

Merge Sort

Divide-and-Conquer

Divide-and-conquer is a general algorithm design paradigm:

- **Divide**: divide the input data S in two disjoint subsets S_1 and S_2
- **Recur**: solve the subproblems associated with S_1 and S_2
 - the base case for the recursion are subproblems of size 0 or 1
- **Conquer**: combine the solutions for S_1 and S_2 into a solution for S

Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

- Like heap-sort
 - Uses a comparator
 - Has $O(n \log n)$ running time
- Unlike heap-sort
 - Does not use an auxiliary priority queue
 - Accesses data in a sequential manner (suitable to sort data on a disk)

Merge Sort

Merge-sort on an input sequence S with n elements consists of three steps:

- **Divide**: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
- **Recur**: recursively sort S_1 and S_2
- **Conquer**: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

mergeSort(S_1, C)

mergeSort(S_2, C)

$S \leftarrow merge(S_1, S_2)$

Merging two sorted sequences

The **conquer** step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B

Merging two sorted sequences, each with $n/2$ elements, takes $O(n)$ time

Algorithm *merge*(A, B)

Input sequences A and B with $n/2$ elements each

Output sorted sequence of $A \cup B$

$S \leftarrow$ empty sequence

while $\neg A.isEmpty() \wedge \neg B.isEmpty()$

if $A.first().element() < B.first().element()$

$S.insertLast(A.remove(A.first()))$

else

$S.insertLast(B.remove(B.first()))$

while $\neg A.isEmpty()$

$S.insertLast(A.remove(A.first()))$

while $\neg B.isEmpty()$

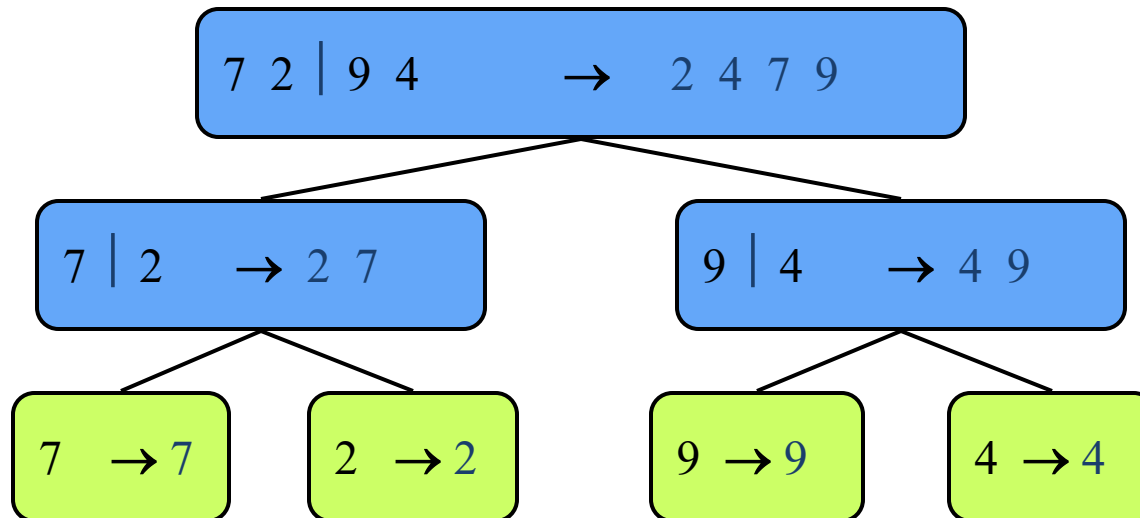
$S.insertLast(B.remove(B.first()))$

return S

Merge-Sort Tree

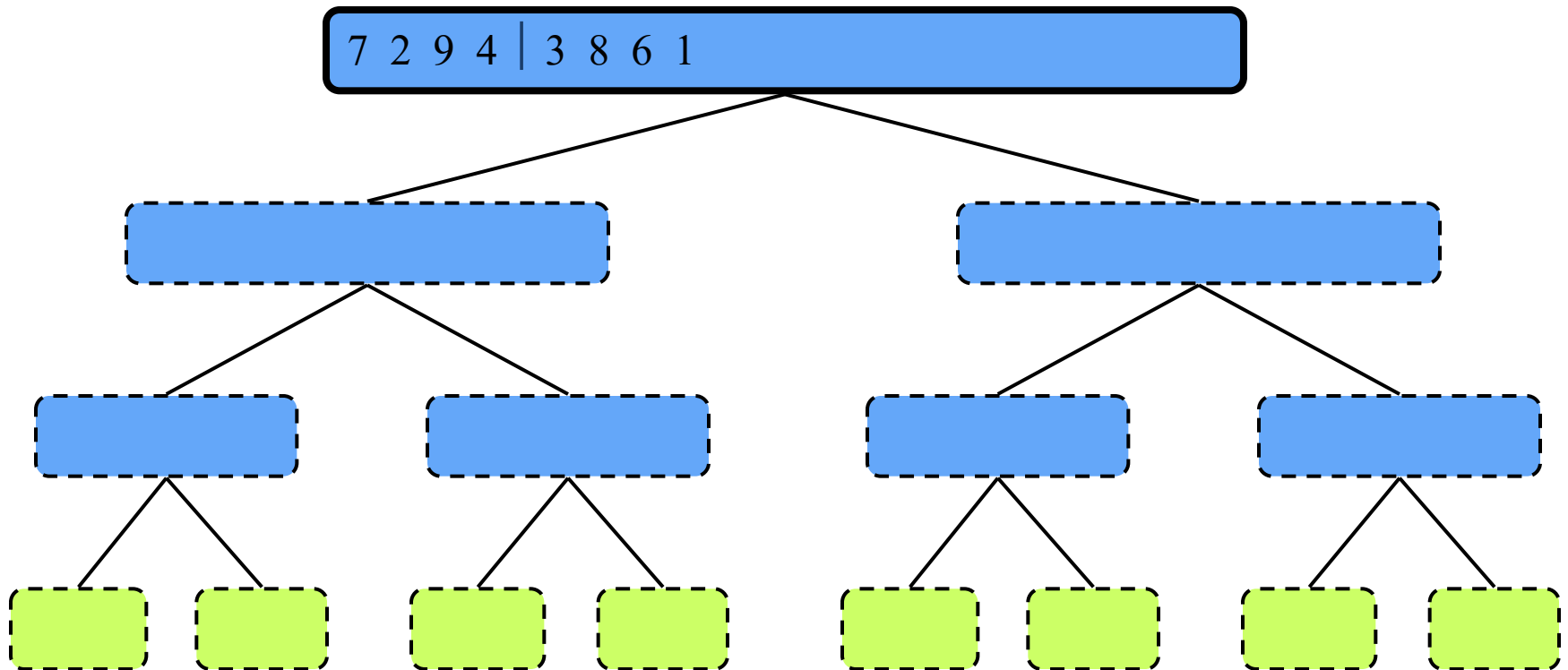
An execution of merge-sort is depicted by a binary tree

- each node represents a recursive call of merge-sort and stores
 - unsorted sequence **before** the execution and its partition
 - sorted sequence at the **end** of the execution
- the root is the initial call
- the leaves are calls on subsequences of size 1



