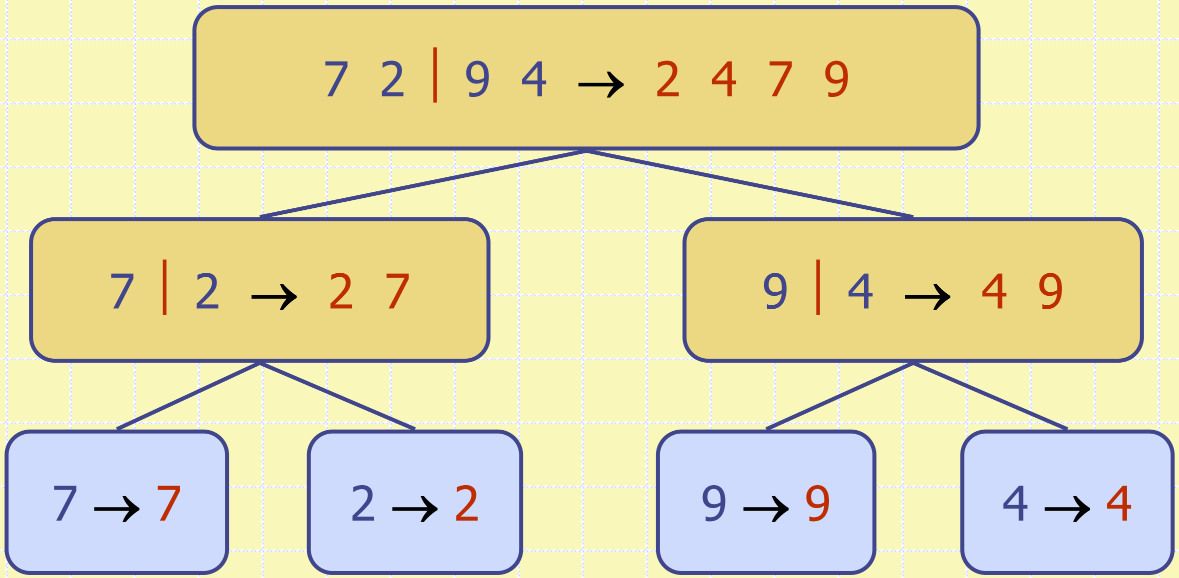
Merge-Sort

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

Coverage

- Merge-Sort



Divide-and-Conquer

- Divide-and conquer is a general algorithmic 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. In these cases the problem can be solved directly.

Merge-Sort

- Merge-sort uses divide-and-conquer to perform the sorting operation.
- 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 and implemented by means of a doubly linked list, takes $O(n)$ time.

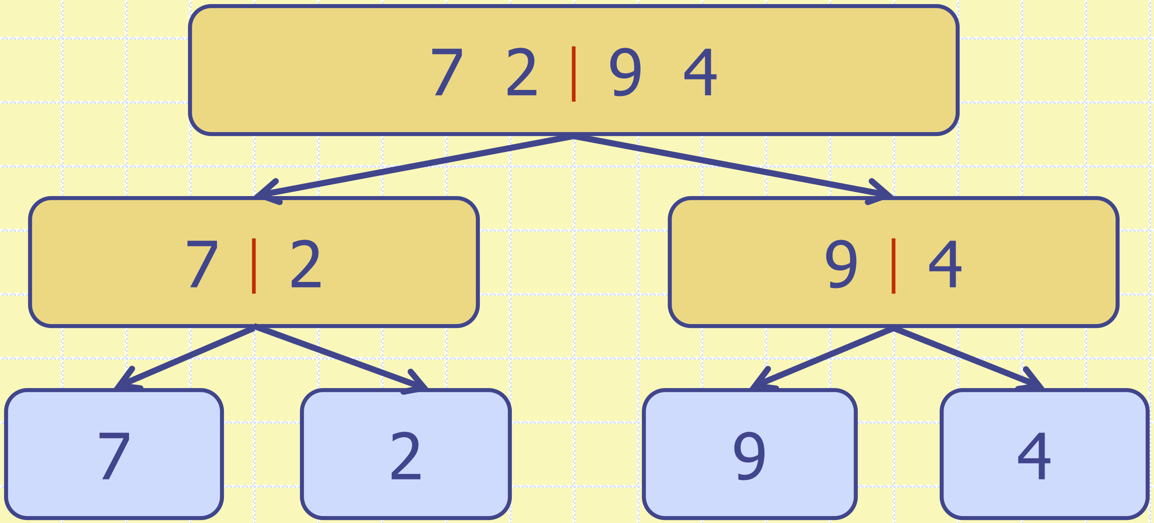
Algorithm *merge*(A, B)
    Input sequences A and B with
        $n/2$ elements each
    Output sorted sequence of $A \cup B$

    $S \leftarrow$ 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

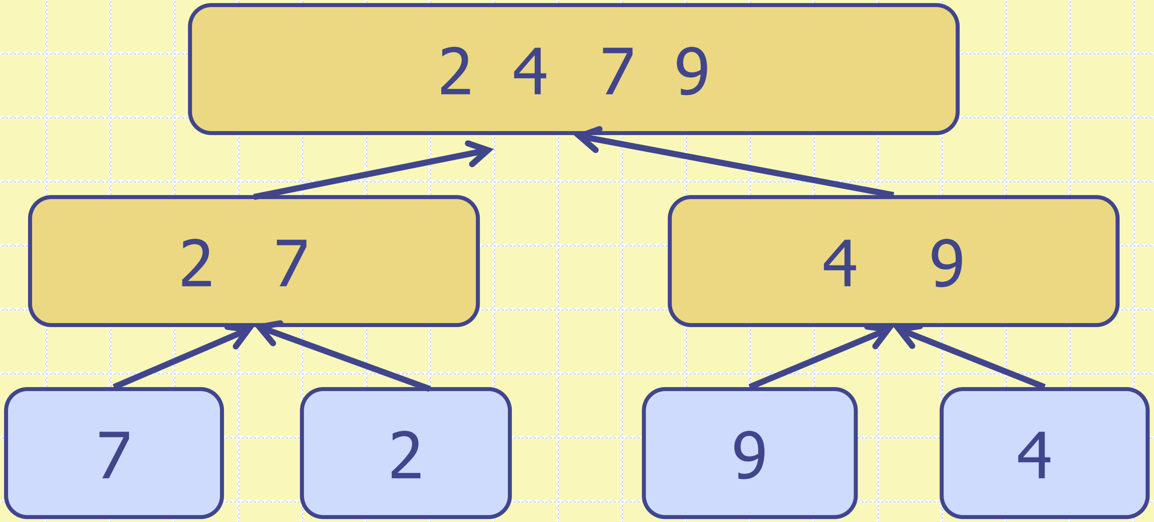
- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1

