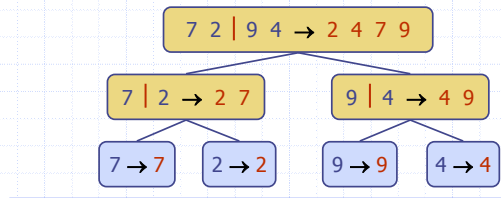


# Merge Sort



# 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  $S_1$  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 (§ 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

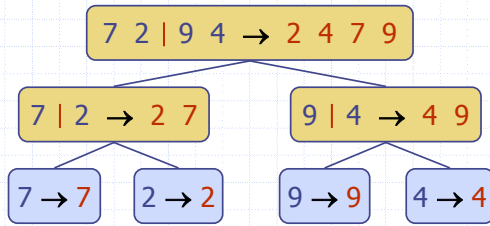
- ◆ The conquer step of merge-sort consists of merging two sorted sequences  $A$  and  $B$  into a sorted sequence  $S$  containing the union of the elements of  $A$  and  $B$
- ◆ Merging two sorted sequences, each with  $n/2$  elements and implemented by means of a doubly linked list, takes  $O(n)$  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  $\neg A.isEmpty() \wedge \neg B.isEmpty()$ 
    if  $A.first().element() < B.first().element()$ 
       $S.insertLast(A.remove(A.first()))$ 
    else
       $S.insertLast(B.remove(B.first()))$ 
  while  $\neg A.isEmpty()$ 
     $S.insertLast(A.remove(A.first()))$ 
  while  $\neg B.isEmpty()$ 
     $S.insertLast(B.remove(B.first()))$ 
  return  $S$ 
    