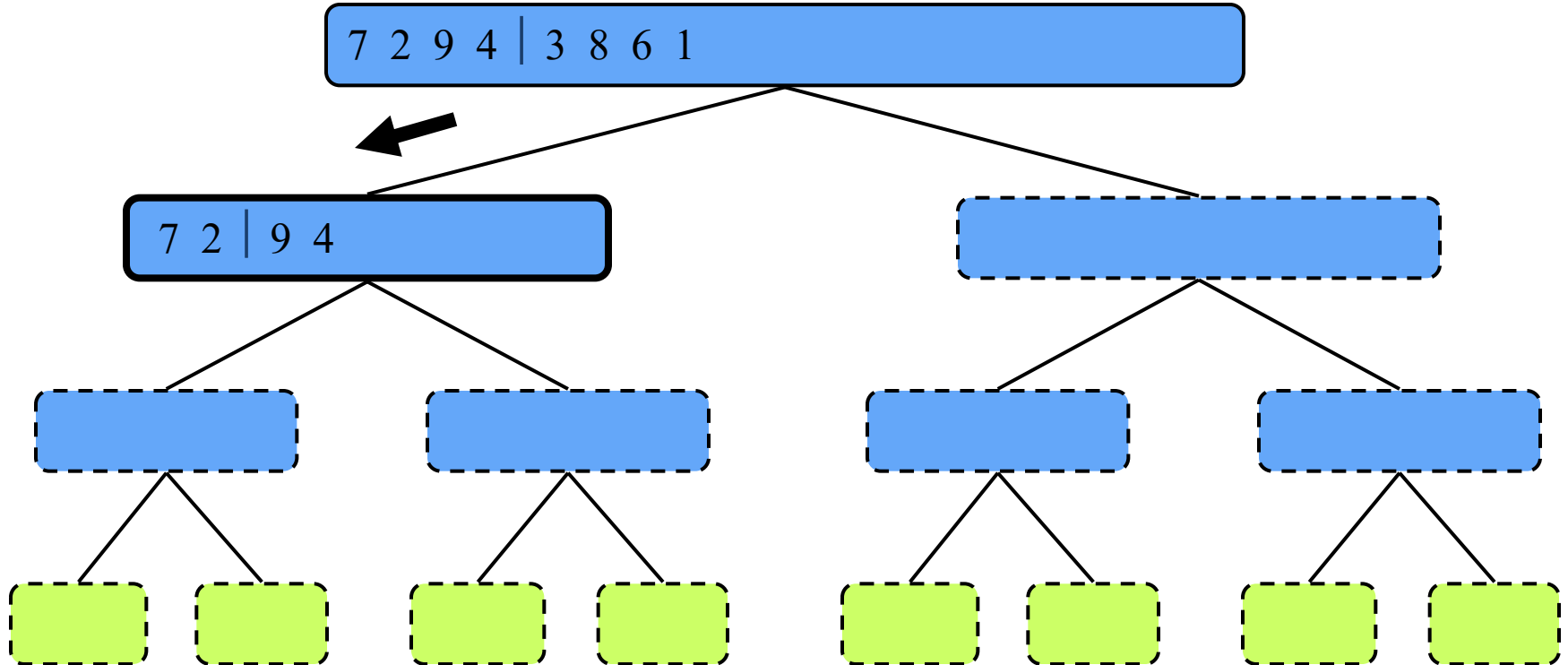
Execution Example

- Partition



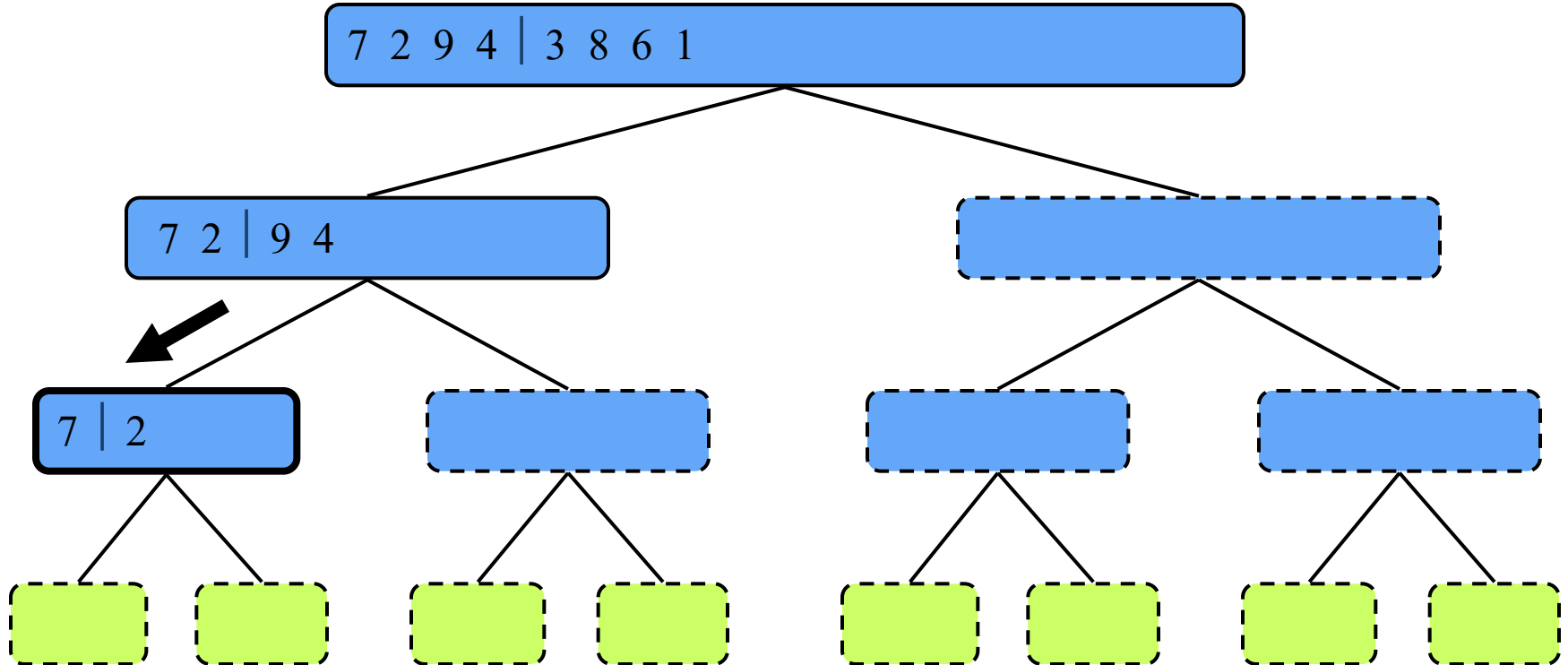
Execution Example (cont.)

