Lecture 4
Divide & conquer: sorting, max subarray, median finding

Spring 2022

Zhihua Jiang

Part I

- Sorting
  - Insertion Sort
  - Merge Sort
- Priority Queues
- Heaps
- Heapsort

The problem of sorting

Input: array $A[1\ldots n]$ of numbers.

Output: permutation $B[1\ldots n]$ of A such that $B[1] \leq B[2] \leq \cdots \leq B[n]$.

e.g. $A = [7, 2, 5, 5, 9.6] \rightarrow B = [2, 5, 5, 7, 9.6]$

How can we do it efficiently ?

Why Sorting?

- Obvious applications
  - Organize an MP3 library
  - Maintain a telephone directory
- Problems that become easy once items are in sorted order
  - Find a median, or find closest pairs
  - Binary search, identify statistical outliers
- Non-obvious applications
  - Data compression: sorting finds duplicates
  - Computer graphics: rendering scenes front to back

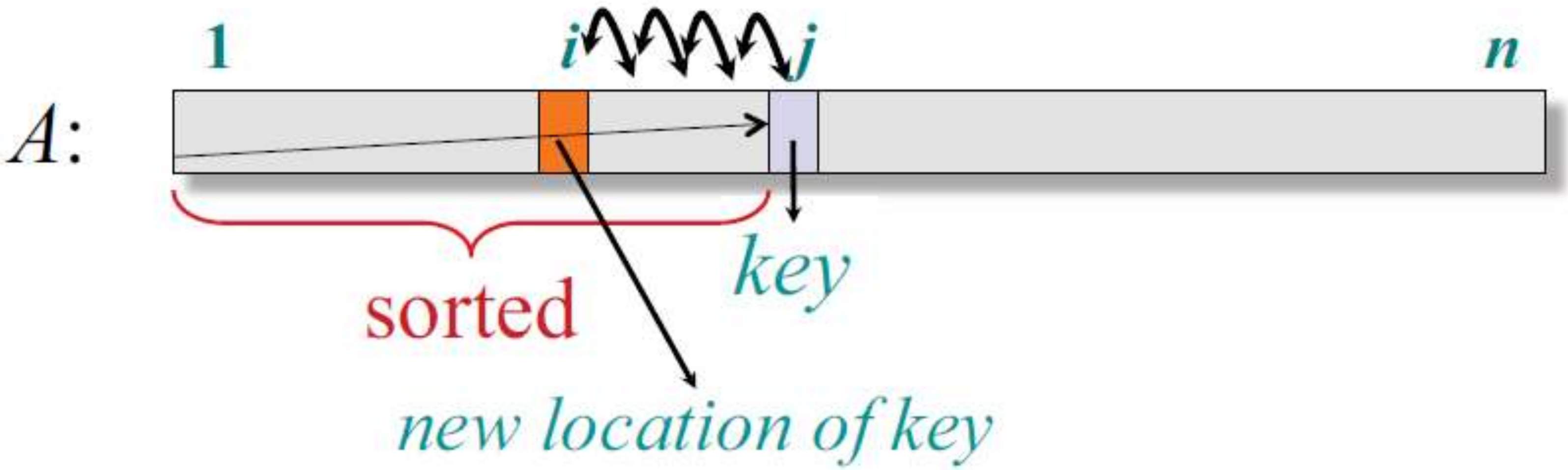
Insertion sort

INSERTION-SORT (A, n) $\quad \triangleright\ A[1 \,..\, n]$

for $j \leftarrow 2$ **to** n

insert key A[j] **into the (already sorted) sub-array** A[1 .. j-1]**.**
by pairwise key-swaps down to its right position

Illustration of iteration j



Example of insertion sort

8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*done*

Running time? $\Theta(n^2)$ because $\Theta(n^2)$ compares and $\Theta(n^2)$ swaps

e.g. when input is $A = [n, n-1, n-2, \ldots, 2, 1]$

Binary Insertion sort

BINARY-INSERTION-SORT (A, n) $\triangleright A[1 \,..\, n]$

for $j \leftarrow 2$ **to** n

insert key A[j] **into the (already sorted) sub-array** A[1 .. j-1].

Use binary search to find the right position

Binary search with take $\Theta(\log n)$ time.
However, shifting the elements after insertion will still take $\Theta(n)$ time.

Complexity: $\Theta(n \log n)$ comparisons

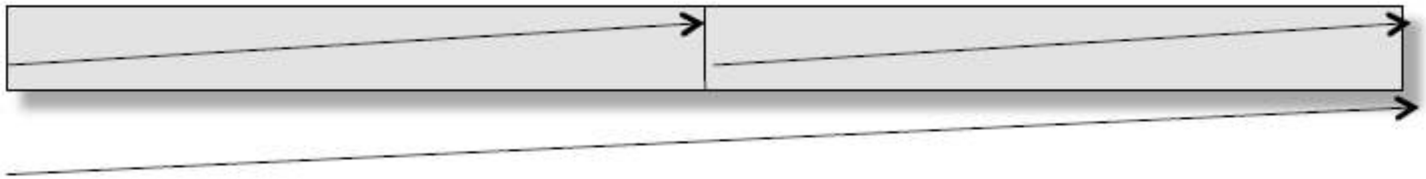
(n^2) swaps

Meet Merge Sort

divide and conquer

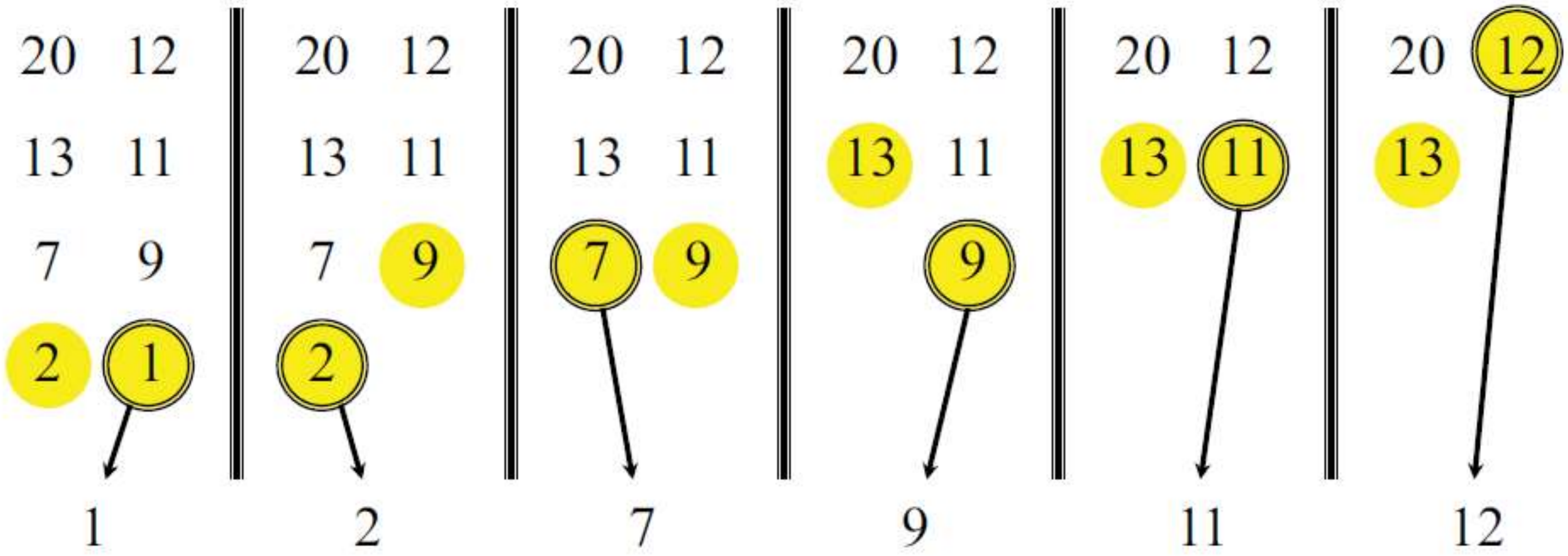
MERGE-SORT $A[1 \,..\, n]$

1. If $n = 1$, done (nothing to sort).
2. Otherwise, recursively sort $A[\,1 \,..\, n/2\,]$ and $A[\,n/2+1 \,..\, n\,]$.
3. "***Merge***" the two sorted sub-arrays.



