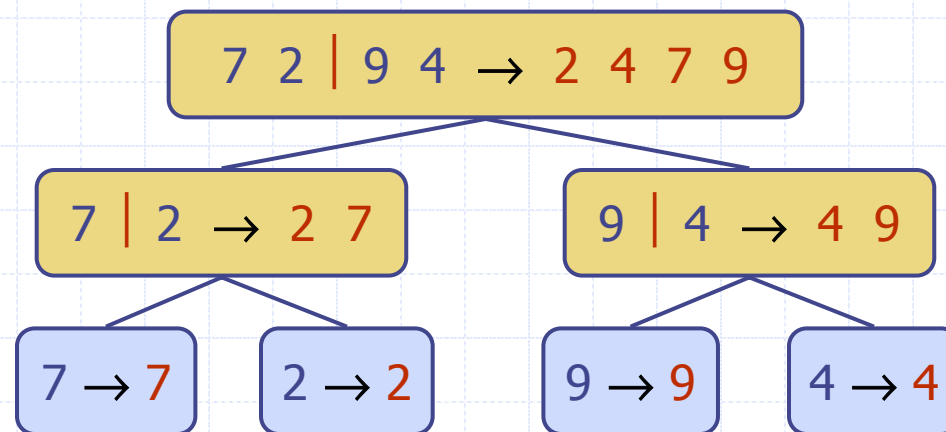


Merge Sort



Outline and Reading

- ◆ Divide-and-conquer paradigm (§11.1.1)
- ◆ Merge-sort (§11.1.2)
 - Algorithm
 - Merging two sorted sequences
 - Merge-sort tree
 - Execution example
 - Analysis
- ◆ Generic merging and set operations (§11.1.2)
- ◆ Summary of sorting algorithms (§11.5)

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
 - **Conquer**: combine the solutions for S_1 and S_2 into a solution for S
- ◆ The base case for the recursion are subproblems of size 0 or 1
- ◆ **Merge-sort** is a sorting algorithm based on the divide-and-conquer paradigm
- ◆ Like heap-sort
 - It uses a comparator
 - It has $O(n \log n)$ running time
- ◆ Unlike heap-sort
 - It does not use an auxiliary priority queue
 - It 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)

