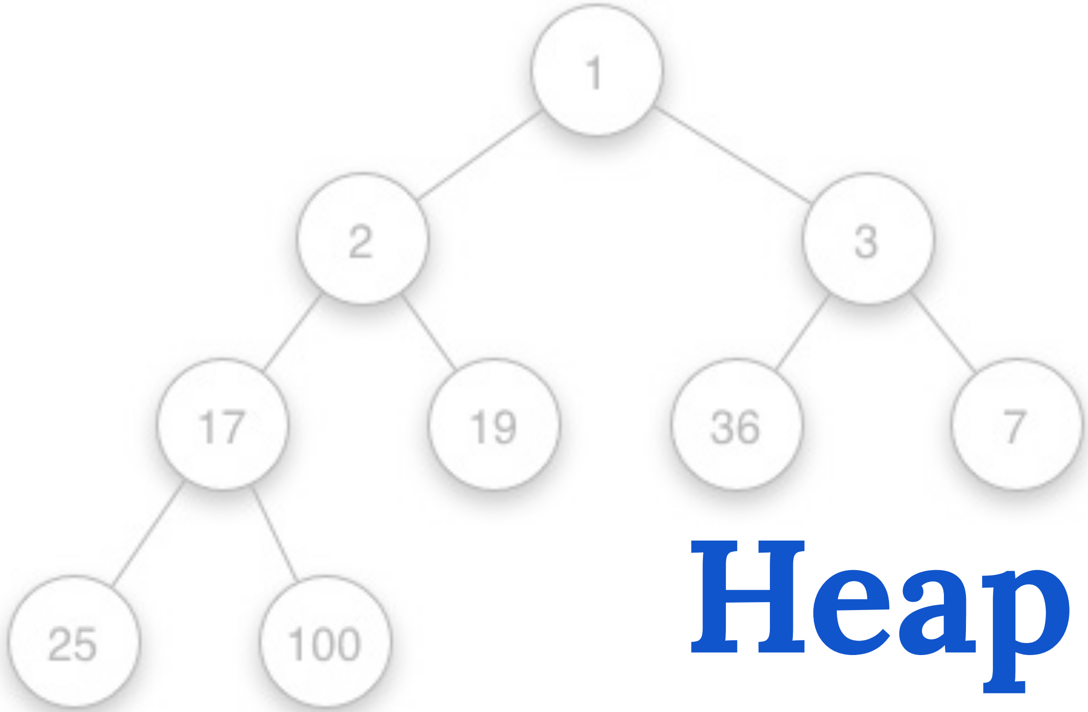


Heap

1	2	3	17	19	36	7	25	100

Objective

- Understand heap and its types (min-heap and max-heap)
- Implement heap data structure and operations
- Implement heap sort
- Use python **heapq** module
- Identify problems that can be efficiently solved using heap

Prerequisites

- Arrays
- Tree

Lecture Flow

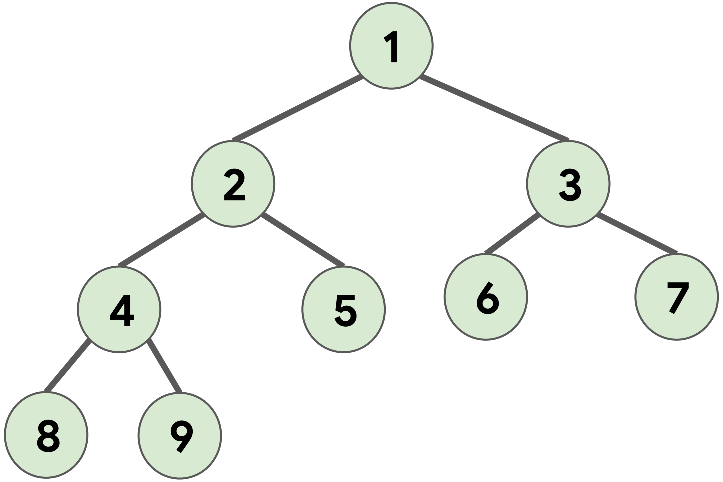
1. Motivation problem
2. Definition
3. Types
4. Representation
5. Operations
6. Time and space complexity
7. Practice
8. Heap construction
9. Heap sort
10. Heapq module
11. Problem patterns
12. Real world applications
13. Common pitfalls
14. Quote of the day

Motivation Problem

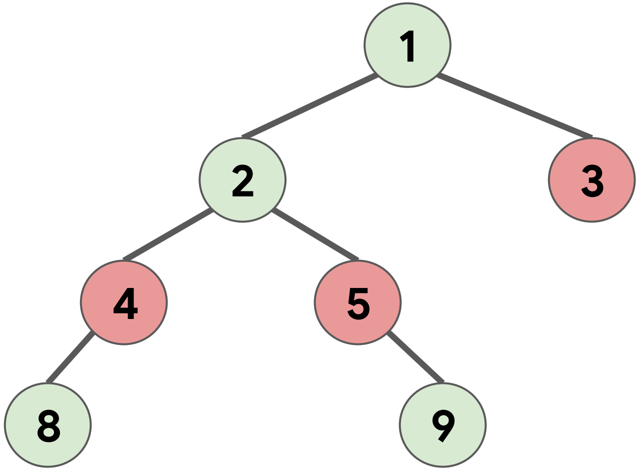
Definition

What is Heap?

A binary heap is a **complete binary tree** that satisfies the **heap order property.**



Complete binary tree



Not complete binary tree

Heap order property

The heap order property is a key characteristic that defines the structure of a heap.

Max heap order property

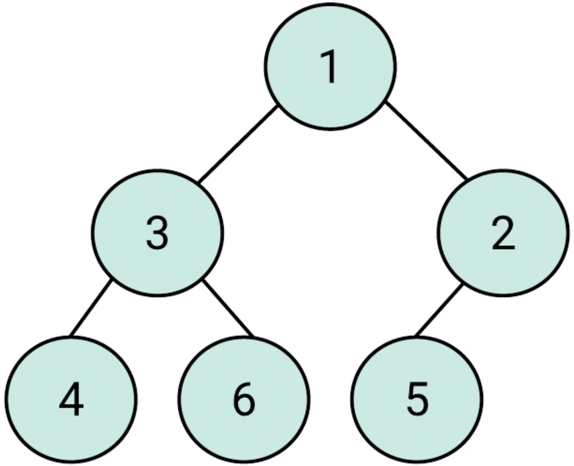
Value of node >= Value of all its descendants

Min heap order property

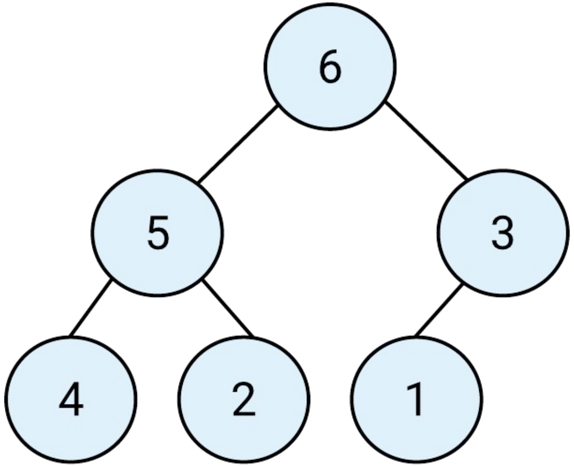
Value of node <= Value of all its descendants

For all nodes!

