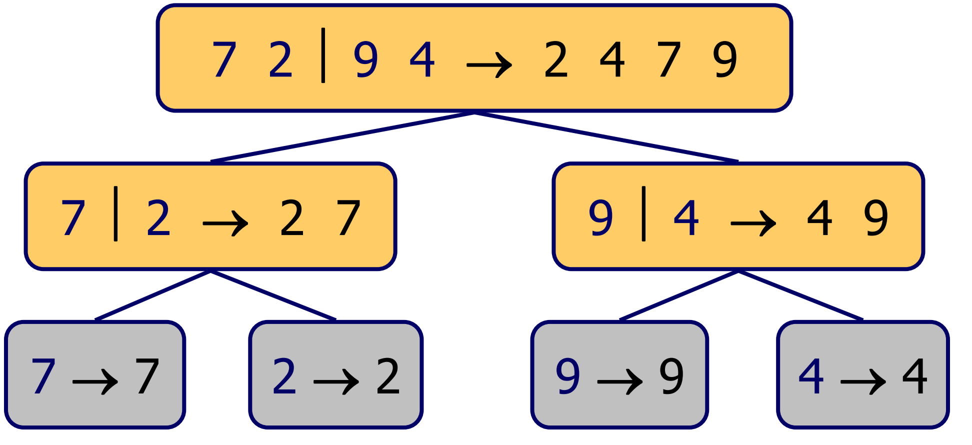
# Merge Sort




Nov 2, 2009

# Outline

- Divide-and-conquer paradigm
- Merge-sort
  - Algorithm
  - Merging two sorted sequences
  - Merge-sort tree
  - Execution example
  - Analysis

# 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

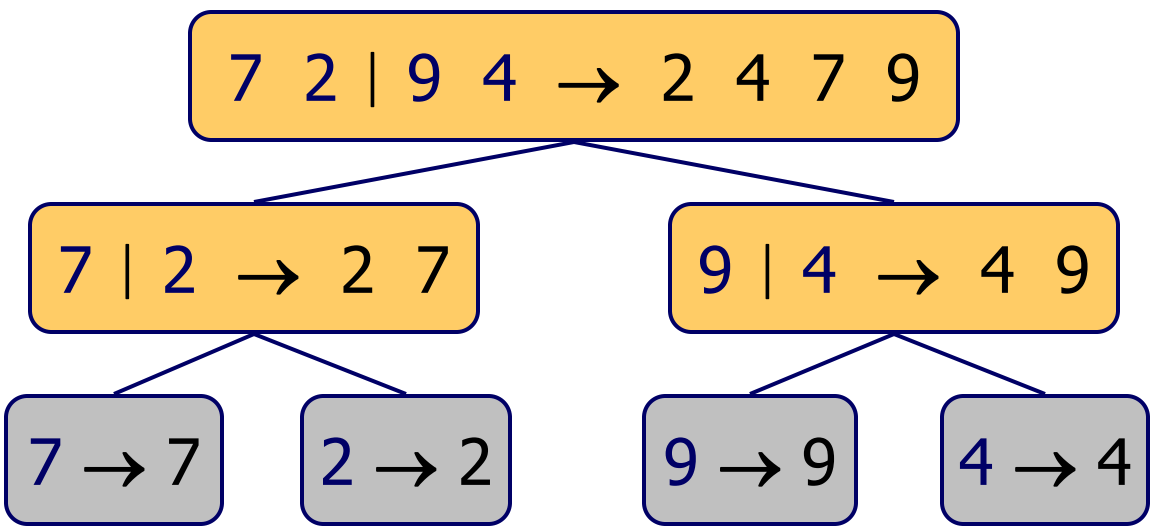
- The conquer step of merge-sort consists of merging two sorted sequences ***A*** and ***B*** into a sorted sequence ***S*** containing the union of the elements of ***A*** and ***B***
- Merging two sorted sequences, each with ***n***/2 elements and implemented by means of a doubly linked list, takes ***O***(***n***) time

```
Algorithm merge(A, B)
    Input sequences A and B with
        n/2 elements each
    Output sorted sequence of A ∪ B

    S ← empty sequence
    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        if A.first().element() < B.first().element()
            S.insertLast(A.remove(A.first()))
        else
            S.insertLast(B.remove(B.first()))
    while ¬A.isEmpty()
        S.insertLast(A.remove(A.first()))
    while ¬B.isEmpty()
        S.insertLast(B.remove(B.first()))
    return S
```

