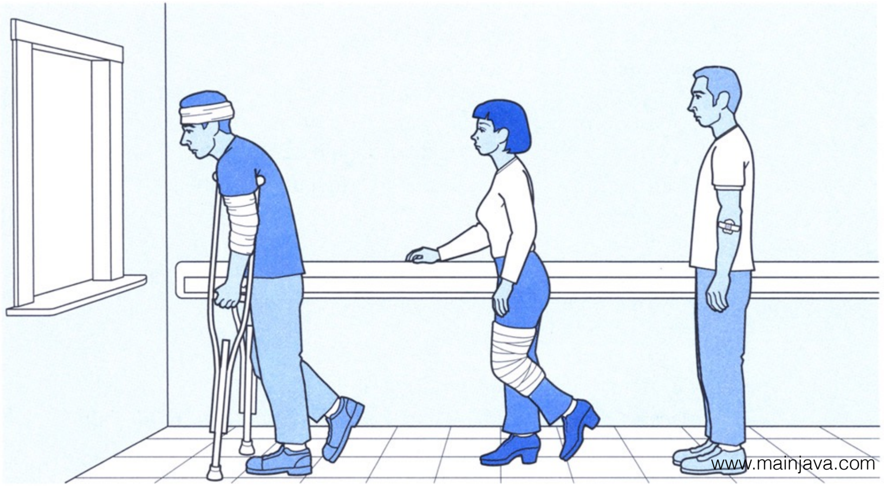


CSCI 241

Lecture 16:
Heaps and Priority Queue, Continued

Announcements

- No quiz this Friday!

Announcements

- Midterm Exam is Friday
    - Covers material through today
    - Format similar to quizzes
    - Available 8am-11:59pm
    - 60 minute time limit
- No class meeting on Friday.
- No lab deliverable this week - use lab for review.
- A3 should be out later today. Implementing your heap is good review for the midterm!

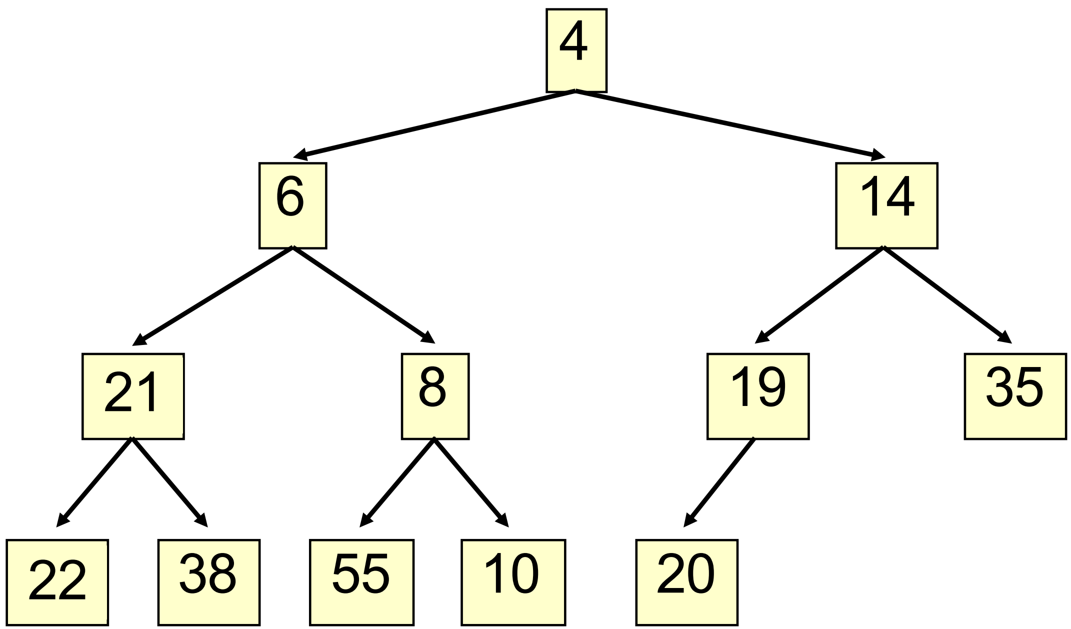
Goals

- 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.
- Know how to implement a Priority Queue using a heap
- Know how to sort an array using Heapsort

A heap is a special binary tree.

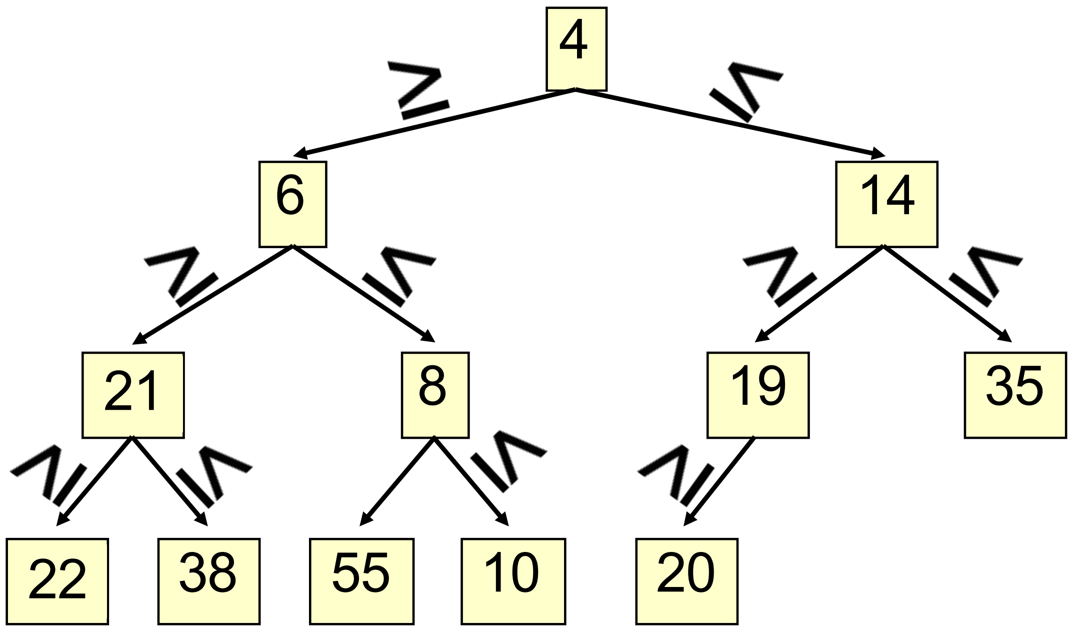
1. **Heap Order Invariant:**

Each element $\geq$ its parent.



