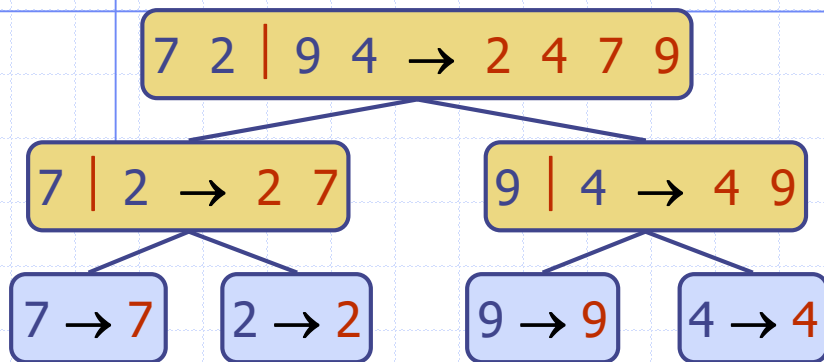


Lesson 5

Merge Sort: Collapsing Infinity To a Point



Wholeness of the Lesson

Merge Sort is a Divide and Conquer sorting algorithm which, by overcoming the limitations inherent in inversion-bound sorting algorithms, is able to sort lists in $O(n \log n)$ time, even in the worst case. The Divide and Conquer strategy is an example of the simple principle of "Do Less and Accomplish More." This technique makes it possible to break the inversion-bound barrier for sorting algorithms, to obtain very fast running times.

MergeSort: A Divide-and-Conquer Algorithm

◆ Recall: Divide-and conquer is a general algorithm design paradigm. MergeSort implements this design:

- **Divide**: divide the input data S in two disjoint subsets S_1 and S_2
- **Conquer**: solve the subproblems associated with S_1 and S_2
- **Combine**: combine the solutions for S_1 and S_2 into a solution for S

◆ Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

Merge-Sort

- ◆ Merge-sort on an input sequence S with n integers consists of three steps:
 - **Divide**: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
 - **Conquer**: recursively sort S_1 and S_2
 - **Combine**: merge S_1 and S_2 into a single sorted sequence

Algorithm *mergeSort*(S)

Input sequence S with n

Output sequence S sorted

if $S.size() > 1$ **then**

$(S_1, S_2) \leftarrow partition(S, n/2)$

mergeSort(S_1)

mergeSort(S_2)

$S \leftarrow merge(S_1, S_2)$

return S

Merging Two Sorted Sequences

- ◆ The *combine* 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 arrays, each with $n/2$ elements takes $O(n)$ time

Algorithm *merge*(A, B)

Input sorted sequences A and B with $n/2$ integers each

Output sorted sequence S of $A \cup B$

$S \leftarrow$ empty sequence

while $\neg A.isEmpty() \wedge \neg B.isEmpty()$ **do**

if $A.first() \leq B.first()$ **then**

$S.insertLast(A.remove(A.first()))$

else

$S.insertLast(B.remove(B.first()))$

while $\neg A.isEmpty()$ **do**

$S.insertLast(A.remove(A.first()))$

while $\neg B.isEmpty()$ **do**

$S.insertLast(B.remove(B.first()))$

return S

Implementation of Merge

```
public void merge(int[] tempStorage,
                 int lowerPointer,
                 int upperPointer,
                 int upperBound) {

    int j = 0; //tempStorage index
    int lowerBound = lowerPointer;
    //total number of elements to rearrange
    int n = upperBound - lowerBound + 1;

    //view the range [lowerBound,upperBound] as two arrays
    //[lowerBound, mid], [mid+1,upperBound] to be merged
    int mid = upperPointer - 1;
    while(lowerPointer <= mid && upperPointer <= upperBound){
        if(theArray[lowerPointer] <= theArray[upperPointer]){
            tempStorage[j++] = theArray[lowerPointer++];
        }
        else {
            tempStorage[j++] = theArray[upperPointer++];
        }
    }
}
```

Merge (continued)

```
//left array may still have elements
while(lowerPointer <= mid) {
    tempStorage[j++] = theArray[lowerPointer++];
}

//right array may still have elements
while(upperPointer <= upperBound){
    tempStorage[j++] = theArray[upperPointer++];
}

//replace the range [lowerBound,upperBound] in theArray with
//the range [0,n-1] just created in tempStorage
for(j=0; j<n; ++j) {
    theArray[lowerBound+j] = tempStorage[j];
}
}
```

Implementation of MergeSort, In-Place

```
int[] theArray;  
  
//public sorter  
public int[] sort(int[] input){  
    int n = input.length;  
    int[] tempStorage = new int[n];  
    tempStorage = input;  
    mergeSort(tempStorage, 0, n-1);  
    theArray = tempStorage;  
    return theArray;  
}
```

(continued)

```
void mergeSort(int[] temp, int lower, int upper) {  
    if(lower==upper){  
        return;  
    }  
    else {  
        int mid = (lower+upper)/2;  
        mergeSort(temp,lower,mid);  
        mergeSort(temp,mid+1, upper);  
        merge(temp,lower,mid+1,upper);  
    }  
}
```


Worst-case Analysis

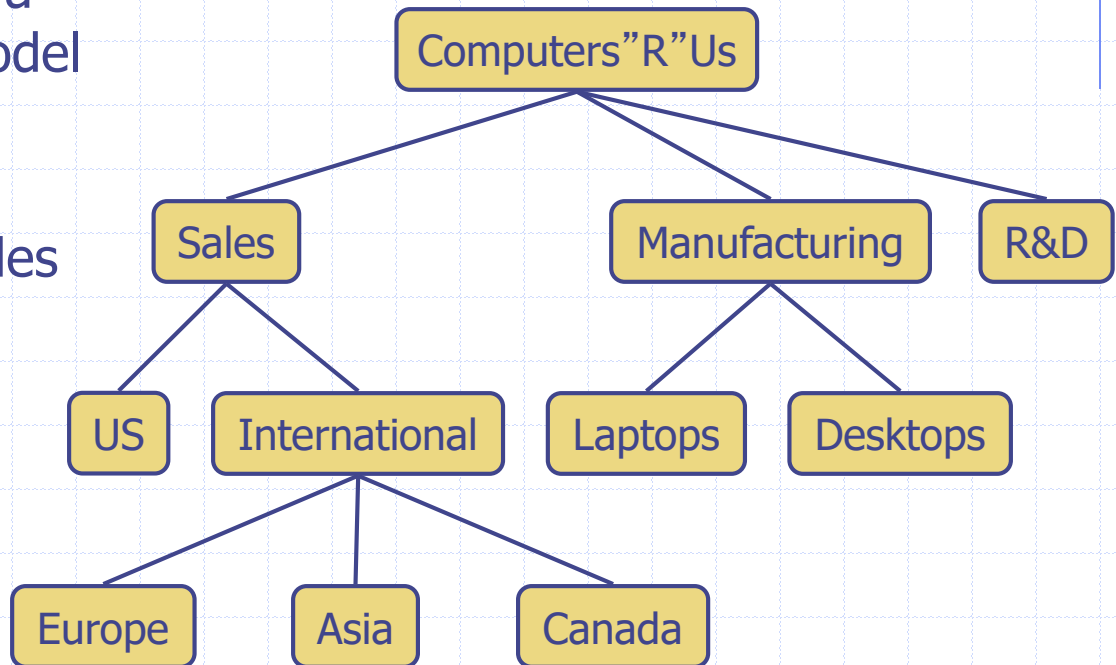
$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$

