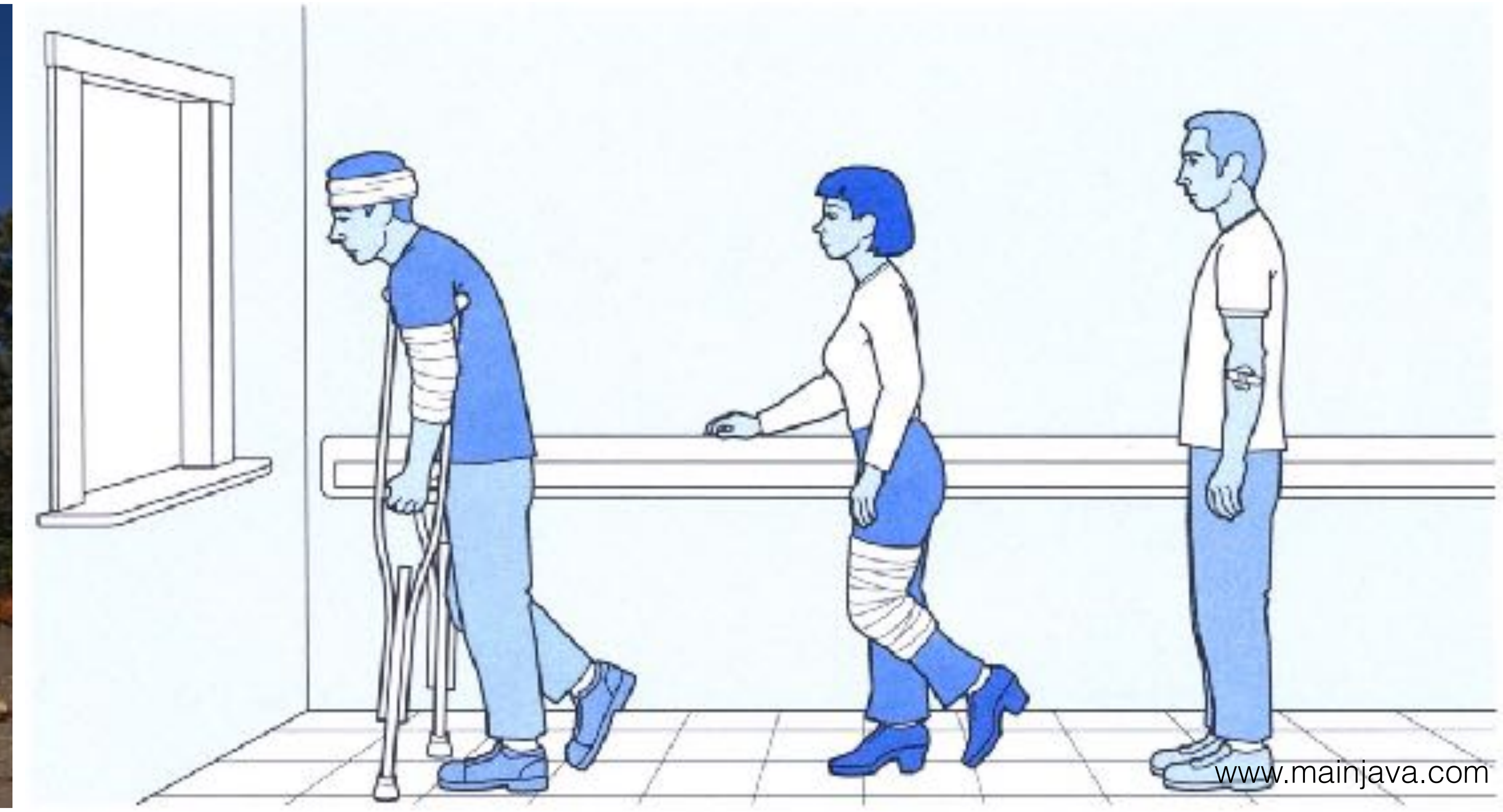


CSCI 241

Lecture 15
Heaps

Announcements

- Office hours today:
  - Nick: Today 2-4 in CF163
  - Me: Today 2:30-4 in CF461; Friday’s OH canceled.
- Quiz 4 released on Gradescope.
  - Will be on Canvas soon.
- No quiz this Friday! 🎉

About the Exam

- Friday, November 2nd in class.
- One double-sided 8.5x11” sheet of **hand-written** notes.
- Quizzes are the most efficient way to study.
  - If you have a problem accessing your graded quiz, contact me.
- I made you (me) a study guide: https://github.com/wehrwein-teaching/csci241_18f_studyguide/wiki
  - At the bottom: crowdsourced list of pointers to practice problems and ABCD questions. Please contribute as you study!

Goals

- Understand how to implement a Priority Queue using a heap
- Understand the storage mechanism for heaps.
- Be prepared to implement a heap's add, peek, and poll operations.

Priority Queue: heap implementation

- A heap is a **concrete** data structure that can be used to **implement** a Priority Queue

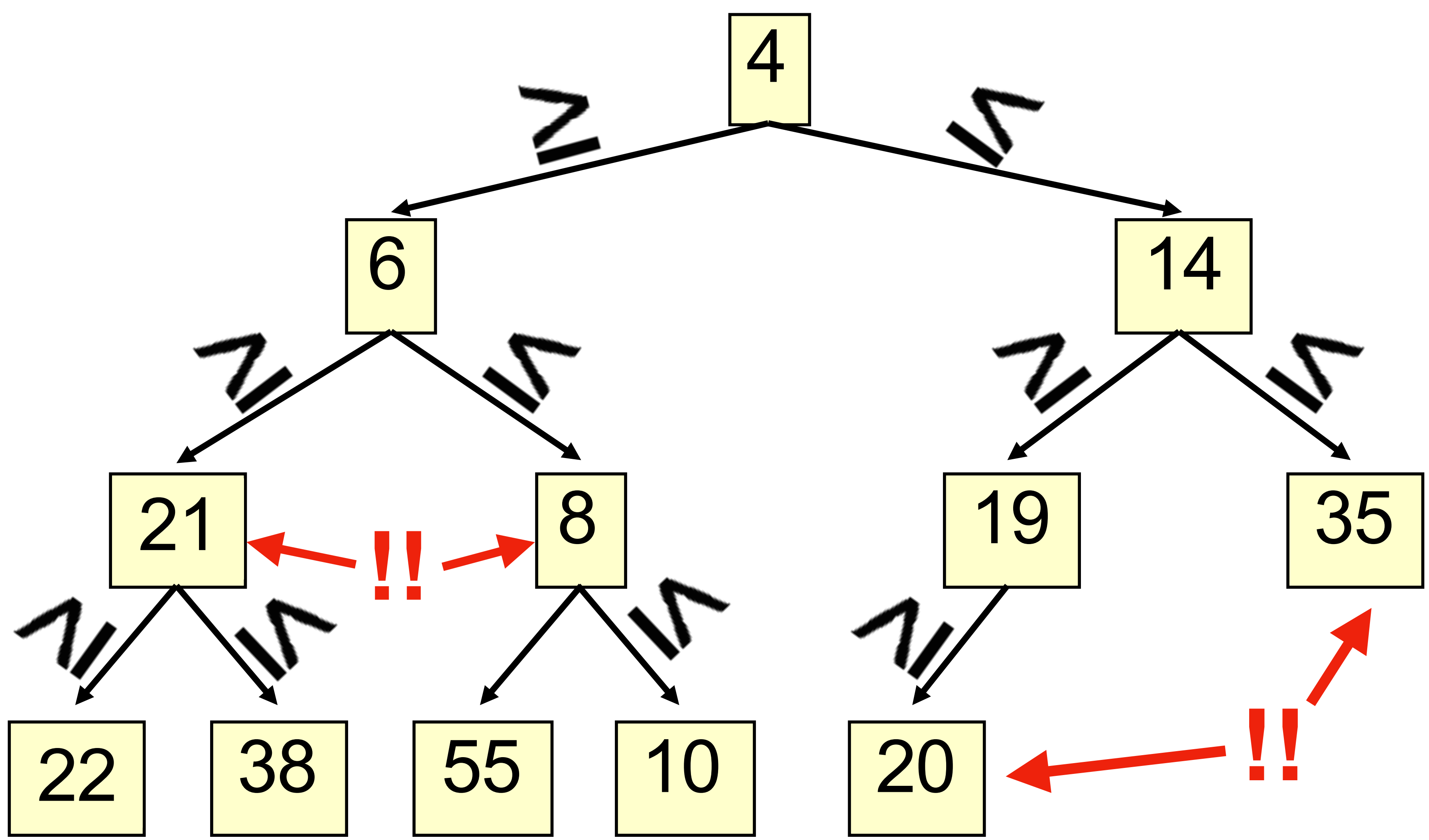
- Better runtime complexity than either list implementation:
  - **peek()** is O(1)
  - **poll()** is O(log n)
  - **add()** is O(log n)

- Not to be confused with *heap memory*, where the Java virtual machine allocates space for objects – different usage of the word heap.

A heap is a special binary tree with two additional properties.

A heap is a special binary tree.

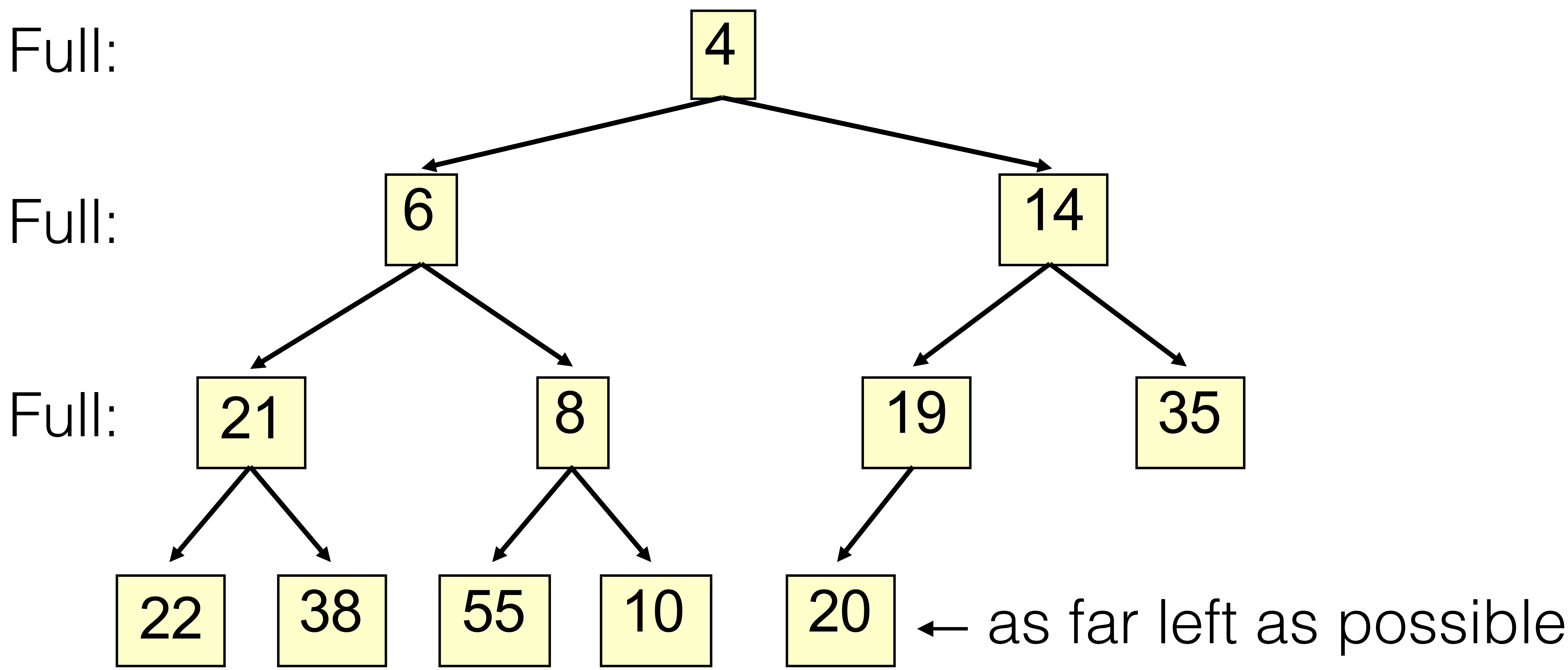
1. **Heap Order Invariant:**
Each element >= its parent.