Key subroutine: **MERGE**

Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of n elements (linear time).

Analyzing merge sort

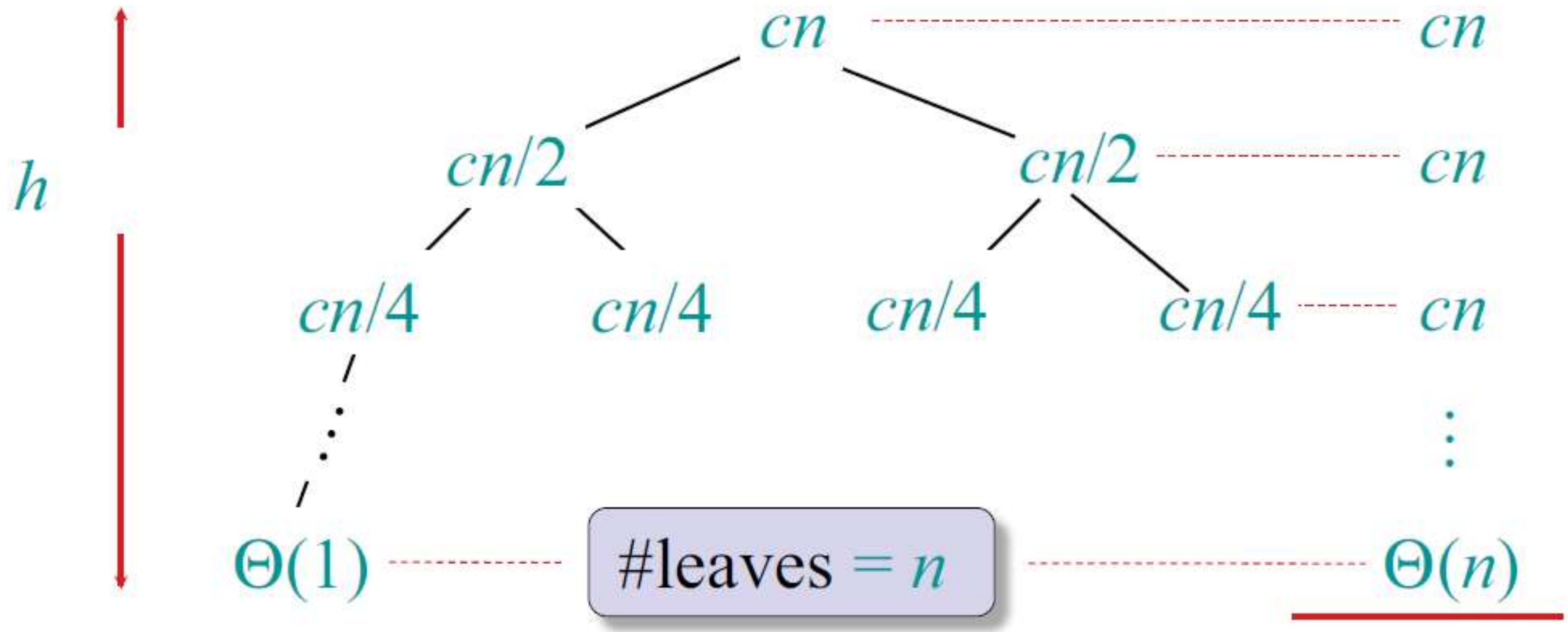
MERGE-SORT $A[1\,..\,n]$	$T(n)$
1. If $n = 1$, done.	$\Theta(1)$
2. Recursively sort $A[\,1\,..\,\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,..\,n\,]$.	$2T(n/2)$
3. ***"Merge"*** the two sorted lists	$\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

$T(n) = ?$

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

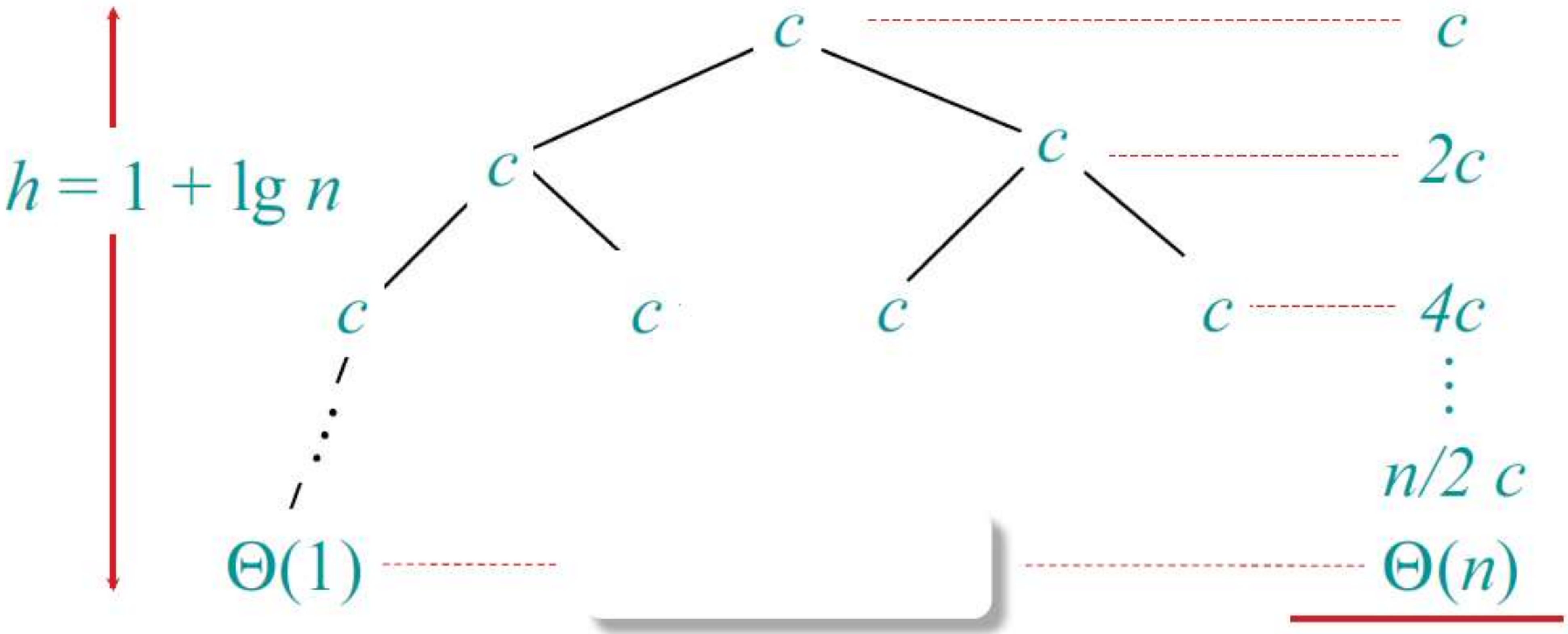


Total $= \Theta(n \lg n)$

Equal amount of work done at each level

Tree for different recurrence

Solve $T(n) = 2T(n/2) + c$, where $c > 0$ is constant.



