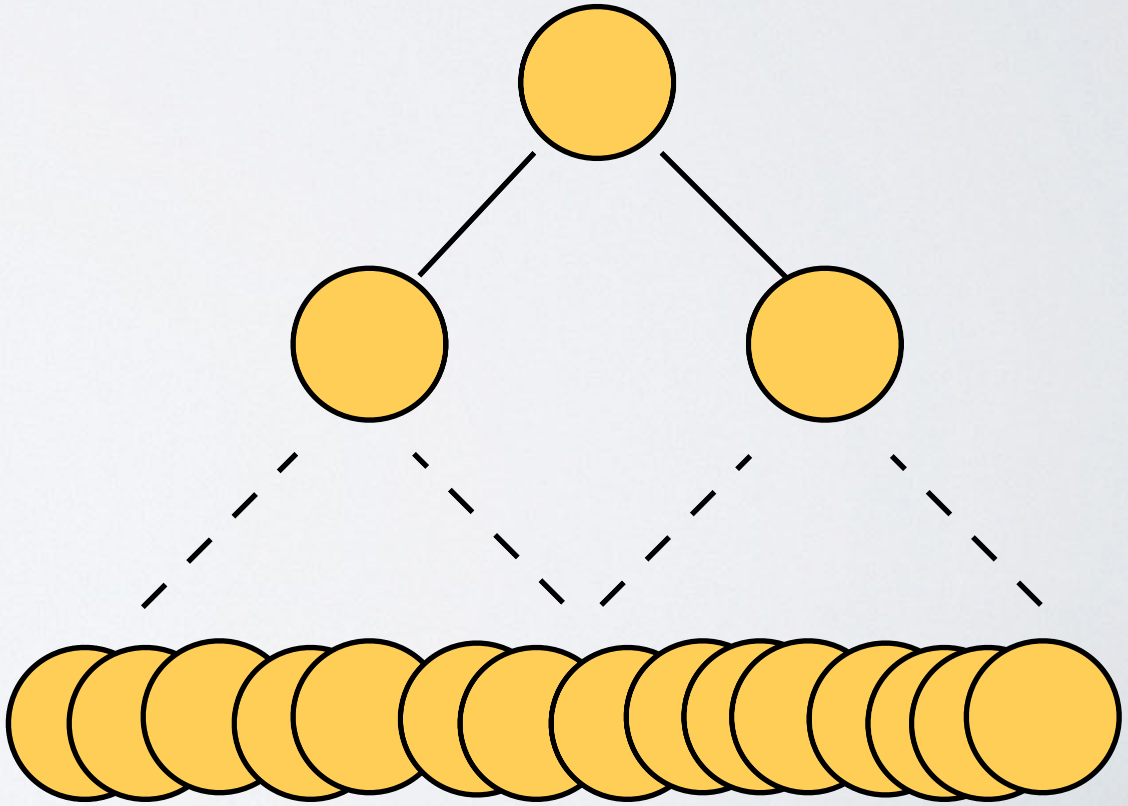
Priority Queues & Heaps

CS16: Introduction to Data Structures & Algorithms

Summer 2021

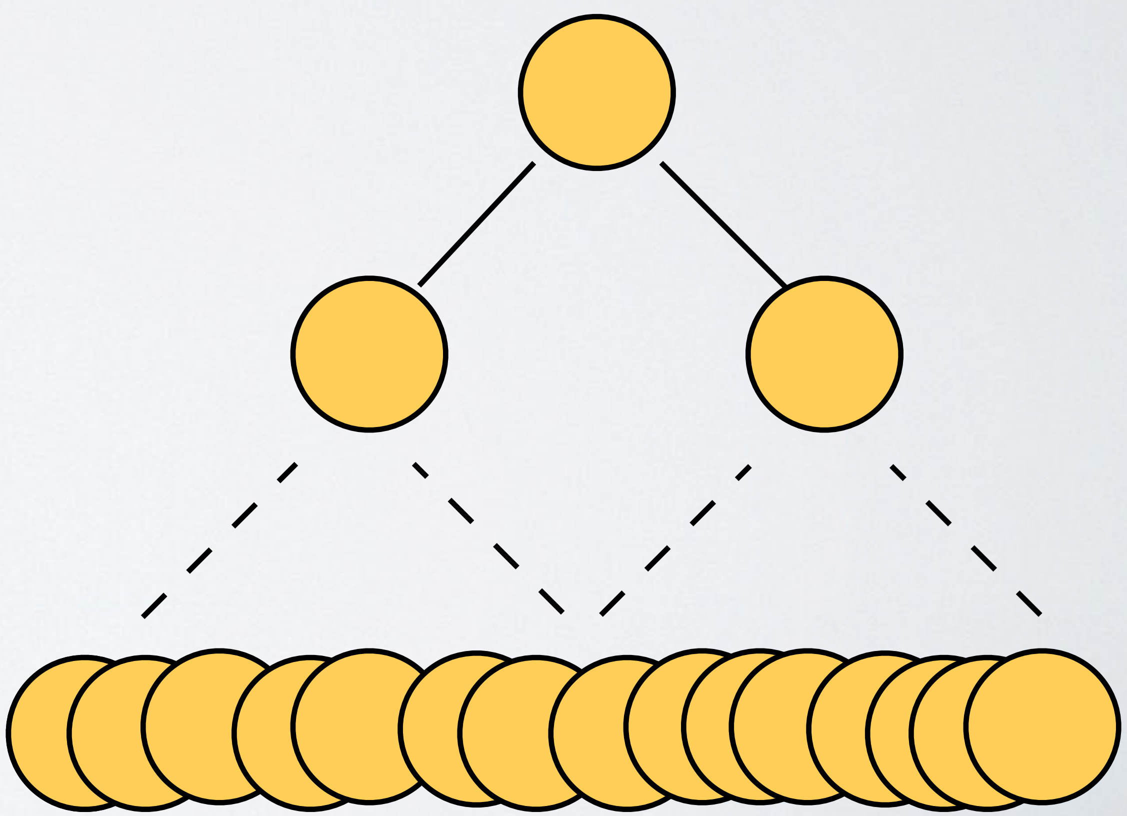
Why trees, revisited

- Trees: natural representation of hierarchical data
  - Expression trees, directories, parse trees, etc.
- Also used for organizing data that **aren't** inherently hierarchical
- Why?
- Consider a perfect binary tree with N nodes
  - Height is log N



Why trees, revisited

- Two operations:
  - Operation 1 looks at every **node** in the tree once, doing a constant amount of work per node. Runtime?
  - Operation 2 looks at every **level** of the tree once, doing a constant amount of work per level. Runtime?



Motivation

- Priority queues store items with various priorities
- Priority queues are everywhere
  - Plane departures: some flights have higher priority than others
  - Bandwidth management: real-time traffic like Skype transmitted first
  - Student dorm room allocations
  - …

Priority Queue ADT

- Stores key/element pairs
  - key determines position in queue
- **`insert(key, element)`**:
  - inserts element with key
- **`removeMin( )`**:
  - removes pair w/ smallest key and returns element

Using a priority queue

```
PQ = PriorityQueue
PQ.insert(2, “Practice banjo”)
PQ.insert(1, “Prepare lecture”)
PQ.insert(3, “Eat avocado”)
PQ.removeMin()
PQ.removeMin()
```

PQ contents:

Using a priority queue

```
PQ = PriorityQueue
PQ.insert(2, “Practice banjo”)
PQ.insert(1, “Prepare lecture”)
PQ.insert(3, “Eat avocado”)
PQ.removeMin()
PQ.removeMin()
```

PQ contents:

```
(2, “Practice banjo”)
```

Using a priority queue

```
PQ = PriorityQueue
PQ.insert(2, “Practice banjo”)
PQ.insert(1, “Prepare lecture”)
PQ.insert(3, “Eat avocado”)
PQ.removeMin()
PQ.removeMin()
```

PQ contents:

```
(2, “Practice banjo”)

(1, “Prepare lecture”)
```

Using a priority queue

```
PQ = PriorityQueue
PQ.insert(2, “Practice banjo”)
PQ.insert(1, “Prepare lecture”)
PQ.insert(3, “Eat avocado”)
PQ.removeMin()
PQ.removeMin()
```

PQ contents:

```
(2, “Practice banjo”)
(1, “Prepare lecture”)
(1, “Eat avocado”)
```

Using a priority queue

```
PQ = PriorityQueue
PQ.insert(2, “Practice banjo”)
PQ.insert(1, “Prepare lecture”)
PQ.insert(3, “Eat avocado”)
PQ.removeMin()
PQ.removeMin()
```

PQ contents:

```
(2, “Practice banjo”)

(1, “Eat avocado”)
```

Using a priority queue

```
PQ = PriorityQueue
PQ.insert(2, “Practice banjo”)
PQ.insert(1, “Prepare lecture”)
PQ.insert(3, “Eat avocado”)
PQ.removeMin()
PQ.removeMin()
```

