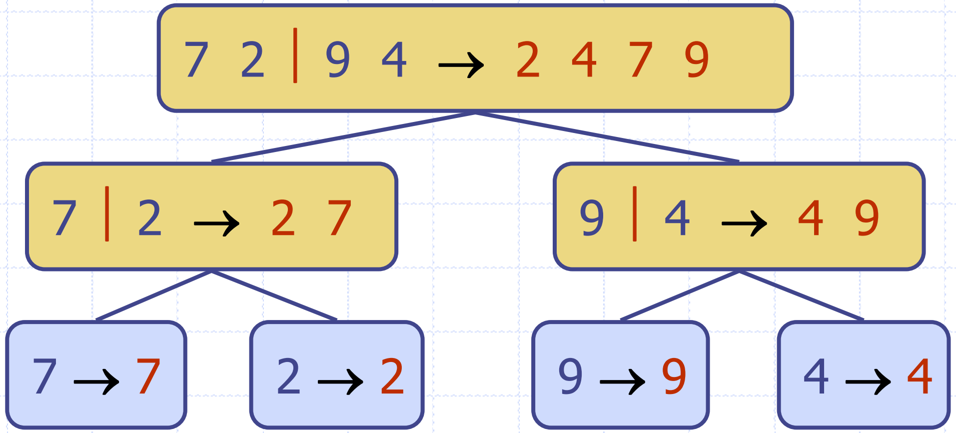
DATA STRUCTURES AND ALGORITHMS

Dr Samabia Tehsin

Bs (ai)

Merge Sort



We will look at this table later …

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)

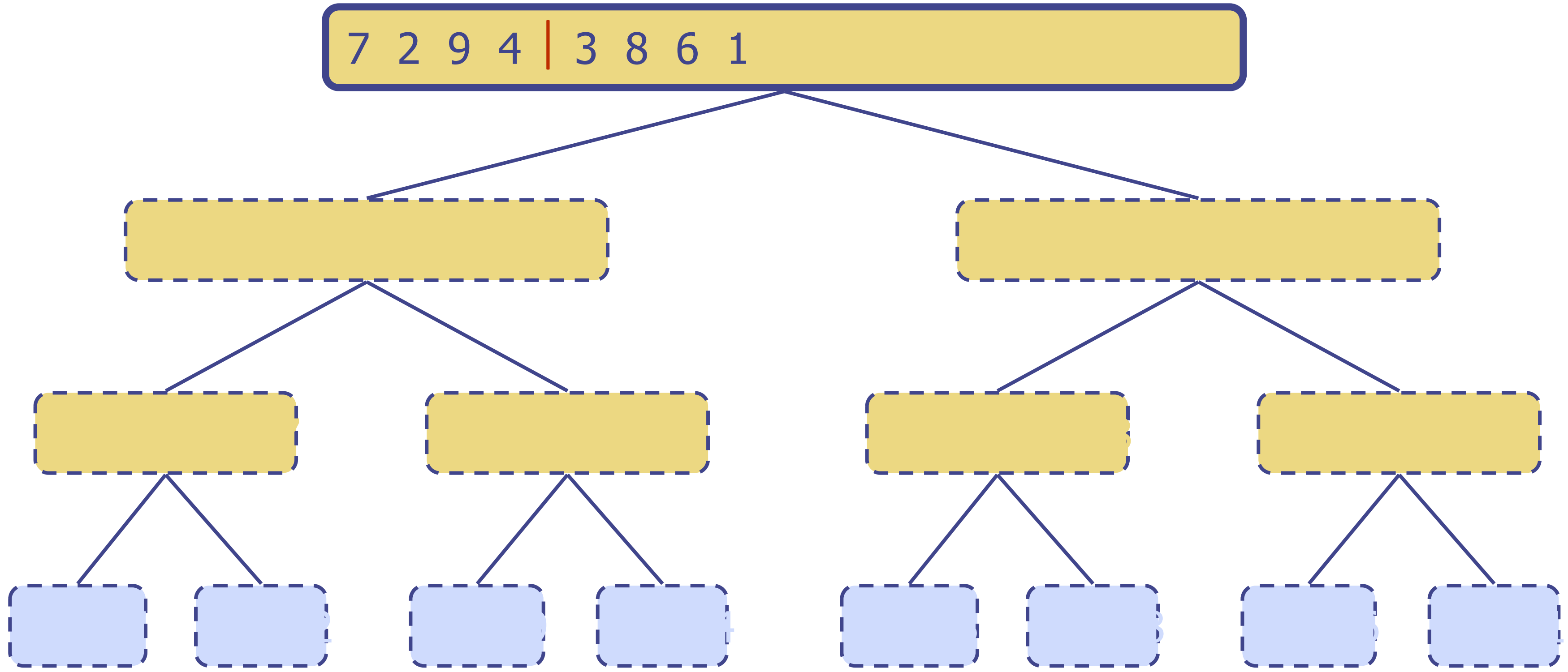
New things that we will learn from this part

