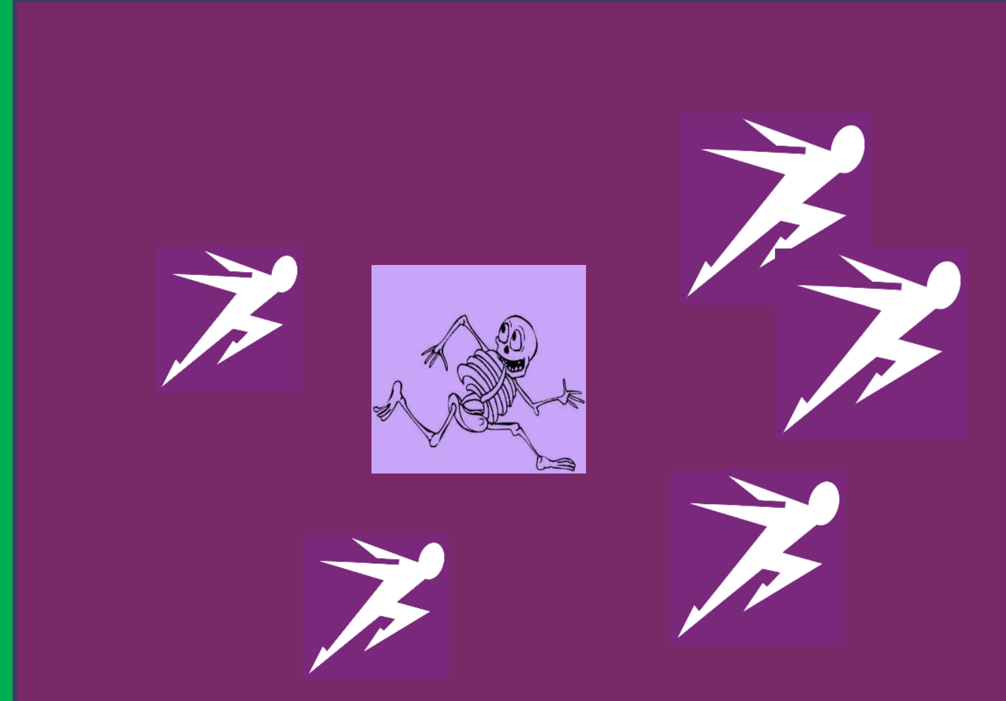


O(n log n)

HEAP SORT AND *QUICKSORT*

Lecture 33

On a *merge sort* recurrence

$$x(n) = 2x\left(\frac{n}{2}\right) + n; x(1) = 1.$$

Expanding the right-hand side, we get

$x(n) = 2\left[2x\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 4x\left(\frac{n}{4}\right) + 2n.$

Repeating this process ***k* = *log*$_2$*n*** times, we further obtain

$$x(n) = 4\left[2x\left(\frac{n}{8}\right) + n/4\right] + 2\mathrm{n} = 8\mathrm{x}\left(\frac{\mathrm{n}}{8}\right) + 3\mathrm{n} =$$

$= \ldots = 2^k x(n/2^k) + kn = n + n \log_2 n = O(n \log_2 n).$

Heapsort

Assignment

- Re-visit Lecture 24 and change its heap algorithms to work with max-heaps;
- Re-visit Lecture 25 to recall the implementation of a heap as an array
- Read sections 8.8 and 8.9 and do the self-check excercises

Objective and recollections

- Run another *n(log n)* show *without* using extra *O(n)* space—an improvement over the *Merge Sort*
- A *heap* is a complete (leaves are left-justified) binary tree of objects, in which the smallest object is at the root, while each the left subtree and the right subtree is a heap, too.

First Version of a Heapsort Algorithm

- The following algorithm
  - places an array's data into a heap,
  - then removes each heap item (O(*n* log *n*)) and moves it back into the array
- This version of the algorithm requires *n* extra storage locations

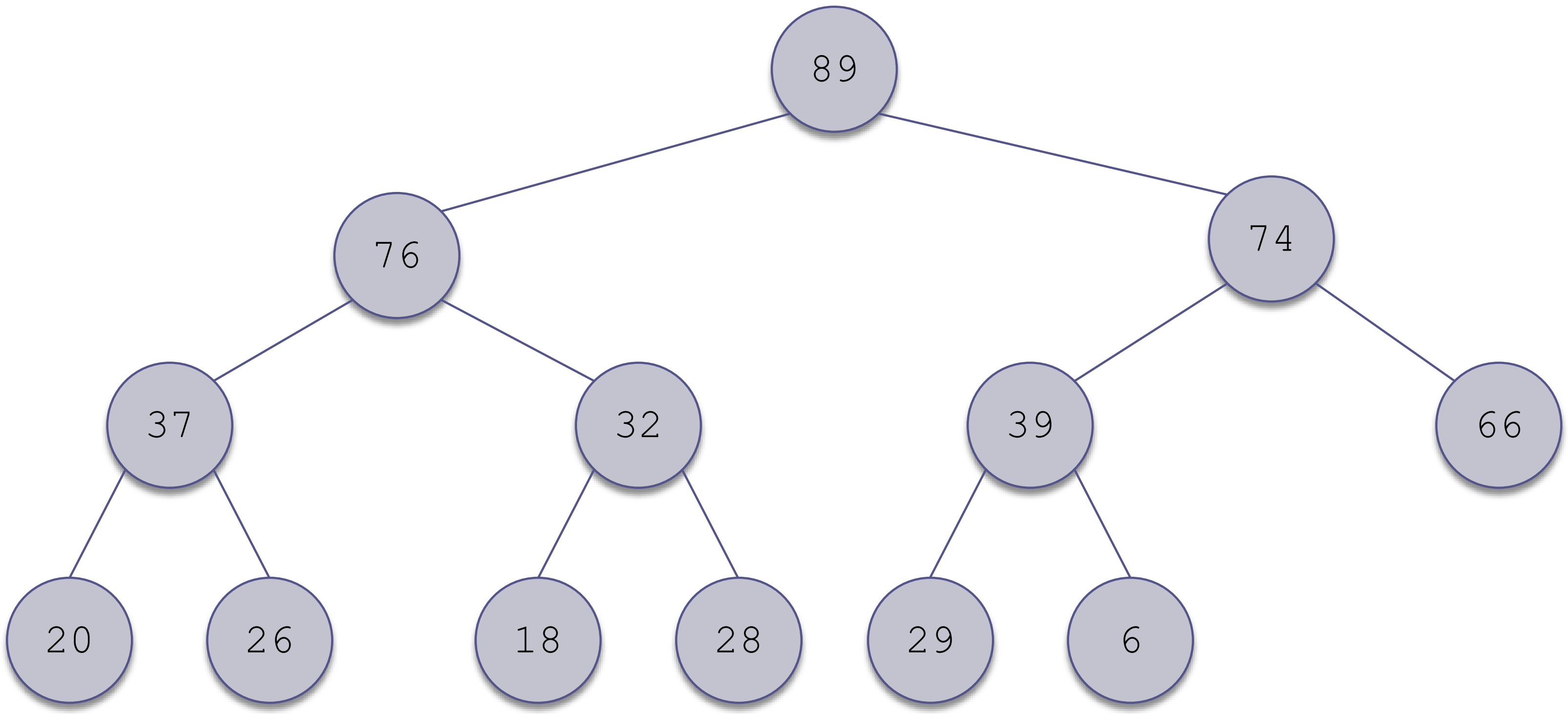
Heapsort Algorithm: First Version

1. **Insert each value from the array to be sorted into a priority queue (heap).**
2. **Set `i` to 0**
3. **`while` the priority queue is not empty**
4. **Remove an item from the queue and insert it back into the array at position `i`**
5. **Increment `i`**

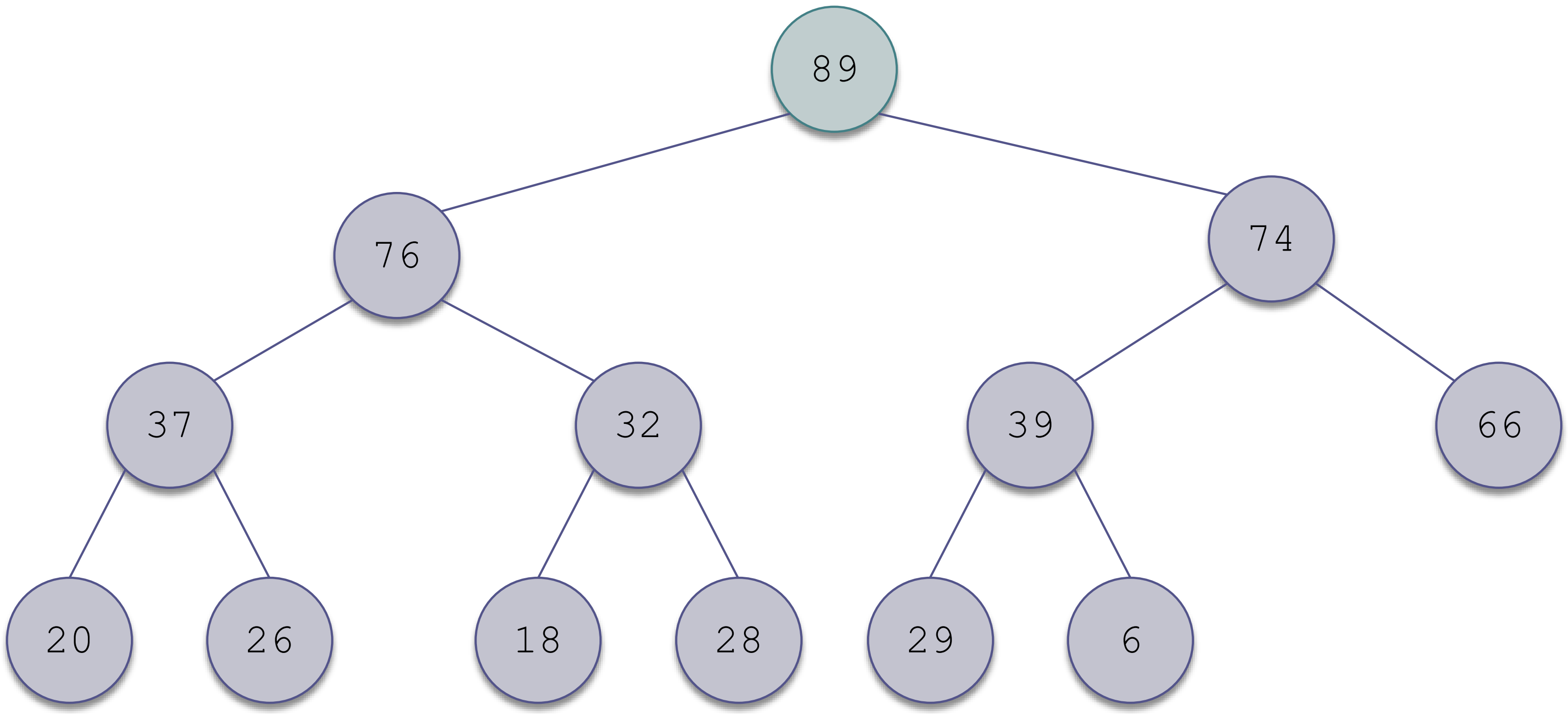
Revising the Heapsort Algorithm

- In heaps we've used so far, each parent node’s value was smaller than the values of its children
- We can build a heap (*max-heap*) so that each parent node value is *larger* than that of either children
- Then we
  - Swap the top item with the item at the bottom of the heap
  - Rebuild the heap, ignoring the item moved to the bottom

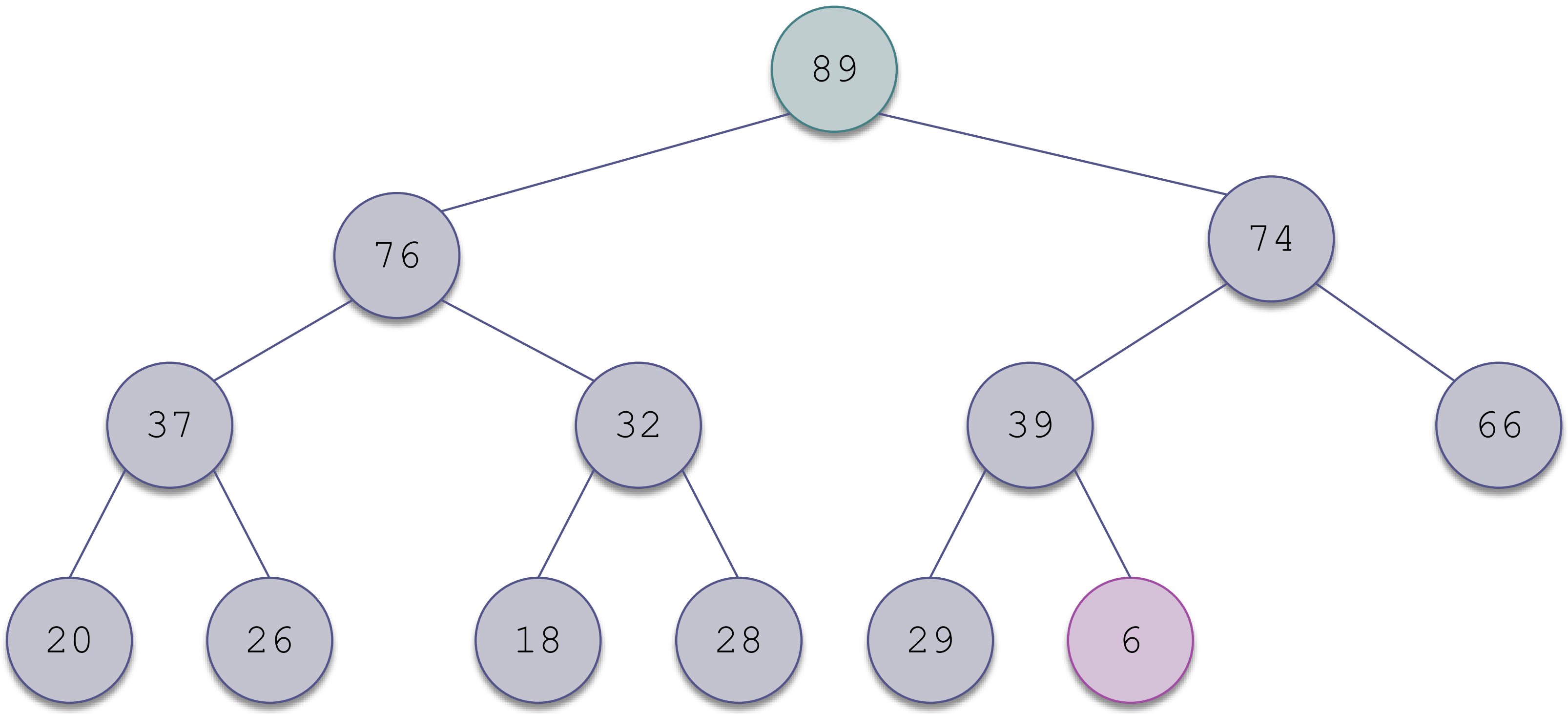
Trace of Heapsort



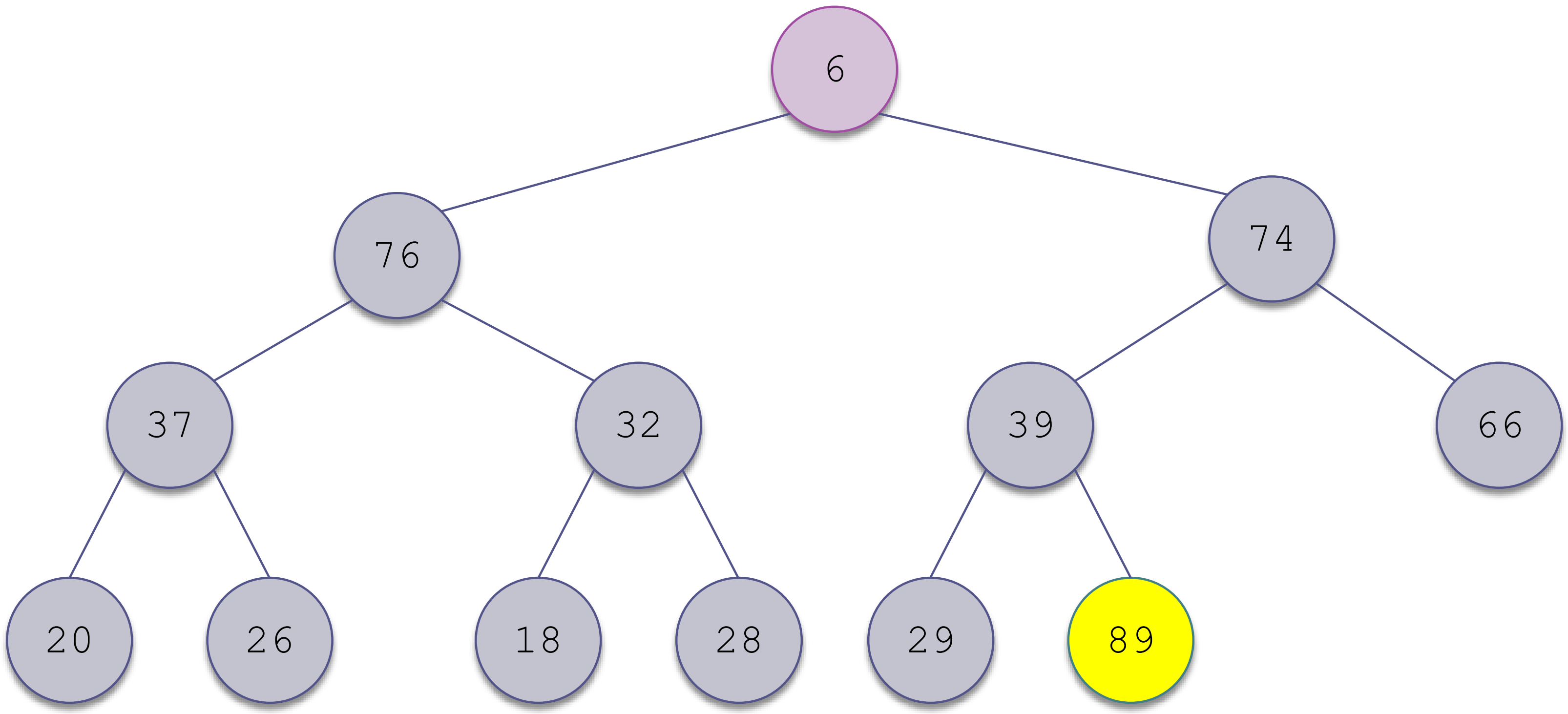
Trace of Heapsort (cont.)



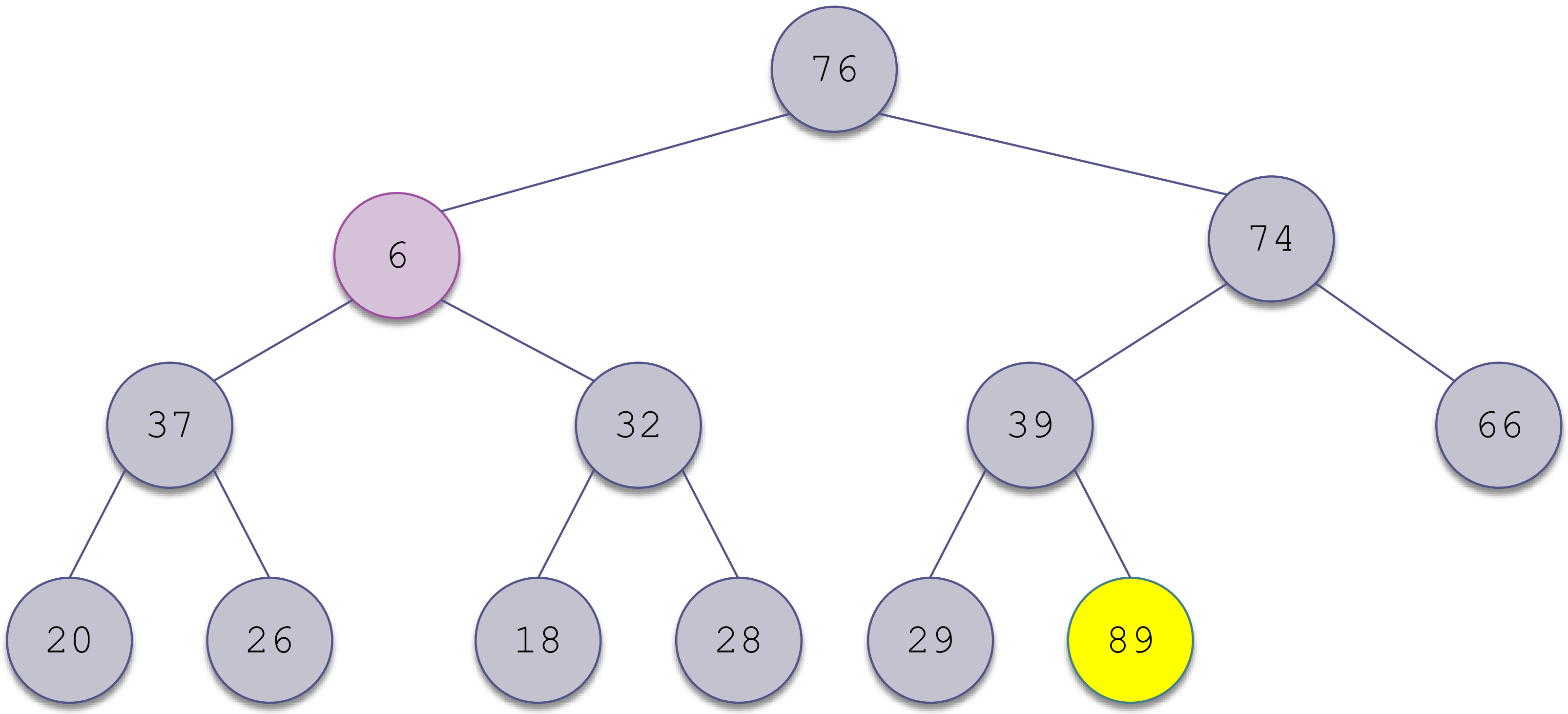
Trace of Heapsort (cont.)



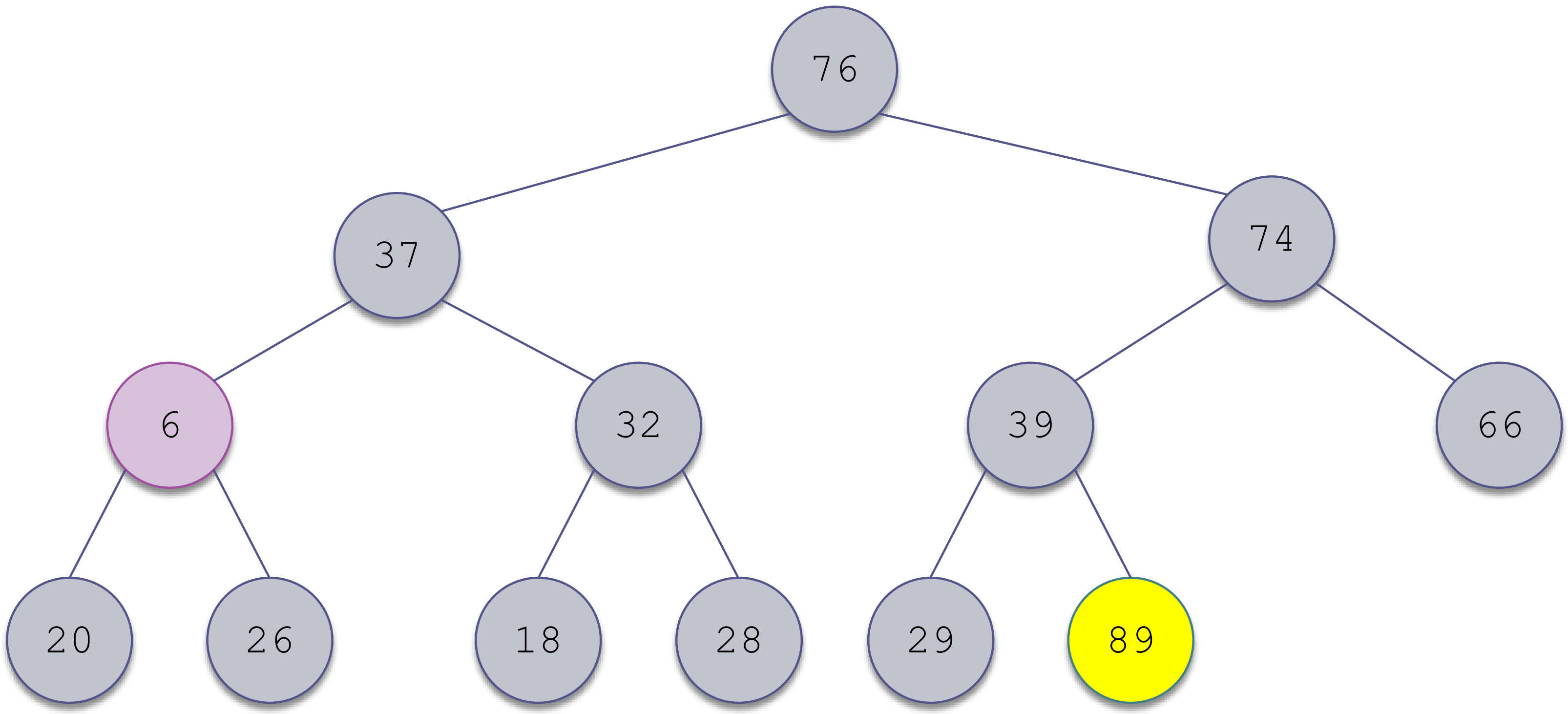
Trace of Heapsort (cont.)



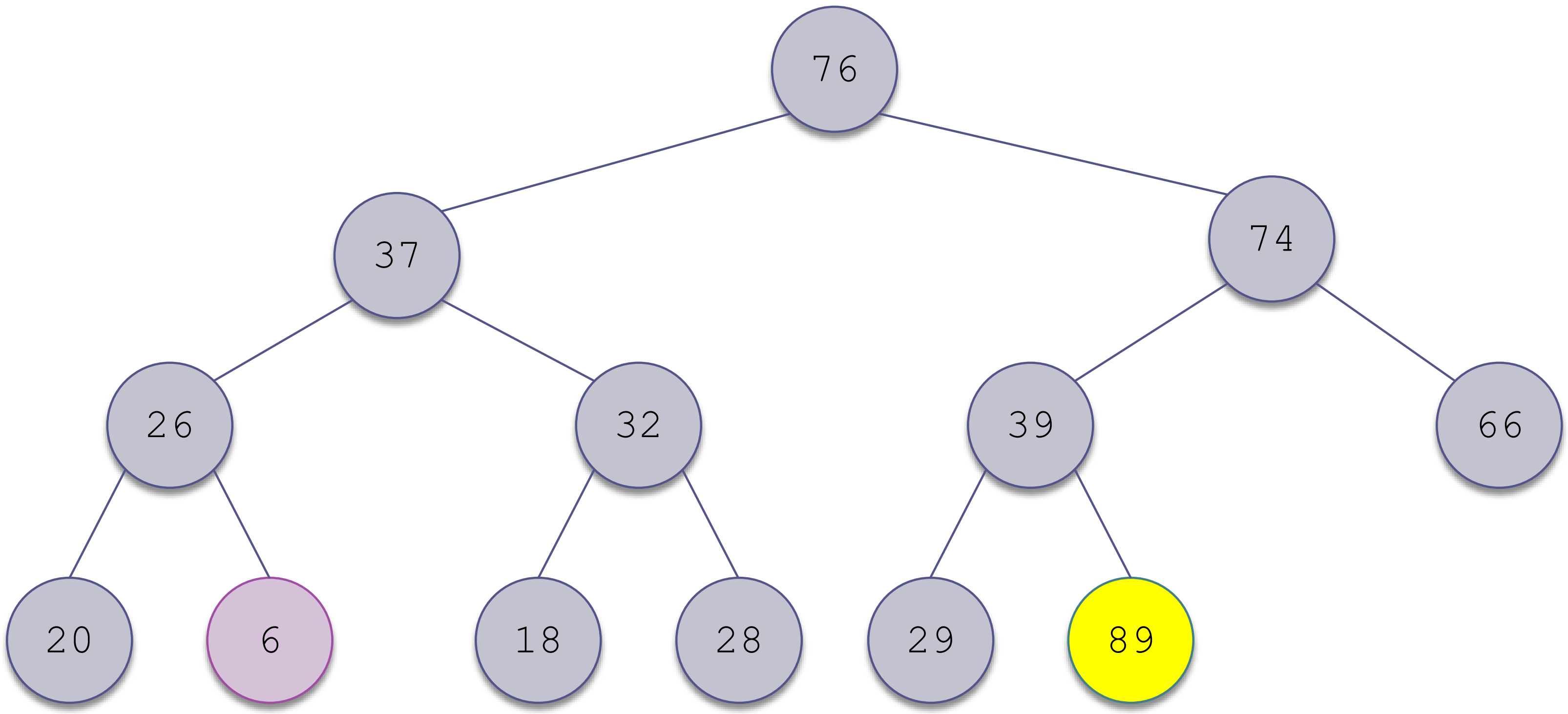
Trace of Heapsort (cont.)



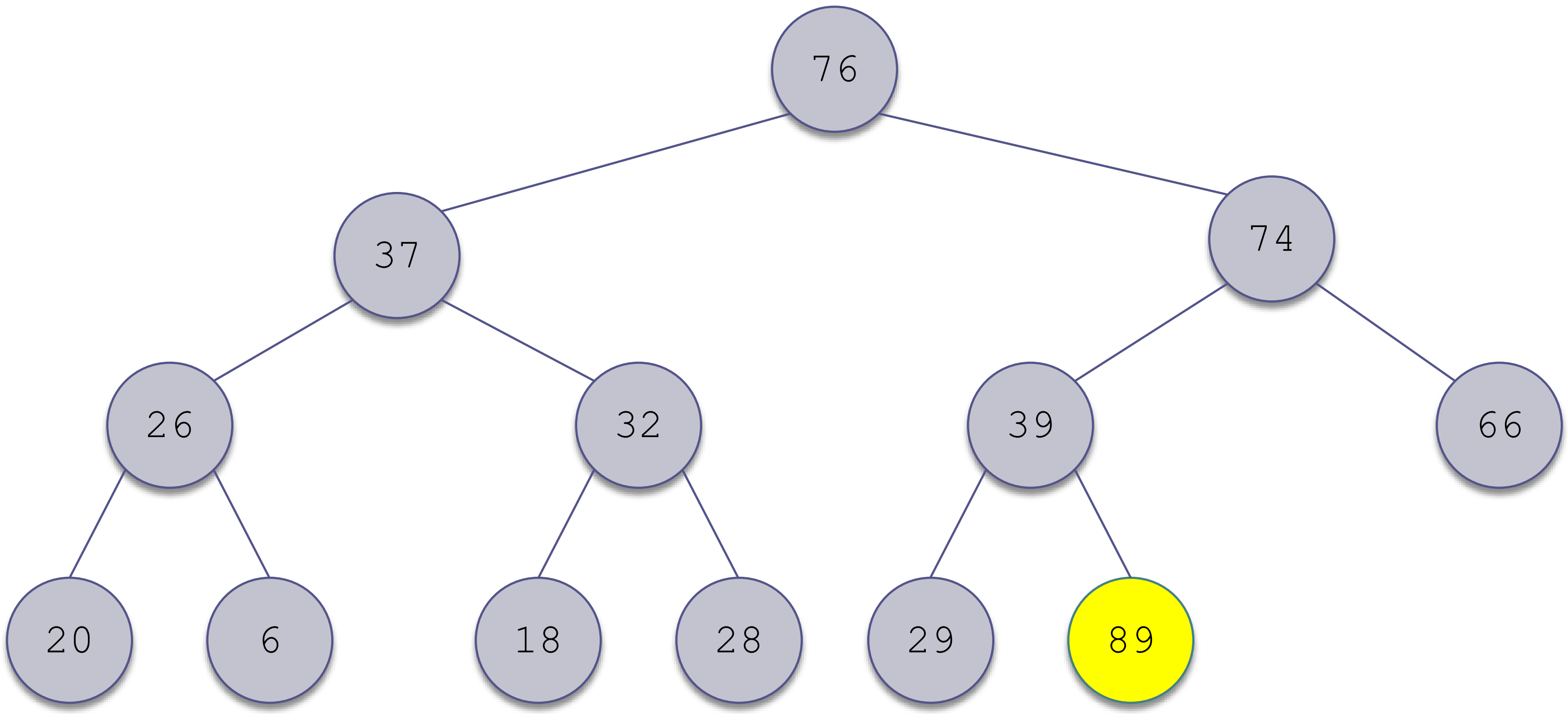
Trace of Heapsort (cont.)



