Binary Heap

Dr. Chung-Wen Albert Tsao

Overview

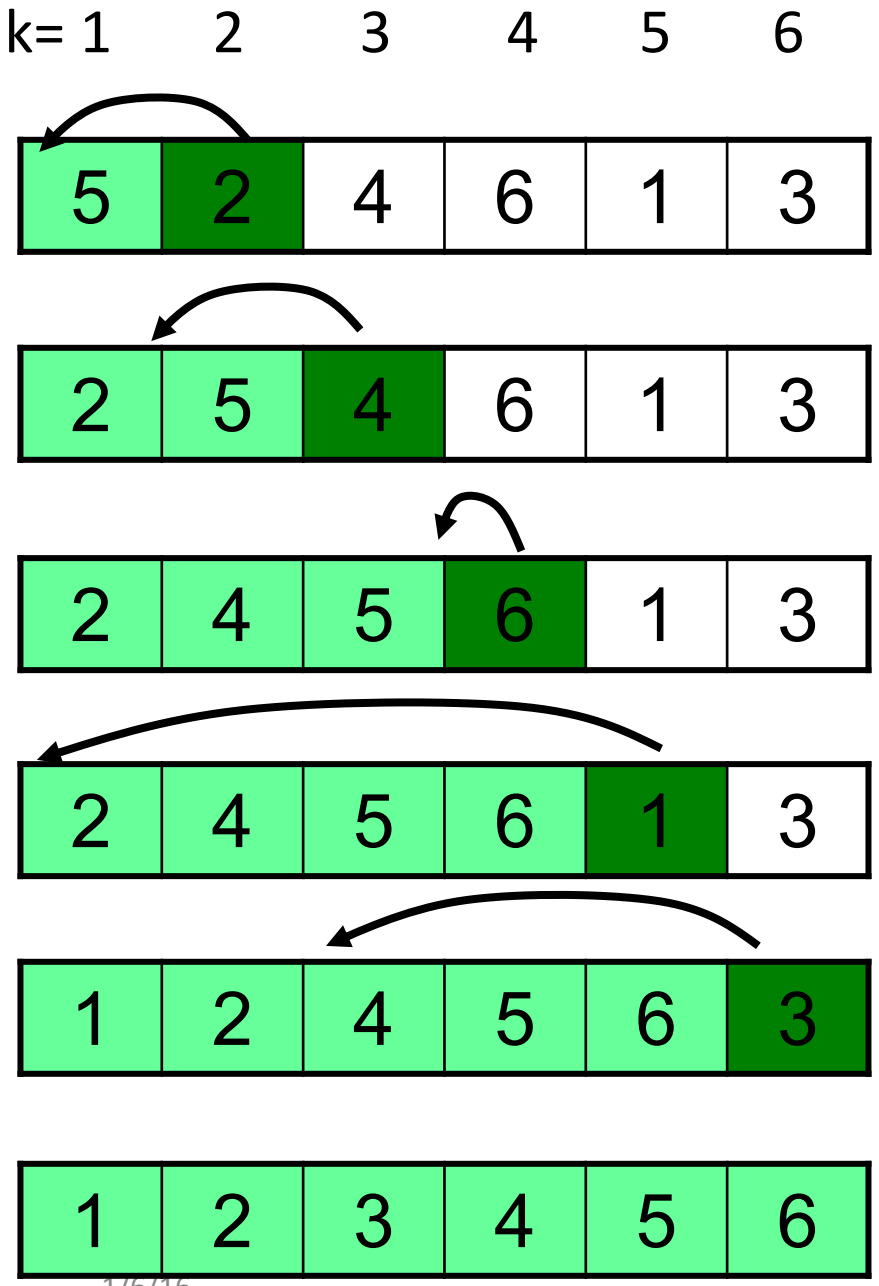
- Sorting overview
- Heaps
- Heapify
- BuildHeap
- Heapsort
- Priority Queues

Analyzing sorts

- Running time?
  - $O(n^2)$, O(nlogn), O(n)?
- Space – is it in-place?
  - Using a small, constant amount of extra storage space.
- Stable?
  - the initial order of equal items is preserved

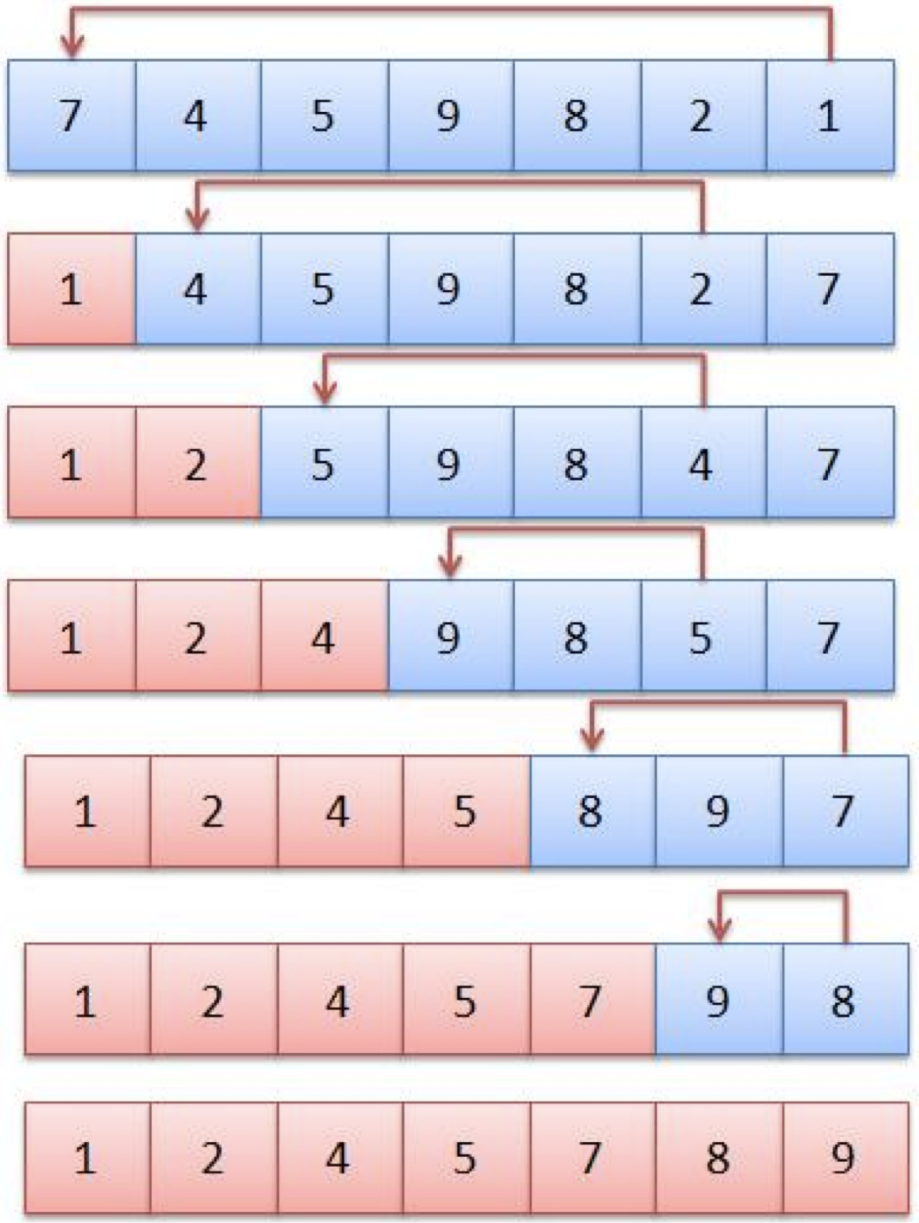
Analyzing sorts

- Running time?
  - $O(n^2)$, O(nlogn), O(n)?
- Space – is it in-place?
  - Using a small, constant amount of extra storage space.
- Stable?
  - the initial order of equal items is preserved



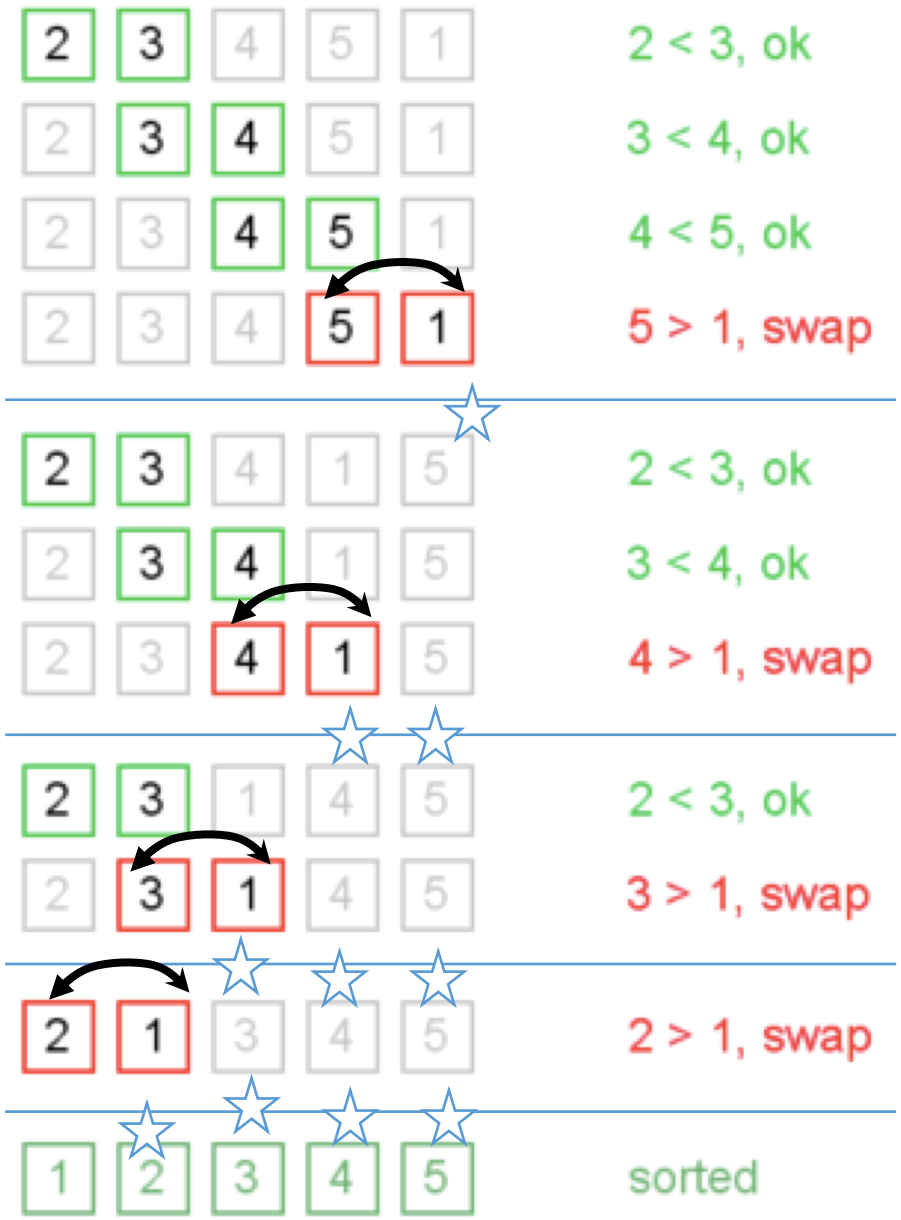
Insertion Sort

- Worst case $O(n^2)$
  - elements in reverse sorted order
- Best case O(n)
  - Already sorted order
- Stable
- At iteration k, first k elements are sorted, and they are the same first k elements from the unsorted set.



Selection Sort

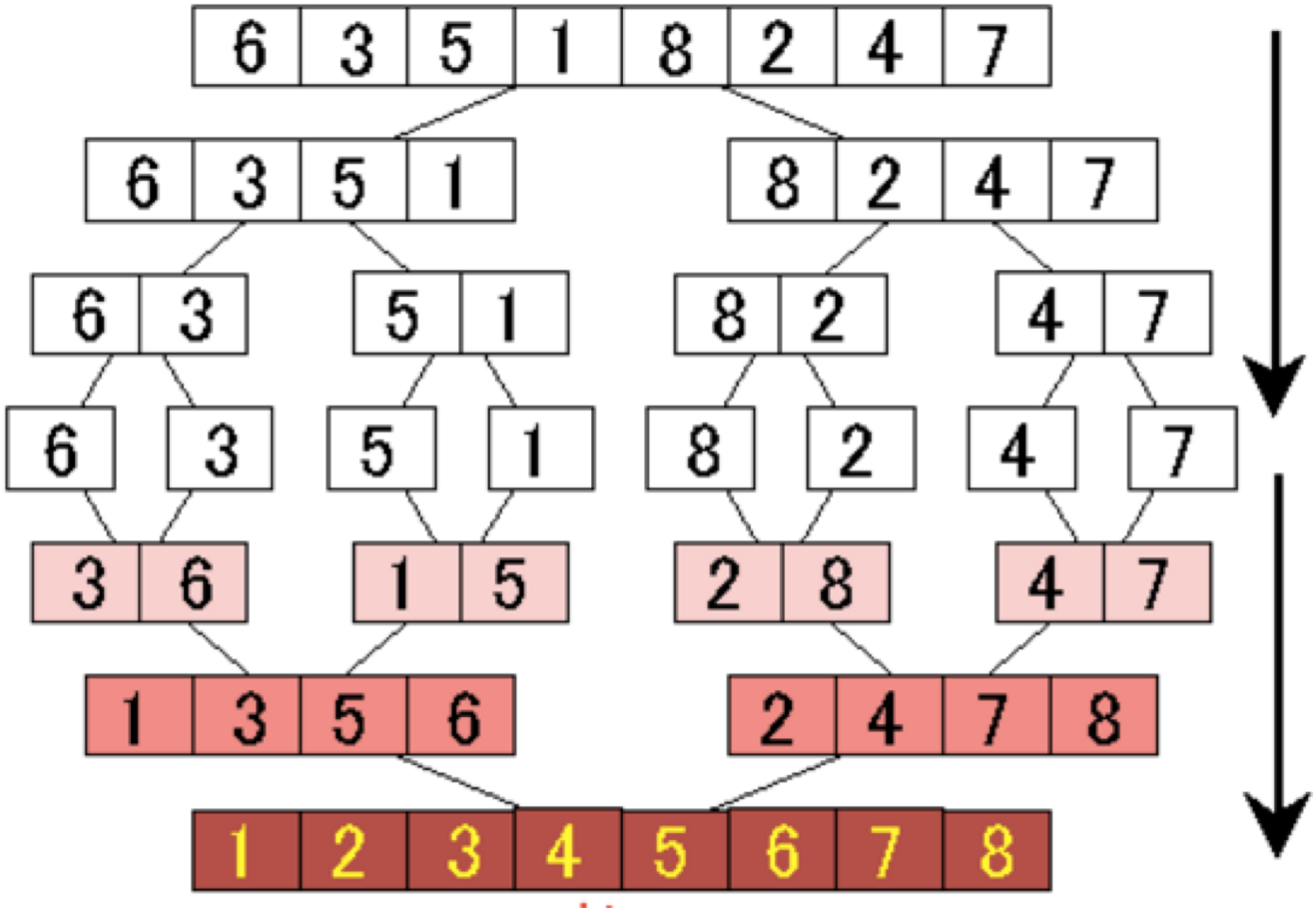
- $O(n^2)$
- Not Stable
- Does it perform better/worse for already sorted input? Reverse-sorted input?
- At iteration k, first k elements are sorted, and they are the same first k elements from the unsorted set.



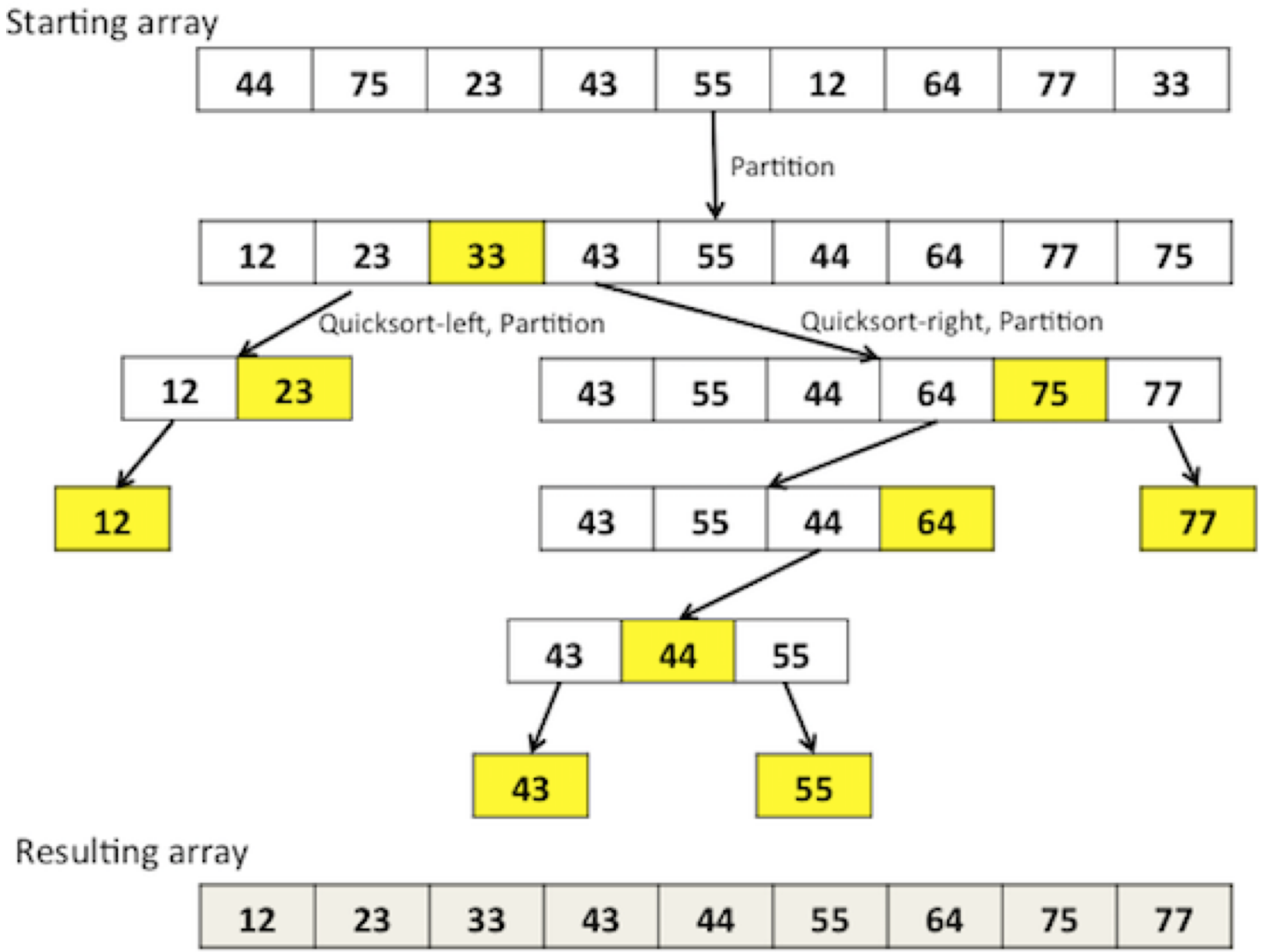
Bubble (Sinking, Ripple) Sort

- Main idea: Go through each element, and if it's out of order with its neighbor, swap them.
- If you can go through entire array with no swaps, you're done
- After i^{th} pass, at least last i positions are sorted and in final correct order.
- Worst case: $O(n^2)$
  - has a big constant
- Best case: O(n)
  - if already sorted

Merge Sort



- Divide-and-conquer
- Recursively sort ½ the set
- Then merge them together: O(n)
- O(n *lg* n)
- No real best or worst case
- Can be made to be in-place



Quicksort

1. Pick a pivot element

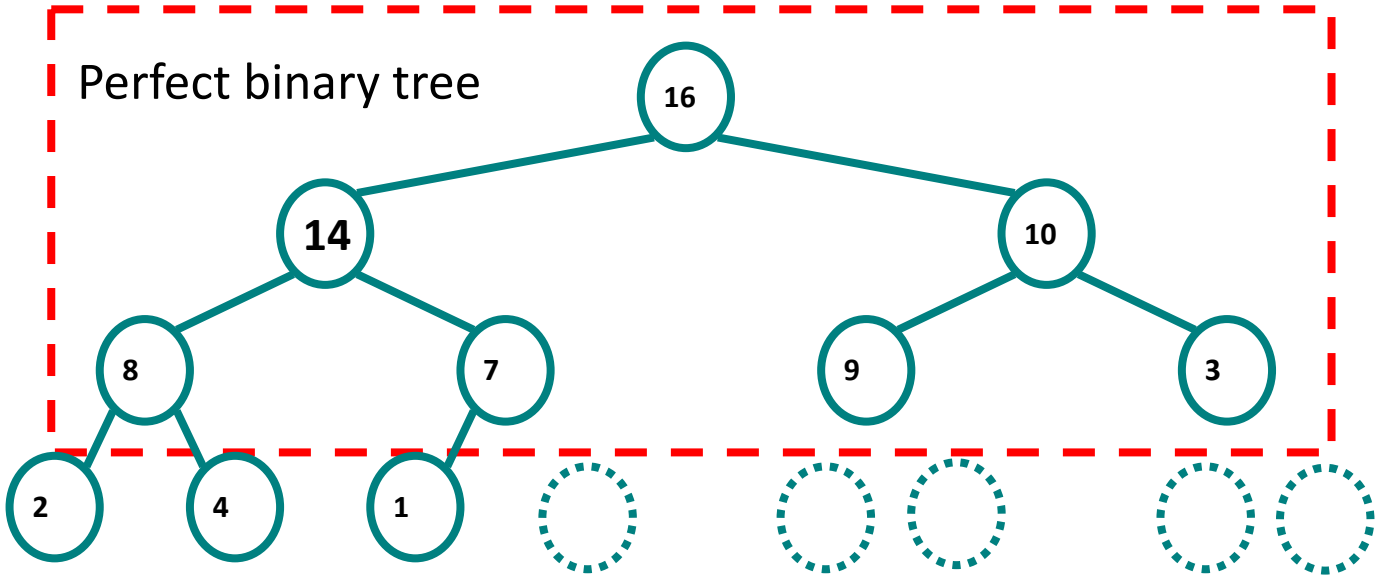
2. Put elements <pivot on the left

3. Put elements> pivot on right.

3. Sort the left and right

Heaps

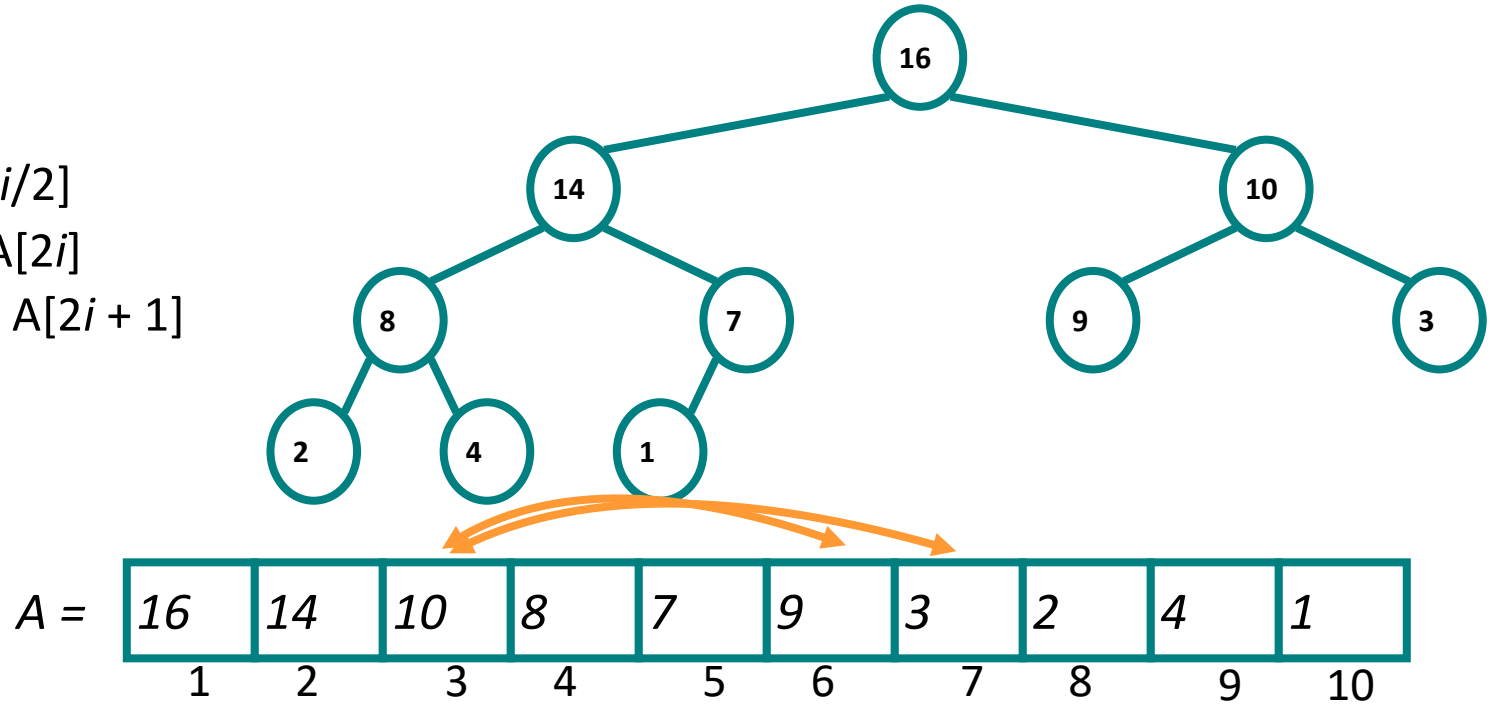
- A *heap* is a complete binary tree:



*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*

(Max) Heap Property

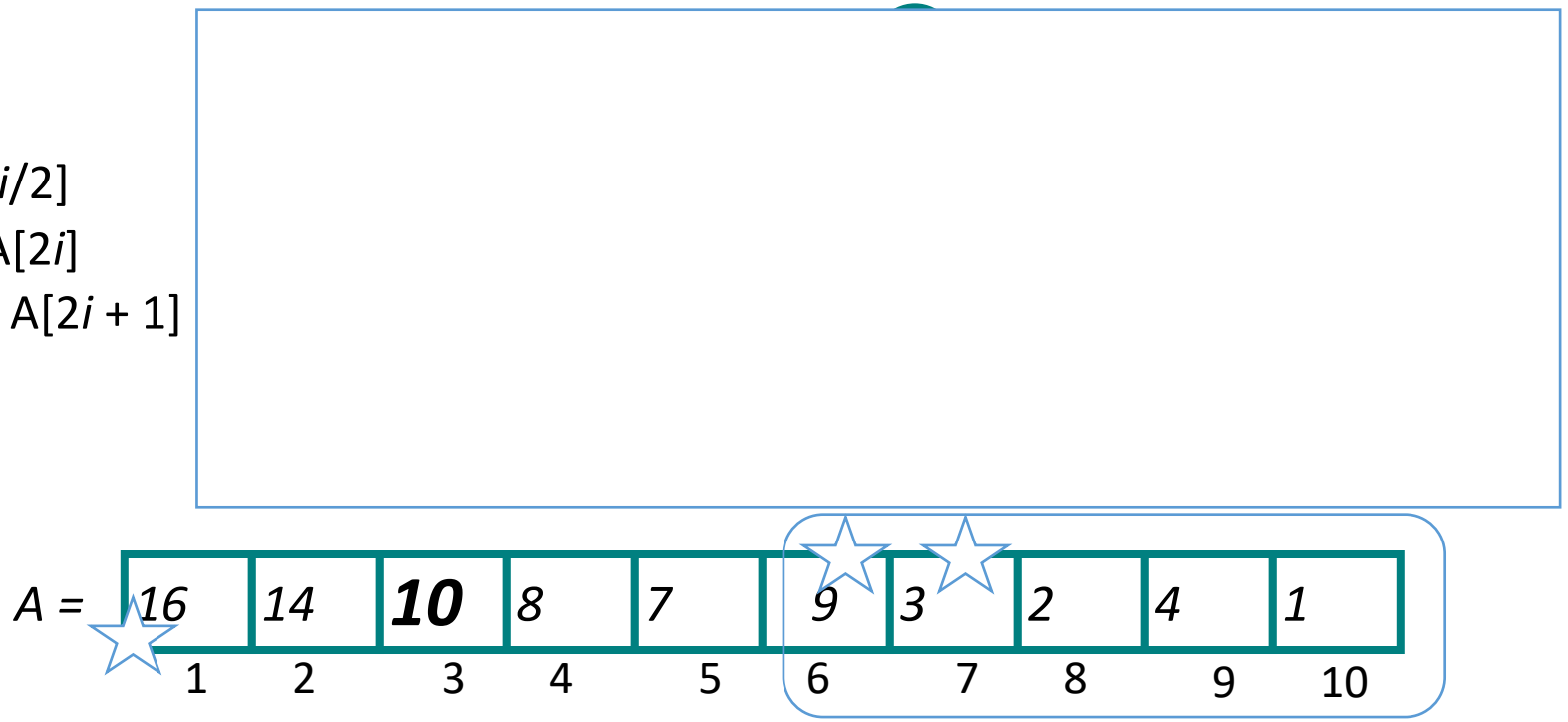
- Parent bigger than its children (and all of its descendants)
- Implemented as arrays:
  - Root=A[1]
  - Node *i* = A[*i*]
    - Parent = A[*i*/2]
    - Left child = A[2*i*]
    - Right child = A[2*i* + 1]



- Is {23, 17, 14, **6**, 13, 10, 1, **5**, **7**, 12} a heap?
  1 2 3 4 5 6 7 8 9 10

Heap Property

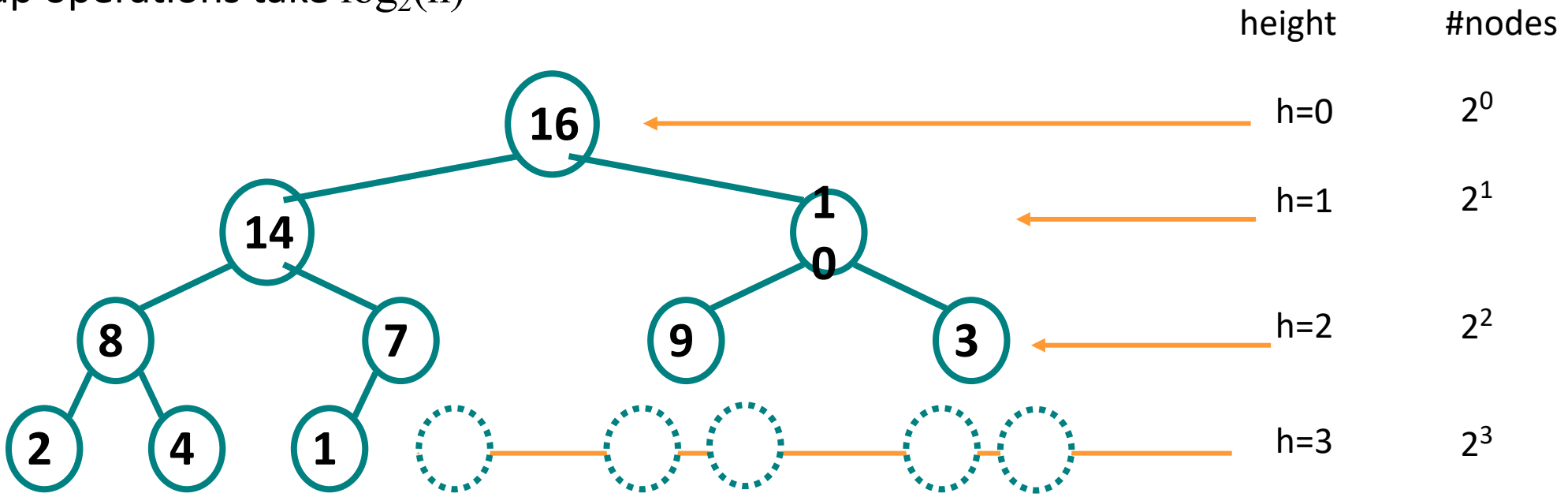
- Parent bigger than its children (and all of its descendants)
- Implemented as arrays:
  - Root=A[1]
  - Node *i* = A[*i*]
    - Parent = A[*i*/2]
    - Left child = A[2*i*]
    - Right child = A[2*i* + 1]



