



堆 Heaps

钮鑫涛 王智彬
Nanjing University
2025 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!



Heap

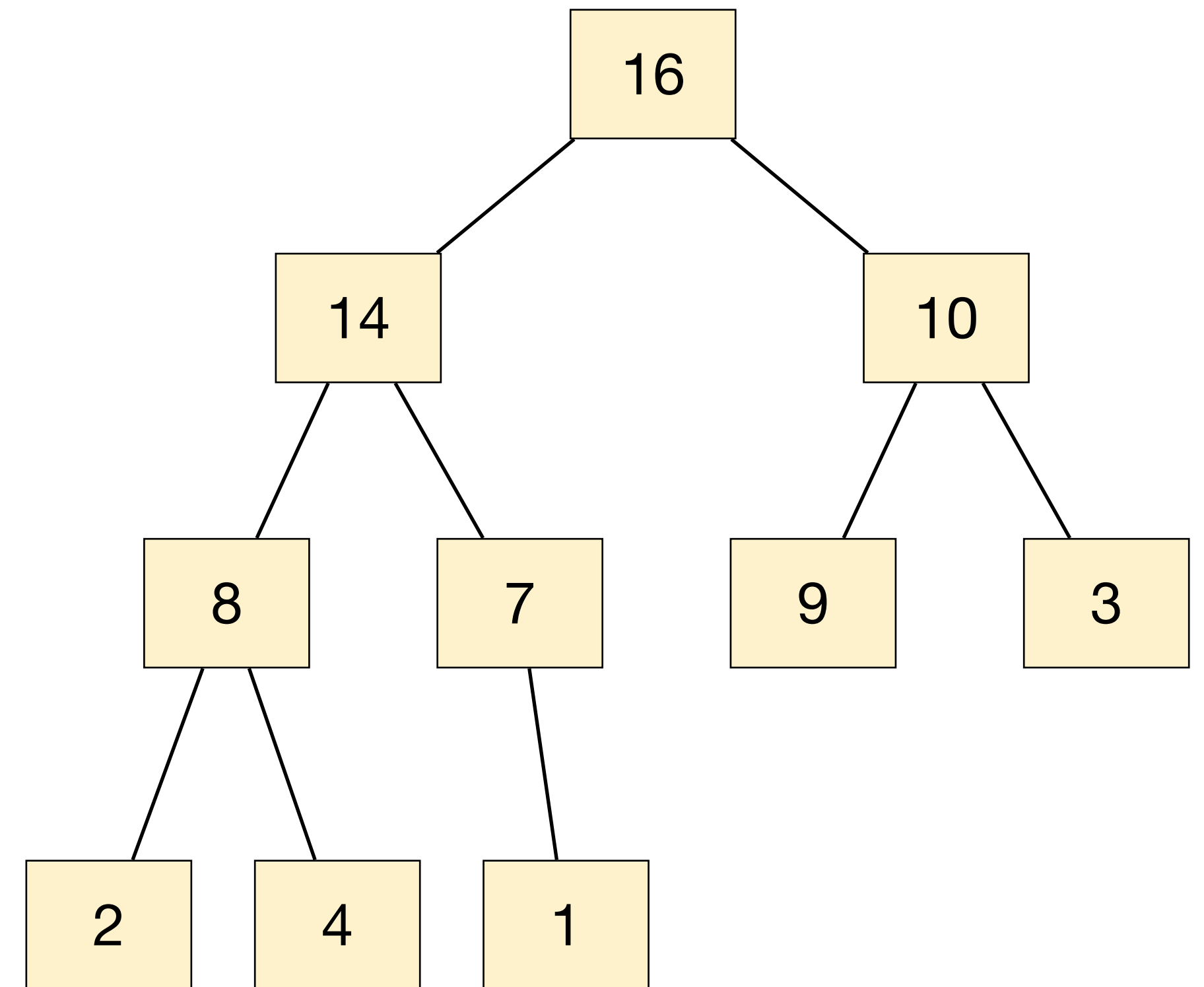
- In computer science, a *heap* is data structure which means “a disorganized pile.”
 - In fact, this word has other meanings in computer science, which refers to *heap memory* used for dynamic memory allocation. This topic, however, is **unrelated** to the data structure in this course!





Binary Heap

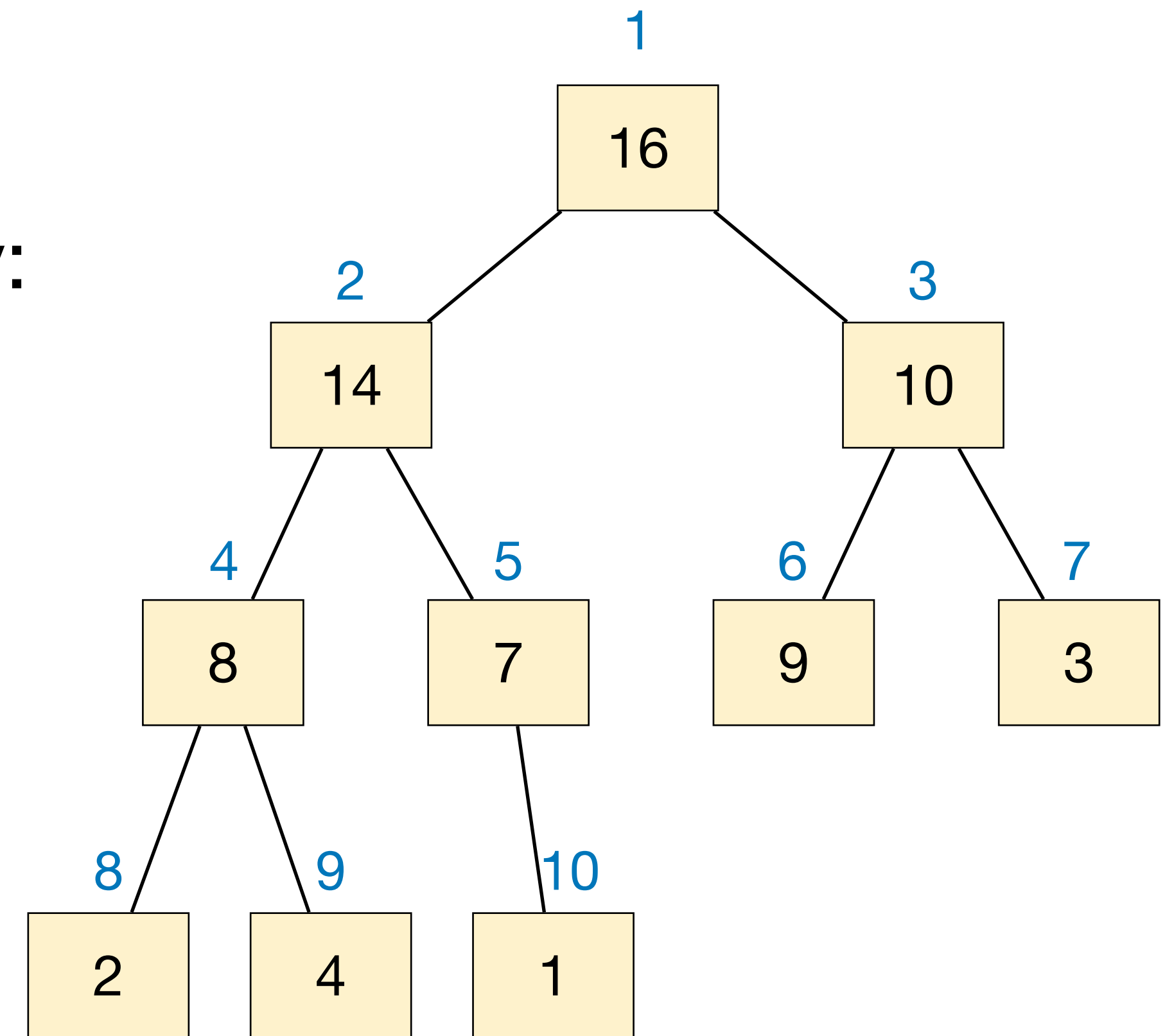
- A binary heap is a **complete binary tree**, in which each node represents an item.
 - ▶ A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- Values in the nodes satisfy **heap-property**.
 - ▶ **Max-heap**: for each node except root, value of that node $\leq$ value of its parent.
 - ▶ **Min-heap**: for each node except root, value of that node $\geq$ value of its parent.





Binary Heap

- We can use an array to represent a binary heap. Obtaining parent and children are easy:
 - Parent of node u : $\lfloor idx_u/2 \rfloor$
 - Left child of u : $2 \cdot idx_u$
 - Right child of u : $2 \cdot idx_u + 1$
 - All in $O(1)$ time!



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



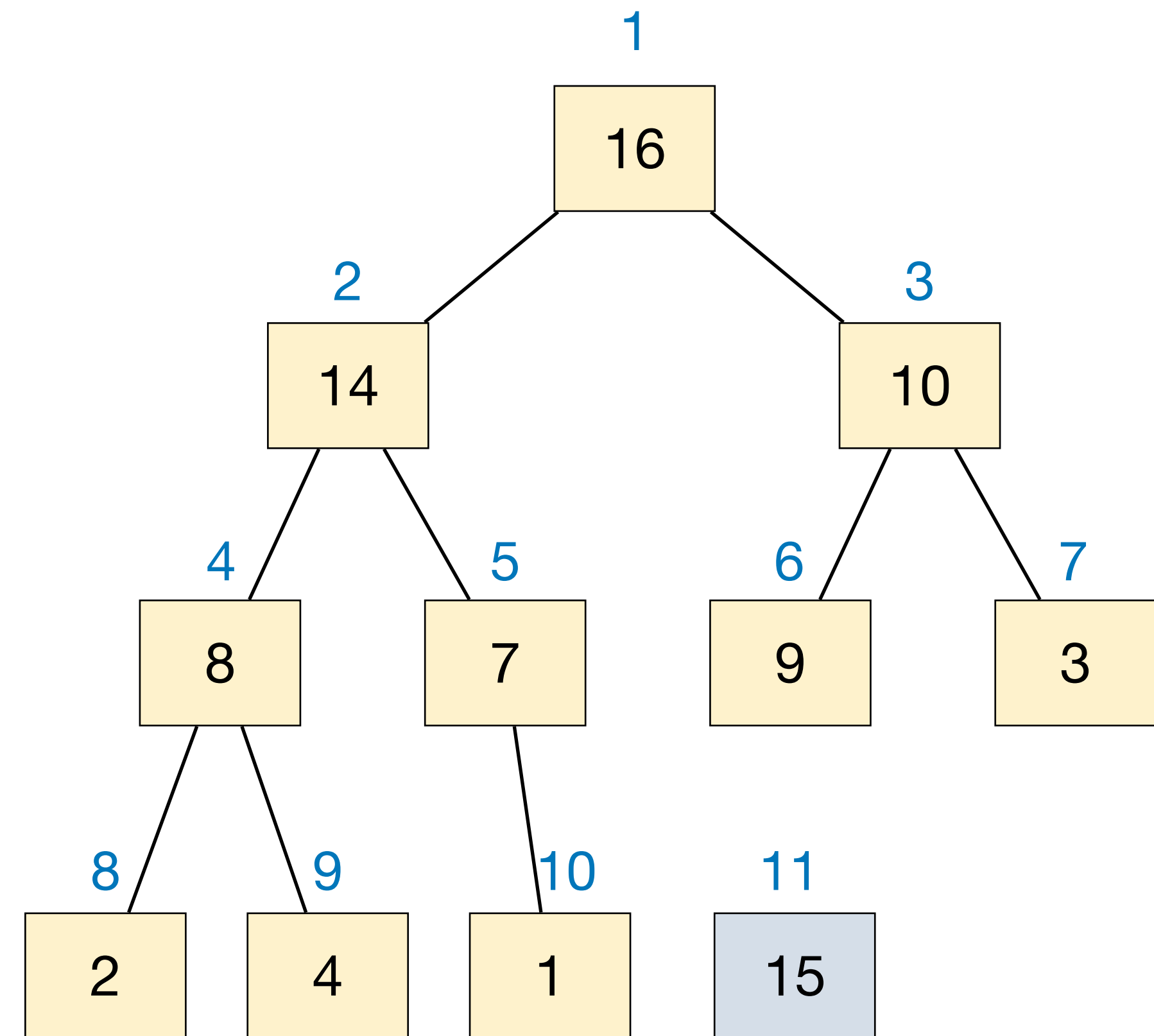
Common operations of Binary Max-Heap

- Consider max-heap as an example. (Min-heap is similar.)
- Most common operations:
 - **HeapInsert**: insert an element into the heap.
 - **HeapGetMax**: return the item with maximum value. Runtime is $O(1)$
 - **HeapExtractMax**: remove the item with maximum value from the heap and return it.
- Other operations (which we'll see later)...



Max-Heap – HeapInsert

- Insert an item into a binary max-heap represented by an array.
 - Simply put the item to the end of the array.

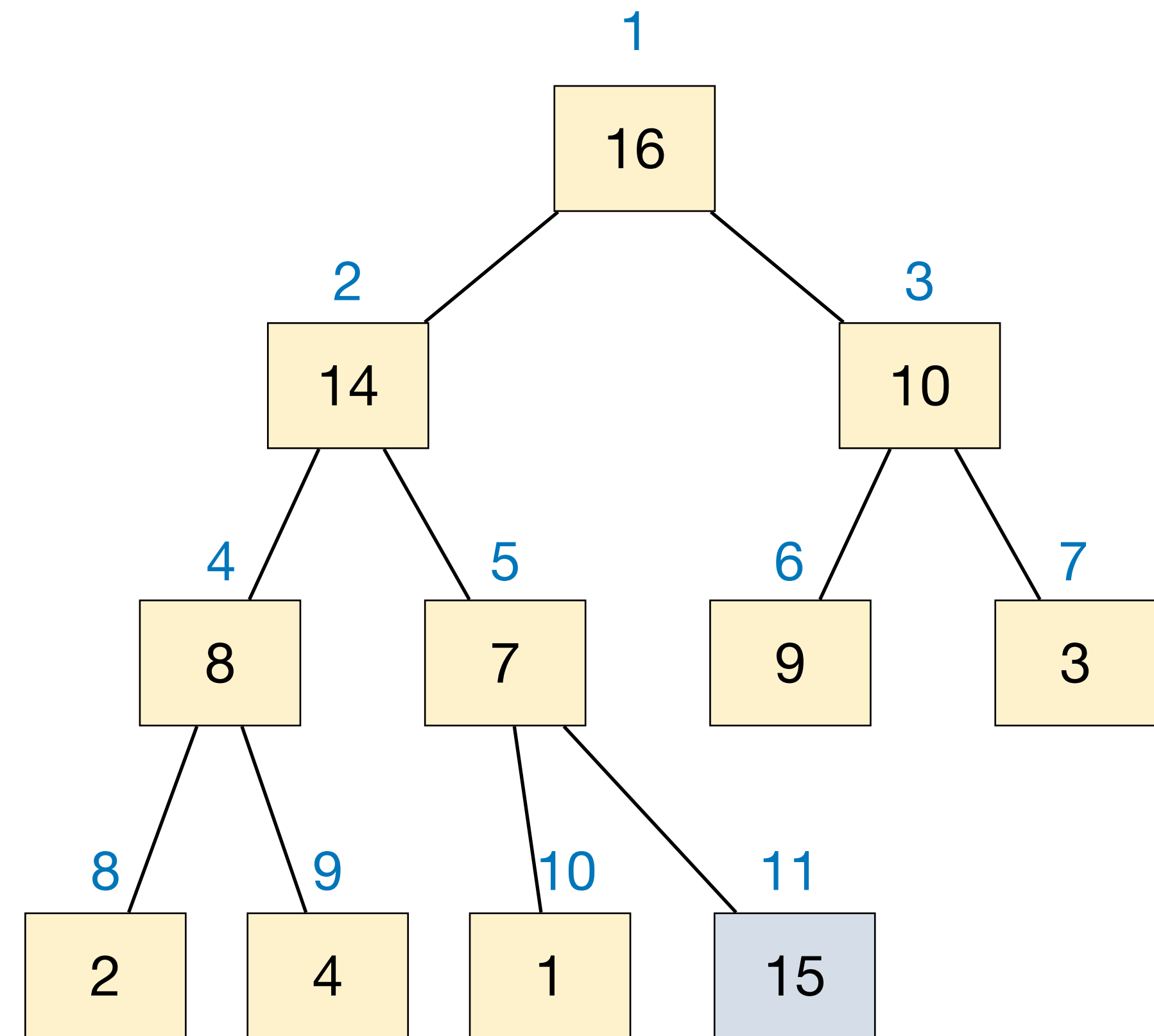


1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15



Max-Heap – HeapInsert

- Insert an item into a binary max-heap represented by an array.
 - Simply put the item to the end of the array.
 - We need to maintain **heap property** after insertion: along the path to root, compare and swap. (Why?)

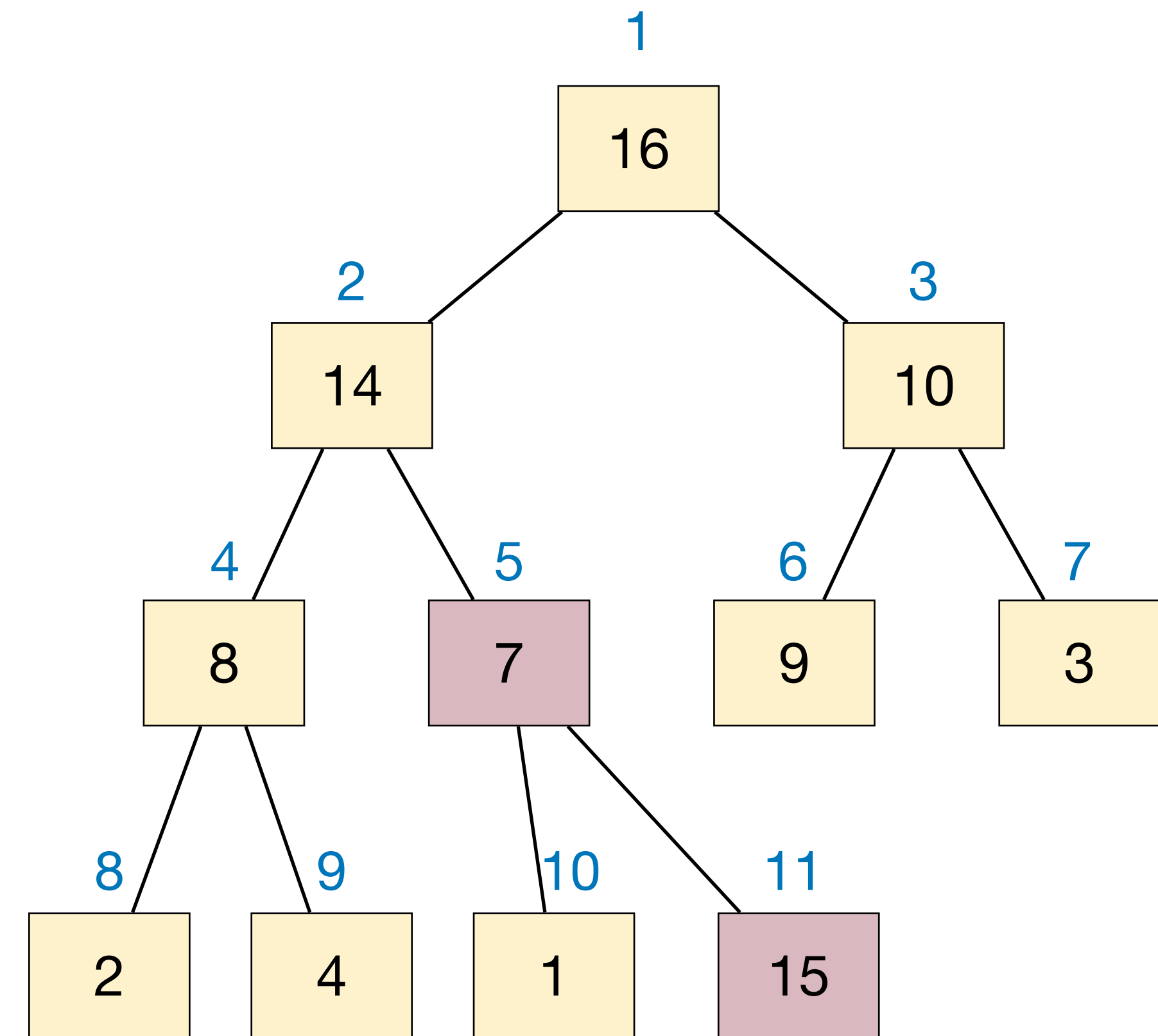


1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15



Max-Heap – HeapInsert

- Insert an item into a binary max-heap represented by an array.
 - Simply put the item to the end of the array.
 - We need to maintain **heap property** after insertion: along the path to root, compare and swap. (Why?)

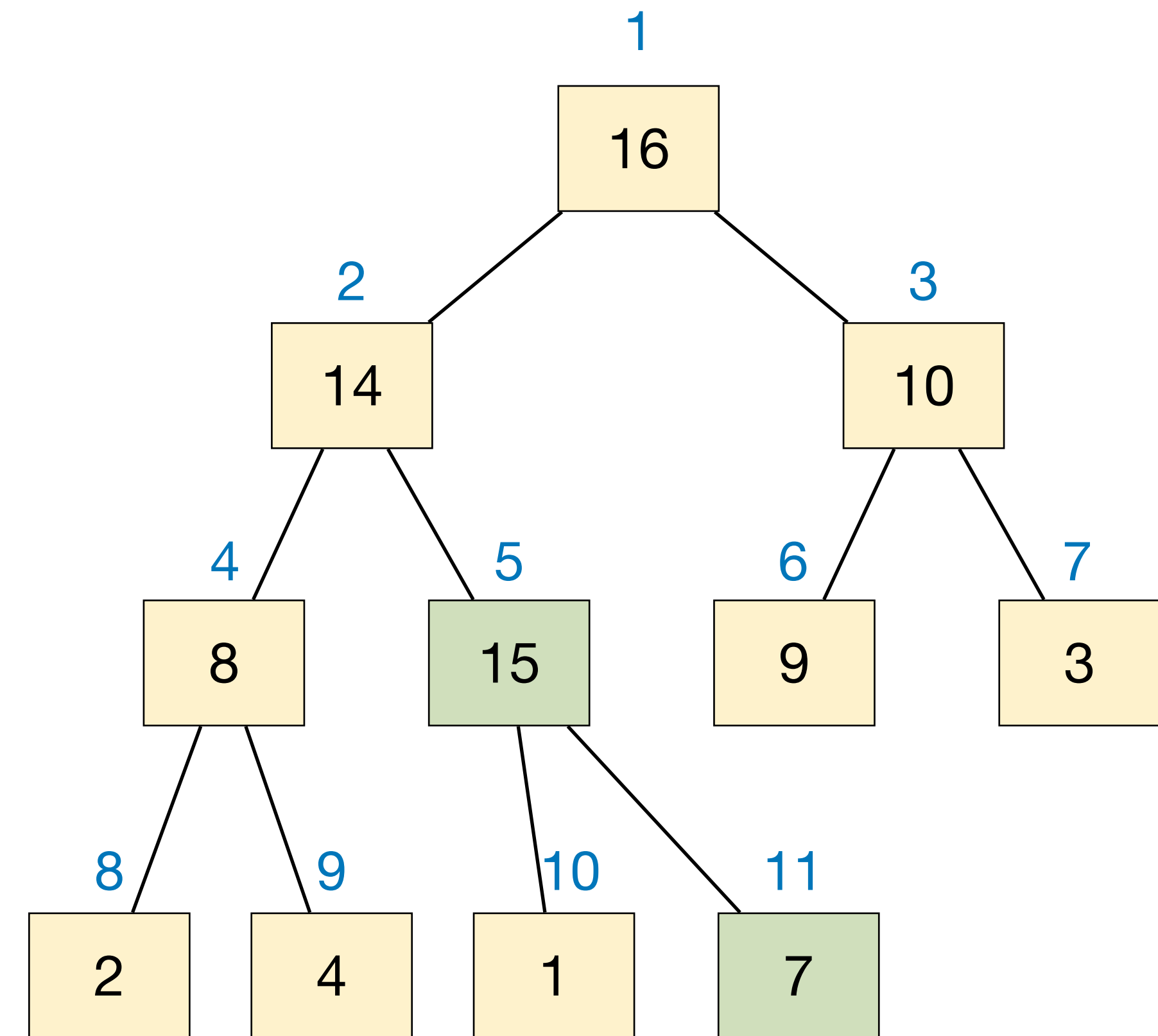


1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	15



Max-Heap – HeapInsert

- Insert an item into a binary max-heap represented by an array.
 - Simply put the item to the end of the array.
 - We need to maintain **heap property** after insertion: along the path to root, compare and swap. (Why?)