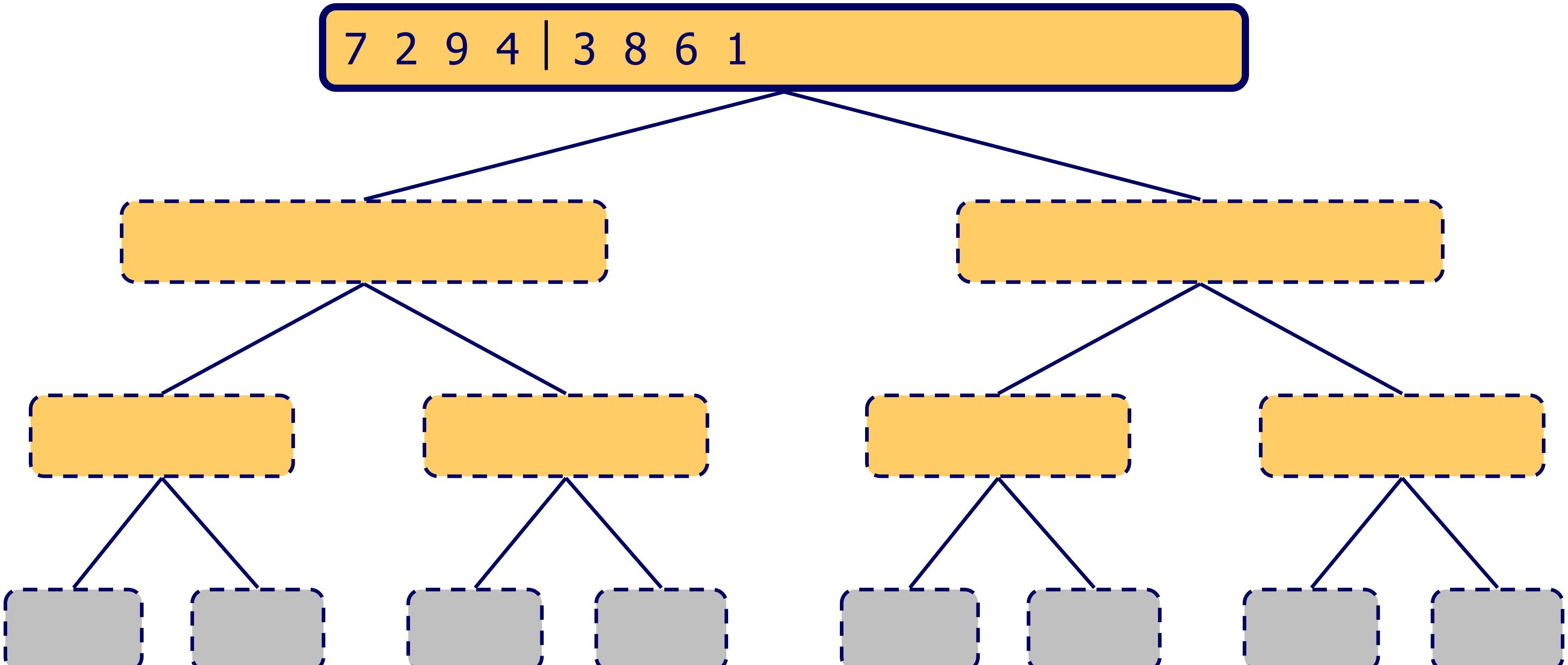
# Merge-Sort Tree

- 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



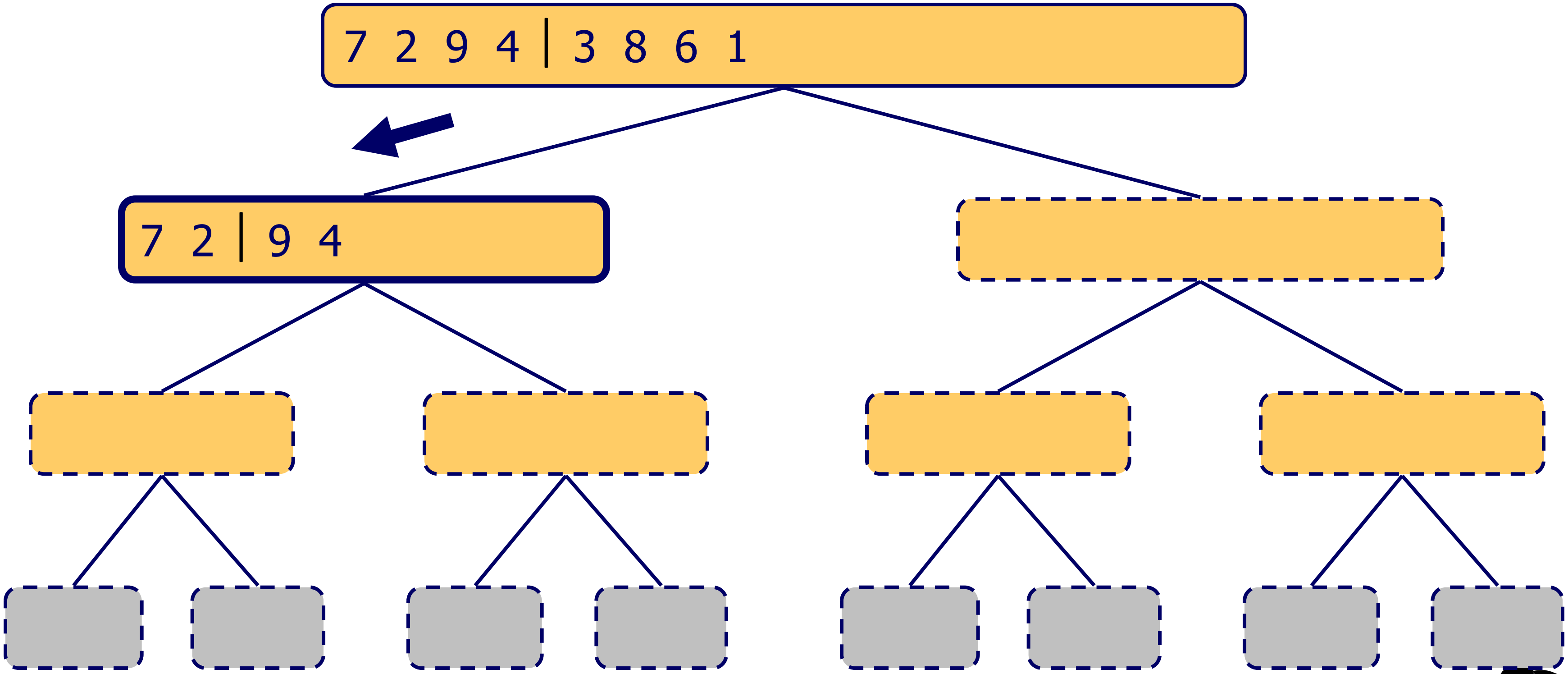
# Execution Example

- Partition



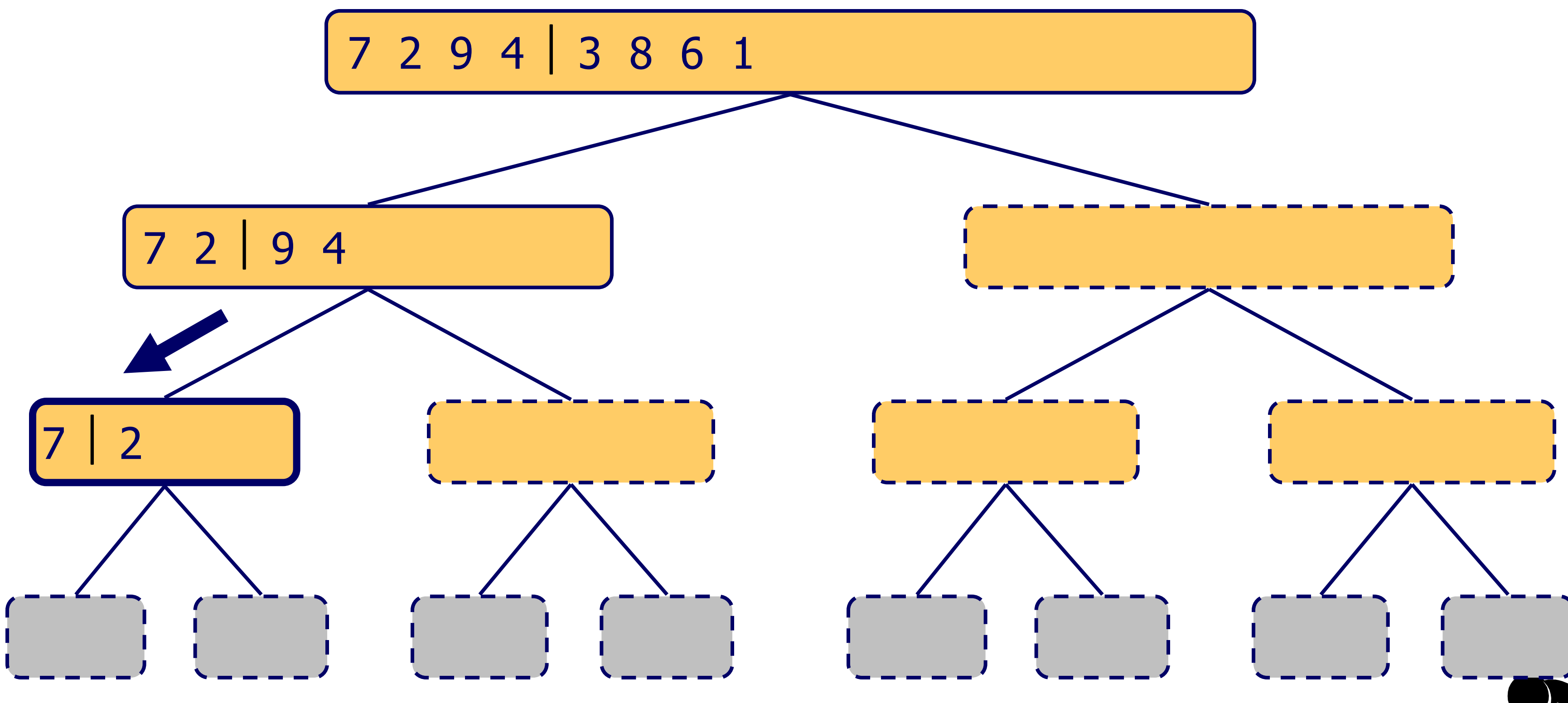
# Execution Example (cont.)

- Recursive call, partition



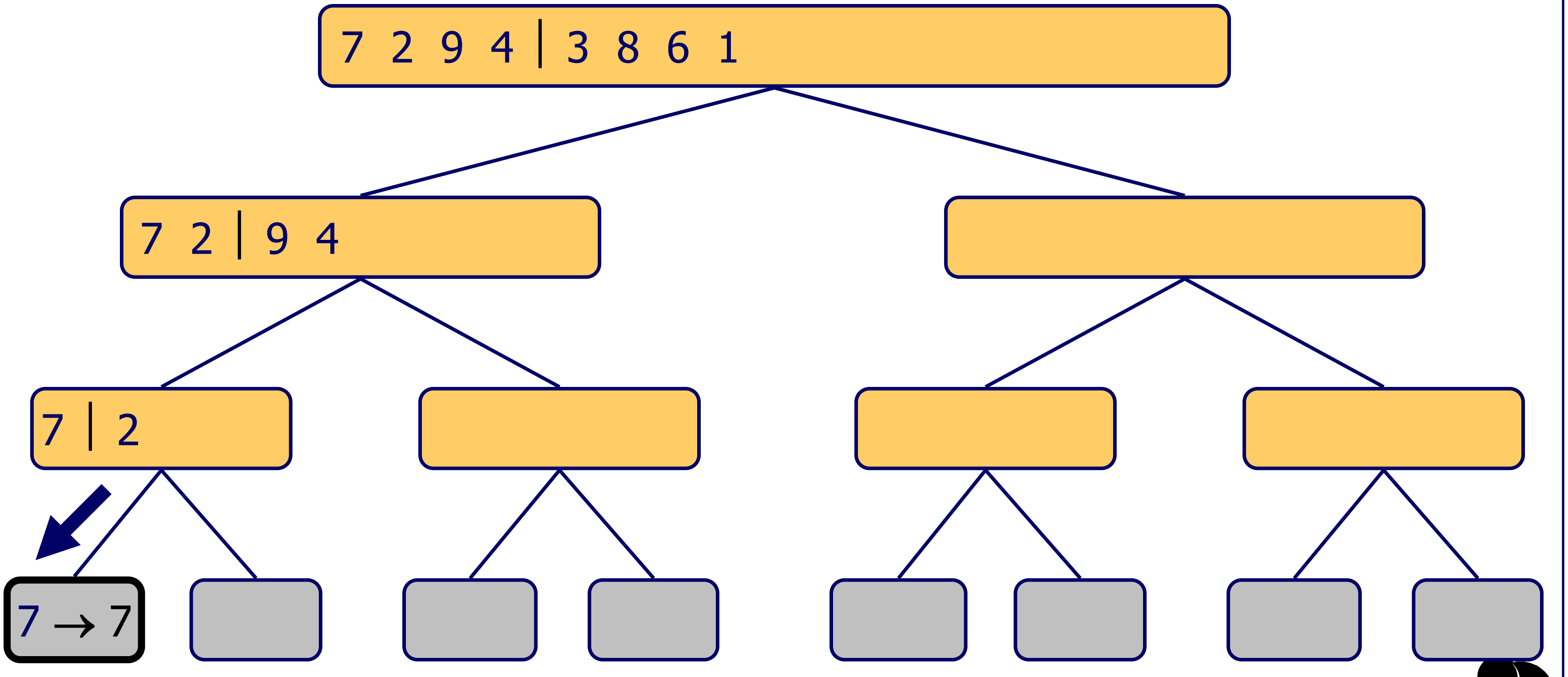
# Execution Example (cont.)

