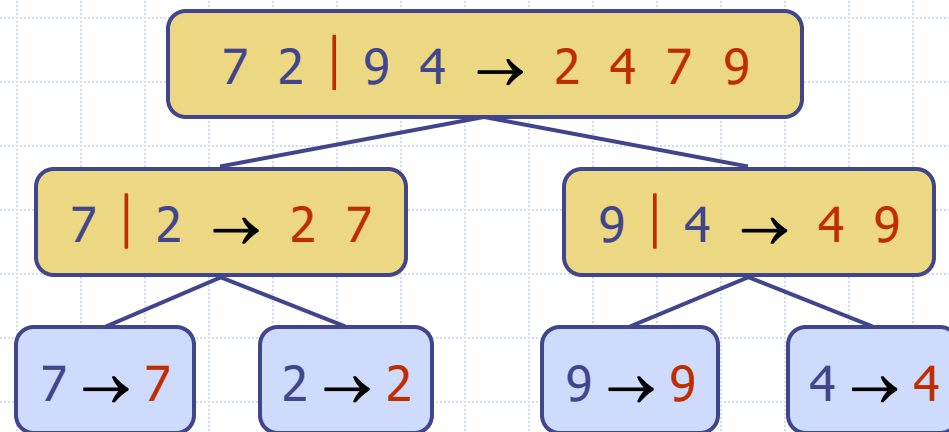


Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Merge Sort



Aim and Learning Objectives

- ◆ To be able to *understand* and *describe* the merge-sort algorithm
- ◆ To be able to *analyze* the complexity of the merge-sort algorithm
- ◆ To be able to *implement* the merge-sort algorithm and *apply* it to solve problems

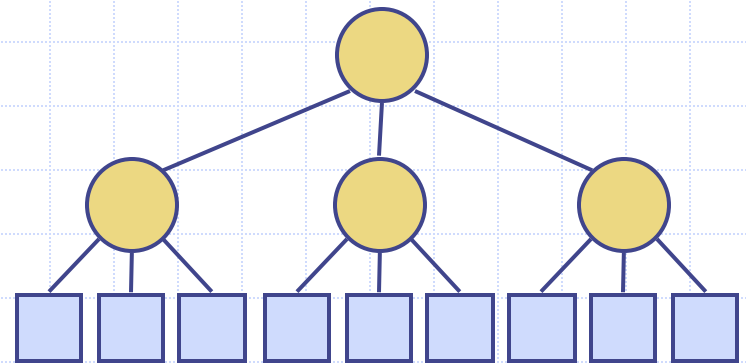
Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser,
Data Structures and Algorithms in Java, 6th Edition,
2014.

- Chapter 13. Sorting and Selection

Divide-and-Conquer

- ◆ **Divide-and conquer** is a general algorithm design paradigm:
 - **Divide**: divide the input data S in two or more disjoint subsets S_1, S_2, \dots
 - **Conquer**: solve the subproblems recursively
 - **Combine**: combine the solutions for S_1, S_2, \dots , into a solution for S
- ◆ The base case for the recursion are subproblems of constant size



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)

Input sequence S with n elements

Output sequence S sorted according to C

if $S.size() > 1$

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

mergeSort(S_1)

mergeSort(S_2)

$S \leftarrow merge(S_1, S_2)$

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 and implemented by means of a doubly linked list, 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 $A.first().element() < B.first().element()$

