Sorting Algorithms

Sorting

- Putting an unsorted list of data elements into order – sorting - is a very common and useful operation
- We describe efficiency by relating the number of comparisons to the number of elements in the list (N)

Simple Sorts

- In this section we present three "simple" sorts
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Properties of these sorts
  - use an unsophisticated brute force approach
  - are not very efficient
  - are easy to understand and to implement

Selection Sort -- Example

	initial		after $i = 0$		after $i = 1$		after $i = 2$		after $i = 3$		after $i = 4$
0	4	<u>0</u>	1	0	1	0	1	0	1	0	1
1	2	1	2	<u>1</u>	2	1	2	1	2	1	2
2	3	2	3	2	3	<u>2</u>	3	2	3	2	3
3	1	3	4	3	4	3	4	<u>3</u>	4	3	4
4	6	4	6	4	6	4	6	4	6	<u>4</u>	5
5	5	5	5	5	5	5	5	5	5	5	6

Selection Sort

```
for (i = 0; i < n-1; i++)
{
  lowindex = i;
  for (j = i+1; j < n; j++)
  {
     if (A[j].key < A[lowindex].key) {
         lowindex = j;
     }
  }
  swap(A[i], A[lowindex]);
}
```

Selection Sort algorithm is $O(N^2)$

Insertion Sort -- Example

	initial	after $i = 1$	after $i = 2$	after $i = 3$	after $i = 4$	after $i = 5$
0	4	2	2	1	1	1
1	2	4	3	2	2	2
2	3	3	4	3	3	3
3	1	1	1	4	4	4
4	6	6	6	6	6	5
5	5	5	5	5	5	6

Insertion Sort

```
for (i = 1; i < n; i++)
{
   j = i;
   while (j!= 0 && A[j] < A[j-1])
   {
       swap(A[j], A[j-1]);
       j = j-1;
   }
}
```

Insertion Sort algorithm is O(N^2)

Bubble Sort -- Example

	initial	after $i = 0$	after $i = 1$	after $i = 2$	after $i = 3$	after $i = 4$
0	4	1	1	1	1	1
1	2	4	2	2	2	2
2	3	2	4	3	3	3
3	1	3	3	4	4	4
4	6	5	5	5	5	5
5	5	6	6	6	6	6

Bubble Sort

```
for (i = 0; i < n-1; i++)
{
  for (j = n - 1; j > i; j--)
  {
     if (A[j].key < A[j-1].key)
         swap(A[j], A[j-1]);
  }
}
```

Bubble Sort algorithm is O(N^2)

Heap Sort

- In *max heap*, the maximum value of a heap is in the root node.
- The general approach of the Heap Sort is as follows:
  - take the root (maximum) element off the heap, and put it into its place.
  - reheap the remaining elements. (This puts the next-largest element into the root position.)
  - repeat until there are no more elements.
- For this to work we must first arrange the original array into a heap

Heap Sort -- Example

	initial	step 1	step 2	step 3	step 4	step 5
1	4	4	4	4	6	6
2	2	2	2	2	4	4
3	3	3	3	3	3	5
4	1	1	1	1	1	1
5	6	6	6	6	2	2
6	5	5	5	5	5	3

Heap Sort -- Example

	after phase 1	step 1	step 2	step 3	step 4	step 5
1	6	5	4	3	2	1
2	4	4	2	2	1	2
3	5	3	3	1	3	3
4	1	1	1	4	4	4
5	2	2	5	5	5	5
6	3	6	6	6	6	6

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 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

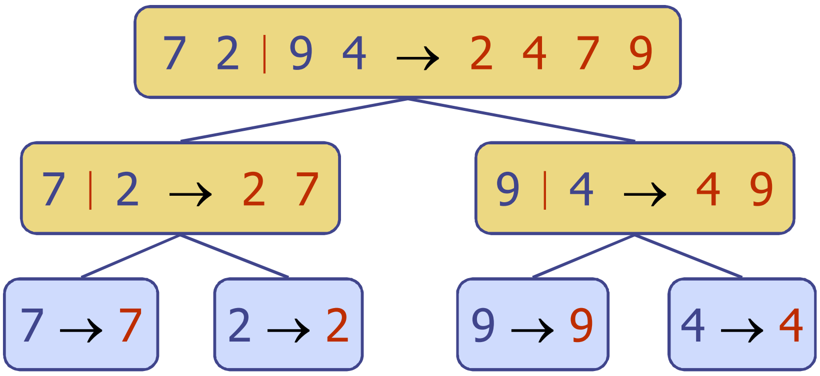
- The conquer step of merge-sort consists of merging two sorted sequences A and B into a sorted sequence S containing the union of the elements of A and B
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time

