ALGORITHMS AND DATA STRUCTURES II

Lecture 2

Heaps,

Heapsort Algorithm,

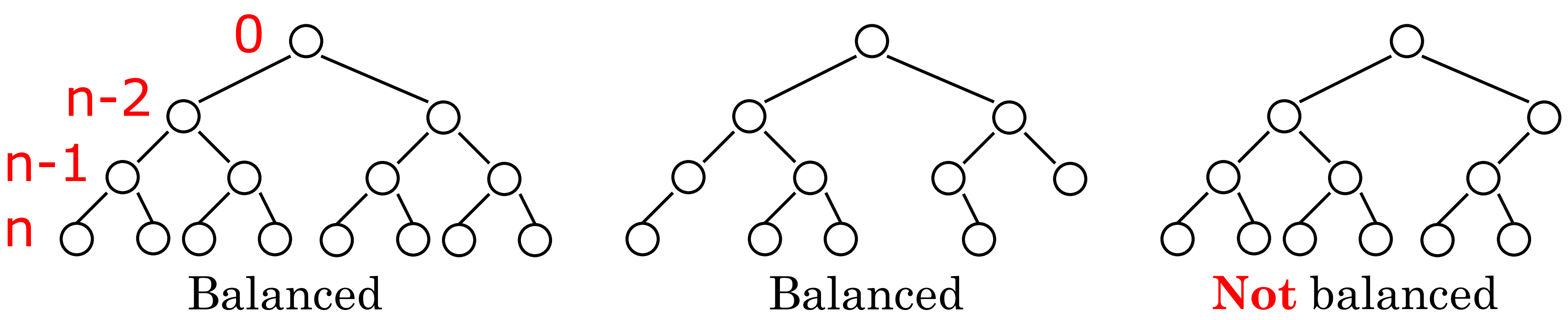
Priority Queues

Lecturer: K. Markov

markov@u-aizu.ac.jp

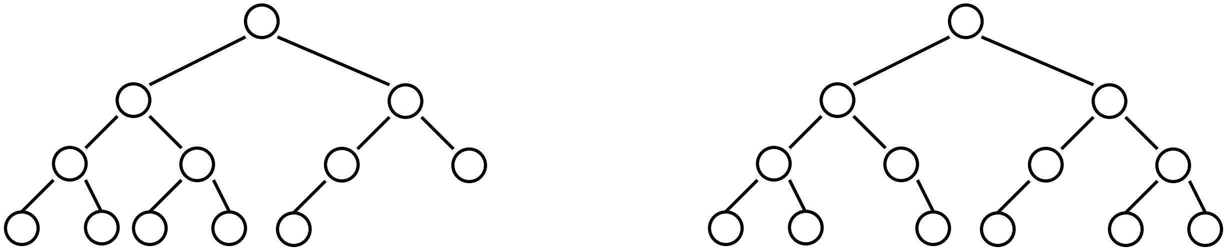
HEAPS

- Binary trees
  - The depth of a node is its distance from the root.
  - The depth of a tree is the depth of the deepest node.
- A binary tree of depth n is balanced if all the nodes at depths 0 through n-2 have two children



HEAPS

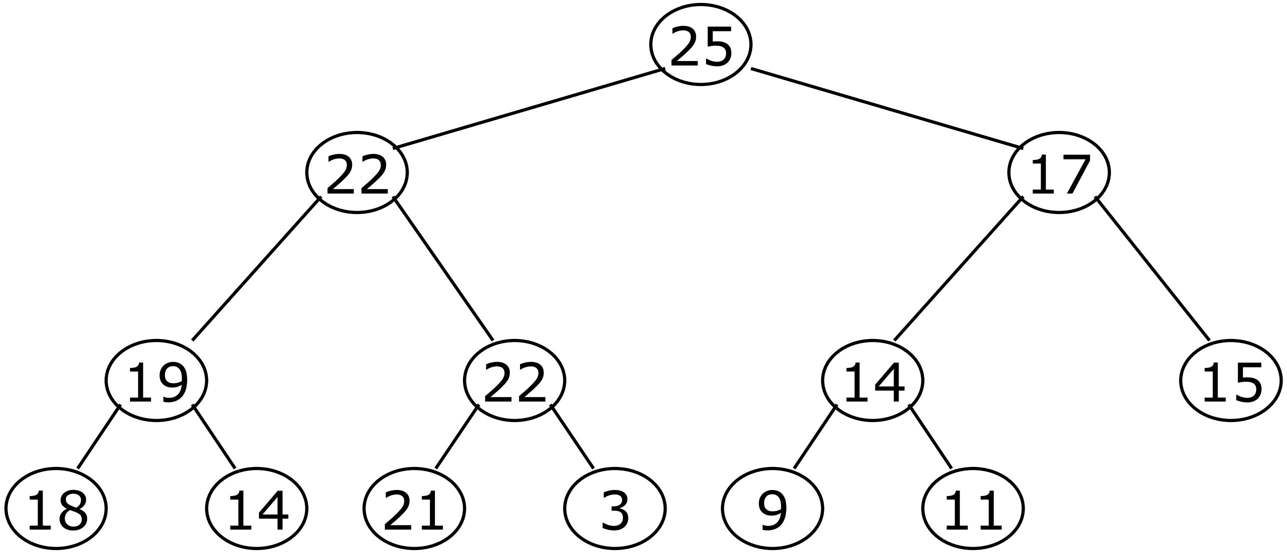
- A balanced binary tree is left-justified if:
  - all the leaves are at the same depth, or
  - all the leaves at depth `n+1` are to the left of all the nodes at depth `n`.



Left-justified | **Not** left-justified

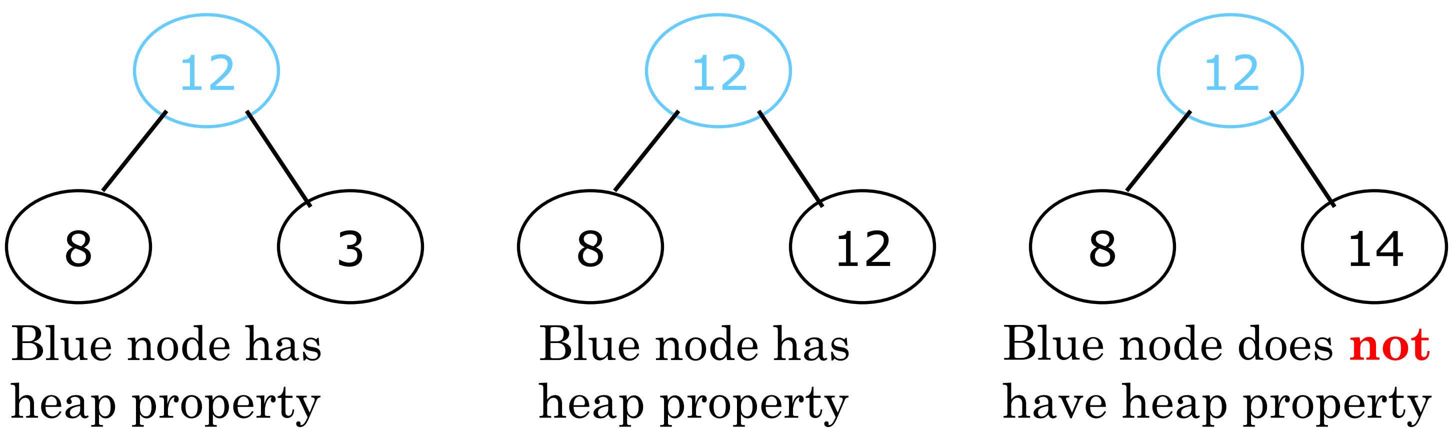
HEAPS

- What is a heap?
  - *A balanced, left-justified binary tree in which no node has a value greater than the value in its parent.*