- Leaves are indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$

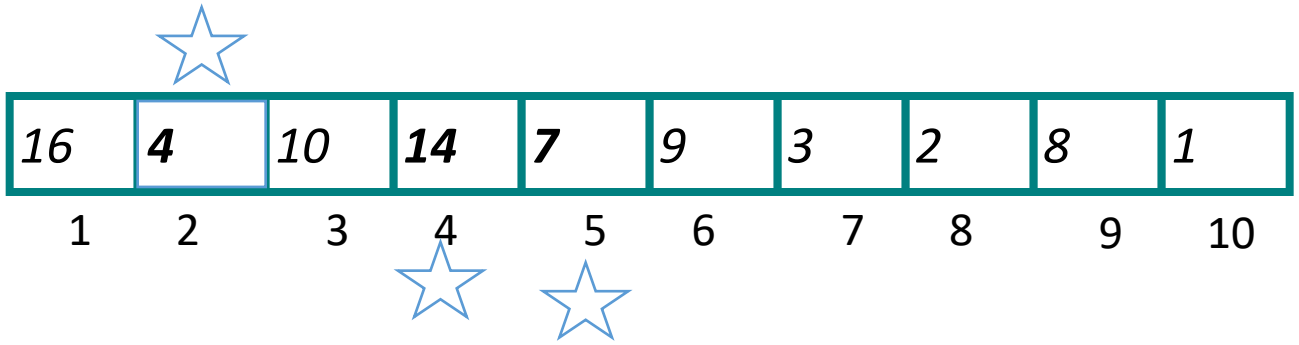
Heap Height

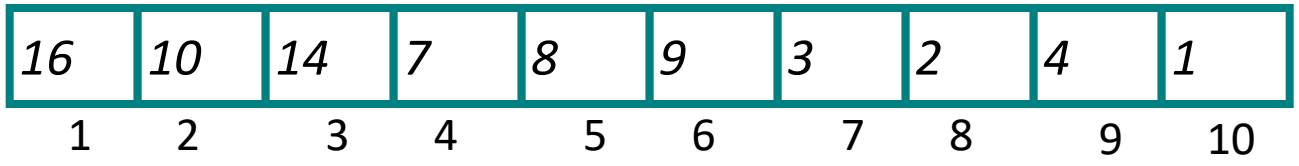
- *height of an n-element heap =* $\log_2 n$. Why?
  - heap operations take $\log_2(n)$



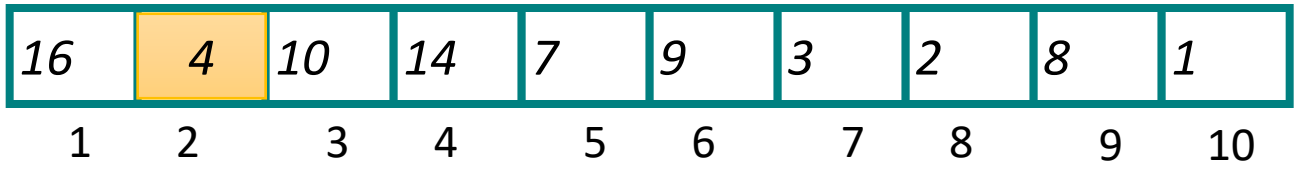
  - What are the max and min number of elements in a heap of height *h*?
    - between (2^h) and $(2^{h+1}-1)$

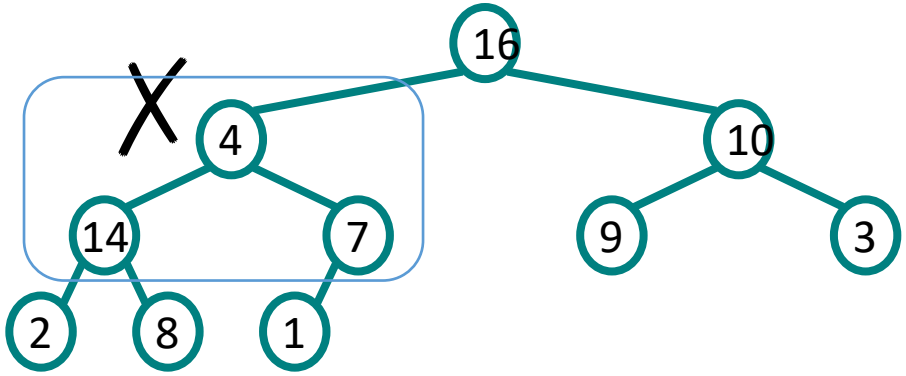
Are they heaps?



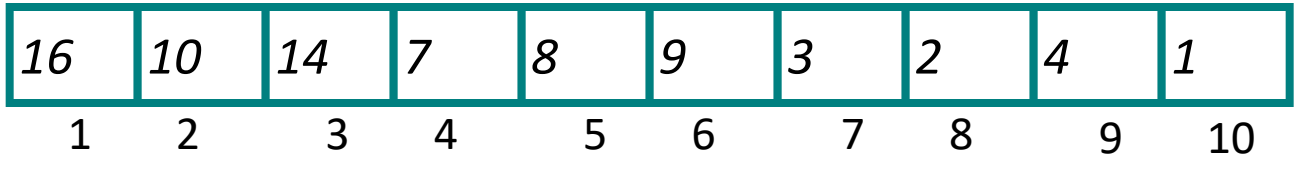


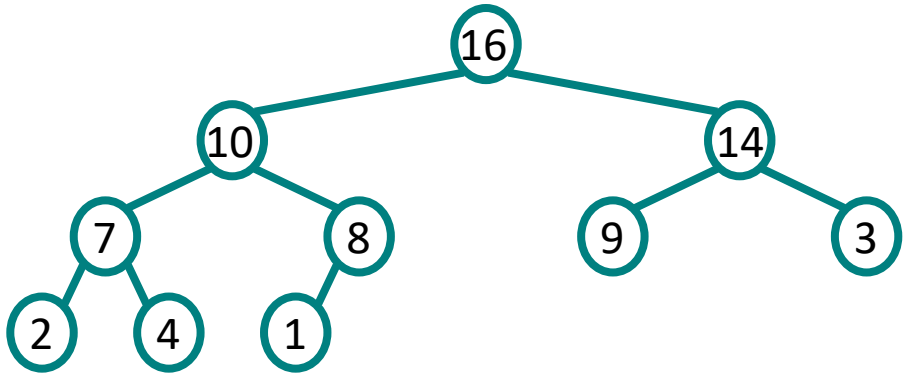
Are they heaps?





H





Violation to heap property: a node has value < one of its children
How to find that?
How to resolve that?

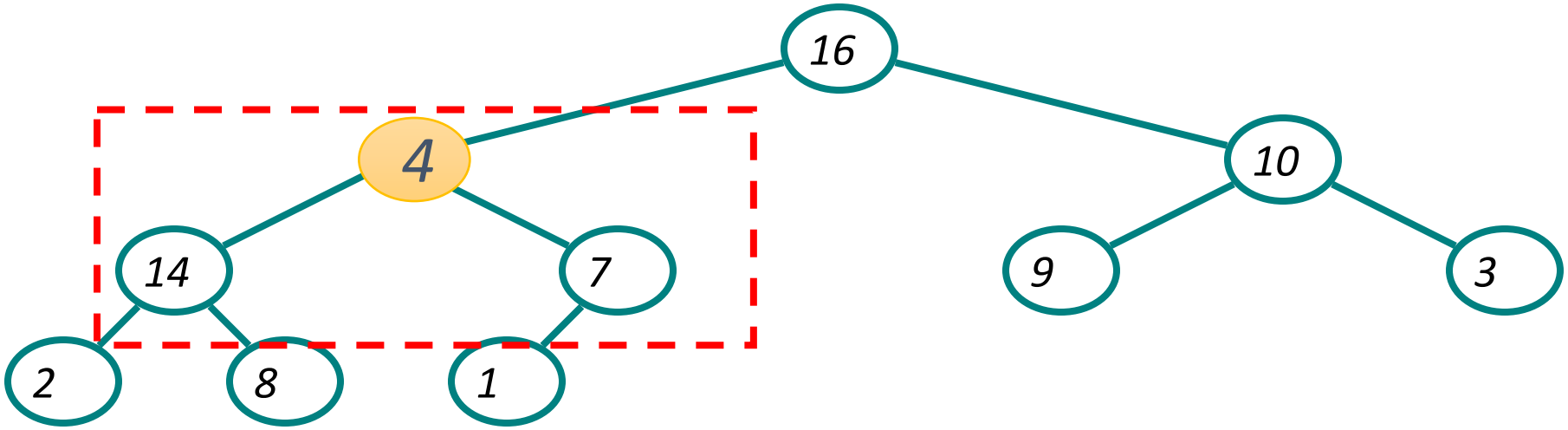
Heap Operations: Heapify()

- **`Heapify()`**: maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Problem: The subtree rooted at *i* may violate the heap property
  - Action: let the value of the parent node "sift down" so subtree at *i* satisfies the heap property
    - Fix up the relationship between i, l, and r recursively

Heap Operations: Heapify()

```
Heapify(A, i)
```

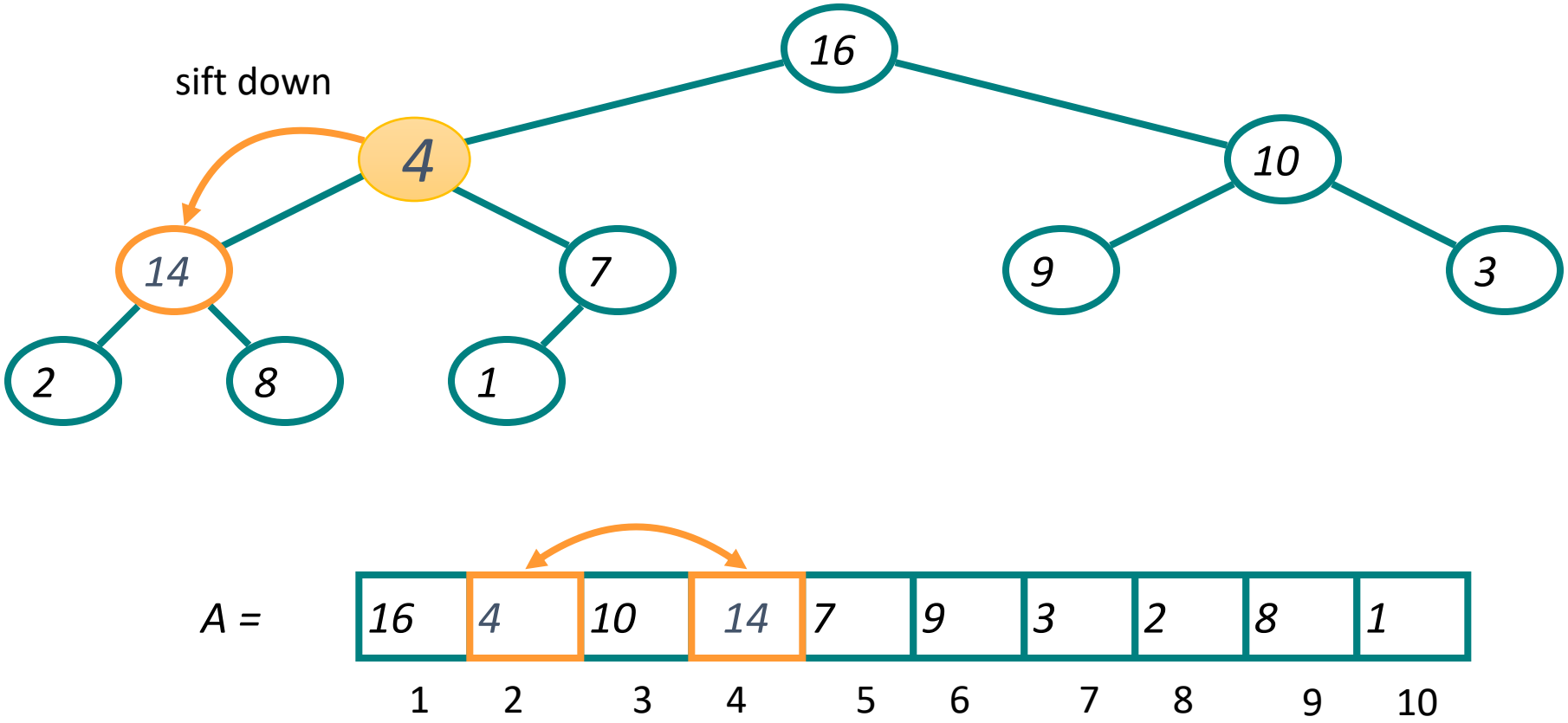
Heapify() Example



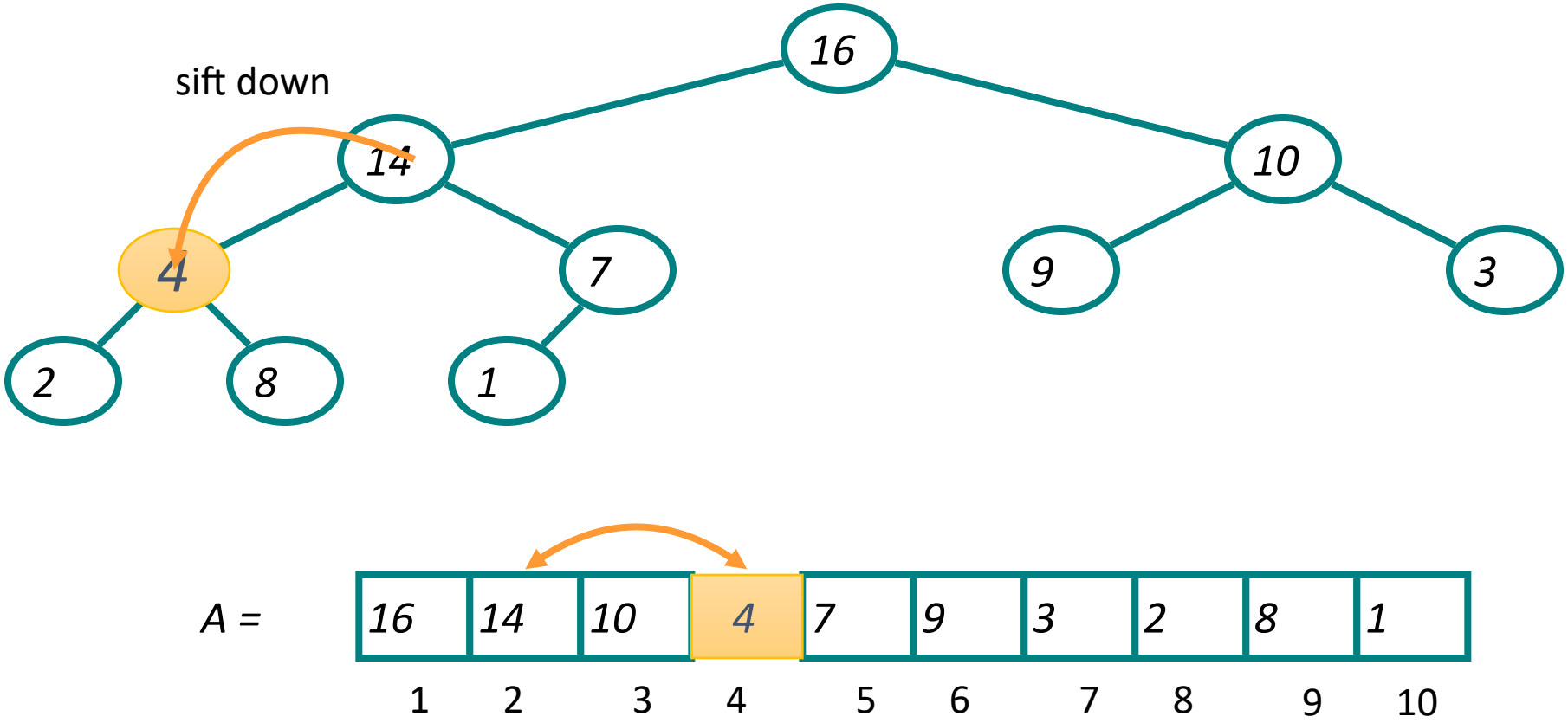
A =

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

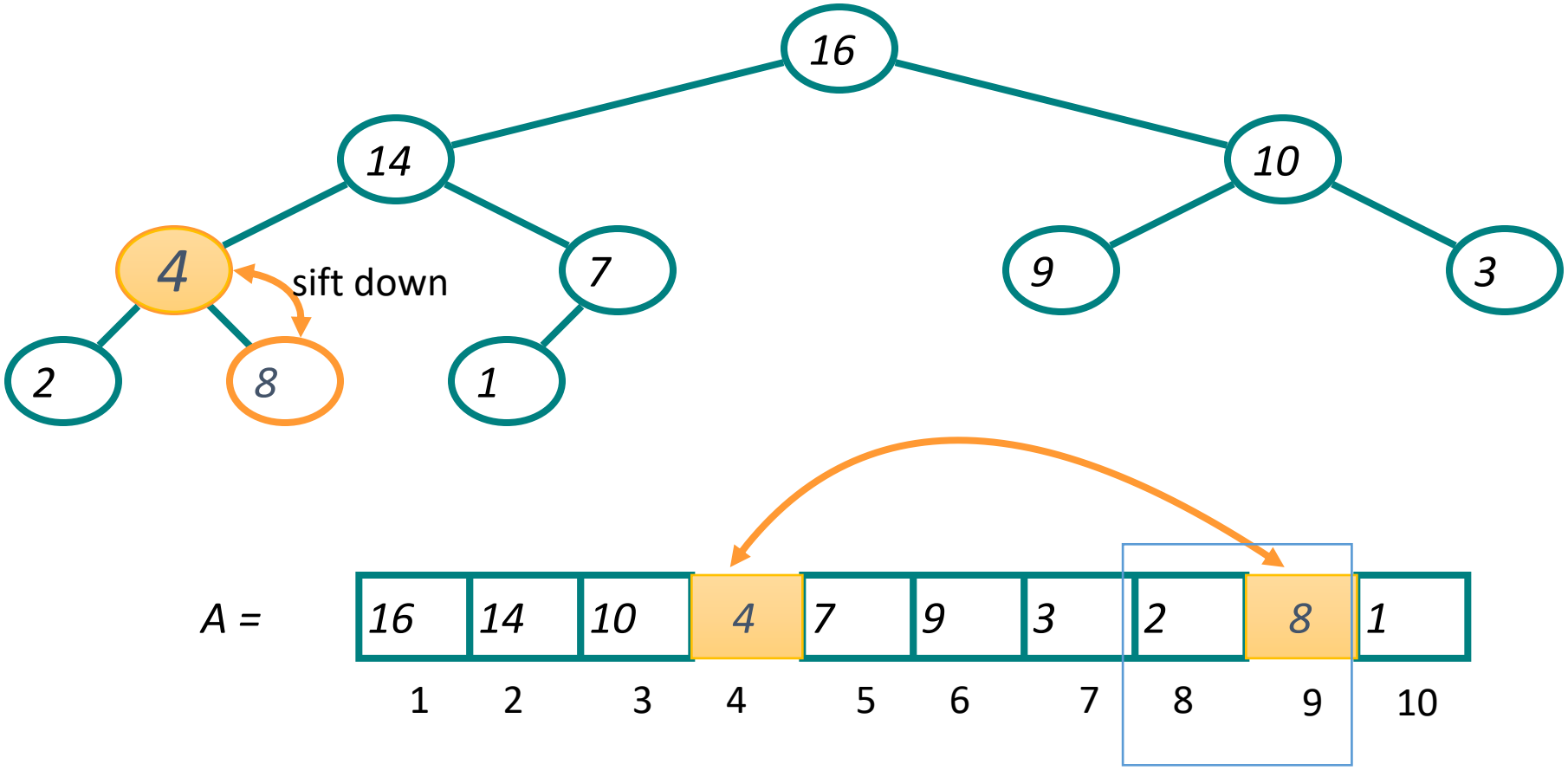
Heapify() Example



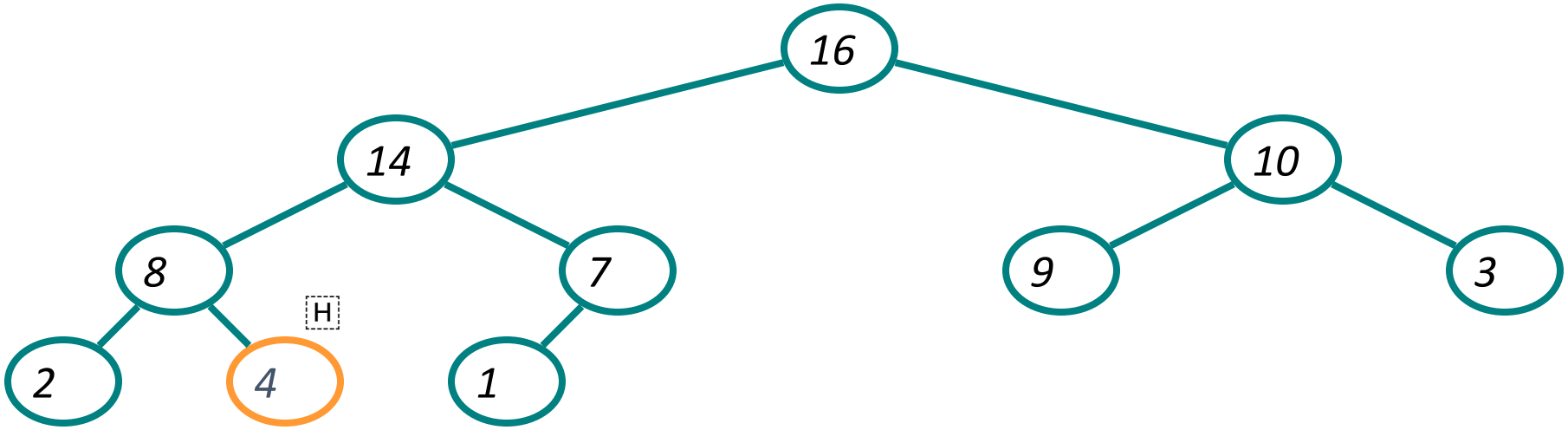
Heapify() Example

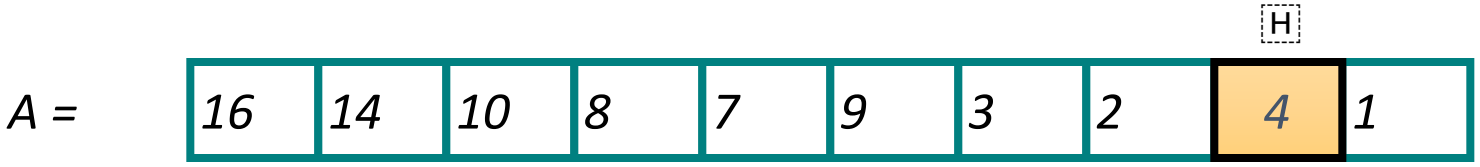


Heapify() Example



Heapify() Example





Analyzing Heapify(): Sift (Percolating) down

- The *running time of* ***`Heapify()`*** *?*
  - Fixing up relationships between *i*, *l*, and *r* takes $\Theta(1)$ time
  - *How many times can* ***`Heapify()`*** *recursively call itself?*

- $T(n) = O(\lg n)$, which is height of a complete binary tree with n leaves

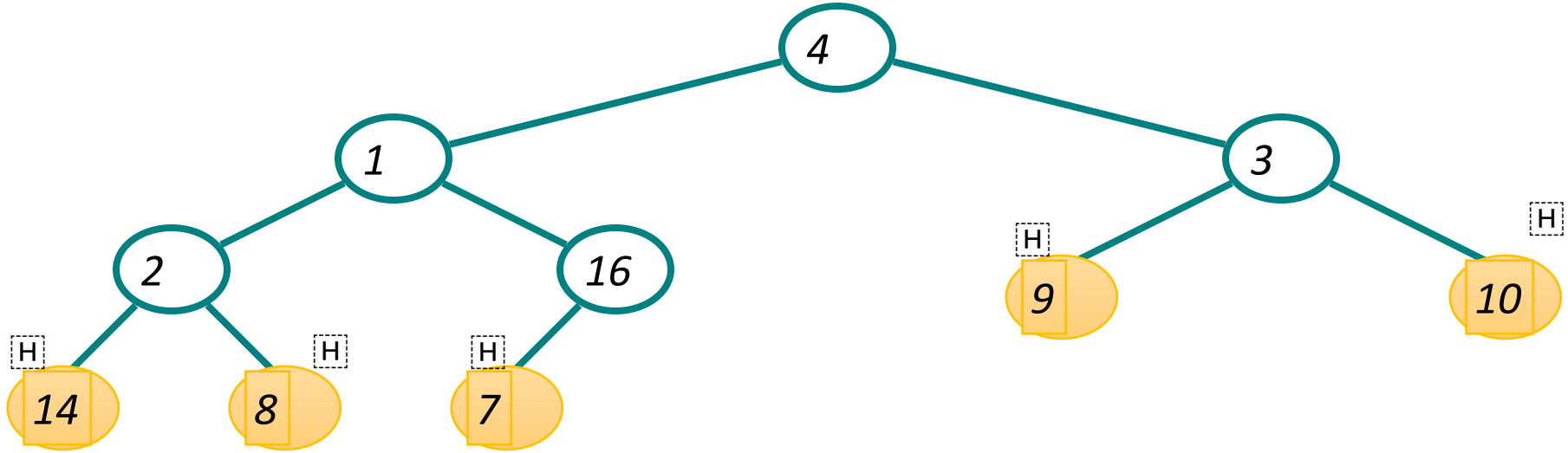
Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - *The nodes after n/2 are all leaves, which are single-node heap*
  - *Iterate* array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

```
BuildHeap(A){ // A is an unsorted array
 A.heap_size = length(A);
 for (i = A.length/2 downto 1)
     Heapify(A, i); // make A a heap
}
```

BuildHeap() Example

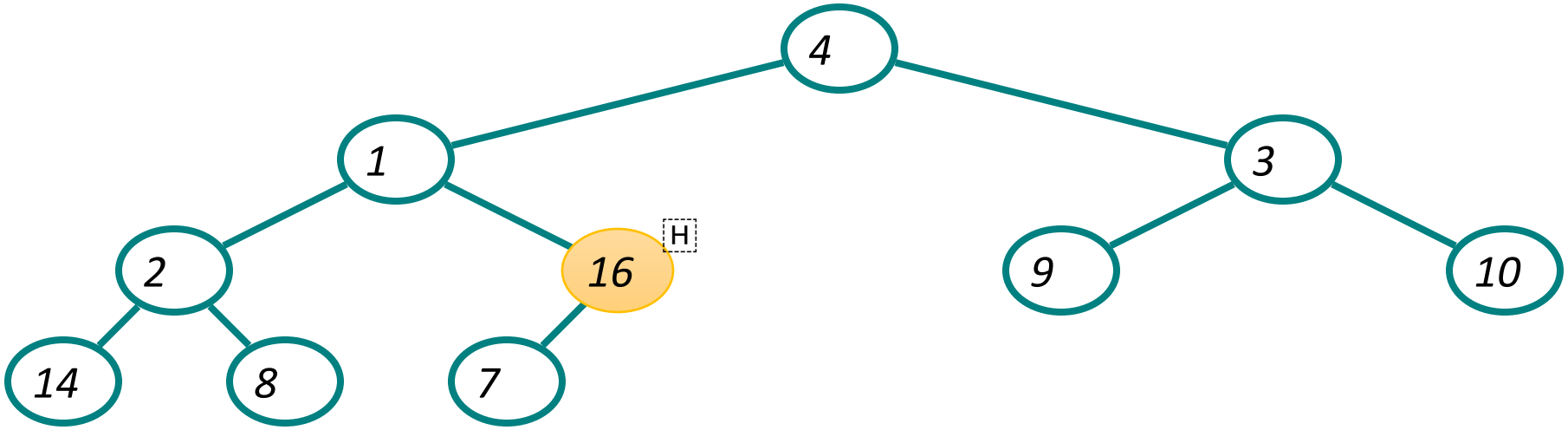
- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



