Heapsort and Binary Heaps

Heapsort

Efficient in-place sorting with a binary heap

Introduced by J. Williams (1964) 

In-place adaptation by R. Floyd (1964) 

Note: Heapsort not a stable sort (sorting does not preserve input order of identical keys)

Heapsort complexity (for n data items)

Average case: $\rightarrow O(n \log n)$ same as quicksort

Worst case: $\rightarrow O(n \log n)$ better than quicksort

Storage: $\Theta(n)$ for items, $O(1)$ for sort (in-place)

Example heapsort applications:

Linux kernel avoid quicksort $O(n^2)$ worst-case

Introsort hybrid quicksort heapsort insertion sort

References/Notes/Image credits:
J. Williams, "Algorithm 232 – Heapsort", Communications of the ACM (1964)
(J. Williams photo) https://ottawacitizen.remembering.ca/obituary/j-w-j-williams-1065926154
(adaptation of heapsort) R. Floyd, "Algorithm 245 – Treesort 3", Communications of the ACM (1964)
(R. Floyd photo) https://wiki.alquds.edu/?query=File:Robert_W._Floyd.jpg
(Linux kernel sort.c) https://github.com/torvalds/linux/blob/master/lib/sort.c (last accessed 27/12/2022)
https://en.wikipedia.org/wiki/Introsort

Max and Min Binary Heaps

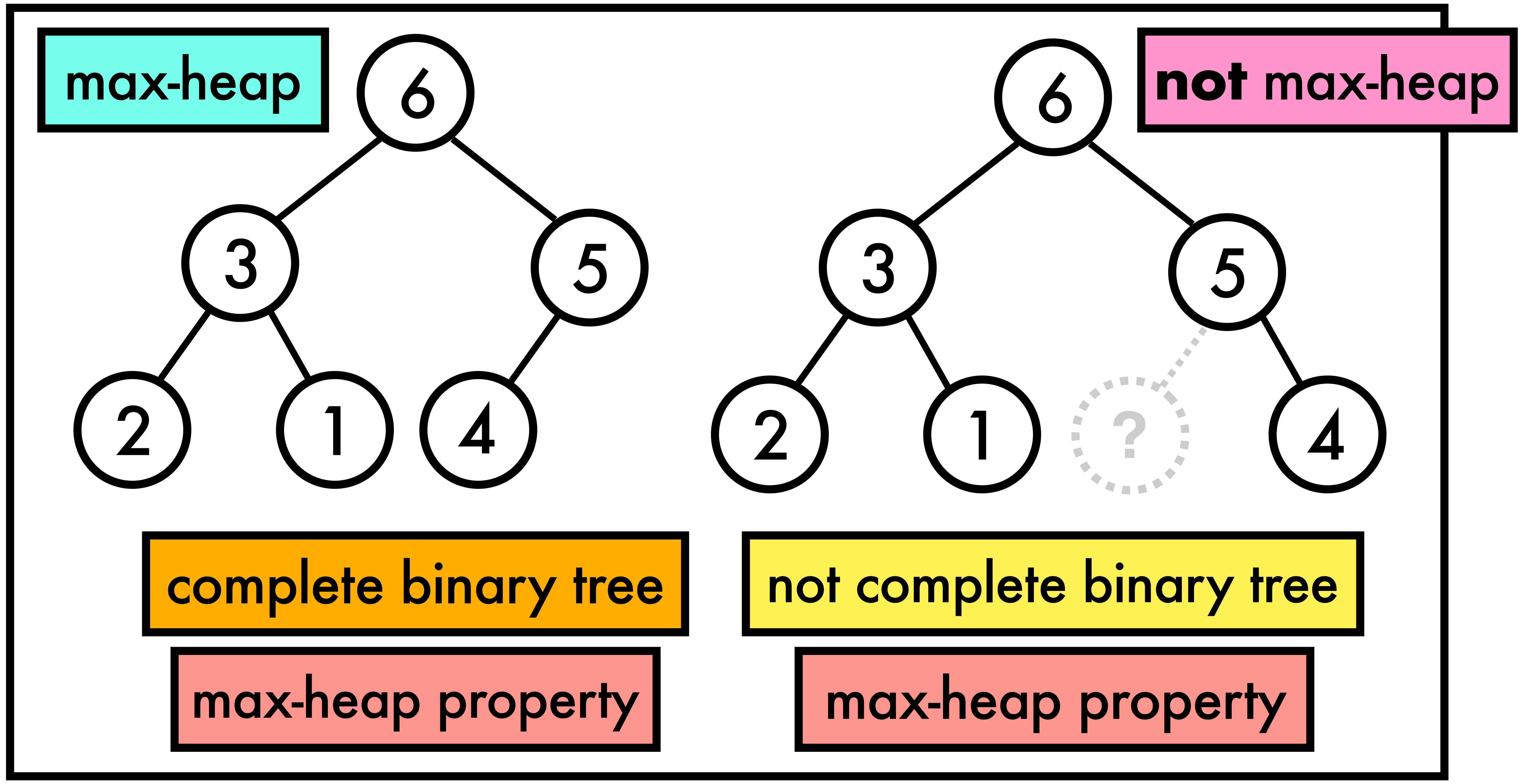
A (binary) max-heap is a binary tree satisfying:

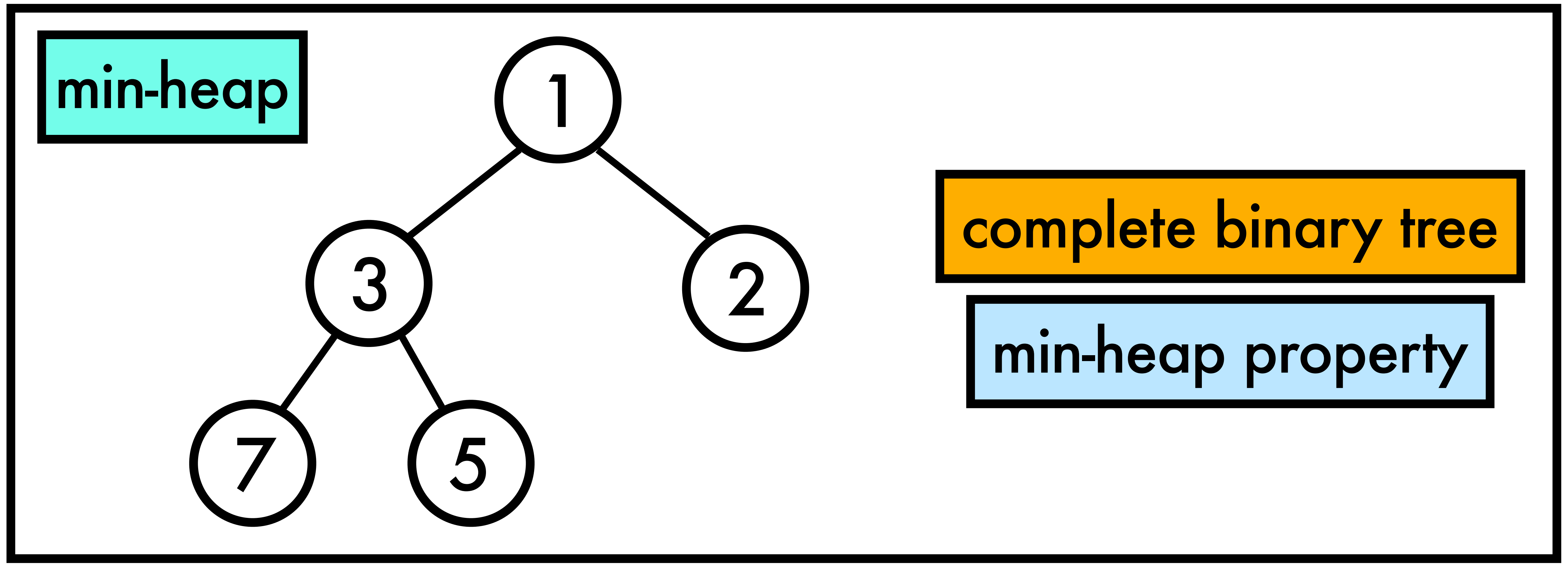
Shape property: the binary tree is complete - all levels except the last are full. If the last is not full, it is filled from left to right

Max-heap property: the key of each node is $\geq$ to the keys of its children

A min–heap is similar - it satisfies the same shape property but with a min-heap property:

Min-heap property: the key of each node is $\leq$ the keys of its children





References:
(Binary heap definition) https://en.wikipedia.org/wiki/Binary_heap

Note: we often drop the "binary" in "binary heap" (but d-ary heaps are possible)

Representing Binary Heaps with Arrays

A heap of `heap_size` nodes can be represented as an **array**, A of length n with $n \geq$ `heap_size`

We'll focus on the **max-heap** (used in **heapsort**)