- Divide-and-Conquer rationale

- Complexity analysis based on recurrence relation

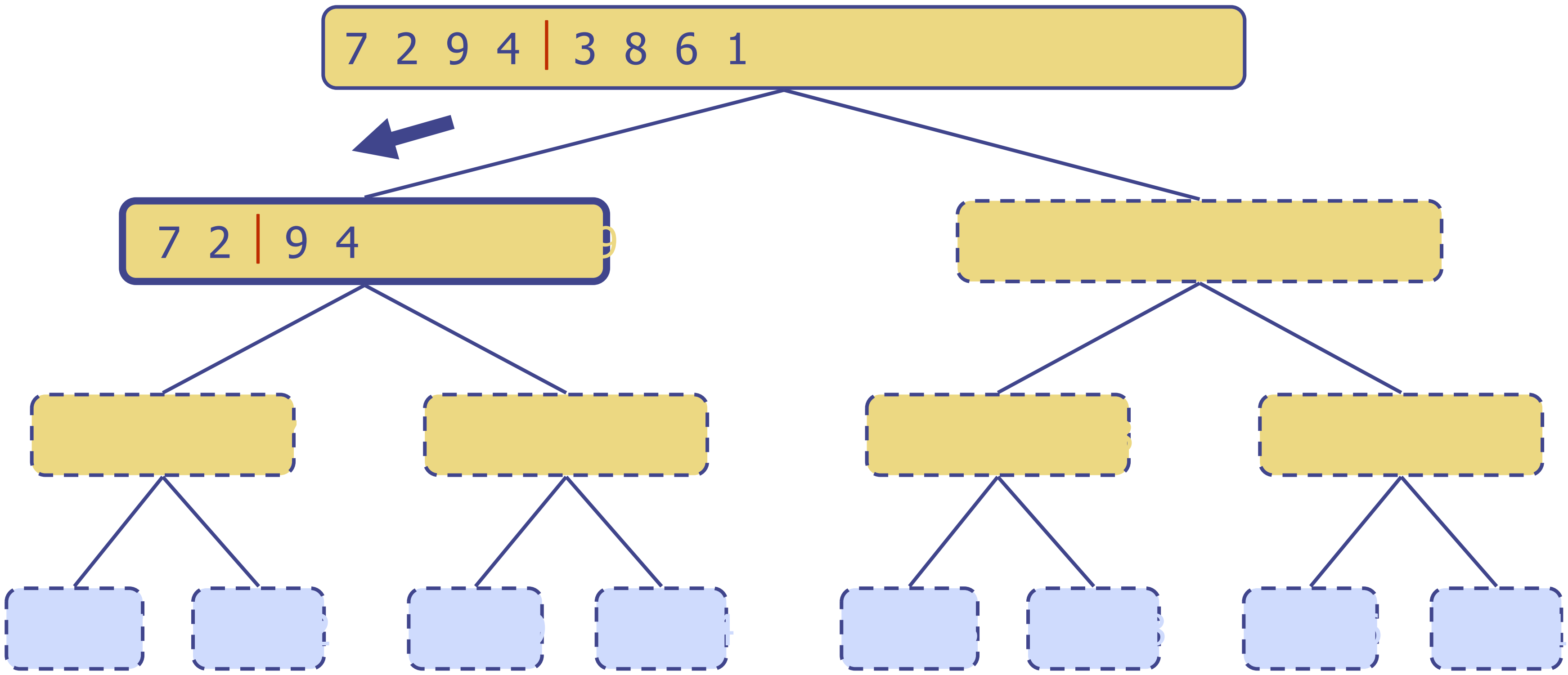
Execution Example

- Partition



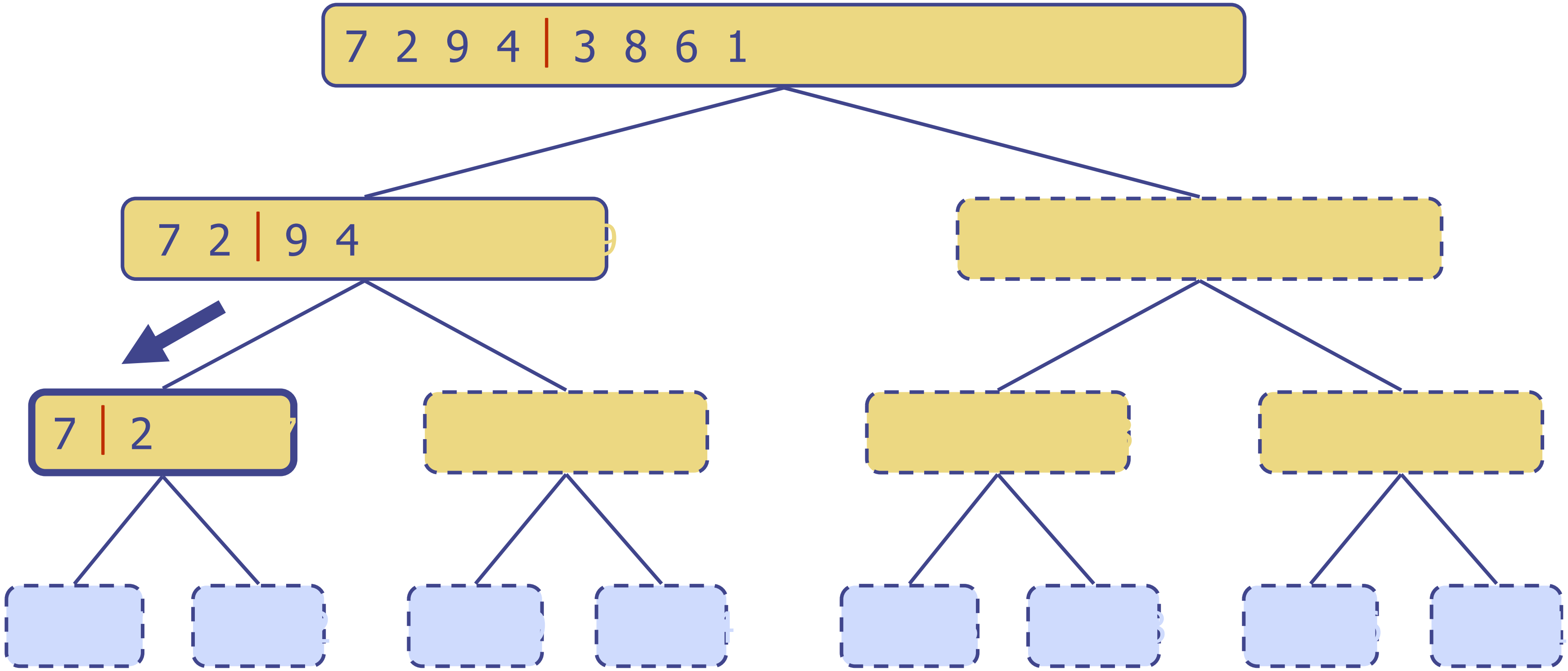
Execution Example (cont.)