- Recursive call, partition



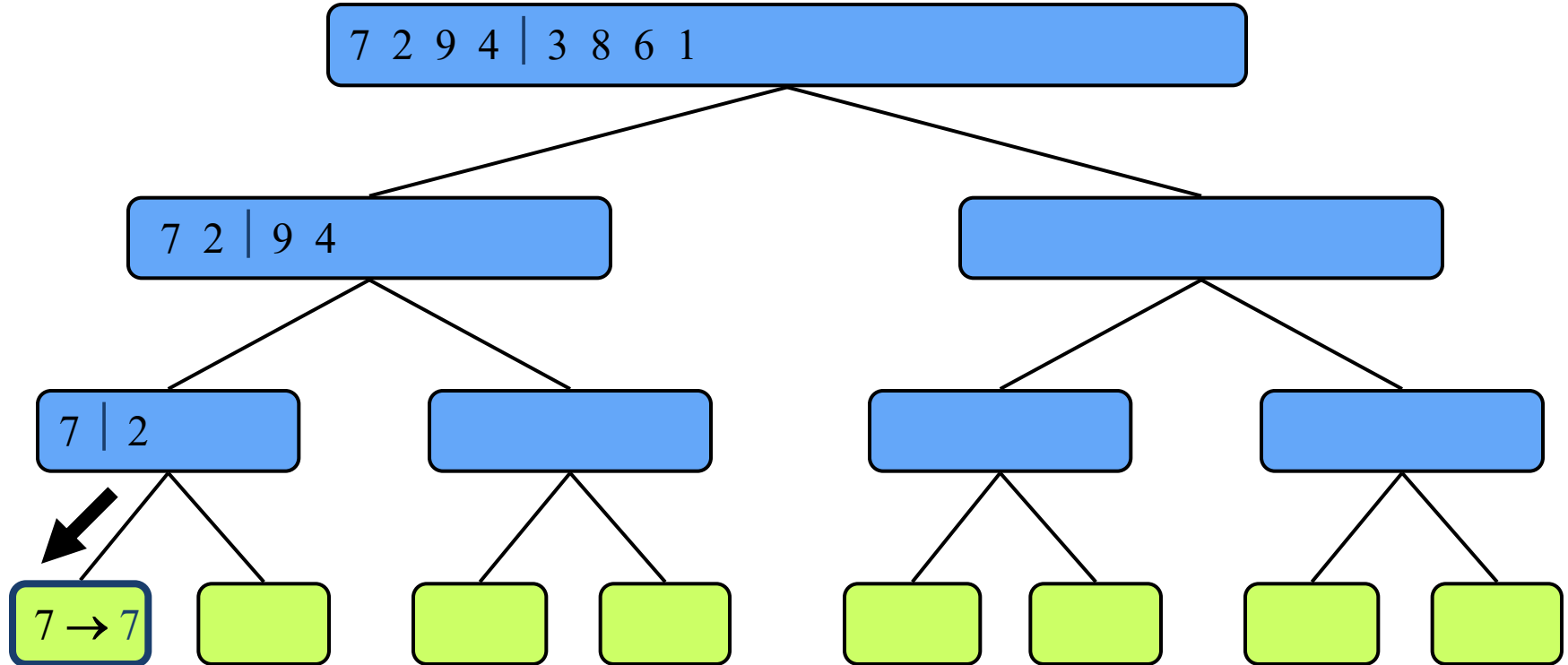
Execution Example (cont.)

- Recursive call, partition



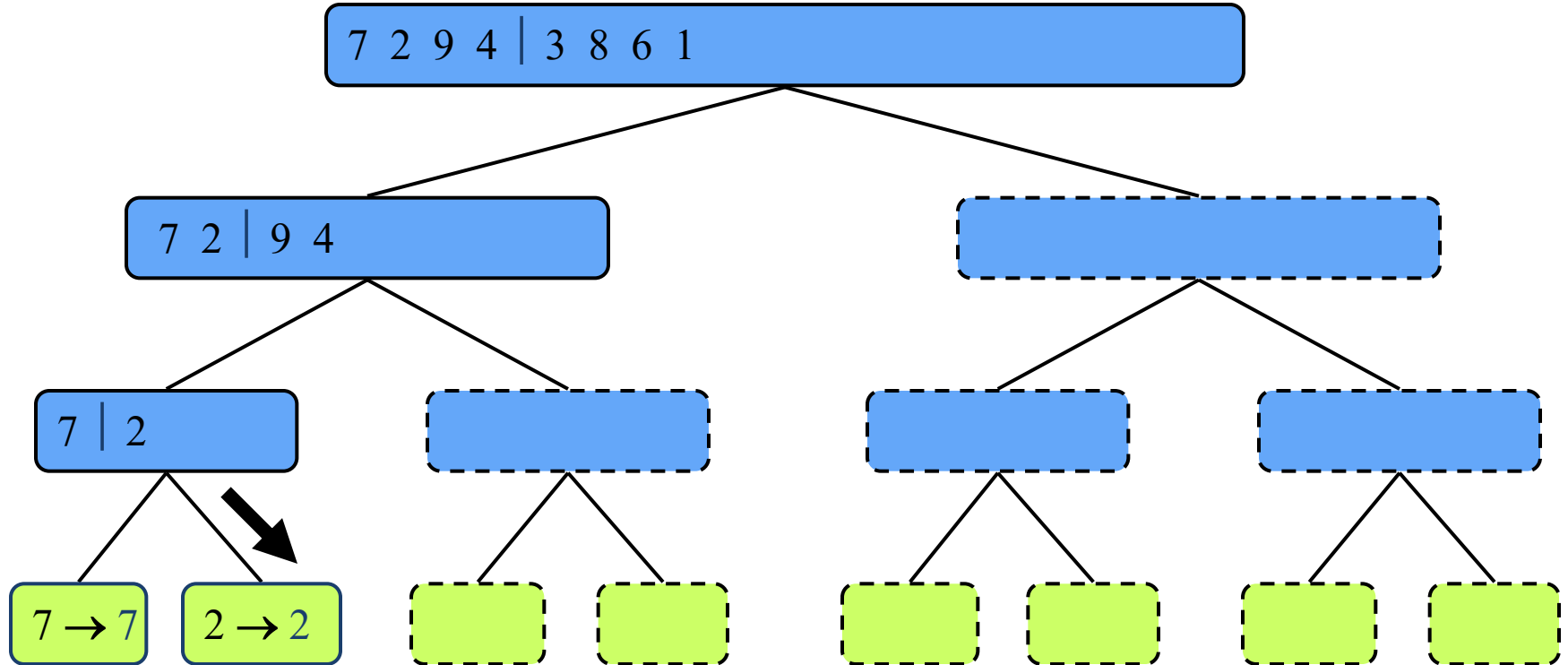
Execution Example (cont.)

