EEL 4837
Programming for Electrical Engineers II

Ivan Ruchkin
Assistant Professor
Department of Electrical and Computer Engineering
University of Florida at Gainesville
iruchkin@ece.ufl.edu
http://ivan.ece.ufl.edu

Heaps

Readings:

- Weiss 6.1–6.3
- Horowitz 2.4
- Cormen 6

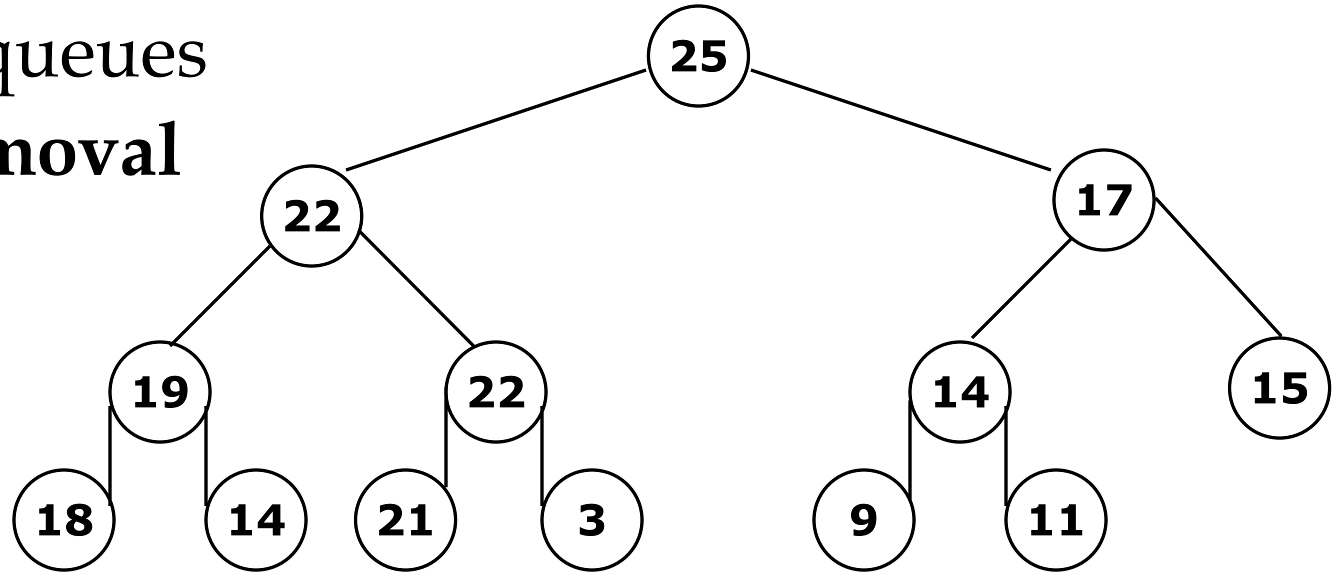
Heap Data Structure

In short, a **heap** is a *binary tree* with the following properties:

- Balanced
- Left-justified or Full
- (Max) **Heap property:** no node has a value greater than its parent

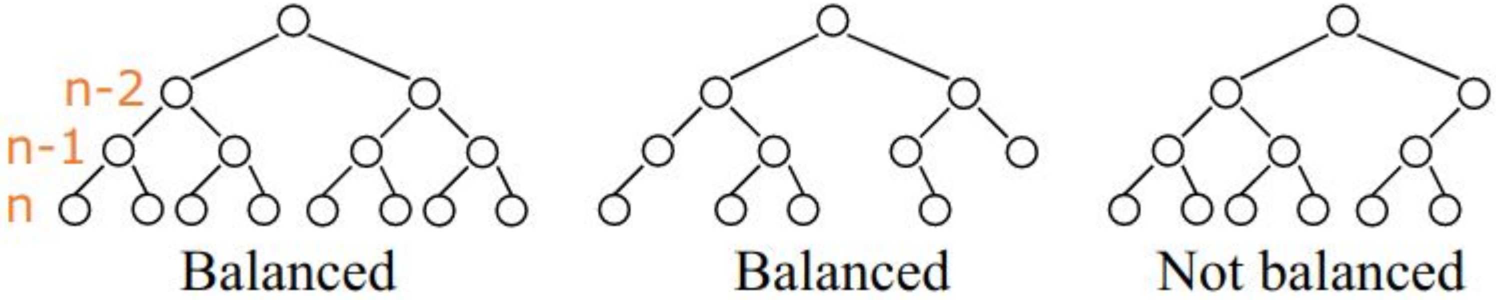
Motivation: sorting and priority queues

Want quick insertion and removal



Balanced Binary Trees

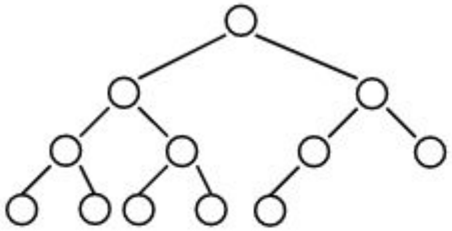
- Recall:
  - The **depth of a node** is its distance from the root
  - The **depth of a tree** is the depth of the deepest node
- A binary tree of depth n is called **balanced** if all the nodes at depths 0 through n-2 have two children
  - So only the leaves may be "missing"



