DESIGN AND ANALYSIS OF ALGORITHMS

CS 4120/5120

SORTING - HEAPSORT

AGENDA

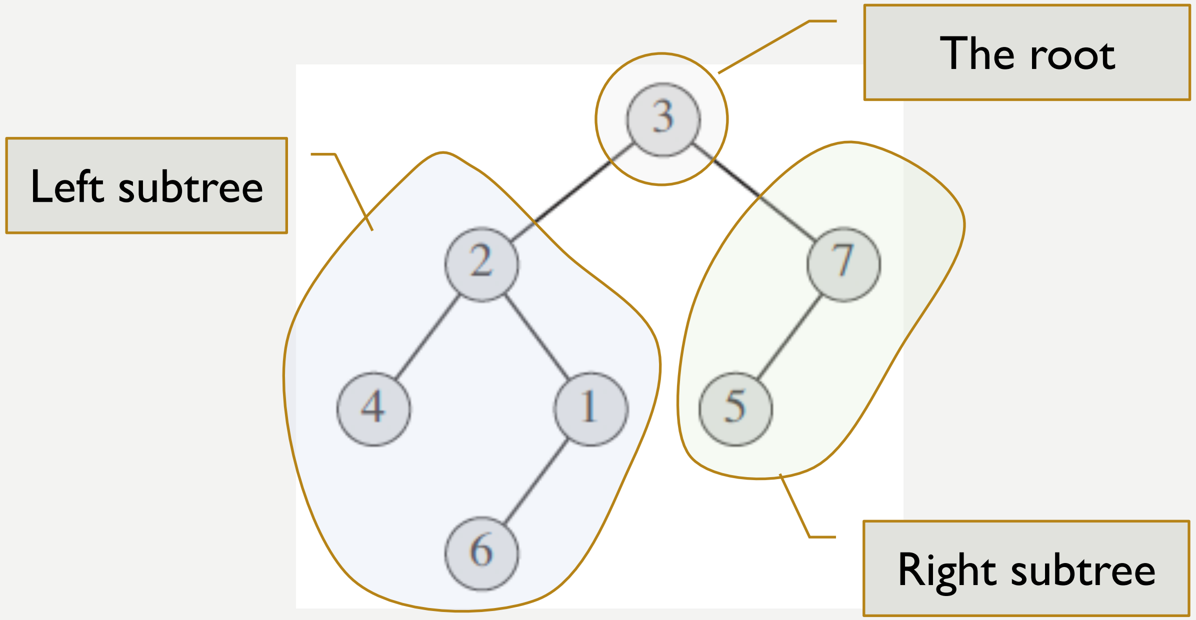
- Data structure
  - Binary tree
  - Heap
- Max-heapify
- Building heap
- Heapsort

HEAPSORT

- A new algorithm design technique: using a **data structure**.
  - Example
    - Solve binary search problem by constructing a binary search tree.
- Heapsort sorts a given array using a data structure called the ***heap***.

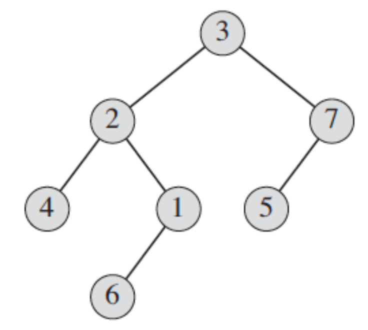
BINARY TREE

- A ***binary tree*** T is a structure defined on a finite set of nodes that either
  - contains no nodes, or
  - is composed of three disjoint set of nodes:
    - a ***root*** node,
    - a binary tree called its ***left subtree***, and
    - a binary tree called its ***right subtree***.
- Each node of a binary tree has a **degree** no more than 2.

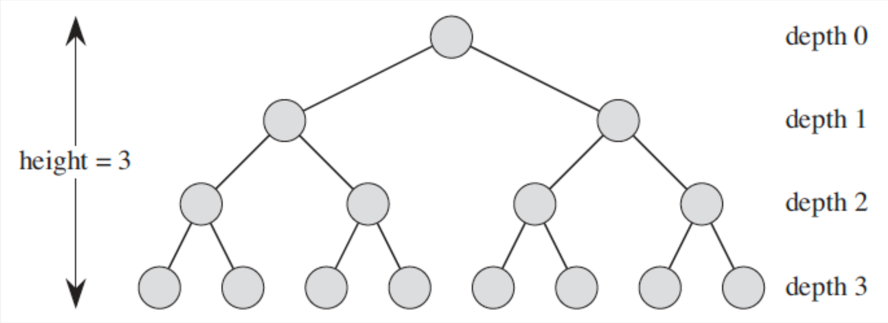


BINARY TREE TERMS

- The value of a node is referred to as the ***key*** of the node.
- The ***height*** of a binary tree with n nodes is the deepest level
  - $\text{dep}th = \lg n$, where $\lg n$ is the height of the tree.

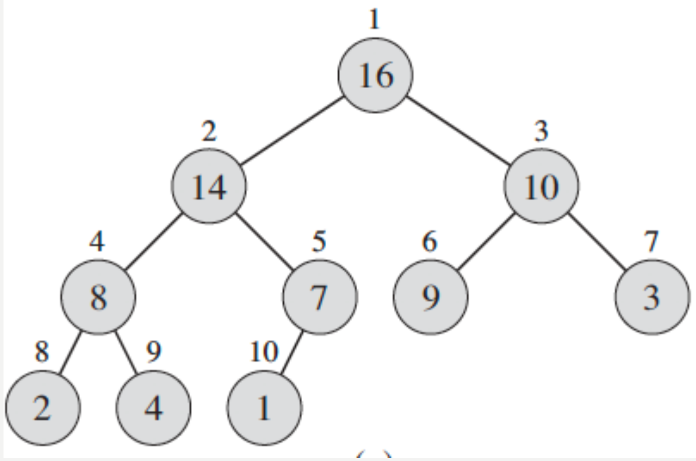


- A ***complete*** binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



HEAP
NEARLY COMPLETE BINARY TREE

- A heap can be implemented by an array, denoted as A.
  - The array has two attributes
    - $\boldsymbol{A.length}$: the number of elements in the array, and
    - $\boldsymbol{A.heap\text{-}size}$: the number of elements in the heap that are stored within array A.
      - Only the elements in $A[\mathbf{1} \,..\, A.heap\text{-}size]$, where $0 \leq A.heap\text{-}size \leq A.length$ are valid elements of the heap.
    - Note that element $A[1]$ is always in the heap.



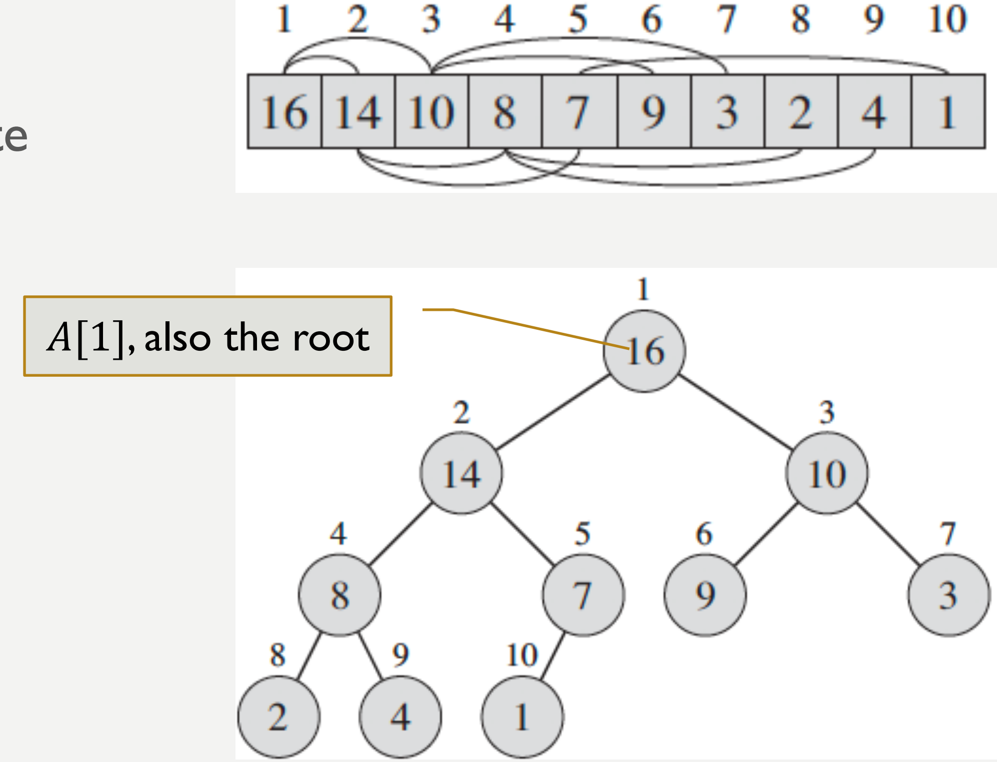
HEAP AND ARRAY

- Element $A[1]$ is the root.
- Given the index i of a node, we can easily compute **the indices of** its parent, left, and right child.

