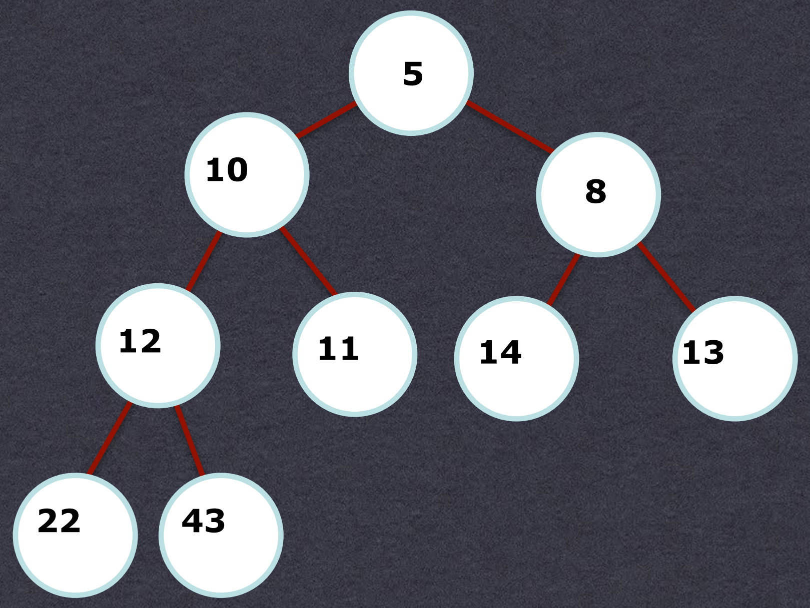
- This weekend (Nov 11-12), YHack will be hosting a 24-hour hackathon
- Registration link: tiny.cc/yminihack
- Registration deadline: Nov 9
- Additional information: yhackmini.org

Priority Queues and Heaps



Revisiting Queues

- How do people board a plane?
  - Mainly a queue but “not everyone is the same”
  - First class, frequent flyers, families, …

Airplane boarding example

- Passengers join the “queue”

    void enqueue(key, person)

- Need to know first in line
- How to compare?

    Priority, arrival time, time, …

Complicated formula!

On priorities

- Combination of traits
  - Fare
  - Arrival time
  - Frequent flyer status, etc…
- Need to “compare” people

Back to queues

- Three fundamental operations

    insert(key, element) = ENQUEUE

    min_element()

    remove_min() = DEQUEUE

Example

Operation	***Output***	***Priority Queue***
insert(5, A)	***-***	***{(5, A)}***
insert(9, C)	***-***	***{(5, A), (9, C)}***
insert(3, B)	***-***	***{(5, A), (9, C), (3, B)}***
insert(7, D)	***-***	***{(5, A), (9, C), (3, B), (7, D)}***
min_element()	***B***	***{(5, A), (9, C), (3, B), (7, D)}***
min_key()	***3***	***{(5, A), (9, C), (3, B), (7, D)}***

Inserting in an unsorted container

- Insert?

  Array List/Linked list insertion

  O(1) unless we need to expand

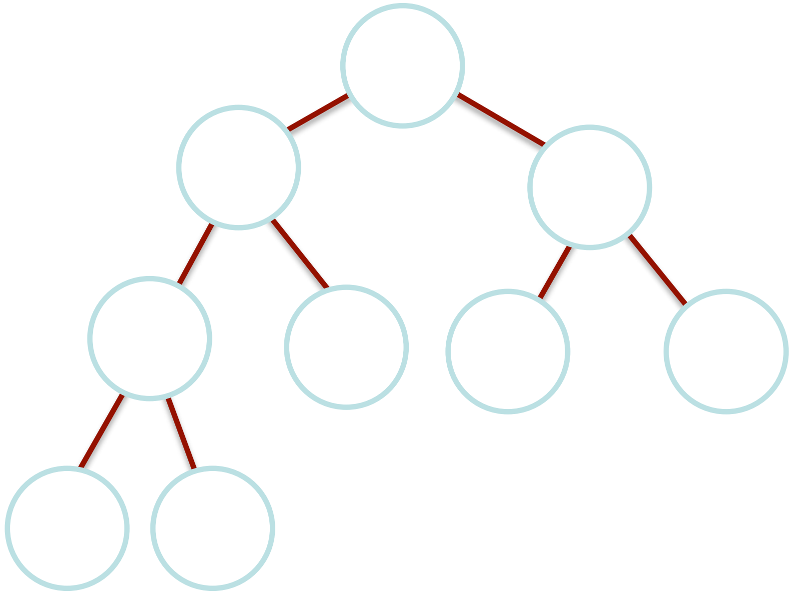
- min_element?min_key?remove_min()?

  Search through whole list and find smallest

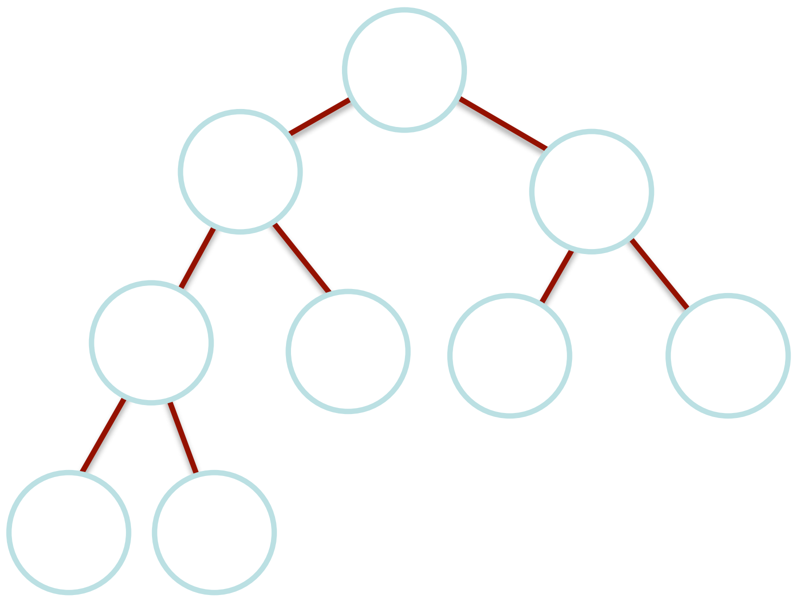
  O(n) **always**

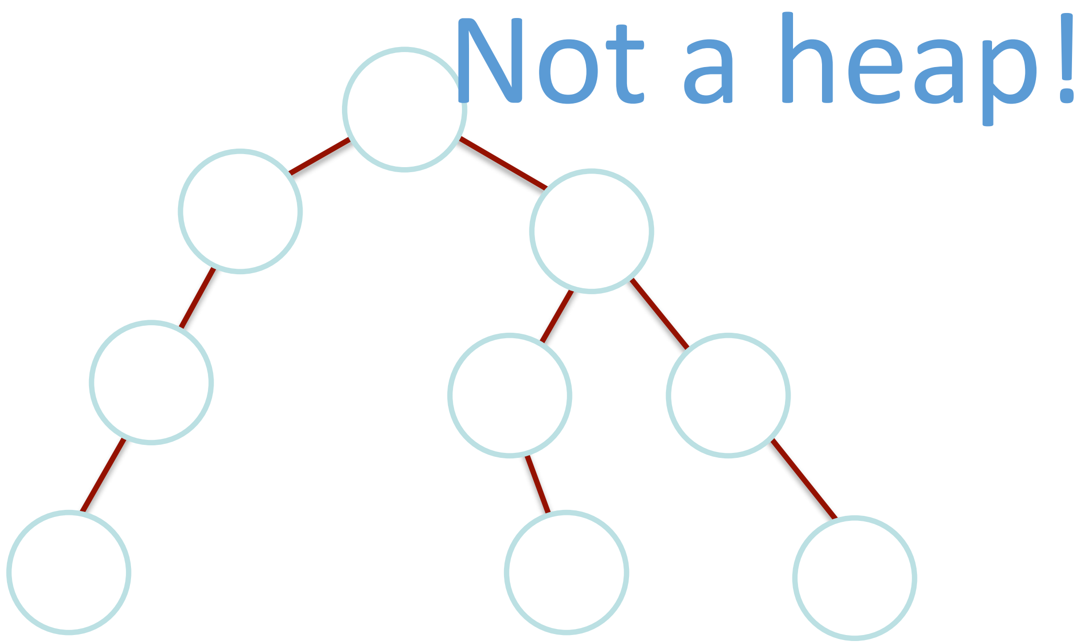
Introducing Heaps

- Tree based data structure
  - Root, parent, child, leaf
  - Binary Tree
    - At most two children



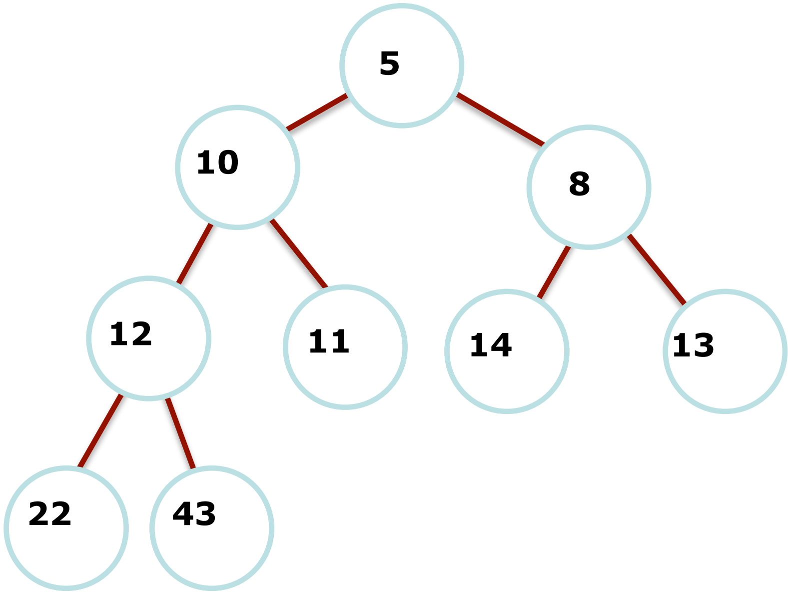
Shape invariant

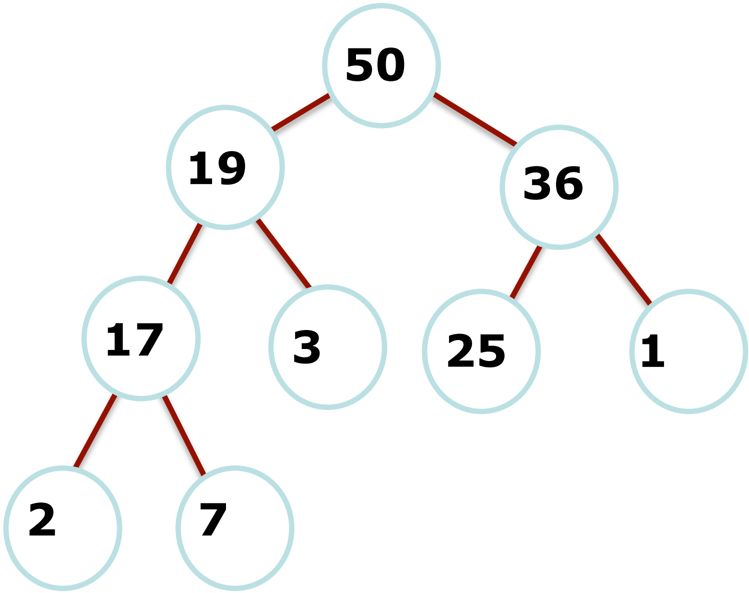




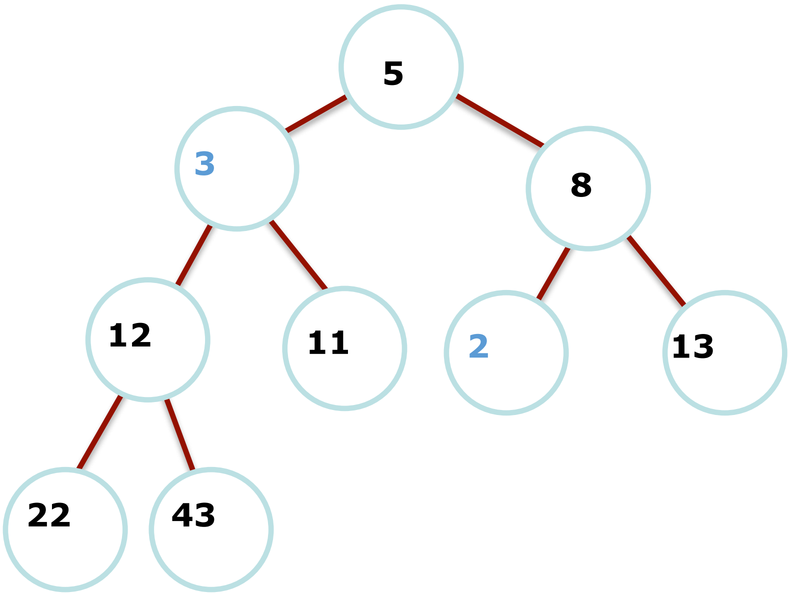
- Nodes filled from **left** to **right** and **top** to **bottom**

Value invariant



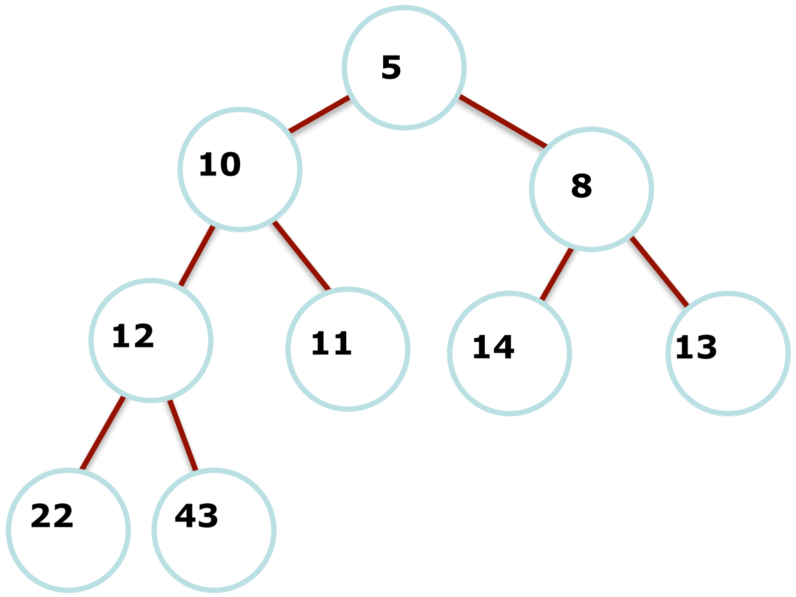


Not a heap!

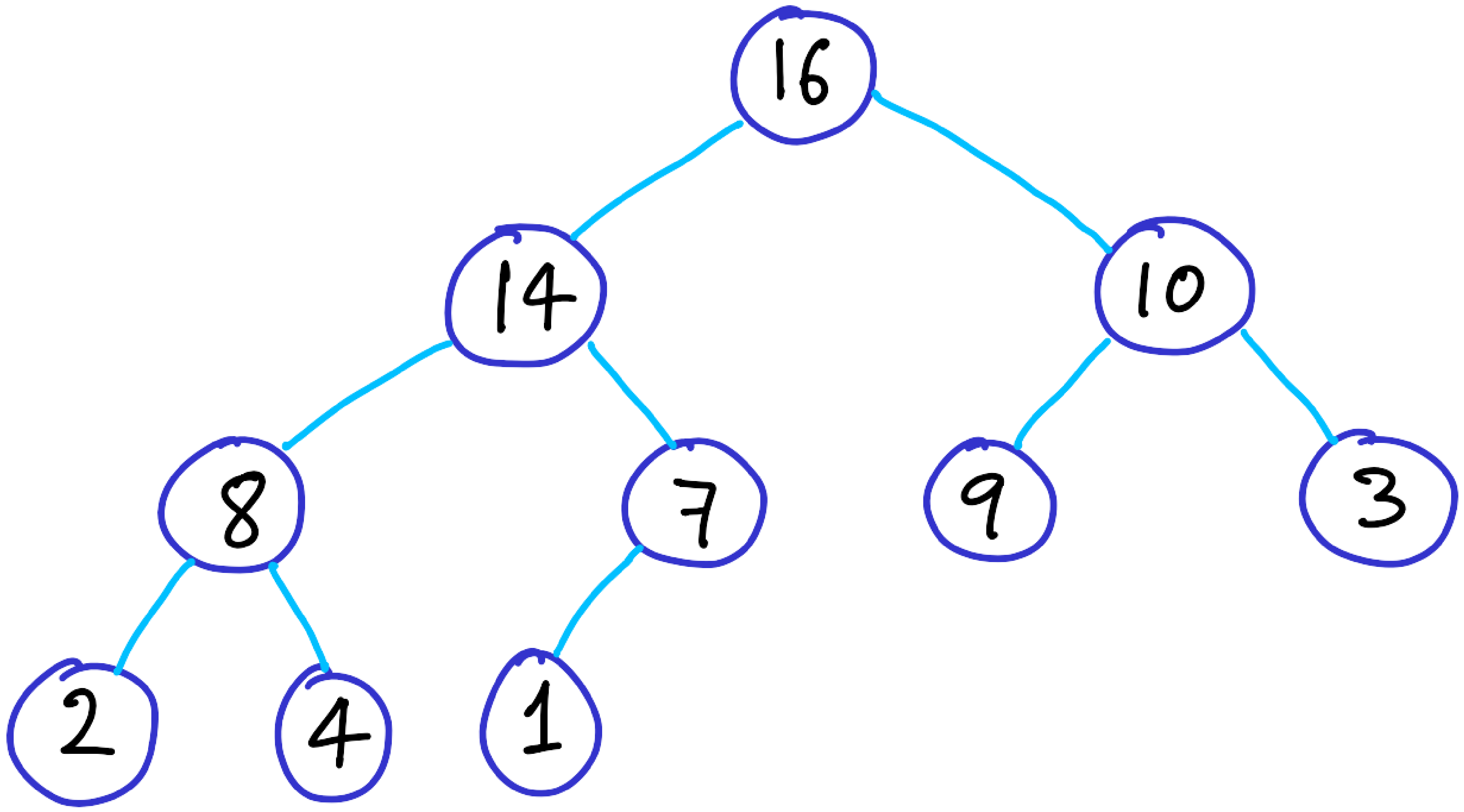


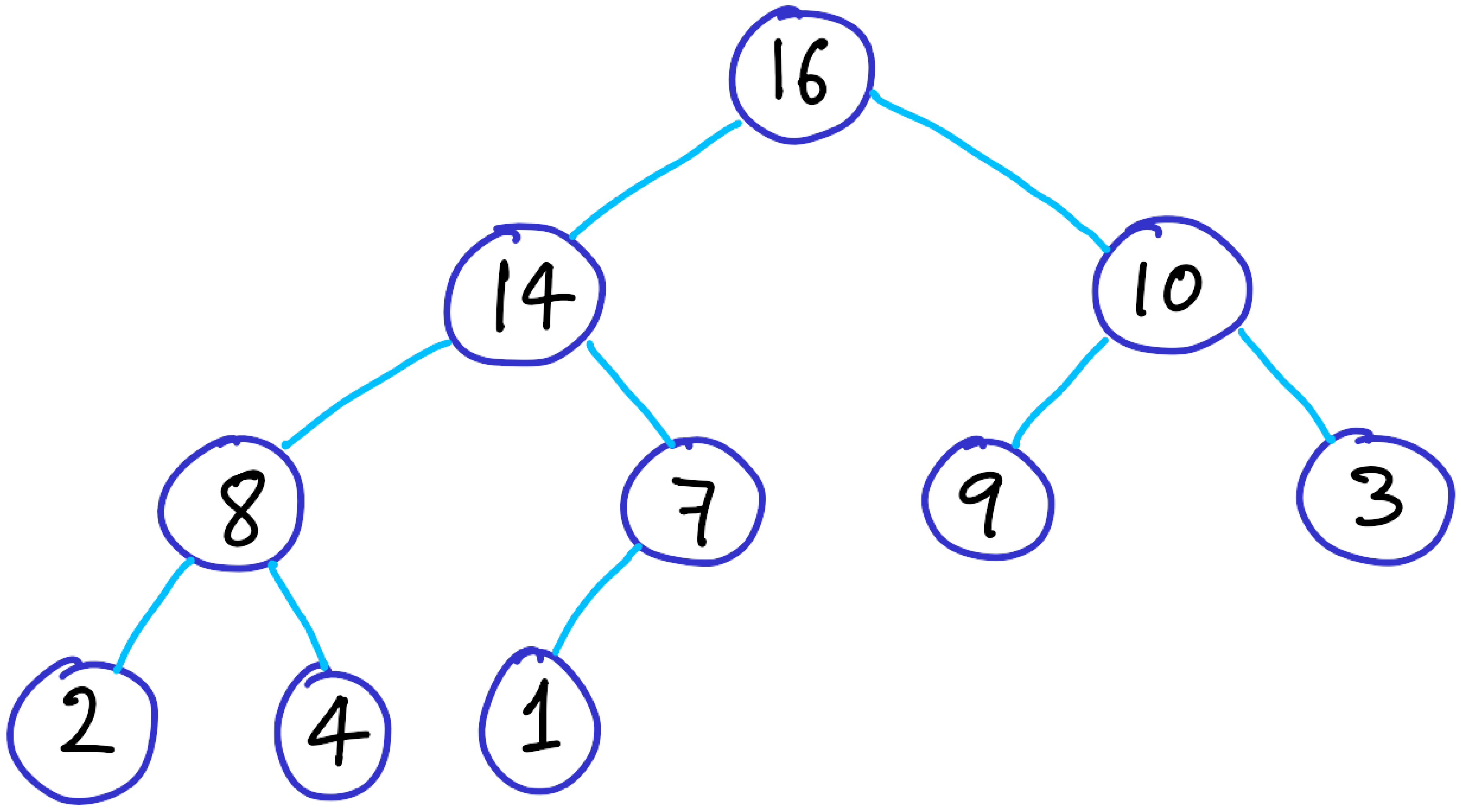
- Option 1: Parent is **smaller** than both children (minHeap)
- Option 2: Parent is **larger** than both children (maxHeap)

Usage?



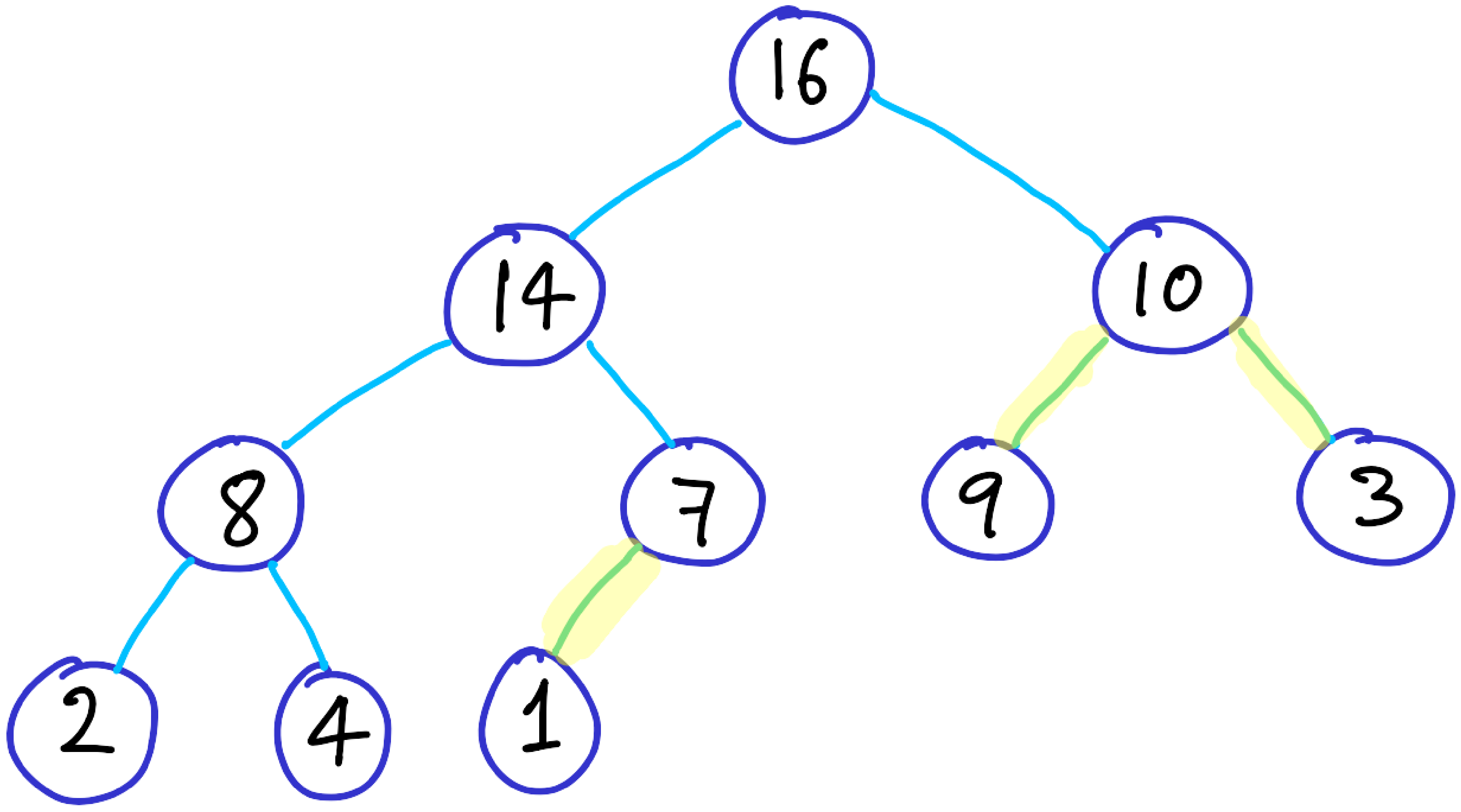
- Super easy to find minimum in minHeap (maximum in maxHeap)
  - Opposite Is hard
- Simple DS, should be easy to update


16
14
10
8
7
9
3
2
4
1



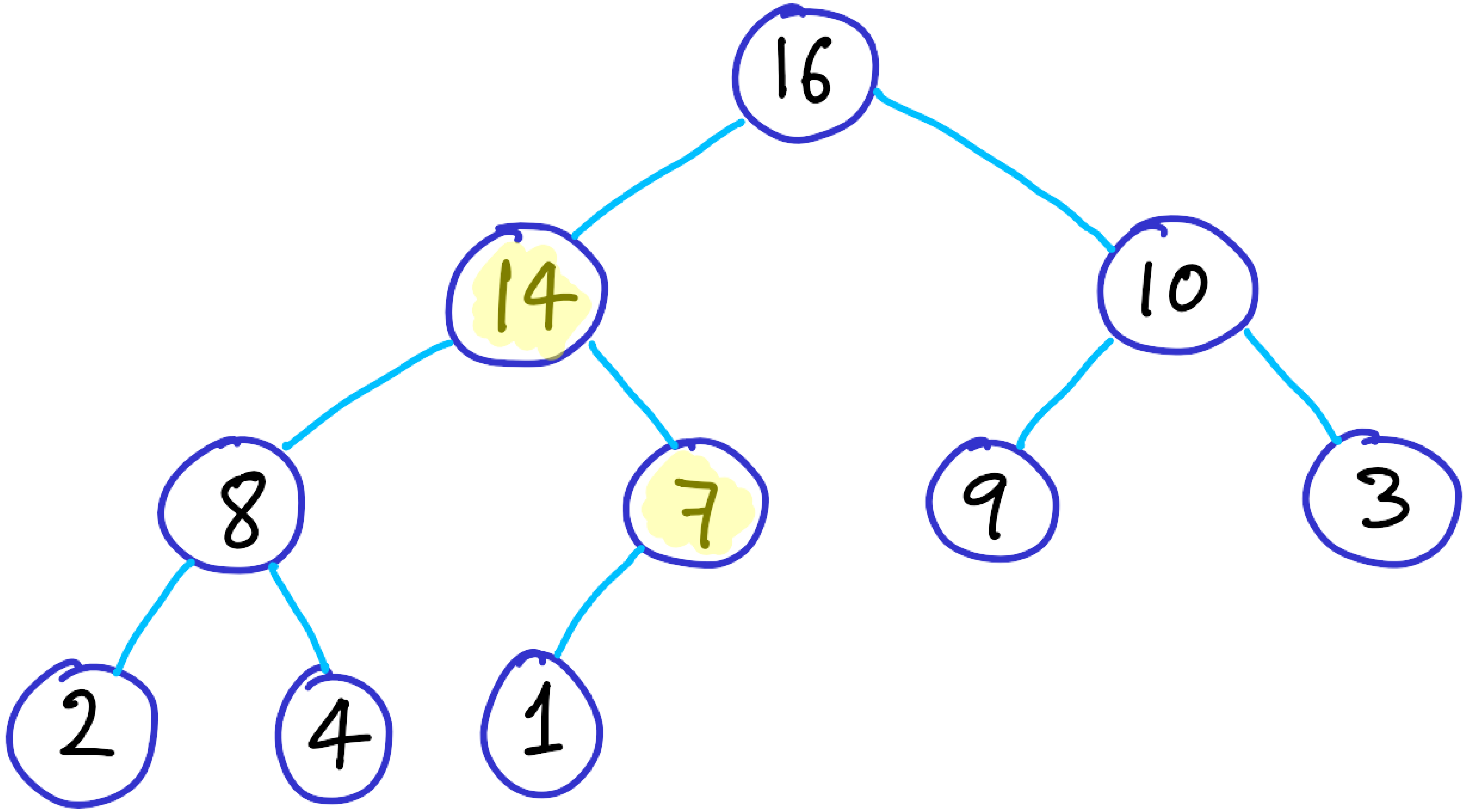
Rules:

