Sorting

- Explain Merge-sort
- Explain quick-sort
- Analyze sorting algorithms
- Compare sorting algorithms

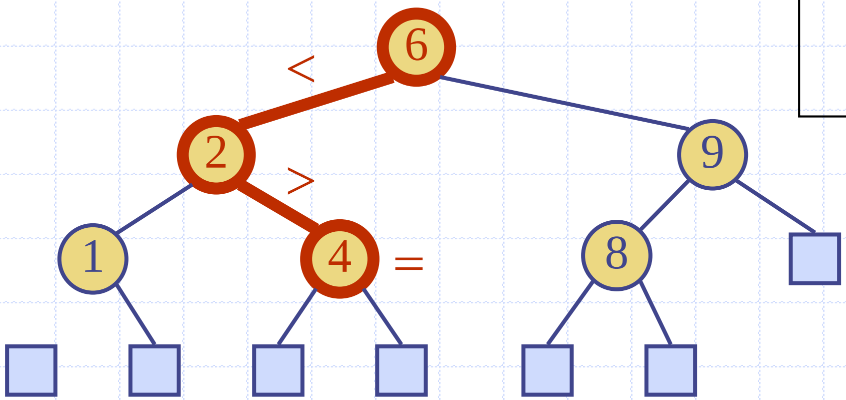
Search Trees - Review

- Use a **search-tree structure** to efficiently implement a ***sorted map***
- *A* ***binary search tree*** is a *proper binary tree:* each internal position p stores a key-value pair (k, v) such that:
  - Keys stored in the **left subtree** of p are **less** than k.
  - Keys stored in the **right subtree** of p are **greater** than k.
- Operations:
  - Search
  - Insert
  - Delete

Search Trees - Review

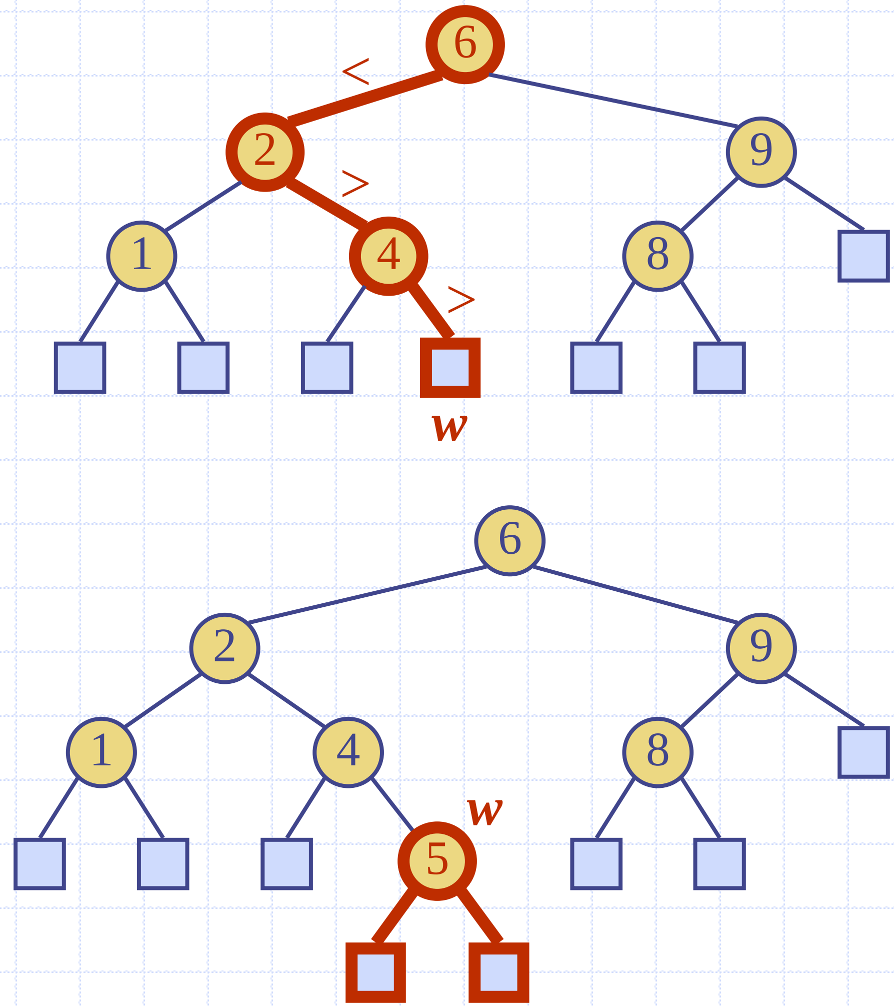
- Search for a key:
- Example: get(4):
  - Call TreeSearch(4,root)

Algorithm *TreeSearch*(k, v)
  if *T.isExternal* (v)
    return v
  if $k < key(v)$
    return *TreeSearch*(k, *left*(v))
  else if $k = key(v)$
    return v
  else { $k > key(v)$ }
    return *TreeSearch*(k, *right*(v))



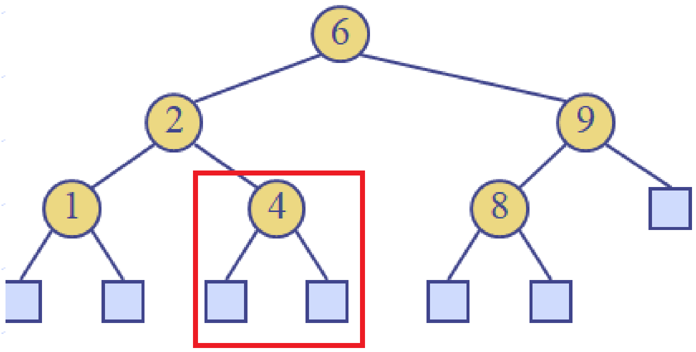
Search Trees - Review

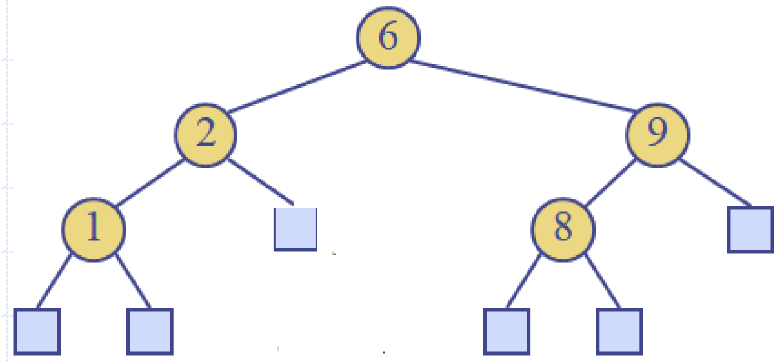
- **Insertion:**
- To perform operation put(**k**, o), we search for key **k** (using TreeSearch)
- Assume **k** is not already in the tree, and let **w** be the leaf reached by the search
- We insert **k** at node **w** and expand **w** into an internal node
- Example: insert 5



Deletion – node has no children

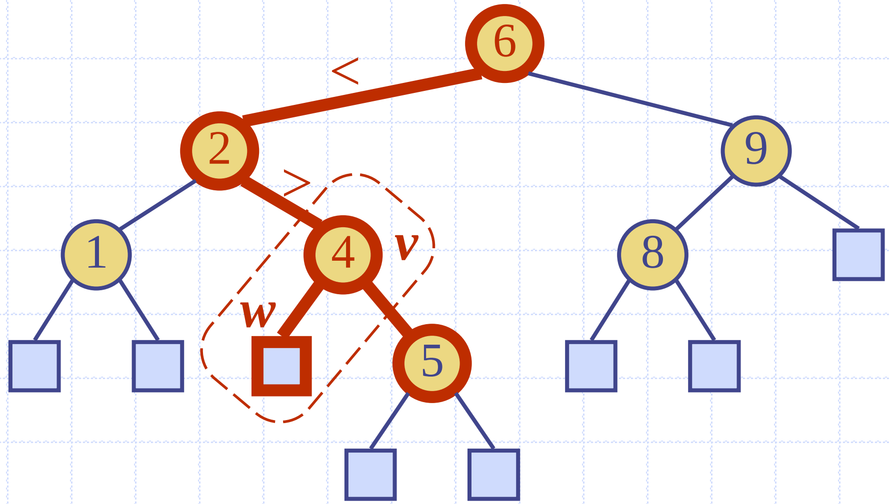
- Assume key *k* is in the tree, and let *v* be the node storing *k*
  - *Remove v and its leafs (placeholders)*
  - *Replace v with a leaf (placeholder)*
- Example: **remove 4**

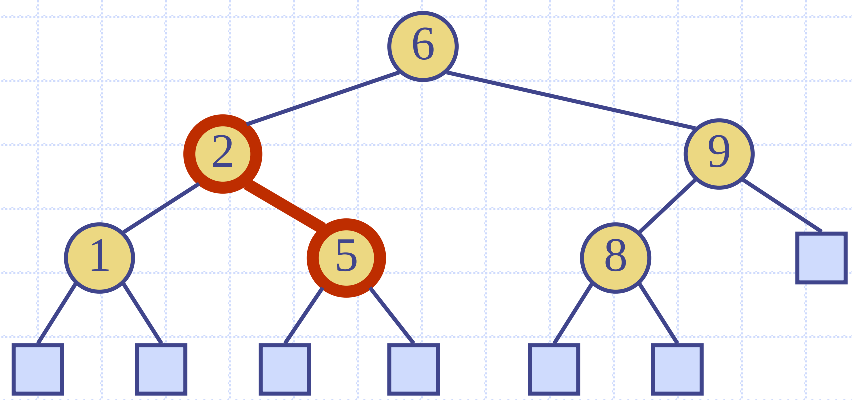




Deletion – node has one child

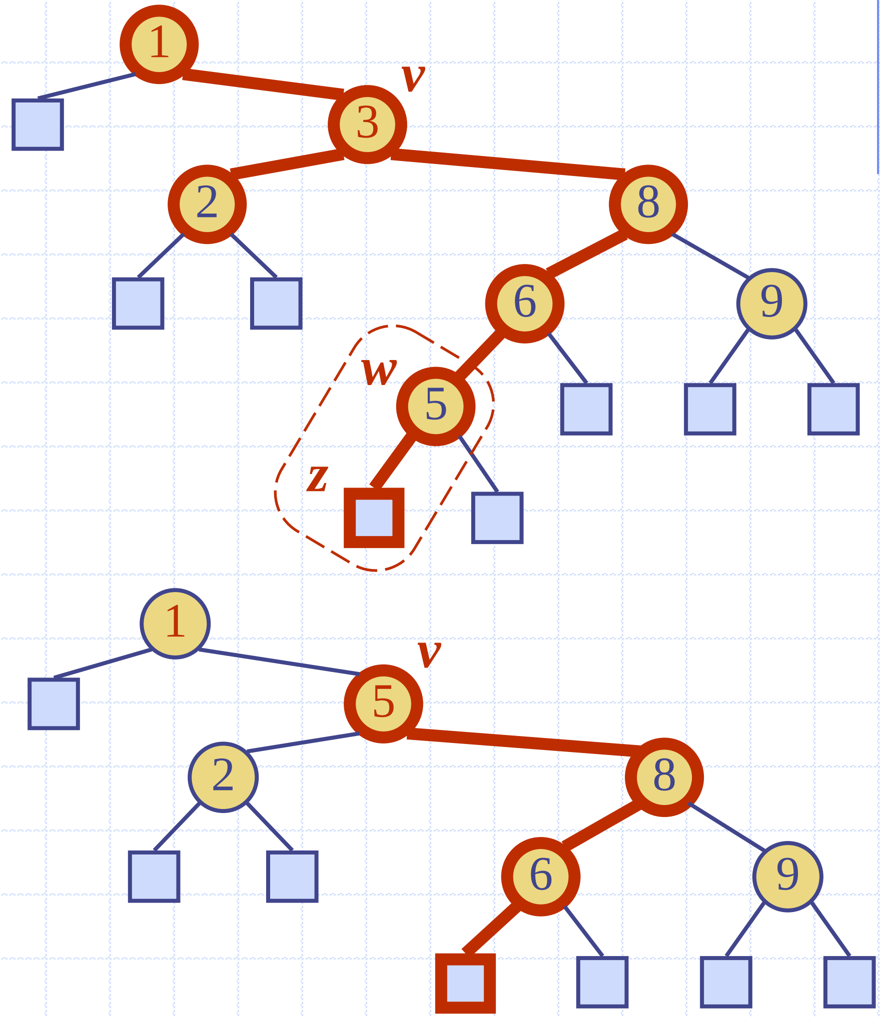
- To perform operation **remove**(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If **node v has a leaf child** w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: **remove 4**





