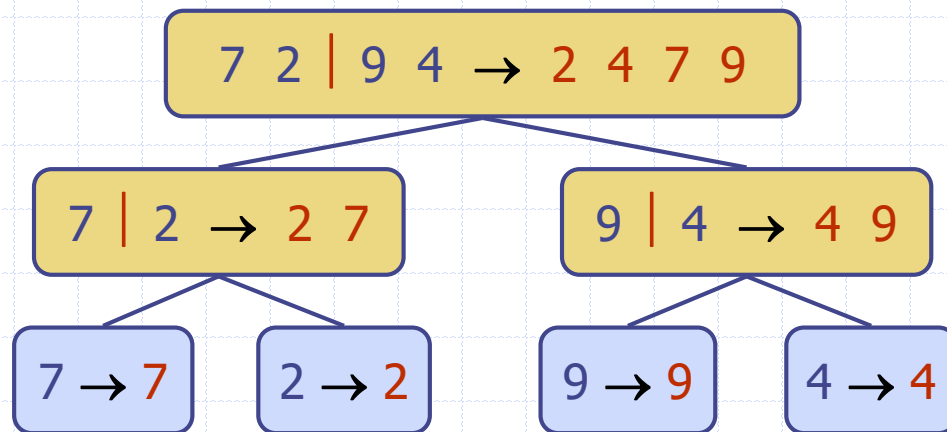


# Merge Sort



# Divide-and-Conquer (§ 10.1.1)

- ◆ **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 (§ 10.1)

- ◆ 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**  $S.size() > 1$

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

*mergeSort*( $S_1, C$ )

*mergeSort*( $S_2, C$ )