Types of Heaps



Min heap



Max heap

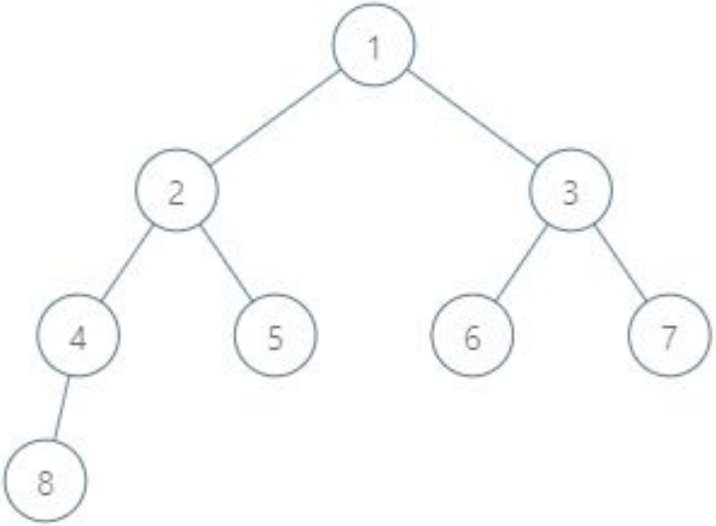
Representations

Using Binary TreeNode Class

Binary heaps can be efficiently implemented using a binary tree structure, allowing for logarithmic time complexity for different heap operations.

Trees can be represented using arrays which makes implementations of heap operations easier.

Representing Binary Tree Using Array



Parent index	Left child index	Right child index
0	1	2
1	3	4
2	5	6
3	7	-

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Parent index	Left child index	Right child index
0	1	2
1	3	4
2	5	6
3	7	-
p	2p + 1	2p + 2

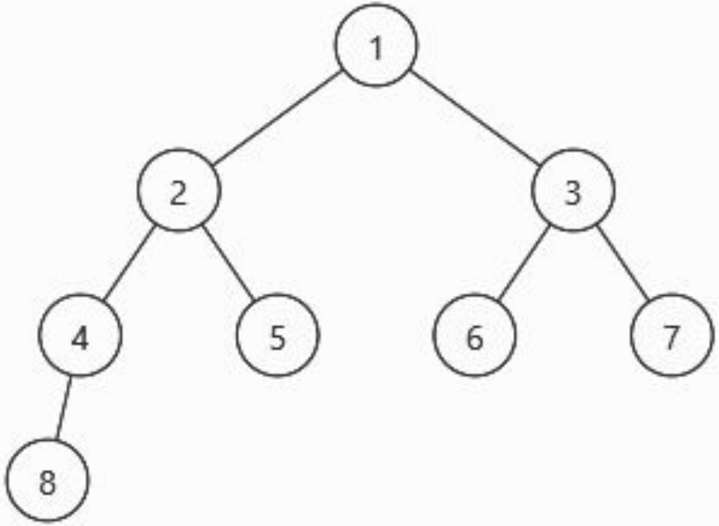
0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Representing Binary Tree Using Array

parent_idx = (child_idx - 1) // 2

left_child_idx = 2 * parent_idx + 1

right_child_idx = 2 * parent_idx + 2



Operations

In this lecture, we are going to focus on **min heaps**. However, the concepts discussed can seamlessly be extended to **max heaps** as well.

Insertion

To insert new value to min heap

1. Add it to the binary tree, however, we need to **maintain the complete binary tree property**, thus either place it
    a. *beyond the rightmost node at the bottom level of the tree, or*
    b. *as the leftmost position of a new level, if the bottom level is already full (or if the heap is empty).*

2. After this action, the tree is complete, but it **may violate the heap-order property.**

3. Fulfill the heap order property. **How?**

Up-heap Bubbling After Insertion

Compare the newly added value (Kn) with that of the parent (Kp)

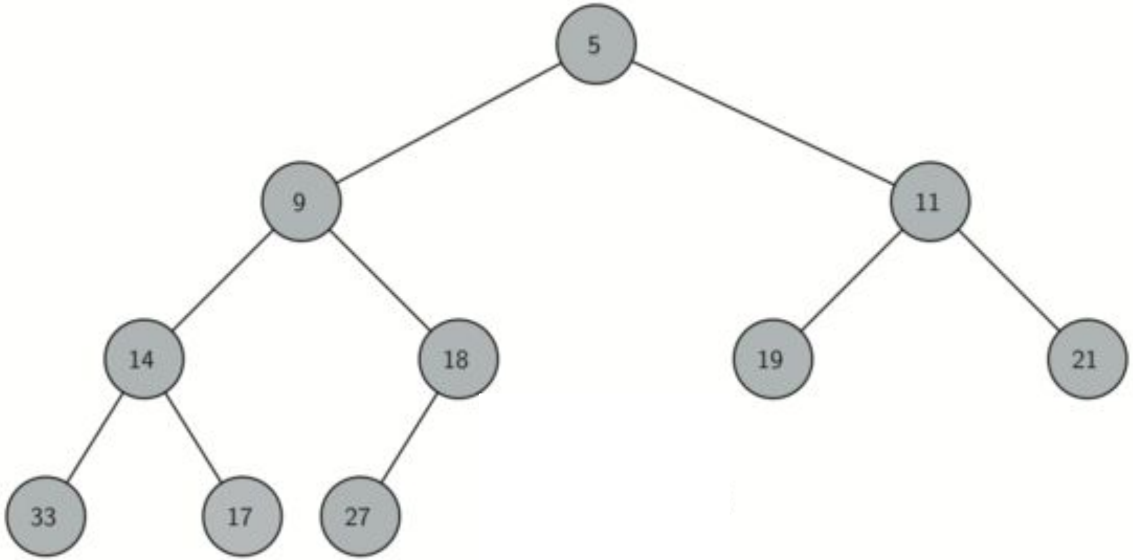
1. If Kn >= Kp, the min heap order property is satisfied, the algorithm terminates

2. If Kn < Kp, Violation!
    a. Swap child and parent, This swap causes the new item to **move up one level**. Hence the up heap bubbling.

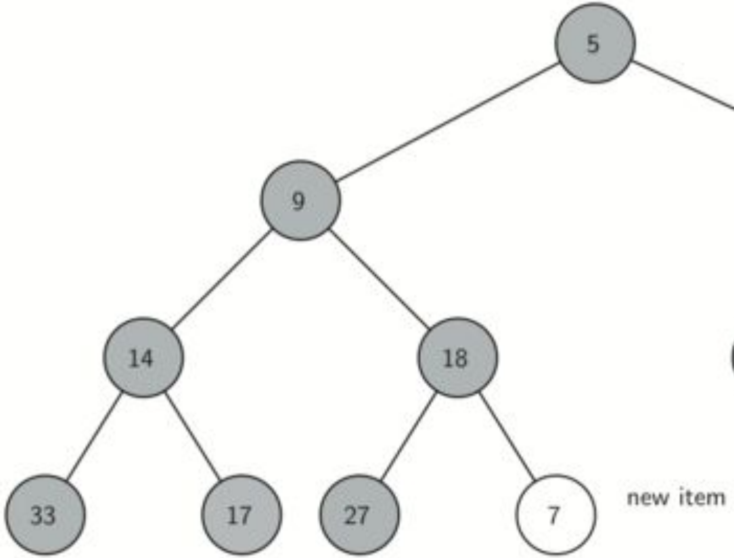
    b. Again the heap order property may be violated, so we repeat the process, going up in tree until no violation of the heap-order property occurs.

