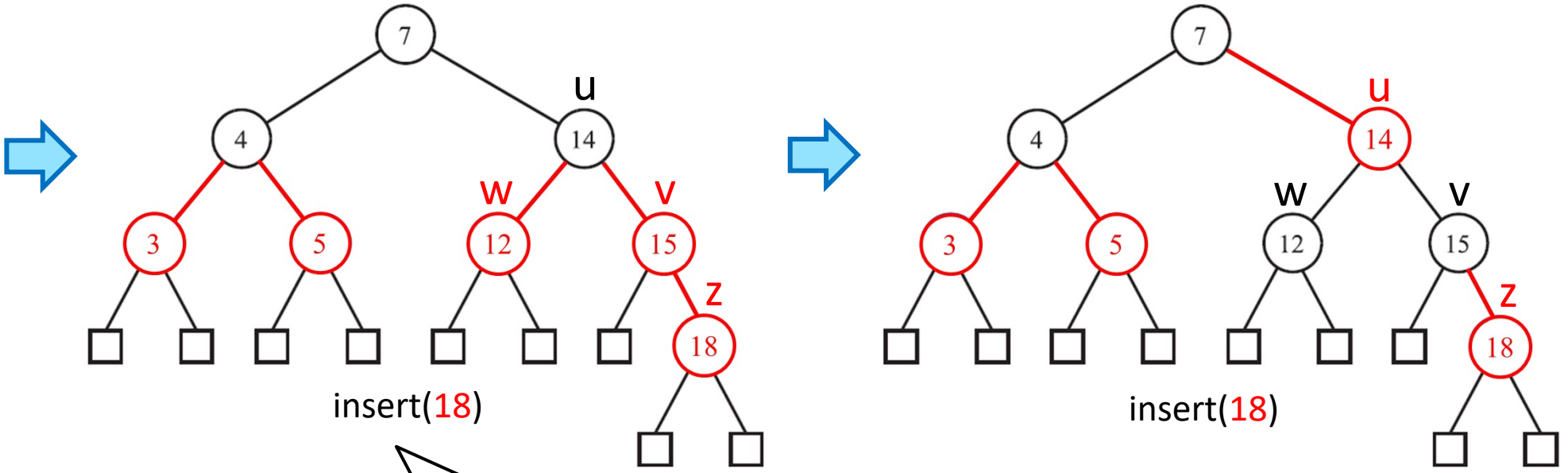
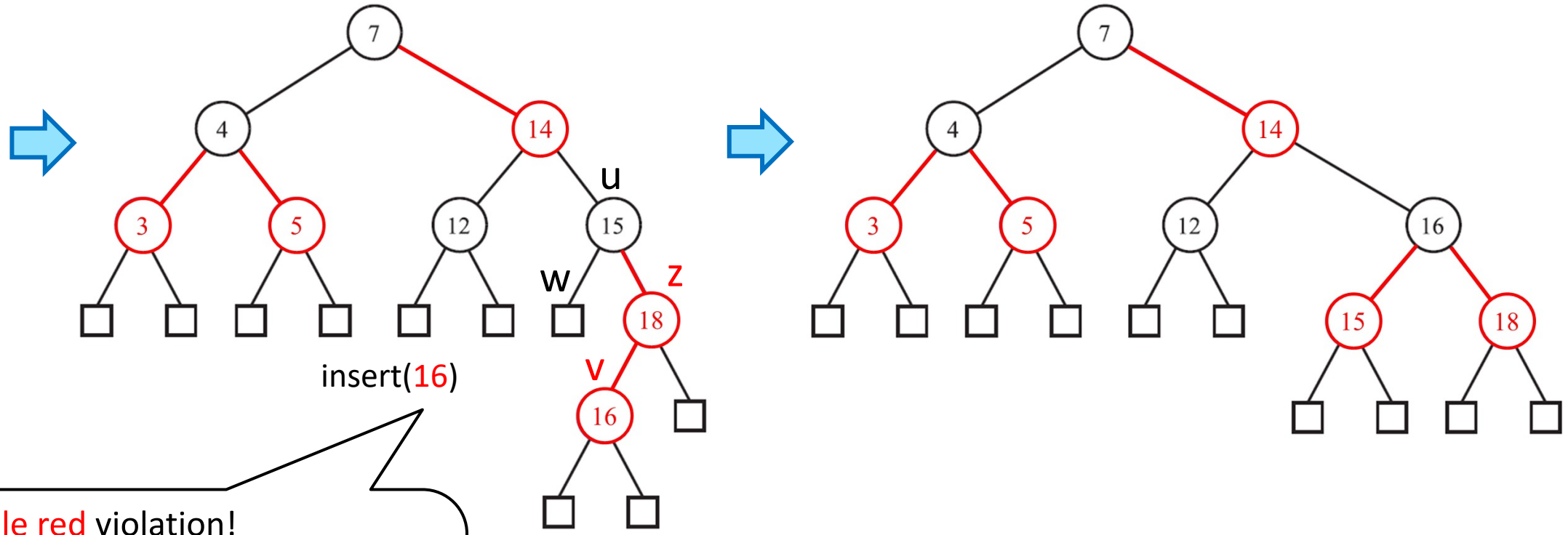
RED-BLACK – INSERTION EXAMPLES



Double red violation!

- Case 2: w is red (do recoloring)
  - Make v and w **black**
  - Make u red (unless it's the root)

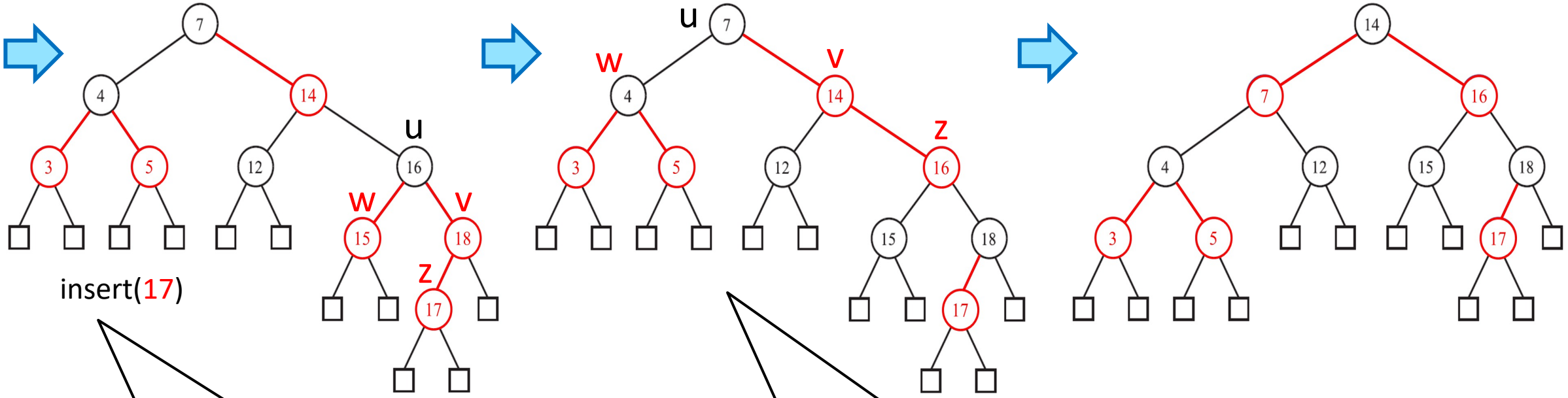
RED-BLACK – INSERTION EXAMPLES



Double red violation!

- Case 1: w is **black**
  - Rename (u,v,z) as (a,b,c) and use trinode restructuring
  - Make a and c red and b **black**

RED-BLACK – INSERTION EXAMPLES



Double red violation!

- Case 2: w is red (do recoloring)
  - Make v and w **black**
  - Make u red (unless it's the root)

This results in another double red violation higher up the tree!

- Case 1: w is **black**
  - Rename (u,v,z) as (a,b,c) and use trinode restructuring
  - Make a and c red and b **black**

RED-BLACK – REMOVAL

- We will skip the details of how an entry is removed from a **Red**-**Black** tree, since it involves dealing with many special cases that can be confusing sometimes
- All you need to know is that removal may require O(log n) recoloring operations and two trinode restructuring operations
- Note that **recoloring** is *simpler and slightly faster* than trinode restructuring, which is why we would rather perform the former rather than the latter

COMPARISON OF DIFFERENT TREES

We discussed three type of **self-balancing search trees**:

- AVL trees:
  - **Removing** an entry may take O(log n) **trinode restructuring** operations
- (2,4) trees:
  - **Inserting** an entry may take O(log n) **split** operations
  - **Removing** an entry may take O(log n) **fusions** and/or 1 **transfer** operation
- Red-Black trees:
  - **Insertion** may take O(log n) **recoloring** operations and/or 1 **trinode restructuring**
  - **Removal** may take O(log n) **recoloring** operations and/or 2 **trinode restructuring**

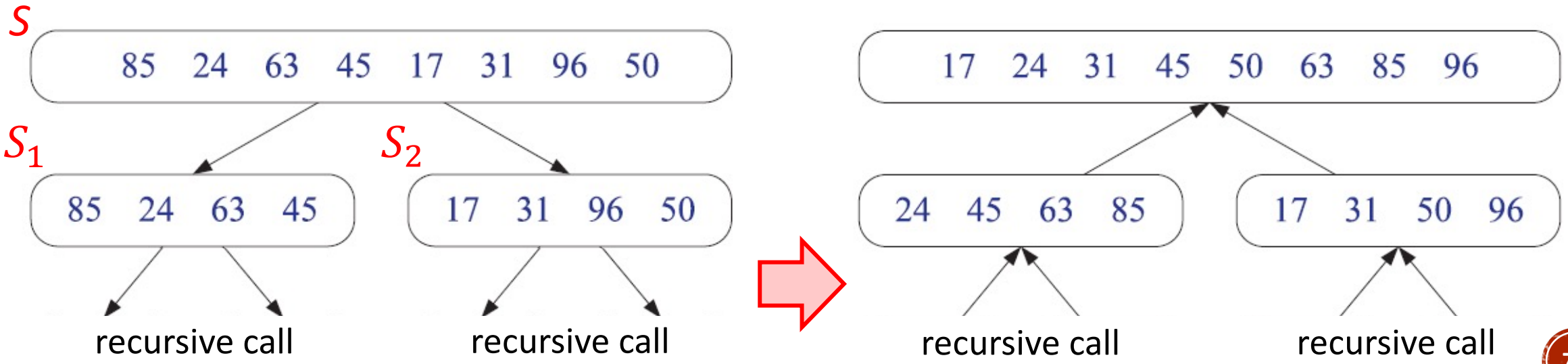
Recoloring is the fastest operation, *hence Red-Black trees tend to be faster*

CHAPTER 11:
SORTING

MERGE-SORT

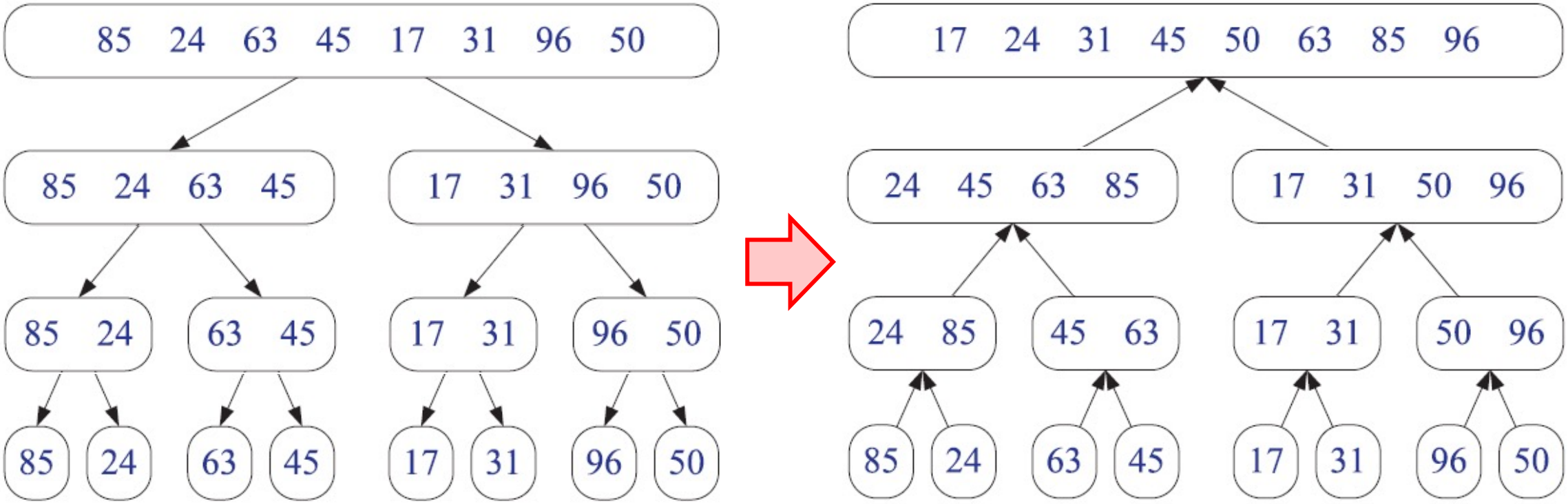
MERGE-SORT

- Given a sequence S of n elements, the algorithm "merge-sort" sorts S as follows:
  1. Divide: If S contains less than two element, return S immediately; otherwise **divide** S by putting the first ⌈n/2⌉ elements in S_1 and the remaining ⌊n/2⌋ in S_2
  2. Recur: Recursively **sort** S_1 and S_2
  3. Conquer: Put back the elements into S by **merging** the sorted sequences S_1 and S_2 into a sorted sequence.



MERGE-SORT TREE

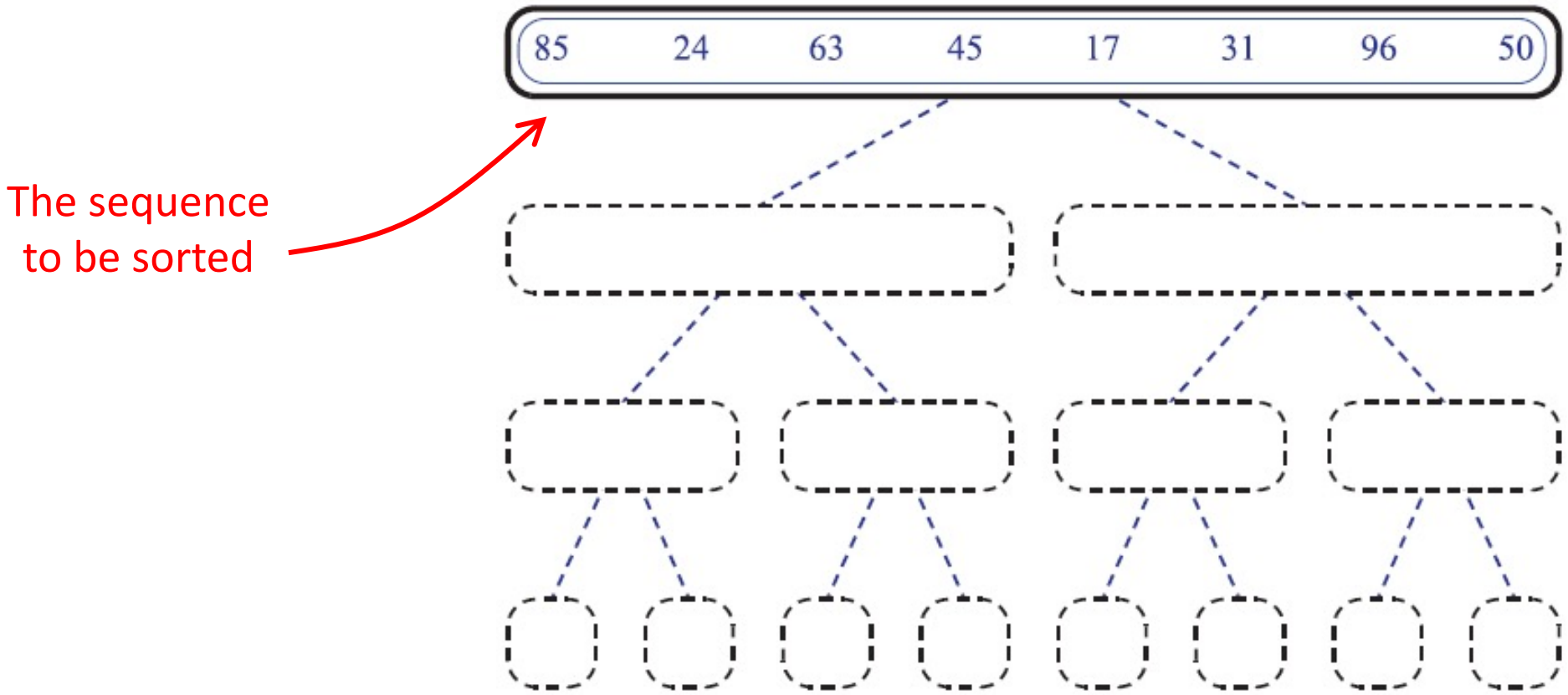
- Here is an example of a merge-sort tree



- Since the input size roughly halves at each recursive call, the height of the merge-sort tree is O(log n). In fact, it is proven that the height equals $\lceil \log n \rceil$

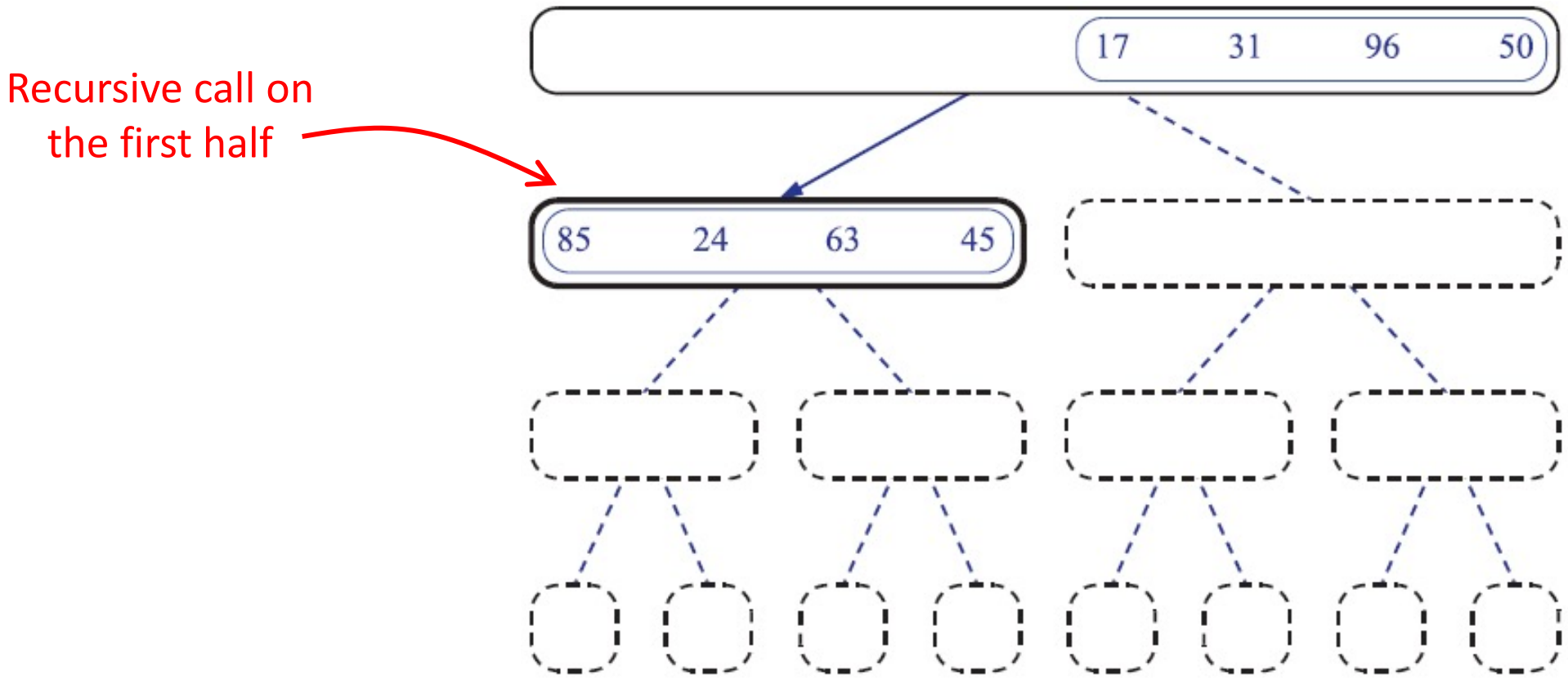
MERGE-SORT TREE

- Here is a **more detailed example** showing that the first half is sorted recursively before starting with the second half



