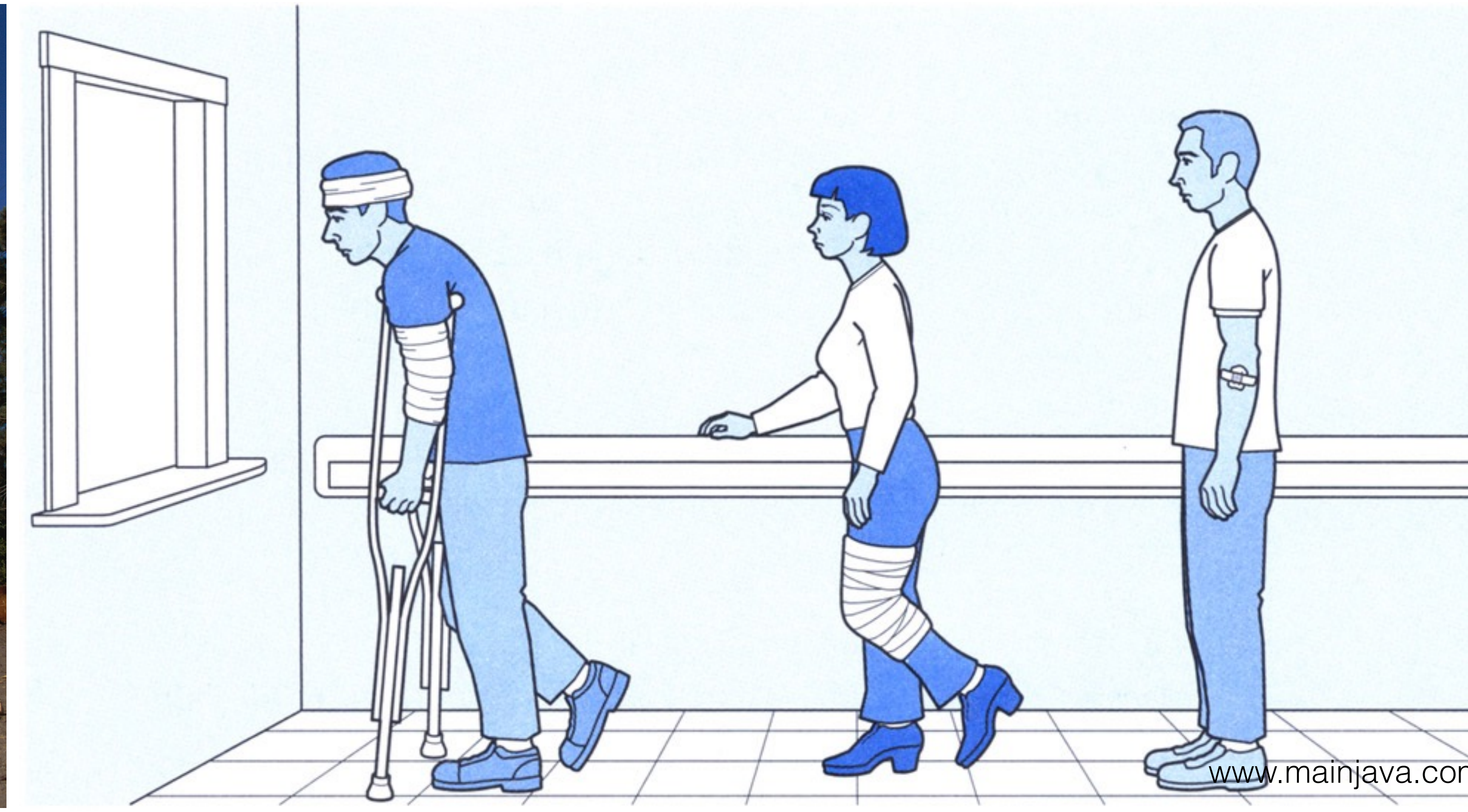



# CSCI 241

## Lecture 14
## Heaps and the Priority Queue ADT, Continued

# Happenings

- Wednesday, 2/20 – Peer Lecture Series: Unity Workshop
  5 pm in CF 420
- Wednesday, 2/20 – CS Research Info Session
  5 pm in CF 105
- Wednesday, 2/20 – Grace Hopper Info Panel
  5 pm in AW 203
- Thursday, 2/21 – CSCI Faculty Candidate: Research Talk
  4 pm in **CF 226**
- Friday, 2/22 – CSCI Faculty Candidate: Teaching Talk
  4 pm in **CF 227**
- Saturday & Sunday, 2/23 – 2/24 – Winter Game Jam
  10 am – 10 pm in CF 105, 162, 164

# Announcements

- No quiz this Friday!

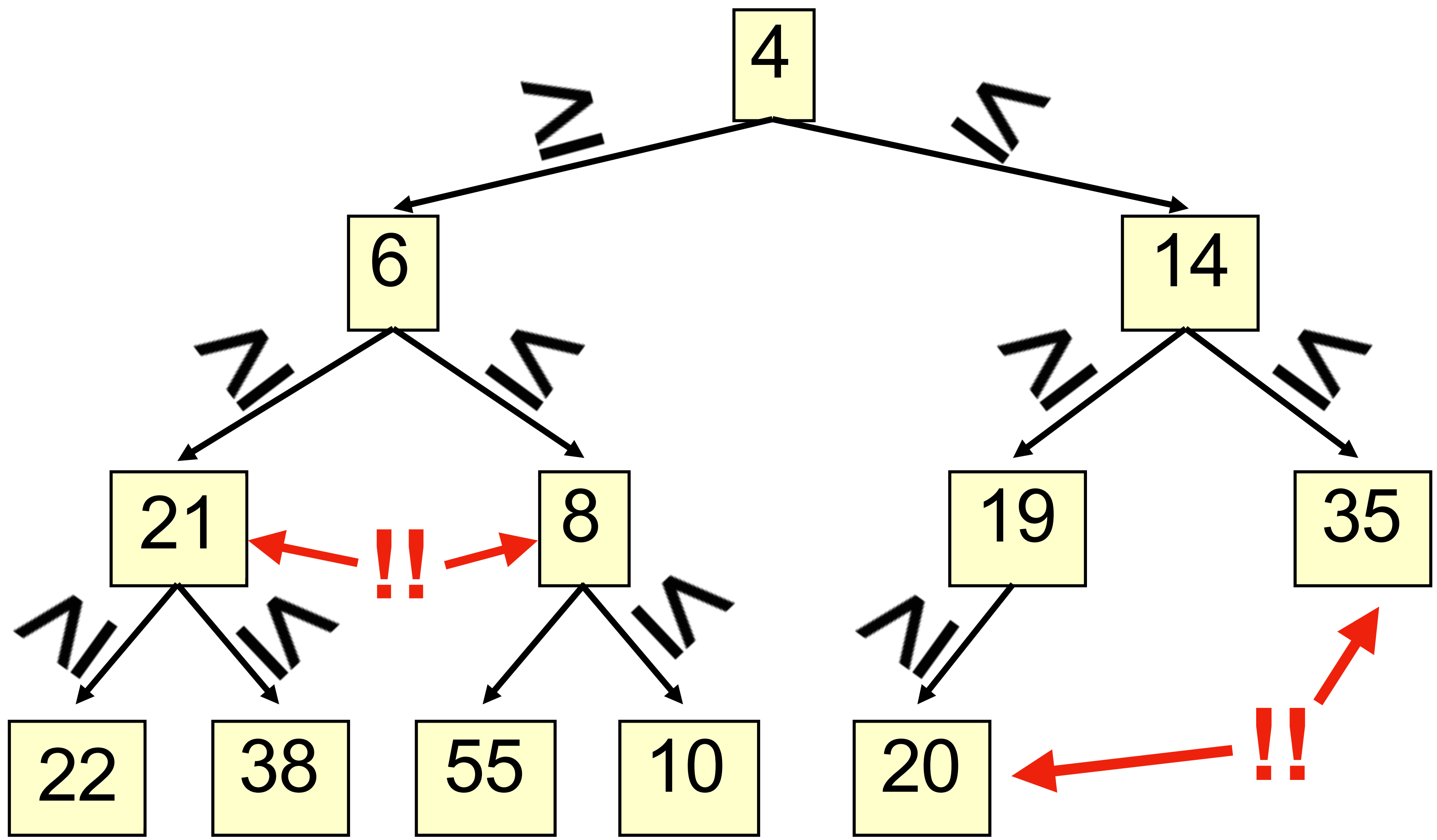
# Goals

- Know the definition and properties of a heap.
- Know how heaps are stored in practice.
- Know how to implement the `add`, `peek`, and `poll` heap operations.
- Understand the purpose and interface of the Priority Queue ADT.
- Understand how to implement a Priority Queue using a heap

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

# A heap is a special binary tree.

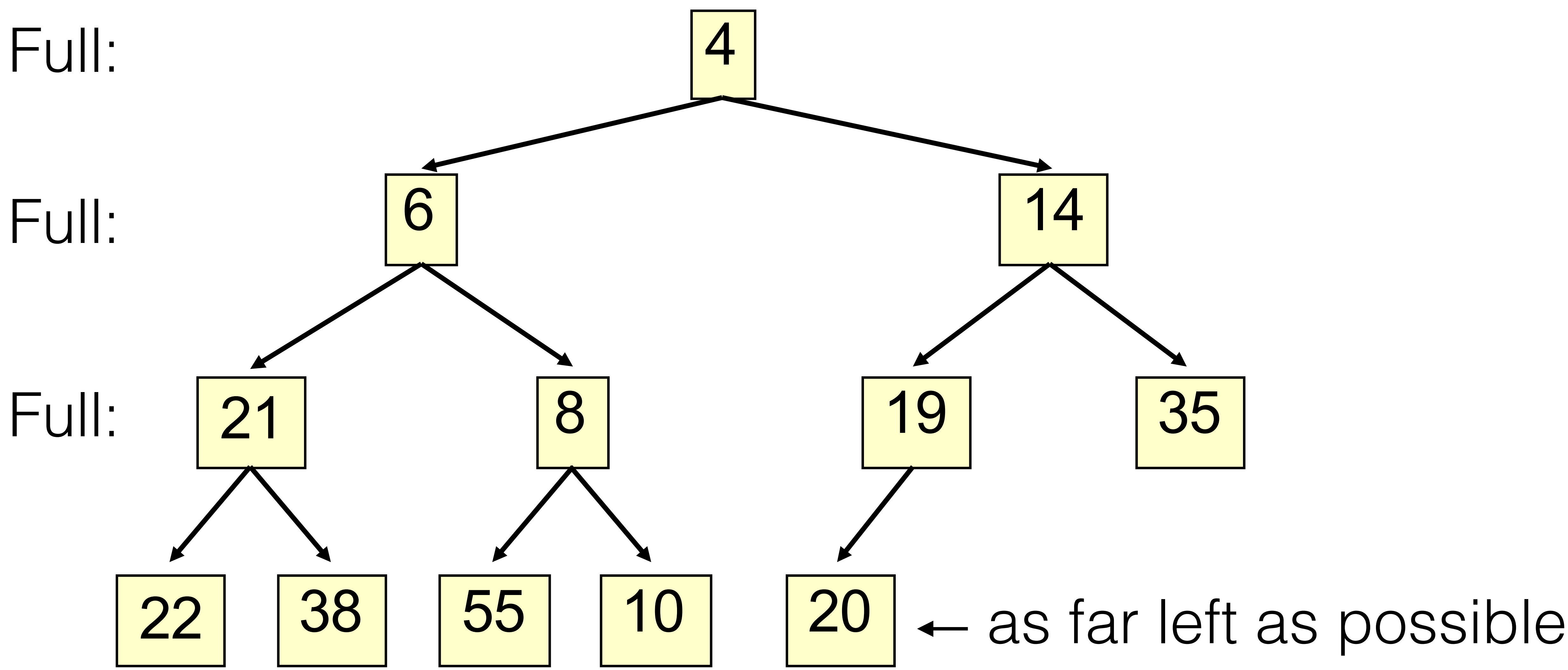
1. **Heap Order Invariant:**
Each element >= its parent.