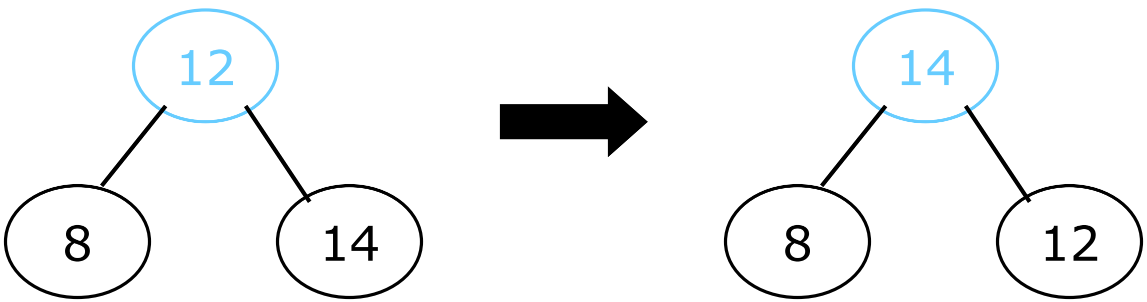
HEAPS

- Heap property
  - A node has the heap property if the value in the node is as large as or larger than the values in its children.



HEAPS

- “Heapify”
  - Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child

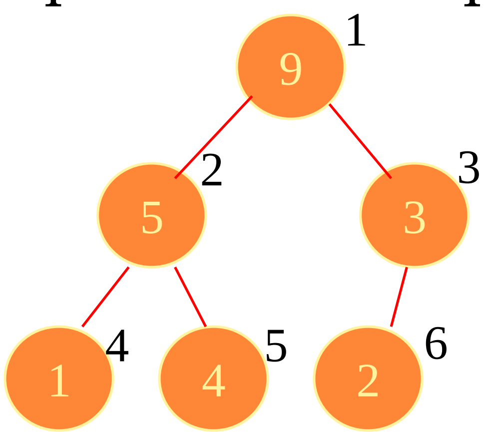


Blue node does **not** have heap property

Blue node has heap property

HEAPS

- Array representation
  - In practice, it is easier and more efficient to implement a heap using an array.



1	2	3	4	5	6
9	5	3	1	4	2

  - Relationships between indexes of parents and children.

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

LEFT(i) {return 2i}

RIGHT(i) {return 2i+1}

HEAPS

- Maintaining heap property
  - **Input**: array ***A*** and index ***i*.**
  - **Output**: sub-tree rooted at ***i*** with heap property.

```
def MaxHeapify (A, i)
    l = LEFT (i)
    r = RIGHT (i)
    if l <= A.heap_size and A[l] > A[i]:   // if L child exists and is > A[i]
        largest = l
    else:
        largest = i
    if r <= A.heap_size and A[r] > A[largest]:
        largest = r
    if largest ≠ i:
        swap (A[i], A[largest])
        MaxHeapify (A, largest)           // heapify the subtree
```

HEAPS

- Constructing a heap
  - <u>Bottom-up:</u> Put everything in an array and then heapify the trees in a bottom-up way.
  - Given a heap of n nodes, what's the index of the last parent? ($\lfloor n/2 \rfloor$)

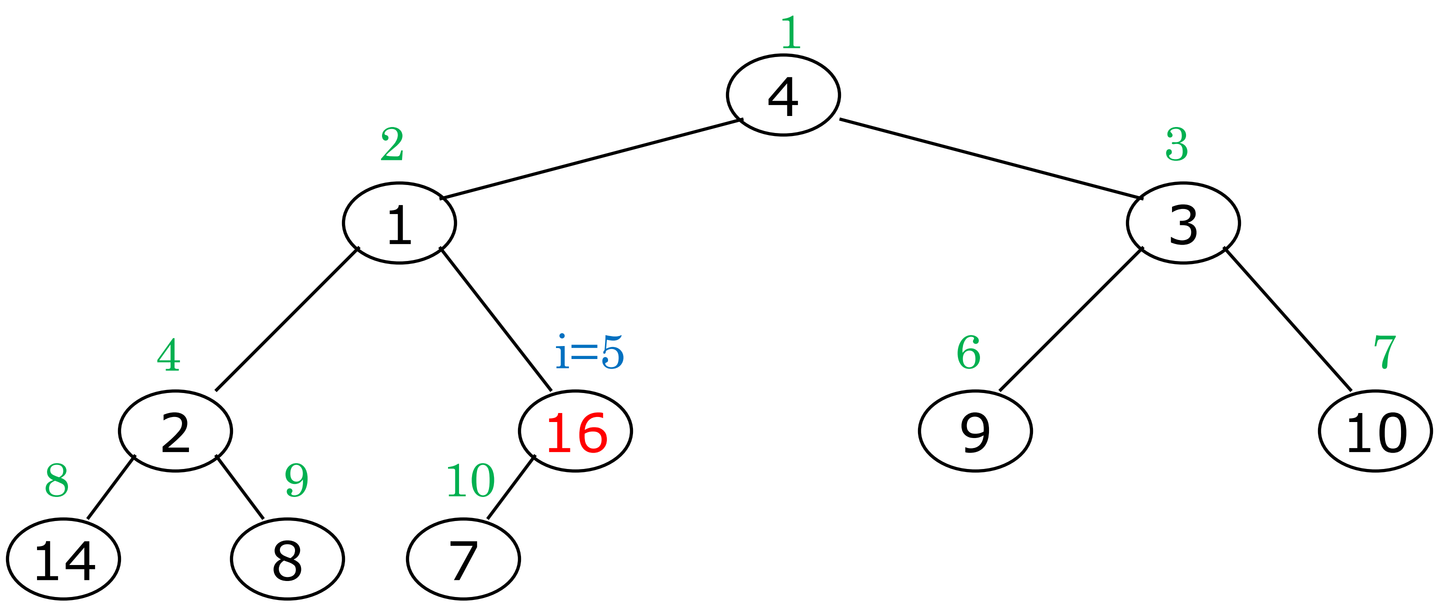
```
def HeapBottomUp (A)
    //Constructs a heap from the elements
    //of a given array by the bottom-up algorithm
    //Input: An array A[1..n] of orderable items
    //Output: A heap A[1..n]
    A.heap_size = A.length
    for i = ⌊A.length / 2⌋ to 1
        MaxHeapify (A, i)
```