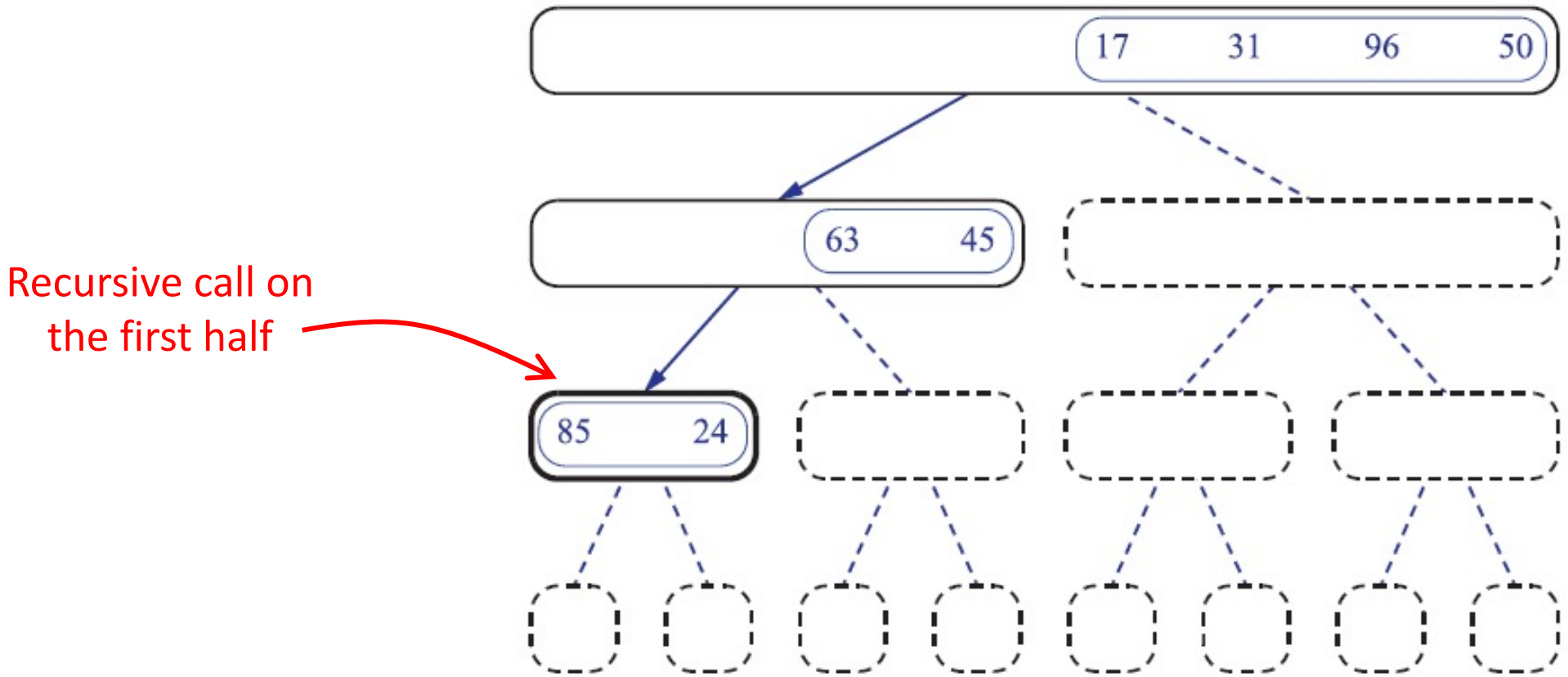
MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half



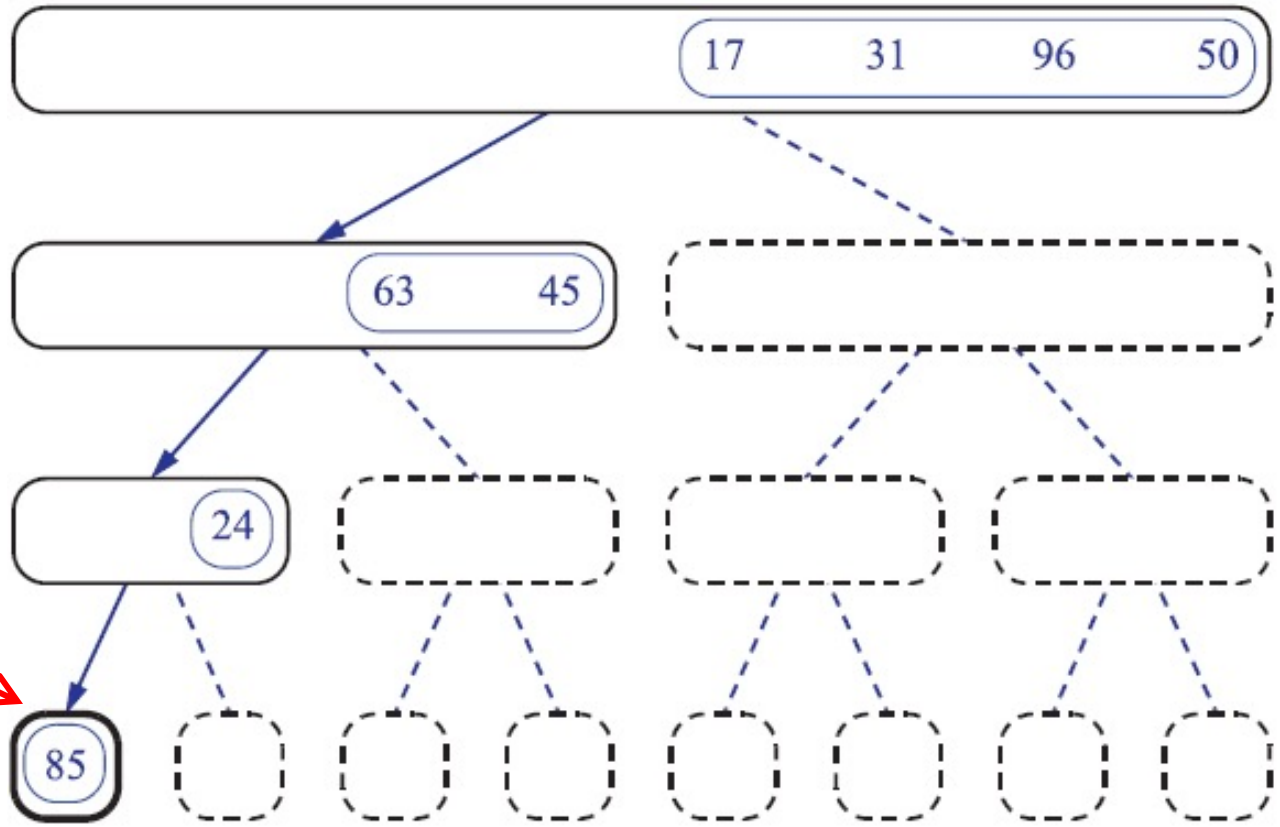
MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half



MERGE-SORT TREE

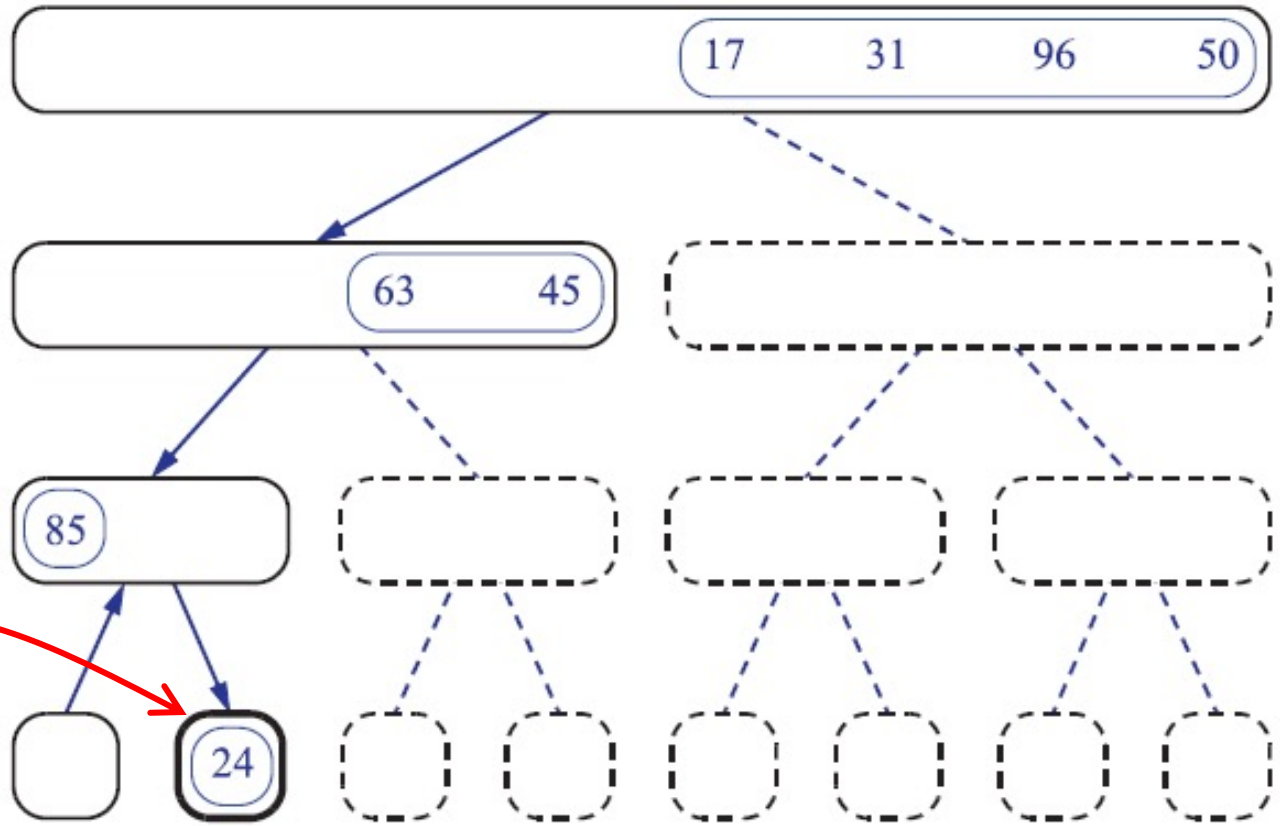
- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half



First half is less than 2 elements; no need for recursion

MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half

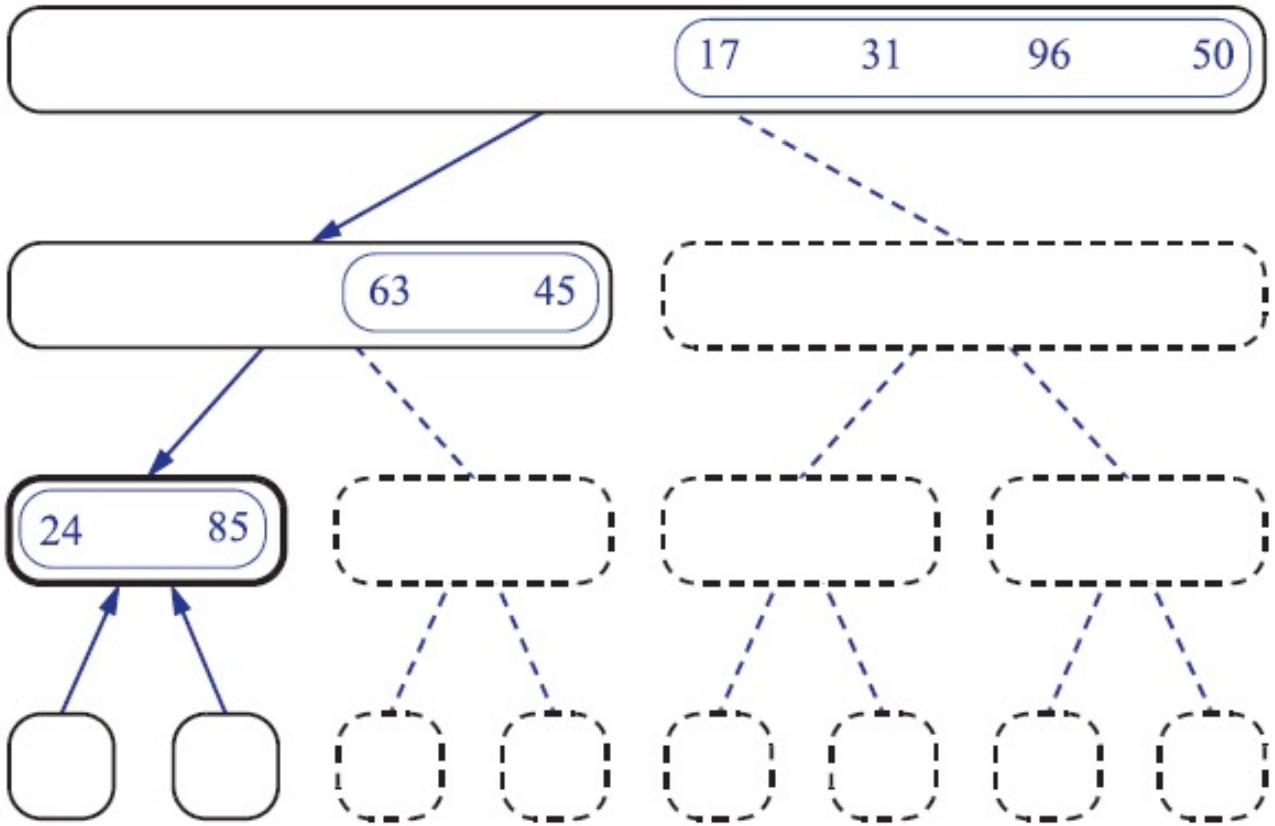


Second half is less than 2 elements; no need for recursion

MERGE-SORT TREE

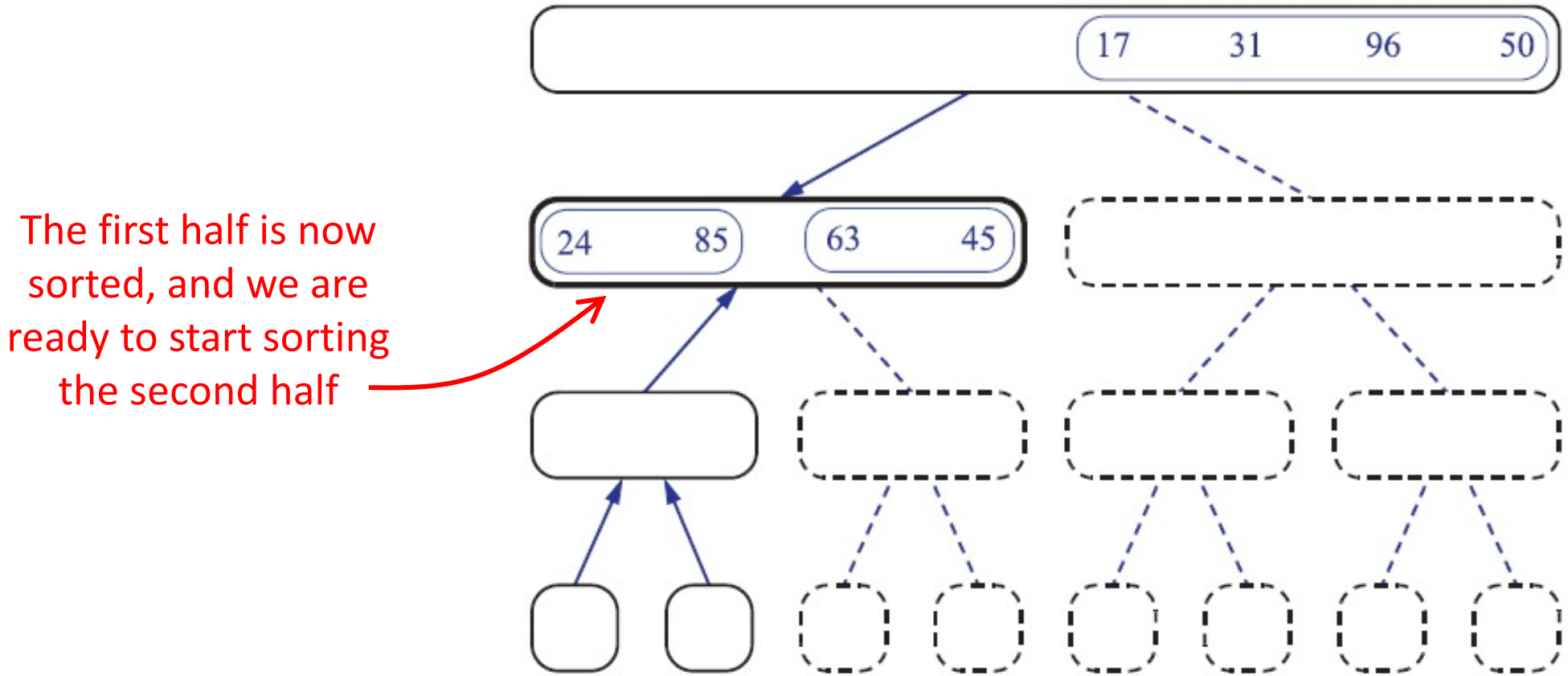
- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half

Merging the two halves (85) and (24) involves **sorting** them, which is why we now have (24) before (85)



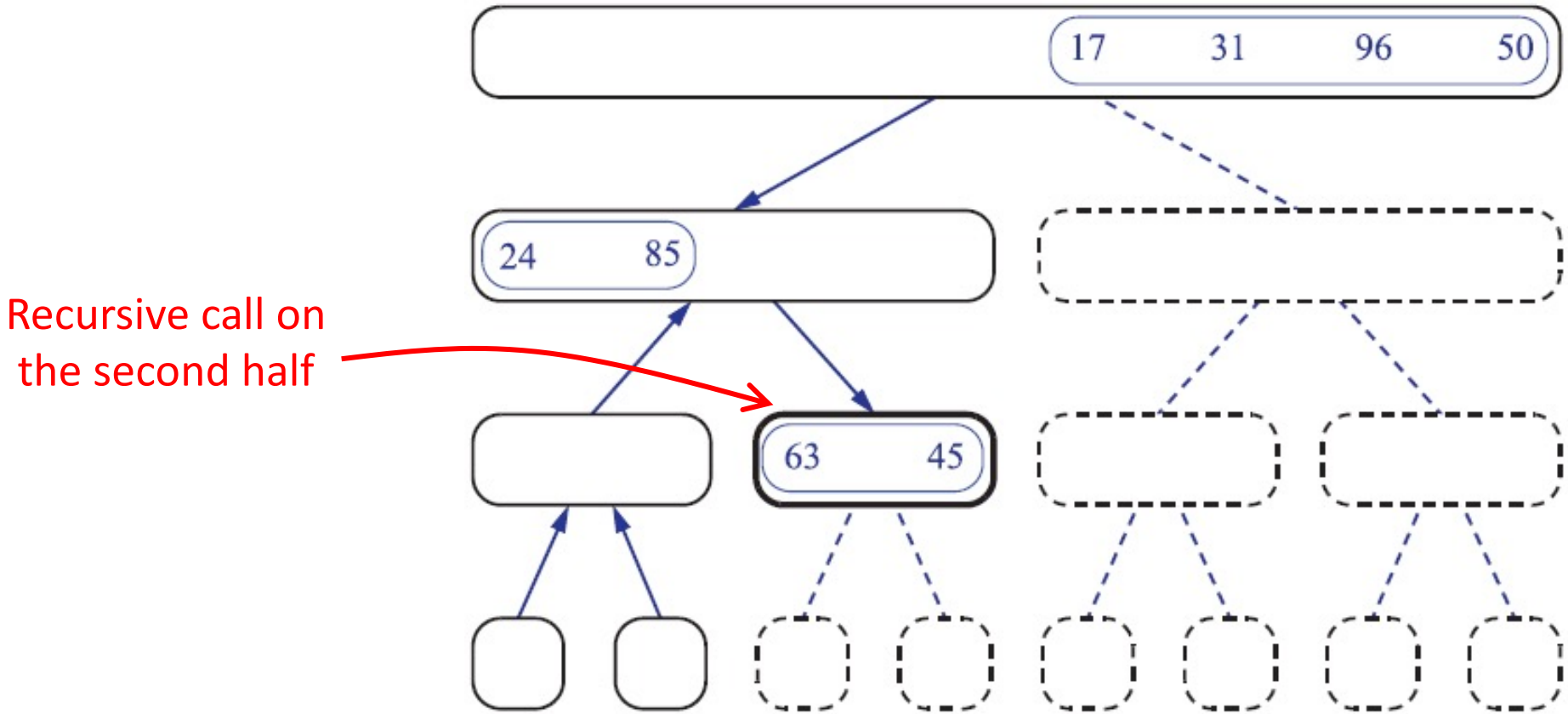
MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half



