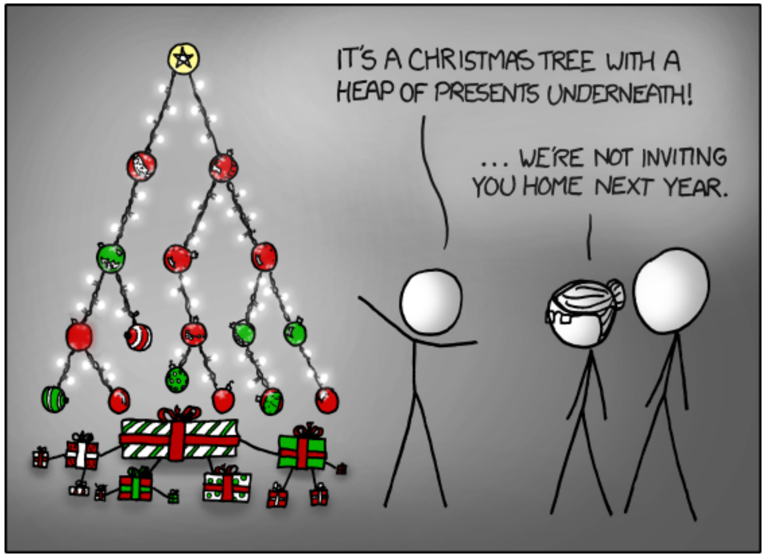
Heap sort


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

http://xkcd.com/835/

Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

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

- 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.

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?

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

Heap sort

- Starting with a preliminary ...
  - *Everything* that we discussed last time works exactly the same with either max-heaps or min-heaps (insertions, removals, percolations, etc.).

Heap sort

- Starting with a preliminary ...
  - *Everything* that we discussed last time works exactly the same with either max-heaps or min-heaps (insertions, removals, percolations, etc.).
  - Of course, matching all conditions — for example, when percolating up, we saw that we need to promote the lower of the two children.

Heap sort

- Starting with a preliminary ...
  - *Everything* that we discussed last time works exactly the same with either max-heaps or min-heaps (insertions, removals, percolations, etc.).
  - Of course, matching all conditions — for example, when percolating up, we saw that we need to promote the lower of the two children.
    - With a max-heap, of course, we promote the larger of the two children
    - Same thing when percolating down — we swap with the larger of the children, instead of with the lower as we saw last time (for min-heaps)

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.

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.

    (BTW... why is the run time $\Theta(\log n!)$??)

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.

    (BTW... why is the run time $\Theta(\log n!)$??)

    If the heap is initially empty, then with each insertion, the size grows by 1 element; adding the run times of the insertions we get:

    $$\log 1 + \log 2 + \log 3 + \cdots + \log(n-1) + \log n = \log(n!)$$

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 ...

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 ... And it all happened in $\Theta(n \log n)$

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 ... And it all happened 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

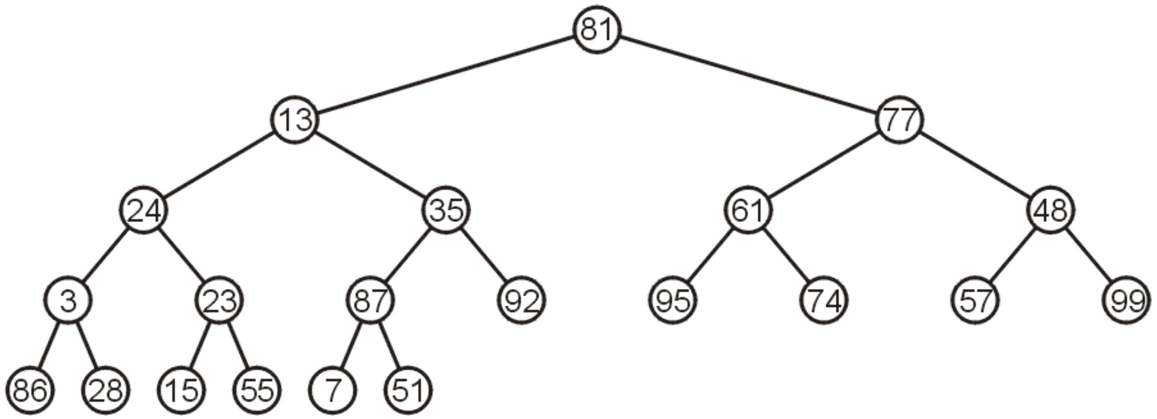
- Basic idea (i.e., what we're after):
  - Oh, wait — we wanted to do it in-place...
  - With the approach described, not only we're using additional storage ($\Theta(n)$), but we need a heap, instead of just an array.
    - Ok, except that we saw that heaps can be implemented using array storage (maintaining a complete binary tree)
  - Another key aspect 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 max-heap.
  - It will become clear why we want to use a max-heap.
- Let's look at an example:

