Topic 8
Sorting, Sets, and Selection
Lecture 8a - Sorting

CSCI 240

Data Structures and Algorithms

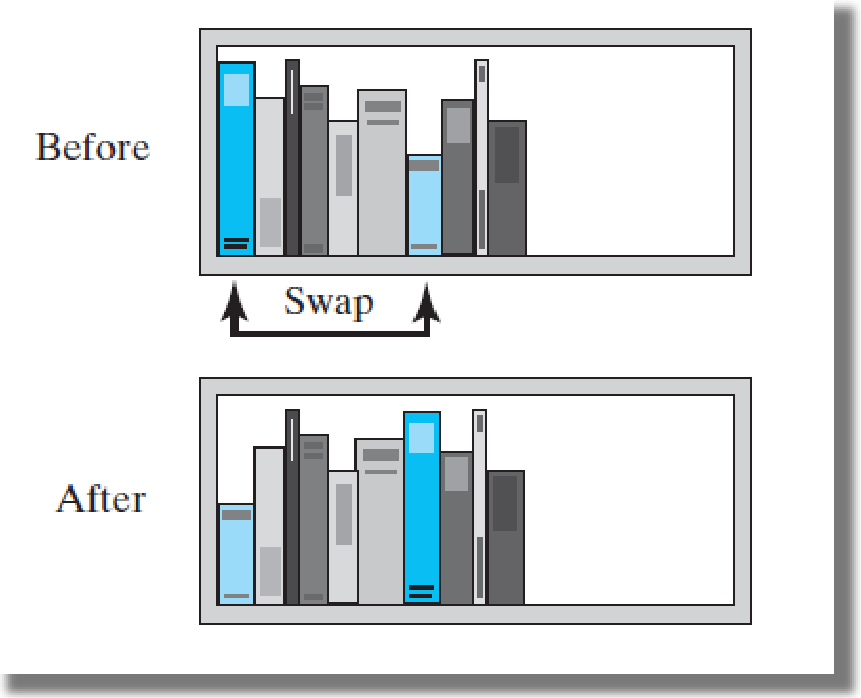
Prof. Dominick Atanasio

Sorting

- We seek algorithms to arrange items, ai
  such that
  a1 ≤ a2 ≤ . . . ≤ an
- Sorting an array is usually easier than sorting a Linke List
- Efficiency of a sorting algorithm is significant

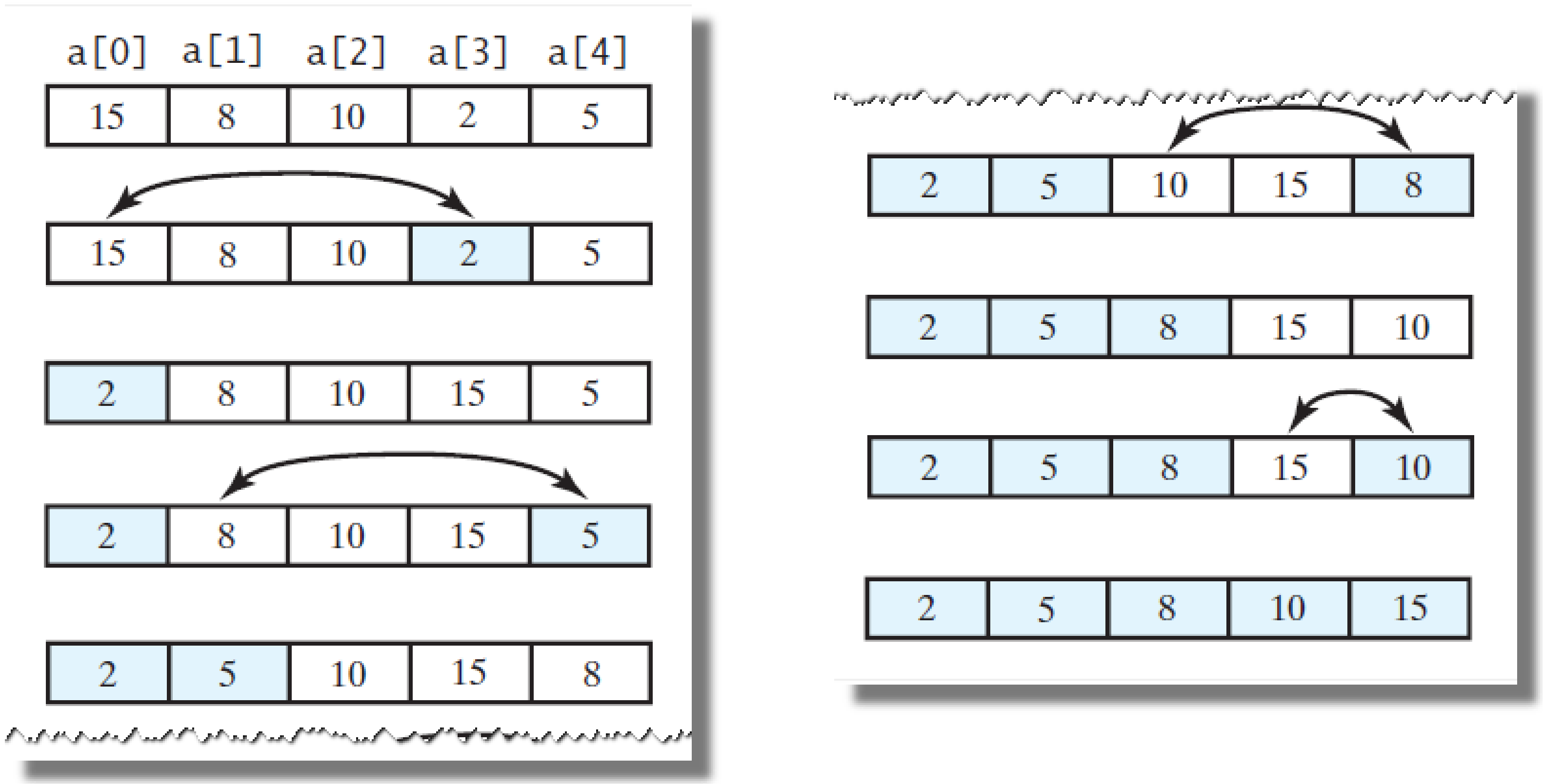
Selection Sort

- FIGURE 8-1 Before and after exchanging the shortest book and the first book

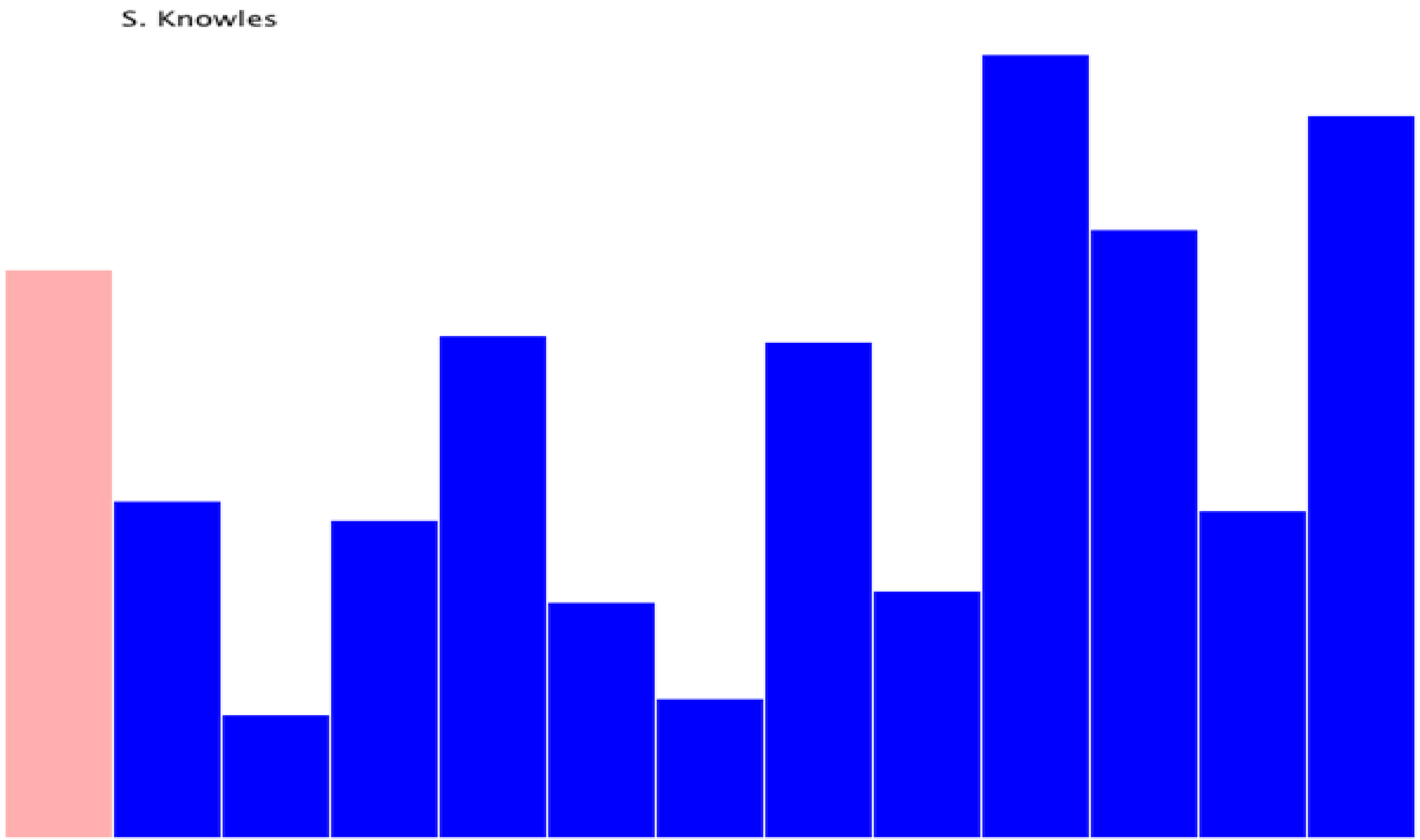


Selection Sort

- FIGURE 8-2 A selection sort of an array of integers into ascending order



Selection Sort



Iterative Selection Sort

- This pseudocode describes an iterative algorithm for the selection sort

```
Algorithm selectionSort(a, n)
// Sorts the first n entries of an array a.

for (index = 0; index < n - 1; index++)
{
    indexOfNextSmallest = the index of the smallest value among
                          a[index], a[index + 1], . . . , a[n - 1]
    Interchange the values of a[index] and a[indexOfNextSmallest]
    // Assertion: a[0] ≤ a[1] ≤ . . . ≤ a[index], and these are the smallest
    // of the original array entries. The remaining array entries begin at a[index + 1].
}
```

Recursive Selection Sort

- Recursive selection sort algorithm

```
Algorithm selectionSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (first < last)
{
   indexOfNextSmallest = the index of the smallest value among
                         a[first], a[first + 1], . . . , a[last]
   Interchange the values of a[first] and a[indexOfNextSmallest]
   // Assertion: a[0] ≤ a[1] ≤ . . . ≤ a[first] and these are the smallest
   // of the original array entries. The remaining array entries begin at a[first + 1].
   selectionSort(a, first + 1, last)
}
```

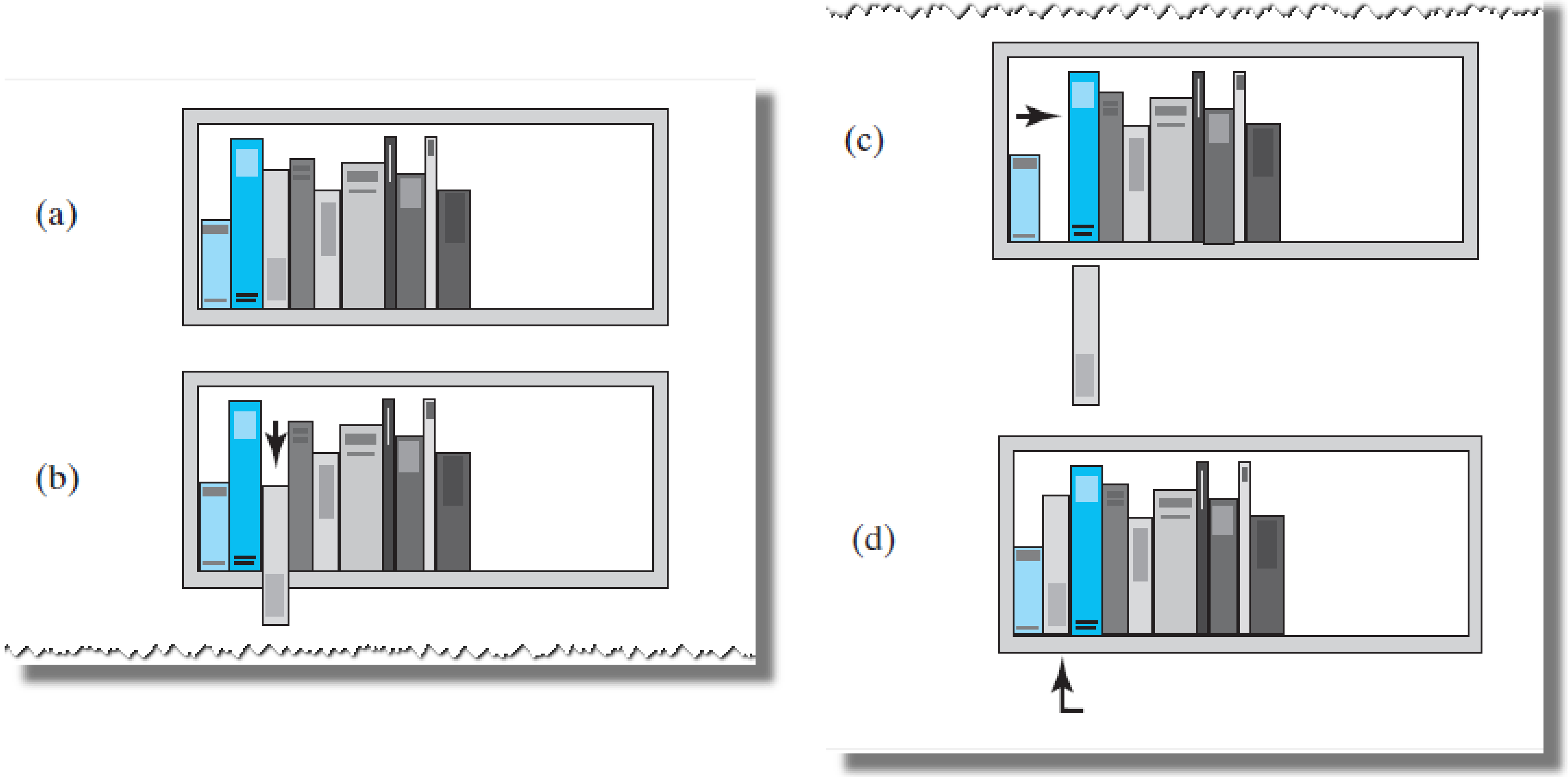
Efficiency of Selection Sort?

