Lecture 9: Sorting II: Heaps

Lecture Overview

- Priority Queues
- Heaps
- Heapsort

Readings

CLRS 2.1, 2.2, 2.3, 6.1, 6.2, 6.3 and 6.4

Priority Queues

This is an abstract datatype implementing a set S of elements, each associated with a key. Supports 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):	increases the value of element x's key to new value k (assumed to be as large as current value)

Heaps

An implementation of a priority queue. It is an array object, visualized as a nearly complete binary tree.

Heap Property: The key of a node is $\geq$ than the keys of its children; e.g., Figure 1.

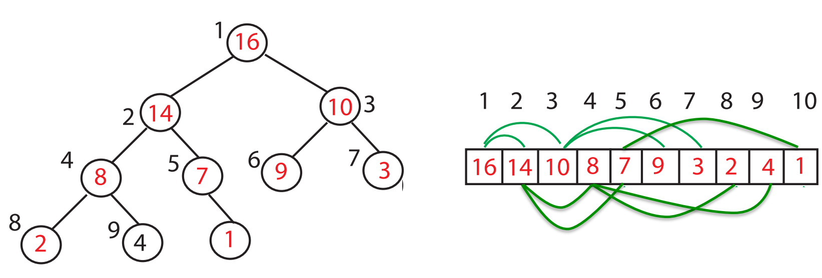

Figure 1: Binary Heap

NOTE: For convenience, the first index in the array is 1.

Visualizing an Array as a Tree

root of tree: first element in the array—corresponding index = 1,
node with index i:
parent(i) = $\lfloor \frac{i}{2} \rfloor$; returns index of node's parent, e.g. parent(5)=2
left(i) = $2i$; returns index of node's left child, e.g. left(4)=8
right(i) = $2i + 1$; returns index of node's right child, e.g. right(4)=9

Note: no pointers required! **Height** of a binary heap $O(\log_2 n)$.

Heap-Size Variable

For flexibility we may only need to consider the first few elements of an array as part of the heap. The variable heap-size denotes the number of items of the array that are part of the heap: $A[1], \ldots, A[\text{heap-size}]$;

Max-Heaps vs Min-Heaps

Max Heaps satisfy the *Max-Heap Property* : for all i, $A[i] \geq \max\{A[\text{left}(i)], A[\text{right}(i)]\}$. If left($i$) or right($i$) is undefined, replace $A[\text{left}(i)]$, respectively $A[\text{right}(i)]$, by $-\infty$. In particular, if node i has no children, the property is trivially satisfied.

Everything we describe applies to the construction and operation of *Min Heaps*, satisfying the *Min-Heap Property* : for all i, $A[i] \leq \min\{A[\text{left}(i)], A[\text{right}(i)]\}$. If left($i$) or right($i$) is undefined, replace $A[\text{left}(i)]$, respectively $A[\text{right}(i)]$, by $+\infty$. In particular, if node i has no children, the property is trivially satisfied.

Operations with Heaps

build_max_heap: produce a max-heap from unordered input array in $O(n)$;
max_heapify: correct a single violation of the heap property at the root of a subtree in $O(\log n)$;
heapsort: sort an array of size n in $O(n \log n)$ via the use of heaps;
insert, extract_max: $O(\lg n)$

Max_Heapify(A,i)

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. See Figure 2; then read the pseudocode below.

$$l \leftarrow \mathsf{left}(i)$$
$$r \leftarrow \mathsf{right}(i)$$

```
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)
```

Example

Build_Max_Heap(A)

$A[1 \cdots n]$ converted to a max_heap *Observation*: Elements $A[\lfloor n/2 \rfloor + 1 \cdots n]$ are all leaves of the tree (**why?** $2i > n$, for $i > \lfloor n/2 \rfloor + 1$).

```
                  Build_Max_Heap(A):
                       heap_size(A) = length(A)
O(n) times        for i ← ⌊ length[A]/2⌋ downto 1
O(log n) time          do Max_Heapify(A, i)
     O(n log n) overall
```

See Figure 3 for an example.

