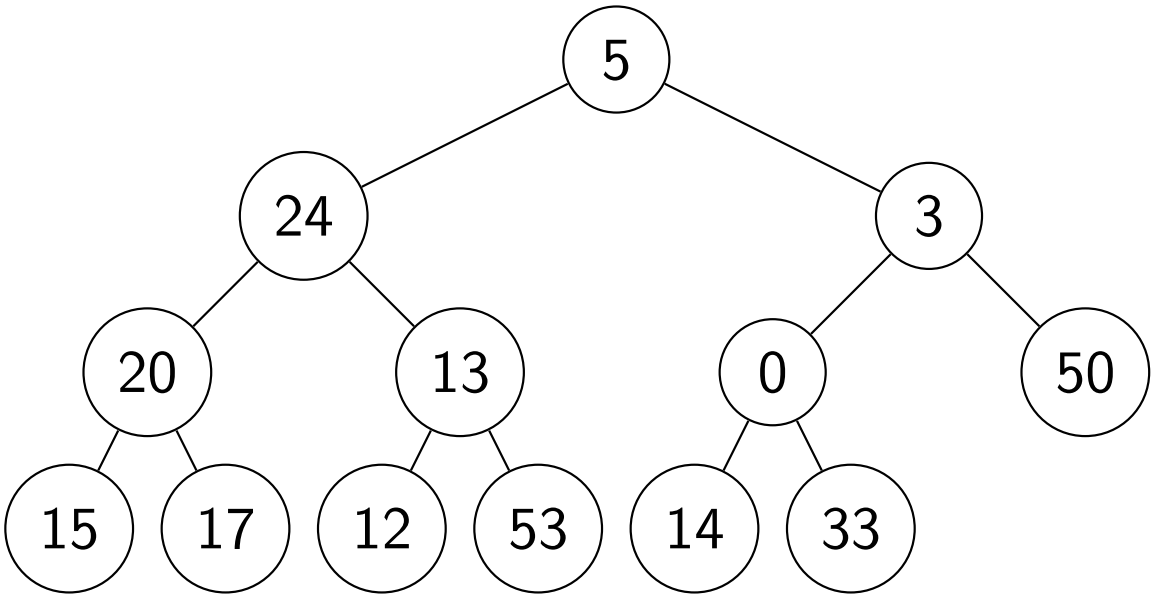
Heapifying and Heap Sort

Helen Cameron

Comp 2140

Heapifying an Array

What if you have an array full of values (not in heap order) that you want to turn into a priority queue?



0	1	2	3	4	5	6	7	8	9	10	11	12
5	24	3	20	13	0	50	15	17	12	53	14	33

How NOT to Heapify

Do NOT insert the values one at a time into a heap.

This idea is too slow ($O(n \log n)$) compared to the efficient heapify algorithm (which is $O(n)$).

Efficient Heapify Algorithm

(Leaves Up to Root)

Efficient Heapify Algorithm

Heapify is faster if you work **from the leaves up to root.**

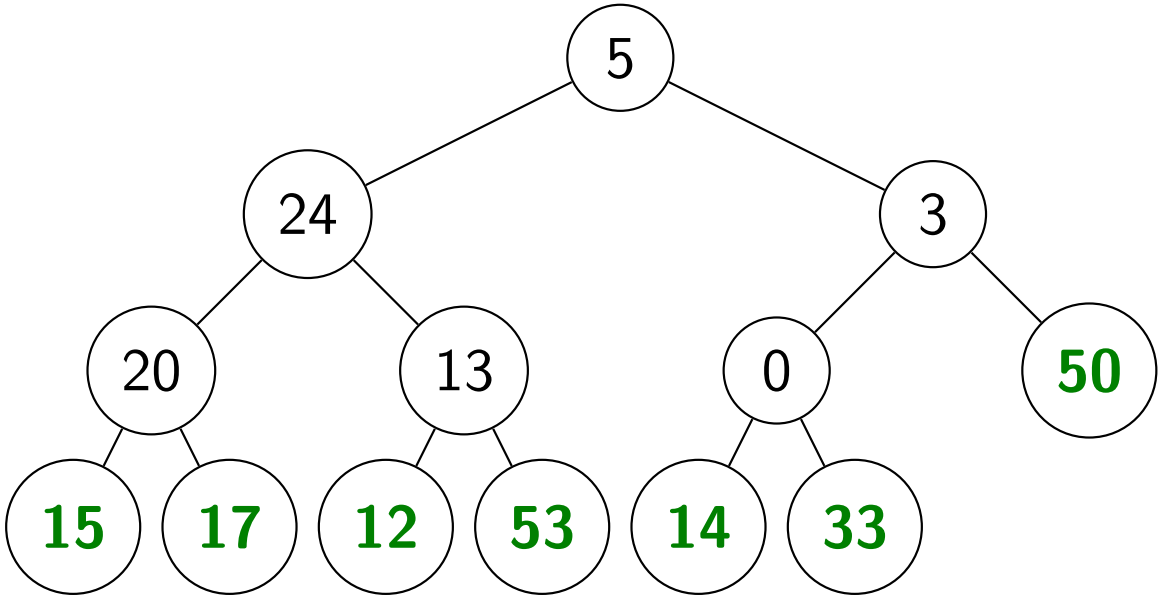
Loop **from right to left** through the array, **sifting down** a value at each step.

Because we're working from the leaves up, at each iteration of the loop, the children (or child) of the current value are already heaps.

So an iteration of the loop is just like the fixing-up step of `deleteMax` after we've placed the last value in the root position.

Efficient Heapify Algorithm: Example

At the start, each leaf is a heap by itself:

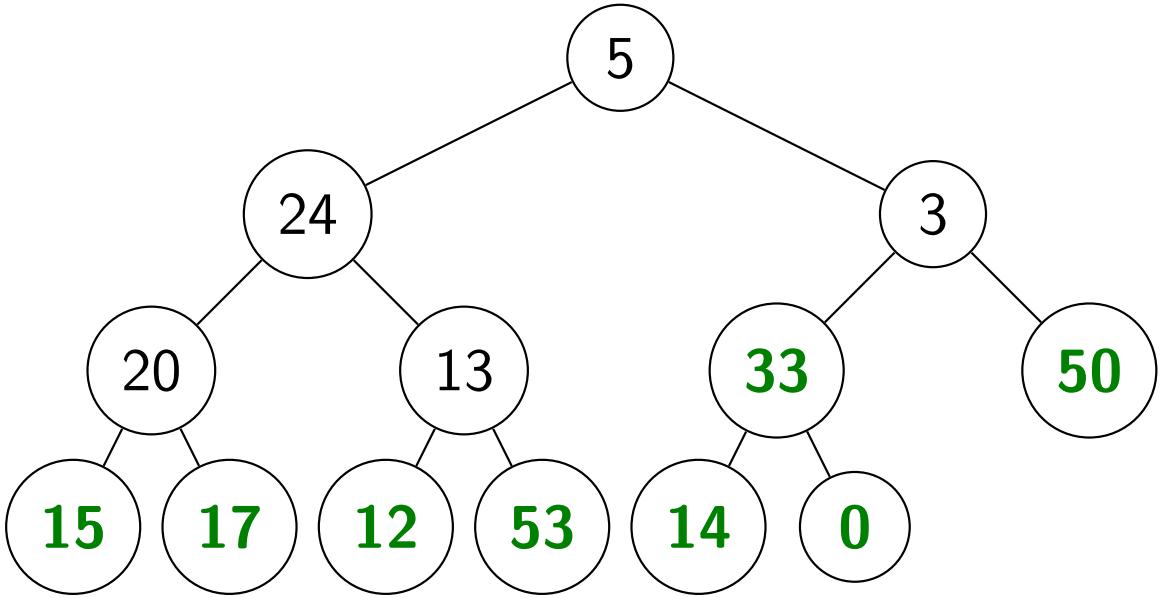


0	1	2	3	4	5	6	7	8	9	10	11	12
5	24	3	20	13	0	**50**	**15**	**17**	**12**	**53**	**14**	**33**

Now sift down 0.

