[CSED233-01] Data Structure

Priority Queue and Heap

Jaesik Park

POSTECH

Announcement

- Programming assignment #2
  - Announced: March 24
  - Due date: April 7 midnight
- Office Hour
  - We have two sessions
    - At 1PM~2PM
    - Every Tuesday: Professor
    - Every Thursday: Teaching Assistants

Unique Binary Tree (by Two Traversals)

- We can identify the binary tree uniquely by two traversal sequences like:
  - (*postorder* & *inorder*), (*preorder* & *inorder*), (*level-order* & *inorder*)
    - *inorder*: to find Left & Right child/subtrees
    - *postorder*: to find the Root (the last in *postorder*)
    - *preorder*: to find the Root (the first/ in *preorder*)
    - *level-order*: to find the Root

- However, the other combinations leaves some ambiguity in the tree structure

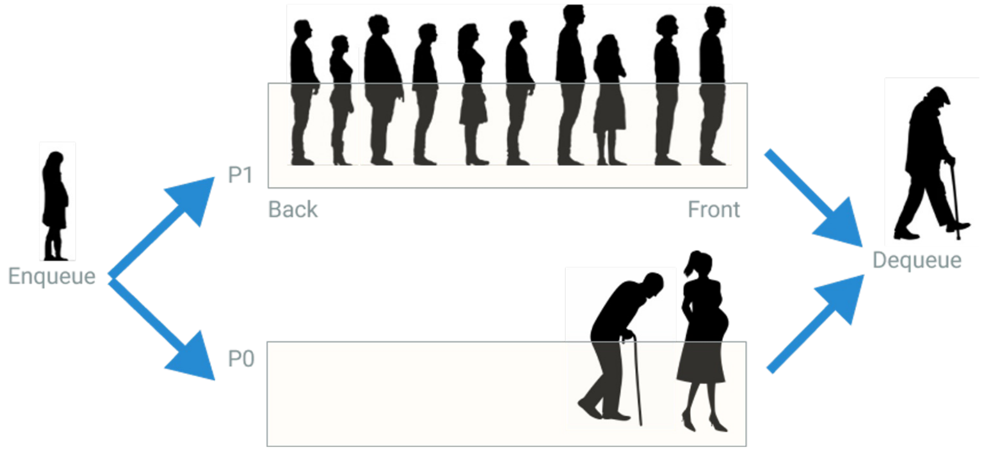
- What about preorder and postorder?

Check Why ☺

- Given a preorder traversal and a postorder traversal,
  - can we reconstruct a general tree? No!
  - can we reconstruct a binary tree? No!
  - can we reconstruct a complete binary tree? Yes!

Priority Queue

- We learned queue
  - FIFO (First-In First-Out) list
  - Similar to top in the stack, here we have front and rear
  - $Q = <a_1, a_2, \ldots, a_n>$
  - Enqueue, Dequeue, …
- How to incorporate priority for queue?
- Priority queue consists of a set of elements (organized by *priority*)
  - Each element x has a *priority* $p(x)$ (also called *importance* or *key*)
    - Not necessarily unique
  - Supports the following operations:
    - *Insert(x, H)* – arbitrary element insertion
    - *DeleteMin(H)* = *Min(H)* + *Delete*
      - Delete elements in the order of priority



Priority Queue's Implementation

- Obvious ways to implement

	Insert	*DeleteMin*
Normal queue	O(1)	O(*n*)
Unsorted linked list	O(1)	O(*n*)
Sorted linked list	O(*n*)	O(1)

- O(n) seems too much… Can't we implement this better?

- Heap!

Heap

- Tree-based data structure that satisfies the *heap property*
  - *if B is a child node of A, then $p(A) \leq p(B)$*
  - Implies that an element with the lowest priority is always in the root node (*min-heap*) ↔ *max-heap*
- To efficiently implement a priority queue
  - *Insert* & *DeleteMin*: O(log *n*)
- There are different types of heaps
  - Binary heap
  - Binominal heap
    - Supports quickly merging two heaps
  - Fibonacci heap, 2-3 heap, etc.
- We will learn binary heap as an example ☺

Binary Heap

- Satisfying two properties:

(1) Complete binary tree (Structural property)
- Can be implemented in an array

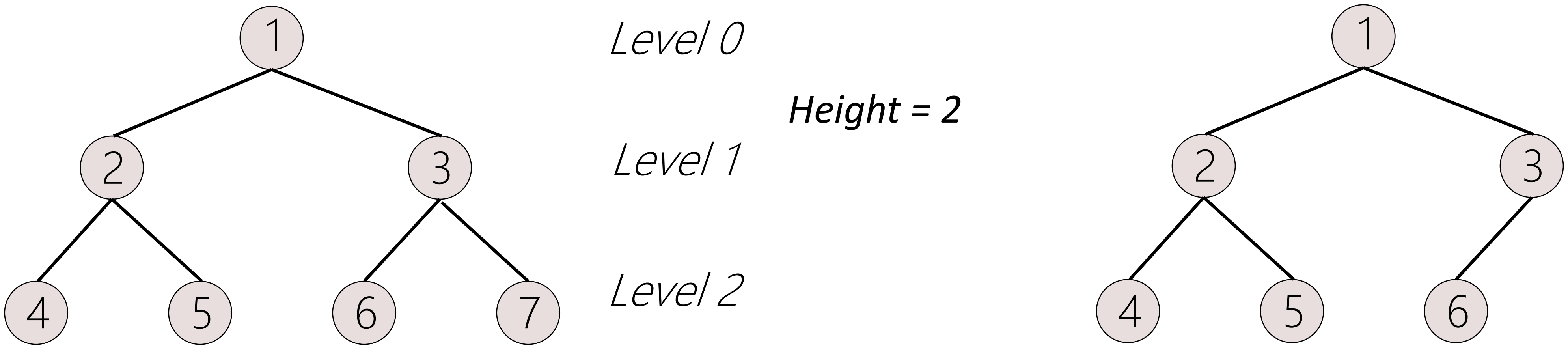
(2) Min tree (Heap order property)
- $p(node) \leq p(children)$

Min-Heap

(2') $p(node) \geq p(children)$ ⇔ Max-heap

Recap: Complete Binary Tree

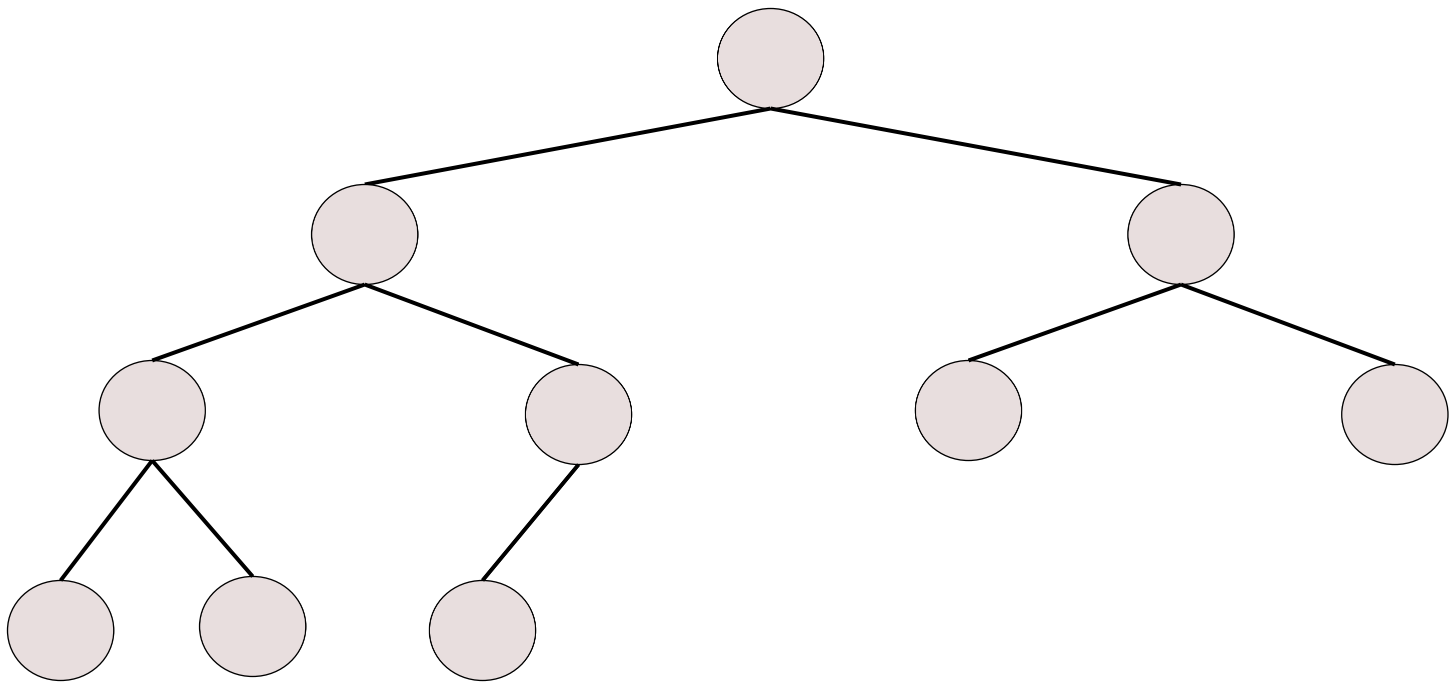
- Relaxed definition of a full binary tree
- A binary tree of height *h* is complete, if
  - All levels (possibly except *h*) are completely full
  - Level *h* (leaf level) is filled from left to right