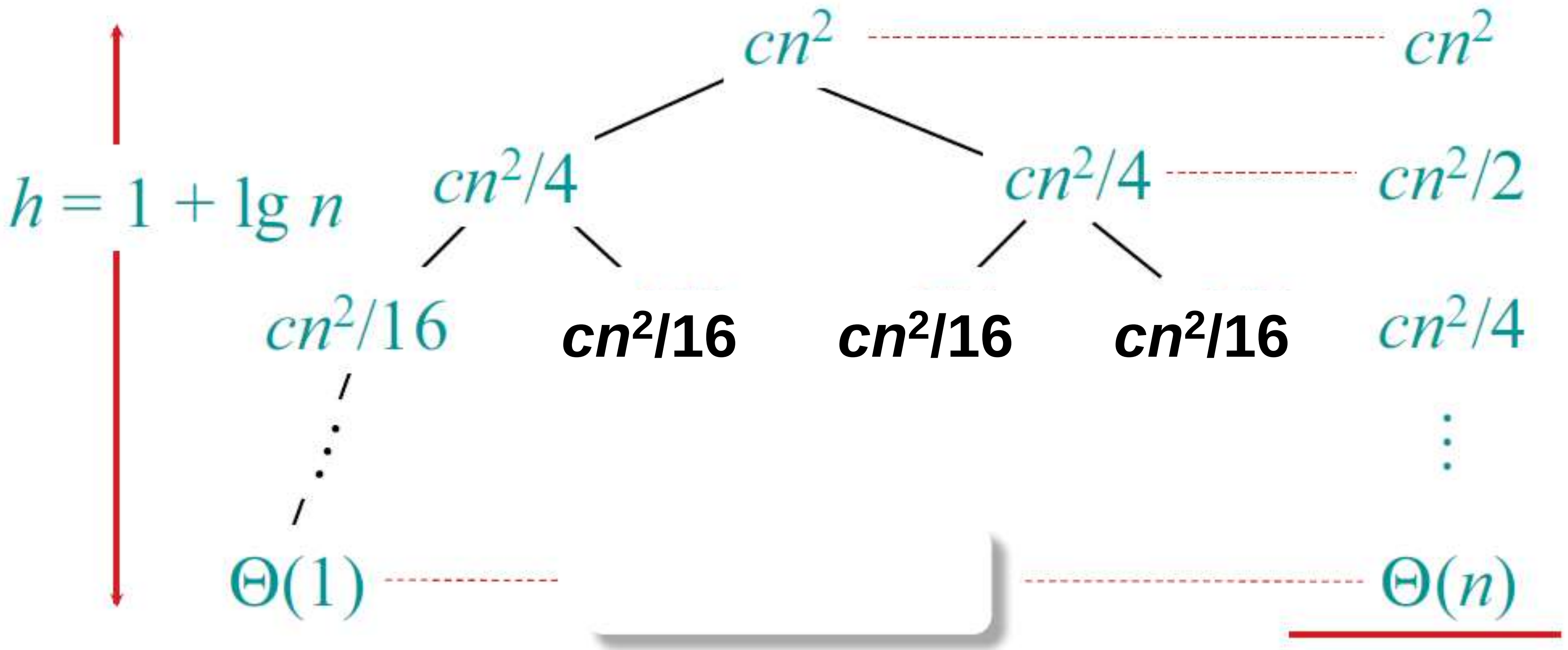
Note that $1+2+4+\ldots+n/2=n-1$

Total $= \Theta(n)$

All the work done at the leaves

Tree for yet another recurrence

Solve $T(n) = 2T(n/2) + cn^2$, $c > 0$ is constant.



Note that $1 + ½ + ¼ + \ldots < 2$

Total $= \Theta(n^2)$

All the work done at the root

Priority Queue

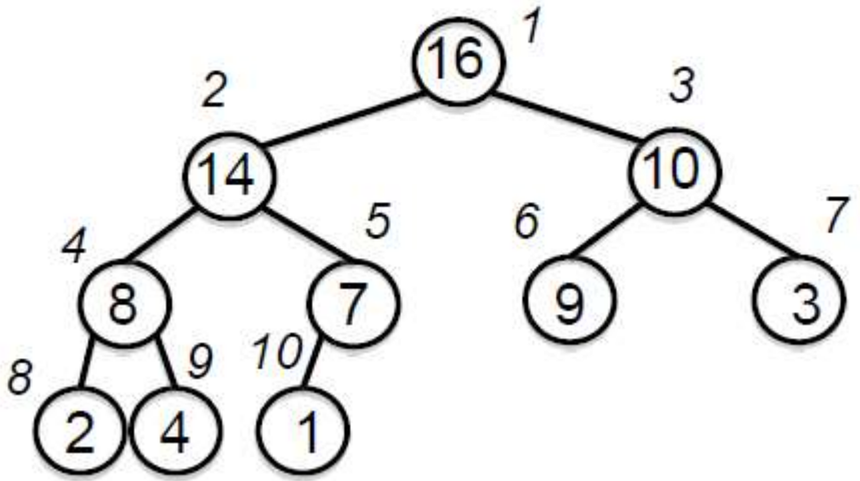
A data structure implementing a set S of elements, each associated with a key, supporting the following operations:

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

Heap

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

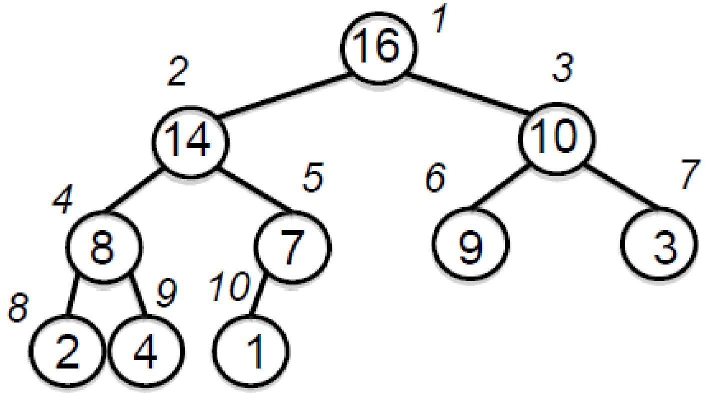
Heap as a Tree

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

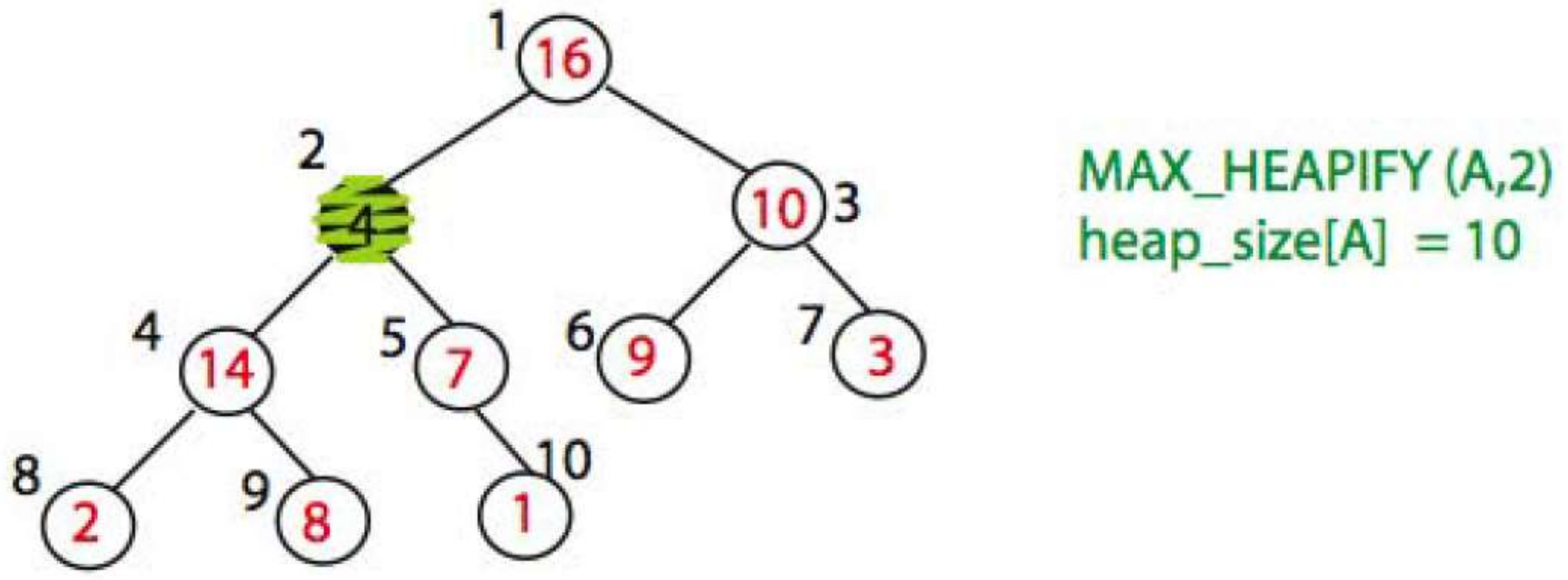
insert, extract_max, heapsort

Max_heapify

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

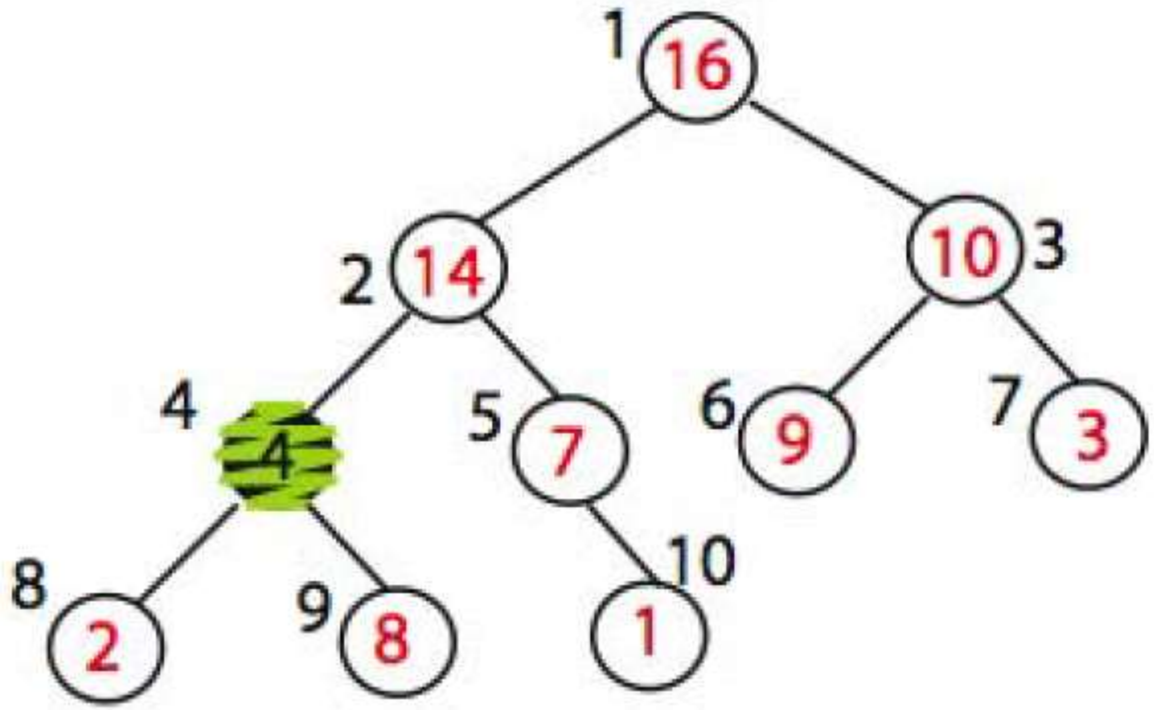
- If element A[*i*] violates the max-heap property, correct violation by "trickling" element A[*i*] down the tree, making the subtree rooted at index *i* a max-heap

Max_heapify (Example)



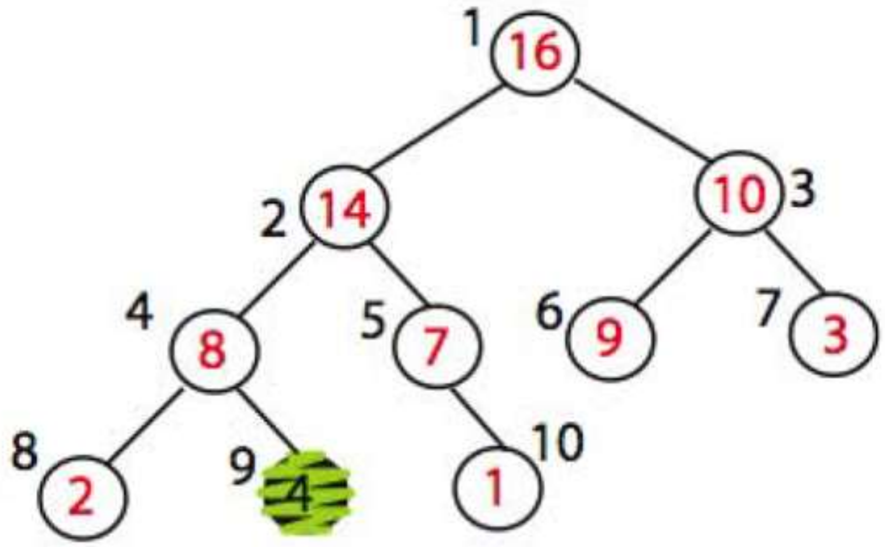
Node 10 is the left child of node 5 but is drawn to the right for convenience

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

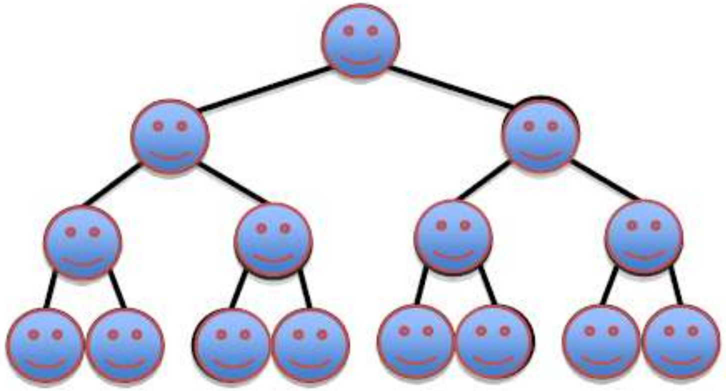
Time= O(log n)

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)

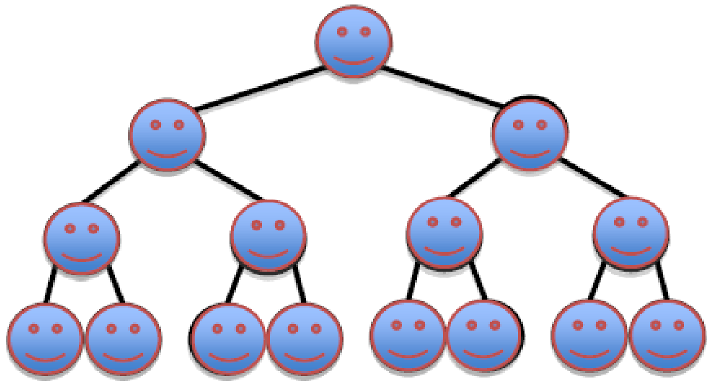
Why start at n/2?

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



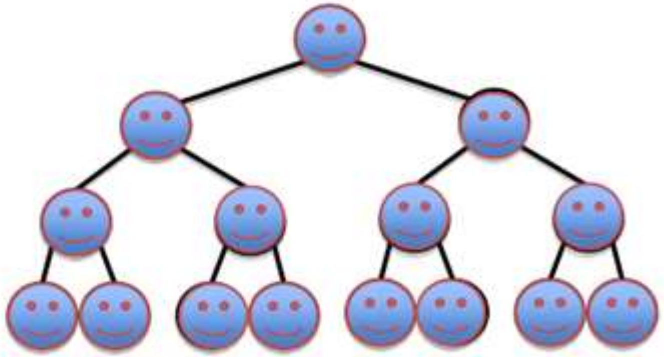
Build_Max_Heap(A):
for i=n/2 downto 1
do Max_Heapify(A, i)

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.

$\lceil n/4 \rceil$
$\lceil n/8 \rceil$

Build_Max_Heap(A) Analysis

Converts $A[1\ldots n]$ 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)

Exercise

- Show that $\left(\frac{1}{2^0}+\frac{2}{2^1}+\frac{3}{2^2}+\ldots+\frac{k+1}{2^k}\right)$ is bounded by a constant.

