CMSC 132: Object-Oriented Programming II

Sorting

What Is Sorting?

- To arrange a collection of items in some specified order.
  - Numerical order
  - Lexicographical order
- ***Input:*** sequence $<a_1, a_2, \ldots, a_n>$ of numbers.
- ***Output:*** permutation $<a'_1, a'_2, \ldots, a'_n>$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.
- Example
  - Start → 1 23 2 56 9 8 10 100
  - End → 1 2 8 9 10 23 56 100

Why Sort?

- A classic problem in computer science.
  - Data requested in sorted order
    - e.g., list students in increasing GPA order
- Searching
  - To find an element in an array of a million elements
    - Linear search: average **500,000** comparisons
    - Binary search: worst case **20** comparisons
  - Database, Phone book
- Eliminating duplicate copies in a collection of records
- Finding a missing element, Max, Min

Sorting Algorithms

- **Selection Sort** **Bubble Sort**
- **Insertion Sort** **Shell Sort**
- $T(n)=O(n^2)$ **Quadratic growth**
- In clock time

10,000	**3 sec**	**20,000**	**17 sec**
50,000	77 sec	100,000	5 min

- Double input -> 4X time
  - Feasible for small inputs, quickly unmanagable
- Halve input -> 1/4 time
  - Hmm... can recursion save the day?
  - If have two sorted halves, how to produce sorted full result?

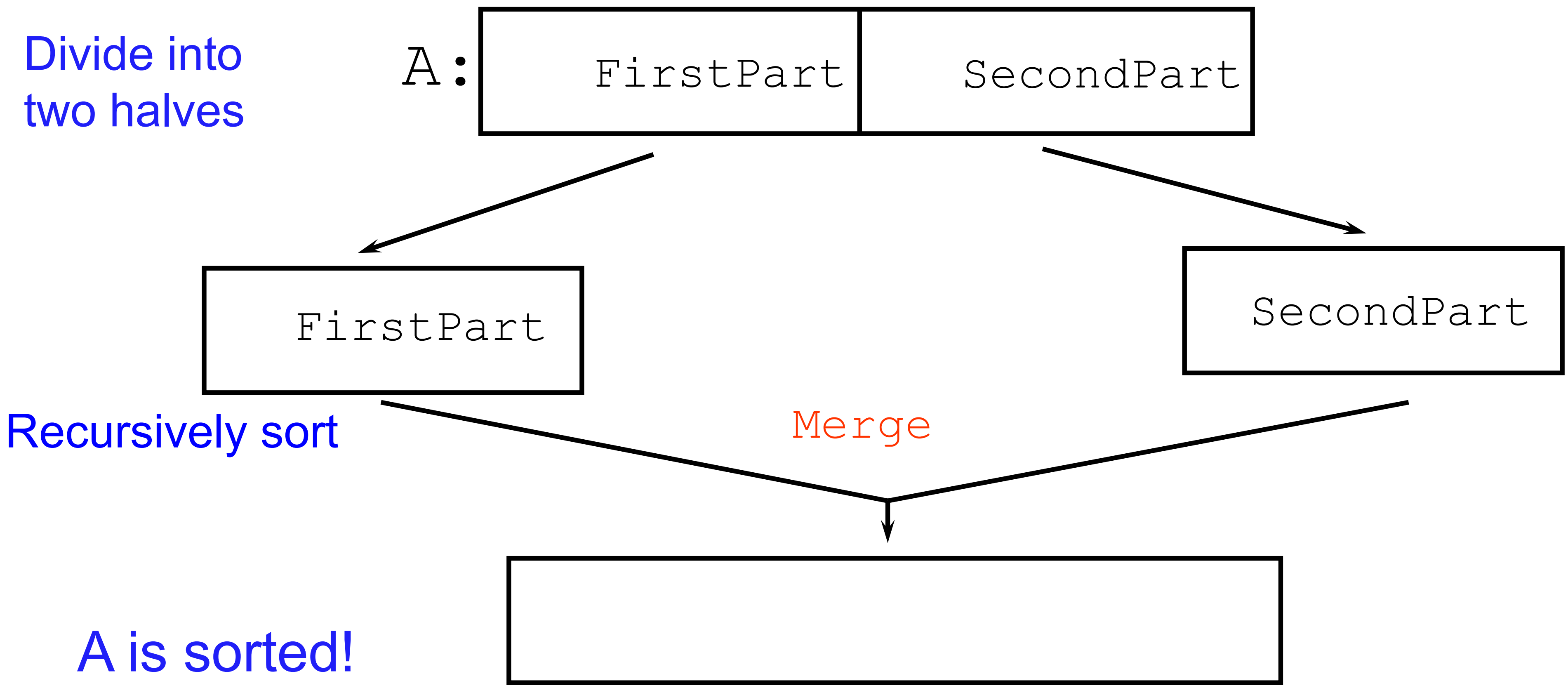
Divide and Conquer

1. Base case: the problem is small enough, solve directly
2. Divide the problem into two or more similar and smaller subproblems
3. Recursively solve the subproblems
4. Combine solutions to the subproblems

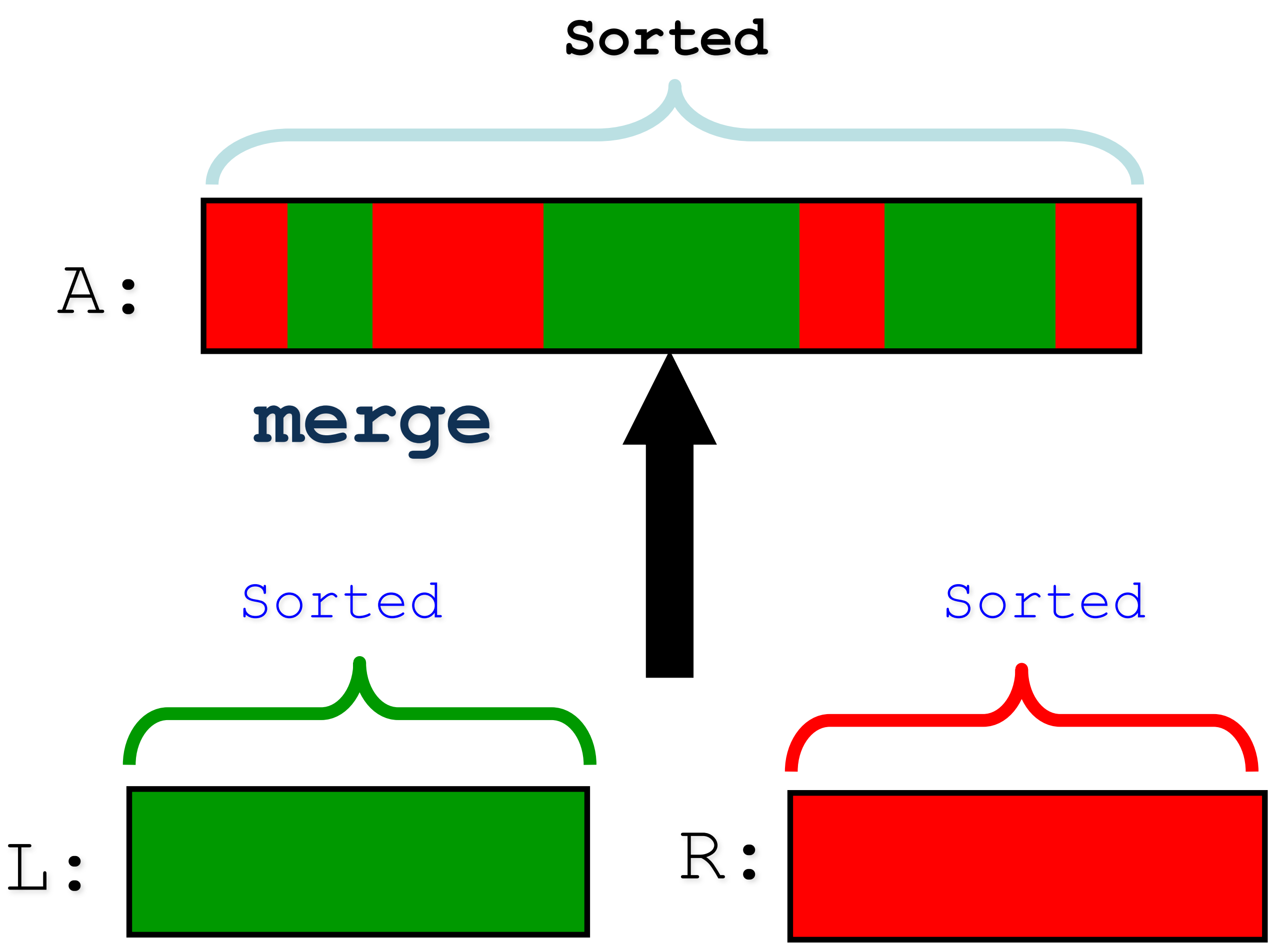
Merge Sort

- Divide and conquer algorithm
- Worst case: O(nlogn)
- Stable
  - maintain the relative order of records with equal values
  - Input: 12, 5, 8, 13, 8, 27
  - Stable: 5, 8, 8, 12, 13, 27
  - Not Stable:5, 8, 8, 12, 13, 27

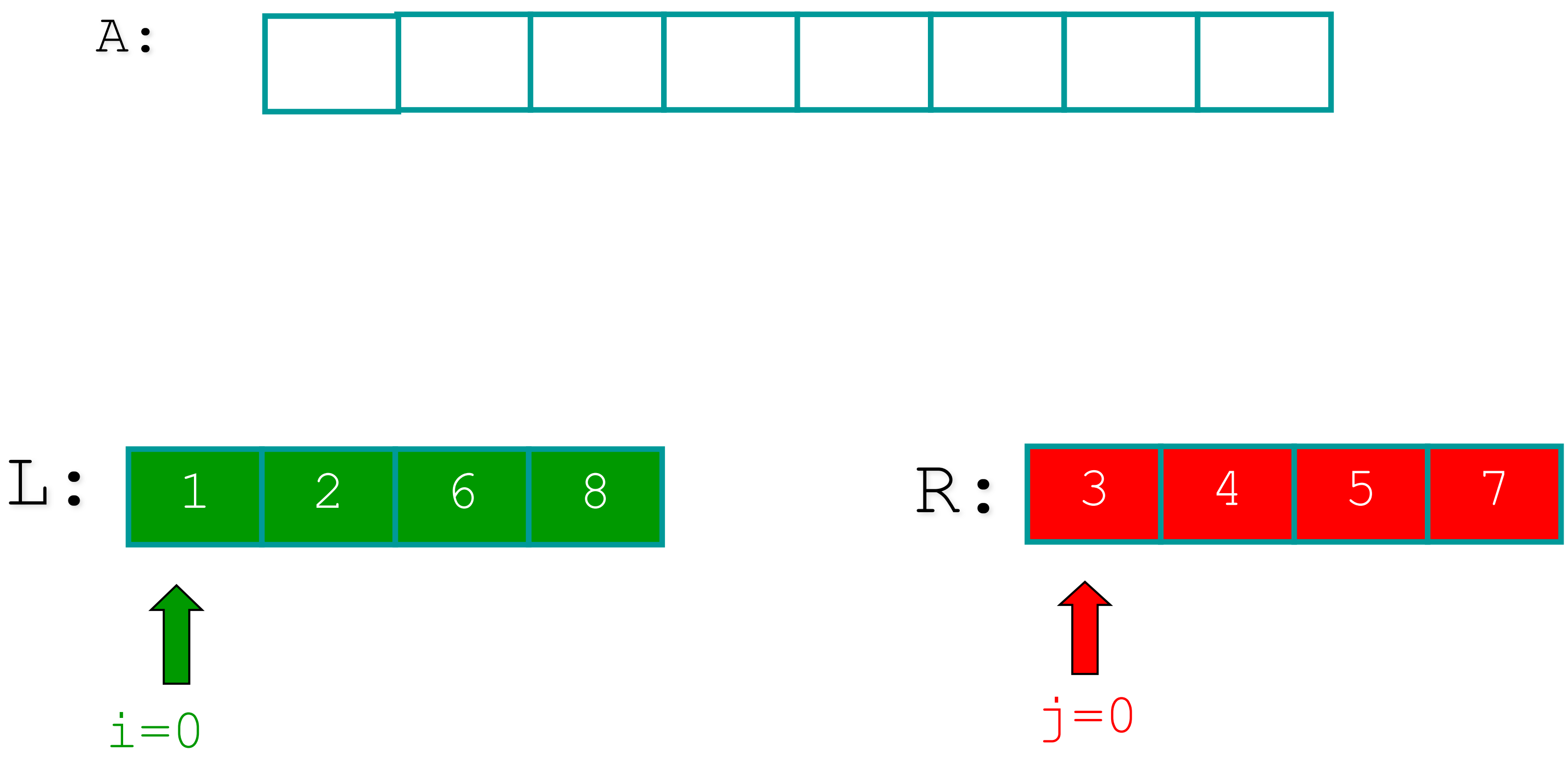
Merge Sort: Idea



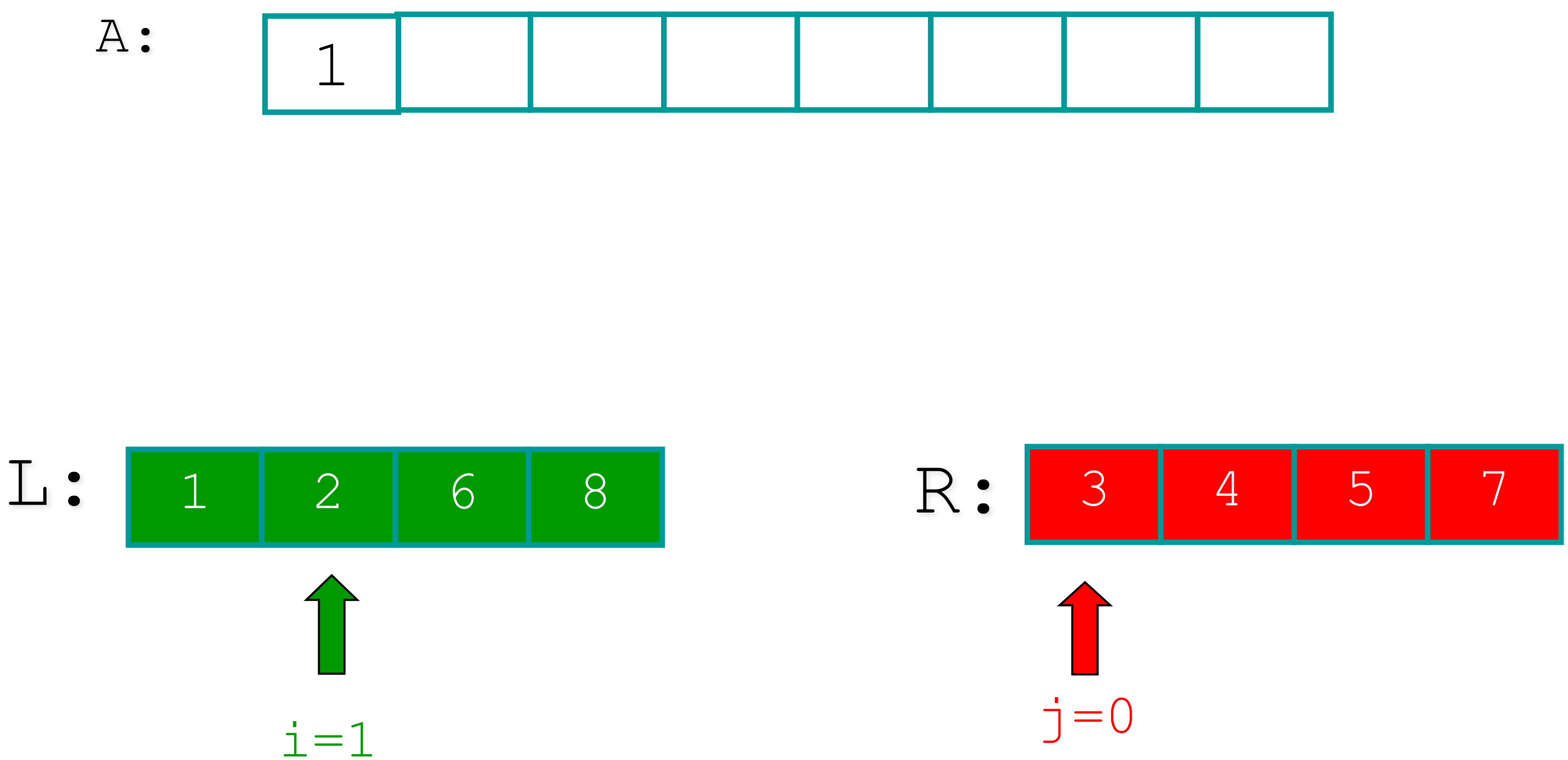
Merge-Sort: Merge



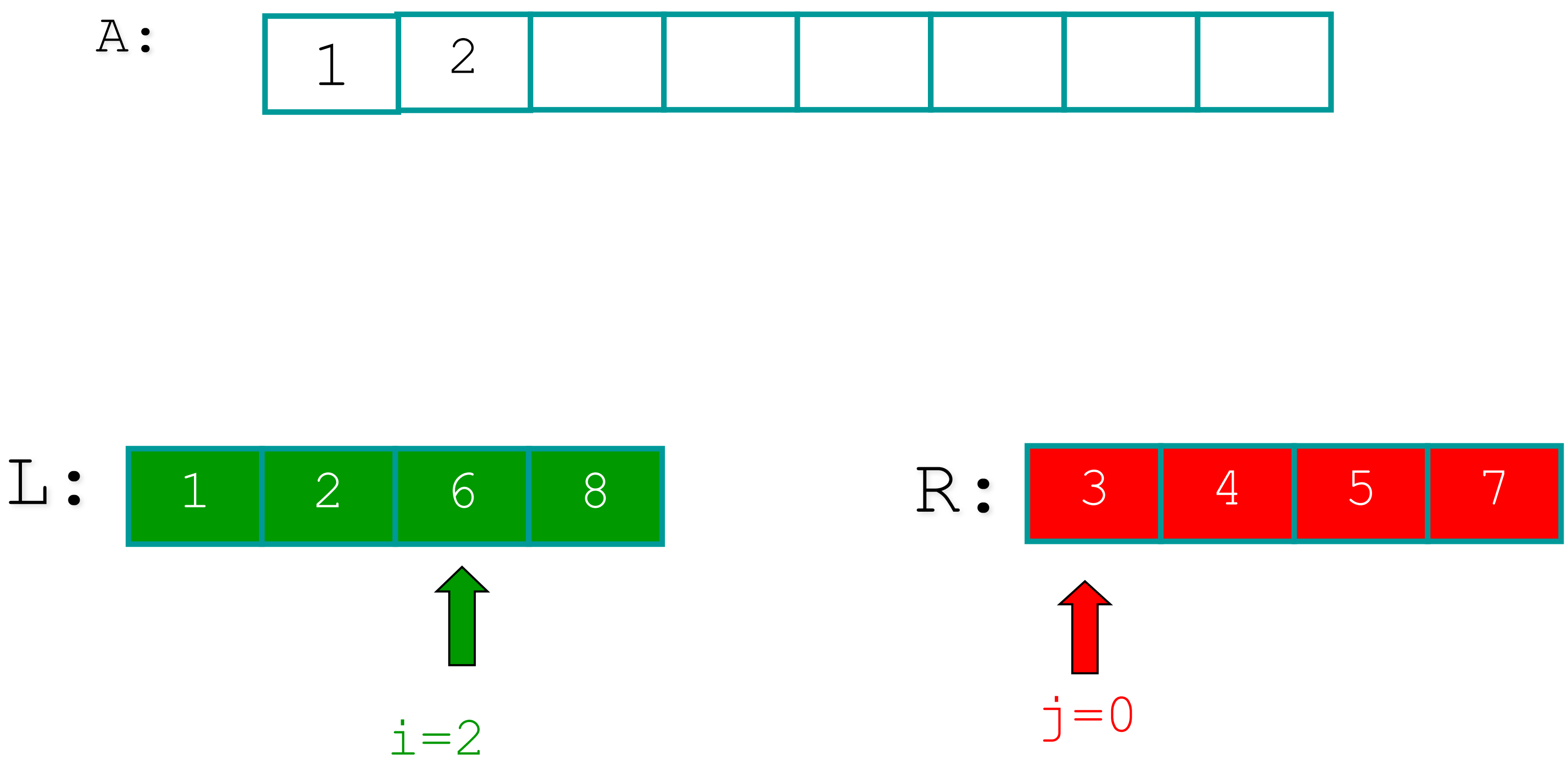
Merge Example



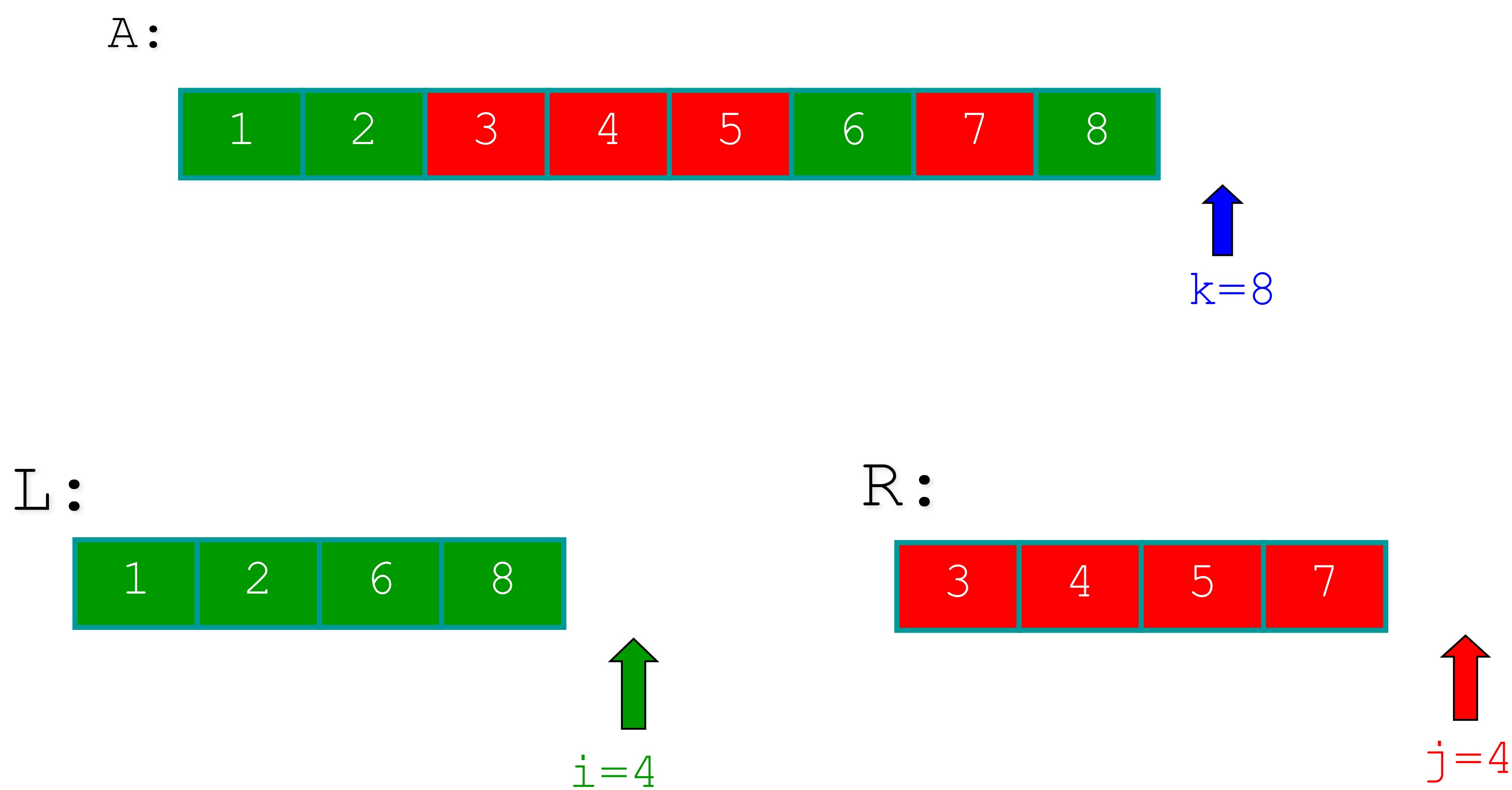
Merge Example



Merge Example



Merge Example cont.