- Recursive call, partition



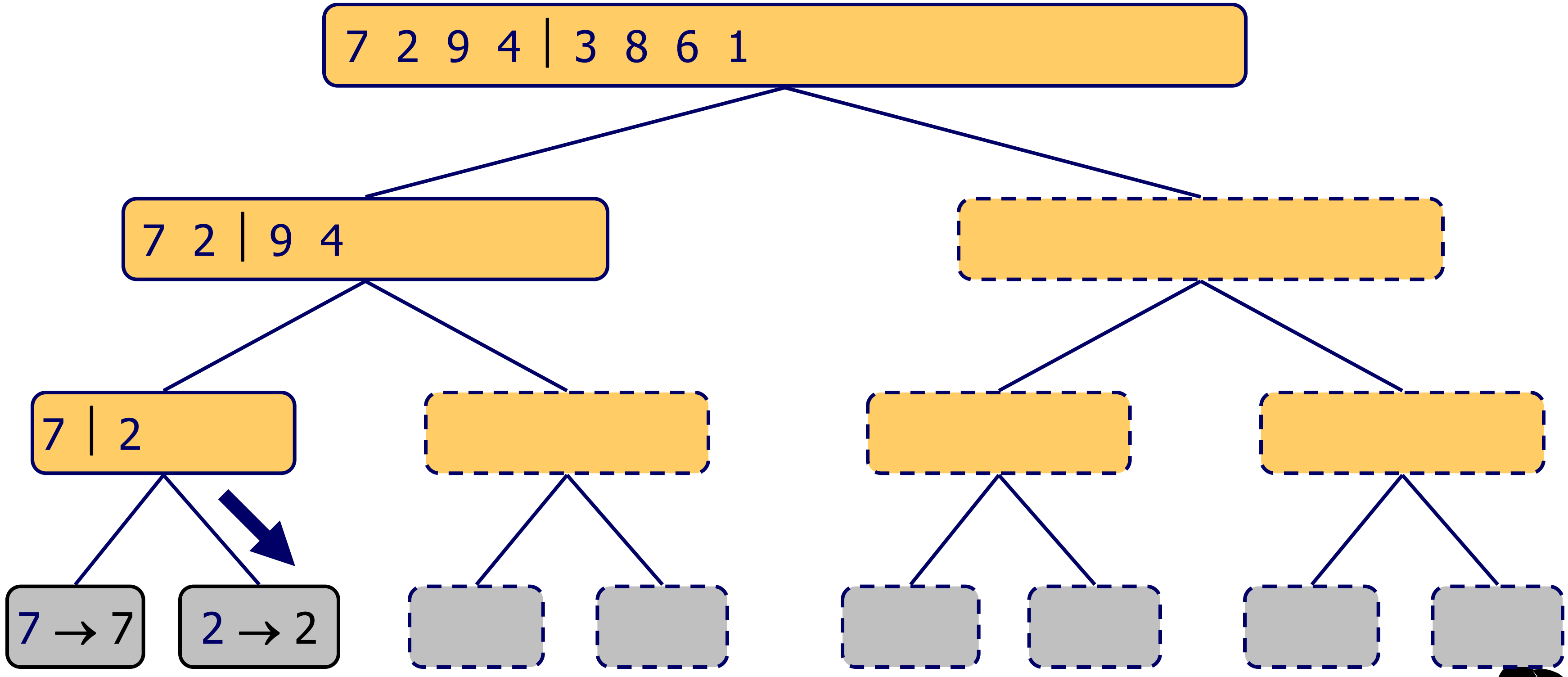
# Execution Example (cont.)

- Recursive call, base case



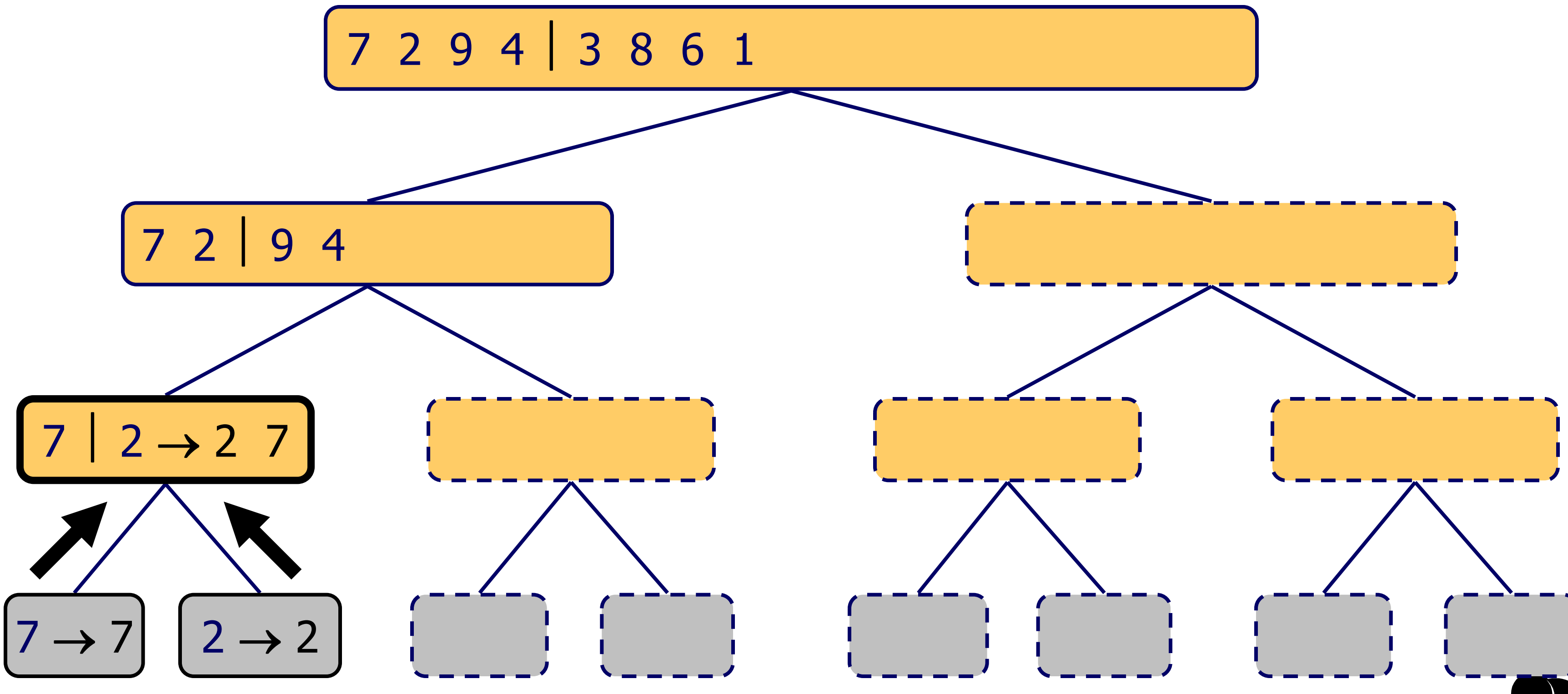
# Execution Example (cont.)

- Recursive call, base case



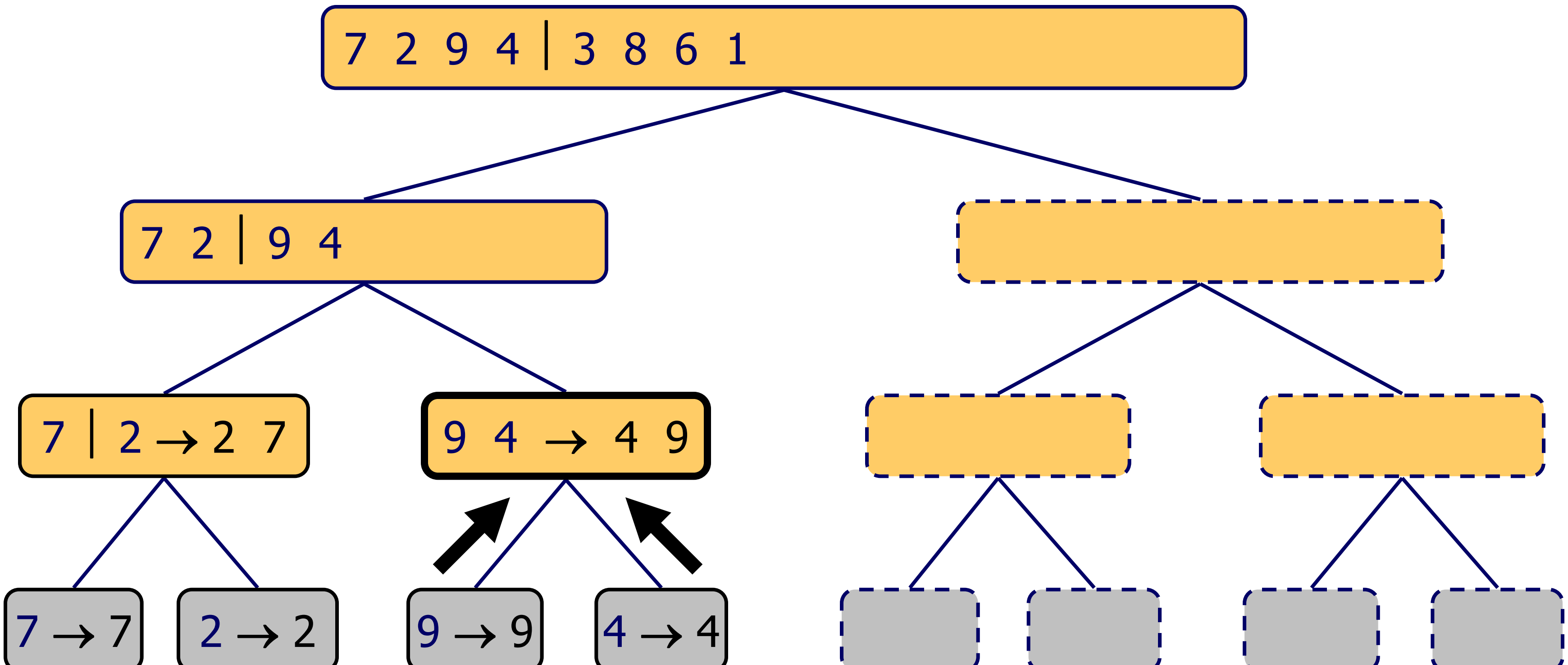
# Execution Example (cont.)