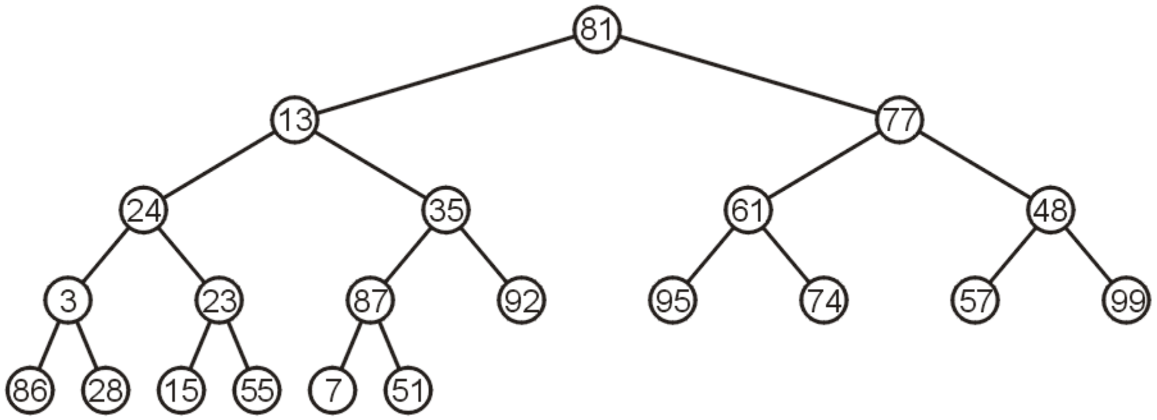
Heap sort

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



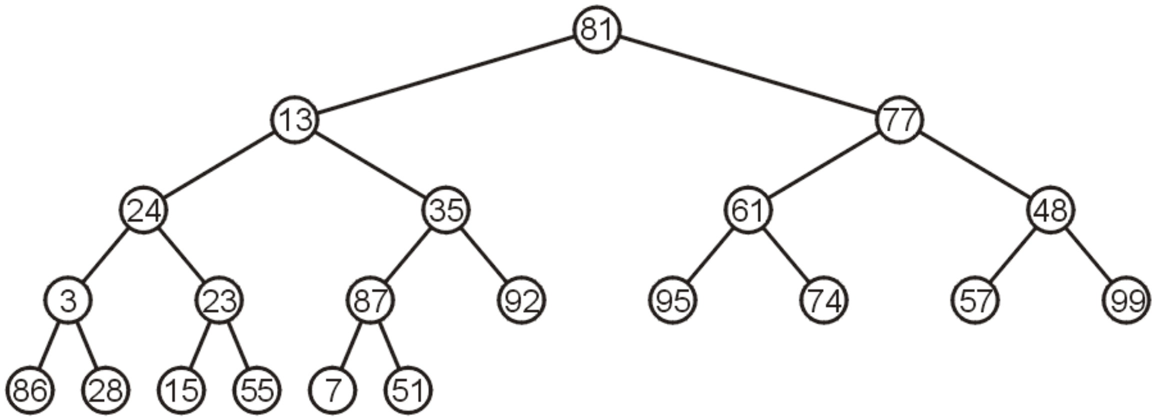
Heap sort

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



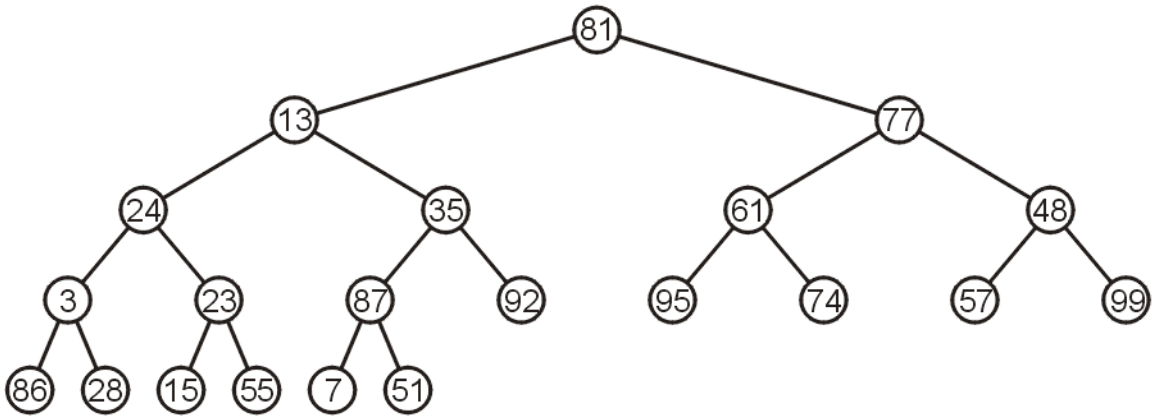
Heap sort

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



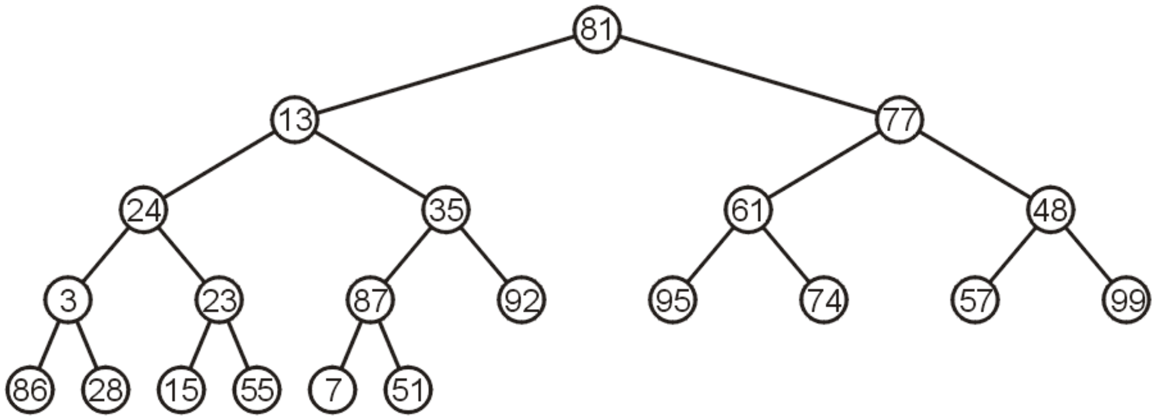
Heap sort

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



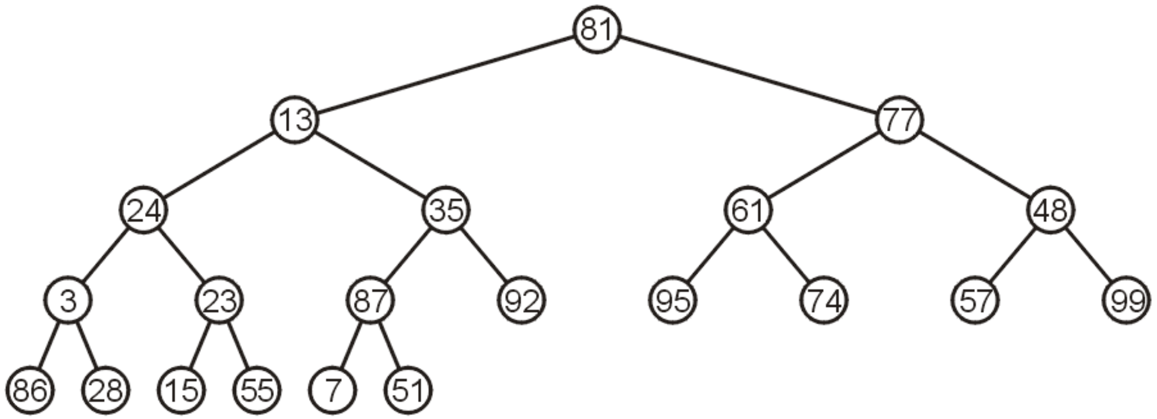
Heap sort

- 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.



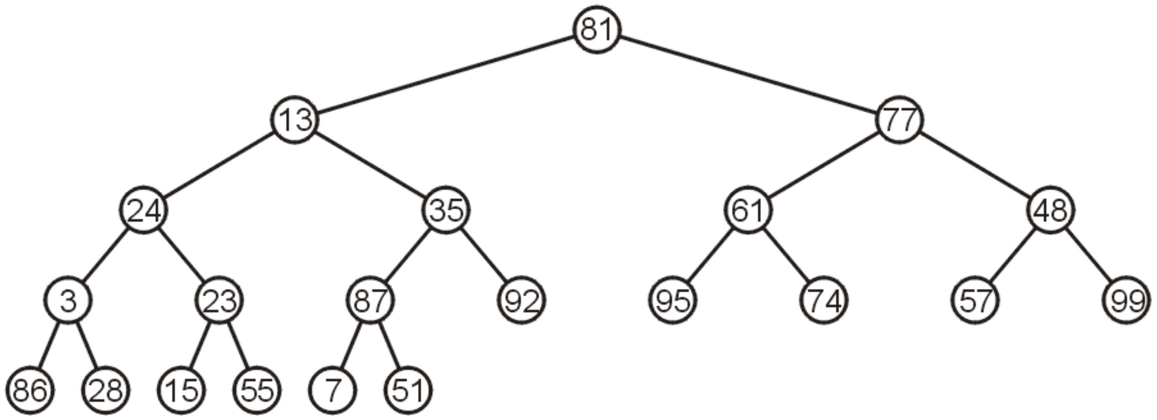
Heap sort

- 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!



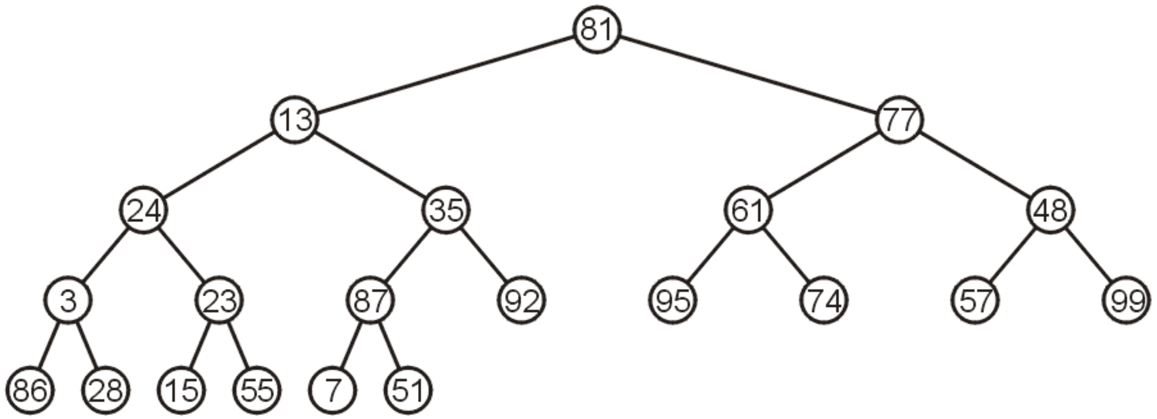
Heap sort

- 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!



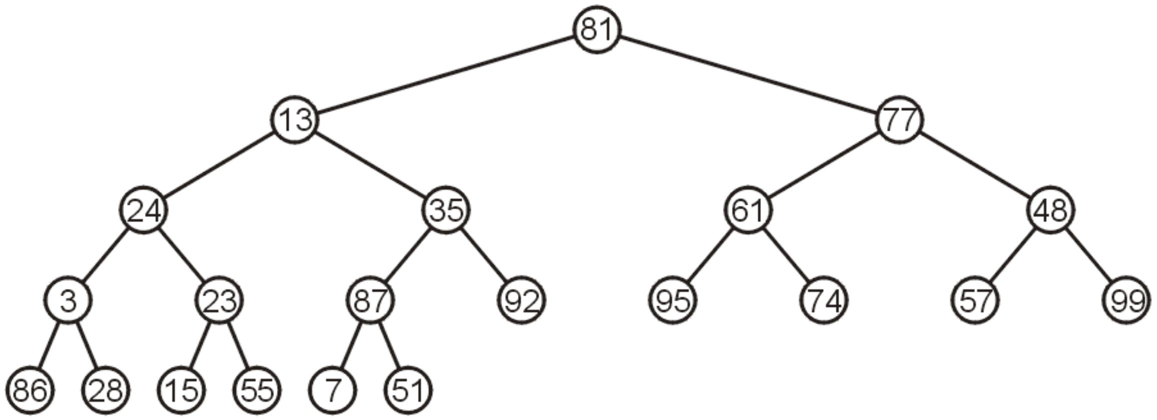
Heap sort

- 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.



