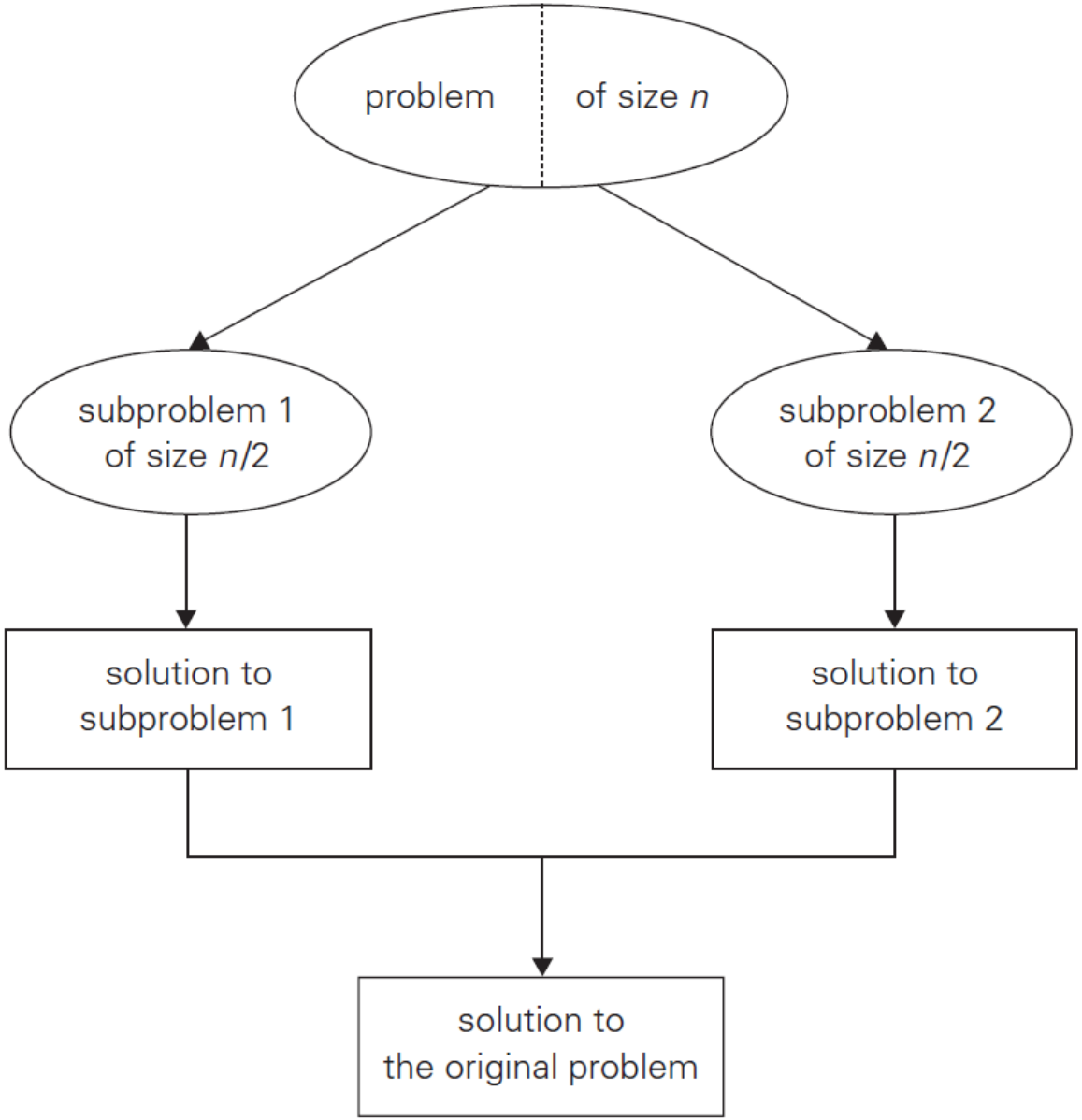
Divide and Conquer

(Chapter 5)

This week:

