L02.02
Heap

50.004 Introduction to Algorithm
Gemma Roig
(slides adapted from Dr. Simon LUI)
ISTD, SUTD

Revision – master theorem
Heap

Revision – master theorem

Master theorem

Consider $T(n) = aT(n/b) + f(n)$

$T(n) = 2T(n/2) + \log n$ ← So, a=2, b=2, f(n) = logn

Let's check master theorem rule 1:
f(n) = logn, which is $O(n^{\log_2 2-\varepsilon})=O(n^{1-\varepsilon})$, so rule 1 is true
so, $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$

Let's check master theorem rule 2:
f(n) = logn, which is not $\Theta(n^{\log_2 2})=\Theta(n)$, so rule 2 is wrong

Let's check master theorem rule 3:
f(n) = logn, which is not $\Omega(n^{\log_2 2+\varepsilon})=\Omega(n^{1+\varepsilon})$, so rule 3 is wrong

Master theorem

Consider $T(n) = aT(n/b) + f(n)$

$T(n) = 2T(n/2) + n$ ← So, a=2, b=2, f(n) = n

Let's check master theorem rule 1:
f(n) = n, which is not $O(n^{\log_2 2-\varepsilon})=O(n^{1-\varepsilon})$, so rule 1 is wrong

Let's check master theorem rule 2:
f(n) = n, which is $\Theta(n^{\log_2 2})=\Theta(n)$, so rule 1 is TRUE
So, $T(n) = \Theta(n^{\log_2 2}\log n) = \Theta(n\log n)$

Let's check master theorem rule 3:
f(n) = n, which is not $\Omega(n^{\log_2 2+\varepsilon})=\Omega(n^{1+\varepsilon})$, so rule 1 is wrong

Master theorem

Consider $T(n) = aT(n/b) + f(n)$

$T(n) = 2T(n/2) + n^2$ ← So, a=2, b=2, f(n) = n^2

Let's check master theorem rule 1:
f(n) = n^2, which is not $O(n^{\log_2 2-\varepsilon})=O(n^{1-\varepsilon})$, so rule 1 is wrong

Let's check master theorem rule 2:
f(n) = n^2, which is not $\Theta(n^{\log_2 2})=\Theta(n)$, so rule 2 is wrong

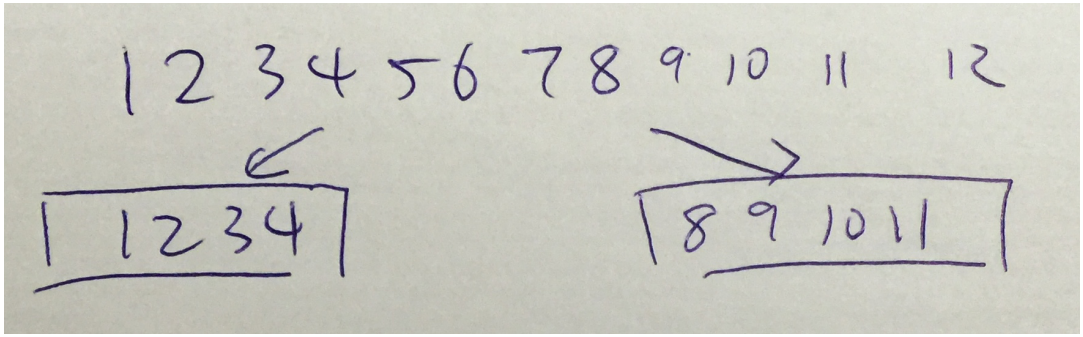
Let's check master theorem rule 3:
f(n) = n^2, which is $\Omega(n^{\log_2 2+\varepsilon})=\Omega(n^{1+\varepsilon})$,
$af(n/b) <= cf(n)$, $c<1$ — $2\ (n/2)^2 <= 0.5\ n^2$
so rule 3 is true
So, $T(n) = \Theta(f(n)) = \Theta(n^2)$

Meaning of a, b, f(n)

Consider $T(n) = aT(n/b) + f(n)$

- a means how many segments are divided
- n/b means the number of elements in the sub-segment

e.g. $T(n) = 2T(n/3) + c$ It may look like this



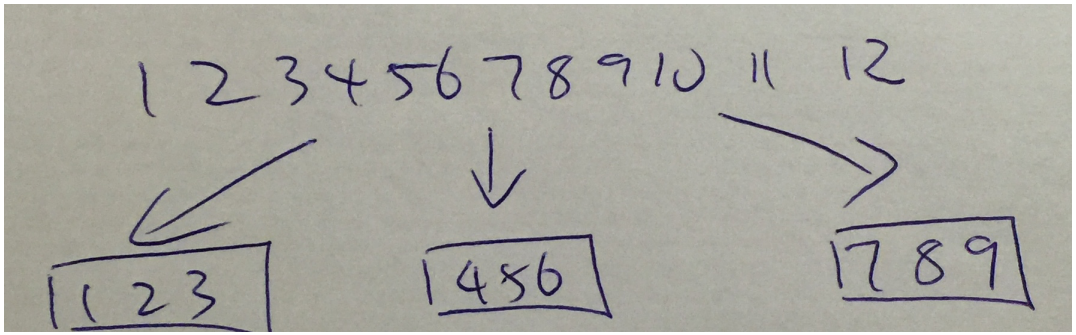
n=12
Divided into 2 segments
each segment has n/3 = 4 elements

Meaning of a, b, f(n)

Consider $T(n) = aT(n/b) + f(n)$

- a means how many segments are divided
- n/b means the number of elements in the sub-segment

e.g. $T(n) = 3T(n/4) + c$ It may look like this



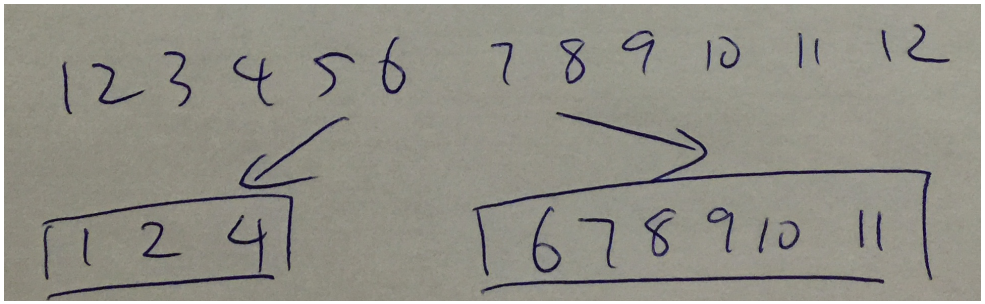
n=12
Divided into 3 segments
each segment has n/4 = 3 elements

Meaning of a, b, f(n)

Consider $T(n) = aT(n/b) + f(n)$

- a means how many segments are divided
- n/b means the number of elements in the sub-segment

e.g. $T(n) = T(n/4) + T(n/2) + c$ It may look like this



n=12

Divided into 1 segment of n/4 elements, and 1 segment of n/2 elements

Meaning of a, b, f(n)

Consider $T(n) = aT(n/b) + f(n)$

- a means how many segments are divided
- n/b means the number of elements in the sub-segment

So, T(n) only tell us the running time of the algorithm
T(n) doesn't care the elements value
(e.g. if a=2, b=2, it can split [1,2,3,4] into [1,2][3,4] or [1,3][2,4] or [1,1][2,2]… etc)
(e.g. if a=2, b=4/3, it can split [1,2,3,4] into [1,2,3][1,2,4] or [1,2,4][2,2,2]… etc)

… the values depends on your algorithm
- Values in sub-segments can overlap
- You can withdraw any elements too

Heap

Chapter 6
of the CLRS book 3rd edition

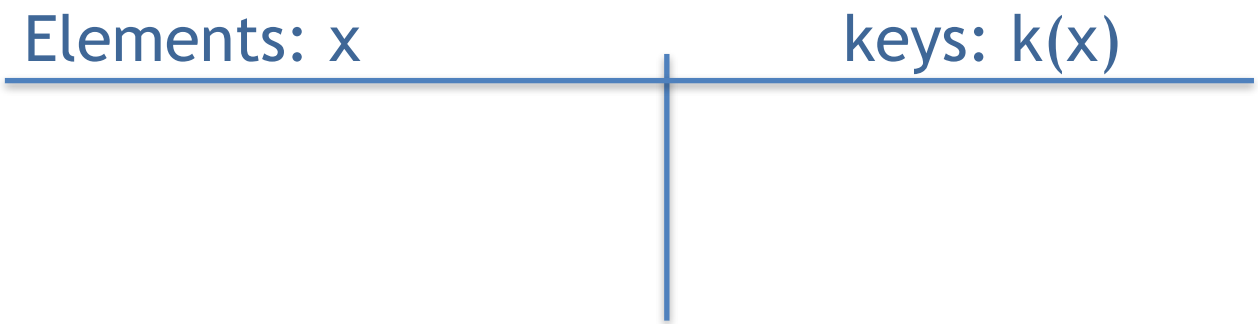
Objective

- Motivate the heap data structure
- How the heap works
- Heap-related algorithms

