MergeSort

MergeSort

- MergeSort is a *divide and conquer* method of sorting

MergeSort Algorithm

- MergeSort is a recursive sorting procedure that uses at most **O(n lg(n))** comparisons.
- To sort an array of **n** elements, we perform the following steps in sequence:
- If **n < 2** then the array is already sorted.
- Otherwise, **n > 1**, and we perform the following three steps in sequence:
  1. **Sort** the **left half** of the the array using MergeSort.
  2. **Sort** the **right half** of the the array using MergeSort.
  3. **Merge** the sorted left and right halves.

How to Merge

Here are two lists to be merged:

First: (12, 16, 17, 20, 21, 27)

Second: (9, 10, 11, 12, 19)

Compare**12** and **9**

First: (12, 16, 17, 20, 21, 27)

Second: (10, 11, 12, 19)

New: (9)

Compare **12** and **10**

First: (12, 16, 17, 20, 21, 27)

Second: (11, 12, 19)

New: (9, 10)

Merge Example

Compare **12** and **11**

First: (12, 16, 17, 20, 21, 27)

Second: (12, 19)

New: (9, 10, 11)

Compare **12** and **12**

First: (16, 17, 20, 21, 27)

Second: (12, 19)

New: (9, 10, 11, 12)

Merge Example

Compare **16** and **12**

- **First: (16, 17, 20, 21, 27)**
- **Second: (19)**
- **New: (9, 10, 11, 12, 12)**

Compare **16** and **19**

- **First: (17, 20, 21, 27)**
- **Second: (19)**
- **New: (9, 10, 11, 12, 12, 16)**

Merge Example

Compare **17** and **19**

- **First: (20, 21, 27)**
- **Second: (19)**
- **New: (9, 10, 11, 12, 12, 16, 17)**

Compare **20** and **19**

- **First: (20, 21, 27)**
- **Second: ( )**
- **New: (9, 10, 11, 12, 12, 16, 17, 19)**

Merge Example

Checkout **20** and **empty list**

First: ( )

Second: ( )

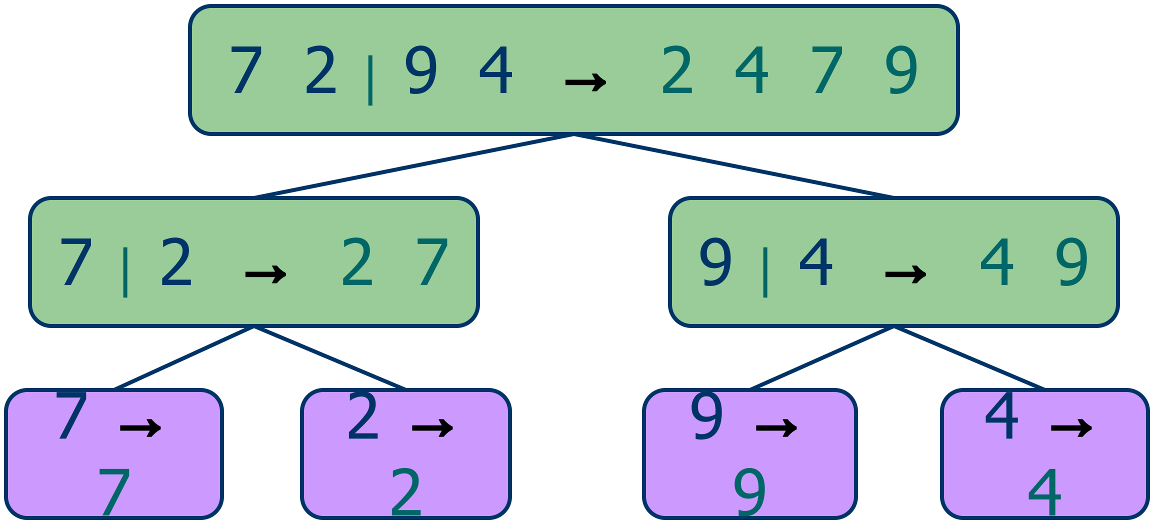
New: (9, 10, 11, 12, 12, 16, 17, 19, 20, 21, 27)

MergeSort

Original	24	13	26	1	12	27	38	15							
Divide in 2	24	13	26	1		12	27	38	15						
Divide in 4	24	13		26	1		12	27		38	15				
Divide in 8	24		13		26		1		12		27		38		15
Merge 2	13	24			1	26			12	27			15	38	
Merge 4	1	13	24	26					12	15	27	38			
Merge 8	1	12	13	15	24	26	27	38							