A heap is a special binary tree.

2. **Complete:** no holes!

- All levels except the last are **full**.
- Nodes in last level are as far left as possible.



Heap operations

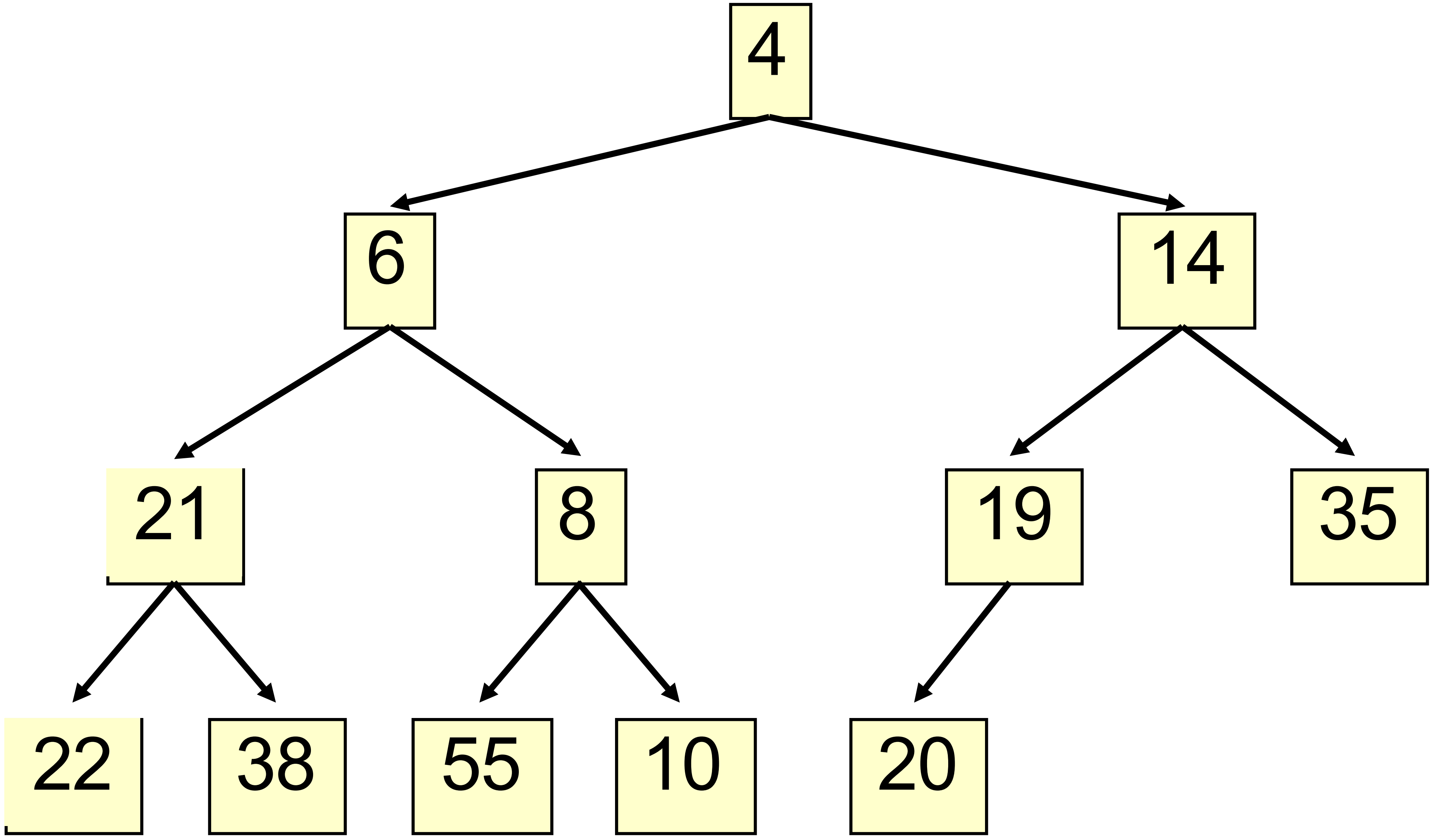
```
interface PriorityQueue {
 boolean add(Object e); // insert e
 Object peek(); // return min element
 Object poll(); // remove/return min element
 void clear();
 boolean contains(Object e);
 boolean remove(Object e);
 int size();
 Iterator iterator();
}
```

add(e)

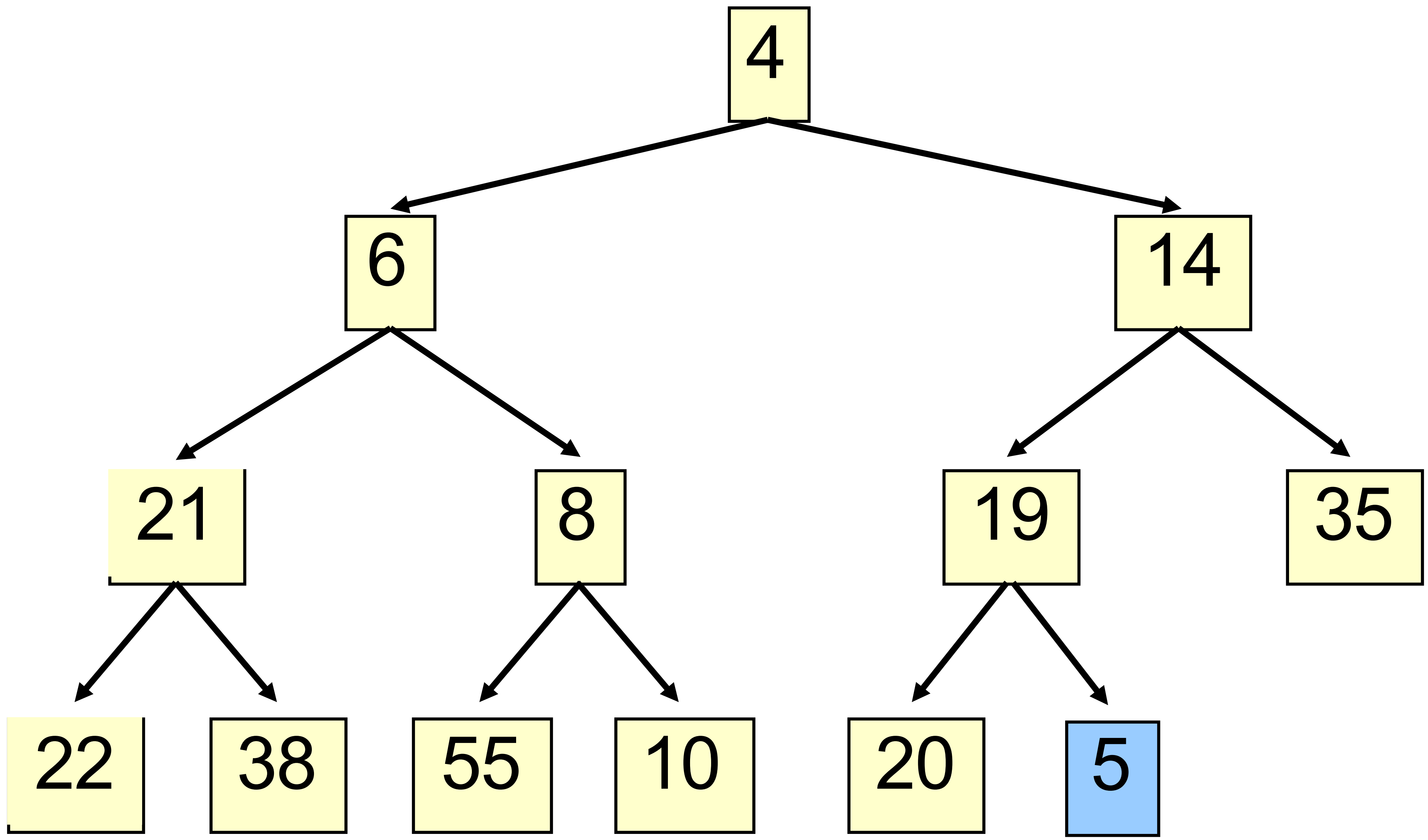
Algorithm:

- Add e in the wrong place
- While e is in the wrong place
  - move (“bubble”) e towards the right place

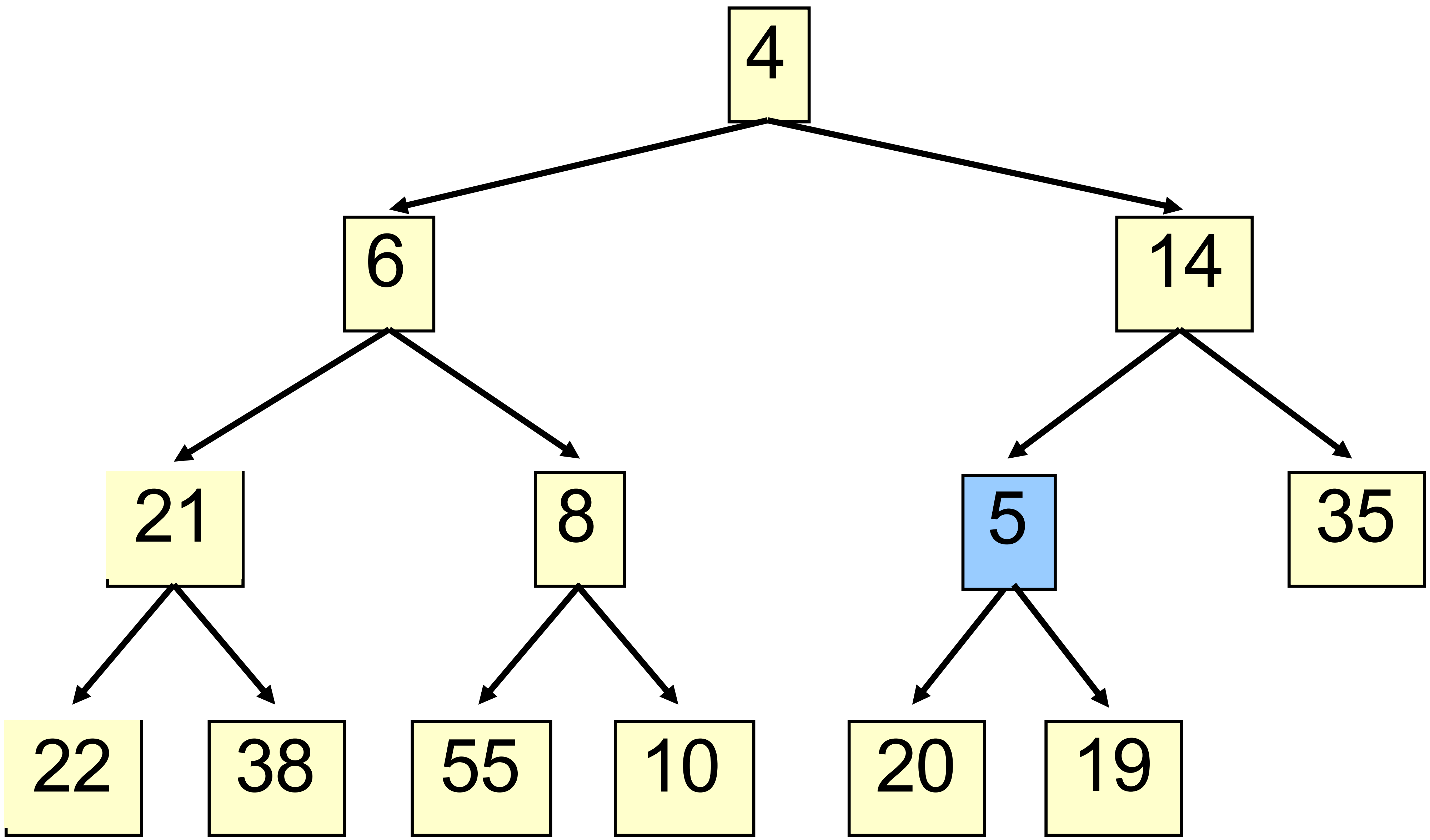
add(e)