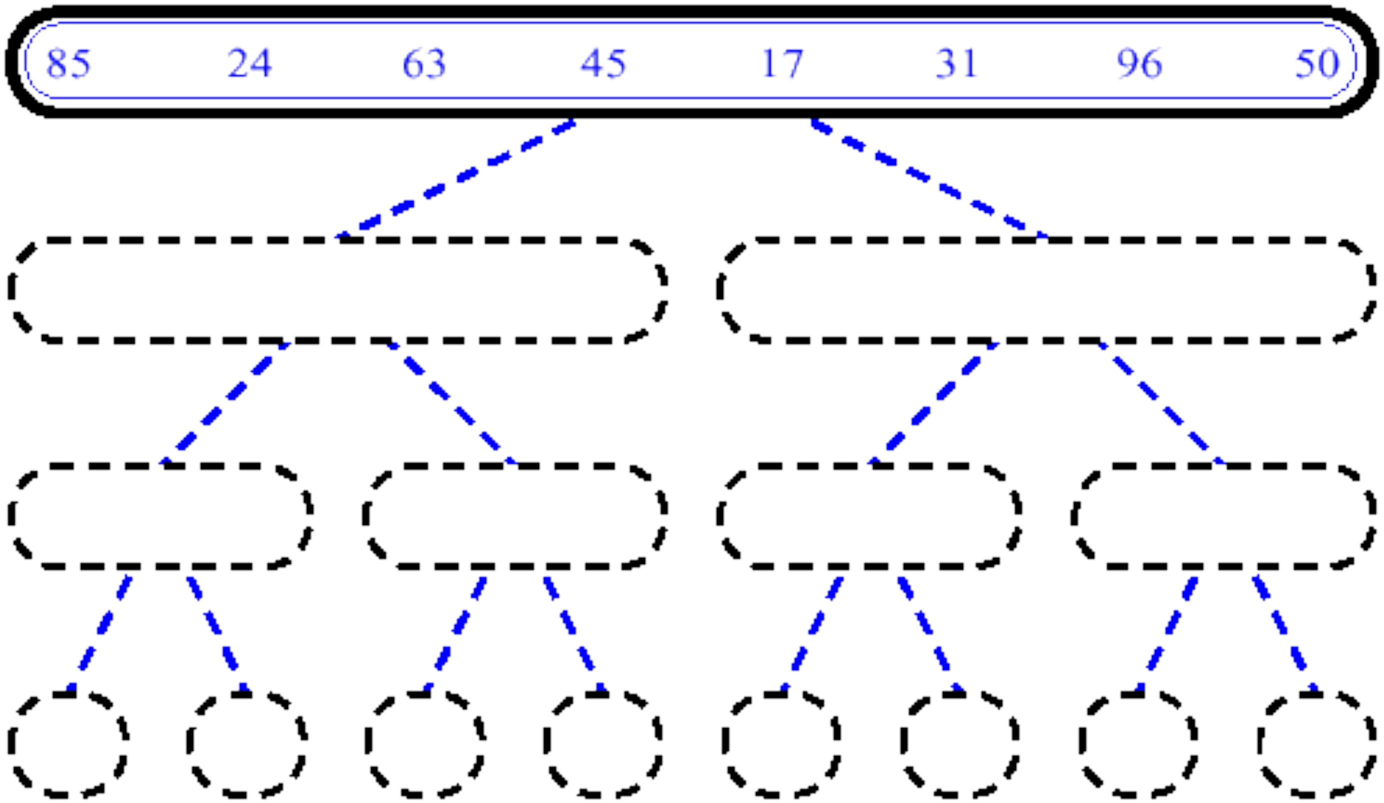
Merge sort algorithm

MERGE-SORT ***A*[1 . . *n*]**

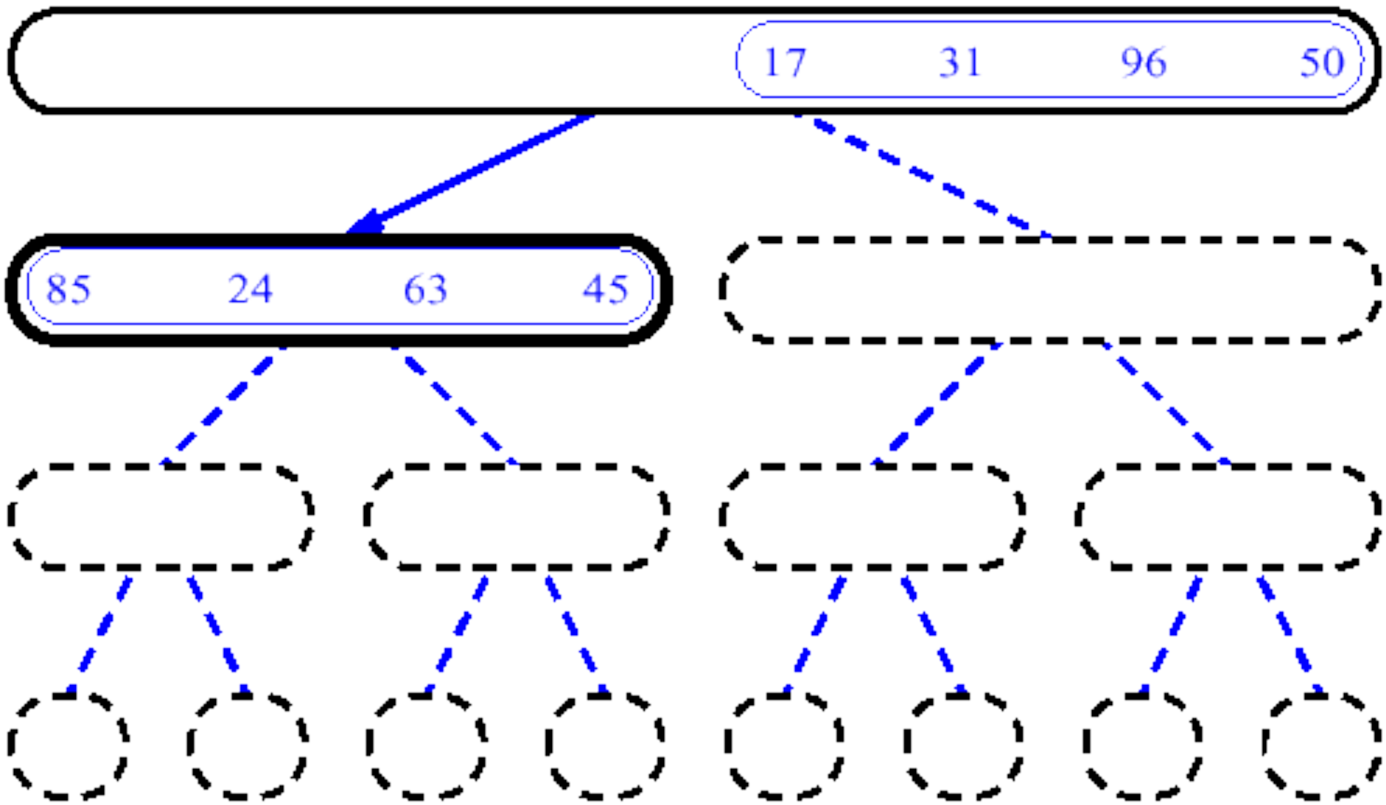
1. If $n = 1$, done.
2. Recursively sort $A[\,1\,.\,.\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil+1\,.\,.\,n\,]$.
3. "***Merge***" the 2 sorted lists.

Key subroutine: **MERGE**

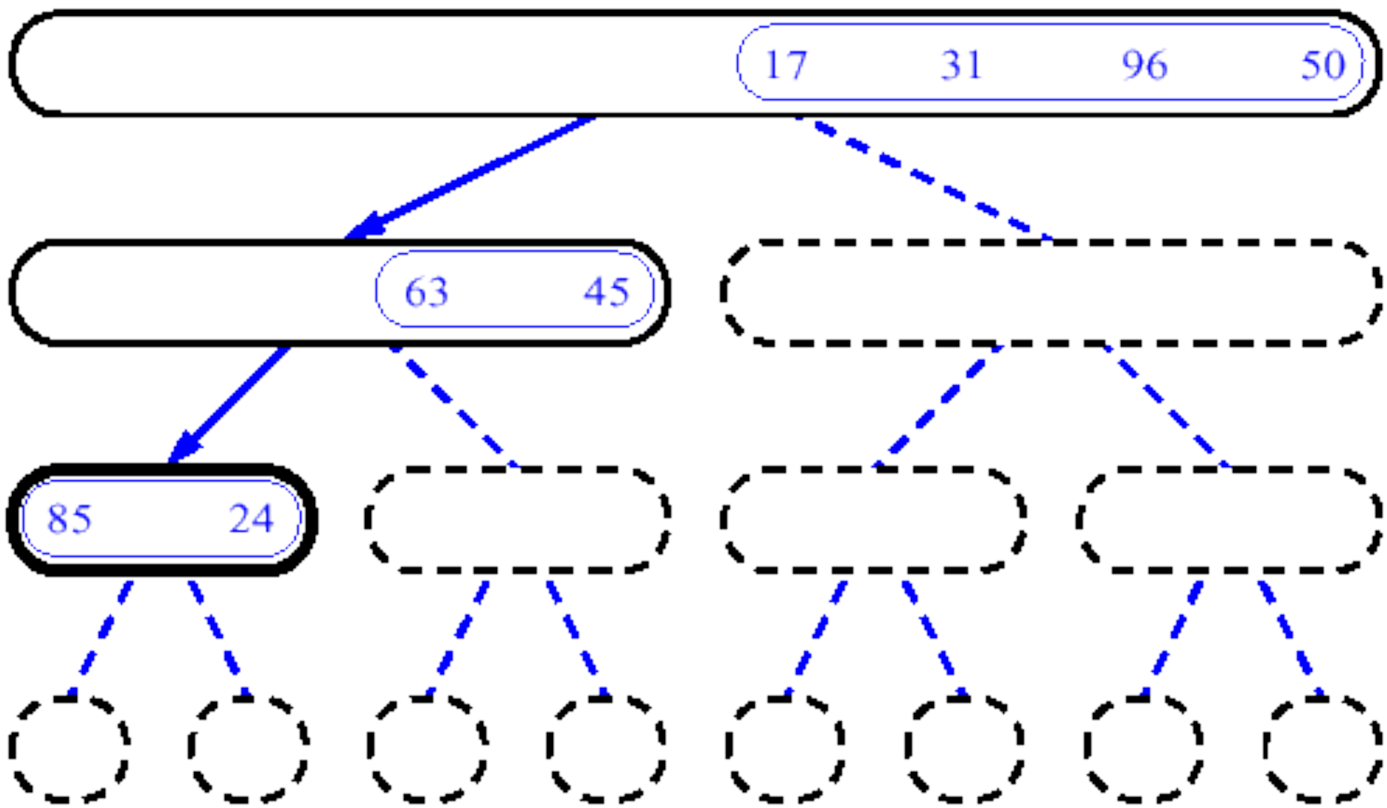
Merge sort (Example)



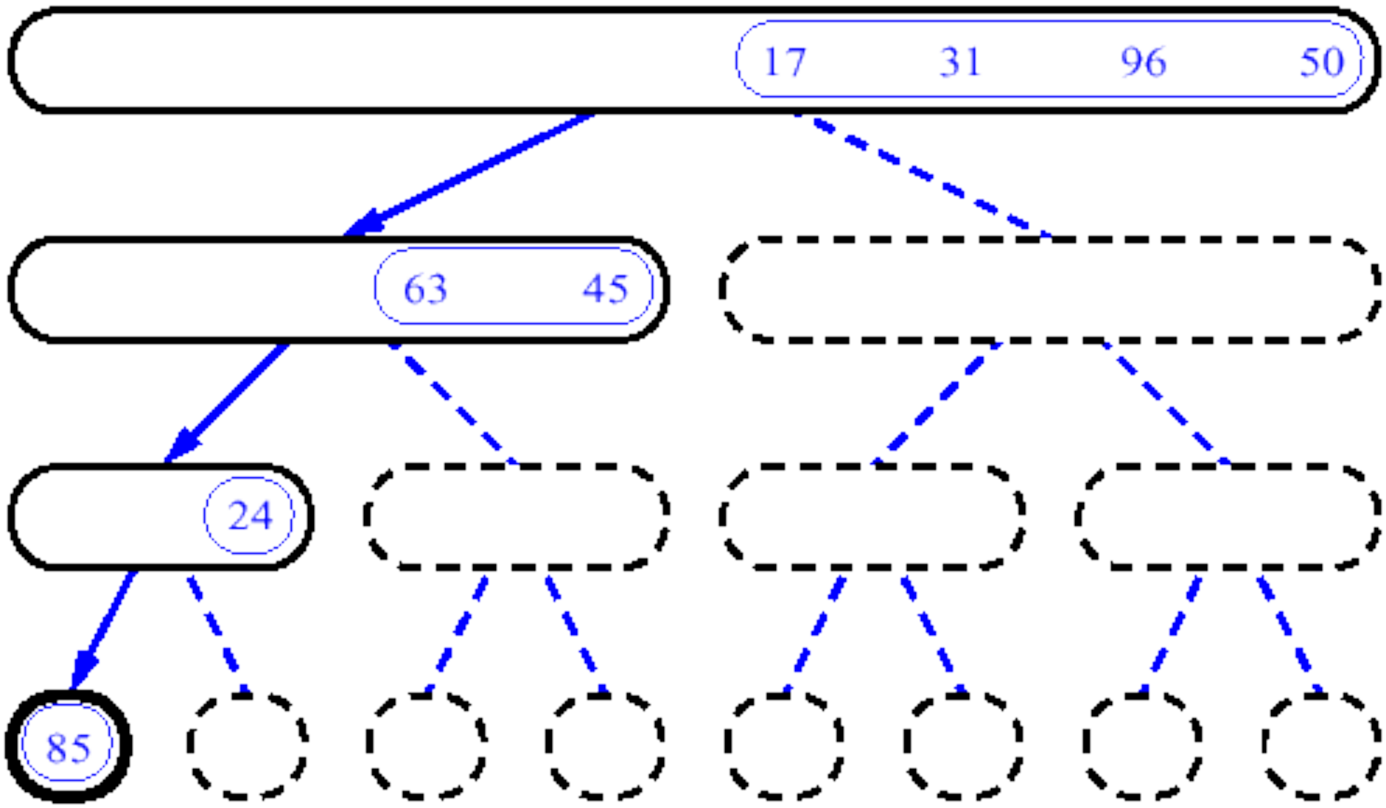
Merge sort (Example)



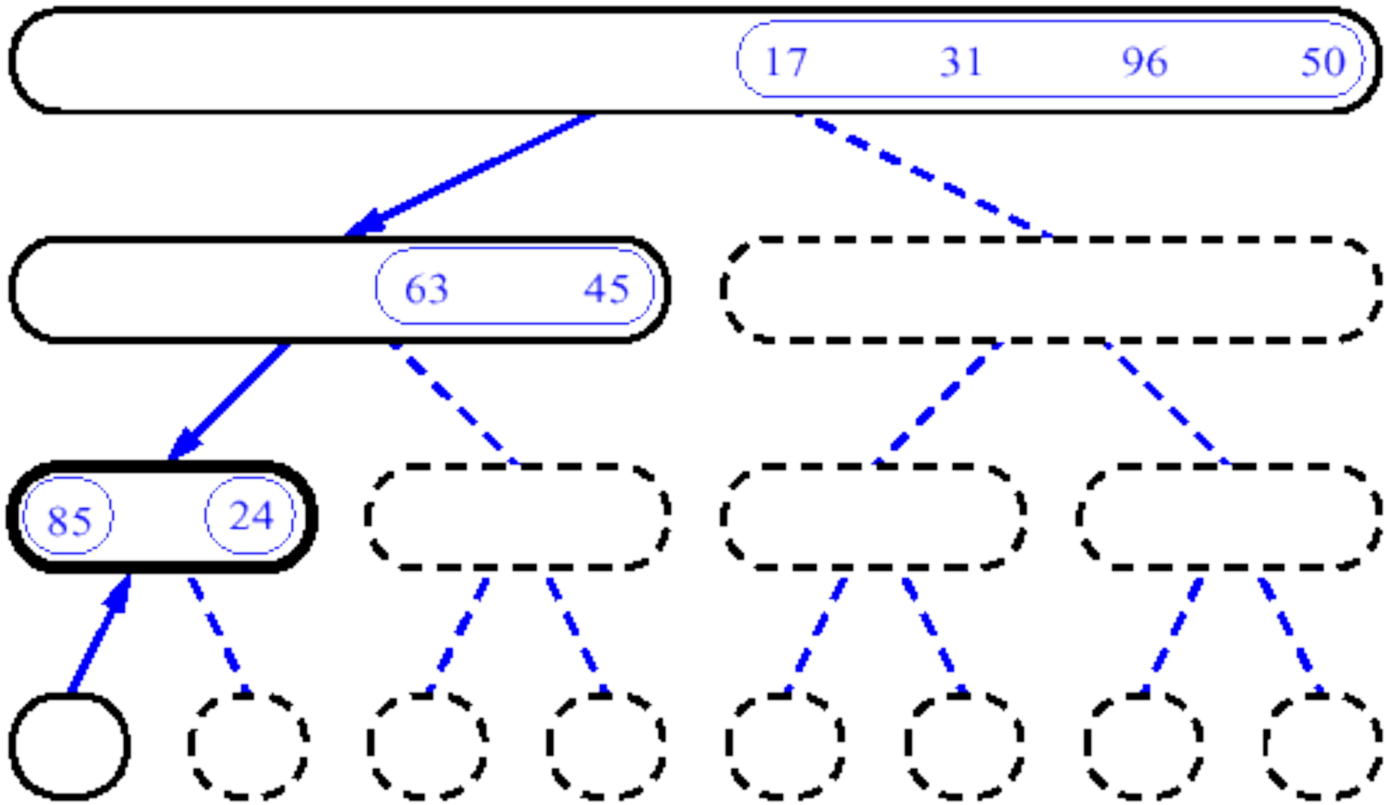
Merge sort (Example)



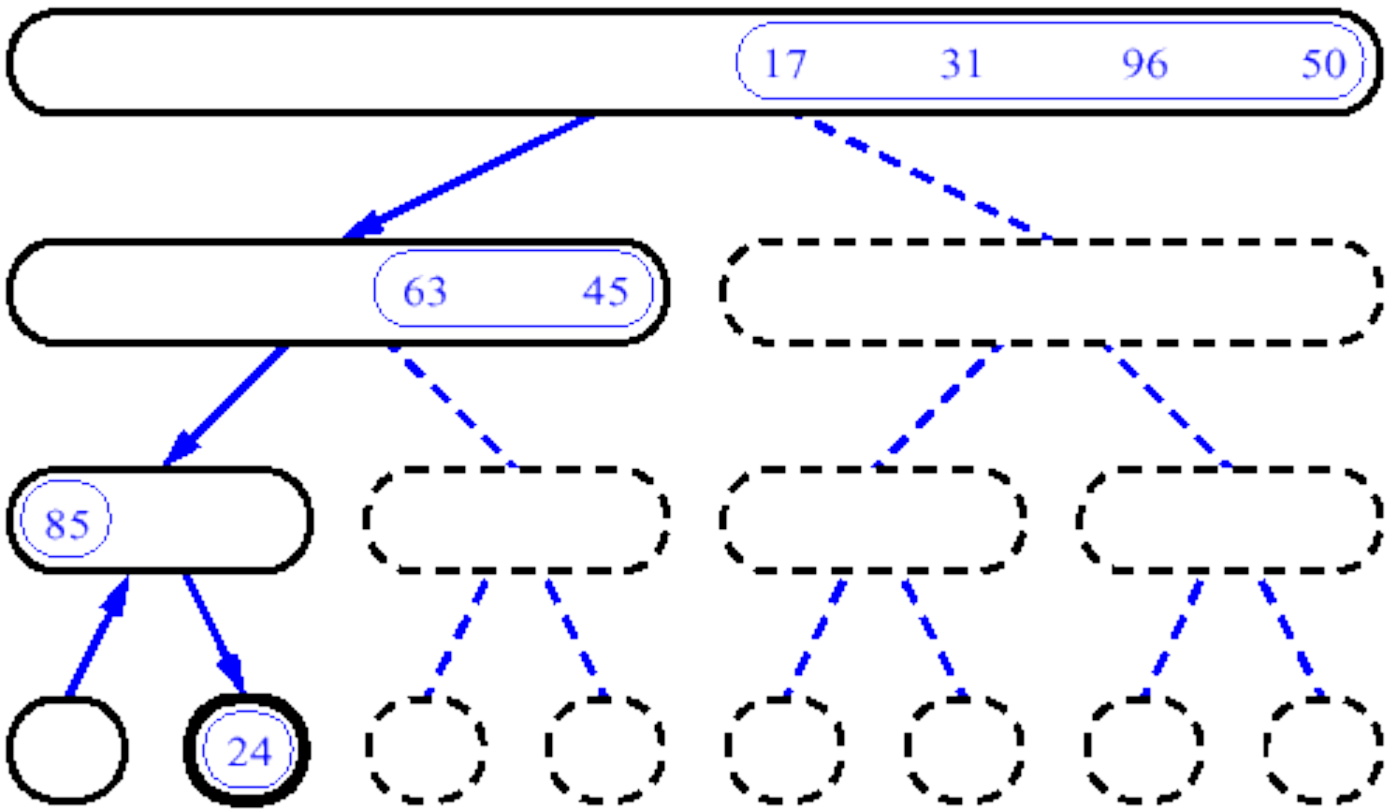
Merge sort (Example)