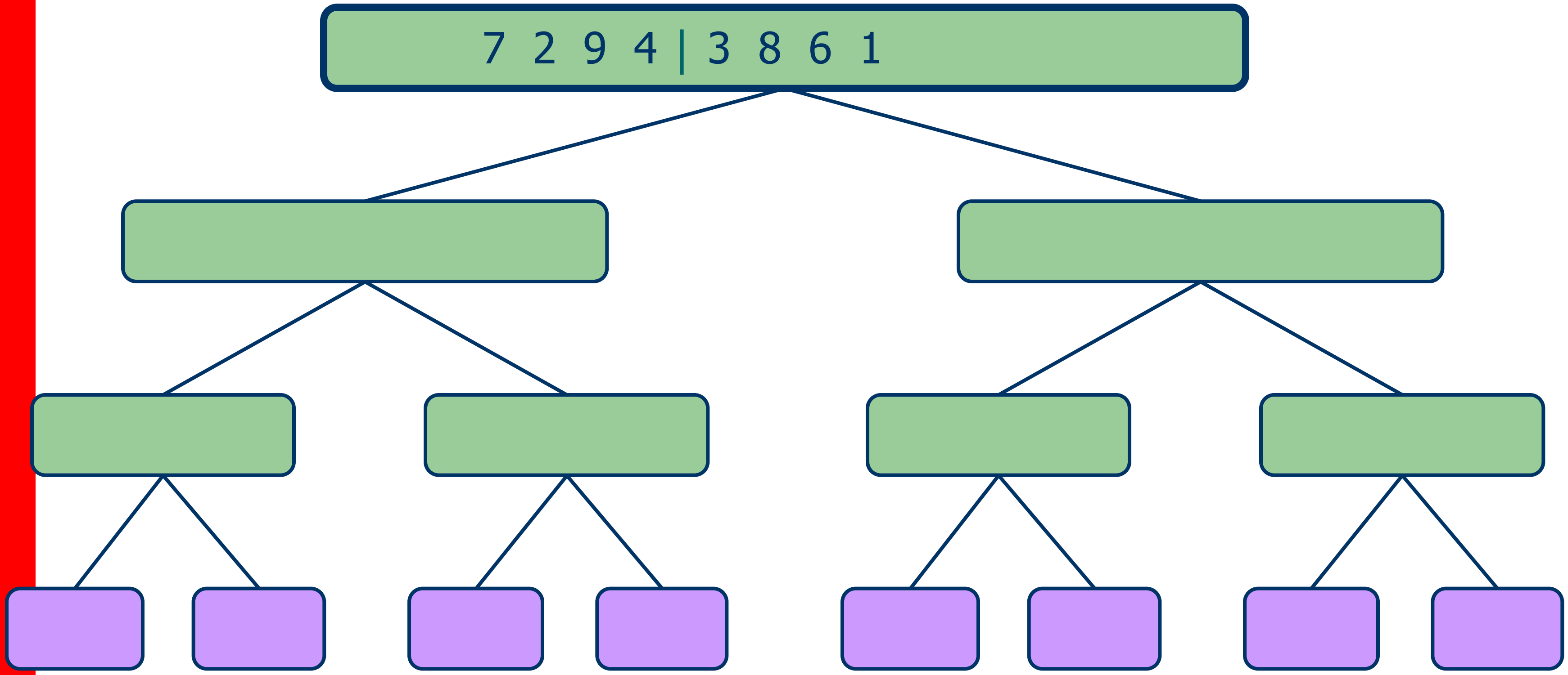
Merge-Sort Tree

- 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



