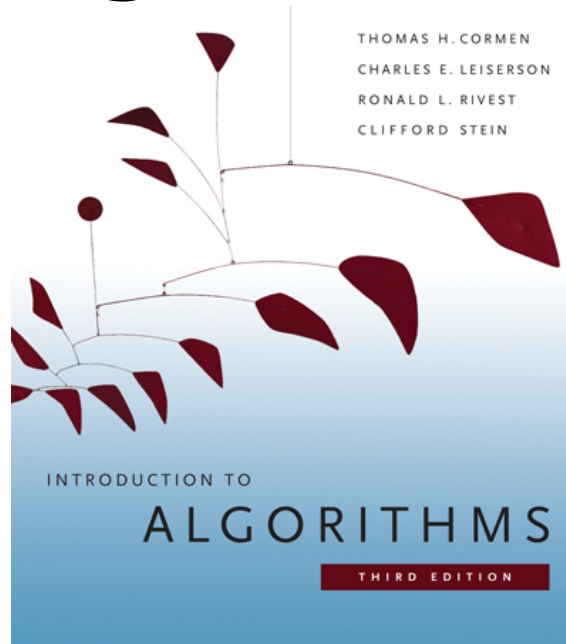


6.006- *Introduction to Algorithms*



Lecture 9

Prof. Constantinos Daskalakis

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

Last time:

- Mergesort for sorting n elements in $O(n \log n)$

This time:



vs



Lecture Overview

- Priority Queue
 - Heaps
 - Heapsort: a new $O(n \log n)$ sorting algorithm

Priority Queue

Any data structure storing a set S of elements, each associated with a key, which supports the following operations:

$\text{insert}(S, x)$: insert element x into set S

$\text{max}(S)$: return element of S with largest key

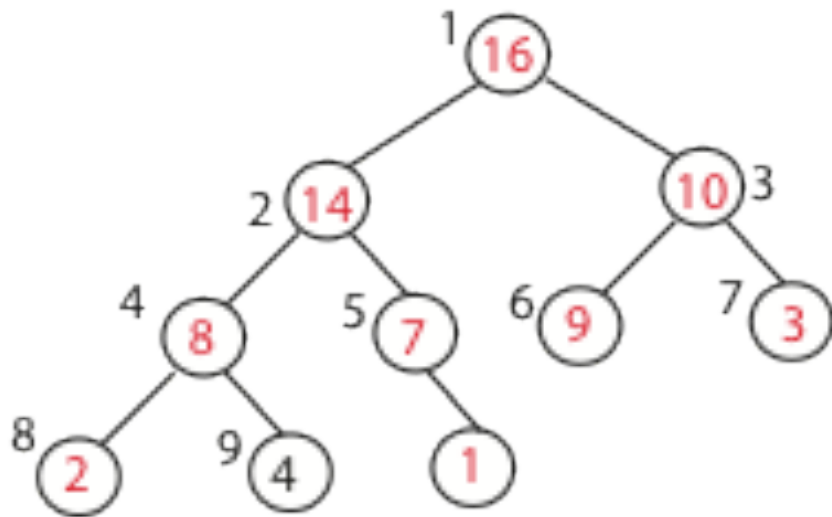
$\text{extract_max}(S)$: return element of S with largest key and remove it from S

$\text{increase_key}(S, x, k)$: change the key-value of element x to k , if k is larger than current value

Heap

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

Max-Heap Property: The key of a node is \geq than the keys of its children.



Visualizing an Array as a Tree

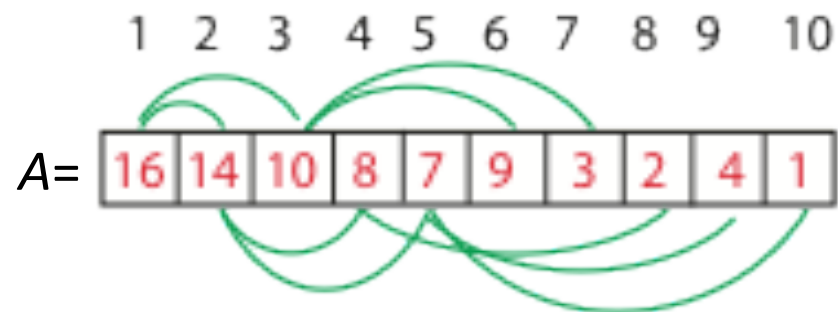
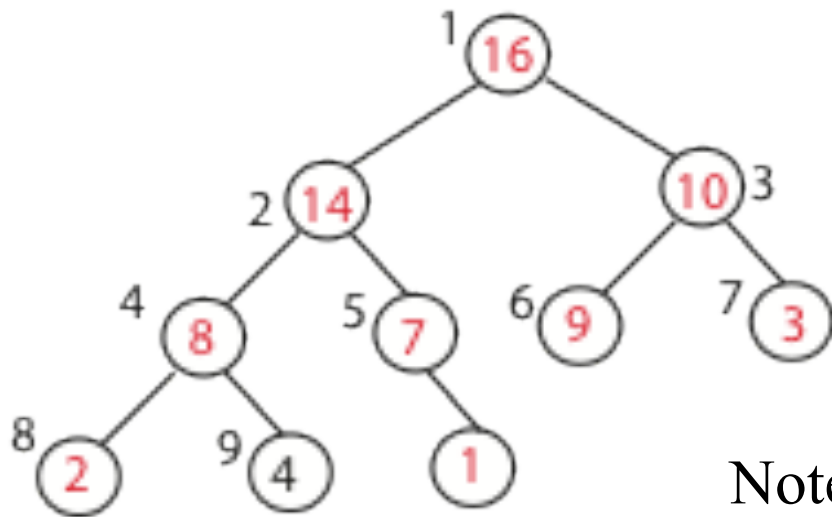
Root of tree: first element in the array, corresponding to index = 1

