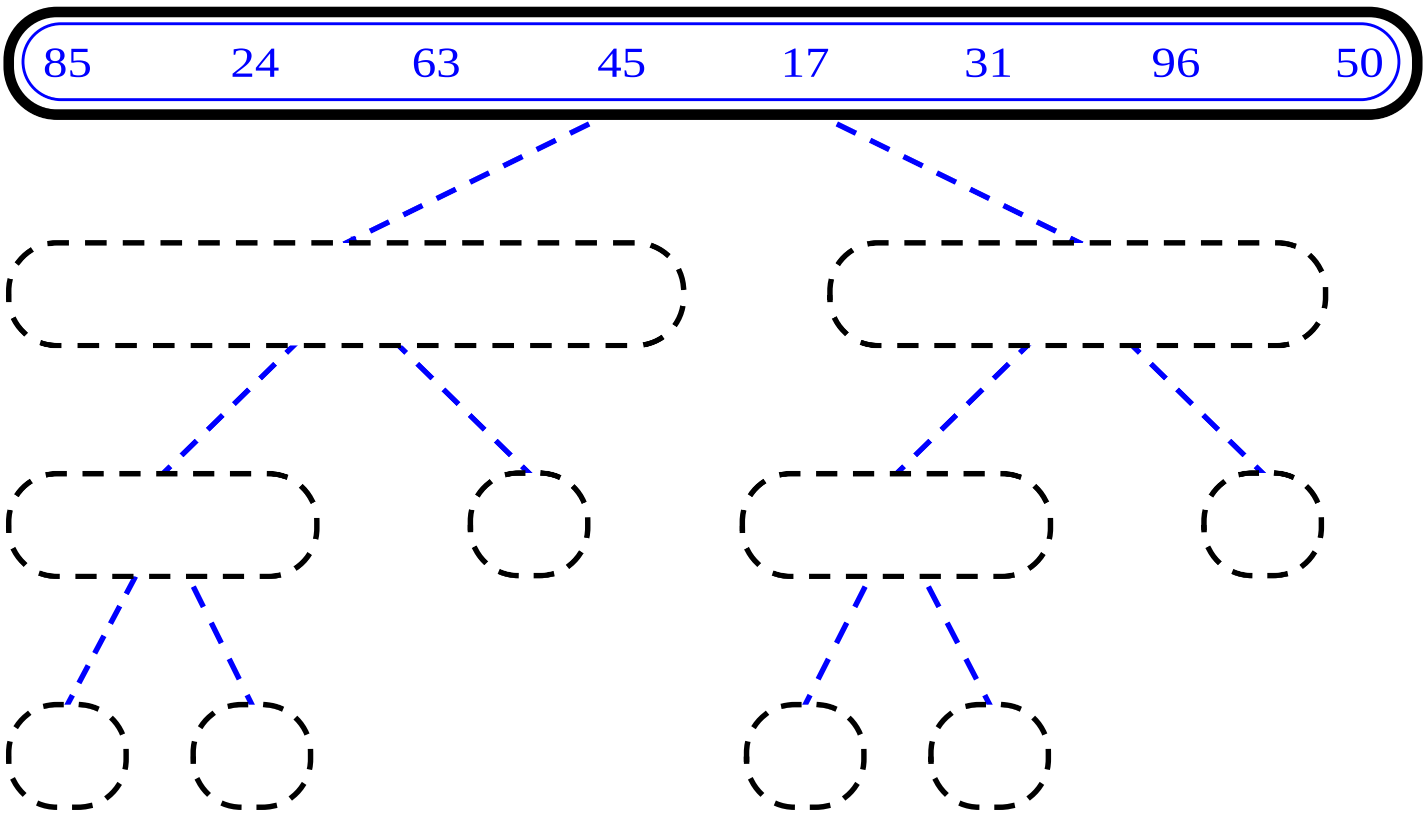
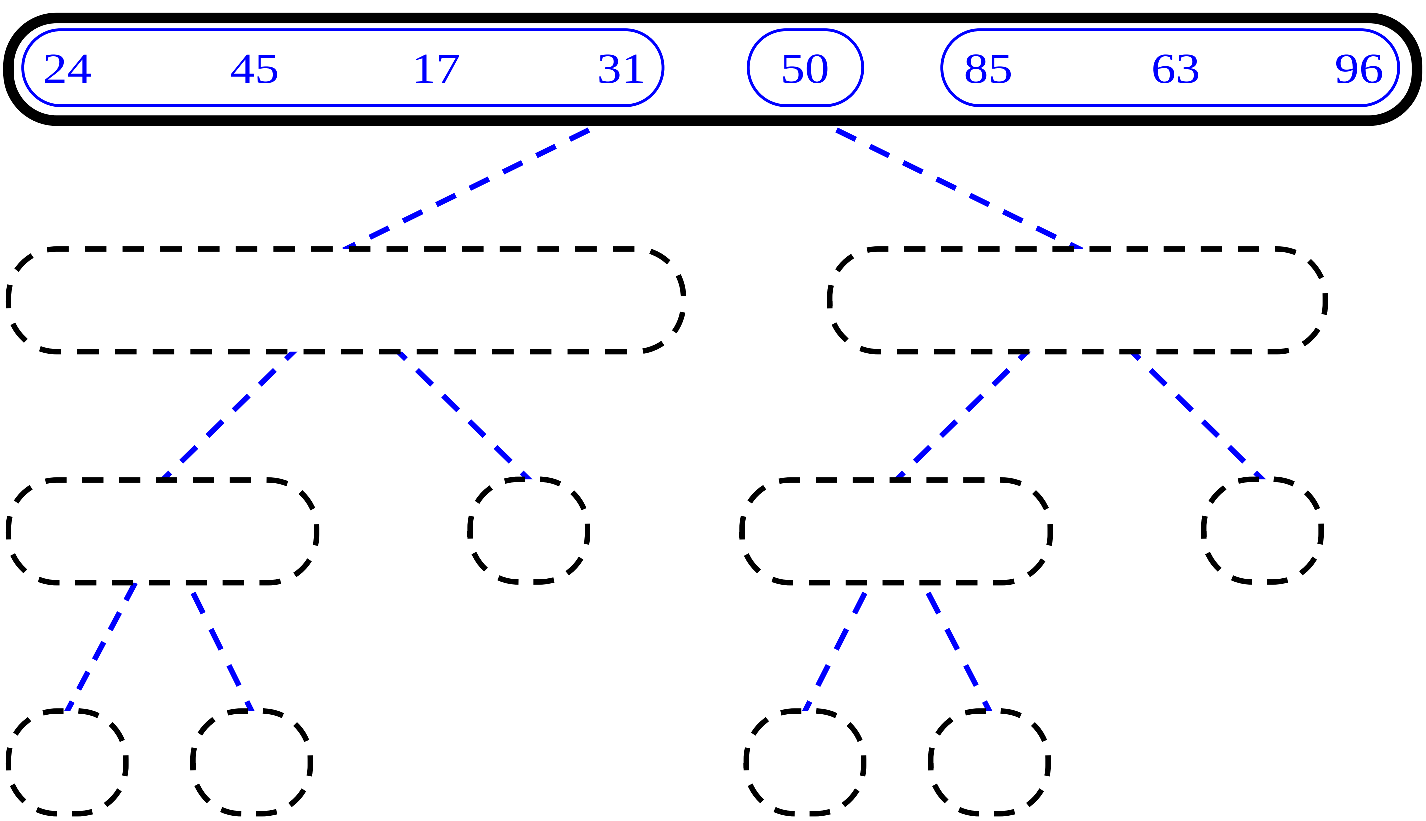
Quick-Sort