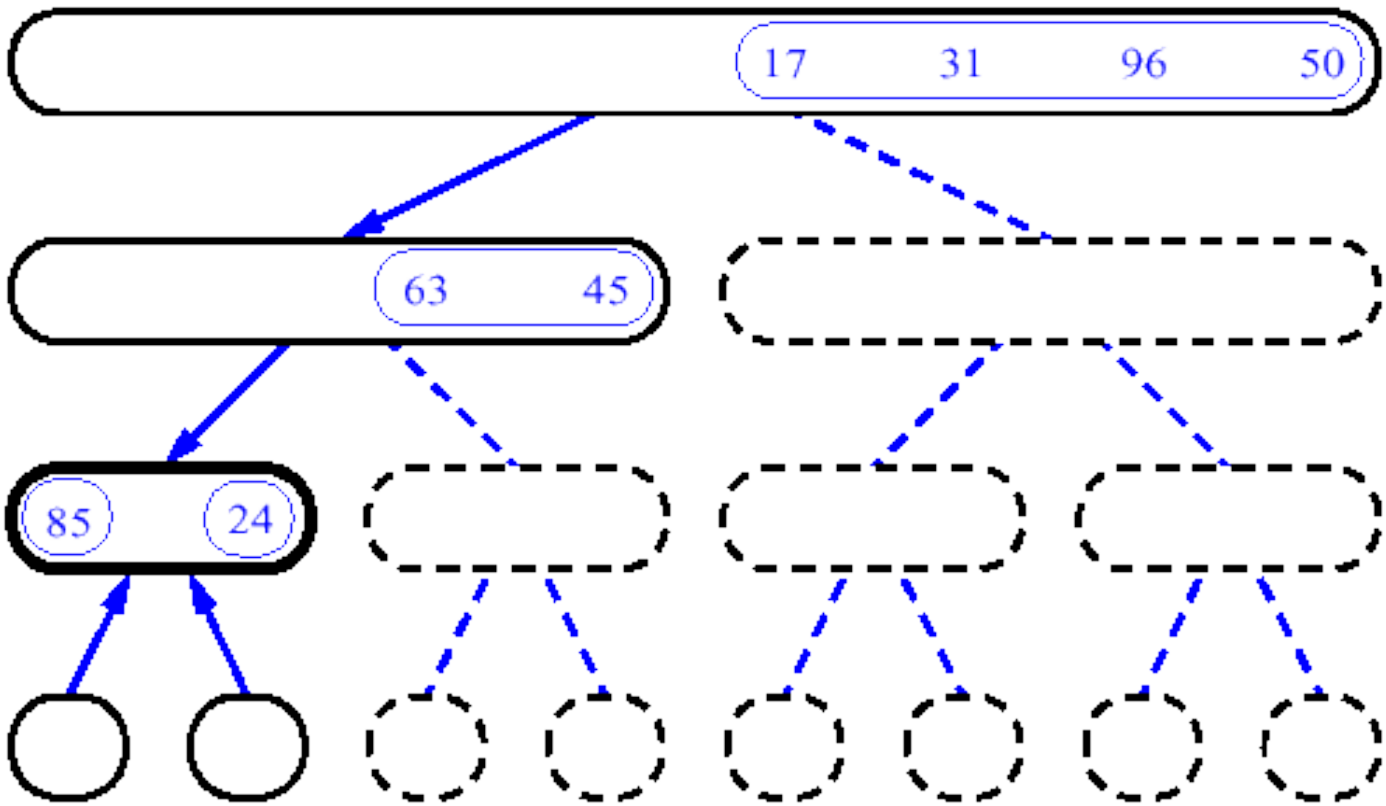
Merge sort (Example)



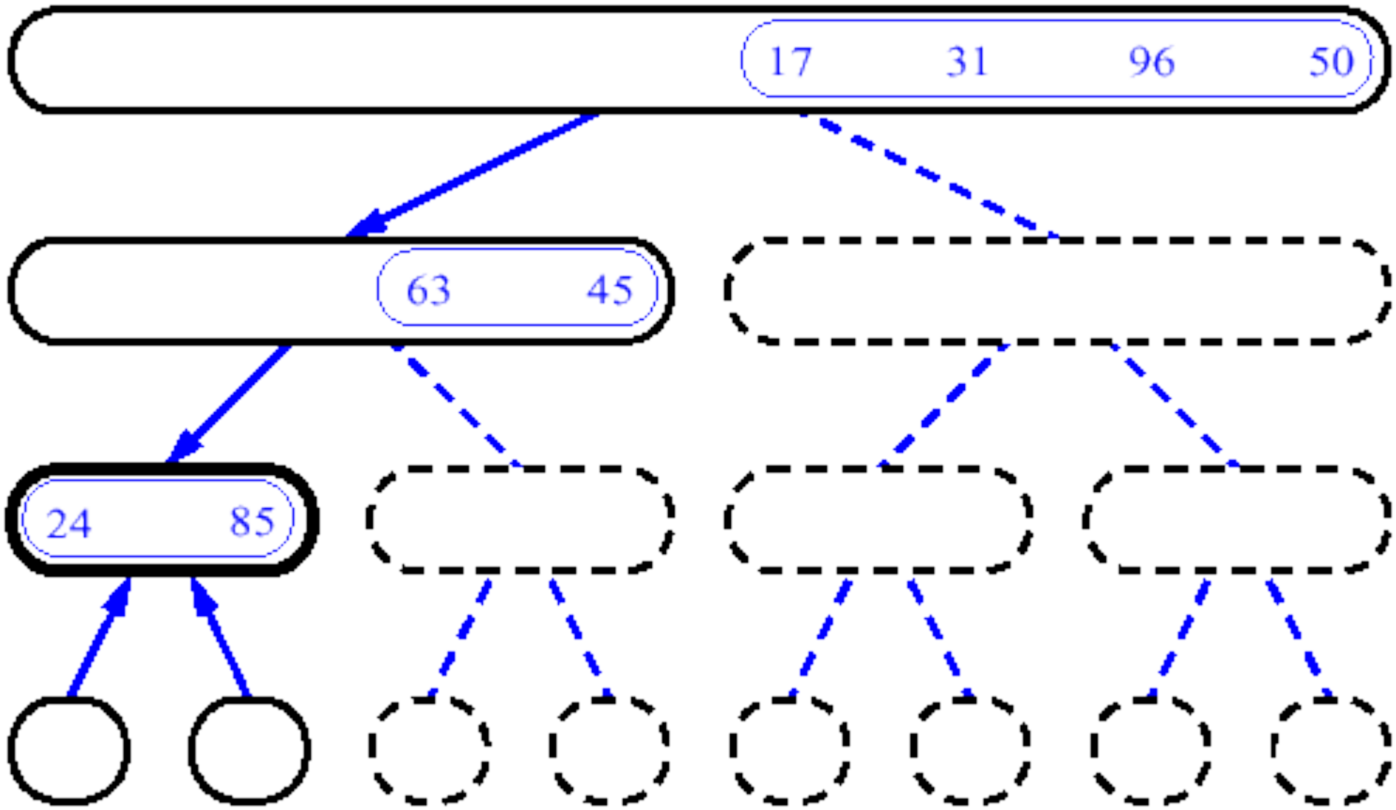
Merge sort (Example)



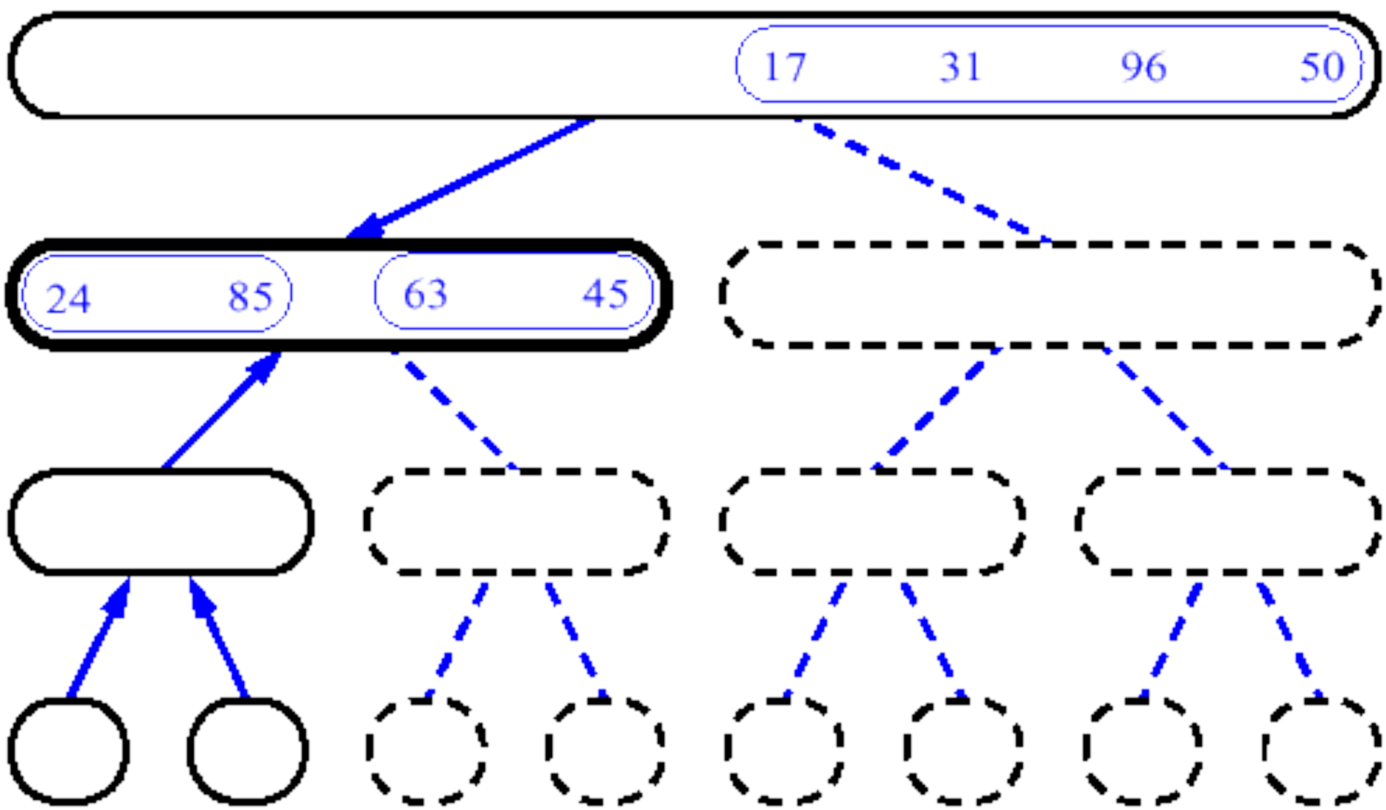
Merge sort (Example)



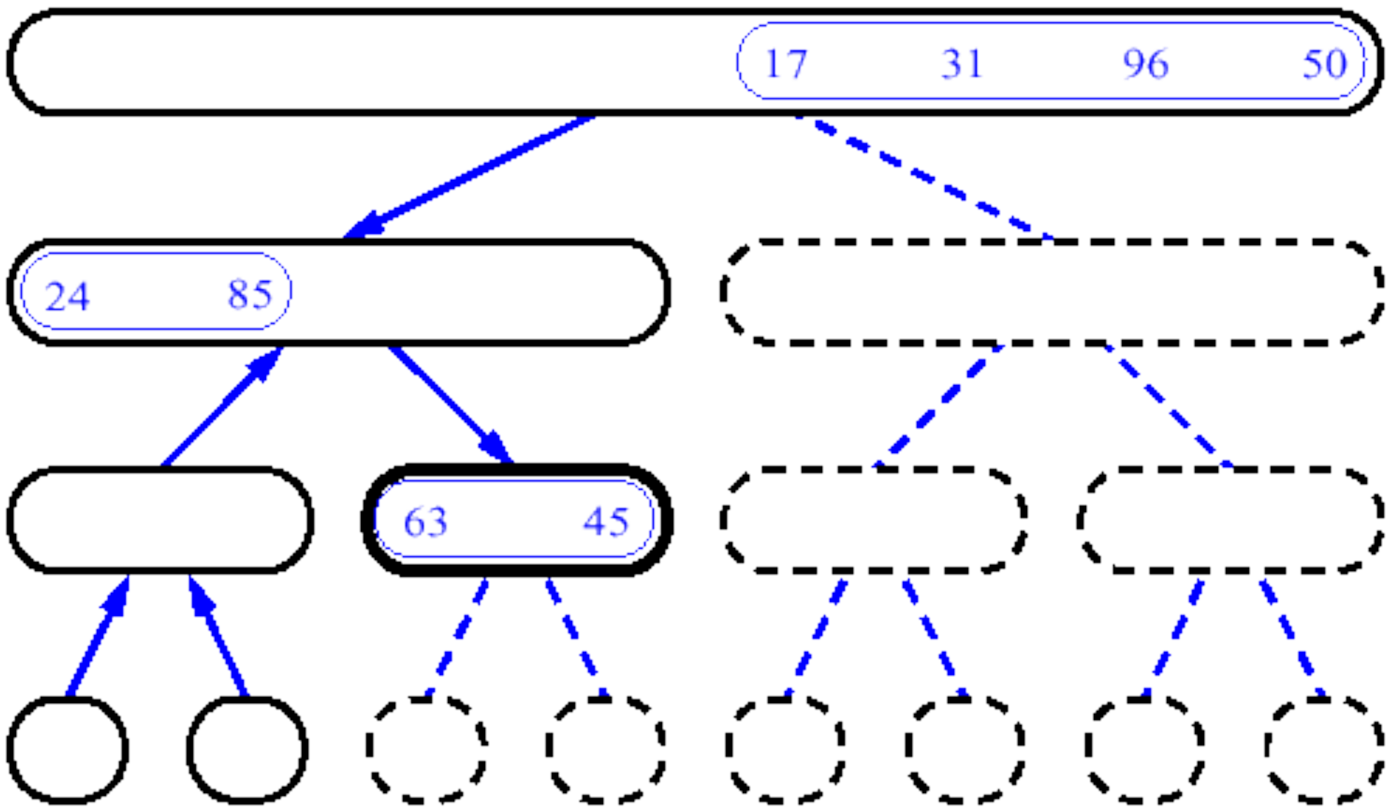
Merge sort (Example)



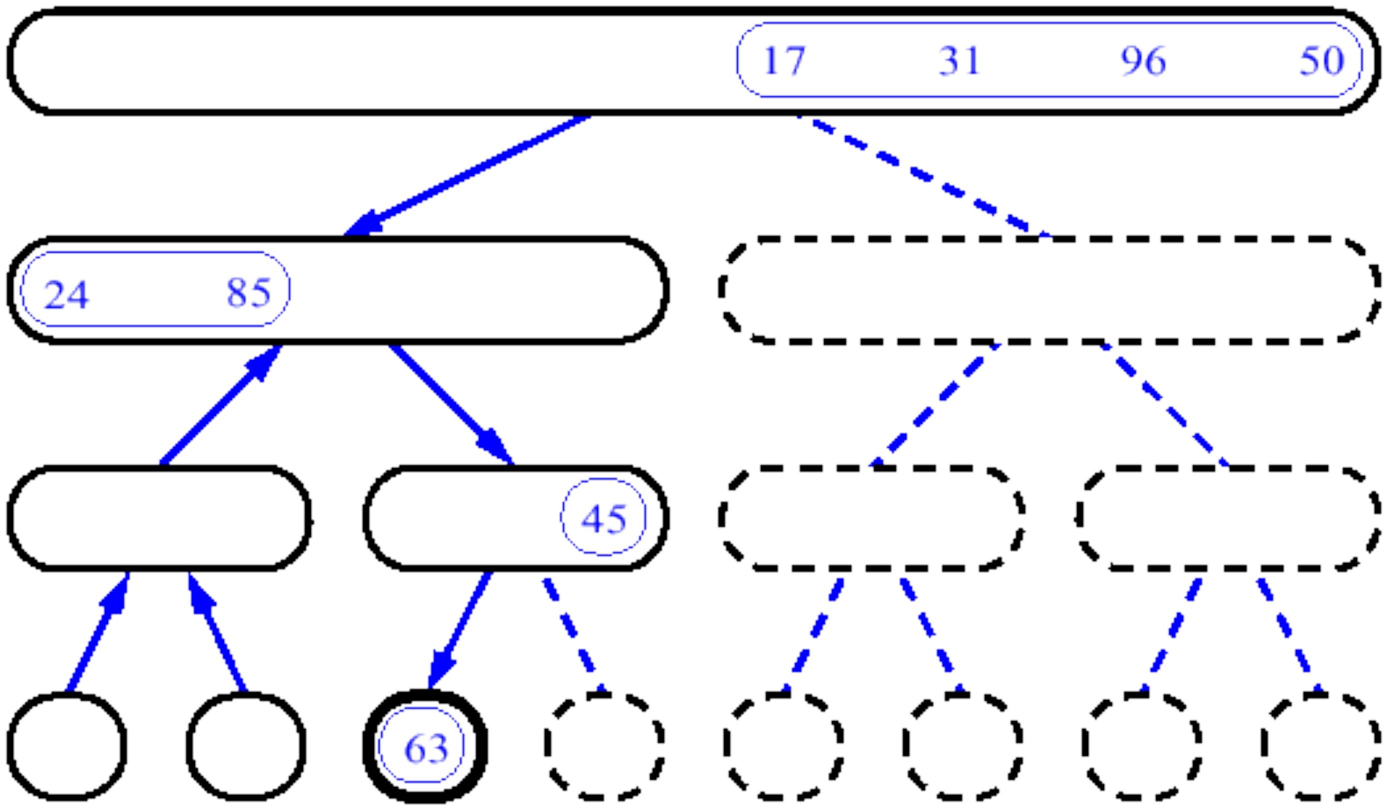
Merge sort (Example)



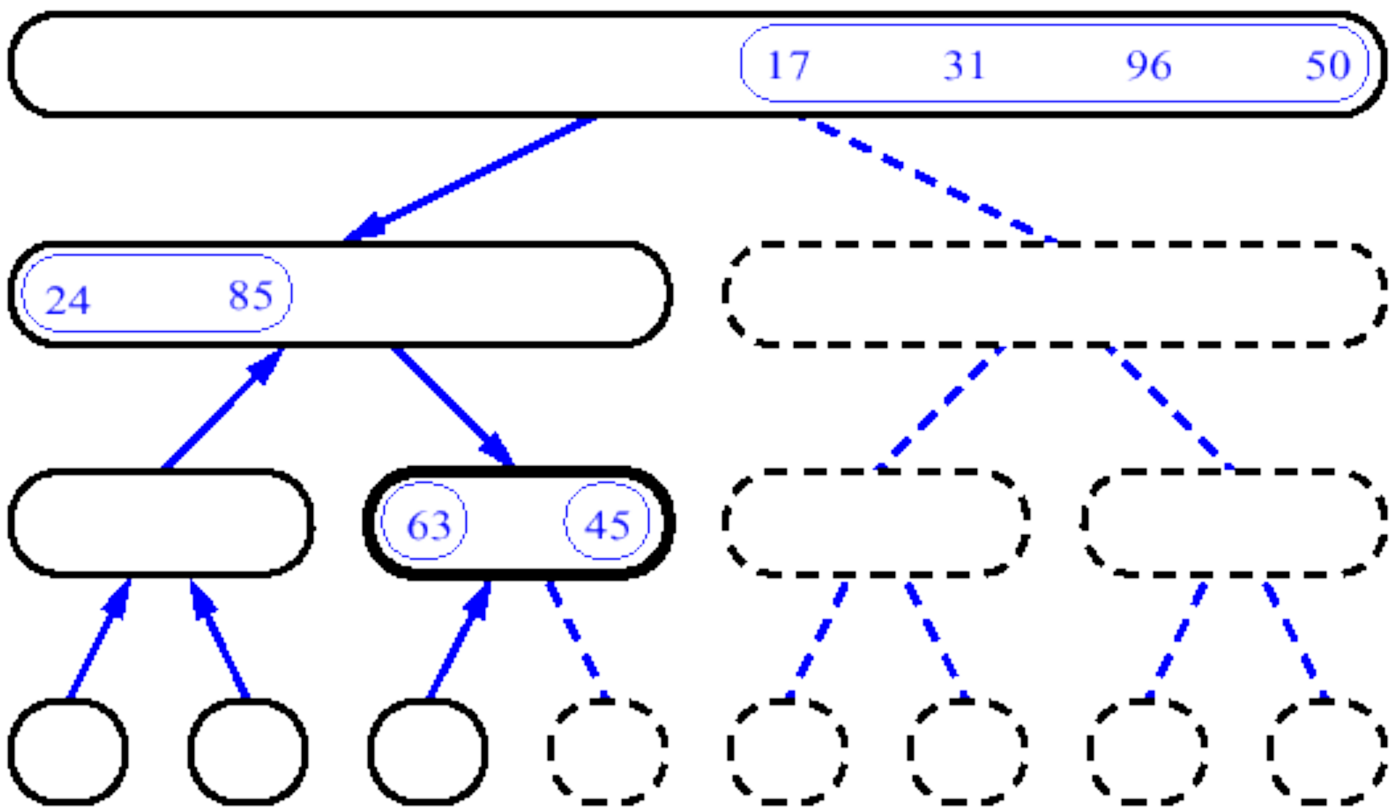
Merge sort (Example)



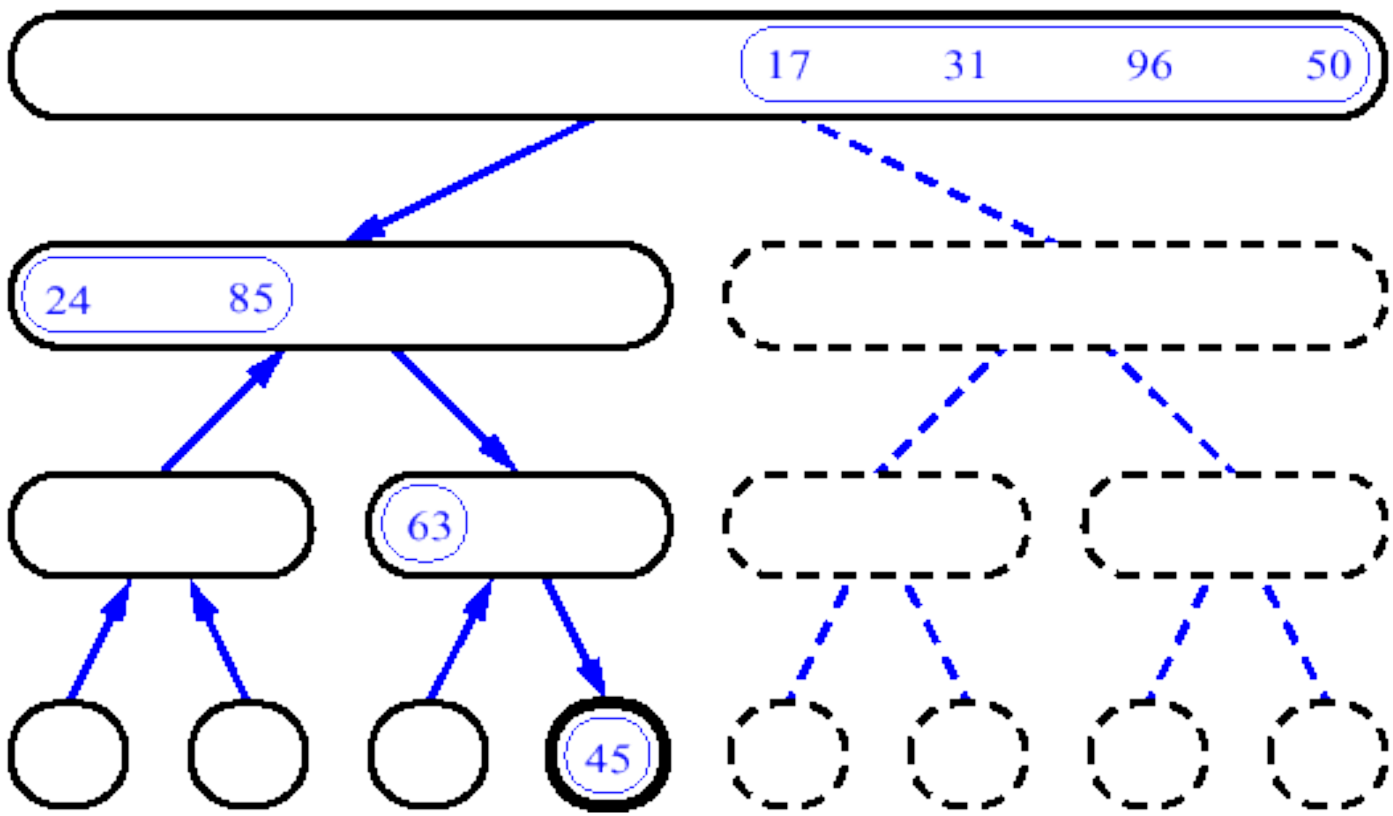
Merge sort (Example)