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.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
```

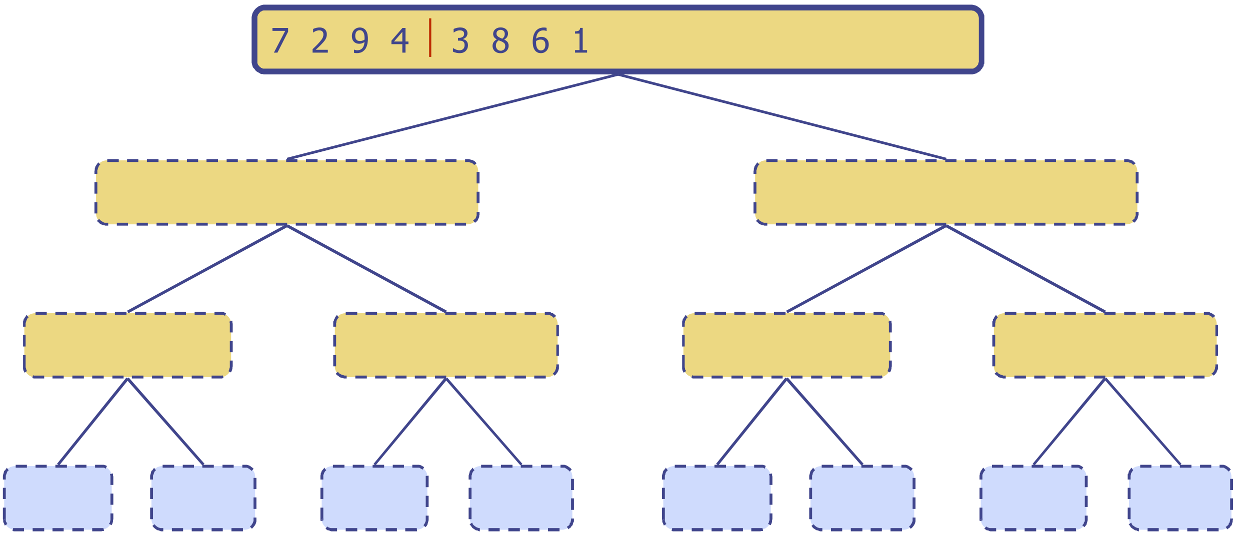

Merge-Sort Tree

- 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



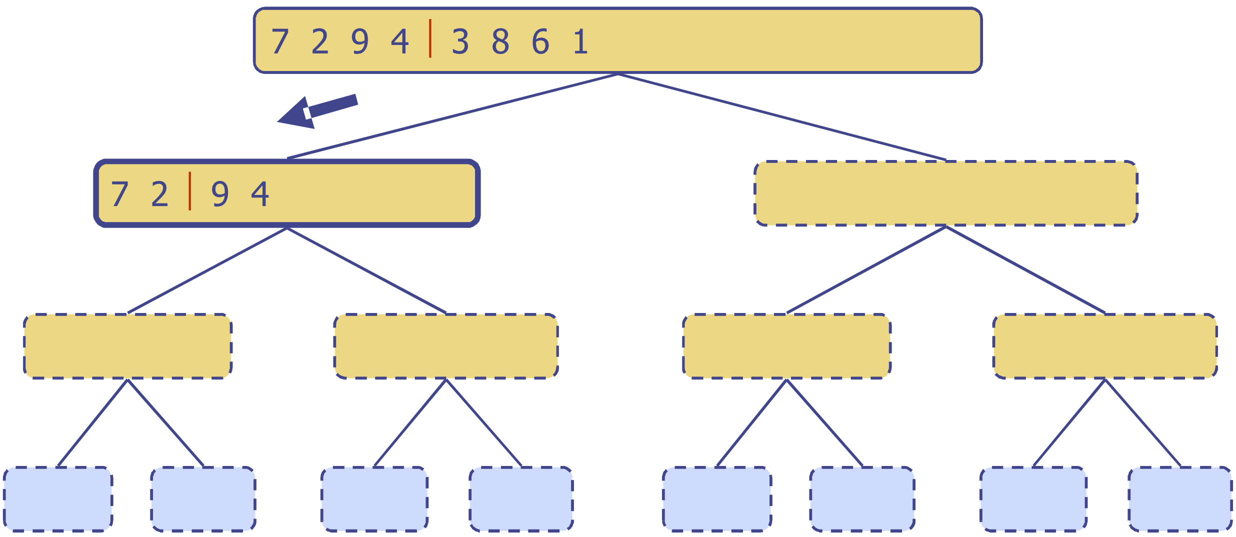
Execution Example

- Partition



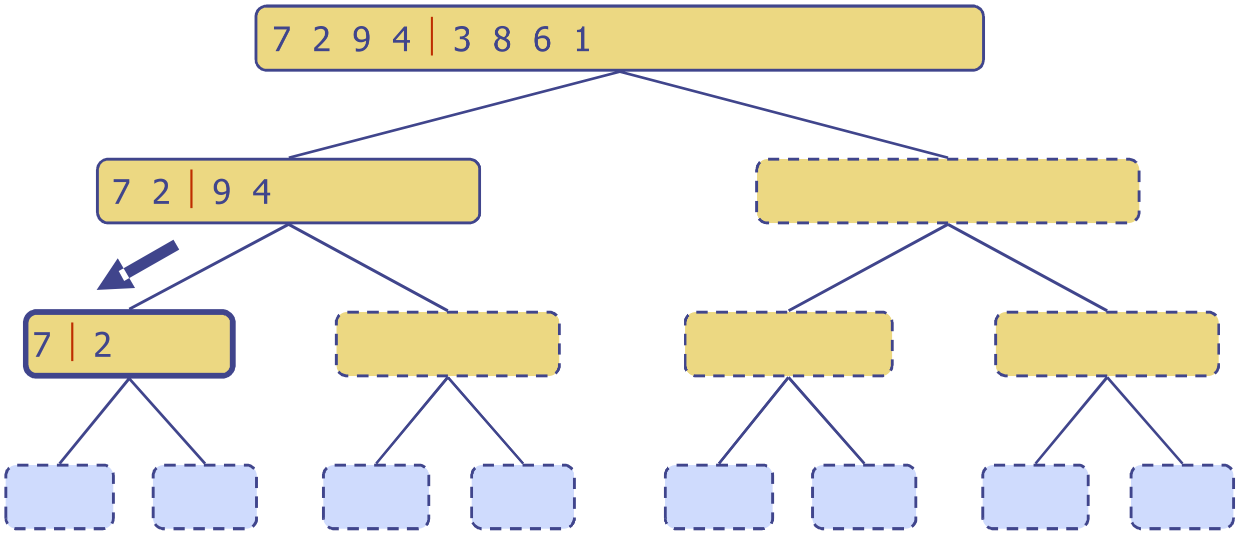
Execution Example (cont.)

- Recursive call, partition



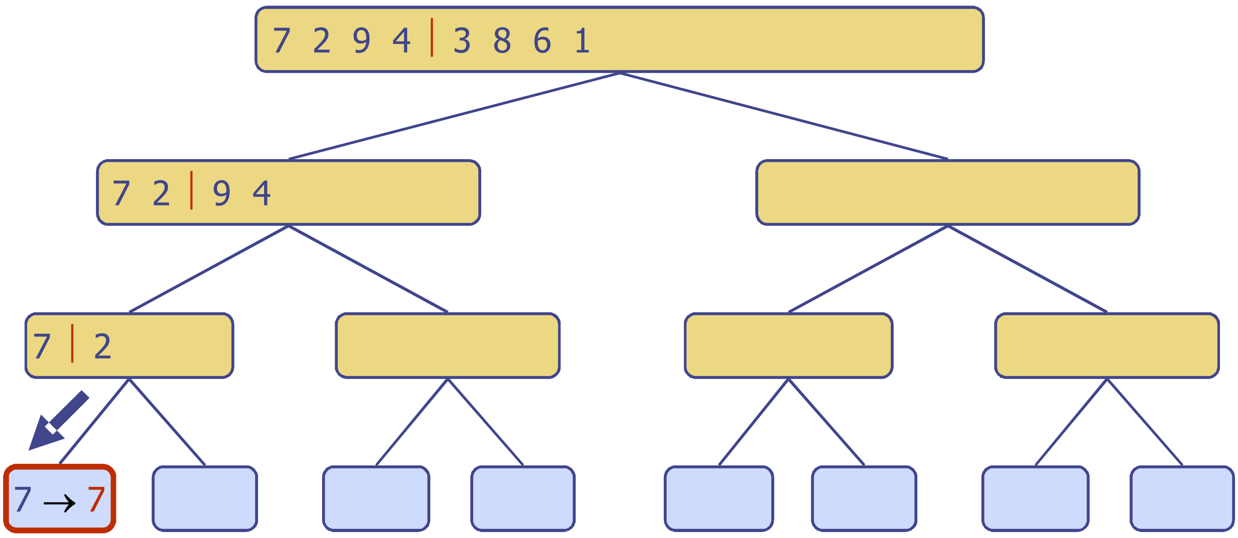
Execution Example (cont.)

- Recursive call, partition