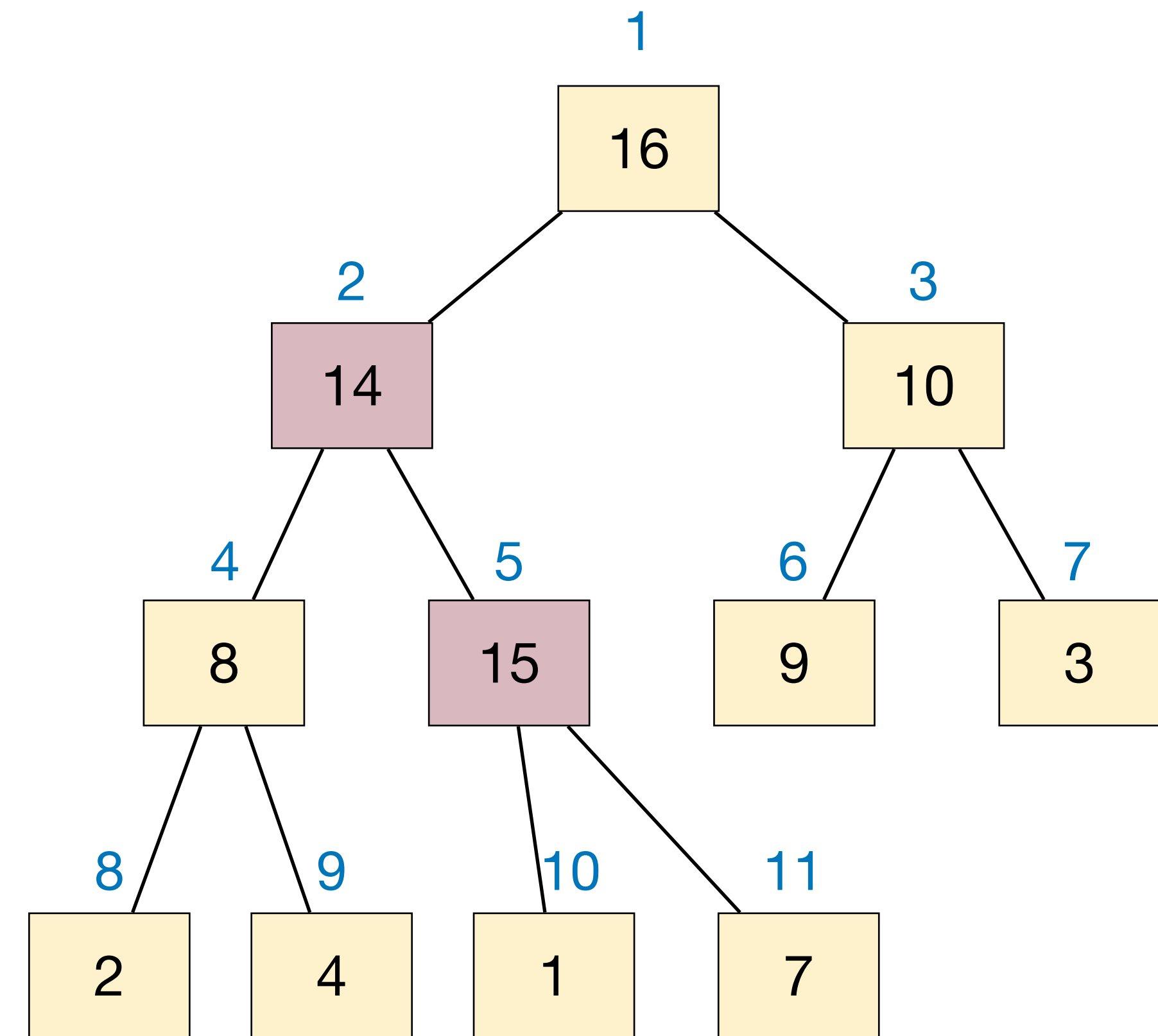


1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	15	9	3	2	4	1	7



Max-Heap – HeapInsert

- Insert an item into a binary max-heap represented by an array.
 - Simply put the item to the end of the array.
 - We need to maintain **heap property** after insertion: along the path to root, compare and swap. (Why?)

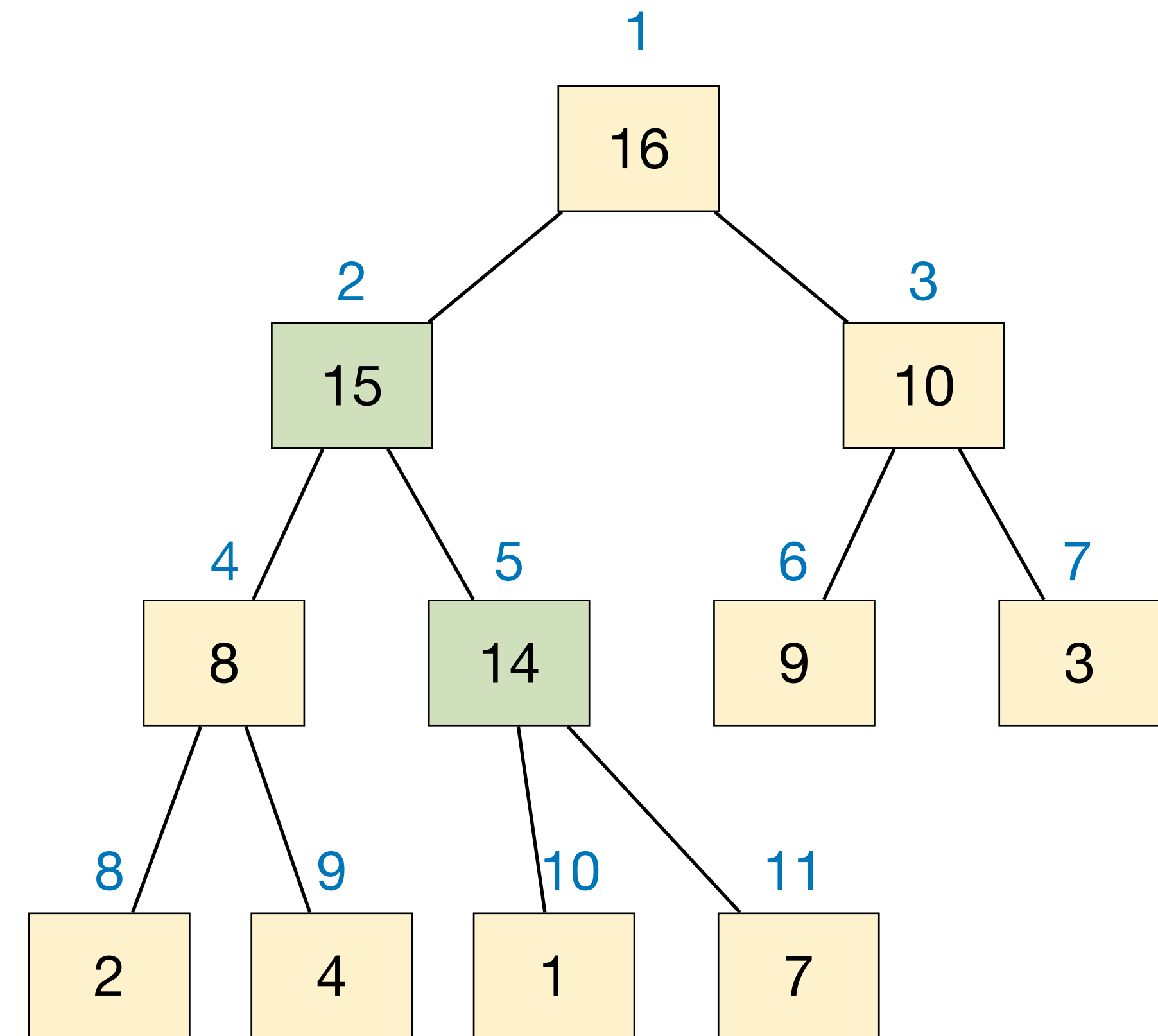


1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	15	9	3	2	4	1	7



Max-Heap – HeapInsert

- Insert an item into a binary max-heap represented by an array.
 - Simply put the item to the end of the array.
 - We need to maintain **heap property** after insertion: along the path to root, compare and swap. (Why?)

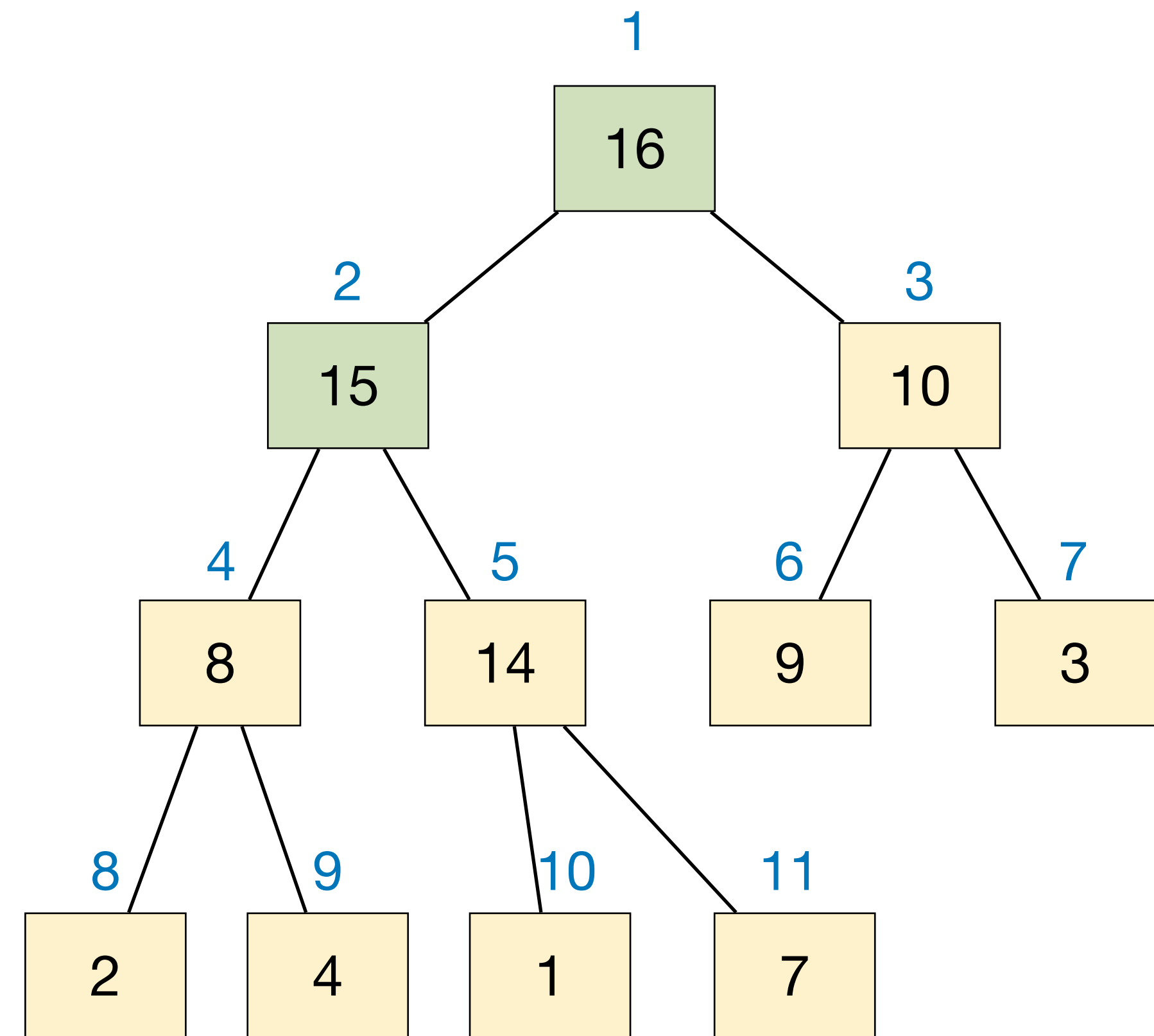


1	2	3	4	5	6	7	8	9	10	11
16	15	10	8	14	9	3	2	4	1	7



Max-Heap – HeapInsert

- Insert an item into a binary max-heap represented by an array.
 - Simply put the item to the end of the array.
 - We need to maintain **heap property** after insertion: along the path to root, compare and swap. (Why?)



1	2	3	4	5	6	7	8	9	10	11
16	15	10	8	14	9	3	2	4	1	7



Max-Heap – HeapInsert

HeapInsert(A, x):

$heap_size \ += 1$

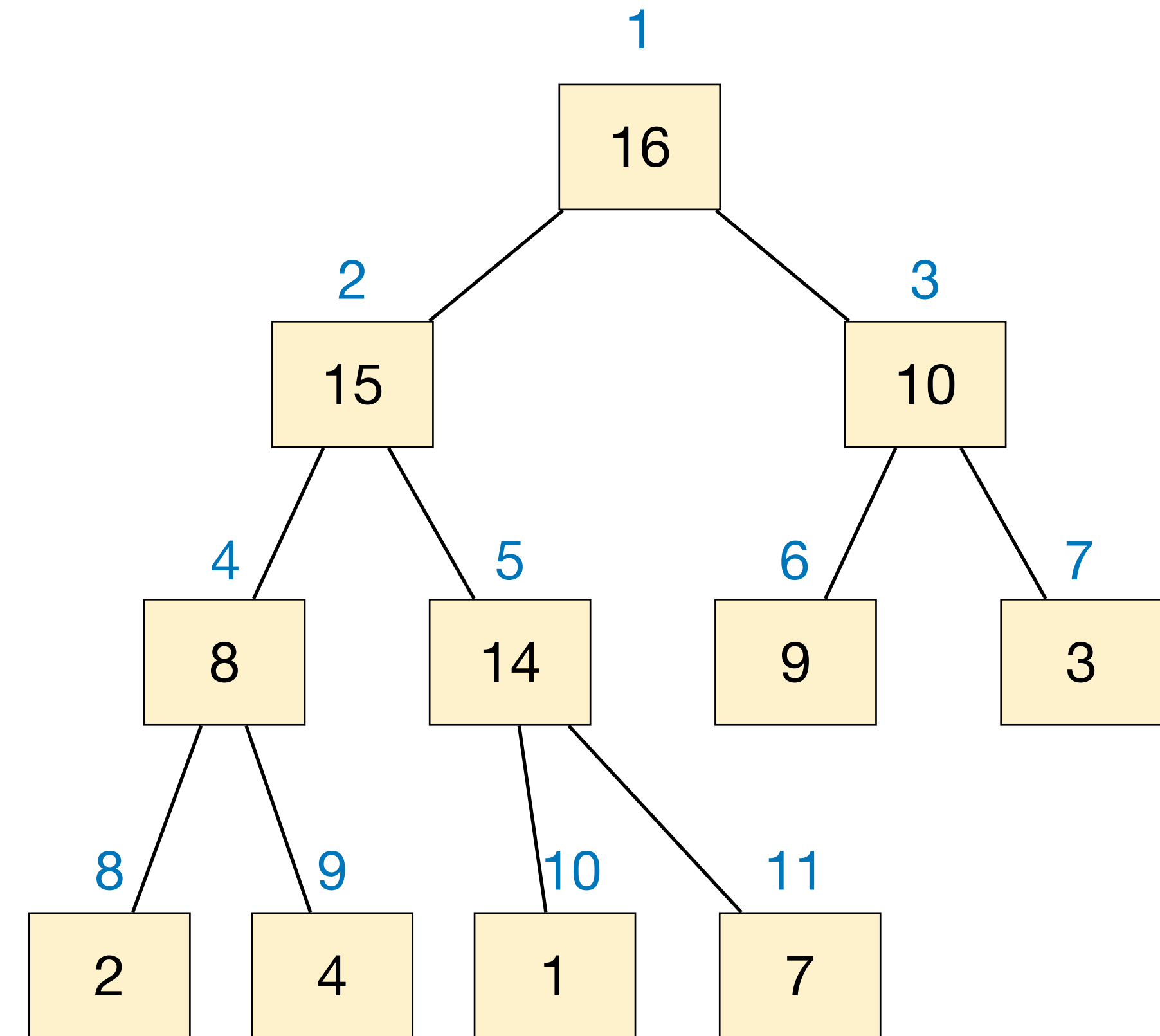
$A[heap_size] \ := x$

$idx \ := heap_size$

while $idx > 1$ **and** $A[Floor(idx / 2)] < A[idx]$

$Swap(A[Floor(idx / 2)], A[idx])$

$idx \ := Floor(idx / 2)$



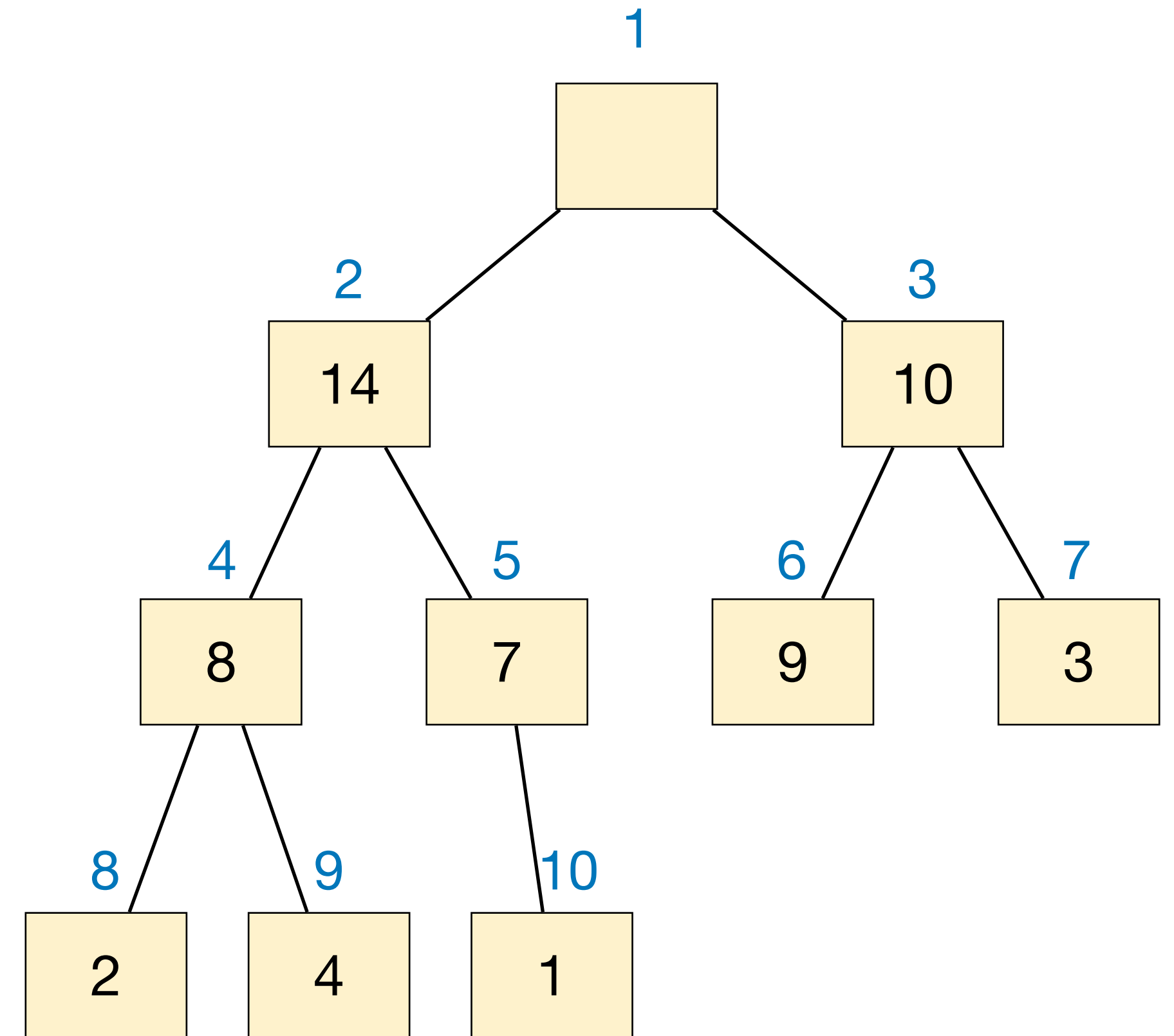
Runtime is $O(\lg n)$

1	2	3	4	5	6	7	8	9	10	11
16	15	10	8	14	9	3	2	4	1	7



Max-Heap — HeapExtractMax

- Remove the maximum item from the heap and return it.
 - Remove and return root is simple, but then what to do?