If a node's index is i then:

$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$; returns index of node's parent, e.g. $\text{parent}(5)=2$

$\text{left}(i) = 2i$; returns index of node's left child, e.g. $\text{left}(4)=8$

$\text{right}(i) = 2i + 1$; returns index of node's right child, e.g. $\text{right}(4)=9$



Note: No pointers required!

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 what prefix of the array is part of the heap:

$$A[1], \dots, A[\text{heap-size}];$$

Operations with Heaps

- *Max_Heapify* (A, i)

Correct a single violation of the heap property occurring at the root i of an otherwise perfect subtree...

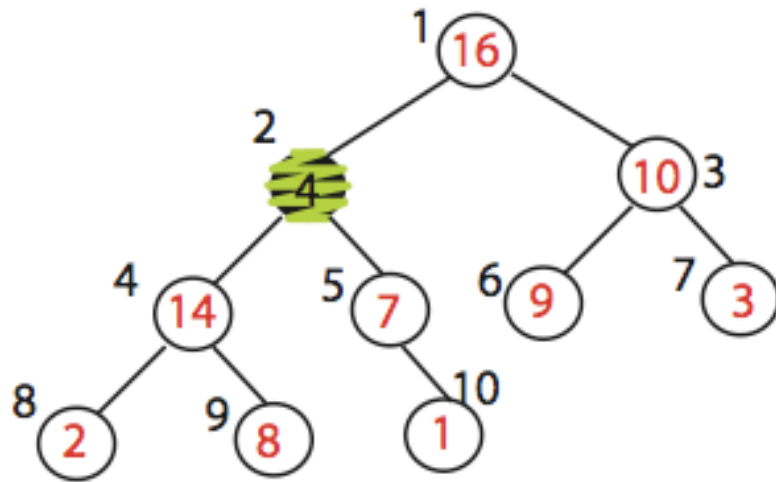
Setting: Assume that the trees rooted at $\text{left}(i)$ and $\text{right}(i)$ are max-heaps, but element $A[i]$ violates the max-heap property;

i.e. $A[i]$ is smaller than at least one of $A[\text{left}(i)]$ or $A[\text{right}(i)]$.

Goal: fix the subtree rooted at i .

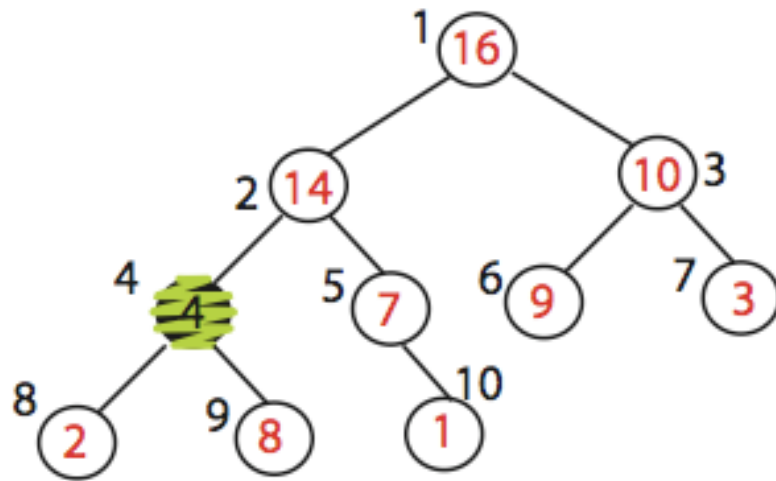
How? Trickle element $A[i]$ down the tree to its right place.