Efficiency of Selection Sort

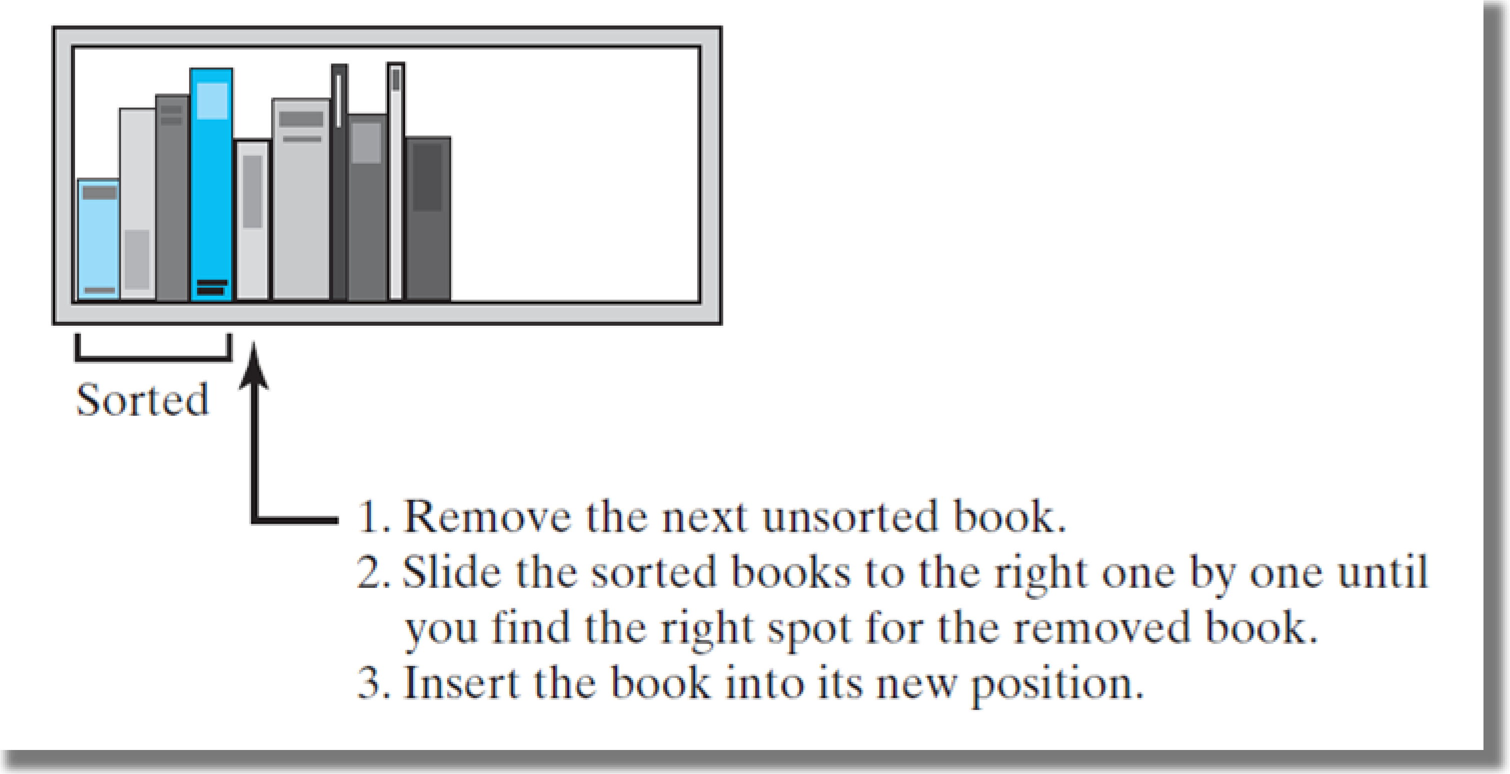
- Selection sort is O(n2) regardless of the initial order of the entries.
  - Requires $O(n^2)$ comparisons
  - Does only $O(n)$ swaps

Insertion Sort

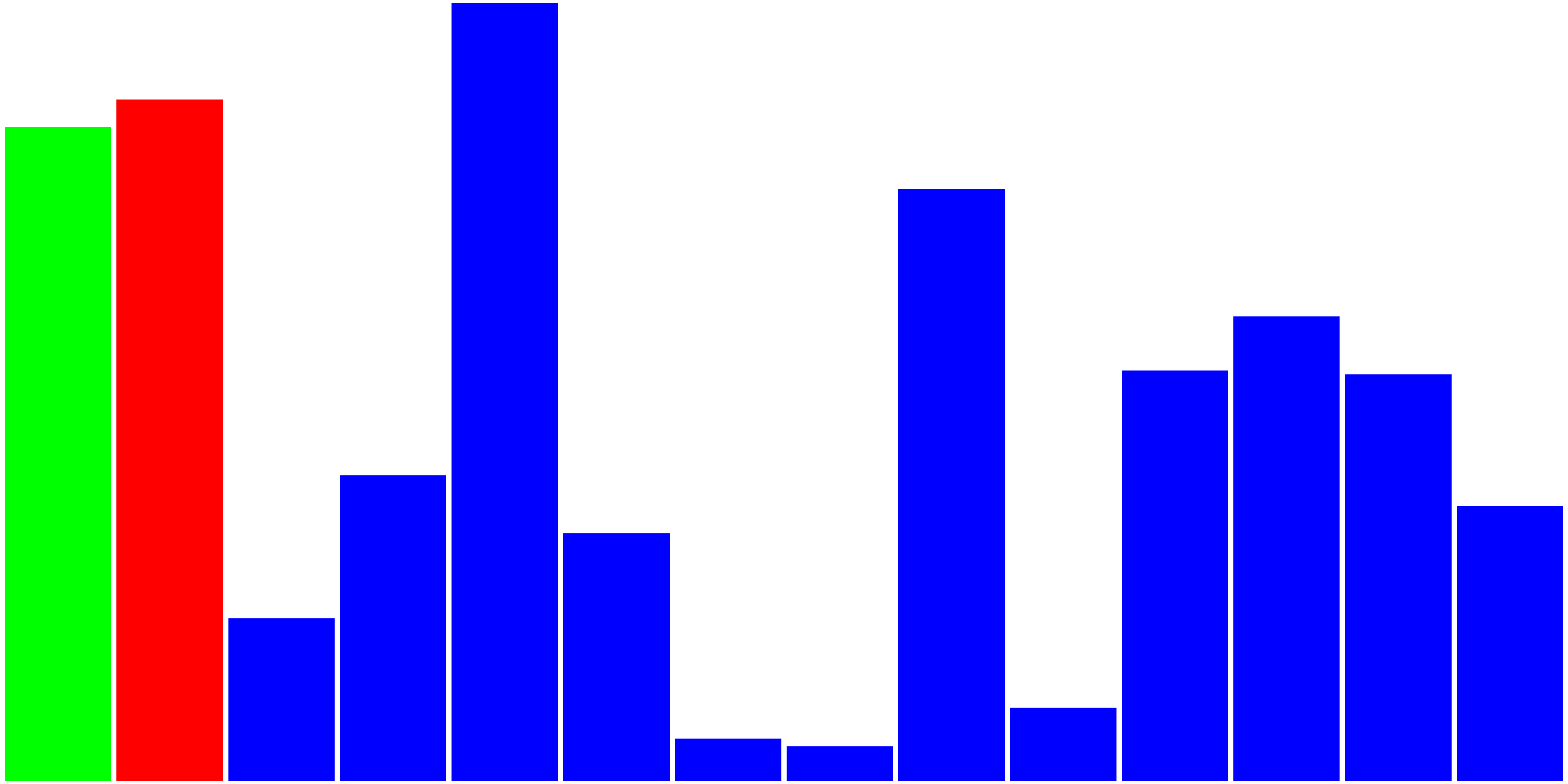
- FIGURE 8-3 The placement of the third book during an insertion sort



- FIGURE 8-4 An insertion sort of books



Insertion Sort



Iterative Insertion Sort

- Iterative algorithm describes an insertion sort of the entries at indices first through last of the a

```
Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] iteratively.

for (unsorted = first + 1 through last)
{
   nextToInsert = a[unsorted]
   insertInOrder(nextToInsert, a, first, unsorted - 1)
}
```

Iterative Insertion Sort

- Pseudocode of method, insertInOrder, to perform the insertions.

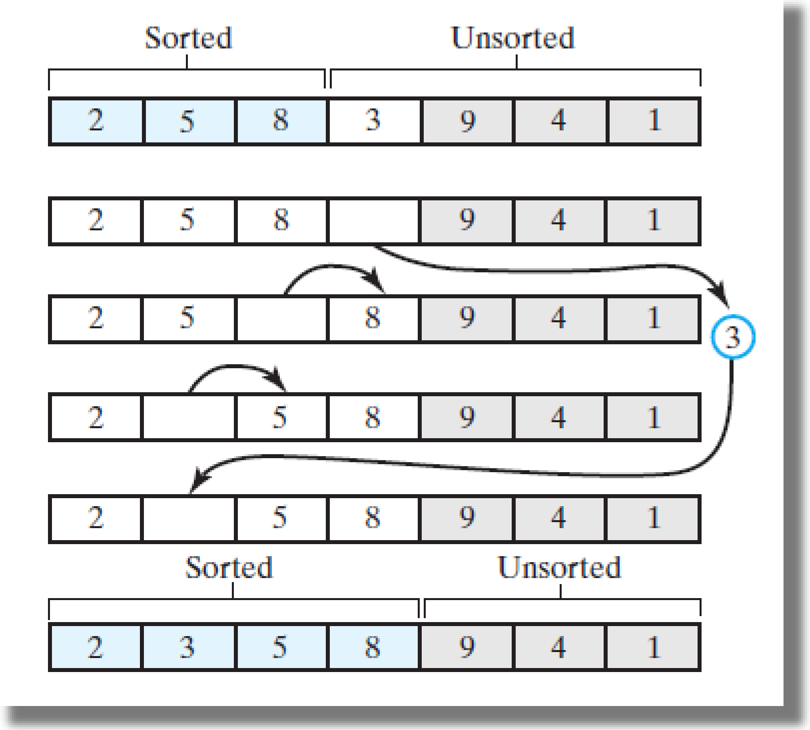
```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted entries a[begin] through a[end].

index = end                    // Index of last entry in the sorted portion
// Make room, if needed, in sorted portion for another entry
while ( (index >= begin) and (anEntry < a[index]) )
{
   a[index + 1] = a[index] // Make room
   index--
}
// Assertion: a[index + 1] is available.

a[index + 1] = anEntry      // Insert
```

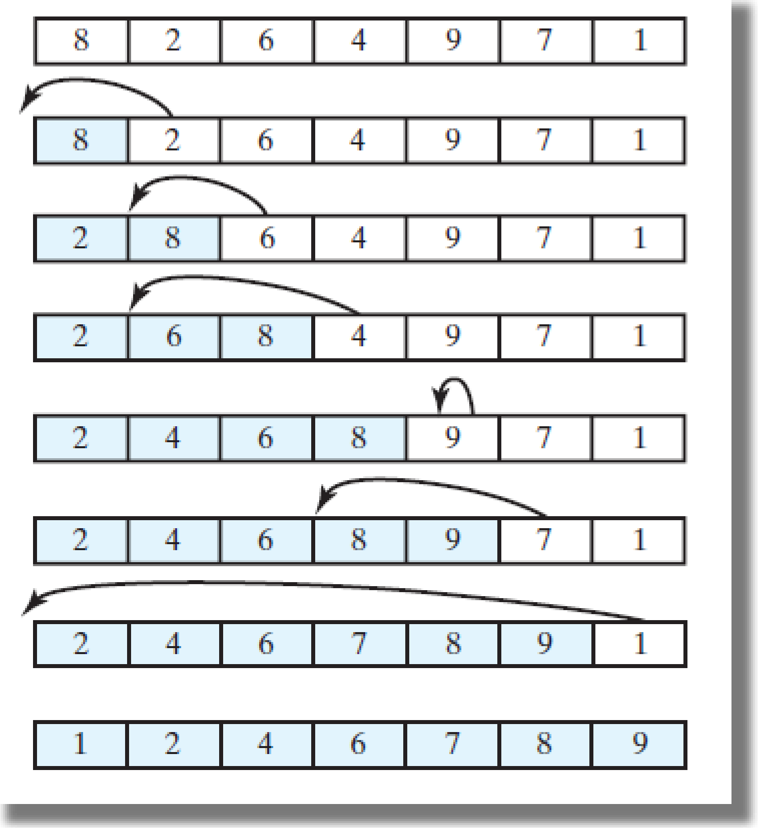
Iterative Insertion Sort

- FIGURE 8-5 Inserting the next unsorted entry into its proper location within the sorted portion of an array during an insertion sort



Iterative Insertion Sort

- FIGURE 8-6 An insertion sort of an array of integers into ascending order



Recursive Insertion Sort

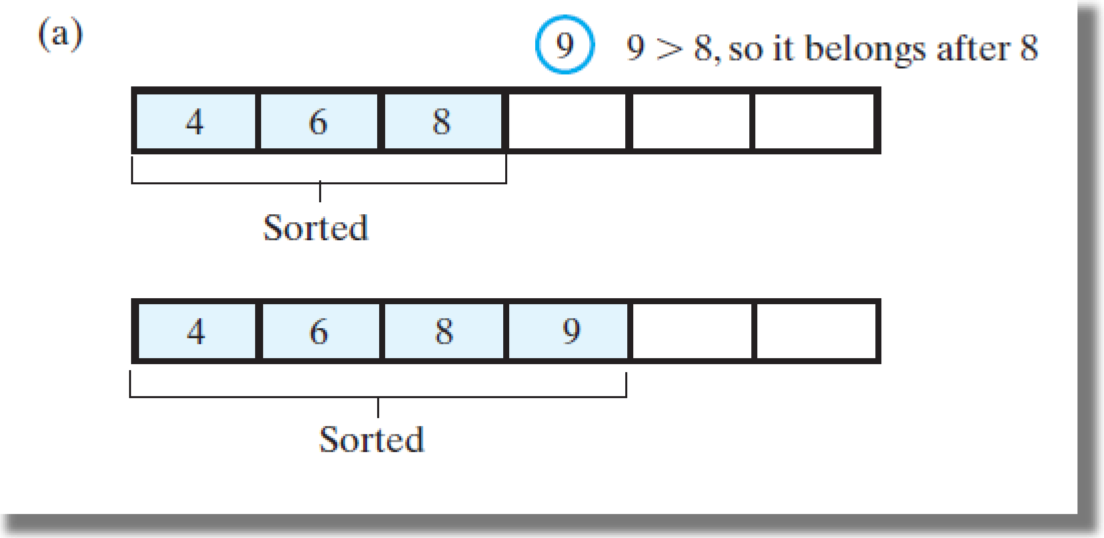
- This pseudocode describes a recursive insertion sort.

```
Algorithm insertionSort(a, first, last)
// Sorts the array entries a[first] through a[last] recursively.

if (the array contains more than one entry)
{
    Sort the array entries a[first] through a[last - 1]
    Insert the last entry a[last] into its correct sorted position within the rest of the array
}
```