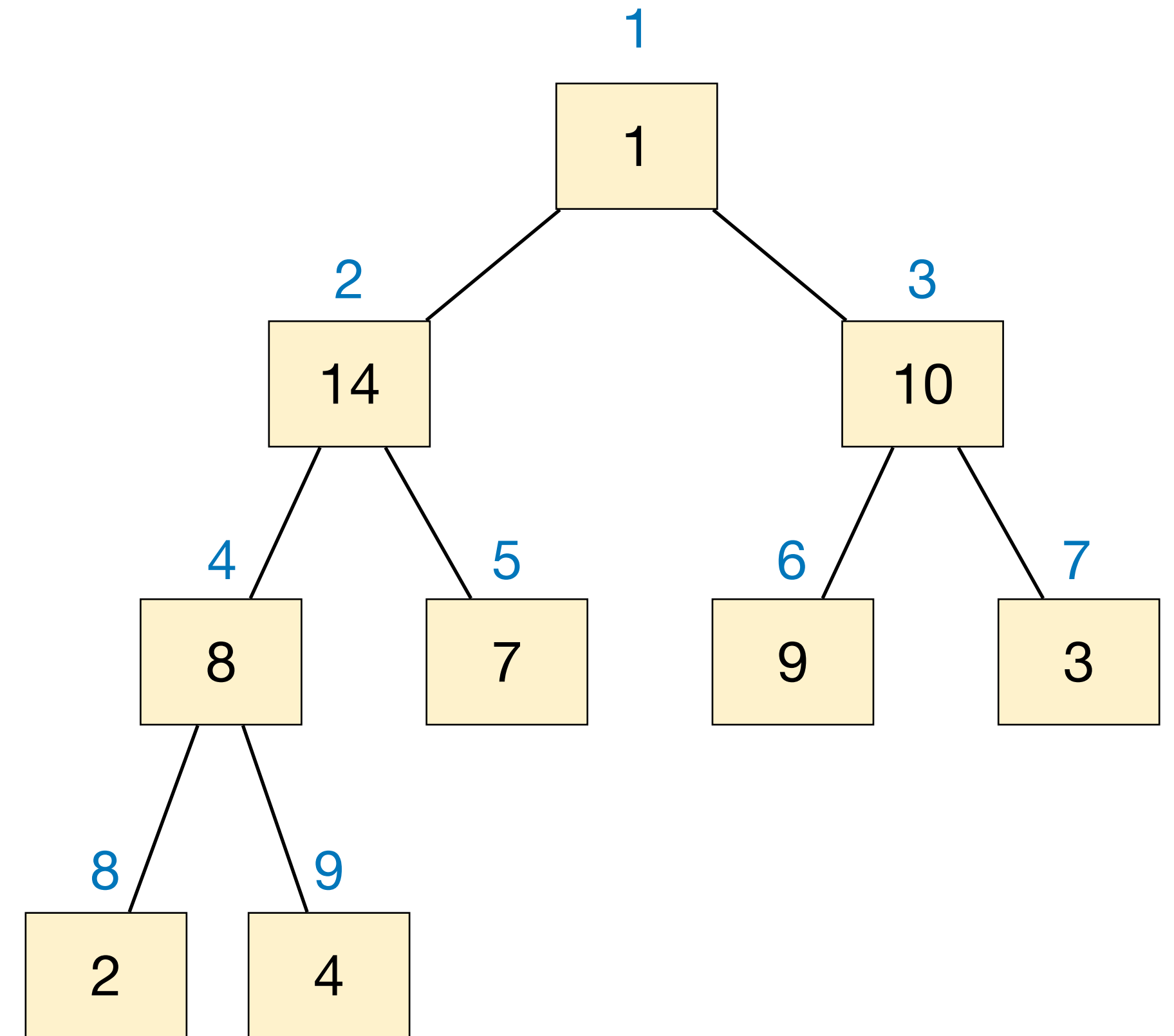


1	2	3	4	5	6	7	8	9	10
	14	10	8	7	9	3	2	4	1



Max-Heap — HeapExtractMax

- Remove the maximum item from the heap and return it.
 - Remove and return root is simple, but then what to do?
 - Move the last item to the root!

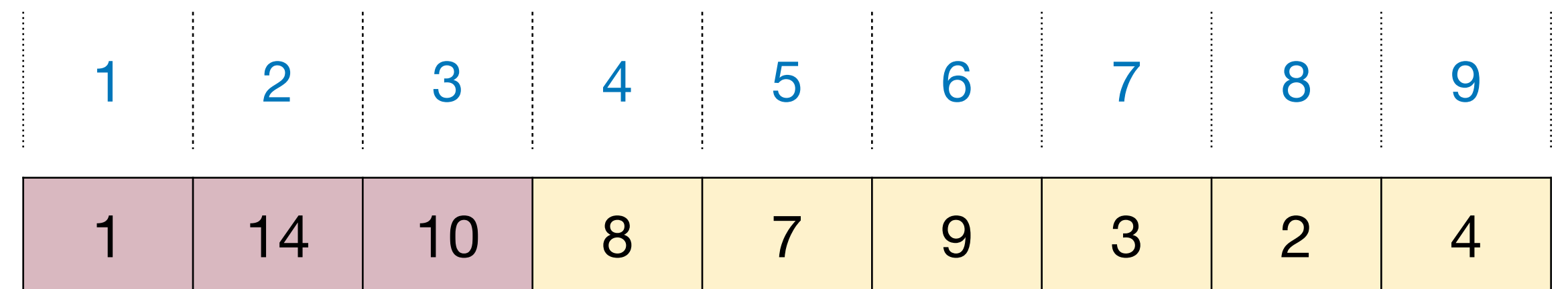
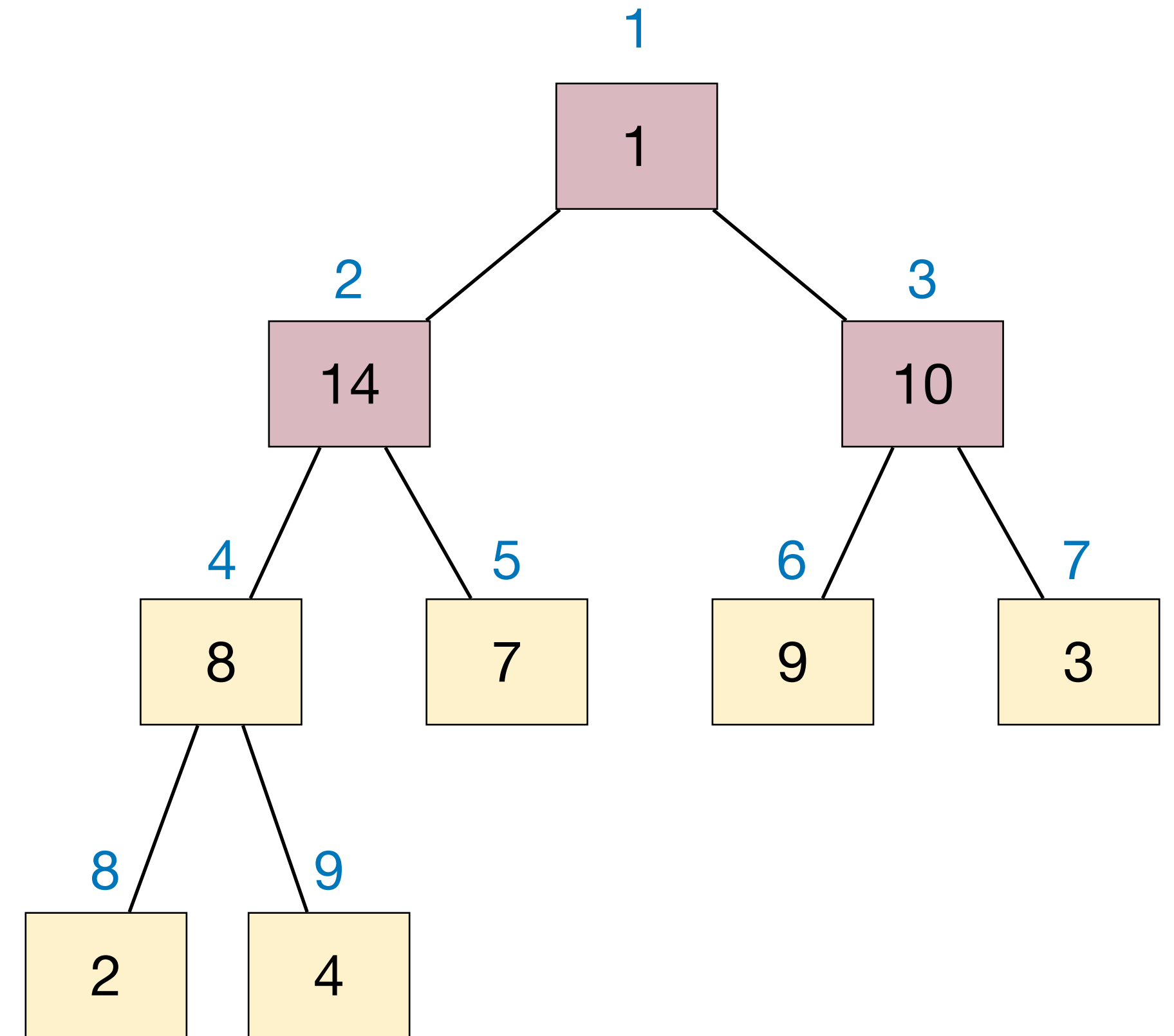


1	2	3	4	5	6	7	8	9
1	14	10	8	7	9	3	2	4



Max-Heap — HeapExtractMax

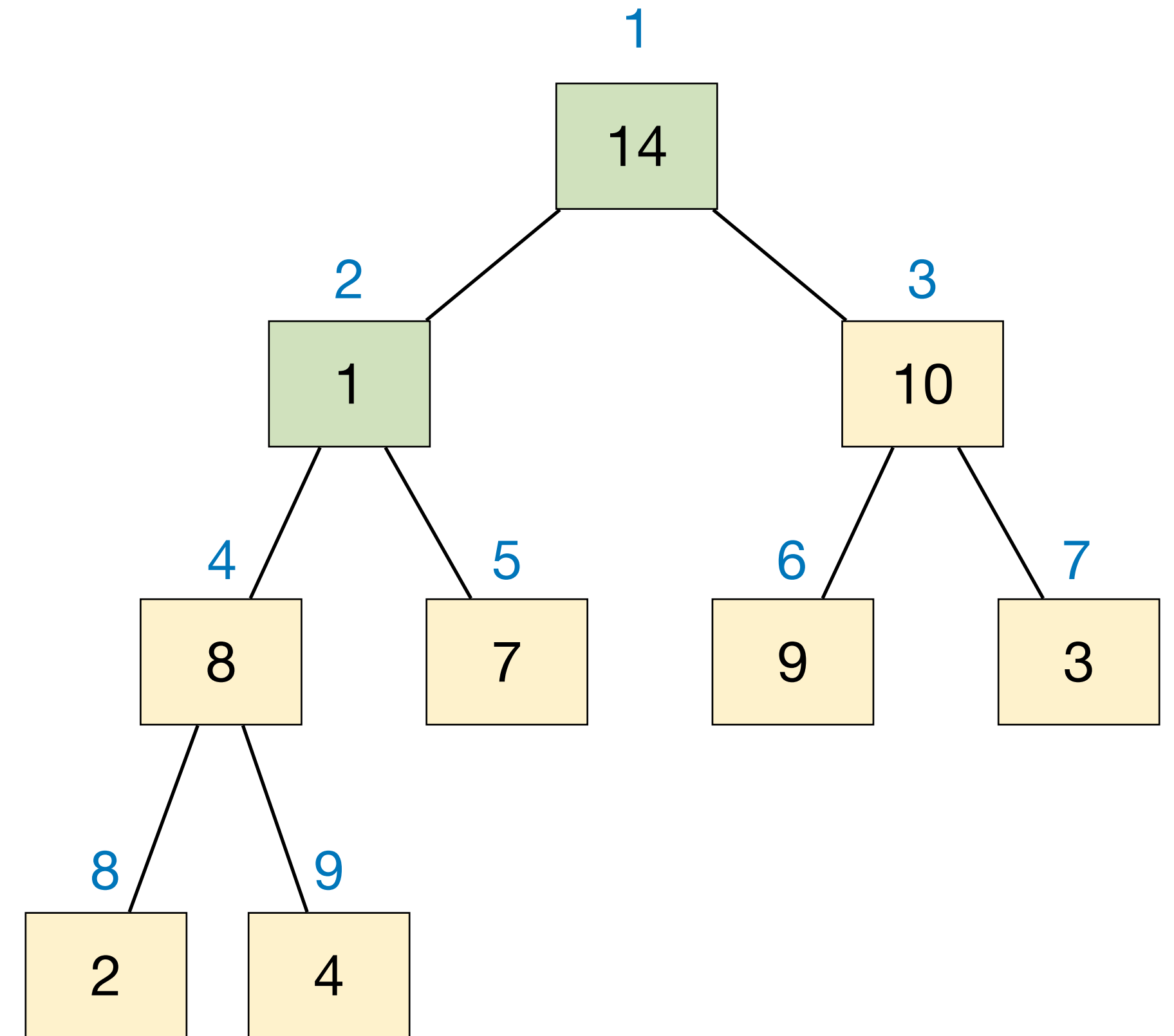
- Remove the maximum item from the heap and return it.
 - Remove and return root is simple, but then what to do?
 - Move the last item to the root!
 - Again, we need to maintain the heap property: **compare** with children, swap with bigger one; do this recursively





Max-Heap — HeapExtractMax

- Remove the maximum item from the heap and return it.
 - Remove and return root is simple, but then what to do?
 - Move the last item to the root!
 - Again, we need to maintain the heap property: **compare** with children, swap with bigger one; do this recursively

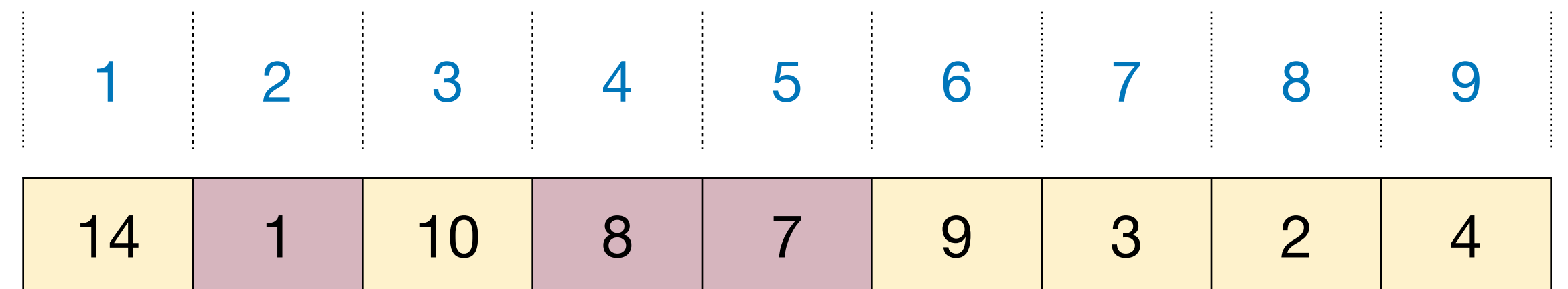
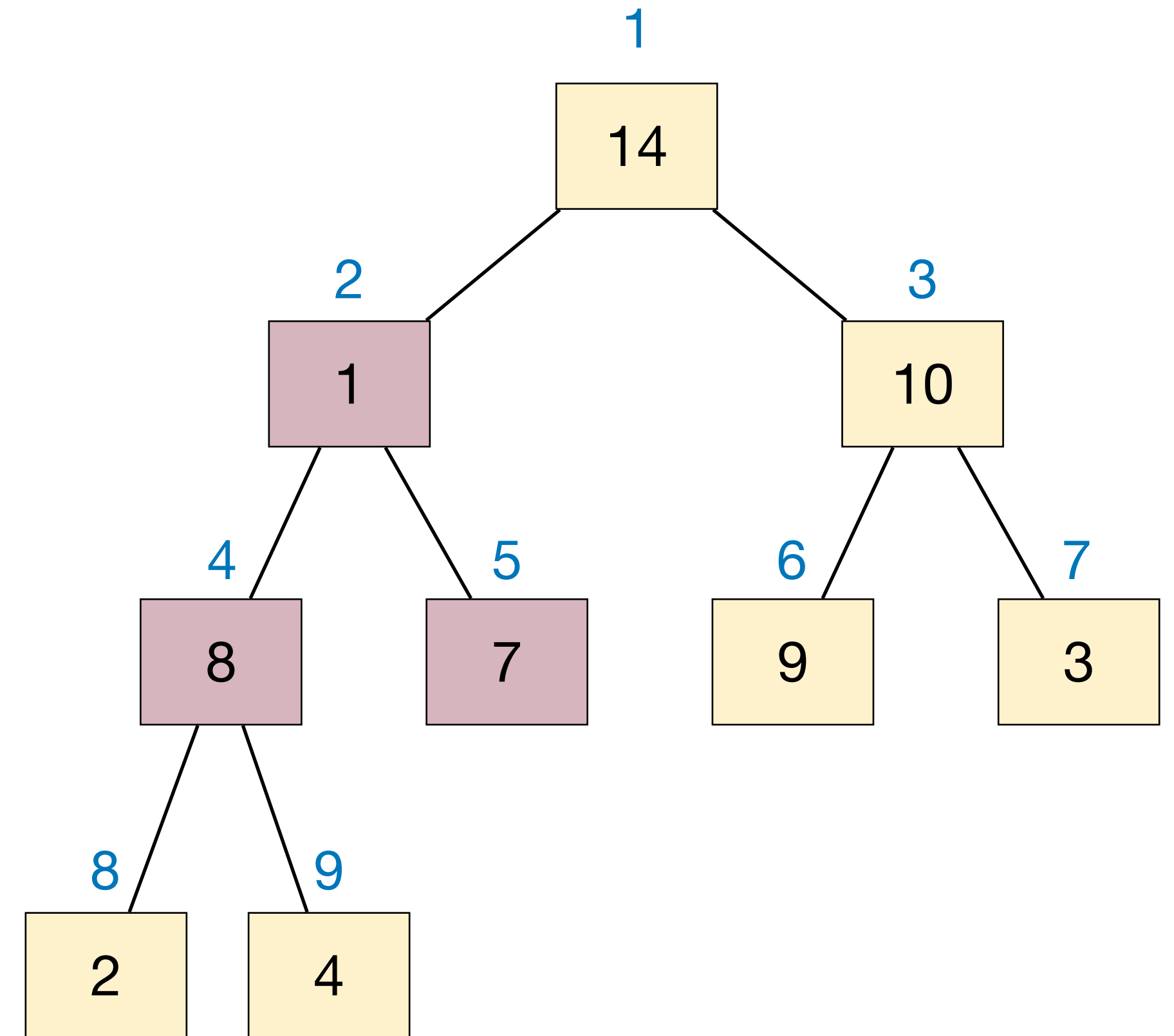


1	2	3	4	5	6	7	8	9
14	1	10	8	7	9	3	2	4



Max-Heap — HeapExtractMax

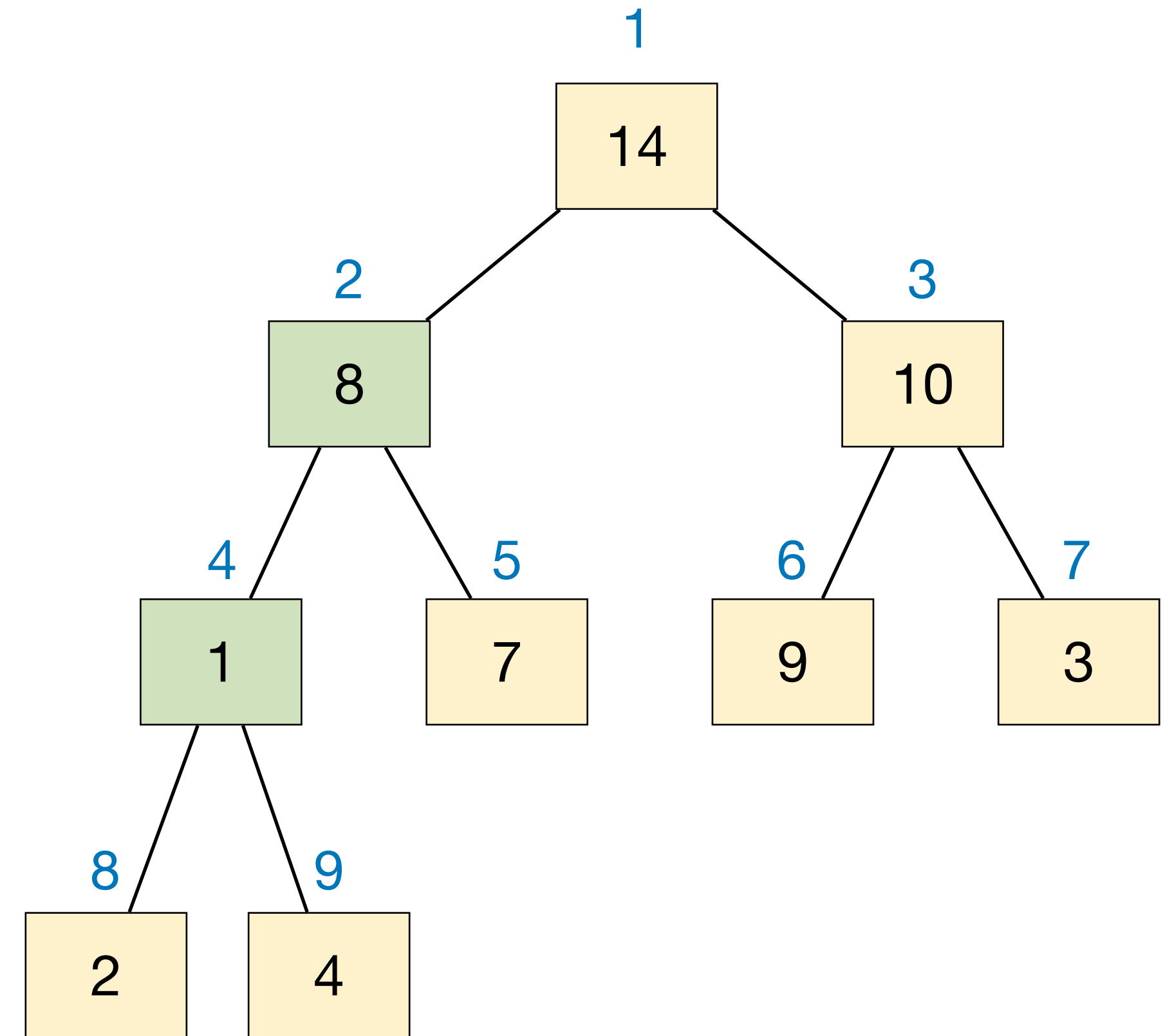
- Remove the maximum item from the heap and return it.
 - Remove and return root is simple, but then what to do?
 - Move the last item to the root!
 - Again, we need to maintain the heap property: **compare** with children, swap with bigger one; do this recursively





Max-Heap — HeapExtractMax

- Remove the maximum item from the heap and return it.
 - Remove and return root is simple, but then what to do?
 - Move the last item to the root!
 - Again, we need to maintain the heap property: **compare** with children, swap with bigger one; do this recursively