Max_Heapify (Example)



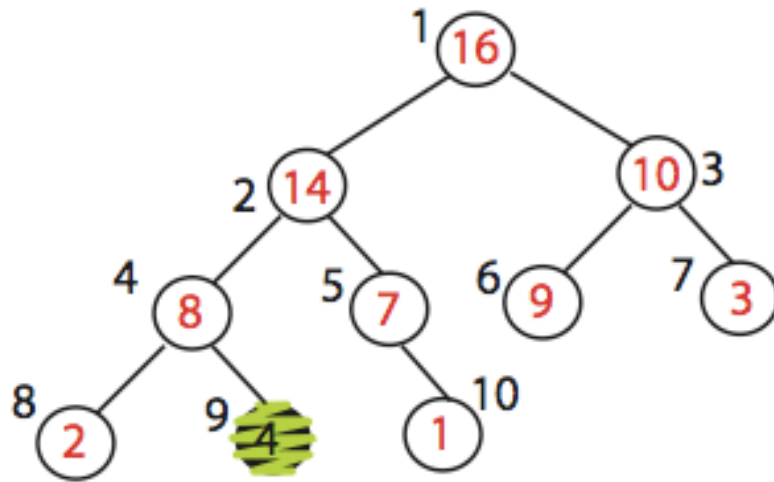
MAX_HEAPIFY (A,2)
heap_size[A] = 10

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

Max_Heapify (Pseudocode)

Max_heapify (A, i)

Find the index of the largest element among $A[i]$, $A[\text{left}(i)]$ and $A[\text{right}(i)]$

```
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 this index is different than i, exchange $A[i]$ with largest element; then recurse on subtree

```
if largest ≠ i
    then exchange A[i] and A[largest]
    MAX_HEAPIFY(A, largest)
```

If $A[i]$ is smaller than both $A[\text{left}(i)]$ and $A[\text{right}(i)]$ why do I insist on swapping with largest and not with any one of them, arbitrarily?

Operations with Heaps

- *Max_Heapify* (A, i)

Correct a single violation of the heap property occurring at the root i of an otherwise perfect subtree.

Time $O(\log n)$.

- *Build_Max_Heap* (A)

Produce a max-heap from an unordered array A .

Build_Max_Heap(A)

Convert $A[1 \dots n]$ to a max heap.

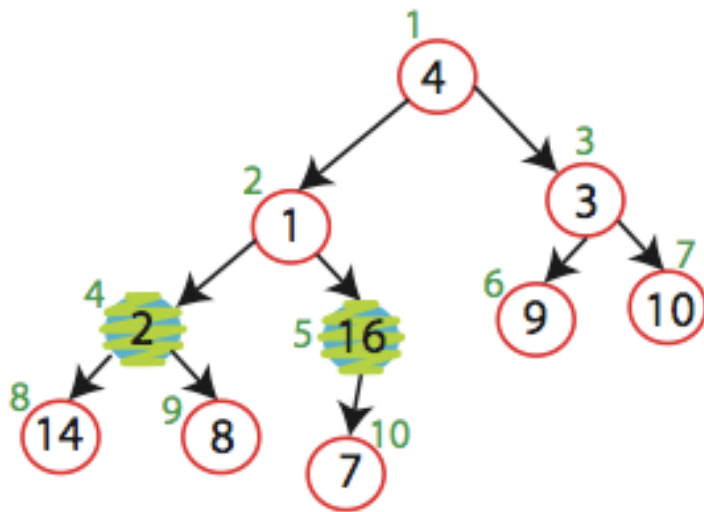
Observation: Elements $A[\lfloor n/2 \rfloor + 1 \dots n]$ are leaves of the tree

because $2i > n$, for all $i \geq \lfloor n/2 \rfloor + 1$

so heap property may only be violated at nodes $1 \dots \lfloor n/2 \rfloor$ of the tree

