2IL50 Data Structures

2019-20 Q3

Lecture 3: Heaps

Technische Universiteit
Eindhoven
University of Technology

Solving recurrences

one more time ...

Solving recurrences

Easiest: Master theorem
caveat: not always applicable

Alternatively: Guess the solution and use the substitution method to prove that your guess is correct.

How to guess:

1. expand the recursion
2. draw a recursion tree

Example

Example(A)

▶ A is an array of length n

1. $n = A.\mathrm{length}$
2. **if** $n == 1$
3. **then return** $A[1]$
4. **else begin**
5. Copy $A[1 \ldots \lceil n/2 \rceil]$ to auxiliary array $B[1 \ldots \lceil n/2 \rceil]$
6. Copy $A[1 \ldots \lceil n/2 \rceil]$ to auxiliary array $C[1 \ldots \lceil n/2 \rceil]$
7. b = Example(B); c = Example(C)
8. **for** $i = 1$ **to** n
9. **do for** $j = 1$ **to** i
10. **do** $A[i] = A[j]$
11. **return** 43
12. **end**

Let $T(n)$ be the worst case running time of Example on an array of length n.

Lines 1,2,3,4,11, and 12 take $\Theta(1)$ time.

Lines 5 and 6 take $\Theta(n)$ time.

Line 7 takes $\Theta(1) + 2\,T(\lceil n/2 \rceil)$ time.

Lines 8 until 10 take $\sum_{i=1}^{n} \sum_{j=1}^{i} \Theta(1) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$ time.

If $n = 1$ Lines 1,2, and 3 are executed, else Lines 1,2, and 4 until 12 are executed.

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

➡ use master theorem ...

The master theorem

Let a and b be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

Then we have:

1. If $f(n) = O(n^{(\log_b a)-\varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ for some constant $\varepsilon > 0$,

   and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n,

   then $T(n) = \Theta(f(n))$

Quiz

Recurrence	Master theorem?	
1. $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^3)$	yes	$T(n) = \Theta(n^3)$
2. $T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n)$	yes	$T(n) = \Theta(n^2)$
3. $T(n) = T\left(\frac{n}{2}\right) + 1$	yes	$T(n) = \Theta(\log n)$
4. $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$	no	$T(n) = \Theta(n \log n)$
5. $T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n^2)$	yes	$T(n) = \Theta(n^2 \log n)$
6. $T(n) = \sqrt{n}T(\sqrt{n}) + n$	no	$T(n)$

Substitution

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ 7T(\lfloor n/3 \rfloor) + n^2 & \text{if } n > 2 \end{cases}$$

Claim: $T(n) = O(n^2)$

(To show: exist constants c and n_0 such that $T(n) \leq cn^2$ for all $n \geq n_0$)

Proof: by induction on n

Base case ($n = 1$): $1 \leq c \cdot n^2 = c \cdot 1^2 = c$ for $c \geq 1$

Base case ($n = 2$): $1 \leq c \cdot n^2 = c \cdot 2^2 = 4c$ for $c \geq 0.25$

Inductive step: IH: Assume that for all $1 \leq k < n$ it holds that $T(k) \leq ck^2$.

Then

$$\begin{aligned} T(n) &= 7T(\lfloor n/3 \rfloor) + n^2 \\ &\leq 7 \cdot c \cdot (\lfloor n/3 \rfloor)^2 + n^2 && \text{(by IH)} \\ &\leq 7/9 \cdot cn^2 + n^2 \\ &= cn^2 - 2/9 \cdot cn^2 + n^2 \\ &\leq cn^2 && \text{(for } c \geq 9/2 \text{ we have } -2/9 \cdot cn^2 + n^2 \leq 0) \end{aligned}$$

Substitution

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ 7T(\lfloor n/3 \rfloor) + n^2 & \text{if } n > 2 \end{cases}$$

Claim: $T(n) = O(n^2)$

(To show: exist constants c and n_0 such that
$T(n) \leq cn^2$ for all $n \geq n_0$)

Proof: by induction on n

Base case ($n = 1$): [...]

Base case ($n = 2$): [...]

Inductive step: [...]

Let $n_0 = 1$ and $c = 9/2$.
By induction it holds that $T(n) = O(n^2)$. ∎

Tips

Analysis of recursive algorithms:
find the recursion and solve with master theorem if possible

Analysis of loops: summations

Some standard recurrences and sums:

$$T(n) = 2T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n \log n)$$

$$\sum_{i=1}^{n} i = \frac{1}{2}n(n+1) = \Theta(n^2)$$

$$\sum_{i=1}^{n} i^2 = \frac{1}{6}n(n+1)(2n+1) = \Theta(n^3)$$

Heaps

Event-driven simulation

Stores a set of events, processes first event (highest priority)

Supporting data structure:

- insert event
- find (and extract) event with highest priority
- change the priority of an event

Priority queue

Max-priority queue

abstract data type (ADT) that stores a set S of elements,
each with an associated key (integer value).