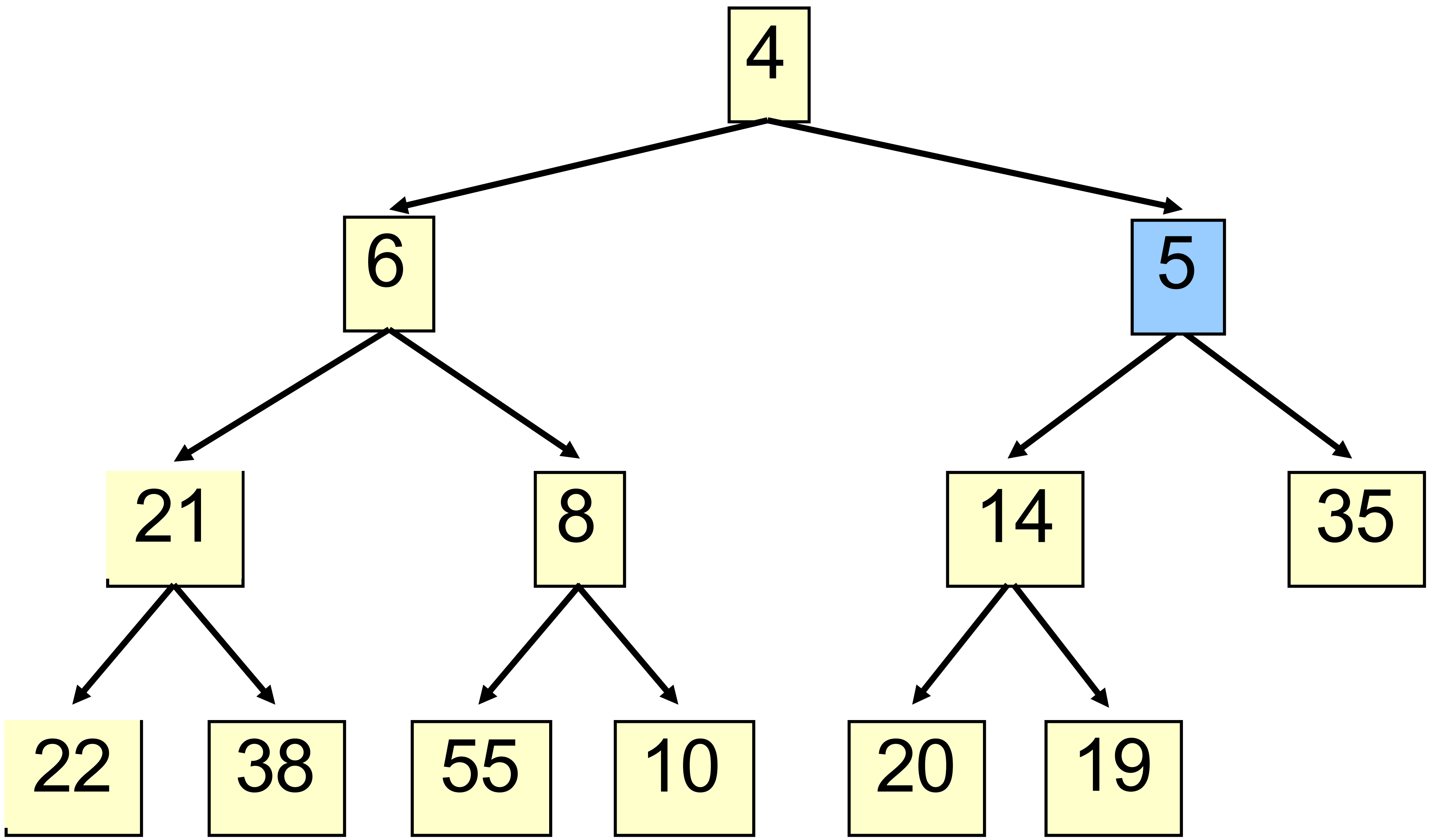
add(e)



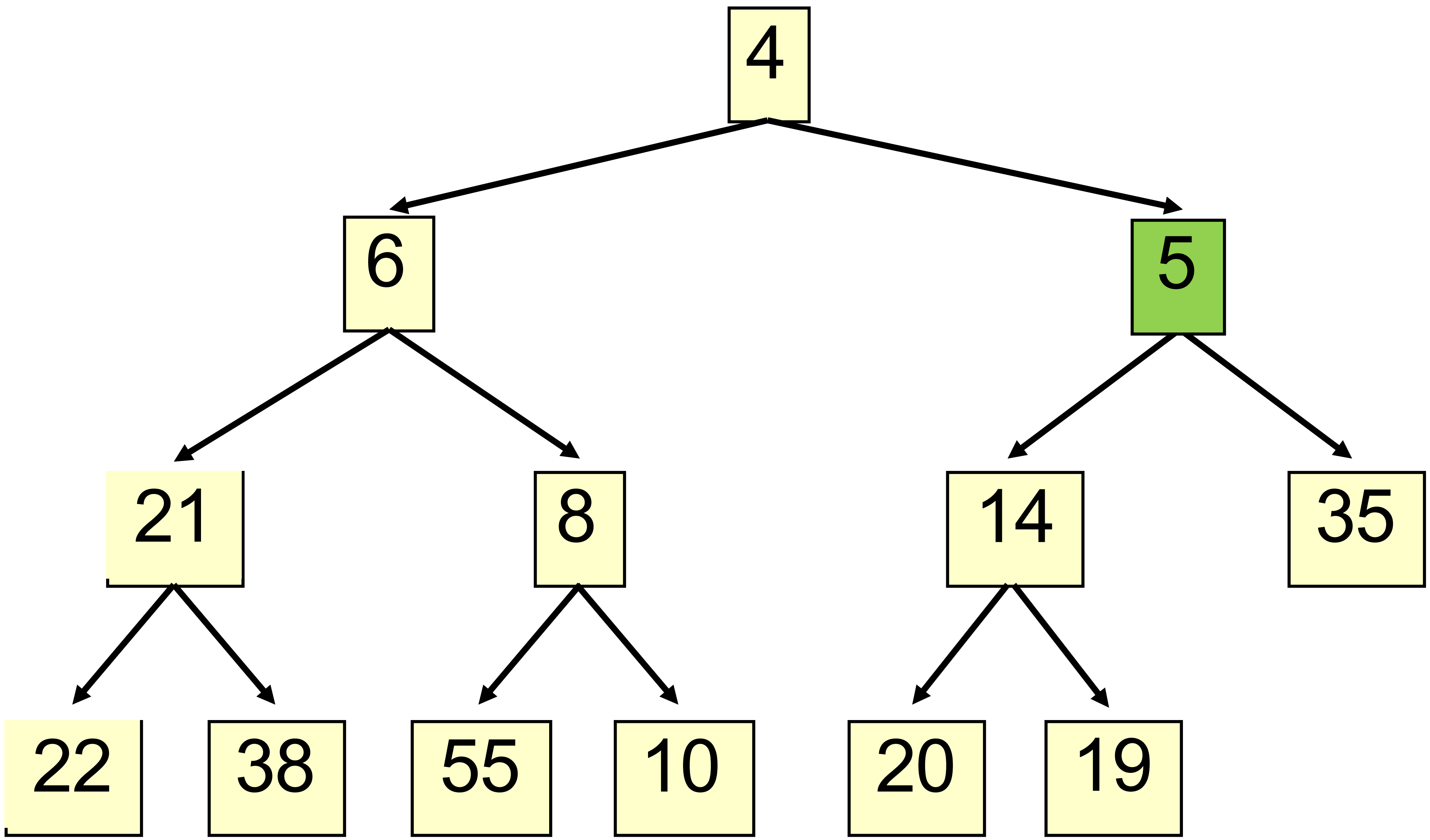
add(e)



add(e)



add(e)



add(e)

Algorithm:

- Add e in the wrong place (the leftmost empty leaf)
- While e is in the wrong place (it is less than its parent)
  - move e towards the right place (swap with parent)

The heap invariant is maintained!

What’s the runtime?

- O(number of swap/bubble operations)
  = O(height of tree)

- A **complete** tree must be **balanced** (can you prove this?)
  =>height is **O(log n)**

- Maximum number of swaps is **O(log n)**

add(e)

Algorithm:

- Add e in the wrong place (the leftmost empty leaf)
- While e is in the wrong place (it is less than its parent)
  - move e towards the right place (swap with parent)

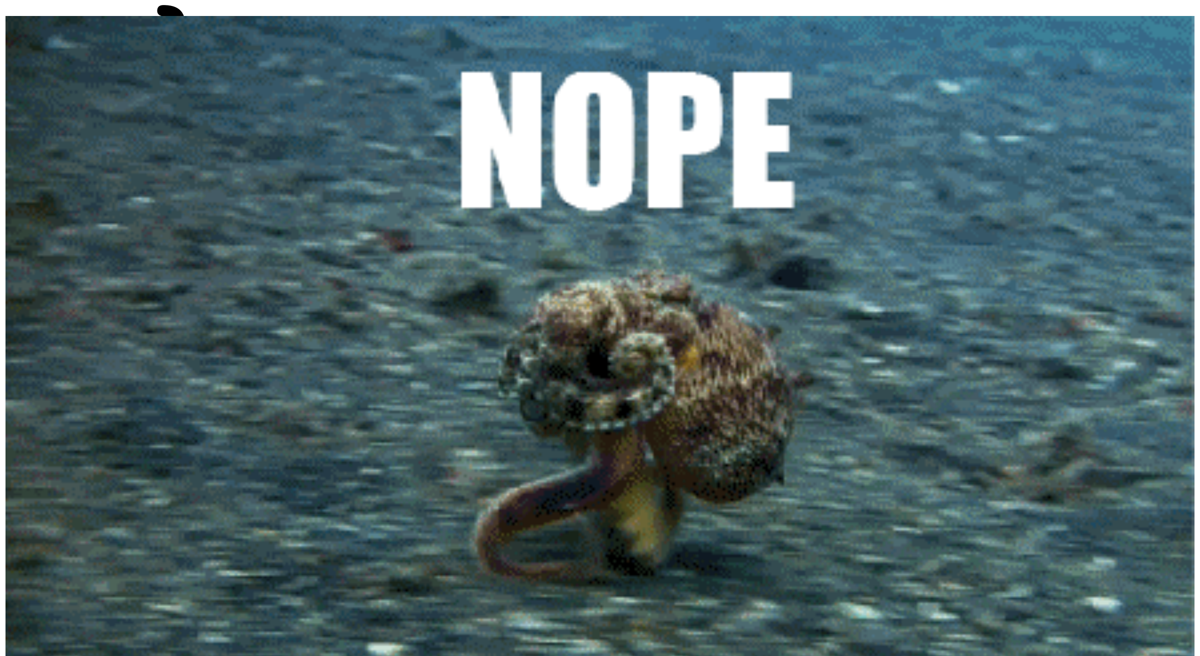
The heap property is maintained!