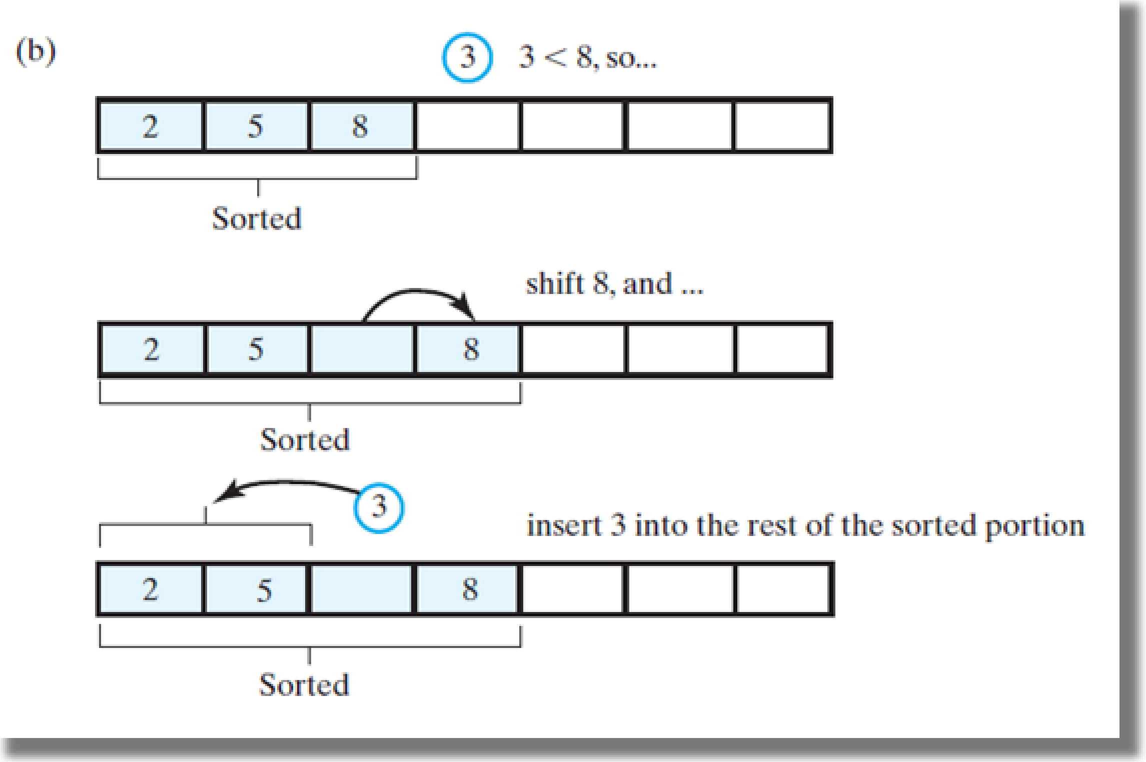
Recursive Insertion Sort

- FIGURE 8-8 Inserting the first unsorted entry into the sorted portion of the array. (a) The entry is greater than or equal to the last sorted entry



Recursive Insertion Sort

- FIGURE 8-8 Inserting the first unsorted entry into the sorted portion of the array. (b) the entry is smaller than the last sorted entry



Recursive Insertion Sort

- The algorithm insertInOrder
- Note: insertion sort efficiency (worst case) is $O(n^2)$

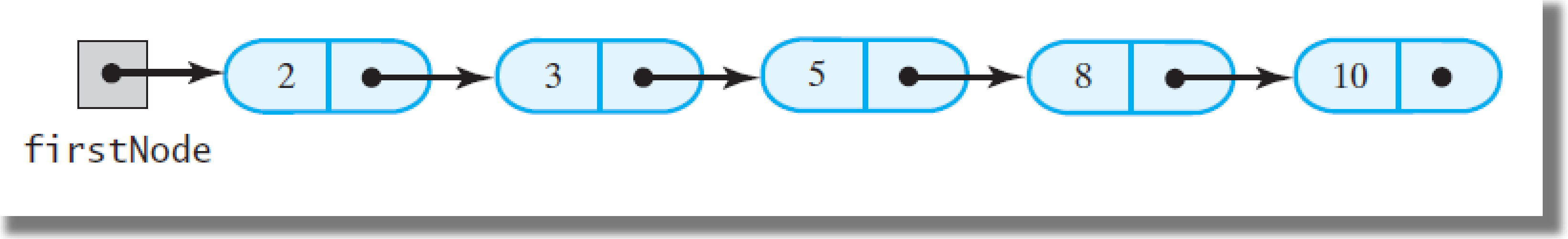
```
Algorithm insertInOrder(anEntry, a, begin, end)
// Inserts anEntry into the sorted array entries a[begin] through a[end].
// Revised draft.

if (anEntry >= a[end])
   a[end + 1] = anEntry

   else if (begin < end)
   {
      a[end + 1] = a[end]
      insertInOrder(anEntry, a, begin, end - 1)
   }
   else // begin == end and anEntry < a[end]
   {
      a[end + 1] = a[end]
      a[end] = anEntry
   }
```

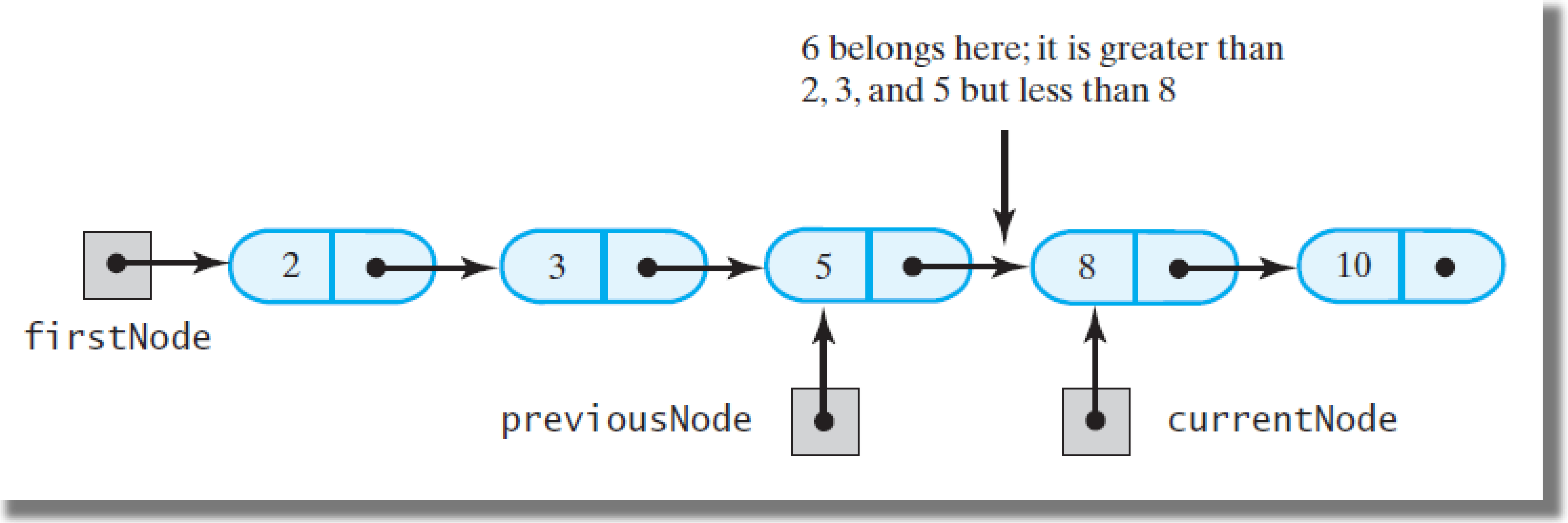
Insertion Sort of a Linked List

- FIGURE 8-8 A chain of integers sorted into ascending order



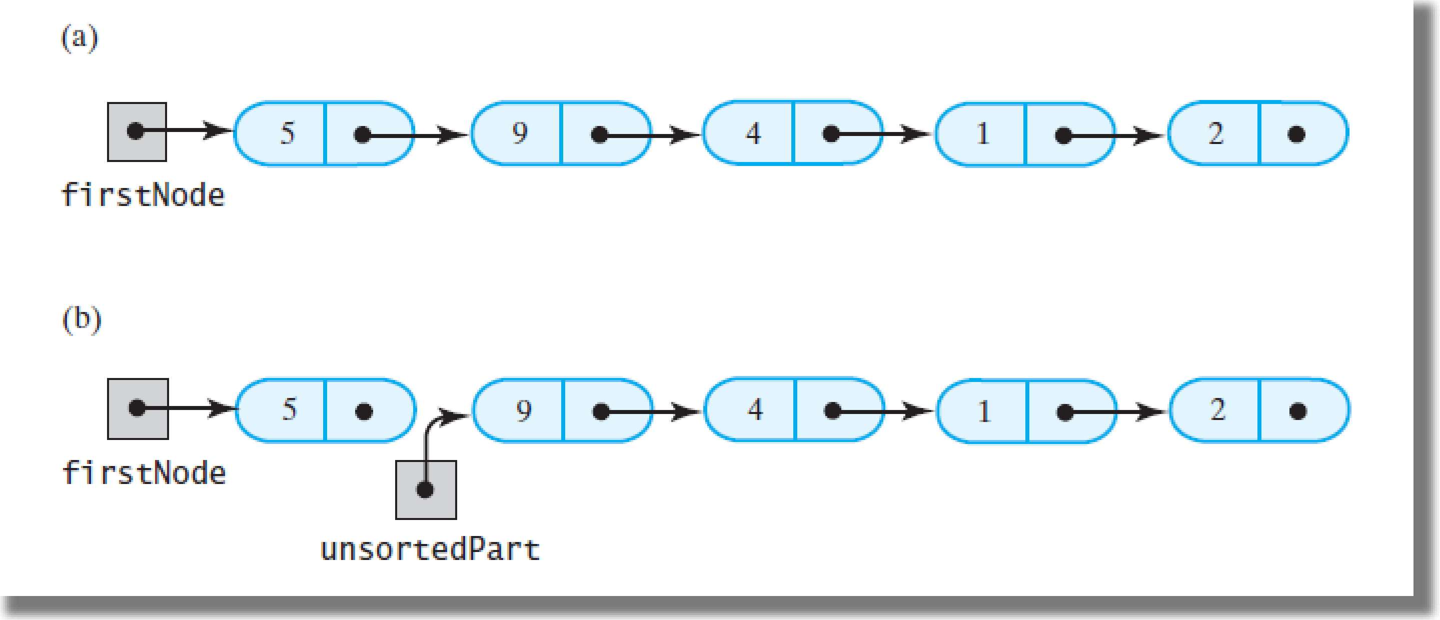
Insertion Sort of a Linked List

- FIGURE 8-9 During the traversal of a chain to locate the insertion point, save a reference to the node before the current one

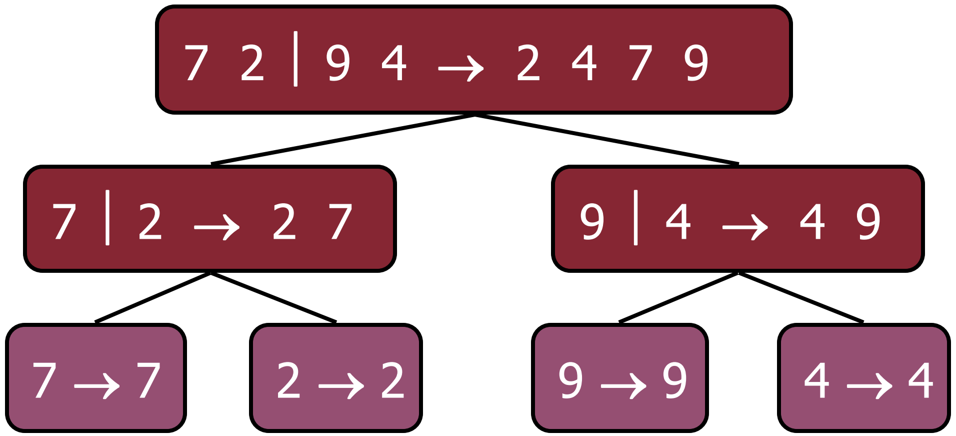


Insertion Sort of a Linked List

- FIGURE 8-10 Breaking a chain of nodes into two pieces as the first step in an insertion sort:
- (a) the original chain;
  (b) the two pieces



Merge Sort



Divide-and-Conquer (§ 10.1.1)

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two disjoint subsets S1 and S2
  - Recur: solve the subproblems associated with S1 and S2
  - Conquer: combine the solutions for S1 and S2 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 accesses data in a sequential manner (suitable to sort data on a disk)

Merge-Sort (§ 10.1)