Parent-child relations represented with **indices**

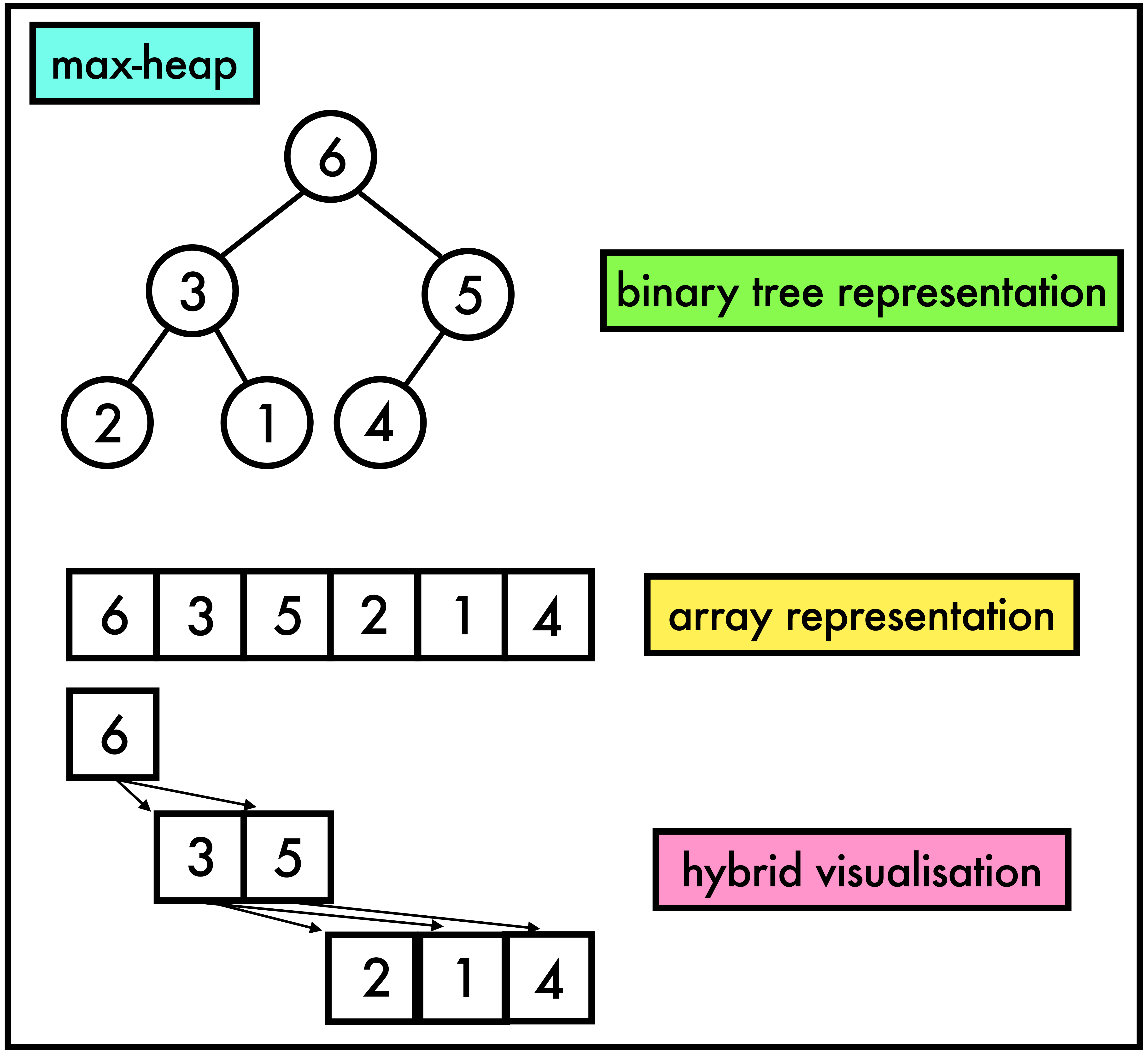
The heap **root** corresponds to array index 0

For node with index i:

- the **parent** has index $\lfloor (i-1)/2 \rfloor$
- the **left child** has index $2i+1$
- the **right child** has index $2i+2$

Max-heap property:

$A[i] \leq A[\text{parent}(i)]$ for $1 \leq i <$ `heap_size`



References:
(0-indexed binary heap) https://en.wikipedia.org/wiki/Binary_heap
(array representation) T. Cormen et al., "Introduction to algorithms", Chap 6.1, MIT press (2022)

Binary Heap Height

Binary heap height is $\Theta(\log n)$

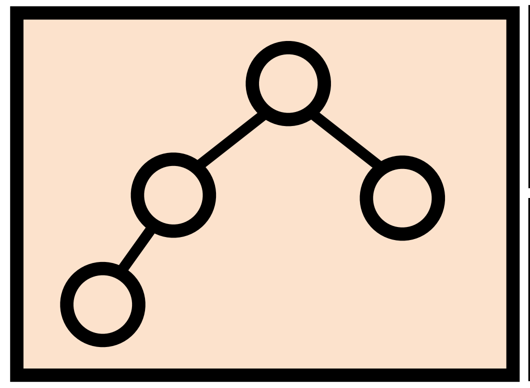
The height, h, of a heap with n keys is $h = \lfloor \log n \rfloor$ i.e. $\Theta(\log n)$

Proof:

Due to the shape property, the heap is a complete binary tree

A heap of height h has at least:

$1 + 2 + 4 + \ldots + 2^{h-1} + 1 = 2^h - 1 + 1 \ = 2^h$ nodes

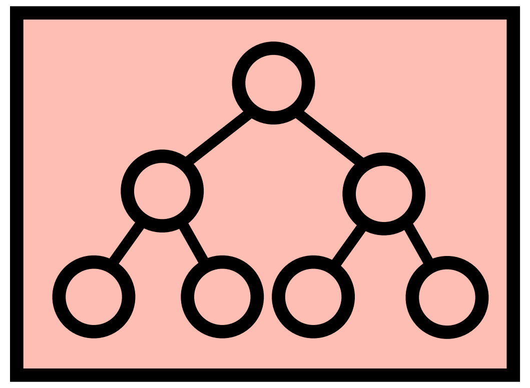


at least 2^h nodes

one node at lowest level

A heap of height h has at most:

$2^{h+1} - 1$ nodes



at most $2^{h+1} - 1$ nodes

lowest level is full

$\implies \ 2^h \leq n \leq 2^{h+1} - 1$

$\implies \ h \leq \log n$ and $\log(n+1) \leq h + 1$

Note: $\log n < \log(n+1)$

$x \leq y < x + 1, \ x \in \mathbb{Z}, \ y \in \mathbb{R}$

$\implies \ x = \lfloor y \rfloor$

$\implies \ h = \lfloor \log n \rfloor$

References:
M. T. Goodrich et al., "Algorithm design and applications", Chap. 5 (2015)

Fixing Max-Heap Violations

To maintain the max-heap property, we fix heap violations ("bubbling" violating keys down heap)