- Merge



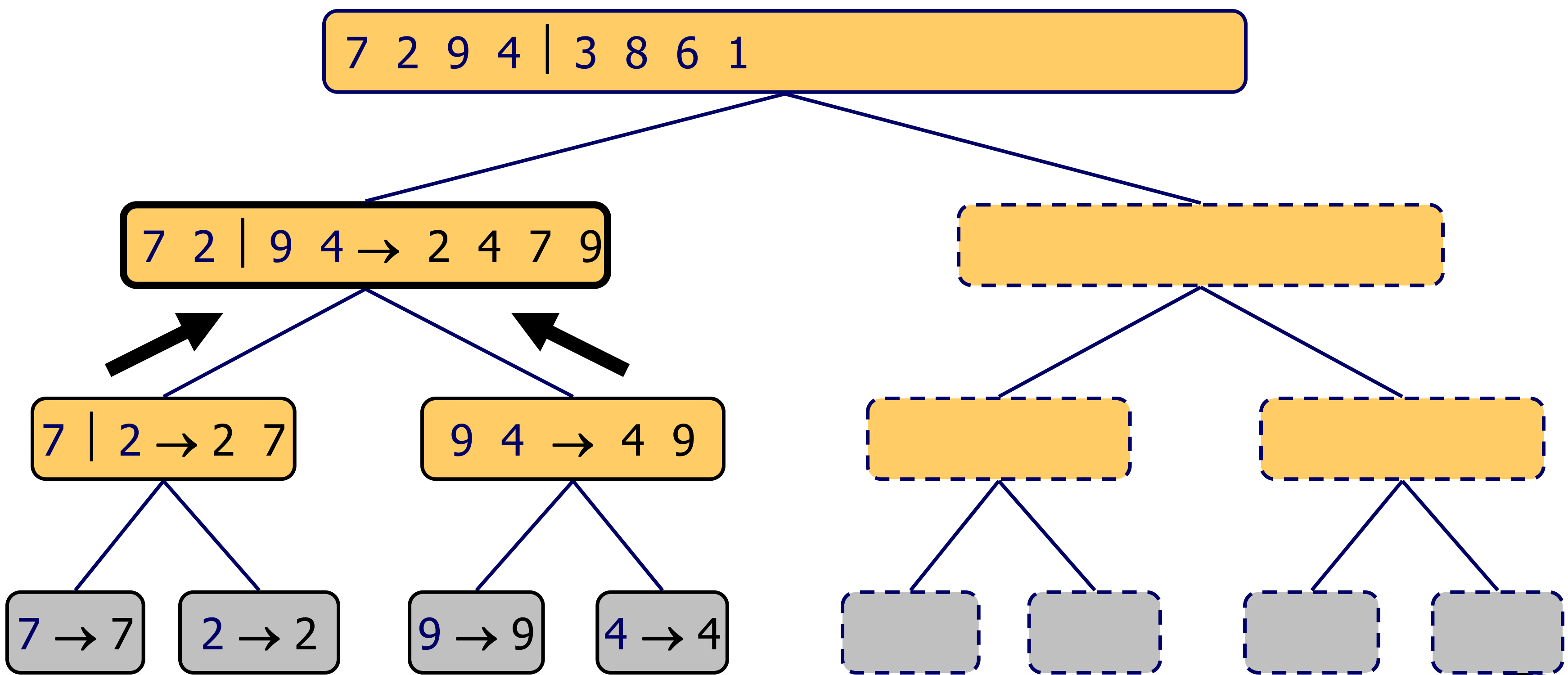
# Execution Example (cont.)

- Recursive call, …, base case, merge



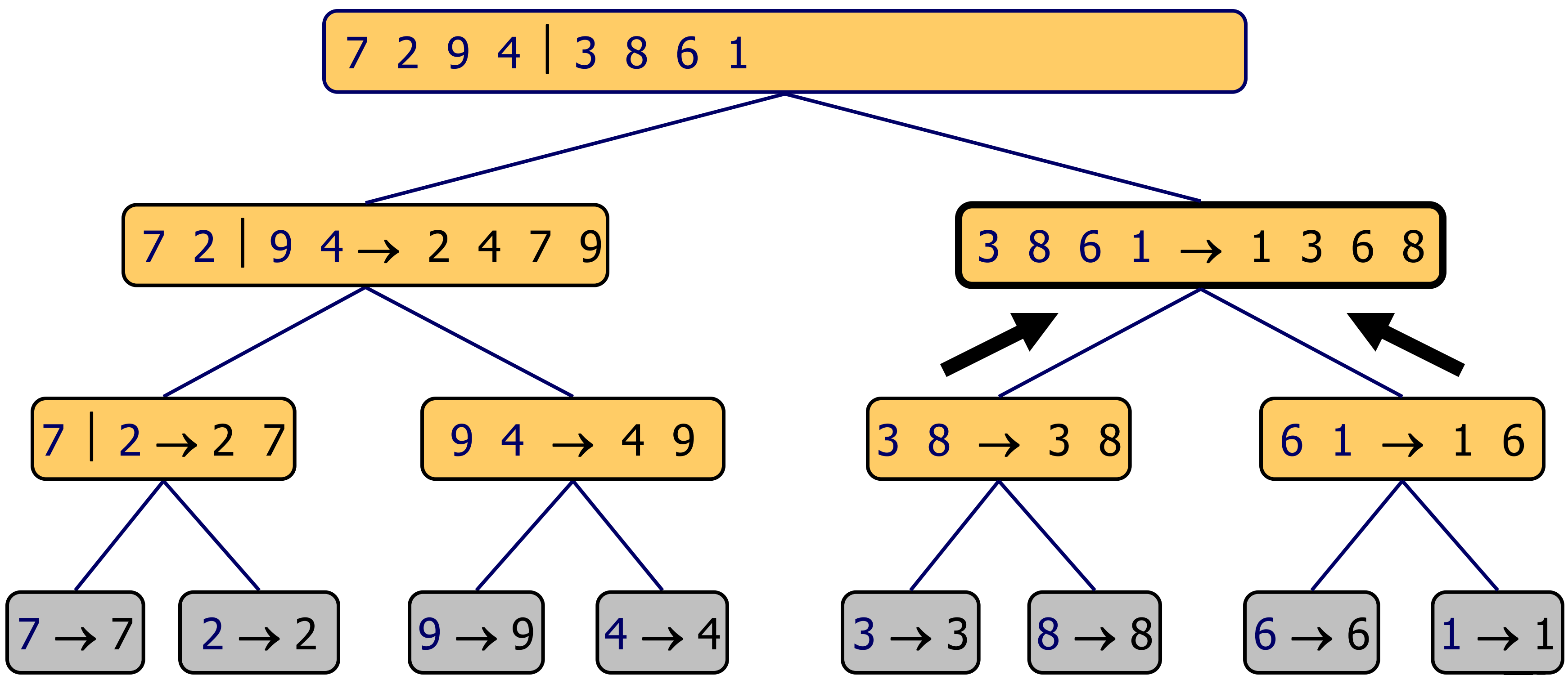
# Execution Example (cont.)

- Merge



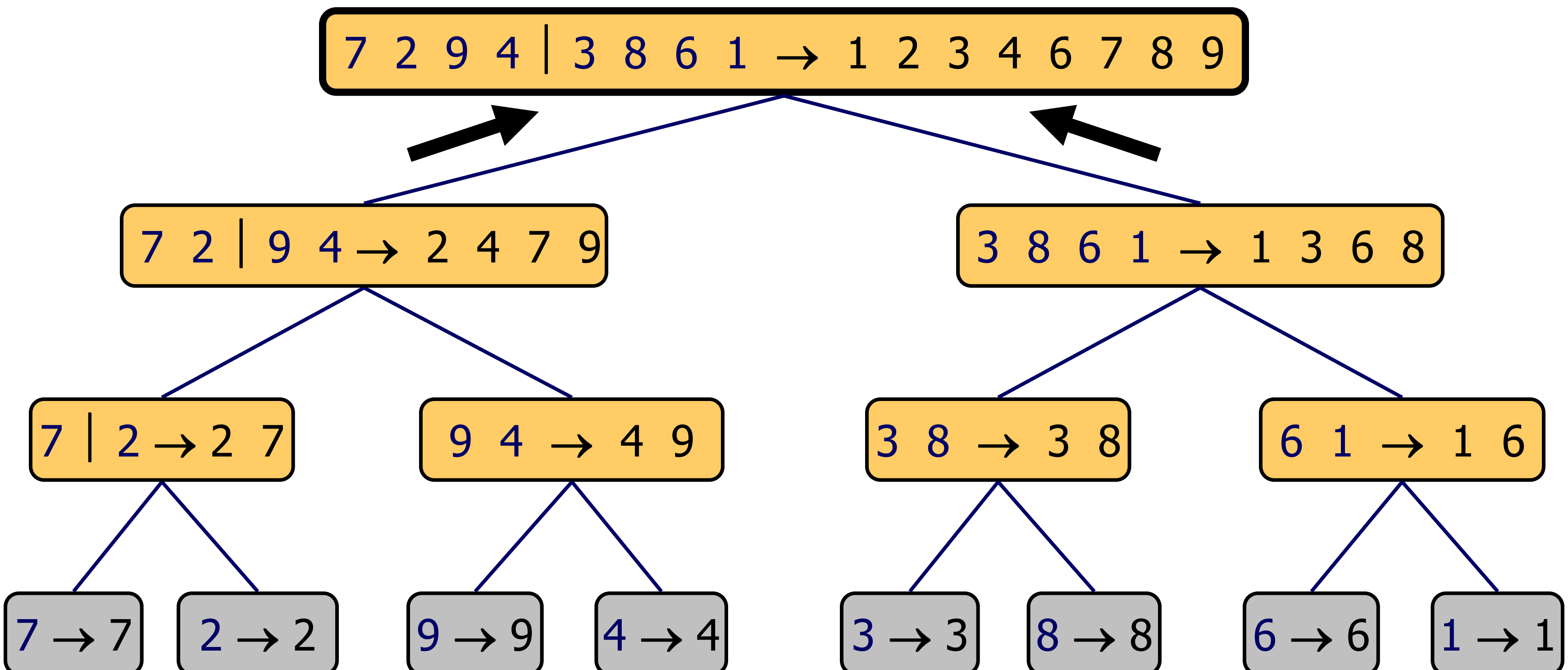
# Execution Example (cont.)

