CSC 212

Data Structures and Abstractions

Fall 2020

Prof. Marco Alvarez

Priority Queues and Heaps

Collections. Insert and delete items. Which item to delete?

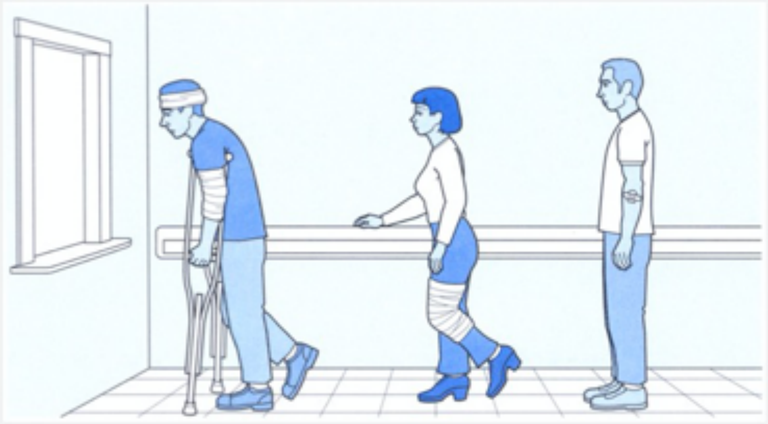
Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Randomized queue. Remove a random item.

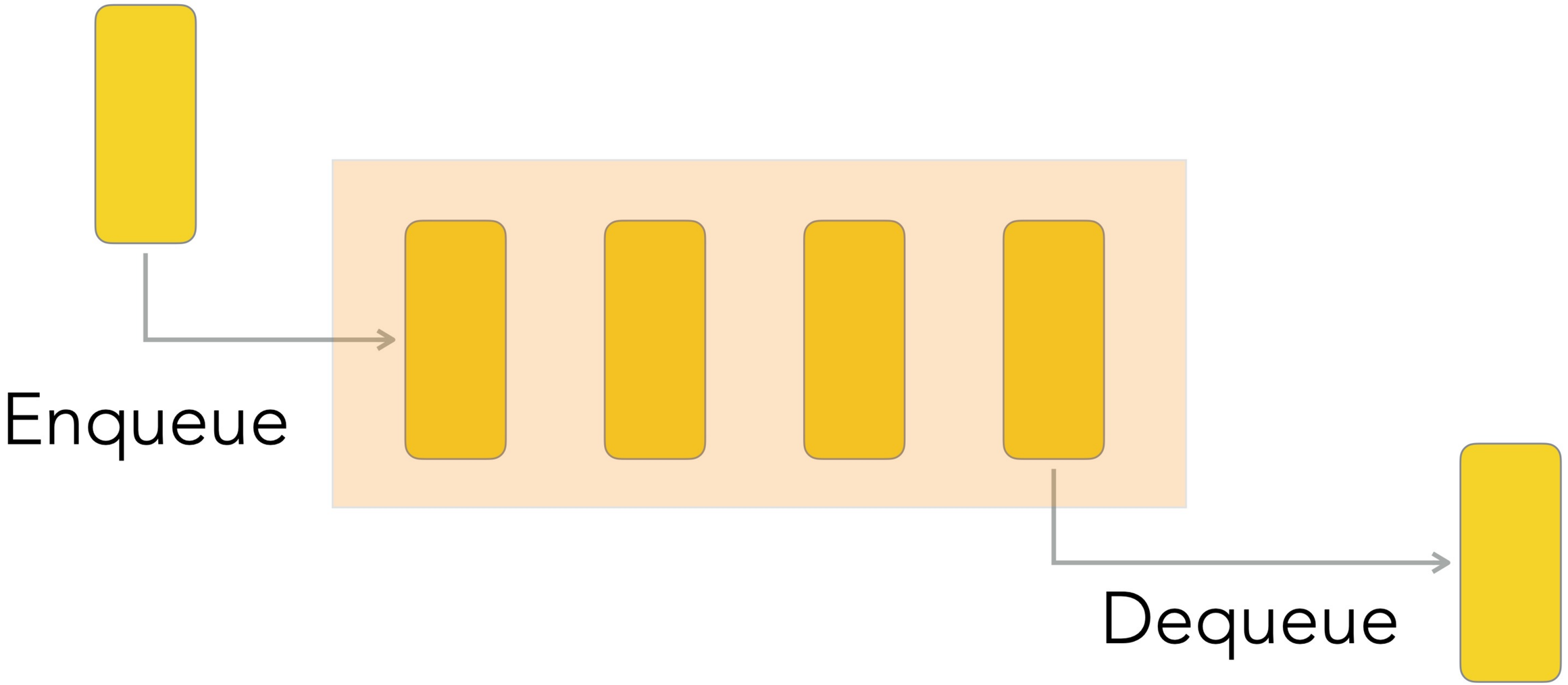
Priority queue. Remove the largest (or smallest) item.

Generalizes: stack, queue, randomized queue.

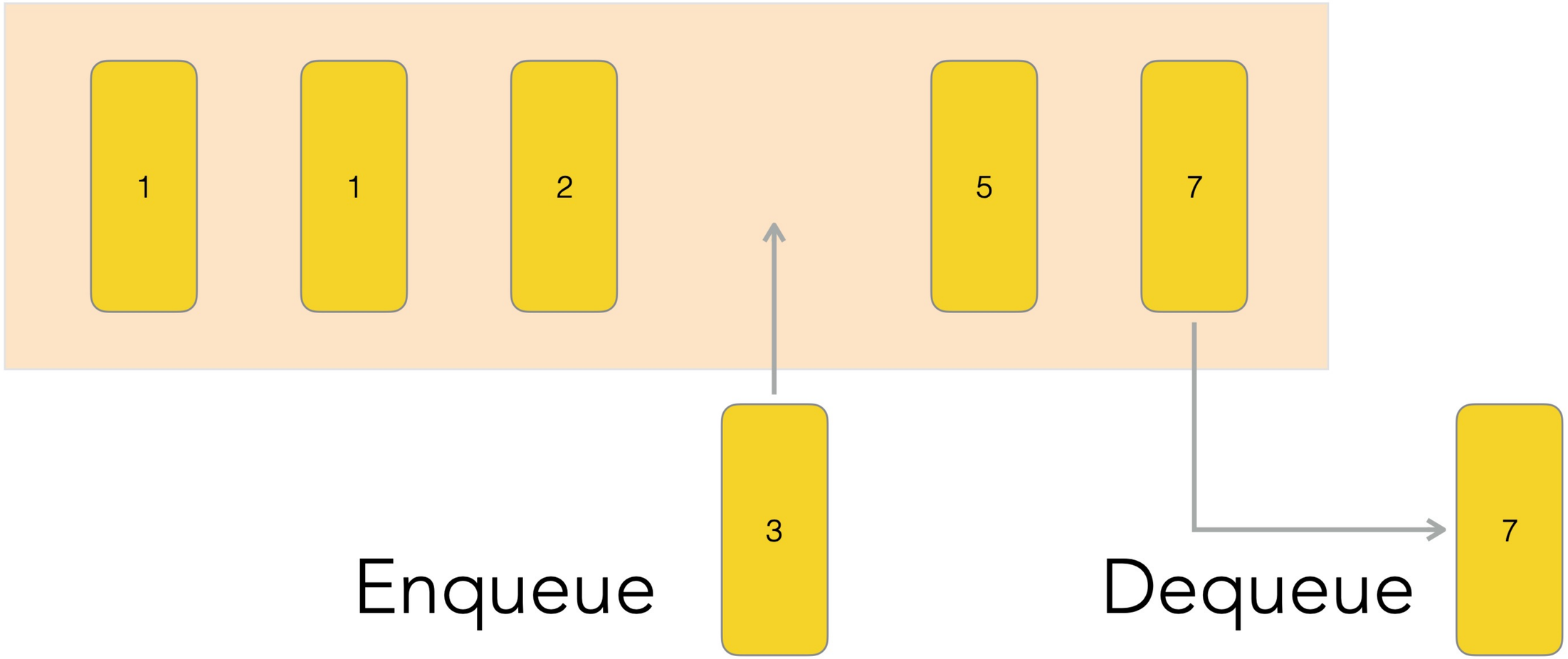


credit: COS 226, Princeton University

Queues



Priority Queues



Applications

Data Compression (huffman trees)

Network Routing

Process Scheduling (CPUs)

Artificial Intelligence (search)

Graph Algorithms

Stream Data Algorithms

HPC Task Scheduling

• • •

Priority Queues

Collections of <Key,Value> pairs

keys are objects on which an **order** is defined

Priority Queues

Collections of <Key,Value> pairs

keys are objects on which an **order** is defined

Every pair of keys must be comparable according to a **total order:**

Priority Queues

Collections of <Key,Value> pairs

keys are objects on which an **order** is defined

Every pair of keys must be comparable according to a **total order:**

$$\text{Reflexive Property: } k \leq k$$

$$\text{Antisymmetric Property: if } k_1 \leq k_2 \text{ and } k_2 \leq k_1, \text{ then } k_1 = k_2$$

$$\text{Transitive Property: if } k_1 \leq k_2 \text{ and } k_2 \leq k_3, \text{ then } k_1 \leq k_3$$

Priority Queues

Queues

basic operations: **enqueue**, **dequeue**

always remove the item least recently added

Priority Queues

Queues

basic operations: **enqueue, dequeue**

always remove the item least recently added

Priority Queues (MaxPQ)

basic operations: **insert, removeMax**

always remove the item with **highest (max) priority**

Priority Queues

Queues

basic operations: **enqueue, dequeue**

always remove the item least recently added

Priority Queues (MaxPQ)

basic operations: **insert, removeMax**

always remove the item with **highest (max) priority**

Can also be implemented as a MinPQ

Example (MinPQ)

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

From Algorithm Design and Applications, Goodrich & Tamassia

Performance?

	Sorted Array/List	Unsorted Array/List
insert		
removeMax		
max		

Performance

	Sorted Array/List	Unsorted Array/List
insert	O(n)	O(1)
removeMax	O(1)	O(n)
max	O(1)	O(n)

std::priority_queue

Defined in header <queue>

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

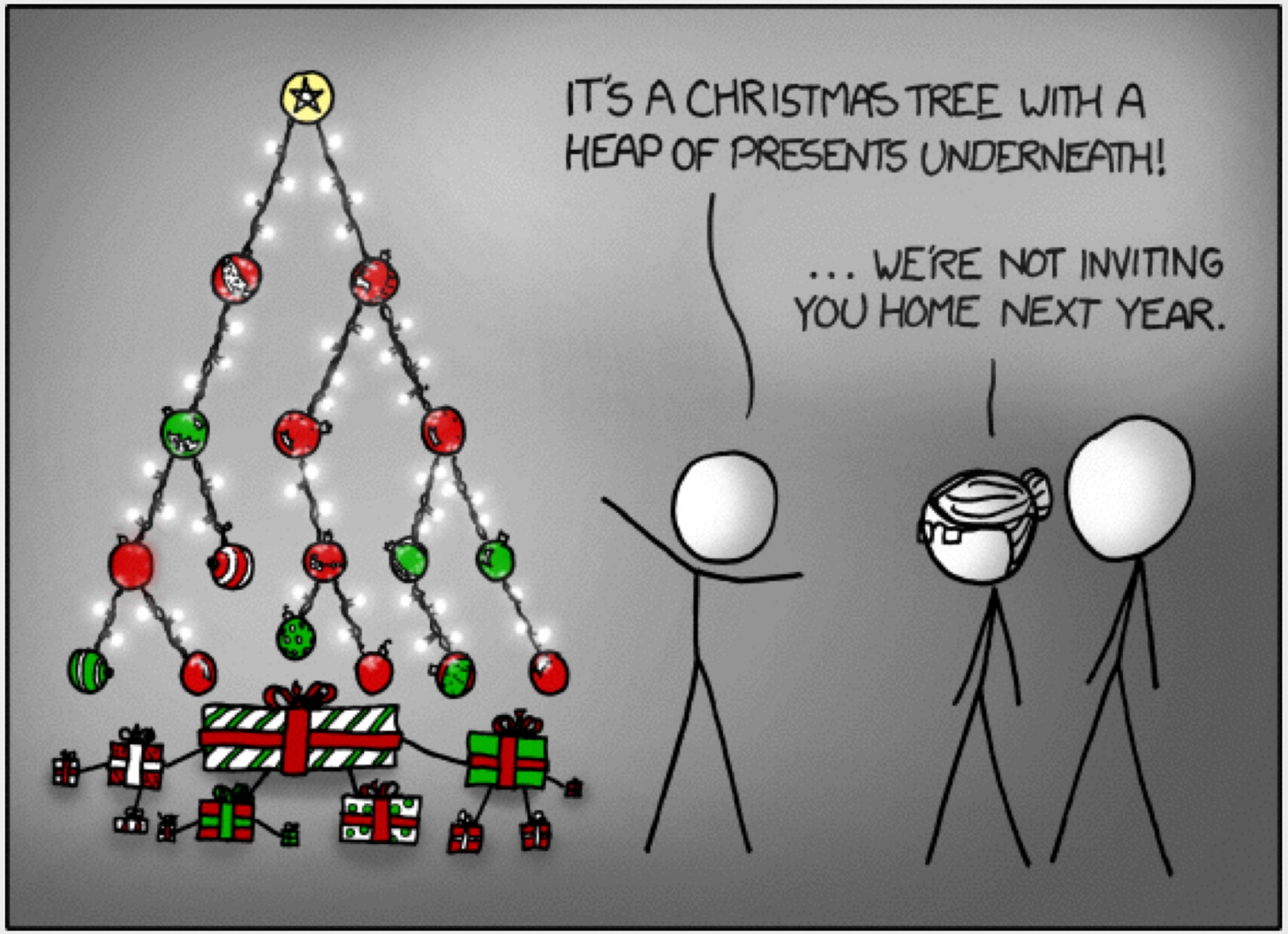
A priority queue is a container adaptor that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.