- Recursive call, partition



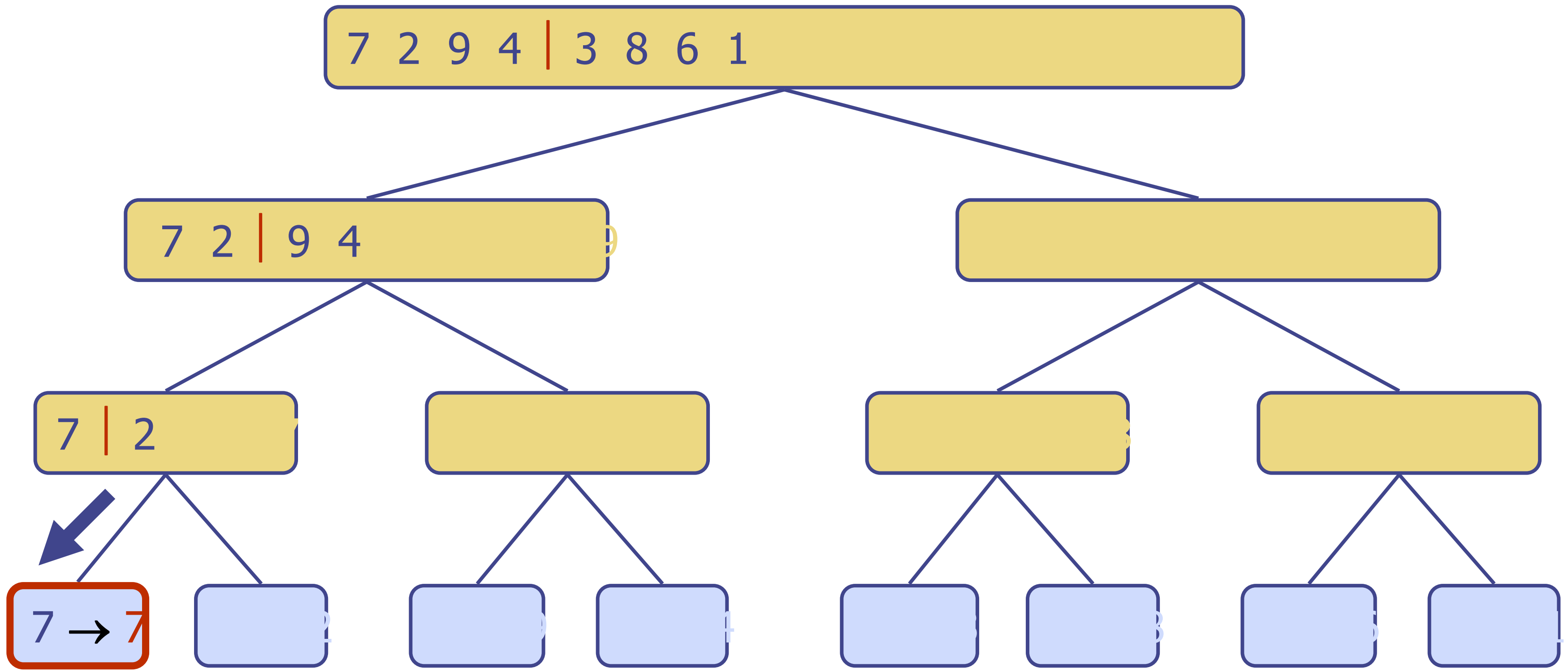
Execution Example (cont.)