$S \leftarrow merge(S_1, S_2)^{**}$

*divides S into 2 (nearly) equally-sized subsets

**merge() algo on slide 5

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

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

else

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

while $\neg A.isEmpty()$

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

while $\neg B.isEmpty()$

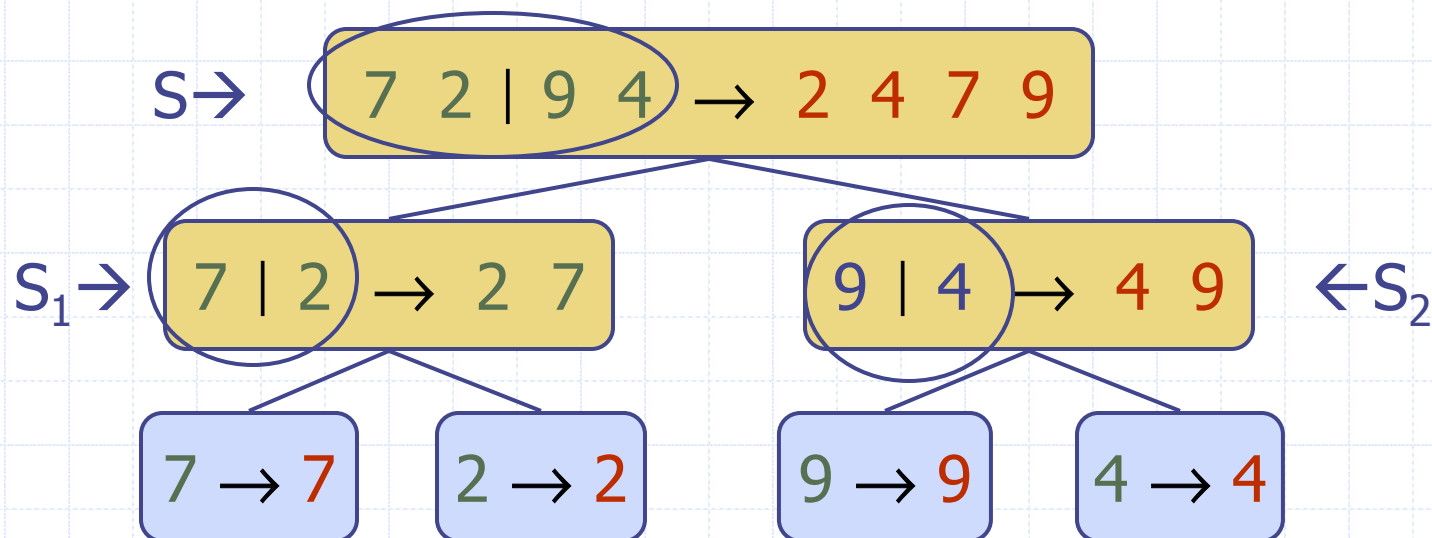
$S.insertLast(B.remove(B.first()))$

return S

**normally we don't use 1-letter variables*

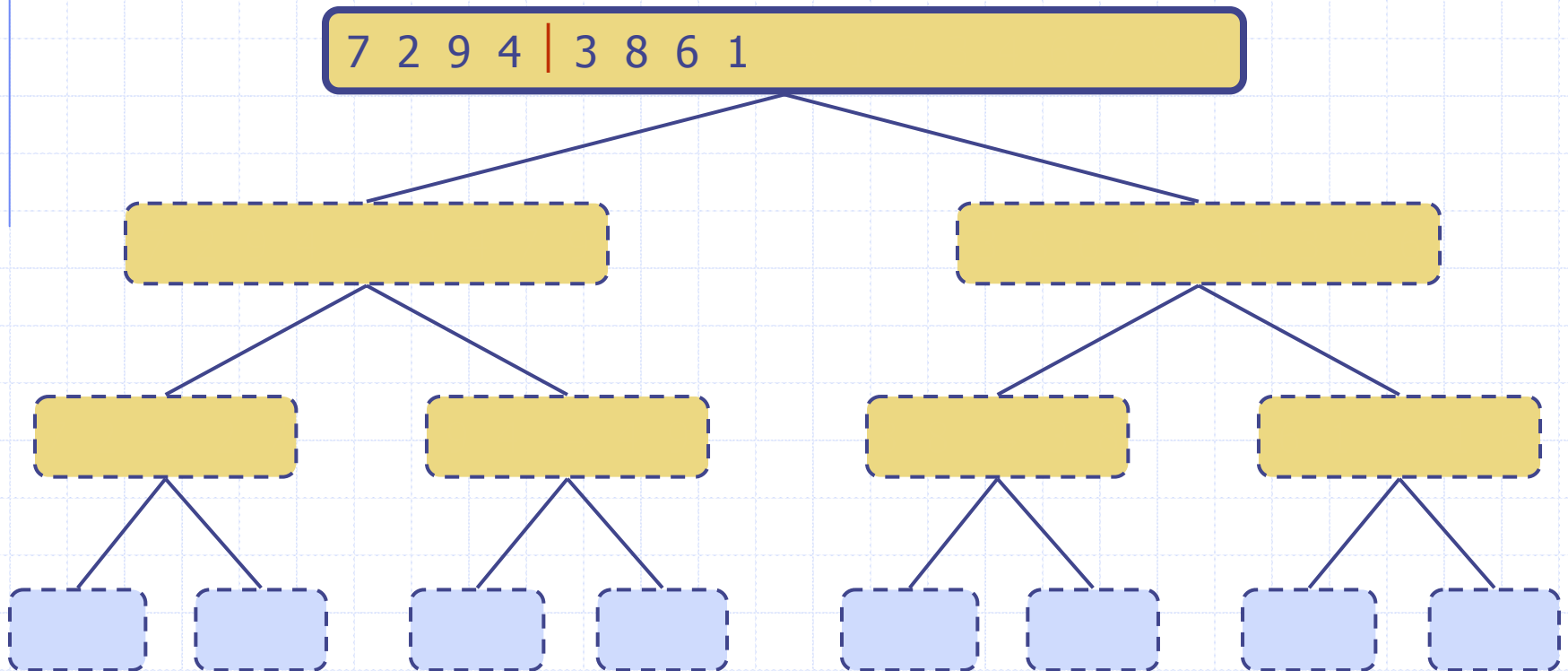
Merge-Sort Tree

- ◆ 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



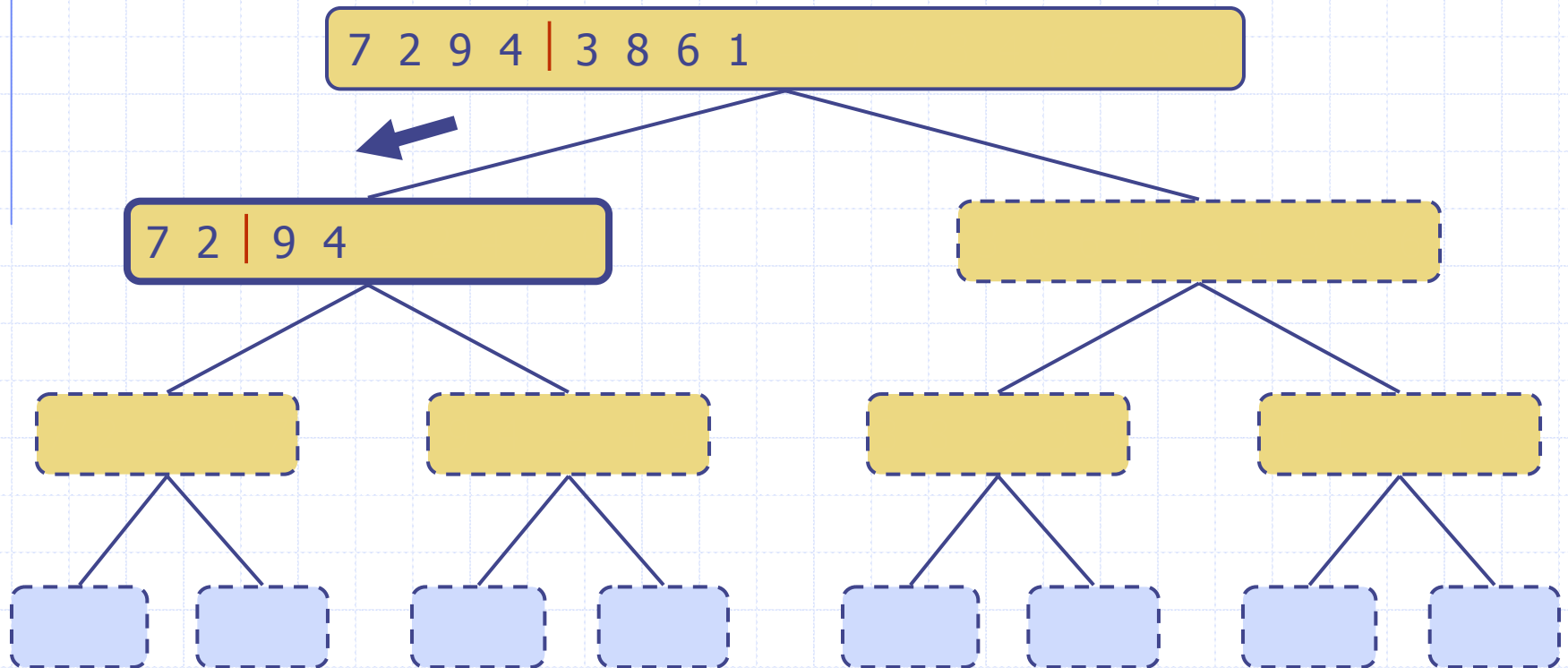
Execution Example

◆ Partition



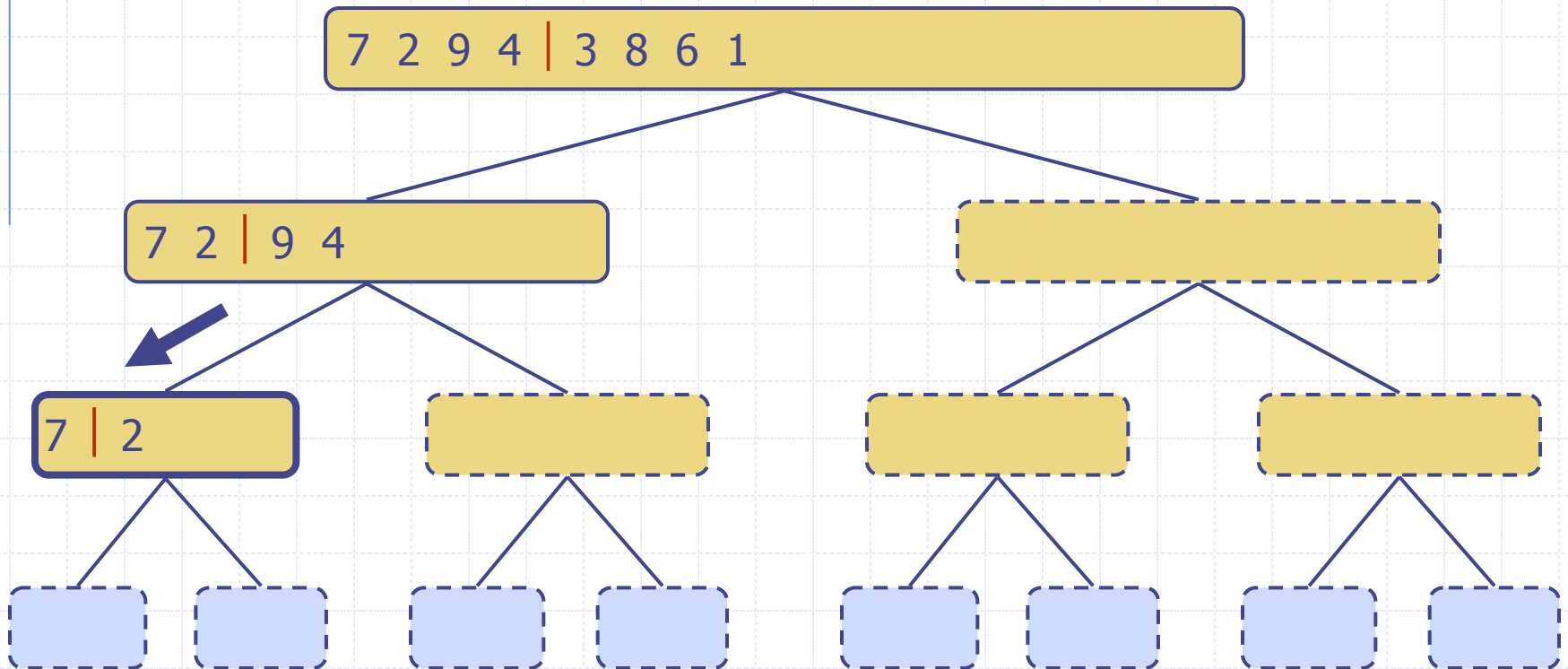
Execution Example (cont.)

