Middlebury

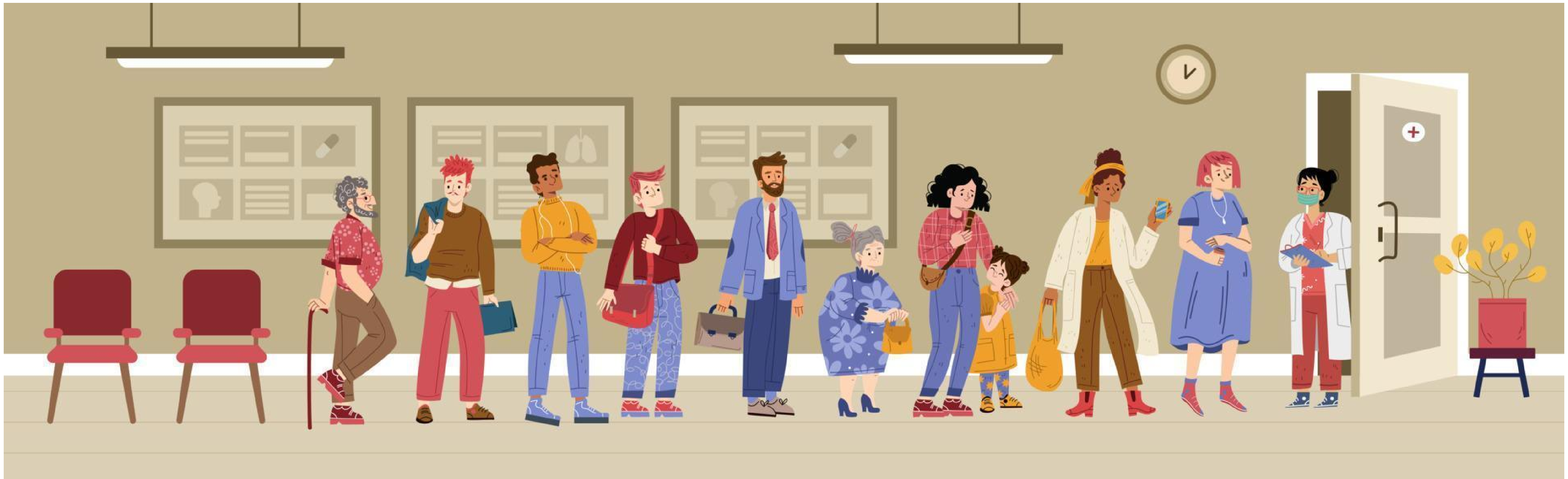
CSCI 201: Data Structures

Fall 2024

Lecture 8R: Heaps, priority queues

Goals for today:

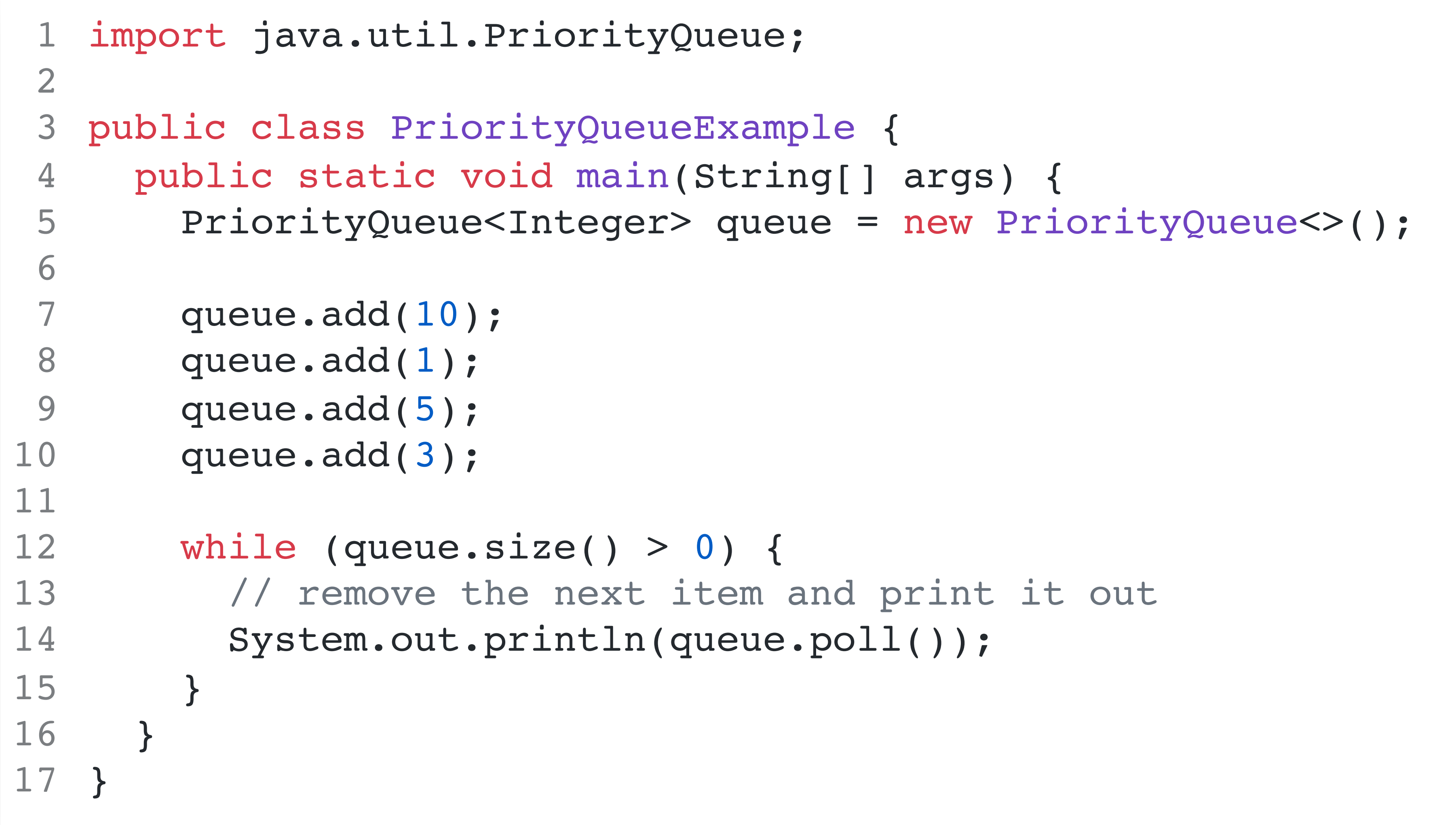
- **Motivation:** queues where items are `remove`d according to a *priority* (**priority queues**).
- How would we design a **priority queue** using the structures we've learned so far?
- Use a **complete binary tree** to implement a **heap** (min & max).
- Represent a complete binary tree using an **array**.



A priority queue is an abstract data type which can be implemented with different data structures. But what should we use?

Main things we want:

- Ability to `add` a new item into a priority queue.
- Ability to **query** (`peek`) or **remove** (`poll`) next item with highest priority.



```java
import java.util.PriorityQueue;

public class PriorityQueueExample {
  public static void main(String[] args) {
    PriorityQueue<Integer> queue = new PriorityQueue<>();

    queue.add(10);
    queue.add(1);
    queue.add(5);
    queue.add(3);

    while (queue.size() > 0) {
      // remove the next item and print it out
      System.out.println(queue.poll());
    }
  }
}
```

- How does this work?
- For regular queues, we used either `ArrayList` or `LinkedList`.
- Let's try using these for priority queues too.
- But how do we find the *highest* priority item?

Idea 1: look for it!

Idea 2: keep the items sorted!

Exploring these ideas with an ArrayList or LinkedList.

(hpi)

💡 **Idea 1:** (unsorted) add to beginning or end, *look* for highest priority item.

- ArrayList: add to end: O(1) (amortized) search for hpi: O(n)

- LinkedList: with tail add to end: O(1) search for hpi: O(n)

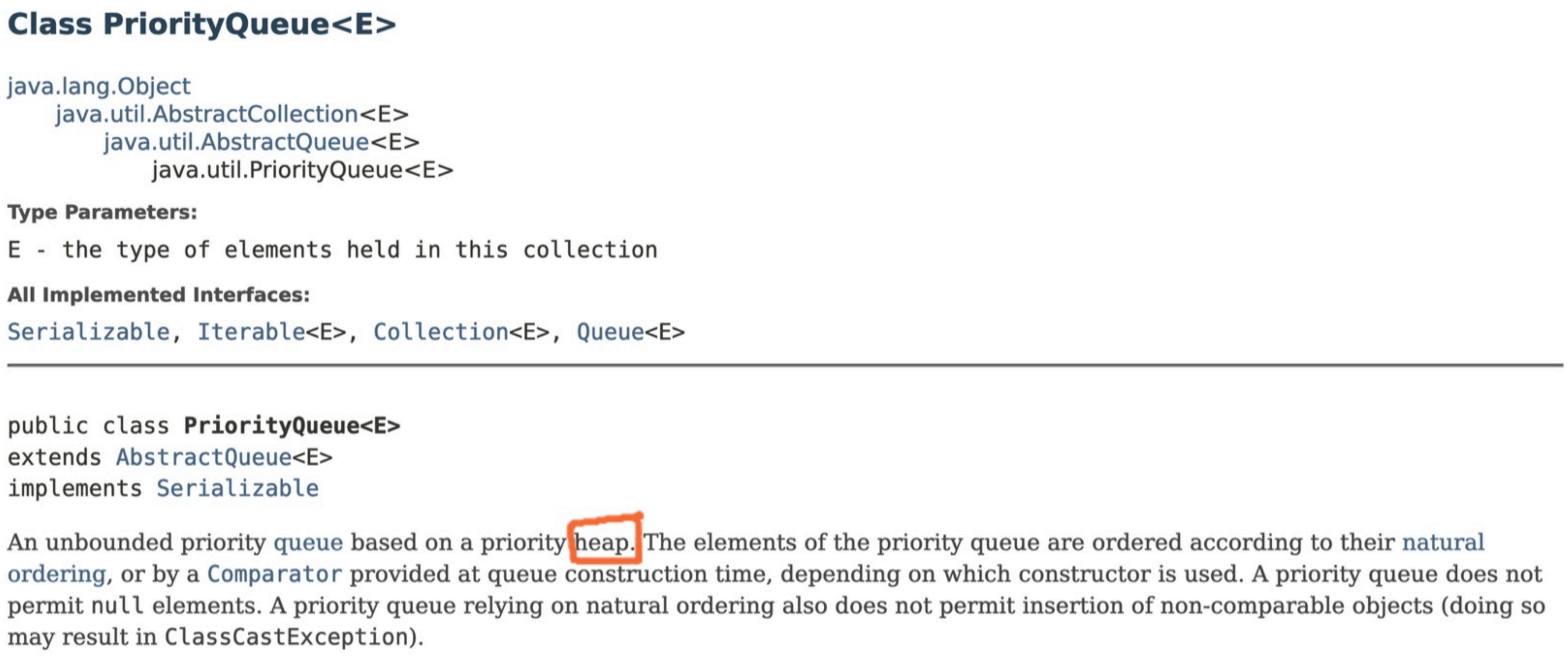
💡 **Idea 2:** (sorted) add in appropriate place, remove highest priority item from beginning or end.