- Recursive call, …, merge, merge



# Execution Example (cont.)

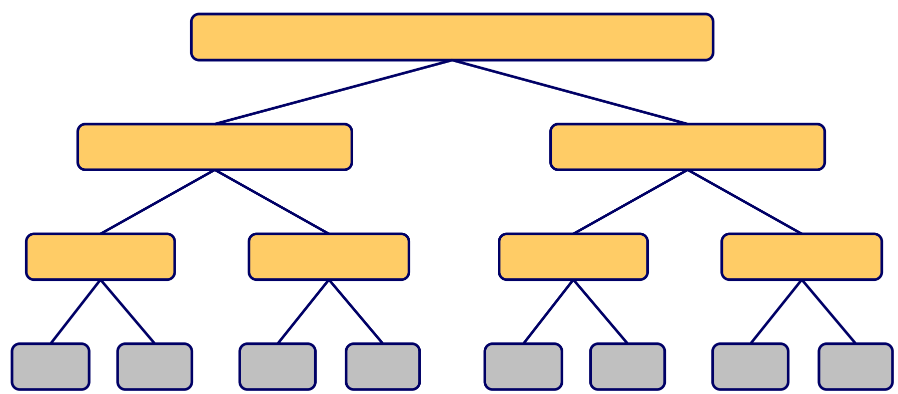
- Merge



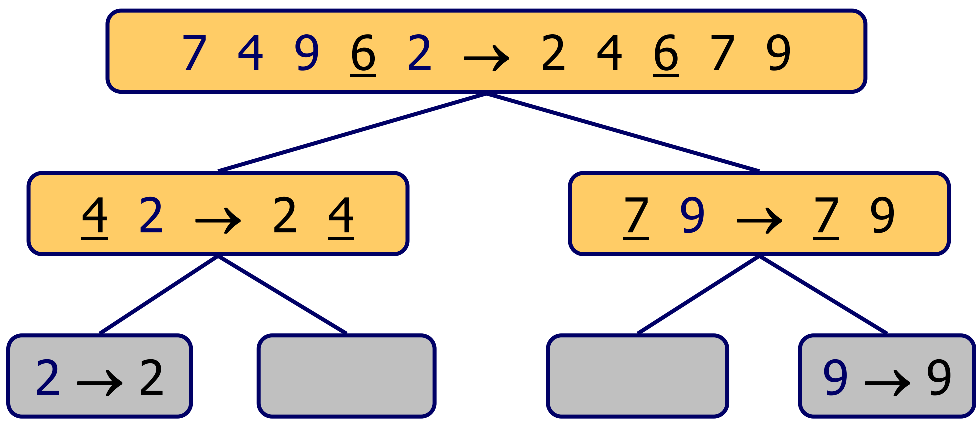
# Analysis of Merge-Sort

- 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 | #seqs | size |
|---|---|---|
| 0 | 1 | $n$ |
| 1 | 2 | $n/2$ |
| $i$ | $2^i$ | $n/2^i$ |
| … | … | … |



# Quick-Sort

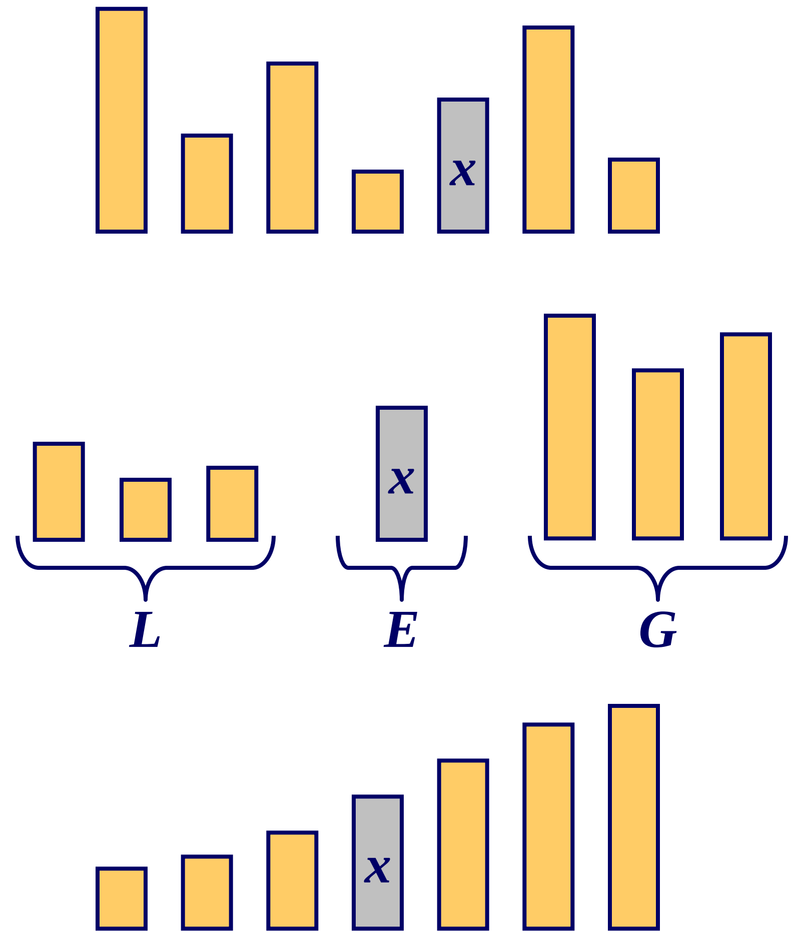


# Outline

- Quick-sort
  - Algorithm
  - Partition step
  - Quick-sort tree
  - Execution example
- Analysis of quick-sort
- Summary of sorting algorithms

# 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 $G$



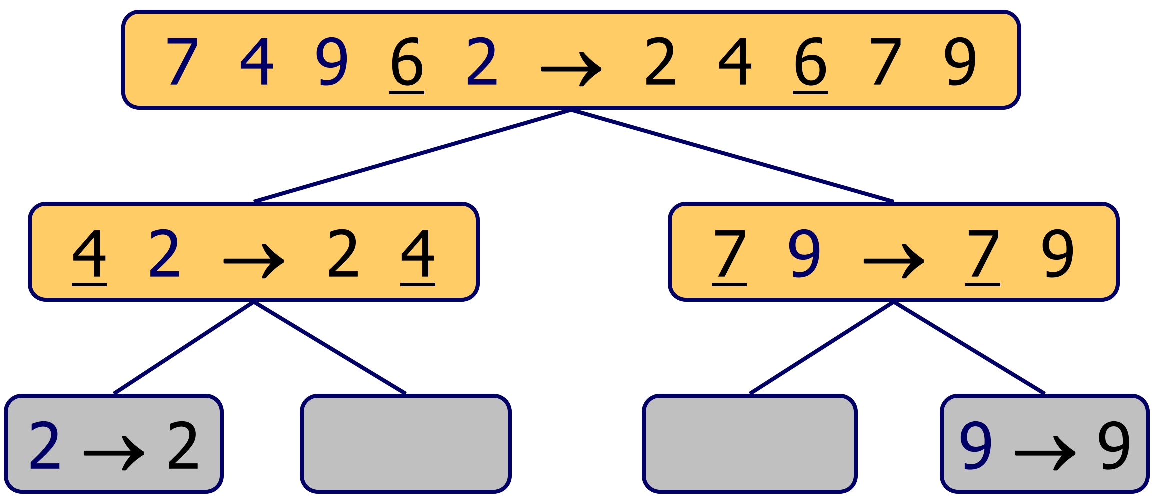
# Partition

- 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.
    $L, E, G \leftarrow$ empty sequences
    $x \leftarrow S.remove(p)$
    **while** $\neg S.isEmpty()$
        $y \leftarrow S.remove(S.first())$
        **if** $y < x$
            $L.insertLast(y)$
        **else if** $y = x$
            $E.insertLast(y)$
        **else** $\{ y > x \}$
            $G.insertLast(y)$
    **return** $L, E, G$

