7 Heaps and Priority Queues

For COMSC 132

Sam Bowne

Oct 6, 2024

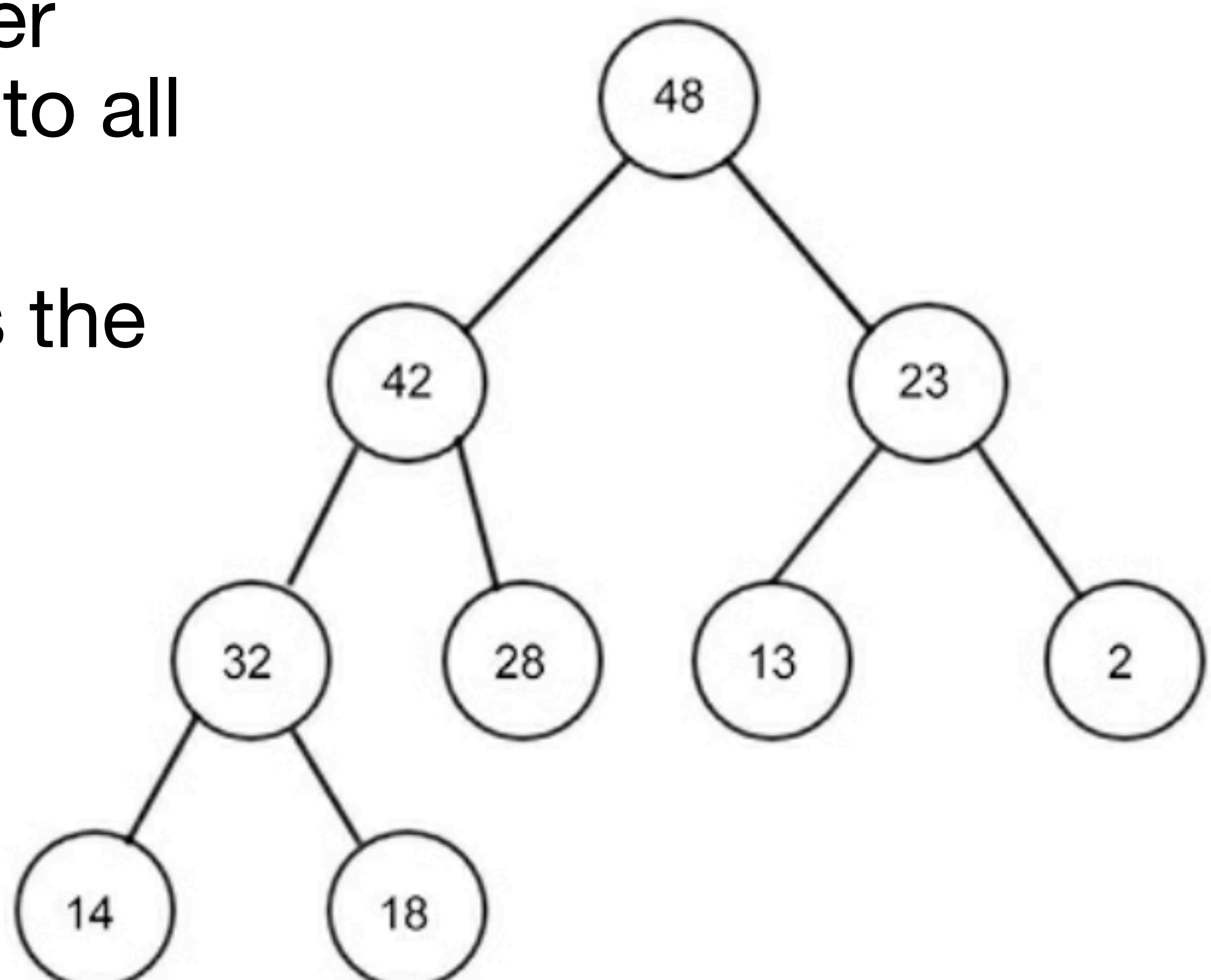
Heaps

Heaps

- A specialization of a tree
- Nodes are ordered, with a **heap property**
- Two types
  - **max heap**
  - **min heap**

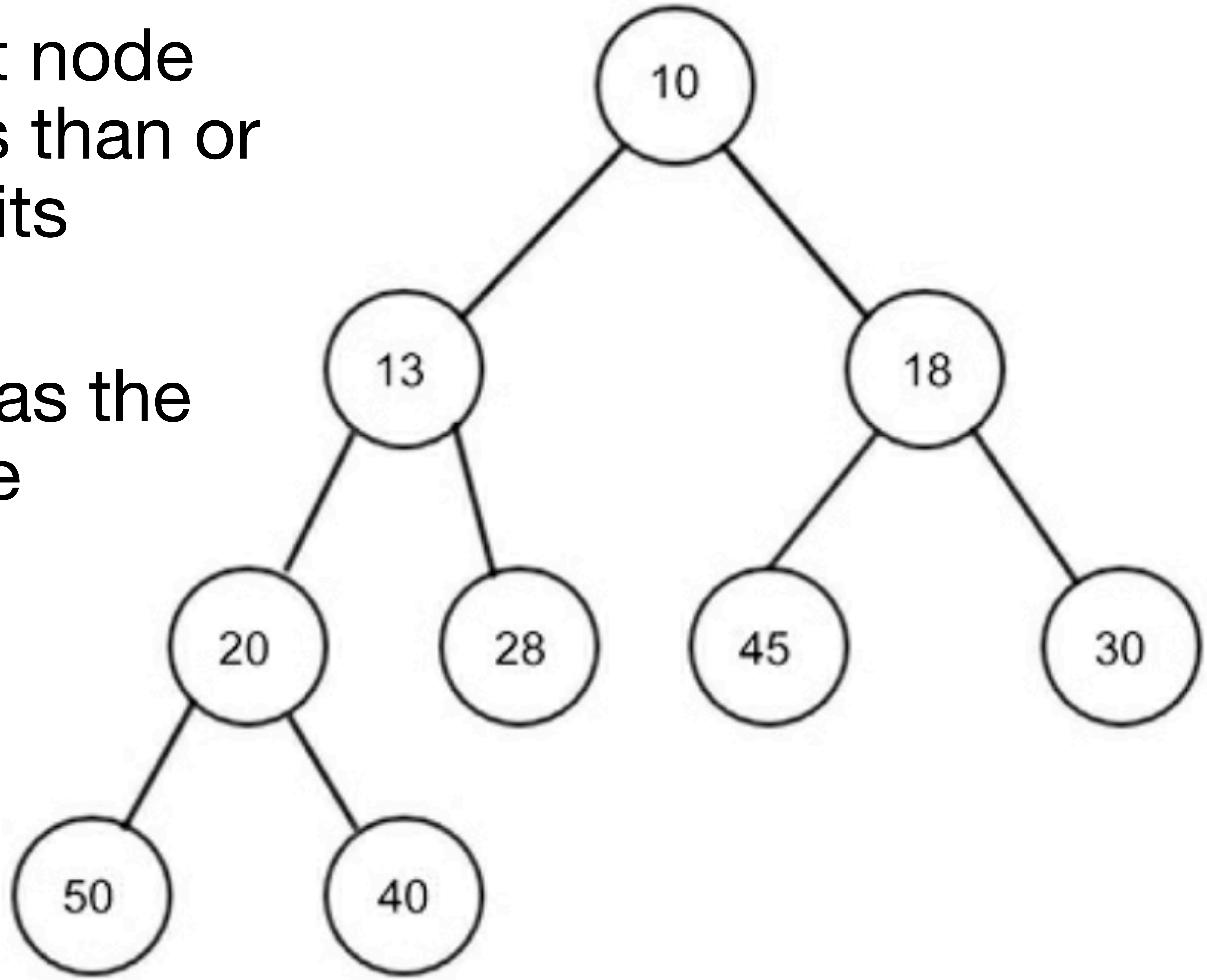
max heap

- Each parent node value is greater than or equal to all its children
- root node has the highest value