Priority queue

- "Priority queue" = **abstract data type** (ADT) implementing a set S of elements (each associated with a key) supporting the following operations

k(x) = key of element x



Key: encodes some useful property of the data structure

Priority queue

- "Priority queue" = **abstract data type** (ADT) implementing a set S of elements (each associated with a key) supporting the following operations

k(x) = key of element x

Elements: x	keys: k(x)
Student ID	student record
Bus route	estimated travel time
Moves in a game.	Score obtained

Key: encodes some useful property of the data structure

Priority queue

- "Priority queue" = **abstract data type** (ADT) implementing a set S of elements (each associated with a key) supporting the following operations

  k(x) = key of element x

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

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

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

  increase_key(S,x,k'): increase the value of element x's key k(x) to new value k'

Complexity

- If we use an array to implement the priority queue
- Start with an empty S
- Keep the element in S as follow:
  - (Array i) in the order they arrive
  - (Array ii) sort S in decreasing order, right after any new element is inserted

operation	Array i	Array ii
max(S)	Θ(n)	Θ(1)
insert(S,x)	Θ(1)	Θ(n)
extract_max(S)	Θ(n)	Θ(n)
increase_key(S,x,k)	Θ(n)	Θ(n)

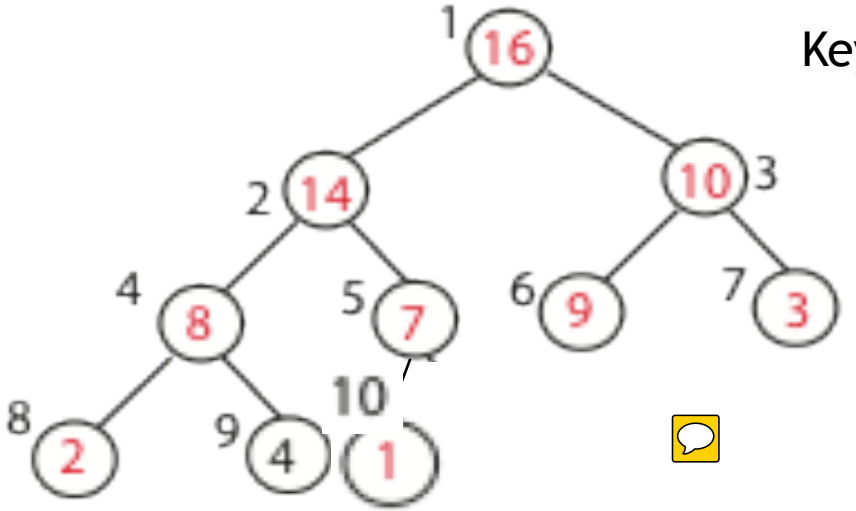
Complexity

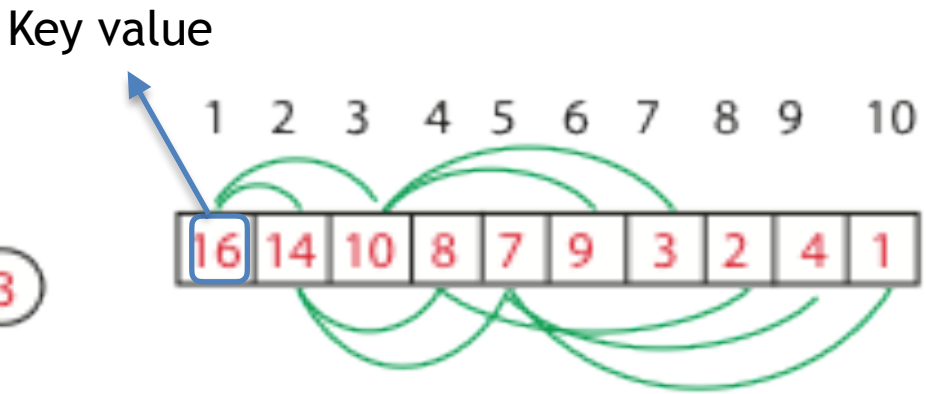
- … and this is heap

operation	complexity (i)	complexity (ii)	Heap
max(S)	Θ(n)	Θ(1)	Θ(1)
insert(S,x)	Θ(1)	Θ(n)	Θ(logn)
extract_max(S)	Θ(n)	Θ(n)	Θ(logn)
increase_key(S,x,k)	Θ(n)	Θ(n)	Θ(logn)

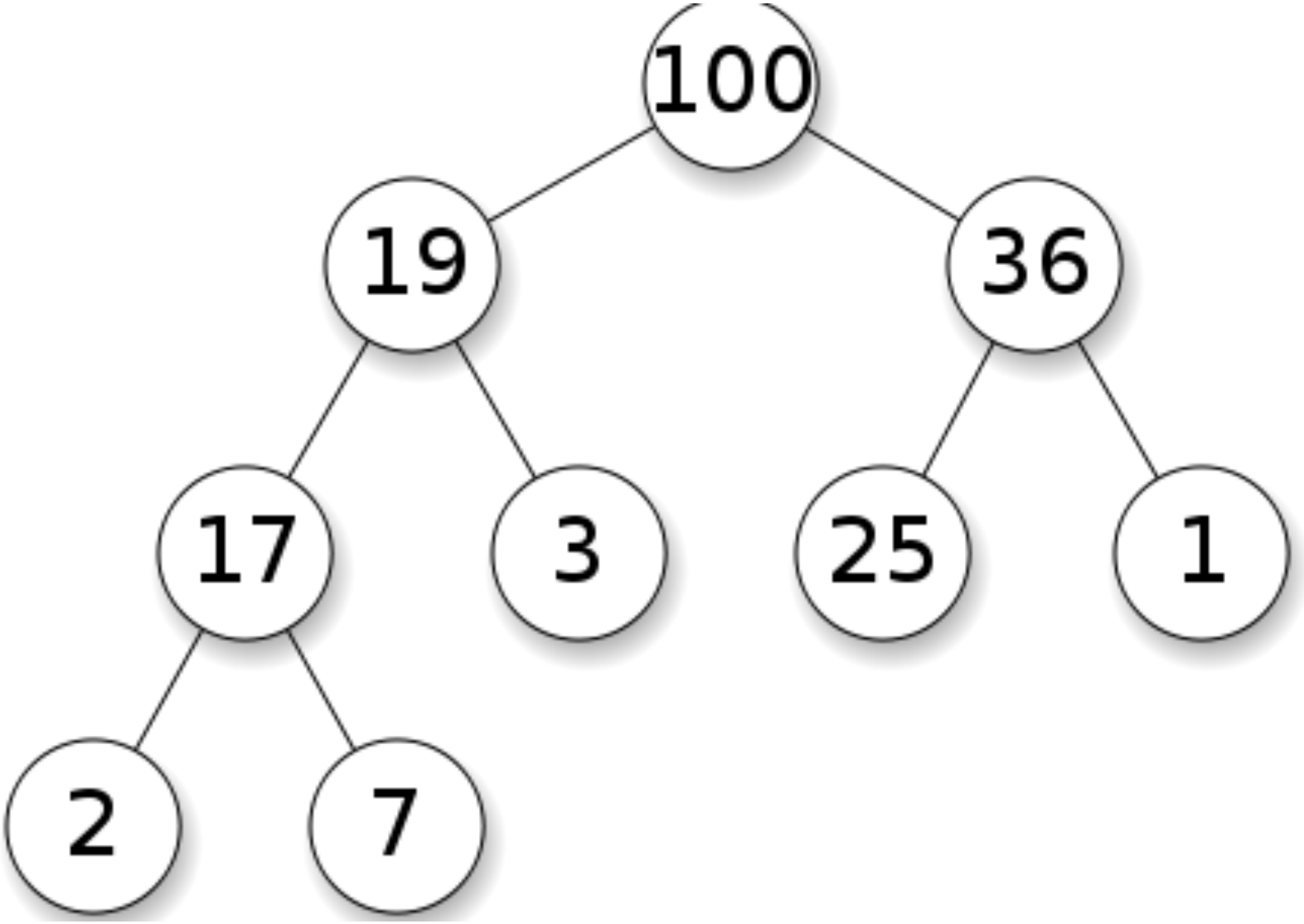
Heap

- Implementation of a priority queue
- **Heap** = an array visualized as a nearly complete binary tree
  - It is an array, but can be used as tree
- Max (Min) Heap Property: the key of a node is ≥ (≤) than the keys of its children