Deletion – node has two children

- We consider the case where the key k to be removed is stored at a node v whose **children are both internal**
  - we **find the internal node w that follows v in an inorder traversal**
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)
- Example**: remove 3**



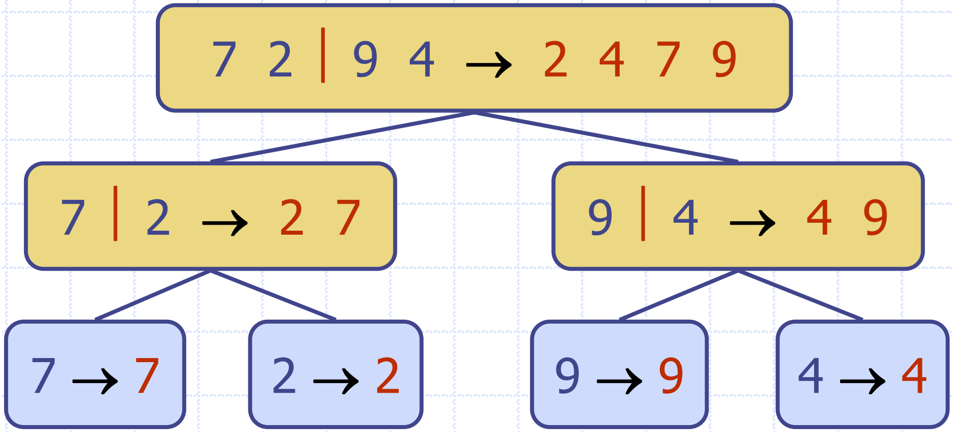
Performance

- On average, a binary search tree with n keys generated from a random series of insertions and removals of keys has expected height $O(\log n)$

Method	Running Time
size, isEmpty	$O(1)$
get, put, remove	$O(h)$
firstEntry, lastEntry	$O(h)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(h)$
subMap	$O(s+h)$
entrySet, keySet, values	$O(n)$

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Merge Sort



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 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 sorted sequences** S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)
    Input sequence S with n elements
    Output sequence S sorted according to C
    if $S.size() > 1$
        $(S_1, S_2) \leftarrow partition(S, n/2)$
        $mergeSort(S_1)$
        $mergeSort(S_2)$
        $S \leftarrow merge(S_1, S_2)$

Merging Two Sorted Sequences

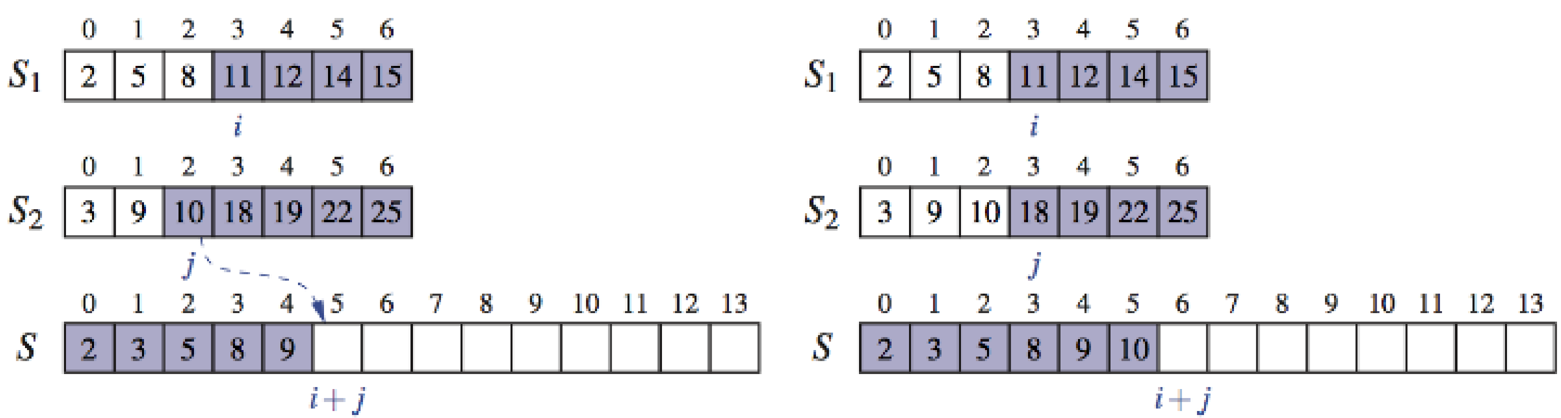
- The conquer step of merge-sort consists of **merging two sorted sequences** A and B into a sorted sequence S containing the **union** of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and **implemented by means of a doubly linked list**, takes $O(n)$ time

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

    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
```

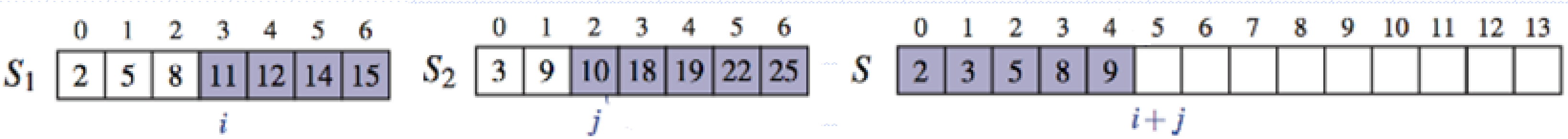
Java Merge Implementation

```
/** Merge contents of arrays S1 and S2 into properly sized array S. */
public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {
  int i = 0, j = 0;
  while (i + j < S.length) {
    if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
      S[i+j] = S1[i++];                    // copy ith element of S1 and increment i
    else
      S[i+j] = S2[j++];                    // copy jth element of S2 and increment j
  }
}
```



Java Merge Implementation

- Picture explanation:
  - i=3, j=2
  - S1[3] > S2[2] // 11>10
  - S[i+j] = S[5] = S2[2] //**10**
- Code explanation:
  - if j==S2.length **or** (i<S1.length and S1[i]<S2[j]) copy element from S1
  - Otherwise:
    - Otherwise, copy element from S2

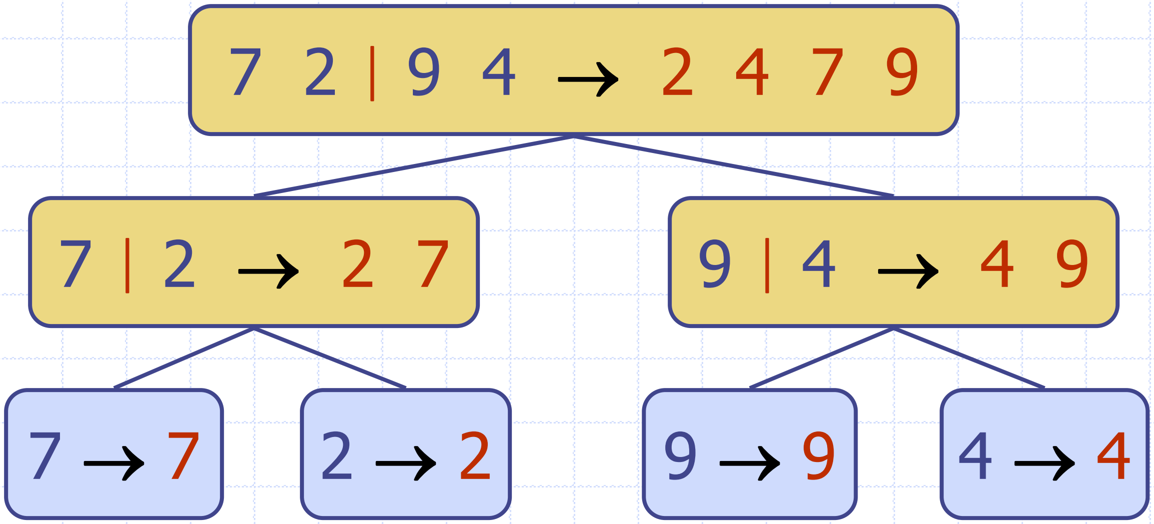


Java Merge-Sort Implementation