PARENT (i)
return $\lfloor i/2 \rfloor$

LEFT (i)
return $2i$

RIGHT (i)
return $2i + 1$

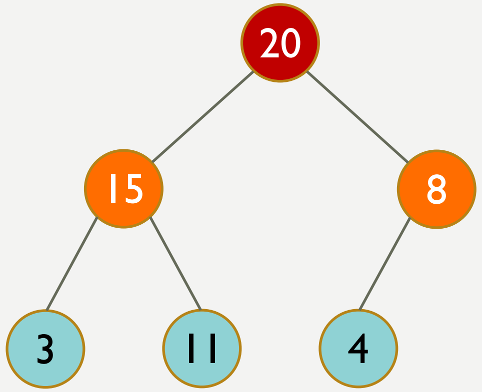


The **equivalent** array and tree representations.

HEAP PROPERTIES
MAX-HEAP

- For every node i other than the root, $A[PARENT(i)] \geq A[i]$
  - Facts
    - The value of a node is **at most** the value of its parents.
    - Element $\boldsymbol{A[1]}$ **is the largest element** in a **max**-heap and is stored at the root.
    - The subtree rooted at a node contains values no larger than that contained at the node itself.
- **The choice of this course.**

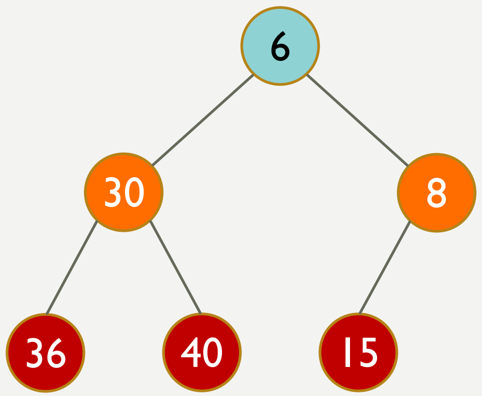
PARENT (i)
return $\lfloor i/2 \rfloor$



HEAP PROPERTIES
MIN-HEAP

- For every node i other than the root, $A[PARENT(i)] \leq A[i]$
  - Facts
    - The value of a node is **at least** the value if its parents.
    - Element $\boldsymbol{A[1]}$ **is the smallest element** in a **min**-heap and is stored at the root.
    - The subtree rooted at a node contains values no smaller than that contained at the node itself.

PARENT (i)
return $\lfloor i/2 \rfloor$



HEAP PROPERTIES
PRACTICE

- Consider the following array implementation of heaps. Fill out the blanks. Fill NA if necessary.
  - $A.heap\text{-}size =$ __________.
    - $A[\text{PARENT}(7)] =$ _____.
    - Determine the $heap\text{-}size$ of the array by finding the far-right element that does not maintain heap properties.

Index	1	2	3	4	5	6	7	8	9	10	11	12
$A[12]$	1	6	7	11	20	16	40	22	18	19	31	50

HEAP PROPERTIES
PRACTICE

- Consider the following array implementation of heaps. Fill out the blanks. Fill NA if necessary.
  - $B.heap\text{-}size =$ ________.
    - $B[\text{RIGHT}(4)] =$ ____.

Index	1	2	3	4	5	6	7	8	9	10
$B[10]$	44	39	37	26	2	30	22	3	25	0

HEAP PROPERTIES
PRACTICE

- Consider the following array implementation of heaps. Fill out the blanks. Fill NA if necessary.
  - $C.heap\text{-}size =$ ________.
    - $C[\text{LEFT}(6)] =$ ____.

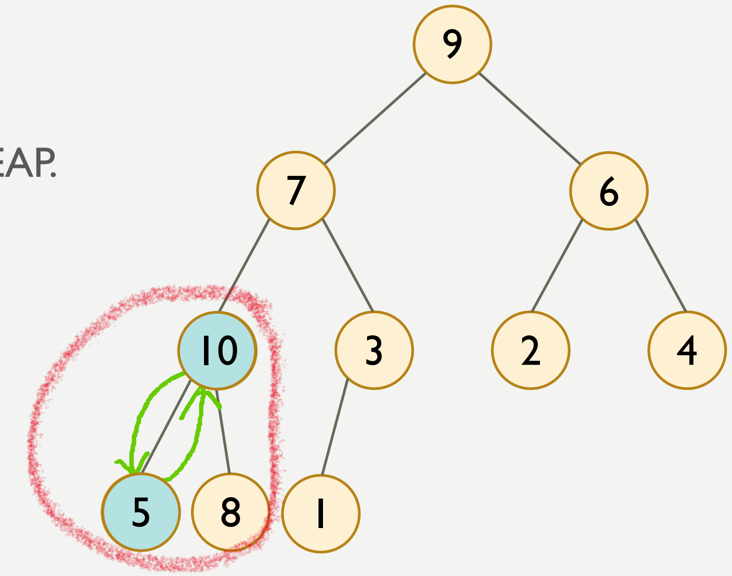
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$C[15]$	35	26	24	23	19	17	14	12	6	11	3	13	4	20	25

MAX-HEAPIFY

- Consider the example below

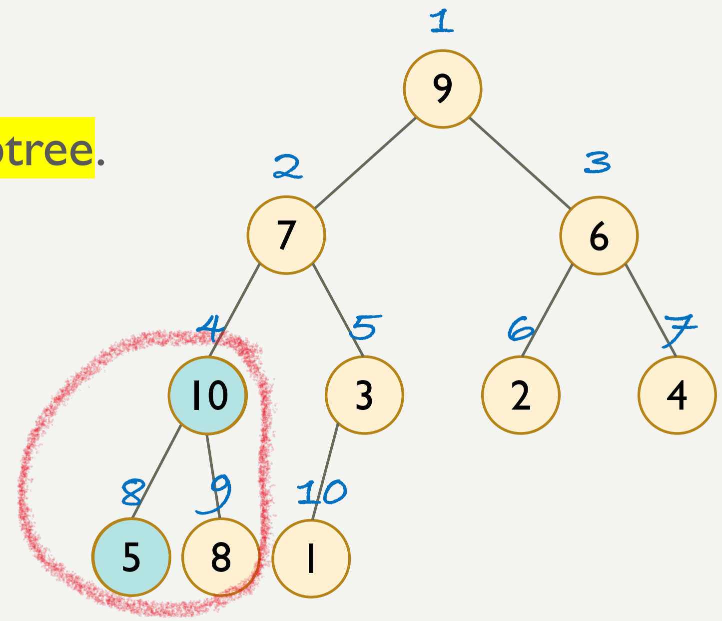
Index	1	2	3	4	5	6	7	8	9	10
$A[10]$	9	7	6	5	3	2	4	10	8	1

  - Does A qualify as a MAX-HEAP?
    - No, the subtree rooted at $A[$ _____ $] =$ _______ is not a MAX-HEAP.
  - How to locally rearrange array elements to transform the subtree into a MAX-HEAP?
  - Is the resulting array a MAX-HEAP?



THE MAX-HEAPIFY PROCEDURE

- Algorithm: MAX-HEAPIFY
  - Input: An array $A[1..n]$, an index i into the array.
  - Output: A the subtree rooted at i is a max-heap.
- Goal: The **max-heap properties are preserved** for the subtree.
- In the previous example, we can think of it as the result of calling MAX-HEAPIFY $(A, 4)$.



THE MAX-HEAPIFY ALGORITHM

- Input: array A and index i into array A.
  - **Find** the largest of a parent-children structure
    - the parent (the node indexed by i)
    - the left child (the node indexed by $2i$ if it exists)
    - the right child (the node indexed by $2i + 1$ if it exists)
  - **Put** the largest as the parent
    - Recurse to maintain max-heap property if possible
- Output?

MAX-HEAPIFY (A, i)
l = LEFT (i)
r = RIGHT (i)
if $l \leq A.leap\text{-}size$ and $A[l] > A[i]$
$\quad\quad largest = l$
else $largest = i$
if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$
$\quad\quad largest = r$
if $largest \neq i$
$\quad\quad$ exchange $A[i]$ with $A[largest]$
$\quad\quad$ MAX-HEAPIFY $(A, largest)$

THE MAX-HEAPIFY ALGORITHM
IN ACTION

- Perform MAX-HEAPIFY$(A, 4)$ on the instance. Show the recursions in the order of their invocations. Assume $A.length = A.leap\text{-}size$.

Index	1	2	3	4	5	6	7	8	9	10
$A[10]$	9	7	6	5	3	2	4	10	8	1

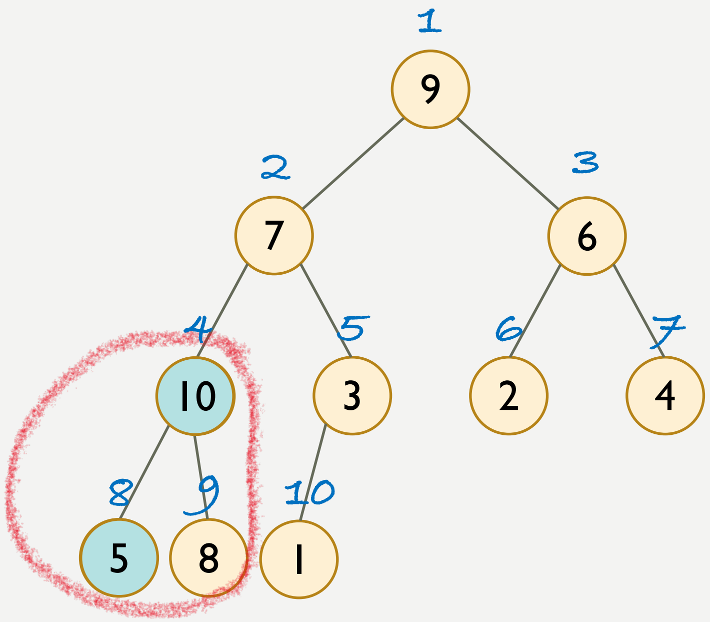
- What is the resulting array?