- binary
- max
- complete



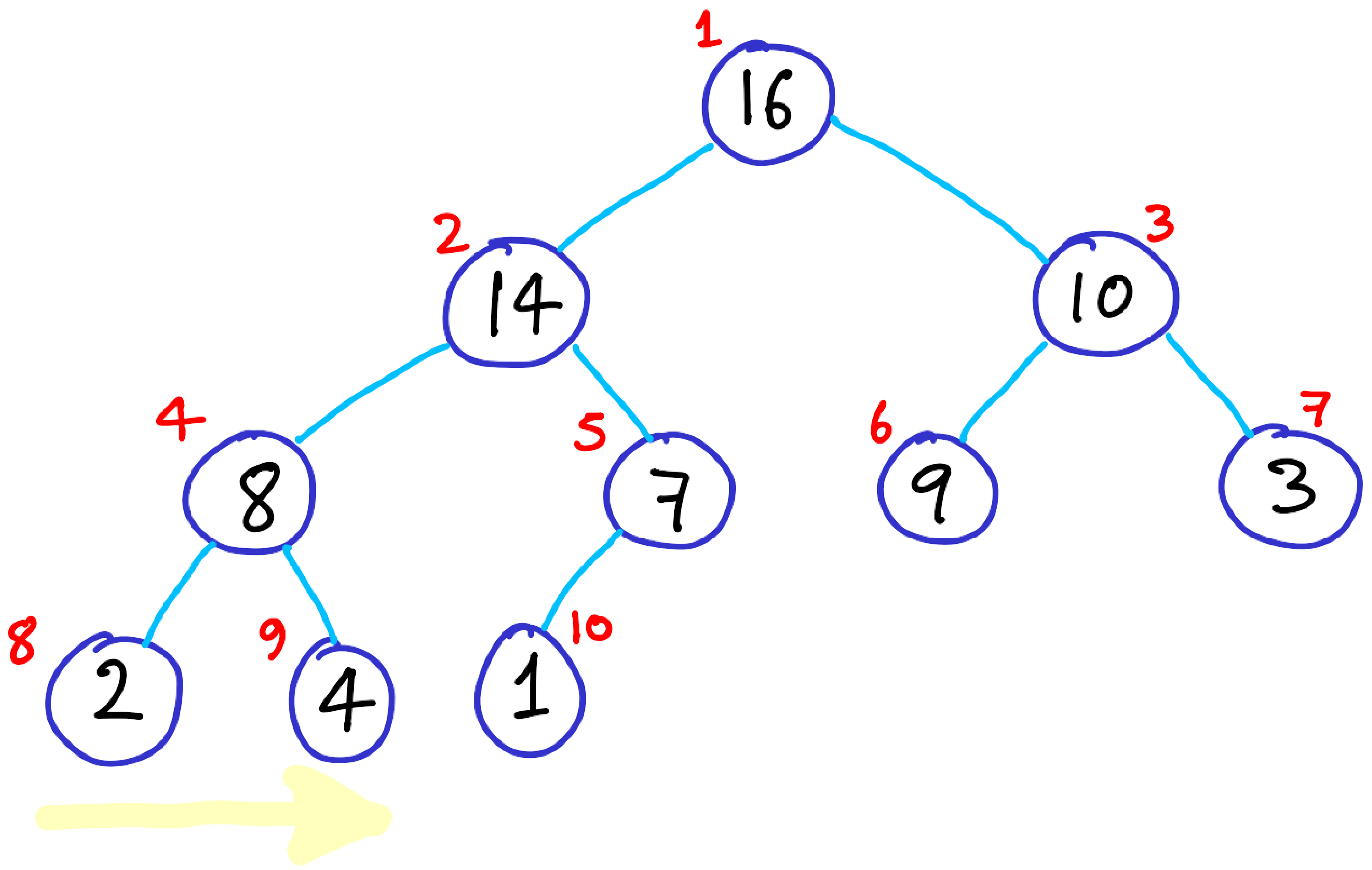
Rules:

- binary: internal nodes have 1 or 2 children
- max
- complete



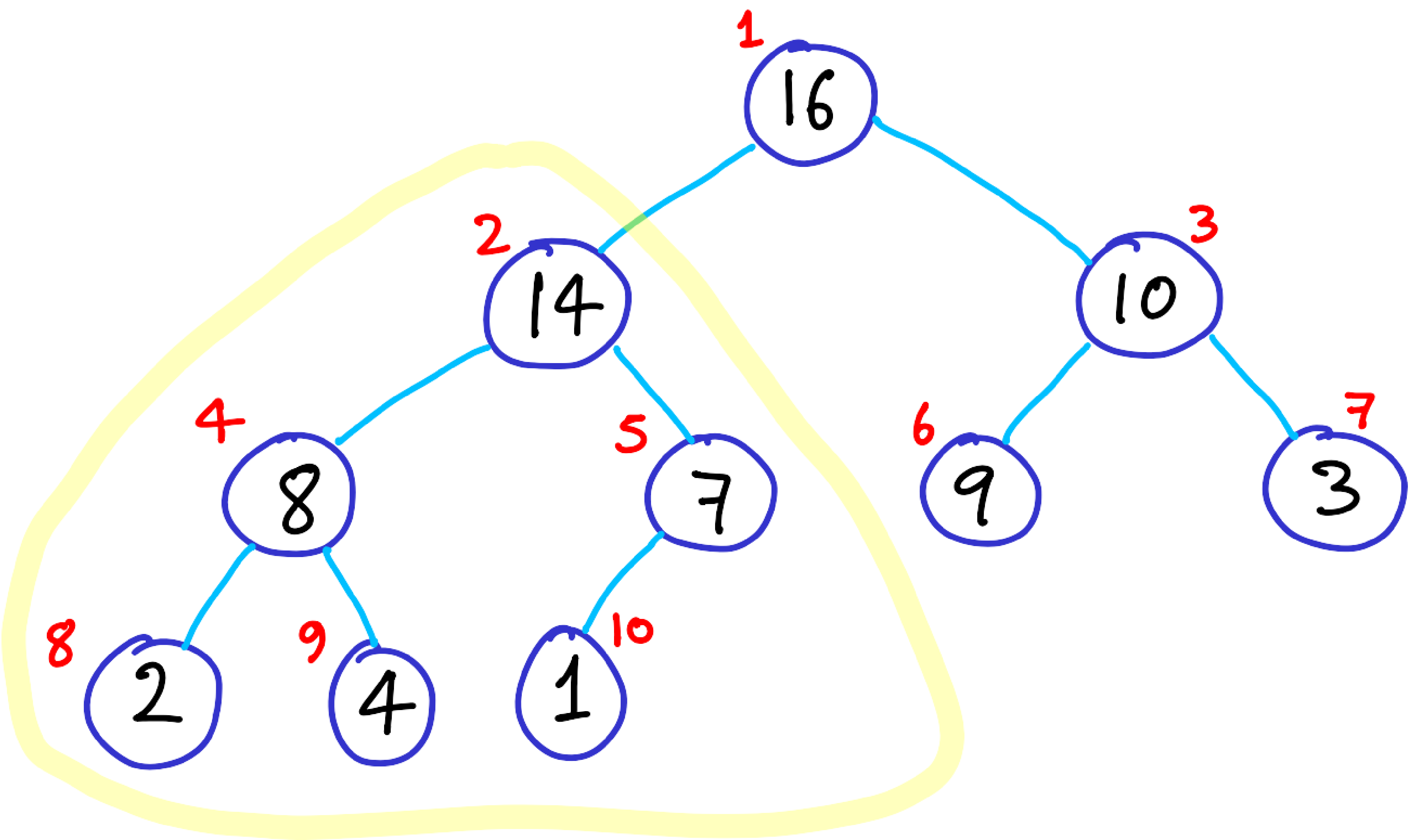
Rules:

- binary: internal nodes have 1 or 2 children
- max: parent ≥ child
- complete



Rules:

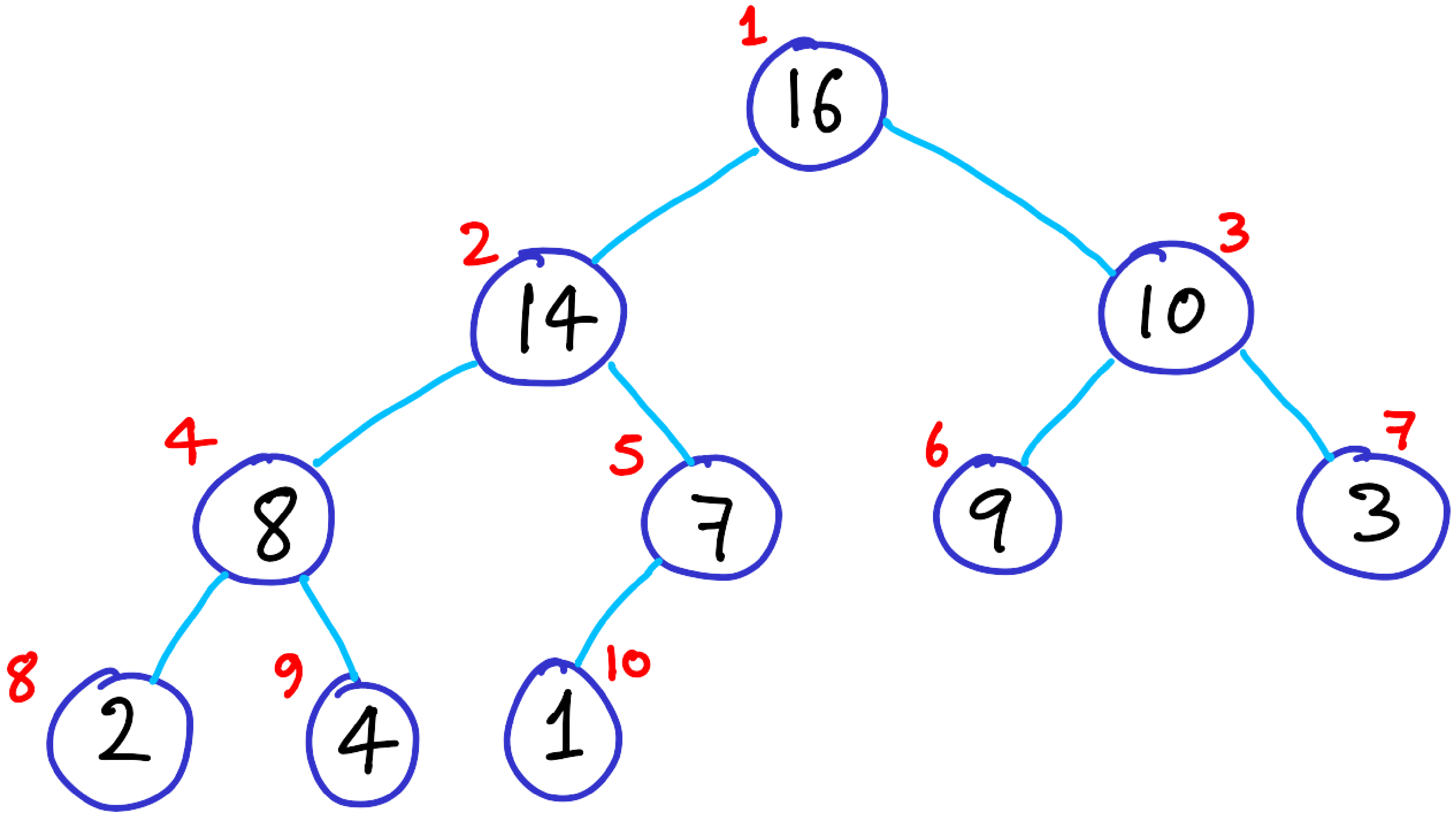
- **binary:** internal nodes have 1 or 2 children
- **max:** parent ≥ child
- **complete:** all levels filled (lowest can be partial, left to right)



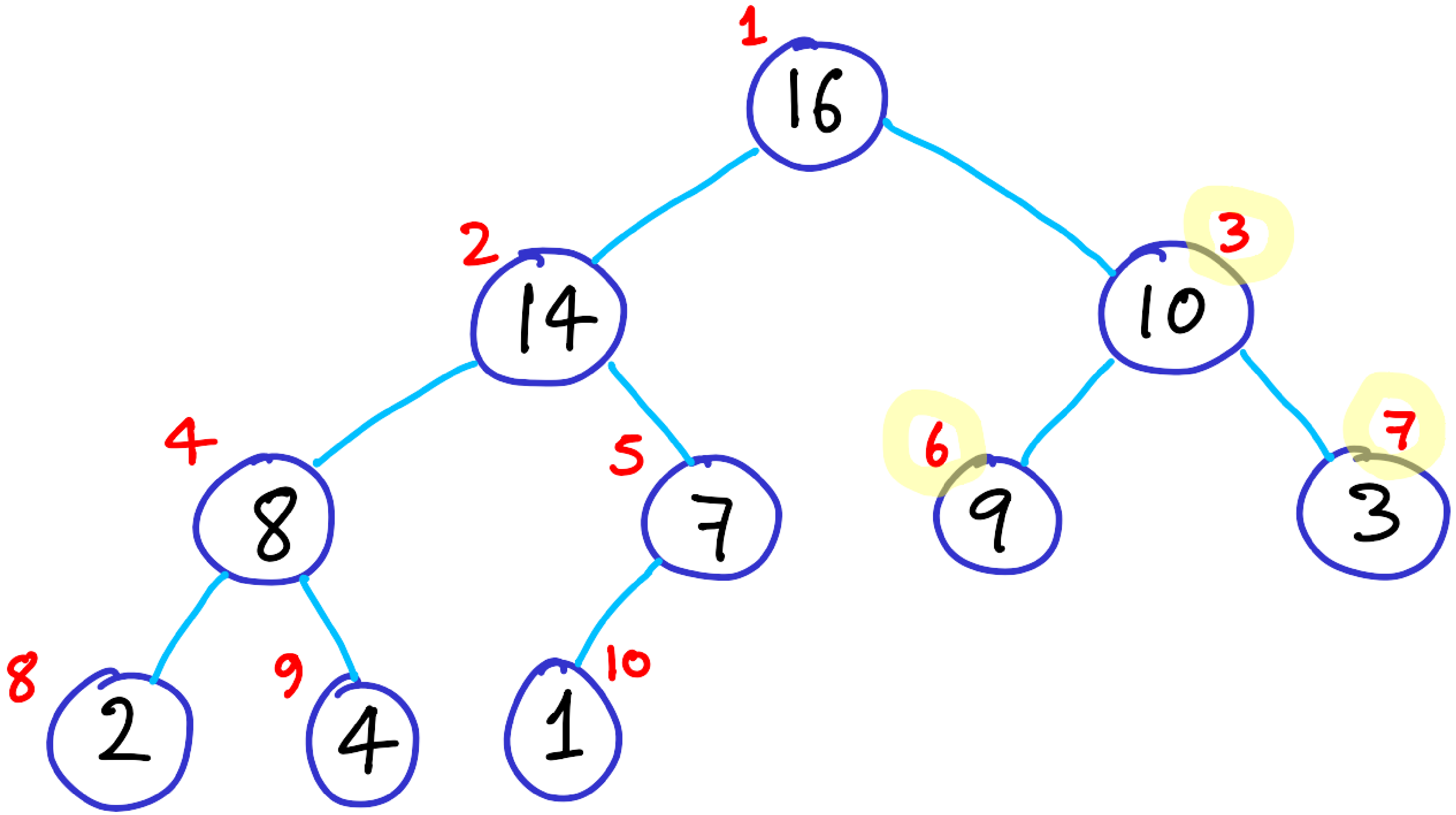
[Notice every subtree is also a heap]

Rules:

- **binary:** internal nodes have 1 or 2 children
- **max:** parent ≥ child
- **complete:** all levels filled (lowest can be partial, left to right)



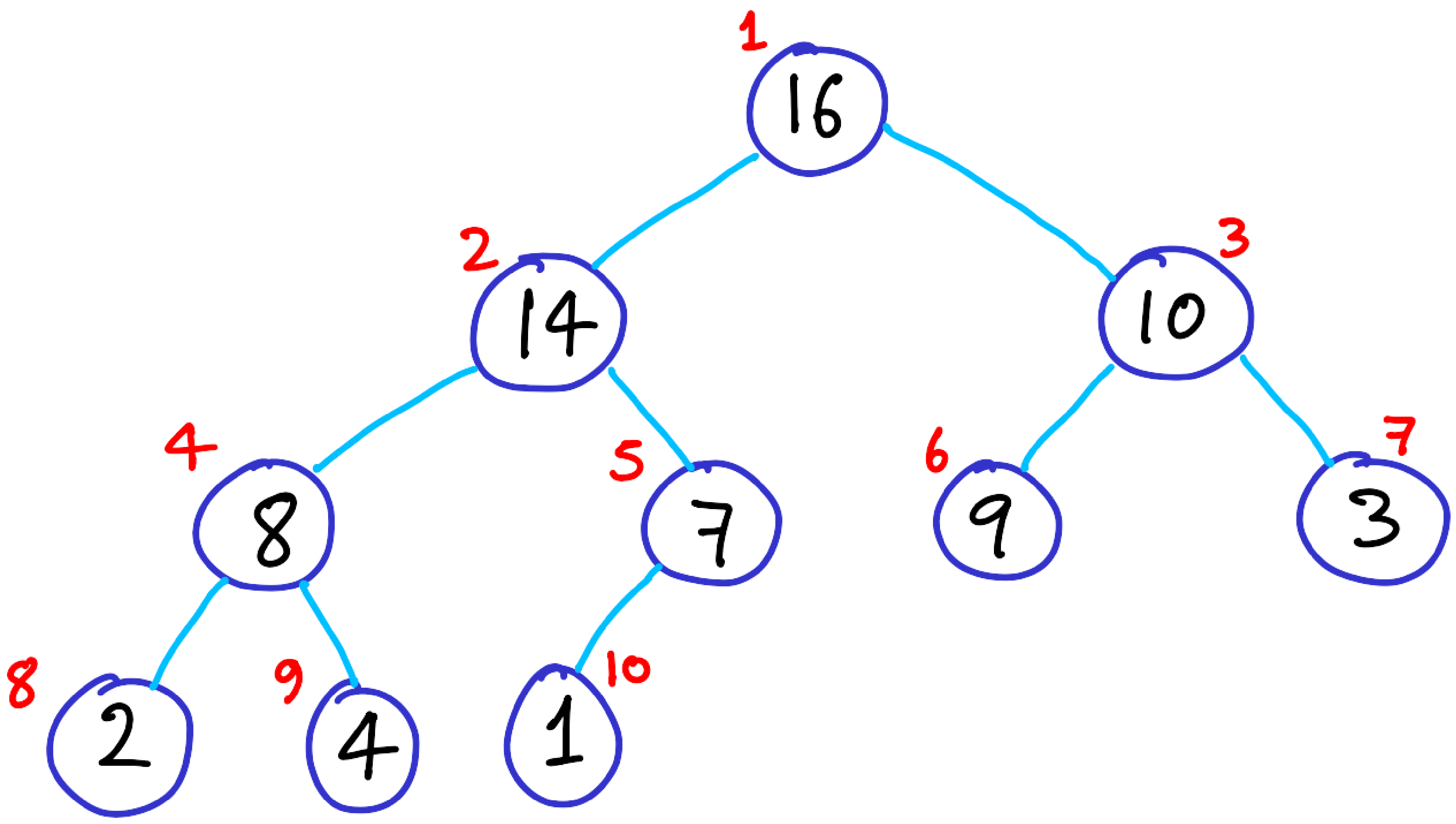
How can we identify the indices of the children of a given node?



How can we identify the indices of the children of a given node?

left-child(i) = 2i

right-child(i) = 2i+1

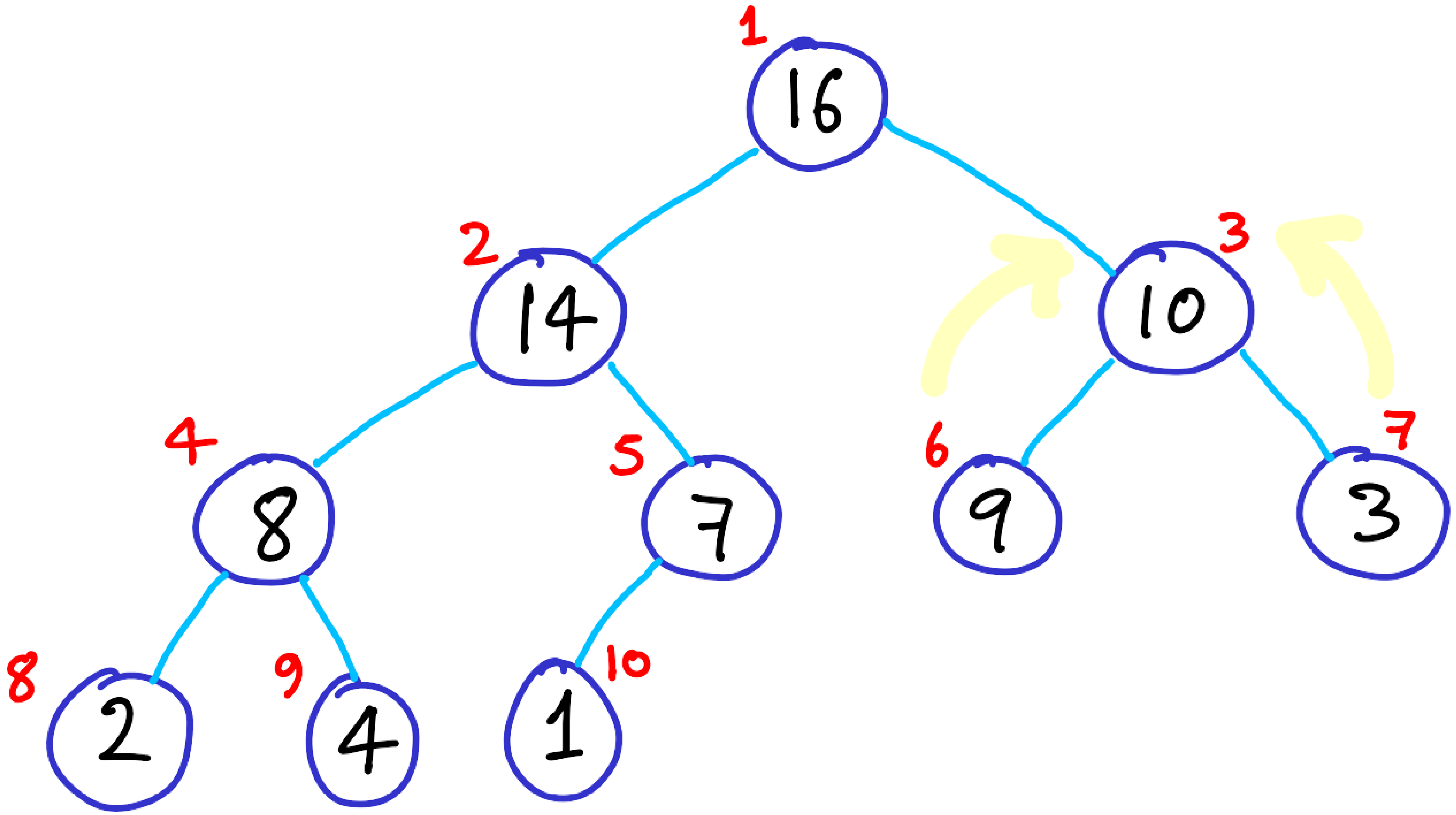


How can we identify the indices of the children of a given node?

left-child(i) = $2i$

right-child(i) = $2i+1$

parent(i) = ?



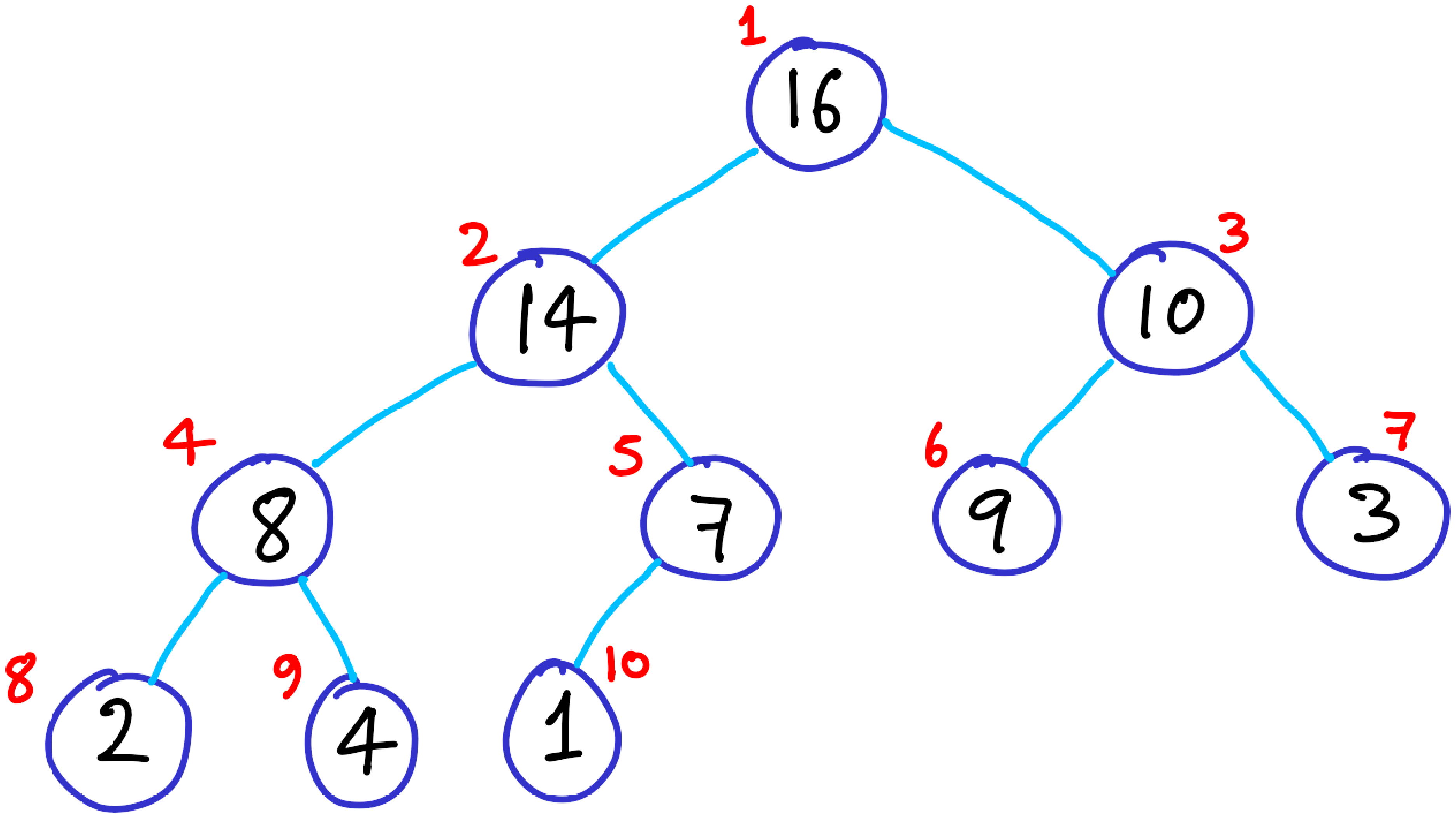
How can we identify the indices of the children of a given node?

left-child(i) = $2i$

right-child(i) = $2i+1$

parent(i) = $\lfloor i/2 \rfloor$

How can we identify the indices of the children of a given node?



left-child(i) = $2i$

right-child(i) = $2i+1$

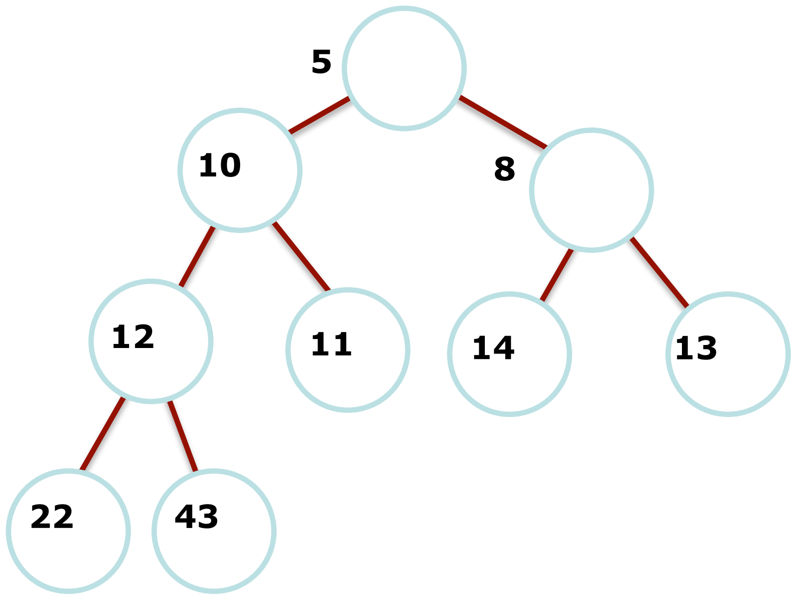
parent(i) = $\lfloor i/2 \rfloor$

Use array to store heap

(avoid wasting space with pointers)

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

minElement()

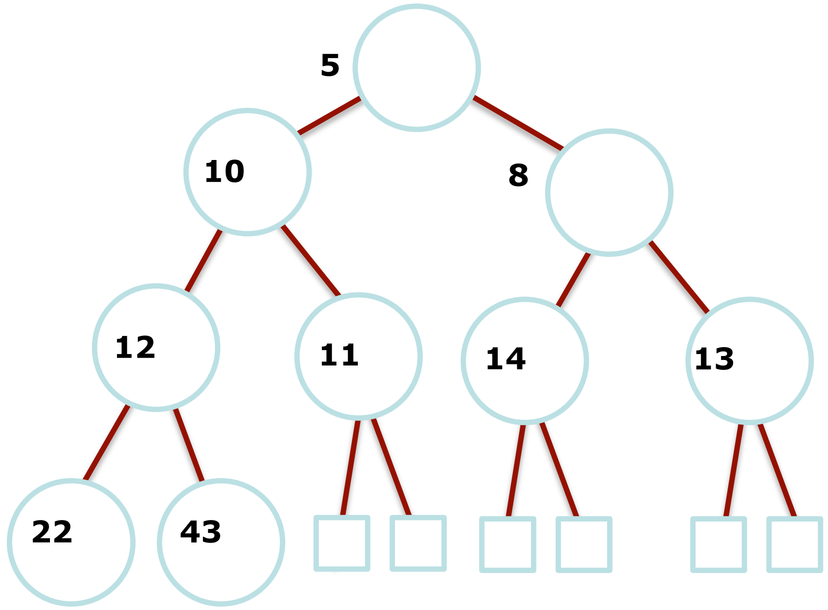


- Just return top of heap

Just an array!