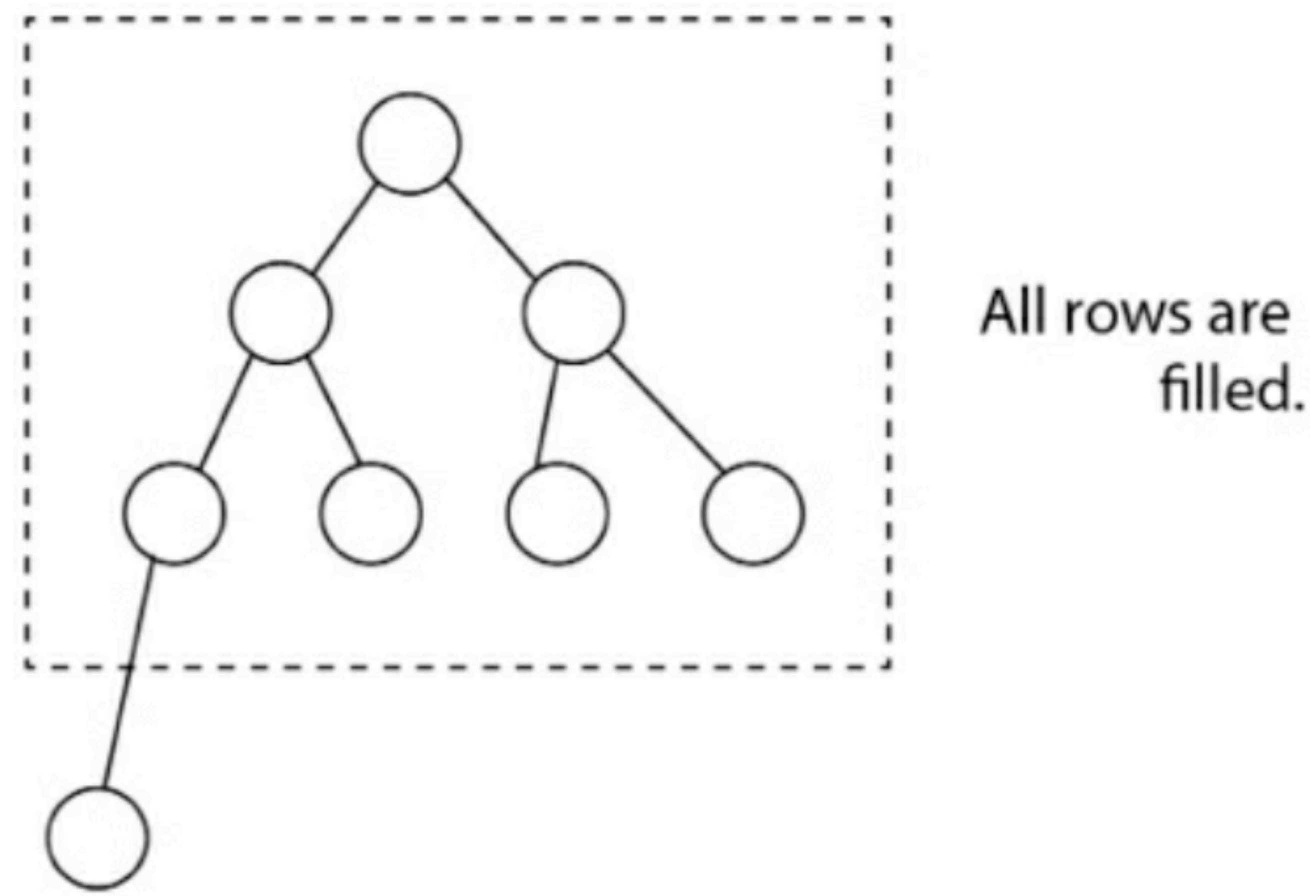
min heap

- Each parent node value is less than or equal to all its children
- root node has the lowest value



Binary Trees

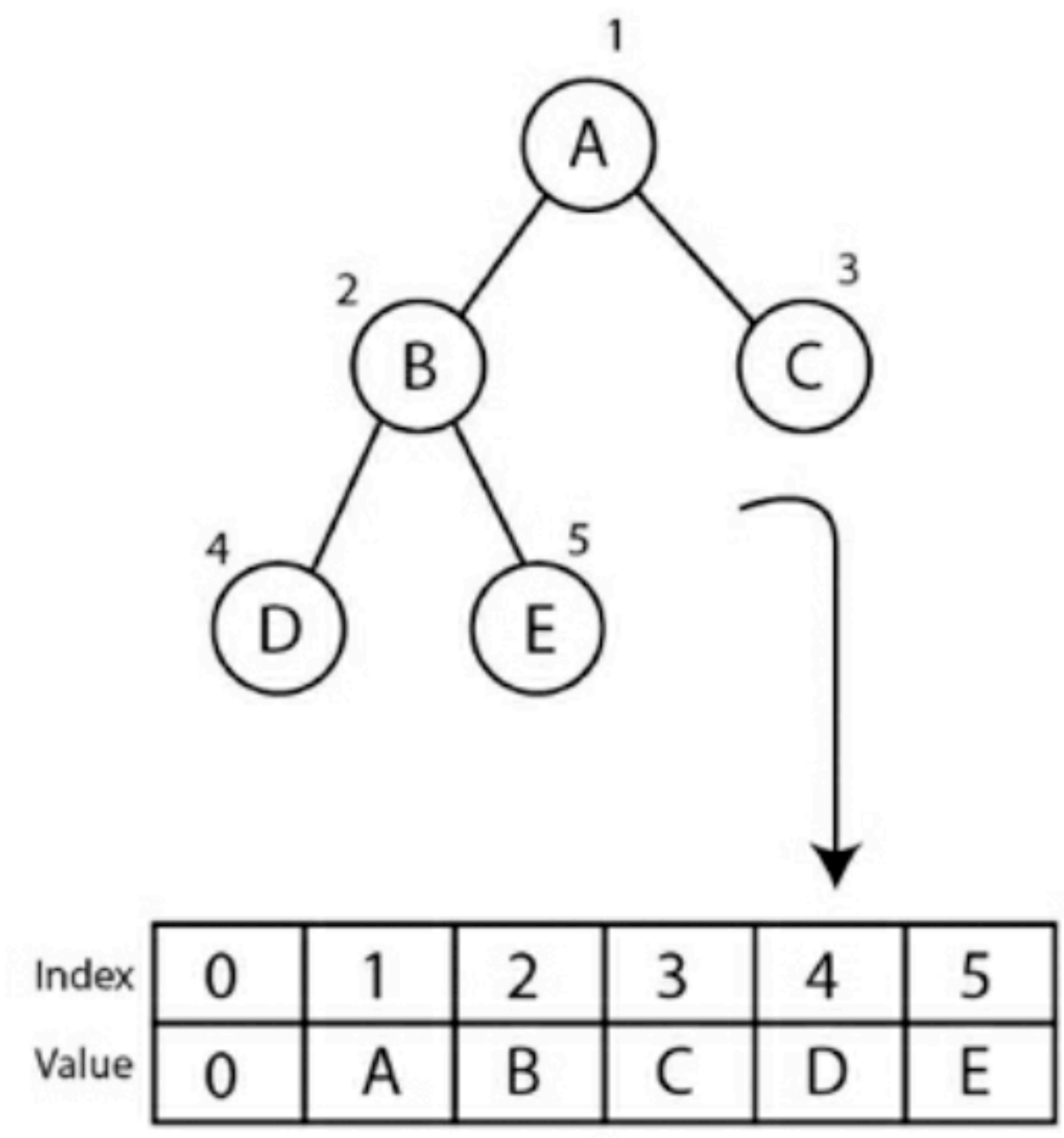
- A heap can be any kind of tree
- Most commonly, it's a binary tree
- A **complete binary tree** fills each row before starting the next one



Index positions

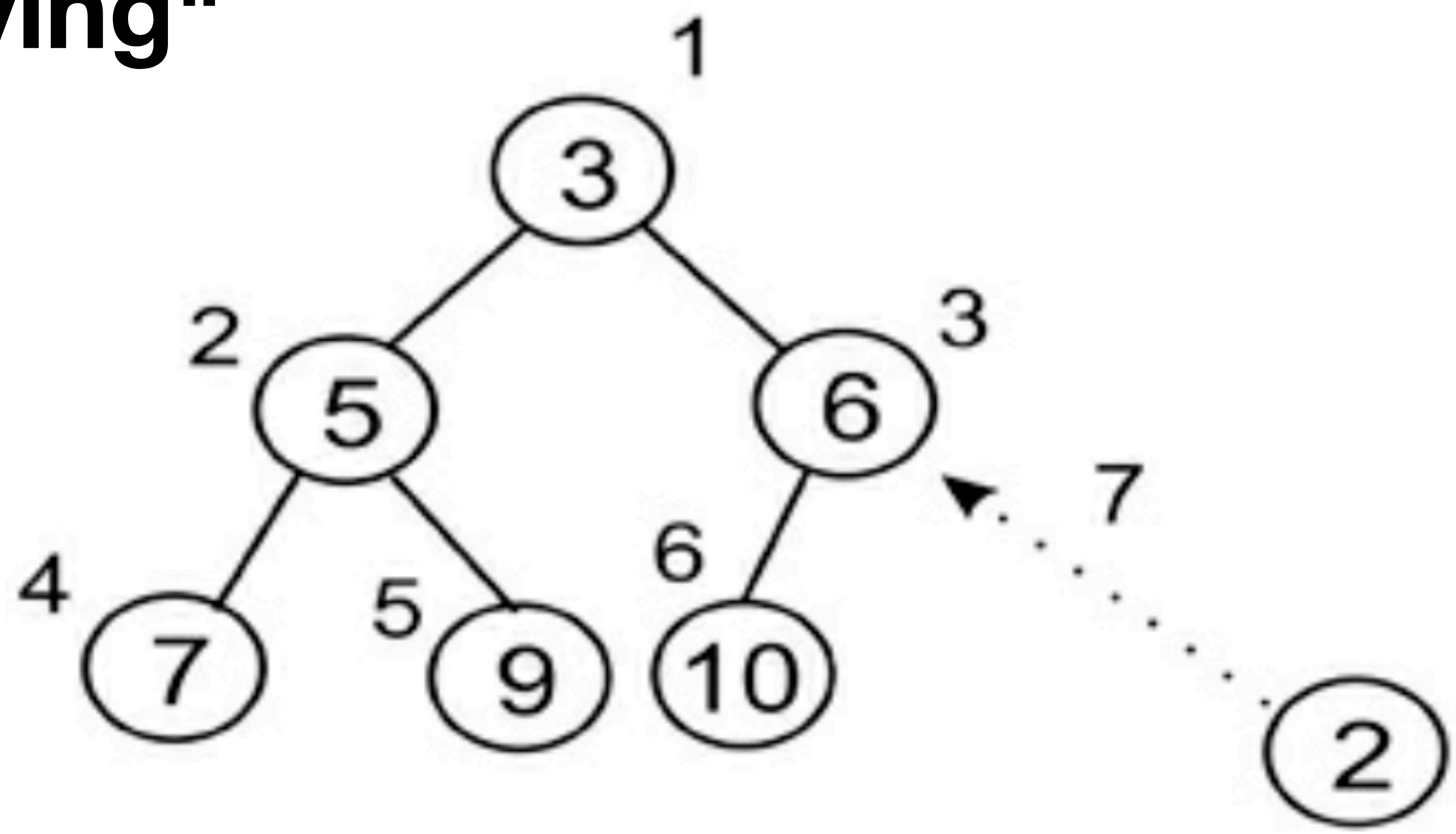
- The children of node at index ***n***
  - Left child at index ***2n***
  - Right child at index **2*n*+1**
- Dummy 0 at index 0

```
class MinHeap:
    def __init__(self):
        self.heap = [0]
        self.size = 0
```