Index	1	2	3	4	5	6	7	8	9	10
$A[10]$	9	7	6	10	3	2	4	5	8	1

MAX-HEAPIFY (A, i)
$l =$ LEFT (i)
$r =$ RIGHT (i)
if $l \leq A.leap\text{-}size$ and $A[l] > A[i]$
$largest = l$
else $largest = i$
if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$
$largest = r$
if $largest \neq i$
exchange $A[i]$ with $A[largest]$
MAX-HEAPIFY $(A, largest)$

THE MAX-HEAPIFY ALGORITHM

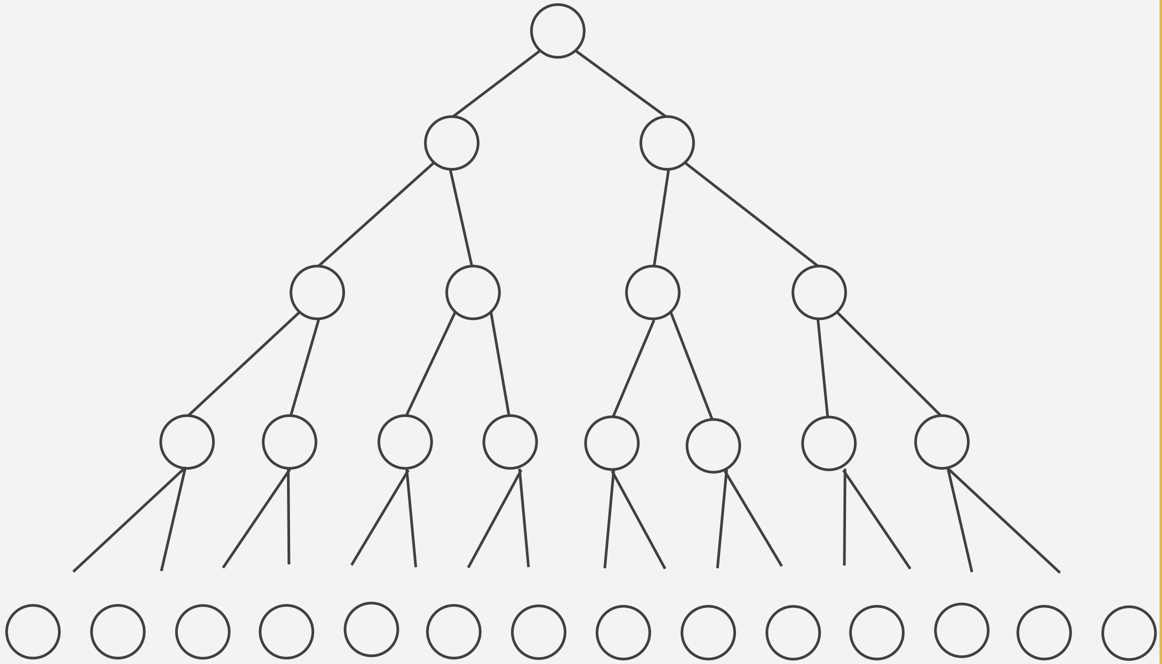
- Input: array A and index i into array A.
- Output: **The subtree rooted at i is a max-heap.**



MAX-HEAPIFY (A, i)
$l =$ LEFT (i)
$r =$ RIGHT (i)
if $l \leq A.leap\text{-}size$ and $A[l] > A[i]$
$\quad largest = l$
else $largest = i$
if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$
$\quad largest = r$
if $largest \neq i$
$\quad$ exchange $A[i]$ with $A[largest]$
$\quad$ MAX-HEAPIFY $(A, largest)$

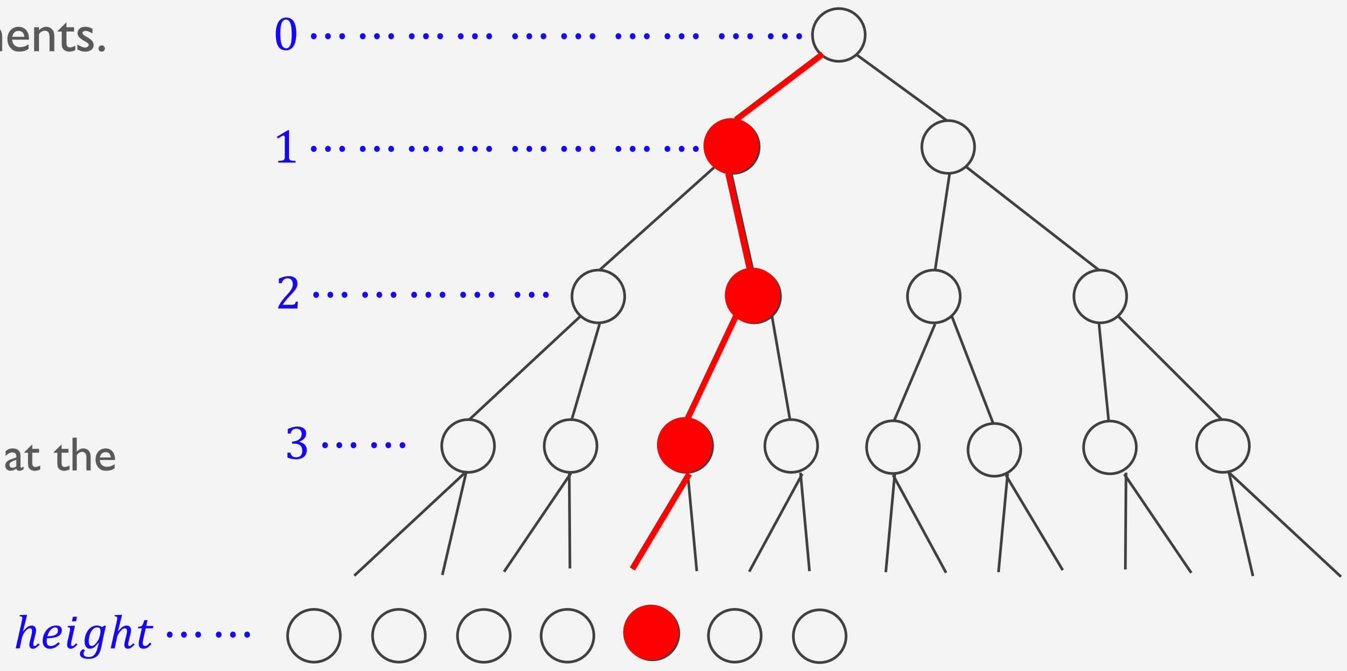
THE MAX-HEAPIFY ALGORITHM
RUNNING TIME ANALYSIS - HEAP

- Consider a max-heap with n elements.
  - If the max-heap happens to be a ***complete*** binary tree with the bottom level **completely filled**, the height of the tree is $\underline{\mathbf{lg}(\boldsymbol{n}+\mathbf{1})-\mathbf{1}}$.
  - There are $\underline{(\boldsymbol{n}+\mathbf{1})/\mathbf{2}}$ nodes at the bottom level.



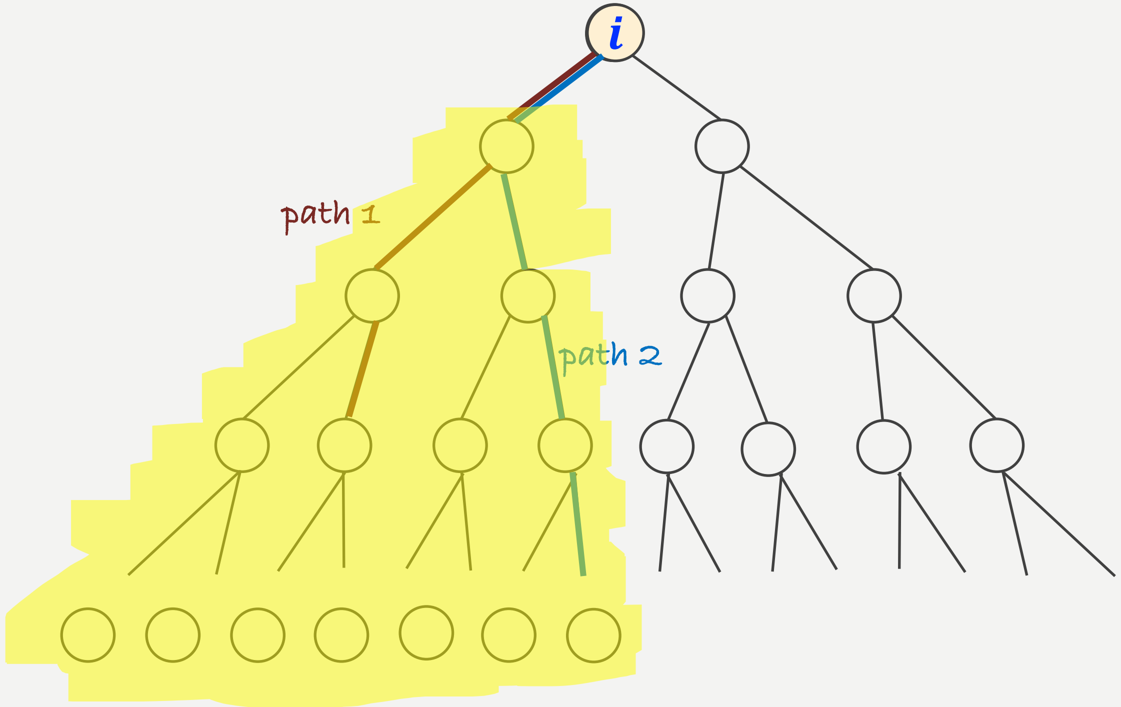
THE MAX-HEAPIFY ALGORITHM RUNNING TIME ANALYSIS - HEAP

- Consider a max-heap with n elements.
  - If the max-heap happens to be ***complete*** binary tree with the bottom level exactly ***half filled***, the height of the tree is $\underline{\ \mathbf{lg(2(\boldsymbol{n}+1)/3)}\ }$.
  - There are $\underline{\ \mathbf{(\boldsymbol{n}+1)/3}\ }$ nodes at the bottom level.