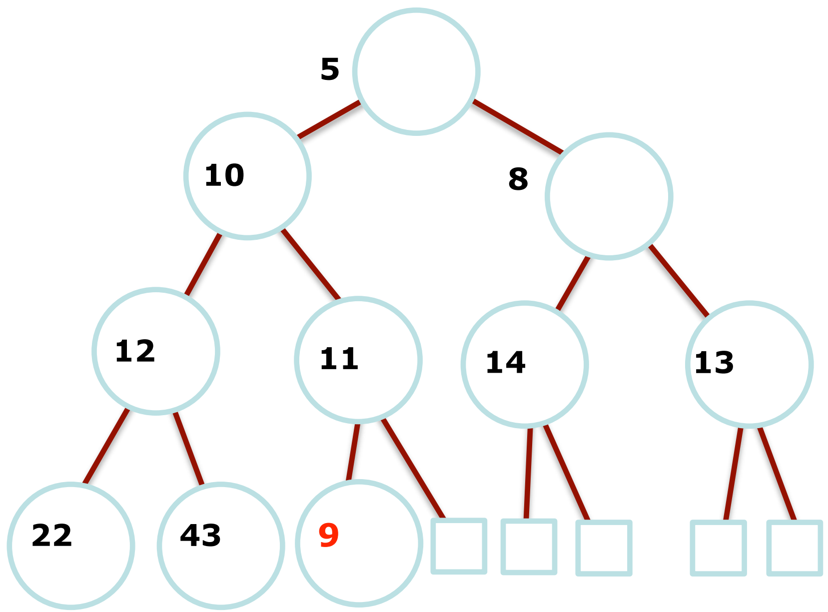
Return h[1]

Insert



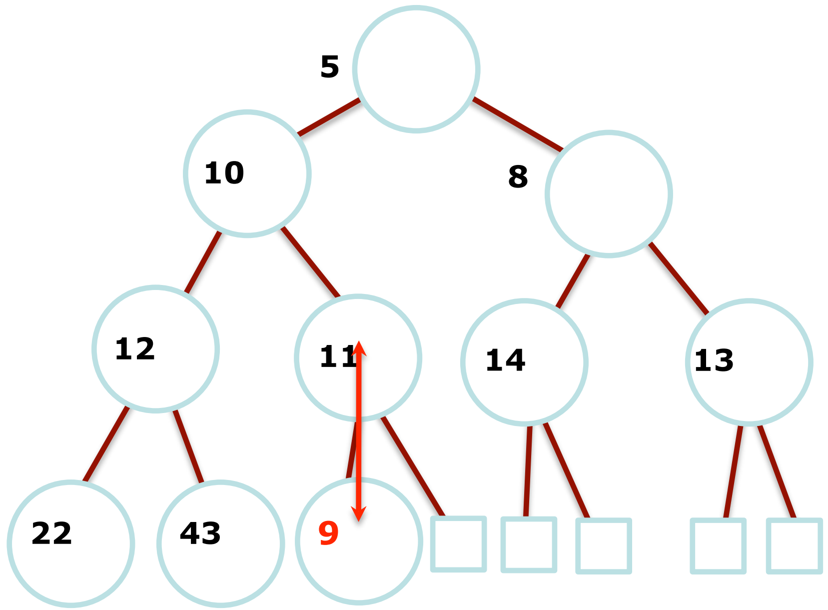
- The shape invariant tells us where to insert
- How to preserve the value invariant?
- Need to **float** the number

Example: insert(9)



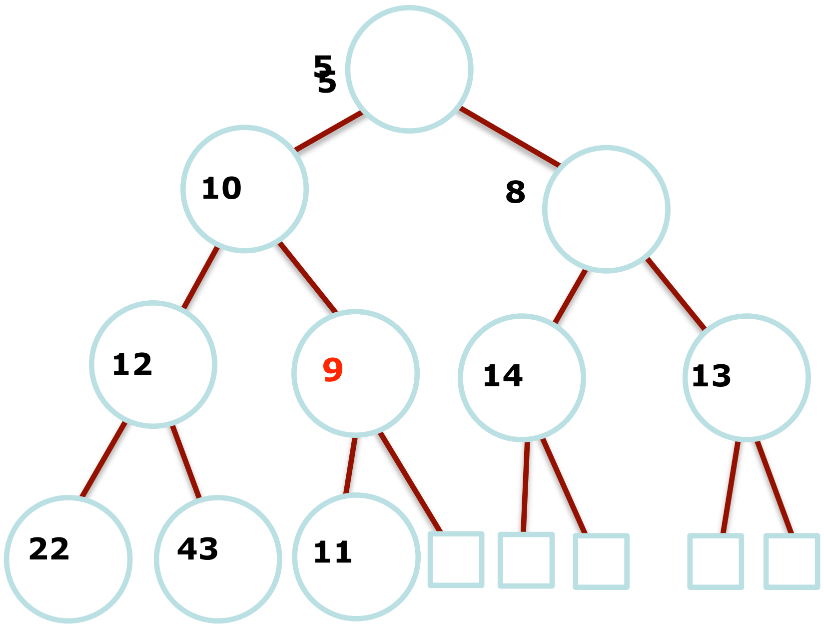
- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to **float** the number

Example: insert(9)



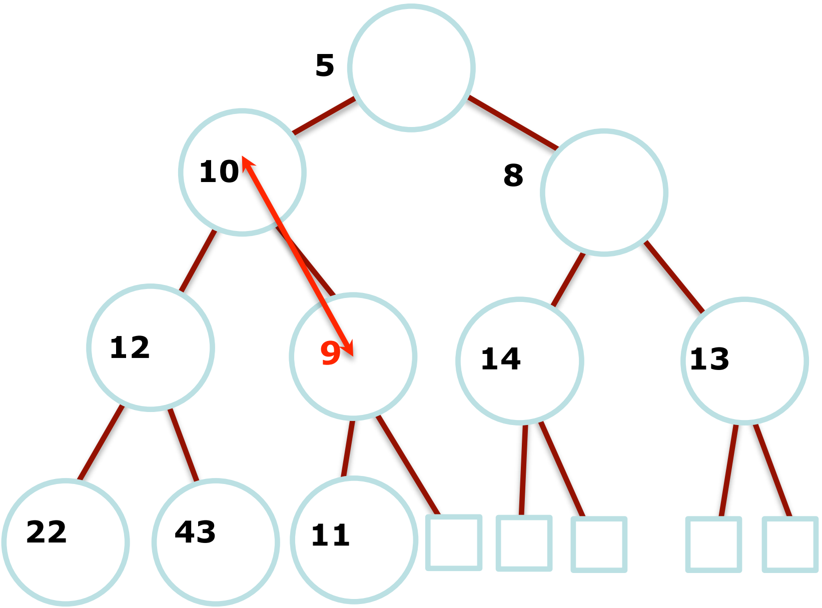
- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to **float** the number

Example: insert(9)



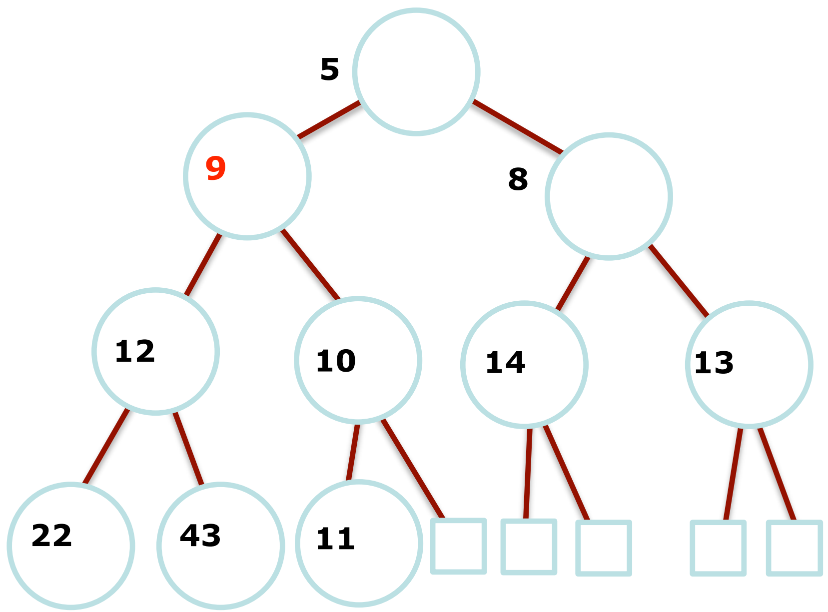
- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to **float** the number

Example: insert(9)



- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to **float** the number

Example: insert(9)



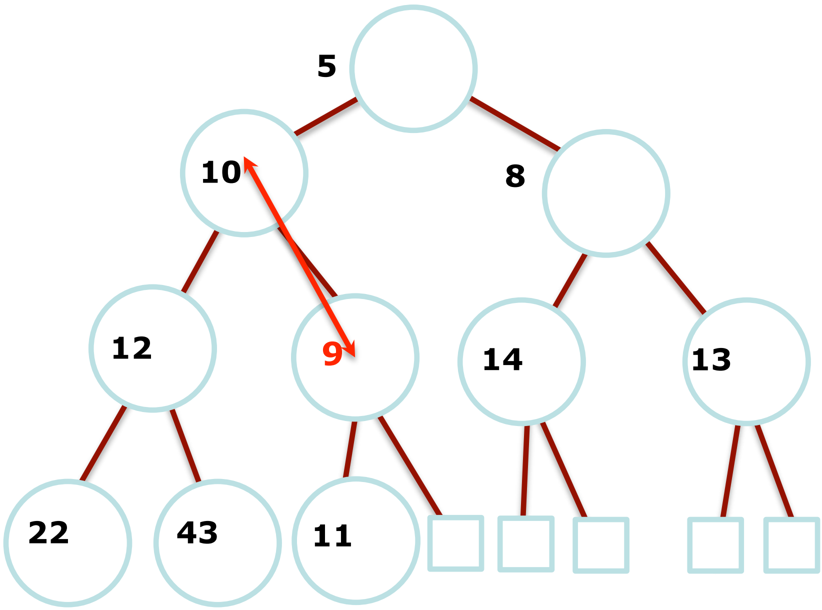
- The shape invariant tells us the location
- How to preserve the value invariant?
- Need to **float** the number

Done!

Implementing Insertions

- h[numElem]=newNumber
- **float**(numElem)
- numElem++;

Value	5	10	8	12	11	14	13	22	43						
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



Float operation

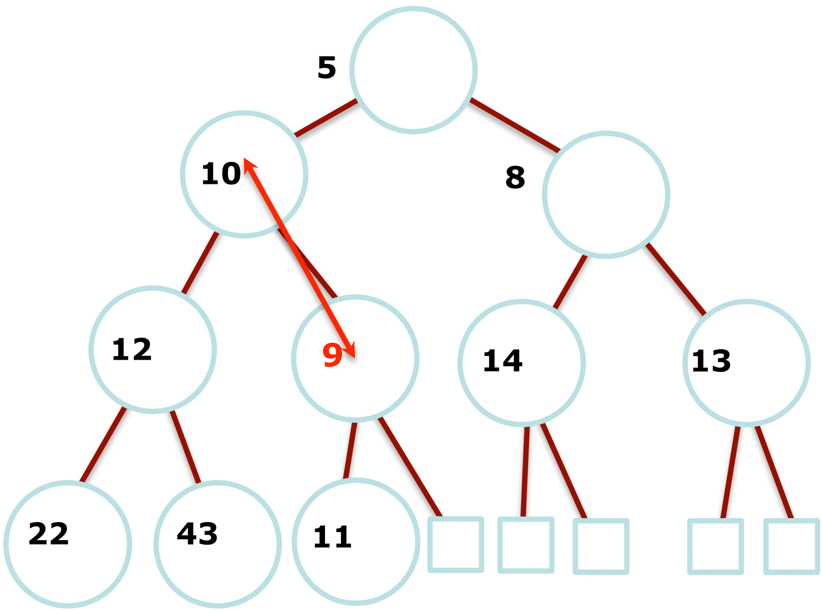
float(index)

If index==1 return

If h[parent(index)]> h[index]

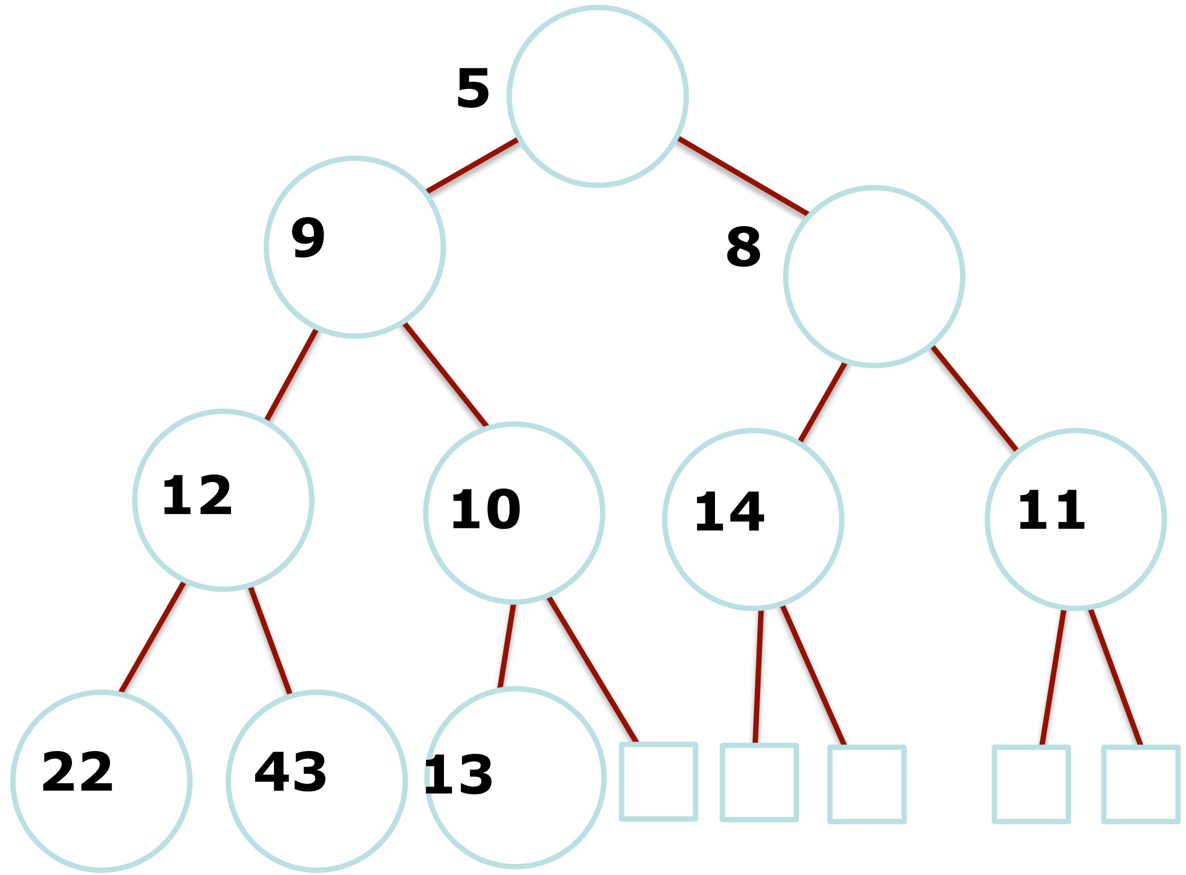
Swap index and parent(index)

float(parent(index))



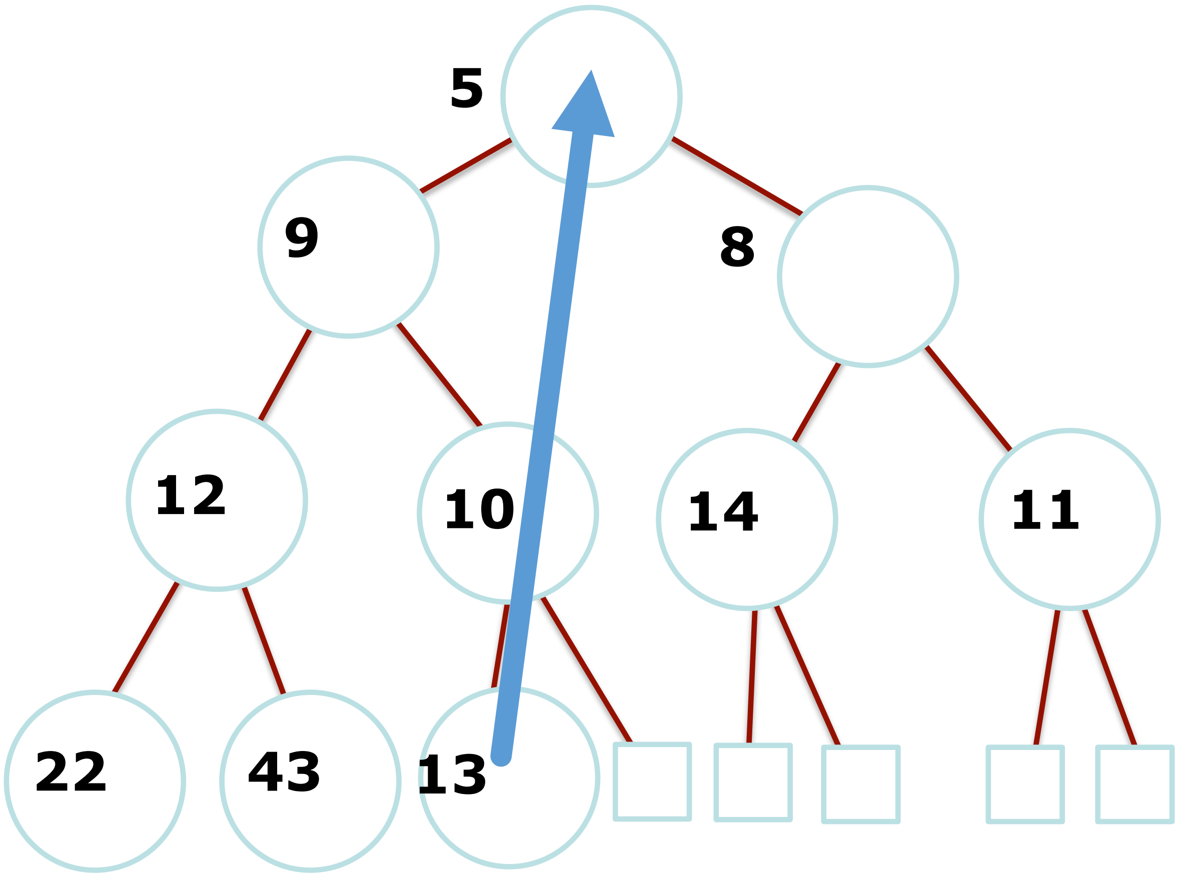
runtime? O(h), where h is height of tree

What about remove_min?



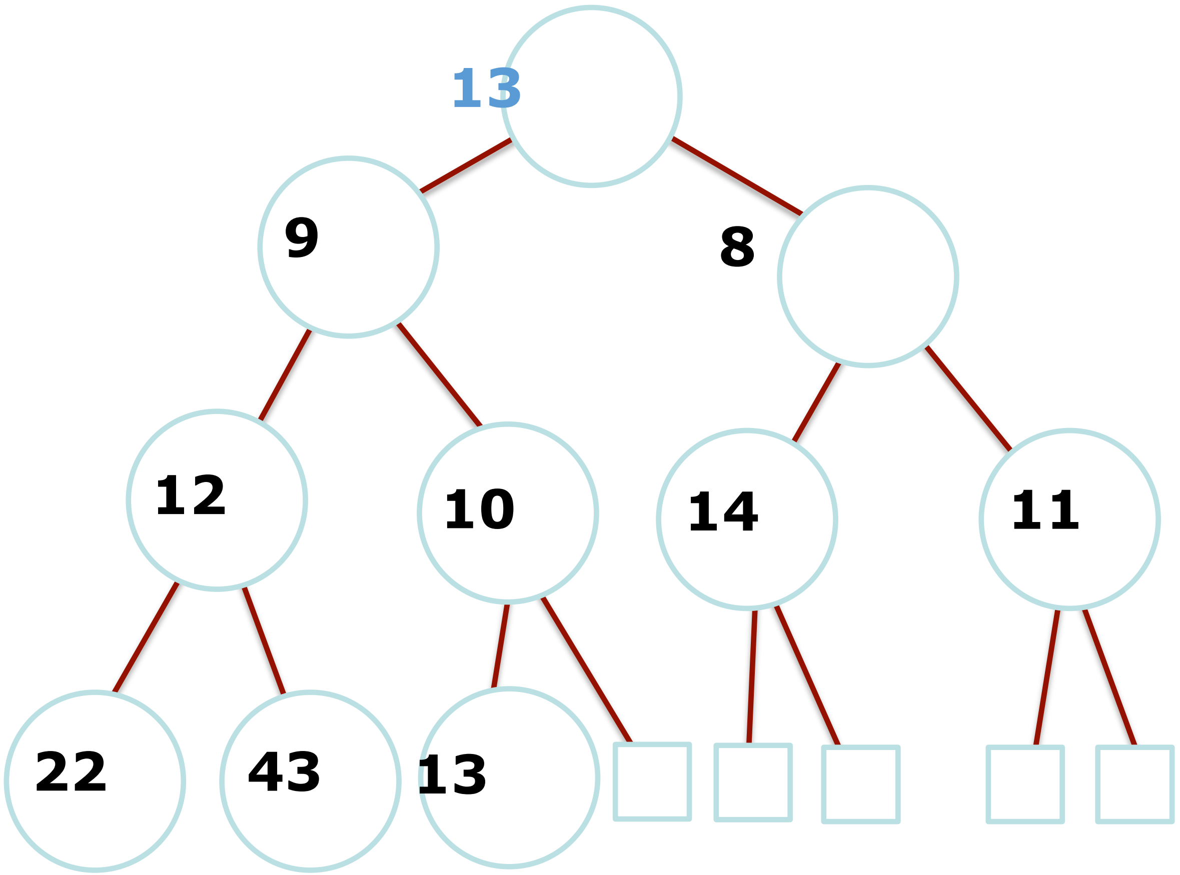
Preserve the shape invariant!
Can only remove last leaf

What about remove_min?



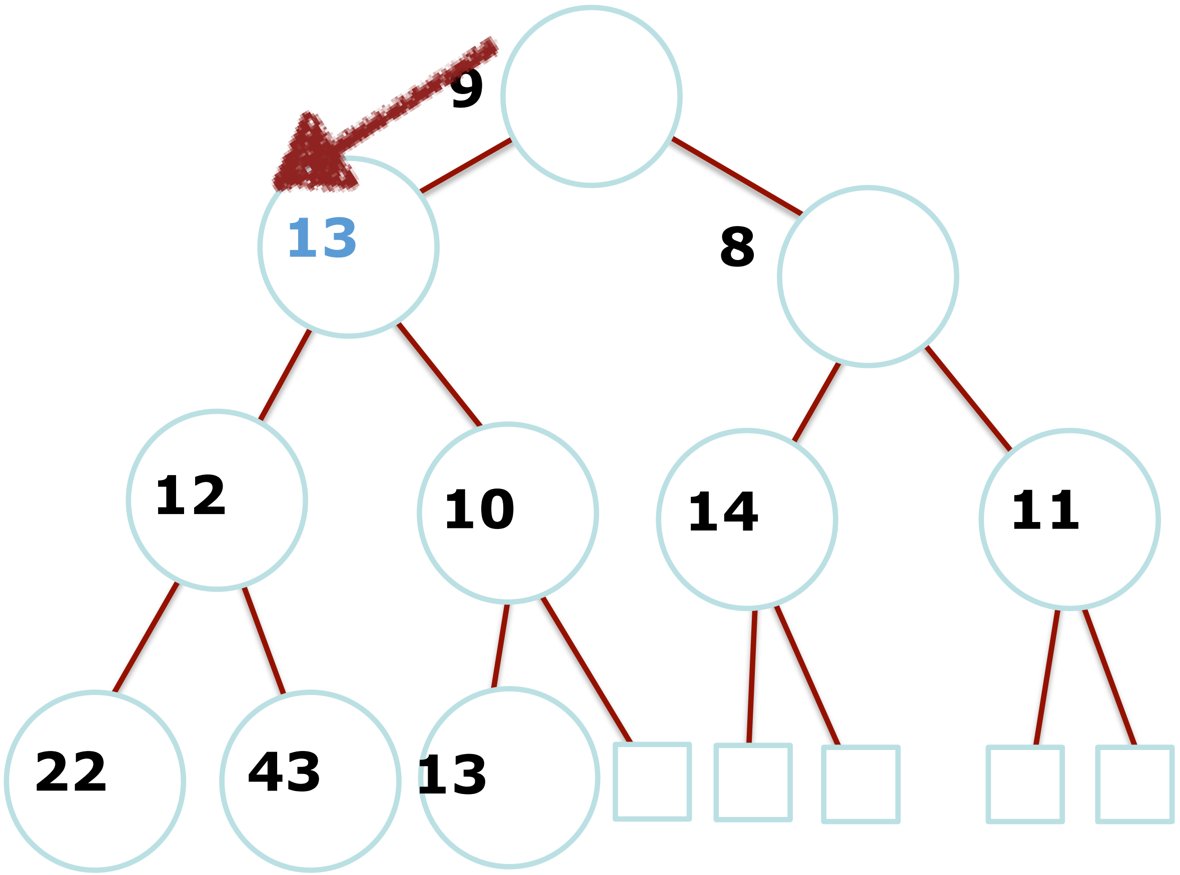
Move last to top

What about removal?



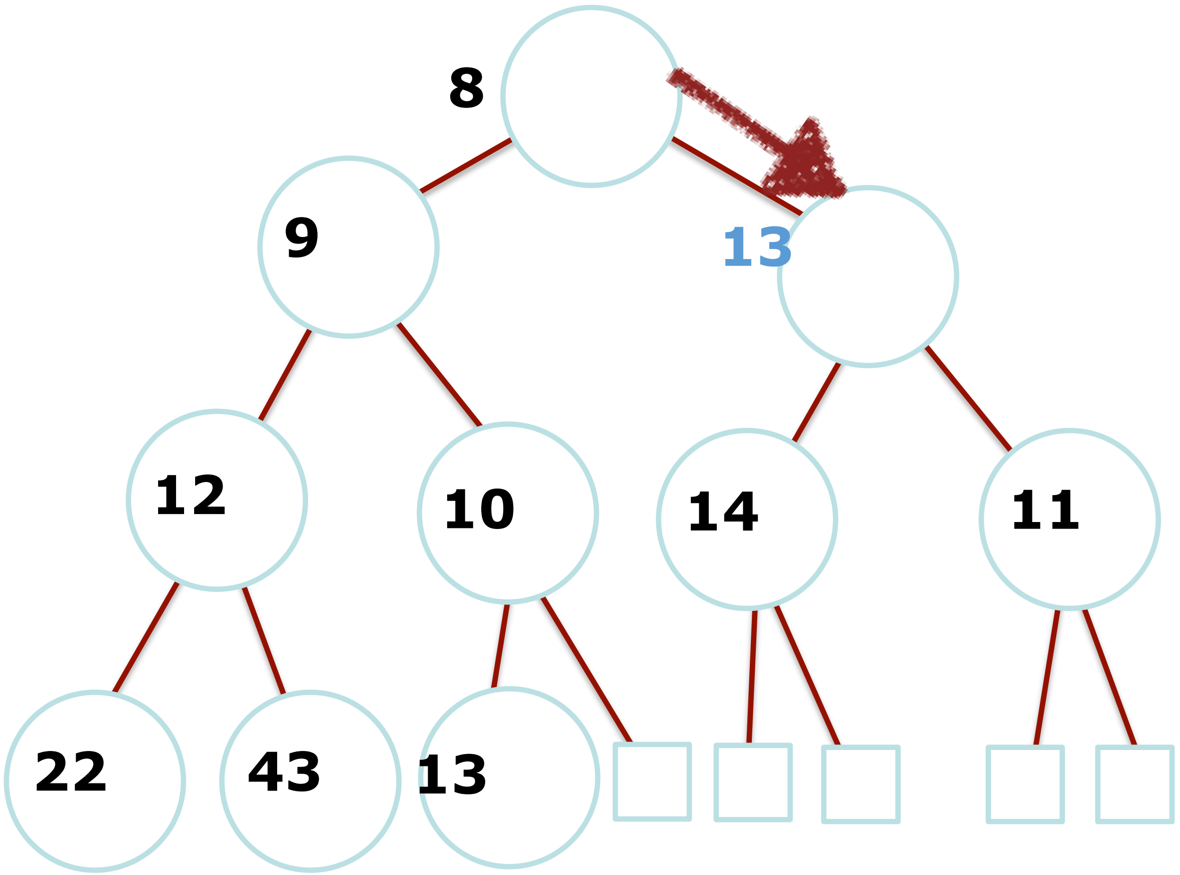
Need to sink 13. Where?

Oops!