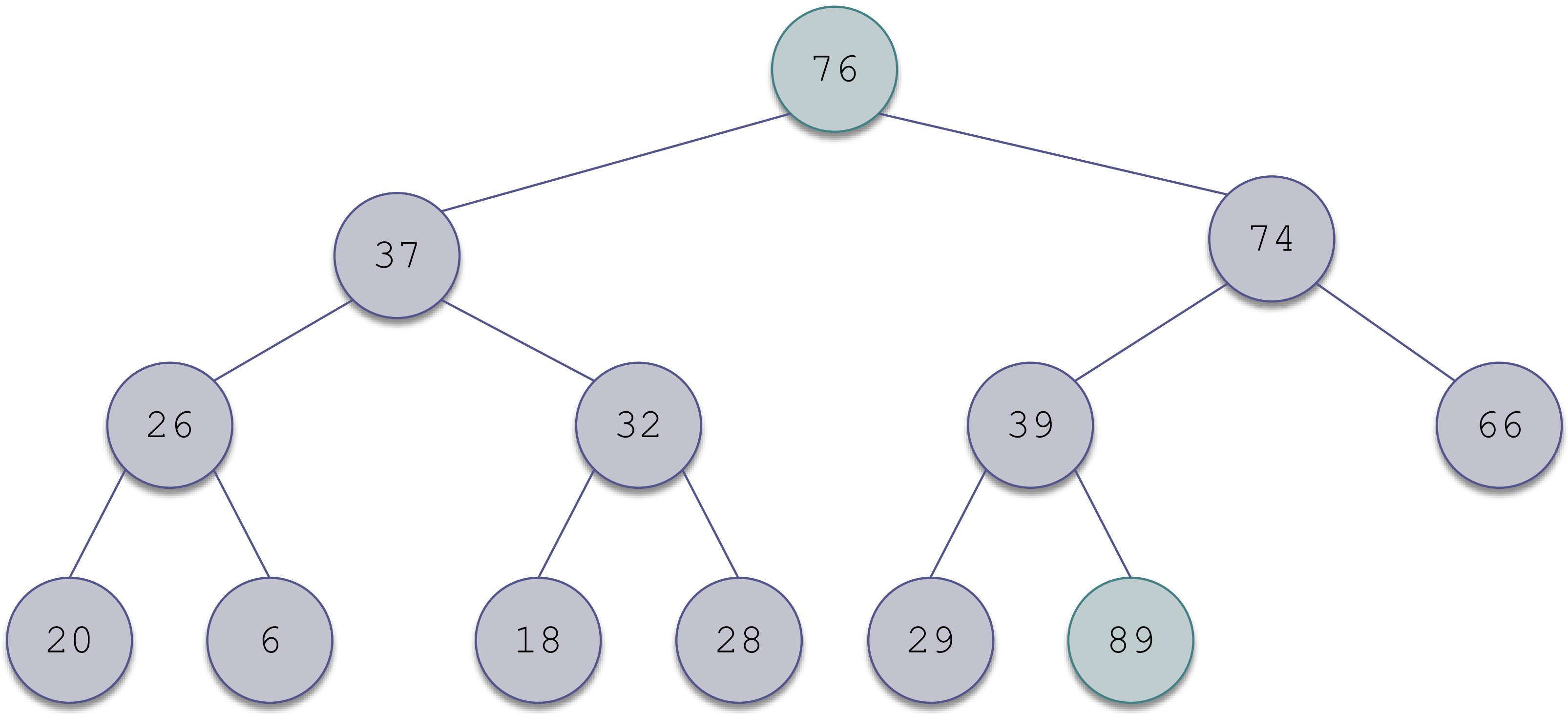
Trace of Heapsort (cont.)



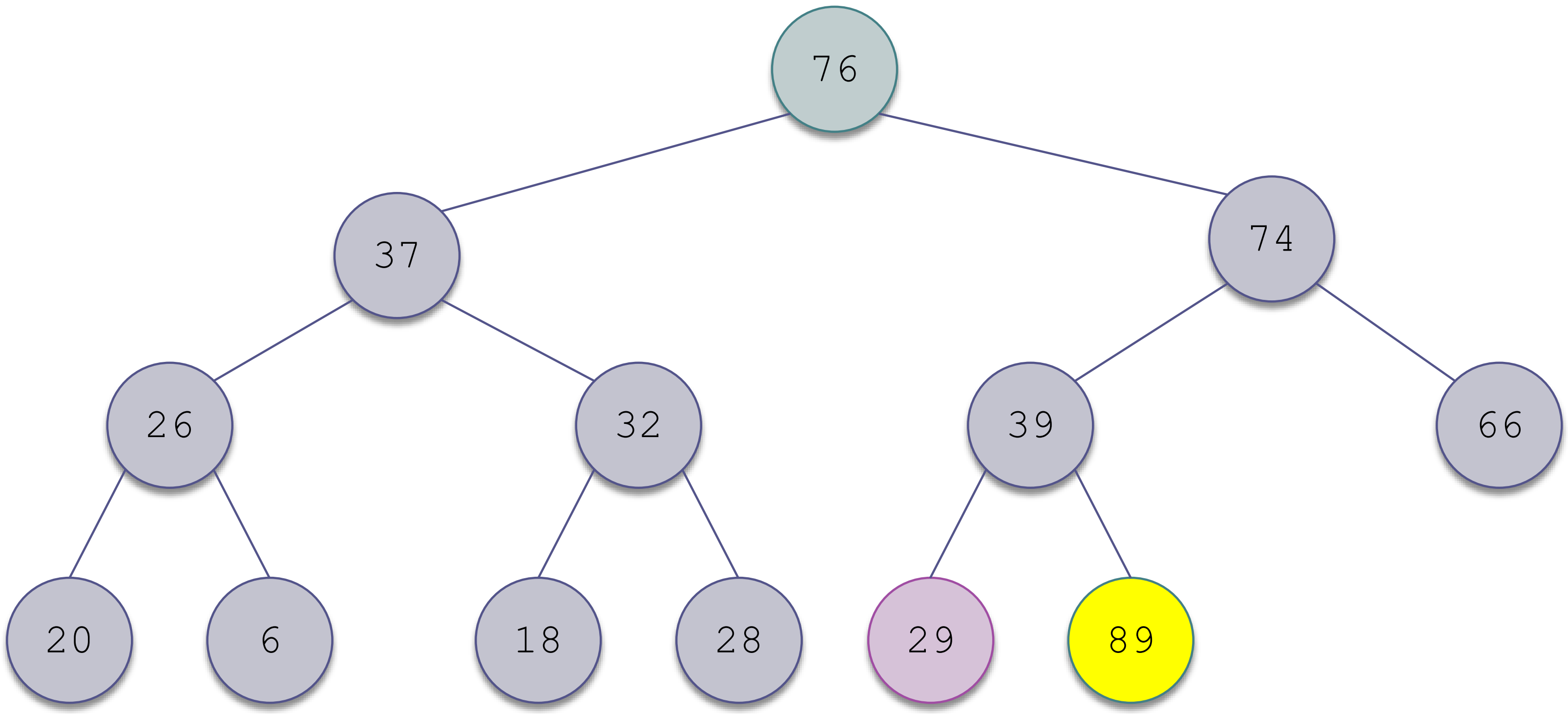
Trace of Heapsort (cont.)



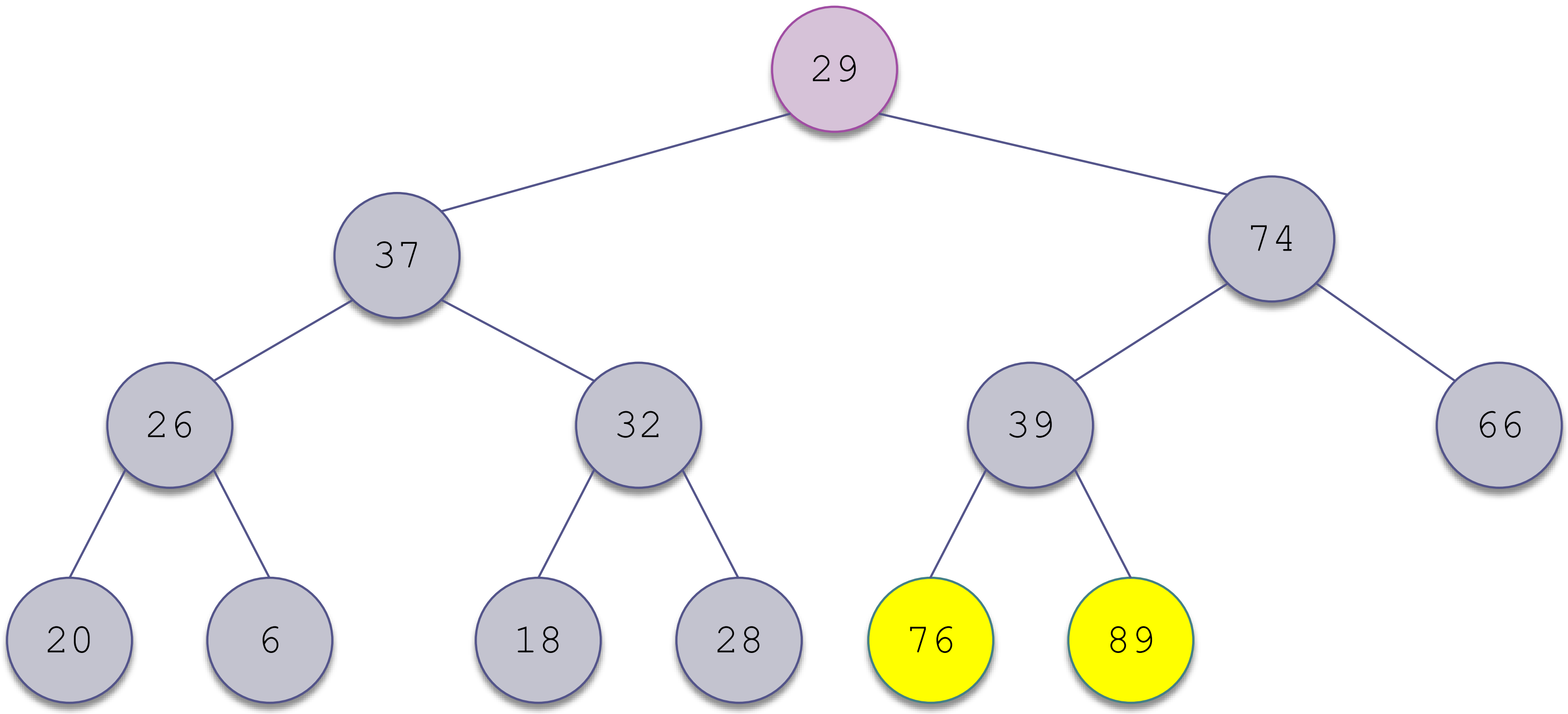
Trace of Heapsort (cont.)



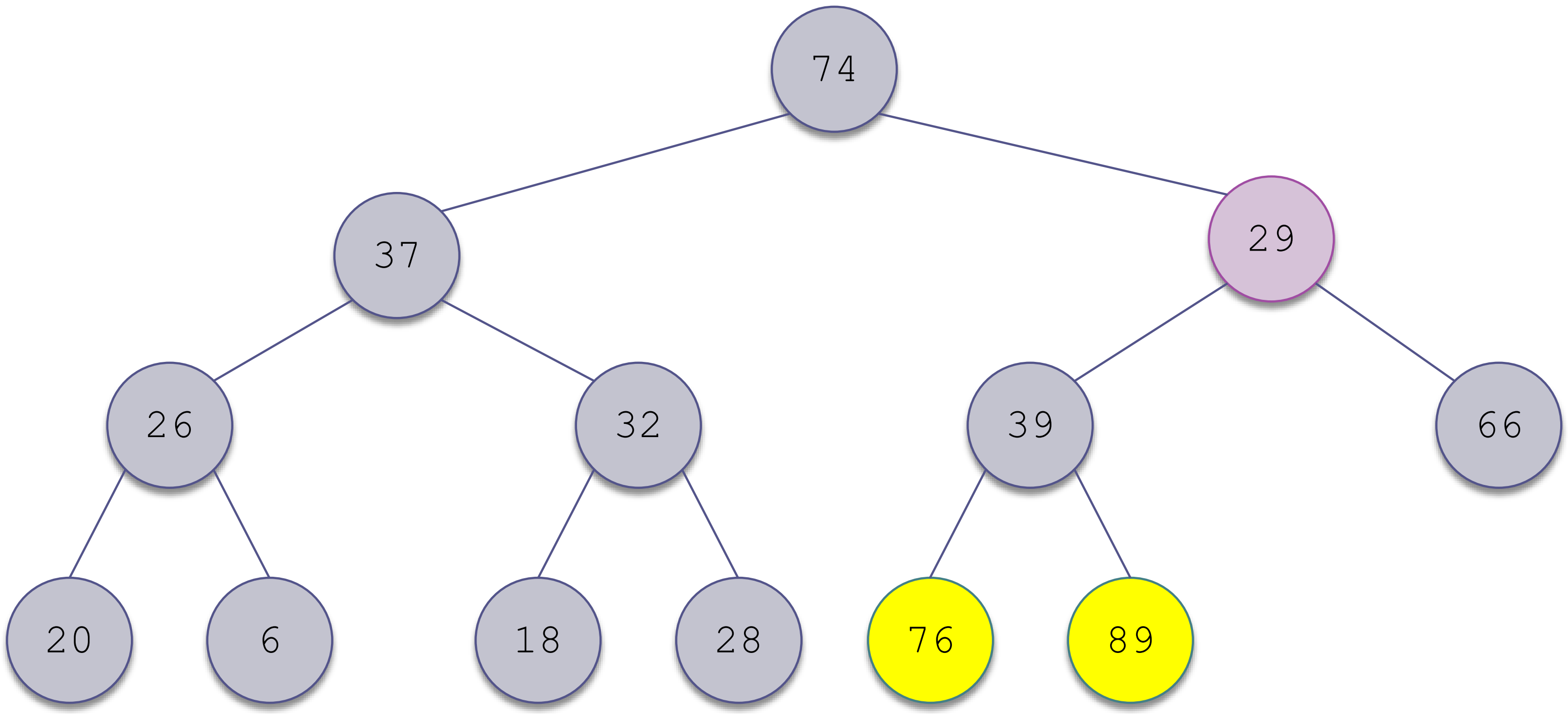
Trace of Heapsort (cont.)



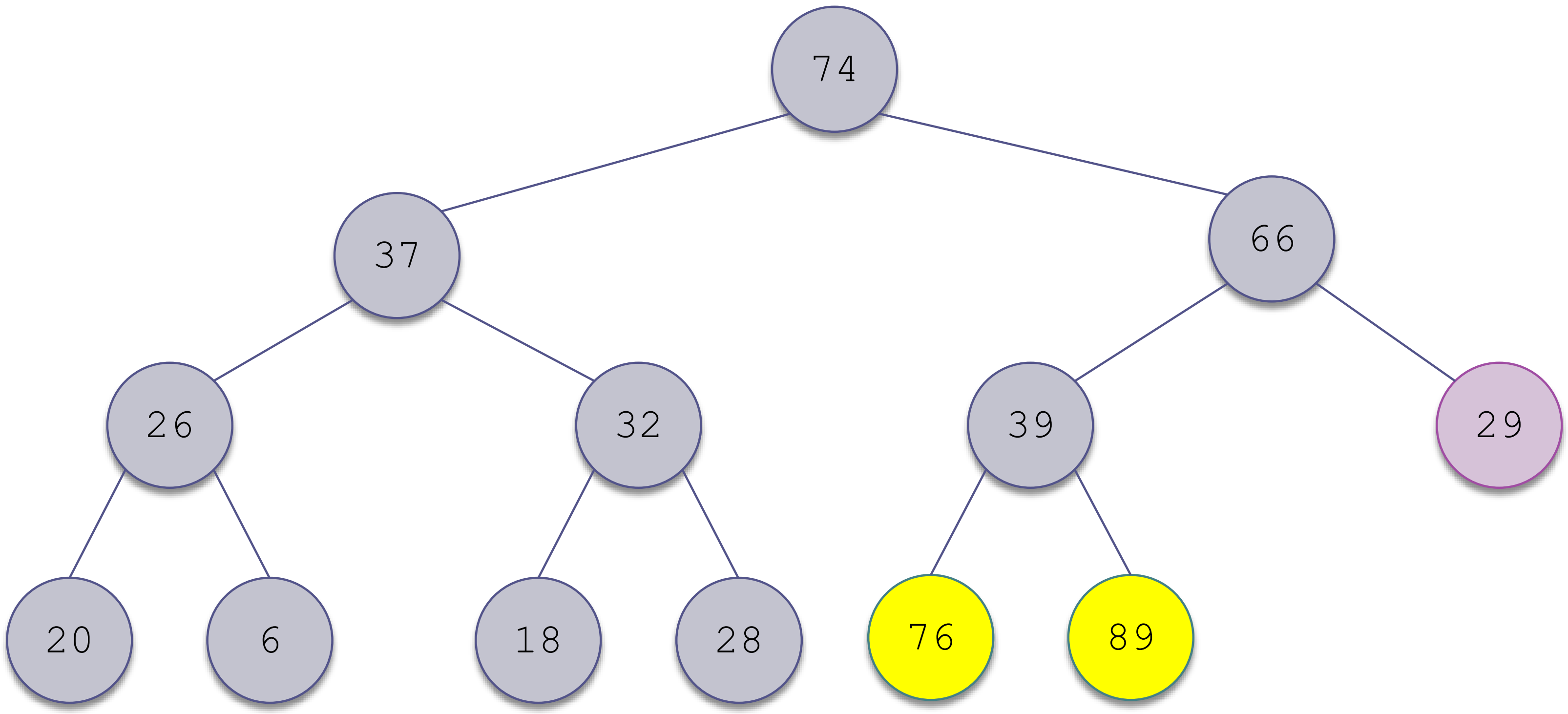
Trace of Heapsort (cont.)



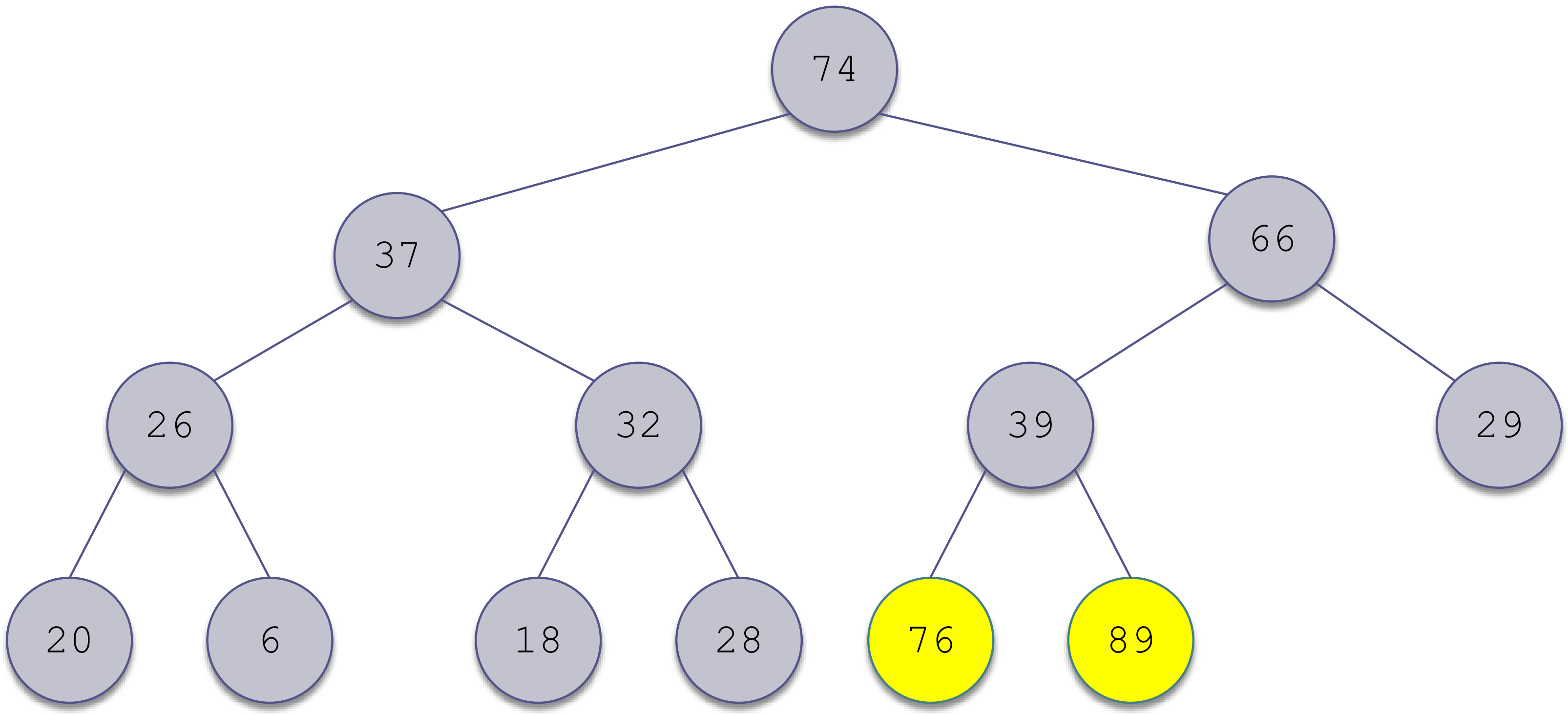
Trace of Heapsort (cont.)



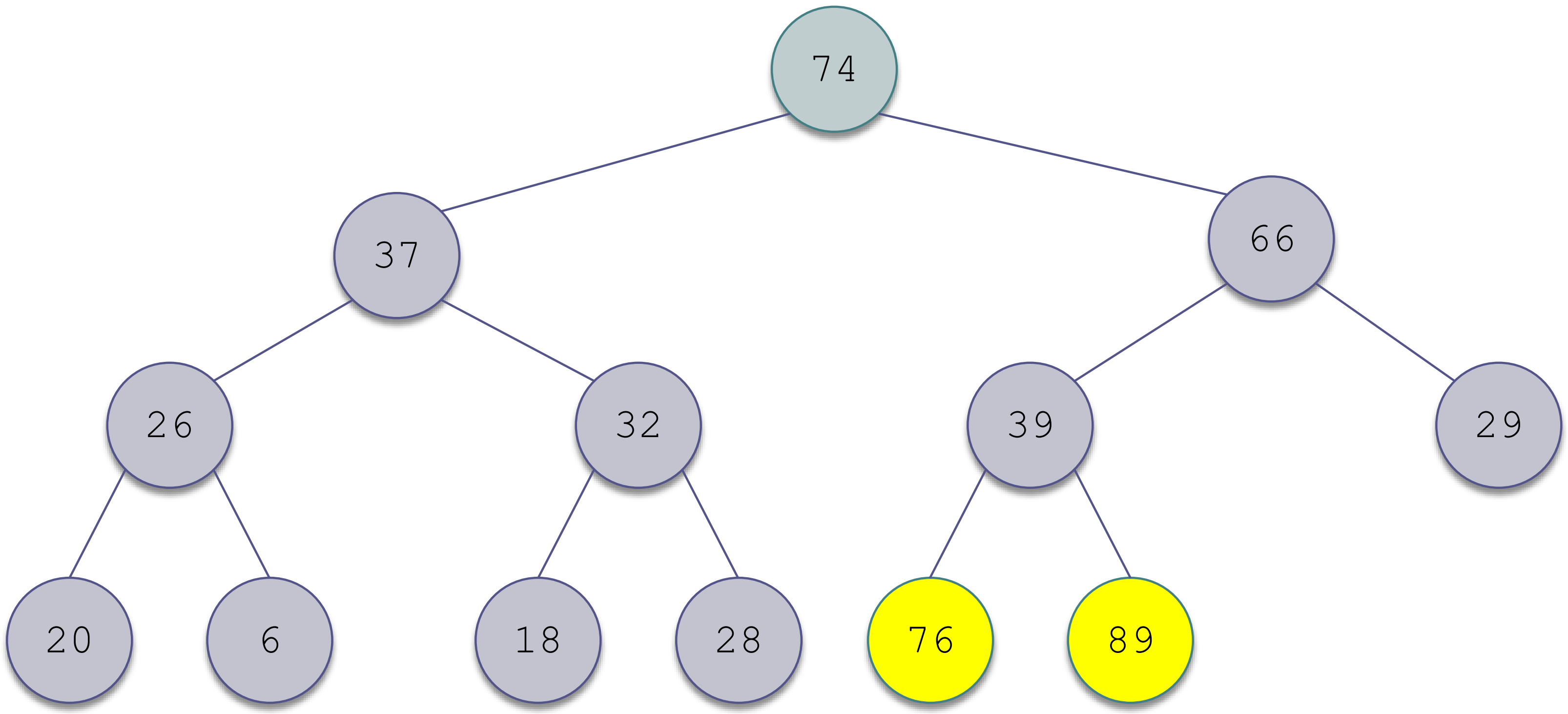
Trace of Heapsort (cont.)



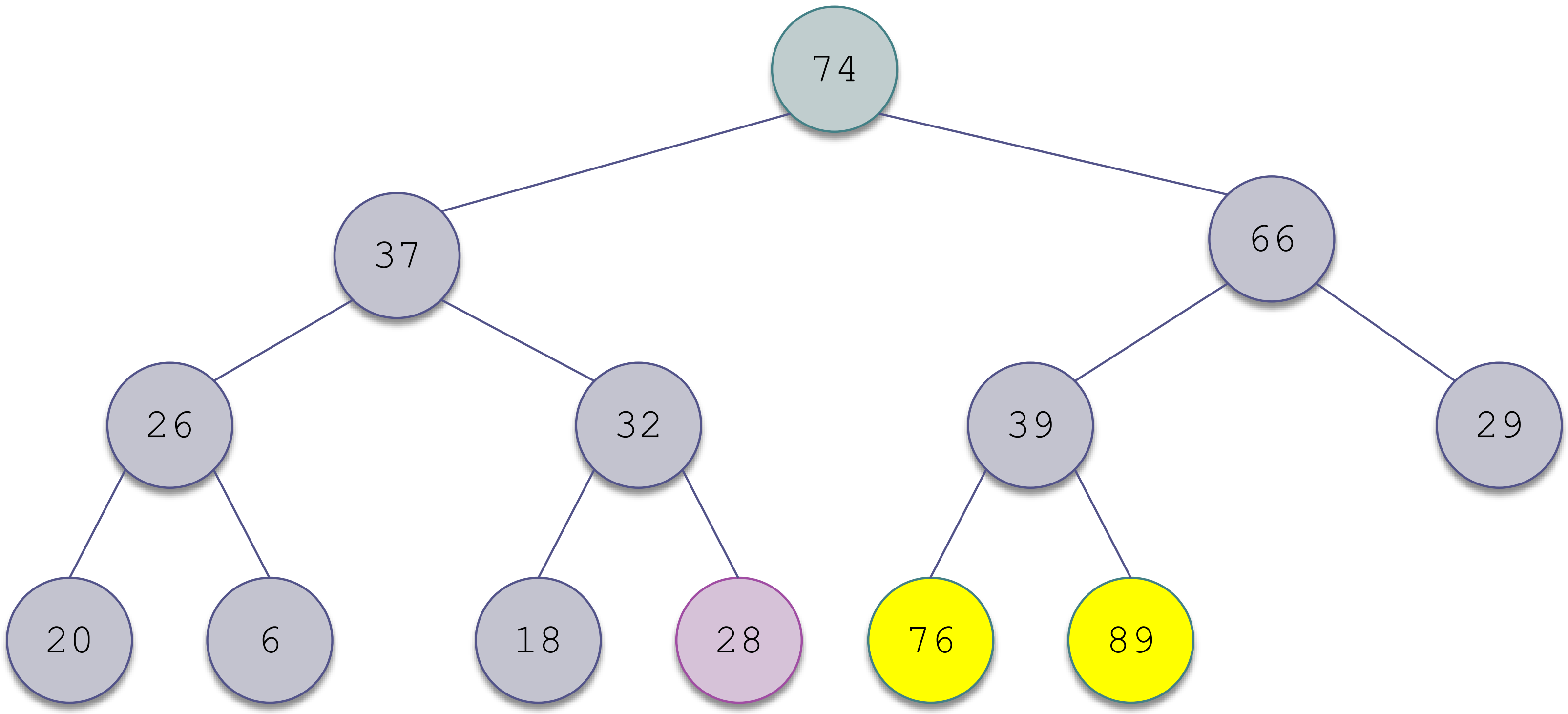
Trace of Heapsort (cont.)



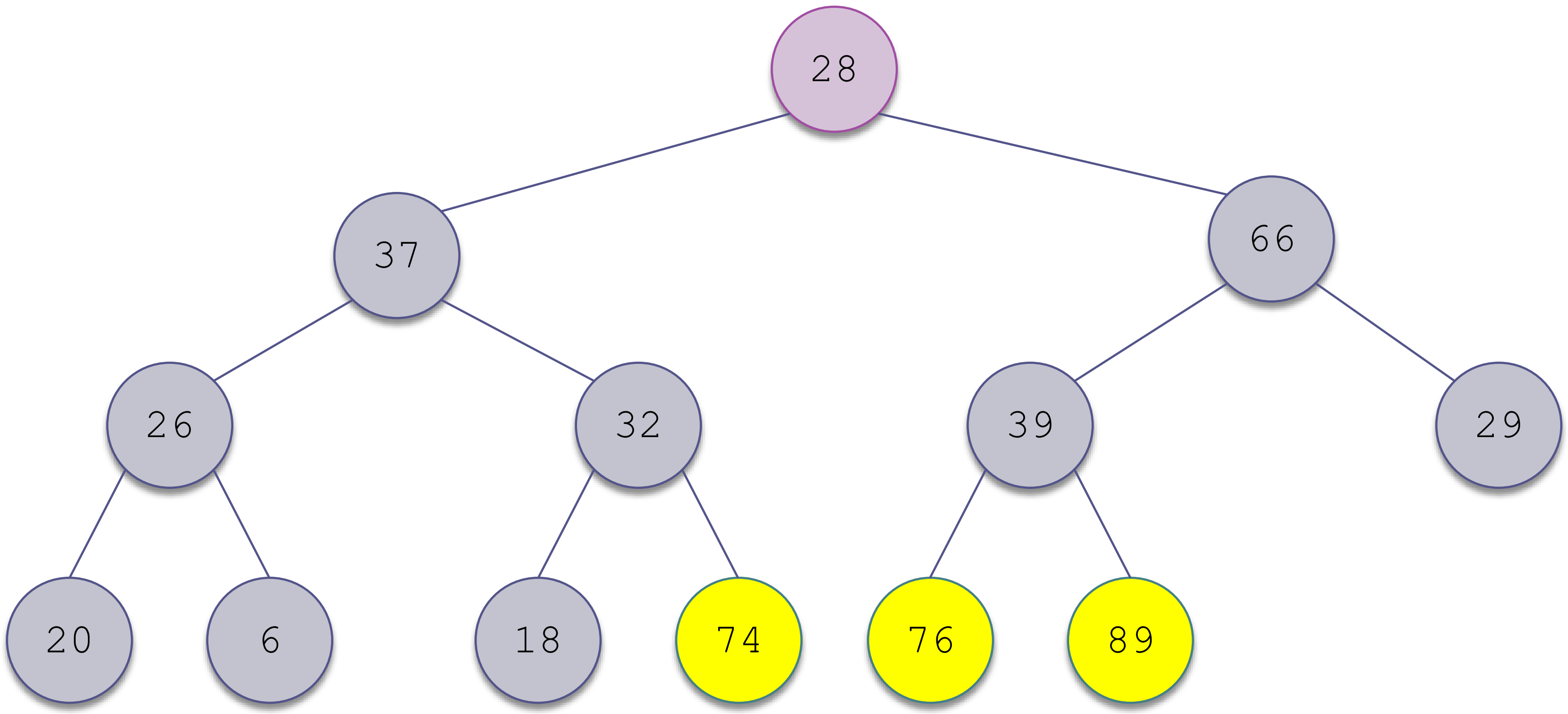
Trace of Heapsort (cont.)



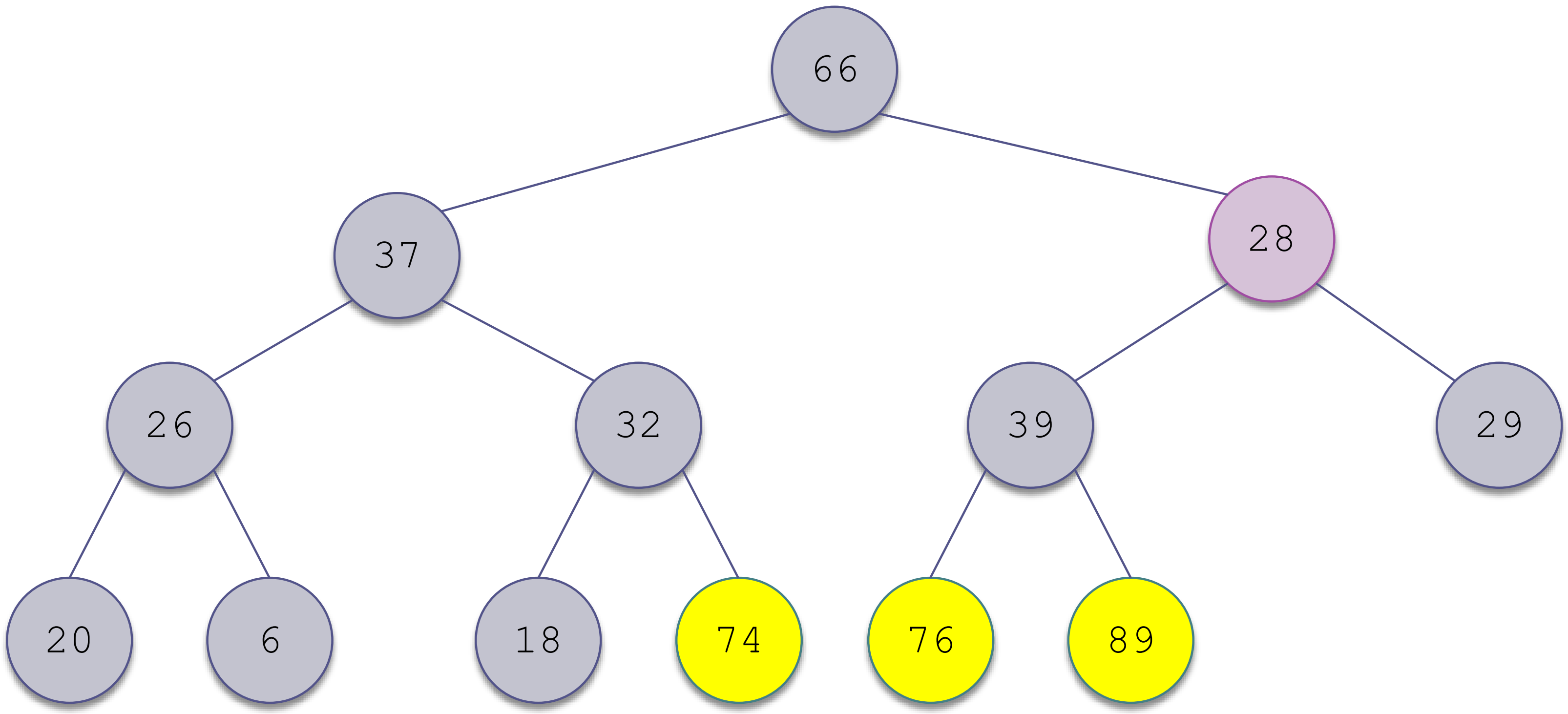
Trace of Heapsort (cont.)