- ArrayList: add : O(n) remove from end: O(1)

- LinkedList: with tail add: O(n) remove from beginning/end: O(1)

Is there a way to have something in between $O(1)$ and $O(n)$ for both add and poll?

Yes. We can use a *heap* (like what Java uses).



Class PriorityQueue<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractQueue<E>
            java.util.PriorityQueue<E>

Type Parameters:

`E - the type of elements held in this collection`

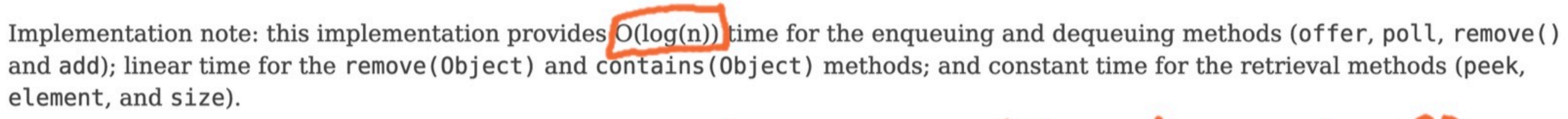
All Implemented Interfaces:

`Serializable, Iterable<E>, Collection<E>, Queue<E>`

```
public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable
```

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time, depending on which constructor is used. A priority queue does not permit `null` elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in `ClassCastException`).

https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

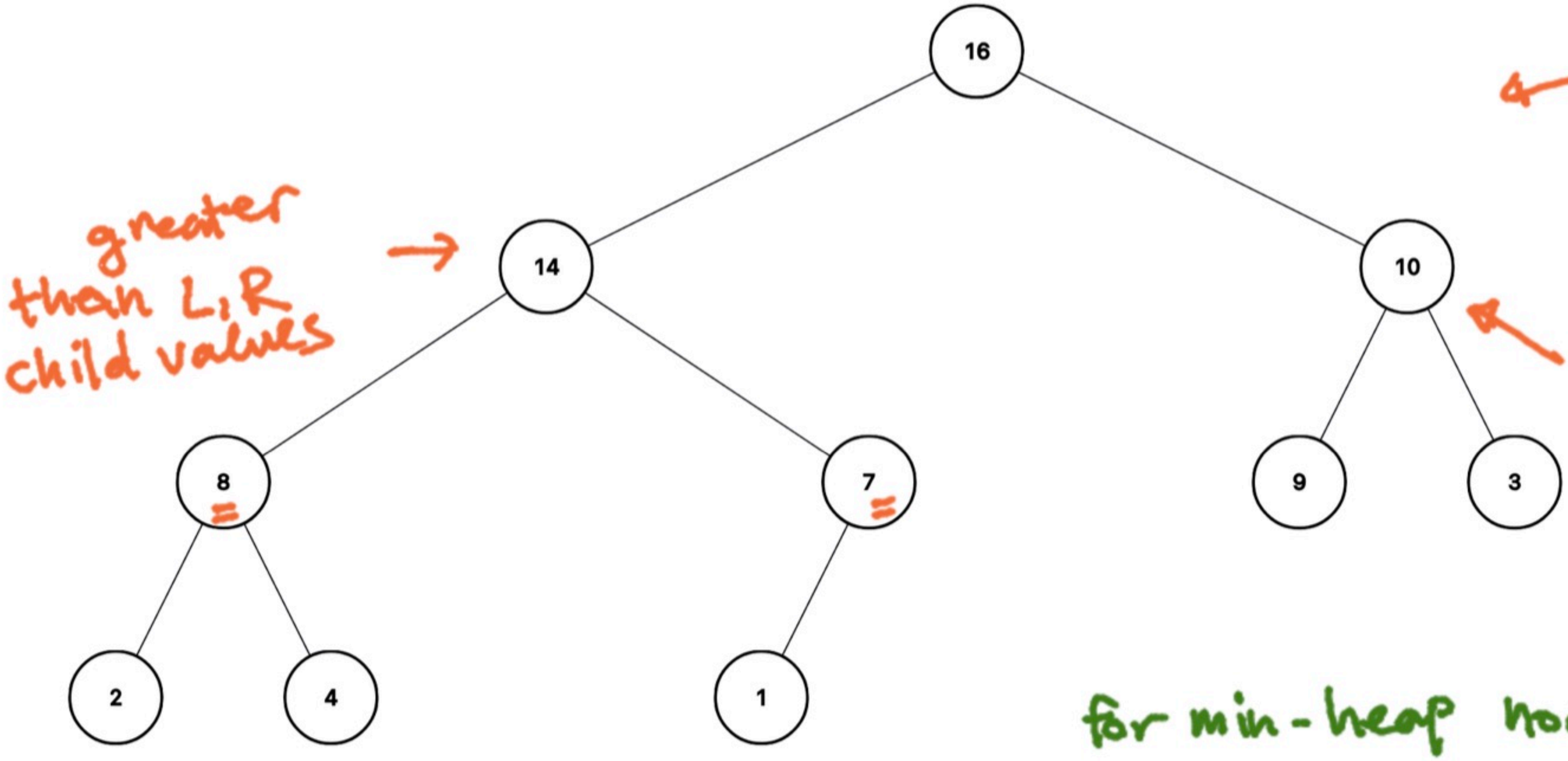


Implementation note: this implementation provides O(log(n)) time for the enqueing and dequeing methods (`offer`, `poll`, `remove()` and `add`); linear time for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

both add + poll are O(log n)   how??

A heap is a binary tree with two extra properties.

1. It is complete. → all levels filled except (possibly) last level, which is filled from left to right
2. It satisfies a *heap property*:
    - **For a max-heap:** Every node value is *greater* than (or equal to) the values of its child nodes. So the root is the largest!
    - **For a min-heap:** Every node value is *less* than (or equal to) the values of its child nodes. So the root is the smallest!
    - We need to maintain this property when adding to (`add`) or removing from (`poll`) the heap.



greater than L,R child values →

← example of a max-heap (root is largest value)

greater than L,R child values

for min-heap nodes are ≤ child values.

→

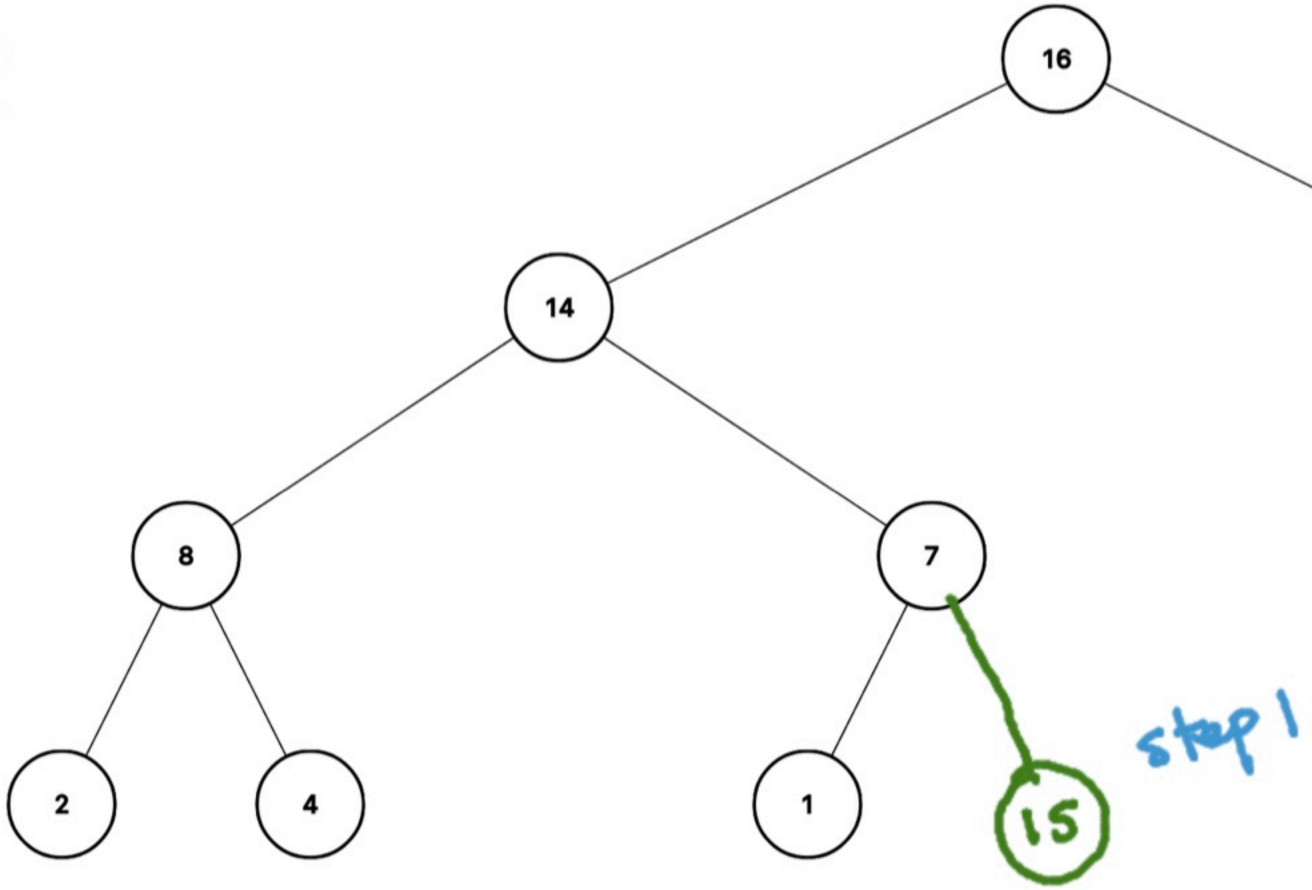
filled left → right on last level.