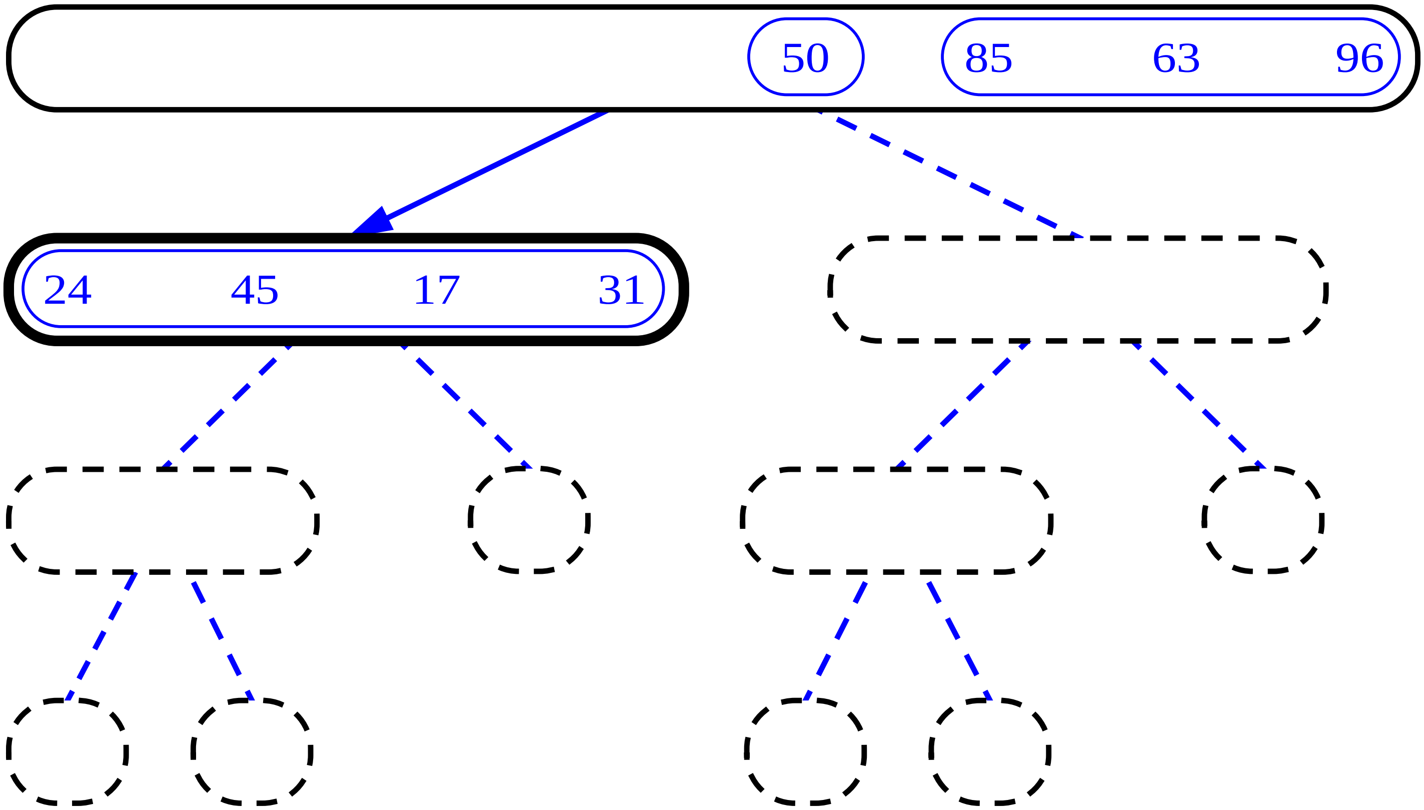
- To understand quick-sort, let's look at a high-level description of the algorithm
- 1) **Divide** : If the sequence *S* has 2 or more elements, select an element *x* from *S* to you pivot. Any arbitrary element, like the last, will do. Remove all the elements of *S* and divide them into 3 sequences:
  - *L*, holds *S*'s elements less than *x*
  - *E*, holds *S*'s elements equal to *x*
  - *G*, holds *S*'s elements greater than *x*
- 2) **Recurse**: Recursively sort *L* and *G*
- 3) **Conquer**: Finally, to put elements back into *S* in order, first inserts the elements of *L*, then those of *E*, and those of *G*.
- Here are some pretty diagrams....