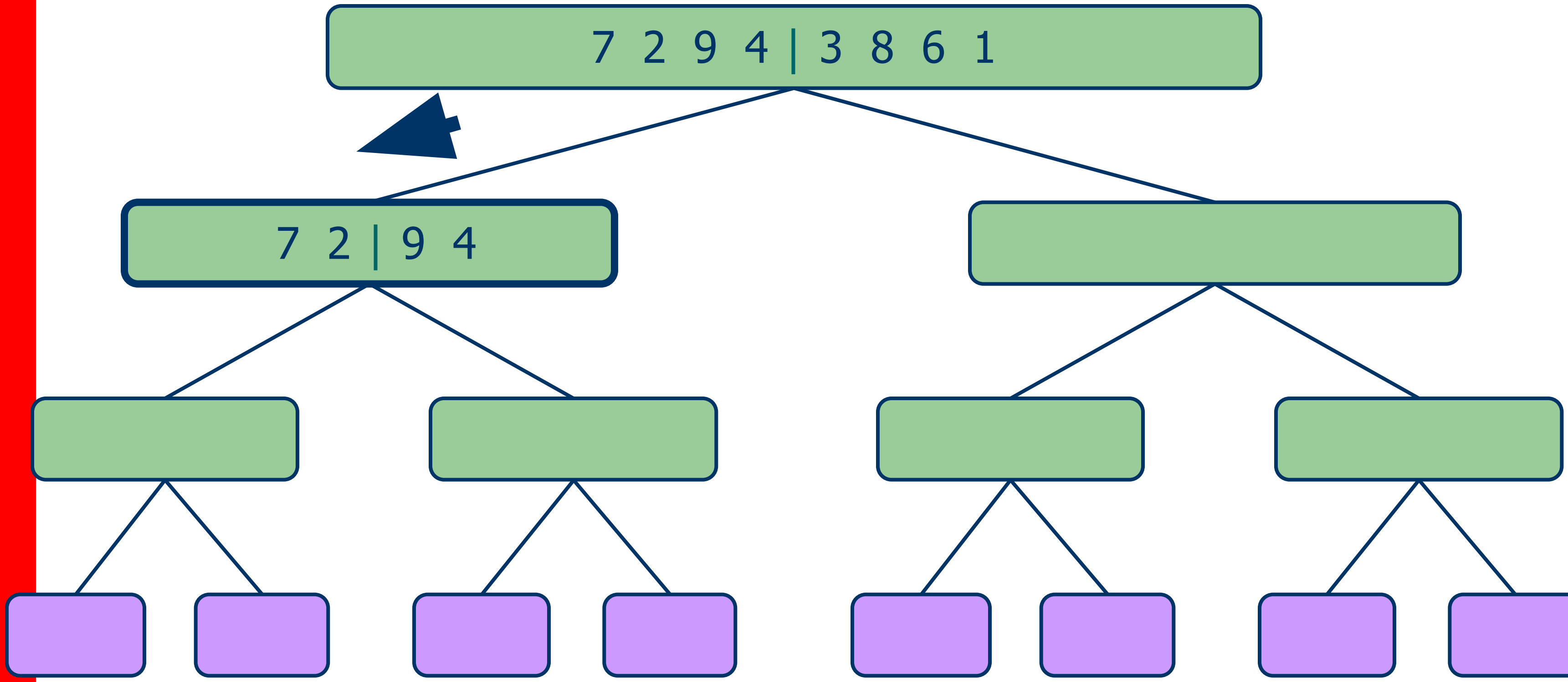
Execution Example

- Partition



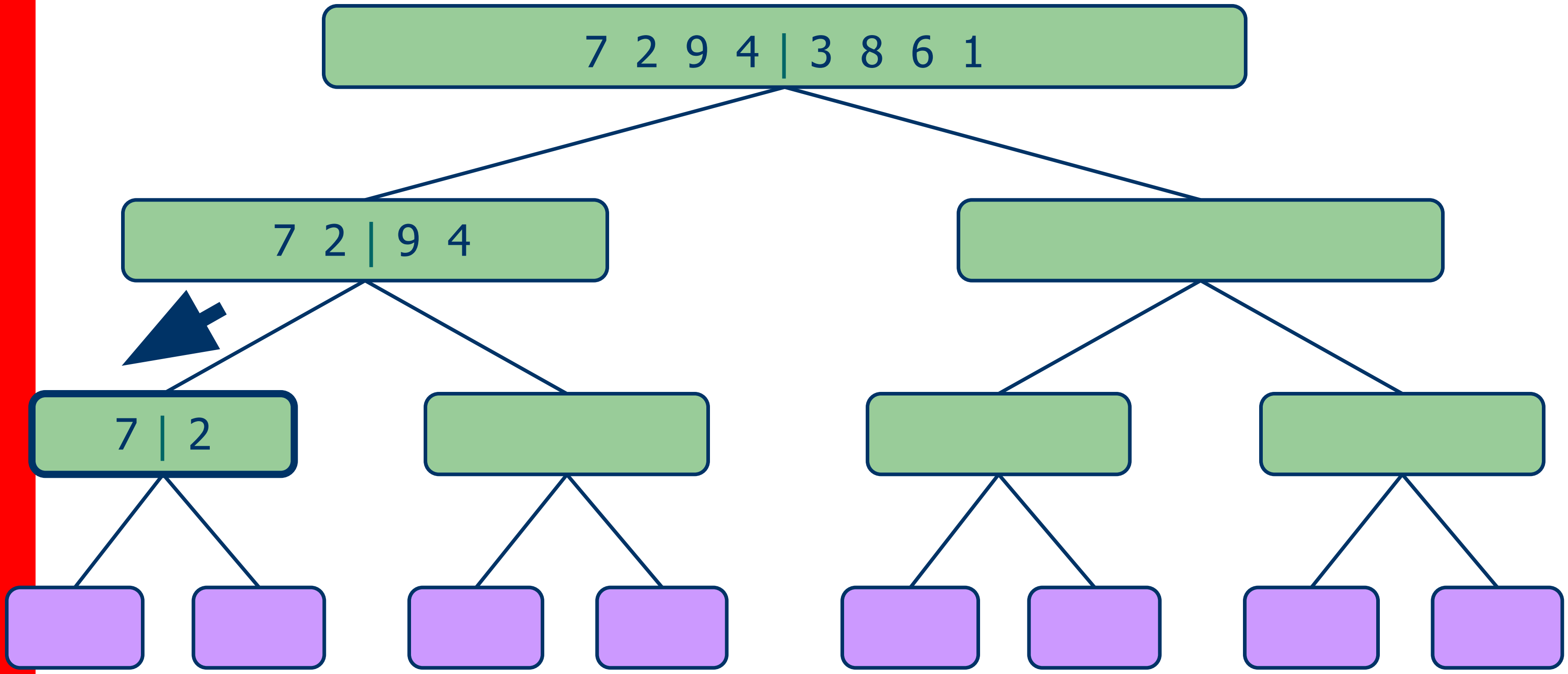
Execution Example (cont.)

