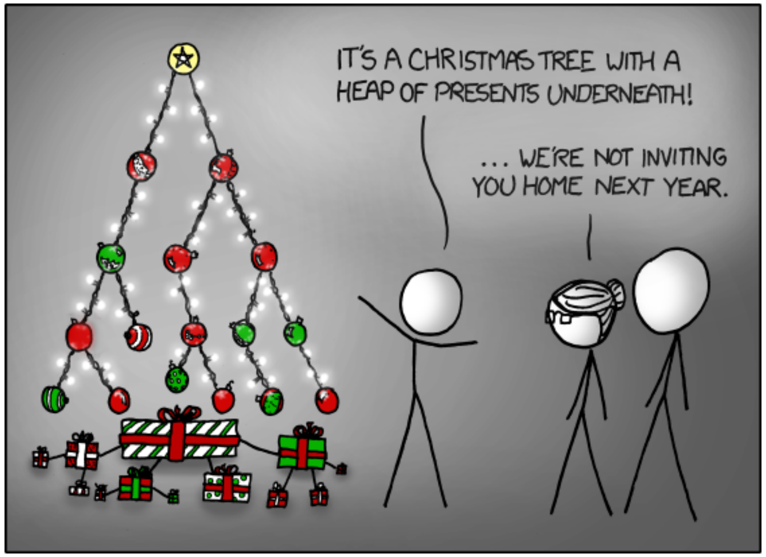
# Heap sort

***Carlos Moreno***
**cmoreno@uwaterloo.ca**
EIT-4103


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.

http://xkcd.com/835/

**https://ece.uwaterloo.ca/~cmoreno/ece250**

These slides, the course material, and course web site are based on work by Douglas W. Harder

# Heap sort

Standard reminder to set phones to silent/vibrate mode, please!

## Heap sort

- Last time, on ECE-250...
  - Talked about heaps as a structure suitable to implement priority queues.
  - Discussed the detail that dequeuing can be seen as extracting the values in order.
  - Noticed one obstacle that we'd face if we try to use such structure for the purpose of sorting.

## Heap sort

- Even before that...
  - We had seen that $\log(n!) = \Theta(n \log n)$
    - $\log(n!) = \log n + \log(n-1) + \log(n-2) + \cdots + \log 3 + \log 2$
  - Also, we had seen that $\sum_{k=1}^{n} k\,2^k = 2\left(n\,2^n - 2^n + 1\right)$

## Heap sort

- Even before that...
  - We had seen that $\log(n!) = \Theta(n \log n)$
    - $\log(n!) = \log n + \log(n-1) + \log(n-2) + \cdots + \log 3 + \log 2$
  - Also, we had seen that $\sum_{k=1}^{n} k\,2^k = 2\left(n\,2^n - 2^n + 1\right)$

    ... say that we leave it at $\Theta(n\,2^n)$

## Heap sort

- During today's lesson:
  - Discuss the notion of max-heap (vs. min-heap, which is what we saw last time)
  - Introduce the Heap sort algorithm
  - Discuss its run time
  - Discuss heapification and its run time

## Heap sort

- During today's lesson:
  - Discuss the notion of max-heap (vs. min-heap, which is what we saw last time)
  - Introduce the Heap sort algorithm
  - Discuss its run time
  - Discuss heapification and its run time ( ... *heapification* ... How cool is that !!)

## Heap sort

- Starting with a preliminary ...
  - Max-heaps vs. min-heaps:
    - The constraints in the nodes in a binary tree that define it to be a heap is based on the parent node being *less than* either one of the children.
    - What if we defined it to be that the parent node has to be *greater than* either one of the children?
      - Clearly, the structure, the kinds of tricks that we can do and that are guaranteed to work given the constraints, would continue to work (as long as we adapt them to the constraint "backwards")

## Heap sort

- Starting with a preliminary ...
  - This is basically the notion of a max-heap — the relationship between parent and children defines the type of heap:
    - Parent < either one of the children: Min-heap
    - Parent > either one of the children: Max-heap
  - *Everything* that we discussed last time works exactly the same with either max-heaps or min-heaps (insertions, removals, percolations, etc.).

## Heap sort

- Moving on to the heap sort ...

## Heap sort

- Basic idea (i.e., what we're after):
  - Heaps allow us to do insertions in worst-case logarithmic time; thus, $n$ insertions would take $\Theta(n \log n)$ — more specifically, it would take $\Theta(\log n!)$, but that's $\Theta(n \log n)$ as we recall.
  - After that, removing each element (always the lowest value) takes logarithmic time each; so, removing them all takes another $\Theta(\log n!)$
    - But that means that we took a sequence of values and output the same values in sorted sequence ... In $\Theta(n \log n)$ — how cool is that !!

## Heap sort

- Basic idea (i.e., what we're after):
  - Oh, wait — we wanted to do it in-place...
  - With the approach described, not only we're using additional storage ( $\Theta(n)$ ), but we need a heap, instead of just an array.