HEAPS

- Constructing a heap

A

4	1	3	2	16	9	10	14	8	7

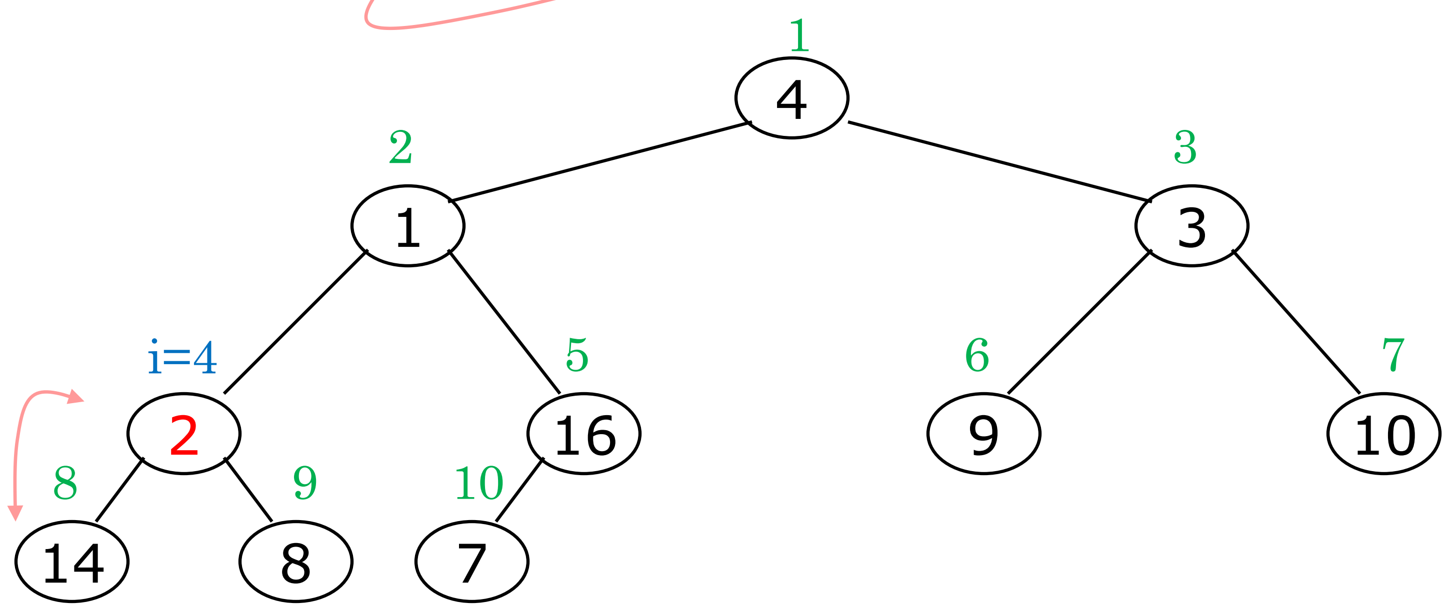


HEAPS

- Constructing a heap

A

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

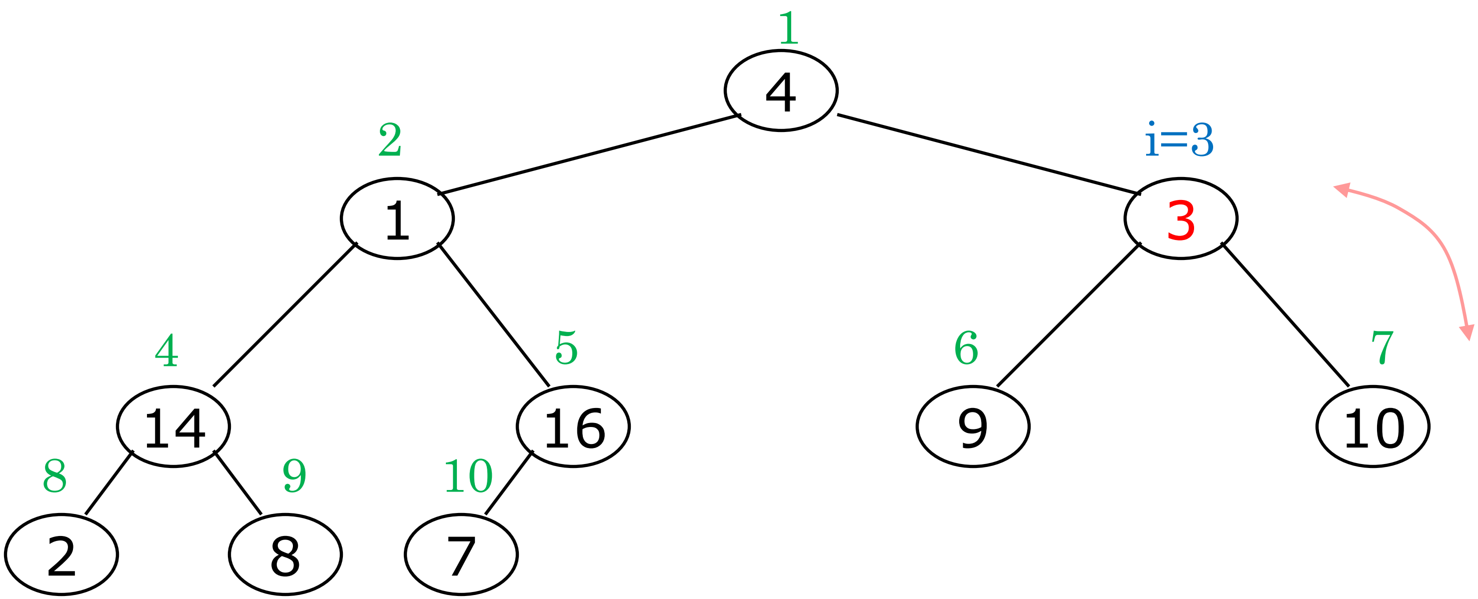


Heaps

- Constructing a heap

A

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

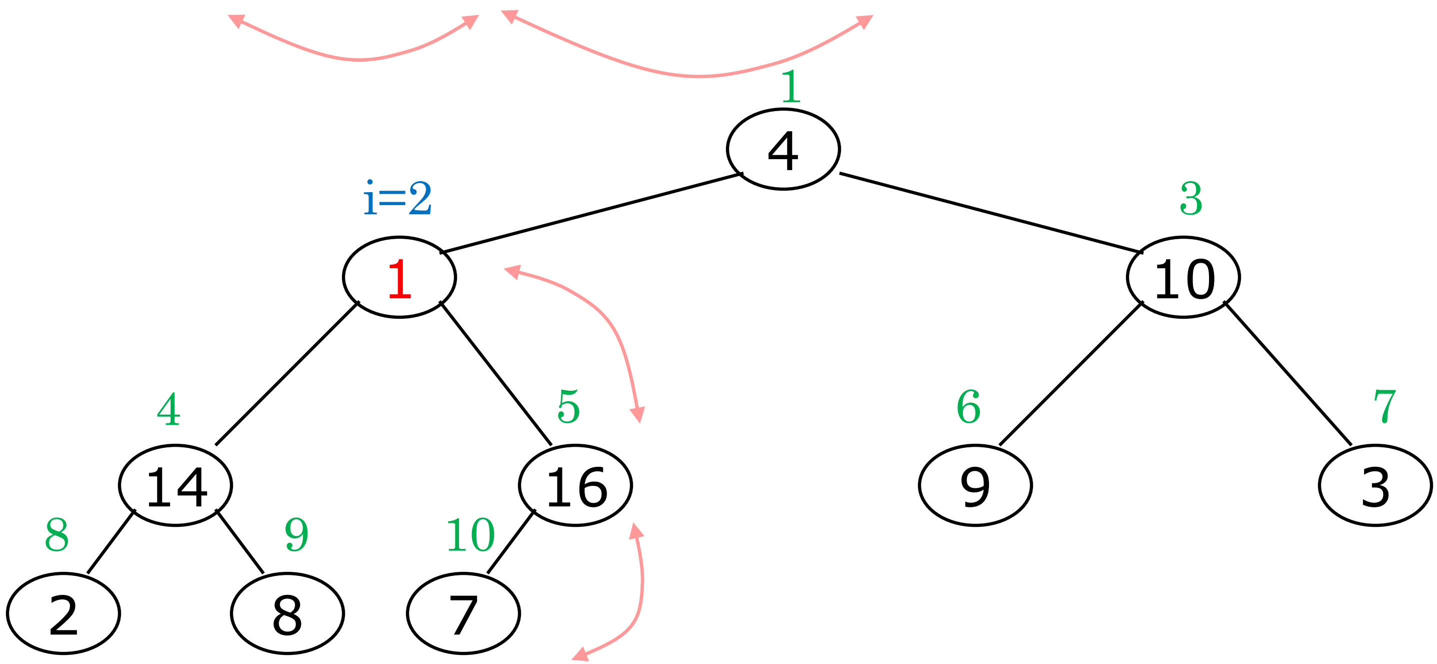


Heaps

- Constructing a heap

A

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

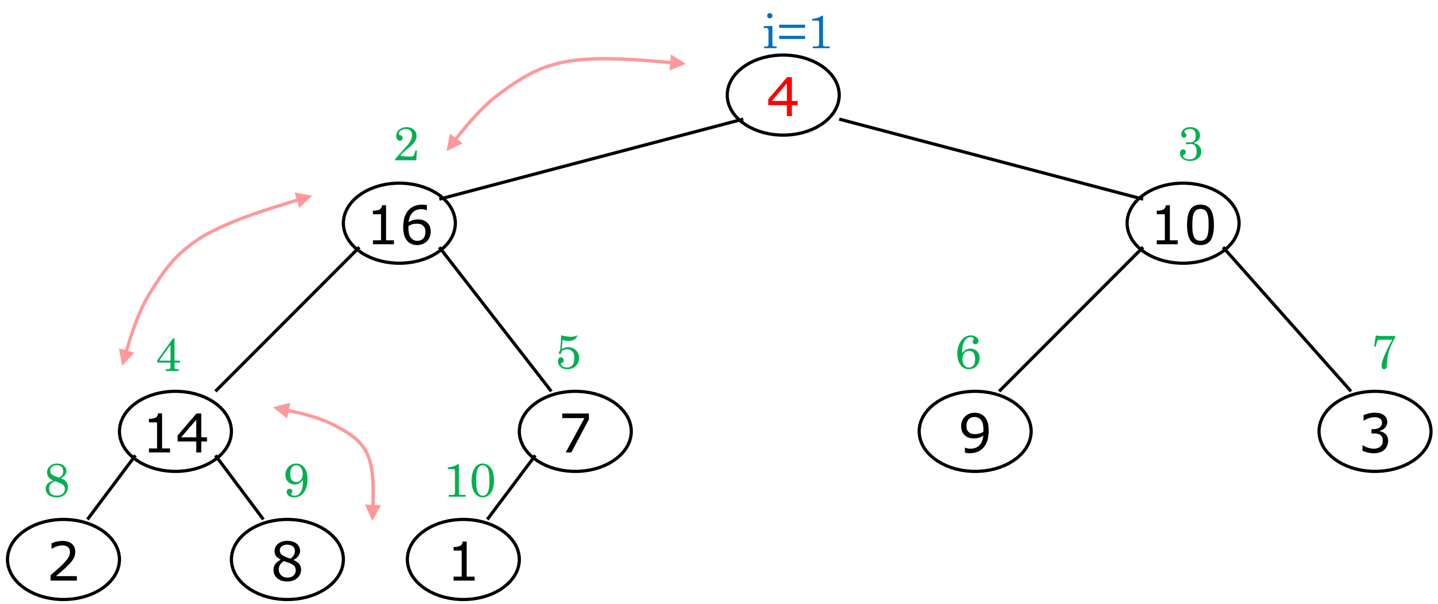


HEAPS

- Constructing a heap

A

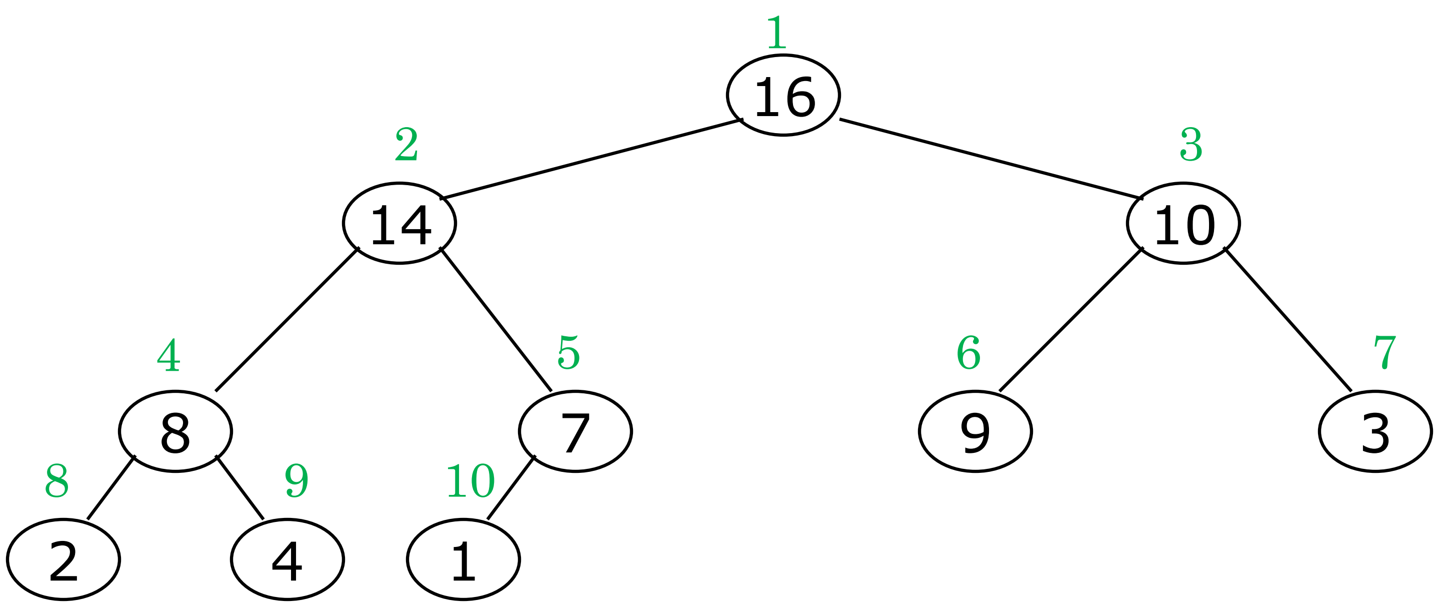