A user-provided `Compare` can be supplied to change the ordering, e.g. using `std::greater<T>` would cause the smallest element to appear as the `top()`.

Working with a `priority_queue` is similar to managing a heap in some random access container, with the benefit of not being able to accidentally invalidate the heap.

Member functions

(constructor)	constructs the `priority_queue` (public member function)
(destructor)	destructs the `priority_queue` (public member function)
operator=	assigns values to the container adaptor (public member function)
Element access	
top	accesses the top element (public member function)
Capacity	
empty	checks whether the underlying container is empty (public member function)
size	returns the number of elements (public member function)
Modifiers	
push	inserts element and sorts the underlying container (public member function)
emplace (C++11)	constructs element in-place and sorts the underlying container (public member function)
pop	removes the top element (public member function)
swap	swaps the contents (public member function)



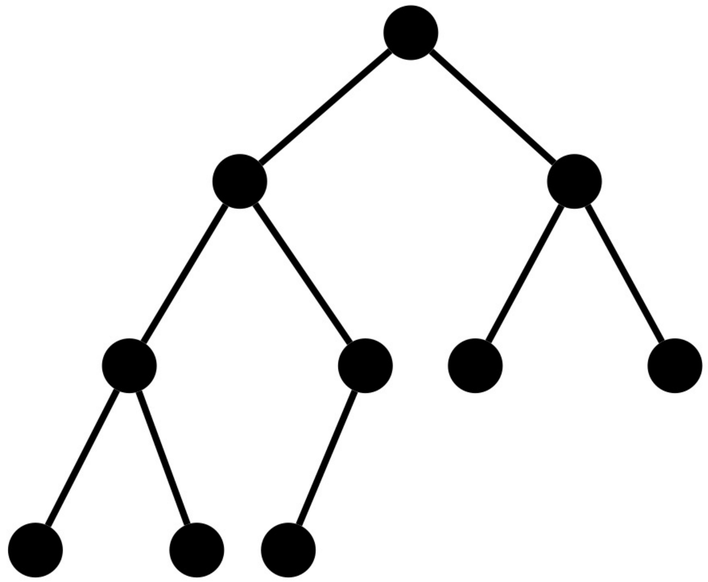
From https://xkcd.com/835/

Heaps

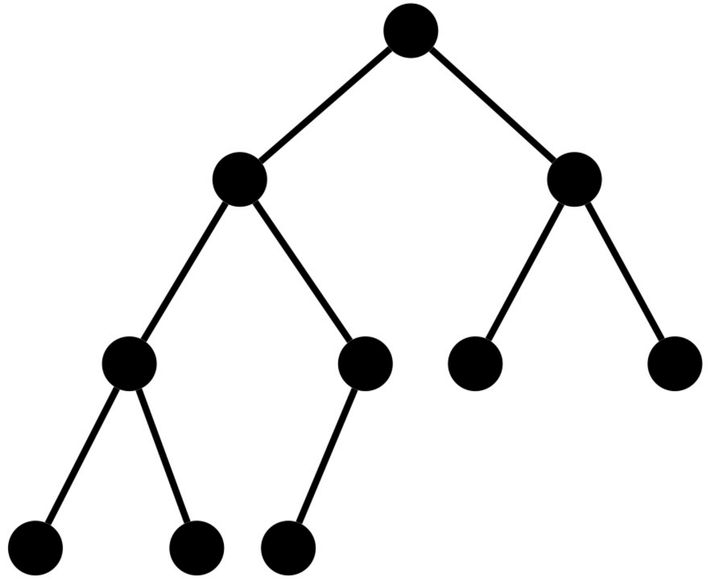
(max) Heap

Structure Property

a heap is a **complete binary tree**



(max) Heap



Structure Property

a heap is a **complete binary tree**

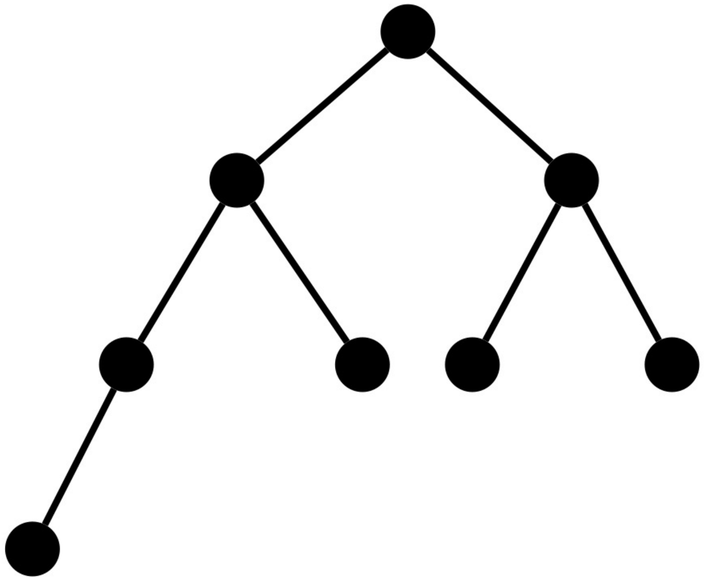
Heap-Order Property

for every node **x**, **key(parent(x)) >= key(x)**

except the root, which has no parent

Height of a heap?

What is the minimum number of nodes in a complete binary tree of height **h**?



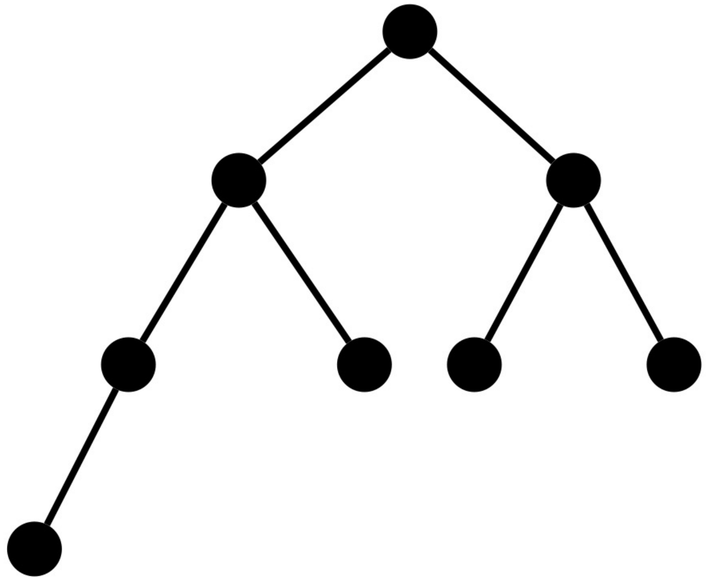
Height of a heap?

What is the minimum number of nodes in a complete binary tree of height **h**?

$$n \geq 2^h$$

$$\log n \geq \log 2^h$$

$$\log n \geq h$$



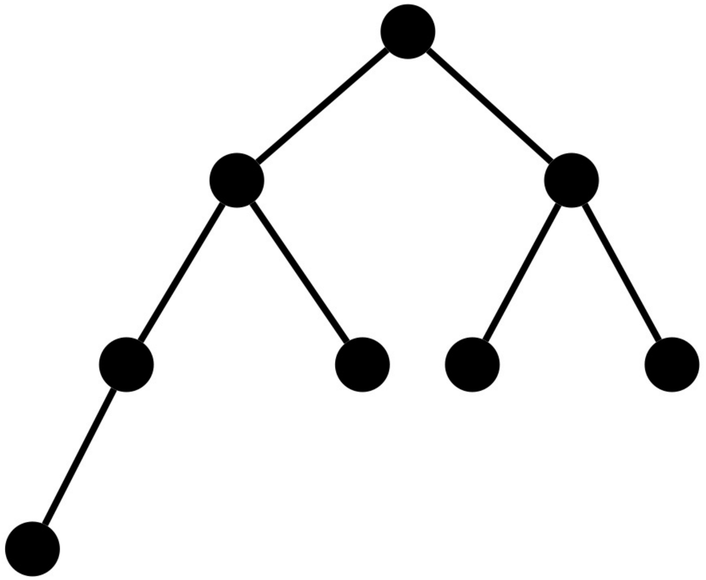
Height of a heap?

What is the minimum number of nodes in a complete binary tree of height **h**?

$$n \geq 2^h$$

$$\log n \geq \log 2^h$$

$$\log n \geq h$$



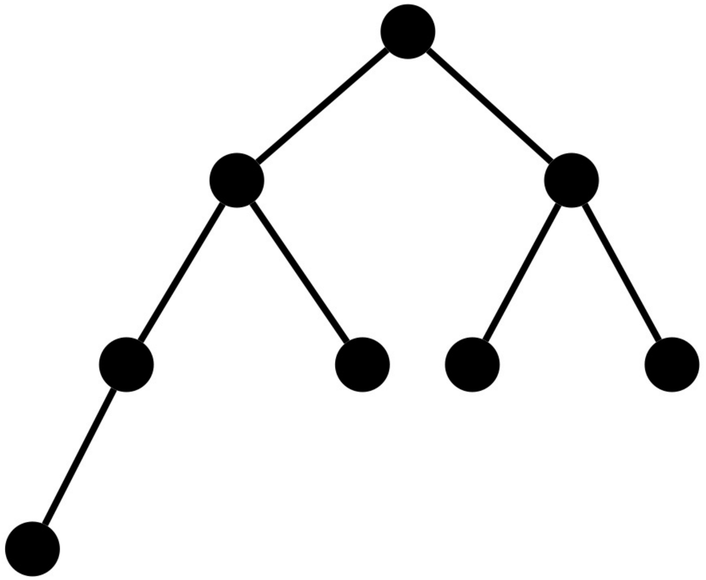
Height of a heap?

What is the minimum number of nodes in a complete binary tree of height **h**?

$$n \geq 2^h$$

$$\log n \geq \log 2^h$$

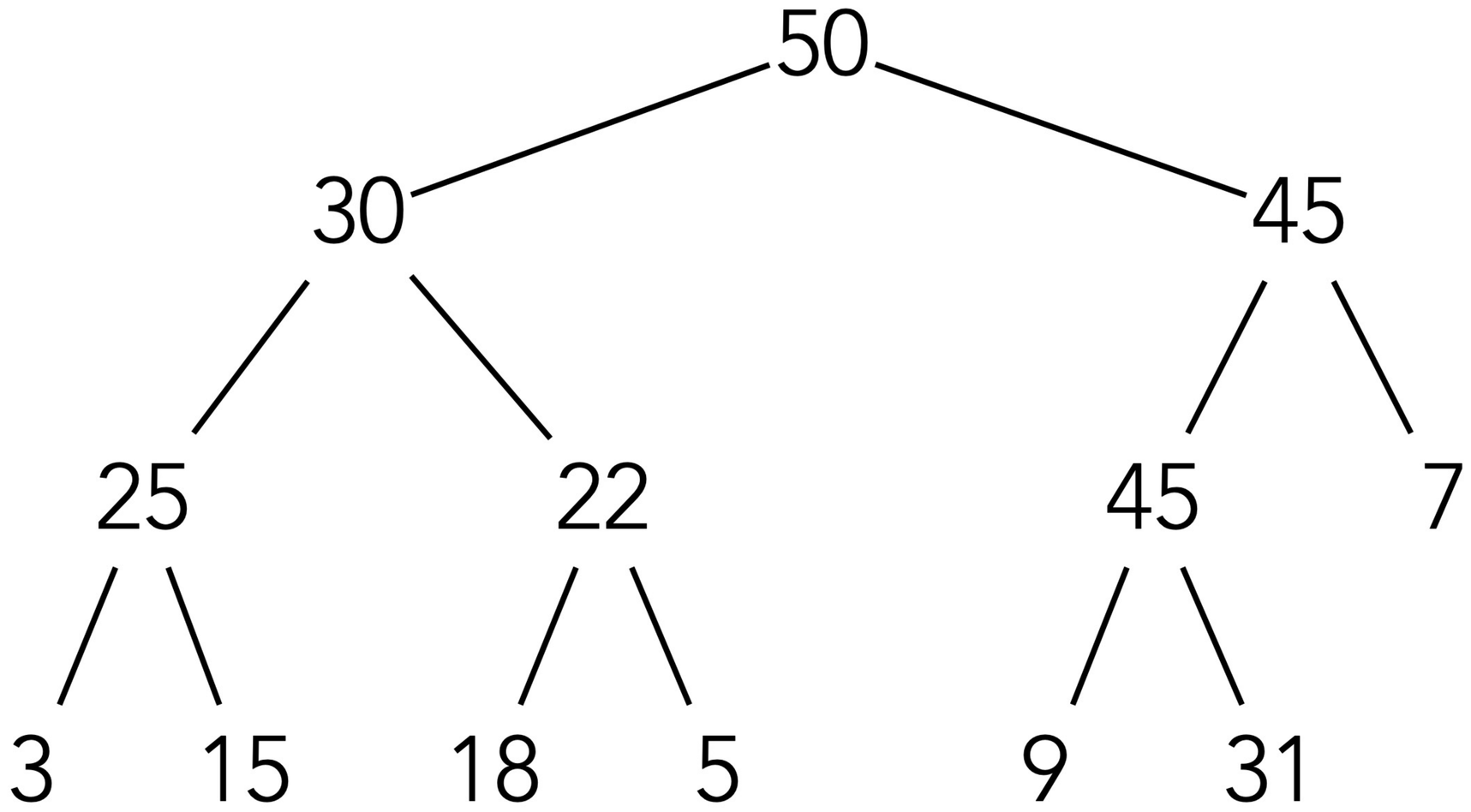
$$\boxed{\log n \geq h}$$



Example

Structure Property?