- Recursive call, base case



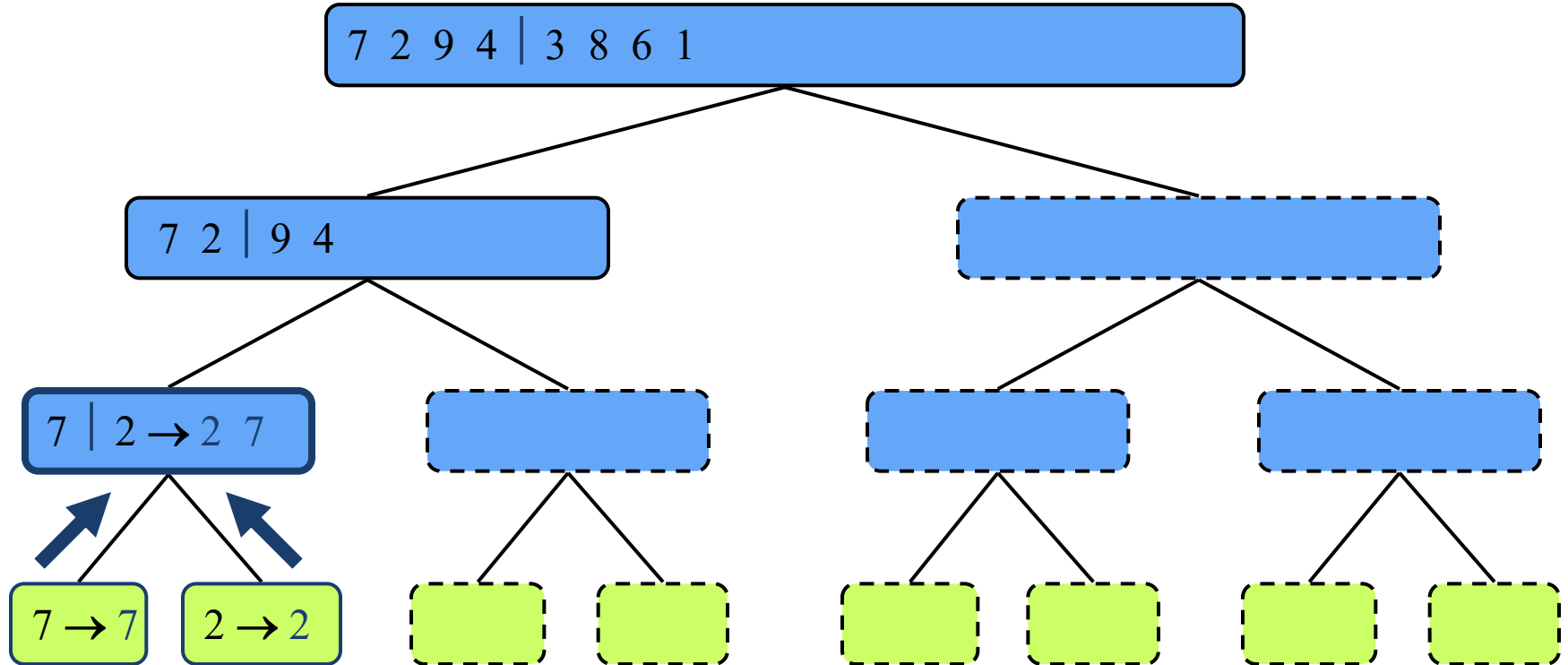
Execution Example (cont.)