◆ Recursive call, partition



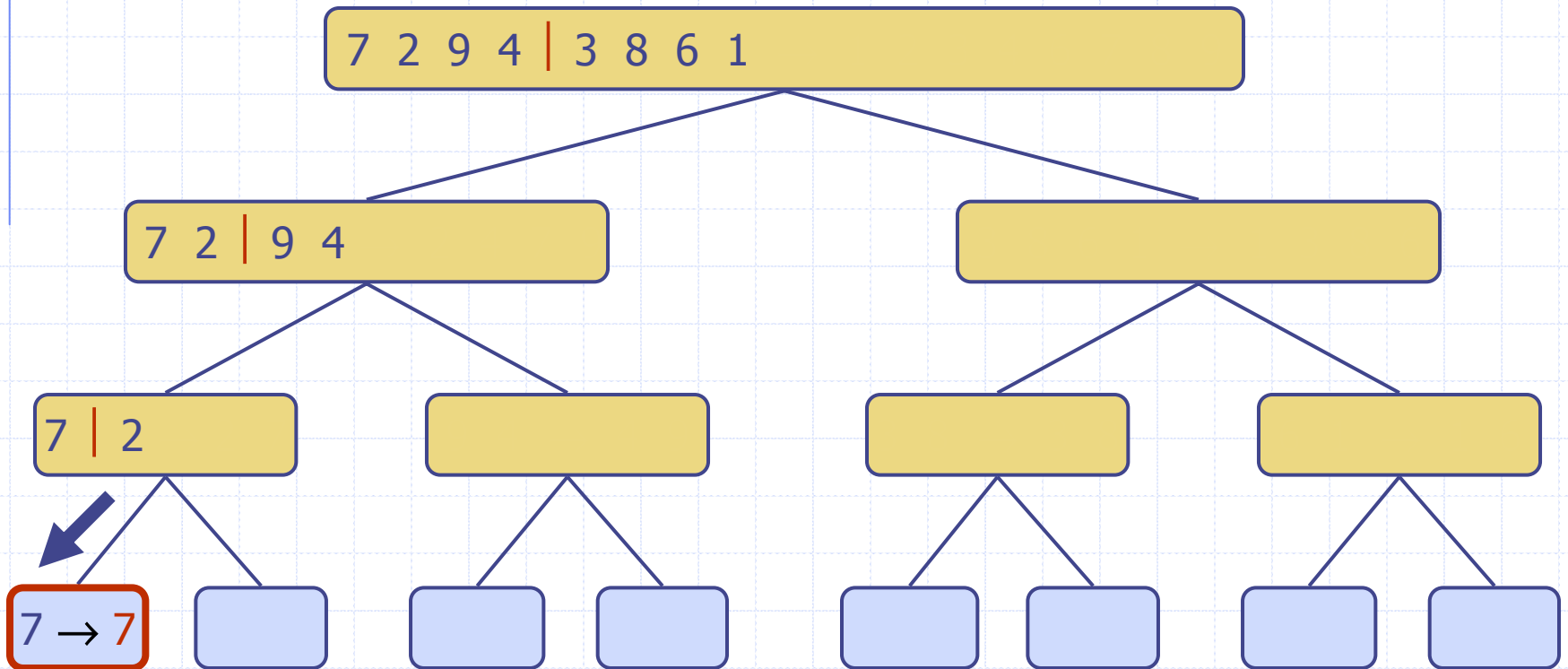
Execution Example (cont.)

◆ Recursive call, partition



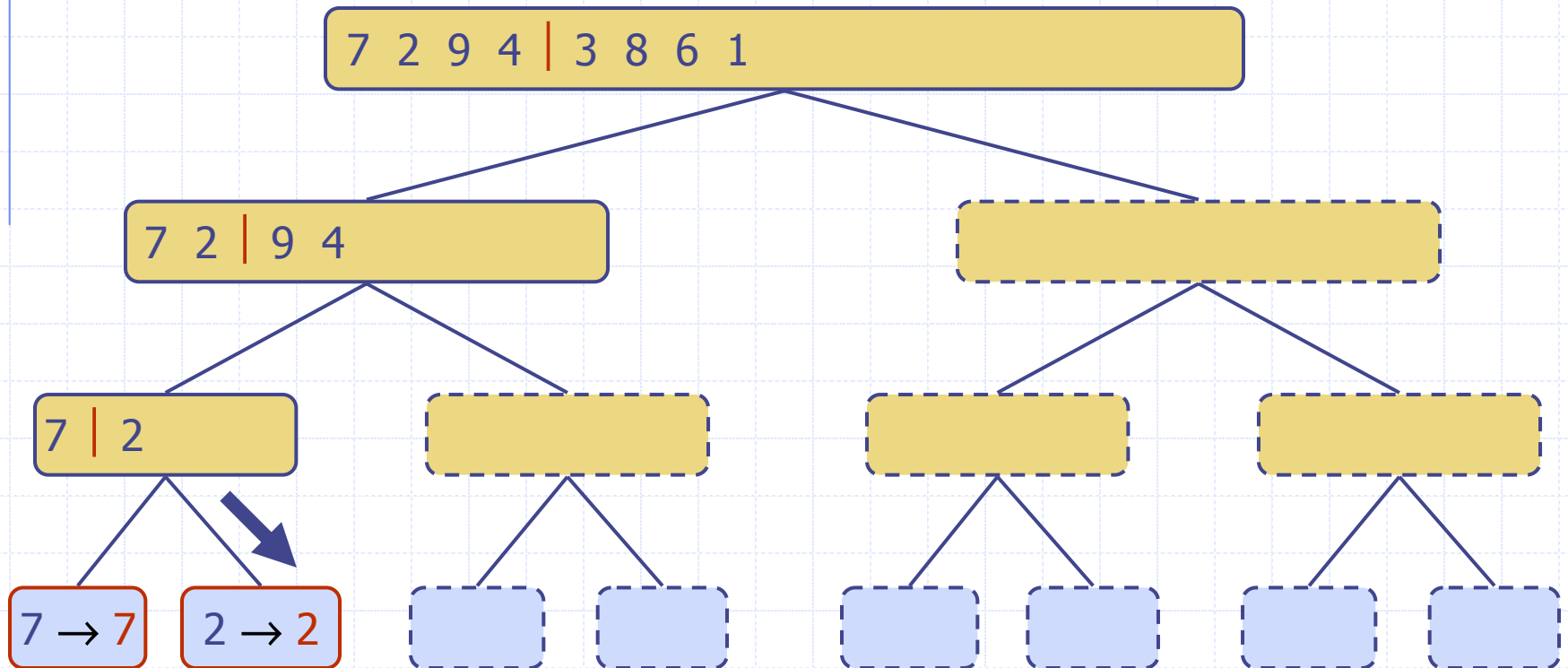
Execution Example (cont.)