Adding (add) a value to the heap:

1. Make a new leaf node (maintaining a complete binary tree) to hold this value.
2. Set the current node to this new leaf node.
3. while heap property not satisfied:
    - Swap the values of the current node with the parent node.
    - Set the current node to the parent node.

add(15)

max-heap

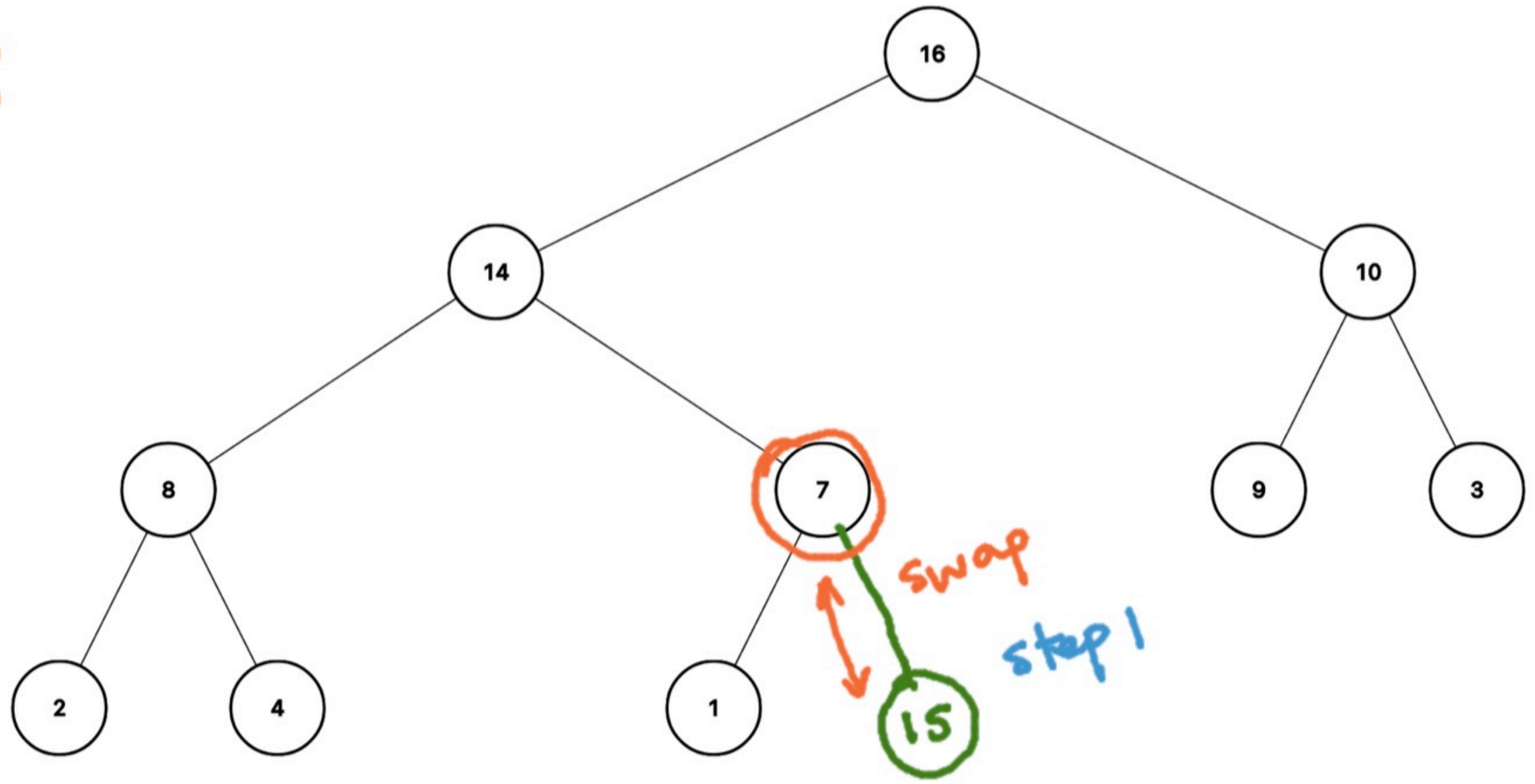


Adding (add) a value to the heap:

1. Make a new leaf node (maintaining a complete binary tree) to hold this value.
2. Set the current node to this new leaf node.
3. while heap property not satisfied:
    - Swap the values of the current node with the parent node.
    - Set the current node to the parent node.

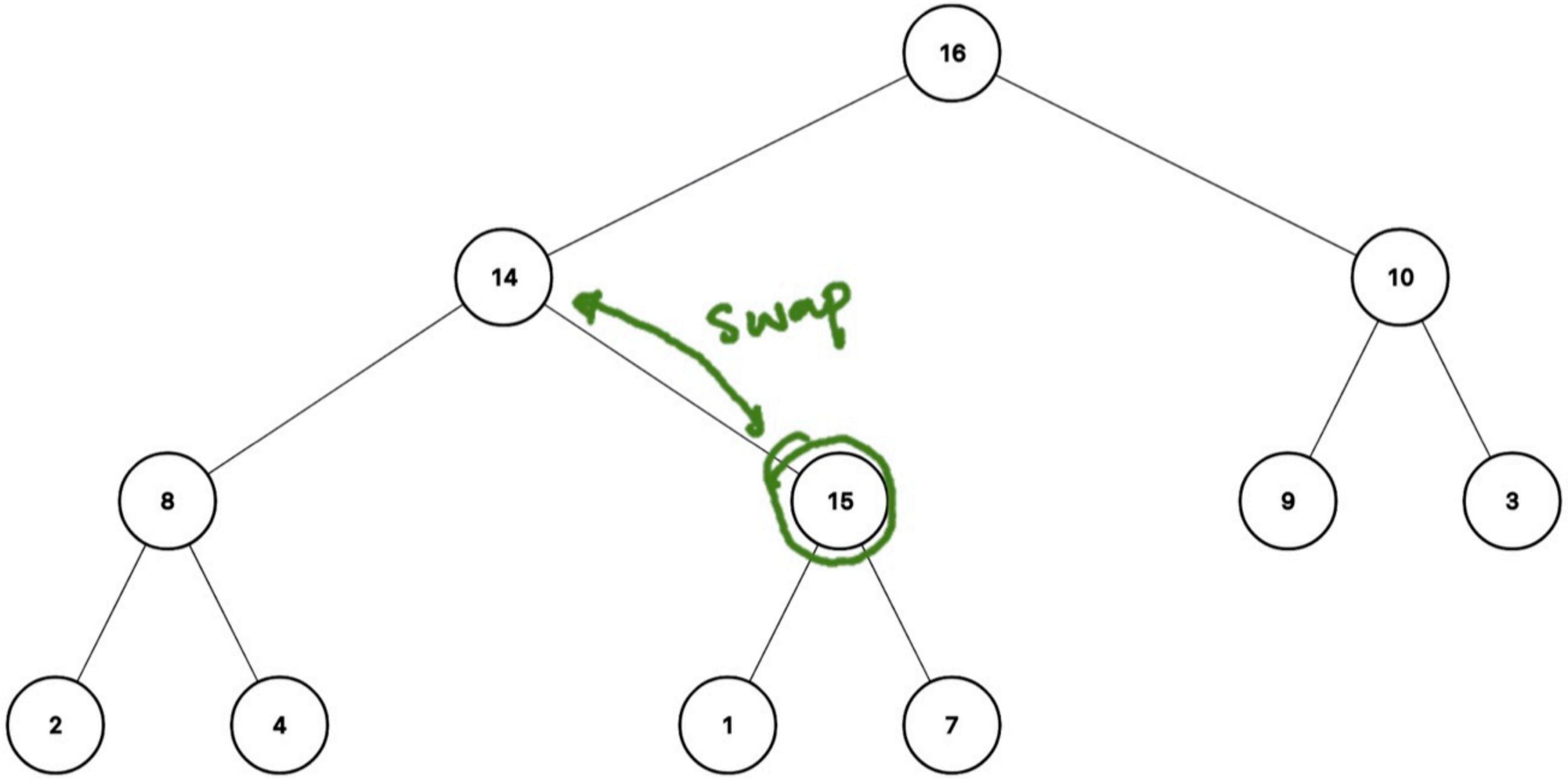
add(15)