Divide



Merge-Sort Tree

Recur & Conquer



Merge-Sort Tree

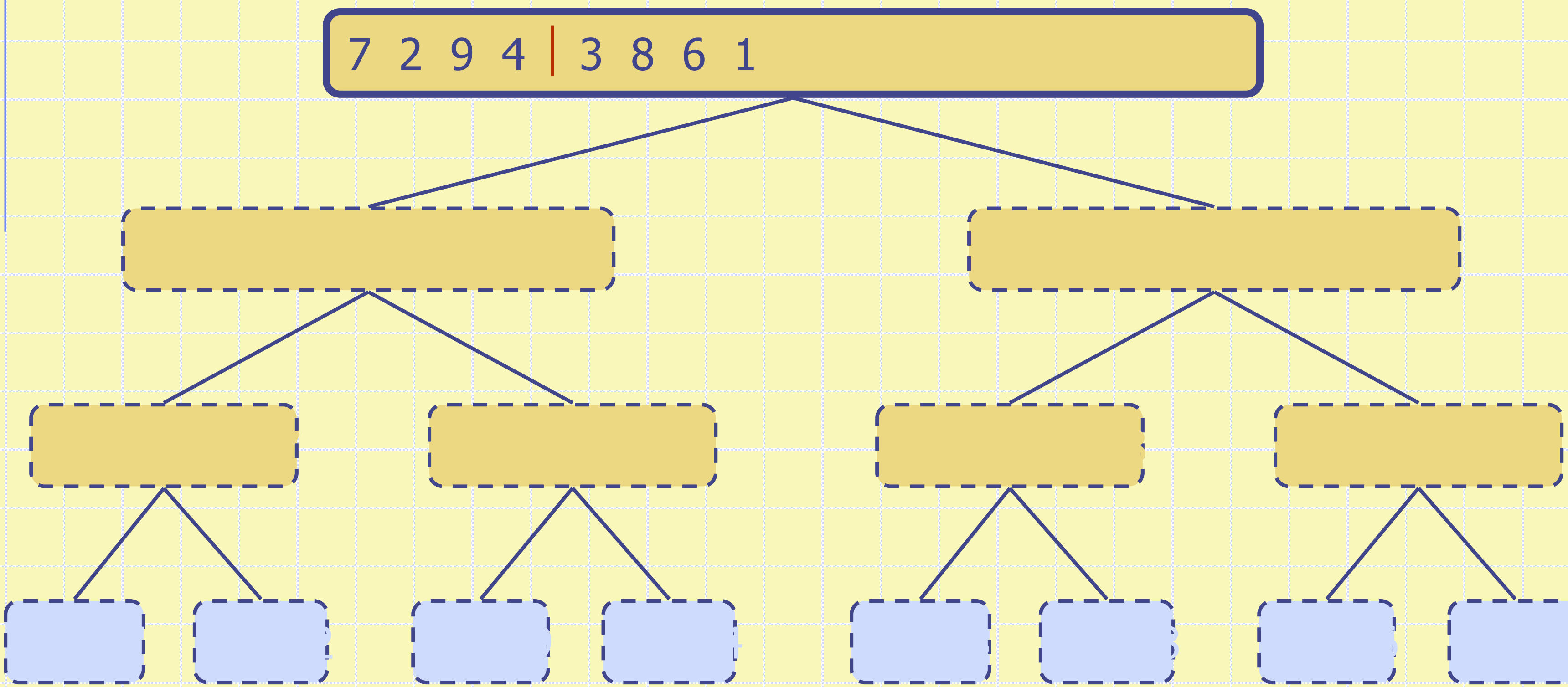
Example*:

6 5 3 1 8 7 2 4

*Reference: http://en.wikipedia.org/wiki/Merge_sort

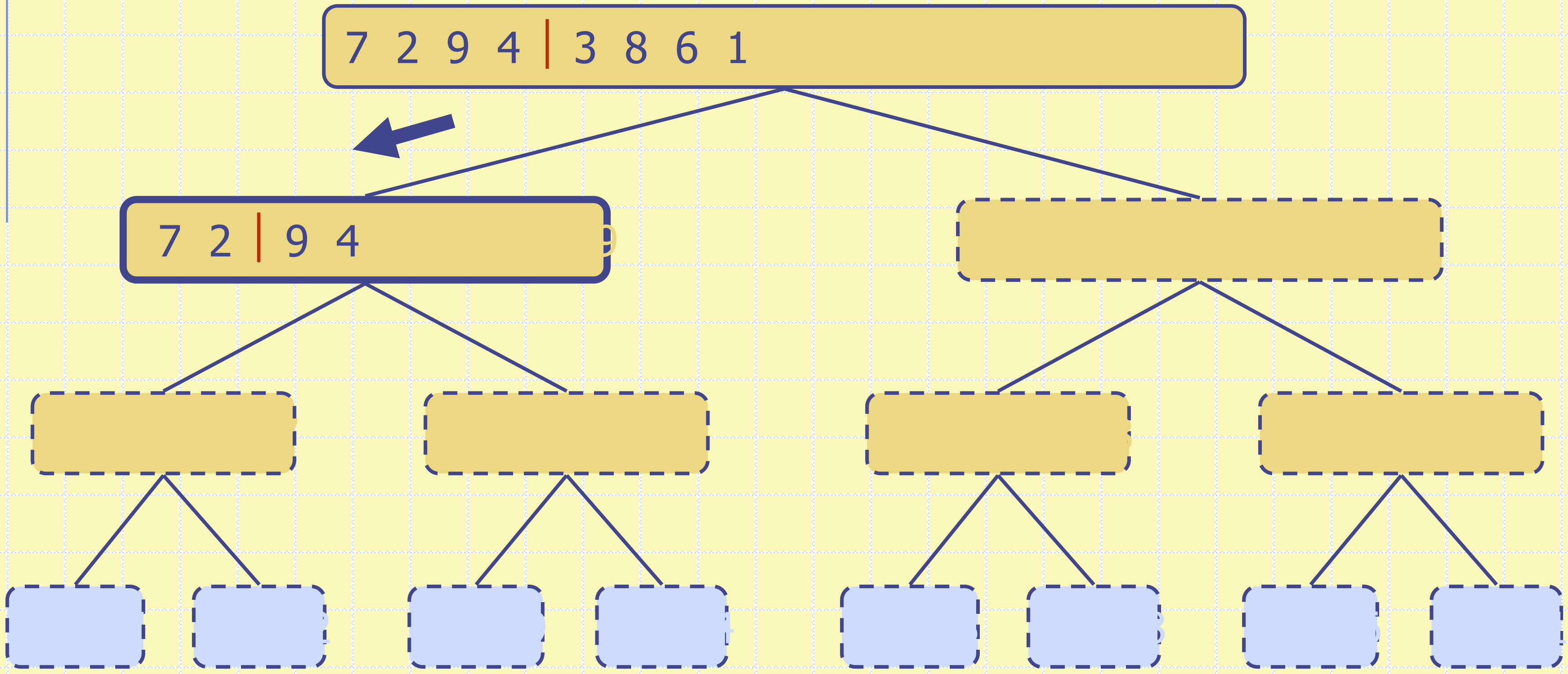
Execution Example

- Partition



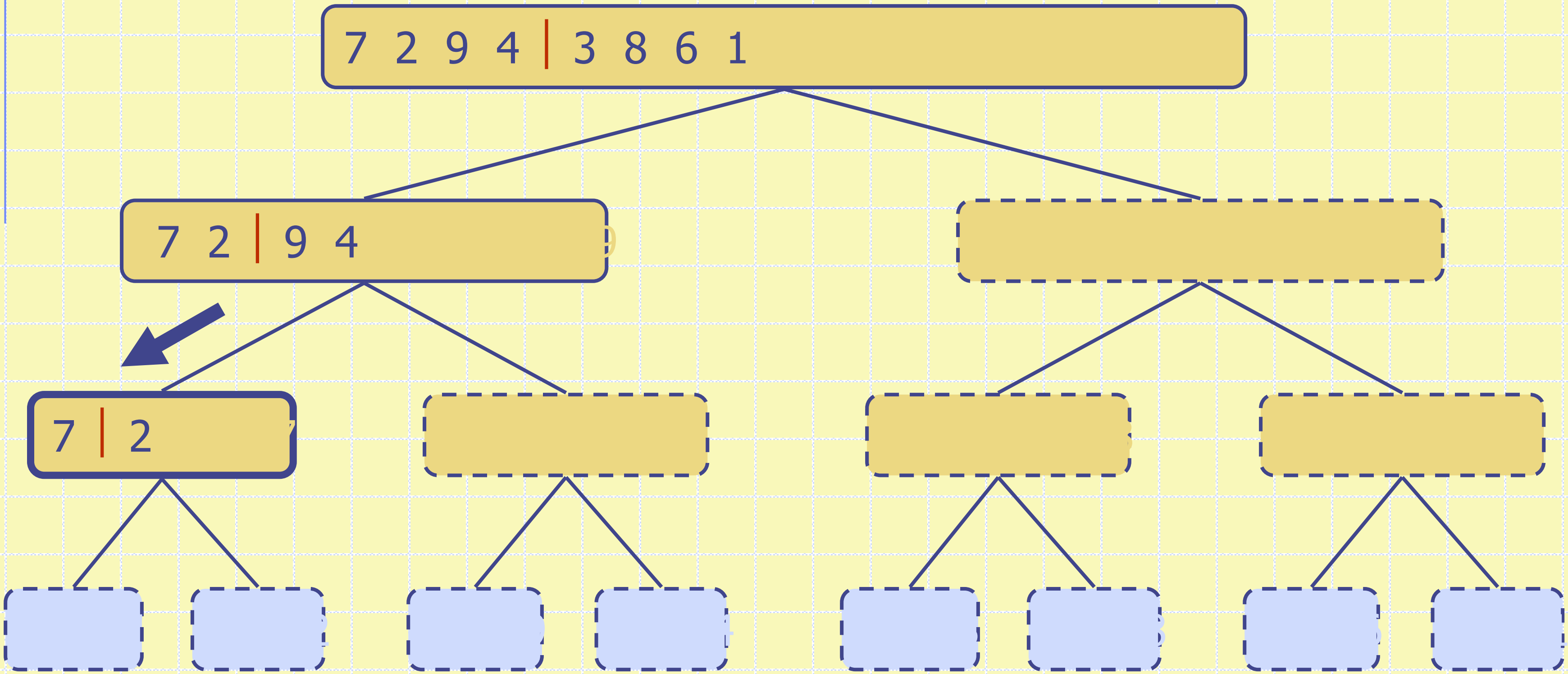
Execution Example (cont.)