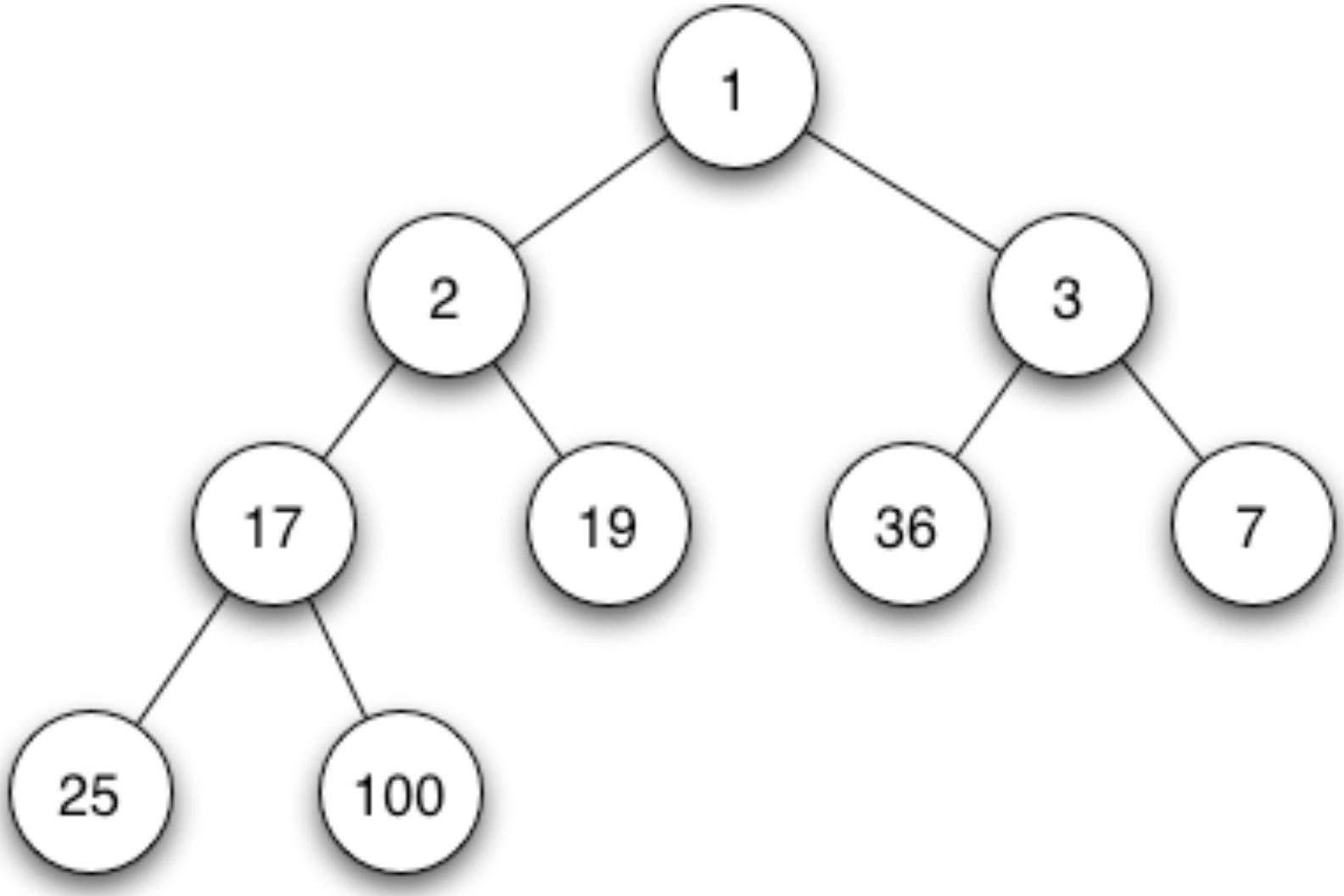




Max heap



Min heap



Heap = visualizing an array as a tree

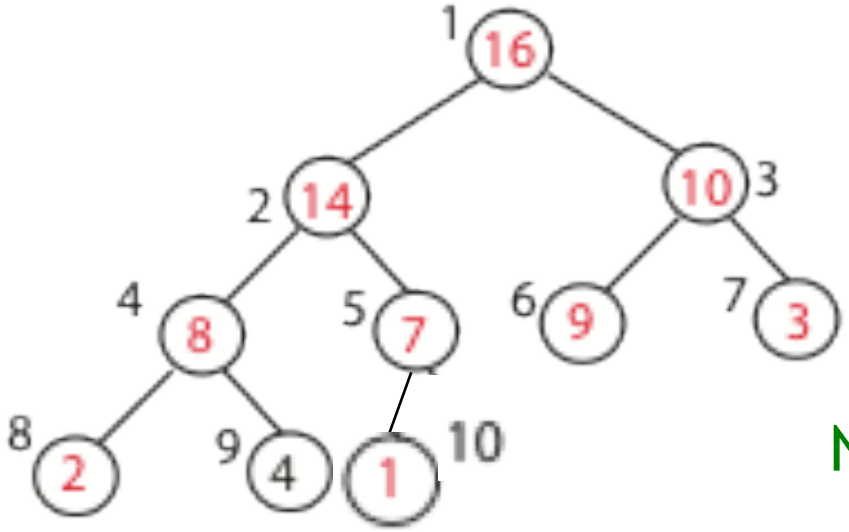
root of tree: first element in the array, corresponding to index $i = 1$

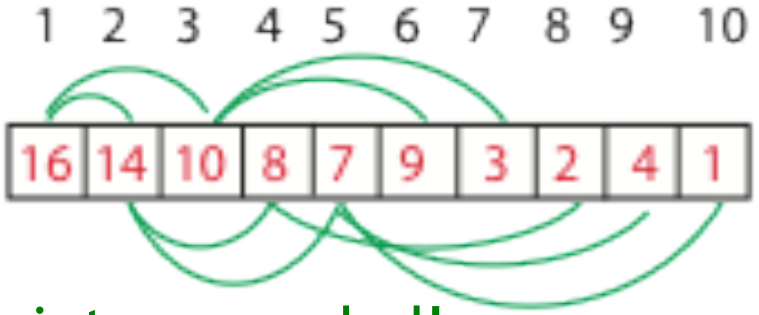
If a node's index is i then:

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

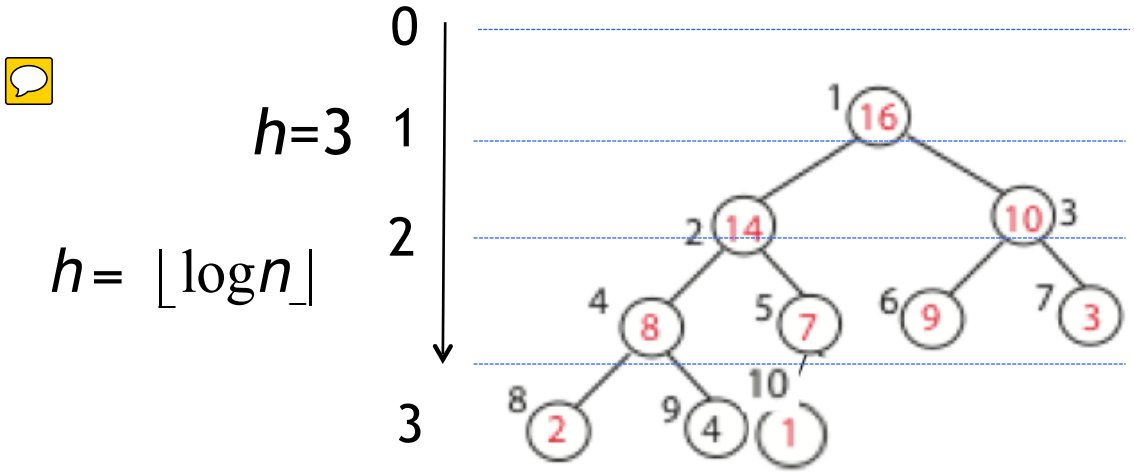




No pointers needed!

Basic property of a heap

- No pointers required to implement a tree!
- Height of a binary heap O(log *n*)



Any path from root to a leaf has $h \le \log n = O(\log n)$

Operations with Heaps

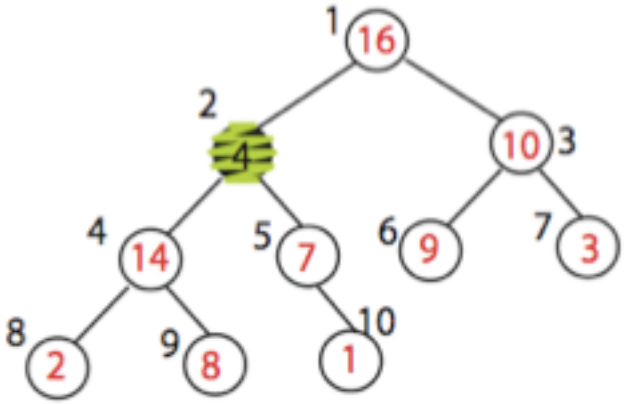
build_max_heap : produce a max-heap from an unordered array. Complexity: O(n)

max_heapify : correct **a single violation** of the heap property **occurring at the root of a sub-tree**. Complexity: O(log n)

insert, extract_max : Basic operations of a heap Complexity: O(log n)

heapsort : sort an array of size n using heap. Complexity: O(n log n)

think of an algorithm to correct the inconsistency in node 2
What is the complexity?