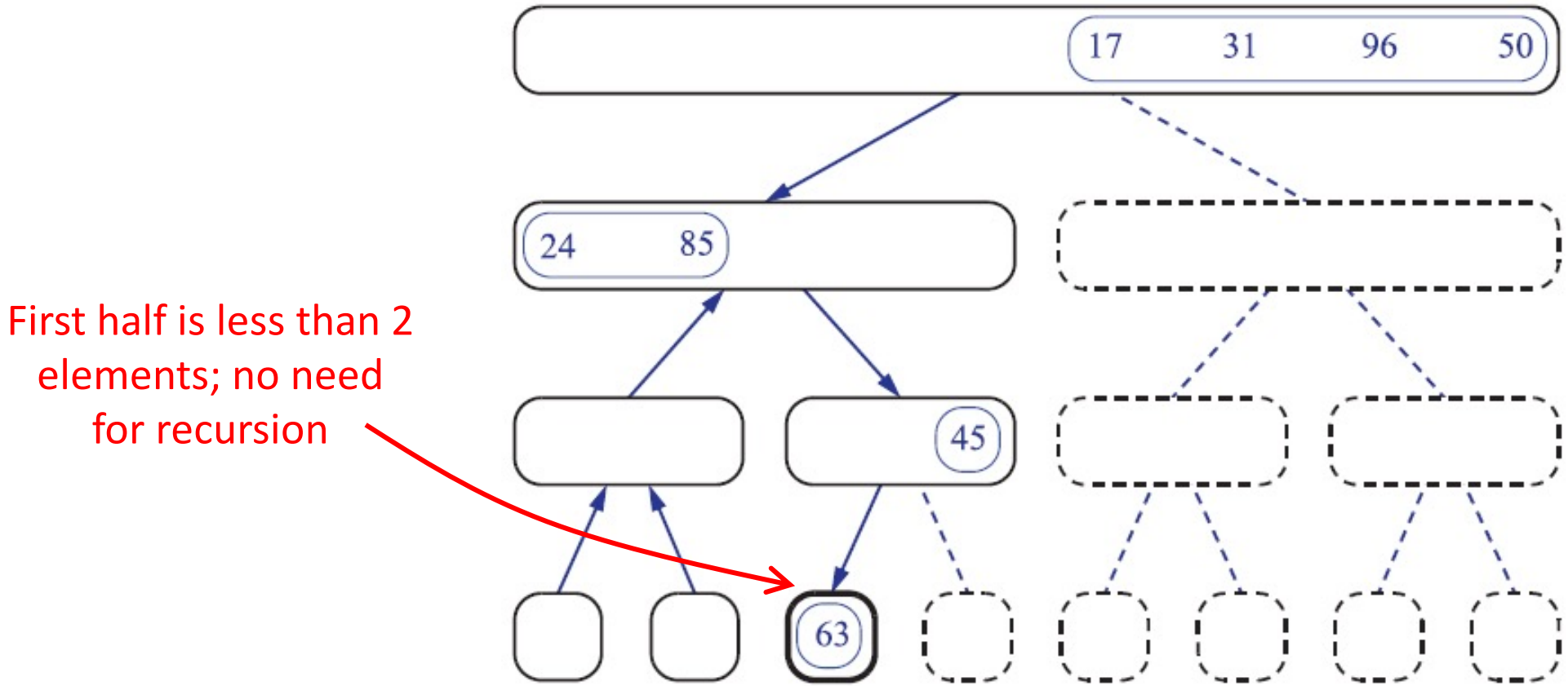
MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half



MERGE-SORT TREE

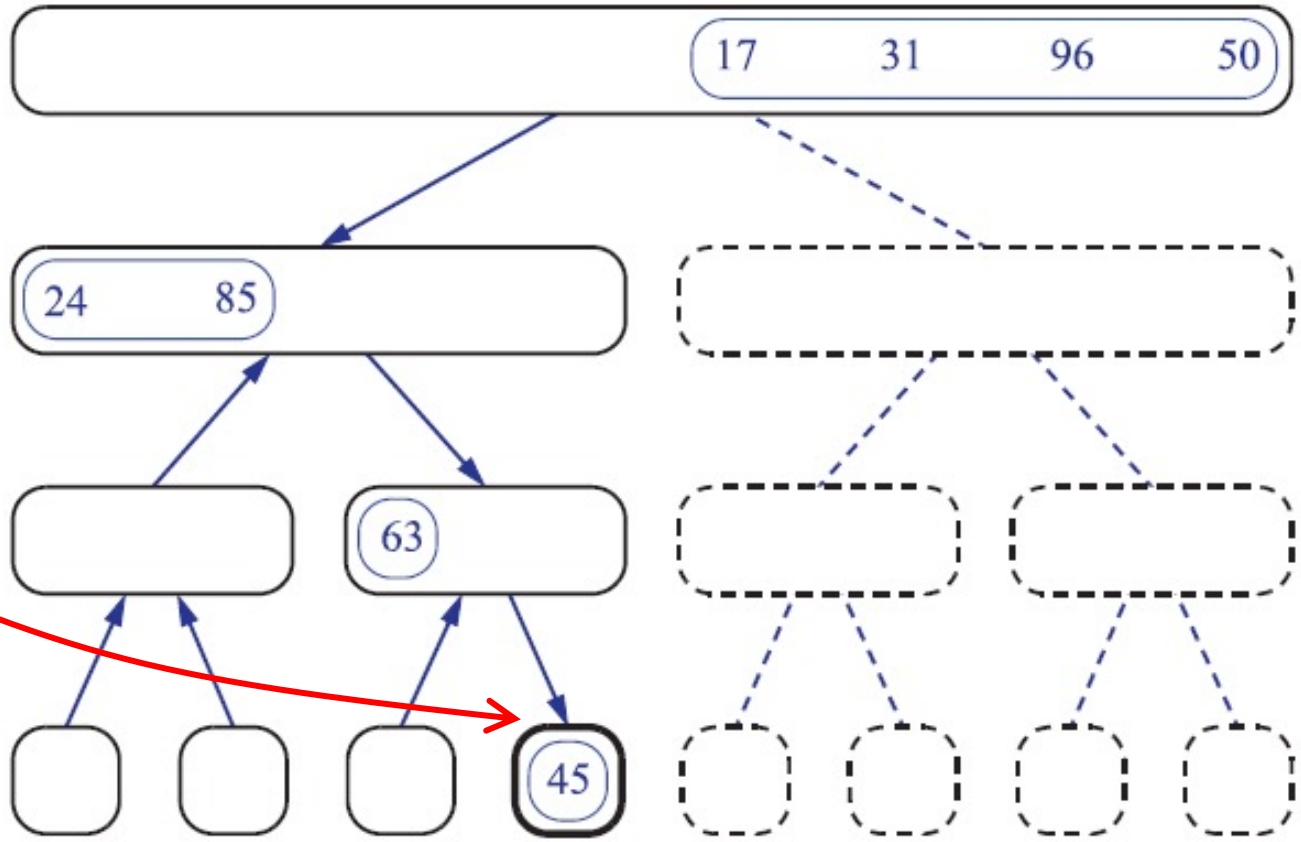
- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half



MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half

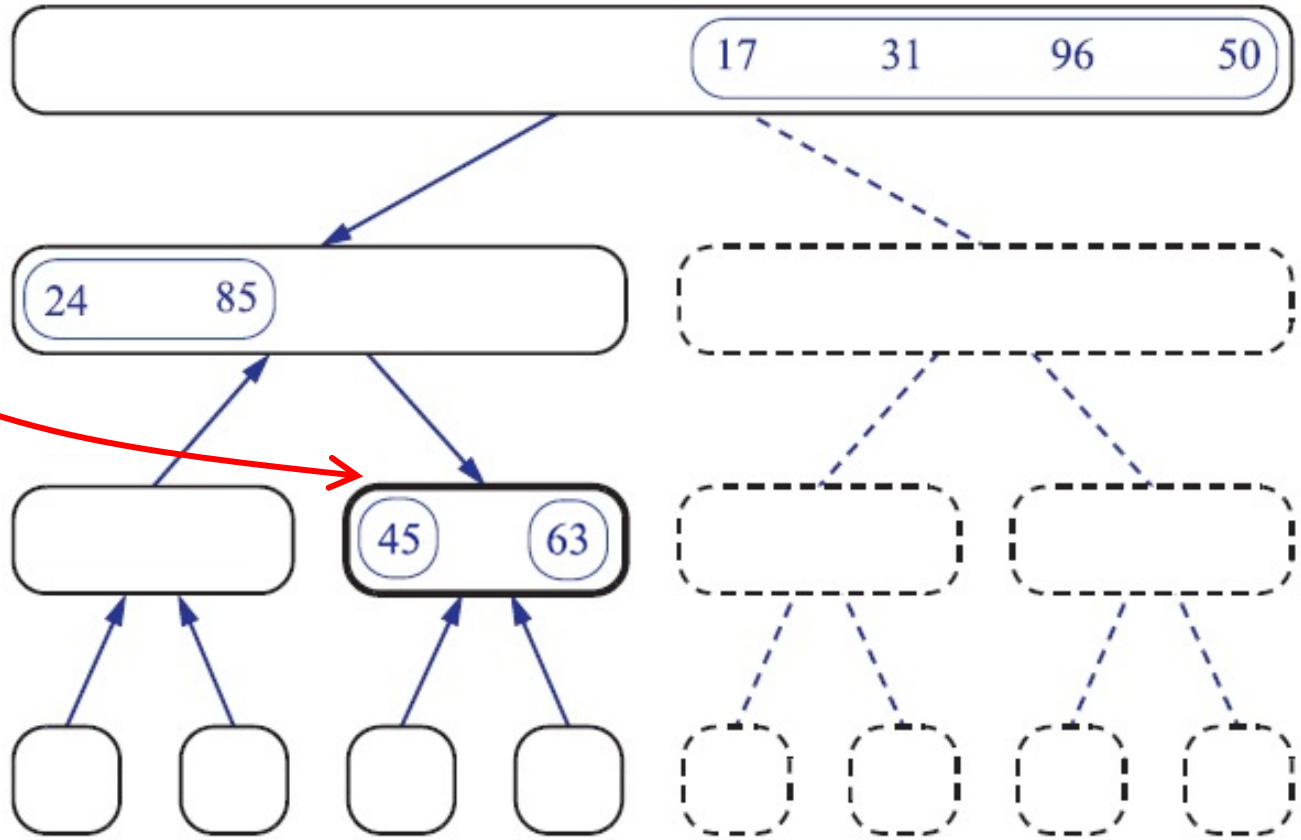
Second half is less than 2 elements; no need for recursion



MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half

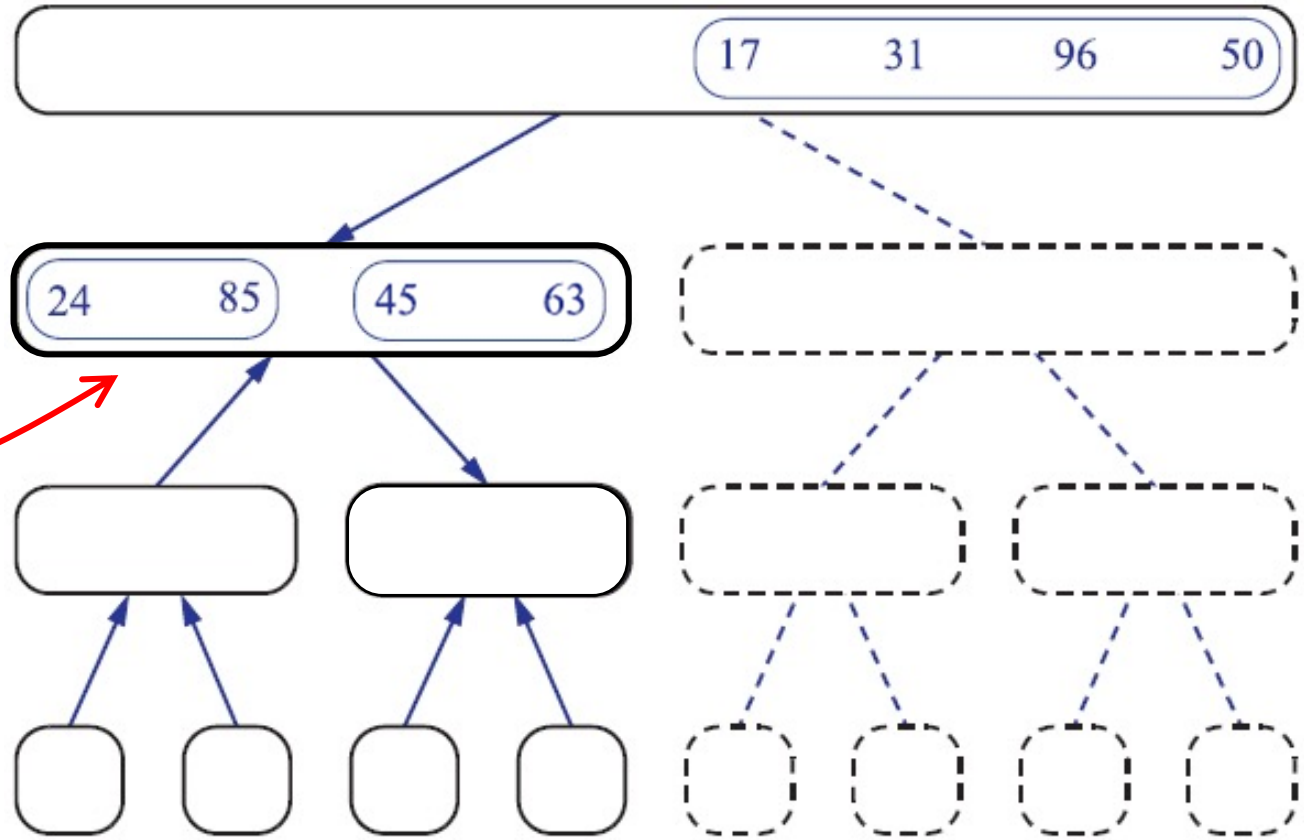
Merging the two halves (63) and (45) **involves sorting** them first, which is why we now have (45) before (63)



MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half

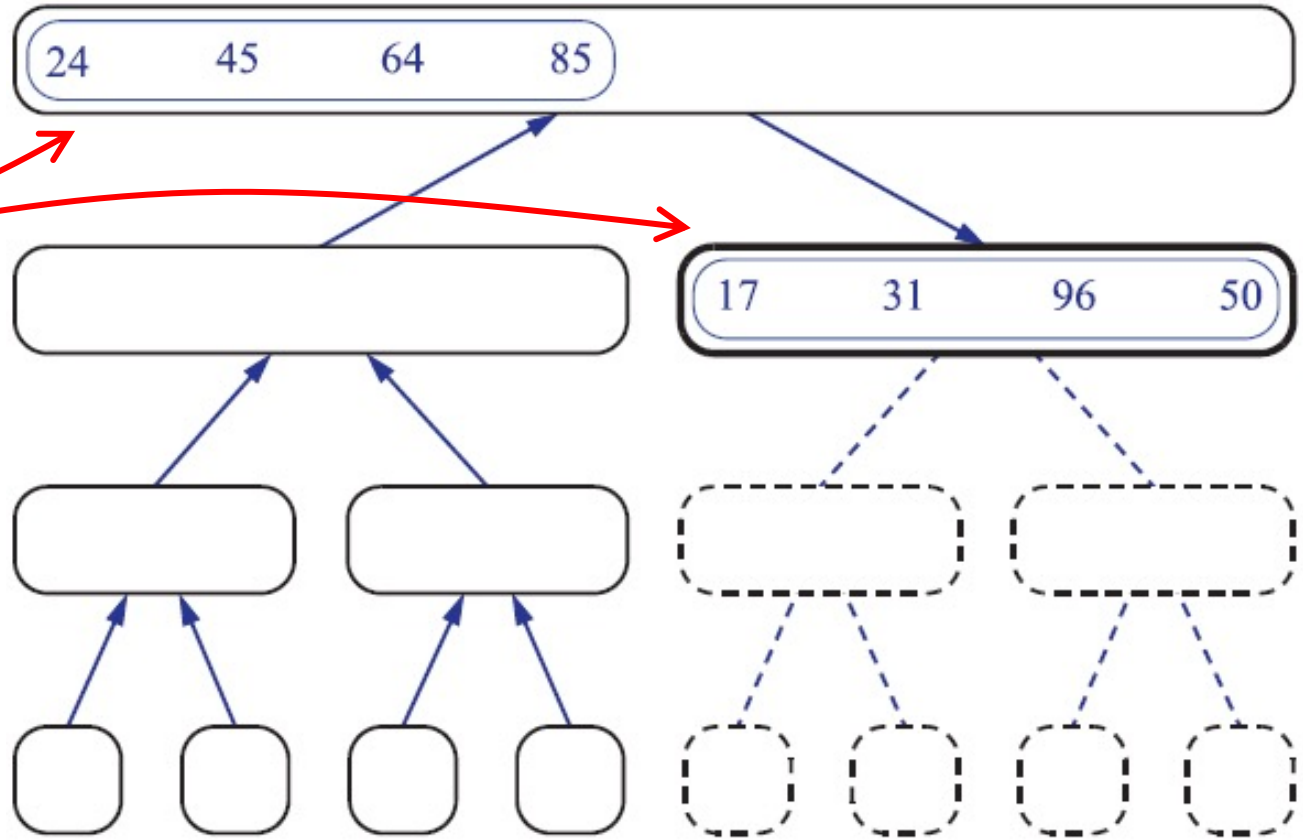
Merging the two halves **involves sorting** them first, which is what we will do next



MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half

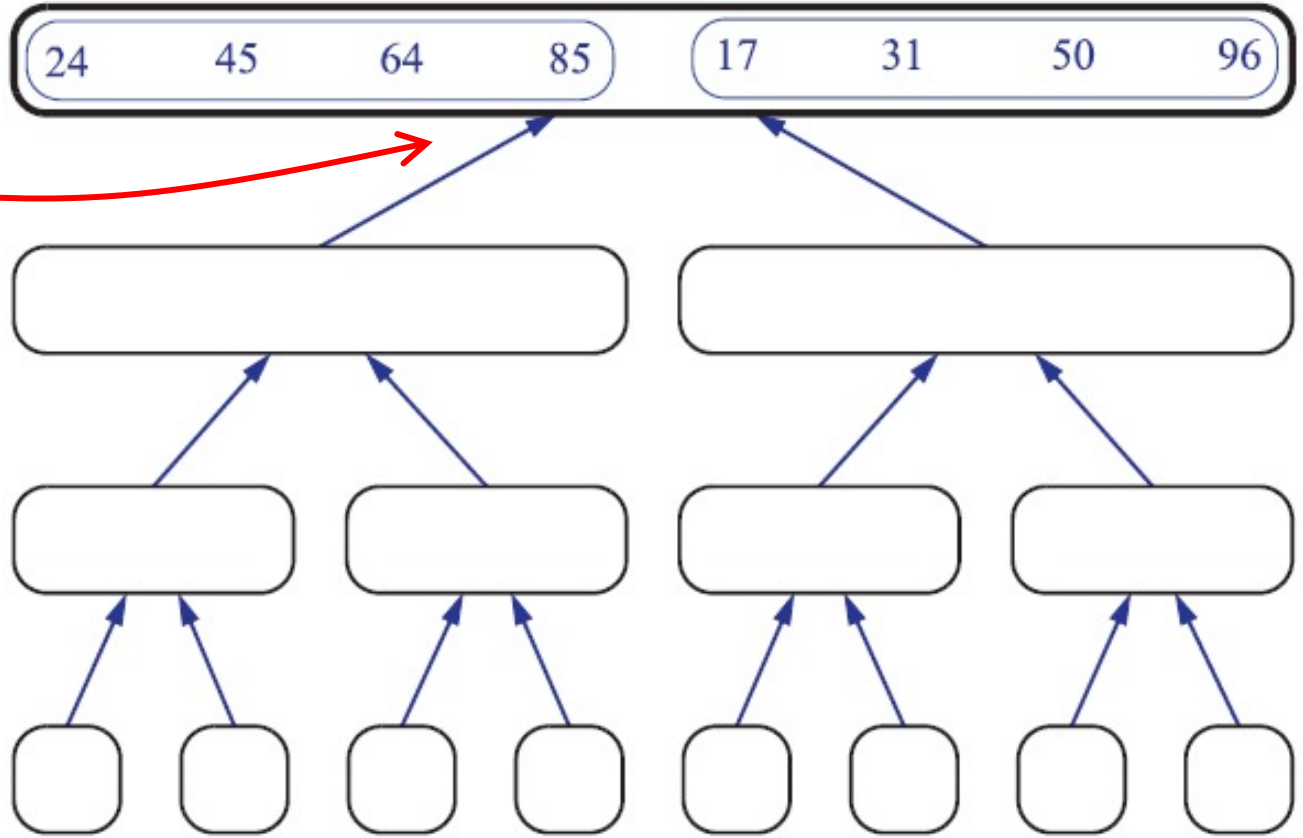
Now that the first half is sorted and merged, we make a recursive call on the second half



