorting Strategy:

- 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.

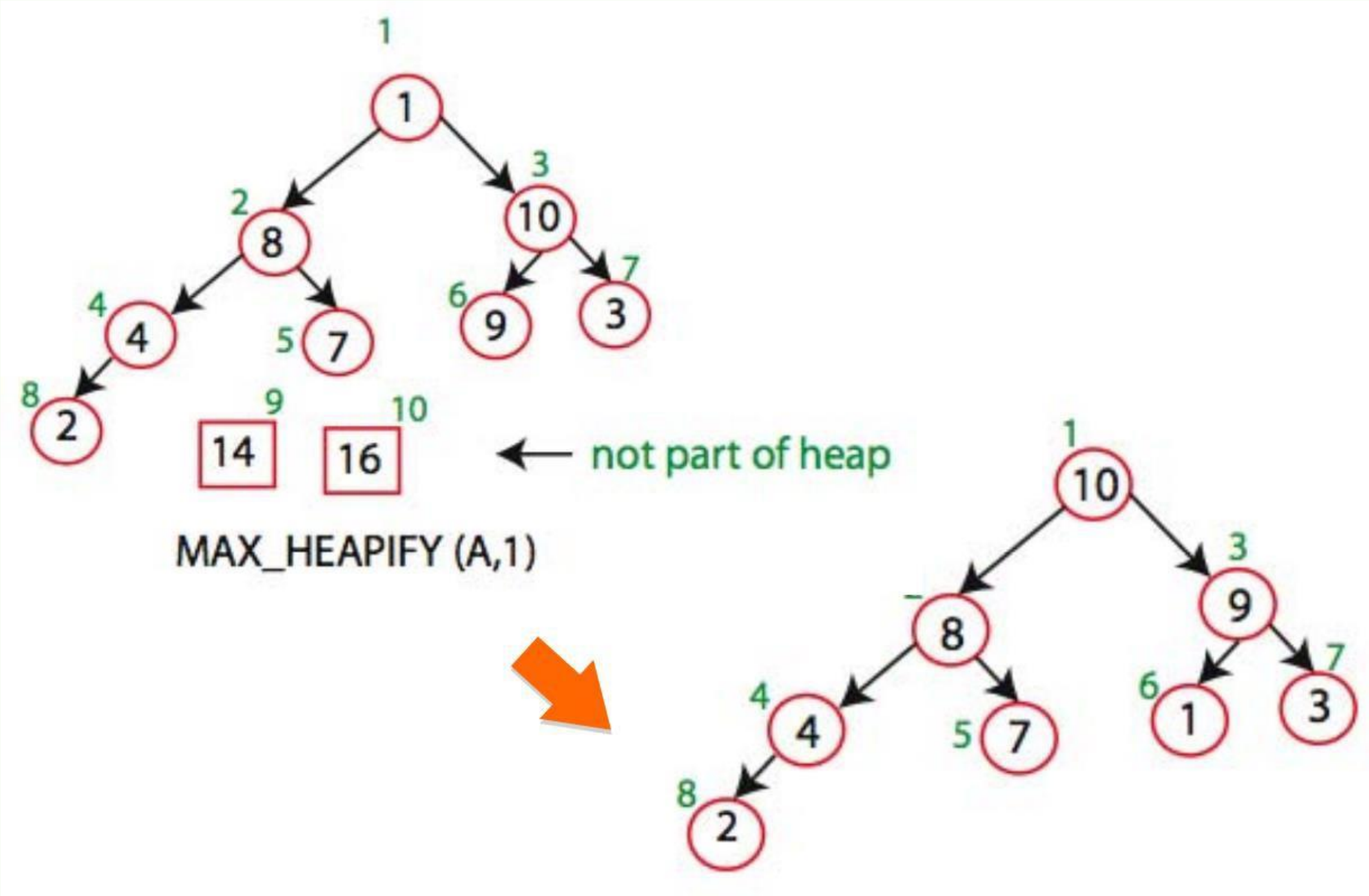
Heapsort Demo



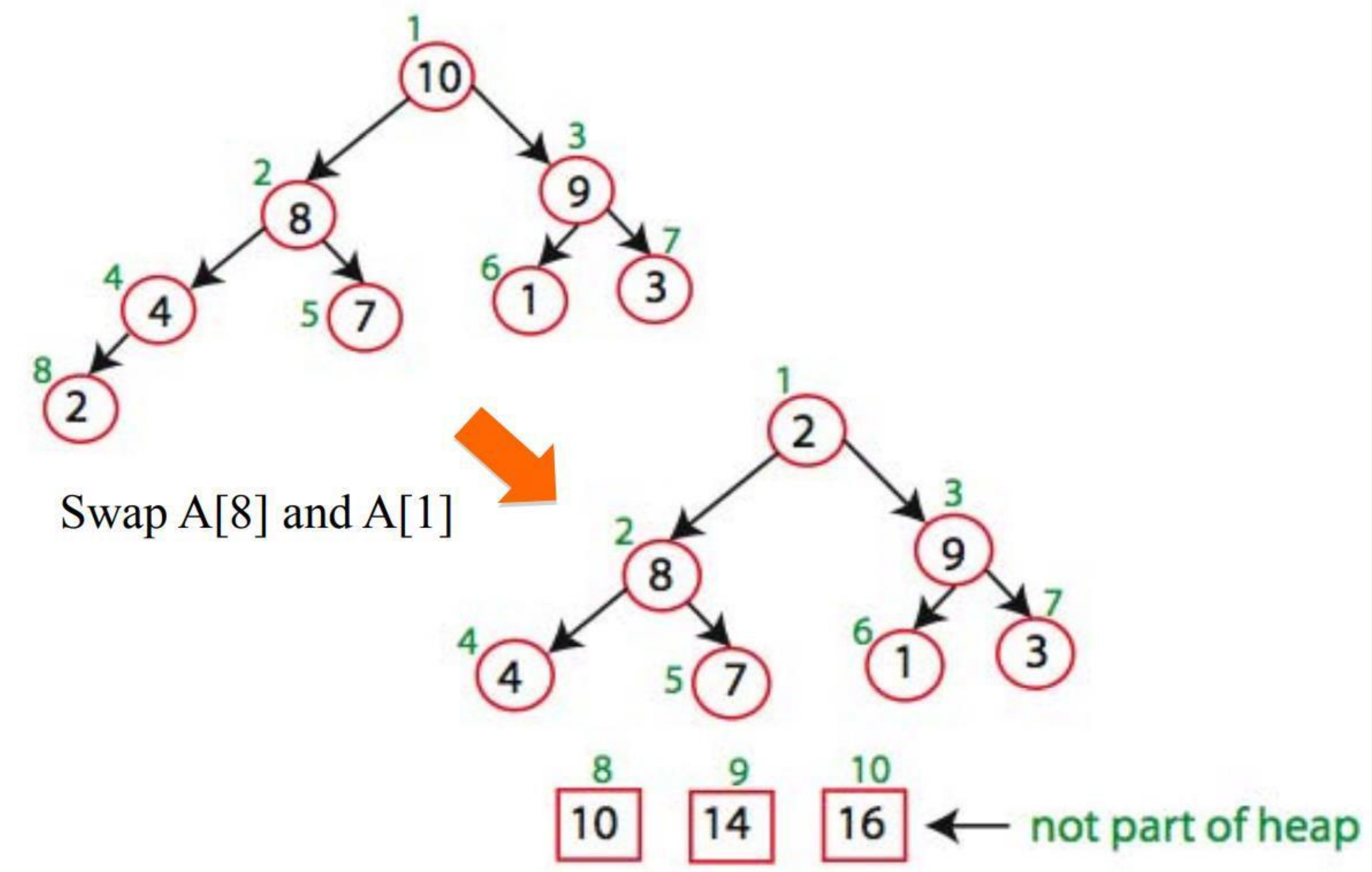
Heapsort Demo



Heapsort Demo



Heapsort Demo



Heapsort

- after n iterations the Heap is empty
- every iteration involves a swap and a max_heapify operation;