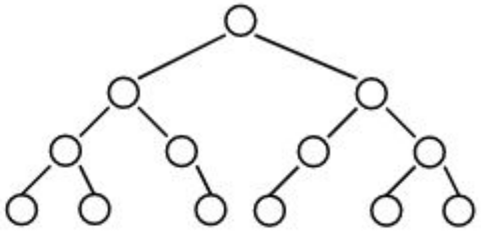
(and perfect)

Left-Justified Binary Trees

- A *balanced binary tree* of depth n is **left-justified** (or **complete**) if:
  - It has 2^n nodes at depth n (the tree is "full"), or
  - It has 2^k nodes at depth k, for all $k < n$, *and* all the leaves at depth n are *as far left as possible*



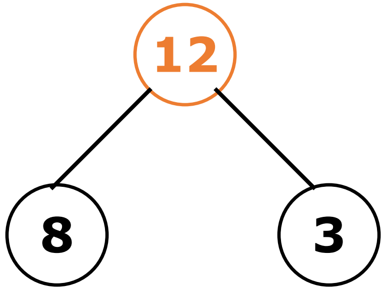
Left-justified



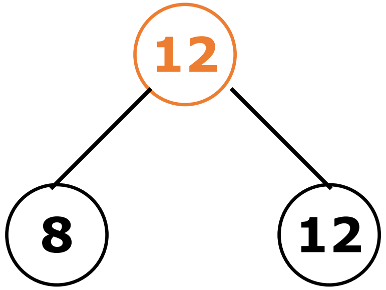
Not left-justified

The Heap Property

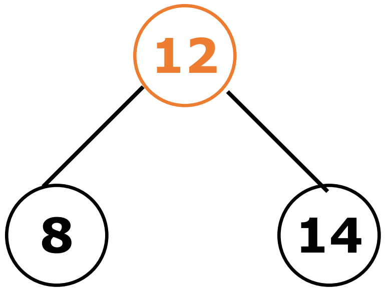
- A node has the (max) **heap property** if the value in the node is greater or equal than the values in its (immediate) children



Orange node has heap property



Orange node has heap property



Orange node does not have heap property

- All leaf nodes automatically have the heap property
- A binary tree is a **heap** if **all** of its nodes have the heap property

Building Up to Heapsort

- How to build a heap
- How to maintain a heap
- How to use a heap to sort data

Midterm Course Evaluations

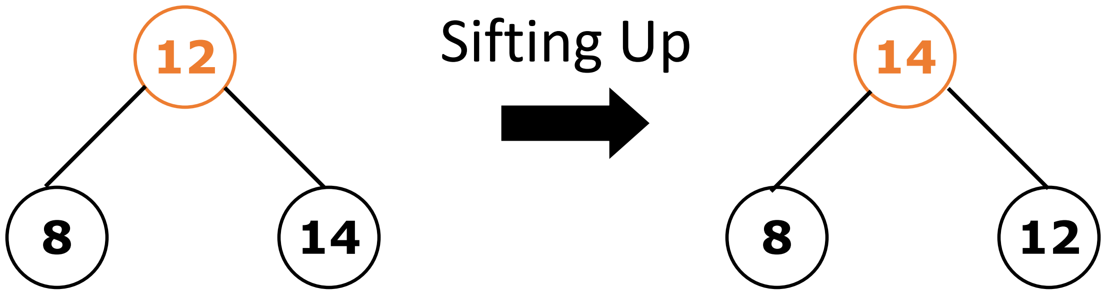
ECE | Florida
Electrical & Computer Engineering

- Midterm course evaluation now open
  - Access at https://ufl.bluera.com/ufl/
  - Evaluation Period- February 20-March 3

- Please complete as soon as possible
- Gives me an opportunity to incorporate valuable feedback before the end of the semester!

siftUp

- Given a node that does **not** have the heap property, **sift up** is giving it the heap property by exchanging its value with **the value of the larger child**