2. $a_n = \frac{n+1}{2^n}$, $a_0 = \frac{1}{2^0}$

$$S_n = \frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \cdots + \frac{n}{2^{n-1}} + \frac{n+1}{2^n} \quad ①$$

$$\frac{1}{2} S_n = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots + \frac{n}{2^n} + \frac{n+1}{2^{n+1}} \quad ②$$

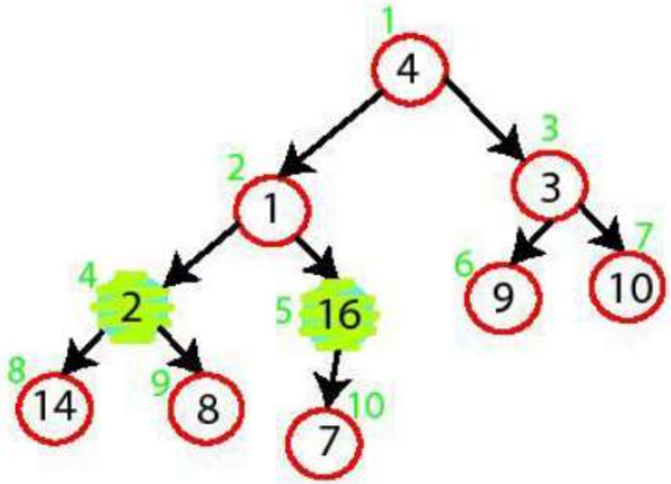
① - ② : $$\frac{1}{2} S_n = \frac{1}{2^0} + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^n} \boxed{- \frac{n+1}{2^{n+1}}}$$

$$= 1 + \frac{\frac{1}{2}\left(1 - \frac{1}{2^n}\right)}{1 - \frac{1}{2}} - \frac{n+1}{2^{n+1}}$$

$$= 2 - \frac{1}{2^n} - \frac{n+1}{2^{n+1}} = 2 - \frac{n+3}{2^{n+1}}$$

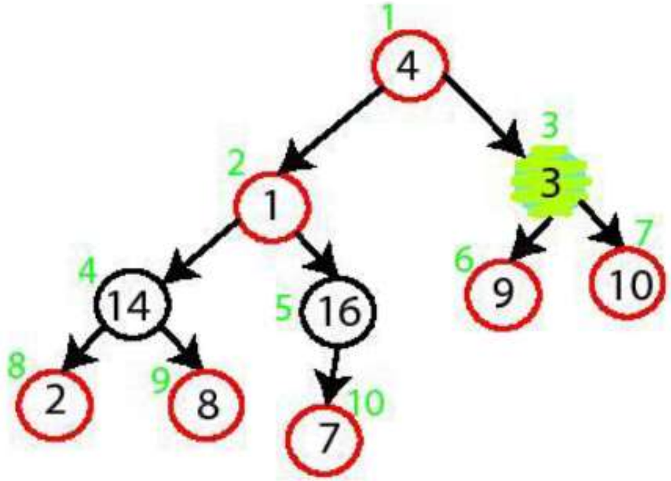
$\Rightarrow S_n = 4 - \frac{n+3}{2^n} < 4 \Rightarrow$ upper bound is 4.

Build-Max-Heap Demo



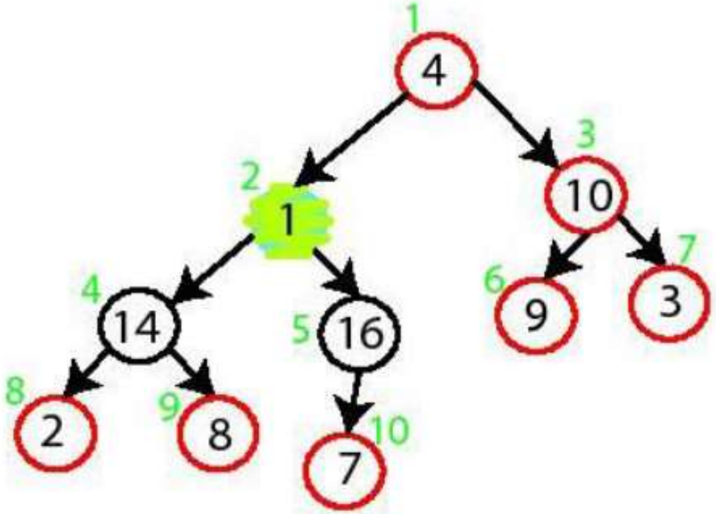
A									
4	1	3	2	16	9	10	14	8	7

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

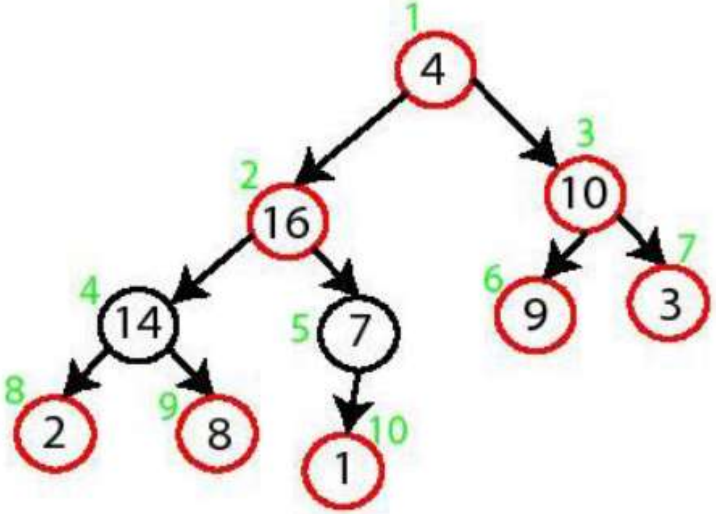


MAX-HEAPIFY (A,3)
Swap A[3] and A[7]

Build-Max-Heap Demo



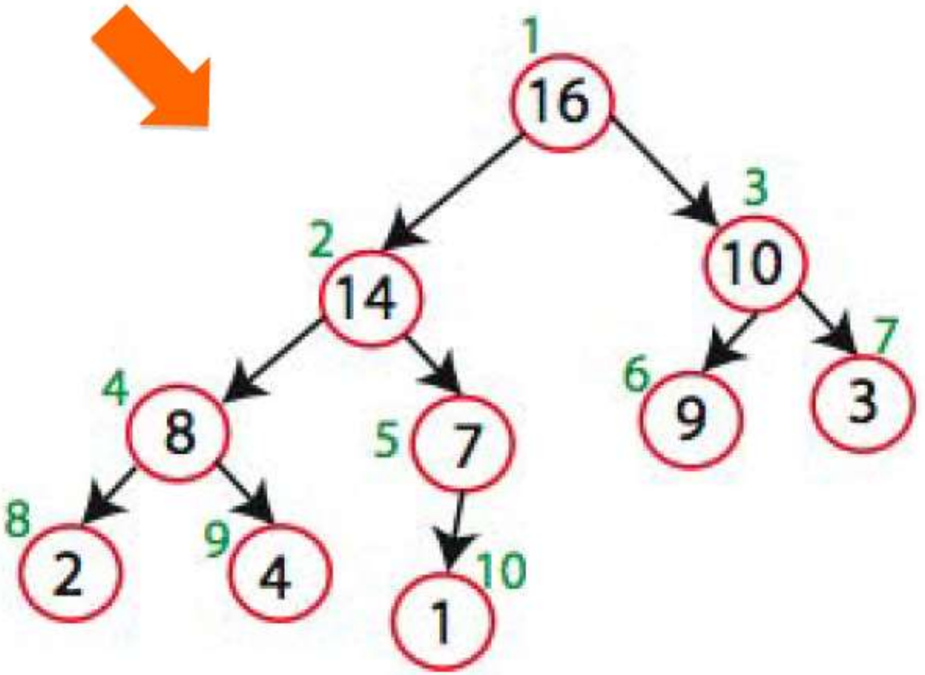
MAX-HEAPIFY (A,2)
Swap A[2] and A[5]
Swap A[5] and A[10]



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;