Efficient Heapify Algorithm: Example

To sift down 0, we had to exchange 0 and its larger child 33:

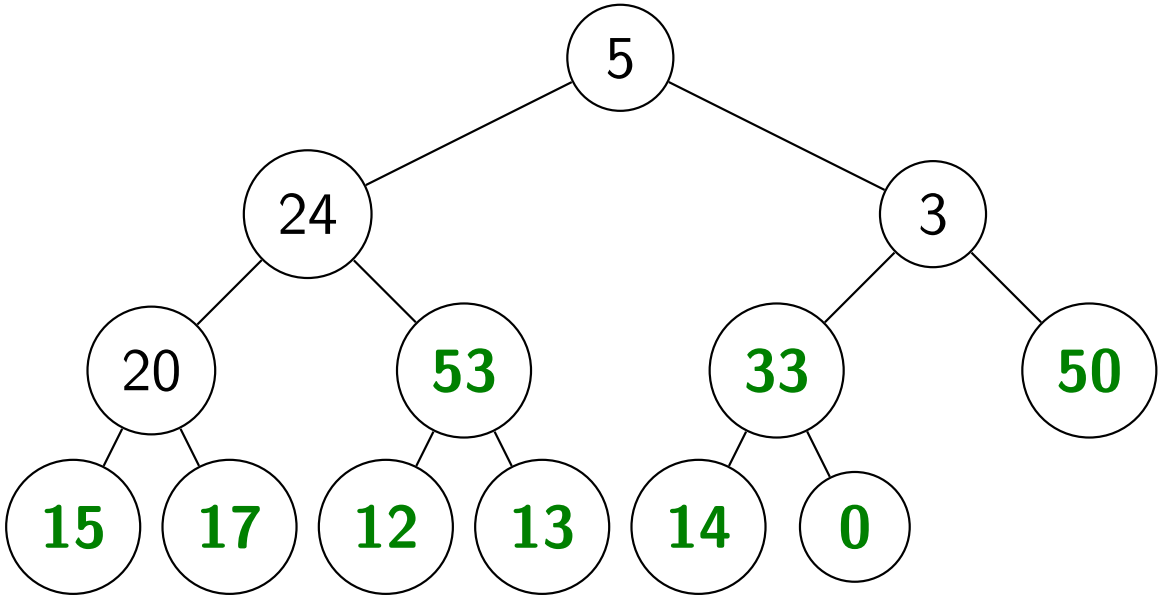


0	1	2	3	4	5	6	7	8	9	10	11	12
5	24	3	20	13	**33**	**50**	**15**	**17**	**12**	**53**	**14**	**0**

Now sift down 13.

Efficient Heapify Algorithm: Example

To sift down 13, we had to exchange 13 and its larger child 53:

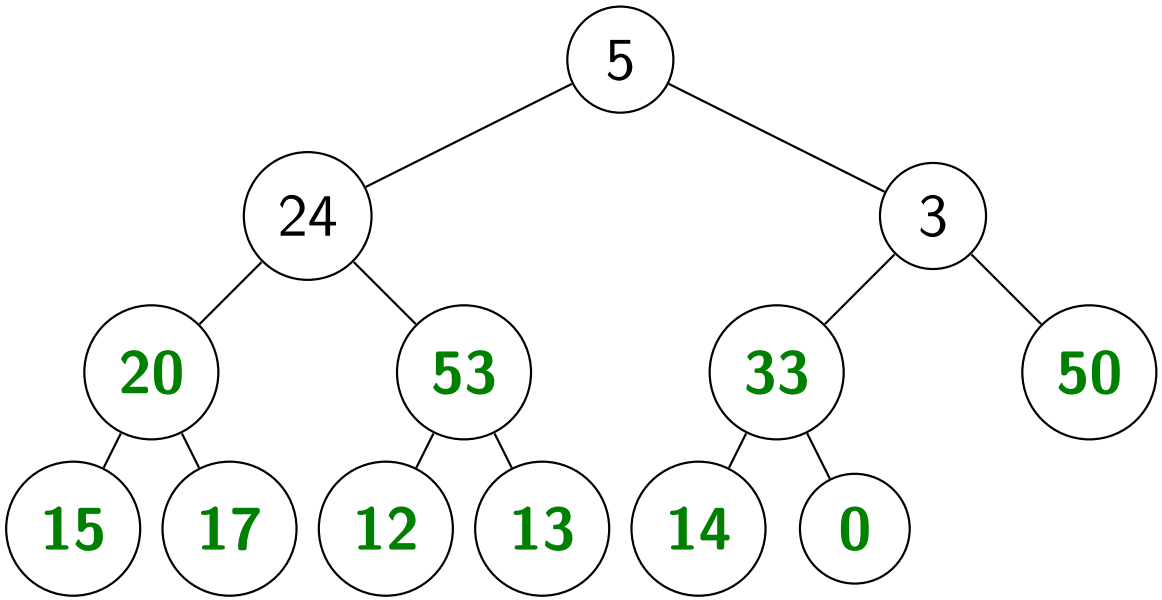


0	1	2	3	4	5	6	7	8	9	10	11	12
5	24	3	20	**53**	**33**	**50**	**15**	**17**	**12**	**13**	**14**	**0**

Now sift down 20.

Efficient Heapify Algorithm: Example

To sift down 20, we don't need to do anything because 20 is larger than both of its children already:

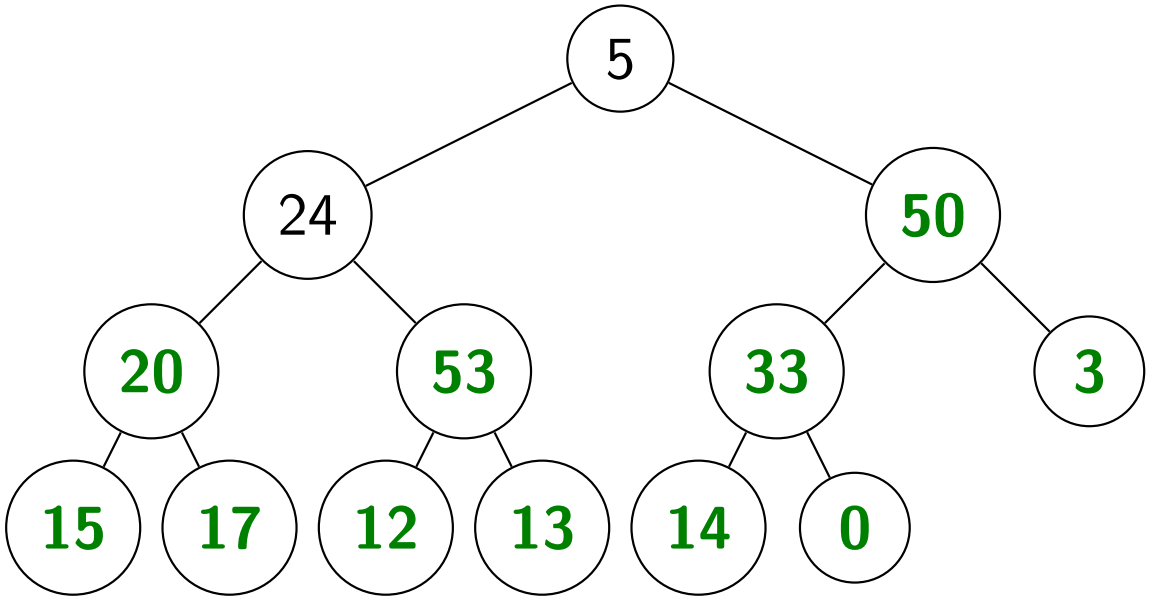


0	1	2	3	4	5	6	7	8	9	10	11	12
5	24	3	**20**	**53**	**33**	**50**	**15**	**17**	**12**	**13**	**14**	**0**

Now sift down 3.

Efficient Heapify Algorithm: Example

To sift down 3, we exchange 3 with 50, the larger of its two children:

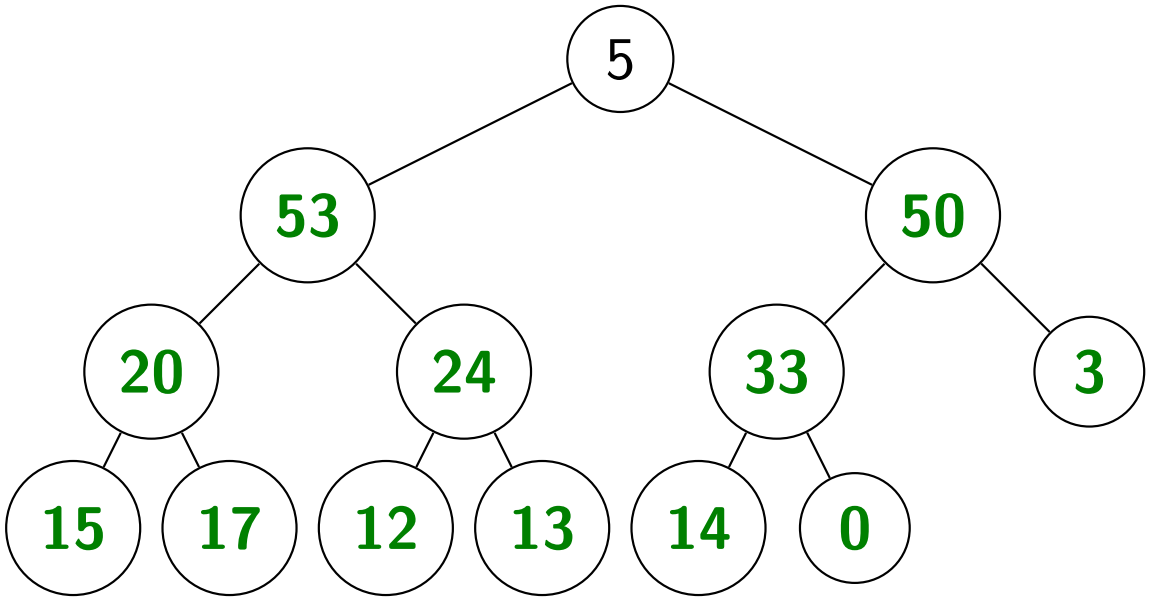


0	1	2	3	4	5	6	7	8	9	10	11	12
5	24	**50**	**20**	**53**	**33**	**3**	**15**	**17**	**12**	**13**	**14**	**0**

Now sift down 24.