THE MAX-HEAPIFY ALGORITHM WORST-CASE RUNNING TIME

- What is the **worst**-case scenario?
  - Call MAX-HEAPIFY $(A,$ ___ $)$
  - The original element $A[i]$ floats down to ______________.
  - In other words, the algorithm recurses on the shaded subtree in the **worst**-case scenario
    - Recursing on the bigger subtree



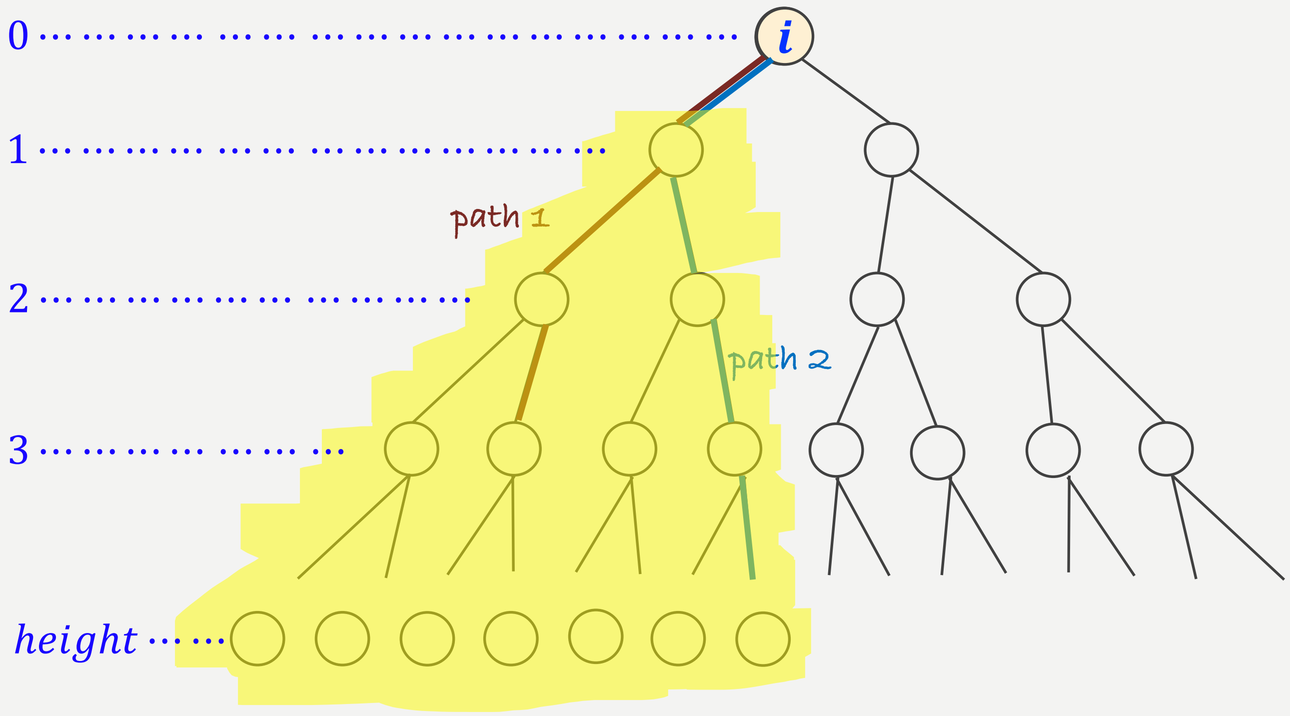
THE MAX-HEAPIFY ALGORITHM WORST-CASE RUNNING TIME

- In the **worst**-case scenario
  - The algorithm recurses on the shaded subtree.
  - The # of nodes in the shaded subtree is

$$\sum_{k=1}^{height} 2^{k-1} = 2^{1-1} \cdot \frac{2^{height} - 1}{2 - 1}$$

$$= 1 \cdot 2^{\lg\left(\frac{2(n+1)}{3}\right)} - 1$$

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

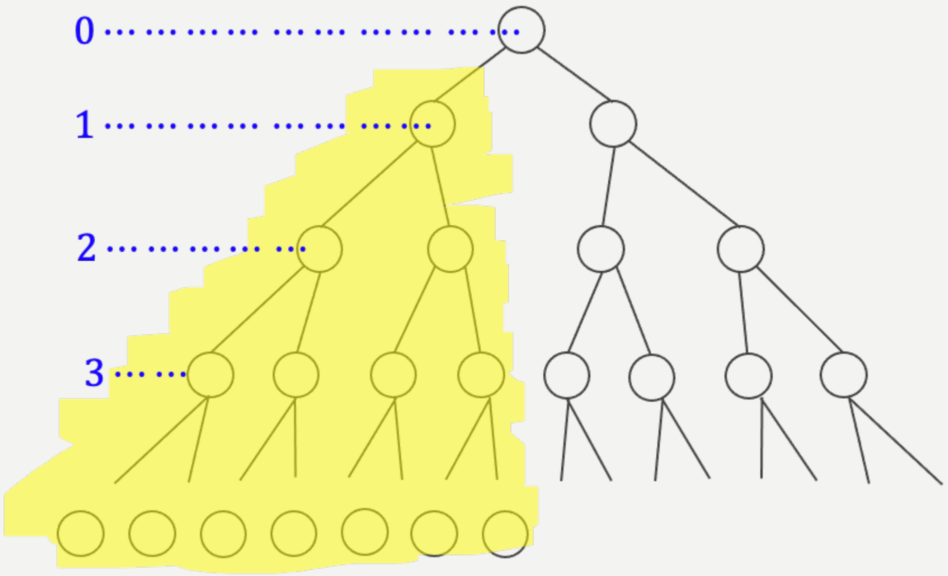


THE MAX-HEAPIFY ALGORITHM RUNNING TIME FUNCTION

- In the **worst**-case scenario, the algorithm will recurse on at most ________ nodes of the tree.
- The running time function can be formulated as

  $T(n) \leq T(2n/3) + \Theta(1)$

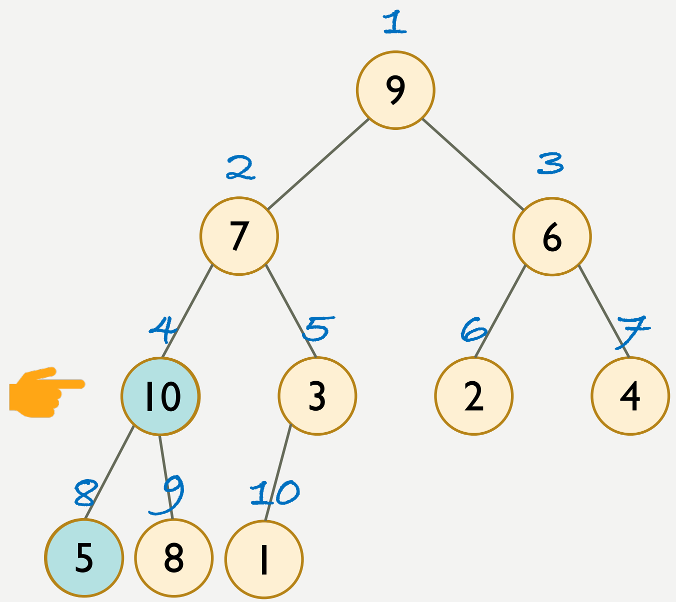
- $T(n) = \boldsymbol{O}(\mathbf{lg}\, \boldsymbol{n})$



MAX-HEAPIFY REVIEW

- MAX-HEAPIFY (A, i)
  - Outcome
    - The subtree rooted at $A[i]$ is a max-heap.
  - Locally max-heapified

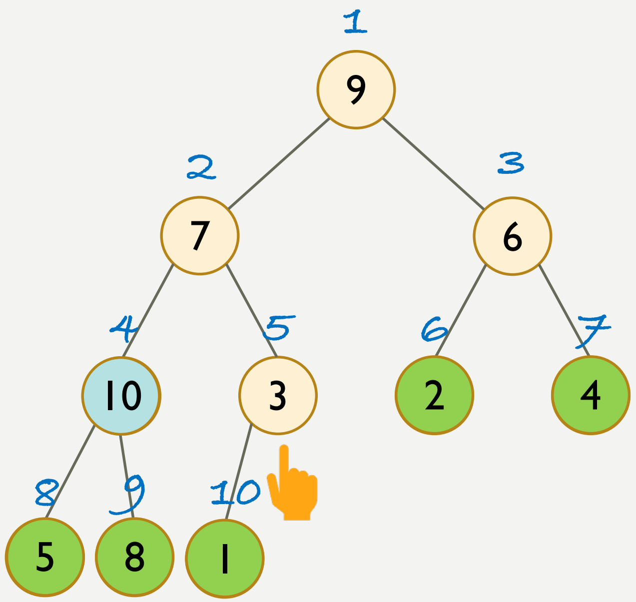
- How to make the array globally max-heapified?
- Which element to begin with?
  - 5, 8, 1, 2, or 4?