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

else

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

while $\neg A.isEmpty()$

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

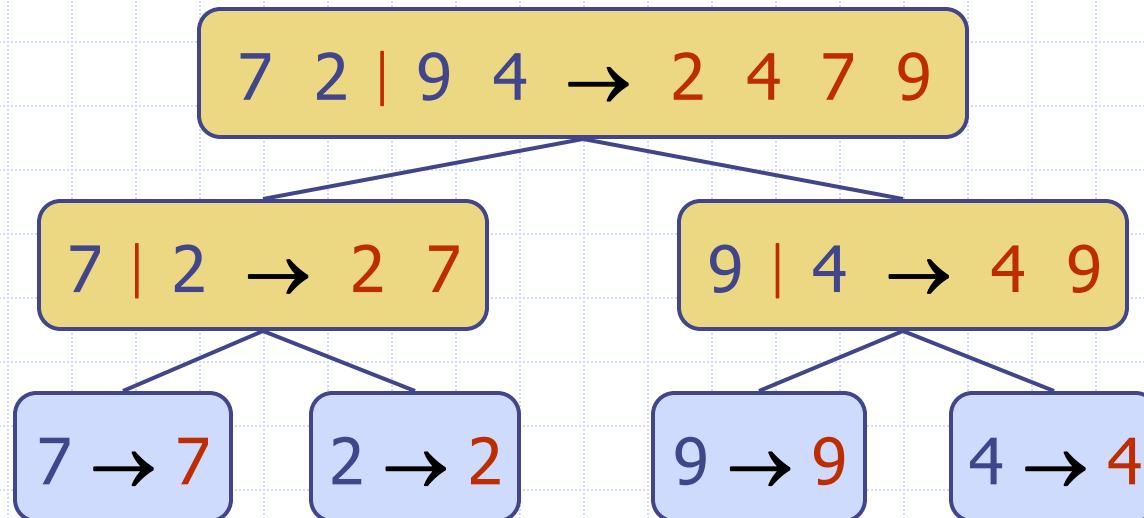
while $\neg B.isEmpty()$

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