(or we could write $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn$)

1. Assuming n a power of 2, this becomes
$$T(n) = 2T(n/2) + O(n)$$
2. By the Guessing Method or Master Formula, we conclude
 $T(n)$ is $O(n \log n)$

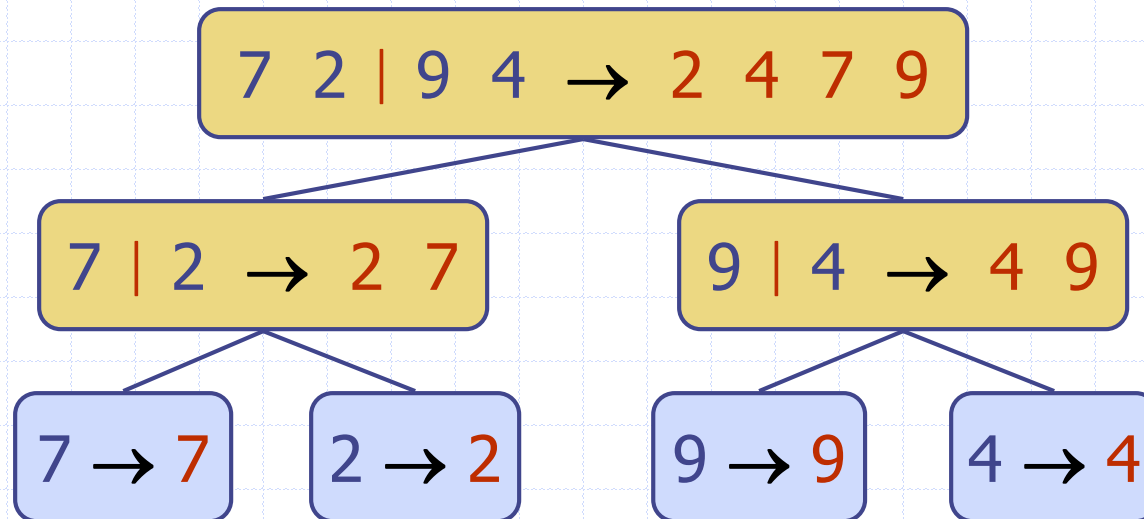
Trees

- ◆ In computer science, a tree is an abstract model of a hierarchical structure
- ◆ A tree consists of nodes with a parent-child relation
- ◆ Applications:
 - Organization charts
 - File systems
 - Programming environments



Merge-Sort Tree

- ◆ An execution of merge-sort may be 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