## Heap sort

- Basic idea (i.e., what we're after):
  - Oh, wait — we wanted to do it in-place...
  - With the approach described, not only we're using additional storage ( $\Theta(n)$ ), but we need a heap, instead of just an array.
    - Ok, except that we saw that heaps can be implemented using array storage (maintaining a complete binary tree)

## Heap sort

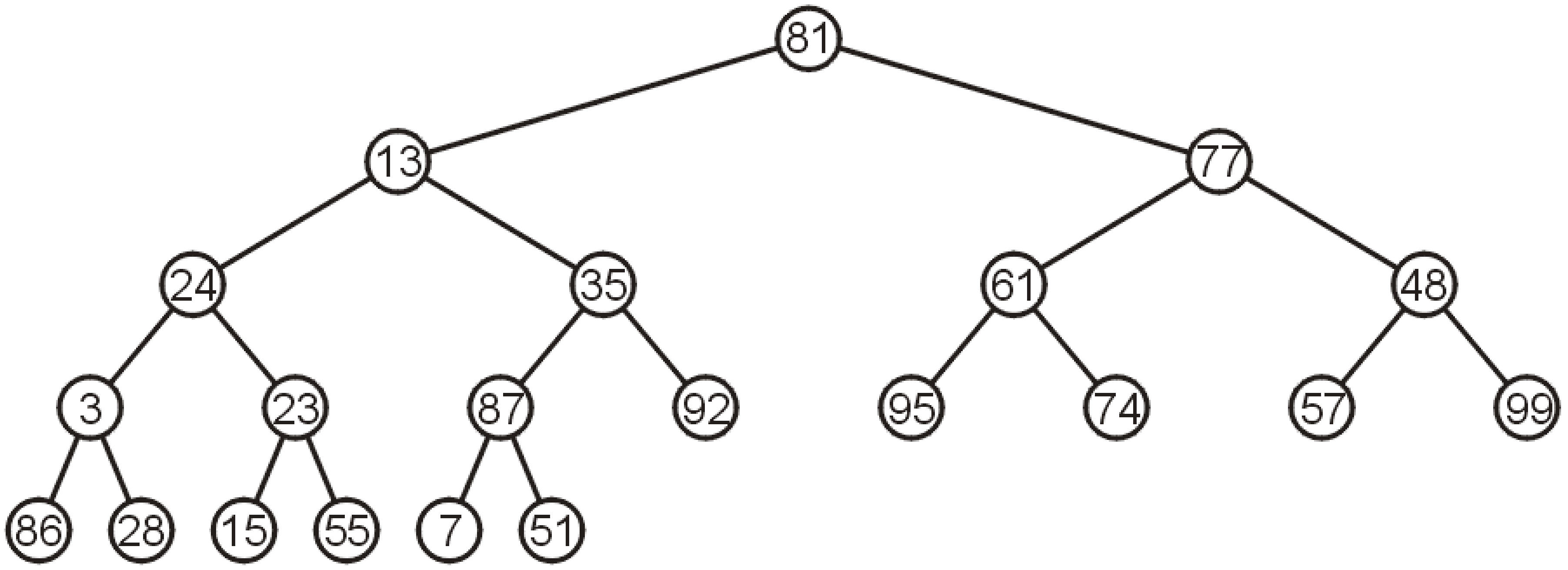
- Basic idea (i.e., what we're after):
  - Oh, wait — we wanted to do it in-place...
  - With the approach described, not only we're using additional storage ( $\Theta(n)$ ), but we need a heap, instead of just an array.
    - Ok, except that we saw that heaps can be implemented using array storage (maintaining a complete binary tree)
  - The other aspect that should make us optimistic is that we saw how most operations with heaps are done by just swapping nodes — this suggests that doing things in-place should be feasible!

## Heap sort

- Since we're starting with an array of arbitrary values, and we want to work in-place (that is, we'll want that array to be the heap itself), it makes sense to solve the problem of turning a binary tree with arbitrary, unconstrained data, into a valid heap.
- Let's look at an example:

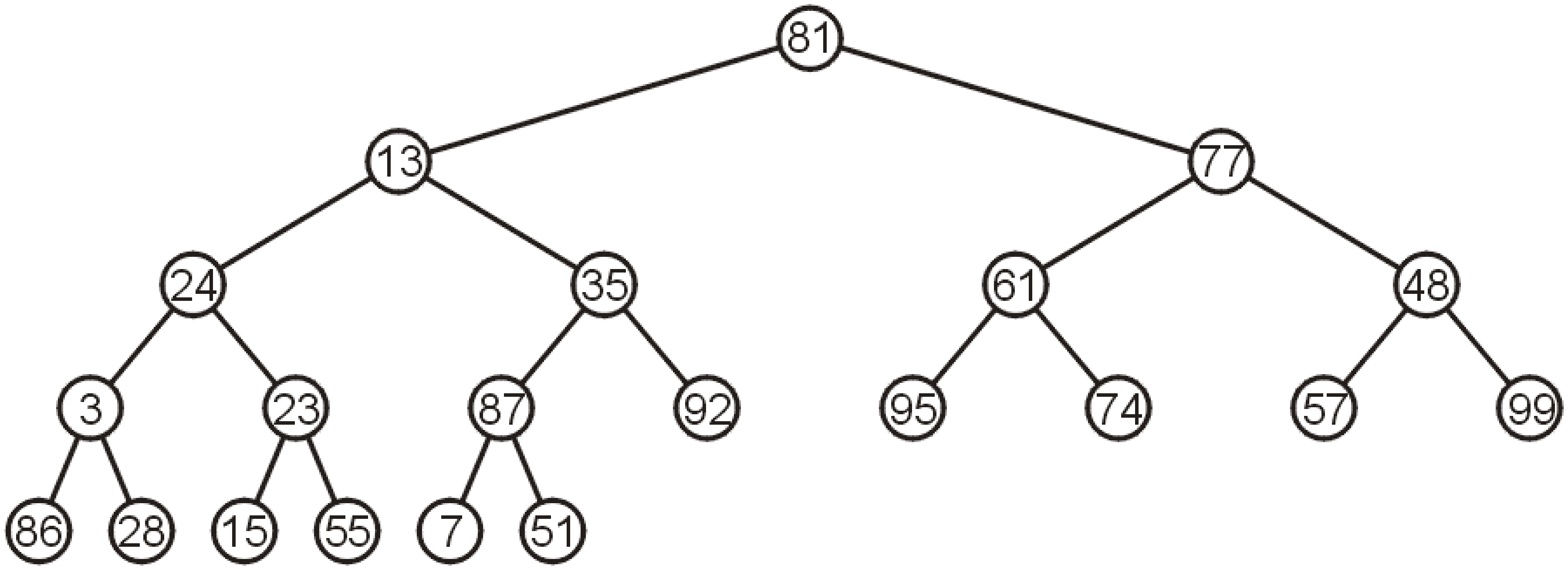
## Heap sort

- This binary tree is not a heap (or a binary search tree, for that matter):