PQ contents:

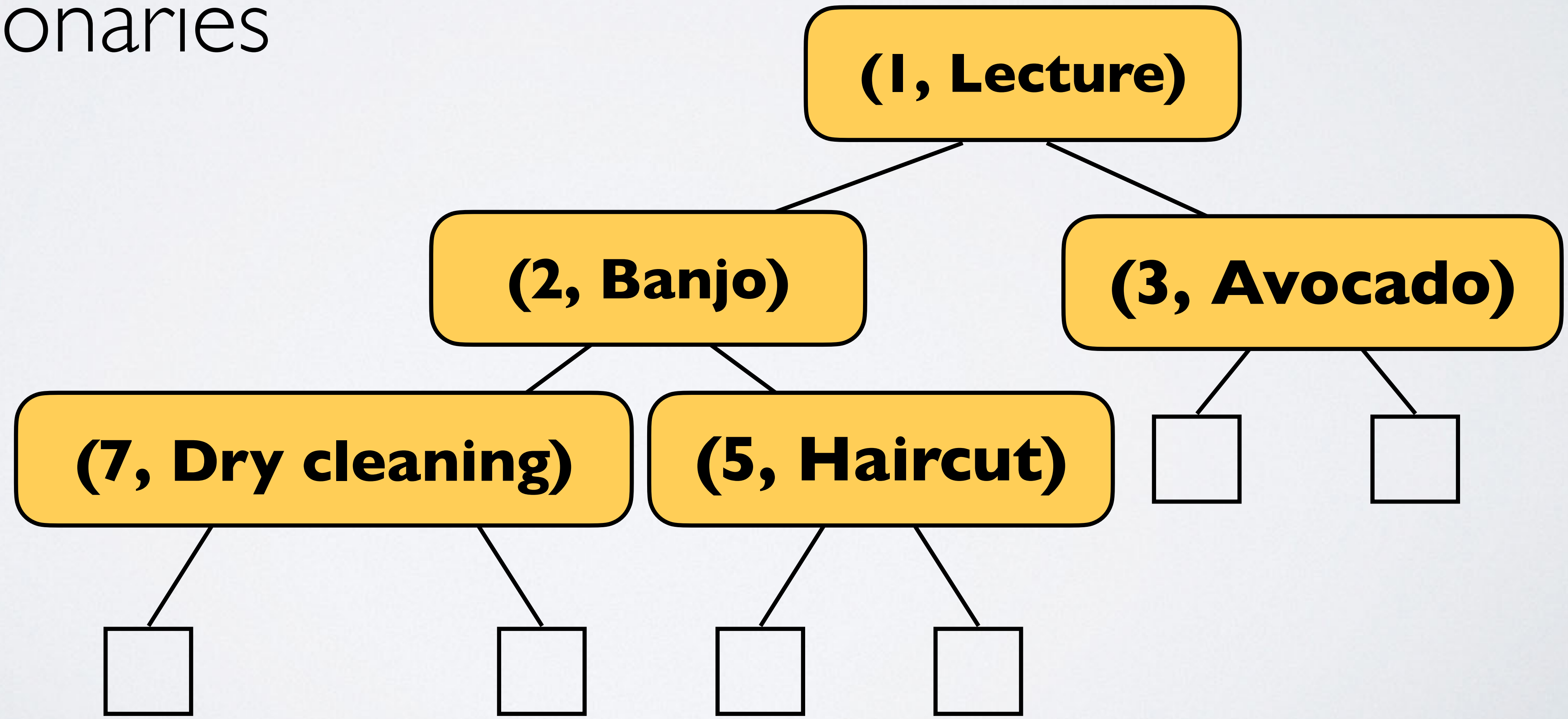
```
(1, “Eat avocado”)
```

Naive PQ implementation

- Store elements in an expandable array called **`data`**
- **`insert(key, element)`**: $\boxed{O(1)}$
  - add **`(key, element)`** to end of **`data`**
- **`removeMin( )`**: $\boxed{O(n)}$
  - scan through **`data`**, remove and return element with smallest key
- Runtimes?

Heaps!

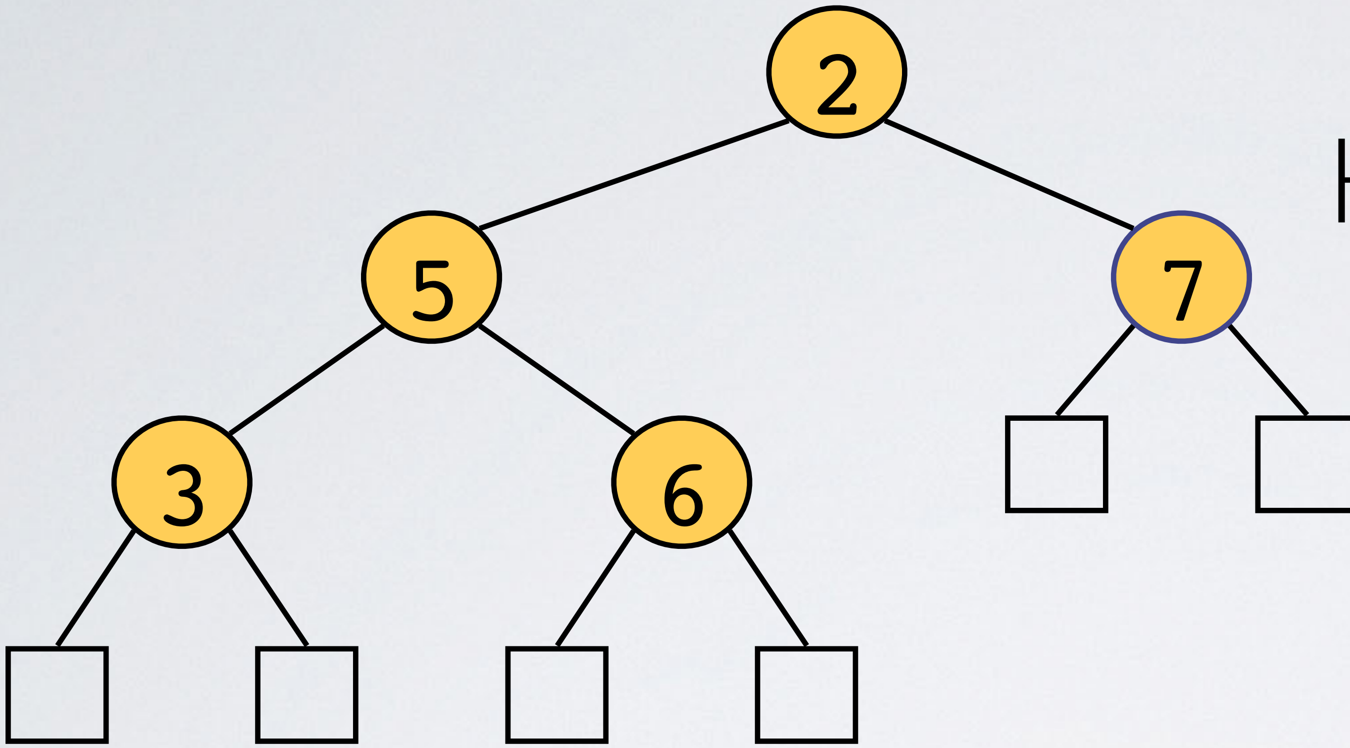
- Tree-based PQ implementation
- **Data structure**, not an ADT
  - Heaps are to PQs as Hash tables are to dictionaries