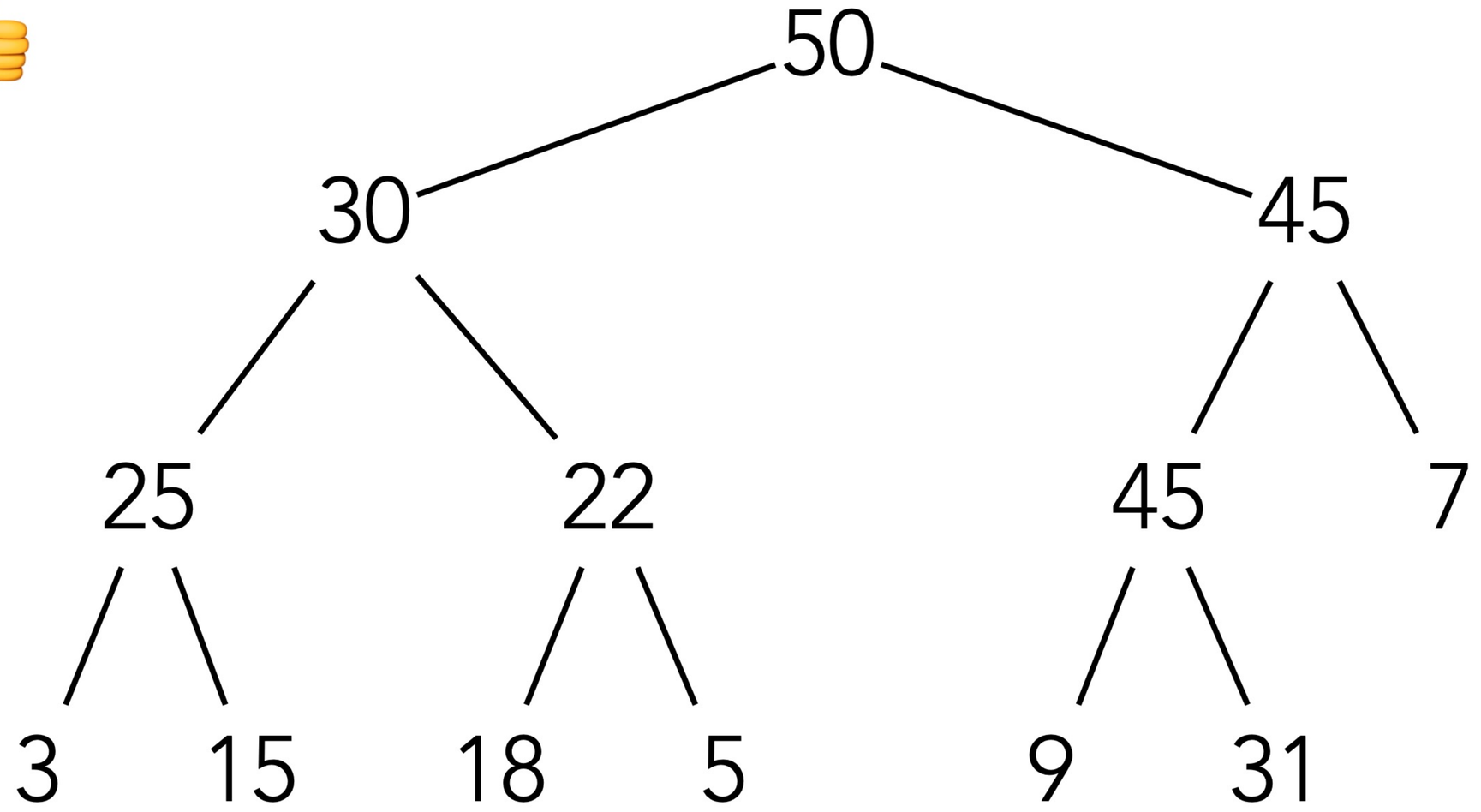
Heap-order Property?



Example

Structure Property? 👍

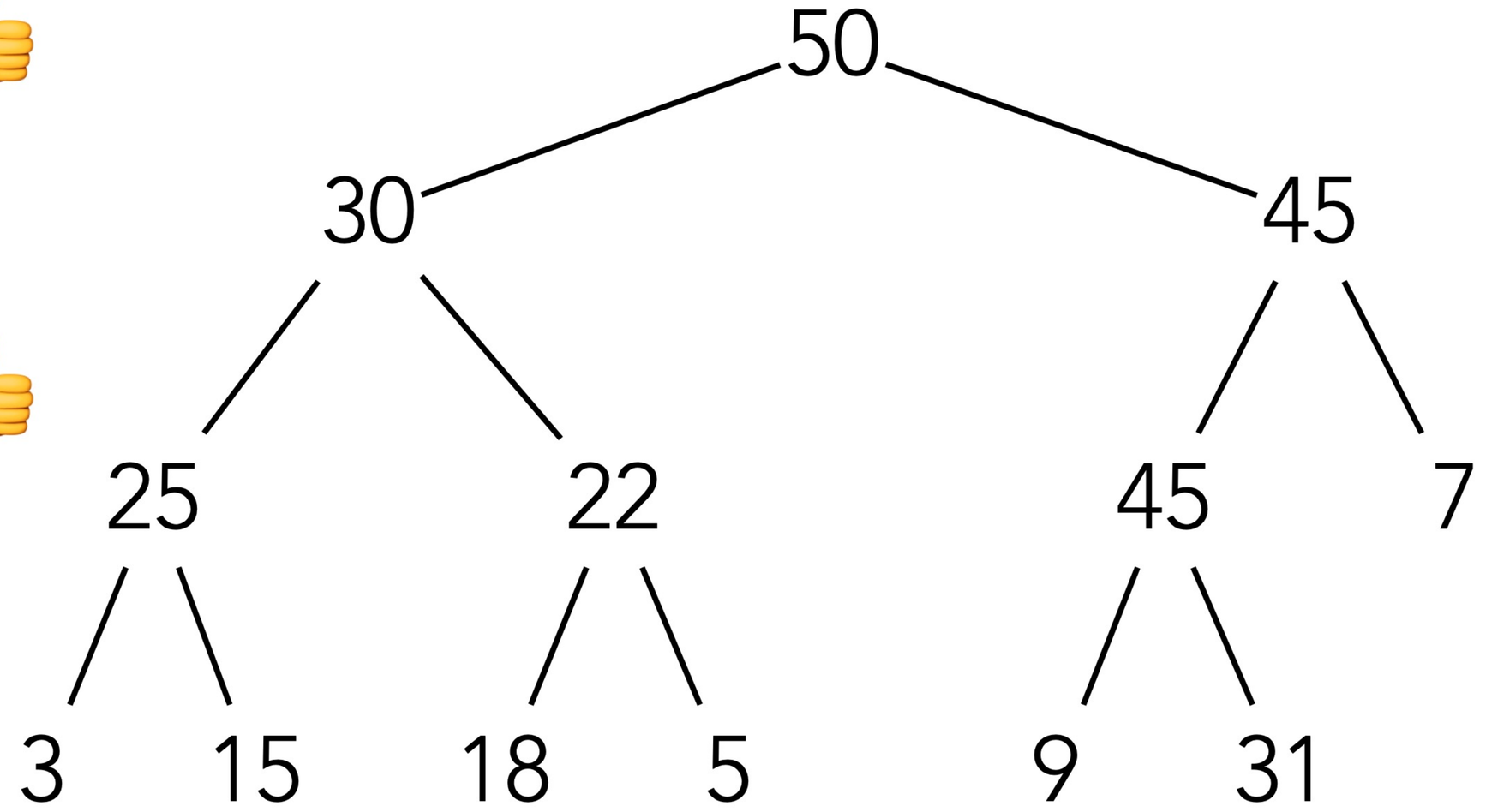
Heap-order Property?