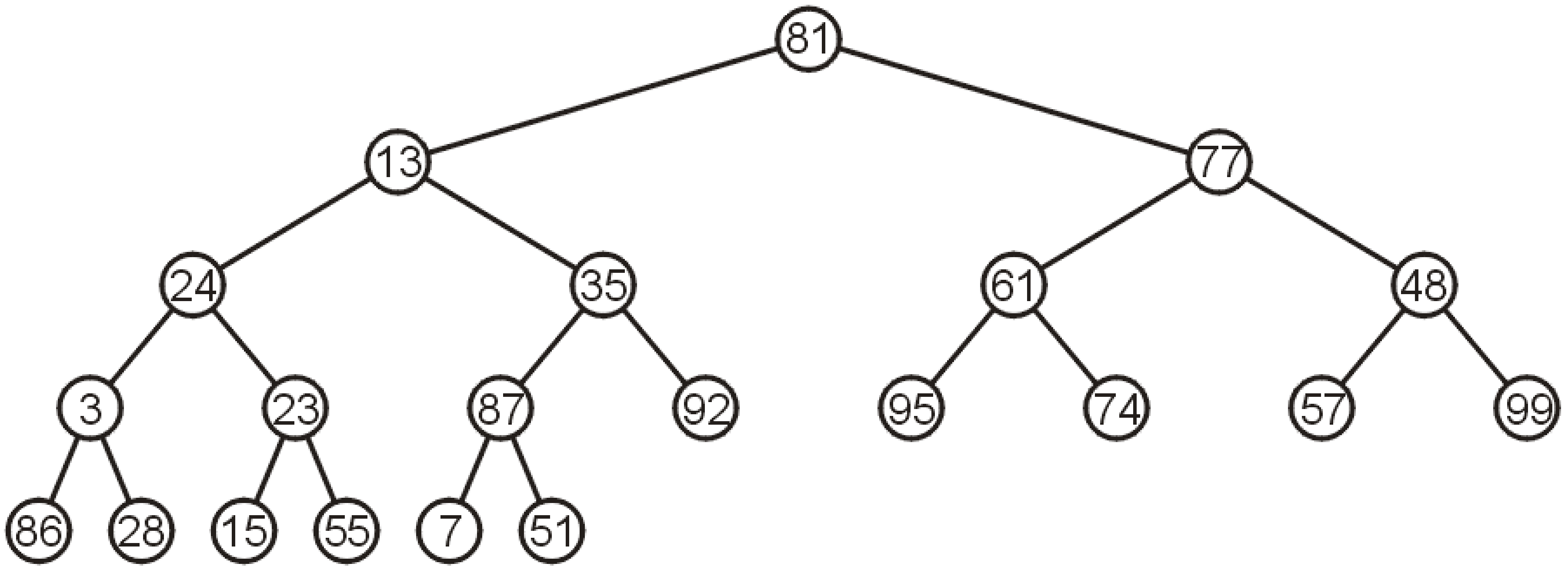
## Heap sort

- However, some of the sub-trees are *guaranteed* to be heaps (right? which ones?)