# 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

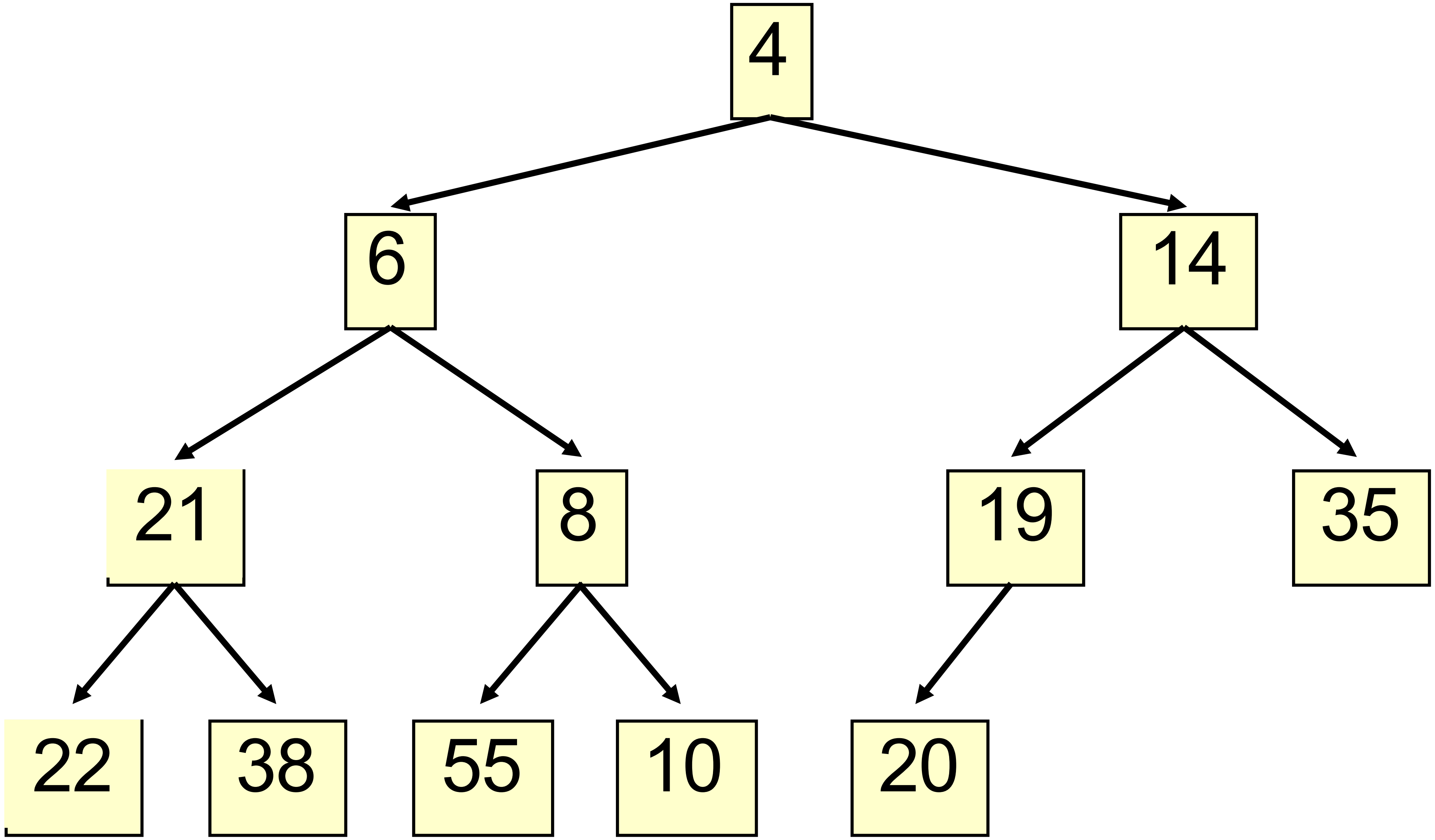
```
interface PriorityQueue<V v, P p> {
 // insert value v with priority p
 void add(V v, P p);

 // return value with min priority
 V peek();

 // remove/return value with min priority
 V poll();

 // more methods…
}
```

```
void add(V v, P p);
```

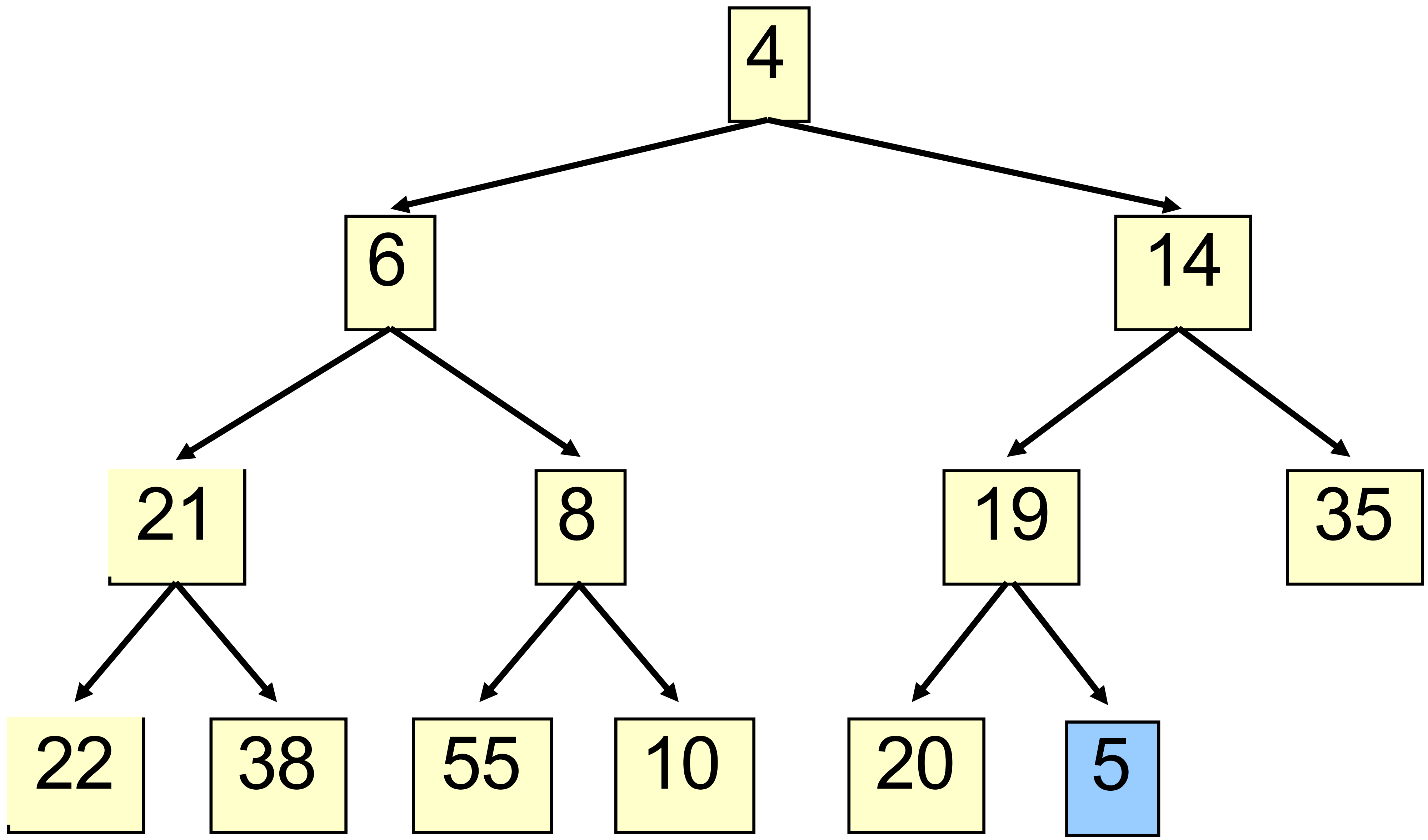
**Algorithm:**

- Add v in the wrong place
- While v is in the wrong place
  - move v towards the right place