Insertion & Up-heap Bubbling Simulation

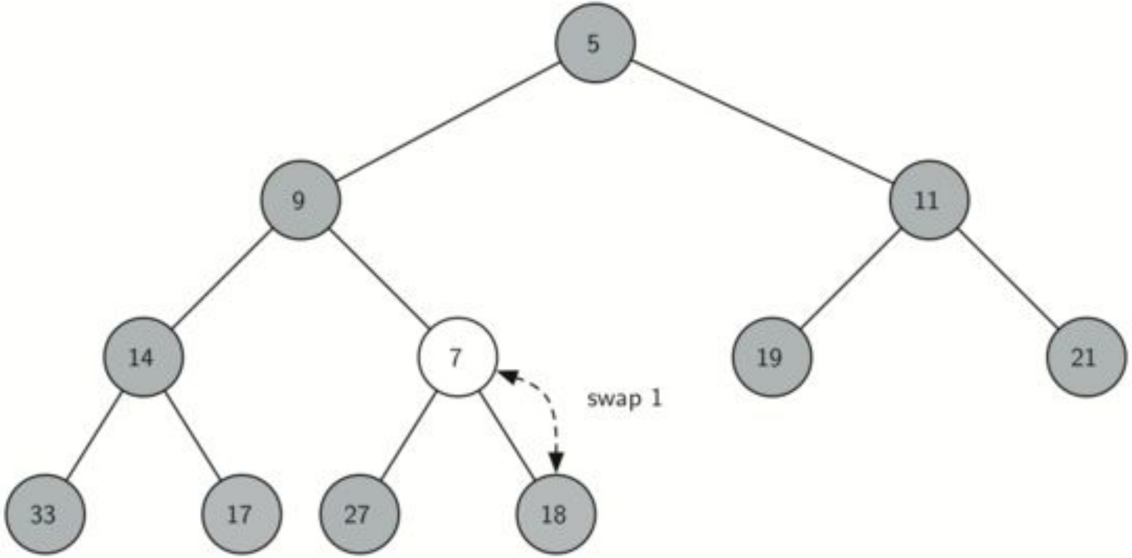
Let's insert **7** to the following heap



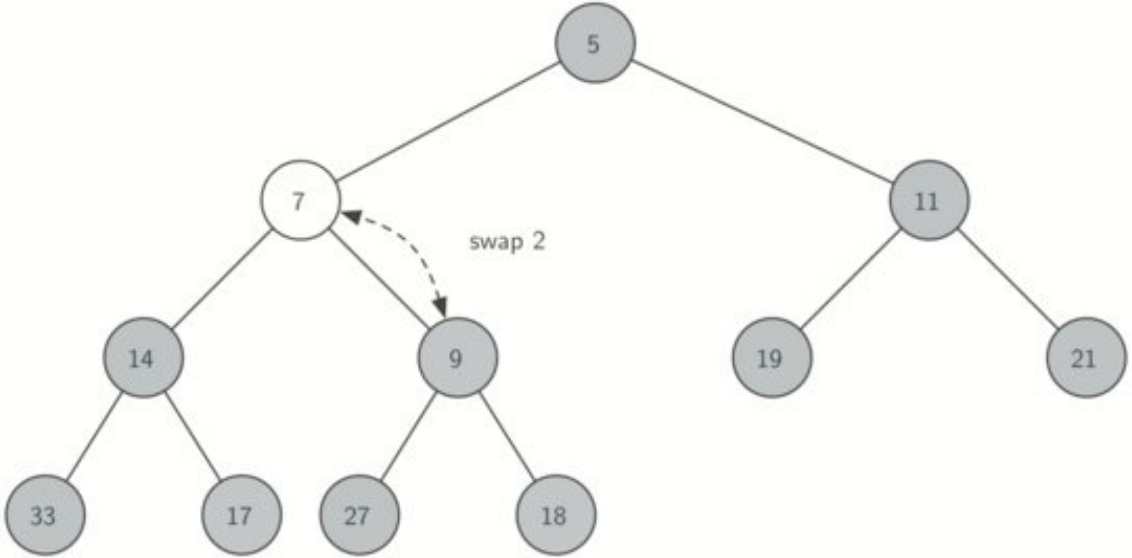
Insertion & Up-heap Bubbling Simulation



Insertion & Up-heap Bubbling Simulation



Insertion & Up-heap Bubbling Simulation



Time & Space Complexity of Insertion

Heapify up

- Time complexity - _____
- Space complexity - _____

Time & Space Complexity

Heapify up

- Time complexity - **O(log n)**
- Space complexity - **O(1)**

Why?

Can you implement heapify up (up-heap bubbling)?

Playground

Solution

```
def parent(j):
    return (j - 1) // 2

def swap(heap,i, j):
    heap[i], heap[j] = heap[j], heap[i]

def heappush(heap,value):
    heap.append(value)
    current = len(heap) - 1
    heapify_up(heap, current)

def heapify_up(heap, j):
    p = parent(j)
    if j > 0 and heap[j] < heap[p]:
        swap(heap,j, p)
        heapify_up(heap, p)
```

Removing Smallest Element

*The most straightforward element to remove from a tree is the **leaf**, while in an array, it corresponds to the **last element**.*

Let's take advantage of that!

1. Swap the root of the heap (first element in our array representation) with the last element in the array.

2. Remove the last element

3. The tree **remains complete binary tree**, however, it is very likely to **violate min heap order property**

4. Fulfill the min heap order property. **How?**

Down-heap Bubbling After Removal of Root Element

Let the **root** value be **(Kr)**, we can face the following two cases:

1. The root only has left child, let's denote the value of left child with **Kc**

2. The root has two children, let's denote the minimum value of the two by **Kc**

   i.e Kc = min(Kleft, Kright) … min value of the left and right child

Continued...

Then,

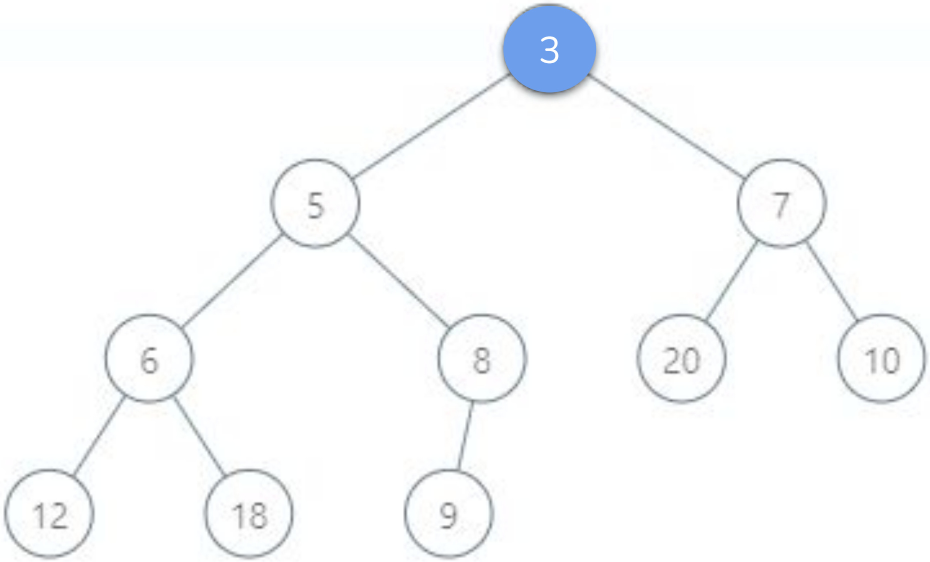
1. If Kr <= **Kc,** the heap order property is satisfied, the algorithm terminates