2. Find maximum element A[1];

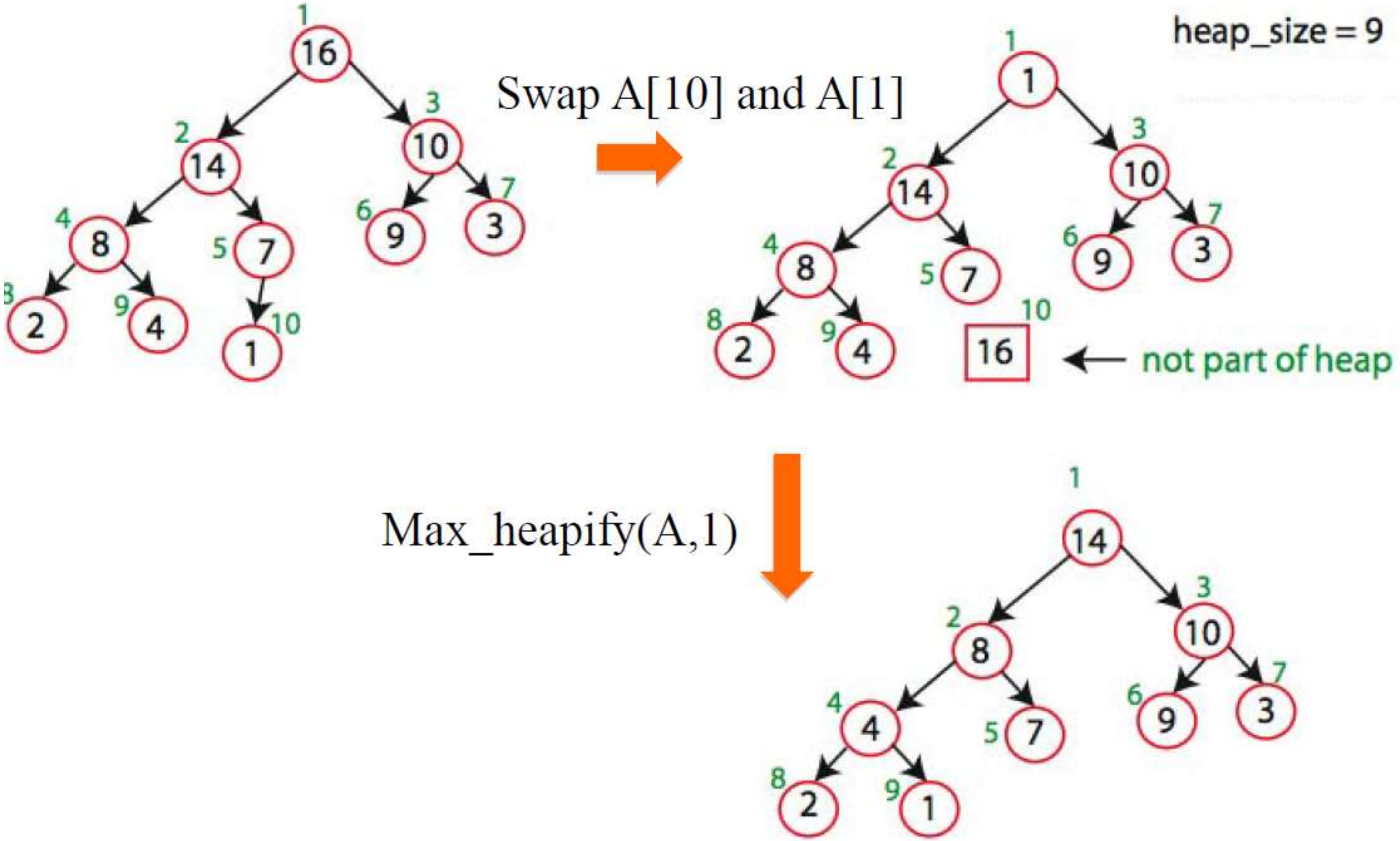
3. Swap elements A[n] and A[1]:
   now max element is at the end of the array!

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

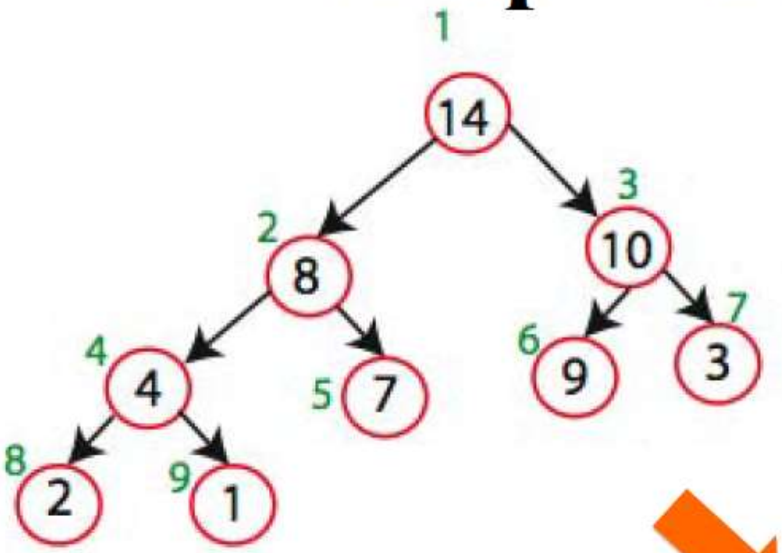
5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.