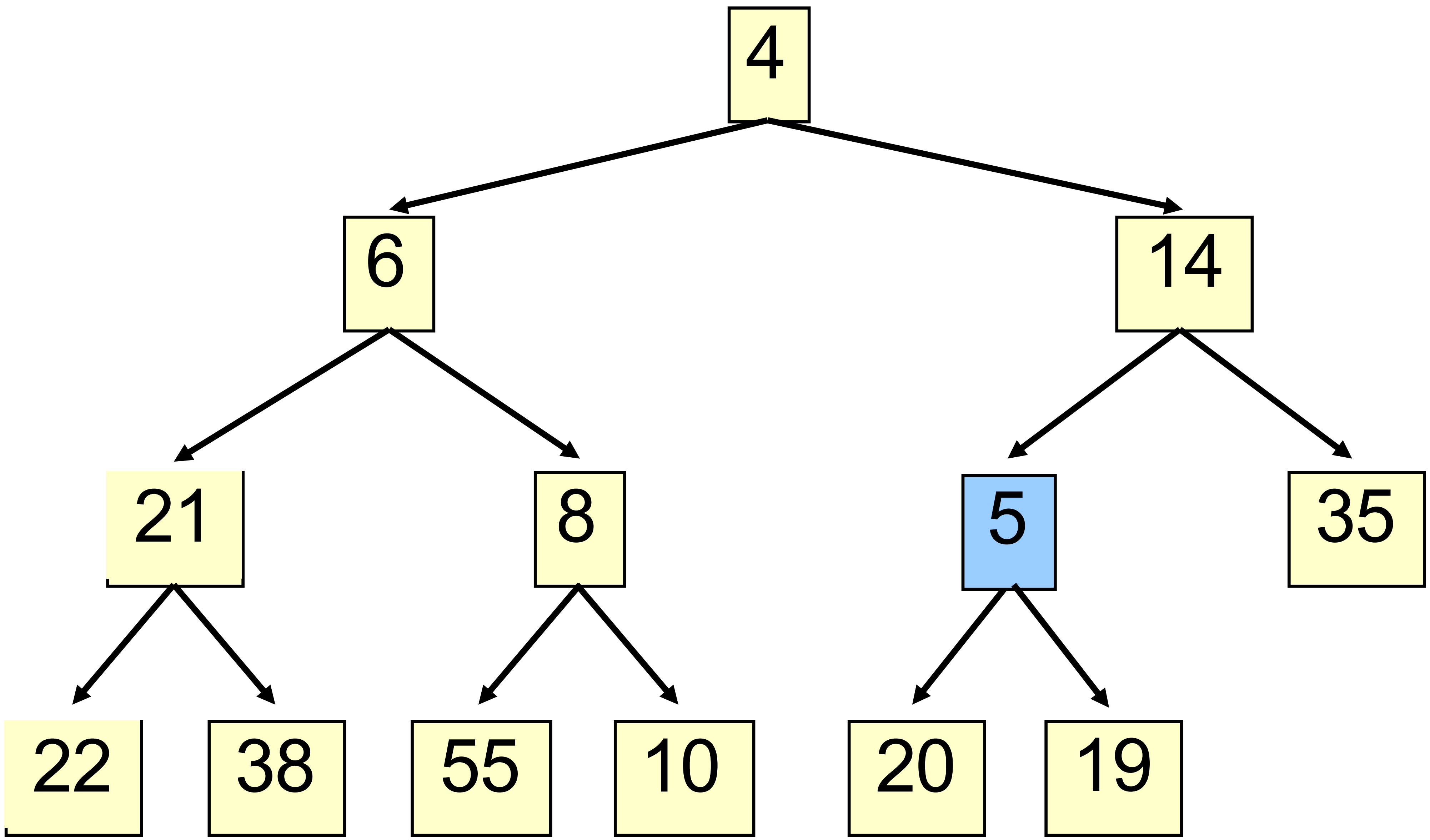
```
void add(V v, P p);
```



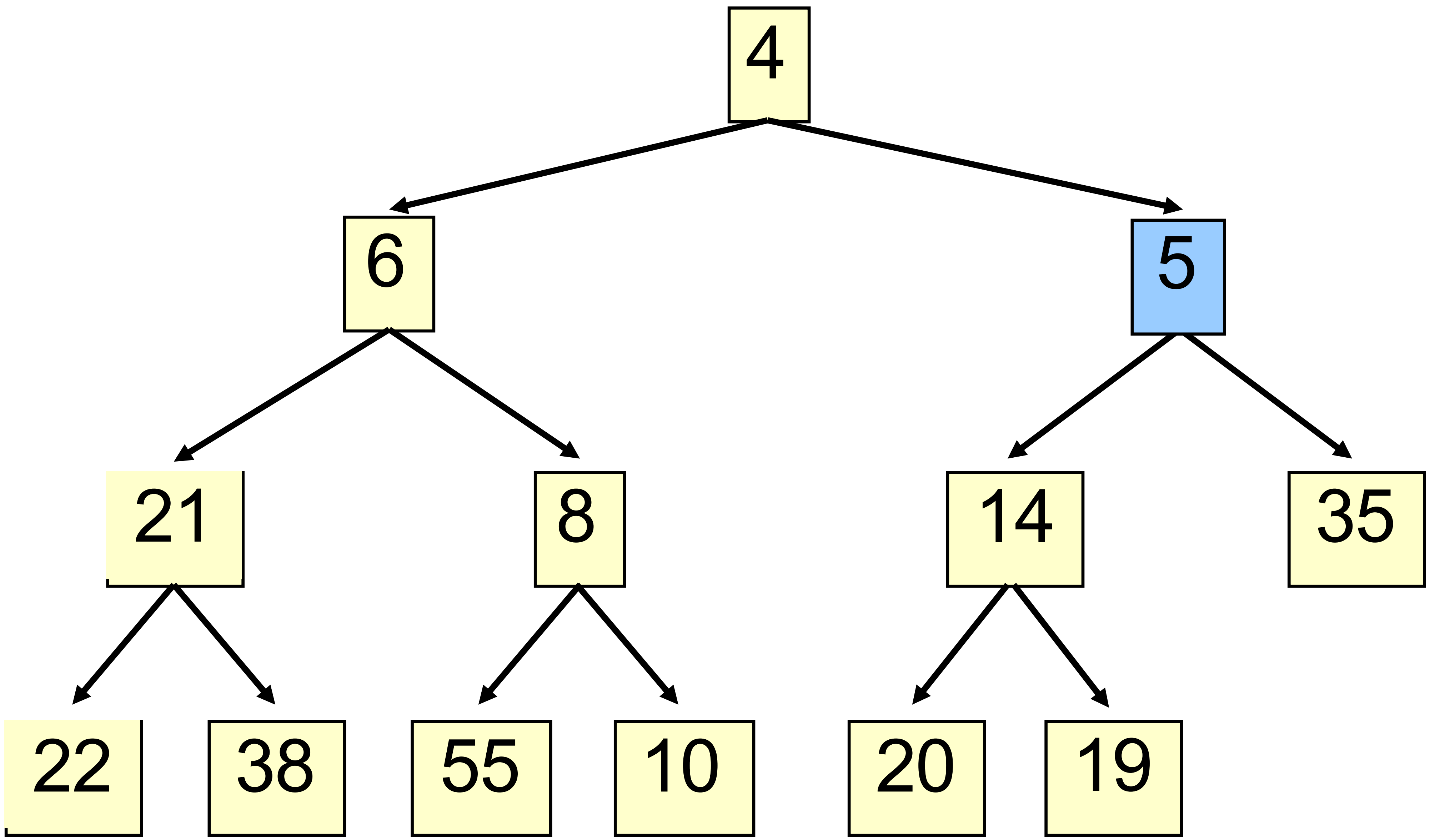
```
void add(V v, P p);
```



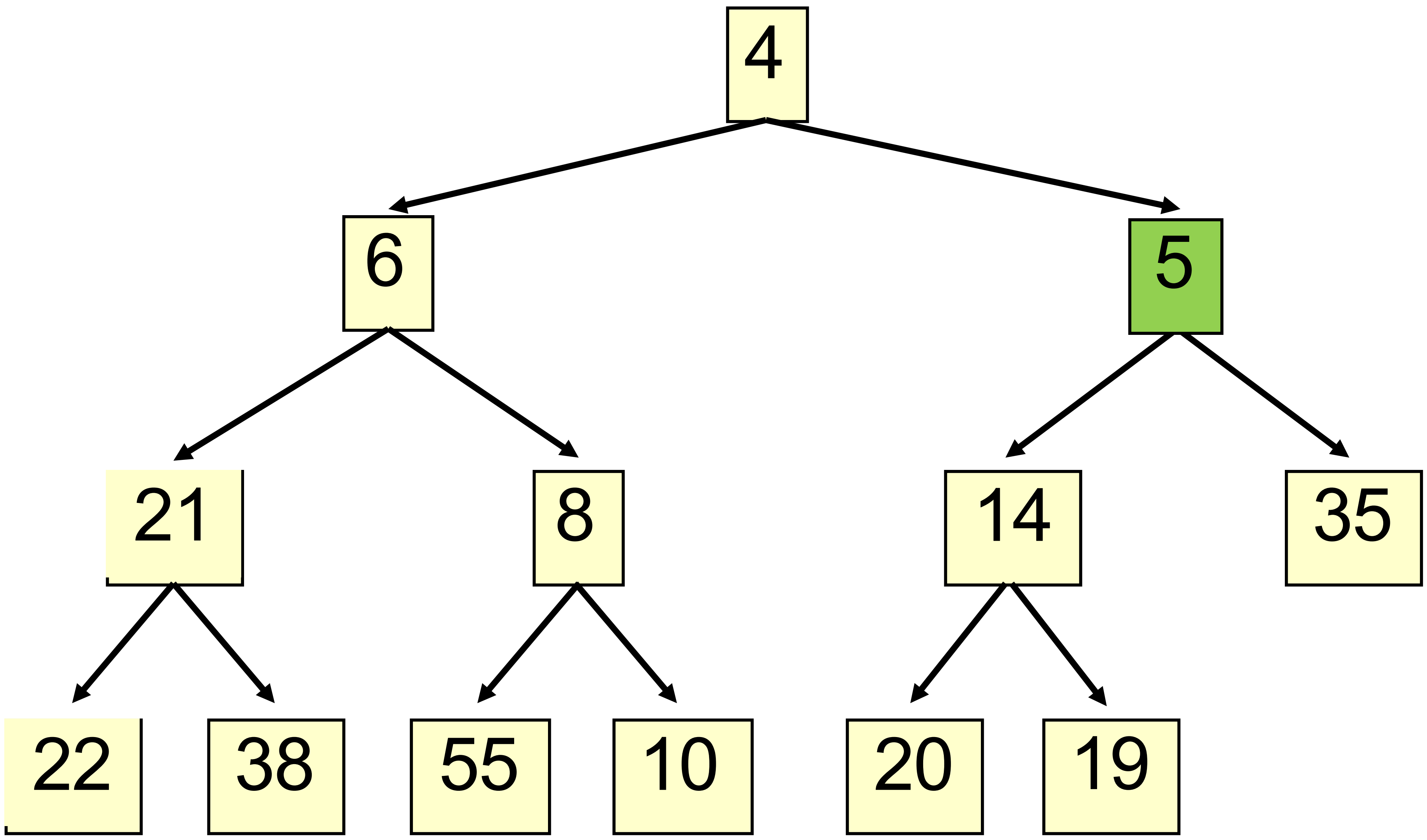
```
void add(V v, P p);
```



```
void add(V v, P p);
```



```
void add(V v, P p);
```



```
void add(V v, P p);
```

**Algorithm:**

- Add v in the wrong place (the leftmost empty leaf)
- While v is in the wrong place (its p is less than its parent's)
  - move v towards the right place (swap with parent)

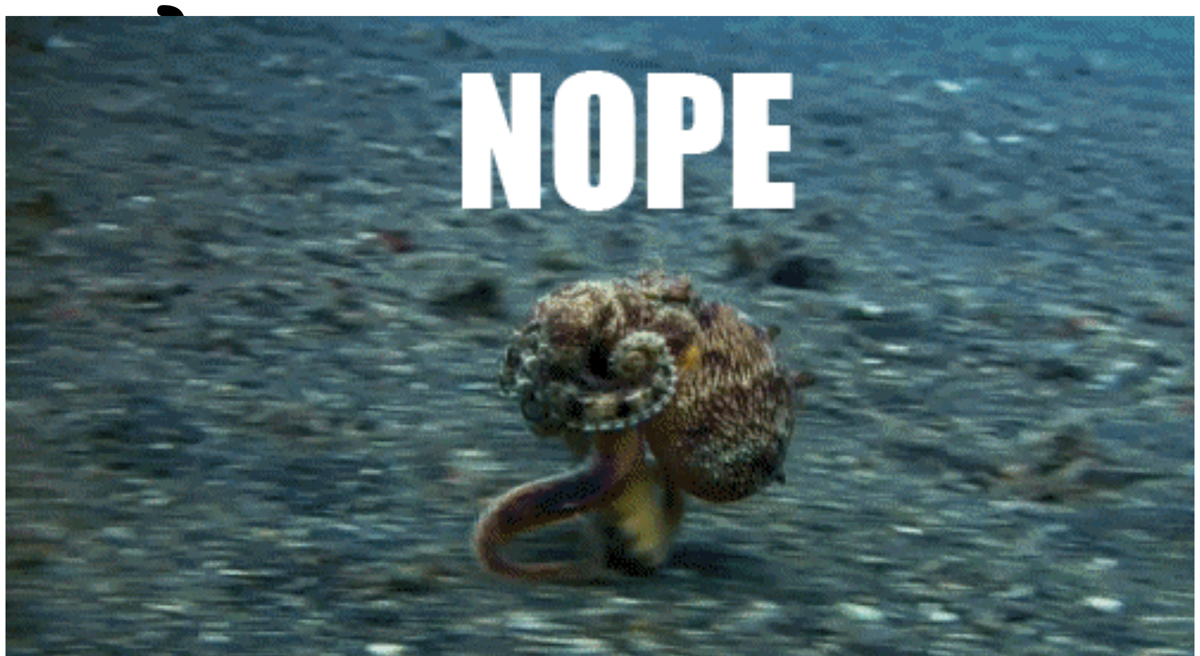
The heap invariant is maintained!

