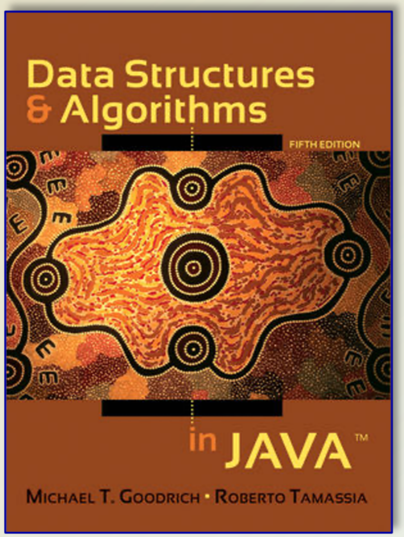
Data Structure & Algorithms in JAVA

5th edition

Michael T. Goodrich
Roberto Tamassia



Chapter 11: Sorting, Sets, and Selection

CPSC 3200

Algorithm Analysis and Advanced Data Structure

Chapter Topics

- Insertion Sort.
- Selection Sort.
- Bubble Sort.
- Heap Sort.
- Merge-sort.
- Quick-sort.

Insertion Sort

Algorithm InsertionSort(A):

Input: An array A of n comparable elements

Output: The array A with elements rearranged in nondecreasing order

```
for i ← 1 to n−1 do
    {Insert A[i] at its proper location in A[0],A[1],...,A[i−1]}
    cur ← A[i]
    j ← i−1
    while j ≥ 0 and a[j] > cur do
        A[j+1] ← A[j]
        j ← j−1
    A[j+1] ← cur {cur is now in the right place}
```

Selection Sort

Algorithm SelectionSort(A)

Input: An array A of n comparable elements

Output: The array A with elements rearranged in nondecreasing order

n := length[A]

for i ← 1 to n **do**

j ← FindIndexOfSmallest(A, i, n)

swap A[i] with A[j]

retrun A

Algorithm FindIndexOfSmallest(A, i, n)

smallestAt ← i

for j ← (i+1) to n **do**

if (A[j] < A[smallestAt])

smallestAt ← j

return smallestAt

Bubble Sort

Algorithm BubbleSort(A)

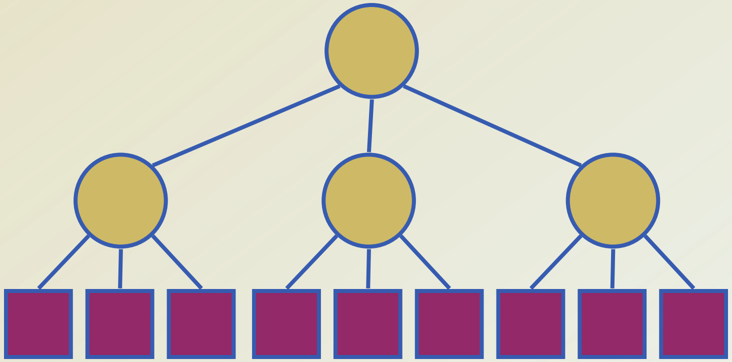
Input: An array A of n comparable elements

Output: The array A with elements rearranged in nondecreasing order

```
for i ← 0 to N - 2 do
    for J ← 0 to N - 2  do
        if (A( J ) > A( J + 1 ) then
            temp ← A( J )
            A( J ) ← A( J + 1 )
            A( J + 1 ) ← temp
return A
```

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - **Divide:** divide the input data $\boldsymbol{S}$ in two or more disjoint subsets $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …
  - **Recur:** solve the subproblems recursively
  - **Conquer:** combine the solutions for $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …, into a solution for $\boldsymbol{S}$
- The base case for the recursion are subproblems of constant size.
- Analysis can be done using recurrence equations.



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 $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1, C)$
    $mergeSort(S_2, C)$
    $S \leftarrow merge(S_1, S_2)$