NOTE: The trivial analysis of the algorithm noted above, shows that the running time is $O(n \log n)$. Observe, however, that Max_Heapify only takes $O(1)$ time for the nodes that are one level above the leaves, and in general $O(\ell)$ for the nodes that are ℓ levels above the leaves. Using this observation, it can be shown that the overall time for Build_Max_Heap(A) is $O(n)$.

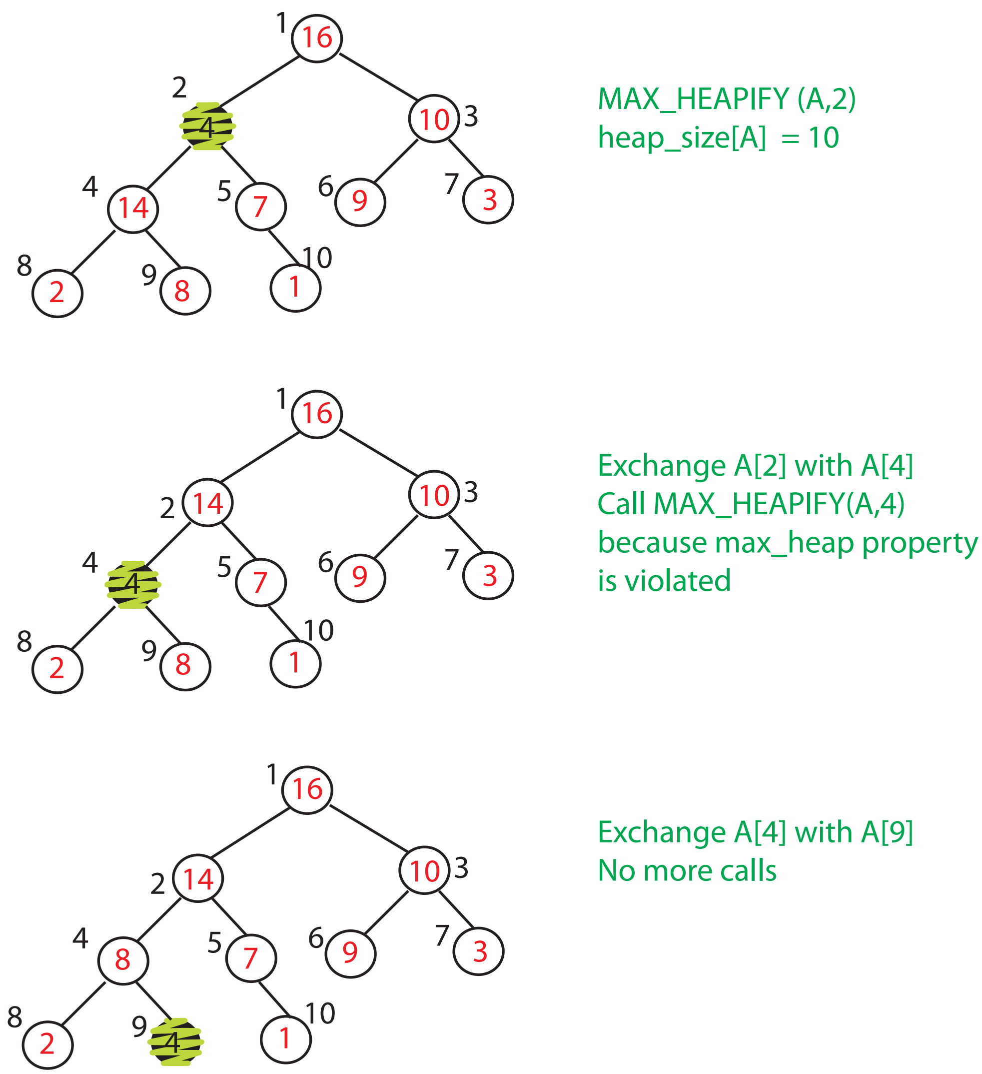

Figure 2: MAX_HEAPIFY Example

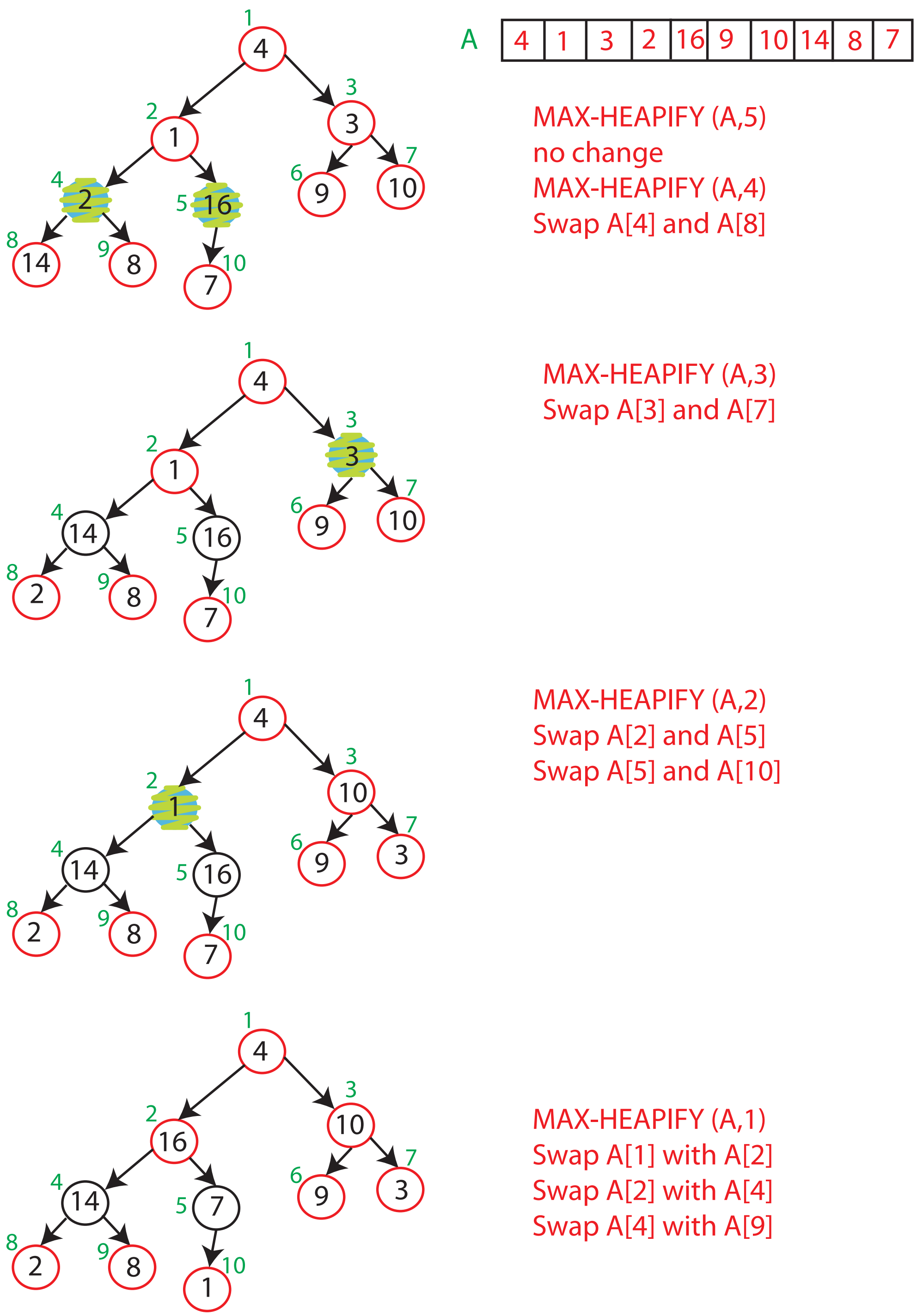

Figure 3: Example: Building Heaps

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 right position;
- Discard node n from heap (decrement heap-size variable);
- New root could violate max_heap property, but children remain max_heaps. Run max_heapify to fix this;

Heap Sort Algorithm

$O(n)$	Build_Max_Heap(A):
n times	for i =length[A] downto 2
	do exchange $A[1] \longleftrightarrow A[i]$
	heap_size[A] = heap_size[A] $- 1$
$O(\log n)$	MAX_HEAPIFY$(A, 1)$
	$O(n \log n)$ overall

See Figure 4 for an illustration.