Heap Properties

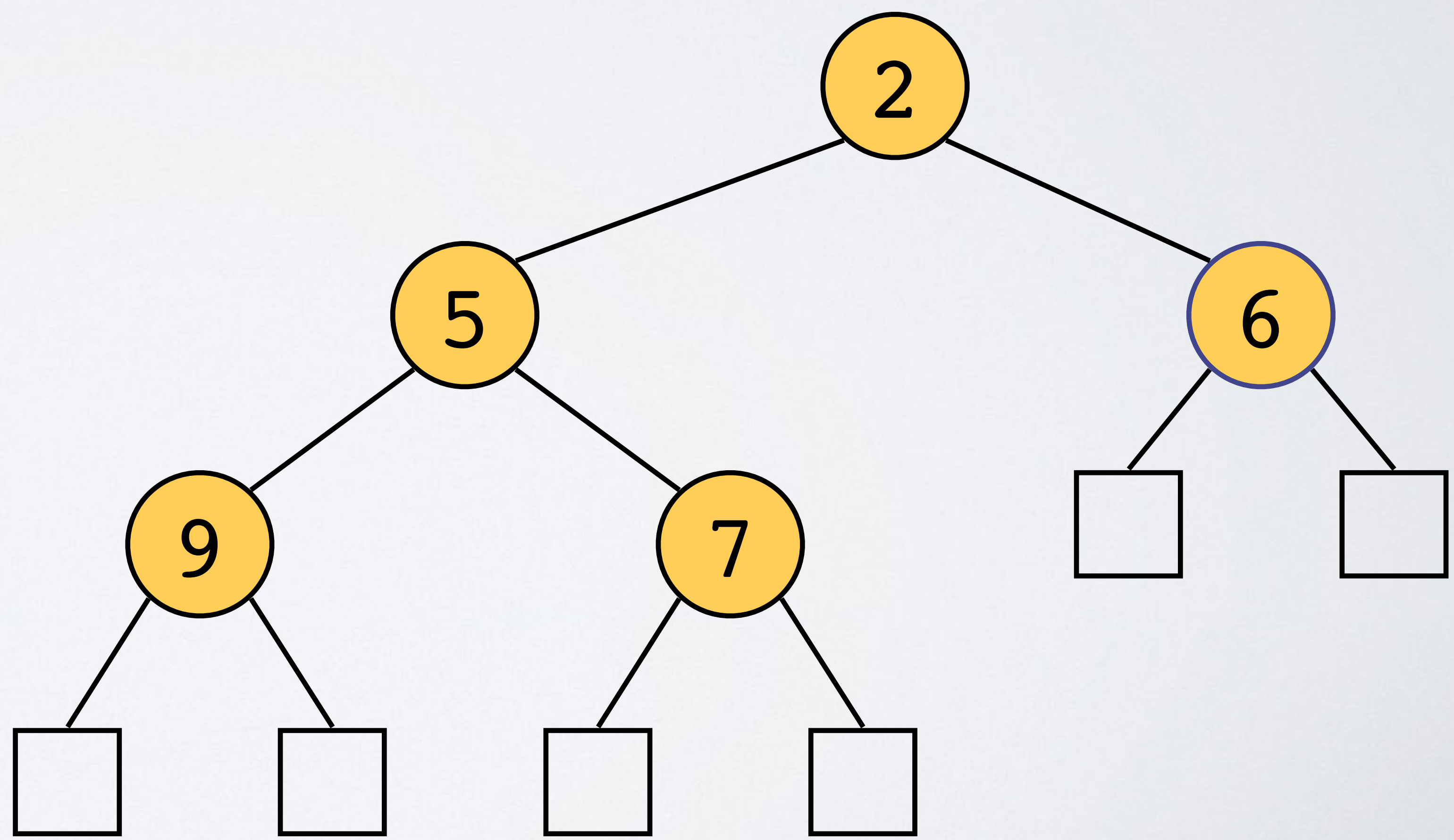
- Binary tree
  - each node has at most **`2`** children
- Each node has a priority (key)
- Heap has an order
  - min-heap: n.parent.key ≤ n.key
- Left-complete
- Height of **`O(log n)`**

Valid heaps?

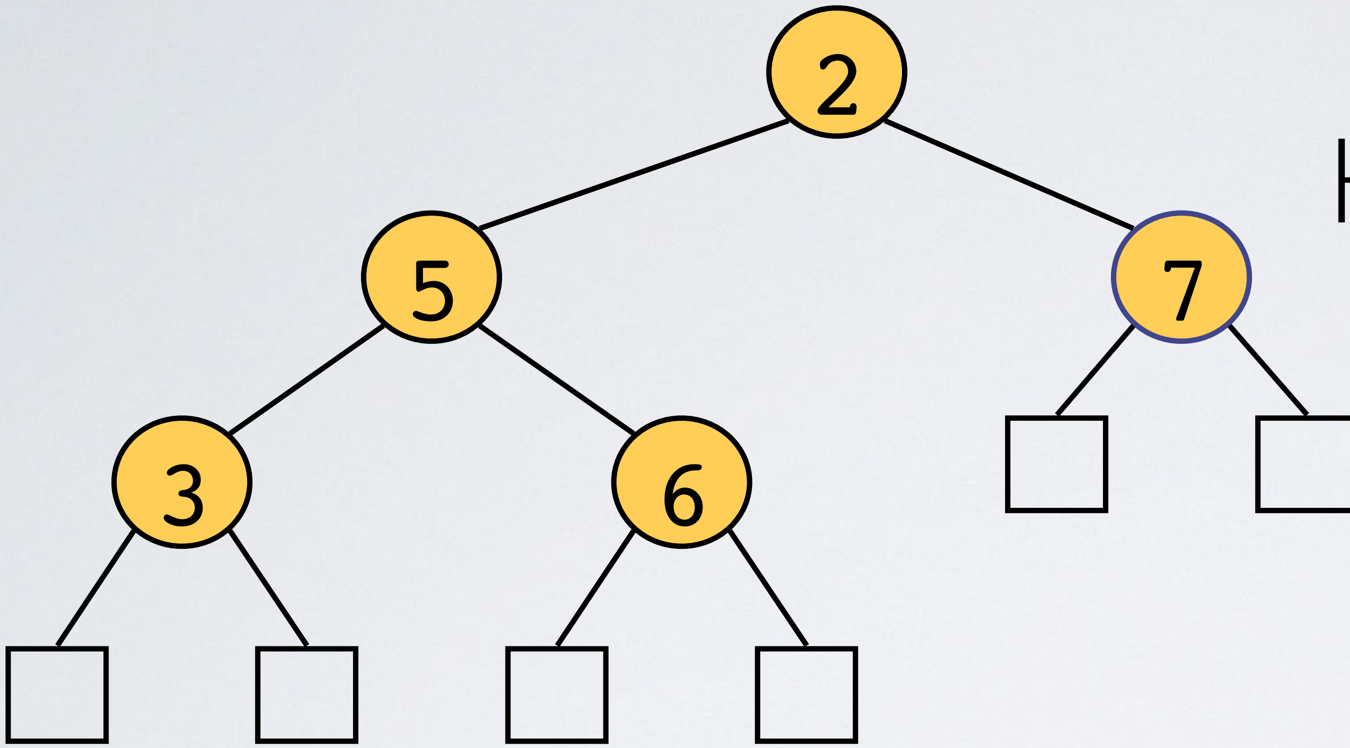


Heap A

Heap B

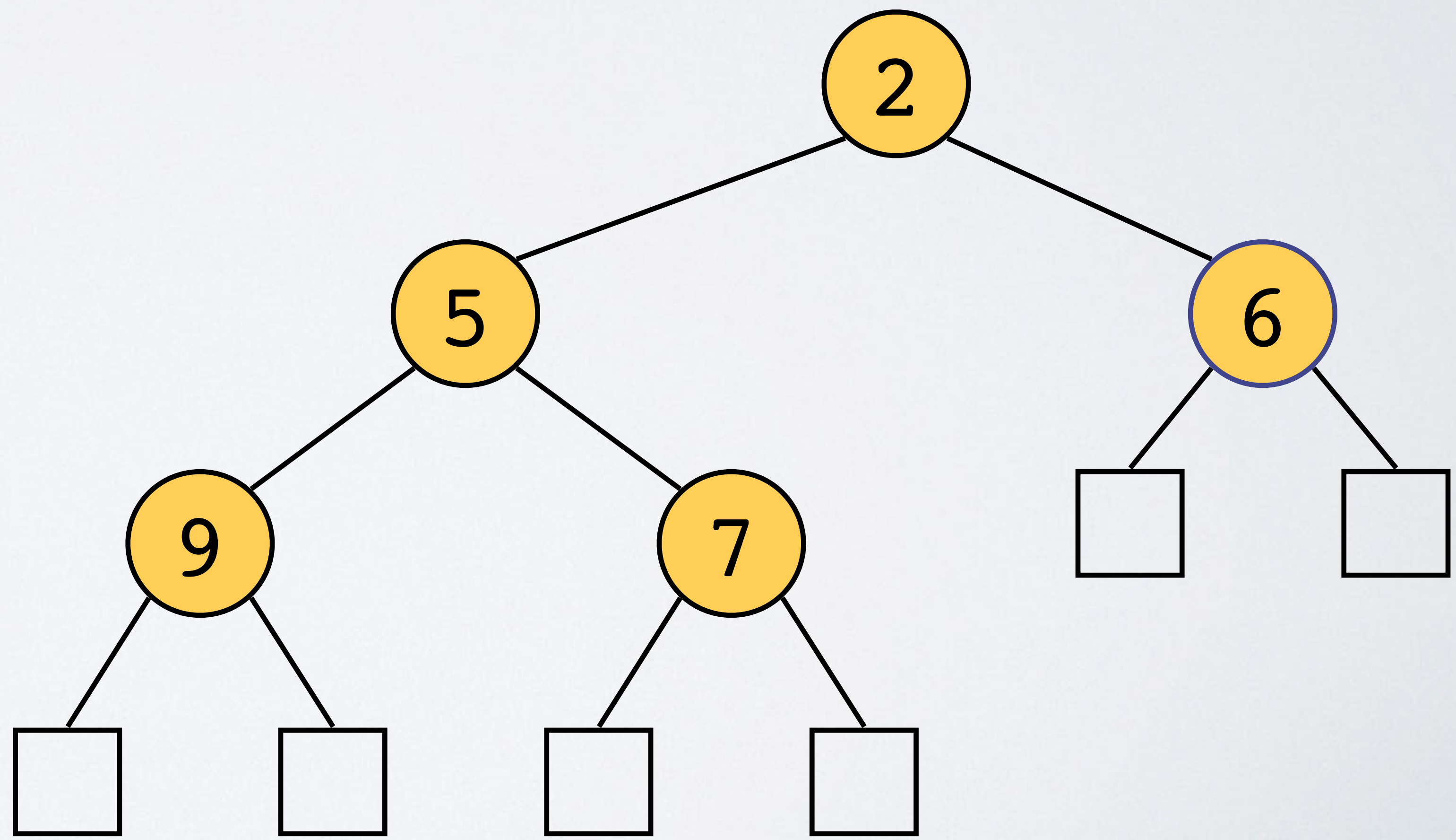


Valid heaps?