- Recursive call, partition



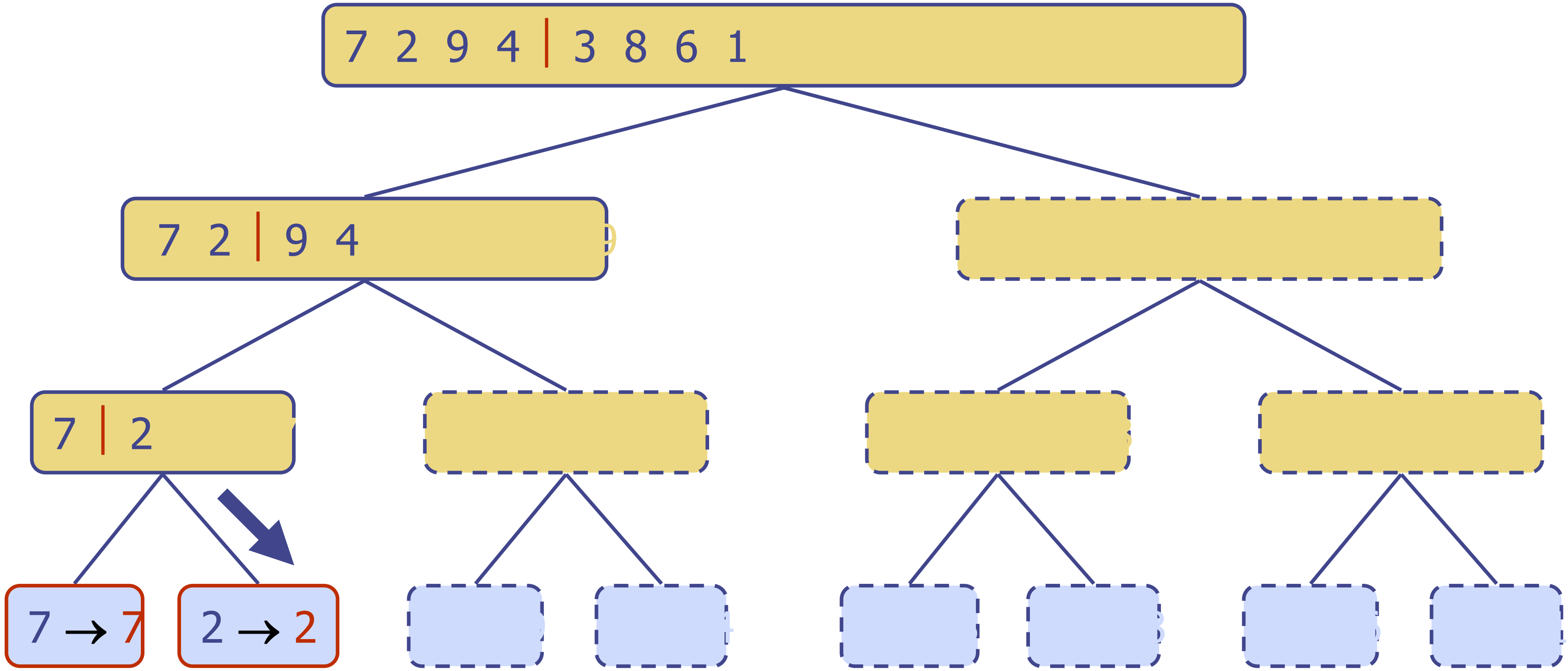
Execution Example (cont.)

- Recursive call, base case



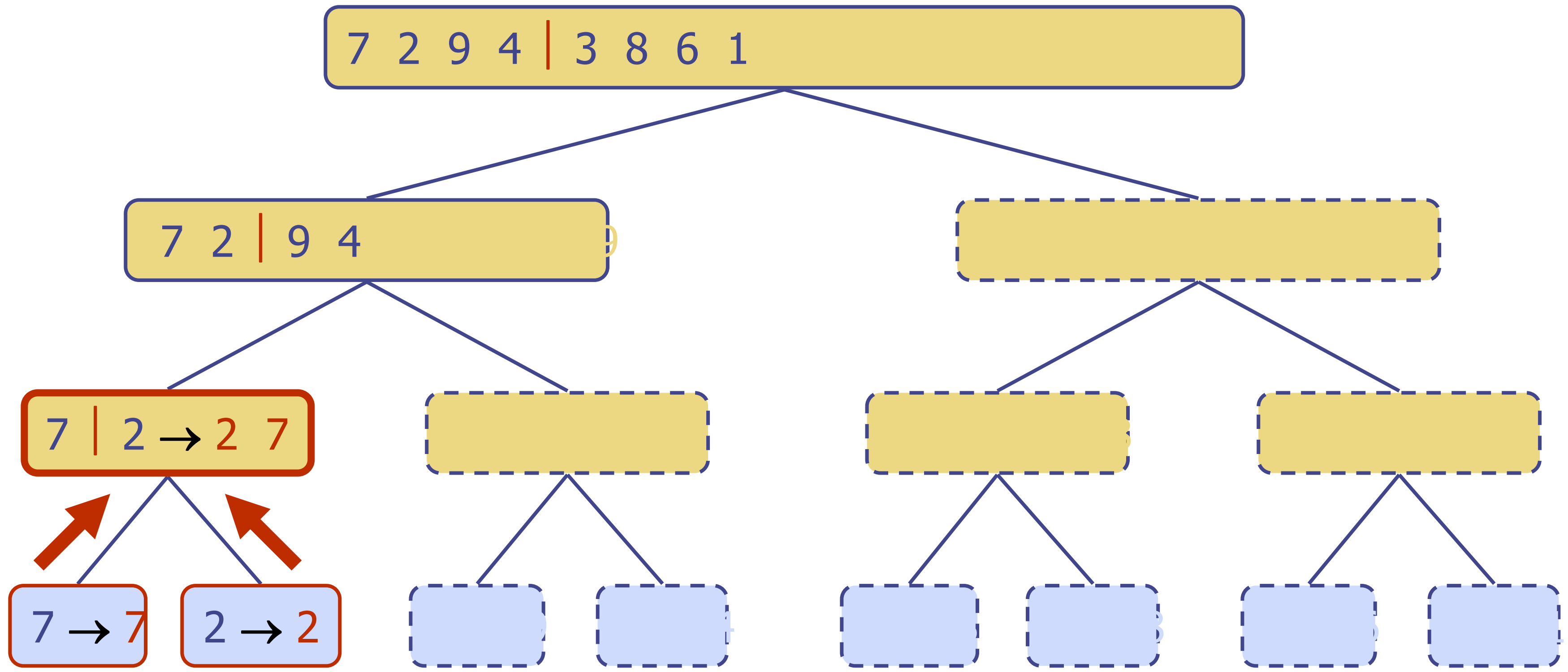
Execution Example (cont.)