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

# 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**  $\neg A.empty() \wedge \neg B.empty()$

**if**  $A.front() < B.front()$

$S.addBack(A.front()); A.eraseFront();$

**else**

$S.addBack(B.front()); B.eraseFront();$

**while**  $\neg A.empty()$

$S.addBack(A.front()); A.eraseFront();$

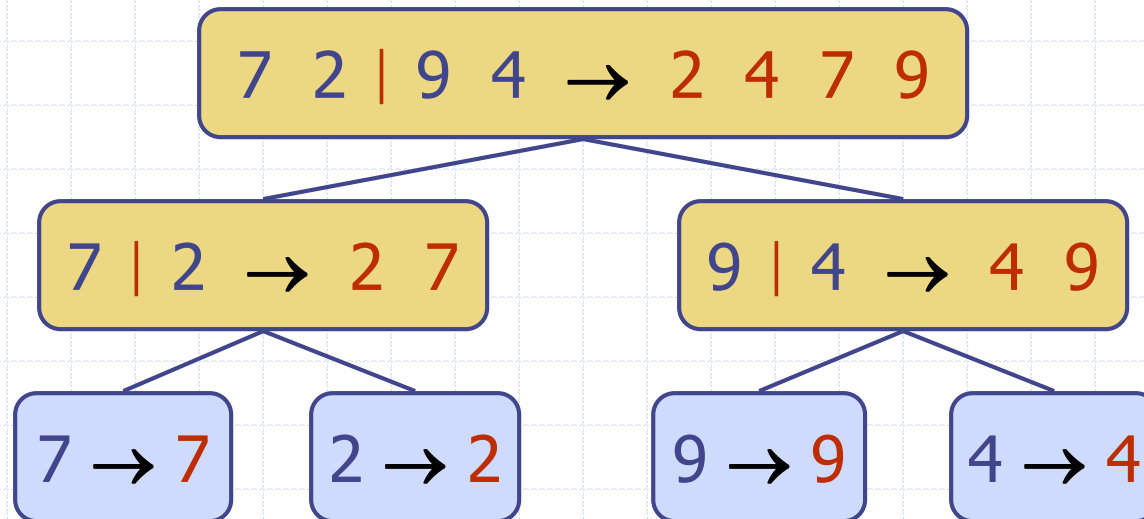