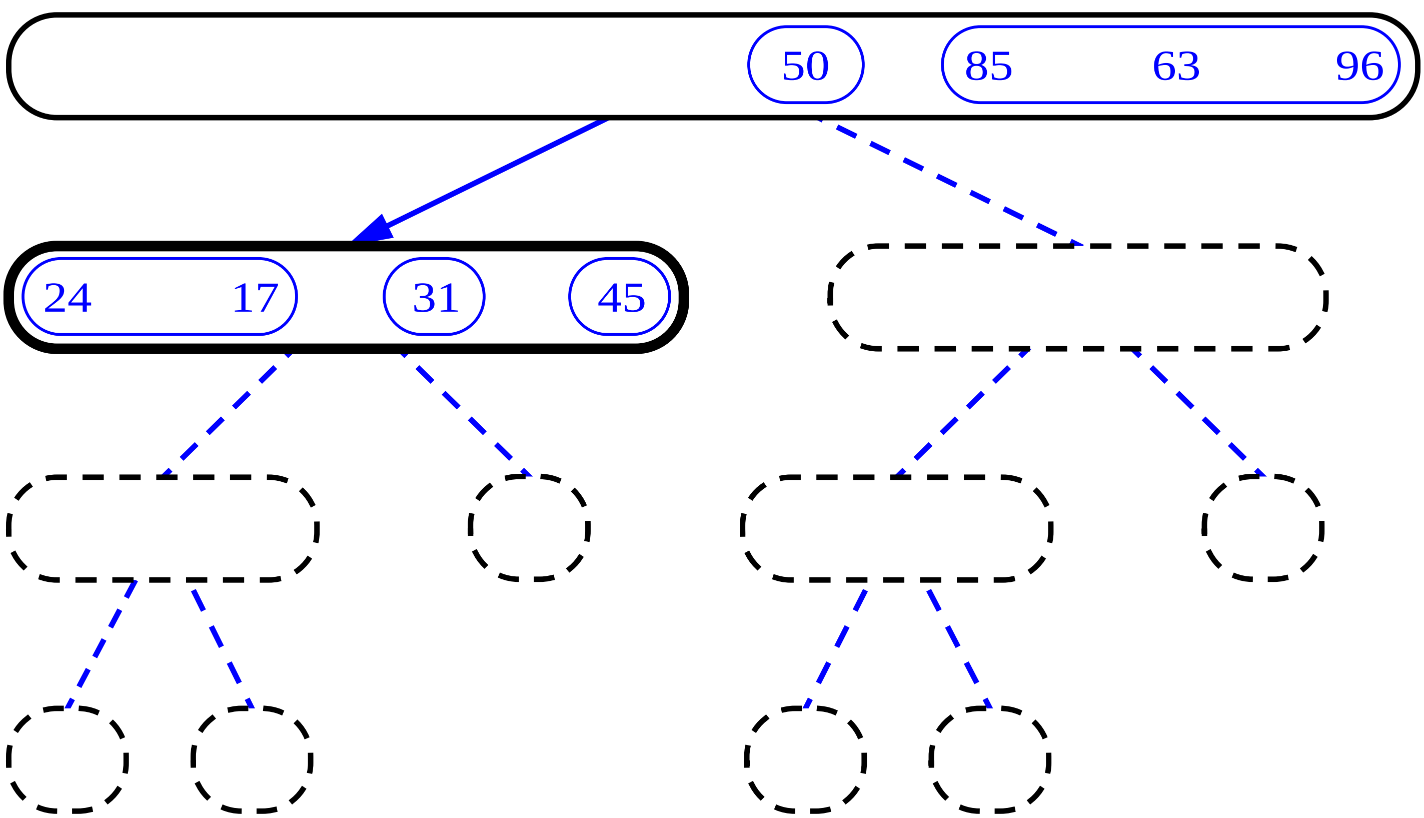
Quick-Sort Tree



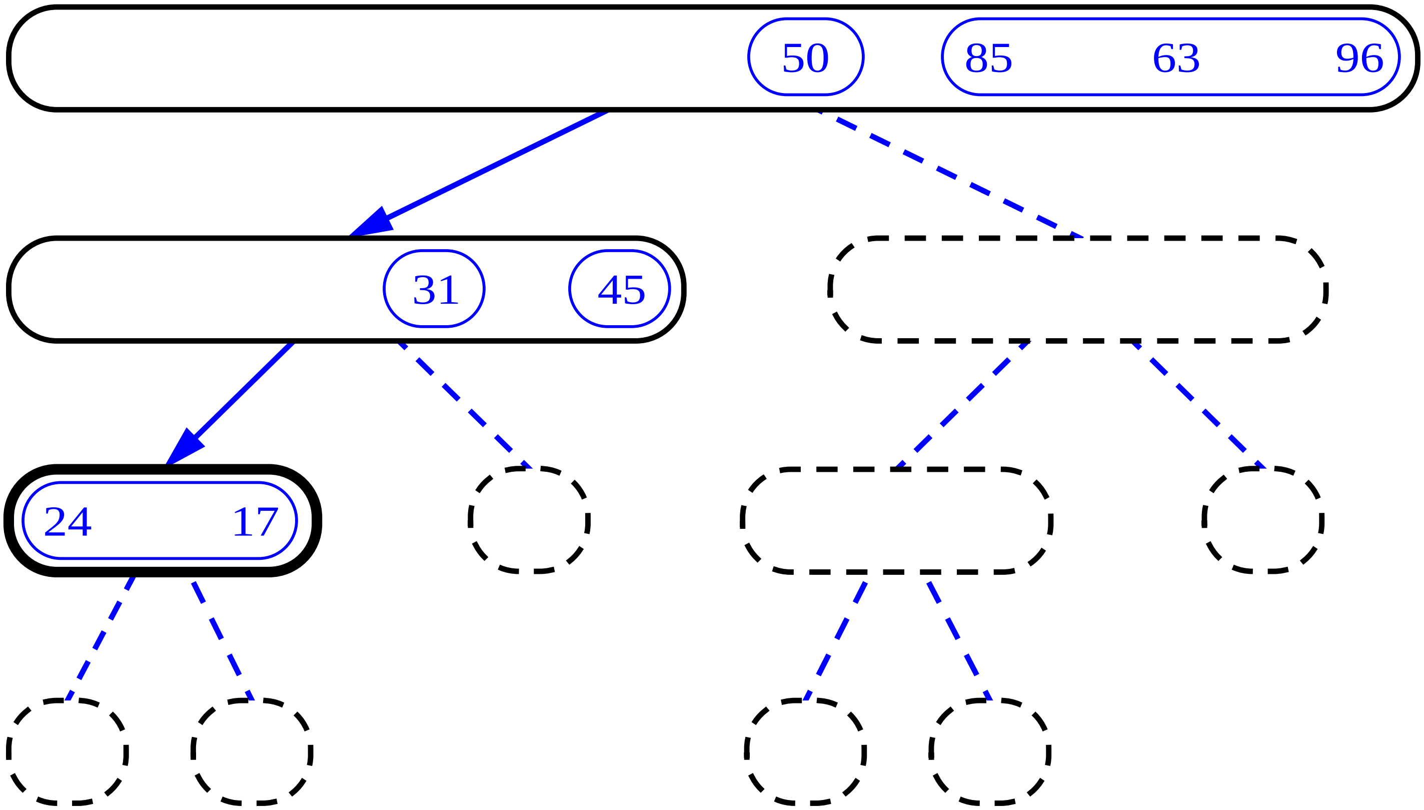


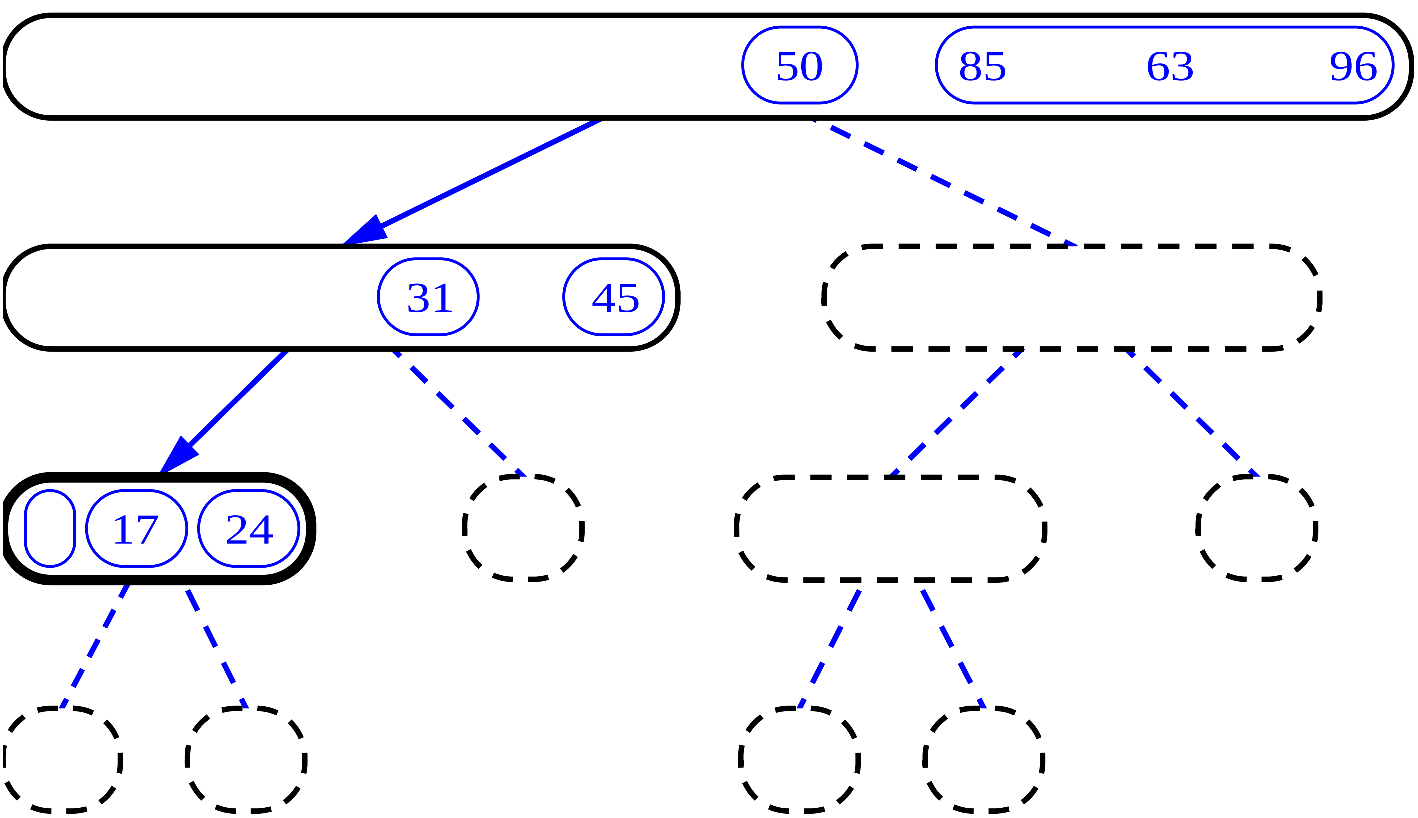
Quick-Sort Tree (cont.)



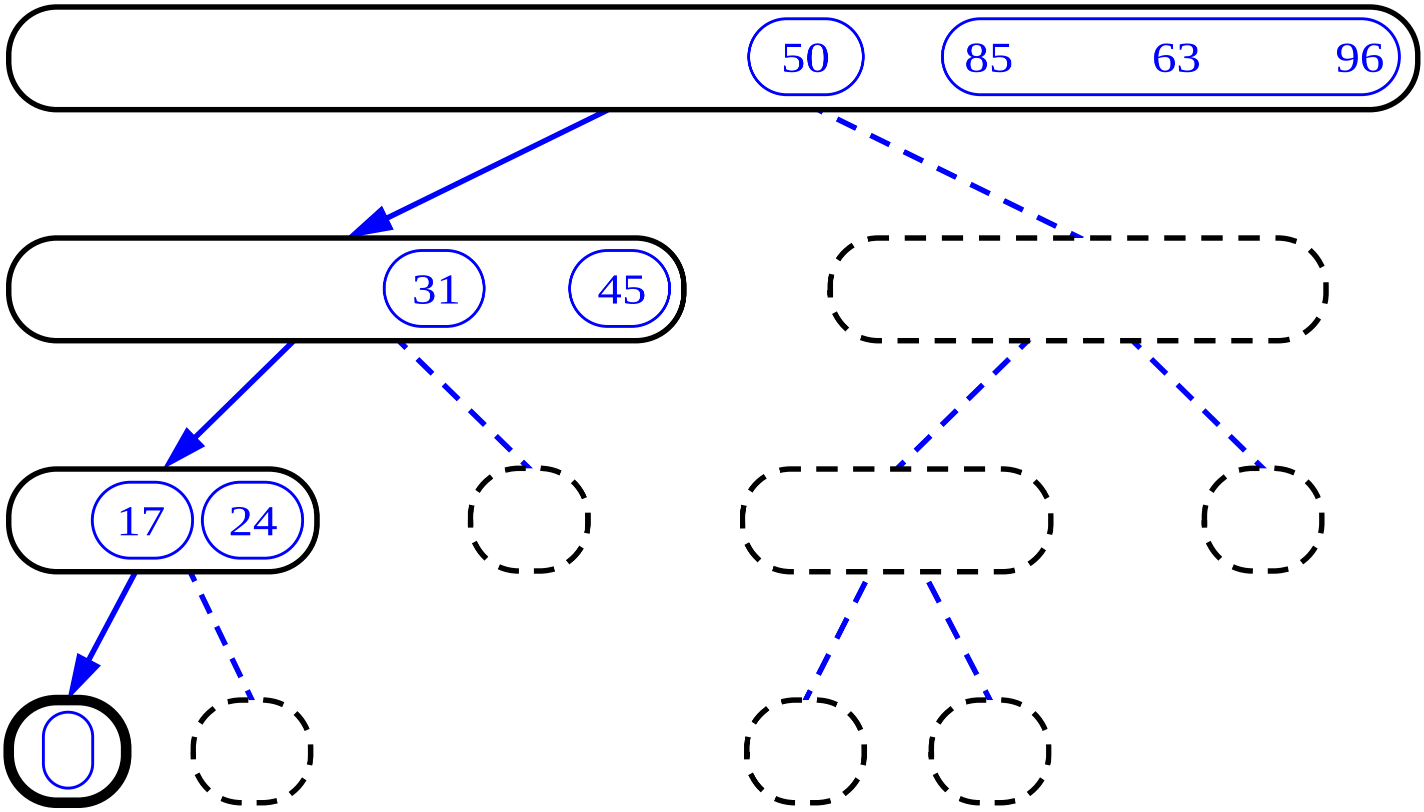


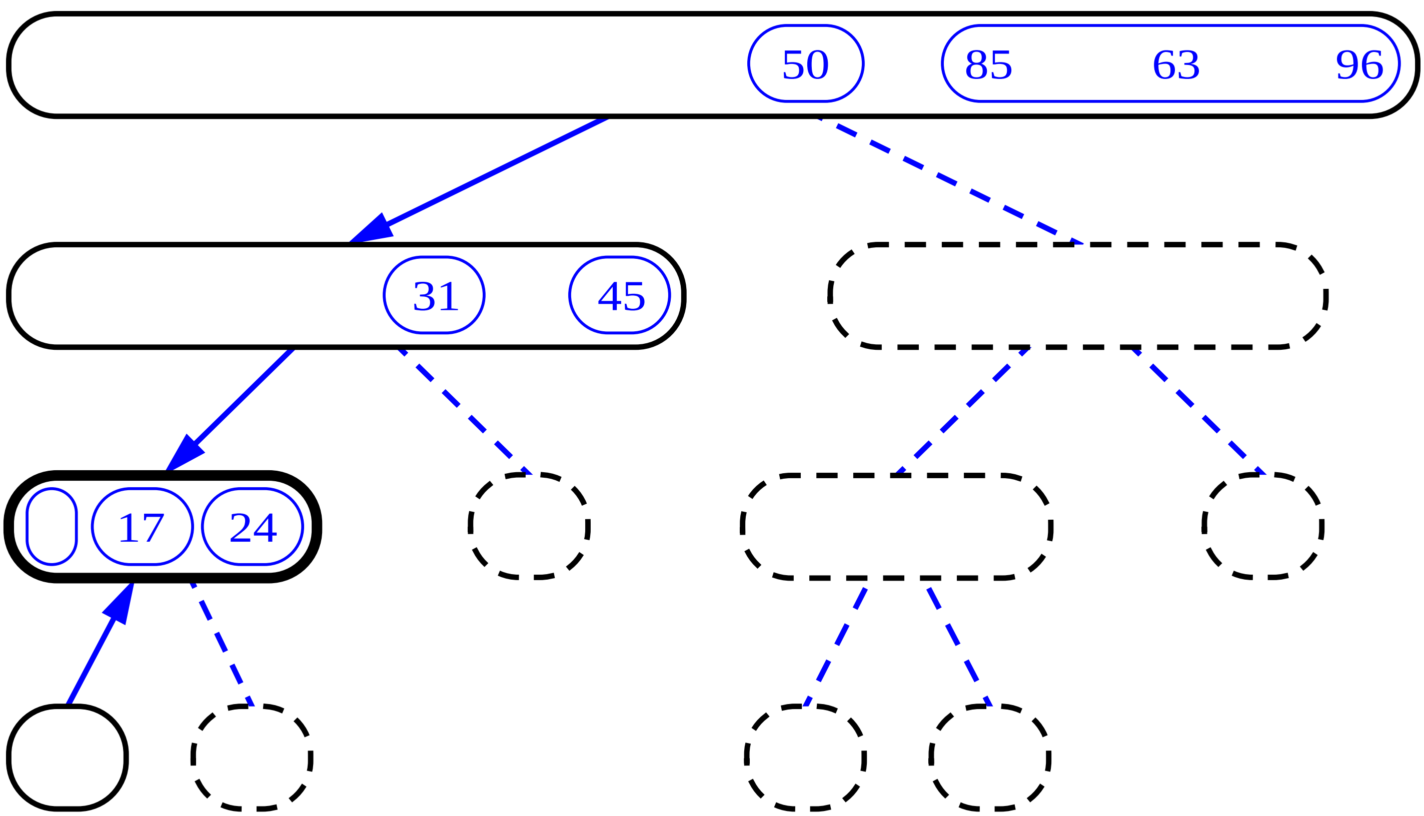
Quick-Sort Tree (cont.)