BUILD A HEAP

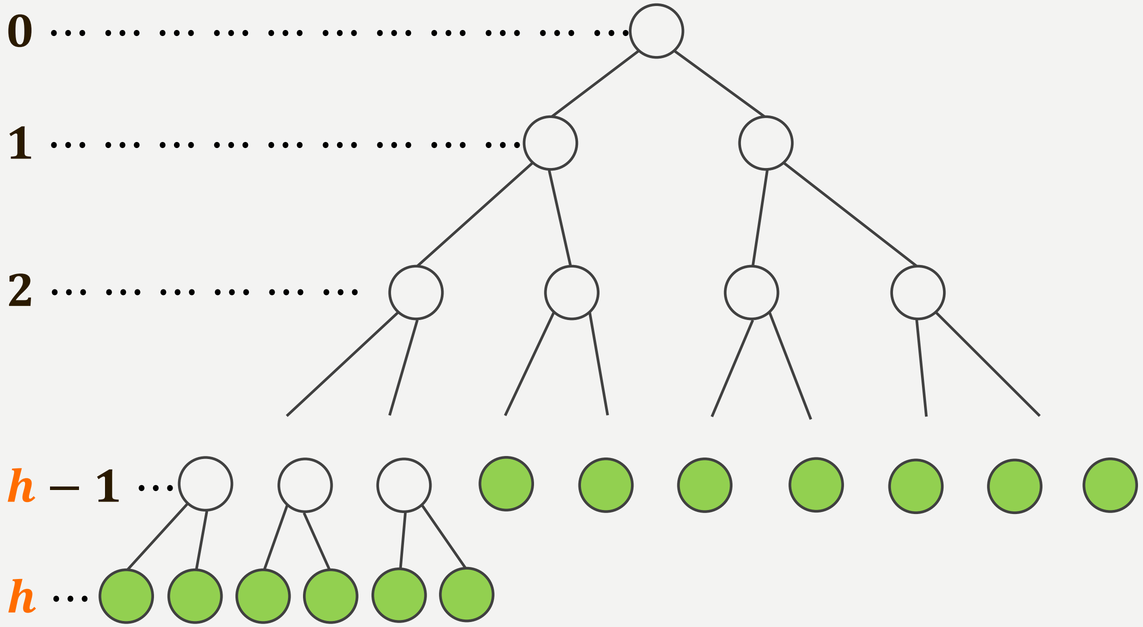
- To build a heap out of an array, there is no point in MAX-HEAPIFYing the leaves.
- The most efficient way is to **start** with the far-right **non-leaf** nodes.
- Consider the equivalent implementations shown on the right. The far-right **non-leaf** node is indexed by _________.
- What about when $A.length = n$?

Index	1	2	3	4	5	6	7	8	9	10
$A[10]$	9	7	6	10	3	2	4	5	8	1



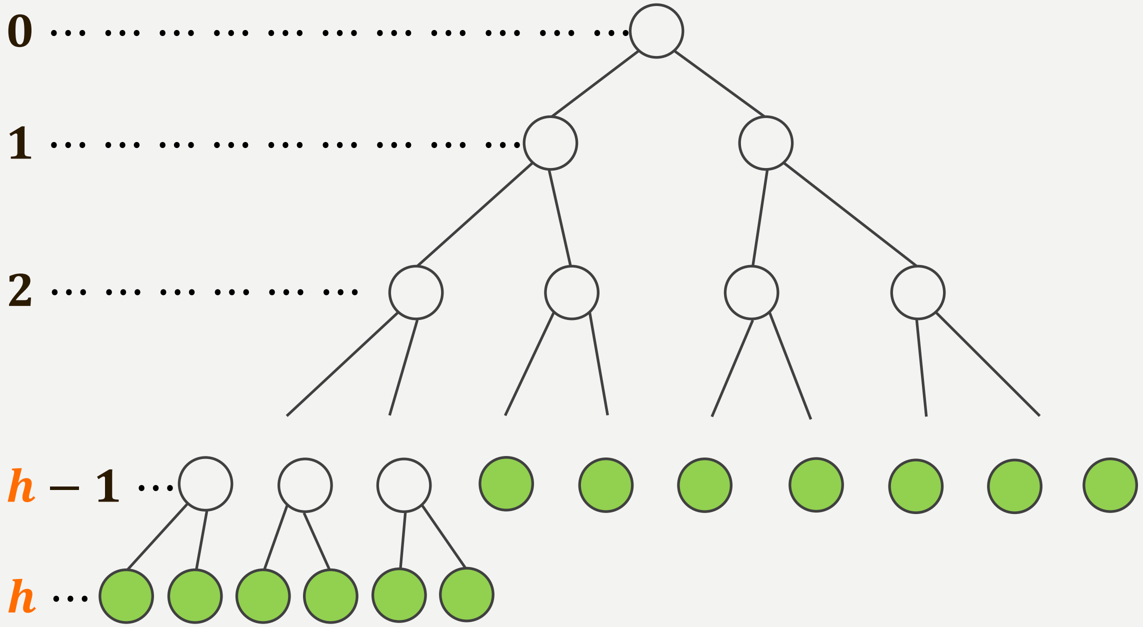
BUILD A HEAP
START AT $A[\lfloor n/2 \rfloor]$

- Consider an array implementation $A[1..n]$ of a MAX-HEAP.
  - The far-right **non-leaf** node is indexed by $\lfloor \boldsymbol{n/2} \rfloor$.
    - If you are interested in the work, you may try to answer the questions below.
      - The # of nodes of levels $0 \sim \boldsymbol{h} - \mathbf{1}$ is _____.
      - There are _____ **leaves** remaining at level $\boldsymbol{h}$.
      - There are _____ **leaves** at level $\boldsymbol{h} - \mathbf{1}$.
      - The far-right ______ elements are **leaves**.



BUILD A HEAP
START AT $A[\lfloor n/2 \rfloor]$

- Consider an array implementation $A[1..n]$ of a MAX-HEAP.
  - The far-right **non-leaf** node is indexed by $\lfloor \boldsymbol{n/2} \rfloor$.
    - If you are interested in the work, you may try to answer the questions below.
      - The # of nodes of levels $0 \sim \boldsymbol{h-1}$ is _____.
      - There are _____ **leaves** remaining at level $\boldsymbol{h}$.
      - There are _____ **leaves** at level $\boldsymbol{h-1}$.
      - The far-right ______ elements are **leaves**.



BUILDING A HEAP
IN ACTION

- Use the MAX-HEAPIFY procedure in a **bottom-up manner** to convert an array $A[1..n]$, where $n = A.length$, into a max-heap.
  - Starting at $A[\lfloor n/2 \rfloor]$.
- Instance
  - Given a random array. Build a MAX-HEAP out of the array.
    - Input

Index	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

    - Output

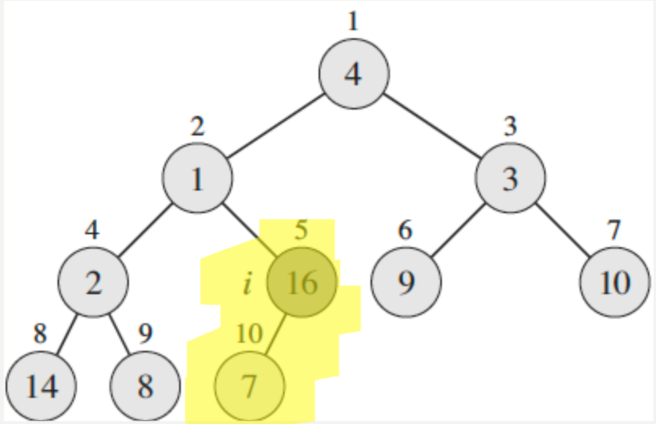
Index	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1

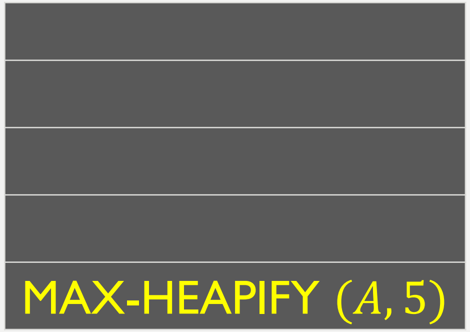
BUILDING A HEAP
IN ACTION

- Build a MAX-HEAP out of the given array.

Index	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

- Show the **changes in the stack** for each iteration.
  - Iteration I
    - Stack_push MAX-HEAPIFY $(A, 5)$
    - Stack_pop MAX-HEAPIFY $(A, 5)$



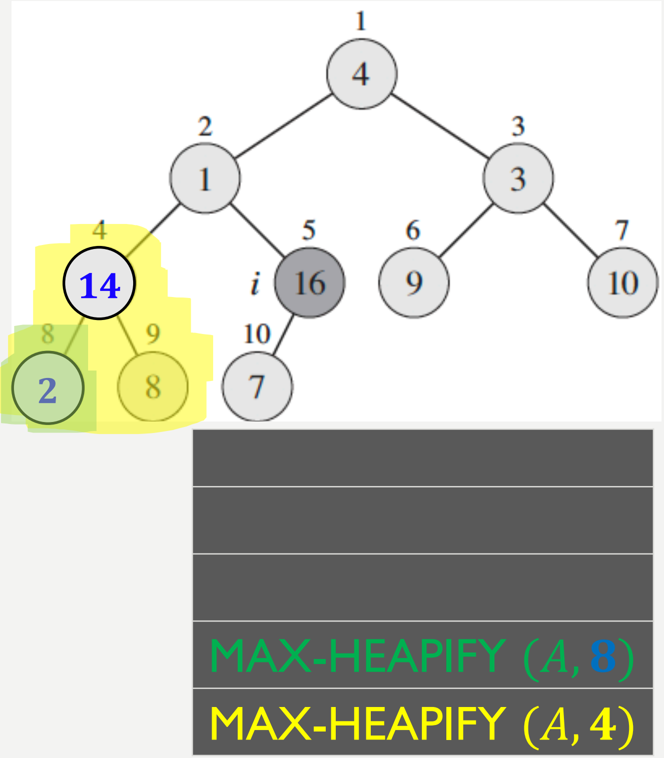


BUILDING A HEAP
IN ACTION

- Build a MAX-HEAP out of the given array.