```

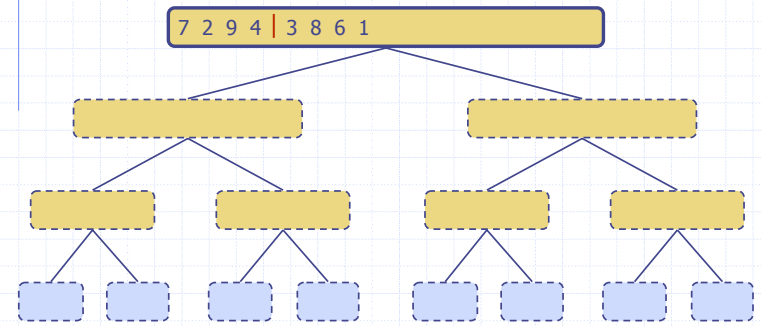
# Merge-Sort Tree

- ◆ 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



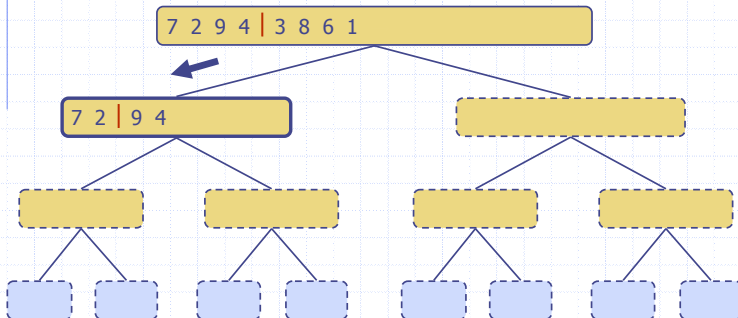
# Execution Example

## ◆ Partition



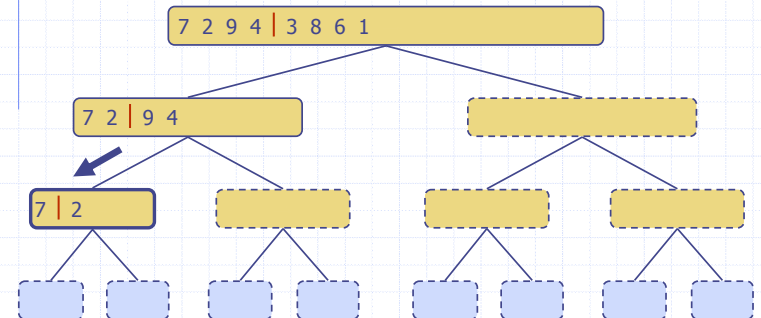
# Execution Example (cont.)

## ◆ Recursive call, partition



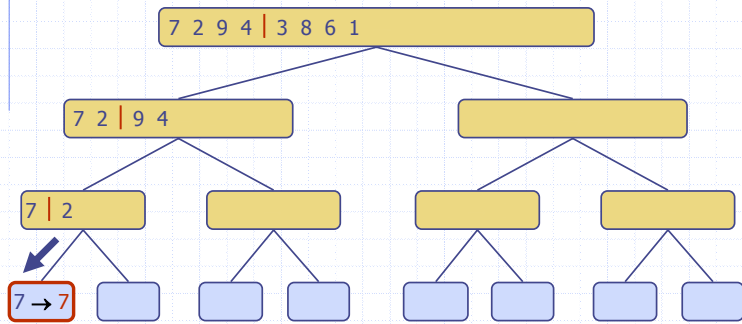
# Execution Example (cont.)

## ◆ Recursive call, partition



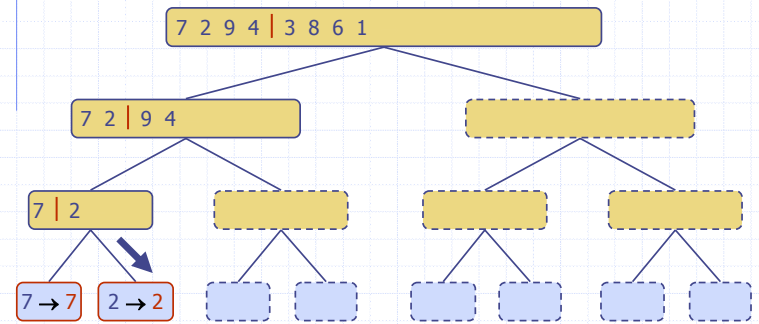
## Execution Example (cont.)

◆ Recursive call, base case



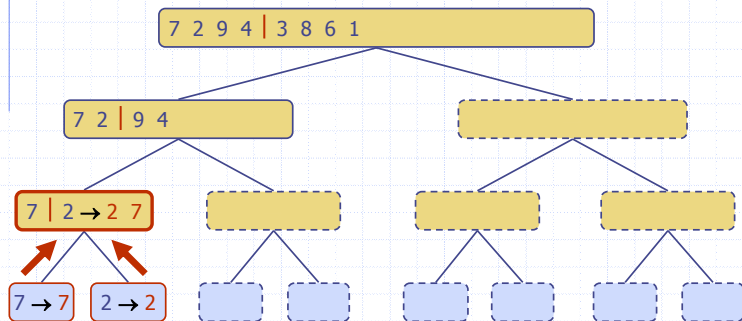
## Execution Example (cont.)

◆ Recursive call, base case



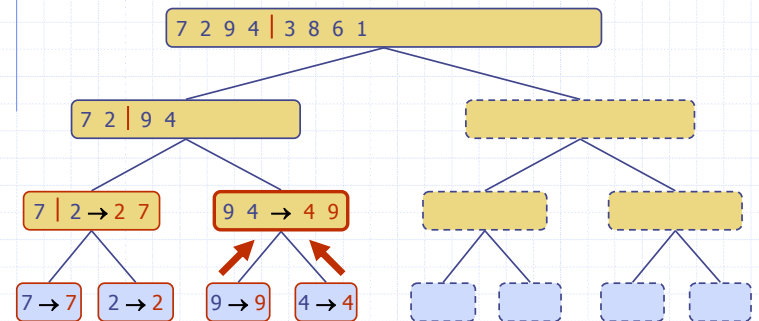
## Execution Example (cont.)

◆ Merge



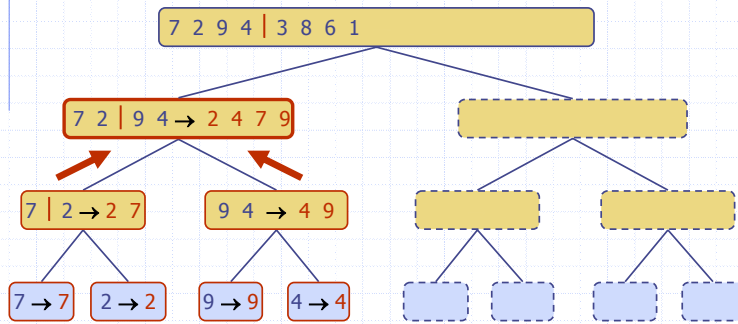
## Execution Example (cont.)

◆ Recursive call, ..., base case, merge



## Execution Example (cont.)

### ◆ Merge



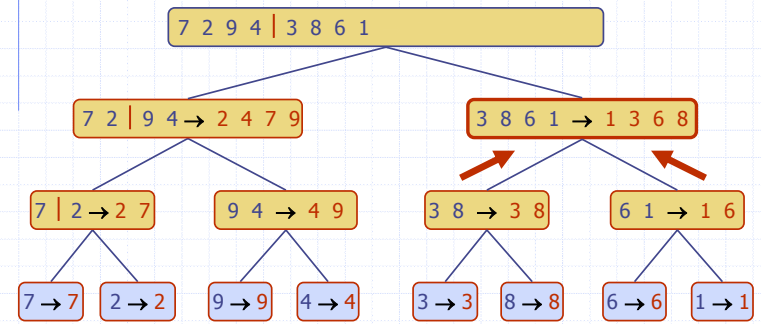
© 2004 Goodrich, Tamassia

Merge Sort

13

## Execution Example (cont.)

### ◆ Recursive call, ..., merge, merge



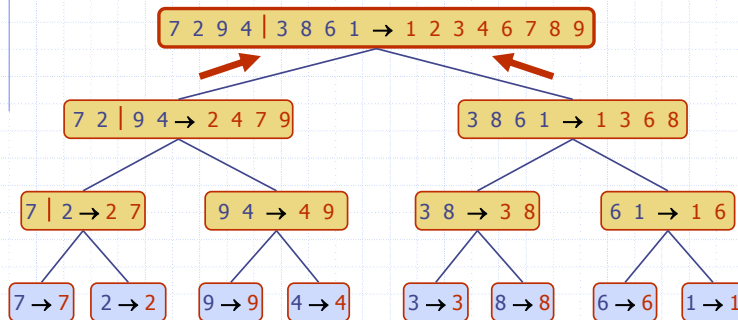
© 2004 Goodrich, Tamassia

Merge Sort

14

## Execution Example (cont.)

### ◆ Merge



© 2004 Goodrich, Tamassia

Merge Sort

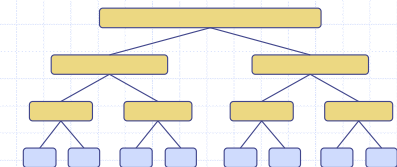
15

## 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	$n/2$
$i$	$2^i$	$n/2^i$
...	...	...



© 2004 Goodrich, Tamassia

Merge Sort

16

# Summary of Sorting Algorithms

Algorithm	Time	Notes
selection sort	$O(n^2)$	<ul style="list-style-type: none"> <li>slow</li> <li>in-place</li> <li>for small data sets (&lt; 1K)</li> </ul>
insertion sort	$O(n^2)$	<ul style="list-style-type: none"> <li>slow</li> <li>in-place</li> <li>for small data sets (&lt; 1K)</li> </ul>
heap sort	$O(n \log n)$	<ul style="list-style-type: none"> <li>fast</li> <li>in-place</li> <li>for large data sets (1K — 1M)</li> </ul>
merge- sort	$O(n \log n)$	<ul style="list-style-type: none"> <li>fast</li> <li>sequential data access</li> <li>for huge data sets (&gt; 1M)</li> </ul>

# Nonrecursive Merge-Sort