- Recursive call, partition



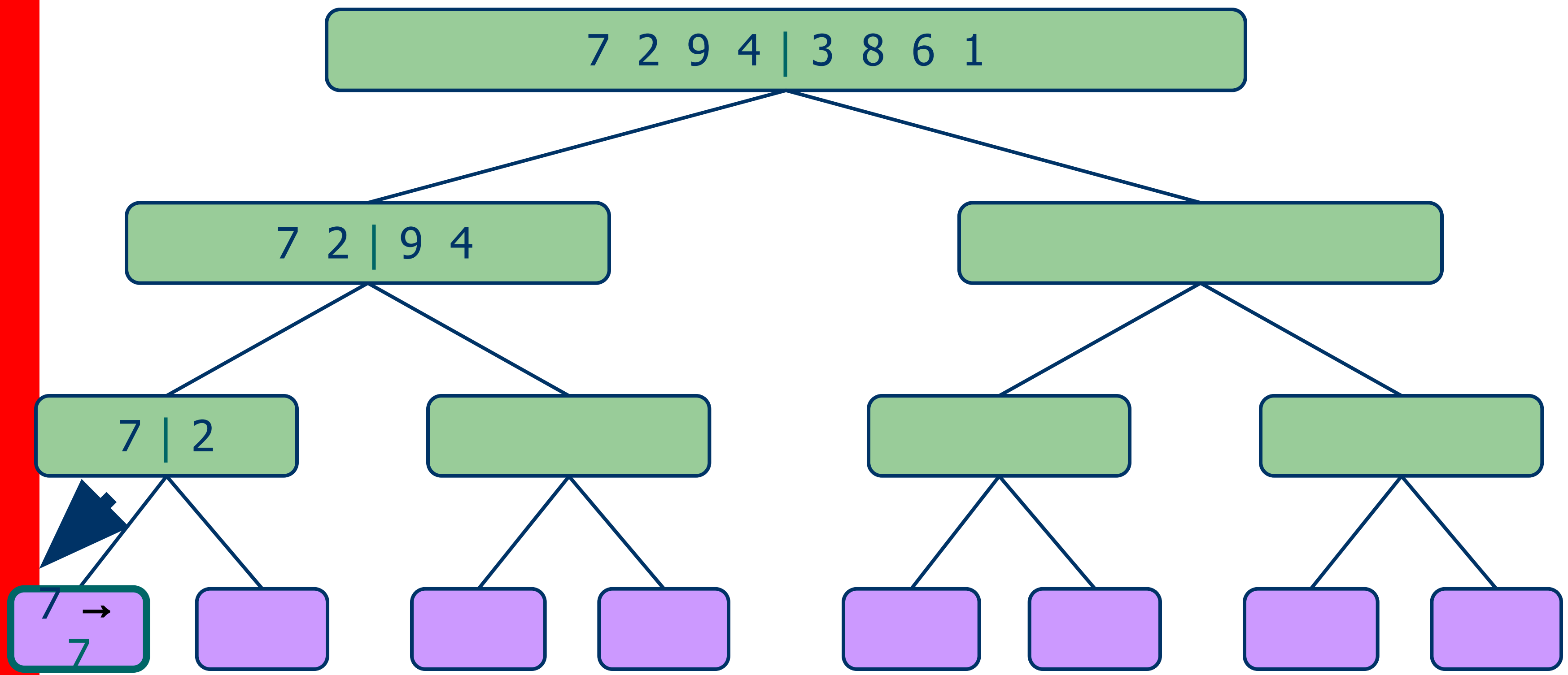
Execution Example (cont.)

- Recursive call, partition



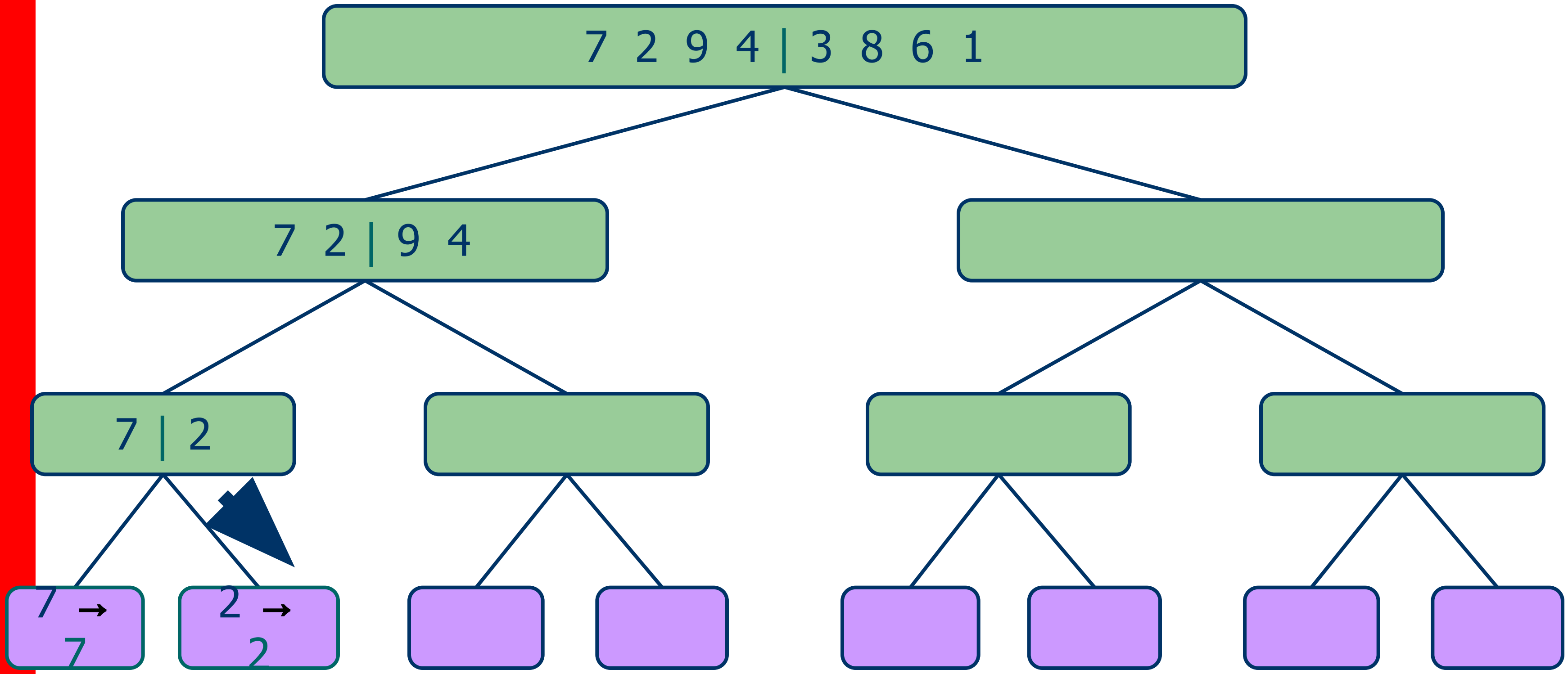
Execution Example (cont.)