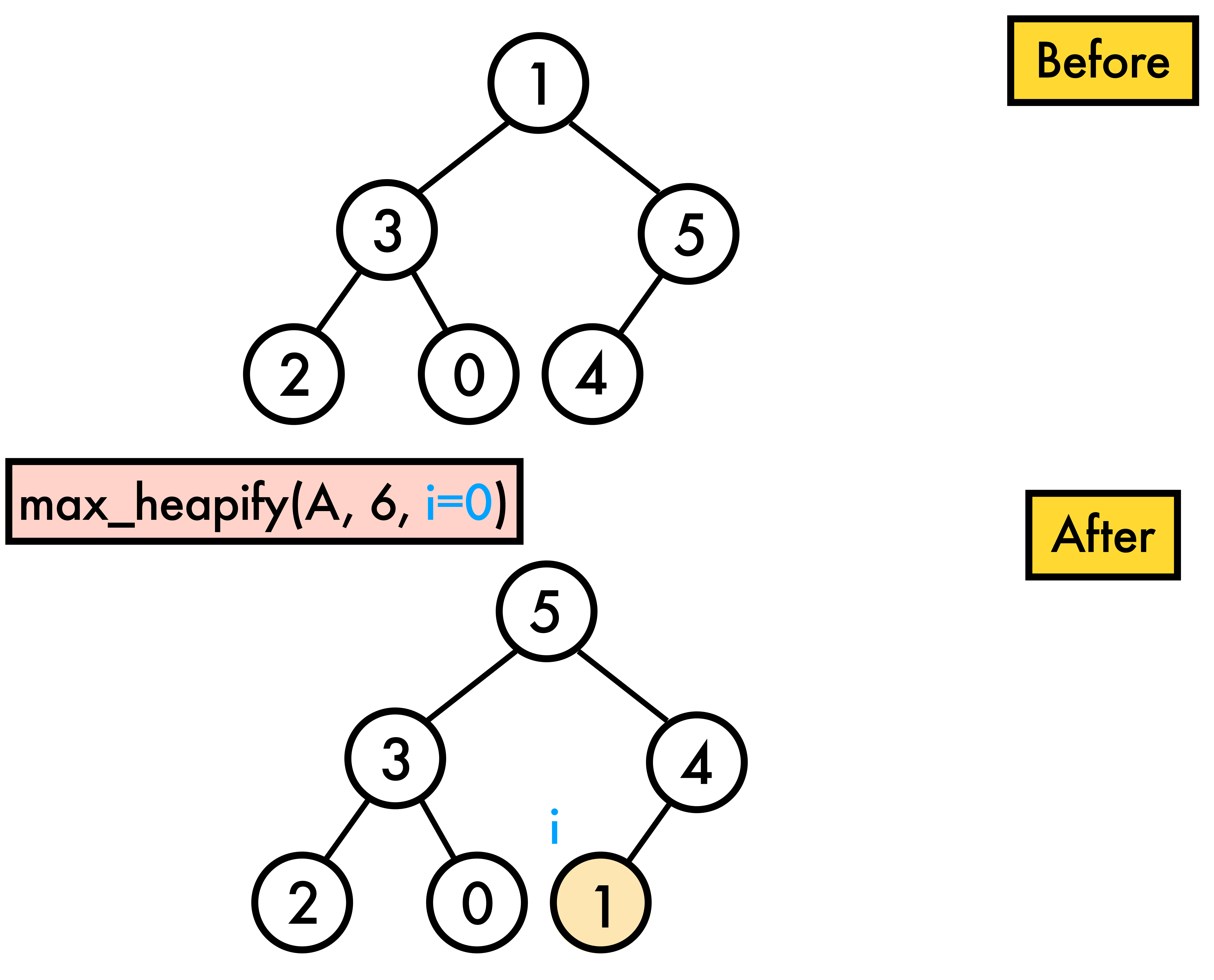
Achieved with `max_heapify()` helper function

Assumes left and right child are valid max-heaps

```
def max_heapify(A, heap_size, i):
    left = left_child(i)
    right = right_child(i)
    max_i = i
    if left < heap_size and A[left] > A[max_i]:
        max_i = left
    if right < heap_size and A[right] > A[max_i]:
        max_i = right
    if max_i != i:
        A[i], A[max_i] = A[max_i], A[i]
        max_heapify(A, heap_size, max_i)
```

```
def left_child(i):
    return 2 * i + 1
```

```
def right_child(i):
    return 2 * i + 2
```



Complexity $O(h) = O(\log n)$ (h is heap height)

References:
T. Cormen et al., "Introduction to algorithms", Chap 6.2, MIT press (2022)

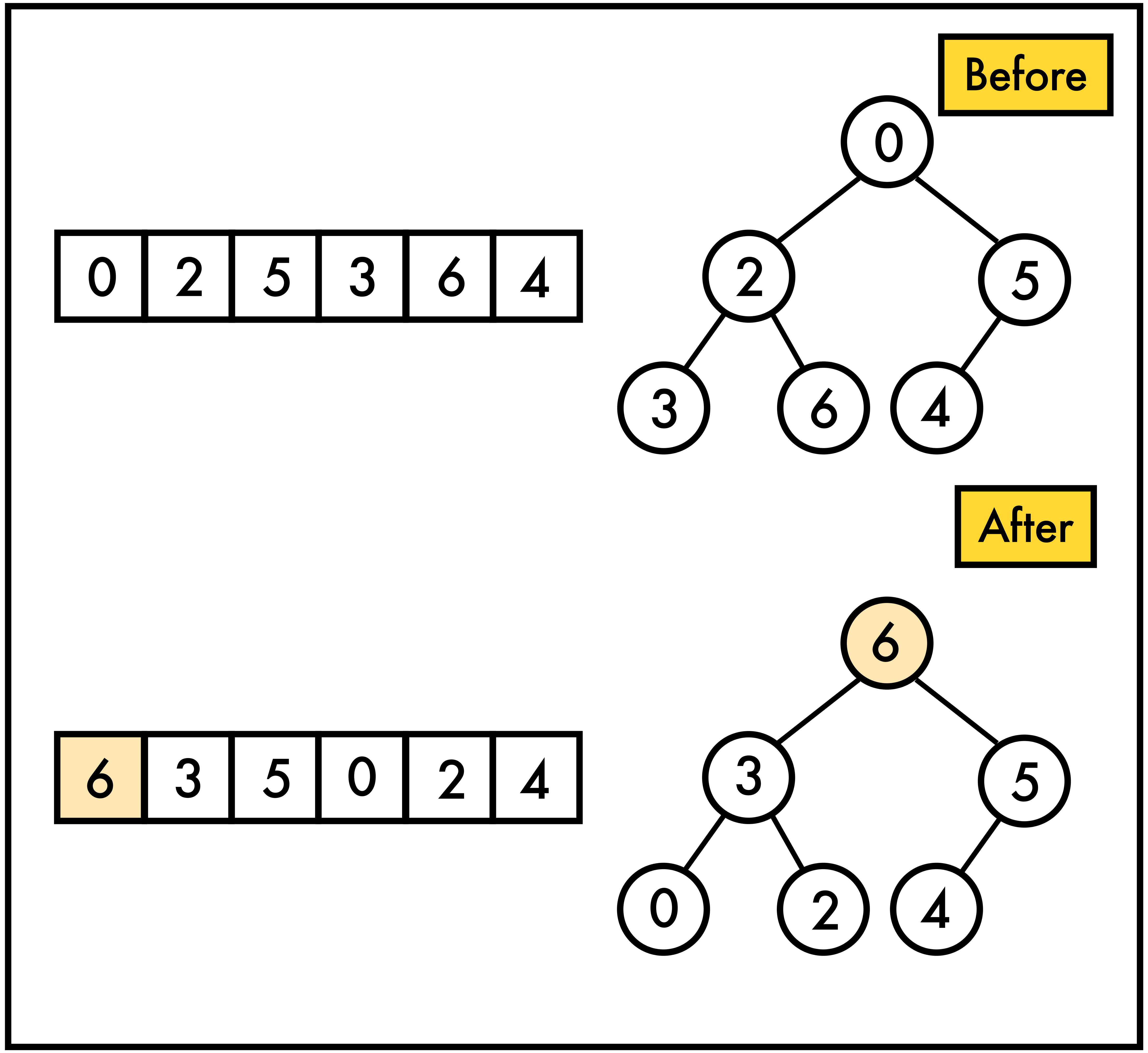
Building a Max-Heap

Somewhat remarkably, we can build a max-heap from an unsorted array in linear time

Note that each leaf node is a valid max-heap

Strategy: call max_heapify on left half of array

```
def build_max_heap(A):
    heap_size = len(A)
    for i in range(heap_size // 2 - 1, -1, -1):
        max_heapify(A, heap_size, i)
```



References:
L. Xinyu, "Elementary Algorithms", Chap. 8 (2022)
T. Cormen et al., "Introduction to algorithms", Chap 6.2, MIT press (2022)

Max-Heap Construction has Linear Complexity

Heap construction is $\Theta(n)$ (CLRS)

`build_max_heap()` contains a for loop from $\lfloor n/2 \rfloor - 1$ to 0 so construction is $\Omega(n)$

To show construction is $O(n)$, we can write the total cost as:

$$\sum_{h \in \text{node heights}} \text{num. nodes at height } h \times \text{cost of max\_heapify at height } h$$

We then use three observations: a heap with n keys has height $\lfloor \log n \rfloor$

a heap with n keys has at most $\lceil n/2^{h+1} \rceil$ nodes at height h

max_heapify is $O(h)$ for node at height h

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot Ch \leq \sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} \cdot Ch$$