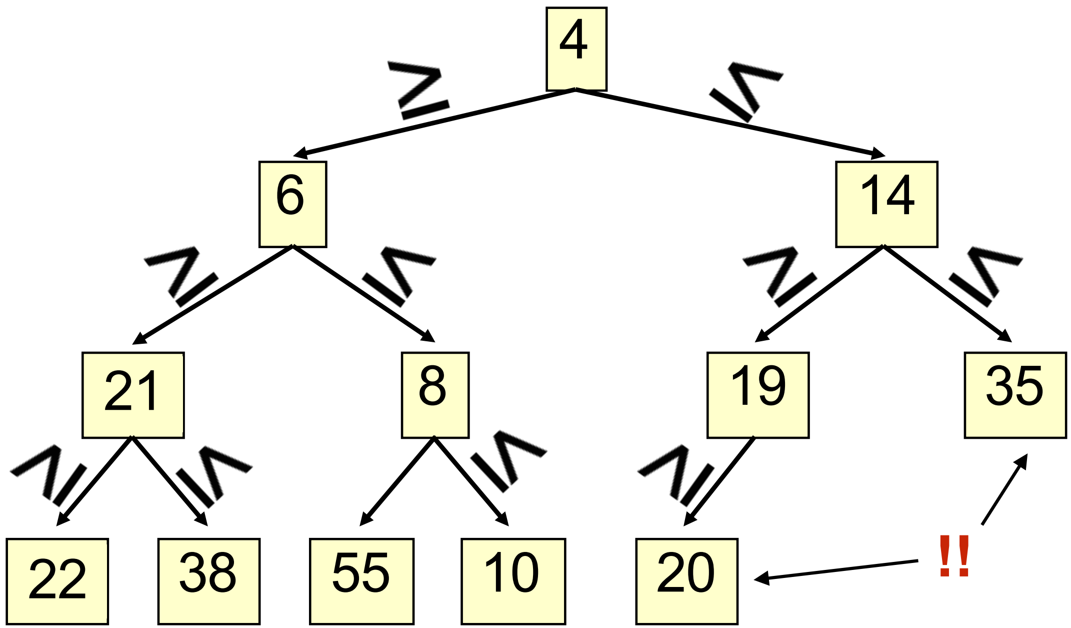
A heap is a special binary tree.

1. **Heap Order Invariant:**
Each element $\geq$ its parent.



A heap is a special binary tree.

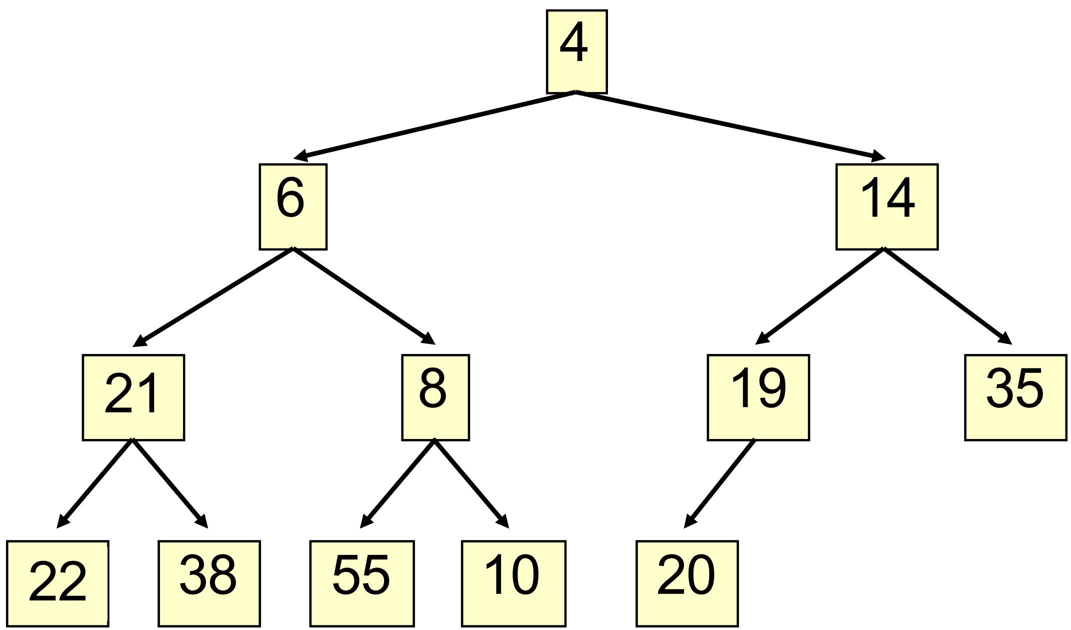
1. **Heap Order Invariant:**
Each element $\geq$ its parent.



A heap is a special binary tree.

2. **Complete:** no holes!

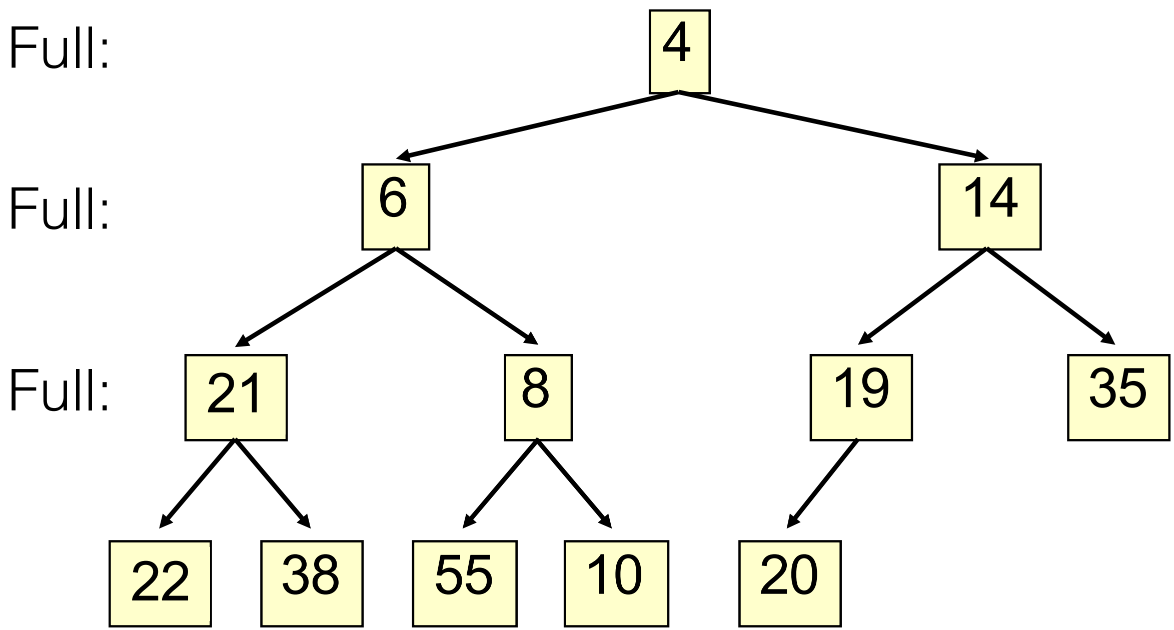
- All levels except the last are full.
- Nodes in last level are as far left as possible.



A heap is a special binary tree.

2. **Complete:** no holes!