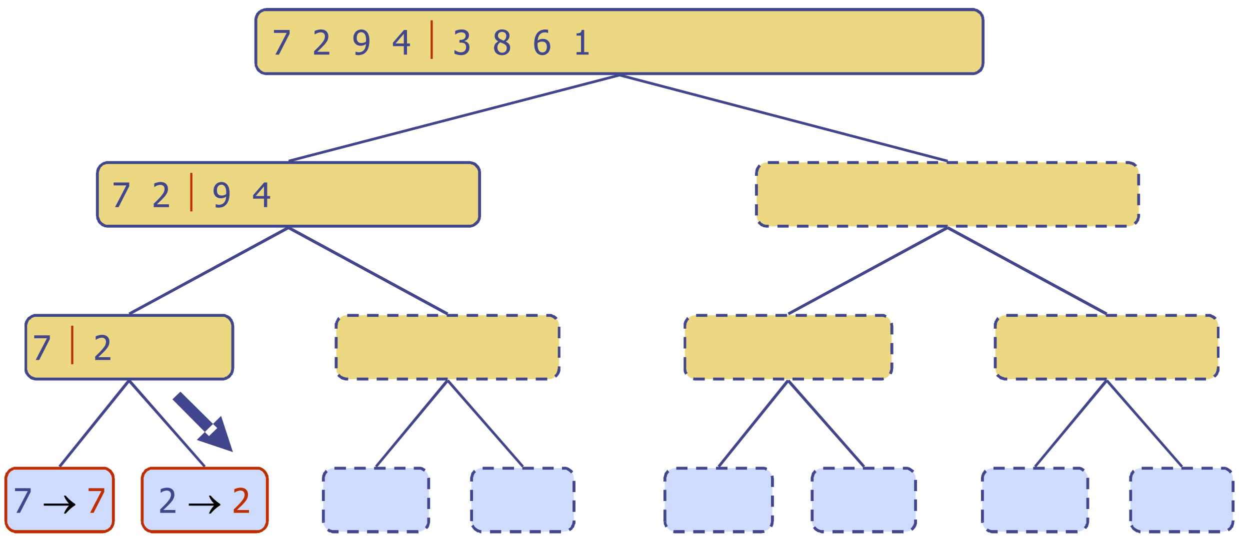
Execution Example (cont.)

- Recursive call, base case



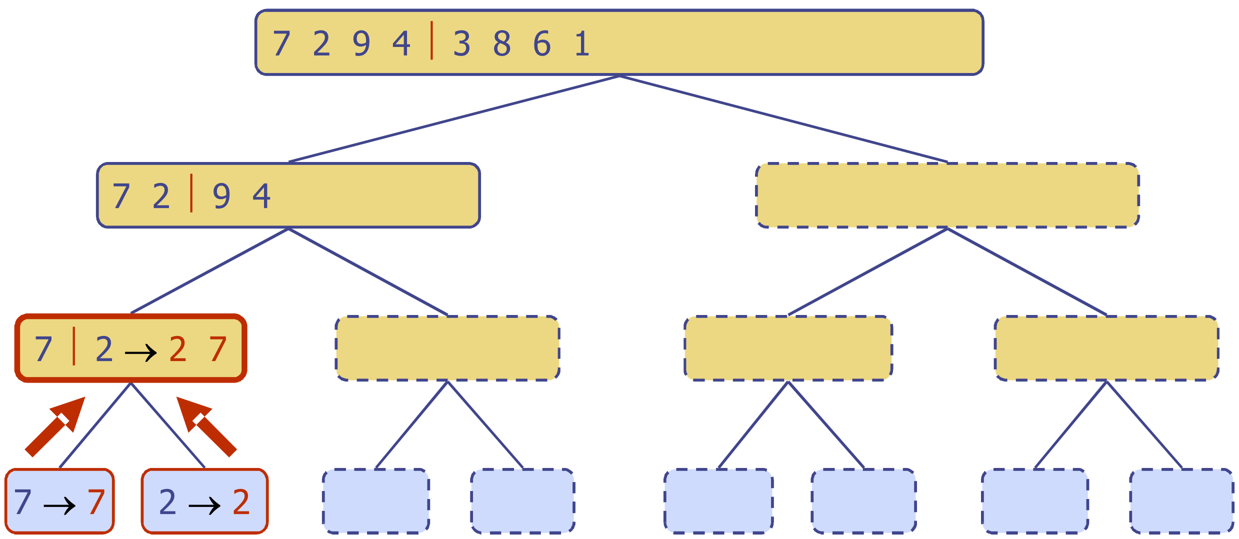
Execution Example (cont.)

- Recursive call, base case



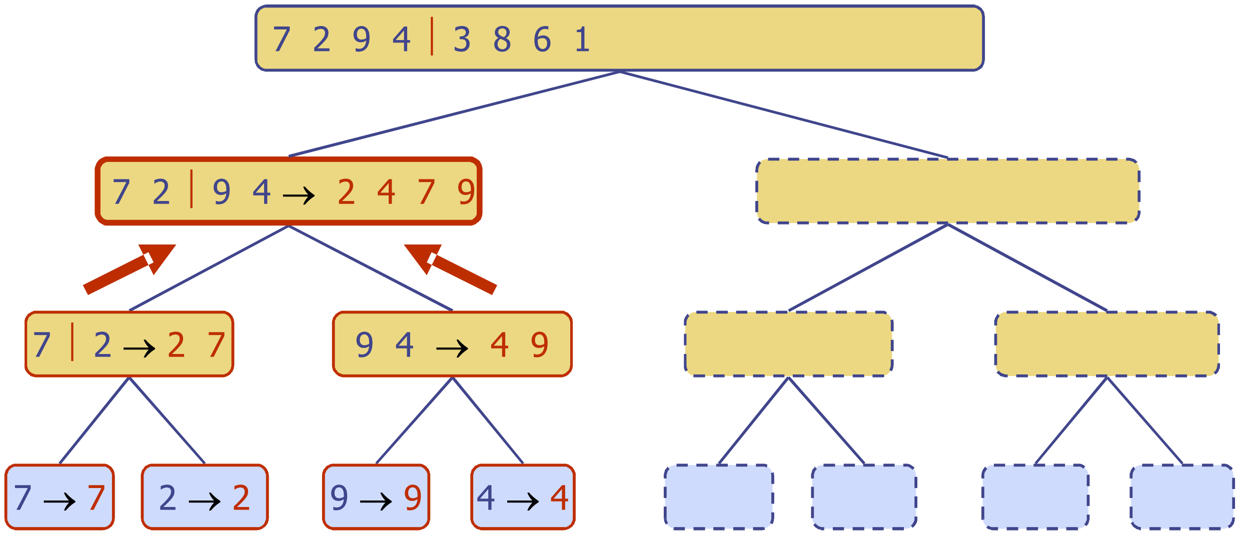
Execution Example (cont.)

- Merge



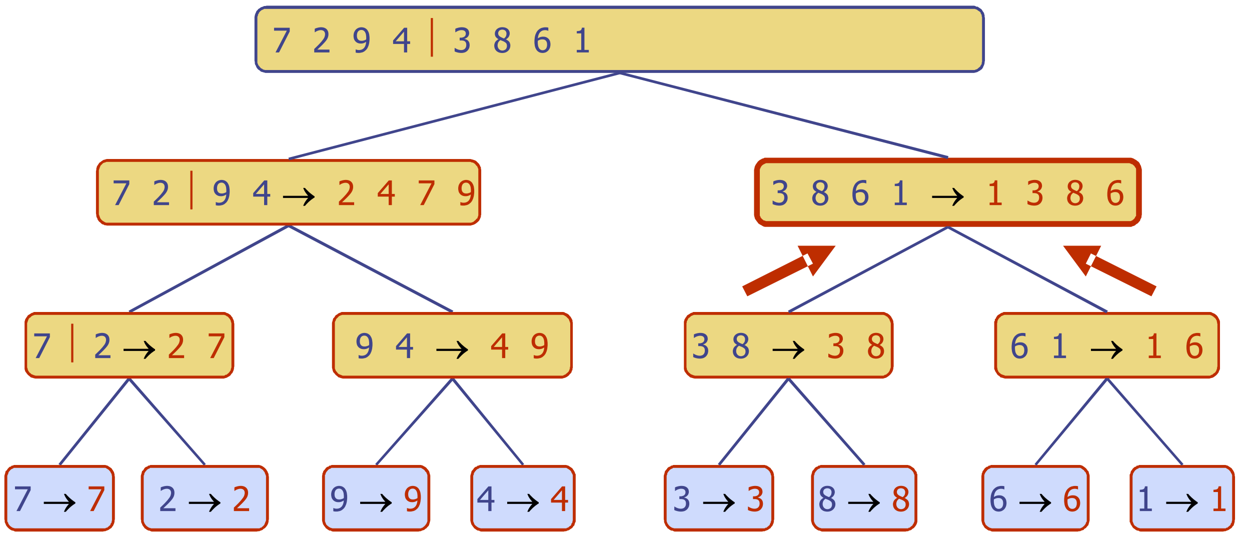
Execution Example (cont.)

- Merge



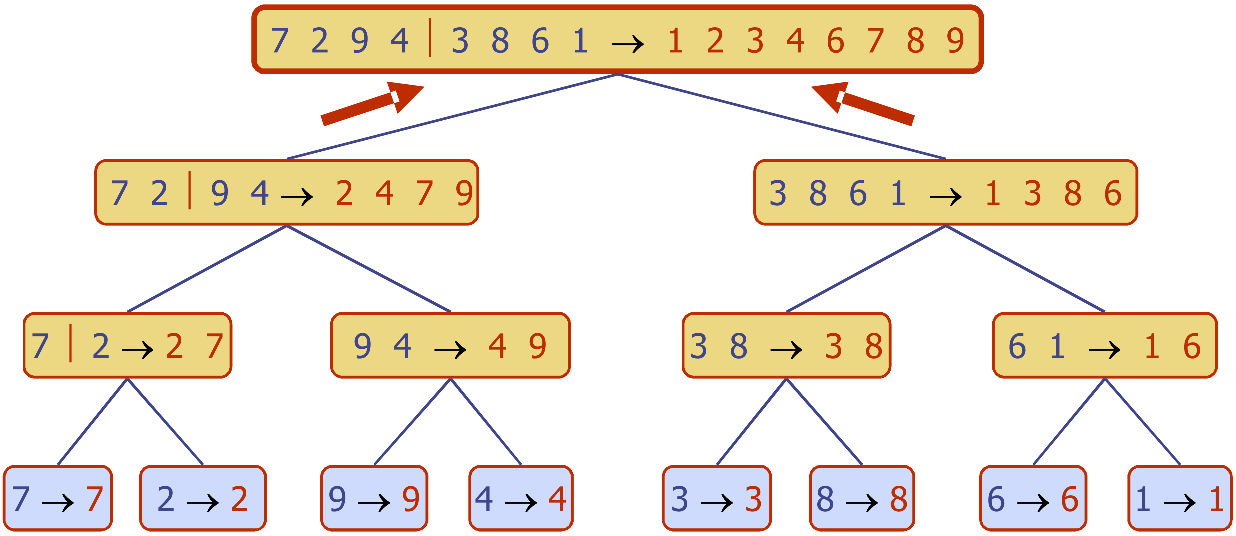
Execution Example (cont.)

- Recursive call, …, merge, merge



Execution Example (cont.)

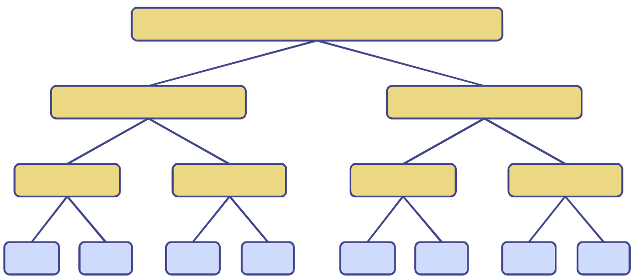