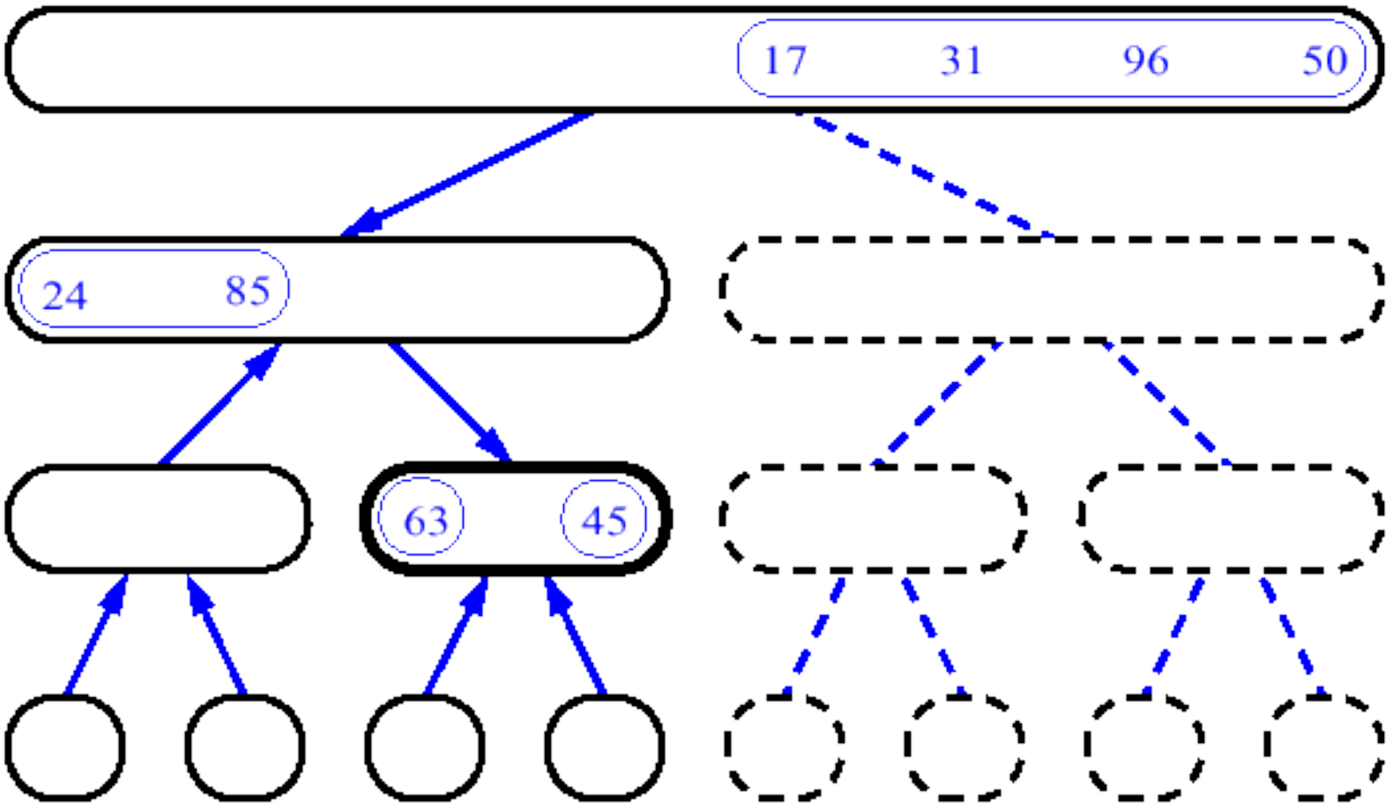
Merge sort (Example)



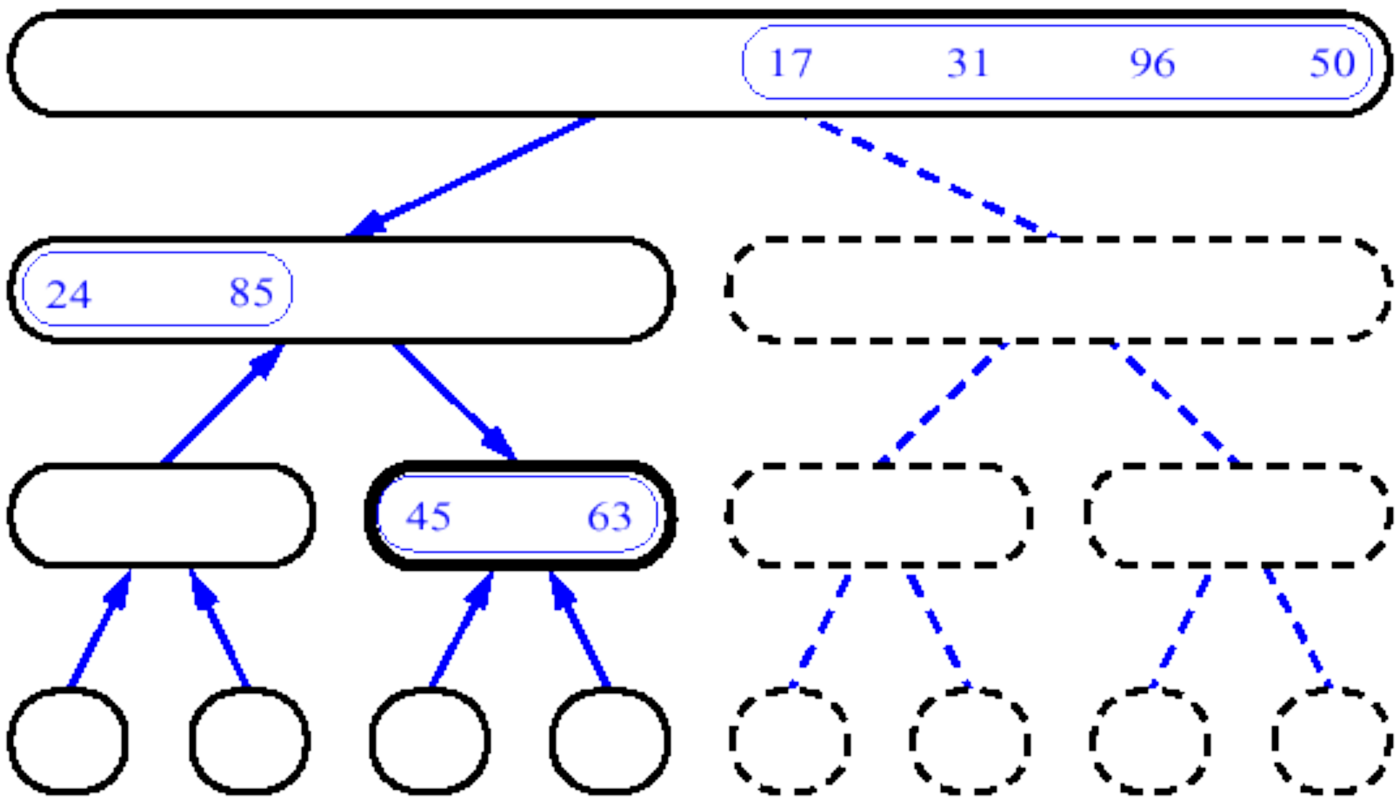
Merge sort (Example)



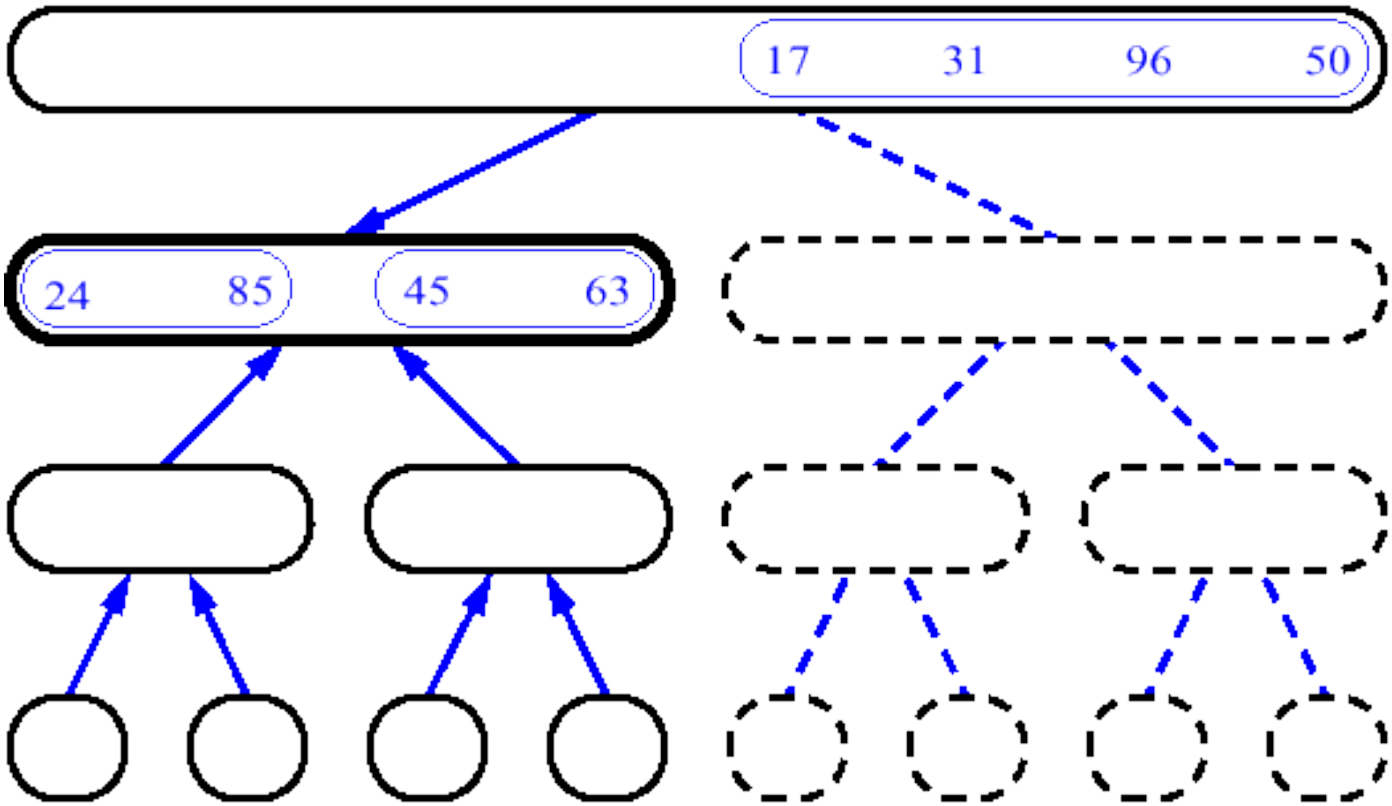
Merge sort (Example)



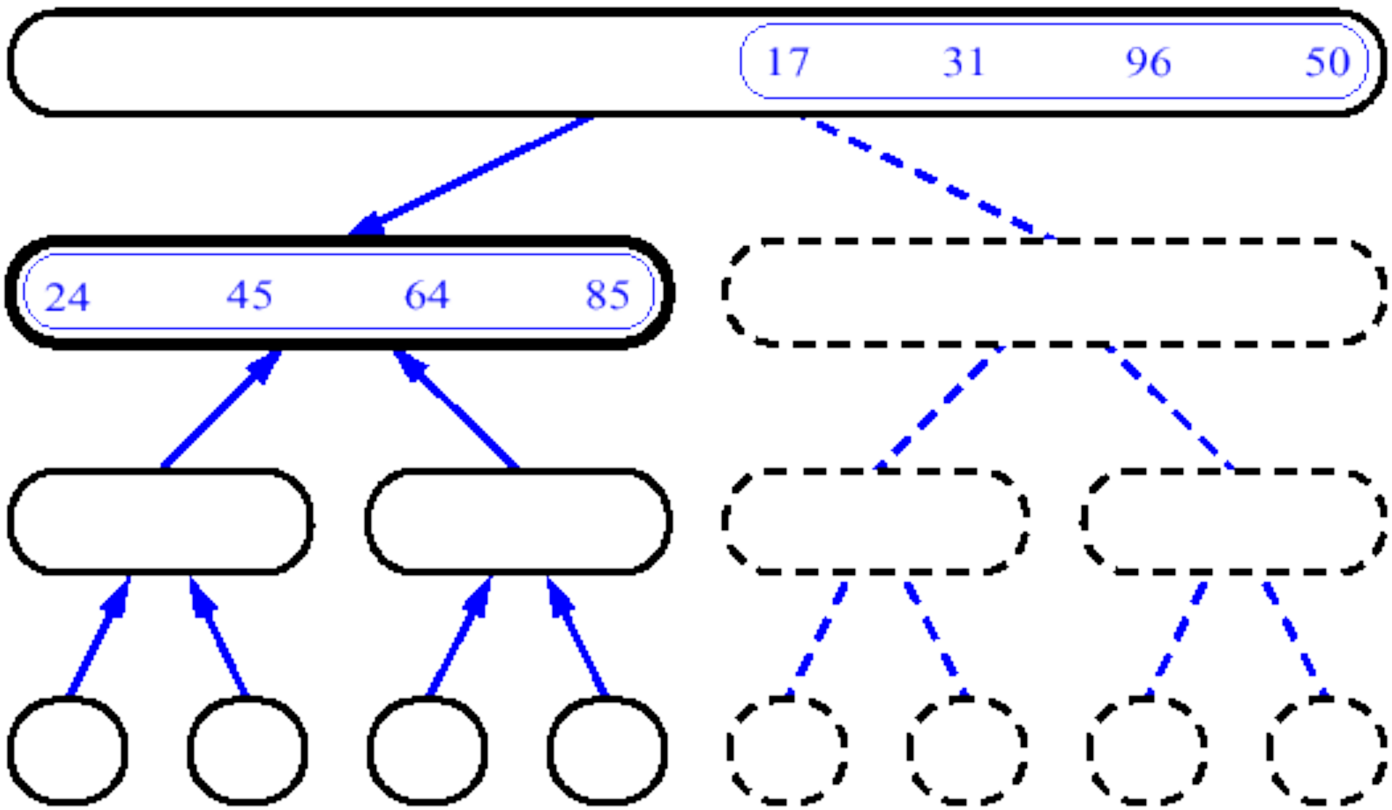
Merge sort (Example)



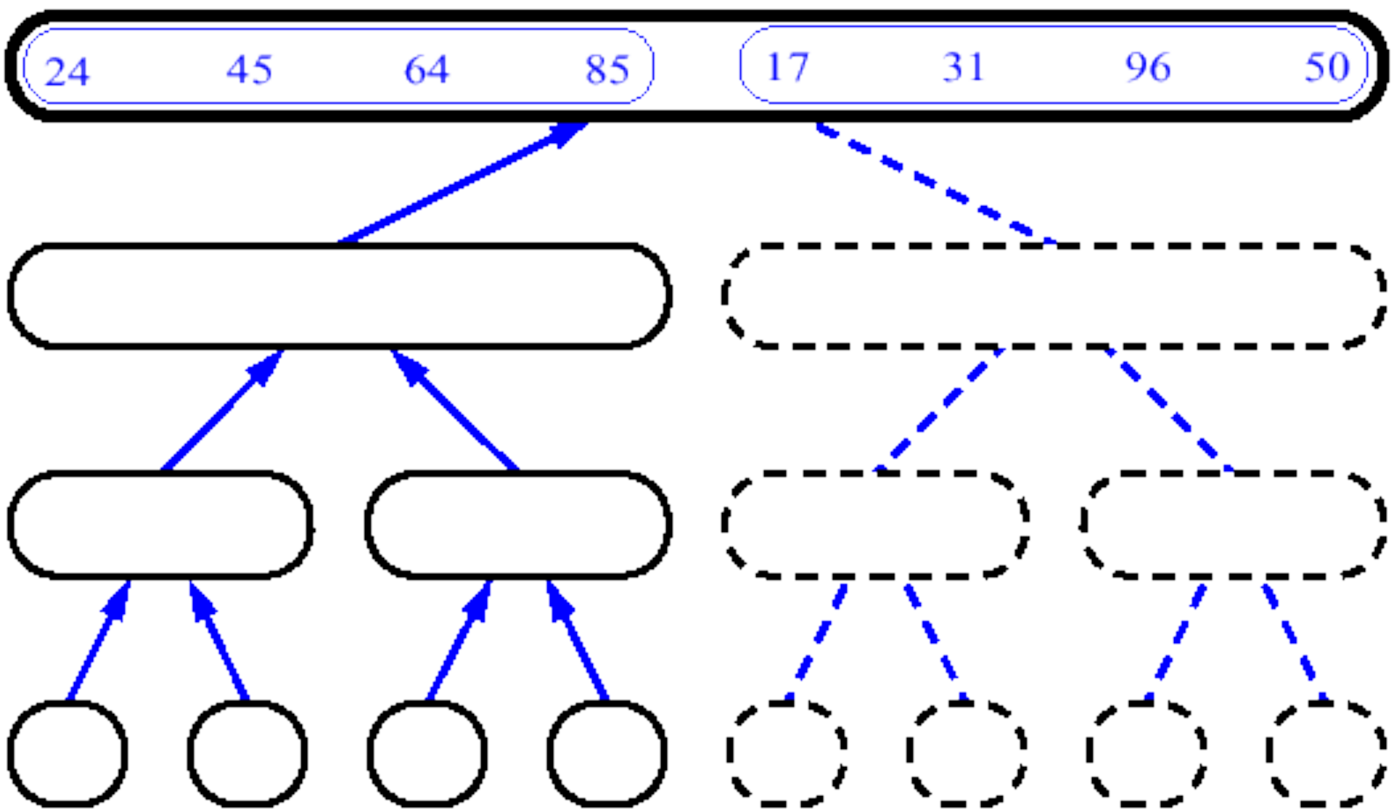
Merge sort (Example)



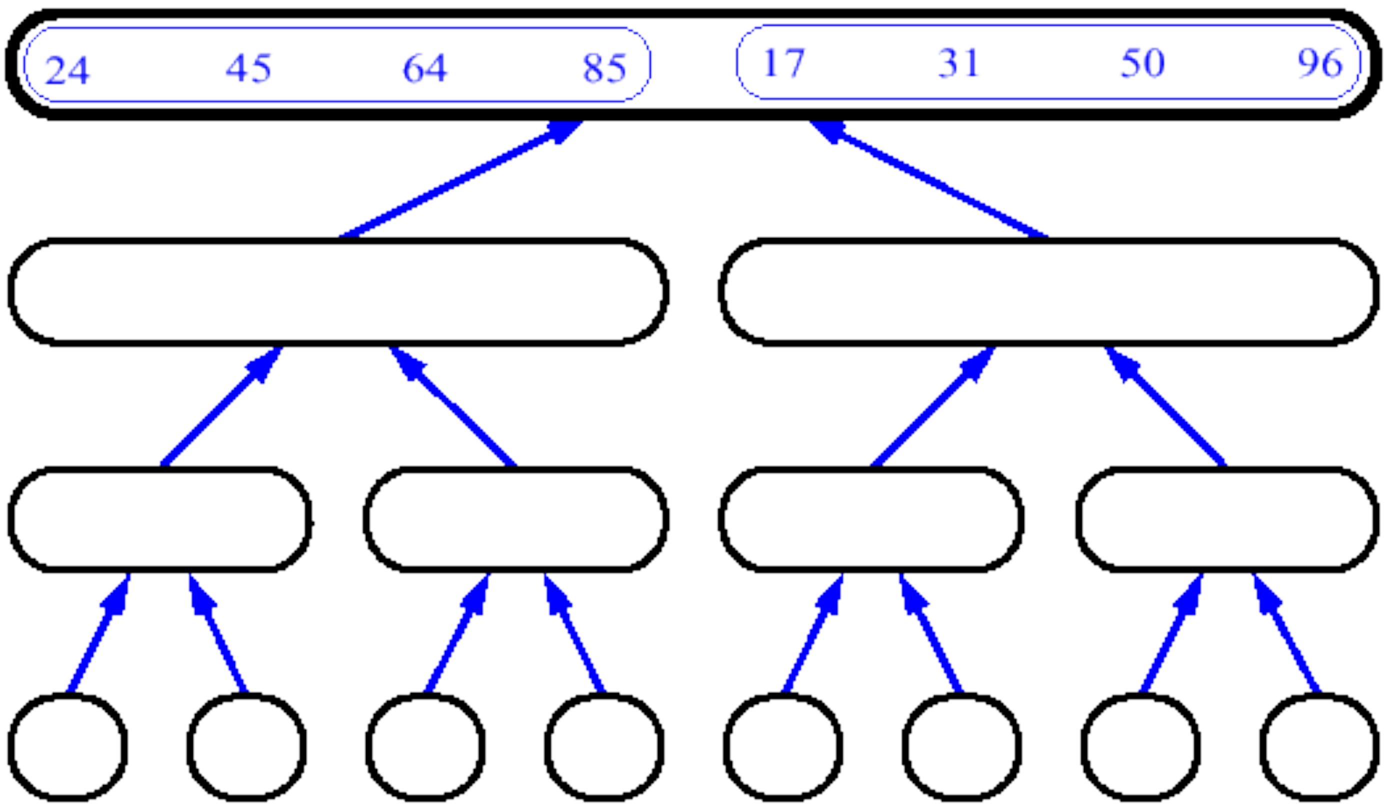
Merge sort (Example)



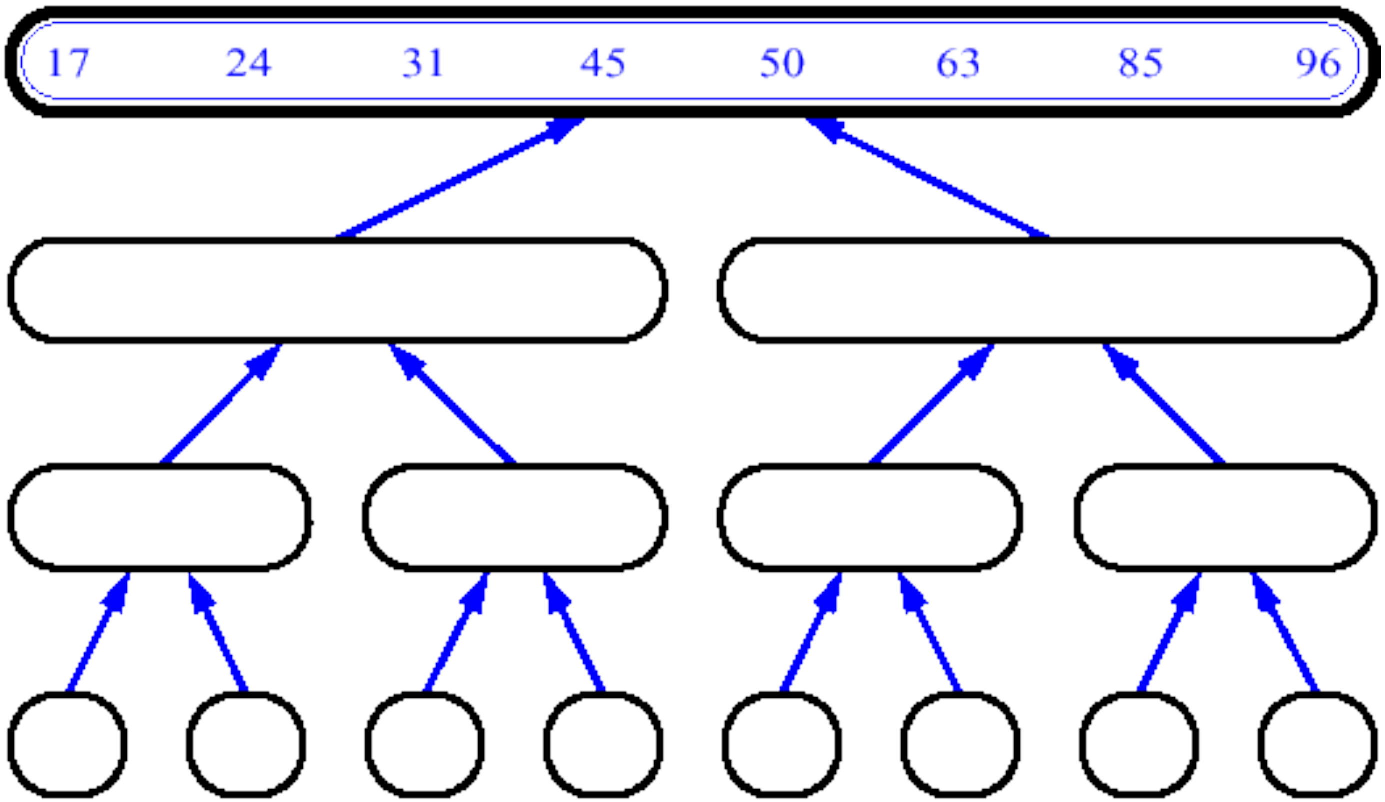
Merge sort (Example)