Using that $\lceil n/2^{h+1} \rceil \leq n/2^h$

since $\lceil u \rceil \leq 2u$ for $u \geq 1/2$

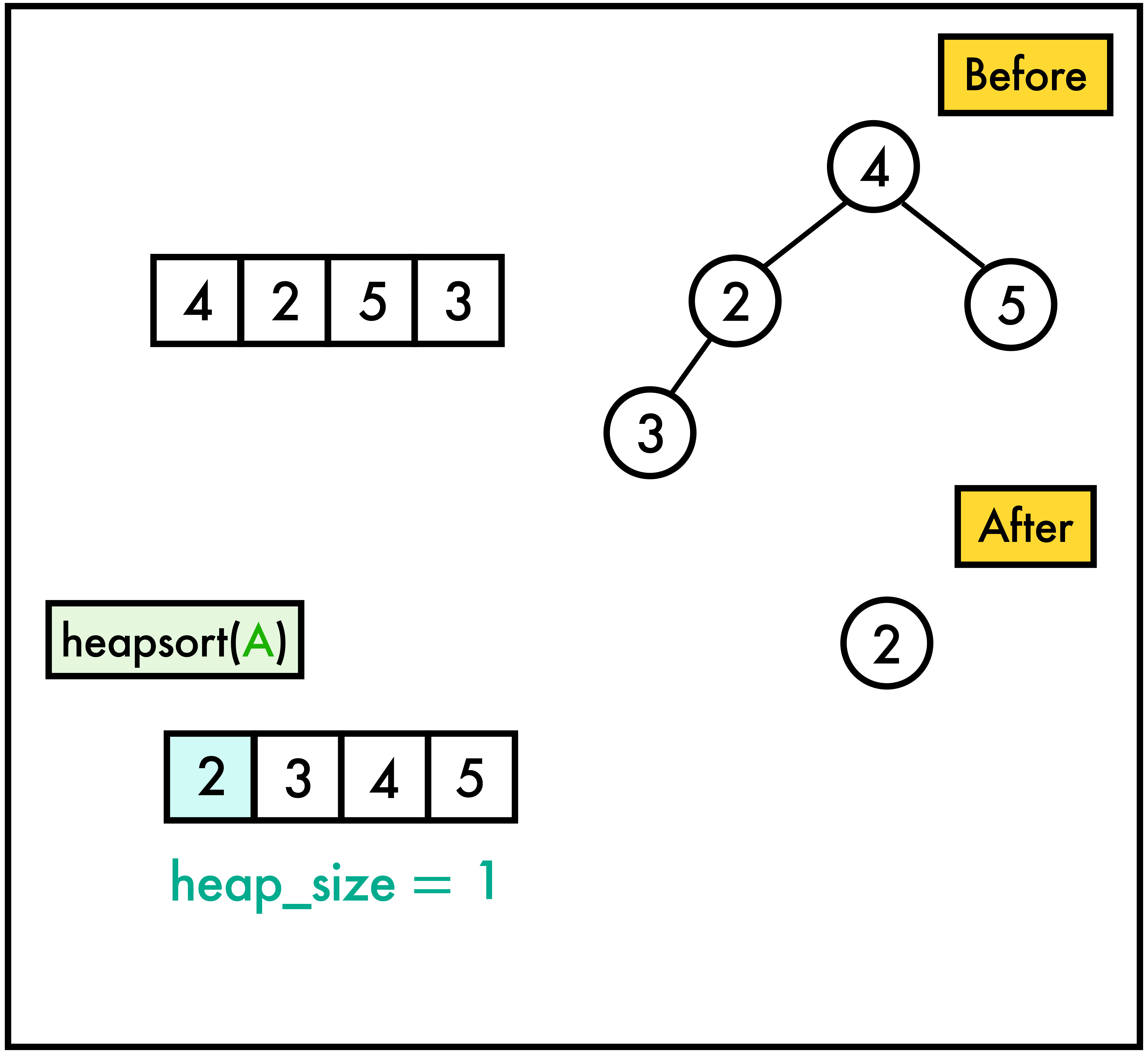
$$= Cn \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq Cn \sum_{h=0}^{\infty} \frac{h}{2^h} \leq Cn \cdot \text{constant} = O(n)$$

References:
H. Ding, "Proof for The Complexity of Building A Heap", https://www.cse.msu.edu/~huding/331material/timecomplexity_for_heap.pdf
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap 6.3, MIT press (2022)

Heapsort

```
def heapsort(A): # Floyd variant (in-place)
    build_max_heap(A)
    heap_size = len(A)
    while heap_size > 1:
        A[heap_size - 1], A[0] = A[0], A[heap_size - 1]
        heap_size = heap_size - 1
        max_heapify(A, heap_size, 0)
```

Complexity $O(n \log n)$



References:
L. Xinyu, "Elementary Algorithms", Chap. 8 (2022)
(CLRS) T. Cormen et al., "Introduction to algorithms", Chap 6.4, MIT press (2022)

Application of Heaps: Priority Queues

Priority Queues

Beyond heapsort, binary heaps are also often used to implement Priority Queues (PQs)

Priority Queue is an Abstract Data Type

A Priority Queue contains a collection of values each associated with a key

As with heaps there are max-PQs and min-PQs

Max-Priority Queue Interface

`get_max()` - return value with largest key

`pop_max()` - pop & return value with largest key

`insert(value, key)` - insert `value` with a key of `key` into the priority queue

`increase_key(value, key)` increase the key associated with `value` to `key`

Min-Priority Queue	insert	get_min	pop_min	decrease_key

Some applications of Priority Queues

Bandwidth manager (max-PQ)	Job scheduler (max-PQ)	Dijkstra's algorithm (min-PQ)	Huffman Coding (min-PQ)

References:
T. Cormen et al., "Introduction to algorithms", Chap 6.5, MIT press (2022)
(Applications of priority queues) https://en.wikipedia.org/wiki/Priority_queue#Applications

Max Priority Queues - get_max()/pop_max()

Assumptions

Keys/values stored in the priority queue as dicts

`{"key": key, "value": value}`

Values stored in the priority queue are unique

Class `MaxPQ` has attributes `heap_size` and `A`

```
def get_max(self): # self is a MaxPQ
  return self.A[0]["value"]
```

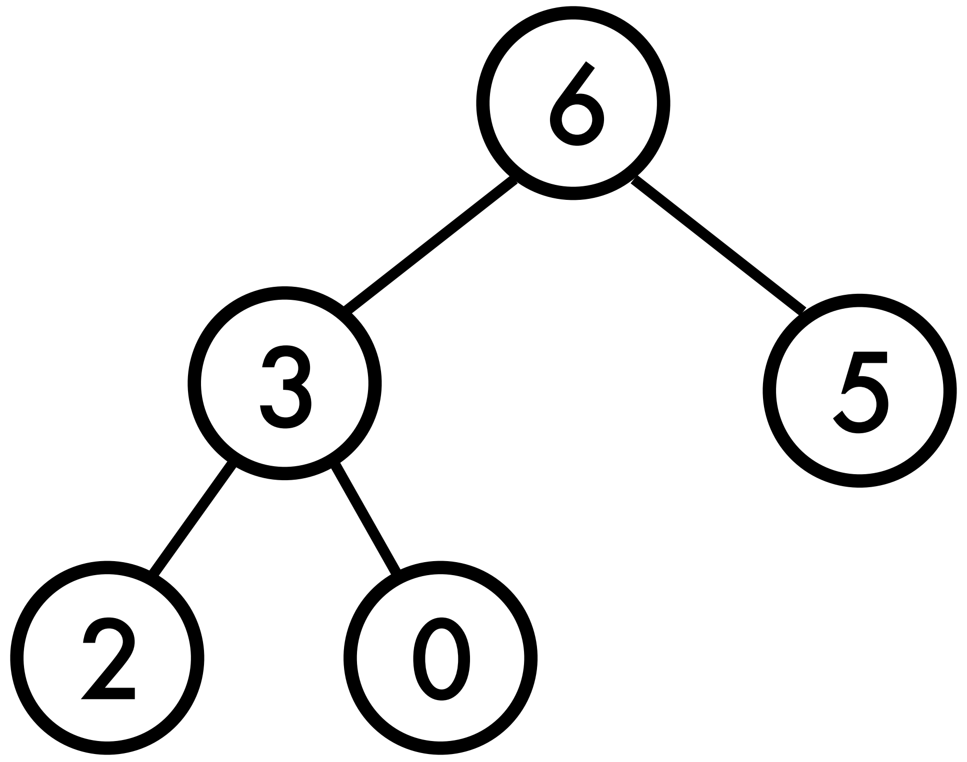
Complexity $\Theta(1)$

```
def pop_max(self): # self is a maxPQ
    max_value = self.get_max()
    self.A[0] = self.A[self.heap_size - 1]
    self.heap_size -= 1
    max_heapify(self.A, self.heap_size, 0)
    return max_value
```