```
/** Merge-sort contents of array S. */
public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
  int n = S.length;
  if (n < 2) return;                                    // array is trivially sorted
  // divide
  int mid = n/2;
  K[ ] S1 = Arrays.copyOfRange(S, 0, mid);              // copy of first half
  K[ ] S2 = Arrays.copyOfRange(S, mid, n);              // copy of second half
  // conquer (with recursion)
  mergeSort(S1, comp);                                  // sort copy of first half
  mergeSort(S2, comp);                                  // sort copy of second half
  // merge results
  merge(S1, S2, S, comp);                  // merge sorted halves back into original
}
```

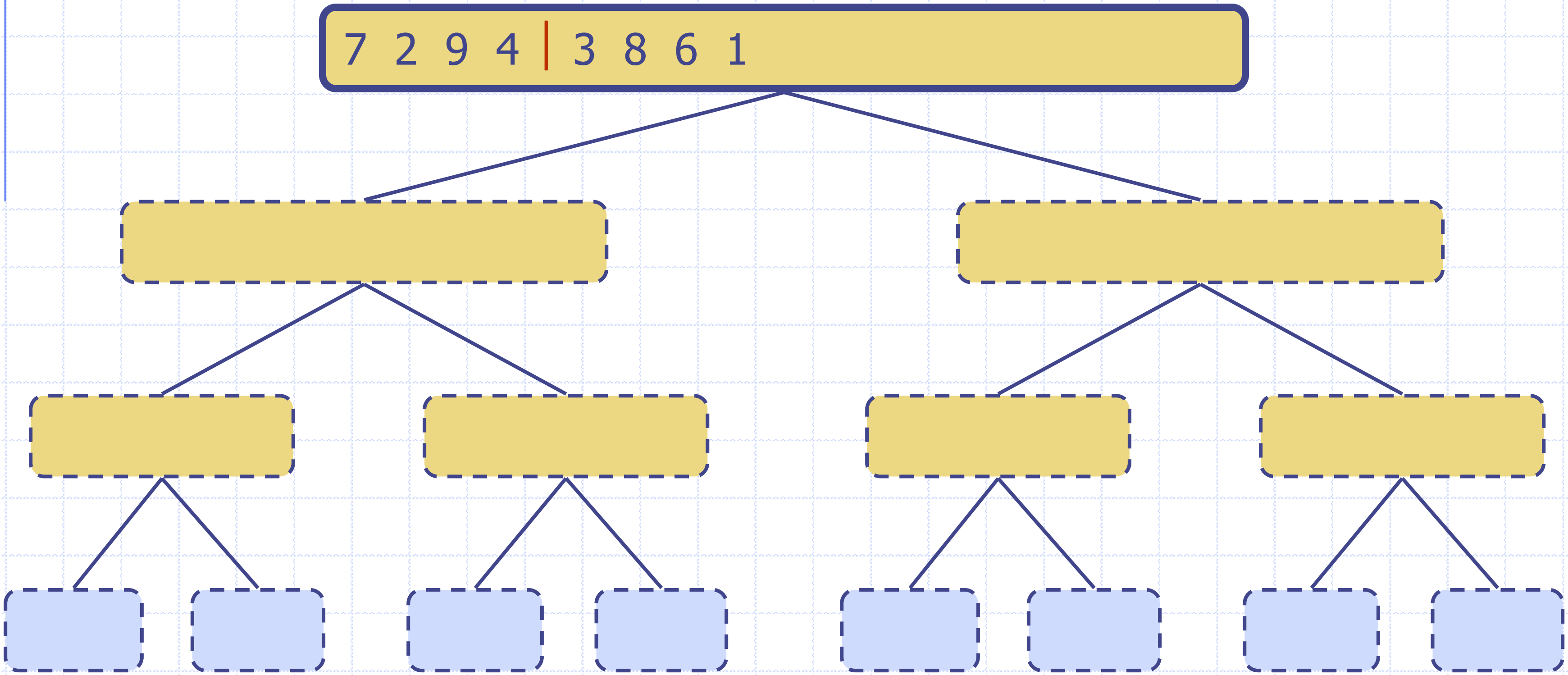
Merge-Sort Tree

- 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



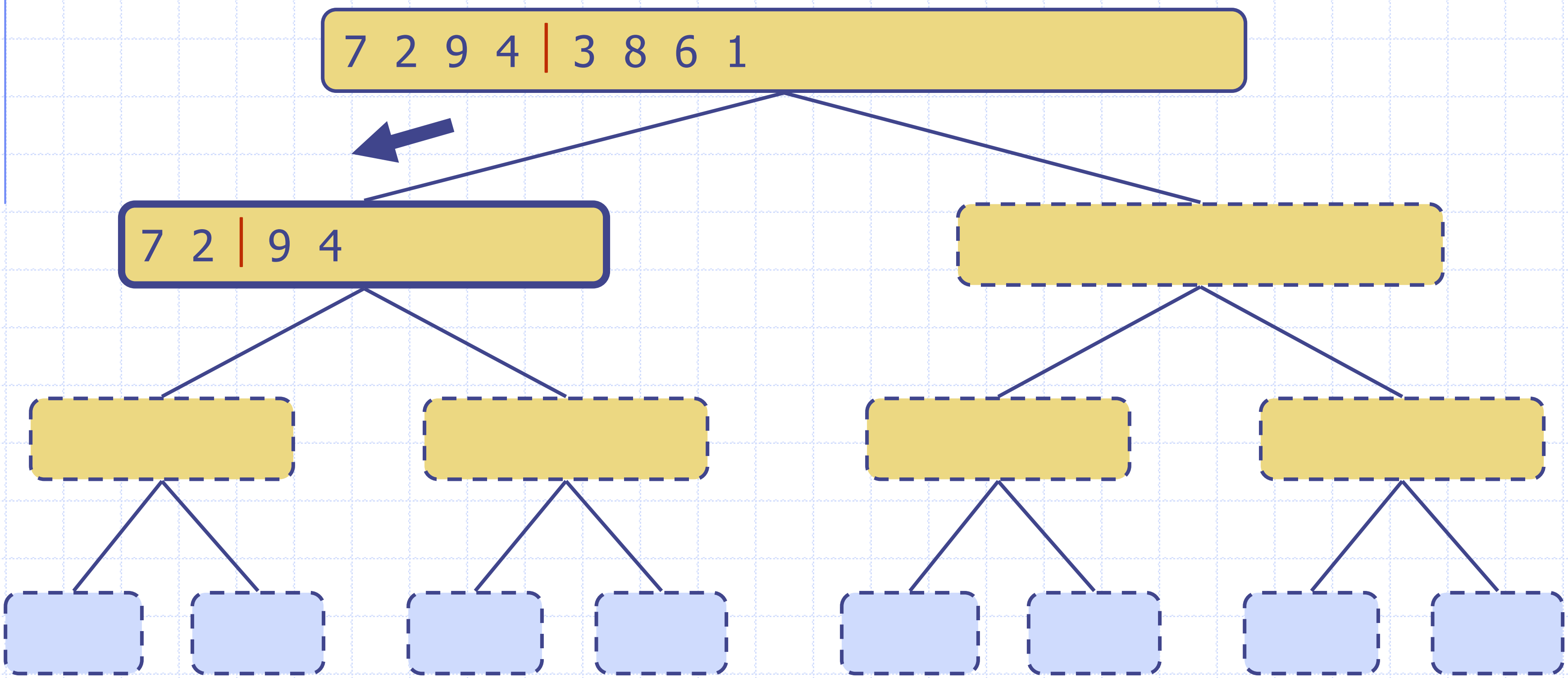
Execution Example

- Partition



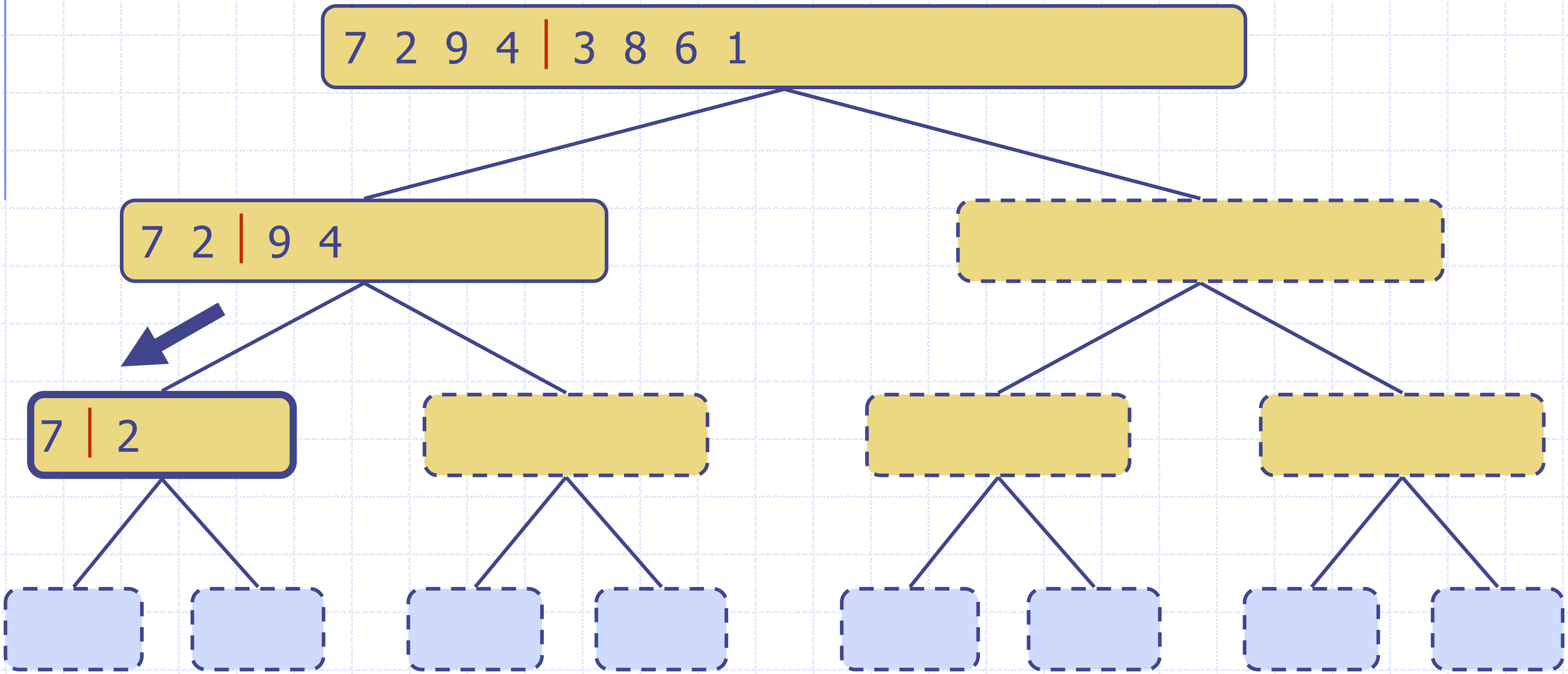
Execution Example (cont.)

- Recursive call, partition



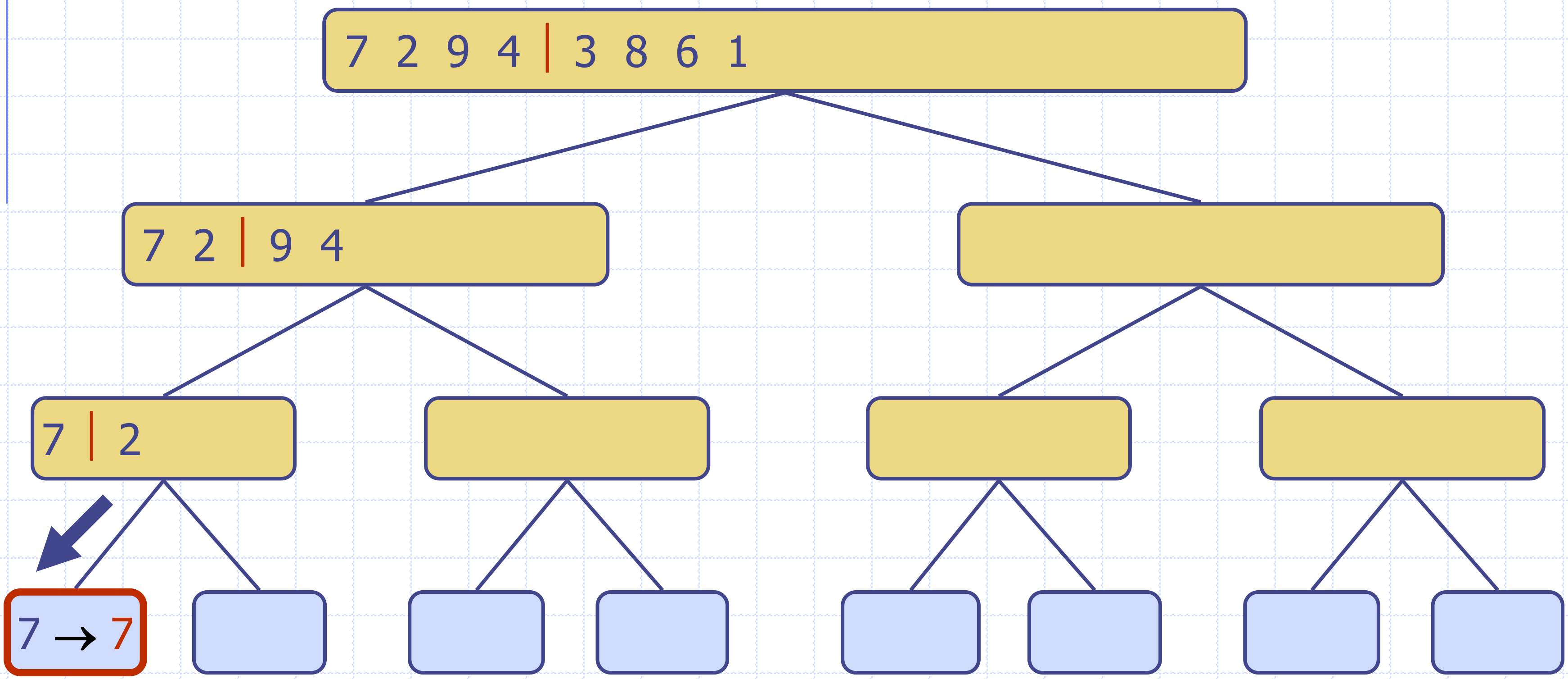
Execution Example (cont.)

- Recursive call, partition



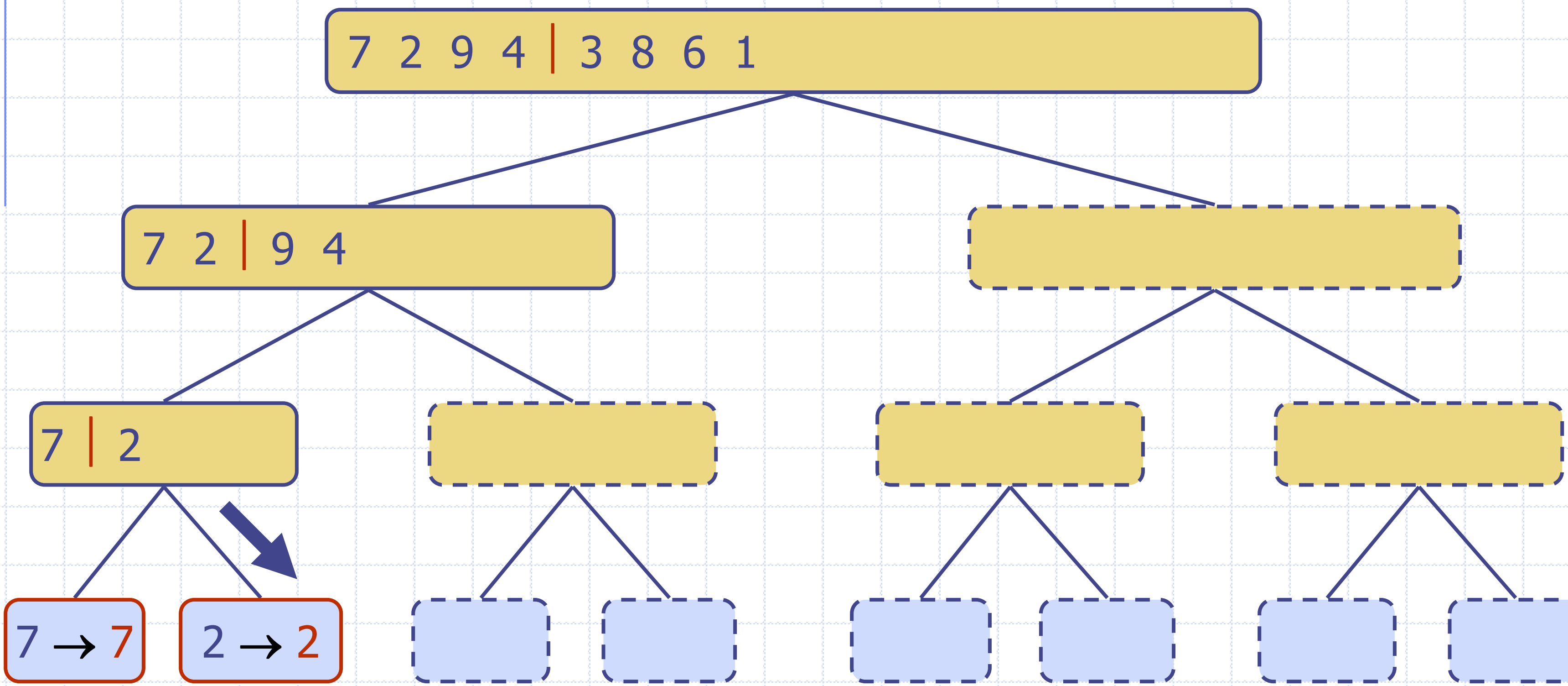
Execution Example (cont.)