Merge sort (Example)



Merge sort (Example)



Analysis of merge sort

$T(n)$	**MERGE-SORT** $A[1 \,.\,.\, n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
$\Theta(n)$	***3.*** ***"Merge"*** the 2 sorted lists

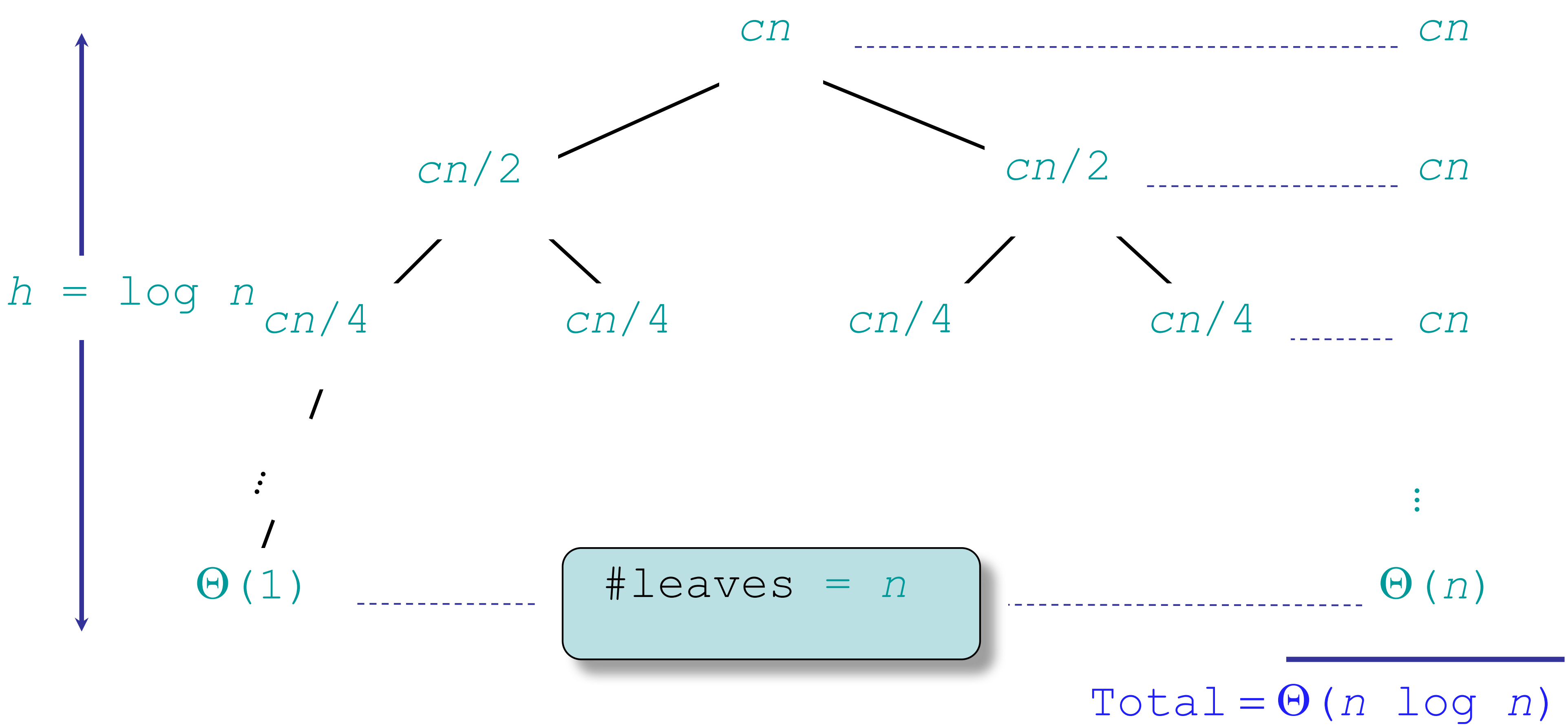
Analyzing merge sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

$$T(n) = \Theta(n \lg n) \qquad (\mathrm{n} > 1)$$

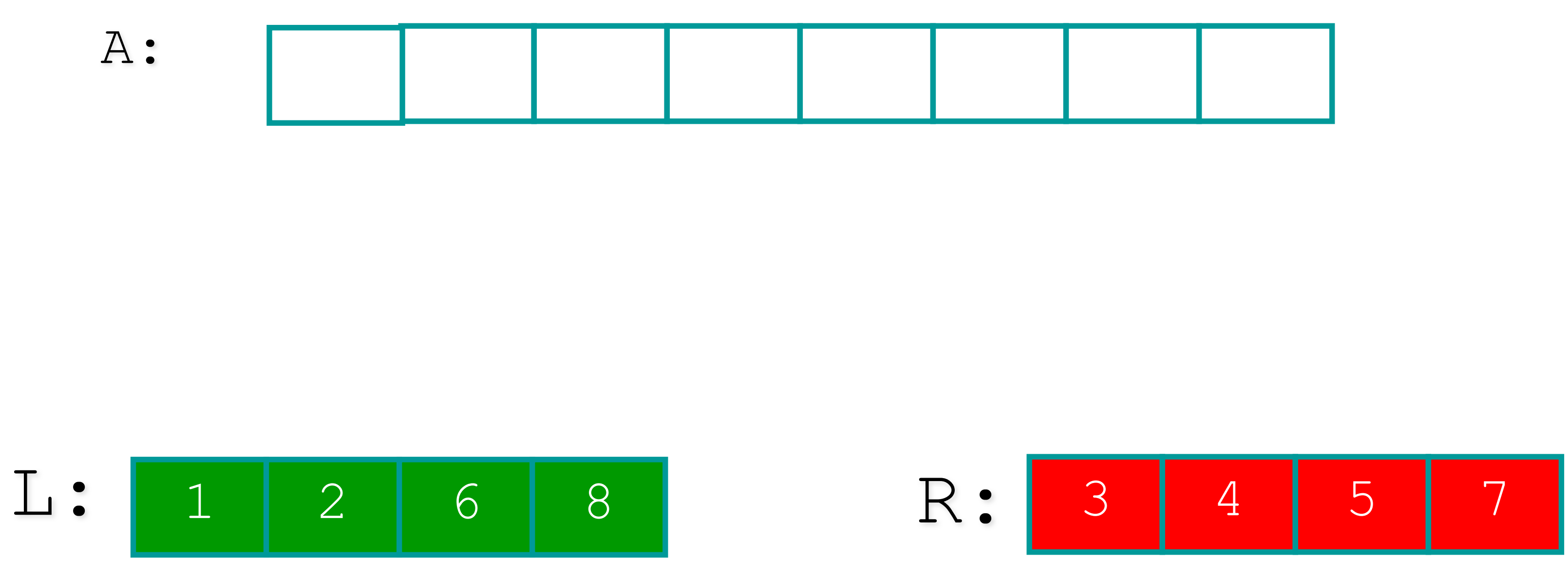
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Memory Requirement

Needs additional *n* locations because it is difficult to merge two sorted sets in place



Merge Sort Conclusion

- Merge Sort: $O(n \log n)$
  - asymptotically beats insertion sort in the worst case
  - In practice, merge sort beats insertion sort for $n > 30$ or so
- Space requirement:
  - $O(n)$, not in-place

HEAPSORT

Heapsort

- Merge sort time is O(n log n) but still requires, temporarily, n extra storage locations
- *Heapsort* does not require any additional storage
- As its name implies, heapsort uses a heap to store the array

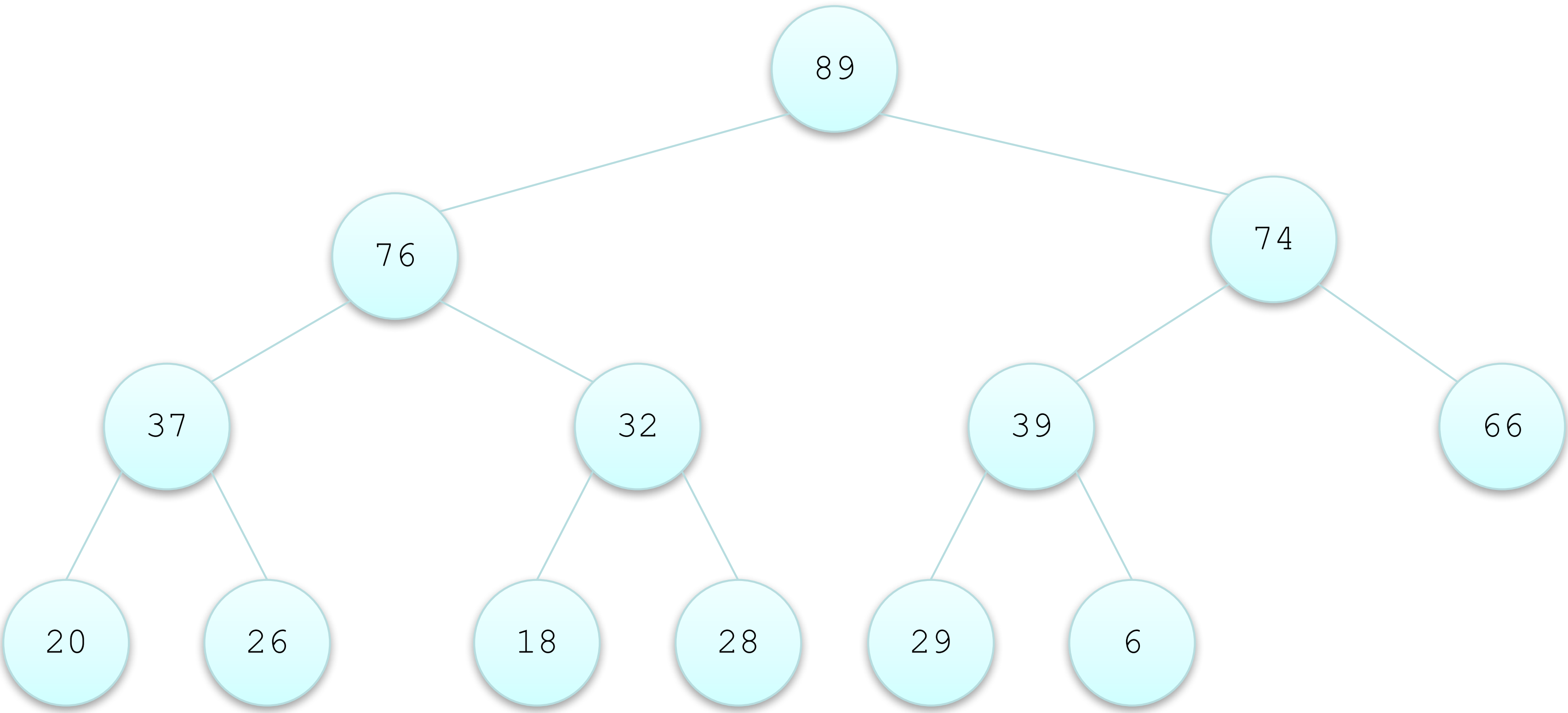
Heapsort Algorithm

- When used as a priority queue, a heap maintains a smallest value at the top
- 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

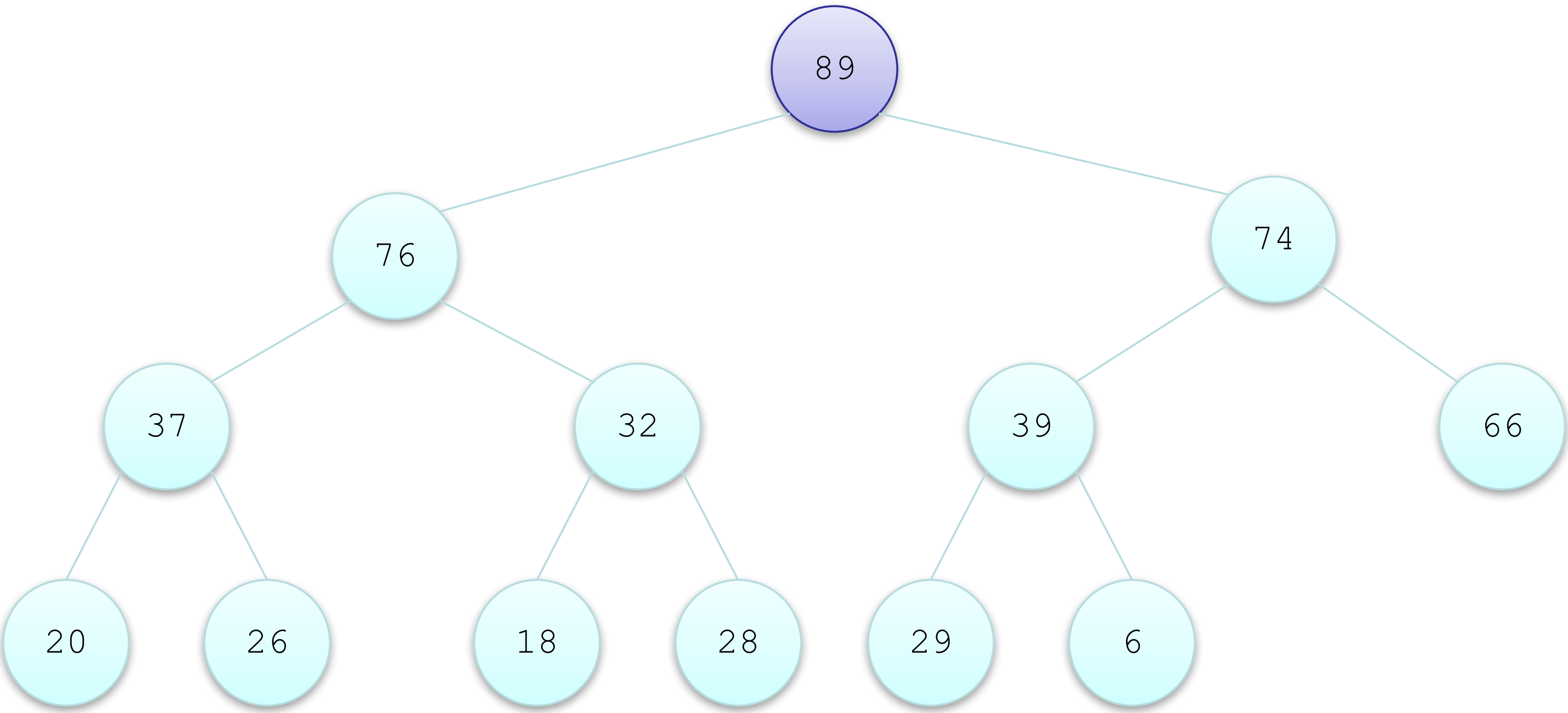
Heapsort Algorithm: First Version

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**

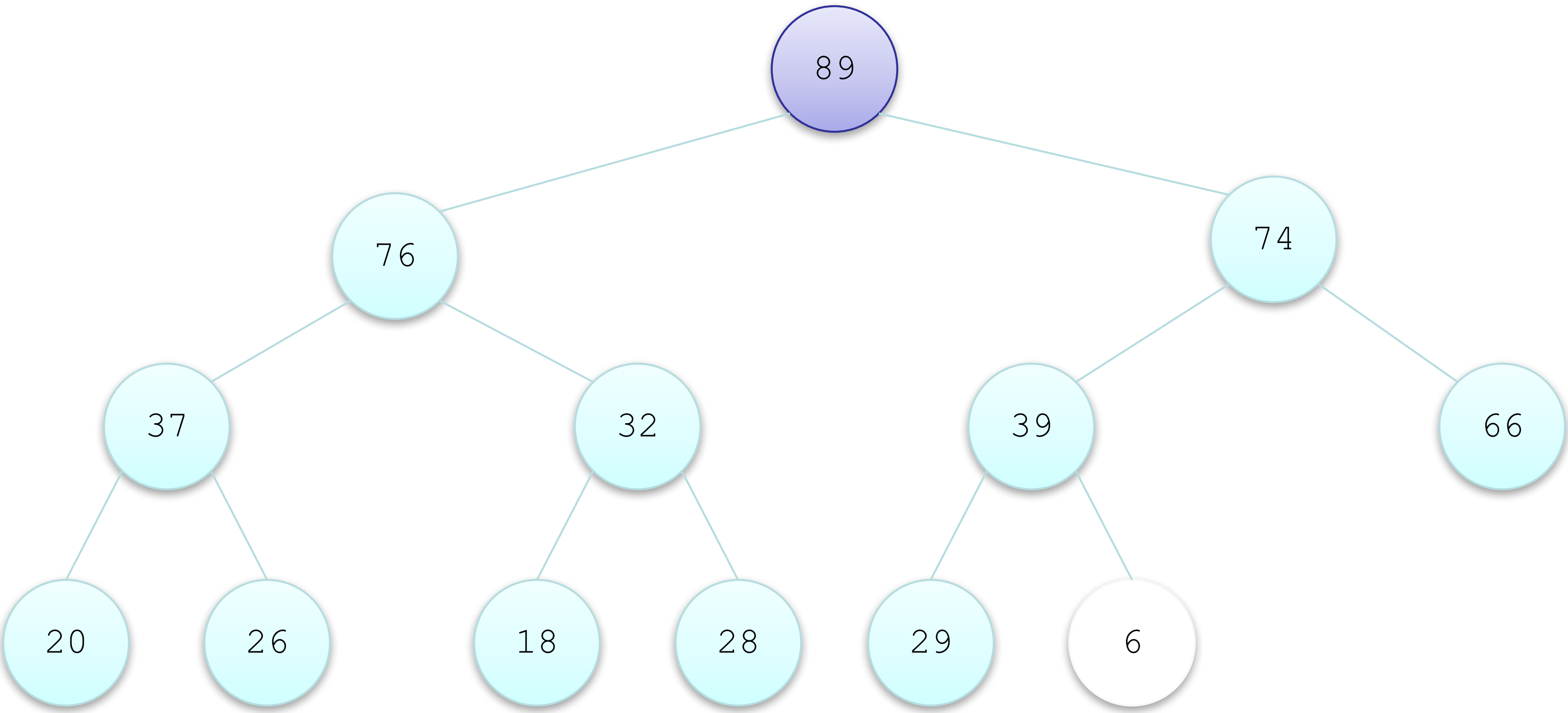
Trace of Heapsort



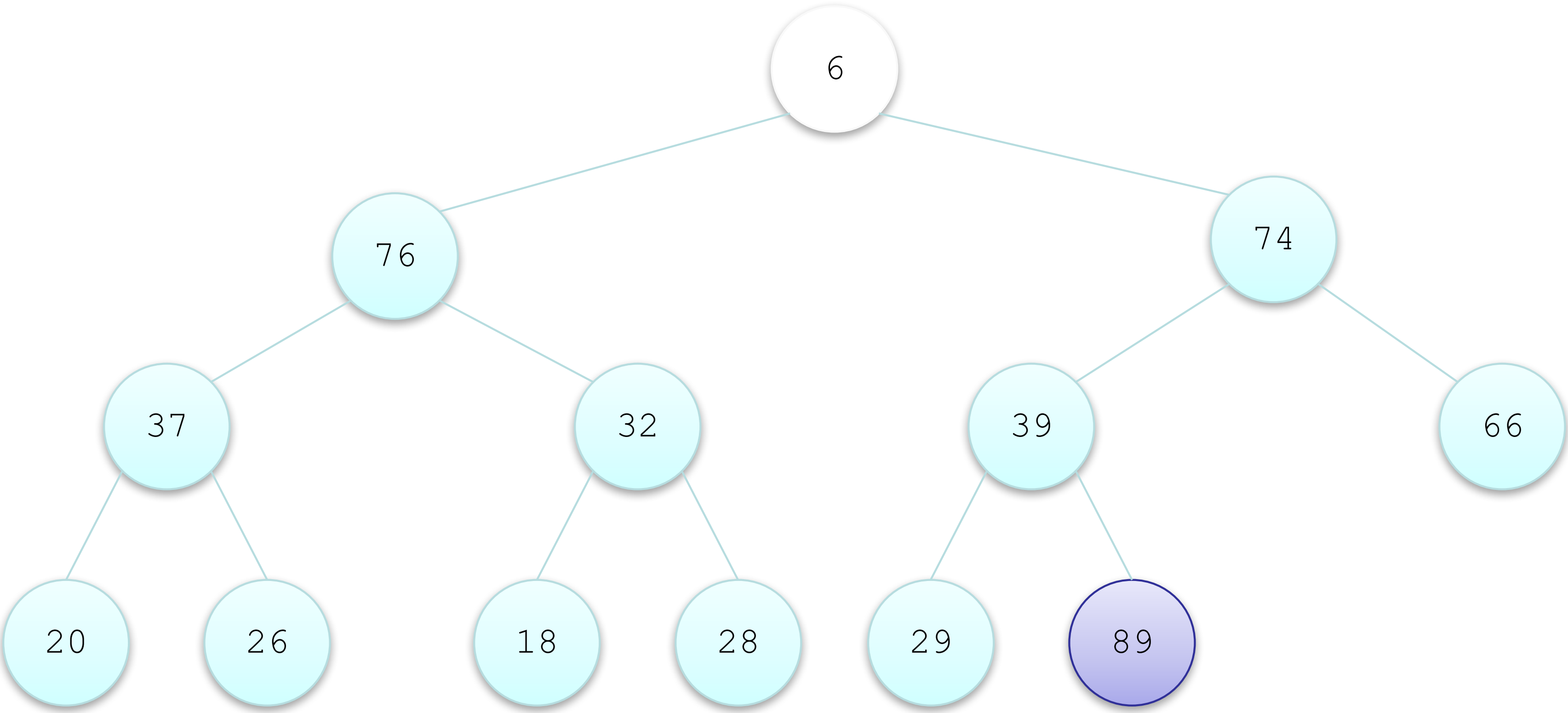
Trace of Heapsort (cont.)