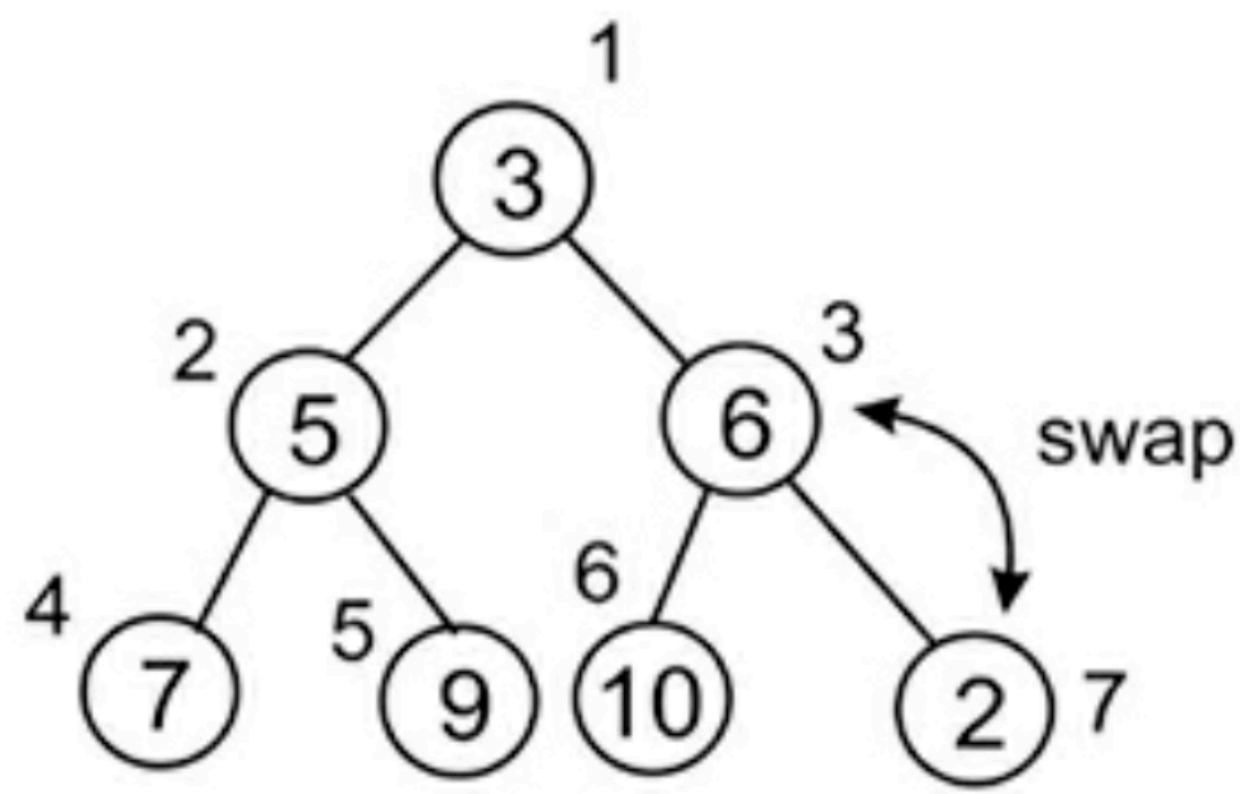
Inserting into a heap

- Add new element at the end
- Then rearrange the nodes to restore the **heap property**
  - "**heapifying**"



Inserting into a heap

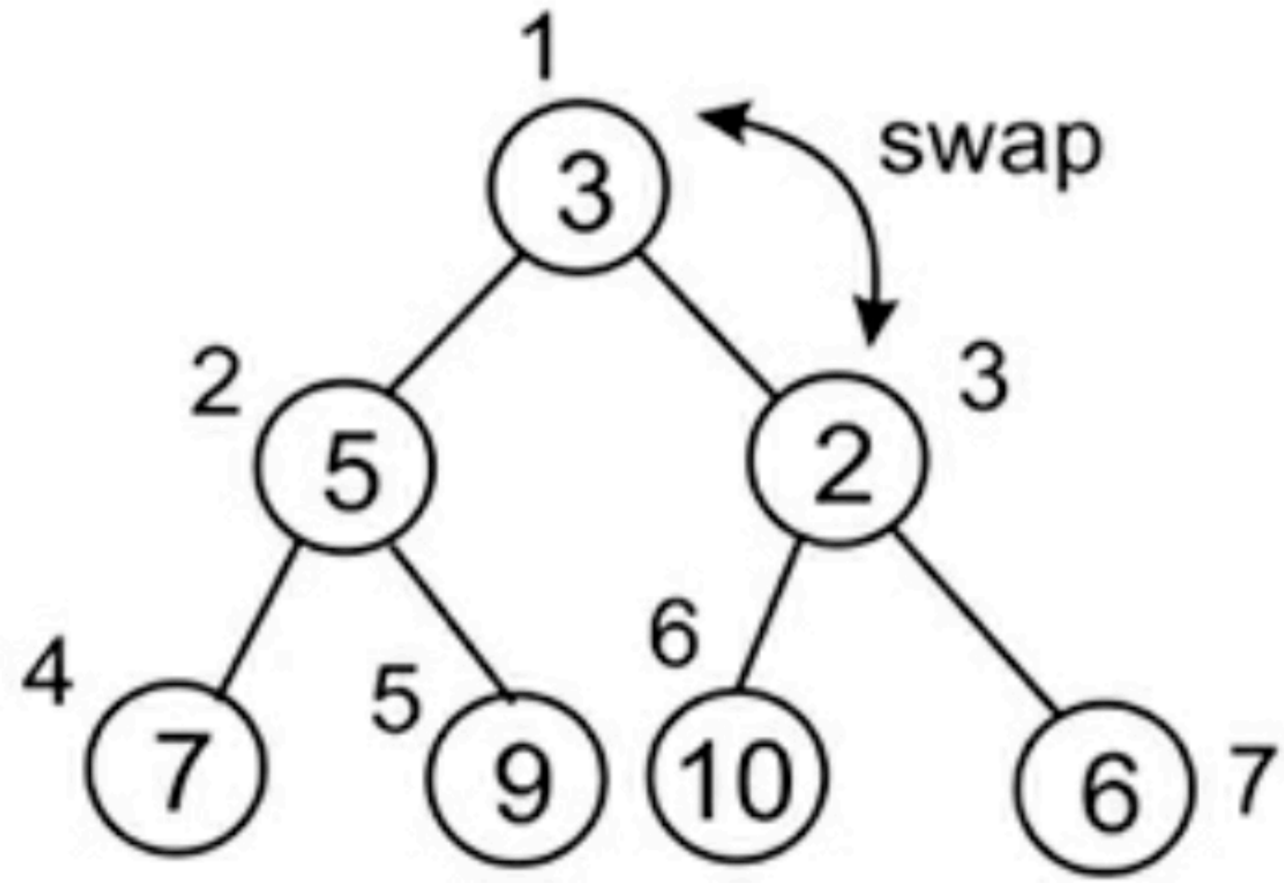
- Compare new node to its parent
- Swap if necessary