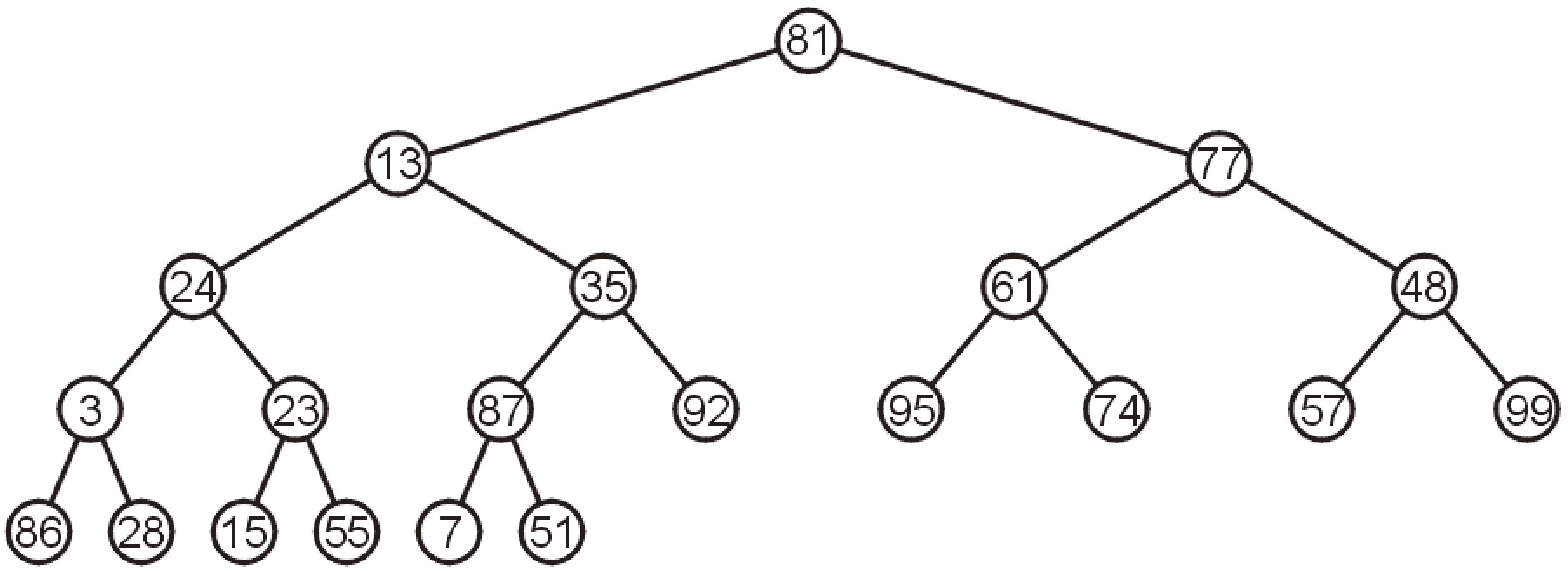
## Heap sort

- However, some of the sub-trees are *guaranteed* to be heaps (right? which ones?)
  - Hint: there's 11 sub-trees guaranteed to be heaps.



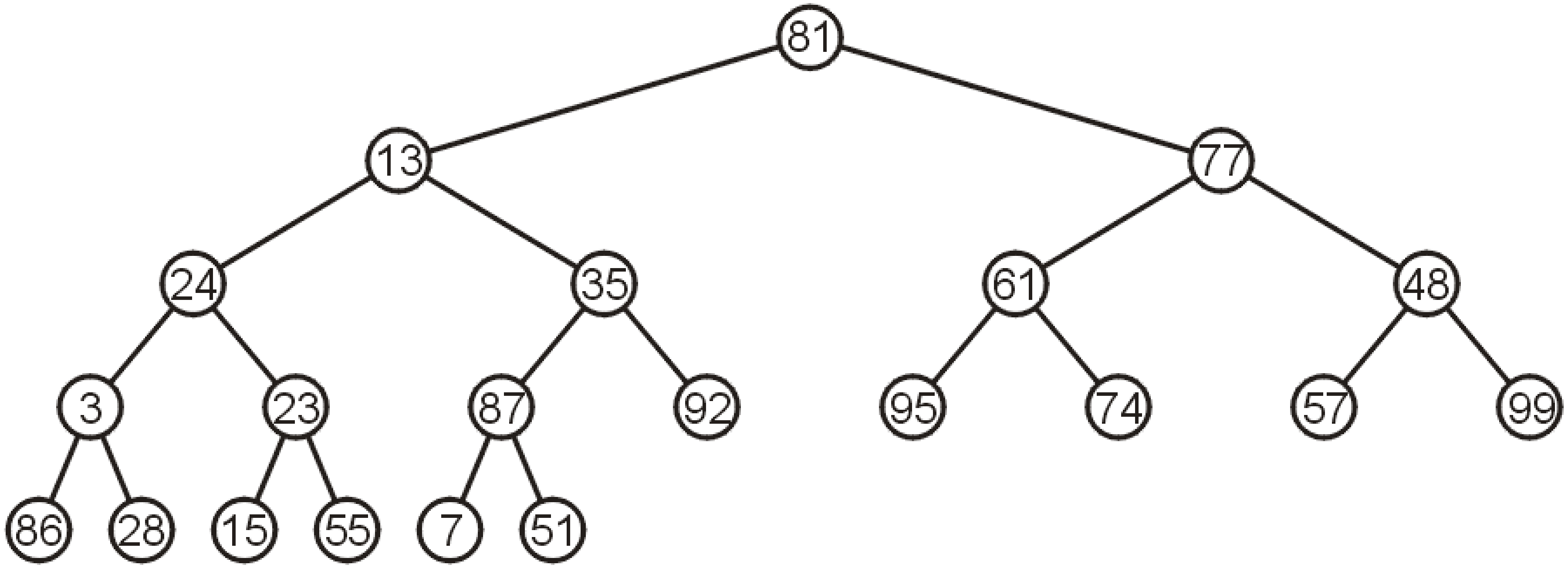
## Heap sort

- The leaf nodes are (by definition) heaps, even if “trivial” heaps.