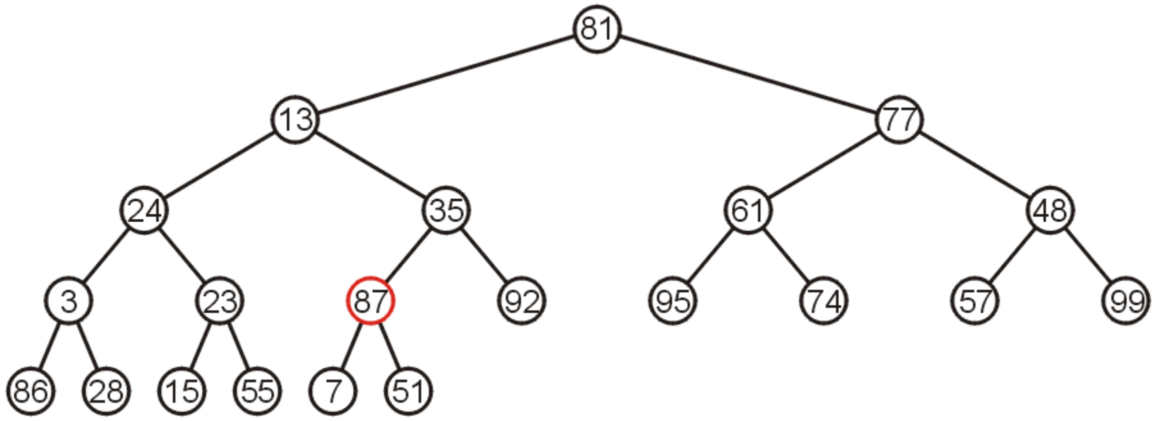
Heap sort

- We start processing the elements from the end, so we start with 51, and go to its parent, 87



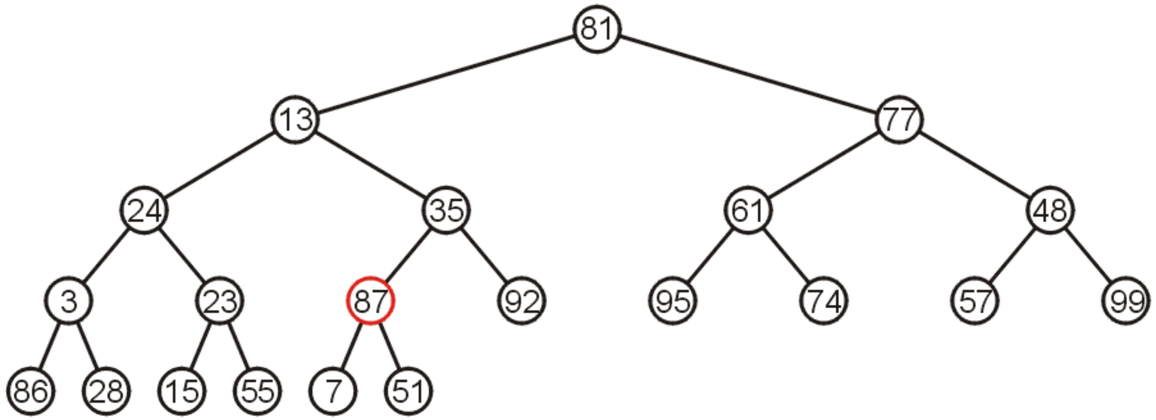
Heap sort

- We start processing the elements from the end, so we start with 51, and go to its parent, 87



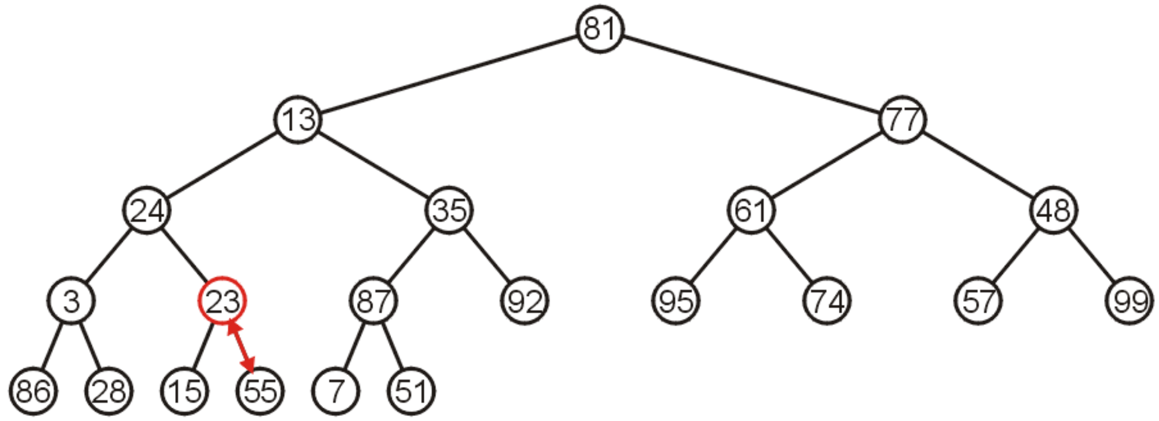
Heap sort

- We start processing the elements from the end, so we start with 51, and go to its parent, 87
  - All good here here — the tree rooted at 87 is already a max-heap; nothing to do there.



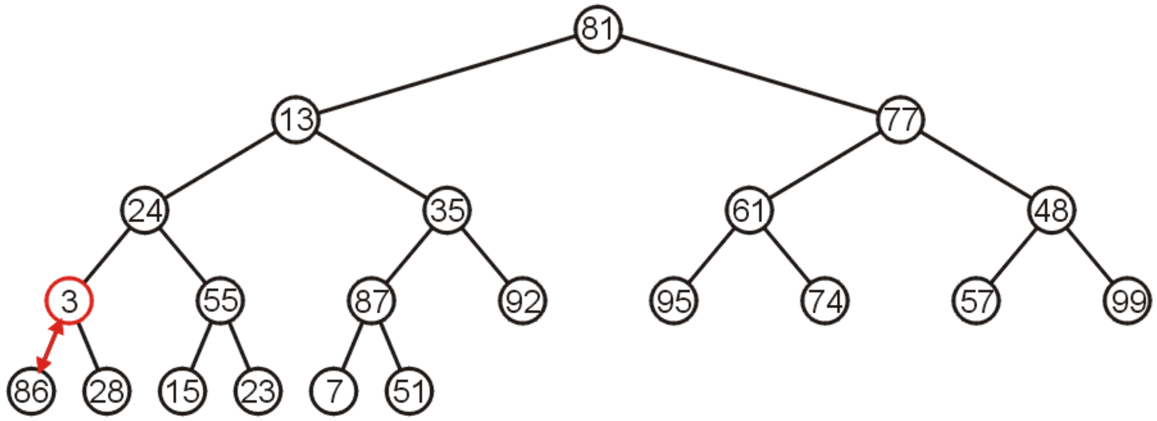
Heap sort

- The sub-tree at 23 is not a max-heap; but since everything below is, then swapping with 55 (the larger of the children) creates a max-heap.



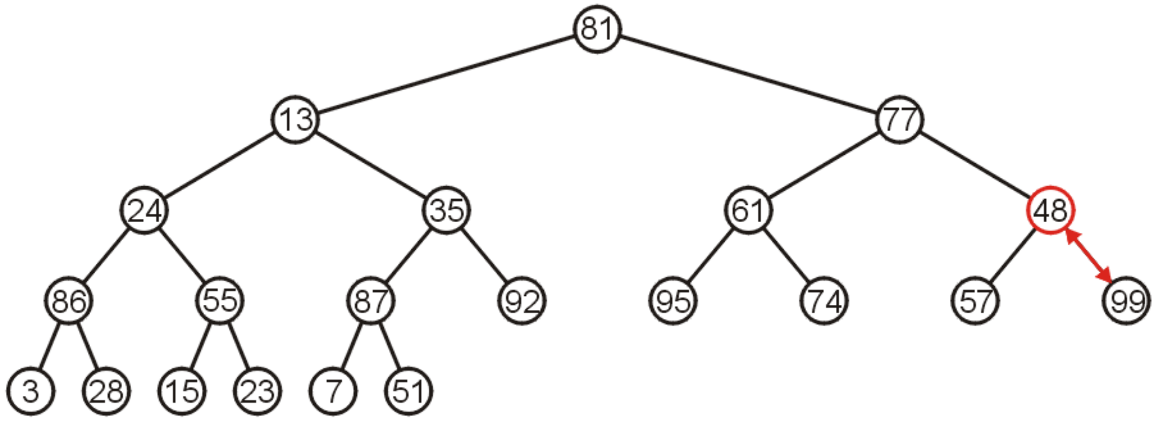
Heap sort

- Same here, swapping with 86 (the larger of the children).
  - At this point (well, after swapping these), we'll have that every sub-tree at depth 3 is a max-heap.



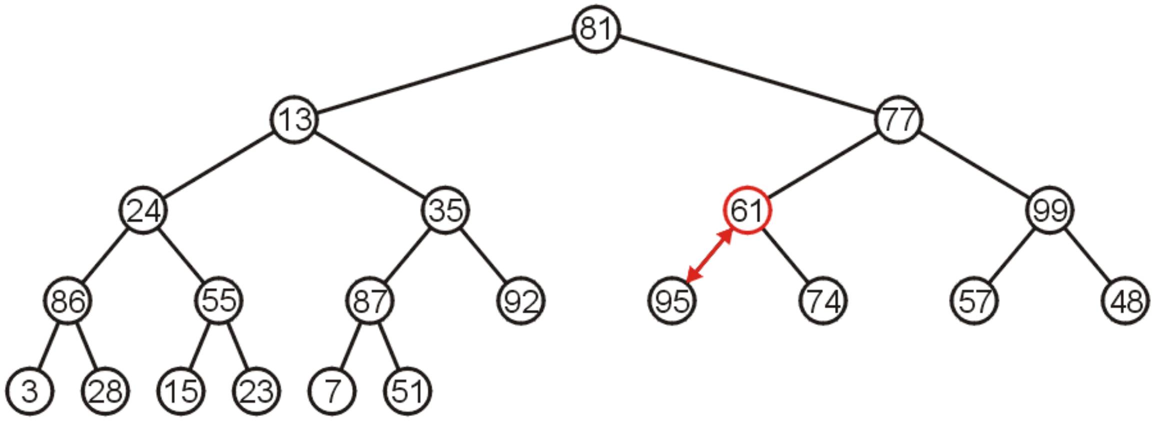
Heap sort

- So, we continue at the next level (up), with the sub-tree rooted at 48 — we swap 48 with 99 (the larger of its children).