- Merge



Analysis of Merge-Sort

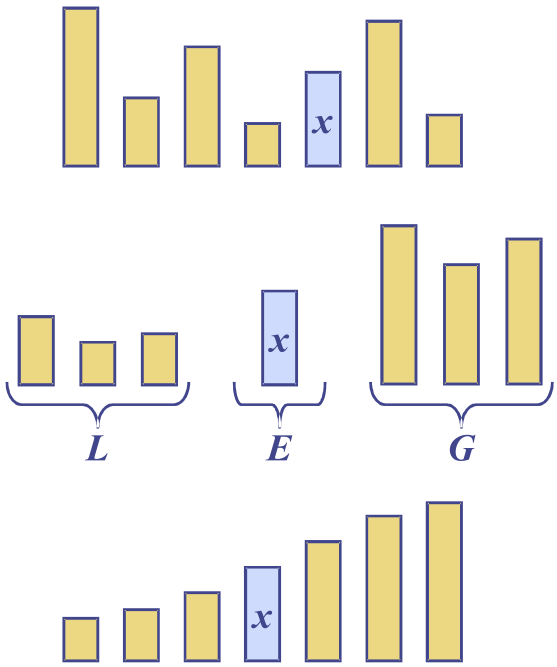
- The height h of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
- The overall amount of work done at the nodes of depth i is $O(n)$
  - we partition and merge 2^i sequences of size $n/2^i$
  - we make 2^{i+1} recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

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



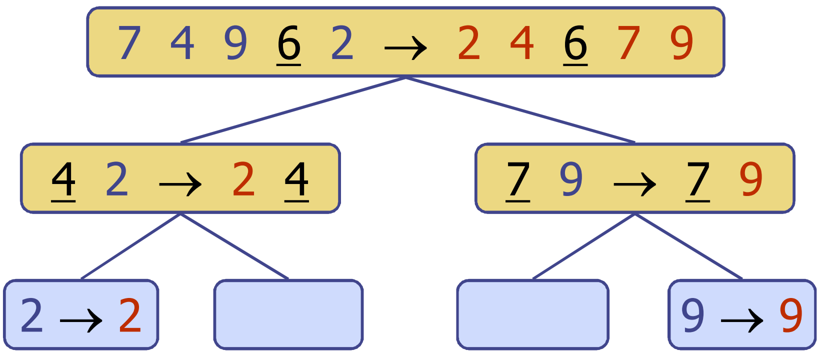
Quick-Sort

- 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