full binary tree with 7 nodes

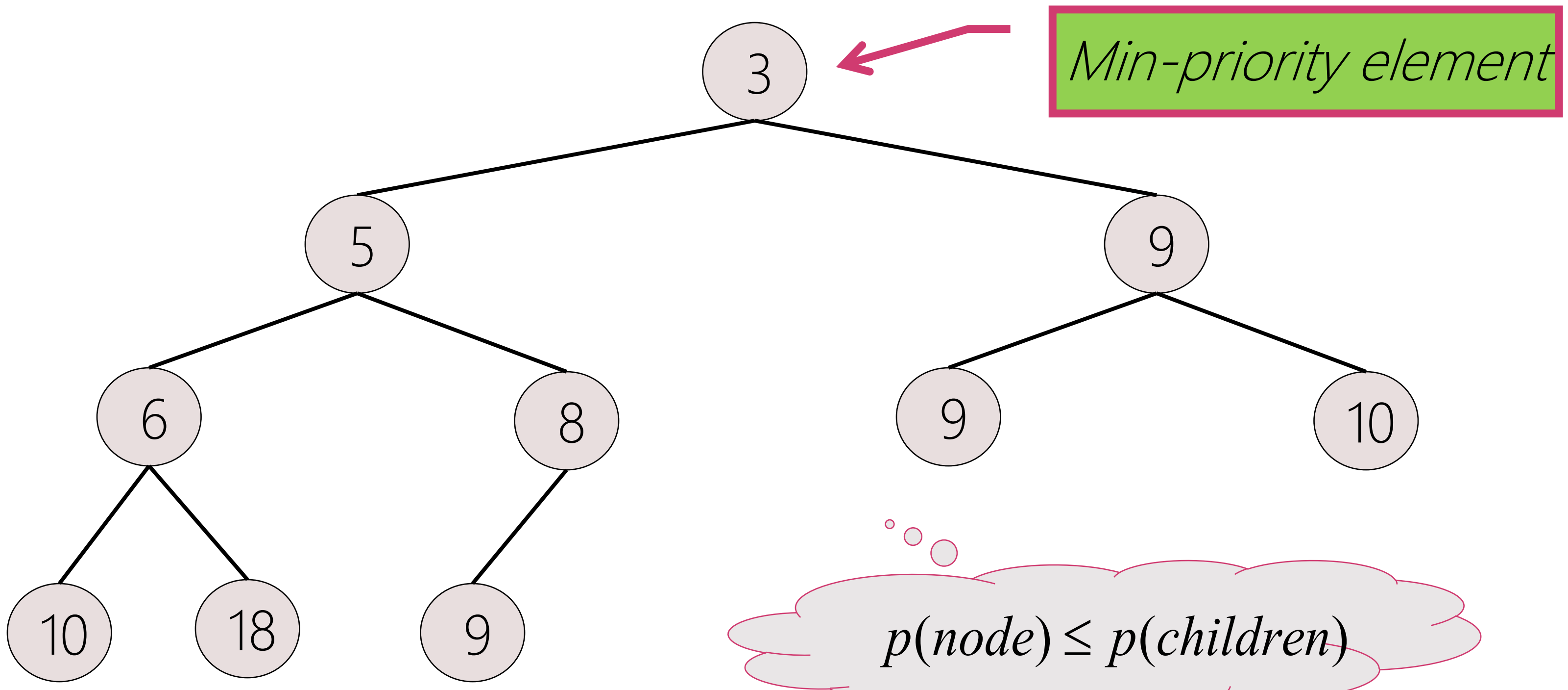
Complete binary tree with 6 nodes

Min-Heap: Structural Property



- Complete binary tree (CBT) with 10 nodes

Min-Heap: Heap Order Property

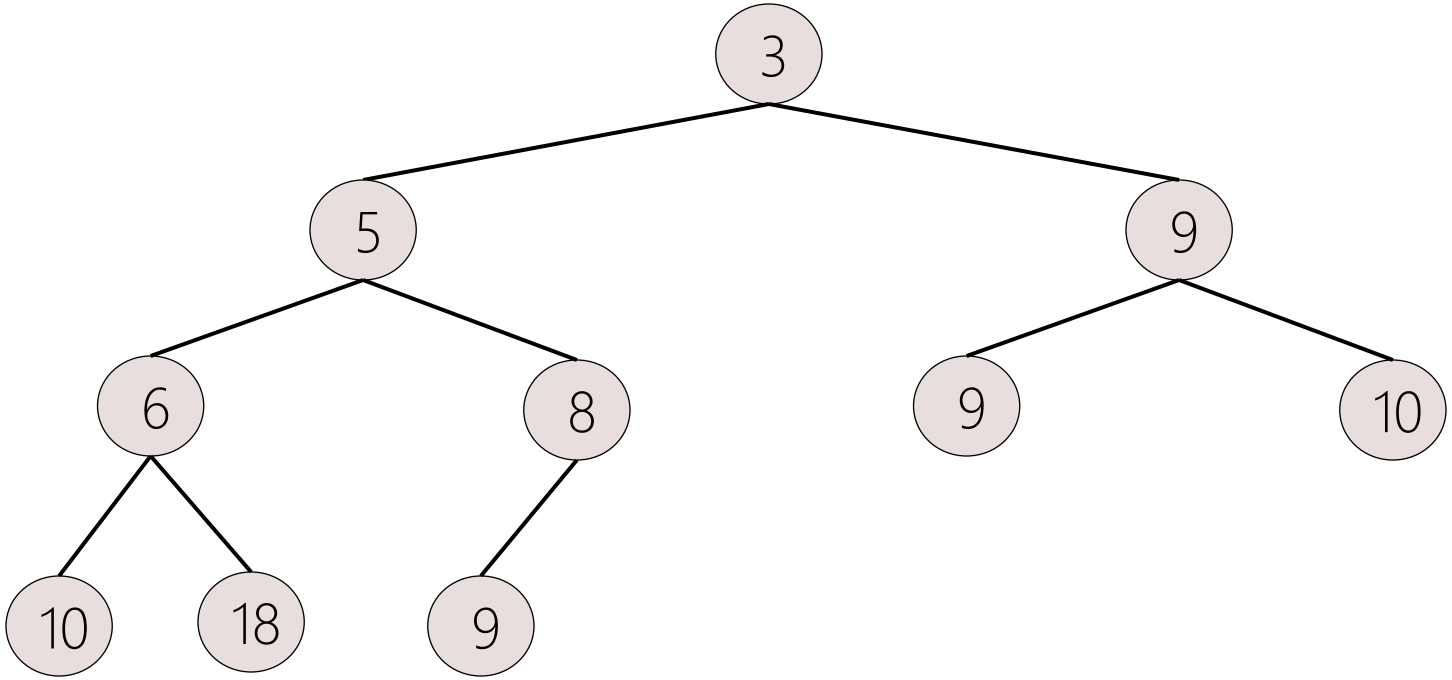


$$p(node) \leq p(children)$$

- Complete binary tree (CBT), satisfying *min-heap order property*

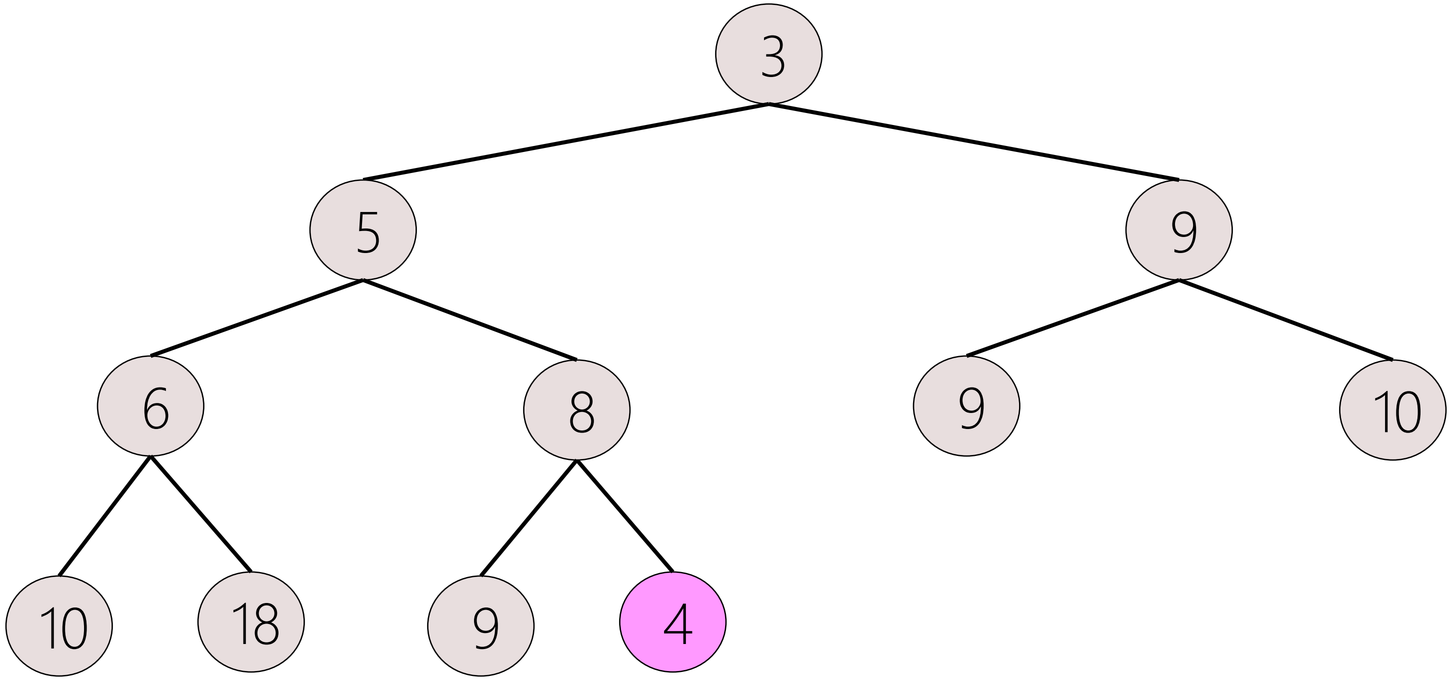
➔ Min-Heap

Insert into Min-Heap



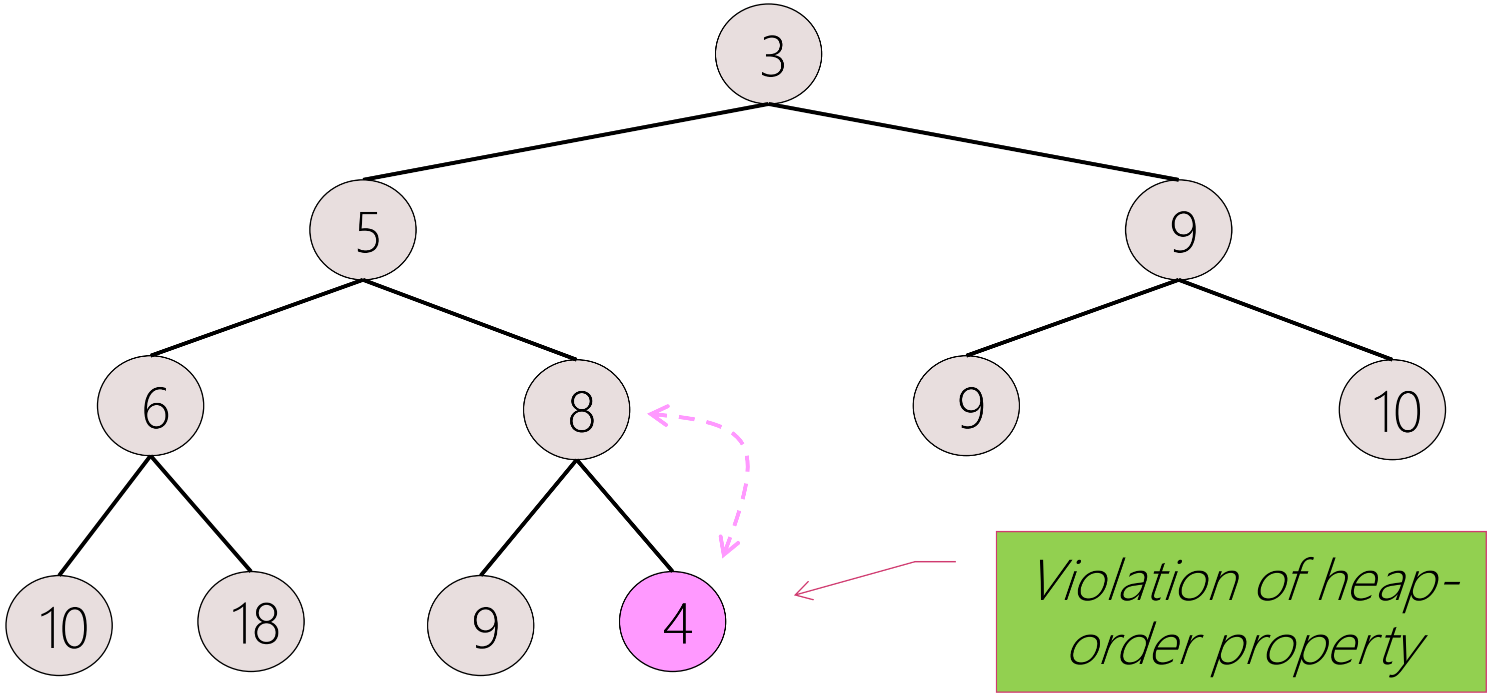
- Now, we want to insert a new element x into the heap H
  - *Insert(x, H), p(x) = 4*

Insert into Min-Heap: *Hole Creation*



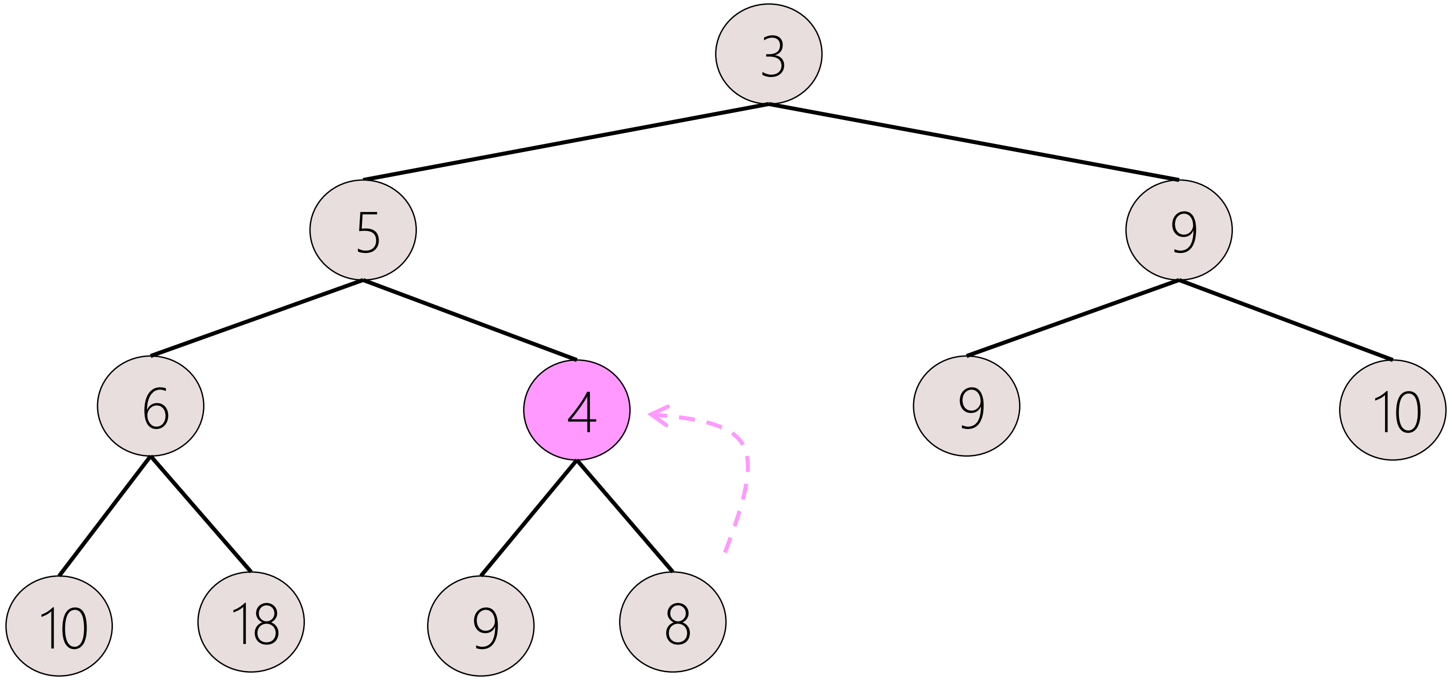
- Step-1: Create a hole & then store *x* in it
  - To satisfy the structural property (CBT), a new node must be added to the *rightmost* position of the *lowest* level

Insert into Min-Heap: *Up-Heap Bubbling*



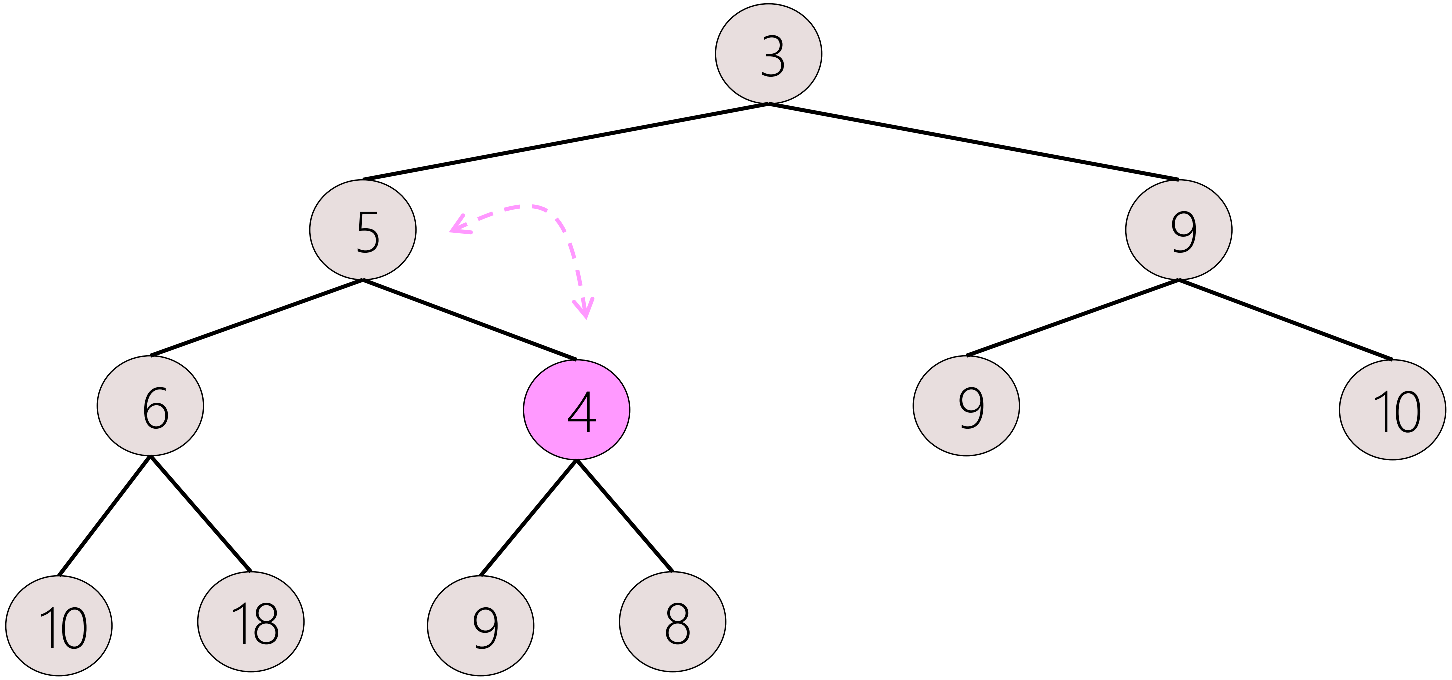
- Step-2: Restore the heap order property
  - Compare $p(x)$ with $p(parent)$ & swap them if necessary
    - Upward movement by swapping 4 & 8

Insert into Min-Heap: *Up-Heap Bubbling*



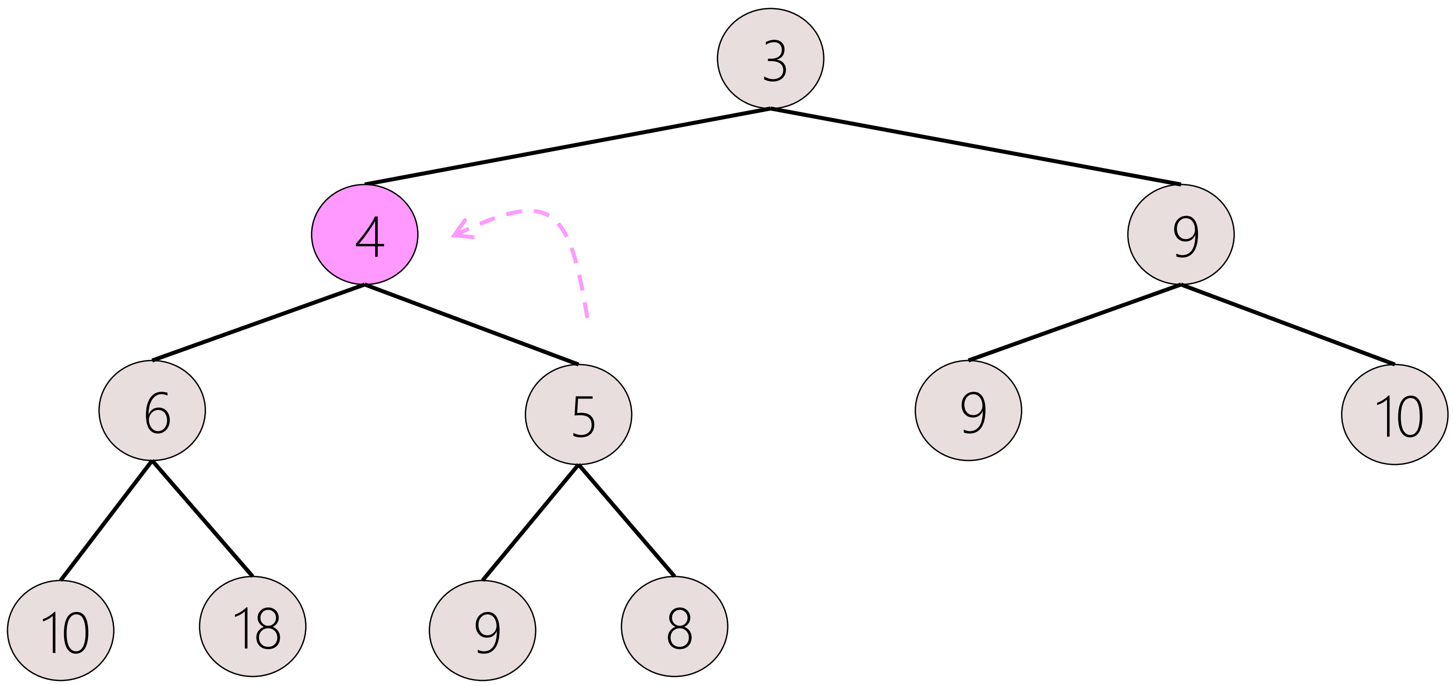
- Step-2: Restore the heap order property
  - Compare *p*(*x*) with *p*(*parent*) & swap them if necessary
    - Upward movement by swapping 4 & 8
    - Called "*Up-heap Bubbling*"