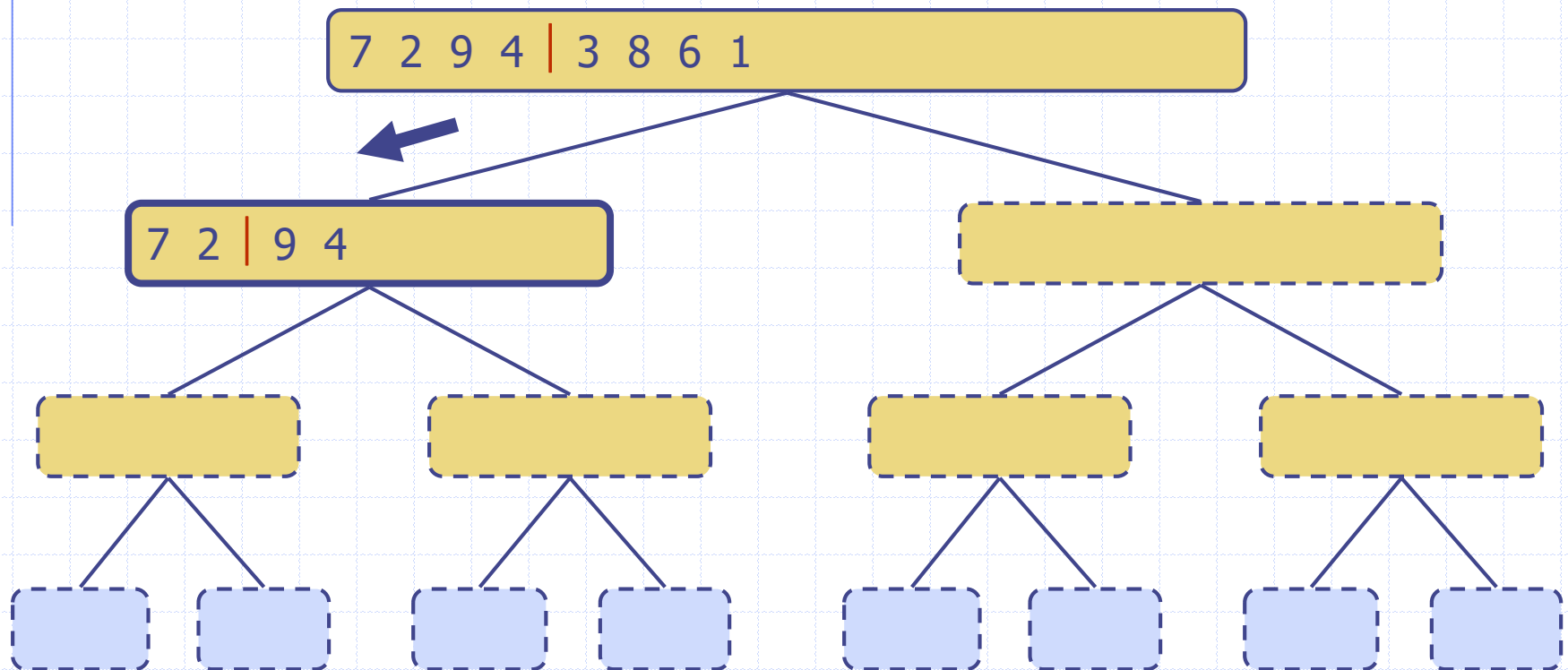


◆ Partition



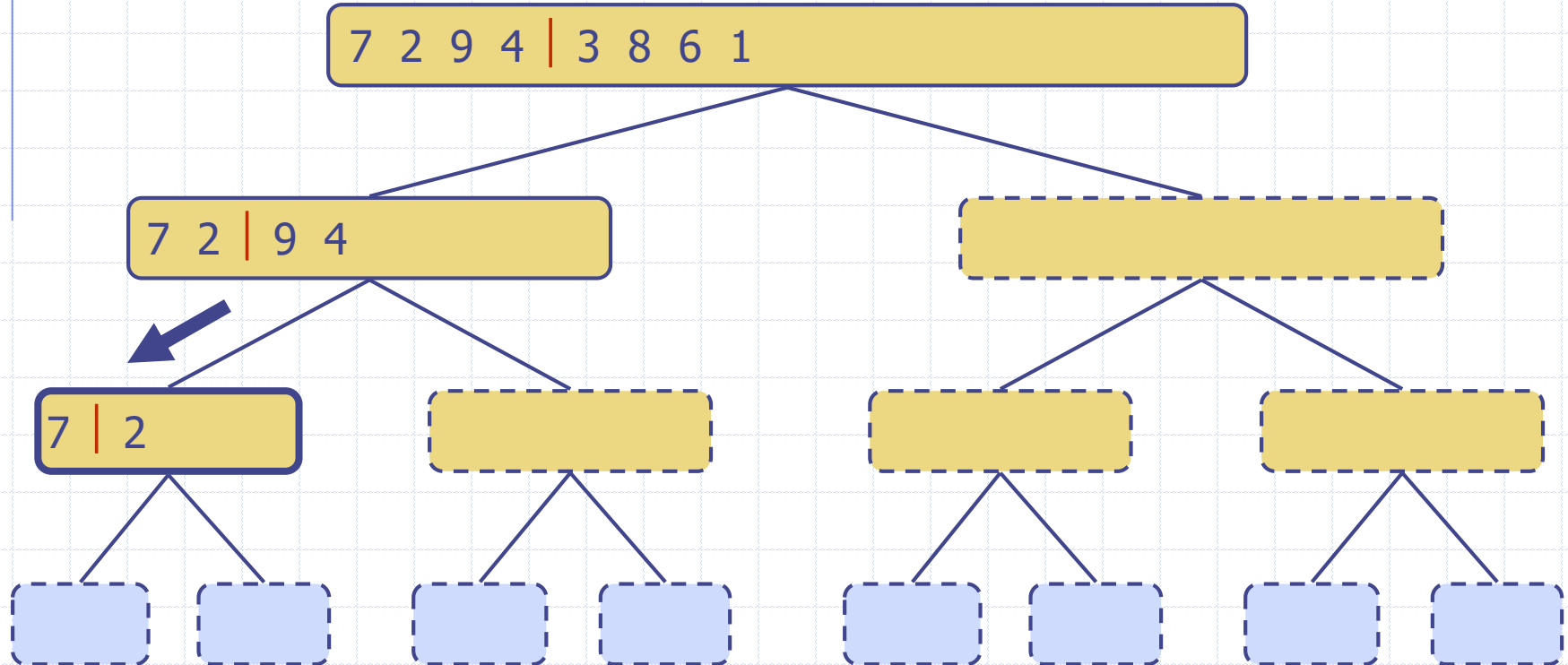
Execution Example (cont.)

◆ Recursive call, partition



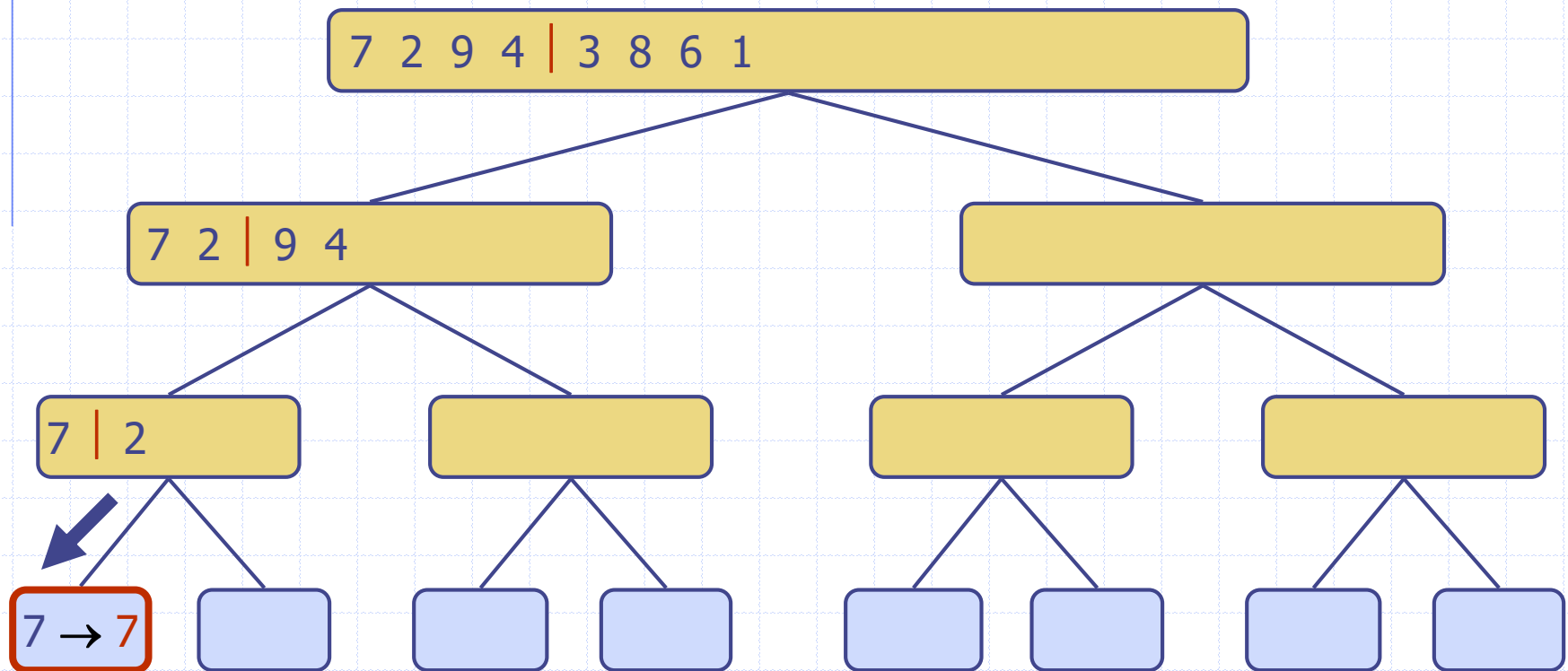
Execution Example (cont.)