Operations

Insert(S, x):	inserts element x into S, that is, $S \leftarrow S \cup \{x\}$
Maximum(S):	returns the element of S with the largest key
Extract-Max(S):	removes and returns the element of S with the largest key
Increase-Key(S, x, k):	give $\mathrm{key}[x]$ the value k condition: k is larger than the current value of $\mathrm{key}[x]$

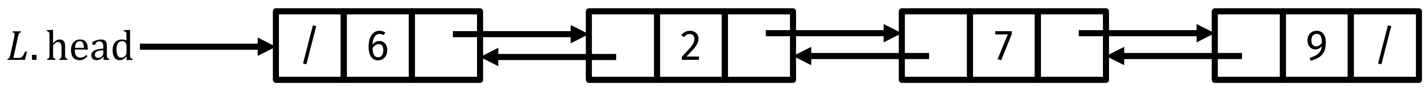
Min-priority queue …

Implementing a priority queue

	Insert	Maximum	Extract-Max	Increase-Key
sorted list				
sorted array				

(Doubly) linked list

Linked list collection of objects stored in linear order, with objects pointing to their predecessor and successor



$L.\text{head}$ points to the first object

$L.\text{head} = NIL$ if L is empty

Object x:

- $x.\text{prev}$ points to the predecessor
- $x.\text{next}$ points to the successor
- $x.\text{key}$, $x.\text{data}$

$x.\text{prev} = NIL$ if x is first

$x.\text{next} = NIL$ if x is last

Operations

- Search(L, key) $O(n)$
- Insert(L, x) $O(1)$
- Delete(L, x) $O(1)$

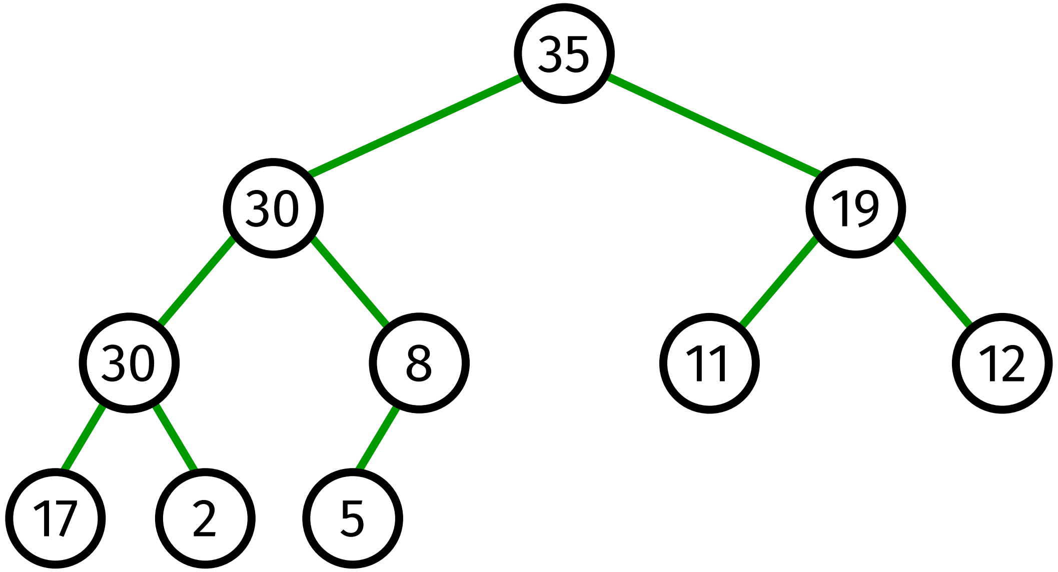
Implementing a priority queue

	Insert	Maximum	Extract-Max	Increase-Key
sorted list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
sorted array	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$?	$\Theta(n)$

Today

	Insert	Maximum	Extract-Max	Increase-Key
heap	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

Max-heap



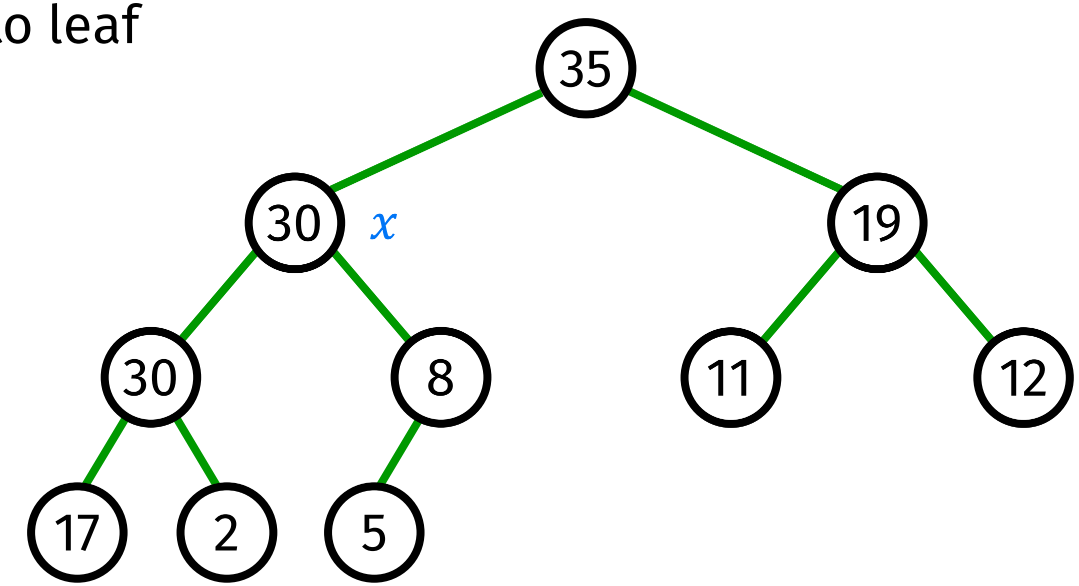
Heap
    nearly complete binary tree, filled on all levels except possibly the lowest.
    (lowest level is filled from left to right)

Max-heap property: for every node i other than the root $\mathrm{key}[\mathrm{Parent}(i)] \geq \mathrm{key}[i]$