Max_heapify (A,i)

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

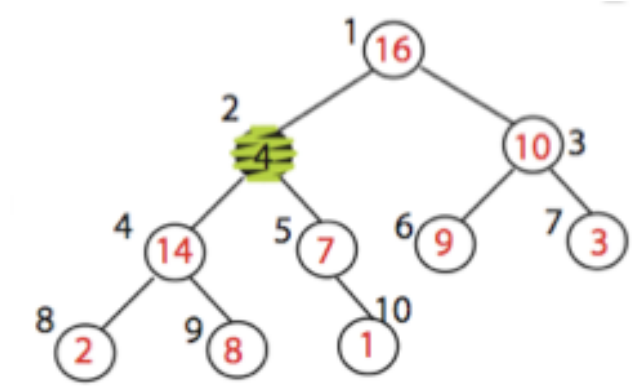
Assume that the trees rooted at left(*i*) and right(*i*) are max-heaps, but element A[*i*] violates the max-heap property;

i.e. A[*i*] is smaller than at least one of A[left(*i*)] or A[right(*i*)].

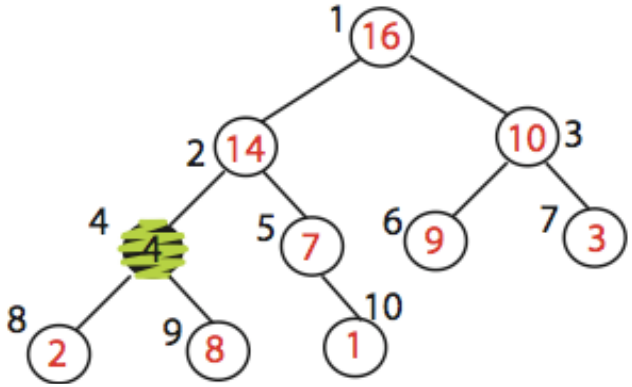
The goal is to correct the violation.

Do this by trickling element A[*i*] down the tree, making the subtree rooted at index *i* a max-heap.

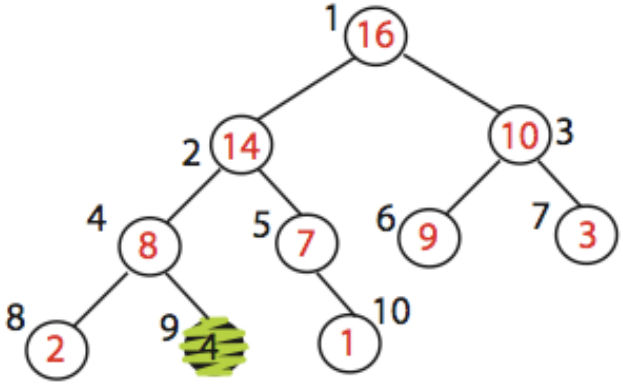
Max_heapify (Example)



MAX_HEAPIFY (A,2)



Exchange A[2] with A[4]
Call MAX_HEAPIFY(A,4)
because max_heap property
is violated



Exchange A[4] with A[9]
No more calls

Max_heapify (Pseudocode)

Max_heapify (A, *i*)

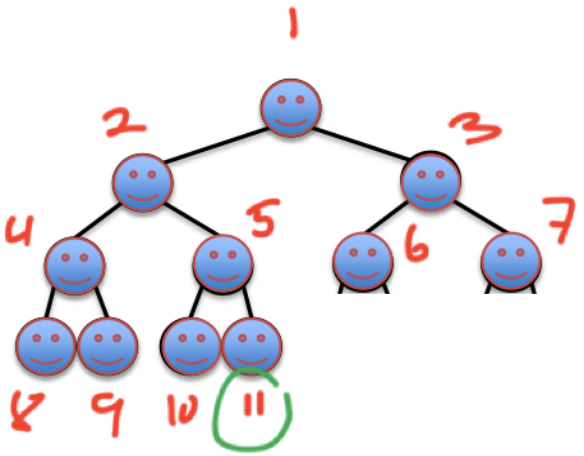
Find the index of the largest element among A[*i*], A[left(*i*)] *and* A[right(*i*)]