Index	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

- Show the **changes in the stack** for each iteration.
  - Iteration **2**
    - Stack_push MAX-HEAPIFY $(A, \mathbf{4})$
    - Stack_push MAX-HEAPIFY $(A, \mathbf{8})$
    - Stack_pop MAX-HEAPIFY $(A, \mathbf{8})$
    - Stack_pop MAX-HEAPIFY $(A, \mathbf{4})$

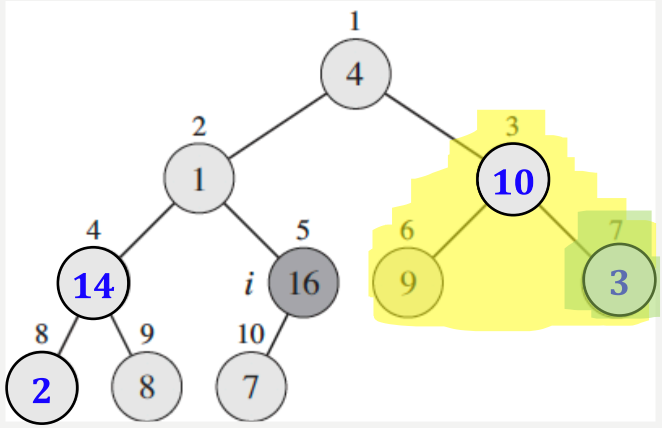


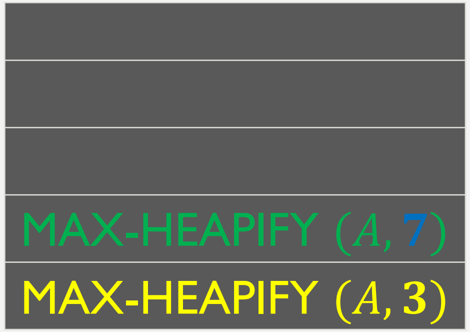
BUILDING A HEAP
IN ACTION

- Build a MAX-HEAP out of the given array.

Index	1	2	3	4	5	6	7	8	9	10
A	4	1	3	14	16	9	10	2	8	7

- Show the **changes in the stack** for each iteration.
  - Iteration **3**
    - Stack_push MAX-HEAPIFY $(A, 3)$
    - Stack_push MAX-HEAPIFY $(A, 7)$
    - Stack_pop MAX-HEAPIFY $(A, 7)$
    - Stack_pop MAX-HEAPIFY $(A, 3)$



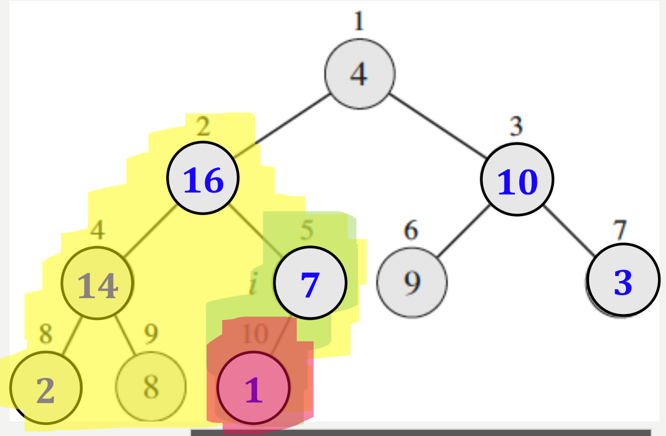


BUILDING A HEAP
IN ACTION

- Build a MAX-HEAP out of the given array.

Index	1	2	3	4	5	6	7	8	9	10
A	4	1	10	14	16	9	3	2	8	7

- Show the **changes in the stack** for each iteration.
  - Iteration **4**
    - Stack_push MAX-HEAPIFY $(A, \mathbf{2})$
    - Stack_push MAX-HEAPIFY $(A, \mathbf{5})$
    - Stack_push MAX-HEAPIFY $(A, \mathbf{10})$
    - Stack_pop MAX-HEAPIFY $(A, \mathbf{10})$
    - Stack_pop MAX-HEAPIFY $(A, \mathbf{5})$
    - Stack_pop MAX-HEAPIFY $(A, \mathbf{2})$



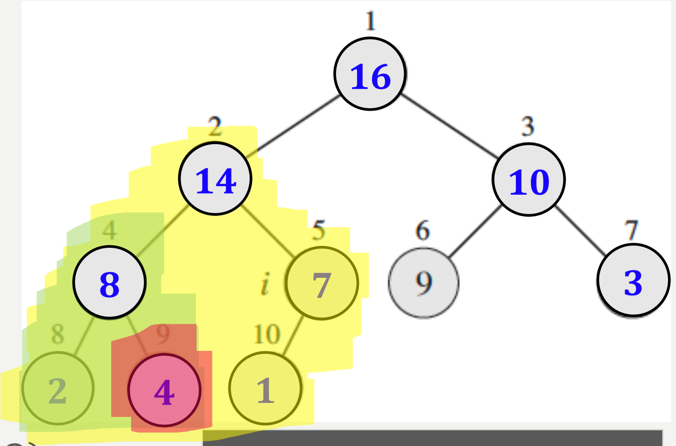
MAX-HEAPIFY $(A, \mathbf{10})$
MAX-HEAPIFY $(A, \mathbf{5})$
MAX-HEAPIFY $(A, \mathbf{2})$

BUILDING A HEAP
IN ACTION

- Build a MAX-HEAP out of the given array.

Index	1	2	3	4	5	6	7	8	9	10
A	4	16	10	14	7	9	3	2	8	1

- Show the **changes in the stack** for each iteration.
  - Iteration **5**
    - Stack_push MAX-HEAPIFY $(A, 1)$
    - Stack_push MAX-HEAPIFY $(A, 2)$
    - Stack_push MAX-HEAPIFY $(A, 4)$
    - Stack_push MAX-HEAPIFY $(A, 9)$
    - Stack_pop MAX-HEAPIFY $(A, 9)$
    - Stack_pop MAX-HEAPIFY $(A, 4)$
    - Stack_pop MAX-HEAPIFY $(A, 2)$
    - Stack_pop MAX-HEAPIFY $(A, 1)$



MAX-HEAPIFY $(A, 9)$
MAX-HEAPIFY $(A, 4)$
MAX-HEAPIFY $(A, 2)$
MAX-HEAPIFY $(A, 1)$

THE BUILD-MAX-HEAP ALGORITHM

- Input: Array $A[1..n]$
- The running time function $T(n) =$

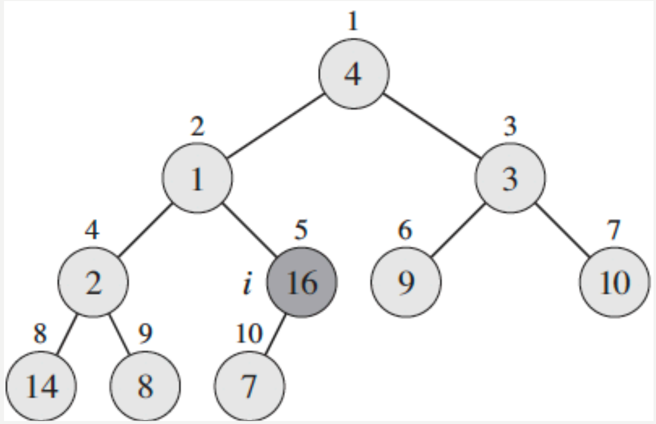
BUILD-MAX-HEAP (A)		Cost	Time
1	$A.heap\text{-}size = A.length$	$\Theta(1)$	1
2	**for** $i = \lfloor A.length/2 \rfloor$ **downto** 1	$\Theta(1)$	$\lfloor n/2 \rfloor + 1$
3	MAX-HEAPIFY (A, i)	$O(\lg n)$	$\lfloor n/2 \rfloor$

- The asymptotic upper-bound of $T(n)$ is $\boldsymbol{T(n) = O(n \lg n)}$

THE OUTPUT OF THE BUILD-MAX-HEAP ALGORITHM

- Before BUILD-MAX-HEAP (A)

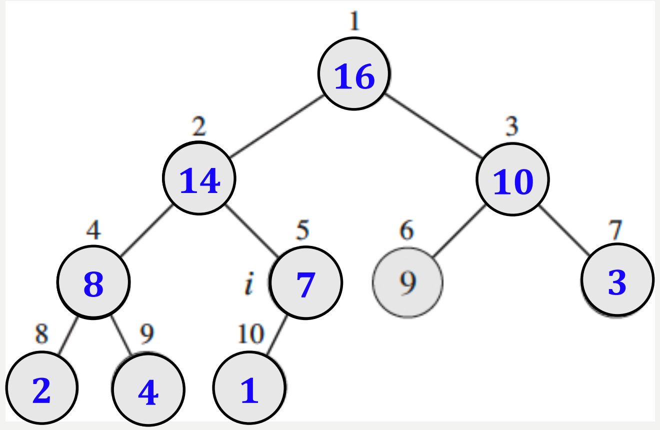
Index	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



Is the resulting array sorted?