**while**  $\neg B.empty()$

$S.addBack(B.front()); B.eraseFront();$

**return**  $S$

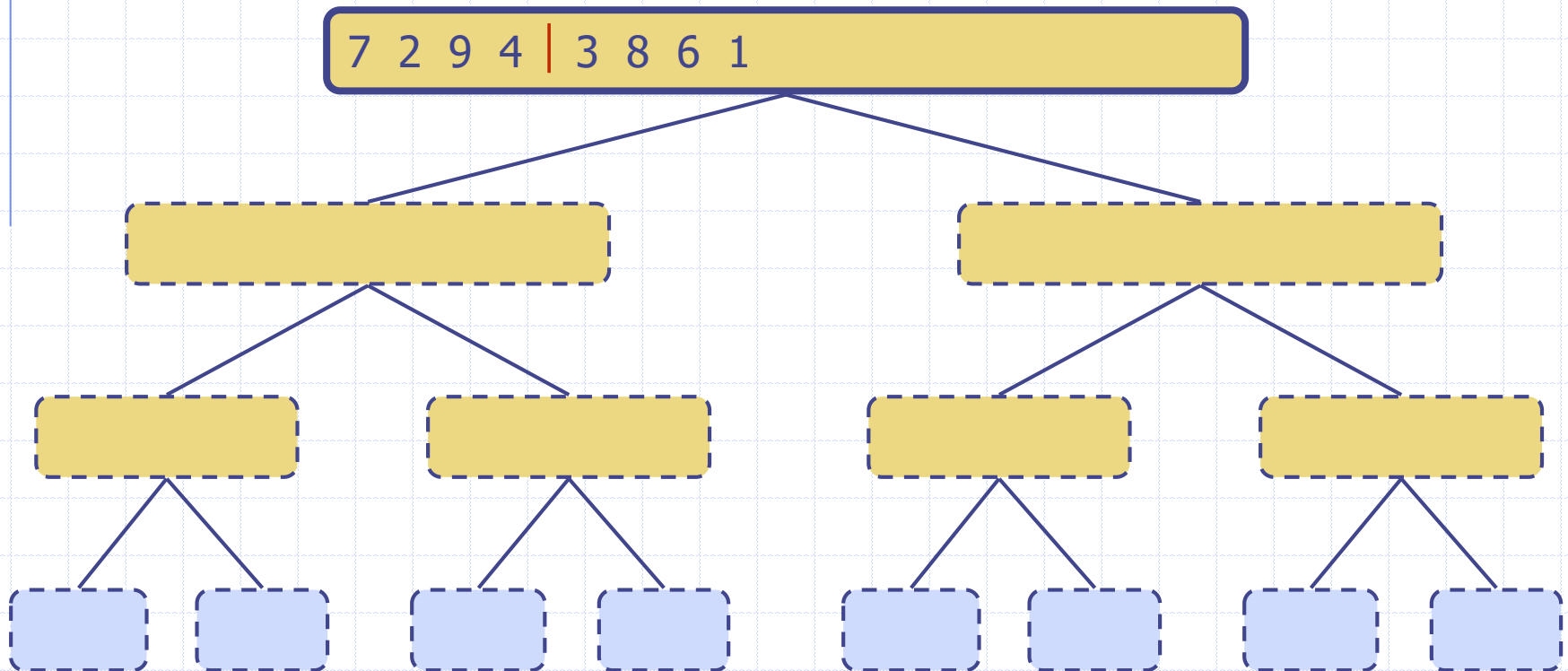
# Merge-Sort Tree

- ◆ 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



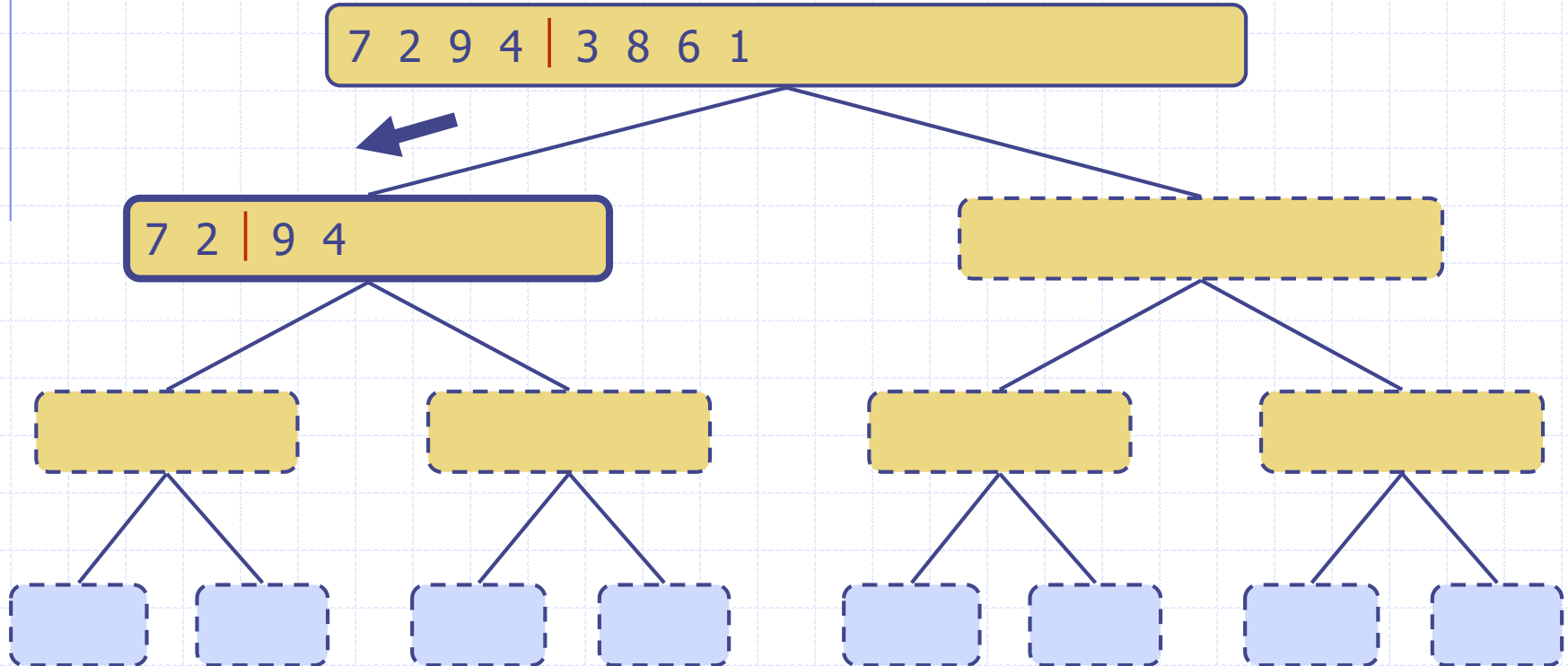
# Execution Example

## ◆ Partition



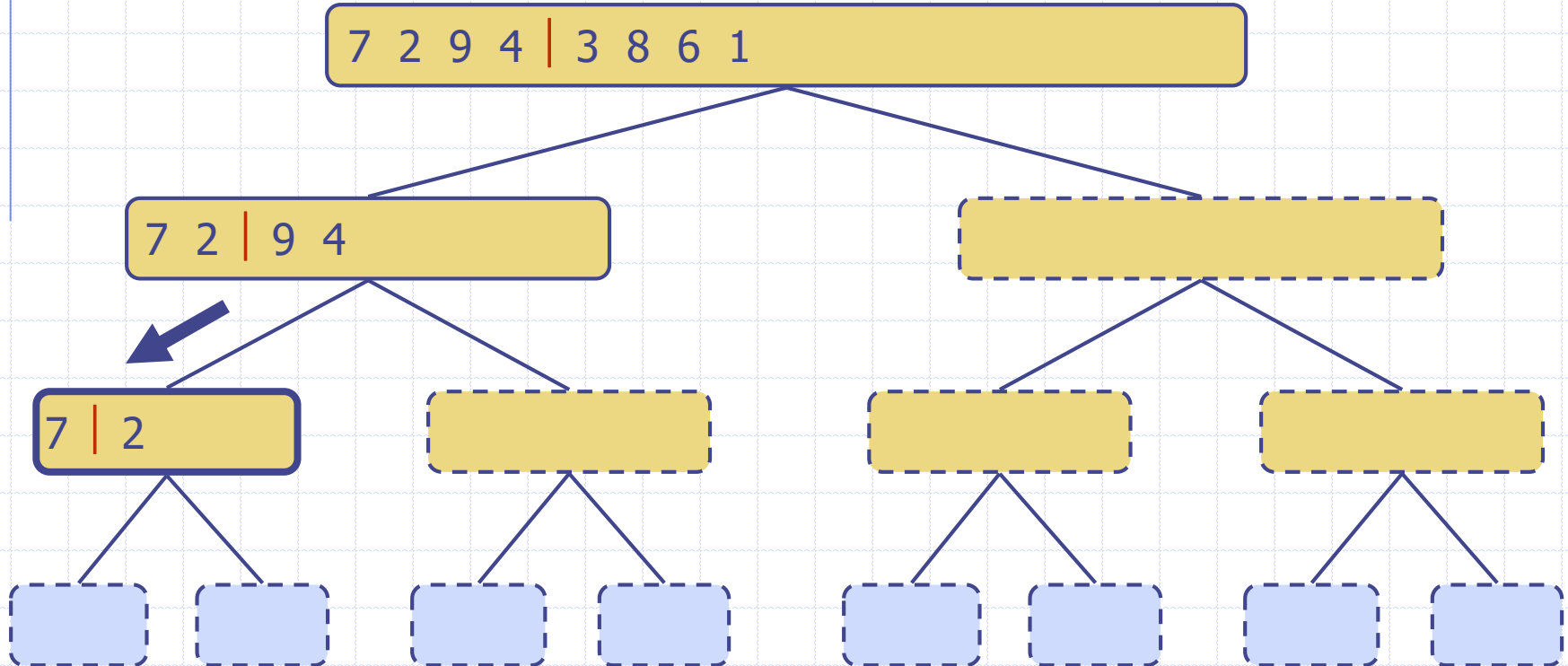
# Execution Example (cont.)