```
Build_Max_Heap(A):  
    heap_size(A) = length(A)  
    for  $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$  downto 1  
        do Max_Heapify(A, i)
```

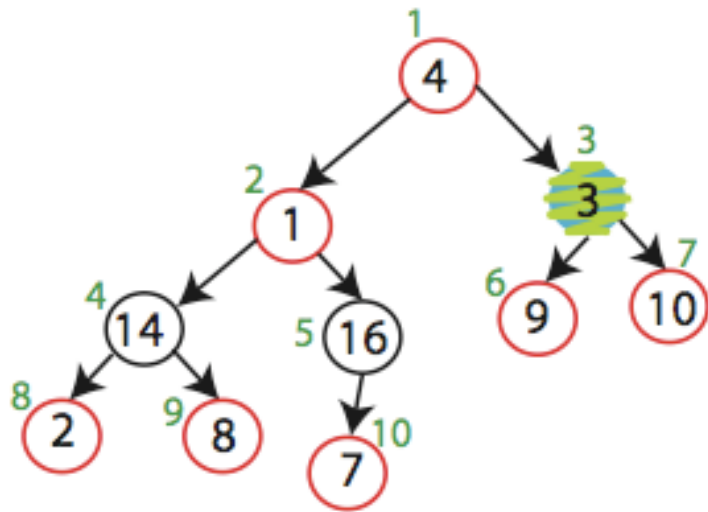
Build_Max_Heap (Example Execution)



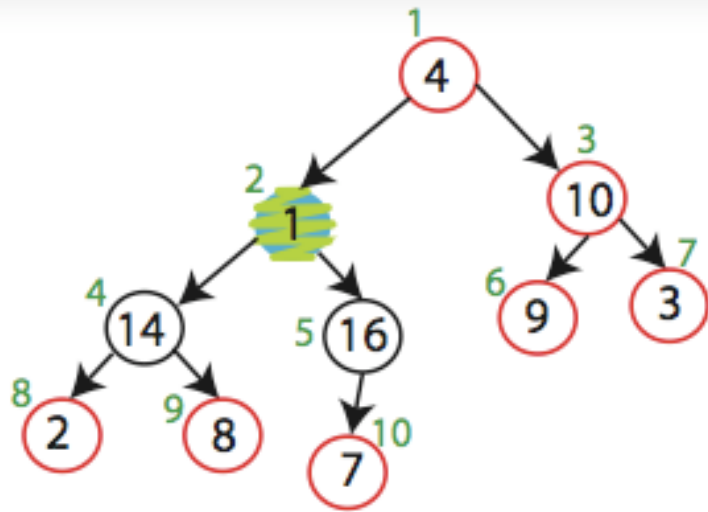
A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

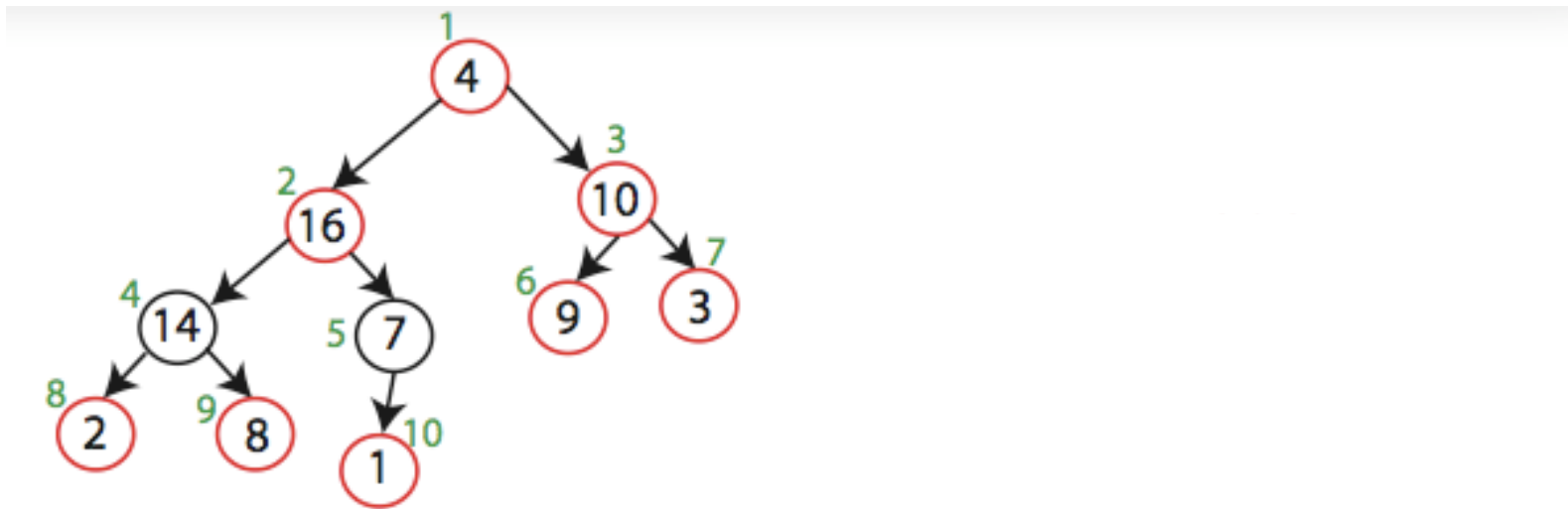
Build_Max_Heap (Example Execution)




Build_Max_Heap (Example Execution)



Build_Max_Heap (Example Execution)



Running Time: $O(n \log n)$, since I need to Heapify $O(n)$ times.

Observe, however, that Heapify only pays $O(1)$ 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.  $O(n)$ time overall!

Operations with Heaps

- *Max_Heapify* (A, i)

- Correct a single violation of the heap property occurring at the root i of an otherwise perfect subtree.
- Time $O(\log n)$.

- *Build_Max_Heap* (A)

- Produce a max-heap from an unordered array A .