4	**16**	10	**14**	7	9	3	2	**8**	1



Heaps

- Constructing a heap

A									
16	14	10	8	7	9	3	2	4	1



HEAPS

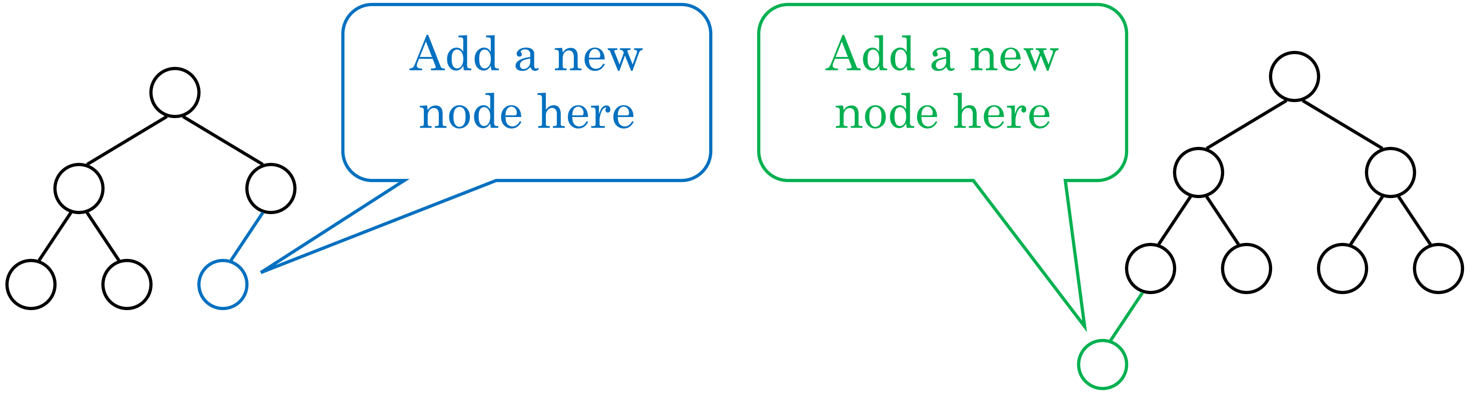
- Time complexity of heap construction
  - An $\boldsymbol{n}$ node heap has height $\boldsymbol{h} = \lfloor \log \boldsymbol{n} \rfloor$.
  - There are at most $\lceil \boldsymbol{n}/2^{\boldsymbol{h}+1} \rceil$ nodes at any height $\boldsymbol{h}$.
  - **MaxHeapify** complexity is $\boldsymbol{O}(\log \boldsymbol{n}) = \boldsymbol{O}(\boldsymbol{h})$
  - The time complexity of **HeapBottomUp** is then bounded from above by