Heap A

No

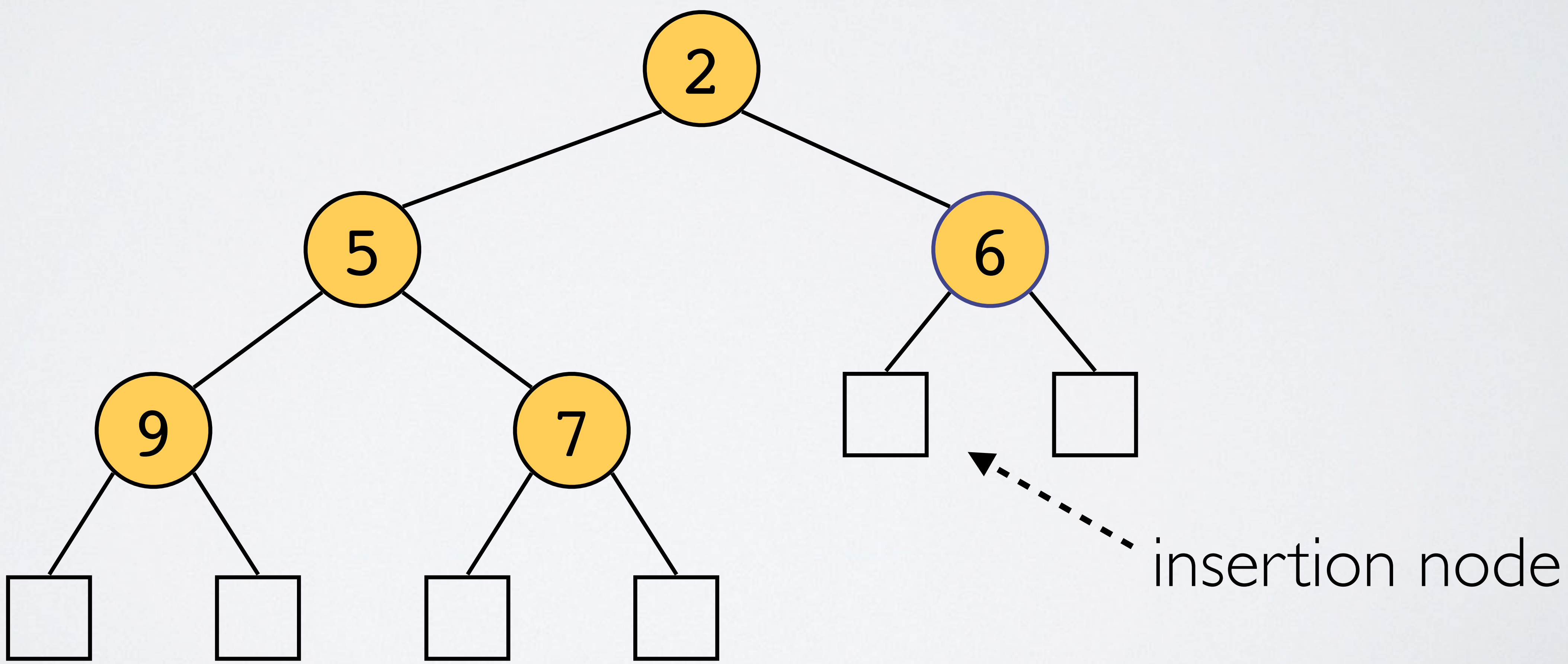


Heap B

Yes

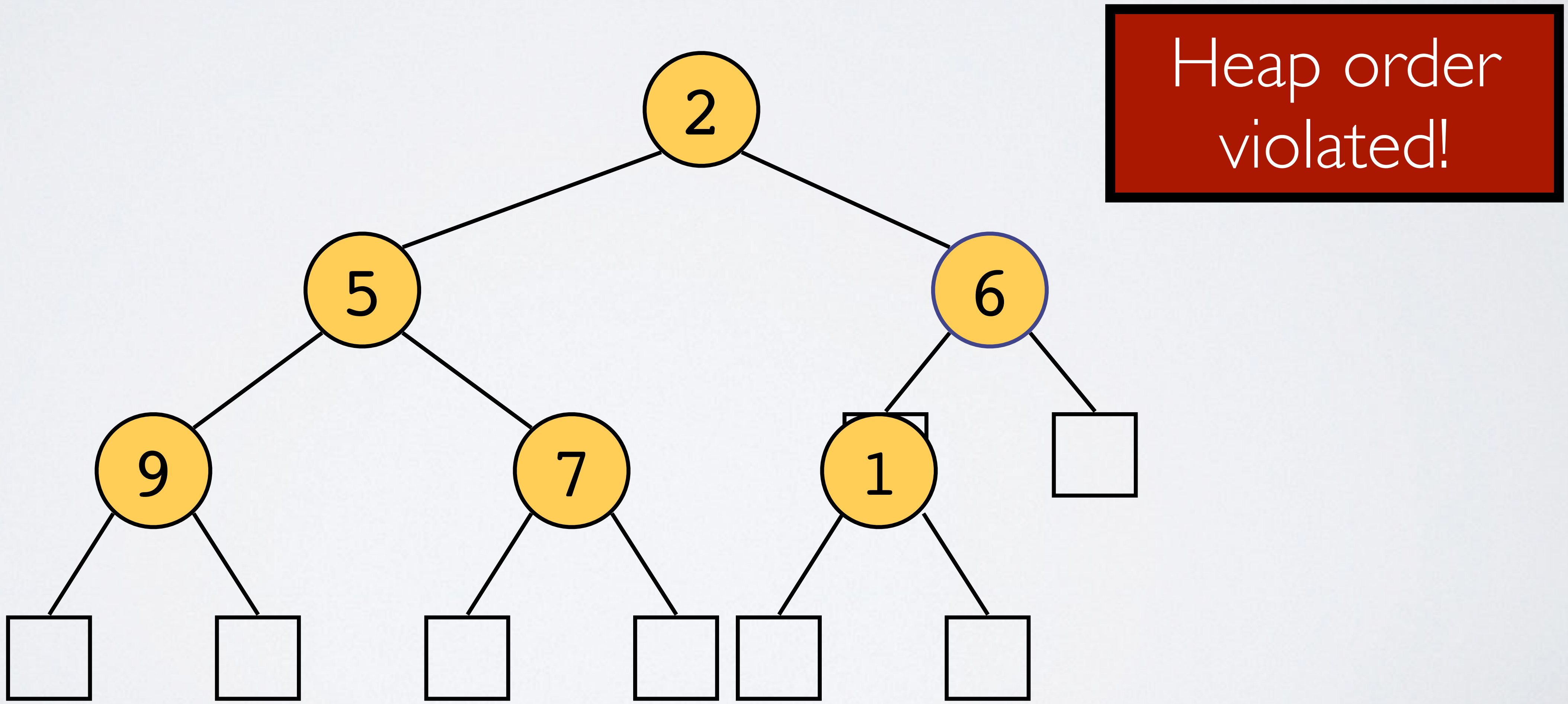
Heap: `insert`

- Need to keep track of "insertion node"
  - leaf where we will insert new node…
  - …so we can keep heap left-complete