◆ Recursive call, base case



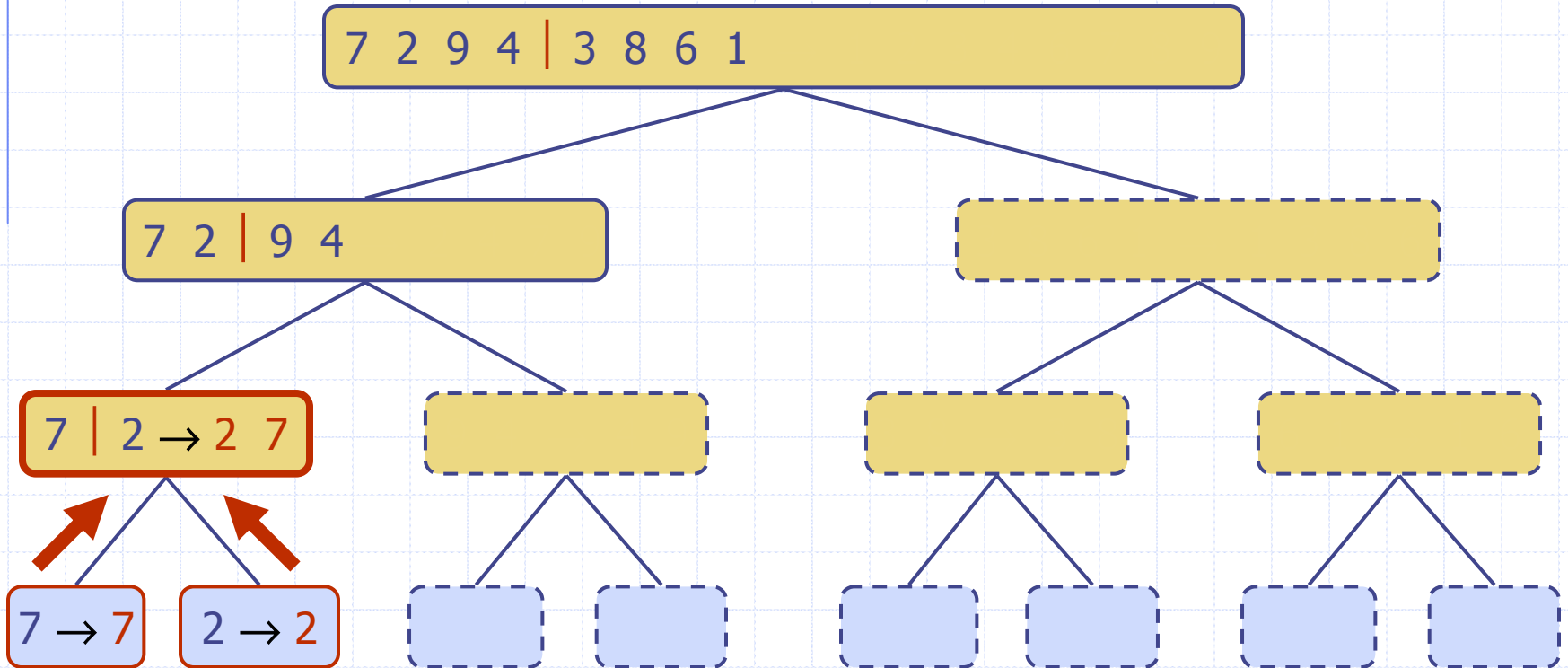
Execution Example (cont.)