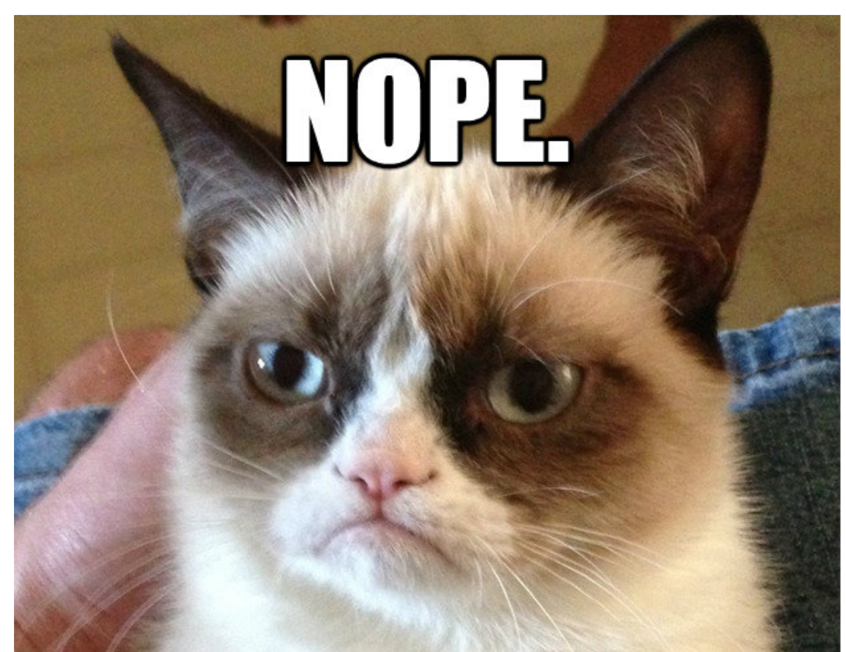
Implementing Heaps

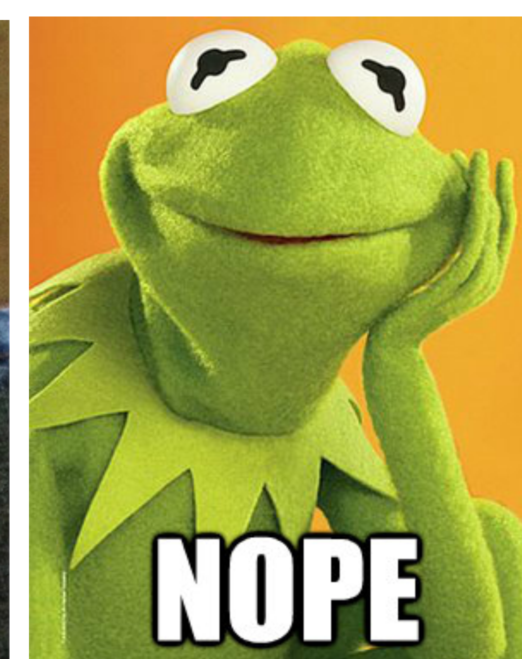
```
public class HeapNode {
   private int value;
   private HeapNode left;
   private HeapNode right;
   ...
}
public class Heap {
   HeapNode root;
   ...
```

Implementing Heaps

```
public class HeapNope {
   private int value;
   private HeapNope left;
   private HeapNope right;
   ...
}
```

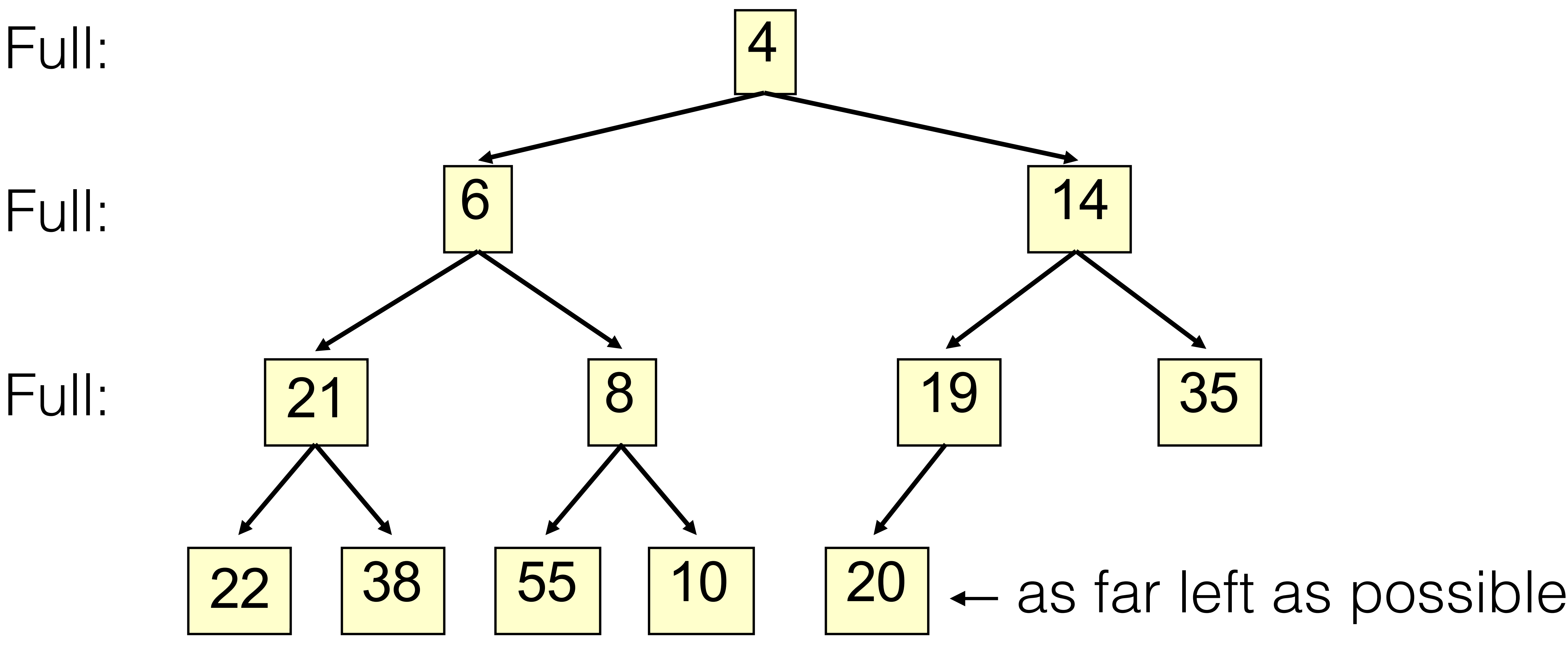






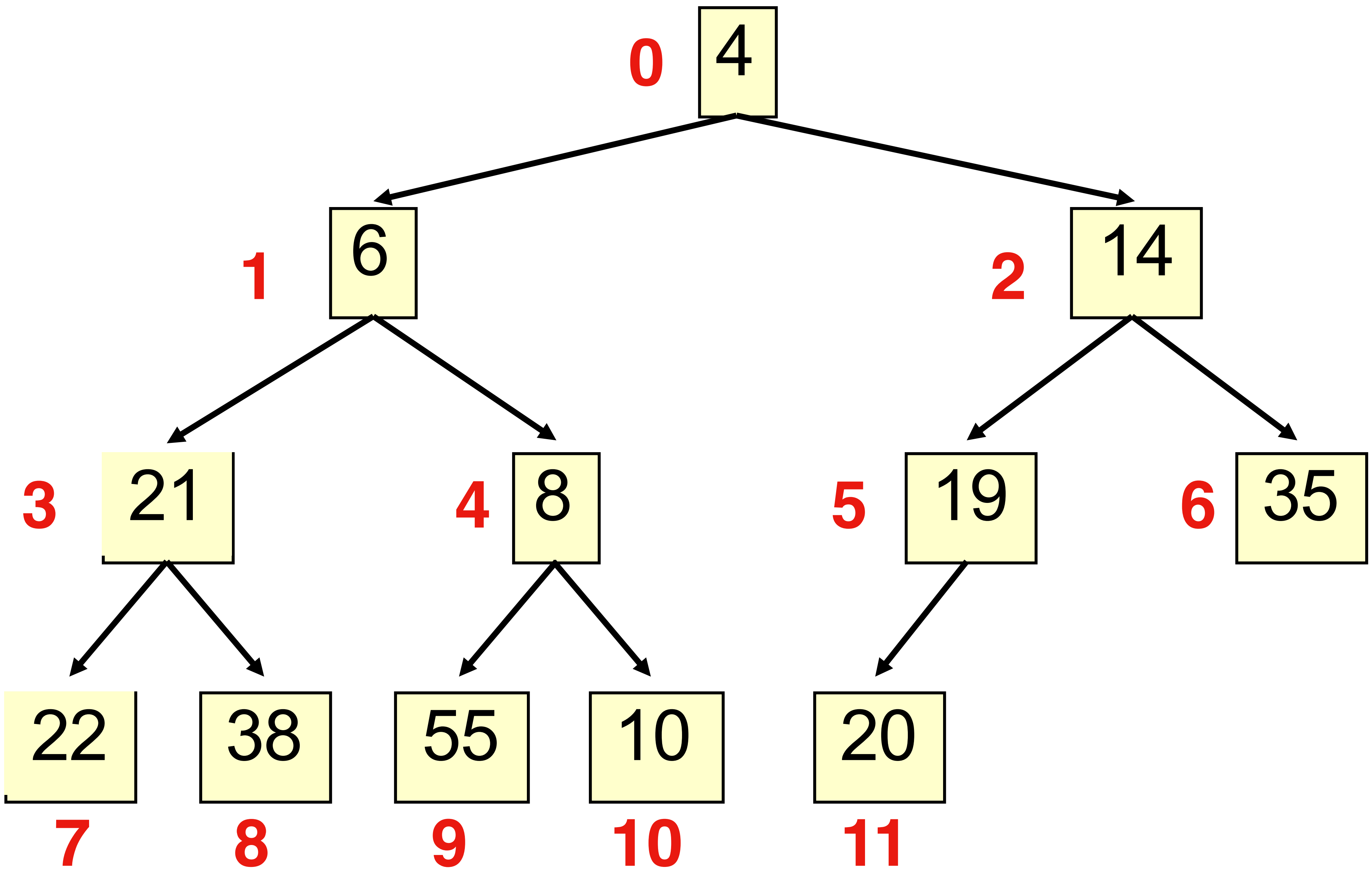
A heap is a special binary tree.

2. **Complete:** no holes!



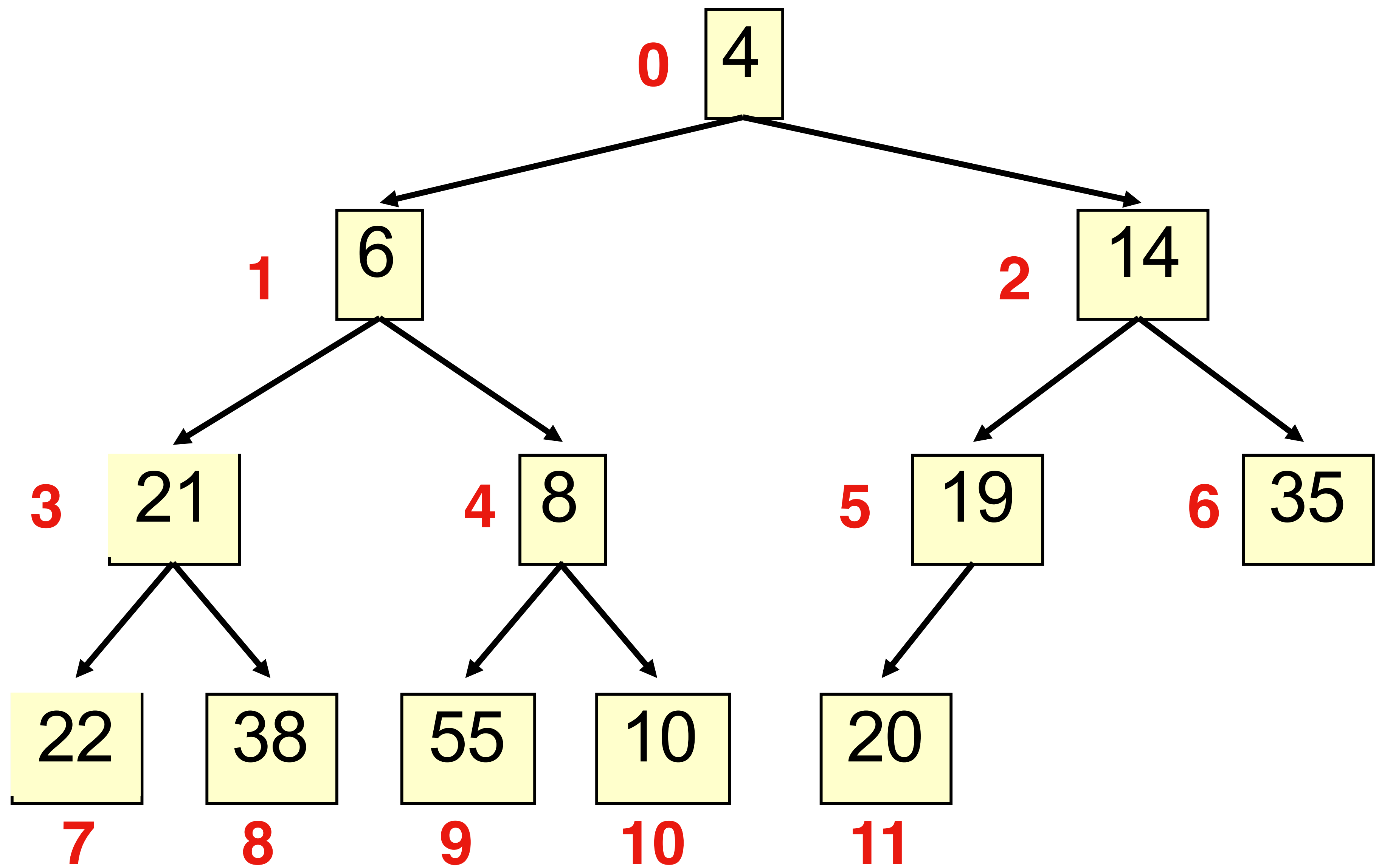
Numbering Nodes

Level-order traversal:



2. Complete: **no holes!**

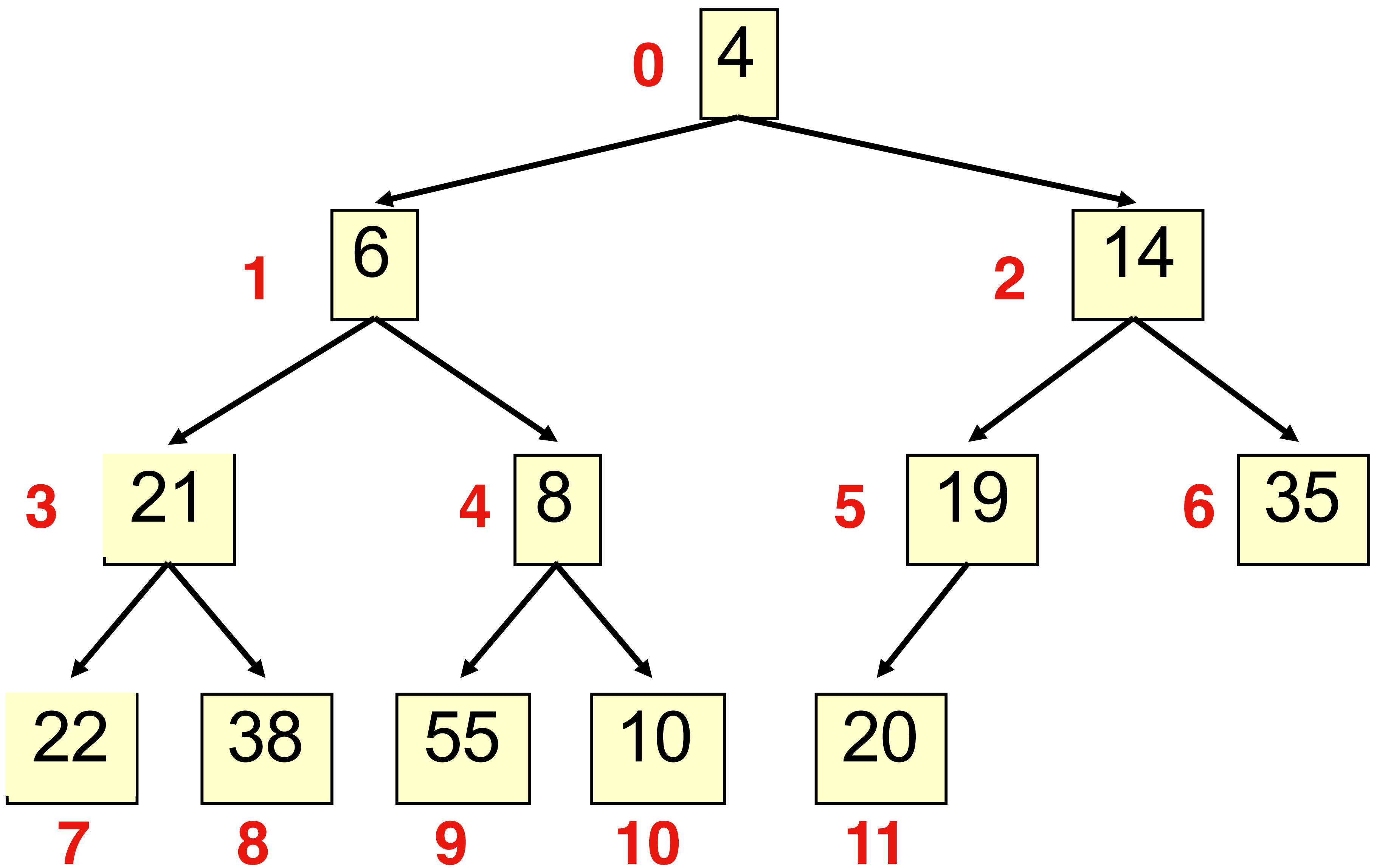
Numbering Nodes



node **k**'s parent is

node **k**'s children are nodes and

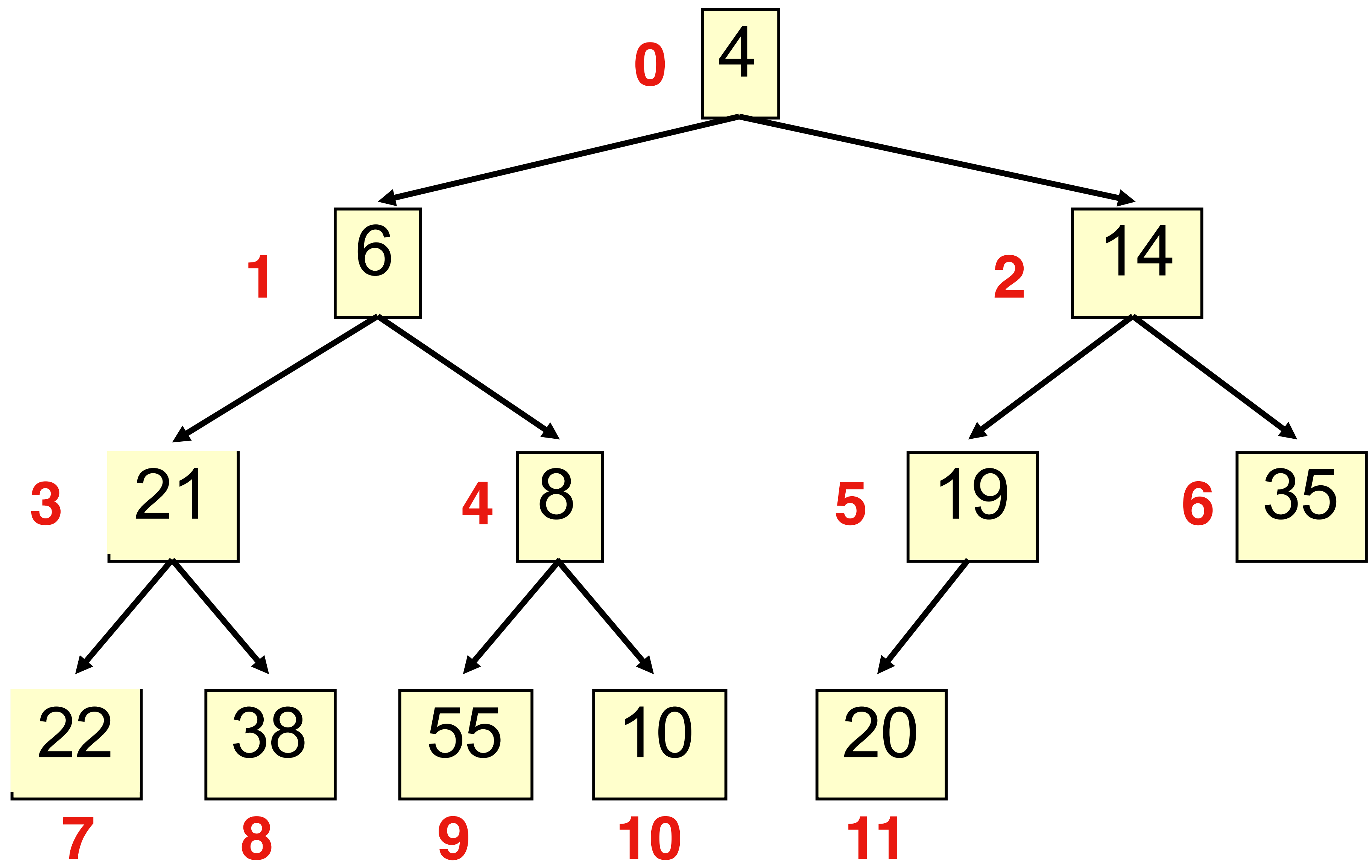
Numbering Nodes



node **k**'s parent is **(k – 1)/2**

node **k**'s children are nodes and

Numbering Nodes



node **k**'s parent is **(k – 1)/2**

node **k**'s children are nodes **2k + 1** and **2k + 2**

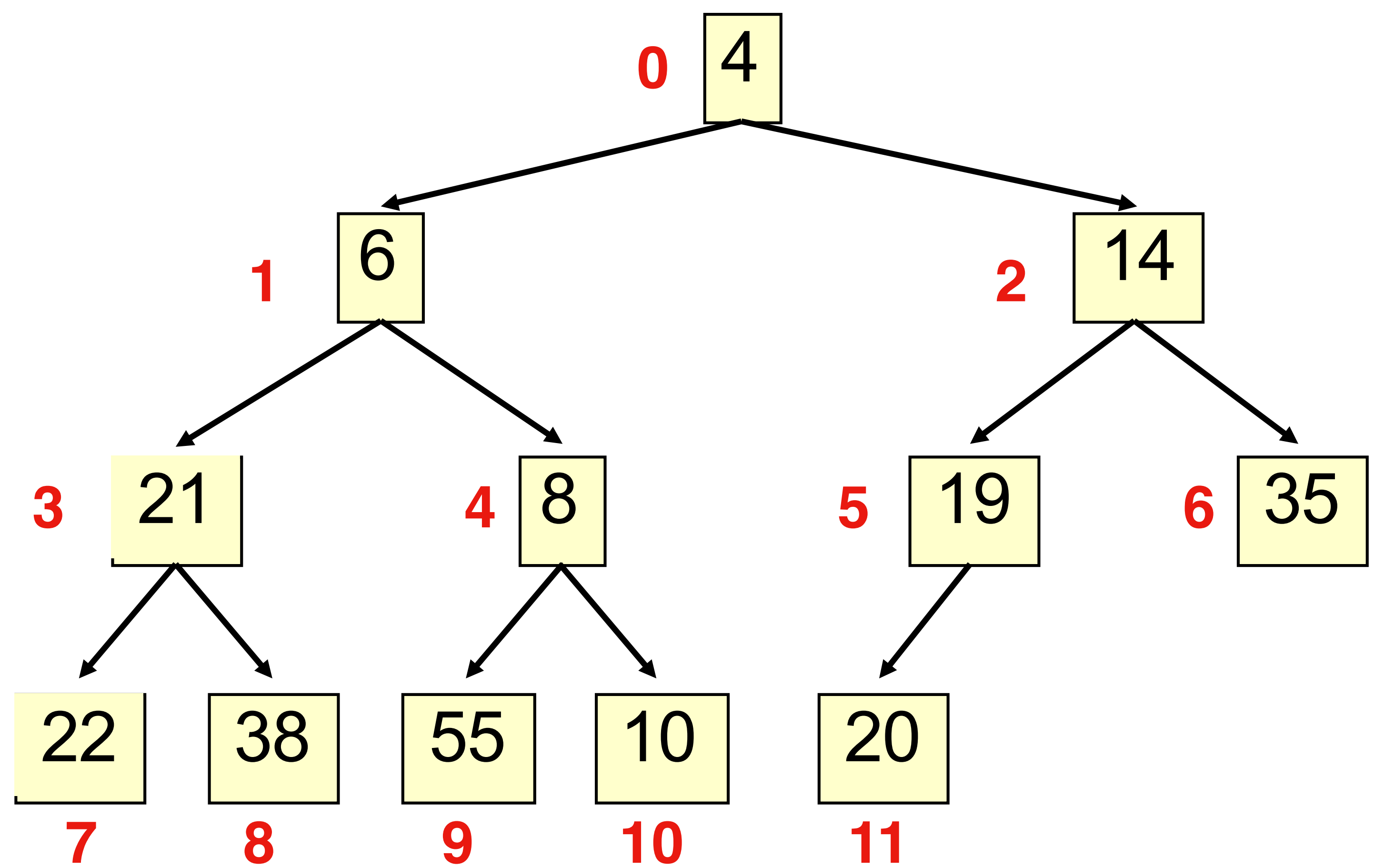
Implementing Heaps

```
public class Heap {
   private int[] heap;
   private int size;
   ...
}
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	6	14	21	8	19	35	22	38	55	10	20				

Implicit Tree Structure

2. Complete: **no holes!**



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	6	14	21	8	19	35	22	38	55	10	20				

Heap it real, part 2.

Here's a heap, stored in an array:

[1 5 7 6 7 10]