Insert into Min-Heap: *Up-Heap Bubbling*



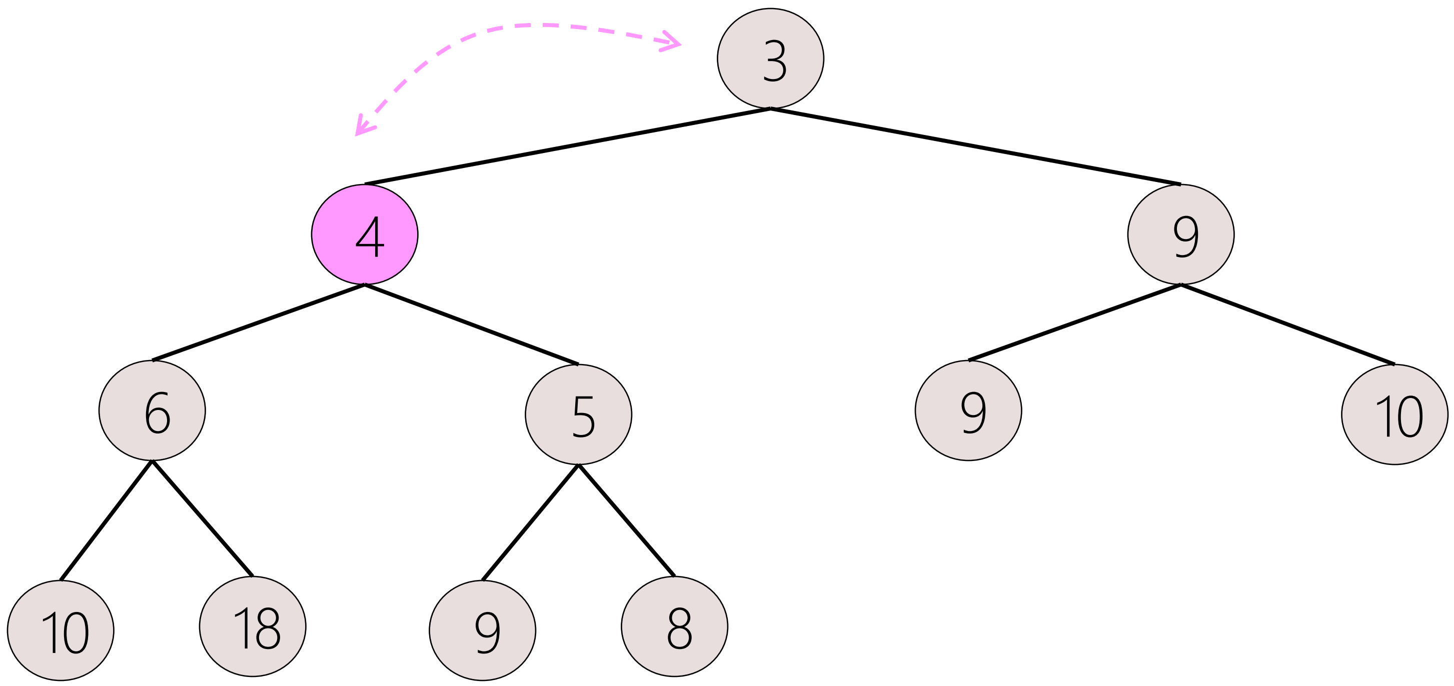
- Step-2: Restore the heap order property
  - Compare $p(x)$ with $p(parent)$ & swap them if necessary
    - Upward movement by swapping 4 & 5

Insert into Min-Heap: *Up-Heap Bubbling*



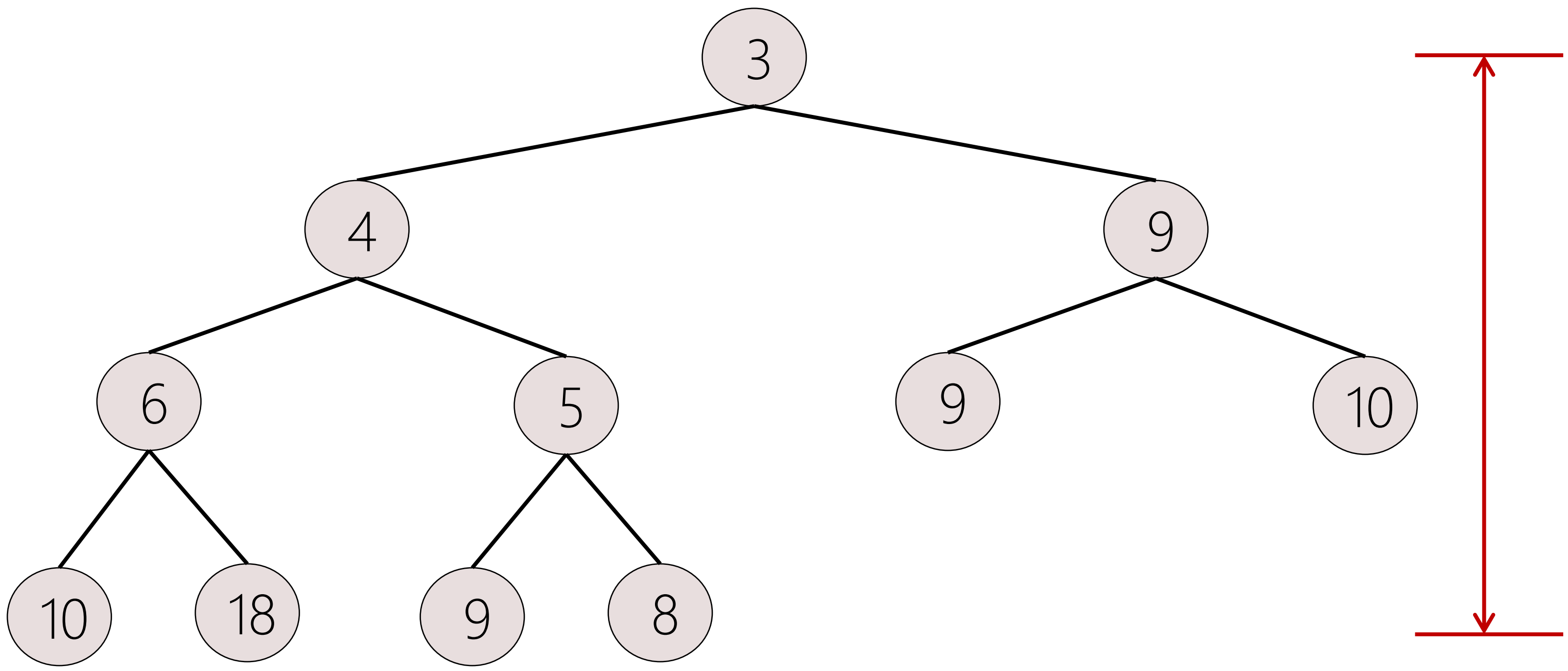
- Step-2: Restore the heap order property
  - Compare *p*(*x*) with *p*(*parent*) & swap them if necessary
    - Upward movement by swapping 4 & 5

Insert into Min-Heap: *Up-Heap Bubbling*



- Step-2: Restore the heap order property
  - Compare *p*(*x*) with *p*(*parent*) & swap them if necessary
    - No more up-heap bubbling
- Insertion completed

Insert into Min-Heap: Time Complexity

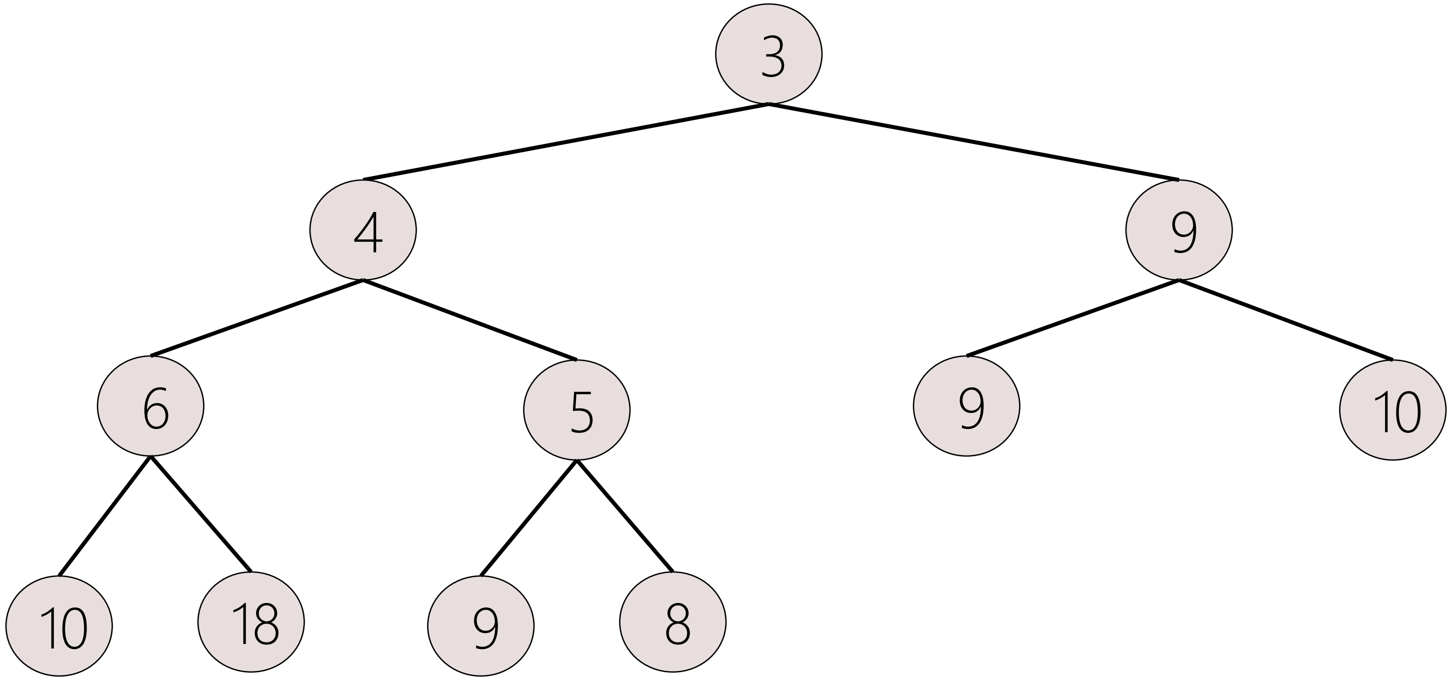


$height(H) = O(\log(n))$

- Time complexity of insertion?

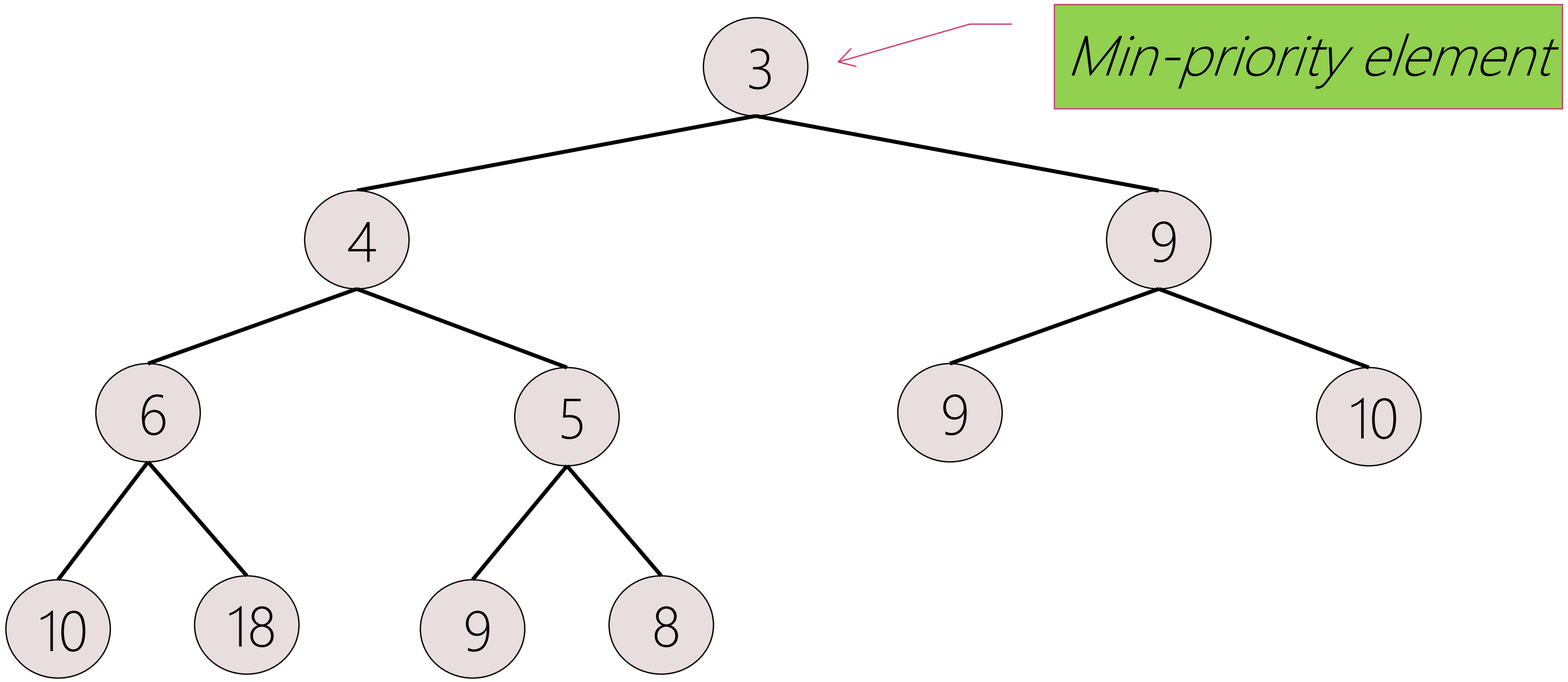
$$= O(\log(n))$$

DeleteMin from Min-Heap



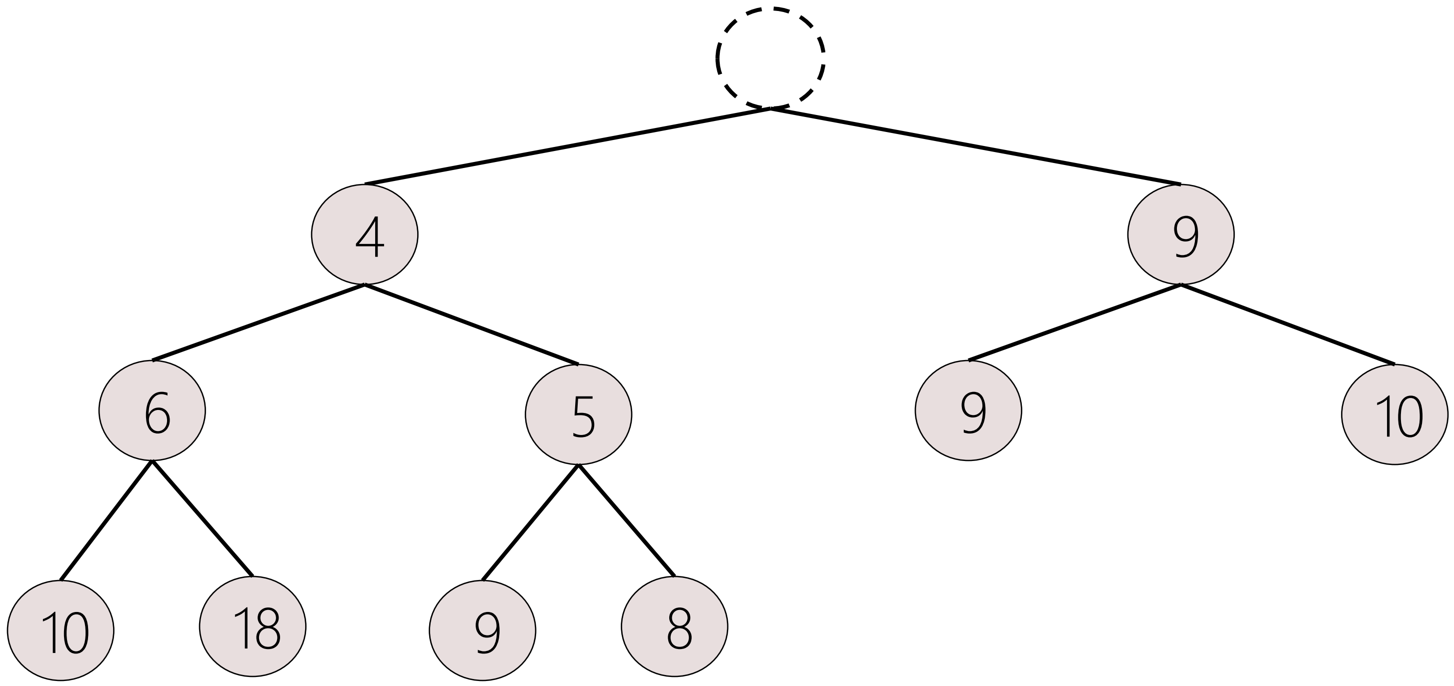
- We want to delete an element with the lowest priority
  - *DeleteMin(H)*

DeleteMin from Min-Heap



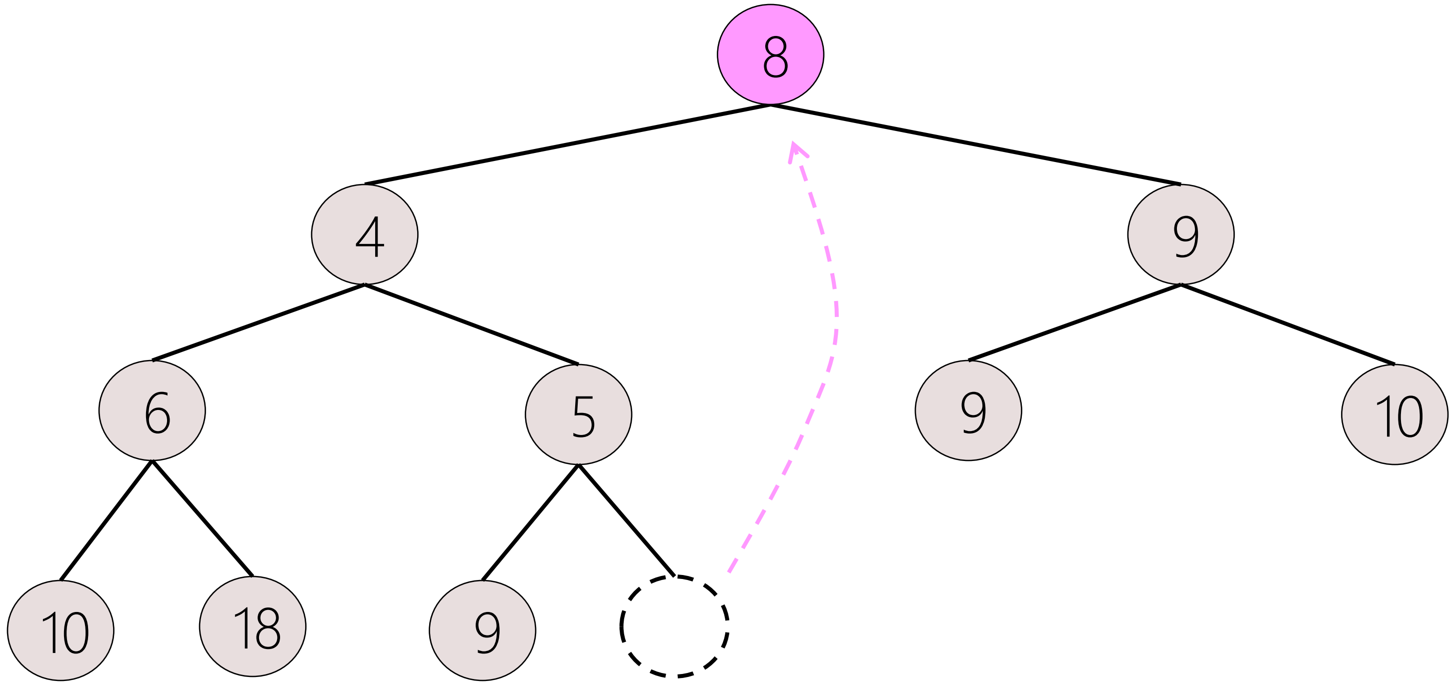
- Step-1: Remove the root & then move the last element to the hole (root)