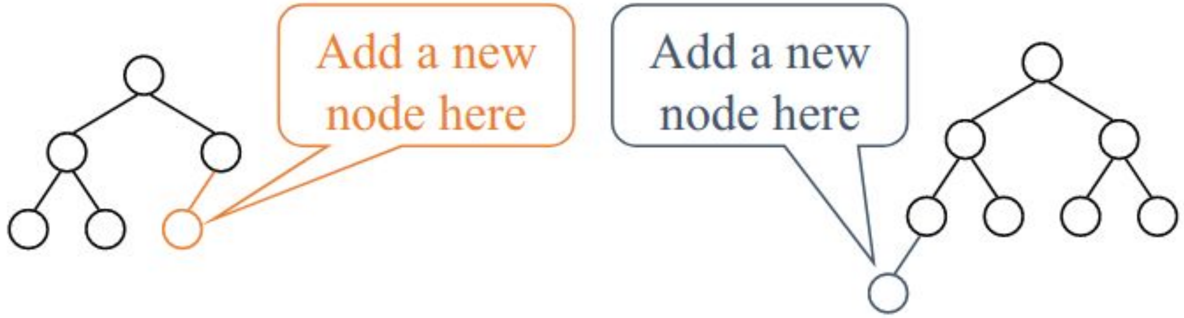


Orange node does not have heap property

Orange node has heap property

Constructing a Heap I

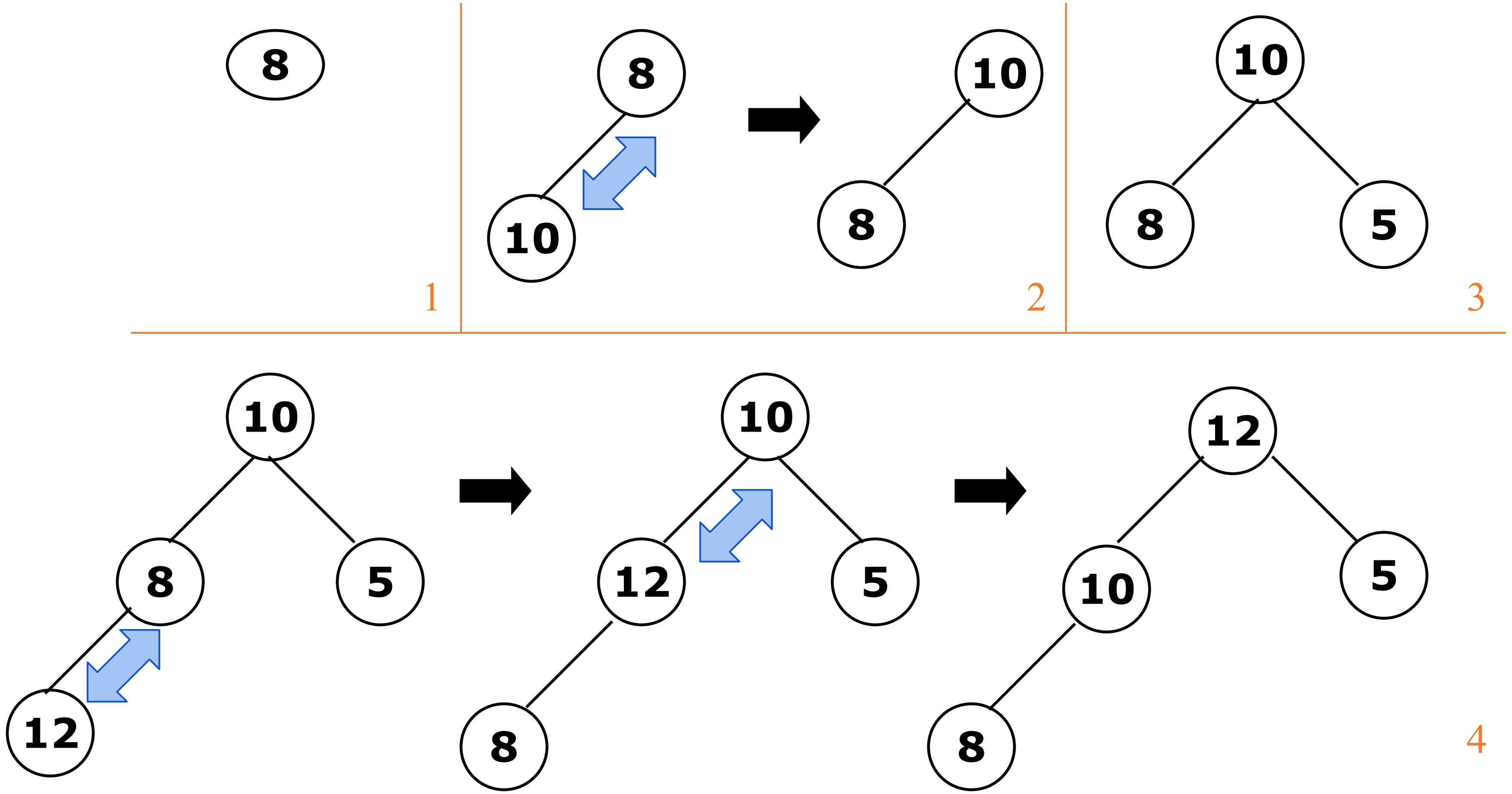
- *Base case:* a tree consisting of a single node is automatically a heap
- We construct a heap by adding nodes *one at a time*:
  - Add the node just to the **right of the rightmost node in the deepest level**
  - If the deepest level is full, **start a new level**
- *Example:*