- Recursive call, base case



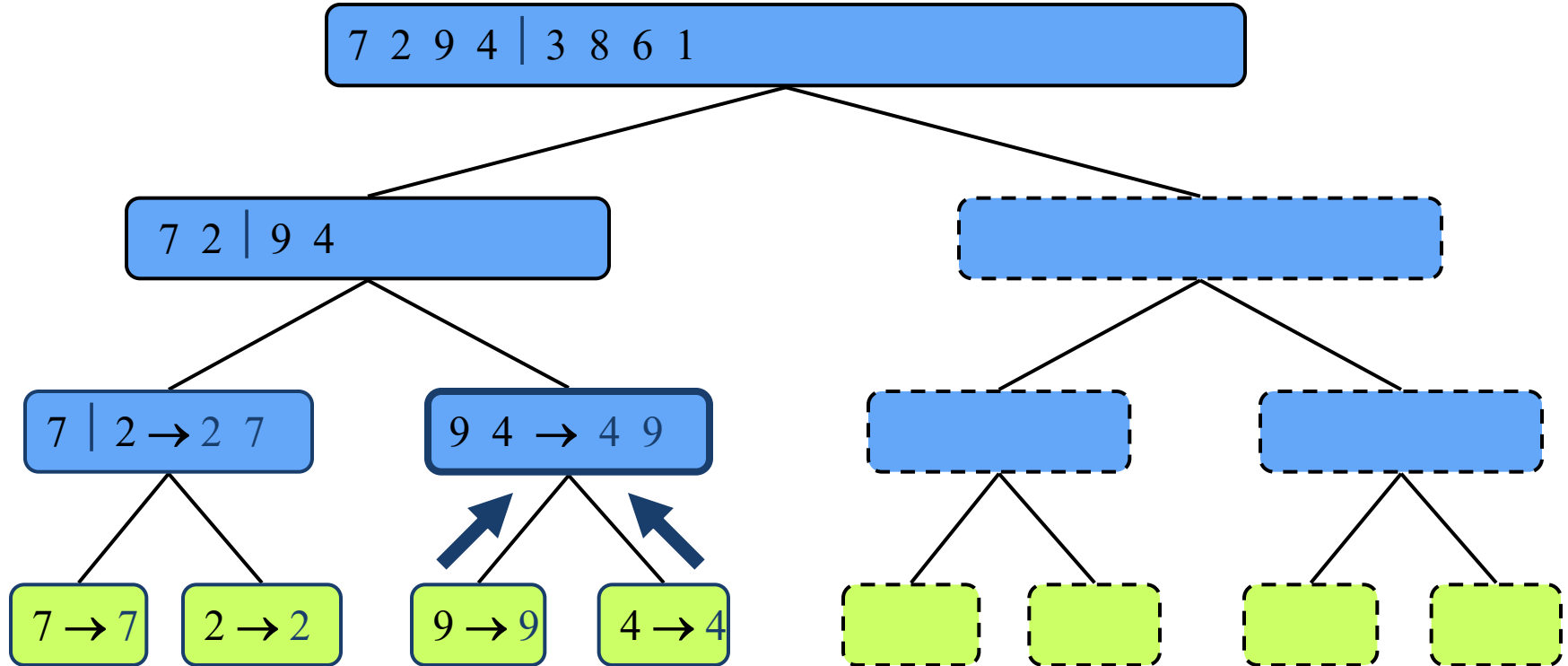
Execution Example (cont.)

- Merge



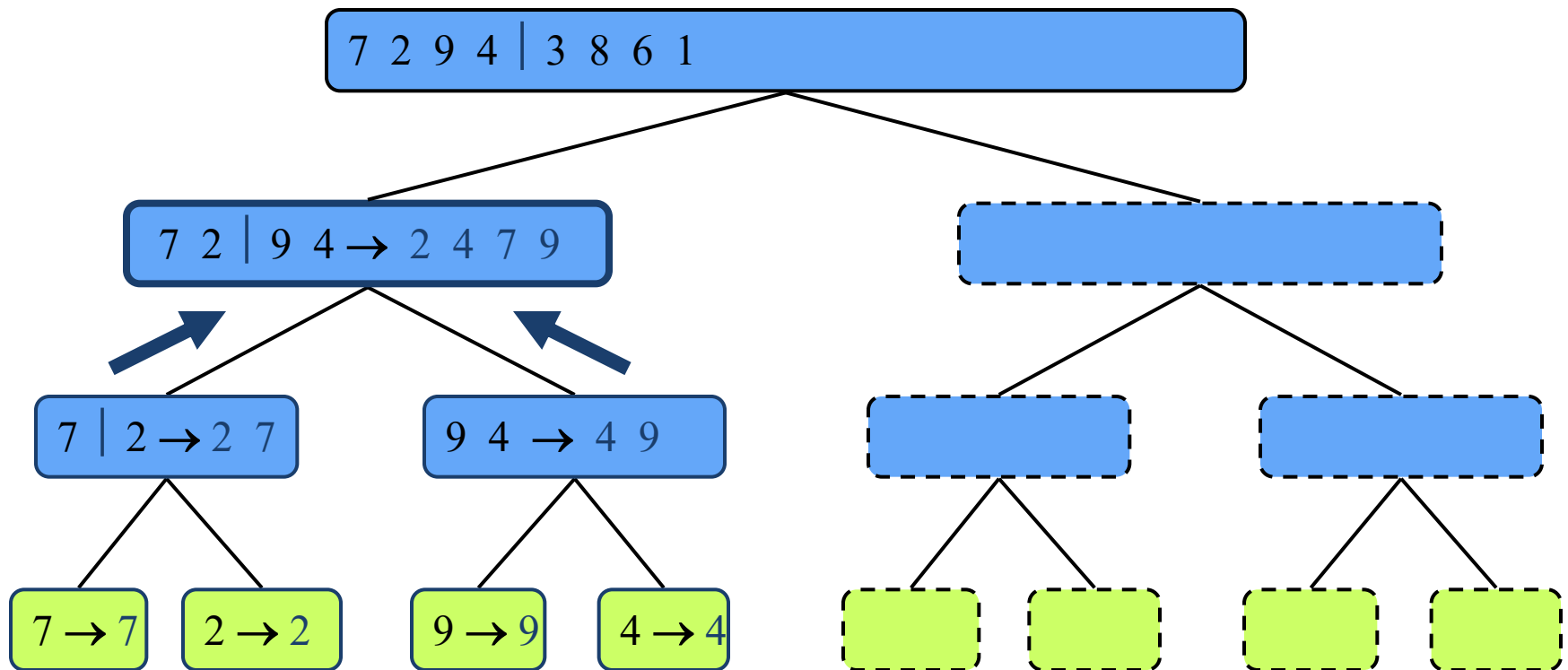
Execution Example (cont.)