- Time $O(n)$

- *Heapsort* (A)

- Sort an array A using heaps.

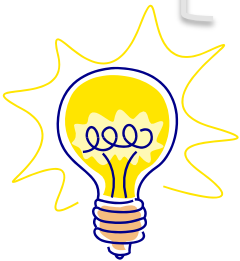
The Naïve Algorithm...

Sorting Strategy:

Find largest element of array, place it in last position; then find the largest among the remaining elements, and place it next to the largest, etc...

In notation:

- $O(n^2)$ {
1. last_element = n;
 2. Find maximum element $A[i]$ of array $A[1 \dots \text{last_element}]$;
 3. Swap $A[i]$ and $A[\text{last_element}]$;
 4. last_element = last_element - 1;
 5. Go to step 2



We have a fast data structure for step 2!
(which is also the most costly)

Heapsort

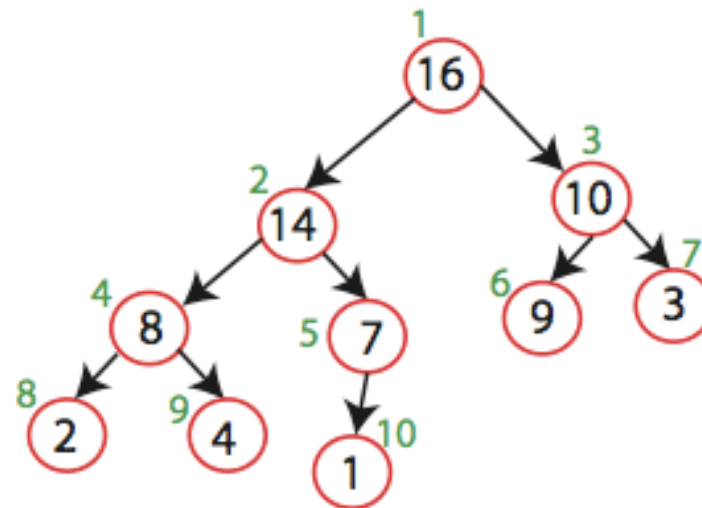
Sorting Strategy:

1. Build Max Heap from unordered array;

Heapsort

A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

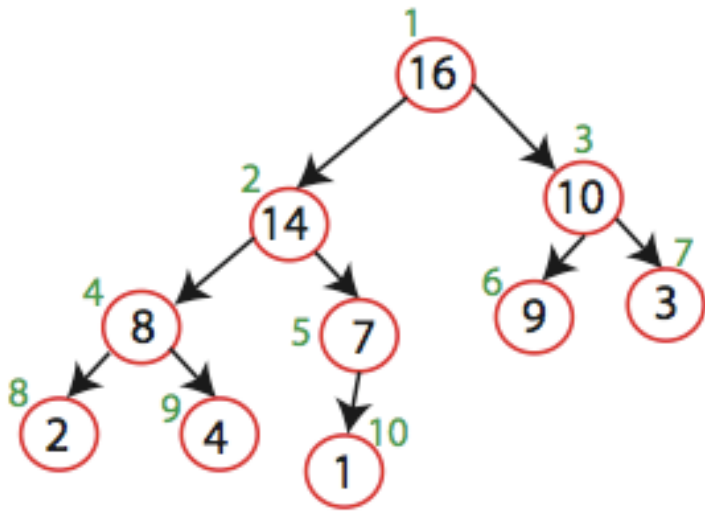


Heapsort

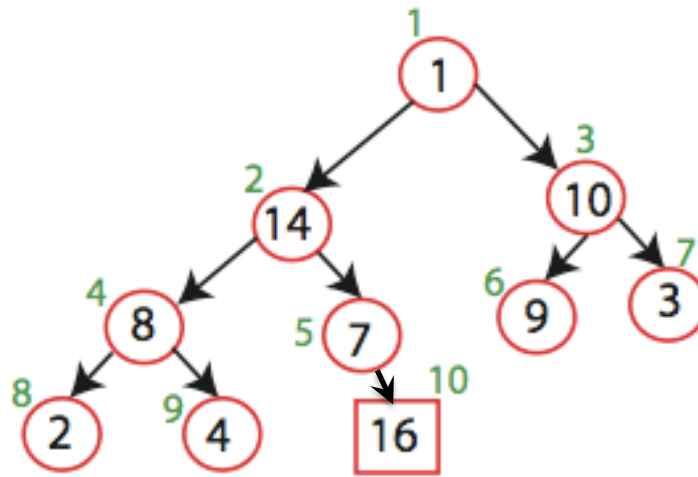
Sorting Strategy:

1. Build Max Heap from unordered array;