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

since $\displaystyle\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$

HEAPS

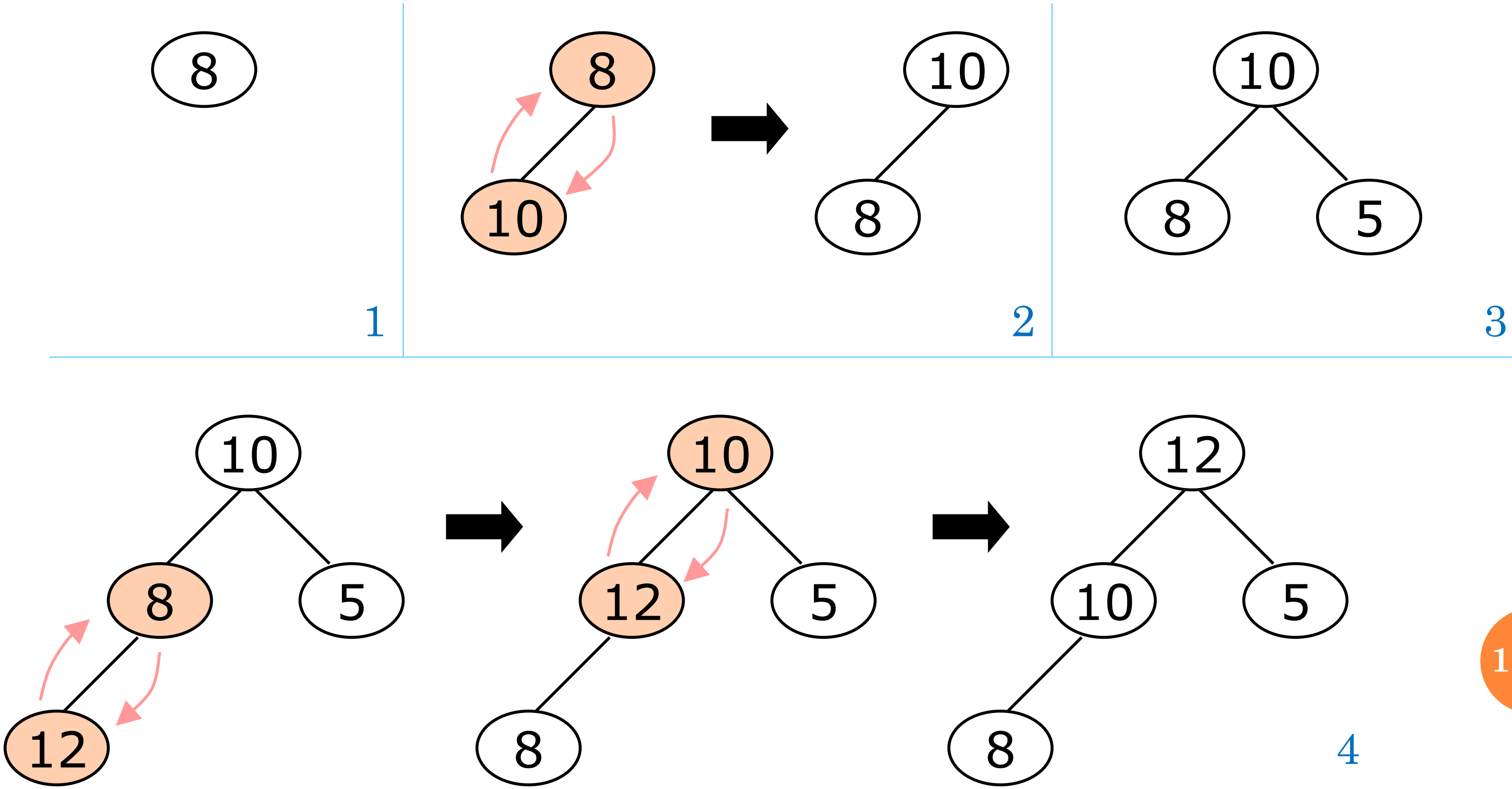
- New element insertion.
  - Insert element at the last position in heap.
    - Add the node just to the right of the rightmost node in the deepest level
    - If the deepest level is full, start a new level.



  - Compare with its parent, and exchange them if it violates the heap property.
  - Continue comparing the new element with nodes up the tree until the heap property condition is satisfied.