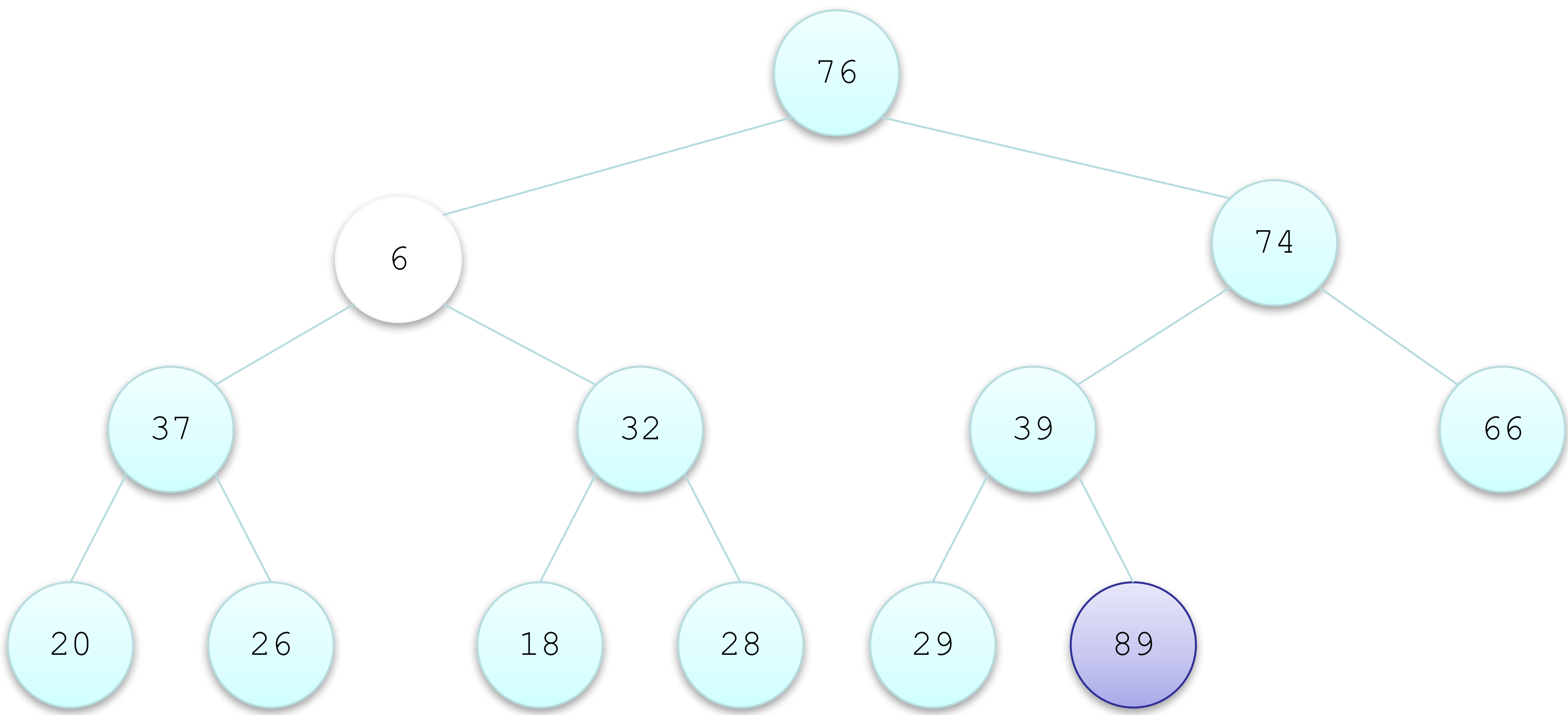
Trace of Heapsort (cont.)



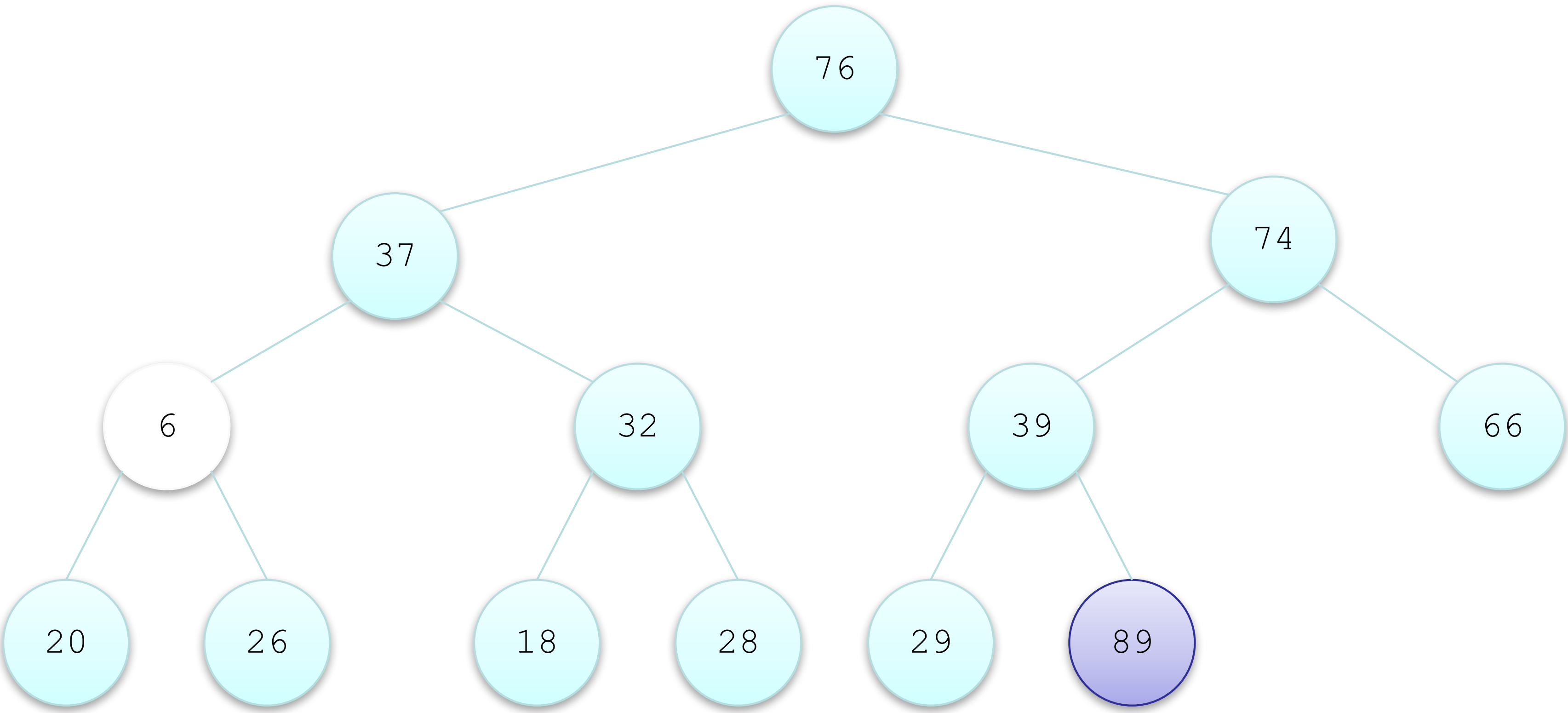
Trace of Heapsort (cont.)



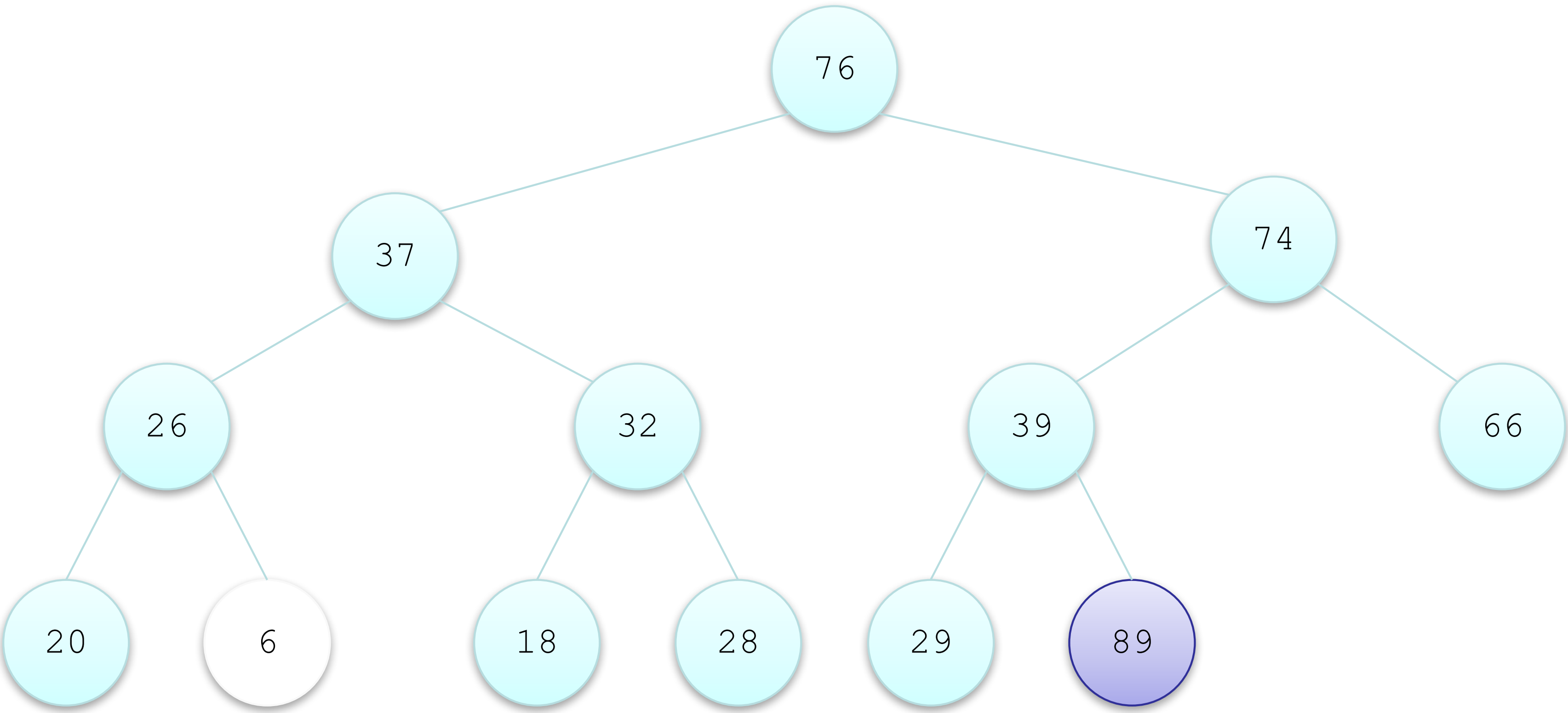
Trace of Heapsort (cont.)



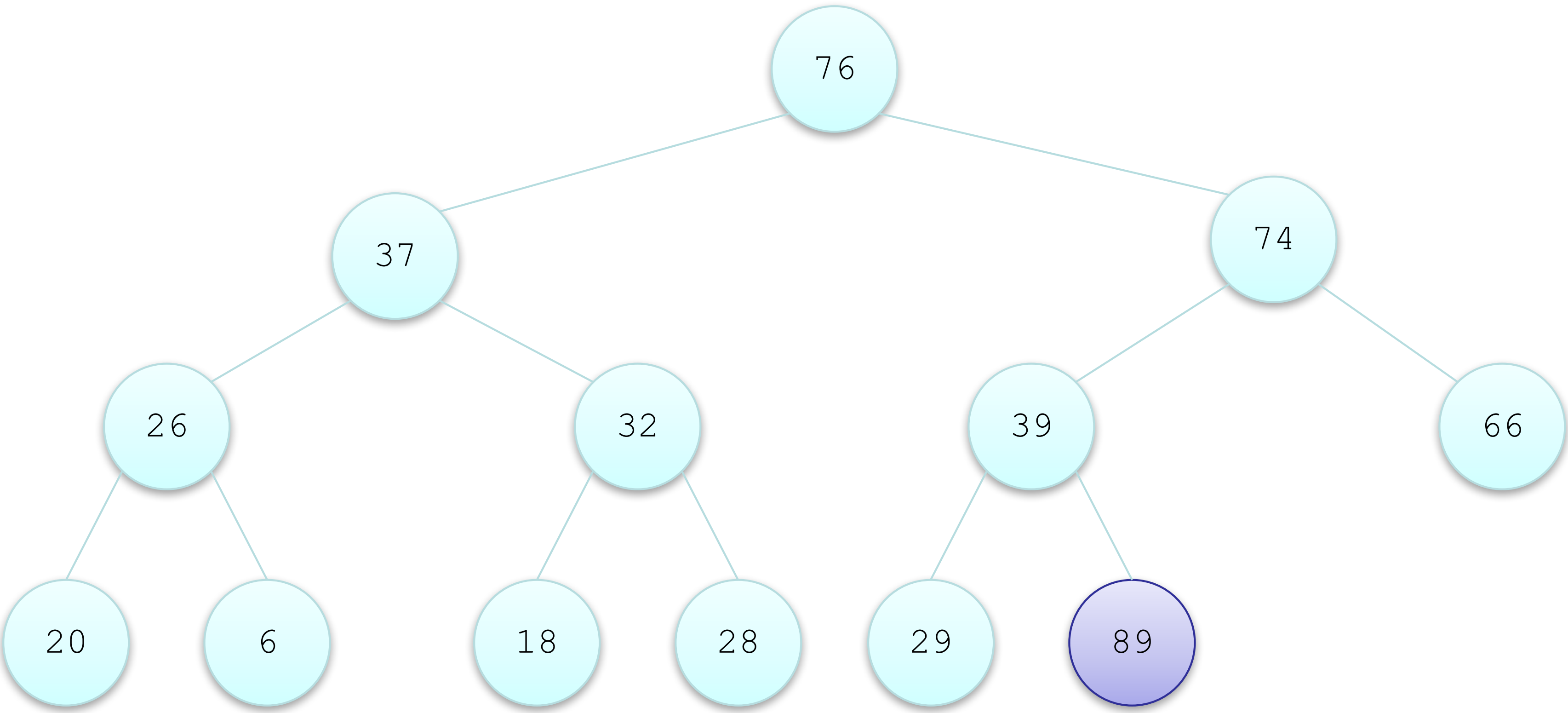
Trace of Heapsort (cont.)



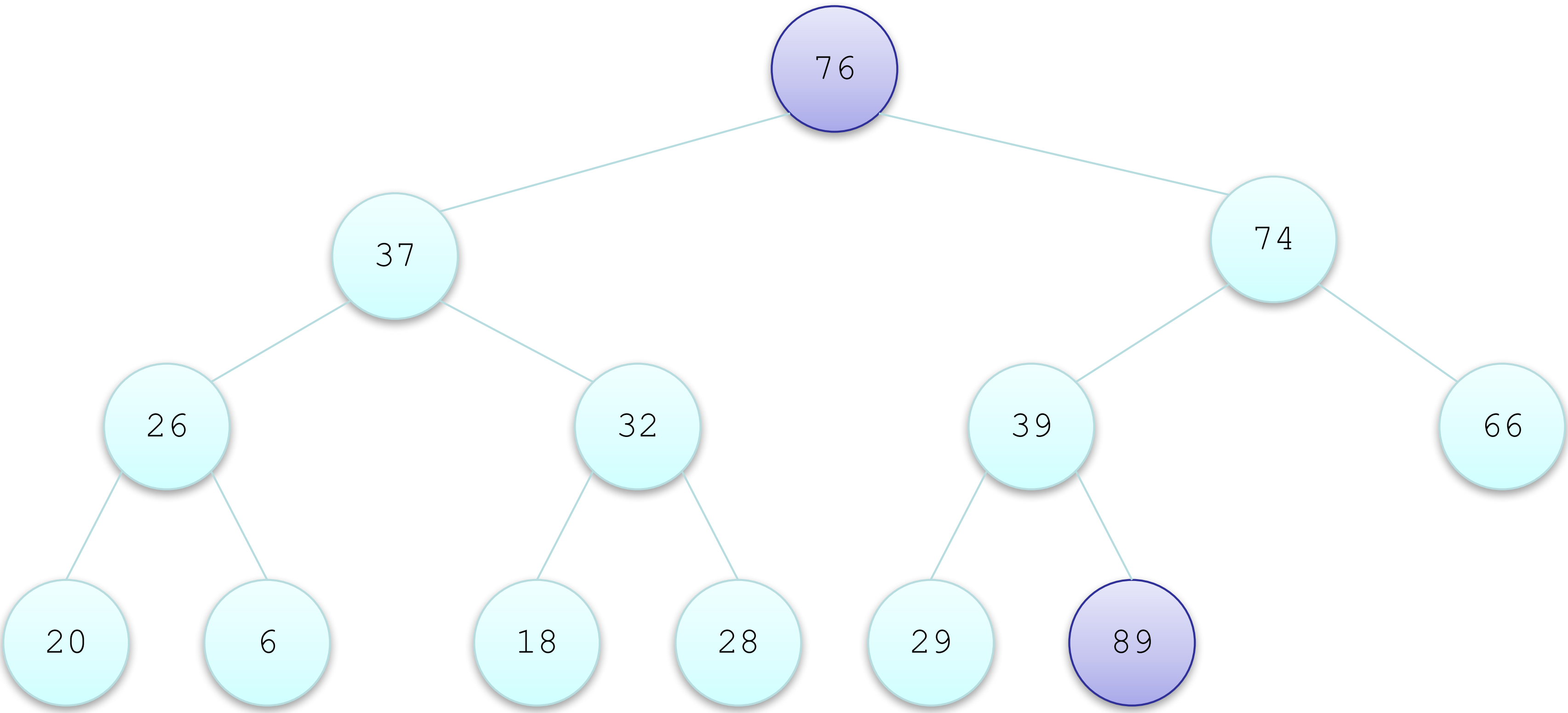
Trace of Heapsort (cont.)



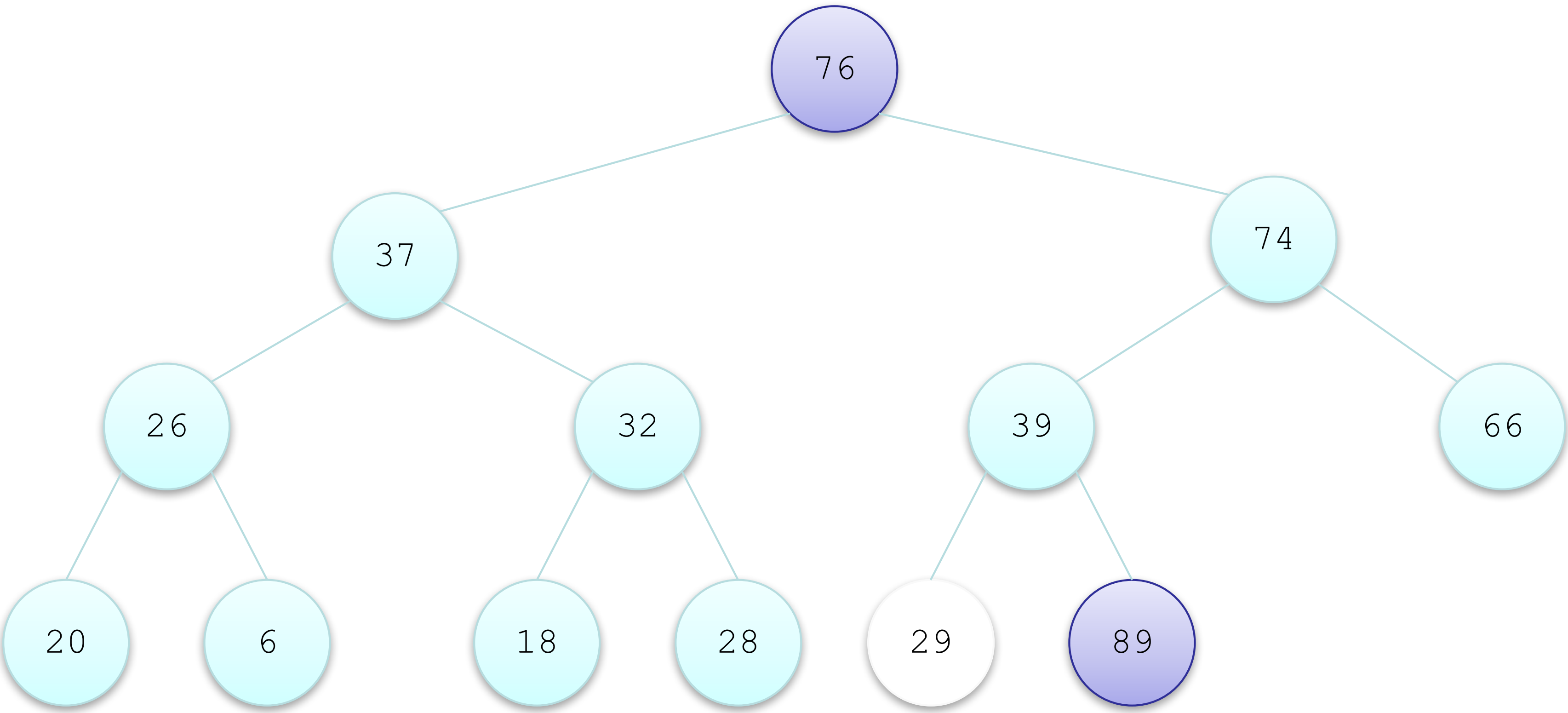
Trace of Heapsort (cont.)



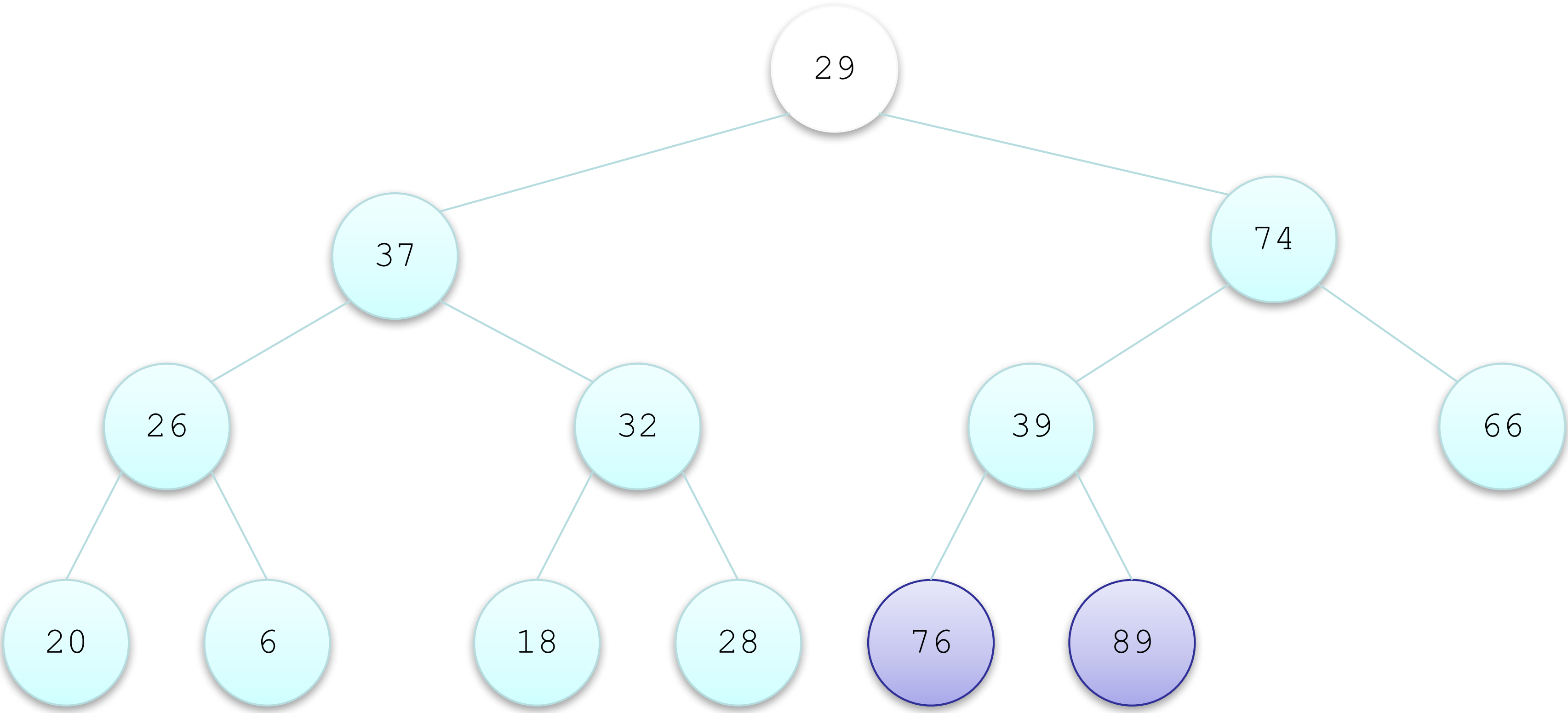
Trace of Heapsort (cont.)