2. Find maximum element; this is $A[1]$;
3. Swap elements $A[n]$ and $A[1]$:
now max element is at the end of the array!

Heapsort



Swap elements $A[10]$ and $A[1]$

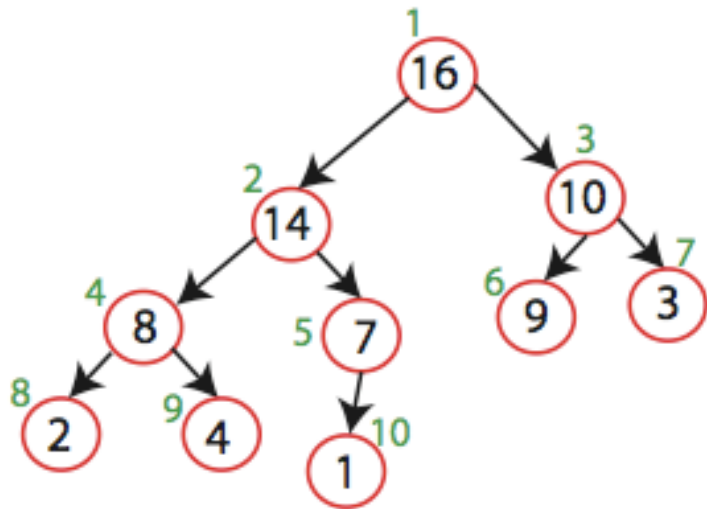


Heapsort

Sorting Strategy:

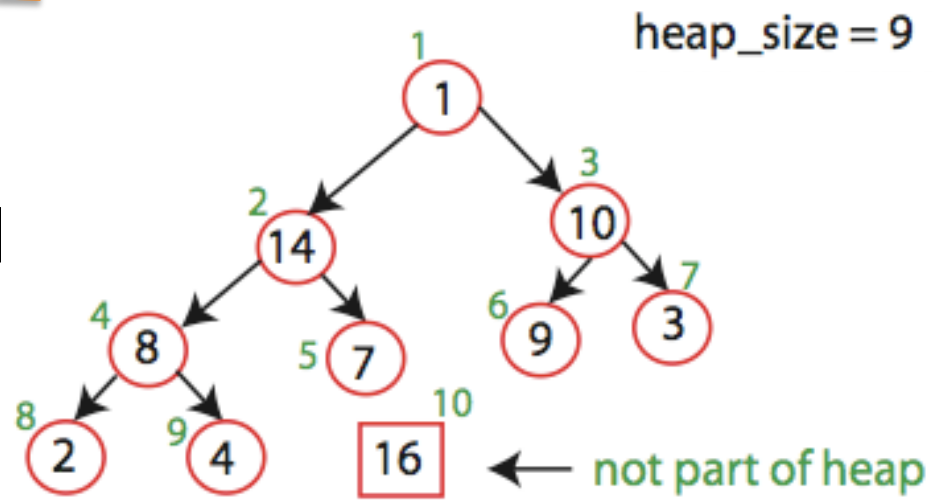
1. Build Max Heap from unordered array;
2. Find maximum element $A[1]$;
3. Swap elements $A[n]$ and $A[1]$:
now max element is at the end of the array!
4. Discard node n from heap
(by decrementing heap-size variable)

Heapsort



Swap elements $A[10]$ and $A[1]$

$\text{heap_size} = \text{heap_size} - 1$

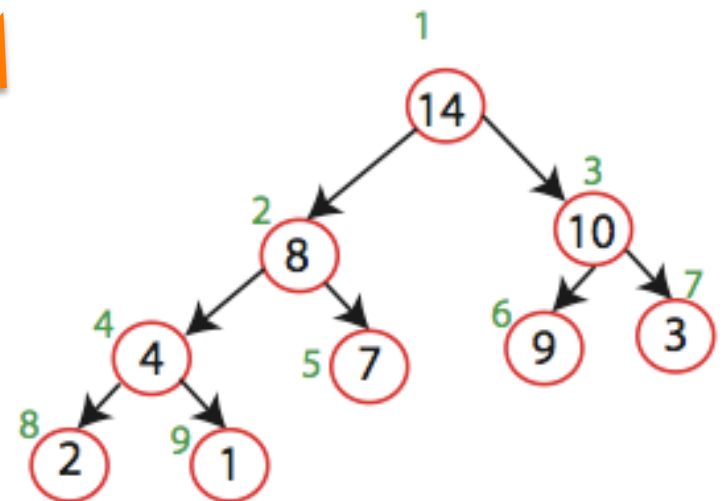
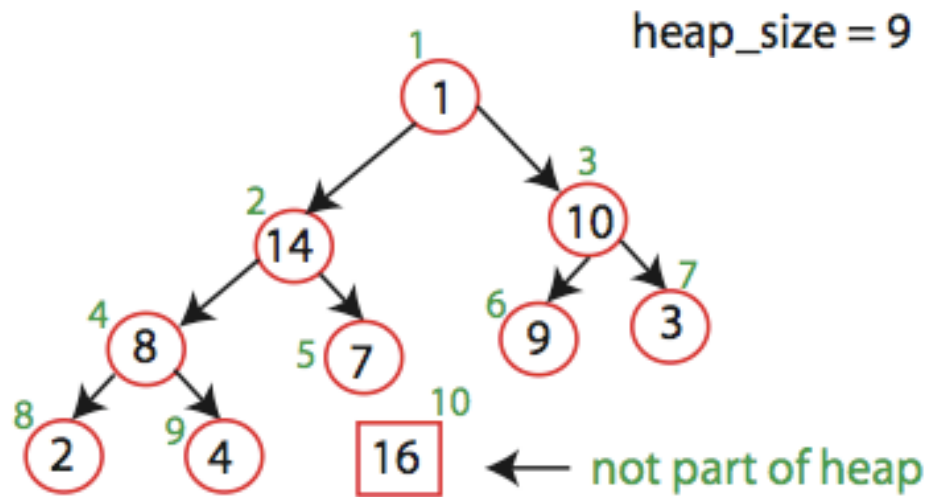


Heapsort

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.

Heapsort

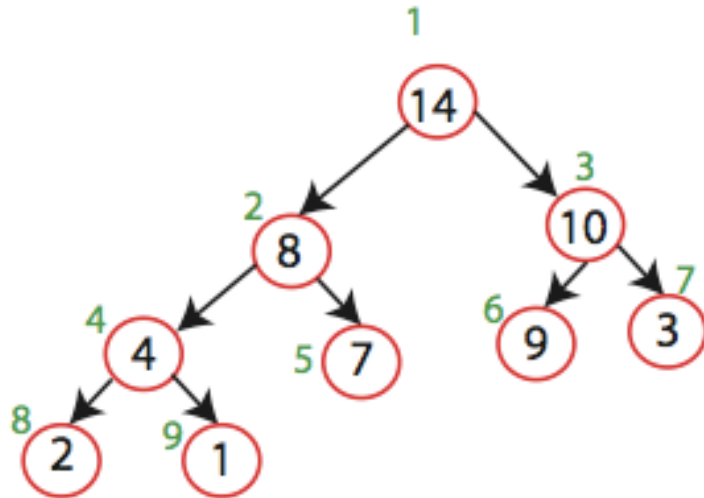


Heapsort

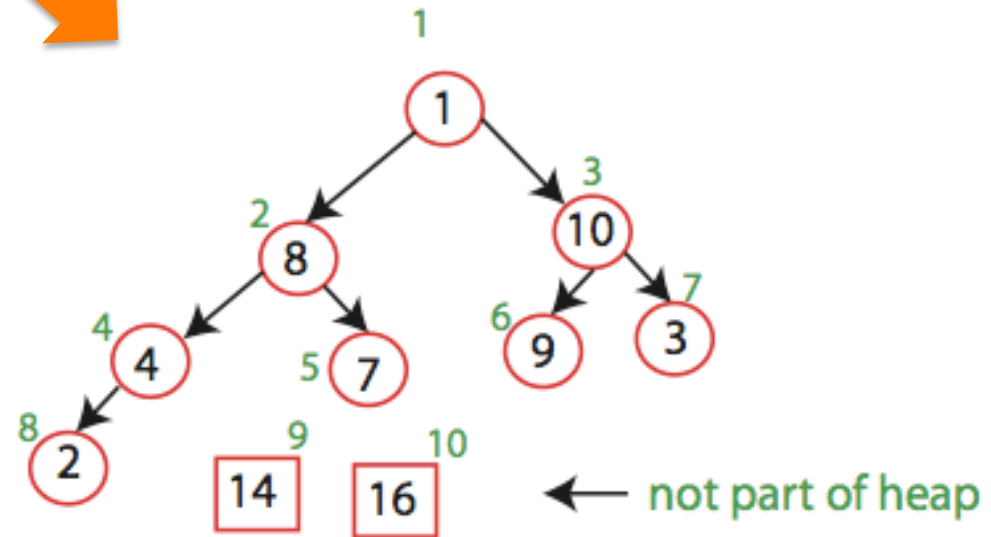
Sorting Strategy:

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.