- Recursive call, base case



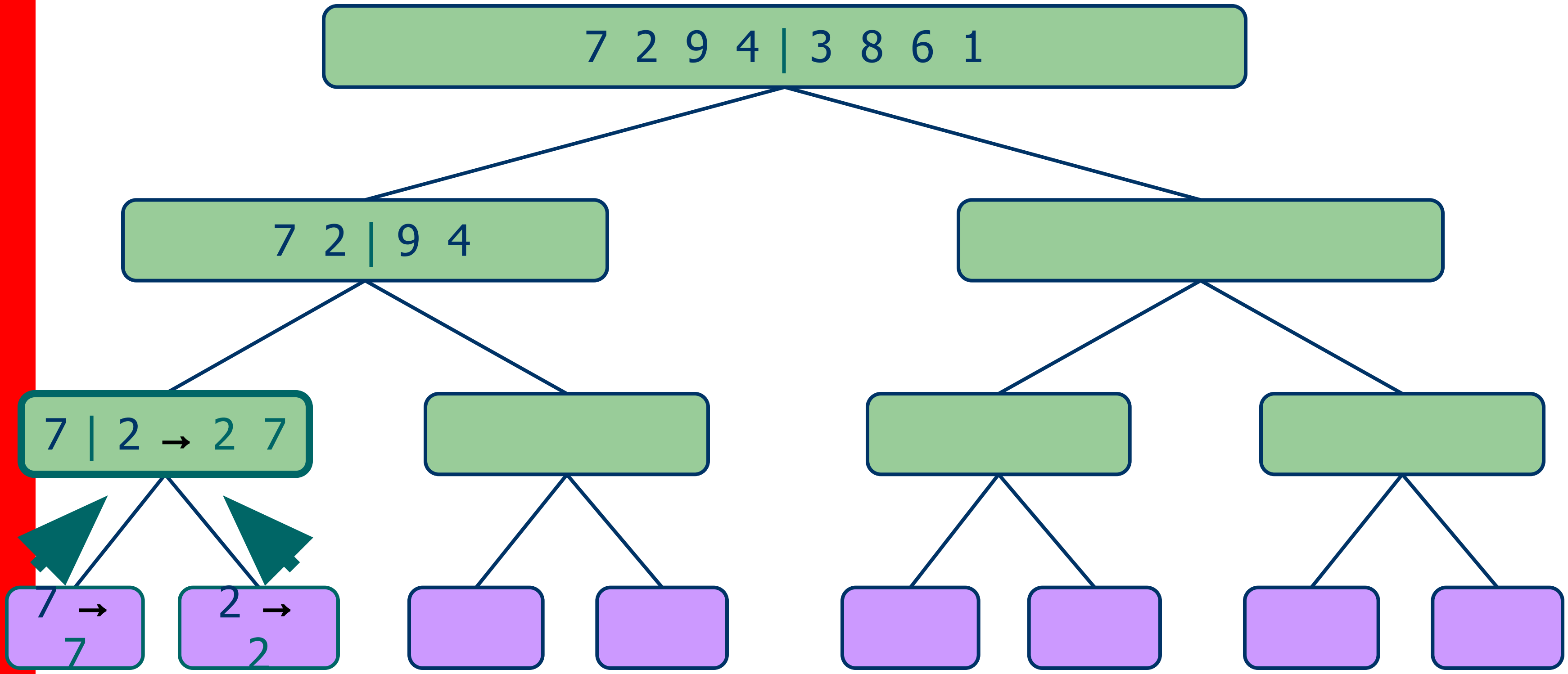
Execution Example (cont.)