◆ Recursive call, partition



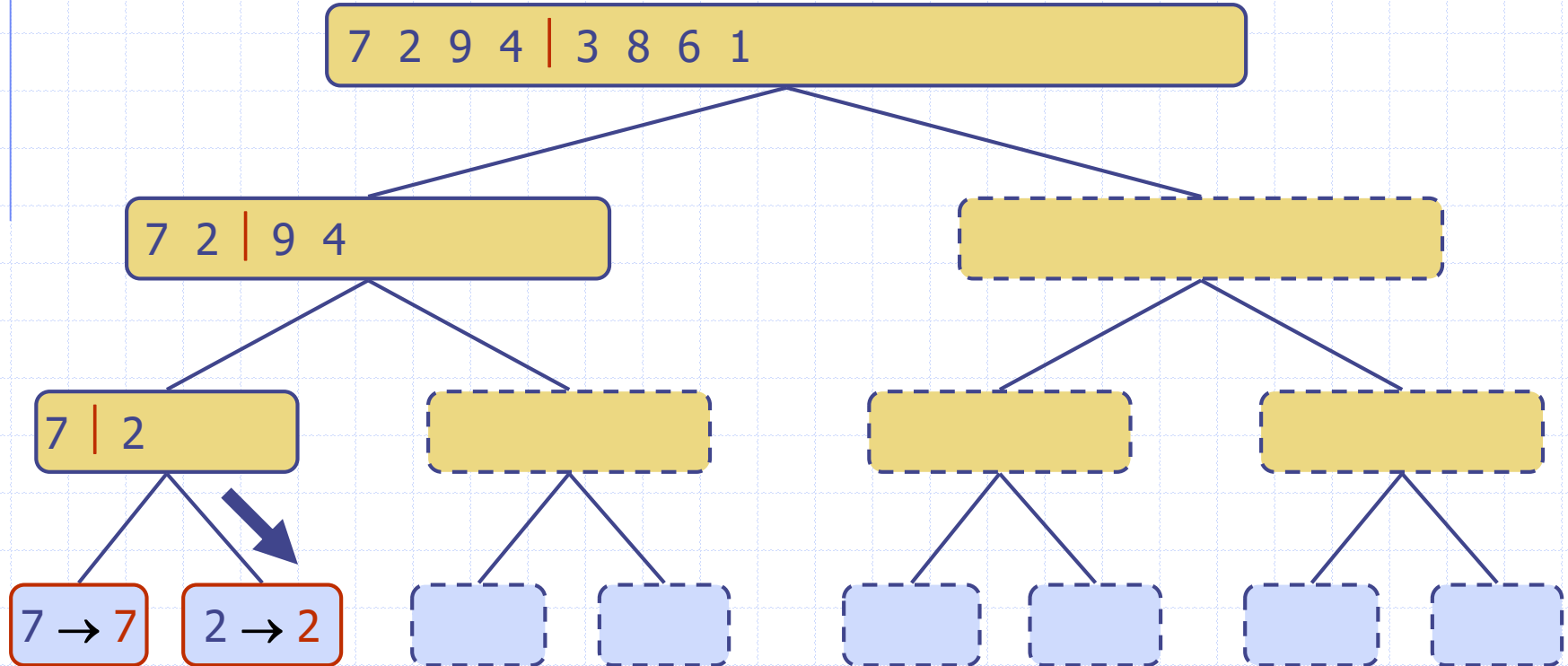
Execution Example (cont.)

◆ Recursive call, base case



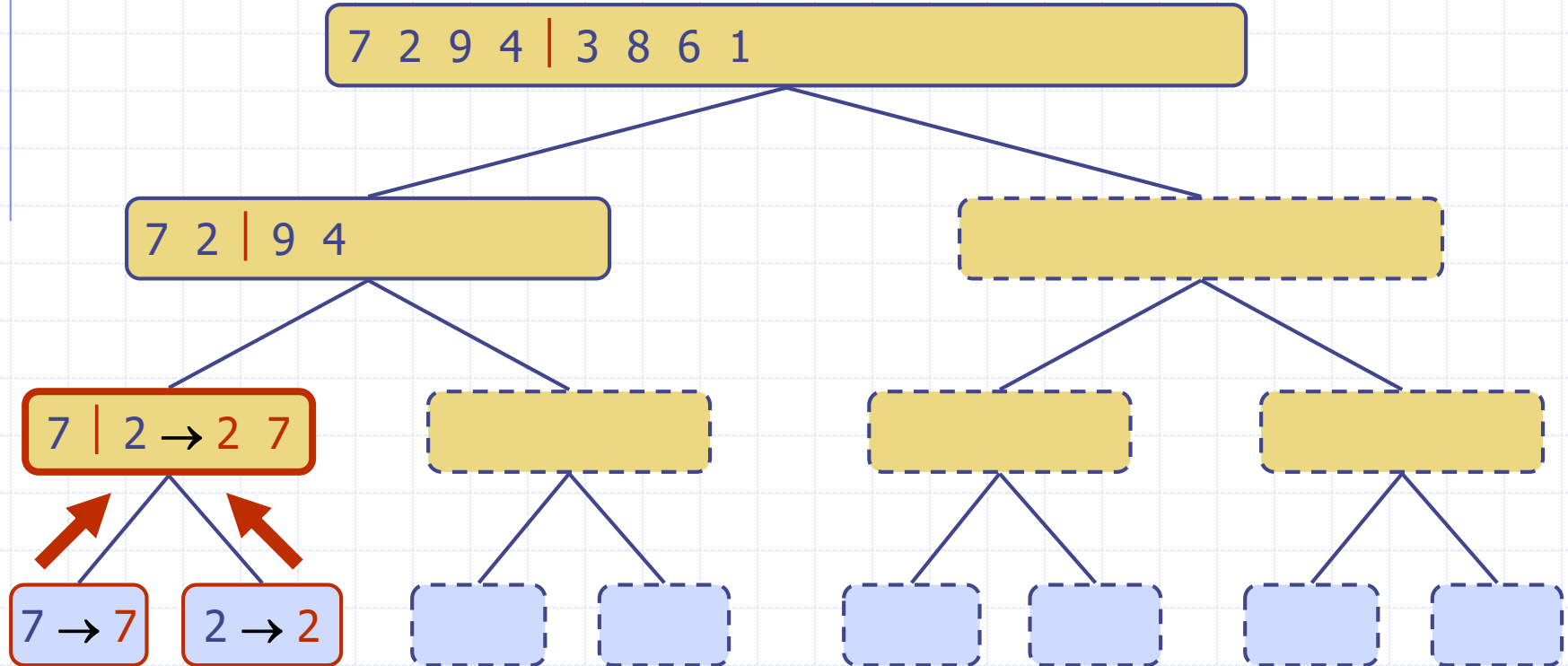
Execution Example (cont.)