max-heap



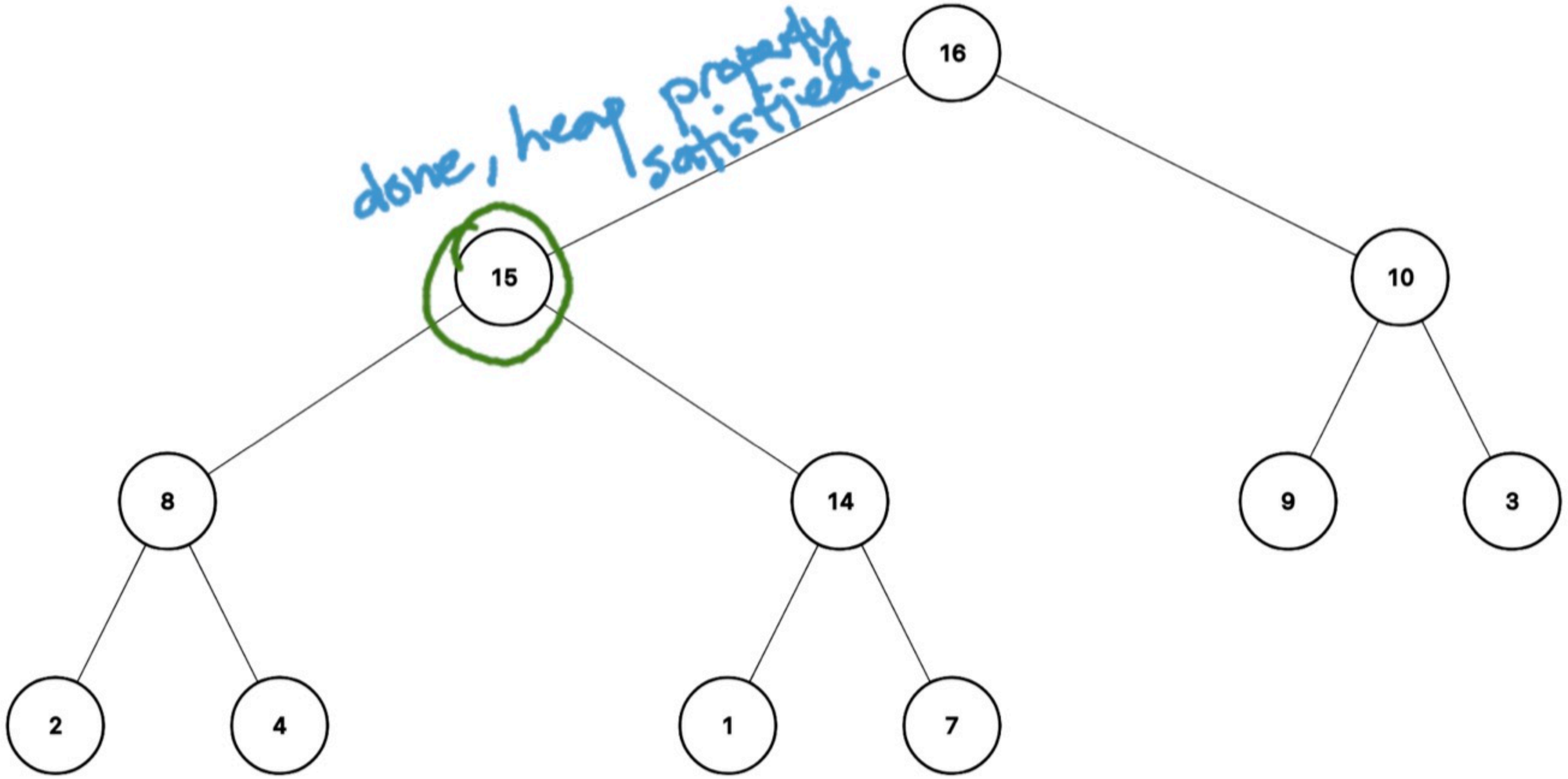
Adding (add) a value to the heap:

1. Make a new leaf node (maintaining a complete binary tree) to hold this value.
2. Set the current node to this new leaf node.
3. `while` heap property not satisfied:
    - Swap the values of the current node with the parent node.
    - Set the current node to the parent node.

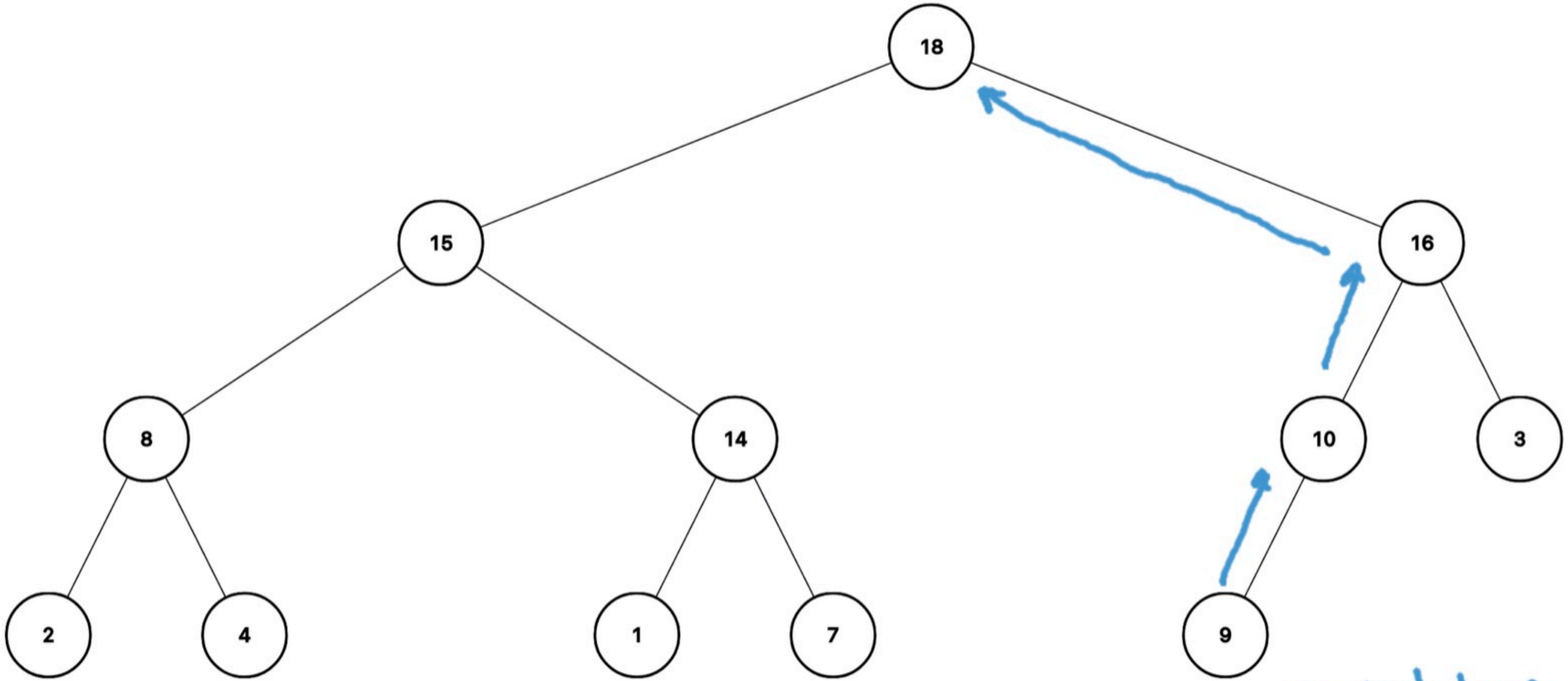


Adding (add) a value to the heap:

1. Make a new leaf node (maintaining a complete binary tree) to hold this value.
2. Set the current node to this new leaf node.
3. while heap property not satisfied:
    - Swap the values of the current node with the parent node.
    - Set the current node to the parent node.



Exercise: add the value 18 to the heap, i.e. `add(18)`.



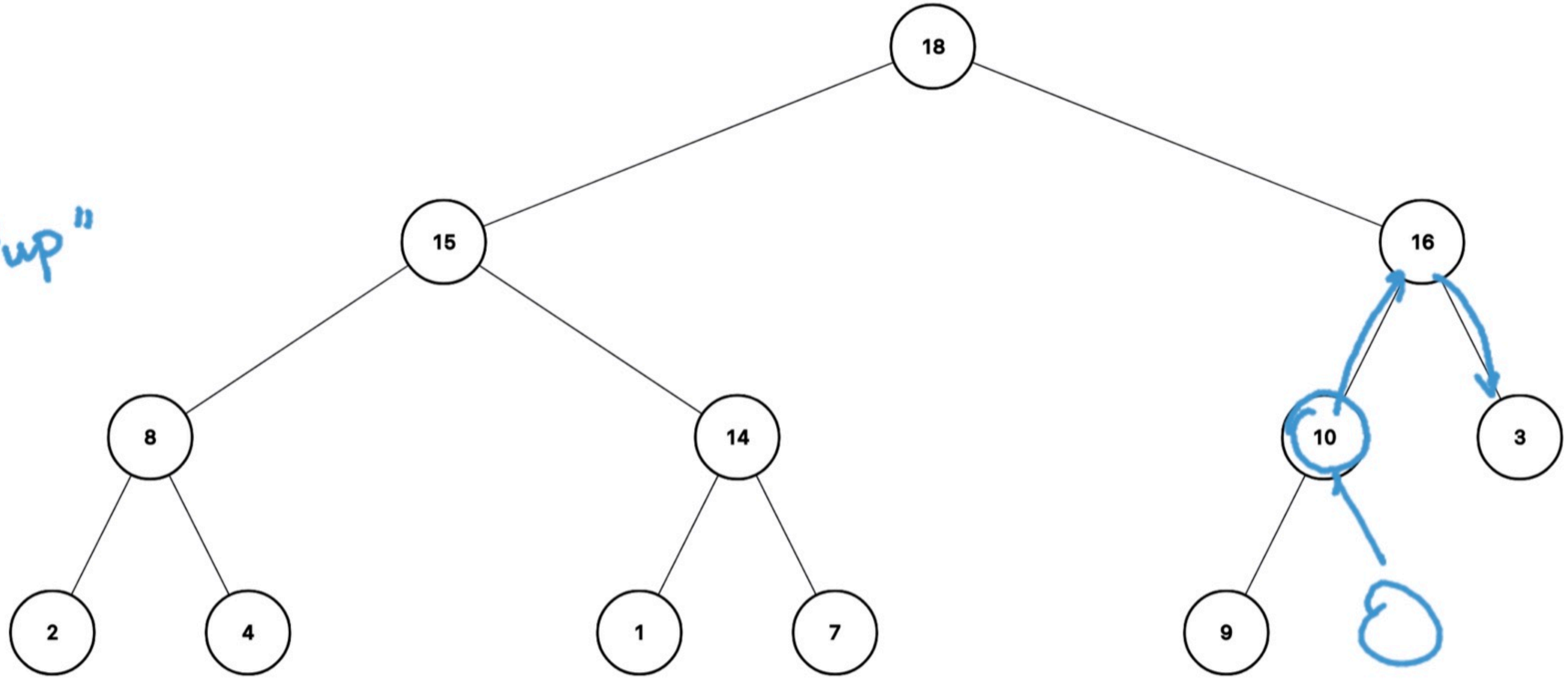
started here

complexity of add: $O(\text{height})$

$O(\log n)$

Exercise: add the value 18 to the heap, i.e. add(18).