- Recursive call, base case



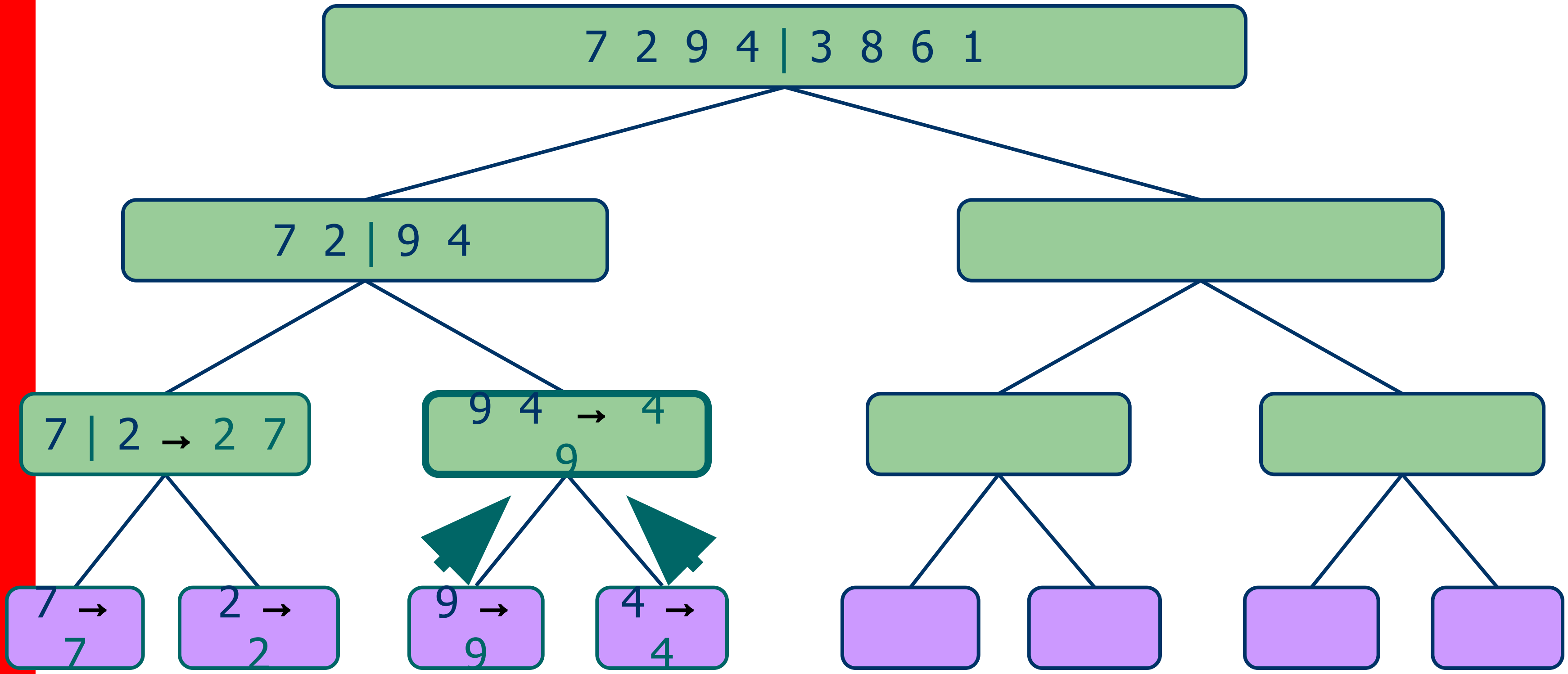
Execution Example (cont.)