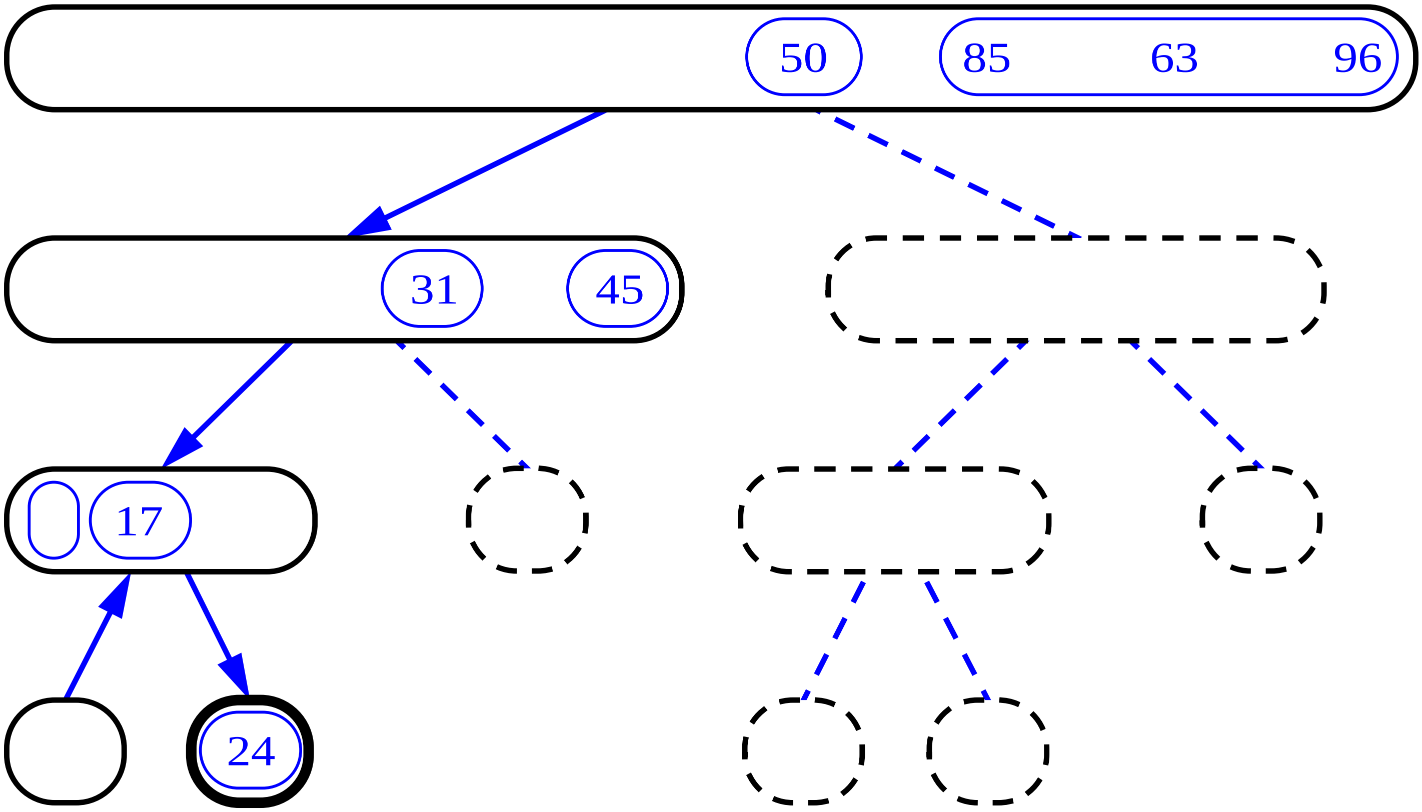


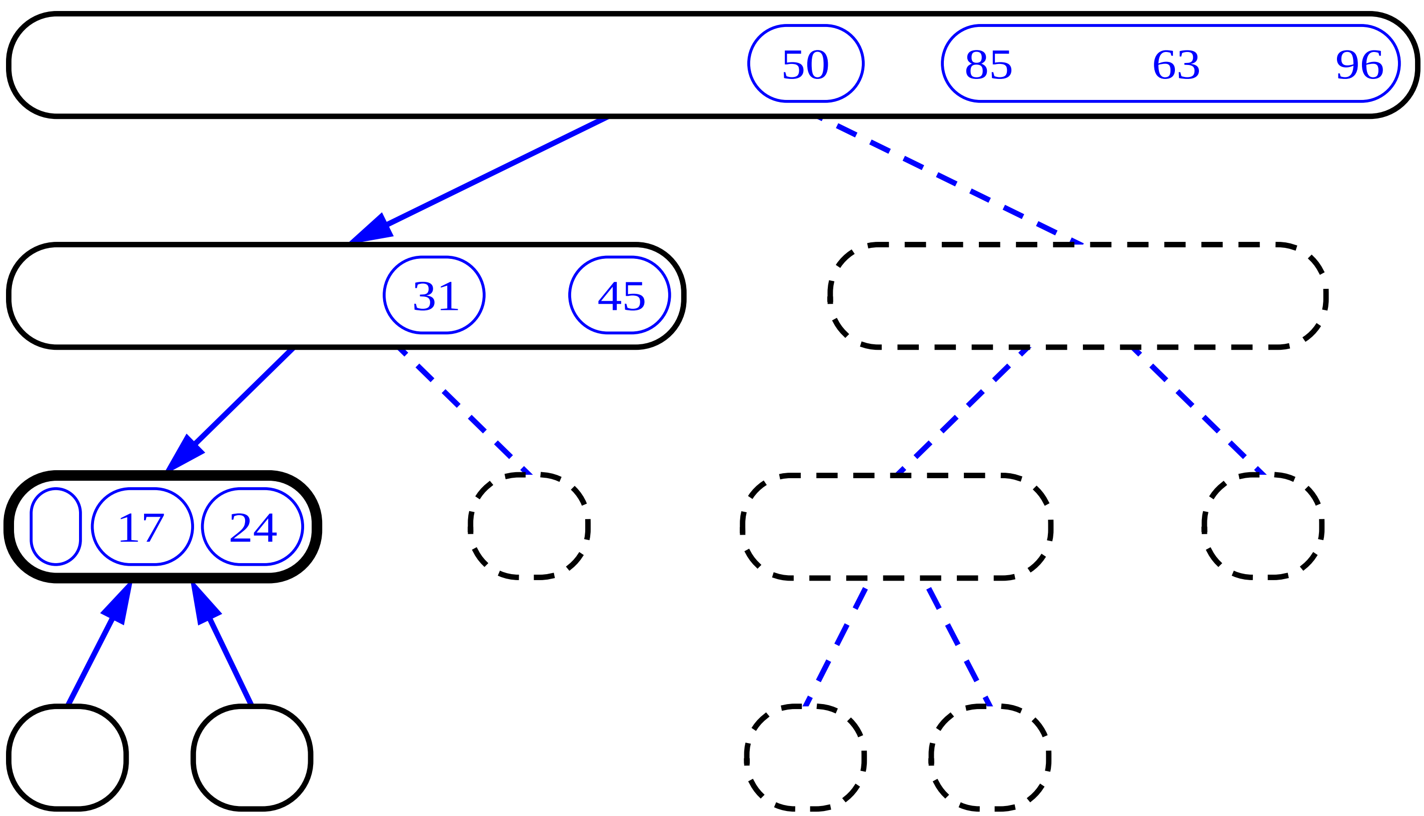
Quick-Sort Tree (cont.)