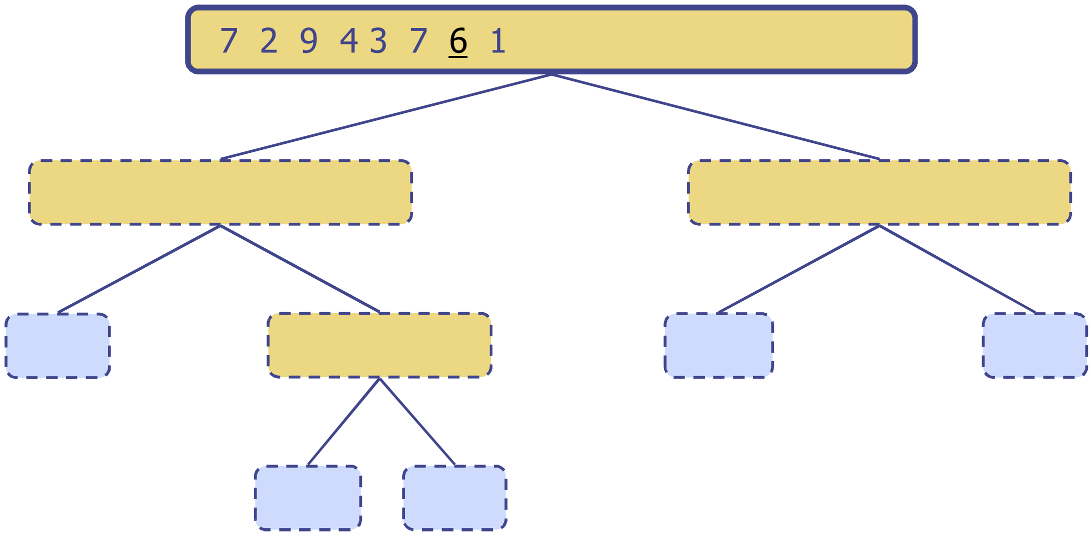
Quick-Sort Tree

- 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



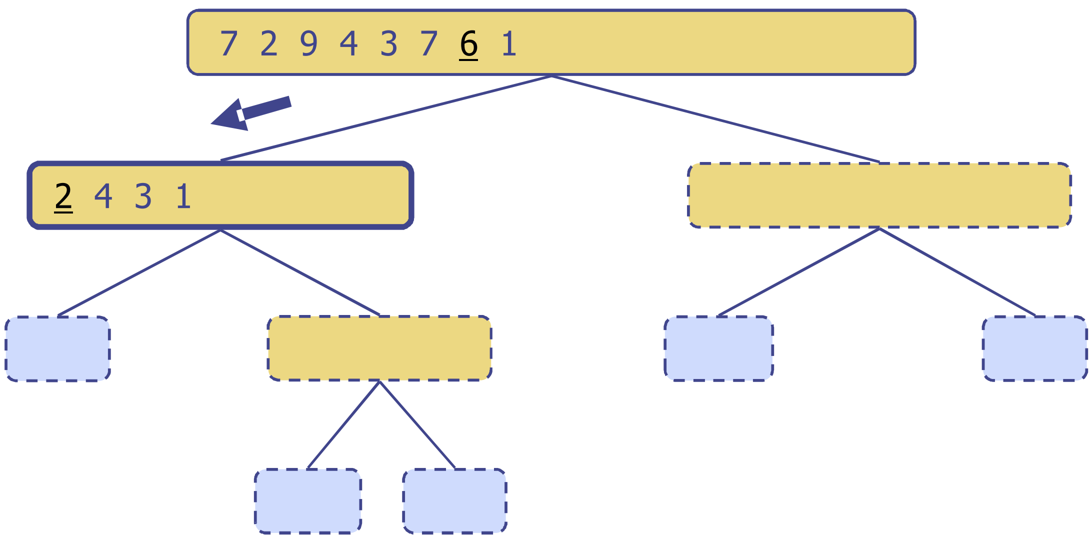
Execution Example

- Pivot selection



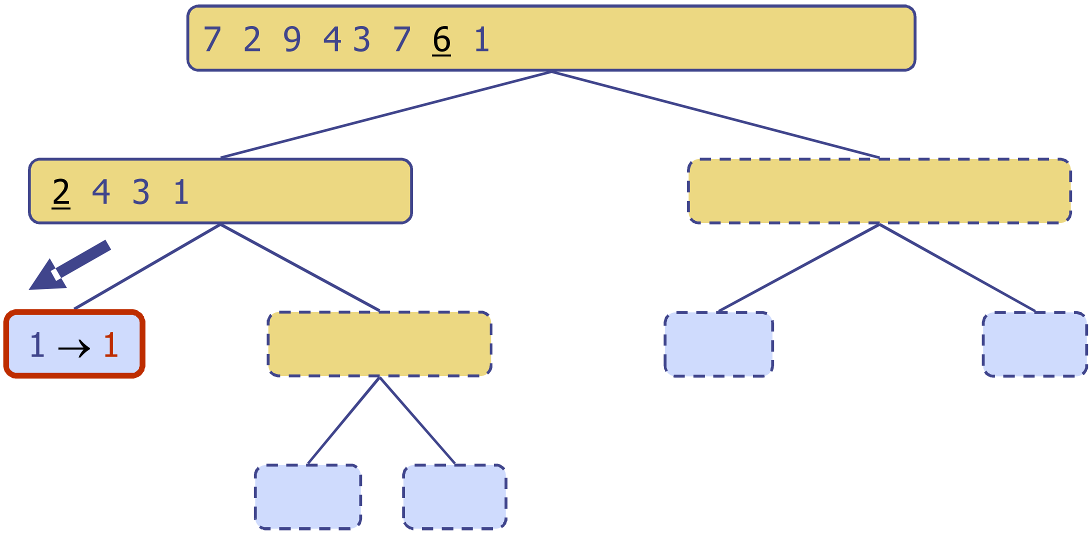
Execution Example (cont.)

- Partition, recursive call, pivot selection



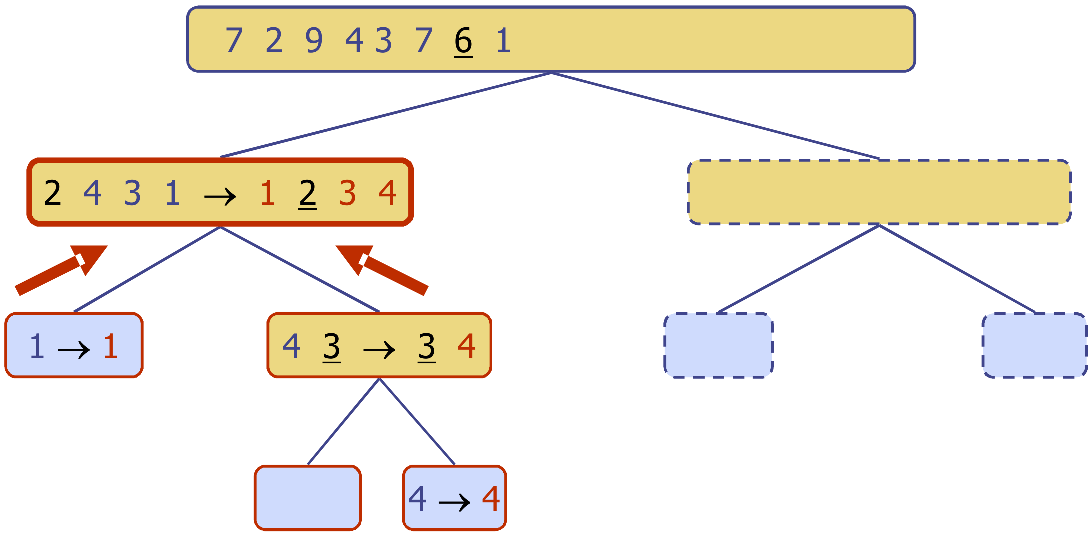
Execution Example (cont.)

- Partition, recursive call, base case



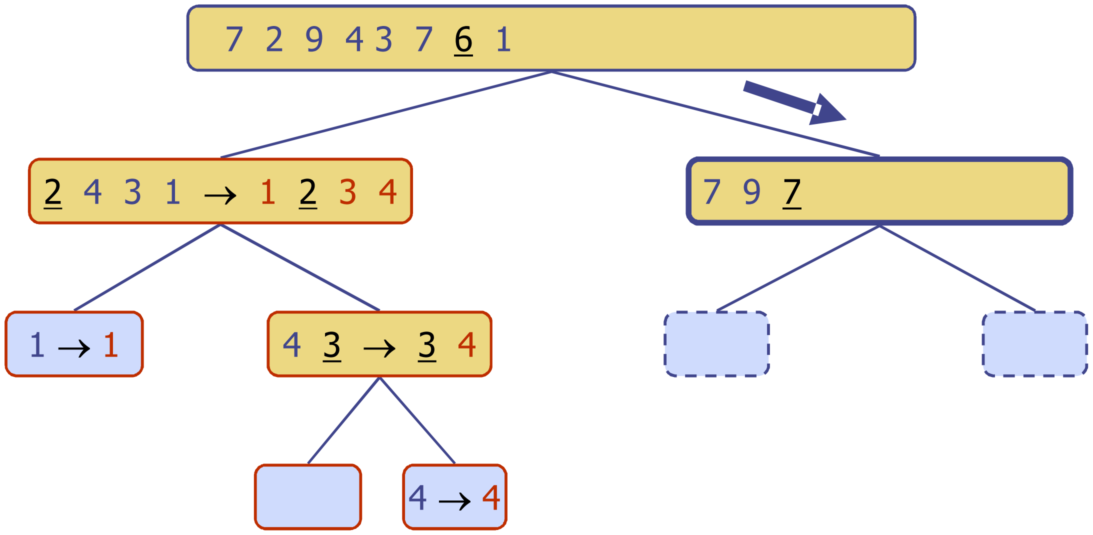
Execution Example (cont.)