Complexity $O(\log n)$

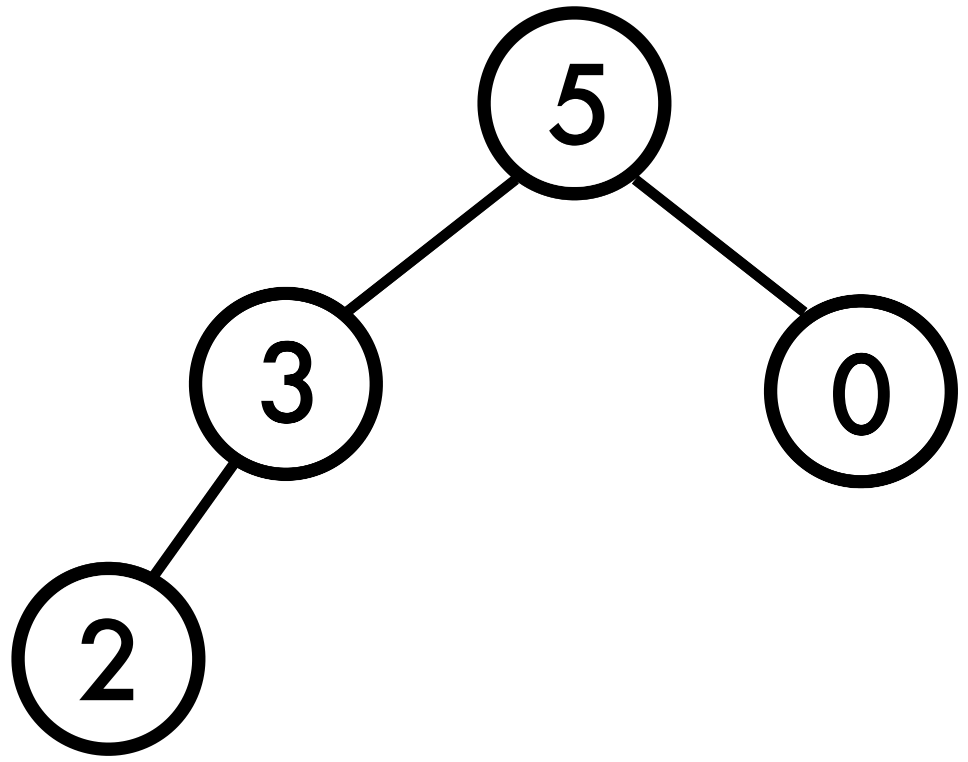
Before

Keys	6	3	5	2	0
Values	B	A	C	L	M



After

Keys	5	3	0	2	6
Values	M	A	C	L	B



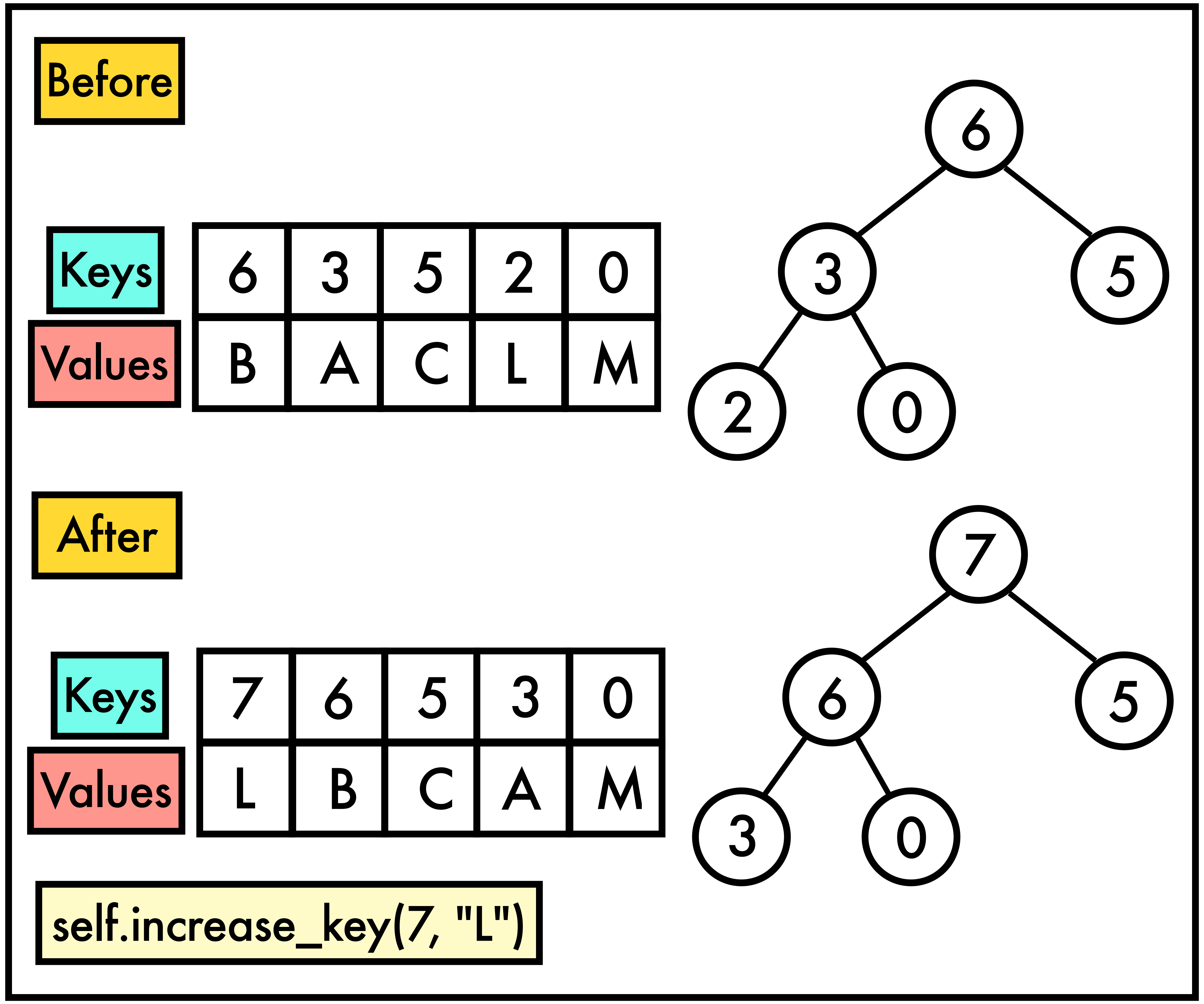
max_value = B

References:
L. Xinyu, "Elementary Algorithms", Chap. 8 (2022)
T. Cormen et al., "Introduction to algorithms", Chap 6.5, MIT press (2022)

Max Priority Queues - increase_key()

```python
def increase_key(self, key, value): # self is a MaxPQ
    # locate index of `value` in underlying array using an
    # auxiliary data structure (e.g. a red-black tree)
    i = self.value2index[value]
    assert key >= self.A[i]["key"], "requested to decrease key"
    self.A[i]["key"] = key # increase the key
    while i > 0 and self.A[i]["key"] > self.A[parent(i)]["key"]:
       self.A[i], self.A[parent(i)] = self.A[parent(i)], self.A[i]
       i = parent(i)
```

Complexity $O(\log n)$



References:
L. Xinyu, "Elementary Algorithms", Chap. 8 (2022)
T. Cormen et al., "Introduction to algorithms", Chap 6.5, MIT press (2022)

Max Priority Queues - insert()

```
def insert(self, key, value):  # self is a MaxPQ
    if self.heap_size == len(self.A):
        # expand underlying array to avoid heap overflow
        self.A.append(None)
    tmp_key = float("-inf")
    self.A[self.heap_size] = {"key": tmp_key, "value": value}
    self.heap_size += 1
    self.increase_key(key, value)
```

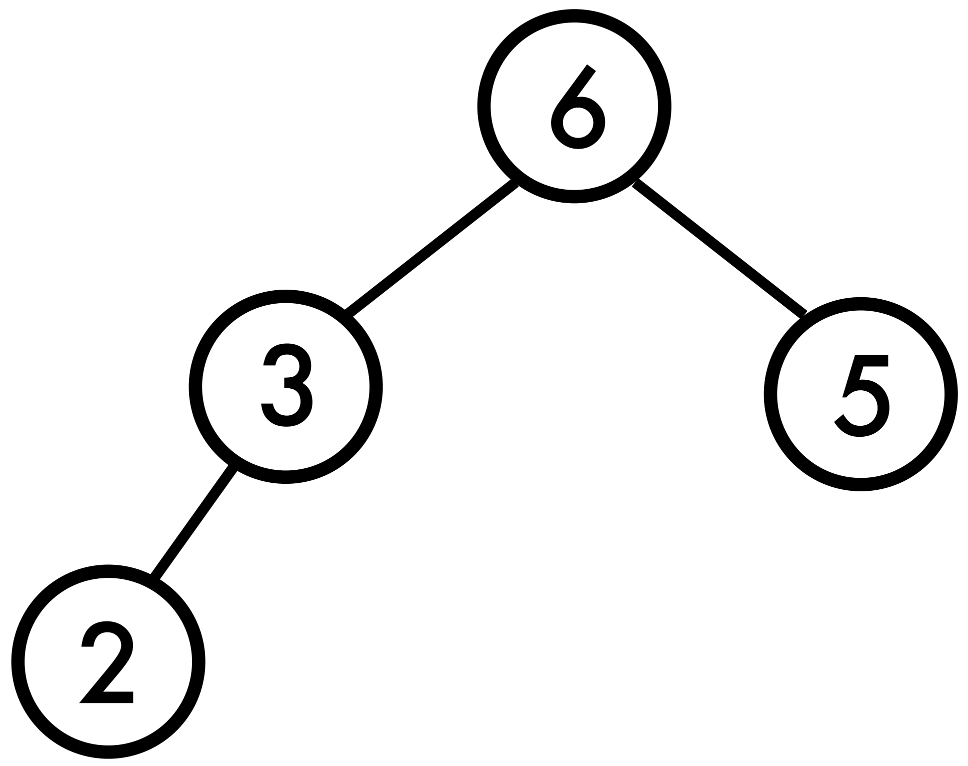
Complexity $O(\log n)$

Important Max Priority Queue operations			
pop_max	$O(\log n)$	insert	$O(\log n)$