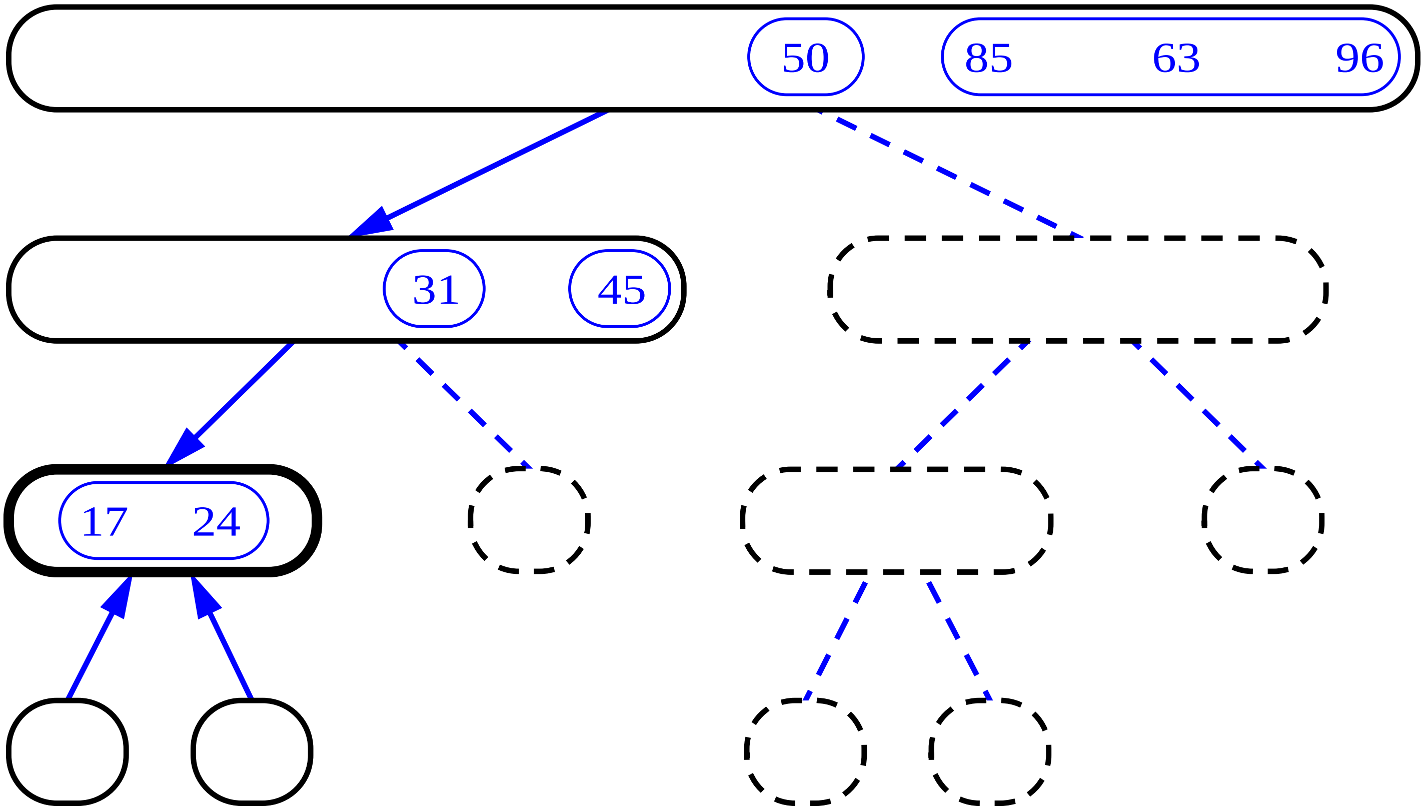


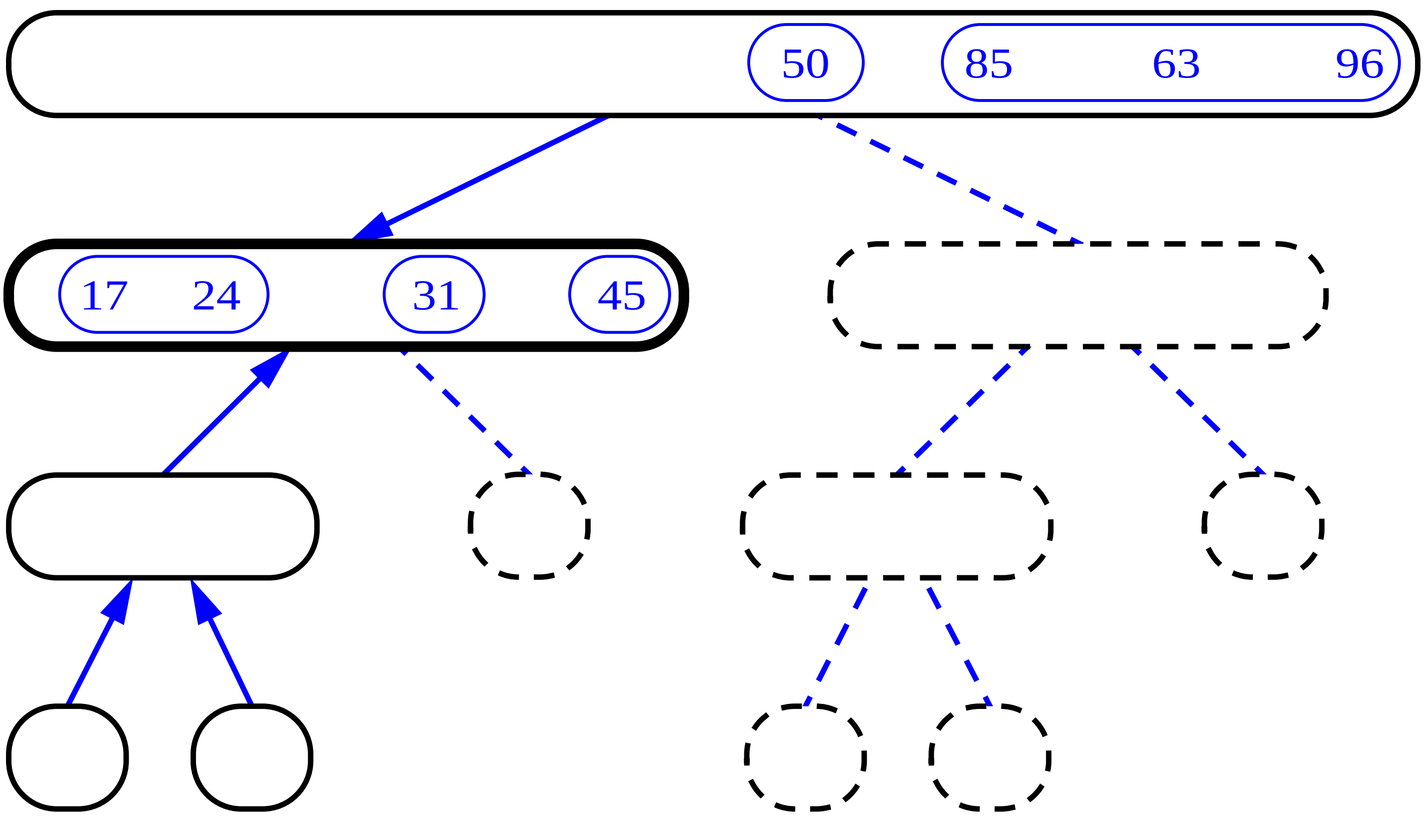
Quick-Sort Tree (cont.)