- Recursive call, ..., base case, merge

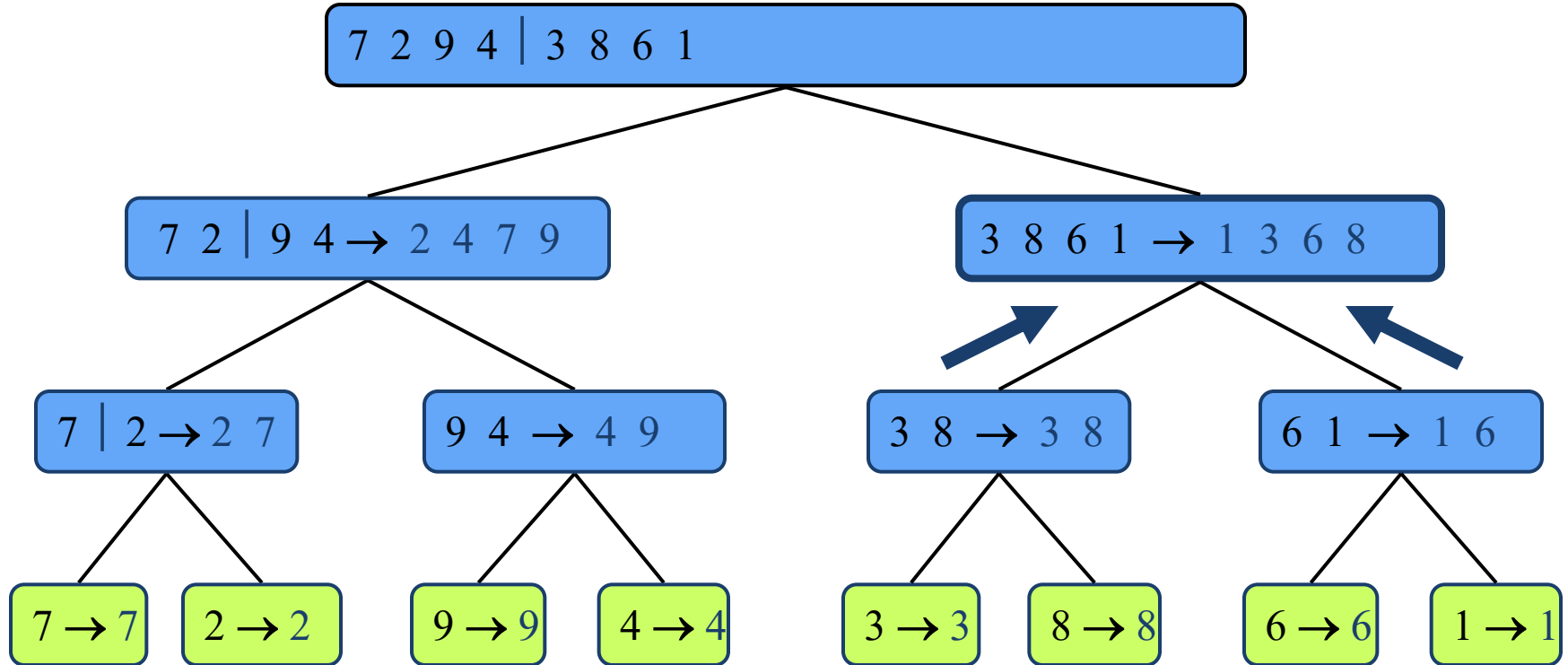


Execution Example (cont.)