A =

4	1	3	2	16	9	10	14	8	7

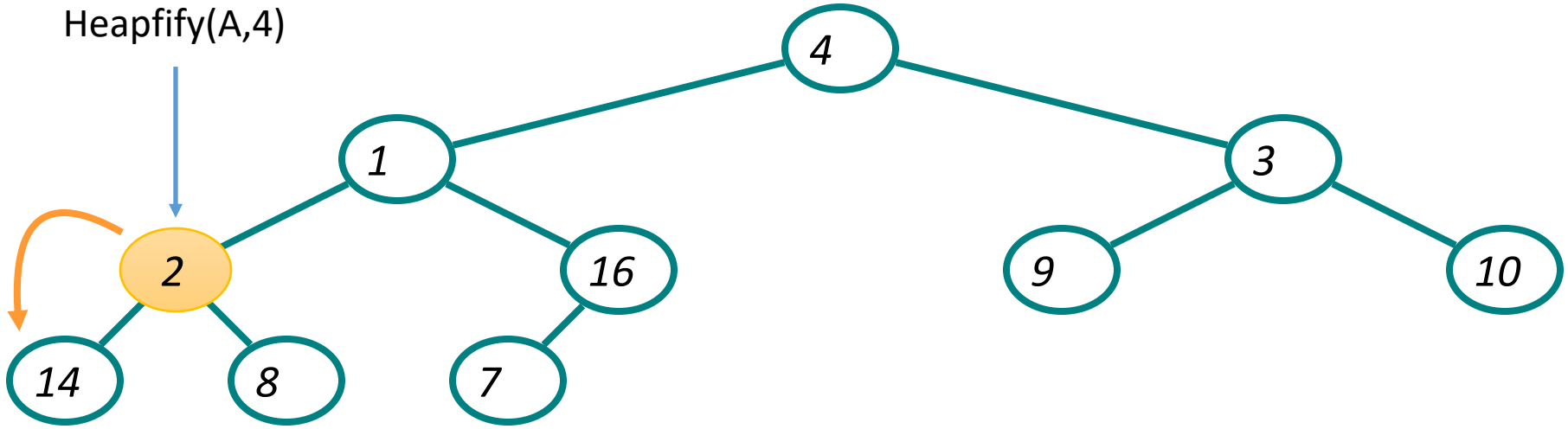
BuildHeap() Example

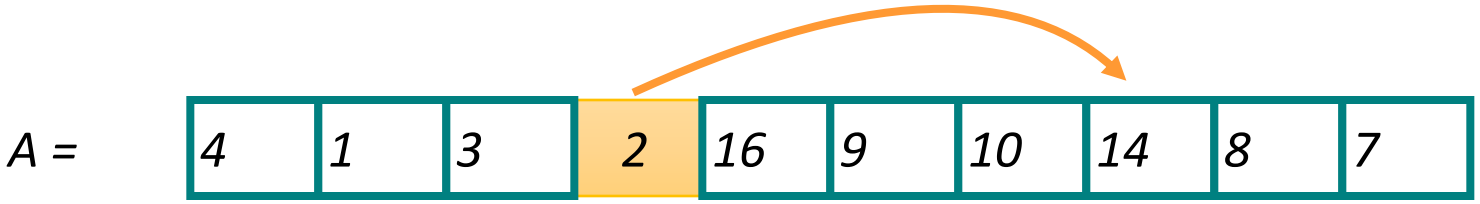


A =

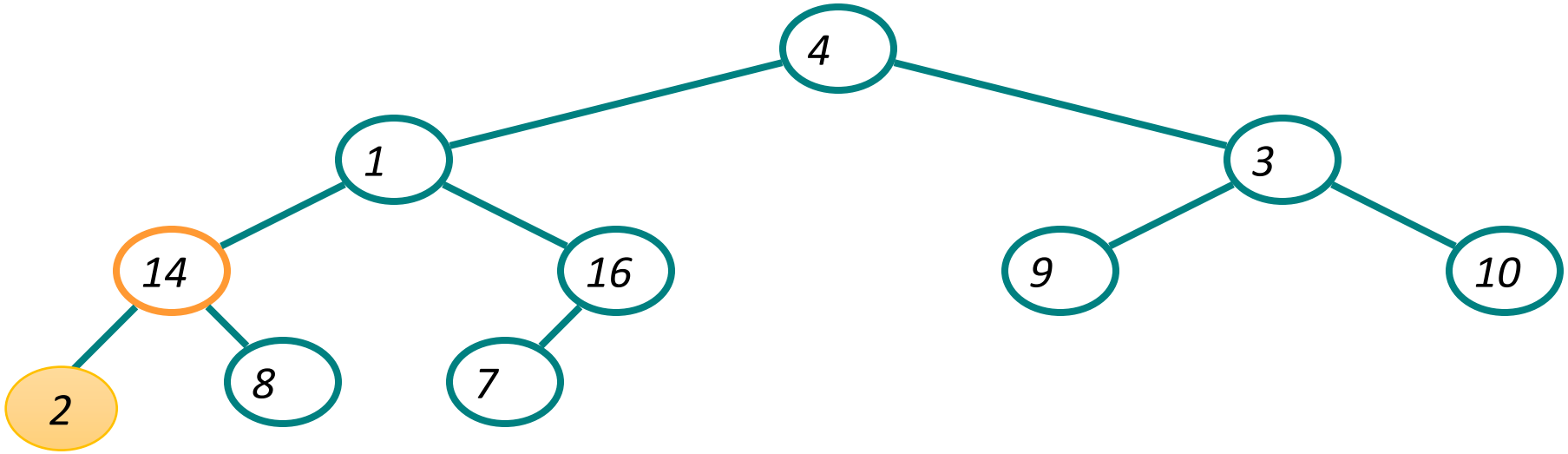
4	1	3	2	16	9	10	14	8	7

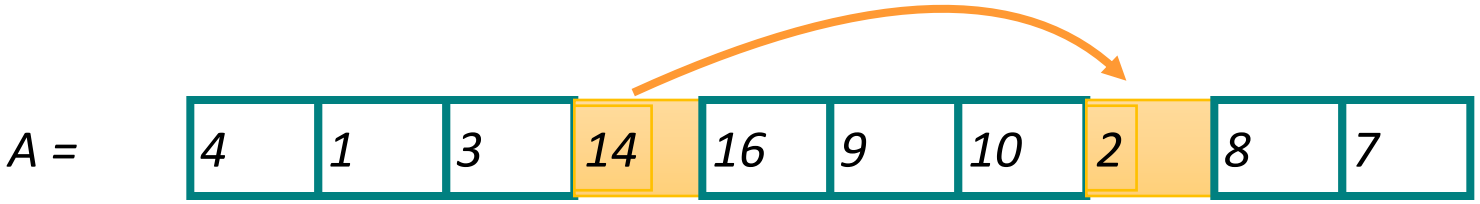
BuildHeap() Example



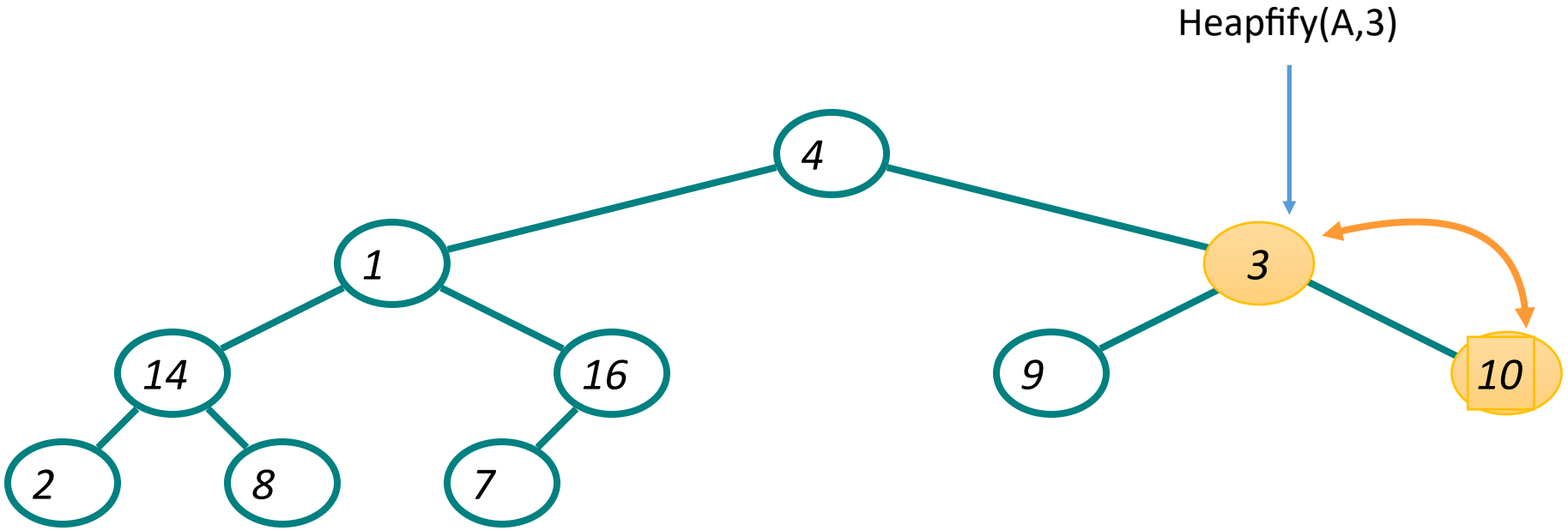


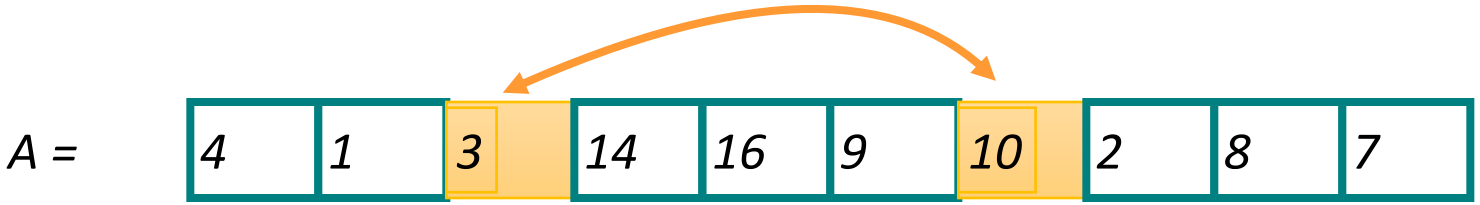
BuildHeap() Example



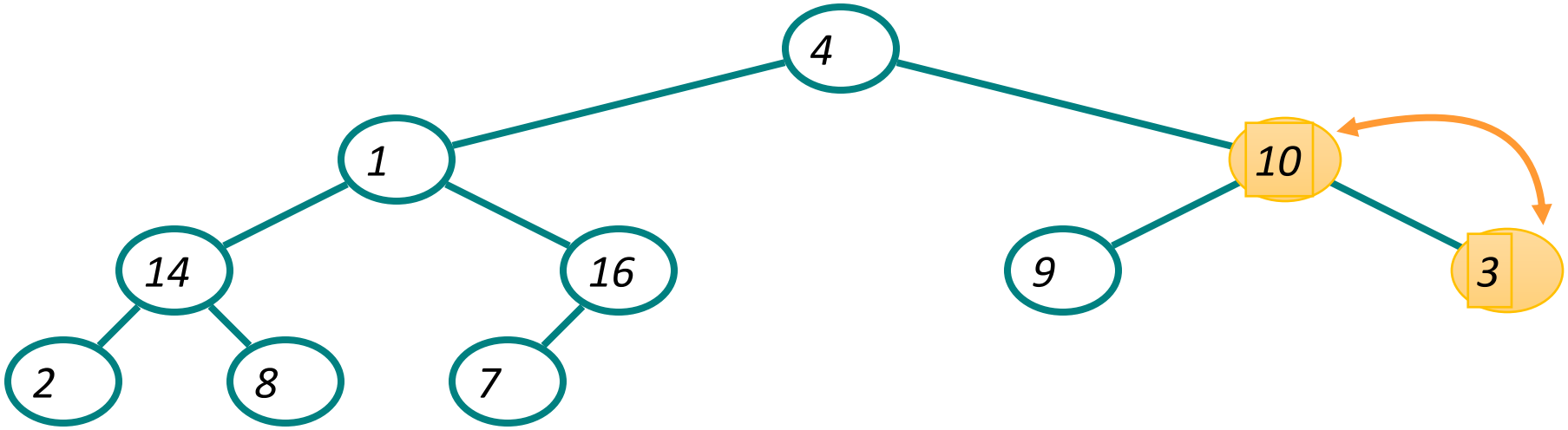


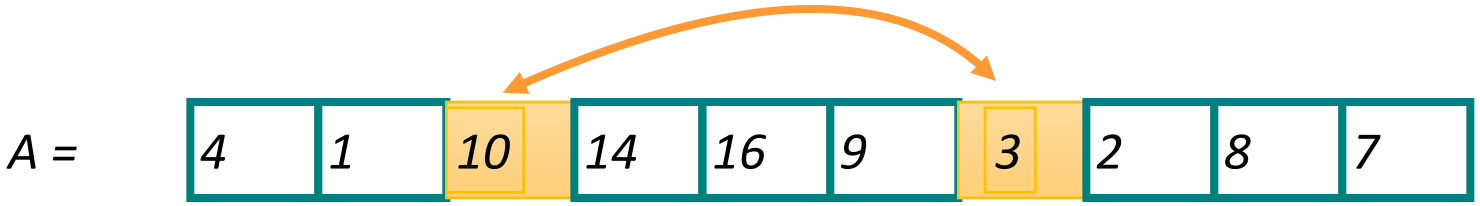
BuildHeap() Example



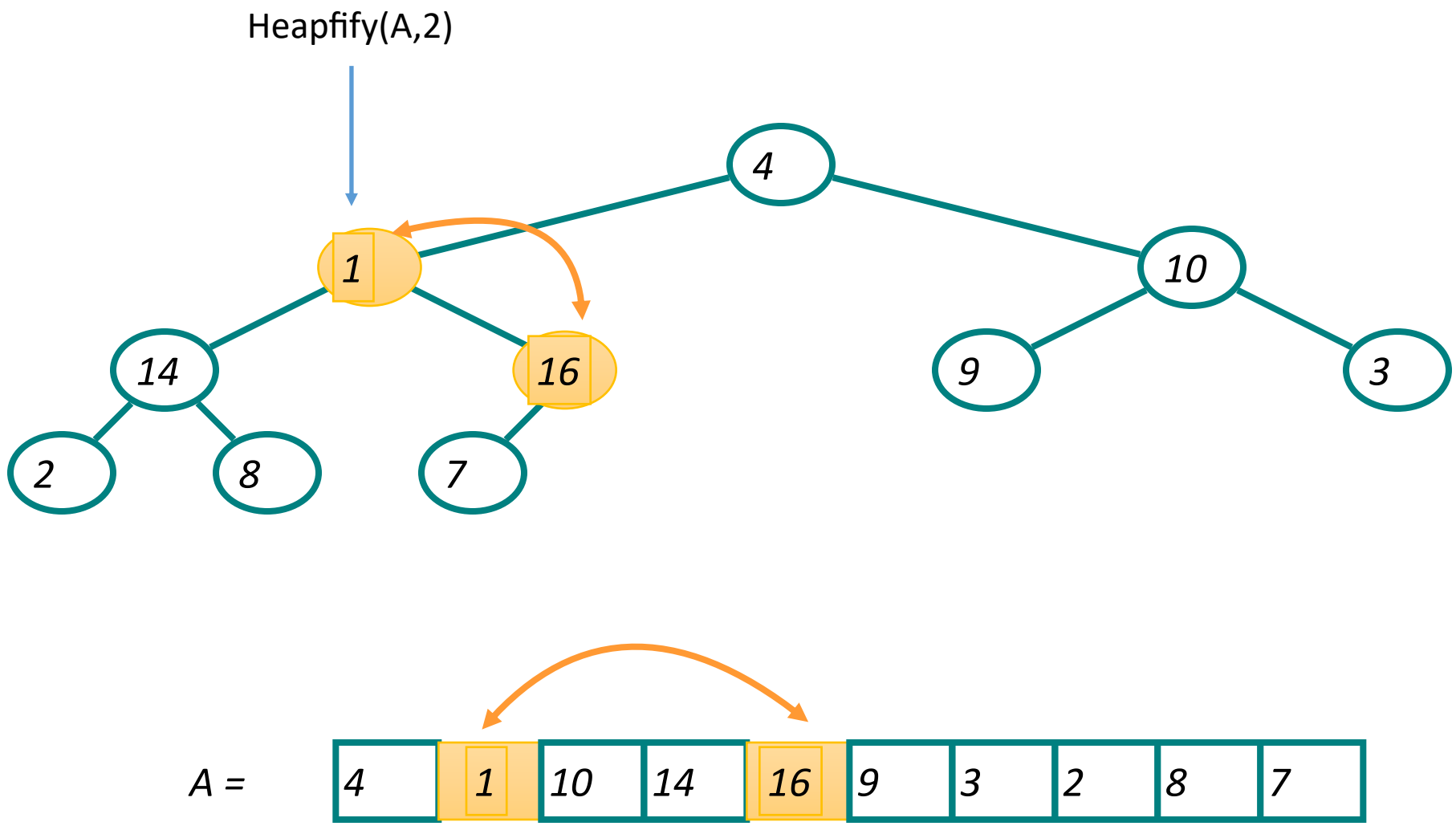


BuildHeap() Example

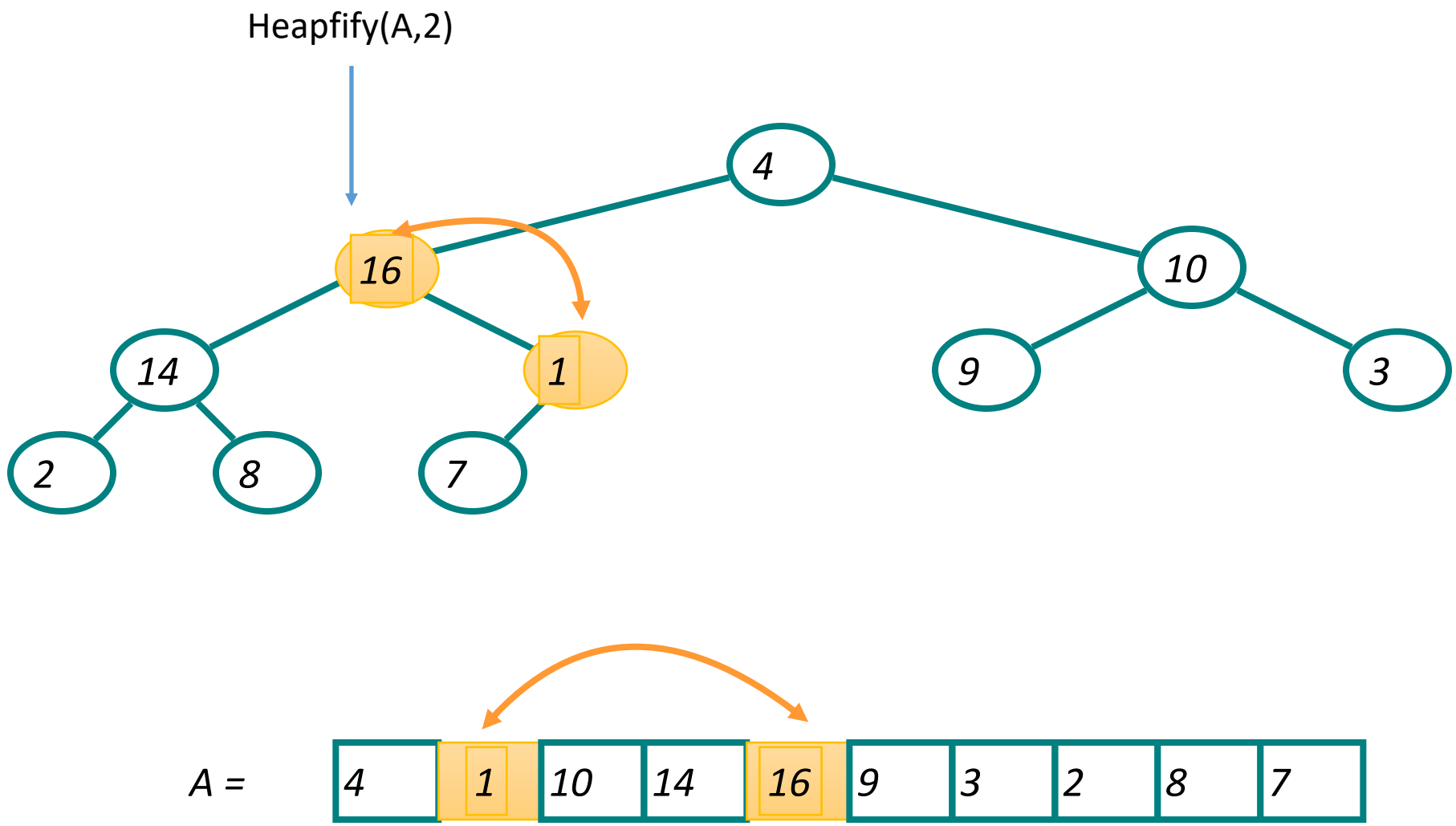




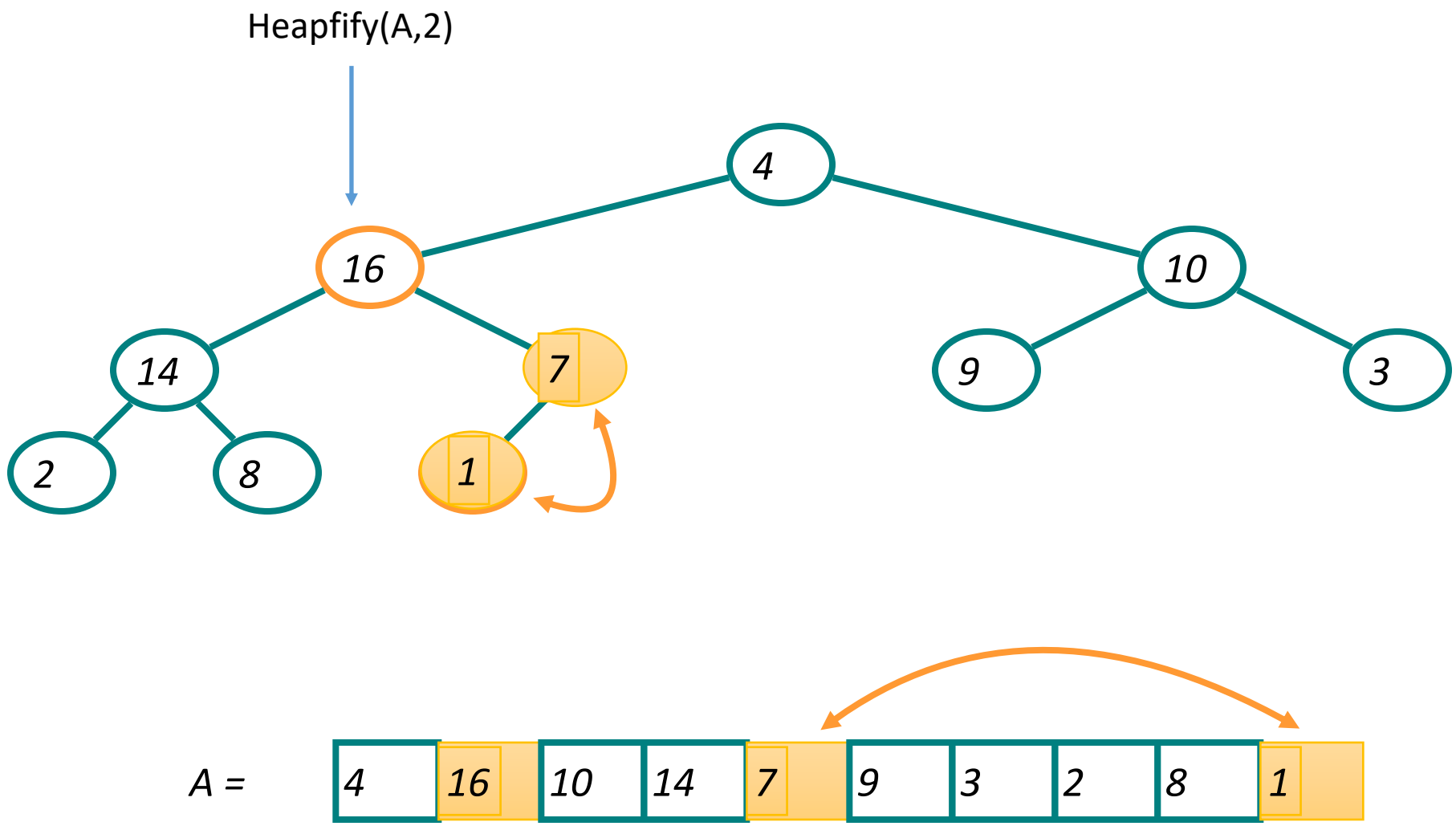
BuildHeap() Example



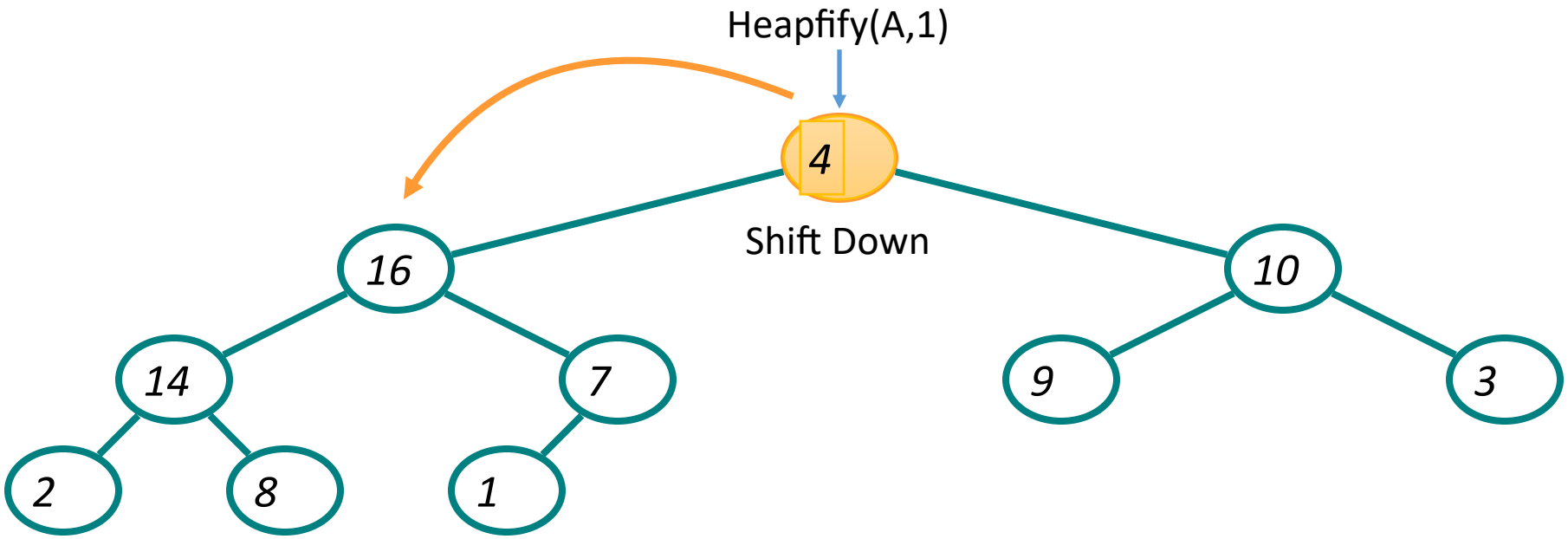
BuildHeap() Example



BuildHeap() Example



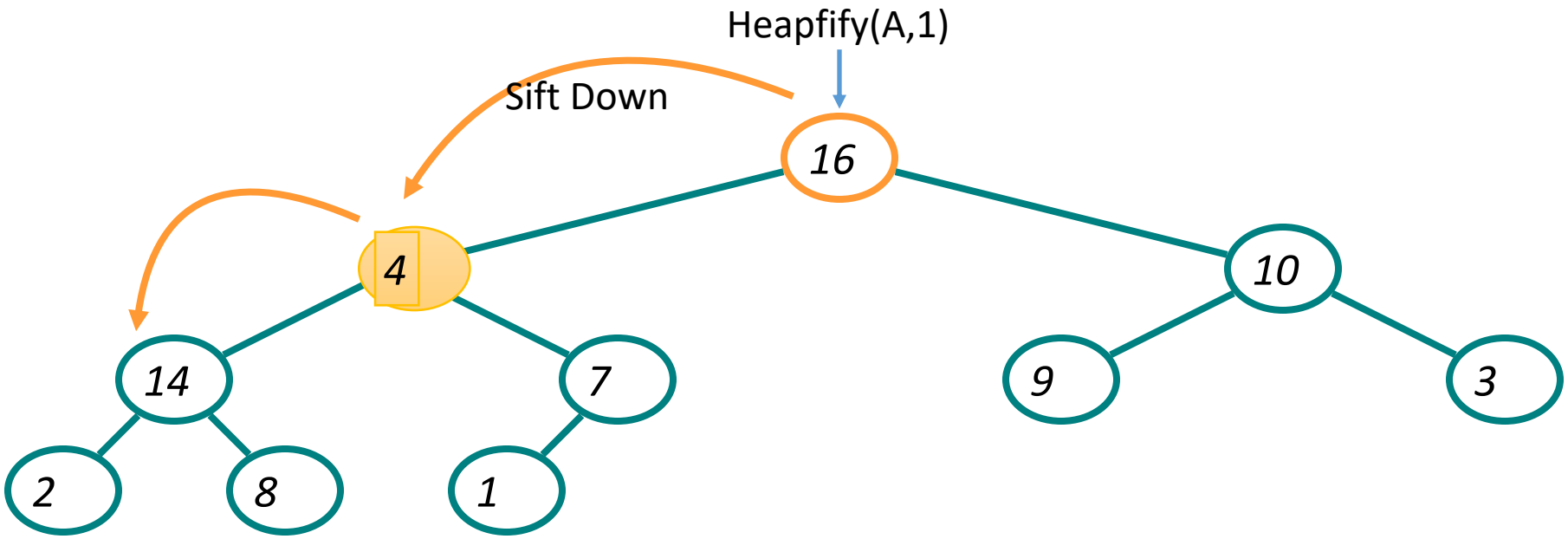
BuildHeap() Example

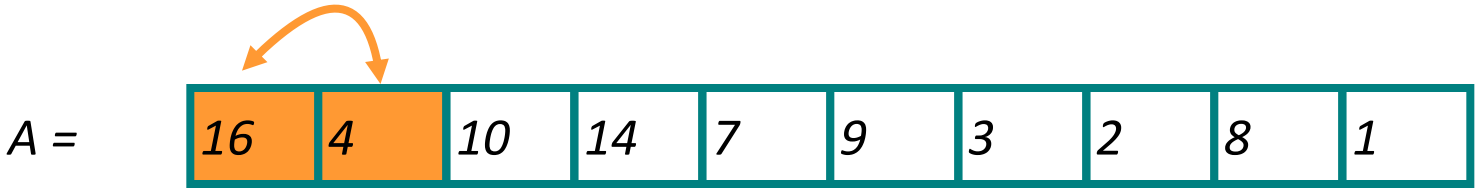


A =

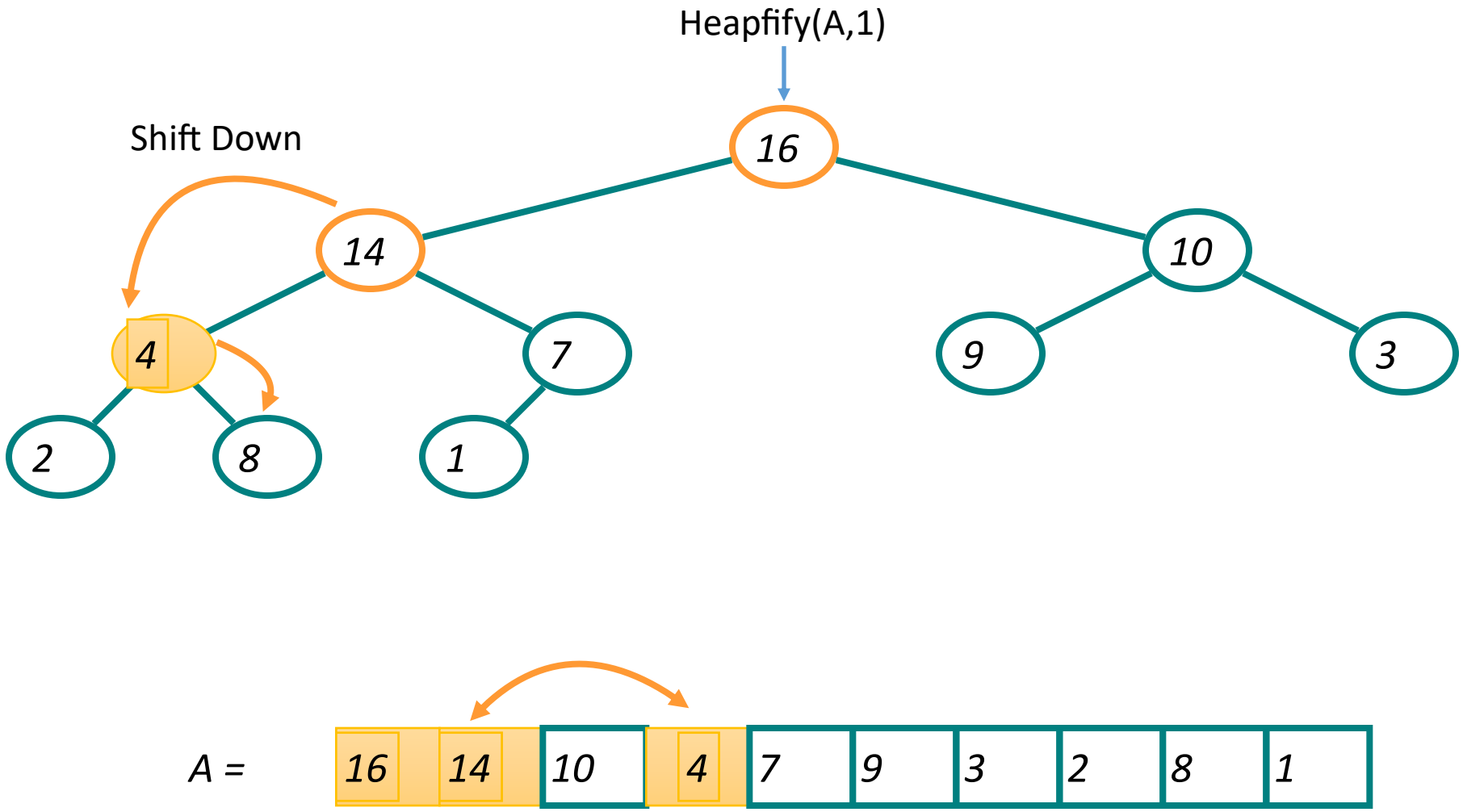
4	16	10	14	7	9	3	2	8	1

BuildHeap() Example

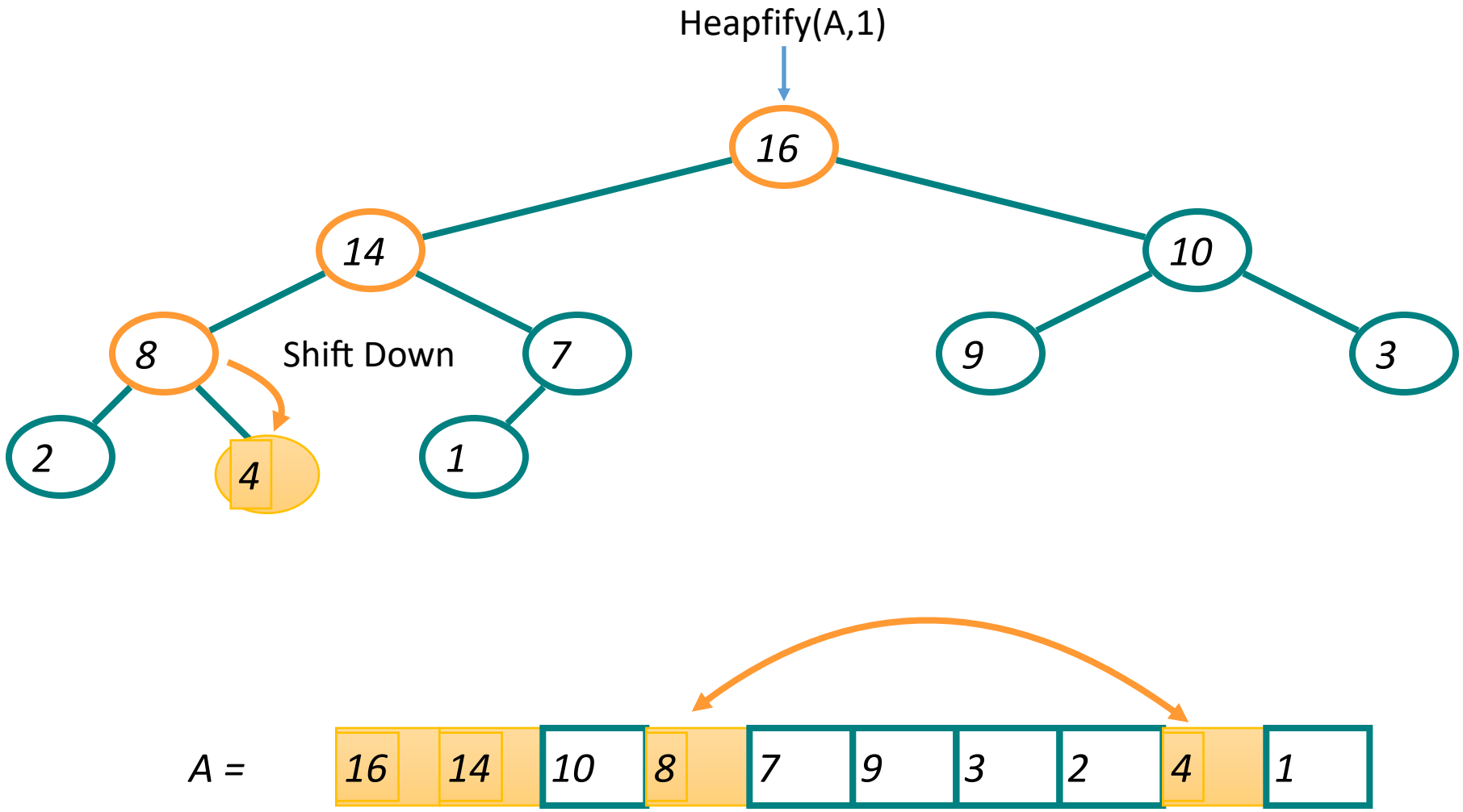




BuildHeap() Example

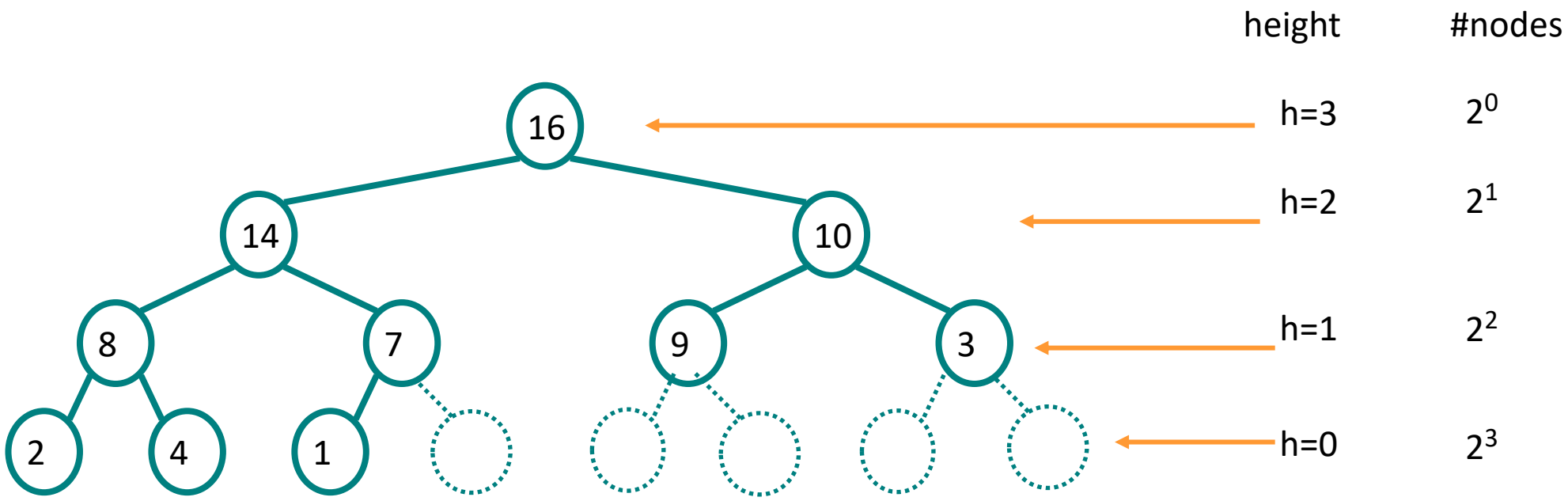


BuildHeap() Example



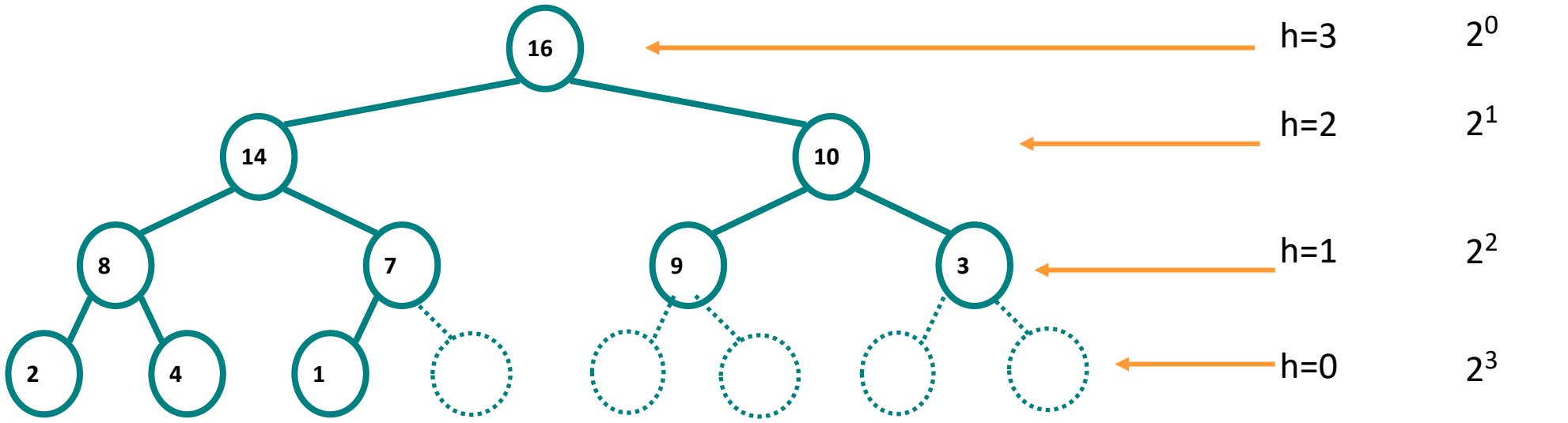
Analyzing BuildHeap()

- Each call to **`Heapify()`** takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *n*)
  - A tighter bound is O(*n*)
    - *How can this be? Is there a flaw in the above reasoning?*



A complete binary of n nodes has

- At most n/2 leaves (height h=0)