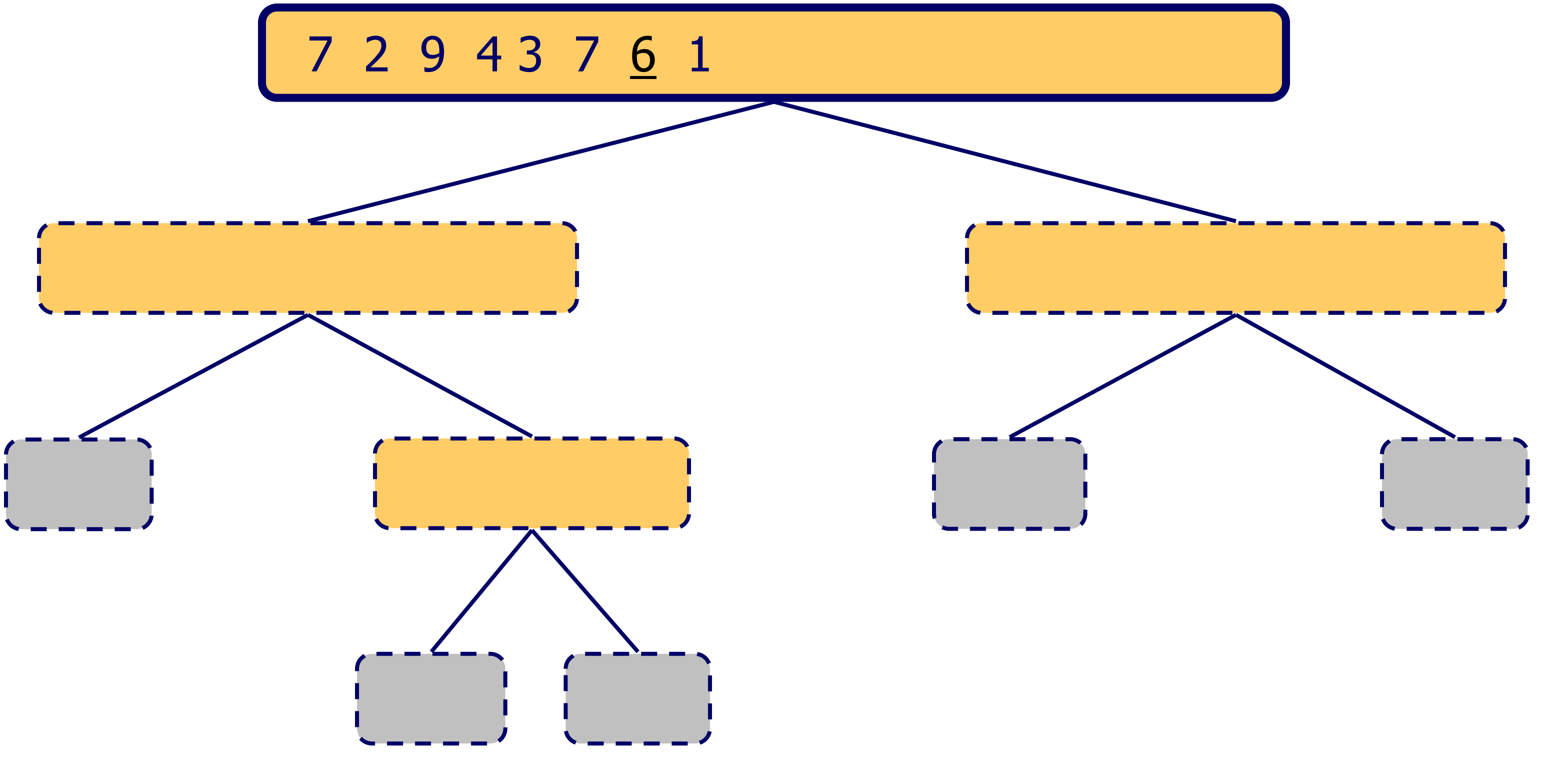
# Quick-Sort Tree

- 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



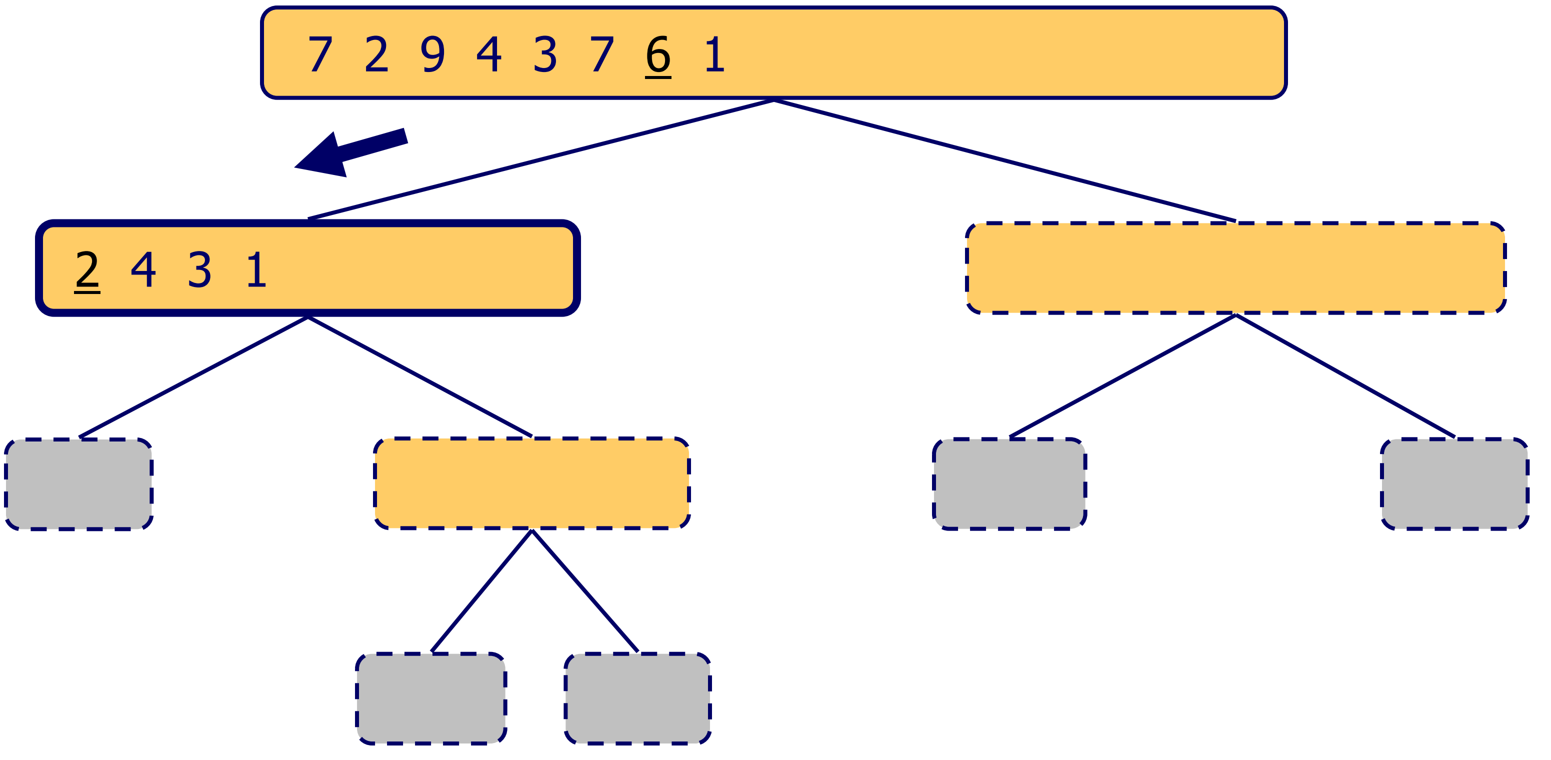
# Execution Example

- Pivot selection



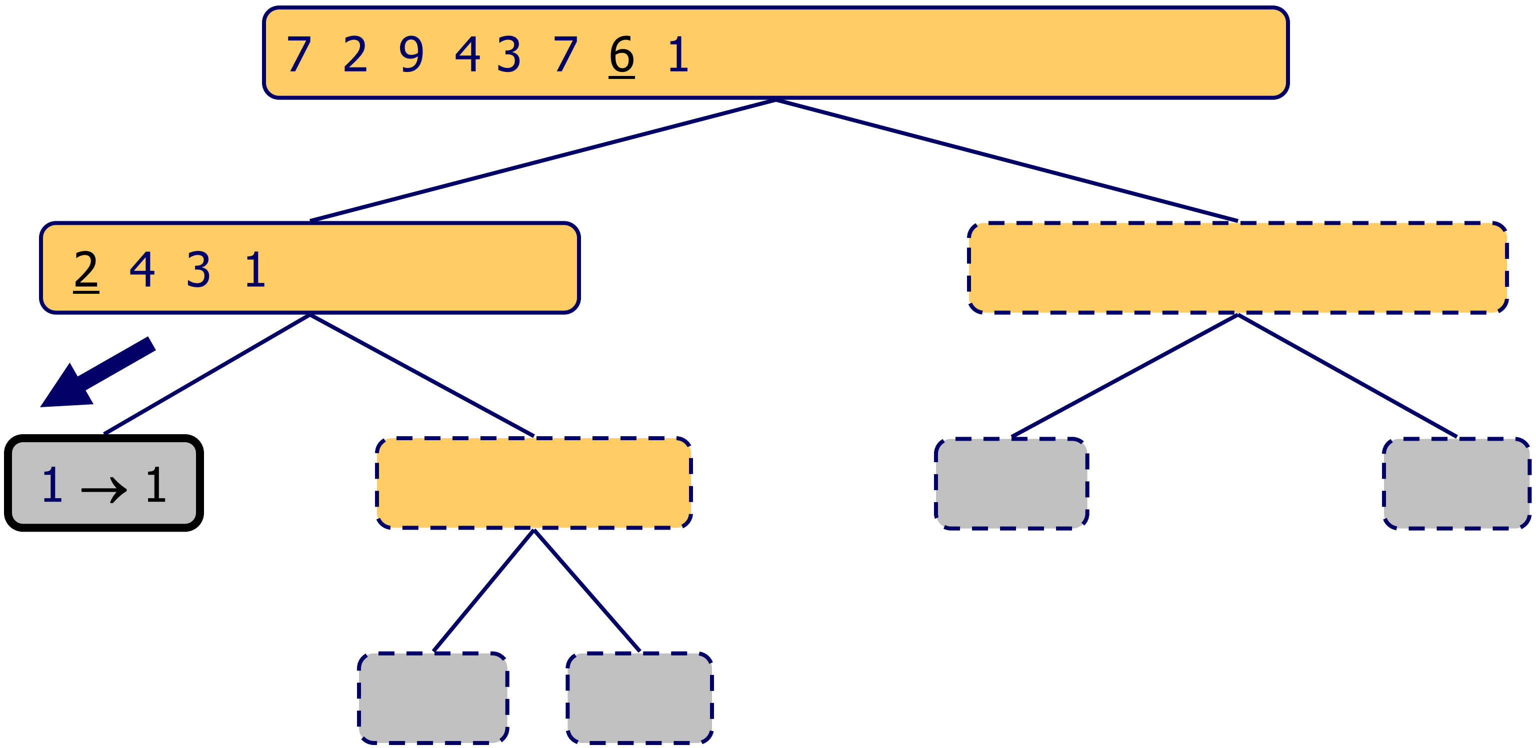
# Execution Example (cont.)