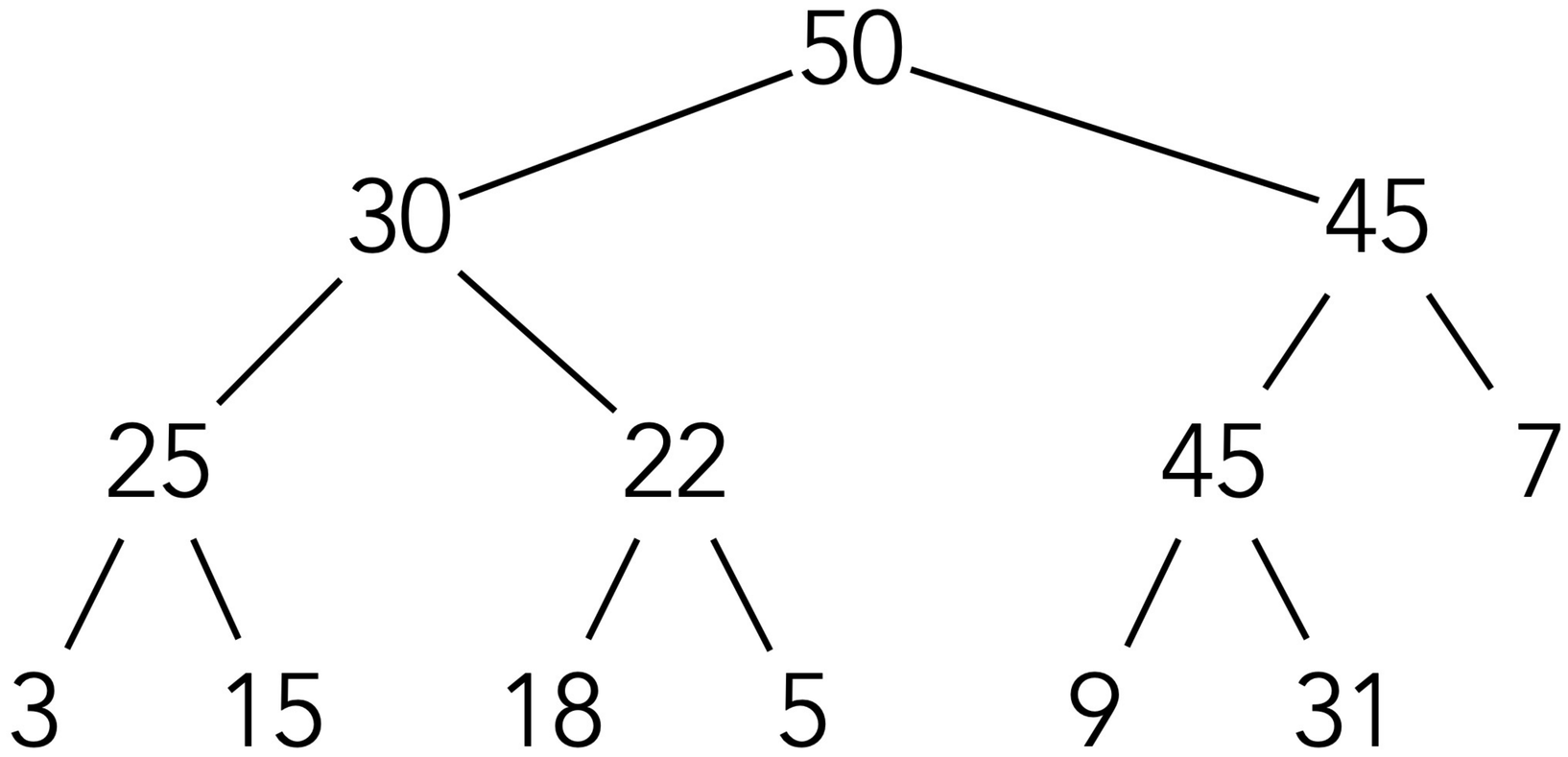
Example

Structure Property? 👍

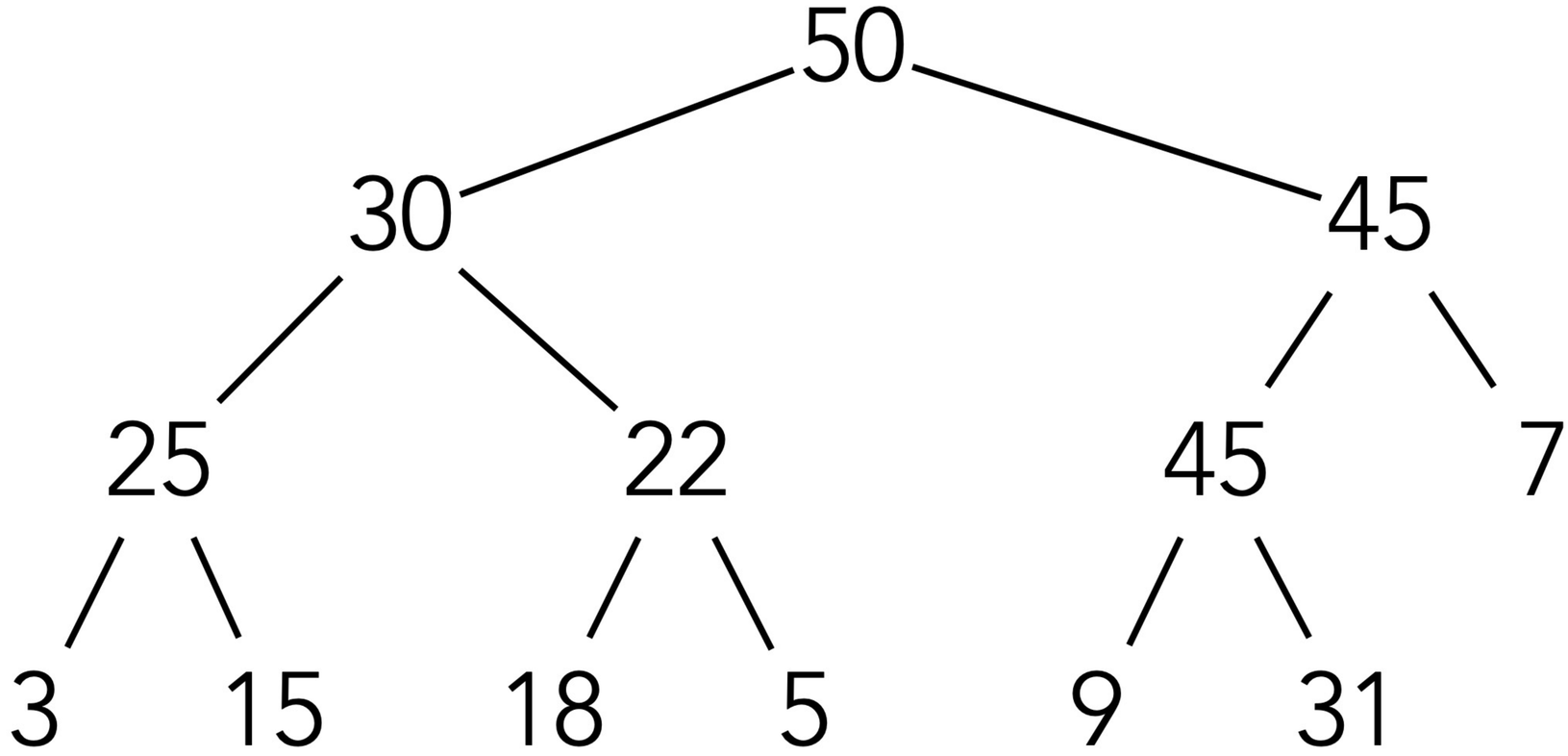
Heap-order Property? 👍



Implementation

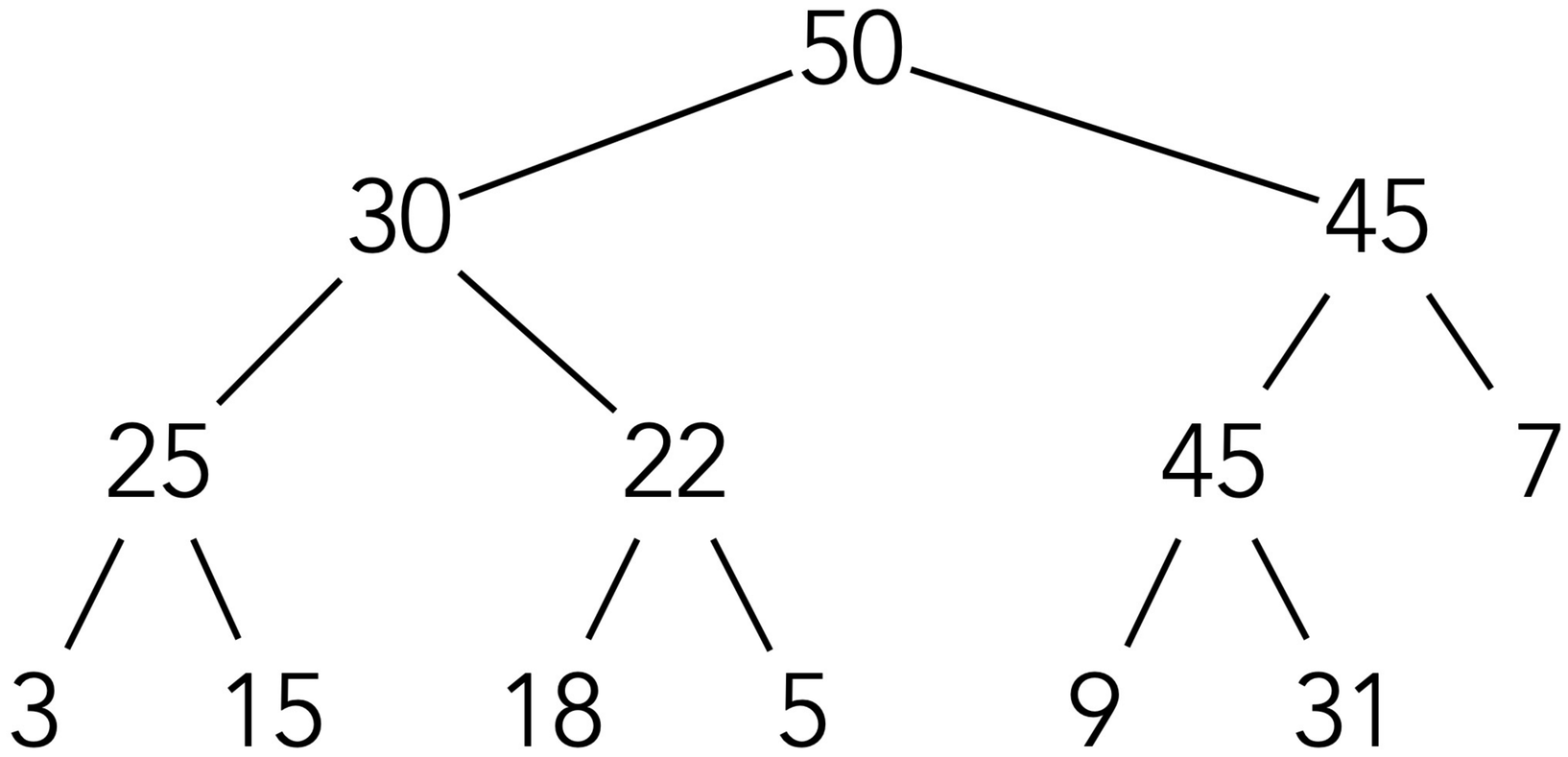


Implementation



Complete tree …

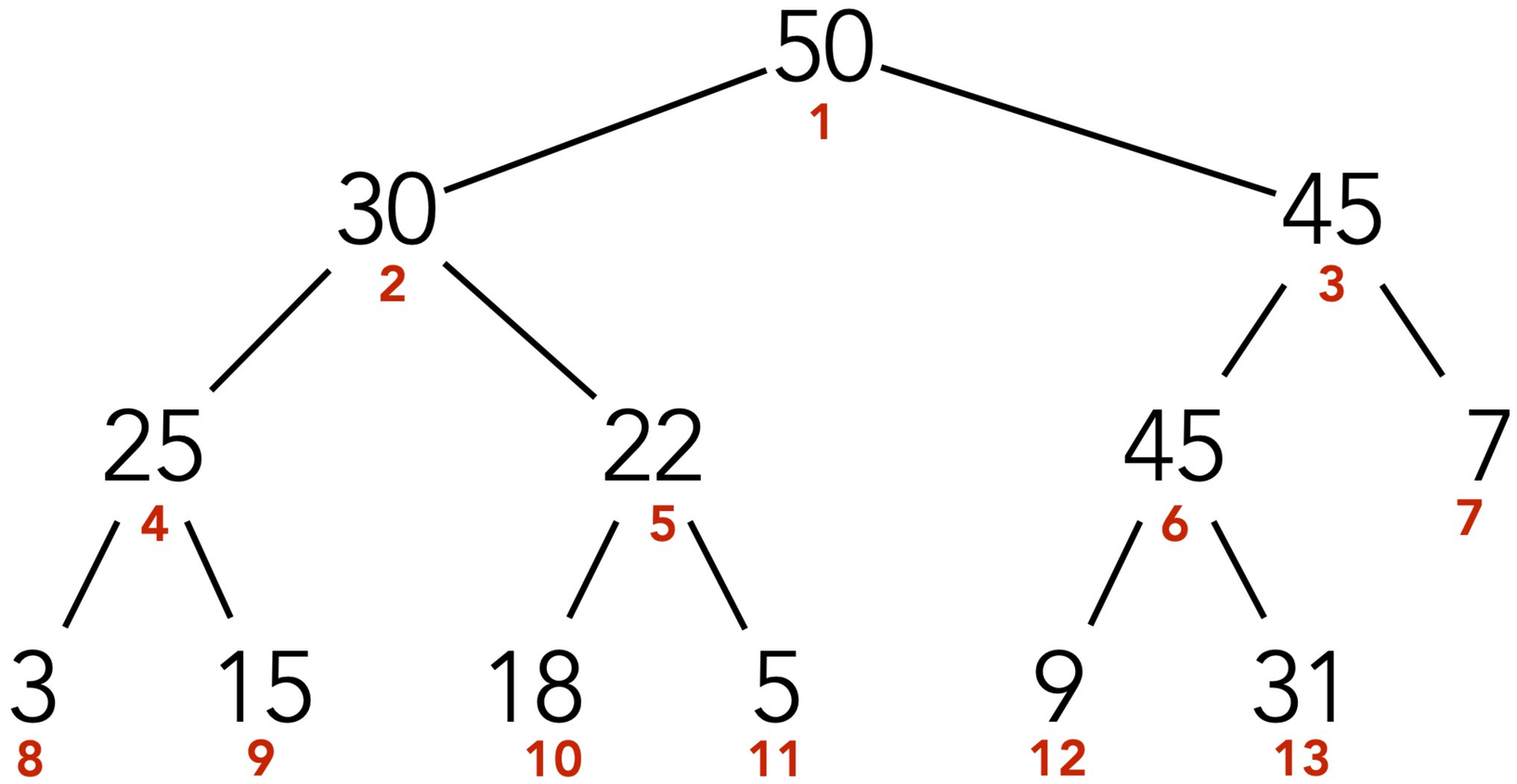
Implementation



Complete tree …

50	30	45	25	22	45	7	3	15	18	5	9	31
1	2	3	4	5	6	7	8	9	10	11	12	13

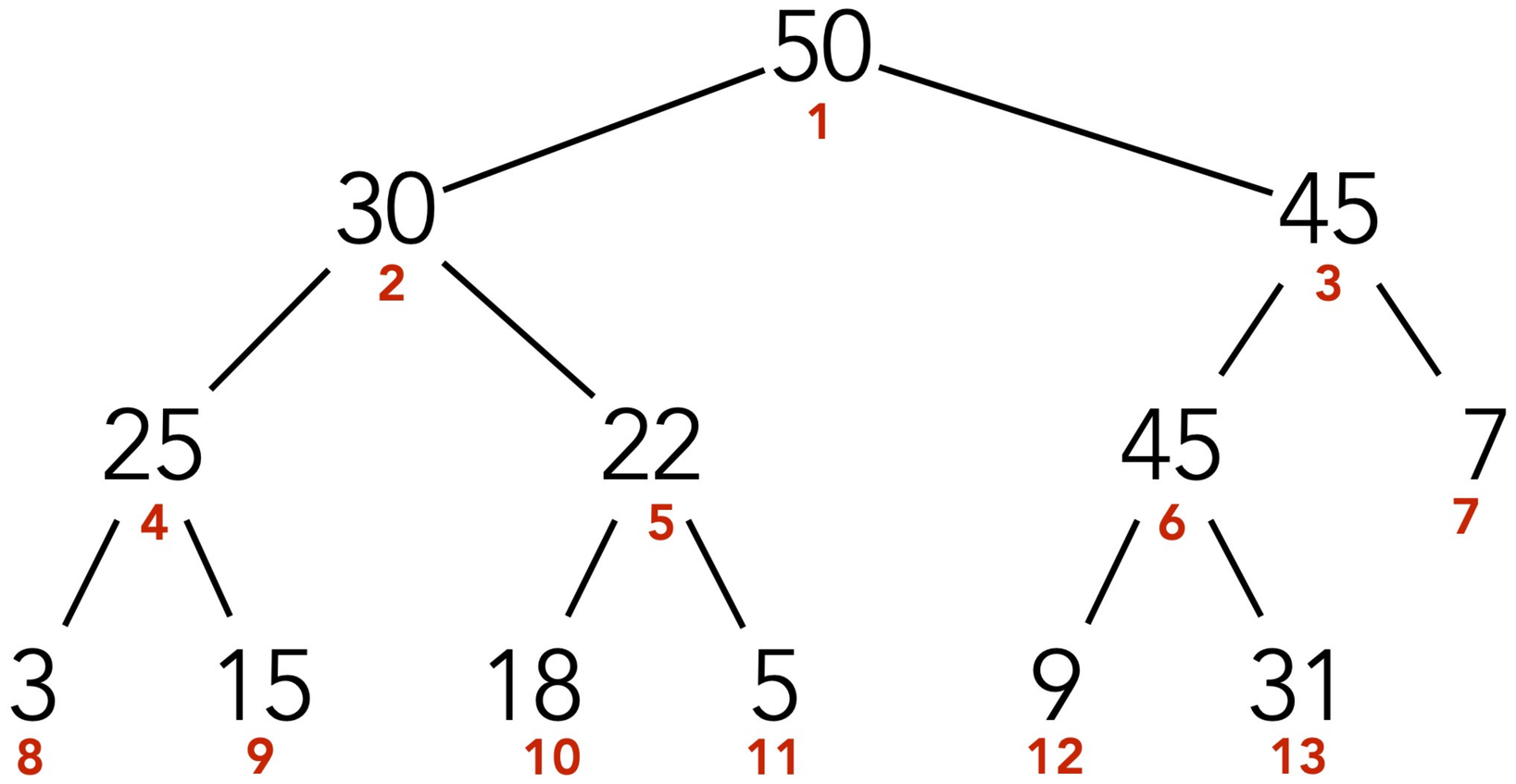
Implementation



Complete tree …

50	30	45	25	22	45	7	3	15	18	5	9	31
1	2	3	4	5	6	7	8	9	10	11	12	13

Implementation



parent(i)