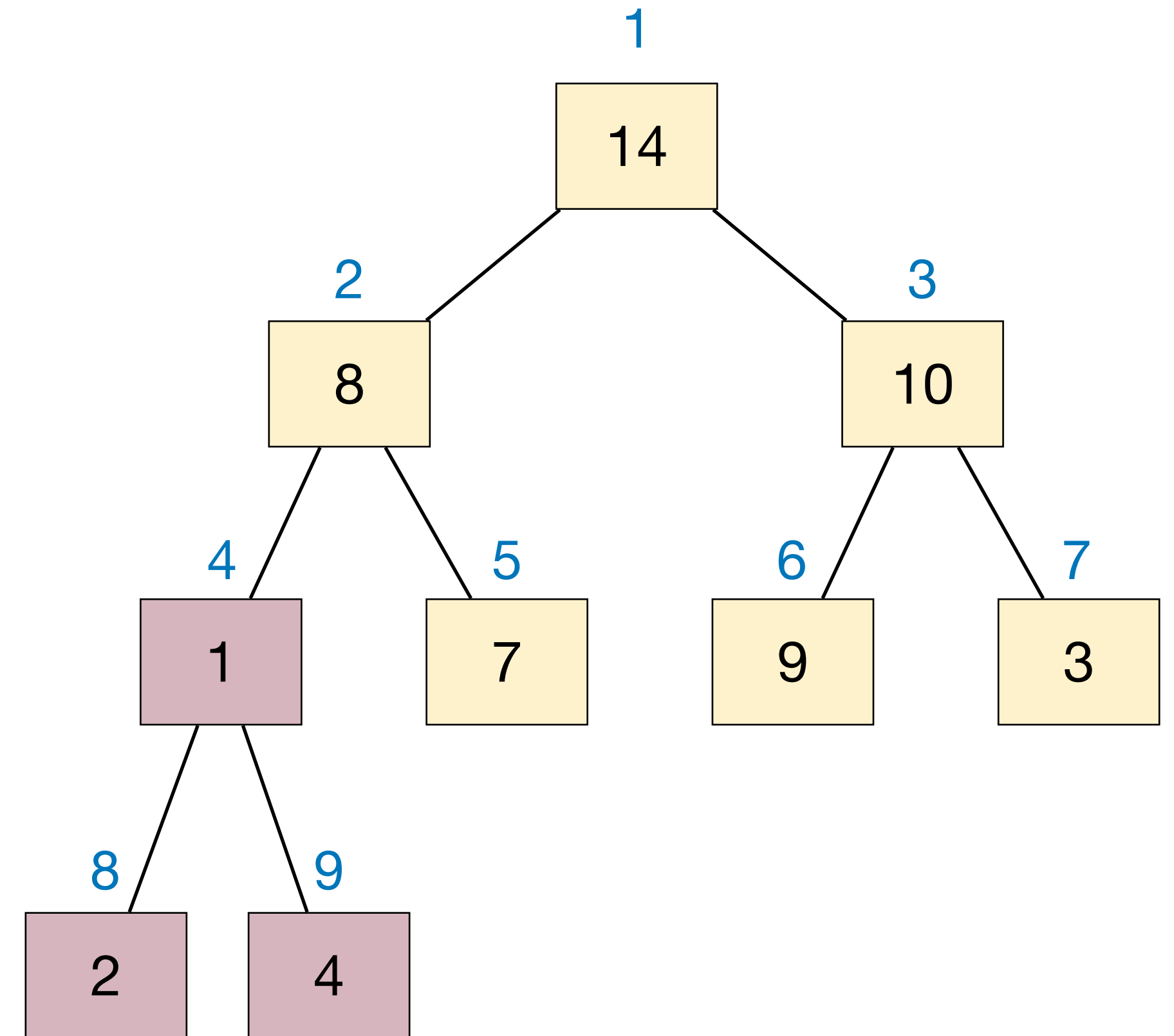


1	2	3	4	5	6	7	8	9
14	8	10	1	7	9	3	2	4



Max-Heap — HeapExtractMax

- Remove the maximum item from the heap and return it.
 - Remove and return root is simple, but then what to do?
 - Move the last item to the root!
 - Again, we need to maintain the heap property: **compare** with children, swap with bigger one; do this recursively

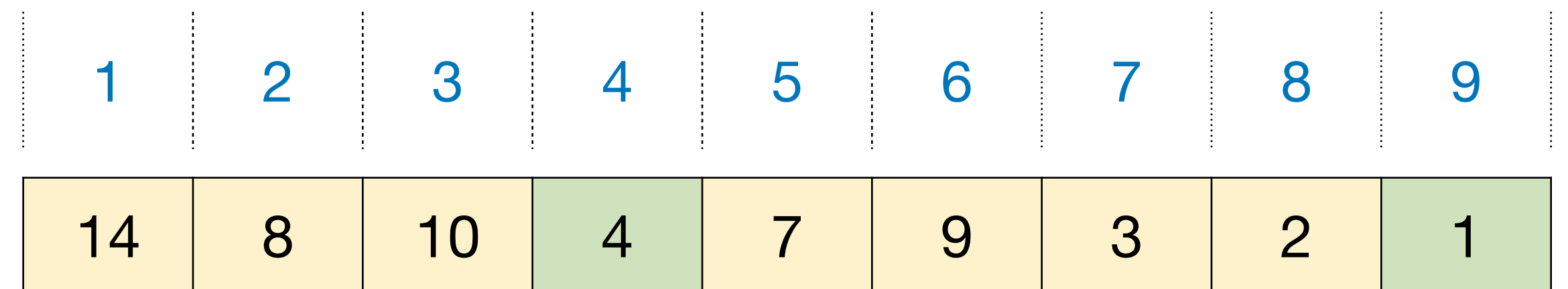
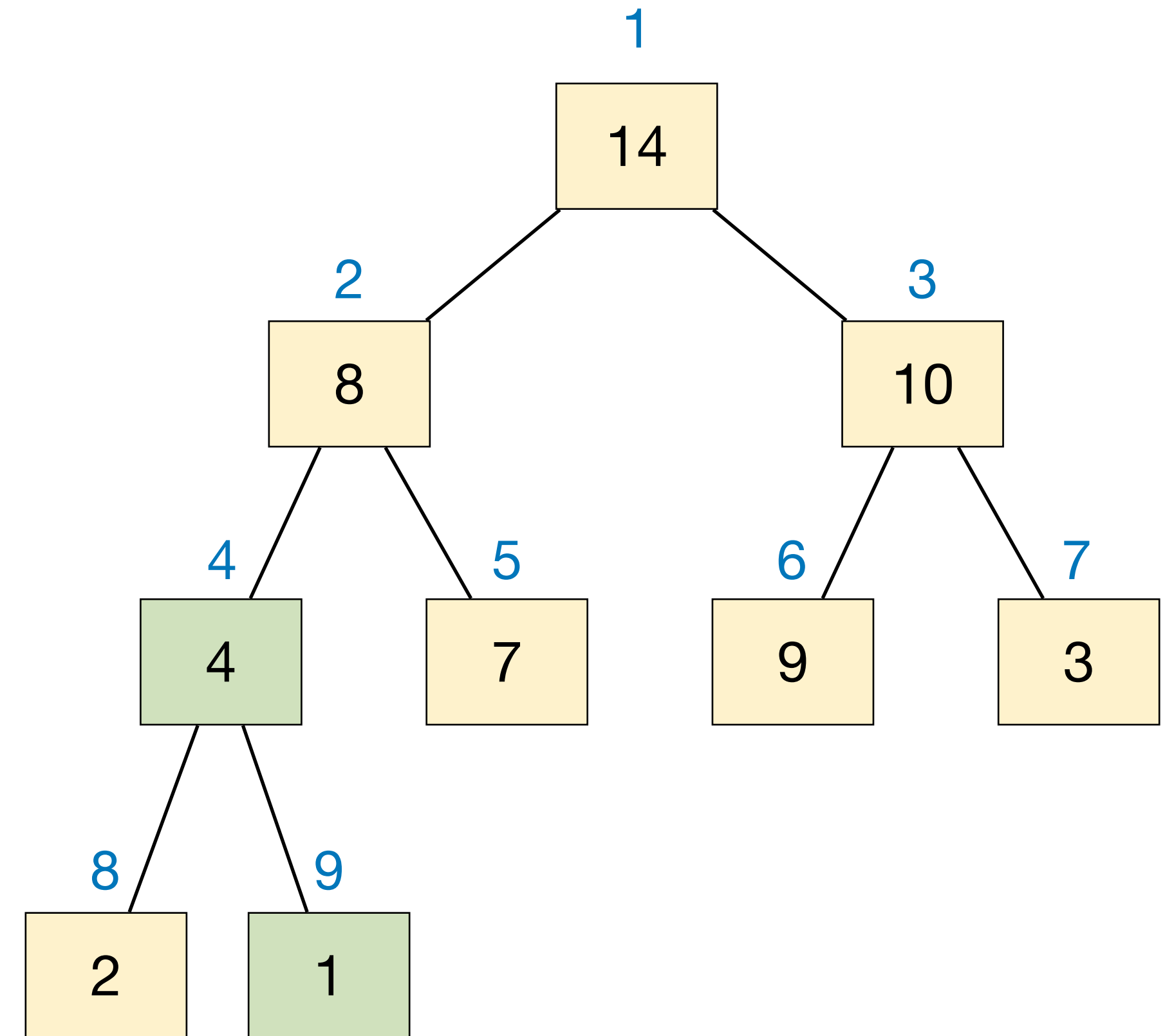


1	2	3	4	5	6	7	8	9
14	8	10	1	7	9	3	2	4



Max-Heap — HeapExtractMax

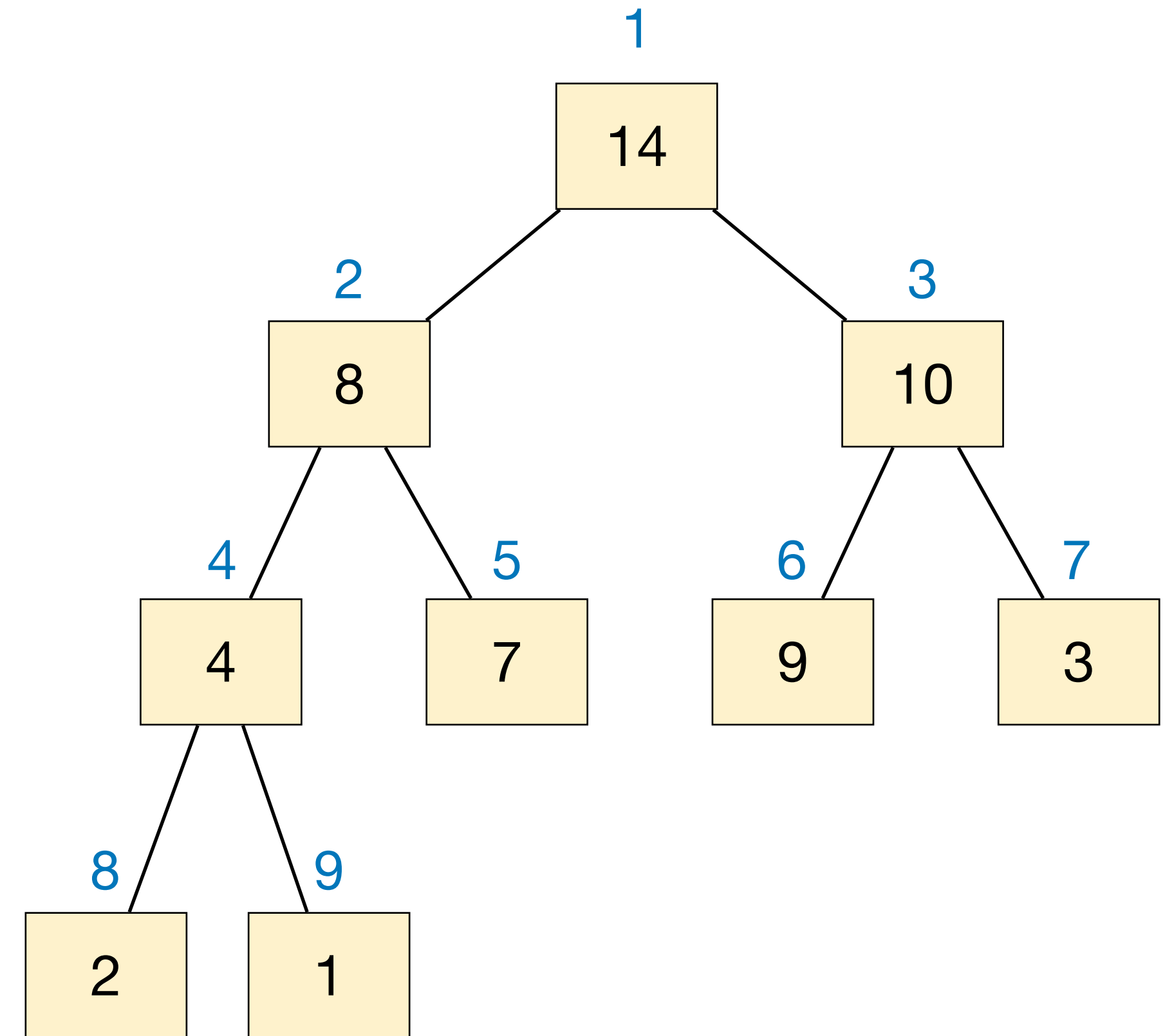
- Remove the maximum item from the heap and return it.
 - Remove and return root is simple, but then what to do?
 - Move the last item to the root!
 - Again, we need to maintain the heap property: **compare** with children, swap with bigger one; do this recursively





Max-Heap — HeapExtractMax

- Remove the maximum item from the heap and return it.
 - Remove and return root is simple, but then what to do?
 - Move the last item to the root!
 - Again, we need to maintain the heap property: **compare** with children, swap with bigger one; do this recursively



1	2	3	4	5	6	7	8	9
14	8	10	4	7	9	3	2	1



Max-Heap — HeapExtractMax

HeapExtractMax(A):

```
max_item := A[1]  
A[1] = A[heap_size--]  
MaxHeapify(1, A)  
return max_item
```

MaxHeapify(*idx*, *A*):

```
idx_l := 2*idx, idx_r := 2*idx + 1  
idx_max := ( idx_l <= heap_size and A[idx_l] > A[idx] ) ? idx_l : idx  
idx_max := ( idx_r <= heap_size and A[idx_r] > A[idx_max] ) ? idx_r : idx_max  
if idx_max != idx  
    Swap (A[idx_max], A[idx])  
    MaxHeapify(idx_max, A)
```

Runtime is $O(\lg n)$



Application of heaps: Priority Queue



Priority Queue

- Recall the `Queue` ADT represents a collection of items to which we can **add** items and **remove** the next item.
 - `Add(item)`: add item to the queue.
 - `Remove()`: remove the next item y from queue, return y .
- The **queuing discipline** decides which item to be removed.
 - First-in-first-out queue (FIFO Queue)
 - Last-in-first-out queue (LIFO Queue, Stack)
 - **Priority queue**: each item associated with a **priority**, **Remove** always deletes the item with max (or min) priority.



Priority Queue

- Use binary heap to implement priority queue
 - `Add(item): HeapInsert(item)`
 - `Remove(): HeapExtractMax()`
 - Other operations: `GetMax()`, `UpdatePriority(item, val)`
 - All these operations finish within $O(\lg n)$ time
- Application of priority queues
 - Scheduling, Event simulation, ...
 - Used in more sophisticated algorithms (will see them later...)



HeapSort

Take an array and make it a max-heap.

HeapSort(I):

$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

In each iteration:

Place **one** item in the array to its final position.

This **one** is the max item in current heap.

That is, place i^{th} biggest item to position $n - i + 1$.

1. Keep a **copy** of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the **copy** of the root item

The loop invariant:

The largest i elements are already in their correct positions.



HeapSort

HeapSort(I):

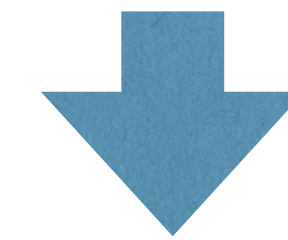
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

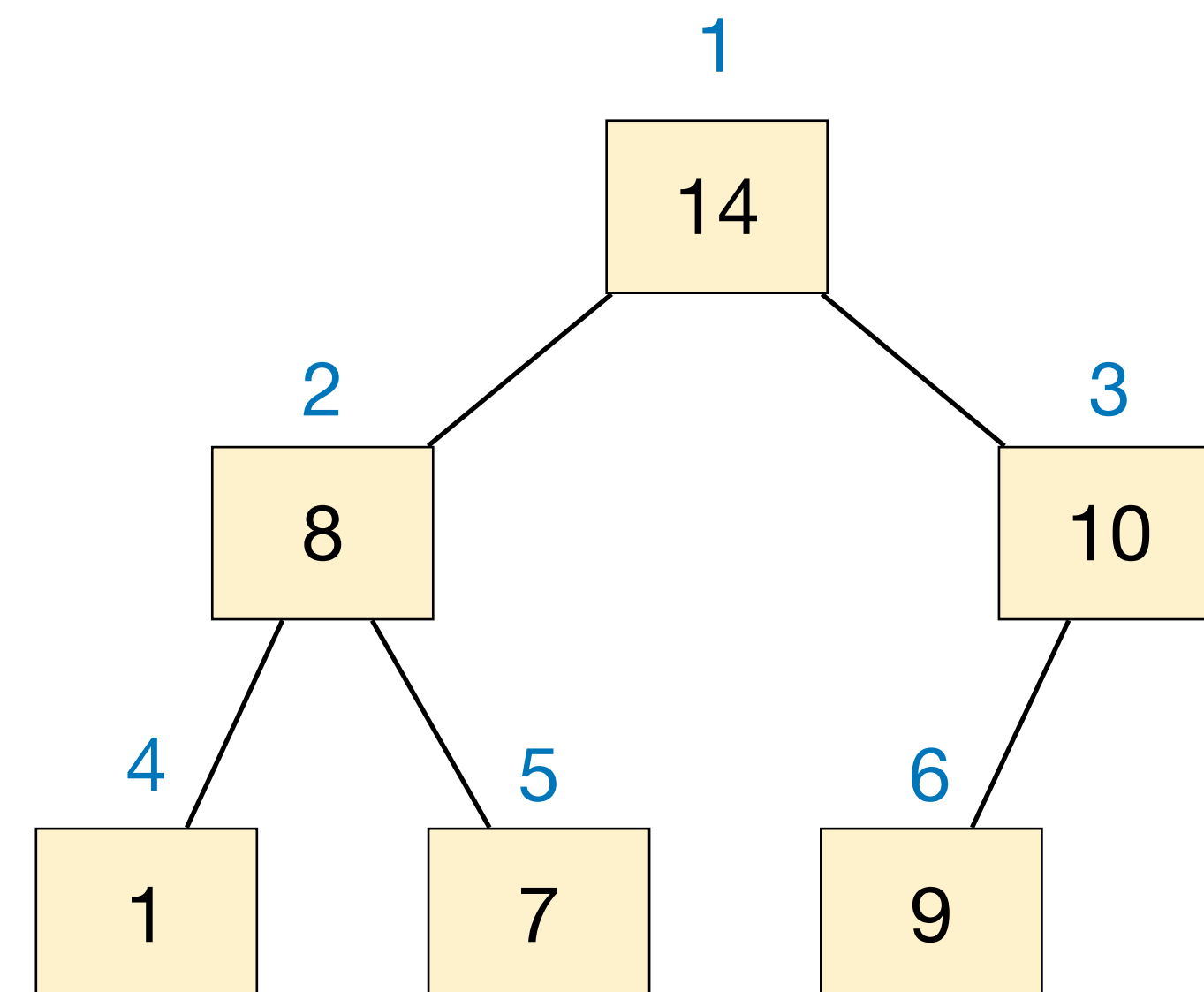
$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
8	1	9	14	7	10



14	8	10	1	7	9
----	---	----	---	---	---





HeapSort

HeapSort(I):

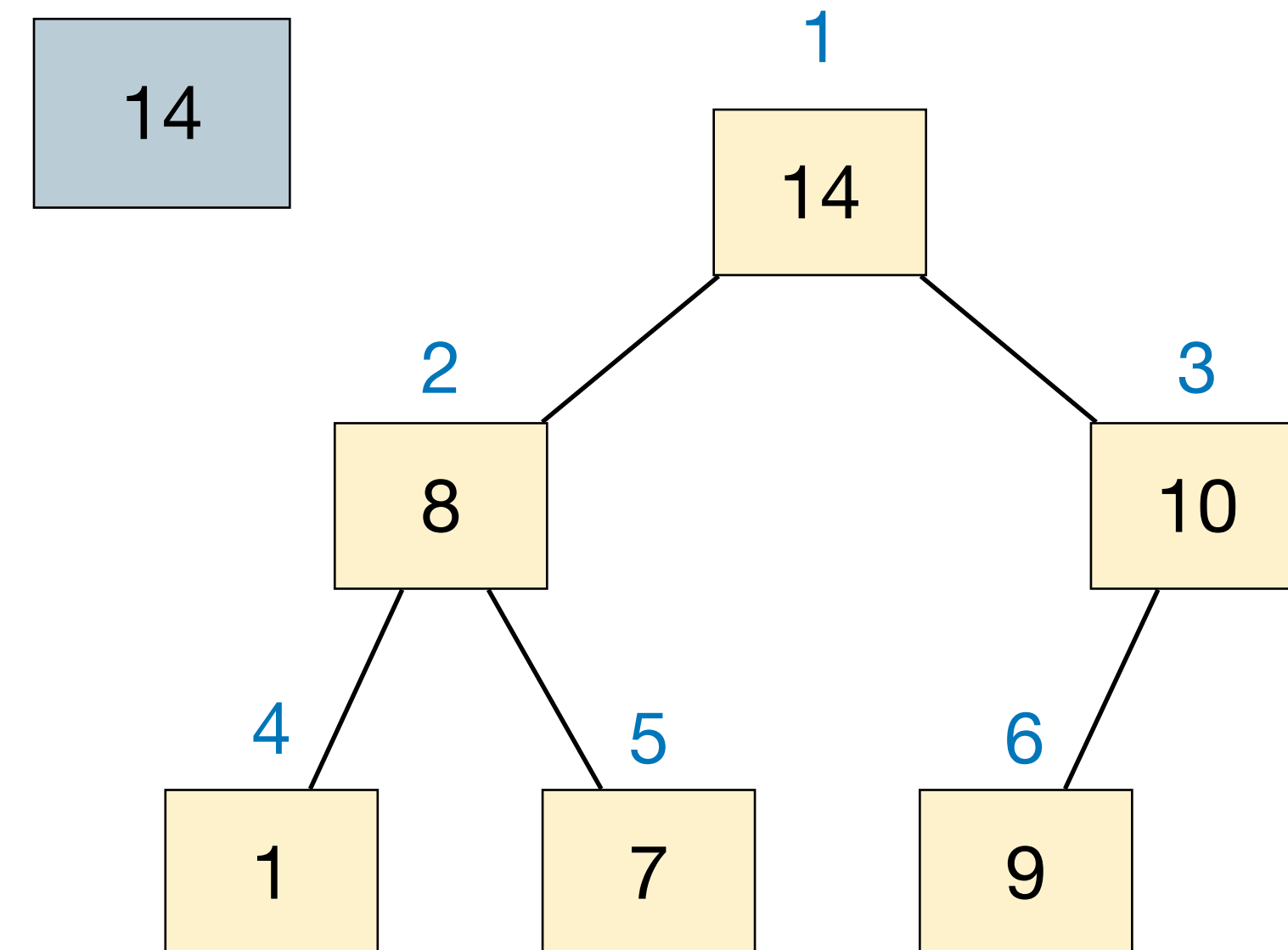
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
14	8	10	1	7	9



$i = 6$



HeapSort

HeapSort(I):