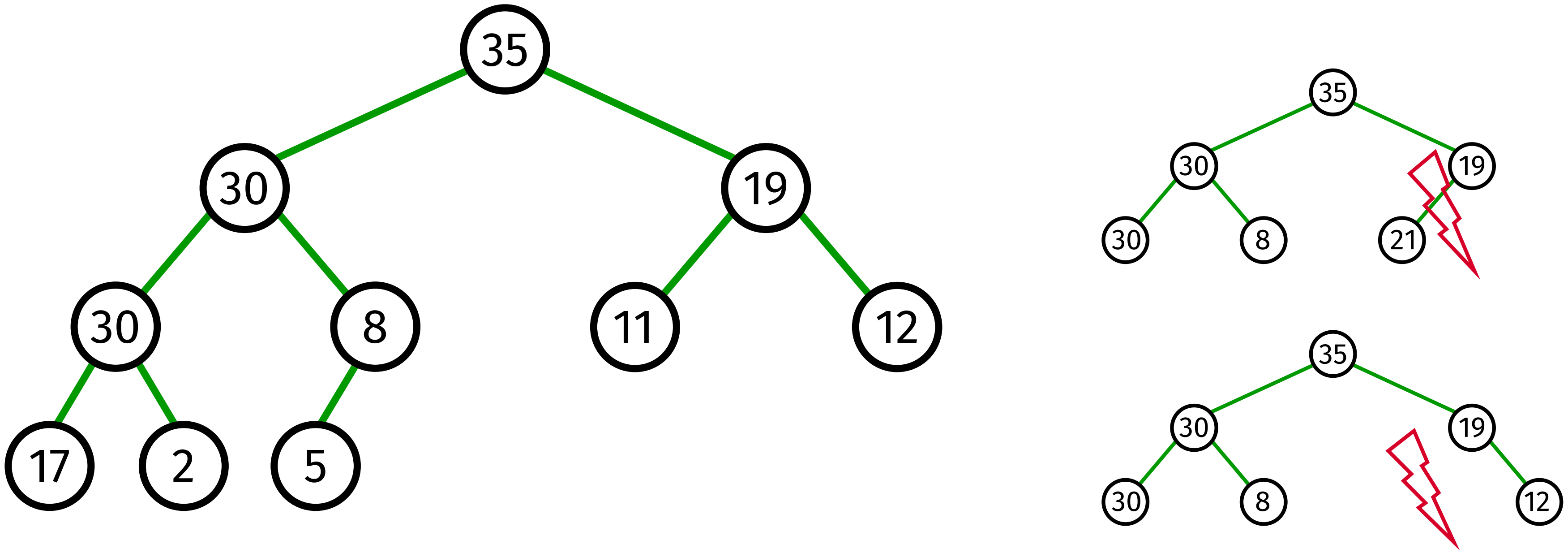
Tree terminology

Binary tree: every node has 0, 1, or 2 children
Root: top node (no parent)
Leaf: node without children
Subtree rooted at node x: all nodes below and including x
Depth of node x: length of path from root to x
Depth of tree: max. depth over all nodes
Height of node x: length of longest path from x to leaf
Height of tree: height of root
Level: set of nodes with same depth

Family tree terminology
Left/right child
Parent
Grandparent ...



Max-heap



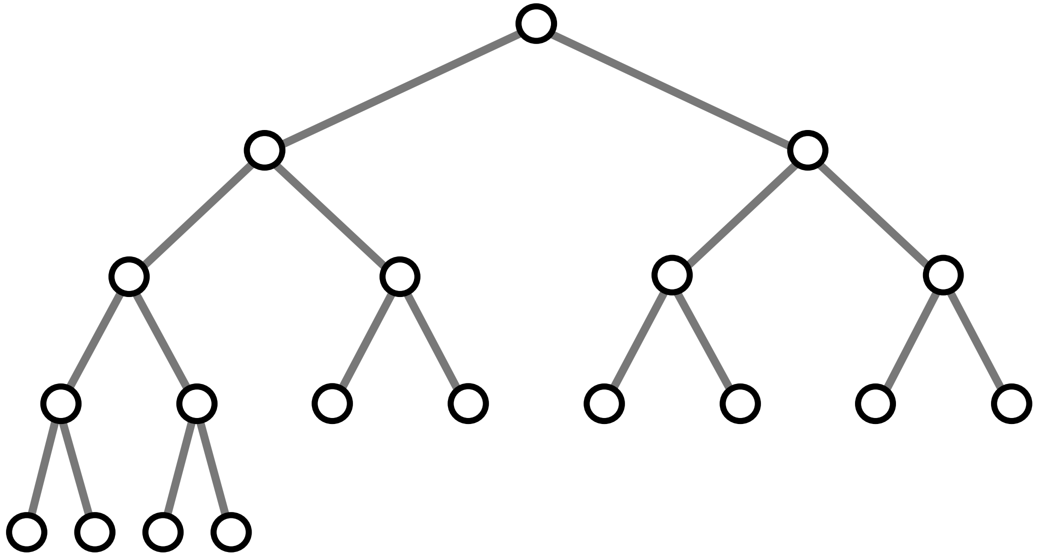
Heap
: nearly complete binary tree, filled on all levels except possibly the lowest.
(lowest level is filled from left to right)

Max-heap property: for every node i other than the root $\mathrm{key}[\mathrm{Parent}(i)] \geq \mathrm{key}[i]$

Properties of a max-heap

Lemma
: The largest element in a max-heap is stored at the root.

Proof:



Properties of a max-heap

Lemma

The largest element in a max-heap is stored at the root.