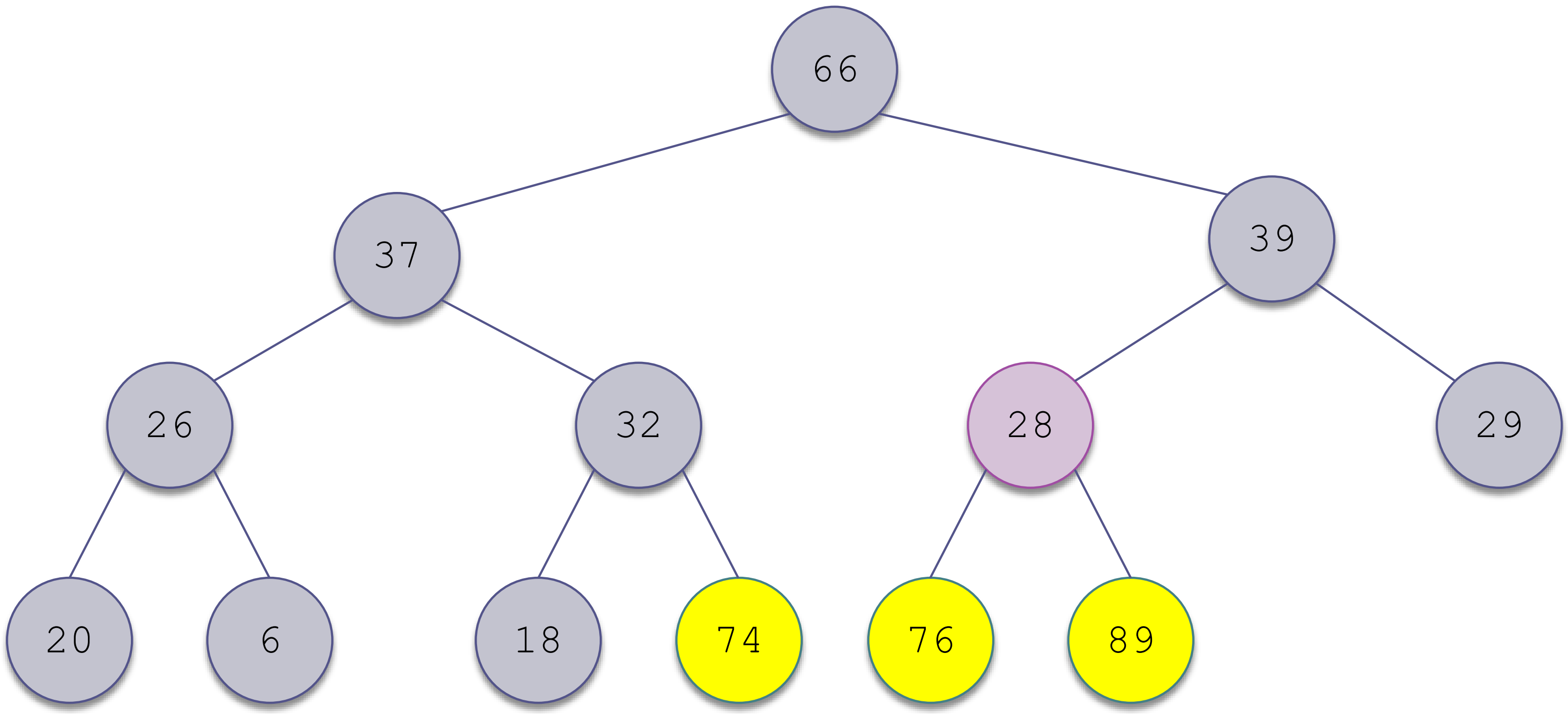
Trace of Heapsort (cont.)



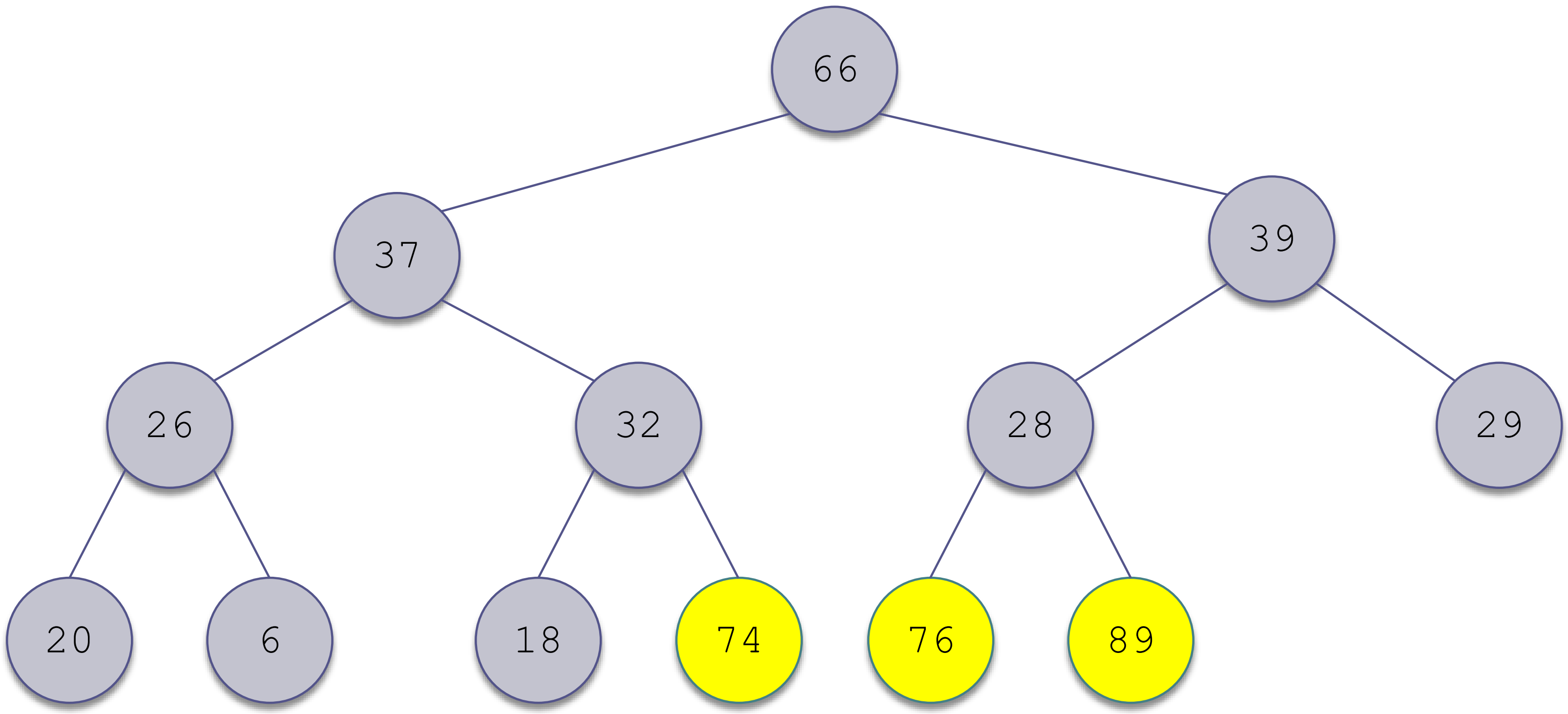
Trace of Heapsort (cont.)



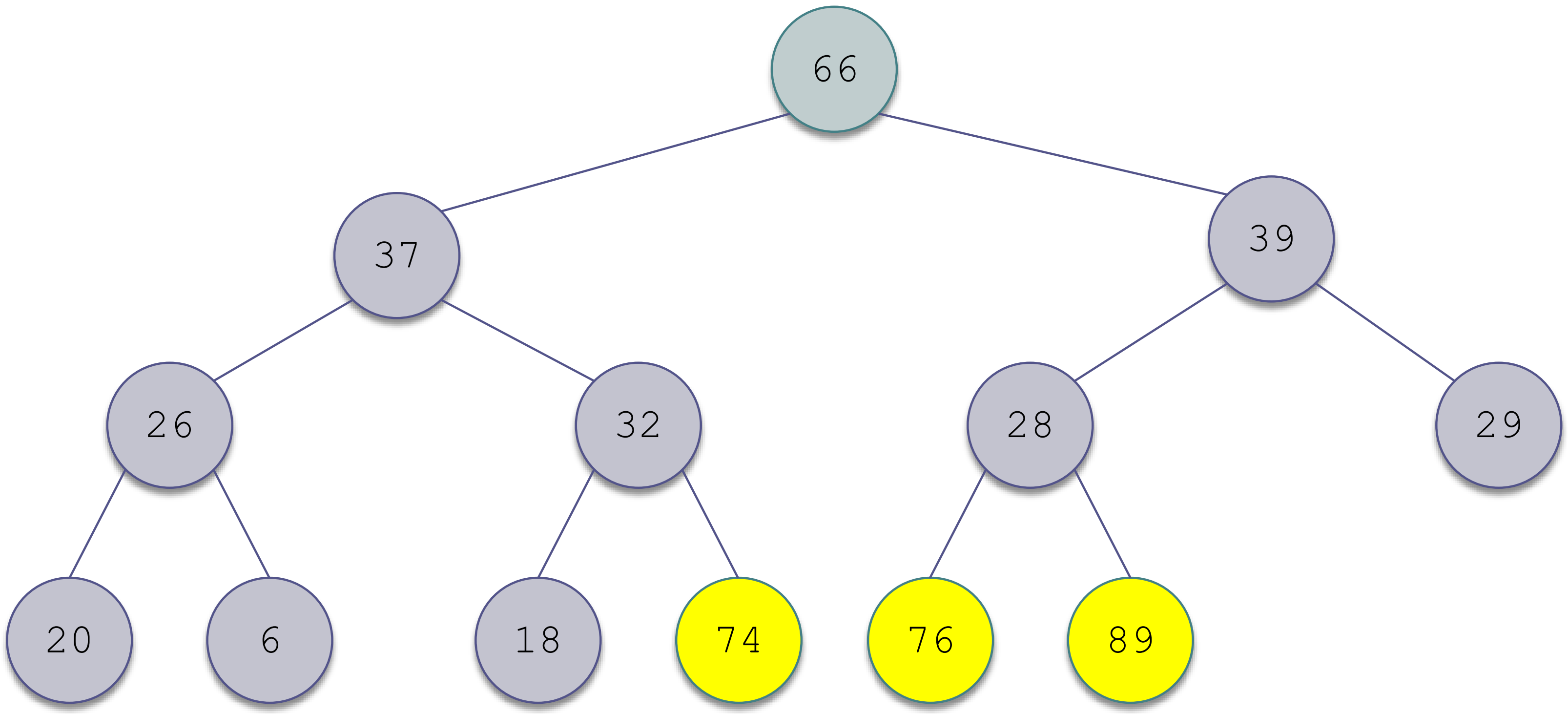
Trace of Heapsort (cont.)



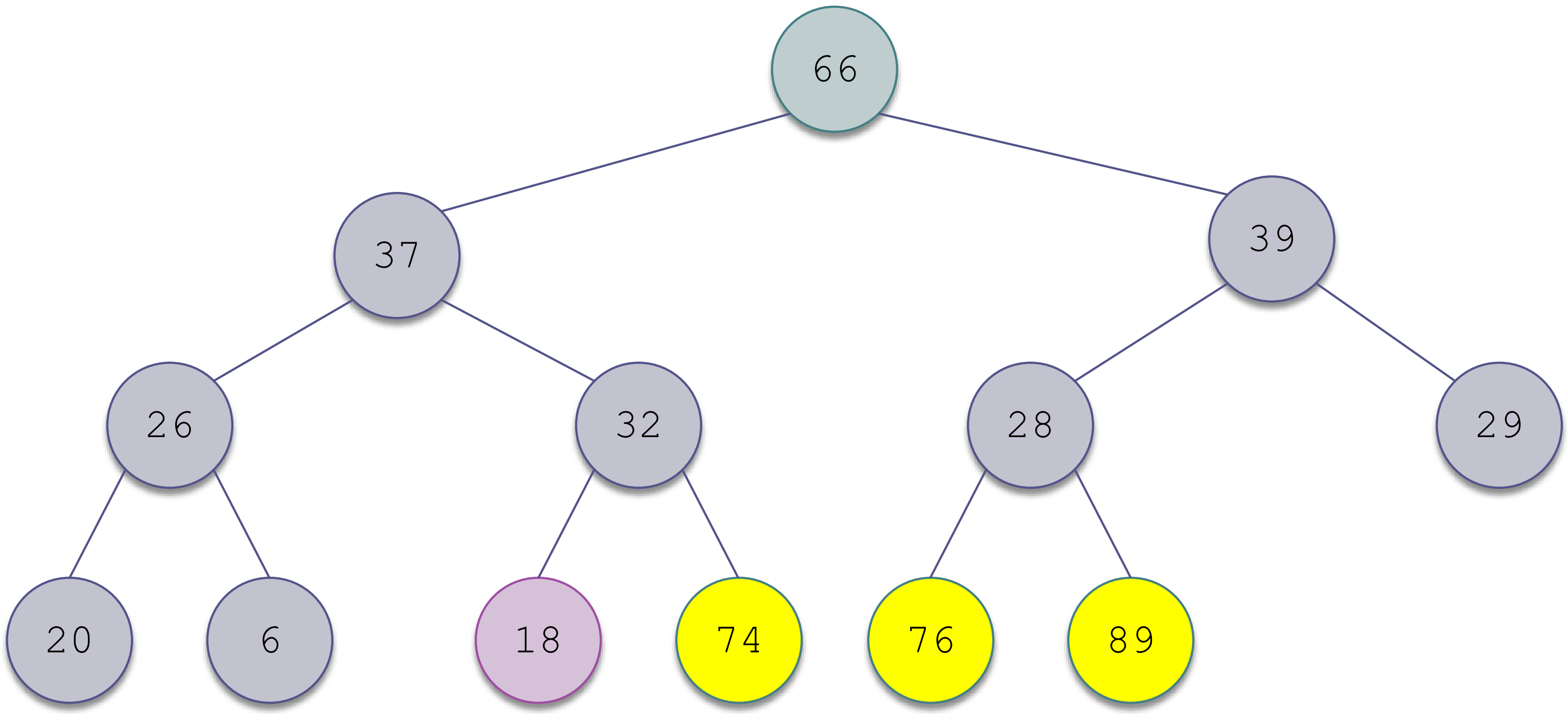
Trace of Heapsort (cont.)



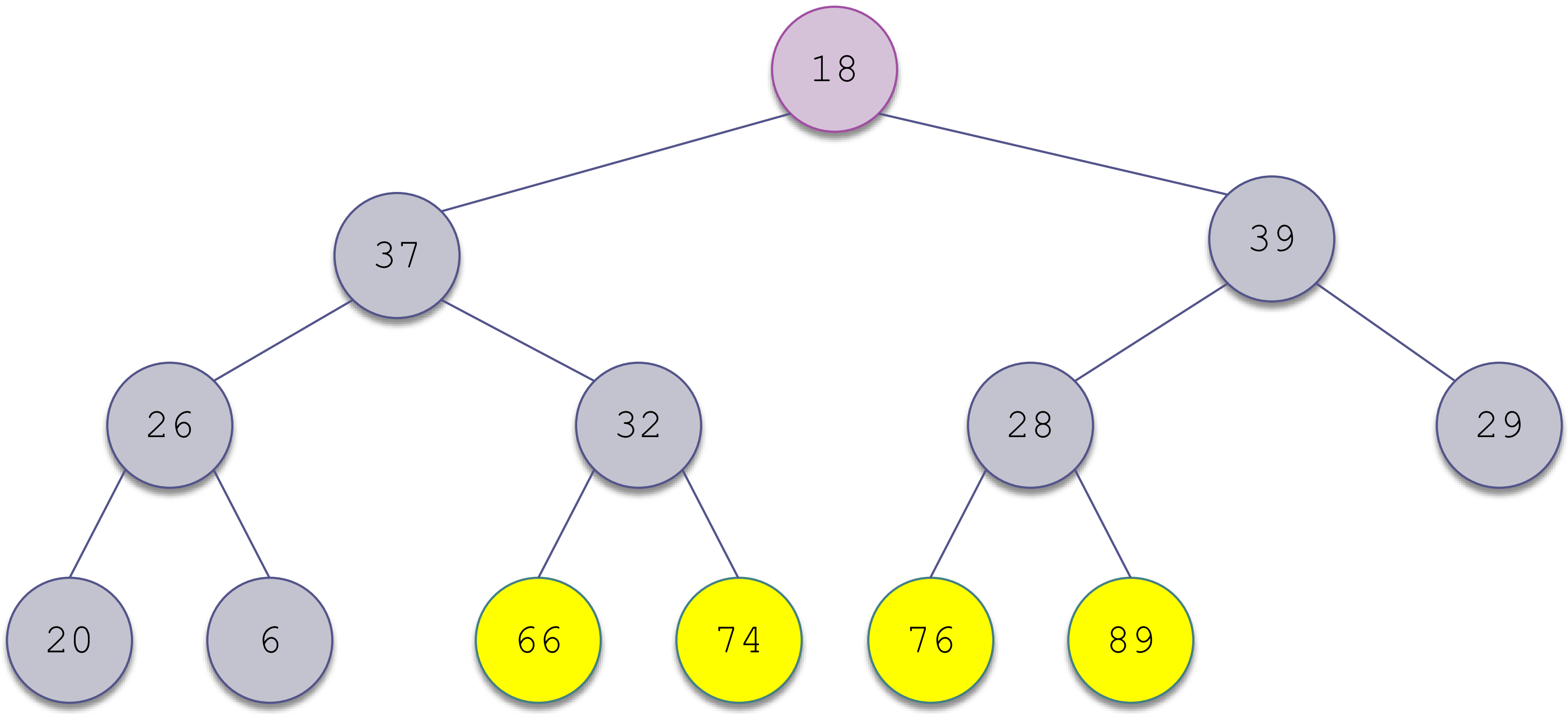
Trace of Heapsort (cont.)



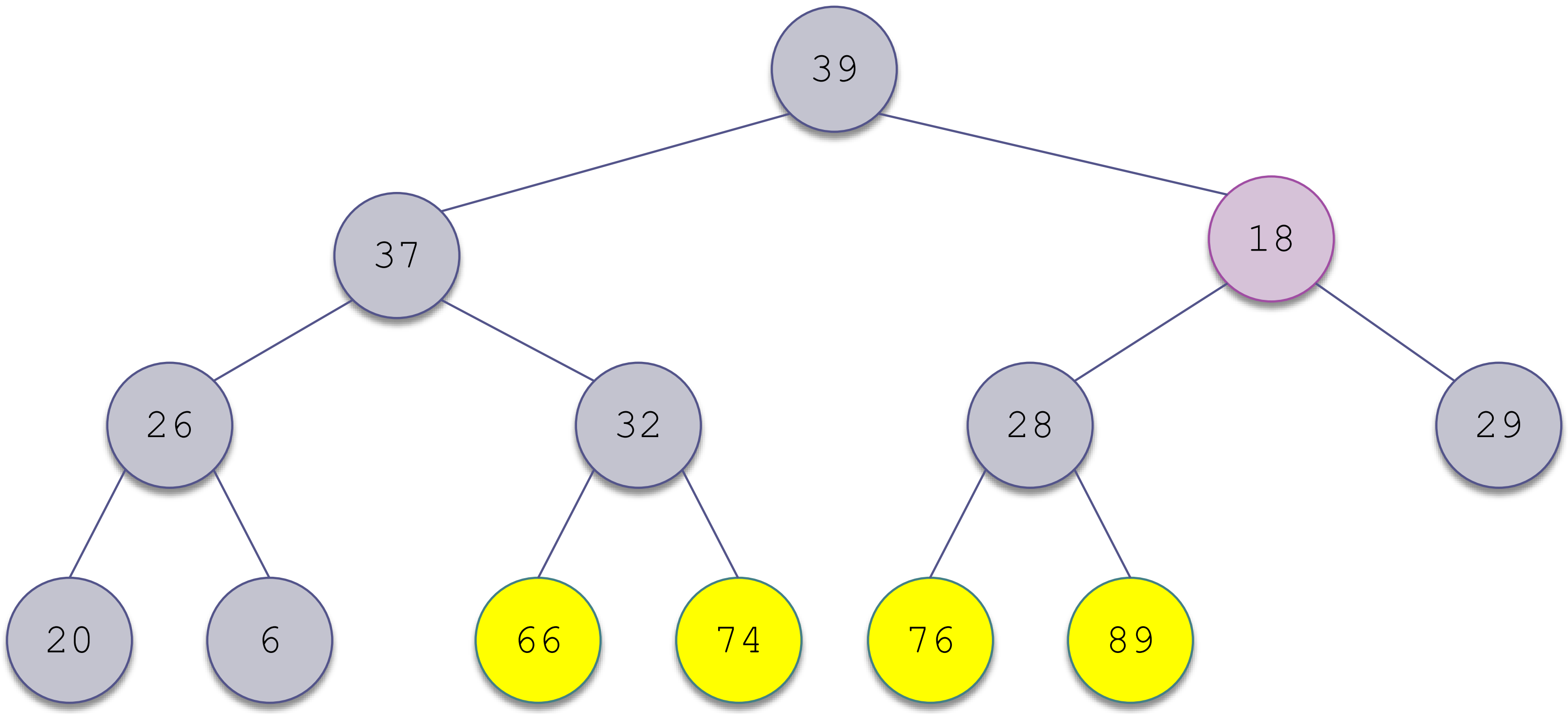
Trace of Heapsort (cont.)



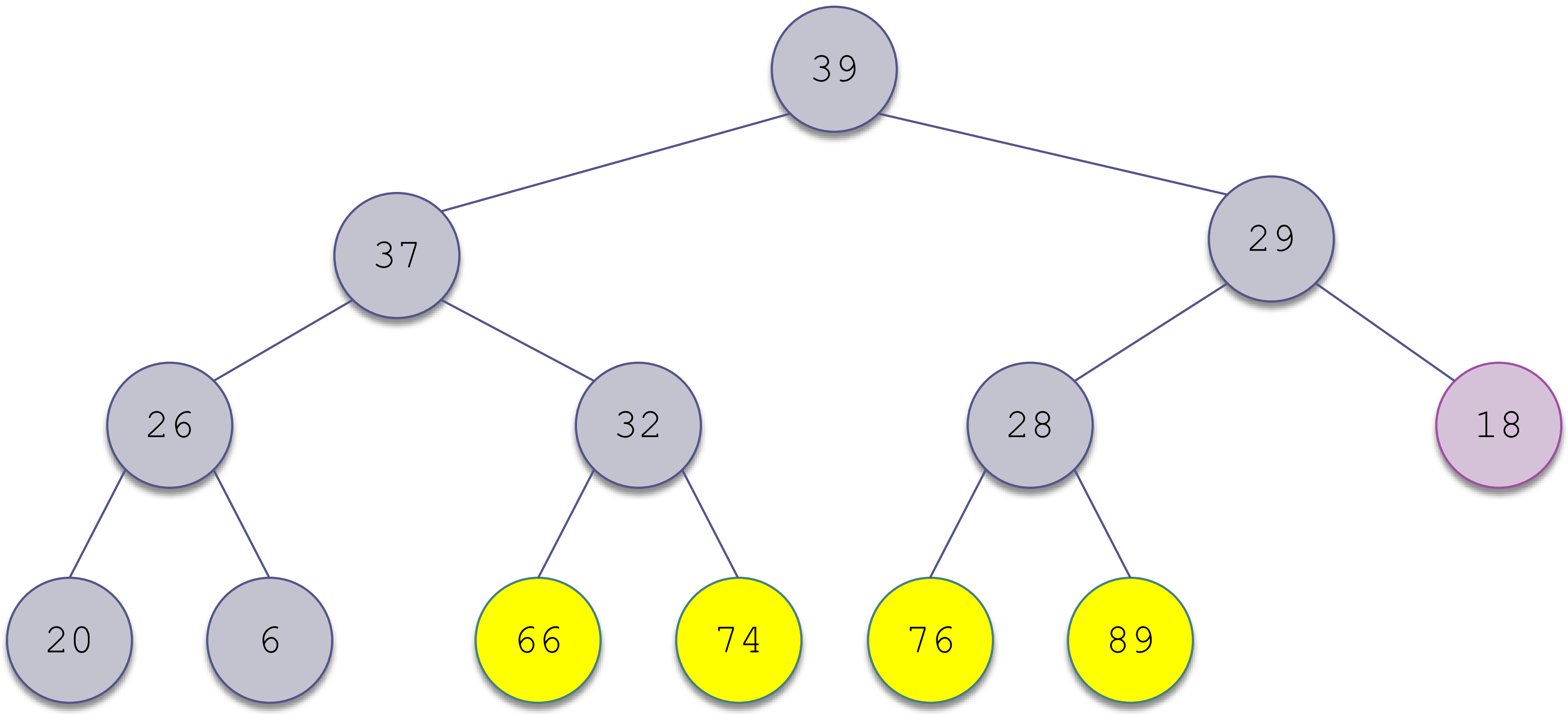
Trace of Heapsort (cont.)



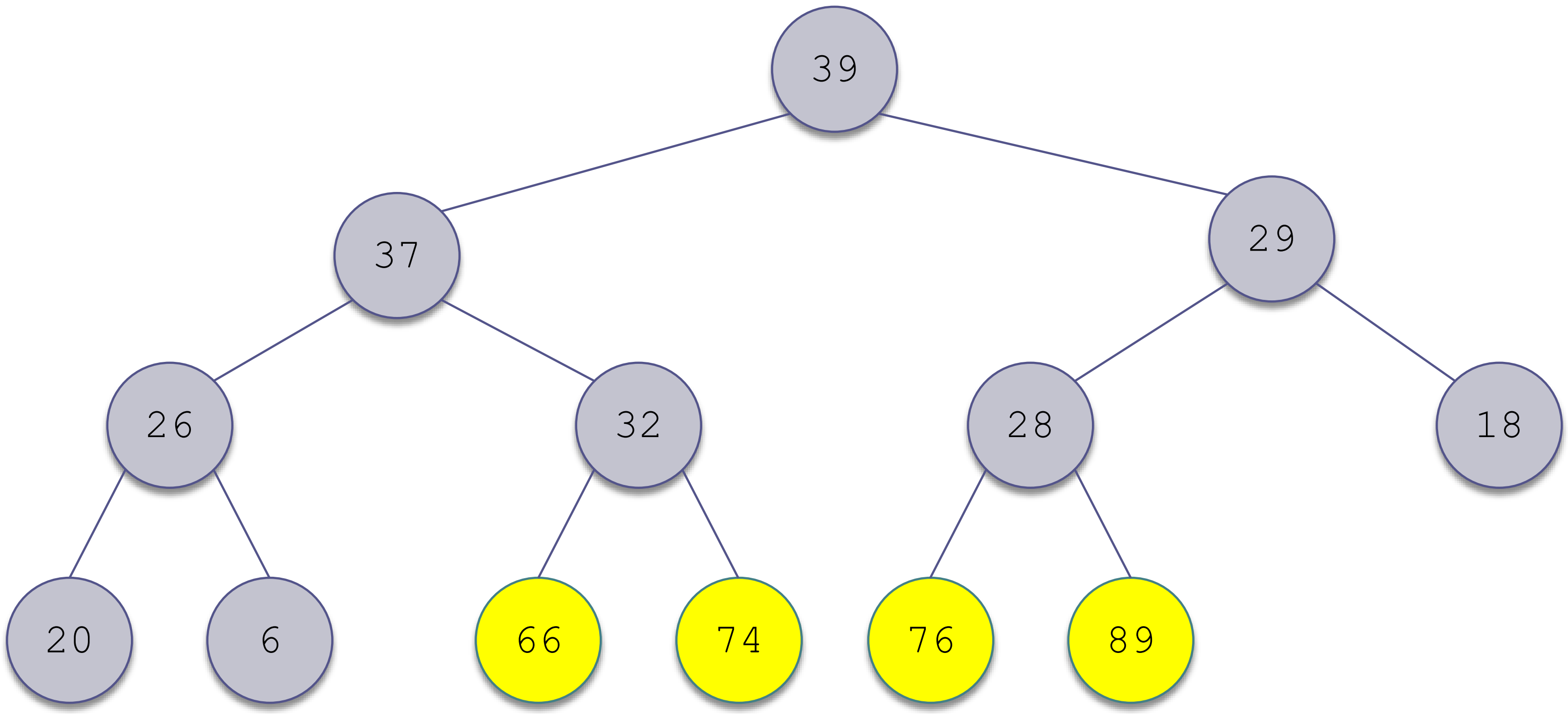
Trace of Heapsort (cont.)



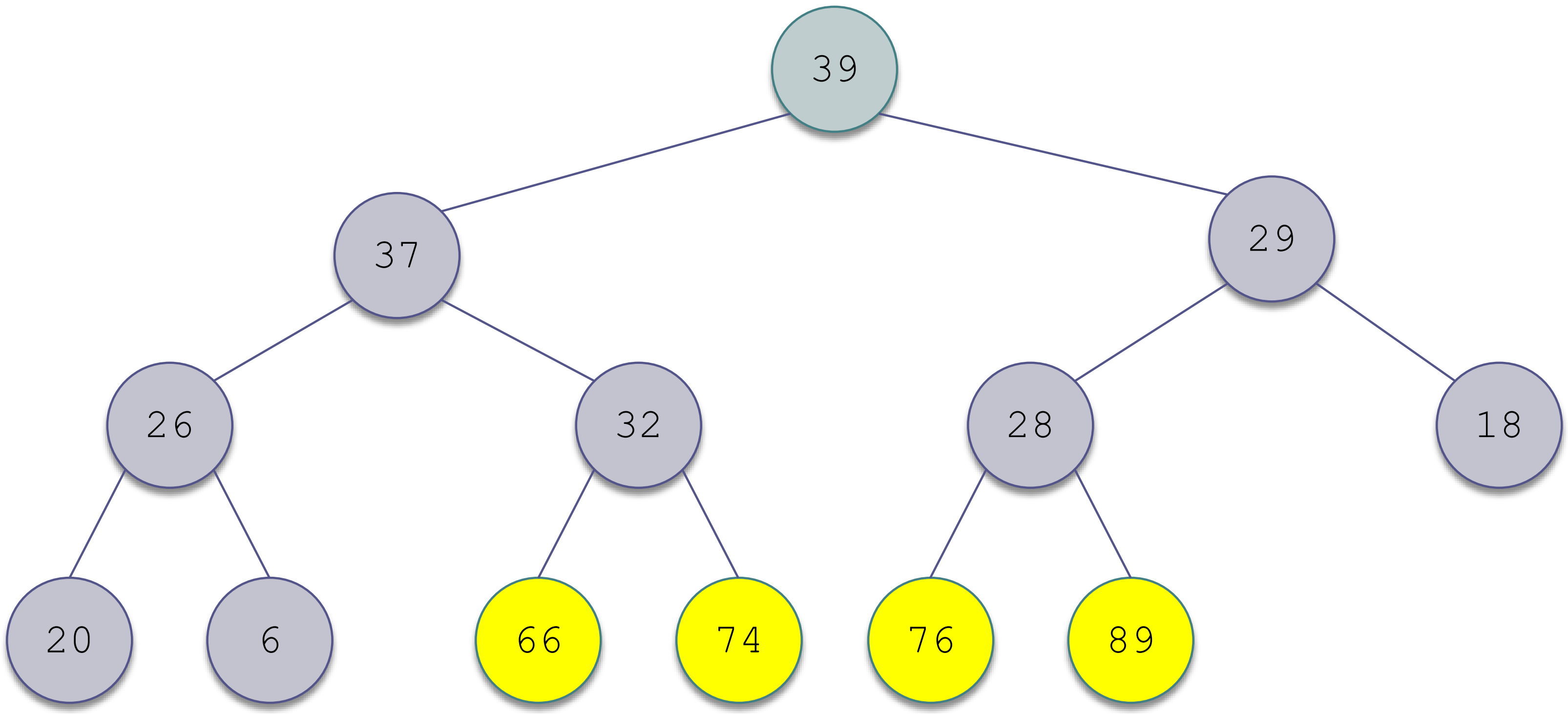
Trace of Heapsort (cont.)