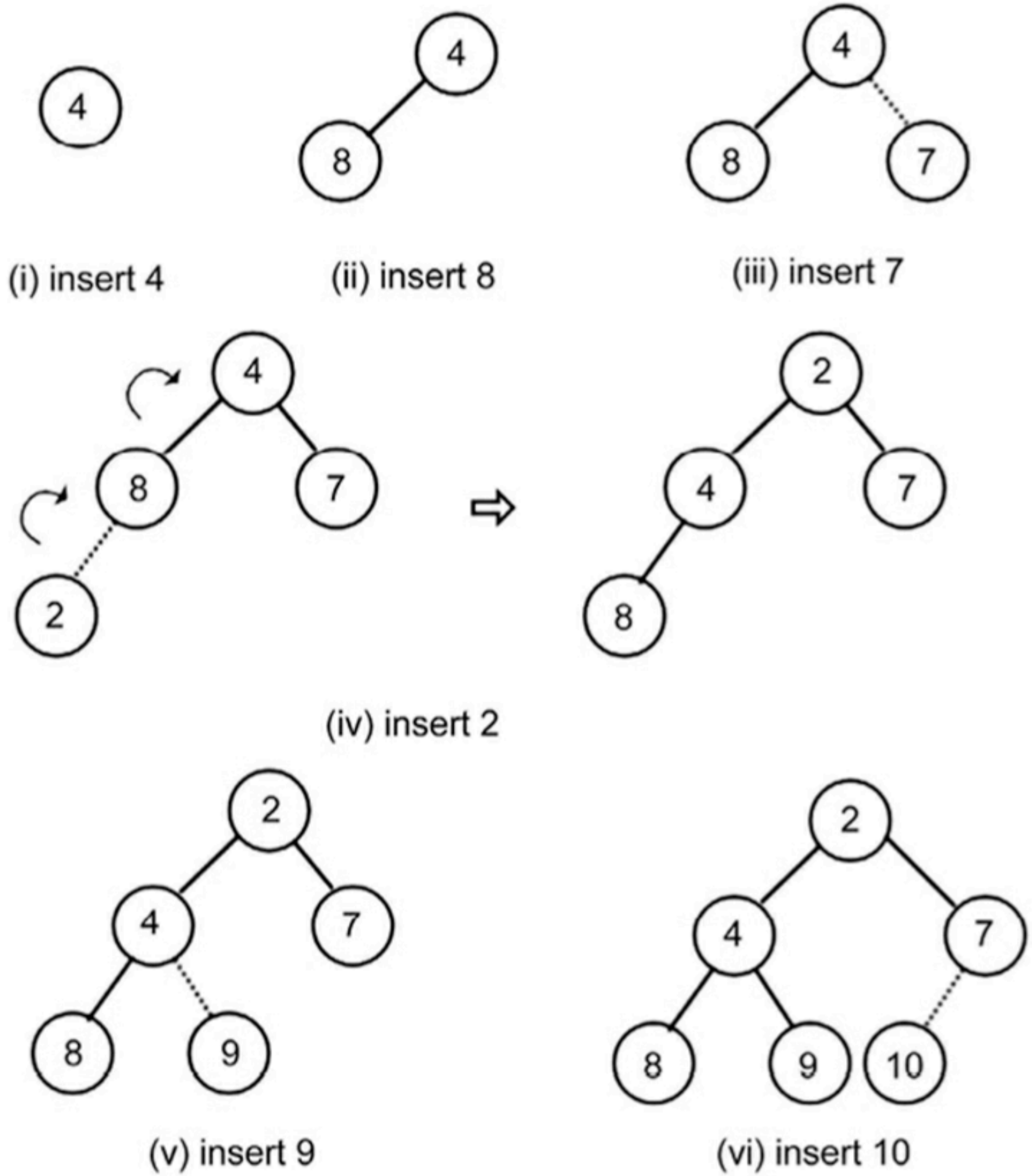


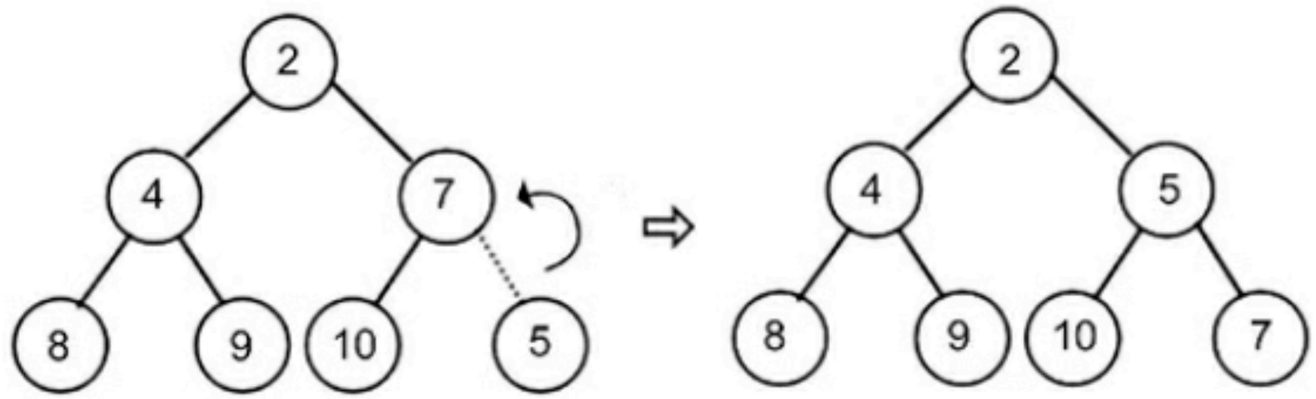
Inserting into a heap

- Repeat the compare-and-swap operation
- Complexity *O*(**log *n***)

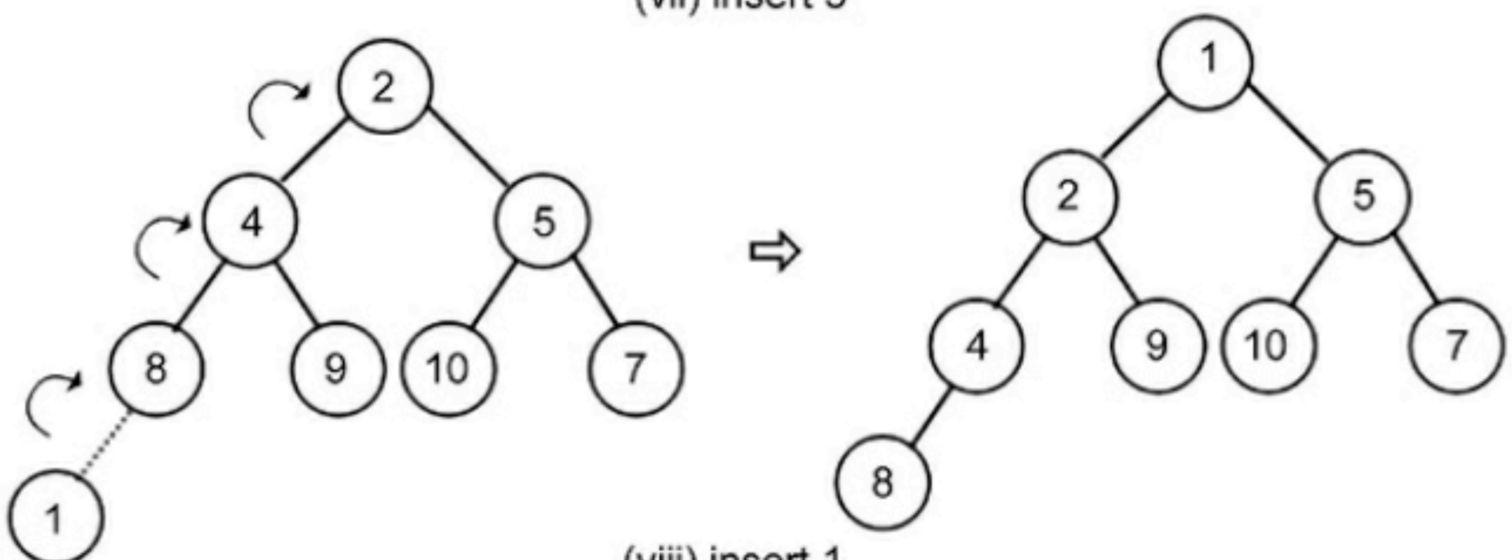
- On following slides, we'll build a heap by inserting these values:
- 4, 8, 7, 2, 9, 10, 5, 1, 3, 6



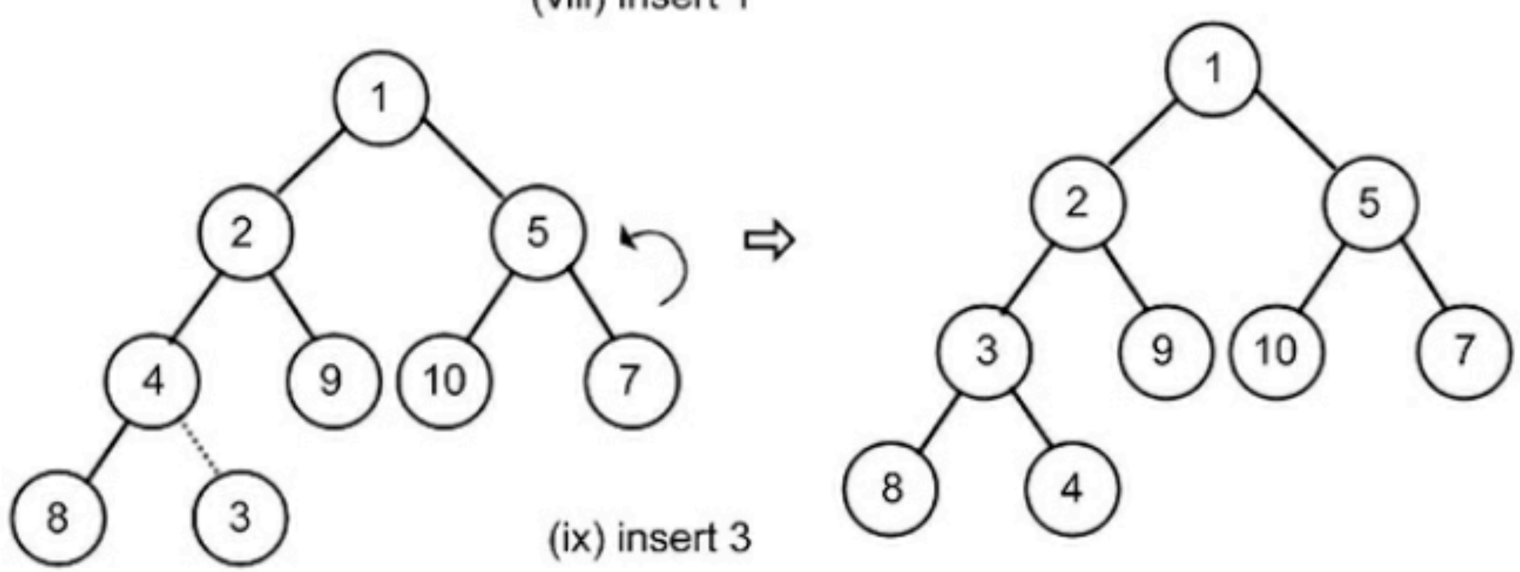

(i) insert 4
(ii) insert 8
(iii) insert 7
(iv) insert 2
(v) insert 9
(vi) insert 10