2. If Kr > **Kc**, **Violation!**
    a. Swap the smaller child with the parent

    b. Continue swapping down tree until no violation of the heap-order property occurs

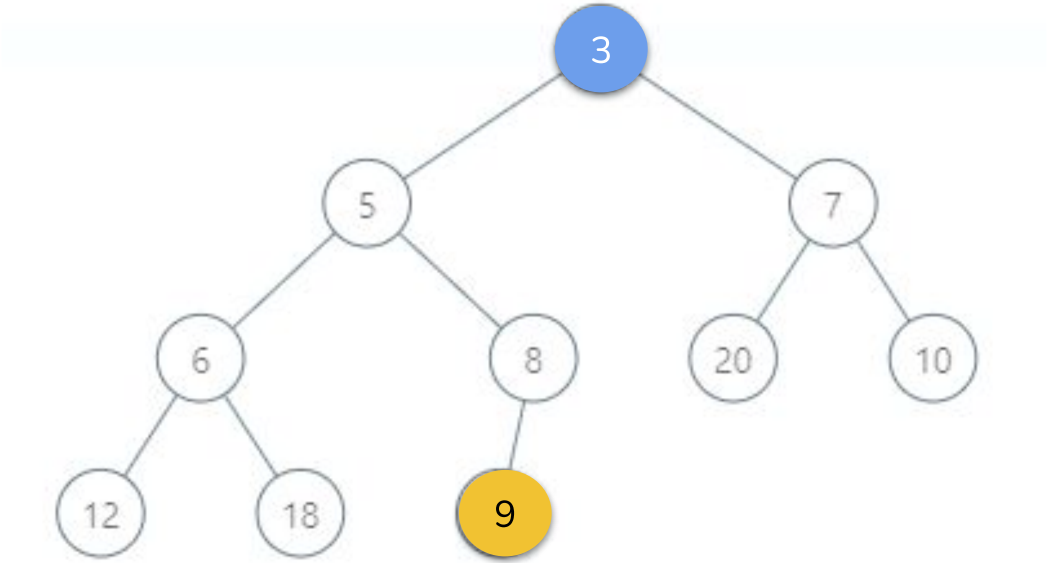
Removal & Down-heap Bubbling Simulation

Let's simulate removal of **3** from the following heap



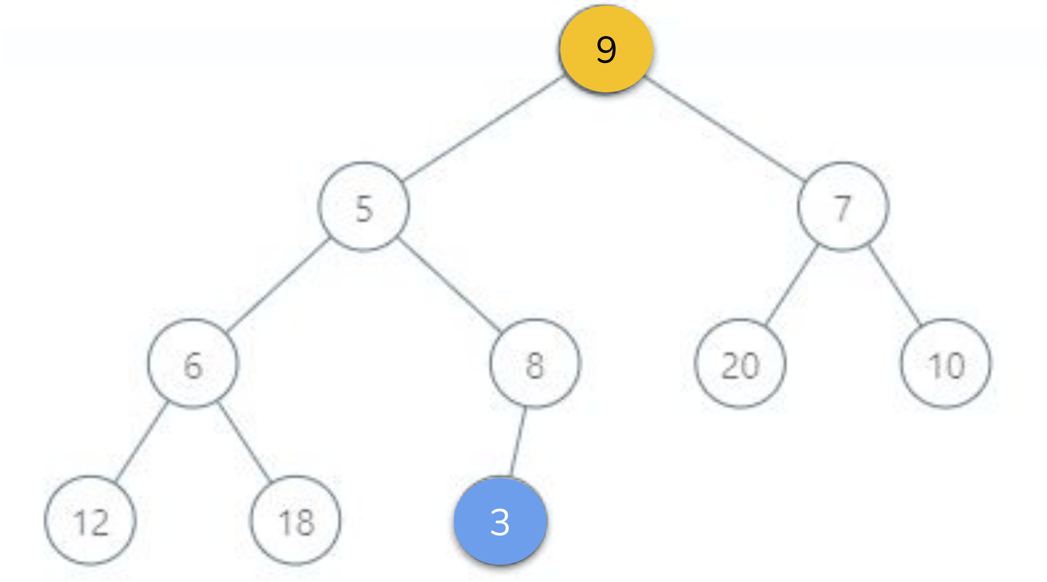
Removal & Down-heap Bubbling Simulation

Swap root with the last element



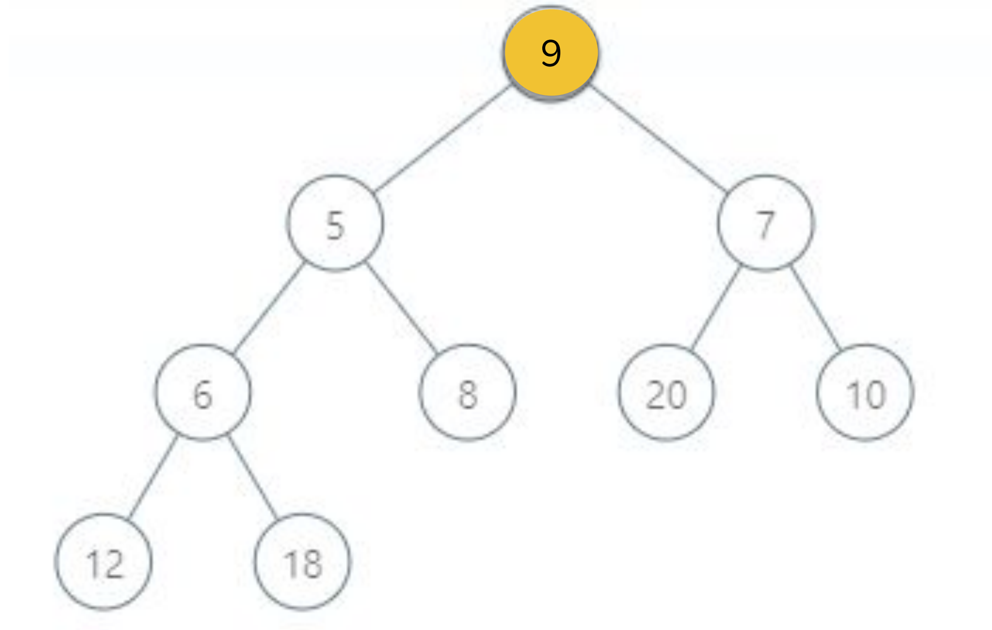
Removal & Down-heap Bubbling Simulation

Remove the last element



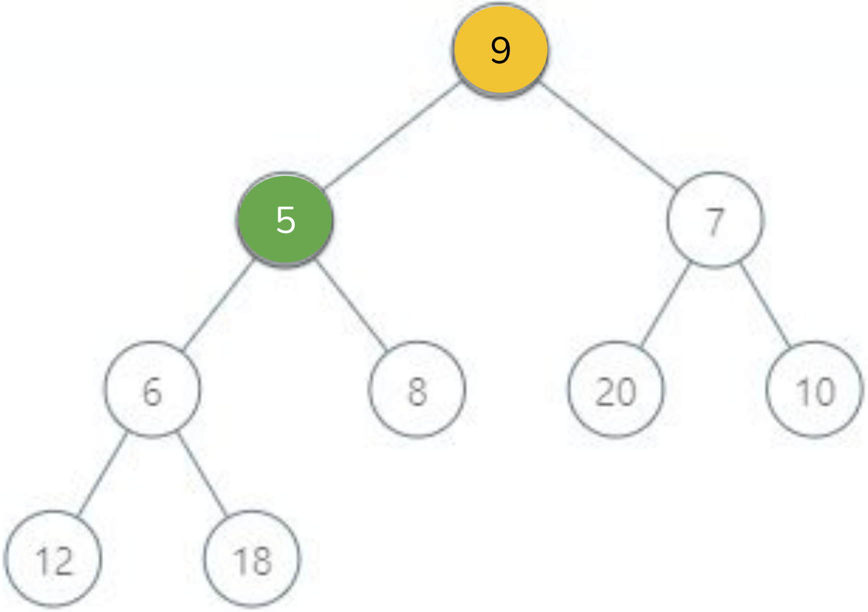