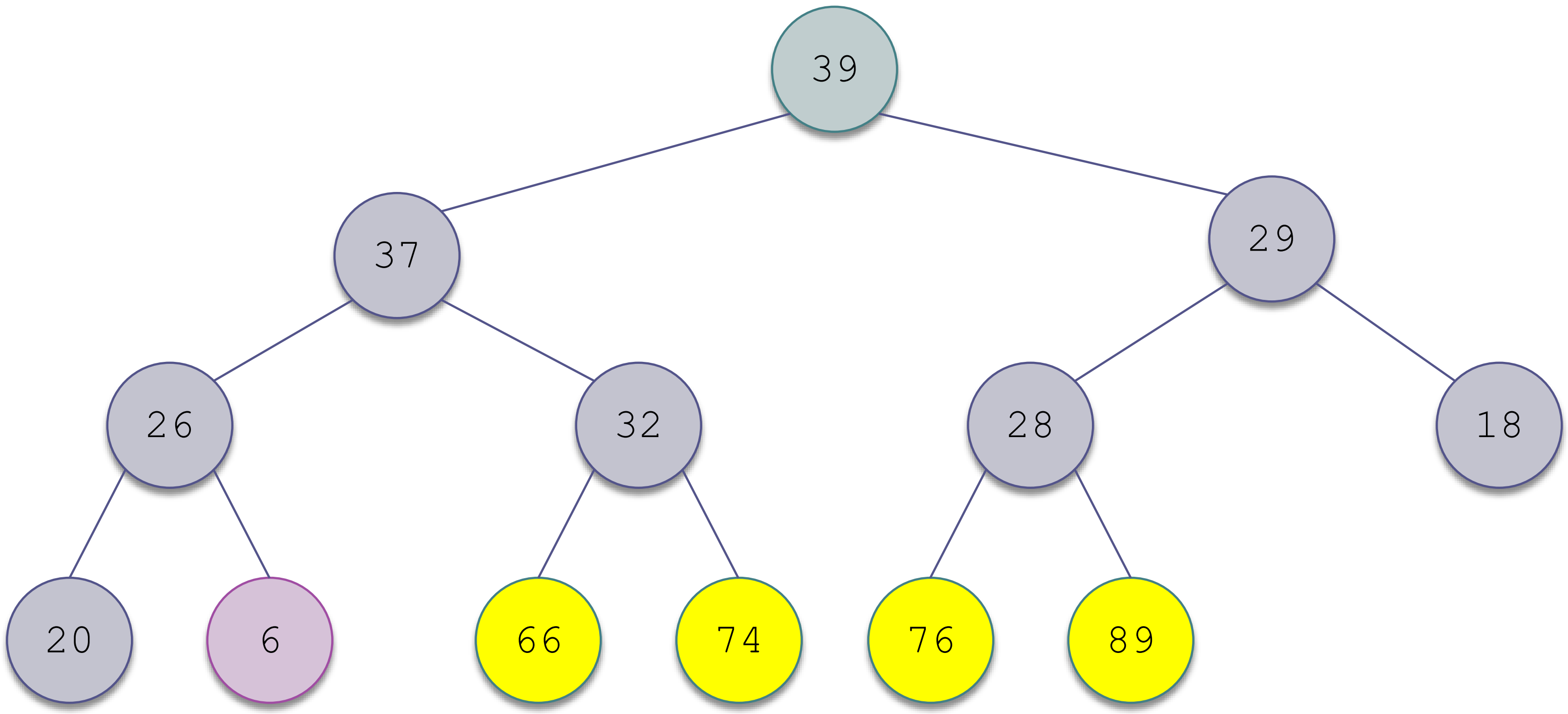
Trace of Heapsort (cont.)



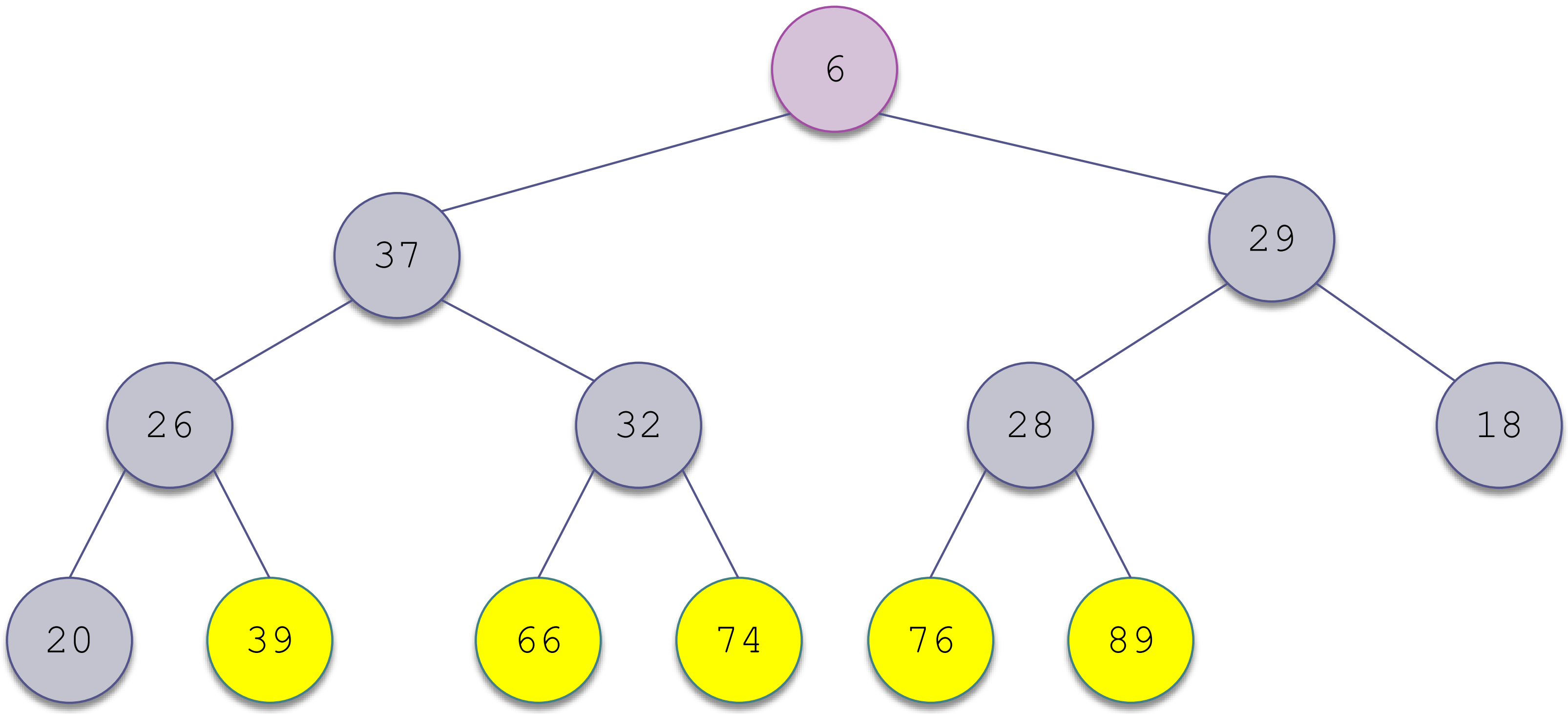
Trace of Heapsort (cont.)



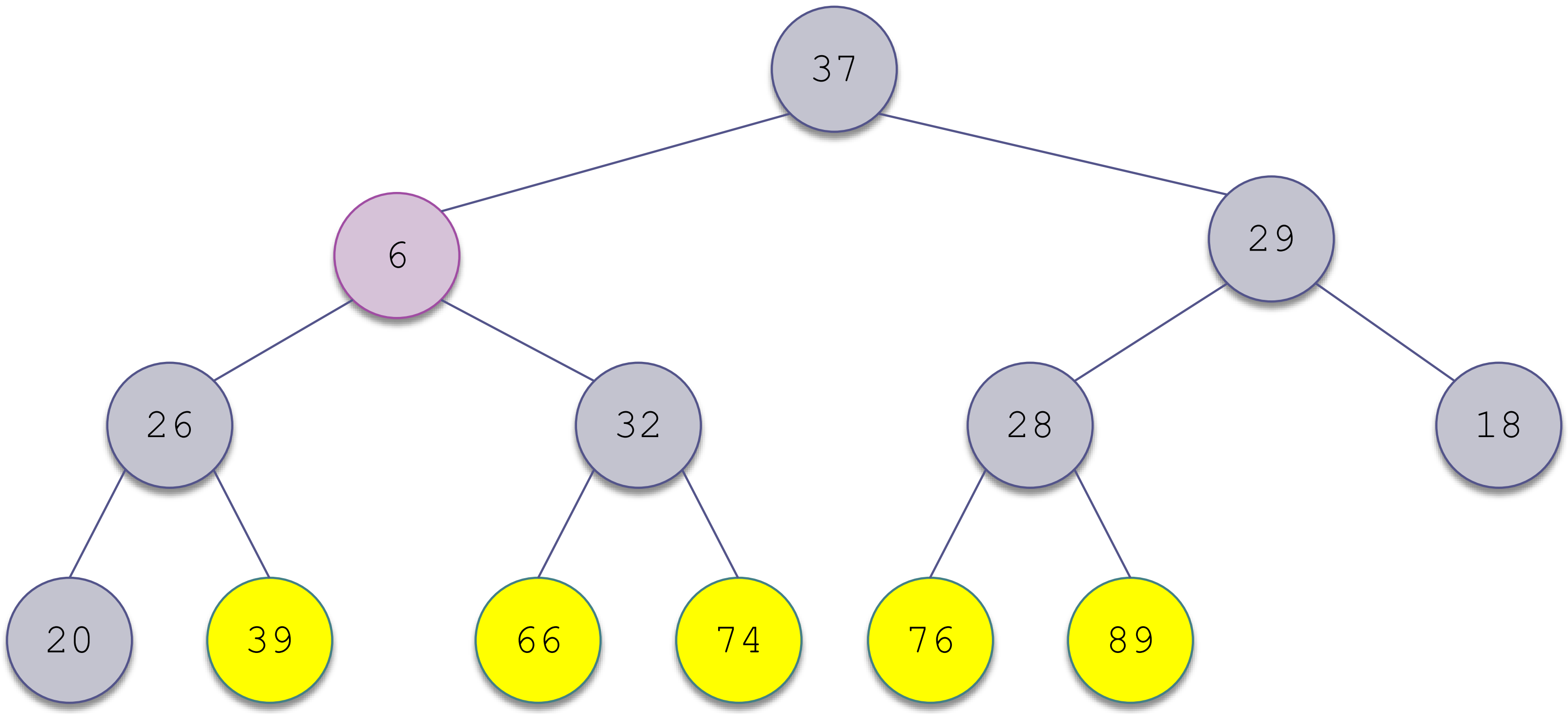
Trace of Heapsort (cont.)



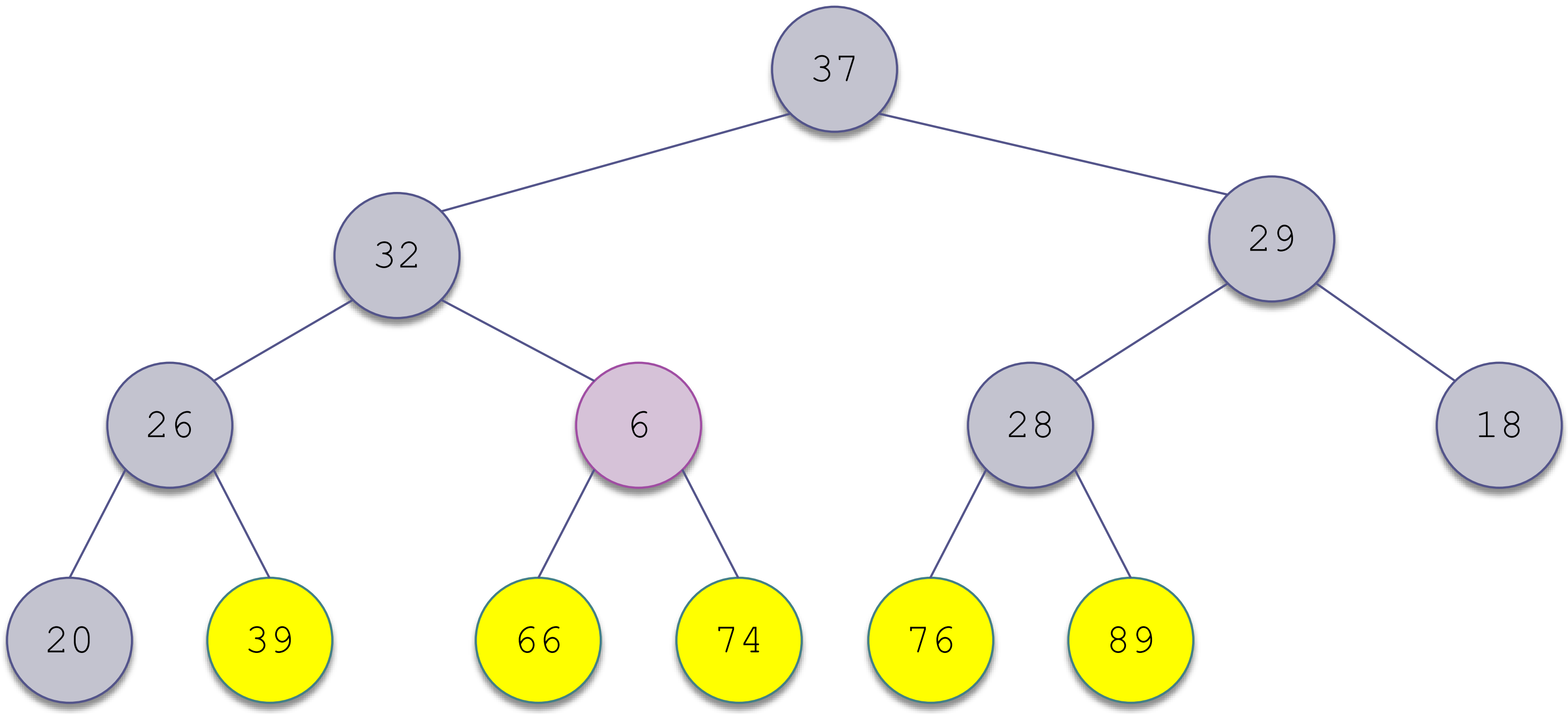
Trace of Heapsort (cont.)



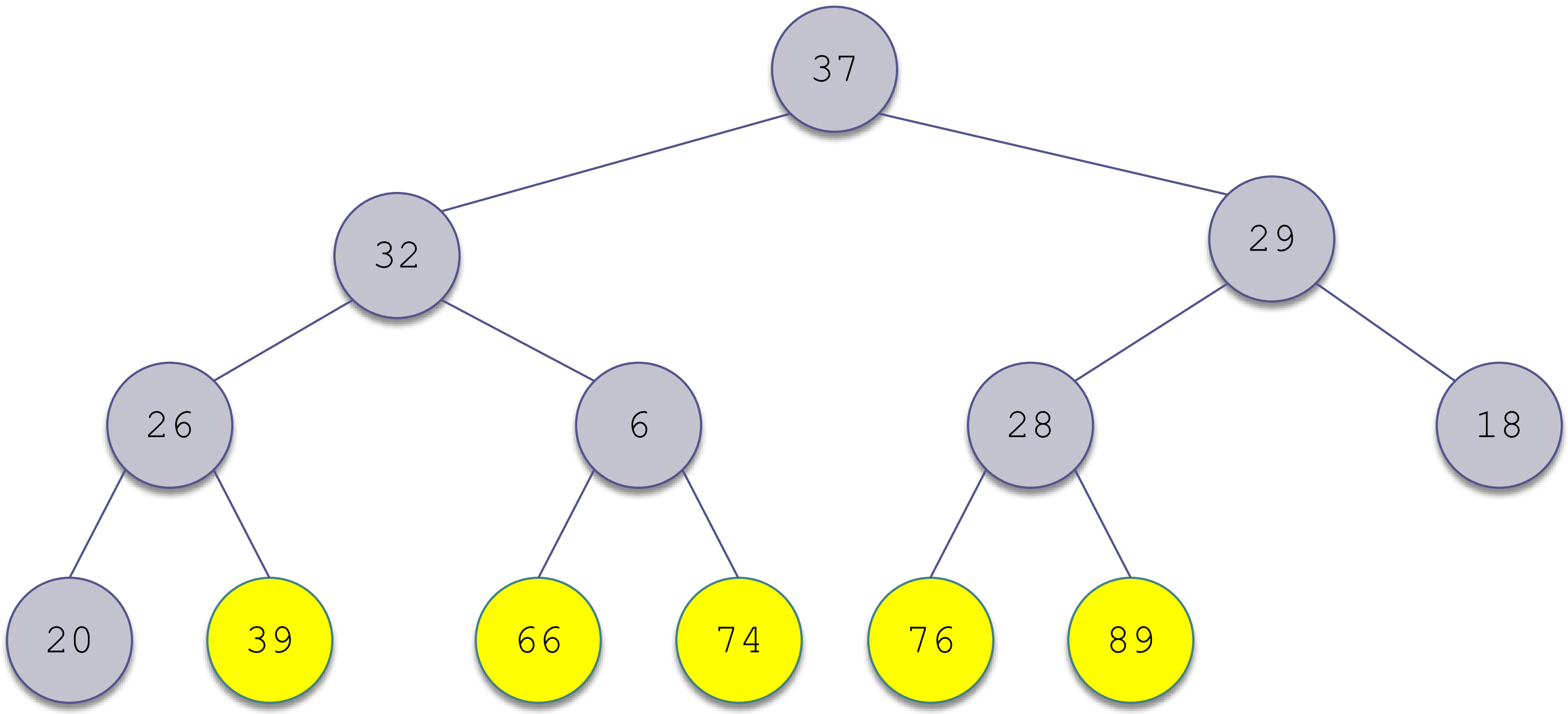
Trace of Heapsort (cont.)



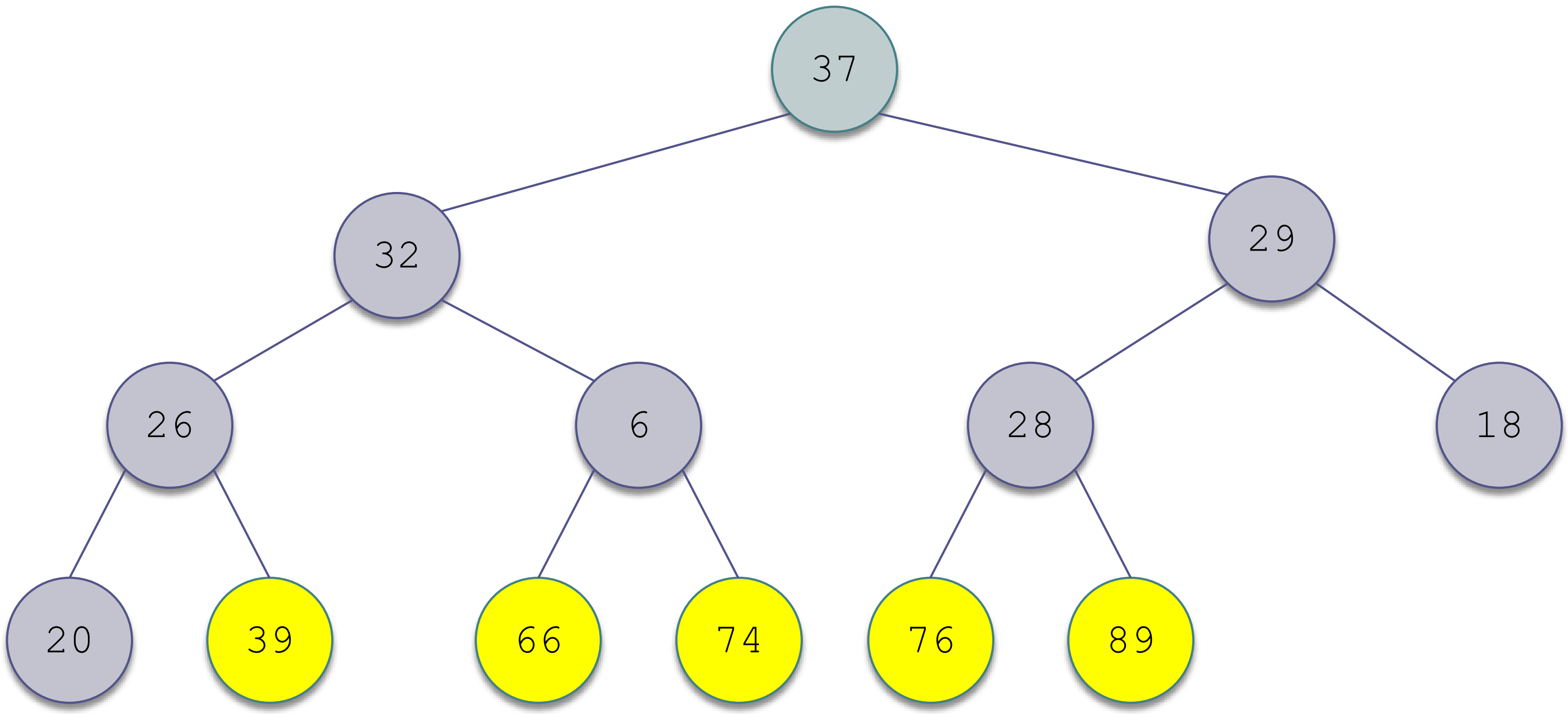
Trace of Heapsort (cont.)



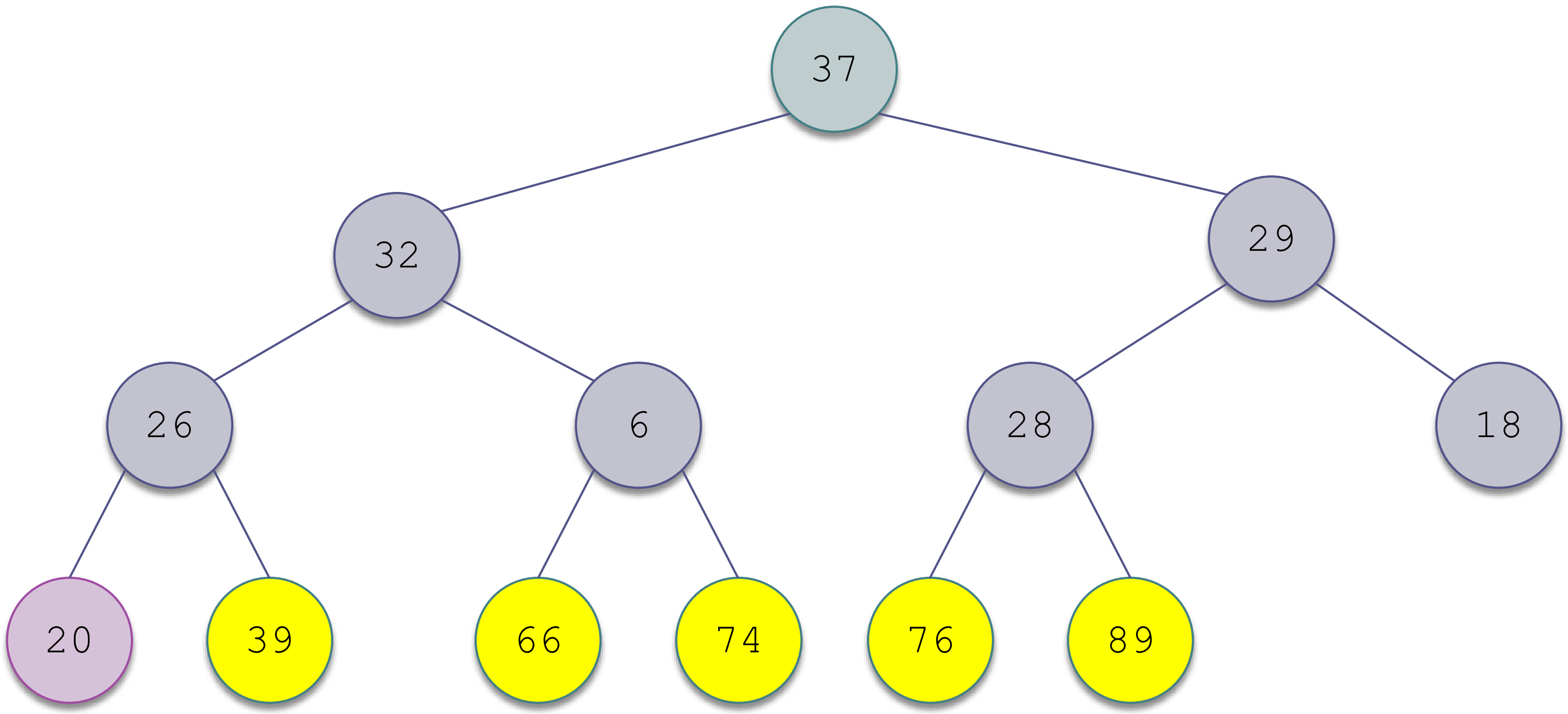
Trace of Heapsort (cont.)



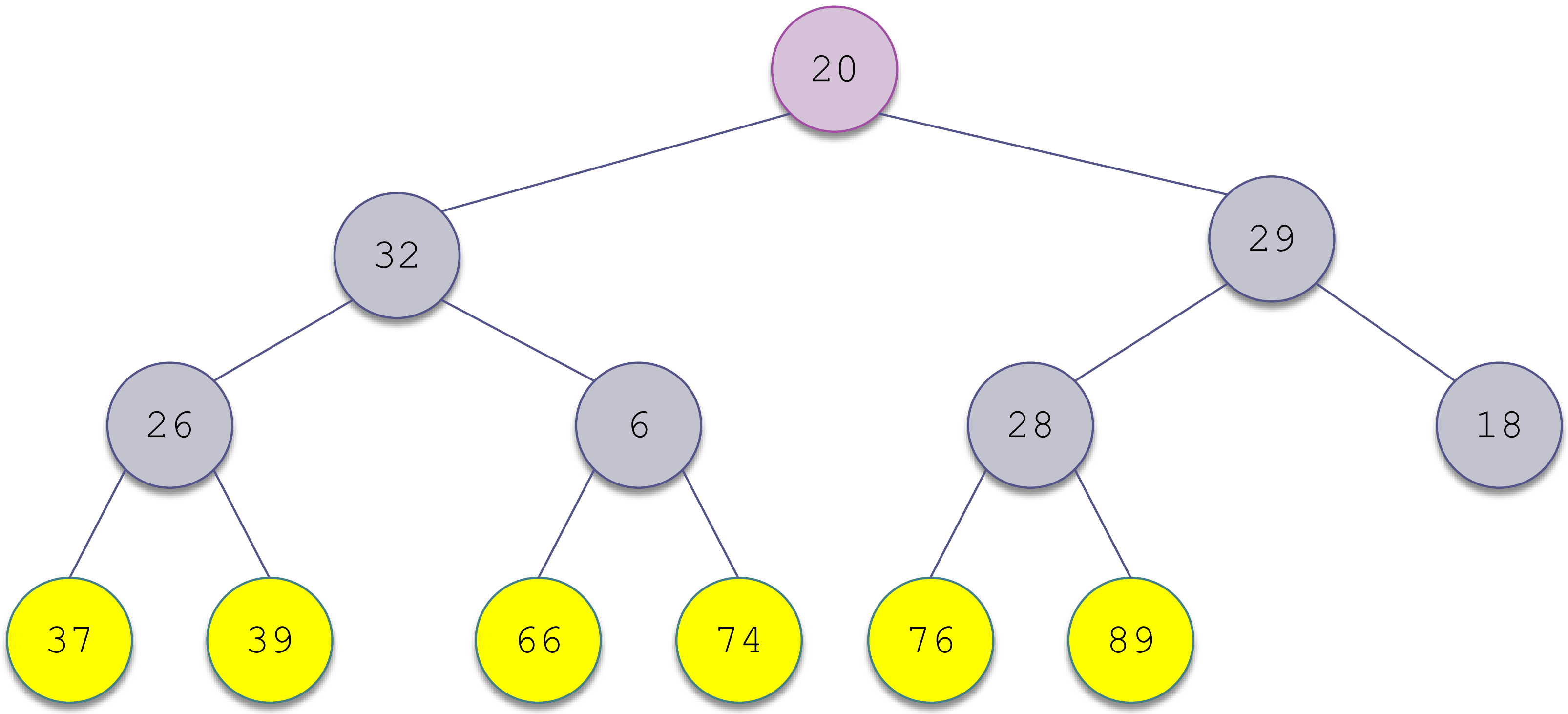
Trace of Heapsort (cont.)



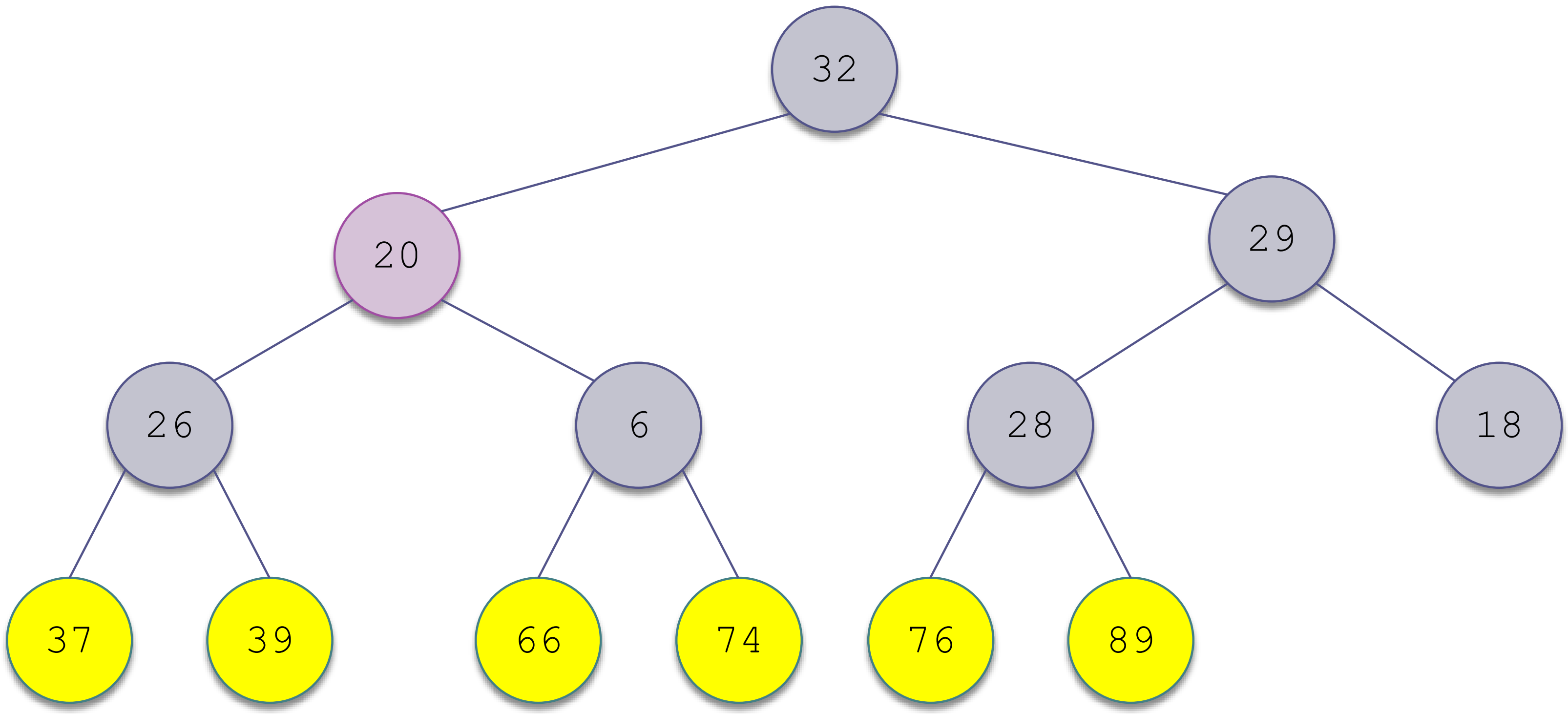
Trace of Heapsort (cont.)



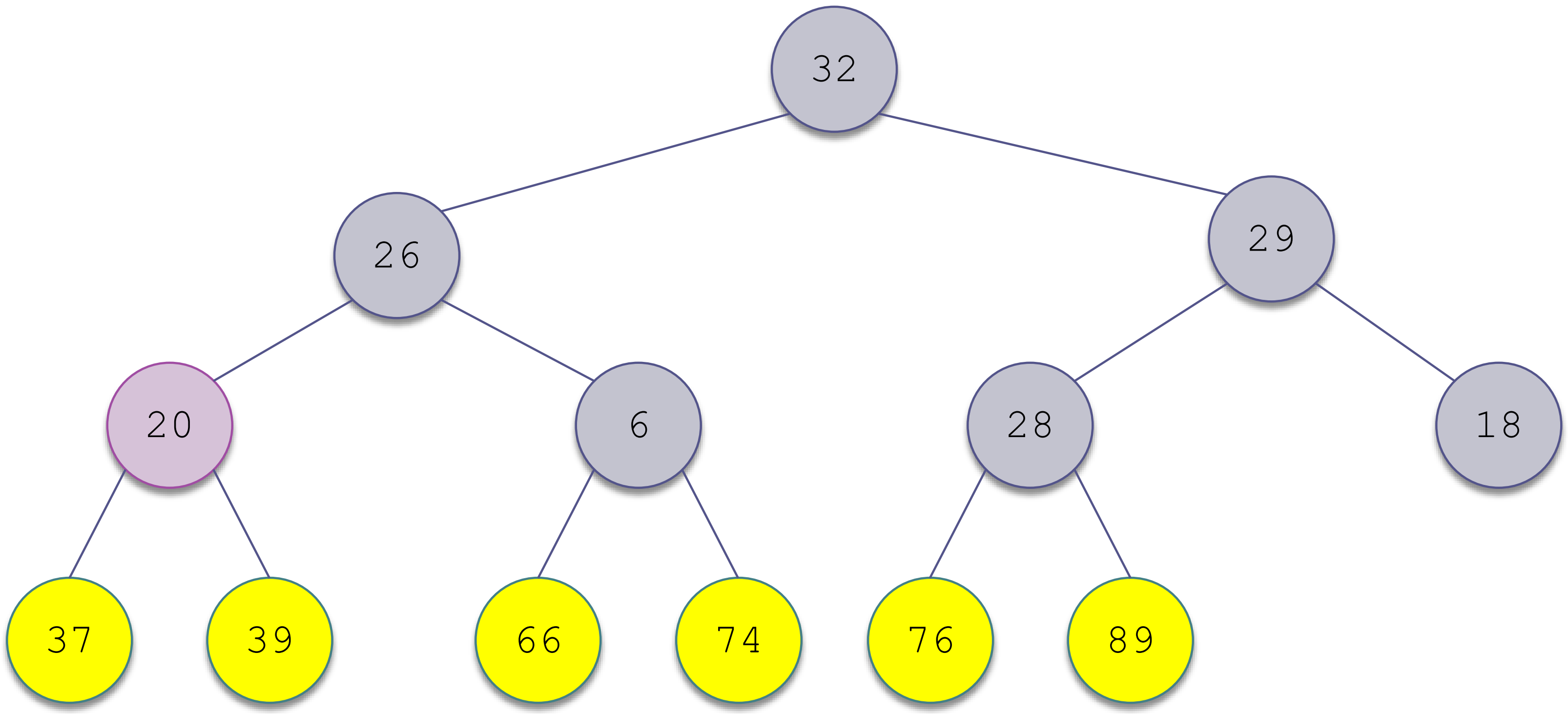
Trace of Heapsort (cont.)