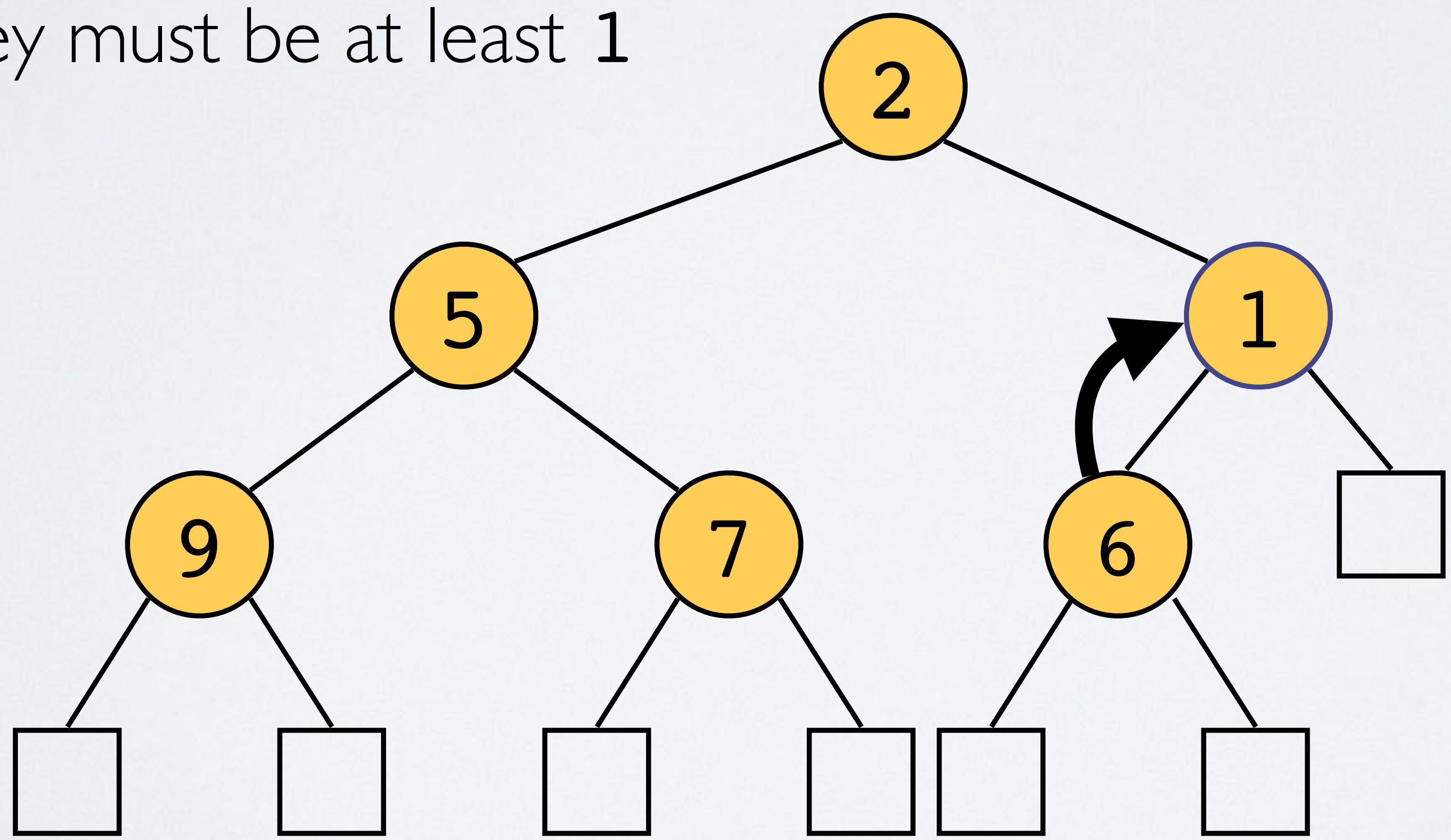
Heap: `insert`

- Ex: insert(`1`)
  - replace insertion node w/ new node



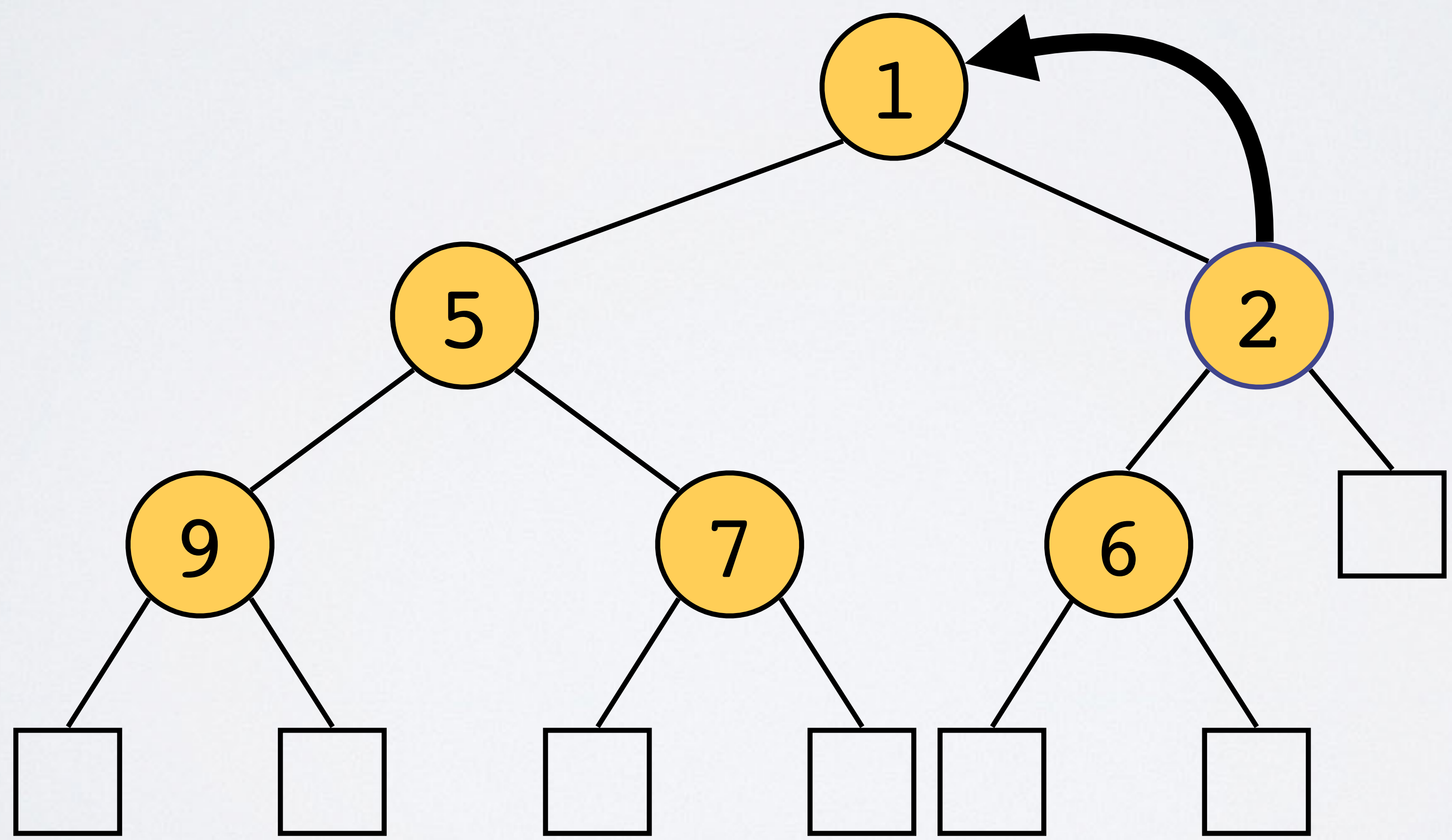
Heap: `upheap`

- Repair heap: swap new element up tree until keys are sorted
- First swap fixes everything below new location
  - since every node below **6**'s old location has to be at least **6**…
  - …they must be at least **1**