Proof: x root

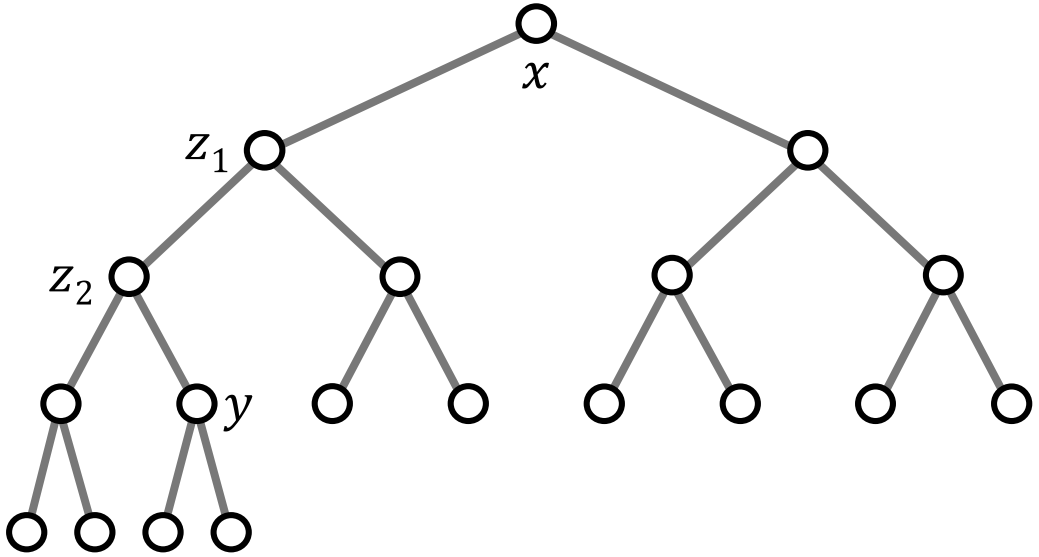
y arbitrary node

$z_1, z_2, \ldots, z_k$ nodes on path between x and y

max-heap property

➡ $\text{key}[x] \geq \text{key}[z_1] \geq \cdots \geq \text{key}[z_k] \geq \text{key}[y]$