- 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

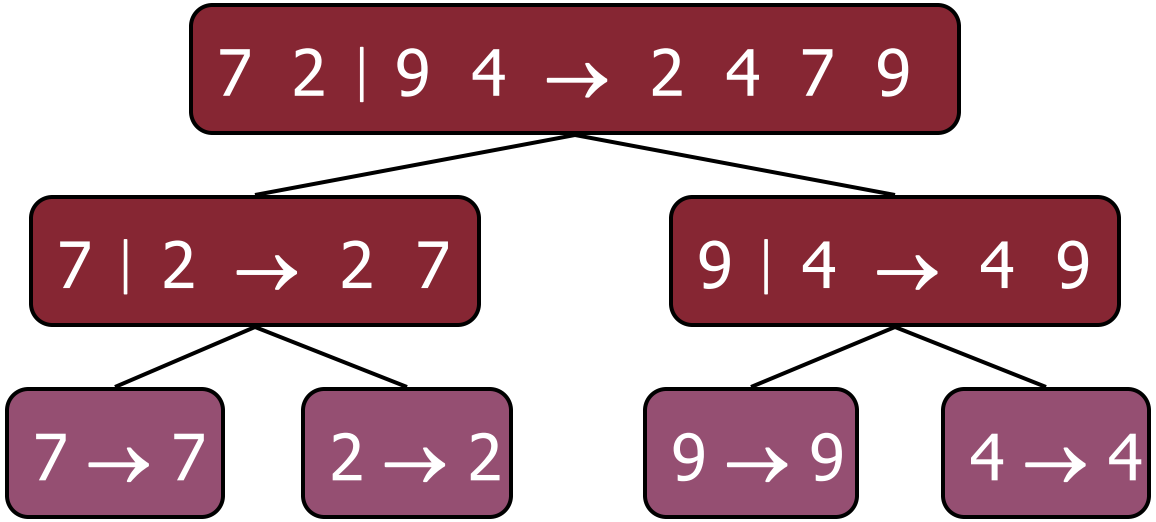
Output sorted sequence of $A \cup B$

$S \leftarrow$ 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

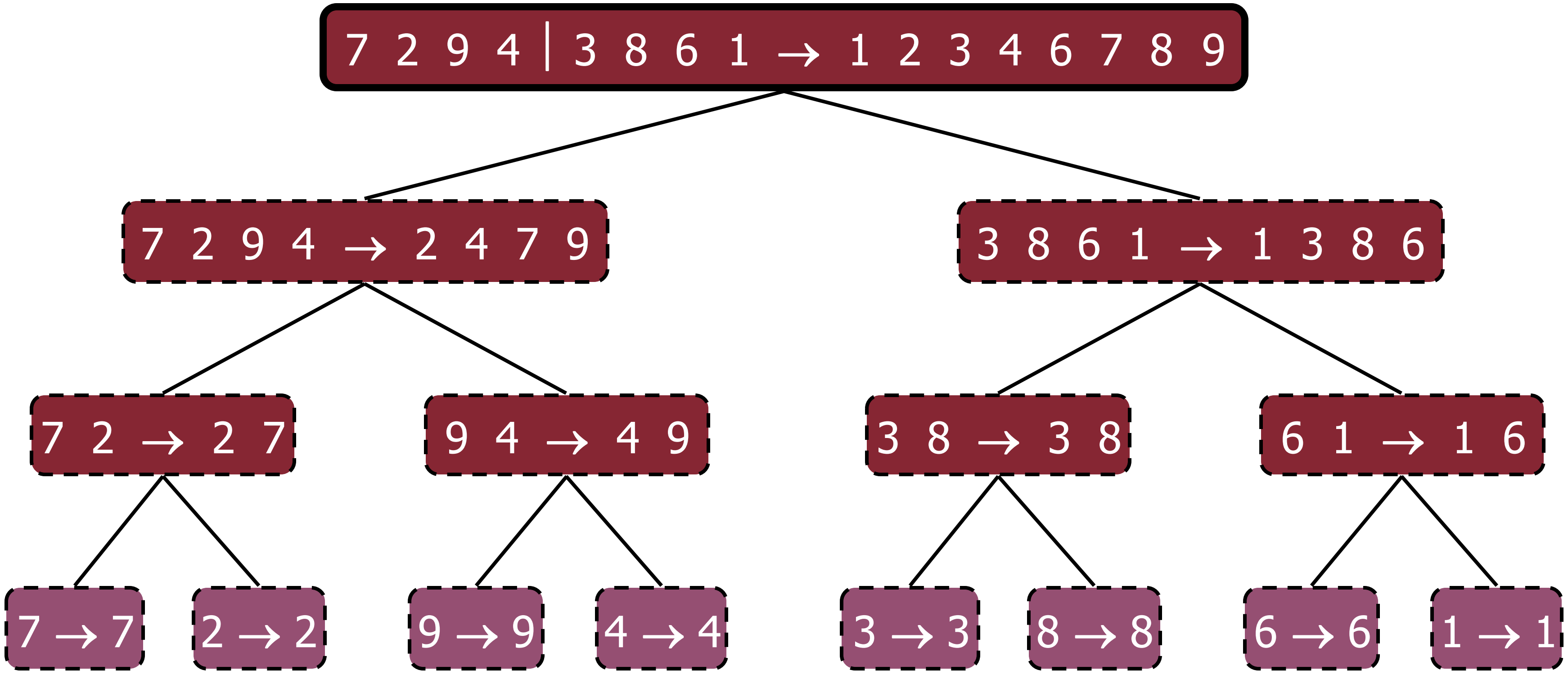
Merge-Sort Tree

- 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



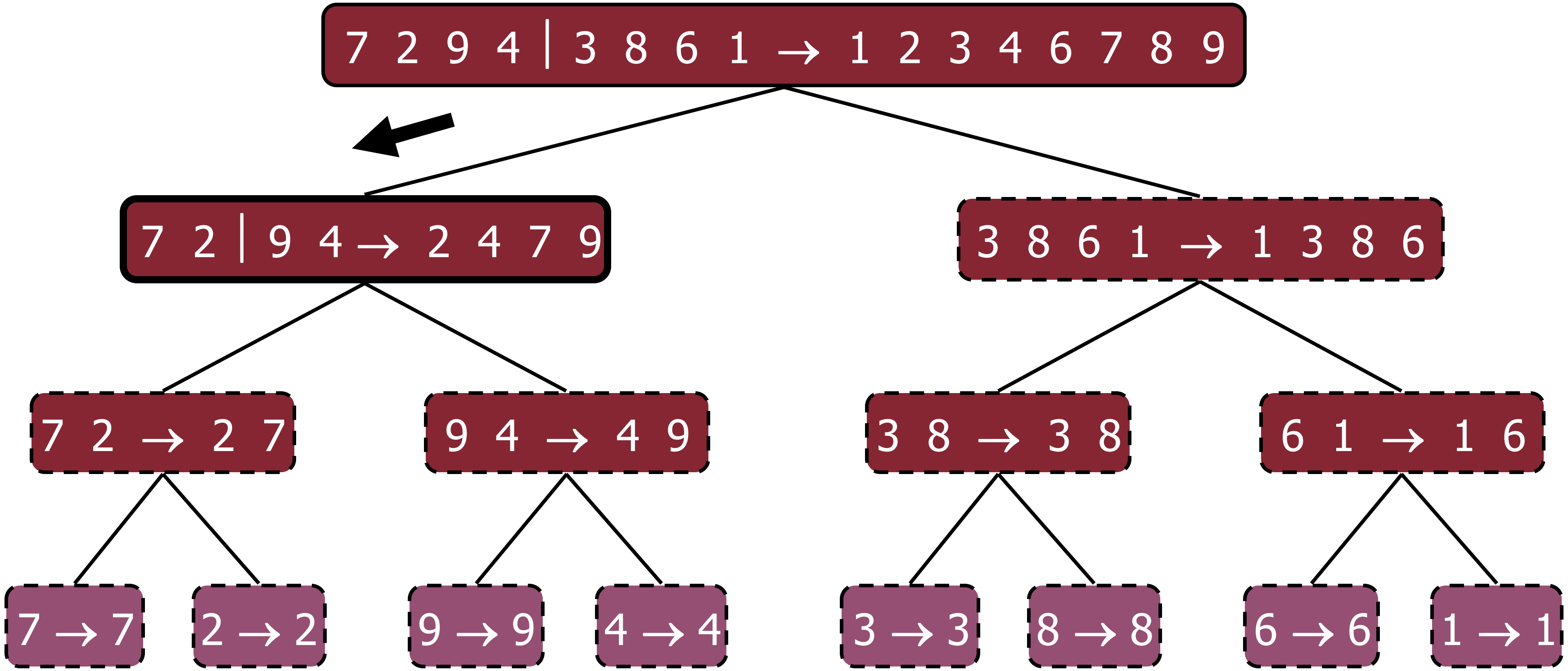
Execution Example

- Partition



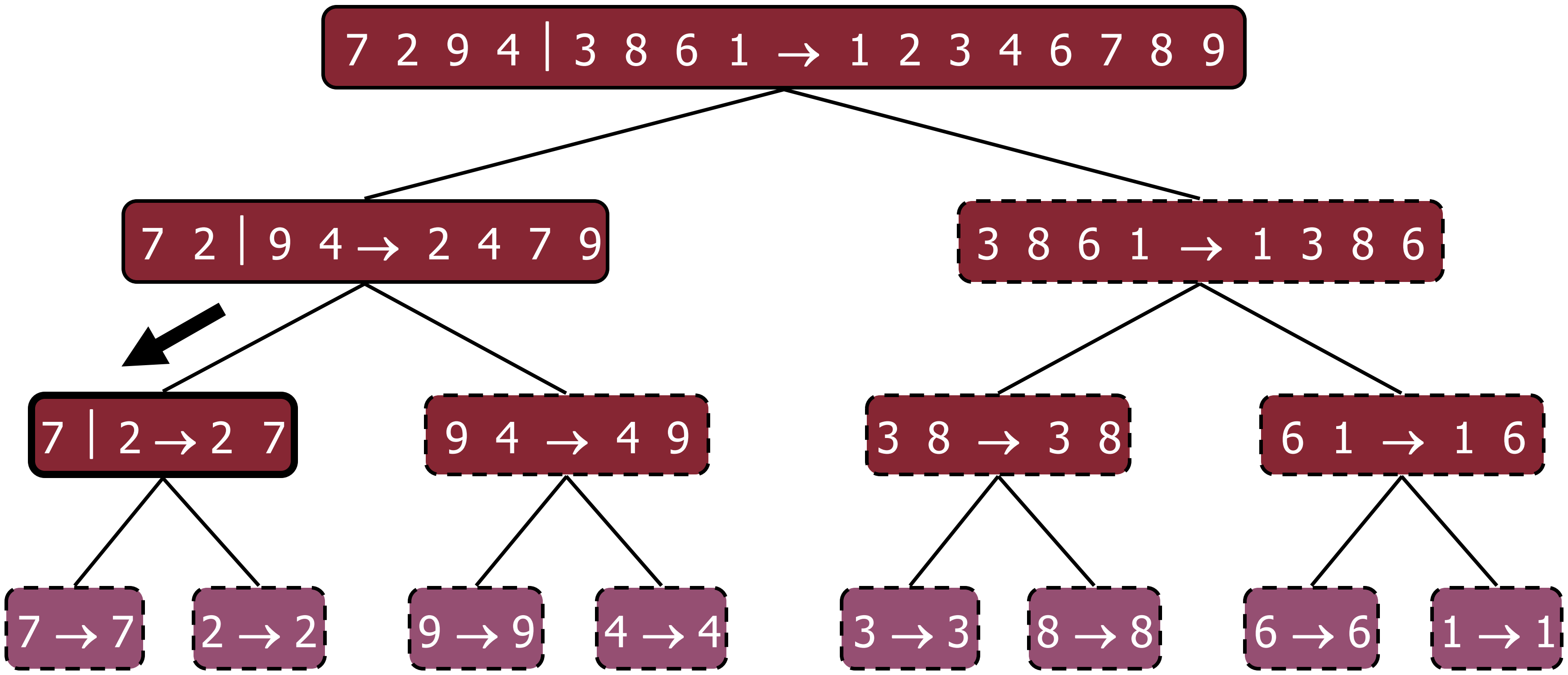
Execution Example (cont.)

- Recursive call, partition



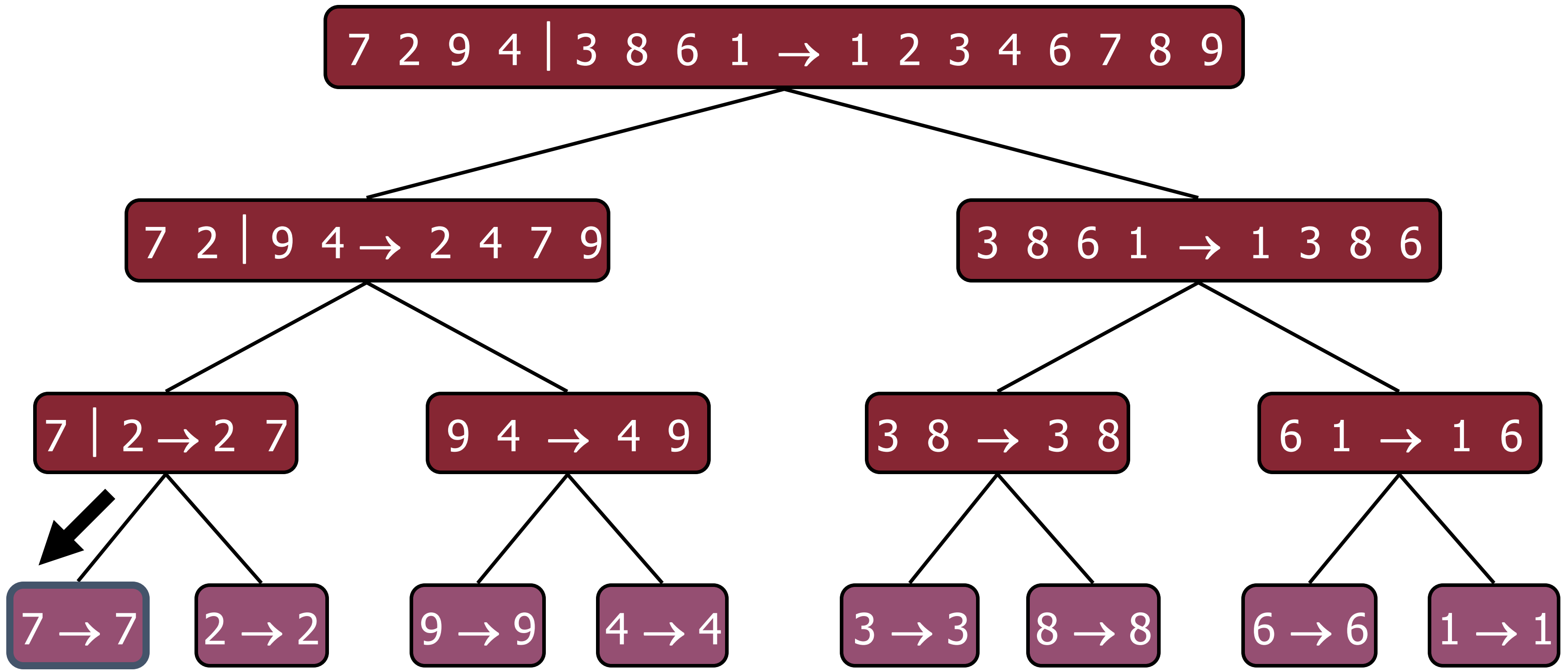
Execution Example (cont.)

- Recursive call, partition



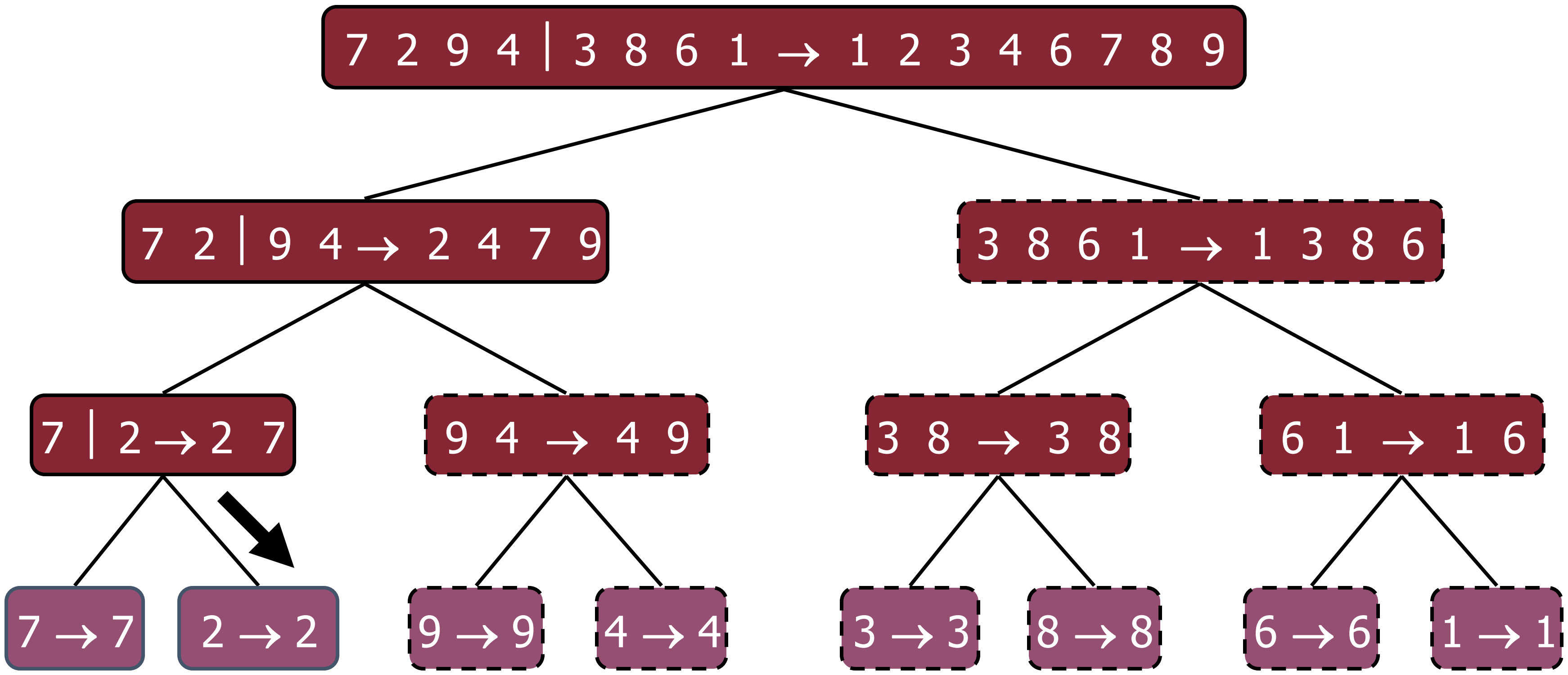
Execution Example (cont.)