heapify "up"

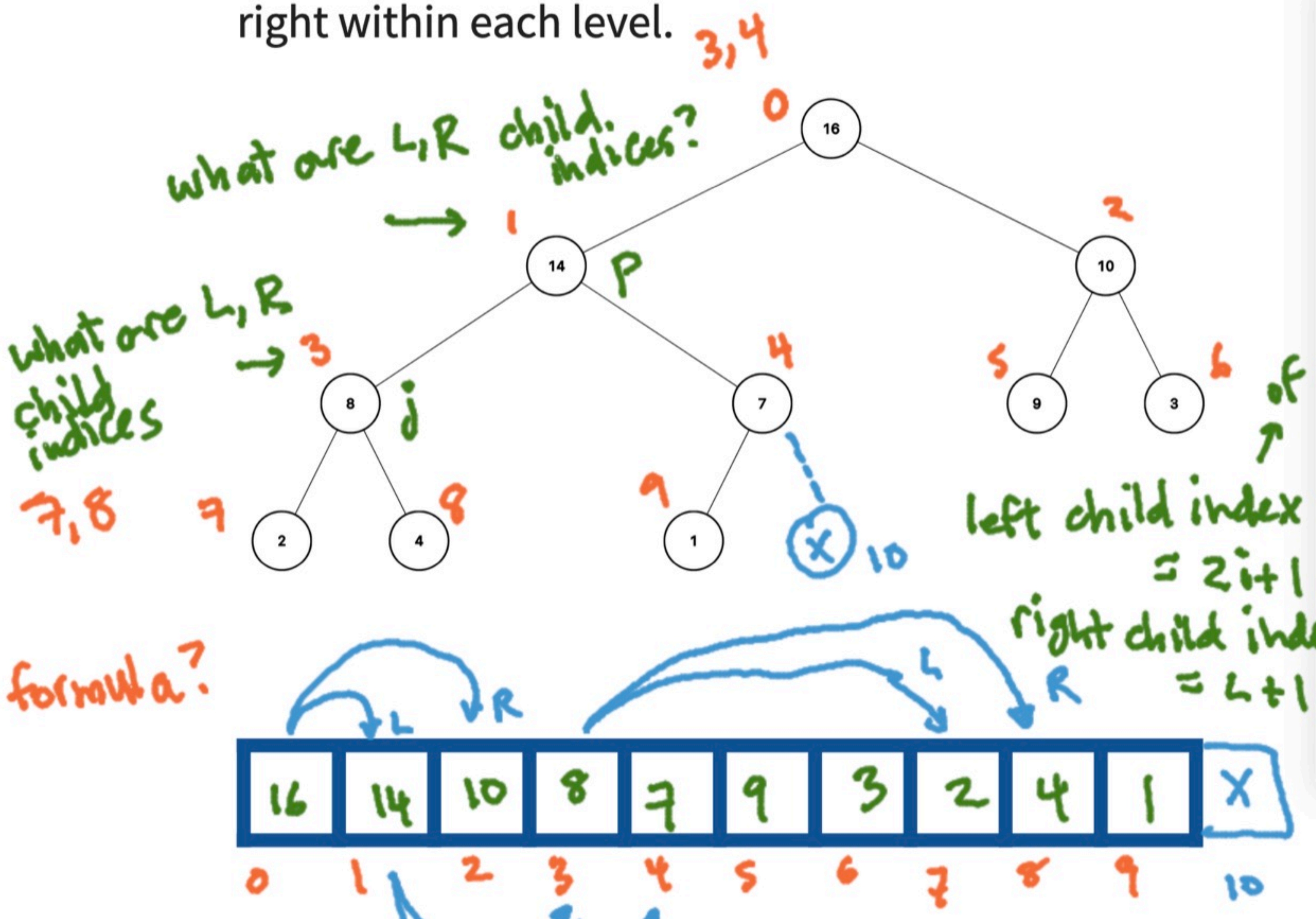


How are we going to implement this?

- We need to be able to go "up": can keep track of parent of each node.
- We also need to be able to "find" the last leaf node:

Alternatively, complete binary trees can be represented nicely with an array.

Main idea: index nodes top-to-bottom and left-to-right within each level.



what are L,R child indices?

what are L,R child indices

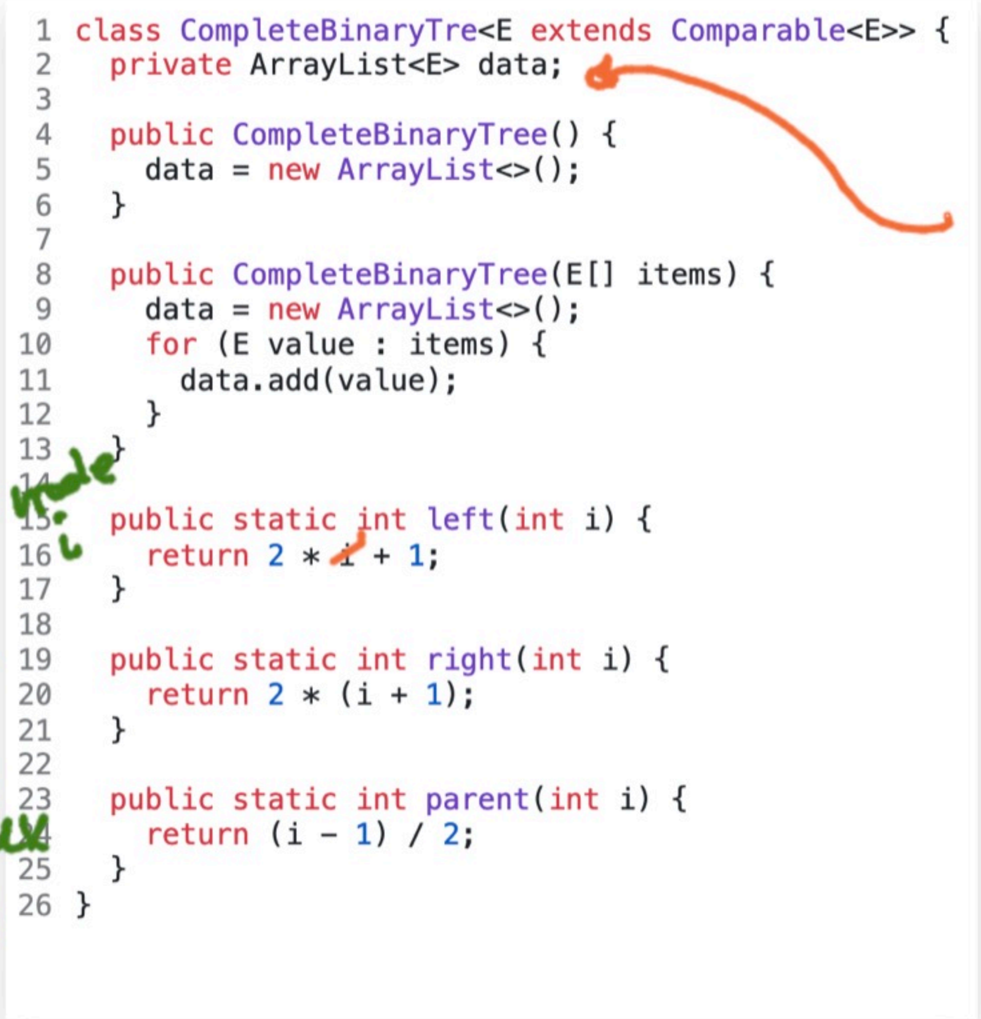
formula?

left child index = 2i+1

right child index = L+1

what about parent?

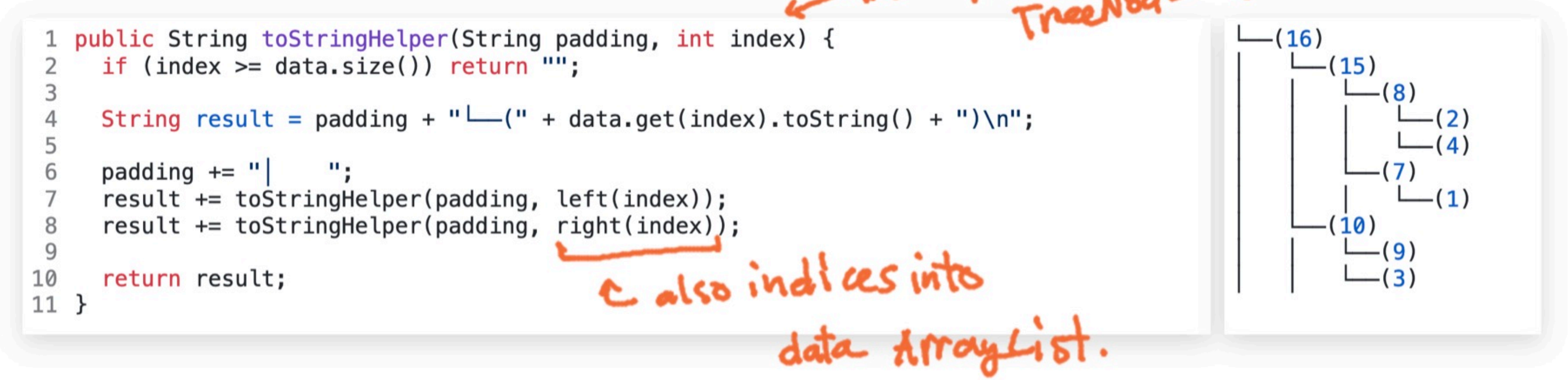
$j = 2p+1 \rightarrow p = \frac{j-1}{2}$



```java
class CompleteBinaryTre<E extends Comparable<E>> {
  private ArrayList<E> data;

  public CompleteBinaryTree() {
    data = new ArrayList<>();
  }

  public CompleteBinaryTree(E[] items) {
    data = new ArrayList<>();
    for (E value : items) {
      data.add(value);
    }
  }

  public static int left(int i) {
    return 2 * i + 1;
  }

  public static int right(int i) {
    return 2 * (i + 1);
  }

  public static int parent(int i) {
    return (i - 1) / 2;
  }
}
```

of node i

Printing the tree using pre-order traversal with our `CompleteBinaryTree` representation.