- Recursive call, base case



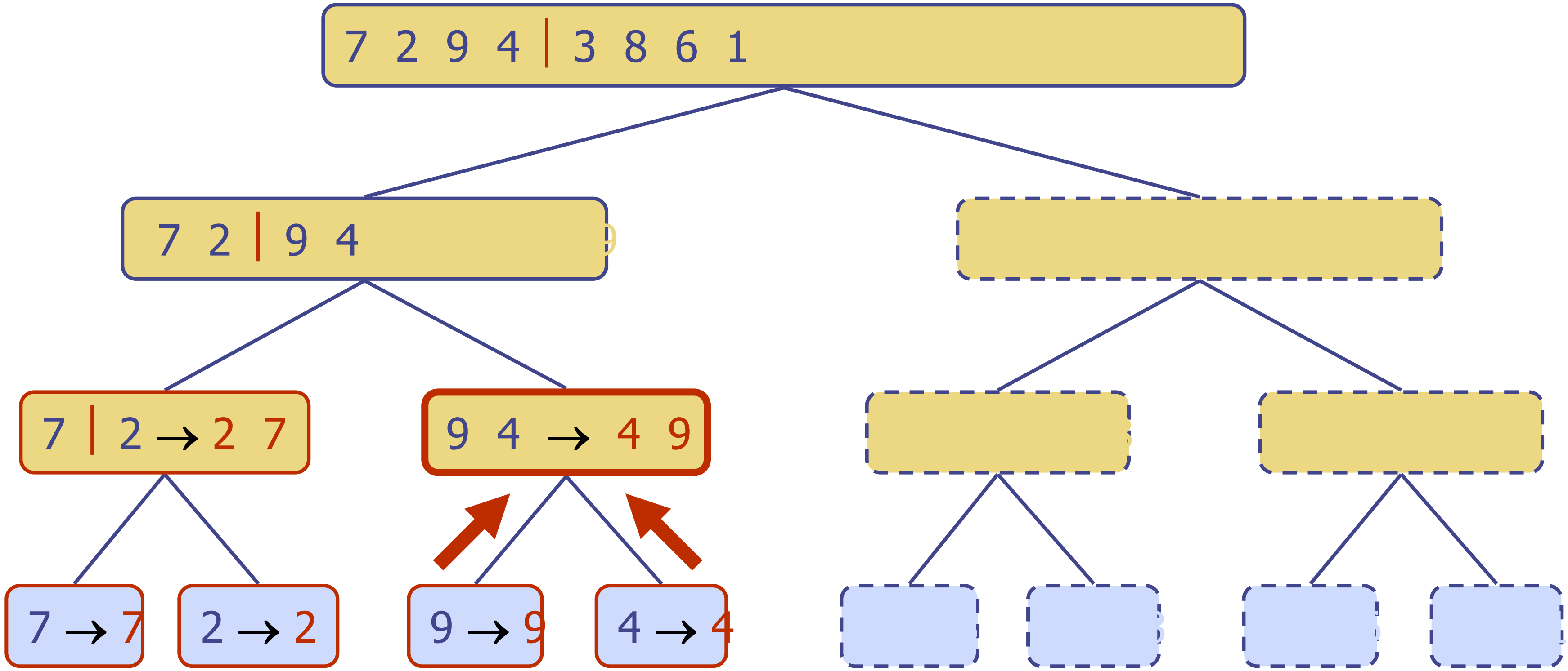
Execution Example (cont.)