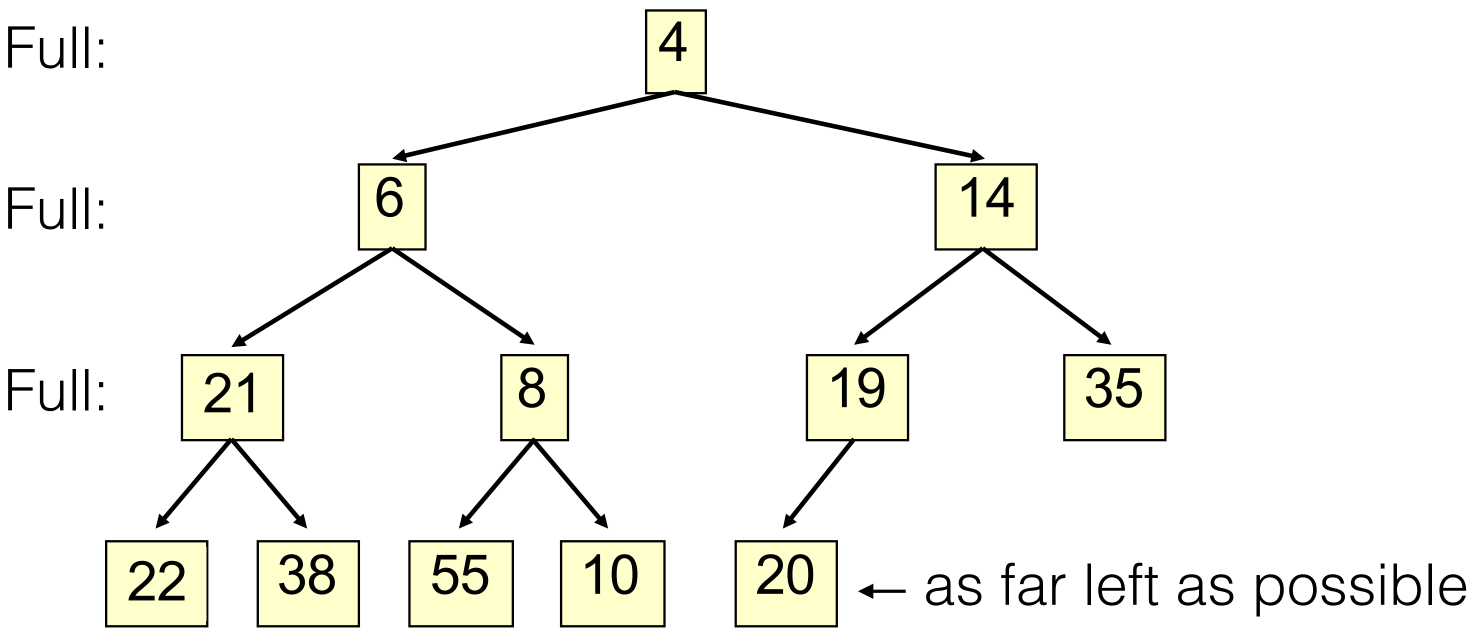
- All levels except the last are full.
- Nodes in last level are as far left as possible.



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

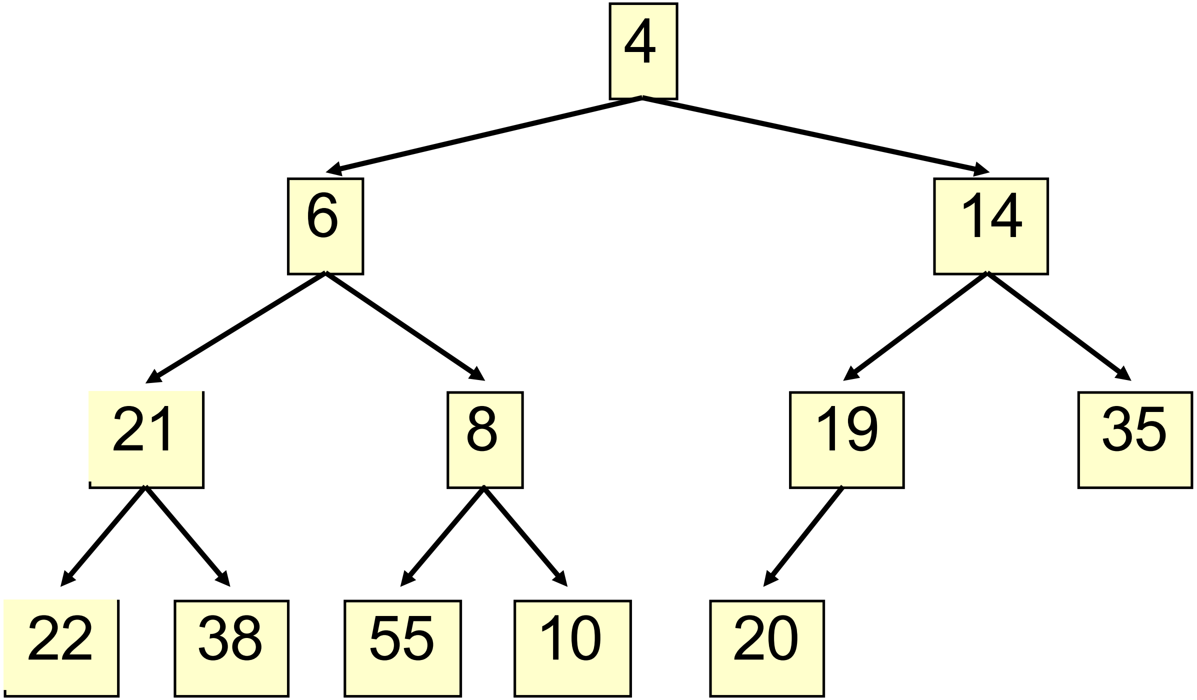
```
interface PriorityQueue<E> {
 boolean add(E e); // insert e
 E peek(); // return min element
 E poll(); // remove/return min element
 void clear();
 boolean contains(E e);
 boolean remove(E e);
 int size();
 Iterator<E> iterator();
}
```

```
void add(E e);
```

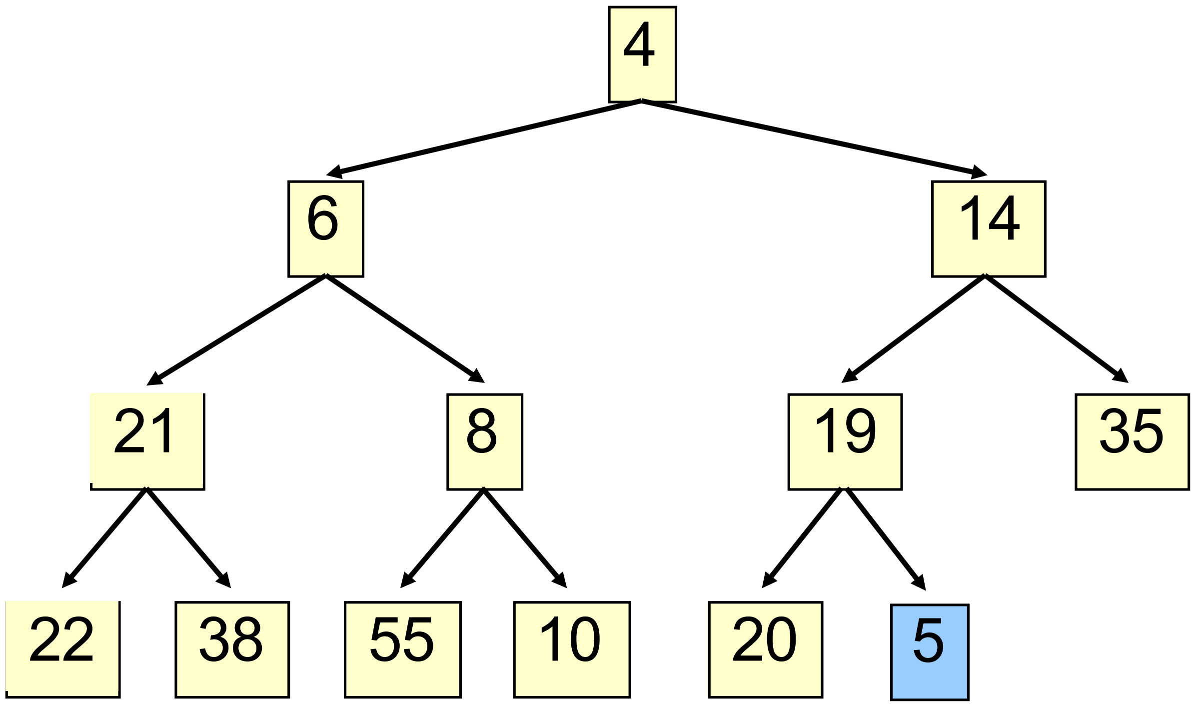
Algorithm:

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

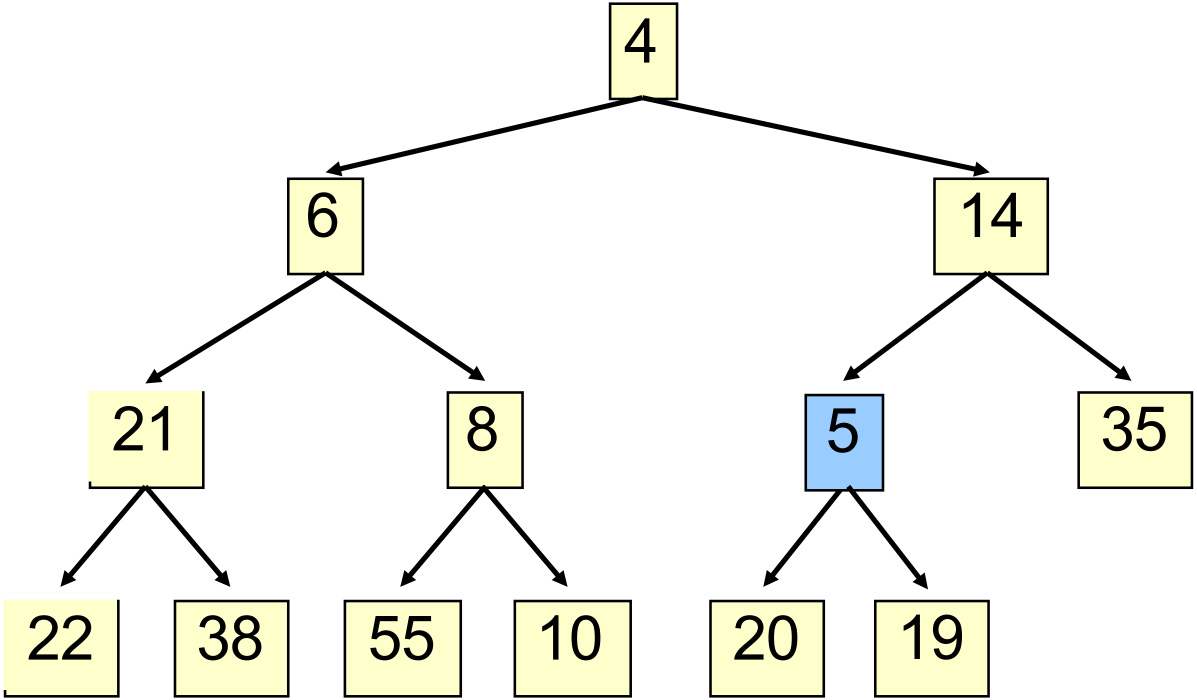
```
void add(E e);
```



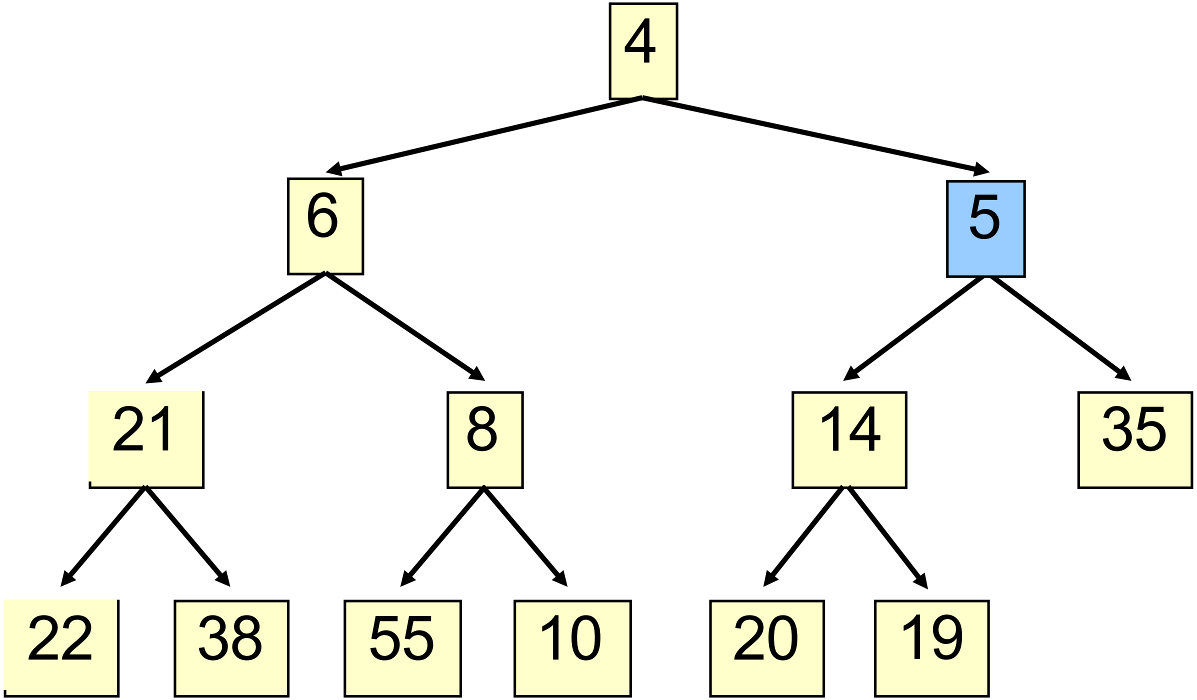
```
void add(E e);
```



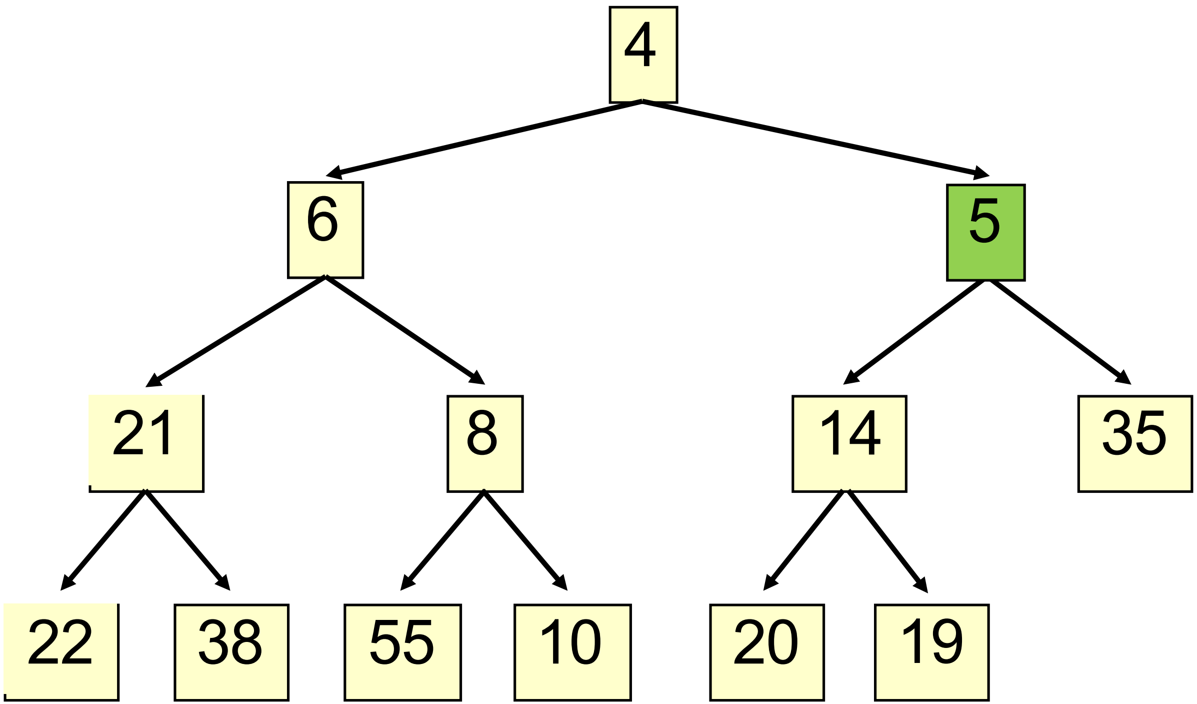
```
void add(E e);
```



```
void add(E e);
```



```
void add(E e);
```



```
void add(E e);
```

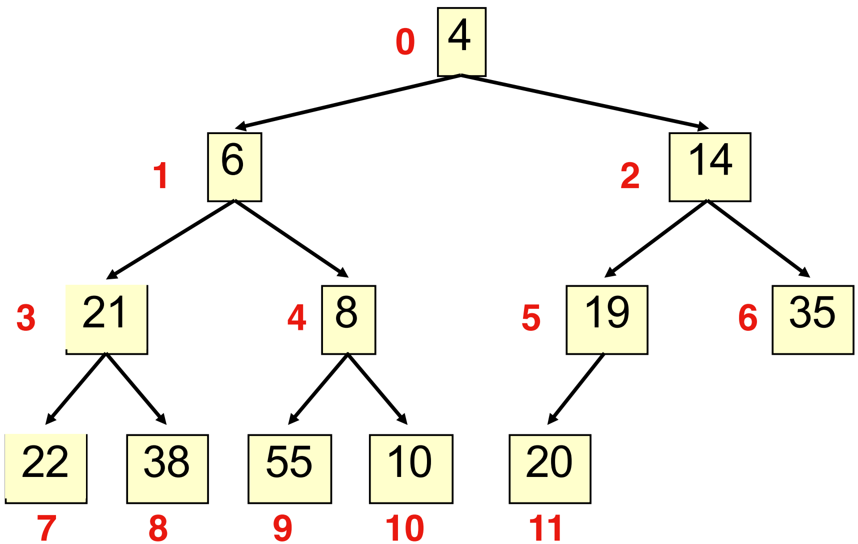
Algorithm:

- 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!

Storing Heaps: Implicit Tree Structure

2. Complete: **no holes!**



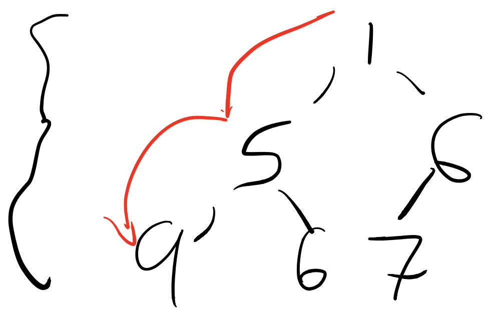