- Recursive call, partition



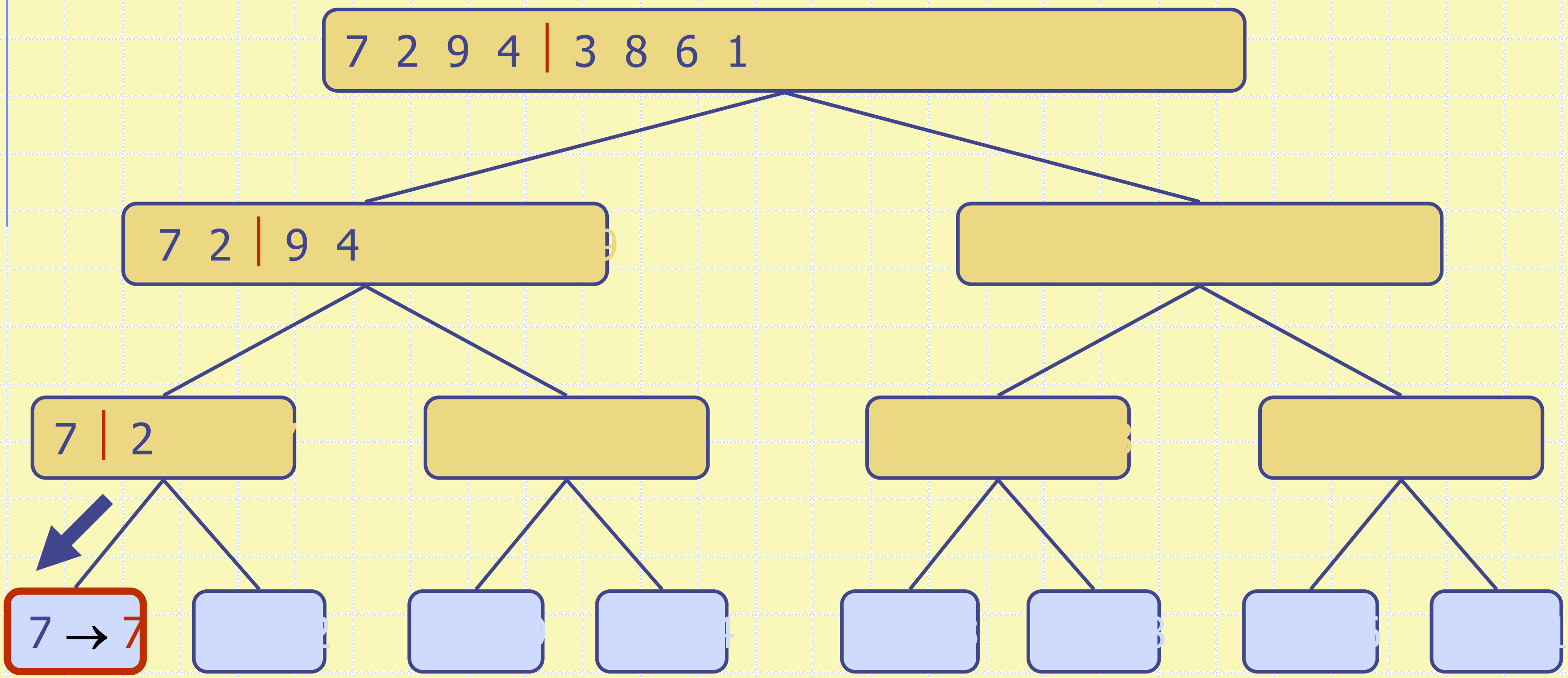
Execution Example (cont.)

- Recursive call, partition



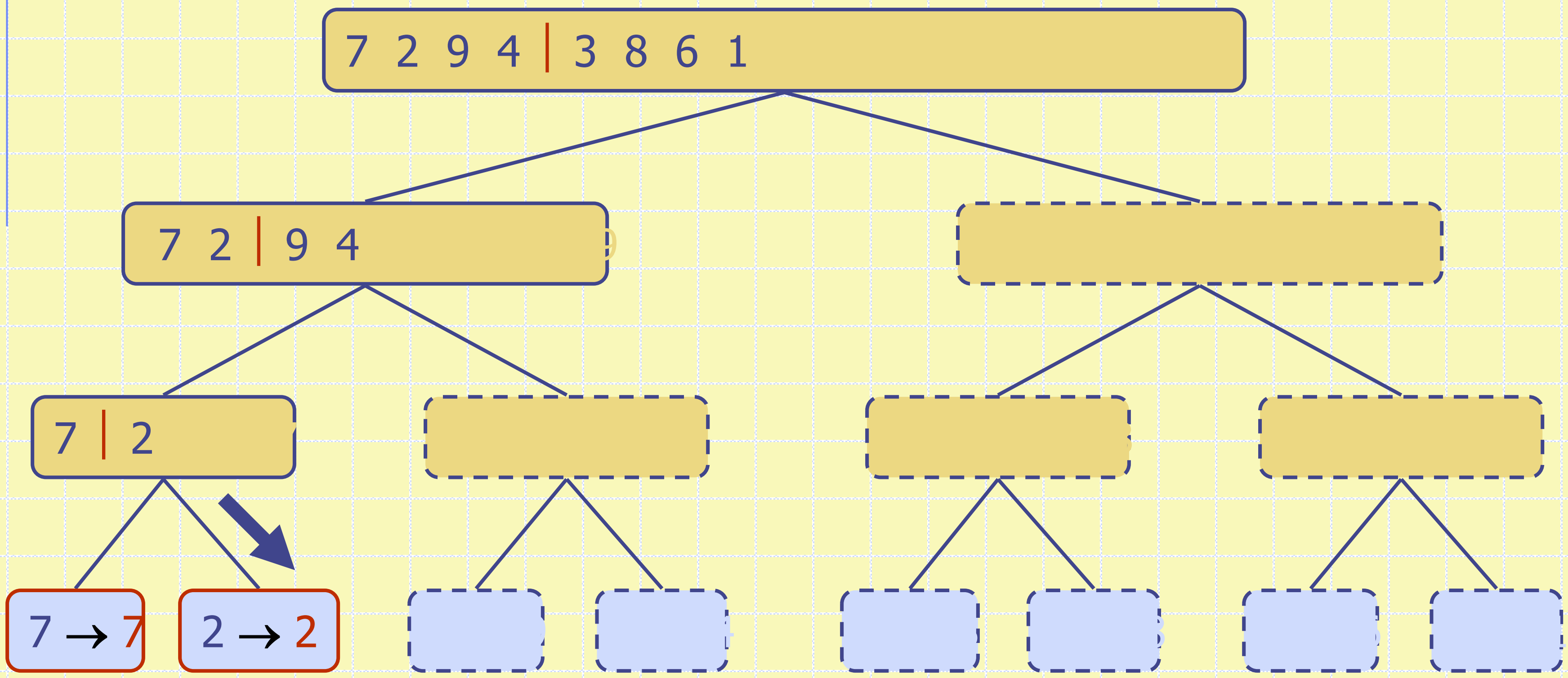
Execution Example (cont.)