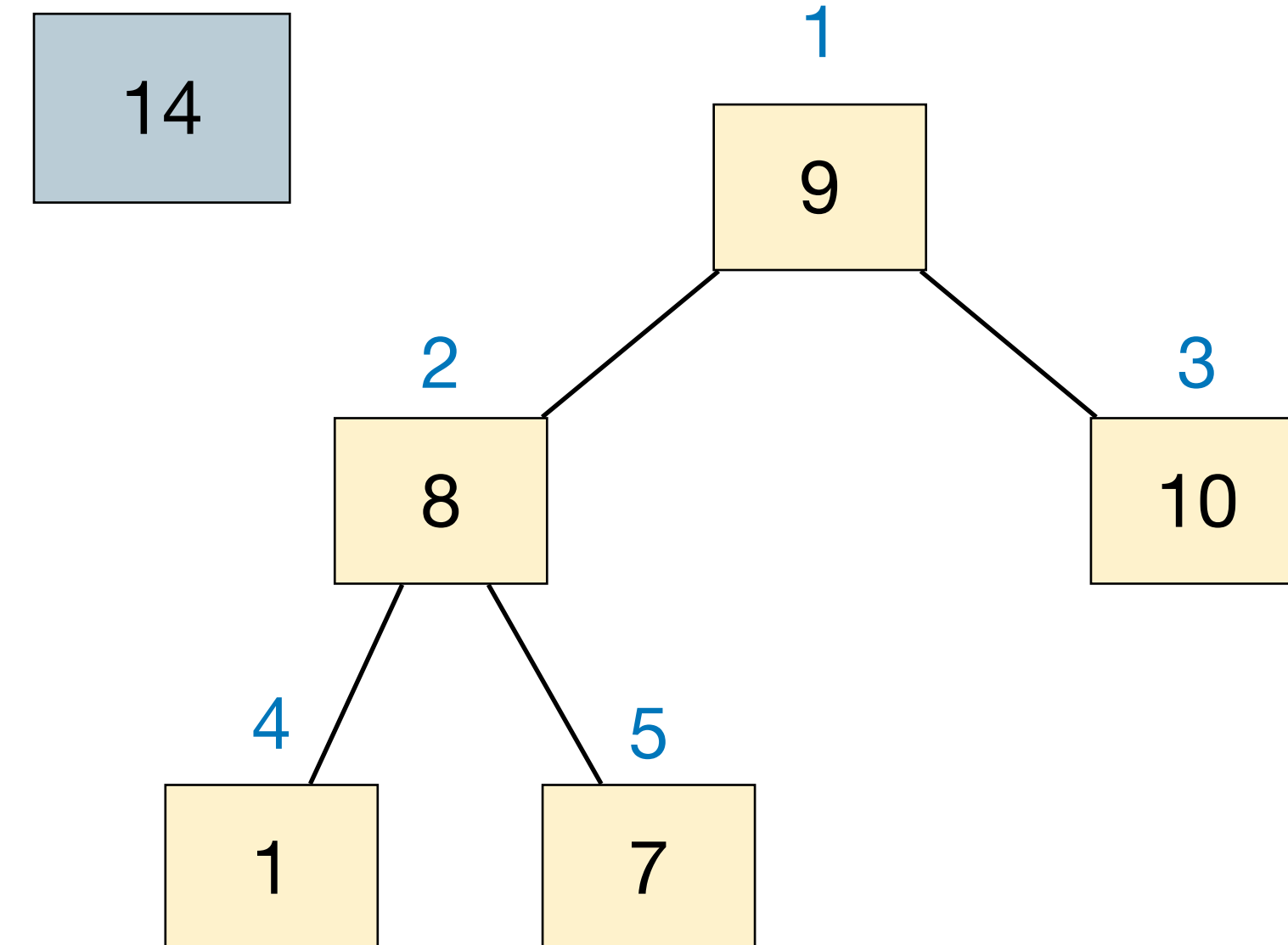
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
9	8	10	1	7	9



$i = 6$



HeapSort

HeapSort(I):

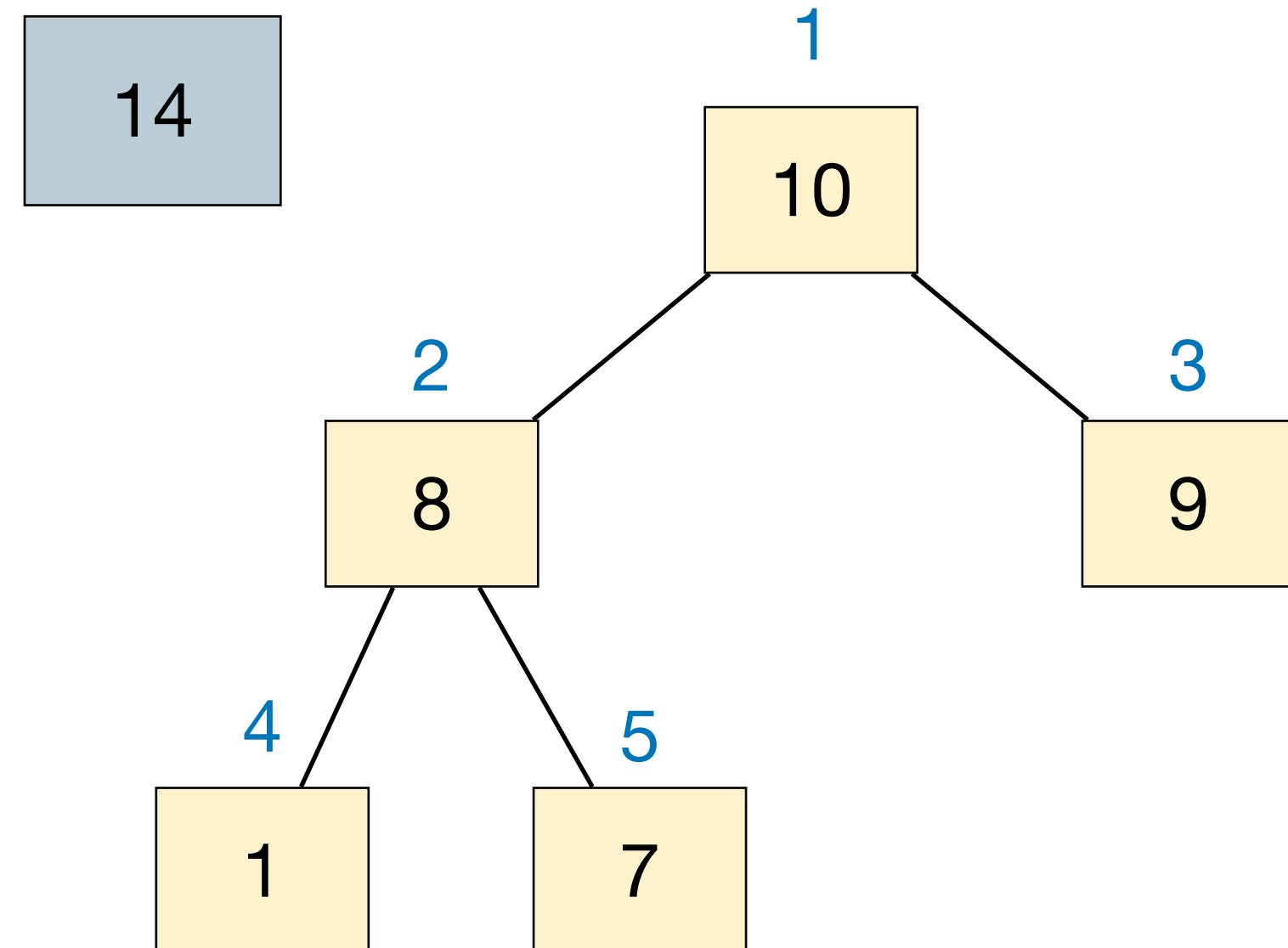
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
10	8	9	1	7	9



$i = 6$



HeapSort

HeapSort(I):

$heap := BuildMaxHeap(I)$

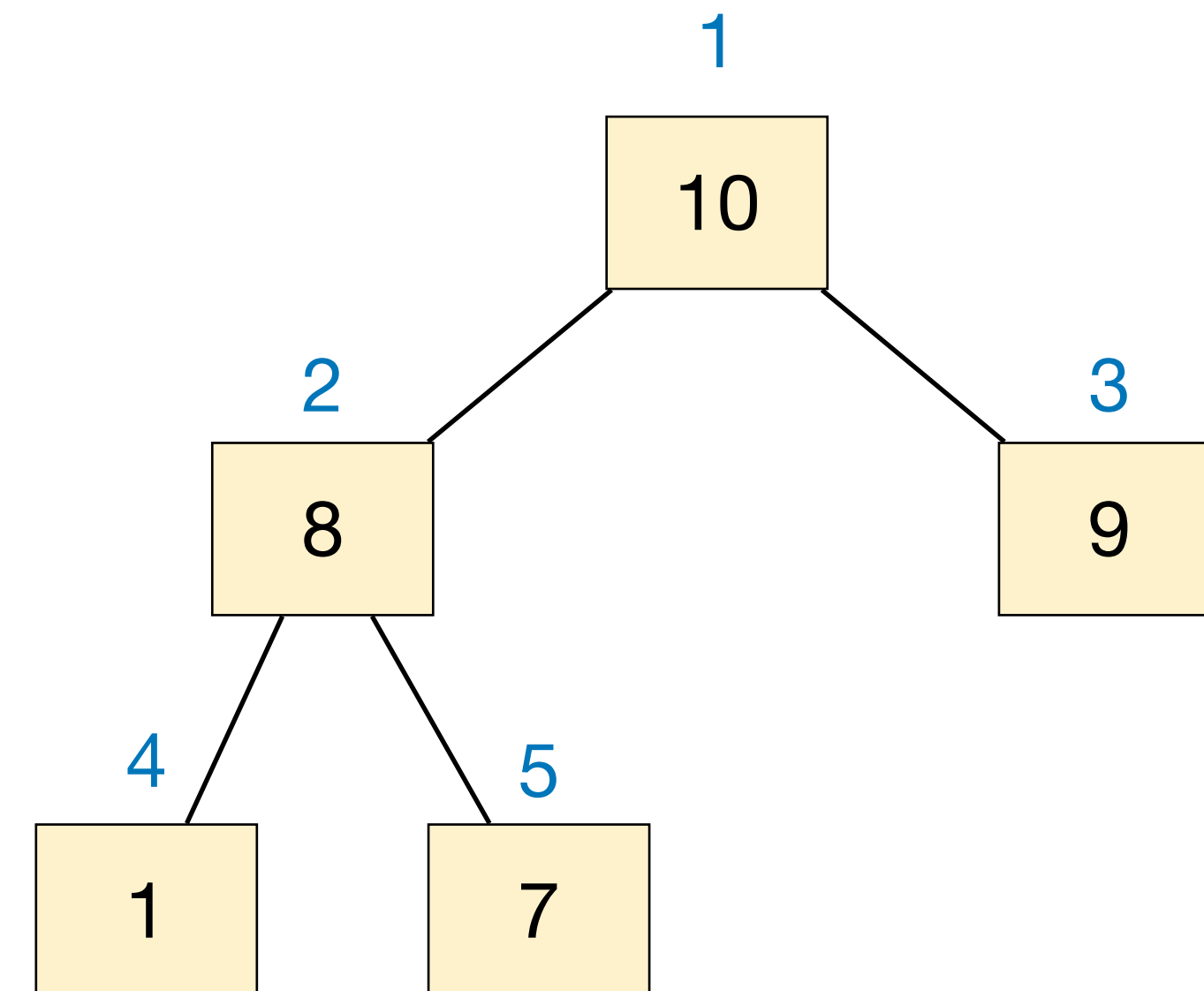
for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
10	8	9	1	7	14

$i = 6$





HeapSort

HeapSort(I):

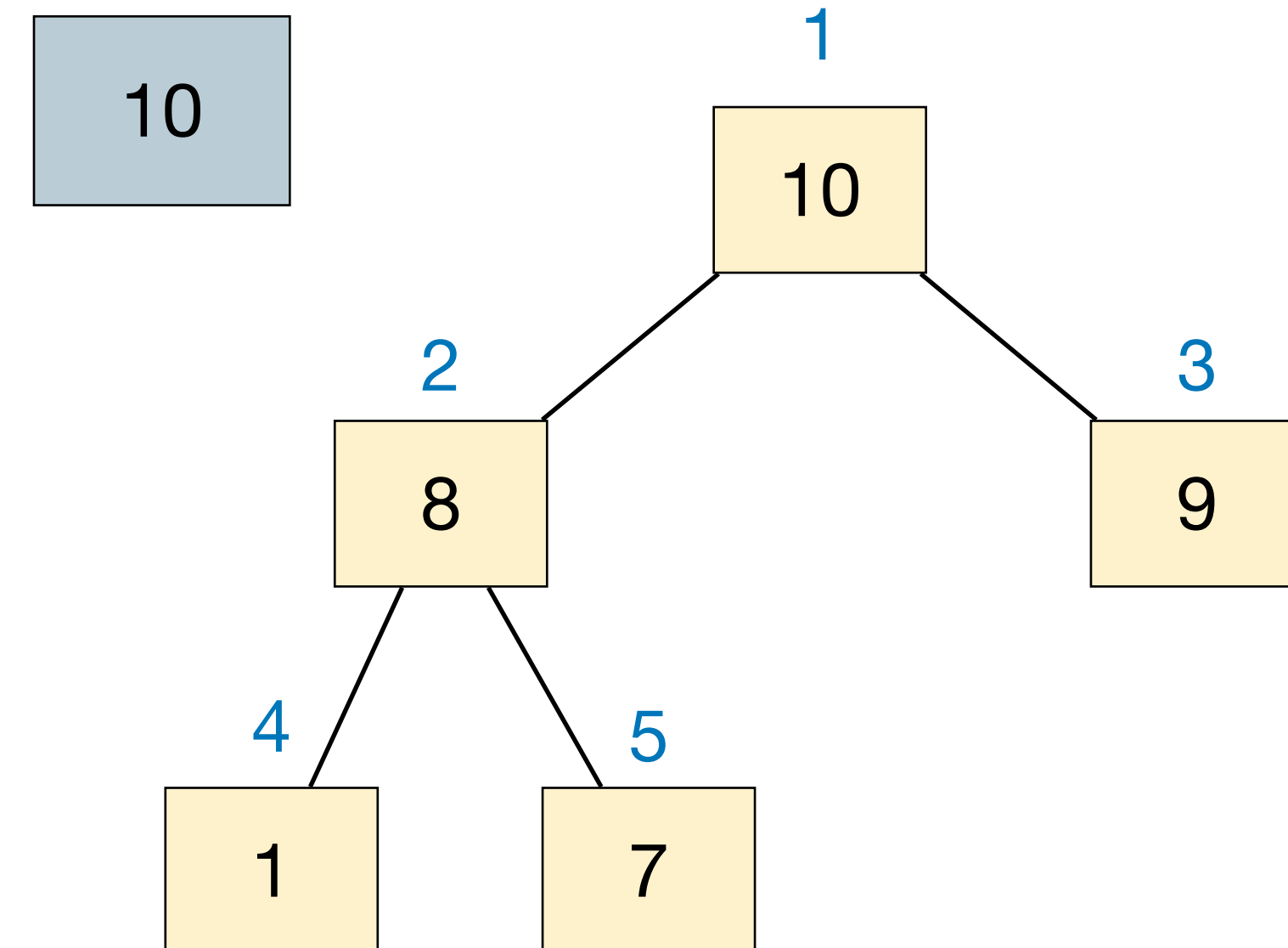
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
10	8	9	1	7	14





HeapSort

HeapSort(I):

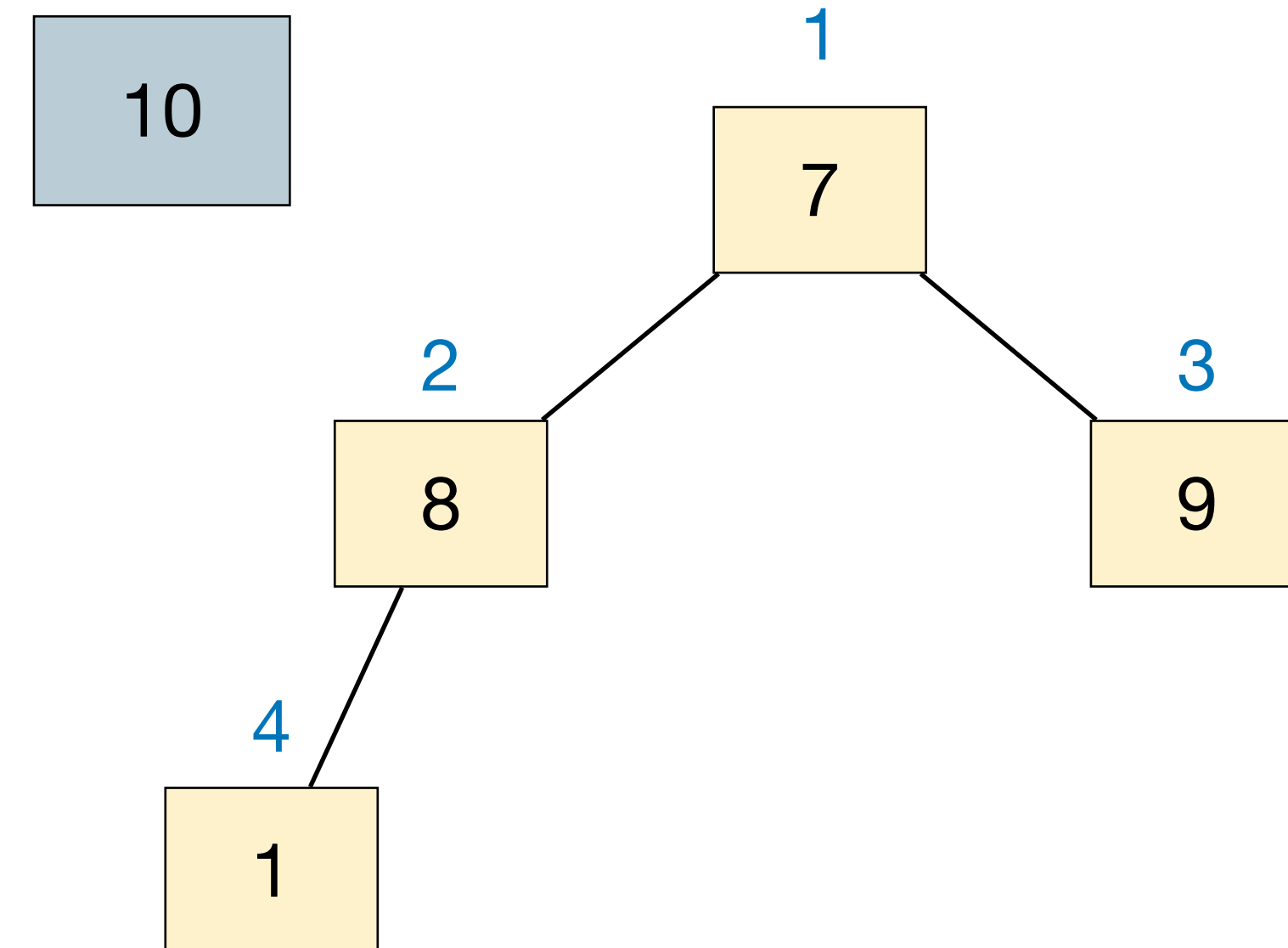
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
7	8	9	1	7	14





HeapSort

HeapSort(I):

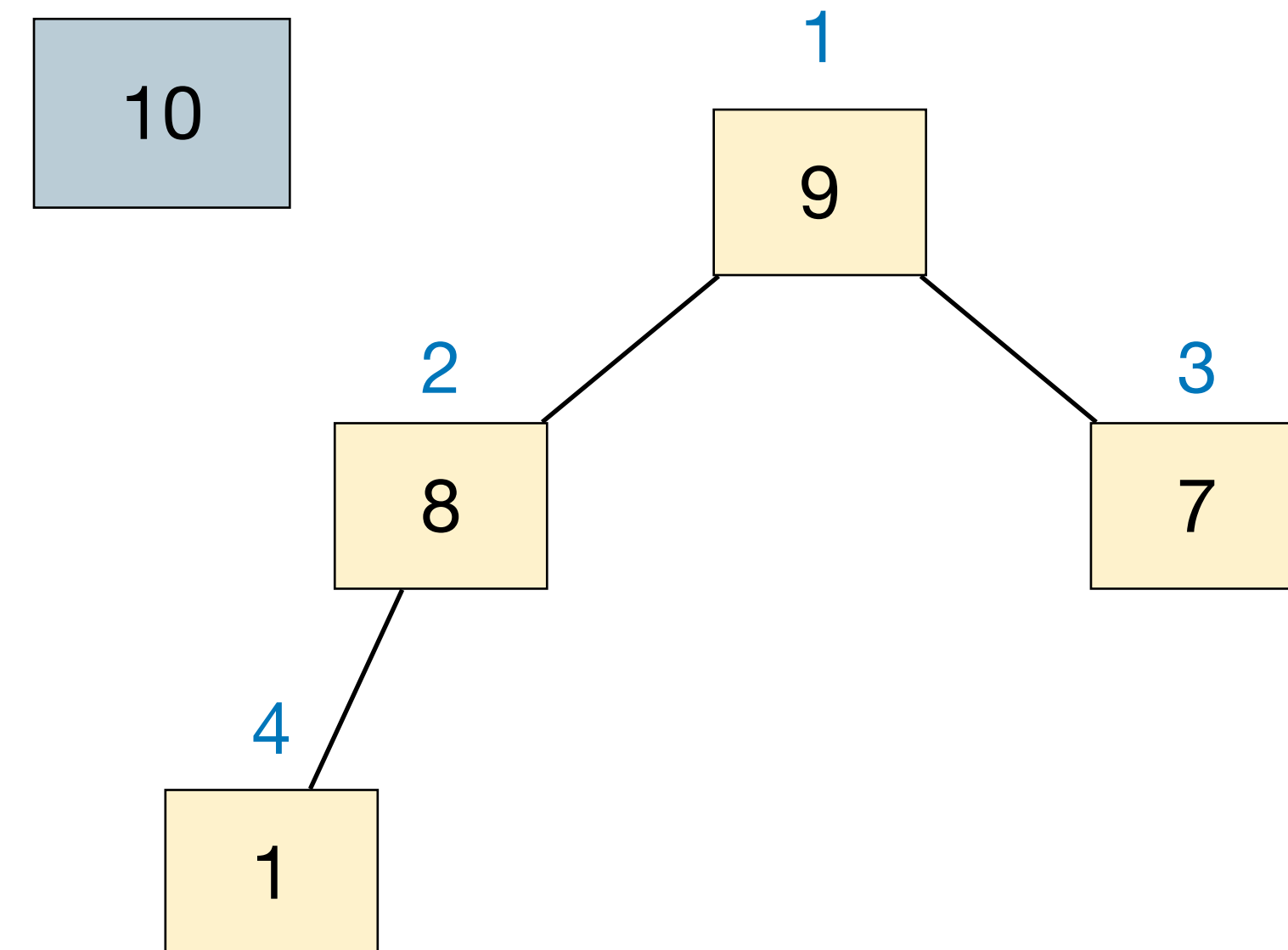
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
9	8	7	1	7	14



$i = 5$



HeapSort

HeapSort(I):