Which of the following is the correct array after execution of **add(4)**?

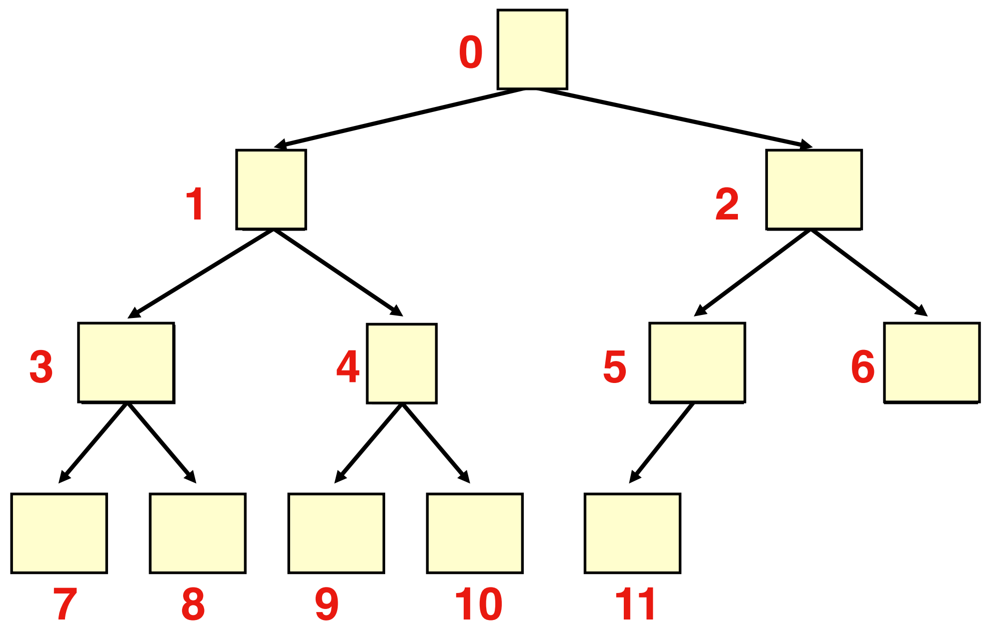
Assume the array has space for the additional element (i.e., doesn't need to grow).

A. [1 5 4 6 7 10 7]

B. [1 5 7 6 7 10 4]

C. [1 4 5 7 6 7 10]

D. [1 5 7 6 4 7 10]



Heap operations

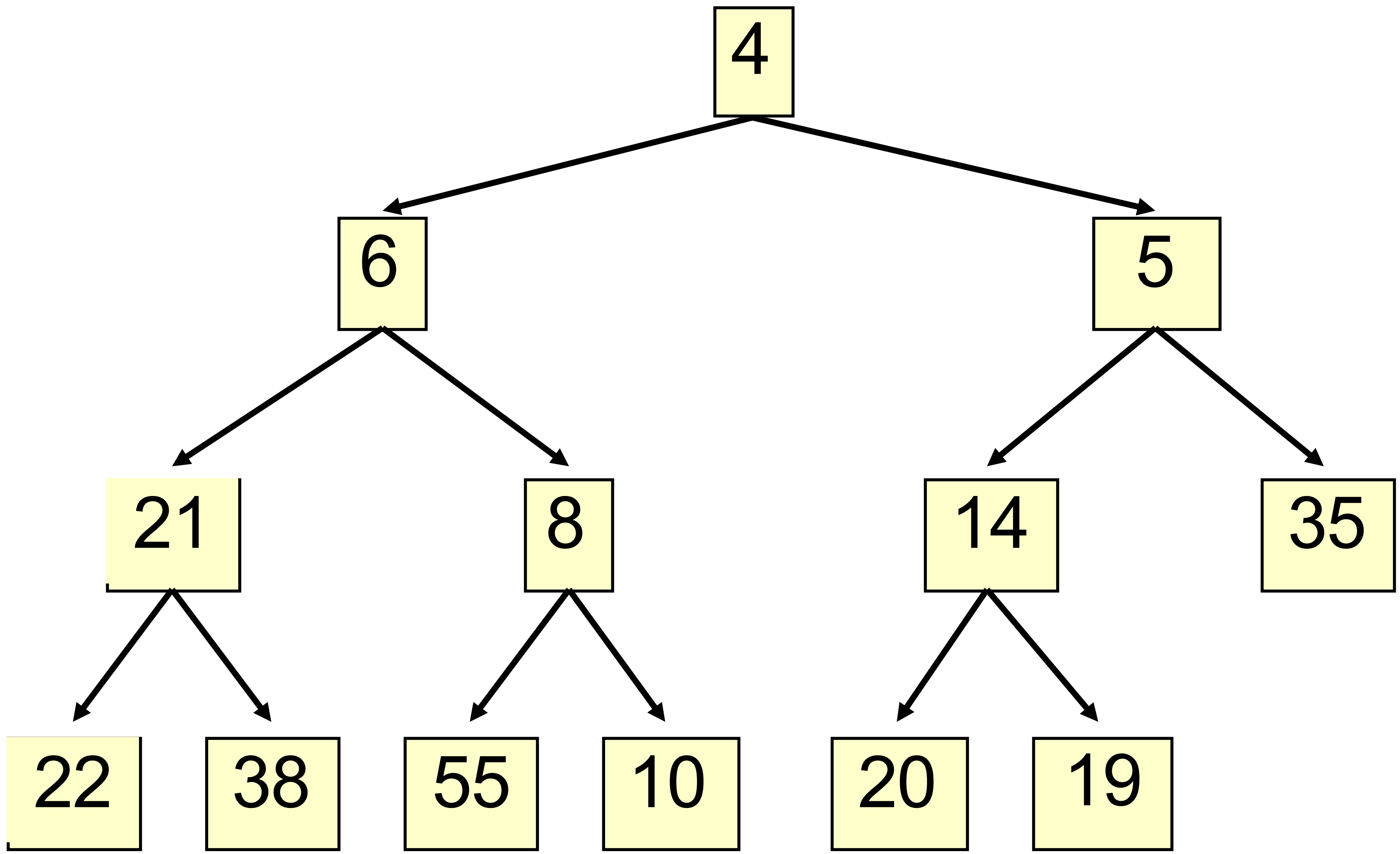
```
interface PriorityQueue {
 boolean add(Object e); // insert e
 Object peek(); // return min element
 Object poll(); // remove/return min element
 void clear();
 boolean contains(Object e);
 boolean remove(Object e);
 int size();
 Iterator iterator();
}
```

poll()

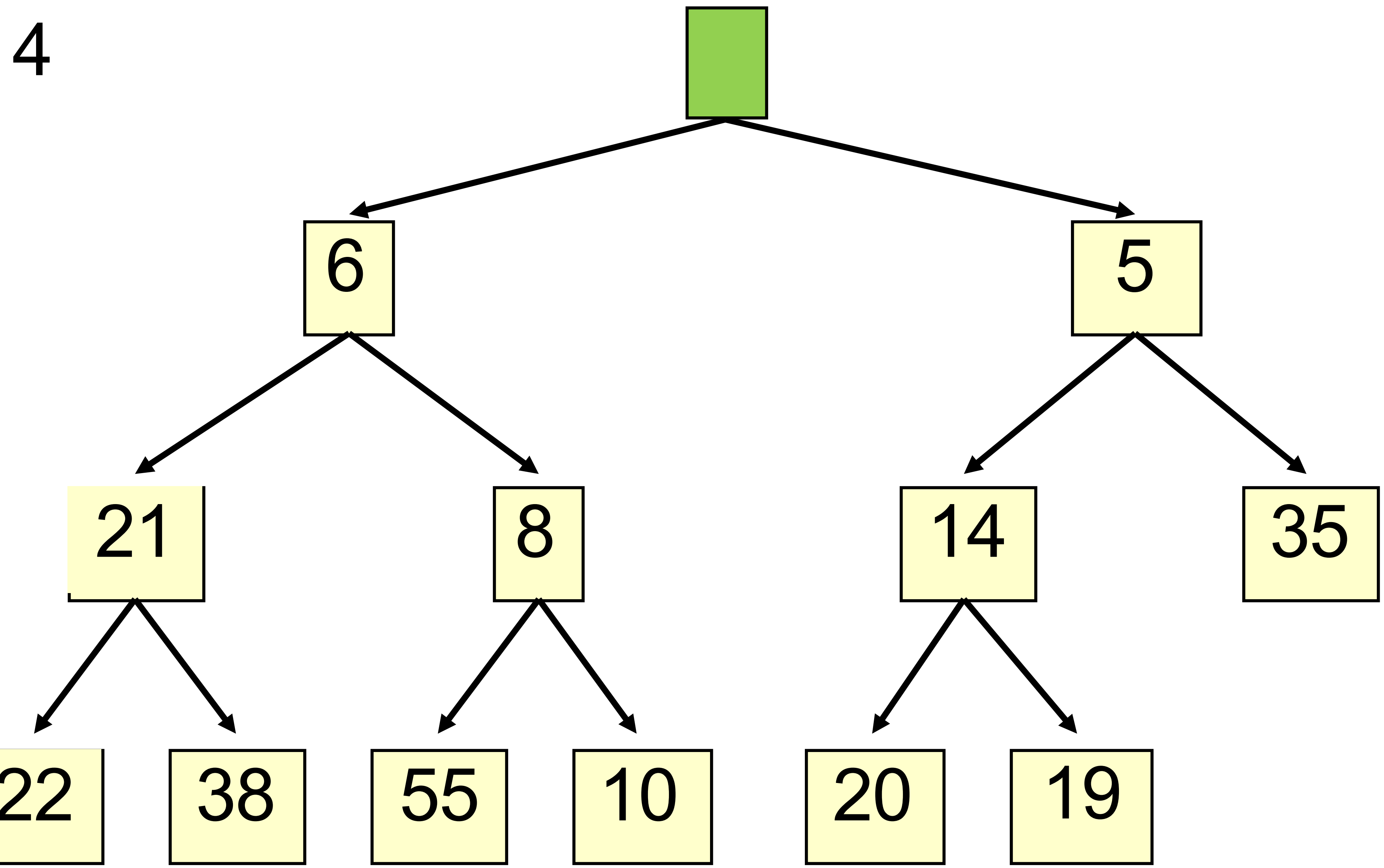
Algorithm:

- Remove and save the smallest thing
- Fill the resulting hole with the wrong thing
- Bubble the wrong thing down to the right place

poll()

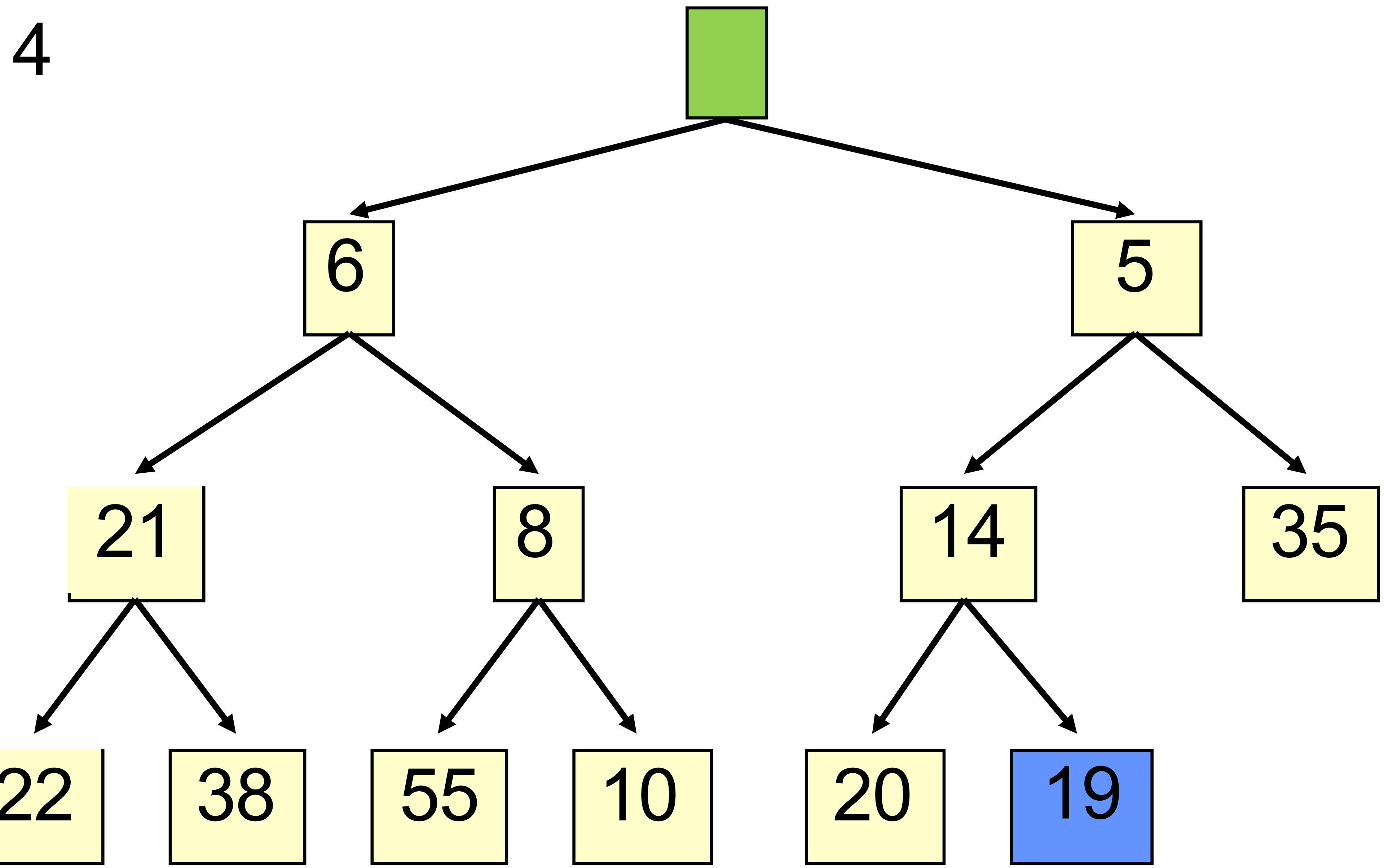