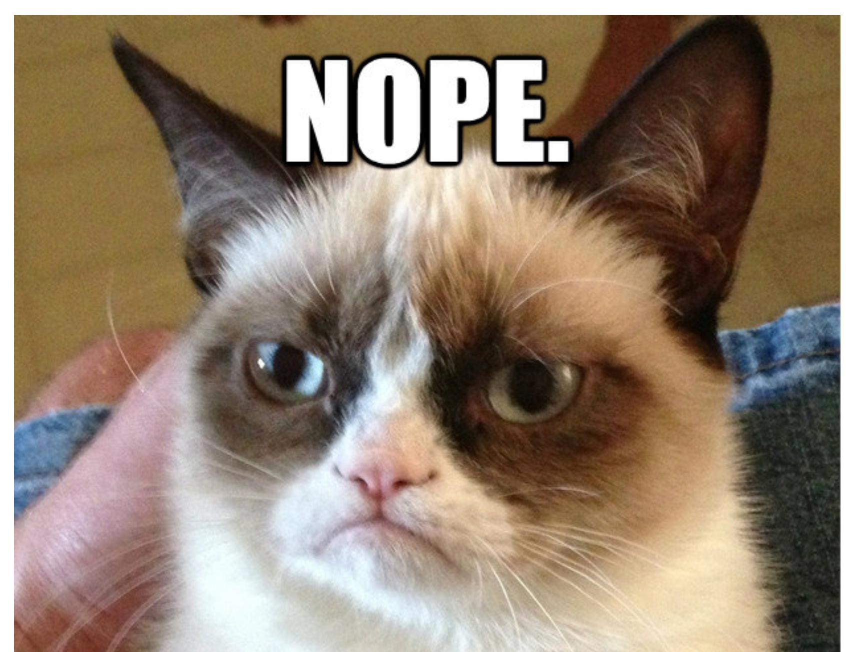
# Implementing Heaps

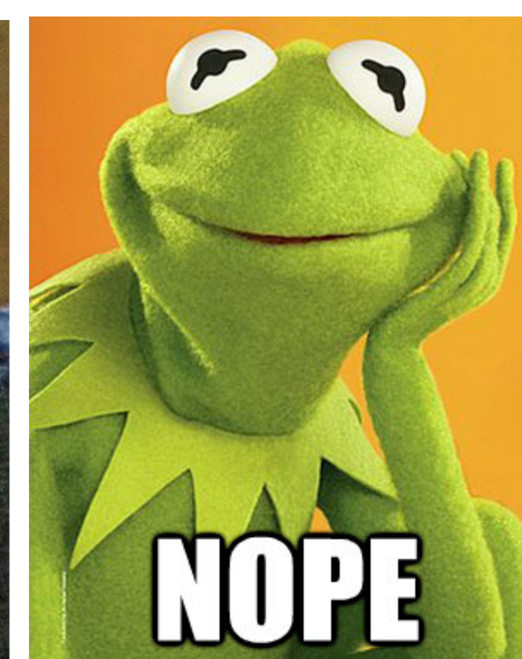
```
public class HeapNode {
    private int value;
    private HeapNode left;
    private HeapNode right;
    ...
}
public class Heap {
    HeapNode root;
    ...
```

# Implementing Heaps

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

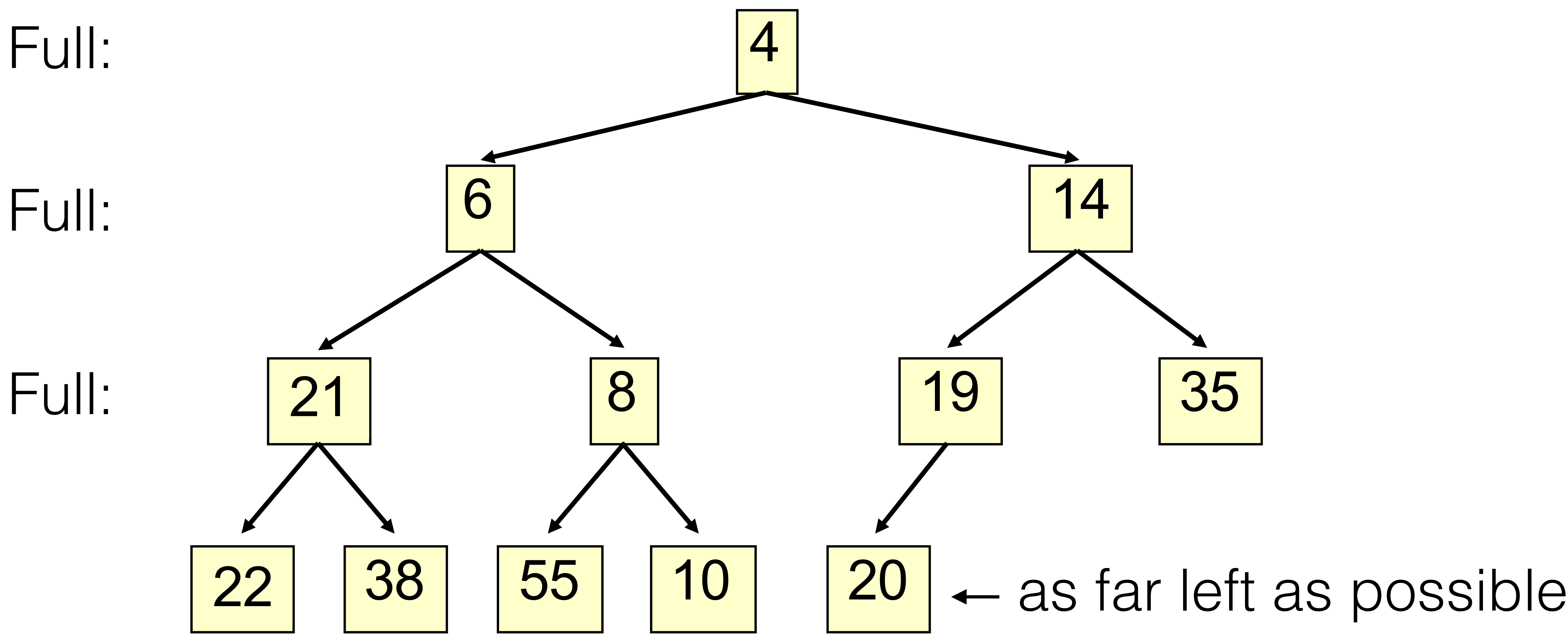


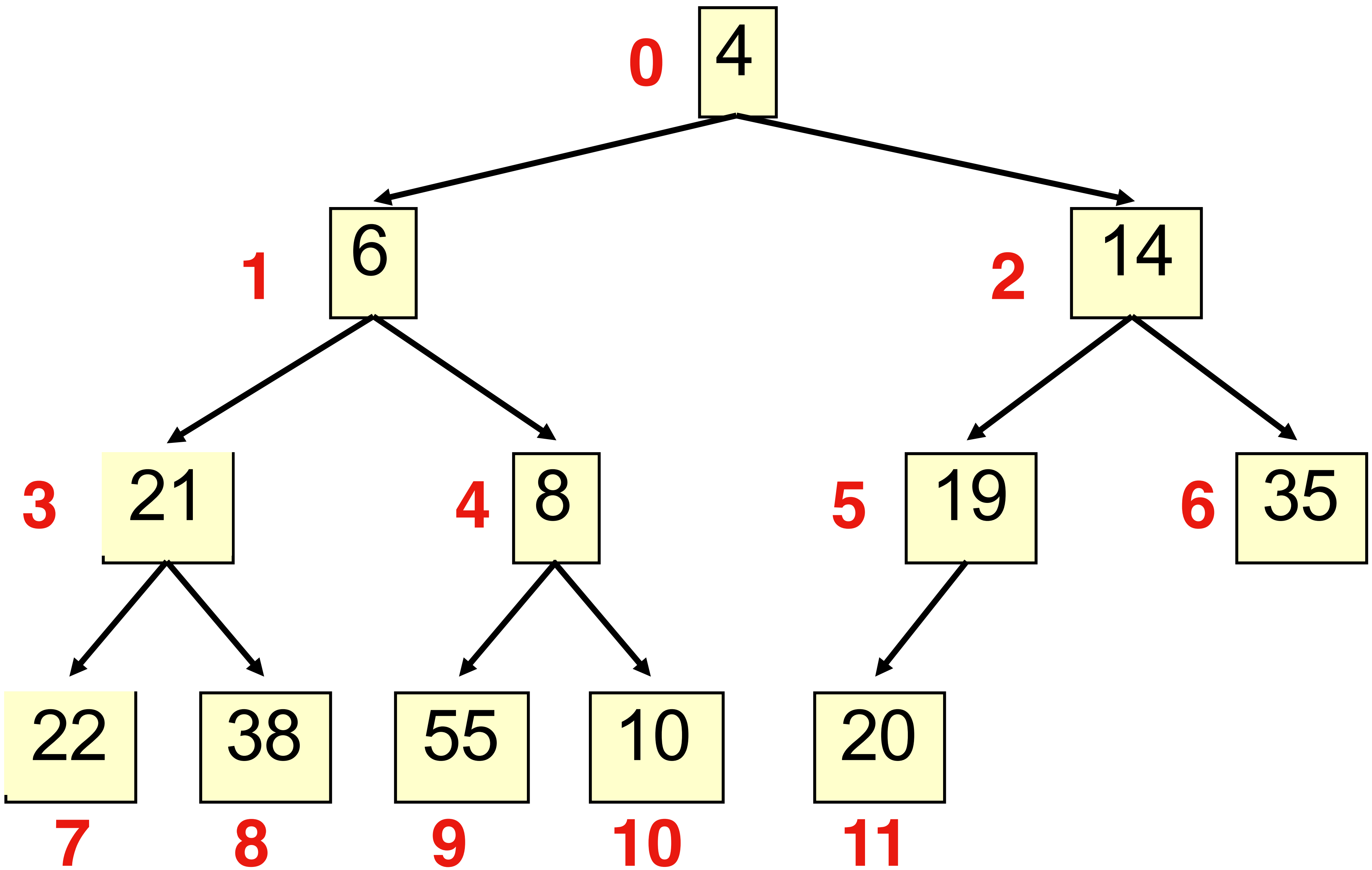
# A heap is a special binary tree.