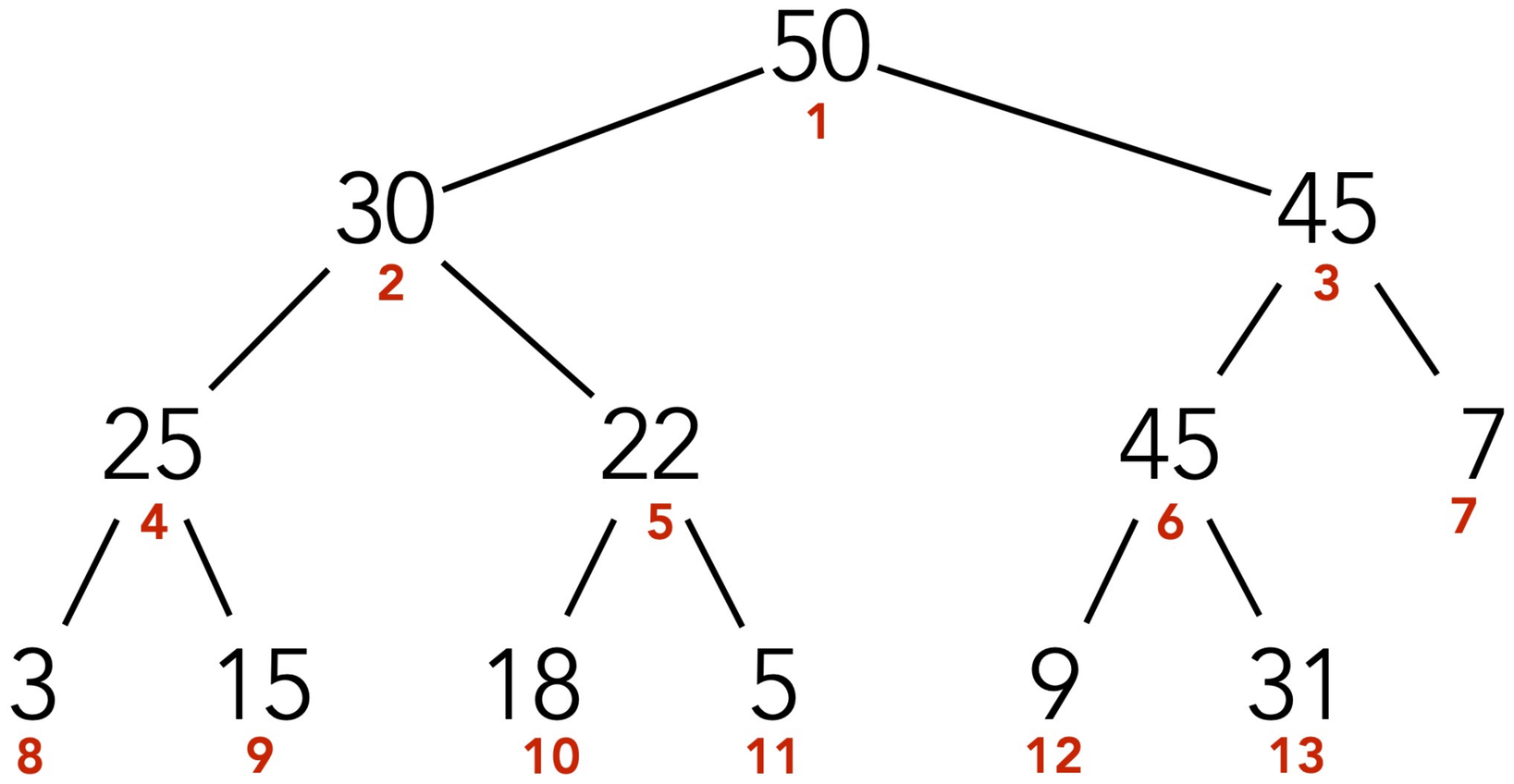
left_child(i)

right_child(i)

Complete tree …

50	30	45	25	22	45	7	3	15	18	5	9	31
1	2	3	4	5	6	7	8	9	10	11	12	13

Implementation



parent(i)

`floor(i/2)`

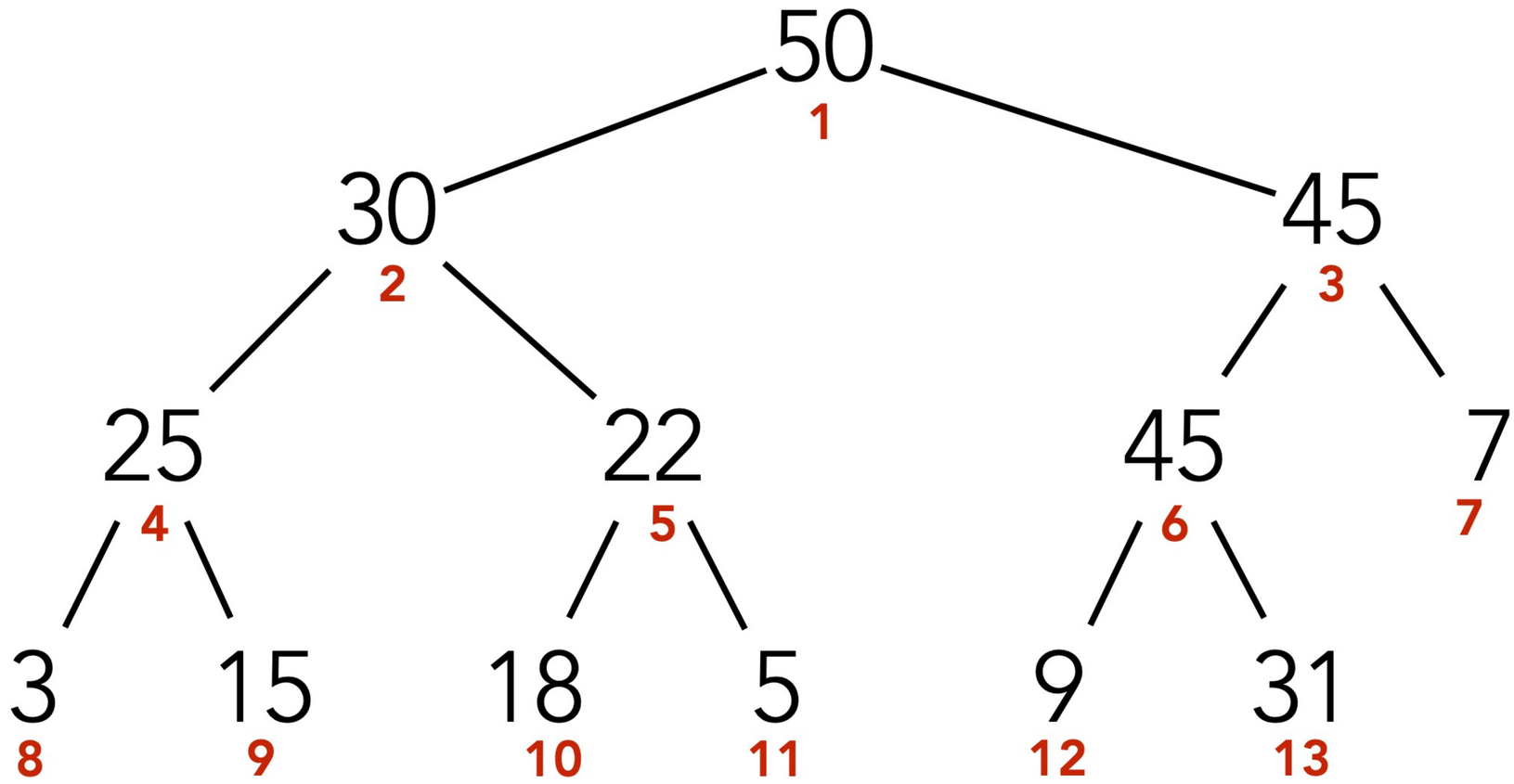
left_child(i)

right_child(i)

Complete tree …

50	30	45	25	22	45	7	3	15	18	5	9	31
1	2	3	4	5	6	7	8	9	10	11	12	13

Implementation



parent(i)

`floor(i/2)`

left_child(i)

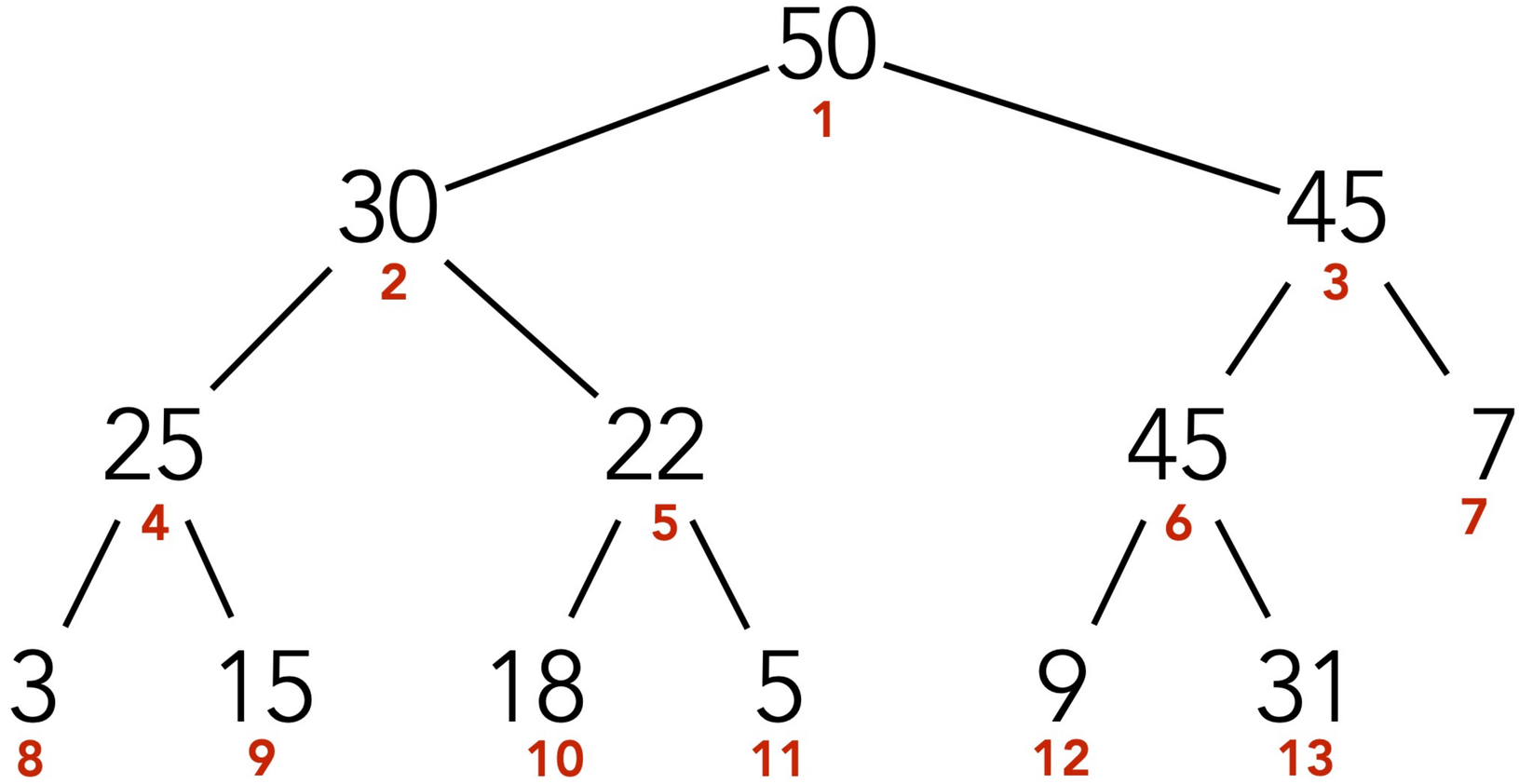
`i*2`

right_child(i)

Complete tree …

50	30	45	25	22	45	7	3	15	18	5	9	31
1	2	3	4	5	6	7	8	9	10	11	12	13

Implementation



parent(i)

`floor(i/2)`

left_child(i)

`i*2`

right_child(i)

`i*2 + 1`

Complete tree …

50	30	45	25	22	45	7	3	15	18	5	9	31
1	2	3	4	5	6	7	8	9	10	11	12	13

insert

Insert

Insert

Append new element to the end of array

Insert

Append new element to the end of array

Check heap-order property

Insert

Append new element to the end of array

Check heap-order property

- if violated, **Up-Heap** (swap with parent)

Insert

Append new element to the end of array

Check heap-order property

- if violated, **Up-Heap** (swap with parent)
  - **repeat** until heap-order is restored

Insert

Append new element to the end of array

Check heap-order property

- if violated, **Up-Heap** (swap with parent)
  - **repeat** until heap-order is restored
- if not, we are done