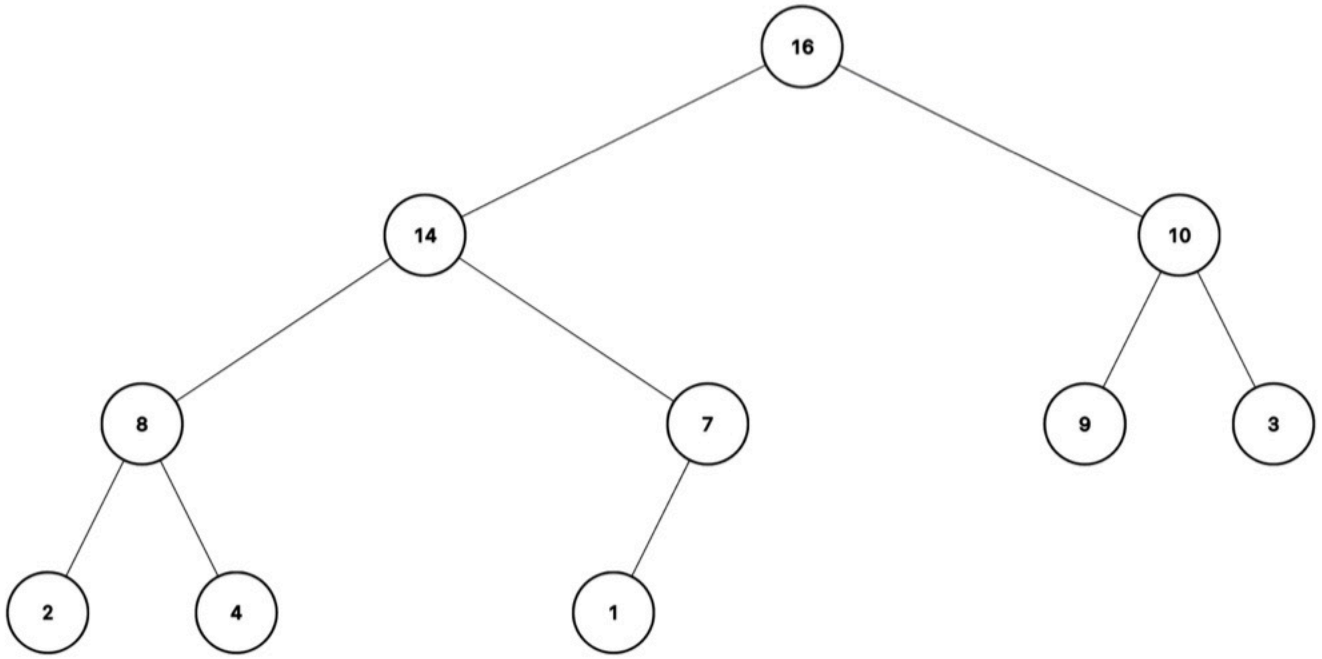


```java
public String toStringHelper(String padding, int index) {
  if (index >= data.size()) return "";

  String result = padding + "└─(" + data.get(index).toString() + ")\n";

  padding += "|    ";
  result += toStringHelper(padding, left(index));
  result += toStringHelper(padding, right(index));

  return result;
}
```

← index, not a TreeNode object

← also indices into data ArrayList.

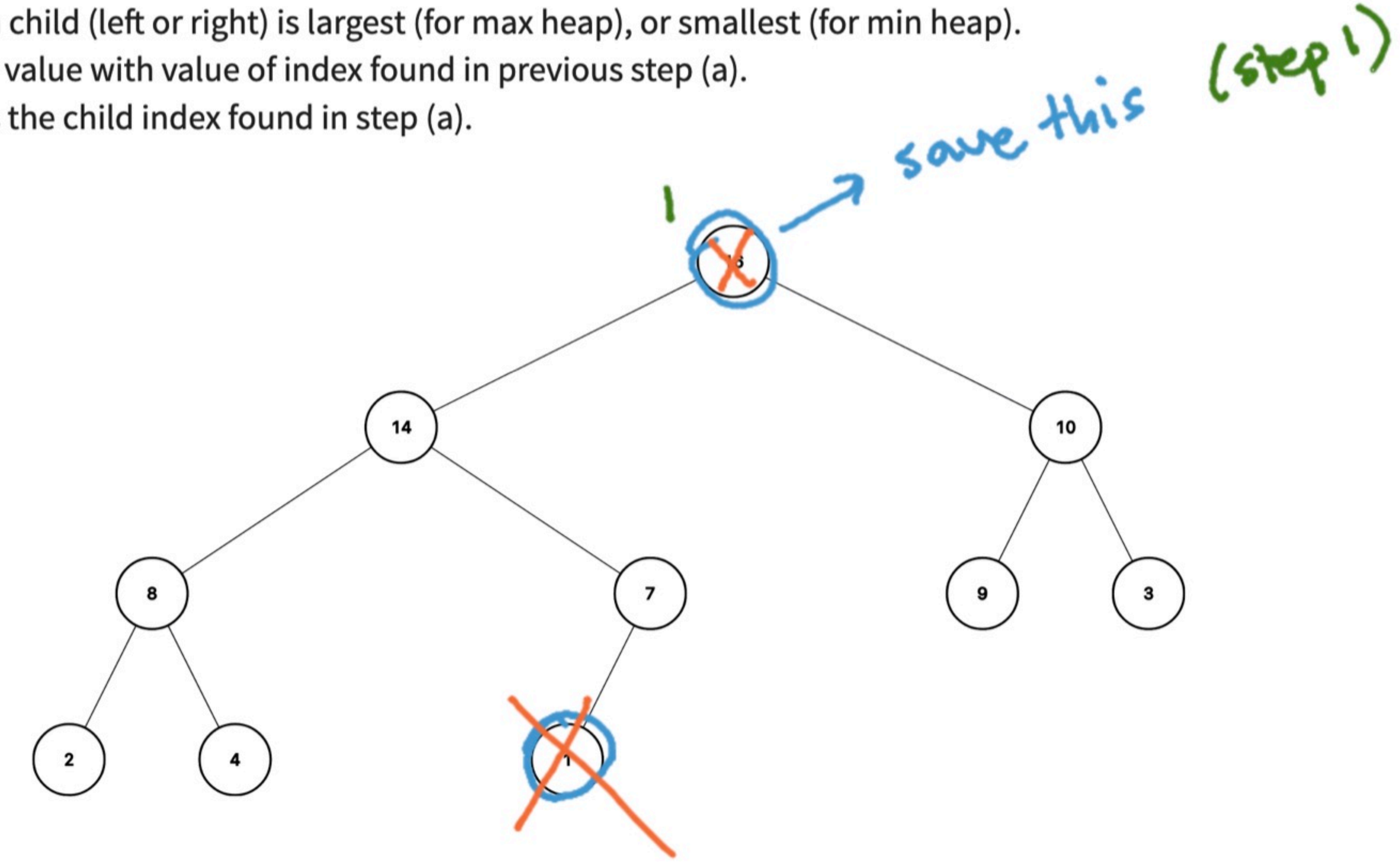
```
└─(16)
│   └─(15)
│   │   └─(8)
│   │   │   └─(2)
│   │   │   └─(4)
│   │   └─(7)
│   │   │   └─(1)
│   └─(10)
│   │   └─(9)
│   │   └─(3)
```



What about removing the top (i.e. highest priority) item from the heap? Implementing the `poll` method.