## 2. **Complete:** no holes!



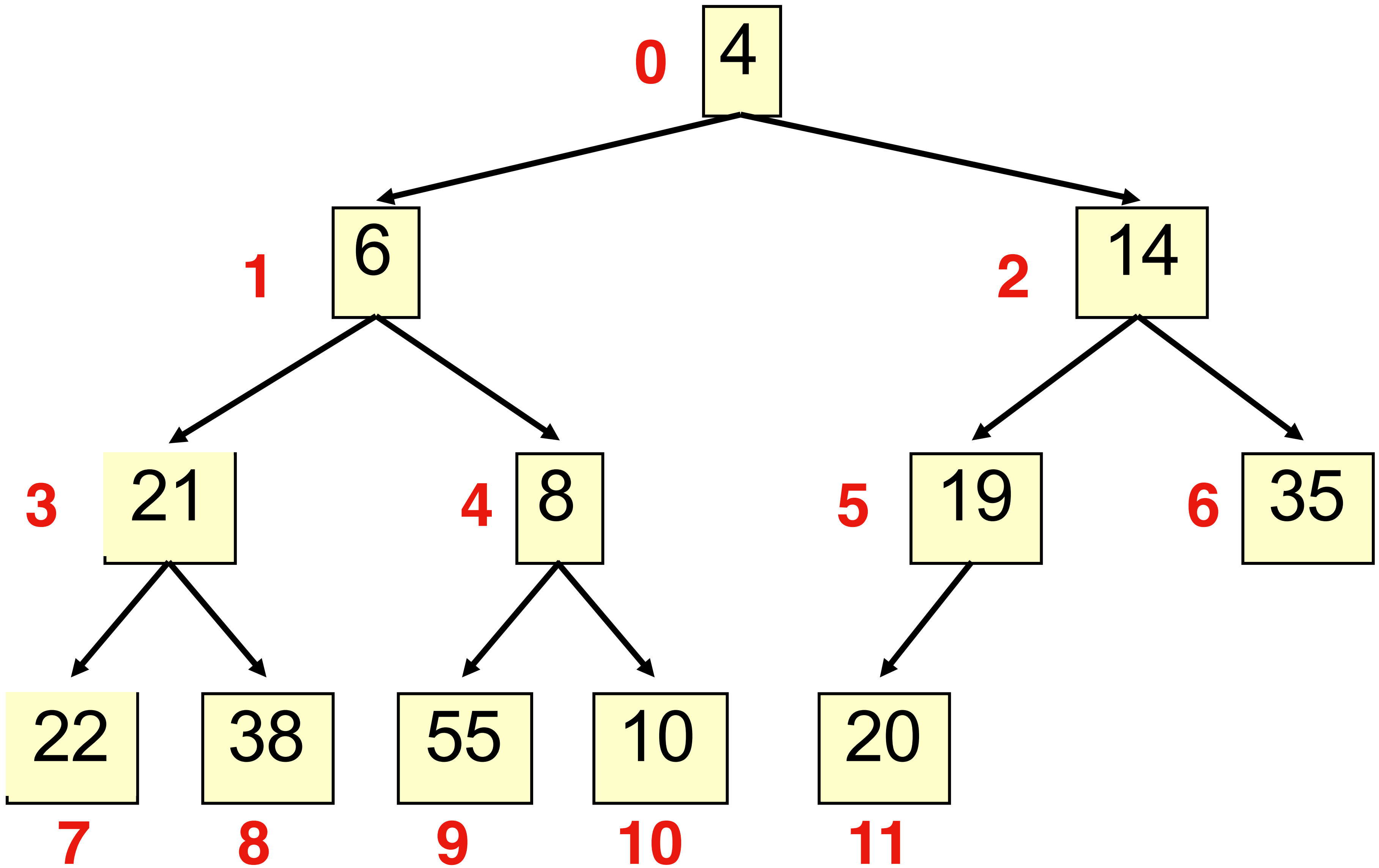
# Numbering Nodes

**Level-order** traversal:




2. Complete: **no holes!**

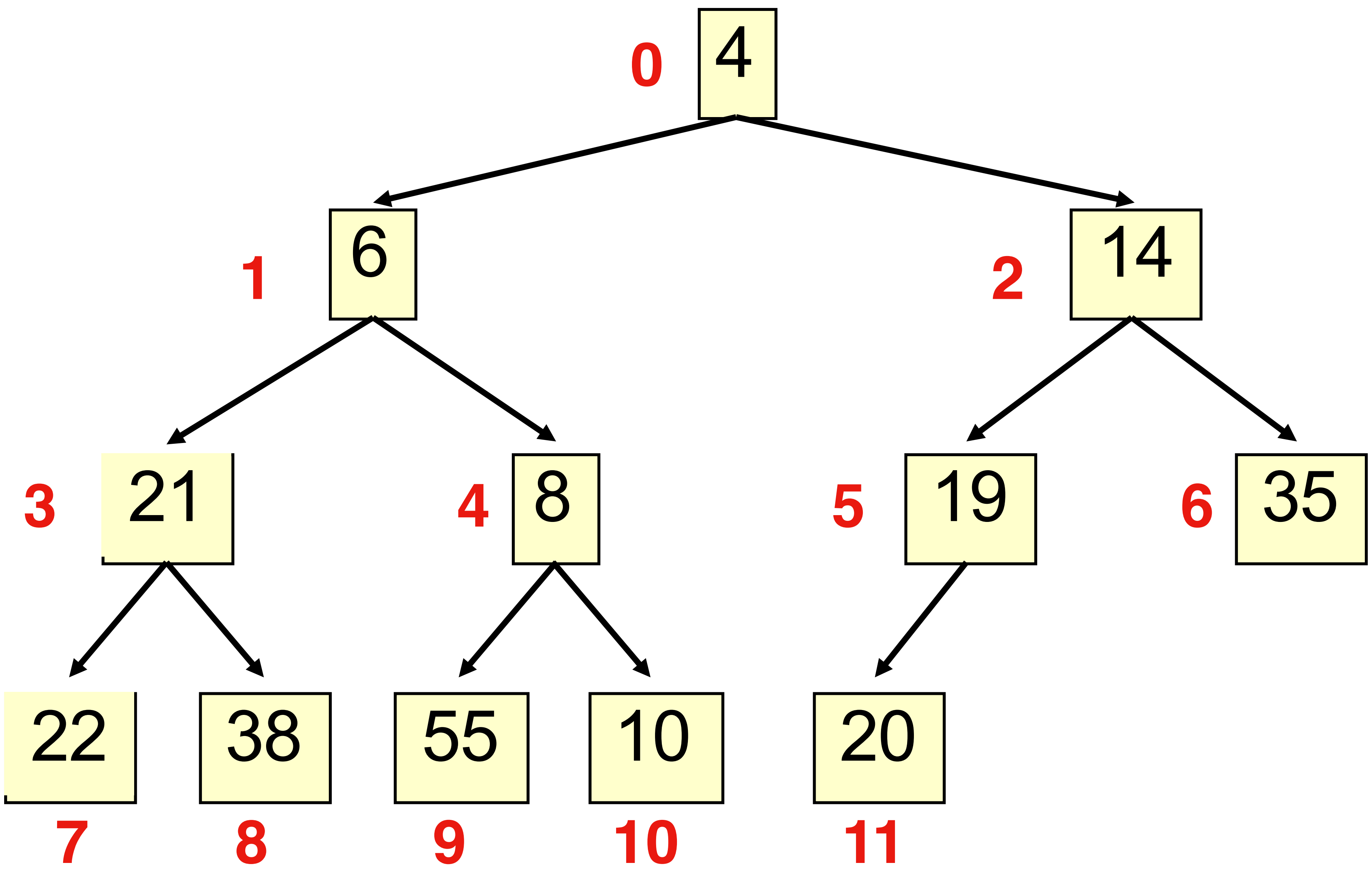
# Numbering Nodes




node **k**'s parent is

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

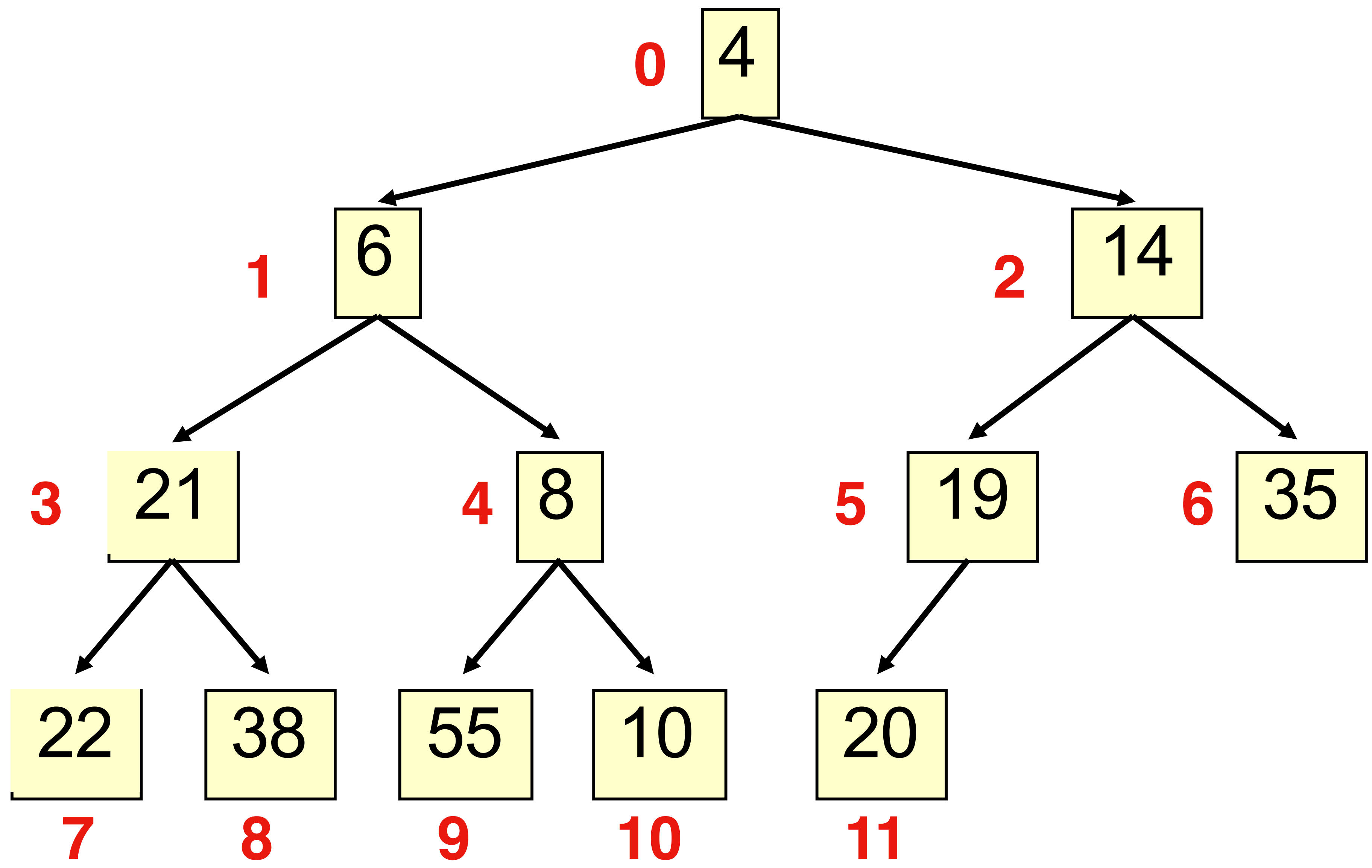
# Numbering Nodes




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**

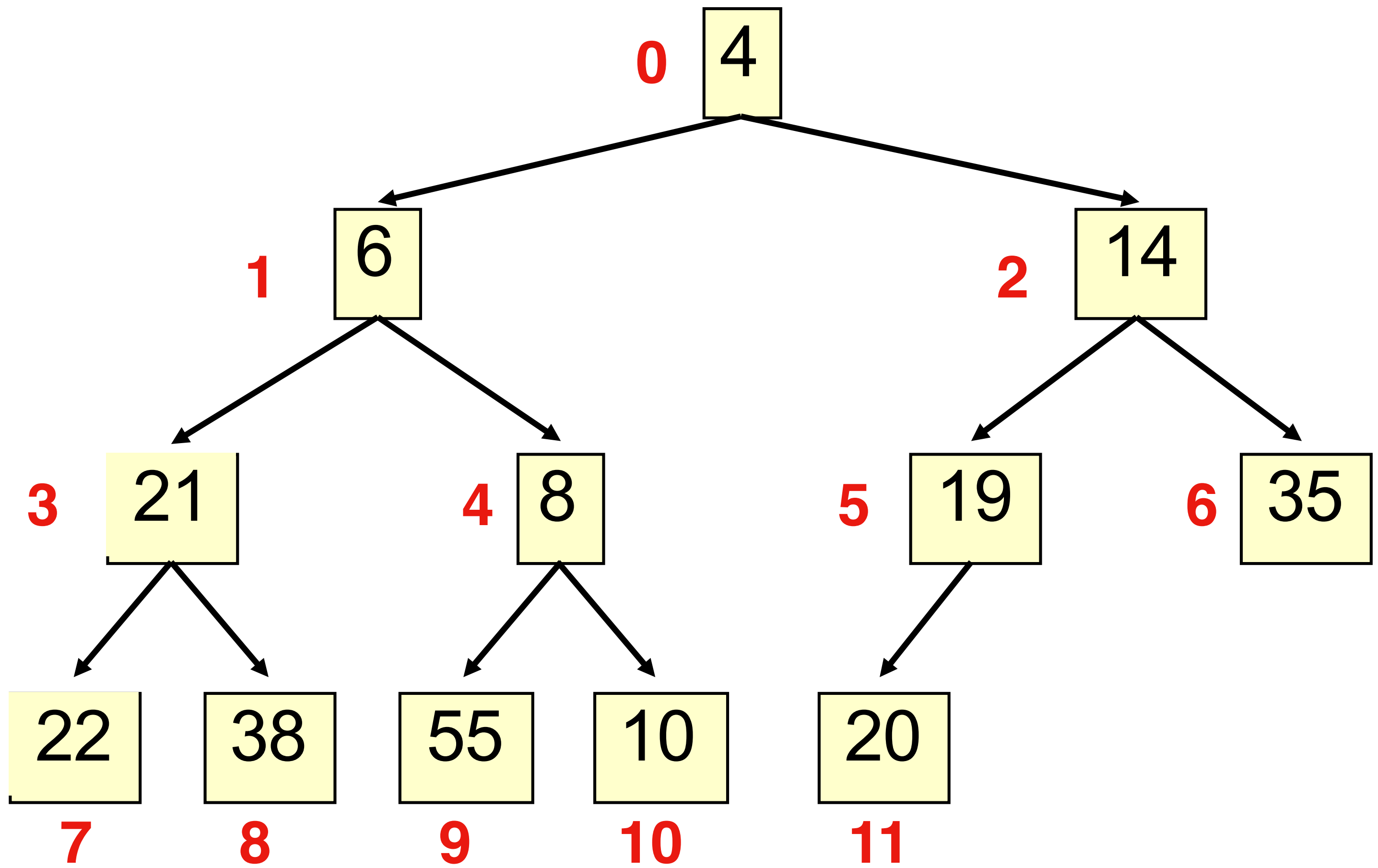
# Implementing Heaps

```
public class Heap {
   private Comparable[] 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]

Write the array after execution of **add(4)**.