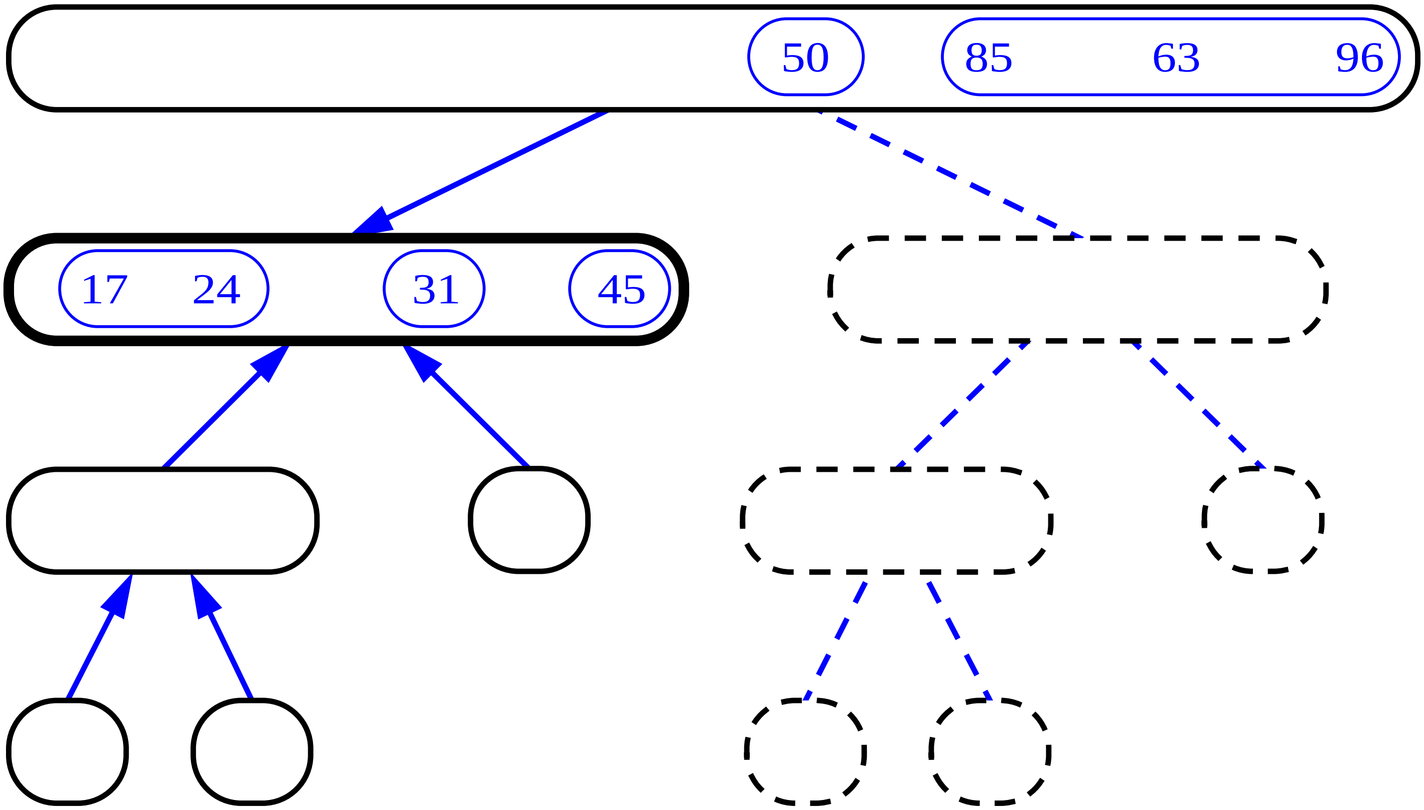


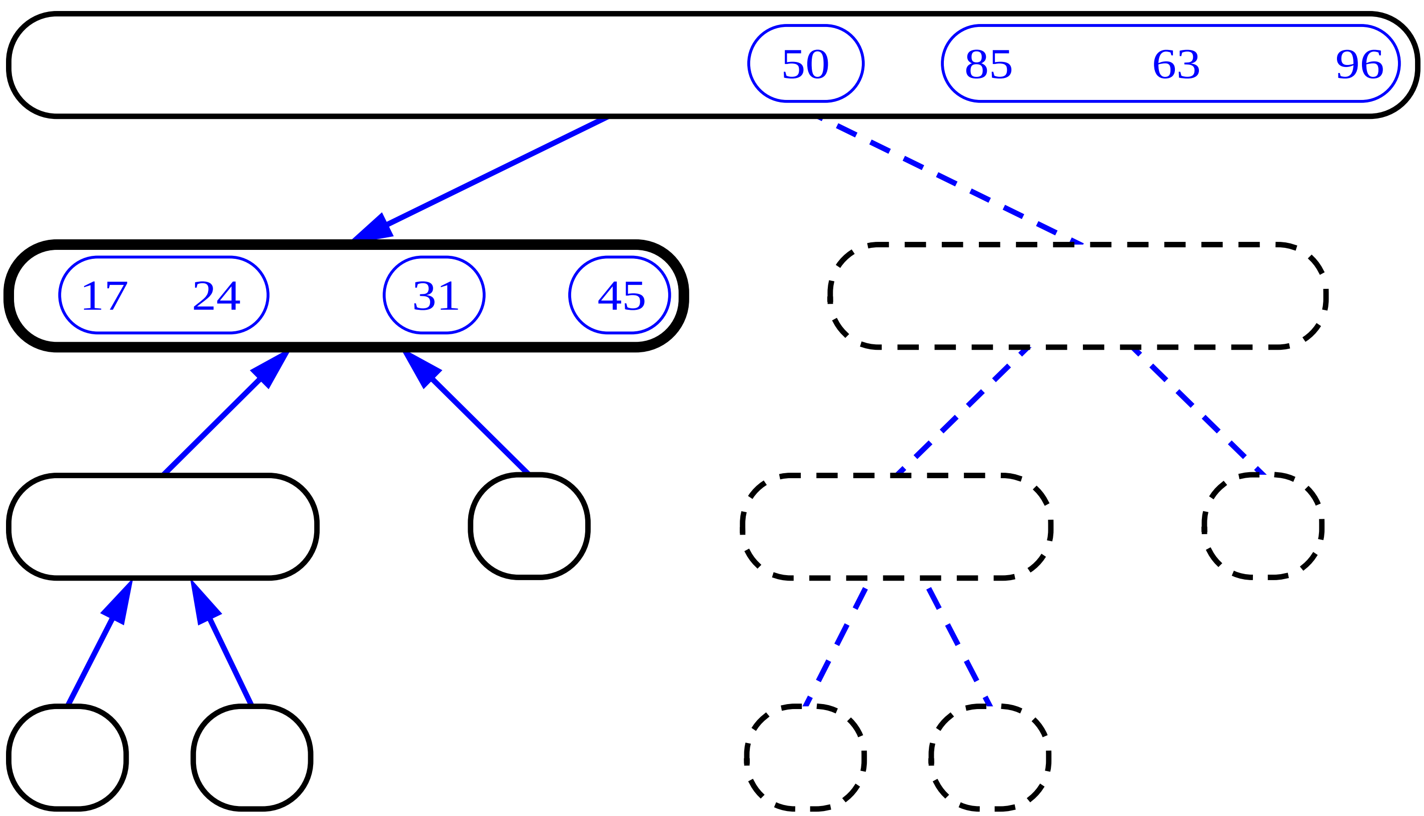
Quick-Sort Tree (cont.)