DeleteMin: Remove Root



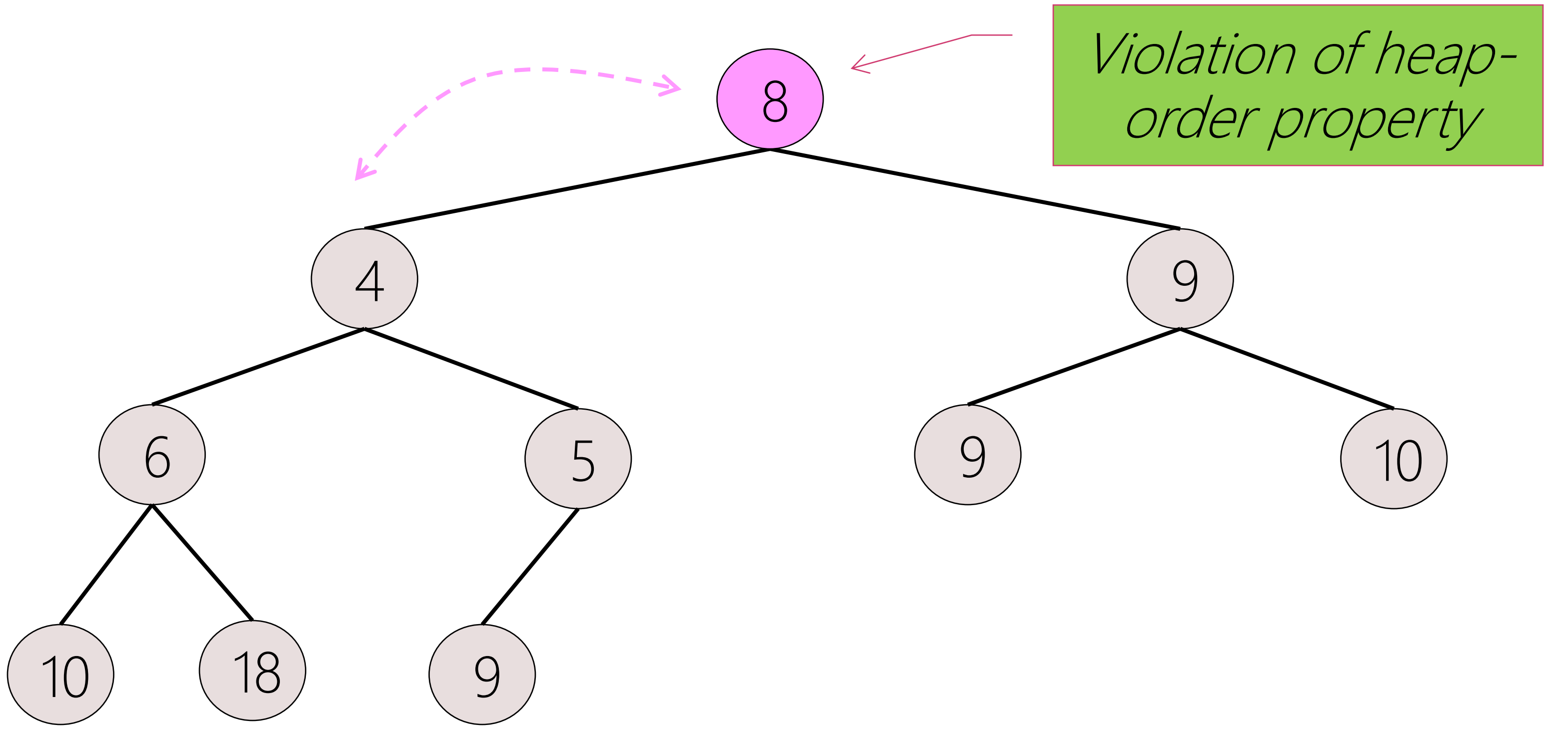
- Step-1: Remove the root & then move the last element to the hole (root)

DeleteMin: Move Last One



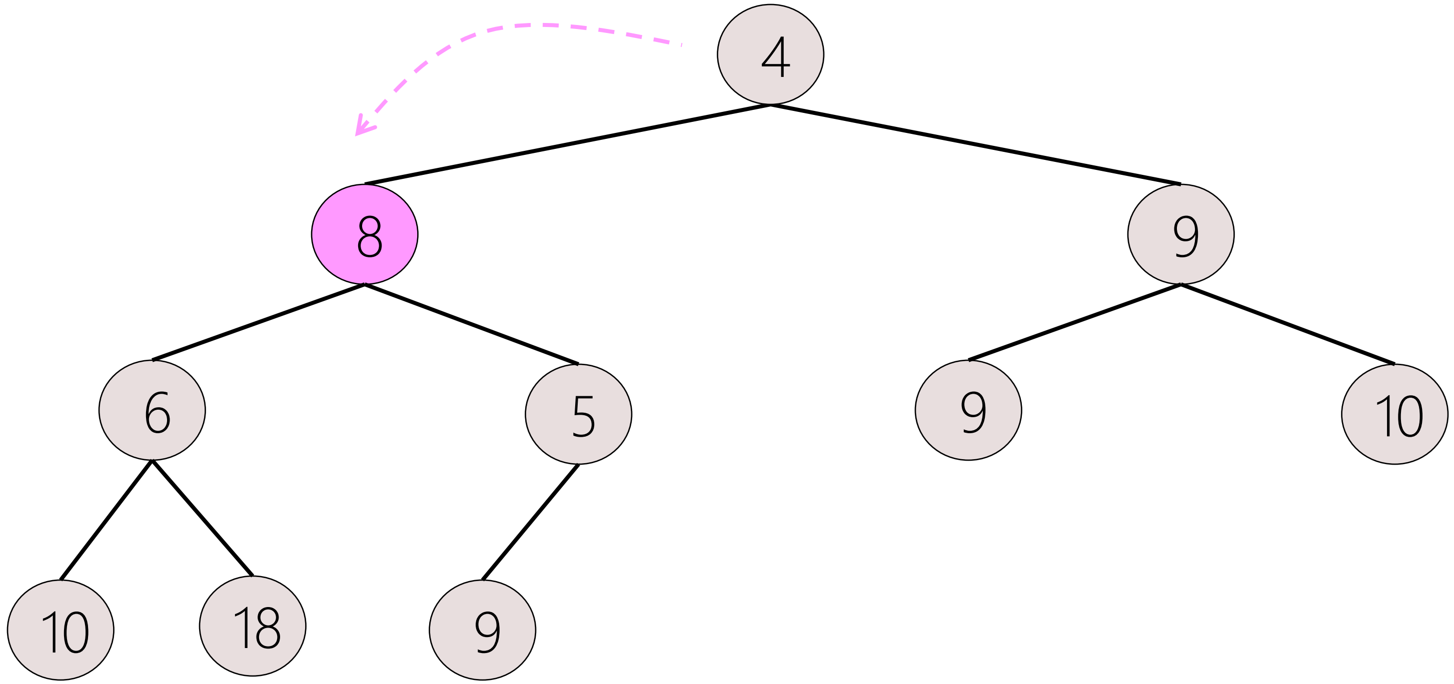
- Step-1: Remove the root & then move the last element to the hole (root)

DeleteMin: Down-Heap Bubbling



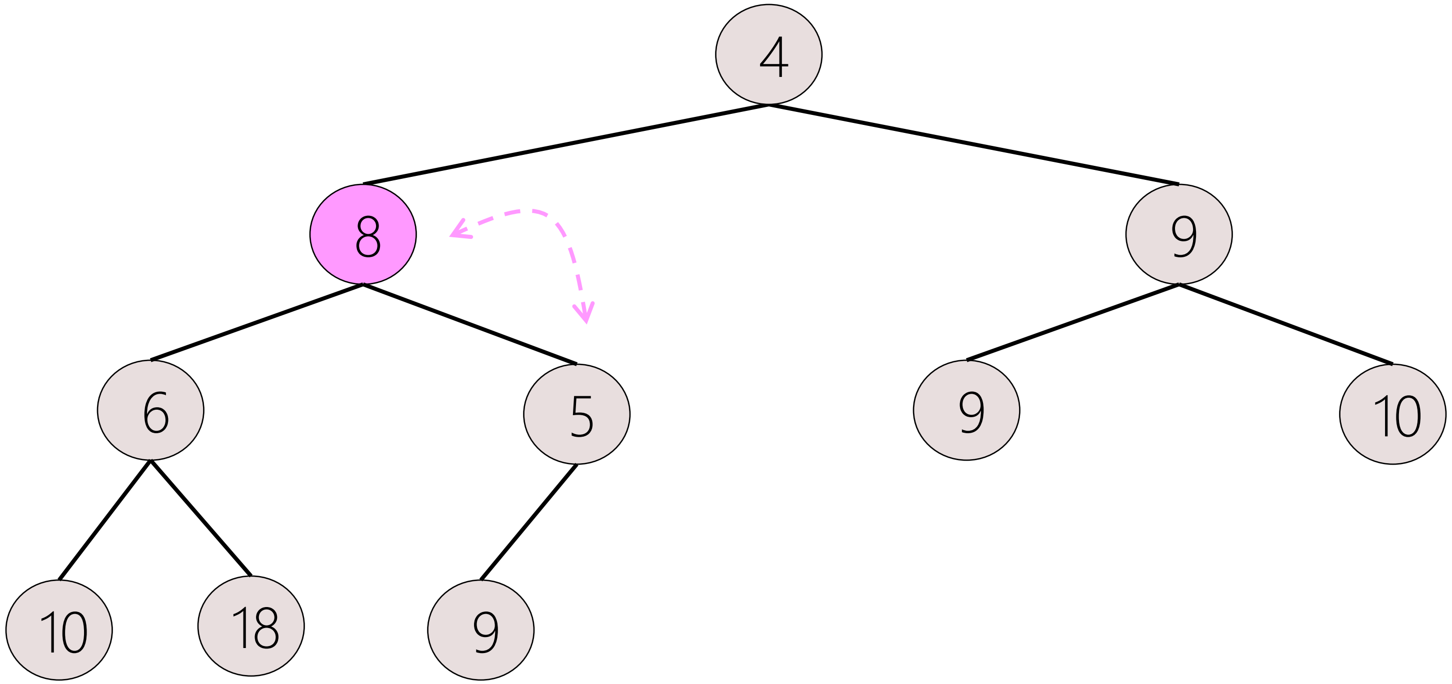
- Step-2: Restore the heap-order property
  - Compare $p(x)$ with $p(children)$ & swap them if necessary
    - Downward movement by swapping 8 & 4

DeleteMin: Down-Heap Bubbling



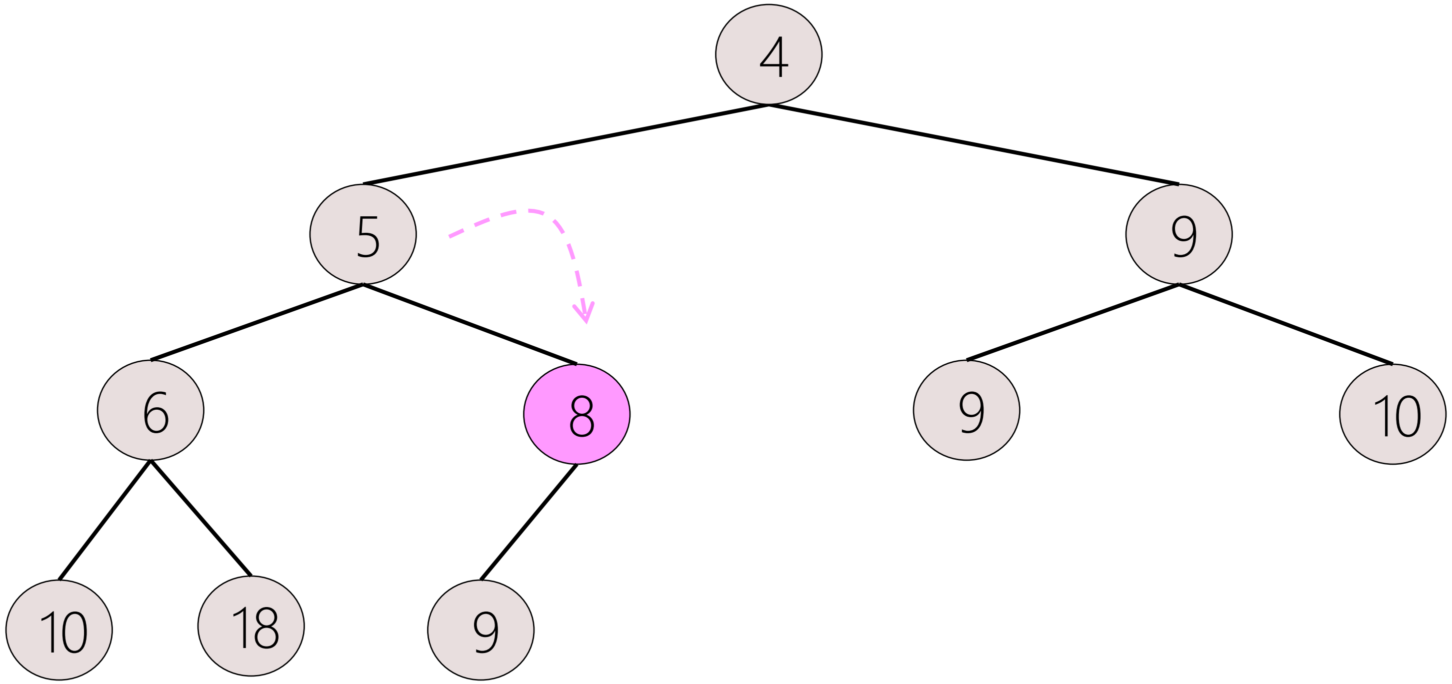
- Step-2: Restore the heap-order property
  - Compare $p(x)$ with $p(children)$ & swap them if necessary
    - Downward movement by swapping 8 & 4
    - Called "*Down-heap Bubbling*"

DeleteMin: Down-Heap Bubbling



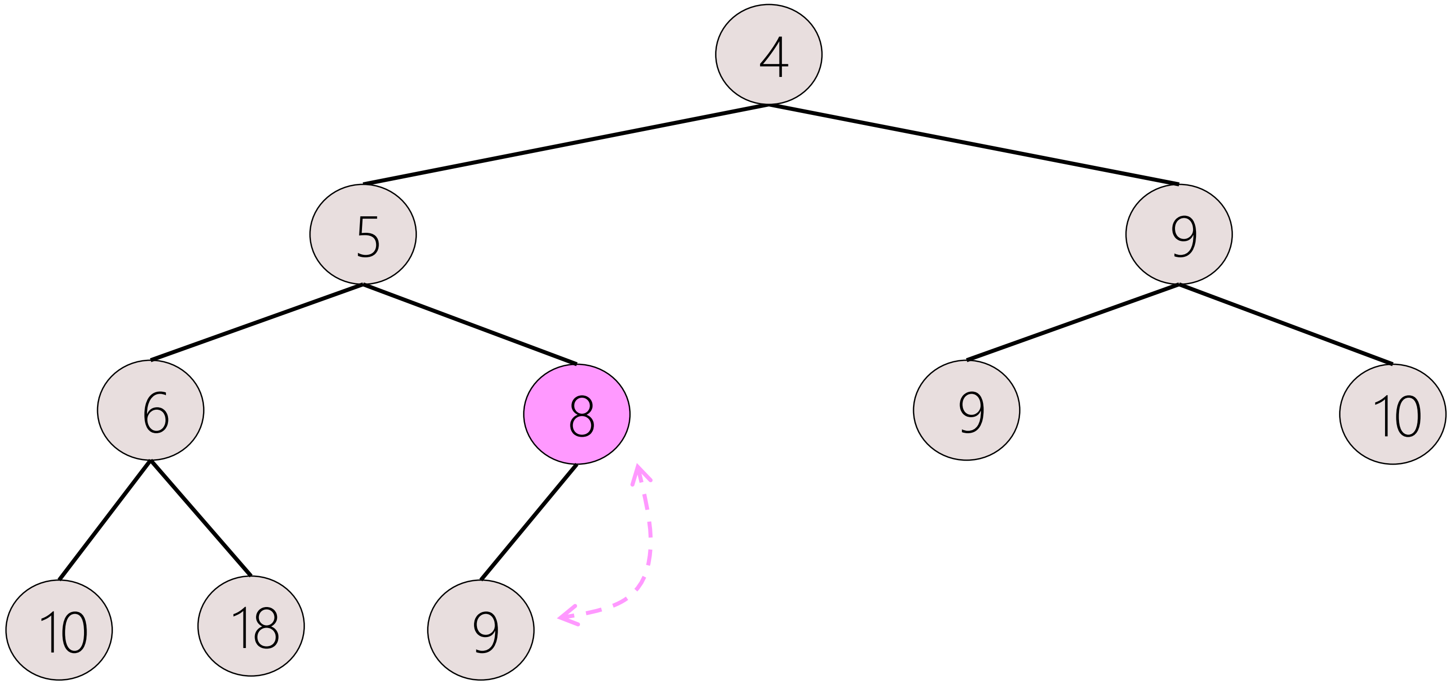
- Step-2: Restore the heap-order property
  - Compare $p(x)$ with $p(children)$ & swap them if necessary
    - Downward movement by swapping 8 & 5