return S

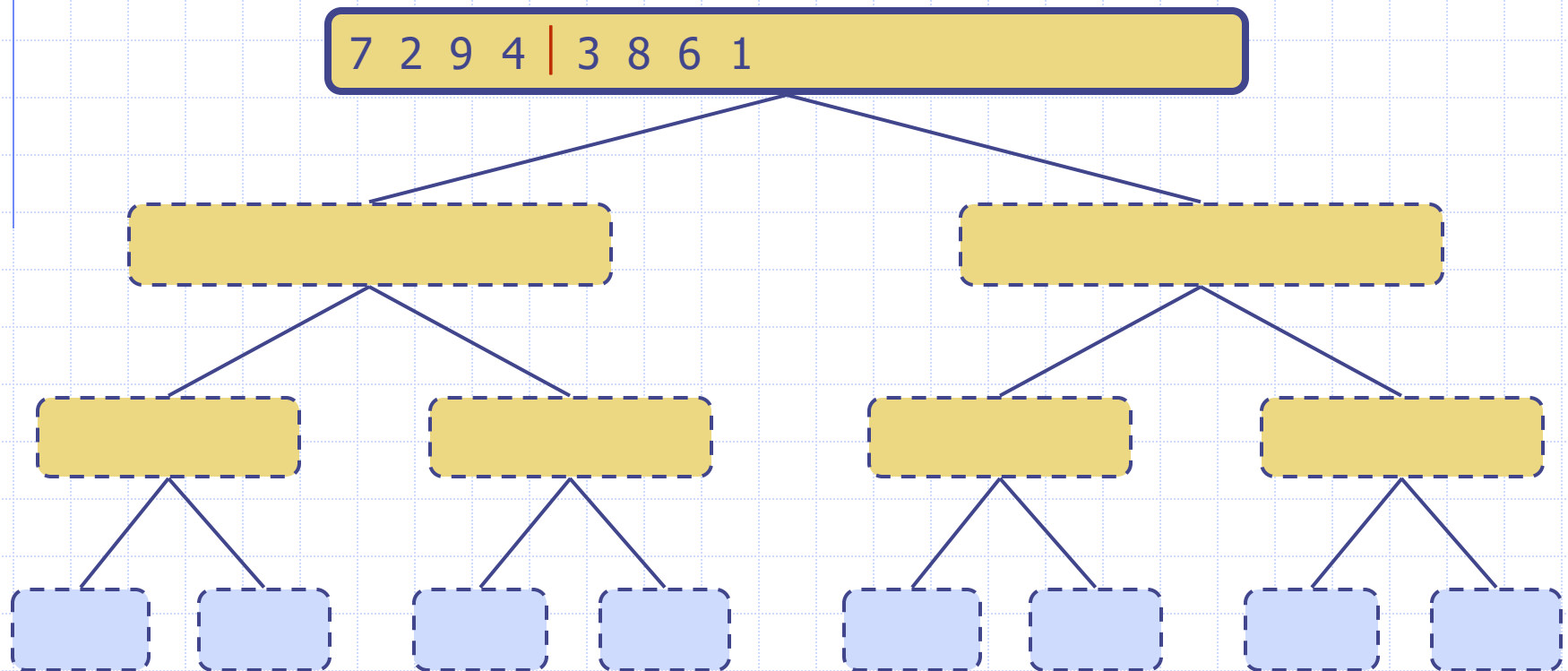
Merge-Sort Tree

- ◆ 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 0 or 1



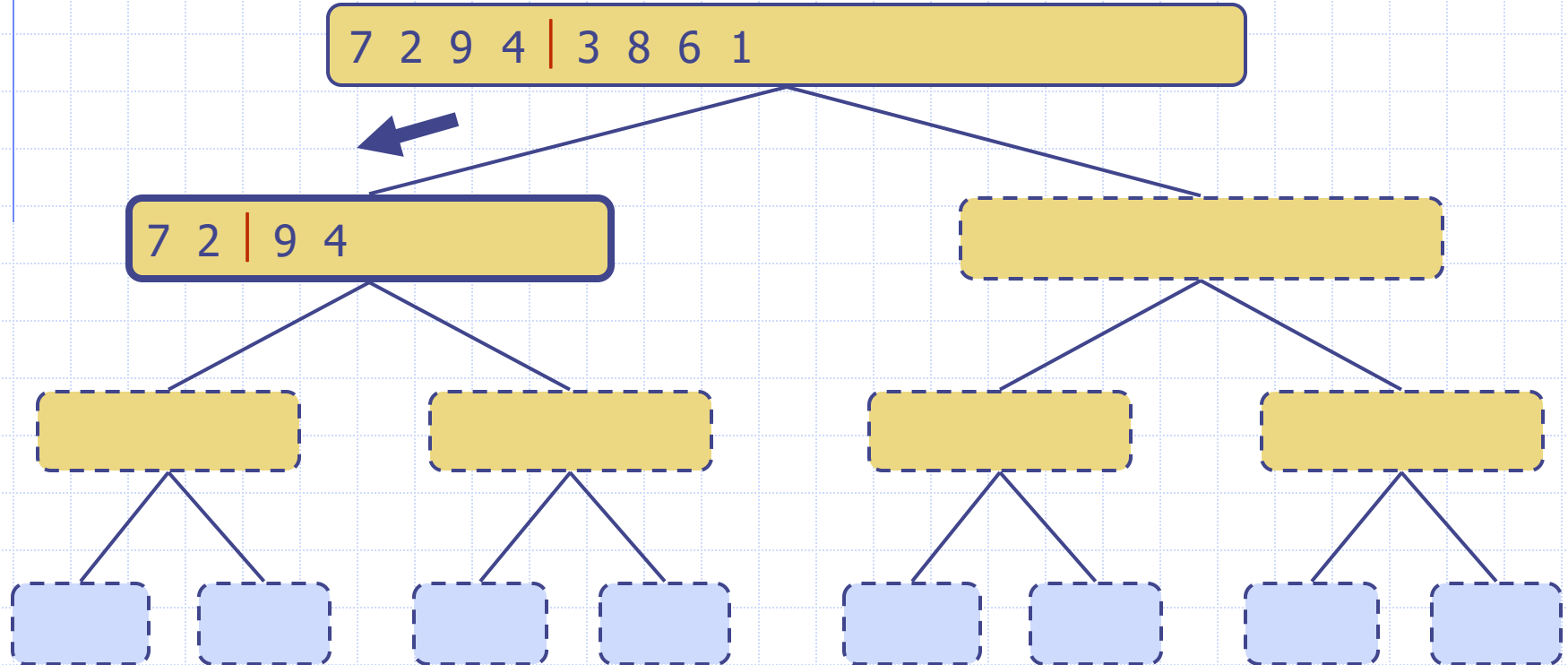
Execution Example

◆ Partition



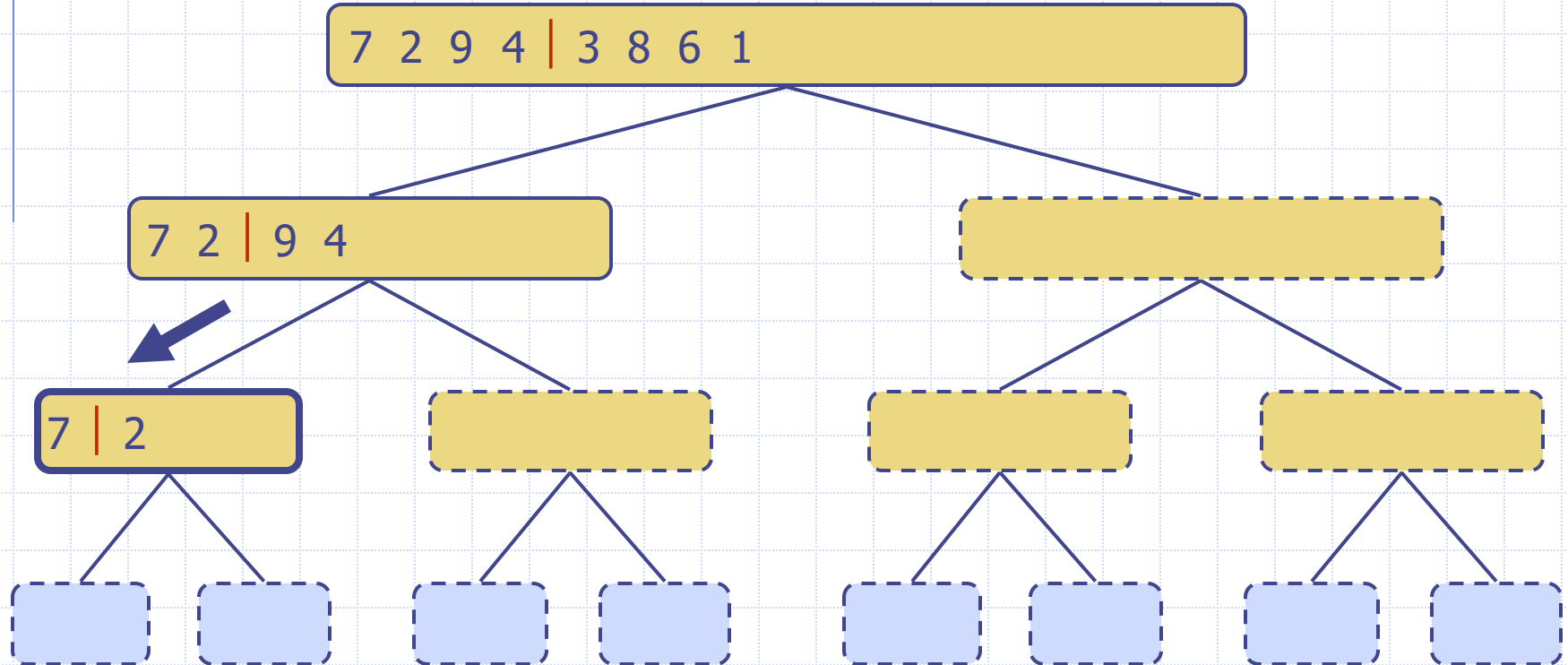
Execution Example (cont.)