Efficient Heapify Algorithm: Example

To sift down 24, we exchange 24 with 53, the larger of its two children, and then 24 is larger than its two children:

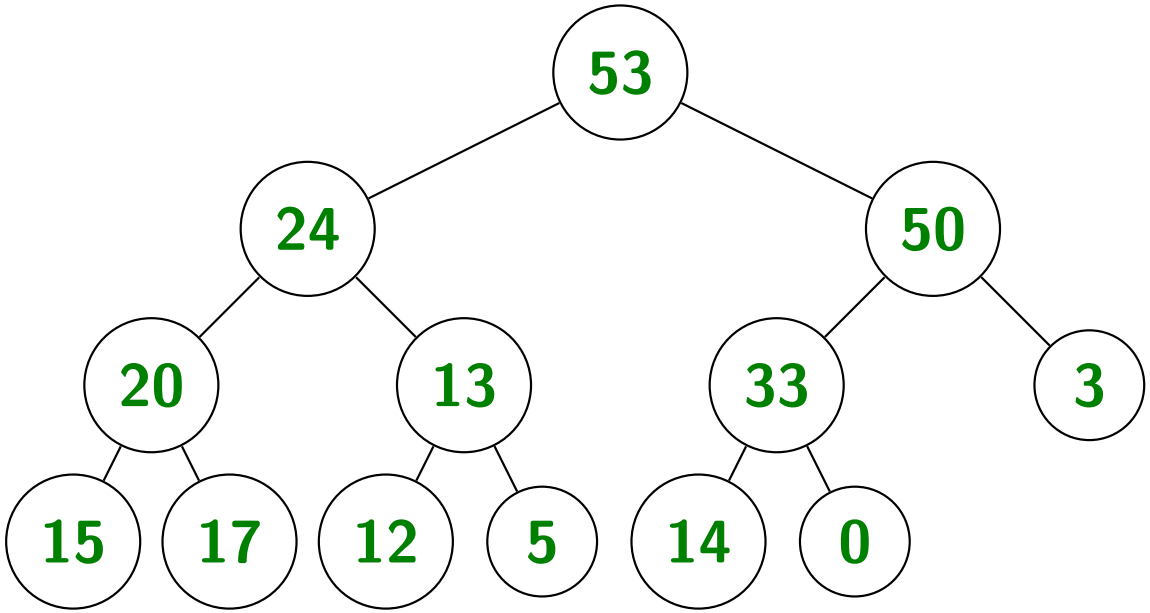


0	1	2	3	4	5	6	7	8	9	10	11	12
5	53	50	20	24	33	3	15	17	12	13	14	0

Finally, sift down 5.

Efficient Heapify Algorithm: Example

To sift down 5, we exchange 5 with 53, then with 24, then with 13:

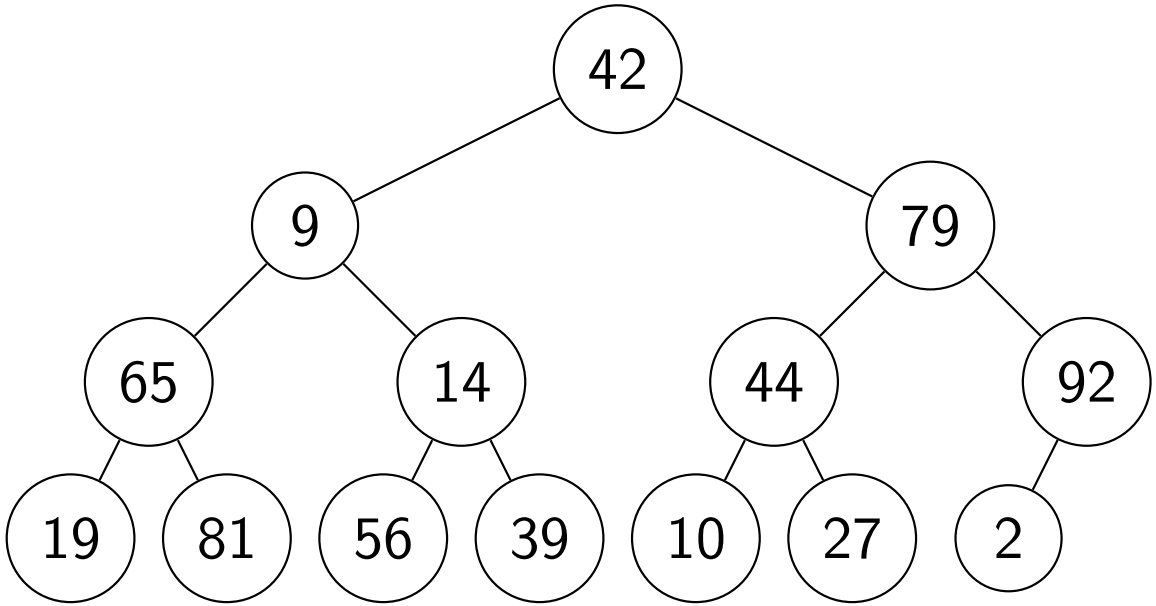


0	1	2	3	4	5	6	7	8	9	10	11	12
53	24	50	20	13	33	3	15	17	12	5	14	0

Now the array is a heap!

Efficient Heapify Algorithm: Example

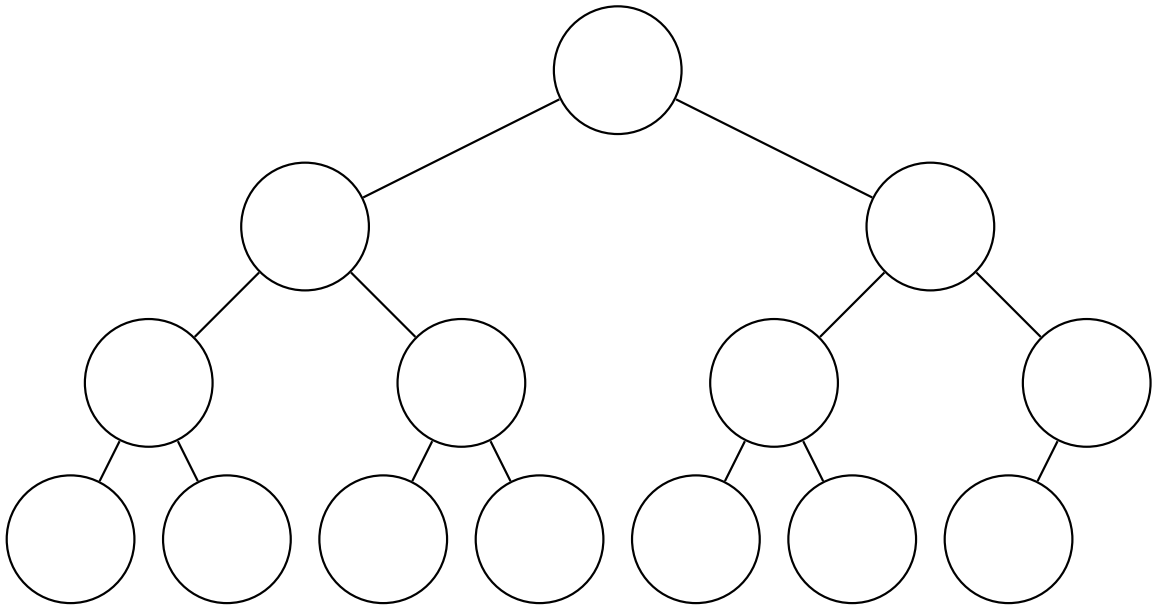
Try it: Heapify the following using the efficient heapify algorithm:

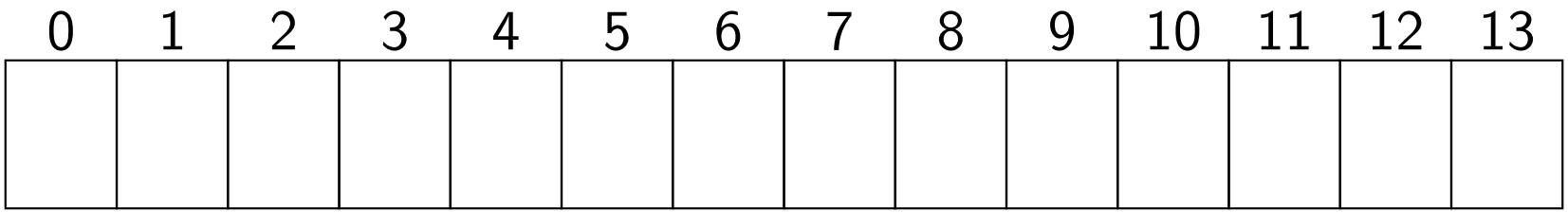


0	1	2	3	4	5	6	7	8	9	10	11	12	13
42	9	79	65	14	44	92	19	81	56	39	10	27	2

Efficient Heapify Algorithm: Example

The results of using the efficient heapify algorithm on the example:





Pseudocode

```
for index i starting at the parent of the rightmost leaf
        in the array and ending at the root
{
  sift down the item at index i in the array;
} // end for-loop
```

Heap Sort

Heap Sort

Heap sort does not need any extra storage. (For comparison, quick sort doesn't need any, either; but merge sort does use an extra array in its merge step.)

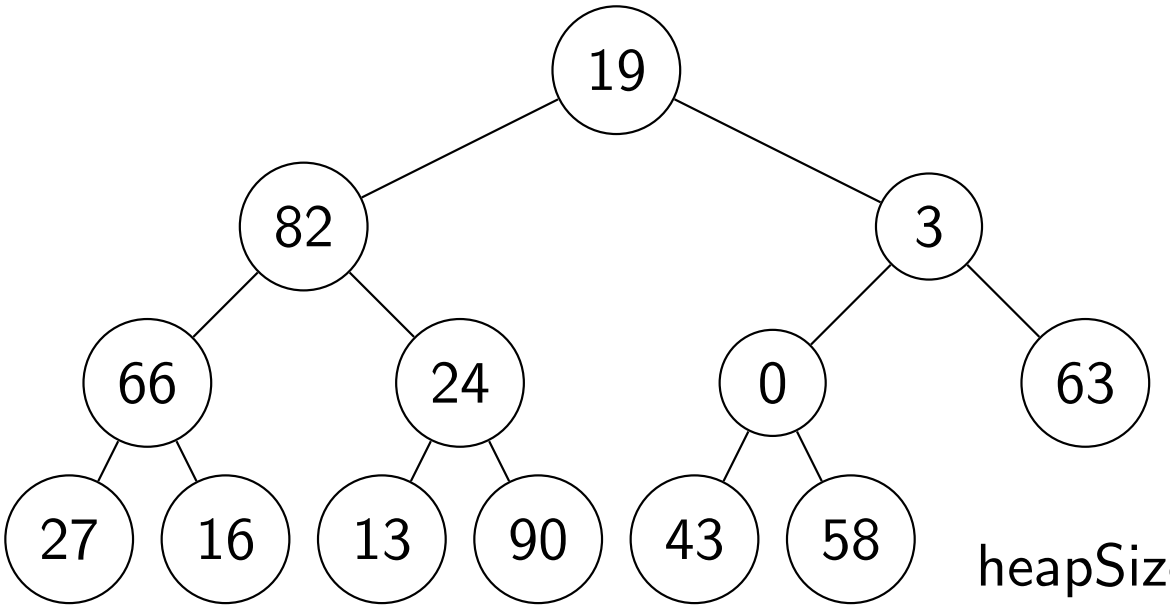
Like other sorts, heap sort starts with an unsorted array.

Heap Sort

First step: **Heapify** the unsorted array using the efficient heapify algorithm.

Heap Sort Example

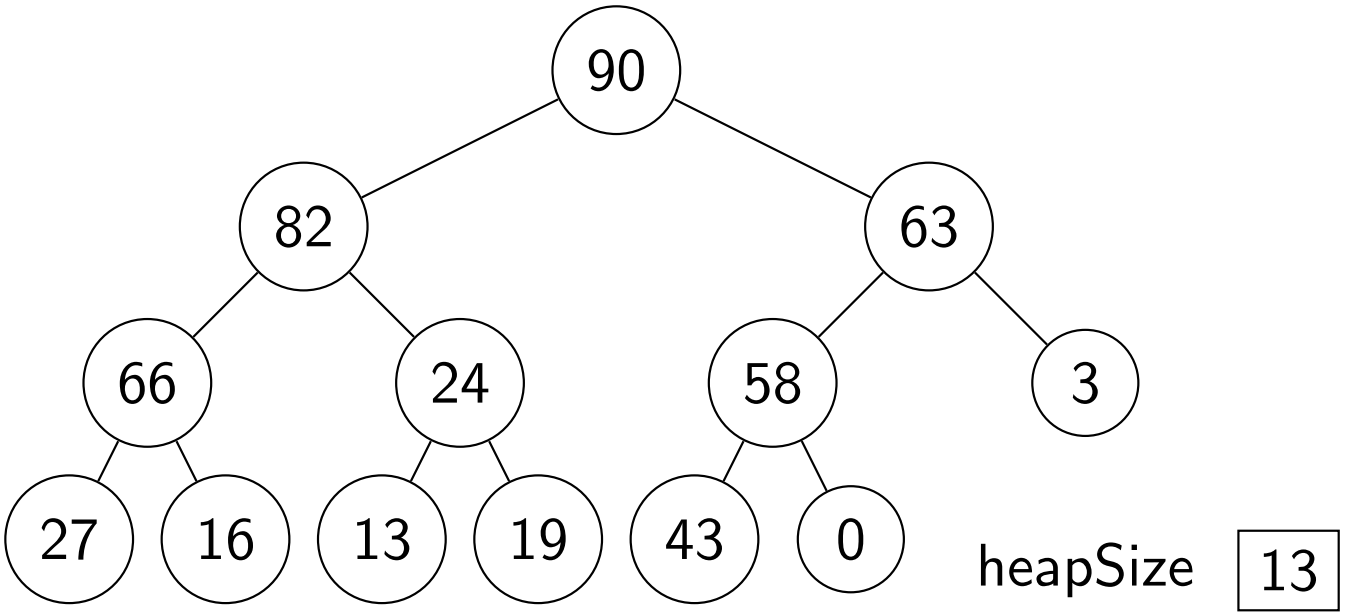
Start with an unsorted array:



0	1	2	3	4	5	6	7	8	9	10	11	12
19	82	3	66	24	0	63	27	16	13	90	43	58

Heap Sort Example

After the first step, it's a max heap:



0	1	2	3	4	5	6	7	8	9	10	11	12
90	82	63	66	24	58	3	27	16	13	19	43	0

Heap Sort

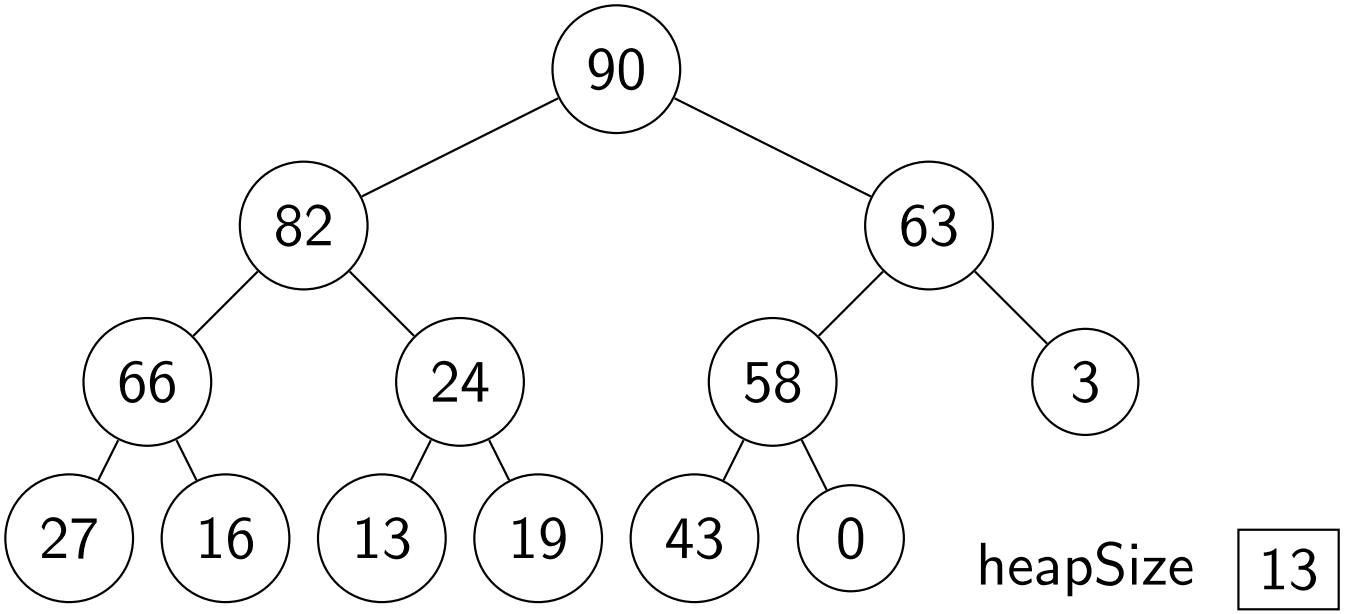
Second step: Repeatedly:

- Do a `deleteMax` and
- **Put the value returned** by `deleteMax` **into the empty position** left by the leaf that was moved to the root in the `deleteMax`.