◆ Recursive call, base case



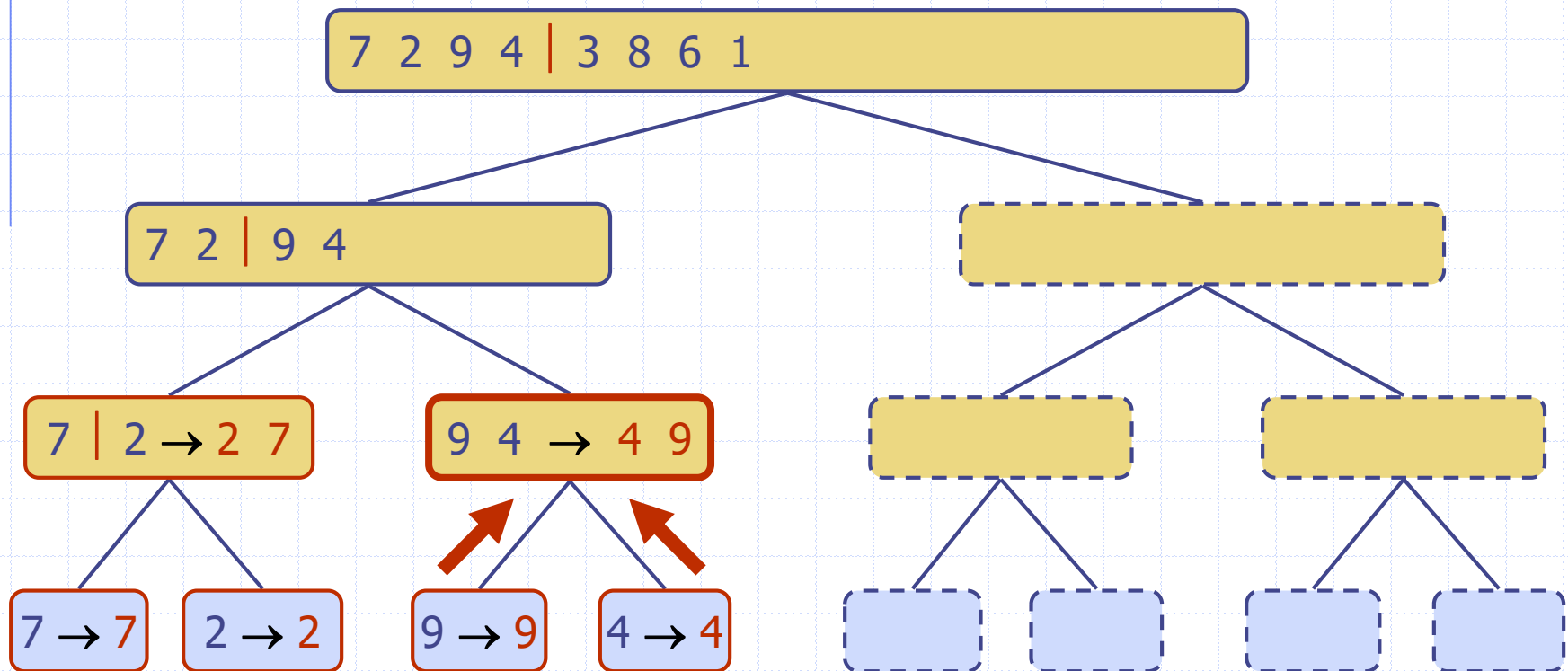
Execution Example (cont.)

◆ Merge



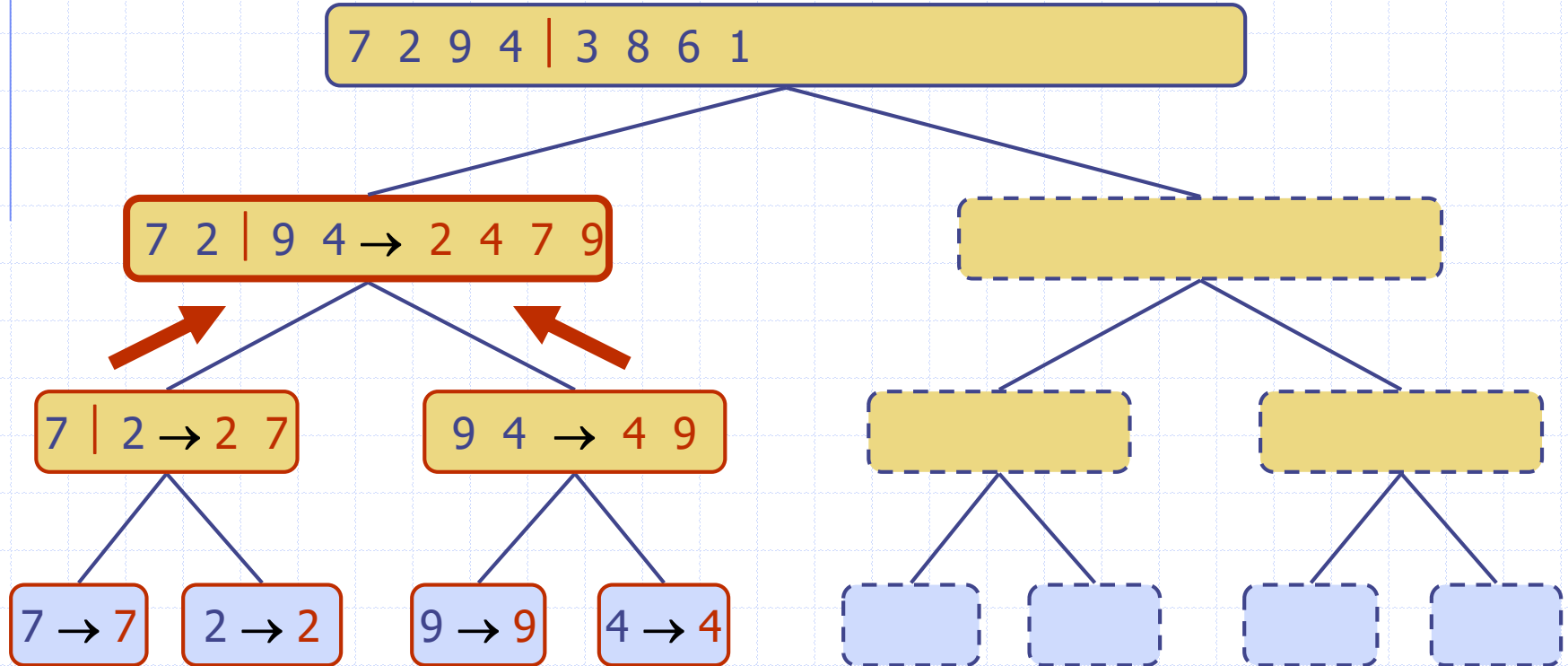
Execution Example (cont.)