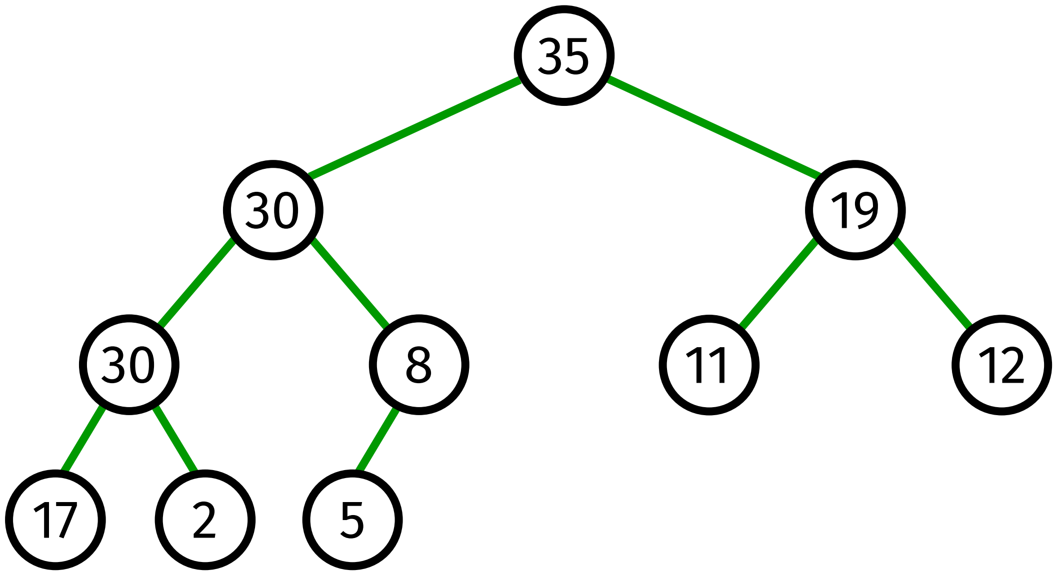
➡ the largest element is stored at x ■



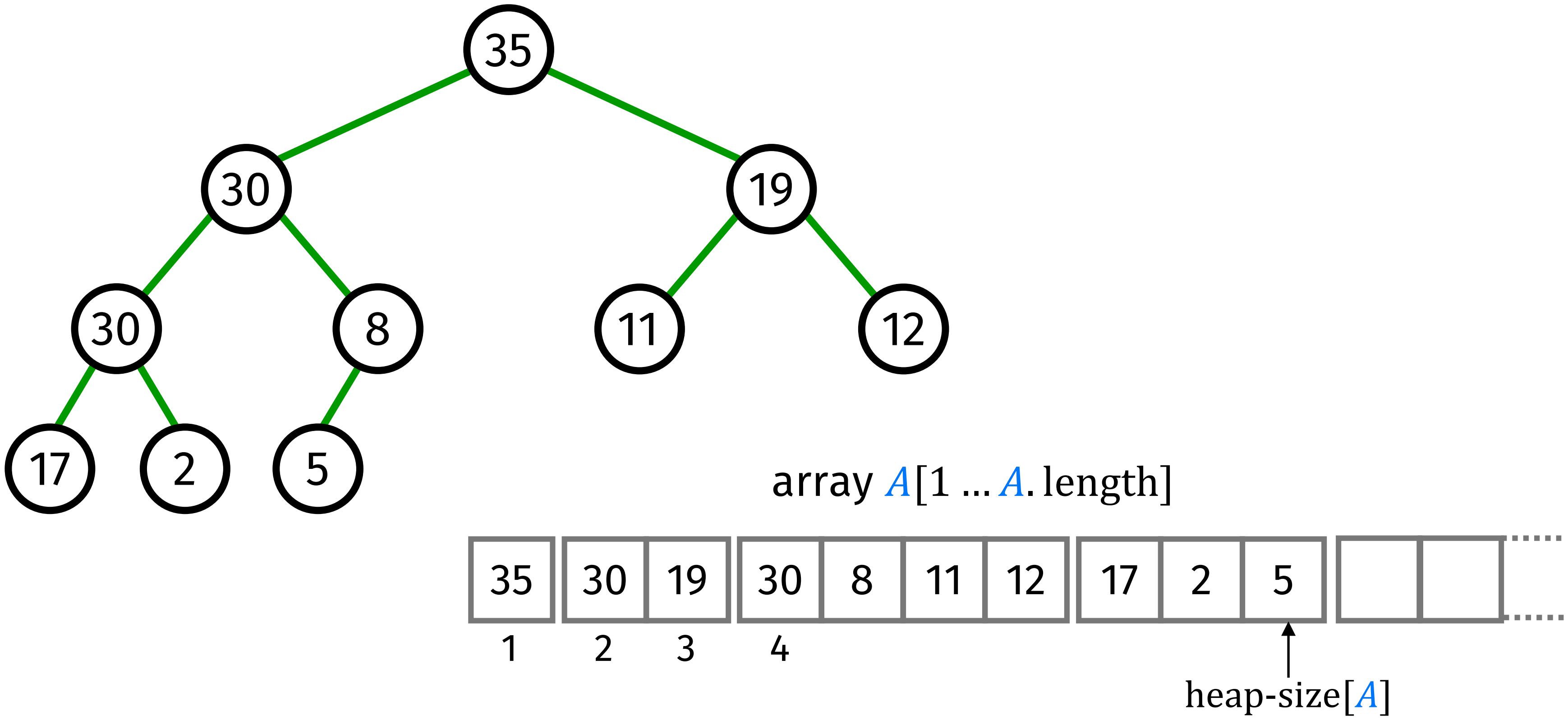
Storing a heap

How to store a heap?

- Tree structure?
- In an array?



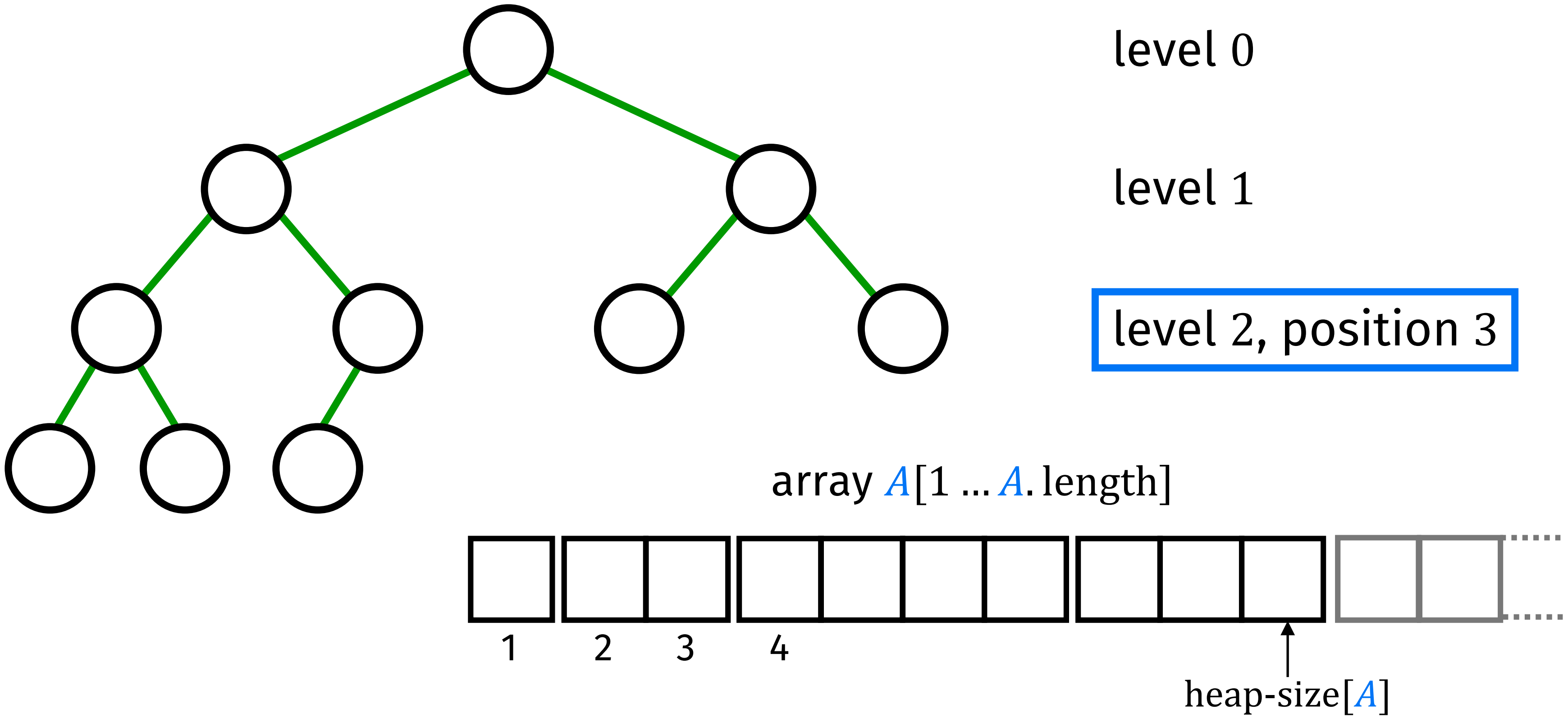
Implementing a heap with an array



$A.\text{length}$ = length of array A

heap-size$[A]$ =number of elements in the heap

Implementing a heap with an array



k^{th} node on level j is stored at position $A[2^j + k - 1]$

left child of node at position i = $\text{Left}(i) = 2i$

right child of node at position i = $\text{Right}(i) = 2i + 1$

parent of node at position i = $\text{Parent}(i) = \lfloor i/2 \rfloor$

Priority queue

Max-priority queue

abstract data type (ADT) that stores a set S of elements, each with an associated key (integer value).