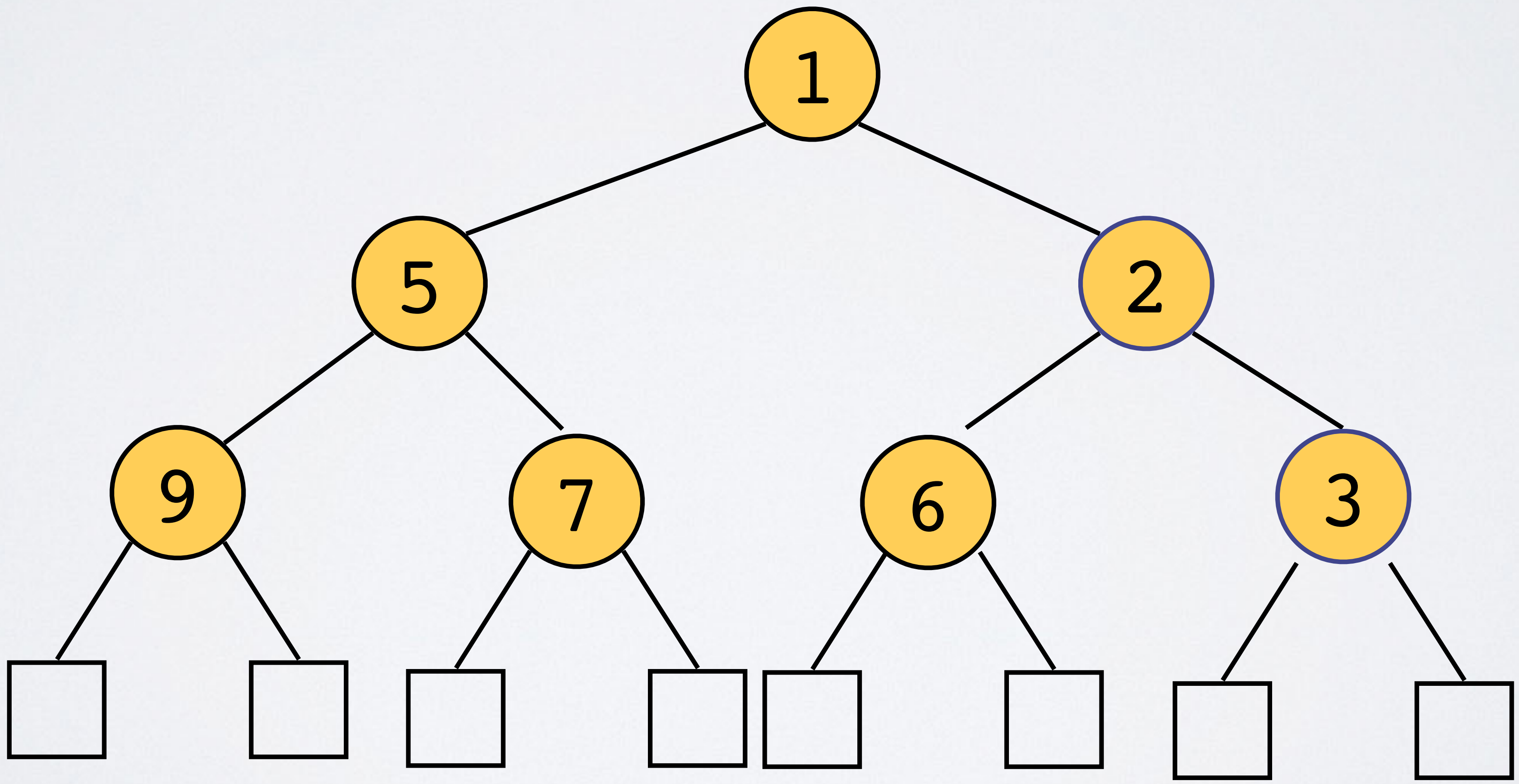
Heap: `upheap`

- One more swap since `1≤2`
- Now left-completeness and order are satisfied



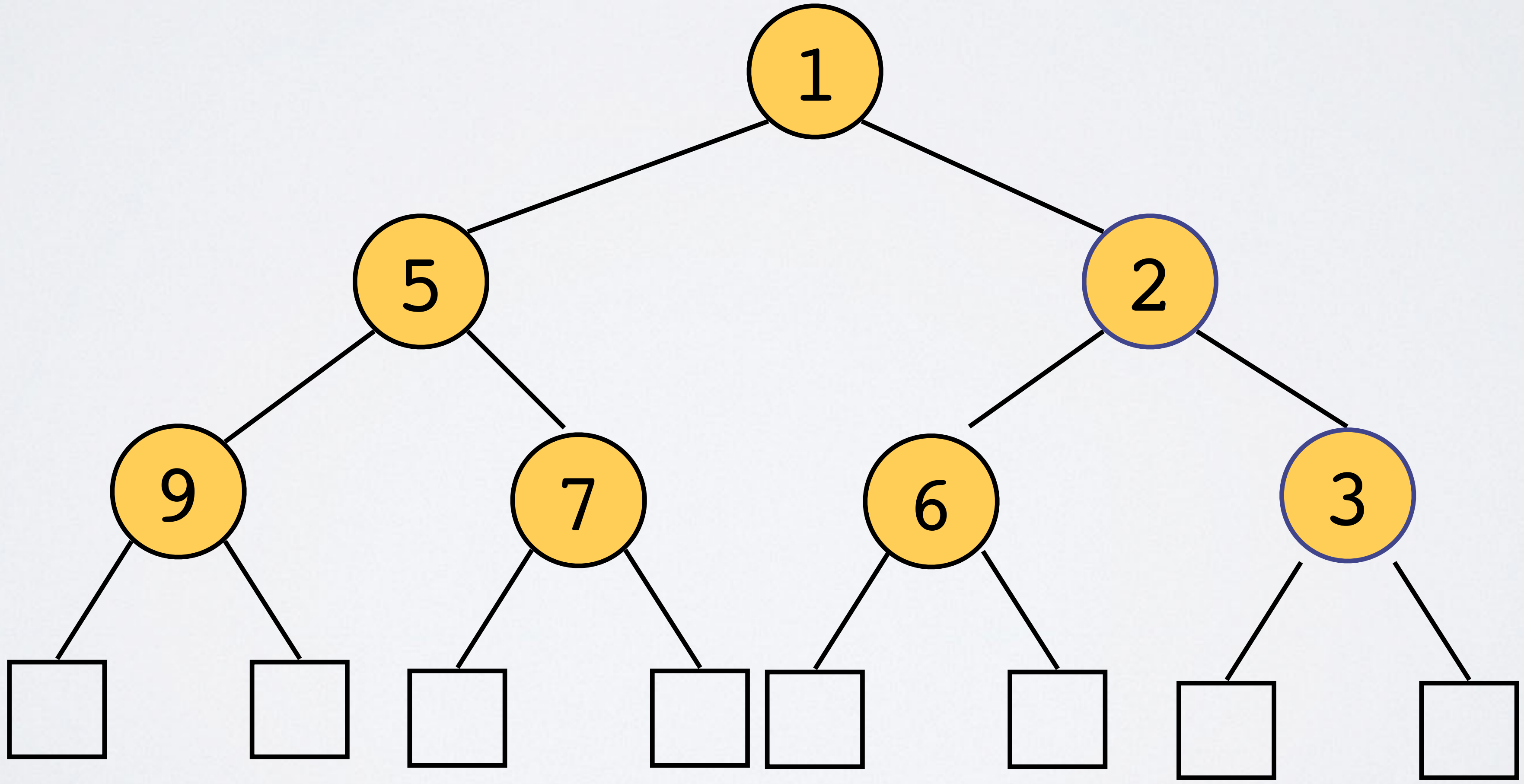
Heap: `insert`

- Ex: insert(3)



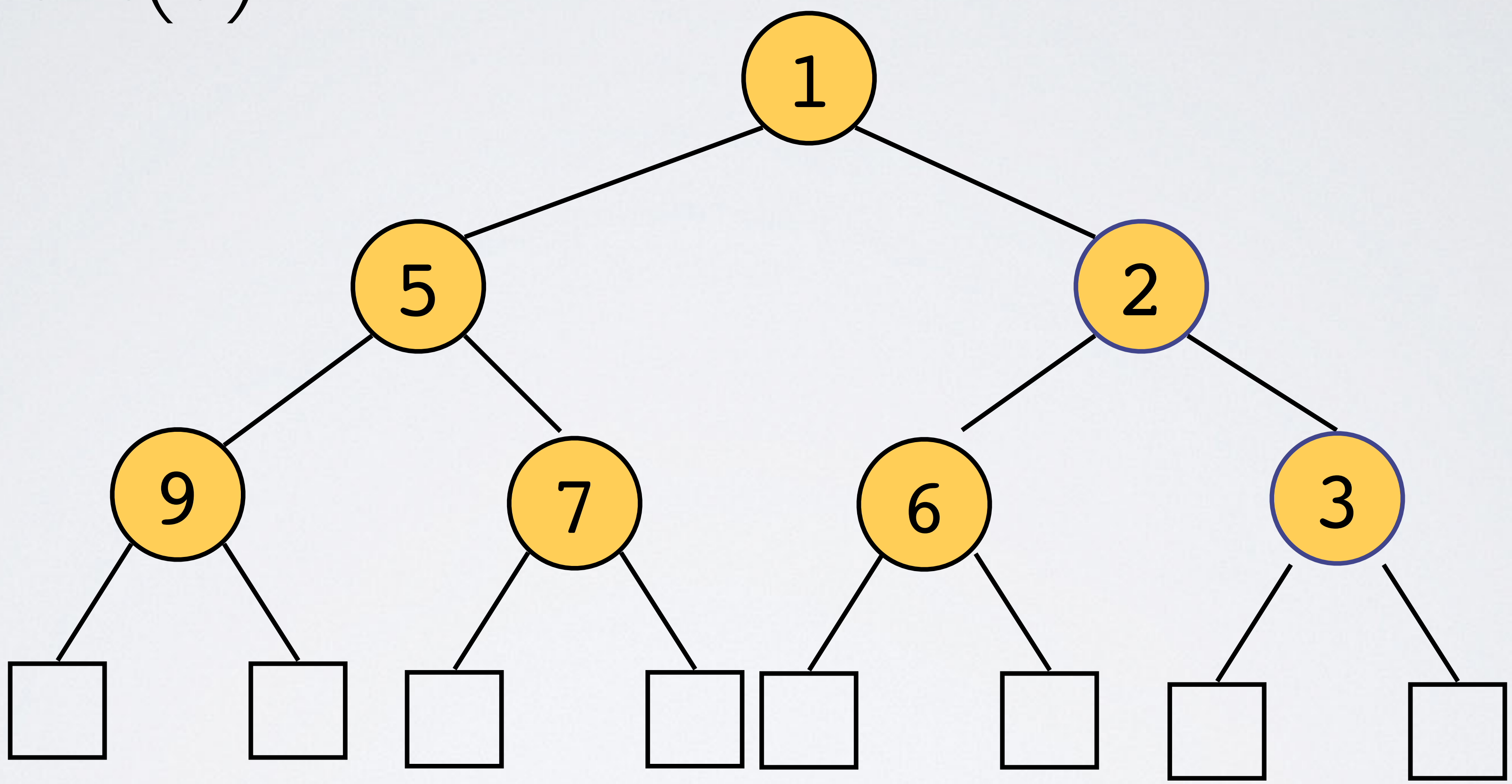
Heap: `insert`

- Ex: insert(8)