## ◆ Recursive call, partition



# Execution Example (cont.)

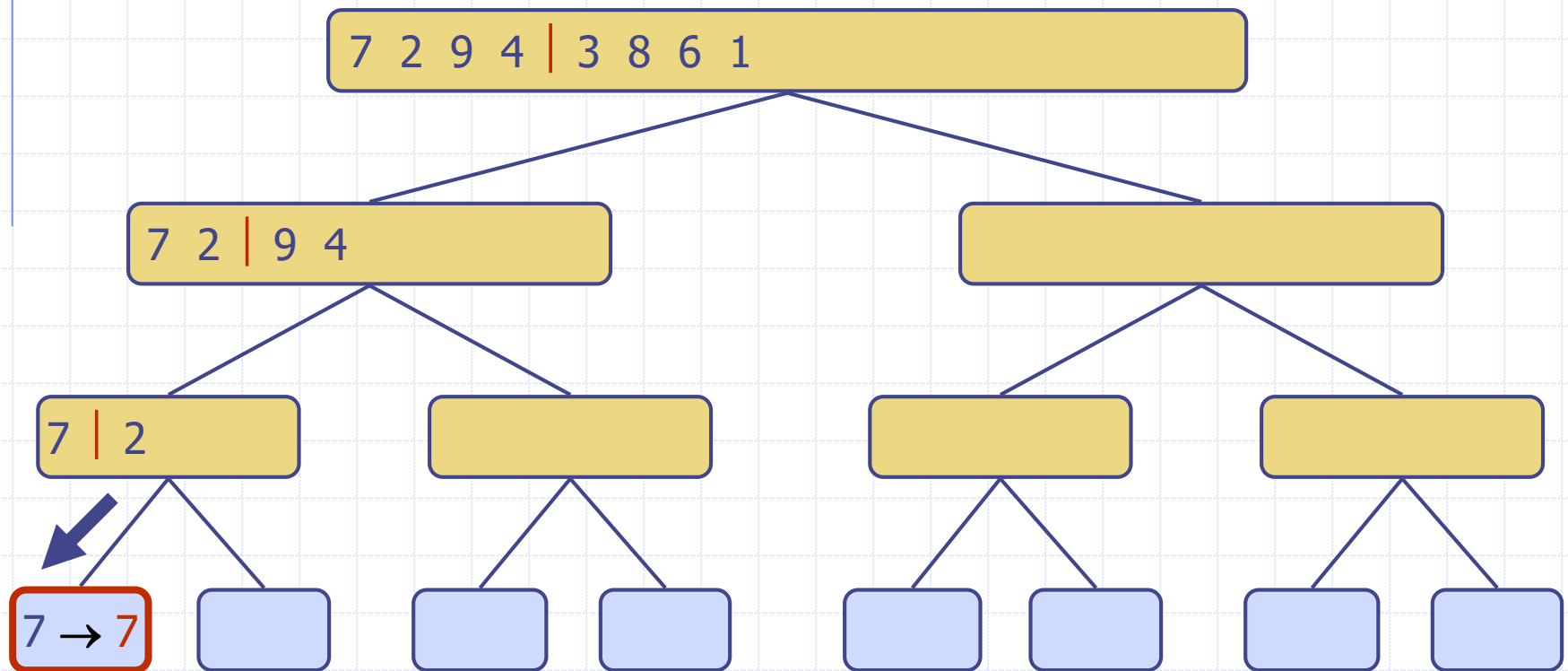
## ◆ Recursive call, partition





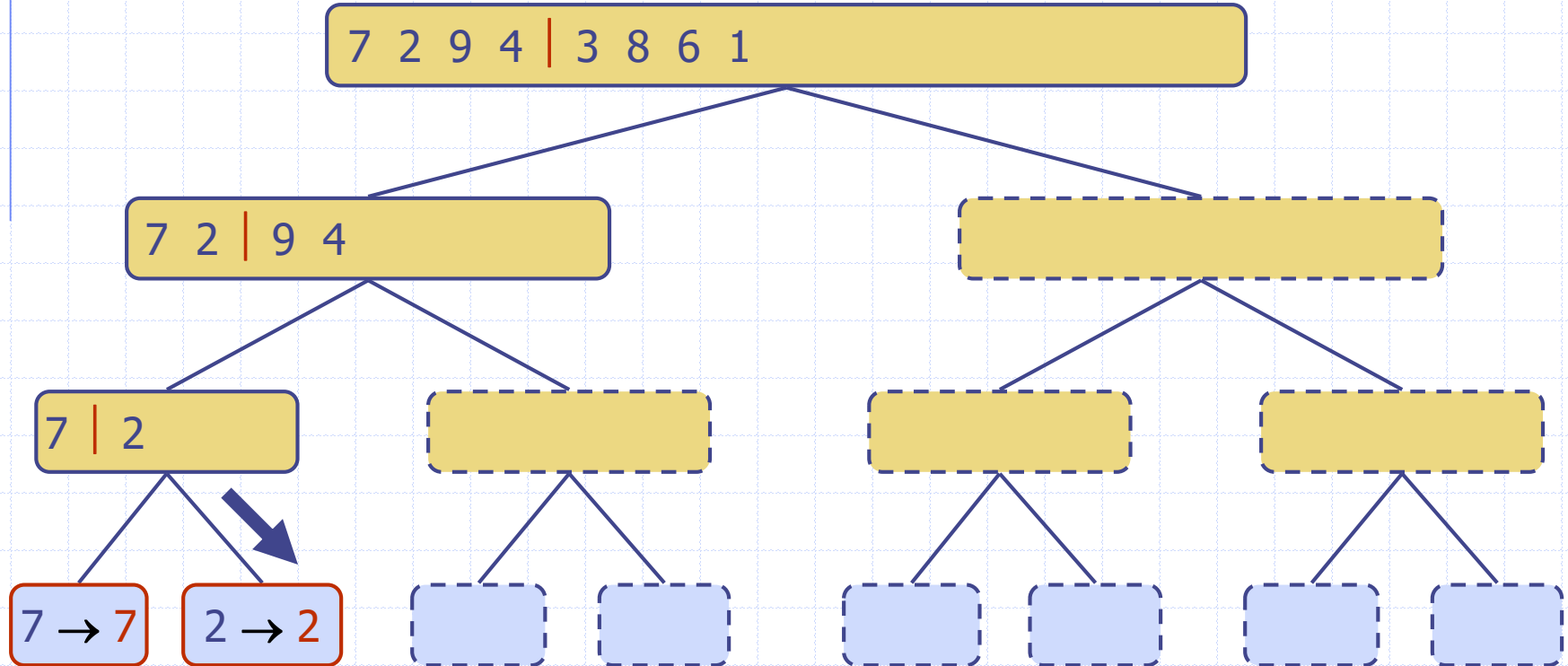
# Execution Example (cont.)