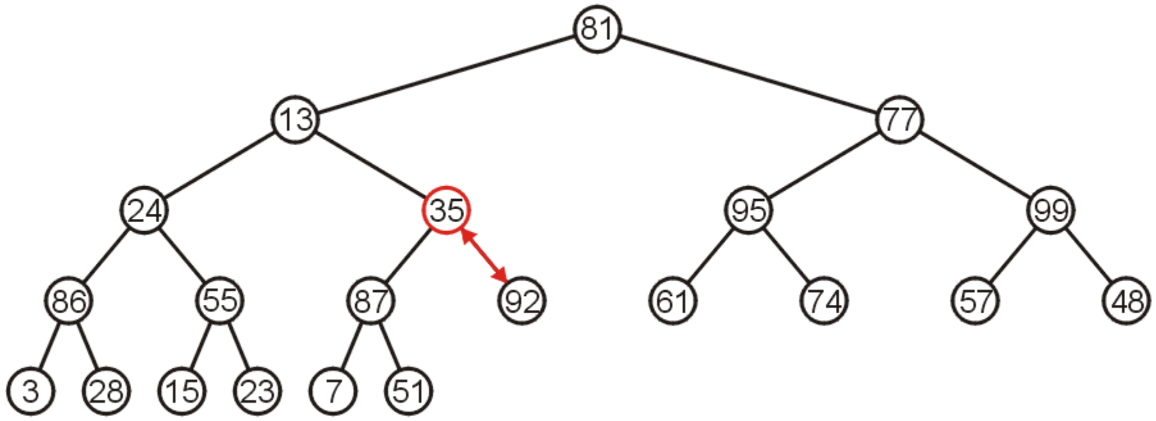
Heap sort

- Then continue ....



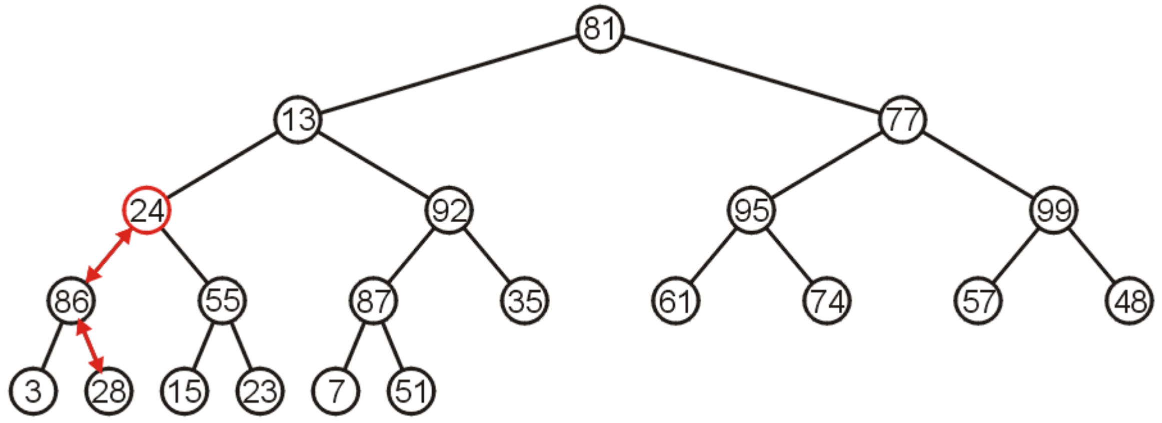
Heap sort

- Then continue ....



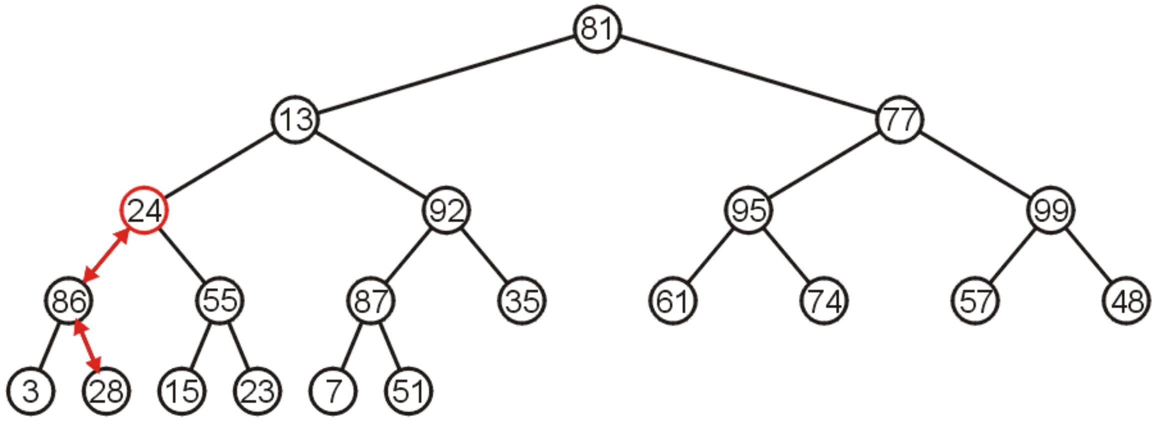
Heap sort

- Then continue ....



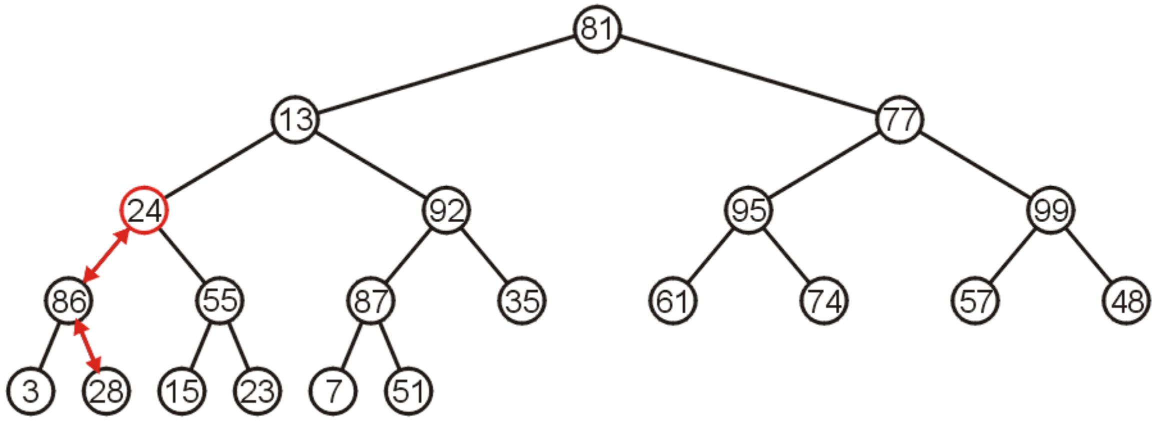
Heap sort

- Then continue .... Oh! surprise! why do we need a double swap in here?



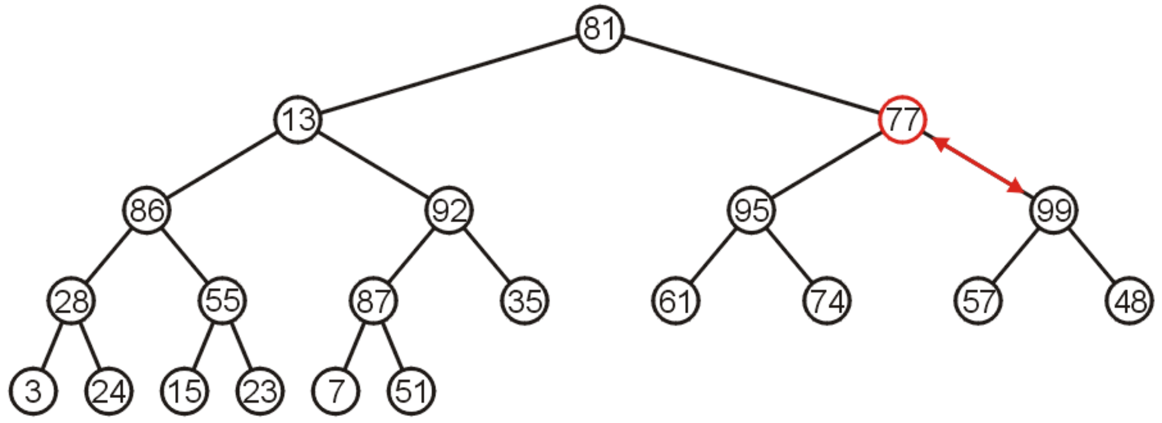
Heap sort

- Then continue .... Oh! surprise! why do we need a double swap in here?
    - Actually, no surprise — we're inserting 24 in the heap below; it is to be expected that we need to percolate it down several steps.