Constructing a Heap II

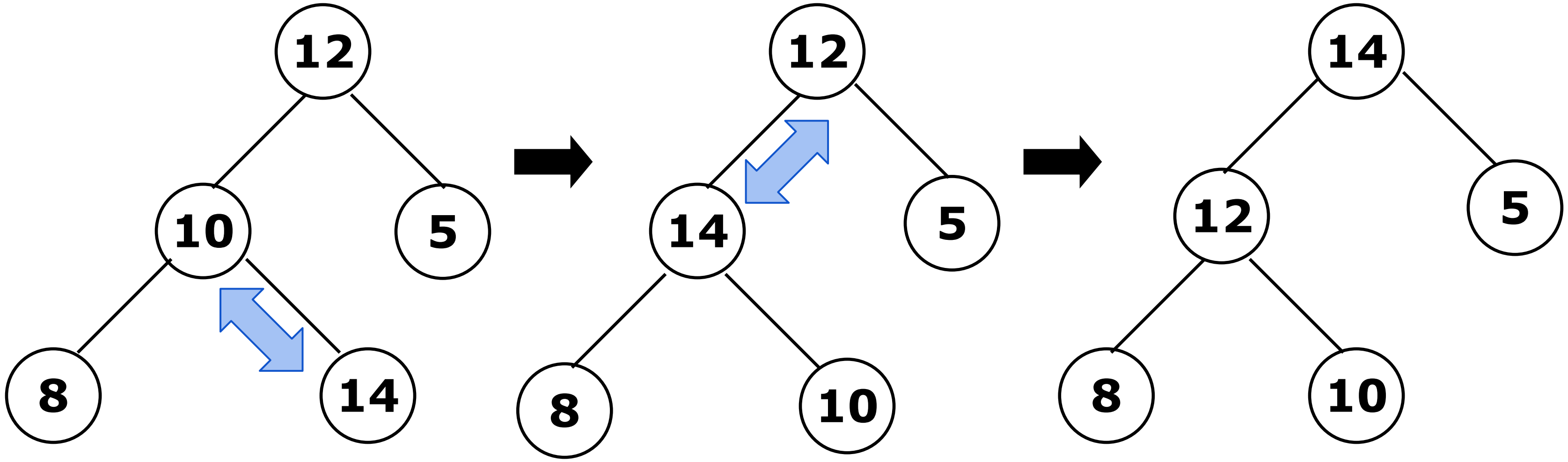
- Each time we add a node, we may destroy the heap property of *its parent node*
- To fix this, we **sift up** after adding
- But when we sift up, the value of the *top node* in the sift may increase, and this may destroy the heap property of **its** parent node
- So we **repeat** the sifting up, moving up in the tree, until either
  - We reach nodes whose values **don't need** to be swapped (because the parent is *still* larger than both children), or
  - We reach the **root**

Constructing a Heap III



Now let's add 14 to the heap

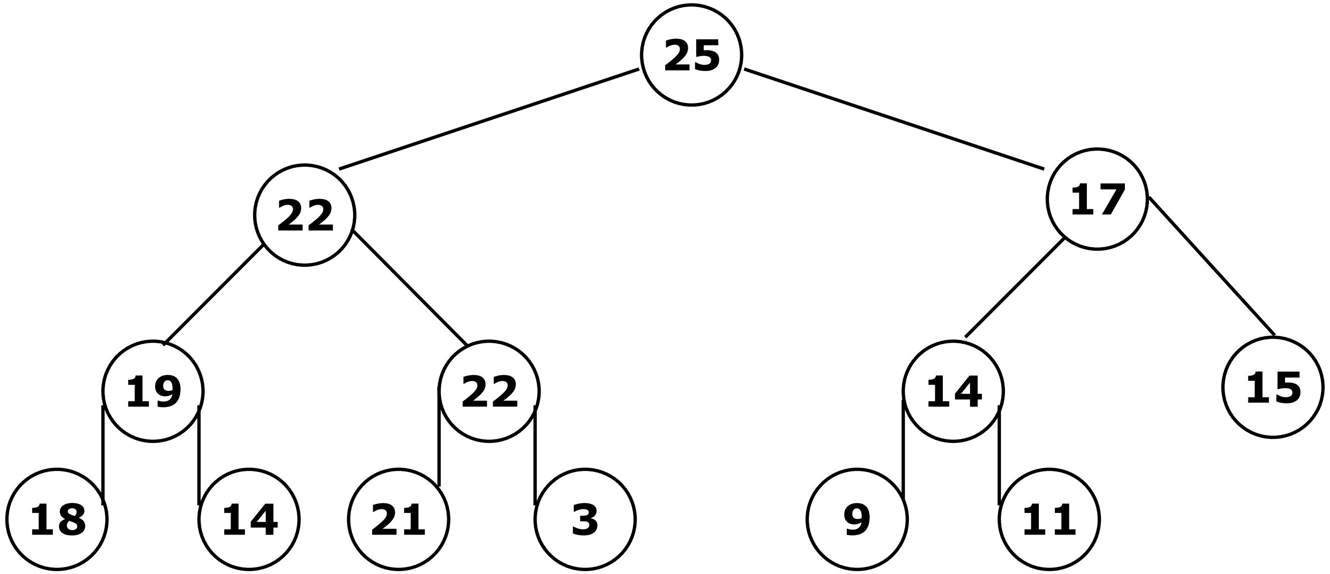
Other Children Are Not Affected



- The node containing 8 is not affected because its parent gets larger, not smaller
- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