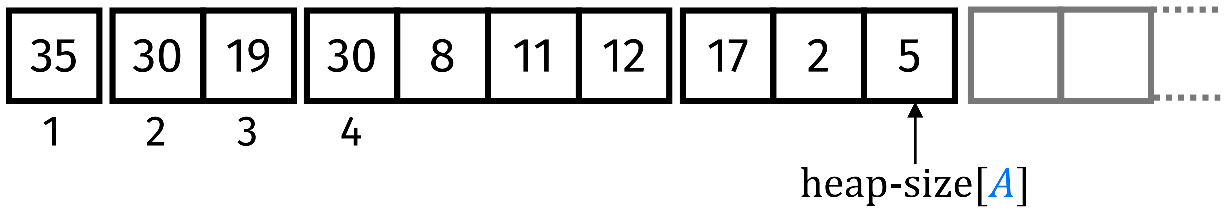
Operations

Insert(S, x):	inserts element x into S, that is, $S \leftarrow S \cup \{x\}$
Maximum(S):	returns the element of S with the largest key
Extract-Max(S):	removes and returns the element of S with the largest key
Increase-Key(S, x, k):	give $\mathrm{key}[x]$ the value k condition: k is larger than the current value of $\mathrm{key}[x]$

Implementing a max-priority queue

Set S is stored as a heap in an array A.

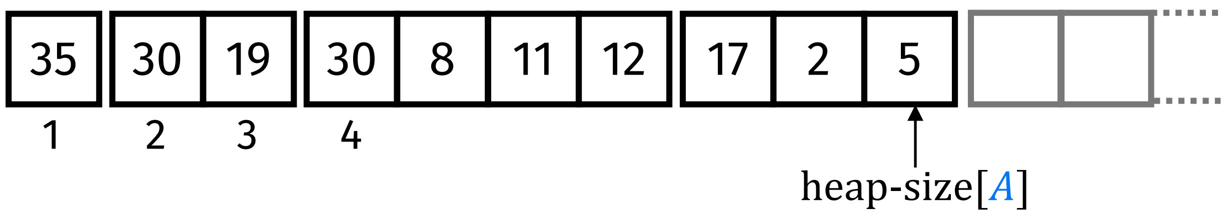
Operations: Maximum, Extract-Max, Insert, Increase-Key.



Implementing a max-priority queue

Set S is stored as a heap in an array A.

Operations: Maximum, Extract-Max, Insert, Increase-Key.



Maximum(A)

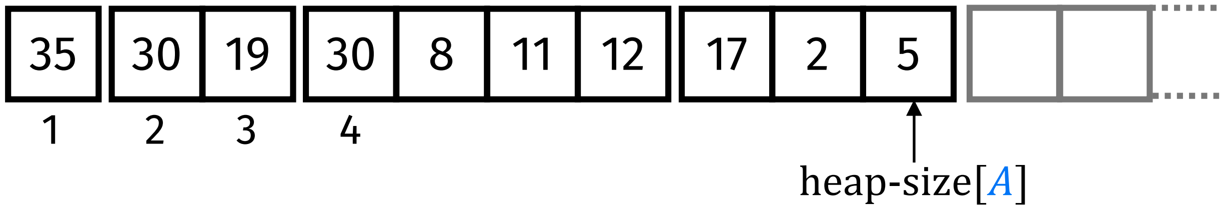
if heap-size$[A] < 1$
  then error
  else return $A[1]$

running time: $O(1)$

Implementing a max-priority queue

Set S is stored as a heap in an array A.

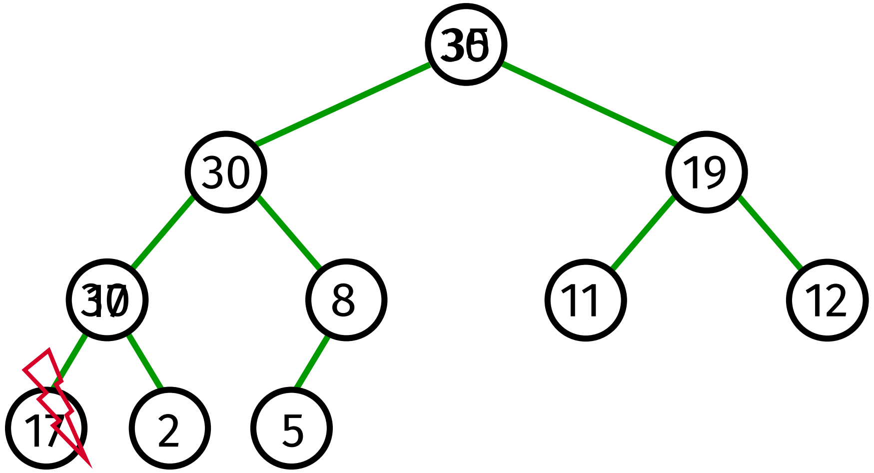
Operations: Maximum, Extract-Max, Insert, Increase-Key.