◆ Recursive call, partition



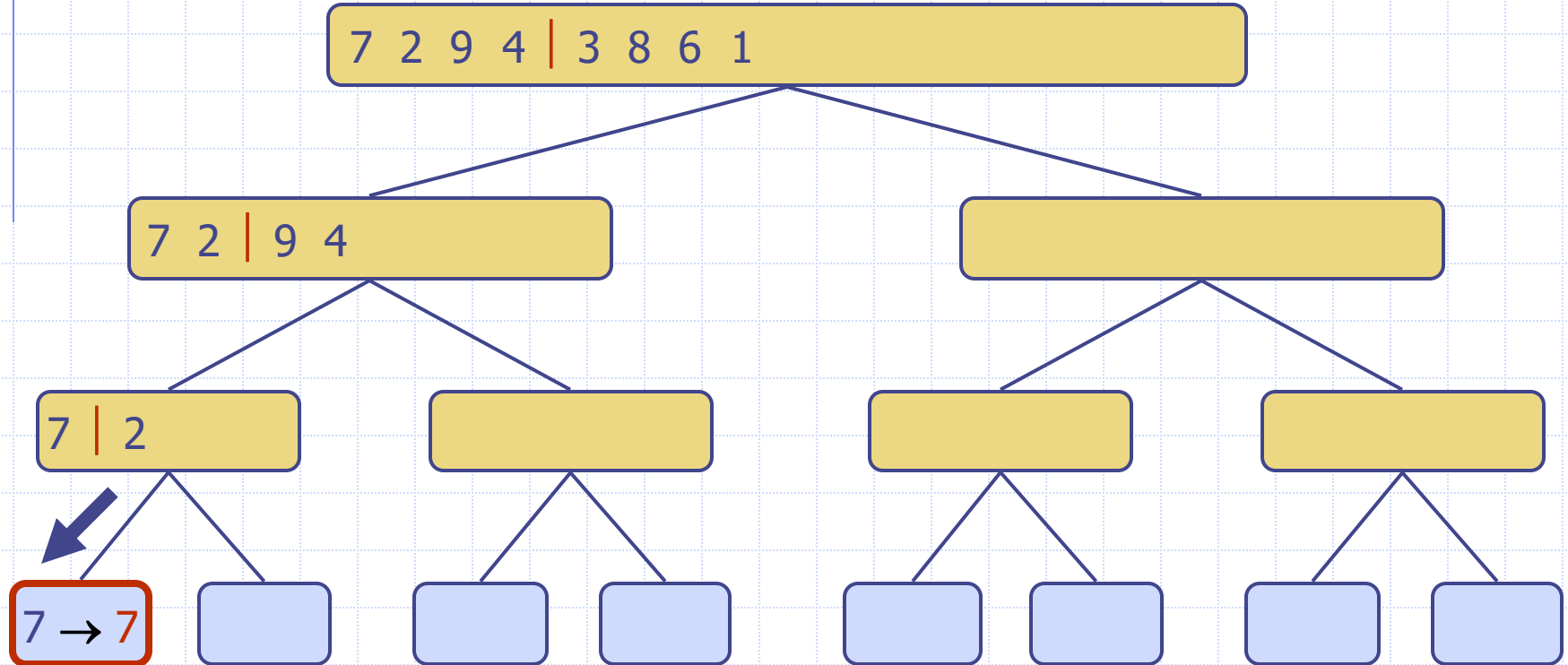
Execution Example (cont.)

◆ Recursive call, partition



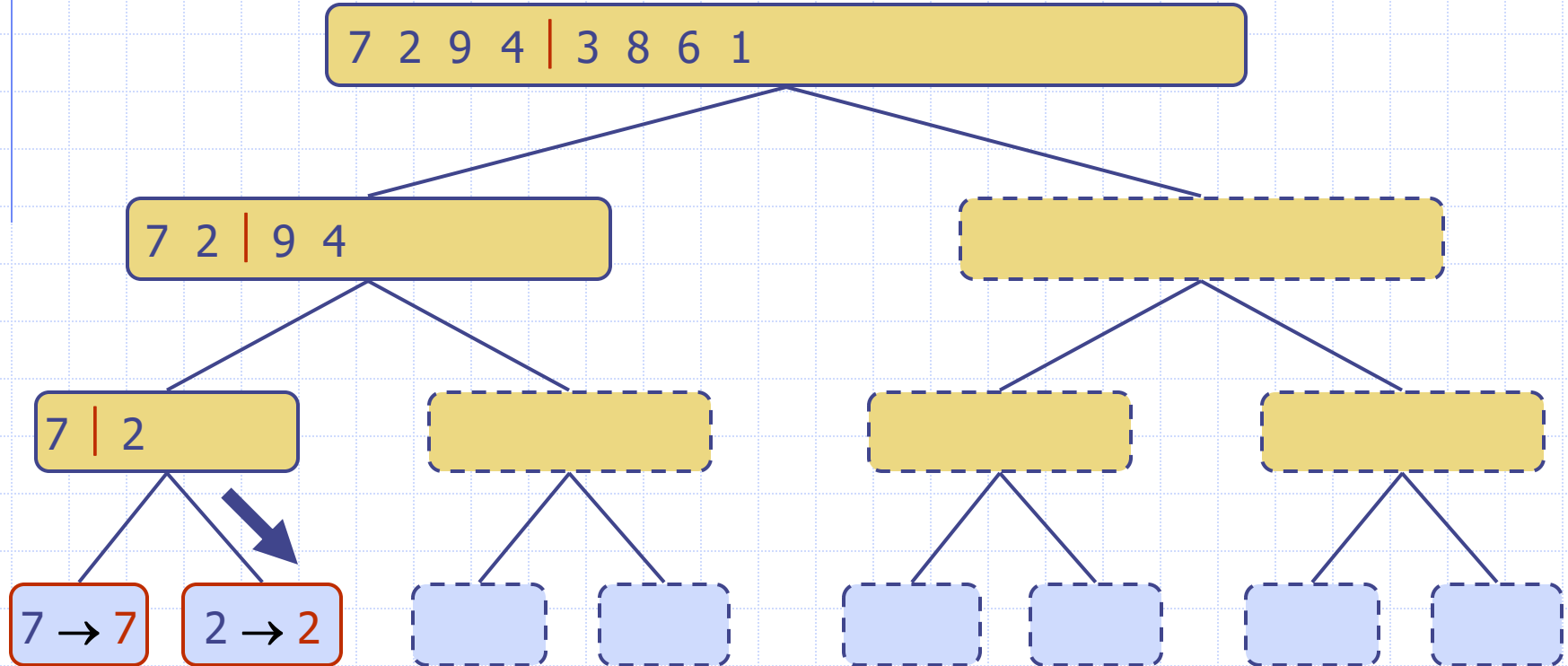
Execution Example (cont.)