- At most n/4 nodes of height h=1
- At most n/8 nodes of height h=2
- At most n/16 nodes of height h=3
- …
- At most $n/2^{h+1}$ nodes of height h



$$T(n) \le \sum_{h=1}^{\lg n} \left\lceil \frac{n}{2^{h+1}} \right\rceil h \le \sum_{h=1}^{\lg n} \frac{nh}{2^h} = n \sum_{h=1}^{\lg n} \frac{h}{2^h} \le 2n$$

- Therefore T(n) = O(n)

Idea of heap sort

HeapSort(A[1..n])

Build a heap from A // O(n)

For $i = n$ down to 1

Retrieve largest element from heap // O(log n)

Put element at end of A

Reduce heap size by one

end

Heapsort

- Given **`BuildHeap()`**, an **in-place** sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling **`Heapify()`**
  - Repeat, always swapping A[1] for A[A.heap_size]

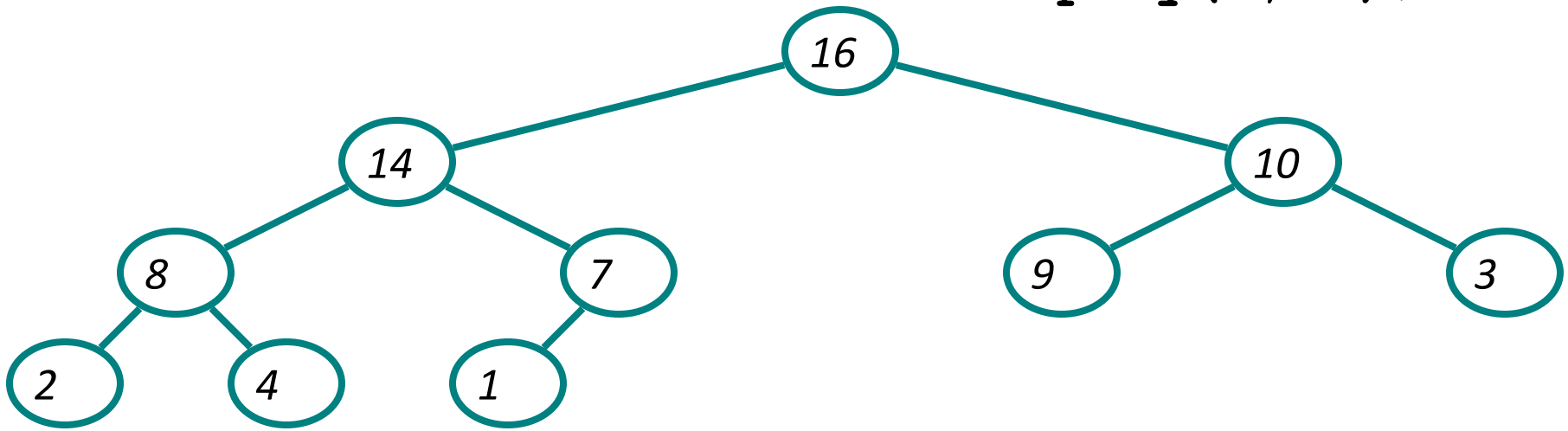
Heapsort

```
Heapsort(A)
{
      BuildHeap(A);
      for (i = length(A) downto 2)
      {
            Swap(A[1], A[i]);
            A.heap_size -= 1;
            Heapify(A, 1);
      }
}
```

Heapsort

- First: build a heap in O(n)

```
Heapsort(A):
  BuildHeap(A);
  for (i = length(A) downto 2):
      Swap(A[1], A[i]);
      A.heap_size -= 1;
      Heapify(A, 1);
```



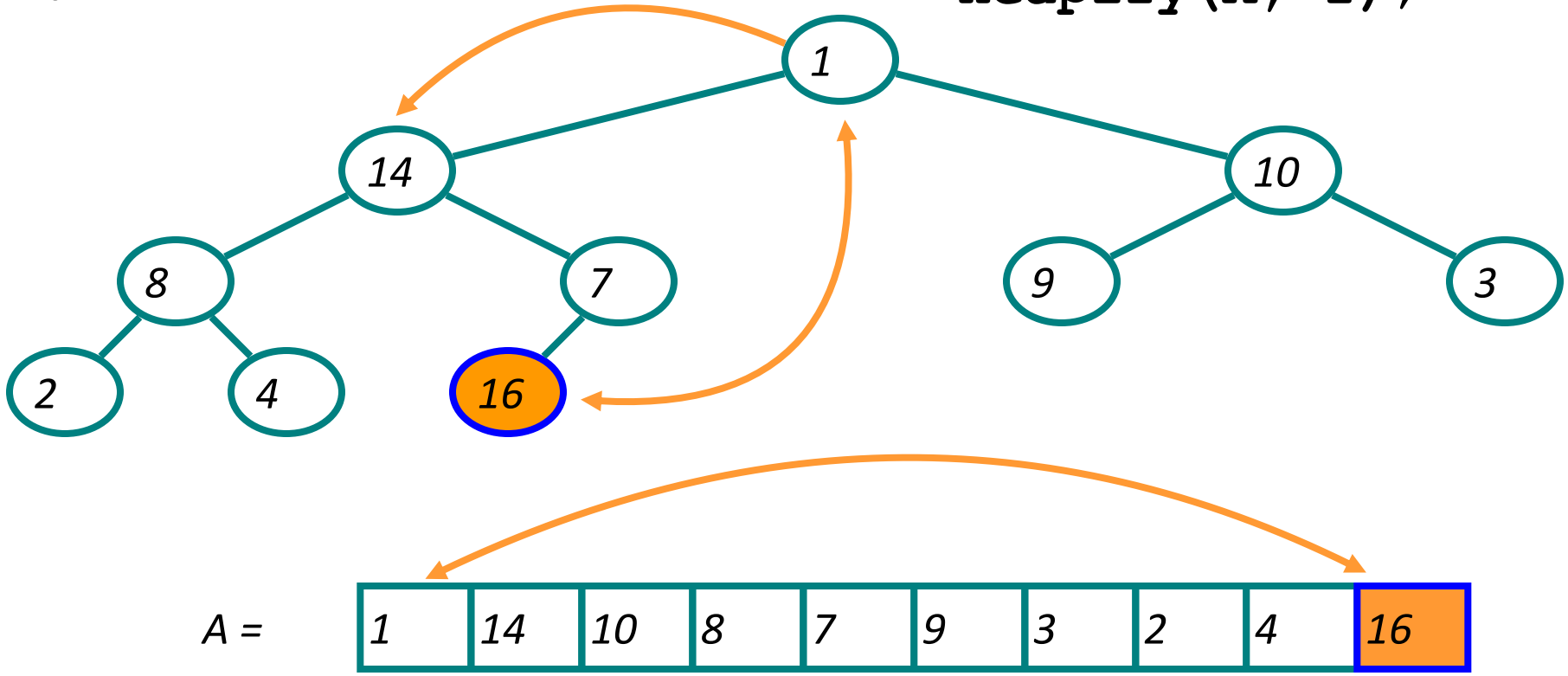
A =

16	14	10	8	7	9	3	2	4	1

Heapsort

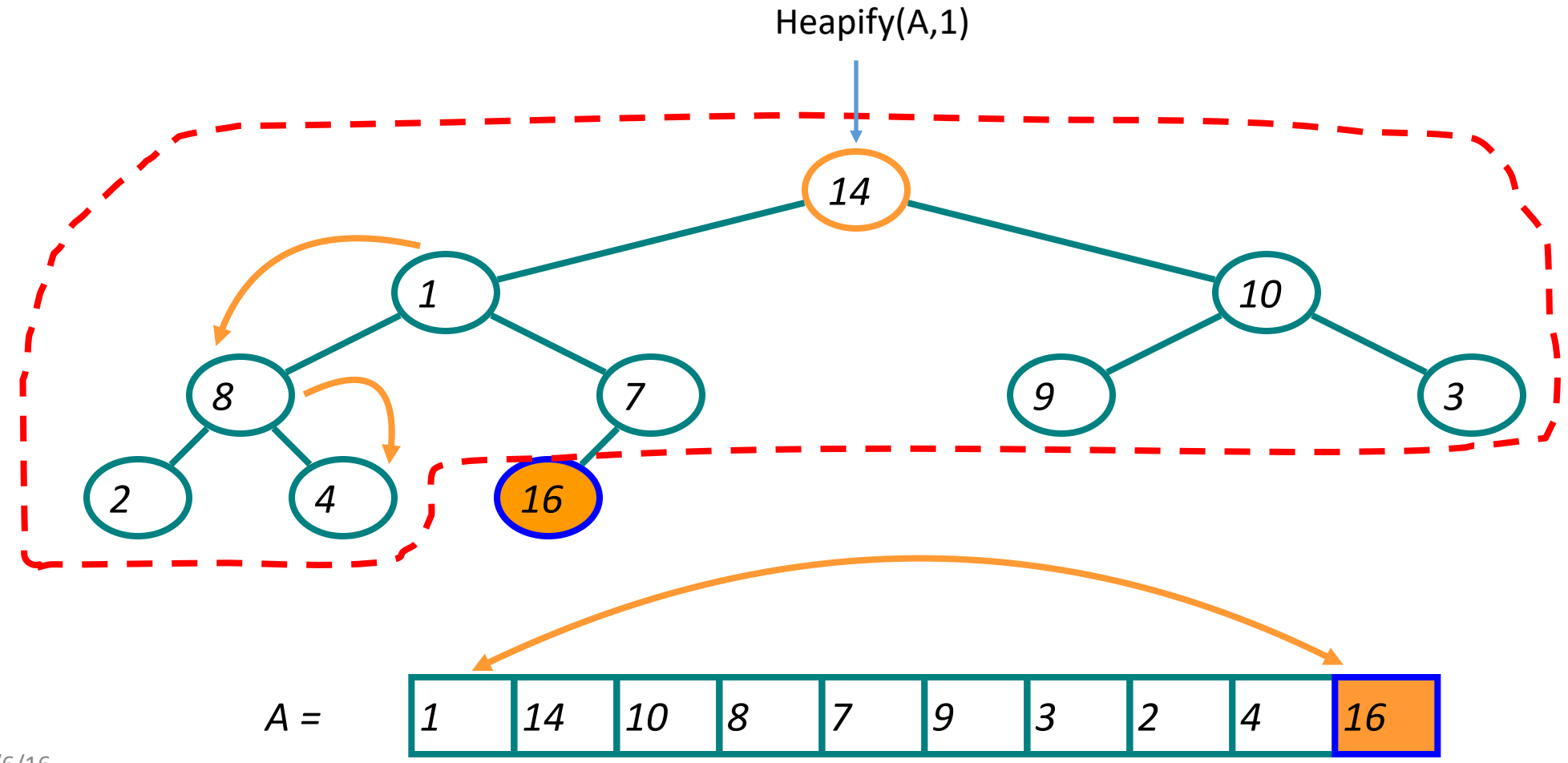
```
Heapsort(A):
  BuildHeap(A);
  for (i = length(A) downto 2):
      Swap(A[1], A[i]);
      A.heap_size -= 1;
      Heapify(A, 1);
```