(vii) insert 5



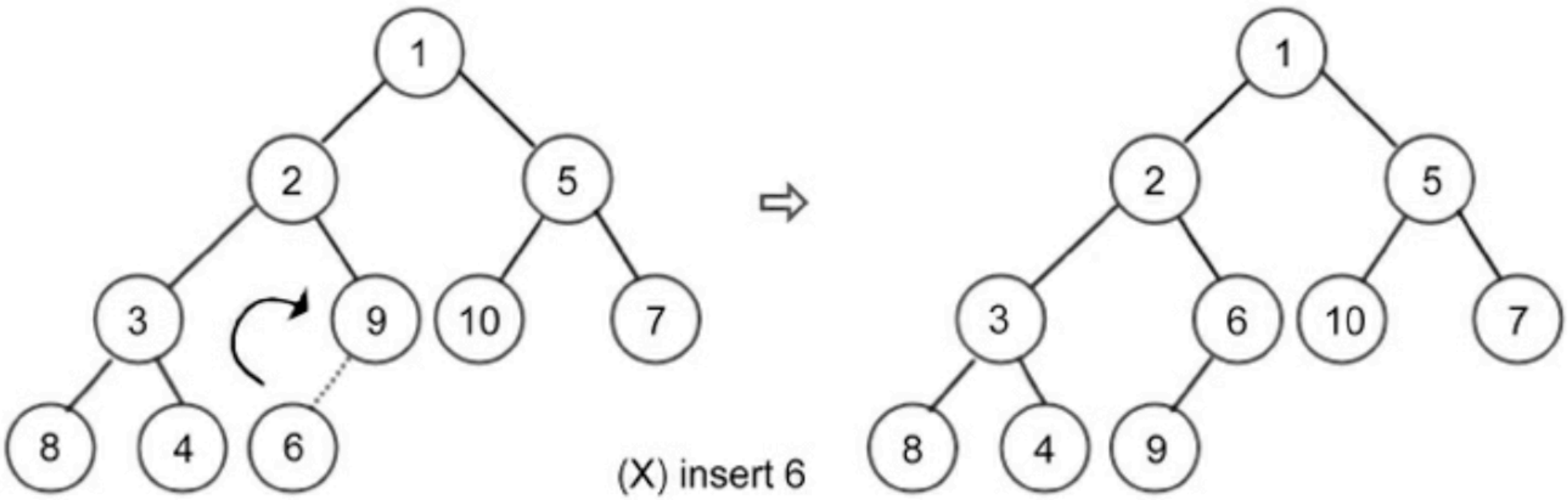
(viii) insert 1



(ix) insert 3

Last step

```python
def insert(self, item):
    self.heap.append(item)
    self.size += 1
    self.arrange(self.size)
```

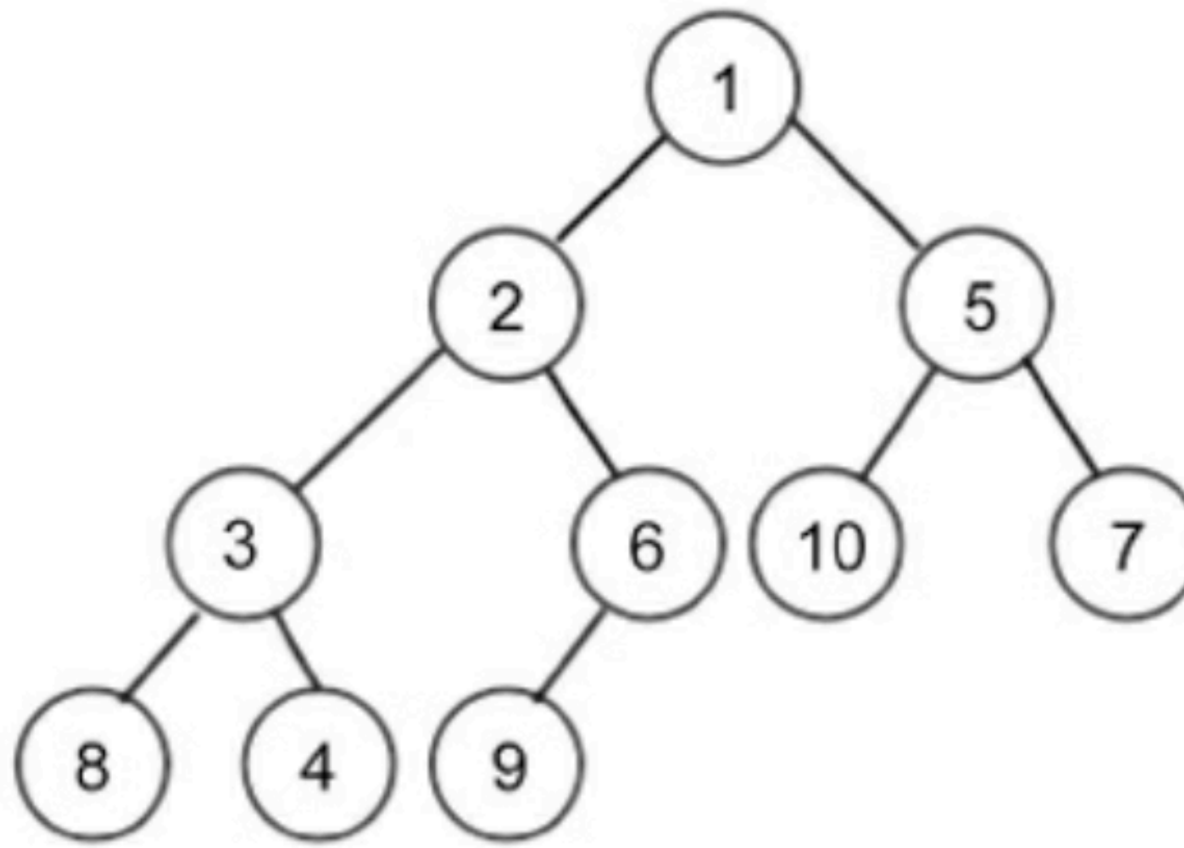



```python
def arrange(self, k):
    while k // 2 > 0:
        if self.heap[k] < self.heap[k//2]:
            self.heap[k], self.heap[k//2] = self.heap[k//2], self.heap[k]
        k //= 2
```

Building the heap

- Code to build the heap from the last few slides
- Complexity appears to be *O(**n** log **n**)* *
- The resulting list is not simply sorted
- * see next slide

```python
h = MinHeap()
for i in (4, 8, 7, 2, 9, 10, 5, 1, 3, 6):
    h.insert(i)
```



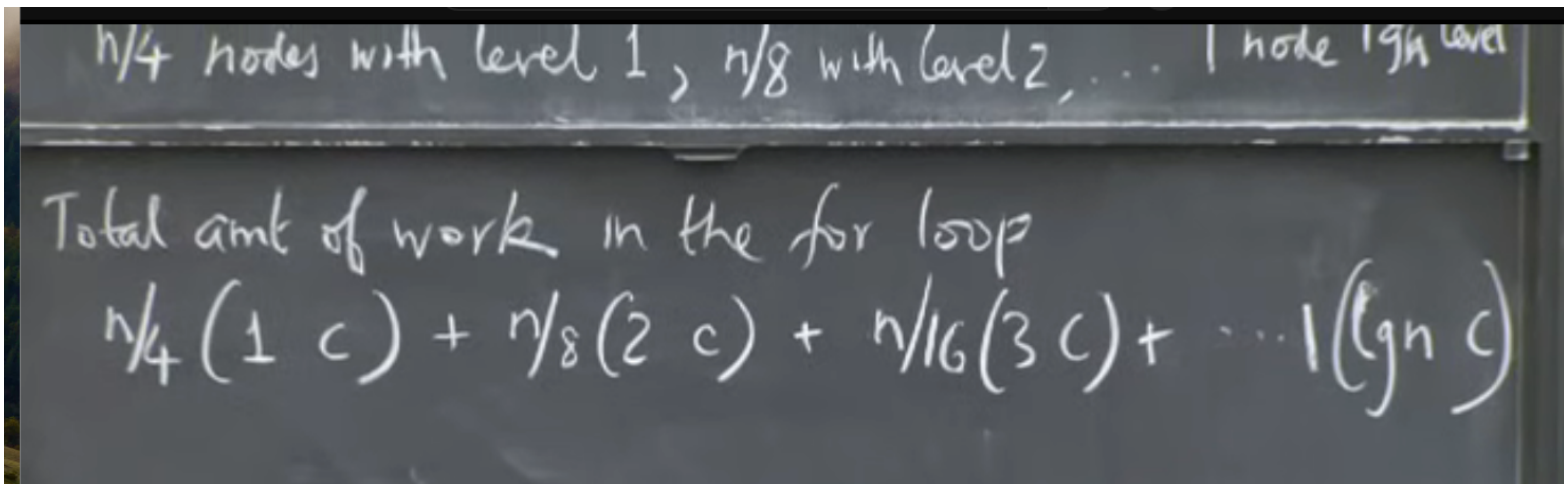

```
[0, 1, 2, 5, 3, 6, 10, 7, 8, 4, 9]
```

Heap build complexity

- We do ***n*** inserts: *O(**n**)*
- After each one, we heapify: *O(*log ***n****)*
- So complexity *O(**n*** log ***n****)*
  - This is an upper bound
- It's actually *O(**n**)* as explained in this video
  - https://youtu.be/B7hVxCmfPtM?t=2082
- Because most nodes stay near the bottom of the tree
  - So heapifying *from the bottom* is really of *O(*1*)*