Important Min Priority Queue operations			
pop_min	$O(\log n)$	insert	$O(\log n)$

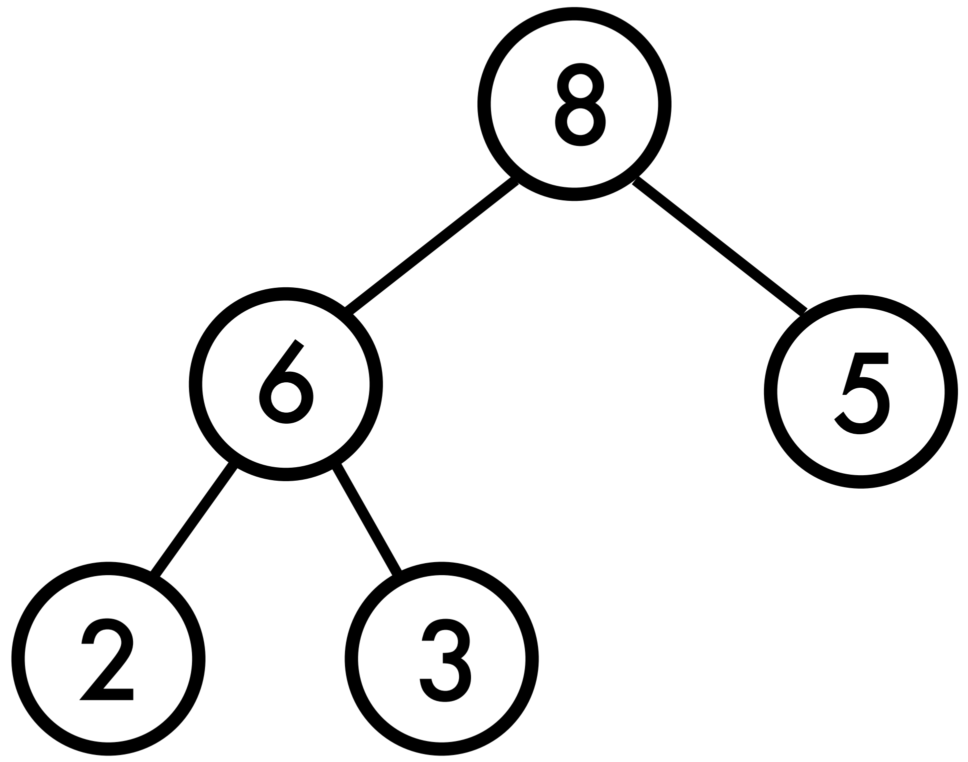
Before

Keys	6	3	5	2
Values	B	A	C	L



After

Keys	8	6	5	2	3
Values	G	B	C	L	A



self.insert(8, "G")

References:
L. Xinyu, "Elementary Algorithms", Chap. 8 (2022)
T. Cormen et al., "Introduction to algorithms", Chap 6.5, MIT press (2022)
M. T. Goodrich et al., "Algorithm design and applications", Chap. 5 (2015)

Quicksort

Quicksort

Widely-used, fast, in-place sorting algorithm

Devised by Tony Hoare in Moscow (1959) 

Won wager with boss (who advocated shellsort)

Published in ALGOL using recursion (1961)

Note: Quicksort not a stable sort (sorting does not preserve input order of identical keys)

Quicksort complexity (for n data items)

Average case: $\rightarrow O(n \log n)$

Worst–case: $\rightarrow O(n^2)$

Storage: $\Theta(n)$ for items, sort: $O(\log n)$ avg. case, $O(n)$ worst case | stack usage | $O(\log n)$ with tail call optim.

Quicksort implementations are typically cache-friendly (linear scans suit caches)

Employed in hybrids (e.g. Introsort - falls back to heapsort to avoid $O(n^2)$ worst-case behaviour)

References/Notes/Image credits:
C. A. R. Hoare, "Algorithm 64: quicksort", *Communications of the ACM* (1961)
(Portrait of Hoare) http://curation.cs.manchester.ac.uk/Turing100/www.turing100.manchester.ac.uk/speakers/invited-list/11-speakers/39-hoare.html
(Quicksort history) https://anothercasualcoder.blogspot.com/2015/03/my-quickshort-interview-with-sir-tony.html
(Elliot Brothers Ltd image) https://en.wikipedia.org/wiki/Elliott_Brothers_(computer_company)#/media/File:Elliott_Sector.jpg
Storage complexity of quicksort: https://stackoverflow.com/a/29746572 (technically you can guarantee $O(\log n)$ worst case memory if you have tail call optimisation)
(Introsort) https://en.wikipedia.org/wiki/Introsort

Quicksort Overview

Quicksort

Uses a recursive, divide-and-conquer strategy:

1. Select a pivot from the array
2. Partition array into 3 subarrays arranged as:

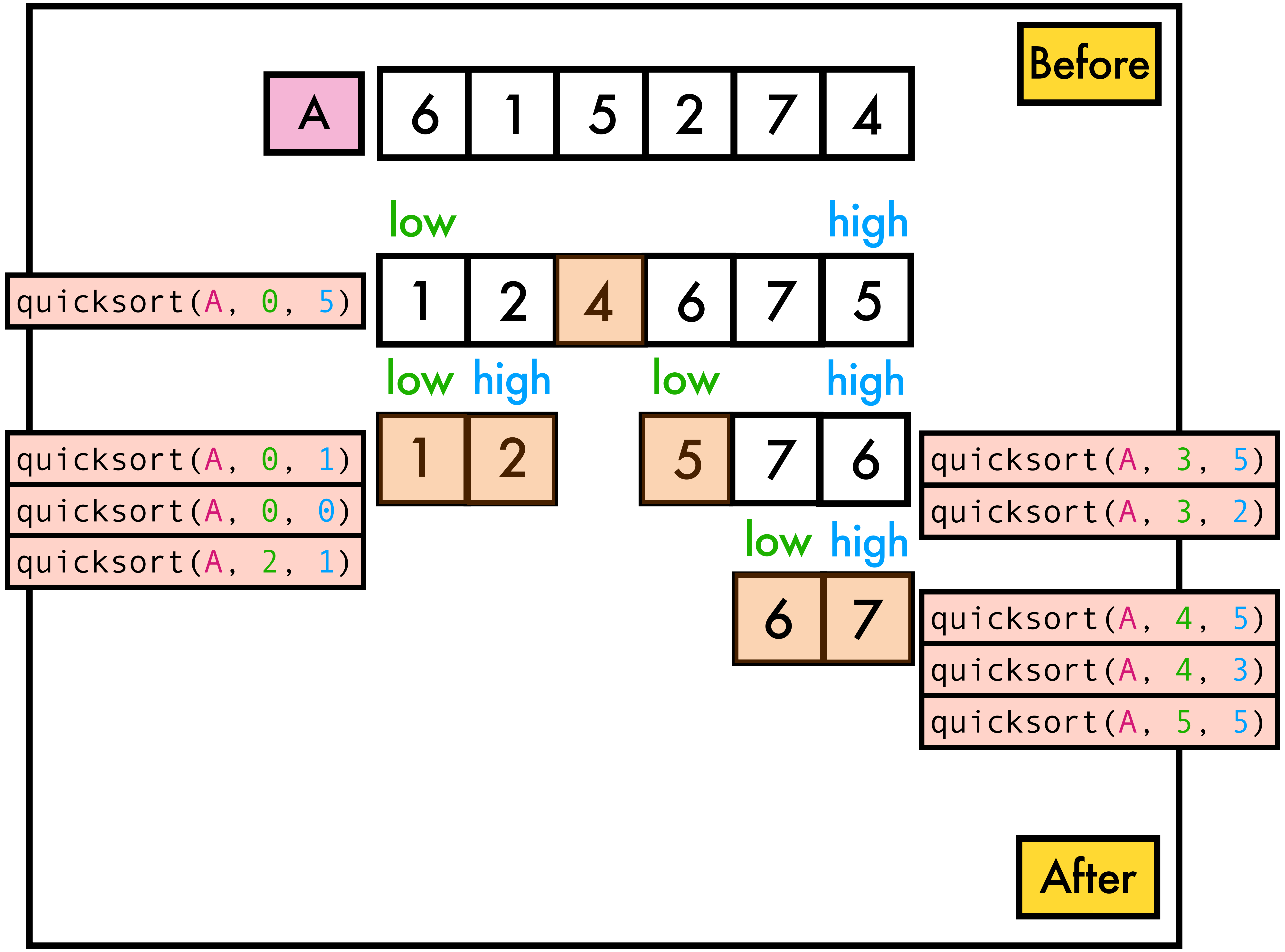
elements ≤ pivot	pivot	elements ≥ pivot

can be empty

3. Recursively quicksort the first and last subarrays

elements ≤ pivot	pivot	elements ≥ pivot

```
def quicksort(A, low, high):
    if low < high:
        pivot = partition(A, low, high)
        quicksort(A, low, pivot-1) # left subarray
        quicksort(A, pivot+1, high) # right subarray
```