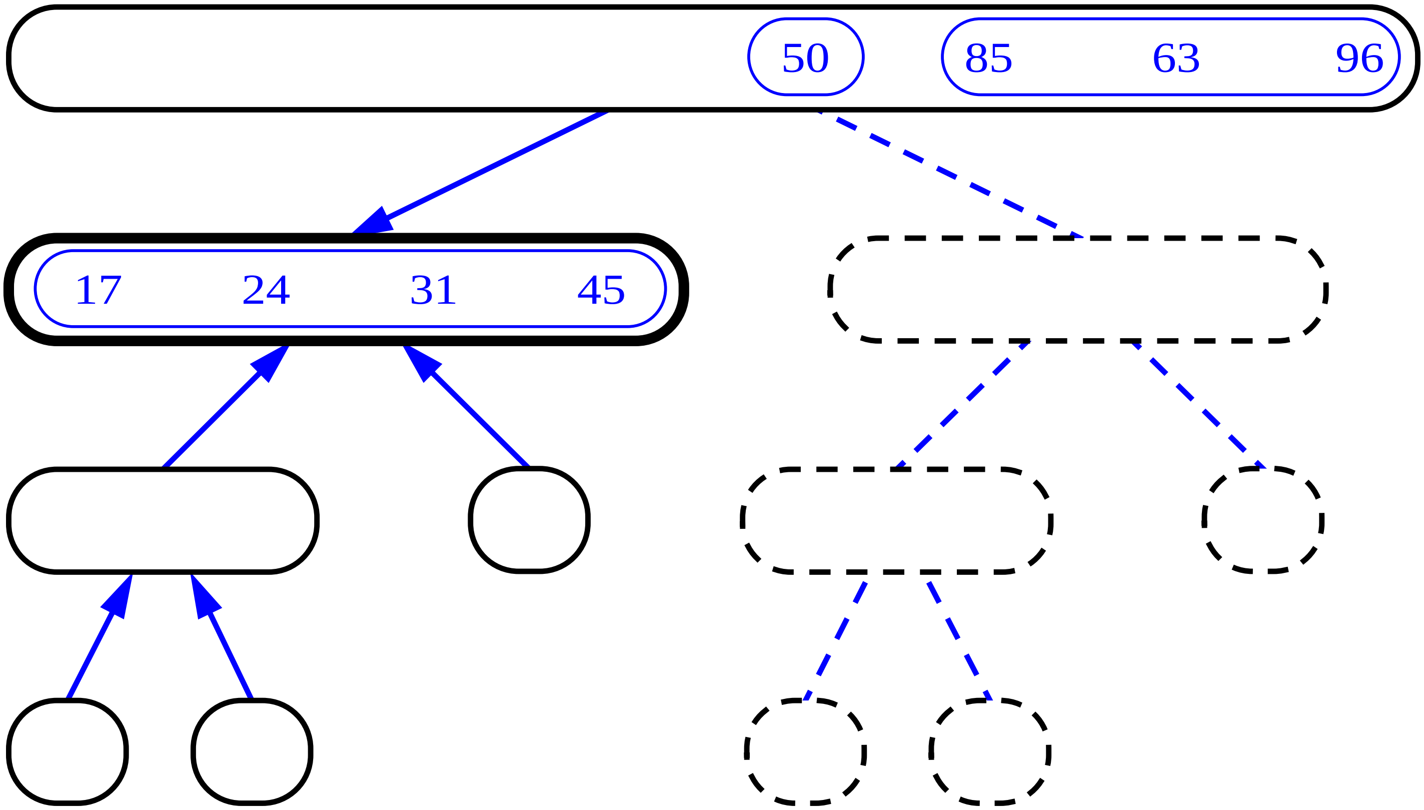


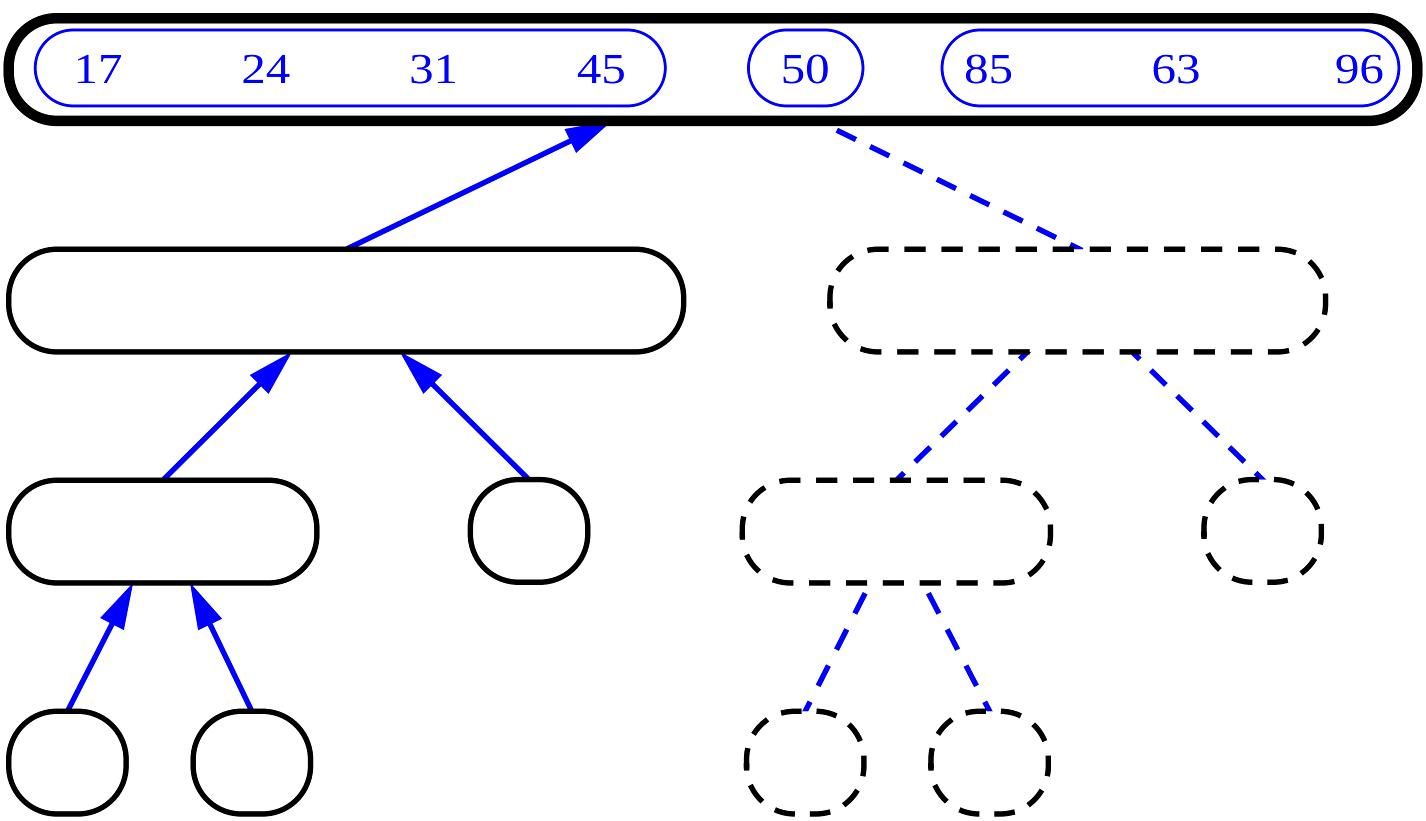
Quick-Sort Tree (cont.)



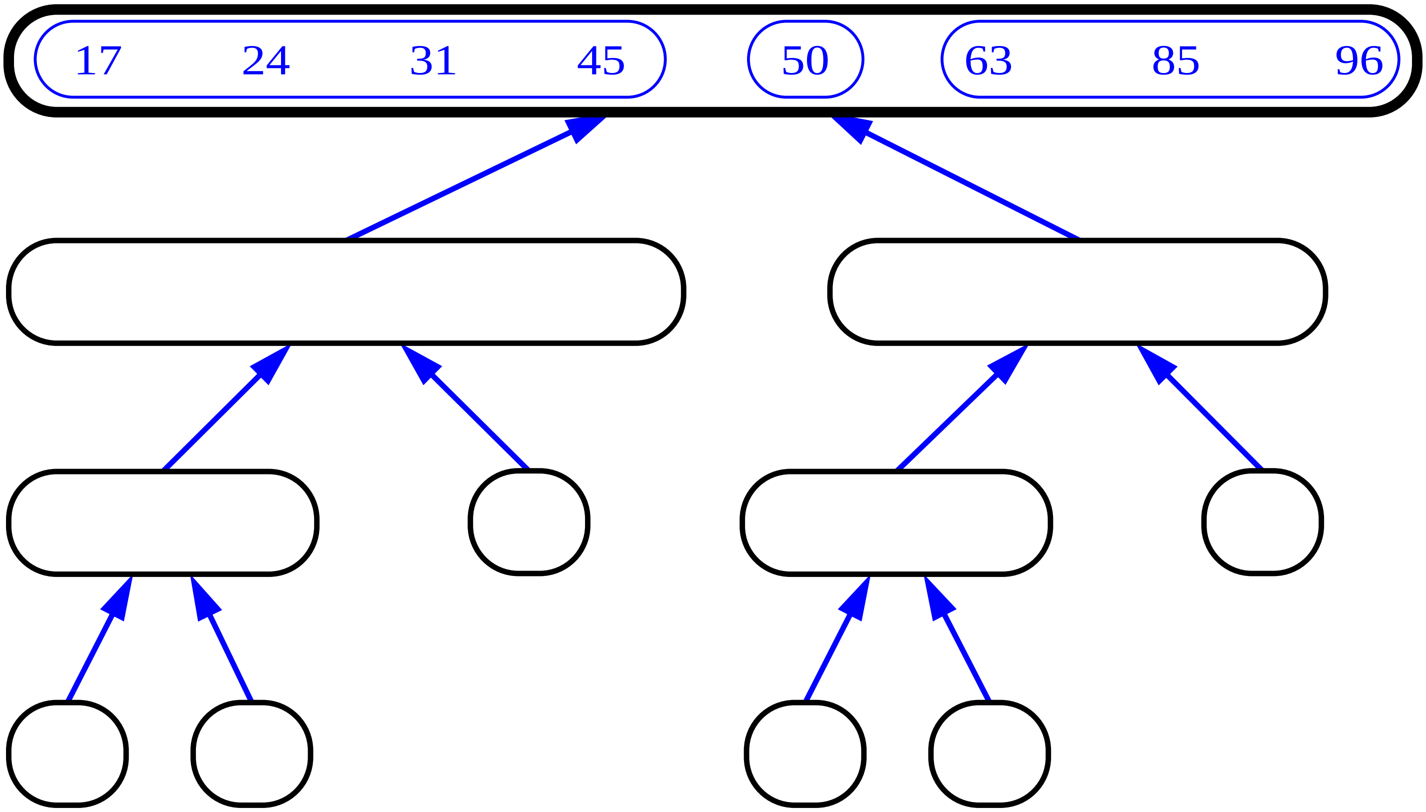


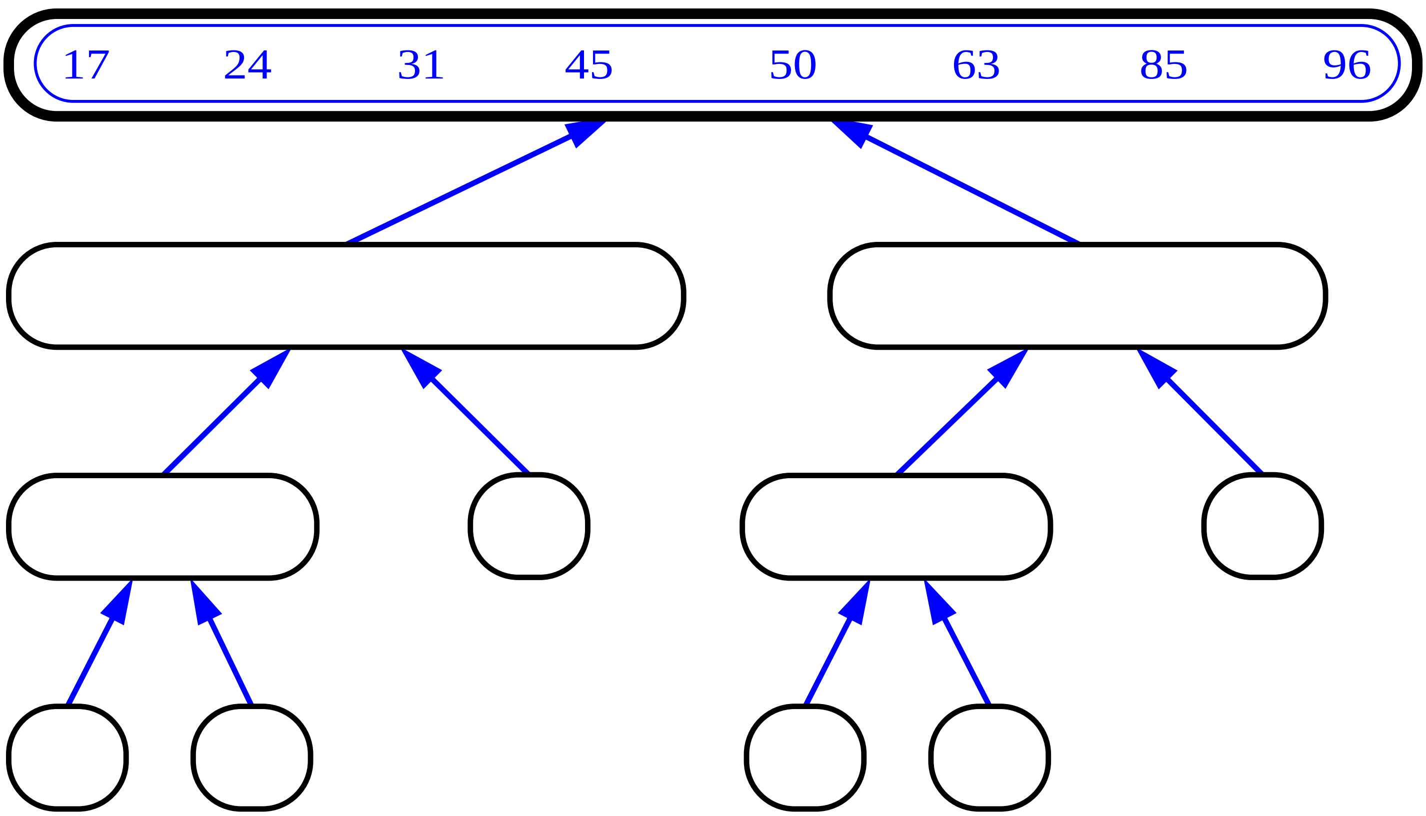
Quick-Sort Tree (cont.)