Assume the array is large enough to store the additional element.

# Heap it real, part 2.

Here's a heap, stored in an array:

[1 5 7 6 7 10]

Write the array after execution of **add(4)**.

Assume the array is large enough to store the additional element.

[1 5 4 6 7 10 7]

# Heap Operations

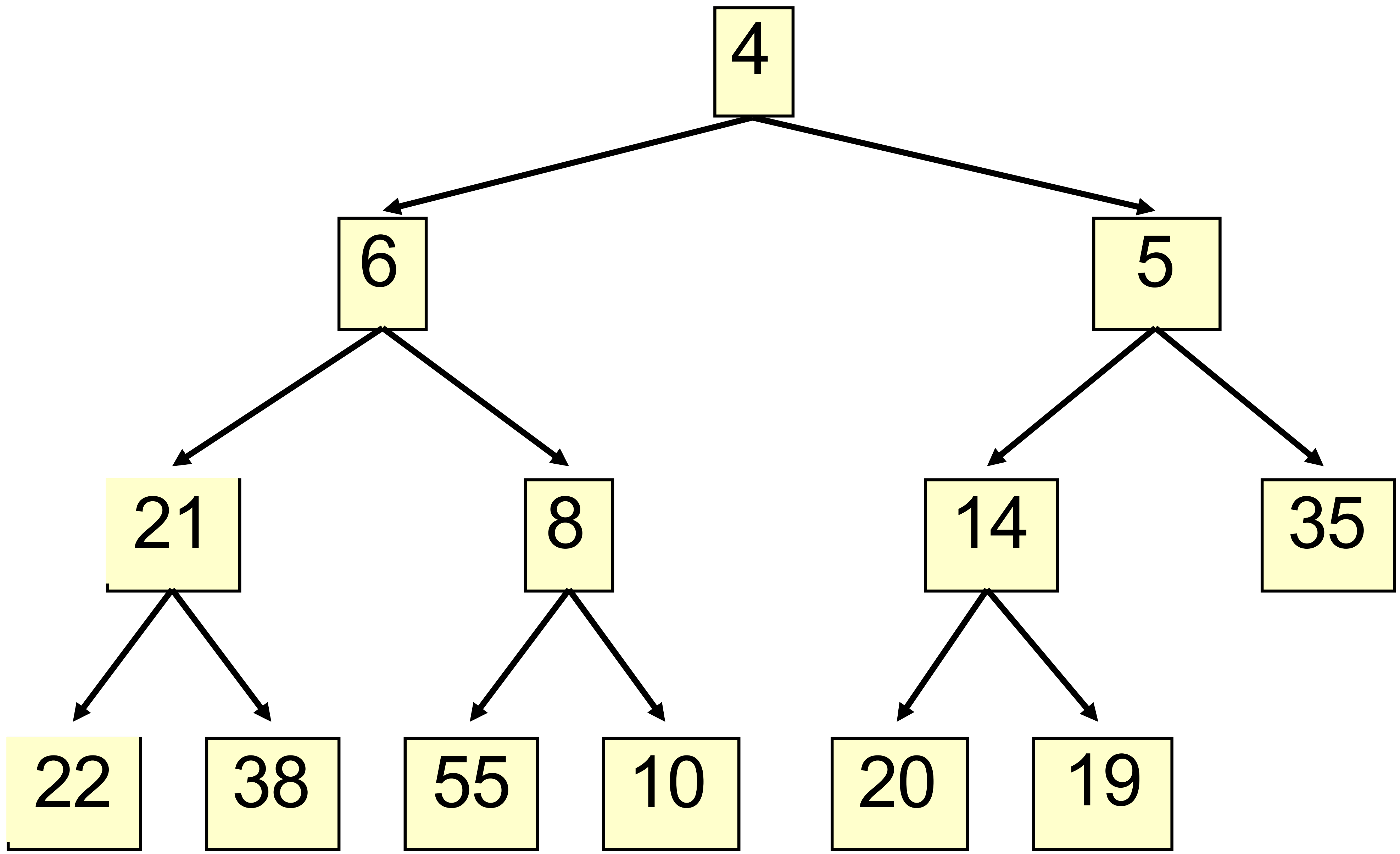
```
interface PriorityQueue<V v, P p> {
 // insert value v with priority p
 void add(V v, P p);

 // return value with min priority
 V peek();

 // remove/return value with min priority
 V poll();

 // more methods…
}
```

```
V poll();
```

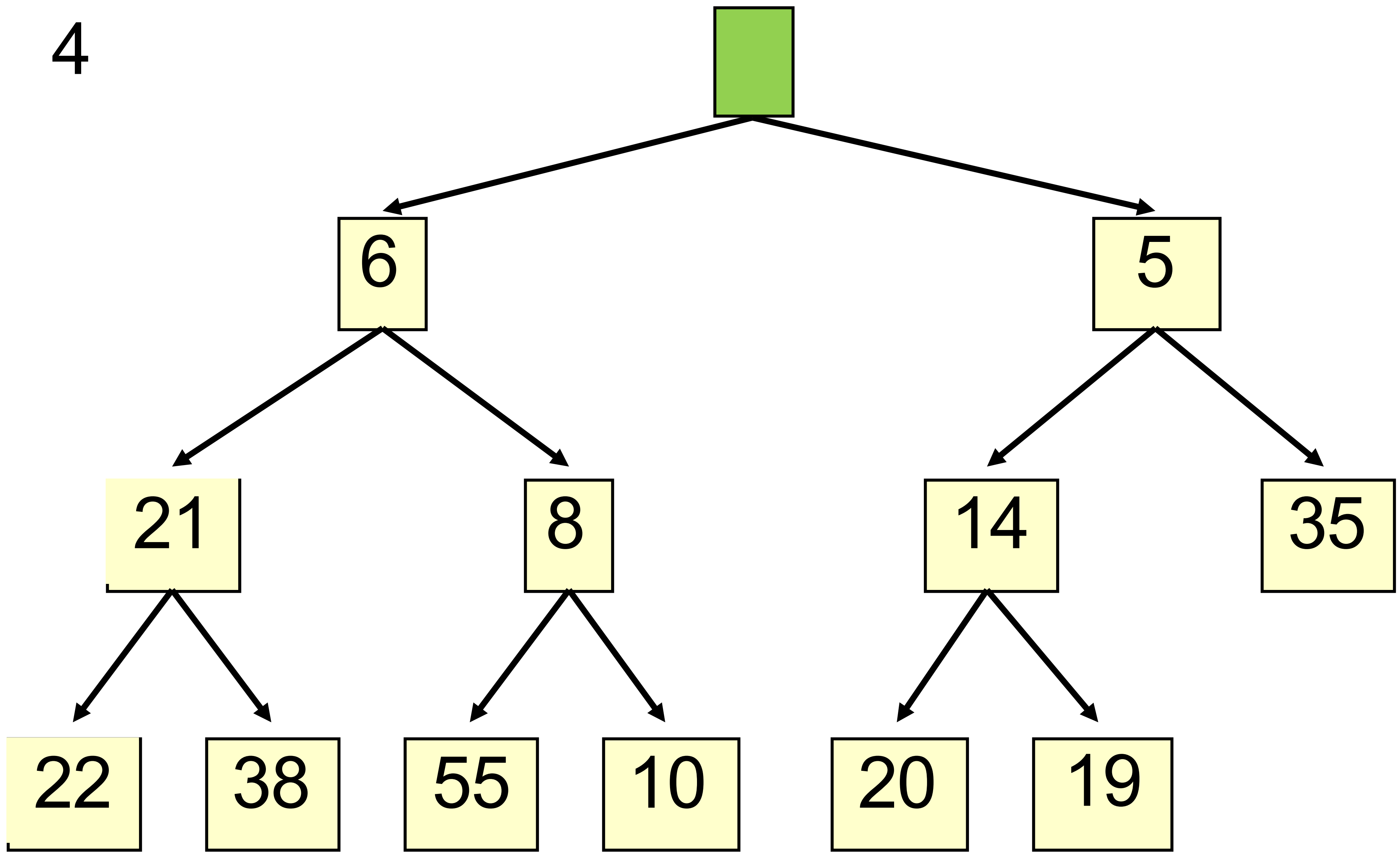
**Algorithm:**

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