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



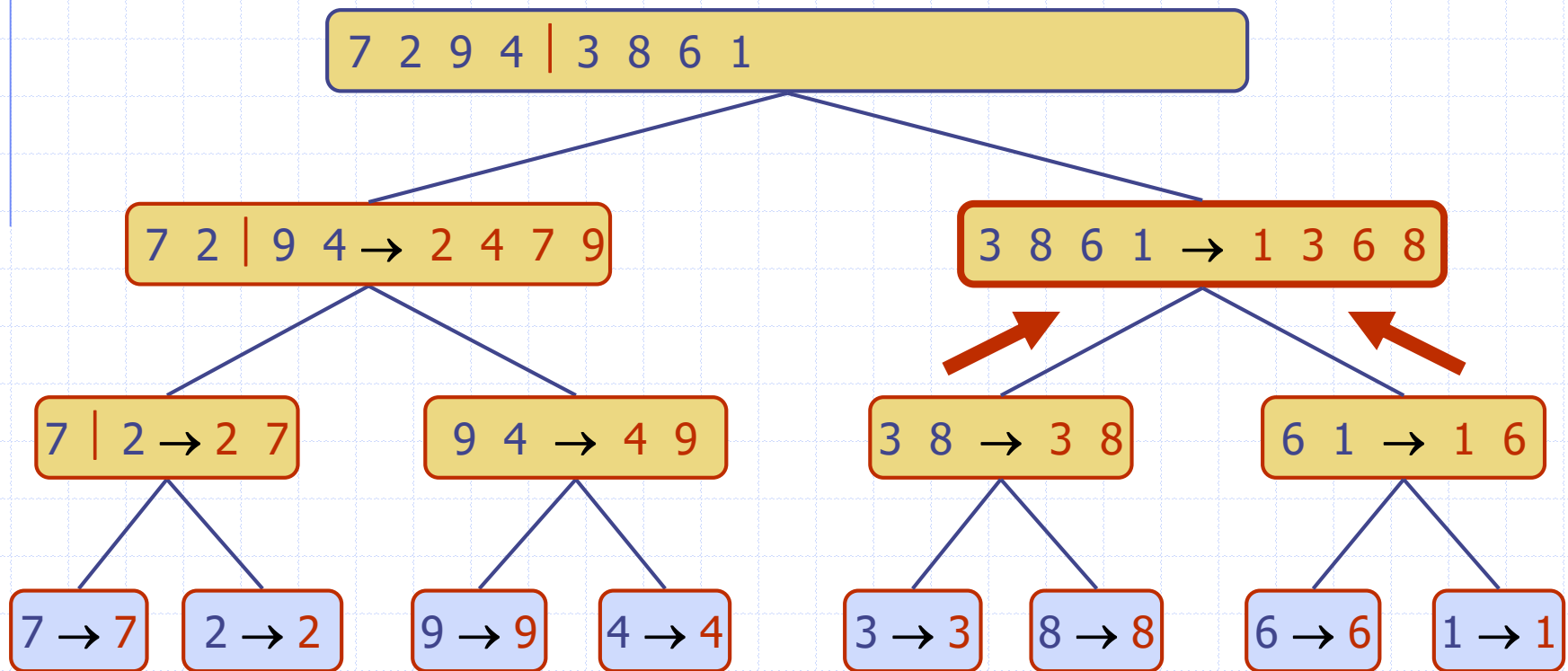
Execution Example (cont.)

◆ Merge



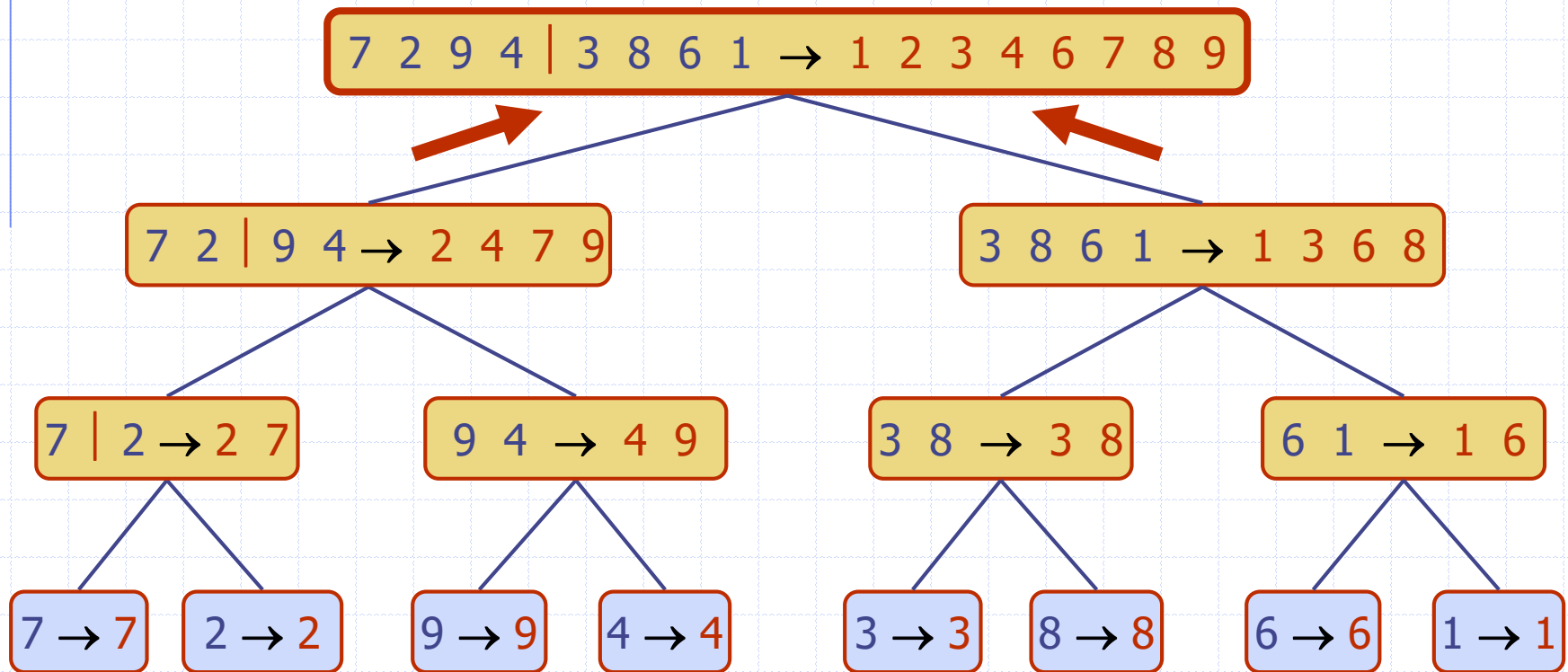
Execution Example (cont.)

◆ Recursive call, ..., merge, merge



Execution Example (cont.)