$heap := BuildMaxHeap(I)$

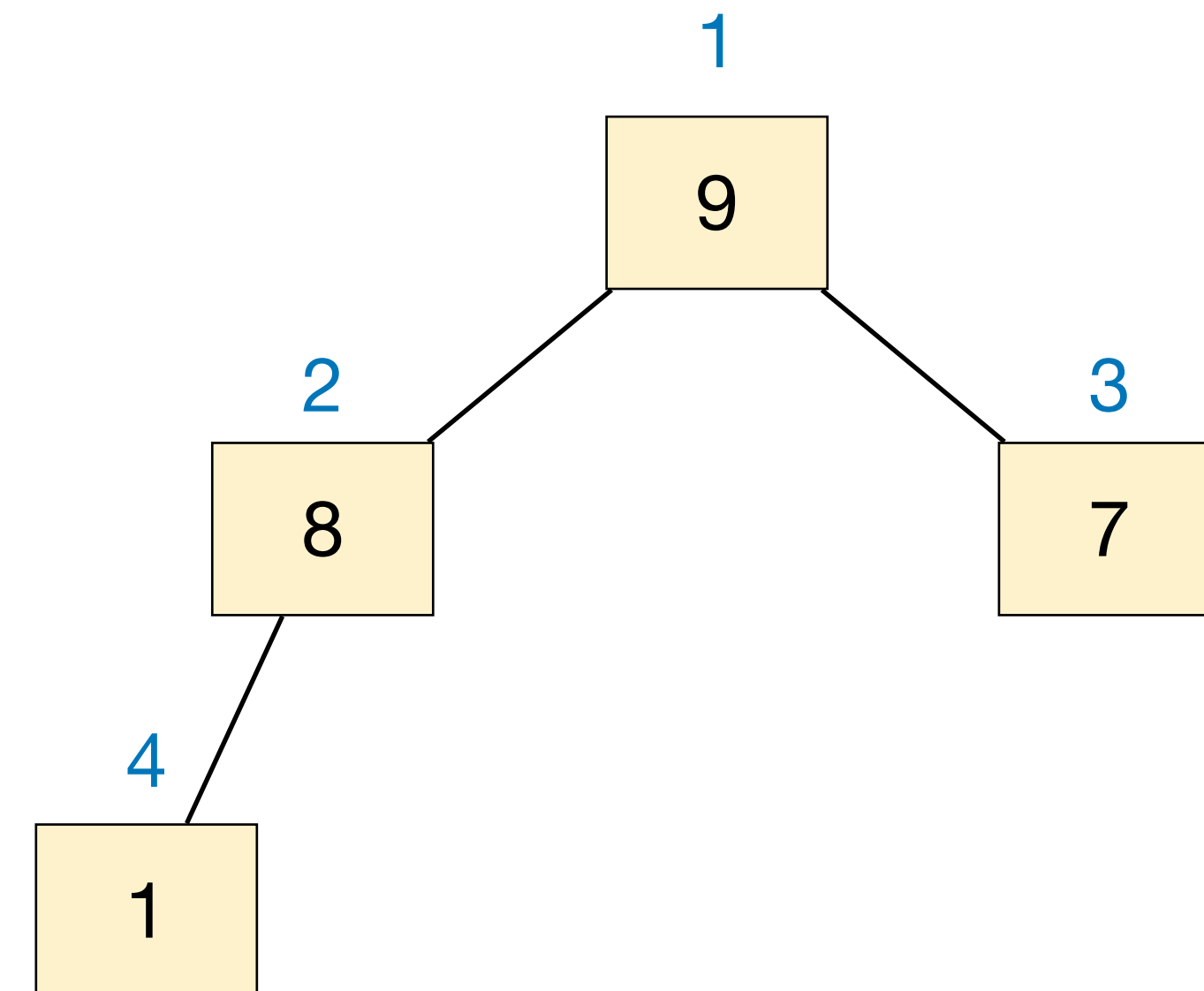
for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
9	8	7	1	10	14

$i = 5$





HeapSort

HeapSort(I):

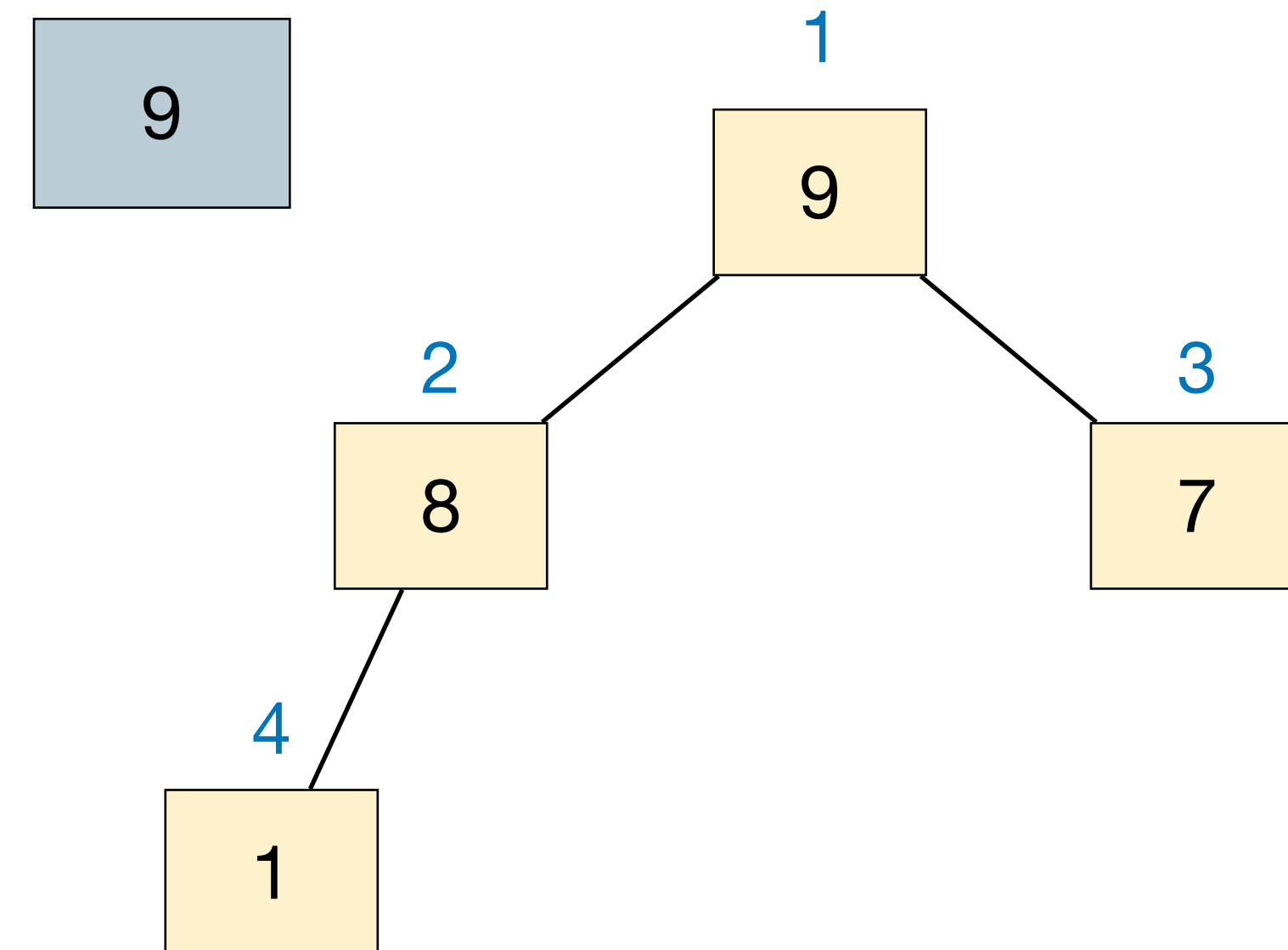
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
9	8	7	1	10	14



$i = 4$



HeapSort

HeapSort(I):

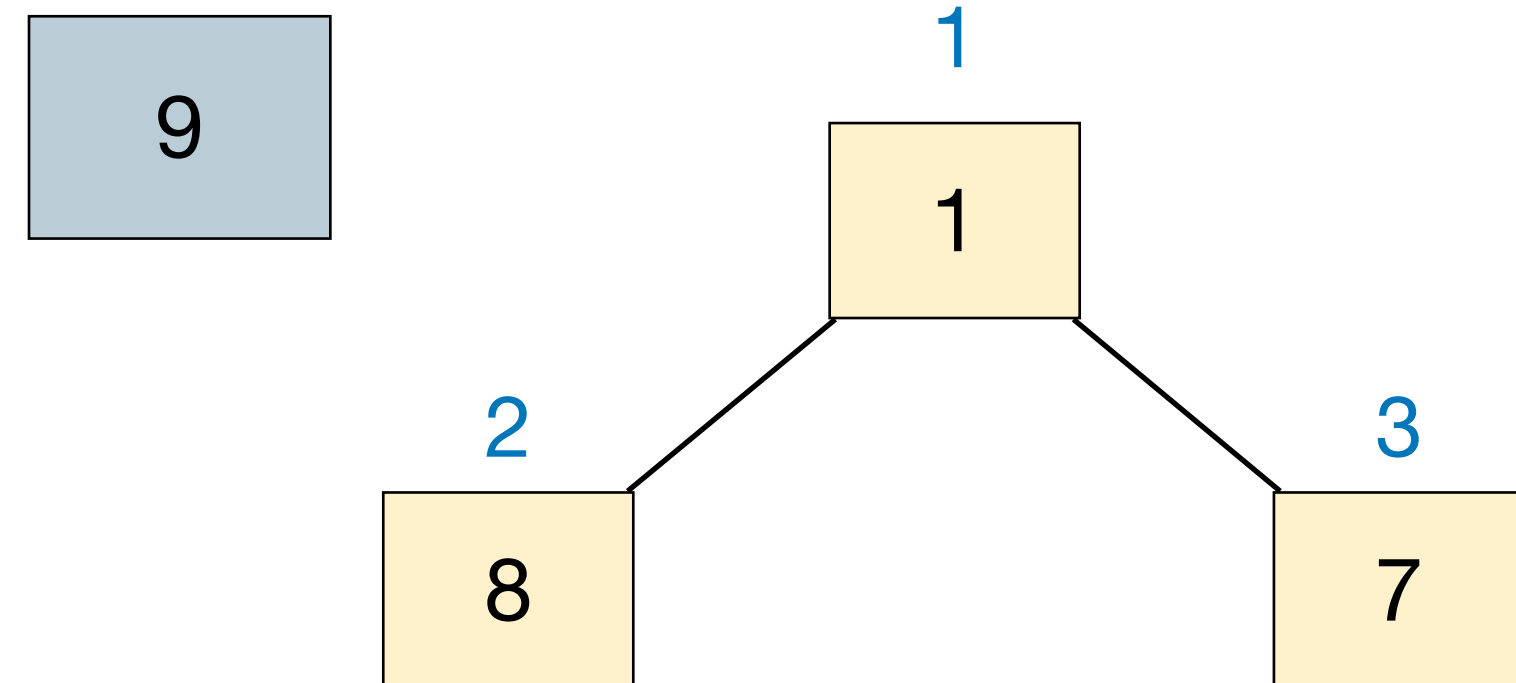
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
1	8	7	1	10	14



$i = 4$



HeapSort

HeapSort(I):

$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

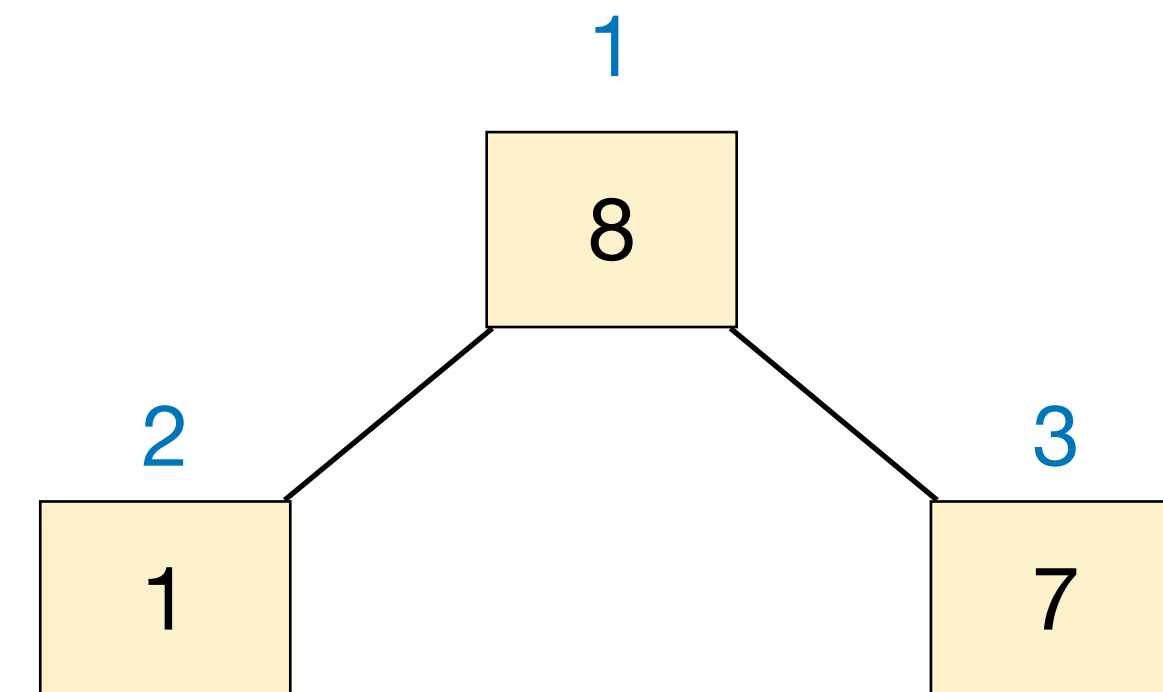
$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
8	1	7	1	10	14



$i = 4$





HeapSort

HeapSort(I):

$heap := BuildMaxHeap(I)$

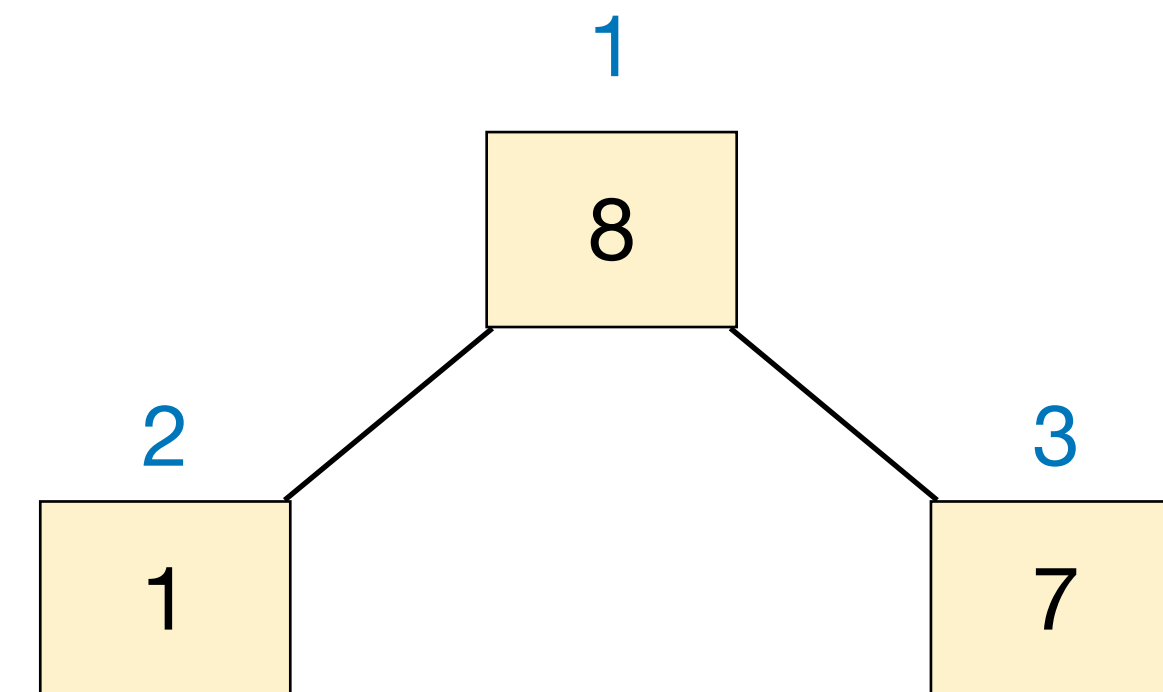
for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
8	1	7	9	10	14

$i = 4$





HeapSort

HeapSort(I):

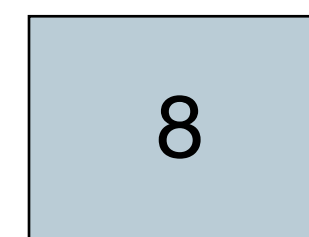
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

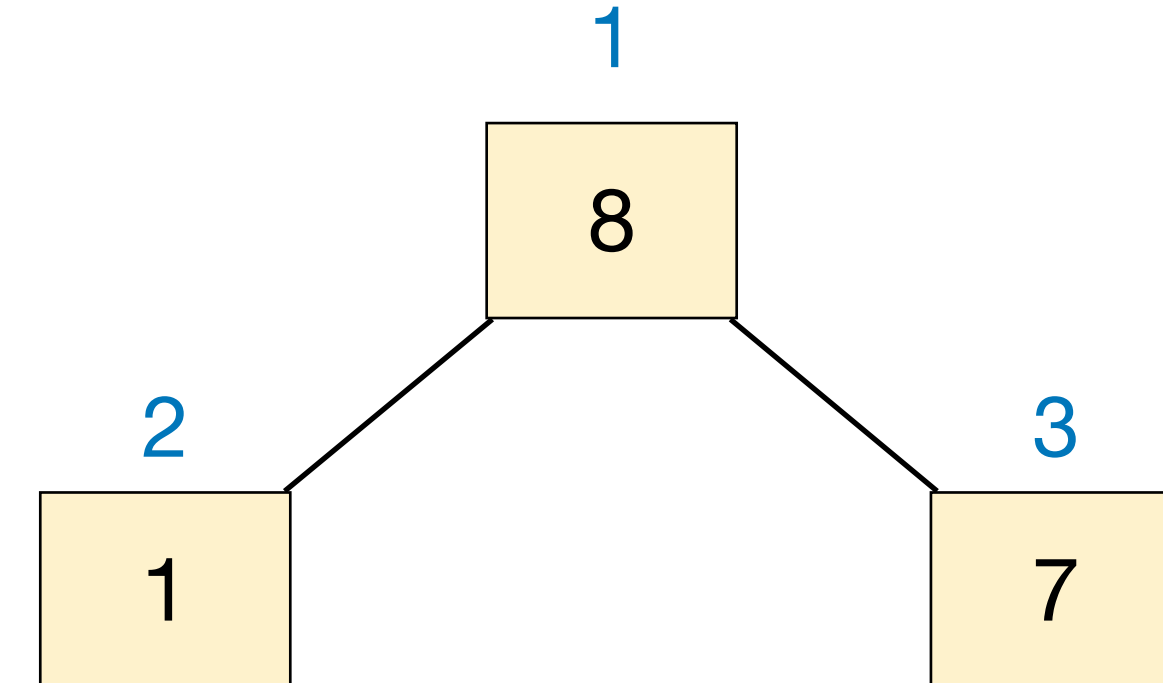
$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
8	1	7	9	10	14



$i = 3$





HeapSort

HeapSort(I):

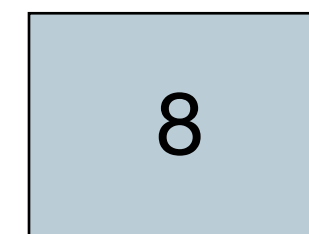
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

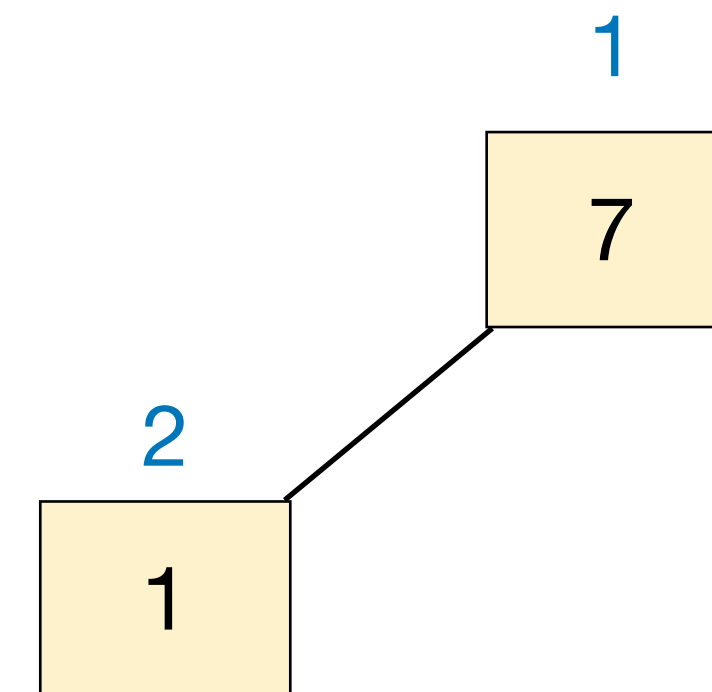
$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
7	1	7	9	10	14



$i = 3$





HeapSort

HeapSort(I):

$heap := BuildMaxHeap(I)$

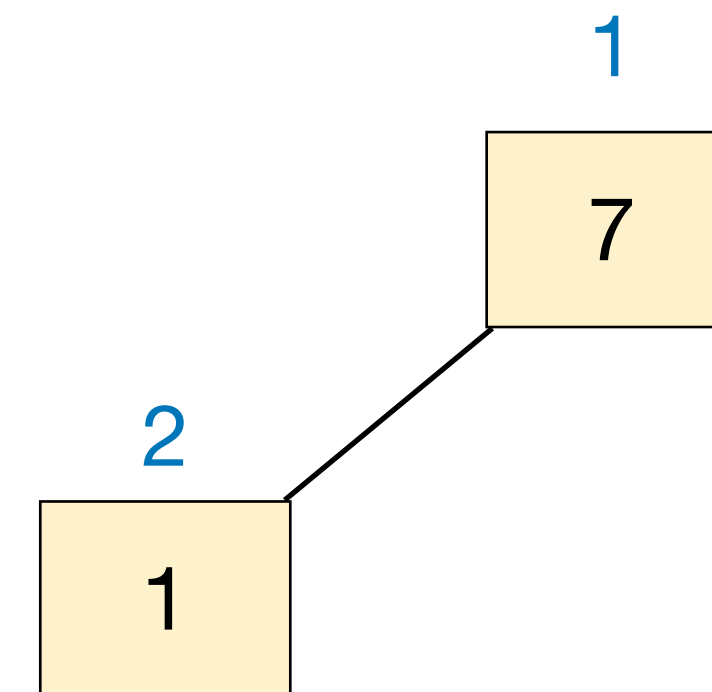
for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
7	1	8	9	10	14

$i = 3$





HeapSort

HeapSort(I):