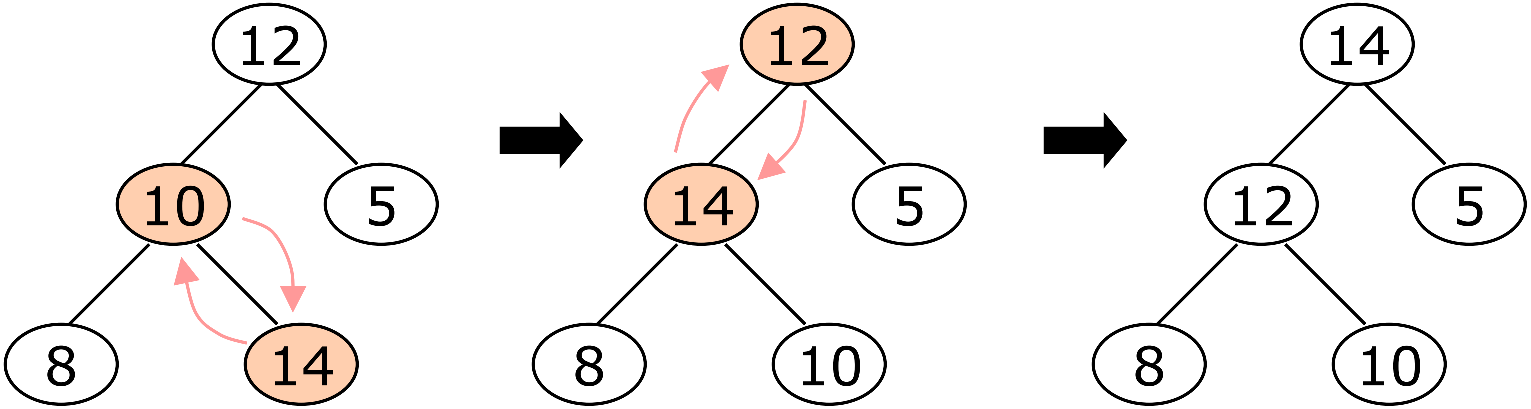
HEAPS

- Top down heap construction.
  - Start with empty heap.
  - Insert all nodes one at a time.



HEAPS

- Top down heap construction.
  - Start with empty heap.
  - Insert all nodes one at a time.



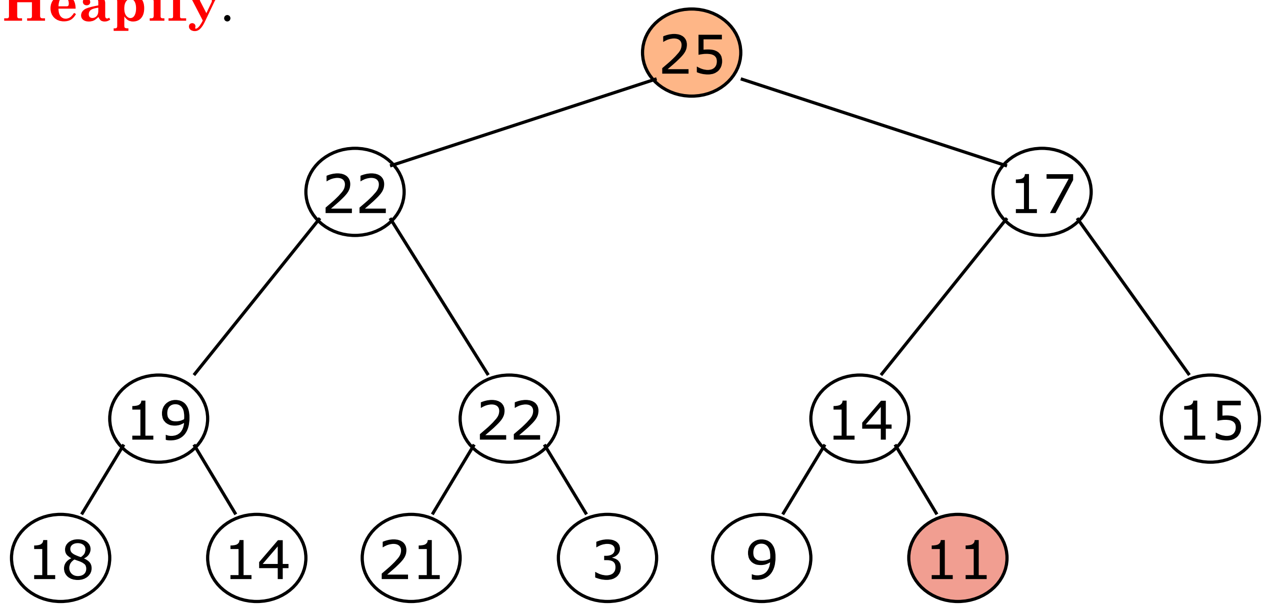
HEAPS

- Root deletion
  - The root of a heap can be deleted and the heap fixed up as follows:

    - Exchange the root with the last leaf .

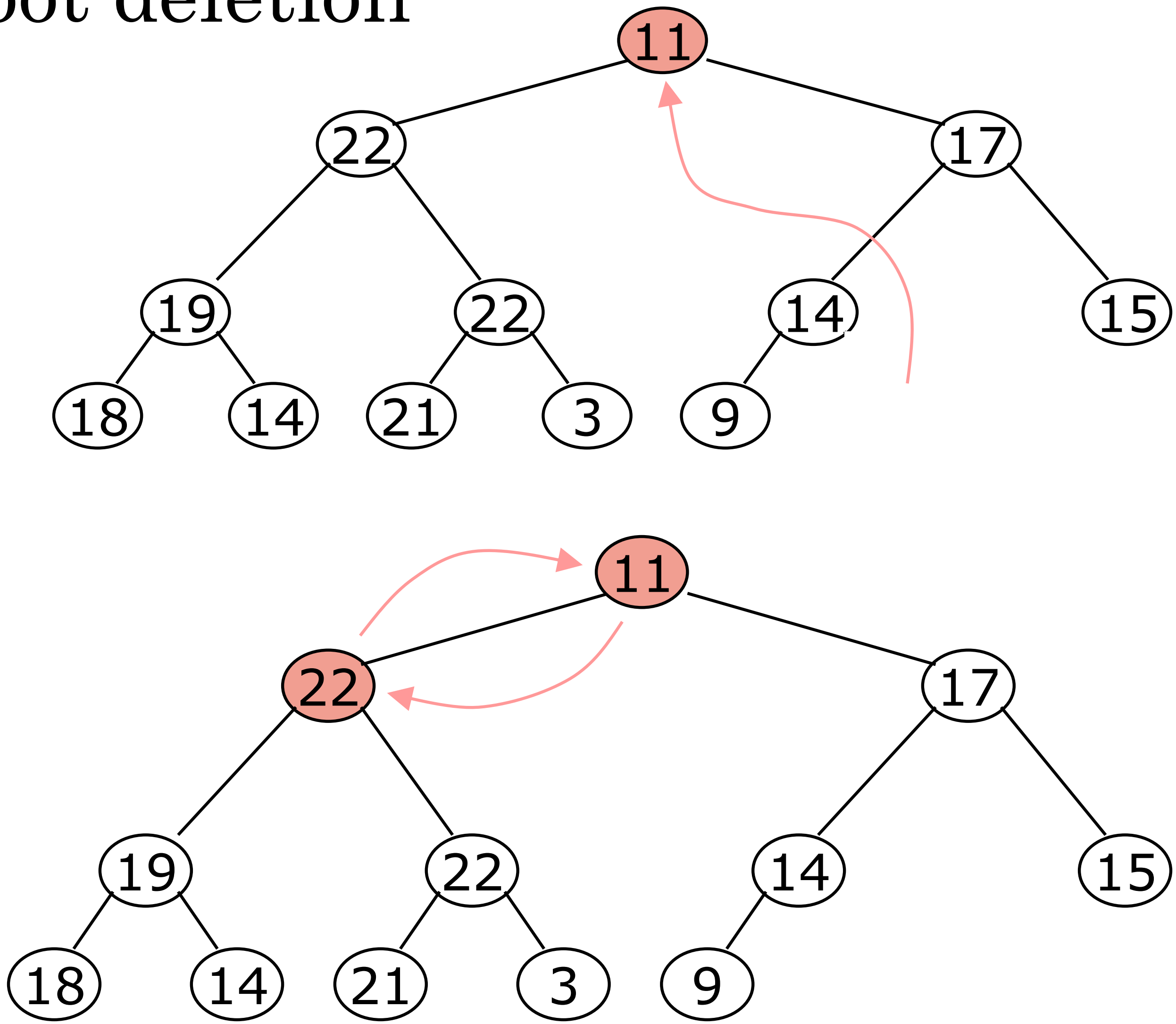
    - Decrease the heap's size by 1.