The sifting up does not require/propagate changes through most of the heap

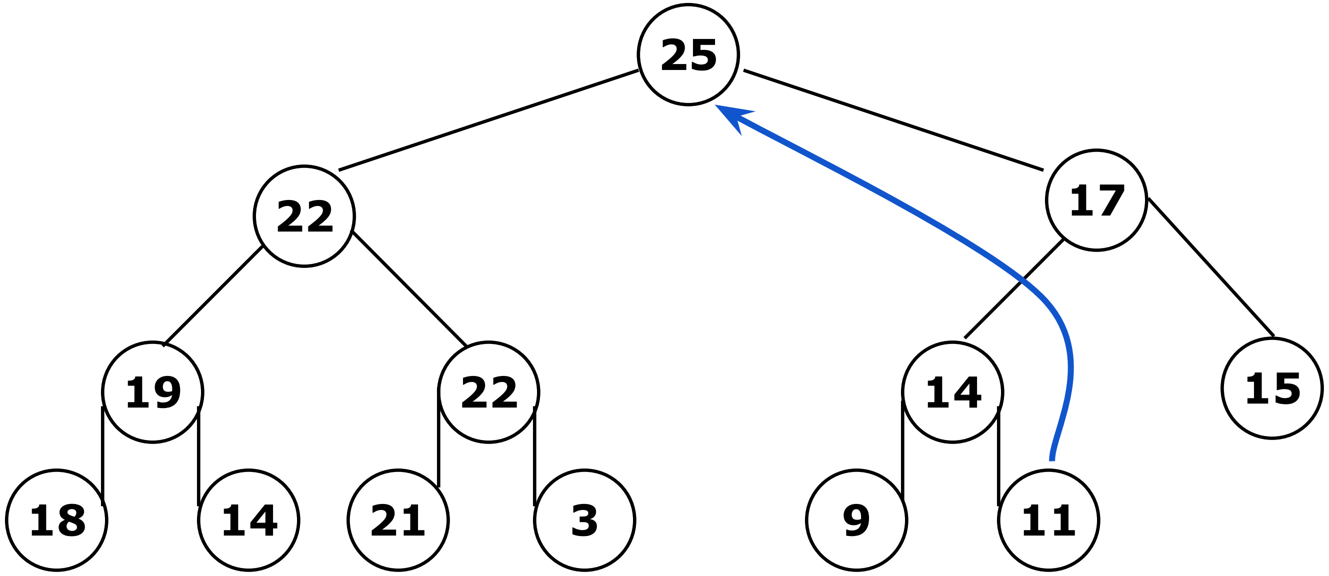
A Sample Heap



- Notice that **heapified** does **not** mean **sorted**
- Heapifying does **not** change the shape of the binary tree
    - This binary tree is *balanced* and *left-justified* because it started out that way

Removing the Root

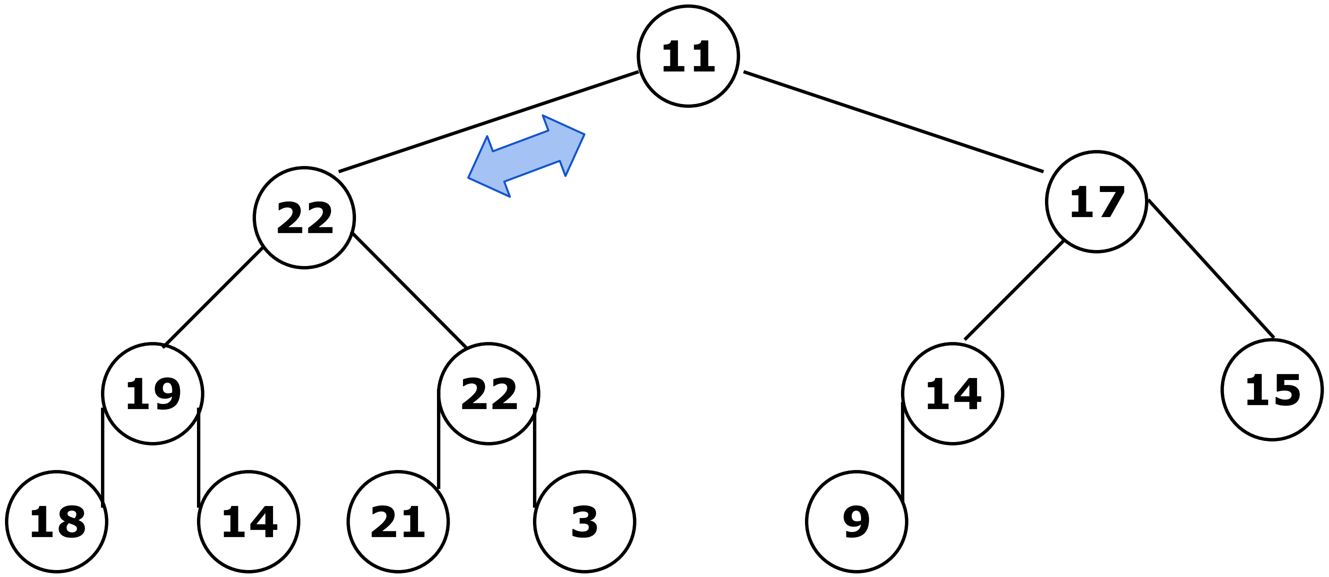
- Suppose we *discard* the root:



- How can we fix the binary tree so it is once again **balanced** and **left-justified**?
- ***Solution:*** remove the rightmost leaf at the deepest level and use it for the new root
  - This fixes the structure of the tree