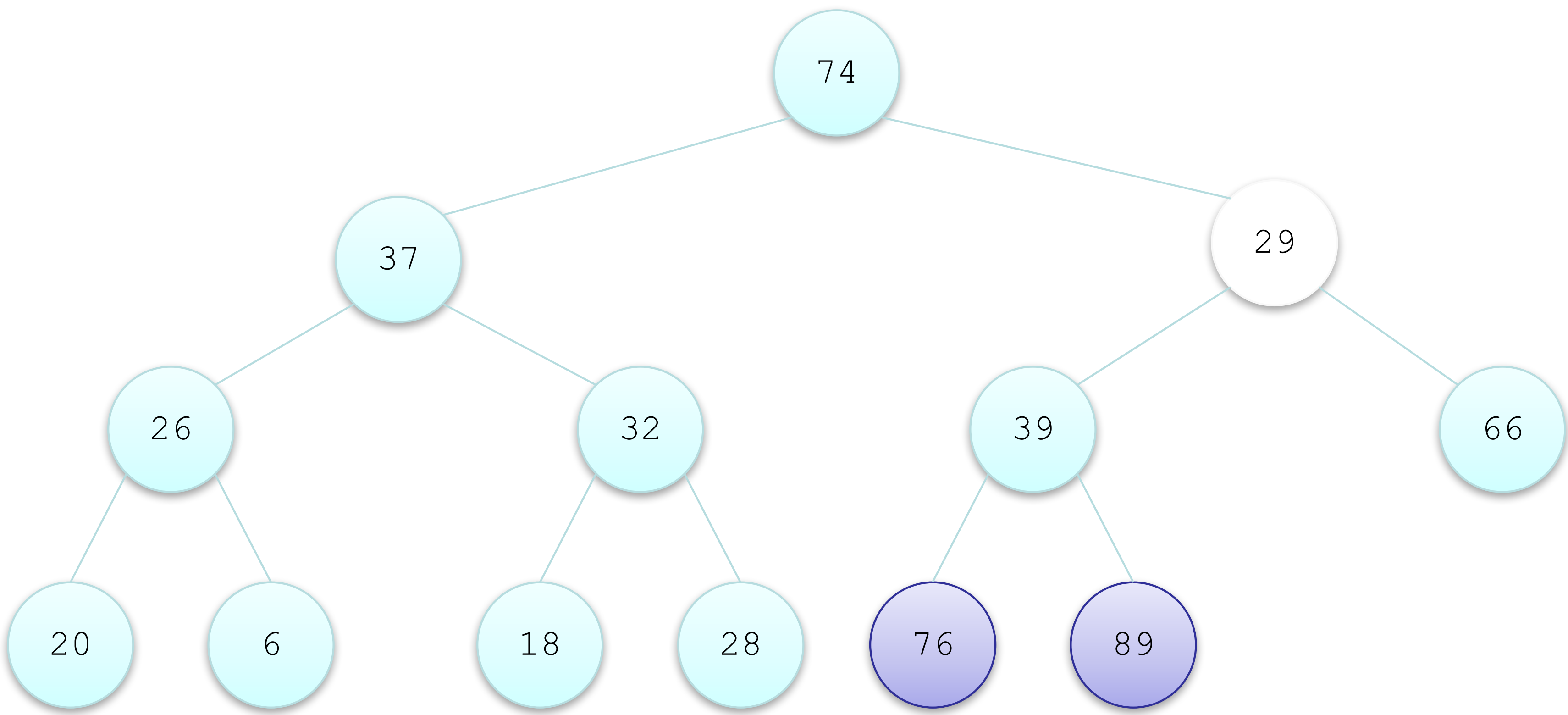
Trace of Heapsort (cont.)



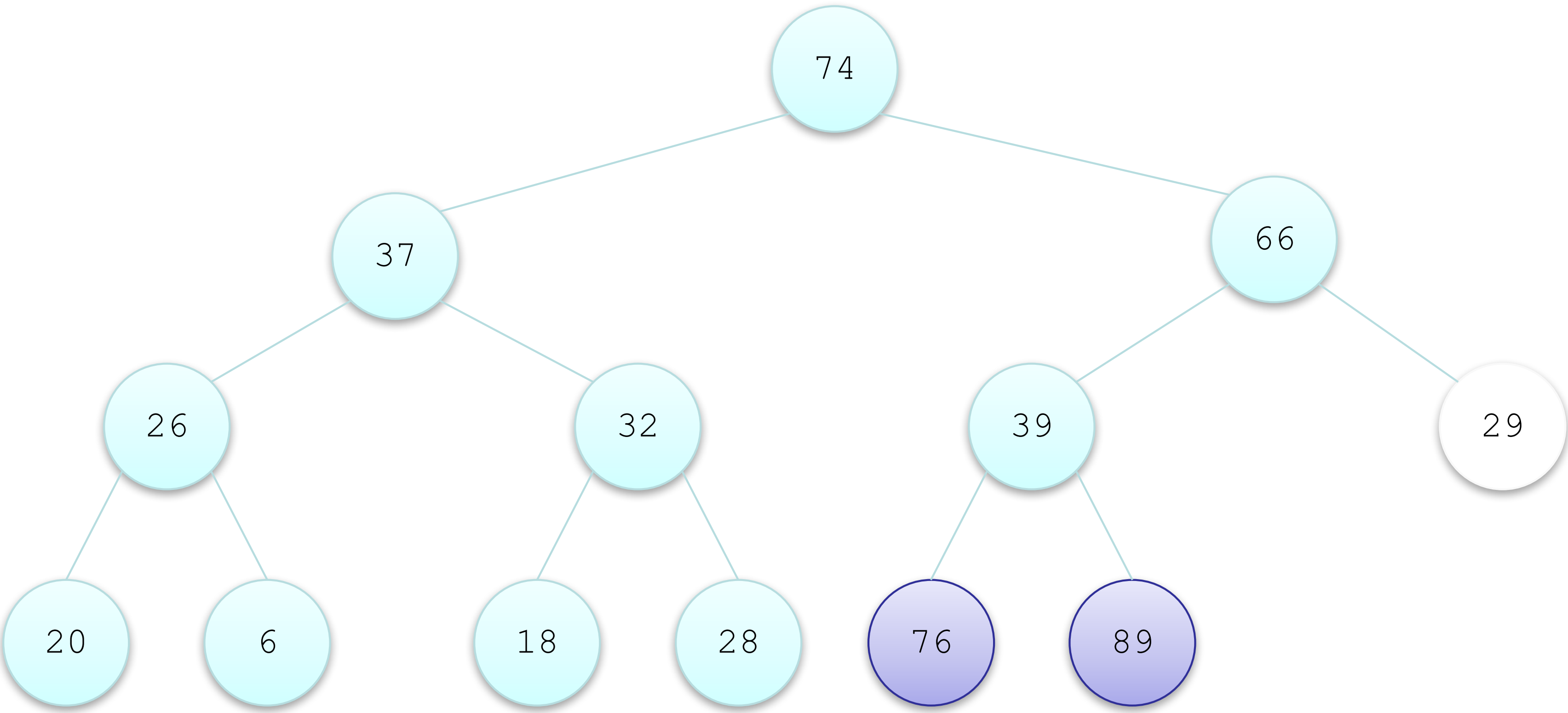
Trace of Heapsort (cont.)



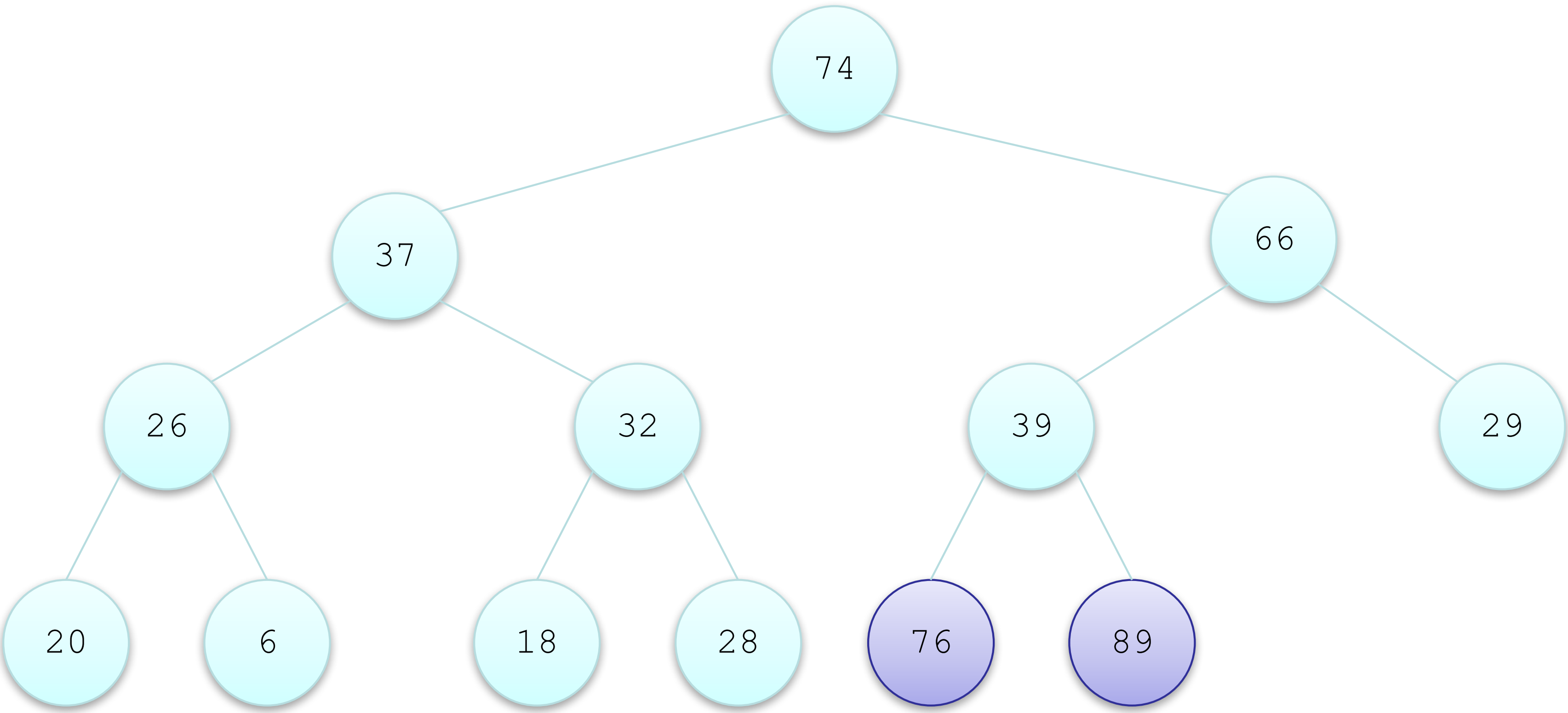
Trace of Heapsort (cont.)



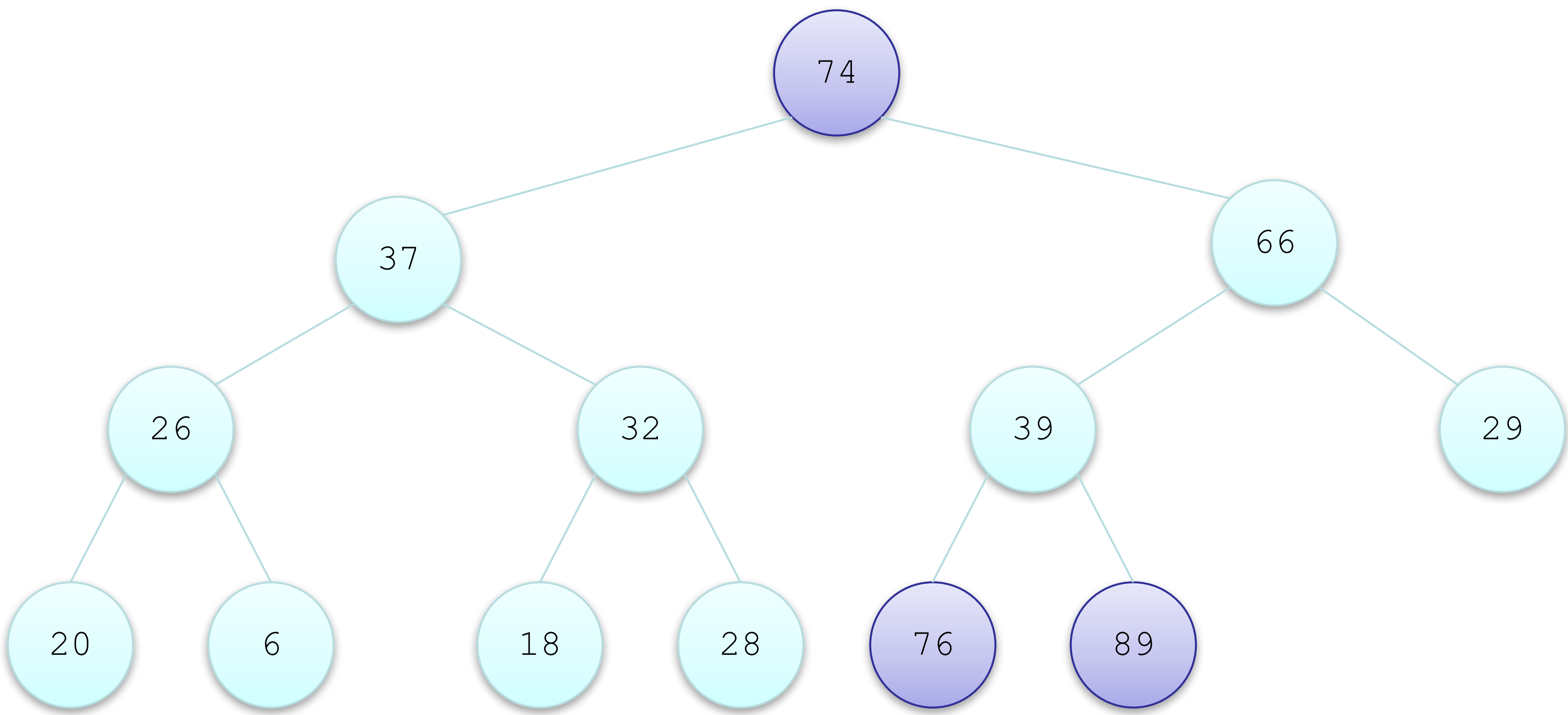
Trace of Heapsort (cont.)



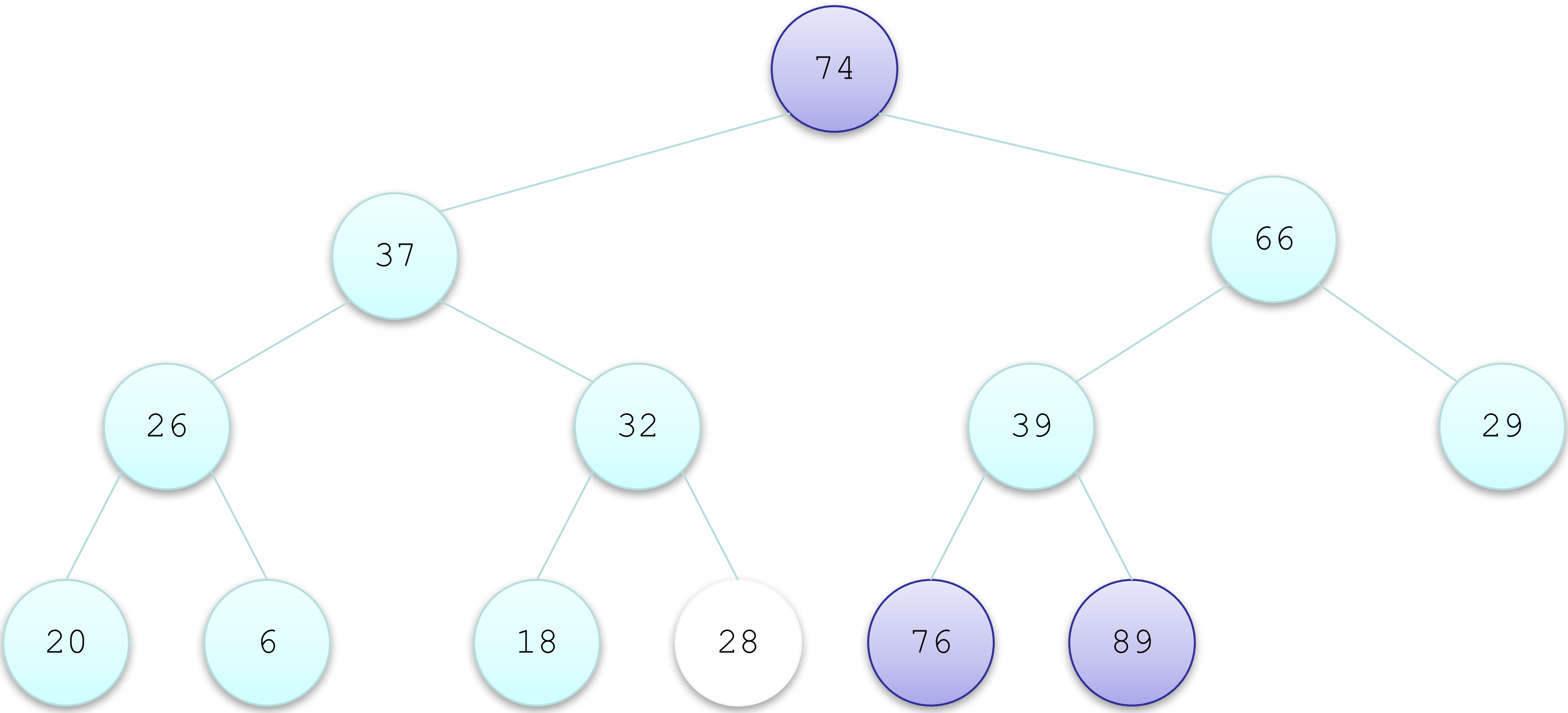
Trace of Heapsort (cont.)



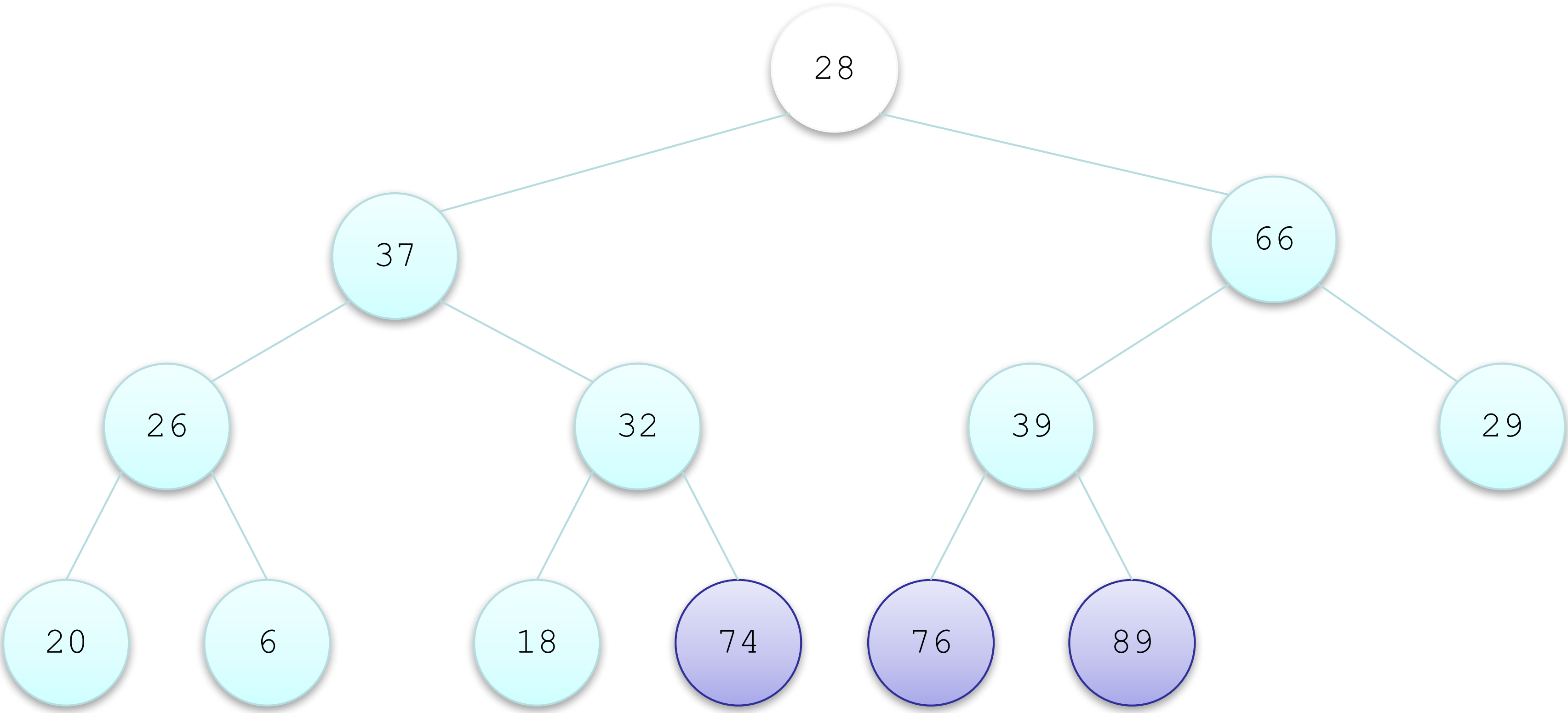
Trace of Heapsort (cont.)



Trace of Heapsort (cont.)

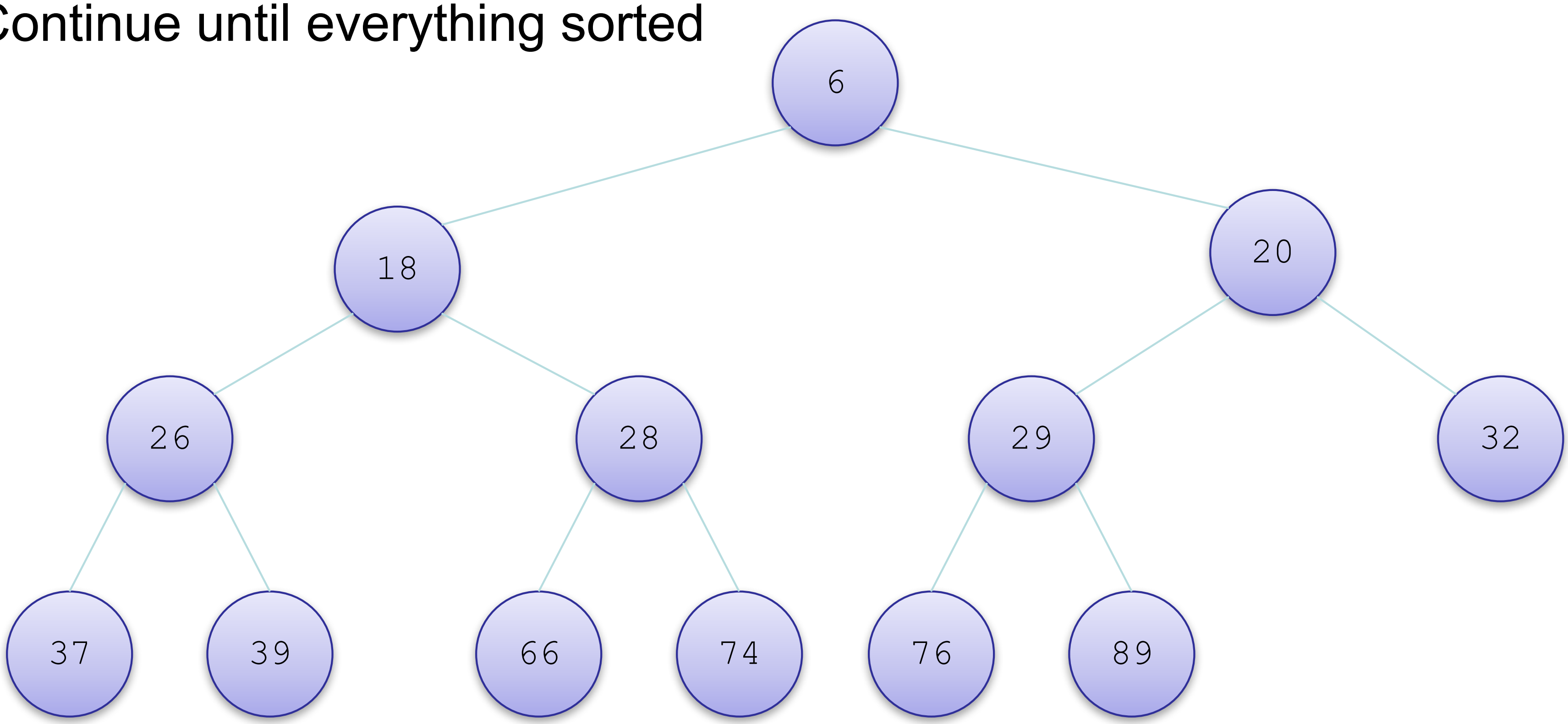


Trace of Heapsort (cont.)