DeleteMin: Down-Heap Bubbling



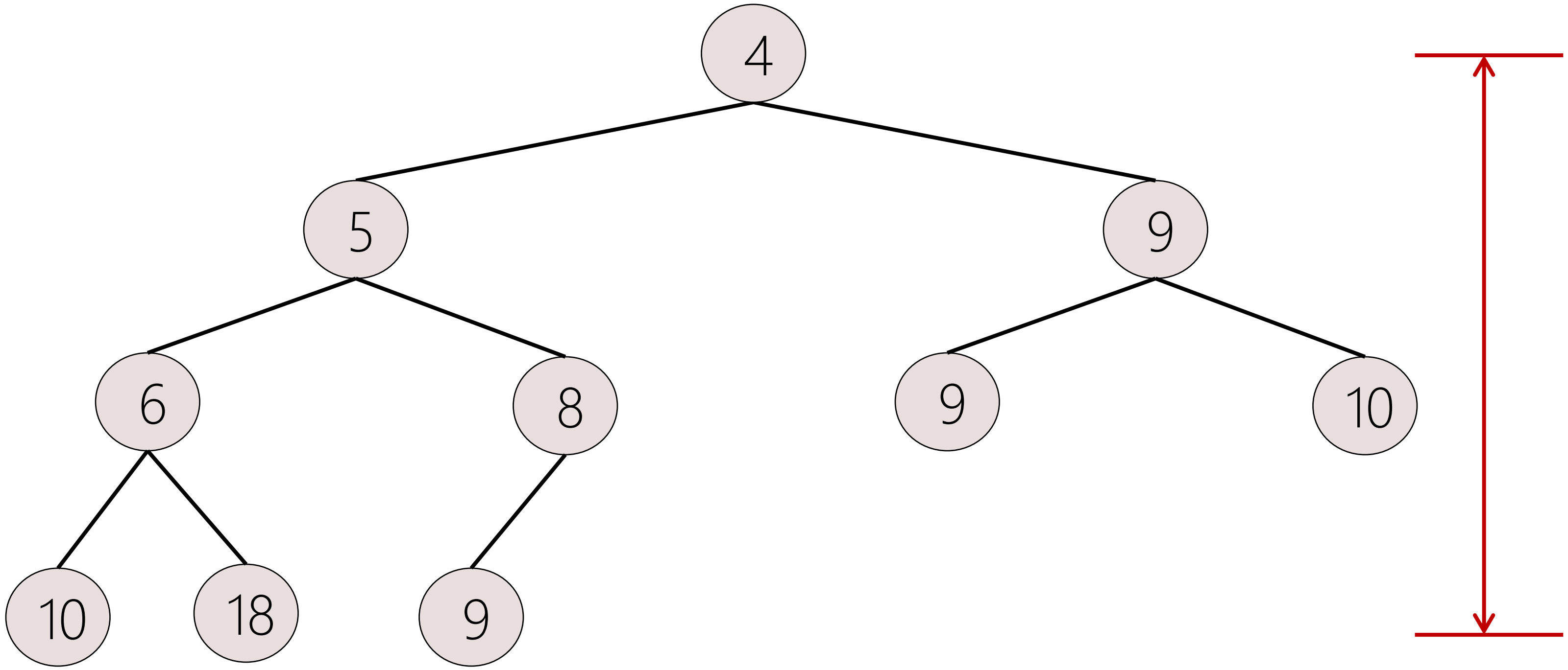
- Step-2: Restore the heap-order property
  - Compare *p*(*x*) with *p*(*children*) & swap them if necessary
    - Downward movement by swapping 8 & 5

DeleteMin: Down-Heap Bubbling



- Step-2: Restore the heap-order property
  - Compare *p*(*x*) with *p*(*children*) & swap them if necessary
    - No more down-heap bubbling
- DeleteMin completed

DeleteMin: Time Complexity



$height(H) = O(\log(n))$

- Time complexity of DeleteMin ?

$= O(\log(n))$

Heap-Building Process

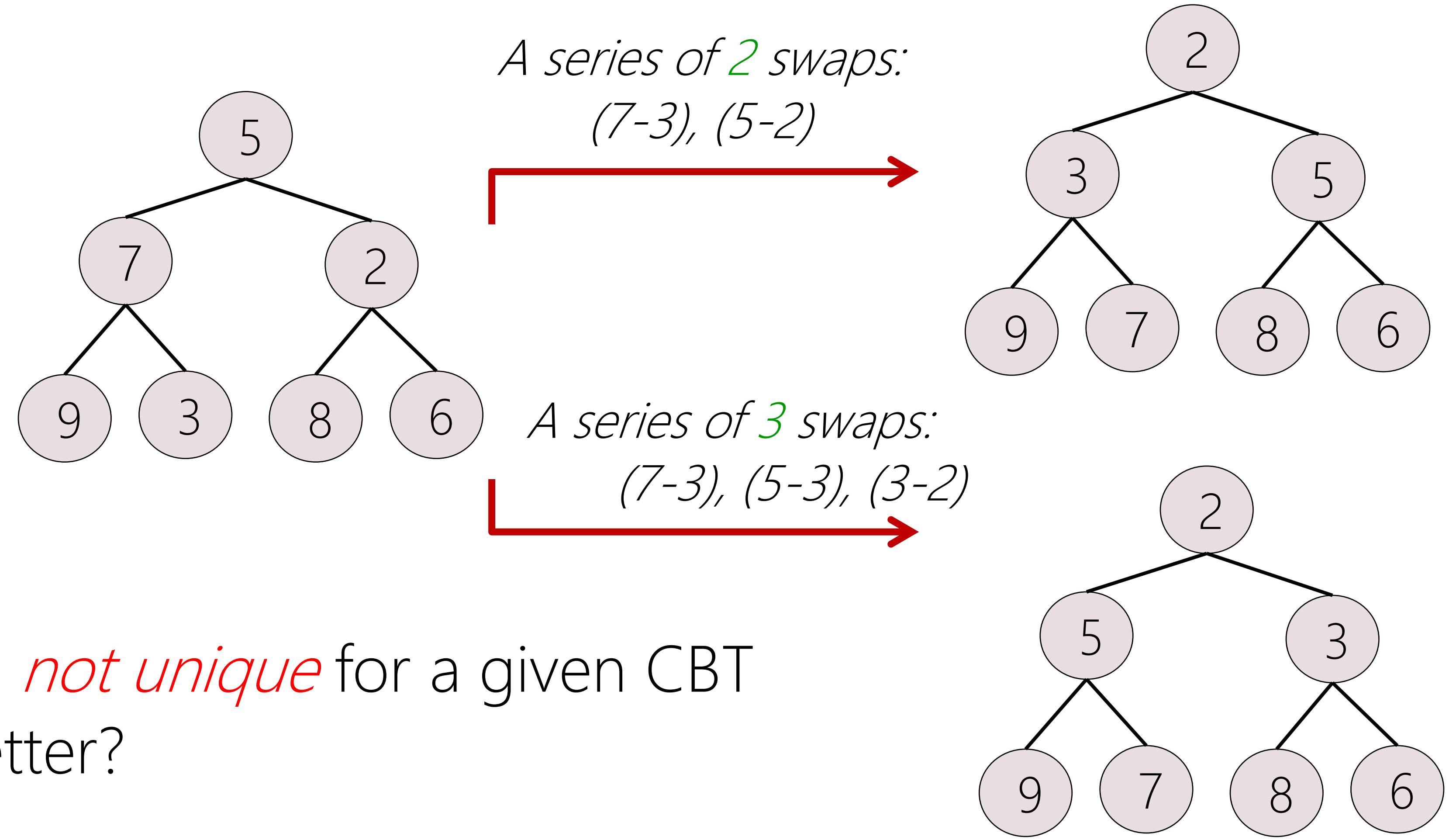
- Obvious way
  - Insert *n* elements one at a time
    - (i.e.) *n* successive insertions
  - Time complexity ?

    = $O(n \log n)$ in the worst case

- More efficient way
  - When all *n* elements are available in advance
  - Algorithm of O(*n*) time:
  - How?

> - Place *n* elements into an *array-based CBT* in any order
> - *Heapify* the complete binary tree

Many Ways of Rearranging CBT



- The heap is *not unique* for a given CBT
- Which is better?

How to Heapify CBT

- How do we pick the *best* rearrangement

- Algorithm (from induction)
  - Input: Array-based CBT

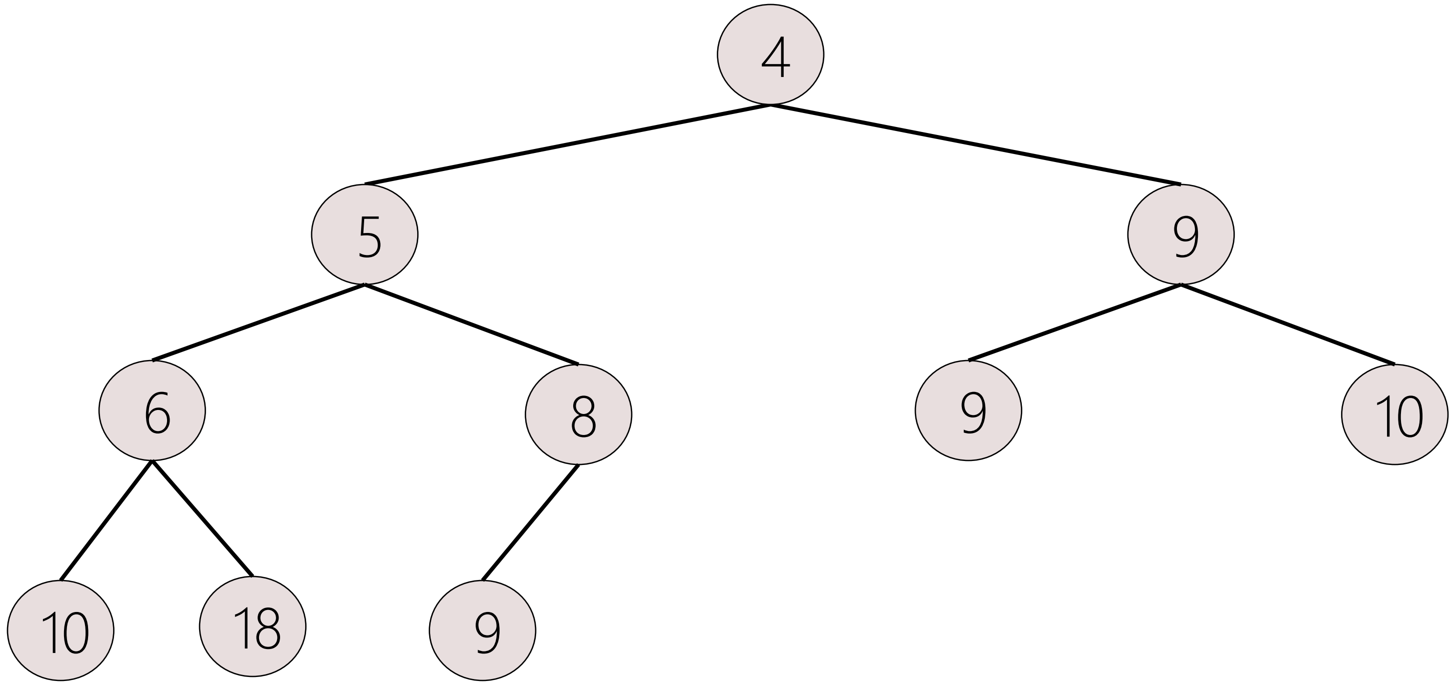
1) Start with the rightmost array position k that has a child

   (NB: the last non-terminal node at index $k = \lfloor n/2 \rfloor$), n is # of nodes

2) If the subtree rooted at k is not a min-heap
   - Rearrange the subtree (by pushing down k until it reaches a level where it is less than its children, or is a leaf node)

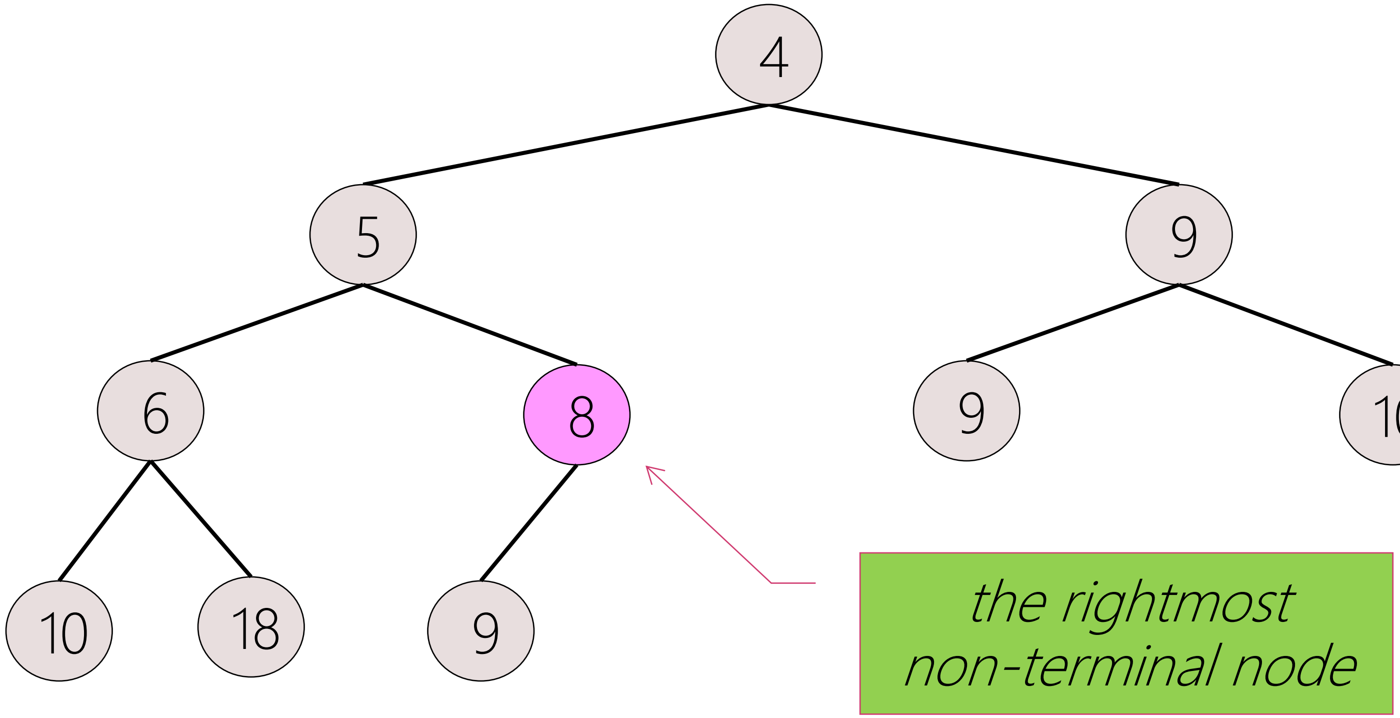
3) Repeat the step-2 at k-1, k-2, …, 1 (from the high index of the array to the low index)