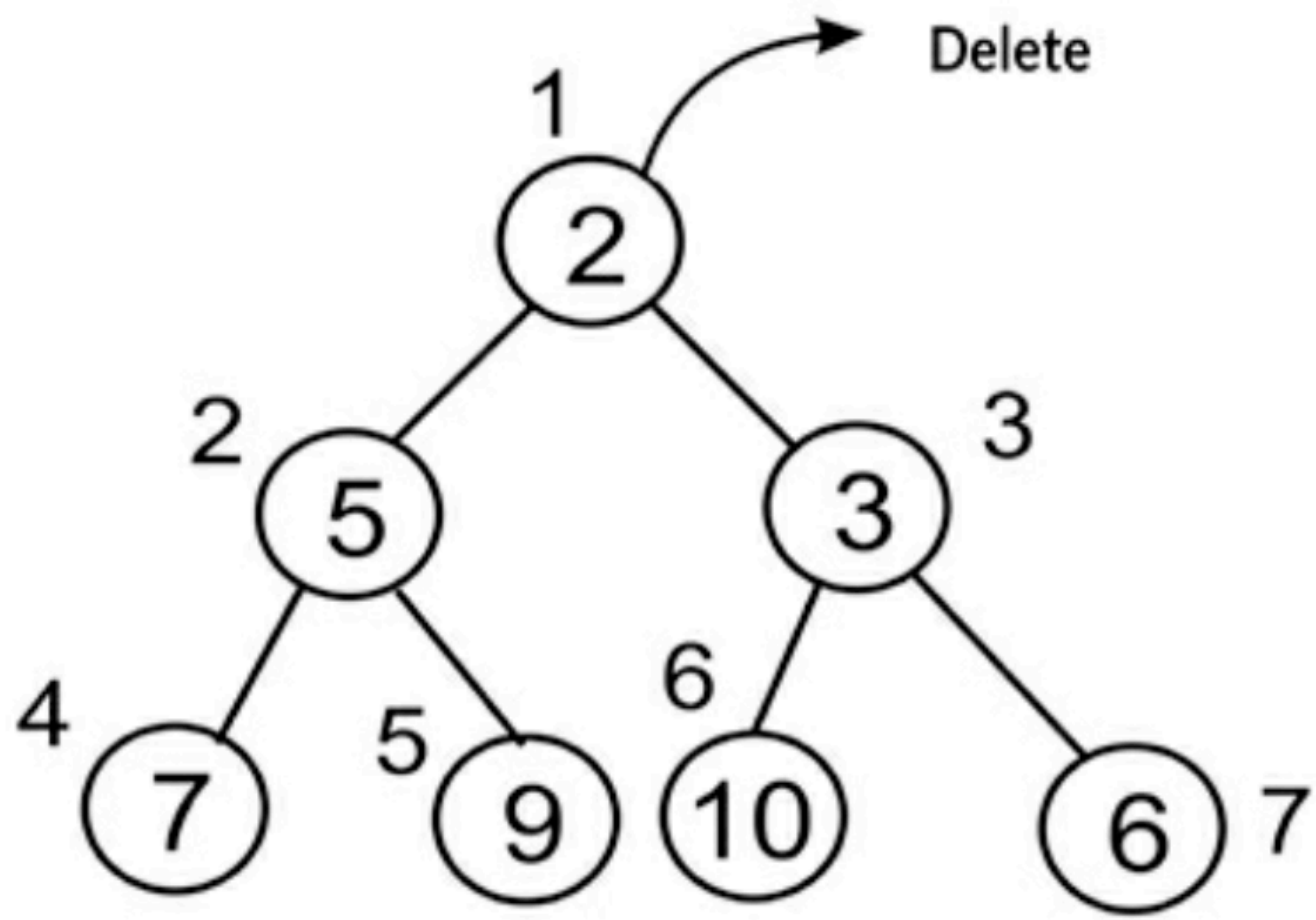
Heap build complexity

- It's actually *O(**n**)* as explained in this video
- https://youtu.be/B7hVxCmfPtM?t=2082