- Merge



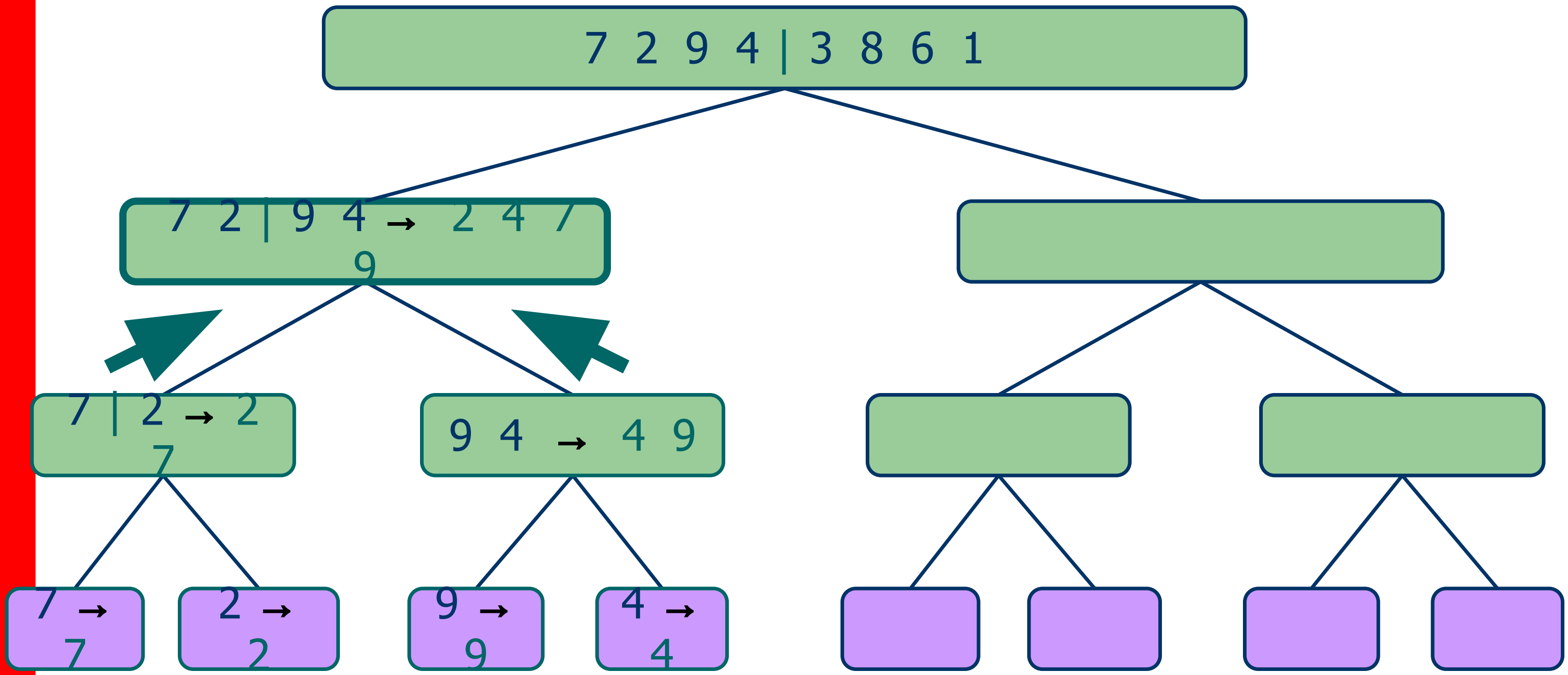
Execution Example (cont.)

- Recursive call, …, base case, merge



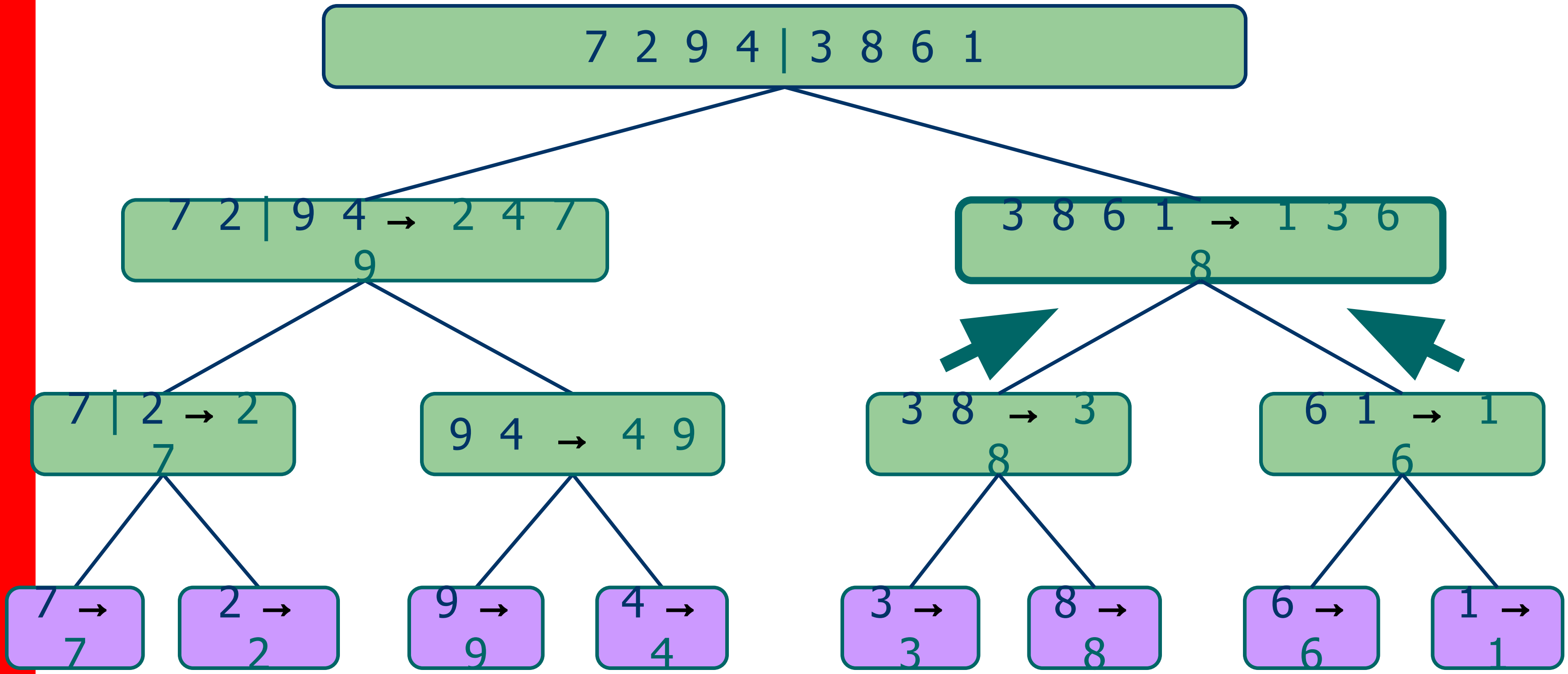
Execution Example (cont.)