◆ Recursive call, base case



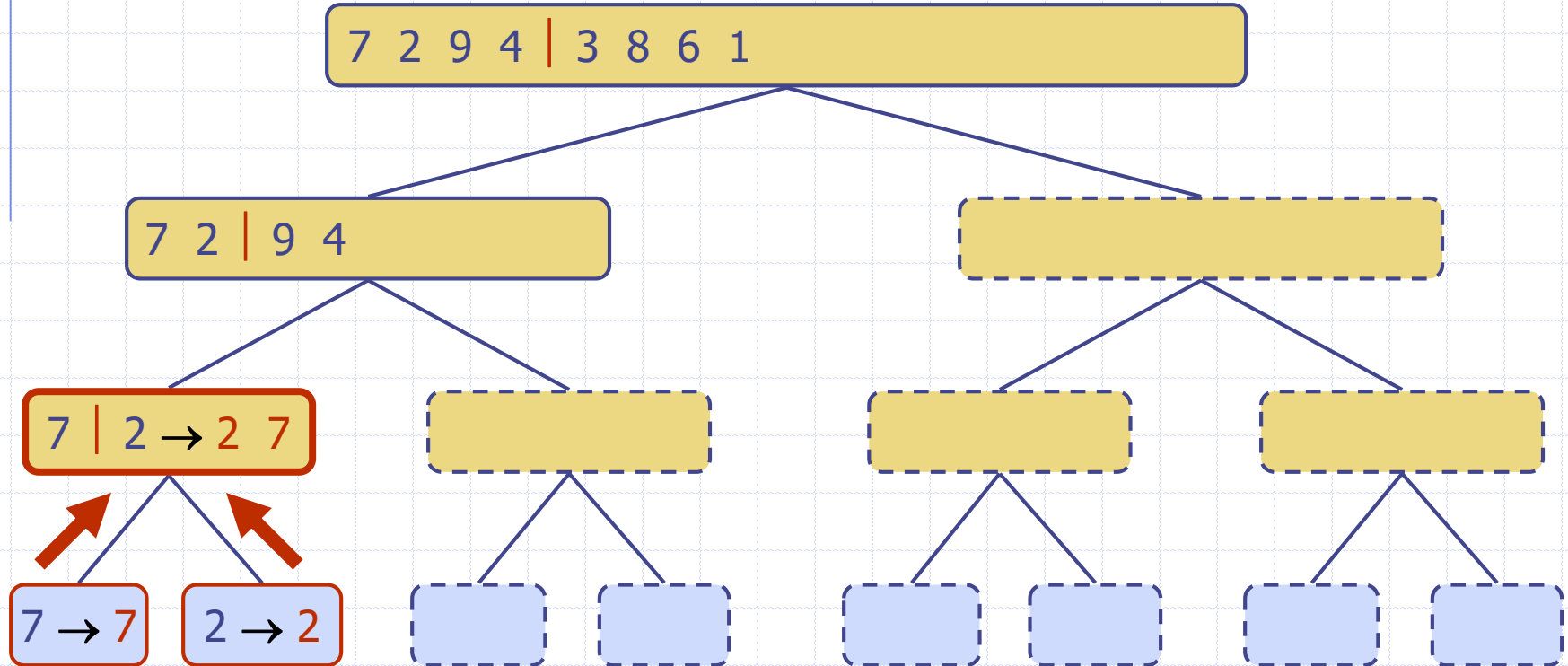
# Execution Example (cont.)