poll()



Remove and save the smallest (root) element

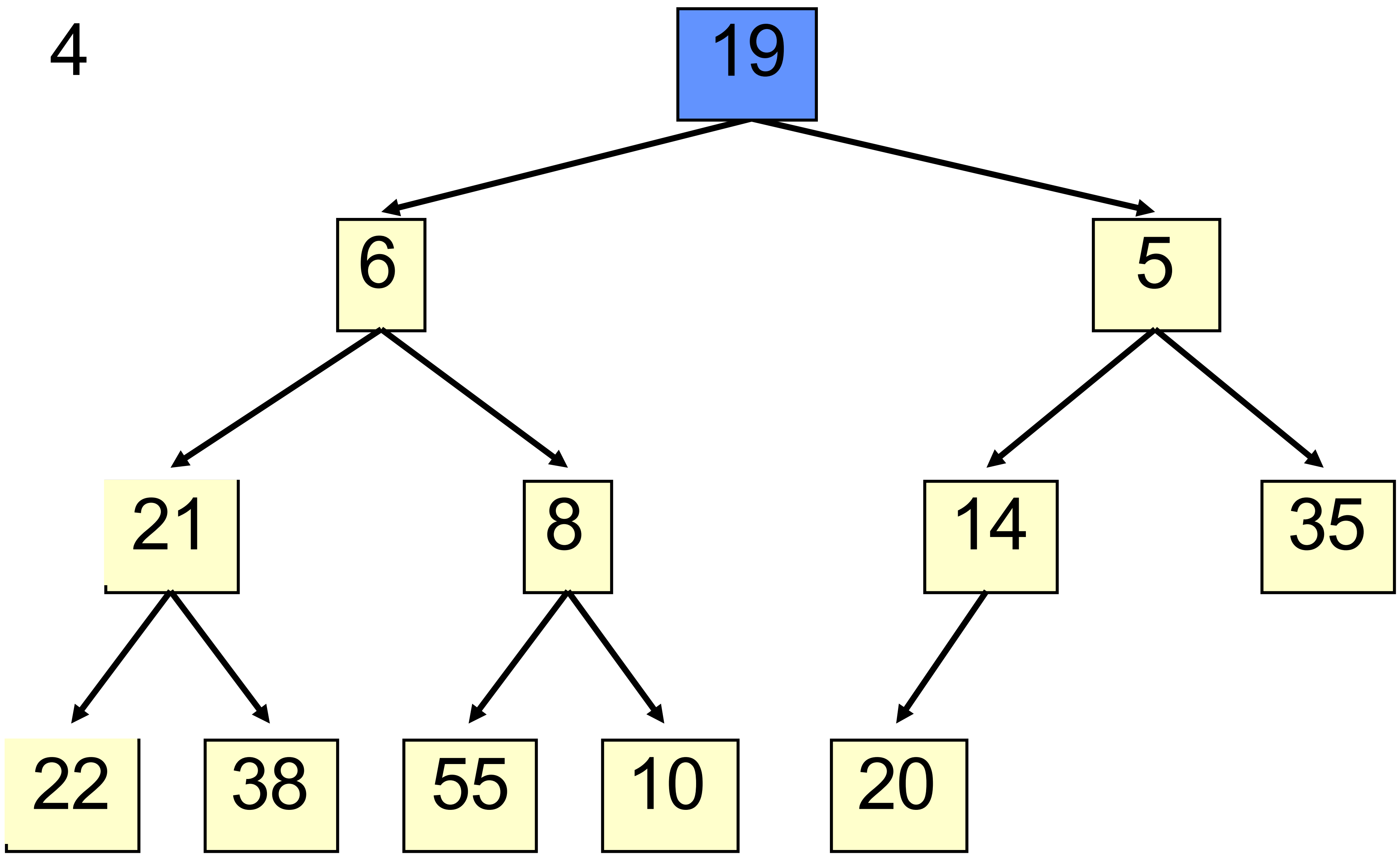
poll()



Move the last element to replace the root

poll()

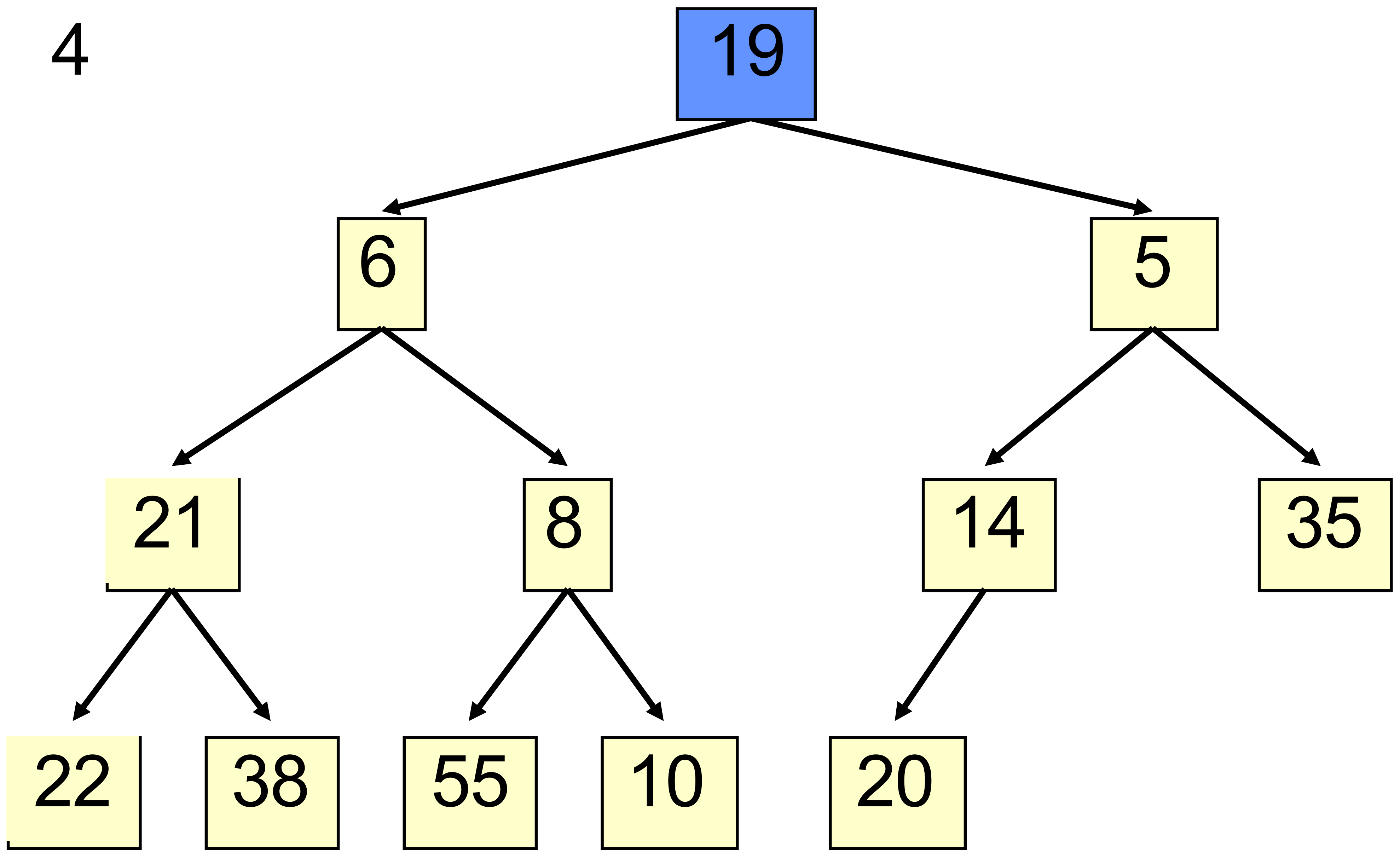
4



Bubble the root value down

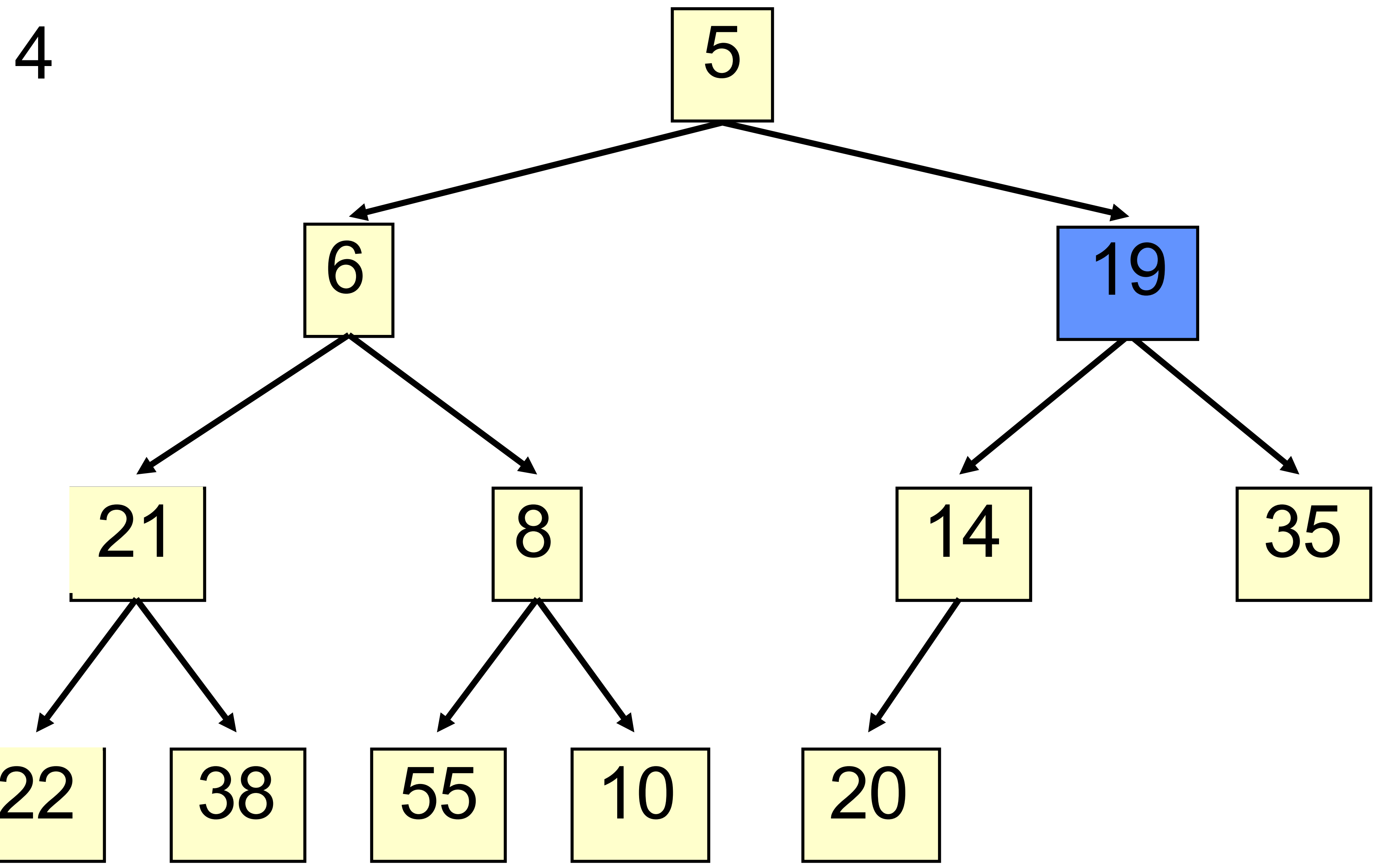
poll()

4



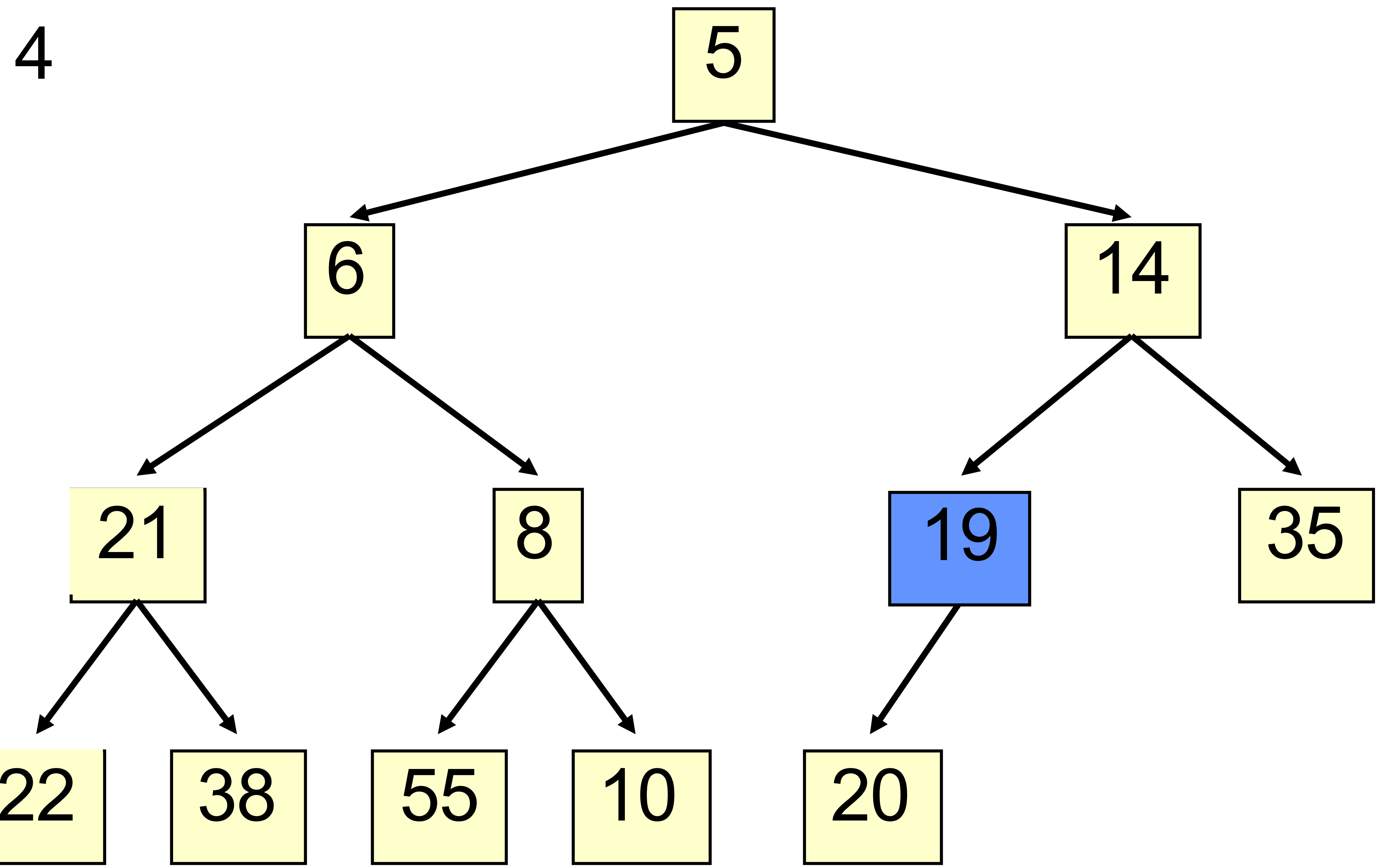
Bubble the root value down, swapping with the **smaller** child

poll()



Bubble the root value down, swapping with the **smaller** child

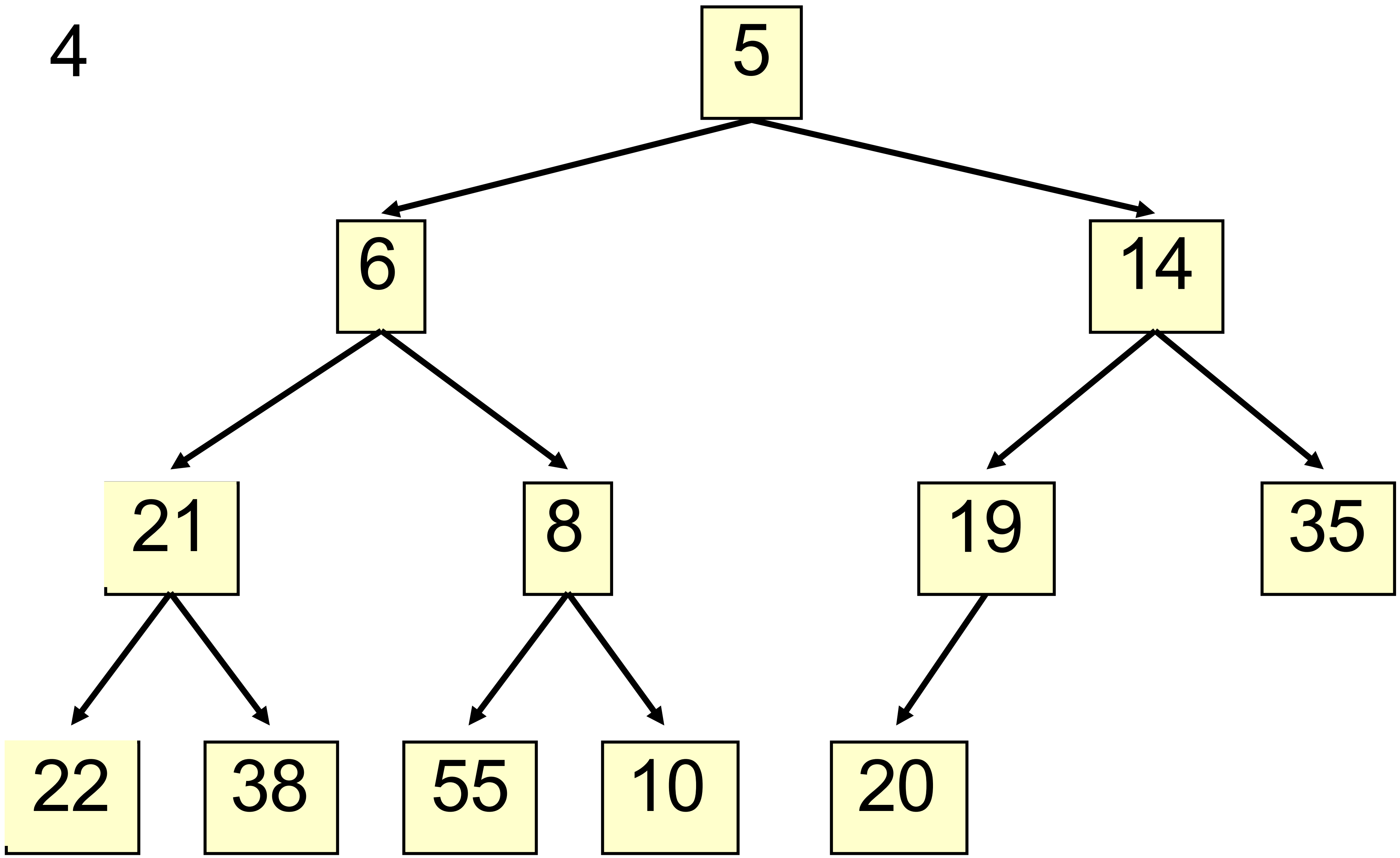
poll()



Bubble the root value down, swapping with the **smaller** child

poll()

4



Return the smallest element.

poll()

Algorithm:

- Remove and save the root (first) element
- Move the last element to the first spot.
- While it is greater than either of its children:
  - Swap it with its smaller child.

Heap operations