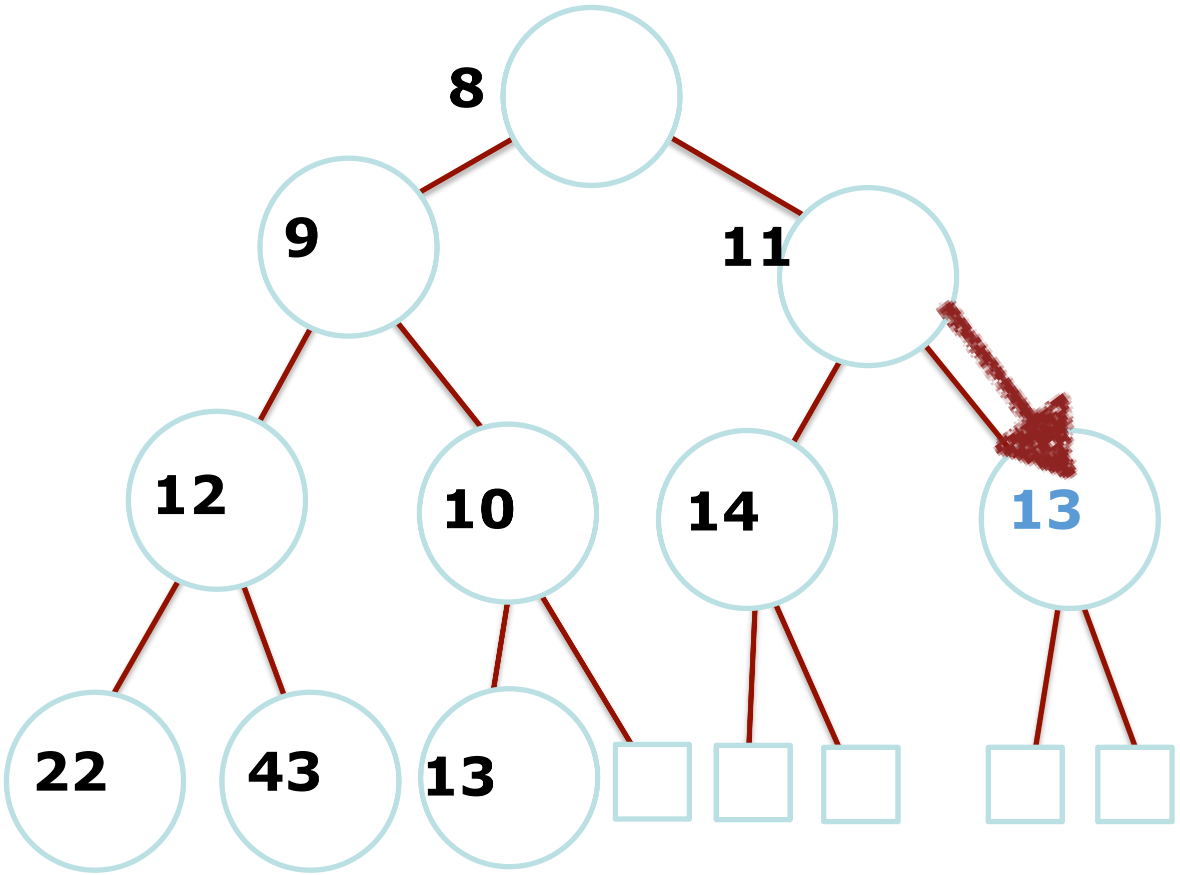
Sibling is not happy

Swap 13 with SMALLEST child

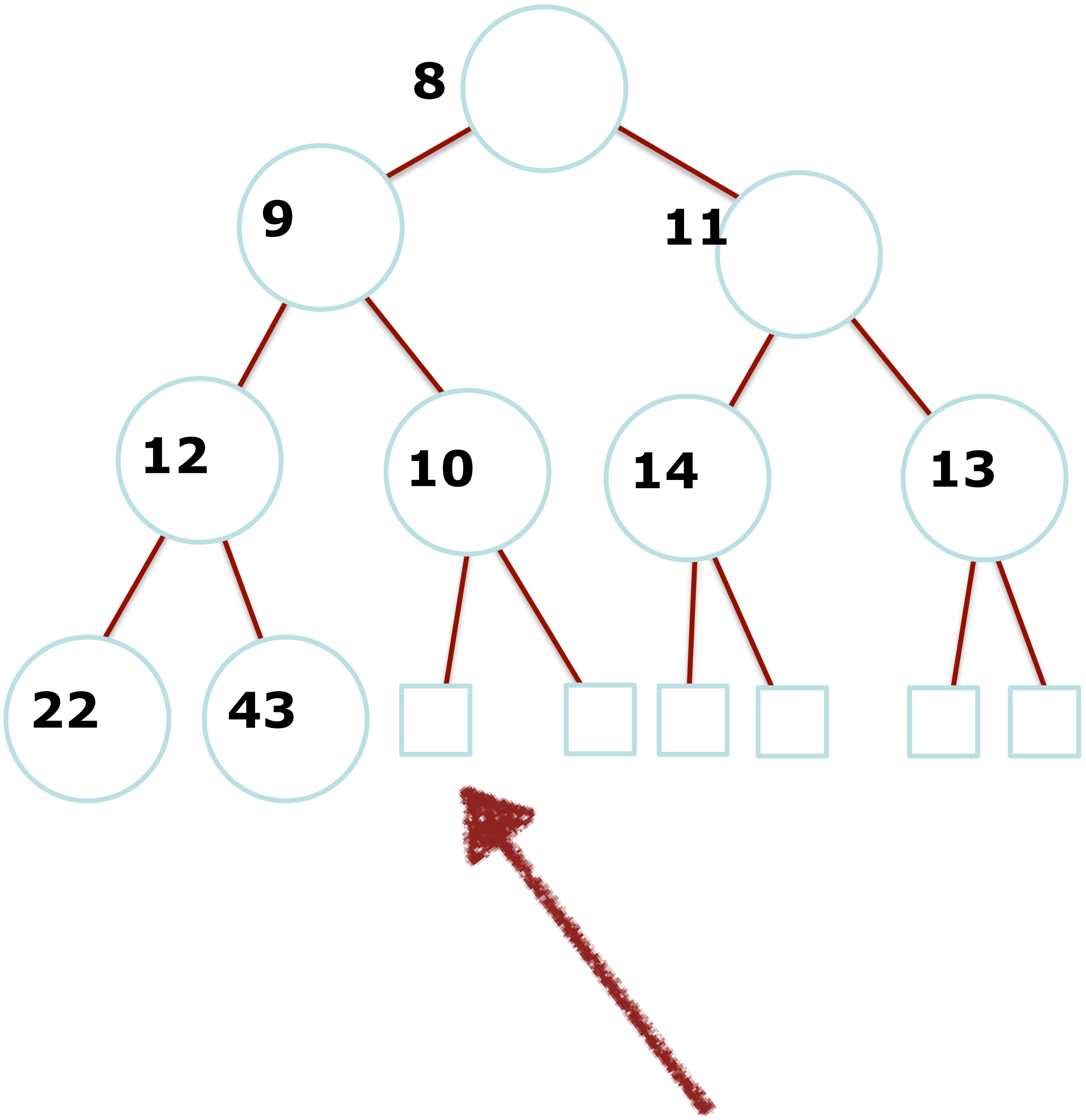


Recursively sink down

Swap 13 with SMALLEST child



Final step: decrease numElem



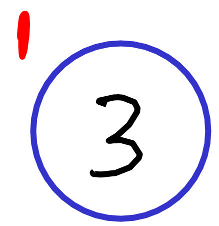
Heap building: the FORWARD METHOD

Heap building: the FORWARD METHOD (left to right)

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

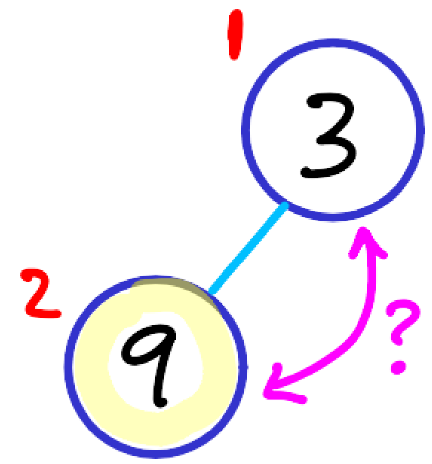
Heap building: the FORWARD METHOD (left to right)

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



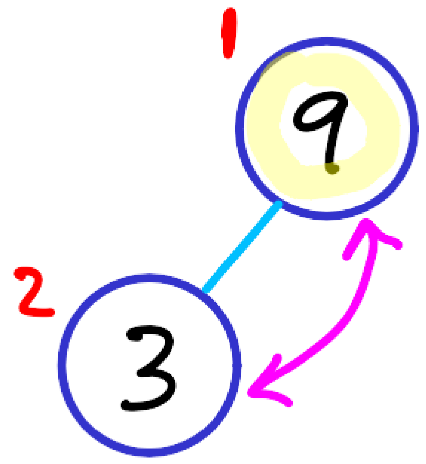
Heap building: the FORWARD METHOD (left to right)

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



Heap building: the FORWARD METHOD (left to right)

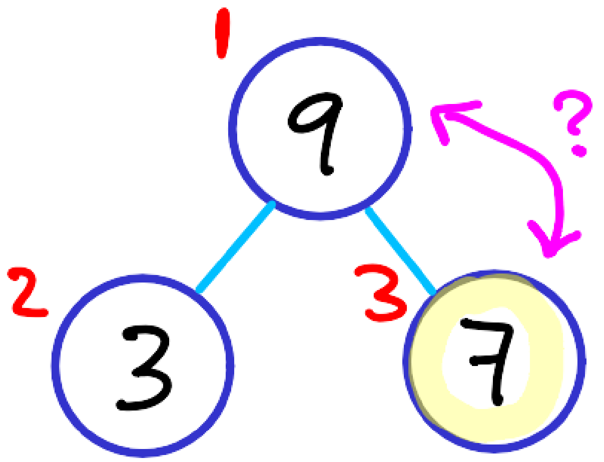
1	2	3	4	5	6	7	8	9	10
9	3	7	10	8	4	14	2	16	1



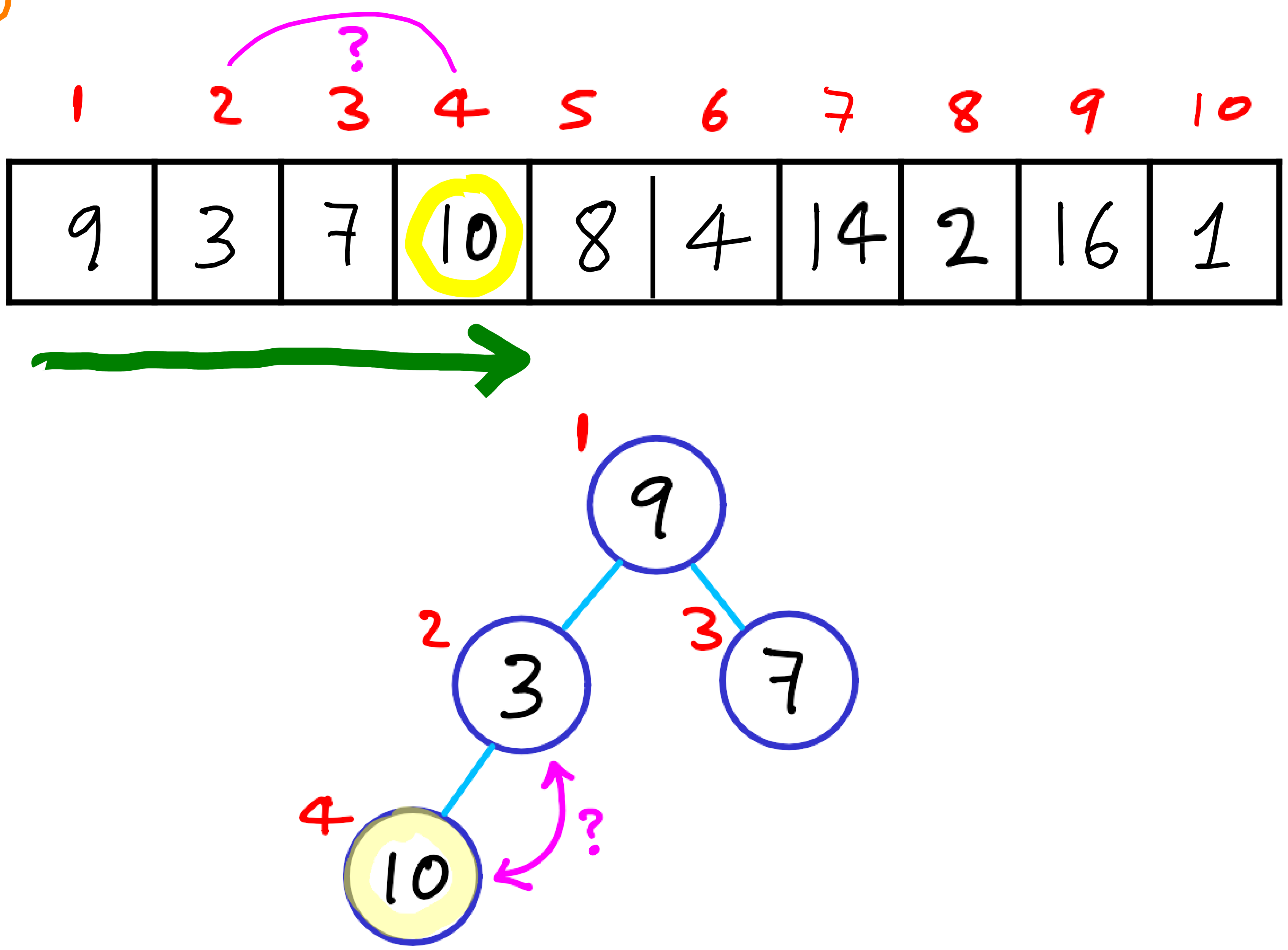
Heap building: the FORWARD METHOD (left to right)

?

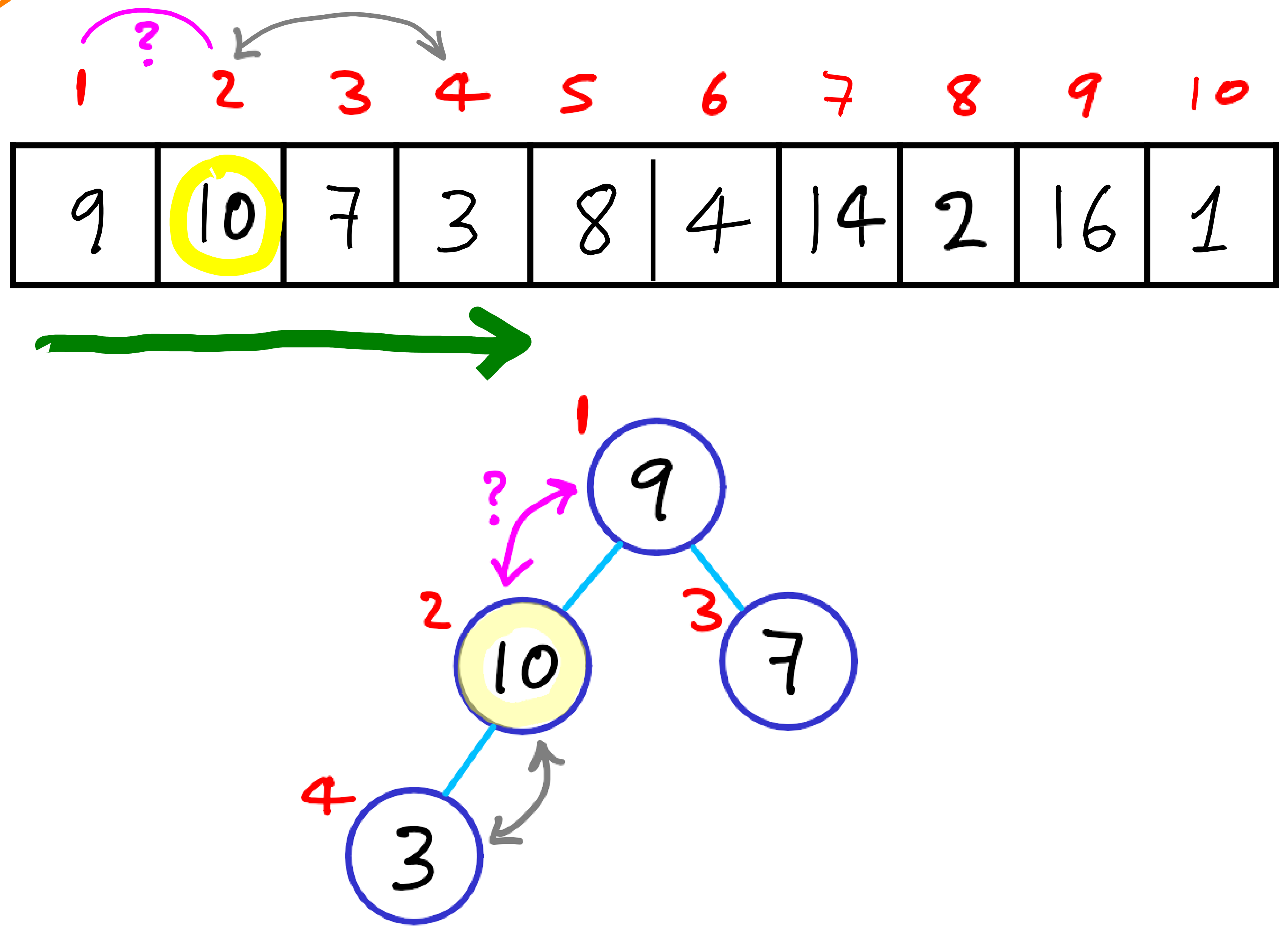
1	2	3	4	5	6	7	8	9	10
9	3	7	10	8	4	14	2	16	1



Heap building: the FORWARD METHOD (left to right)

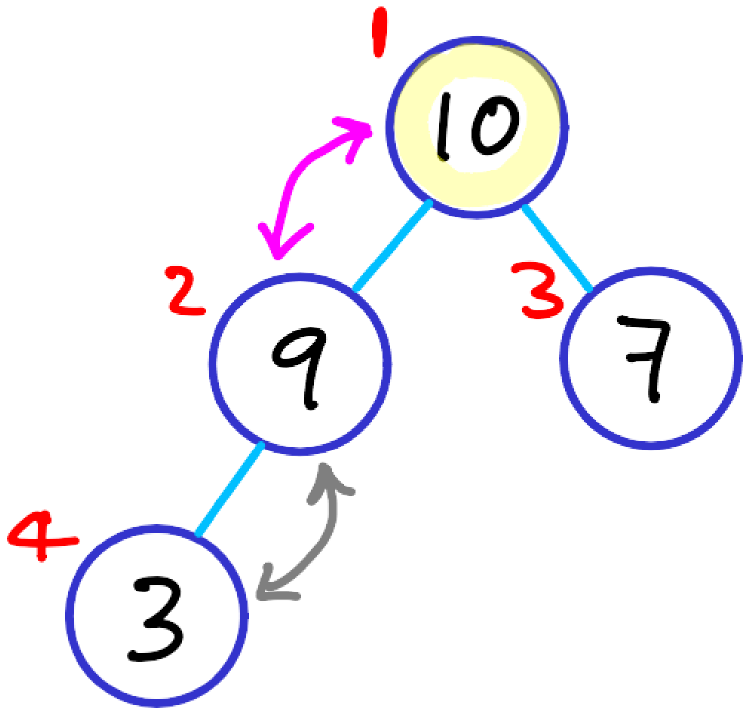


Heap building: the FORWARD METHOD (left to right)



Heap building: the FORWARD METHOD (left to right)

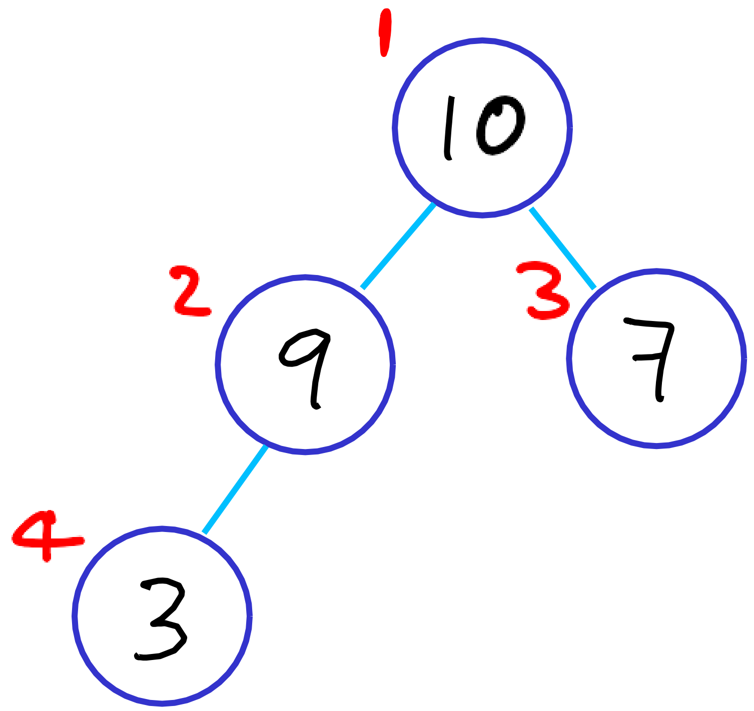
1	2	3	4	5	6	7	8	9	10
10	9	7	3	8	4	14	2	16	1



etc

Heap building: the FORWARD METHOD (left to right)

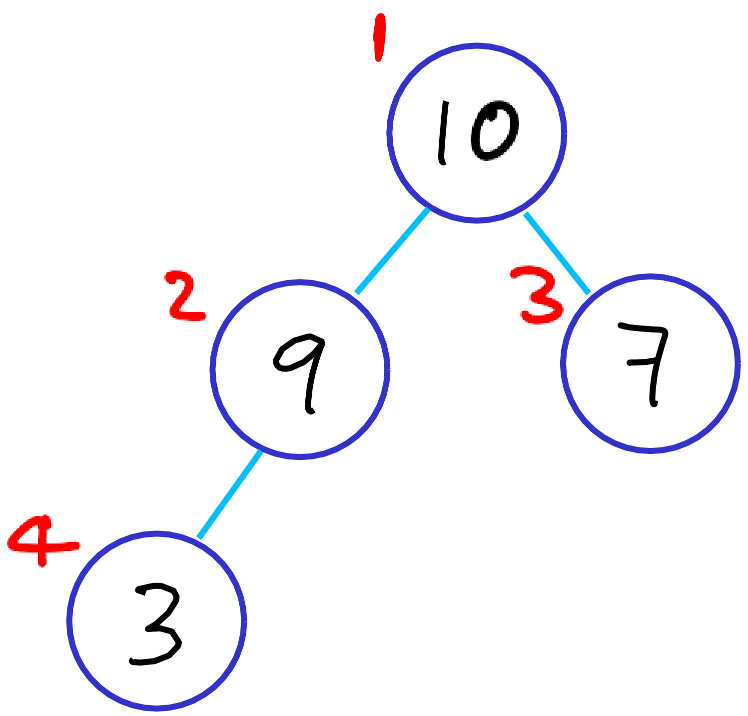
1	2	3	4	5	6	7	8	9	10
10	9	7	3	8	4	14	2	16	1



time?

Heap building: the FORWARD METHOD (left to right)

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

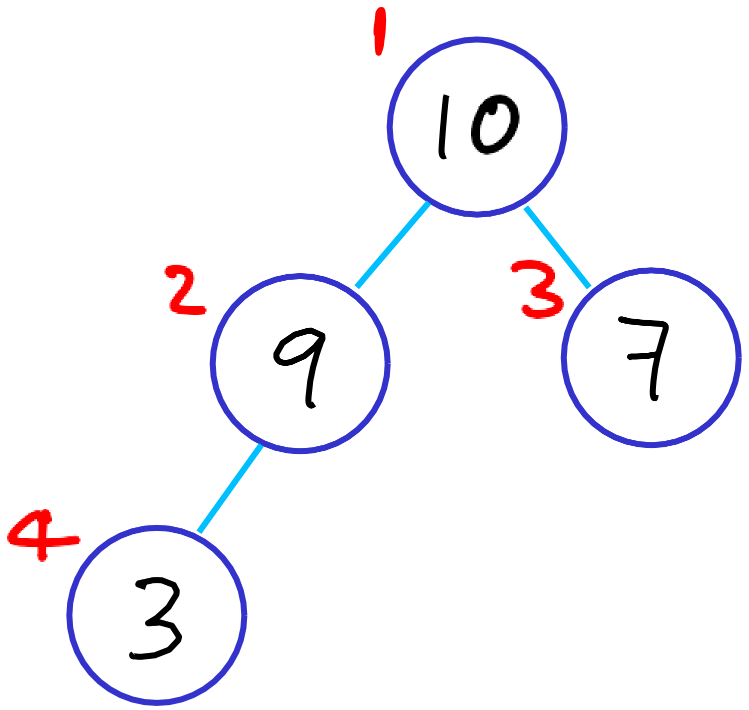


time = $O(n \log n)$

$O(\log n)$ per insertion

Heap building: the FORWARD METHOD (left to right)

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



time = $O(n \log n)$

$O(\log n)$ per insertion

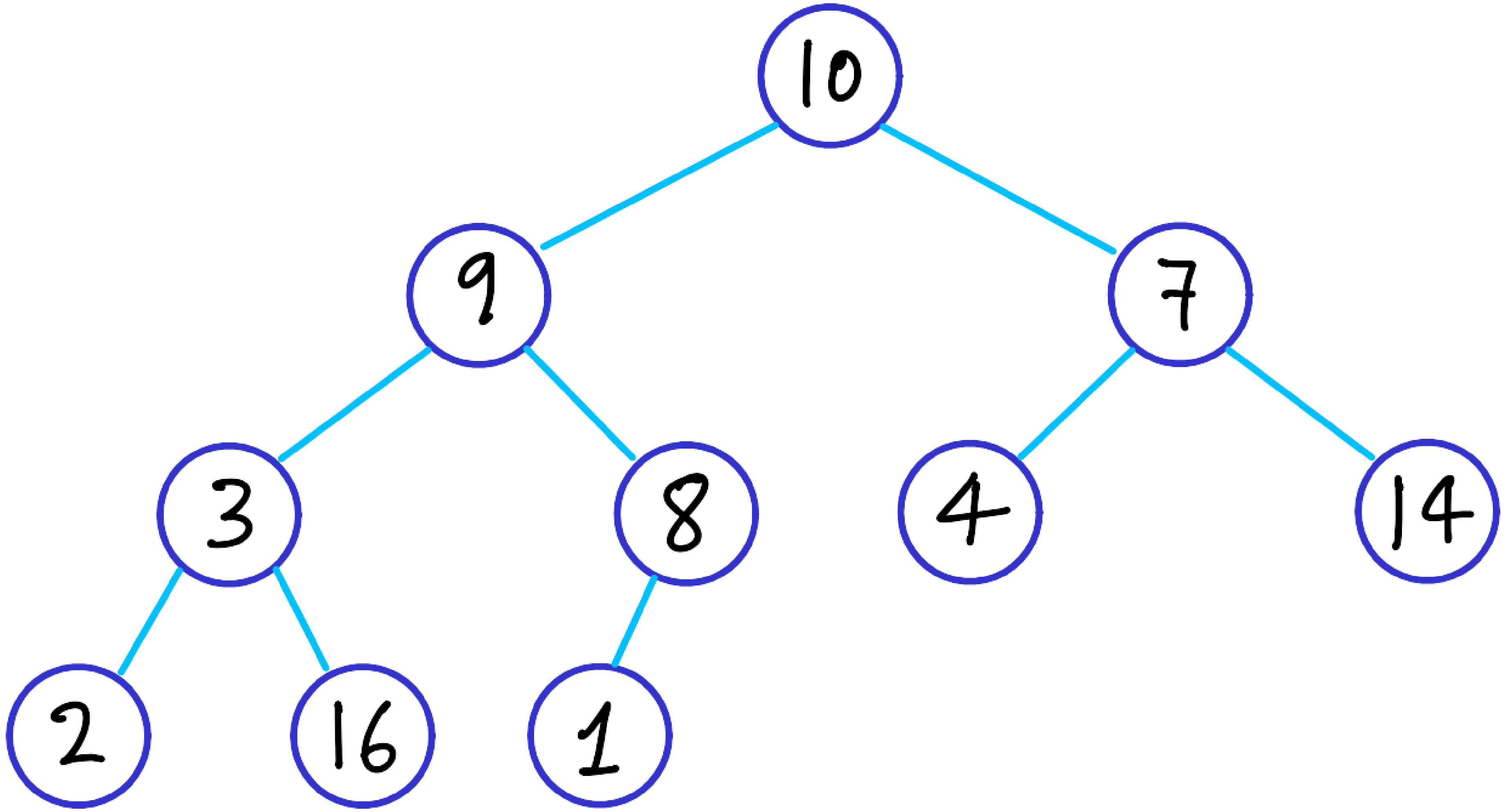
Works for streaming data

Heap building: the REVERSE METHOD (right to left)

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

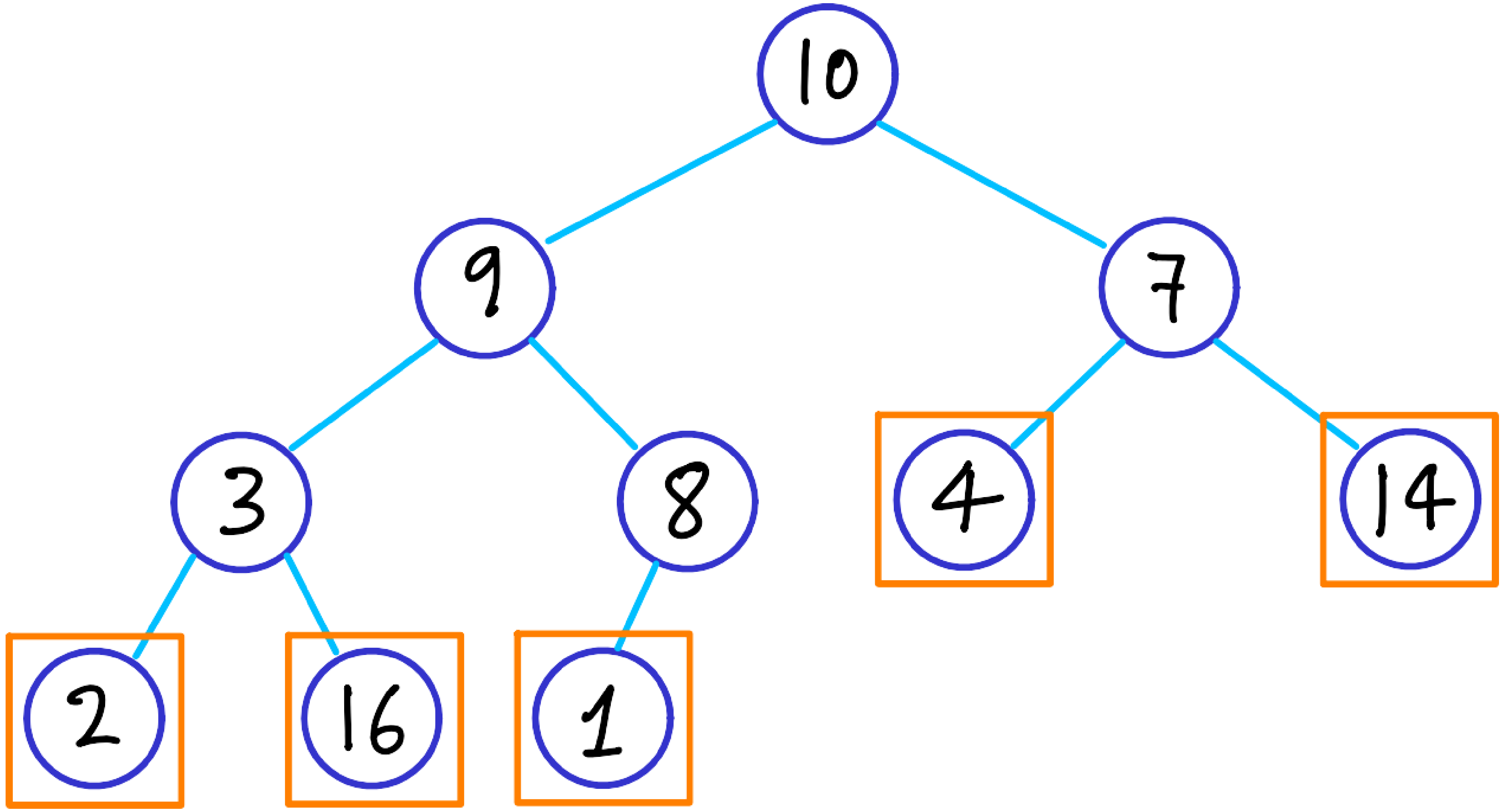
Heap building: the REVERSE METHOD (right to left)

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



Heap building: the REVERSE METHOD (right to left)

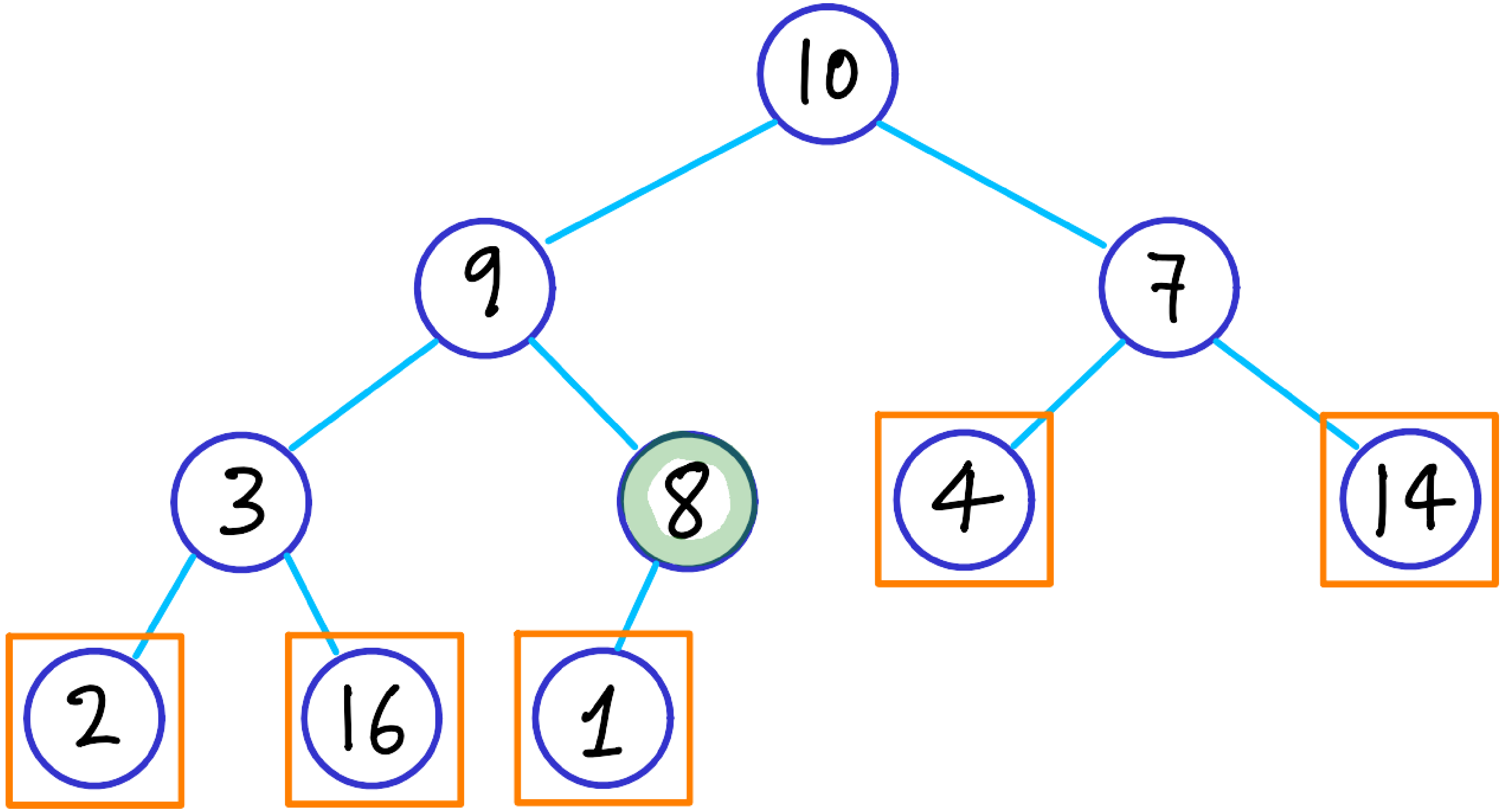
1	2	3	4	5	6	7	8	9	10
10	9	7	3	8	4	14	2	16	1



already heaps

Heap building: the REVERSE METHOD (right to left)