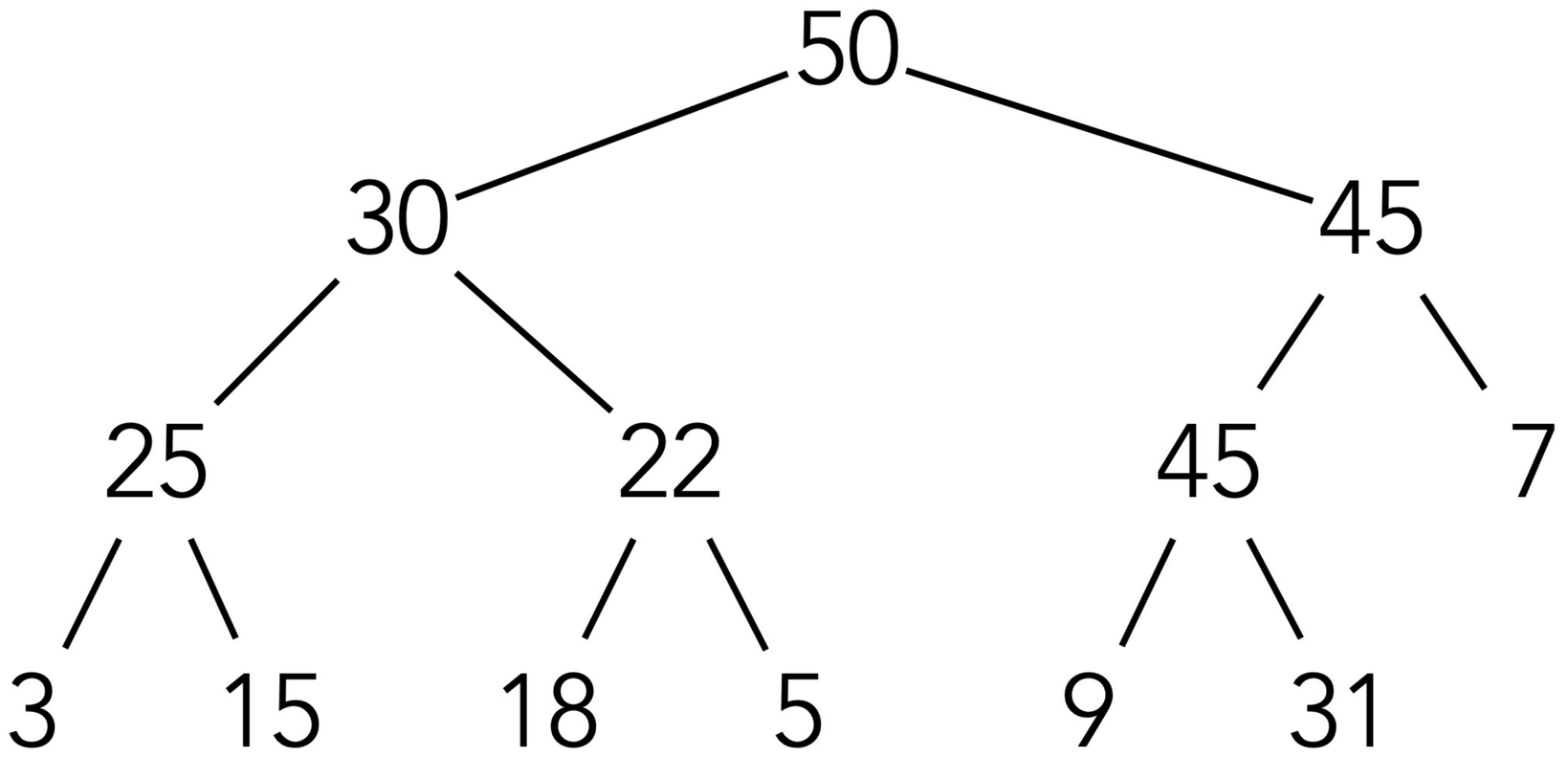
Insert

Append new element to the end of array

Check heap-order property

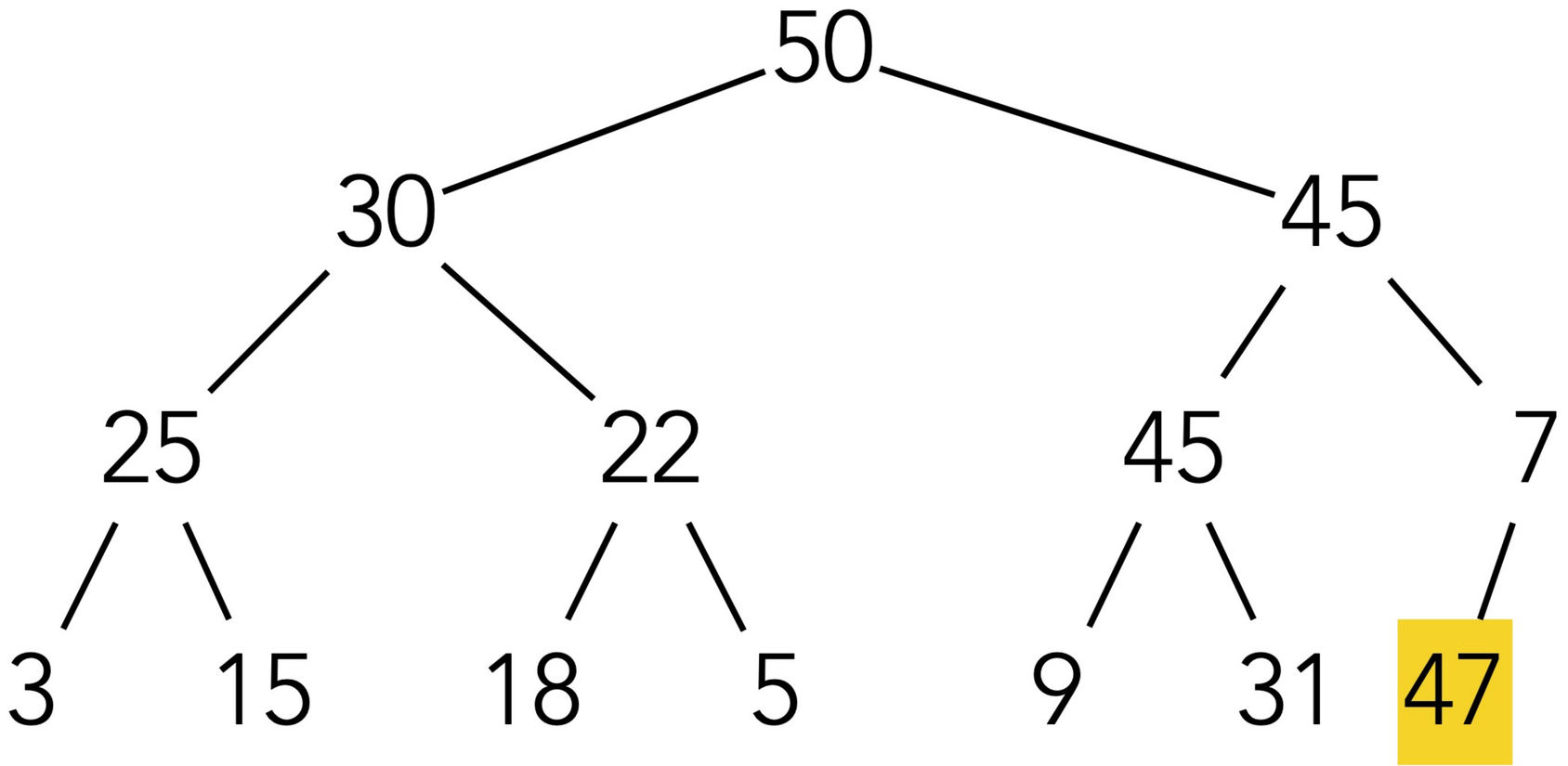
- if violated, **Up-Heap** (swap with parent)
  - **repeat** until heap-order is restored
- if not, we are done

O(log n)