◆ Recursive call, base case



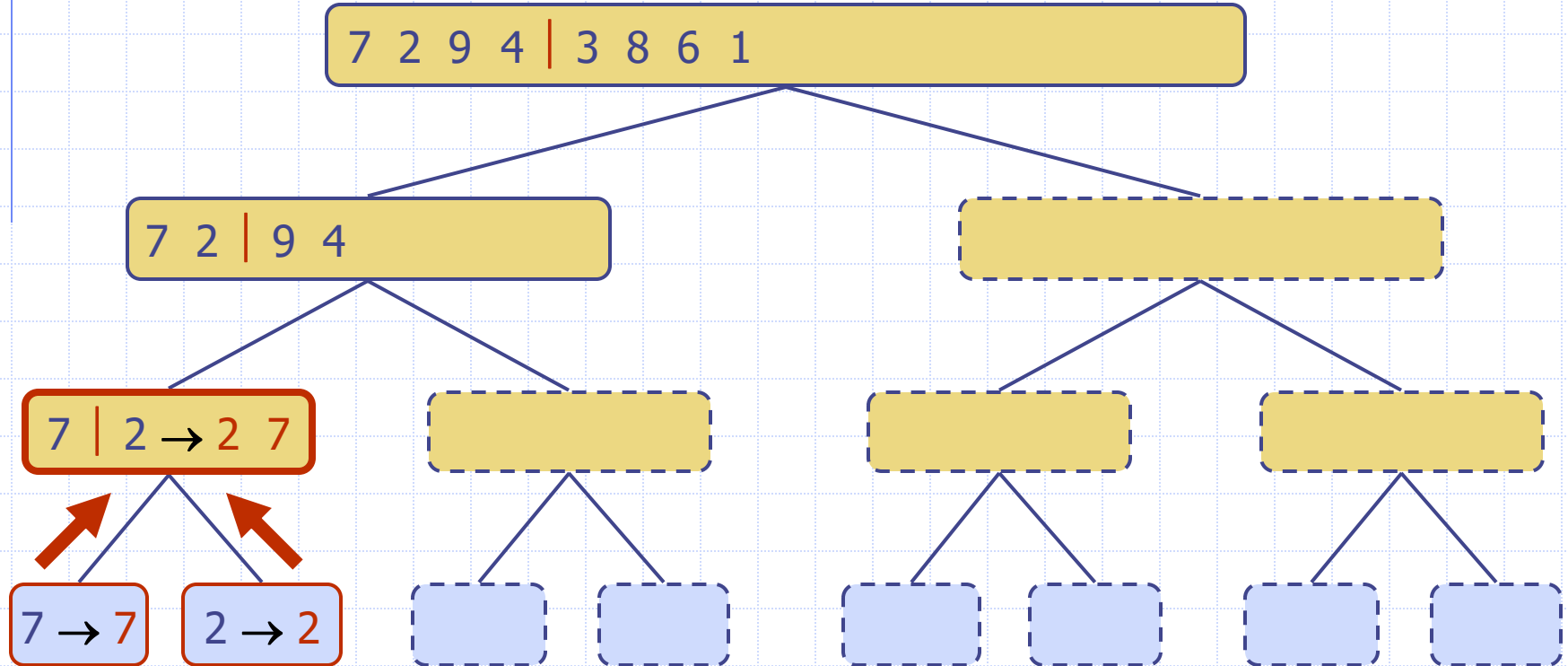
Execution Example (cont.)