MERGE-SORT TREE

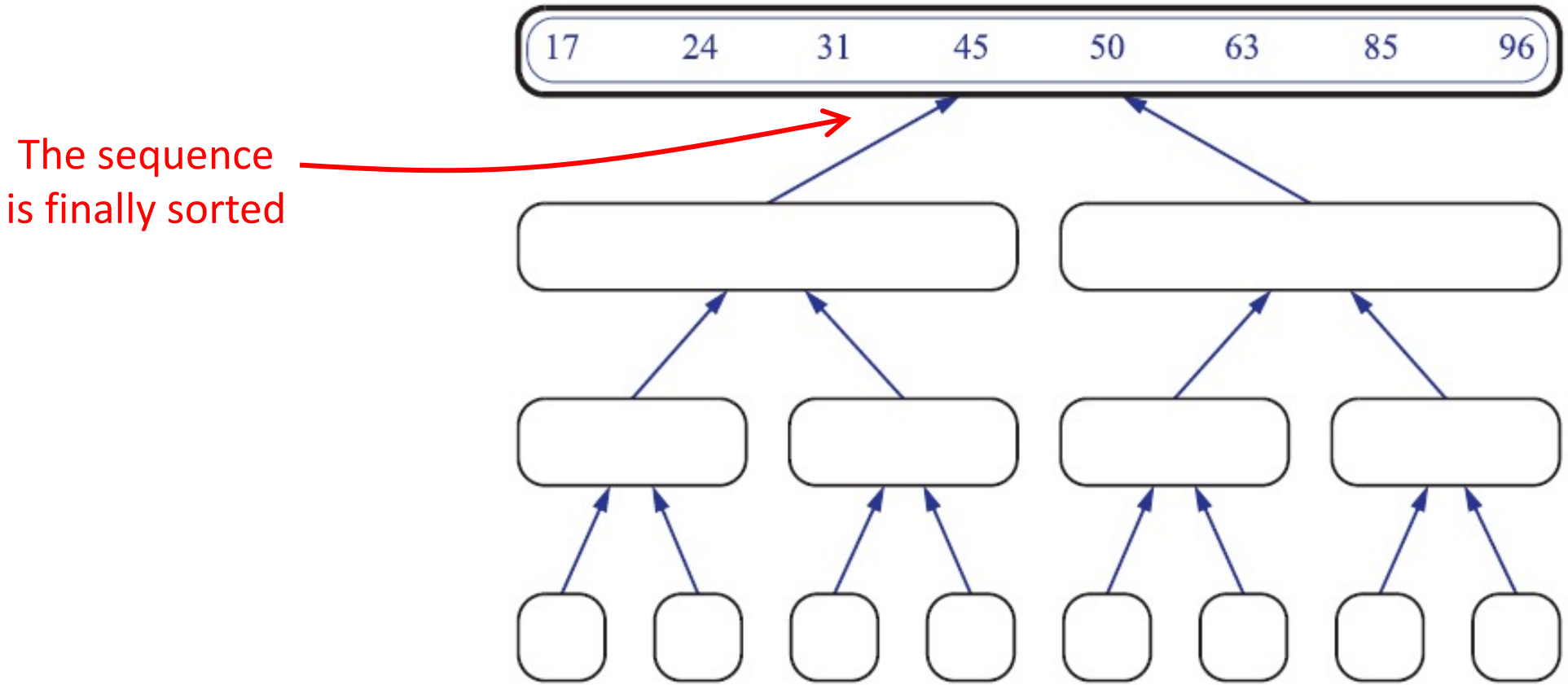
- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half

After many steps, both halves are sorted, and we need to merge them (which **involves sorting**)



MERGE-SORT TREE

- Here is a **more detailed** example showing that the first half is sorted recursively before starting with the second half

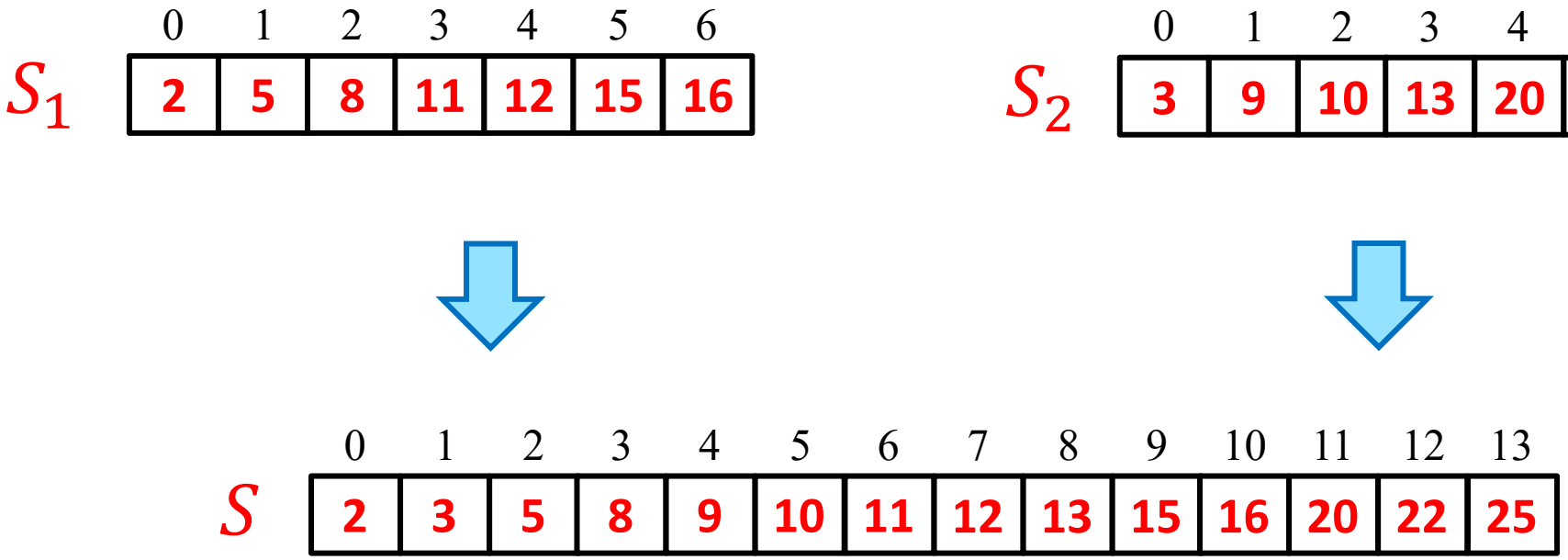


MERGE – ARRAY IMPLEMENTATION

Let's start by analyzing the "merge" step:

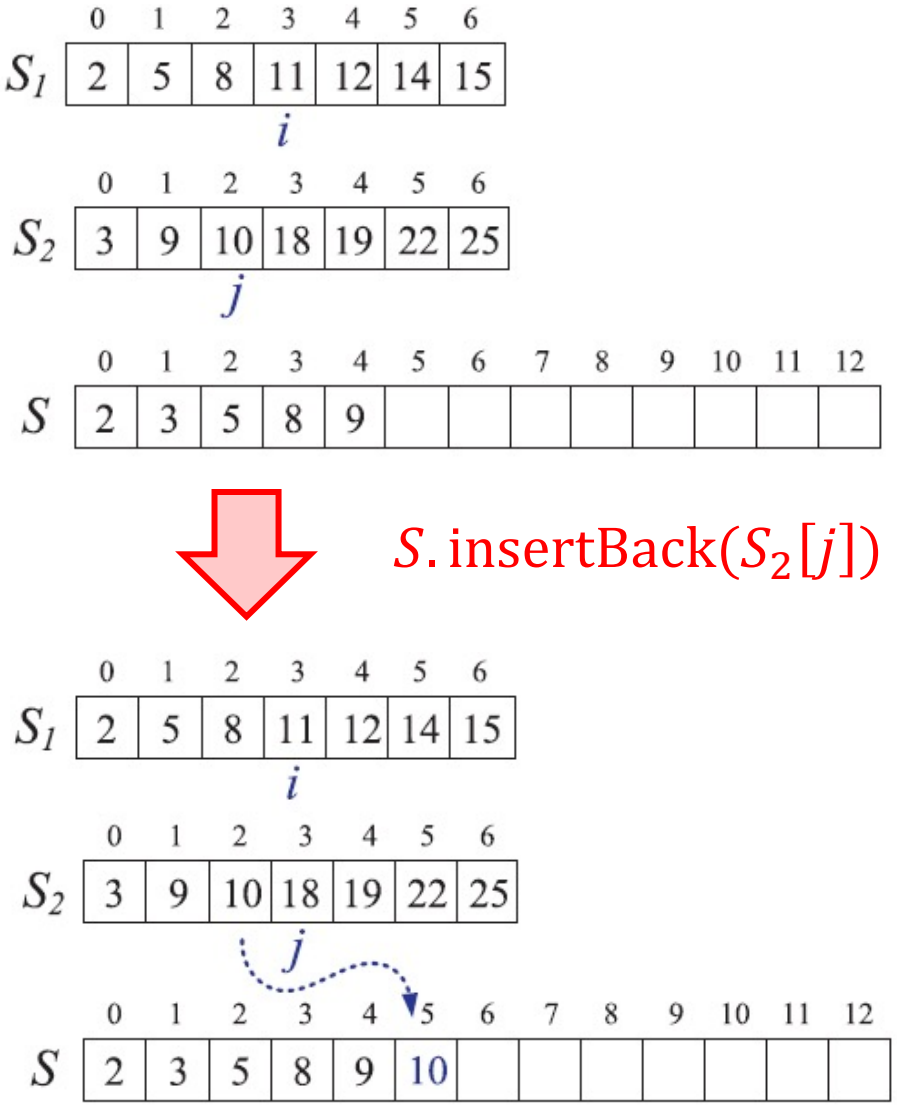
- Given two **sorted** sequences S_1 and S_2, we need to merge them into a **sorted** sequence S
- The analysis depends on whether the sequences are implemented as arrays or as lists

1. With array implementation:



MERGE – ARRAY IMPLEMENTATION

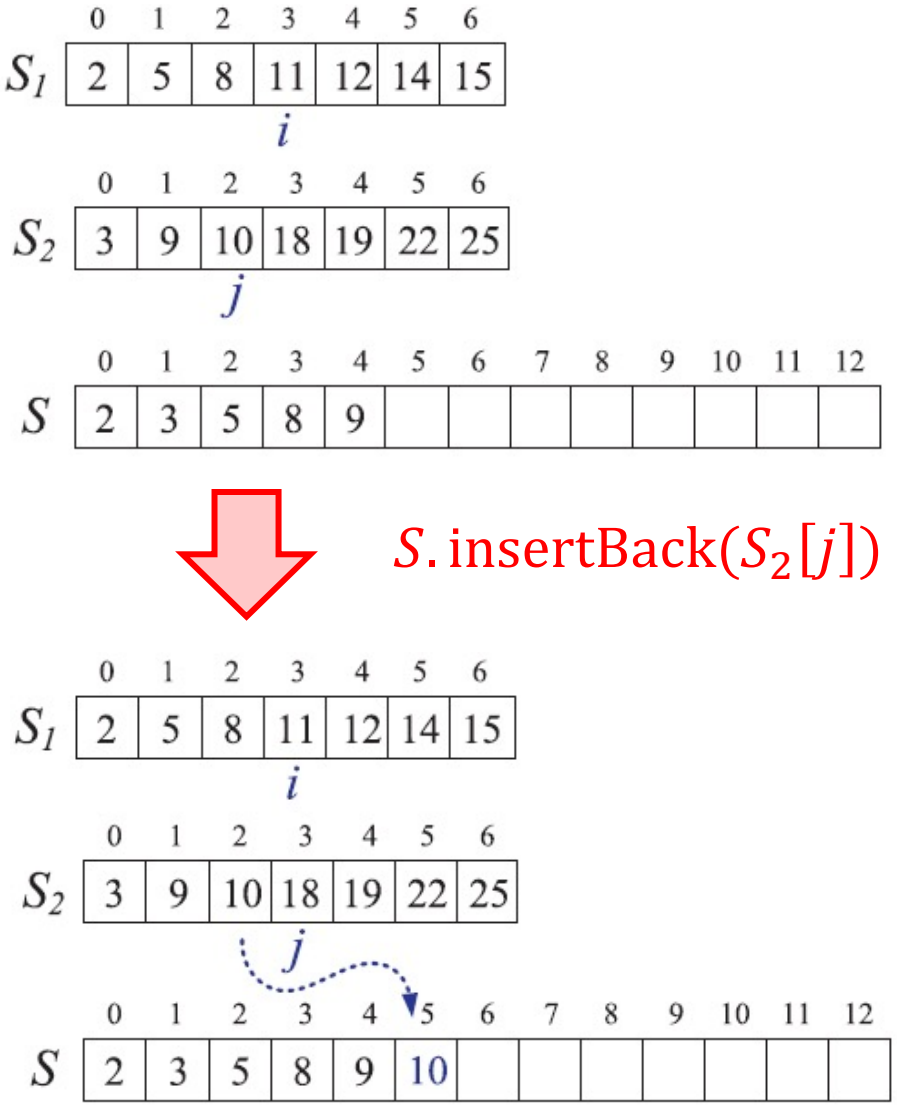
Write the pseudo code of **merge** given an array implementation (see the illustration):



MERGE – ARRAY IMPLEMENTATION

Algorithm merge(S_1, S_2, S):
Input: Sorted sequences S_1 and S_2 and an empty sequence S, all of which are implemented as arrays
Output: Sorted sequence S containing the elements of S_1 and S_2

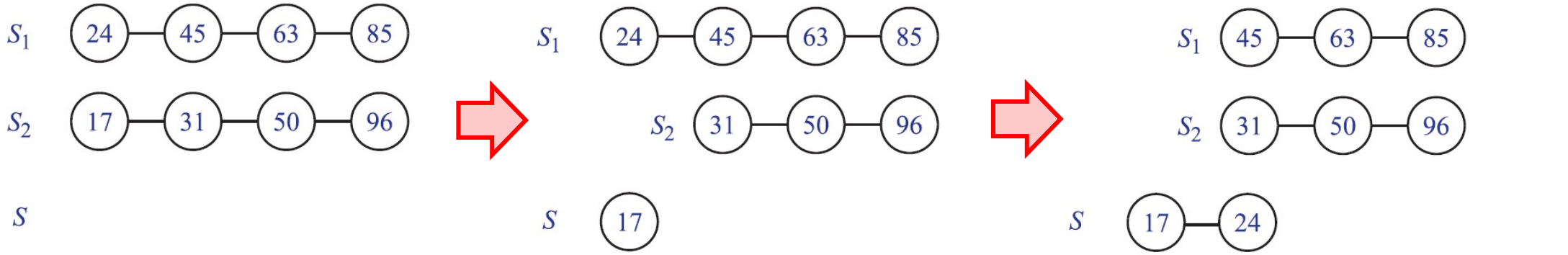
```
i ← j ←0
while i < S1.size() and j < S2.size() do
    if S1[i] ≤ S2[ j] then
        S.insertBack(S1[i]) {copy i^th element of S1 to end of S}
        i ← i+1
    else
        S.insertBack(S2[j]) {copy j^th element of S2 to end of S}
        j ← j+1
while i < S1.size() do {copy the remaining elements of S1 to S}
    S.insertBack(S1[i])
    i ← i+1
while j < S2.size() do {copy the remaining elements of S2 to S}
    S.insertBack(S2[ j])
    j ← j+1
```

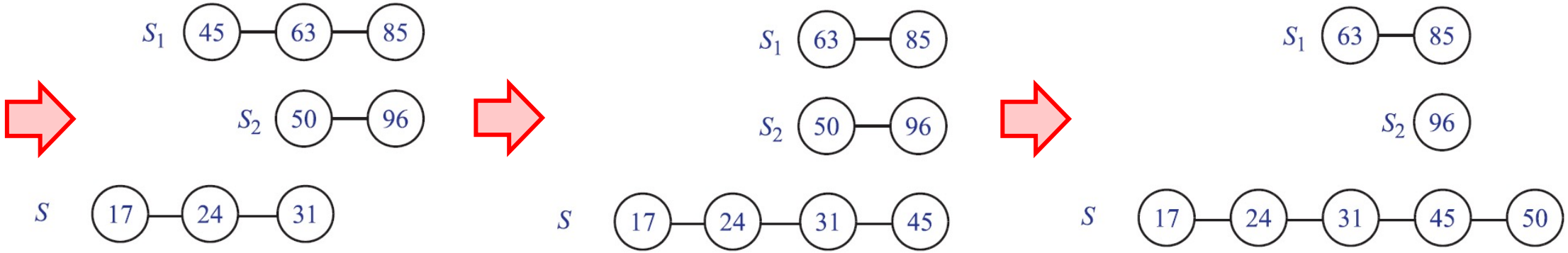


MERGE – LIST IMPLEMENTATION

2. With lists, we remove the element at the **front** of S_1 or S_2 and add it to the **end** of S

- Repeat until S_1 or S_2 is empty, at which point copy the remainder of the other list to S

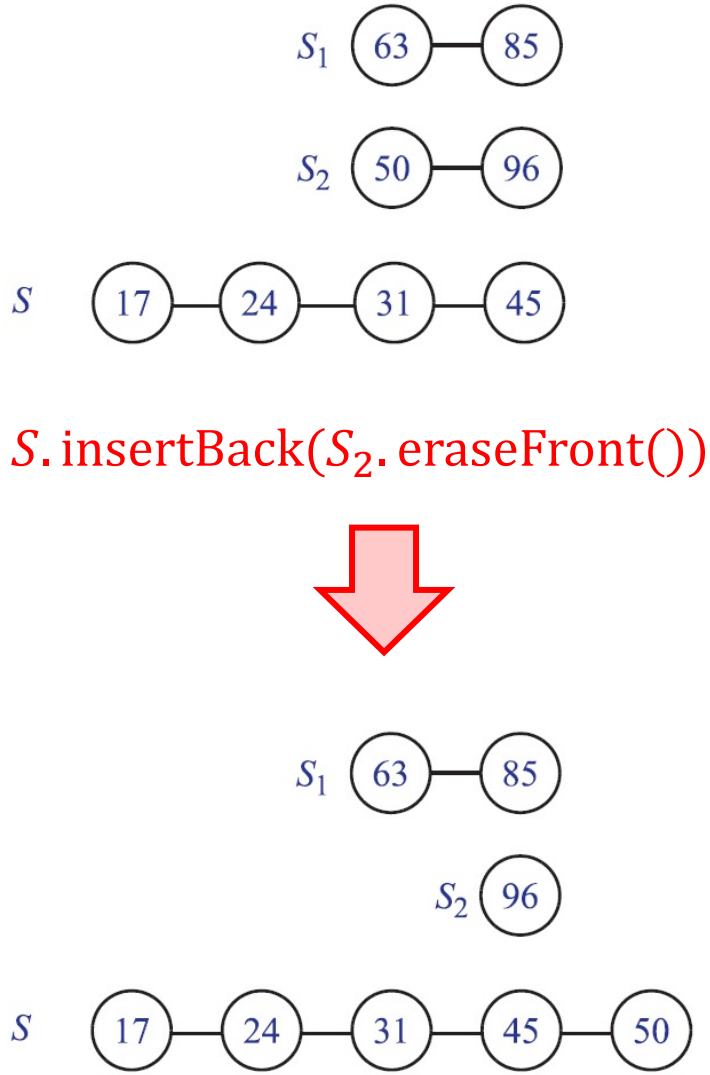




and so on…

MERGE – LIST IMPLEMENTATION

Write the pseudo code of **merge** given a list implementation (see the illustration):



MERGE – LIST IMPLEMENTATION

Algorithm merge(S_1, S_2, S):

Input: Sorted sequences S_1 and S_2 and an empty sequence S, implemented as linked lists

Output: Sorted sequence S containing the elements from S_1 and S_2

while S_1 is not empty **and** S_2 is not empty **do**

if S_1.front().element() ≤ S_2.front().element() **then**

{move the first element of S_1 at the end of S}

S.insertBack(S_1.eraseFront())

else

{move the first element of S_2 at the end of S}

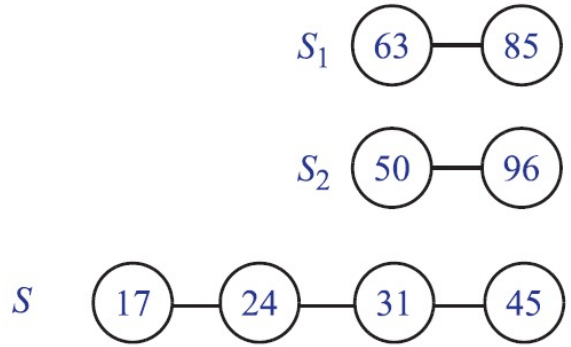
S.insertBack(S_2.eraseFront())

while S_1 is not empty **do** {move the remaining elements of S_1 to S}

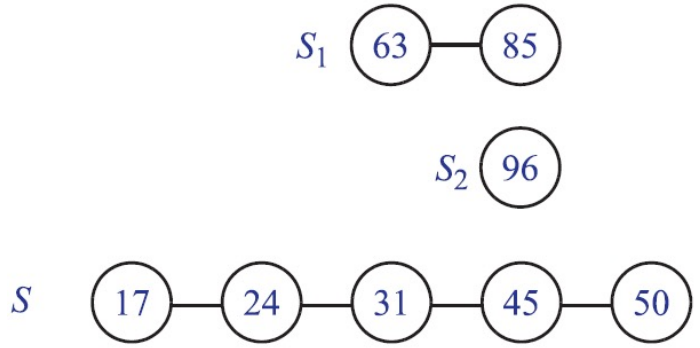
S.insertBack(S_1.eraseFront())

while S_2 is not empty **do** {move the remaining elements of S_2 to S}

S.insertBack(S_2.eraseFront())



S.insertBack(S_2.eraseFront())



COMPLEXITY OF MERGE-SORT

- Here we analyze the complexity of the "merge" step, which merges S_1 and S_2:
  - merge has **three while loops**, the operations inside each loop take O(1) time.
  - But how do we count the total number of iterations over all three loops?