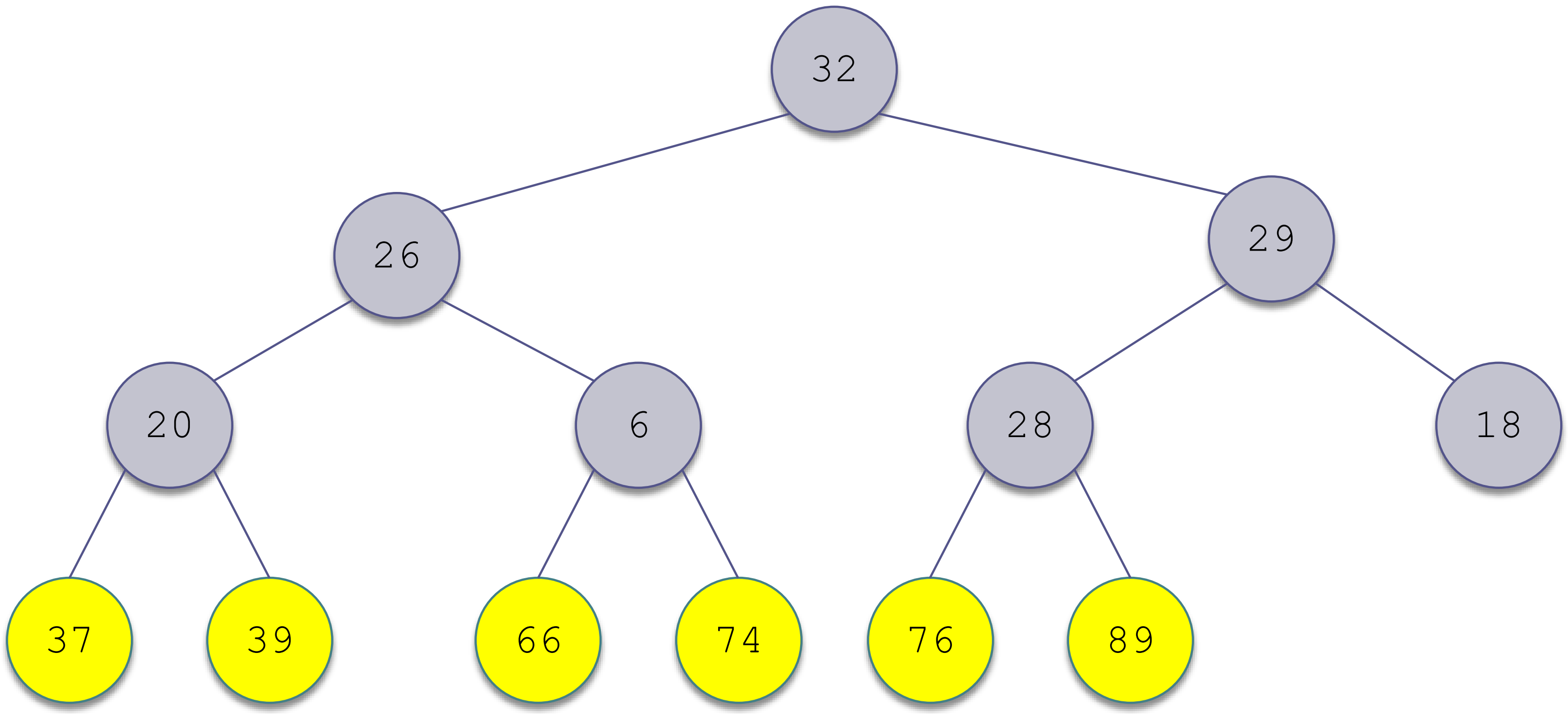
Trace of Heapsort (cont.)



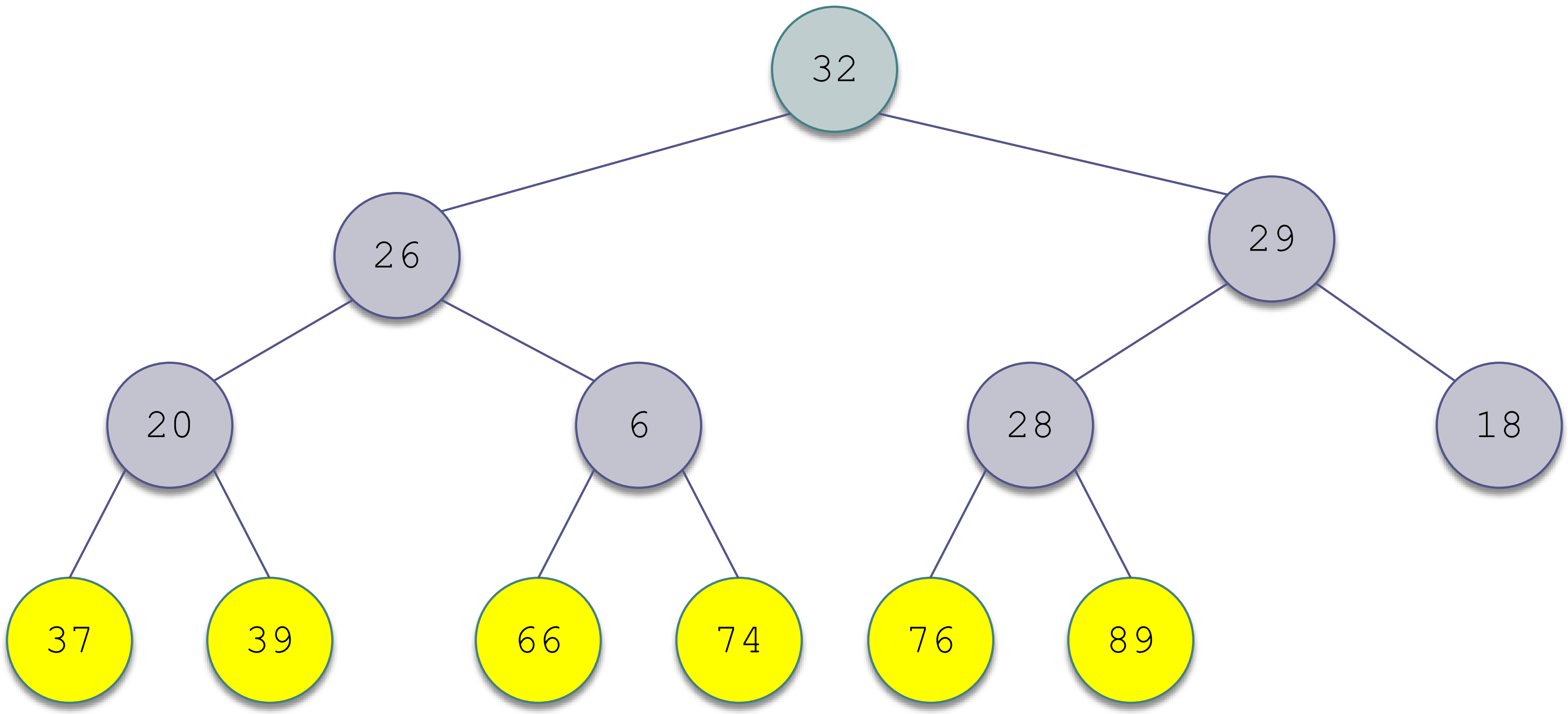
Trace of Heapsort (cont.)



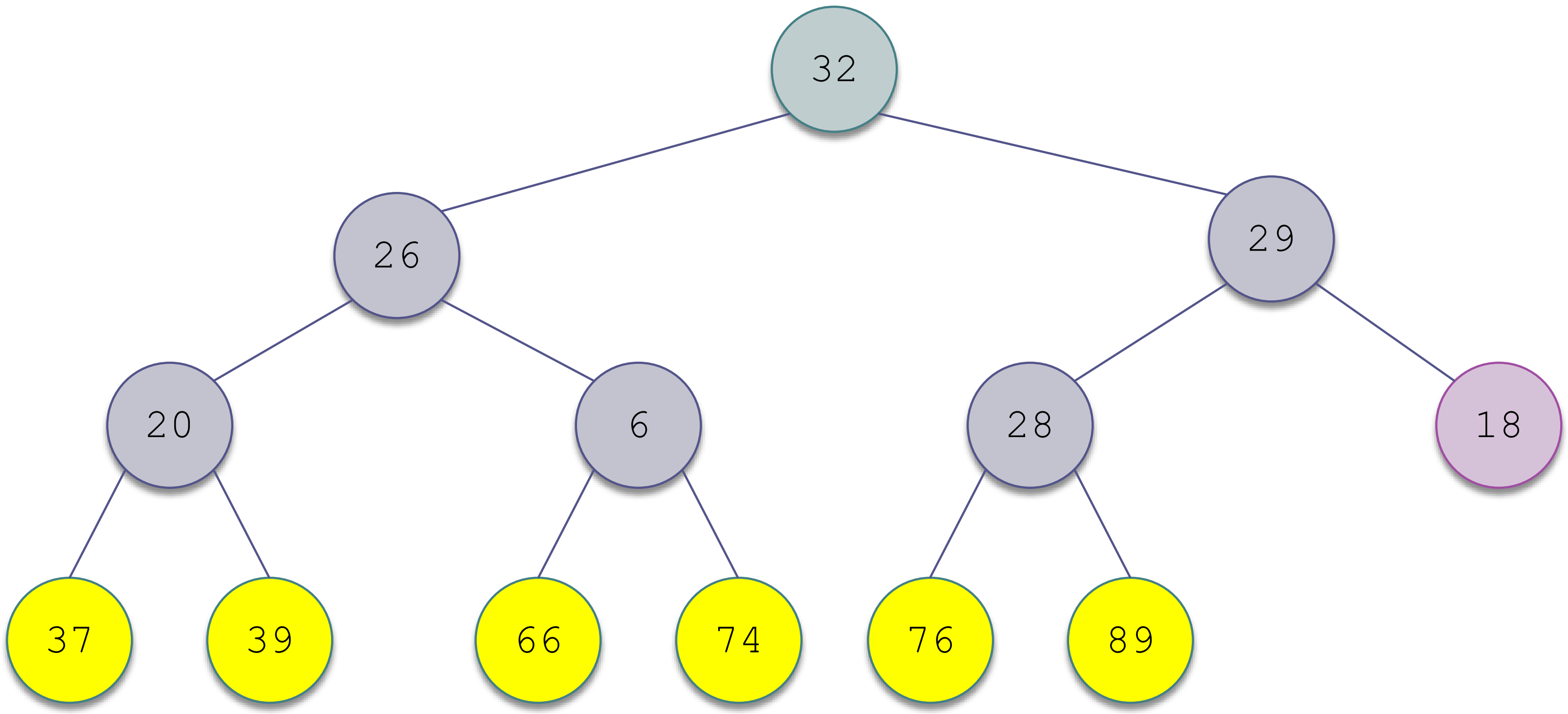
Trace of Heapsort (cont.)



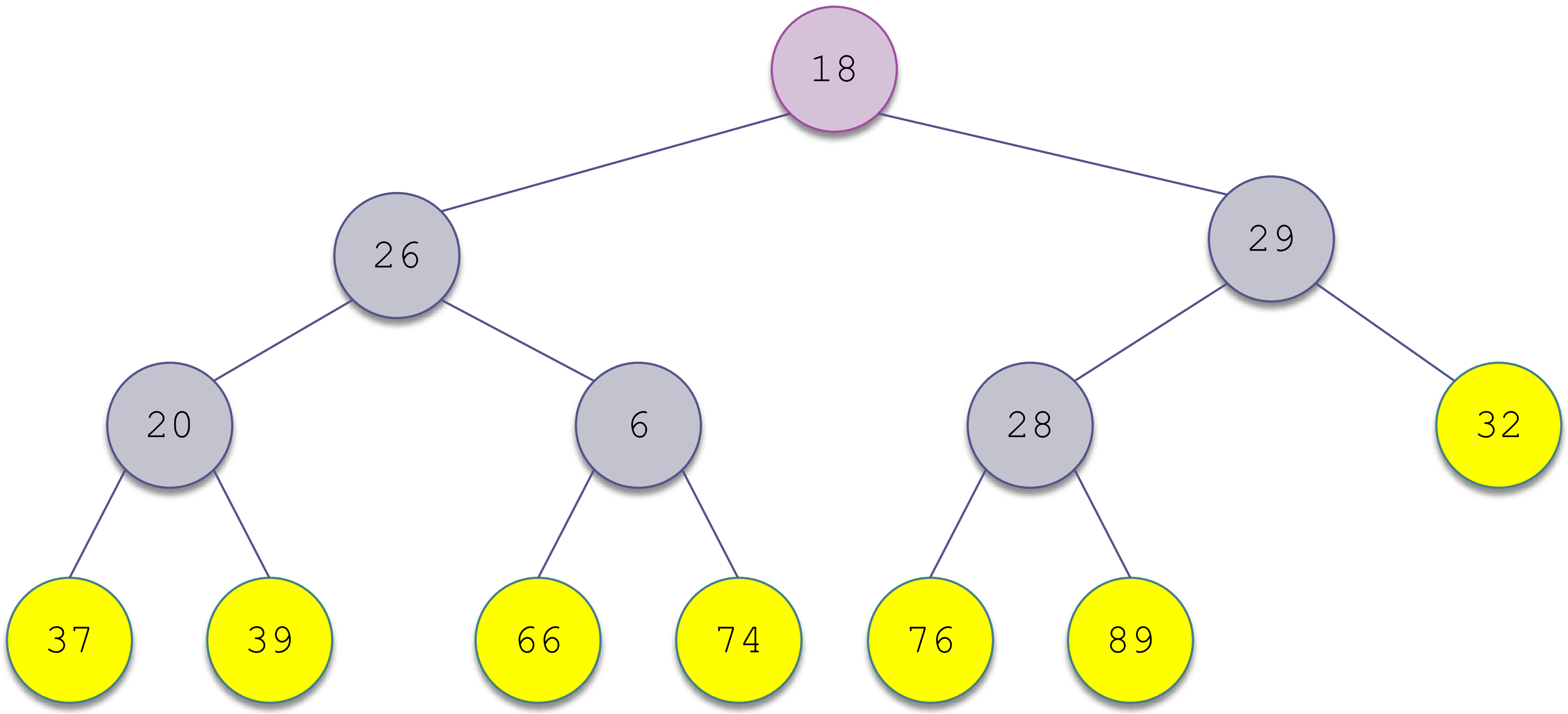
Trace of Heapsort (cont.)



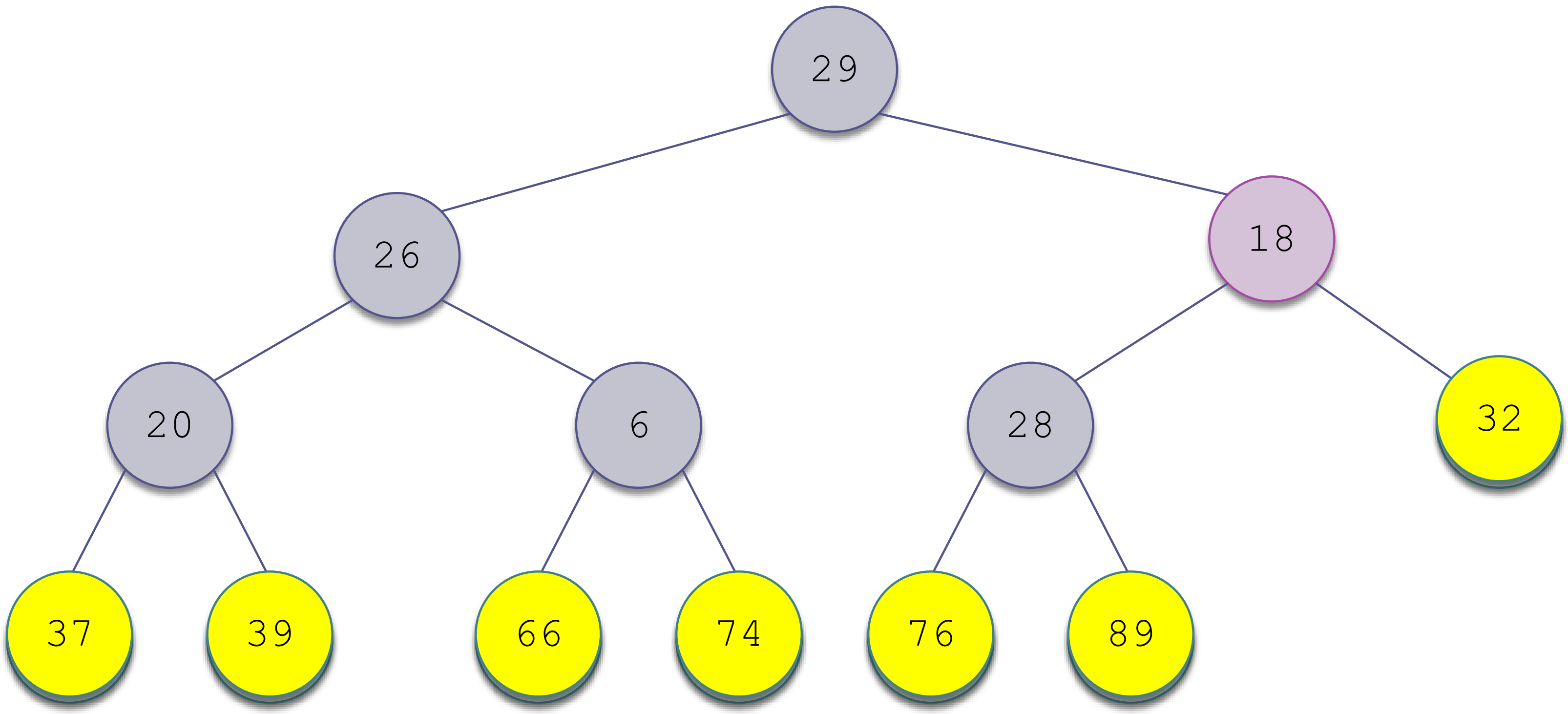
Trace of Heapsort (cont.)



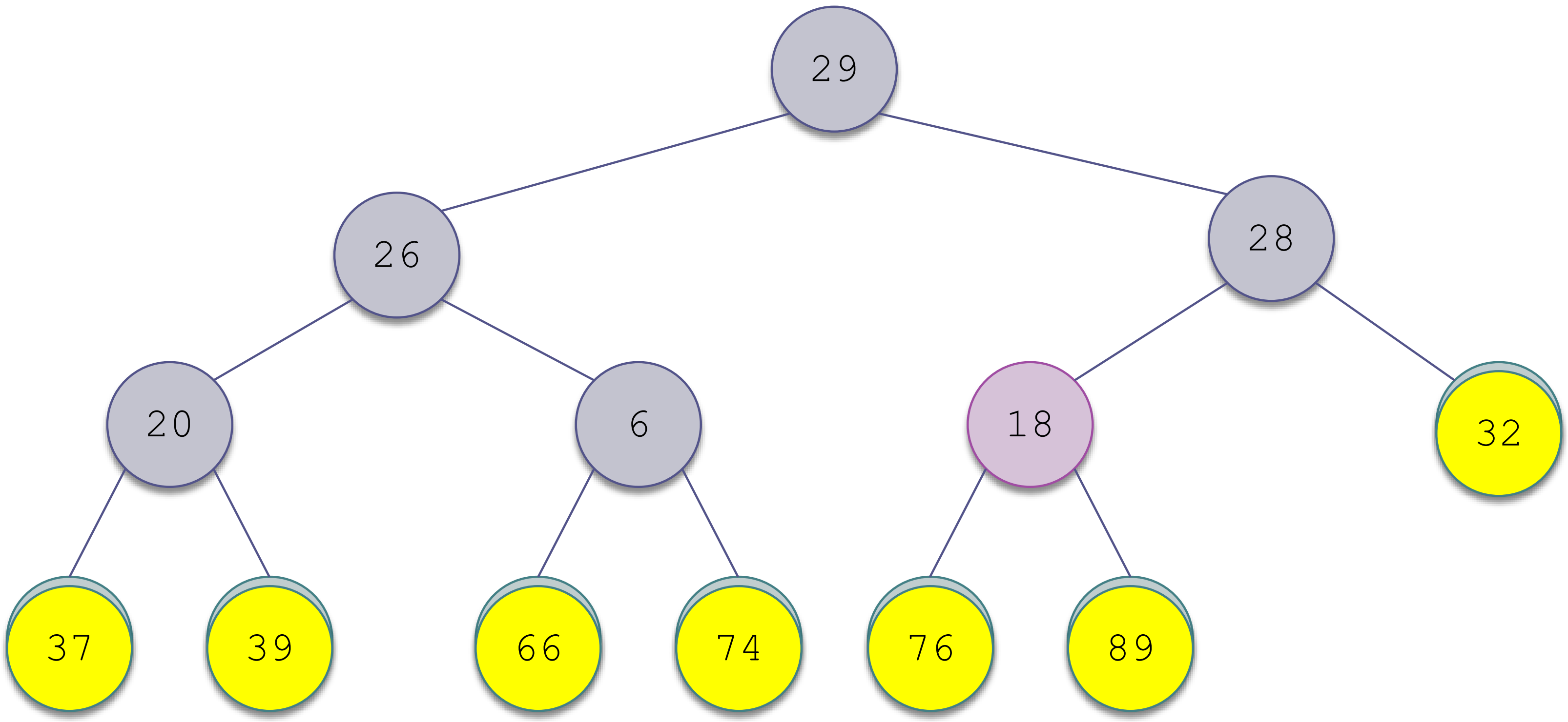
Trace of Heapsort (cont.)



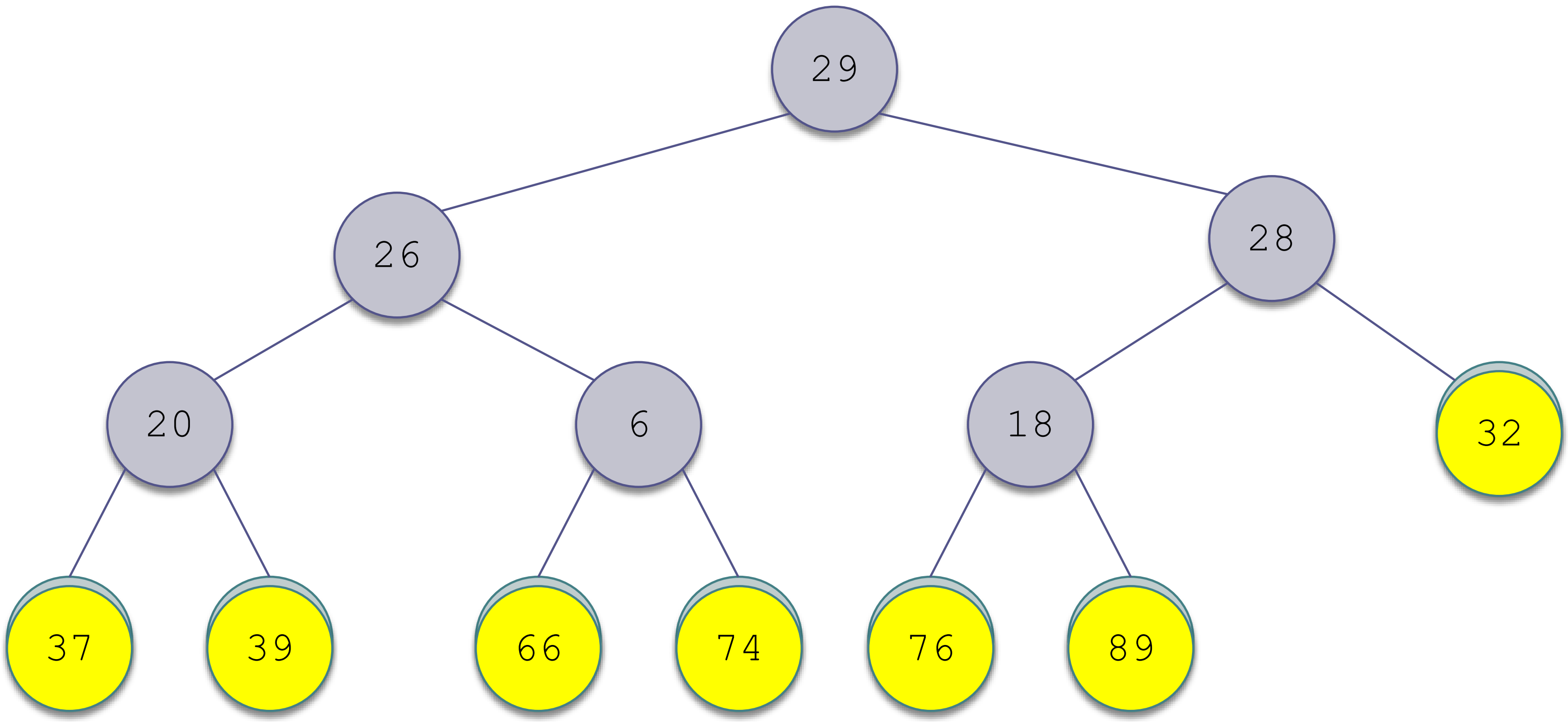
Trace of Heapsort (cont.)



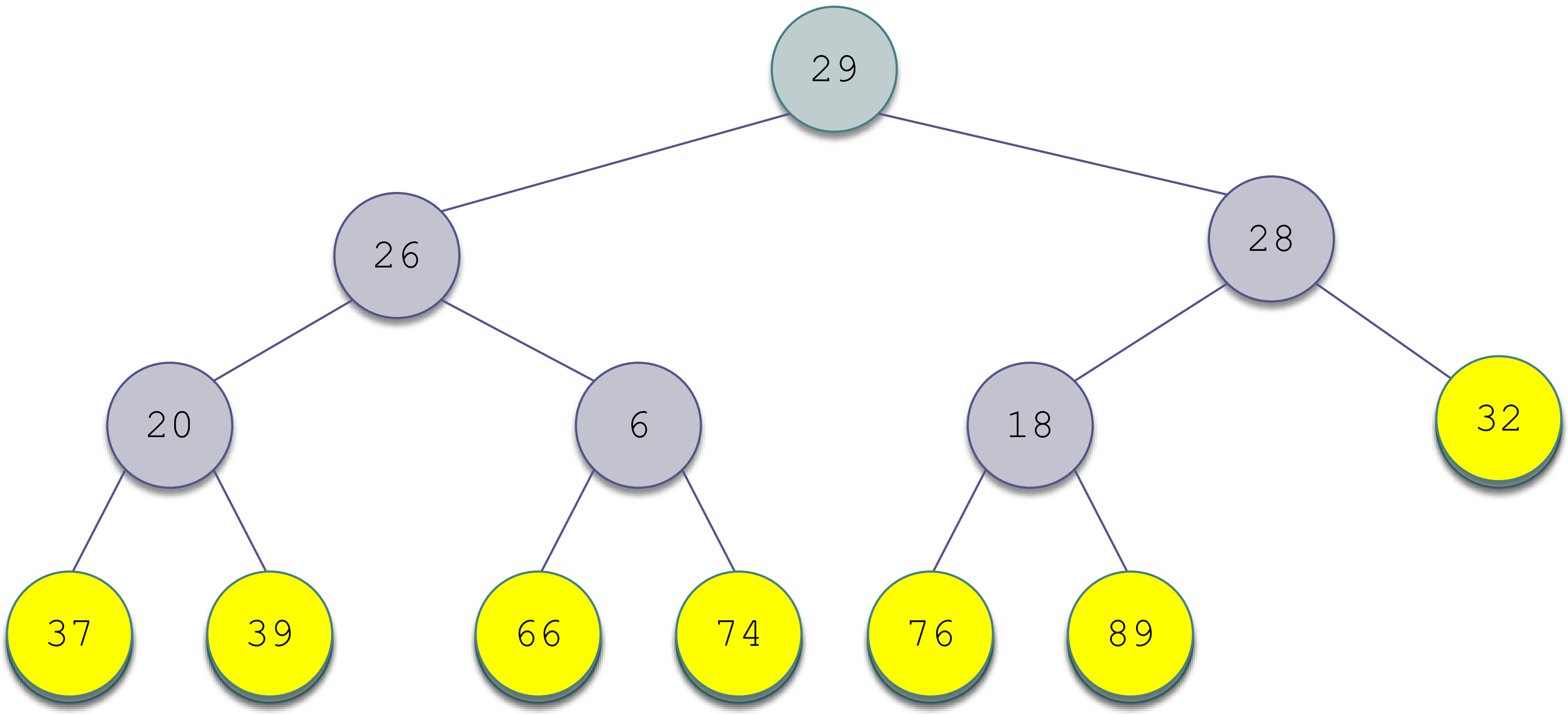
Trace of Heapsort (cont.)



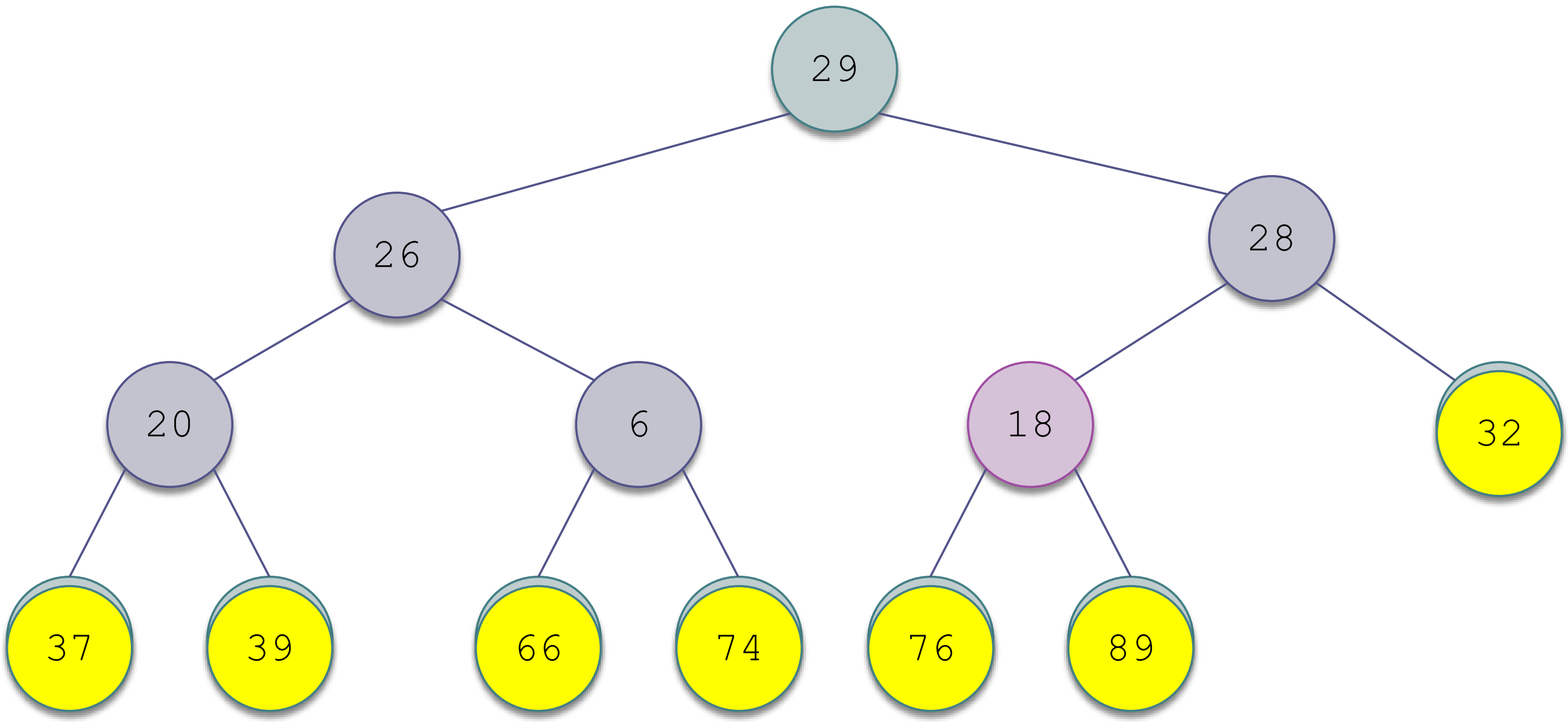
Trace of Heapsort (cont.)