The list is sorted when only one remains the heap (it's the smallest value and it's already in `heap[0]`.

Heap Sort Example

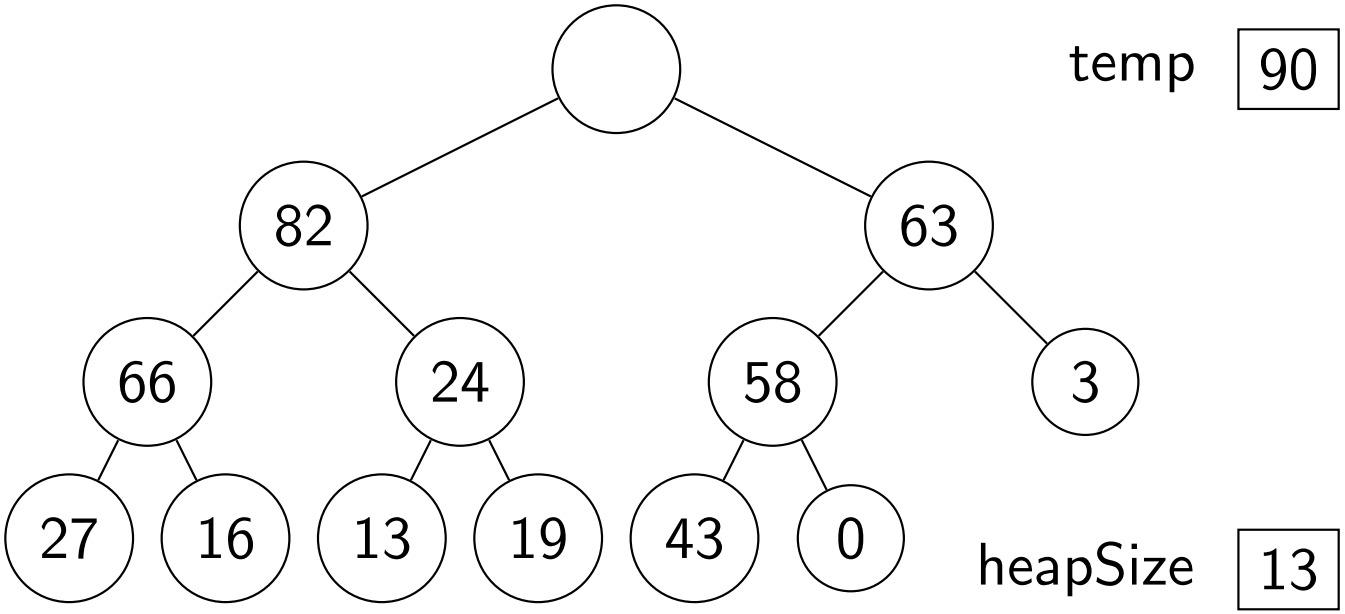
The very first `deleteMax` will remove the largest value (90) . . .



0	1	2	3	4	5	6	7	8	9	10	11	12
90	82	63	66	24	58	3	27	16	13	19	43	0

Heap Sort Example

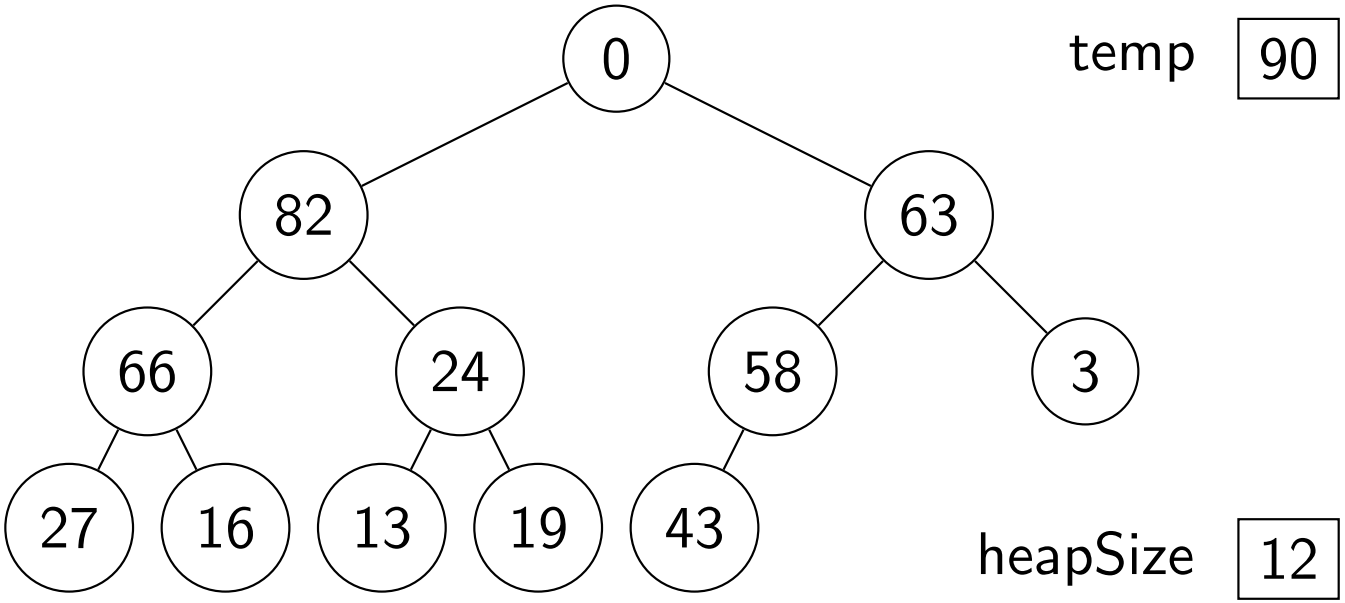
. . . And replace it with the rightmost leaf (0) . . .



0	1	2	3	4	5	6	7	8	9	10	11	12
	82	63	66	24	58	3	27	16	13	19	43	0

Heap Sort Example

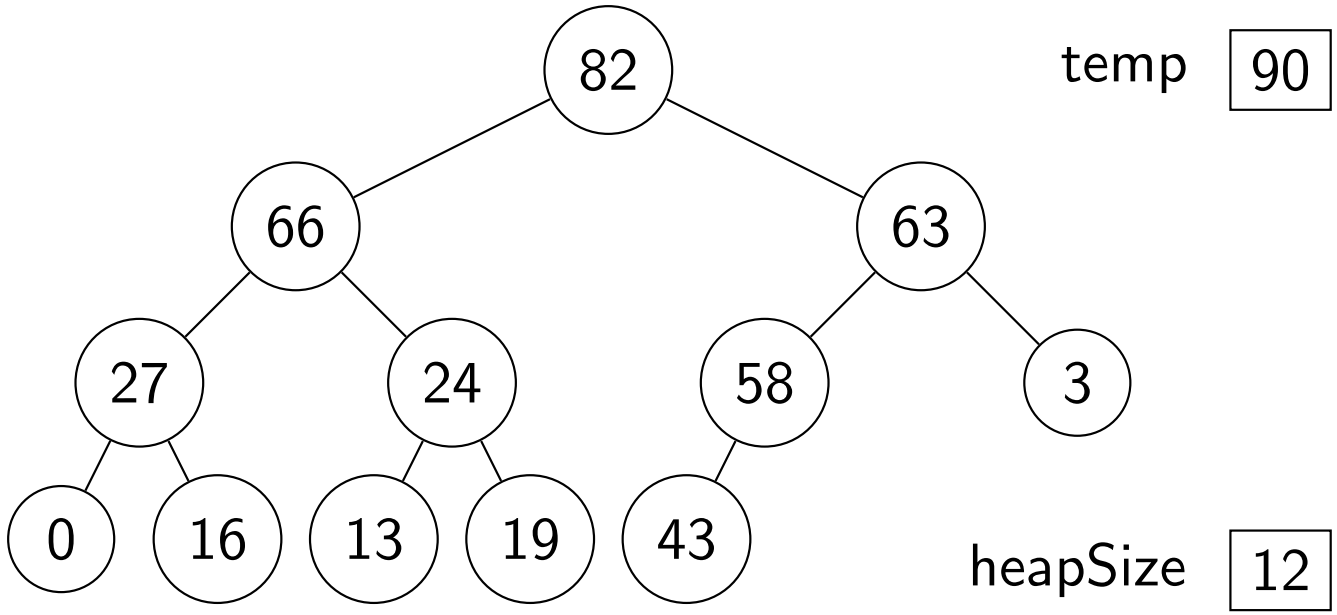
. . . Then sift down as needed . . .