- Swap last and first



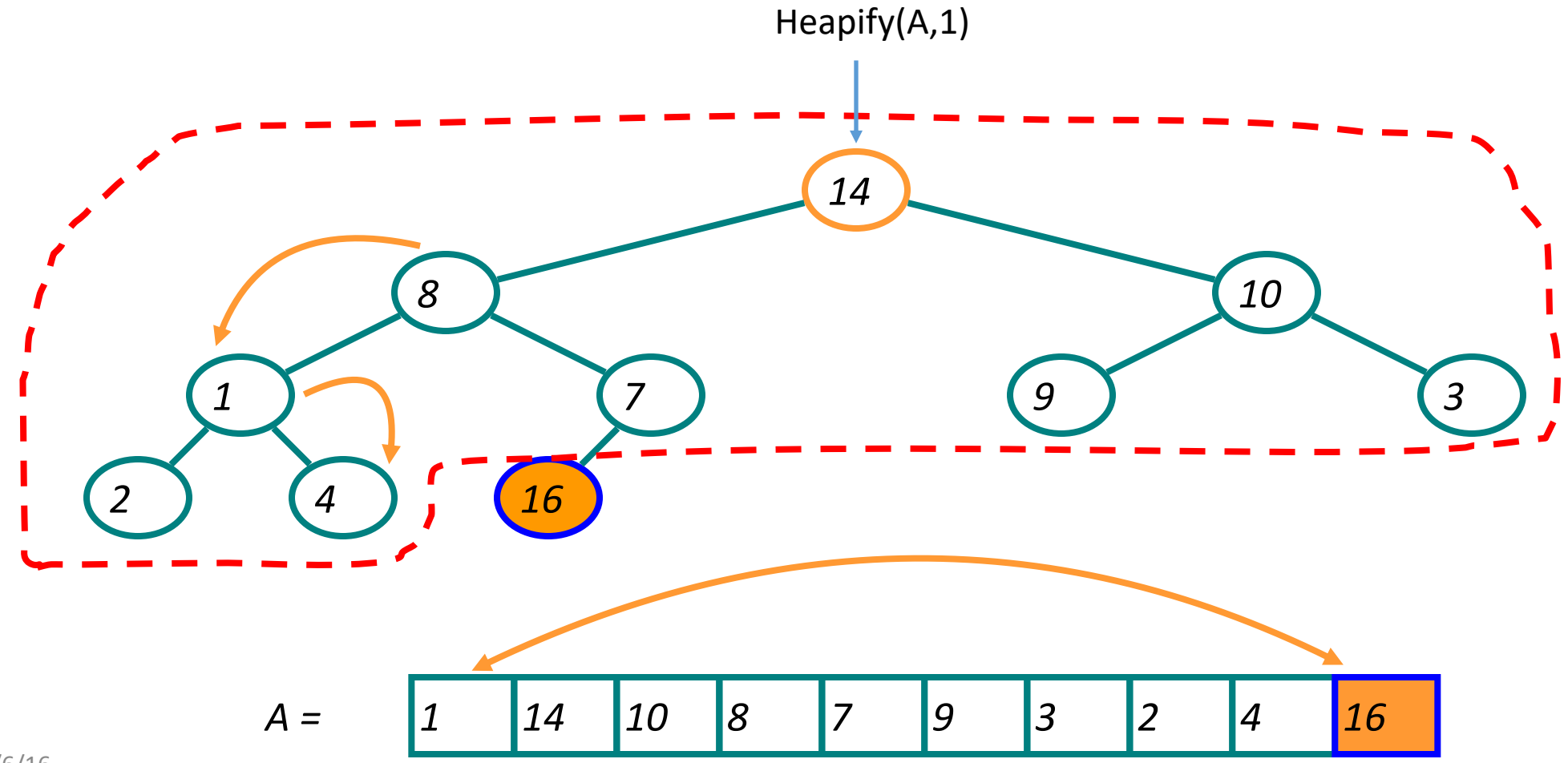
Heapsort Example

- Last element sorted



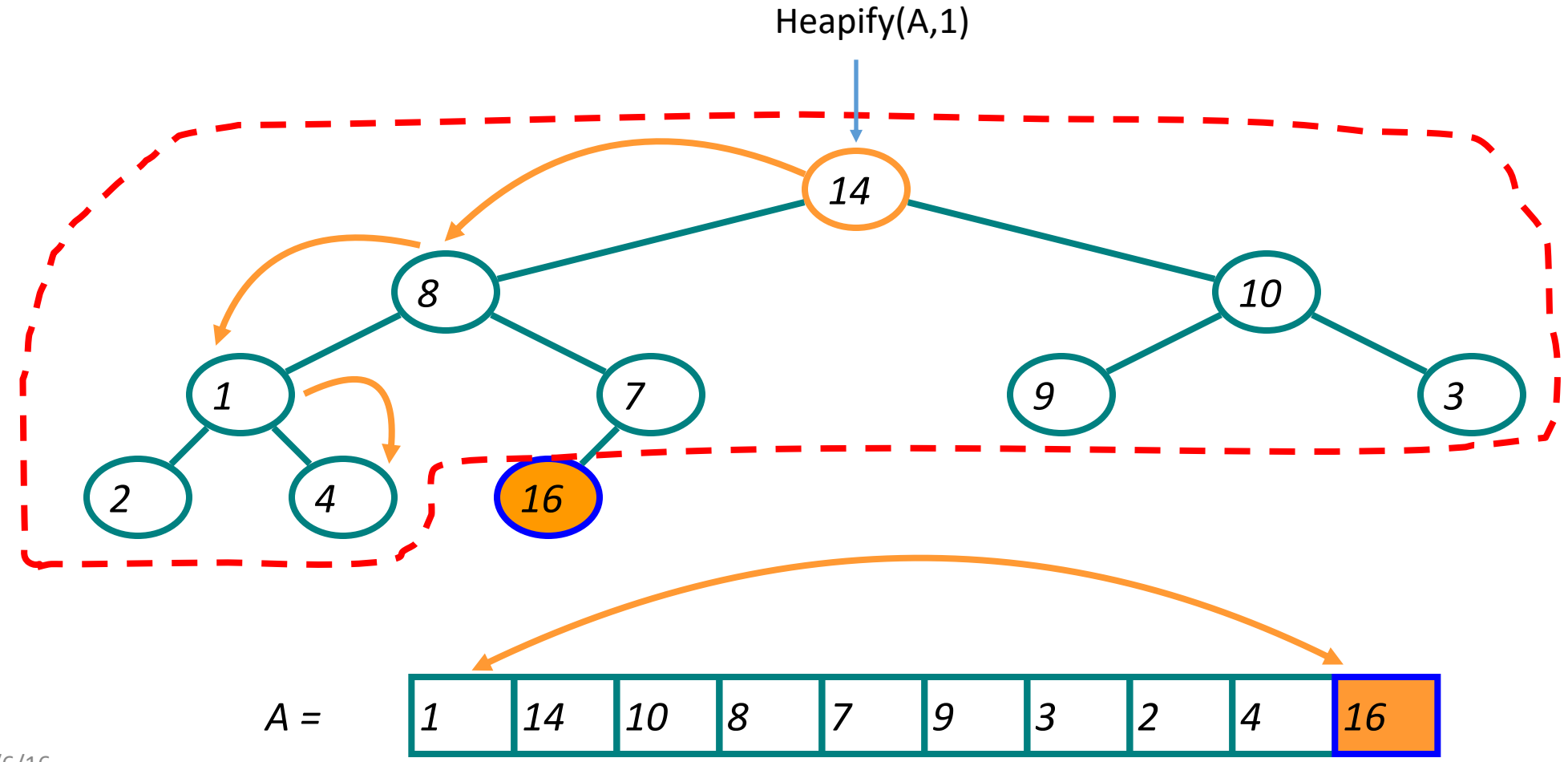
Heapsort Example

- Last element sorted



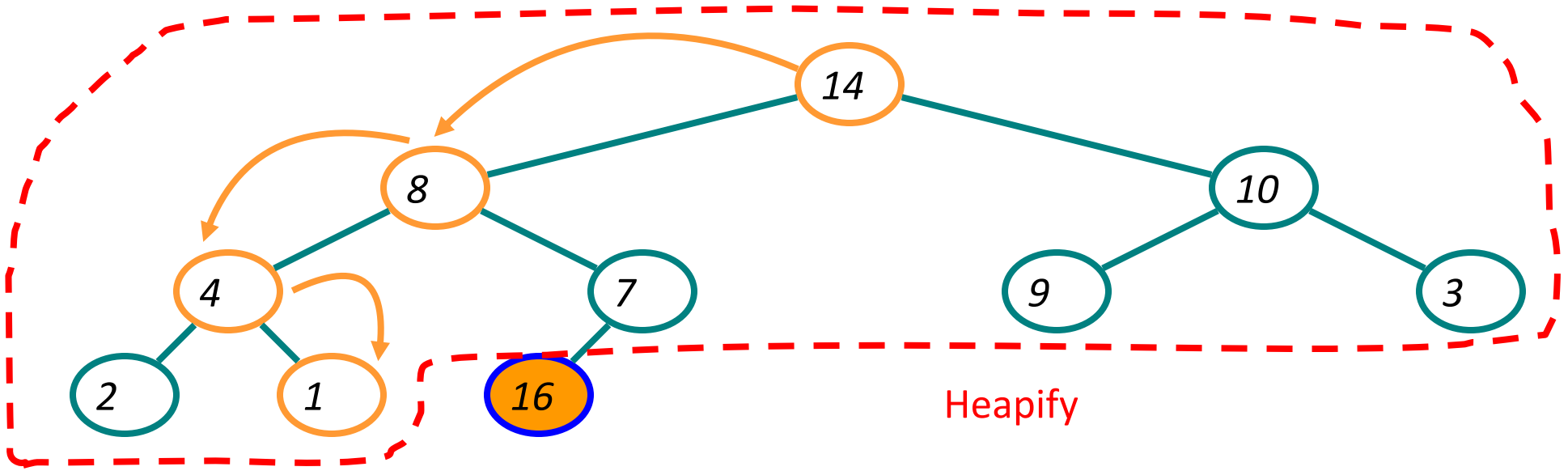
Heapsort Example

- Last element sorted



Heapsort Example

- Restore heap on remaining unsorted elements

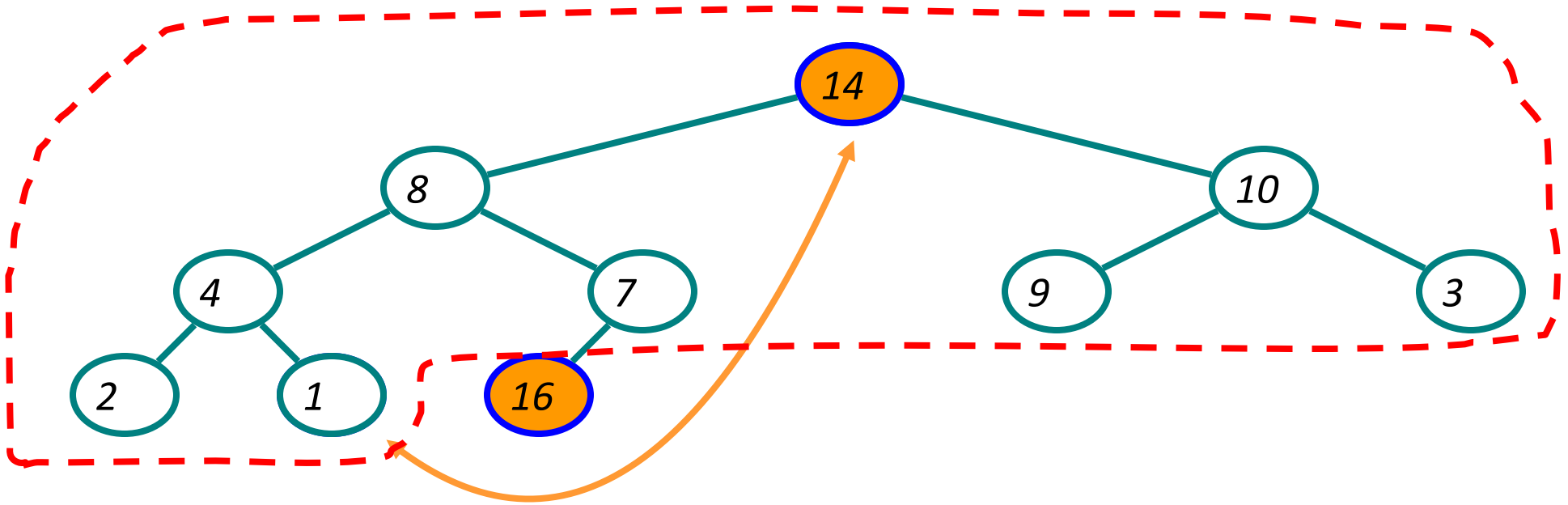


A =

14	8	10	4	7	9	3	2	1	16

Heapsort Example

- Repeat: swap new last and first

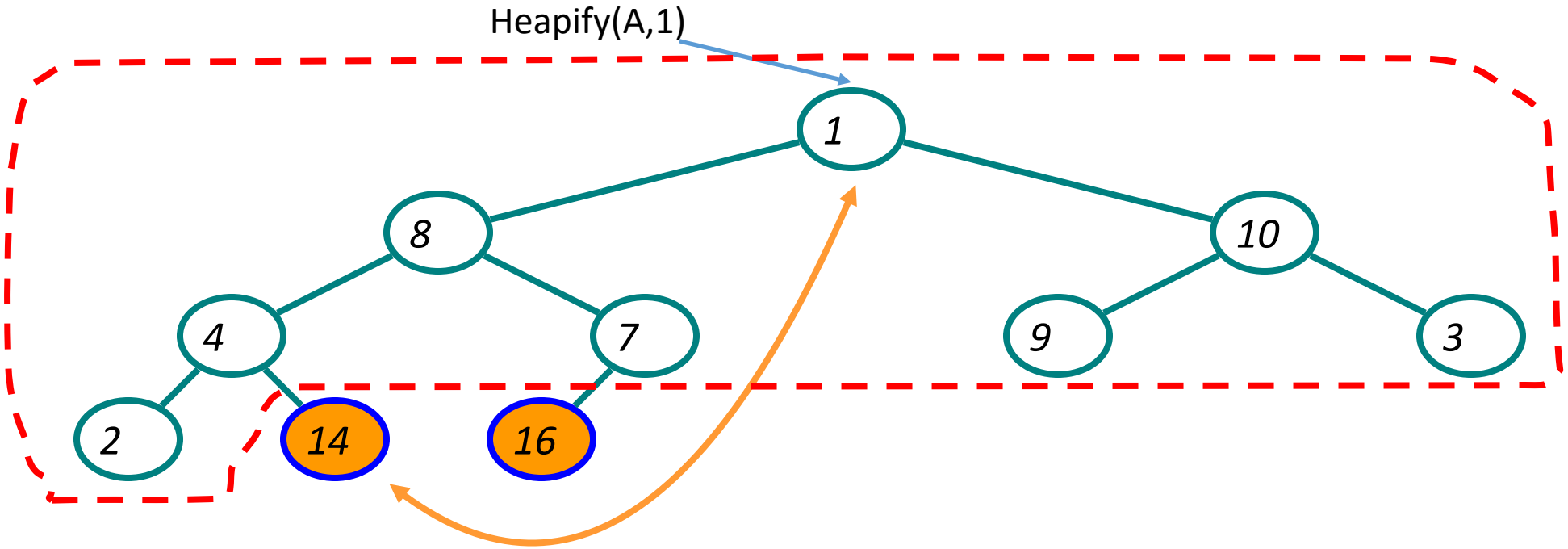


A =

1	8	10	4	7	9	3	2	14	16

Heapsort Example

- Repeat: swap new last and first

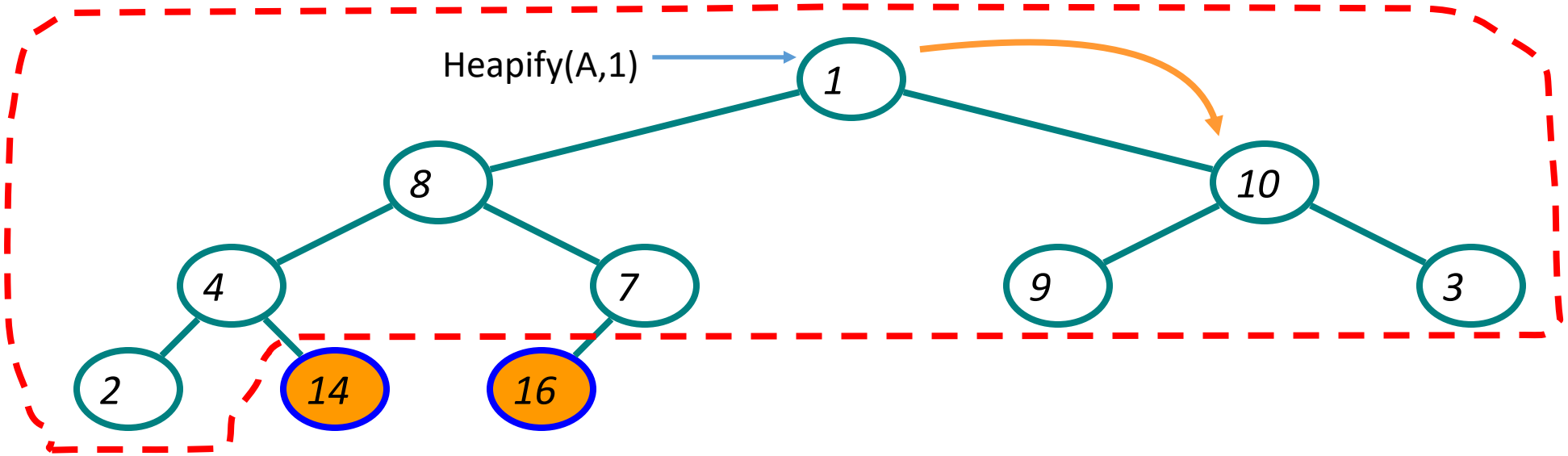


A =

1	8	10	4	7	9	3	2	14	16

Heapsort Example

- Repeat: swap new last and first

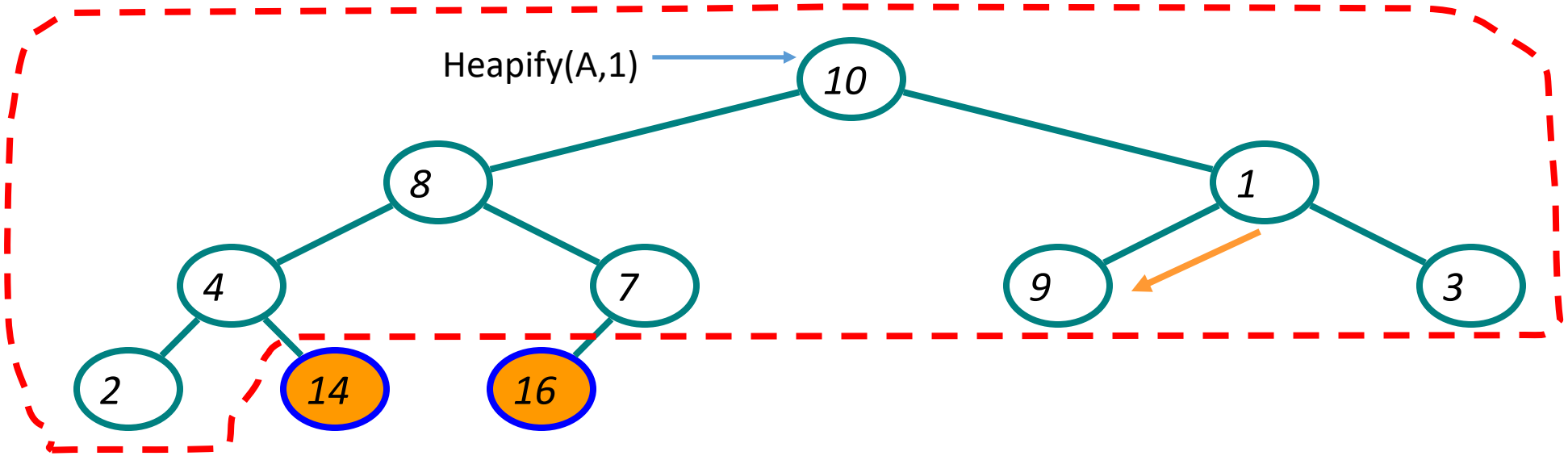


A =

1	8	10	4	7	9	3	2	14	16

Heapsort Example

- Repeat: swap new last and first

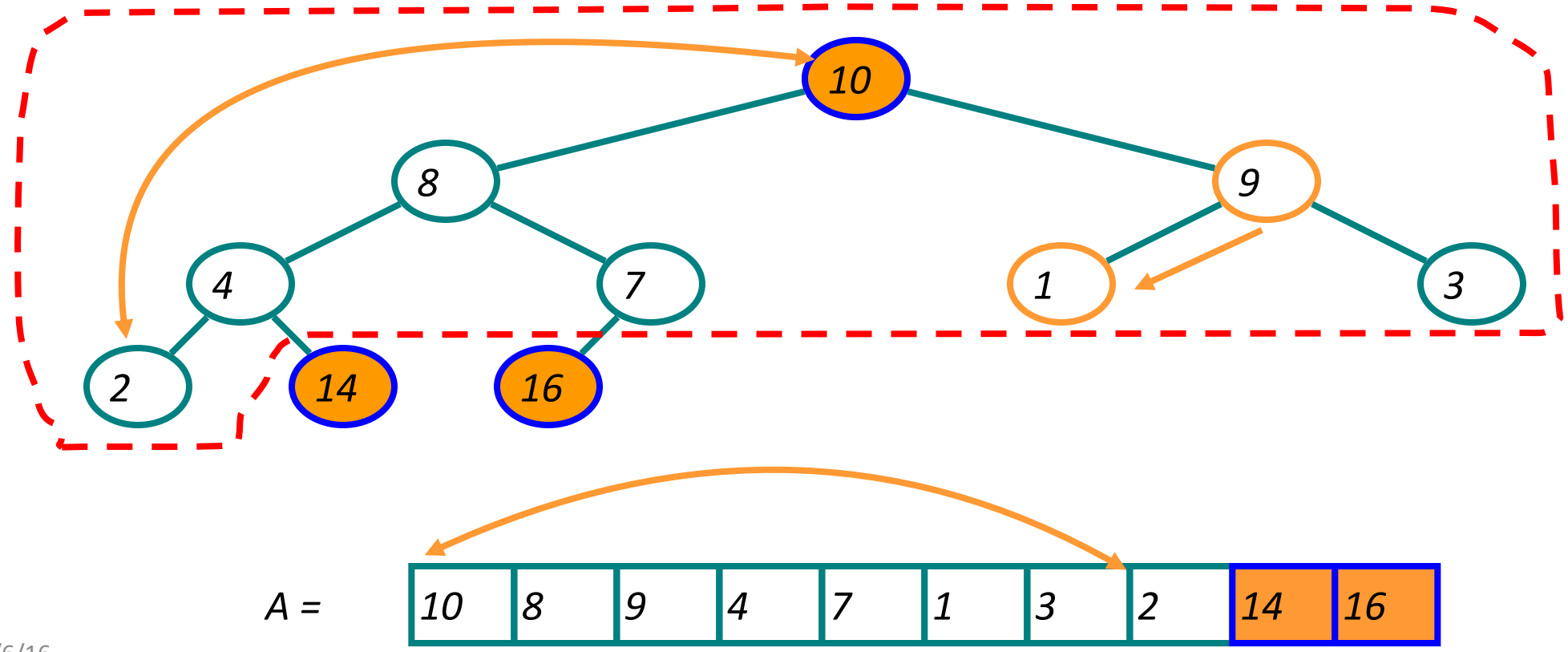


A =

1	8	10	4	7	9	3	2	14	16

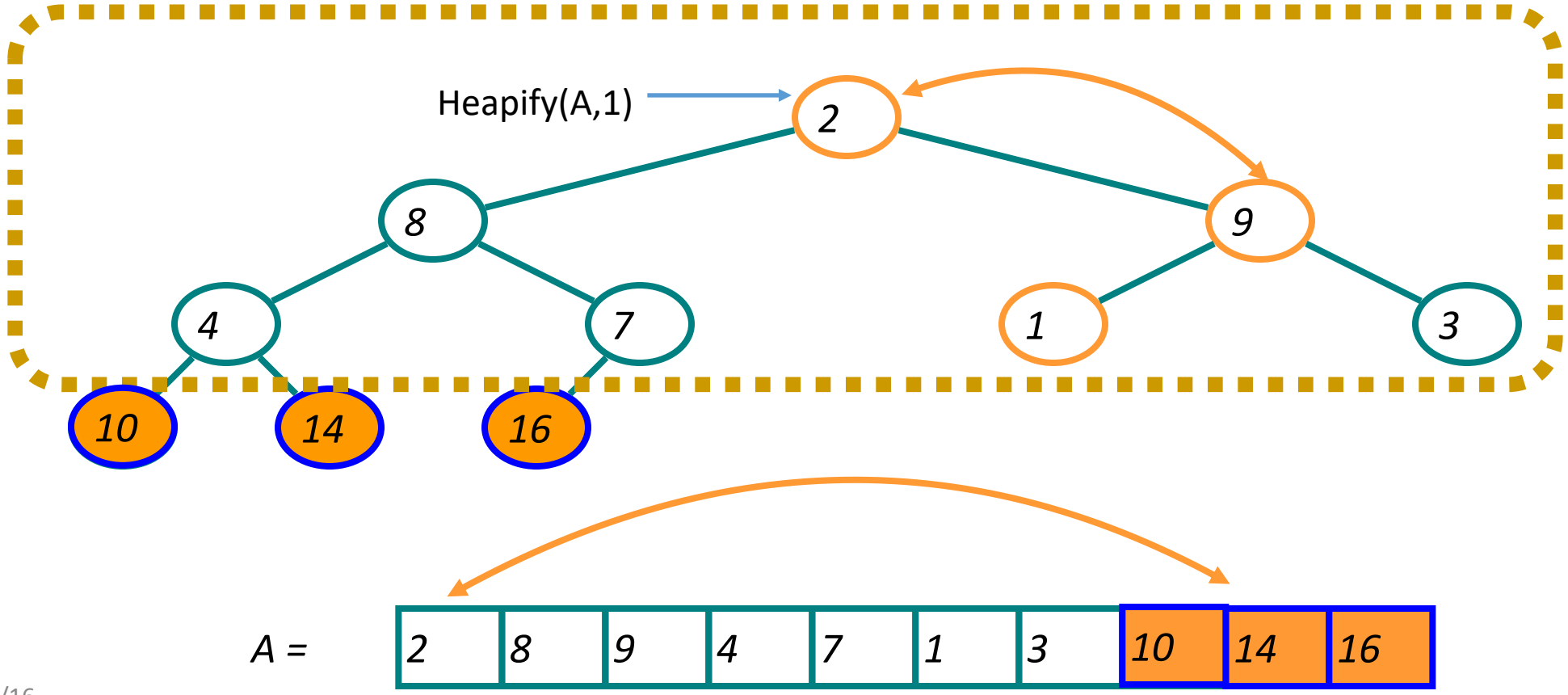
Heapsort Example

- Restore heap



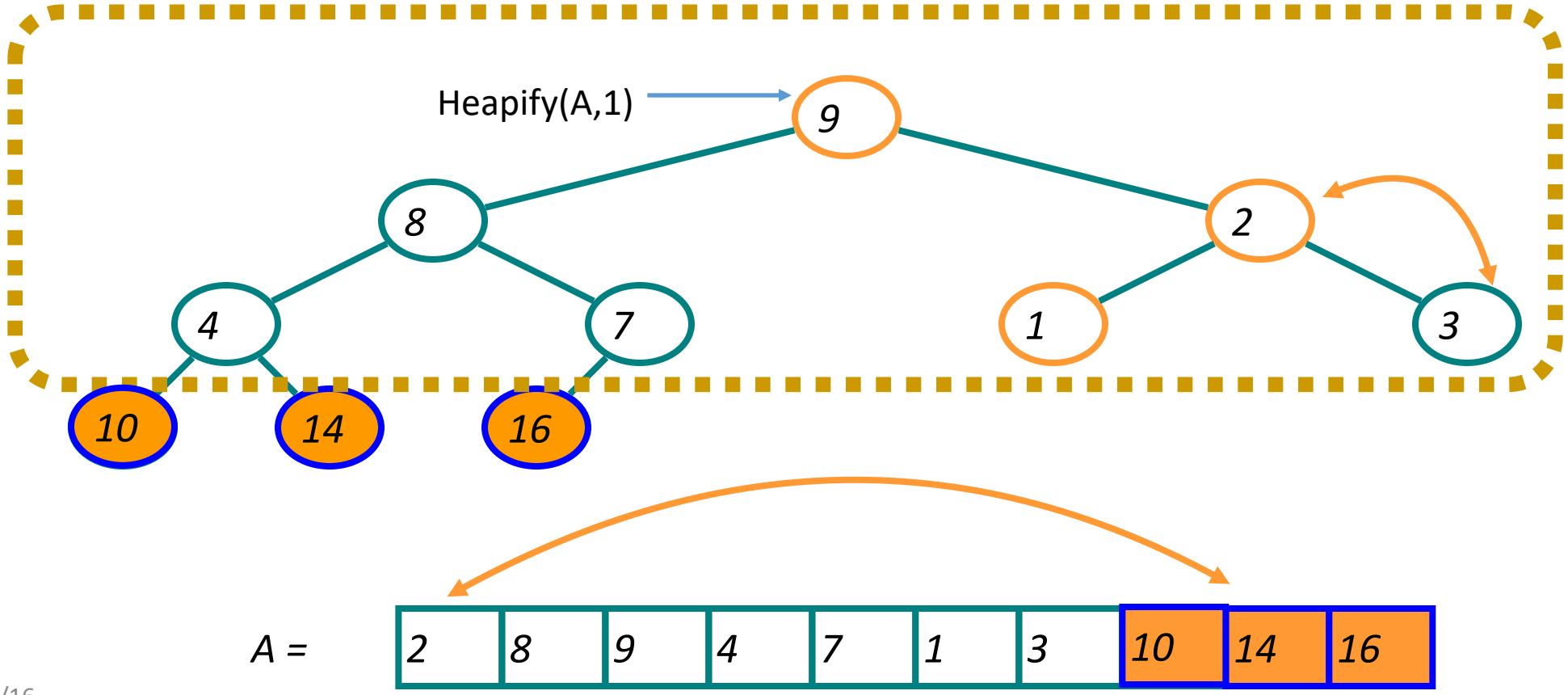
Heapsort Example

- Restore heap



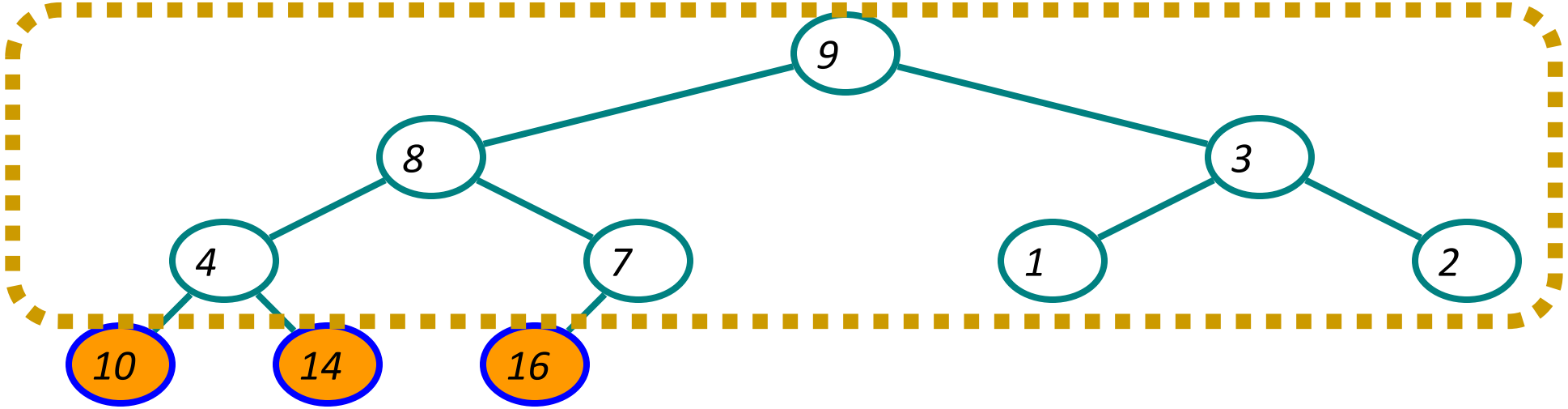
Heapsort Example

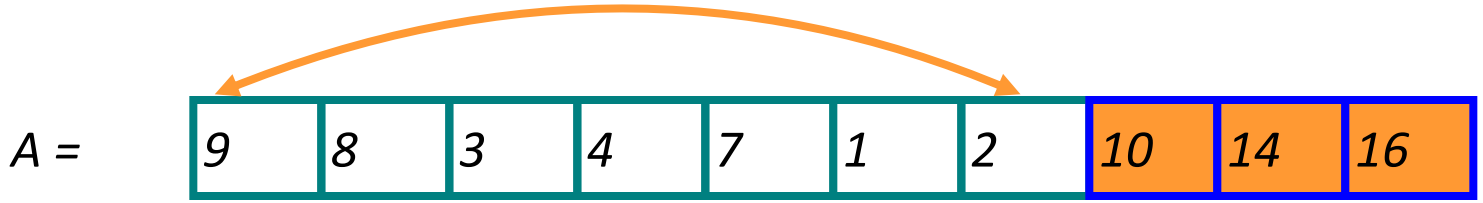
- Restore heap



Heapsort Example

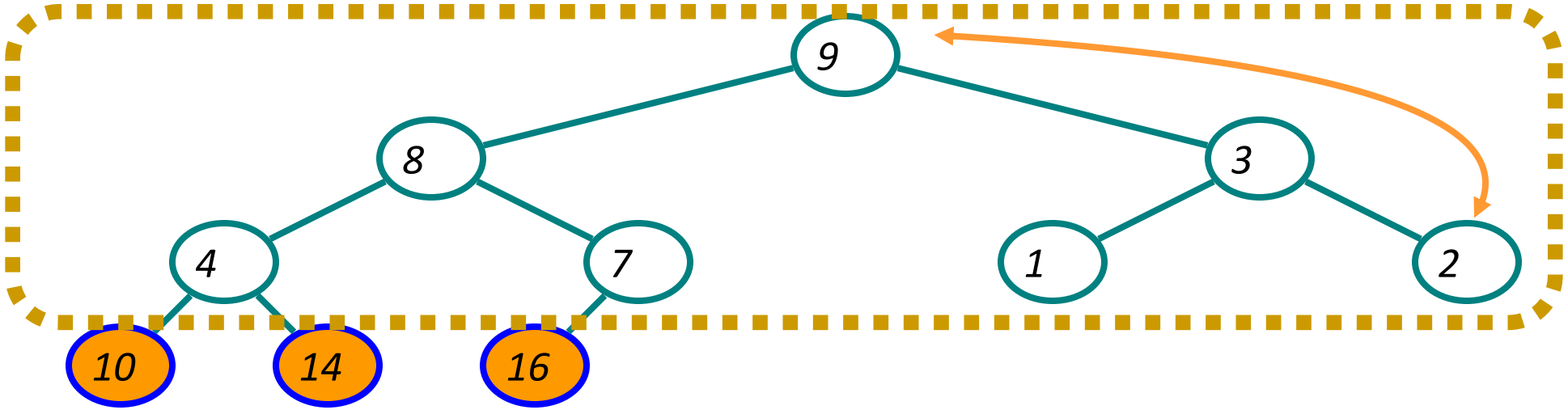
- Repeat

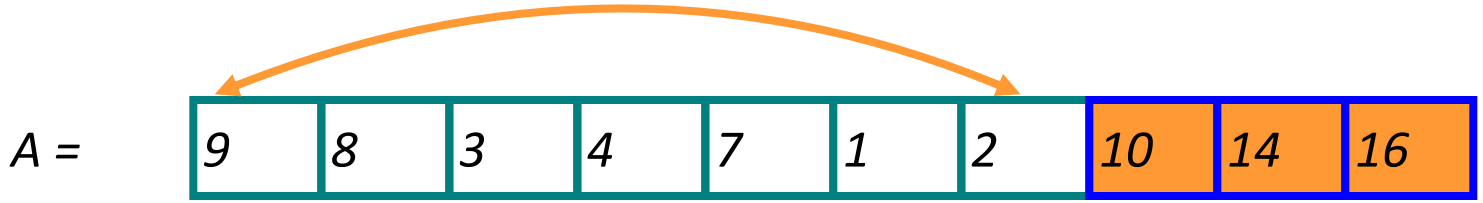




Heapsort Example

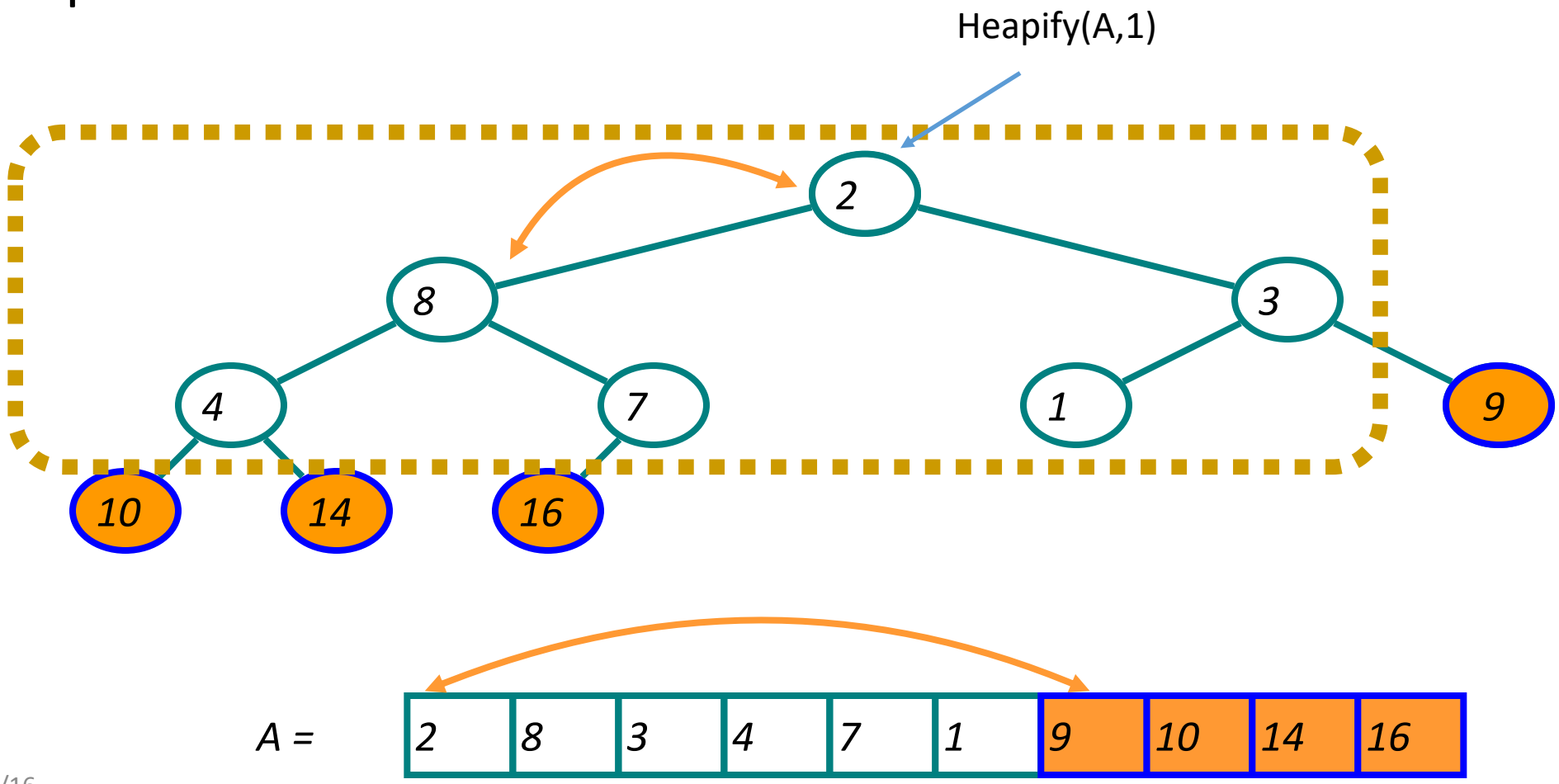
- Repeat





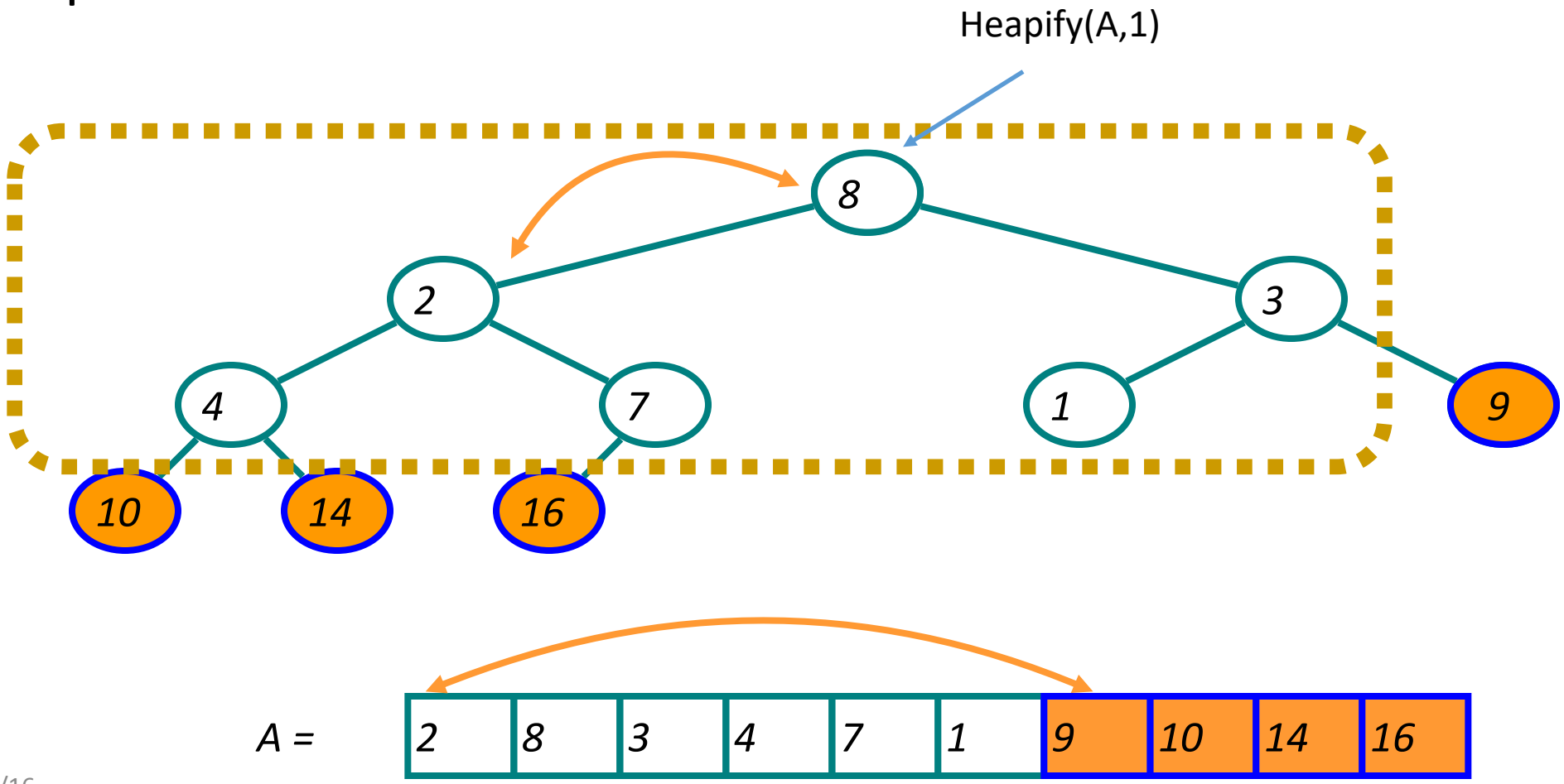
Heapsort Example

- Repeat



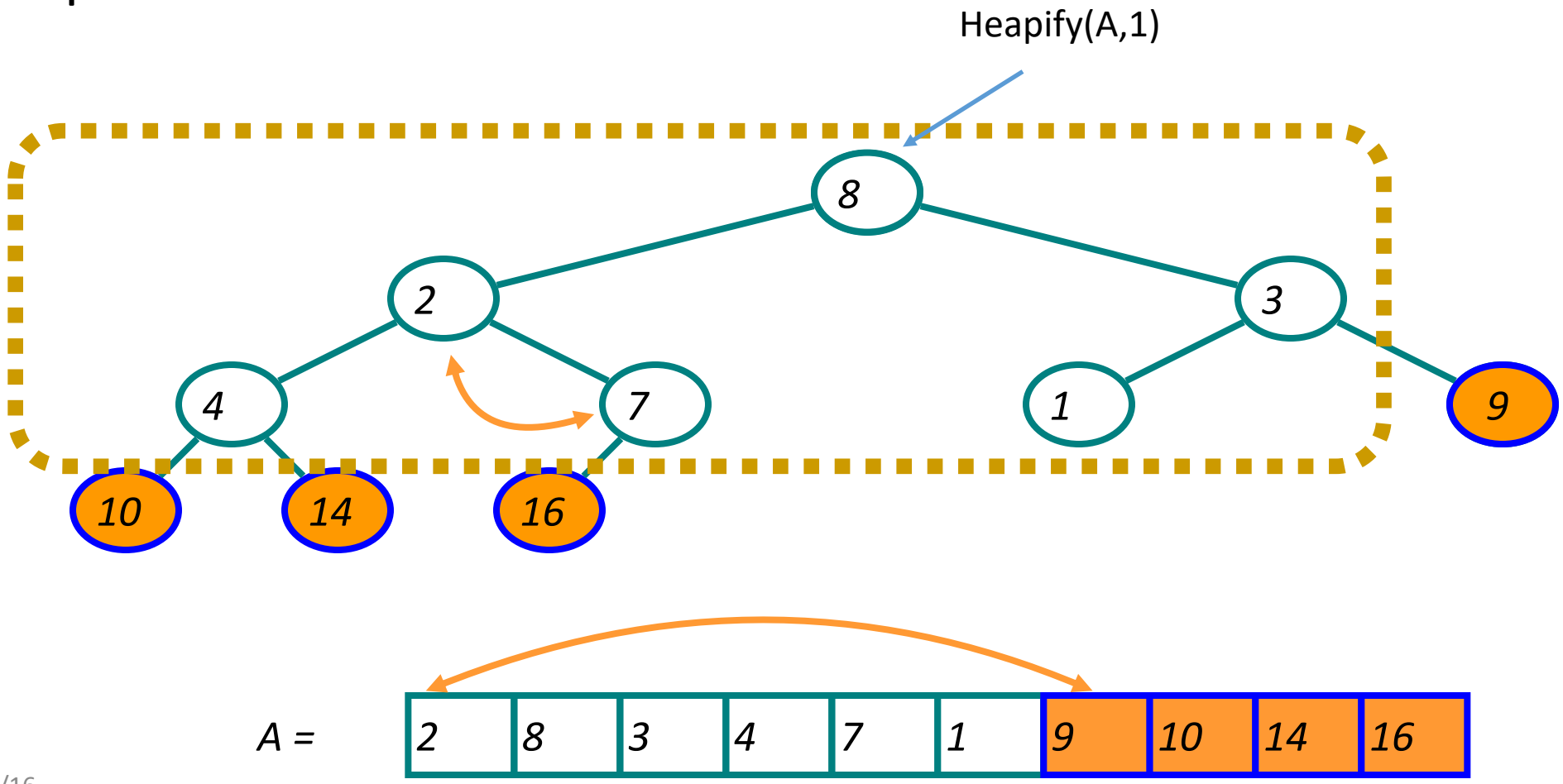
Heapsort Example

- Repeat



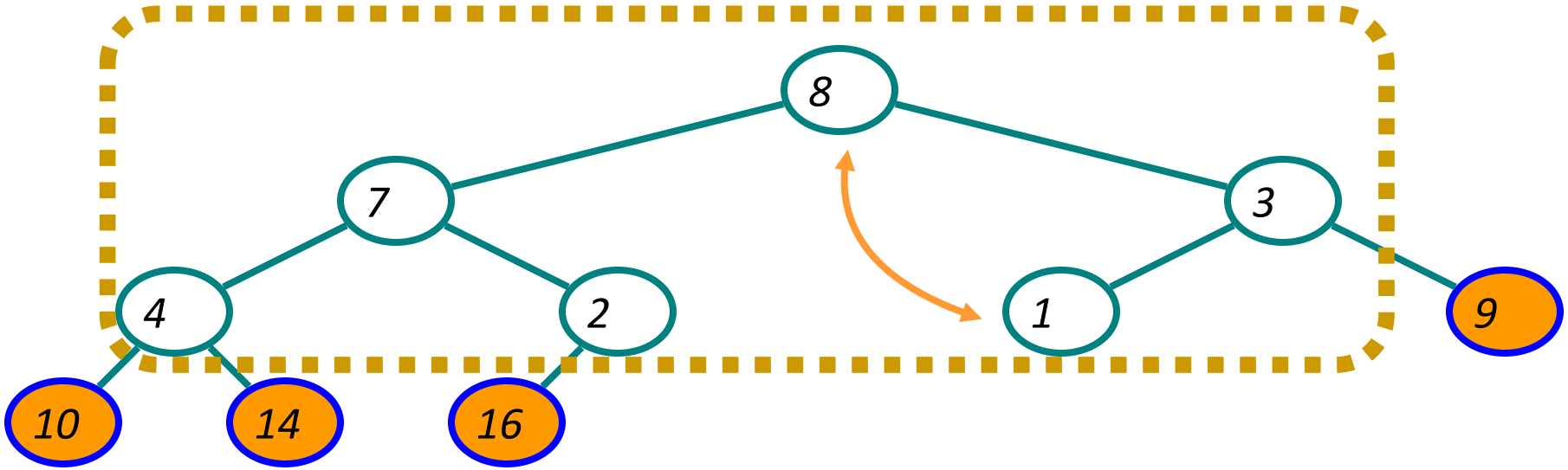
Heapsort Example

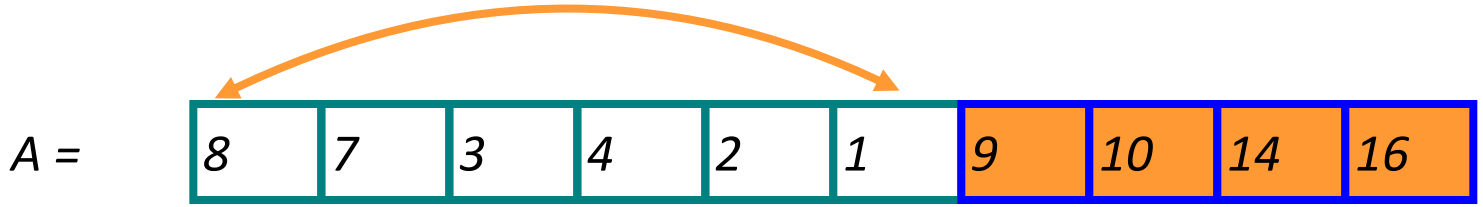
- Repeat



Heapsort Example

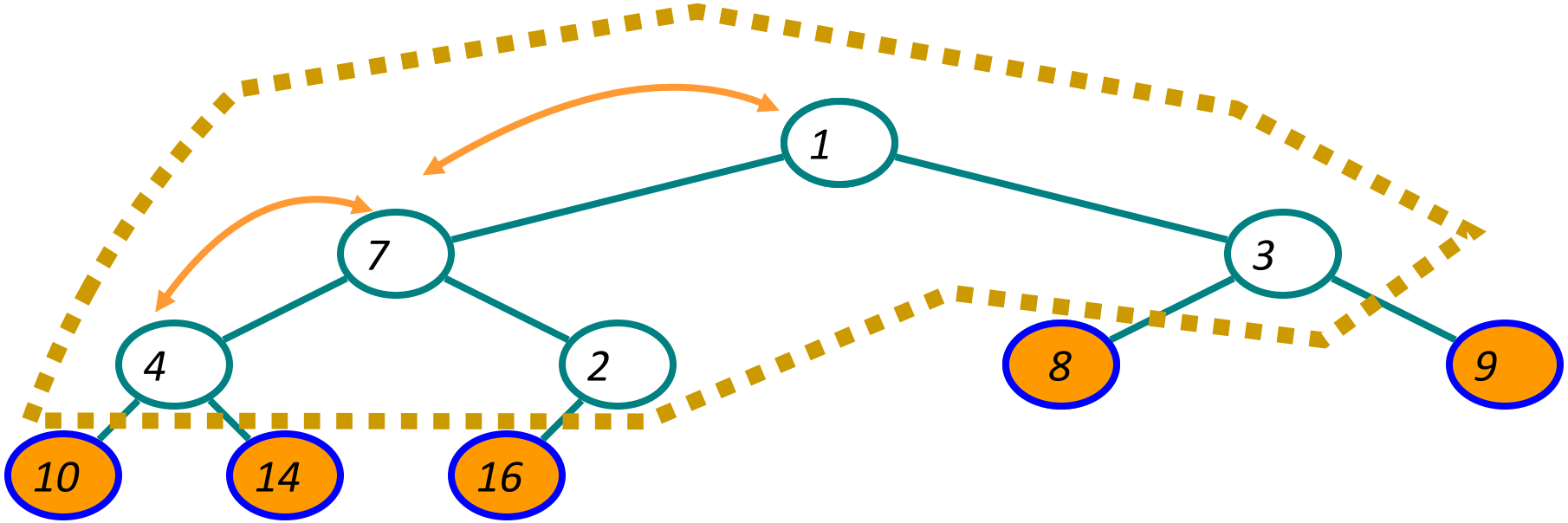
- Repeat

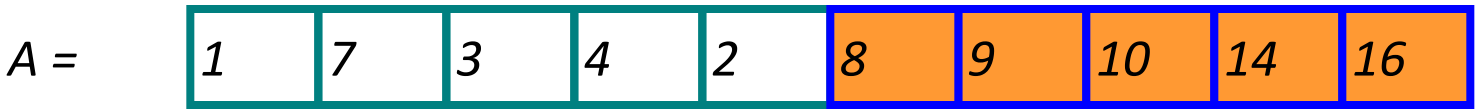




Heapsort Example

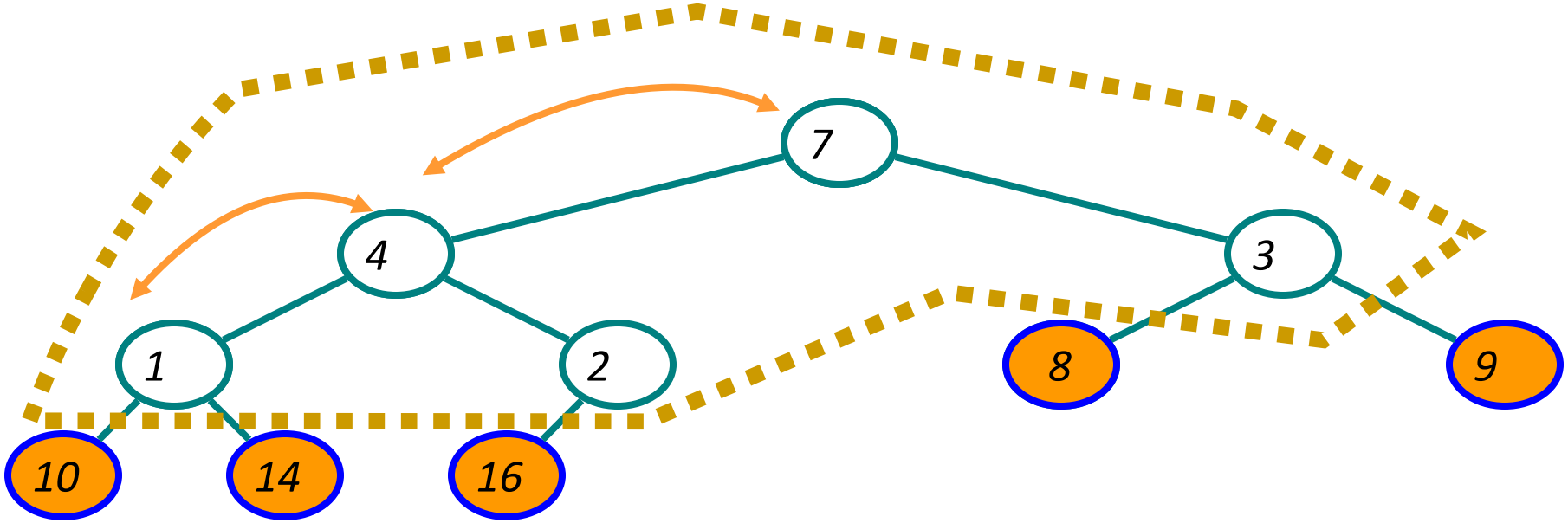
- Repeat

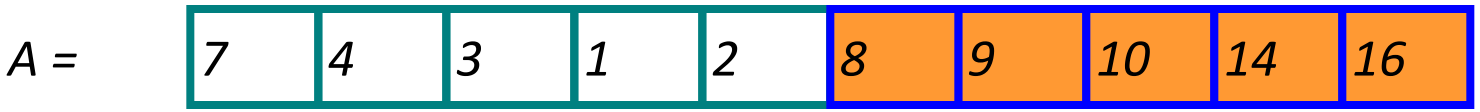




Heapsort Example