Implementing a max-priority queue

Set S is stored as a heap in an array A.

Operations: Maximum, Extract-Max, Insert, Increase-Key.



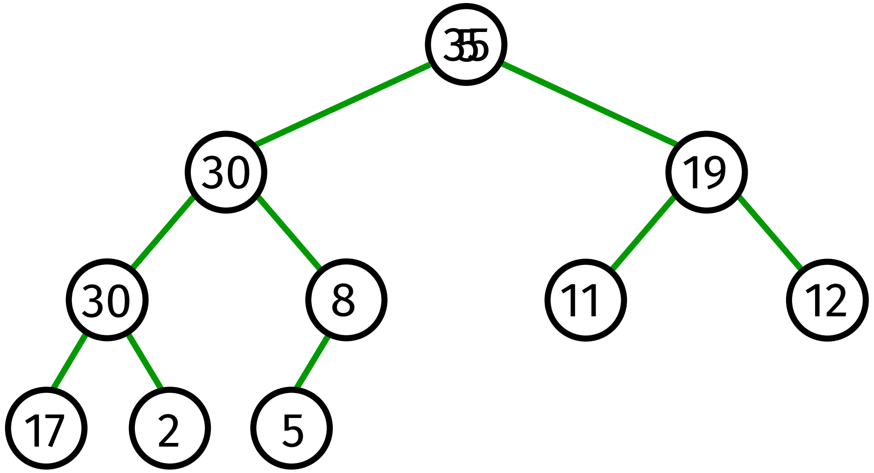
Implementing a max-priority queue

Set S is stored as a heap in an array A.

Operations: Maximum, Extract-Max, Insert, Increase-Key.

Heap-Extract-Max(A)

if heap-size$[A] < 1$
    then error
max $= A[1]$
$A[1] = A[\text{heap-size}[A]]$
heap-size$[A]$ = heap-size$[A] - 1$
Max-Heapify(A,1)
return max



restore heap property

Max-Heapify

Max-Heapify(A, i)

- ensures that the heap whose root is stored at position i has the max-heap property
- assumes that the binary subtrees rooted at $\mathrm{Left}(i)$ and $\mathrm{Right}(i)$ are max-heaps

Max-Heapify

Max-Heapify(A, i)

- ensures that the heap whose root is stored at position i has the max-heap property
- assumes that the binary subtrees rooted at $\mathrm{Left}(i)$ and $\mathrm{Right}(i)$ are max-heaps

Max-Heapify$(A,1)$

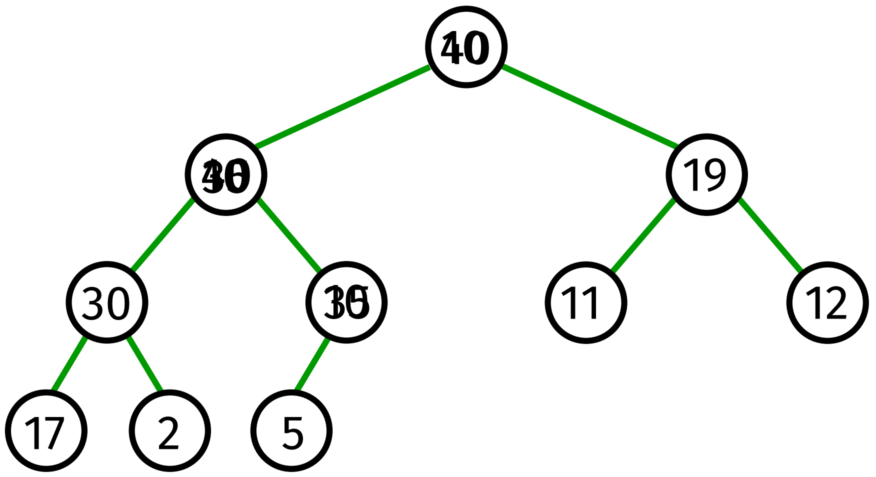
exchange $A[1]$ with largest child

Max-Heapify$(A,2)$

exchange $A[2]$ with largest child

Max-Heapify$(A,5)$

$A[5]$ larger than its children ➡ done.



Max-Heapify

Max-Heapify(A, i)

- ensures that the heap whose root is stored at position i has the max-heap property
- assumes that the binary subtrees rooted at $\mathrm{Left}(i)$ and $\mathrm{Right}(i)$ are max-heaps

if $\mathrm{Left}(i) \leq \text{heap-size}[A]$ and $A[\mathrm{Left}(i)] > A[i]$
    then $\text{largest} = \mathrm{Left}(i)$
    else $\text{largest} = i$
if $\mathrm{Right}(i) \leq \text{heap-size}[A]$ and $A[\mathrm{Right}(i)] > A[\text{largest}]$
    then $\text{largest} = \mathrm{Right}(i)$
if $\text{largest} \neq i$
    then exchange $A[i]$ and $A[\text{largest}]$
          Max-Heapify$(A, \text{largest})$

running time? O(height of the subtree rooted at i) $= O(\log n)$

Implementing a max-priority queue

Set S is stored as a heap in an array A.

Operations: Maximum, Extract-Max, Insert, Increase-Key.

Insert (A, key)

1. $\text{heap-size}[A] = \text{heap-size}[A] + 1$
2. $A[\text{heap-size}[A]] = -\infty$
3. Increase-Key$(A, \text{heap-size}[A], \text{key})$

Implementing a max-priority queue

Set S is stored as a heap in an array A.

Operations: Maximum, Extract-Max, Insert, Increase-Key.