- Partition, recursive call, pivot selection



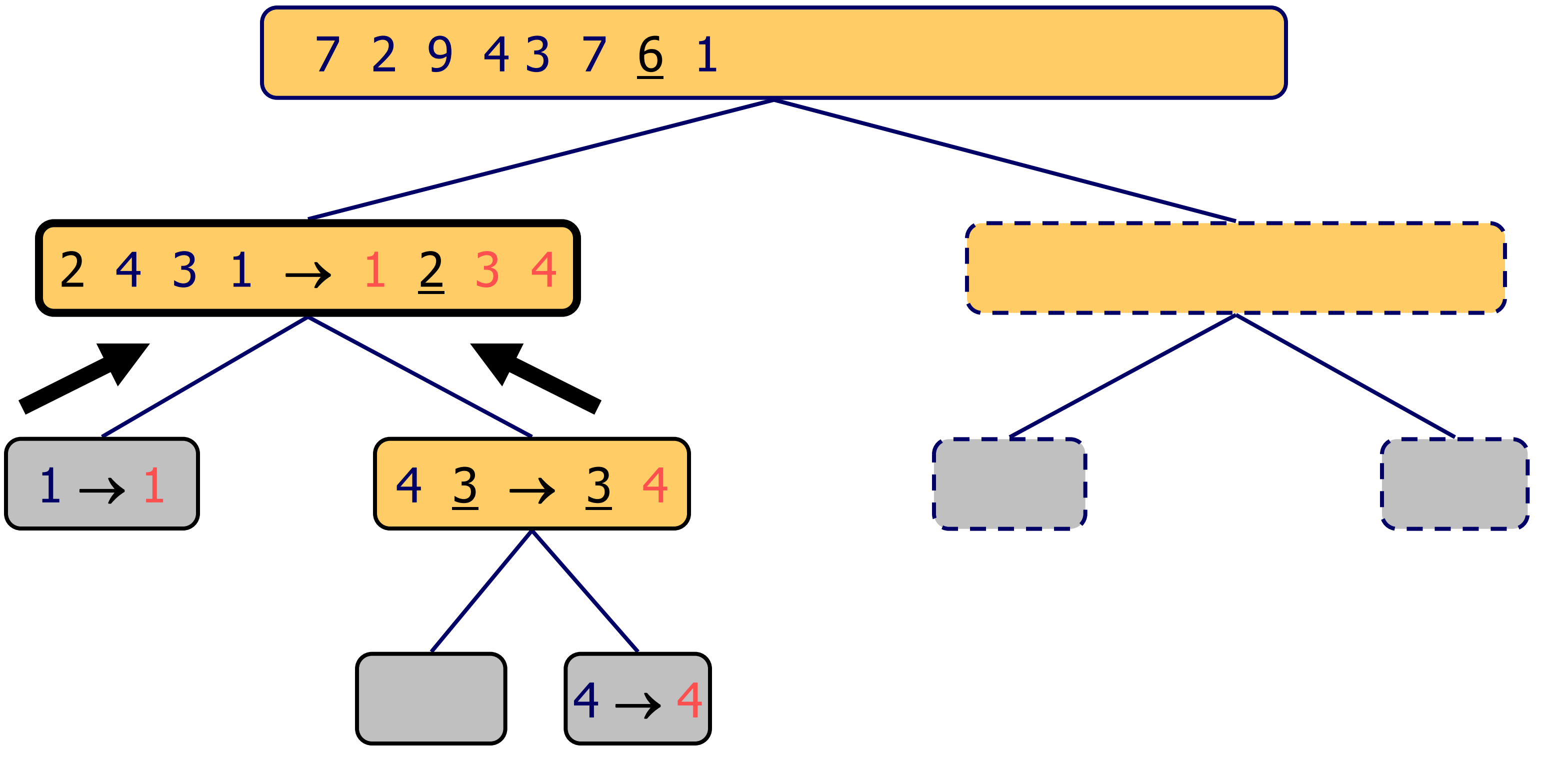
# Execution Example (cont.)

- Partition, recursive call, base case



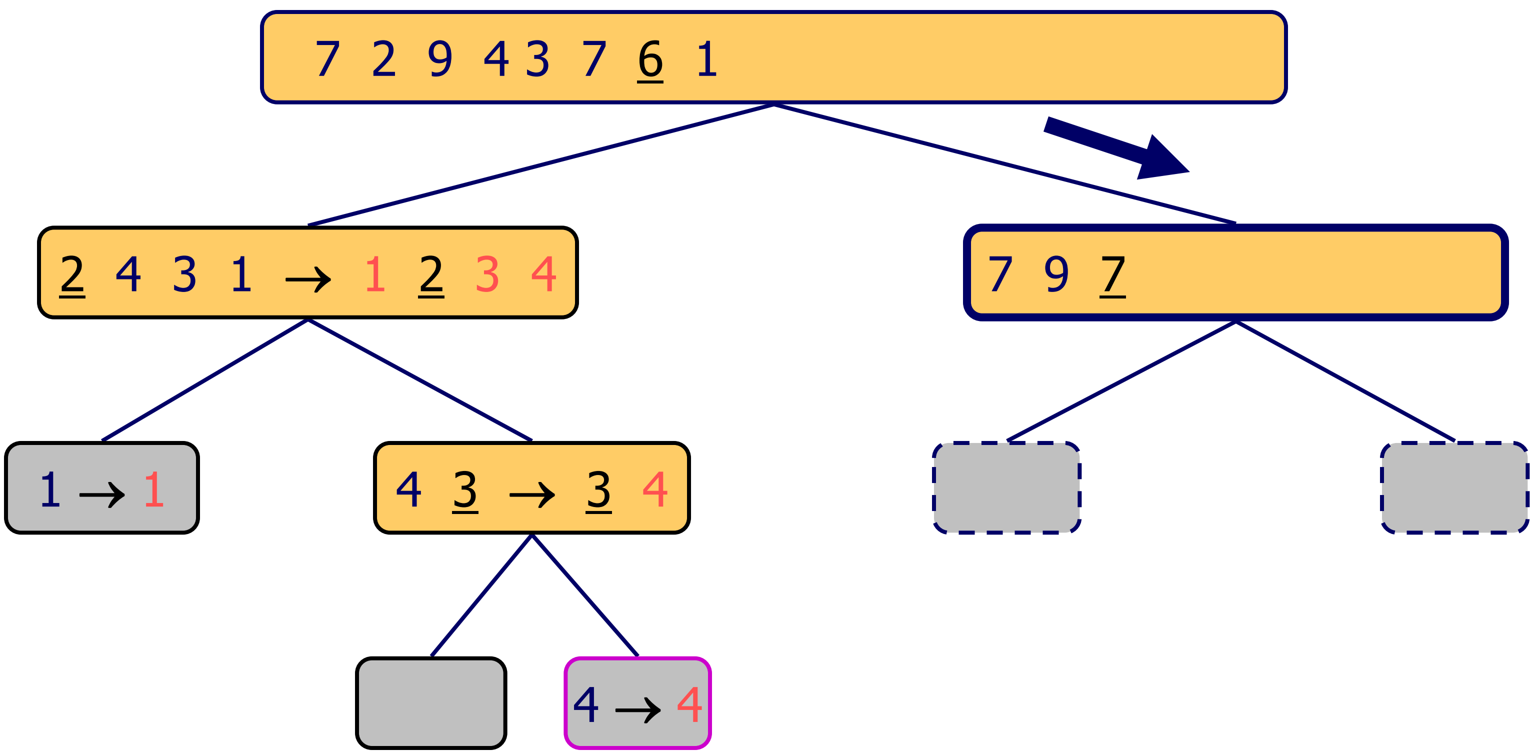
# Execution Example (cont.)

- Recursive call, …, base case, join



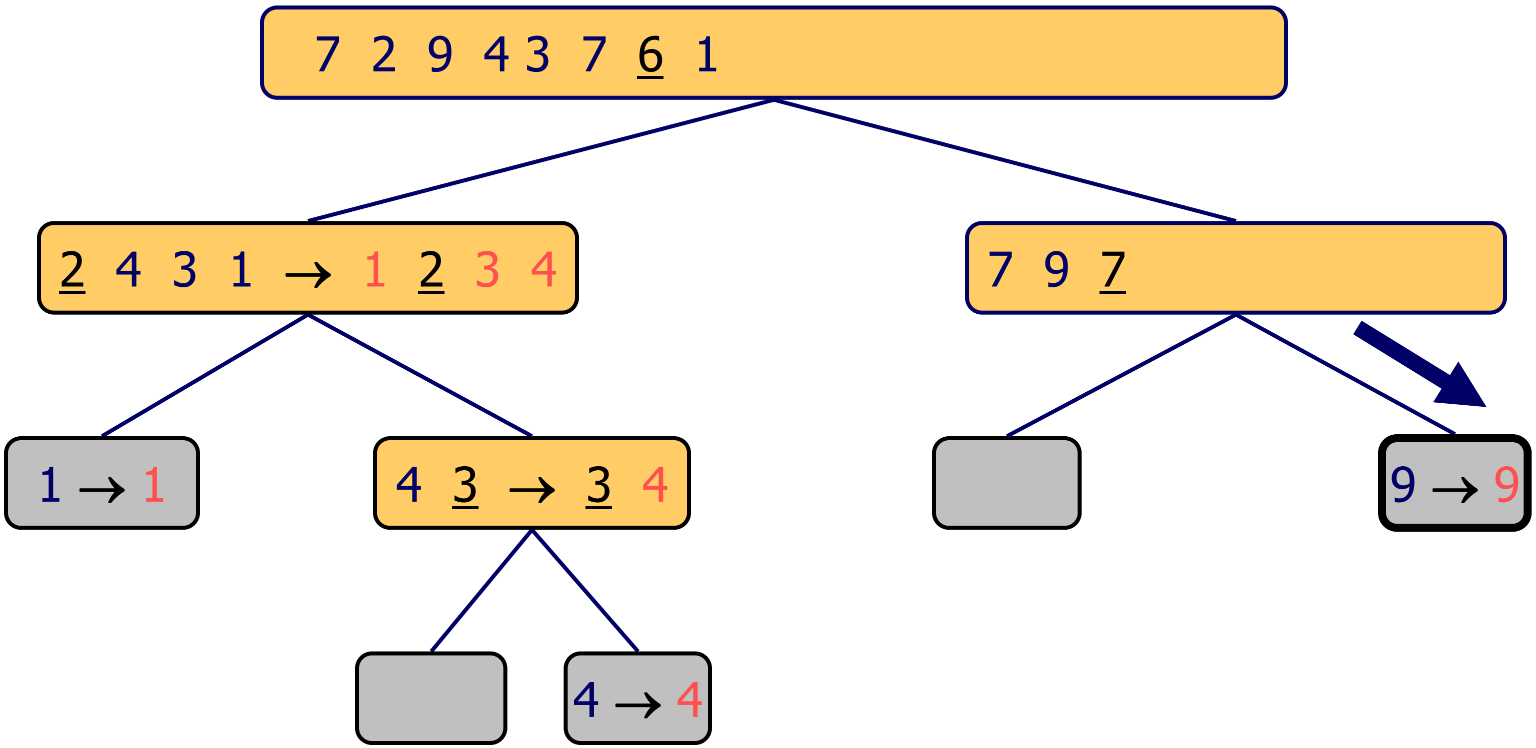
# Execution Example (cont.)