◆ Recursive call, base case



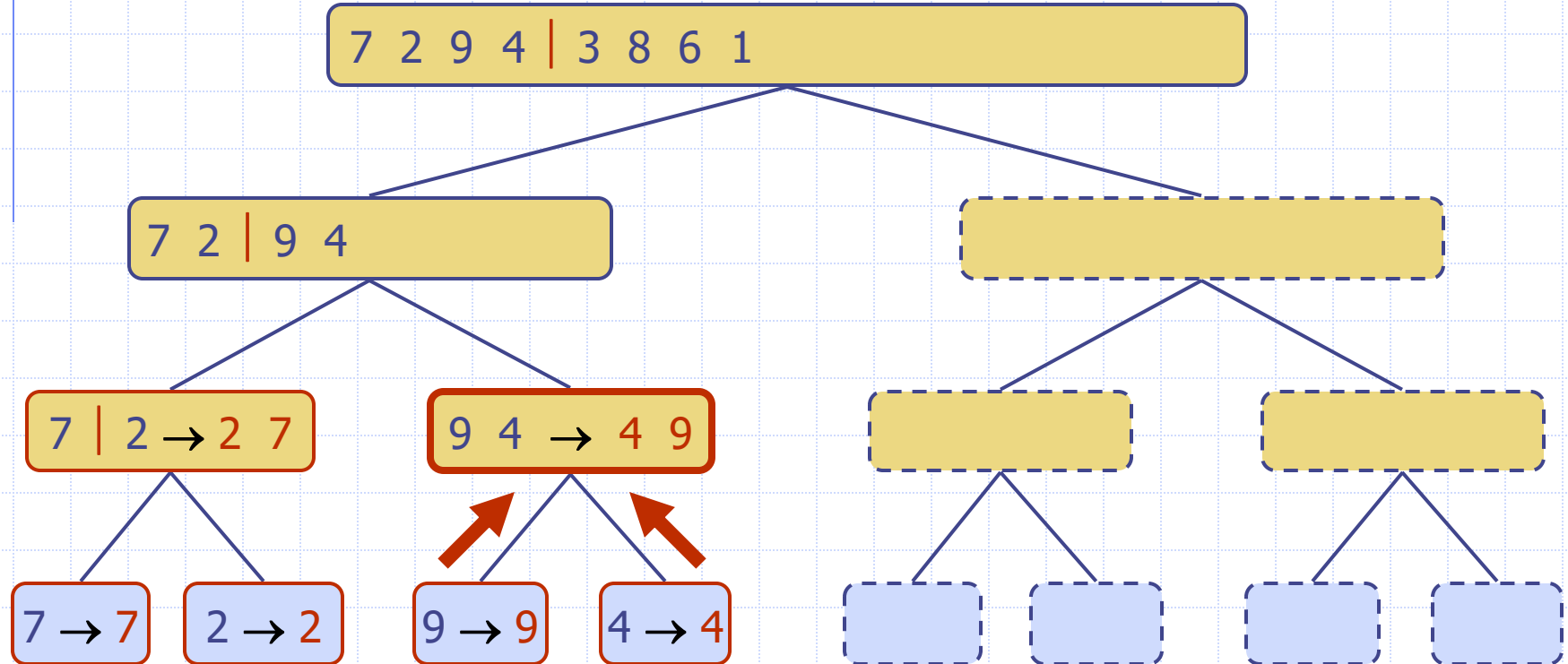
Execution Example (cont.)

◆ Merge



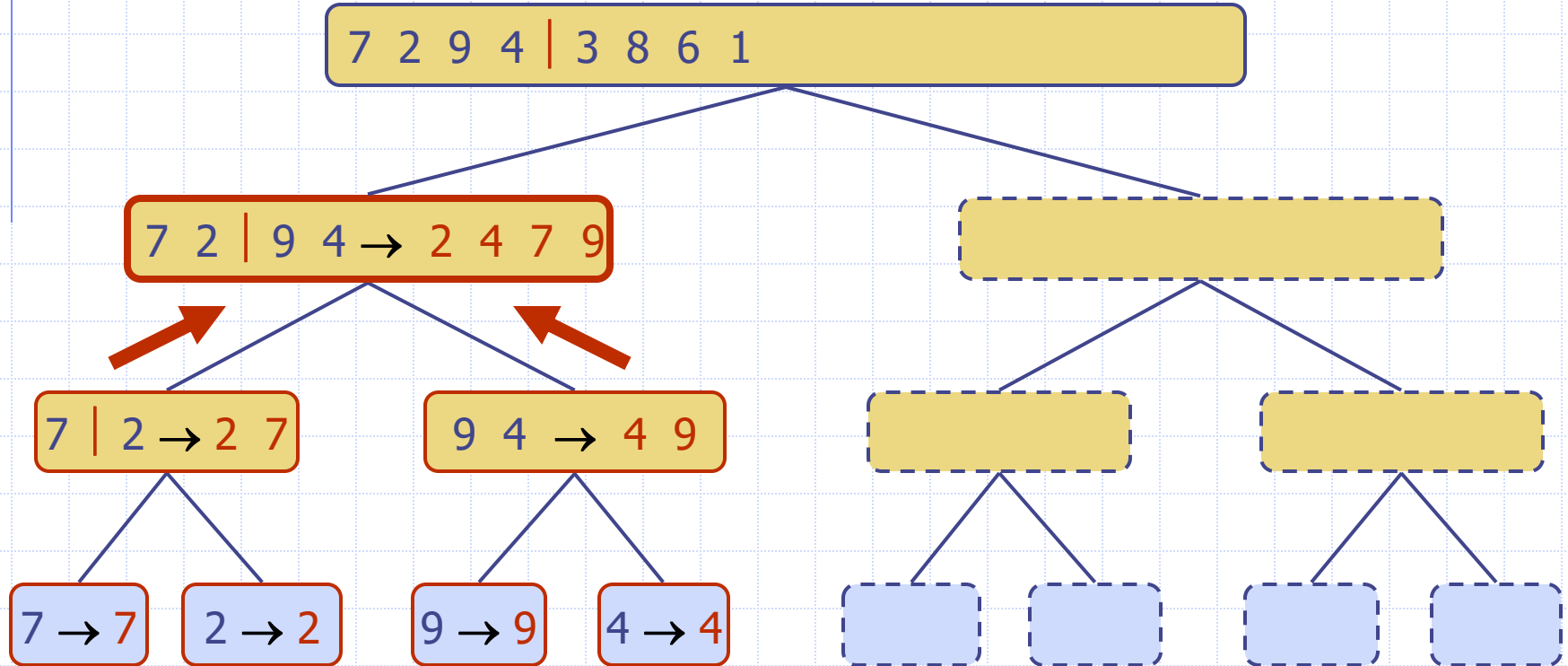
Execution Example (cont.)