- Merge



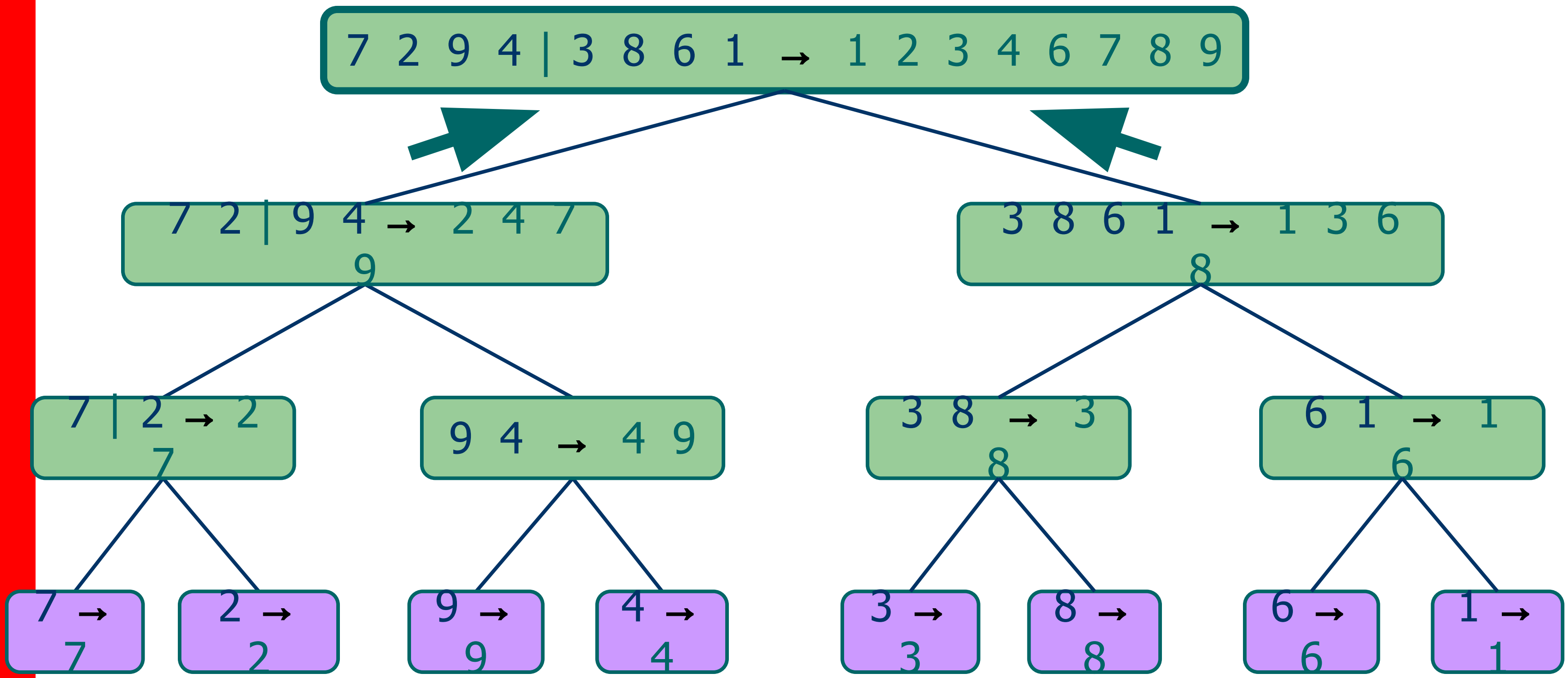
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



Complexity of MergeSort

Pass Number	Number of merges	Merge list length	# of comps / moves per merge
1	2^{k-1} or $n/2$	1 or $n/2^k$	$\le 2^1$
2	2^{k-2} or $n/4$	2 or $n/2^{k-1}$	$\le 2^2$
3	2^{k-3} or $n/8$	4 or $n/2^{k-2}$	$\le 2^3$
.	.	.	.
.	.	.	.
.	.	.	.
$k - 1$	2^1 or $n/2^{k-1}$	2^{k-2} or $n/4$	$\le 2^{k-1}$
k	2^0 or $n/2^k$	2^{k-1} or $n/2$	$\le 2^k$

$k = \log n$

Complexity of MergeSort

Multiplying **the number of merges** by the **maximum number of comparisons** per merge, we get:

$$(2^{k-1})2^1 \quad = 2^k$$

$$(2^{k-2})2^2 \quad = 2^k$$

.

.

.

$$(2^1)2^{k-1} \quad = 2^k$$

$$(2^0)2^k = 2^k$$

k **passes each require** 2^k **comparisons (and moves). But** $k = lg\ n$ **and hence, we get lg(n) · n comparisons or O(n lgn)**