- Merge



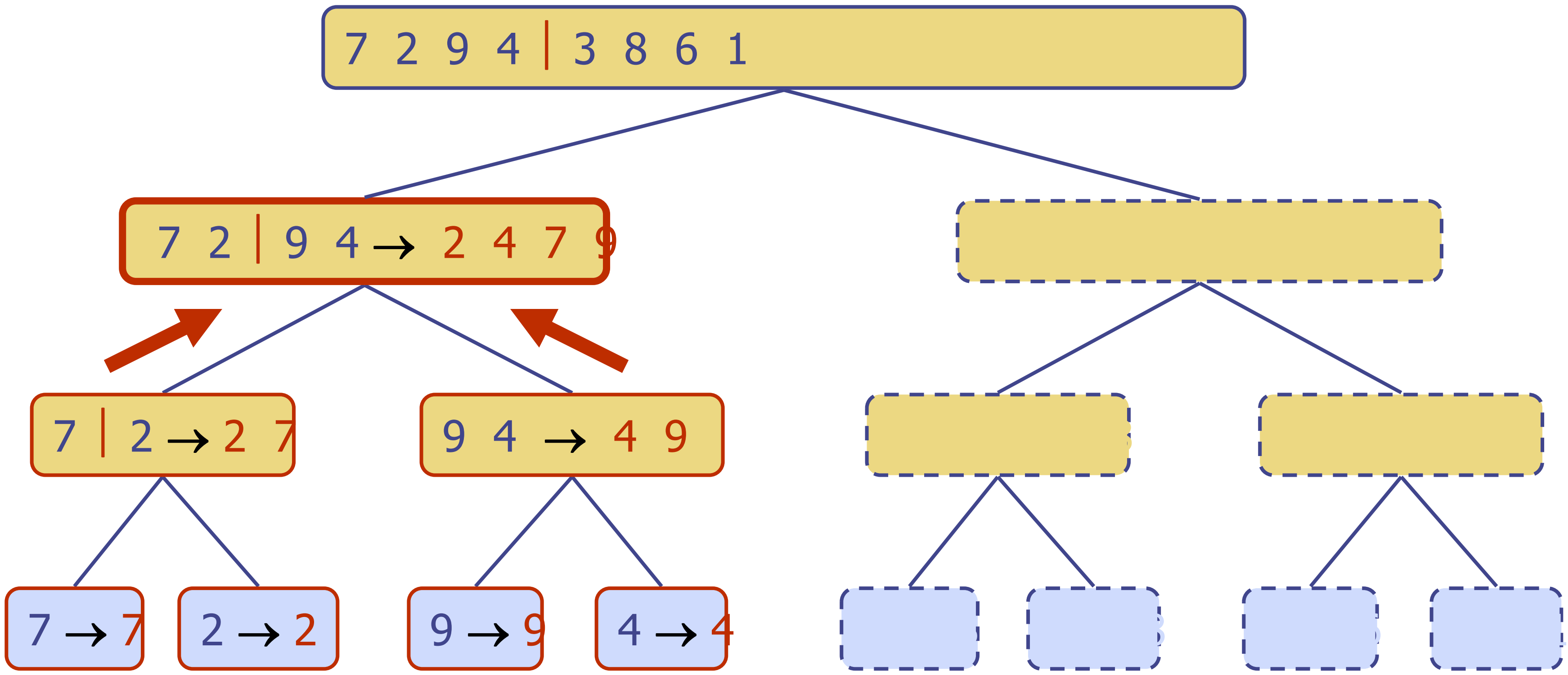
Execution Example (cont.)

- Recursive call, …, base case, merge



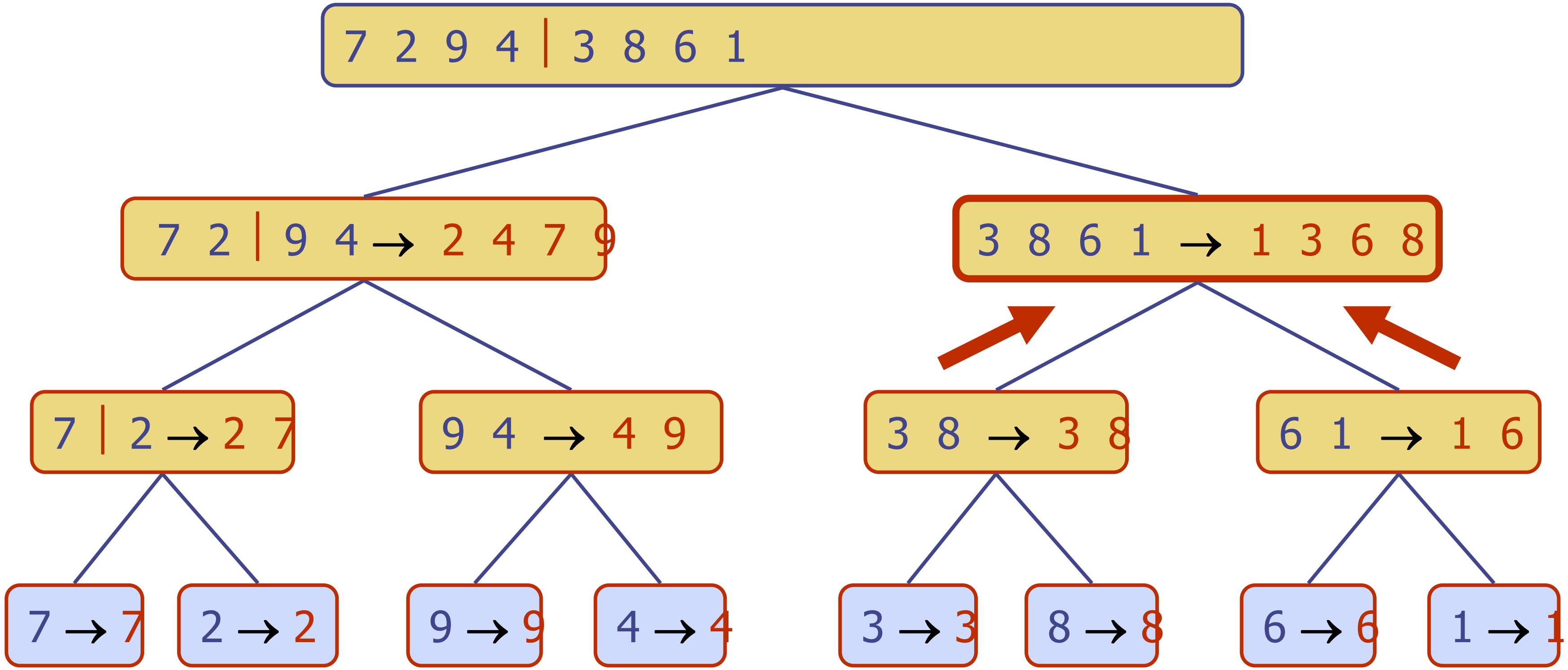
Execution Example (cont.)