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	6	14	21	8	19	35	22	38	55	10	20				

Warmup

What's the height of this heap?

[1 5 6 9 6 7]

A. 1
B. 2
C. 3
D. 4



Add 2, write array?

Heap Operations

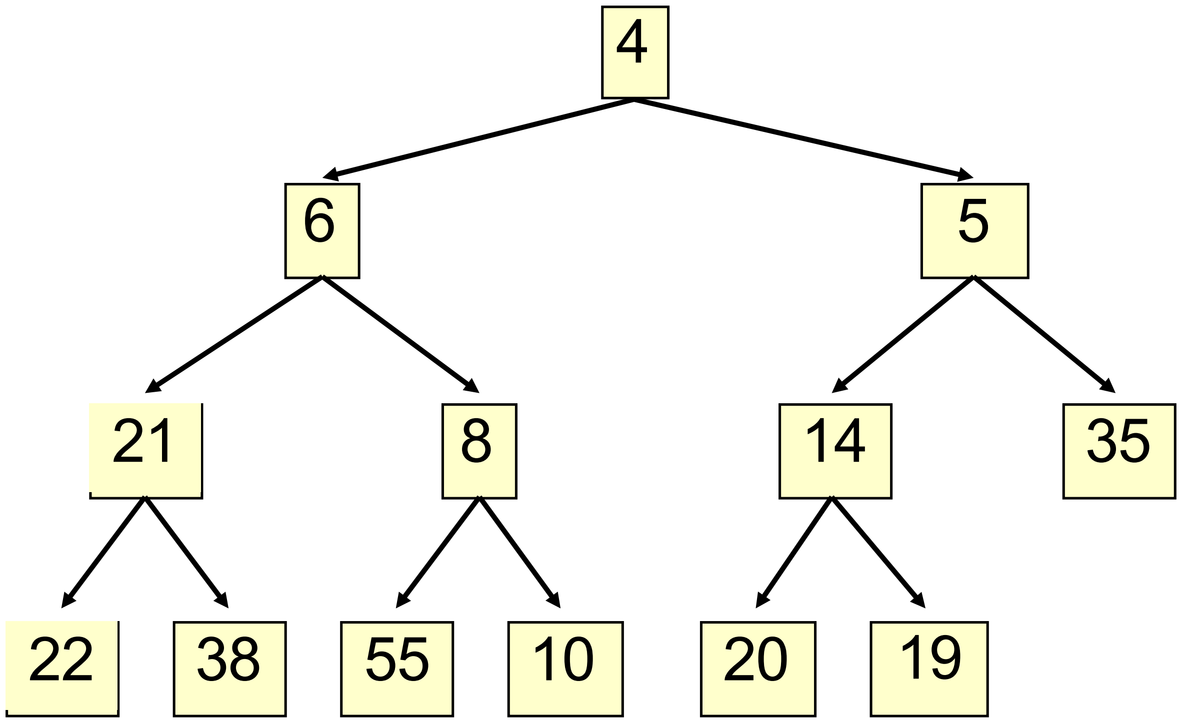
```
interface PriorityQueue<V v, P p> {
  // insert value v with priority p
  void add(V v, P p);

  // return value with min priority
  V peek();           return A[0];

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

  // more methods…
}
```

```
V poll();
```