◆ Recursive call, base case



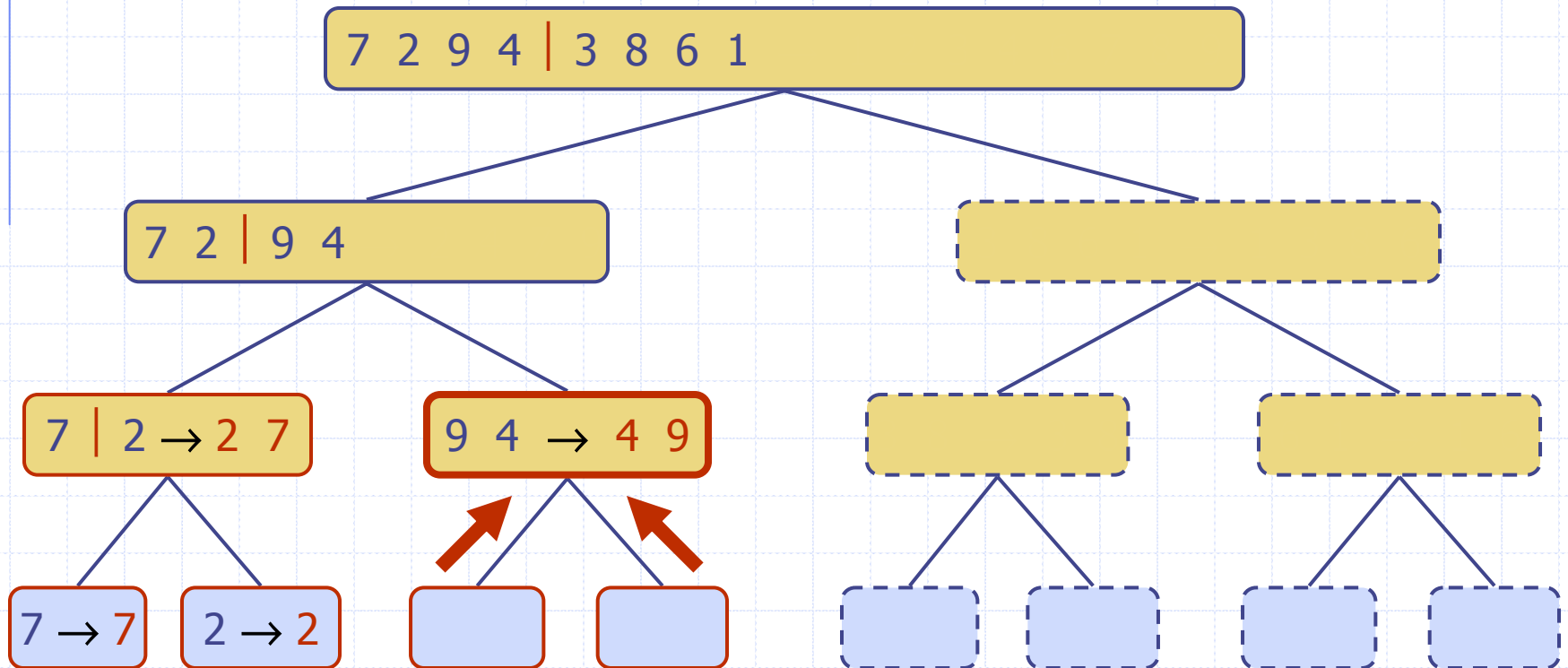
Execution Example (cont.)

◆ Merge



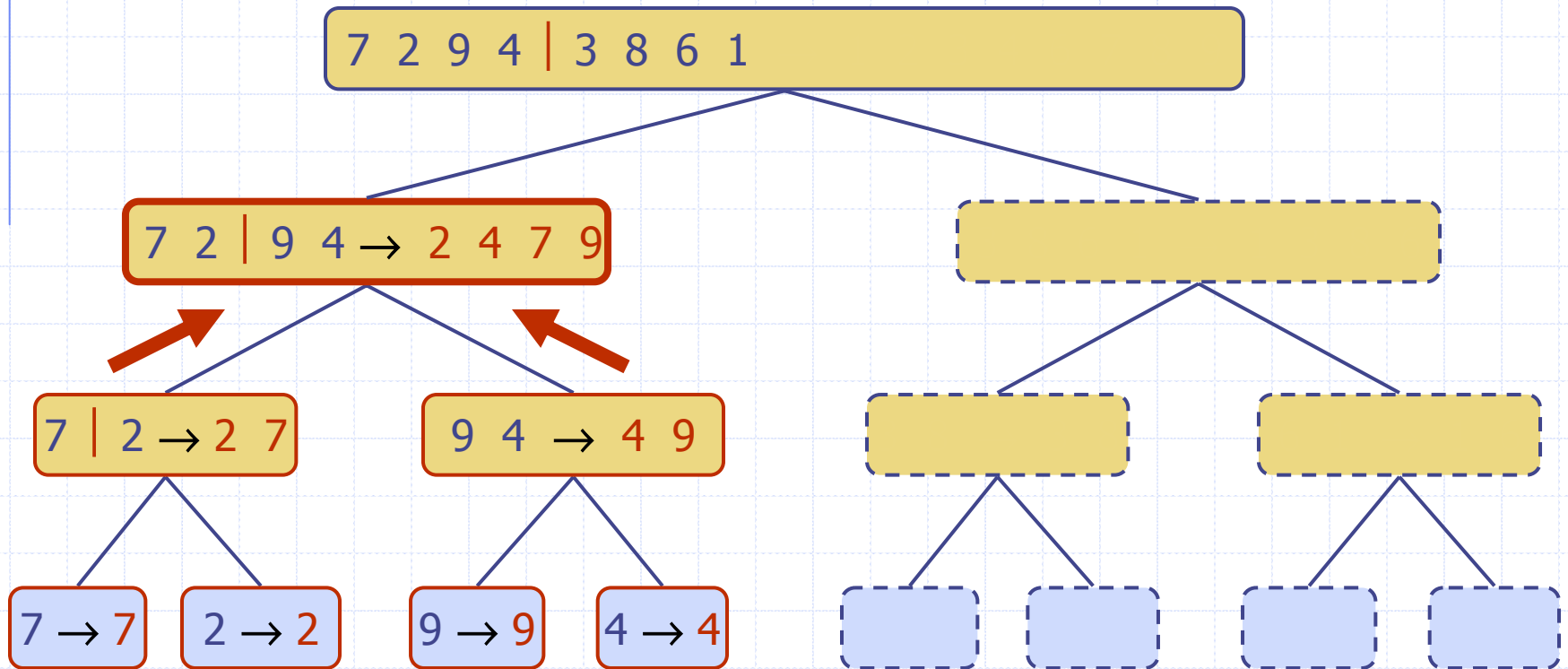
Execution Example (cont.)