- Merge



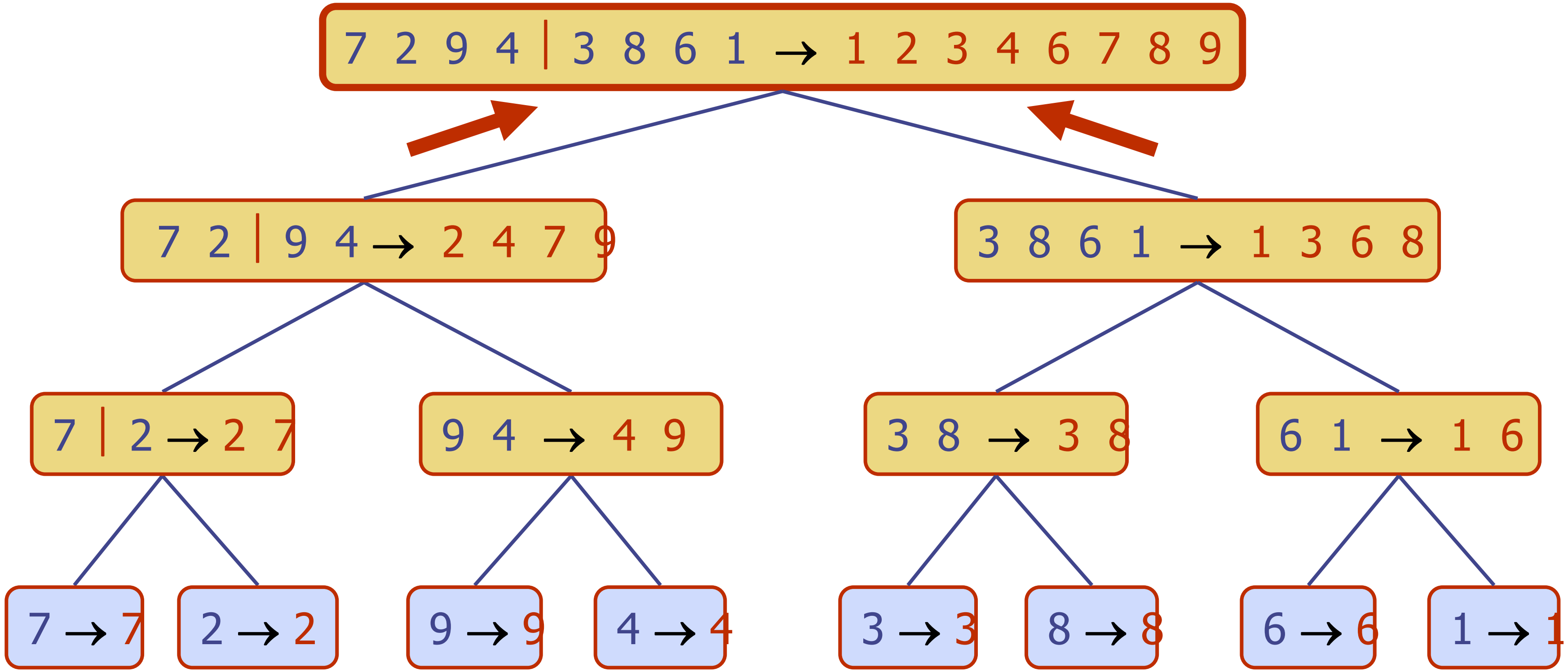
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $\boldsymbol{S}$ in two disjoint subsets $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - Recur: solve the subproblems associated with $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$
  - Conquer: combine the solutions for $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ into a solution for $\boldsymbol{S}$
- 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

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***

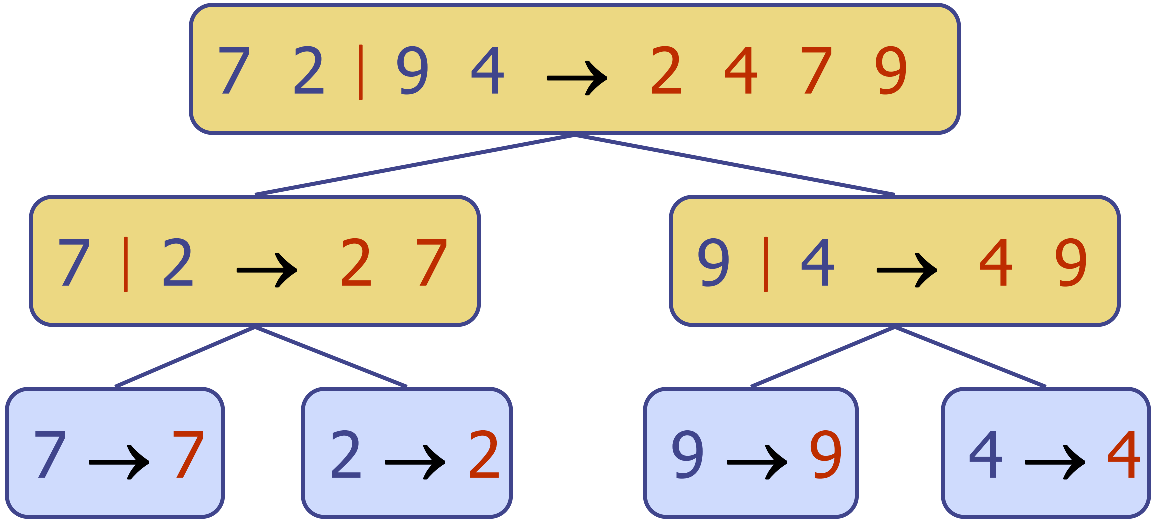
- Merging two sorted sequences, each with ***n***/2 elements and implemented by means of a doubly linked list, takes ***O***(***n***) time