- Recursive call, pivot selection



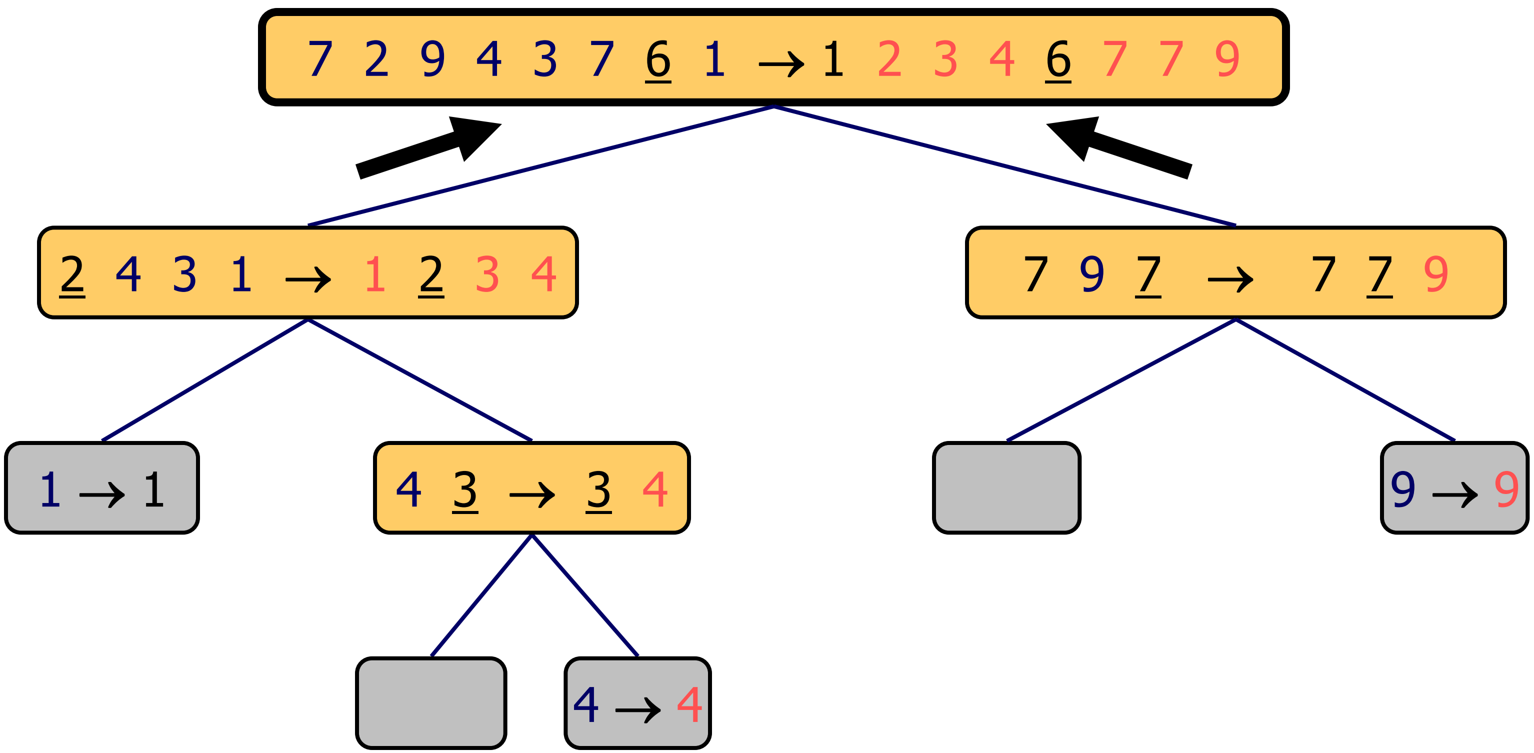
# Execution Example (cont.)

- Partition, …, recursive call, base case



# Execution Example (cont.)

- Join, join



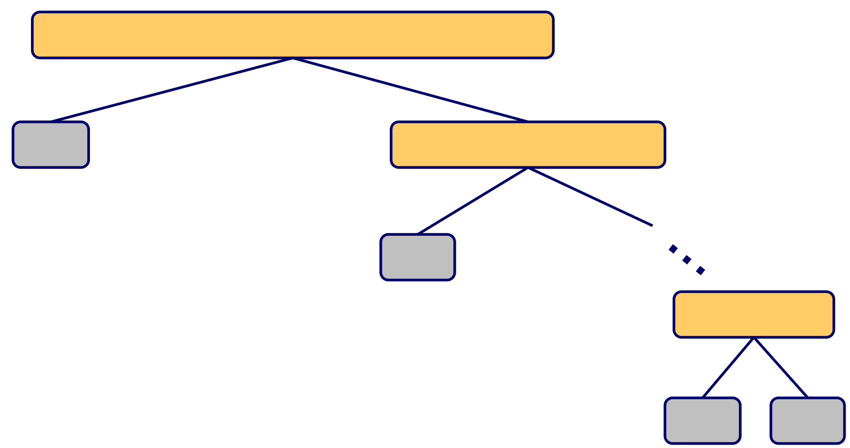
# Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of $L$ and $G$ has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

- Thus, the worst-case running time of quick-sort is $O(n^2)$

| depth | time |
|---|---|
| 0 | $n$ |
| 1 | $n - 1$ |
| … | … |
| $n - 1$ | 1 |



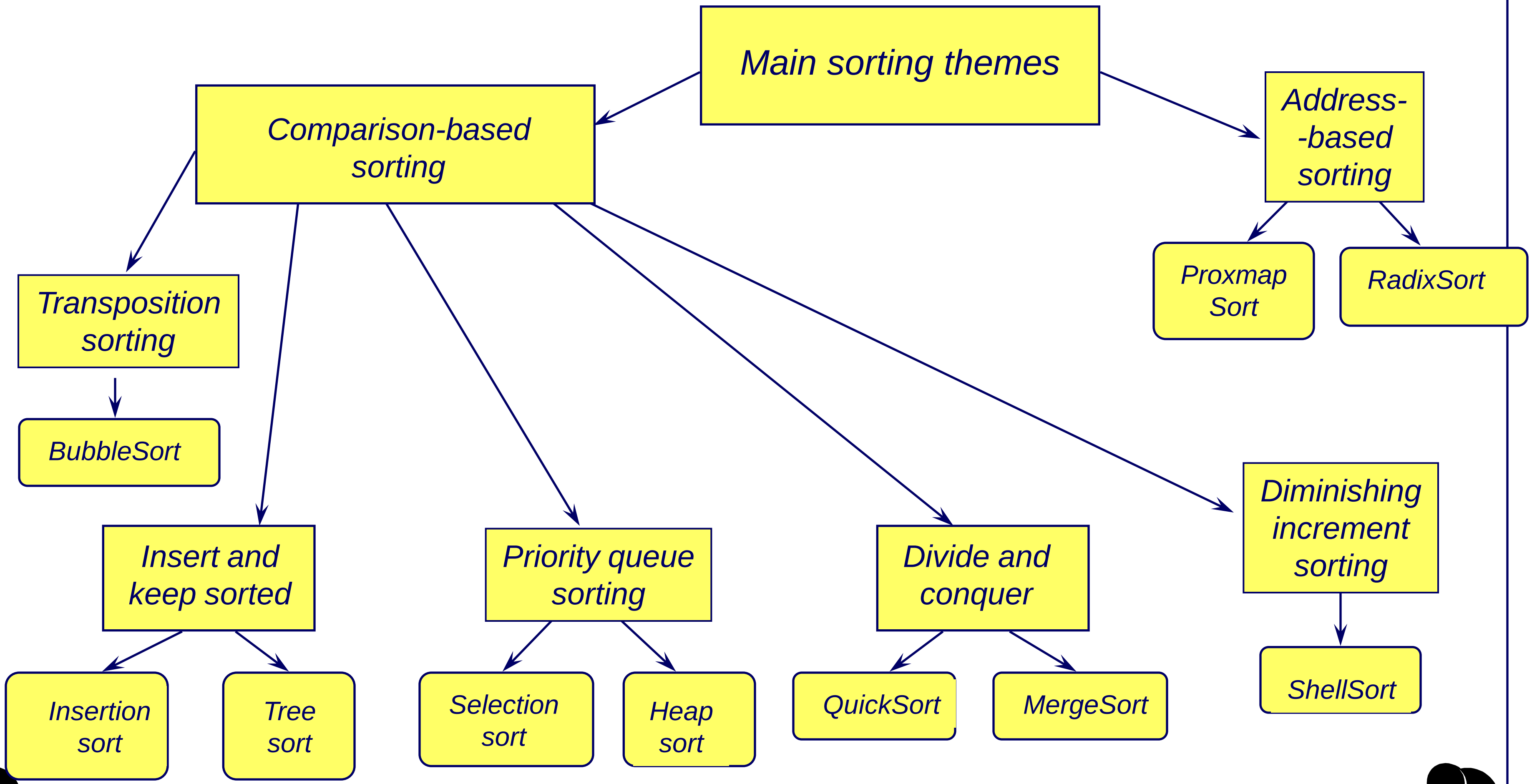
# Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size $s$
  - **Good call:** the sizes of $L$ and $G$ are each less than $3s/4$
  - **Bad call:** one of $L$ and $G$ has size greater than $3s/4$


**Good call** **Bad call**

- A call is good with probability 1/2
  - 1/2 of the possible pivots cause good calls:

# The family of sorting methods



# Summary of Sorting Algorithms

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