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



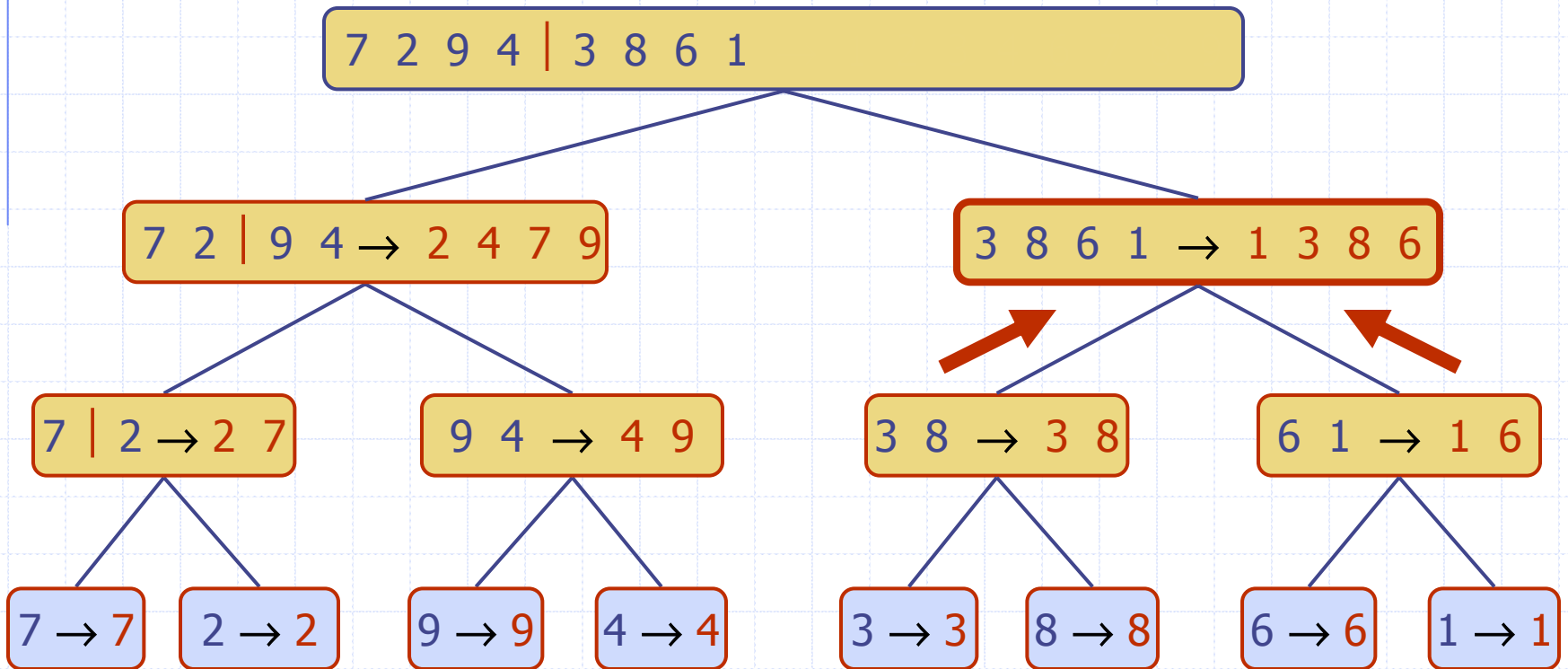
Execution Example (cont.)