Removal & Down-heap Bubbling Simulation

We have removed the minimum element, however, the **heap order is violated.**

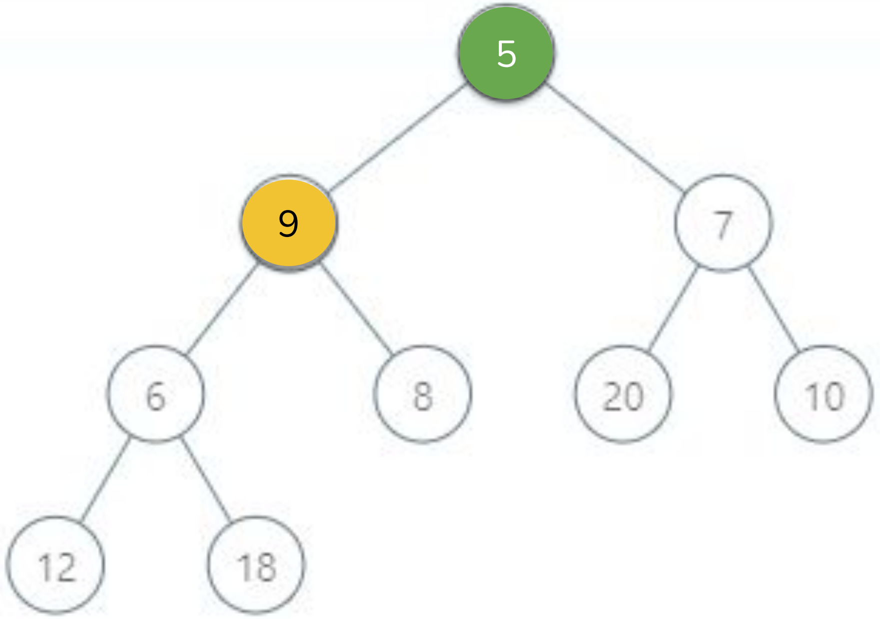


Removal & Down-heap Bubbling Simulation

Our current node, **the root**, **violates** heap order. Thus, promote the **smaller child**

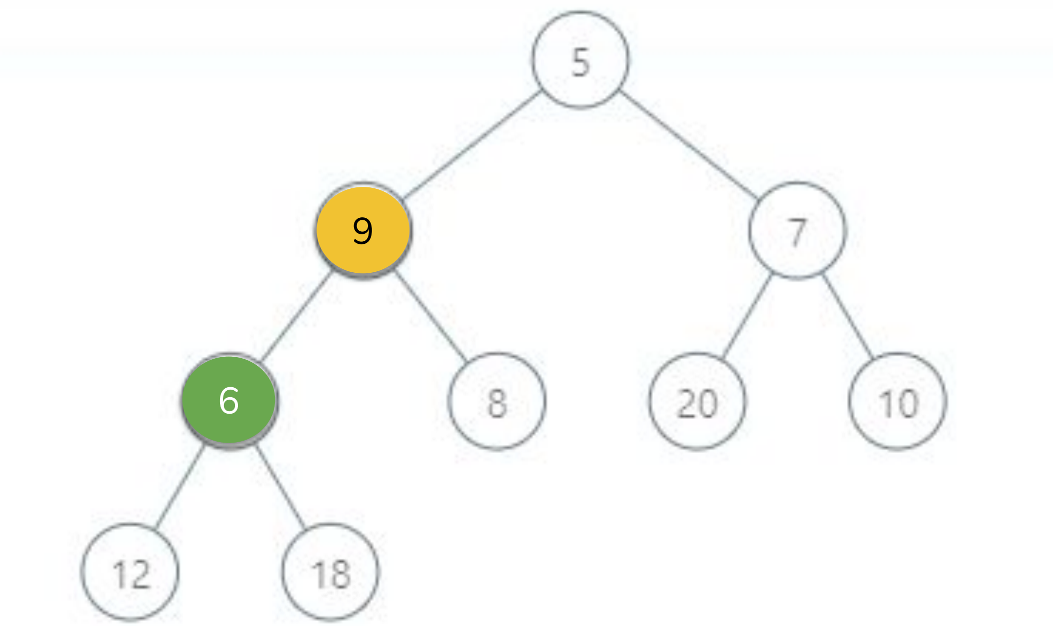


Removal & Down-heap Bubbling Simulation

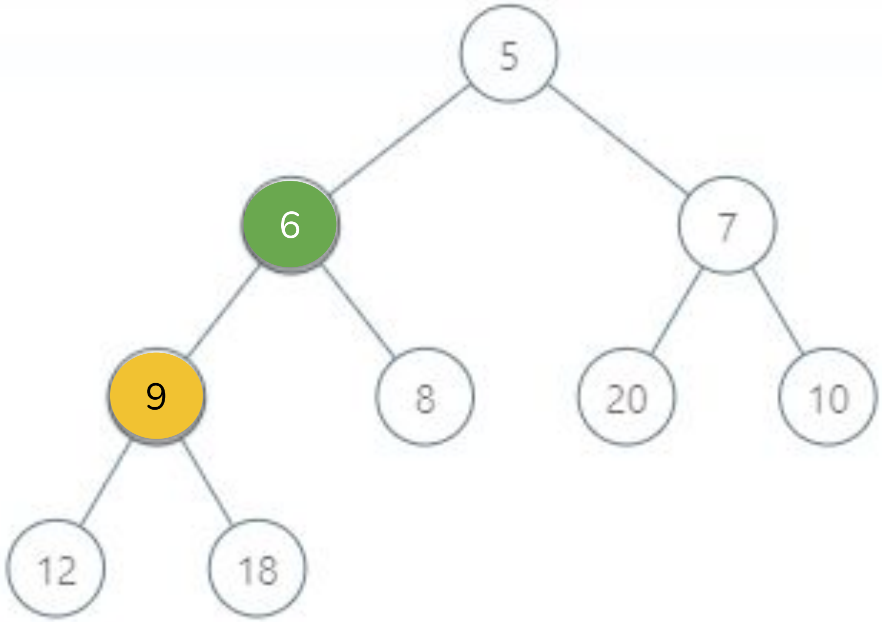


Removal & Down-heap Bubbling Simulation

Our current node, still **violates** heap order. Thus, promote the **smaller child**