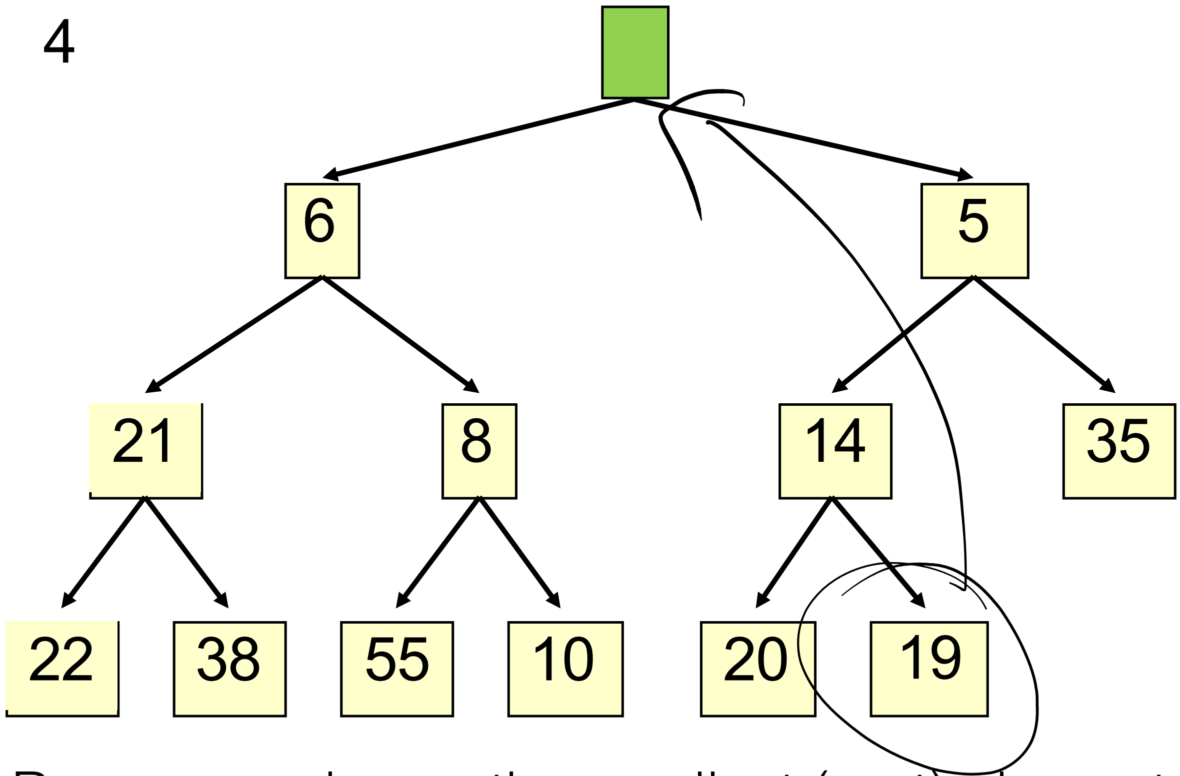
Algorithm:

- ✓ 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();
```



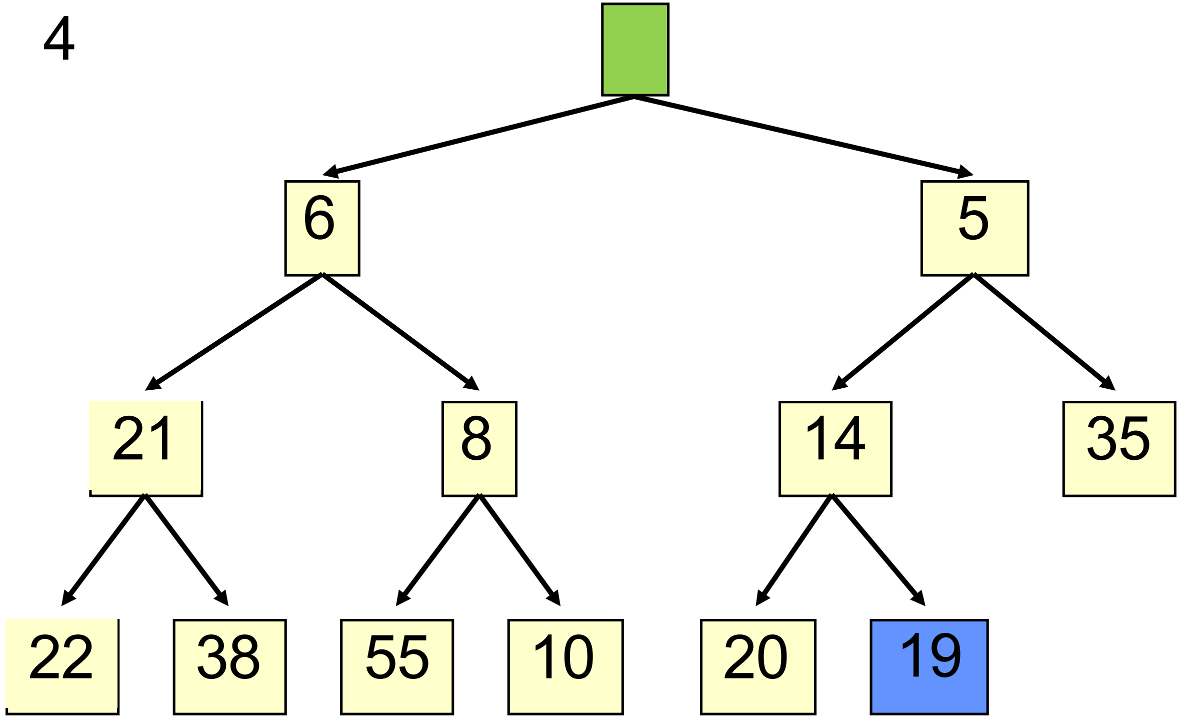
V poll();



Remove and save the smallest (root) element