◆ Recursive call, base case



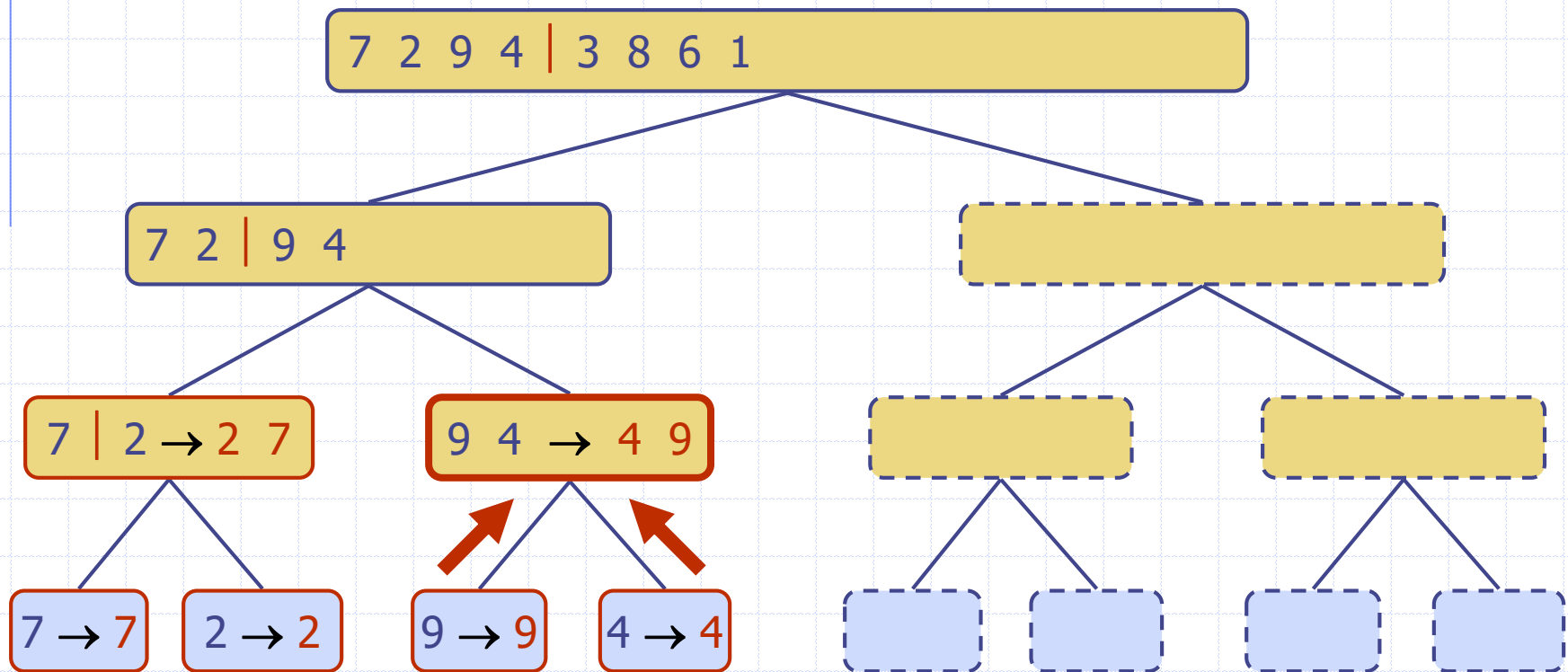
# Execution Example (cont.)