- Recursive call, base case



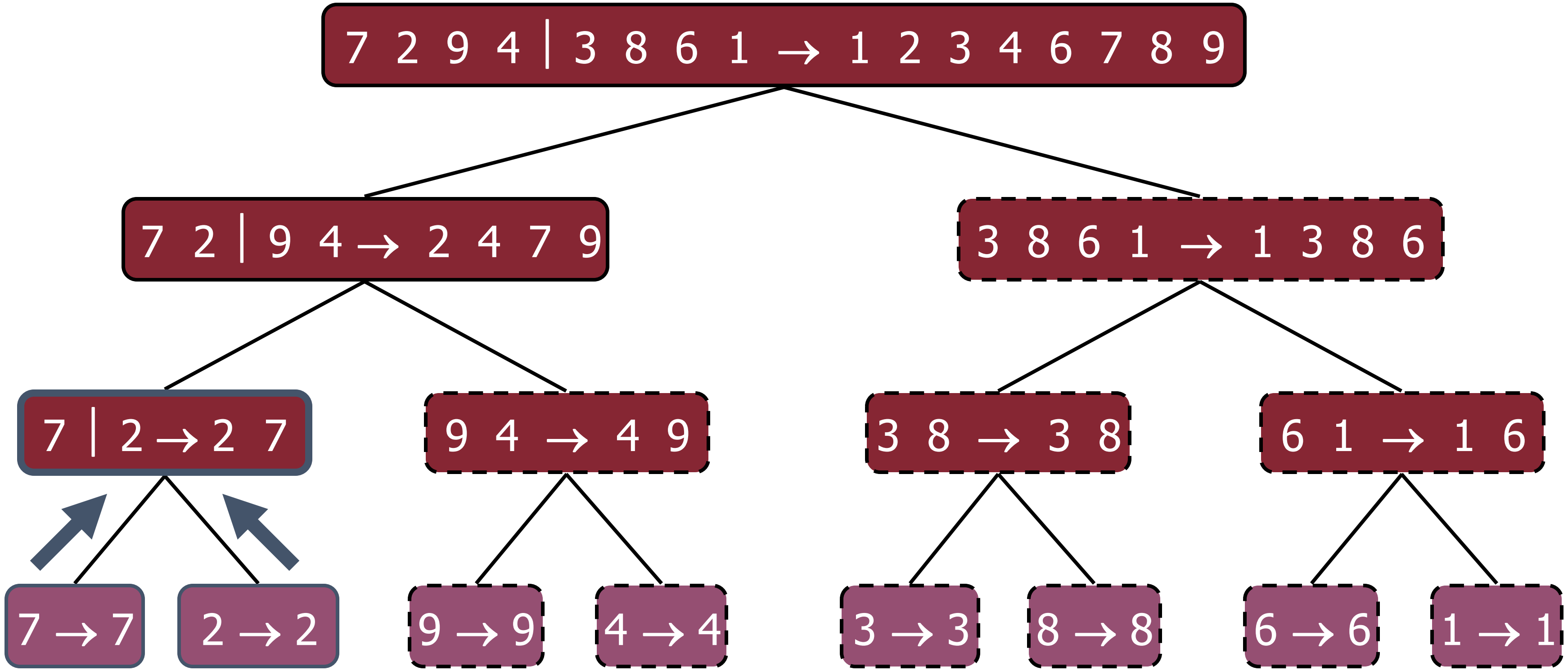
Execution Example (cont.)

- Recursive call, base case



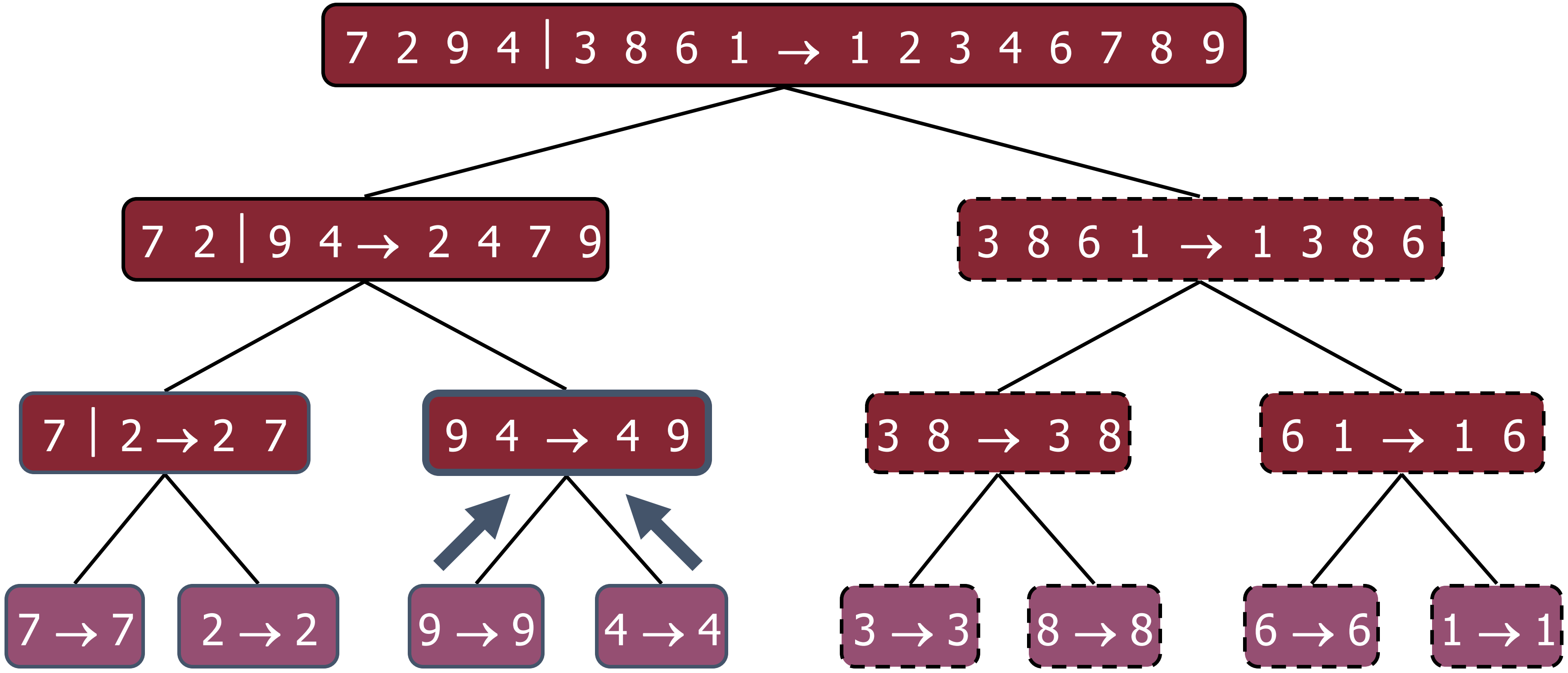
Execution Example (cont.)

- Merge



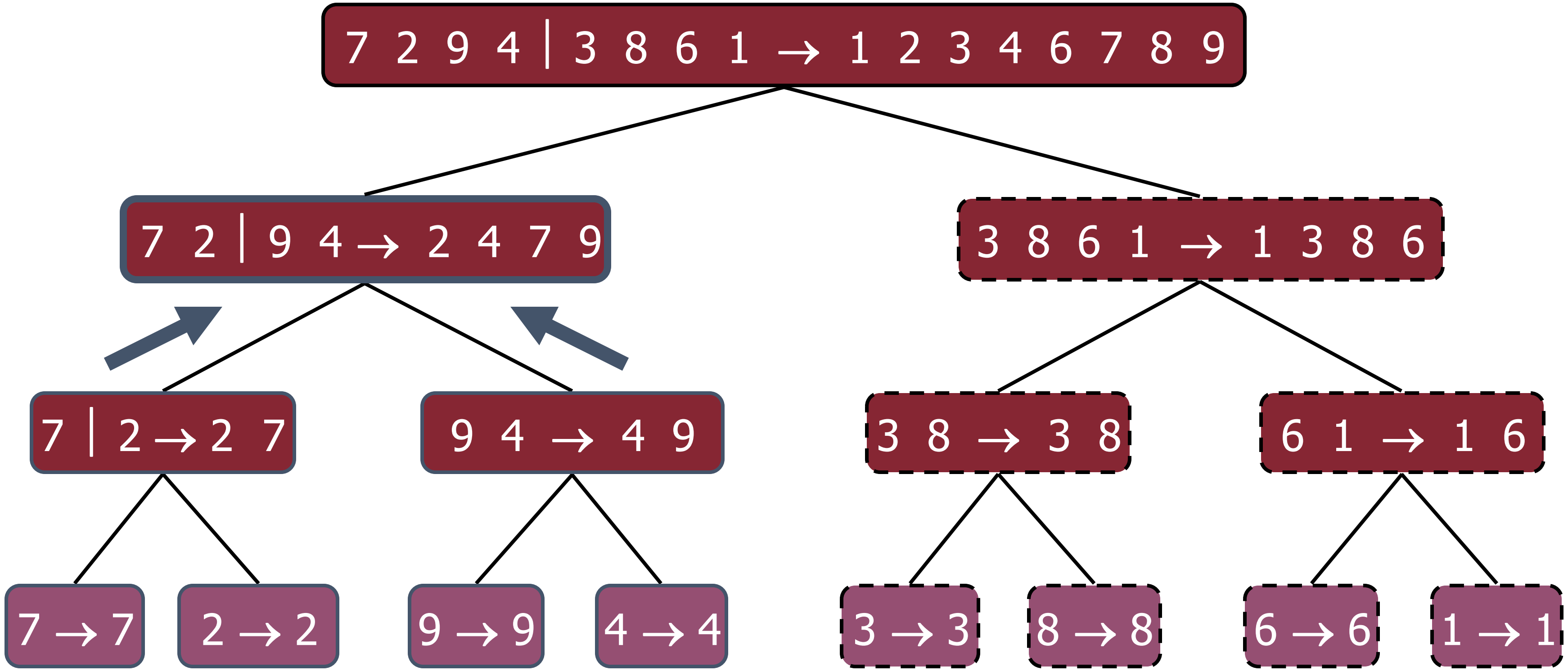
Execution Example (cont.)

- Recursive call, …, base case, merge



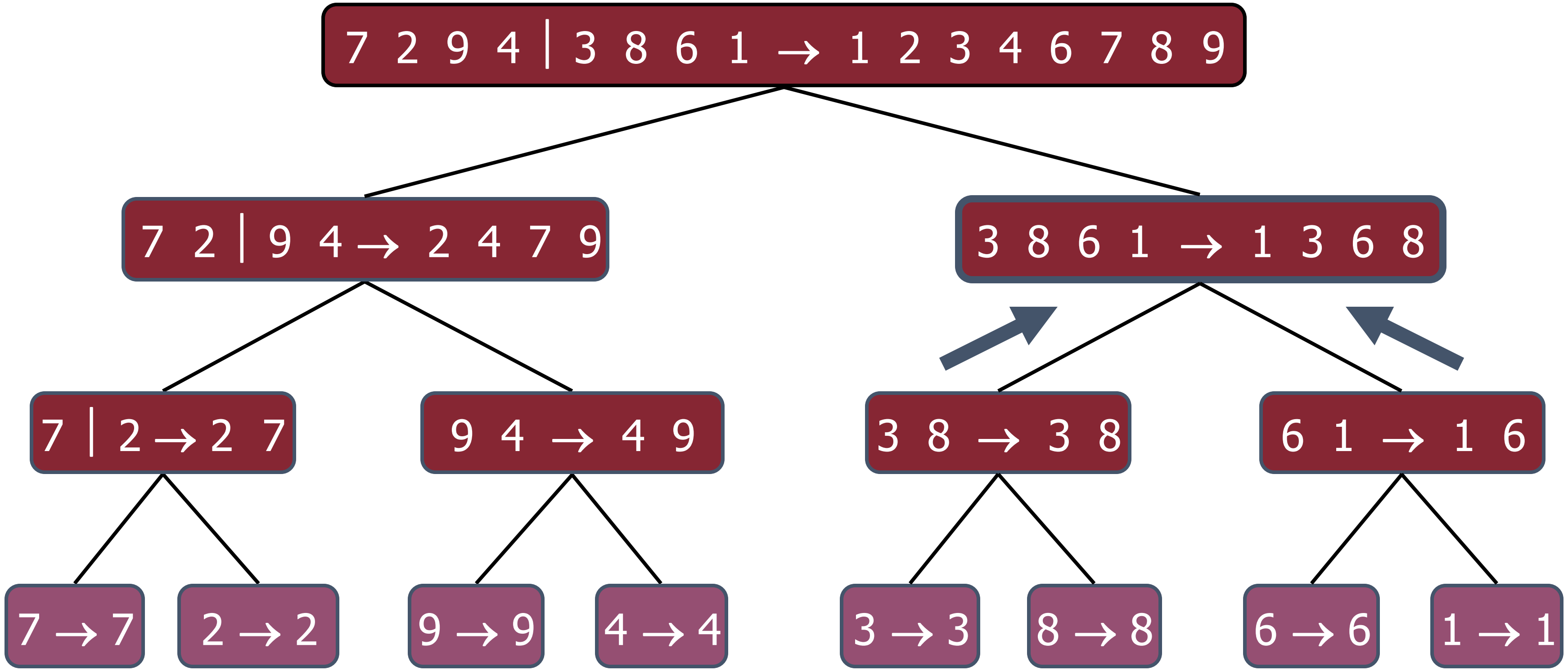
Execution Example (cont.)

- Merge



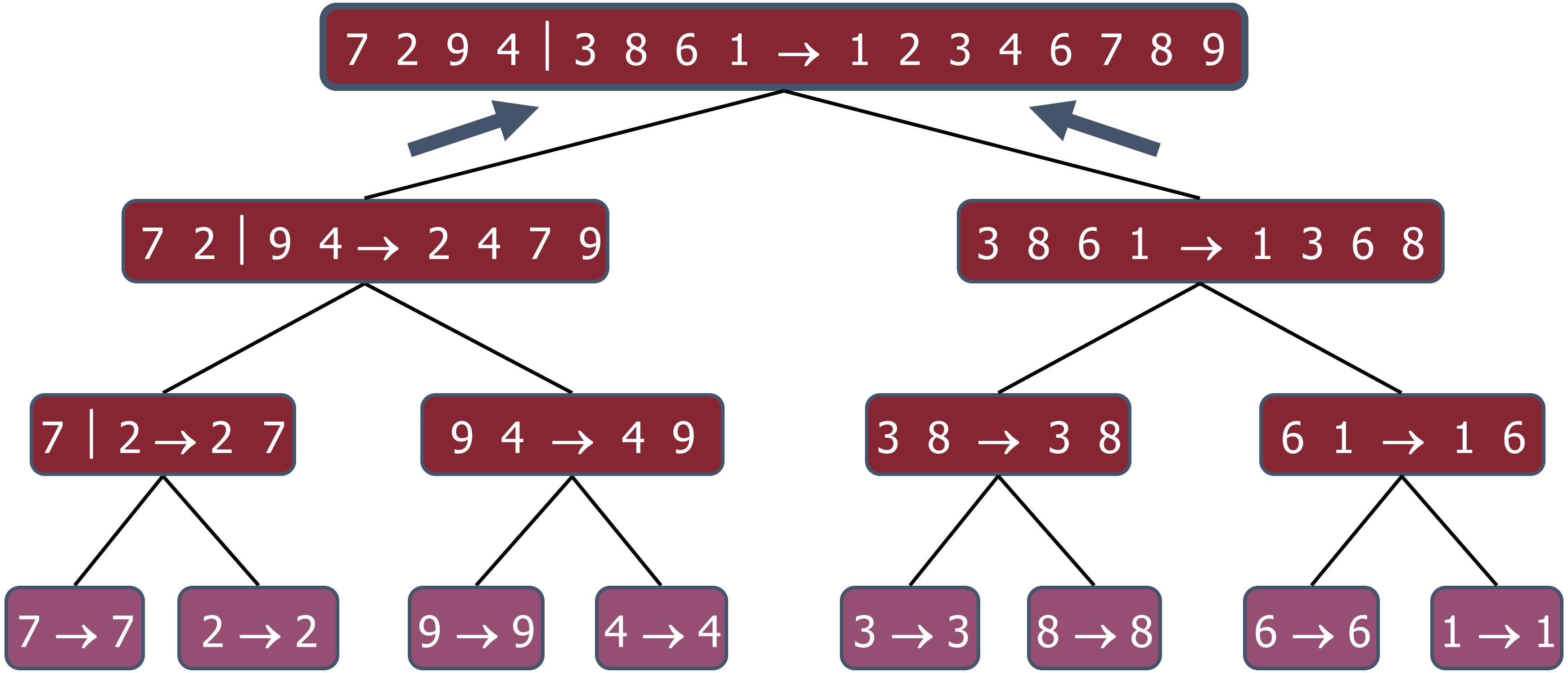
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