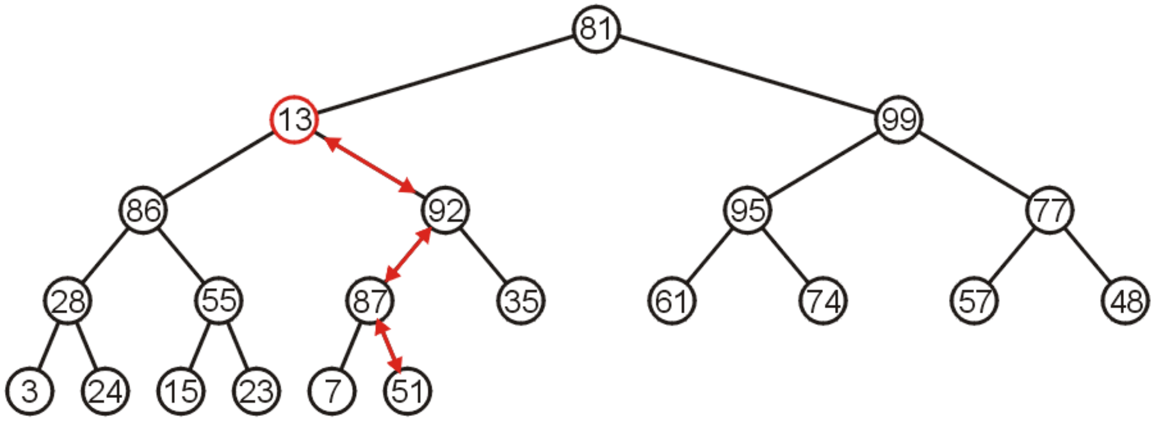
Heap sort

- Now up one level, and continue with 77 ...



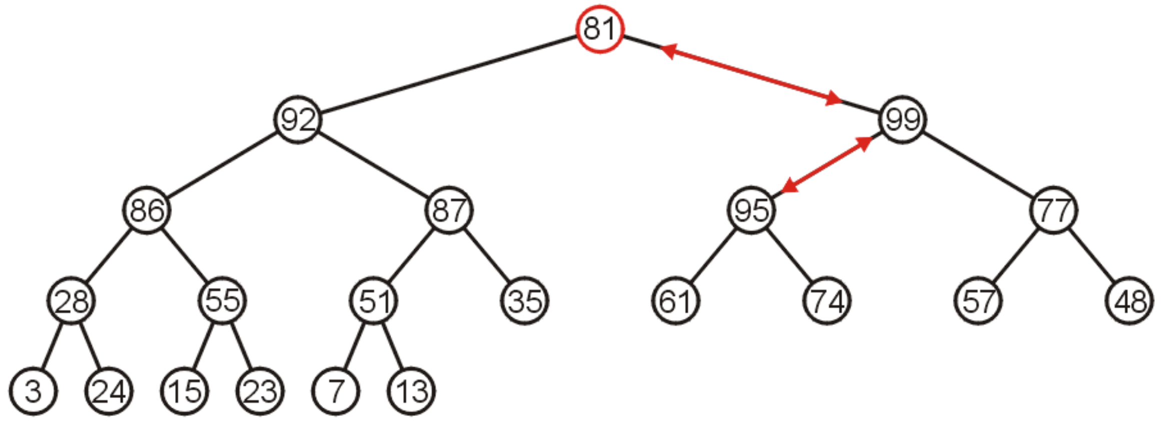
Heap sort

- And continue ...



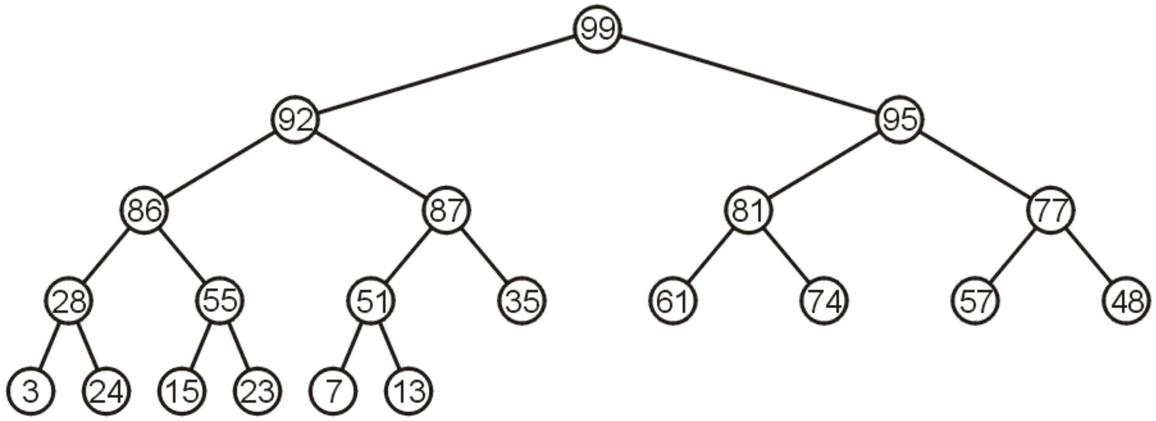
Heap sort

- Then the root, and after percolating it down, we're done ...



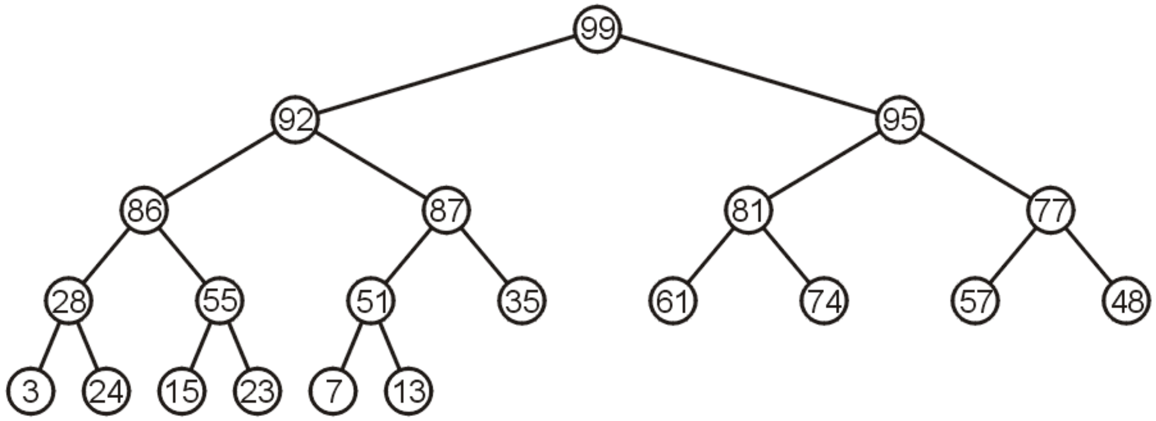
Heap sort

- The end result is, of course, a max-heap:



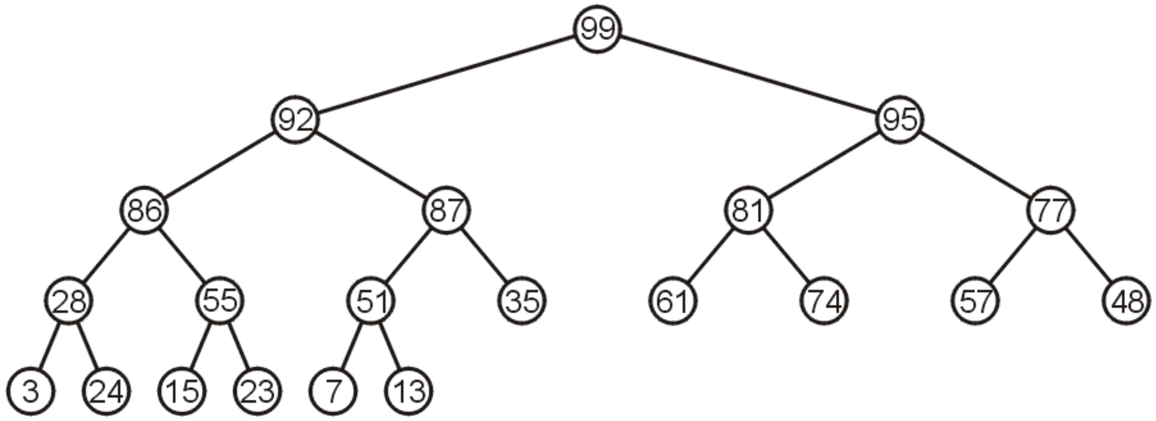
Heap sort

- But the more important detail is ... (anyone?):