◆ Merge



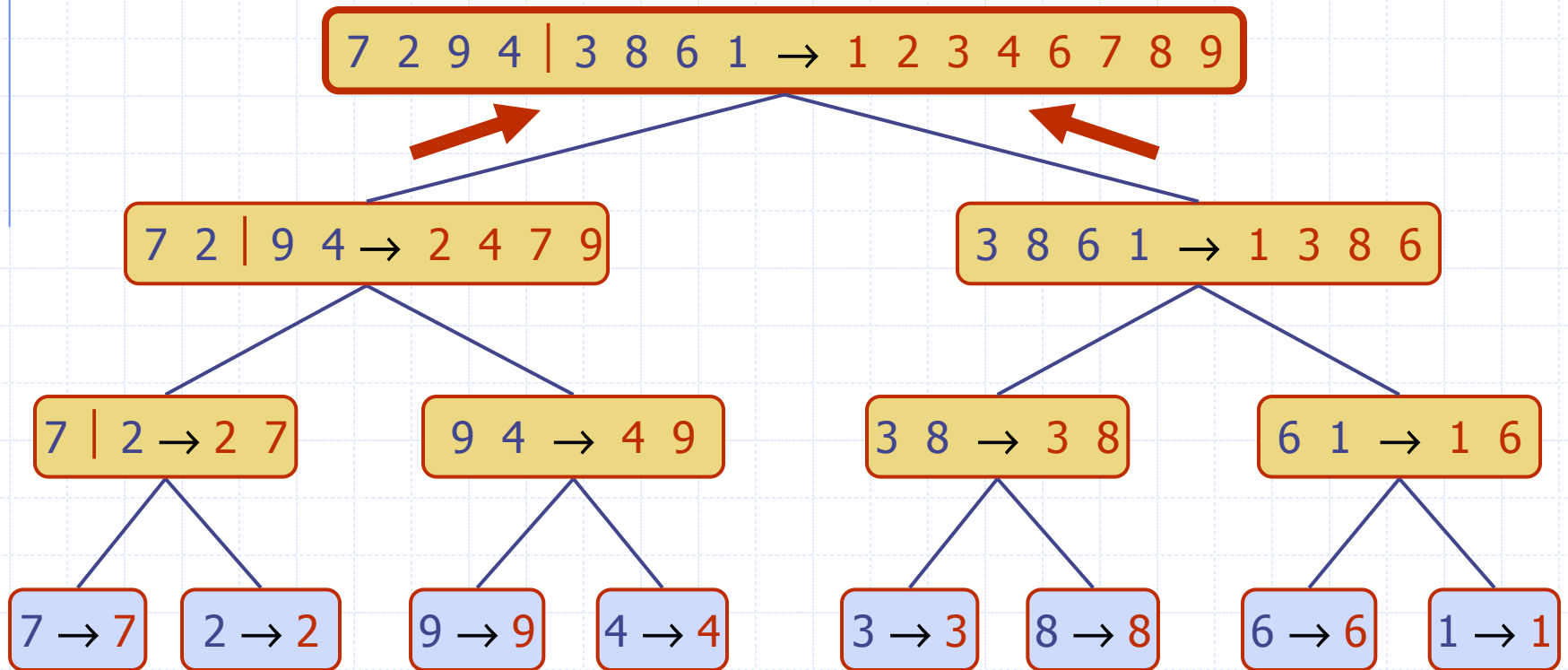
Execution Example (cont.)

◆ 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, divide sequence in half (or +/- .5 if n is odd)
- ◆ The overall amount of work done at the nodes of depth i is $O(n)$
 - partition and merge 2^i sequences of size $n/2^i$ (i is at most $\log n$)
 - we make 2^{i+1} recursive calls
- ◆ Thus, the total running time of merge-sort is $O(n \log n)$

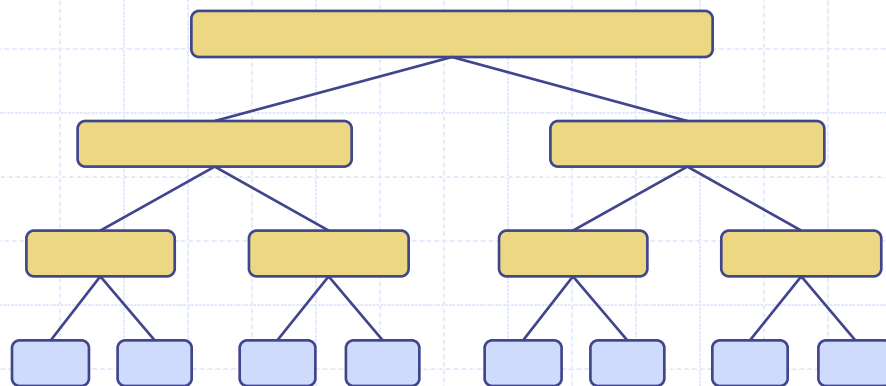
depth	#seqs	size
-------	-------	------

0	1	n
---	---	-----

1	2	$n/2$
---	---	-------

i	2^i	$n/2^i$
-----	-------	---------

...
-----	-----	-----



Generic Merging

- ◆ Generalized merge of two sorted sequences A and B
- ◆ Template method **genericMerge**
- ◆ Auxiliary methods
 - **aIsLess**
 - **bIsLess**
 - **bothAreEqual**
- ◆ Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

Algorithm *genericMerge*(A, B)

$S \leftarrow$ empty sequence

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

$a \leftarrow A.first().element(); b \leftarrow B.first().element()$

if $a < b$

aIsLess(a, S); $A.remove(A.first())$

else if $b < a$

bIsLess(b, S); $B.remove(B.first())$

else { $b = a$ }

bothAreEqual(a, b, S)

$A.remove(A.first()); B.remove(B.first())$

while $\neg A.isEmpty()$

aIsLess(a, S); $A.remove(A.first())$

while $\neg B.isEmpty()$

bIsLess(b, S); $B.remove(B.first())$

return S

Set Operations

- ◆ We represent a set by the sorted sequence of its elements
- ◆ By specializing the auxiliary methods the generic merge algorithm can be used to perform basic set operations:
 - union
 - intersection
 - subtraction
- ◆ The running time of an operation on sets A and B is $O(n_A + n_B)$
- ◆ Set union:
 - *aIsLess(a, S)*
S.insertLast(a)
 - *bIsLess(b, S)*
S.insertLast(b)
 - *bothAreEqual(a, b, S)*
S.insertLast(a)
- ◆ Set intersection:
 - *aIsLess(a, S)*
{ *do nothing* }
 - *bIsLess(b, S)*
{ *do nothing* }
 - *bothAreEqual(a, b, S)*
S.insertLast(a)

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)