- Recursive call, reaches base case



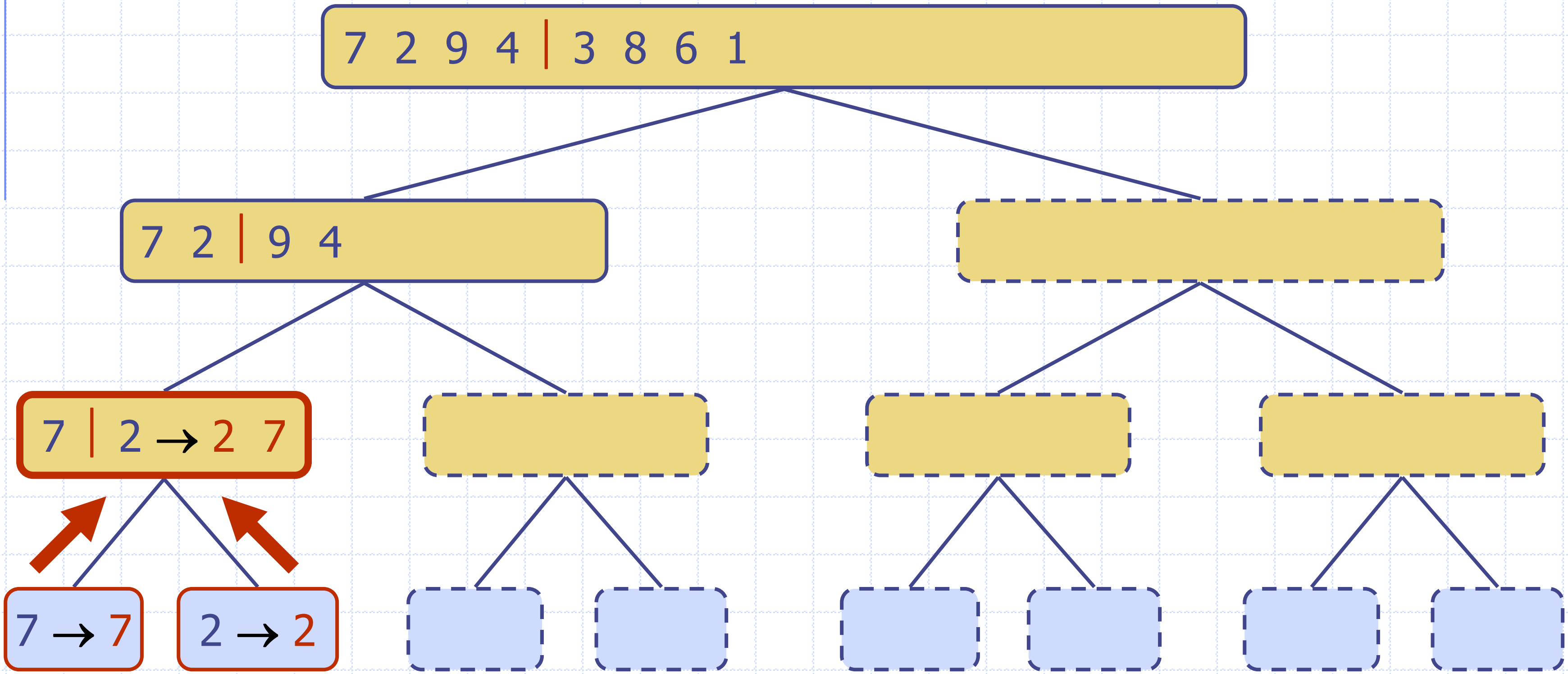
Execution Example (cont.)

- Recursive call, reaches base case



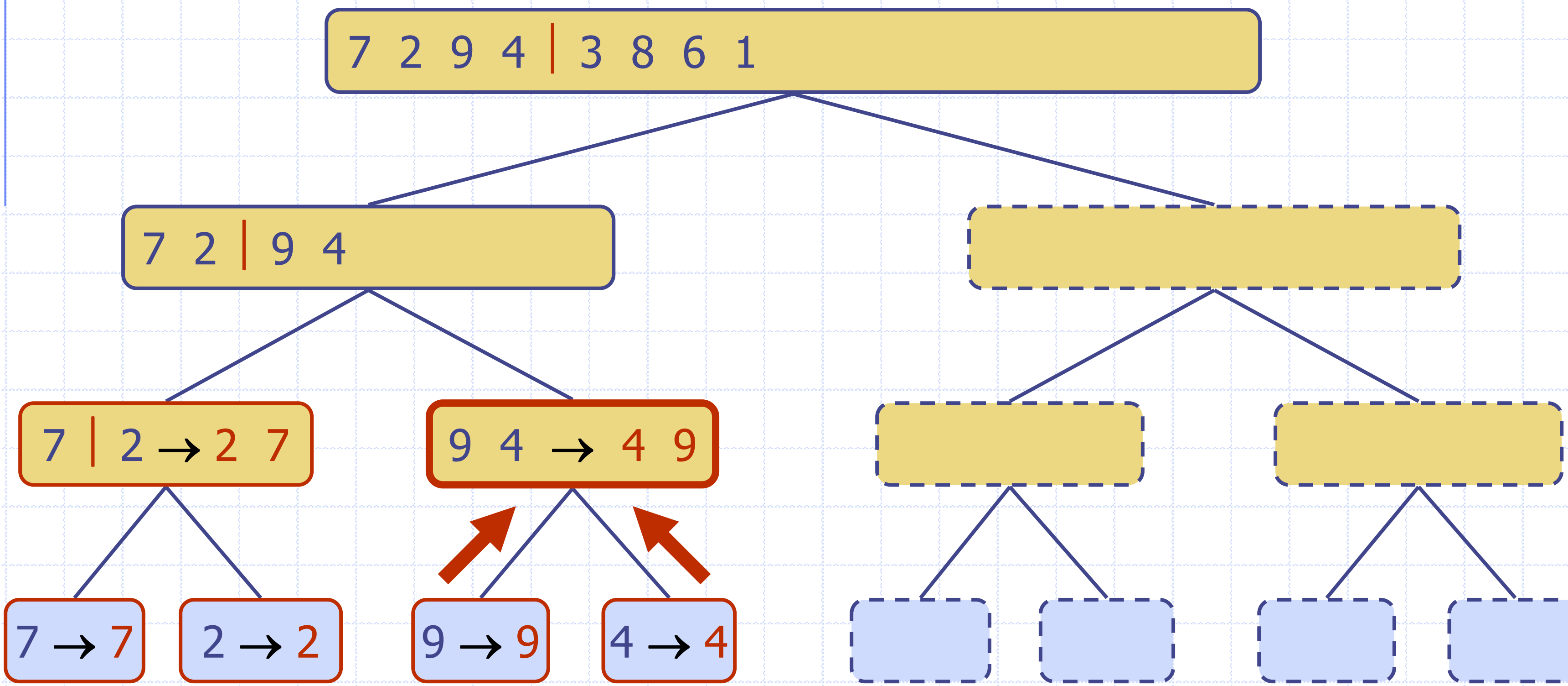
Execution Example (cont.)

- Merge



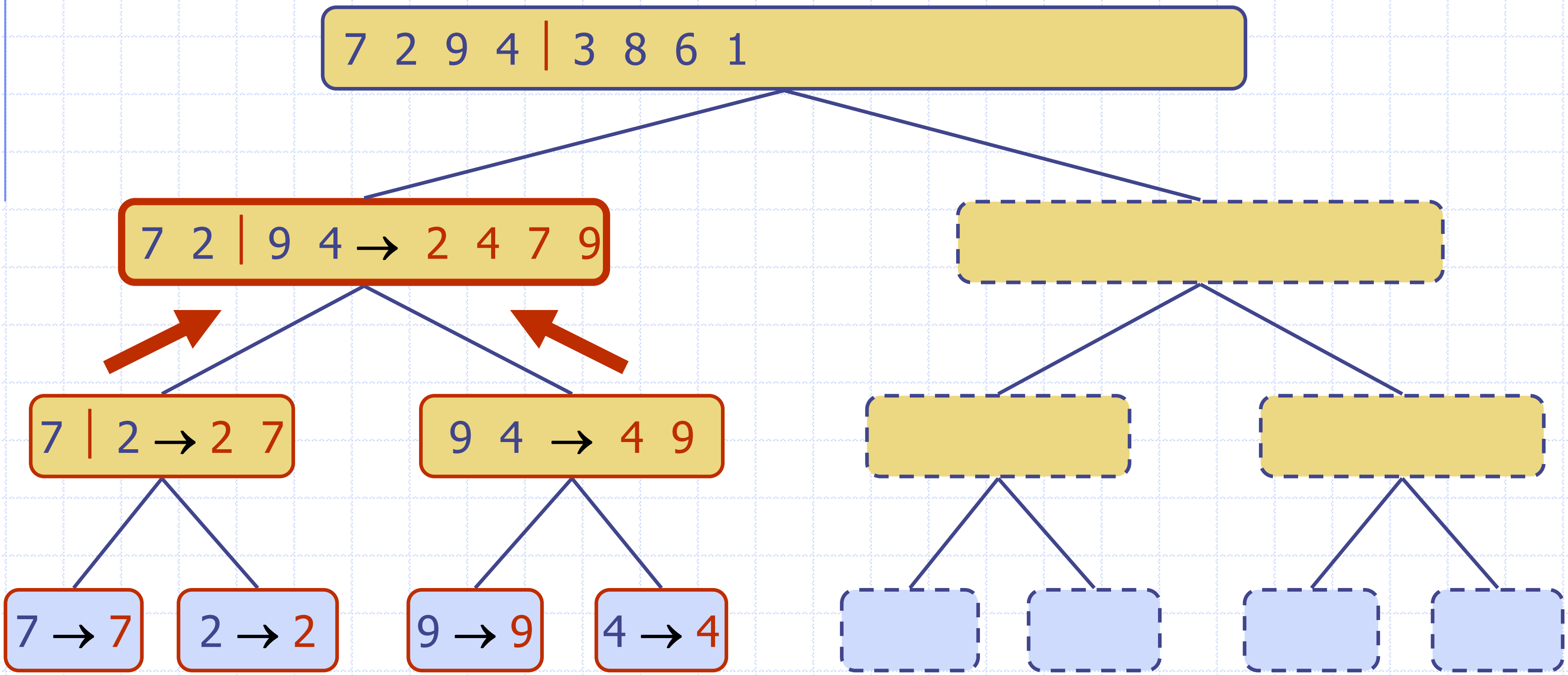
Execution Example (cont.)

- Recursive call, …, base case, merge



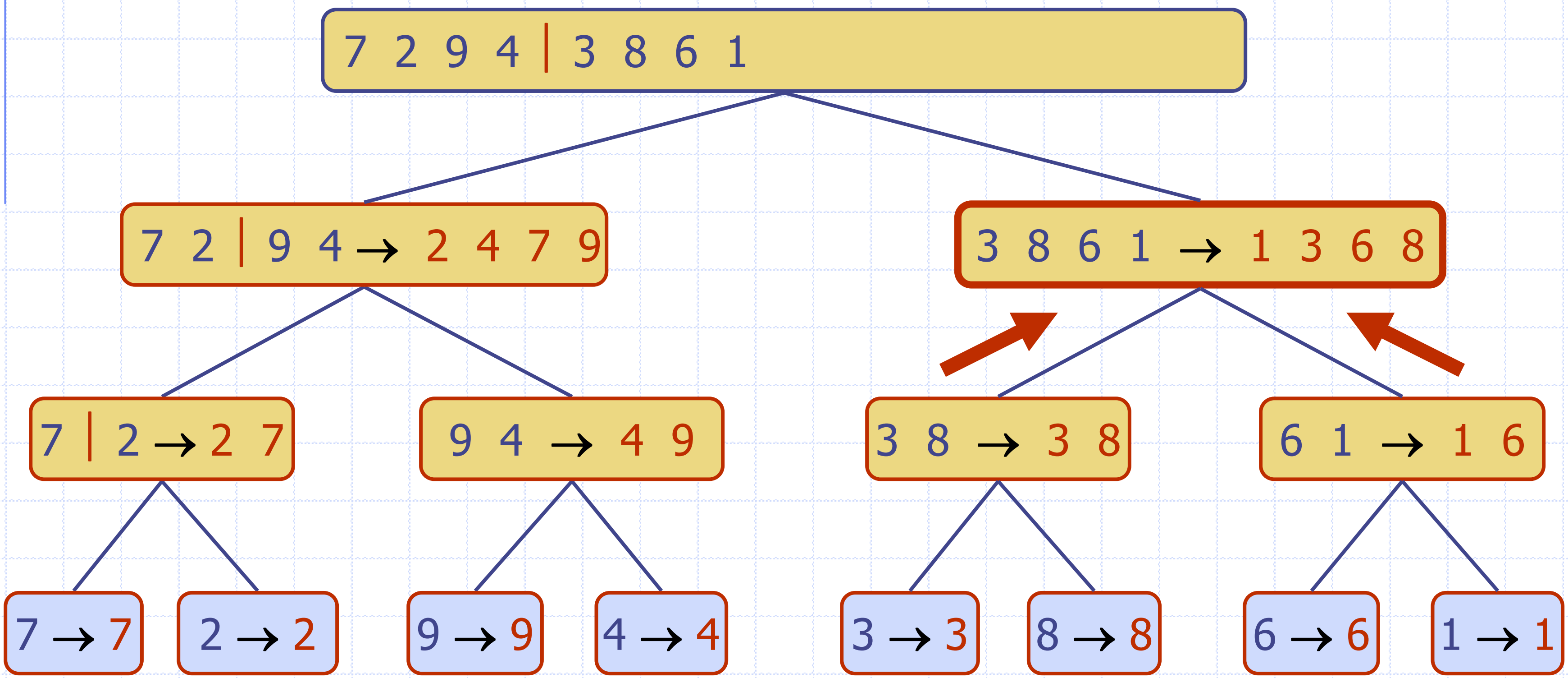
Execution Example (cont.)

- Merge



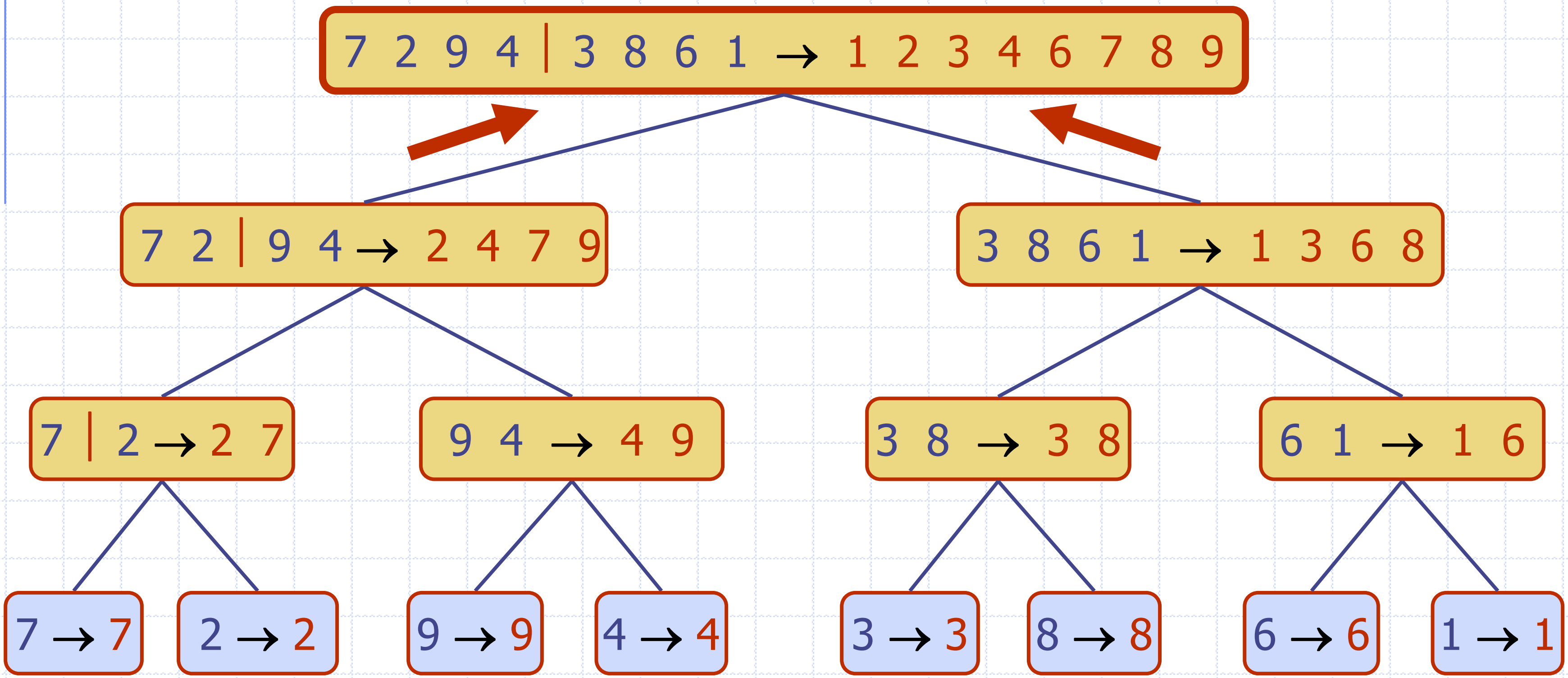
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

- Merge



Analysis of Merge-Sort

- Running time of the **merge** algorithm:
  - Let **n1** and **n2** be the number of elements of S1 and S2, respectively.
  - It is clear that the operations (addLast, removeFirst) performed inside each pass of the while loop take O(1) time.
  - The key observation is that during each iteration of the loop, one element is copied from either S1 or S2 into S (and that element is considered no further).
  - Therefore, the number of iterations of the loop is n1+n2.
- Thus, the running time of algorithm **merge** is **O(n1+n2), hence linear in the size of merged sequence**.

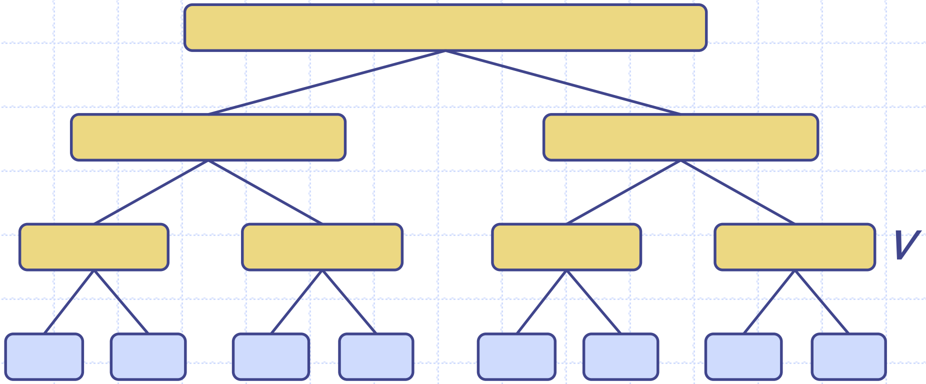
Analysis of Merge-Sort