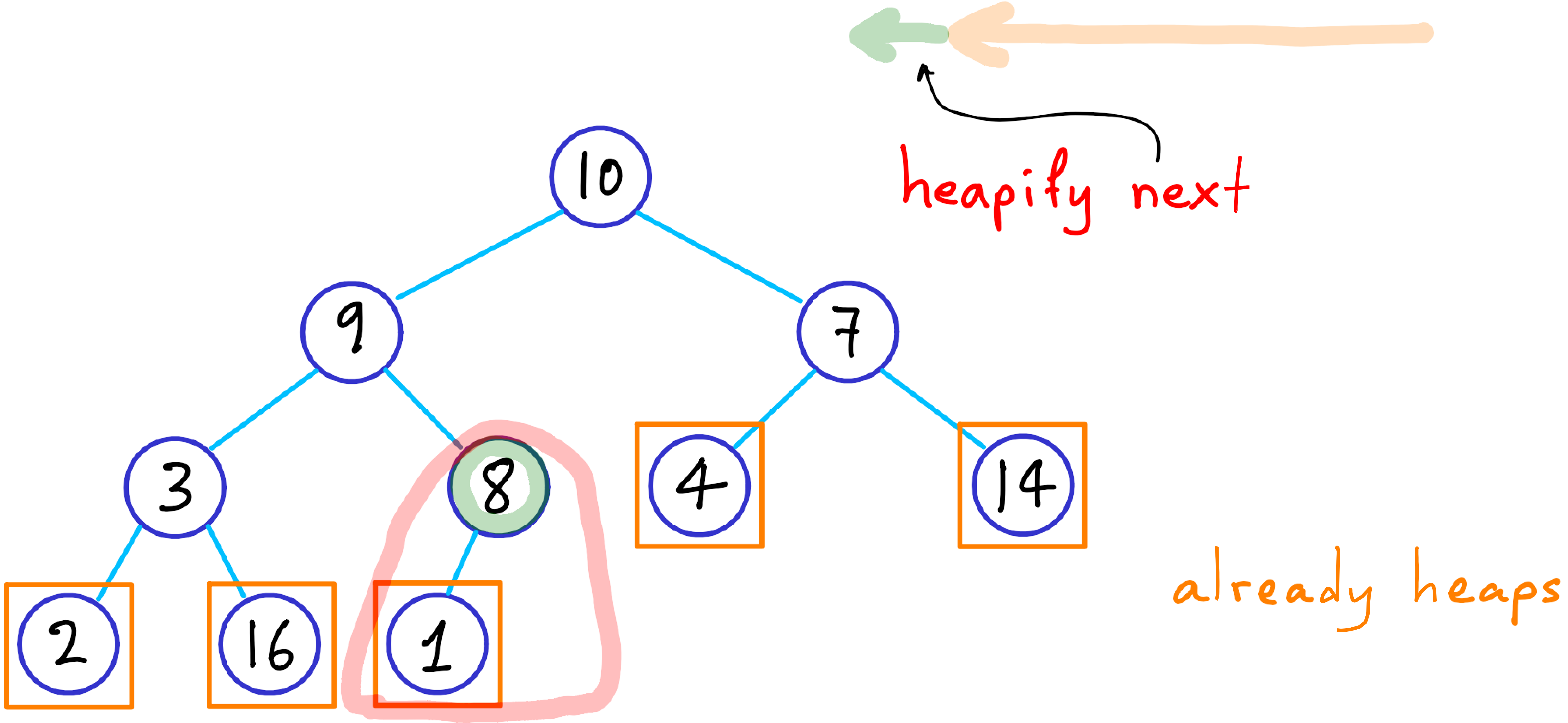
1	2	3	4	5	6	7	8	9	10
10	9	7	3	8	4	14	2	16	1



already heaps

Heap building: the REVERSE METHOD (right to left)

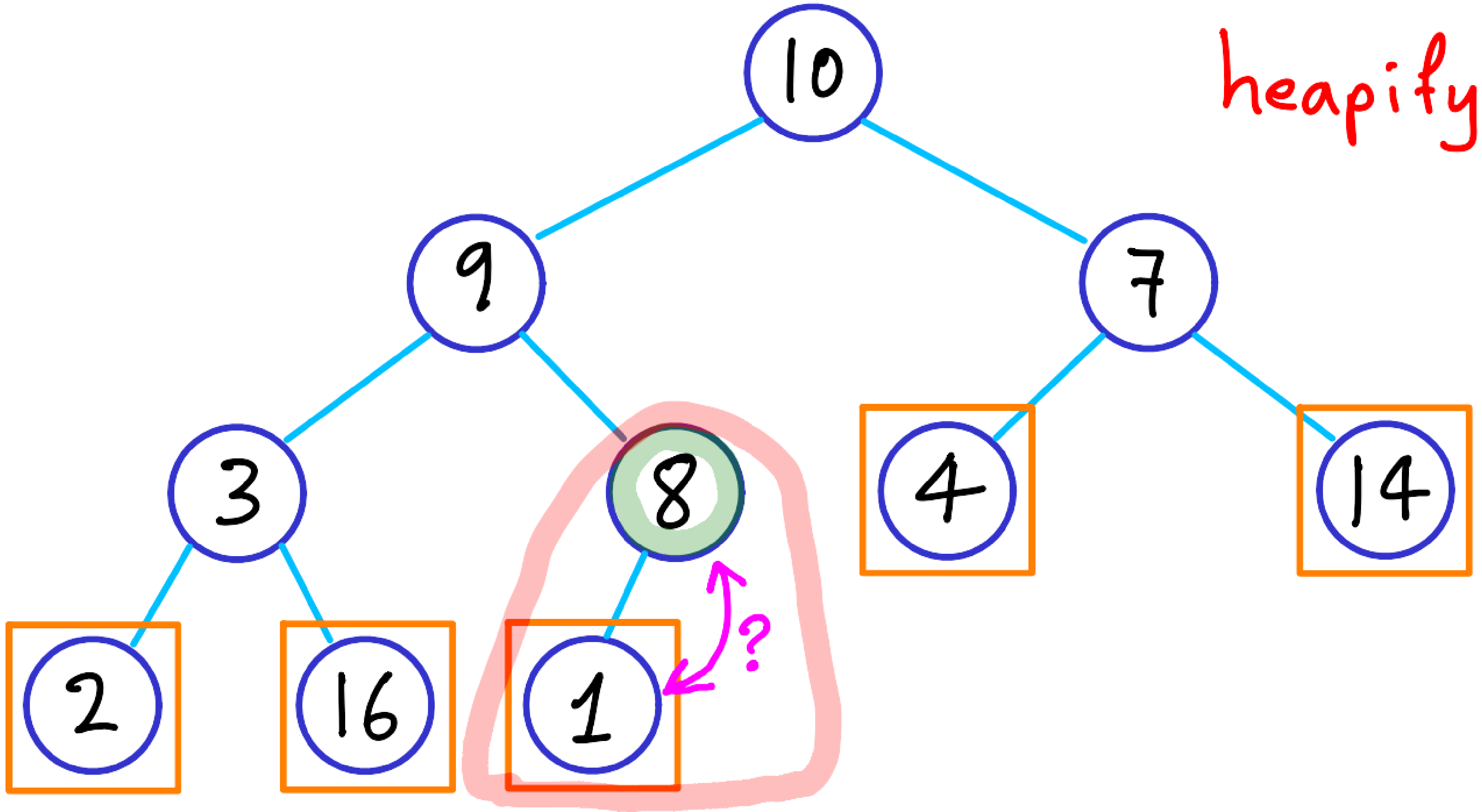
1	2	3	4	5	6	7	8	9	10
10	9	7	3	8	4	14	2	16	1



Heap building: the REVERSE METHOD (right to left)

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

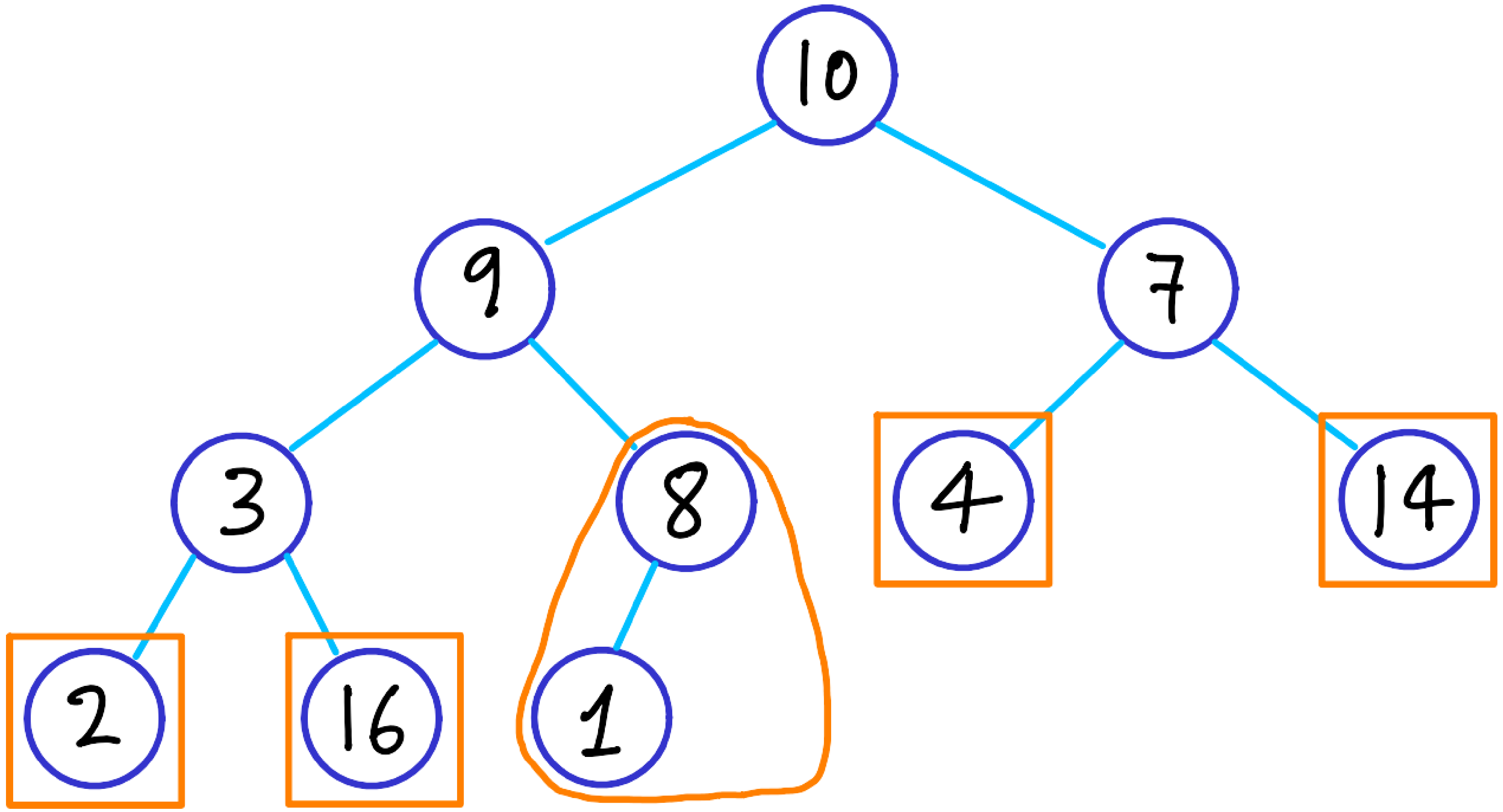
heapify next



already heaps

Heap building: the REVERSE METHOD (right to left)

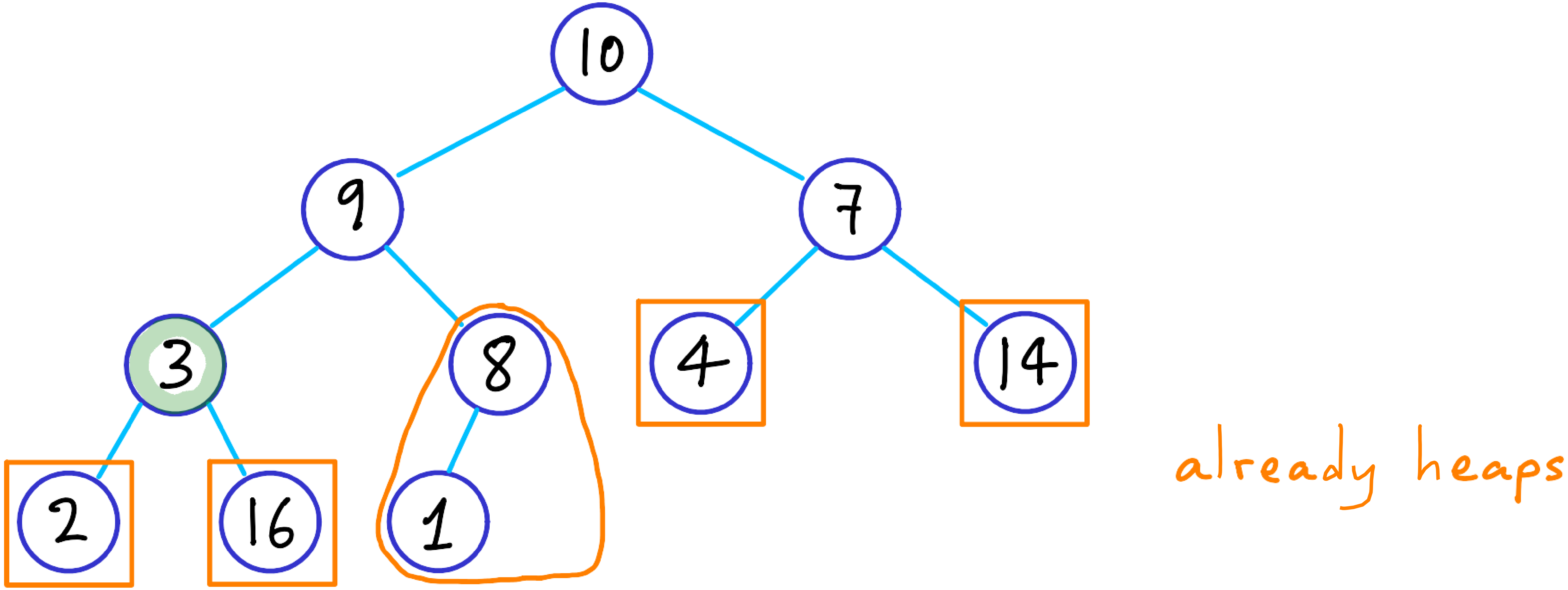
1	2	3	4	5	6	7	8	9	10
10	9	7	3	8	4	14	2	16	1



already heaps

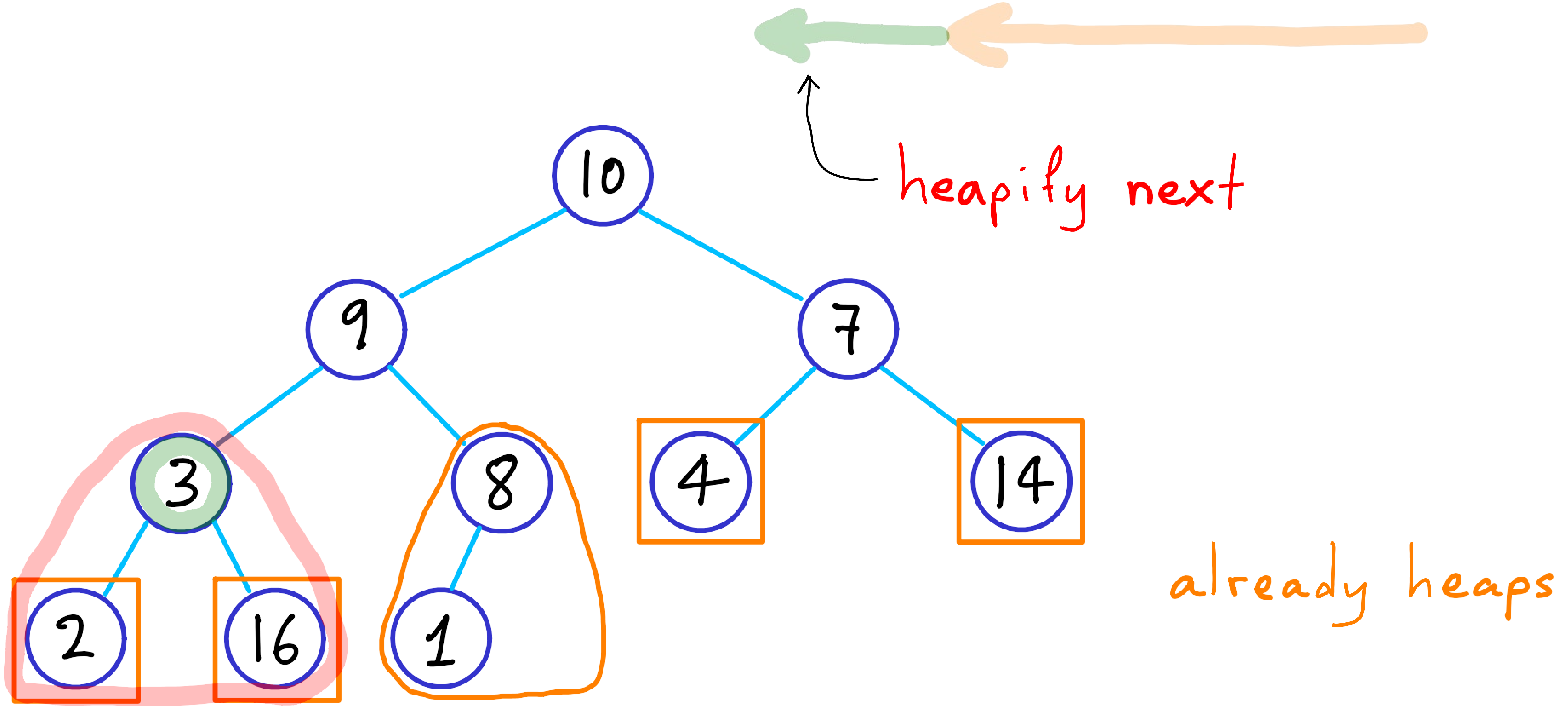
Heap building: the REVERSE METHOD (right to left)

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



Heap building: the REVERSE METHOD (right to left)

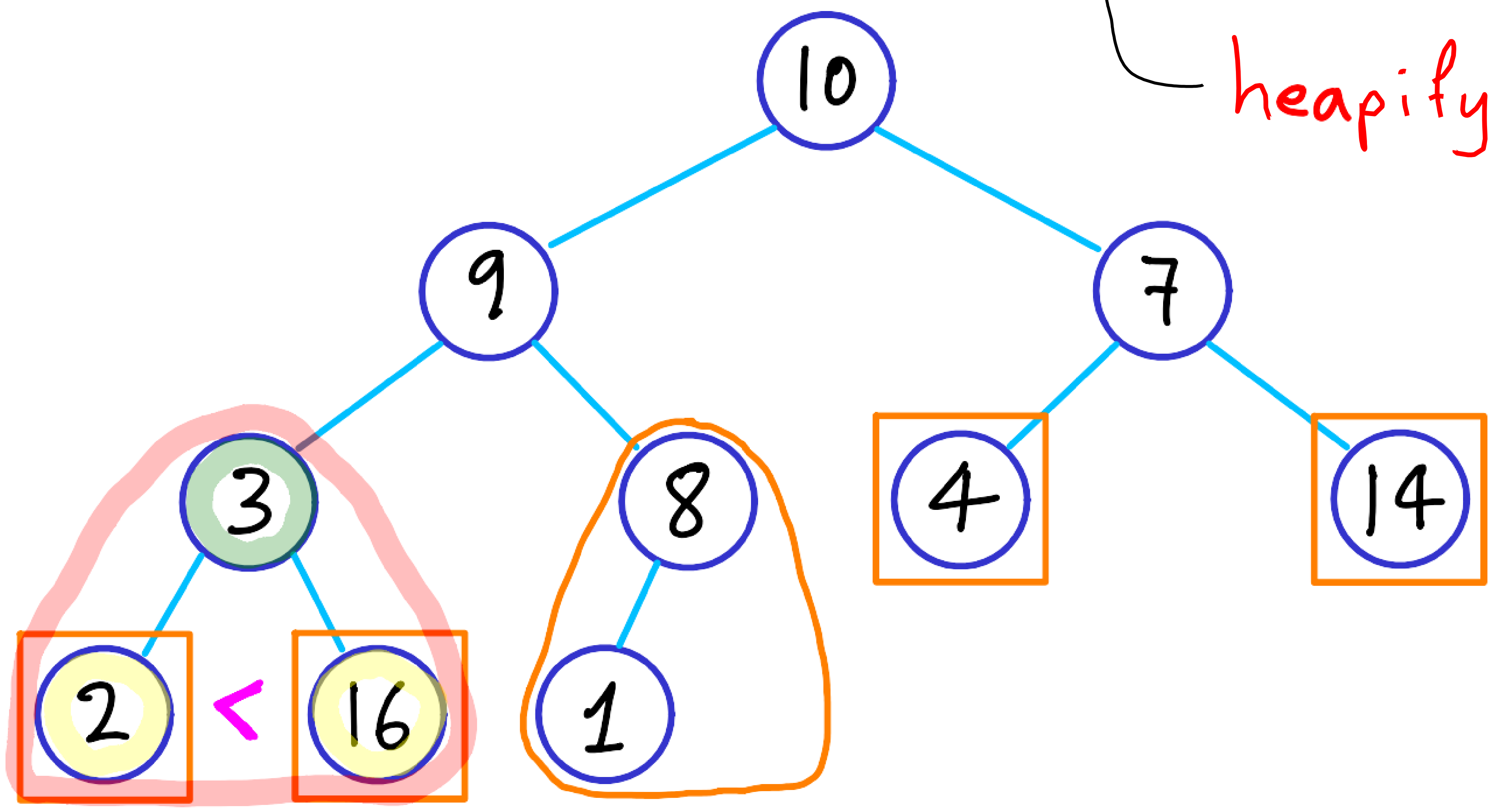
1	2	3	4	5	6	7	8	9	10
10	9	7	3	8	4	14	2	16	1



Heap building: the REVERSE METHOD (right to left)

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

heapify next

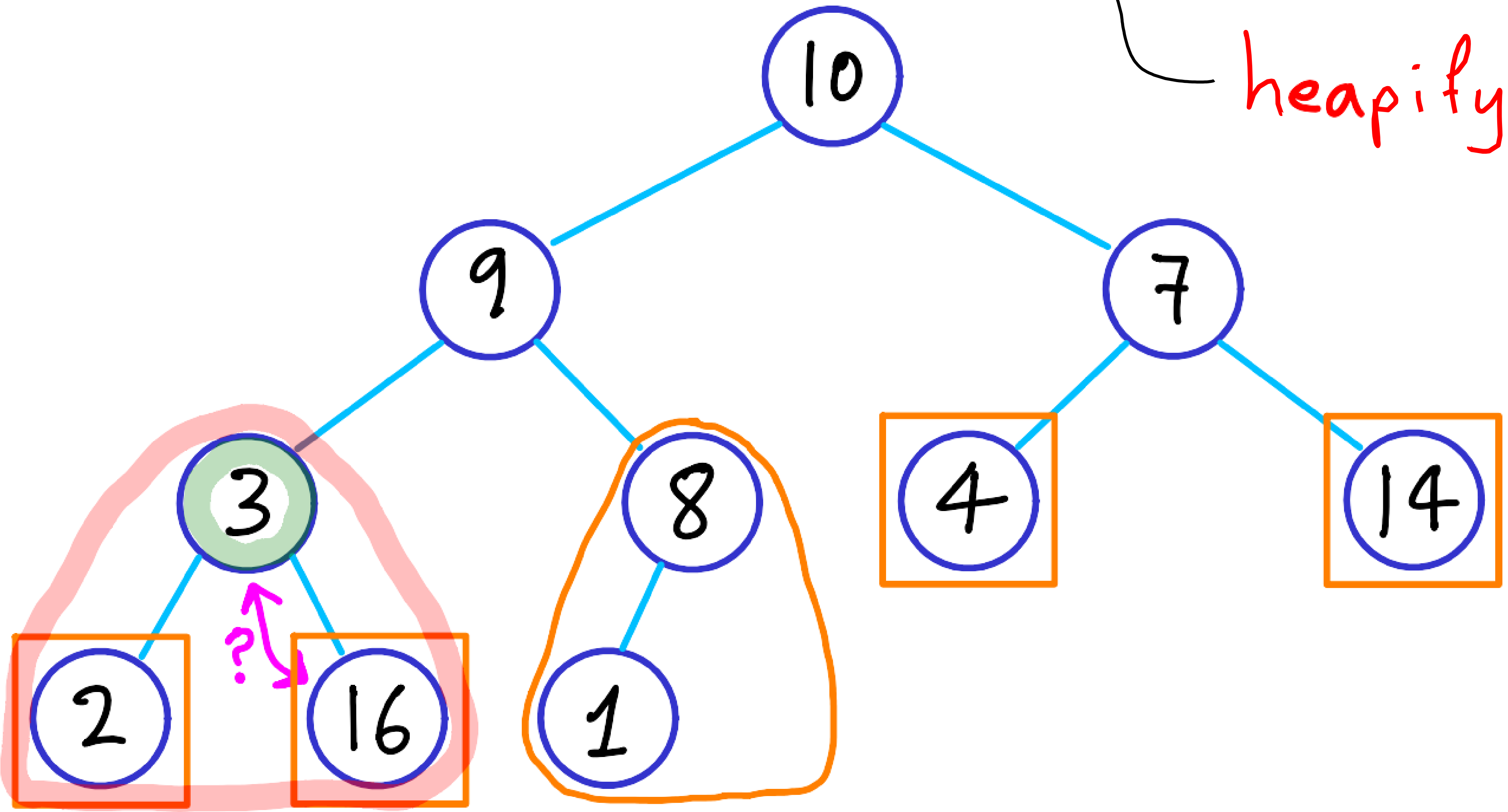


already heaps

Heap building: the REVERSE METHOD (right to left)

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

heapify next

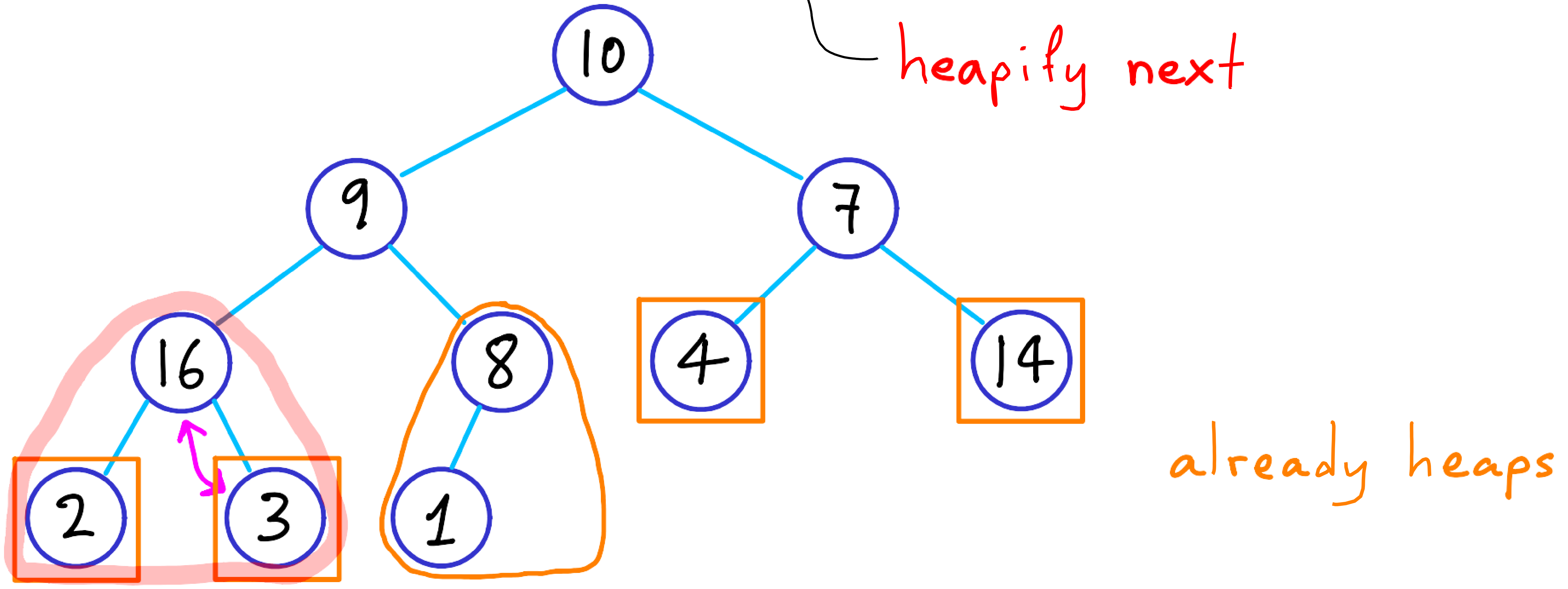


already heaps

Heap building: the REVERSE METHOD (right to left)