Quick-Sort Tree (cont.)





Analysis of Running Time

- Consider a quick-sort tree T:
  - Let $s_i(n)$ denote the sum of the input sizes of the nodes at depth i in T.
- We know that $s_0(n) = n$ since the root of T is associated with the entire input set.
- Also, $s_1(n) = n - 1$ since the pivot is not propagated.
- Thus: either $s_2(n) = n - 3$, or $n - 2$ (if one of the nodes has a zero input size).
- The worst case running time of a quick-sort is then:

$$\boldsymbol{O}\left(\sum_{i=0}^{n-1} s_i(n)\right)$$

Which reduces to:

$$\boldsymbol{O}\left(\sum_{i=0}^{n-1} (n-i)\right) = \boldsymbol{O}\left(\sum_{i=1}^{n} i\right) = \boldsymbol{O}(n^2)$$

- Thus quick-sort runs in time $\boldsymbol{O}(n^2)$ in the worst case.

Analysis of Running Time (contd.)

- Now to look at the best case running time:
- We can see that quicksort behaves optimally if, whenever a sequence S is divided into subsequences L and G, they are of equal size.
- More precisely:
  - $s_0(n) = n$
  - $s_1(n) = n - 1$
  - $s_2(n) = n - (1 + 2) = n - 3$
  - $s_3(n) = n - (1 + 2 + 2^2) = n - 7$

    ...
  - $s_i(n) = n - (1 + 2 + 2^2 + ... + 2^i - 1) = n - 2^i + 1$

    ...
- This implies that T has height $\boldsymbol{O}(\log n)$
- Best Case Time Complexity: $\boldsymbol{O}(n\log n)$

Randomized Quick-Sort

- The main drawback to quick-sort is that it achieves its worst-case time complexity on data sets that are common in practice: sequences that are already sorted (or mostly sorted)
- To avoid this, we modify quick-sort so that it selects the pivot as a *random* element of the sequence
- The expected time of a randomized quick-sort on a sequence of size n is $\boldsymbol{O}(n\log n)$.
- Justification: we say that an invocation of quicksort, on an input sequence of size m is "good" if neither L nor G is less than $m/4$.
  - there are $m/2$ "good" pivots and m/2 "bad" ones
  - The probablility that an invocation is "good" is 1/2
  - Suppose we choose a good pivot at node v: the algorithm recurs on sequences with size at most $(3/4)m$ each
  - On average, the height of the tree representing a randomized quick-sort is at most $2\log_{4/3} n$
- Total time complexity: $\boldsymbol{O}(n \log n)$

In-Place Quick-Sort

- **Divide step**: *l* scans the sequence from the left, and *r* from the right.

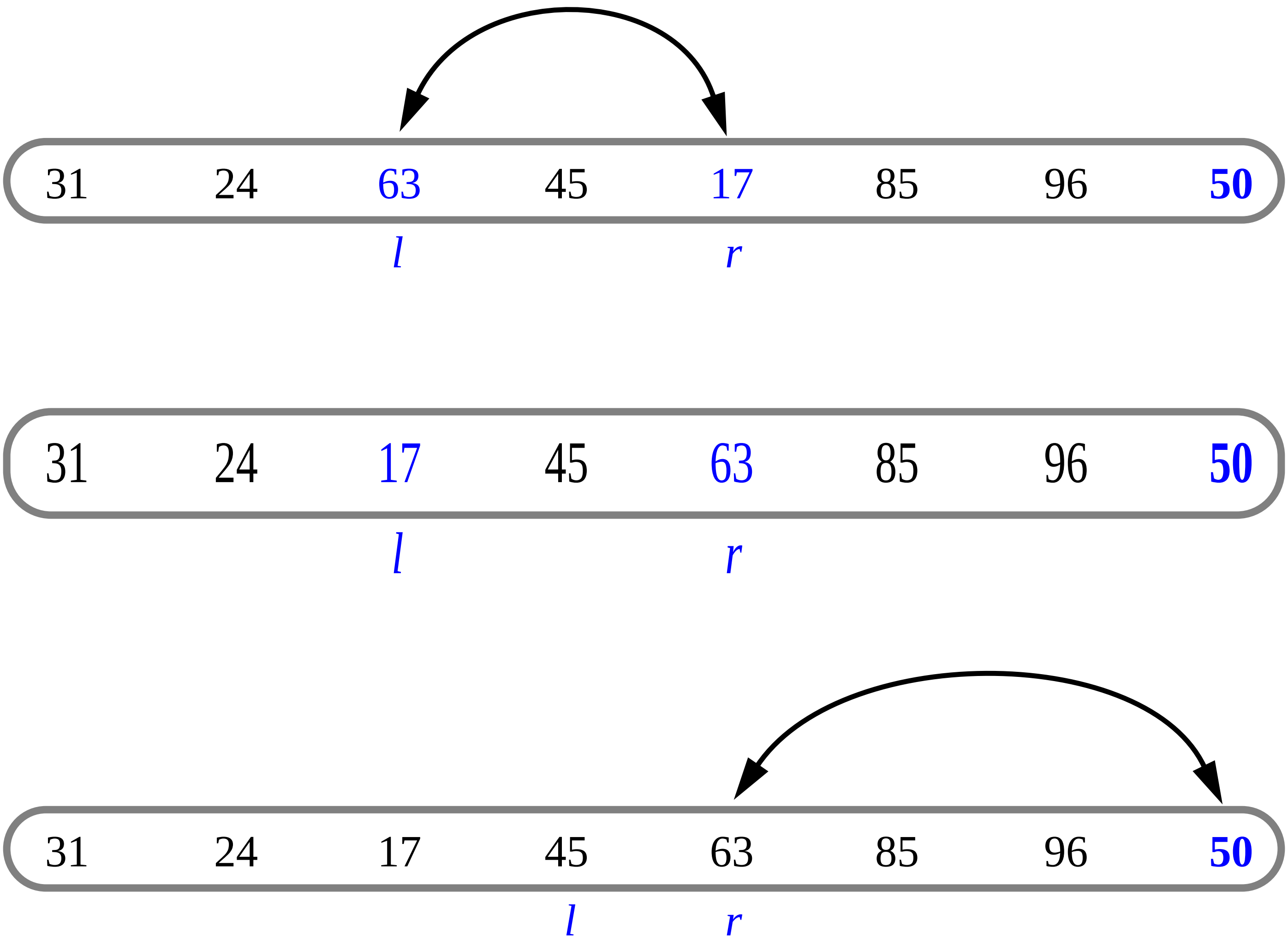
85	24	63	45	17	31	96	**50**
l						*r*	

- A swap is performed when *l* is at an element larger than the pivot and *r* is at one smaller than the pivot.

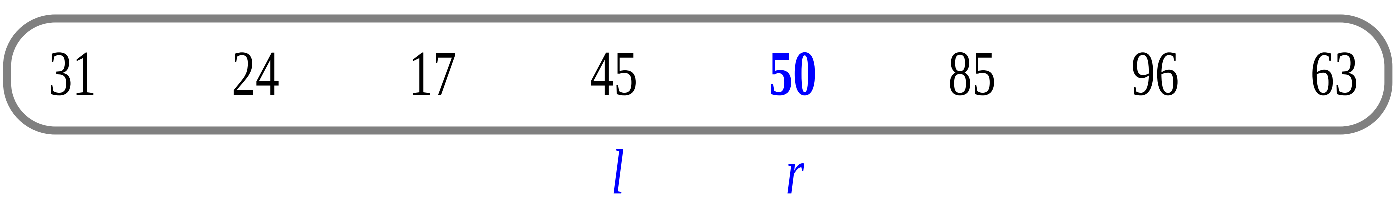
85	24	63	45	17	31	96	**50**
l					*r*		

31	24	63	45	17	85	96	**50**
l					*r*		

In Place Quick Sort (contd.)



- A final swap with the pivot completes the divide step



In Place Quick Sort (contd.)

- pseude-code fragment 8.7

How Fast Can We Sort?

- **Proposition**: The running time of any comparison-based algorithm for sorting an *n*-element sequence S is Ω(*n*log *n*).
- **Justification**:
- The running time of a comparison-based sorting algorithm must be equal to or greater than the depth of the decision tree *T* associated with this algorithm.
- Each internal node of *T* is associated with a comparison that establishes the ordering of two elements of S.
- Thus every external node of *T* represents a distinct permutation of the elements of S.
- Hence *T* must have at least *n*! external nodes which again implies T has a height of at least log(*n*!)
- Since n! has at least n/2 terms that are greater than or equal to n/2, we can see:
- log(n!)   log(n/2)n/2 = (n/2)log(n/2)
- Total Time Complexity: Ω(*n*log *n*).

How Fast Can We Sort? (contd.)

- A graphical representation of a comparison-based algorithm's decision tree.