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

```
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)

Convert A[1…*n*] to a max heap



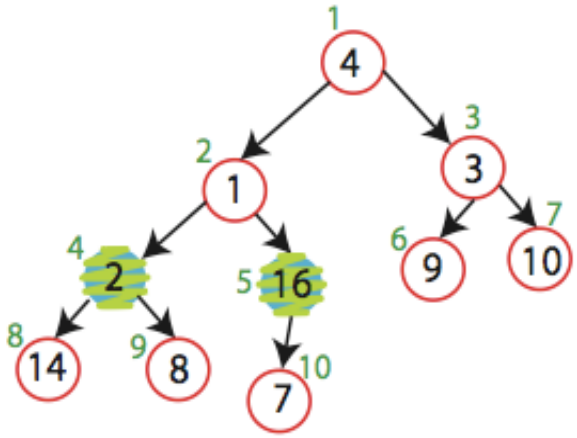
Observation: Elements $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are leaves of the tree

because $2i > n$, for all $i \geq \lfloor n/2 \rfloor + 1$
(if 2i > n, then node i has no left child. So it must be a leave)

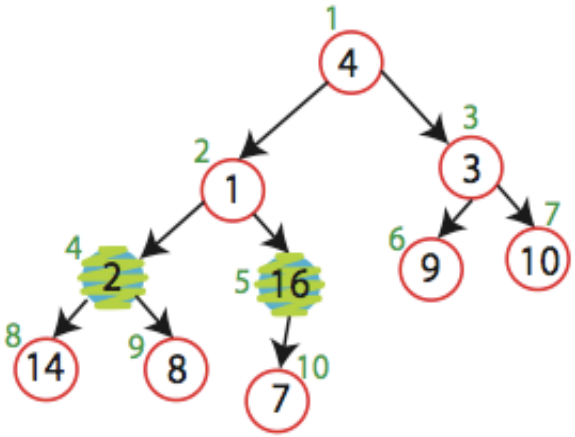
so heap property may only be violated at nodes 1… $\lfloor n/2 \rfloor$ of the tree

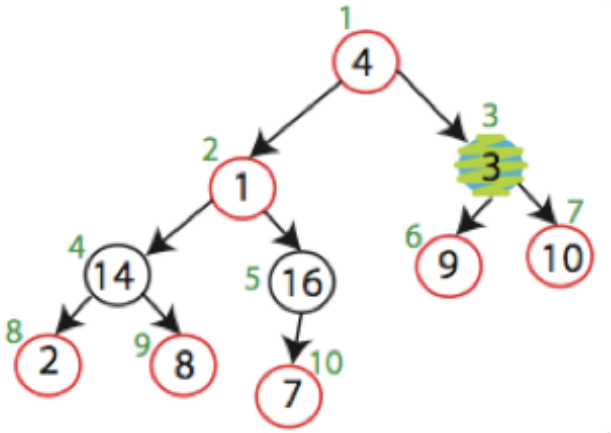
```
Build_Max_Heap(A):
    heap_size(A) = length(A)
    for i ← ⌊ length[A]/2⌋ downto 1
        do Max_Heapify(A, i)
```

Build_Max_Heap (Example execution)

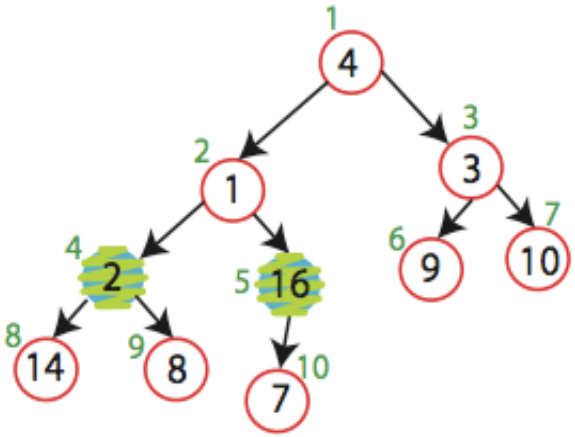


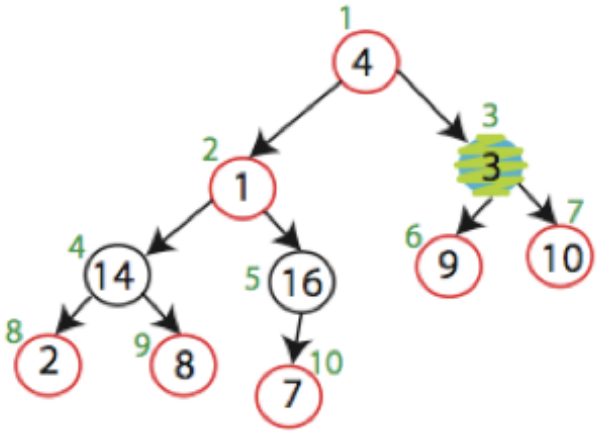
Build_Max_Heap (Example execution)

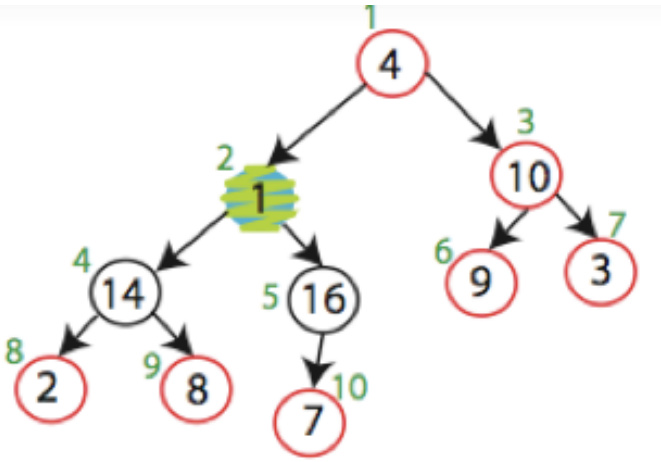




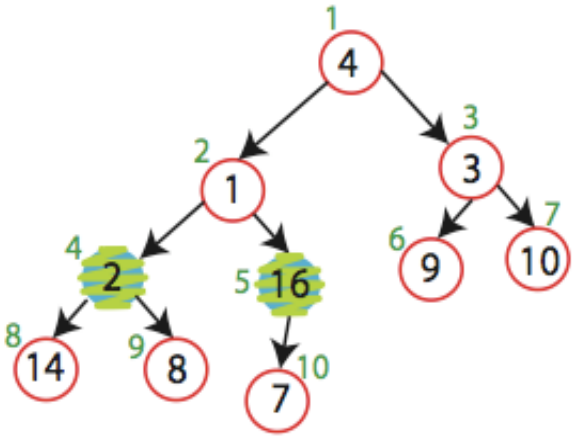
Build_Max_Heap (Example execution)

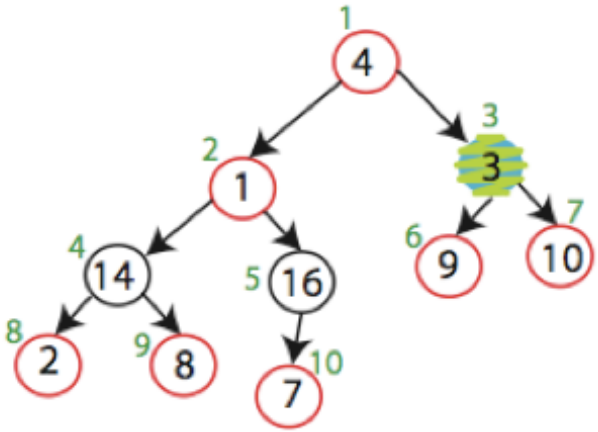


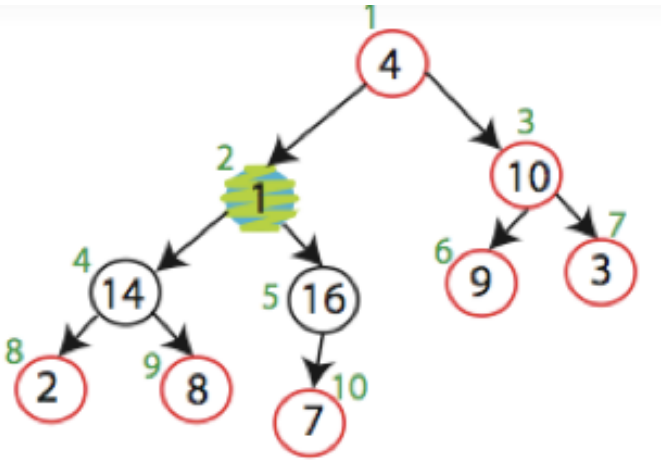


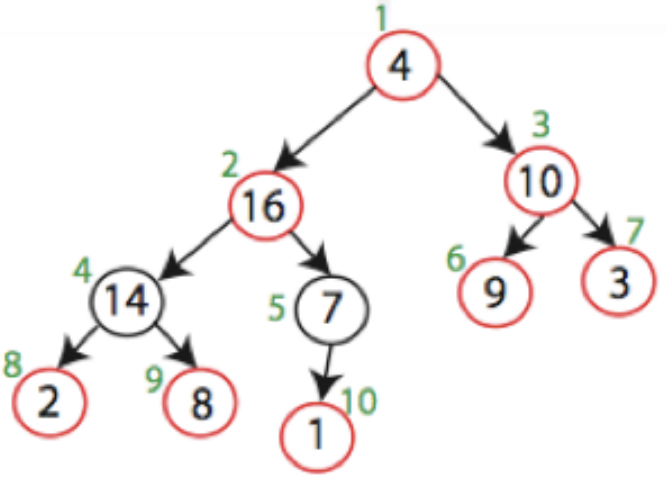


Build_Max_Heap (Example execution)

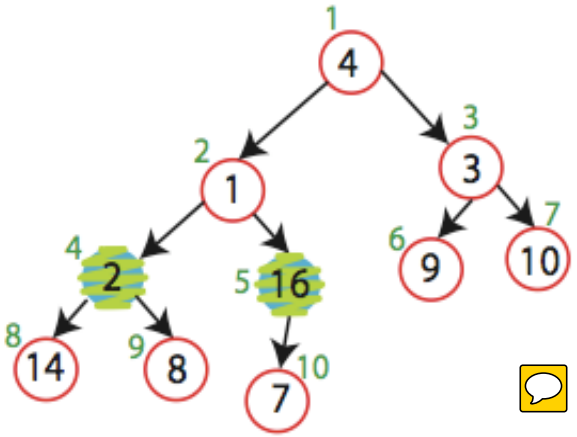


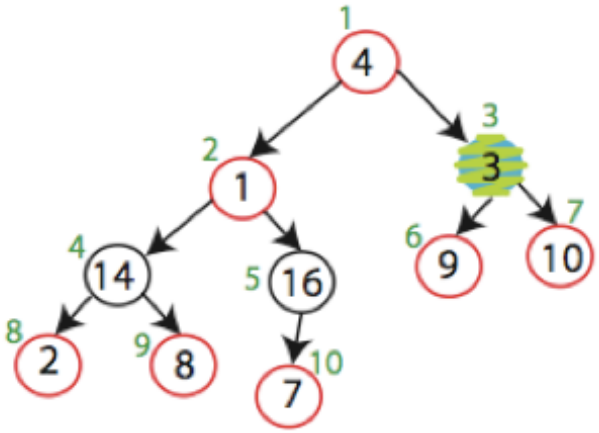


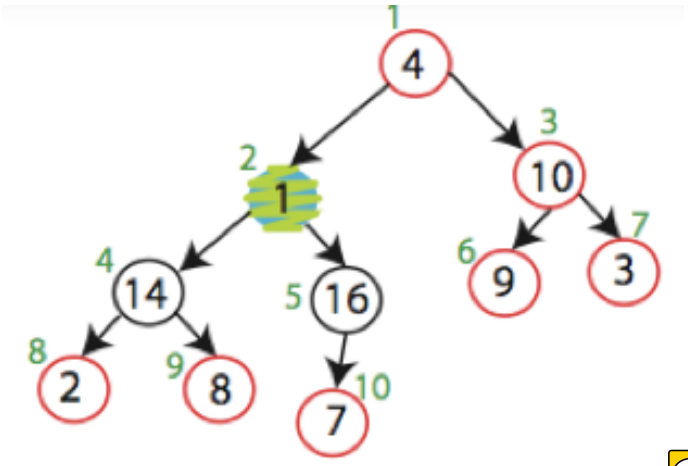