- Running time of the entire merge-sort algorithm:
  - We account for the **amount of time spent within each recursive call**, but excluding any time spent waiting for successive recursive calls to terminate.
  - In the case of our mergeSort method, we account for the **time to divide** the sequence into two subsequences, and the **call to merge** to combine the two sorted sequences, but we exclude the two recursive calls to mergeSort.
  - We use a **merge-sort tree** T to guide our analysis.

Analysis of Merge-Sort

- Consider a recursive call associated with a node ***v*** of the merge-sort tree T:
  - the **divide step** at node ***v*** runs in time proportional to the size of the sequence for **v**.
  - the **merging step** also takes time that is linear in the size of the merged sequence
  - the **size of the sequence** handled by the recursive call associated with ***v*** is equal to $n/2^i$

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



Analysis of Merge-Sort

- The running time of merge-sort is equal to the sum of the times spent at the nodes of *T.*
  - *recall that T* has exactly 2^i nodes at depth *i*.
  - the overall time spent at all the nodes of T at depth i is O($2^i \cdot n/2^i$), which is **O(n)**.
  - Recall that the height h of the merge-sort tree is **O(log n)**.
  - Therefore, algorithm merge-sort sorts a sequence S of size n in **O(n logn)** time, assuming two elements of S can be compared in O(1) time.

Analysis of Merge-Sort

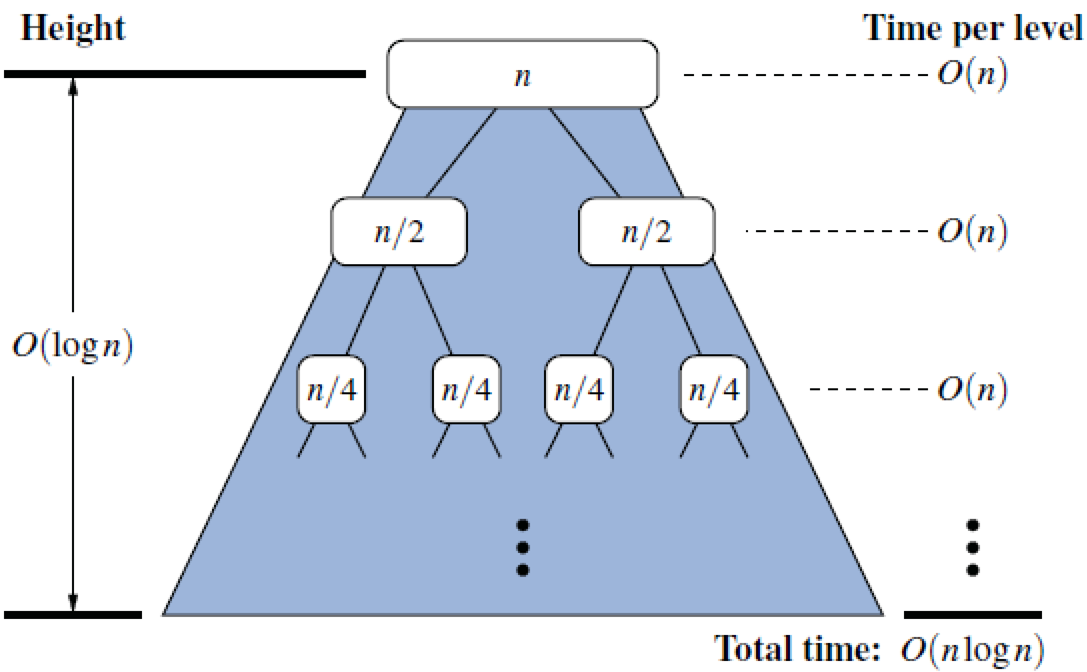


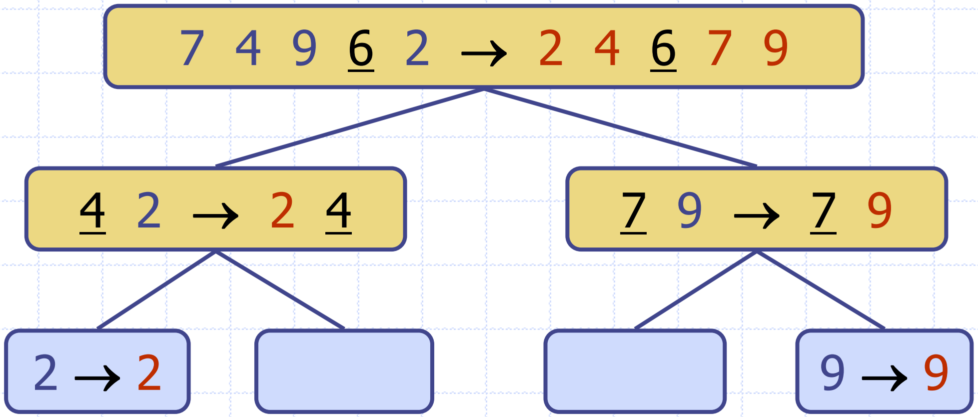
Figure 12.6: A visual analysis of the running time of merge-sort. Each node represents the time spent in a particular recursive call, labeled with the size of its subproblem.

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)

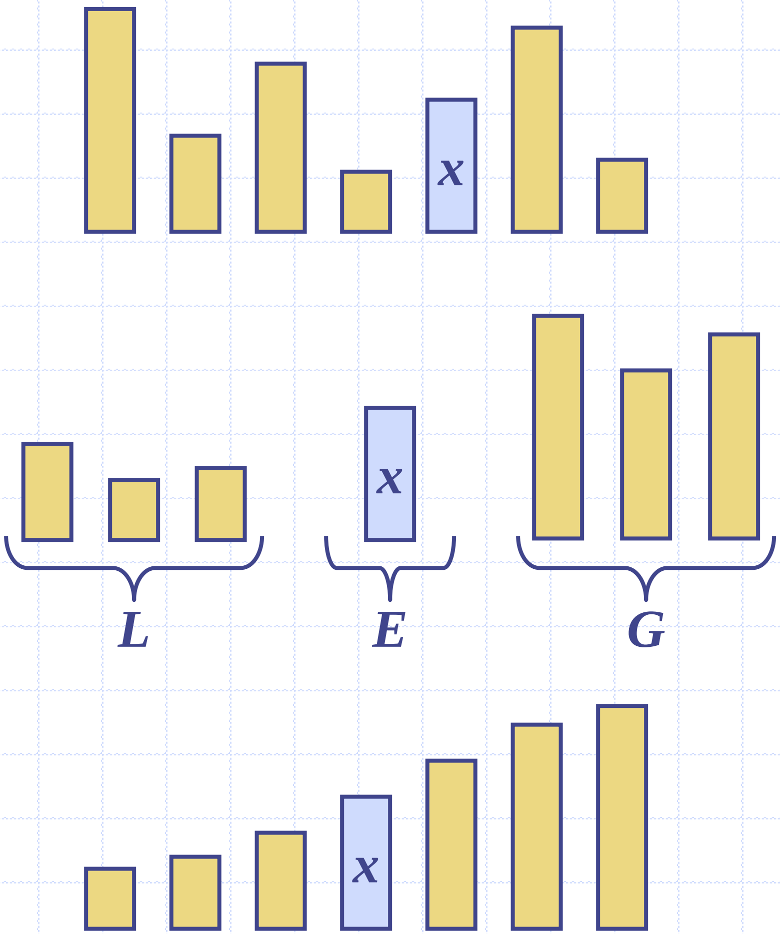
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Quick-Sort

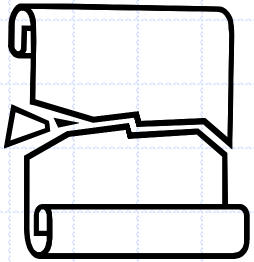


Quick-Sort

- 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

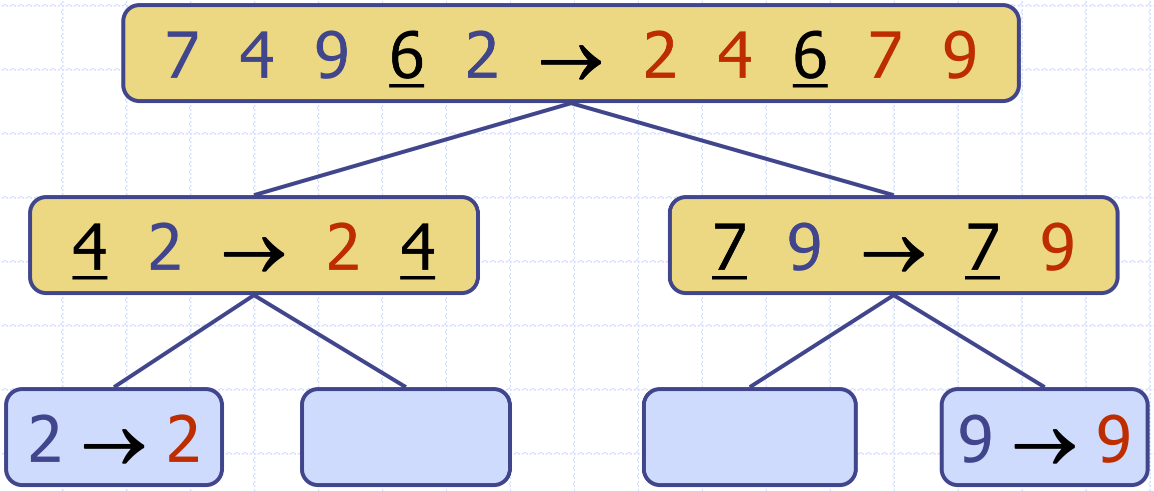
Algorithm *partition*(S, p)
  Input sequence S, position p of pivot
  Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
  $x \leftarrow S.remove(p)$
  while $\neg S.isEmpty()$
    $y \leftarrow S.remove(S.first())$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

Java Implementation