6. Go to Step 2 unless heap is empty.

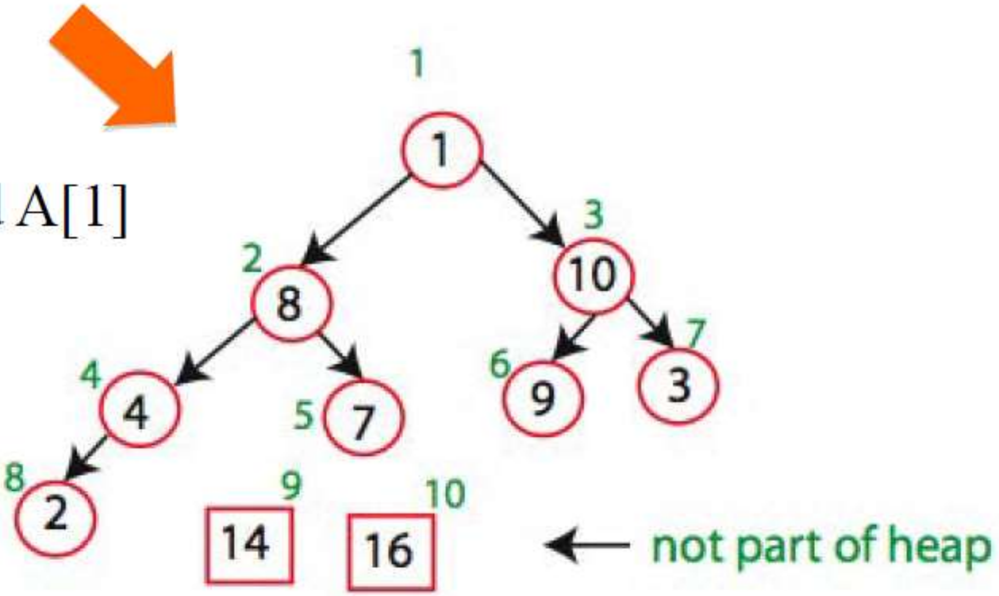
Heap-Sort Demo



Heap-Sort Demo

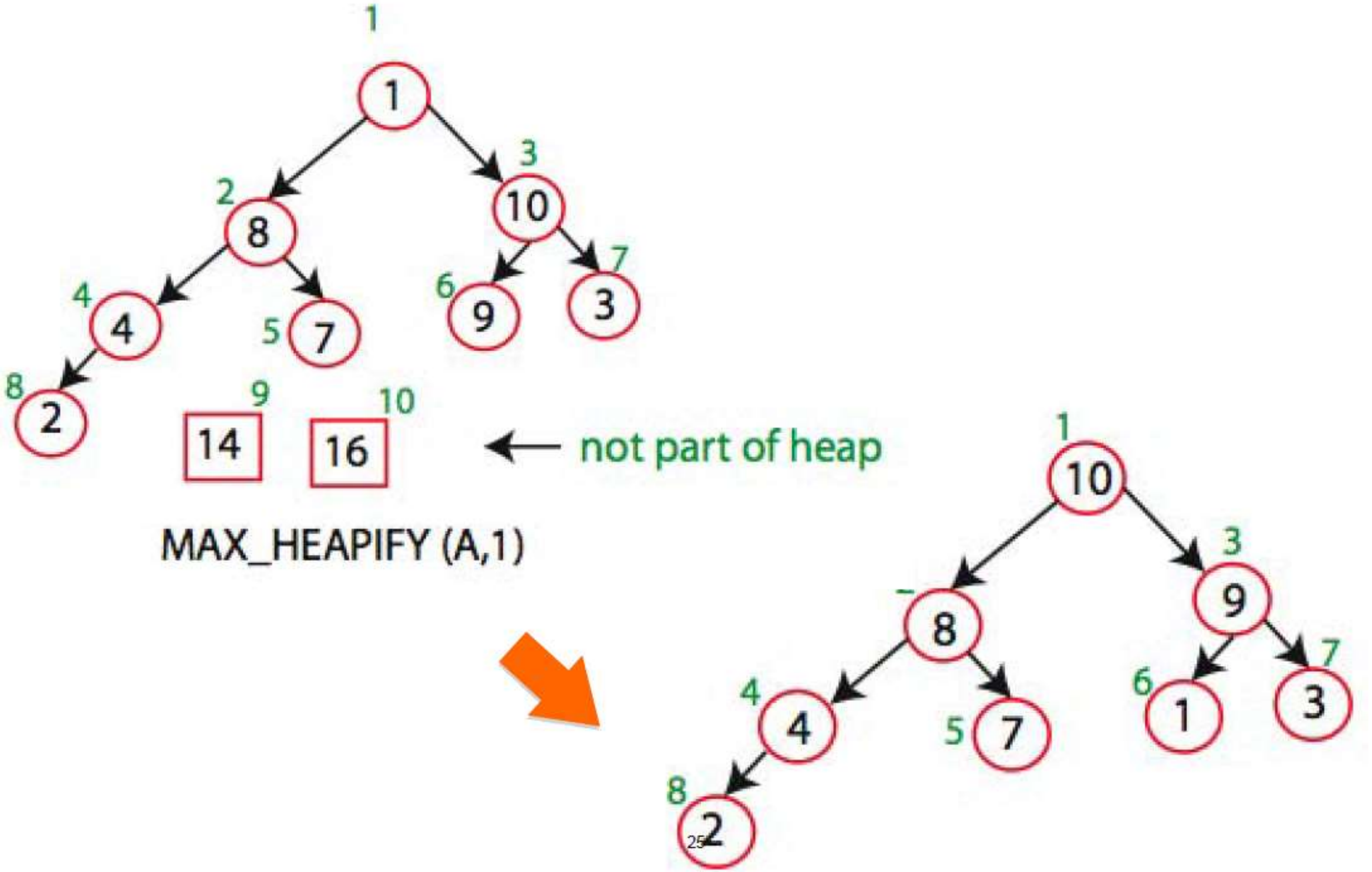


Swap A[9] and A[1]

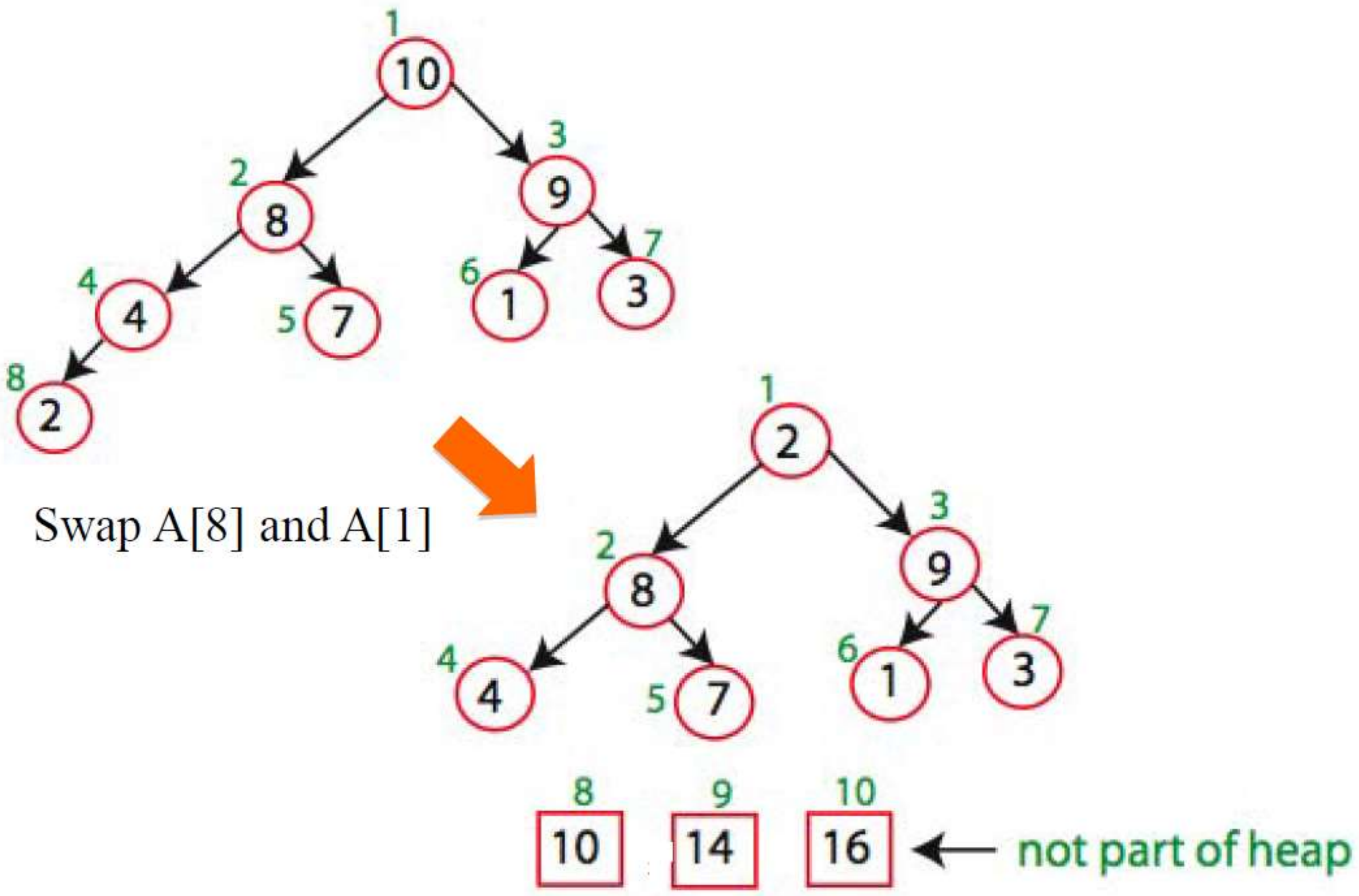


MAX_HEAPIFY (A,1)

Heap-Sort Demo



Heap-Sort Demo



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 $O(n \log n)$

Part II

Median Finding

Given set of n numbers, define $rank(x)$ as number of numbers in the set that are $\leq x$.
Find element of rank $\lfloor \frac{n+1}{2} \rfloor$ (lower median) and $\lceil \frac{n+1}{2} \rceil$ (upper median).

Clearly, sorting works in time $\Theta(n \log n)$.

Can we do better?

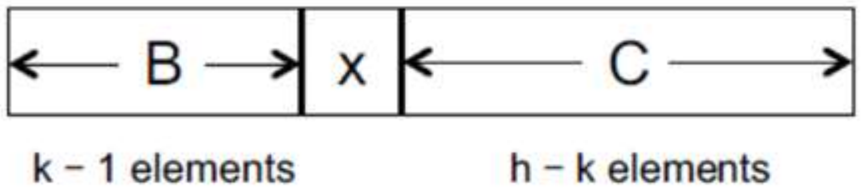
(Hint: for simplification, suppose that *n* numbers are distinct.)

$\textsc{Select}(S, i)$

```
Pick x ∈ S ▷ cleverly
Compute k = rank(x)
B = {y ∈ S | y < x}
C = {y ∈ S | y > x}
if k = i
        return x
else if k > i
        return Select(B, i)
else if k < i
        return Select(C, i − k)
```

$O(n)$?

$T(n/2)$?

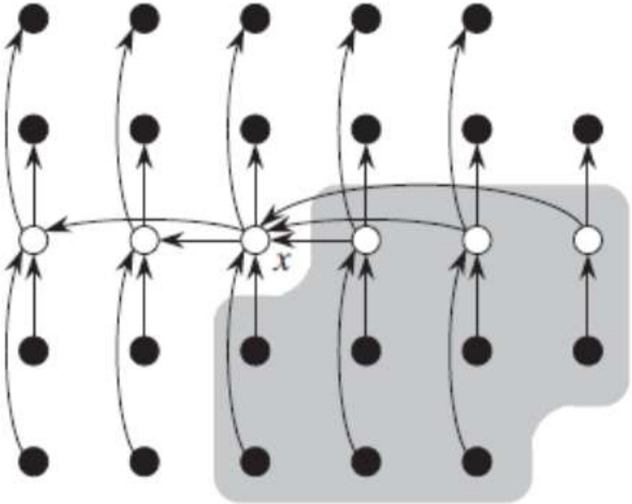


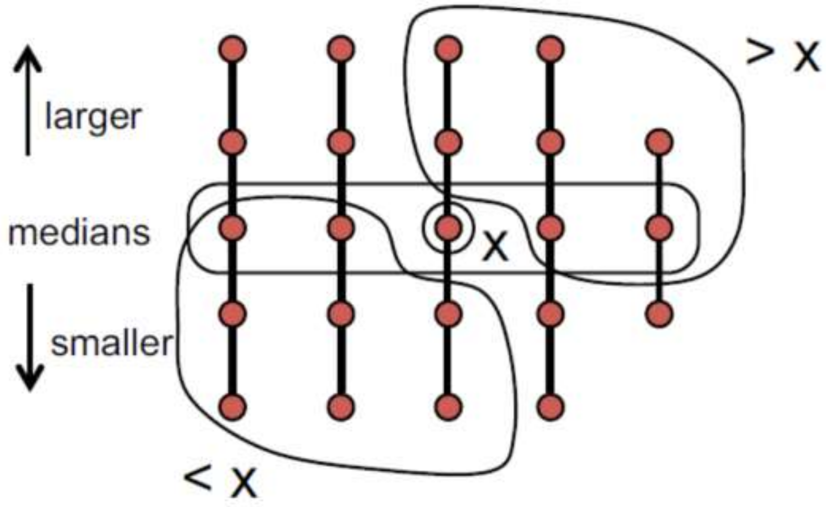
$$T(n) = T(n/2) + O(n) \Rightarrow T(n) = O(n)$$

Picking x Cleverly

Need to pick x so $rank(x)$ is not extreme.