The reHeap Method I

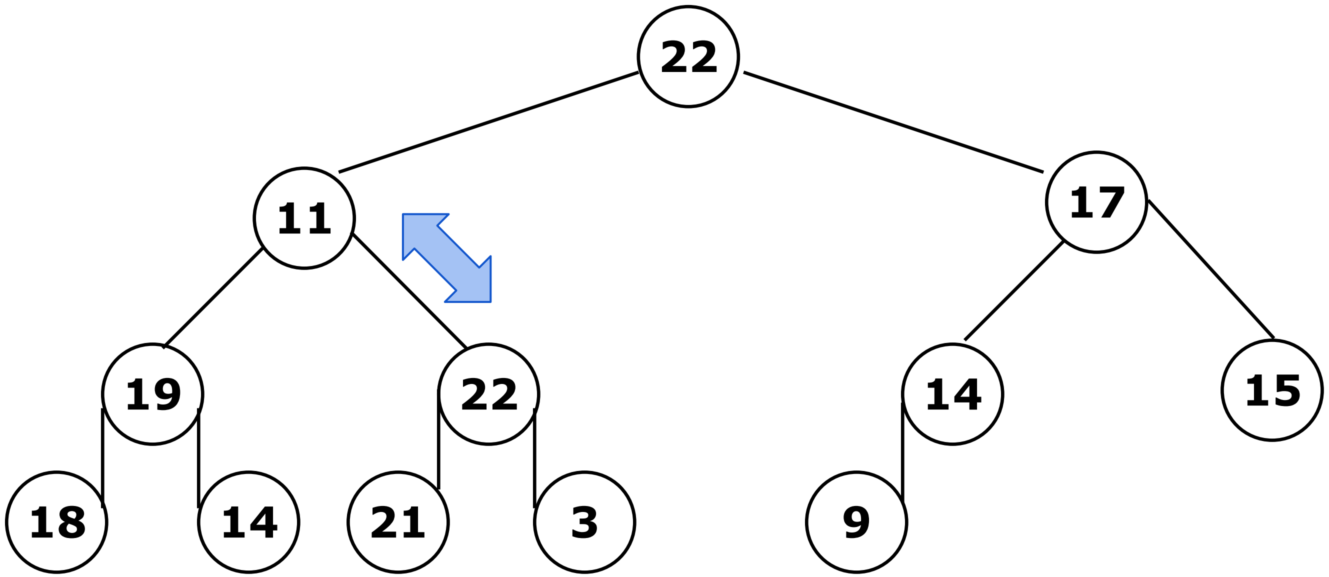
- Our tree is balanced and left-justified, but no longer a heap
- However, **only the root** lacks the heap property



- We can siftDown() the root
- After doing this, **at most one** of its children may have lost the heap property

The reHeap Method II

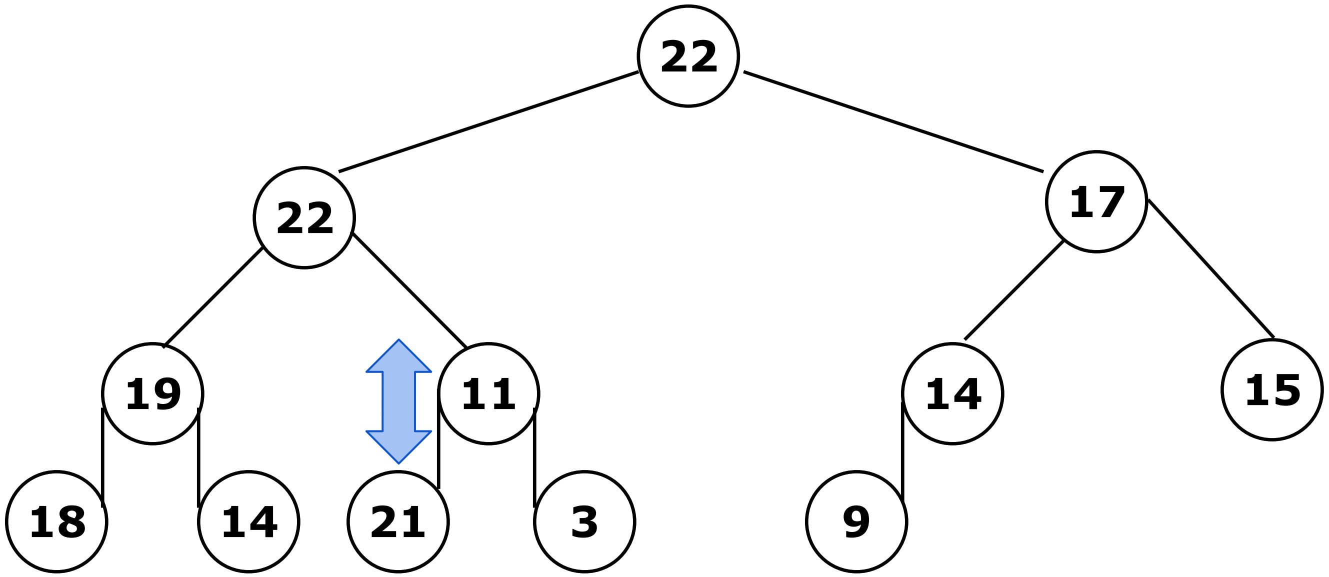
- Now the left child of the root (still the number 11) lacks the heap property