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

  $S \leftarrow$ 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

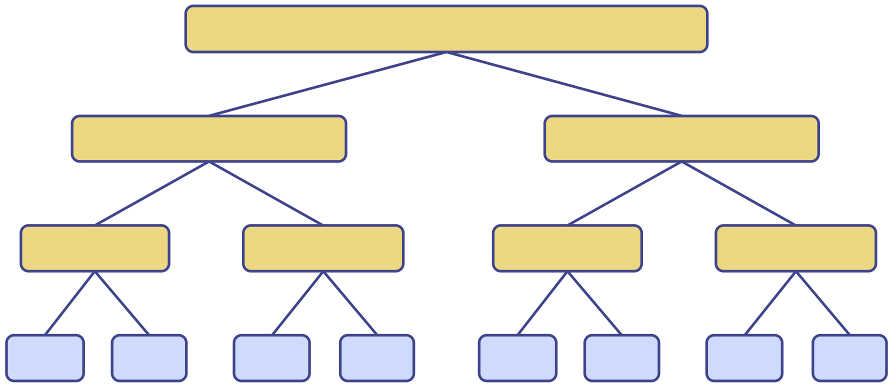
- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences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

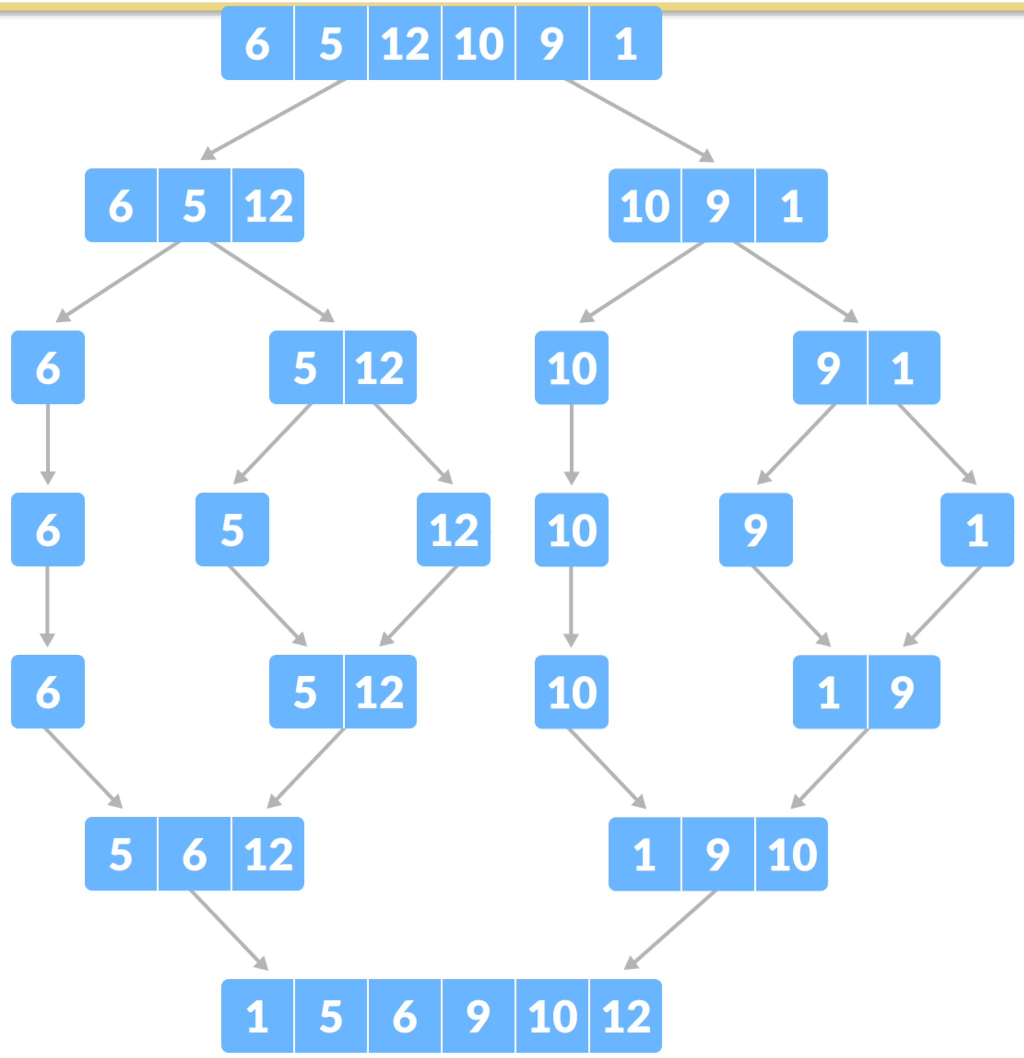


Analysis of Merge-Sort

- 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	$n/2^i$
…	…	…




6 5 12 10 9 1
6 5 12
10 9 1
6
5 12
10
9 1
6
5
12
10
9
1
6
5 12
10
1 9
5 6 12
1 9 10
1 5 6 9 10 12

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)