- Arrange S into columns of size 5 ($\lceil \frac{n}{5} \rceil$ cols)
- Sort each column (bigger elements on top) (linear time)
- Find "median of medians" as x





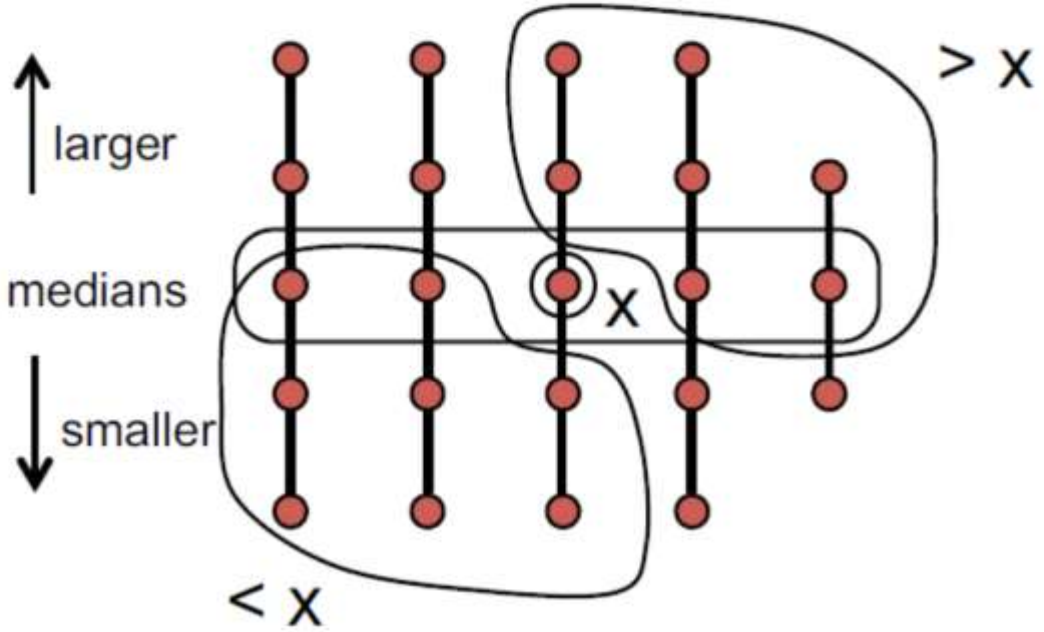
How many elements are guaranteed to be $> x$?

Half of the $\lceil \frac{n}{5} \rceil$ groups contribute at least 3 elements $> x$ except for 1 group with less than 5 elements and 1 group that contains x.

At least $3(\lceil \frac{n}{10} \rceil - 2)$ elements are $> x$, and at least $3(\lceil \frac{n}{10} \rceil - 2)$ elements are $< x$

Recurrence:

$$T(n) = \begin{cases} O(1), & \text{for } n \leq 140 \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 6) + \theta(n), & \text{for } n > 140 \end{cases} \quad (1)$$



Solving the Recurrence

Master theorem does not apply. Intuition $\frac{n}{5} + \frac{7n}{10} < n$.

Prove $T(n) \leq cn$ by induction, for some large enough c.

True for $n \leq 140$ by choosing large c

$$T(n) \leq c\lceil\frac{n}{5}\rceil + c(\frac{7n}{10} + 6) + an \tag{2}$$

$$\leq \frac{cn}{5} + c + \frac{7nc}{10} + 6c + an \tag{3}$$

$$= cn + (-\frac{cn}{10} + 7c + an) \tag{4}$$

If $c \geq \frac{70c}{n} + 10a$, we are done. This is true for $n \geq 140$ and $c \geq 20a$.

The maximum-subarray problem

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8\,..\,11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

A brute-force solution

We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date. A period of n days has $\binom{n}{2}$ such pairs of dates. Since $\binom{n}{2}$ is $\Theta(n^2)$, and the best we can hope for is to evaluate each pair of dates in constant time, this approach would take $\Omega(n^2)$ time. Can we do better?

A solution using divide-and-conquer

Let's think about how we might solve the maximum-subarray problem using the divide-and-conquer technique. Suppose we want to find a maximum subarray of the subarray $A[low \,..\, high]$. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say *mid*, of the subarray, and consider the subarrays $A[low \,..\, mid]$ and $A[mid + 1 \,..\, high]$. As Figure 4.4(a) shows, any contiguous subarray $A[i \,..\, j]$ of $A[low \,..\, high]$ must lie in exactly one of the following places:

- entirely in the subarray $A[low \,..\, mid]$, so that $low \le i \le j \le mid$,
- entirely in the subarray $A[mid + 1 \,..\, high]$, so that $mid < i \le j \le high$, or
- crossing the midpoint, so that $low \le i \le mid < j \le high$.

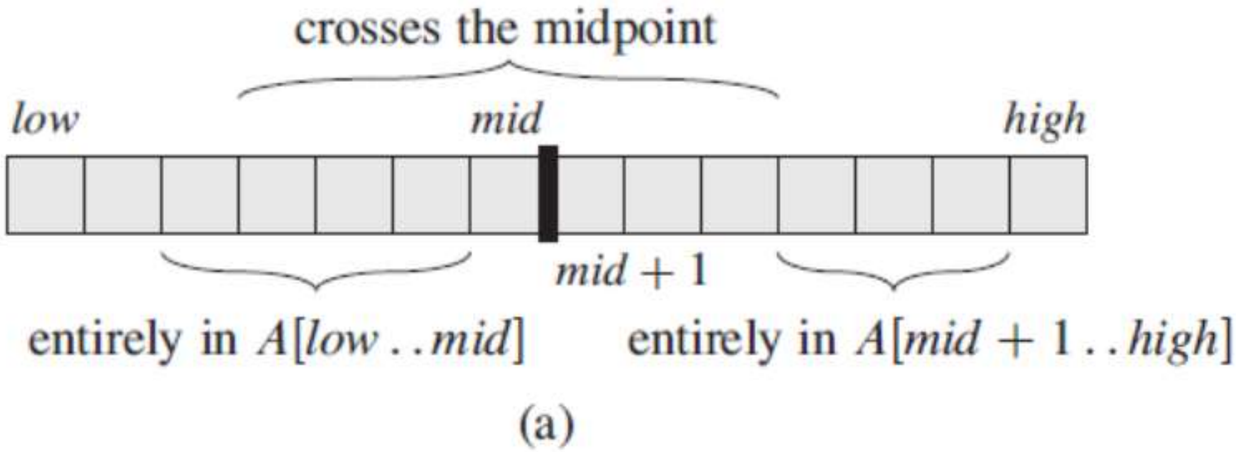


Figure 4.4 **(a)** Possible locations of subarrays of $A[low \,..\, high]$: entirely in $A[low \,..\, mid]$, entirely in $A[mid + 1 \,..\, high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low \,..\, high]$ crossing the midpoint comprises two subarrays $A[i \,..\, mid]$ and $A[mid + 1 \,..\, j]$, where $low \leq i \leq mid$ and $mid < j \leq high$.

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

FIND-MAXIMUM-SUBARRAY(*A*, *low*, *high*)

```
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Can we do better?
(Hint: DP takes
$\theta(n)$.)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$