- Recursive call, base case



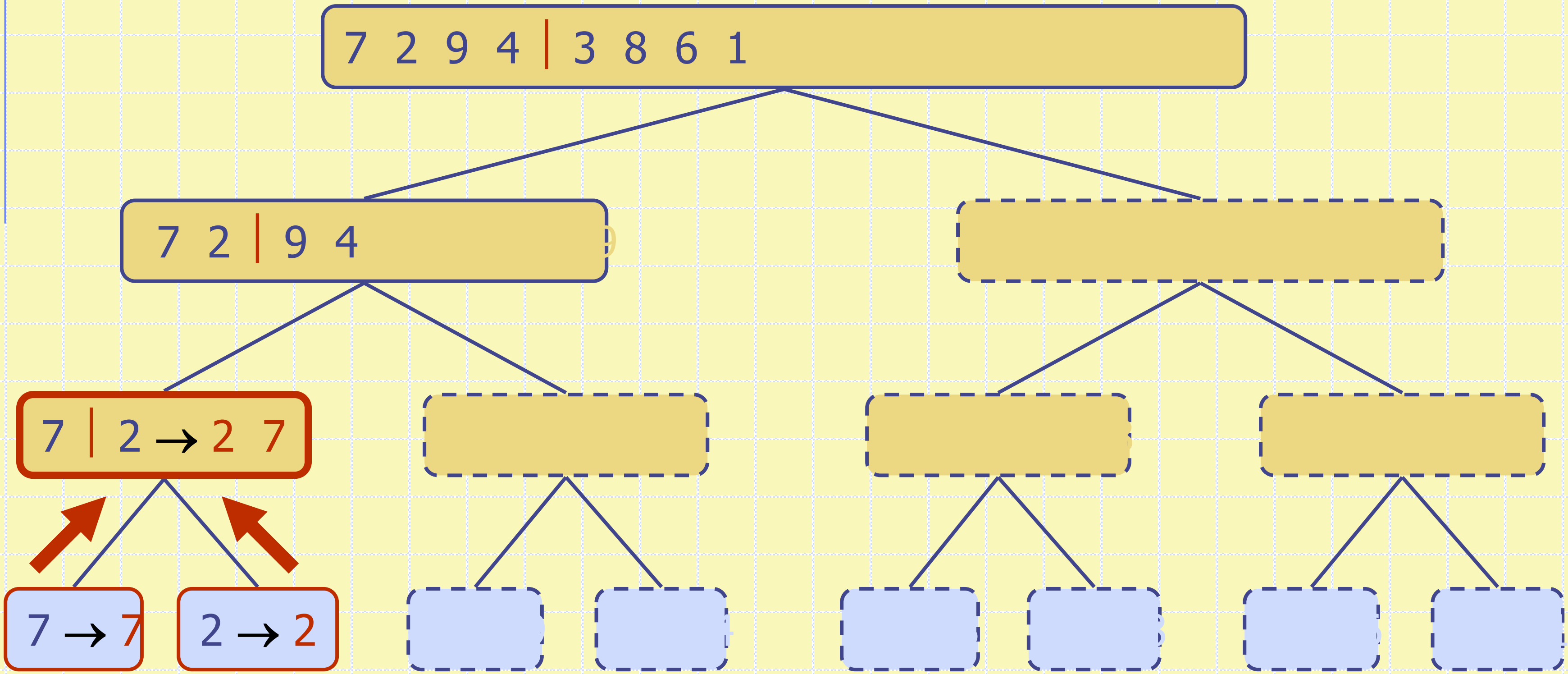
Execution Example (cont.)