- We can siftDown() this node
- After doing this, **at most one** of its children may have lost the heap property

The reHeap Method III

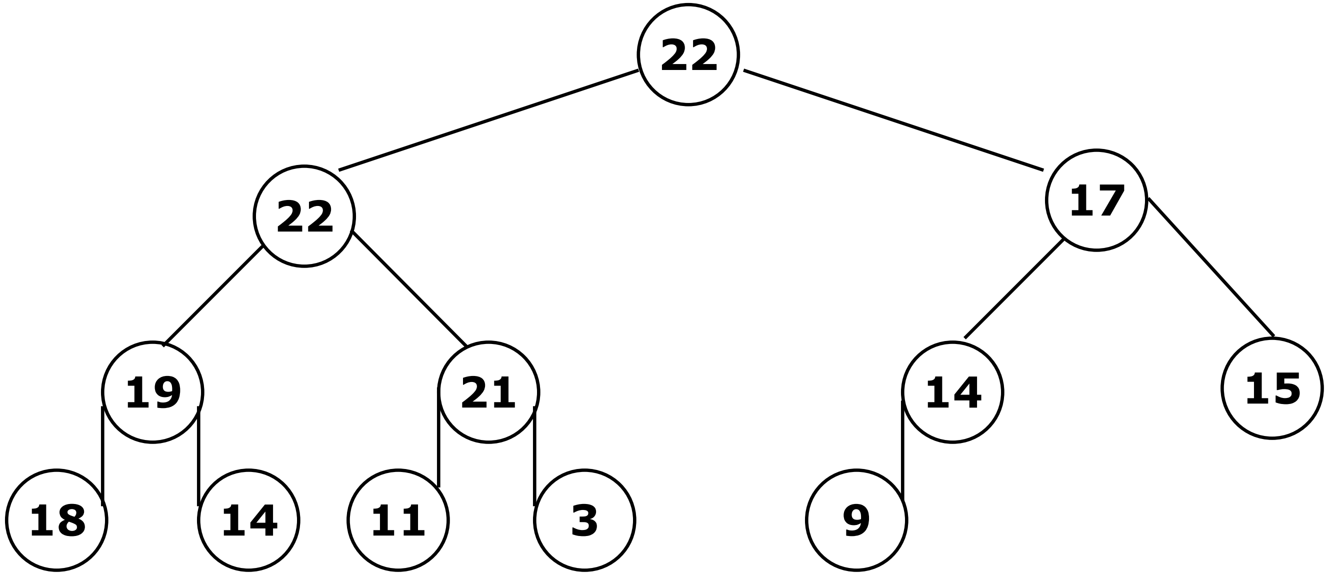
- Now the right child of the left child of the root (still the number 11) lacks the heap property:



- We can siftDown() this node
- After doing this, **at most one** of its children may have lost the heap property — *but it doesn't, because it's a leaf*

The reHeap Method IV

- Our tree is *once again a heap*, because every node in it has the heap property

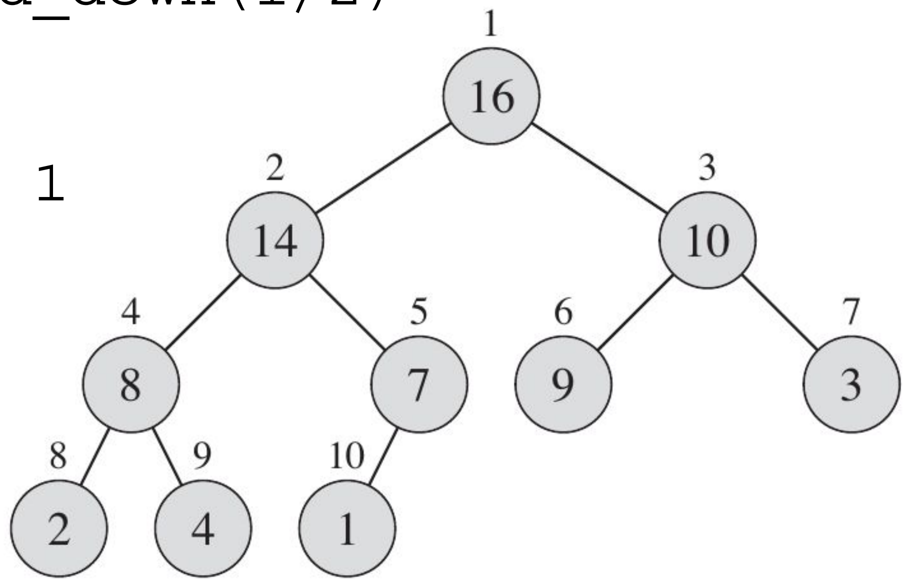


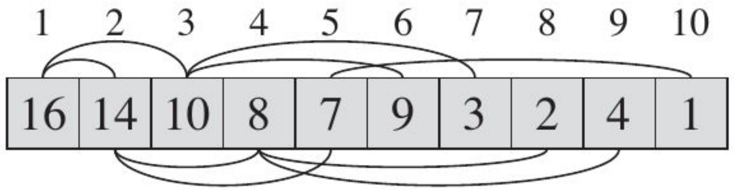
- Once again, the largest (or **a** largest) value is in the root
- We can repeat the **root removal process** until the tree becomes empty
  - This produces a **sequence of values** from largest to smallest