Merging Two Sorted Sequences

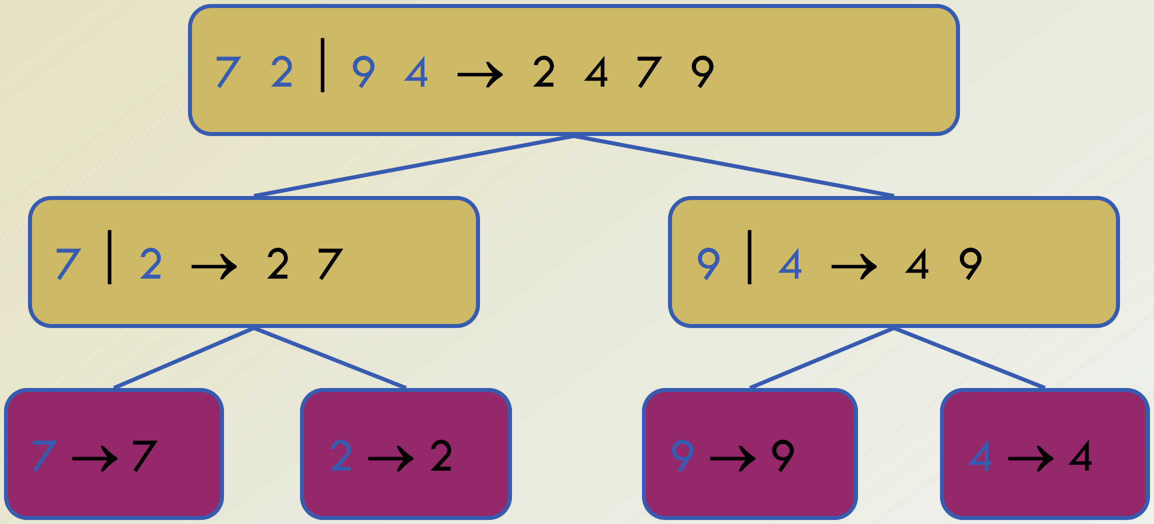
- The conquer step of merge-sort consists of merging two sorted sequences ***A*** and ***B*** into a sorted sequence ***S*** containing the union of the elements of ***A*** and ***B***
- Merging two sorted sequences, each with ***n***/2 elements and implemented by means of a doubly linked list, takes ***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 ← empty sequence
while ¬A.isEmpty() ∧ ¬B.isEmpty()
    if A.first().element() < B.first().element()
        S.addLast(A.remove(A.first()))
    else
        S.addLast(B.remove(B.first()))
while ¬A.isEmpty()