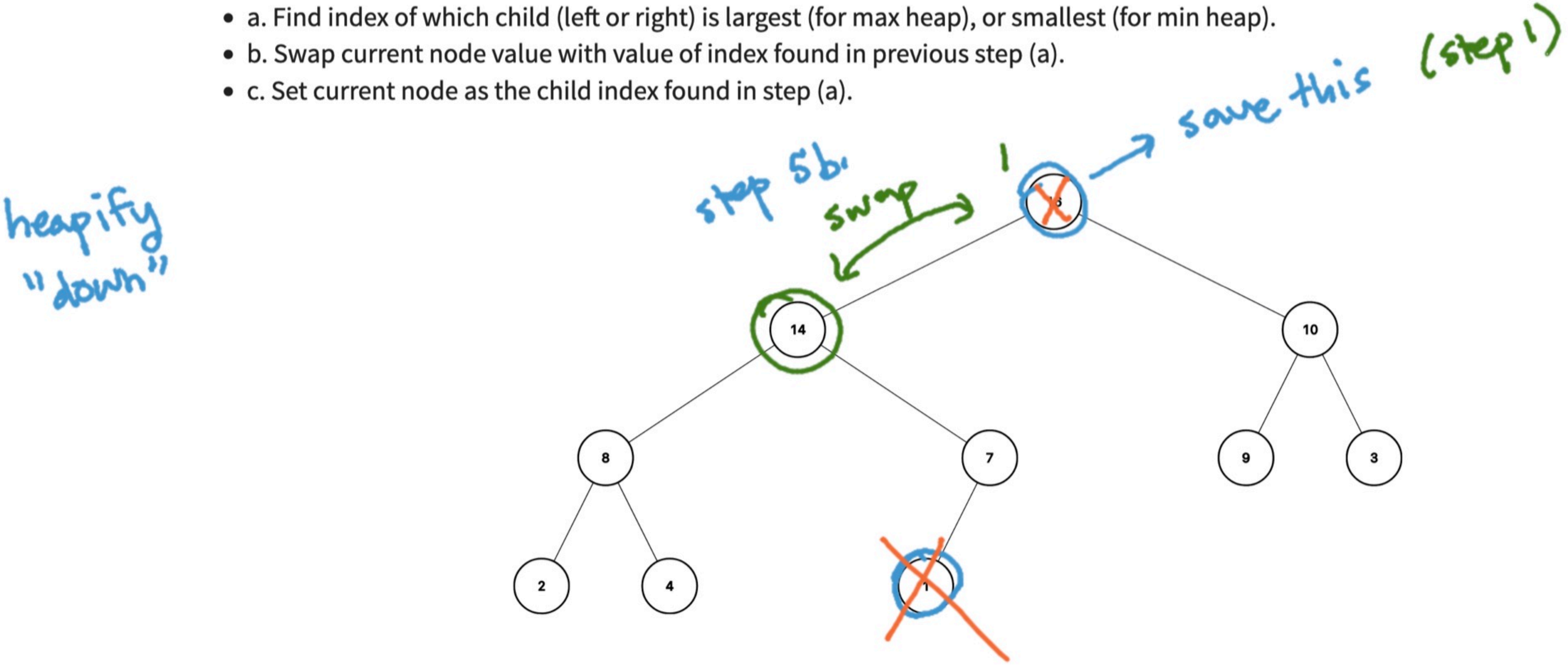
1. Save the root node value (so we can return it later).
2. Set the root node value to the value of *last* node (a leaf).
3. Remove this leaf node.
4. Set the current node as the root node.
5. `while` heap property not satisfied:
   - a. Find index of which child (left or right) is largest (for max heap), or smallest (for min heap).
   - b. Swap current node value with value of index found in previous step (a).
   - c. Set current node as the child index found in step (a).

heapify "down"



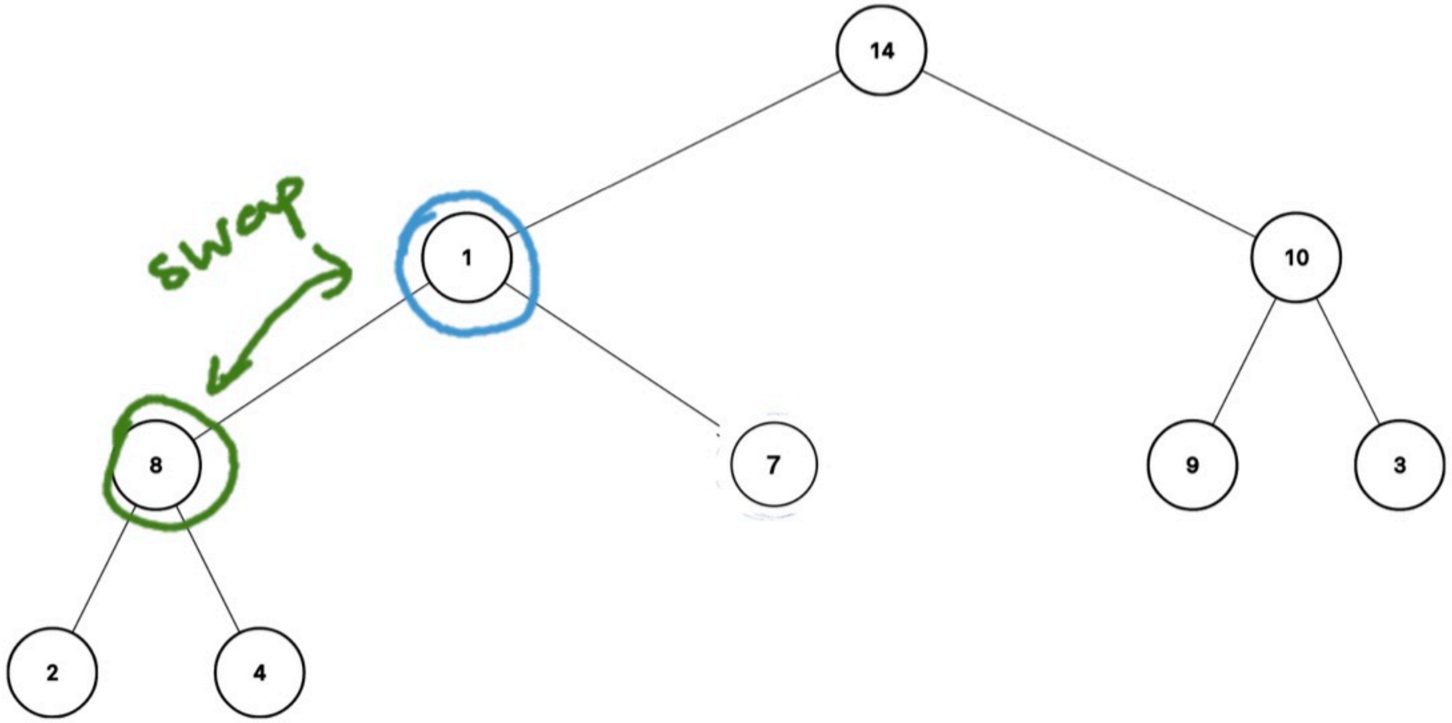
What about removing the top (i.e. highest priority) item from the heap? Implementing the `poll` method.

1. Save the root node value (so we can return it later).
2. Set the root node value to the value of *last* node (a leaf).
3. Remove this leaf node.
4. Set the current node as the root node.
5. `while` heap property not satisfied:
    - a. Find index of which child (left or right) is largest (for max heap), or smallest (for min heap).
    - b. Swap current node value with value of index found in previous step (a).
    - c. Set current node as the child index found in step (a).



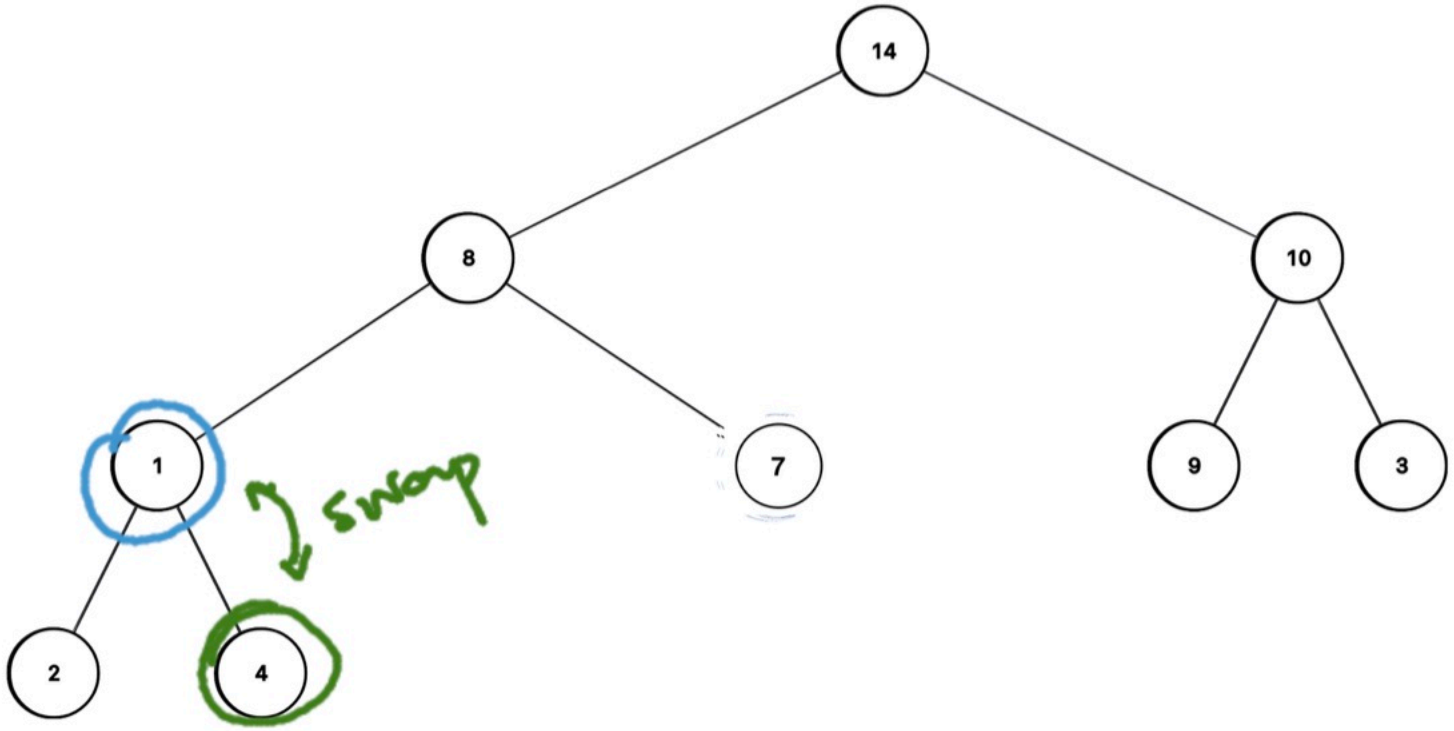
What about removing the top (i.e. highest priority) item from the heap? Implementing the `poll` method.