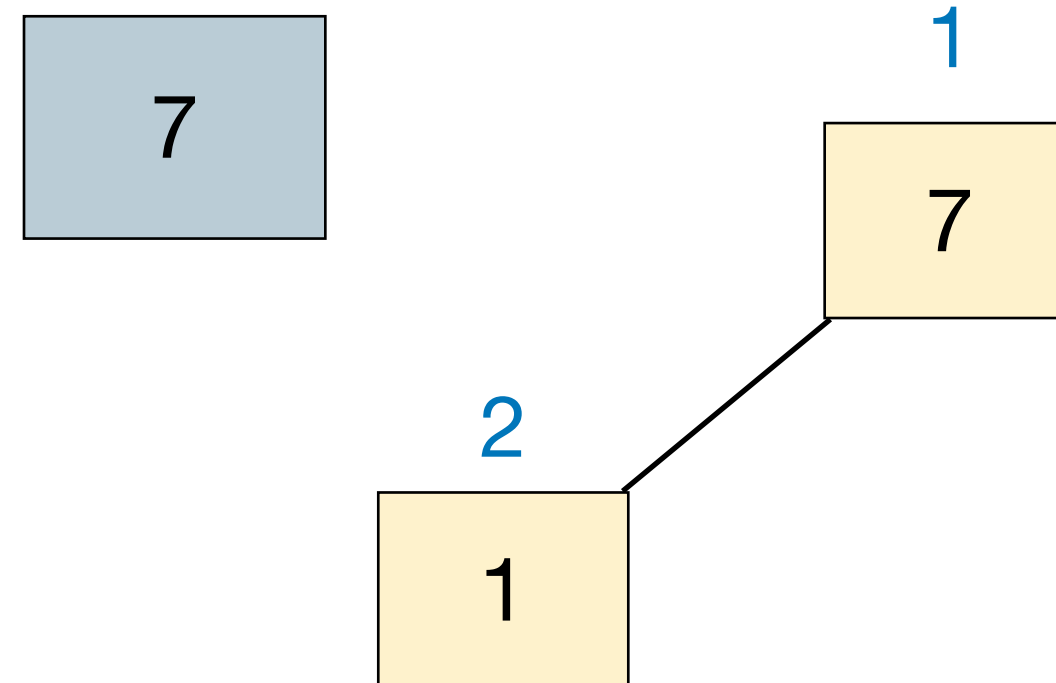
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
7	1	8	9	10	14



$i = 2$



HeapSort

HeapSort(I):

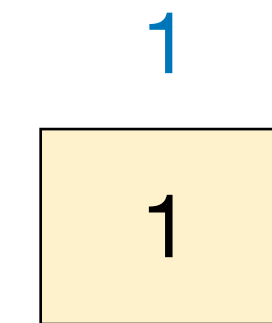
$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
1	1	8	9	10	14



$i = 2$



HeapSort

HeapSort(I):

$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1	2	3	4	5	6
1	7	8	9	10	14

$i = 2$

1
1



HeapSort

HeapSort(I):

$heap := BuildMaxHeap(I)$

for $i := n$ **down to** 2

$cur_max := heap.HeapExtractMax()$

$I[i] := cur_max$

1. Keep a **copy** of the root item
2. Remove last item and put it to root
3. Maintain heap property
4. Return the **copy** of the root item

- Total runtime of these iterations

$$\sum_{i=2}^n O(\lg i) = O(\lg(n!)) = O(n \lg n)$$

Stirling's formula



HeapSort

HeapSort(I):

heap := BuildMaxHeap(*I*)

for *i* := *n* **down to** 2

cur_max := *heap*.HeapExtractMax()

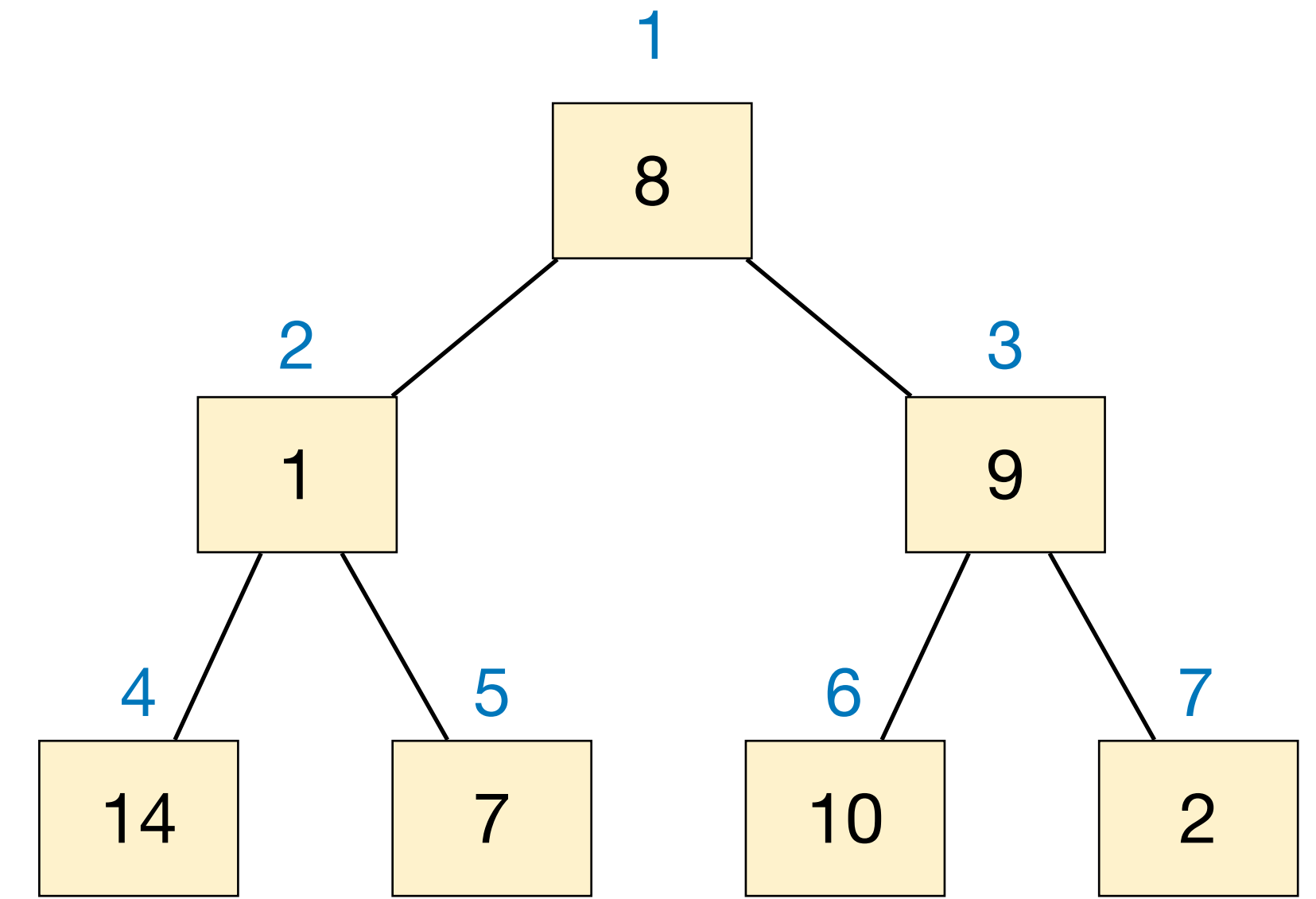
I[*i*] := *cur_max*

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Start with an empty heap, then call HeapInsert n times?
 - Cost is $\sum_{i=1}^n O(\lg i) = O(n \lg n)$
 - Not bad, but we can do better.



HeapSort

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.

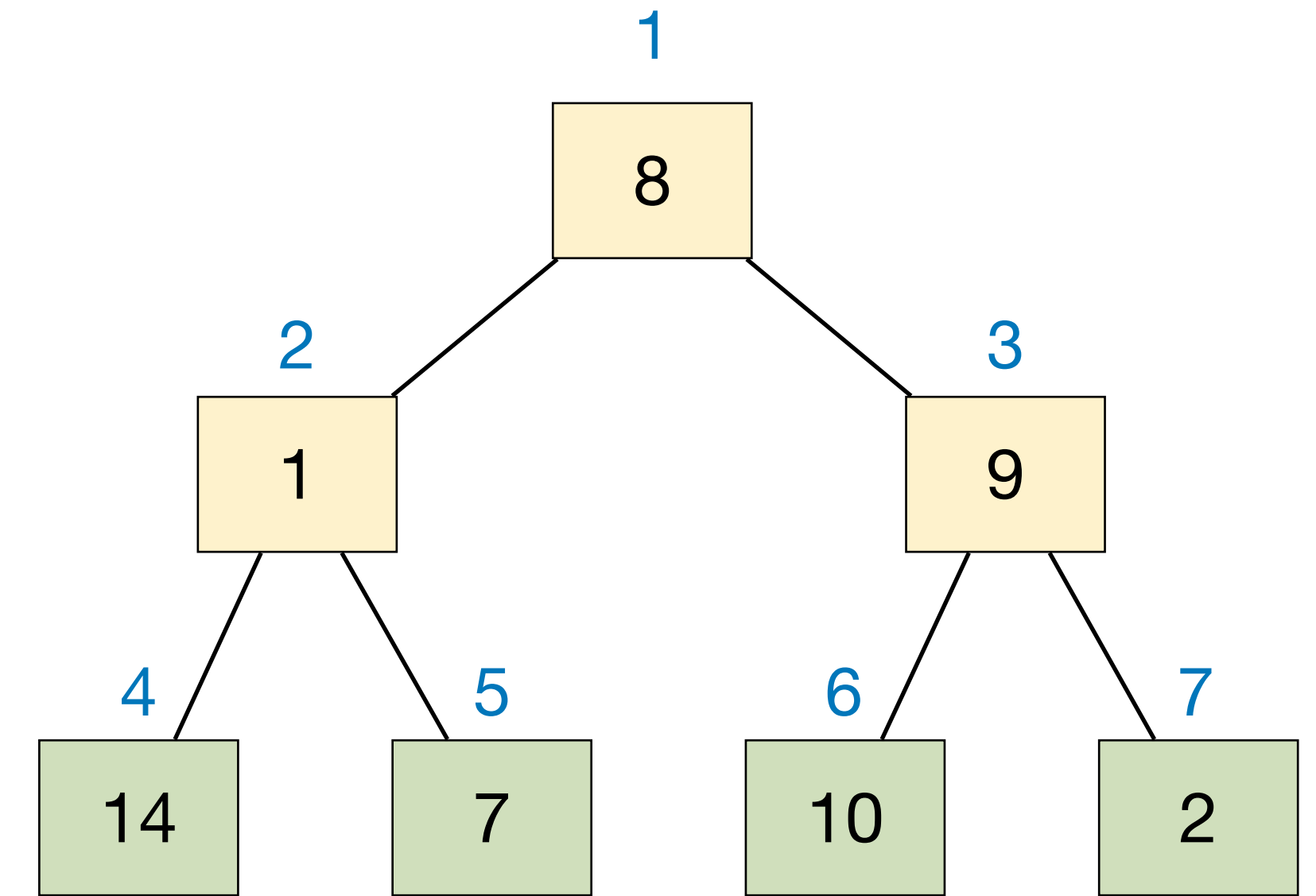


1	2	3	4	5	6	7
8	1	9	14	7	10	2



HeapSort

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
 - Each leaf node is a 1-item heap.

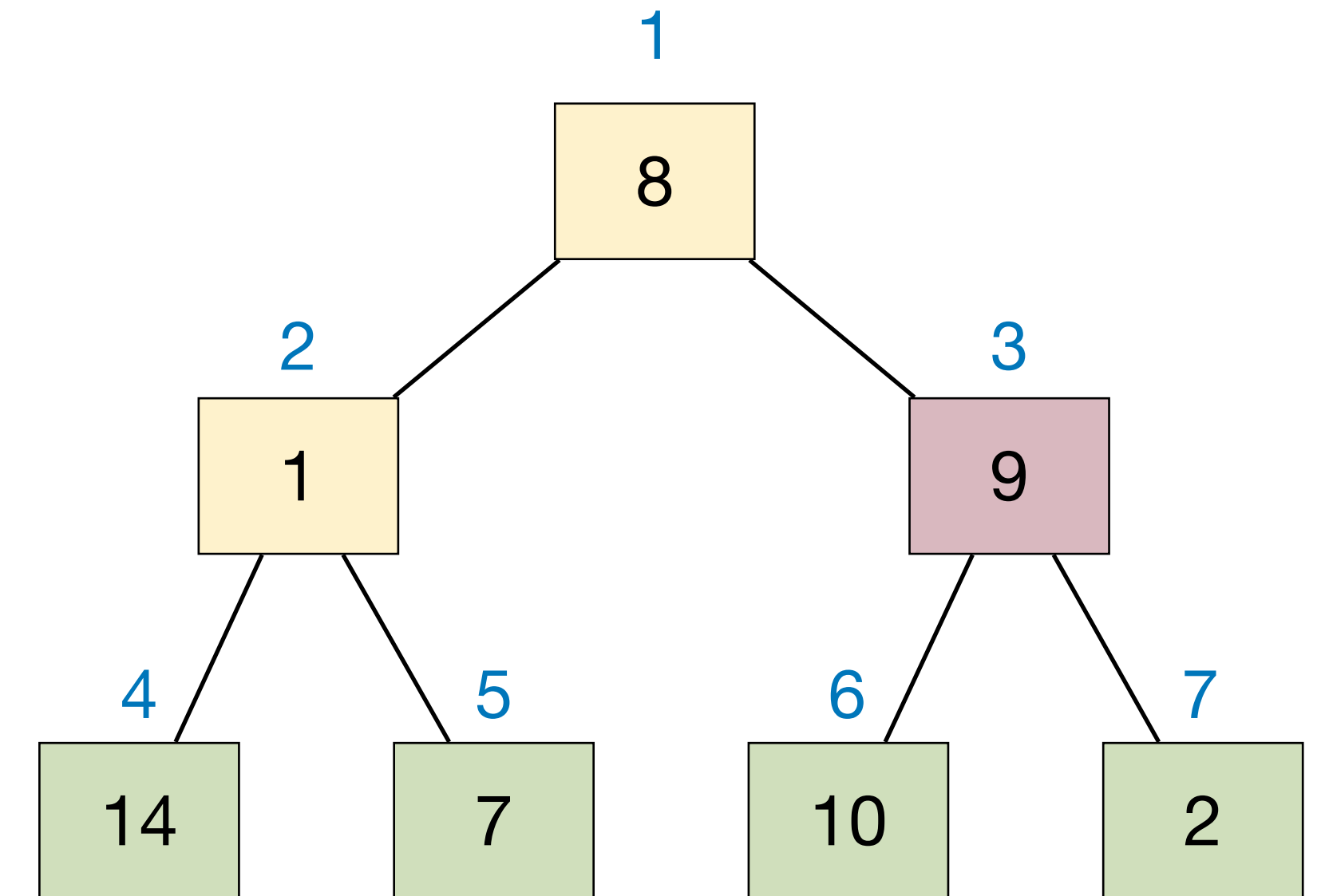


1	2	3	4	5	6	7
8	1	9	14	7	10	2



HeapSort

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
 - Each leaf node is a 1-item heap.
 - Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.

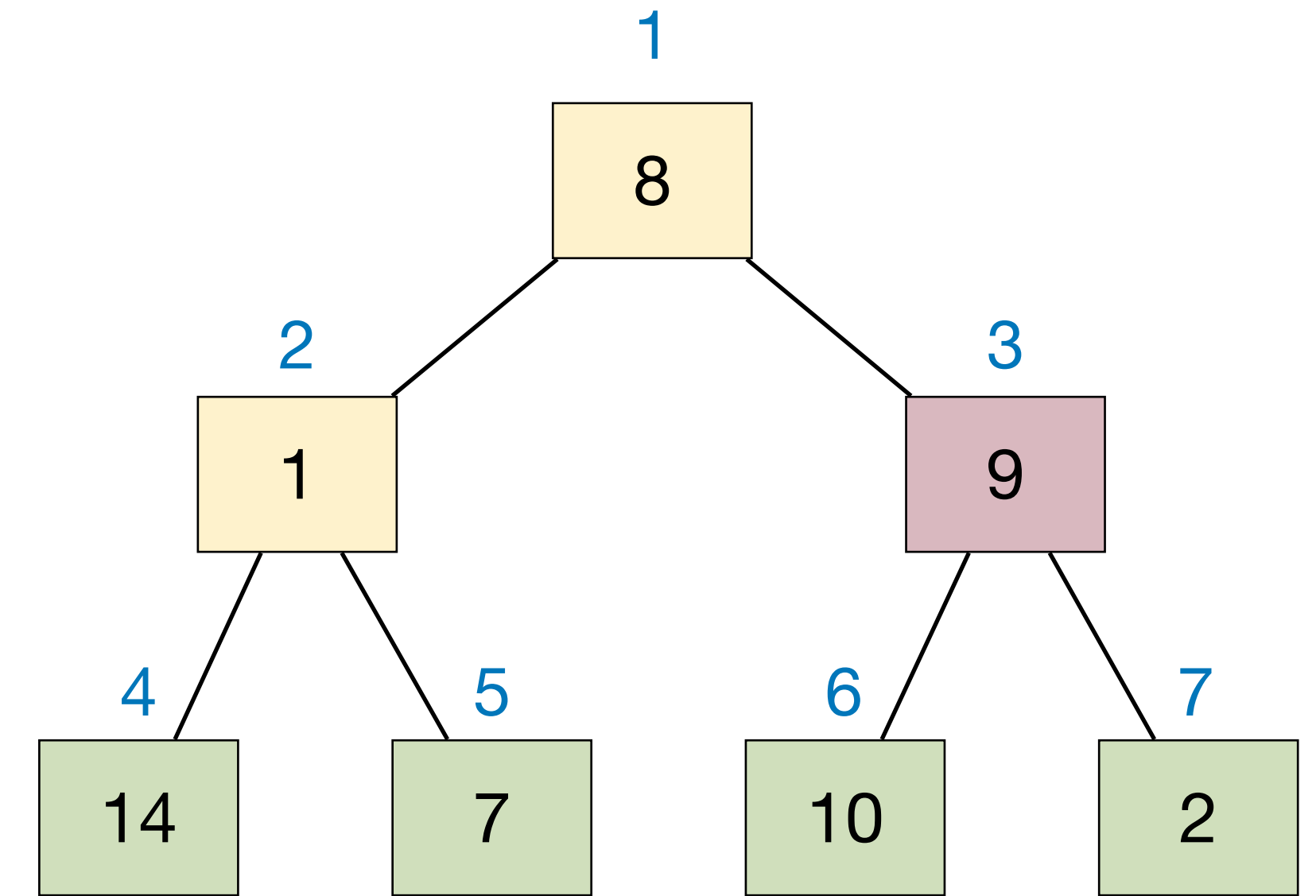


1	2	3	4	5	6	7
8	1	9	14	7	10	2



HeapSort

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
 - Each leaf node is a 1-item heap.
 - Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
 - Maintain heap property during merging: use `MaxHeapify`.

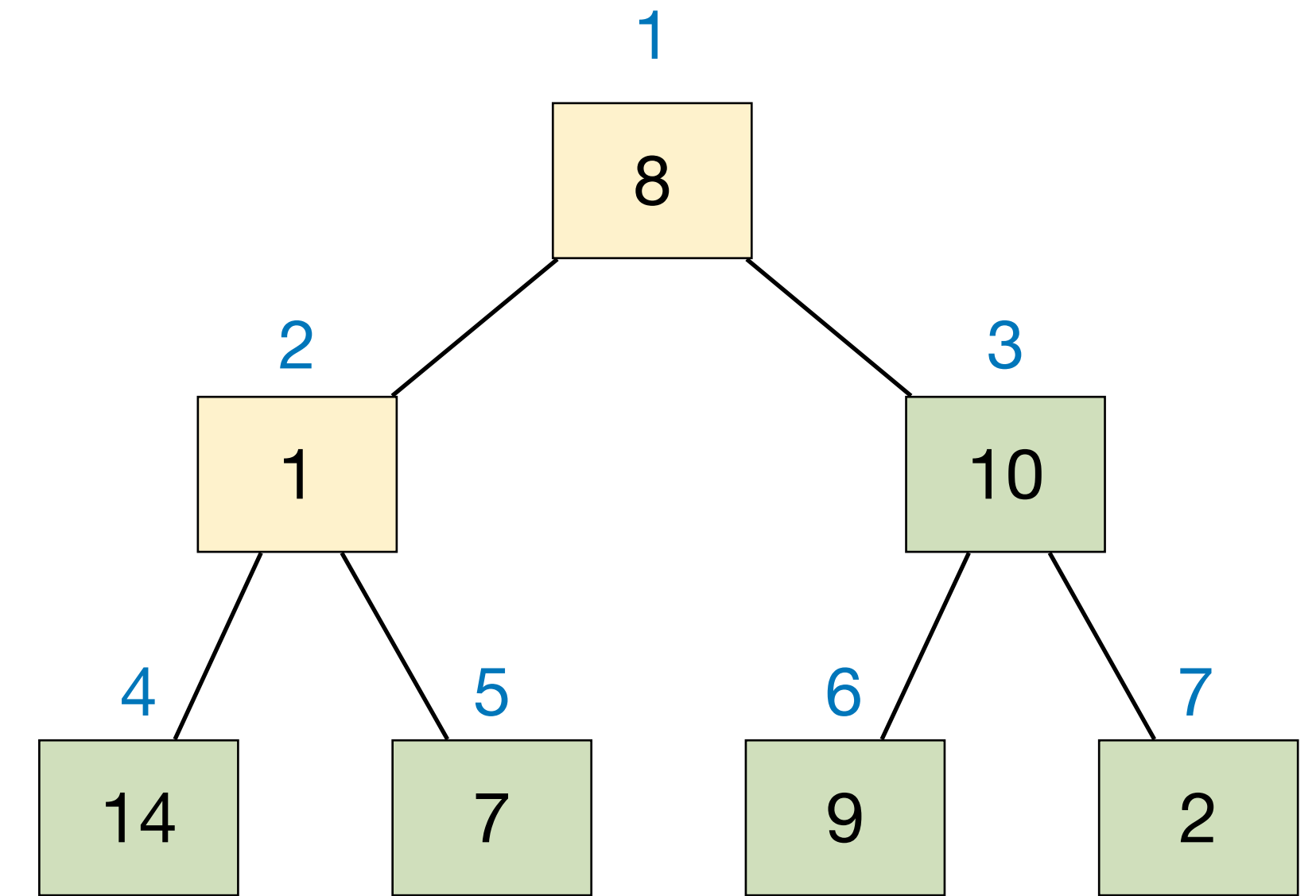


1	2	3	4	5	6	7
8	1	9	14	7	10	2



HeapSort

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
 - Each leaf node is a 1-item heap.
 - Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
 - Maintain heap property during merging: use `MaxHeapify`.