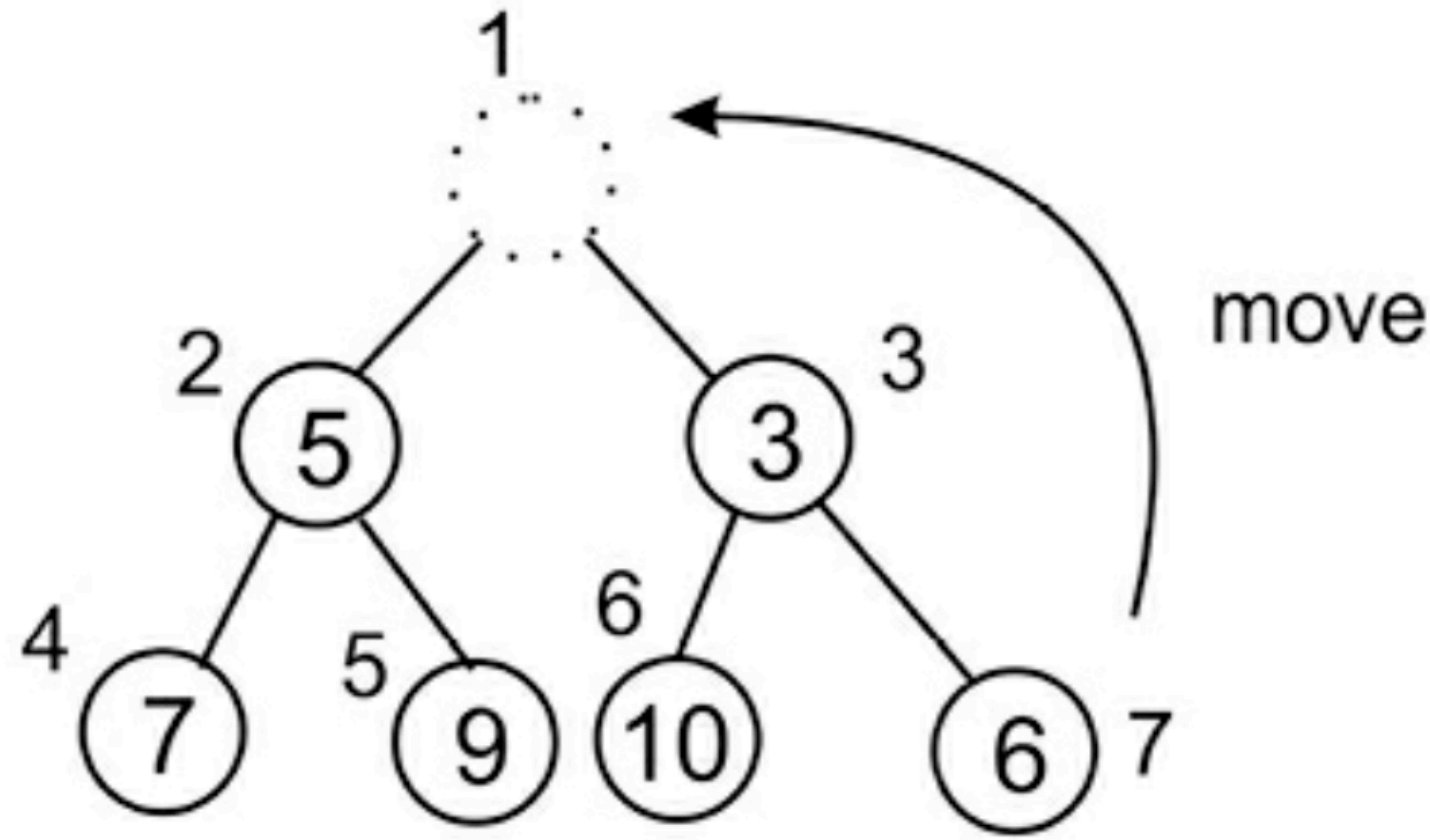
Delete operation

- Most often, we delete the root
  - To find min or max



Delete operation

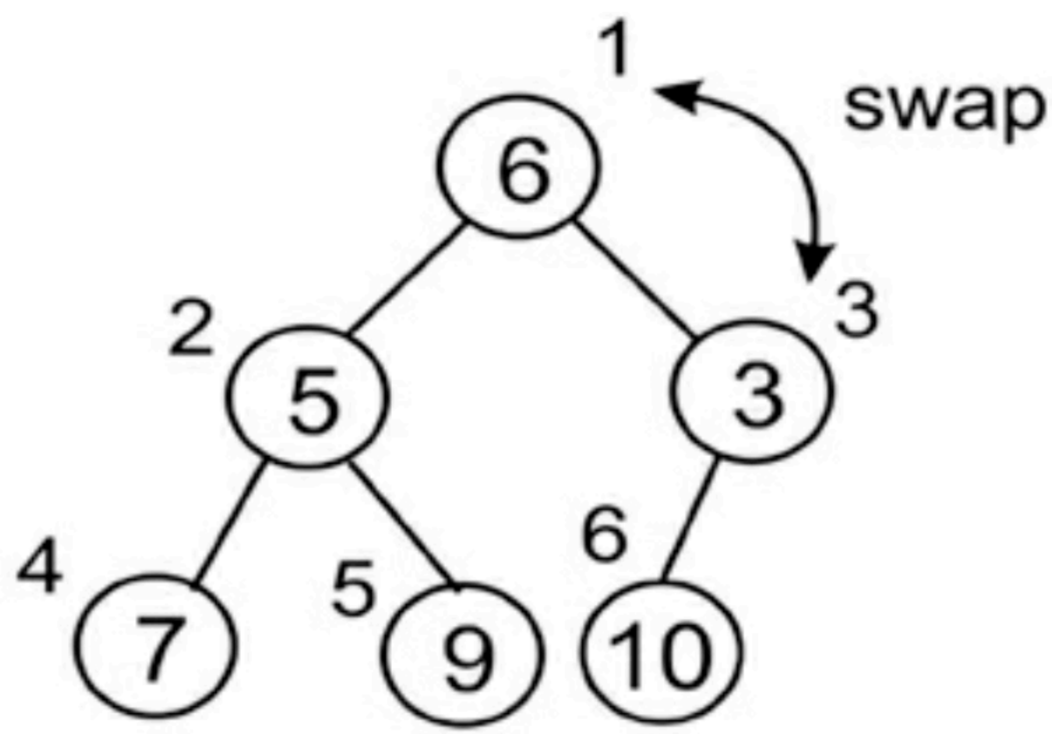
- Move last element to the root

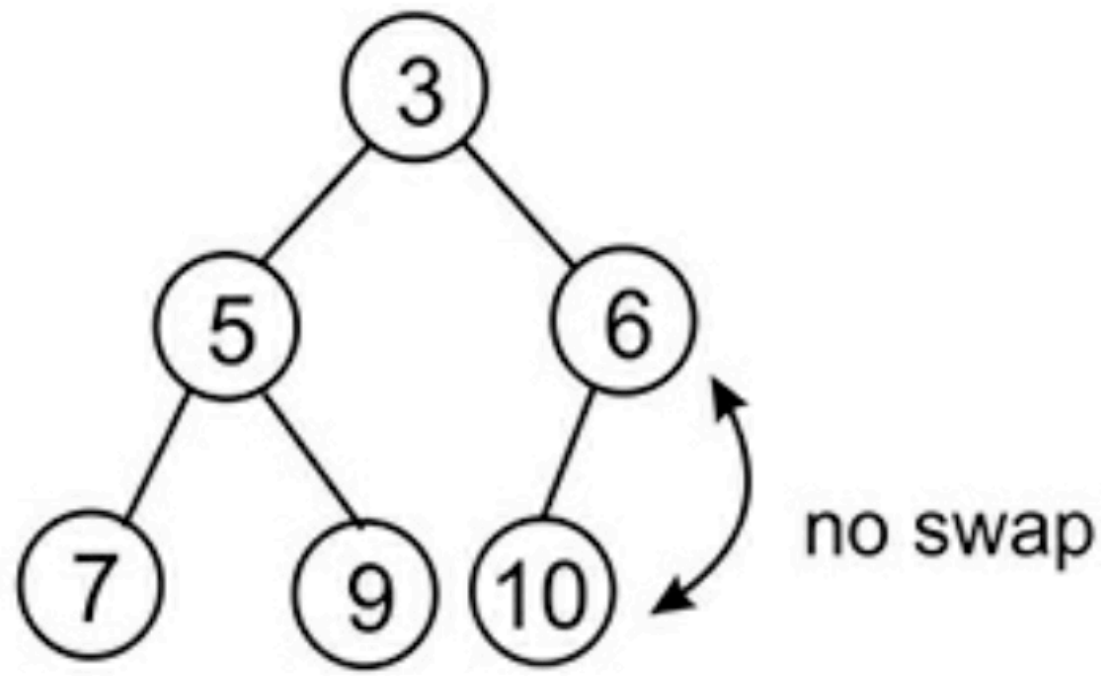


Delete operation

- Heapify the tree

```
def minchild(self, k):
    if k * 2 + 1 > self.size:
        return k * 2
    elif self.heap[k*2] < self.heap[k*2+1]:
        return k * 2
    else:
        return k * 2 + 1
```






```
def sink(self, k):
        while k * 2 <= self.size:
            mc = self.minchild(k)
            if self.heap[k] > self.heap[mc]:
                self.heap[k], self.heap[mc] = self.heap[mc], self.heap[k]
            k = mc
```

Delete at root code

- Shrinks heap by one
- Returns value of root node

```
def delete_at_root(self):
    item = self.heap[1]
    self.heap[1] = self.heap[self.size]
    self.size -= 1
    self.heap.pop()
    self.sink(1)
    return item
```

Example

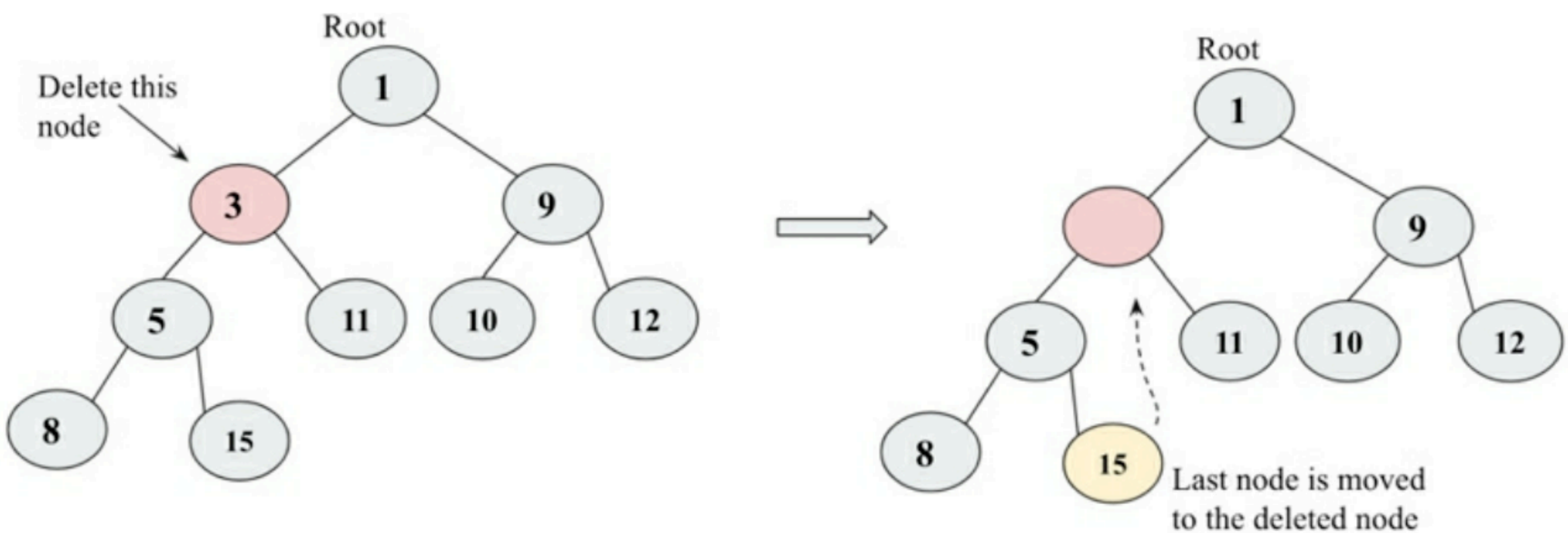
- Builds heap
- Deletes root

```
h = MinHeap()
for i in (2, 3, 5, 7, 9, 10, 6):
    h.insert(i)
print(h.heap)
n = h.delete_at_root()
print(n)
print(h.heap)
```

```
[0, 2, 3, 5, 7, 9, 10, 6]
2
[0, 3, 6, 5, 7, 9, 10]
```