    S.addLast(A.remove(A.first()))
while ¬B.isEmpty()
    S.addLast(B.remove(B.first()))
return S
```

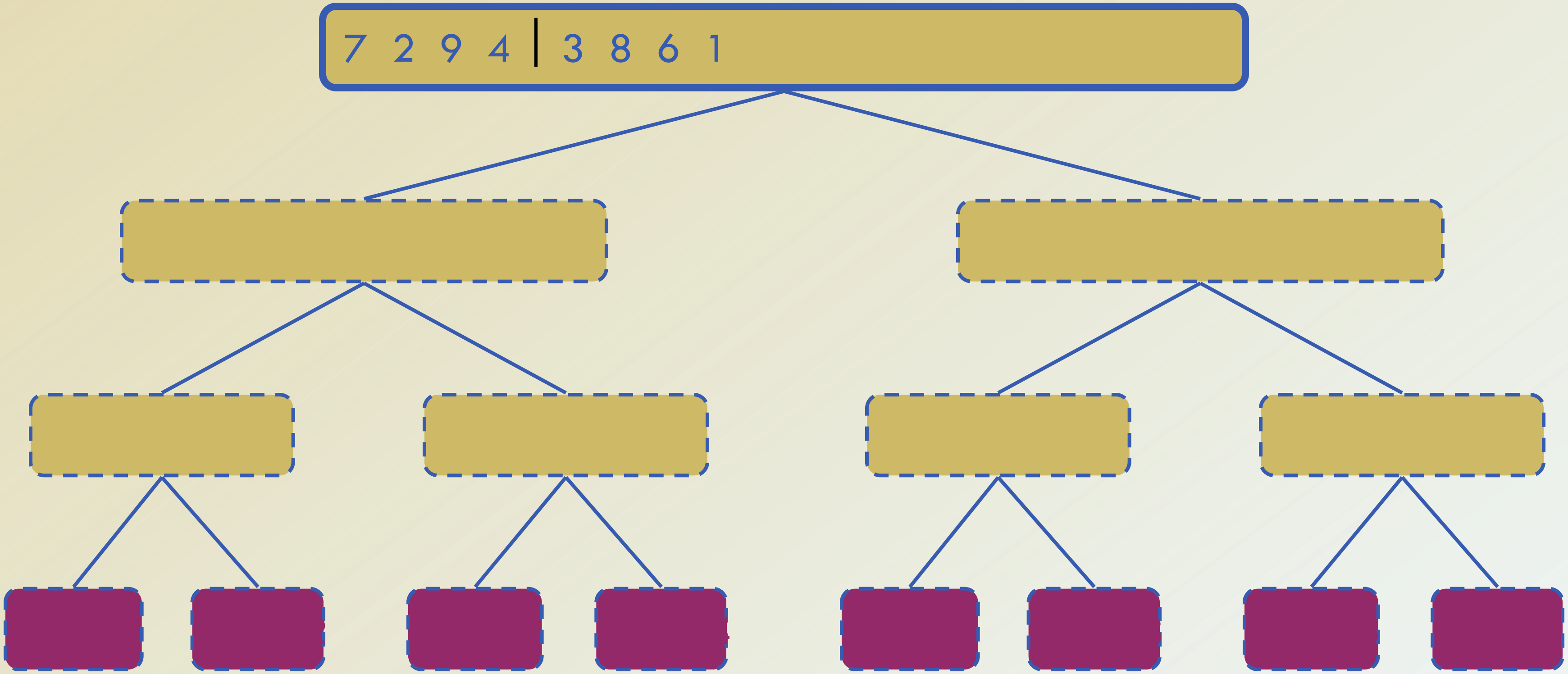
Merge-Sort Tree

- 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



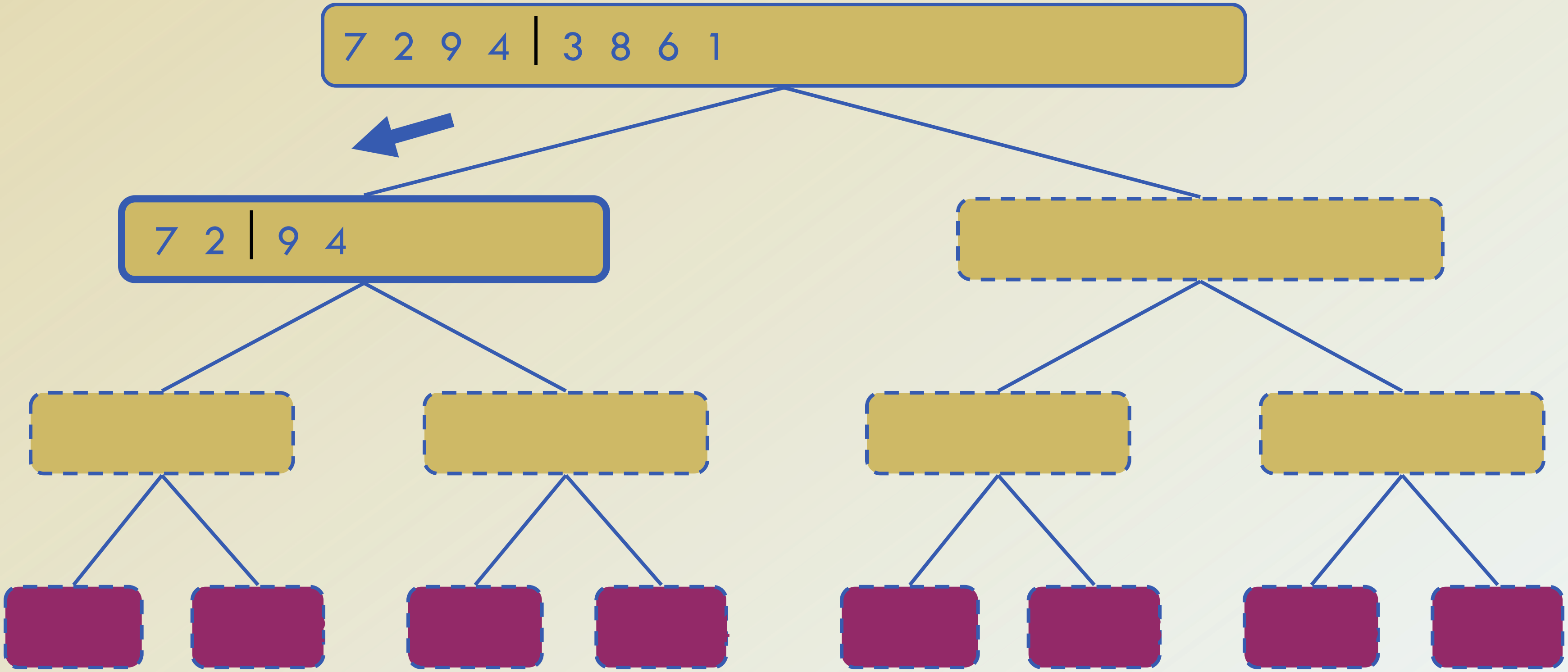
Execution Example

- Partition



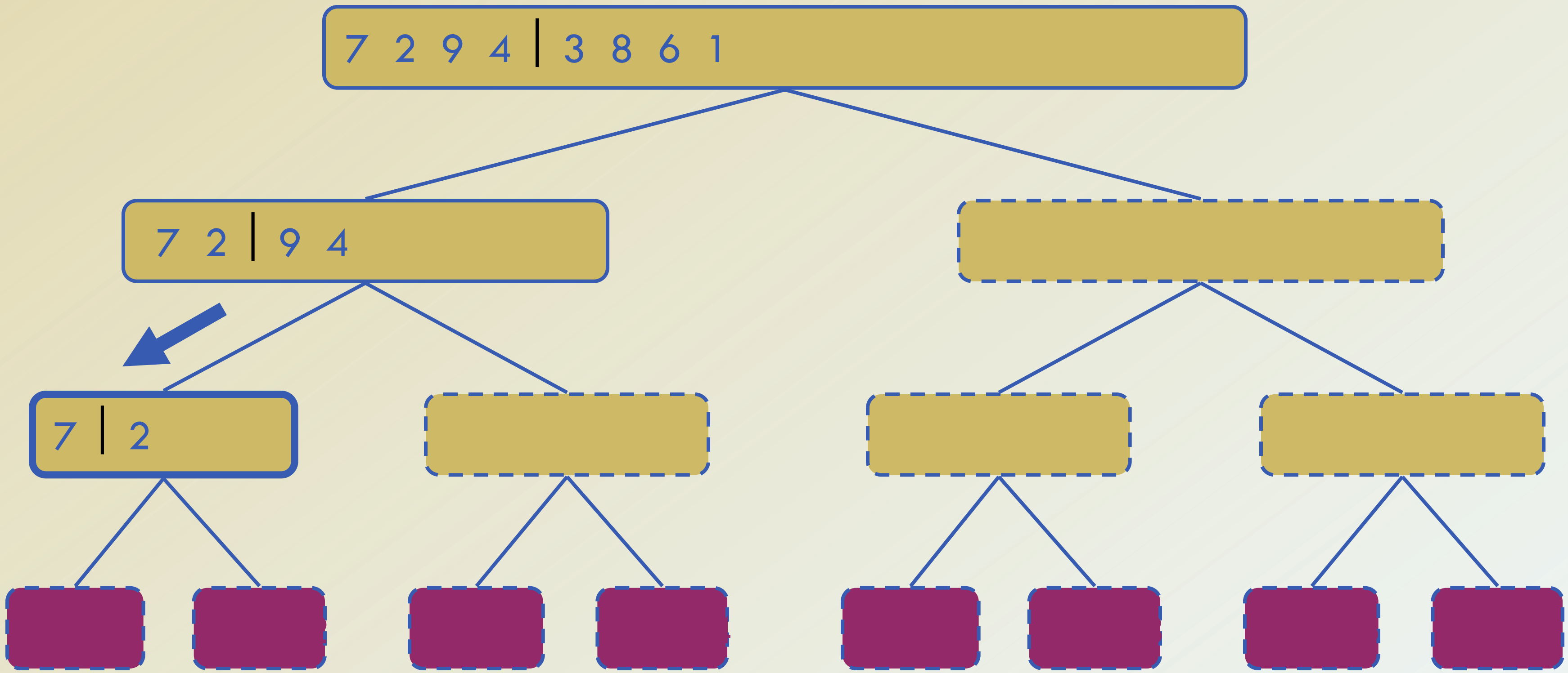
Execution Example (cont.)

- Recursive call, partition



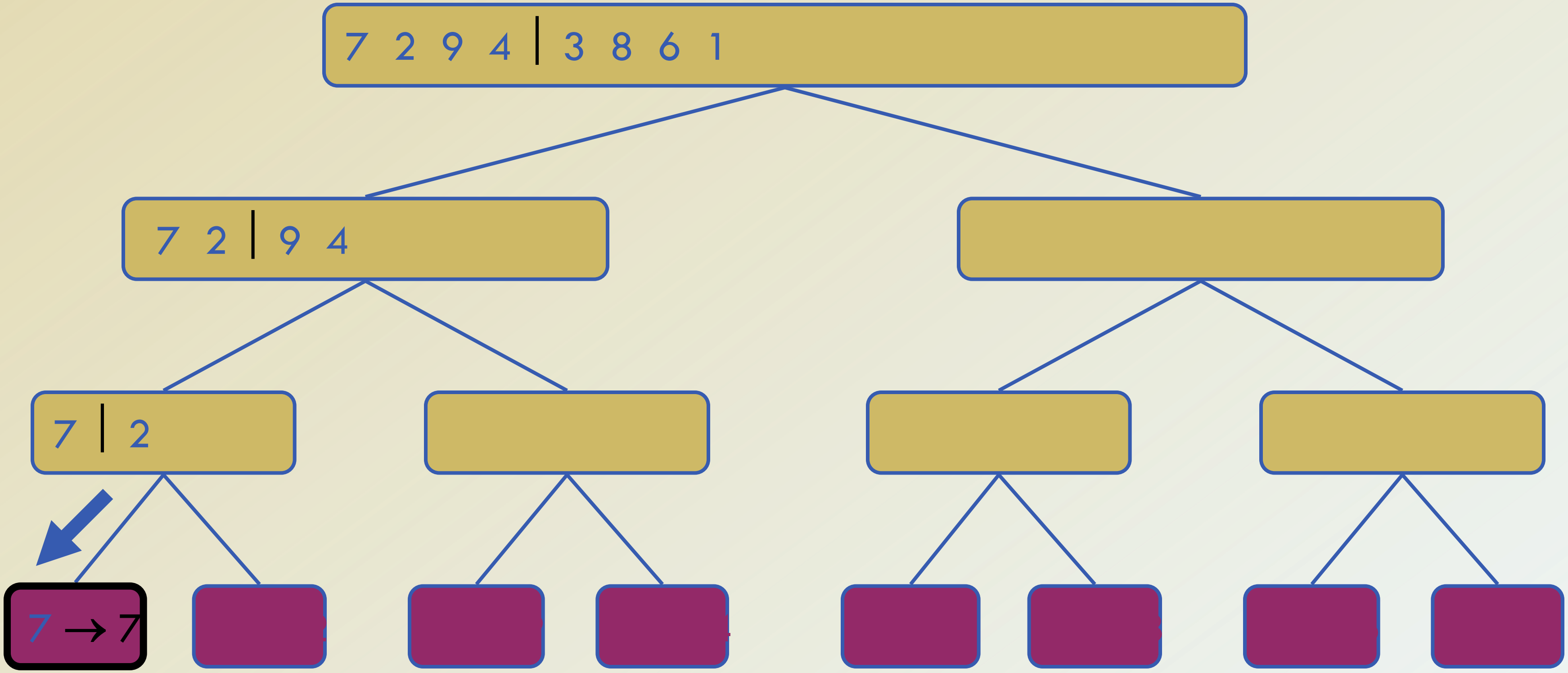
Execution Example (cont.)

- Recursive call, partition



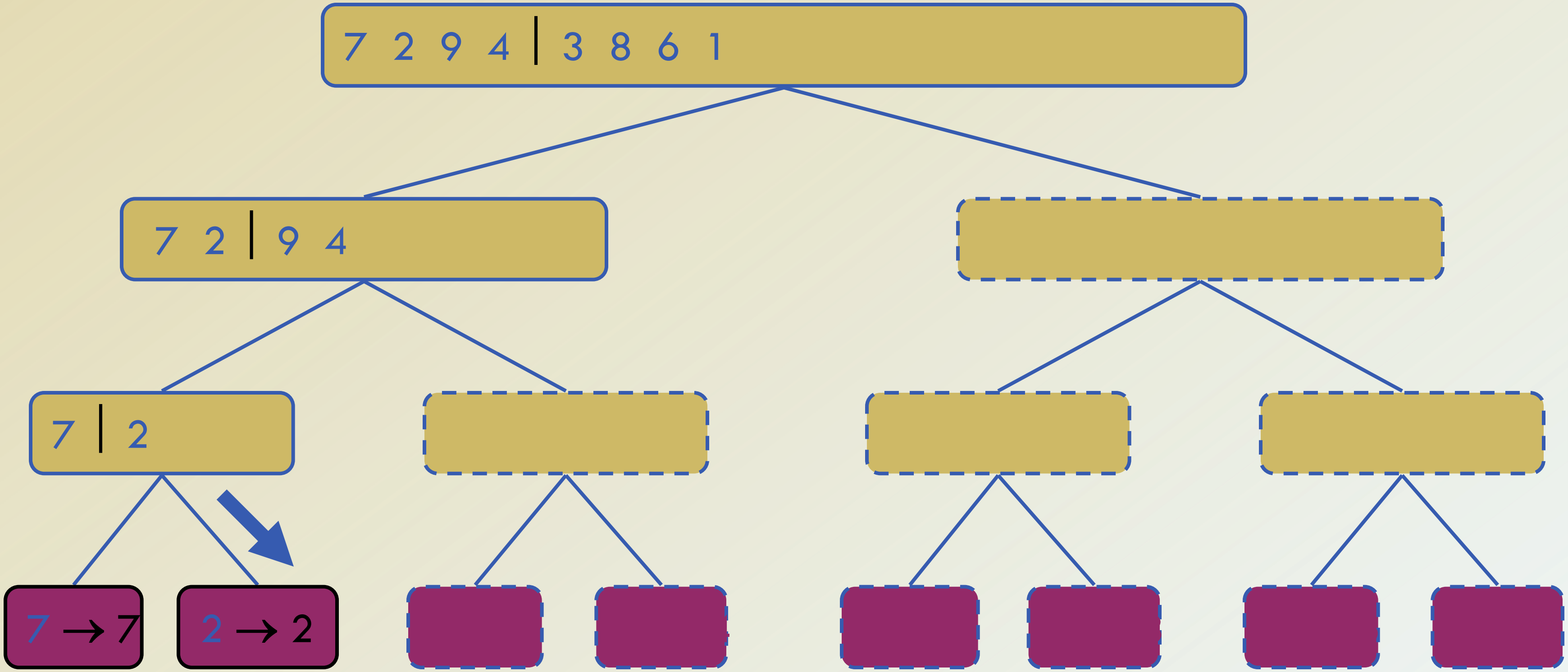
Execution Example (cont.)

- Recursive call, base case



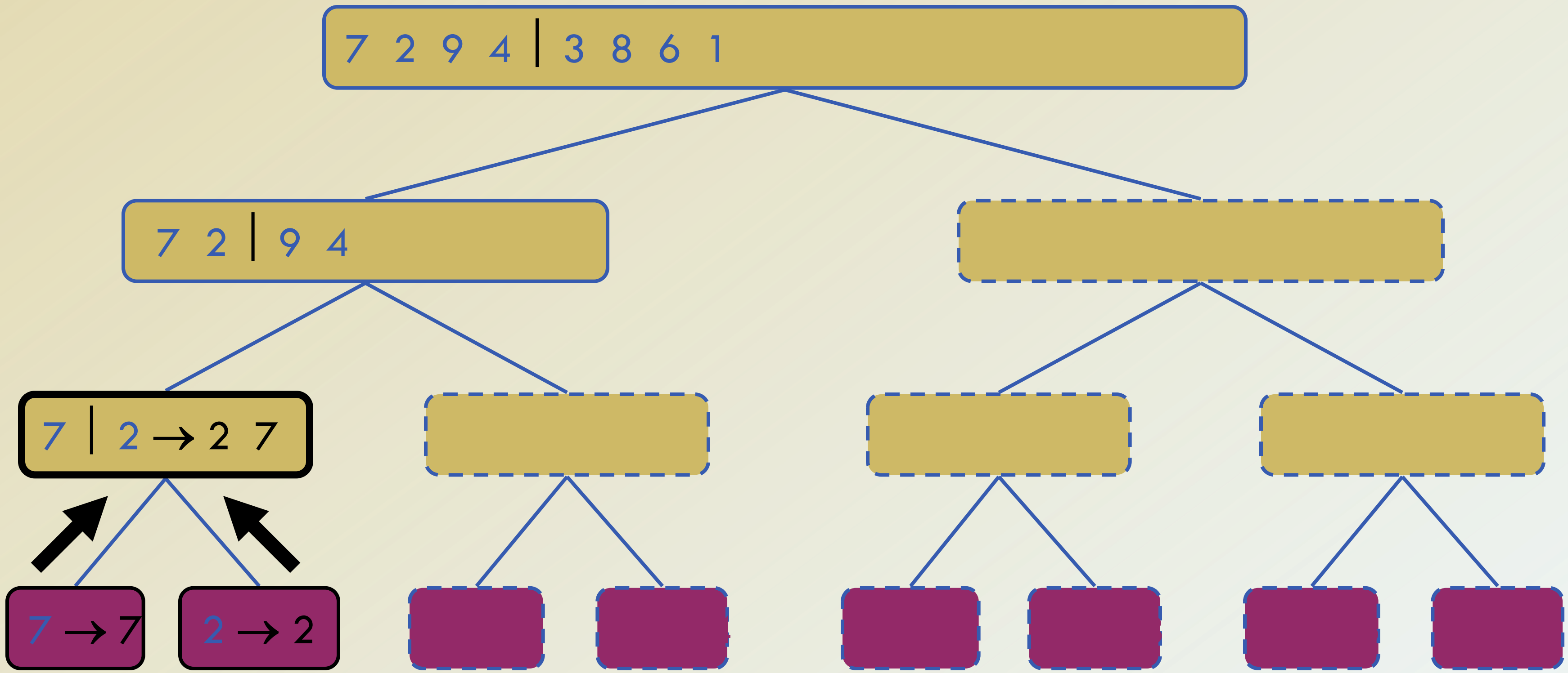
Execution Example (cont.)