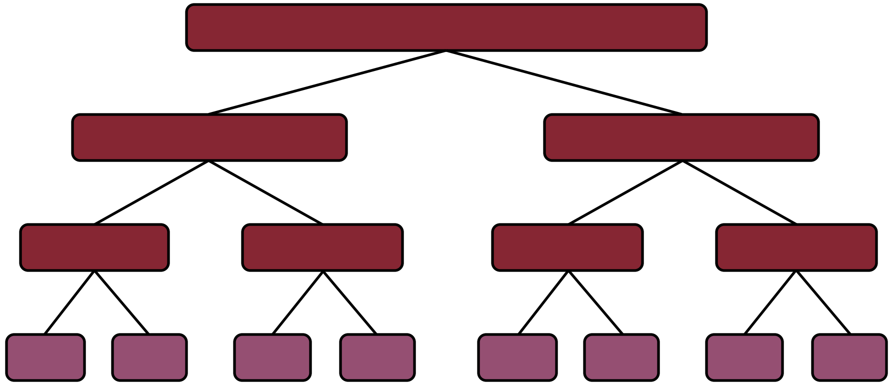
- Merge



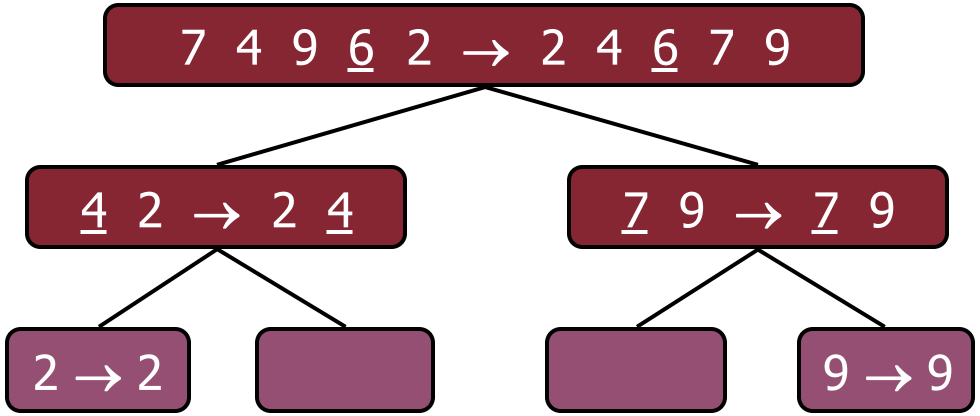
Analysis of Merge-Sort

- 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 2i sequences of size n/2i
  - we make 2i+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	$n/2^i$
…	…	…

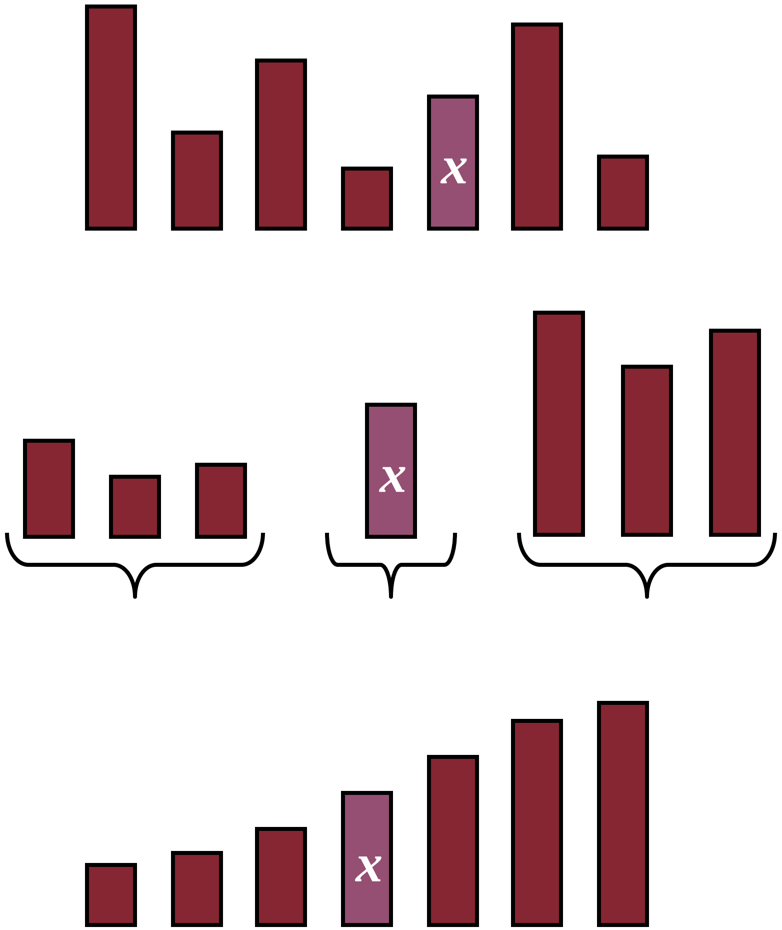


Quick-Sort



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



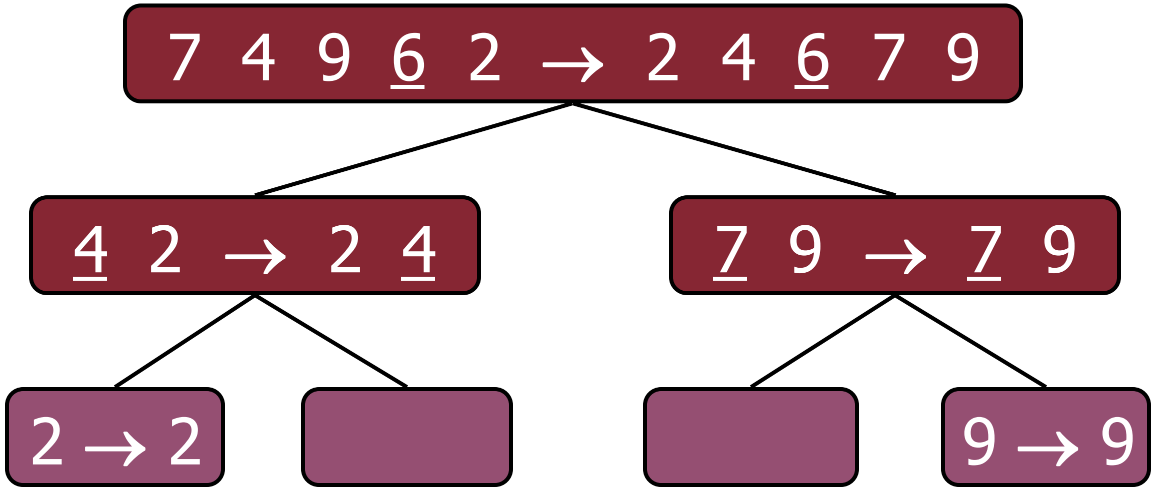
Partition

- 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
  $x \leftarrow S.erase(p)$
  while $\neg S.empty()$
    $y \leftarrow S.eraseFront()$
    if $y < x$
      $L.insertBack(y)$
    else if $y = x$
      $E.insertBack(y)$
    else { $y > x$ }
      $G.insertBack(y)$
  return L, E, G