- After BUILD-MAX-HEAP (A)

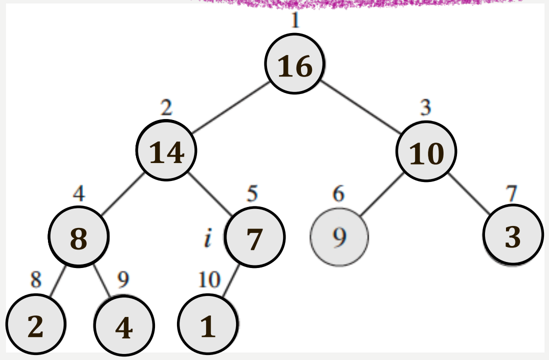
Index	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1



HEAPSORT USING THE BUILD-MAX-HEAP ALGORITHM

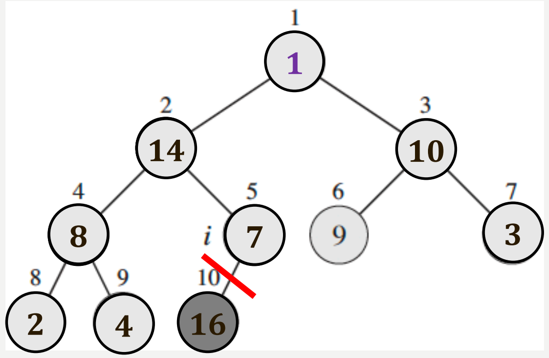
- The array is **nearly/partially** sorted.
- $A[1]$ is **the largest** number in the array.

Index	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1



Remove $A[1]$

Index	1	2	3	4	5	6	7	8	9	10
A	1	14	10	8	7	9	3	2	4	16



THE HEAPSORT PROCEDURE
STARTUP

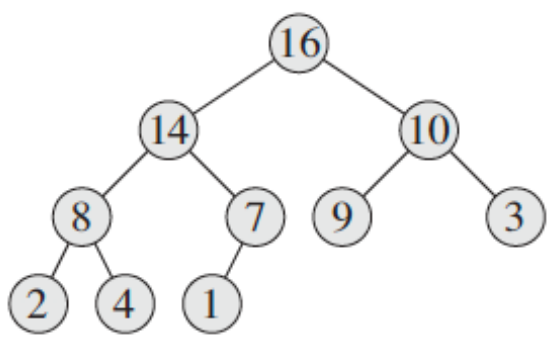
- Input, $i = A.length = \mathbf{10}$

Index	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

- BUILD-MAX-HEAP(A)

Index	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1

- $A.heap\text{-}size = A.length = \mathbf{10}$



(a)

THE HEAPSORT PROCEDURE
ITERATION 1

- Iteration **1**, $i = A.length - \mathbf{0} = \mathbf{10}$

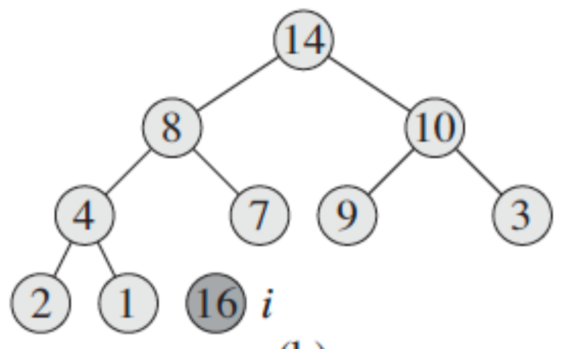
Index	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1

- "Remove" $A[1]$ from array. $A.heap\text{-}size = A.heap\text{-}size - 1 = \mathbf{9}$

Index	1	2	3	4	5	6	7	8	9	10
A	1	14	10	8	7	9	3	2	4	16

- MAX-HEAPIFY$(A, \mathbf{1})$

Index	1	2	3	4	5	6	7	8	9	10
A	14	8	10	4	7	9	3	2	1	16



(b)

THE HEAPSORT PROCEDURE
ITERATION 2

- Iteration **2**, $i = A.length - \mathbf{1} = \mathbf{9}$

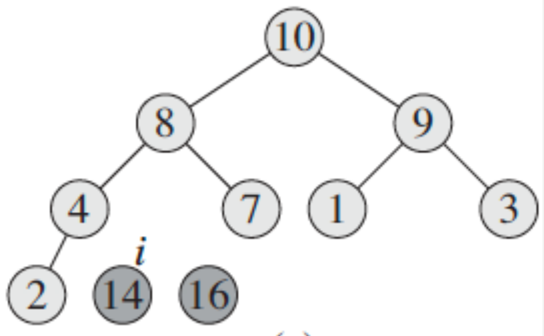
Index	1	2	3	4	5	6	7	8	9	10
A	14	8	10	4	7	9	3	2	1	16

- "Remove" $A[1]$ from array. $A.heap\text{-}size = A.heap\text{-}size - 1 = \mathbf{9} - 1 = \mathbf{8}$

Index	1	2	3	4	5	6	7	8	9	10
A	1	8	10	4	7	9	3	2	14	16

- MAX-HEAPIFY$(A, \mathbf{1})$

Index	1	2	3	4	5	6	7	8	9	10
A	10	8	9	4	7	1	3	2	14	16



(c)

THE HEAPSORT PROCEDURE
ITERATION 3

- Iteration **3**, $i = A.length - \mathbf{2} = 8$

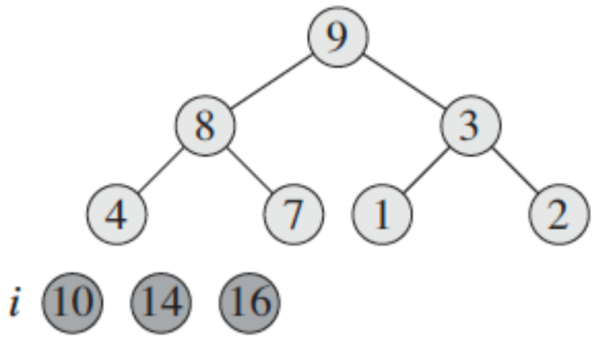
Index	1	2	3	4	5	6	7	8	9	10
A	10	8	9	4	7	1	3	2	14	16

- "Remove" $A[1]$ from array. $A.heap\text{-}size = A.heap\text{-}size - 1 = \mathbf{8} - 1 = \mathbf{7}$

Index	1	2	3	4	5	6	7	8	9	10
A	2	8	9	4	7	1	3	10	14	16

- MAX-HEAPIFY$(A, \mathbf{1})$

Index	1	2	3	4	5	6	7	8	9	10
A	9	8	3	4	7	1	2	10	14	16



(d)

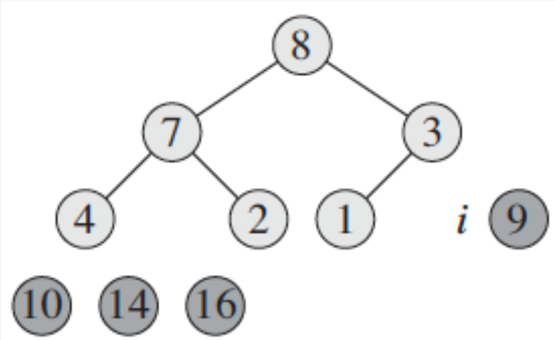
THE HEAPSORT PROCEDURE
ITERATION 4

- Iteration **4**, $i = A.length - \mathbf{3} = \mathbf{7}$

Index	1	2	3	4	5	6	7	8	9	10
A	9	8	3	4	7	1	2	10	14	16

- "Remove" $A[1]$ from array. $A.heap\text{-}size = A.heap\text{-}size - 1 = \mathbf{7} - 1 = \mathbf{6}$

Index	1	2	3	4	5	6	7	8	9	10
A	2	8	3	4	7	1	9	10	14	16



(e)

- MAX-HEAPIFY$(A, 1)$

Index	1	2	3	4	5	6	7	8	9	10
A	8	7	3	4	2	1	9	10	14	16

THE HEAPSORT PROCEDURE
ITERATION 5

- Iteration **5**, $i = A.length - \mathbf{4} = \mathbf{6}$

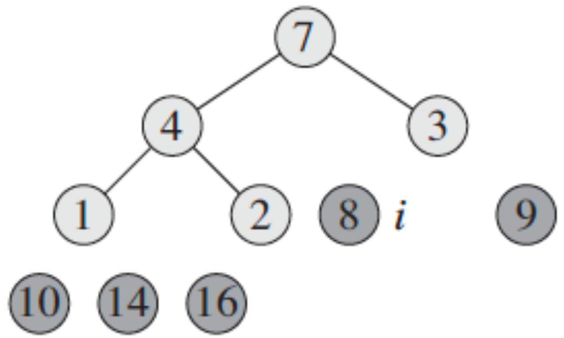
Index	1	2	3	4	5	6	7	8	9	10
A	8	7	3	4	2	1	9	10	14	16

- "Remove" $A[1]$ from array. $A.heap\text{-}size = A.heap\text{-}size - 1 = \mathbf{6} - 1 = \mathbf{5}$