Heapsort

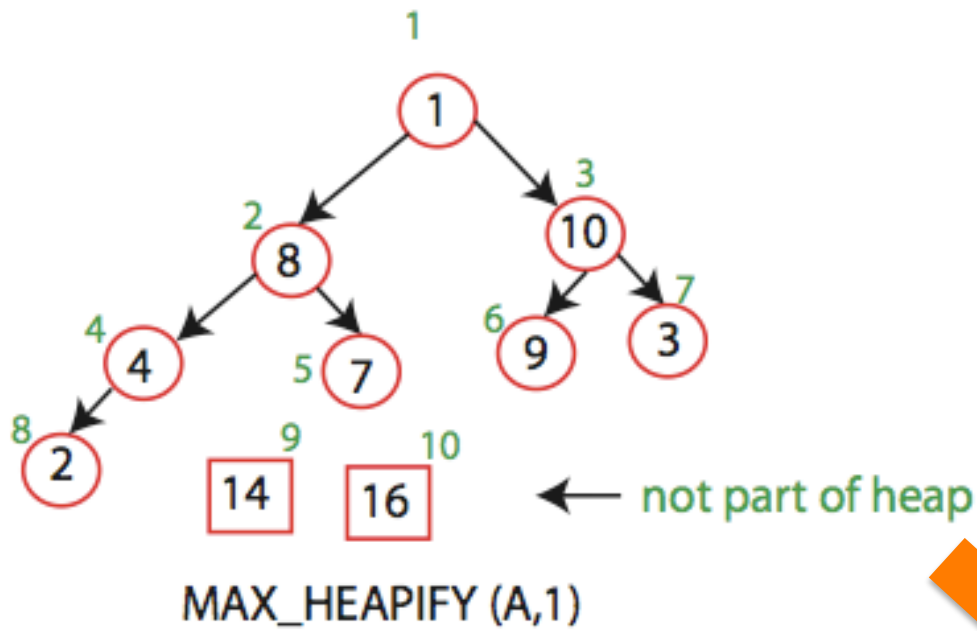


Swap elements A[9] and A[1]

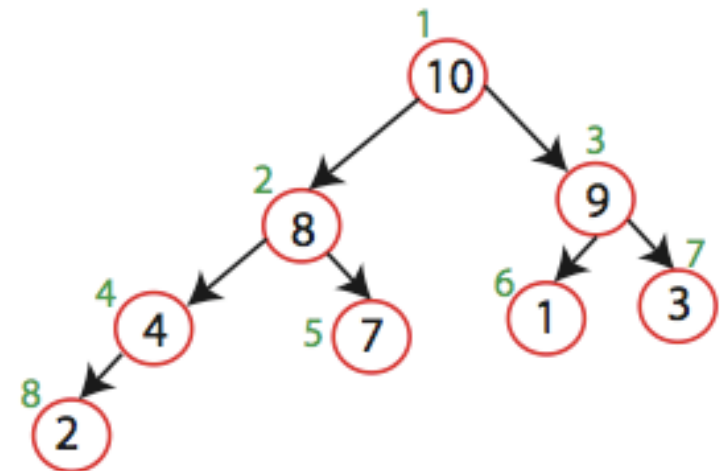


MAX_HEAPIFY (A,1)

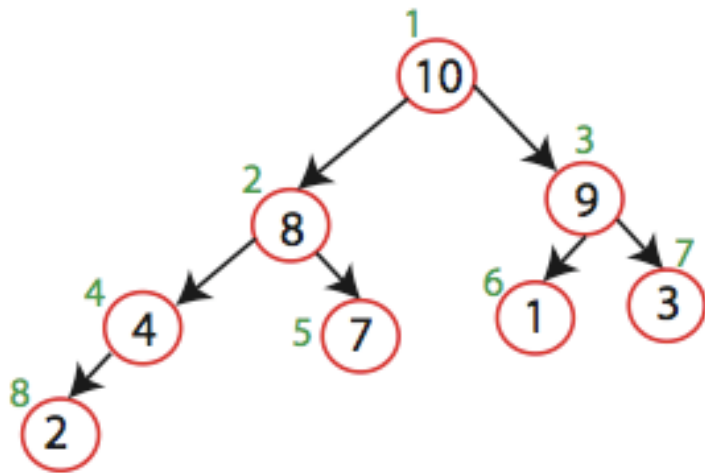
Heapsort



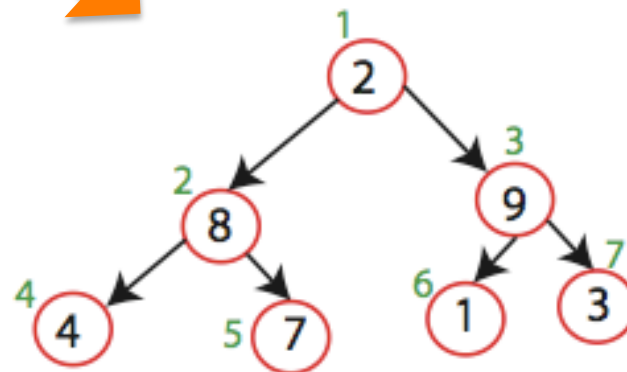
Max_Heapify(A,1)



Heapsort



Swap elements A[8] and A[1]



and so on...

Heapsort Running Time

- $O(n)$ to build heap
- followed by n iterations:
 - in each iteration a swap and a heapify is made;
so $O(\log n)$ time spent in each iteration.

Overall $O(n \log n)$

Operations with Heaps Summary



Max_Heapify :

correct a single violation of the heap property occurring at the root of a subtree in $O(\log n)$;



Build_Max_Heap :

produce a max-heap from an unordered array in $O(n)$;



Heapsort :

sort an array of size n in $O(n \log n)$ using heaps

Insert, Extract_Max ? $O(\log n)$

Heapsort in a Nutshell



Max-Heaps vs Min-Heaps

Max Heaps satisfy the **Max-Heap Property** :

for all i , $A[i] \geq \max\{ A[\text{left}(i)] , A[\text{right}(i)] \}$

Min Heaps satisfy the **Min-Heap Property** :

for all i , $A[i] \leq \min\{ A[\text{left}(i)] , A[\text{right}(i)] \}$

Max-Heaps vs Min-Heaps

Max Heaps satisfy the **Max-Heap Property**:

for all i , $A[i] \geq \max \{ A[\text{left}(i)] , A[\text{right}(i)] \}$

OK, If $\text{left}(i)$ or $\text{right}(i)$ is undefined, replace $A[\text{left}(i)]$, respectively $A[\text{right}(i)]$, by $-\infty$ in the above condition. In particular, if node i has no children, the property is trivially satisfied.

Min Heaps satisfy the **Min-Heap Property**:

for all i , $A[i] \leq \min \{ A[\text{left}(i)] , A[\text{right}(i)] \}$

If $\text{left}(i)$ or $\text{right}(i)$ is undefined, replace $A[\text{left}(i)]$, respectively $A[\text{right}(i)]$, by $+\infty$ in the above condition. In particular, if node i has no children, the property is trivially satisfied.