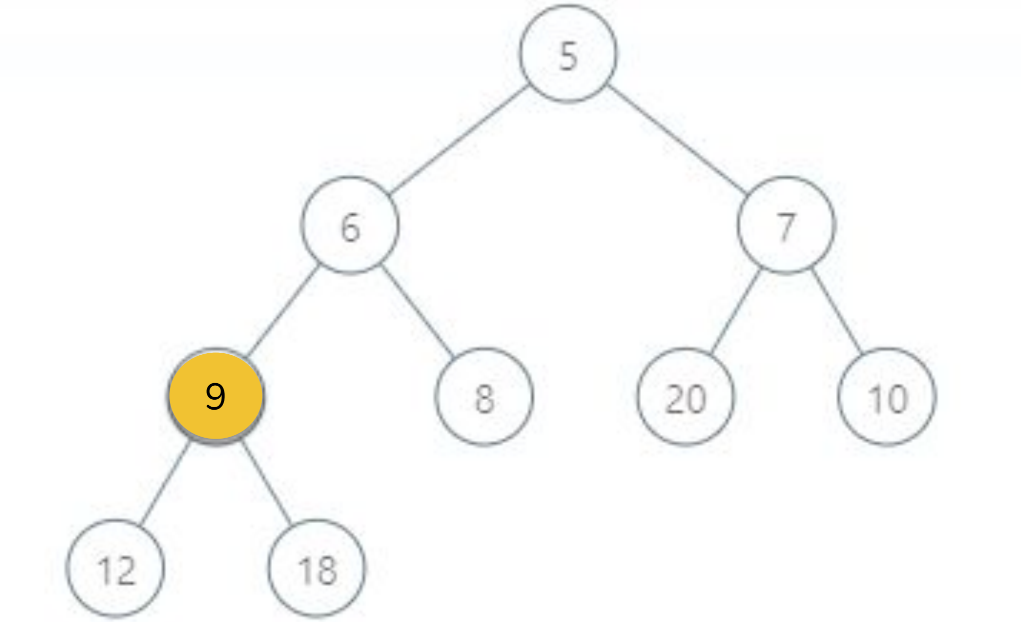


Removal & Down-heap Bubbling Simulation



Removal & Down-heap Bubbling Simulation

Our current node, **does not** violate heap order. **We are done!**



Time & Space Complexity

Heapify down

- Time complexity - ____
- Space complexity - ____

Time & Space Complexity

Heapify down

- Time complexity - **O(log n)**
- Space complexity - **O(1)**

Why?

Can you implement heapify down?

[Playground]

Solution

```
def heapify_down(heap, n, current_idx):
    small_child_idx = current_idx
    left_idx = 2 * current_idx + 1
    right_idx = 2 * current_idx + 2

    if left_idx < n and heap[left_idx] < heap[small_child_idx]:
        small_child_idx = left_idx

    if right_idx < n and heap[right_idx] < heap[small_child_idx]:
        small_child_idx = right_idx

    if small_child_idx != current_idx:
        swap(heap, current_idx, small_child_idx)
        heapify_down(heap, n, small_child_idx)
```

Heap Construction

Method 1

Start with empty heap and insert one element at a time

Time Complexity

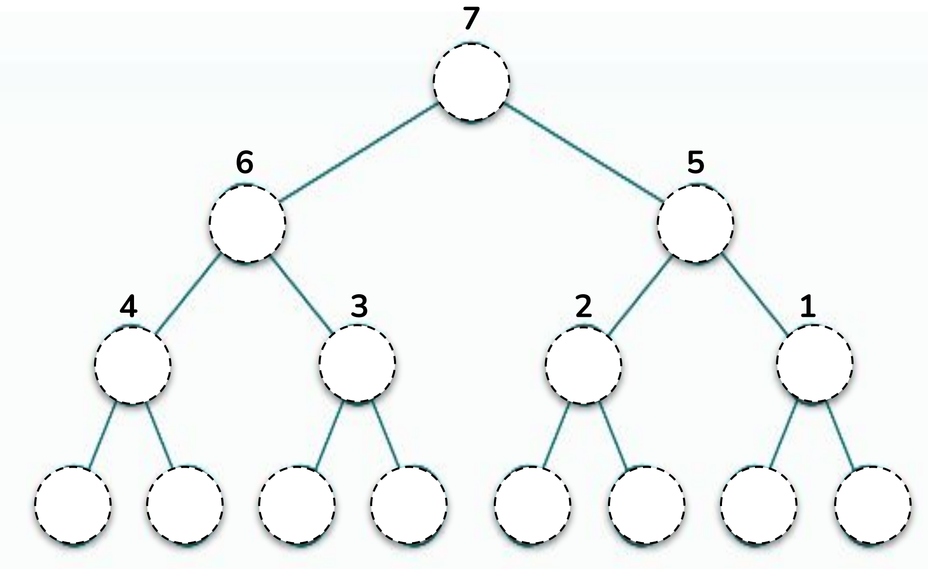
- Time complexity - **O(n log n)**

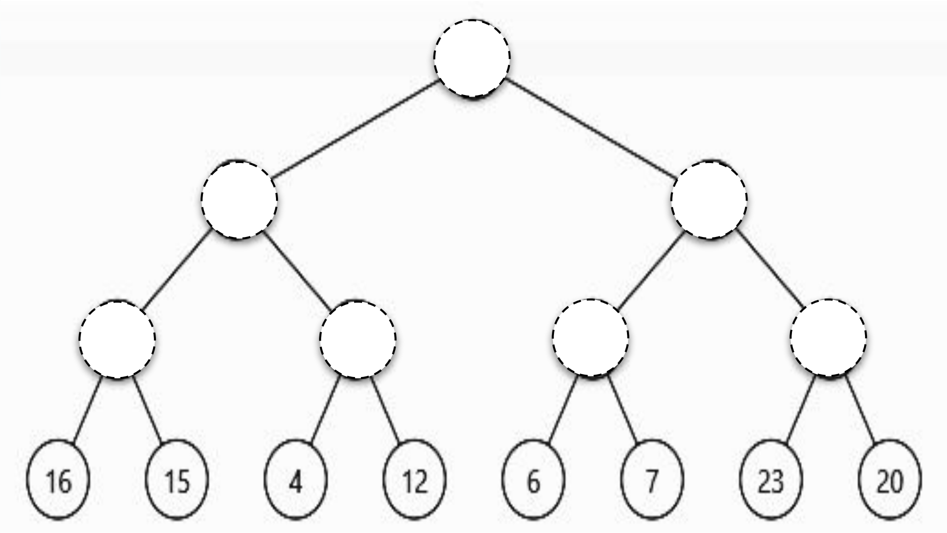
A2SV

Method 2 - Bottom up construction

- We start with the last non-leaf node.
- We then apply the heapify_down algorithm to this node,
- We then move to the previous non-leaf node and apply the heapify_down algorithm to it as well.
- We continue this process, moving up the tree in reverse order, until we reach the root node.