References:
J. Erickson, "Algorithms", Chap. 2, http://algorithms.wtf/ (2019)
T. Cormen et al., "Introduction to algorithms", Chap 7.1, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 13 (2022)

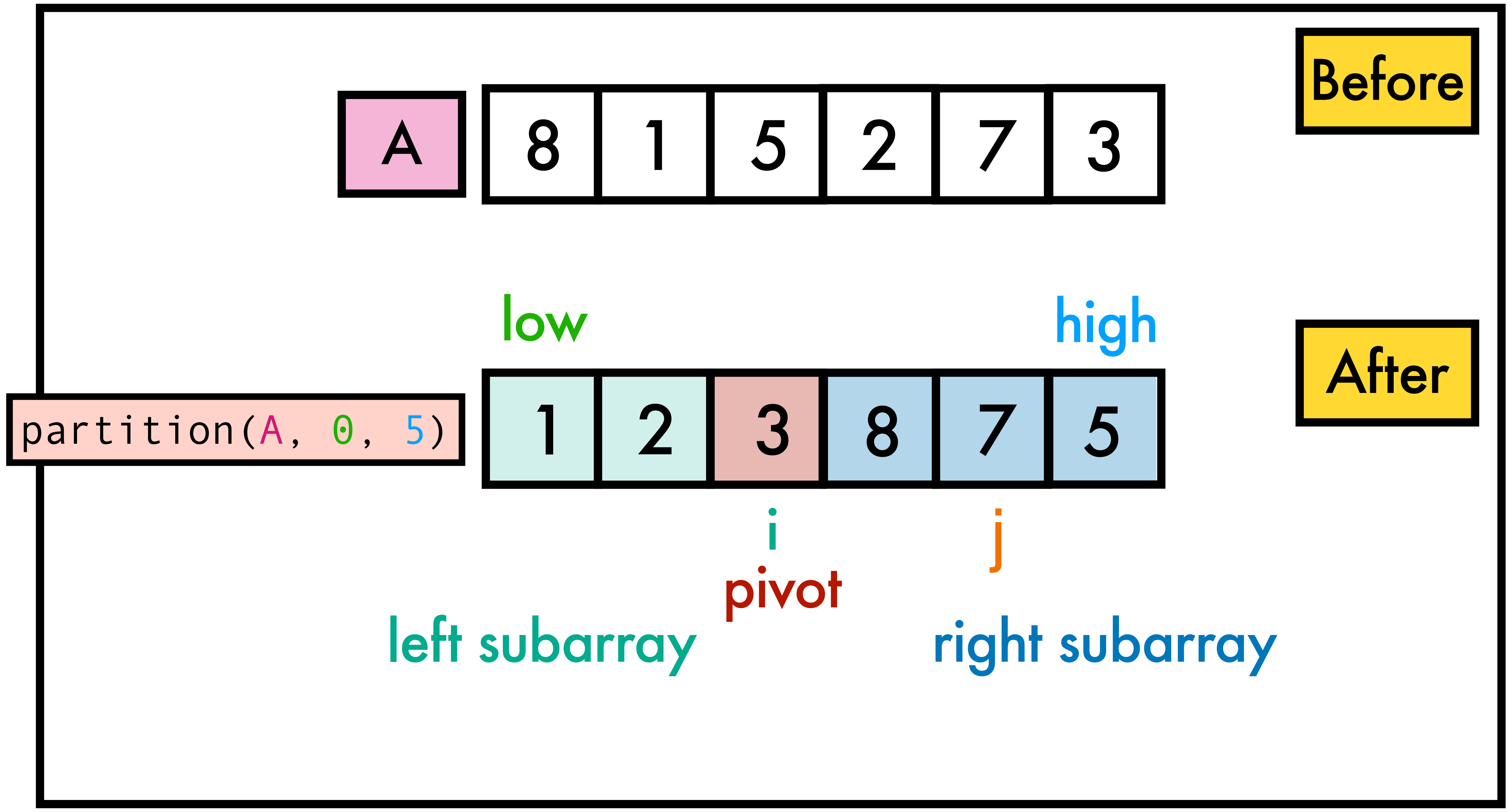
Lomuto Partition

Our first partition function (Lomuto)

partition() does most of the work in quicksort

```
def partition(A, low, high): # lomuto partition
    pivot = high
    pivot_val = A[pivot]
    i = low
    for j in range(low, high):
        if A[j] <= pivot_val:
            A[i], A[j] = A[j], A[i]
            i = i + 1
    A[i], A[pivot] = A[pivot], A[i]
    return i # new pivot
```

Complexity: $\Theta(n)$ where $n = \text{high} - \text{low} + 1$



References:
J. Erickson, "Algorithms", Chap. 2, http://algorithms.wtf/ (2019)
T. Cormen et al., "Introduction to algorithms", Chap 7.1, MIT press (2022)
L. Xinyu, "Elementary Algorithms", Chap. 13 (2022)

Worst-Case Analysis

Unbalanced partitions and quadratic complexity

How can things go badly for quicksort?

We have seen that `partition()` is $\Theta(n)$

Since quicksort is recursive, can write complexity:

$$T(n) = \underbrace{T(r)}_{\text{left subarray}} + \underbrace{T(n-r-1)}_{\text{right subarray}} + \underbrace{\Theta(n)}_{\text{partition}}$$

where r is the rank of the pivot

If we choose the pivot as smallest value $(r = 0)$

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

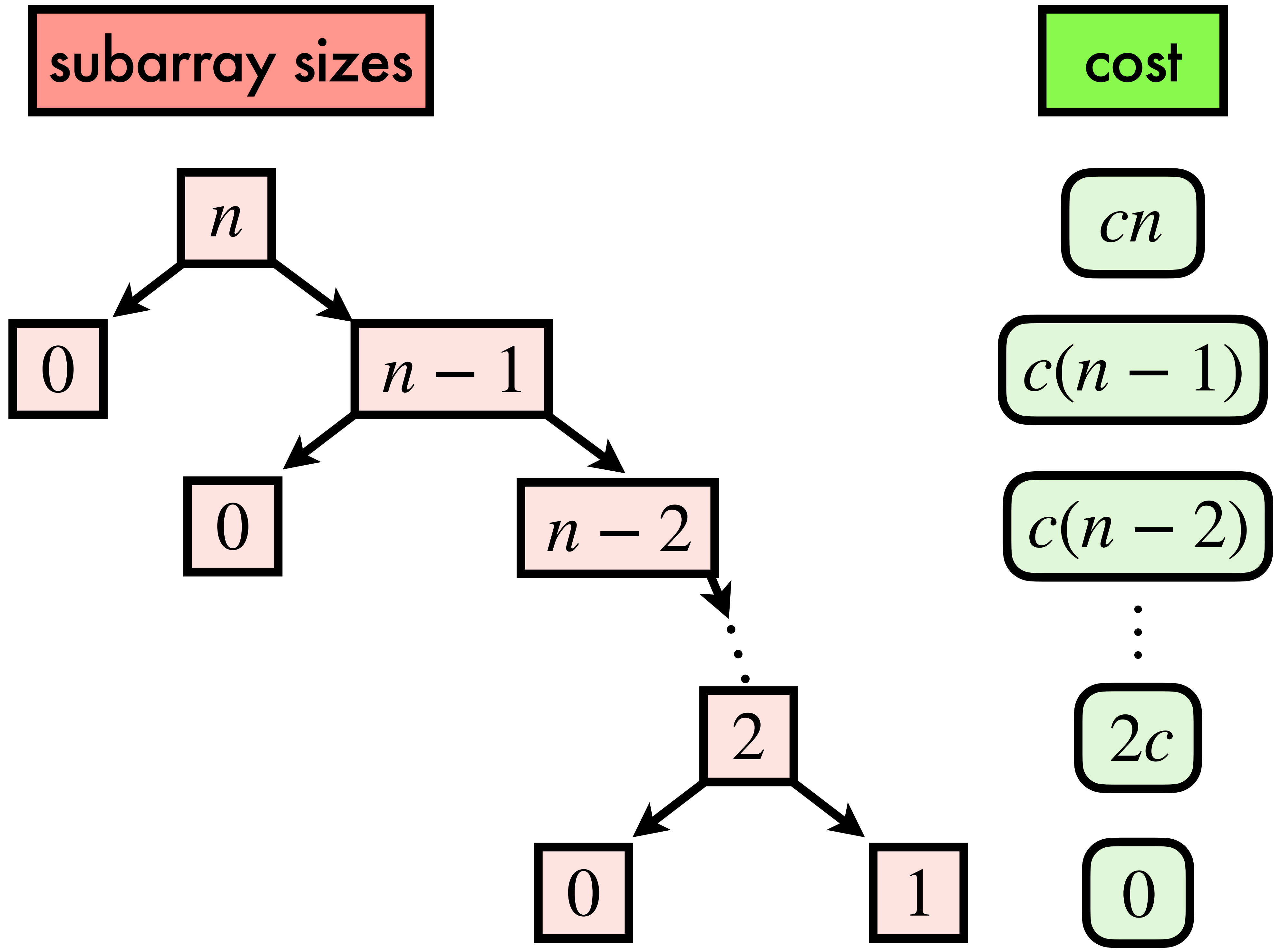
$T(0) = O(1)$ (empty subarray - no comparisons)