- Recursive call, …, base case, join



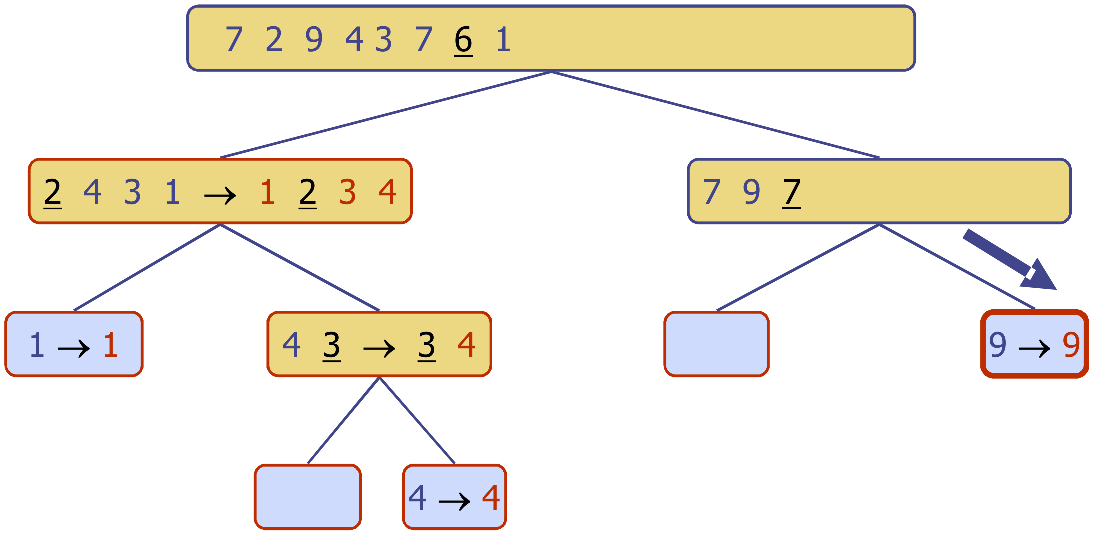
Execution Example (cont.)

- Recursive call, pivot selection



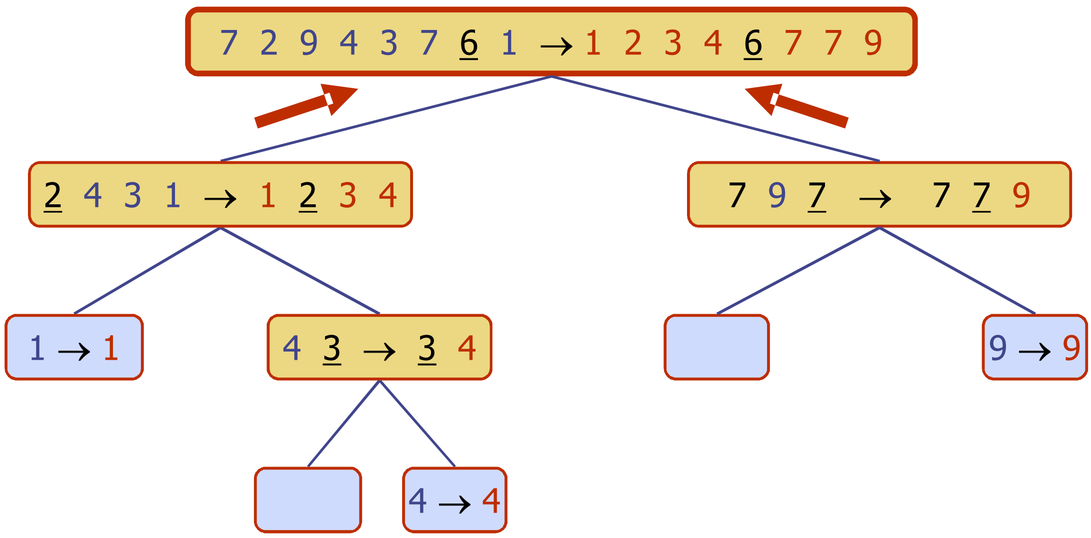
Execution Example (cont.)

- Partition, …, recursive call, base case



Execution Example (cont.)

- Join, join



In-Place Quick Sort

Algorithm *inPlaceQuickSort(S,* a, b)
  if $a \geq b$ **then return** { empty subrange }
  $p \leftarrow$ *S.elementAtRank(b)* {pivot}
  $l \leftarrow a$ { will scan rightward}
  $r \leftarrow b - 1$
  while $l \leq r$
    {find an element larger than pivot}
    while $l \leq r$ **and** *S.elemAtRank(l)* $\leq p$ **do**
      $l \leftarrow l + 1$
    {find an element smaller than pivot}
    while $l \leq r$ **and** *S.elemAtRank(r)* $\geq p$ **do**
      $r \leftarrow r - 1$
    if $l < r$ **then**
      S.swapElements(S.atRank(l), S.atRank(r))
  {put the pivot into its final place}
  S.swapElements(S.atRank(l), S.atRank(b))
  inPlaceQuickSort(S, a, l-1)
  inPlaceQuickSort(S, l+1, b)

In-Place Quick Sort