- Recursive call, base case



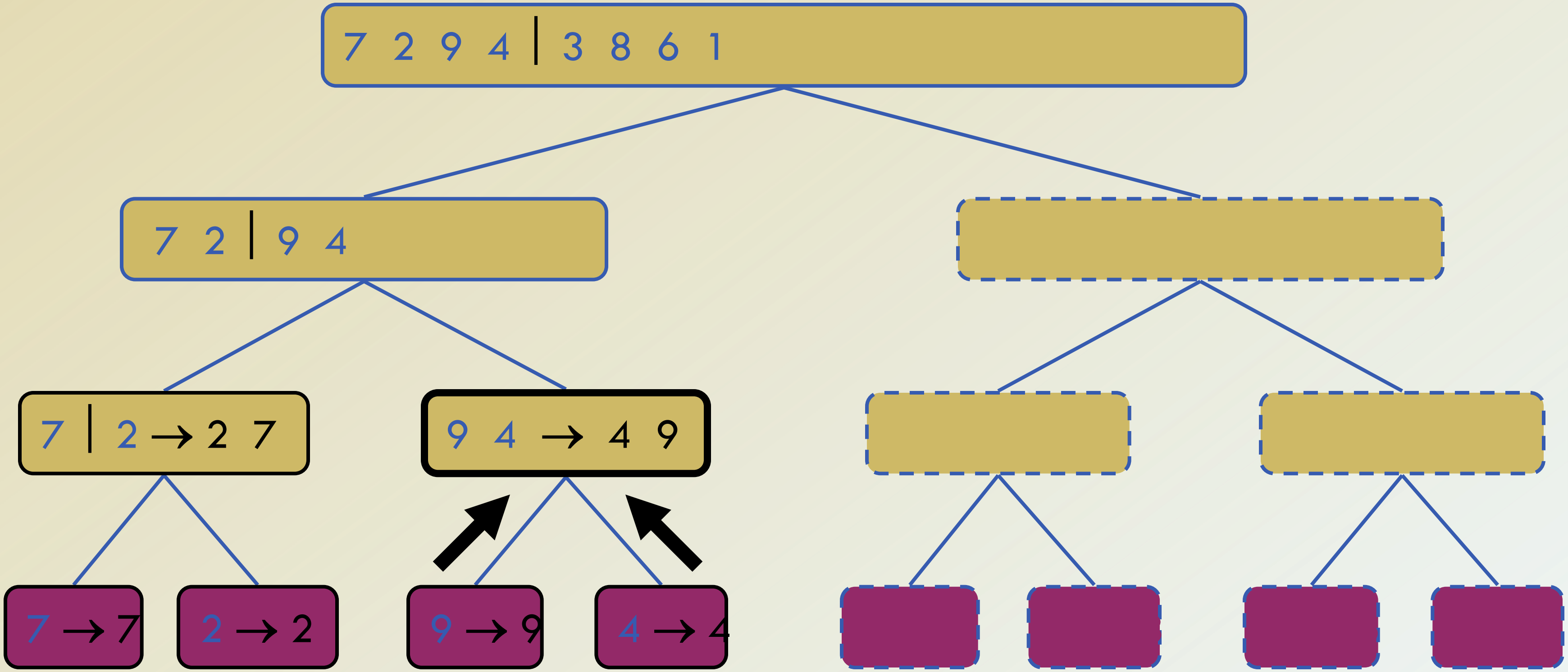
Execution Example (cont.)

- Merge



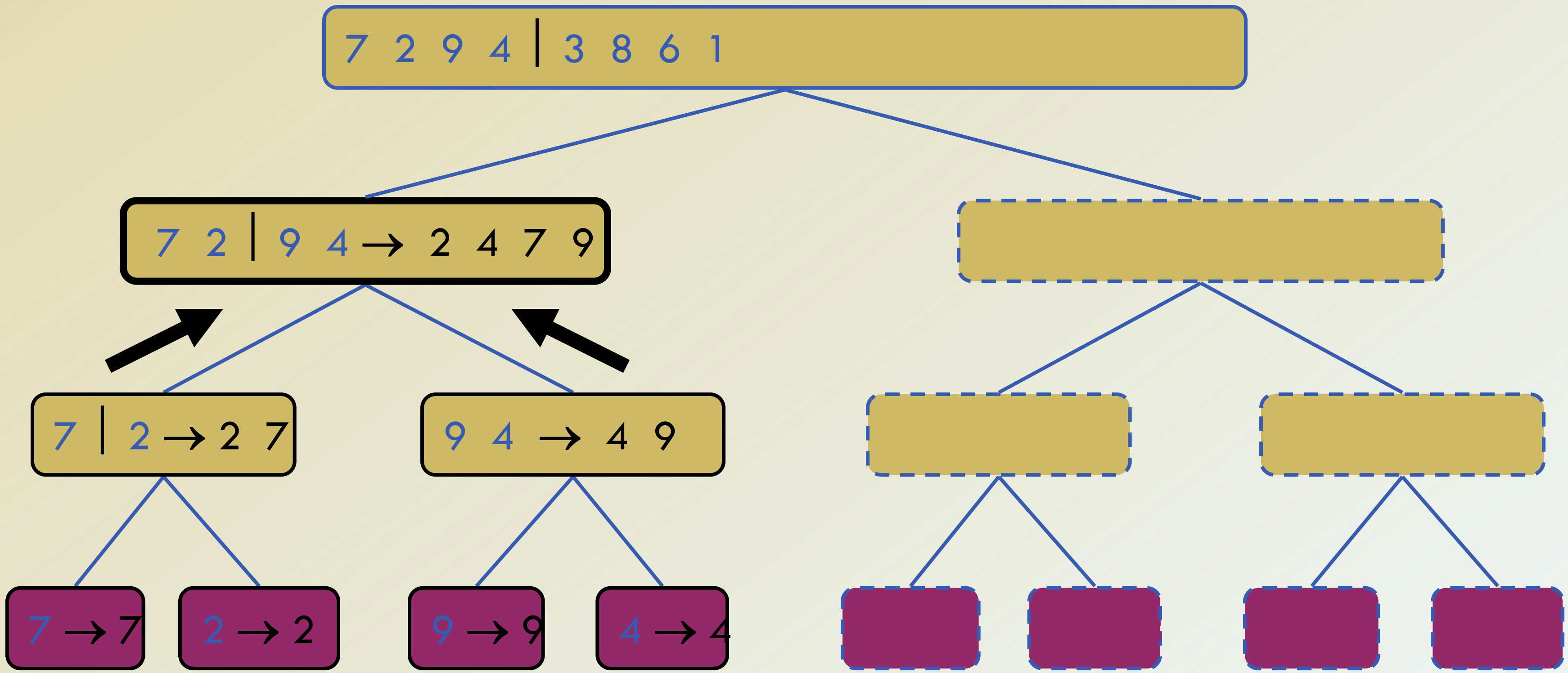
Execution Example (cont.)

- Recursive call, ..., base case, merge



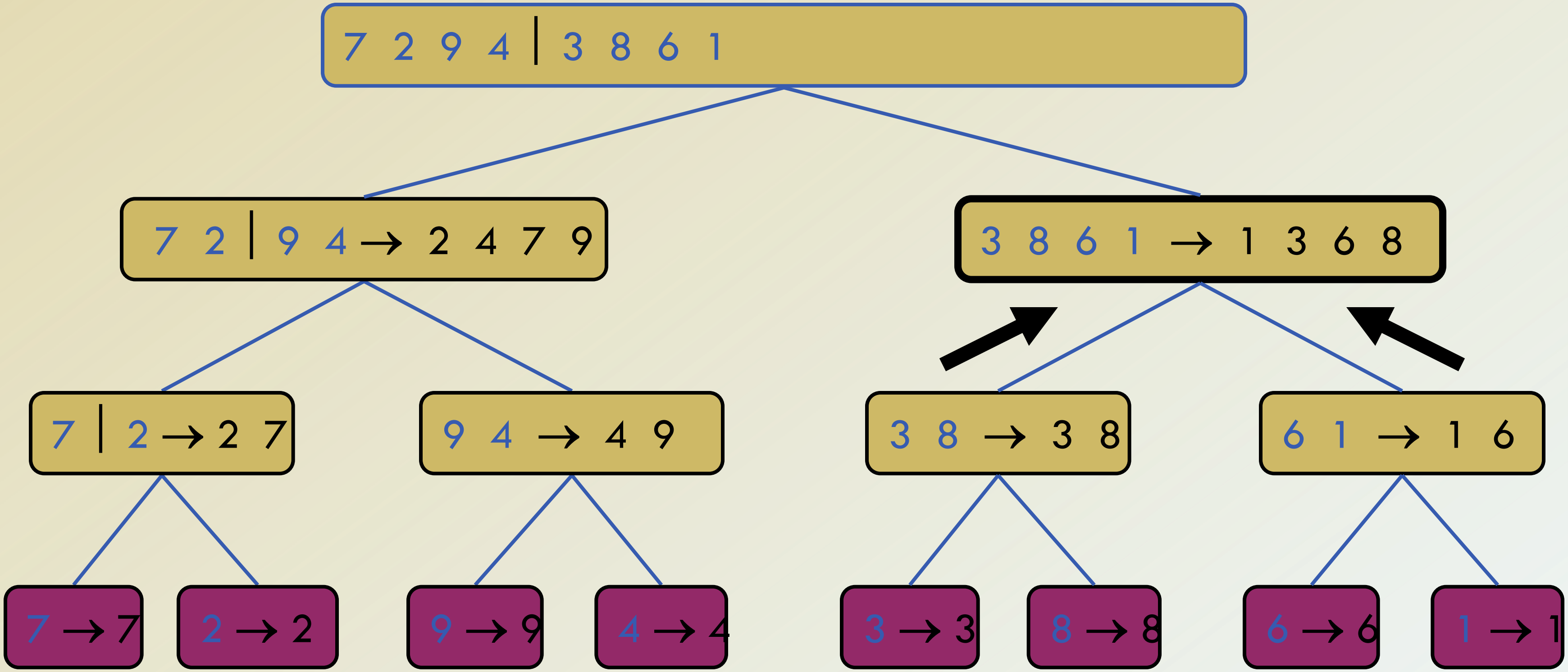
Execution Example (cont.)

- Merge



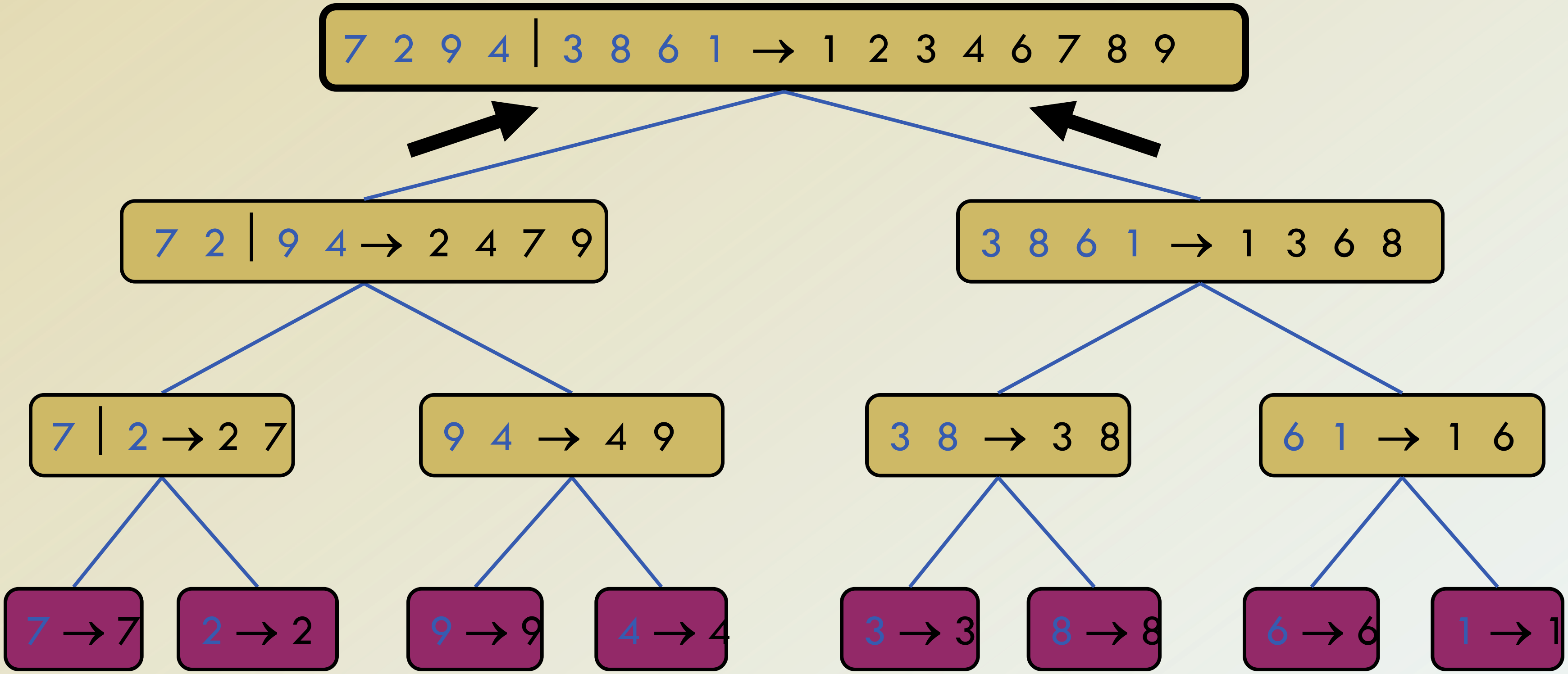
Execution Example (cont.)

- Recursive call, ..., merge, merge



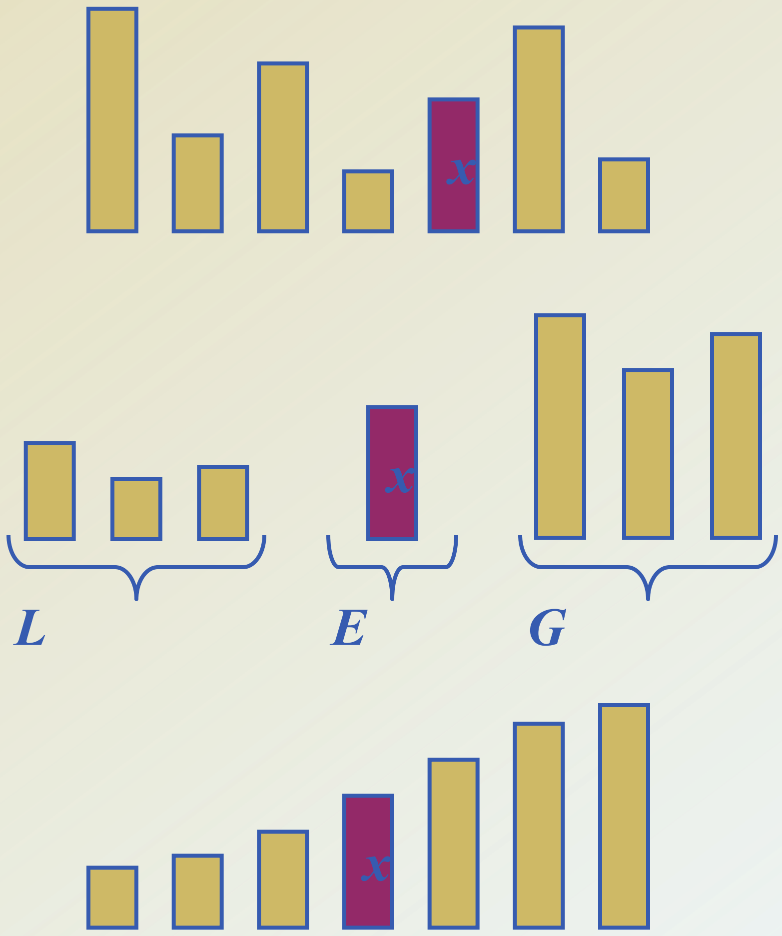
Execution Example (cont.)

- Merge



Quick-Sort

- Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:
  - **Divide:** pick a random element ***x*** (called pivot) and partition ***S*** into