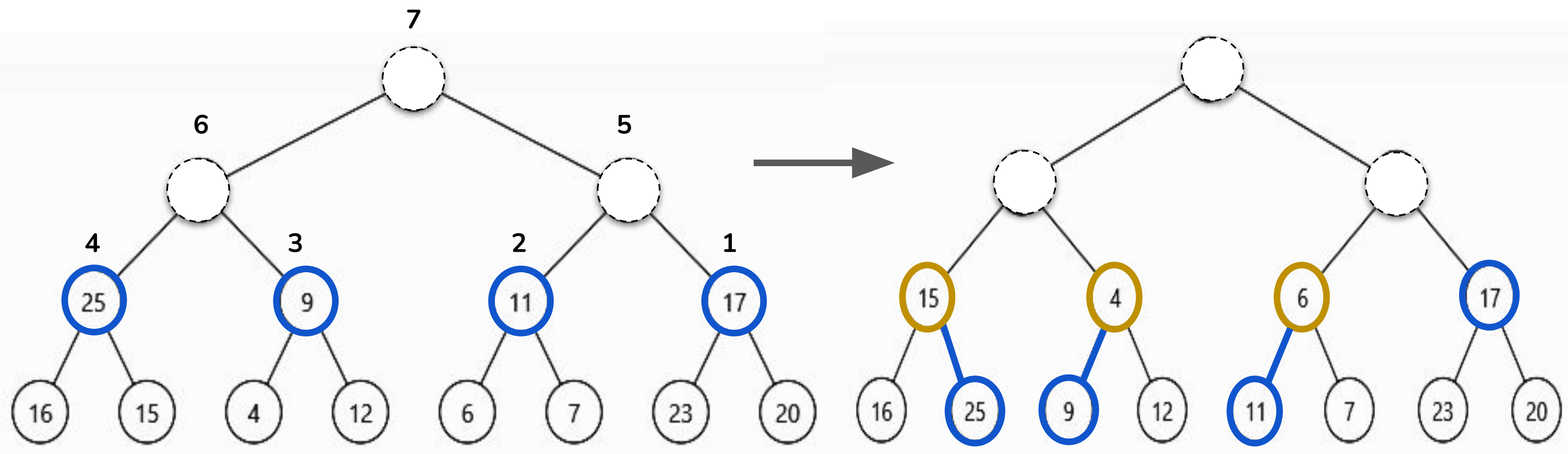
Bottom-up construction simulation



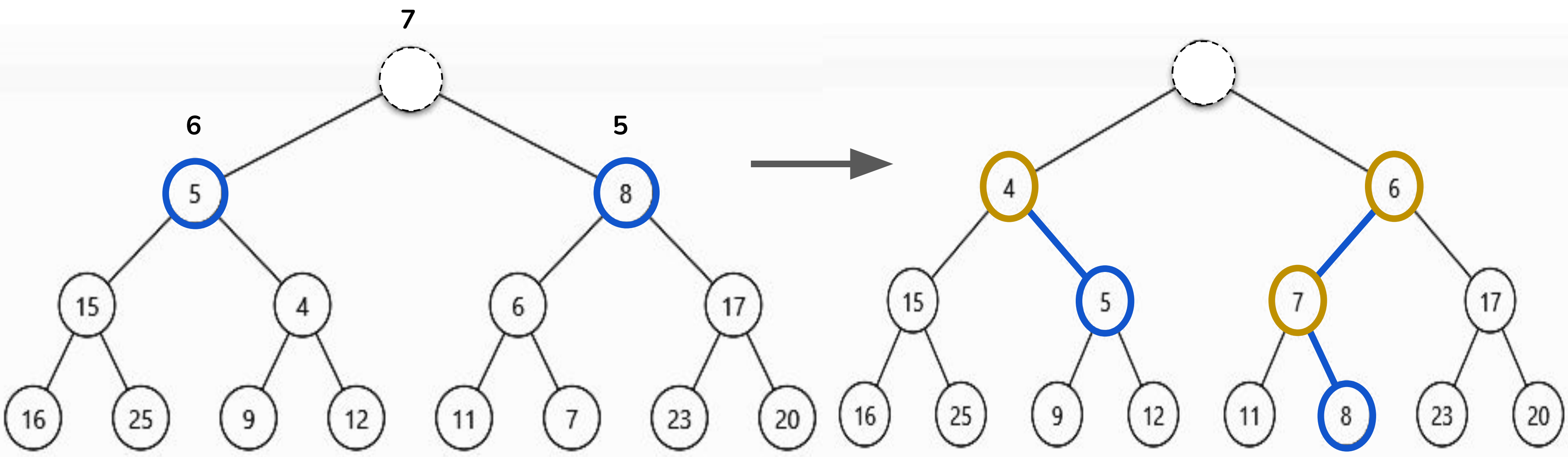


The sequence of numbers from 1 to 7 denotes the order in which we should invoke the heapify_down operation, **i.e starting from the last non-leaf node.**

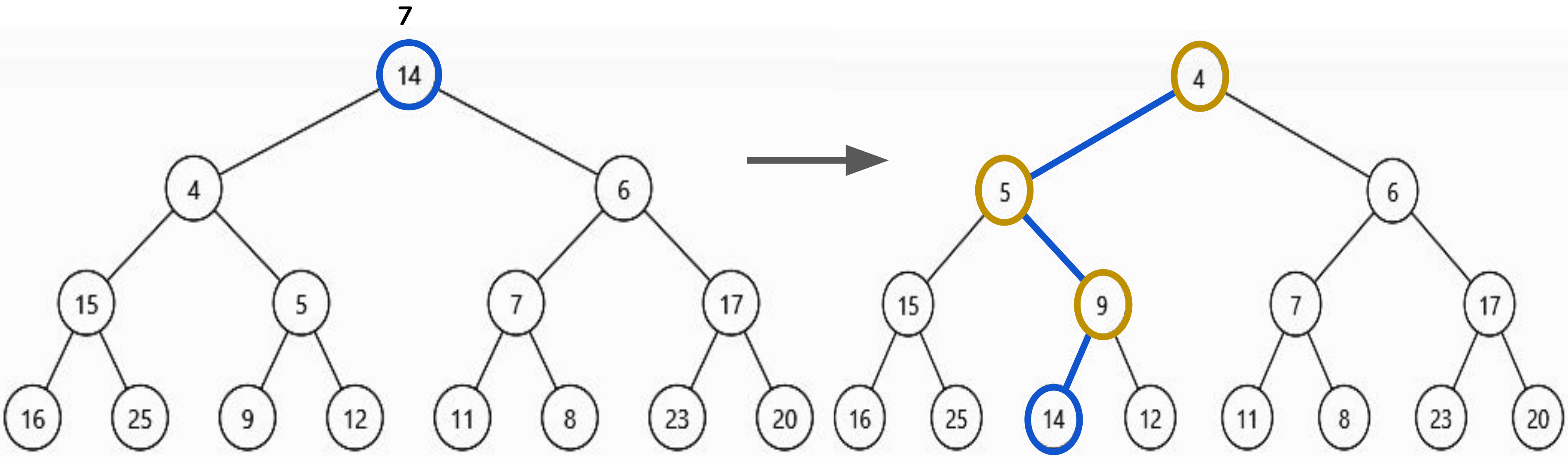
Bottom-up construction simulation



Bottom-up construction simulation



Bottom-up construction simulation



Time & Space Complexity

- Time complexity - ____
- Space complexity - ____

Time & Space Complexity

- Time complexity - **O(n)**
- Space complexity - **O(1)**

Why?