```

public static void mergeSort(Object[] orig, Comparator c) { // nonrecursive
    Object[] in = new Object[orig.length]; // make a new temporary array
    System.arraycopy(orig, 0, in, 0, in.length); // copy the input
    Object[] out = new Object[in.length]; // output array
    Object[] temp; // temp array reference used for swapping
    int n = in.length;
    for (int i=1; i < n; i*=2) { // each iteration sorts all length 2i runs
        for (int j=0; j < n; j+=2*i) // each iteration merges two length i pairs
            merge(in, out, c, j, i); // merge from in to out two length i runs at j
        temp = in; in = out; out = temp; // swap arrays for next iteration
    }
    // the "in" array contains the sorted array, so re-copy it
    System.arraycopy(in, 0, orig, 0, in.length);
}

protected static void merge(Object[] in, Object[] out, Comparator c, int start,
    int inc) { // merge in[start..start+inc-1] and in[start+inc..start+2*inc-1]
    int x = start; // index into run #1
    int end1 = Math.min(start+inc, in.length); // boundary for run #1
    int end2 = Math.min(start+2*inc, in.length); // boundary for run #2
    int y = start+inc; // index into run #2 (could be beyond array boundary)
    int z = start; // index into the out array
    while ((x < end1) && (y < end2))
        if (c.compare(in[x], in[y]) <= 0) out[z++] = in[x++];
        else out[z++] = in[y++];
    if (x < end1) // first run didn't finish
        System.arraycopy(in, x, out, z, end1 - x);
    else if (y < end2) // second run didn't finish
        System.arraycopy(in, y, out, z, end2 - y);
}
    
```

merge runs of length 2, then 4, then 8, and so on

merge two runs in the in array to the out array