0	1	2	3	4	5	6	7	8	9	10	11	12
0	82	63	66	24	58	3	27	16	13	19	43	

Heap Sort Example

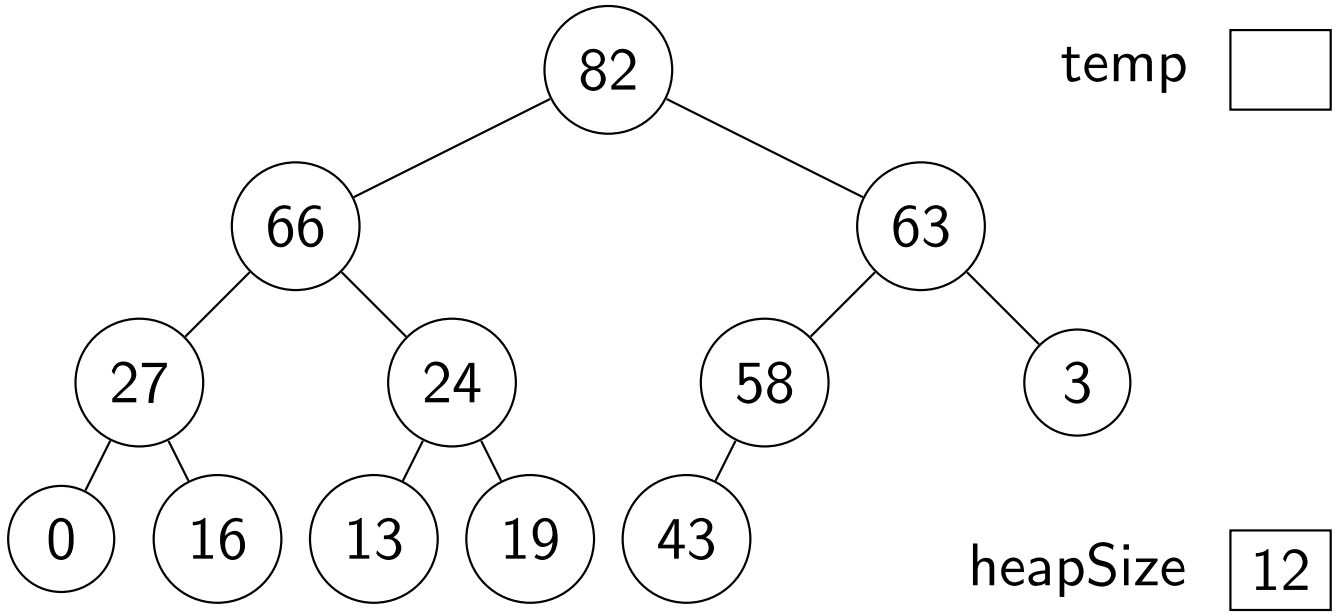
. . . After the `deleteMax`, we place the deleted value (90) into the empty array position vacated by the leaf:



0	1	2	3	4	5	6	7	8	9	10	11	12
82	66	63	27	24	58	3	0	16	13	19	43	

Heap Sort Example

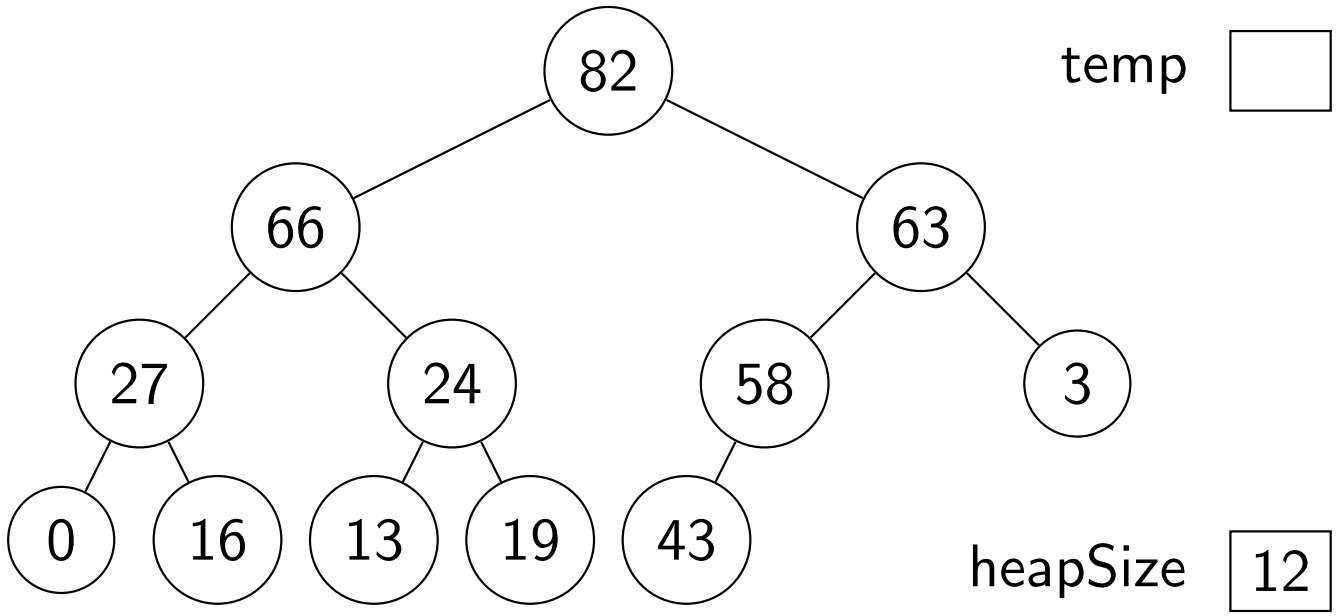
At the end of the first iteration, the heap uses all but the last position in the array and that last position contains a value in its correct sorted position:



0	1	2	3	4	5	6	7	8	9	10	11	12
82	66	63	27	24	58	3	0	16	13	19	43	90

Heap Sort Example

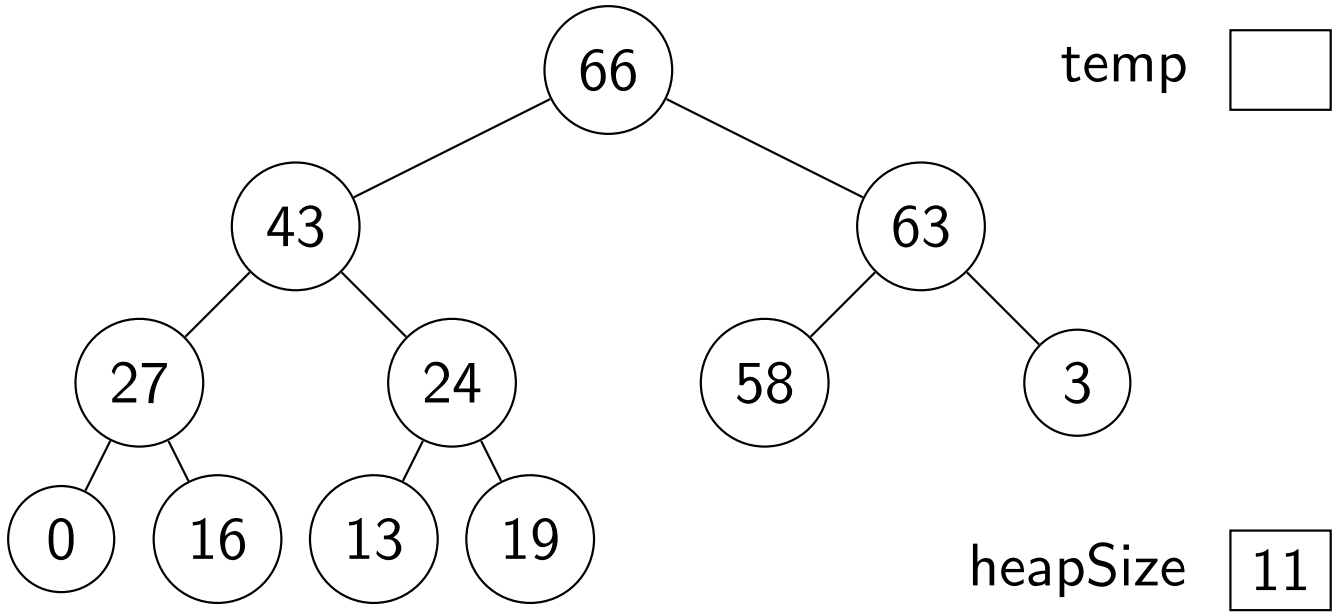
The second iteration, removes 82, puts leaf value 43 in its place, sifts 43 down as needed, and puts 82 into the position where 43 used to be:



0	1	2	3	4	5	6	7	8	9	10	11	12
82	66	63	27	24	58	3	0	16	13	19	43	90

Heap Sort Example

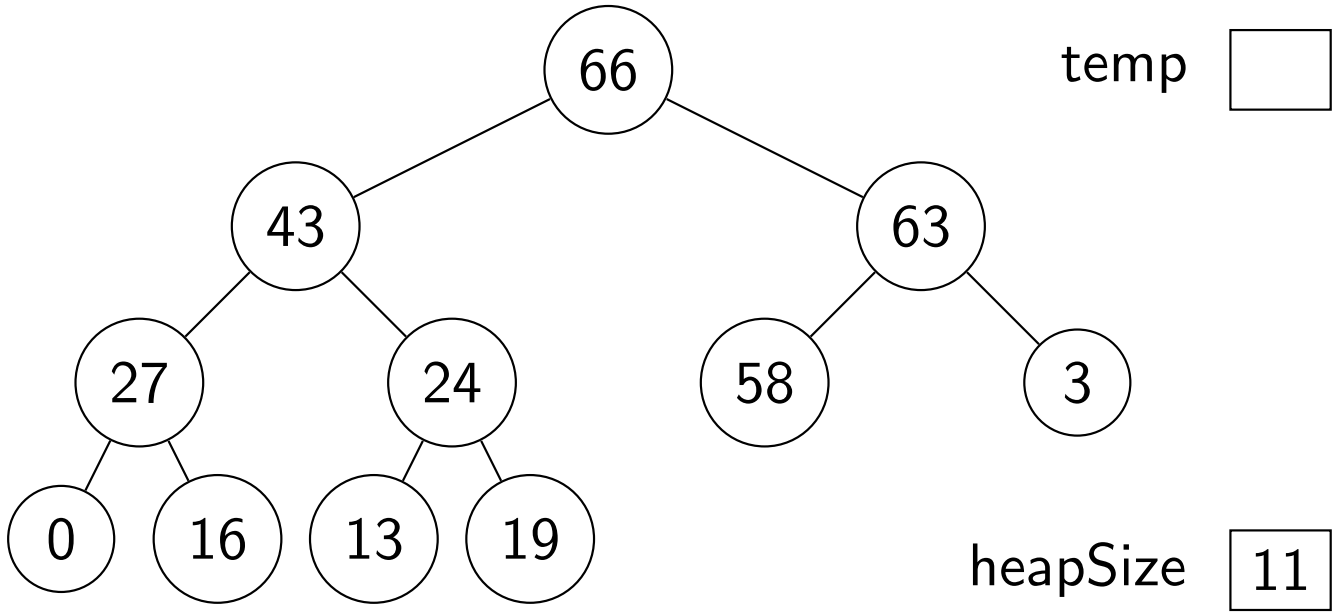
After the second iteration, two values are in their correct sorted positions:



0	1	2	3	4	5	6	7	8	9	10	11	12
66	43	63	27	24	58	3	0	16	13	19	82	90

Heap Sort Example

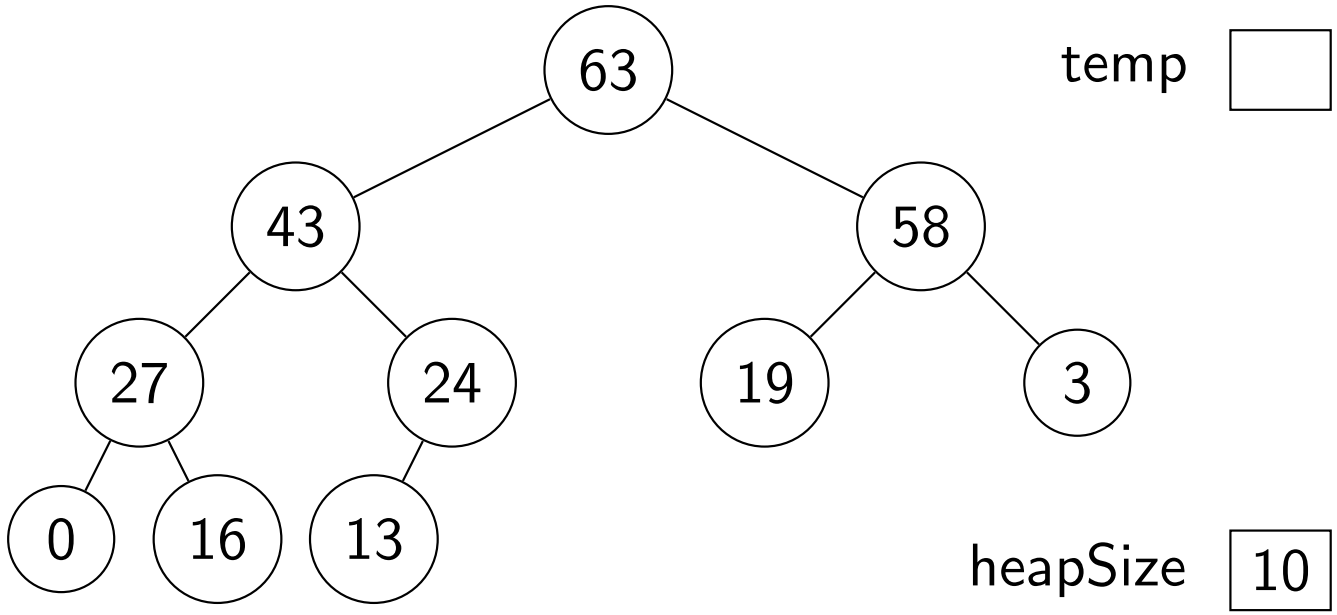
The third iteration deletes 66, moves leaf value 19 up to the root, sifts 19 down as needed, and, finally, places 66 in the empty leaf position vacated by 19.



0	1	2	3	4	5	6	7	8	9	10	11	12
66	43	63	27	24	58	3	0	16	13	19	82	90

Heap Sort Example

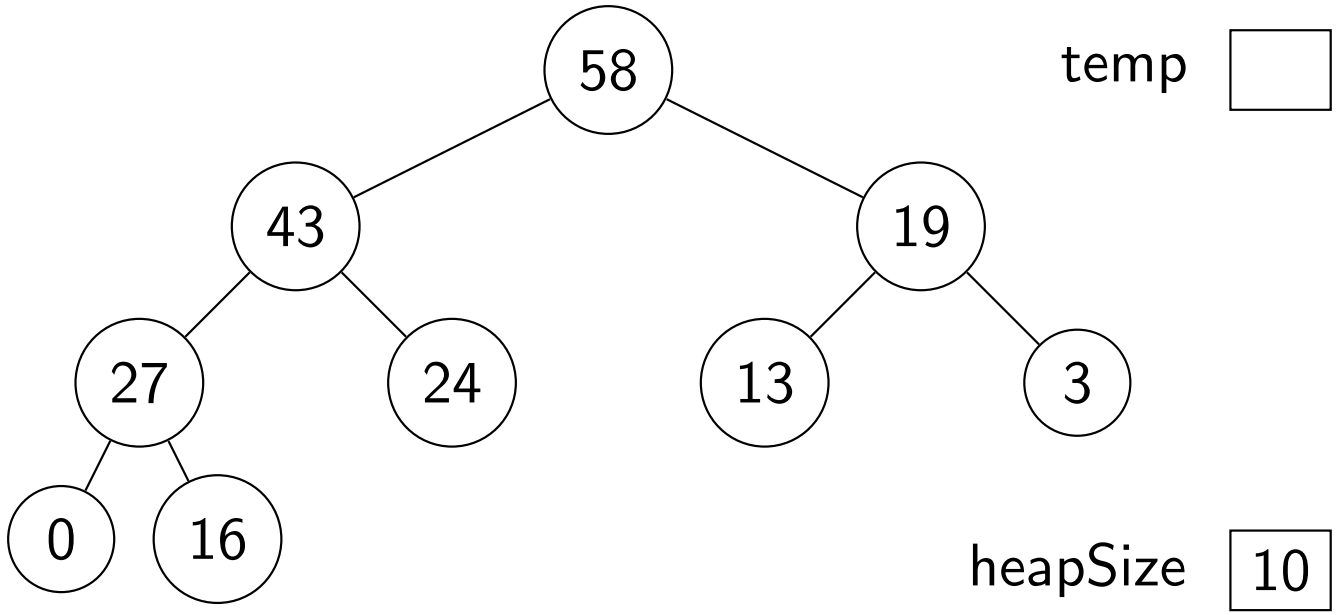
After the third iteration, three values are in their correct sorted positions:



0	1	2	3	4	5	6	7	8	9	10	11	12
63	43	58	27	24	19	3	0	16	13	66	82	90

Heap Sort Example

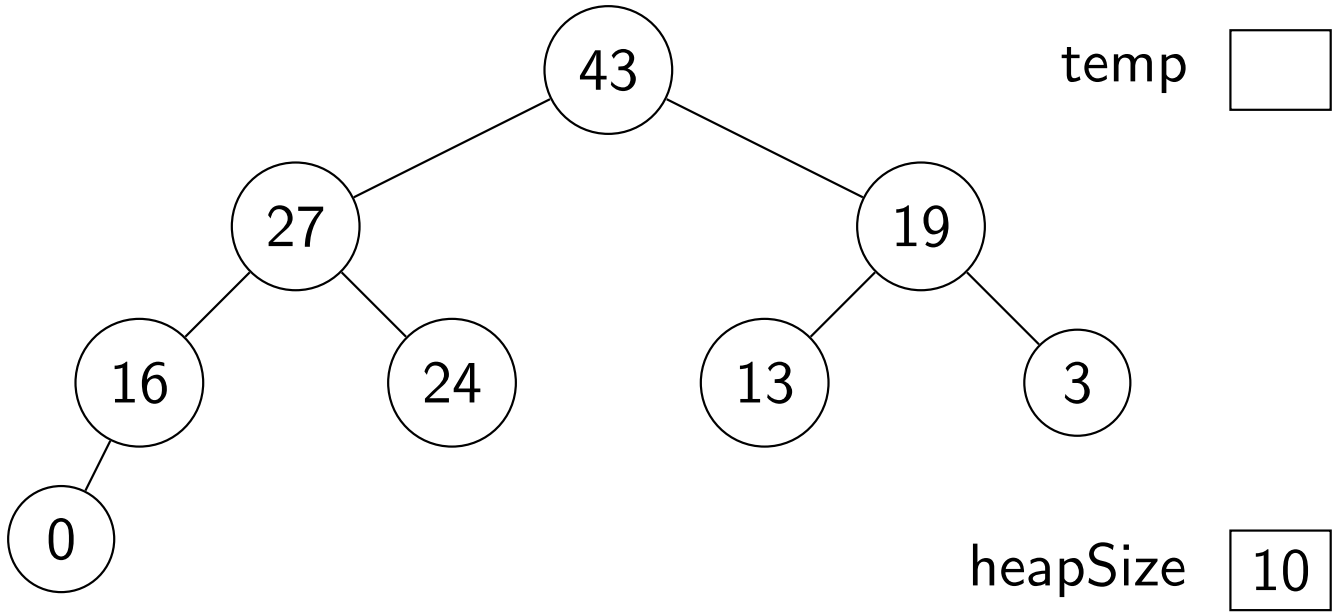
After the fourth iteration, four values are in their correct sorted positions:



0	1	2	3	4	5	6	7	8	9	10	11	12
58	43	19	27	24	13	3	0	16	63	66	82	90

Heap Sort Example

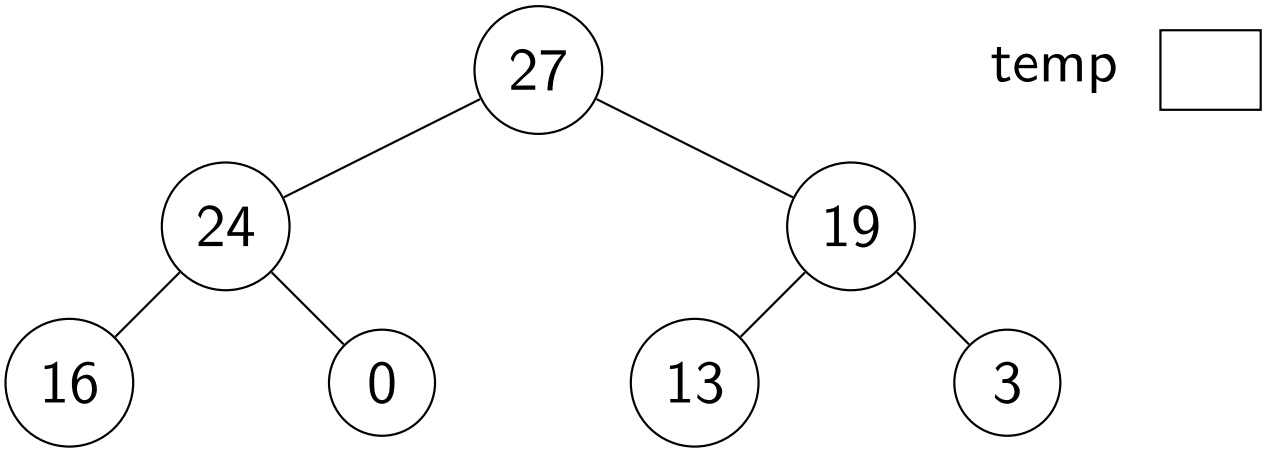
After the fifth iteration, five values are in their correct sorted positions:



0	1	2	3	4	5	6	7	8	9	10	11	12
43	27	19	16	24	13	3	0	58	63	66	82	90

Heap Sort Example

After the sixth iteration, six values are in their correct sorted positions:



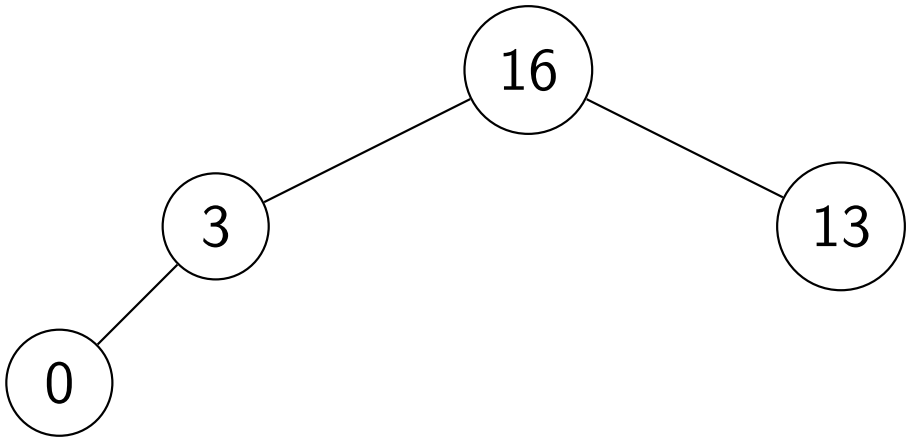
temp

heapSize 7

0	1	2	3	4	5	6	7	8	9	10	11	12
27	24	19	16	0	13	3	43	58	63	66	82	90

Heap Sort Example

After the ninth iteration, nine values are in their correct sorted positions:



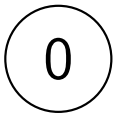
temp

heapSize 4

0	1	2	3	4	5	6	7	8	9	10	11	12
16	3	13	0	19	24	27	43	58	63	66	82	90

Heap Sort Example

After the twelfth iteration, twelve values are in their correct sorted positions:



heapSize 1

0	1	2	3	4	5	6	7	8	9	10	11	12
0	3	13	16	19	24	27	43	58	63	66	82	90

But with only one value (the minimum) left in the heap (in `heap[0]`), **ALL** values are in their correct sorted positions.

Efficiency of Heap Sort

- The first step, heapify, takes $O(n)$ steps with the efficient heapify algorithm.
- The second step is $n-1$ `deleteMax` operations (each costing at most $O(\log n)$ because there are at most n items) $\Rightarrow$ $O(n \log n)$ for this step.

Total running time: $O(n \log n)$**.**

But a good quick sort is faster than heap sort.