    - ***L*** elements less than ***x***
    - ***E*** elements equal ***x***
    - ***G*** elements greater than ***x***
  - **Recur:** sort ***L*** and ***G***
  - **Conquer:** join ***L***, ***E*** and ***G***



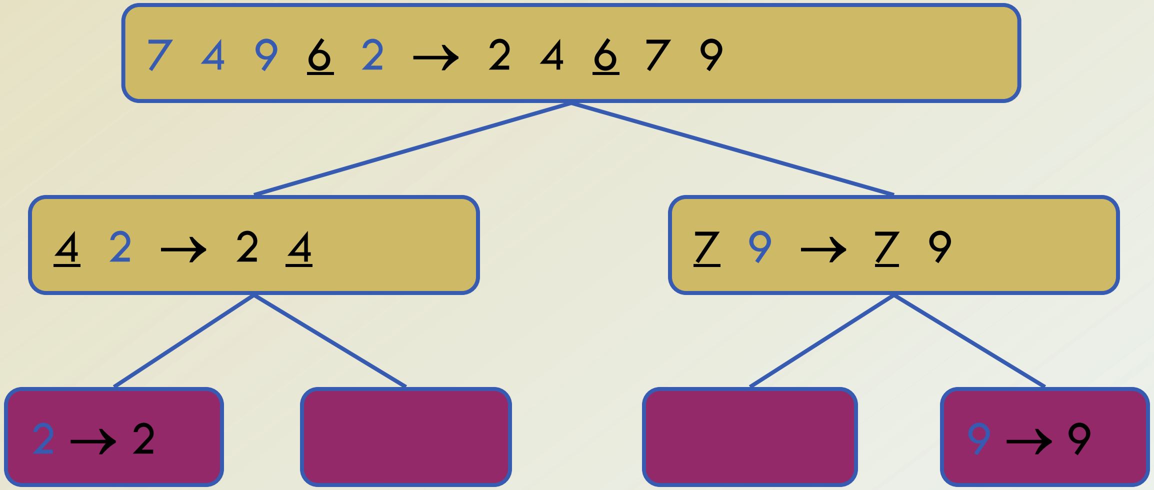
Partition

- We partition an input sequence as follows:
  - We remove, in turn, each element ***y*** from ***S*** and
  - We insert ***y*** into ***L***, ***E*** or ***G***, depending on the result of the comparison with the pivot ***x***
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes ***O***(1) time
- Thus, the partition step of quick-sort takes ***O***(***n***) time.