$T(n) = T(n-1) + \Theta(n) \implies T(n) = \Theta(n^2)$

$T(n) = \Theta(n^2)$ for fixed r not dependant on n

Illustrate runtime complexity with $r = 0$

Partition cost $\Theta(n)$ as cn for some constant c



Total cost $= c(n + (n-1) + (n-2) + \ldots + 2 + 0)$

$$= c\sum_{i=2}^{n} i = c(n(n+1)/2 - 1) \implies \Theta(n^2)$$

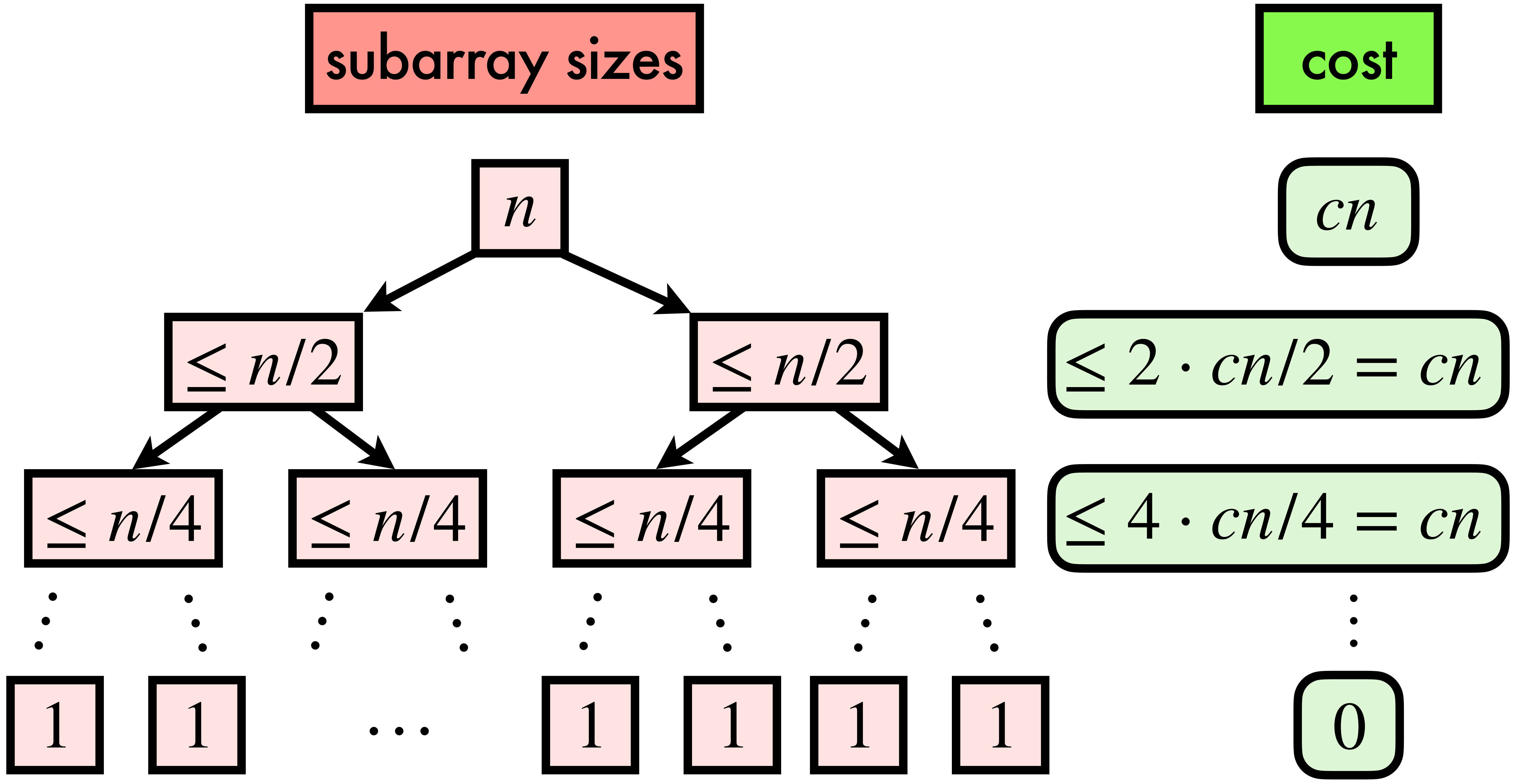
References:
J. Erickson, "Algorithms", Chap. 2, http://algorithms.wtf/
T. Cormen et al., "Introduction to algorithms", Chap 7, MIT press (2022)
https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Balanced Partition Analysis

Quicksort is happy with $r = K \cdot n$ for $0 < K < 1$

By happy, we mean $O(n \log n)$

Consider the simple case when $K = 0.5$:

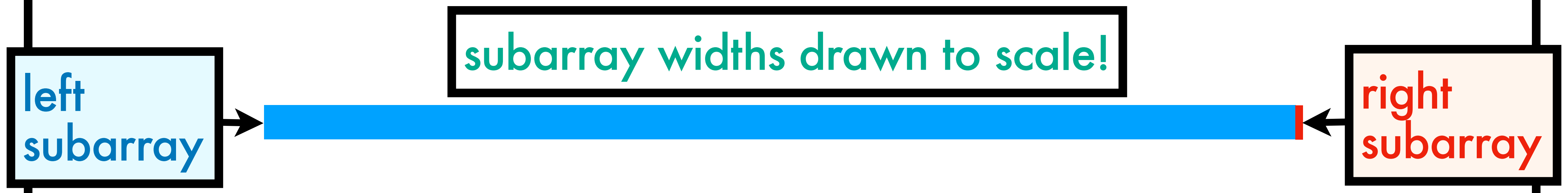


Tree height is $\log n$ and cost at each level $\leq cn$

Total cost $\leq cn \cdot \log n \implies O(n \log n)$

For complexity, what matters is tree depth growth

Suppose $K = 0.995$ (seems unbalanced!)



Tree depth is $\log_{\frac{1000}{995}} n$

So the complexity is still $O(n \log n)$

Some intuition for why "most" partitions will produce $O(n \log n)$ behaviour

References:
Fig. based on https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Engineering Improvements

Behaviour on concerningly common inputs

If array already sorted, Lomuto partitioning has worst case behaviour $O(n^2)$

If all elements the same, Lomuto is also $O(n^2)$

"Median-of-three"

A simple heuristic to get a better pivot: pick median of first, middle and last elements

Useful on sorted input arrays (and other inputs)

Does not rule out worst case behaviour

Randomised quicksort

Pick pivot uniformly at random and swap it with: element at index $n-1$ (if using Lomuto partition)

Expected runtime $O(n \log n)$

The full proof is quite involved (see refs)

Worst case becomes very unlikely (widely used)

Provoking $O(n^2)$ behaviour

"Killer adversary for quicksort" (McIlroy, 1999)

Decide ordering of elements lazily during sorting

Achieving $O(n \log n)$ worst case

Can ensure $O(n \log n)$ worst case quicksort via $O(n)$ median-of-medians algorithm

In practice, this is too slow, so not widely used

References:
(Median-of-3) R. Sedgewick, "Algorithms in C: Fundamentals, Data Structures, Sorting, Searching" (1998)
(Proof for randomised quicksort) https://www.cs.cmu.edu/~avrim/451f11/lectures/lect0906.pdf
T. Cormen et al., "Introduction to algorithms", Chap 7, MIT press (2022)
D. M. McIlroy, "A killer adversary for quicksort", Software: Practice and Experience (1999)
(quickselect) C. A. R. Hoare, "Algorithm 65: Find", *Communications of the ACM* (1961)