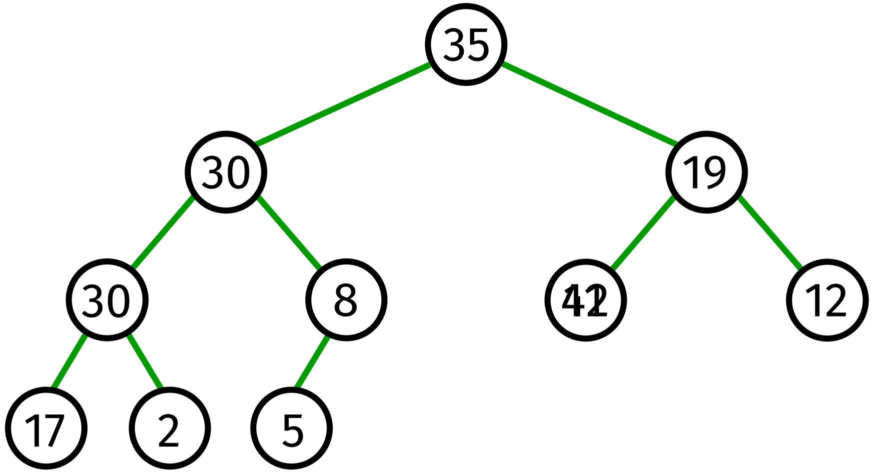
Implementing a max-priority queue

Set S is stored as a heap in an array A.

Operations: Maximum, Extract-Max, Insert, Increase-Key.

Increase-Key(A, i, key)

if $\text{key} < A[i]$
    then error
$A[i] = \text{key}$
while $i > 1$ and $A[\text{Parent}(i)] < A[i]$
        do exchange $A[\text{Parent}(i)]$ and $A[i]$
            $i = \text{Parent}(i)$



running time: $O(\log n)$

Building a heap

Build-Max-Heap(A)

- Input: array $A[1 \dots n]$ of numbers
- Output: array $A[1 \dots n]$ with the same numbers, but rearranged, such that the max-heap property holds

1. heap-size $= A.\text{length}$
2. **for** $i = A.\text{length}$ **downto** 1
3. **do** Max-Heapify(A, i)

starting at $\lfloor A.\text{length}/2 \rfloor$ is sufficient

Loop Invariant

$P(i)$: nodes $i+1, \dots, n$ are each the root of a max-heap

Maintenance

$P(i)$ holds before line 3 is executed,
$P(i-1)$ holds afterwards

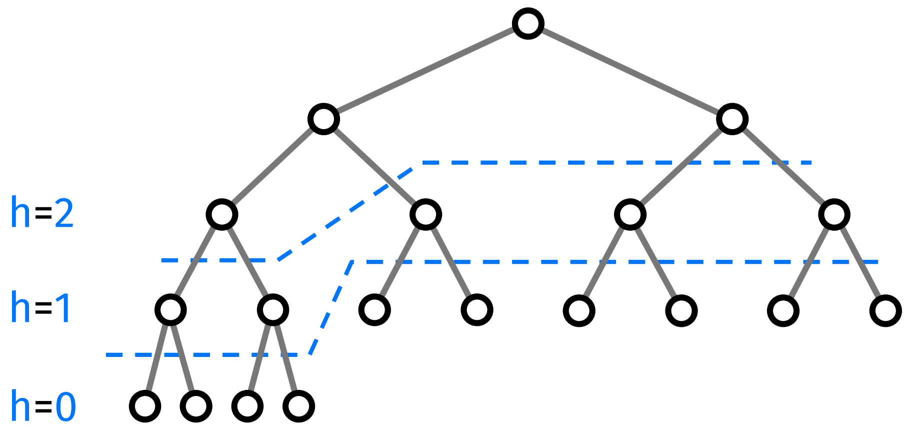
Building a heap

Build-Max-Heap(A)

1. heap-size $= A.\text{length}$
2. **for** $i = A.\text{length}$ **downto** 1
3. **do** Max-Heapify(A,i) → O(height of node i)

height of node i
edges longest simple downward path from i to a leaf.

$$\sum_i O(1 + \text{height of } i)$$
$$= \sum_{0 \le h \le \log n} (\#\text{ nodes of height } h) \cdot O(1+h)$$
$$= \sum_{0 \le h \le \log n} \left(\frac{n}{2^{h+1}}\right) \cdot O(1+h)$$
$$= O(n) \cdot \sum_{0 \le h \le \log n} \left(\frac{h}{2^{h+1}}\right)$$
$$= O(n)$$



The sorting problem

Input: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
Output: a permutation of the input such that $\langle a_{i1} \leq \cdots \leq a_{in} \rangle$

Important properties of sorting algorithms:

running time: how fast is the algorithm in the worst case

in place: only a constant number of input elements are ever stored outside the input array

Sorting with a heap: Heapsort

HeapSort(A)

1. Build-Max-Heap(A)
2. **for** $i = A.\text{length}$ **downto** 2
3. **do** exchange $A[1]$ and $A[i]$
4. $\text{heap-size}[A] = \text{heap-size}[A] - 1$
5. Max-Heapify(A,1)

Loop invariant

$P(i)$: $A[i+1 \ldots n]$ is sorted and contains the $n - i$ largest elements,
$A[1 \ldots i]$ is a max-heap on the remaining elements

Maintenance

$P(i)$ holds before lines 3-5 are executed,
$P(i-1)$ holds afterwards

Running time: $O(n \log n)$

Sorting algorithms

	worst case running time	in place
InsertionSort	$\Theta(n^2)$	yes
MergeSort	$\Theta(n \log n)$	no
HeapSort	$\Theta(n \log n)$	yes