```
Algorithm partition(S, p)
    Input: sequence S, position p of pivot
    Output: subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.addLast(y)
        else if y = x
            E.addLast(y)
        else { y > x }
            G.addLast(y)
    return L, E, G
```

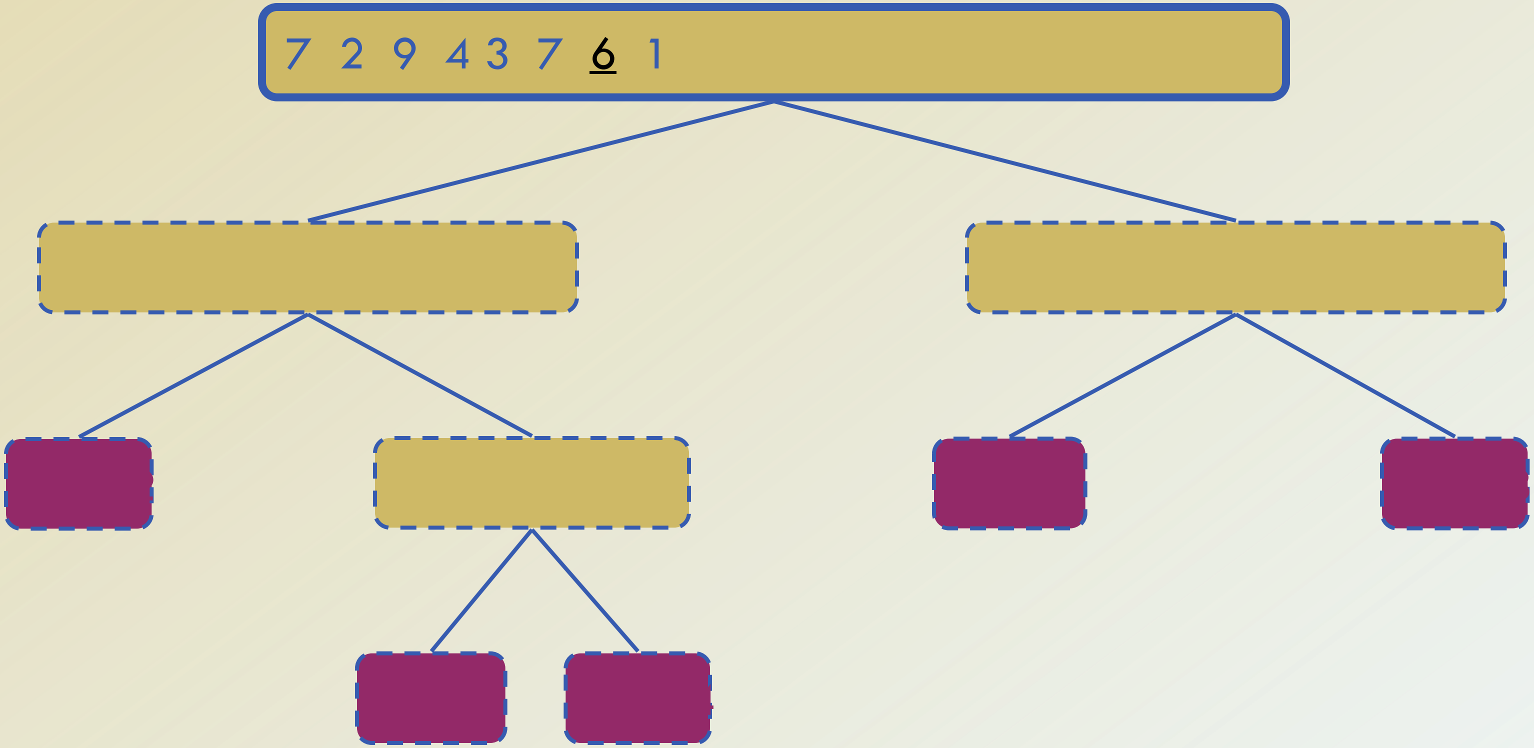
Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
  - Each node represents a recursive call of quick-sort and stores
    - Unsorted sequence before the execution and its pivot
    - 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



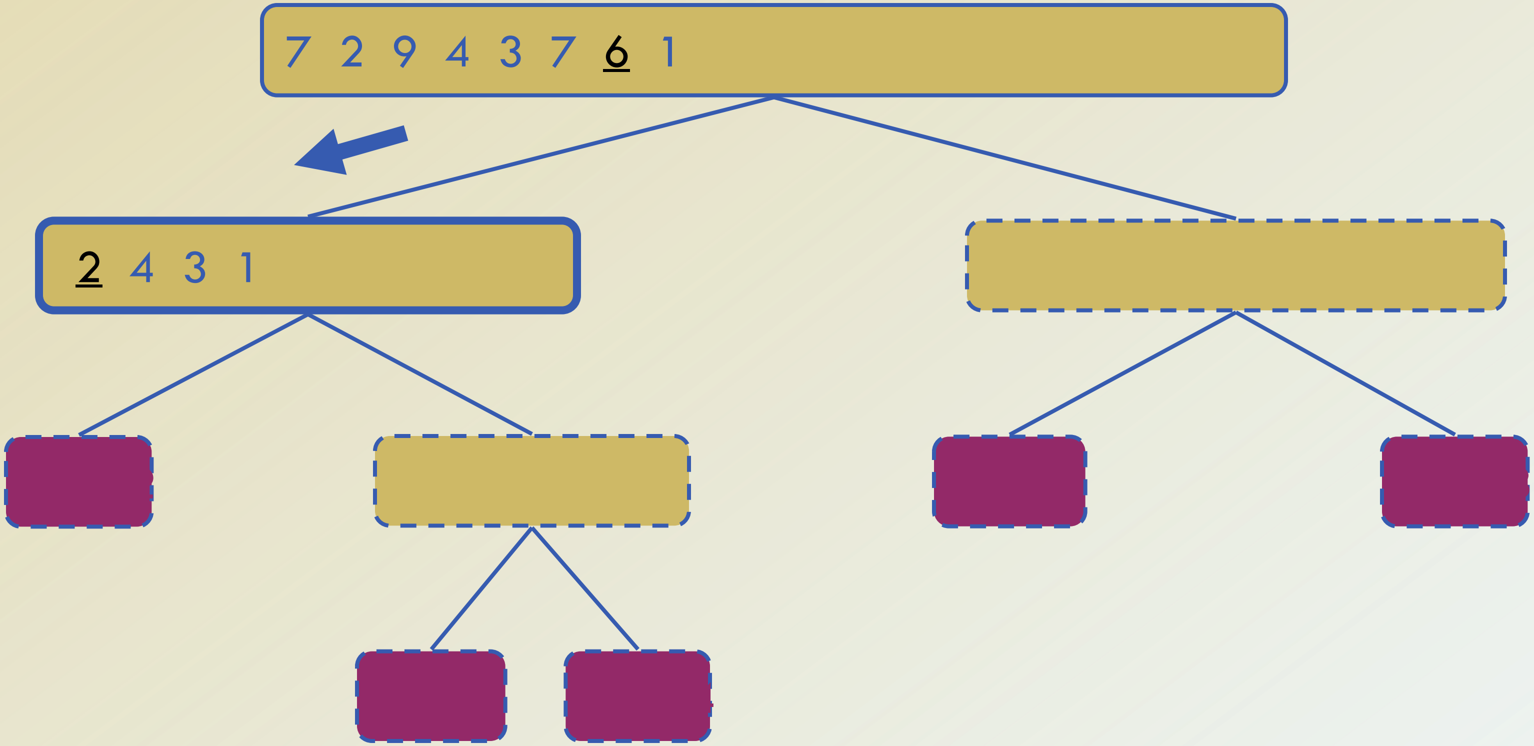
Execution Example

- Pivot selection



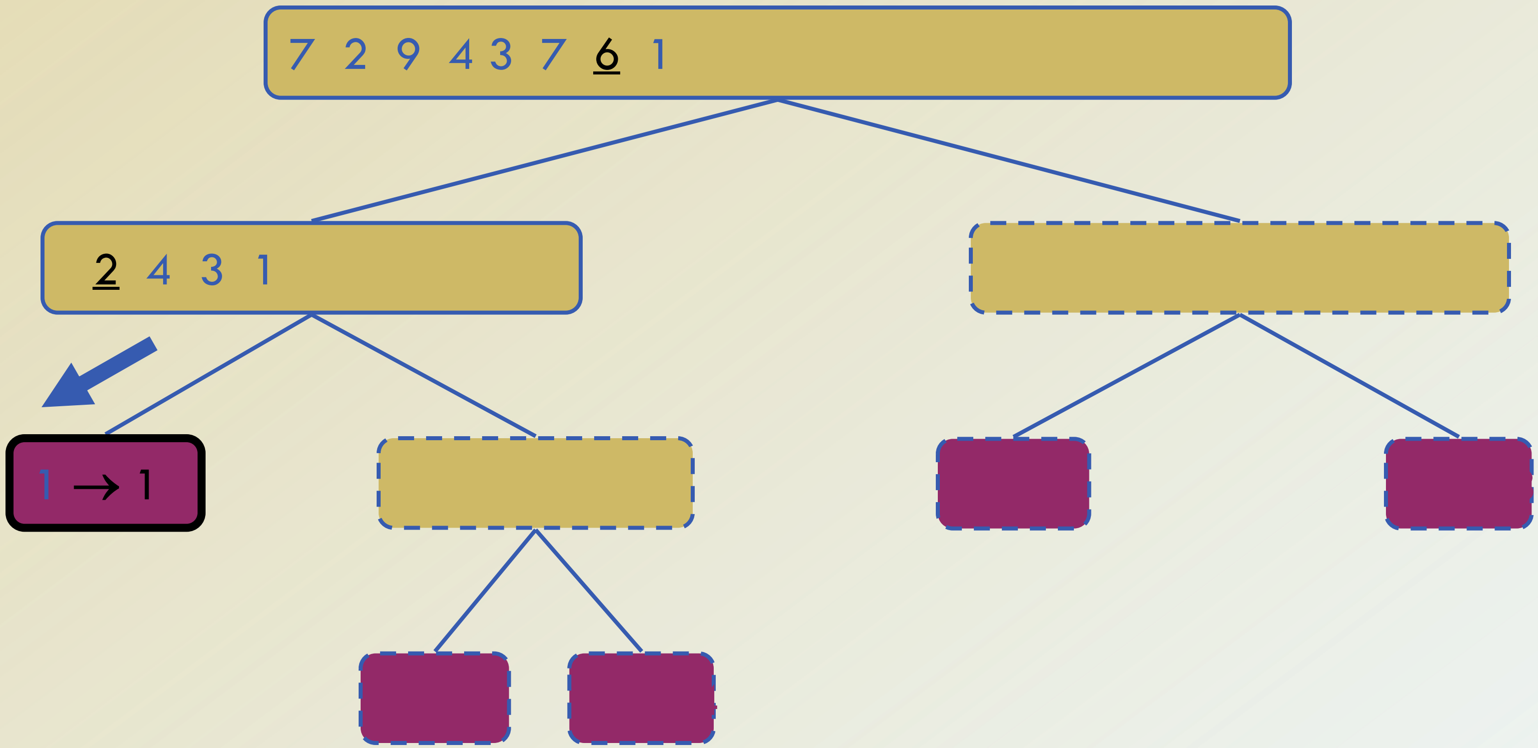
Execution Example (cont.)

- Partition, recursive call, pivot selection



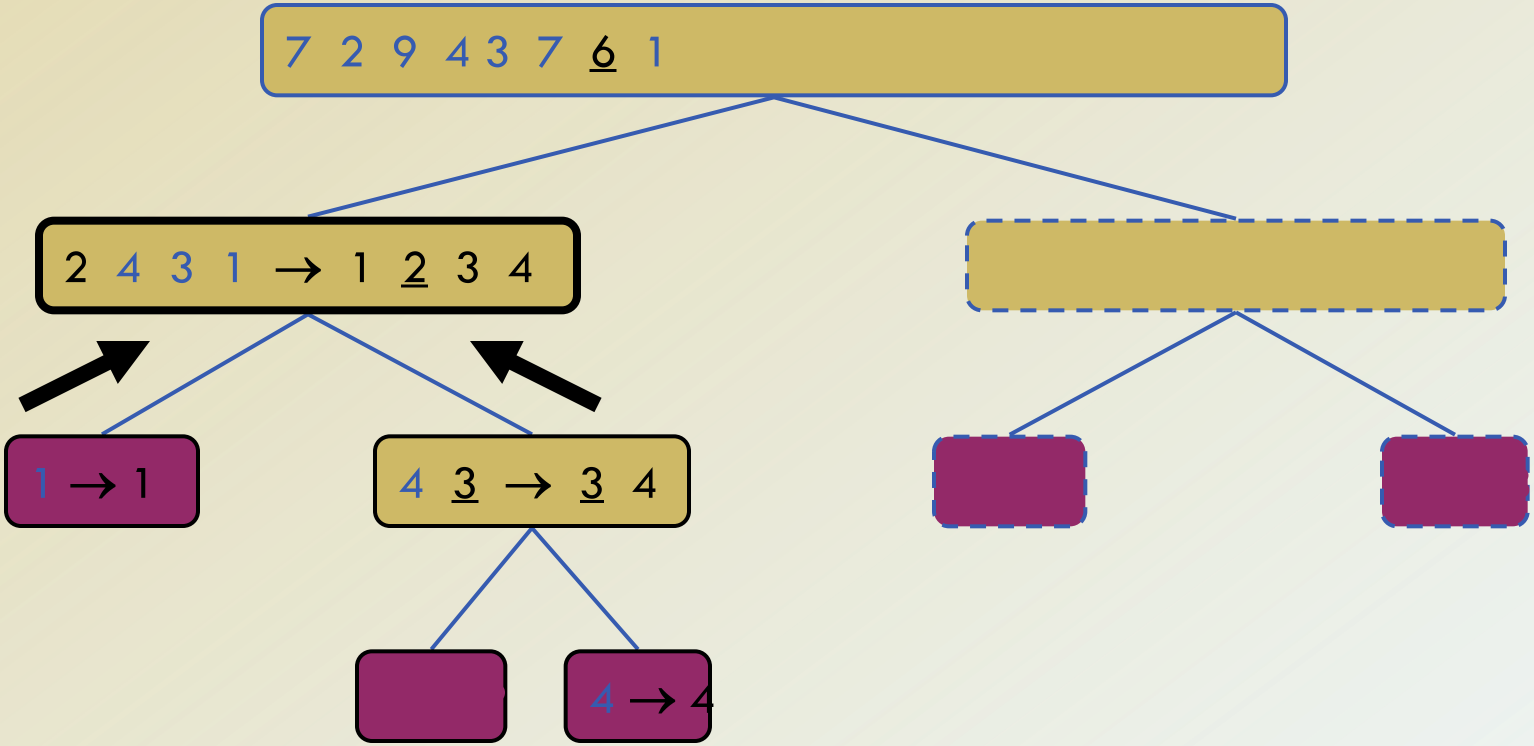
Execution Example (cont.)

- Partition, recursive call, base case



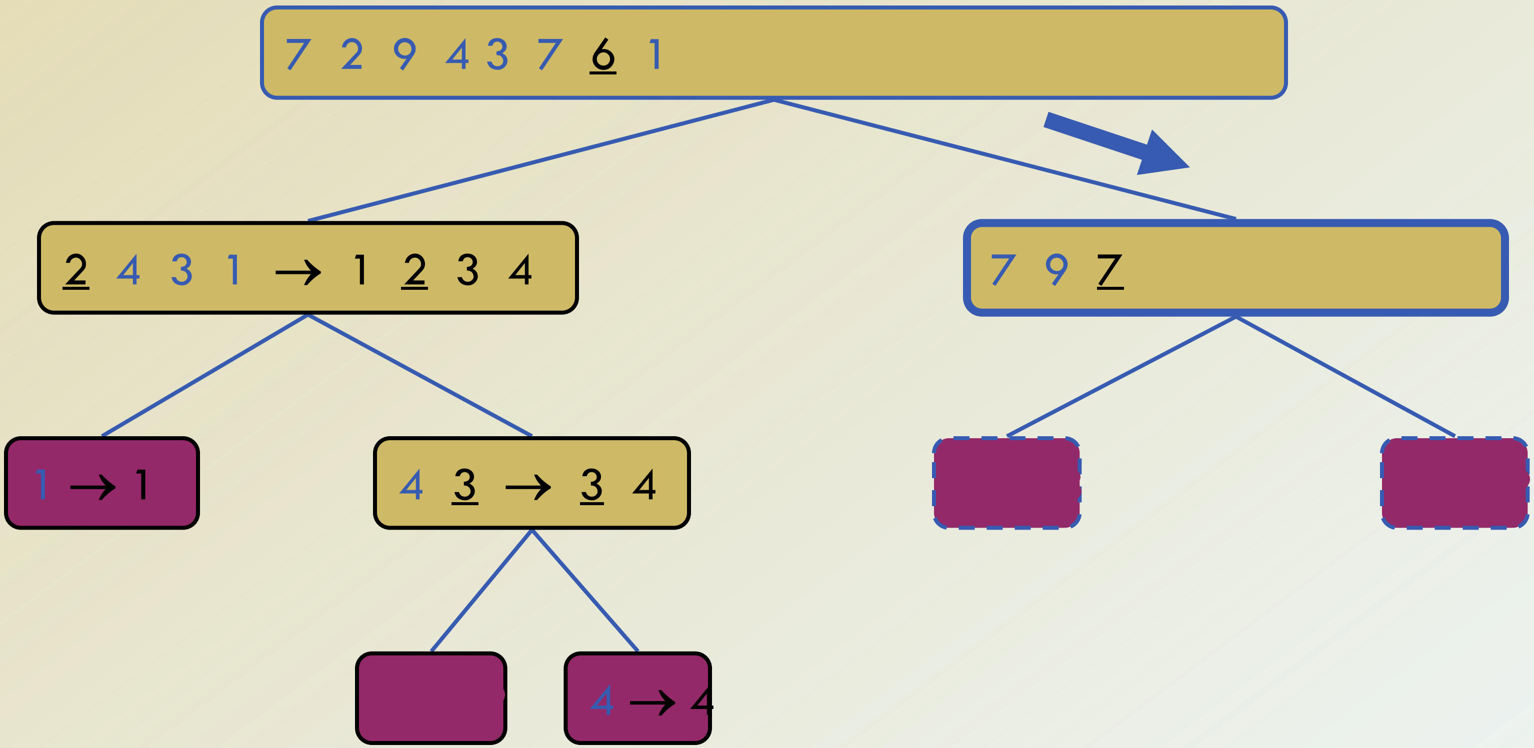