    - Heapify the smaller tree in exactly the same way we did it in **MaxHeapify**.



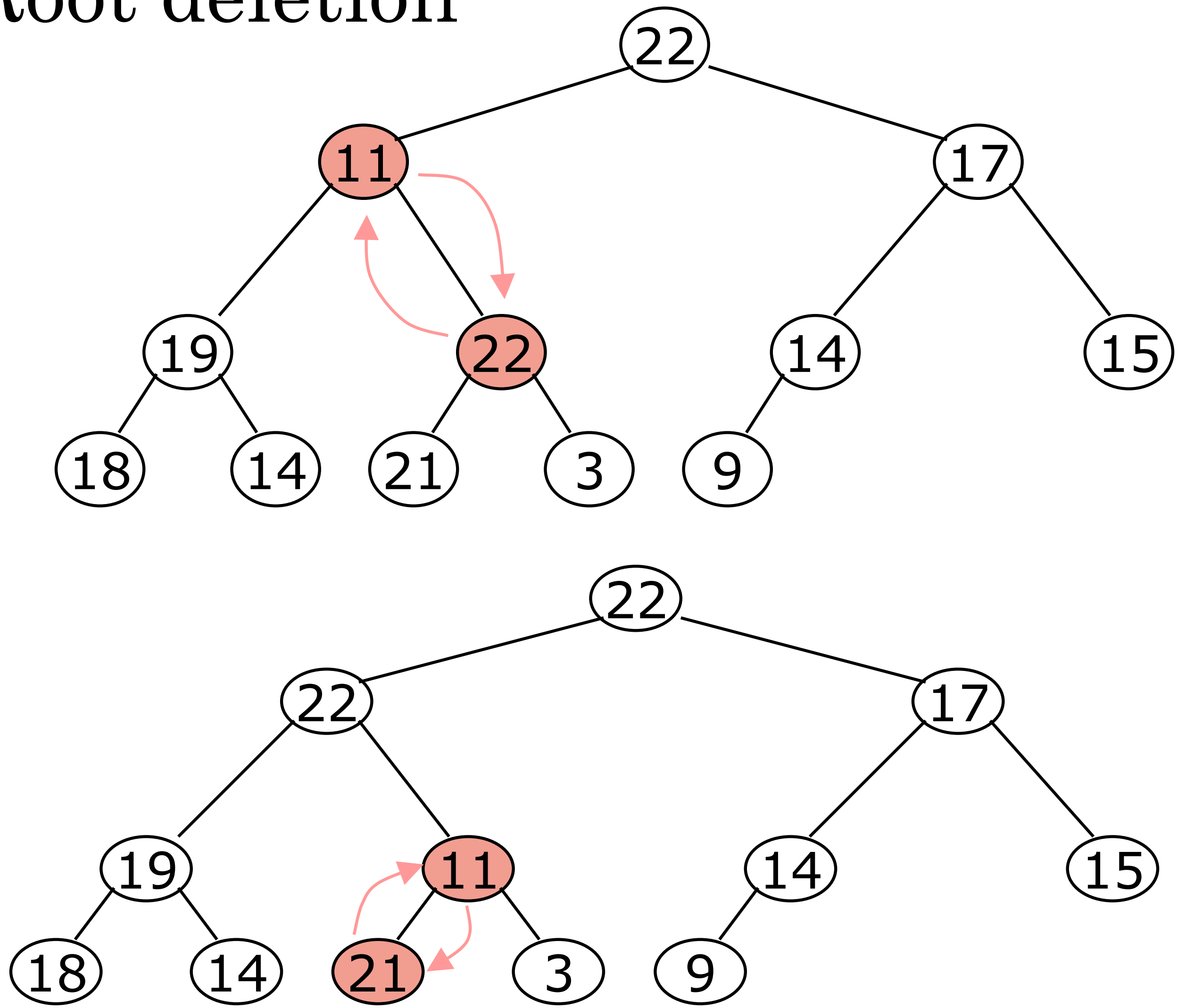
HEAPS

- Root deletion



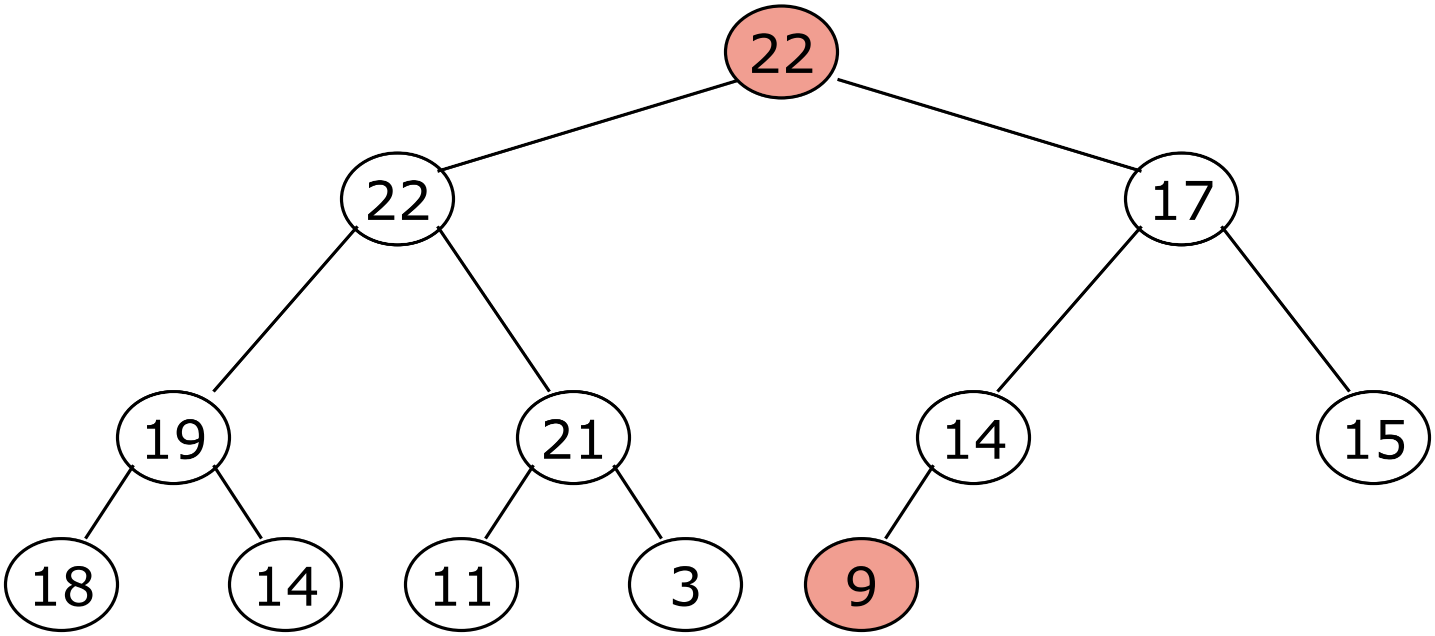
Heaps

- Root deletion



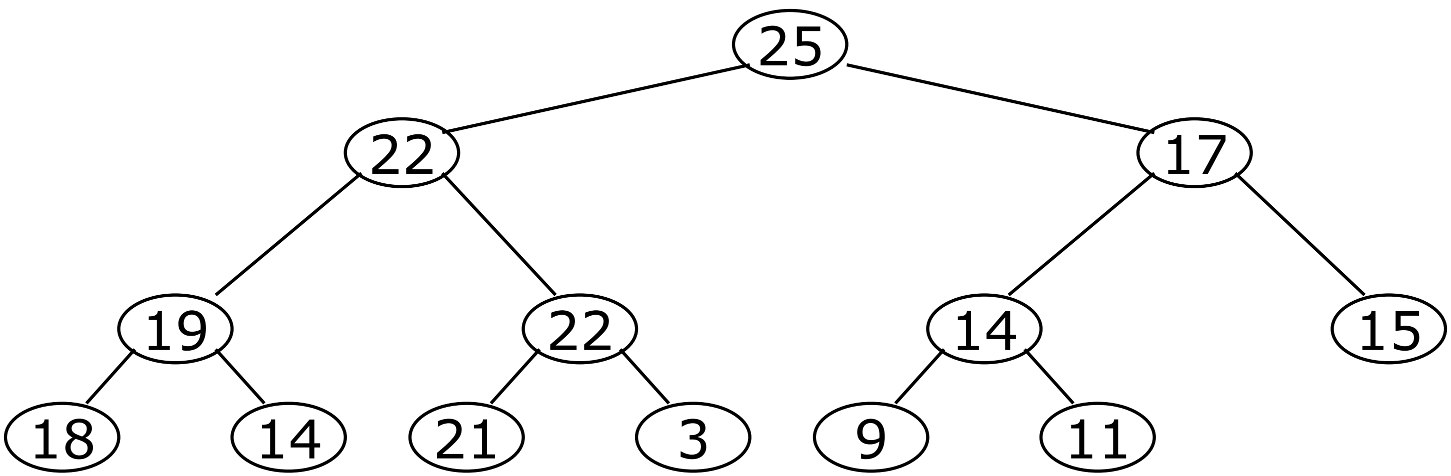
HEAPS

- Root deletion
  - Removing next root



HEAPSORT

- Heapsort algorithm:
  - (Heap construction) Build heap for a given array (either bottom-up or top-down)
  - (Maximum deletion) Apply the root-deletion operation n-1 times to the remaining heap until heap contains just one node.



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

Heapsort

- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...

0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	25

- ...And pretend that the last element in the array no longer exists—that is, the “last index” is 11 (9)

HEAPSORT

- Reheap the root node (index 0, containing 11)

0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	25

0	1	2	3	4	5	6	7	8	9	10	11	12
22	22	17	19	21	14	15	18	14	11	3	9	25

0	1	2	3	4	5	6	7	8	9	10	11	12
9	22	17	19	22	14	15	18	14	21	3	22	25

- …And again, remove and replace the root node
- Repeat until the last becomes first, and the array is sorted!

HEAPSORT

- Animated example

6 5 3 1 8 7 2 4

HEAPSORT

- Pseudo-code of the Heapsort algorithm.

```
def Heapsort (A)
    //Constructs a heap from the elements
    //Sorts elements of the array.
    //Input: An array A[1..n] of orderable items
    //Output: A sorted A[1..n]
    HeapBottomUp (A)
    for i = A.length to 2:   // n-1 times
      swap (A[1], A[i])
      A.heap_size = A.heap_size - 1
      MaxHeapify (A, 1)
```

HEAPSORT

- Analysis of the Heapsort
- Heapsort consists of two parts:
  - Heap construction which has $\boldsymbol{O(n)}$ time complexity.
  - For loop (repeated $\boldsymbol{n-1}$ times) which has time complexity $\boldsymbol{O(\log n)}$.

Total:

$$O(n) + (n-1)O(\log n) = O(n \log n)$$

PRIORITY QUEUE