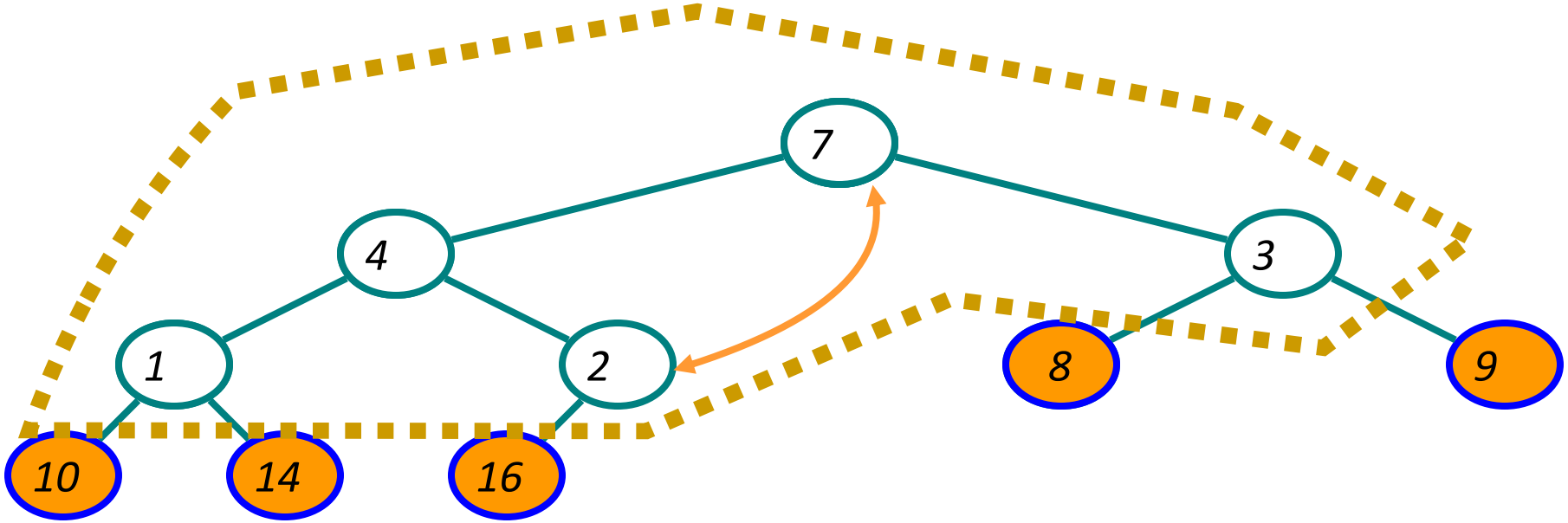
- Repeat





Heapsort Example

- Repeat

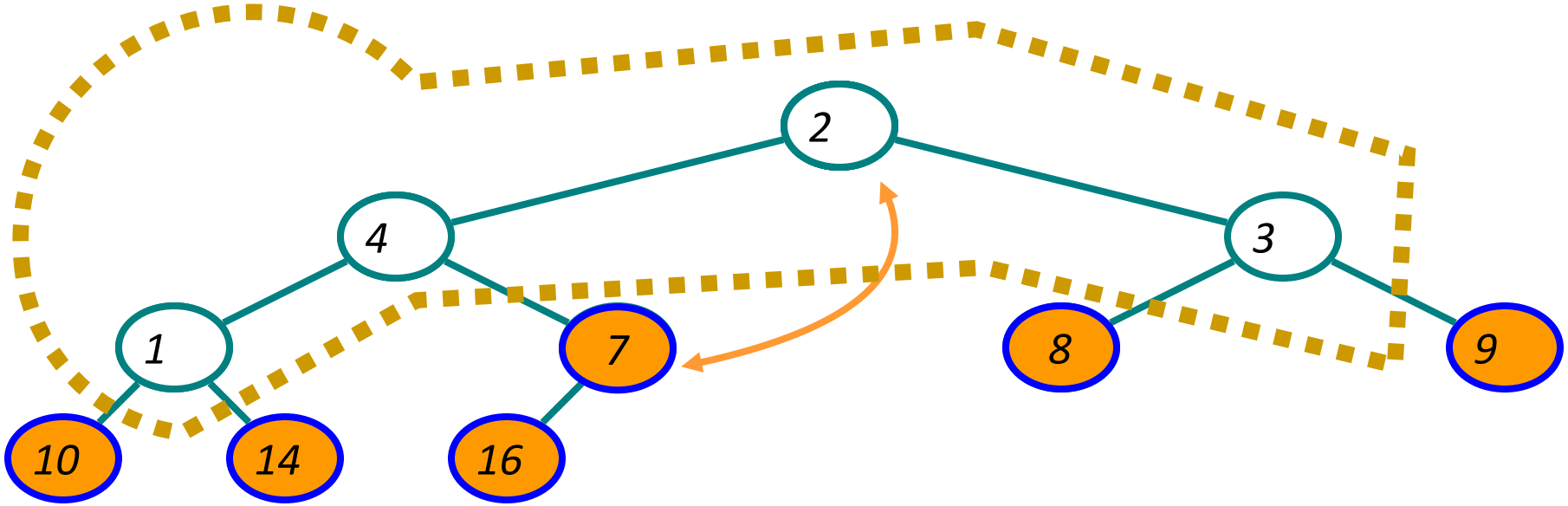


A =

7	4	3	1	2	8	9	10	14	16

Heapsort Example

- Repeat

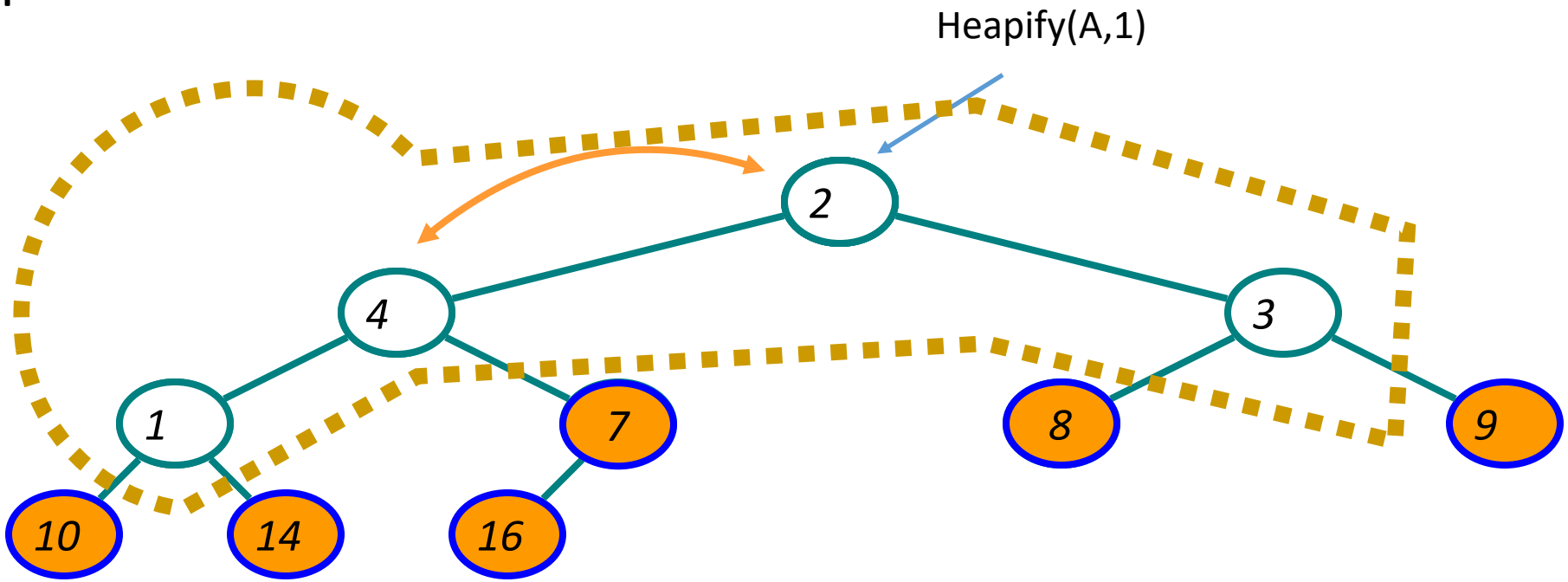


A =

2	4	3	1	7	8	9	10	14	16

Heapsort Example

- Repeat

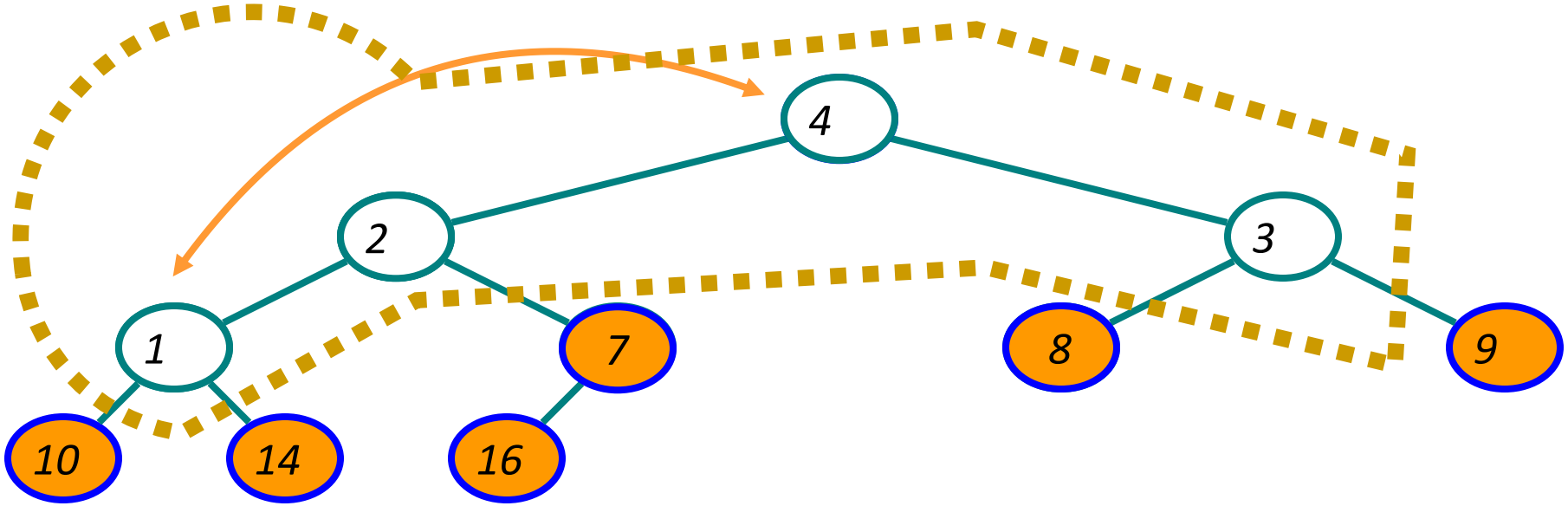


A =

2	4	3	1	7	8	9	10	14	16

Heapsort Example

- Repeat

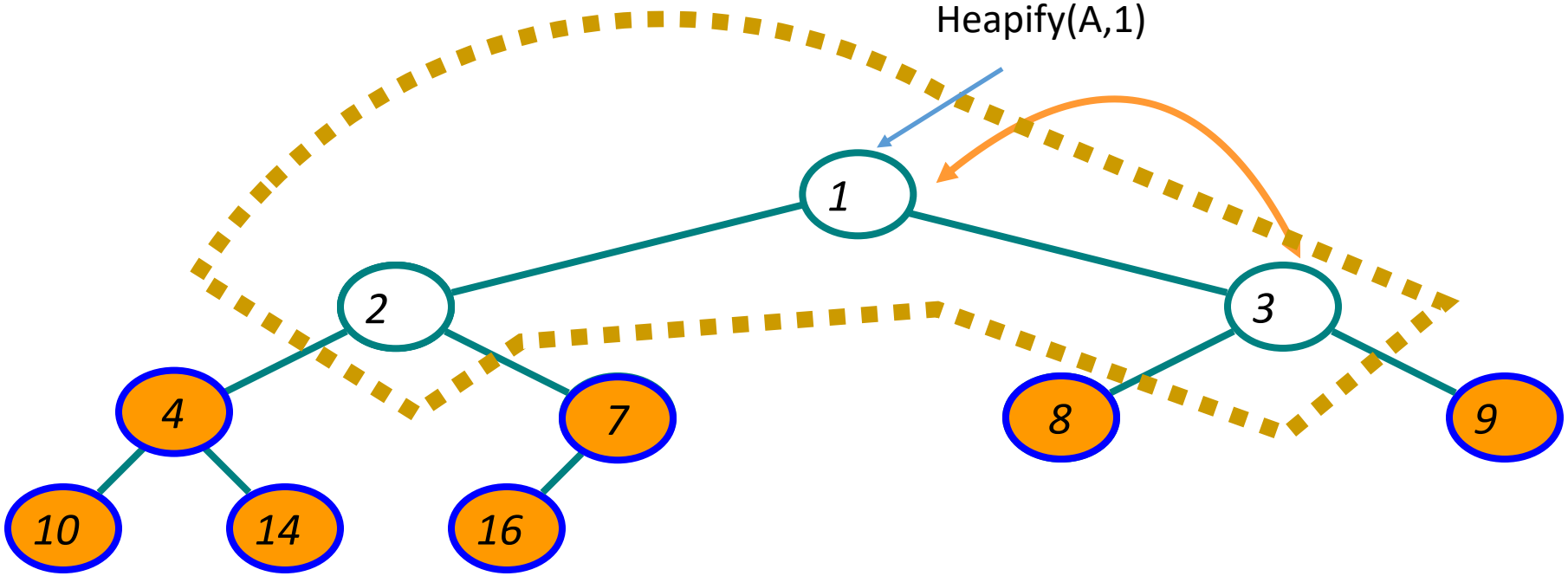


A =

4	2	3	1	7	8	9	10	14	16

Heapsort Example

- Repeat

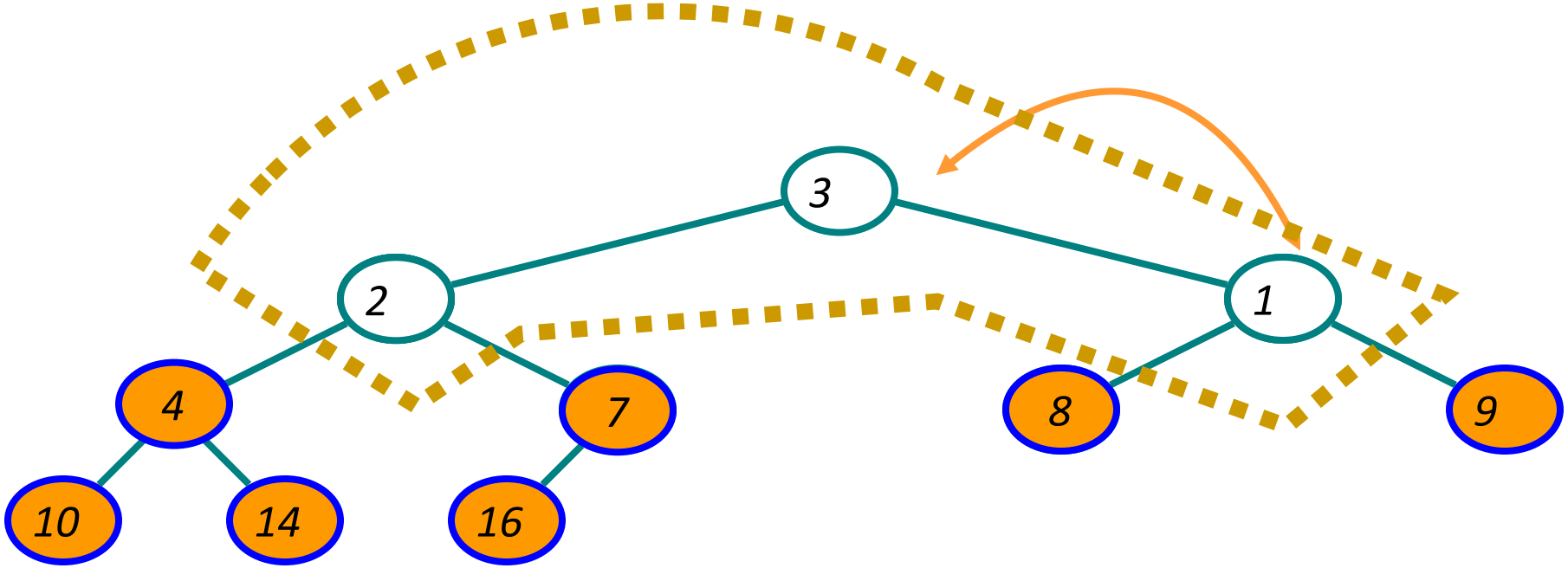


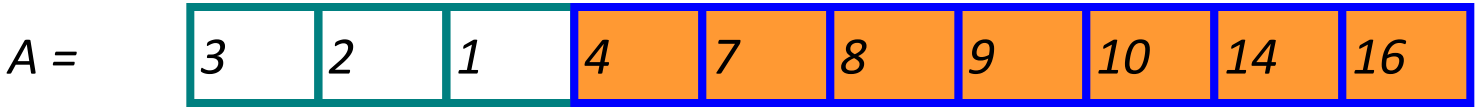
A =

1	2	3	4	7	8	9	10	14	16

Heapsort Example

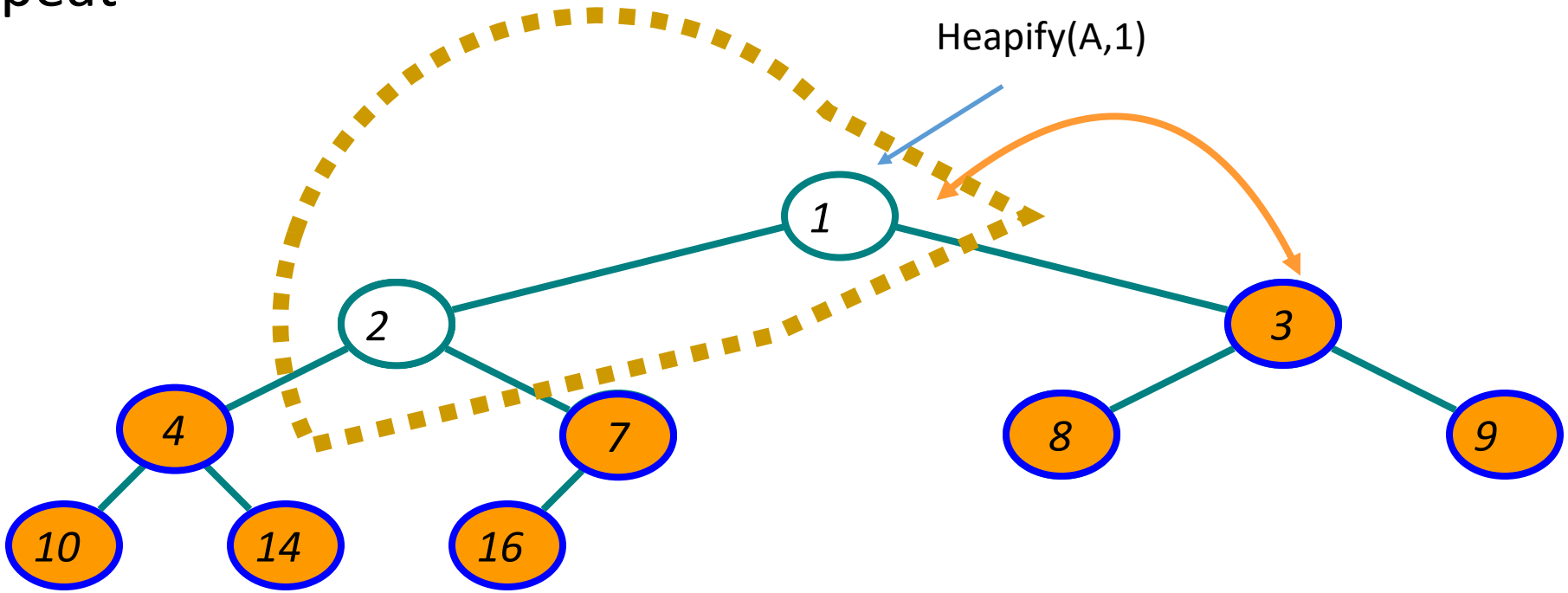
- Repeat





Heapsort Example

- Repeat

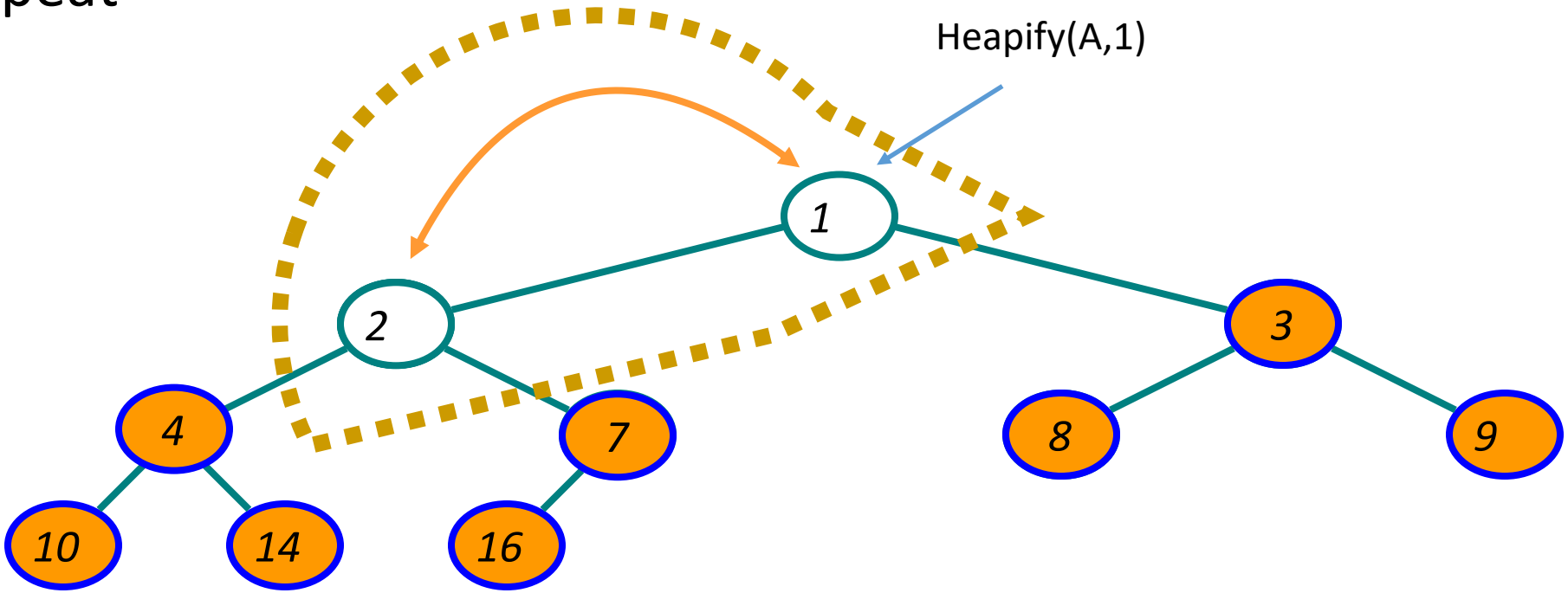


A =

1	2	3	4	7	8	9	10	14	16

Heapsort Example

- Repeat

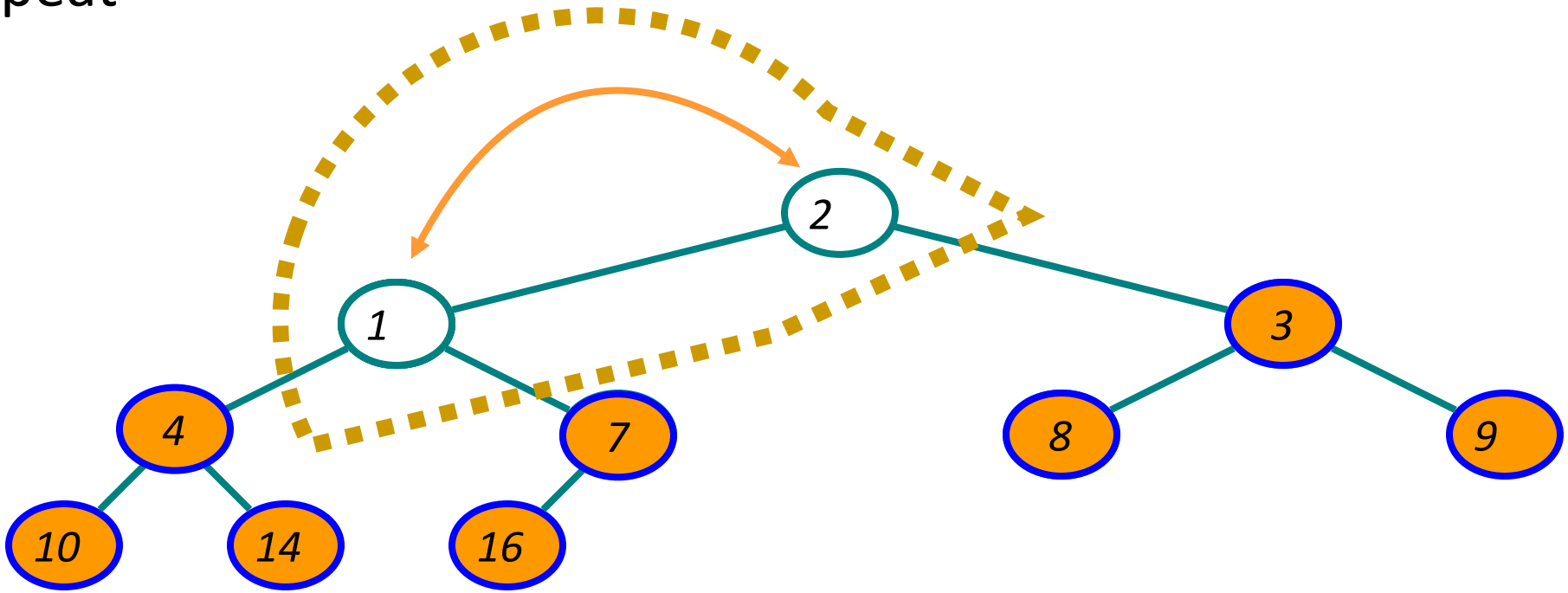


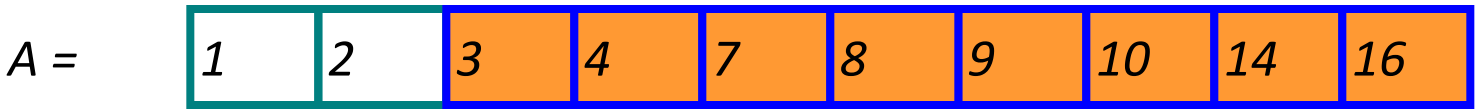
A =

1	2	3	4	7	8	9	10	14	16

Heapsort Example

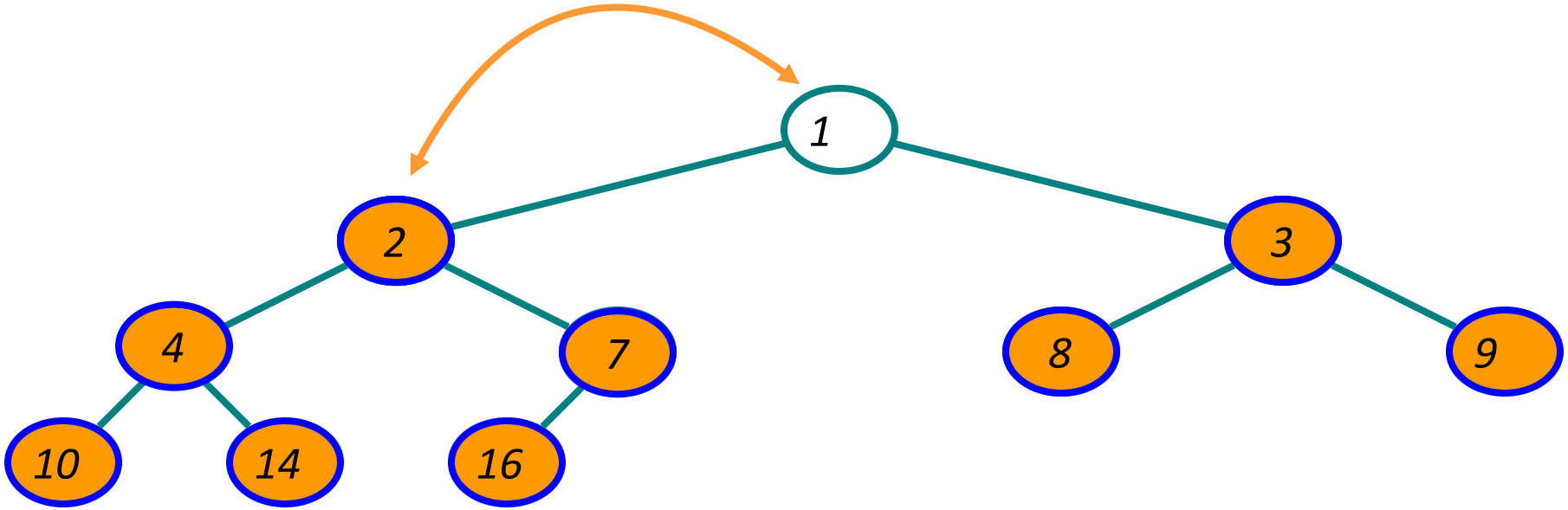
- Repeat

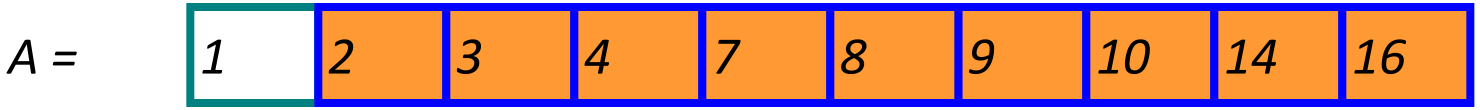




Heapsort Example

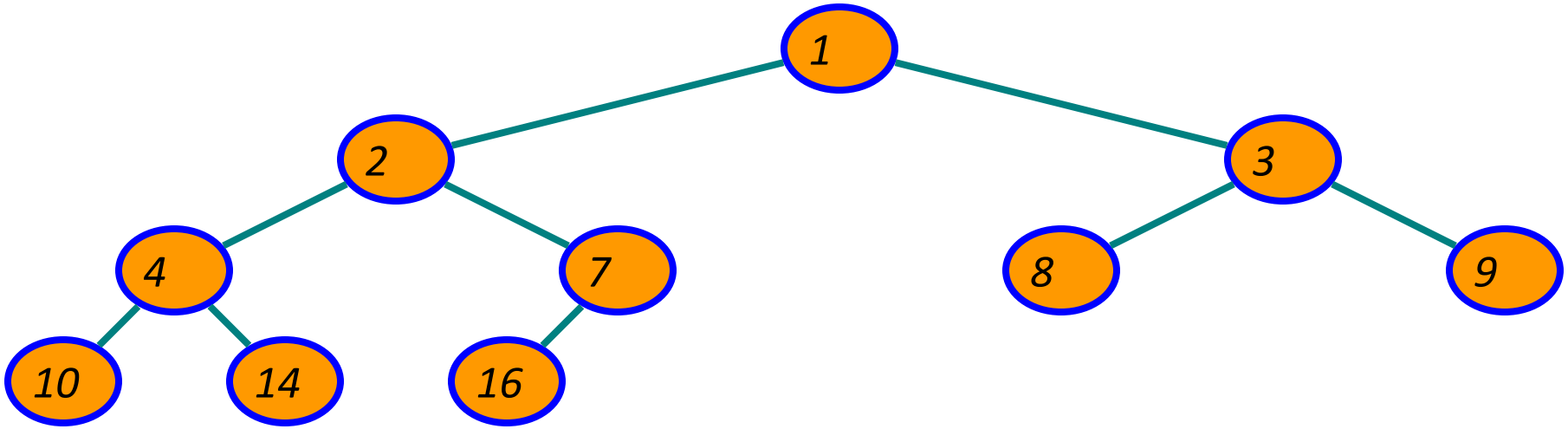
- Repeat

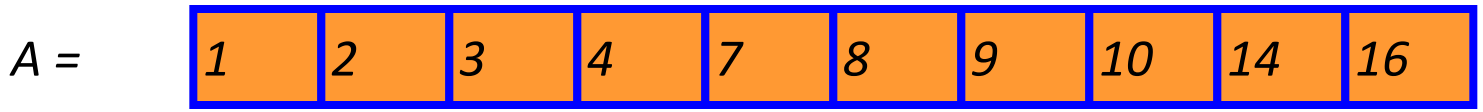




Heapsort Example

- Repeat





Analyzing Heapsort

- The call to **`BuildHeap()`** takes O(*n*) time
- Each of the *n* - 1 calls to **`Heapify()`** takes O(lg *n*) time
- Thus the total time taken by **`HeapSort()`**
  = O(*n*) + (*n* - 1) O(lg *n*)
  = O(*n*) + O(*n* lg *n*)
  = O(*n* lg *n*)

Comparison

	Time complexity	Stable?	In-place?
Merge sort			
Quick sort			
Heap sort			

Comparison

	Time complexity	Stable?	In-place?
Merge sort	Θ (n log n)	Yes	No (or Yes)
Quick sort	Θ(n log n) expected. $\Theta(n^2)$ worst case	No	Yes
Heap sort	Θ (n log n)	No	Yes

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort usually wins
- The heap data structure is incredibly useful for implementing priority queues
  - A data structure for maintaining a set S of elements, each with an associated value or key