Time complexity proof

Can you implement heapify?

```python
def heapify(array):
    # your code goes here
```

Solution

```python
def heapify(array):
    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify_down(arr, n, i)
```

Heap Sort

What is Heap Sort?

Heap sort is a **comparison-based** sorting technique based on binary heap data structure.

Steps:

1. We first find the maximum element and place it at the end.
2. Repeat the same process for the remaining elements.

Implement heap sort

Can you implement heap sort in place?

Heap sort simulation

Let’s sort the following array,

[1, 4, 10, 3, 5, 1]

Heapify (max)

[10, 5, 1, 3, 4, 1]

Heap sort simulation

Swapping max element (root) with last element

10	5	1	3	4	1

Heap sort simulation

But don’t pop it from the list

1	5	1	3	4	10

Heap sort simulation

heapify_down on the root element but our heap is now just the first **n - 1** elements

1	5	1	3	4	10

And that is the use of **n** in our **heapify_down** implementation

```
def heapify_down(heap, n, current_idx):
```

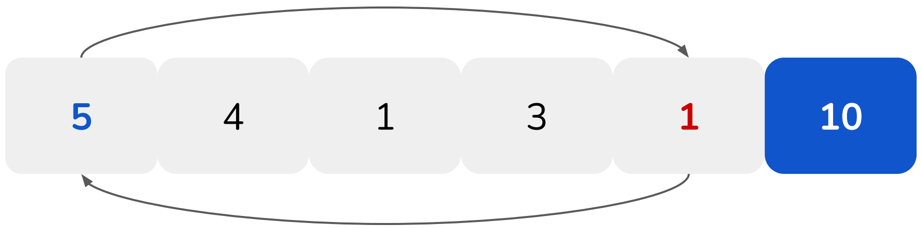
Heap sort simulation

After heapifying the **n - 1** elements

5	4	1	3	1	**10**

Heap sort simulation

Swap max element (root) with last element



Heap sort simulation

heapify_down on the root element but our heap now has **n - 2** elements

1	4	1	3	5	10

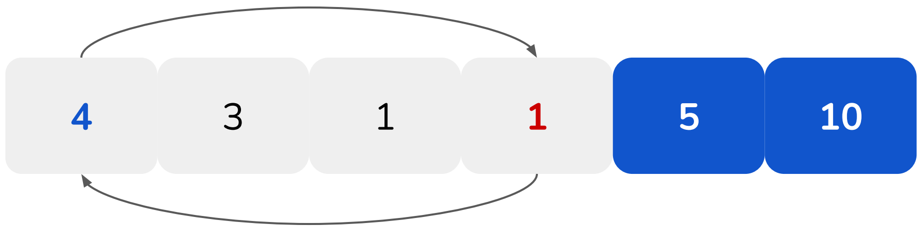
Heap sort simulation

After heapifying the **n - 2** elements

4	3	1	1	5	10

Heap sort simulation

Swap



Heap sort simulation

Then heapify the **n - 3** elements

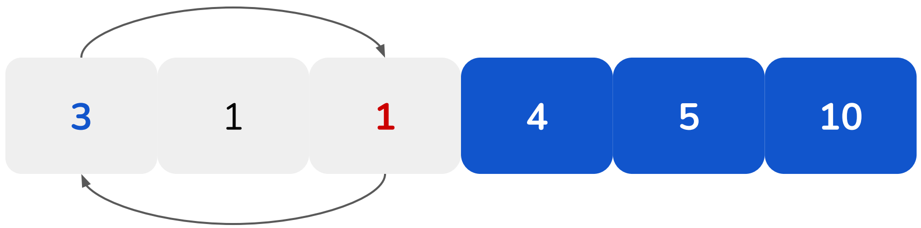
1	3	1	4	5	10

Heap sort simulation

3	1	1	4	5	10

Heap sort simulation

Swap



Heap sort simulation

Then heapify **n - 4** elements and swap

1	1	3	4	5	10

Heap sort simulation

Then heapify **n - 5** elements and swap

1	1	3	4	5	10

Heap sort simulation

We are done!

1	1	3	4	5	10

Time & Space Complexity

- Time complexity - **O(n log n)**
- Space complexity - **O(1)**

Question

Advantages of heap sort

- Efficiency - O(n log n)
- Memory usage - O(1)
- Simplicity
- Can be used in hybrid algorithms like the IntroSort.
- Sorting a nearly sorted (or K sorted) array.
- Finding K-largest (or smallest) elements in an array.

Disadvantage of heap sort

- Unstable
- Scalability - not very efficient when working with highly complex data.

Heapq Module

Heapq Module

The Python **heapq module** offers an implementation of the operations we covered for a **min heap**, and some more functionalities.

- **heapify(iterable)**: this function transforms the iterable into a heap in-place
- **heappush(heap, item)**: this function pushes the item onto the heap while maintaining the heap invariant.
- **heappop(heap)**: This function pops and returns the smallest item from the heap while maintaining the heap invariant.

Heapq Module

- **heapreplace(heap, item):** this function pops and returns the smallest item from the heap, and then pushes the new item onto the heap. It is more efficient than calling **heappop()** followed by **heappush()**

- **heappushpop(heap, item):** this function combines the functionality of heappush() and heappop(), pushing the item onto the heap and then popping and returning the smallest item. It can be more efficient than **heappush()** followed by **heappop()**

Heapq Module

- **nlargest(n, iterable)**: this function returns the n-largest elements from the iterable.
- **nsmallest(n, iterable)**: this function returns the n-smallest elements from the iterable.

Problem Patterns

A. Top K Pattern
B. Merge K Sorted Pattern
C. Two Heap Pattern

Top K Pattern

A common problem that involves finding the k largest or smallest elements from a collection of n elements

Question

Merge K Sorted Pattern

A common problem that involves merging k sorted arrays or lists into a single sorted array or list.

Question

Two Heaps Pattern

Solving problems that require

- Quick access to the smallest and largest elements
- Maintaining a fixed-size window of elements with efficient insertion and removal operations.

Question

Real World Applications

- Heap sort
- Priority Queue
  - Process scheduling in operating systems
- Graph algorithms
  - Shortest path
  - Minimum spanning tree

Common Pitfalls

- Popping from empty heap

```python
heapq.heappop(heap)
```

Bad

```python
if heap:
    heapq.heappop(heap)
```

Good

- Appending to a full heap (fixed size)

```python
if len(heap) < capacity:
    heapq.heappush(heap, item)
```

Check that we haven’t surpassed the required capacity

Practice Questions

Last Stone Weight

Kth Largest Element in a Stream

Kth Largest Element in an Array

Kth Smallest Element in a Sorted Matrix

Single-Threaded CPU

Find K Pairs with Smallest Sums

Top K Frequent Elements

Top K Frequent Words

Furthest Building You Can Reach

Find Median from Data Stream

Minimum Cost to Hire K Workers

Merge k Sorted Lists

The Skyline Problem

Heap Operations

Actions Express Priorities!

Mahatma Gandhi