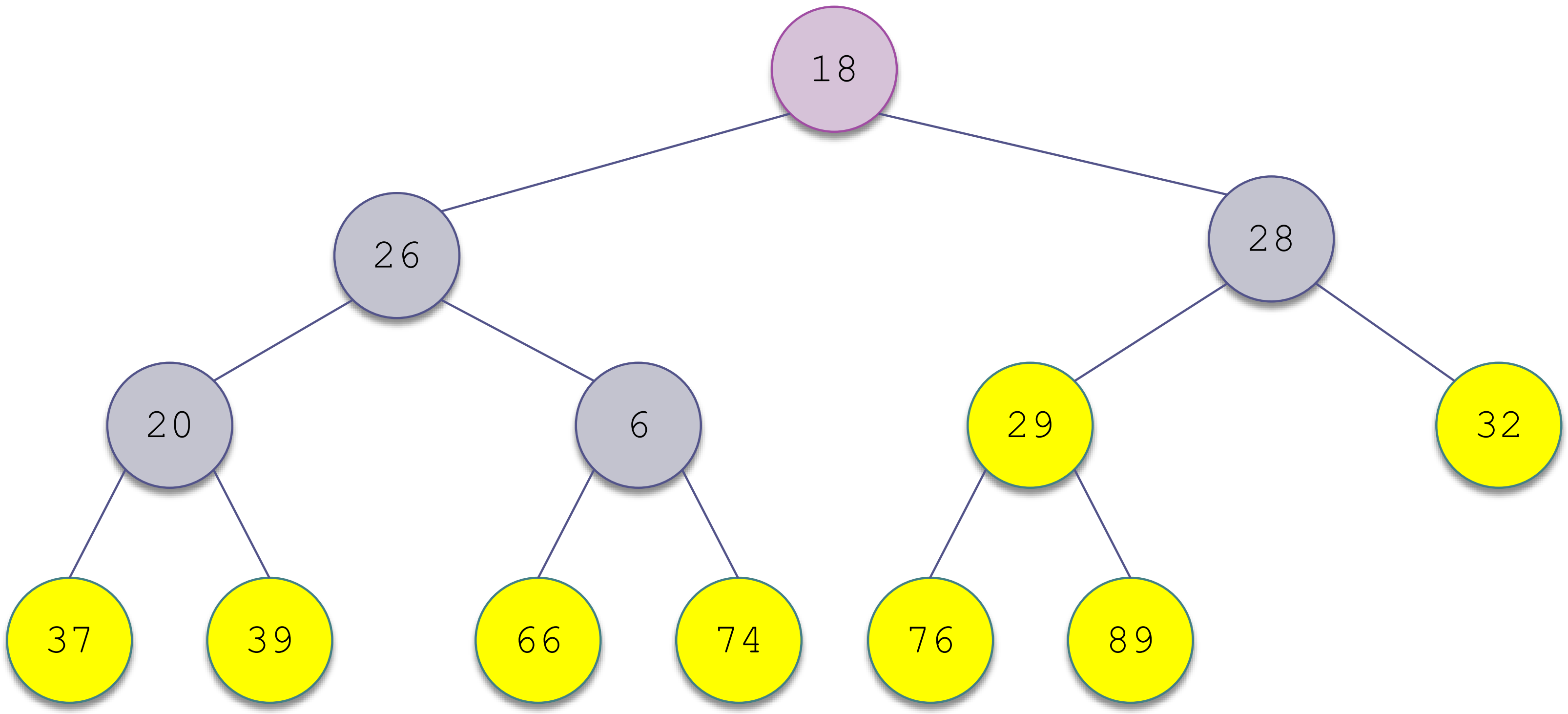
Trace of Heapsort (cont.)



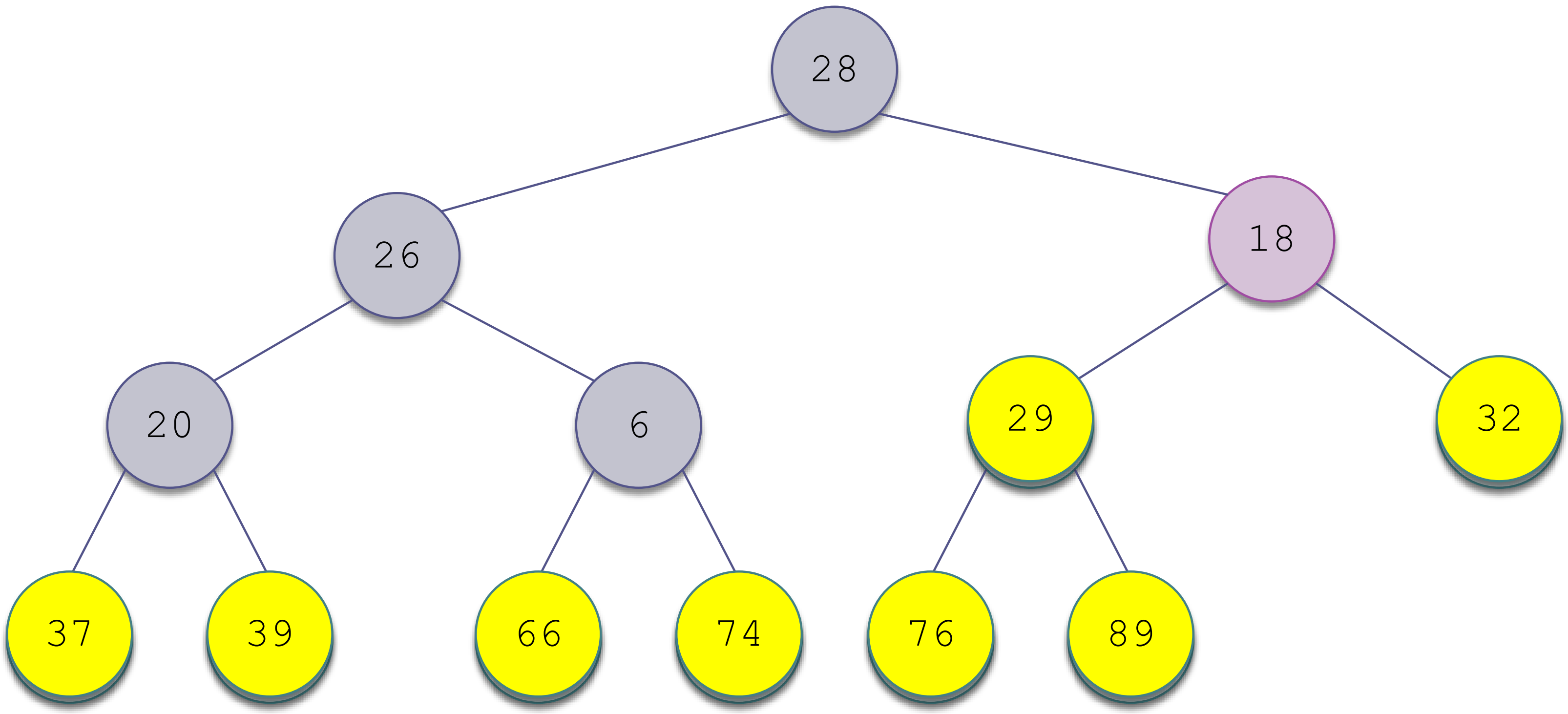
Trace of Heapsort (cont.)



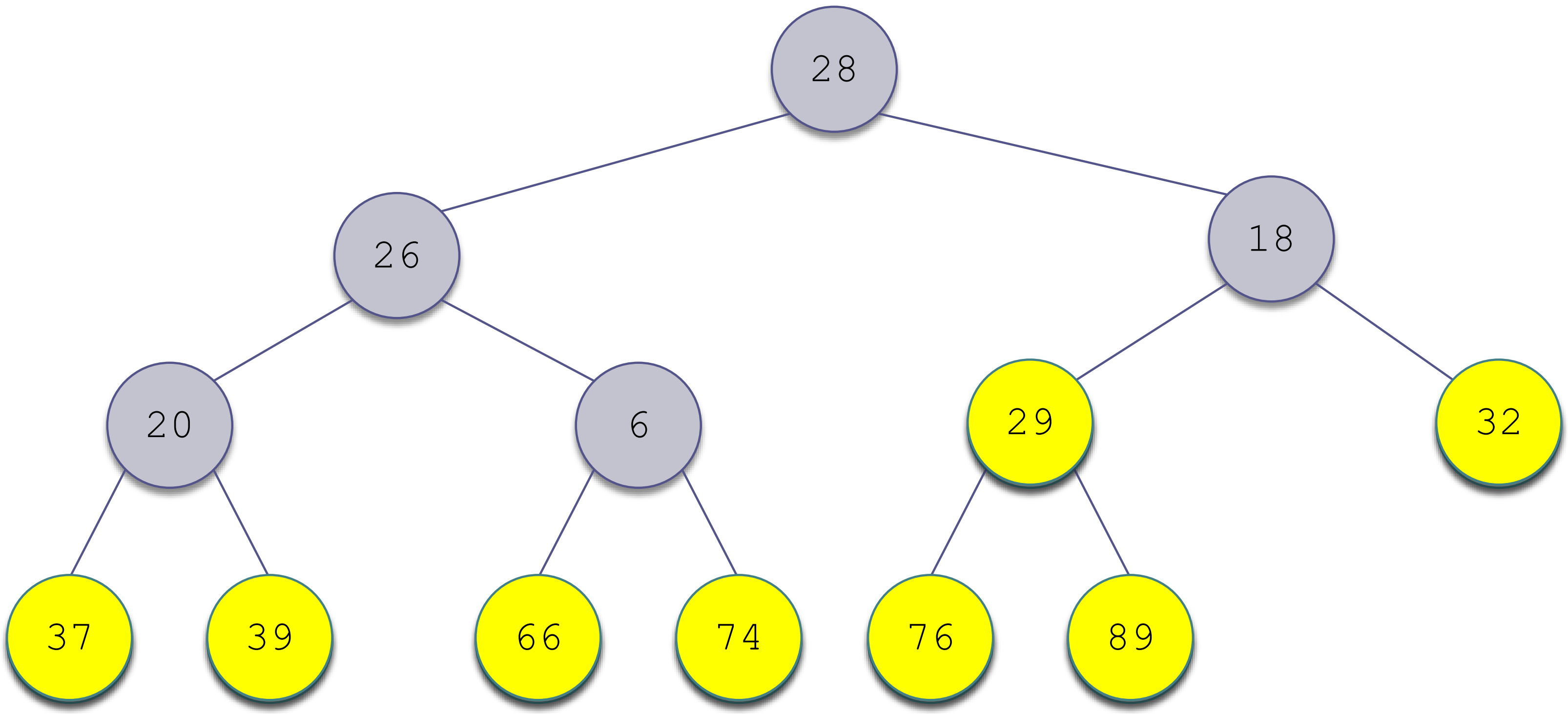
Trace of Heapsort (cont.)



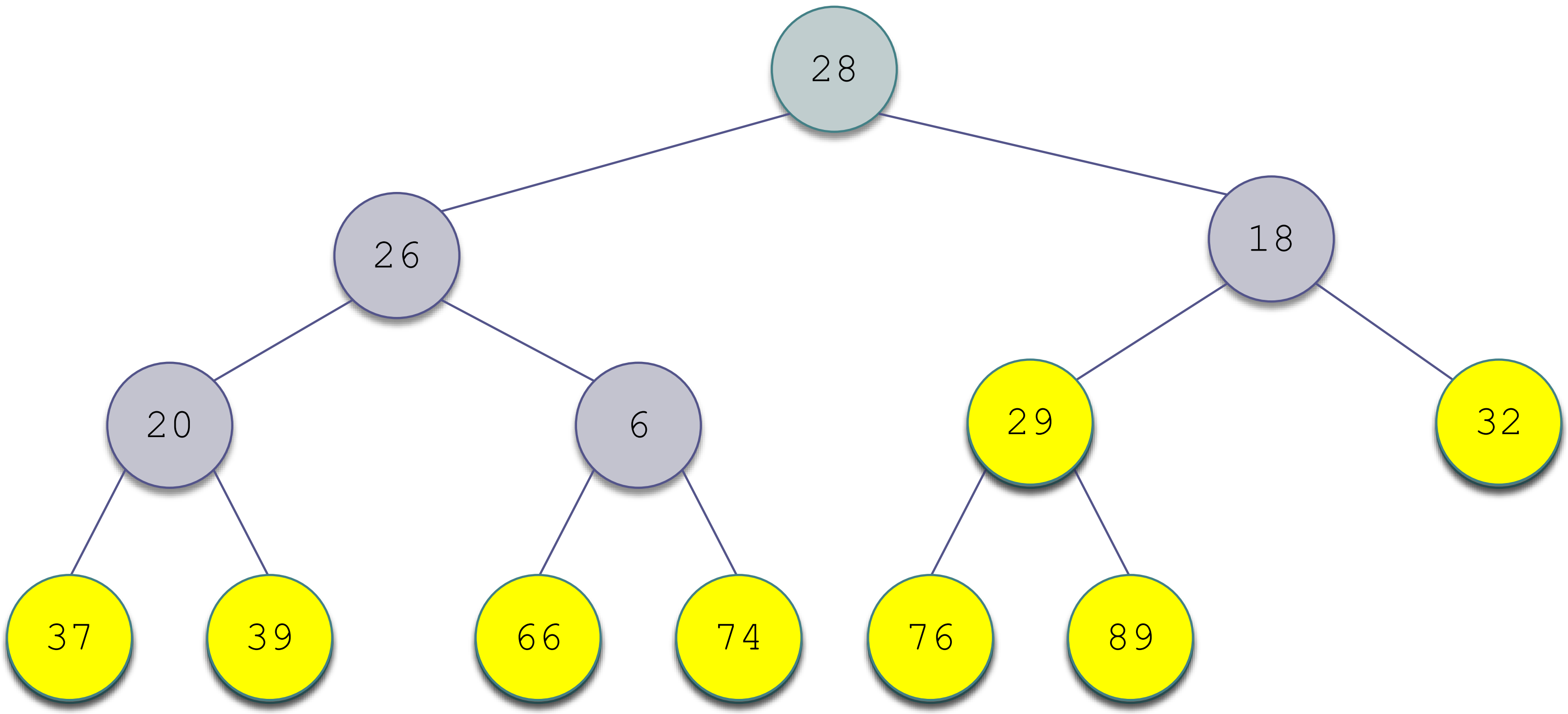
Trace of Heapsort (cont.)



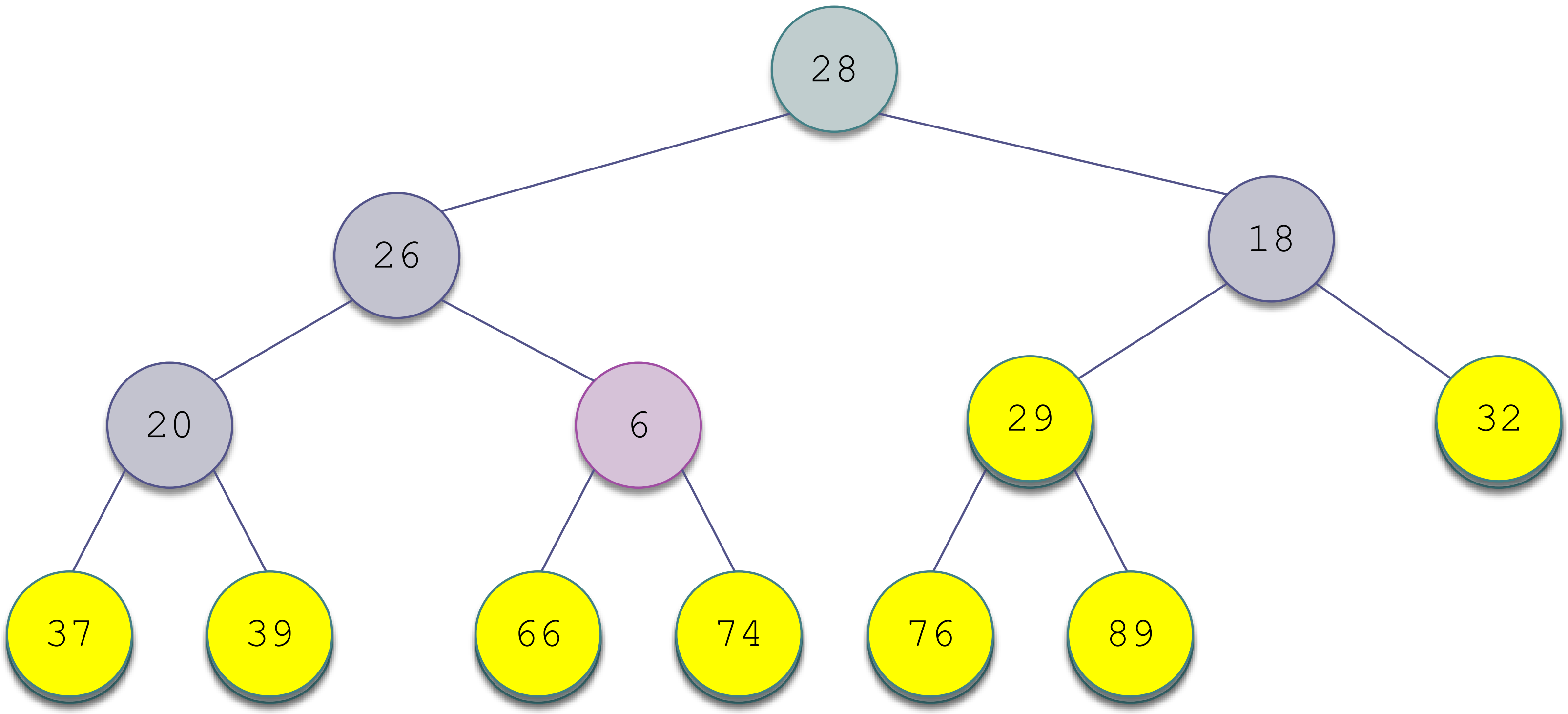
Trace of Heapsort (cont.)



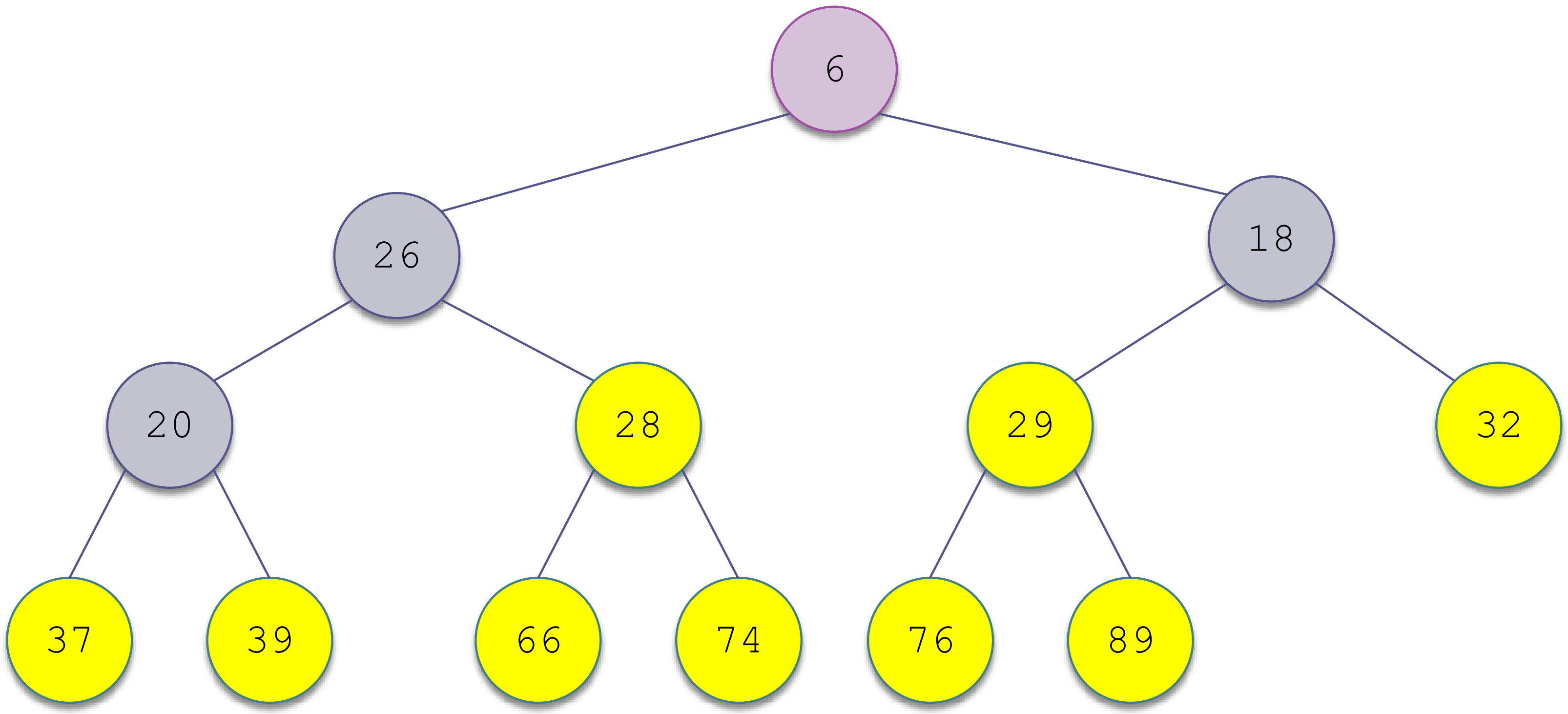
Trace of Heapsort (cont.)



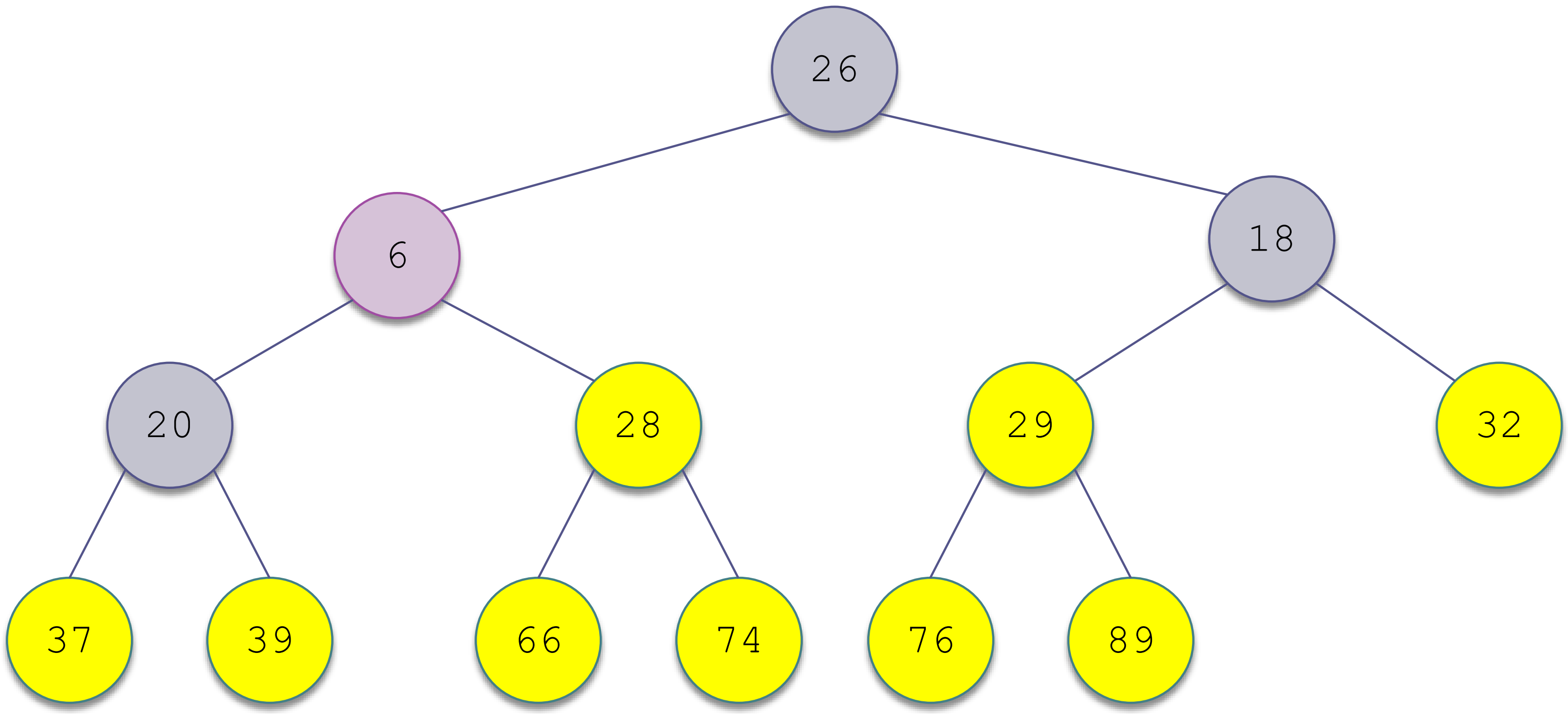
Trace of Heapsort (cont.)



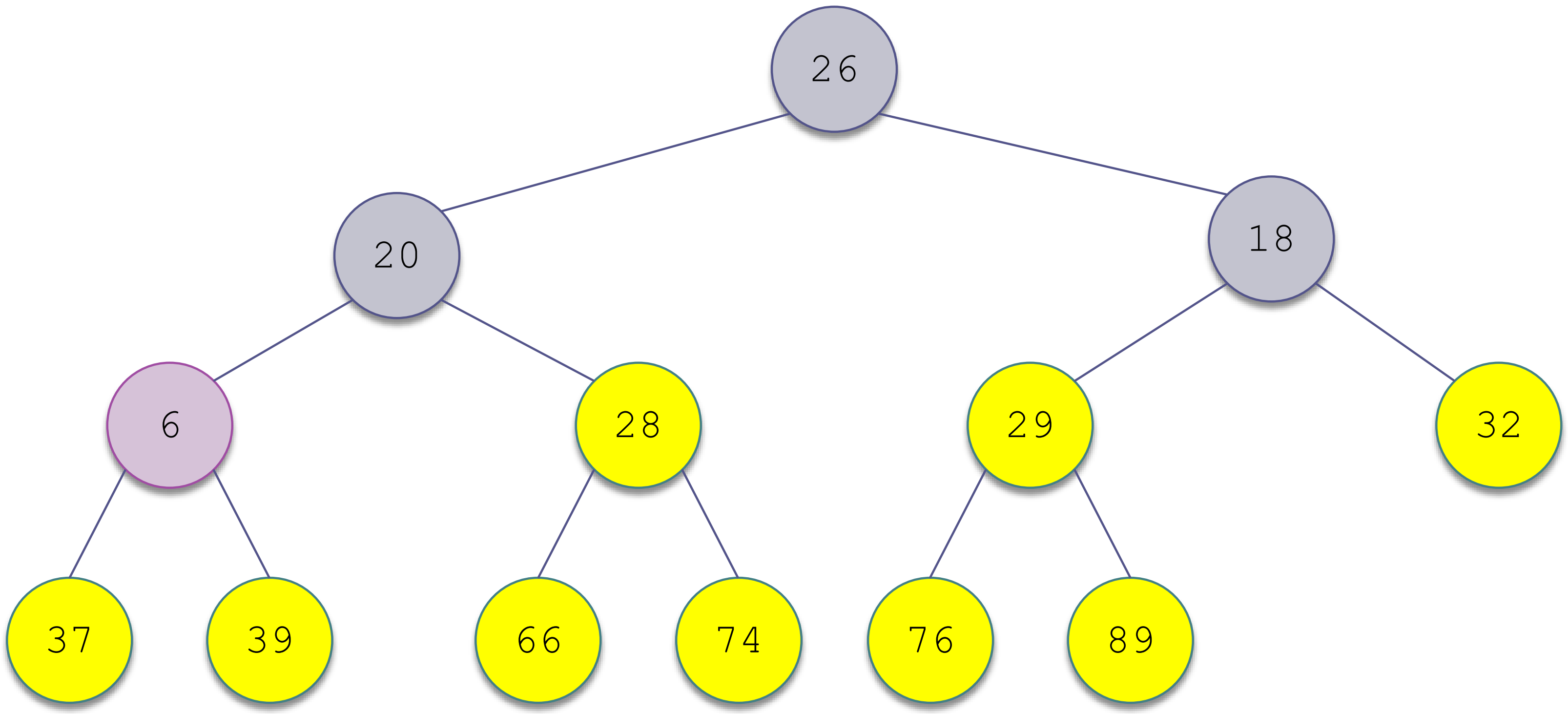
Trace of Heapsort (cont.)



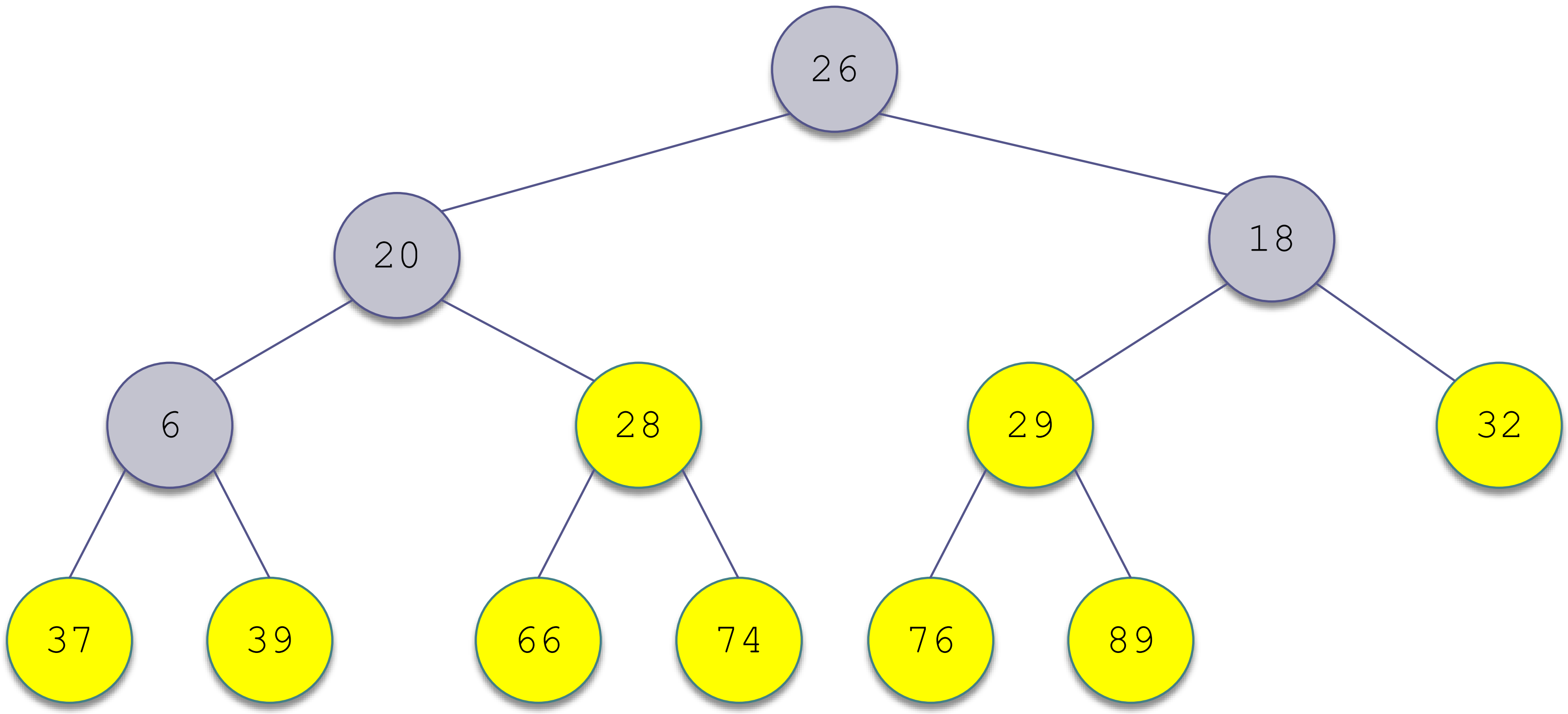
Trace of Heapsort (cont.)



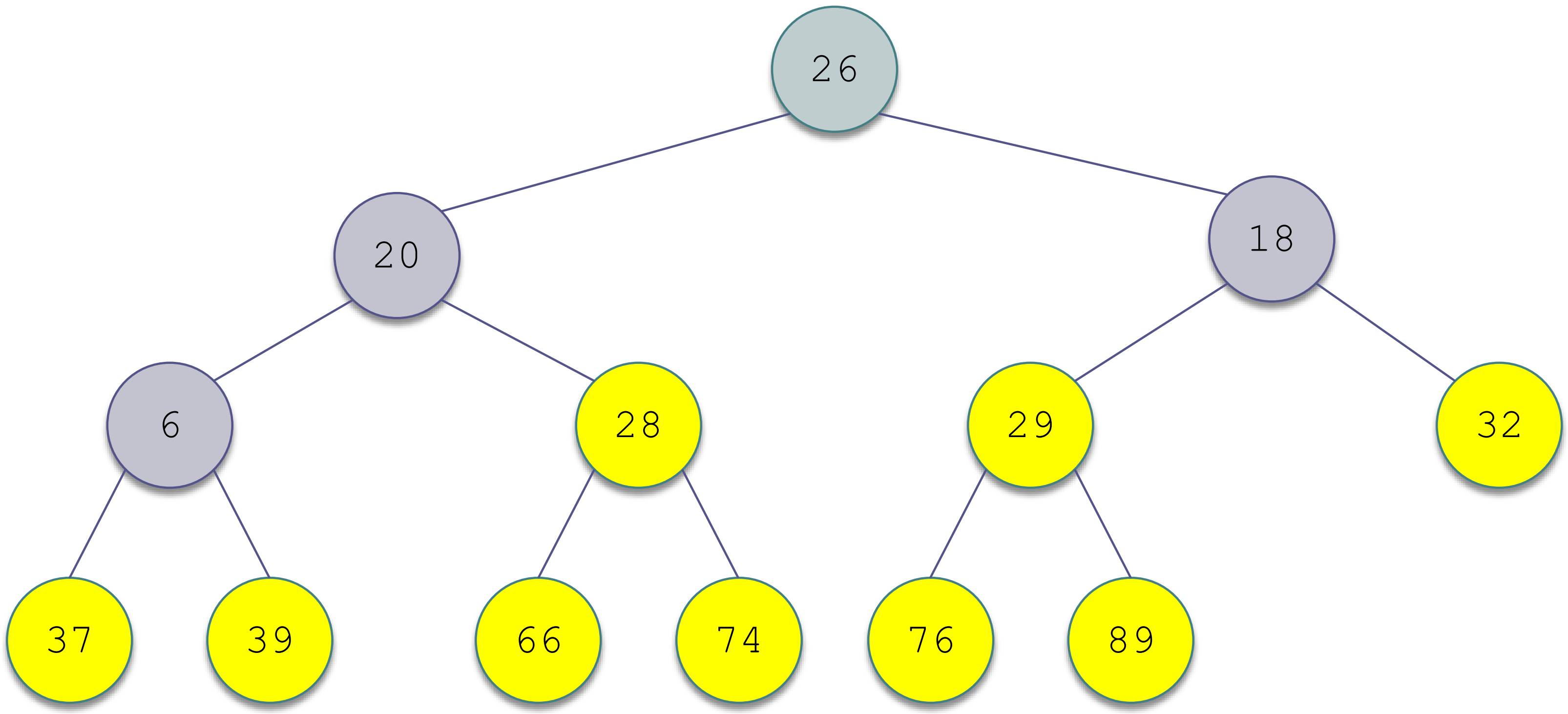
Trace of Heapsort (cont.)