Sorting with Heaps

- What do heaps have to do with sorting an array?
  - Because the binary tree is *balanced* and *left justified,* it can be represented as an **array**
    - The first element is the root, the rest are stored in the level order
    - *Caution:* this works ***only*** with ***balanced, left-justified*** binary trees
  - All our operations on binary trees can be represented as operations on *arrays*

```
parent(i) := round_down(i/2)
left(i) := 2*i
right(i) := 2*i + 1
```

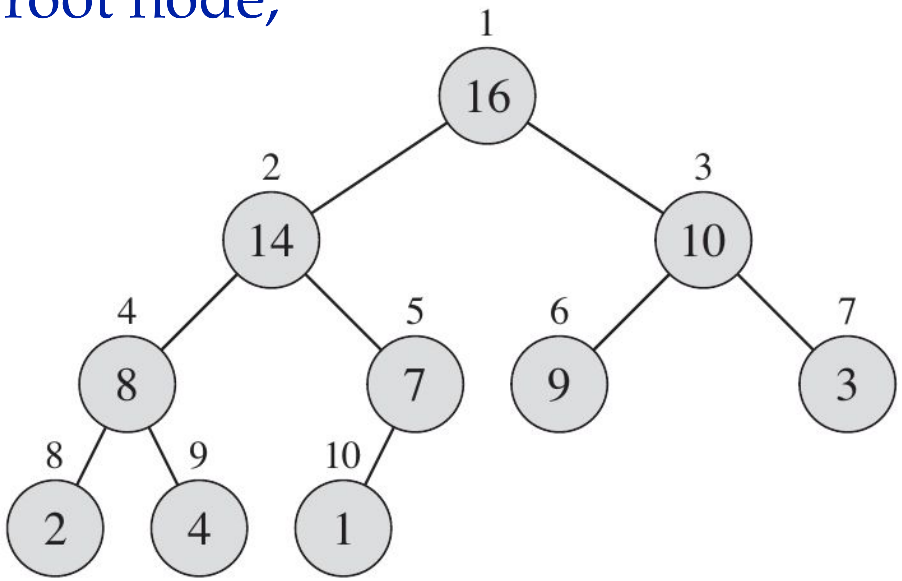


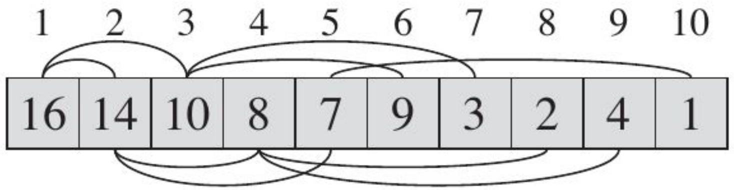


Heapsort

- To sort:

```
heapify the array;
while the array isn't empty {
    swap the root with the last element;
    decrement array size;
    reheap the new root node;
}
```





Key Properties of Heapsort

- Determining location of root and "last node" takes constant time
- Sorts the array in-place (like **insertion sort** and **quicksort**)
- *Overall flow:* heapify; then remove n elements, re-heap each time

Is heapsort stable?

Complexity Analysis

- To **heapify an array**, we can insert array elements one-by-one
  - Adding an element takes $O(1)$
  - **siftUp** takes $O(\log n)$ operations by traversing the balanced tree leaf-to-root
  - Repeated n times, it gives us $O(n \log n)$ overall
- There are more efficient algorithms of the same complexity class (see readings)

Complexity Analysis

- To **reheap the root node**, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
    - The binary tree is perfectly balanced
    - Therefore, this path is O(log n) long
        - And we only do O(1) operations at each node
        - Therefore, reheaping takes O(log n) times
- Since we reheap inside a while loop that we do n times, the total time for the while loop is n*O(log n), or O(n log n)

Complexity Analysis

- Construct the heap $O(n \log n)$
- Remove and re-heap $O(\log n)$
  - Repeat n times $O(n \log n)$
- Total time $O(n \log n) + O(n \log n) = O(n \log n)$

Priority Queue

- A priority queue supports removal of elements in priority order. Implement the following operations of a priority queue q:
  - `enqueue (q, e) // add element with priority e`
  - `int removeMax (q) // remove the item with highest priority`

How can we implement a priority queue with a heap?

Priority Queue Implementation

```
template <typename T>
class PriorityQueue {
  private:
     int elementCount;
     heap<T> h;
  public:
     PriorityQueue (); // Create an empty queue
     void enqueue(T e);
     T removeMax();
}
```

Enqueue

- Insert `e` in `h` at position `elementCount`
- Increment `elementCount`
- Heapify `h` with `siftUp`

RemoveMax

- Remove root
- Swap with last node
- Re-heapify with `reHeap`