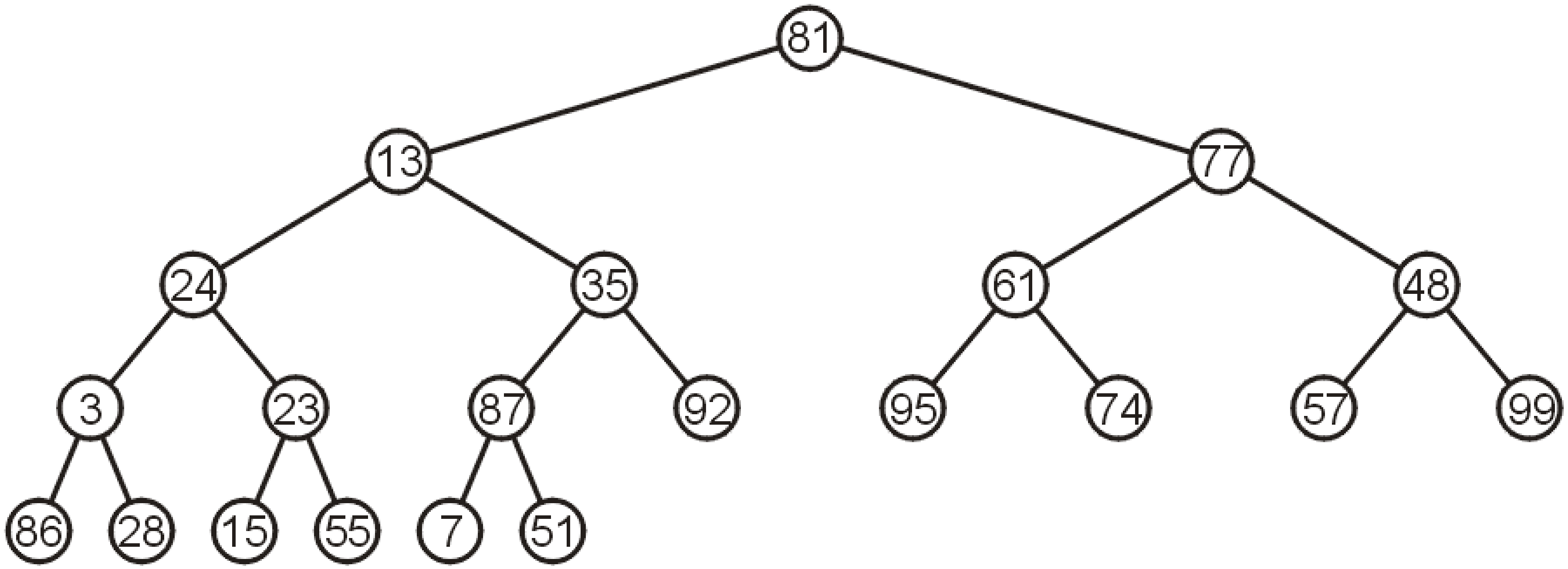
## Heap sort

- The leaf nodes are (by definition) heaps, even if “trivial” heaps.
  - This, perhaps, suggests that we could start adjusting from the leaf nodes working our way up.



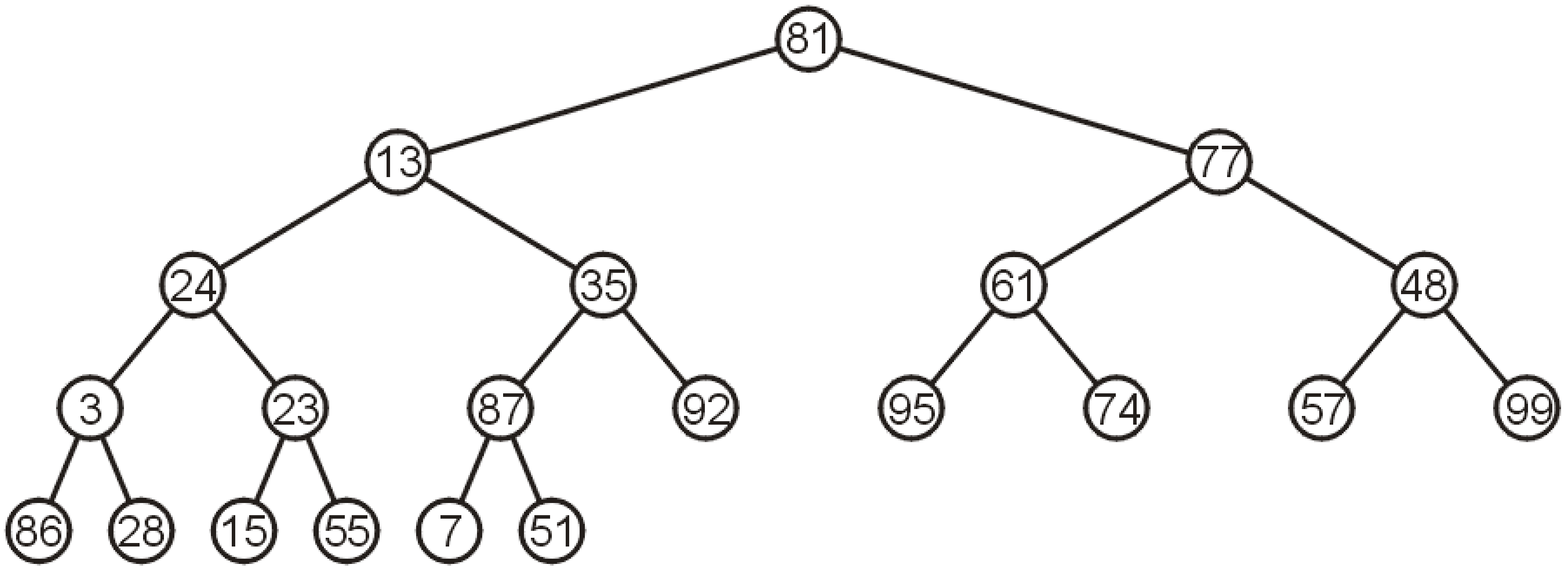
## Heap sort

- In fact, working from the root and working our way down to the leafs would be quite hard! We don't have the heap structure anywhere, and the swaps rely on those constraints!