Trace of Heapsort (cont.)

Continue until everything sorted



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

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

Analysis of Heapsort

- Because a heap is a complete binary tree, it has 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

QUICKSORT

Quicksort

- Developed in 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 until the array is sorted

Merge sort vs Quick Sort

Merge sort

13	89	46	22	57	76	98	34	66	83

split

13	89	46	22	57

76	98	34	66	83

sort recursively

13	22	46	57	89

76	98	34	66	83

merge

13	22	34	46	57	66	76	83	89	98

Merge sort vs Quick Sort

13	89	46	22	57	76	98	34	66	83

Split (smart, extra work here)

13	46	22	57	34

89	76	98	66	83

sort recursively

13	22	34	46	57

66	76	83	89	98

Merge is not necessary

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

Quick Sort Algorithm

```
/* quicksort the subarray from a[lo] to a[hi] */

void sort(Comparable[] a, int lo, int hi) {
   if (hi <= lo) return;
   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

Partition

```
// partition the subarray a[lo..hi] so that a[lo..j-1] <= a[j] <=
a[j+1..hi]
// and return the index j.
int partition(Comparable[] a, int lo, int hi) {
   int i = lo;
   int j = hi + 1;
   Comparable v = a[lo];
   while (true) {
       // find item on lo to swap
       while (less(a[++i], v))
           if (i == hi) break;
      /* find item on hi to swap */
      while (less(v, a[--j]))
        if (j == lo) break;
      // check if pointers cross
      if (i >= j) break;
      exch(a, i, j);
    }
    // put partitioning item v at a[j]
    exch(a, lo, j);
  // now, a[lo .. j-1] <= a[j] <= a[j+1 .. hi]
  return j;
}
```

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
- 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 merge sort

Analysis of Quicksort (cont.)

- The array split may not be the best case, i.e. 50-50
- An exact analysis is difficult (and beyond the scope of this class), but, the running time will be bounded by a constant x $n \log n$

Analysis of Quicksort (cont.)

- A quicksort will give very poor behavior if, each time the array is partitioned, a subarray is empty.
- In that case, the sort will be O(n^2)

- 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.

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.

For simplicity we pick the element with subscript `first`

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

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

For simplicity we pick the element with subscript `first`

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored light purple

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

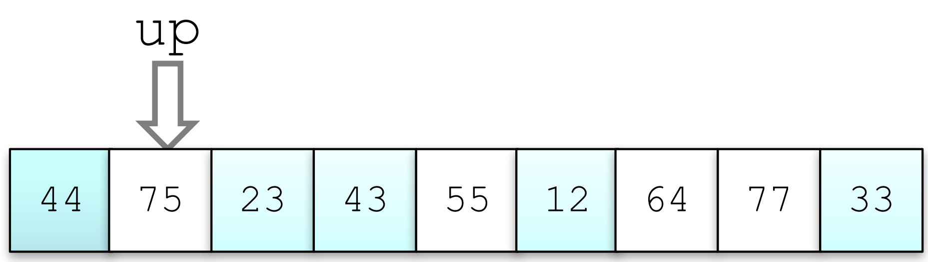
For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored light purple

Trace of Partitioning (cont.)

44	75	23	43	55	12	64	77	33

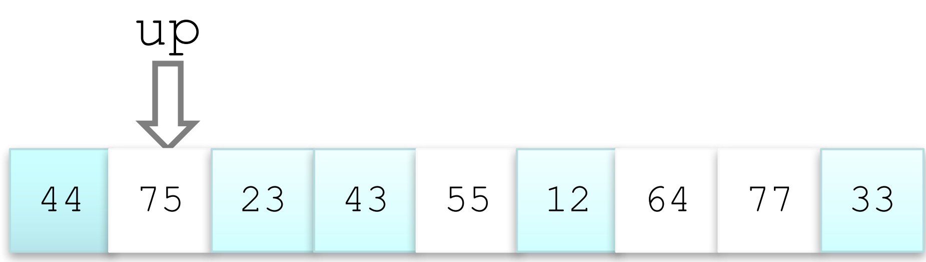
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



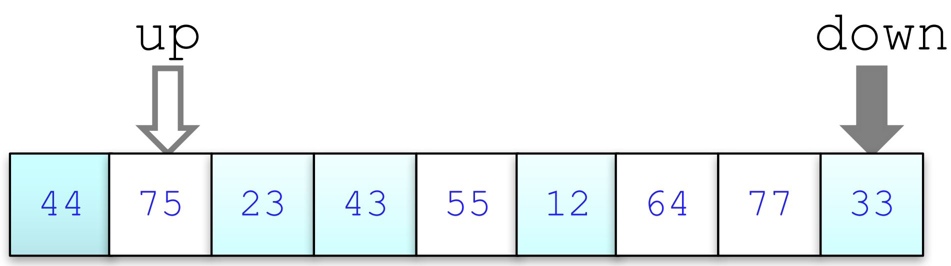
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



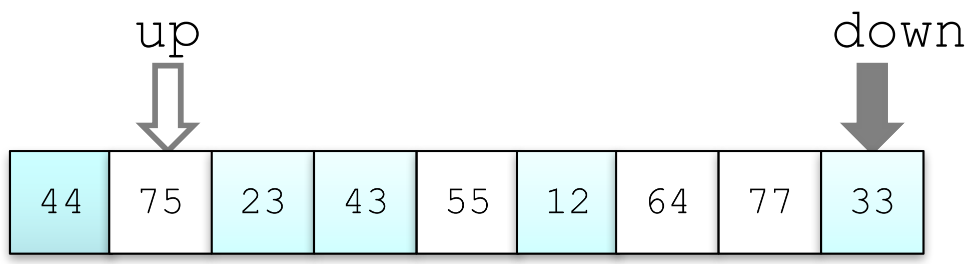
Then search for the first value at the right end of the array that is less than or equal to the pivot value

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

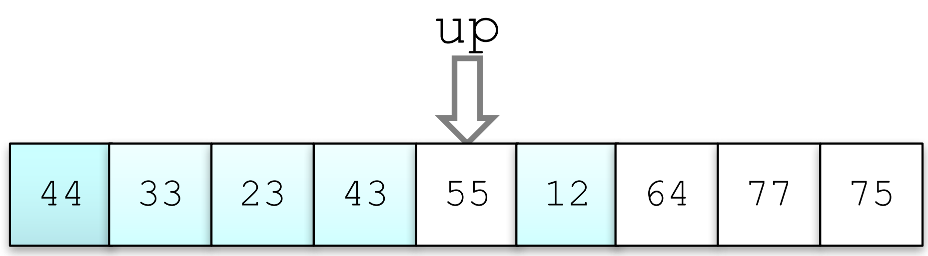
Exchange these values

Trace of Partitioning (cont.)

44	33	23	43	55	12	64	77	75

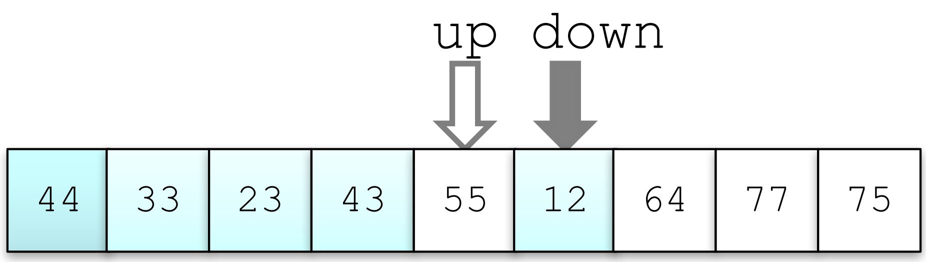
Repeat

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

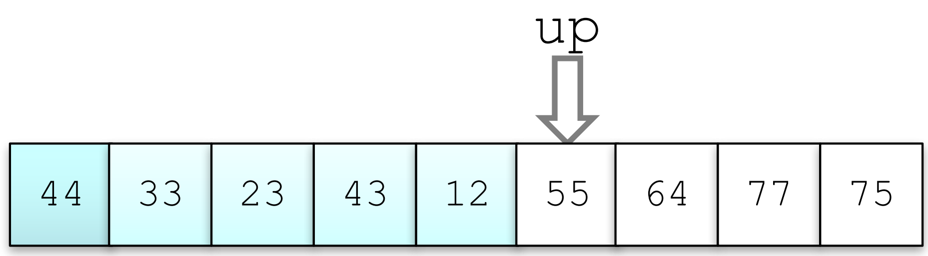
Exchange

Trace of Partitioning (cont.)

44	33	23	43	12	55	64	77	75

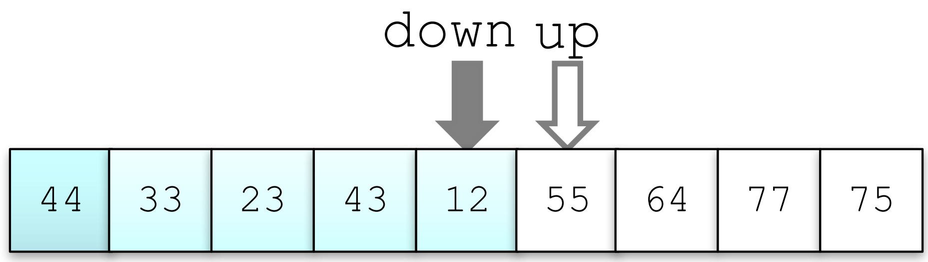
Repeat

Trace of Partitioning (cont.)



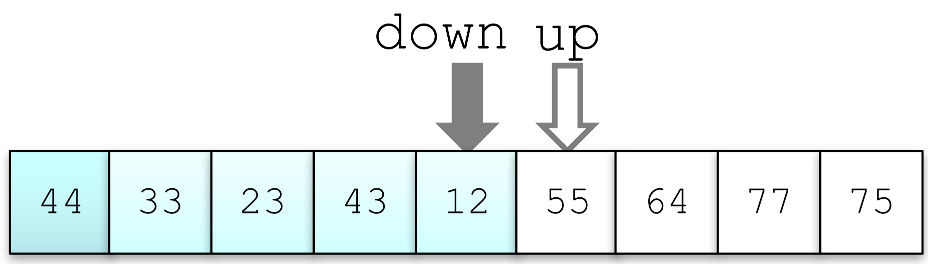
Find first element at left end greater than pivot

Trace of Partitioning (cont.)



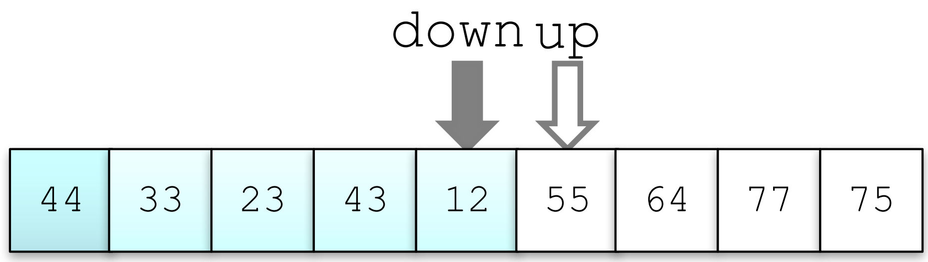
Find first element at right end less than or equal to pivot

Trace of Partitioning (cont.)



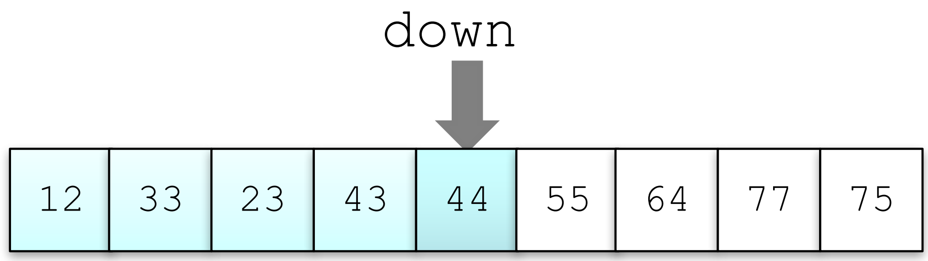
Since `down` has "passed" `up`, do not exchange

Trace of Partitioning (cont.)



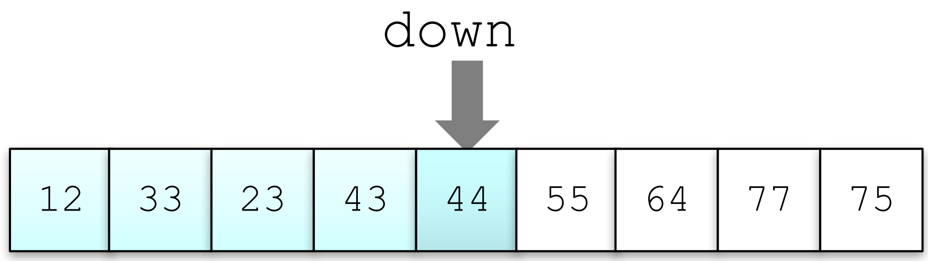
Exchange the pivot value with the value at `down`

Trace of Partitioning (cont.)



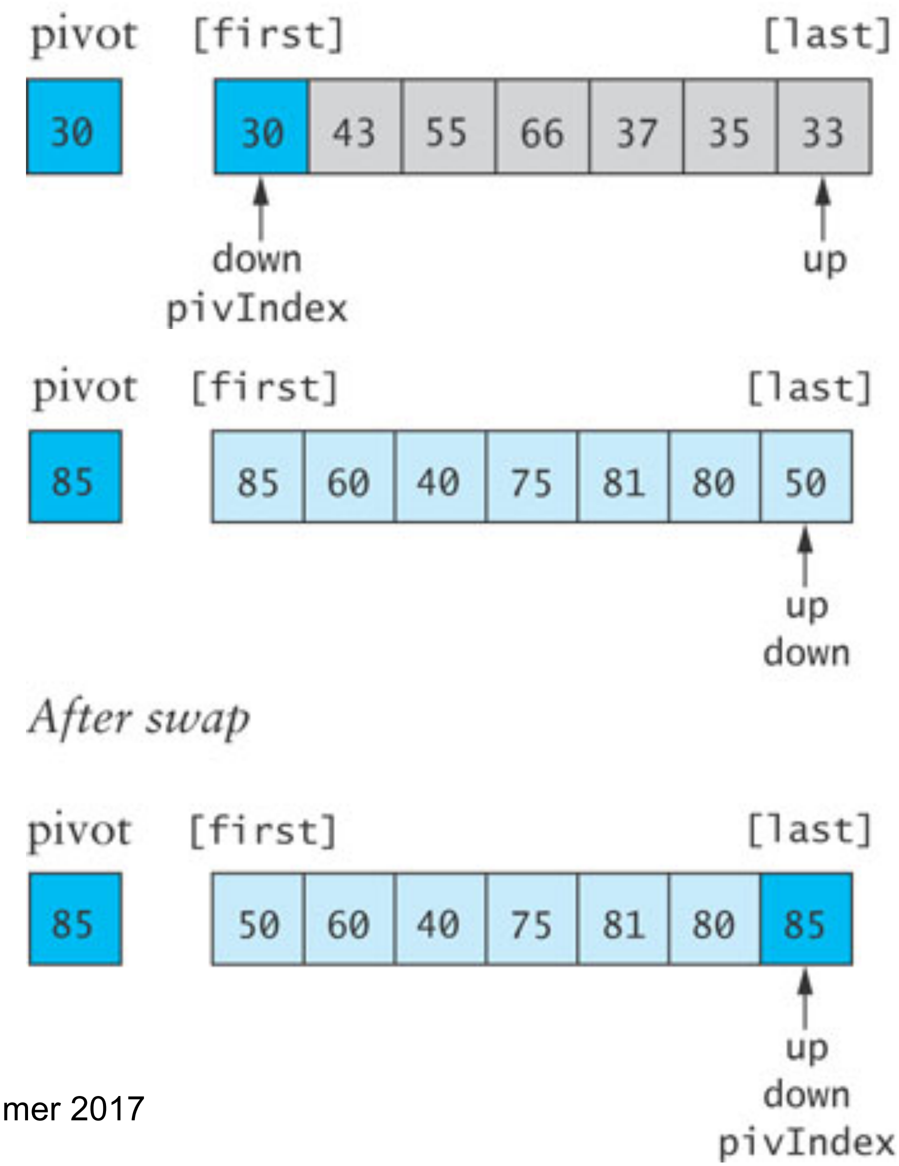
Exchange the pivot value with the value at `down`

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`

Code for `partition` when `Pivot` is the largest or smallest value



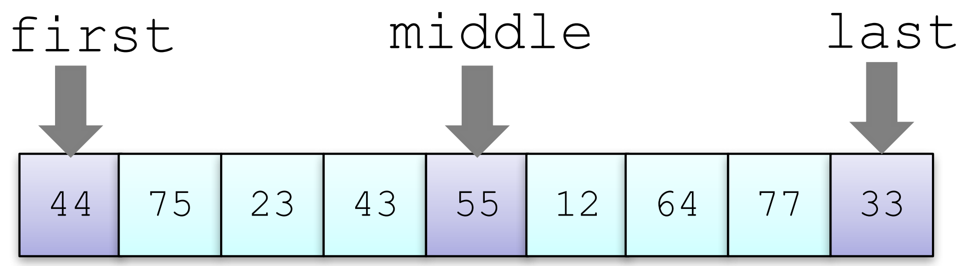
Revised Partition Algorithm

- 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

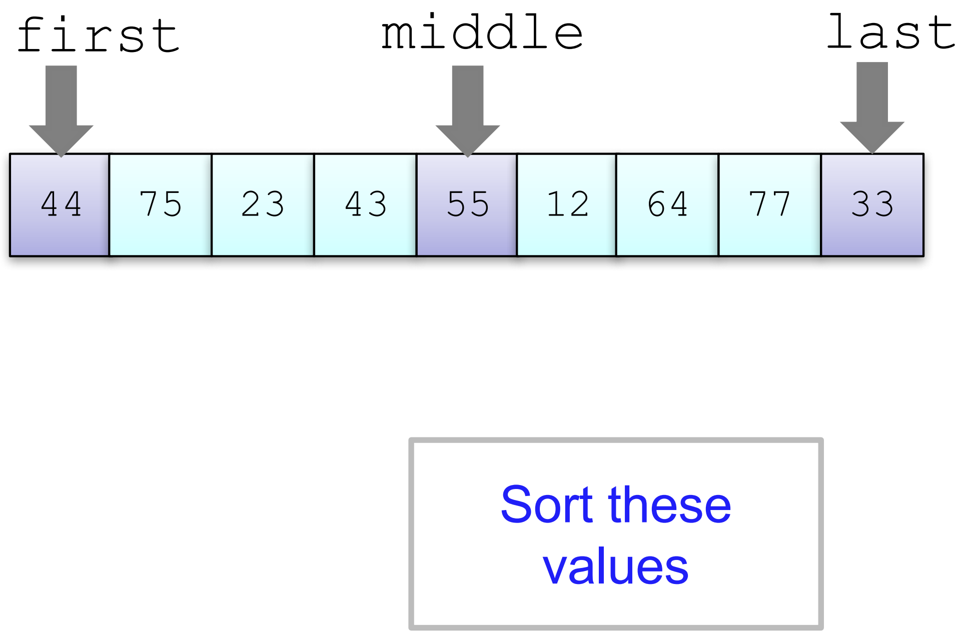
Trace of Revised Partitioning

44	75	23	43	55	12	64	77	33

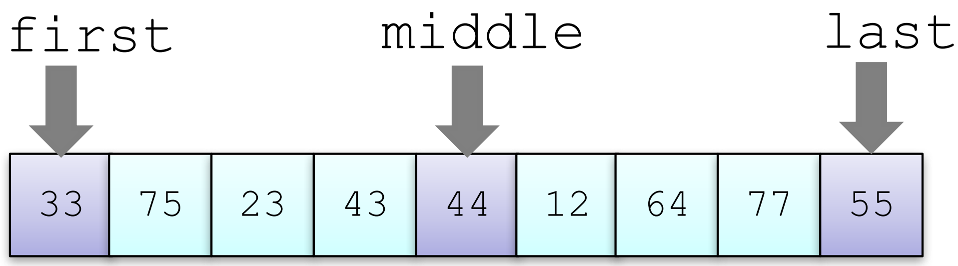
Trace of Revised Partitioning (cont.)



Trace of Revised Partitioning (cont.)

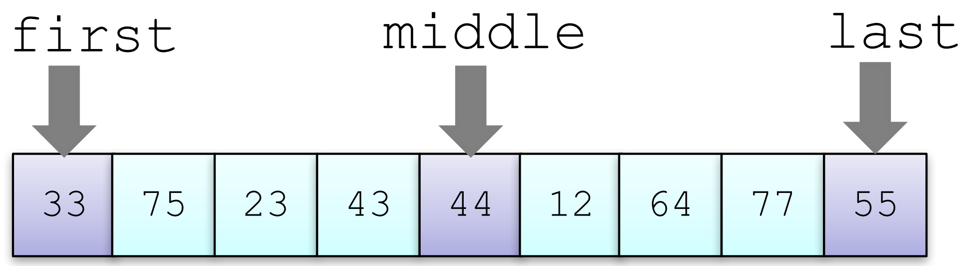


Trace of Revised Partitioning (cont.)



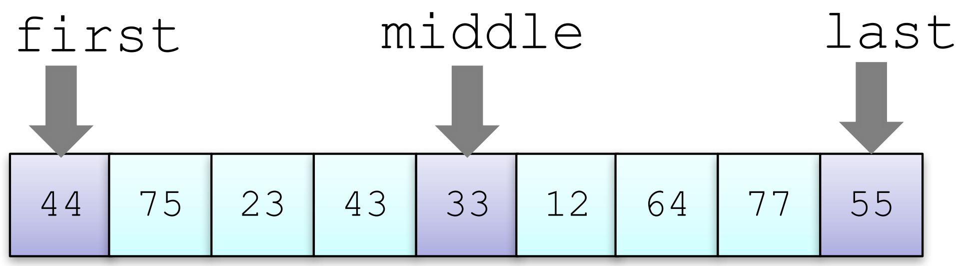
Sort these values

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

Sorting Algorithm Comparison

Name	Best	Averag e	Worst	Memory	Stable
Bubble Sort	*n*	n^2	n^2	1	yes
Selection Sort	n^2	n^2	n^2	1	no
Insertion Sort	n	n^2	n^2	1	yes
Merge Sort	*nlogn*	*nlogn*	*nlogn*	*n*	yes
Quick Sort	*nlogn*	*nlogn*	n^2	*log n*	no
Heap Sort	*nlogn*	*nlogn*	*nlogn*	1	no