  - Supports the operations Insert(), Maximum(), ExtractMax(), changeKey()
- What might a priority queue be useful for?

Priority Queue Operations

- Insert(S, x)
  - insert element x into set S
- Maximum(S)
  - returns the element of S with the maximum key
- ExtractMax(S)
  - removes and returns the element of S with the maximum key
- ChangeKey(S, i, ***k***)
  - Change the key for element i to ***k***
- How could we implement these operations using a heap?

Your personal travel destination list

- A list of places, S, that you want to visit, each with a preference score
- Always visit the place with highest score, **Maximum**(S)
- Remove a place after visiting it, **ExtractMax**(S)
- You frequently add more destinations, **Insert**(S, x)
- You may change score for a place when you have more information, **ChangeKey**(S, i, ***k***)
- What's the best data structure?







Heap vs Array

Heap

- HeapSort: Θ(n log n)
- Maximum: Θ(1)
- ExtractMax: Θ(log n)
- ChangeKey: Θ(log n)
- Insert: Θ(log n)

- Heapify: Θ(log n)
- BuildHeap: Θ(n)

Sorted Array

- Sort: Θ(n log n)
- Maximum: Θ(1)
- ExtractMax: Θ(n) or Θ(1)
- ChangeKey: Θ(n)
- Insert: Θ(n)

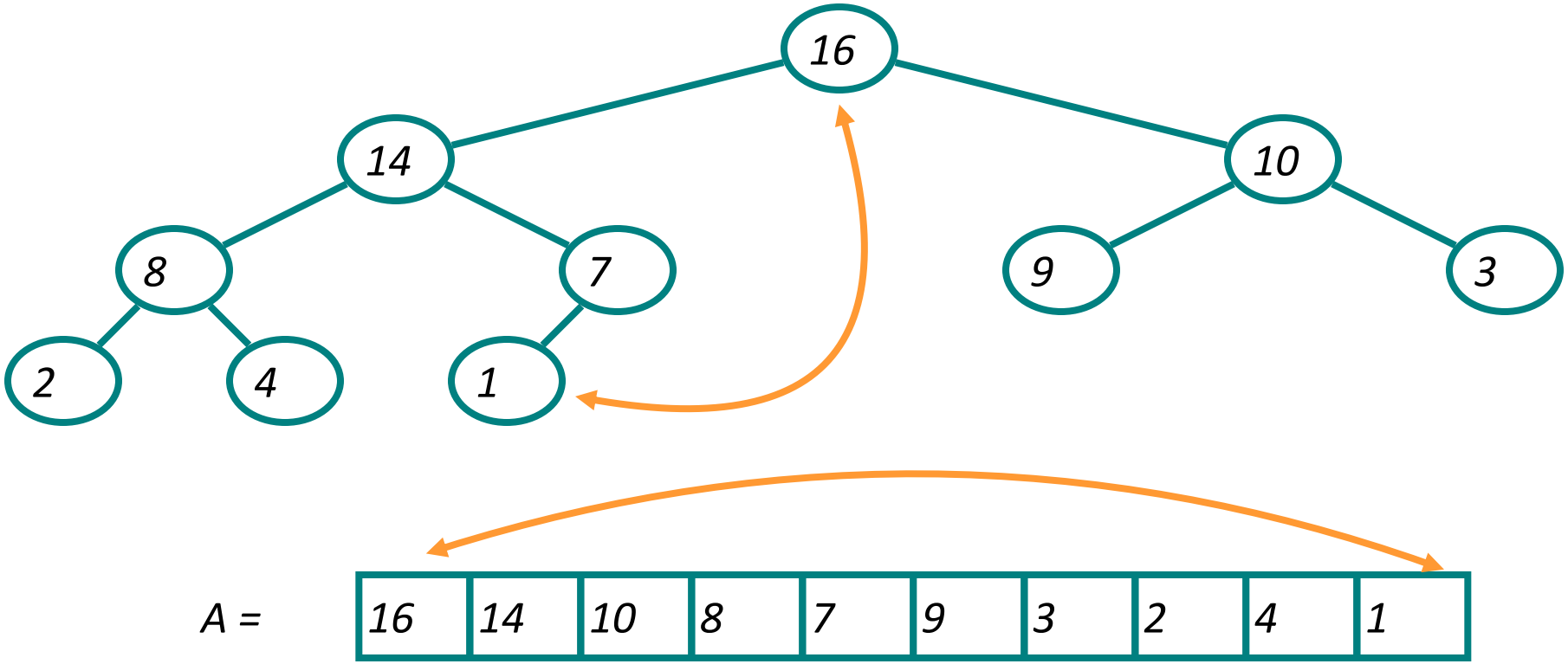
Heap: Maximum

```
HeapMaximum(A)
{
    return A[1];
}
```

Heap: Extract Max

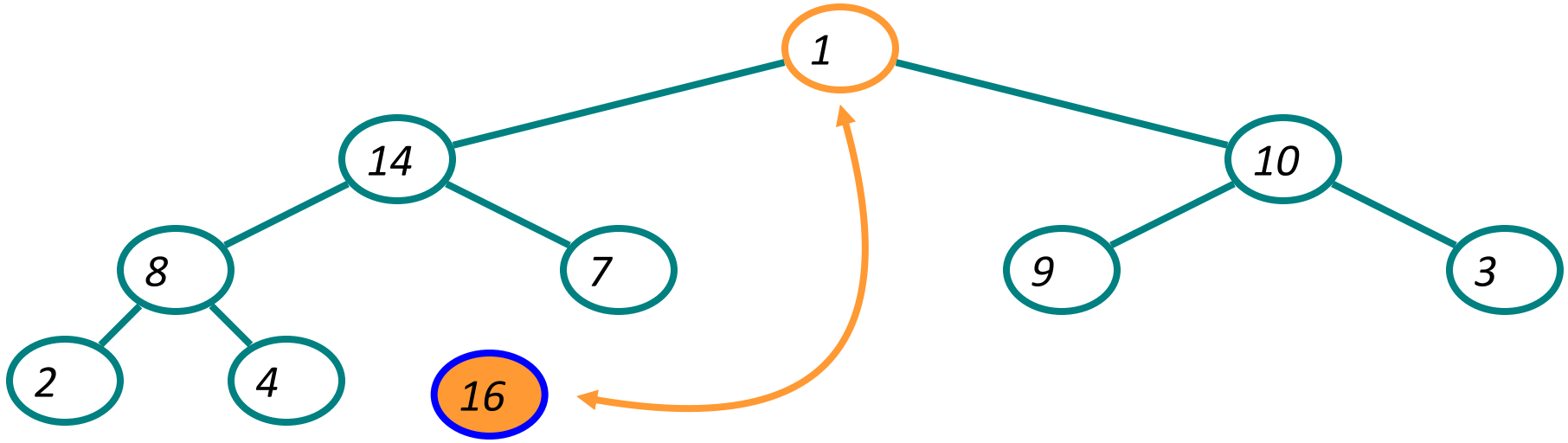
```
HeapExtractMax(A)
{
    if (heap_size[A] < 1) { error; }
    max = A[1];
    A[1] = A[heap_size[A]]
    heap_size[A] --;
    Heapify(A, 1);
    return max;
}
```

Heap: Extract Max

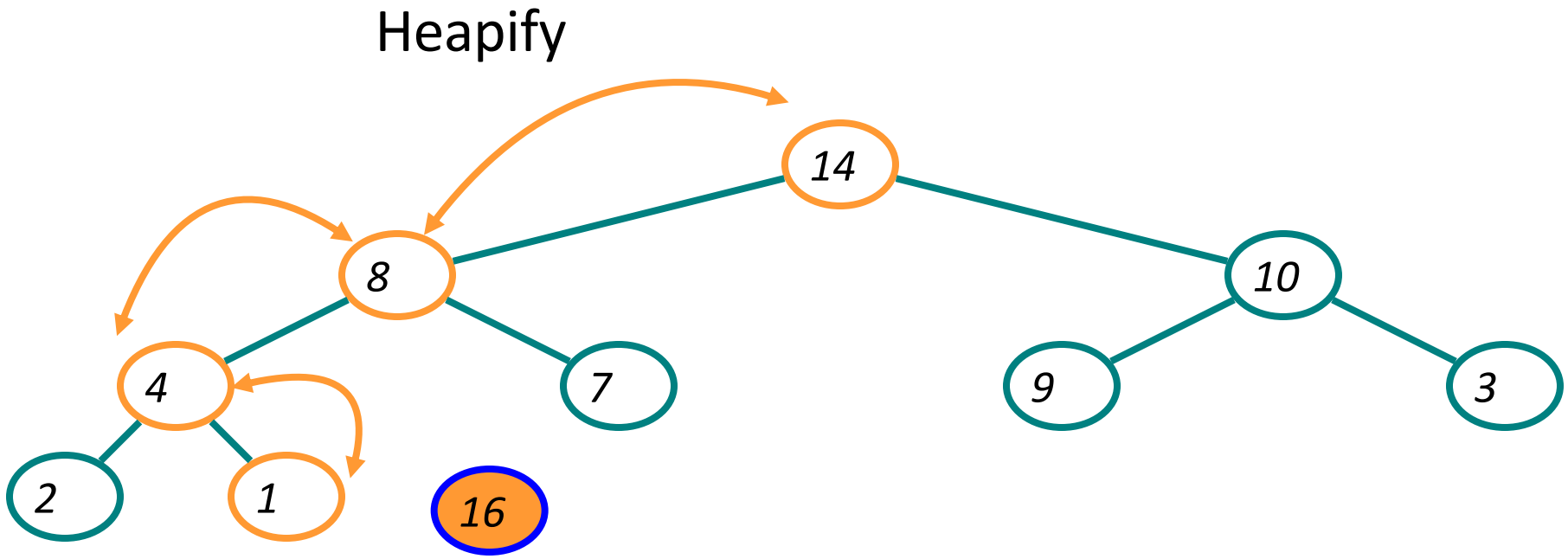


Heap: Extract Max

Swap first and last, then remove last



Heap: Extract Max

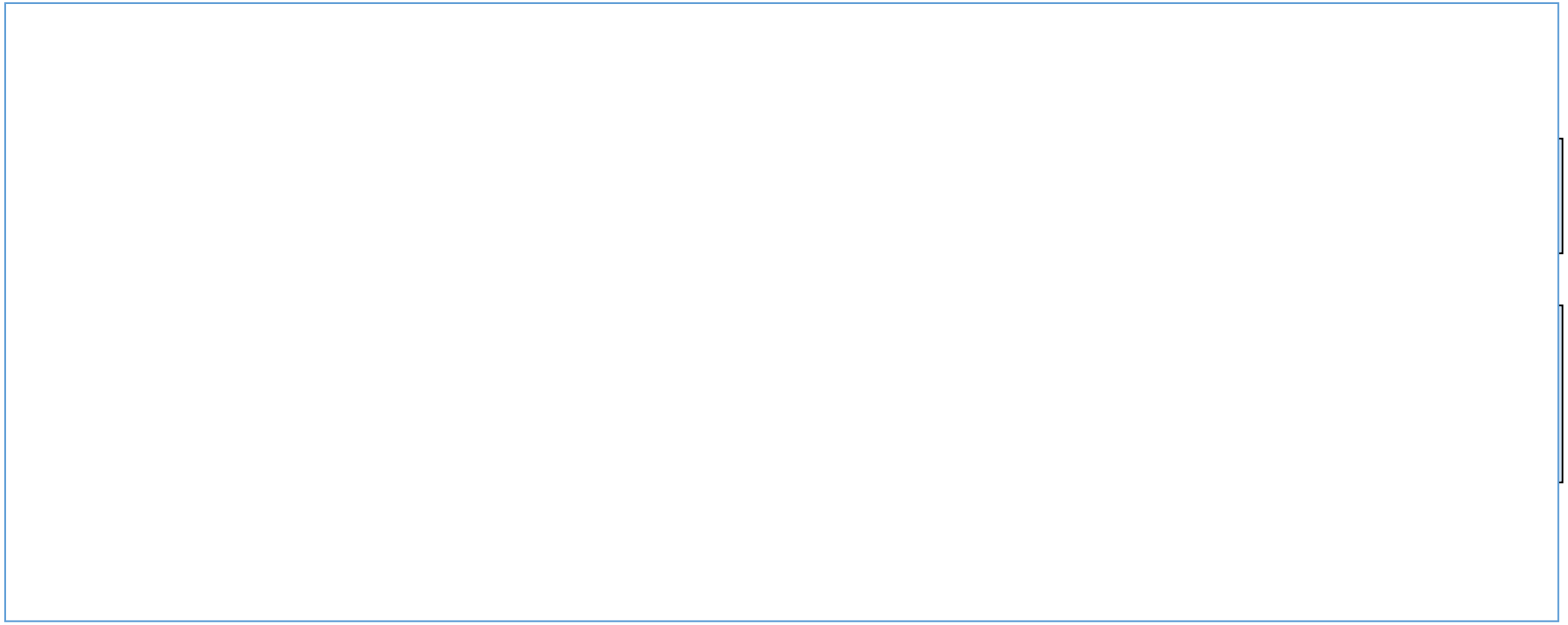


A =

14	8	10	4	7	9	3	2	1

16

Heap Change Key

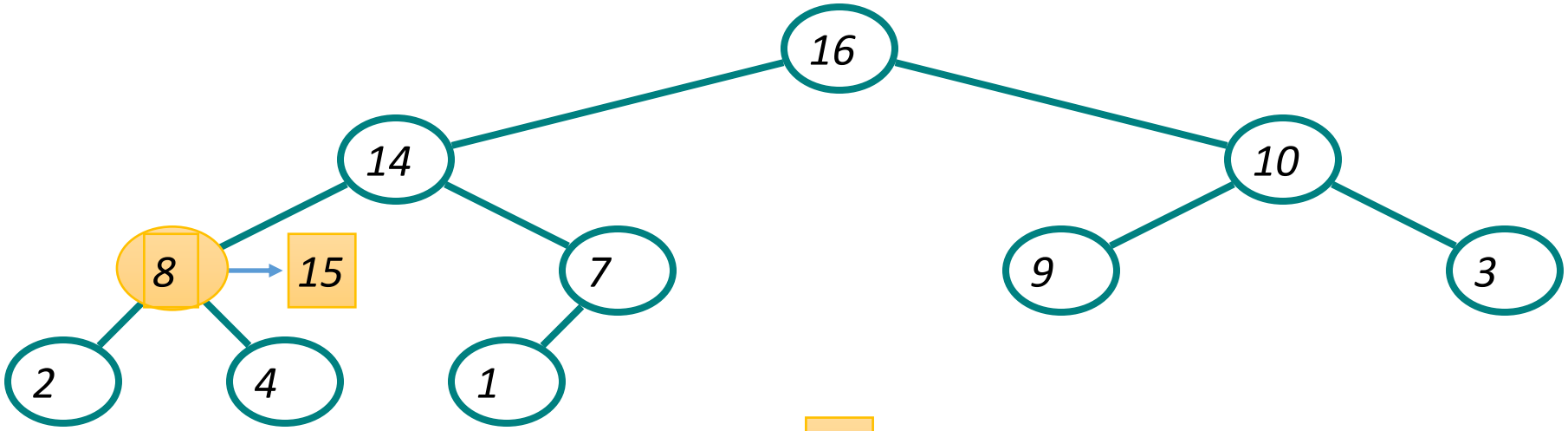


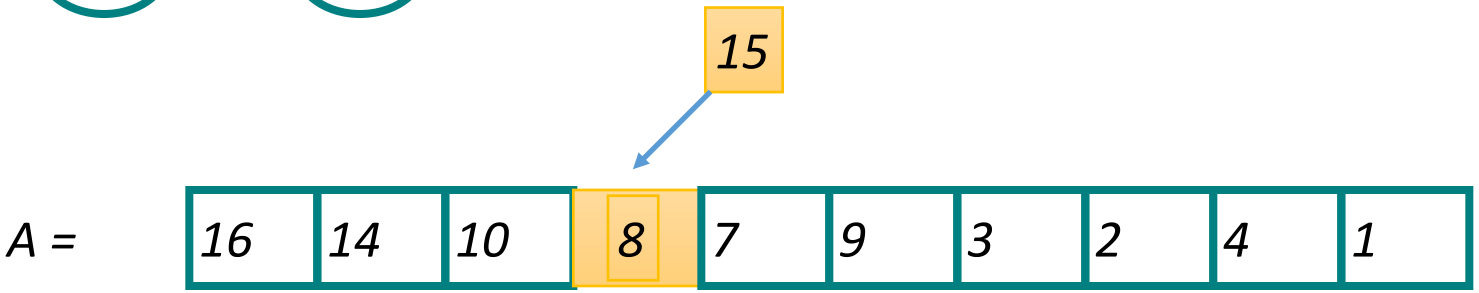
Heap Change Key

HeapChangeKey(A, 4, 15)