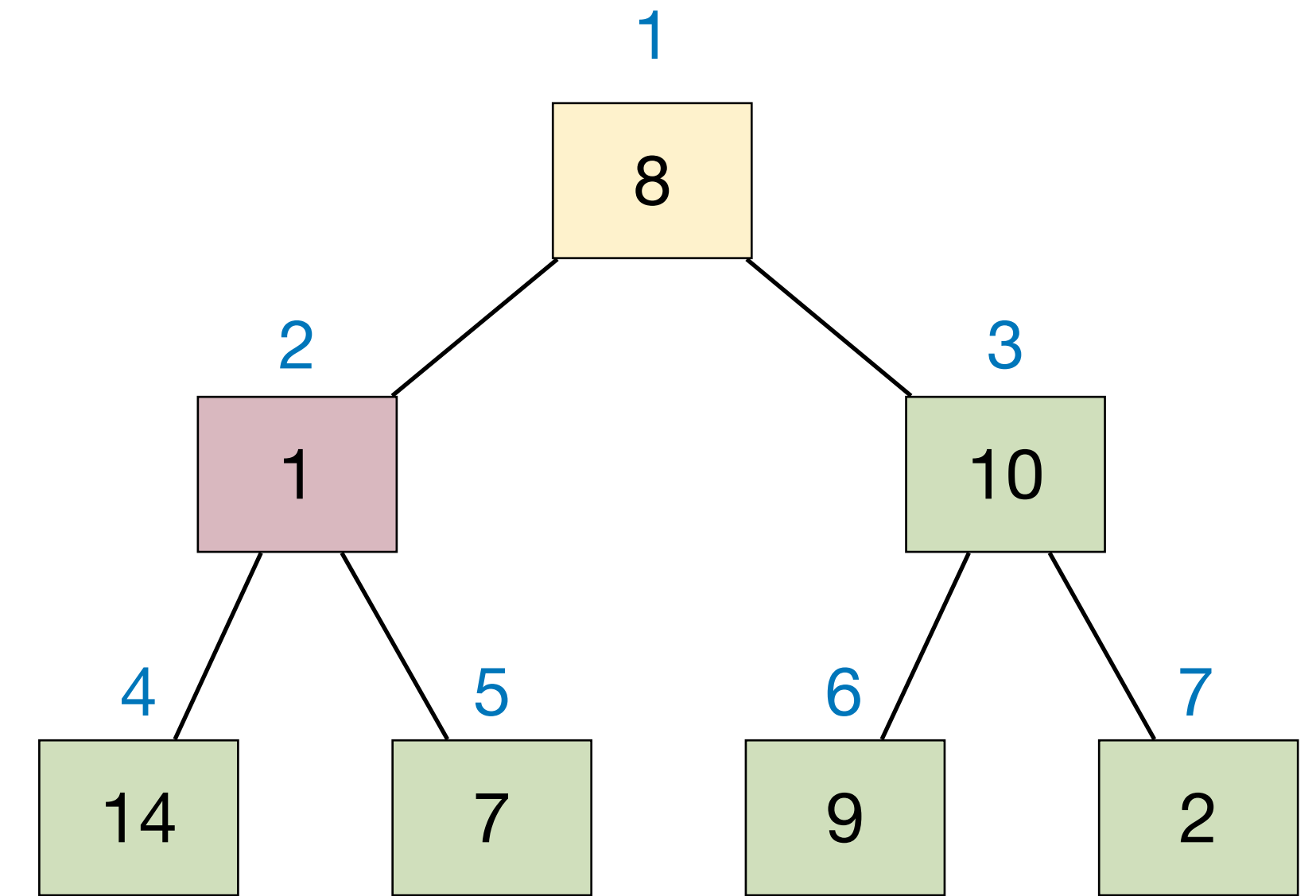


1	2	3	4	5	6	7
8	1	10	14	7	9	2



HeapSort

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
 - Each leaf node is a 1-item heap.
 - Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
 - Maintain heap property during merging: use `MaxHeapify`.

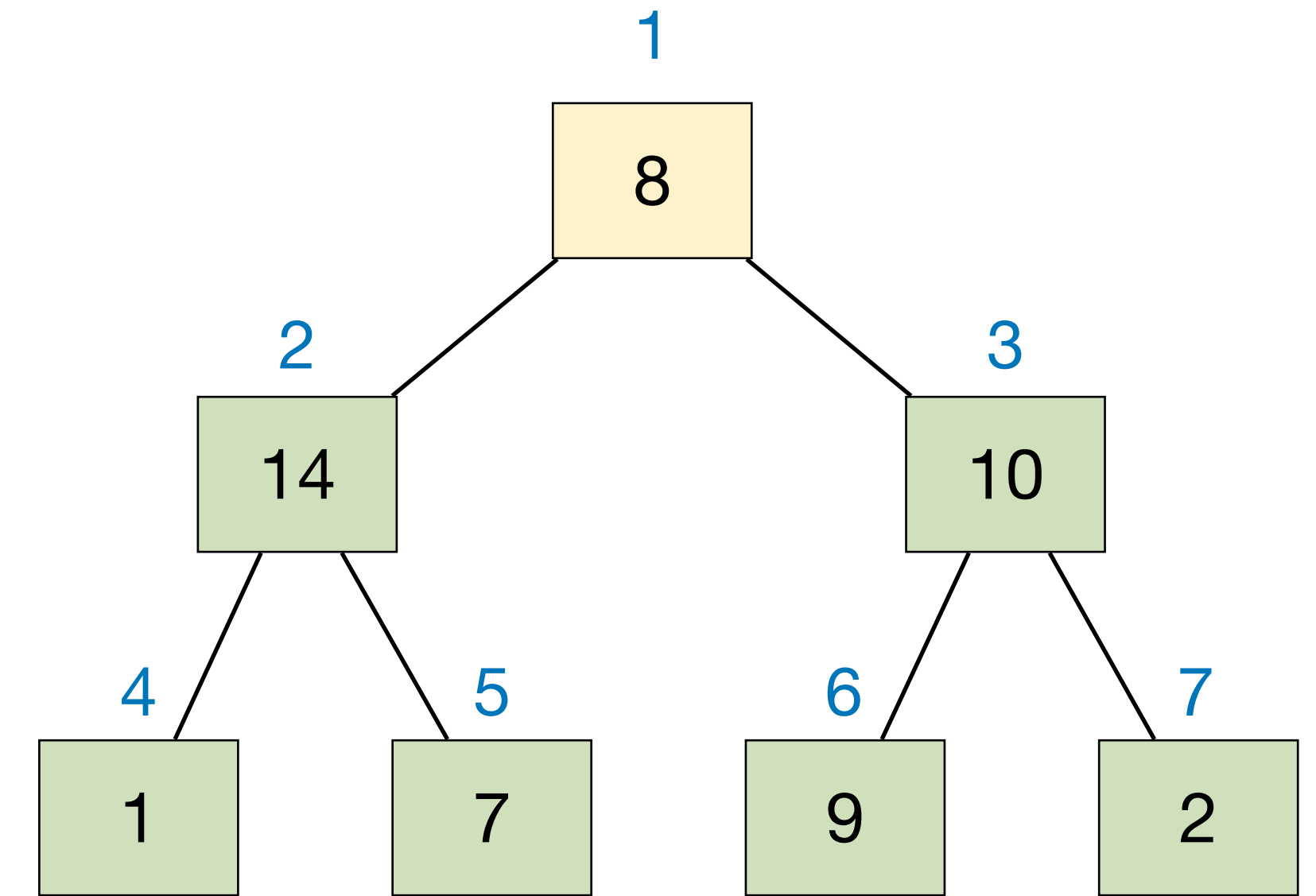


1	2	3	4	5	6	7
8	1	10	14	7	9	2



HeapSort

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
 - Each leaf node is a 1-item heap.
 - Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
 - Maintain heap property during merging: use `MaxHeapify`.

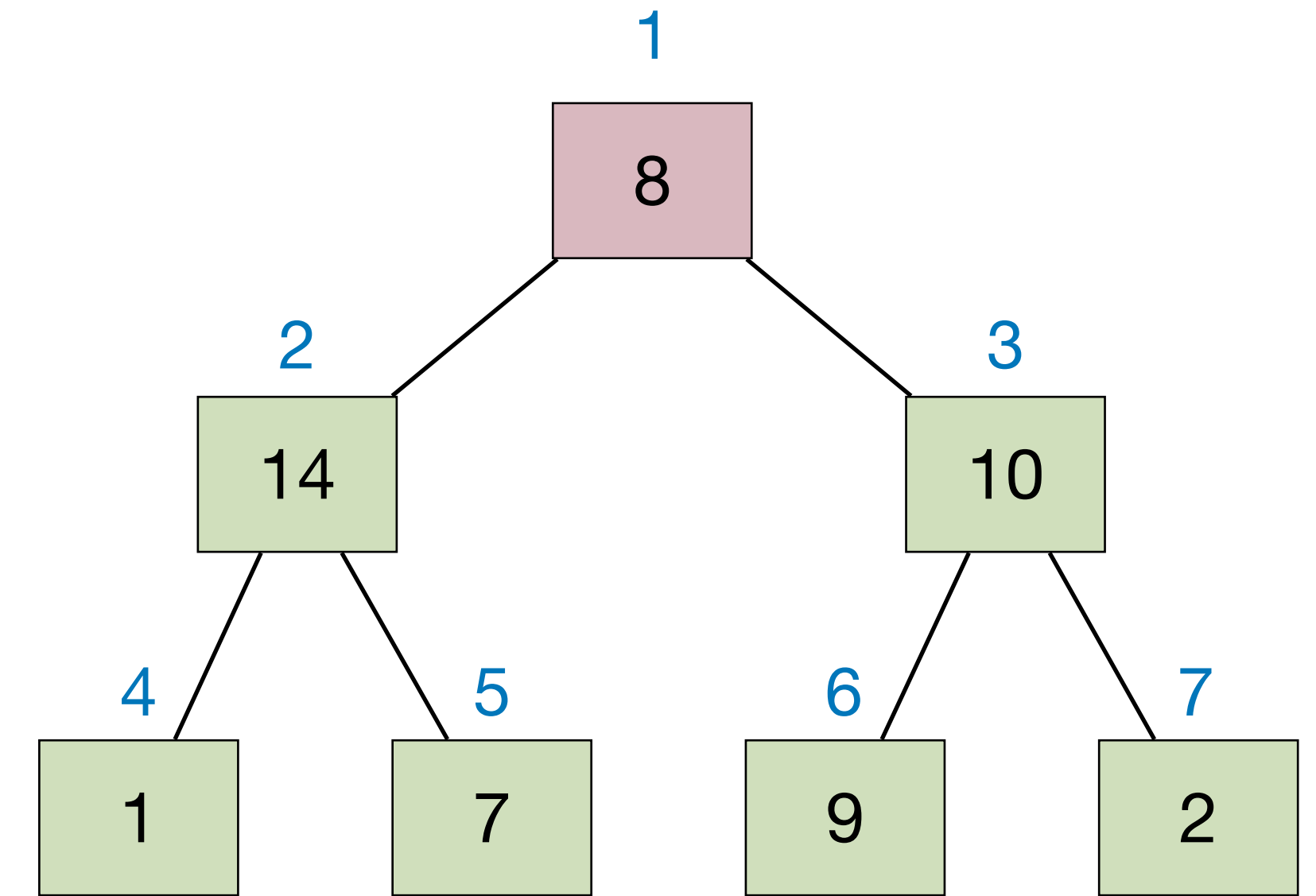


1	2	3	4	5	6	7
8	14	10	1	7	9	2



HeapSort

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
 - Each leaf node is a 1-item heap.
 - Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
 - Maintain heap property during merging: use `MaxHeapify`.

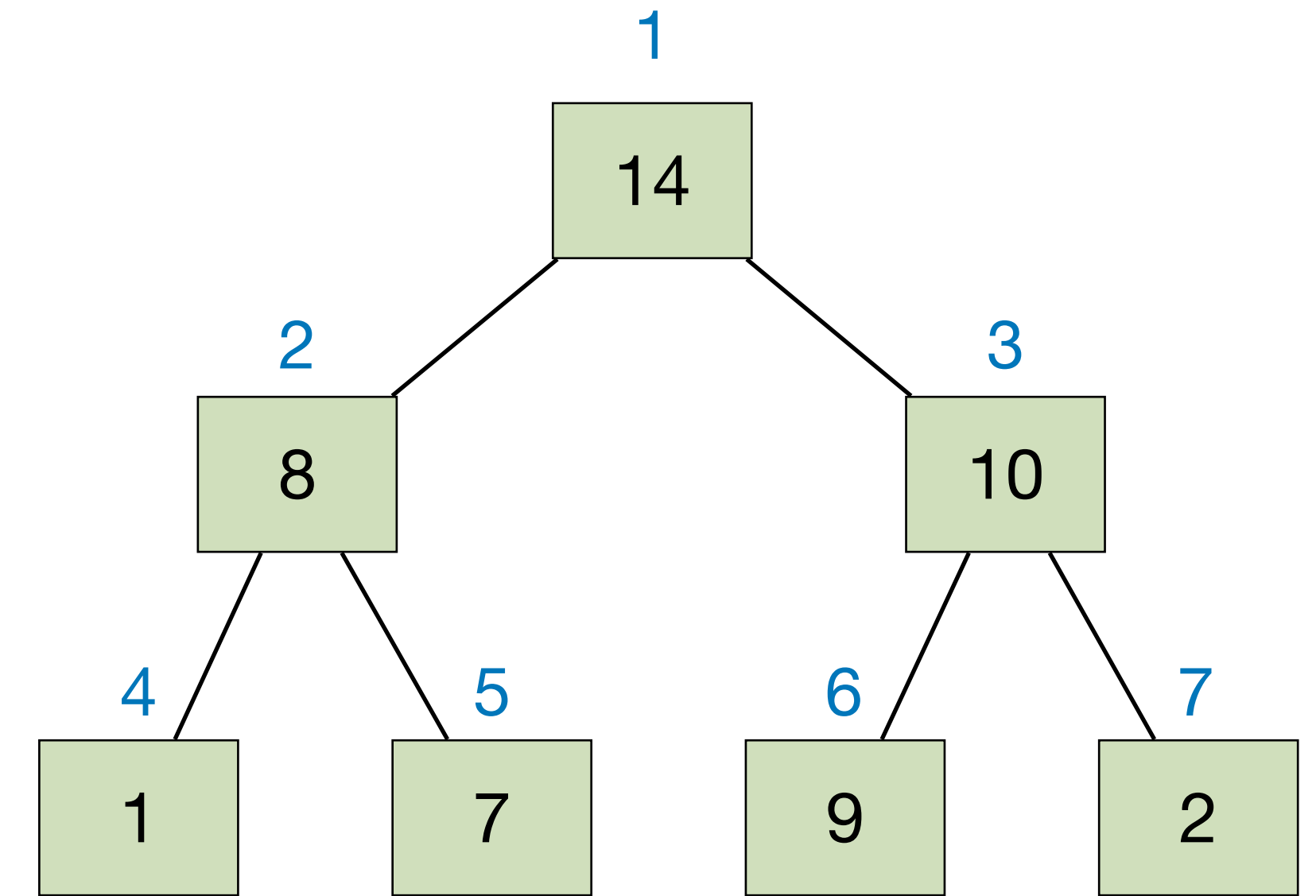


1	2	3	4	5	6	7
8	14	10	1	7	9	2



HeapSort

- Given an array $I[1 \dots n]$, how to build a max-heap?
 - Bottom-up approach: keep merging small heaps into larger ones, until a single heap remains.
 - Each leaf node is a 1-item heap.
 - Go through remaining nodes in index decreasing order: at each node, we are merging two heaps.
 - Maintain heap property during merging: use `MaxHeapify`.



1	2	3	4	5	6	7
14	8	10	1	7	9	2



HeapSort

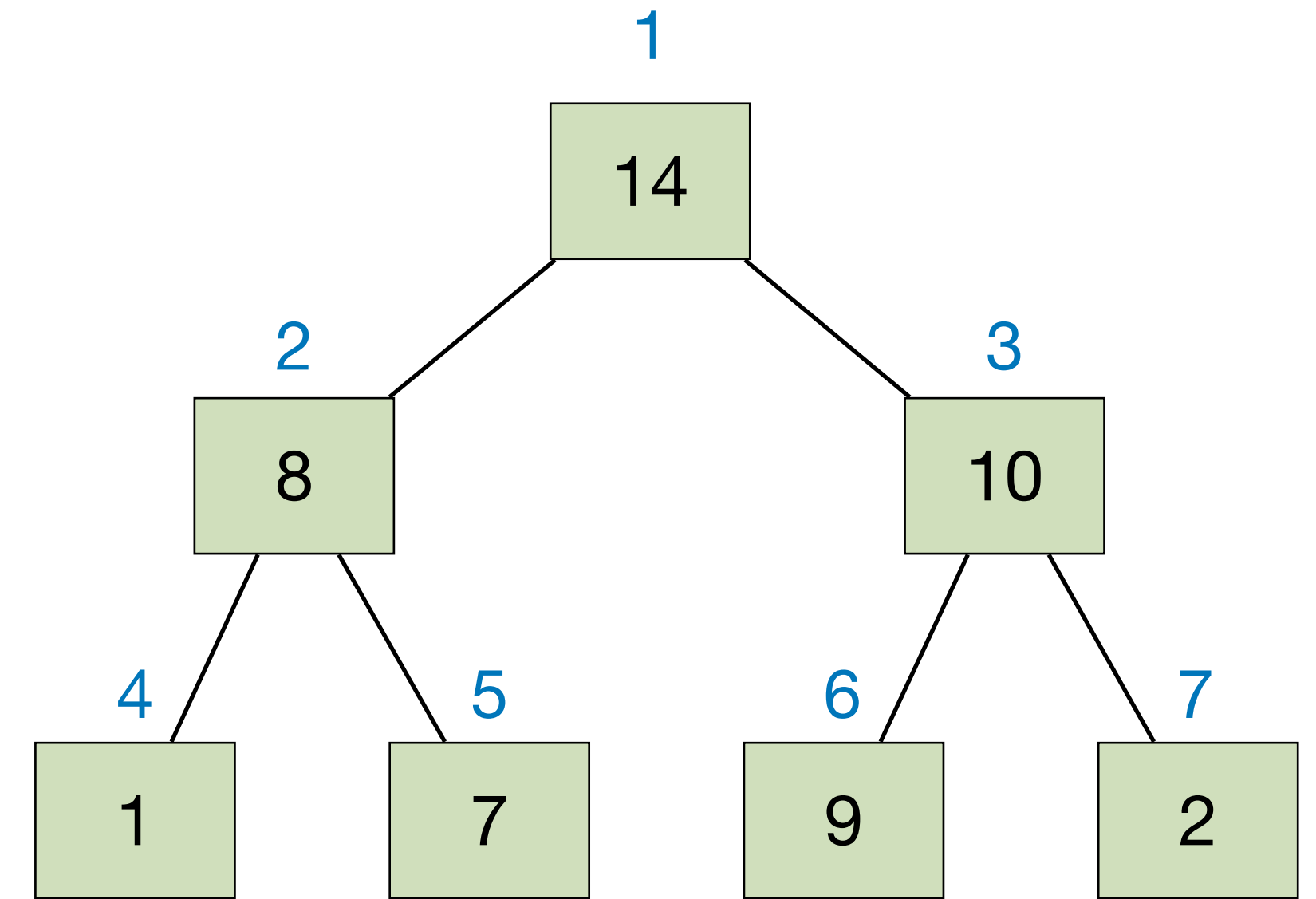
BuildMaxHeap(A):

$heap_size := n$

for $i := \text{Floor}(n/2)$ **down to** 1

$\text{MaxHeapify}(i, A)$

- Time complexity of BuildMaxHeap?
 - $\Theta(n)$ calls to MaxHeapify, each costing $O(\lg n)$, so $O(n \lg n)$?
 - Correct but not tight...



1	2	3	4	5	6	7
14	8	10	1	7	9	2



HeapSort

BuildMaxHeap(A):

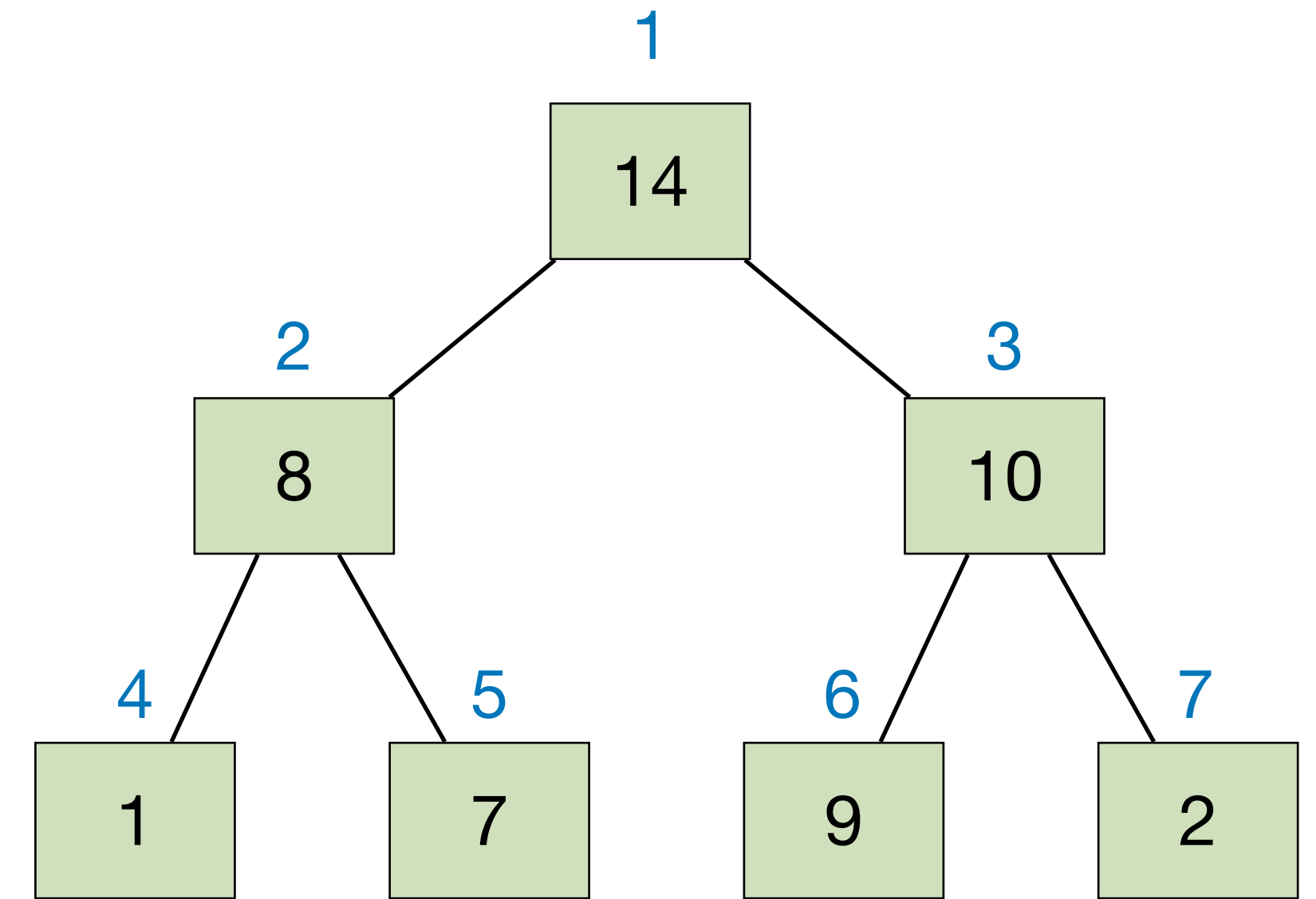
heap_size := *n*

for *i* := *Floor*(*n*/2) **down to** 1

MaxHeapify(*i*, *A*)

- Height of *n*-items heap is $\lfloor \lg n \rfloor$
- Any height *h* has $\leq \lceil \frac{n}{2^{h+1}} \rceil$ nodes
- Cost of all MaxHeapify:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} (\lceil \frac{n}{2^{h+1}} \rceil \cdot O(h)) = O(n \cdot \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n)$$



1	2	3	4	5	6	7
14	8	10	1	7	9	2



HeapSort

HeapSort(I):

heap := BuildMaxHeap(*I*)

for *i* := *n* **down to** 2

cur_max := *heap*.HeapExtractMax()

I[*i*] := *cur_max*

BuildMaxHeap(A):

heap_size := *n*

for *i* := Floor(*n*/2) **down to** 1

MaxHeapify(*i*, *A*)

Time Complexity: $O(n)$

Time Complexity: $O(n \lg n)$

- Time complexity of HeapSort is $O(n \lg n)$.
- Extra space required during execution is $O(1)$.



Further reading

- [CLRS] Ch.6