- Merge

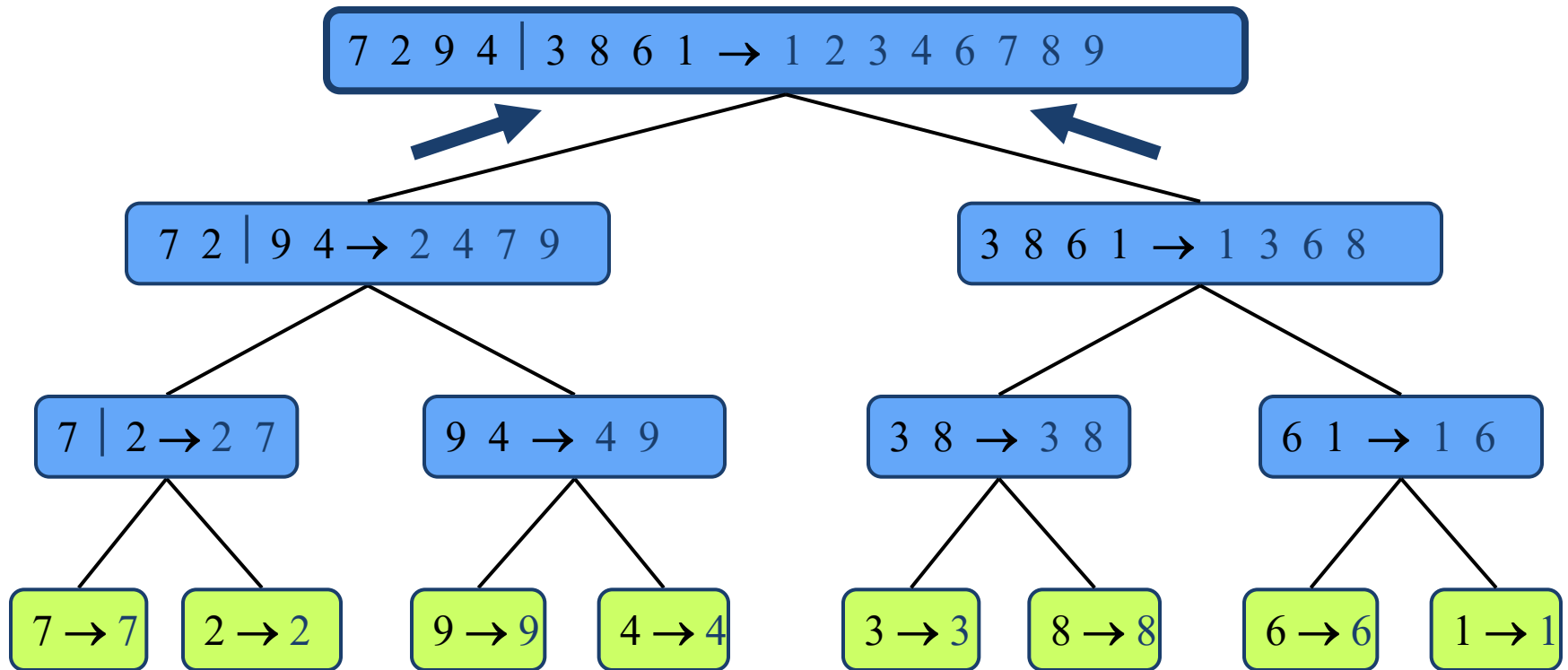


Execution Example (cont.)



Execution Example (cont.)

- Merge



Analysis of Merge-Sort

- The height h of the merge-sort tree is $O(\log n)$
 - at each recursive call we divide the sequence in half
- The overall amount of work done at the nodes of depth i is $O(n)$
 - we partition and merge 2^i sequences of size $n/2^i$
 - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

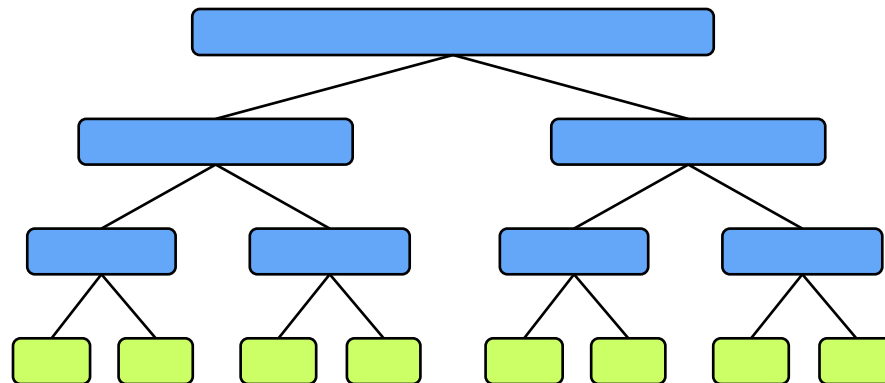
depth #seqs size

0 1 n

1 2 $n/2$

i 2^i $n/2^i$

...



Comparing sorting algorithms

Consider the following when evaluating a sorting algorithm:

- Time complexity
- Space complexity
 - An **in-place** algorithm requires only $n + O(1)$ space, using the already given space for the n elements and $O(1)$ additional space
- Stability
 - A sorting algorithm is **stable** if it preserves the original relative ordering of elements with equal value
 - Ex: Unsorted sequence (**B**, **b**, a, c). Suppose $B = b$ and $a < b < c$.
 - Stable sorted: (a, **B**, **b**, c)
 - Unstable sorted: (a, **b**, **B**, c)
 - Necessary if we want to sort repeatedly by different attributes (i.e., sort by first name, then sort again by last name)

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	<ul style="list-style-type: none">◆ in-place◆ not stable◆ for small data sets (< 1K)
insertion-sort	$O(n^2)$	<ul style="list-style-type: none">◆ in-place◆ stable◆ for small data sets (< 1K)
heap-sort	$O(n \log n)$	<ul style="list-style-type: none">◆ in-place◆ not stable◆ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	<ul style="list-style-type: none">◆ not in-place◆ stable◆ sequential data access◆ for huge data sets (> 1M)

Other

- You are given a query point p and a set S of n other points in two dimensional space. Find k points out of the n points which are nearest to p .