```
V poll();
```

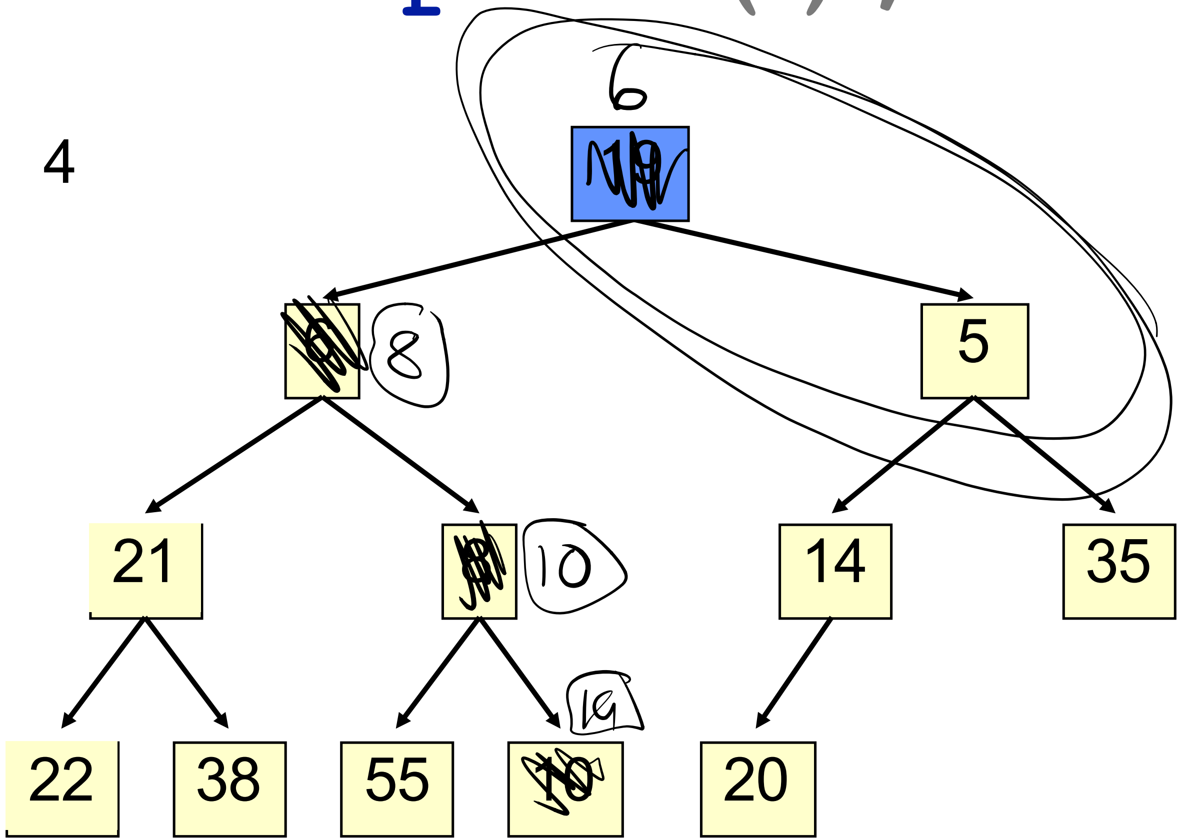
4



Move the last element to replace the root

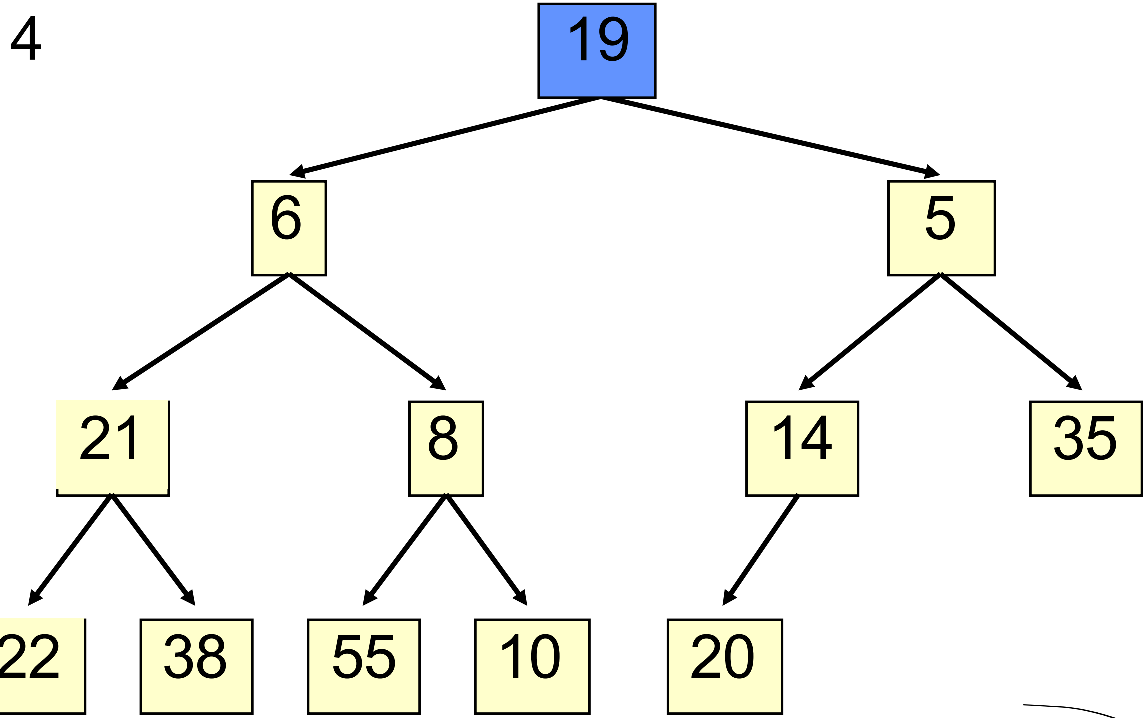
```
V poll();
```

4



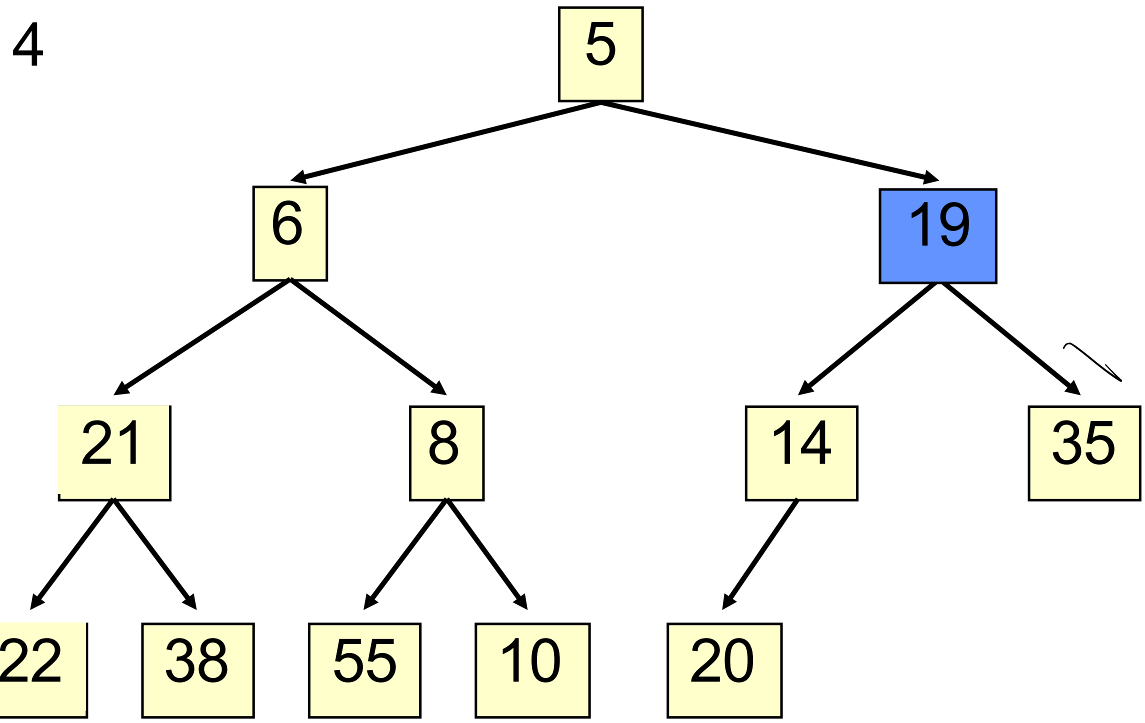
Bubble the root value down

V poll();



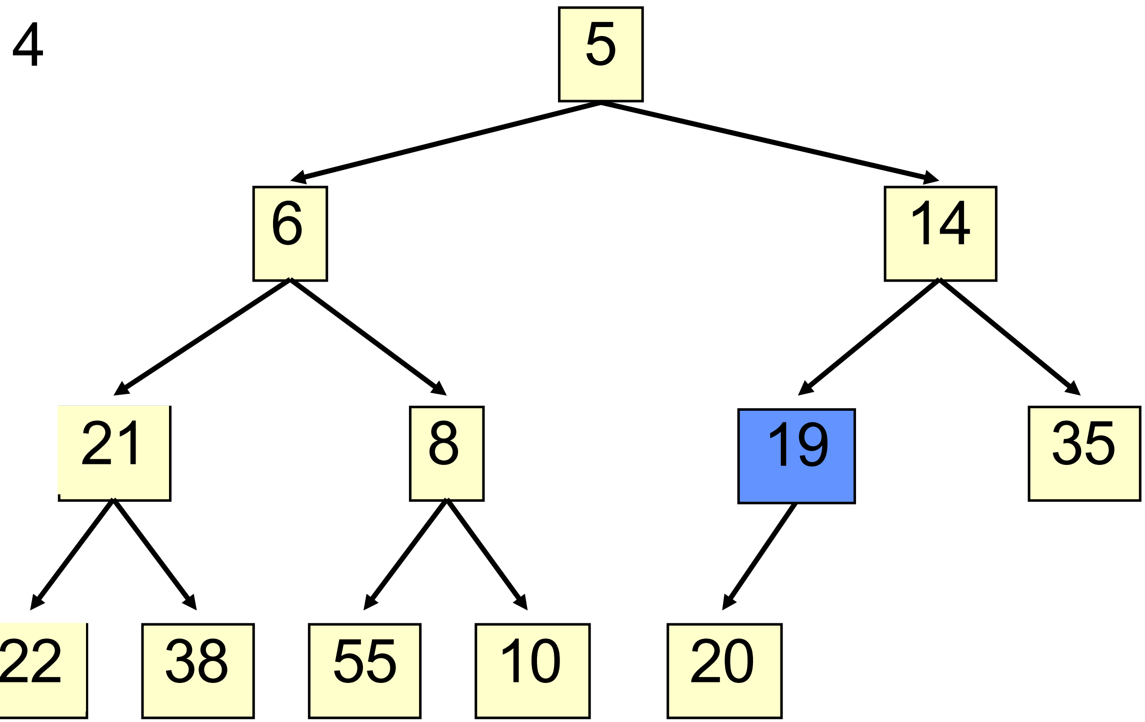
Bubble the root value down, swapping with the **smaller** child

`V poll();`



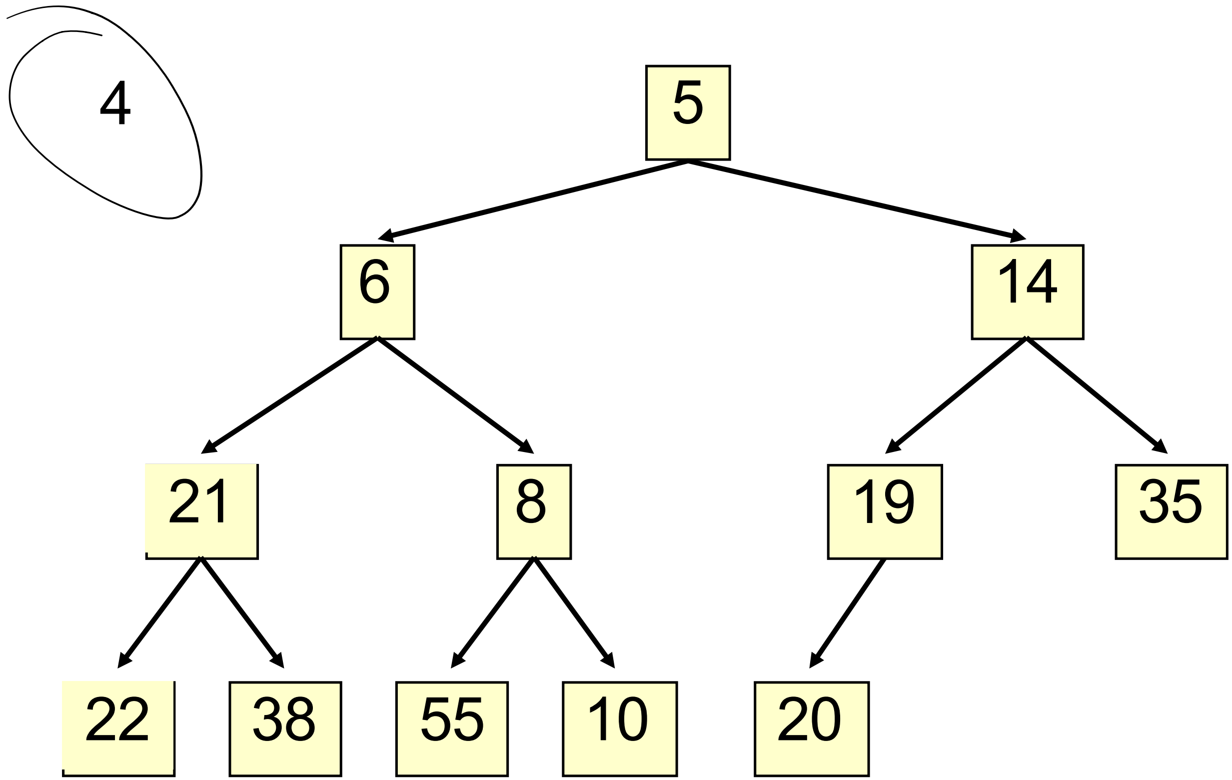
Bubble the root value down, swapping with the **smaller** child

```
V poll();
```



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

`V poll();`



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

Exercise: fill in the runtimes of the rest of the methods.

Heap operations - runtime

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

Details

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

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.

public interface Set

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)` 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` implements `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)` 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**.

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

- The built-in Java `PriorityQueue` interface:
  - Stores single 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 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 (only) 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
  for (int k = 0; k < b.length; k = k+1) {
    h.add(b[k]);
  }

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

n

← O(log n) } n

← O(log n) } n

O(n log n)

Heapsort

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

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

Worst-case runtime:

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) !