Example: Min-Heap → Max-Heap

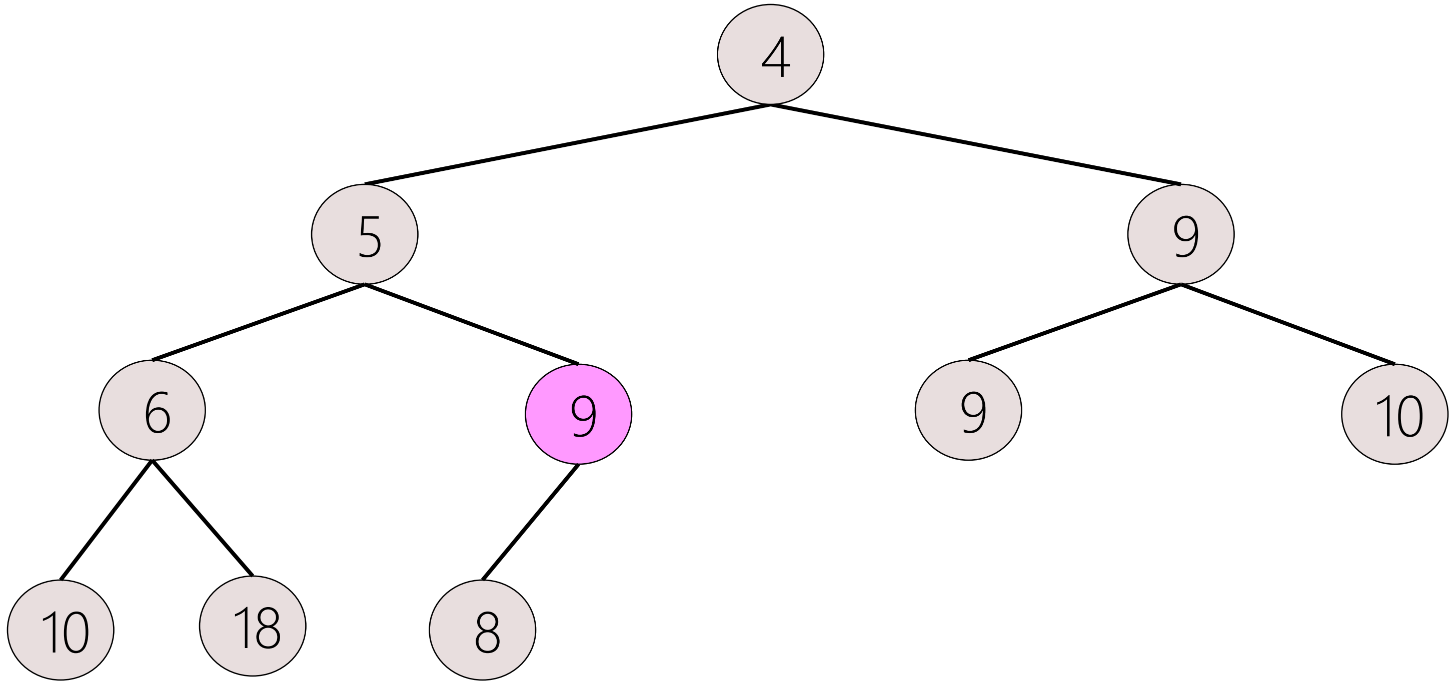


- Heapify the above min-heap into a max-heap

Example: Min-Heap → Max-Heap

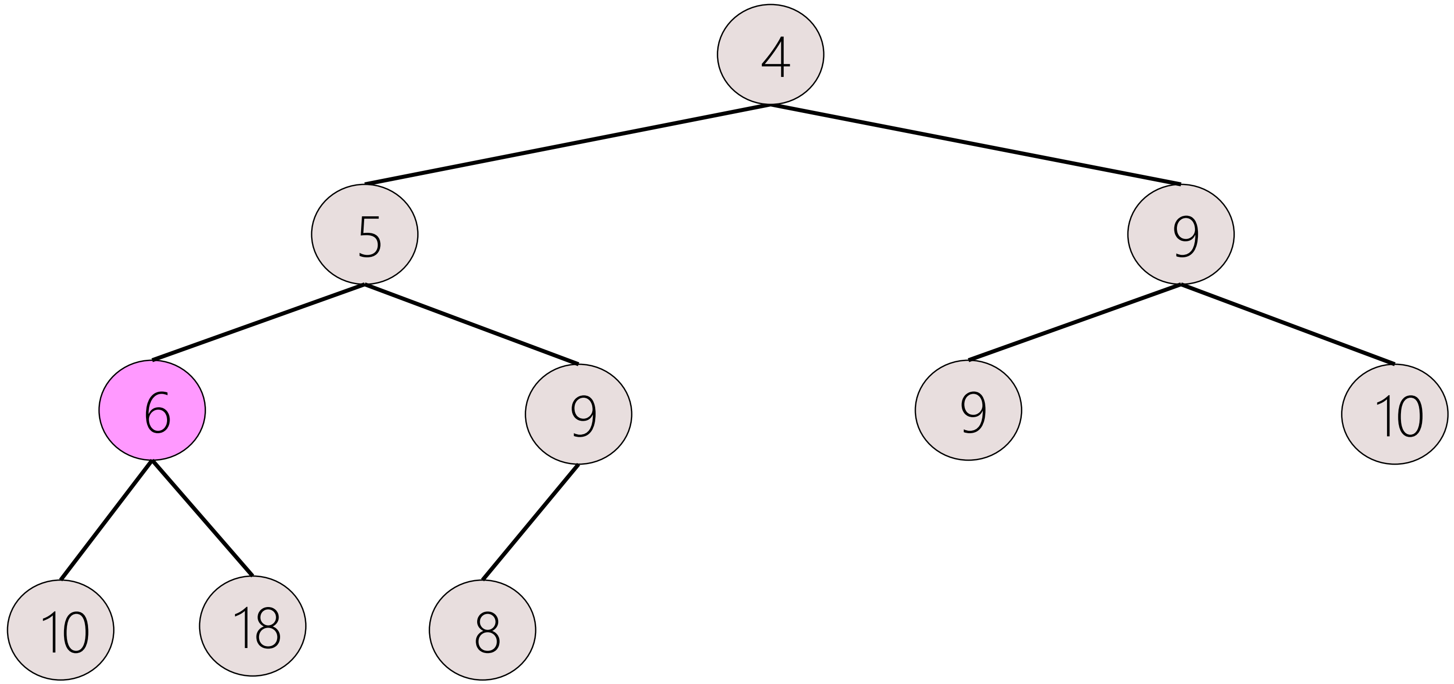


Example: Min-Heap → Max-Heap



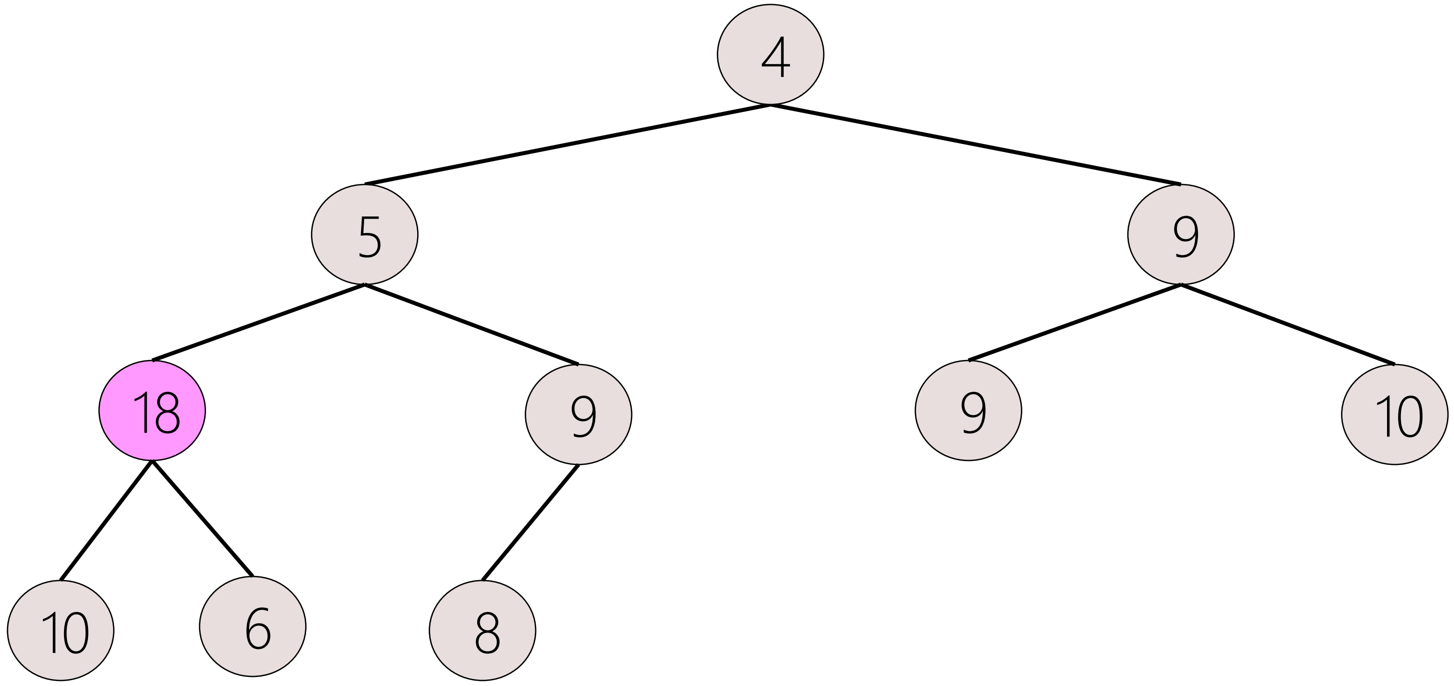
- Down-heap bubbling

Example: Min-Heap → Max-Heap

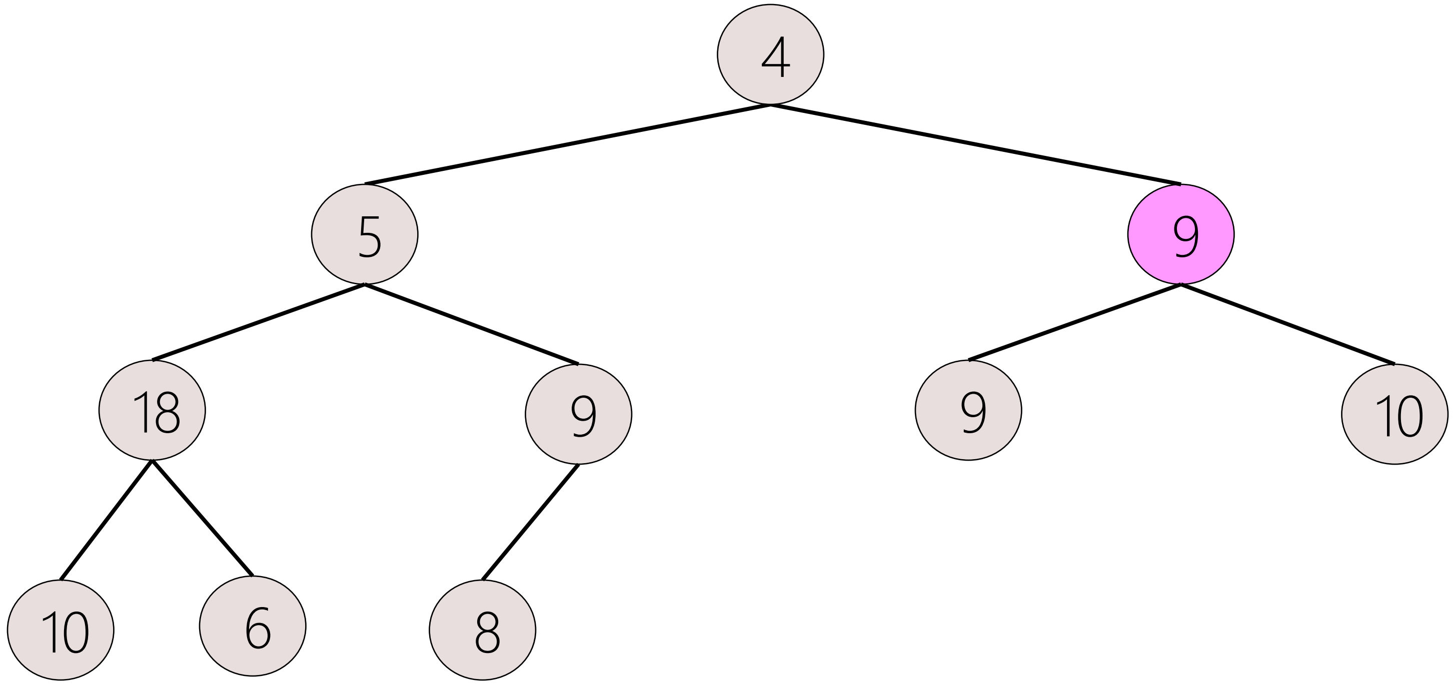


- Move to the next lower array position

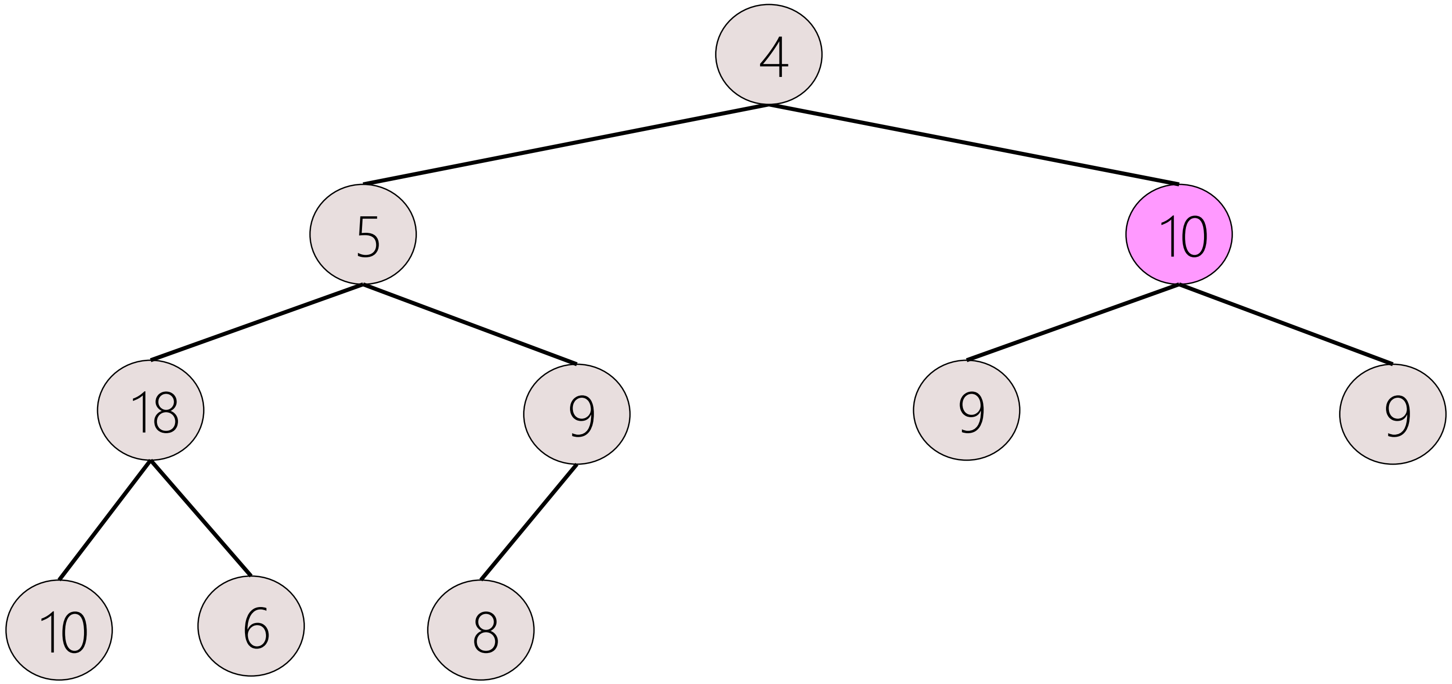
Example: Min-Heap → Max-Heap



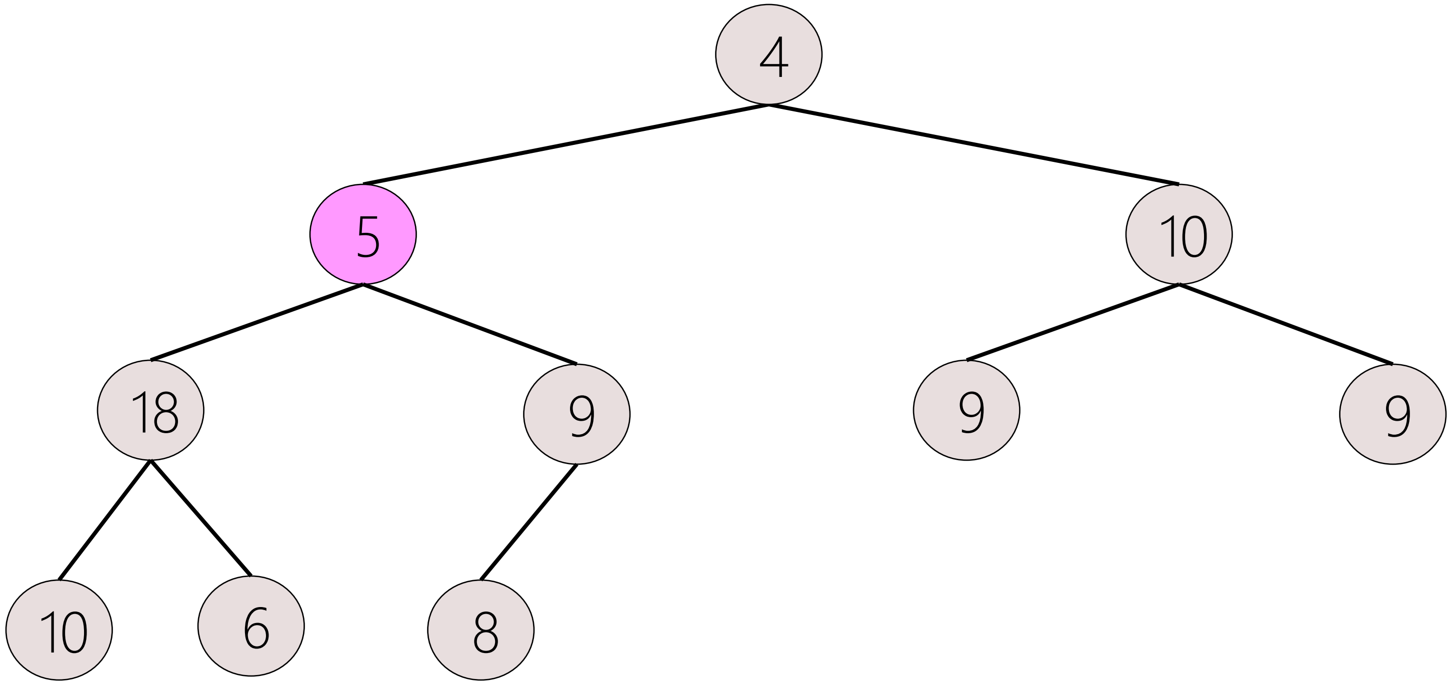
Example: Min-Heap → Max-Heap



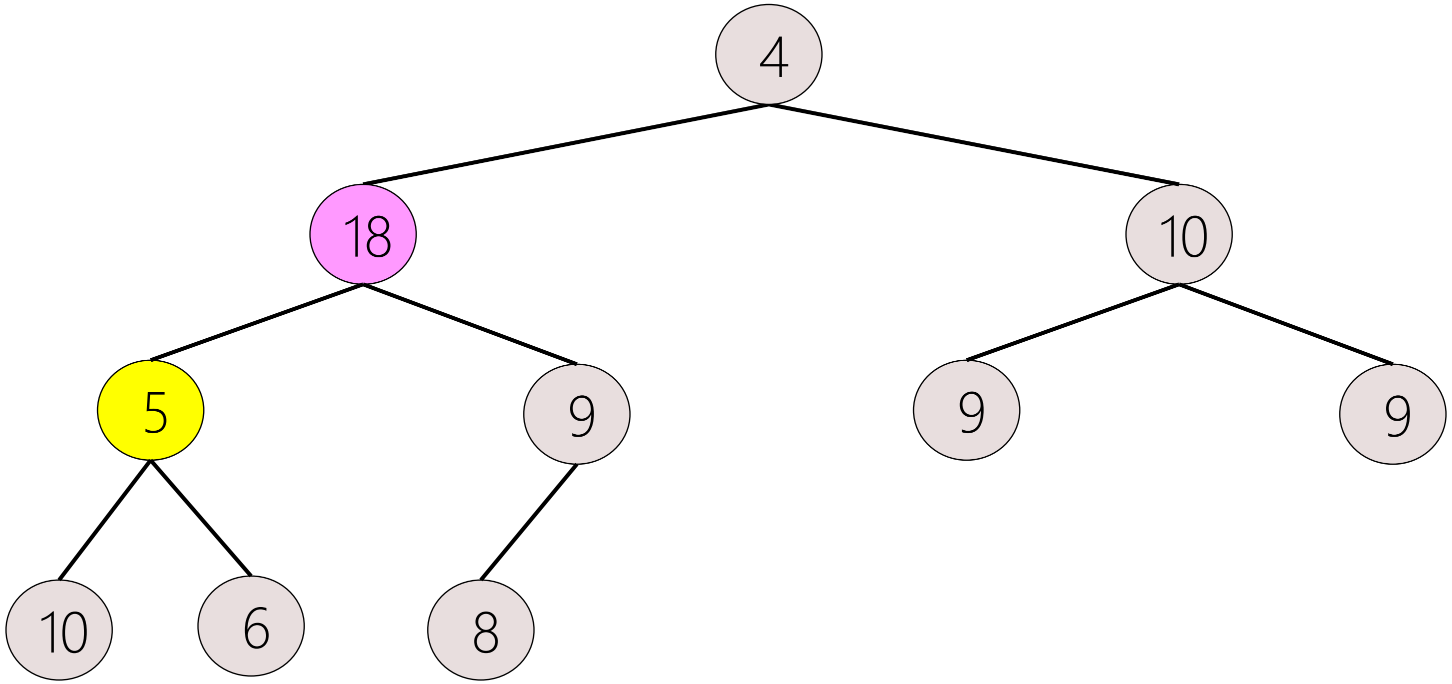
Example: Min-Heap → Max-Heap



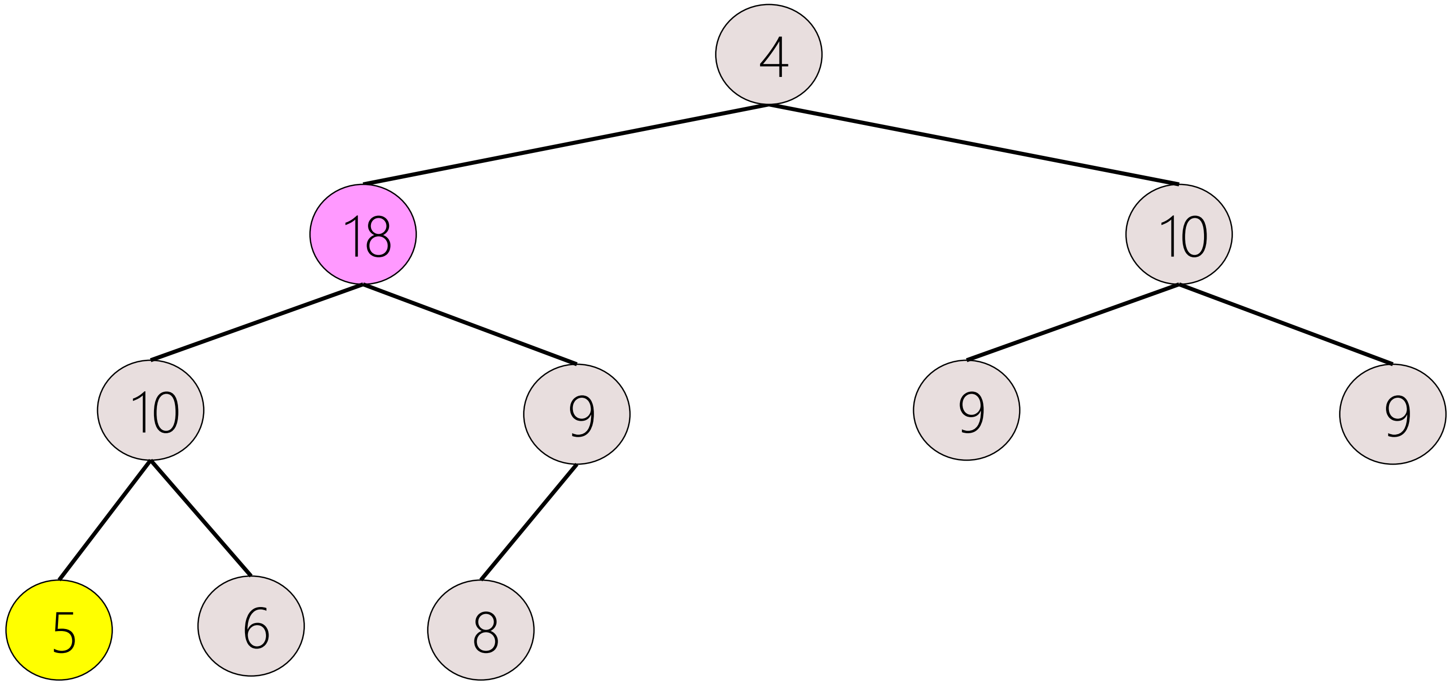
Example: Min-Heap → Max-Heap



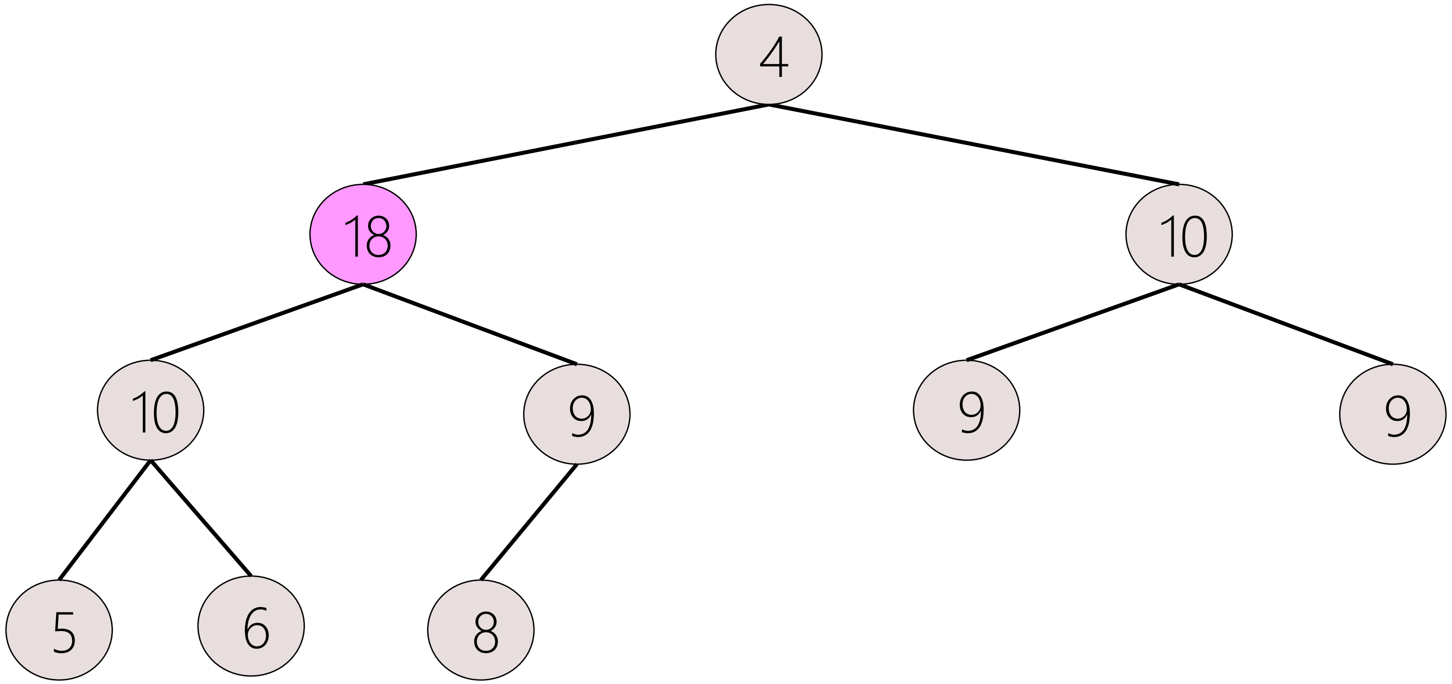
Example: Min-Heap → Max-Heap