◆ Recursive call, ..., base case, merge



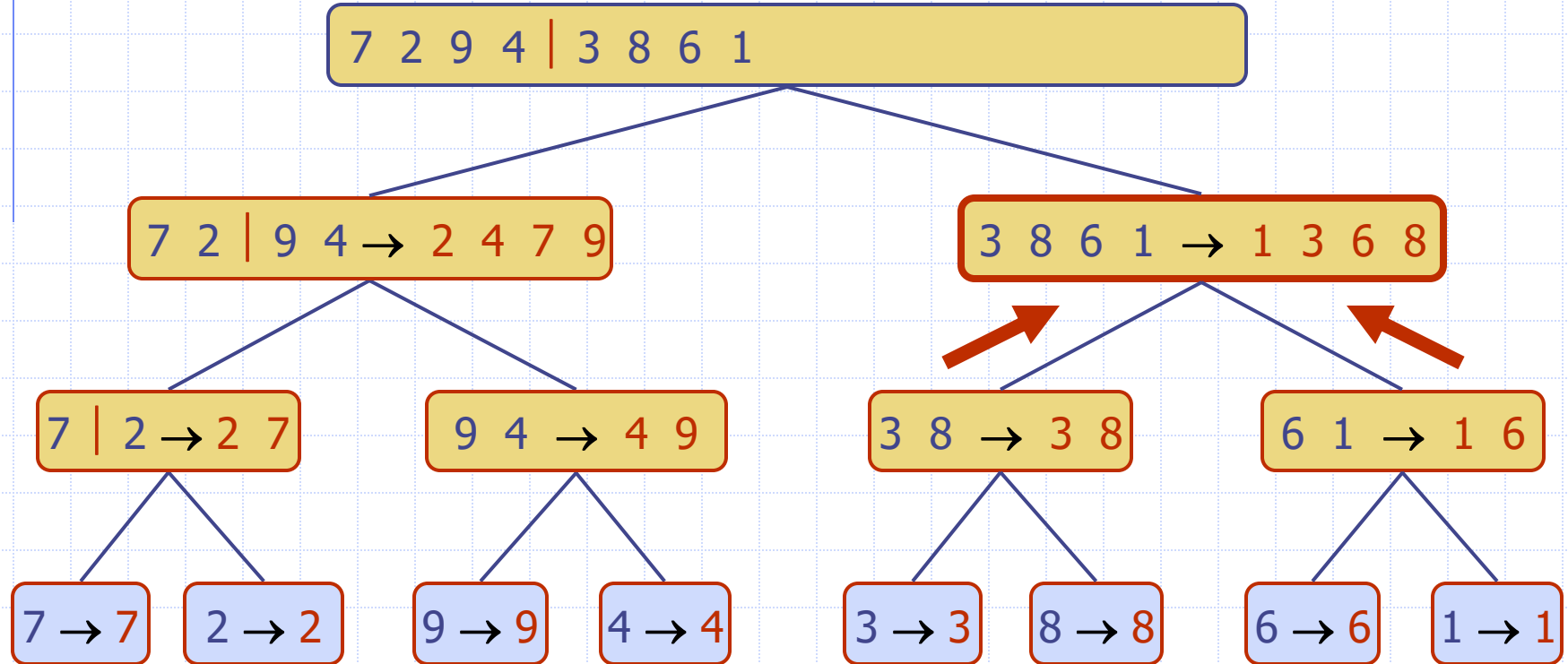
Execution Example (cont.)