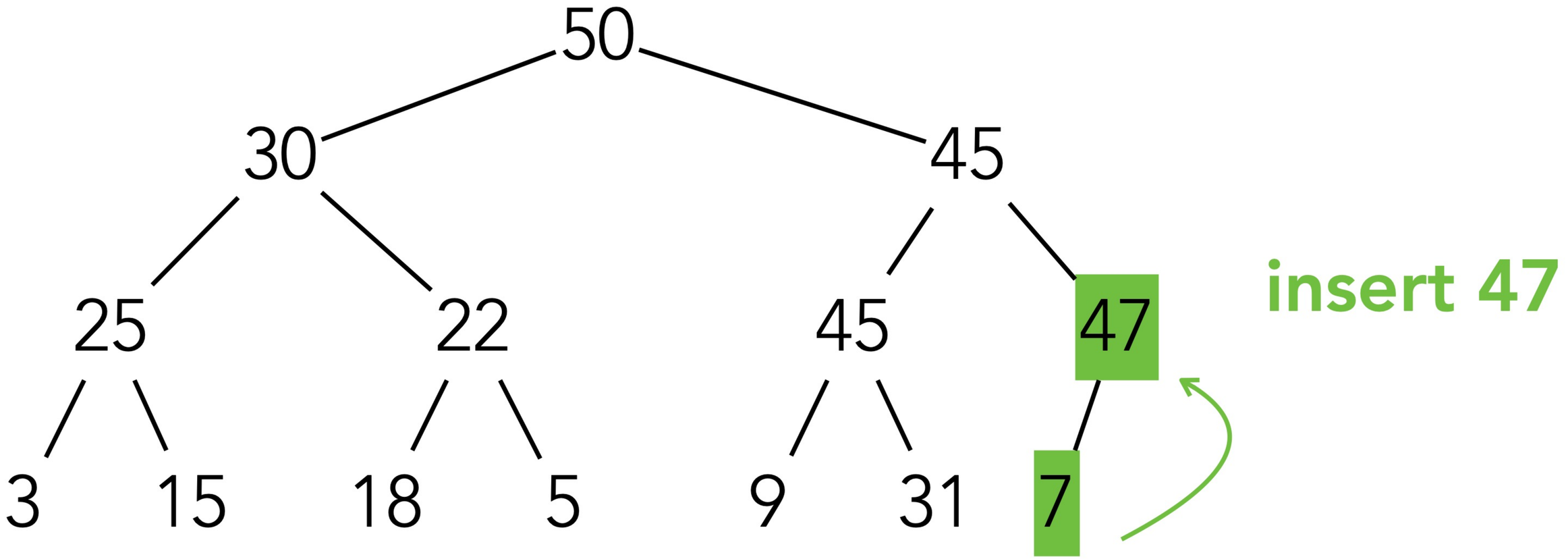
insert 47

50	30	45	25	22	45	7	3	15	18	5	9	31			

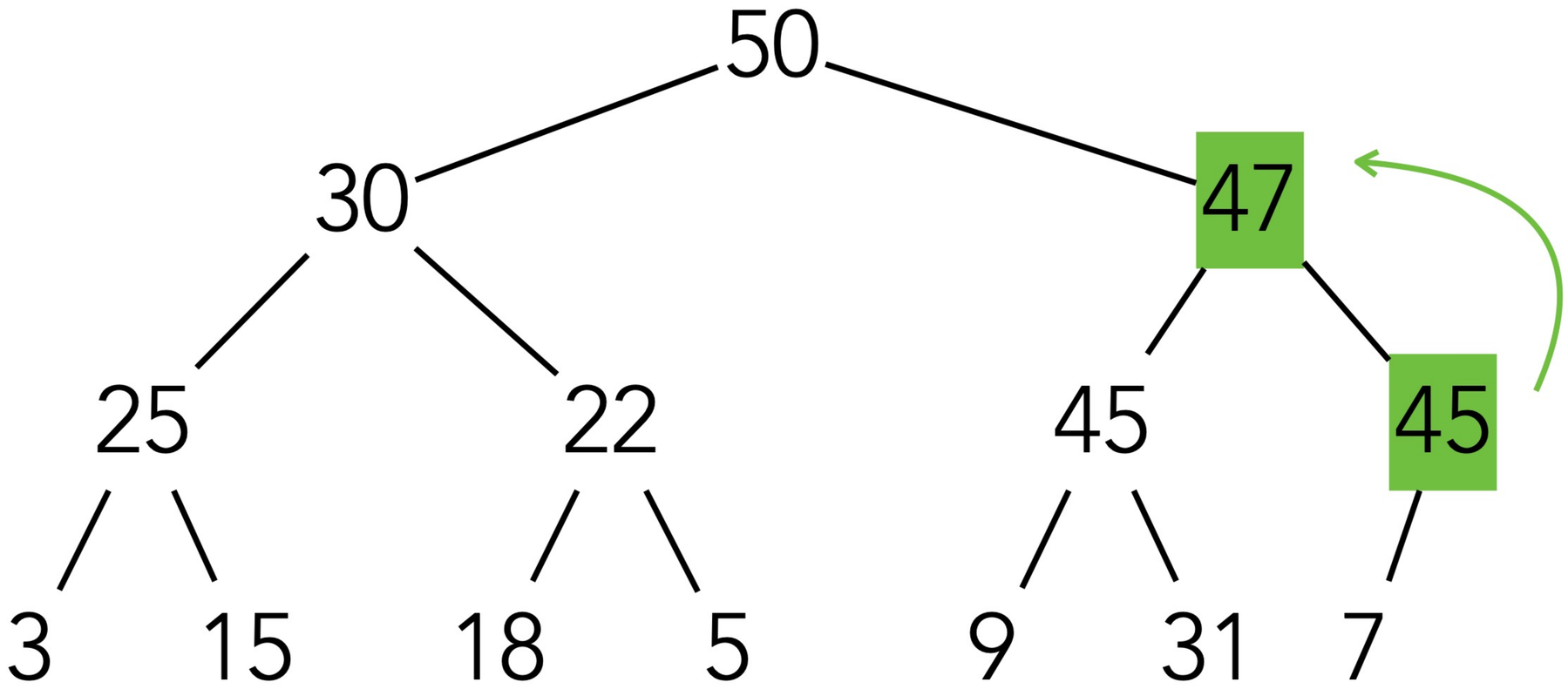


insert 47

50	30	45	25	22	45	7	3	15	18	5	9	31	47		

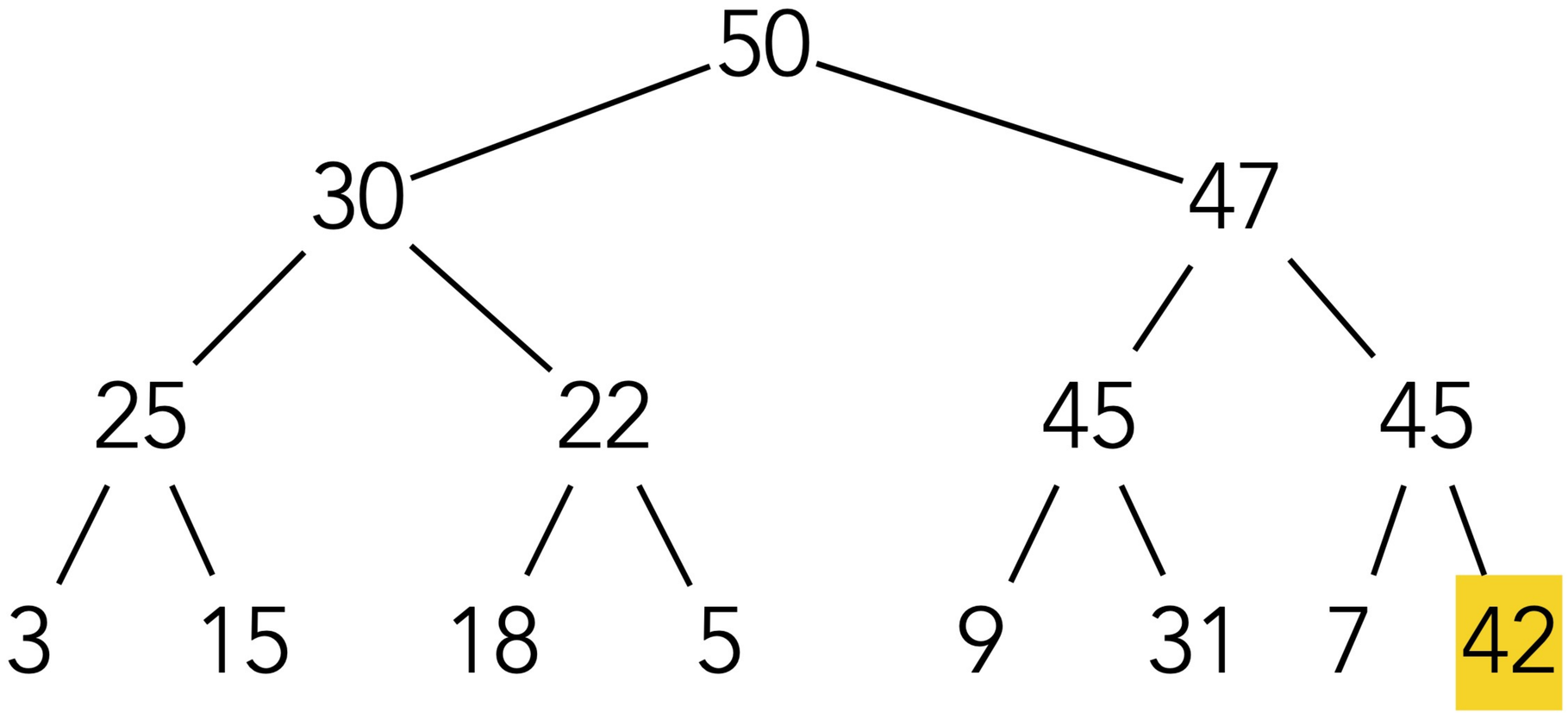


50	30	45	25	22	45	47	3	15	18	5	9	31	7		



insert 47

50	30	47	25	22	45	45	3	15	18	5	9	31	7		



insert 42

50	30	47	25	22	45	45	3	15	18	5	9	31	7	42	

removeMax

removeMax

removeMax

Max element is the the **first** element of the array

removeMax

Max element is the the **first** element of the array

the root of the heap

removeMax

Max element is the the **first** element of the array

the root of the heap

Copy last element of array to first position

removeMax

Max element is the the **first** element of the array

the root of the heap

Copy last element of array to first position

then decrement array size by 1 (removes last element)

removeMax

Max element is the the **first** element of the array

- the root of the heap

Copy last element of array to first position

- then decrement array size by 1 (removes last element)

Check heap-order property

removeMax

Max element is the the **first** element of the array

- the root of the heap

Copy last element of array to first position

- then decrement array size by 1 (removes last element)

Check heap-order property

- if violated, **Down-Heap** (swap with **larger** child)

removeMax

Max element is the the **first** element of the array

- the root of the heap

Copy last element of array to first position

- then decrement array size by 1 (removes last element)

Check heap-order property

- if violated, **Down-Heap** (swap with **larger** child)
  - **repeat** until heap-order is restored

removeMax

Max element is the the **first** element of the array

- the root of the heap

Copy last element of array to first position

- then decrement array size by 1 (removes last element)

Check heap-order property

- if violated, **Down-Heap** (swap with **larger** child)
  - **repeat** until heap-order is restored
- if not, we are done

removeMax

Max element is the the **first** element of the array

- the root of the heap

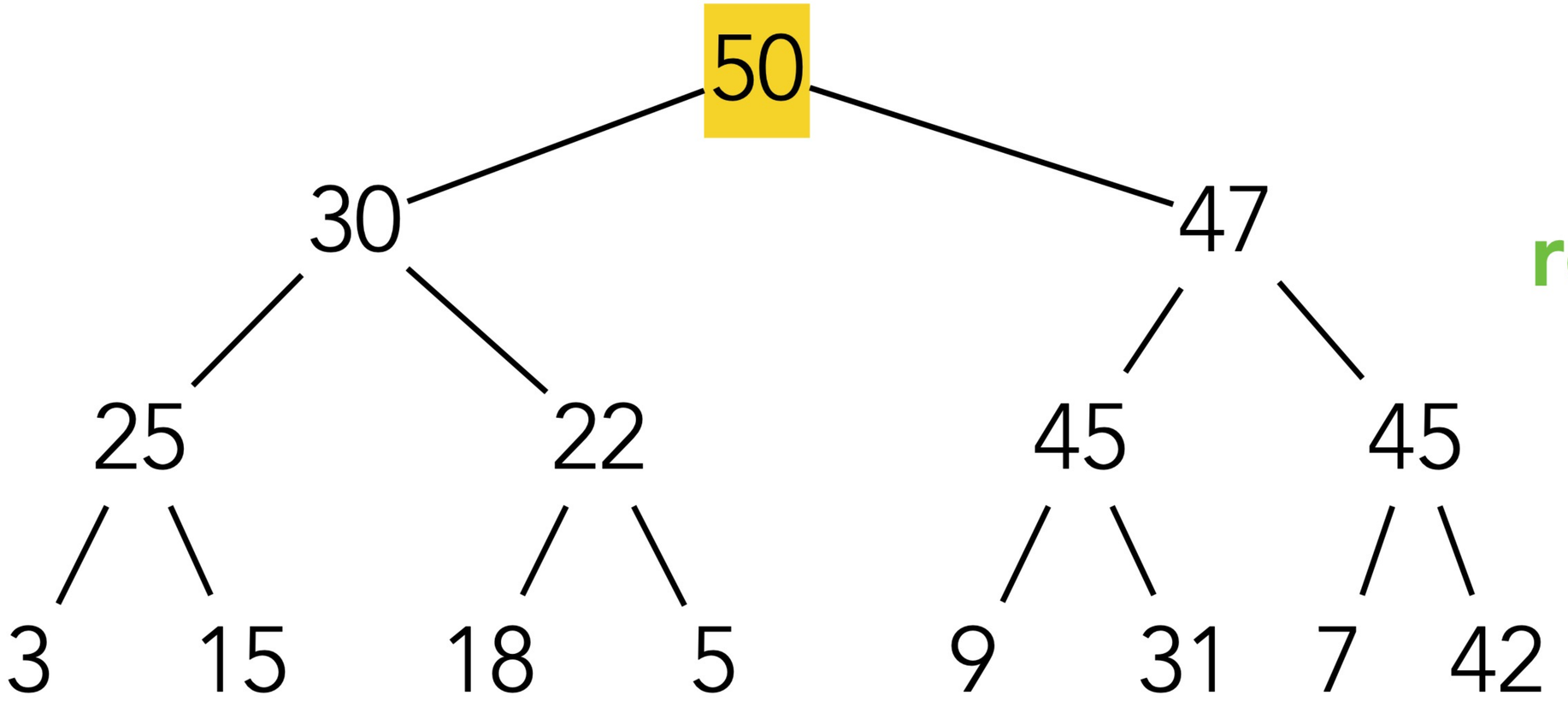
Copy last element of array to first position

- then decrement array size by 1 (removes last element)

Check heap-order property

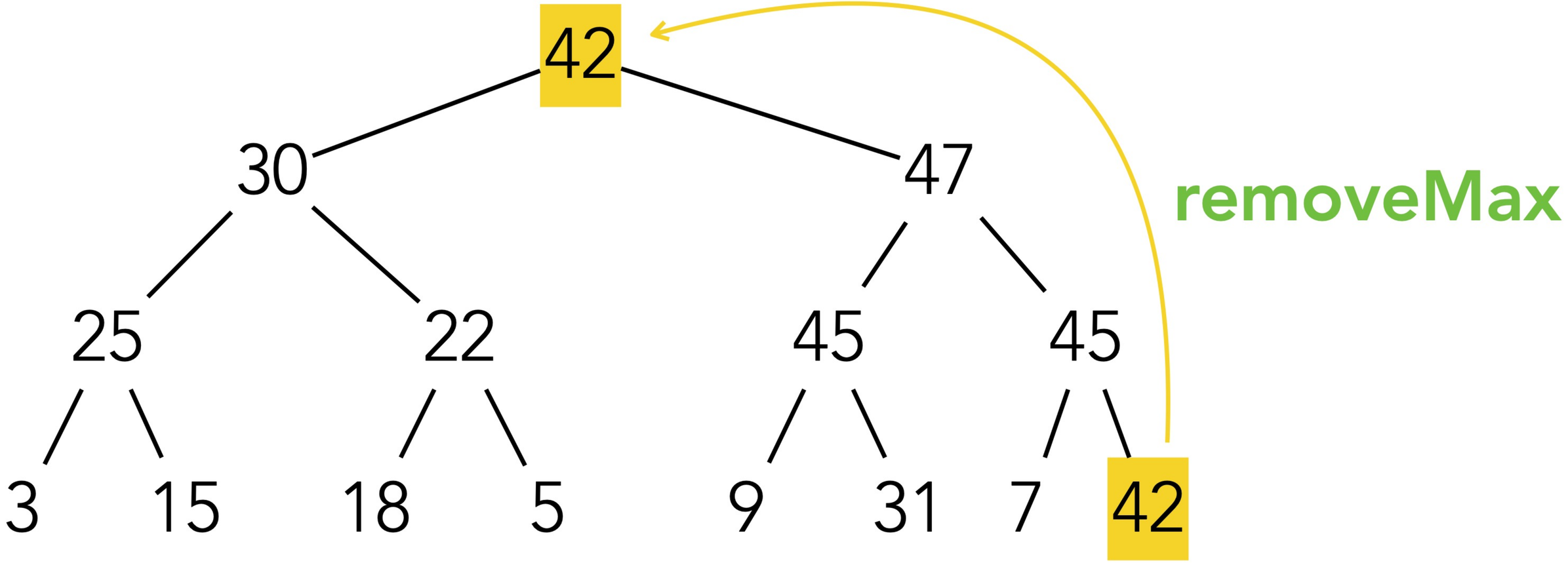
- if violated, **Down-Heap** (swap with **larger** child)
  - **repeat** until heap-order is restored
- if not, we are done

O(log n)