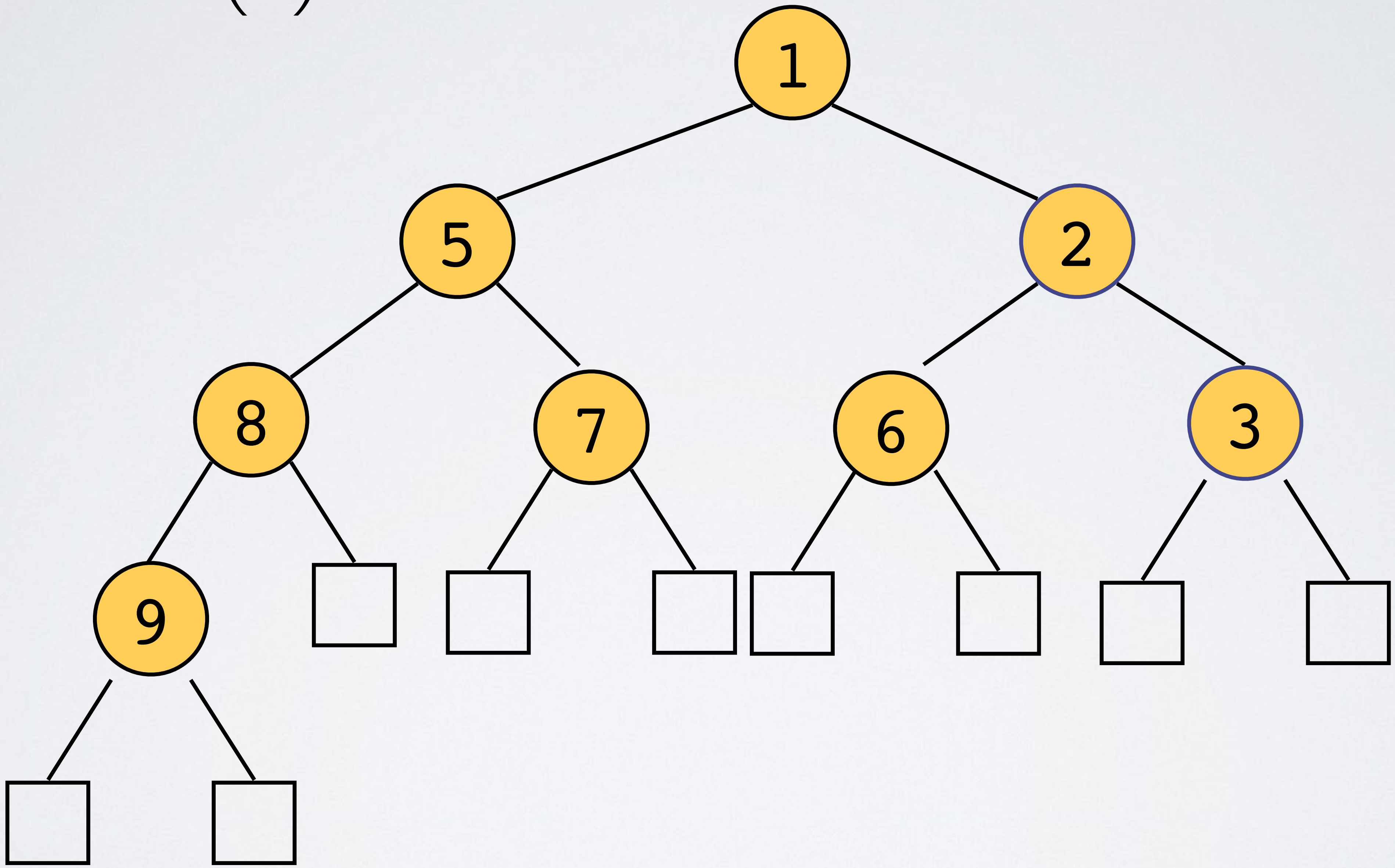
Heap: `insert`

- Ex: insert(8)



Heap: `insert`

- Ex: insert(8)

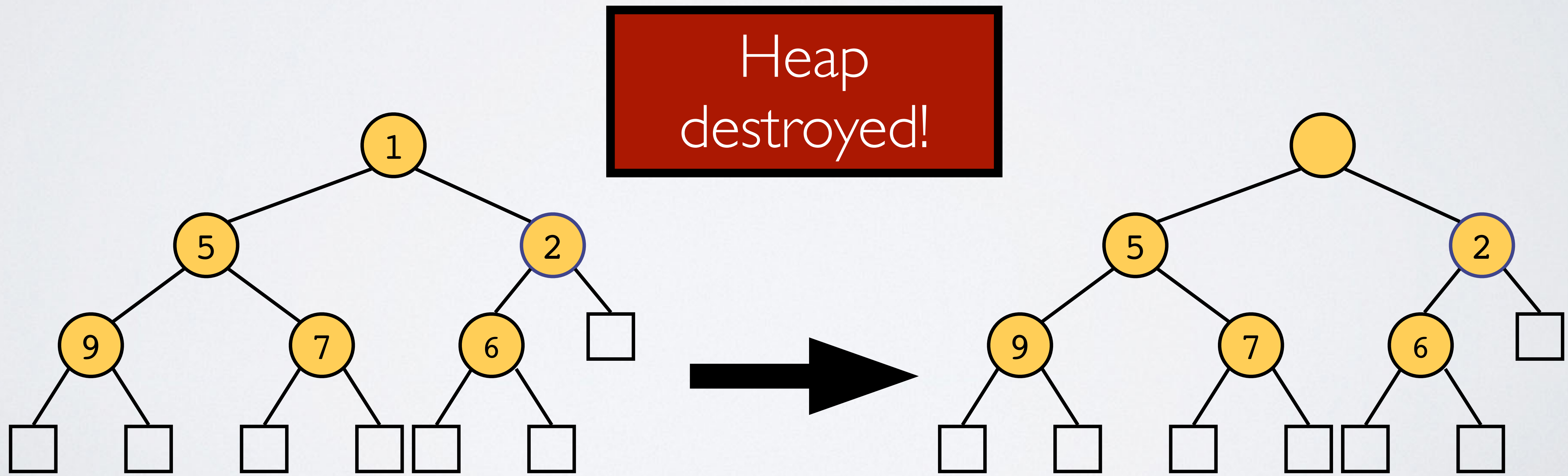


Heap: `upheap` Summary

- After inserting a key `k`, order may be violated
- `upheap` restores order by
  - swapping key upward from insertion node
  - terminates when either the root is reached…
  - …or some node whose parent has key less or equal than `k`
- Heap insertion has runtime
  - `O(log n)`, why?
  - because heap has height `O(log n)`
  - perfect binary tree with `n` nodes has height `log(n+1)-1`