- Recursive call, base case



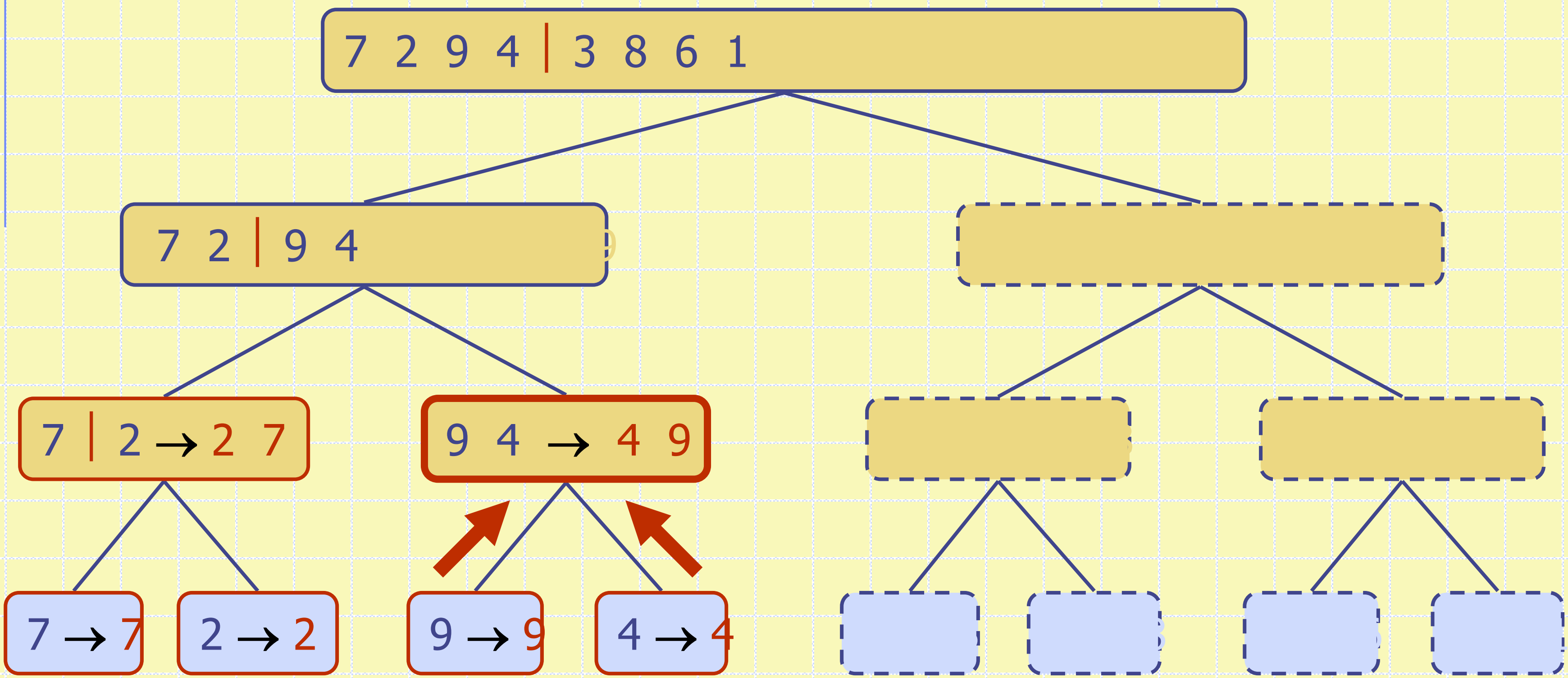
Execution Example (cont.)

- Merge



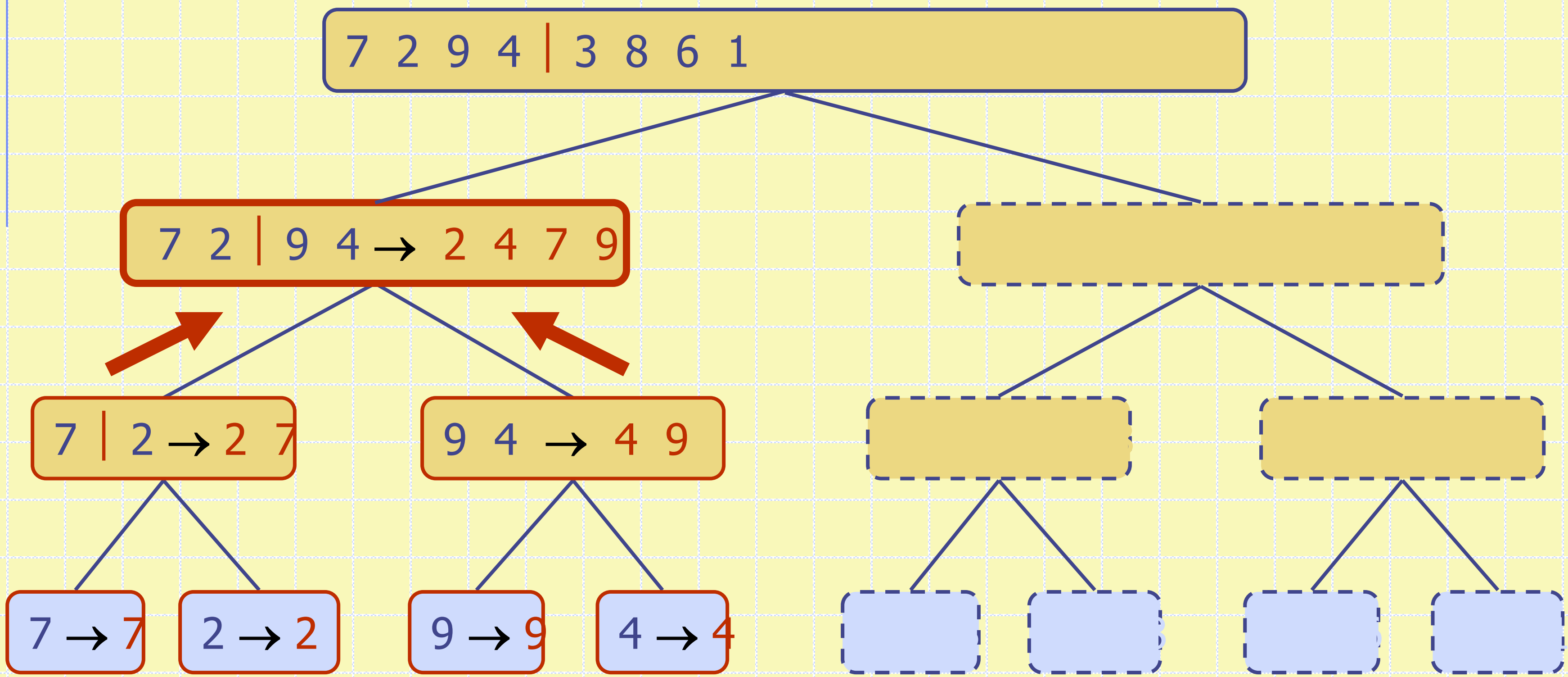
Execution Example (cont.)