- Priority queue is an abstract data type which is like a regular queue or stack data structure, but additionally, each element is associated with a "priority".
  - **stack** — elements are pulled in last-in first-out-order (e.g. a stack of papers)
  - **queue** — elements are pulled in first-in first-out-order (e.g. a line in a cafeteria)
  - **priority queue** — elements are pulled highest-priority-first (e.g. cutting in line, or VIP service).

PRIORITY QUEUE

- Operations on priority queues
  - ***Insert (S, x)*** – insert element ***x*** into queue ***S***.
  - ***Maximum (S)*** – return 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)*** – increases the value of ***x***'s key to the new value ***k*** which should be at least as large as ***x***'s current key value.

PRIORITY QUEUE

- Priority queue is implemented as **heap**.
- Operations on priority queues:
  - ***Extract-Max (S)*** operation

```
def Extract-Max (A)
    // Input: heap A[1..n]
    // Removes and returns the root element
    max = A[1]
    A[1] = A[A.heap_size]
    A.heap_size = A.heap_size - 1
    MaxHeapify (A, 1)
    return max
```


  - Time complexity: $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n})$

Priority Queue

- Operations on priority queues
  - ***Increase-Key (S)*** operation

```
def Heap-Increase-Key (A, i, k)
    // Input: heap A[1..n], element index i, and its new key k.
    // Output: heap A[1..n] conforming to heap property.
    A[i] = k
    while i > 1 and A[Parent(i)] < A[i]:
        swap (A[Parent(i)], A[i])
        i = Parent (i)
```


  - Time complexity: $\boldsymbol{O}(\log \boldsymbol{n})$

Priority Queue

- Operations on priority queues
  - ***Insert (S, x)*** operation

```
def Max-Heap-Insert (A, k)
    // Input: heap A[1..n] and new key k.
    // Output: heap A[1..n+1].
    A.heap_size = A.heap_size + 1
    A[A.heap_size] = MAX_NEGATIVE
    Heap-Increase-Key (A, A.heap_size, k)
```

  - Time complexity: $\boldsymbol{O}(\log n)$

That's All for Today!