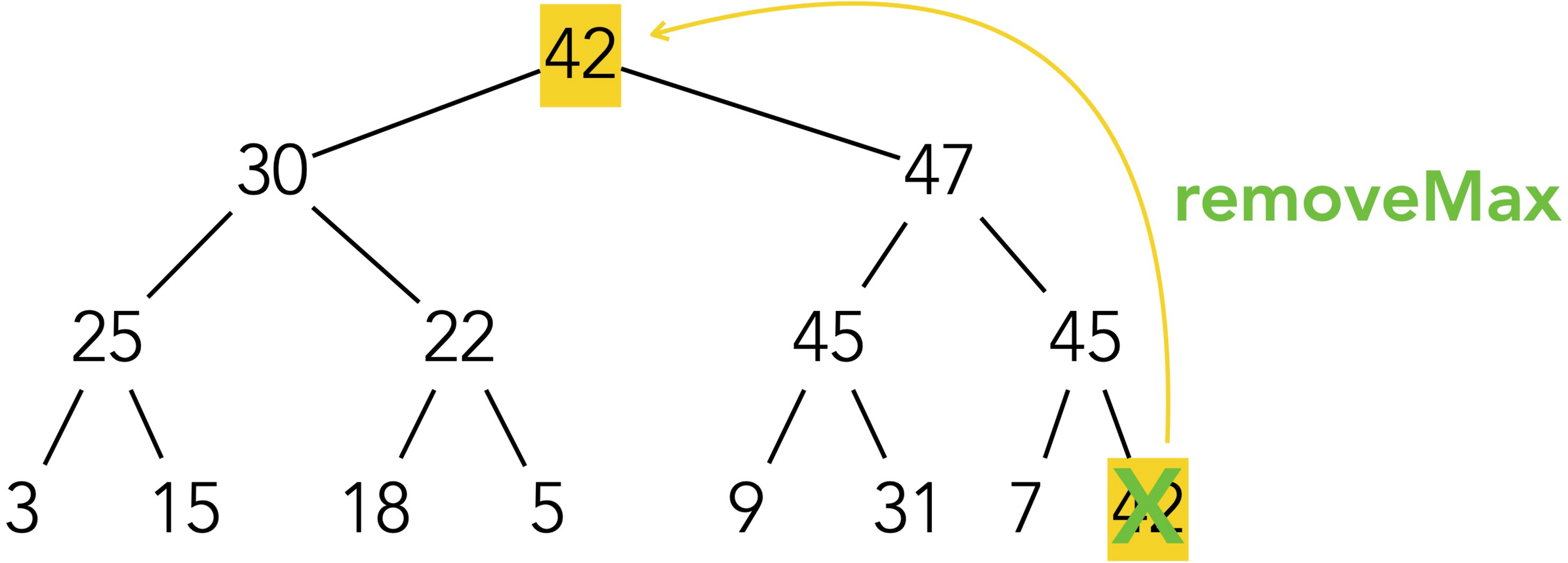


removeMax

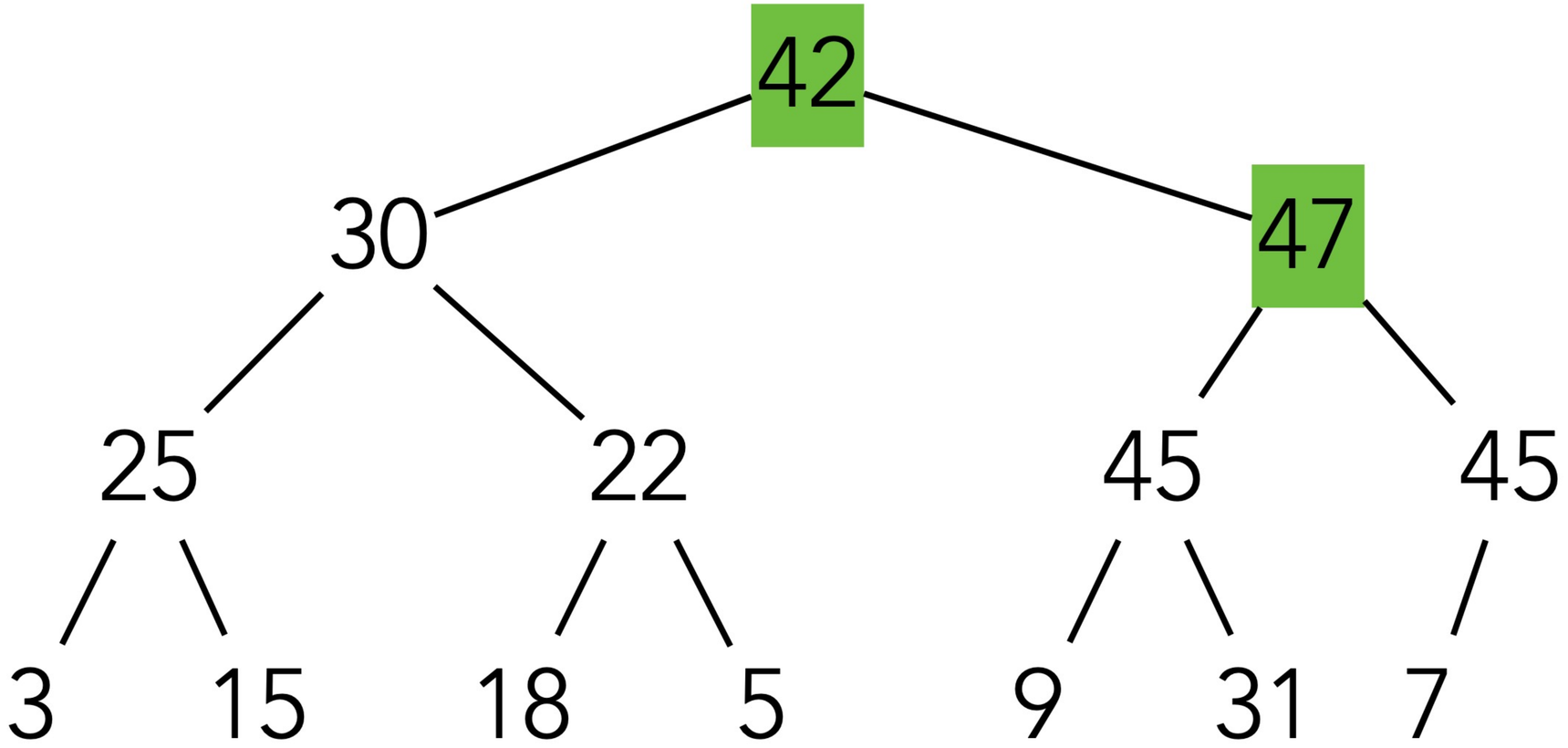
50	30	47	25	22	45	45	3	15	18	5	9	31	7	42	



42	30	47	25	22	45	45	3	15	18	5	9	31	7	42	

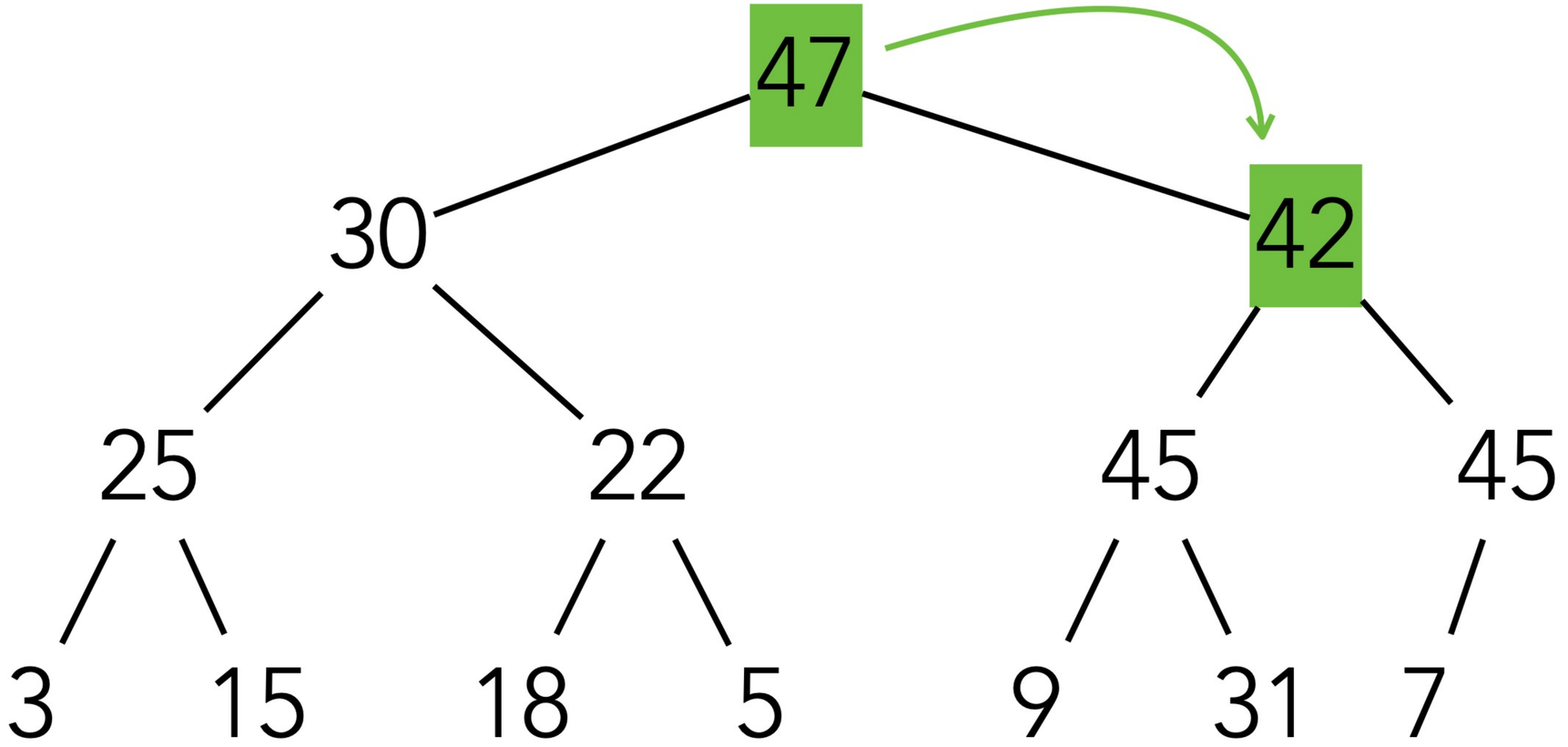


42	30	47	25	22	45	45	3	15	18	5	9	31	7	42	



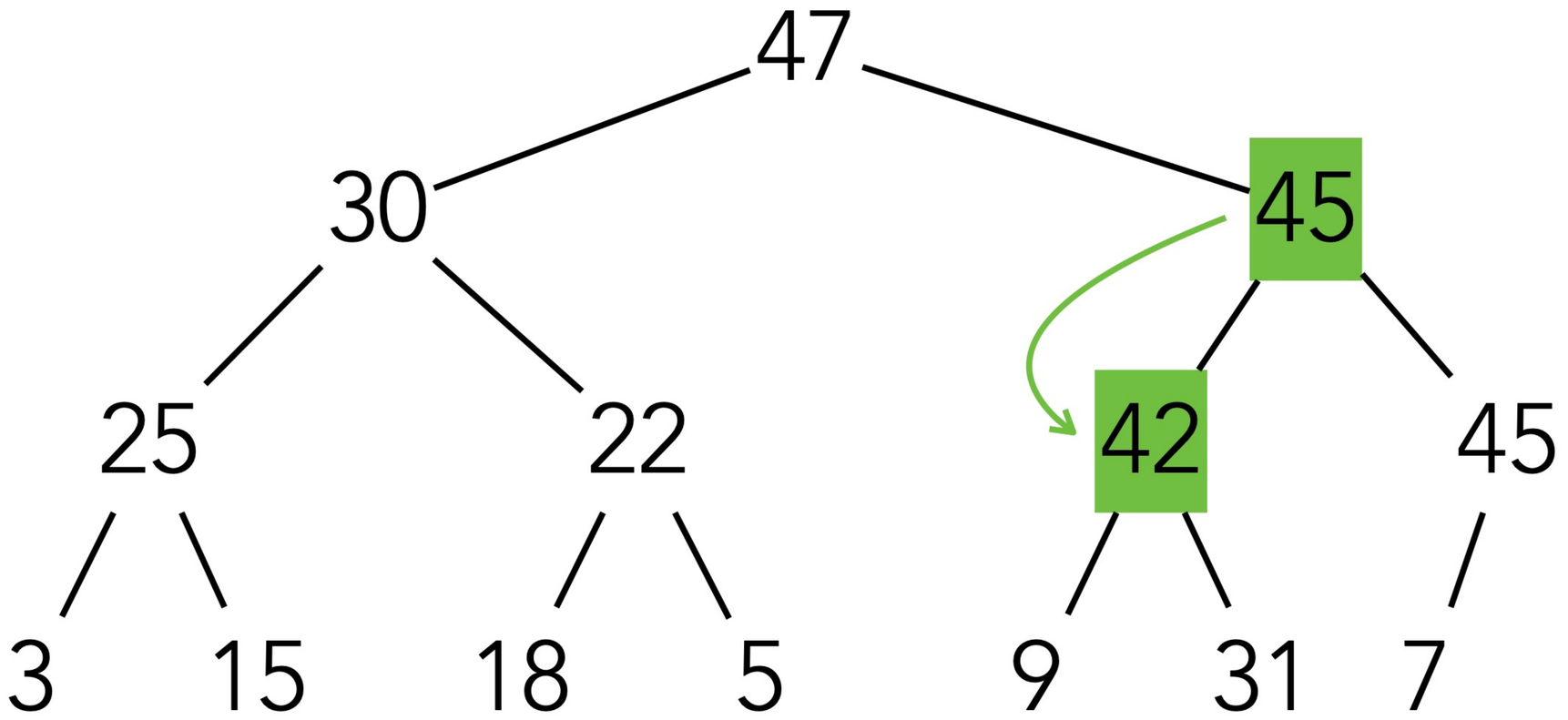
removeMax

42	30	47	25	22	45	45	3	15	18	5	9	31	7		



removeMax

47	30	42	25	22	45	45	3	15	18	5	9	31	7		



removeMax

47	30	45	25	22	42	45	3	15	18	5	9	31	7		

Performance

	Sorted Array/List	Unsorted Array/ List	Heap
insert	O(n)	O(1)	
removeMax	O(1)	O(n)	
max	O(1)	O(n)	
insert N	$O(n^2)$	O(n)	

Performance

	Sorted Array/List	Unsorted Array/ List	Heap
insert	O(n)	O(1)	O(log n)
removeMax	O(1)	O(n)	O(log n)
max	O(1)	O(n)	O(1)
insert N	$O(n^2)$	O(n)	O(n)**

(**) assuming we know the sequence in advance (**buildHeap**)