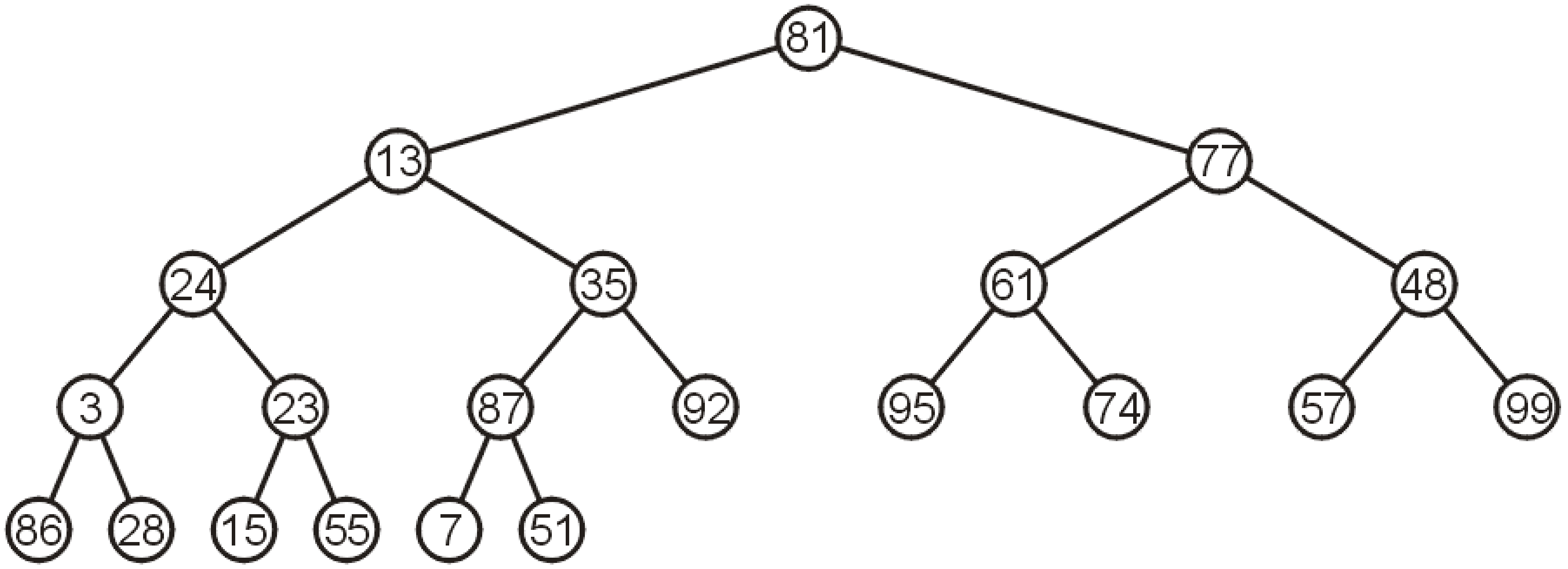
## Heap sort

- If working from the leaf nodes, then when we're done at depth *d*, we know that everything below there is a heap — and that's all we need for percolations from the upper levels to work!



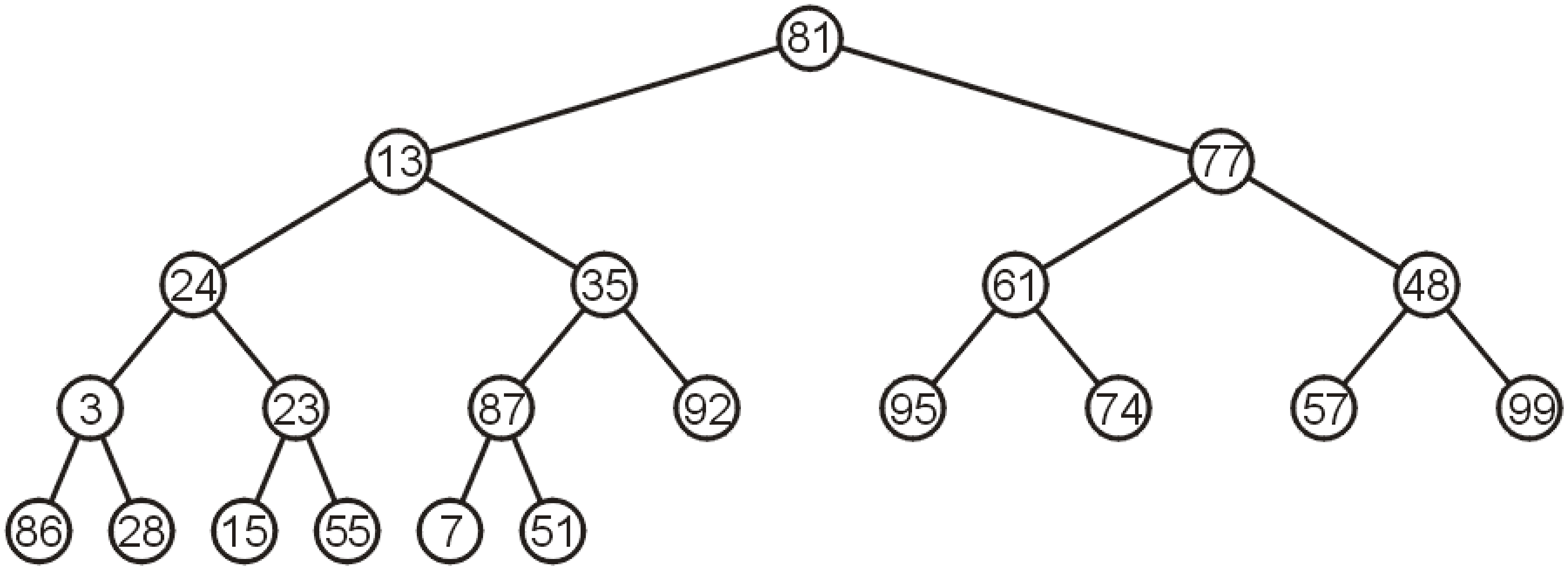
## Heap sort

- Actually, as we'll see, the simpler way is to work by depth, and not by leaf nodes — that is, we start at the deepest level, and move up one level at a time.