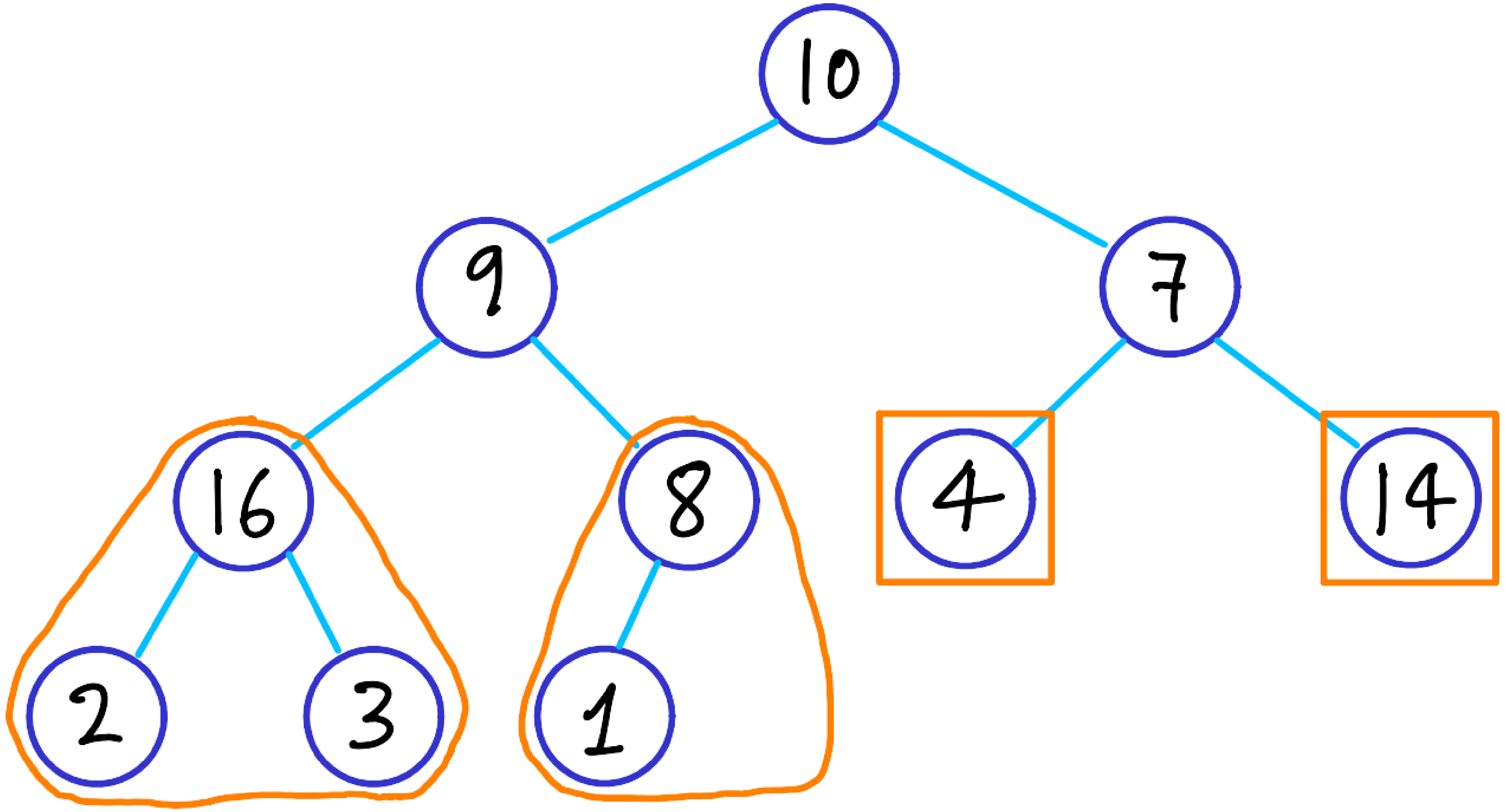
1	2	3	4	5	6	7	8	9	10
10	9	7	16	8	4	14	2	3	1

heapify next



Heap building: the REVERSE METHOD (right to left)

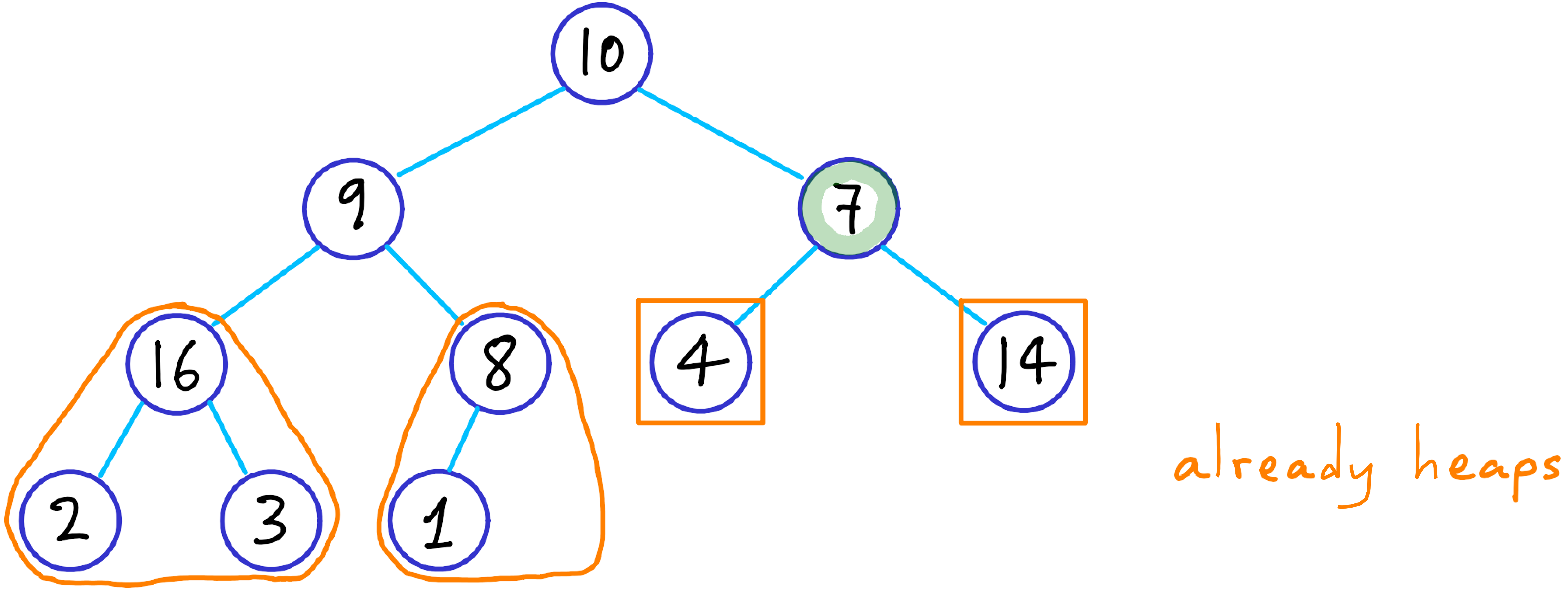
1	2	3	4	5	6	7	8	9	10
10	9	7	16	8	4	14	2	3	1



already heaps

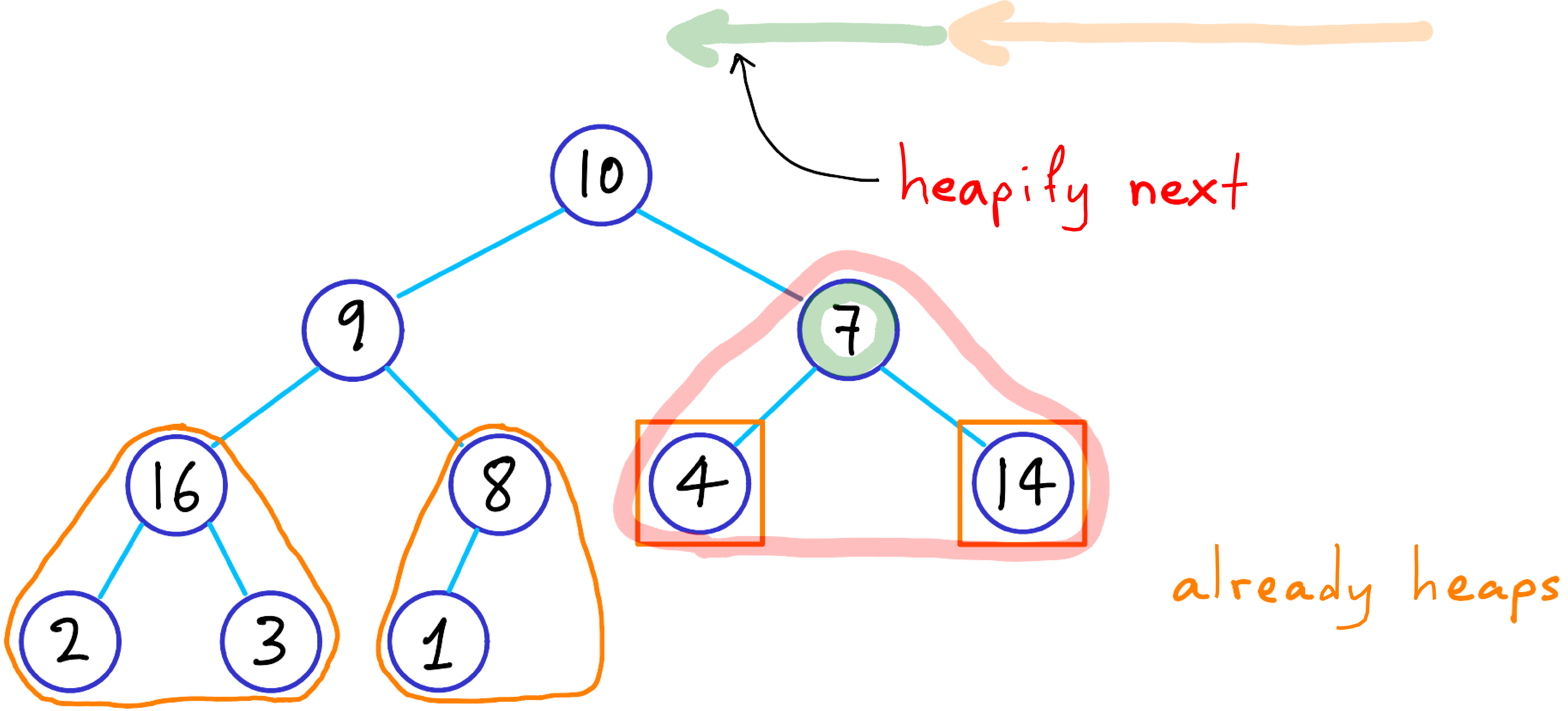
Heap building: the REVERSE METHOD (right to left)

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



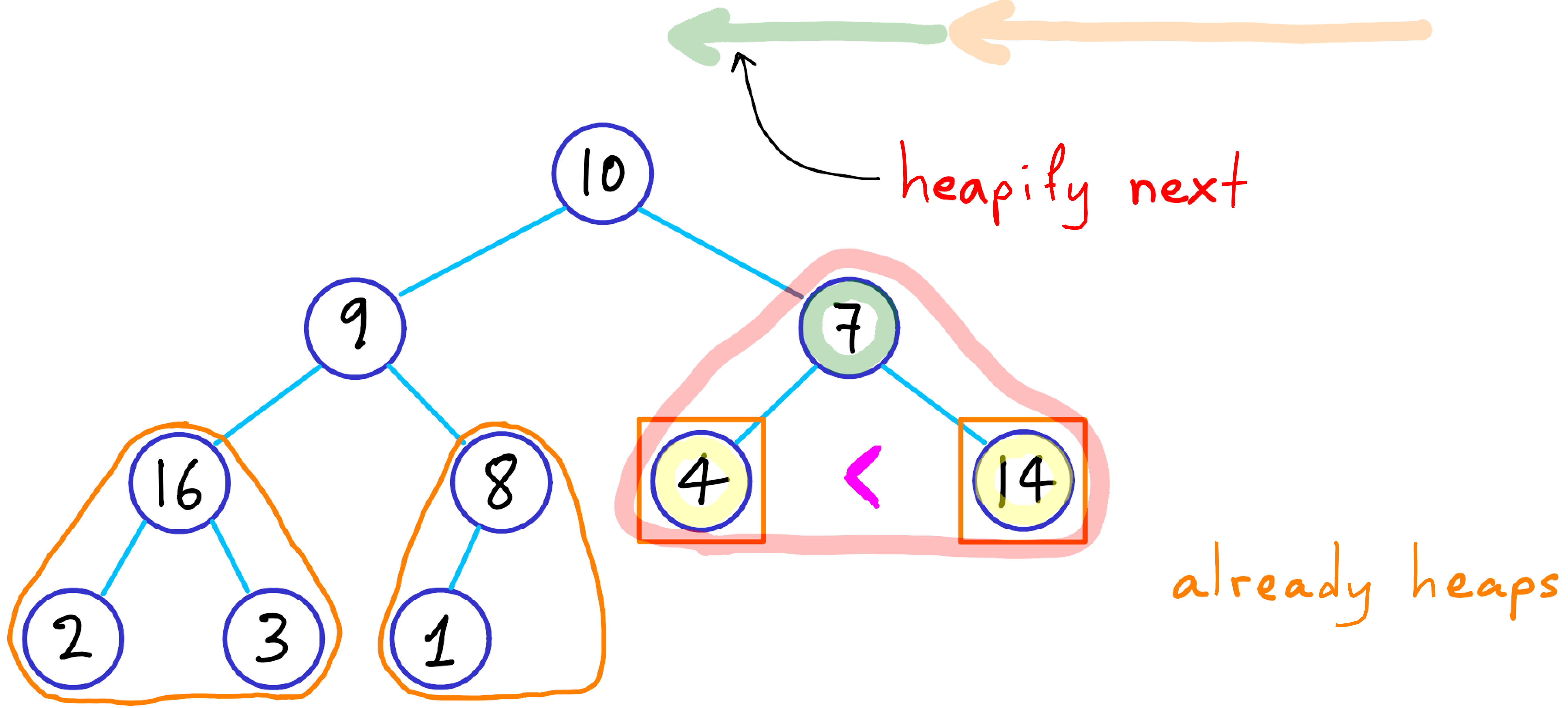
Heap building: the REVERSE METHOD (right to left)

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



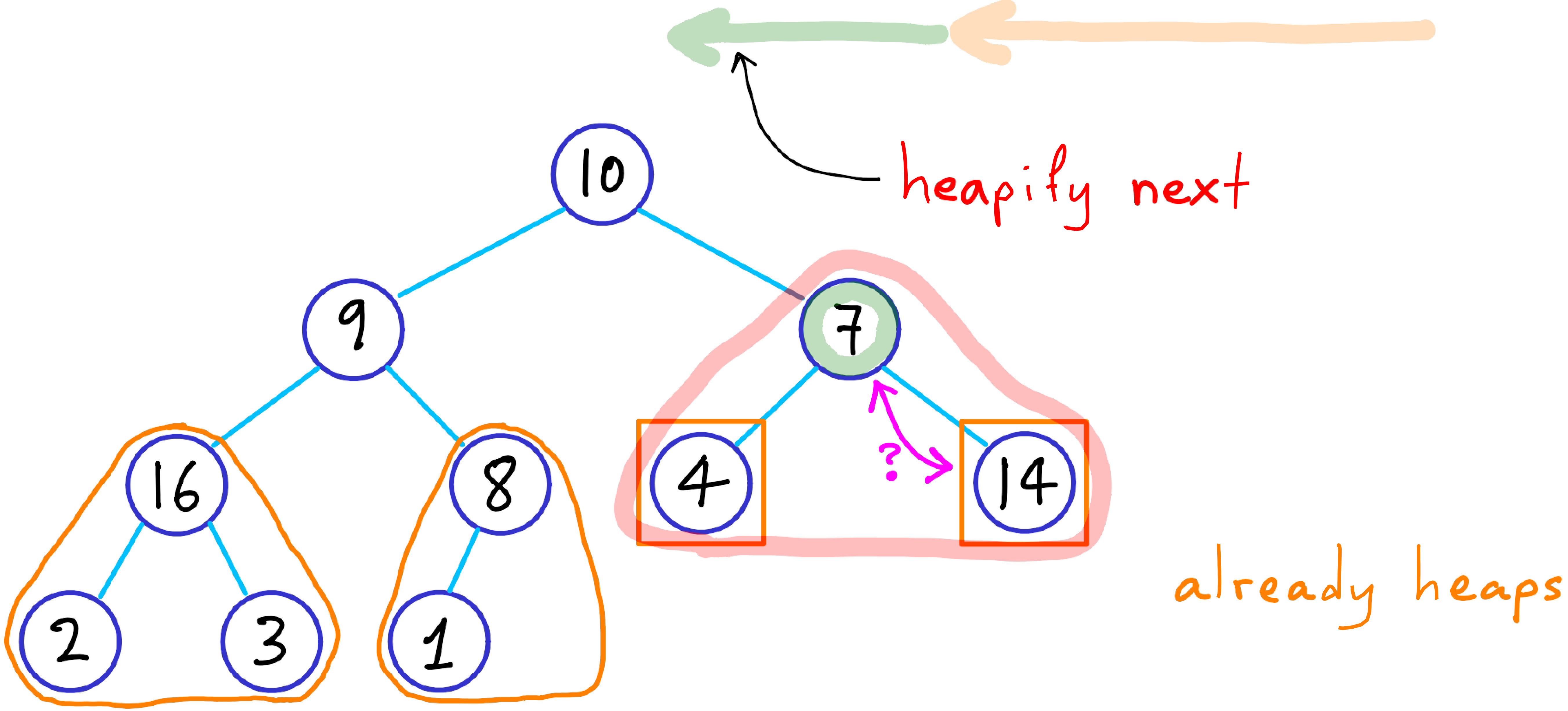
Heap building: the REVERSE METHOD (right to left)

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



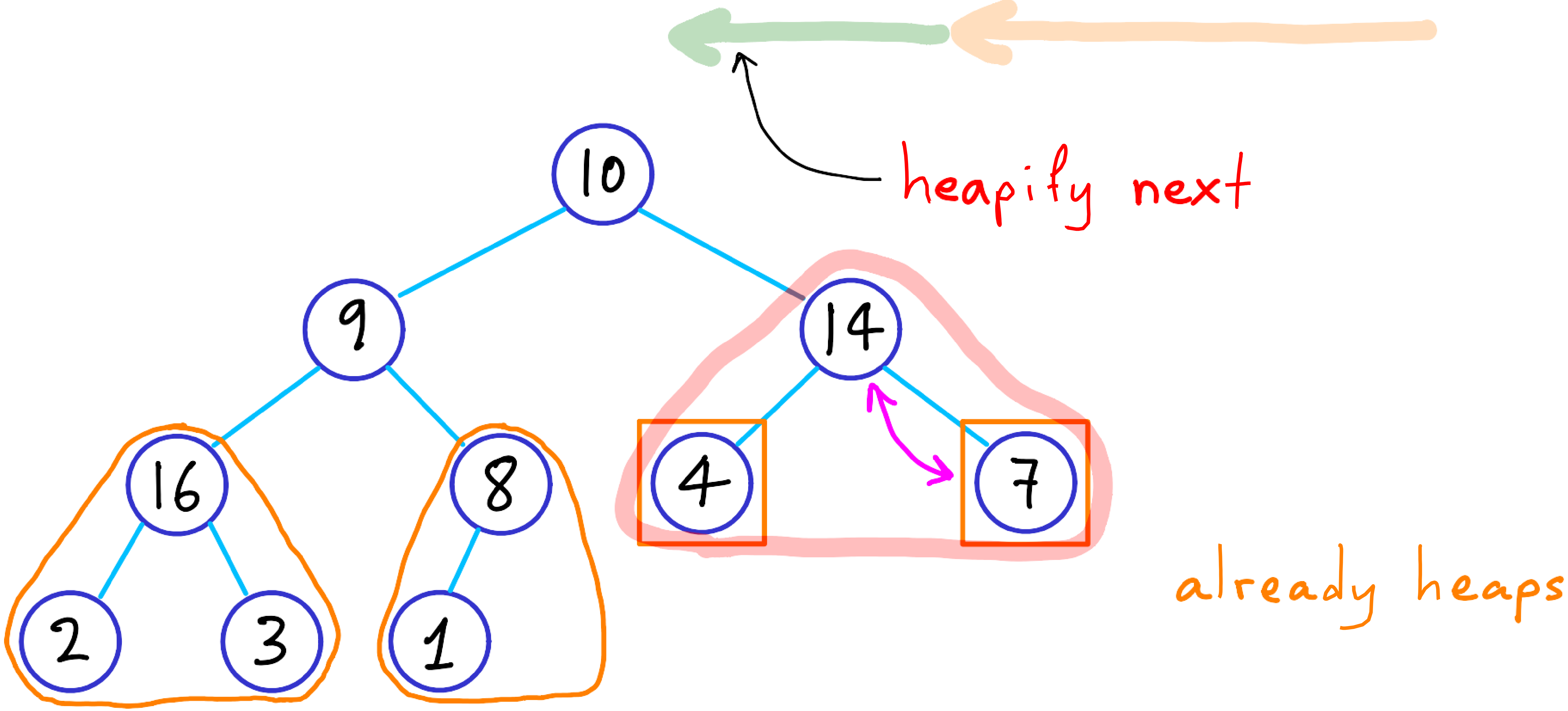
Heap building: the REVERSE METHOD (right to left)

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



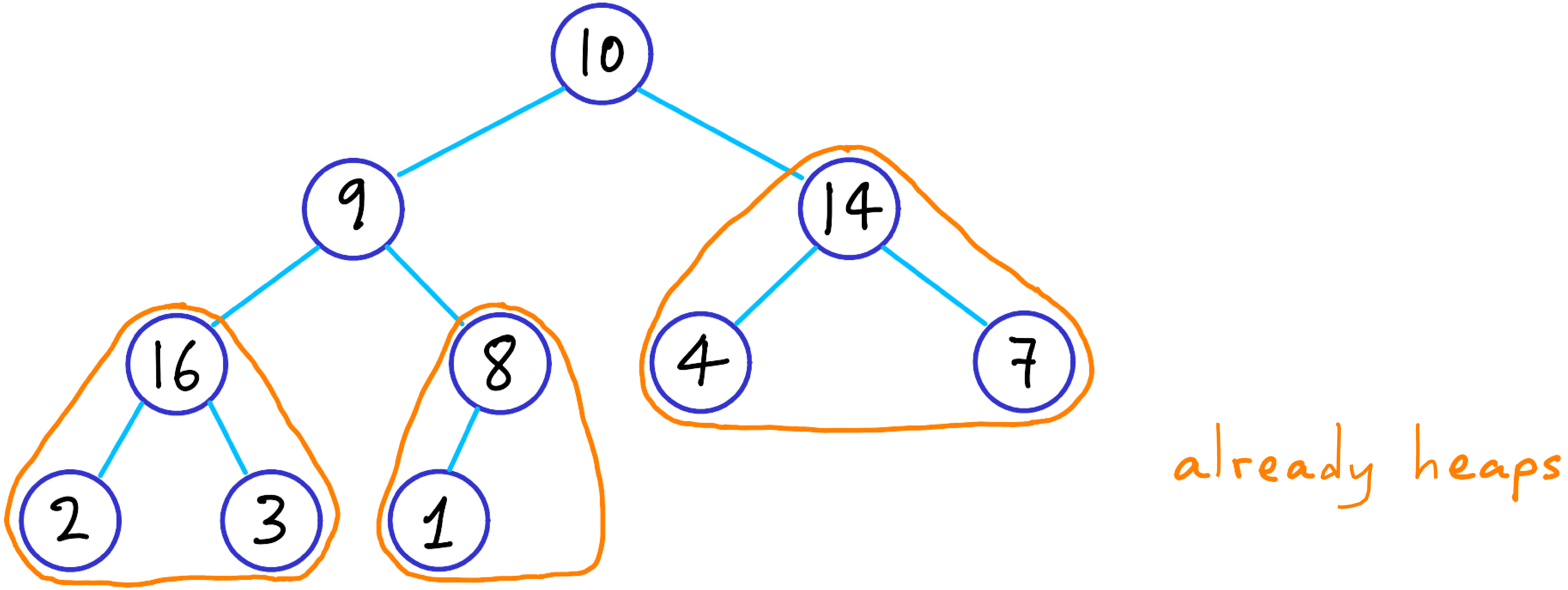
Heap building: the REVERSE METHOD (right to left)

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



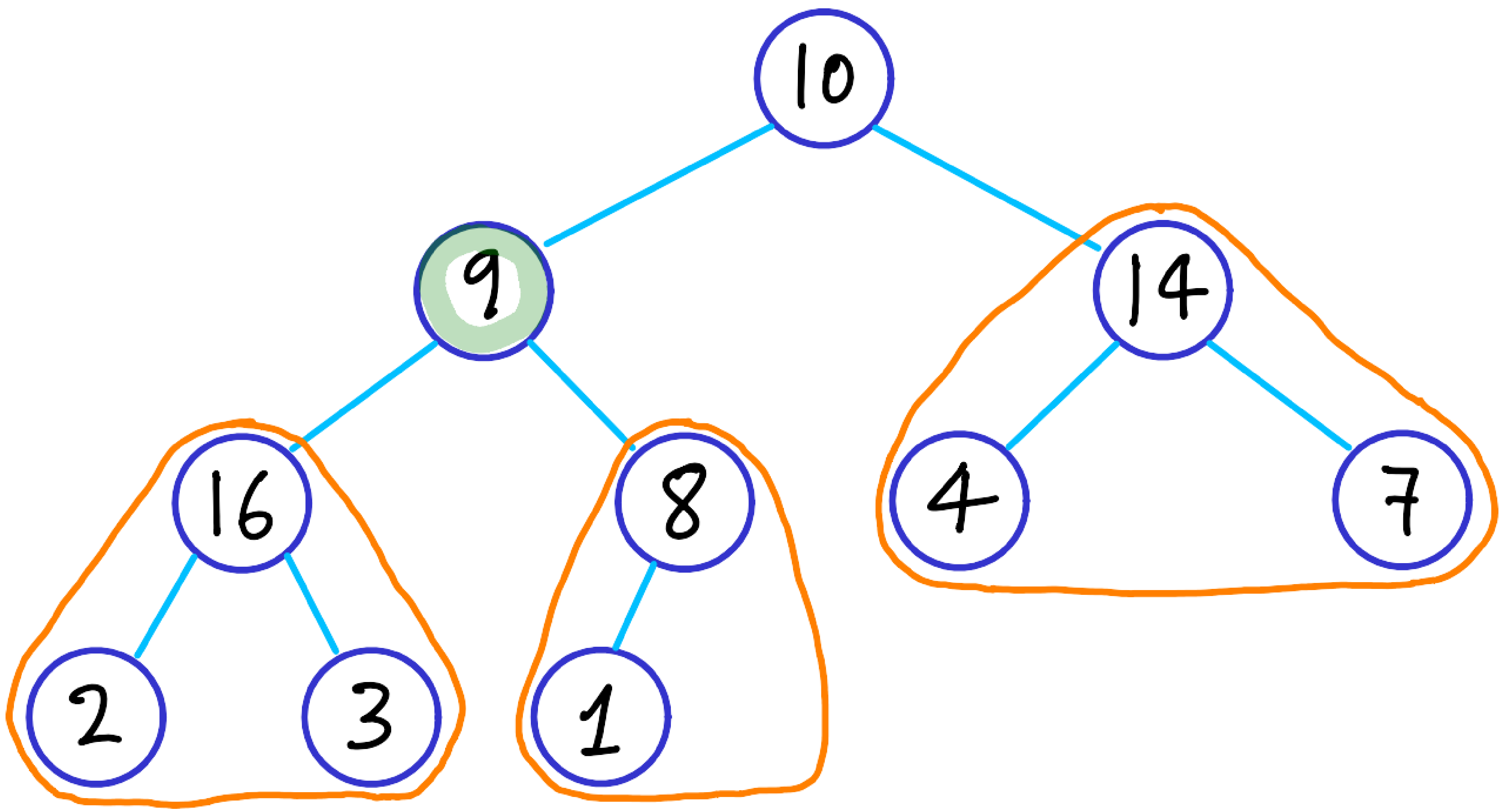
Heap building: the REVERSE METHOD (right to left)

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



Heap building: the REVERSE METHOD (right to left)

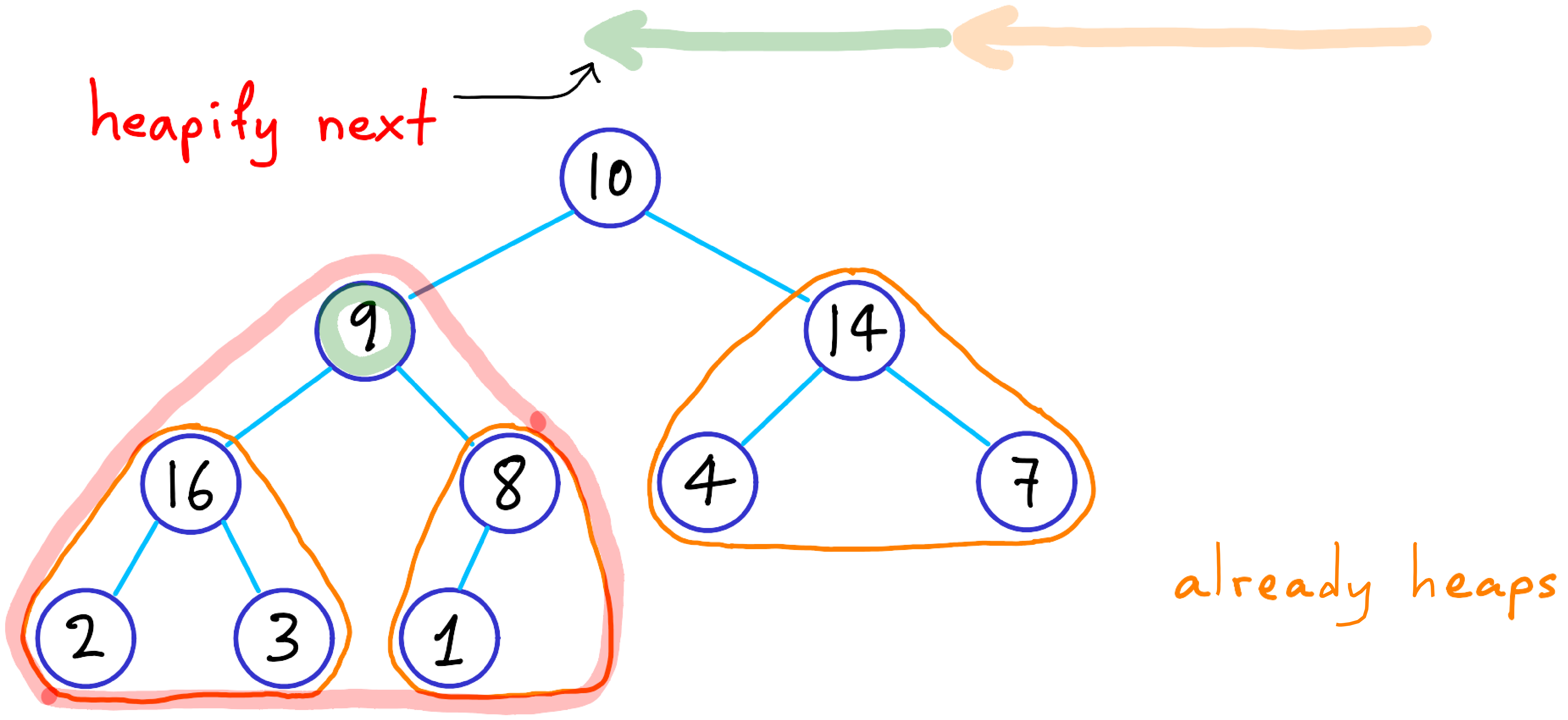
1	2	3	4	5	6	7	8	9	10
10	9	14	16	8	4	7	2	3	1



already heaps

Heap building: the REVERSE METHOD (right to left)