## ◆ Merge



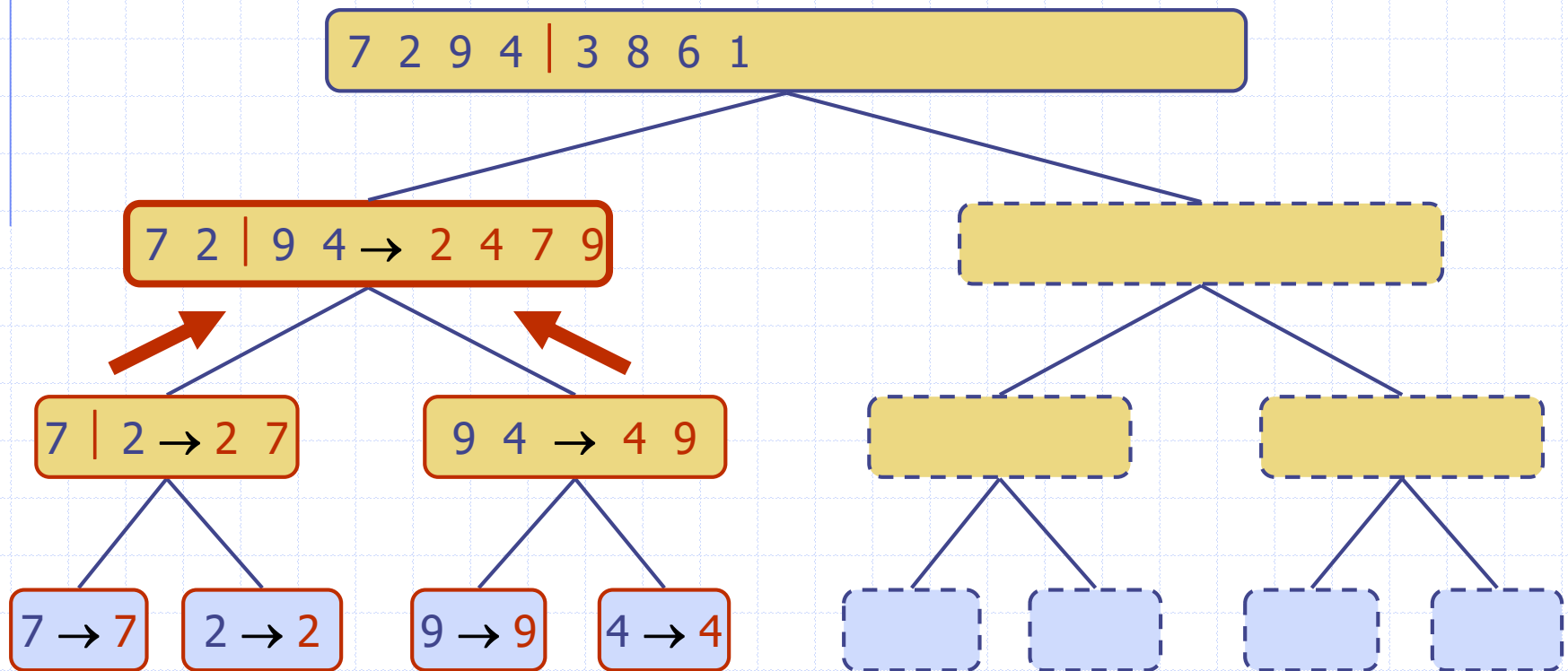
# Execution Example (cont.)

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



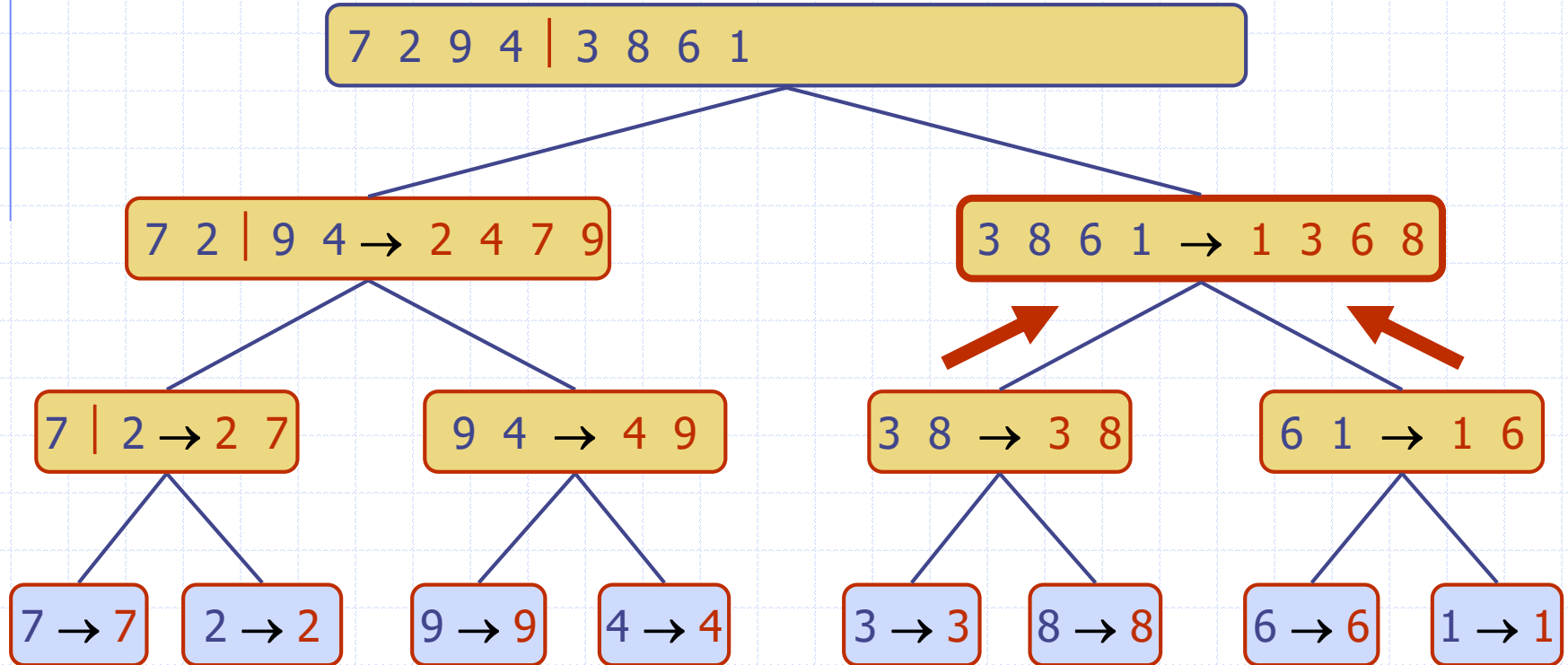
# Execution Example (cont.)