◆ Merge



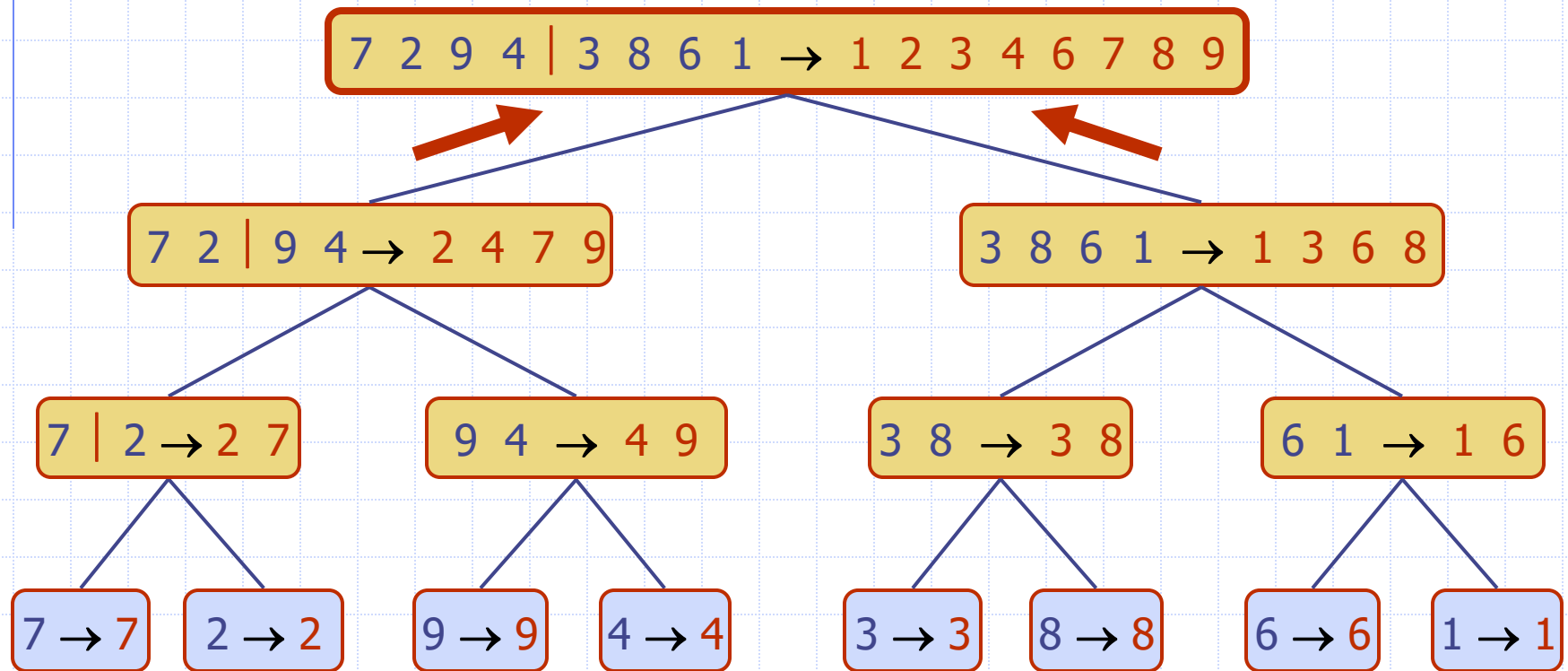
Execution Example (cont.)

◆ Recursive call, ..., merge, merge



Execution Example (cont.)

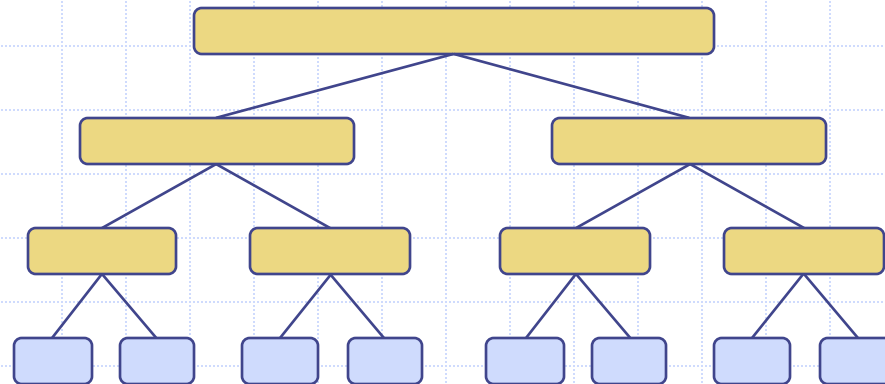
◆ Merge



Analysis of Merge-Sort

- ◆ The height h of the merge-sort tree is $O(\log n)$
 - at each recursive call we divide in half the sequence
- ◆ 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
 - Merge runs in $O(n)$
 - Partition 2^i partitions, $i \leq h \leq \log n$, $2^i \leq 2^h \leq 2^{\log n} = n$
- ◆ 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	$n/2^i$
...



Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	<ul style="list-style-type: none">▪ slow▪ in-place▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	<ul style="list-style-type: none">▪ slow▪ in-place▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	<ul style="list-style-type: none">▪ fast▪ in-place▪ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	<ul style="list-style-type: none">▪ fast▪ sequential data access▪ for huge data sets (> 1M)

Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser,
Data Structures and Algorithms in Java, 6th Edition,
2014.

- Chapter 13. Sorting and Selection