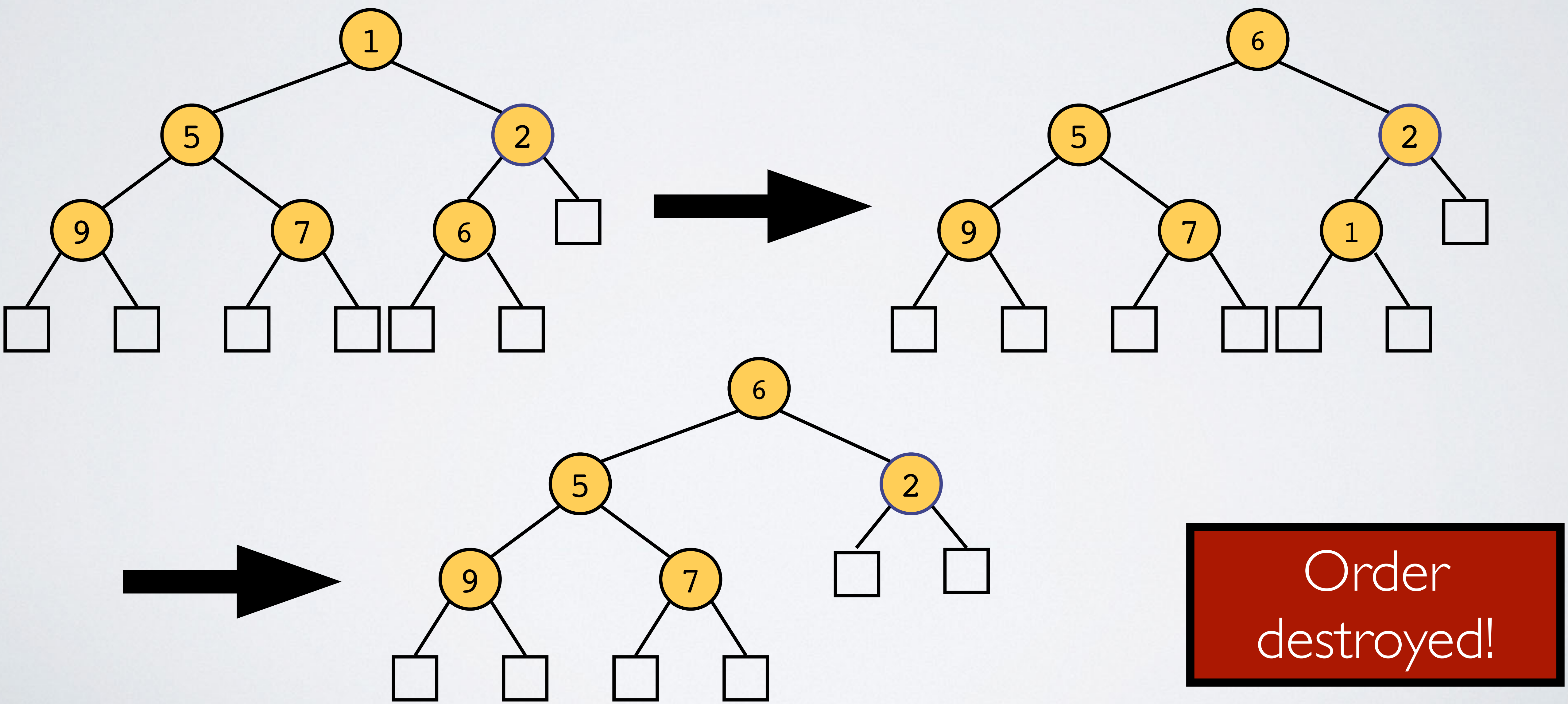
Heap: `removeMin`

- Remove root
  - because it is always the smallest element
- How can we remove root w/o destroying heap?



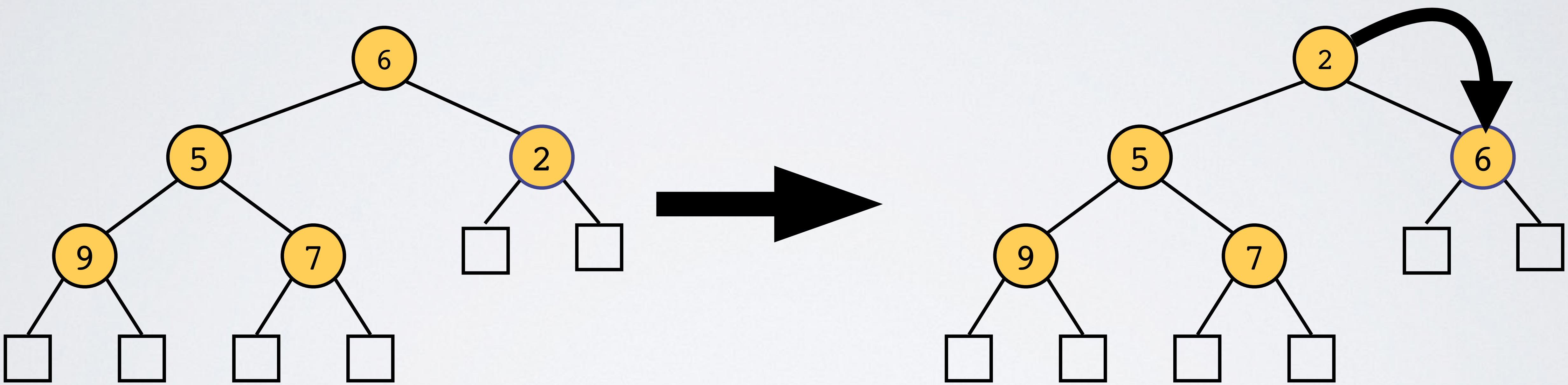
Heap: `removeMin`

- Instead swap root with last element & remove it
  - removing last element is easy



Heap: `removeMin`

- Now swap root down as necessary



Heap is in order!

Heap: `downheap` Summary

- `downheap` restores order by
  - swapping key downward from the root…
  - …with the **smaller** of 2 children
  - terminates when either a leaf is reached or
  - …some node whose children have key `k` or more
- `downheap` has runtime
  - `O(log n)`, why?
  - because heap has height `O(log n)`

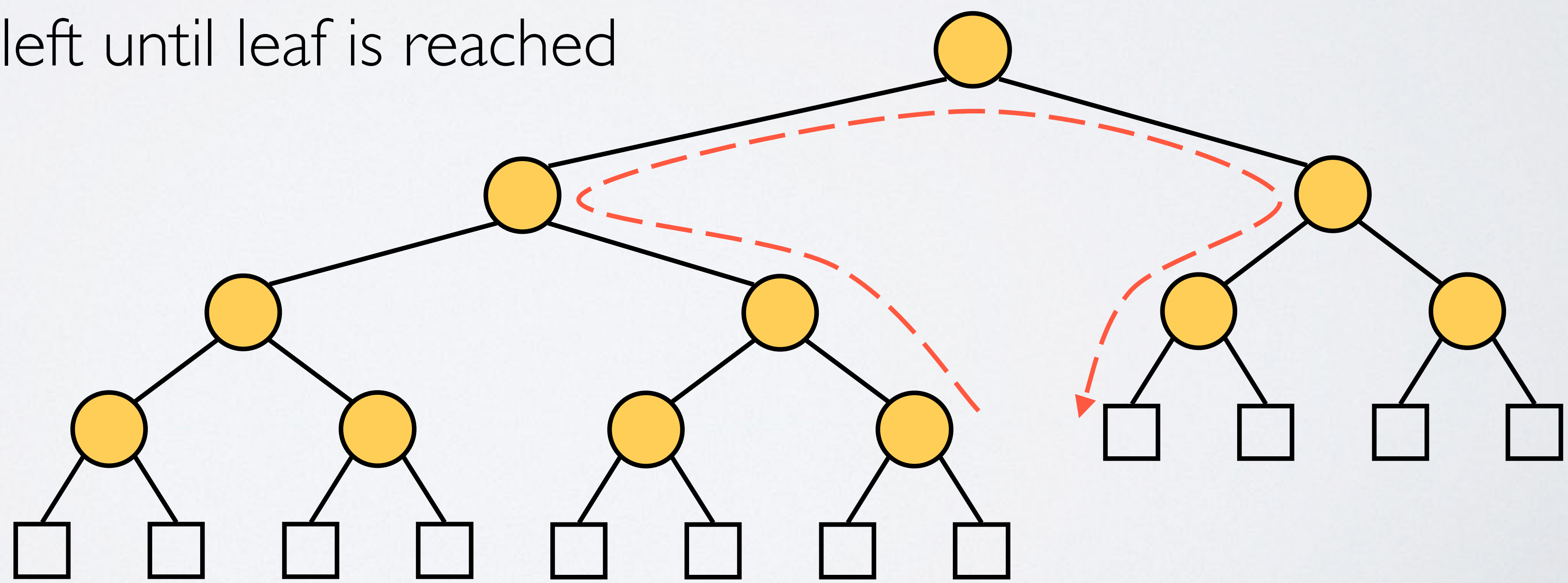
Summary of Heap

- **`insert(key, value)`**
  - insert value at insertion node
    - insertion node must be kept track of
  - **`upheap`** from insertion node as necessary
- **`removeMin( )`**
  - swap root with last item
  - delete (swapped) last item
  - **`downheap`** from root as necessary

Finding Insertion Node

- Can be found in `O(log n)`
- Start at last added node
- Go up until a left child or root is reached
- If left child
  - go to sibling (corresponding right child)
  - then go down left until leaf is reached

Can be done in O(1) time by using additional data structure...need this for project!



Array-based Heap

- Heap with **`n`** keys can be represented w/ array of size **`n+1`**
- Storing nodes in array
    - Node stored at index **`i`**
        - left child stored at index **`2i`**
        - right child stored at index **`2i+1`**
    - Leaves & edges not stored
    - Cell **`0`** not used
- Operations
    - insert: store new node at index **`n+1`**
    - removeMin: swap w/ index **`n`** and remove