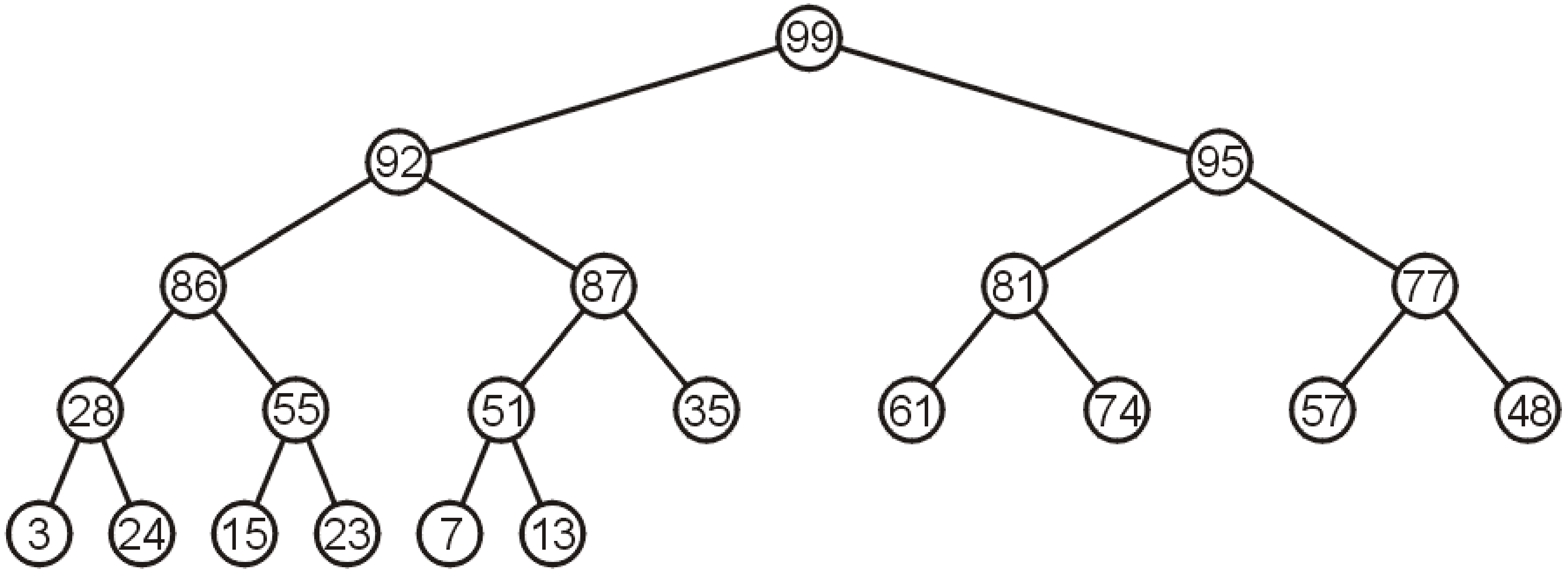
## Heap sort

- We'll look at the procedure in class — with the hints so far, do you want to give it a try?



## Heap sort

- So this is what it will look like once heapified.
- We'll talk about its run time in class.

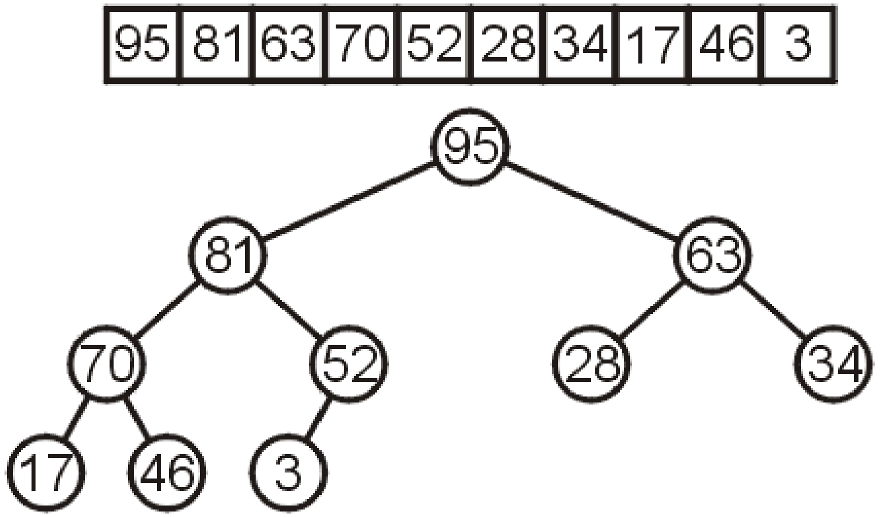


## Heap sort

- What about the rest of the process? Remember that this is stored in an array, and we want to do everything in-place.

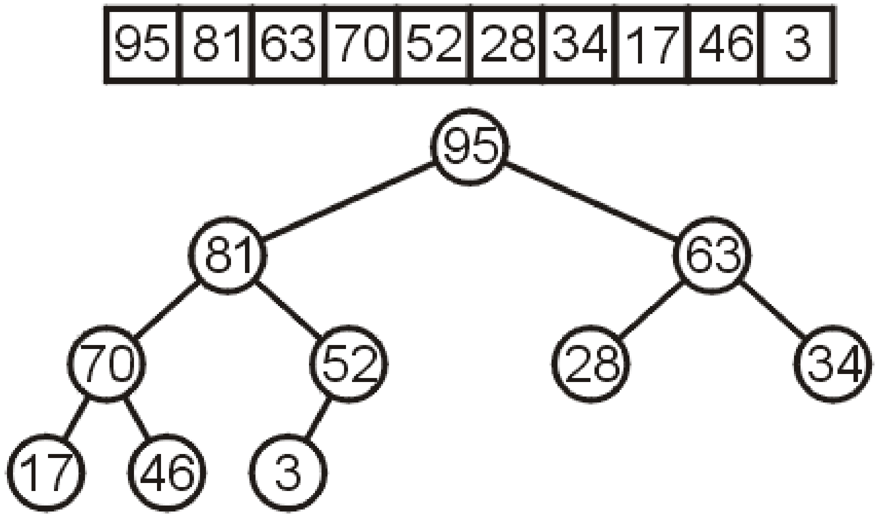
## Heap sort

- Let's work with a “smaller” example — the following array representing the heap shown below:



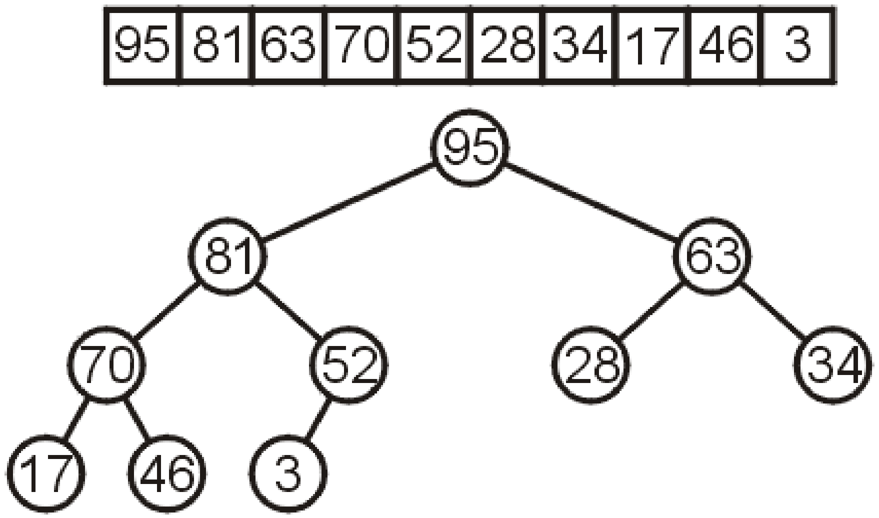
## Heap sort

- 95 is the highest element, and we want it at the end (since we're sorting, so we want an ascending sequence).
- If we remove it, we'd have to move the 3 to the root and adjust....



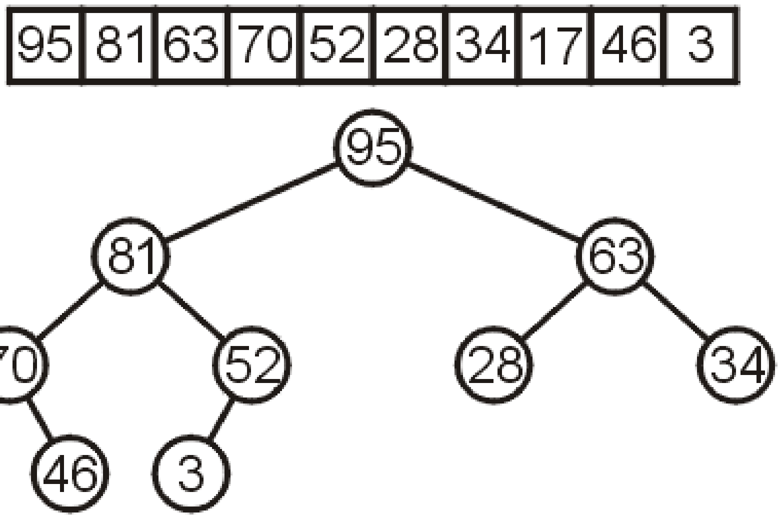
## Heap sort

- Does that perhaps give you any ideas?



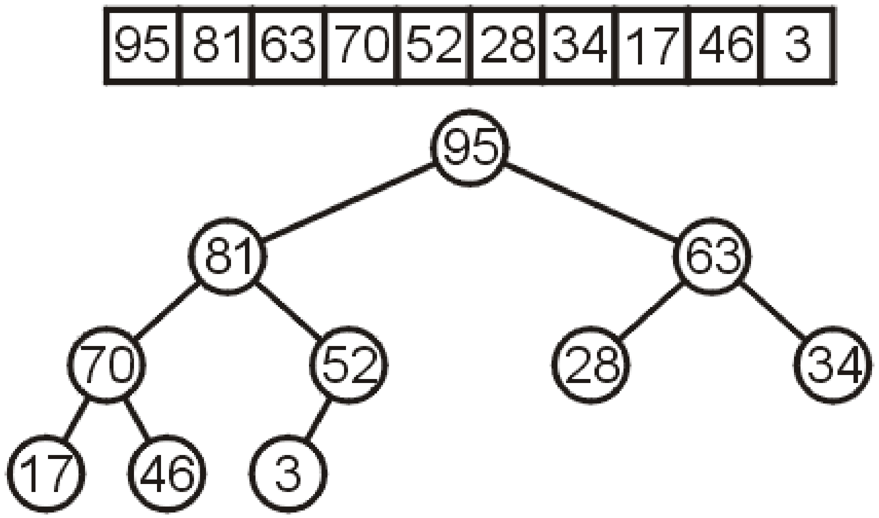
## Heap sort

- Does that perhaps give you any ideas?
- Is it clear at this point why we wanted a max-heap instead of a min-heap?



# Heap sort

- Try it out!!
- We'll complete the discussion in class, and will look at several examples.


95 81 63 70 52 28 34 17 46 3
95
81
63
70
52
28
34
17
46
3