```
interface PriorityQueue {
 boolean add(Object e); // insert e  O(log n)
 Object peek(); // return min        O(1)
 Object poll(); // remove/return min O(log n)
 void clear();                       O(1)
 boolean contains(Object e);         O(n)
 boolean remove(Object e);           O(n)
 int size();                         O(1)
 Iterator iterator();                O(1)
}
```

Details

- Grow the storage array when the heap exceeds its size (could use ArrayList)
- Implementation of bubbling routines
- Implementation of contains() and remove()
- Min vs max heaps
- Efficiently find, remove, and change priority

Heapsort

```
public static void heapsort(int[] b) {
  Heap h = new Heap();
  // put everything into a heap - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    h.add(b[k]);
  }

  // pull everything out in order - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    b[k] = h.poll();
  }
}
```

Heapsort

```
public static void heapsort(int[] b) {
  Heap h = new Heap();
  // put everything into a heap - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    h.add(b[k]);
  }

  // pull everything out in order - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    b[k] = h.poll();
  }
}
```

Worst-case runtime: O(n log n) !

Heapsort

```
public static void heapsort(int[] b) {
  Heap h = new Heap();
  // put everything into a heap - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    h.add(b[k]);
  }

  // pull everything out in order - n*log(n)
  for (int k = 0; k < b.length; k = k+1) {
    b[k] = h.poll();
  }
}
```

Possible to implement **in-place!**

Worst-case runtime: O(n log n) !

Recap - what we know now:

- The two special properties that make a heap.
- How to store a complete binary tree in an array.
- How to add an element to a heap.
- How to remove the smallest element from a heap.
- How to write a worst-case O(n log n) sort.