- Recursive call, …, base case, merge



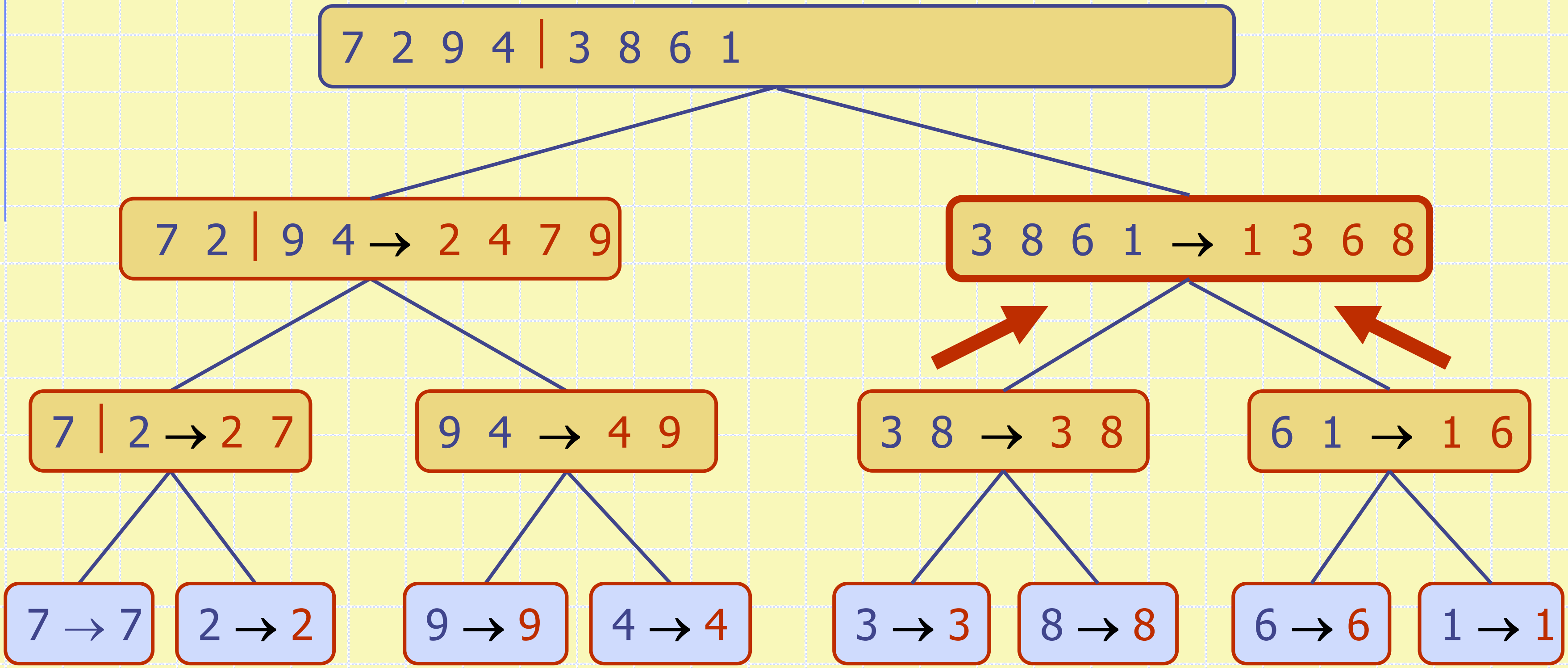
Execution Example (cont.)

- Merge



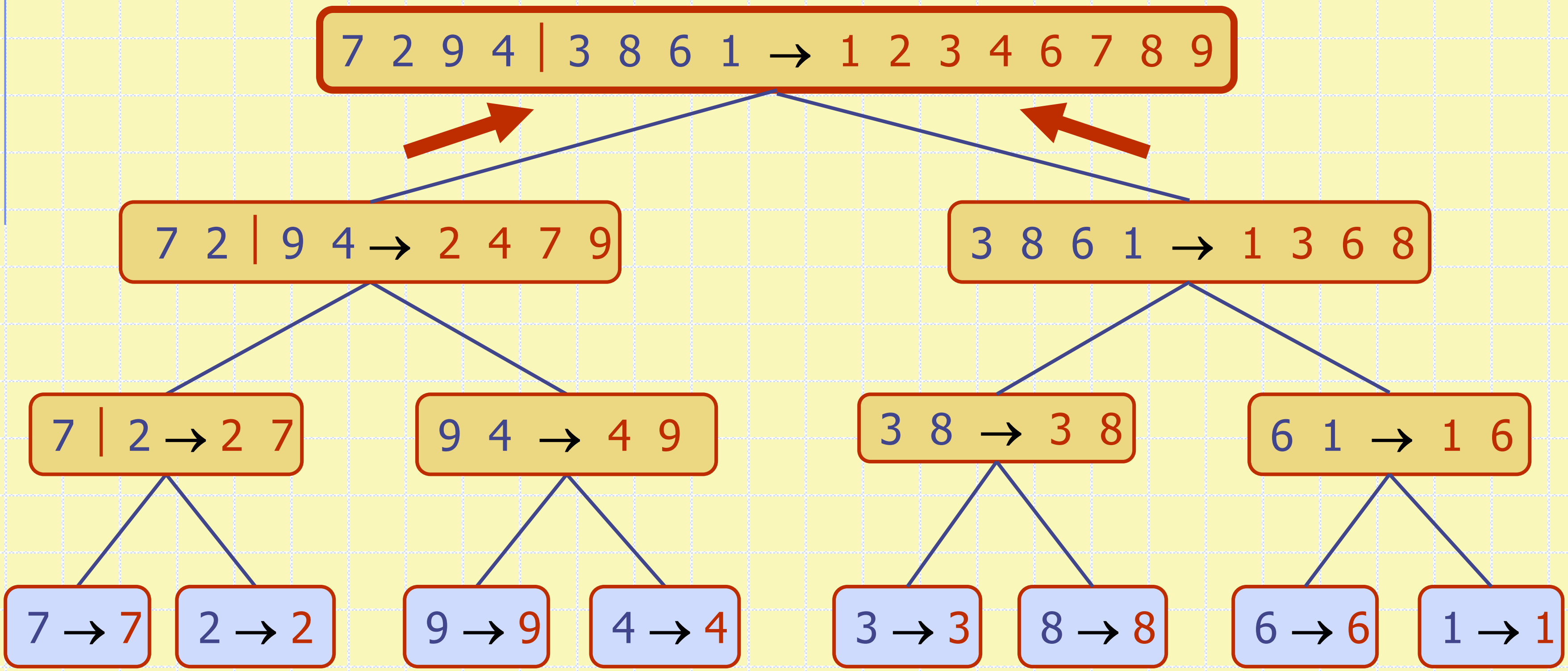
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