Array-based implementation

```
i ← j ←0
while i < S1.size() and j < S2.size() do
    if S1[i] ≤ S2[j] then
        S.insertBack(S1[i])
        i ← i+1
    else
        S.insertBack(S2[j])
        j ← j+1
while i < S1.size() do
    S.insertBack(S1[i])
    i ← i+1
while j < S2.size() do
    S.insertBack(S2[j])
    j ← j+1
```

List-based implementation

```
while S1 is not empty and S2 is not empty do
    if S1.front().element() ≤ S2.front().element() then
        S.insertBack(S1.eraseFront())
    else
        S.insertBack(S2.eraseFront())
while S1 is not empty do
    S.insertBack(S1.eraseFront())
while S2 is not empty do
    S.insertBack(S2.eraseFront())
```

COMPLEXITY OF MERGE-SORT

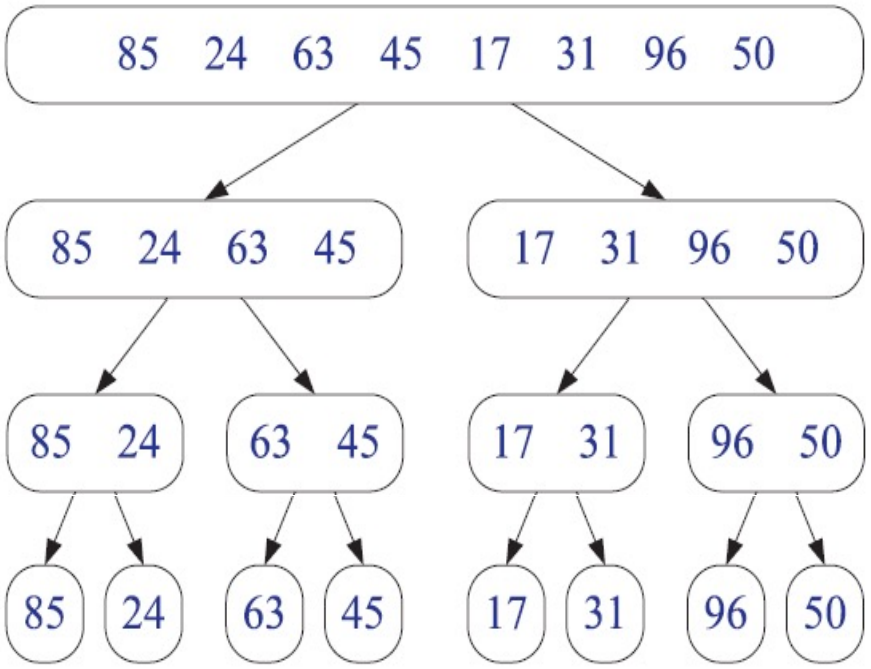
- Let us first analyze the complexity of the "merge" step, which merges S_1 and S_2:
  - merge has **three while loops**, the operations inside each loop take O(1) time.
  - But how do we count the total number of iterations over all three loops?
  - Let n_1 and n_2 be the number of elements of S_1 and S_2
  - Key observation:

    During each iteration of one of the loops, one element is copied or moved from either S_1 or S_2 into S (**and that element is no longer considered**).
  - This implies that the total number of iterations over all three loops is $n_1 + n_2$
  - Thus, the running time of merge is $O(n_1 + n_2)$, i.e., **proportional to the number of elements being merged**!
- Next, we analyze the complexity of the entire merge-sort algorithm

COMPLEXITY OF MERGE-SORT

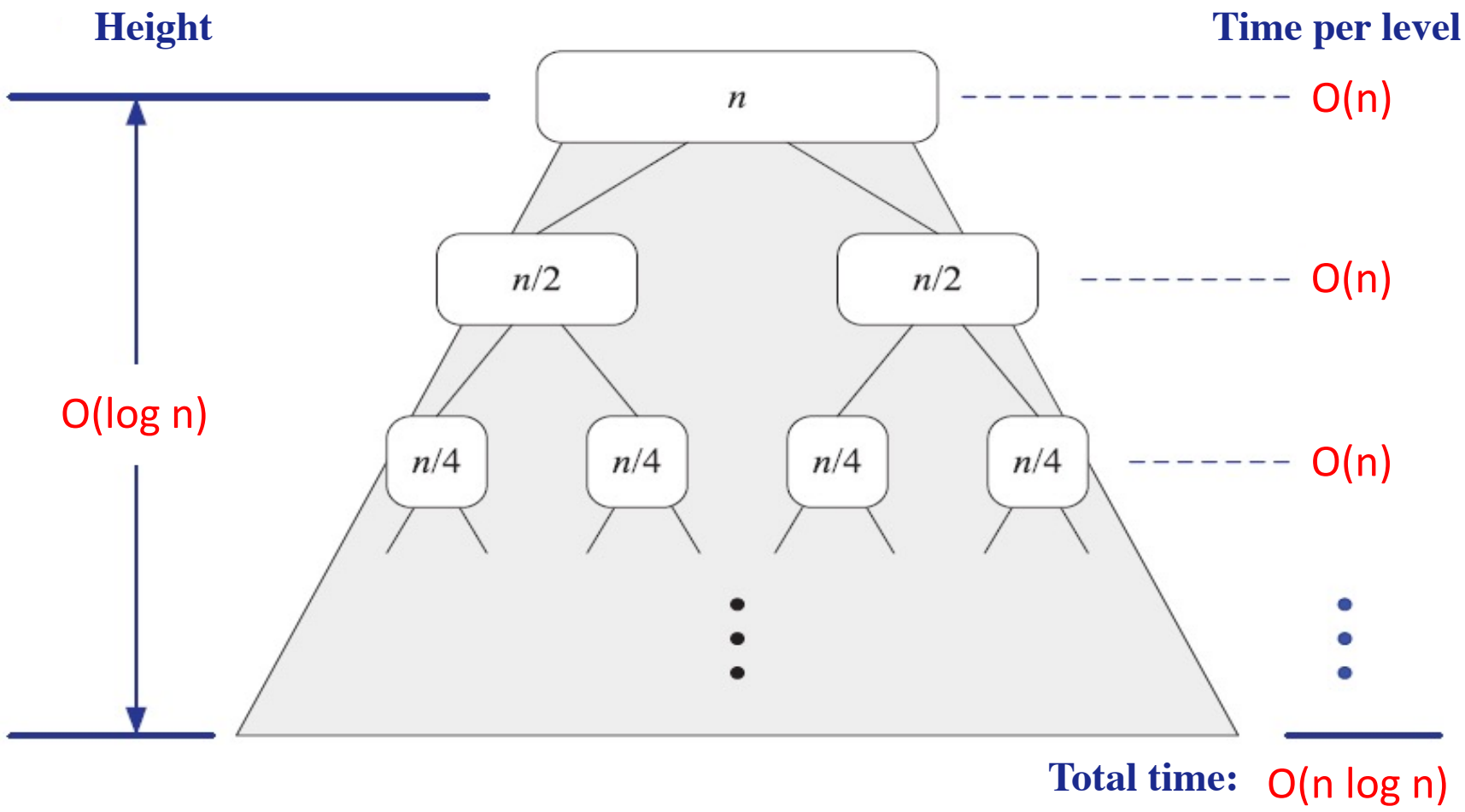
- To simplify the analysis, let's assume that n is a power of 2 (the result of our analysis also holds when n is not a power of 2)
- For each node v in the merge-sort tree, let "**the time spent at** v" be the running time of the recursive call associated with v excluding the time spent waiting for v's children

- Let i be v's depth. Then, v contains $n/2^i$ elements
- The time spent at node v is $O(n/2^i)$ time for **dividing** and **merging** the elements
- Time spent on **all nodes** at depth i is $O(n)$ since:
  1. There are 2^i nodes at depth i
  2. Each node requires $O(n/2^i)$ time
  - ➢ Thus, each level i takes $O(2^i \times n/2^i) = O(n)$ time
- **Overall time** for all $O(\log n)$ levels is $O(n \log n)$



COMPLEXITY OF MERGE-SORT

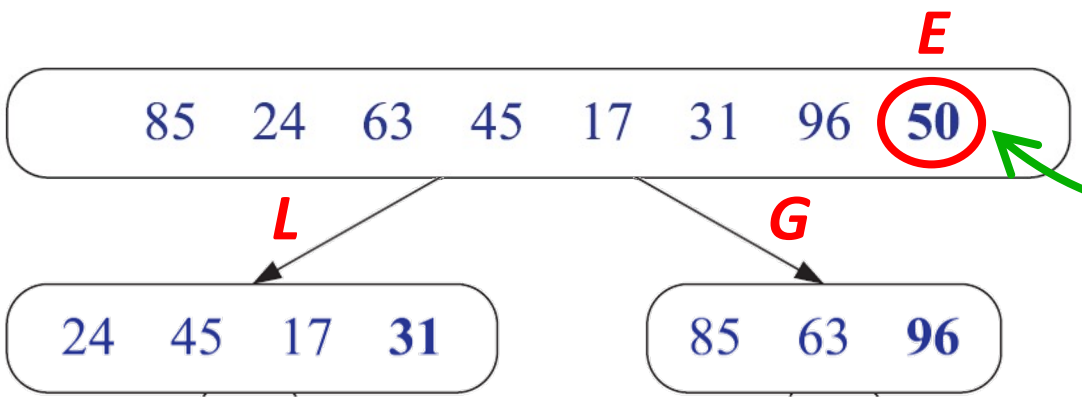
- Here is an illustration summarizing the complexity of merge-sort:



QUICK-SORT

QUICK-SORT

- Quick-sort is a divide-and-conquer algorithm that sort sequence S of n elements:
  1. Divide: If S contains at least two element, select an element p to be the **pivot** and divide S into three sets: L (**less** than p); E (**equal** to p); G (**greater** than p)

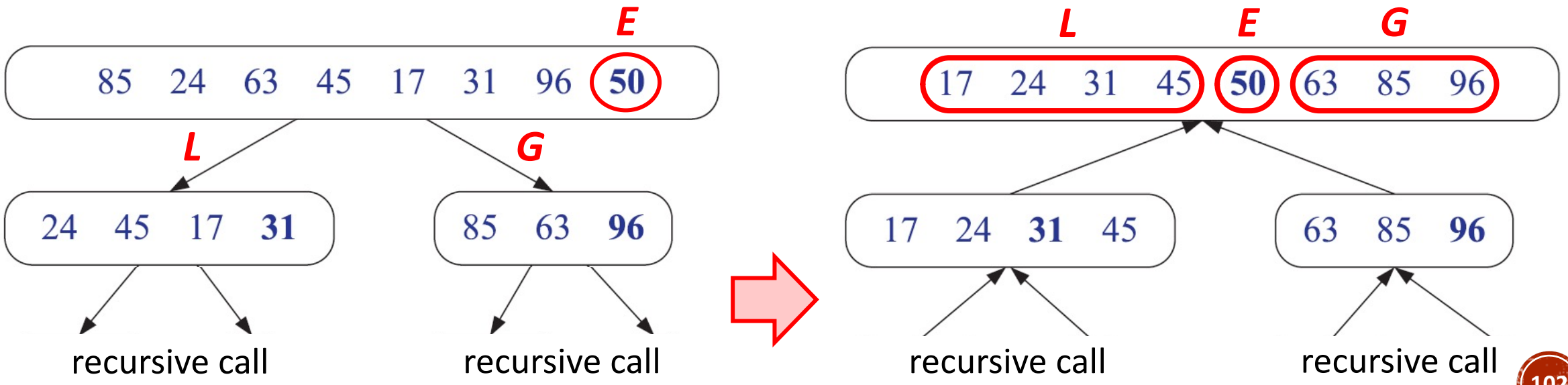


as common practice, choose p to be the last element in S (in this example, p = **50**)

The pivot can be selected randomly as well

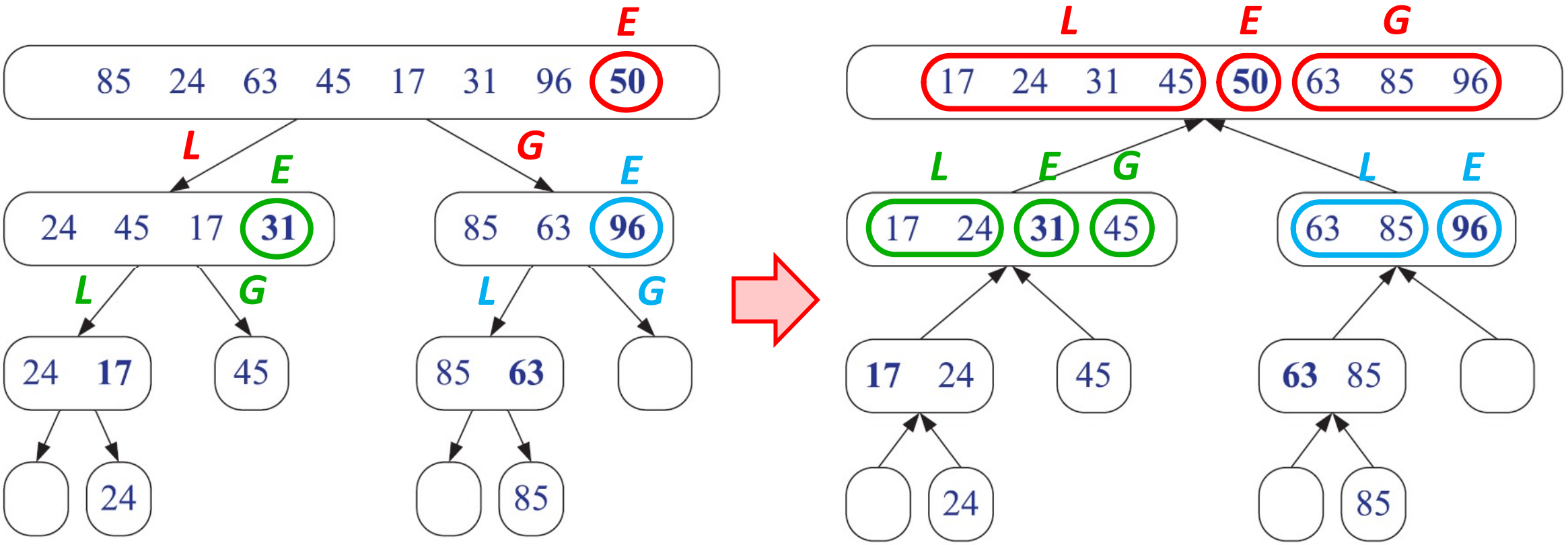
QUICK-SORT

- Quick-sort is a divide-and-conquer algorithm that sort sequence S of n elements:
  1. Divide: If S contains at least two element, select an element p to be the **pivot** and divide S into three sets: L (**less** than p); E (**equal** to p); G (**greater** than p)
  2. Recur: Recursively **sort** L and G
  3. Conquer: Put the elements back into S, by putting L then E then G.