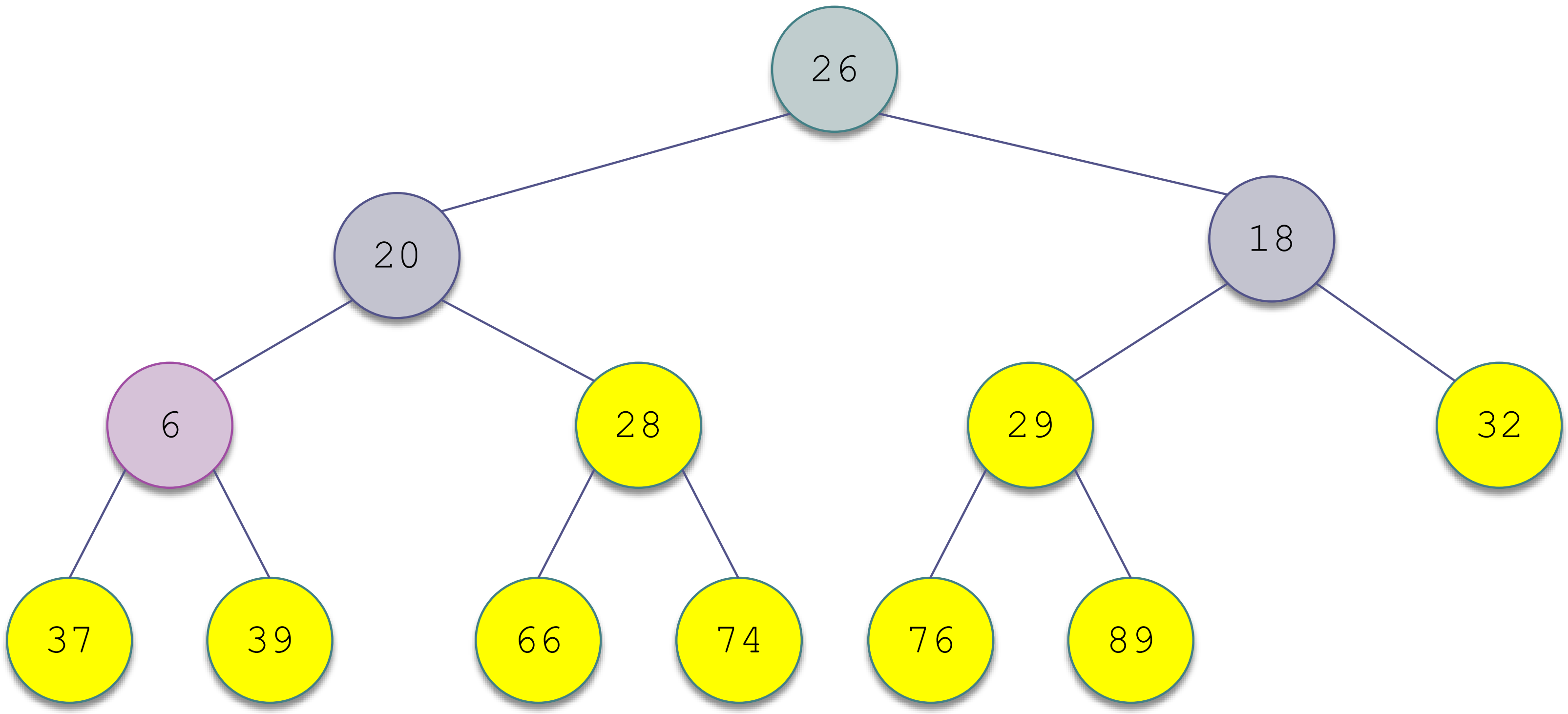
Trace of Heapsort (cont.)



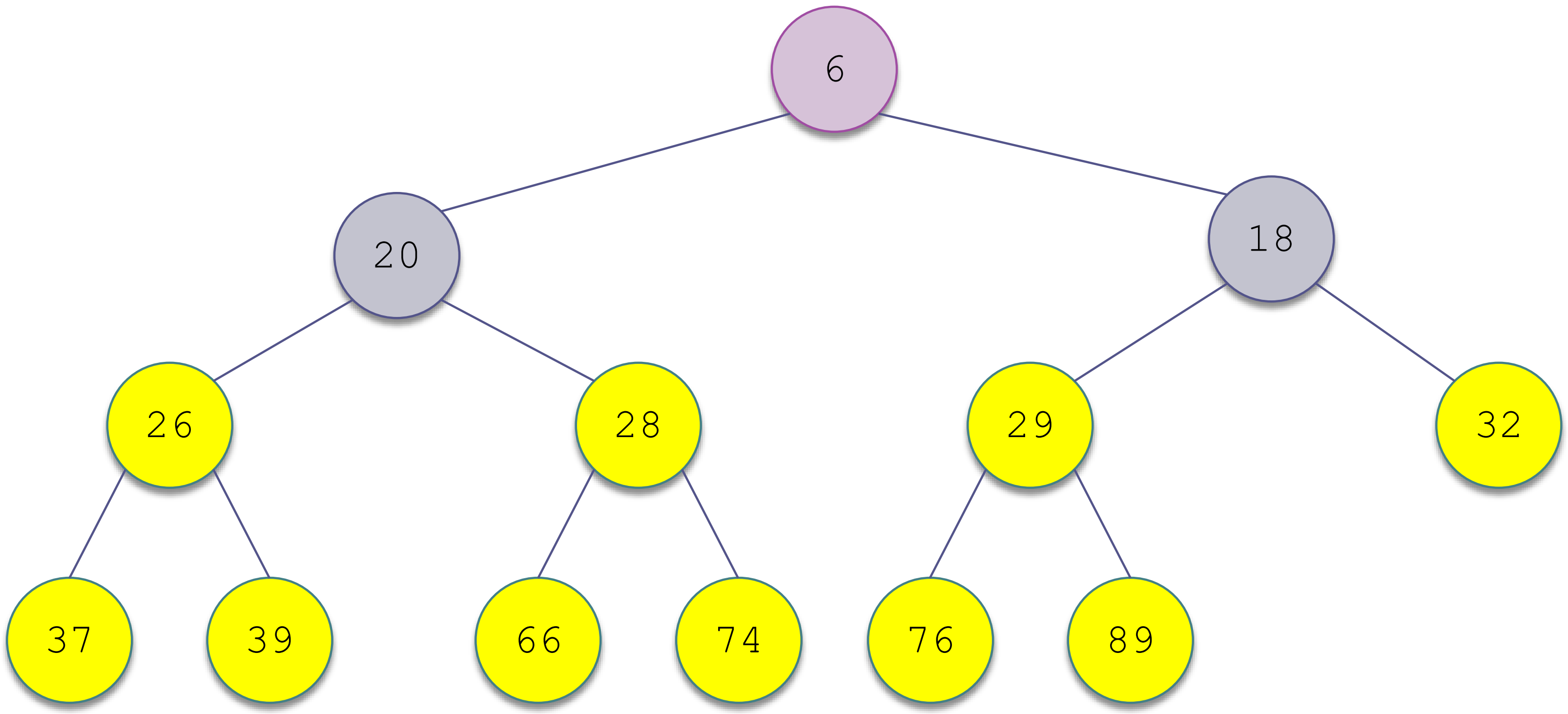
Trace of Heapsort (cont.)



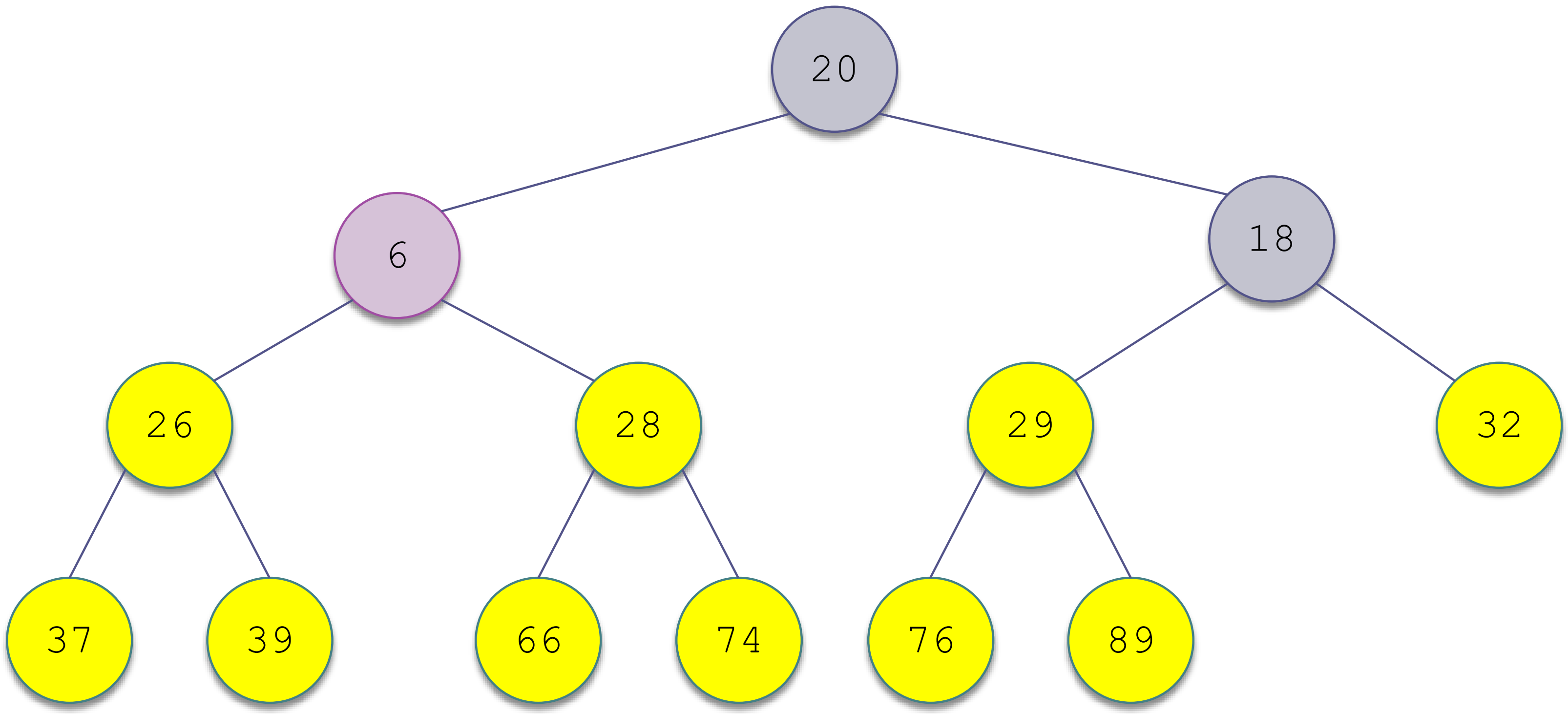
Trace of Heapsort (cont.)



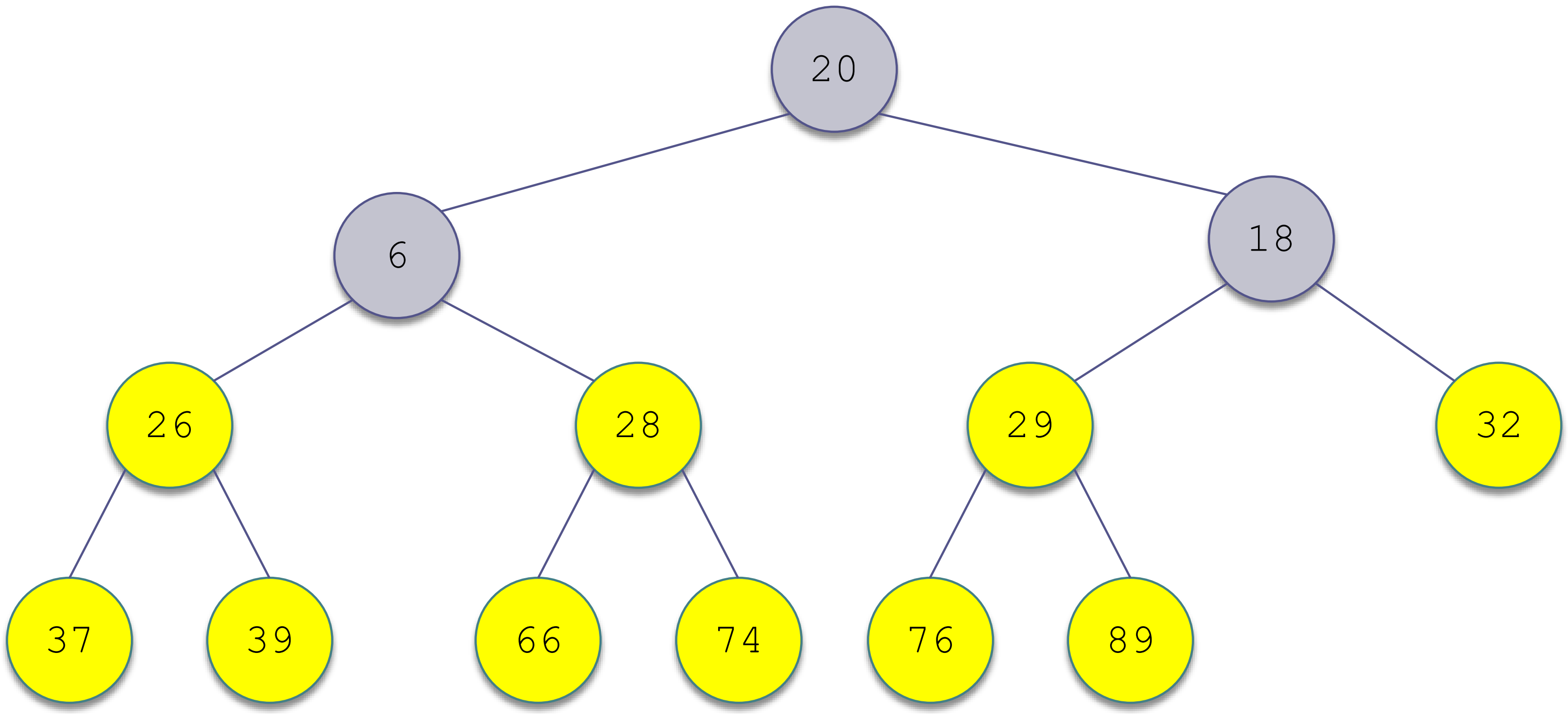
Trace of Heapsort (cont.)



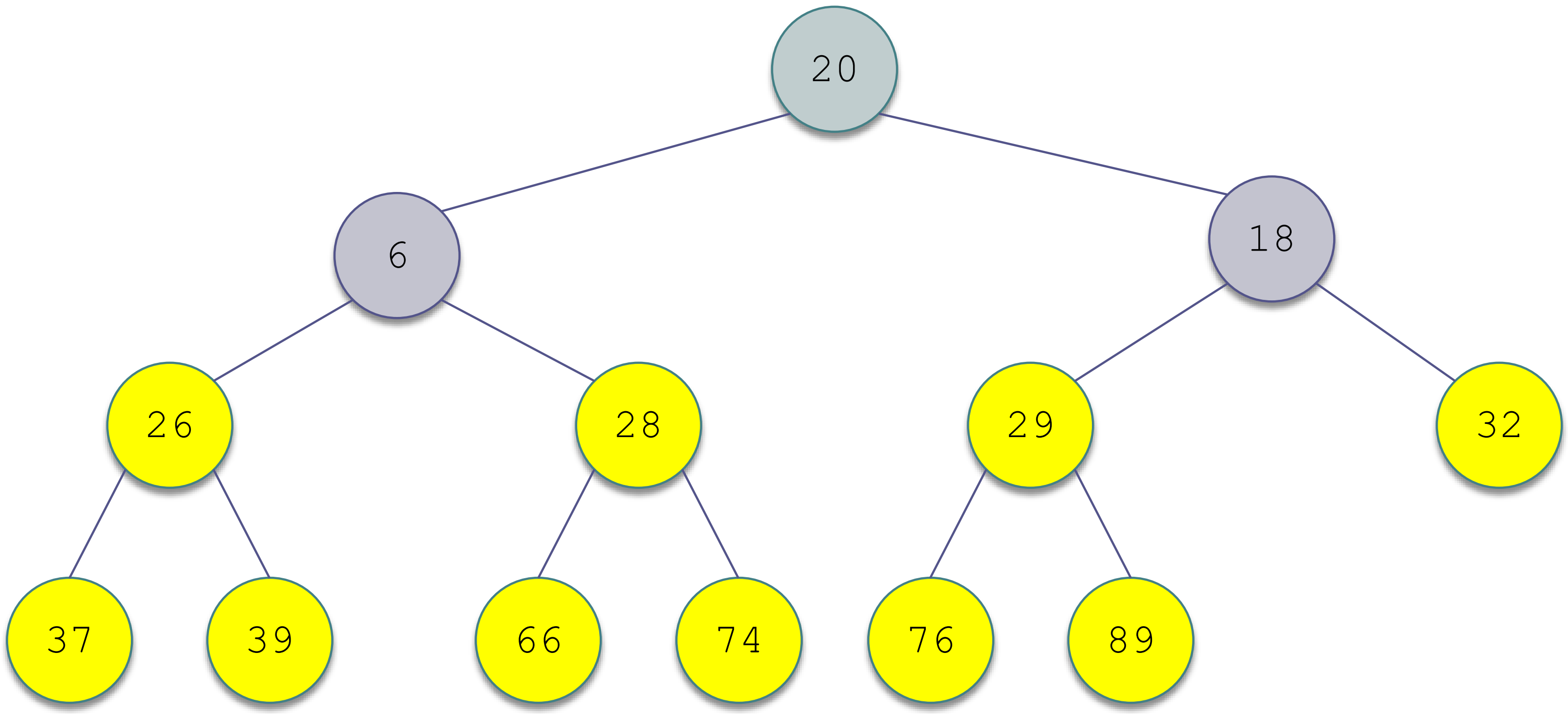
Trace of Heapsort (cont.)



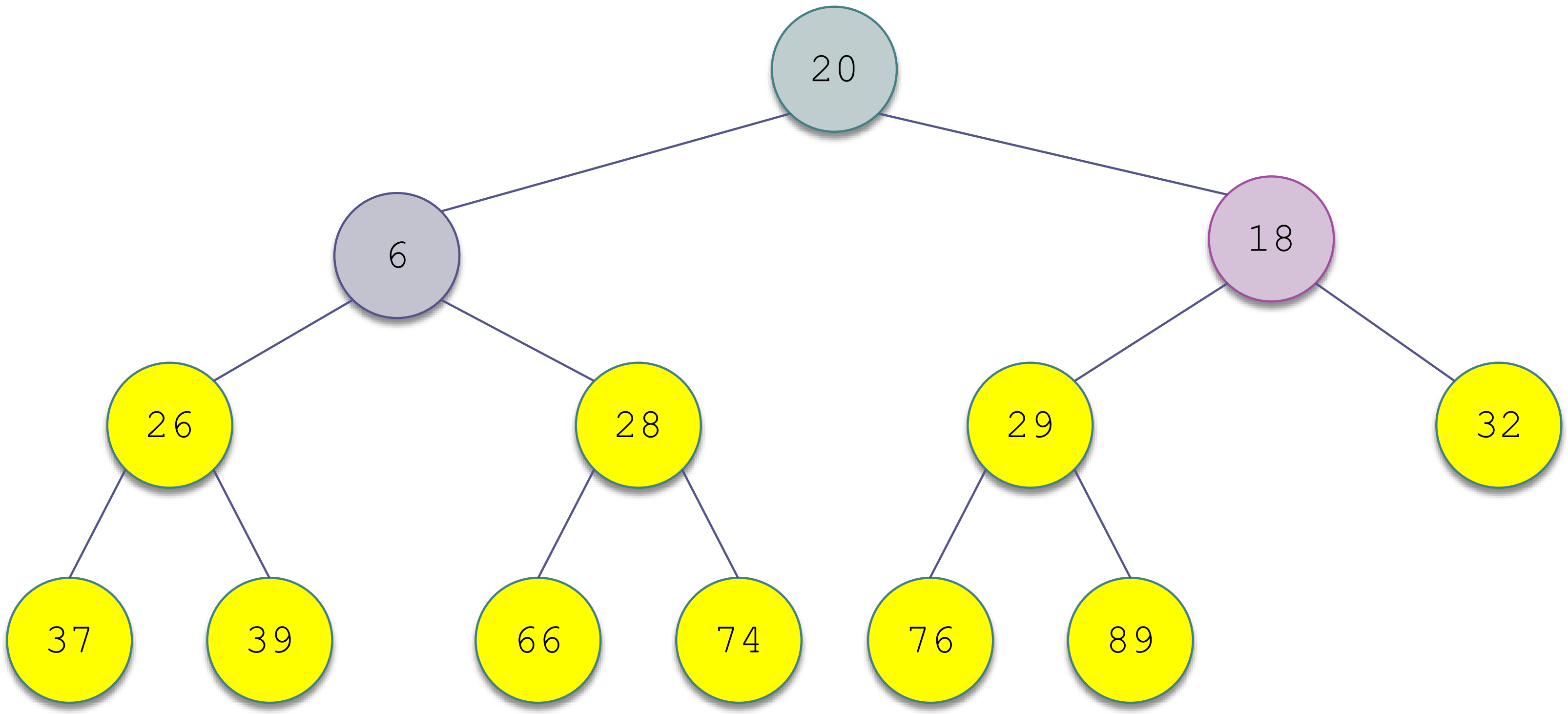
Trace of Heapsort (cont.)



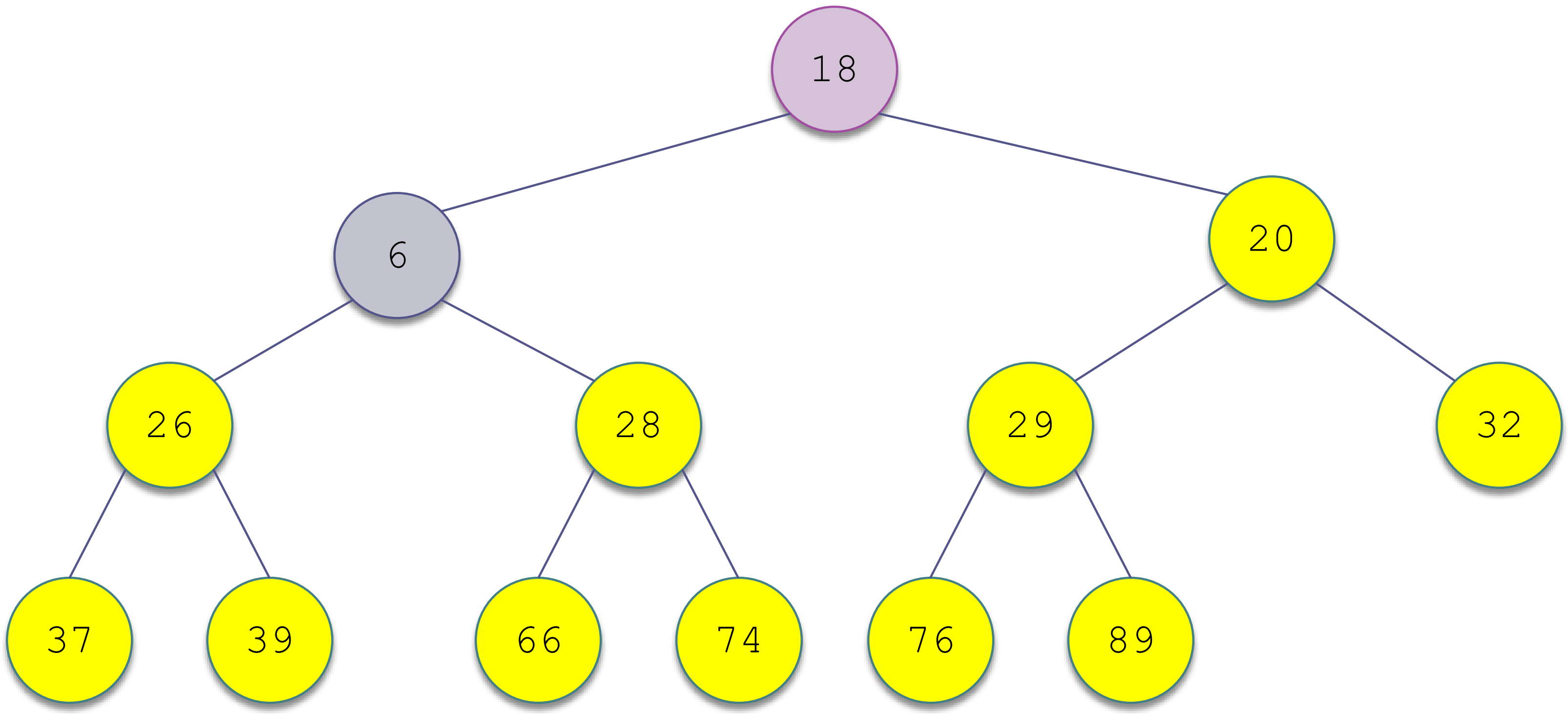
Trace of Heapsort (cont.)



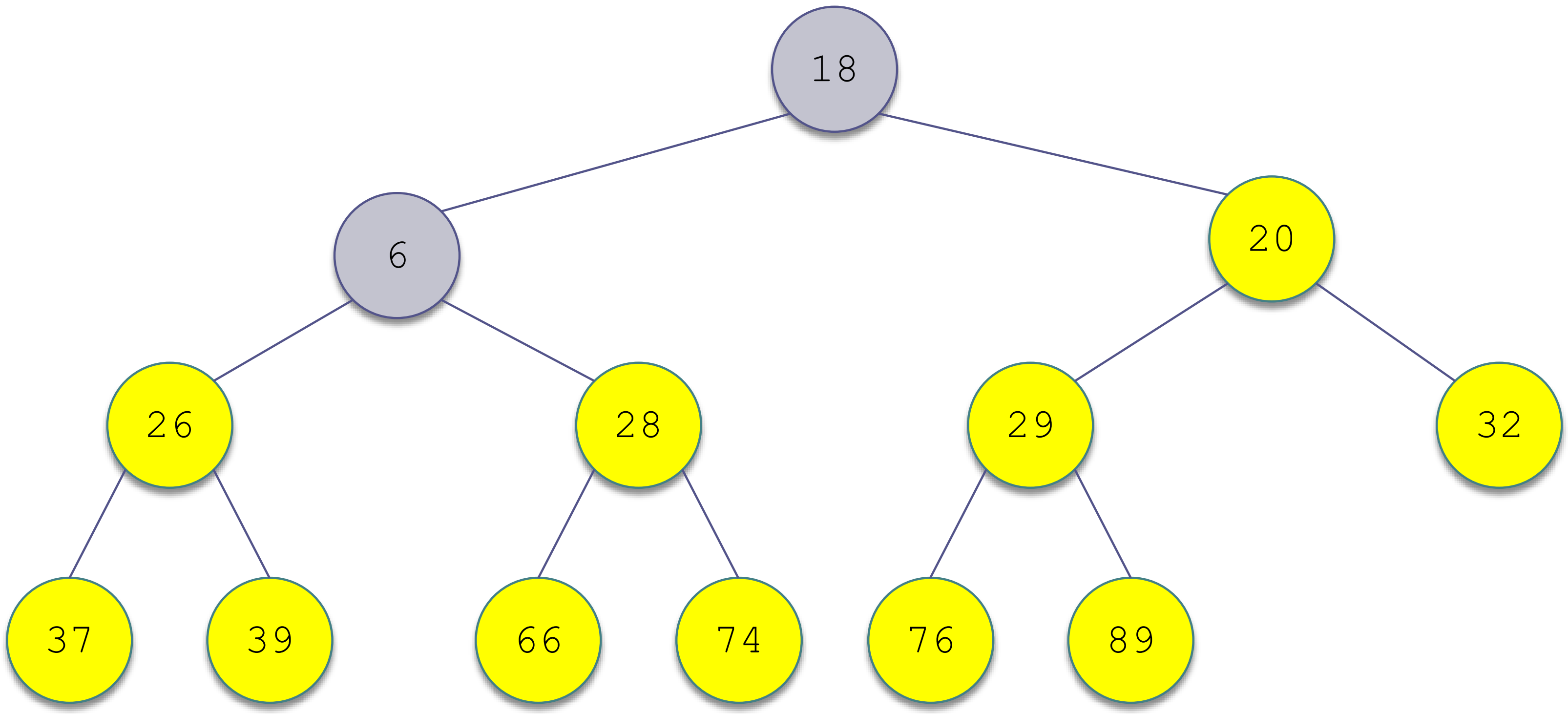
Trace of Heapsort (cont.)



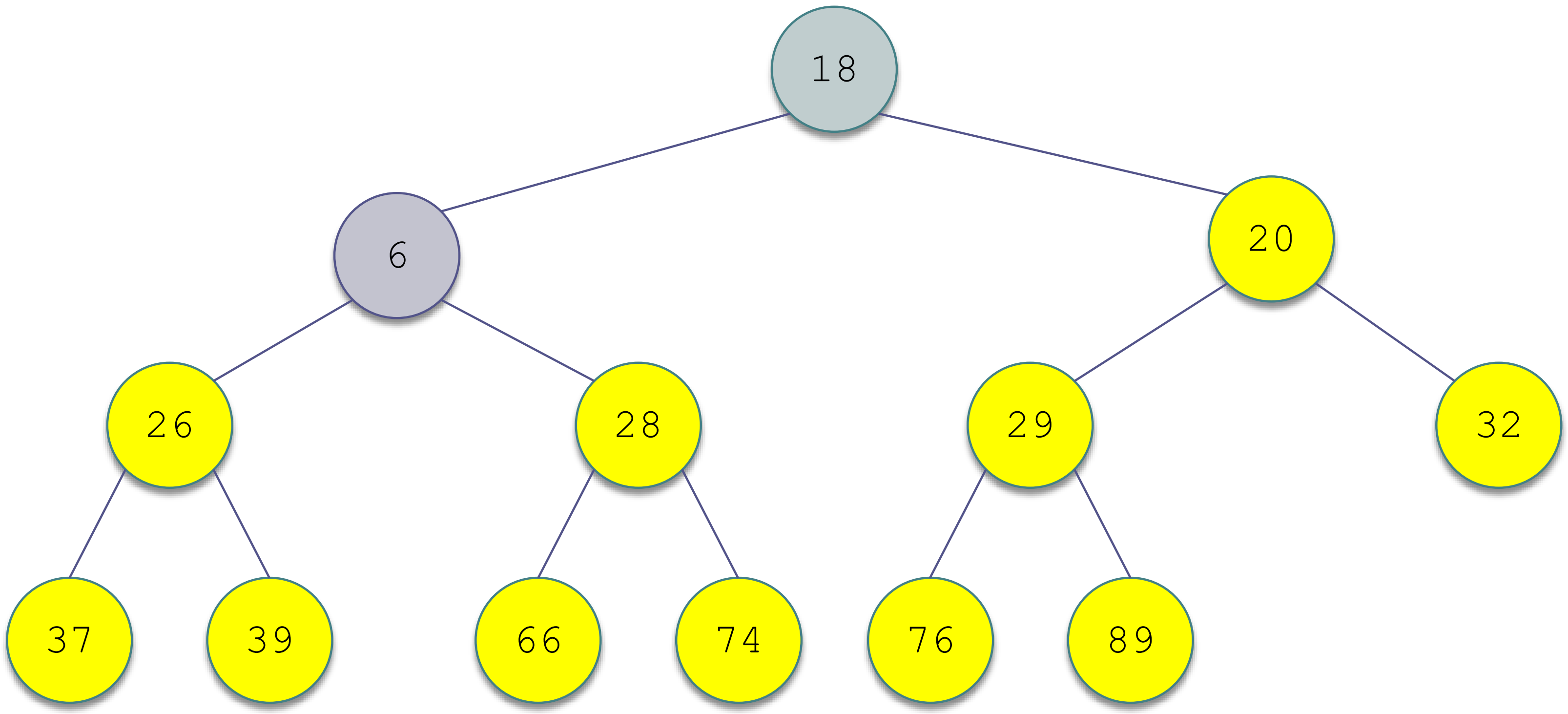
Trace of Heapsort (cont.)