```
/** Quick-sort contents of a queue. */
public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
  int n = S.size( );
  if (n < 2) return;                          // queue is trivially sorted
  // divide
  K pivot = S.first( );                       // using first as arbitrary pivot
  Queue<K> L = new LinkedQueue<>( );
  Queue<K> E = new LinkedQueue<>( );
  Queue<K> G = new LinkedQueue<>( );
  while (!S.isEmpty( )) {                     // divide original into L, E, and G
    K element = S.dequeue( );
    int c = comp.compare(element, pivot);
    if (c < 0)                                // element is less than pivot
      L.enqueue(element);
    else if (c == 0)                          // element is equal to pivot
      E.enqueue(element);
    else                                      // element is greater than pivot
      G.enqueue(element);
  }
  // conquer
  quickSort(L, comp);                         // sort elements less than pivot
  quickSort(G, comp);                         // sort elements greater than pivot
  // concatenate results
  while (!L.isEmpty( ))
    S.enqueue(L.dequeue( ));
  while (!E.isEmpty( ))
    S.enqueue(E.dequeue( ));
  while (!G.isEmpty( ))
    S.enqueue(G.dequeue( ));
}
```

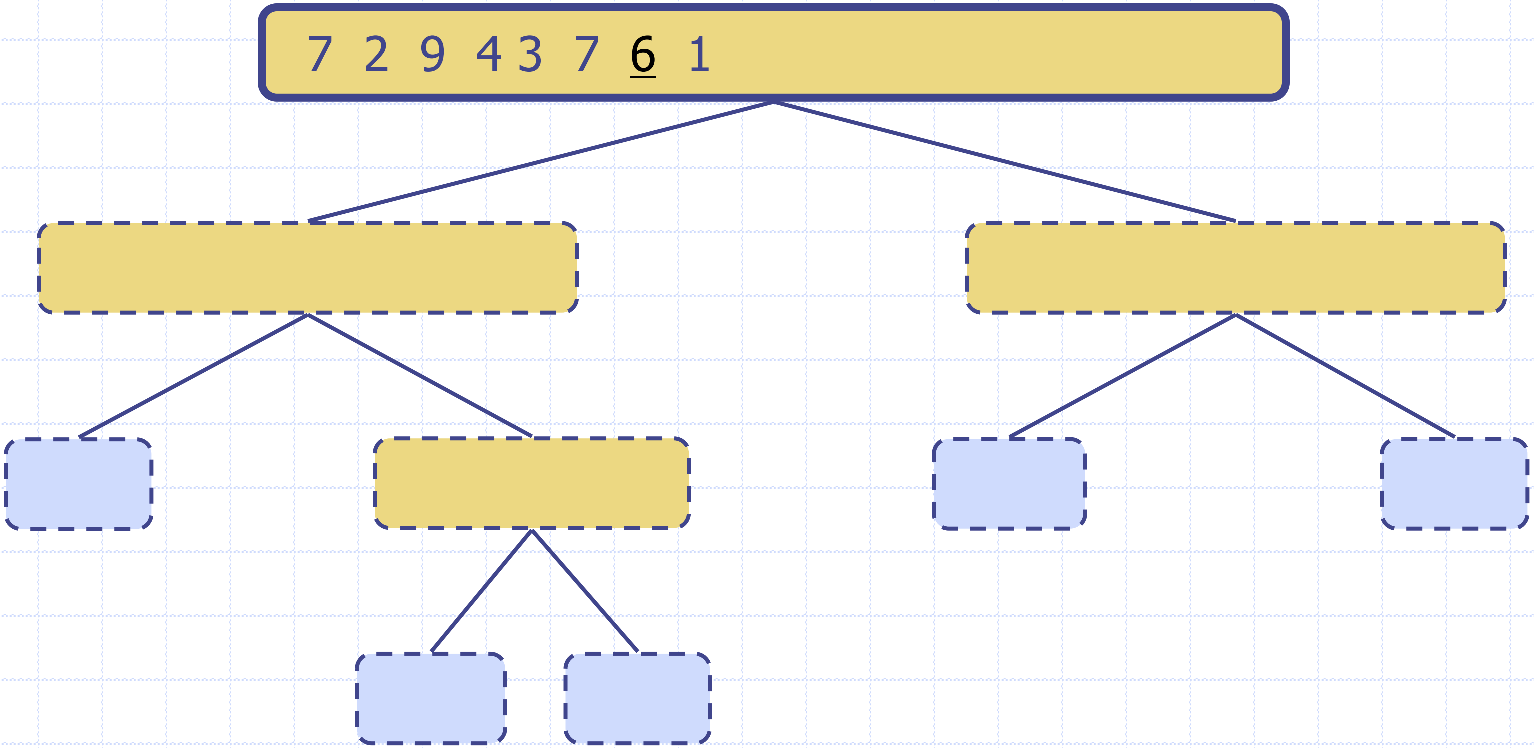
Quick-Sort Tree

- 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



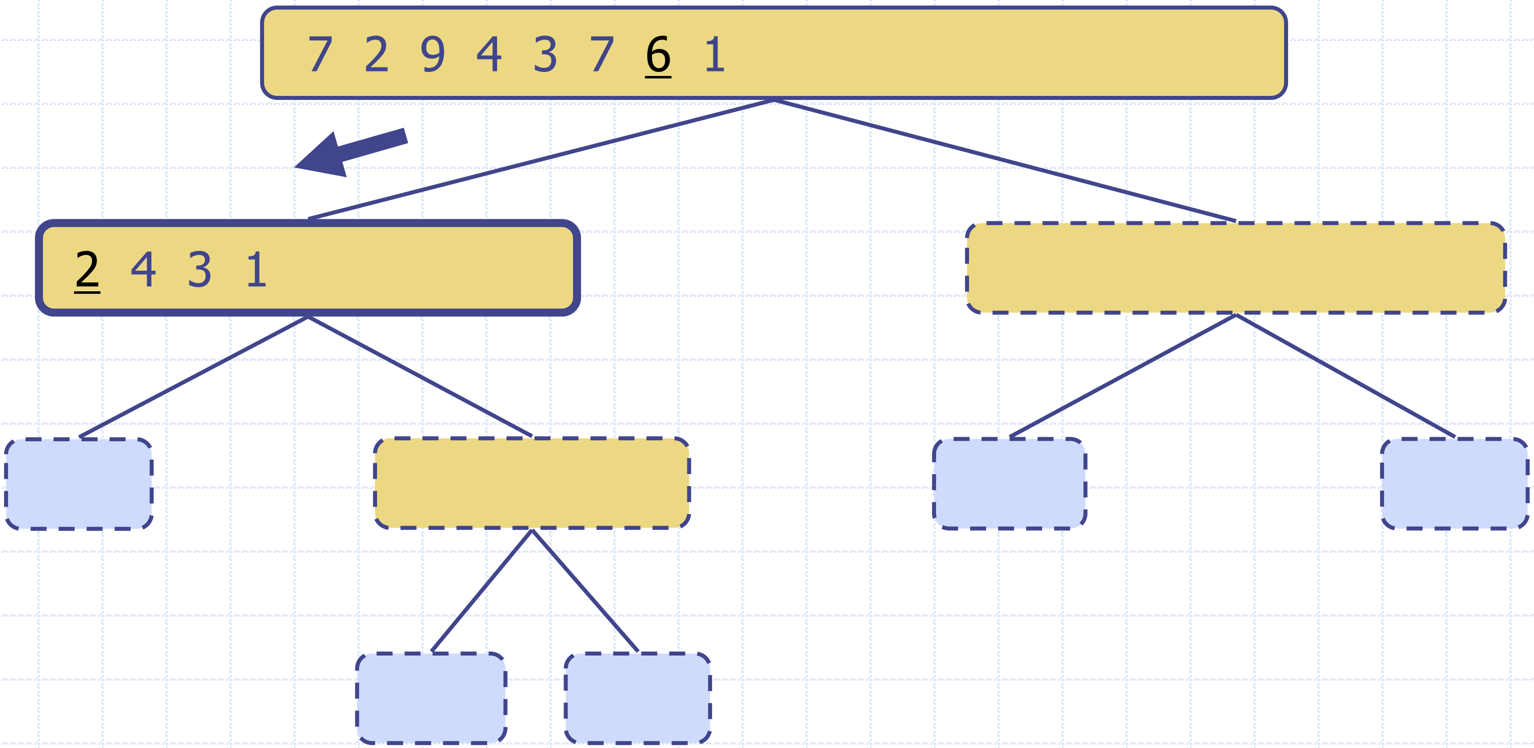
Execution Example

- Pivot selection



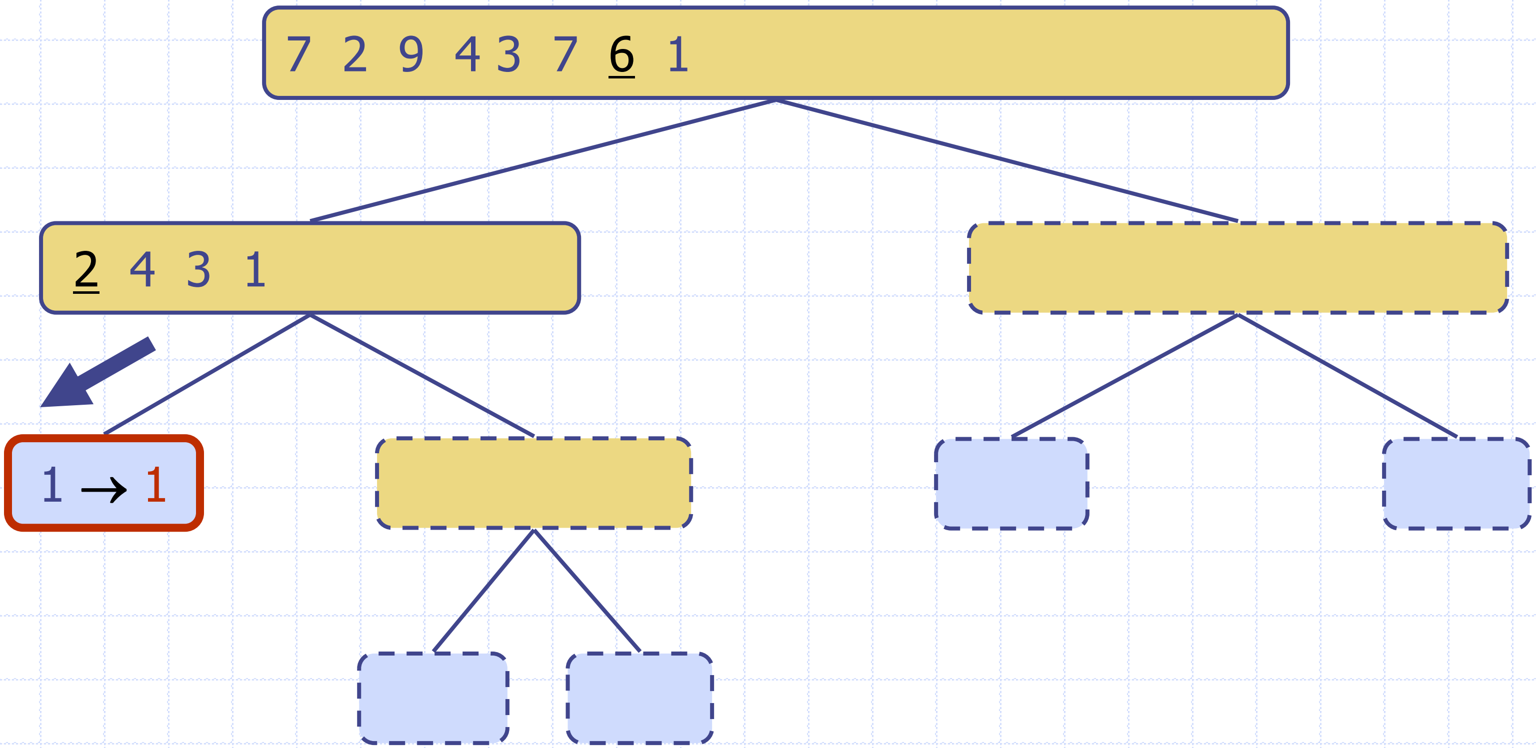
Execution Example (cont.)

- Partition, recursive call, pivot selection



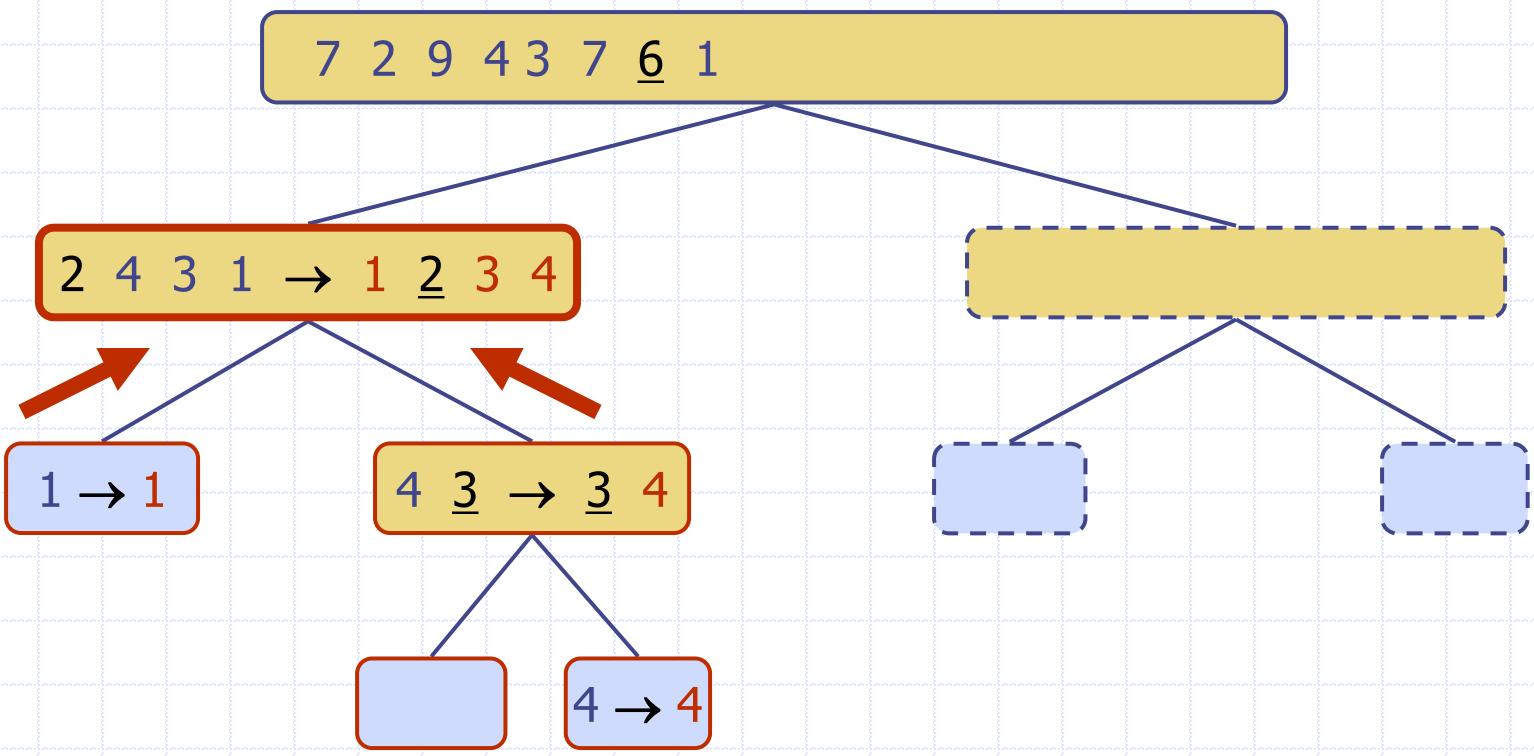
Execution Example (cont.)

- Partition, recursive call, base case



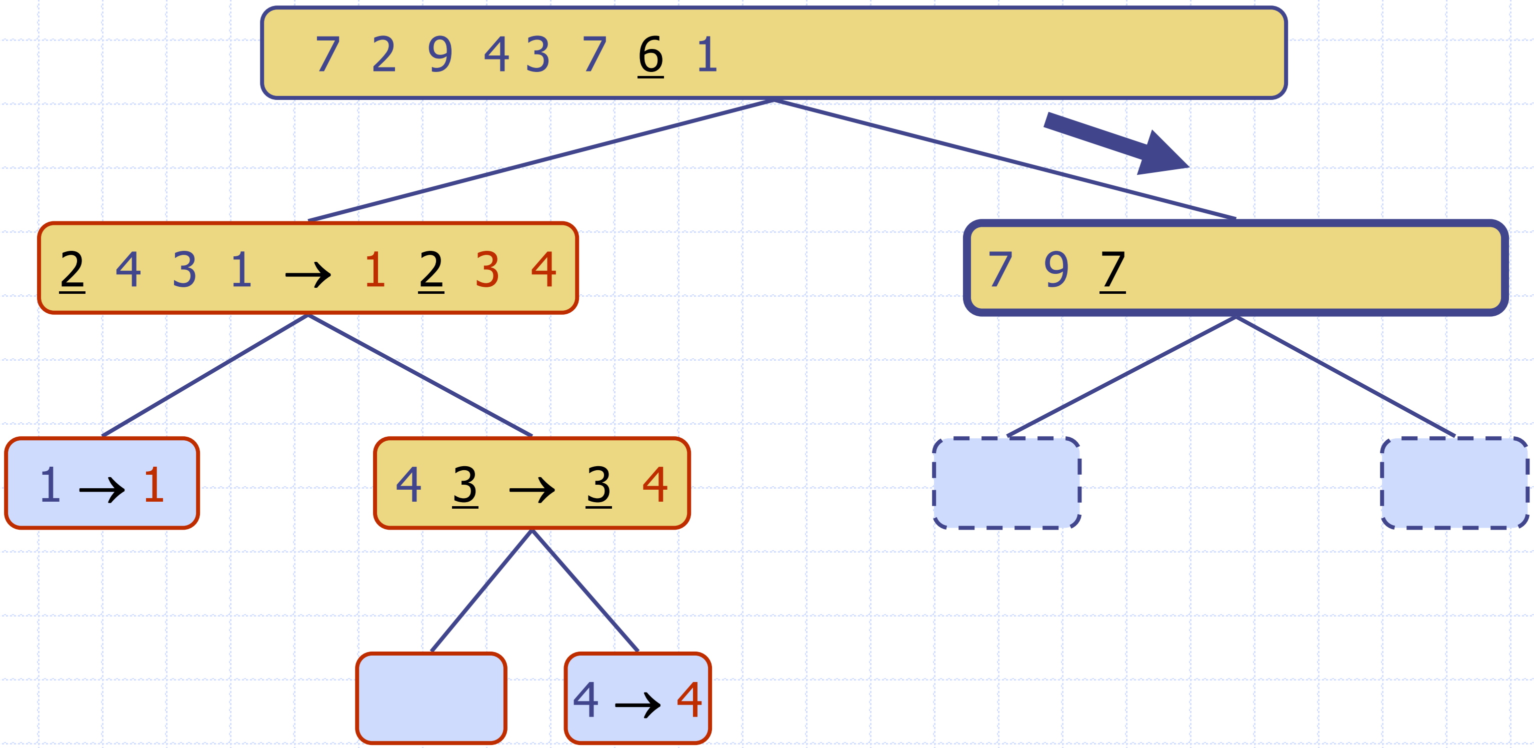
Execution Example (cont.)

- Recursive call, …, base case, join



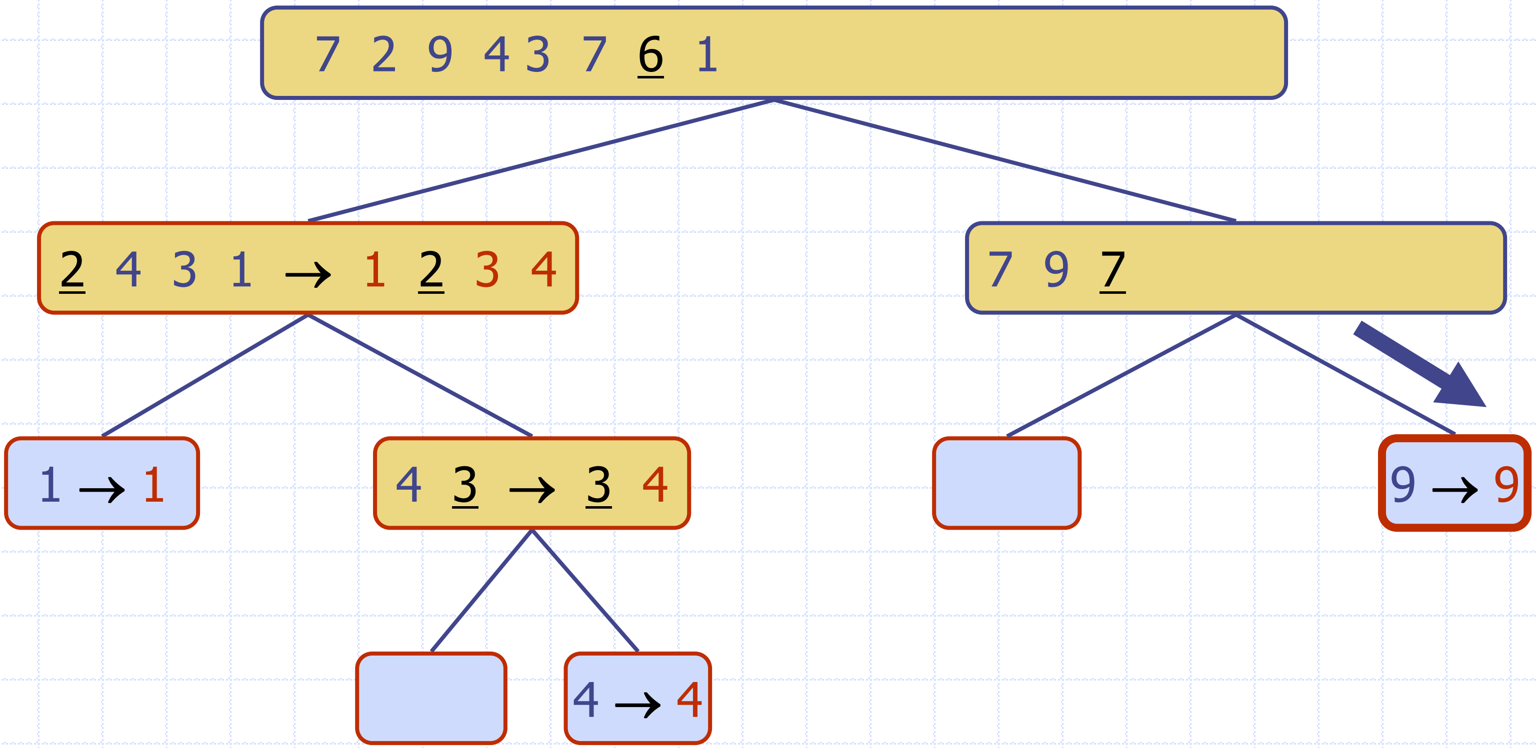
Execution Example (cont.)

- Recursive call, pivot selection



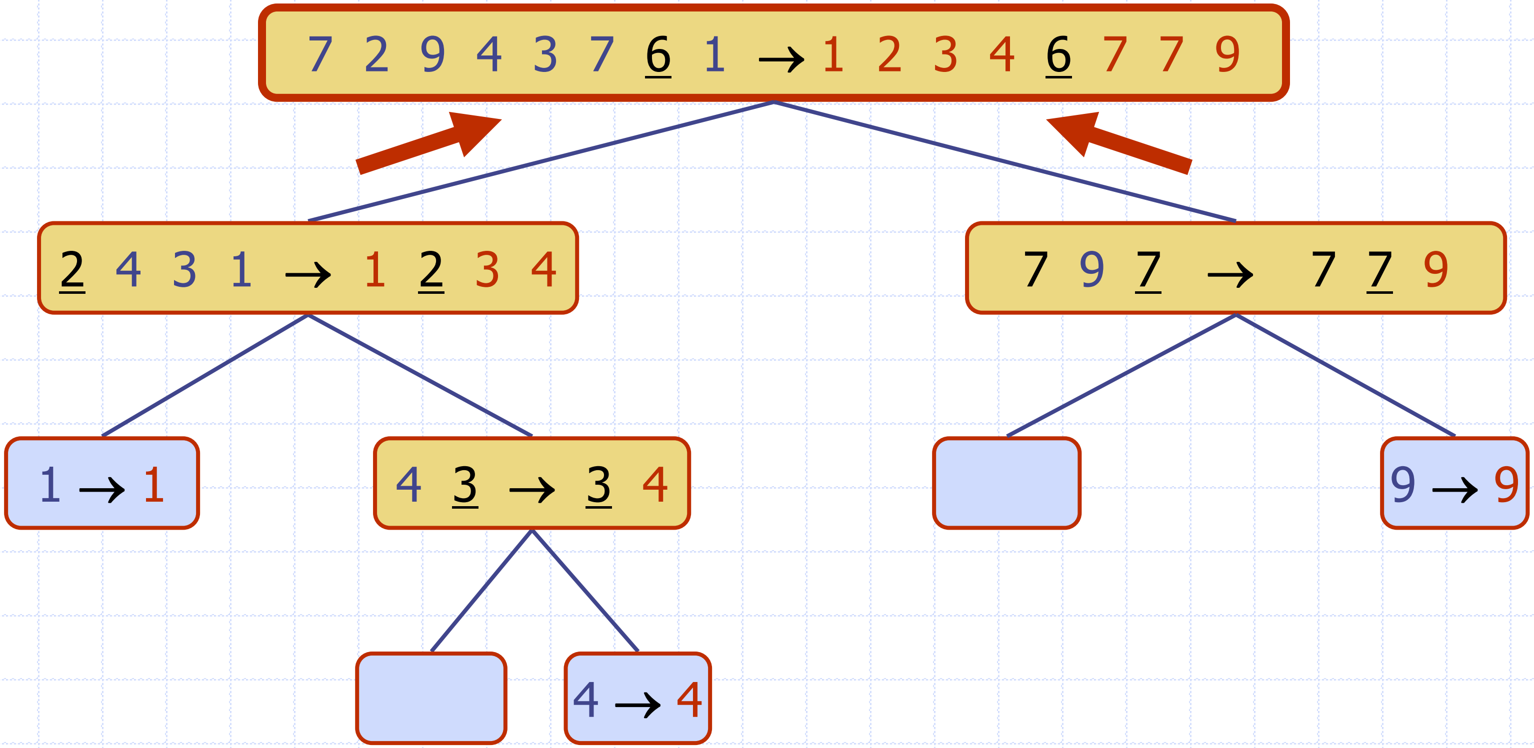
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



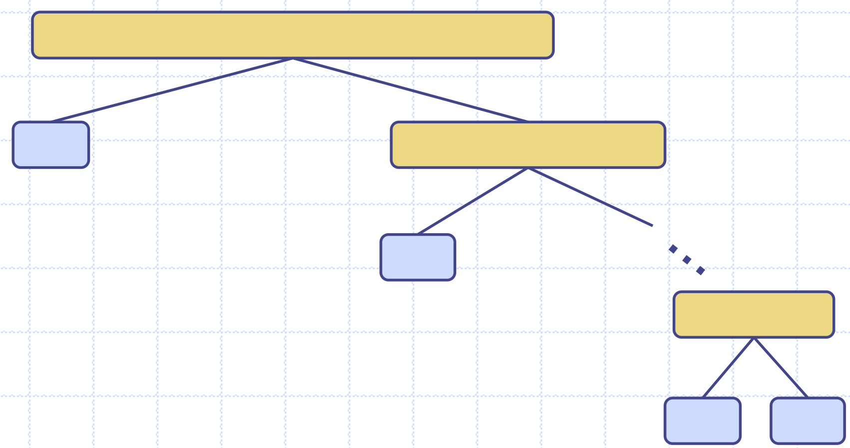
Worst-case Running Time

- 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

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

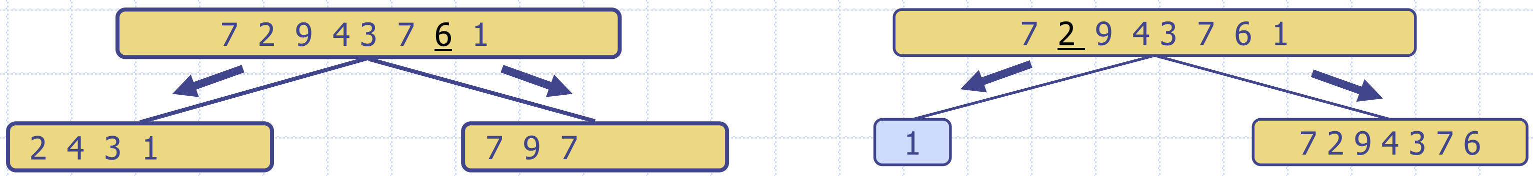
- Thus, the worst-case running time of quick-sort is $O(n^2)$

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