Execution Example (cont.)

- Recursive call, …, base case, join



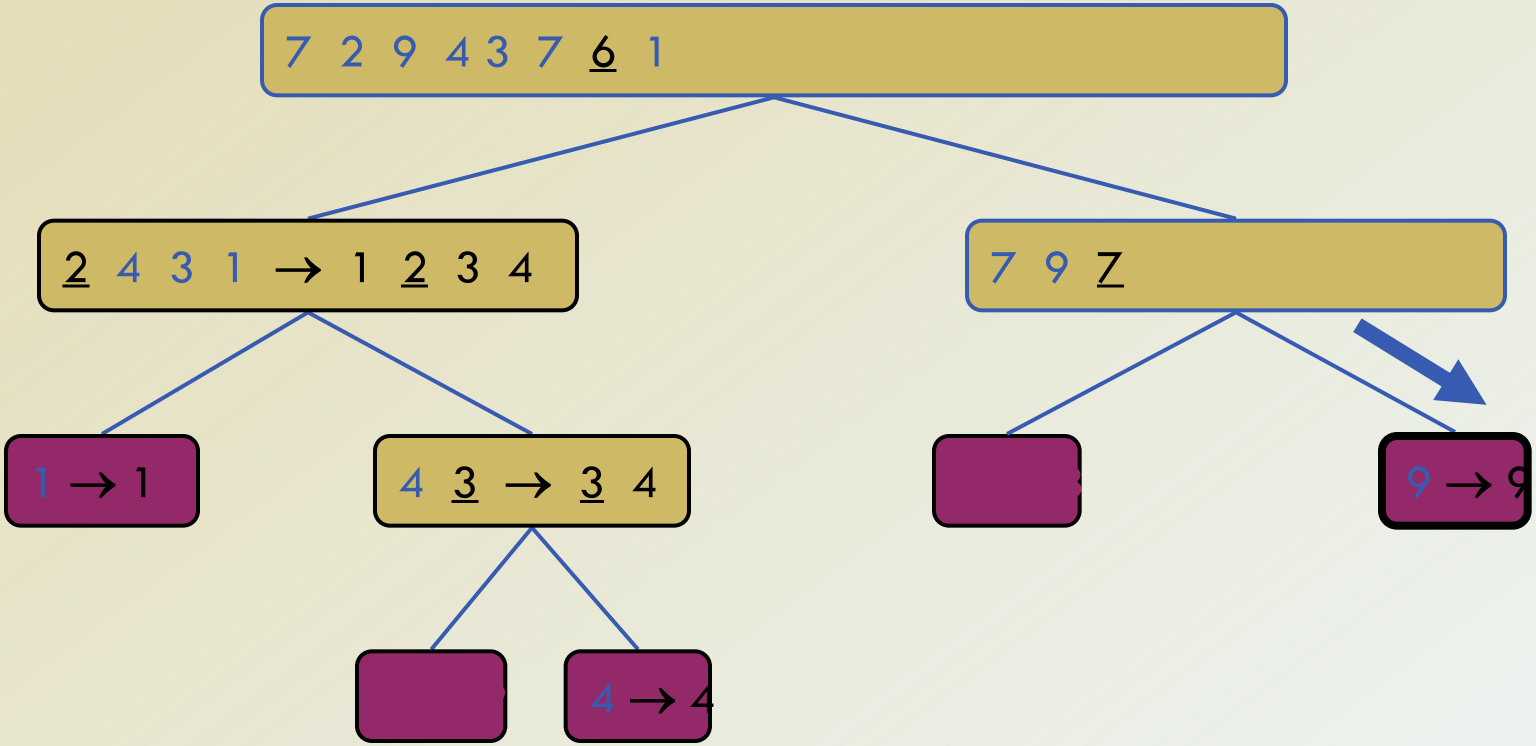
Execution Example (cont.)

- Recursive call, pivot selection



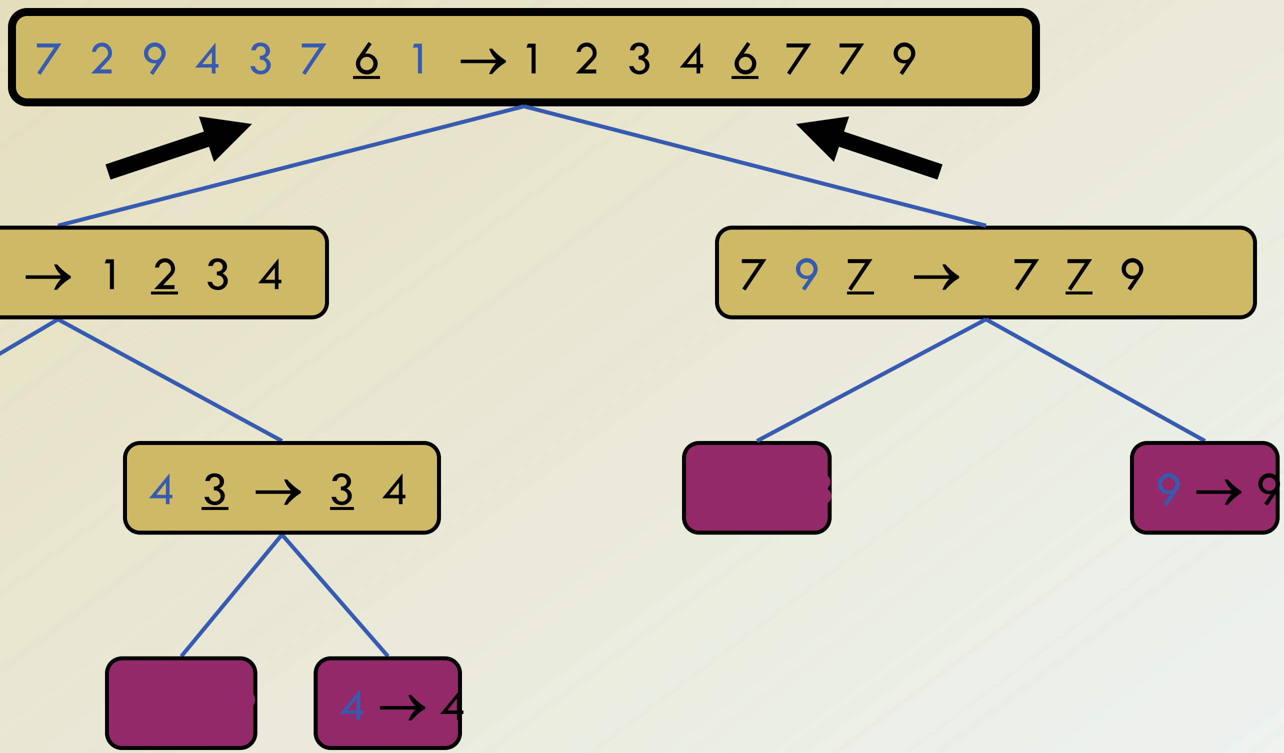
Execution Example (cont.)

- Partition, ..., recursive call, base case



Execution Example (cont.)

- Join, join



In-Place Quick-Sort

- Quick-sort can be implemented to run in-place
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that
  - the elements less than the pivot have rank less than h
  - the elements equal to the pivot have rank between h and k
  - the elements greater than the pivot have rank greater than k
- The recursive calls consider
  - elements with rank less than h
  - elements with rank greater than k

```
Algorithm inPlaceQuickSort(S,a,b):
    Input: An array S of distinct elements; integers a and b
    Output: Array S with elements originally from indices from a to b, inclusive,
    sorted in nondecreasing order from indices a to b
    if a ≥ b then return  {at most one element in subrange}
    p ← S[b]  {the pivot}
    l ← a  {will scan rightward}
    r ← b−1  {will scan leftward}
    while l ≤ r do
        {find an element larger than the pivot}
        while l ≤ r and S[l] ≤ p do
            l ← l+1
        {find an element smaller than the pivot}
        while r ≥ l and S[r] ≥ p do
            r ← r−1
        if l < r then
            swap the elements at S[l] and S[r]
    {put the pivot into its final place}
    swap the elements at S[l] and S[b]
    {recursive calls}
    inPlaceQuickSort(S,a,l −1)
    inPlaceQuickSort(S,l +1,b)
    {we are done at this point, since the sorted subarrays are already consecutive}
```

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
bubble-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)

End of Chapter 11