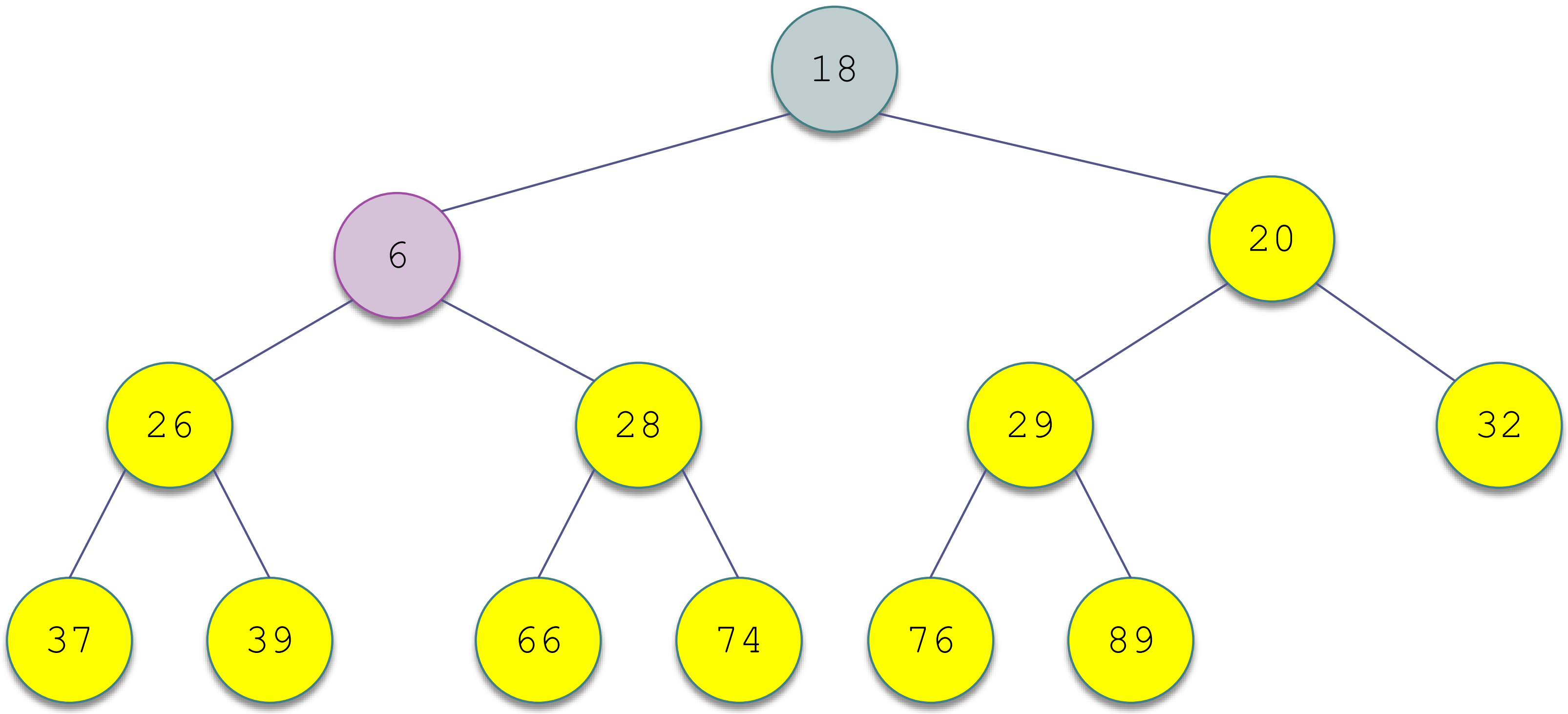
Trace of Heapsort (cont.)



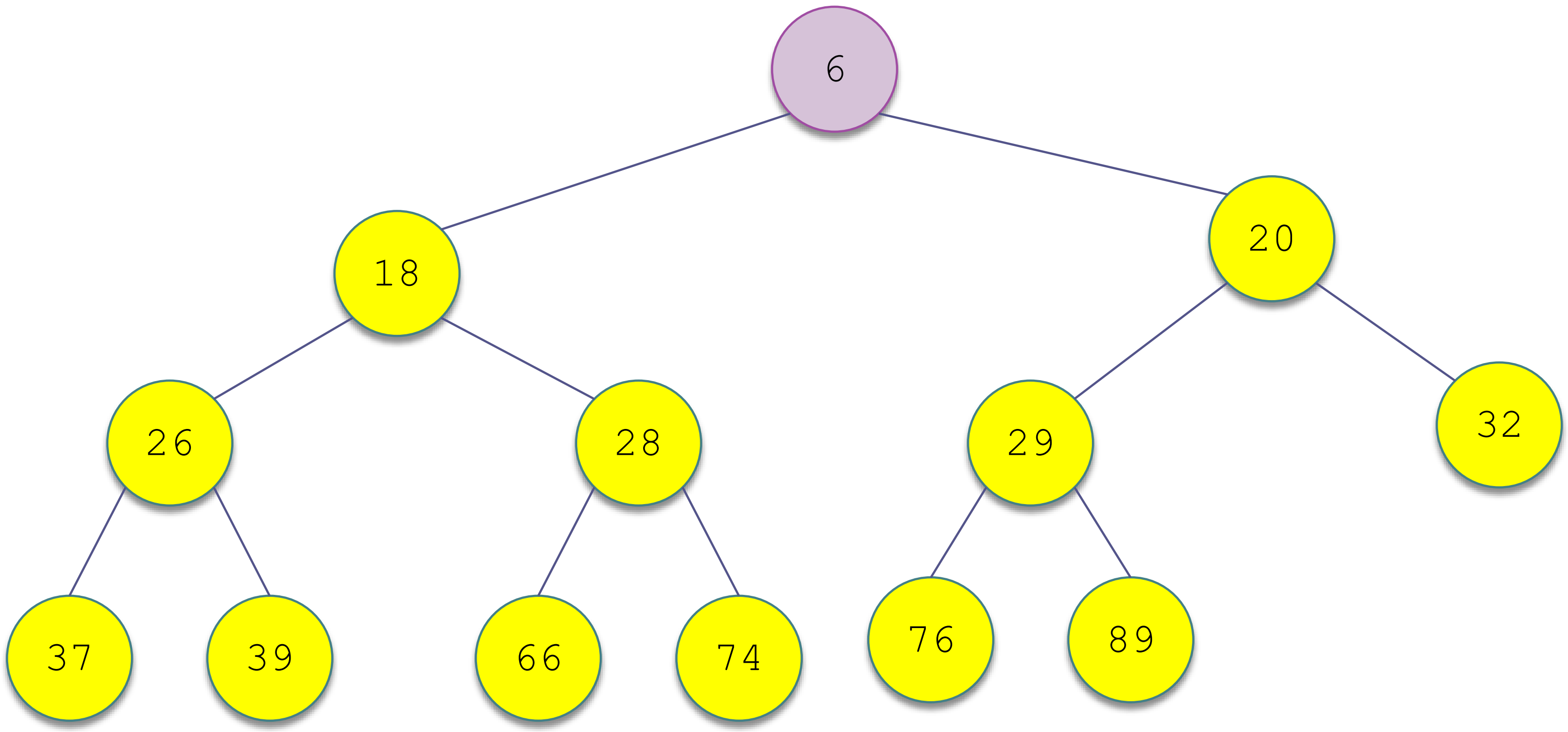
Trace of Heapsort (cont.)



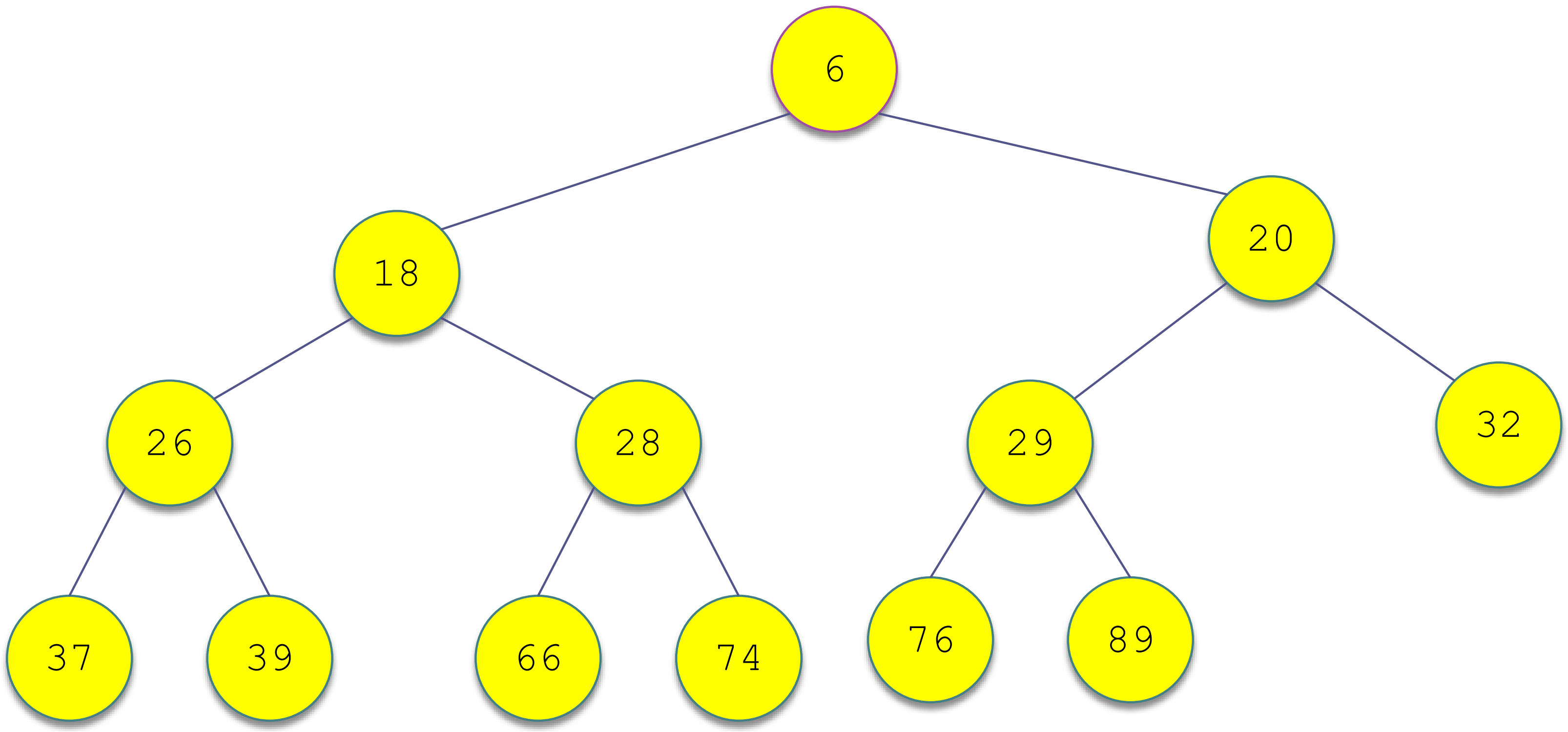
Trace of Heapsort (cont.)



Trace of Heapsort (cont.)



Trace of Heapsort (cont.)



Revising the Heapsort Algorithm

- If we implement the heap as an array
  - each element removed will be placed at the end of the array, and
  - the heap part of the array decreases by one element

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
89	76	74	37	32	39	66	20	26	18	28	29	6

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
76	37	74	26	32	39	66	20	6	18	28	29	89

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
74	37	66	26	32	39	29	20	6	18	28	76	89

⋮

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
6	18	20	26	28	29	32	37	39	66	74	76	89

Algorithm for In-Place Heapsort

Algorithm for In-Place Heapsort

1. **Build a heap by rearranging the elements in an unsorted array**
2. **`while` the heap is not empty**
3. **Remove the first item from the heap by swapping it with the last item in the heap and restoring the heap property**

Algorithm to Build a Heap

- Start with an array `table` of length `table.length`
- Consider the first item to be a heap of one item
- Next, consider the general case where the items in array `table` from `0` through `n-1` form a heap and the items from `n` through `table.length – 1` are not in the heap

Algorithm to Build a Heap (cont.)

Refinement of Step 1 for In-Place Heapsort

1.1 **`while` n is less than `table.length`**

1.2 **Increment n by 1. This inserts a new item into the heap**

1.3 **Restore the heap property**

Analysis of Heapsort

- Because a heap is a complete binary tree it has exactly log *n* levels
- Building a heap of size *n* requires finding the correct location for an item in a heap with log *n* levels
- Each insert (or remove) is O(log *n*)
- With *n* items, building a heap is O(*n* log *n*)
- No extra storage is needed

Code for Heapsort

- Listing 8.7 (`HeapSort.java`, pages 449 - 451)

Quicksort

Quicksort (1962)

- Quicksort selects a specific value called a pivot and rearranges the array into two parts (called *partioning*)
  - all the elements in the left subarray are less than or equal to the pivot
  - all the elements in the right subarray are larger than the pivot
  - The pivot is placed between the two subarrays
- The process is repeated recursively with both subarrays until the whole array is sorted

Trace of Quicksort

44	75	23	43	55	12	64	77	33

Trace of Quicksort (cont.)

44	75	23	43	55	12	64	77	33

Arbitrarily select the first element as the pivot

Trace of Quicksort (cont.)

55	75	23	43	44	12	64	77	33

Swap the pivot with the element in the middle

Trace of Quicksort (cont.)

55	75	23	43	44	12	64	77	33

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right

Trace of Quicksort (cont.)

12	33	23	43	44	55	64	77	75

Partition the elements so that all values less than or equal to the pivot are to the left, and all values greater than the pivot are to the right

Quicksort Example(cont.)

44 is now in its correct position

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

12	33	23	43	44	55	64	77	75

Now apply quicksort recursively to the two subarrays

Trace of Quicksort (cont.)

Pivot value = 12

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 12

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	33	23	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Left and right subarrays have single values; they are sorted

Trace of Quicksort (cont.)

Pivot value = 33

12	23	33	43	44	55	64	77	75

Left and right subarrays have single values; they are sorted

Trace of Quicksort (cont.)

Pivot value = 55

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 64

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 77

12	23	33	43	44	55	64	77	75

Trace of Quicksort (cont.)

Pivot value = 77

12	23	33	43	44	55	64	75	77

Trace of Quicksort (cont.)

Pivot value = 77

12	23	33	43	44	55	64	75	77

Trace of Quicksort (cont.)

12	23	33	43	44	55	64	75	77

Left subarray has single value; it is sorted

Trace of Quicksort (cont.)

12	23	33	43	44	55	64	75	77

Algorithm for Quicksort

- We will get to the partitioning later
- The indexes `first` and `last` are the end points of the array being sorted
- The index of the pivot after partitioning is `pivIndex`

Algorithm for Quicksort

1. **`if first < last` then**
2. **Partition the elements in the subarray `first . . . last` so that the pivot value is in its correct place (subscript `pivIndex`)**
3. **Recursively apply quicksort to the subarray `first . . . pivIndex - 1`**
4. **Recursively apply quicksort to the subarray `pivIndex + 1 . . . last`**

Analysis of Quicksort

- If the pivot value is a random value selected from the current subarray,
  - then statistically half of the items in the subarray will be less than the pivot and half will be greater
- If both subarrays have the same number of elements (best case), there will be log *n* levels of recursion (why?)
- At each recursion level, the partitioning process involves moving every element to its correct position—*n* moves
- Quicksort is O(*n* log *n*), just like the merge sort and heap sort

Analysis of Quicksort

- *Quicksort* will give very poor behavior O(n^2) if, each time the array is partitioned, one subarray is empty.
- Under these circumstances, the overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of *quicksort* a poor performer relative to the quadratic sorts

Code for Quicksort

- Listing 8.8 (`QuickSort.java`, pages 453 - 454)

Algorithm for Partitioning

44	75	23	43	55	12	64	77	33

If the array is randomly ordered, it does not matter which element is the pivot.

We pick the element with subscript `first = 0`

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

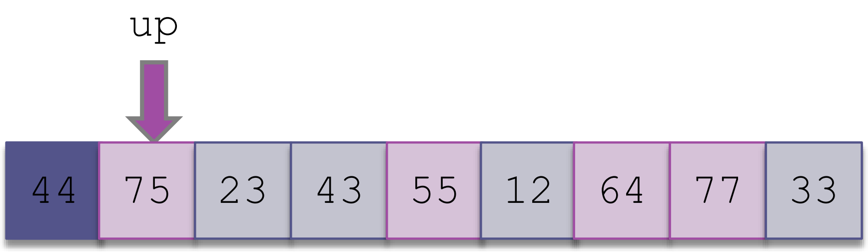
Items that are less than or equal to the pivot are colored gray; items greater than the pivot are colored purple

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

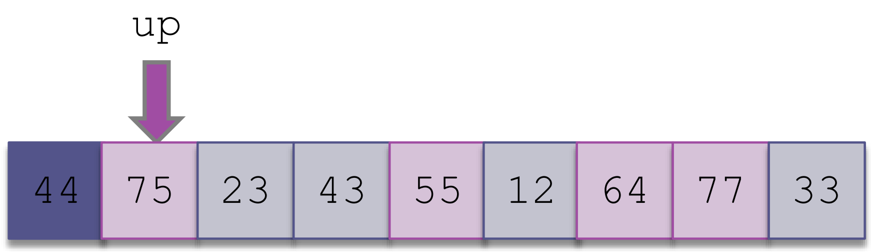
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



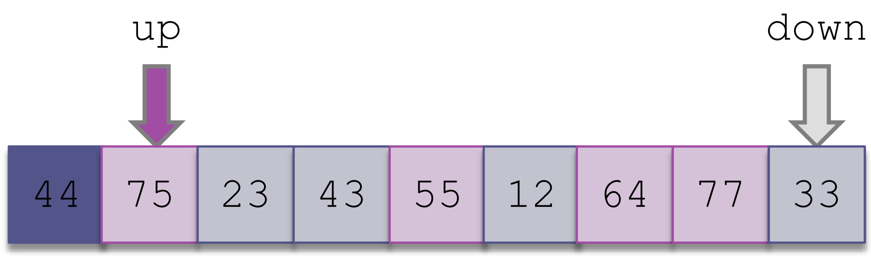
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



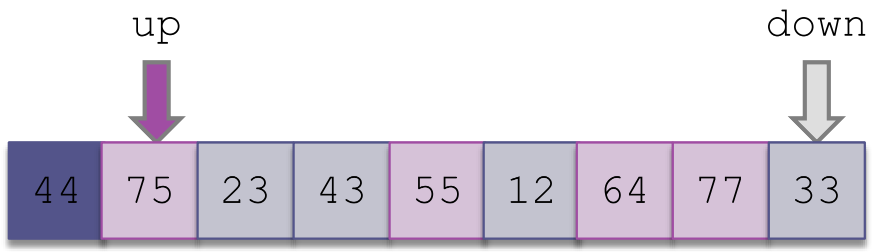
Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)



Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)



Exchange these values

Trace of Partitioning (cont.)

44	33	23	43	55	12	64	77	75

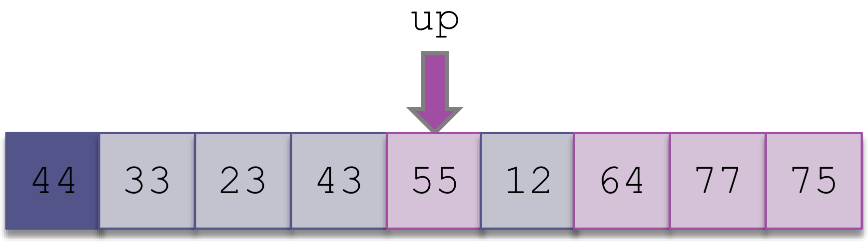
Exchange these values

Trace of Partitioning (cont.)

44	33	23	43	55	12	64	77	75

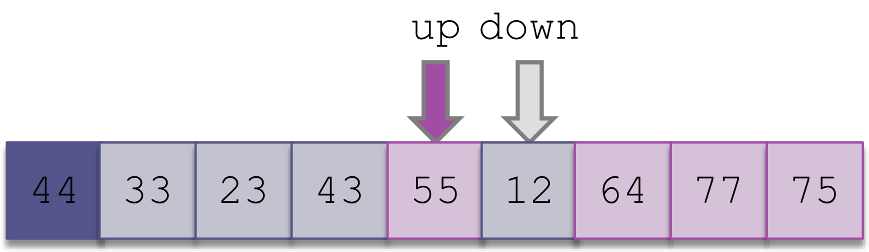
Repeat

Trace of Partitioning (cont.)



Find first value at left end greater than pivot

Trace of Partitioning (cont.)



Find first value at right end less than or equal to pivot

Trace of Partitioning (cont.)

44	33	23	43	12	55	64	77	75

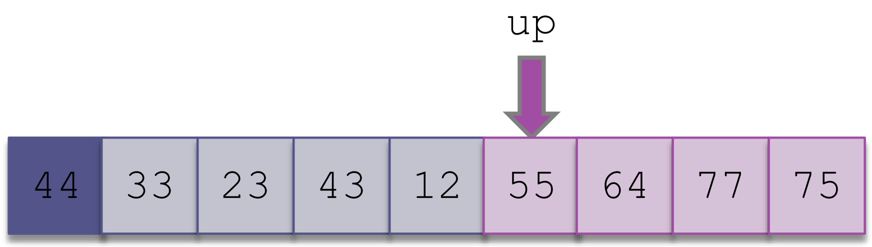
Exchange

Trace of Partitioning (cont.)

44	33	23	43	12	55	64	77	75

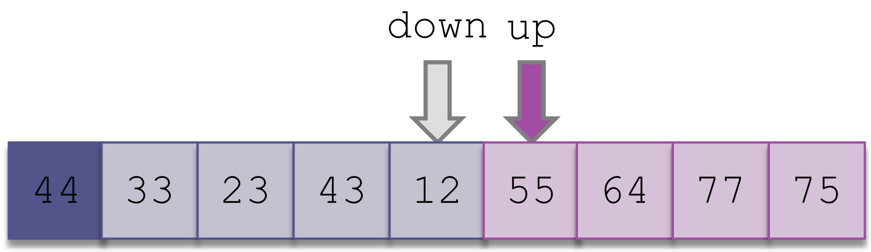
Repeat

Trace of Partitioning (cont.)



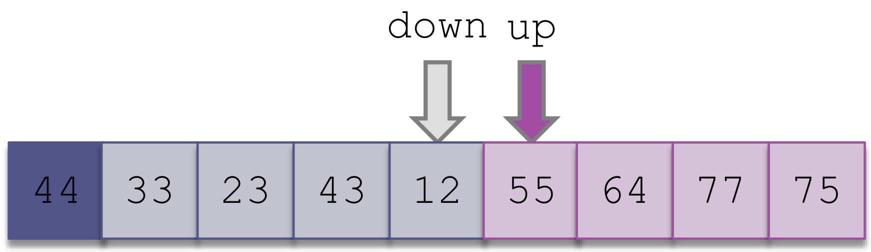
Find first element at left end greater than pivot

Trace of Partitioning (cont.)



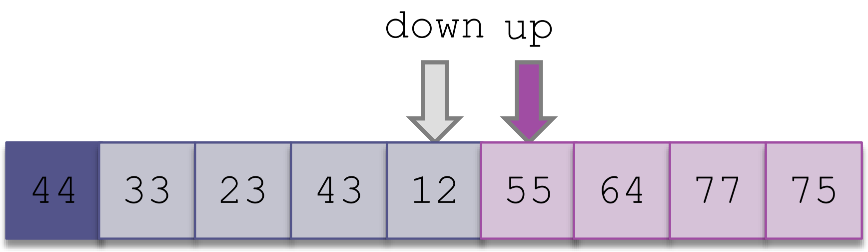
Find first element at right end less than or equal to pivot

Trace of Partitioning (cont.)



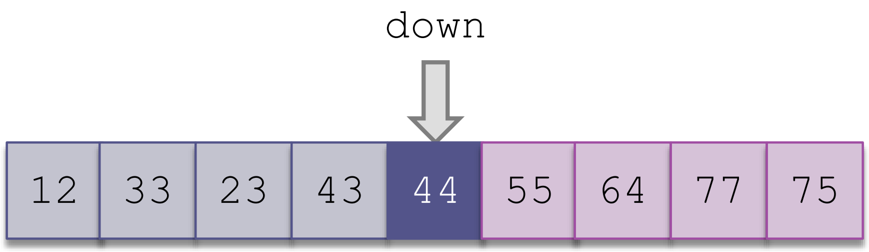
Since `down` has "passed" `up`, do not exchange

Trace of Partitioning (cont.)



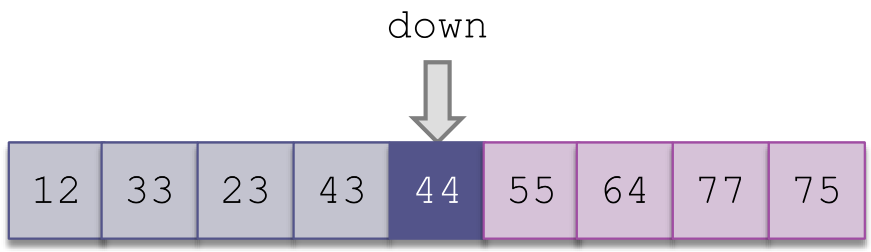
Exchange the pivot value with the value at `down`

Trace of Partitioning (cont.)



Exchange the pivot value with the value at `down`

Trace of Partitioning (cont.)



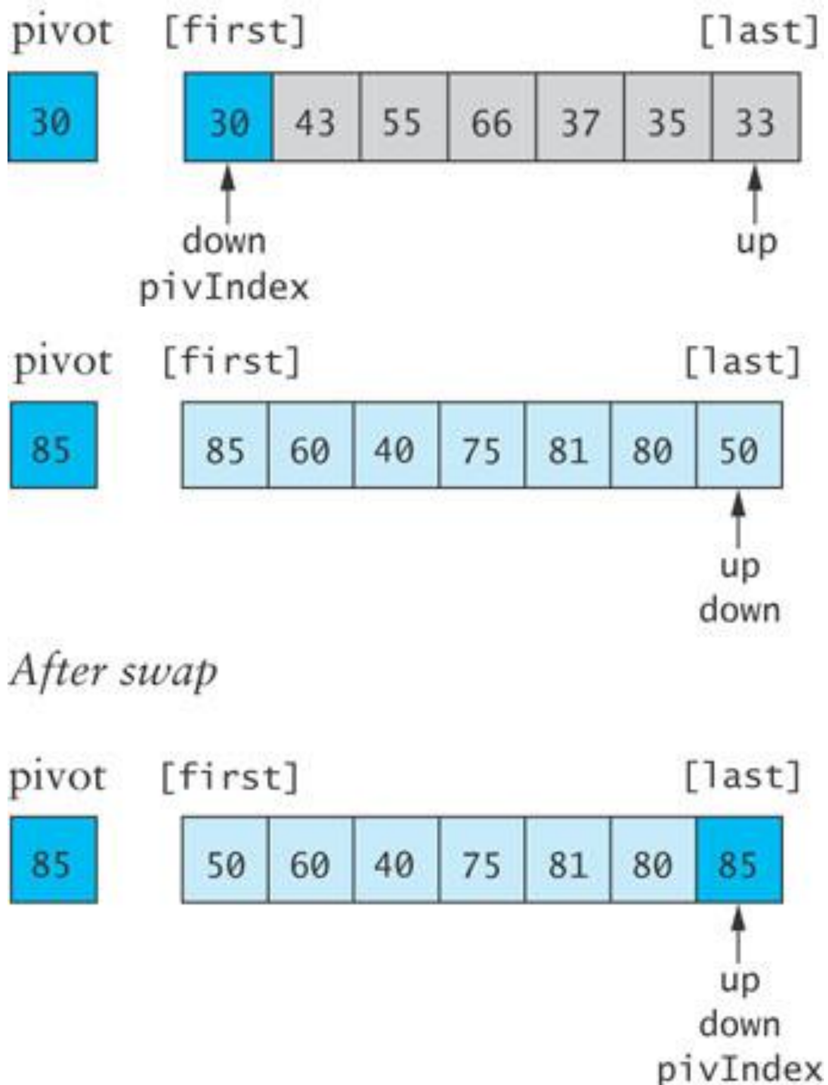
The pivot value is in the correct position; return the value of `down` and assign it to the pivot index `pivIndex`

Algorithm for Partitioning

Algorithm for partition Method

1. Define the pivot value as the contents of `table[first]`.
2. Initialize `up` to `first` and `down` to `last`.
3. `do`
4. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`.
5. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`.
6. `if up < down then`
7. Exchange `table[up]` and `table[down]`.
8. `while up` is to the left of `down`
9. Exchange `table[first]` and `table[down]`.
10. Return the value of `down` to `pivIndex`.

Code for `partition`



Code for `partition` (cont.)

- Listing 8.9 (`QuickSort1`, page 457)

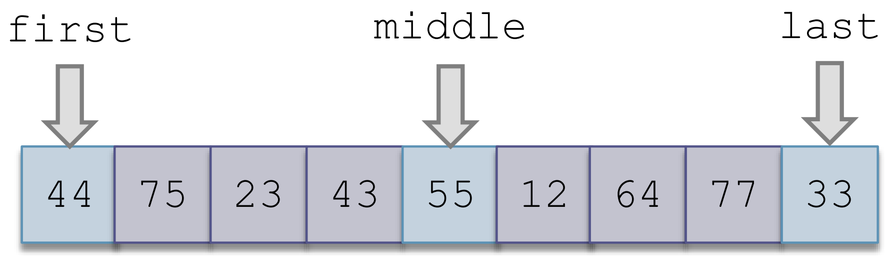
Revised Partition Algorithm

- Quicksort is O(n^2) when each split yields one empty subarray, which is the case when the array is presorted
- A better solution is to pick the pivot value in a way that is less likely to lead to a bad split
  - Use three references: `first`, `middle`, `last`
  - Select the median of the these items as the pivot

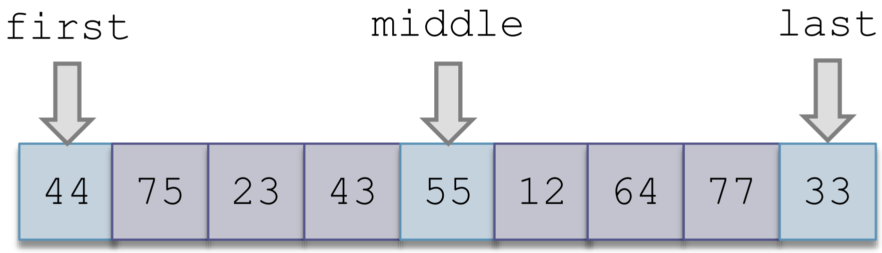
Trace of Revised Partitioning

44	75	23	43	55	12	64	77	33

Trace of Revised Partitioning (cont.)

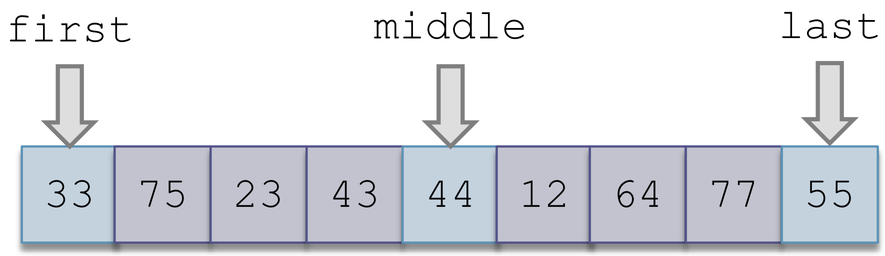


Trace of Revised Partitioning (cont.)



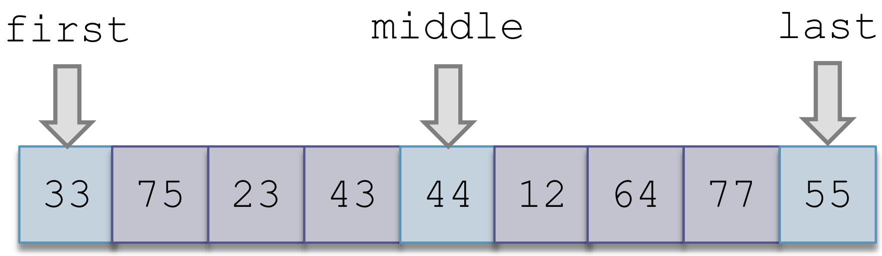
Sort these values

Trace of Revised Partitioning (cont.)



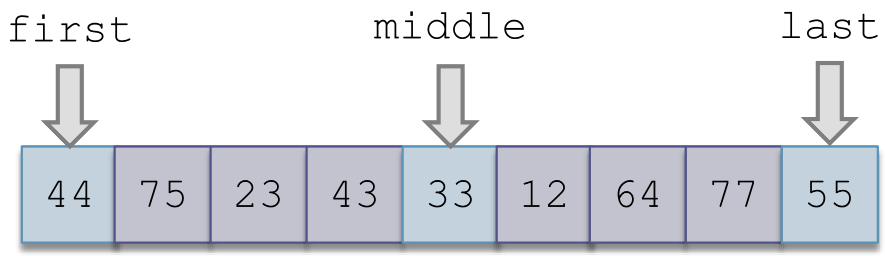
Sort these values

Trace of Revised Partitioning (cont.)



Exchange middle with first

Trace of Revised Partitioning (cont.)



Exchange middle with first

Trace of Revised Partitioning (cont.)

44	75	23	43	33	12	64	77	55

Run the partition algorithm using the first element as the pivot

Algorithm for Revised `partition` Method

Algorithm for Revised partition Method

1. Sort `table[first]`, `table[middle]`, and `table[last]`
2. Move the median value to `table[first]` (the pivot value) by exchanging `table[first]` and `table[middle]`.
3. Initialize `up` to `first` and `down` to `last`
4. `do`
5. Increment `up` until `up` selects the first element greater than the pivot value or `up` has reached `last`
6. Decrement `down` until `down` selects the first element less than or equal to the pivot value or `down` has reached `first`
7. `if up < down` then
8. Exchange `table[up]` and `table[down]`
9. `while up` is to the left of `down`
10. Exchange `table[first]` and `table[down]`
11. Return the value of `down` to `pivIndex`

Code for Revised `partition` Method

- Listing 8.10 (`QuickSort2`, page 459)