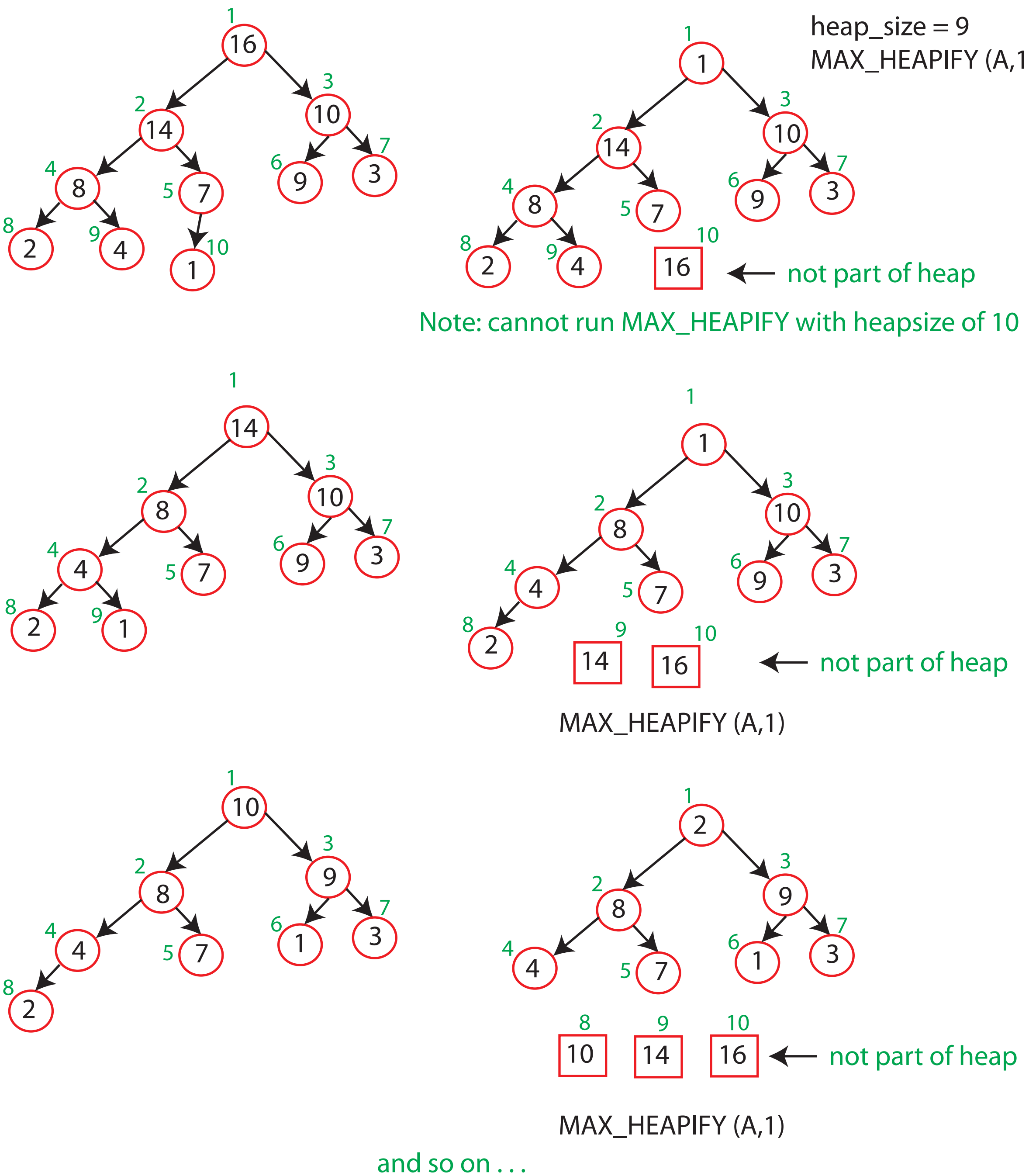

Figure 4: Illustration: Heap Sort Algorithm