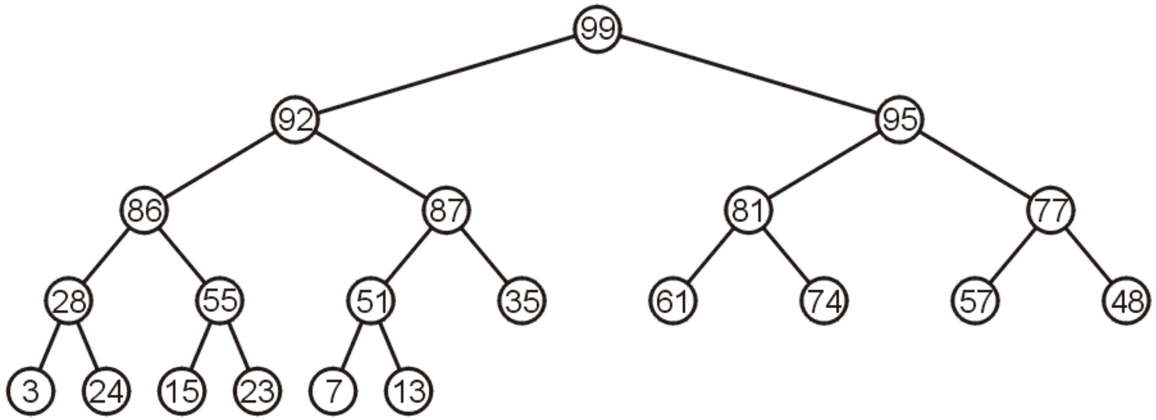
Heap sort

- But the more important detail is ... (anyone?):
  - We call this process *heapification*: how cool is that!!!



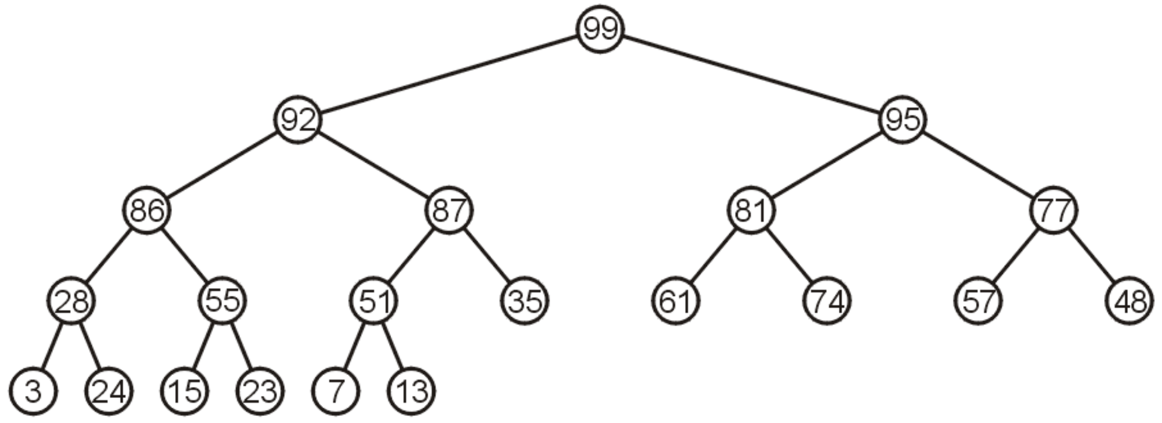
Heap sort

- So, let's see if the run time of this heapification is as cool as its name ...



Heap sort

- Because this is a binary tree, then at depth *k*, we have at most 2^k nodes; each of these could require percolation all the way to the deepest level (in the worst-case)



Heap sort

- Because this is a binary tree, then at depth k, we have at most 2^k nodes; each of these could require percolation all the way to the deepest level (in the worst-case)
- If we are at depth k, then that's $h-k$ swaps required to percolate the element (again, in the worst-case)
- That's a total of $2^k (h-k)$ swaps at depth k.
- The total would be (math on the next slides...)

Heap sort

- Adding for all depths (*k* going from 0 to *h*):

$$\begin{aligned}\sum_{k=0}^{h} 2^k(h-k) &= h\,\frac{2^{h+1}-1}{2-1} - 2\left(h2^h - 2^h + 1\right)\\ &= h2^{h+1} - h - h2^{h+1} + 2^{h+1} - 2\\ &= \left(2^{h+1}-1\right) - (h+1)\\ &= n - \lg(n+1) = \Theta(n)\end{aligned}$$

Heap sort

- So, the thing is called *heapification*, and it runs in linear time....

Heap sort

- So, the thing is called *heapification*, and it runs in linear time....
  - I mean, really, guys: *how cool is that* !!!

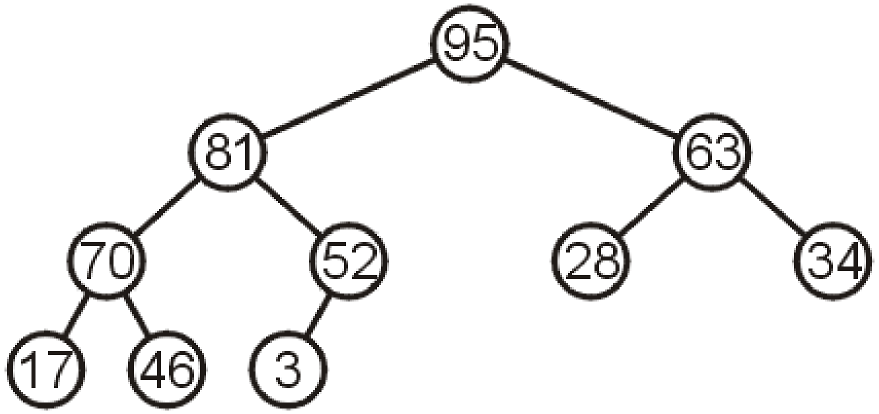
Heap sort

- So, we're done with converting the input array into a max-heap (keep in mind that everything we showed in the previous slides is done in the array itself).
- What about the rest of the process?
  - Keep in mind that we want to do everything in-place.

Heap sort

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

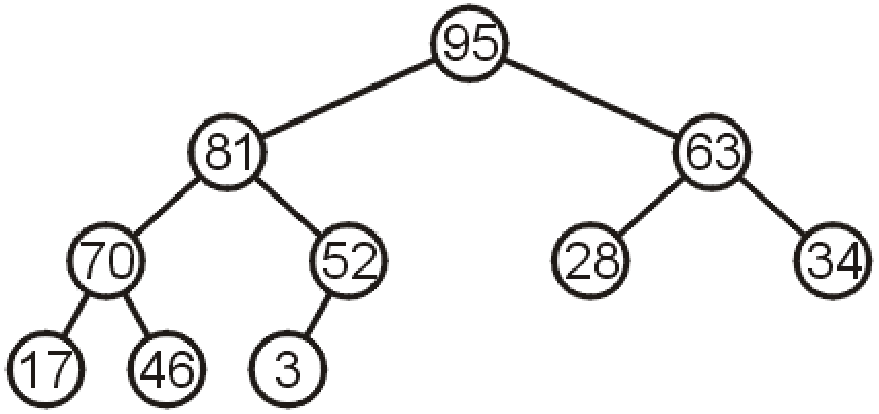
95	81	63	70	52	28	34	17	46	3



Heap sort

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

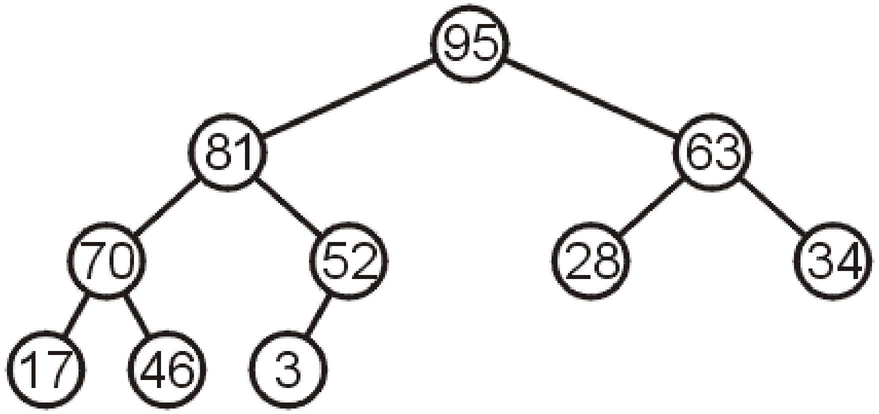
95	81	63	70	52	28	34	17	46	3



Heap sort

- Does that perhaps give you any ideas?

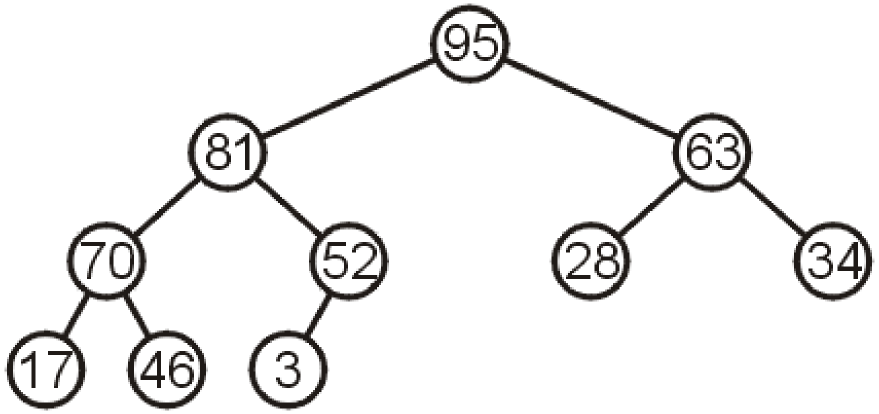
95	81	63	70	52	28	34	17	46	3



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?