The Cost of Sorting Two Sorted Arrays

- The algorithm to merge 2 sorted arrays (possibly of different sizes) can be as follows:

Algorithm *merge*(*A1, A2, A*)
    Input sorted sequences *A 1* and *A2* and empty sequence *A* with sufficient size; all are implemented as arrays
    Output sorted sequence *A* containing $A1 \cup A2$
    $i \leftarrow j \leftarrow 0$
    while $i < A1.size() \wedge j < A2.size()$ **do**
        if *A1.get*(i) <= *A2.get*(j) **then**
            S.addLast(*A1.get*(i))
            $i \leftarrow i + 1$
        else
            S.addLast(*A2.get*(j))
            $j \leftarrow j + 1$
    while $i < A1.size()$ **do**
            S.addLast(*A1.get*(i))
            $i \leftarrow i + 1$
    while $j < A2.size()$ **do**
            S.addLast(*A2.get*(j))
            $j \leftarrow j + 1$

The Cost of Sorting Two Sorted Arrays

- Example

A1

0	1	2	3	4	5	6
2	**5**	**8**	**11**	**12**	**14**	**15**

i

A2

0	1	2	3	4	5	6
3	**9**	**10**	**18**	**19**	**22**	**25**

j

A

0	1	2	3	4	5	6	0	7	8	9	10	11	12
2	**3**	**5**	**8**	**9**									

- We compare the two current elements at the head of the two arrays (which are pointed by ***i*** & ***j***) then insert the smaller one in the final array.
- Hence, actual cost is ***O(n1)*** + ***O(n2)*** , where ***A1*** has ***n1*** elements and ***A2*** has ***n2*** elements ➔ total cost is hence ***O(n)***.

The Cost of Sorting Two Sorted Lists

- The algorithm is quite similar to the one for arrays (see Page 5 of these slides).

- Simply:
  - As long as the two lists are not empty, compare the two entries pointed by the head of the lists,
  - Pickup the smaller one and insert it at the tail/end of the new list; remove this item afterwards
  - If any of the two lists is still not empty, iterate on it and insert its remaining items at the tail of the

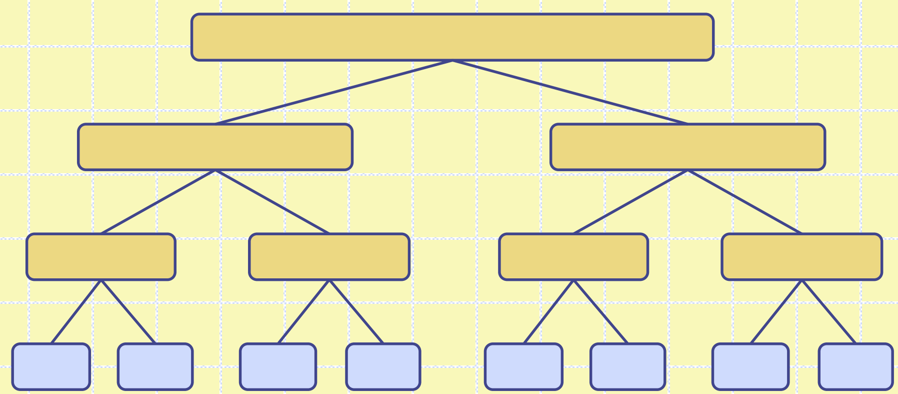
- Again, the actual cost is $O(n1) + O(n2)$, where $n1$ and $n2$ are the number of elements in the two lists.

- Consequently, total cost to sort the two sorted lists is $O(n)$.

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 or 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	# of sequences	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Merge-Sort vs. Heap-Sort

- Like heap-sort
  - It uses a comparator
  - It has $O(n \log n)$ running time.
    - The cost to sort the elements at each level is $O(n)$ and we do that $\log n$ times.
- 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)

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
insertion-sort	$O(n^2)$	▪ slow ▪ in-place ▪ for small data sets (< 1K)
heap-sort	$O(n \log n)$	▪ fast ▪ in-place ▪ for large data sets (1K — 1M)
merge-sort	$O(n \log n)$	▪ fast ▪ sequential data access ▪ for huge data sets (> 1M)