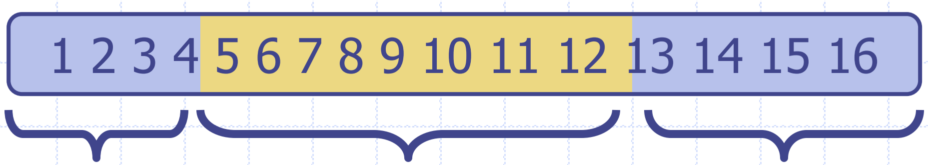
Expected Running Time

- 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**

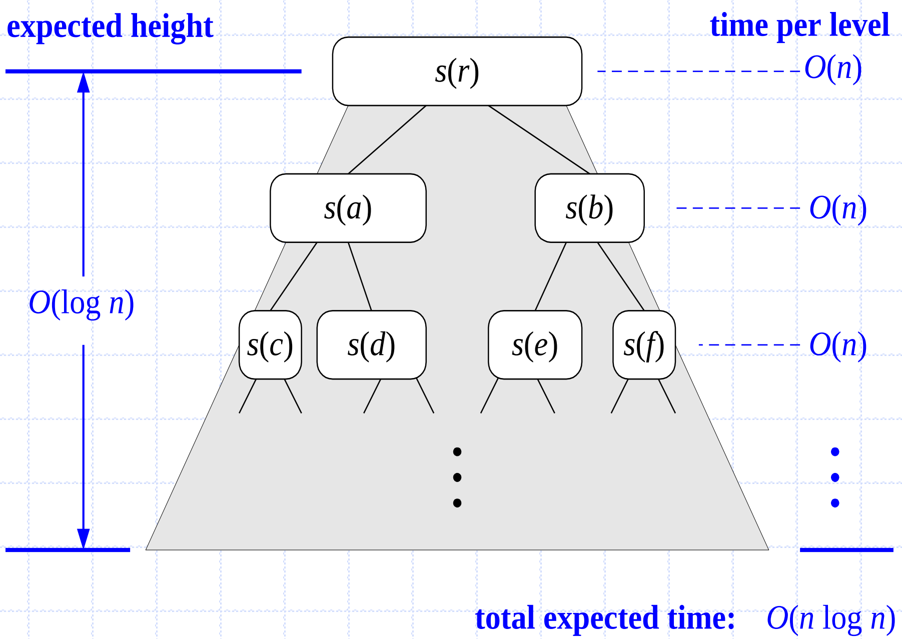
- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



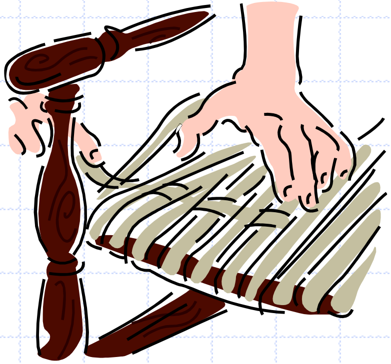
Bad pivots **Good pivots** **Bad pivots**

Expected Running Time, Part 2

- We can analyze the running time of quick-sort with **the same technique used for merge-sort**.
- Namely, we can **identify the time spent at each node** of the quick-sort tree T and sum up the running times for all the nodes.
- 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 $O(n \log n)$**
- With a more rigorous analysis, it can be shown that the running time of randomized quick-sort is **O(nlog n) with high probability**.



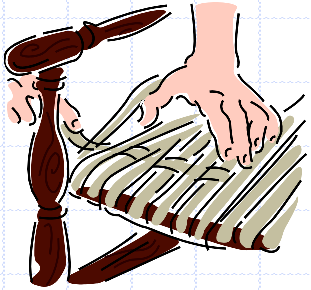
In-Place Quick-Sort



- 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 **6** 9 (pivot = 6)

- Repeat until j and k cross:
  - Scan j to the right until finding an element $\geq$ 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 **6** 9

Java Implementation

```
/** Sort the subarray S[a..b] inclusive. */
private static <K> void quickSortInPlace(K[ ] S, Comparator<K> comp,
                                                                  int a, int b) {
  if (a >= b) return;              // subarray is trivially sorted
  int left = a;
  int right = b−1;
  K pivot = S[b];
  K temp;                          // temp object used for swapping
  while (left <= right) {
    // scan until reaching value equal or larger than pivot (or right marker)
    while (left <= right && comp.compare(S[left], pivot) < 0) left++;
    // scan until reaching value equal or smaller than pivot (or left marker)
    while (left <= right && comp.compare(S[right], pivot) > 0) right−−;
    if (left <= right) {           // indices did not strictly cross
      // so swap values and shrink range
      temp = S[left]; S[left] = S[right]; S[right] = temp;
      left++; right−−;
    }
  }
  // put pivot into its final place (currently marked by left index)
  temp = S[left]; S[left] = S[b]; S[b] = temp;
  // make recursive calls
  quickSortInPlace(S, comp, a, left − 1);
  quickSortInPlace(S, comp, left + 1, b);
}
```

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)
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)