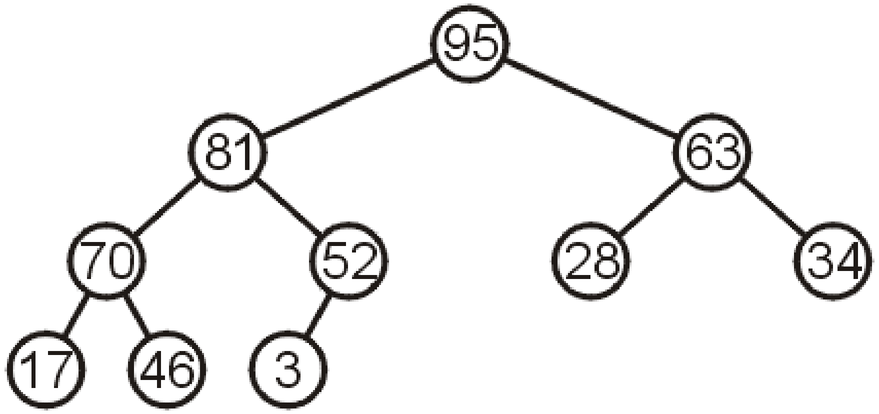
95	81	63	70	52	28	34	17	46	3



Heap sort

- Let's see it working — we dequeue the highest element, which will open up a spot at the end; so, we move that highest element to the end (where it *should* go — recall that we're sorting!)

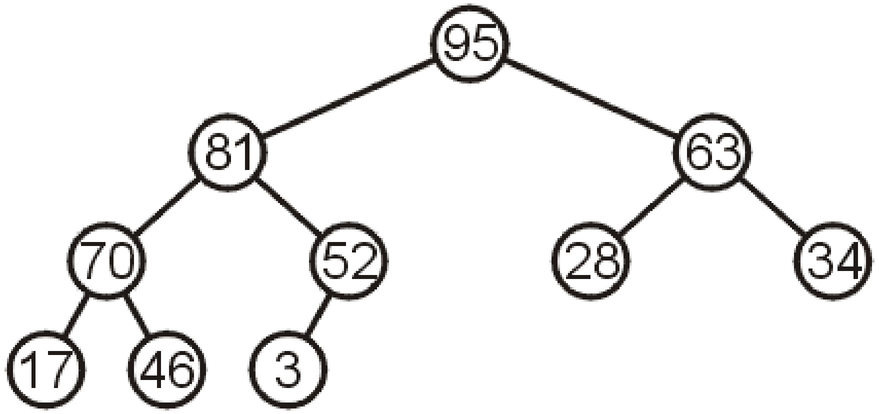
95	81	63	70	52	28	34	17	46	3



Heap sort

- But we also remember that dequeueing is done by moving the last element (so that we maintain a complete binary tree) to the root and percolating down.

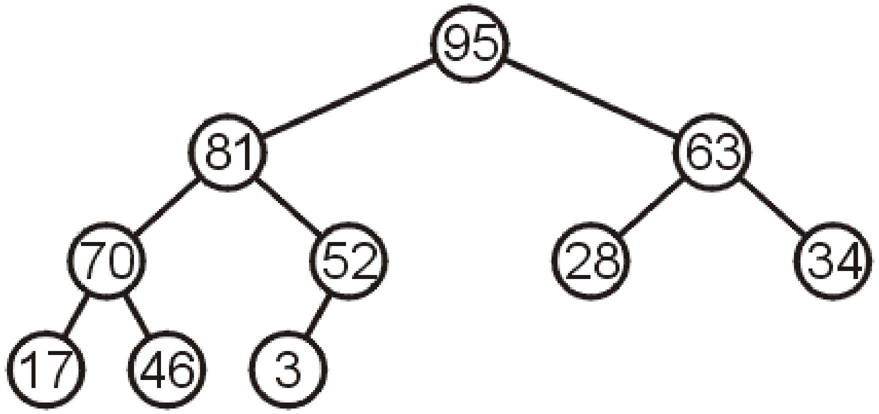
95	81	63	70	52	28	34	17	46	3



Heap sort

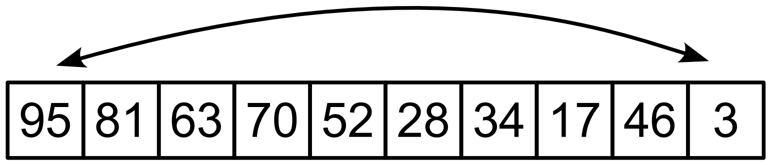
- So, at each step, we just swap the first element with the last one (we need to keep a counter, that is decreased by one at each iteration), then percolate down the one that is now at the root.

95	81	63	70	52	28	34	17	46	3



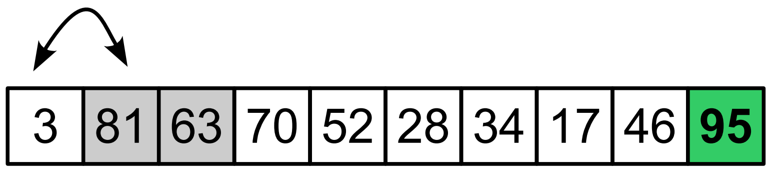
Heap sort

- Swap 95 and 3, then percolate down 3:



Heap sort

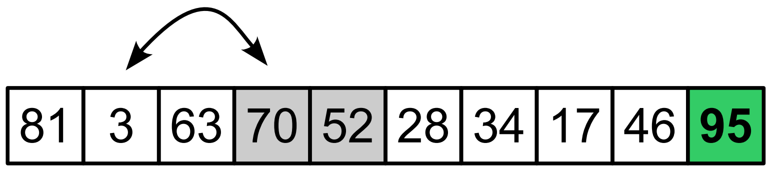
- Swap 95 and 3, then percolate down 3:



- Green highlight indicates part of the output
- Light grey highlight shows the children of the node being percolated
  - Remember that the children of node at *k* are 2*k* and 2*k*+1 (if assigning first element as subscript 1)

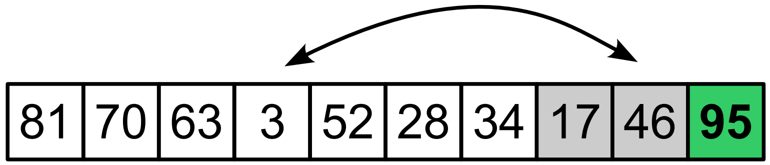
Heap sort

- Swap 95 and 3, then percolate down 3:



Heap sort

- Swap 95 and 3, then percolate down 3:



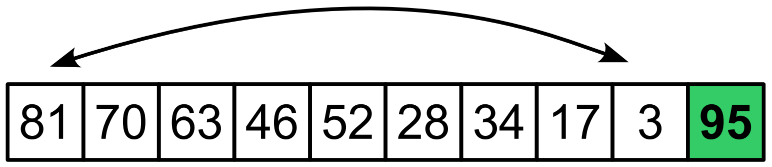
Heap sort

- The fact that 3 made it to the last (back) position is mere coincidence!

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

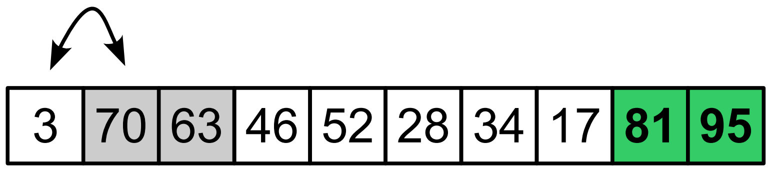
Heap sort

- We now swap 81 and 3, and percolate down 3:



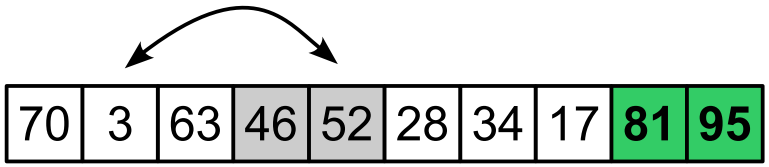
Heap sort

- We now swap 81 and 3, and percolate down 3:



Heap sort

- We now swap 81 and 3, and percolate down 3:



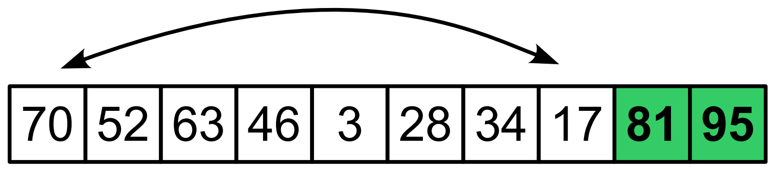
Heap sort

- Done — 52 was a leaf node:

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

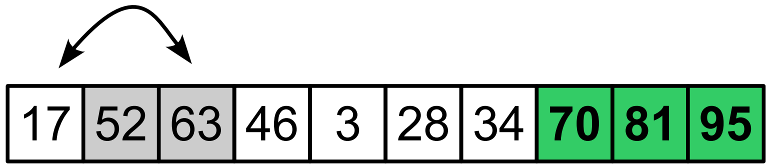
Heap sort

- Now dequeue 70 (swapping with 17 and percolating down 70):



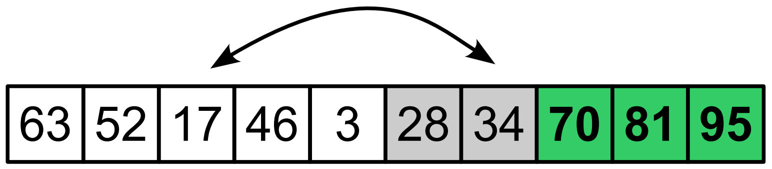
Heap sort

- Now dequeue 70 (swapping with 17 and percolating down 70):



Heap sort

- Now dequeue 70 (swapping with 17 and percolating down 70):



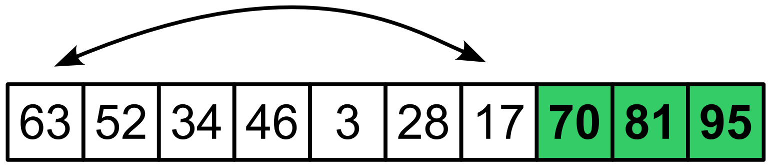
Heap sort

- We repeat this same process for all elements...

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

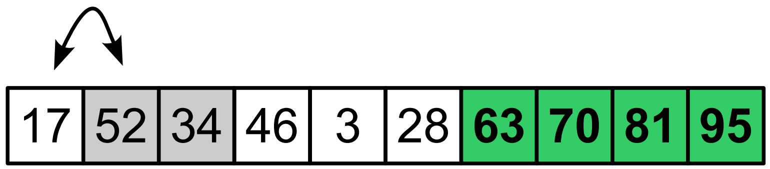
Heap sort

- We repeat this same process for all elements...



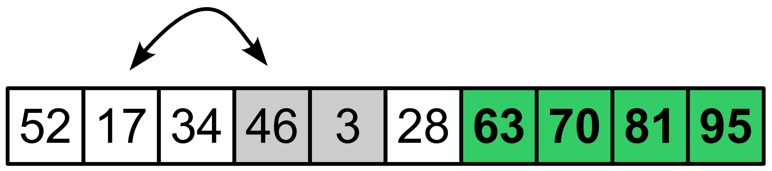
Heap sort

- We repeat this same process for all elements...



Heap sort

- We repeat this same process for all elements...



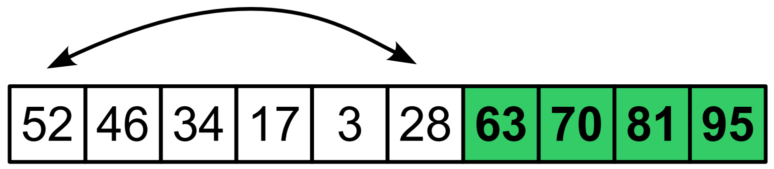
Heap sort

- We repeat this same process for all elements...

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

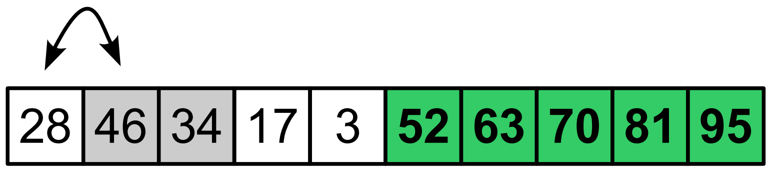
Heap sort

- We repeat this same process for all elements...



Heap sort

- We repeat this same process for all elements...



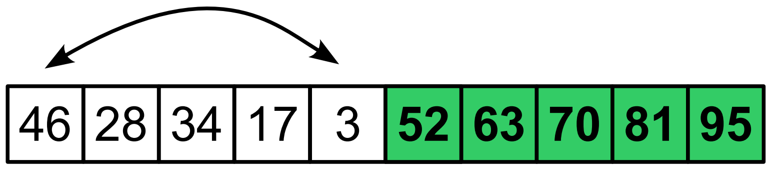
Heap sort

- We repeat this same process for all elements...

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

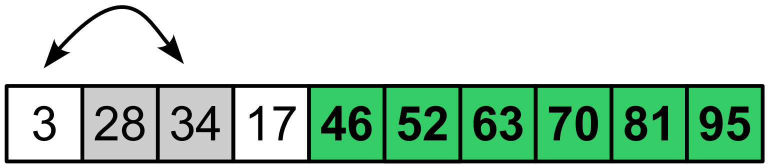
Heap sort

- We repeat this same process for all elements...



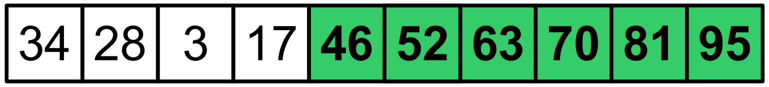
Heap sort

- We repeat this same process for all elements...



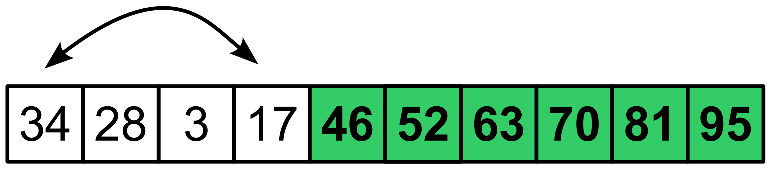
Heap sort

- We repeat this same process for all elements...



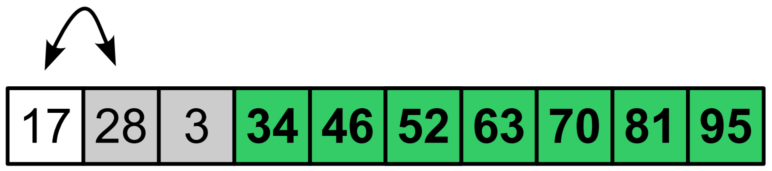
Heap sort

- We repeat this same process for all elements...



Heap sort

- We repeat this same process for all elements...



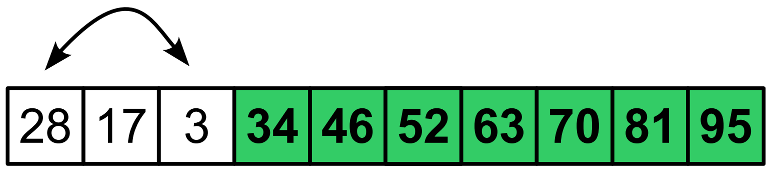
Heap sort

- We repeat this same process for all elements...

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

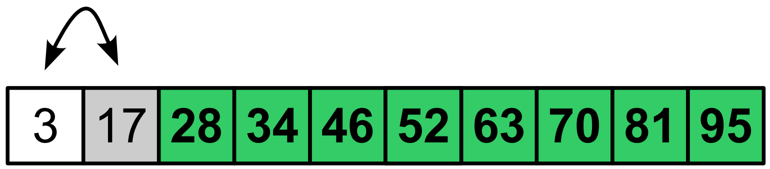
Heap sort

- We repeat this same process for all elements...



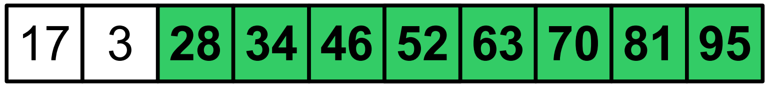
Heap sort

- We repeat this same process for all elements...



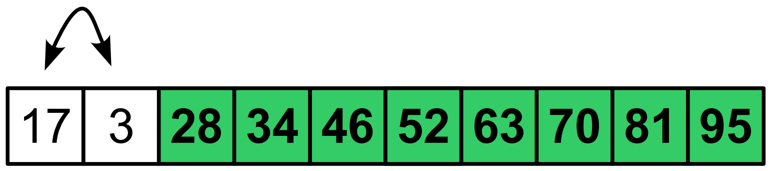
Heap sort

- We repeat this same process for all elements...



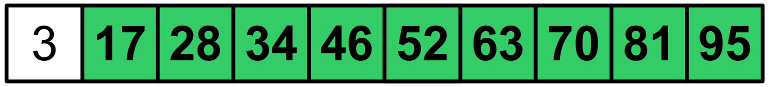
Heap sort

- We repeat this same process for all elements...



Heap sort

- We repeat this same process for all elements...



Heap sort

- The algorithm outputs the sorted array.

3	17	28	34	46	52	63	70	81	95

Summary

- During today's lesson:
  - Saw the distinction between max-heaps and min-heaps.
  - Introduced the Heap sort algorithm
  - Determined its run time
  - Discuss heapification and its run time
  - Argued that an algorithm that includes something called *heapification* has to be considered cool upon cool...
    - But don't worry — we have *Quick sort* coming up next class, so there will be no shortage of coolness !!