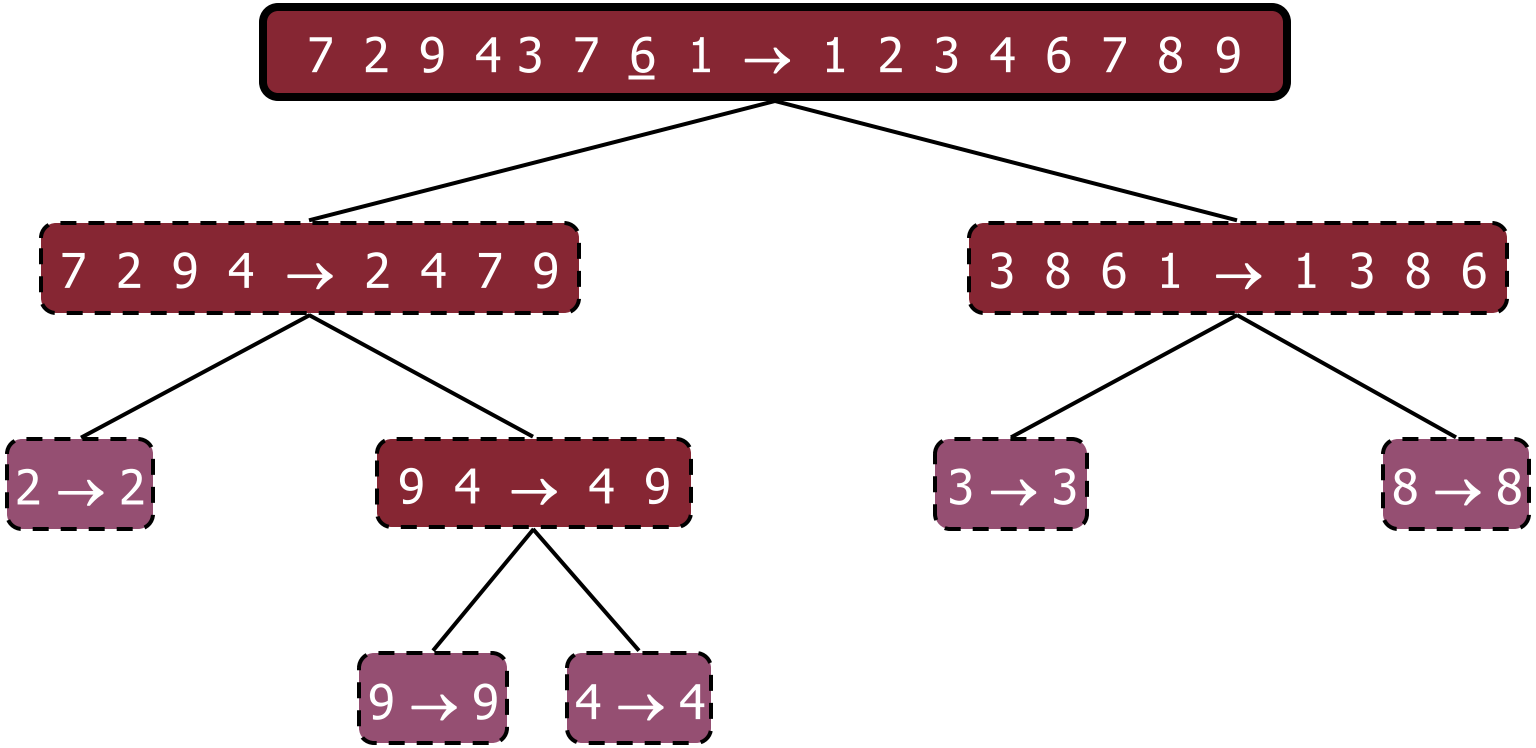
Quick-Sort Tree

- 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



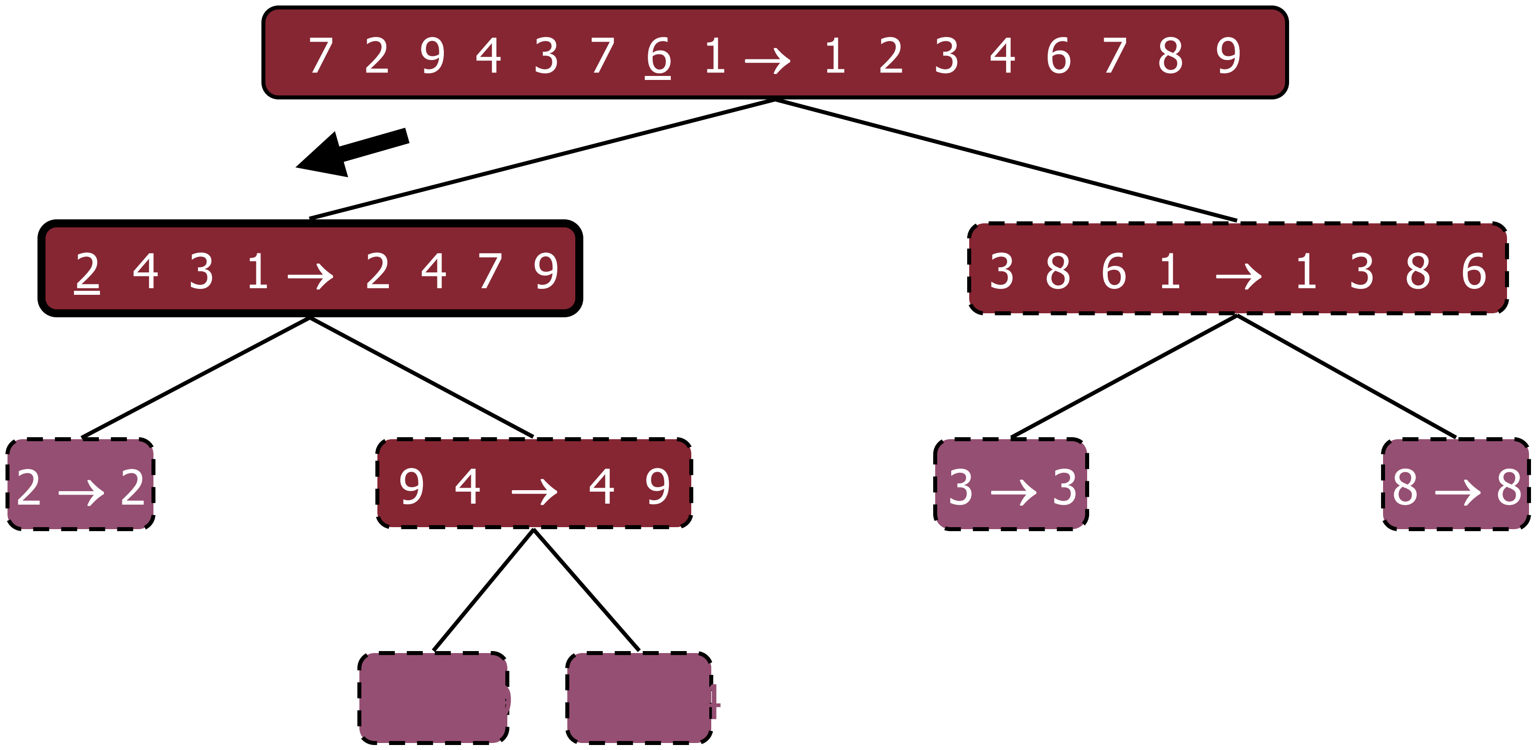
Execution Example

- Pivot selection



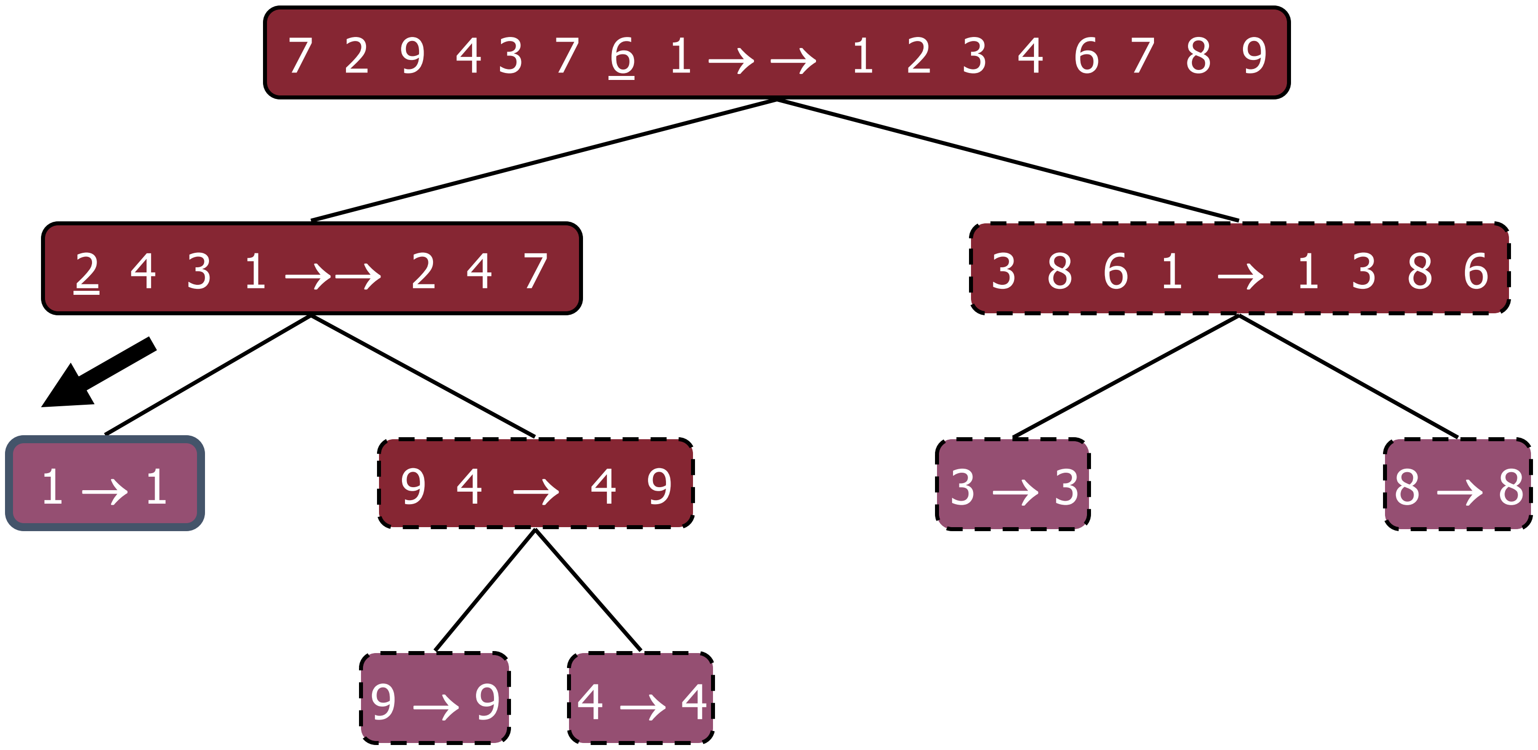
Execution Example (cont.)

- Partition, recursive call, pivot selection



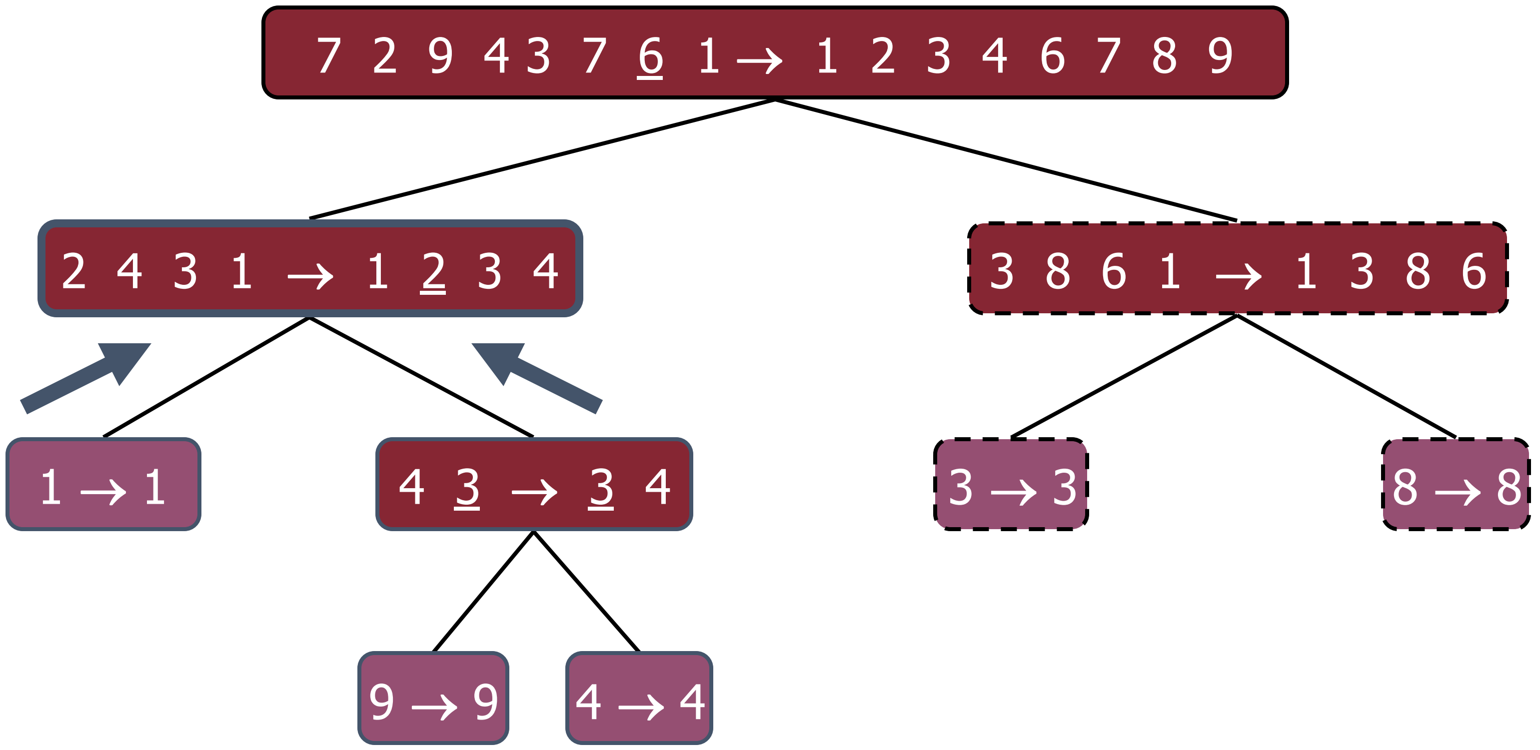
Execution Example (cont.)

- Partition, recursive call, base case



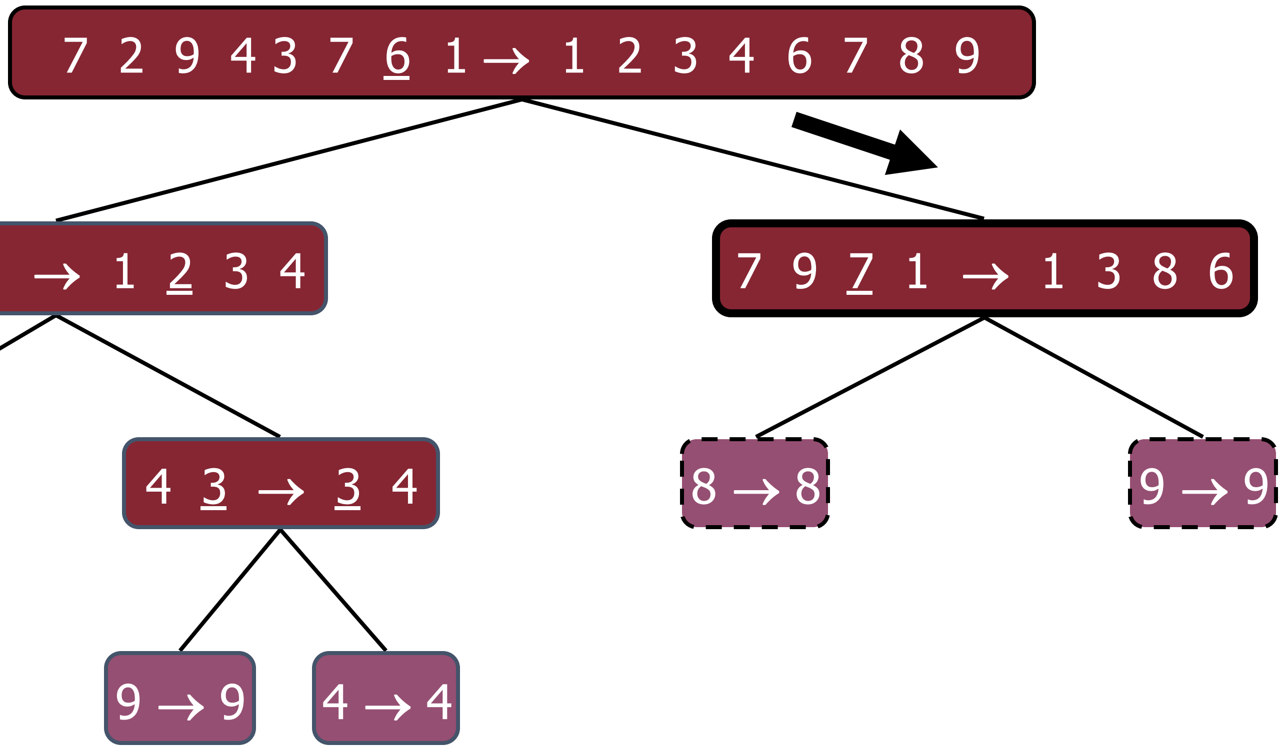
Execution Example (cont.)

- Recursive call, …, base case, join



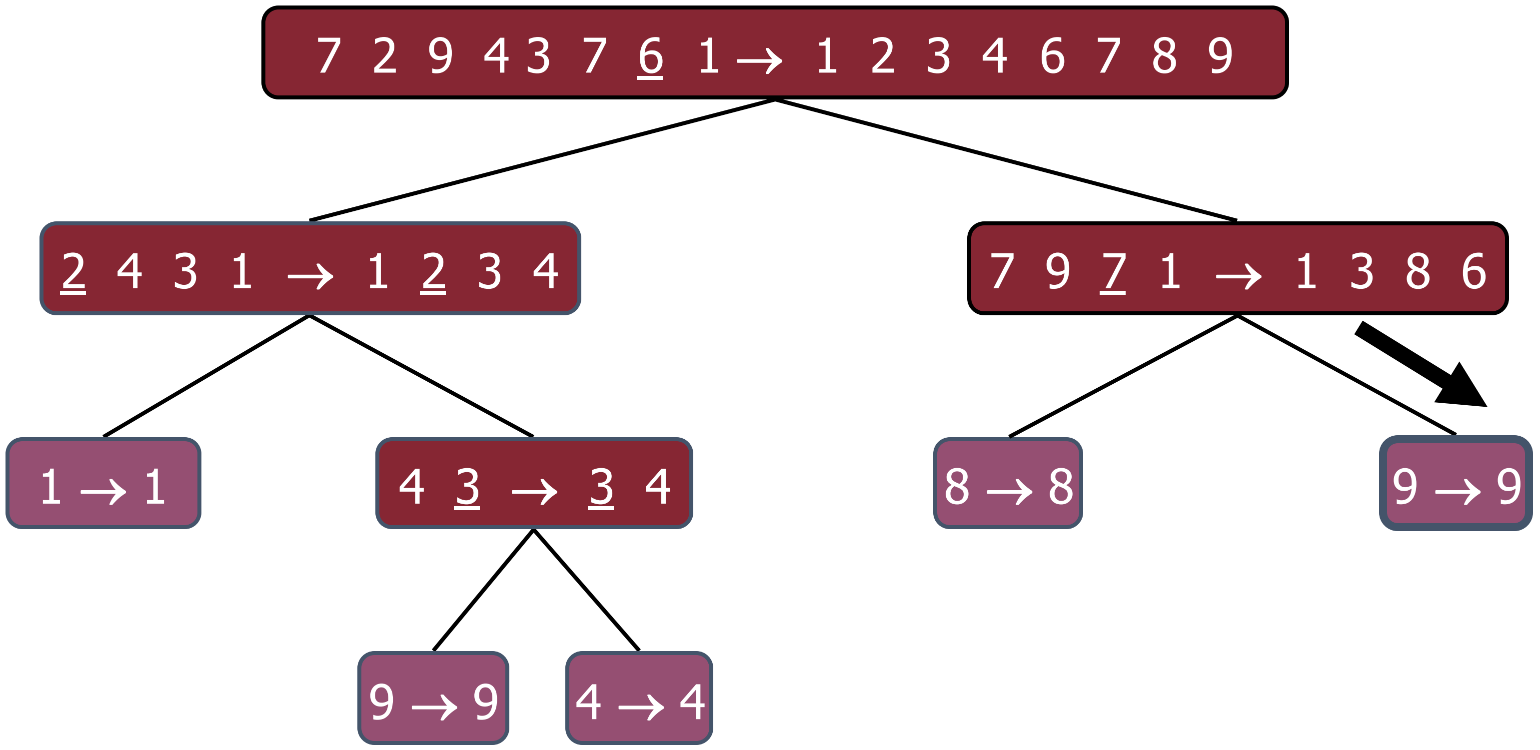
Execution Example (cont.)