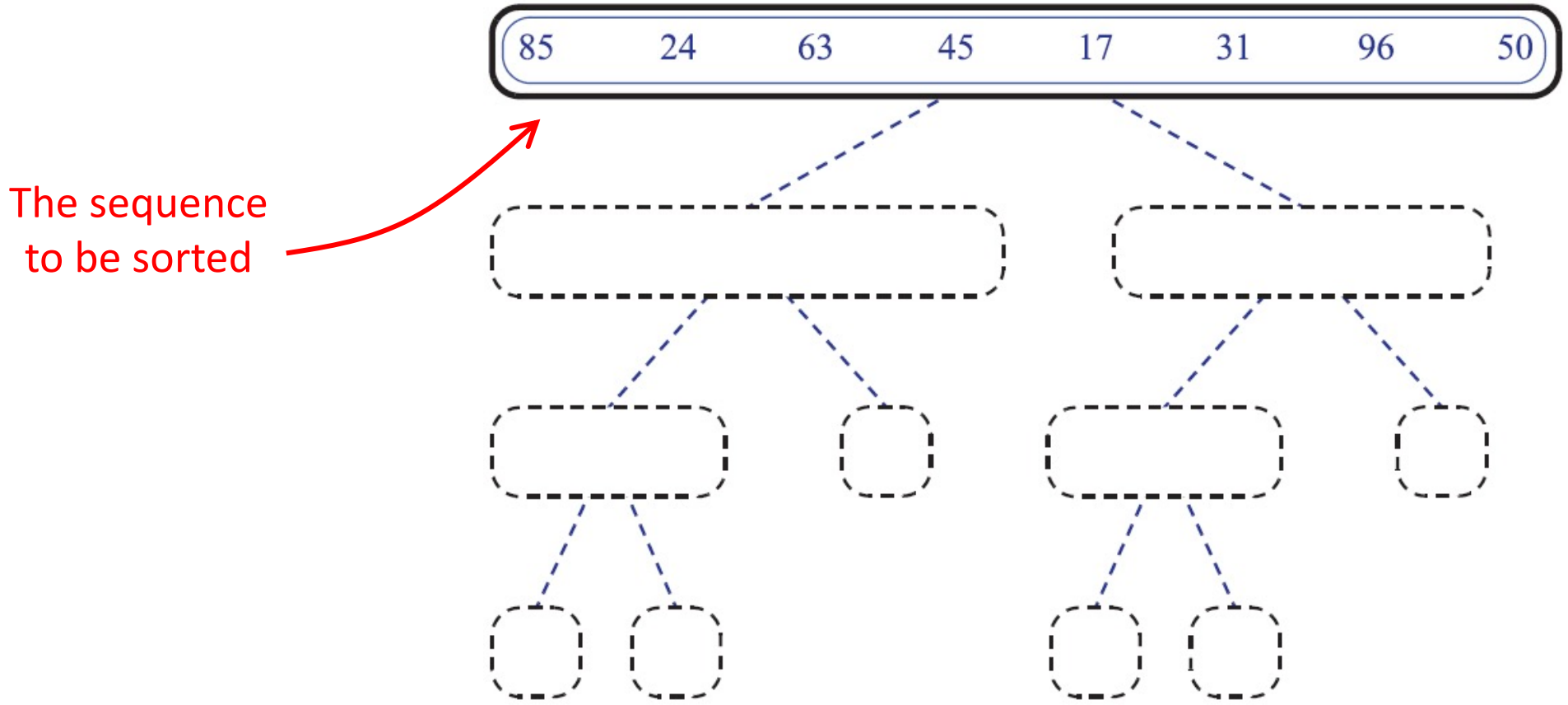
QUICK-SORT TREE

- Here is an example of a quick-sort tree



QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**

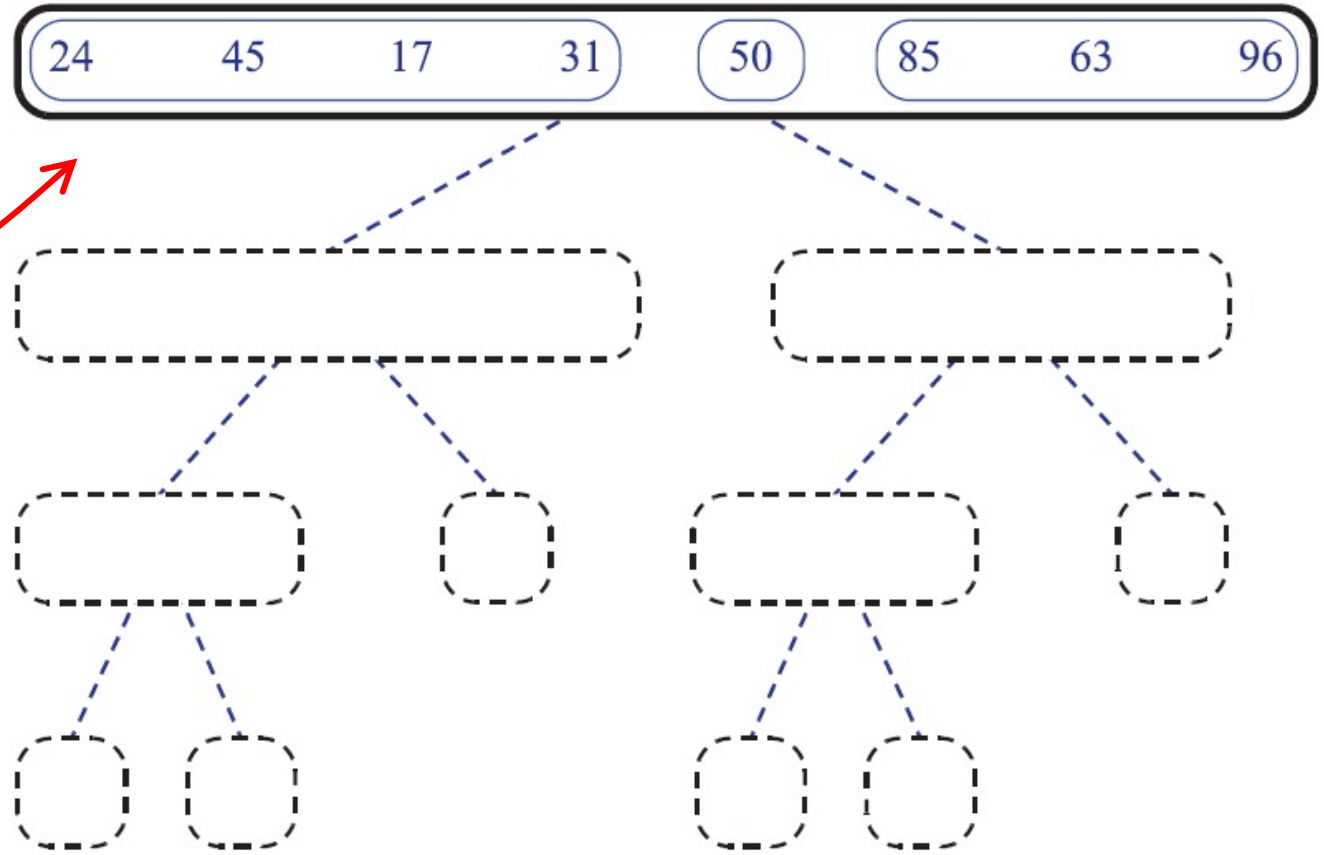


QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**

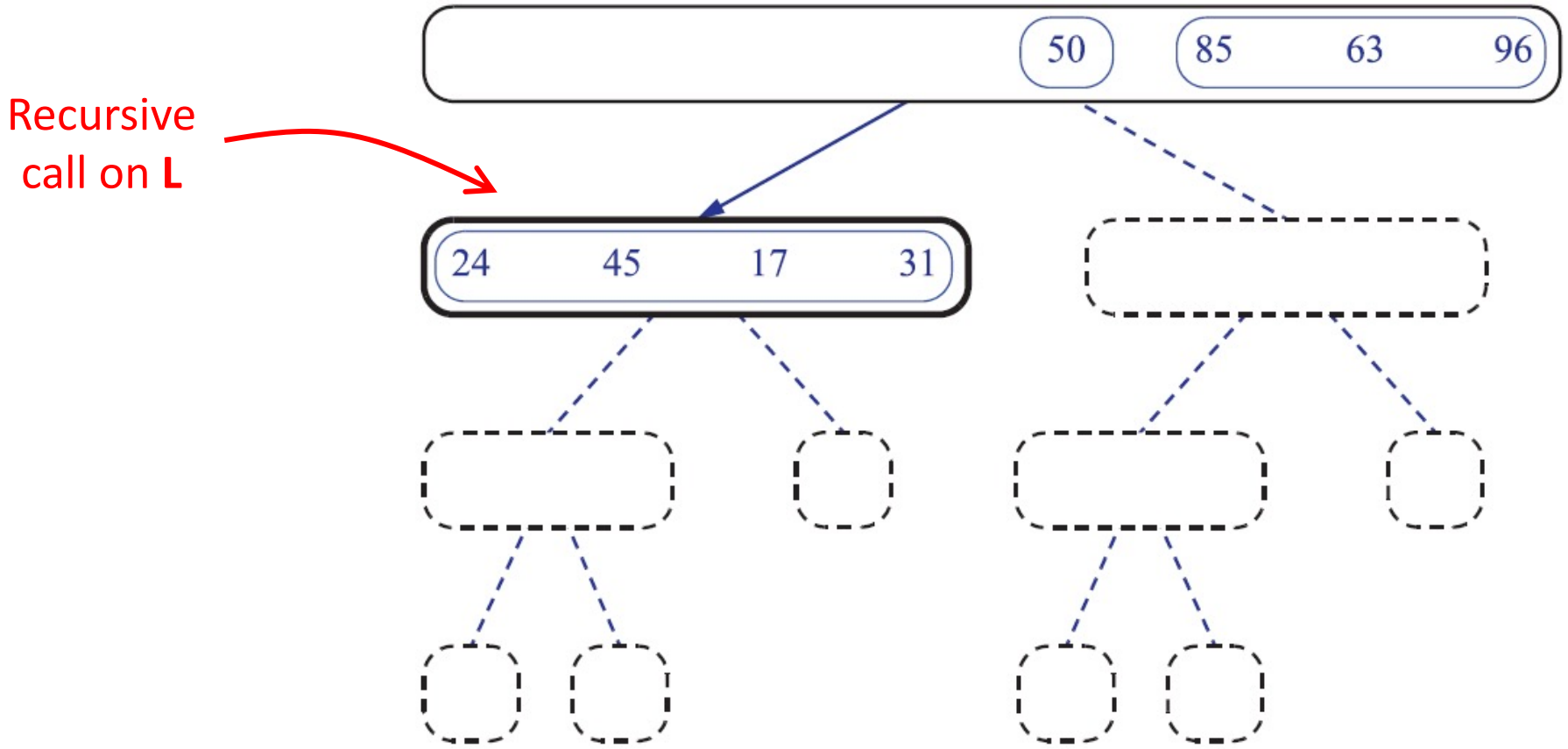
Divide it into:

- **L**: less than the pivot
- **E**: equal to the pivot
- **G**: greater than the pivot



QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**

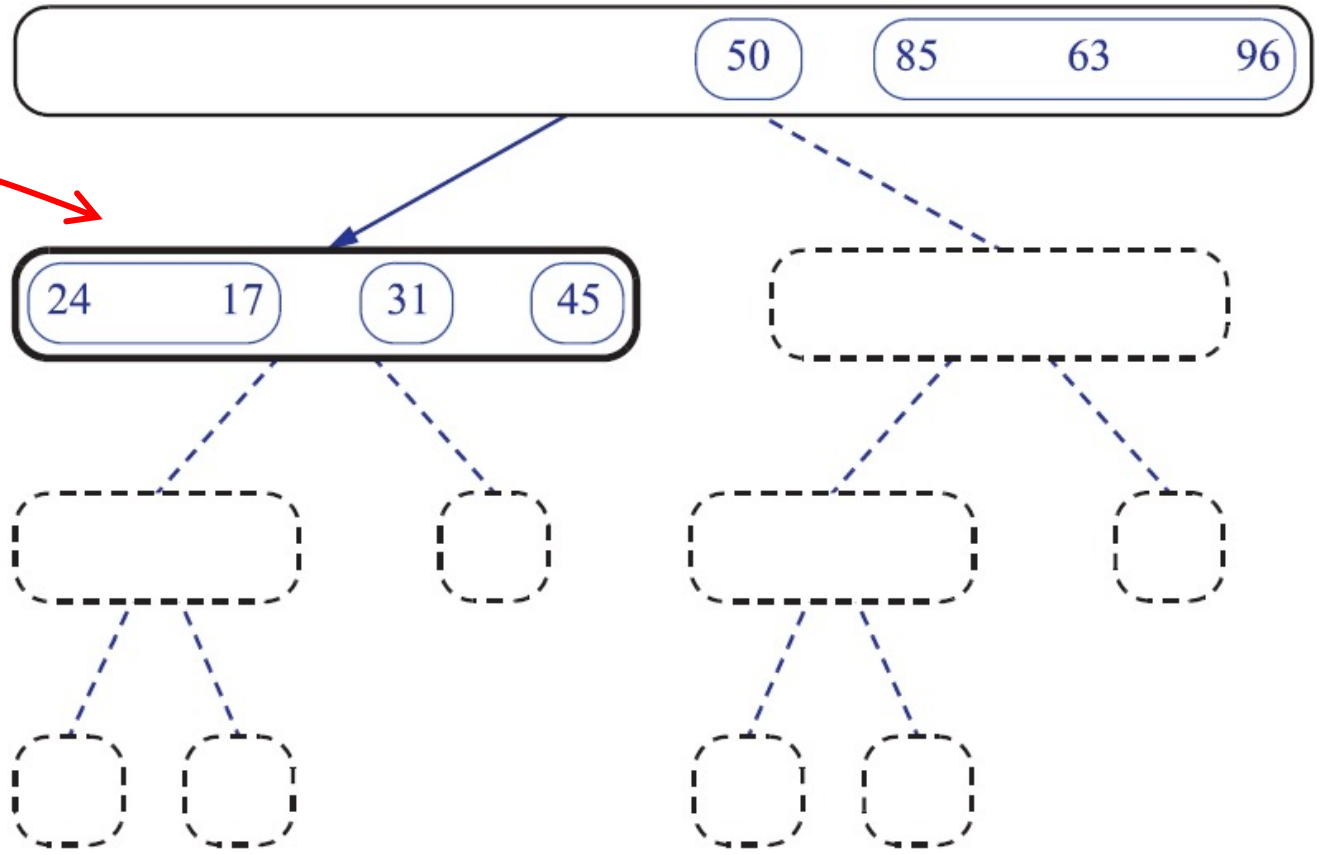


QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**

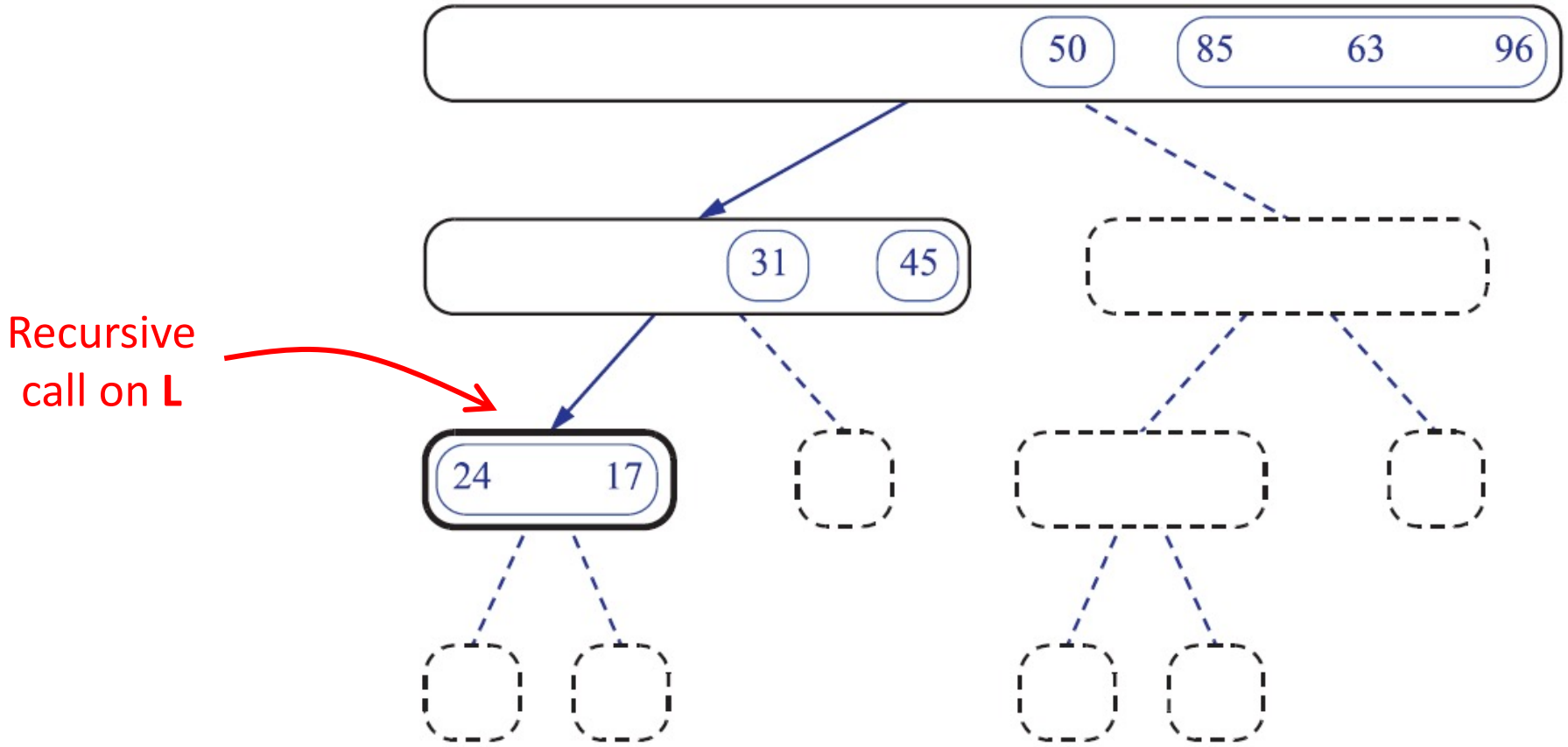
Divide it into:

- **L**: less than the pivot
- **E**: equal to the pivot
- **G**: greater than the pivot



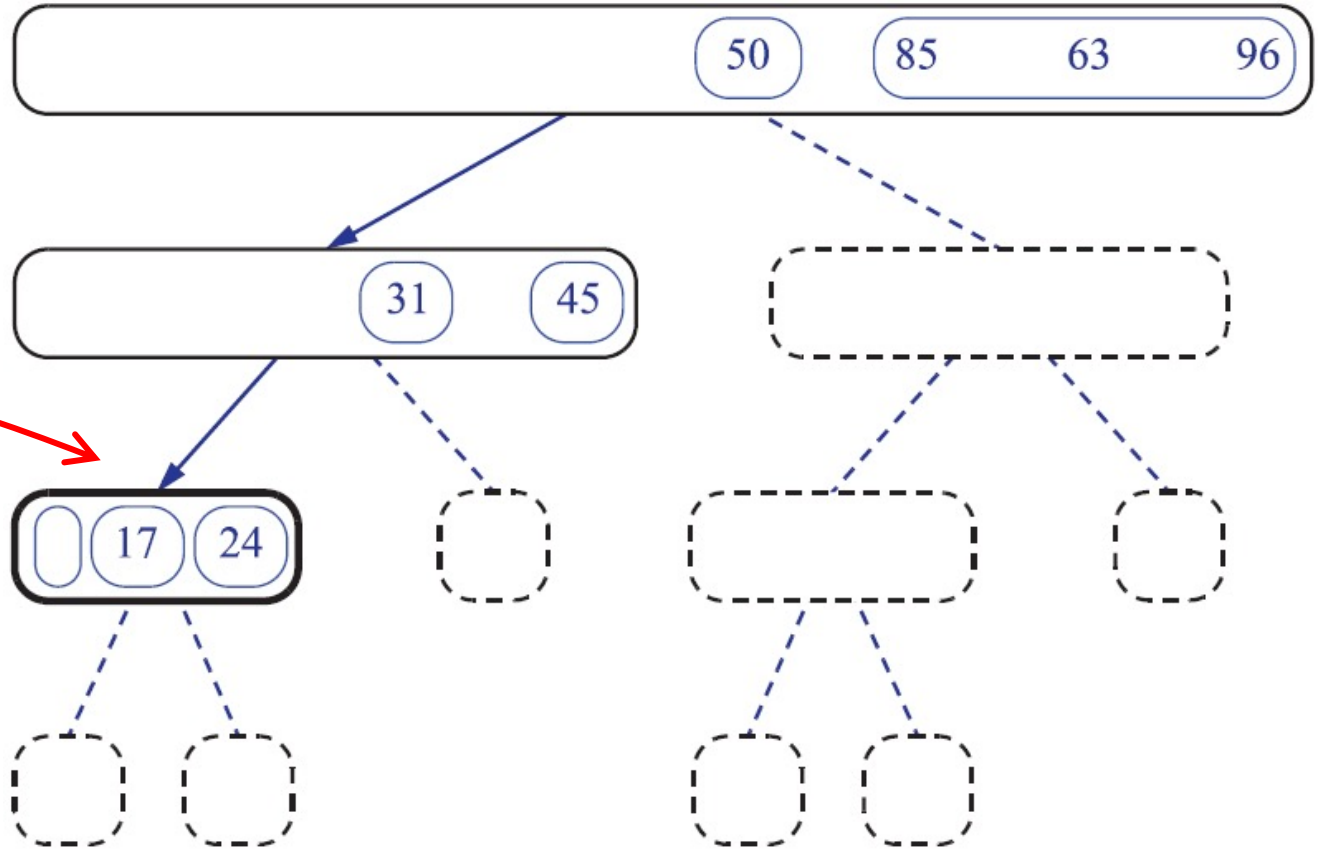
QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**



QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**



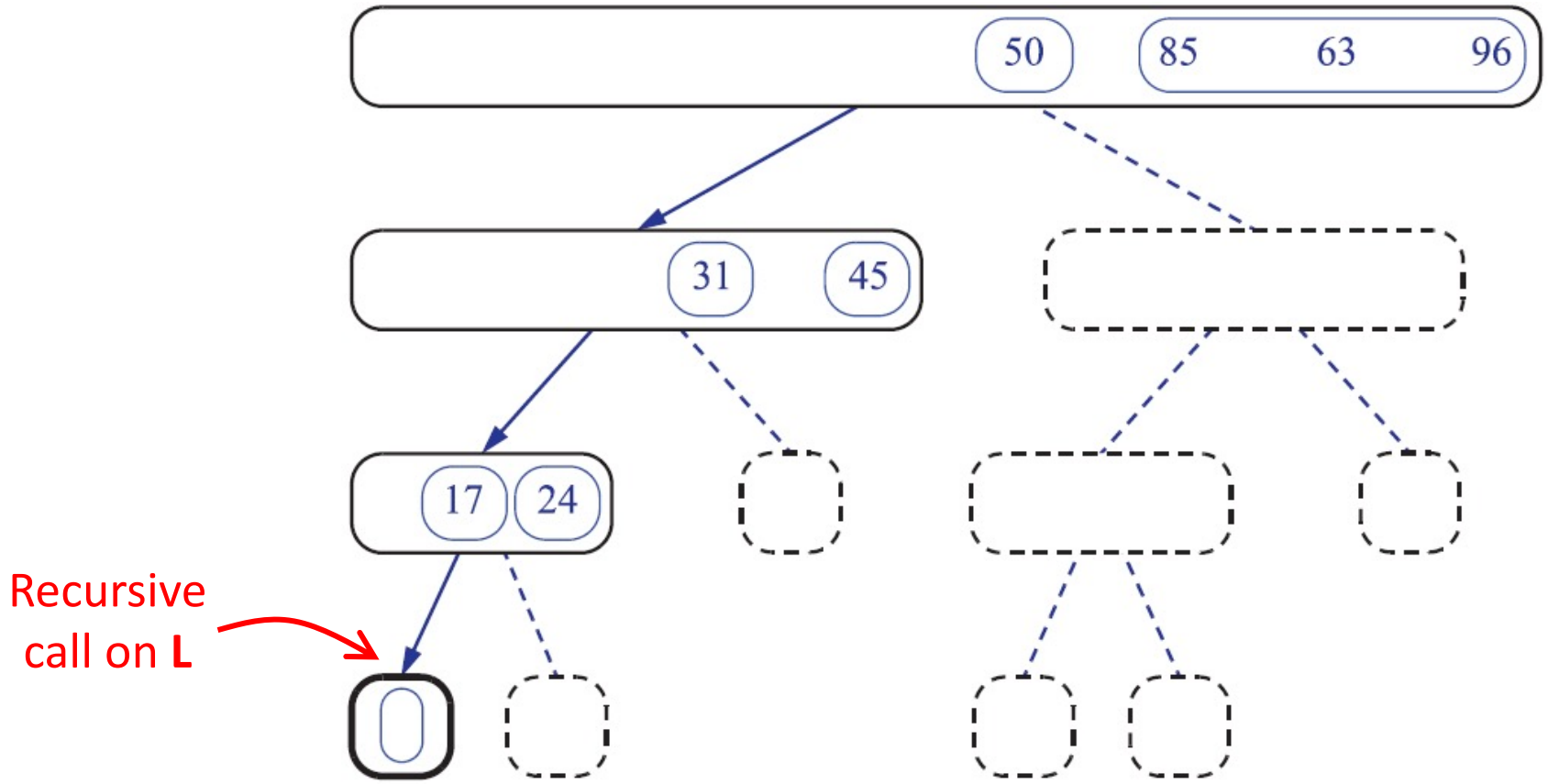
Divide it into:

- **L**: less than the pivot
- **E**: equal to the pivot
- **G**: greater than the pivot

Here, L is empty!