```
V poll();
```

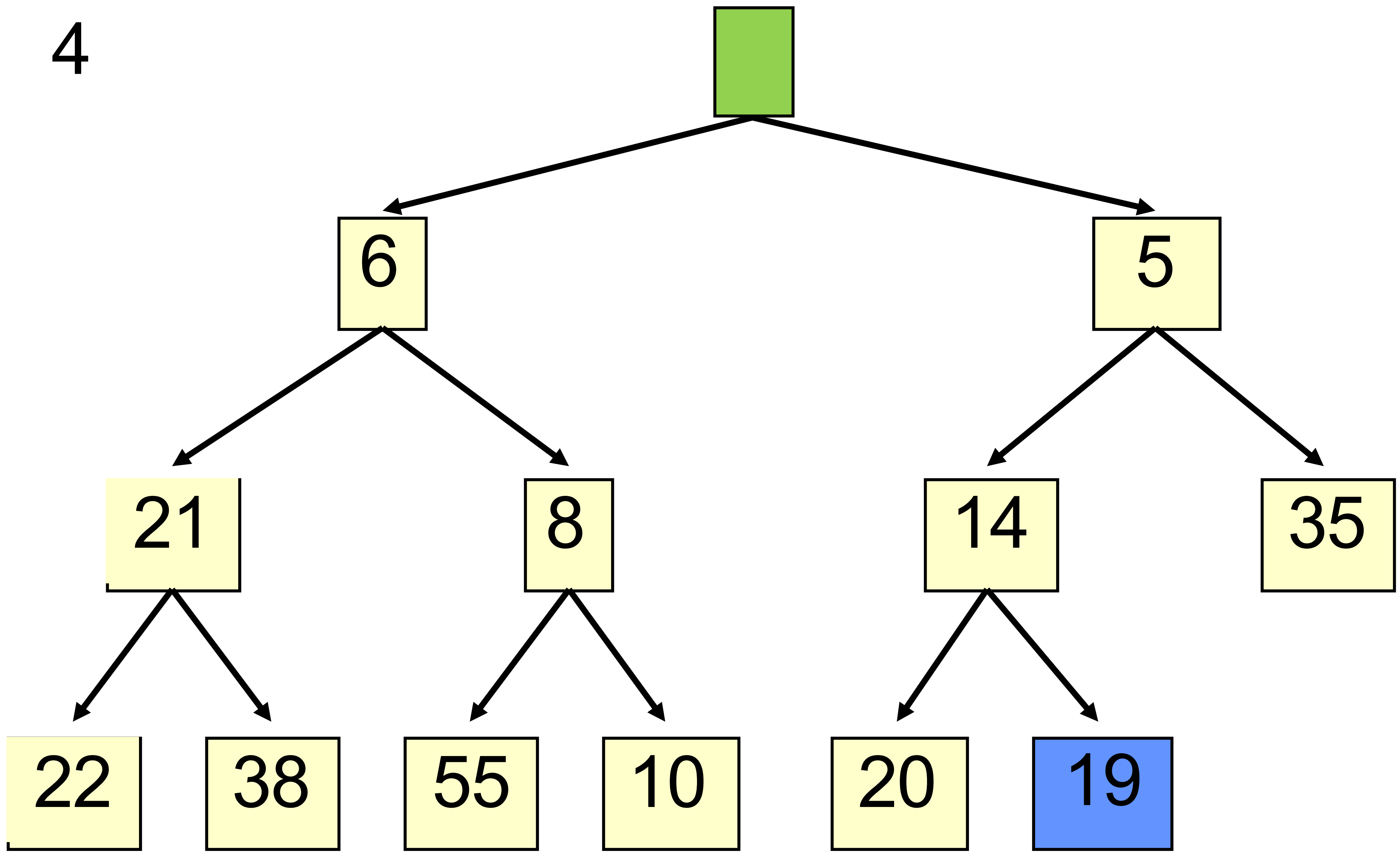


```
V poll();
```




Remove and save the smallest (root) element

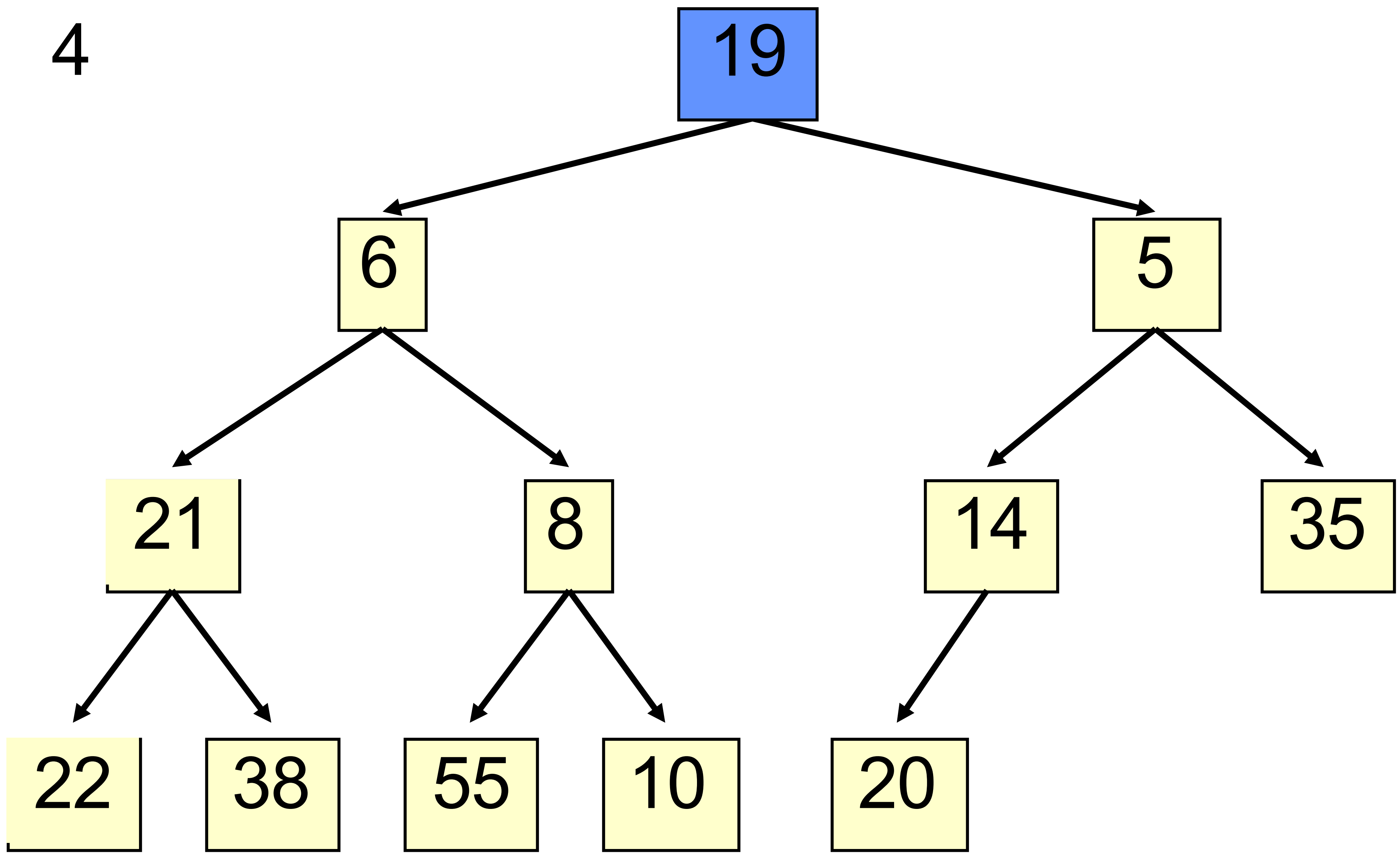
```
V poll();
```




Move the last element to replace the root

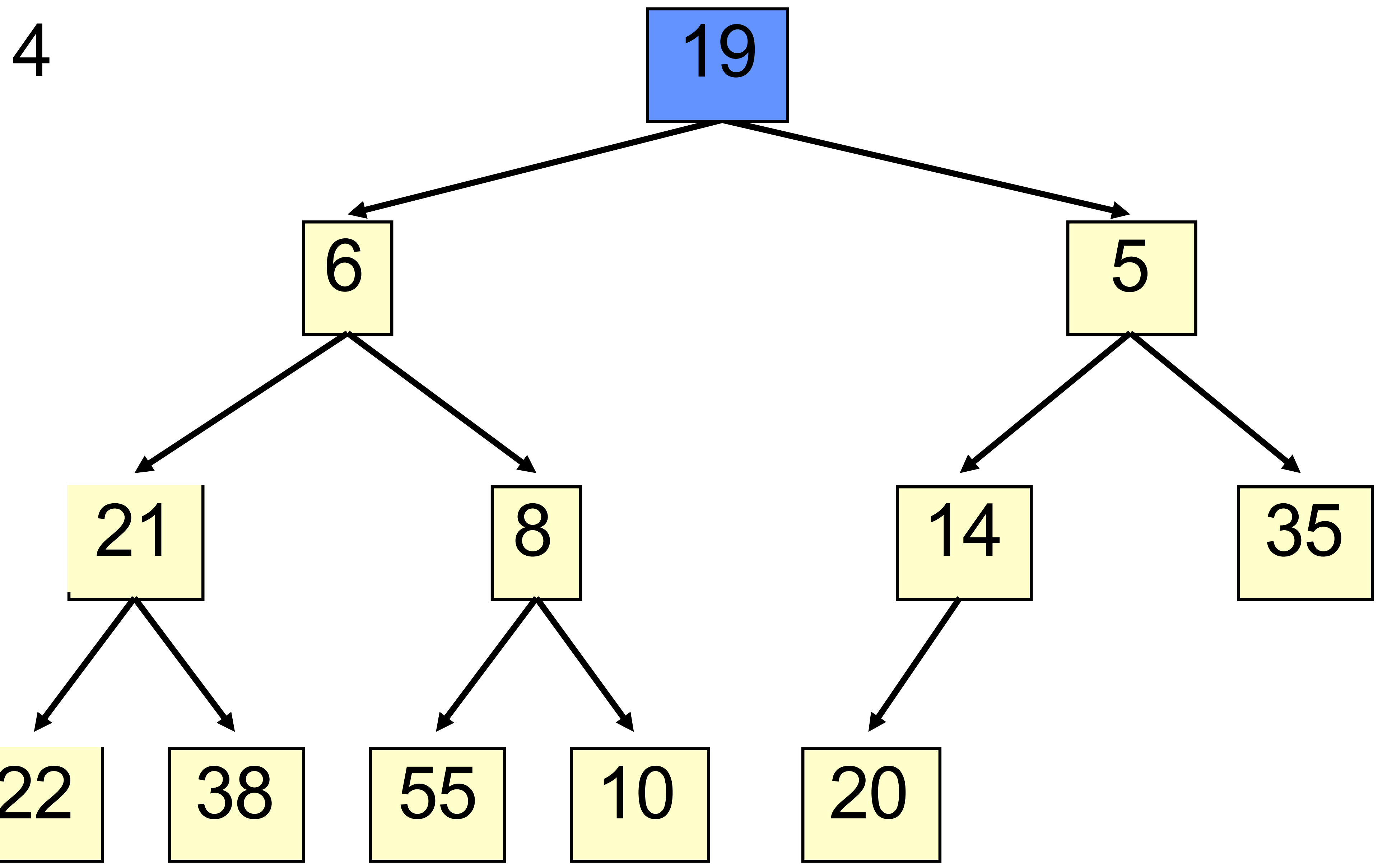
```
V poll();
```

4




Bubble the root value down

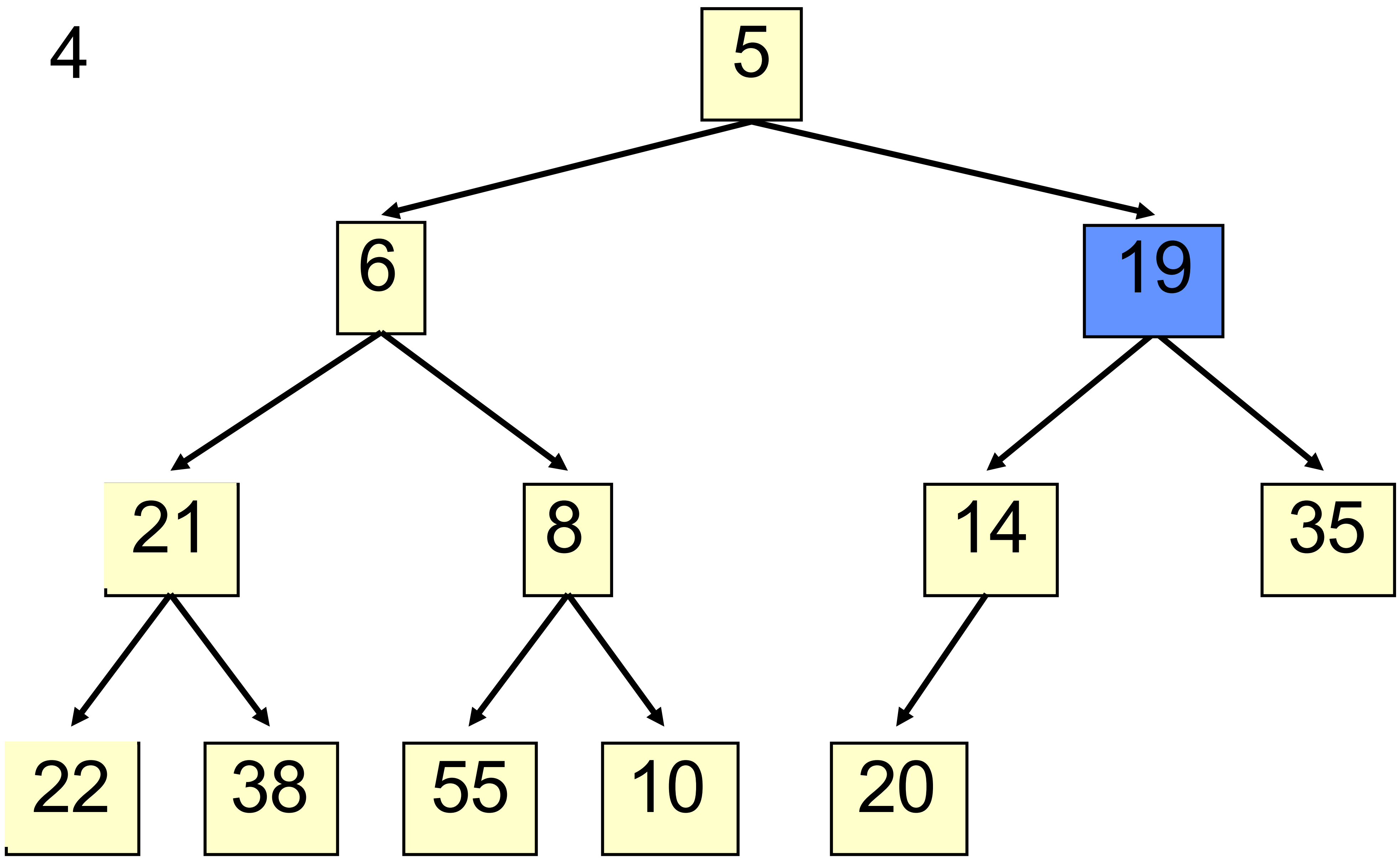
# `V poll();`




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

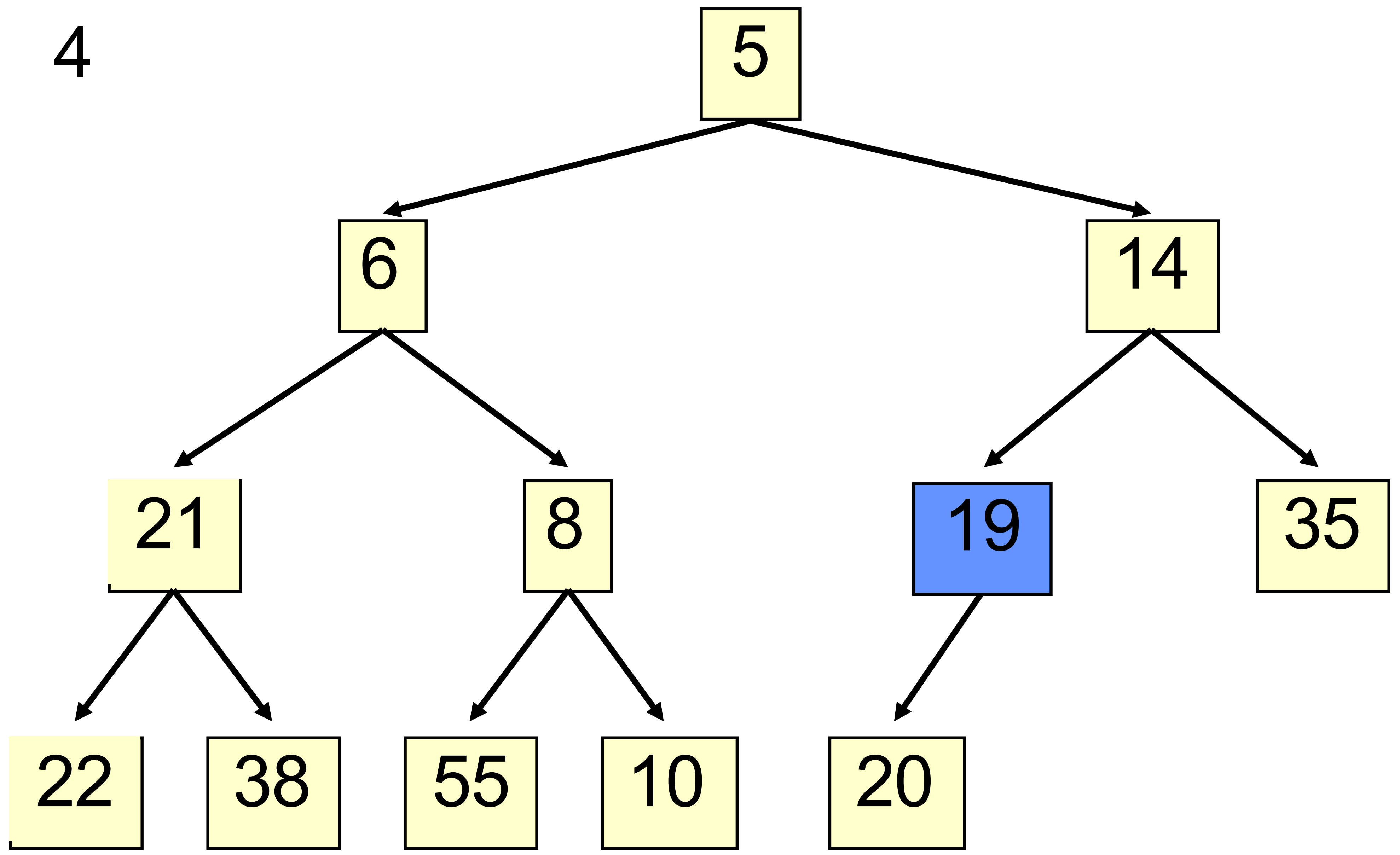
# V poll();

4




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

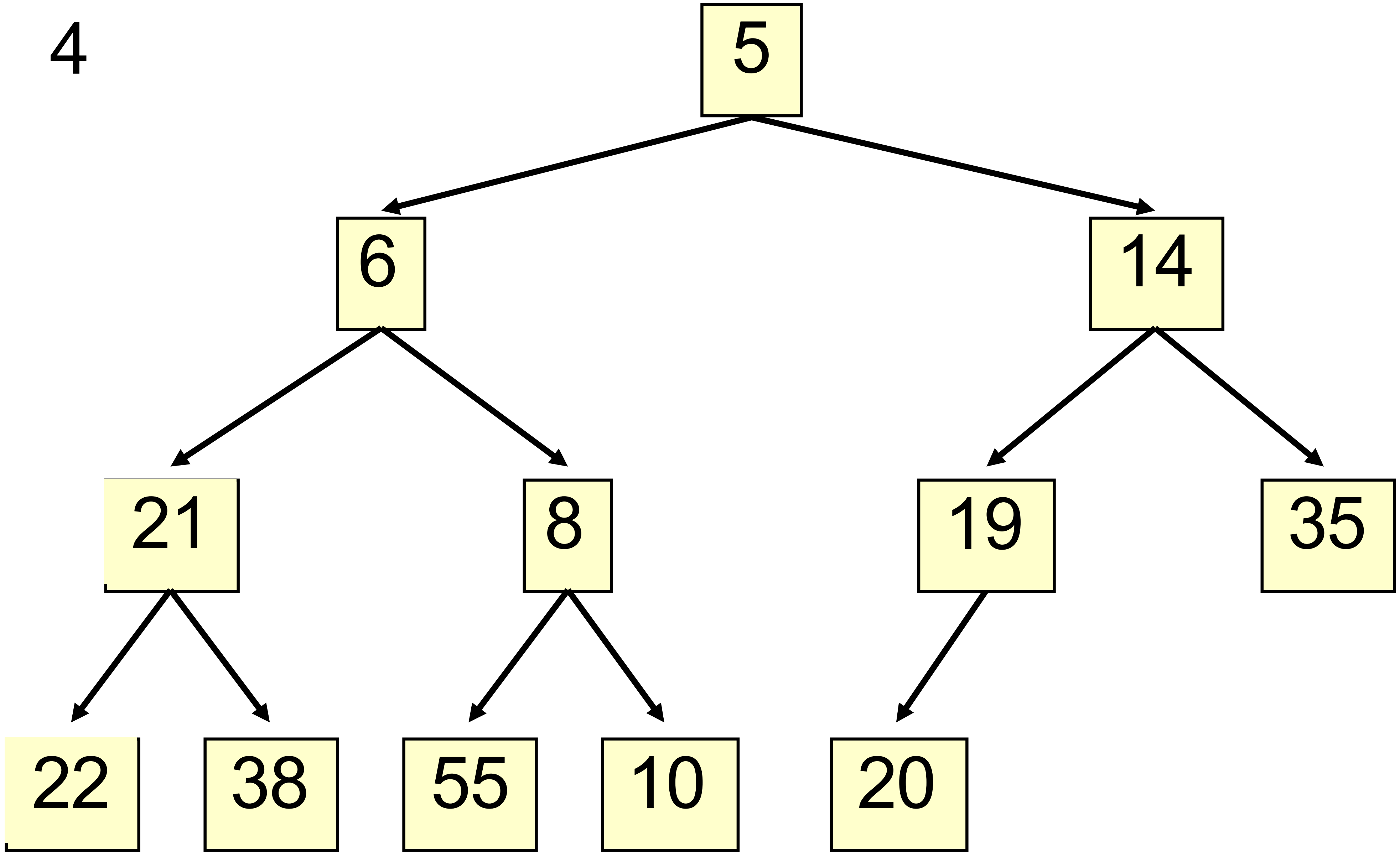
```
V poll();
```




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