1. Save the root node value (so we can return it later).
2. Set the root node value to the value of *last* node (a leaf).
3. Remove this leaf node.
4. Set the current node as the root node.
5. `while` heap property not satisfied:
    - a. Find index of which child (left or right) is largest (for max heap), or smallest (for min heap).
    - b. Swap current node value with value of index found in previous step (a).
    - c. Set current node as the child index found in step (a).



What about removing the top (i.e. highest priority) item from the heap? Implementing the `poll` method.

1. Save the root node value (so we can return it later).
2. Set the root node value to the value of *last* node (a leaf).
3. Remove this leaf node.
4. Set the current node as the root node.
5. `while` heap property not satisfied:
    - a. Find index of which child (left or right) is largest (for max heap), or smallest (for min heap).
    - b. Swap current node value with value of index found in previous step (a).
    - c. Set current node as the child index found in step (a).

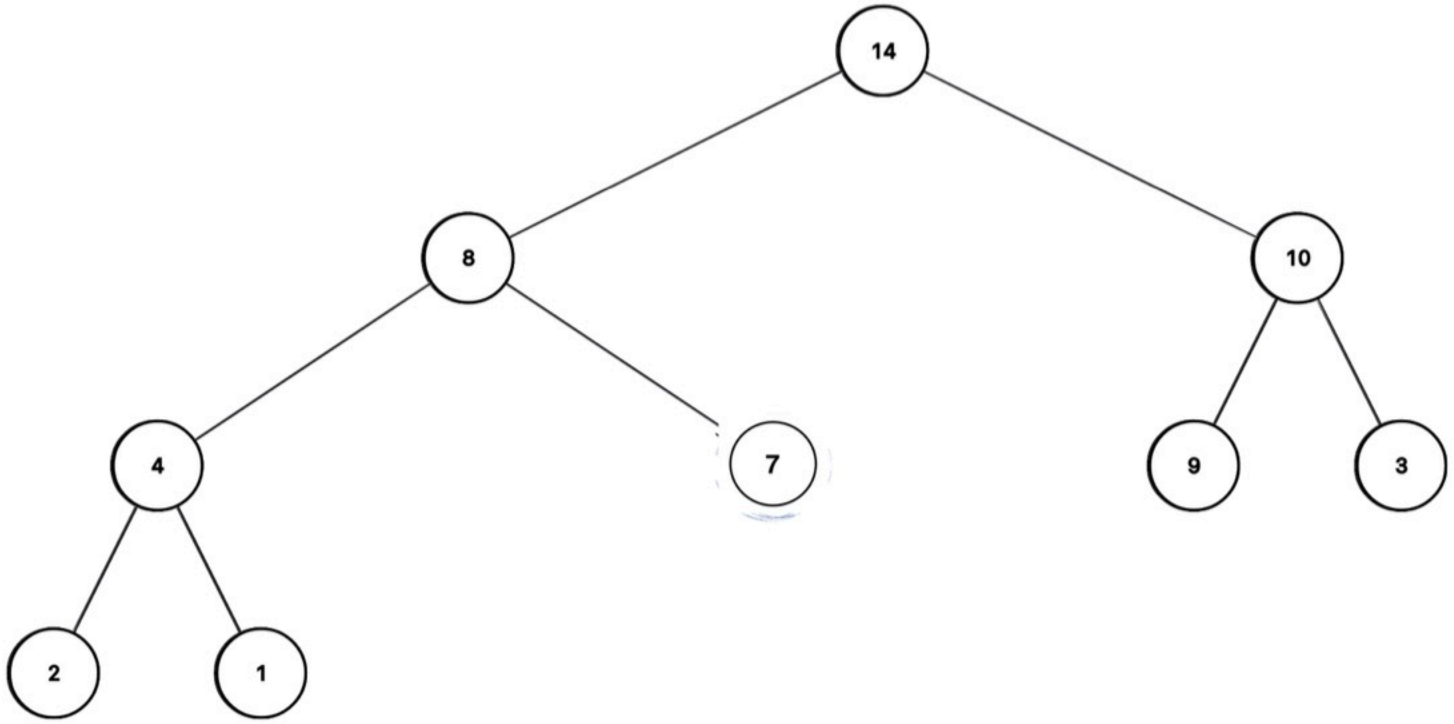


What about removing the top (i.e. highest priority) item from the heap? Implementing the `poll` method.

1. Save the root node value (so we can return it later).
2. Set the root node value to the value of *last* node (a leaf).
3. Remove this leaf node.
4. Set the current node as the root node.
5. `while` heap property not satisfied:
    - a. Find index of which child (left or right) is largest (for max heap), or smallest (for min heap).
    - b. Swap current node value with value of index found in previous step (a).
    - c. Set current node as the child index found in step (a).

complexity of poll ?

O(logn)



Exercise: complete the isMaxHeap method for the CompleteBinaryTree class.

Assume we are checking the **max-heap property** (node values $\geq$ child values).

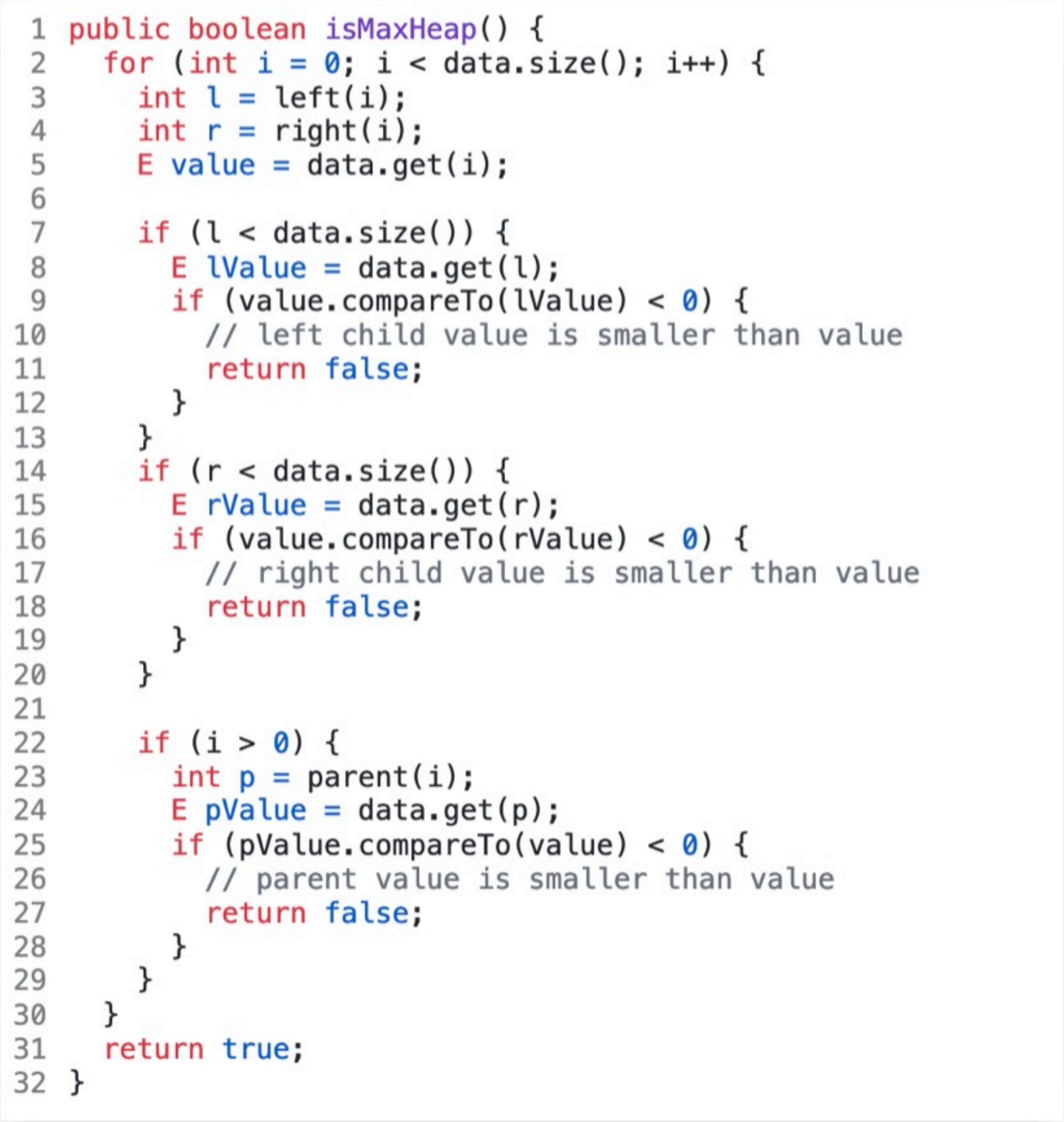
- Loop through all nodes (entire size() of data ArrayList).
- Retrieve indices of left and right children and check heap property.

x. compareTo (y)

x < y -1

x > y +1

x == y 0



```java
public boolean isMaxHeap() {
  for (int i = 0; i < data.size(); i++) {
    int l = left(i);
    int r = right(i);
    E value = data.get(i);

    if (l < data.size()) {
      E lValue = data.get(l);
      if (value.compareTo(lValue) < 0) {
        // left child value is smaller than value
        return false;
      }
    }
    if (r < data.size()) {
      E rValue = data.get(r);
      if (value.compareTo(rValue) < 0) {
        // right child value is smaller than value
        return false;
      }
    }

    if (i > 0) {
      int p = parent(i);
      E pValue = data.get(p);
      if (pValue.compareTo(value) < 0) {
        // parent value is smaller than value
        return false;
      }
    }
  }
  return true;
}
```

Notes:

- **Homework 6** due tomorrow: implement a calculator (using a stack) & mid-semester check-in.
- **Lab 6 tomorrow:** use a priority queue to encode messages efficiently!
- If you want to make your own trees, have a look at this app: https://tree-visualizer.netlify.app/ (trees for today's class were made with it).
- Reminder that Noah (go/noah) and Smith (go/smith) have office hours throughout the week and the 201 Course Assistants have drop-in hours in the late afternoons/evenings (go/cshelp).
- Complete ET 8R by the end of today.