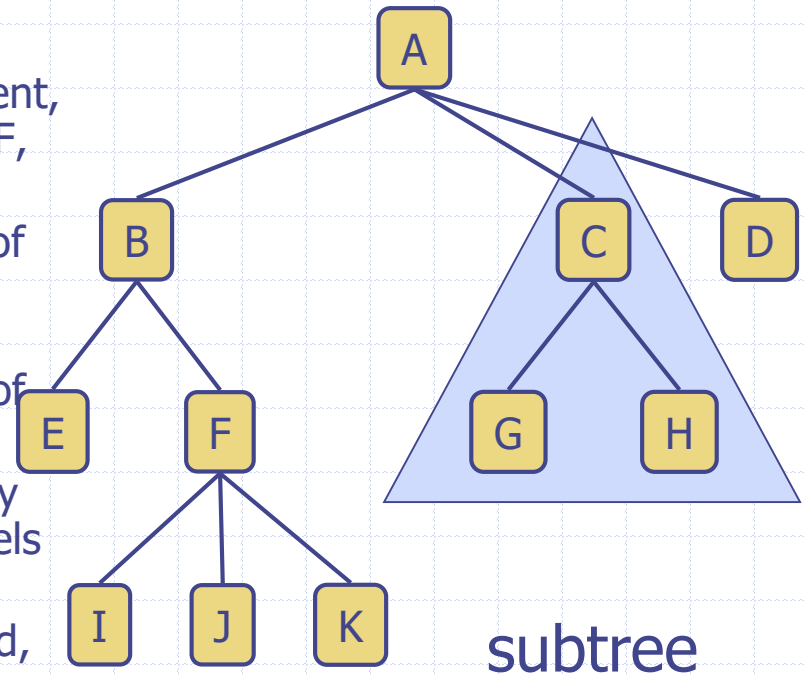
◆ Merge



Tree Terminology

- ◆ **Root:** node without parent (A)
- ◆ **Internal node:** node with at least one child (A, B, C, F)
- ◆ **Leaf** (or “external”) **node** is a node without children (E, I, J, K, G, H, D)
- ◆ **Ancestors of a node:** parent, grandparent, grand-grandparent, etc. (ancestors of K: F, B, A)
- ◆ **Depth of a node:** number of ancestors of the node (depth of K = 3)
- ◆ **Levels of a tree:** Level n of a tree is the set of all nodes having depth n . (Level 1 of this tree is {B, C, D})
- ◆ **Height of a tree:** maximum depth of any node (height of tree = 3). Note: Num levels = height + 1.
- ◆ **Descendant of a node:** child, grandchild, grand-grandchild, etc. (descendants of B are E, F, I, J, K)

- ◆ **Subtree:** tree consisting of a node and its descendants



Lengths of Descending Sequences (1)

Proposition. Suppose $b > 1$ is a real number and $n \geq b$ is an integer. Consider the descending sequence

$$n, \frac{n}{b}, \frac{n}{b^2}, \frac{n}{b^3}, \dots$$

The only positive integer k for which $\lfloor \frac{n}{b^k} \rfloor = 1$ is $\lfloor \log_b n \rfloor$ and the least positive integer m for which $\lfloor \frac{n}{b^m} \rfloor = 0$ is $1 + \lfloor \log_b n \rfloor$.

Proof. Let r be such that

$$(*) \quad b^r \leq n < b^{r+1}.$$

Clearly

$$\lfloor \frac{n}{b^r} \rfloor = 1 \quad \text{and} \quad \lfloor \frac{n}{b^{r+1}} \rfloor = 0.$$

Since $b^r \leq n$, it follows that $\frac{n}{b^{r-1}} \geq b$, and so

$$\lfloor \frac{n}{b^{r-1}} \rfloor \geq b > 1.$$

Therefore, r is the *only* exponent of b for which $\lfloor \frac{n}{b^r} \rfloor = 1$. The fact that $r = \lfloor \log_b n \rfloor$ follows from $r \leq \log_b n < r + 1$ (which follows from $(*)$).

We have shown that $r = \lfloor \log_b n \rfloor$ is the only k for which $\lfloor \frac{n}{b^k} \rfloor = 1$. It follows that $r + 1 = 1 + \lfloor \log_b n \rfloor$ is the least m for which $\lfloor \frac{n}{b^m} \rfloor = 0$.

Lengths of Descending Sequences (2)

Proposition. Suppose $b > 1$ is a real number and $n \geq b$ is an integer. Let S be the decreasing sequence

$$n, \frac{n}{b}, \frac{n}{b^2}, \frac{n}{b^3}, \dots, \frac{n}{b^s}$$

where s is the only exponent such that $\lfloor \frac{n}{b^s} \rfloor = 1$. Let T be the sequence

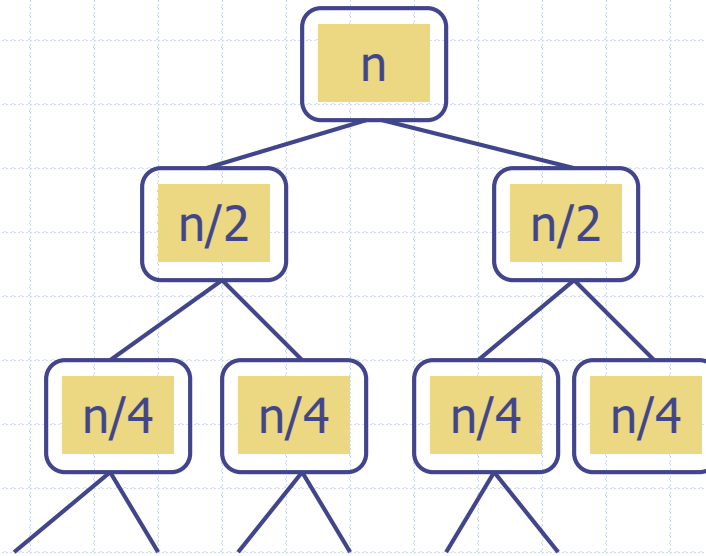
$$n, \frac{n}{b}, \frac{n}{b^2}, \frac{n}{b^3}, \dots, \frac{n}{b^t}$$

where t is the only exponent such that $\lfloor \frac{n}{b^t} \rfloor = 0$. Then

the length of S is $1 + \lfloor \log_b n \rfloor$ and length of T is $2 + \lfloor \log_b n \rfloor$.

Proof. By the previous Proposition, $s = \lfloor \log_b n \rfloor$ so the length of S is $1 + \lfloor \log_b n \rfloor$. Likewise, by the Proposition, $t = 1 + \lfloor \log_b n \rfloor$, so the length of T is $2 + \lfloor \log_b n \rfloor$.

Tree Exercise



Continue building the tree above until each node at the bottom level contains “1”, but no node at a previous level contains a “1” (integer division)

1. What is the height of the tree?
2. Asymptotically, what is the sum of all values contained in the nodes in the tree?

Alternate Analysis of Merge-Sort

- ◆ The height h of the merge-sort tree is $O(\log n)$
 - at each recursive call we divide the sequence in half (n a power of 2)
- ◆ The overall amount of work done at each level is $O(n)$
- ◆ Thus, the total running time of merge-sort is $O(n \log n)$

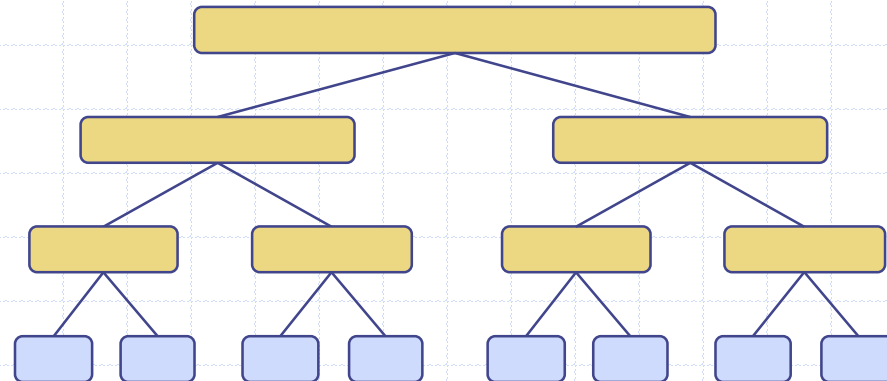
depth	#seqs	size
-------	-------	------

0	1	n
---	---	-----

1	2	$n/2$
---	---	-------

i	2^i	$n/2^i$
-----	-------	---------

...
-----	-----	-----



Comparison with Other Sorting Algorithms

- ◆ Demo confirms that MergeSort's $O(n \log n)$ estimated running time is truly much faster than those of the inversion-bound algorithms and LibrarySort
- ◆ Can see why MergeSort is not inversion bound by example: [4, 3, 2, 1]:
 - #inversions = 6
 - #comparisons = 4