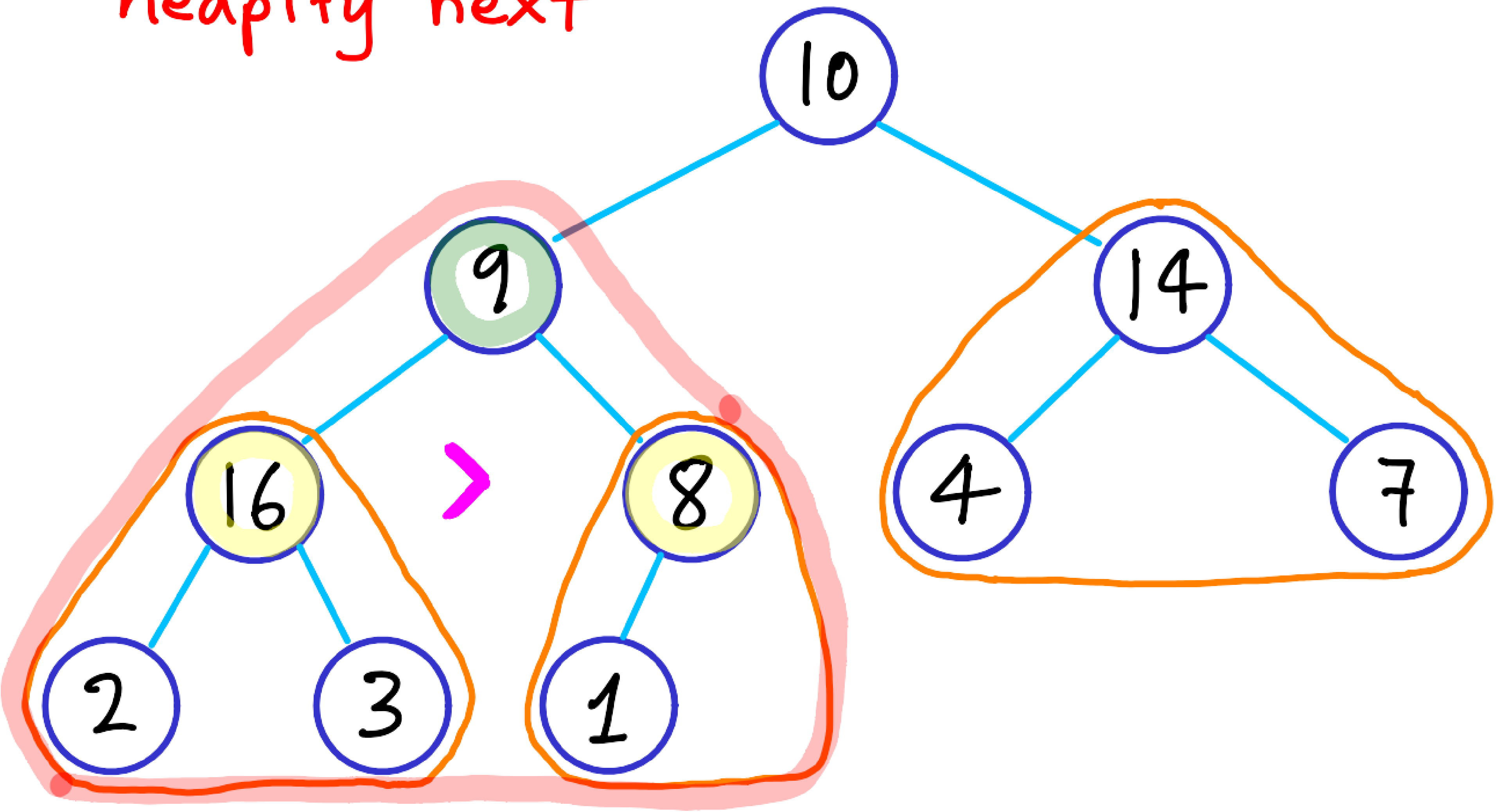
1	2	3	4	5	6	7	8	9	10
10	9	14	16	8	4	7	2	3	1



Heap building: the REVERSE METHOD (right to left)

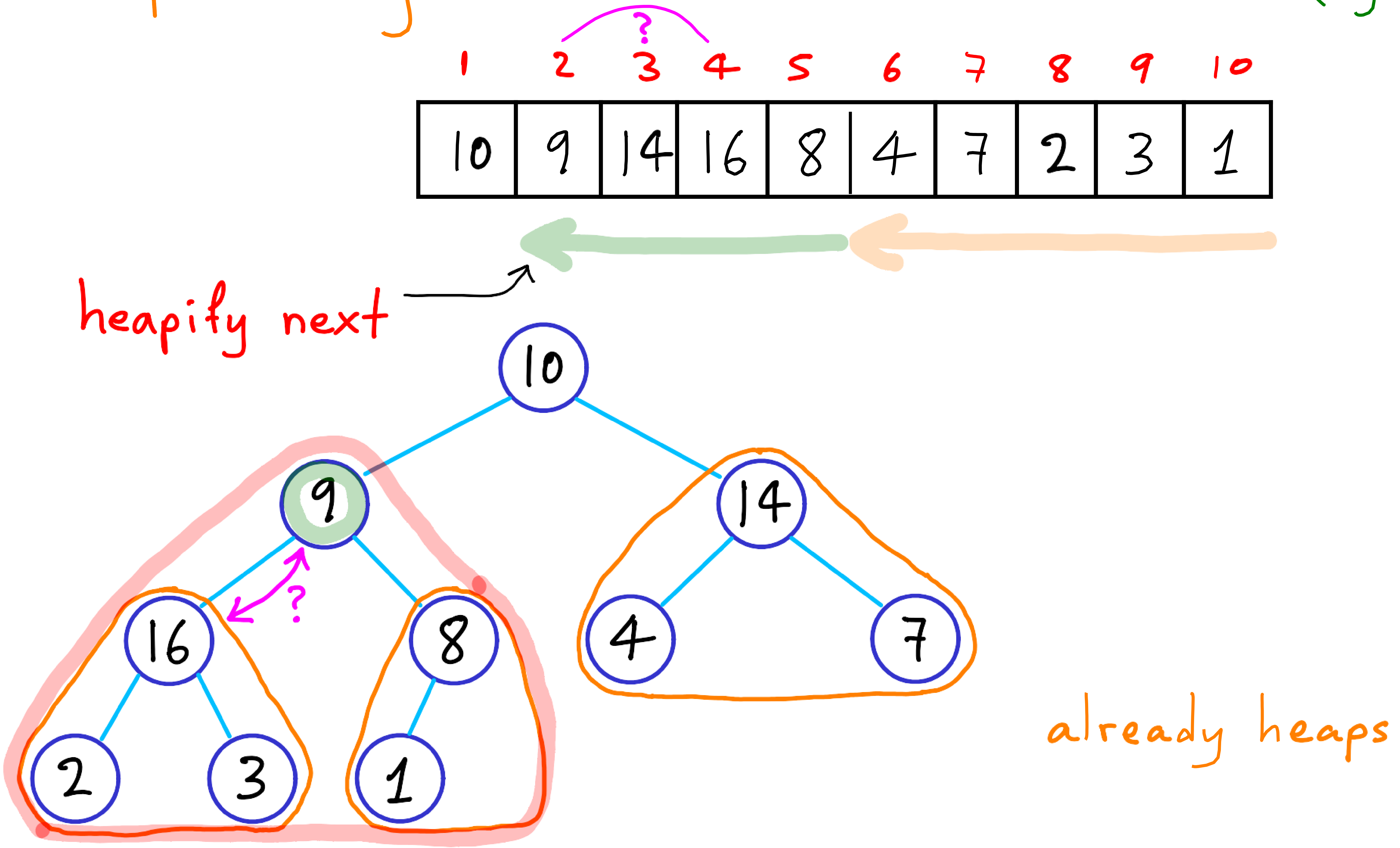
1	2	3	4	5	6	7	8	9	10
10	9	14	16	8	4	7	2	3	1

heapify next



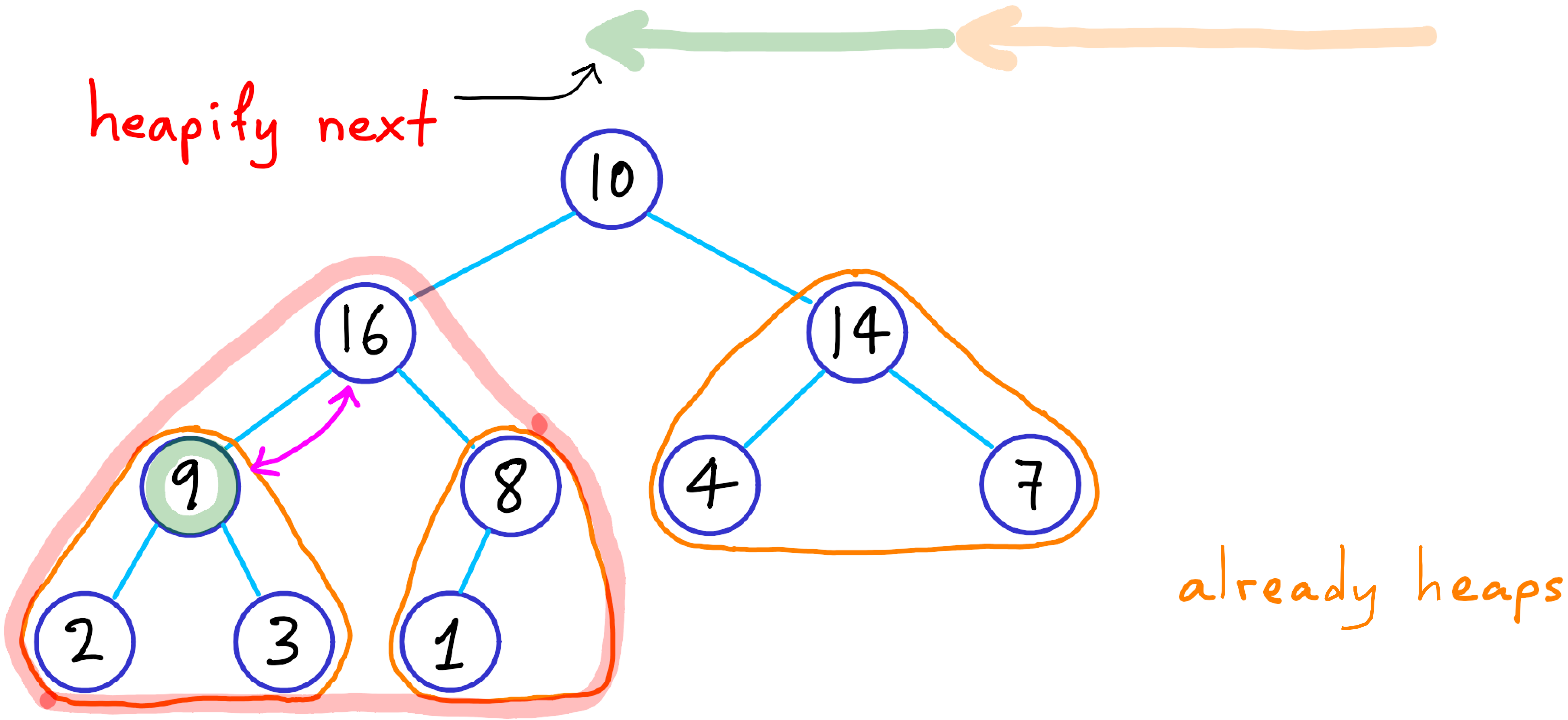
already heaps

Heap building: the REVERSE METHOD (right to left)



Heap building: the REVERSE METHOD (right to left)

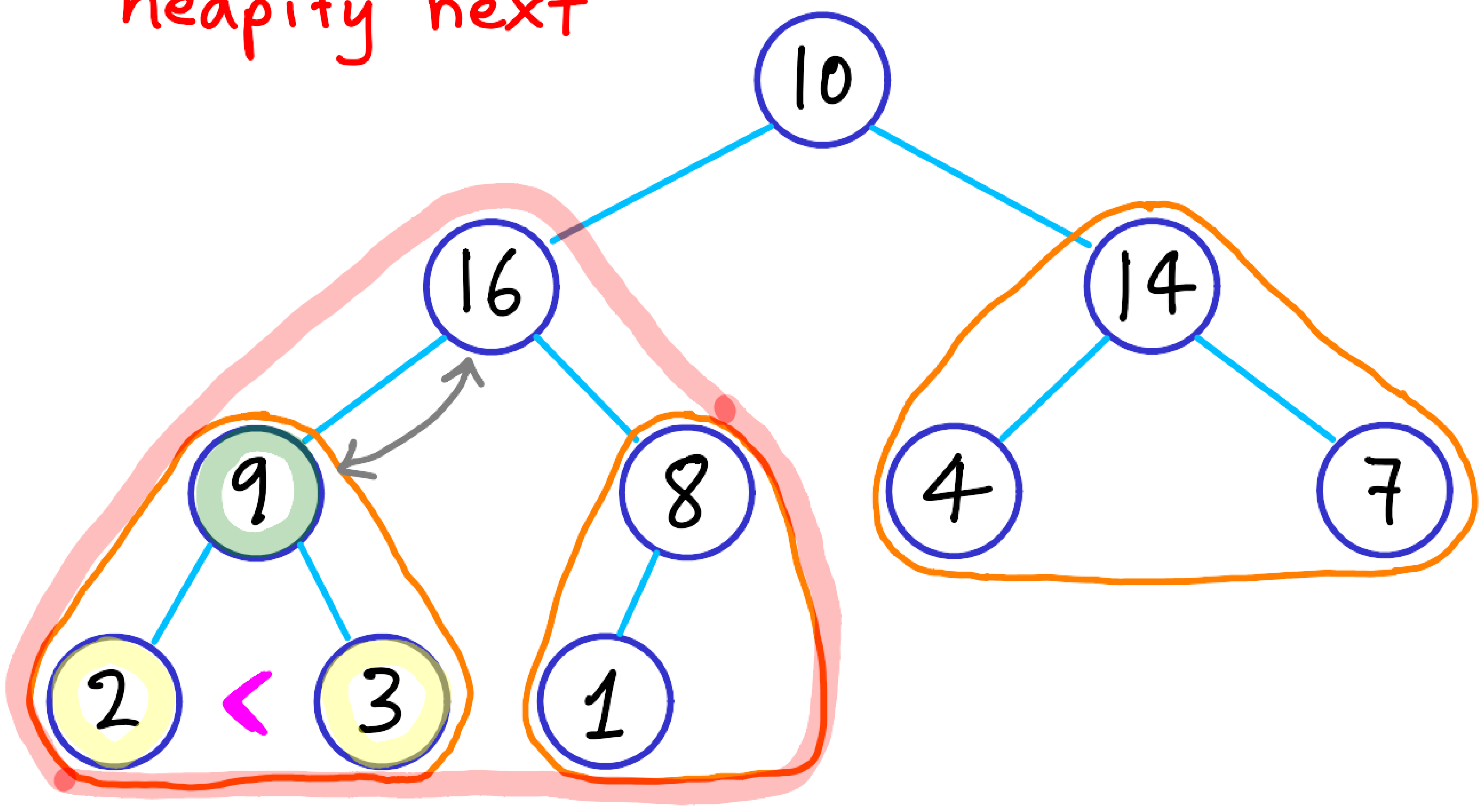
1	2	3	4	5	6	7	8	9	10
10	16	14	9	8	4	7	2	3	1



Heap building: the REVERSE METHOD (right to left)

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

heapify next

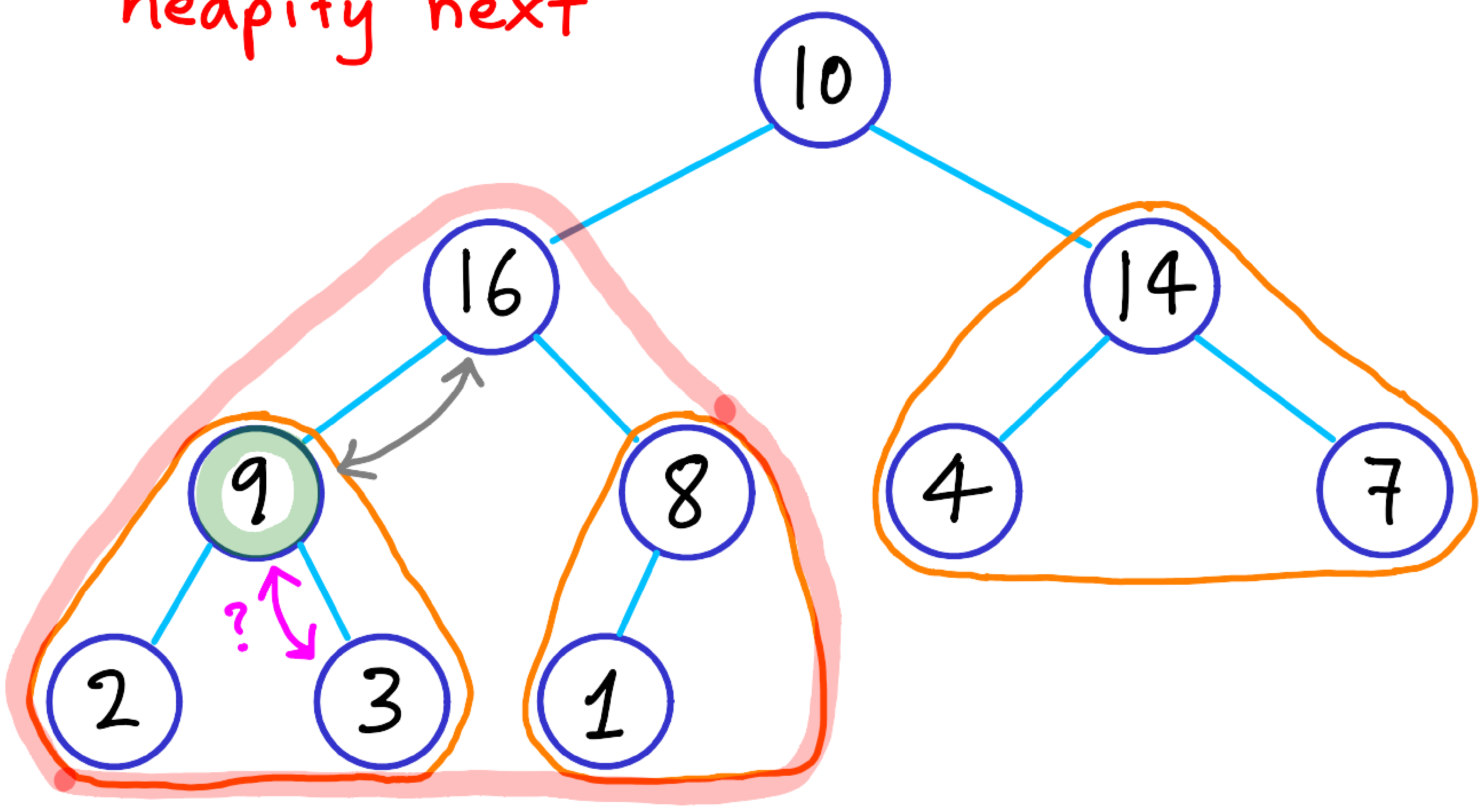


already heaps

Heap building: the REVERSE METHOD (right to left)

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

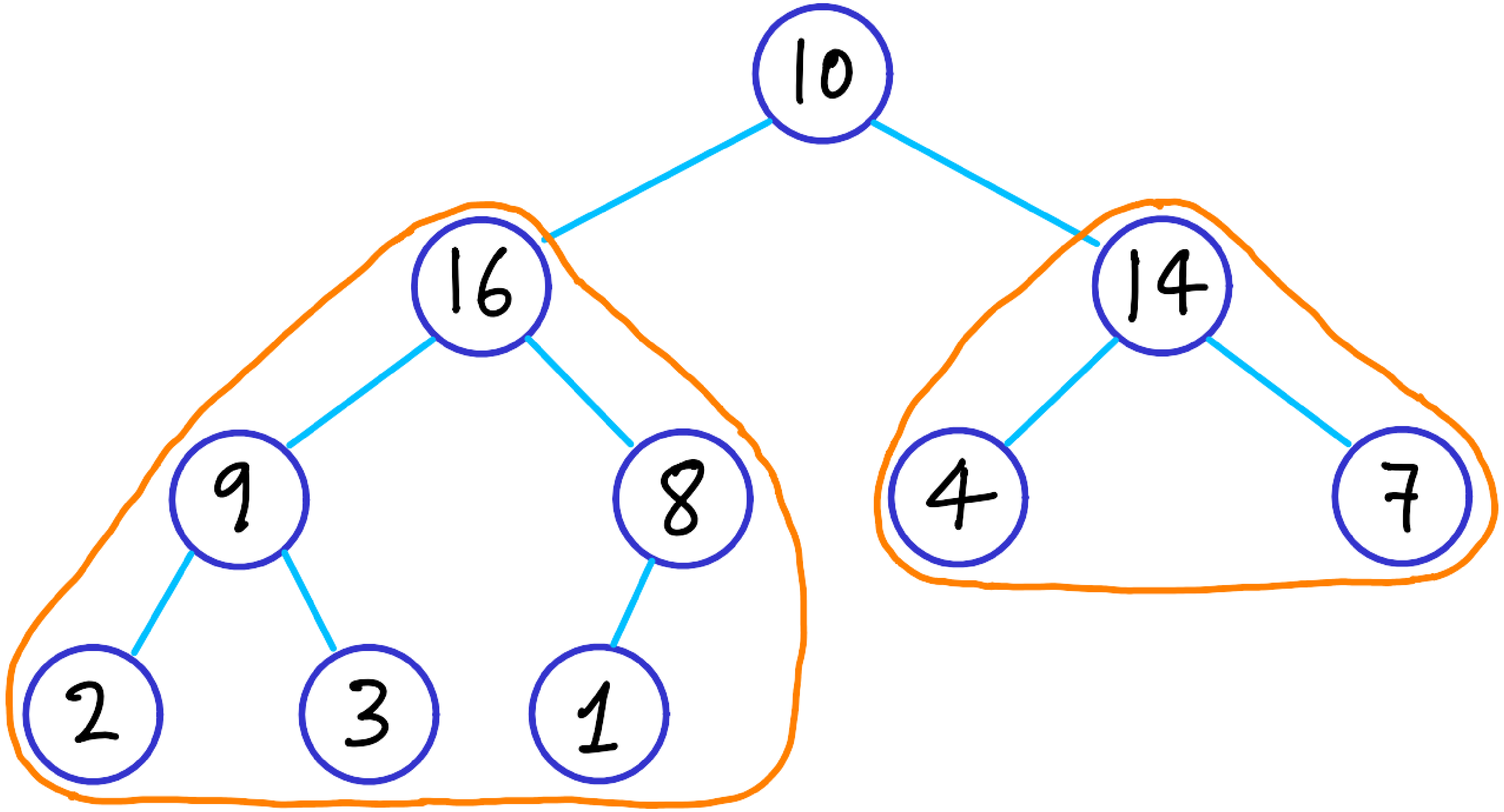
heapify next



already heaps

Heap building: the REVERSE METHOD (right to left)

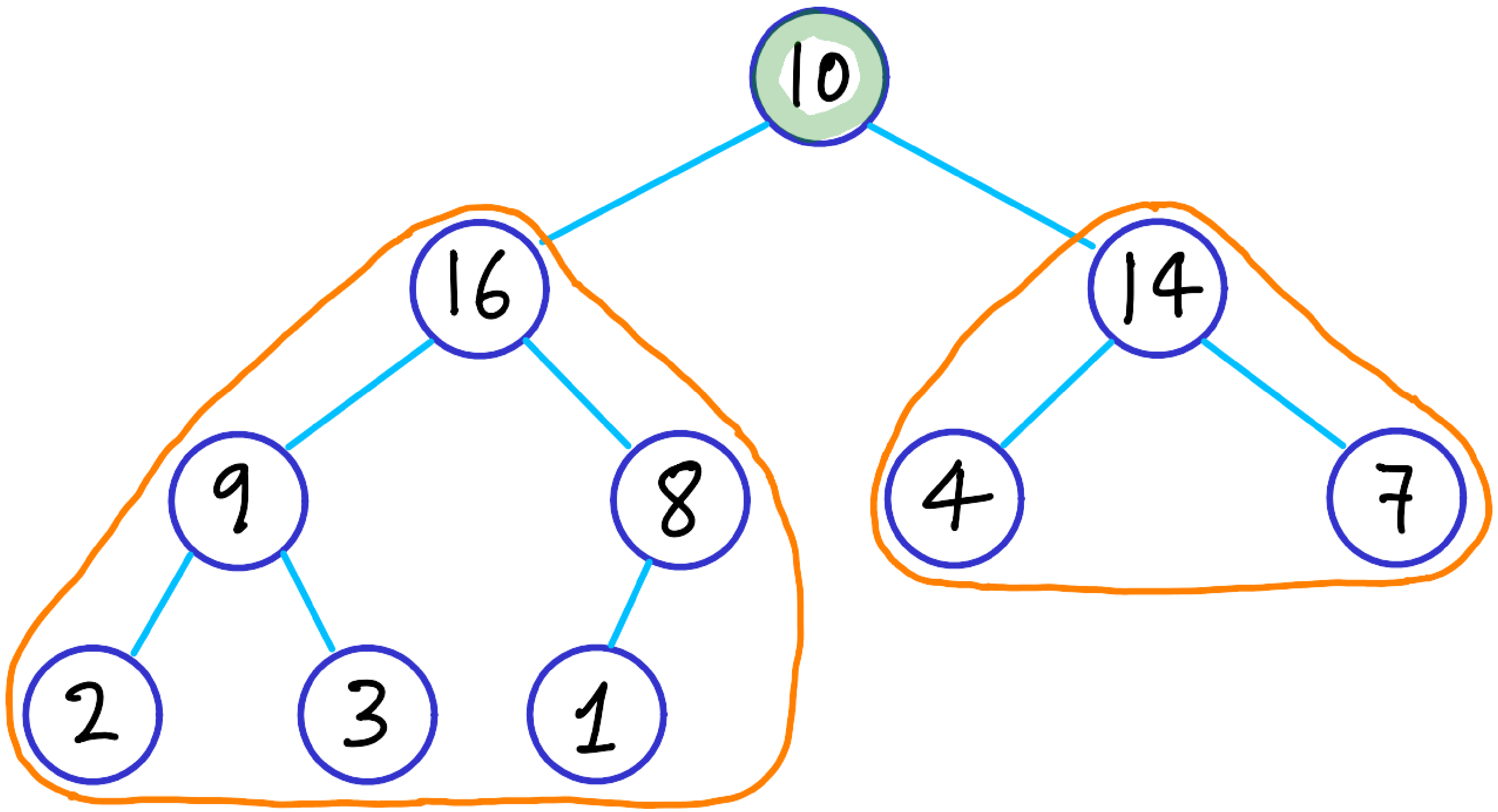
1	2	3	4	5	6	7	8	9	10
10	16	14	9	8	4	7	2	3	1



already heaps

Heap building: the REVERSE METHOD (right to left)

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

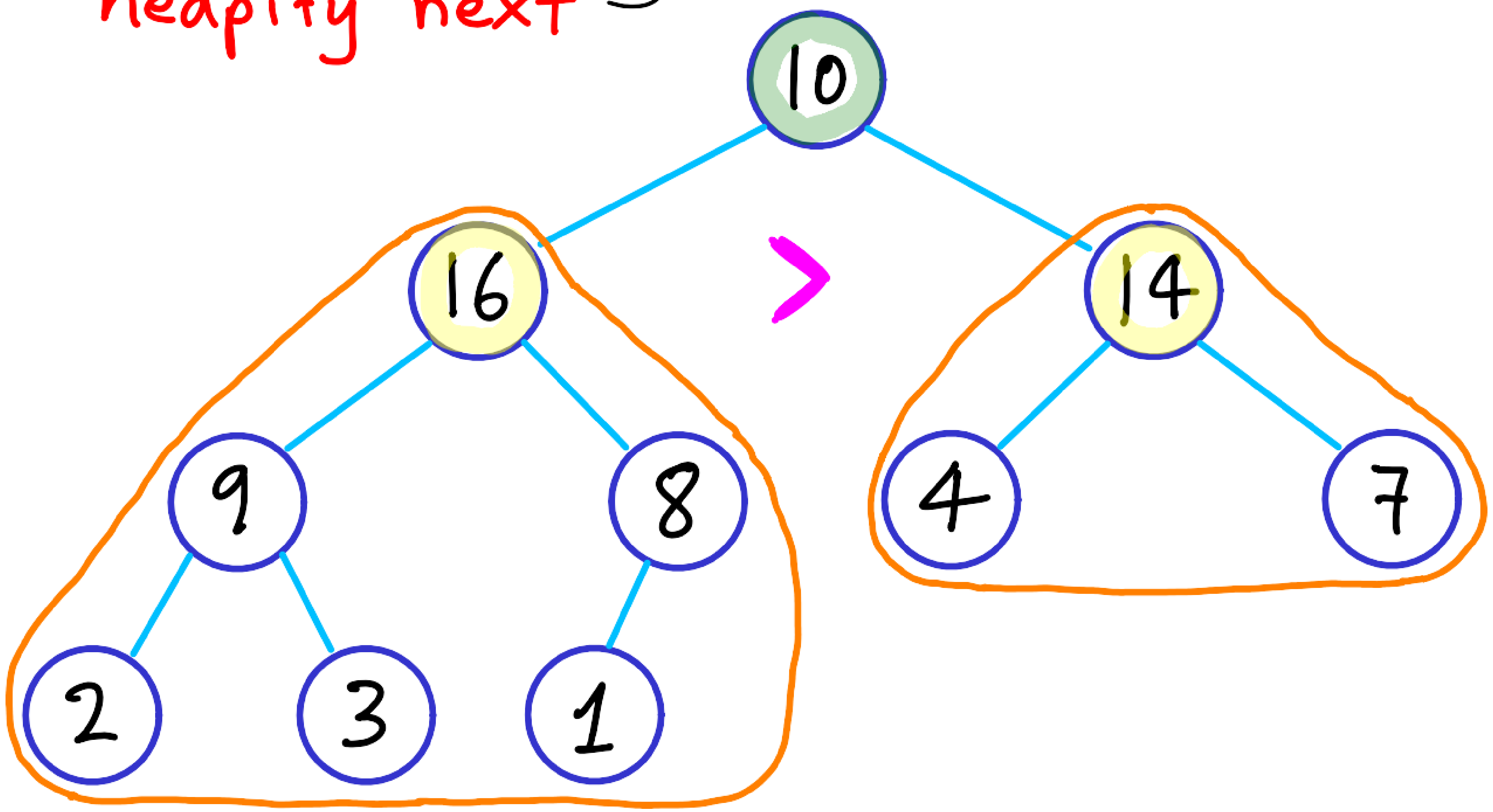


already heaps

Heap building: the REVERSE METHOD (right to left)