```
Algorithm inPlaceQuickSort(S, a, b)
    if a ≥ b then return  { empty subrange }
    p ← S.elementAtRank(a)    {pivot}
    l ← a + 1   { will scan rightward}
    r ← b { will scan leftward}
    while l ≤ r
        {find an element larger than pivot}
        while l ≤ r and S.elemAtRank(l) ≤ p do
            l ← l + 1
        {find an element smaller than pivot}
        while l ≤ r and S.elemAtRank(r) ≥ p do
            r ← r – 1
        if l < r then
            S.swapElements(S.atRank(l), S.atRank(r))
    {put the pivot into its final place}
    S.swapElements(S.atRank(a), S.atRank(r))
    inPlaceQuickSort(S, a, r-1)
    inPlaceQuickSort(S, r+1, b)
```

In-Place Quick Sort

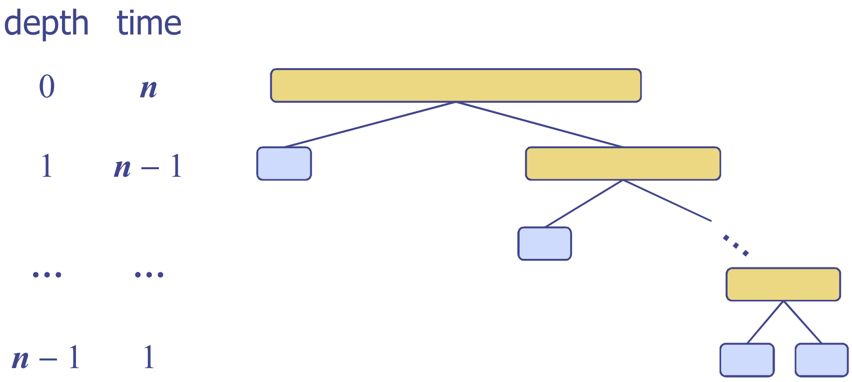
```
Algorithm inPlaceQuickSort(S, a, b)
    if a ≥ b then return  { empty subrange }
    p ← S.elementAtRank(b)     {pivot}
    l ← a   { will scan rightward}
    r ← b - 1 { will scan leftward}
    while l ≤ r
        {find an element larger than pivot}
        while l ≤ r and S.elemAtRank(l) ≤ p do
            l ← l + 1
        {find an element smaller than pivot}
        while l ≤ r and S.elemAtRank(r) ≥ p do
            r ← r – 1
        if l < r then
            S.swapElements(S.atRank(l), S.atRank(r))
    {put the pivot into its final place}
    S.swapElements(S.atRank(l), S.atRank(b))
    inPlaceQuickSort(S, a, l-1)
    inPlaceQuickSort(S, l+1, b)
```

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 $O(n^2)$



Expected Running Time

- On the first call, every element in the array is compared to the dividing value (the "split value"), so the work done is O(N).
- The array is divided into two sub arrays (not necessarily halves)
- Each of these pieces is then divided in two, and so on.
- If each piece is split approximately in half, there are O($\log_2 N$) levels of splits. At each level, we make O(N) comparisons.
- So Quick Sort is an O($N \log_2 N$) algorithm.

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort insertion-sort 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)