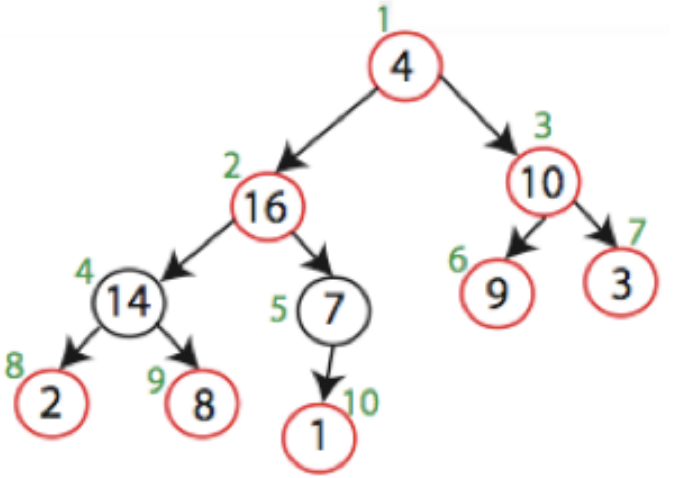


Build_Max_Heap (Example execution)





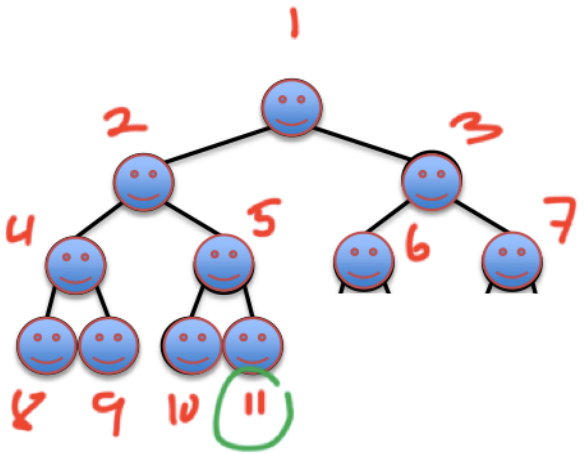




$\Rightarrow \ldots\ldots$

Complexity of Build_Max_Heap?

- An Obvious upper bound for Build_Max_Heap: O(nlogn)
  - Heapify for O(n) times
  - Each heapify = O(logn)



Recall a naïve algorithm for sorting an array

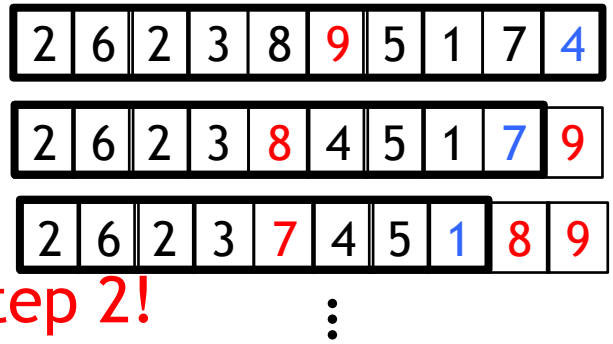
Sorting Strategy:

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:

$\Theta(n^2)$

1. last_el = n; $\Theta(n)$
2. Find maximum element A[i] of array A[1…last_el];
3. Swap A[i] and A[last_el];
4. last_el = last_el - 1;
5. Go to step 2



Now, we have a fast data structure for step 2!
(which is also the most costly)
Let's implement the array as a heap

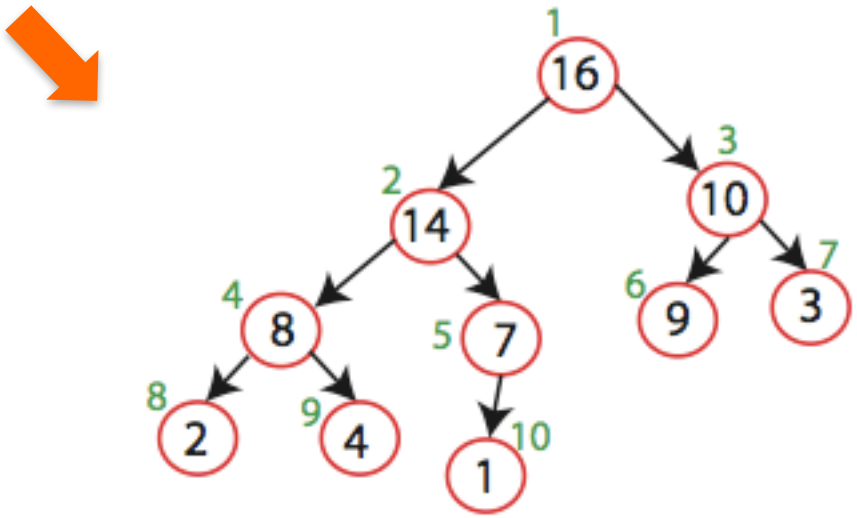
Heap-Sort

Sorting Strategy:

1. Build a Max Heap from an unordered array;

Heap-Sort

A	4	1	3	2	16	9	10	14	8	7



Heap-Sort

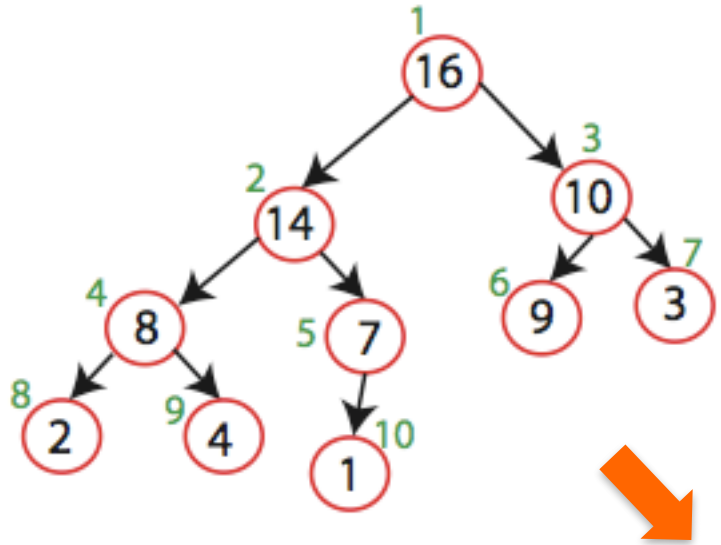
Sorting Strategy:

1. Build a Max Heap from unordered array;

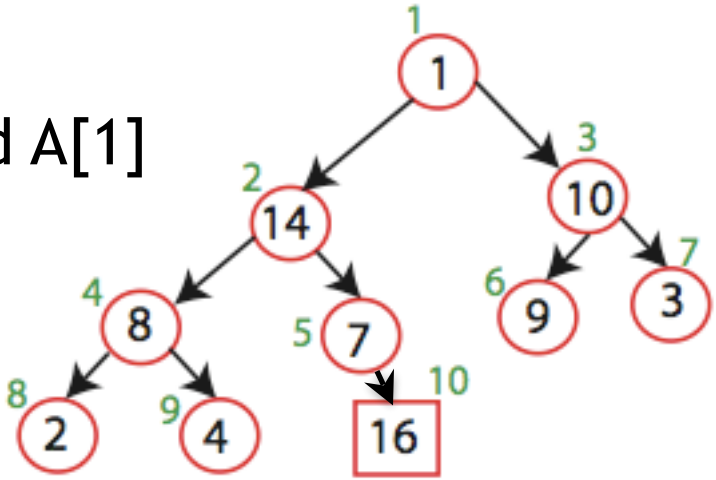
2. Find the maximum element: A[1];

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

Heap-Sort



Swap elements A[10] and A[1]

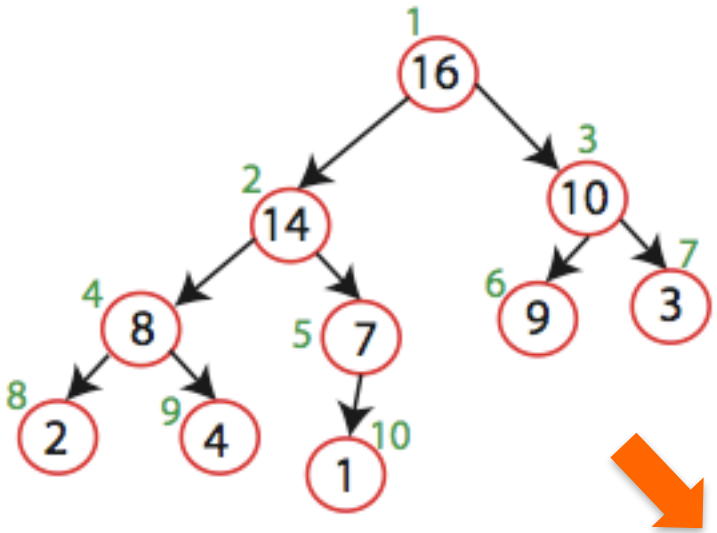


Heap-Sort

Sorting Strategy:

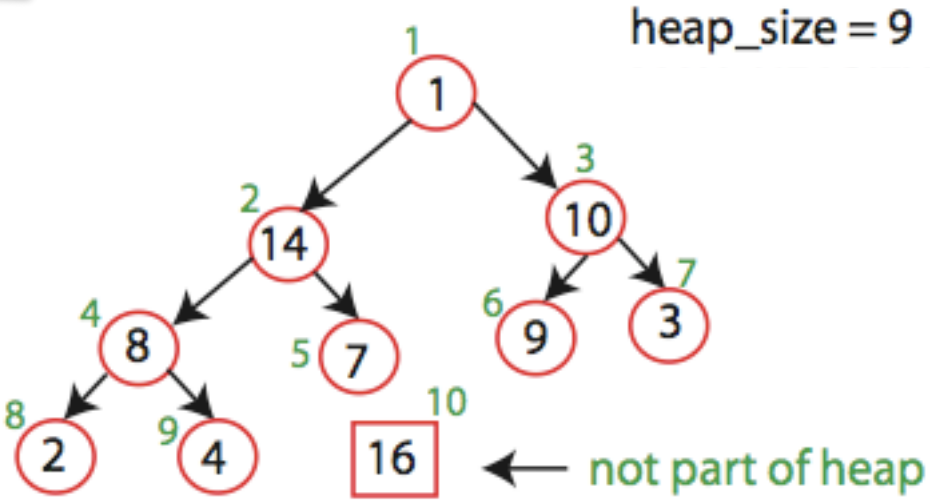
1. Build Max Heap from unordered array;
2. Find maximum element A[1];
3. Swap elements A[*n*] and A[1]:
   now max element is at the end of the array!
4. Discard node *n* from the heap
   (by decrementing heap-size variable)

Heap-Sort



Swap elements A[10] and A[

heap_size = heap_size-1



Heap-Sort

Sorting Strategy:

1. Build Max Heap from unordered array;

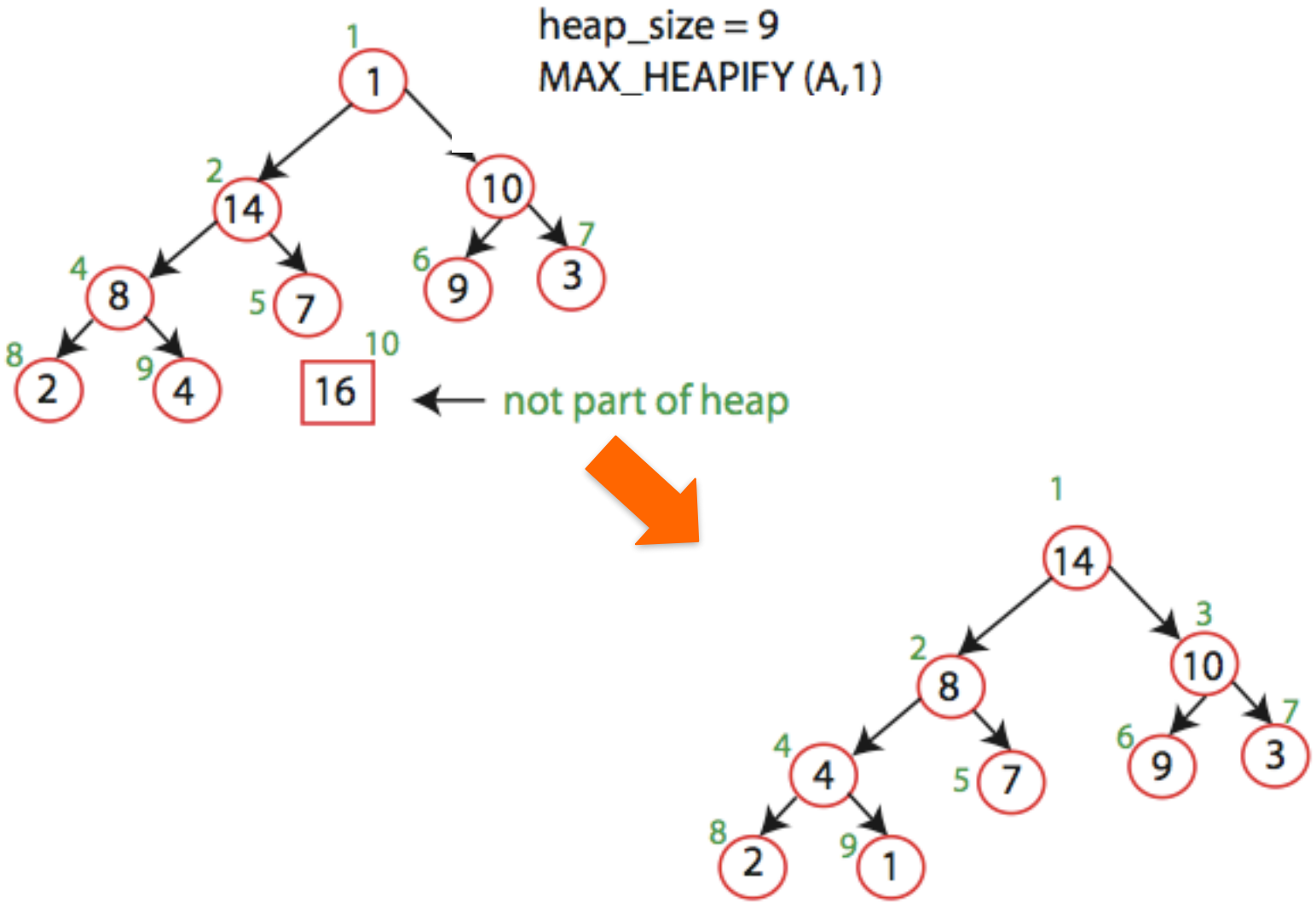
2. Find maximum element A[1];

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

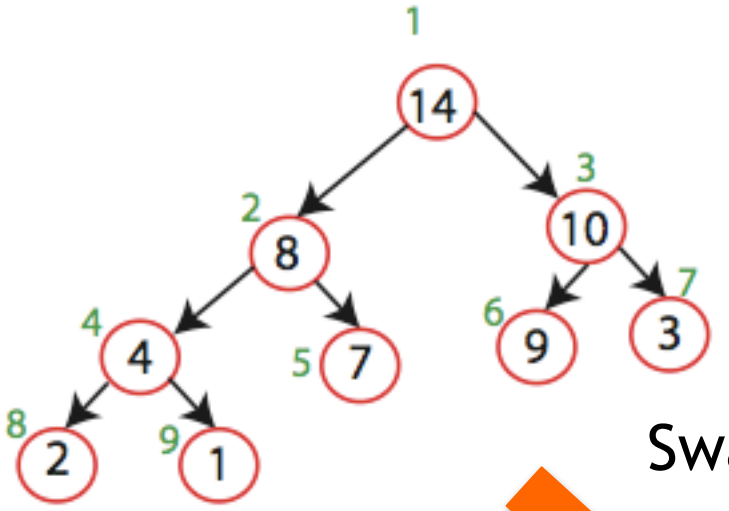

heap_size = 9
MAX_HEAPIFY (A,1)
1
1
2
14
10
4
8
5
7
6
9
7
3
8
2
9
4
10
16
not part of heap
1
14
2
8
3
10
4
4
5
7
6
9
7
3
8
2
9
1

Heap-Sort

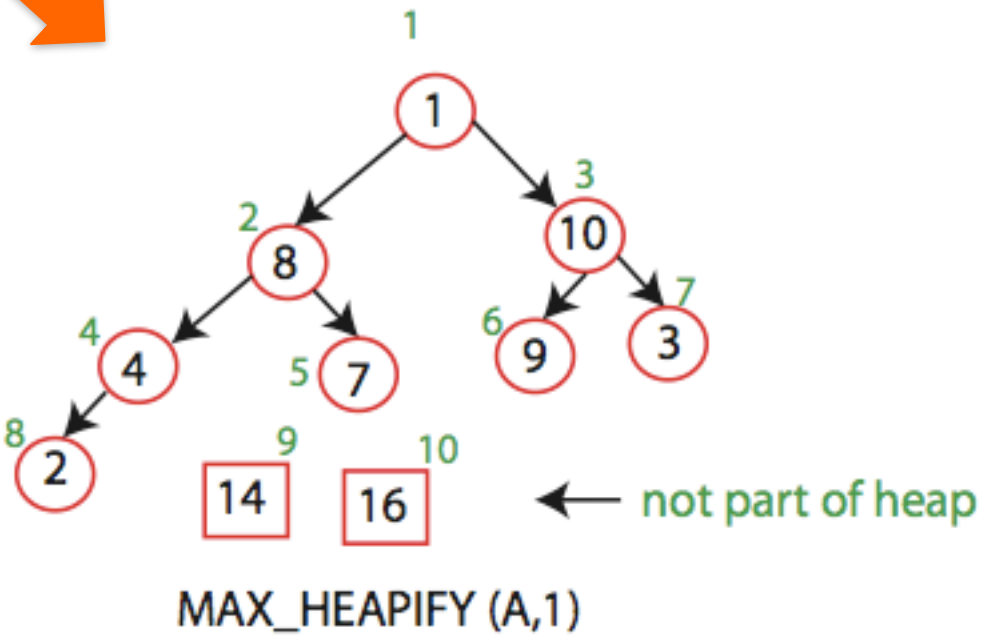
Sorting Strategy:

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.

Heap-Sort

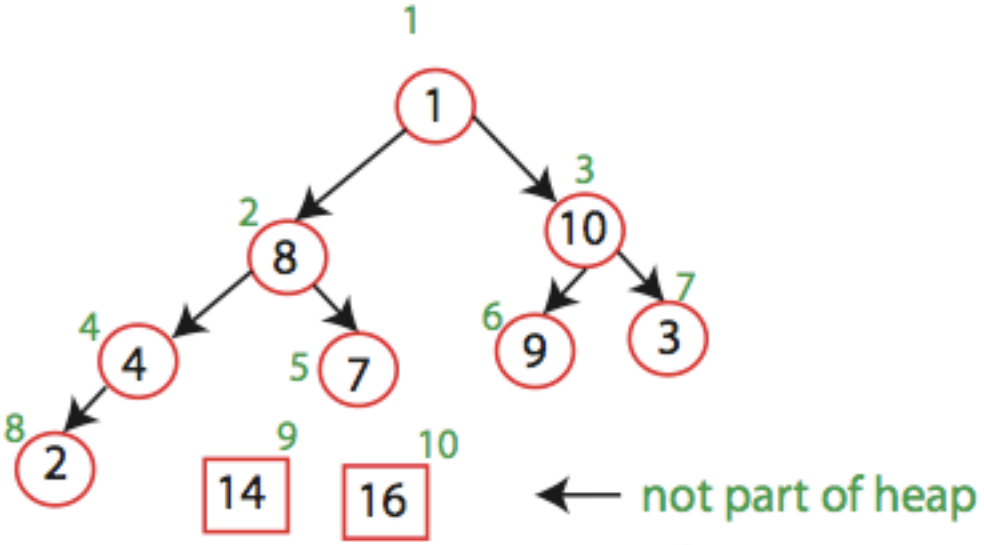


Swap elements A[9] and A[1]



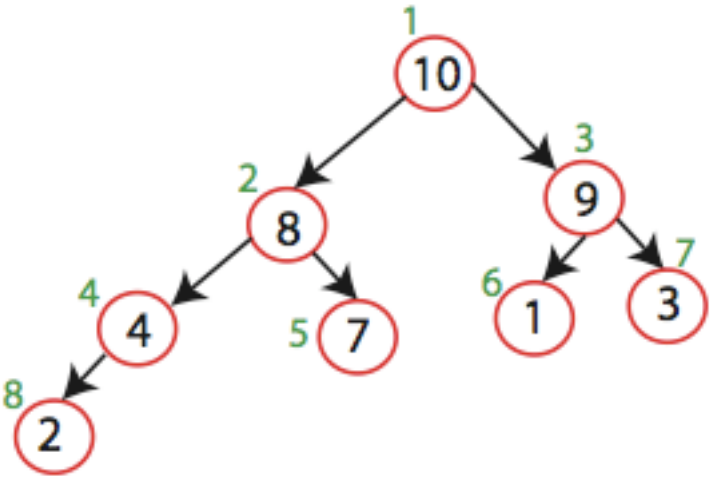
MAX_HEAPIFY (A,1)

Heap-Sort

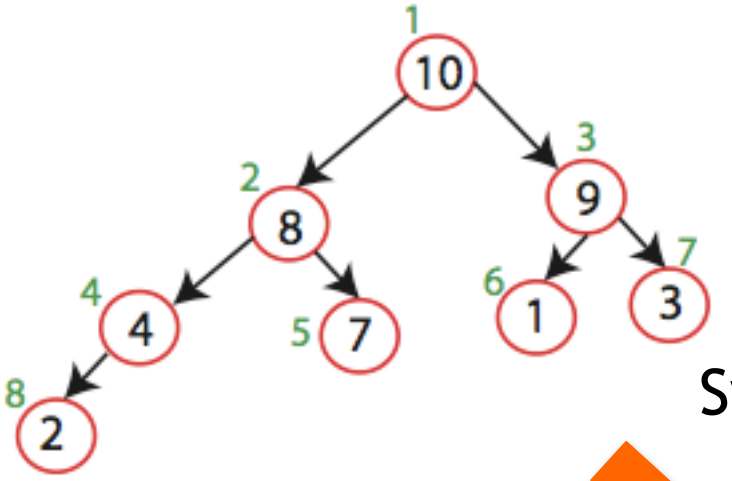


MAX_HEAPIFY (A,1)

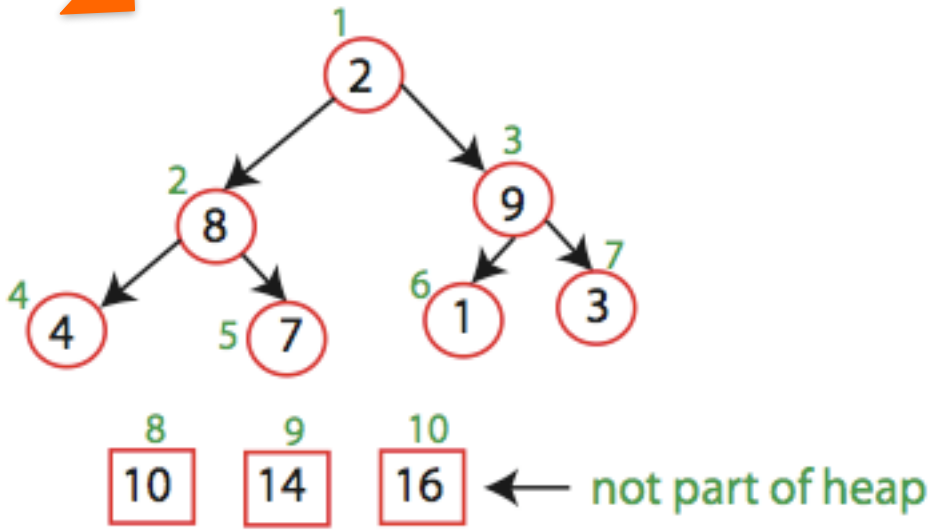
Max_Heapify(A,1)



Heap-Sort



Swap elements A[8] and A[1]



and so on…

Heap-Sort

Running time:

after n iterations the Heap is empty