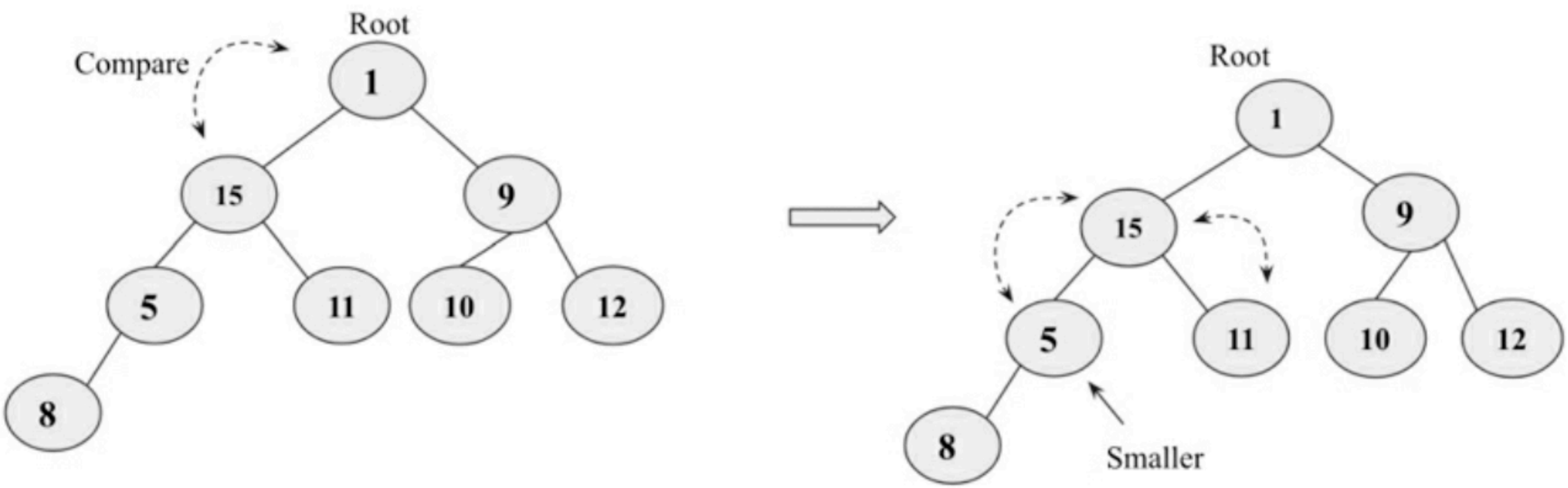
Deleting from a specific location

- Delete element
- Move last element to replace it



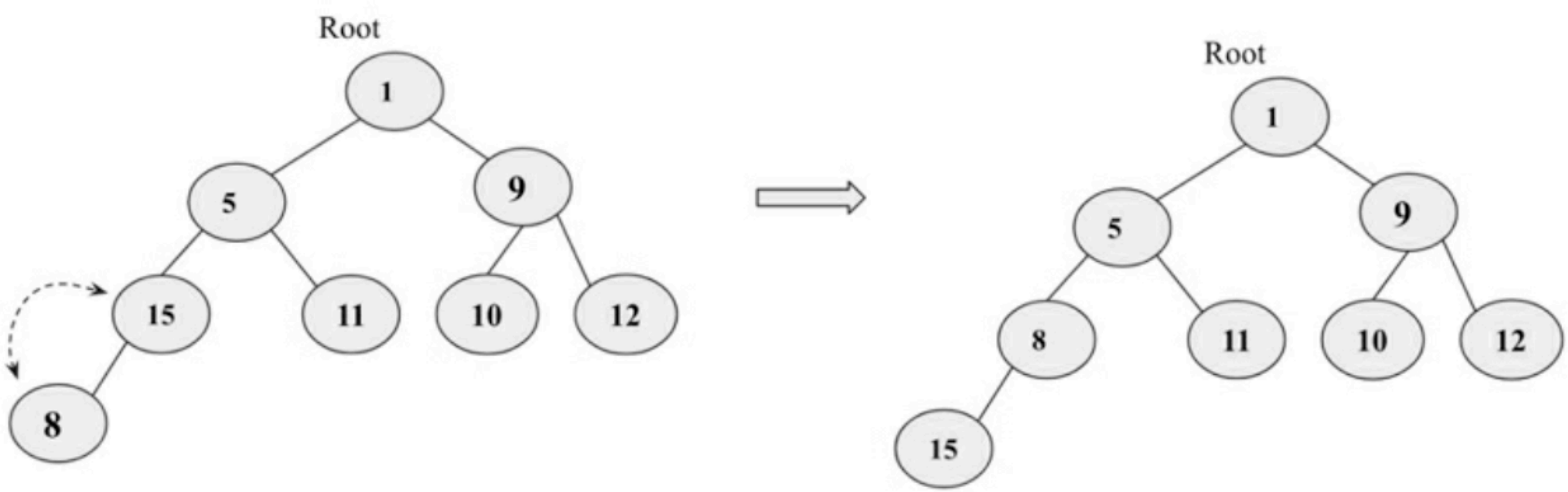
Deleting from a specific location

- Heapifying
- Compare to root node
- Then compare to children



Deleting from a specific location

- After a swap,
- Comparing to children again



Heap sort

- Very suitable for a large number of elements
  1. Create a min-heap from the elements
  2. Read and delete root node, then heapify
  3. Repeat step 2 until we get all the elements

Heap sort complexity

- Building the heap: $O(n)$
- Deleting the root occurs ***n*** times
  - Each time, we heapify from the root: $O(\log n)$
- So heap sort has complexity $O(n \log n)$

Priority Queues

Priority queues

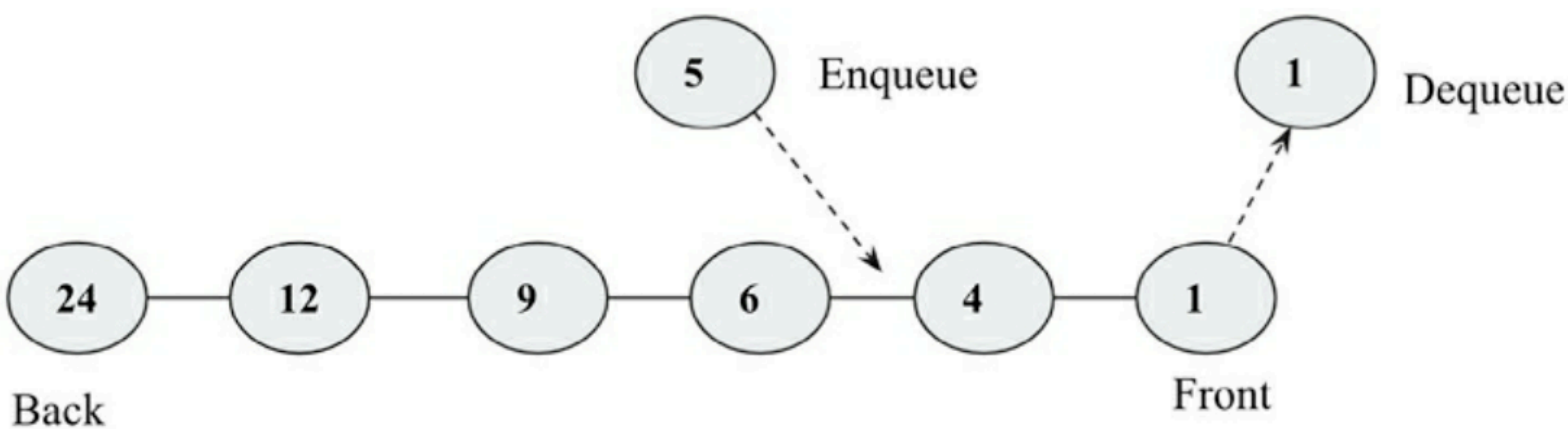
- A simple queue follows the FIFO principle
  - First in, first out
- A **priority queue** attaches a priority to the data
  - Data with highest priority is retrieved first
  - Ties are resolved with FIFO

Priority queue applications

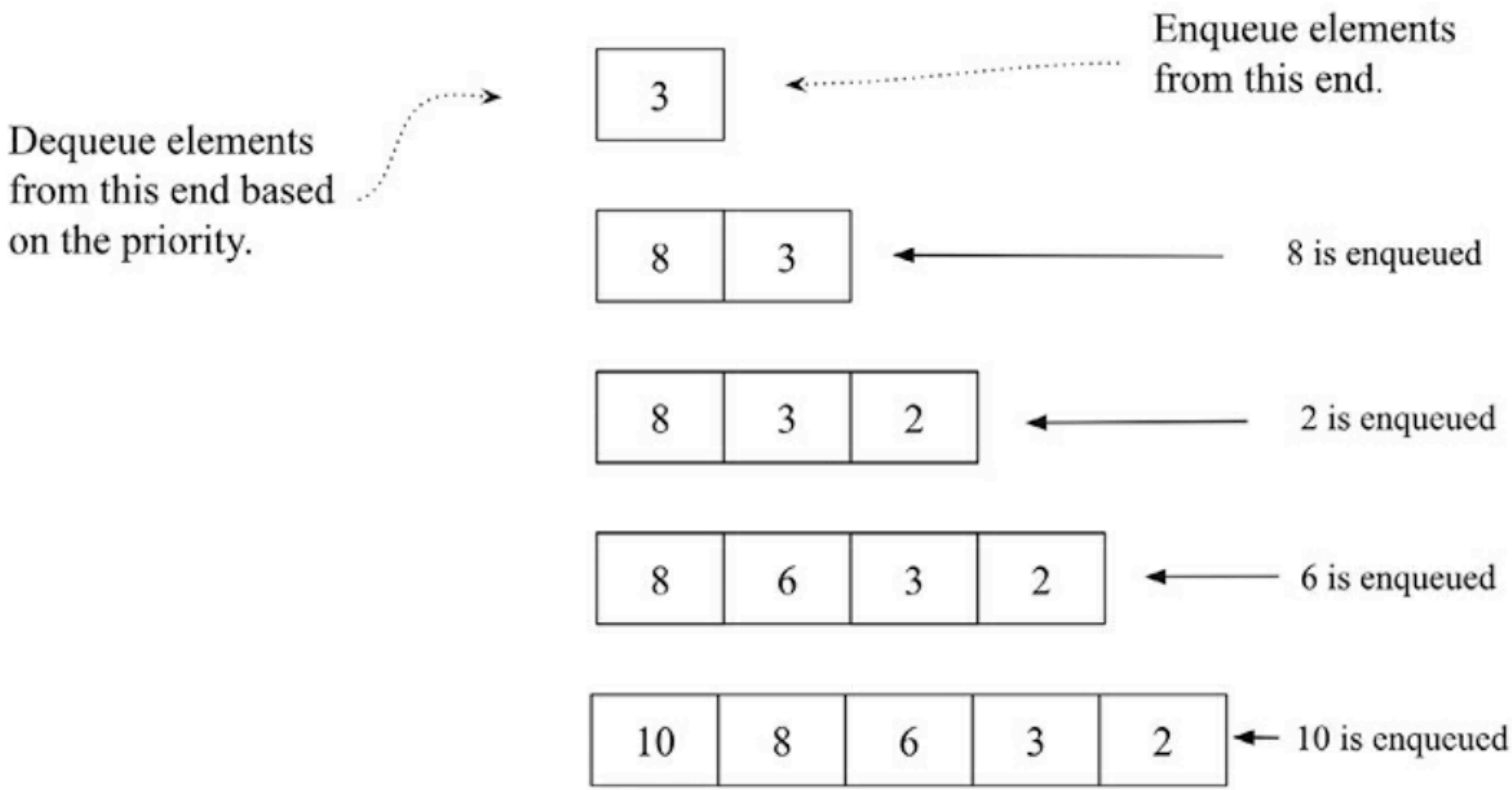
- CPU scheduling
- Dijkstra's shortest path
- A* algorithm
  - To find the shortest path between two nodes
- Huffman codes
  - For data compression

Priority queue

- Numbers represent priorities



Creating a priority queue



Kahoot!

Ch 7