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

heapify next

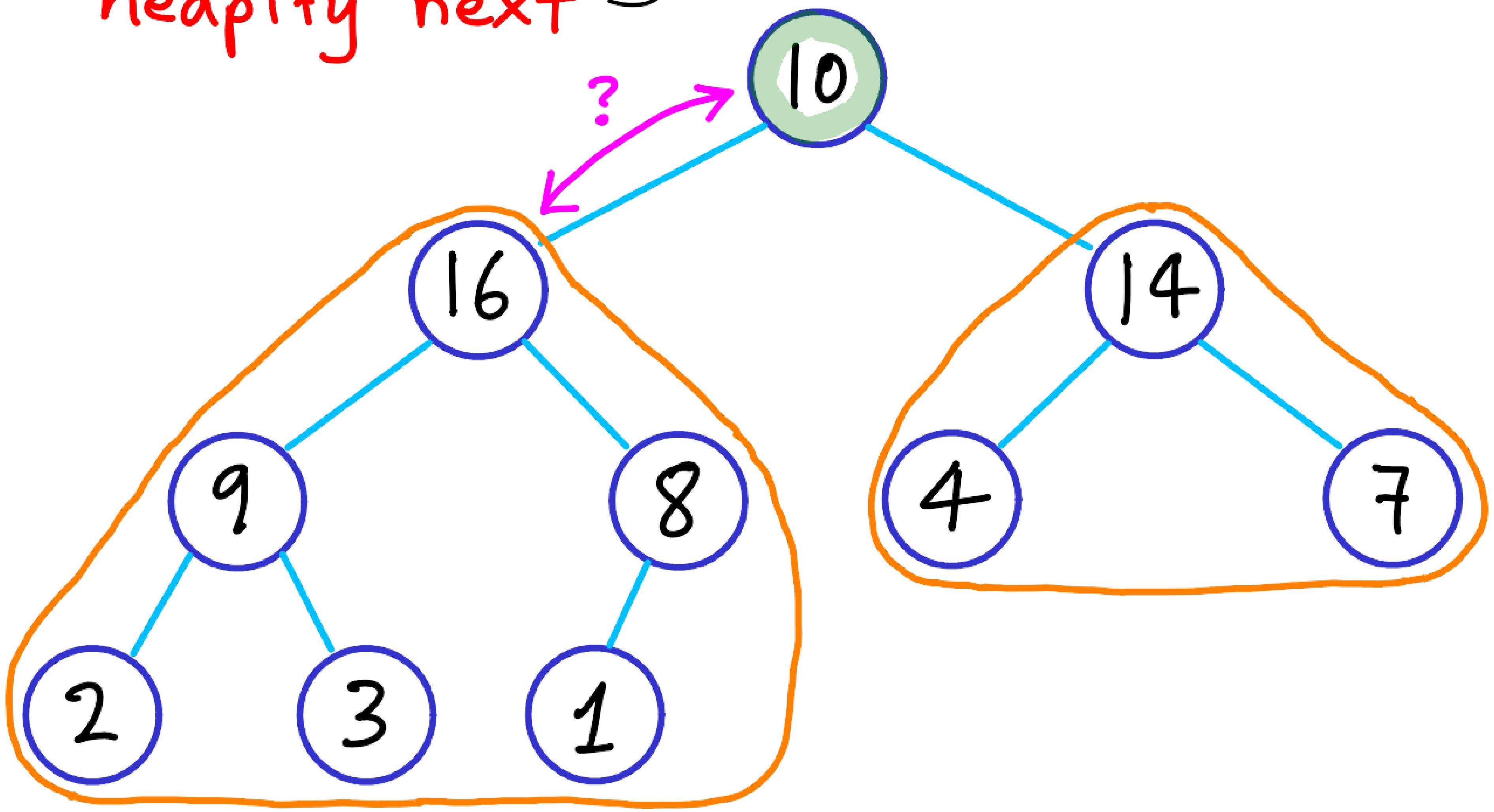


already heaps

Heap building: the REVERSE METHOD (right to left)

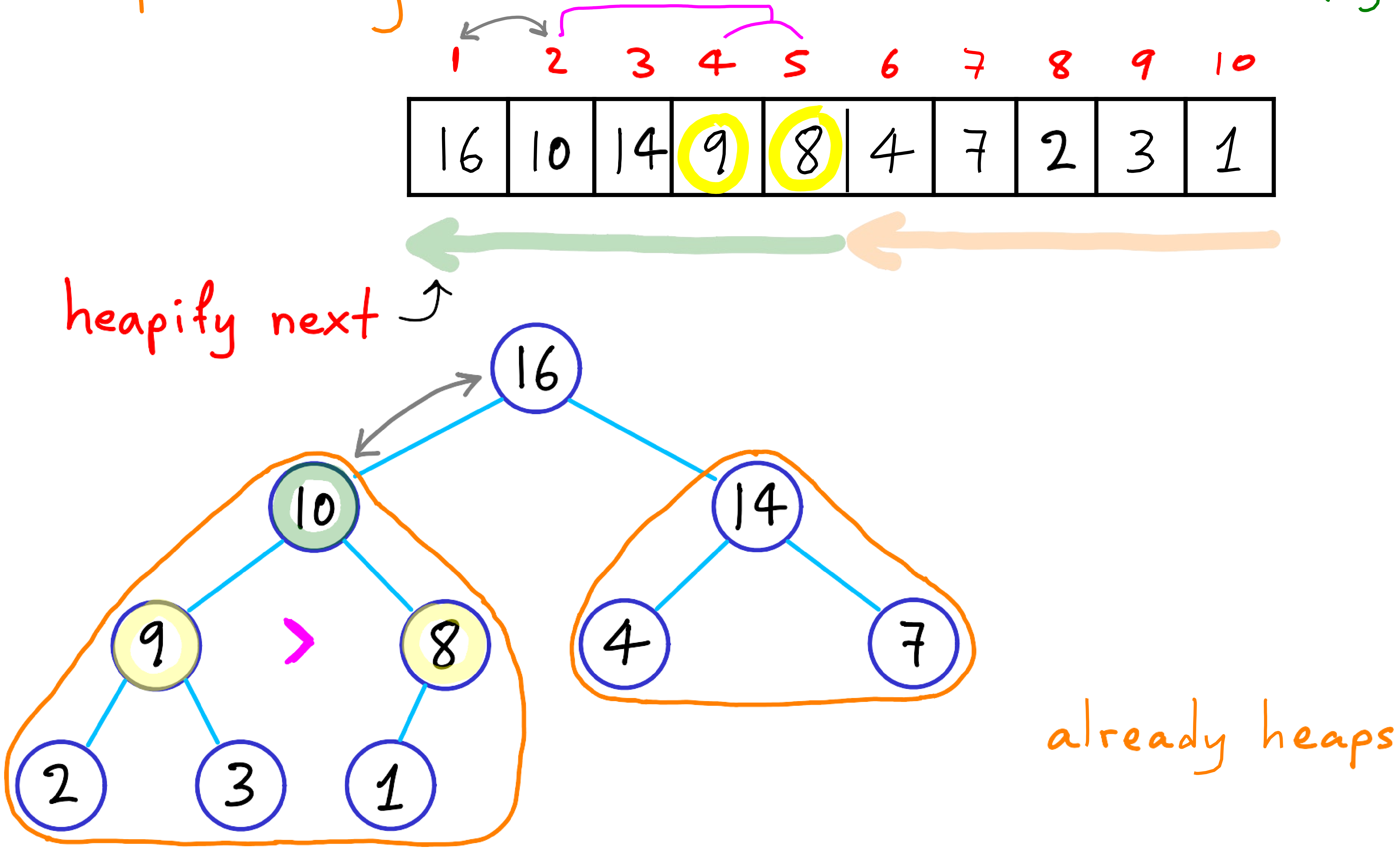
1	2	3	4	5	6	7	8	9	10
10	16	14	9	8	4	7	2	3	1

heapify next



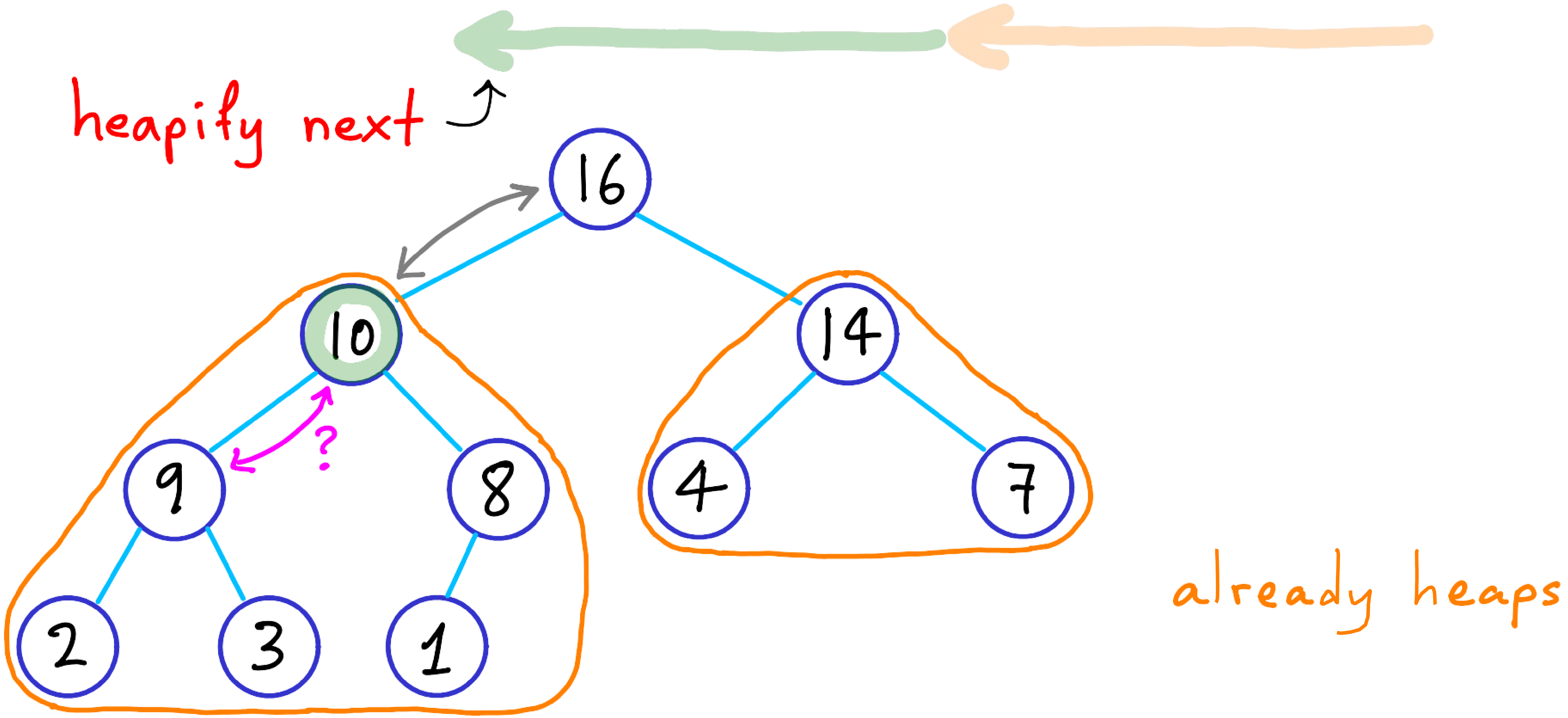
already heaps

Heap building: the REVERSE METHOD (right to left)



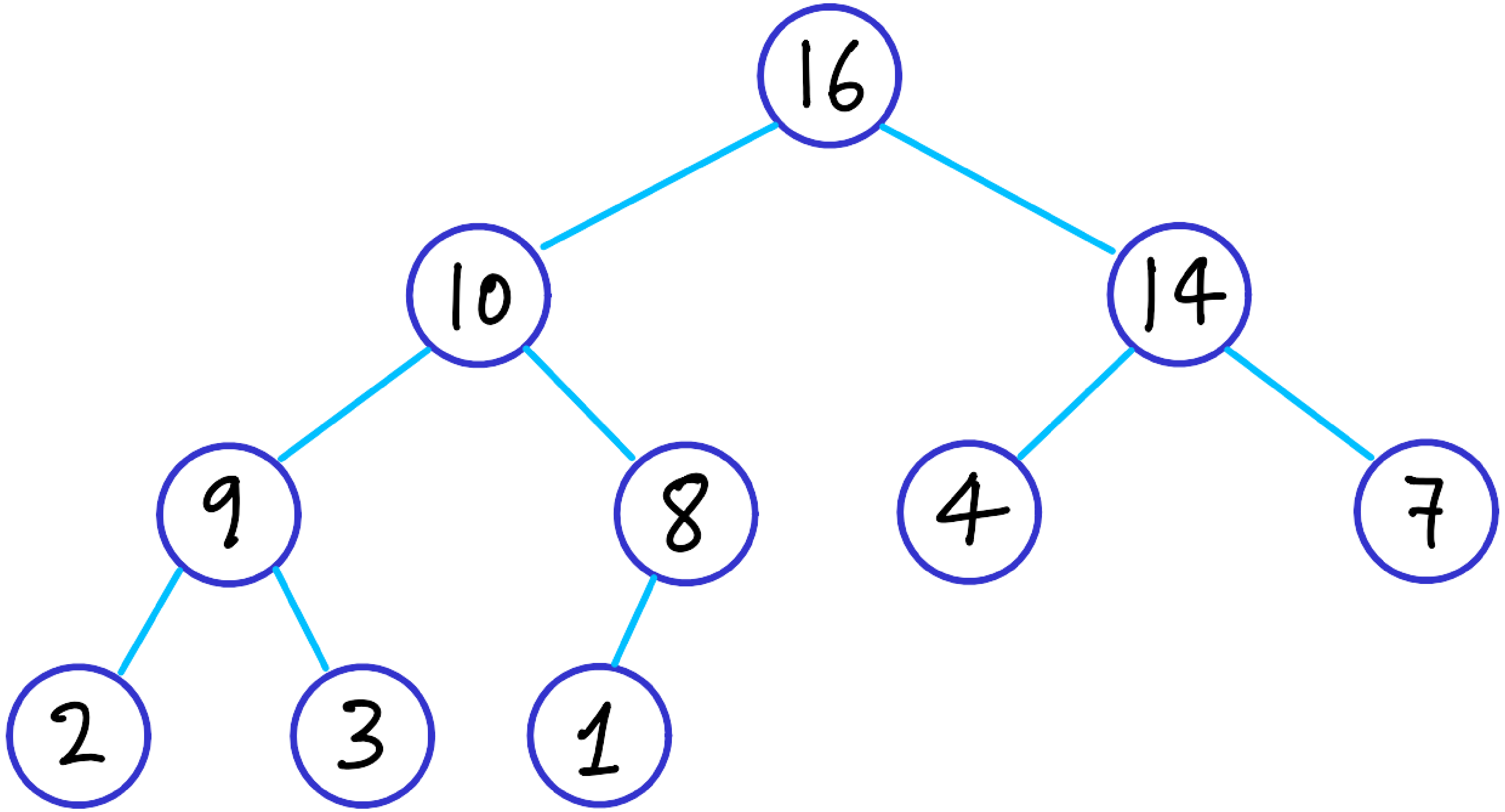
Heap building: the REVERSE METHOD (right to left)

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



Heap building: the REVERSE METHOD (right to left)

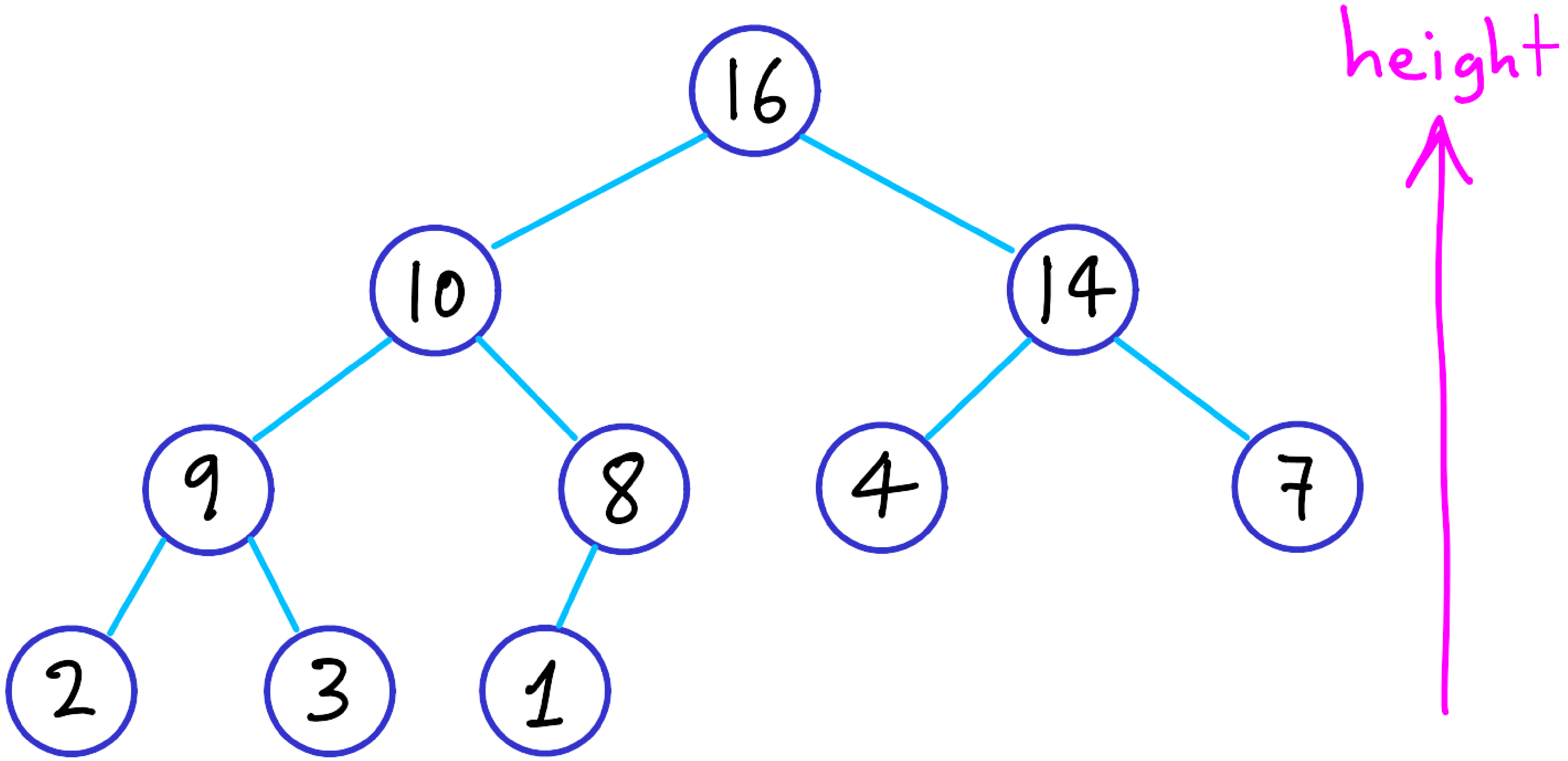
1	2	3	4	5	6	7	8	9	10
16	10	14	9	8	4	7	2	3	1



Time ?

Heap building: the REVERSE METHOD (right to left)

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

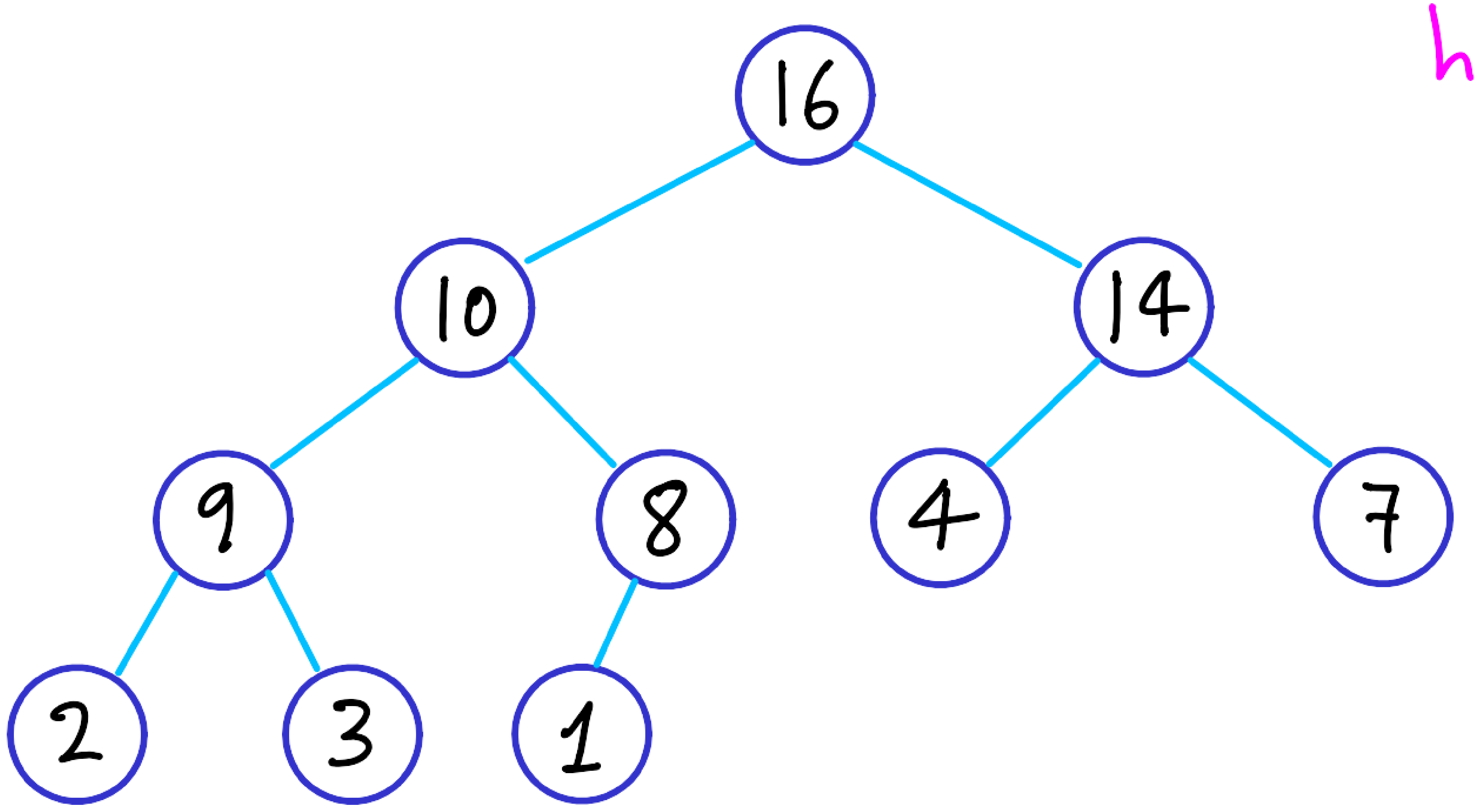


Time ?

$\text{heapify}(x) = O(\text{height}(x))$

Heap building: the REVERSE METHOD (right to left)

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

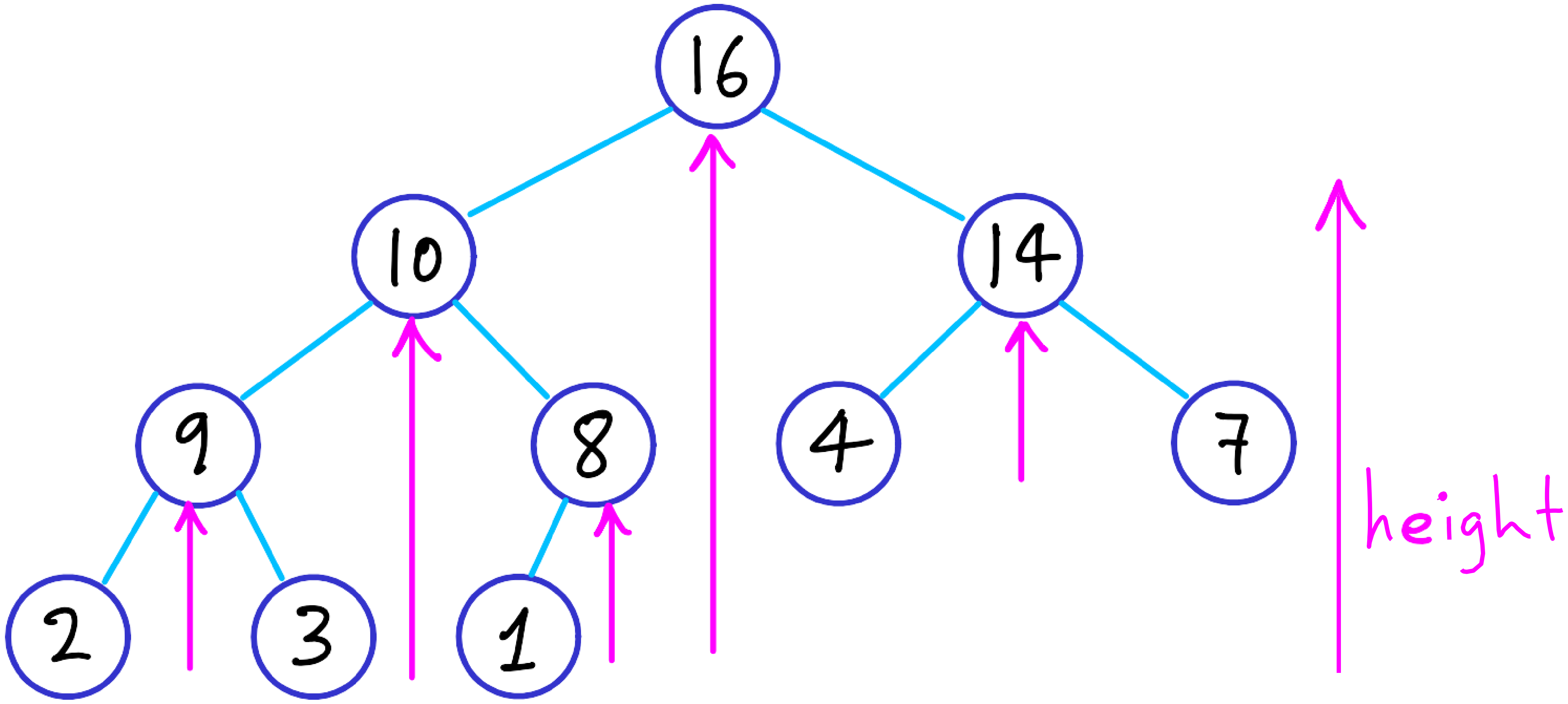


height

Time ?

$$\text{heapify}(x) = O(\text{height}(x))$$

$$\sum_{\text{all } x} \text{height}(x) = O(n \log n)$$



better calculation

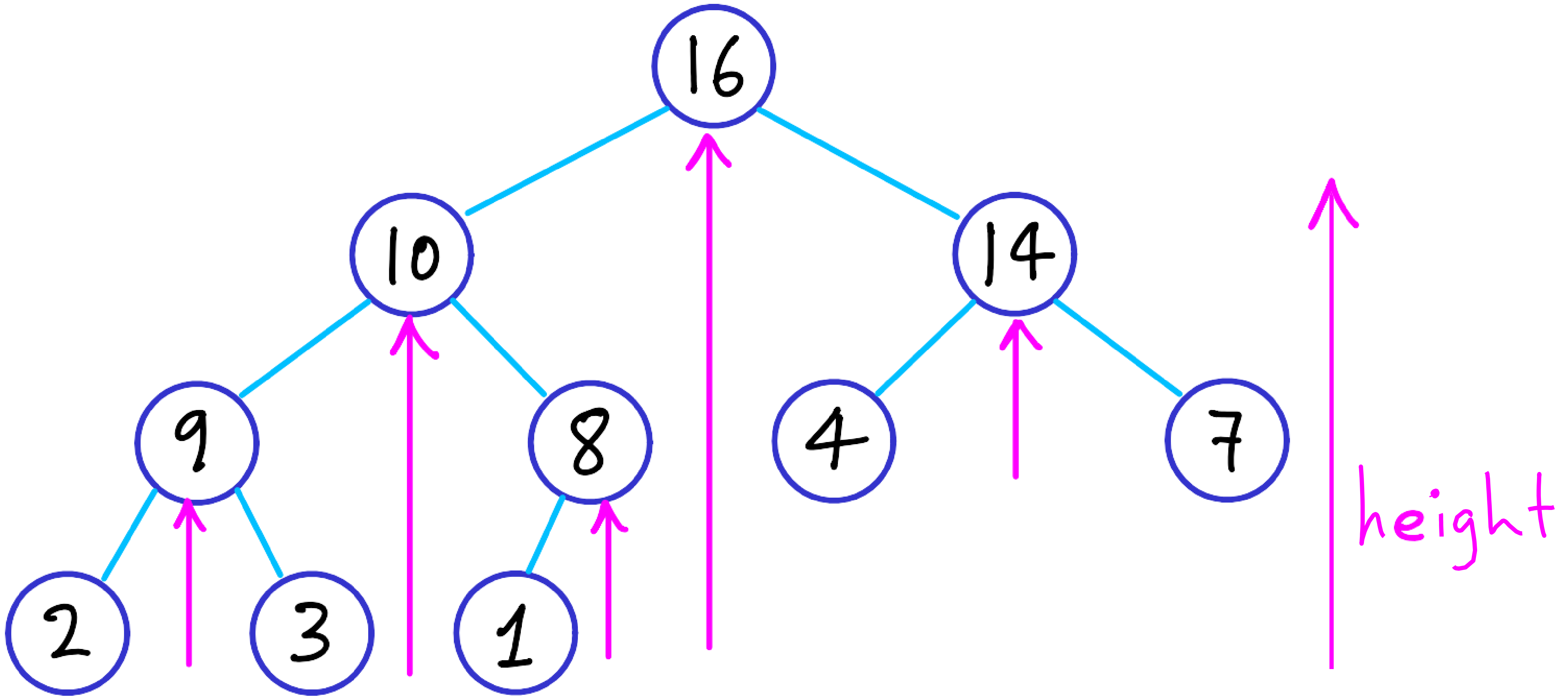
$$\sum_{\text{all } x} \text{height}(x)$$

$\boxed{\text{time?}}$

$O(n)$

better calculation

$$\sum_{\text{all } x} \text{height}(x)$$



$\boxed{\text{time?}}$

$O(n)$

$$\sum \le \underbrace{\frac{n}{2}}_{\#\text{nodes}} \cdot \underbrace{1}_{\text{height}} + \underbrace{\frac{n}{4}}\cdot \underbrace{2} + \underbrace{\frac{n}{8}}\cdot \underbrace{3} + \cdots + \underbrace{2}\cdot \underbrace{((\log n)-1)} + \underbrace{1}_{\#\text{nodes}} \cdot \underbrace{\log n}_{\text{height}}$$

lowest level

root level

Summary

Sequence Type	*insert(k,e)*	*min_element()*	*remove_min()*
Unordered List	***O(1)***	***O(n)***	***O(n)***
Ordered List	***O(n)***	***O(1)***	***O(n)***
Heap	***O(logn)***	***O(1)***	***O(logn)***

Can we do better?

Summary

Sequence Type	***insert(k,e)***	***min_element()***	***remove_min()***
Unordered List	***O(1)***	***O(n)***	***O(n)***
Ordered List	***O(n)***	***O(1)***	***O(n)***
Heap	***O(logn)***	***O(1)***	***O(logn)***
Fibonacci Heaps	***O(1)***	***O(1)***	***O(1)***