- Recursive call, pivot selection



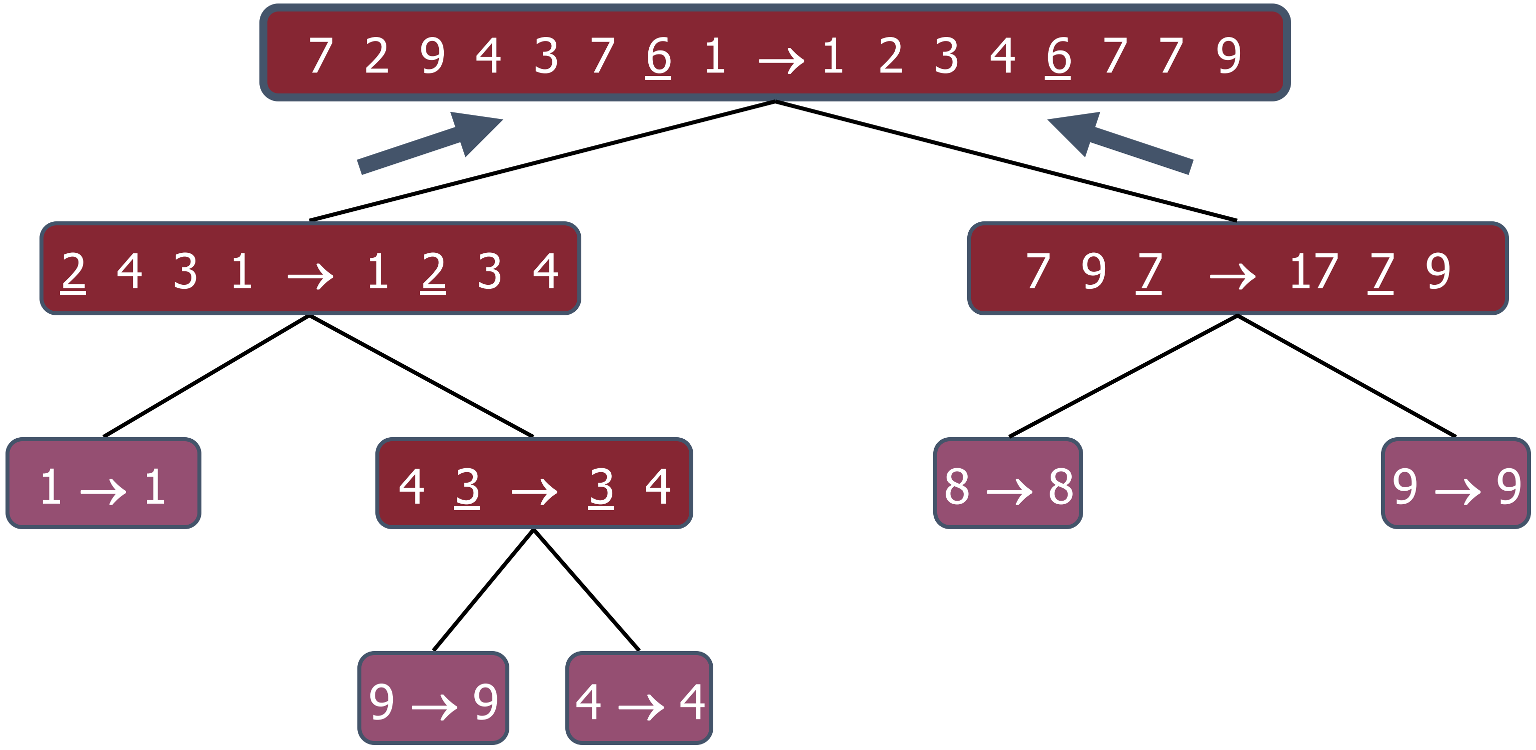
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

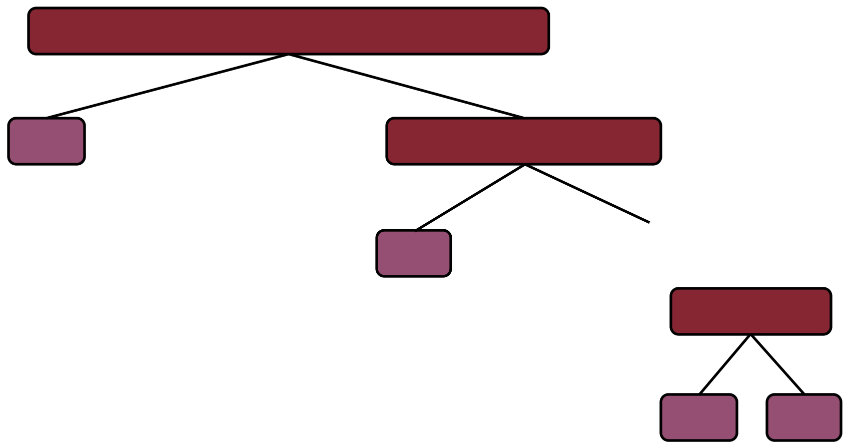
- Join, join



Worst-case Running Time

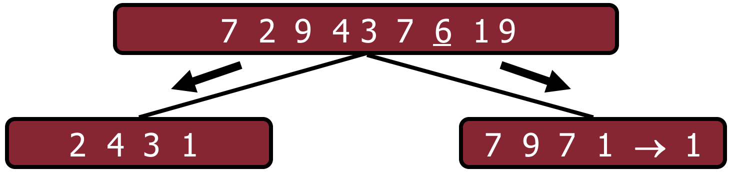
- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size n - 1 and the other has size 0
- The running time is proportional to the sum
- n + (n - 1) + … + 2 + 1
- Thus, the worst-case running time of quick-sort is O(n^2)

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

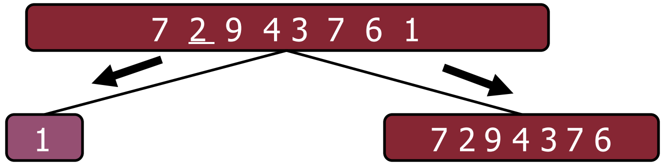


Expected Running Time

- 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

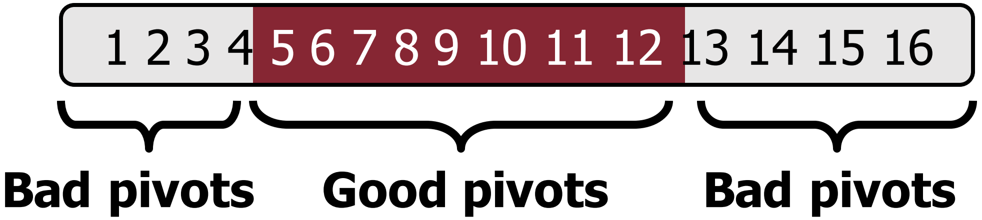


Good call



Bad call

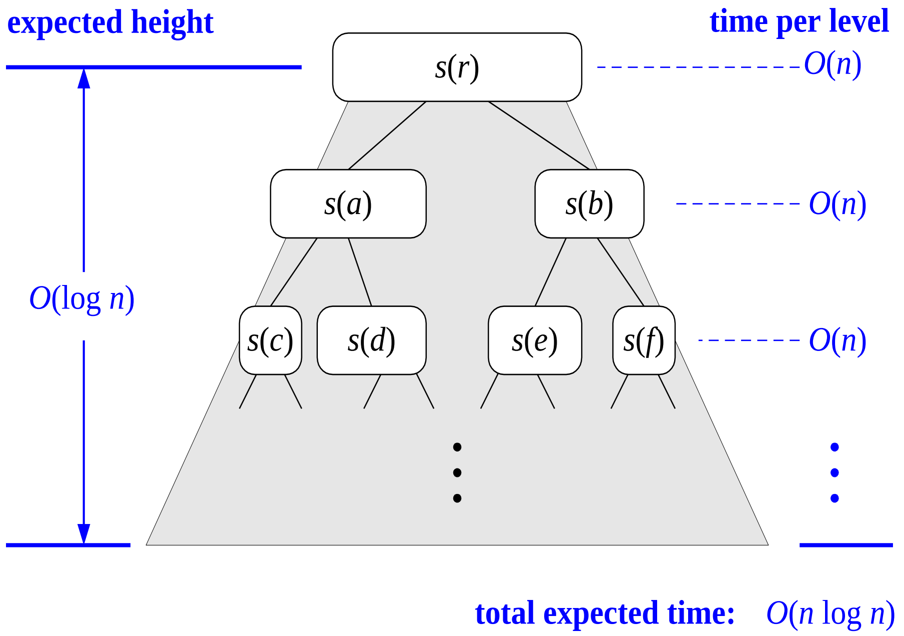
- A call is good with probability 1/2
  - 1/2 of the possible pivots cause good calls:



Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is 2k
- For a node of depth i, we expect
  - i/2 ancestors are good calls
  - The size of the input sequence for the current call is at most (3/4)i/2n

- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



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, l, r)

    Input sequence S, ranks l and r

    Output sequence S with the elements of rank between l and r rearranged in increasing order

    if $l \geq r$

        return

    $i \leftarrow$ a random integer between l and r

    $x \leftarrow S.elemAtRank(i)$

    $(h, k) \leftarrow inPlacePartition(x)$

    $inPlaceQuickSort(S, l, h-1)$

    $inPlaceQuickSort(S, k+1, r)$

In-Place Partitioning

- Perform the partition using two indices to split S into L and E U G (a similar method can split E U G into E and G).

j k

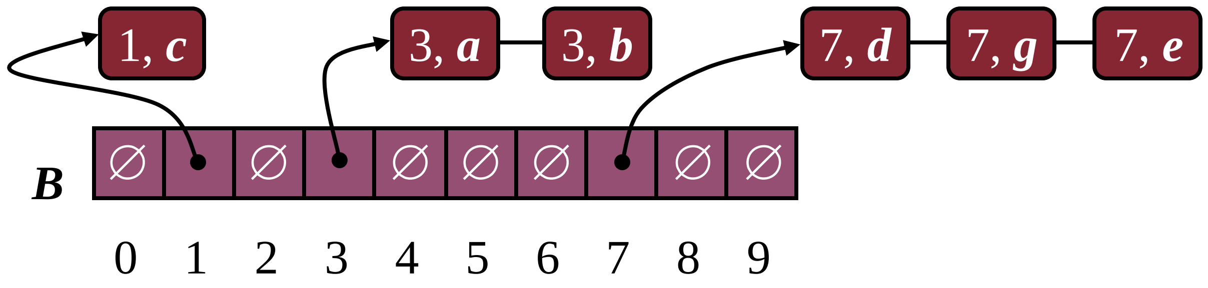
3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 <u>**6**</u> 9 (pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element > x.
  - Scan k to the left until finding an element < x.
  - Swap elements at indices j and k

j k

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 <u>**6**</u> 9

Bucket-Sort and Radix-Sort



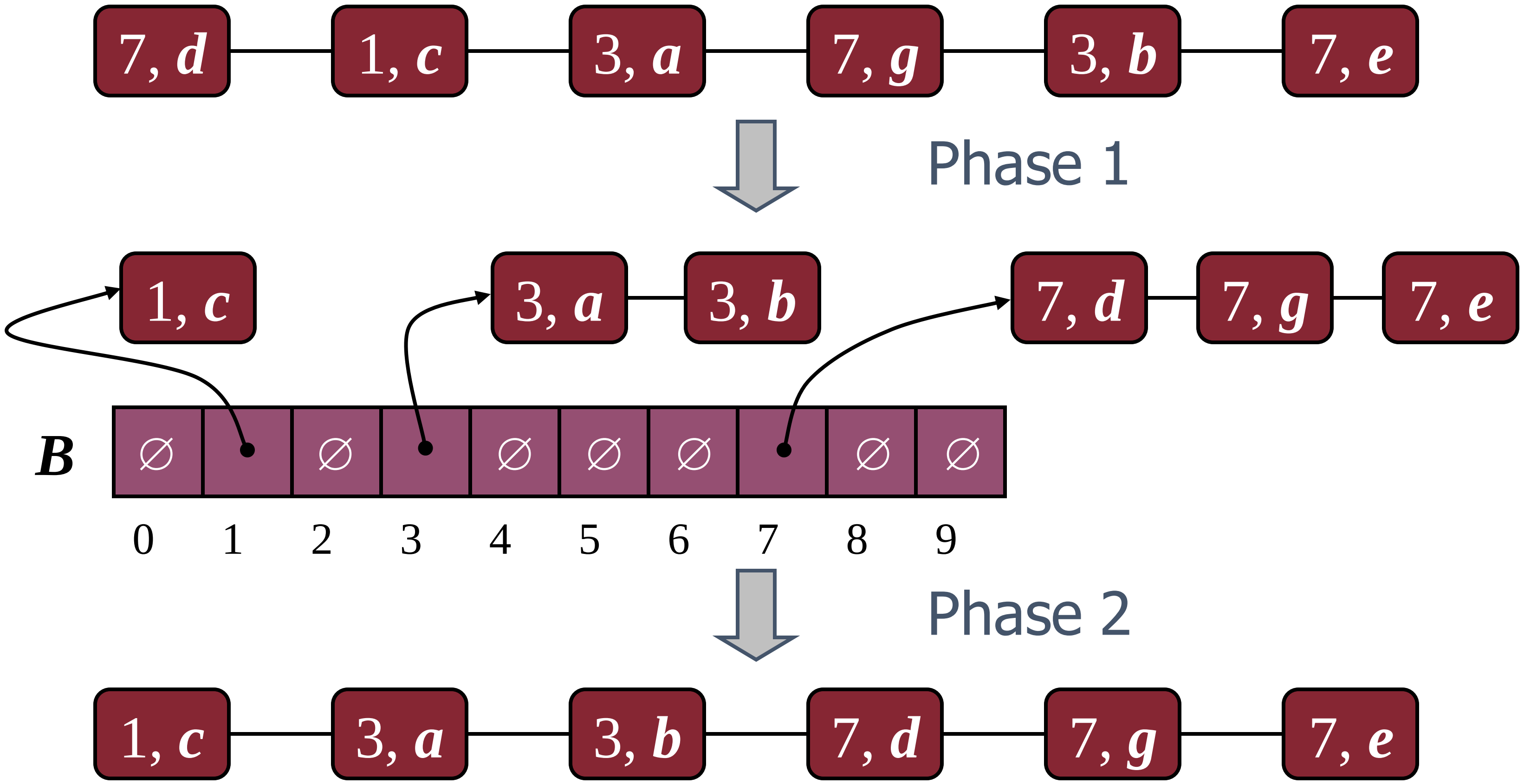
Bucket-Sort

- Let be S be a sequence of n (key, element) entries with keys in the range [0, N - 1]
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
  - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket B[k]
  - Phase 2: For i = 0, …, N - 1, move the entries of bucket B[i] to the end of sequence S
- Analysis:
  - Phase 1 takes O(n) time
  - Phase 2 takes O(n + N) time
- Bucket-sort takes O(n + N) time

Algorithm *bucketSort*(*S*, *N*)
    Input sequence ***S*** of (key, element) items with keys in the range $[0, N - 1]$
    Output sequence ***S*** sorted by increasing keys
    $B \leftarrow$ array of N empty sequences
    while $\neg S.empty()$
        $(k, o) \leftarrow S.front()$
        $S.eraseFront()$
        $B[k].insertBack((k, o))$
    for $i \leftarrow 0$ **to** $N - 1$
        while $\neg B[i].empty()$
            $(k, o) \leftarrow B[i].front()$
            $B[i].eraseFront()$
            $S.insertBack((k, o))$

Example

- Key range [0, 9]



Properties and Extensions

- Key-type Property
  - The keys are used as indices into an array and cannot be arbitrary objects
  - No external comparator
- Stable Sort Property
  - The relative order of any two items with the same key is preserved after the execution of the algorithm

- Extensions
  - Integer keys in the range [a, b]
    - Put entry (k, o) into bucket B[k - a]
  - String keys from a set D of possible strings, where D has constant size (e.g., names of the 50 U.S. states)
    - Sort D and compute the rank r(k) of each string k of D in the sorted sequence
    - Put entry (k, o) into bucket B[r(k)]

Radix-Sort

- Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range [0, N - 1]
- Radix-sort runs in time O(d(n + N))

Algorithm *radixSort*(S, N)
    Input sequence S of d-tuples such
        that $(0, \ldots, 0) \leq (x_1, \ldots, x_d)$ and
        $(x_1, \ldots, x_d) \leq (N-1, \ldots, N-1)$
        for each tuple $(x_1, \ldots, x_d)$ in S
    Output sequence S sorted in
        lexicographic order
    for $i \leftarrow d$ **downto** 1
        bucketSort(S, N)

Radix-Sort for Binary Numbers

- Consider a sequence of n b-bit integers

  $x = x^{b-1} \ldots x^1 x0$
- We represent each element as a b-tuple of integers in the range [0, 1] and apply radix-sort with N = 2
- This application of the radix-sort algorithm runs in O(bn) time
- For example, we can sort a sequence of 32-bit integers in linear time

Algorithm *binaryRadixSort*(S)
    Input sequence S of b-bit integers
    Output sequence S sorted
    replace each element x of S with the item $(0, x)$
    for $i \leftarrow 0$ **to** $b - 1$
        replace the key k of each item (k, x) of S with bit x_i of x
        bucketSort(S, 2)

Example

- Sorting a sequence of 4-bit integers