Main Point

By using a Divide and Conquer strategy, MergeSort overcomes the limitations that prevent inversion-bound sorting algorithms from performing faster than n^2 . An essential characteristic of this strategy is the relationship of whole to part – wholes are successively collapsed and the collapsed values are combined to produce a new whole. This is different from the incremental approach of inversion-bound algorithms. We see here an application of the MVS principle of *akshara*: Creation arises in the collapse of the unbounded value of wholeness to a point.

Handling Duplicates

- ◆ Issue arises during the merge step – if element in left half equals element in right half, insert element in left half first

		d		
--	--	---	--	--

		d		
--	--	---	--	--

- Stability

Name	Date Received
...	...
Dave	11/5/2003
Dave	12/1/2004
Dave	1/8/2005
Dave	4/2/2006
...	...

If you first sort by date (name secondary), then later by name (date secondary), you want dates related to a single name to remain sorted.

Handling Duplicates (cont)

◆ Definition. Suppose

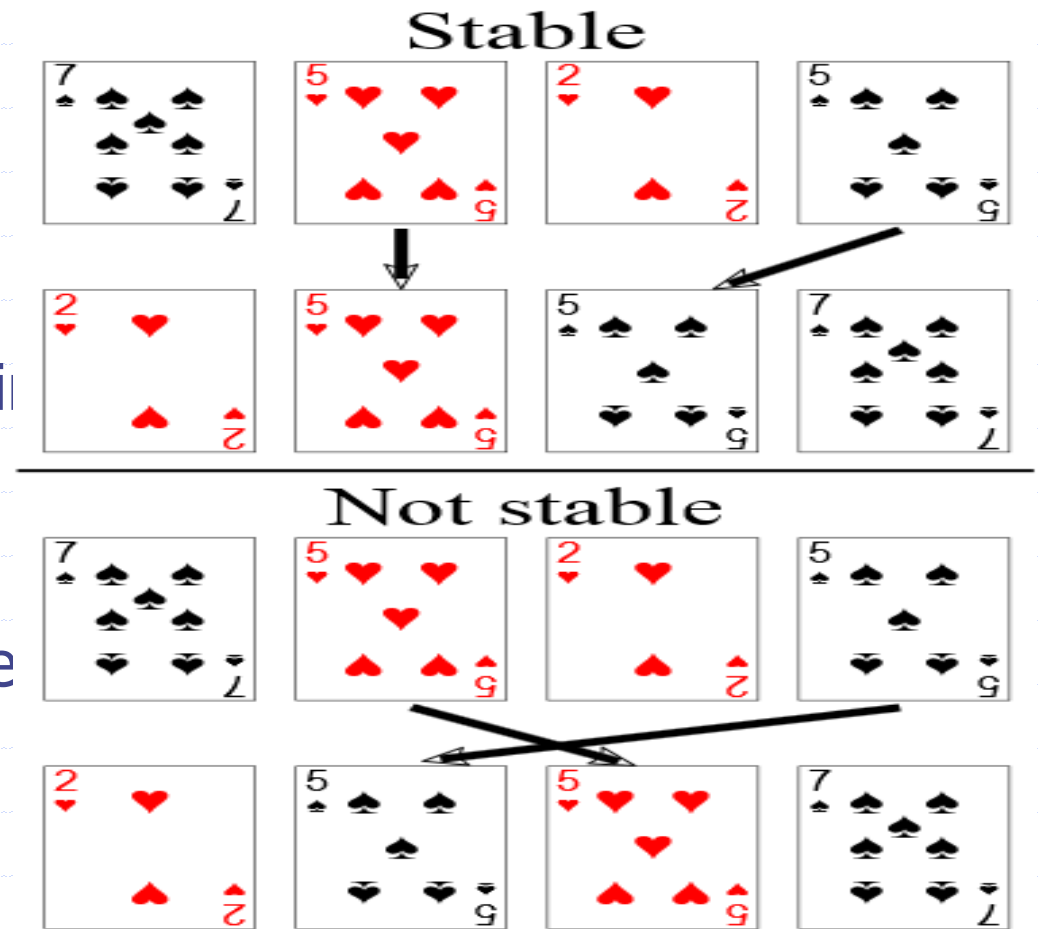
$$S = \langle (k_0, e_0), (k_1, e_1), \dots, (k_n, e_n) \rangle$$

is a list of pairs with keys k_0, k_1, \dots, k_n . A sorting algorithm is *stable* if, whenever it is the case that (k_i, e_i) precedes (k_j, e_j) before sorting (so that $i < j$) and $k_i = k_j$, then it continues to be true after sorting by keys that the pair (k_i, e_i) precedes (k_j, e_j)

*Stable sorting does not change
the order of duplicates*

Stability of Sorting Algorithms

- Sort By rank
- Stable sort -
2 5s Must Remain
in the same order in
the original list
- Stability is not an
Issue if all keys are
different



Stability of Sorting Algorithms

- ◆ MergeSort is stable because of our strategy for handling duplicates during Merge
- ◆ Are InsertionSort, BubbleSort, SelectionSort stable?

Main Point

Stability of a sorting algorithm requires maintenance of nonchange in the midst of change. This is an example in the world of sorting routines of the inner dynamics of outward success, as described in SCI: The more the inner quality of awareness remains established in silence, the more outer dynamism is supported for success and fulfillment.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Merge Sort

1. Inversion-bound sorting algorithms typically examine each successive element in the input array and perform a further step to place this element in an already sorted area. The style of sorting involves a *sequential unfolding*.
2. MergeSort proceeds by repeatedly collapsing the wholeness of the current input array into parts and then synthesizing the parts into a sorted whole. This approach yields a much faster sorting algorithm.
3. *Transcendental Consciousness* is the field of *infinite correlation*, where “an impulse anywhere is an impulse everywhere,” a field of “frictionless flow”.
4. *Impulses within the Transcendental field*. Established in the transcendental field, action reaches fulfillment with minimum effort. Yoga is “skill in action” – efficiency in action, “doing less, accomplishing more”, whereby little needs to be done to accomplish great goals.
5. *Wholeness moving within itself*. In Unity Consciousness, the field of action effortlessly unfolds as the play of one’s own Self, one’s own pure consciousness.