Example: Min-Heap → Max-Heap

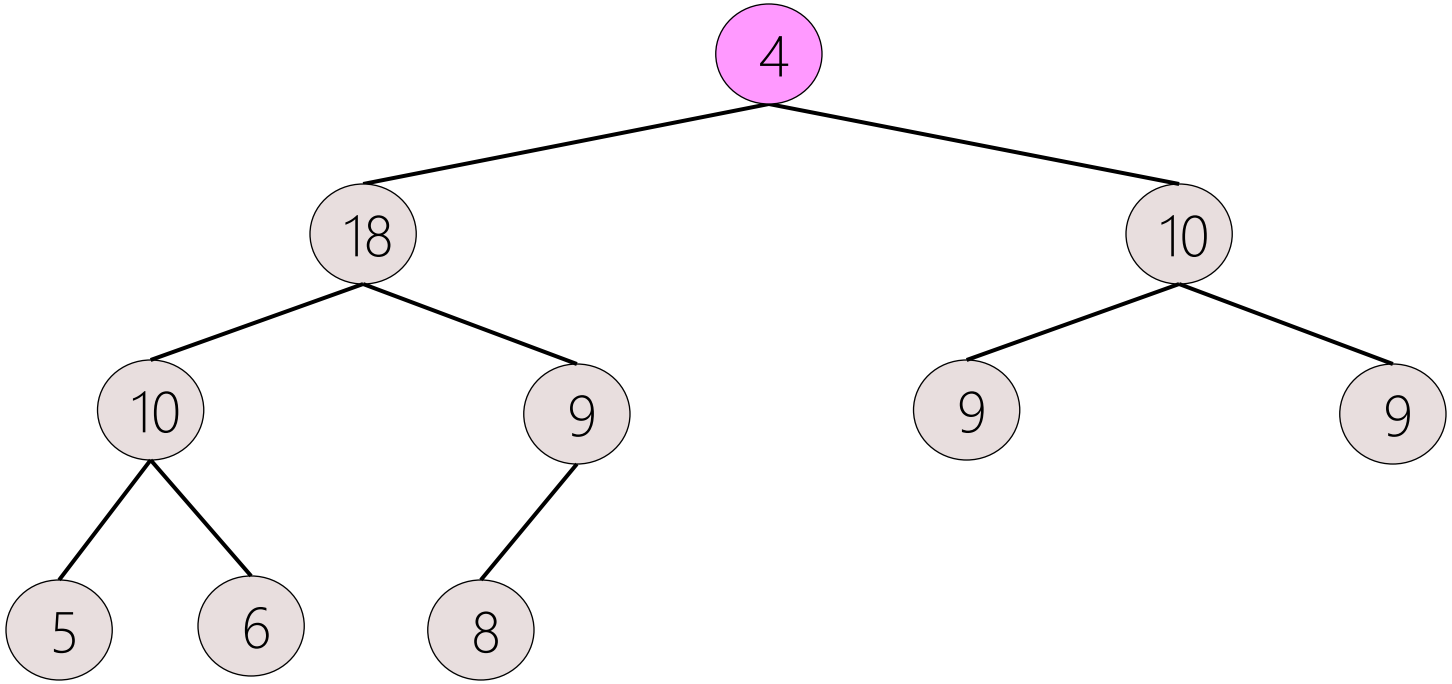


Example: Min-Heap → Max-Heap



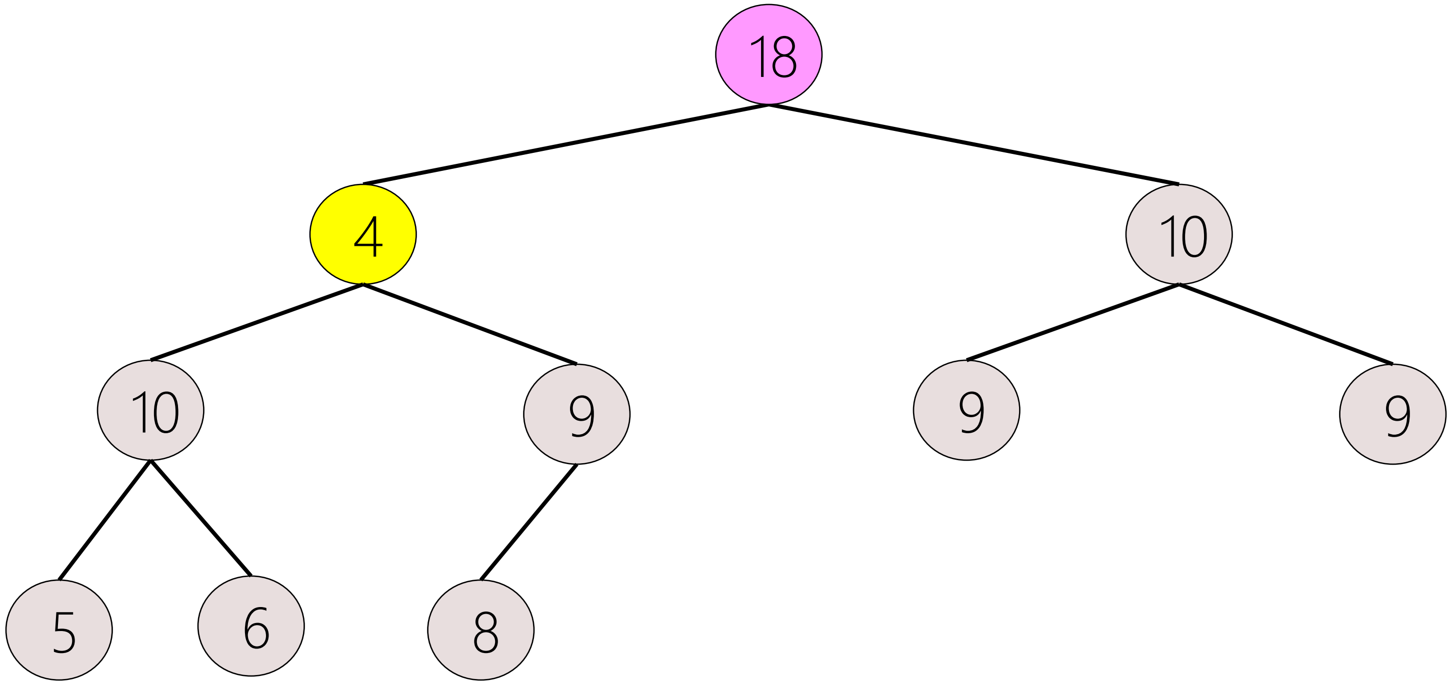
- Done for 5

Example: Min-Heap → Max-Heap

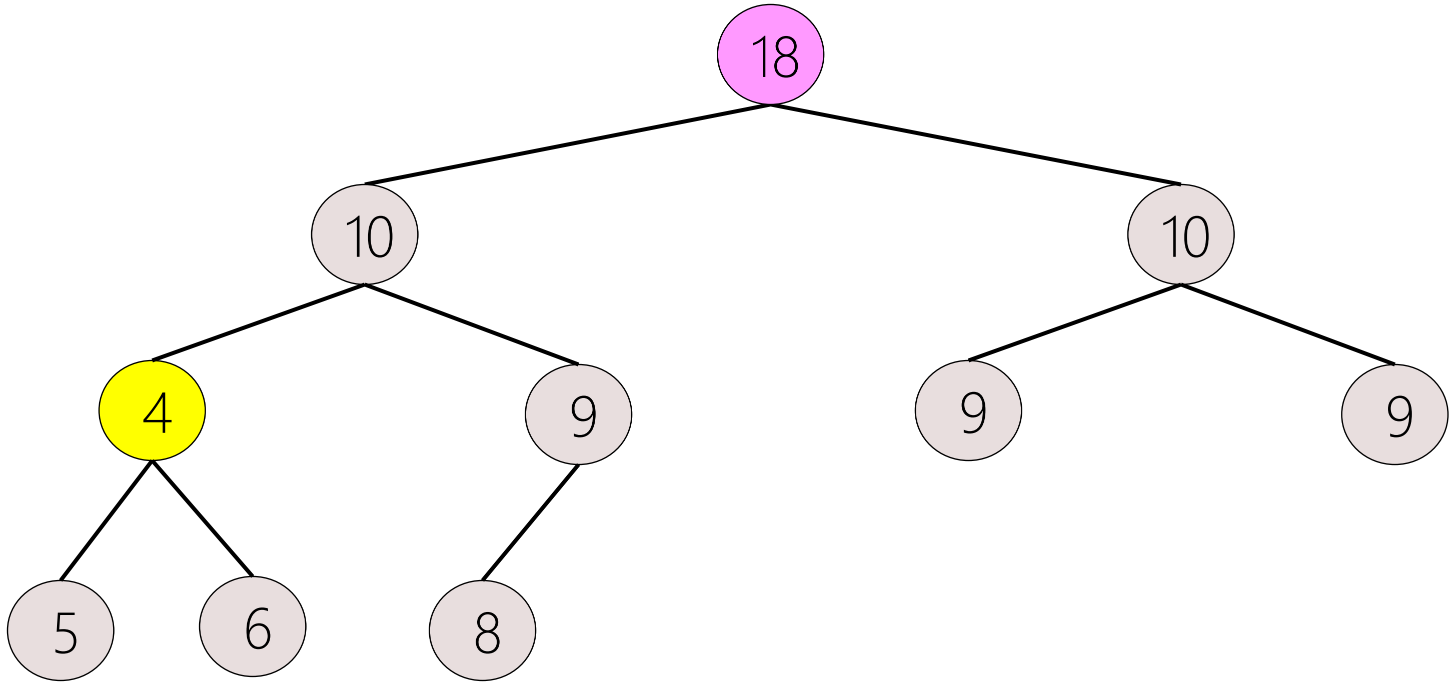


- Move to next lower array position

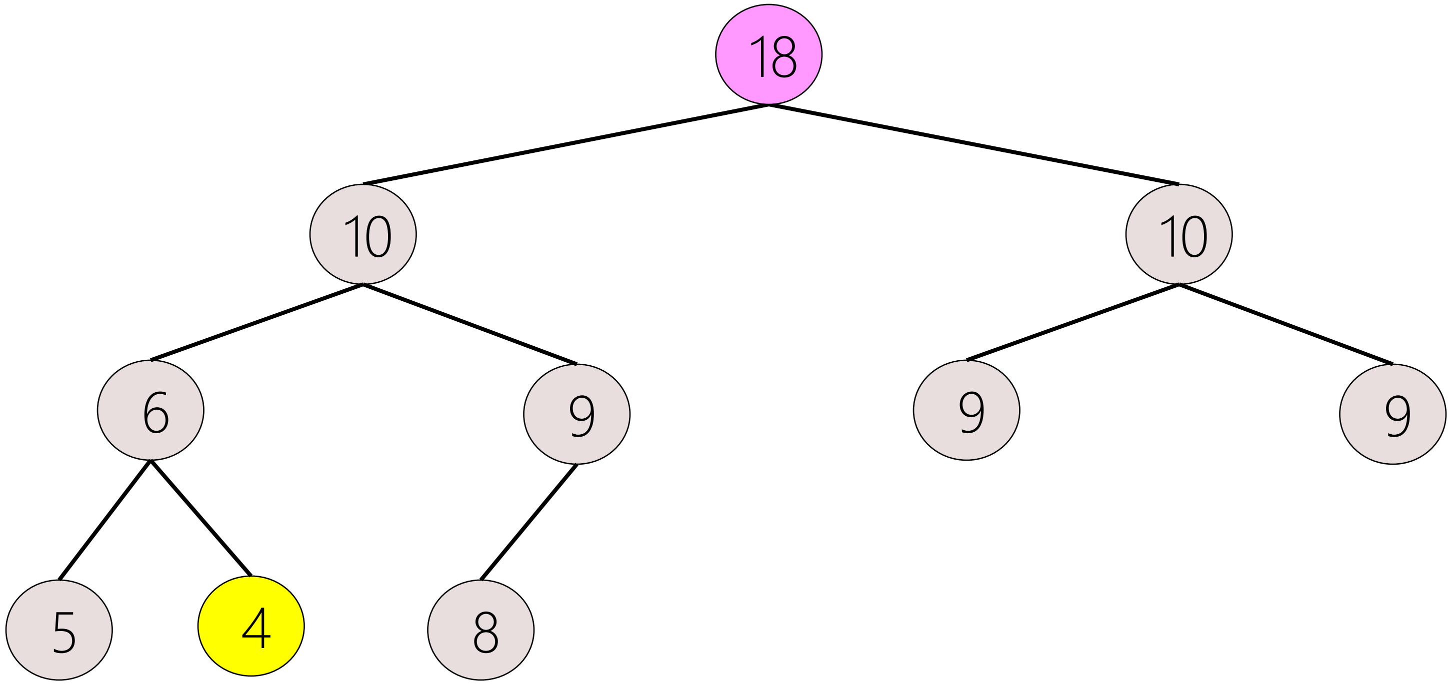
Example: Min-Heap → Max-Heap



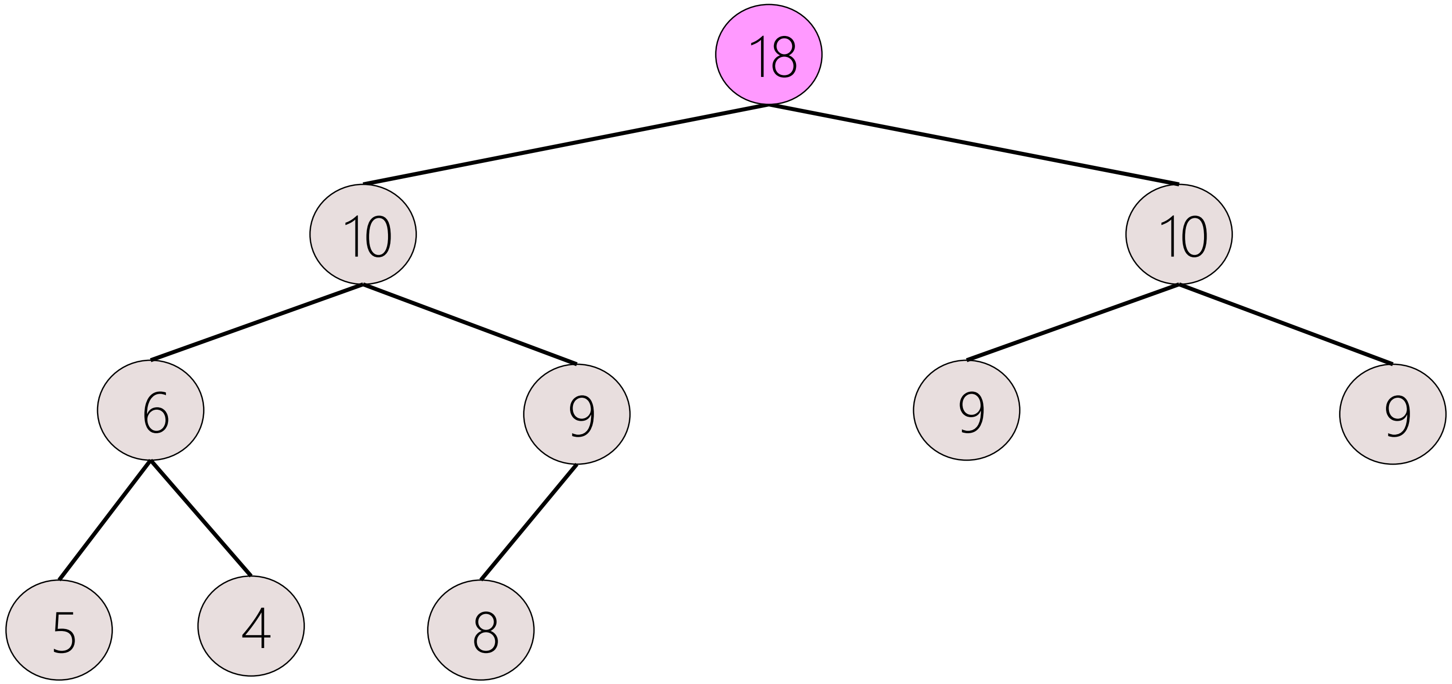
Example: Min-Heap → Max-Heap



Example: Min-Heap → Max-Heap

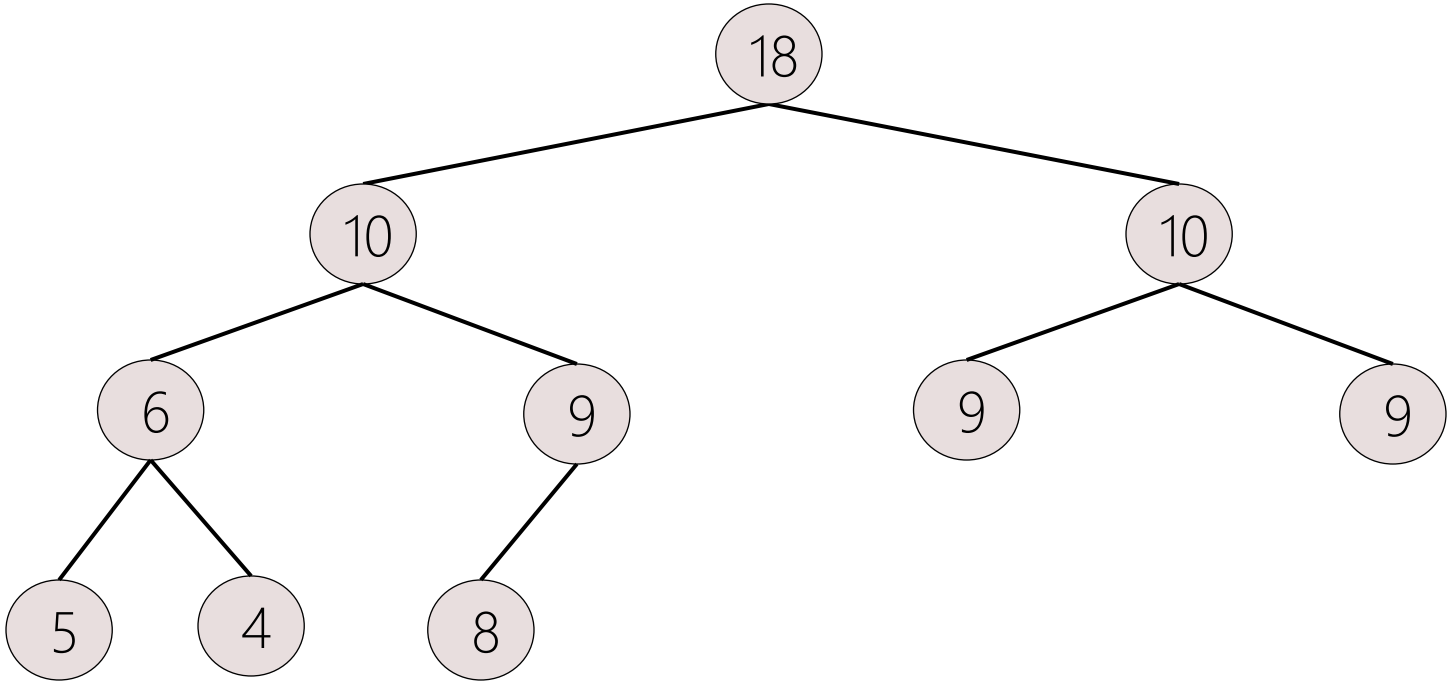


Example: Min-Heap → Max-Heap



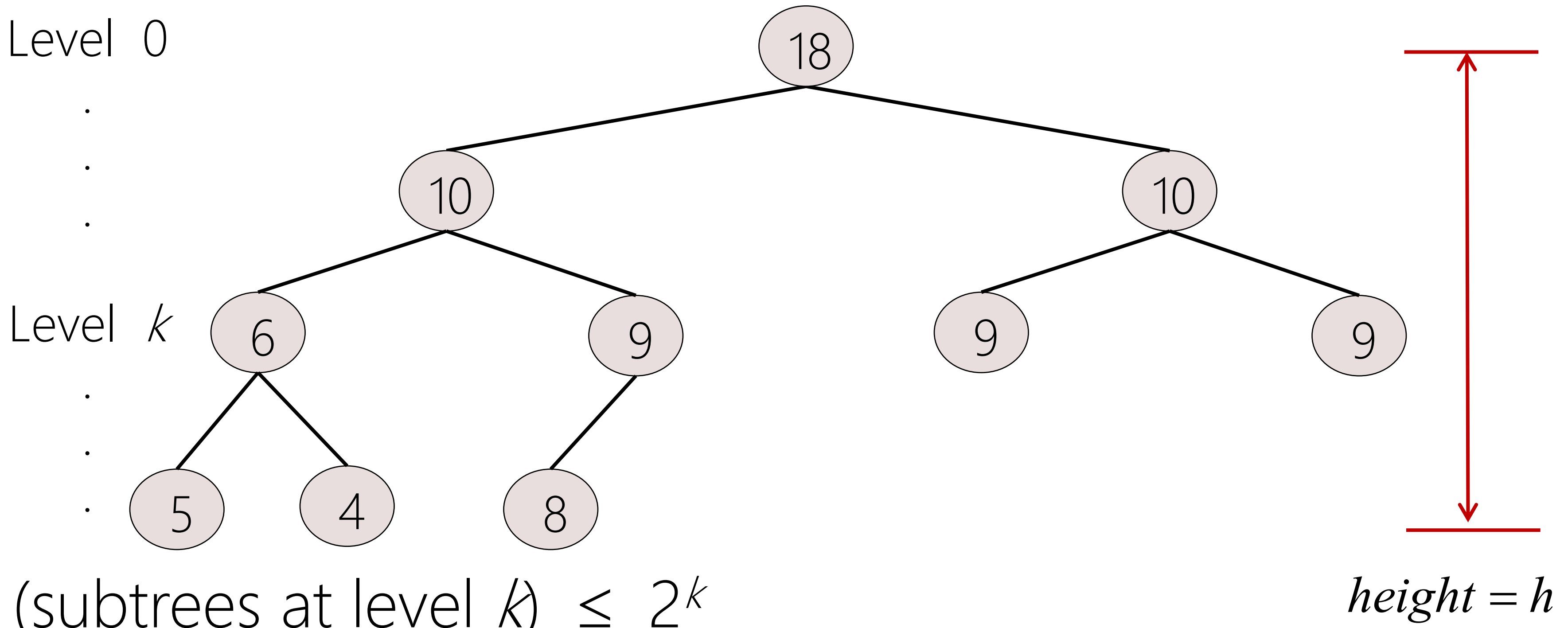
- Done for 4

Example: Min-Heap → Max-Heap



- Done!

Time Complexity of heapifying CBT



- Number (subtrees at level k) $\leq 2^k$
- Time (each subtree) = $O(h - k)$
- Time (all subtrees at level k) $\leq 2^k (h - k)$
- Total time $= O(\sum_{k=0}^{h-1} 2^k (h-k)) = \ldots = O(2^h) = O(n)$

1) Start with the rightmost array position k that has a child
(NB: the last non-terminal node at index $k = \lfloor n/2 \rfloor$), n is # of nodes
2) If the subtree rooted at k is not a min-heap
Rearrange the subtree (by pushing down k until it reaches a level where it is less than its children, or is a leaf node)
3) Repeat the step-2 at k-1, k-2, …, 1 (from the high index of the array to the low index)

References

- Further reading list and references
  - https://en.wikipedia.org/wiki/Priority_queue
  - https://en.wikipedia.org/wiki/Min-max_heap
  - https://en.wikipedia.org/wiki/Binomial_heap

- Slide credit
  - Jaesik Park
  - Seung-Hwan Baek
  - Jong-Hyeok Lee