Index	1	2	3	4	5	6	7	8	9	10
A	1	7	3	4	2	8	9	10	14	16

- MAX-HEAPIFY$(A, \mathbf{1})$

Index	1	2	3	4	5	6	7	8	9	10
A	7	4	3	1	2	8	9	10	14	16



(f)

THE HEAPSORT PROCEDURE
ITERATION 6

- Iteration **6**, $i = A.length - \mathbf{5} = \mathbf{5}$

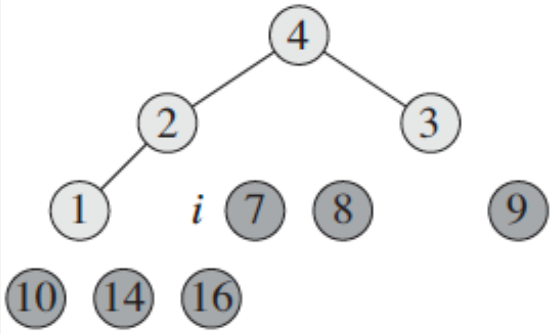
Index	1	2	3	4	5	6	7	8	9	10
A	7	4	3	1	2	8	9	10	14	16

- "Remove" $A[1]$ from array. $A.heap\text{-}size = A.heap\text{-}size - 1 = \mathbf{5} - 1 = \mathbf{4}$

Index	1	2	3	4	5	6	7	8	9	10
A	2	4	3	1	7	8	9	10	14	16

- MAX-HEAPIFY$(A, \mathbf{1})$

Index	1	2	3	4	5	6	7	8	9	10
A	4	2	3	1	7	8	9	10	14	16



(g)

THE HEAPSORT PROCEDURE
ITERATION 7

- Iteration $\mathbf{7}$, $i = A.length - \mathbf{6} = \mathbf{4}$

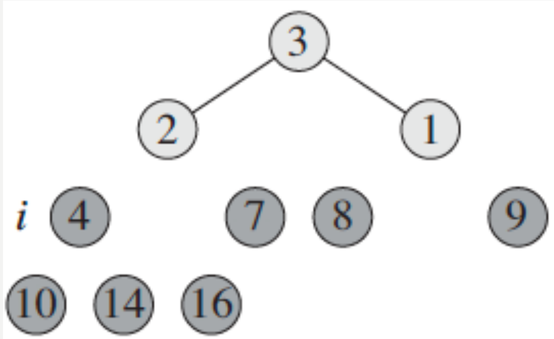
Index	1	2	3	4	5	6	7	8	9	10
A	4	2	3	1	7	8	9	10	14	16

- "Remove" $A[1]$ from array. $A.heap\text{-}size = A.heap\text{-}size - 1 = \mathbf{4}\ - 1 = \mathbf{3}$

Index	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16

- MAX-HEAPIFY$(A, 1)$

Index	1	2	3	4	5	6	7	8	9	10
A	3	2	1	4	7	8	9	10	14	16



(h)

THE HEAPSORT PROCEDURE
ITERATION 8

- Iteration **8**, $i = A.length - \mathbf{7} = \mathbf{3}$

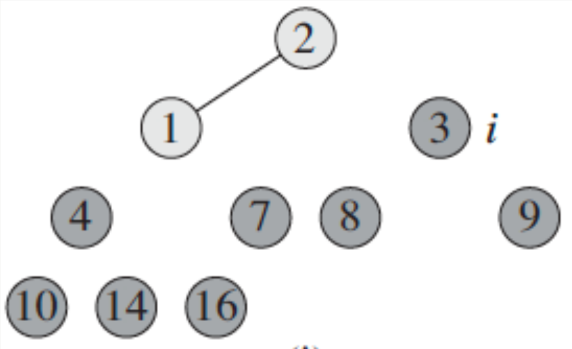
Index	1	2	3	4	5	6	7	8	9	10
A	3	2	1	4	7	8	9	10	14	16

- "Remove" $A[1]$ from array. $A.heap\text{-}size = A.heap\text{-}size - 1 = \mathbf{3} \; - 1 = \mathbf{2}$

Index	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16

- MAX-HEAPIFY$(A, 1)$

Index	1	2	3	4	5	6	7	8	9	10
A	2	1	3	4	7	8	9	10	14	16



(i)

THE HEAPSORT PROCEDURE
ITERATION 9

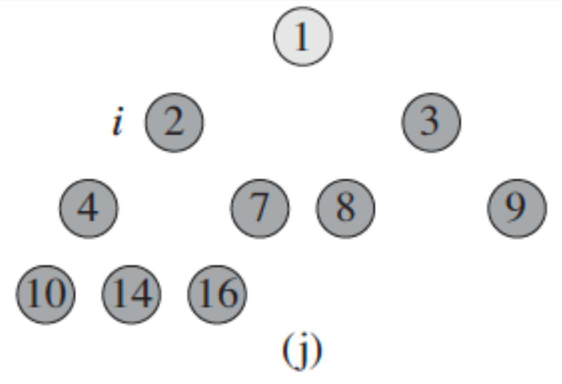
- Iteration **9**, $i = A.length - \mathbf{8} = \mathbf{2}$

Index	1	2	3	4	5	6	7	8	9	10
A	2	1	3	4	7	8	9	10	14	16

- "Remove" $A[1]$ from array. $A.heap\text{-}size = A.heap\text{-}size - 1 = \mathbf{2} - 1 = \mathbf{1}$

Index	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16

- STOP!



HEAPSORT ALGORITHM RUNNING TIME

- Input: array A
- Running time, where $n = A.length$
  $T(n) =$

HEAPSORT (A)	Cost	Time
BUILD-MAX-HEAP (A)	$O(n \lg n)$	1
for $i = A.length$ **downto** 2	$\Theta(1)$	n
exchange $A[1]$ with $A[i]$	$\Theta(1)$	$n-1$
$A.heap\text{-}size = A.heap\text{-}size - 1$	$\Theta(1)$	$n-1$
MAX-HEAPIFY $(A, 1)$	$O(\lg n)$	$n-1$

- Simplify the function $T(n) =$
- The asymptotic upperbound of $T(n)$ is ________.

SPACE COMPLEXITY

- A sorting algorithm that sorts the numbers ***in place*** means that the algorithm ***rearranges*** the numbers ***within*** the array A, with at most a constant number of them stored outside the array any time.
- Which of the following algorithms is/are ***in place*** sorting algorithm?
  - Bubblesort ✓
  - Insertion sort ✓
  - Mergesort
  - Quicksort ✓
  - Heapsort ✓

SPACE COMPLEXITY ANALYSIS OF MERGESORT

- Extra space used by MERGE-SORT algorithm to sort $A[1..n]$
  - The recursive running time $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

Level	Tree	Node	Per-level Cost	Per-level Space Cost
0	cn	1	$1 \times cn$	$2^0 \times (n+2)$
1	cn/2 cn/2	2	$2 \times cn/2$	$2^1 \times (n/2^1+2)$
2	cn/4 cn/4 cn/4 cn/4	4	$2^2 \times cn/2^2$	$2^2 \times (n/2^2+2)$
⋮	⋮ ⋮ ⋮ ⋮ ⋮		⋮	
lg n	T(1) ⋮ … T(1)	n	$\Theta(n)$	

  - Level k of the recursion tree requires ___________ extra space.

$MERGE\text{-}SORT(A, p, r)$
if $p < r$
$\quad q = \lfloor (p+r)/2 \rfloor$
$\quad MERGE\text{-}SORT(A, p, q)$
$\quad MERGE\text{-}SORT(A, q+1, r)$
$\quad MERGE(A, p, q, r)$

$MERGE(A, p, q, r)$
$n_1 = q - p + 1$
$n_2 = r - q$
Let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ **to** n_1
$\quad L[i] = A[p + i - 1]$
for $j = 1$ **to** n_2
$\quad R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ **to** r
$\quad$ **if** $L[i] \le R[j]$
$\quad\quad A[k] = L[i]$
$\quad\quad i = i + 1$
$\quad$ **else** $A[k] = R[j]$
$\quad\quad j = j + 1$

SPACE COMPLEXITY
MERGESORT

- Extra space used by MERGE-SORT algorithm to sort $A[1..n]$
  - The total space cost of the algorithm is

$$S(n) = \sum_{k=0}^{\lg n} 2^k \left(\frac{n}{2^k} + 2\right)$$

$$= \sum_{k=0}^{\lg n} (n + 2^{k+1})$$

$$= \sum_{k=0}^{\lg n} n + \sum_{k=0}^{\lg n} 2^{k+1} \;\; = n(\lg n + 1) + \left(2 \cdot \frac{2^{\lg n + 1} - 1}{2 - 1}\right)$$

$$= n \lg n + 5n - 2 = O(n \lg n)$$

Level	Per-level Space Cost
0	$2^0 \times (n+2)$
1	$2^1 \times (n/2^1+2)$
2	$2^2 \times (n/2^2+2)$
$\vdots$	
lg n	

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
Let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

SORTING ALGORITHMS

- Differences of sorting algorithms that sort any given array $A[1..n]$.

Technique	**Algorithm**	**Time Complexity Bound ($T(n)$)**	**Space Complexity Bound ($S(n)$)**
Naïve approach	Bubblesort		
	Insertion sort	Best-case: Worst-case:	
Divide-and-conquer	Mergesort		
	Quicksort	Best-case: Worst-case:	
Building a data structure	Heapsort		

NEXT UP
DYNAMIC PROGRAMMING

REFERENCE