4th element

Change key value to 15



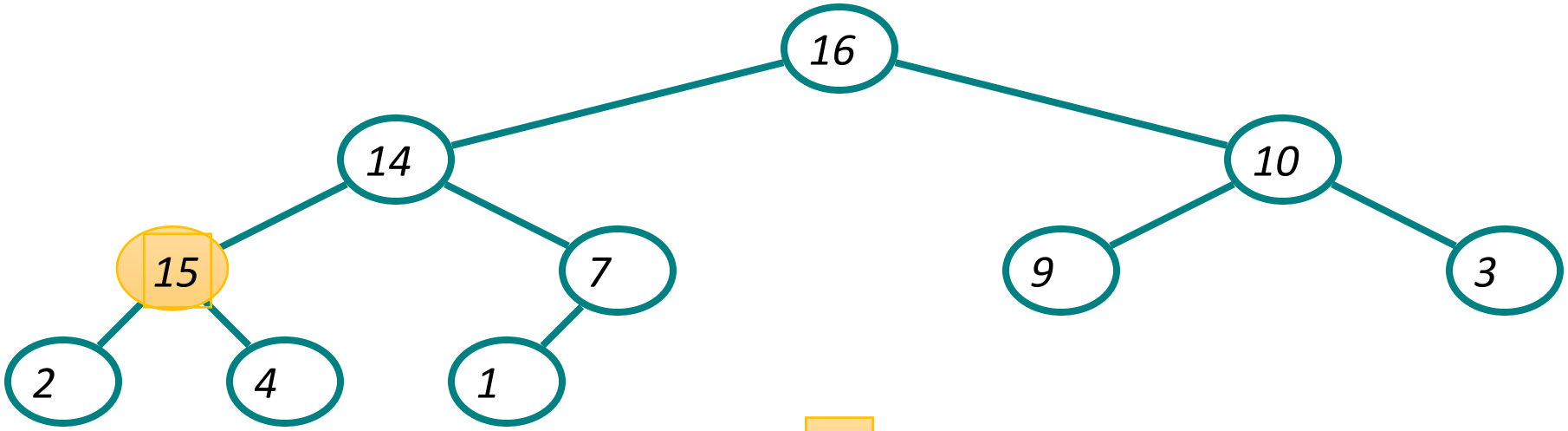


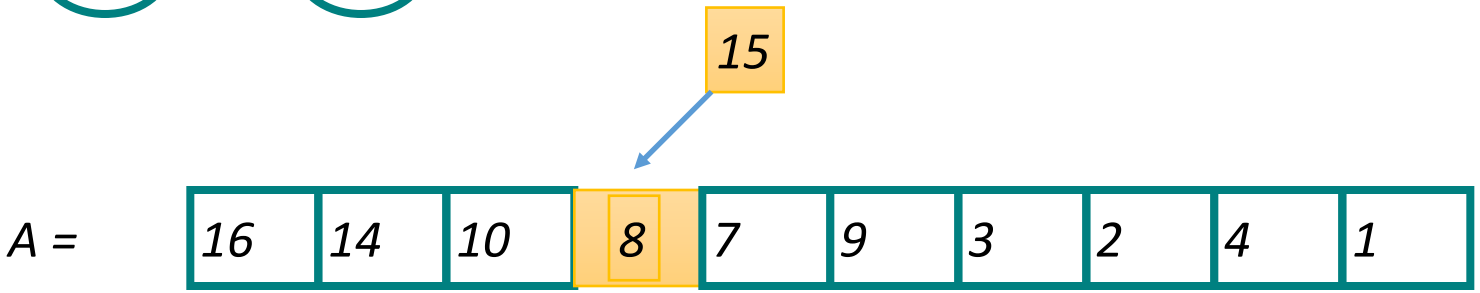
Heap Change Key

HeapChangeKey(A, 4, 15)

4th element

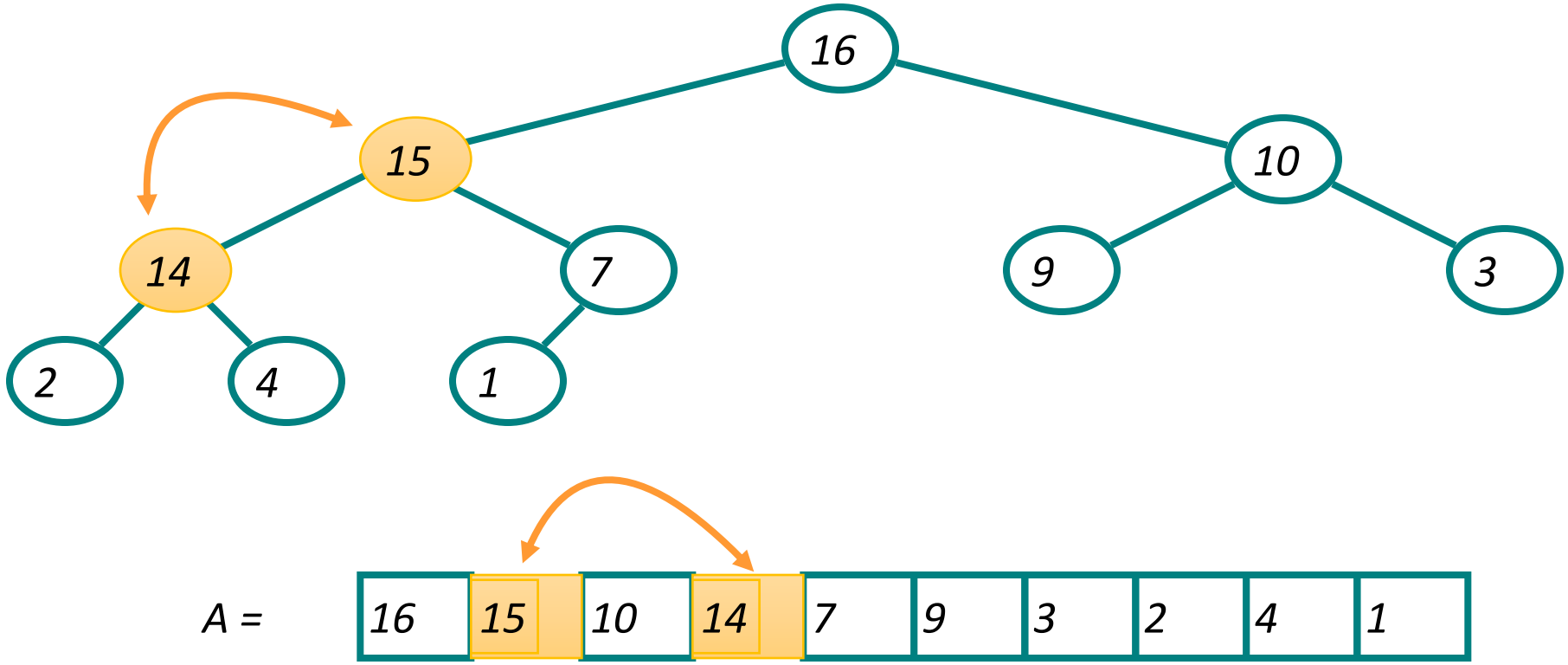
Change key value to 15



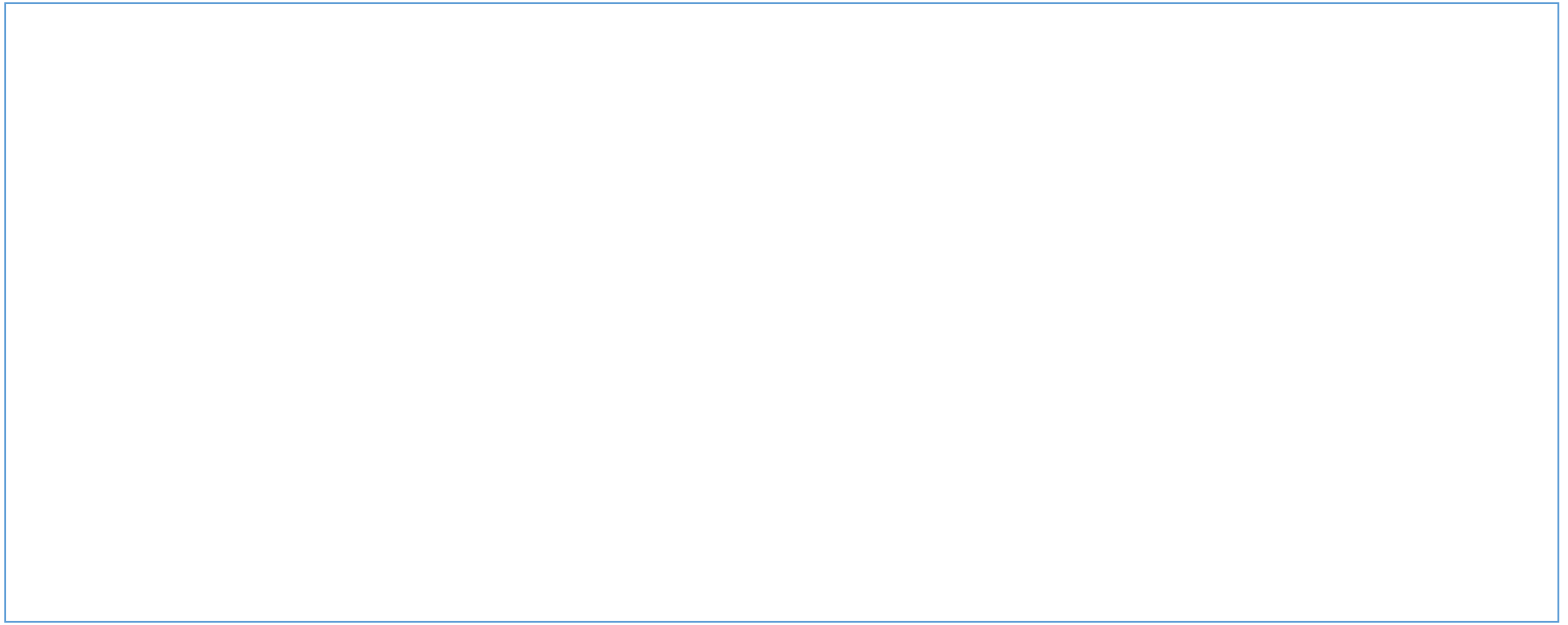


Heap Change Key

HeapChangeKey(A, 4, 15)

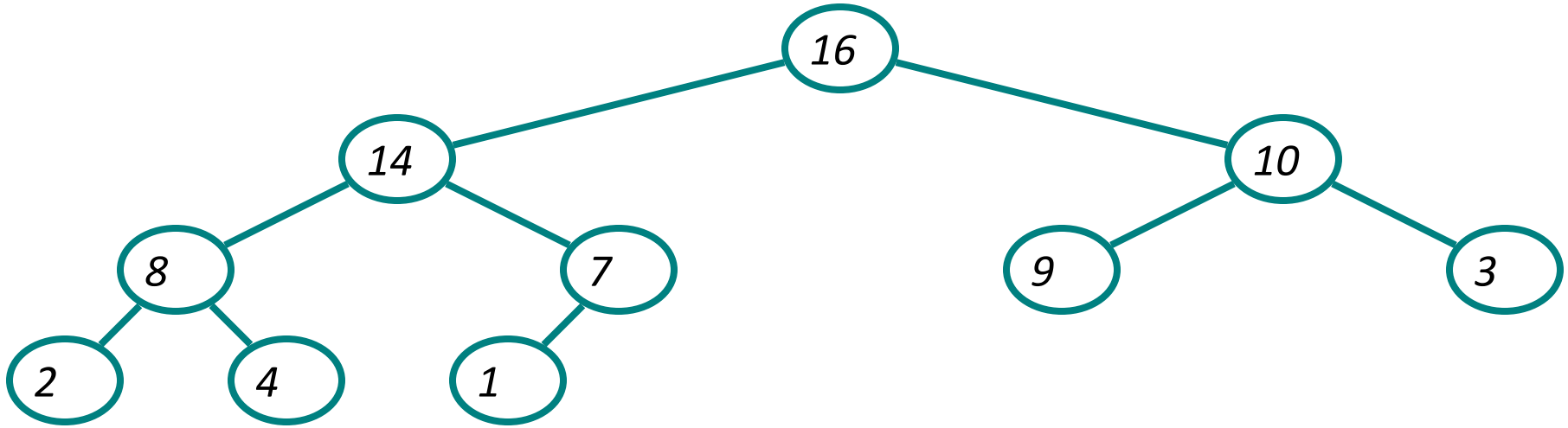


Heap Insert



Heap Insert

HeapInsert(A, 17)

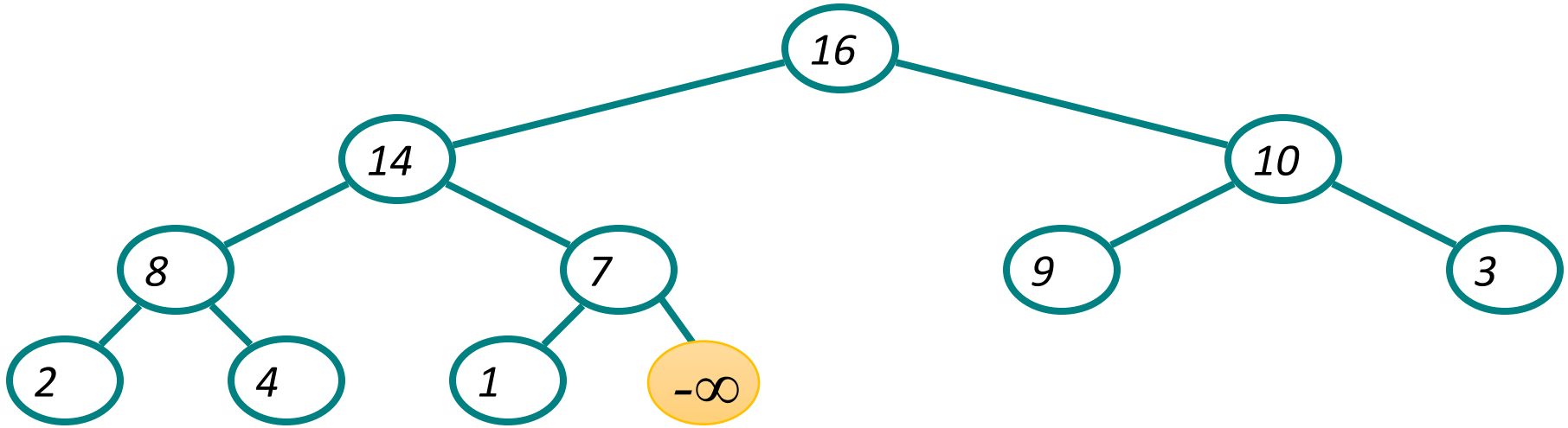


A =

16	14	10	8	7	9	3	2	4	1

Heap Insert

HeapInsert(A, 17)



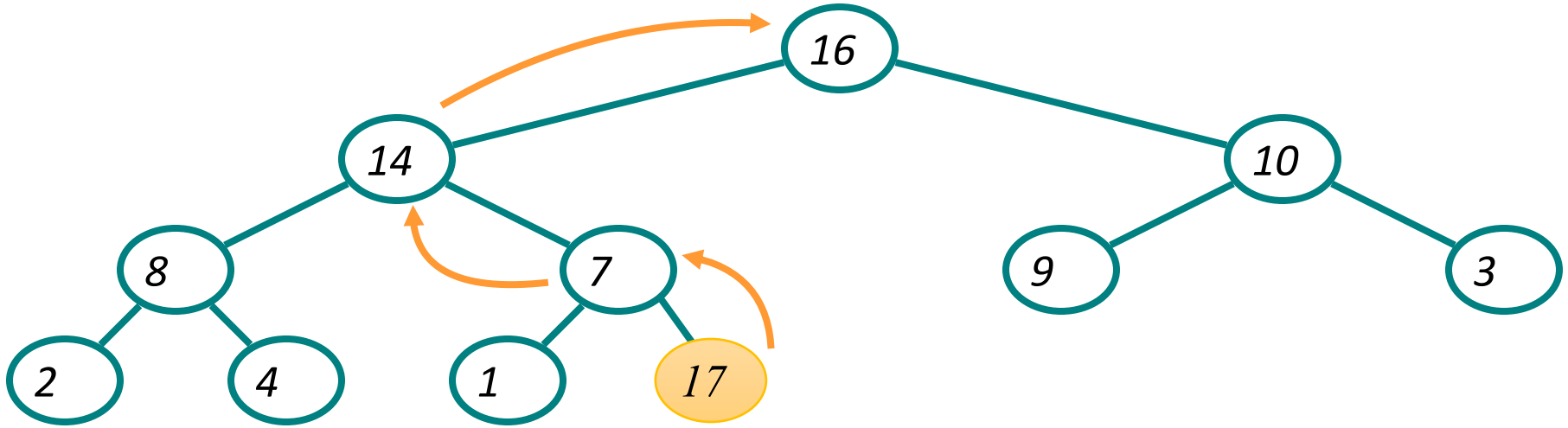
-∞ makes it a valid heap

A =

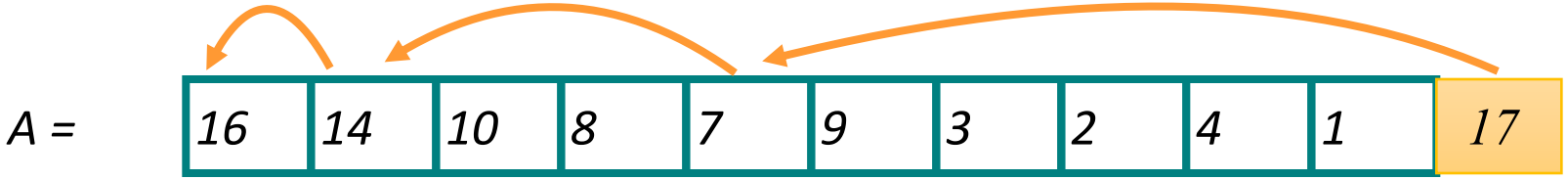
16	14	10	8	7	9	3	2	4	1	-∞

Heap Insert

HeapInsert(A, 17)

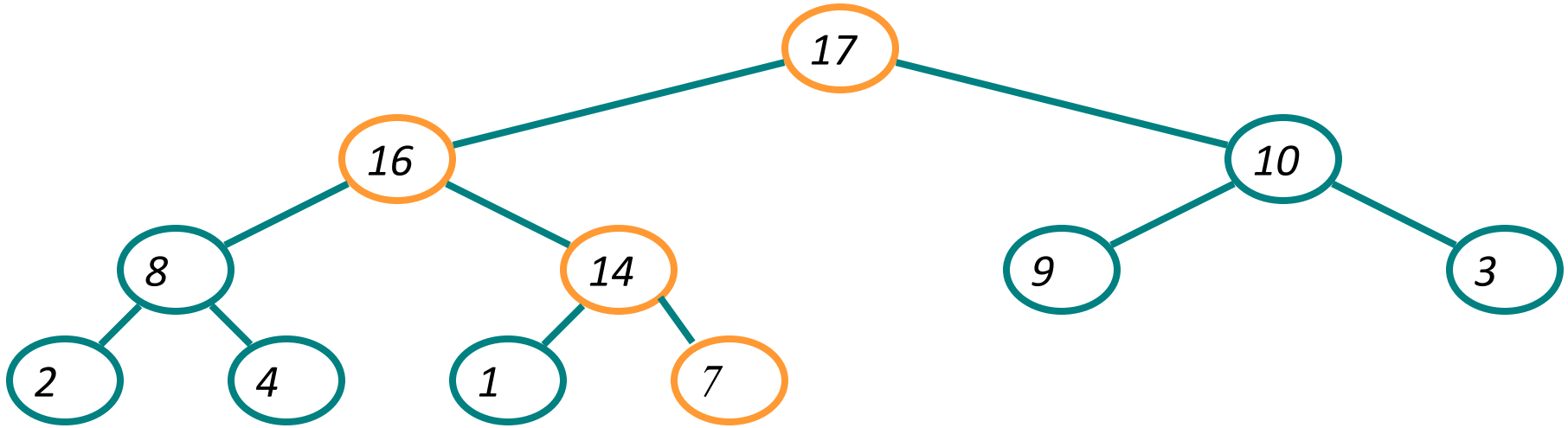


Now call HeapChangeKey



Heap Insert

HeapInsert(A, 17)



A =

17	16	10	8	14	9	3	2	4	1	7

Heap vs Array

Heap

- HeapSort: Θ(n log n)
- Maximum: Θ(1)
- ExtractMax: Θ(log n)
- ChangeKey: Θ(log n)
- Insert/Delete: Θ(log n)

- Heapify: Θ(log n)
- BuildHeap: Θ(n)

Sorted Array

- Sort: Θ(n log n)
- Maximum: Θ(1)
- ExtractMax: Θ(n) or Θ(1)
- ChangeKey: Θ(n)
- Insert/delete: Θ(n)

Quiz Preview

Q#1: For an array A of N numbers, with index starting with 1 (That is, the first number is A[1]). Assume N is an even number. Which of the following is NOT correct?

a) An inversely sorted array of numbers is always a heap. for example: A={9,8,7,6, ..., 1 }
b) All the numbers after A[N/2] must satisfy the heap property.
c) All the numbers after A[N/2] have no children.
d) All the numbers before A[N/2] have two children.

Quiz Preview

Q#2: Heapsort: Consider the following example.

Array:	7	5	6	1	3	2	4
index:	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Step 1. Swap root A[1] with the last node A[7].
Decrease heap size by one.
Heapfiy the new root node.

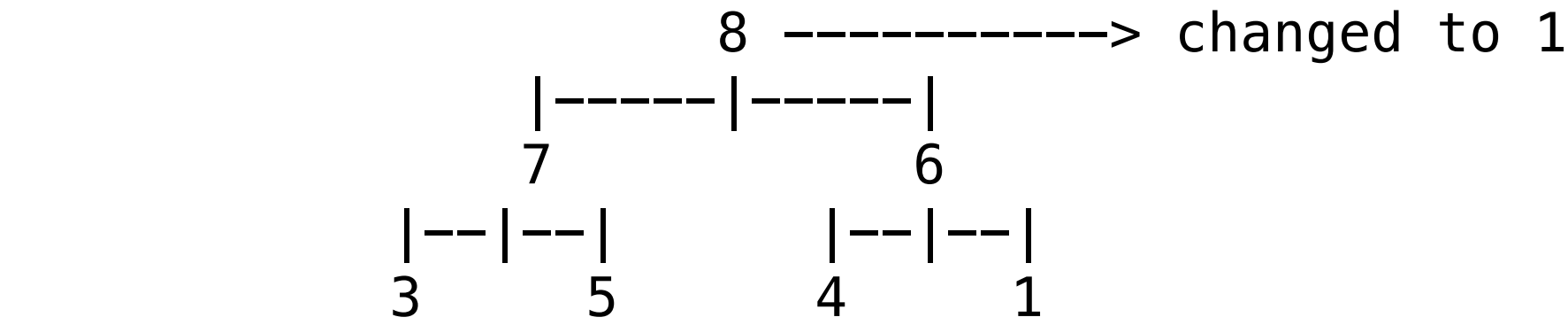
What will the array look like at step 1? Answer: _____ (1pt)

Step 2.Swap the new root node with the last node A[6]=2.
Decrease heap size by one.
Heapfiy the new root node.

What will the array look like at step 2? Answer: _____ (1pt)

Quiz Preview

Q#4: Show the result of the following example where the key of root node A[0] is decreased to 1 from 10.



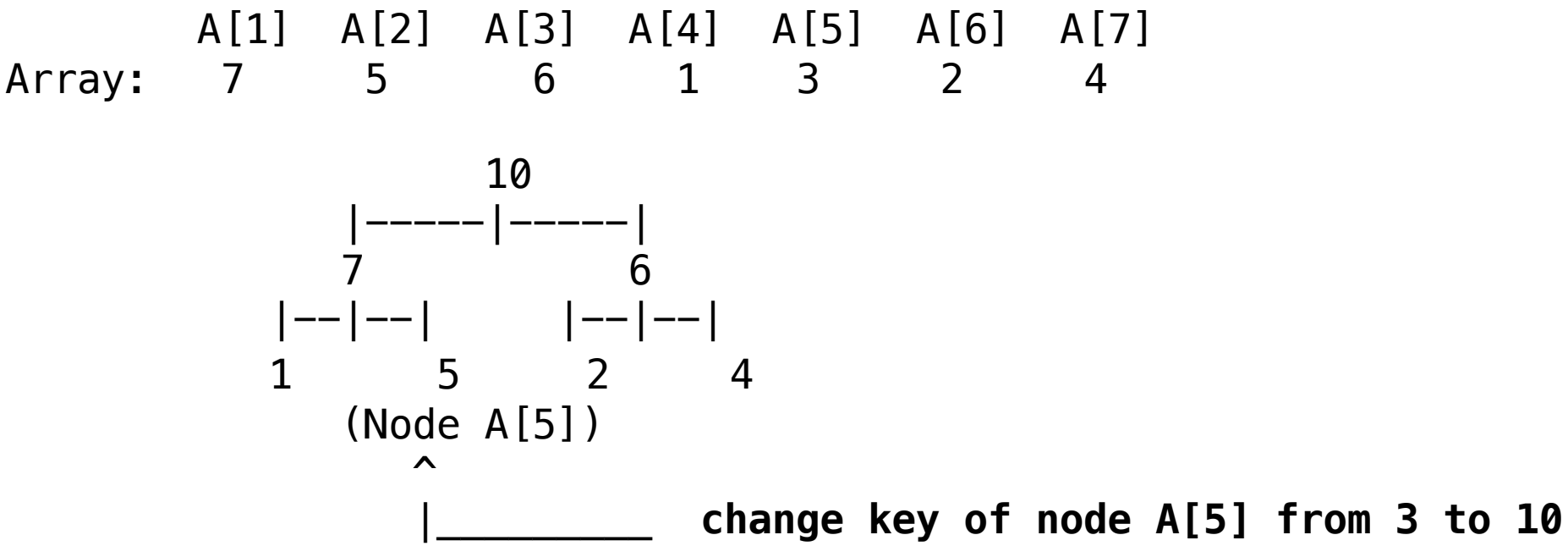
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
10	7	8	3	5	4	6

What will the resulting array look like?
Answer: _____ (1pt)

Quiz Preview

Q#5: In the following example of a heap, the key of node A[5] is increased from 3 to 10. The violation of the heap property caused by this key change can be fixed by swapping nodes in a manner of bubble up.

What will the new array A for this new heap look like?
Answer: _____ (1pt)



Quiz Preview

Q#1: For an array A of N numbers, with index starting with 1 (That is, the first number is A[1]). Assume N is an even number. Which of the following is NOT correct?

a) An inversely sorted array of numbers is always a heap. for example: A={9,8,7,6, ..., 1 }
b) All the numbers after A[N/2] must satisfy the heap property.
c) All the numbers after A[N/2] have no children.
d) All the numbers before A[N/2] have two children. Answer: ? (1pt)

Quiz Preview

Q#2: Heapsort: Consider the following example.

Array: 7 5 6 1 3 2 4
index: A[1] A[2] A[3] A[4] A[5] A[6] A[7]

Step 1. Swap root A[1] with the last node A[7]. Decrease heap size by one. Heapfiy the new root node.
Step 2. Swap the new root node with the last node A[6]=2. Decrease heap size by one. Heapfiy the new root node.
What will the array look like at step 1?

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
a)	6	5	4	1	3	2	(7)
b)	4	5	6	1	3	2	(7)
c)	5	4	6	1	3	2	(7)
d)	7	5	6	1	3	2	(7)

Answer: _____?_____(1pt)

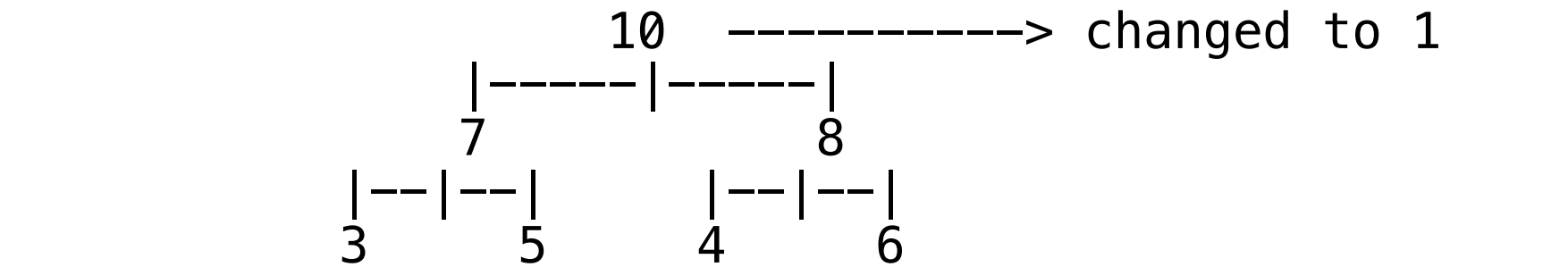
Q#3 What will the array look like at step 2?

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
a)	5	3	4	1	2	6	7
b)	4	5	2	1	3	6	7
c)	5	4	2	1	3	6	7
d)	4	5	2	1	3	7	6

e) None of the above is correct

Answer: _____?_____(1pt)

Q#4: Show the result of the following example where the key of root node A[0] is decreased to 1 from 10.



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

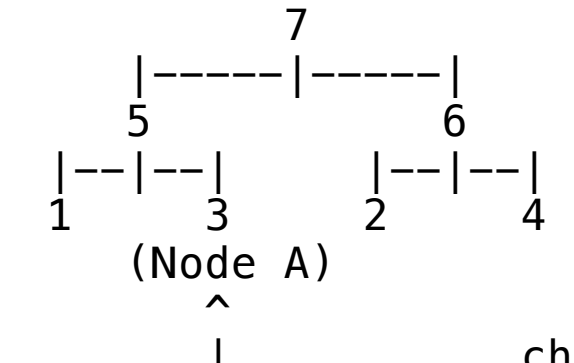
What will the resulting array look like?

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
a)	8	7	1	3	5	4	6
b)	8	7	6	3	5	1	4
c)	8	7	6	3	5	4	1

Answer: _____?_____(1pt)

Q#5: In the following example of a heap, the key of node A[5] is increased from 3 to 10. The violation of the heap property caused by this key change can be fixed by swapping nodes in a manner of bubble up. What will the new array A for this new heap look like?

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
Array:	7	5	6	1	3	2	4



change key of node A[5] from 3 to 10

		A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
a)	Array:	10	7	6	1	5	2	4
b)	Array:	7	5	6	1	10	2	4
c)	Array:	7	10	6	1	5	2	4
d)	Array:	10	7	6	5	1	2	4

Answer: _____?_____(1pt)