every iteration involves a swap and a heapify operation; hence it takes O(log n) time

Overall O(n log n)

Operations with Heaps

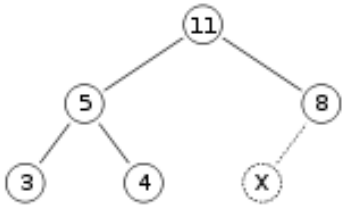
- build_max_heap : produce a max-heap from an unordered array in O(*n*);
- max_heapify : correct a single violation of the heap property occurring at the root of a subtree in O(log *n*);

insert, extract_max, increase_key : O(log n)

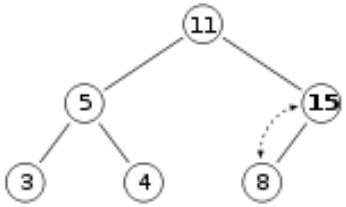
- heapsort : sort an array of size *n* in O(*n* log *n*) using heaps

Insert

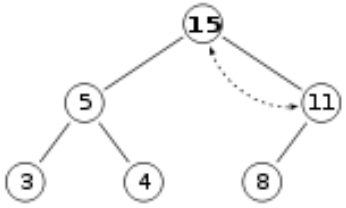
As an example of binary heap insertion, say we have a max-heap



and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 > 8, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:



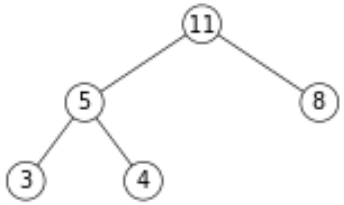
However the heap property is still violated since 15 > 11, so we need to swap again:



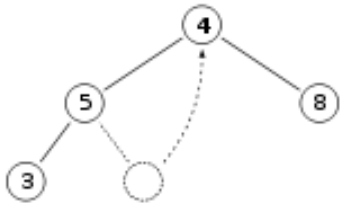
So it takes O(logn)

Extract max

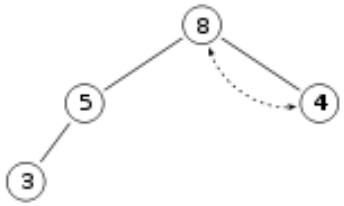
So, if we have the same max-heap as before



We remove the 11 and replace it with the 4.



Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further:



So it takes O(logn)

Increase key

- Very simple…
- Start from the bottom, search up the tree
- Found the key, increase its value

So it takes O(logn)