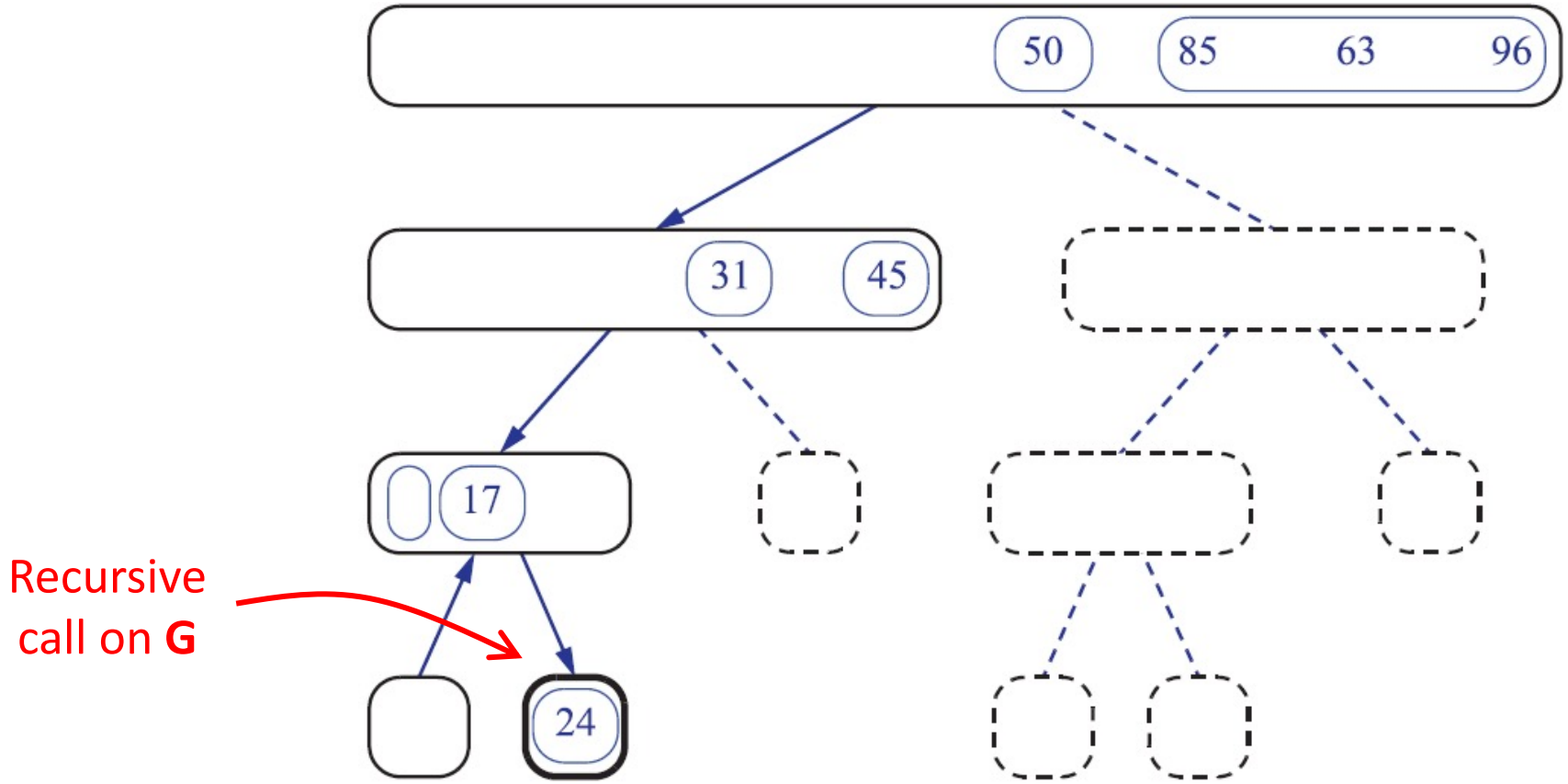
QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**



QUICK-SORT TREE

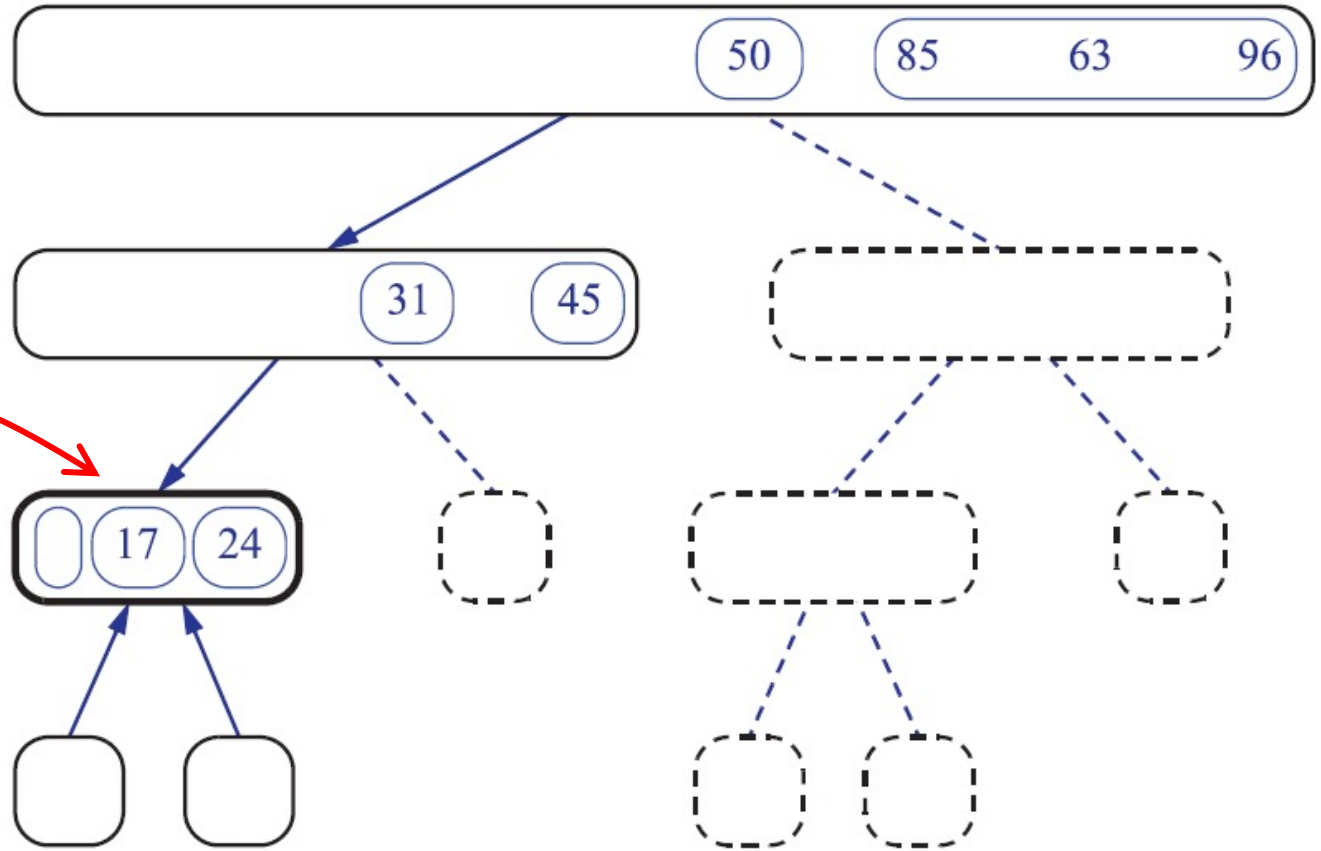
- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**



QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**

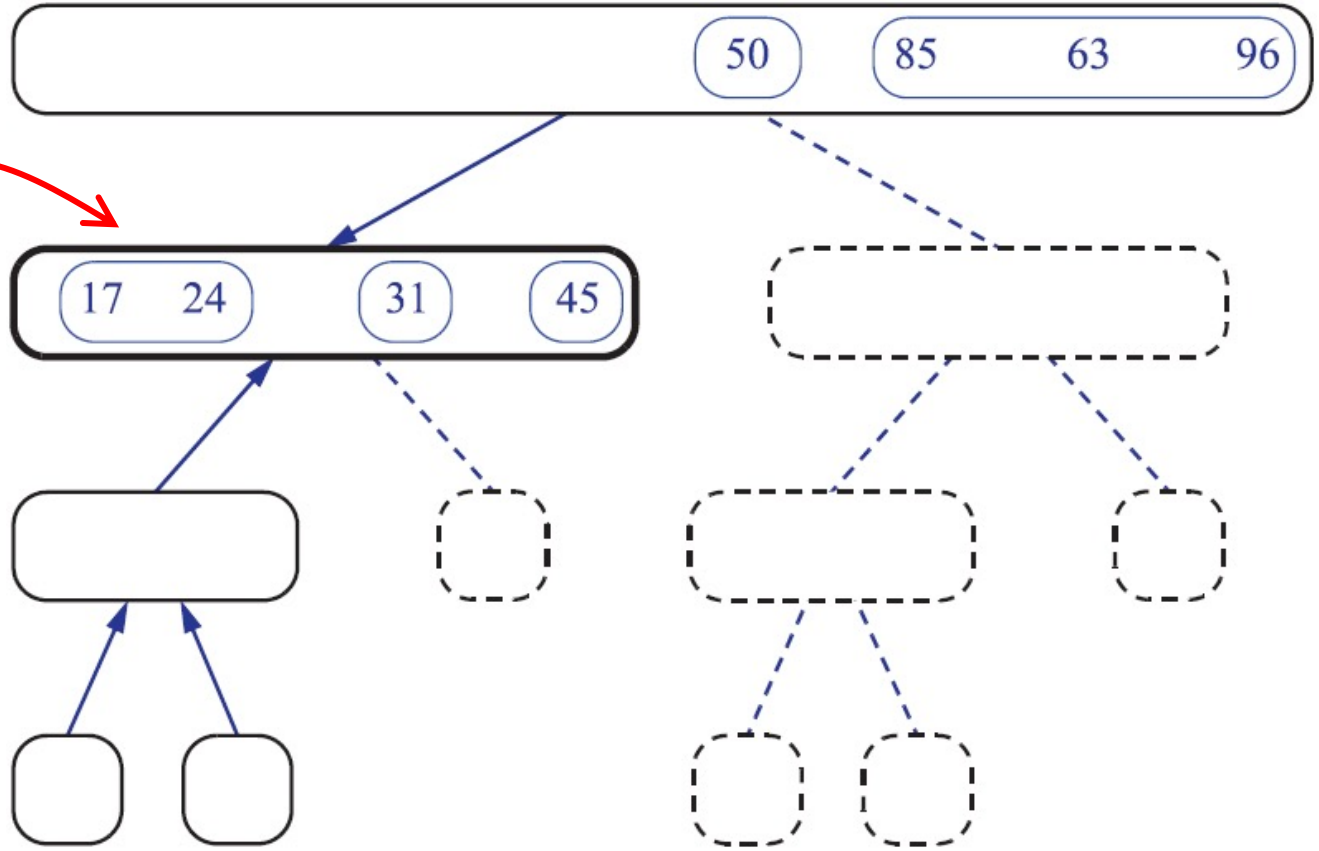
Merge them by:
- first putting **L**
- followed by **E**
- followed by **G**



QUICK-SORT TREE

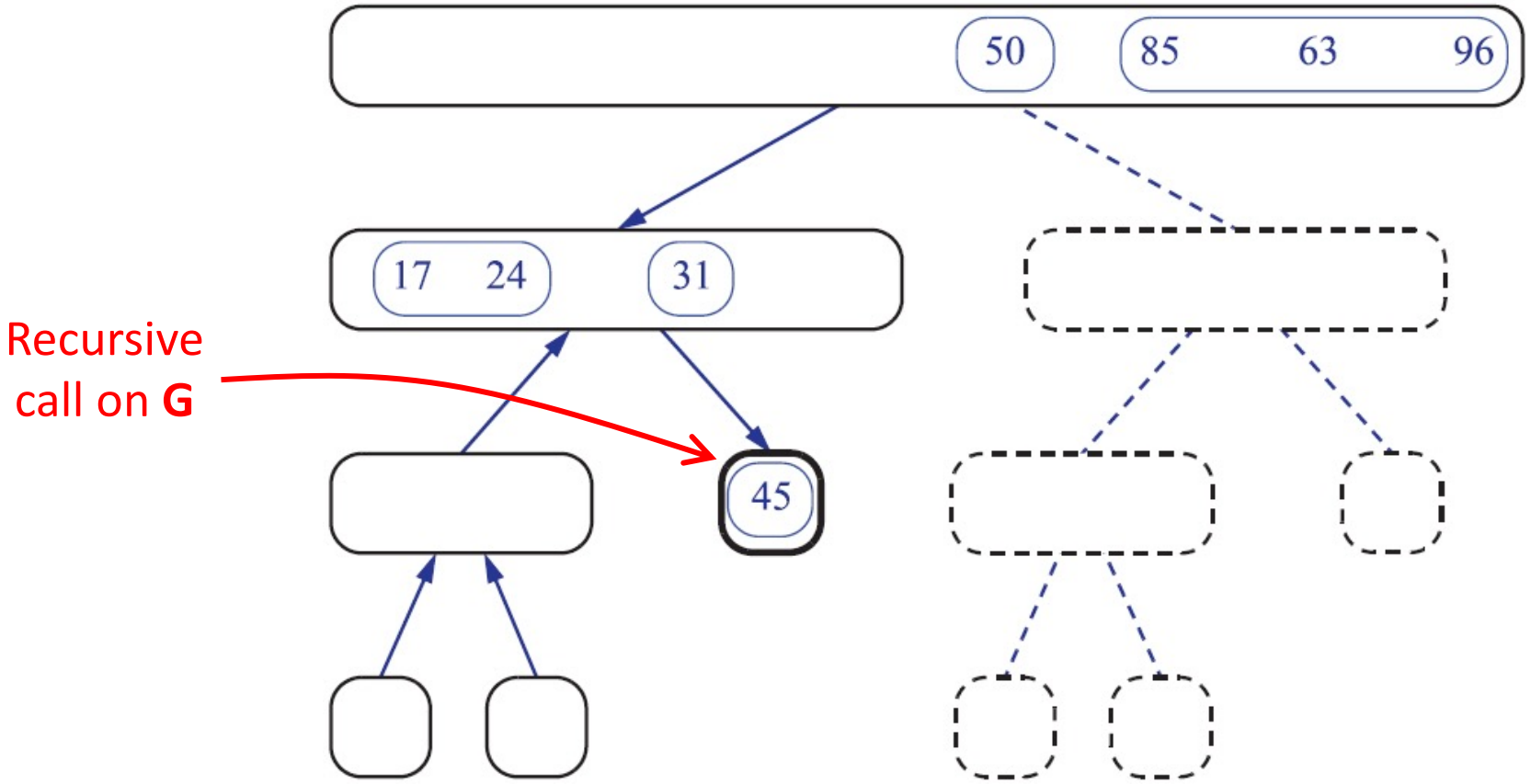
- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**

Now that **L** is sorted, we need to deal with **G**,



QUICK-SORT TREE

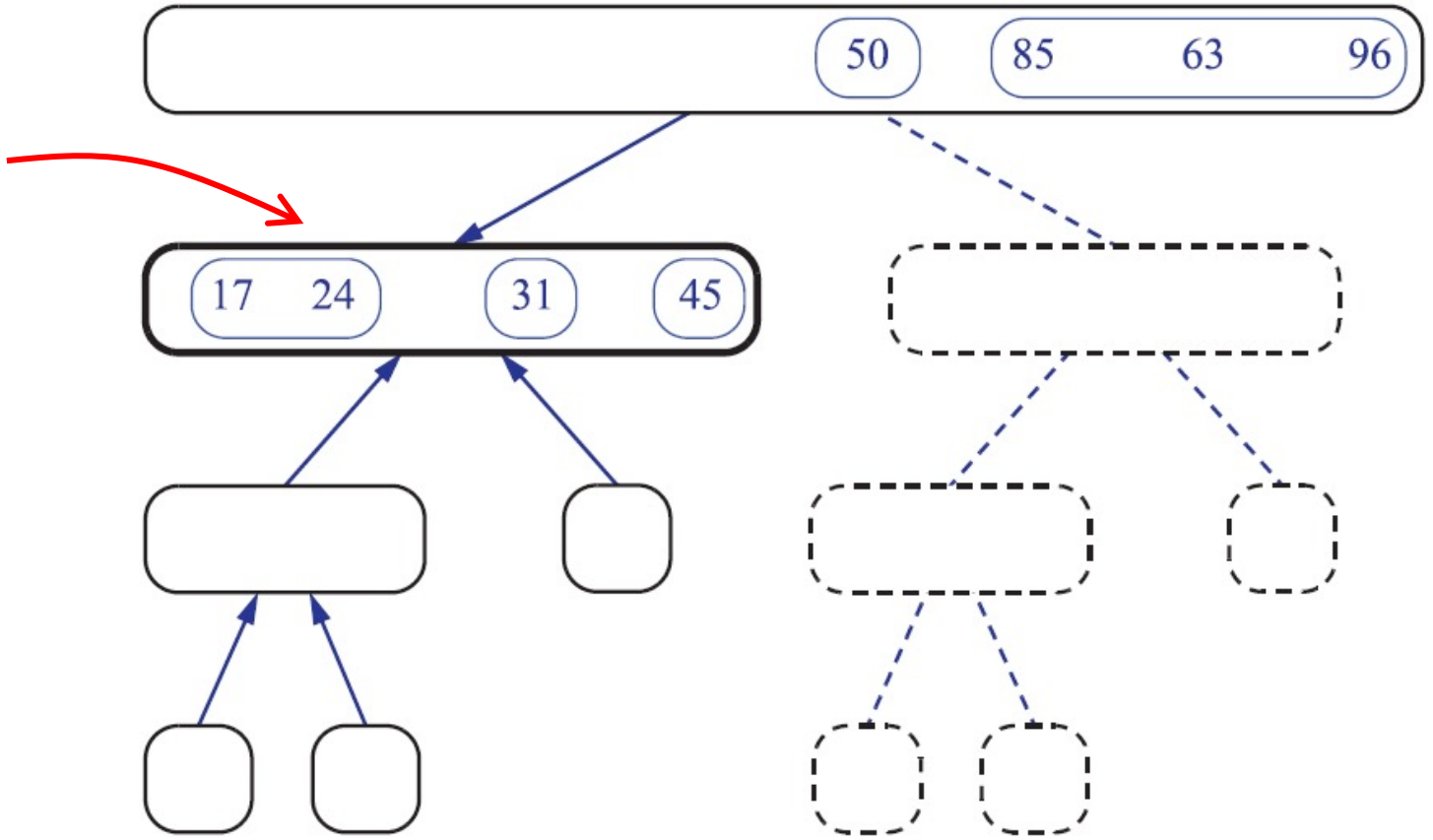
- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**



QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**

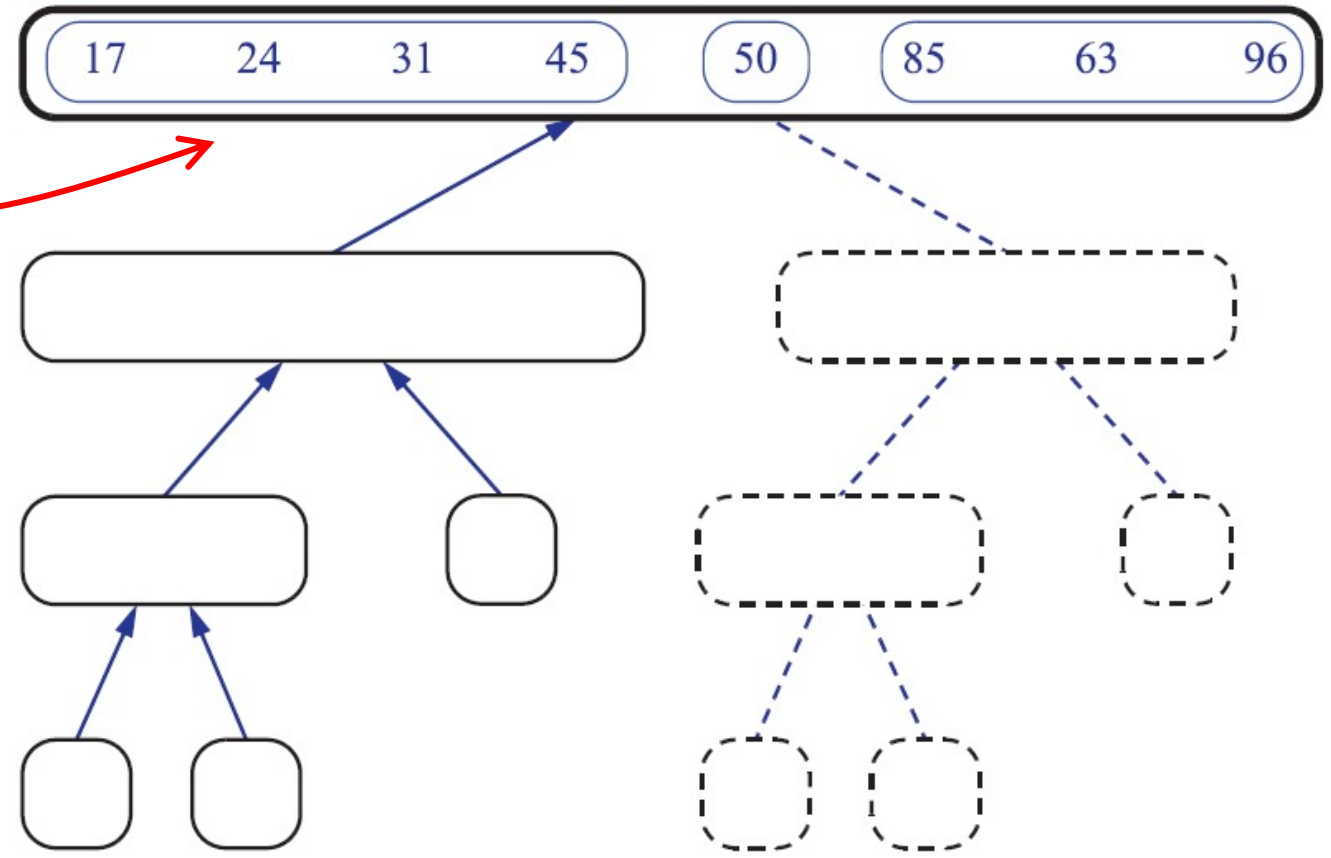
Merge them by:
- first putting **L**
- followed by **E**
- followed by **G**



QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**

Now that **L** is sorted, we need to deal with **G**,

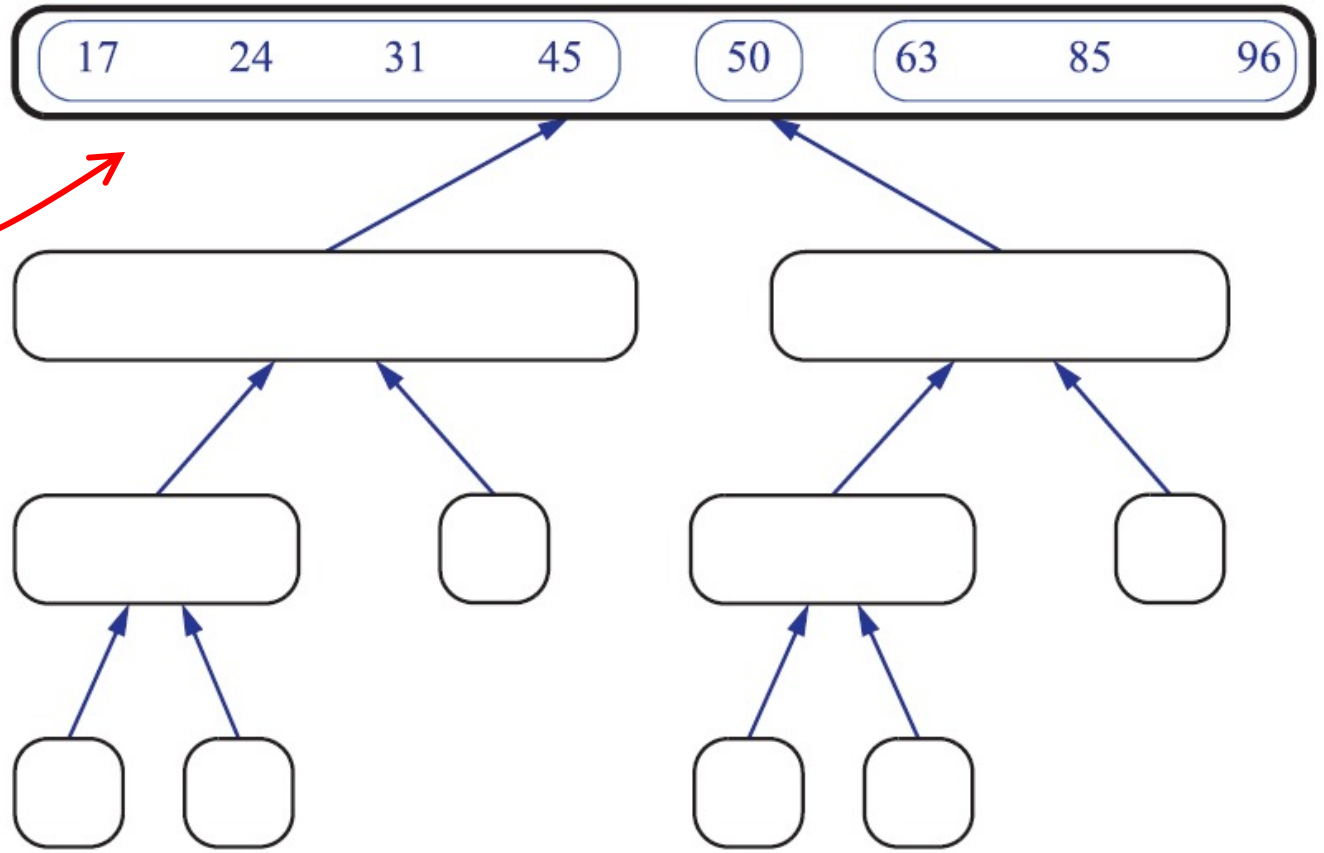


QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**

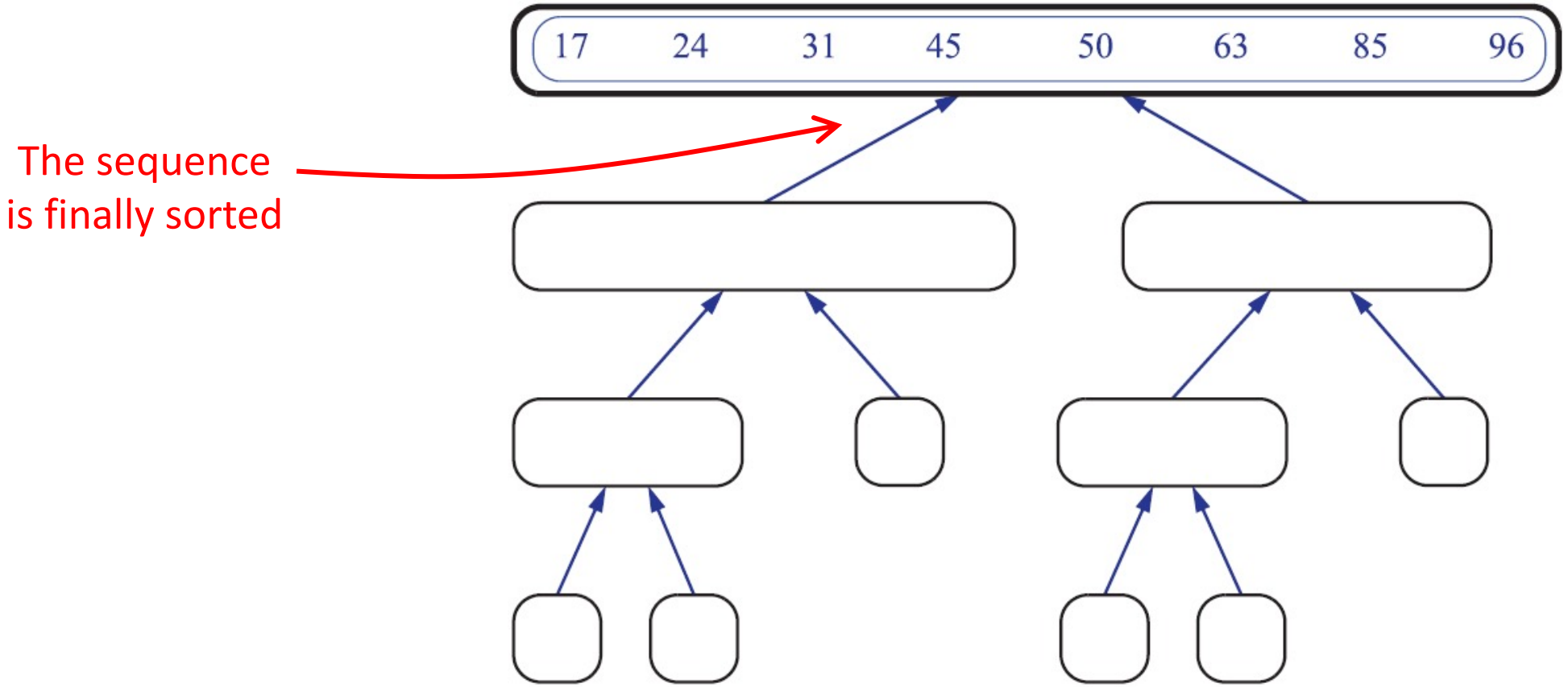
After many recursive calls, **G** is sorted, and now we're ready to merge them by:

- first putting **L**
- followed by **E**
- followed by **G**



QUICK-SORT TREE

- Here is a **more detailed** example showing that the first half, **L**, is sorted recursively before starting with the second half, **G**



```
Algorithm QuickSort(S):
  if S.size() ≤ 1 then
    return
  p ← S.back().element() {the pivot}
  Let L, E, and G be empty sequences
  while !S.empty() do
    if S.back().element() < p then
      L.insertBack( S.eraseBack() )
    else if S.back().element() = p then
      E.insertBack( S.eraseBack() )
    else
      G.insertBack( S.eraseBack() )
  QuickSort(L)
  QuickSort(G)
  while !L.empty() do
    S.insertBack( L.eraseFront() )
  while !E.empty() do
    S.insertBack( E.eraseFront() )
  while !G.empty() do
    S.insertBack( G.eraseFront() )
  return
```

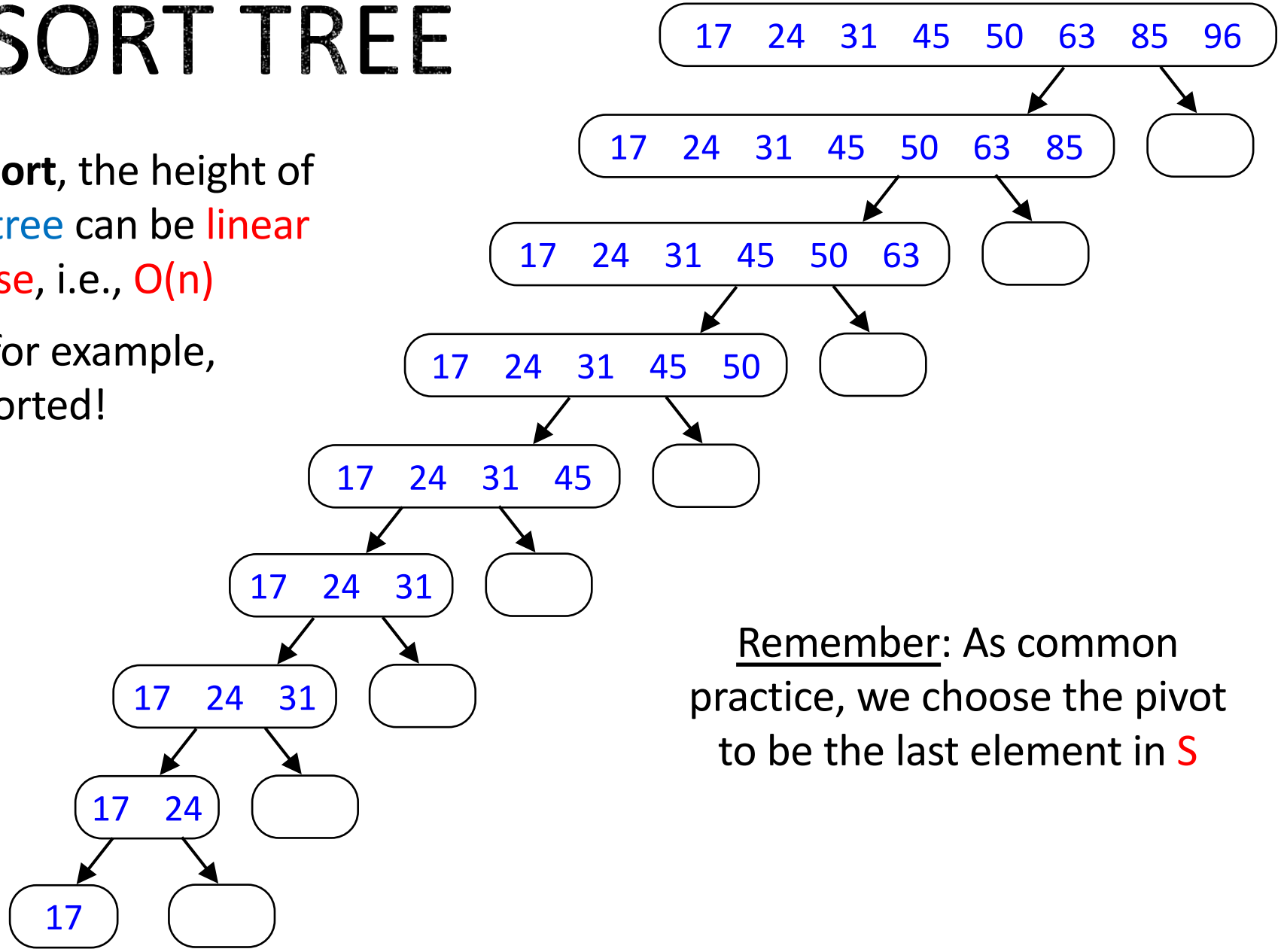
- Scan S starting **from the back.** For each element, remove it from S and put it in **the back** of L, E or G
- **We insert/remove elements from the back** since this can be done efficiently regardless of whether we are dealing with arrays or lists!

- Recursive calls on L and G

- Scan L starting **from the front**. For each element, remove it from L and put it in **the back** of S
- Do the same for E and G
- **We insert elements from the back** since this can be done efficiently regardless of whether we are dealing with arrays or lists!
- But removal from front is inefficient for arrays! ☹ This inefficiency can be avoided by deleting the array after copying all its elements to S

QUICK-SORT TREE

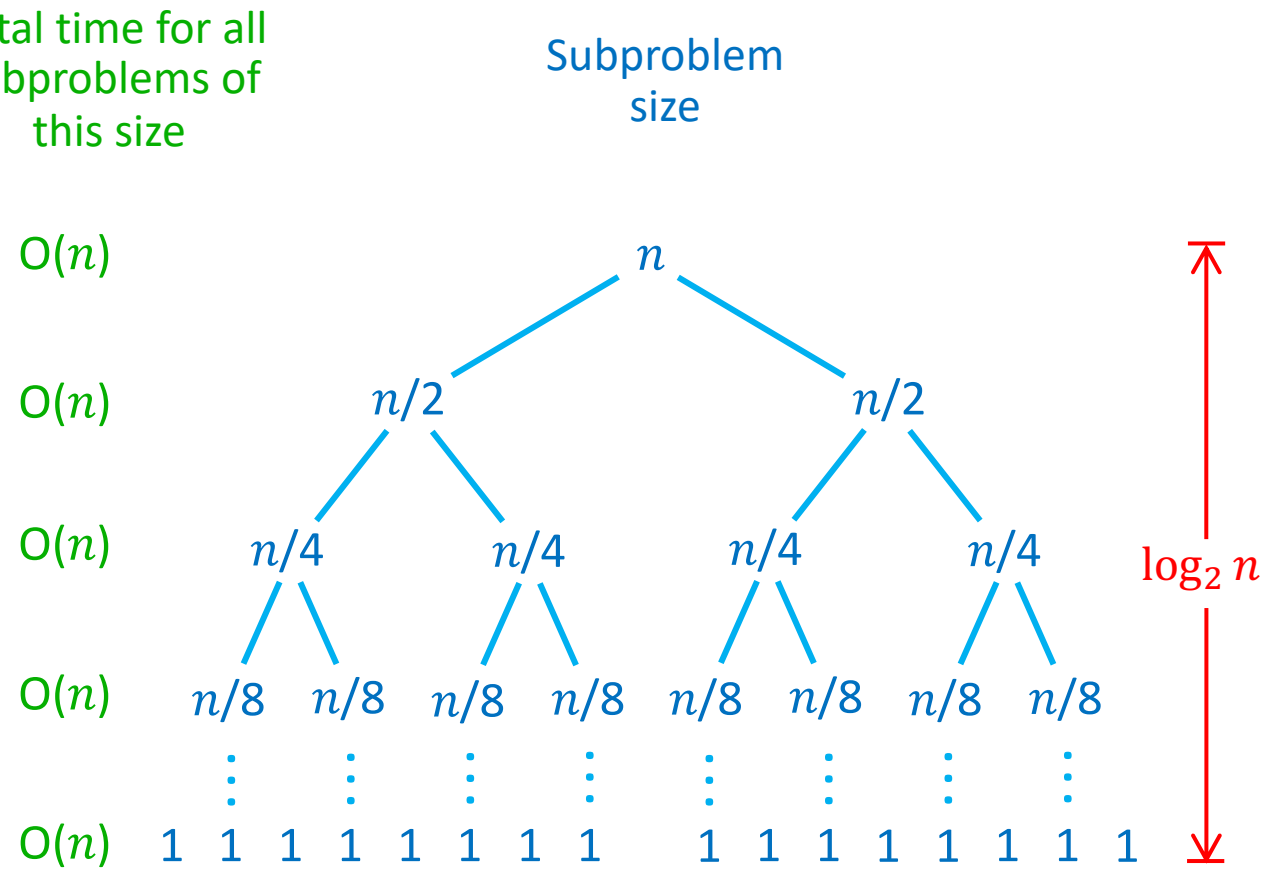
- Unlike **merge-sort**, the height of the quick-sort tree can be linear in the worst case, i.e., O(n)
- This happens, for example, if S is already sorted!



Remember: As common practice, we choose the pivot to be the last element in S

BEST-CASE RUNNING TIME

- The **best case** is if we always pick a pivot that splits the sequence into equal halves!
- Quick-sort runs in $O(n \log n)$ time in the **best case!**
- Moreover, if we get a **good split** 50% of the time, and a **bad split** 50% of the time, we end up with a version of quick-sort that runs in $O(n \log n)$ too!



WORST-CASE RUNNING TIME

- The **worst case** is if we always pick a pivot that splits the sequence into two parts, one of which is empty!
- In the **worst case**, the number of operations performed is:
  $n + (n-1) + \cdots + 2 + 1$
- Thus, quick-sort runs in $O(n^2)$ time in the **worst case!**