```
V poll();
```

4




Return the smallest element.

# `V poll();`

**Algorithm:**

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

# Heap Operations: Runtime

```
interface PriorityQueue<V v, P p> {
 // insert value v with priority p
 void add(V v, P p); O(log n)

 // return value with min priority
 V peek(); O(1)

 // remove/return value with min priority
 V poll(); O(log n)

 // more methods…
}
```

# Review(?) - Interfaces

Java has a thing called an **interface**.

It's like a class, but doesn't have method bodies. It only exists so other classes can **implement** it.

<u>public interface Set</u>

Specifies public method names, specs, parameters, return values, etc.

# Preliminaries - `Comparable`

The **`Comparable`** interface has one method:

***Method Summary***

**All Methods** | **Instance Methods** | **Abstract Methods**

| Modifier and Type | Method and Description |
| --- | --- |
| `int` | `compareTo(T o)`<br>Compares this object with the specified object for order. |

**Returns:**

```
a negative integer if this < o
zero if this is equal to o
a positive integer if this is > o.
```

From A2: you can call w.compareTo(node.word) because `String` <u>implements</u> `Comparable`.

# Preliminaries - `Comparable`

The **`Comparable`** interface has one method:

***Method Summary***

**All Methods** | **Instance Methods** | **Abstract Methods**

| Modifier and Type | Method and Description |
| --- | --- |
| `int` | `compareTo(T o)`<br>Compares this object with the specified object for order. |

If you can compare items, you can sort them using comparison sorts!

They have a well-defined **ordering**.

# Recall: Generics

Key idea: I don’t need to know what T is to implement these!

```
Collection<String> c= ...
c.add(“Hello”)  /* Okay */
c.add(1979);    /* Illegal: compile error! */
```

Generally speaking,

`Collection<String>`

behaves like the parameterized type

`Collection<T>`

where all occurrences of `T` have been replaced by `String`.

# Fancier Generics

What if I care a little bit what T is?

```
SortableCollection<String> c= ...
c.sort();
```

requires T to be Comparable!

# Fancier Generics

What if I care a little bit what T is?

```
SortableCollection<String> c= ...
c.sort();
```

← requires T to be Comparable<T>!

```
interface SortableCollection<T extends Comparable<T>>
{
  ...
}
```

# What's with the **V**'s, **P**'s, and **E**'s: Java's Version

- The Java PriorityQueue interface is a little different:
  - It stores values of generic type **E**.
  - **E** must be **Comparable**.
  - The highest-priority element is the "smallest" element (of type **E**) per the compareTo ordering
  - In other words: if you sorted the elements in the heap, poll would return the first one.
    - But you don't have to sort - the min value is always at the root!

# What's with the **V**'s, **P**'s, and **E**'s: Java's Version

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

# What's with the **V**'s, **P**'s, and **E**'s: A3's Version

- The A3 Heap class:
  - The Heap has two type parameters:

    **Heap<V, P extends Comparable<P>>**

  - It stores each element in an *inner* class

    **Pair<V, P extends Comparable<P>>**,

    A Pair stores a value (of type **V**) together with its priority (of type **P**, which must be Comparable)

  - The highest-priority element is the Pair whose **P** is smallest according to the compareTo ordering
  - Peek and Poll return the *value* (of type **V**) associated with the smallest *priority* (of type **P**).

# What's with the **V**'s, **P**'s, and **E**'s: A3's Version

```
interface PriorityQueue<V v, P p> {
 // insert value v with priority p
 void add(V v, P p);

 // return value with min priority
 V peek();

 // remove/return value with min priority
 V poll();

 // more methods…
}
```

# Magic trick time!

# Heapsort

```
public static void heapsort(int[] b) {
  Heap h = new Heap<Integer>();
  // 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] = poll(b, k);
  }
}
```

# Heapsort

```
public static void heapsort(int[] b) {
  Heap h = new Heap<Integer>();
  // 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] = poll(b, k);
  }
}
```

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