## ◆ Merge



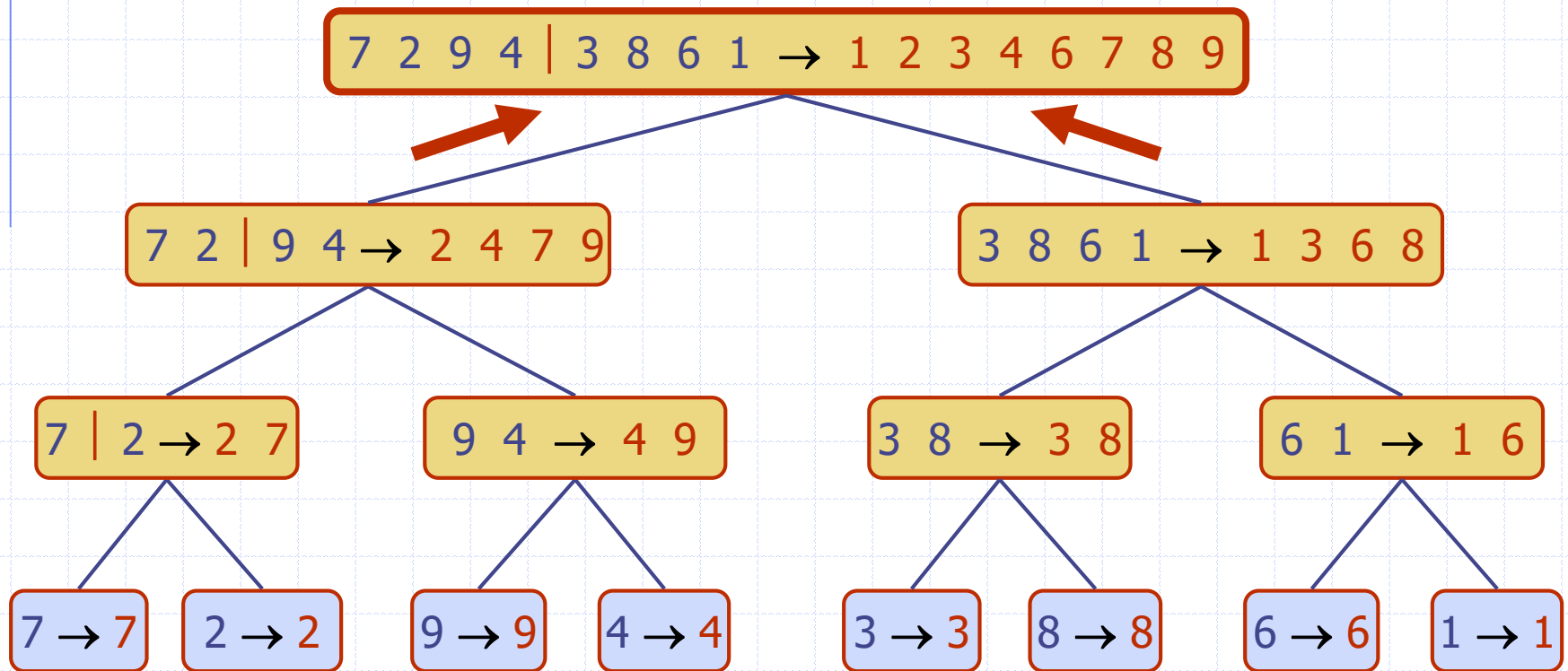
# Execution Example (cont.)

◆ 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
- ◆ Thus, the total running time of merge-sort is  $O(n \log n)$

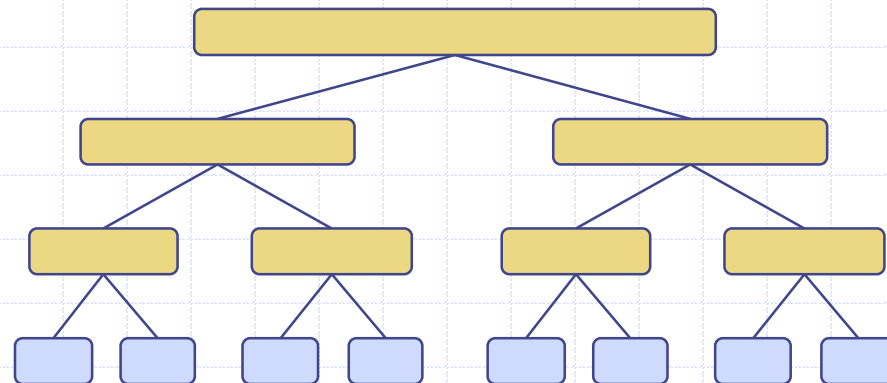
depth	#seqs	size
-------	-------	------

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