- Divide and Conquer technique
- Count a specific key in an array
- Master theorem
- Merge sort
- Binary tree
  - Computing the height
  - Compute the number of leaves

Divide and Conquer technique

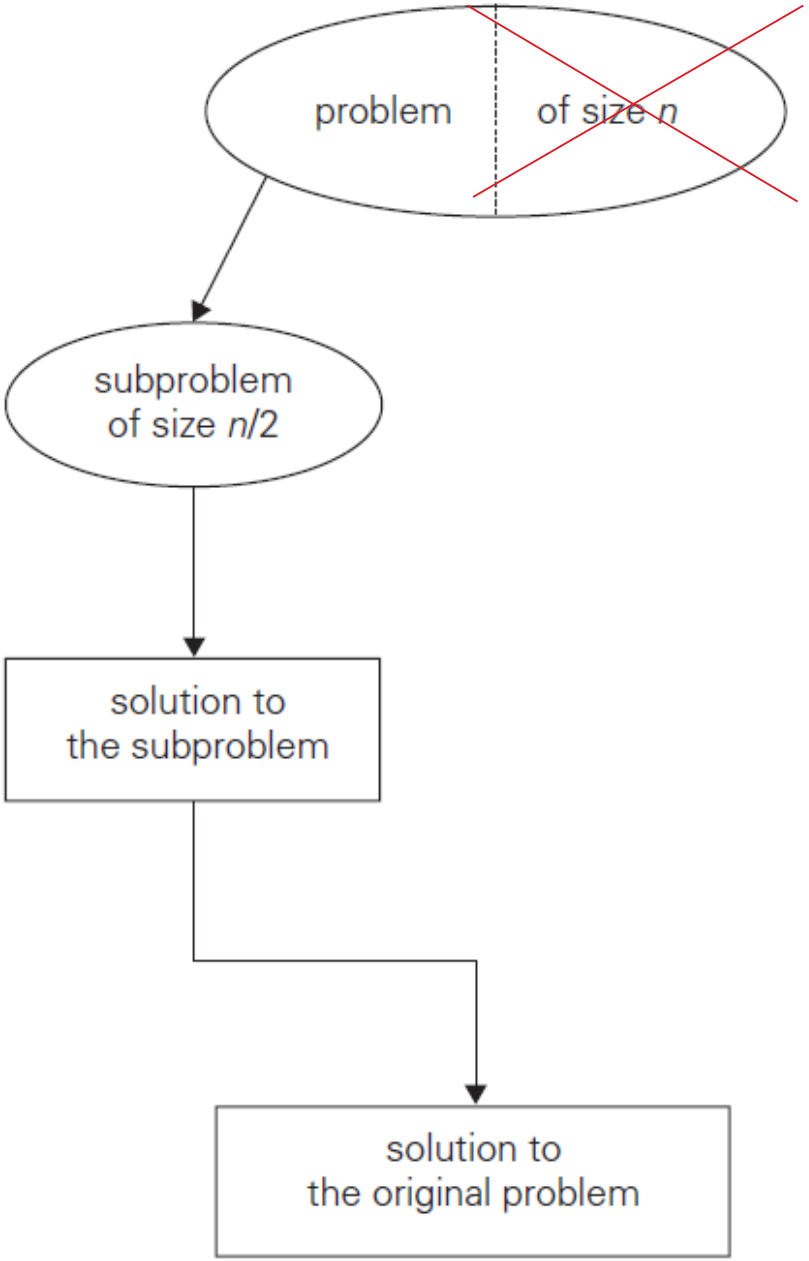


Divide and Conquer technique

A well known algorithm design technique:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances (usually recursively)
3. Obtain solution to original (larger) instance by combining these solutions

Decrease and Conquer (last week)



A Natural Question

- How is this different from decrease and conquer technique
- Think of the fake coin problem:
  - We discarded half the coins at each step
  - So we didn't do any work on those "sub problems"
- For divide and conquer…
  - You need to solve all of the sub problems

This week:

- Divide and Conquer technique
- Count a specific key in an array
- Master theorem
- Merge sort
- Binary tree
  - Computing the height
  - Compute the number of leaves

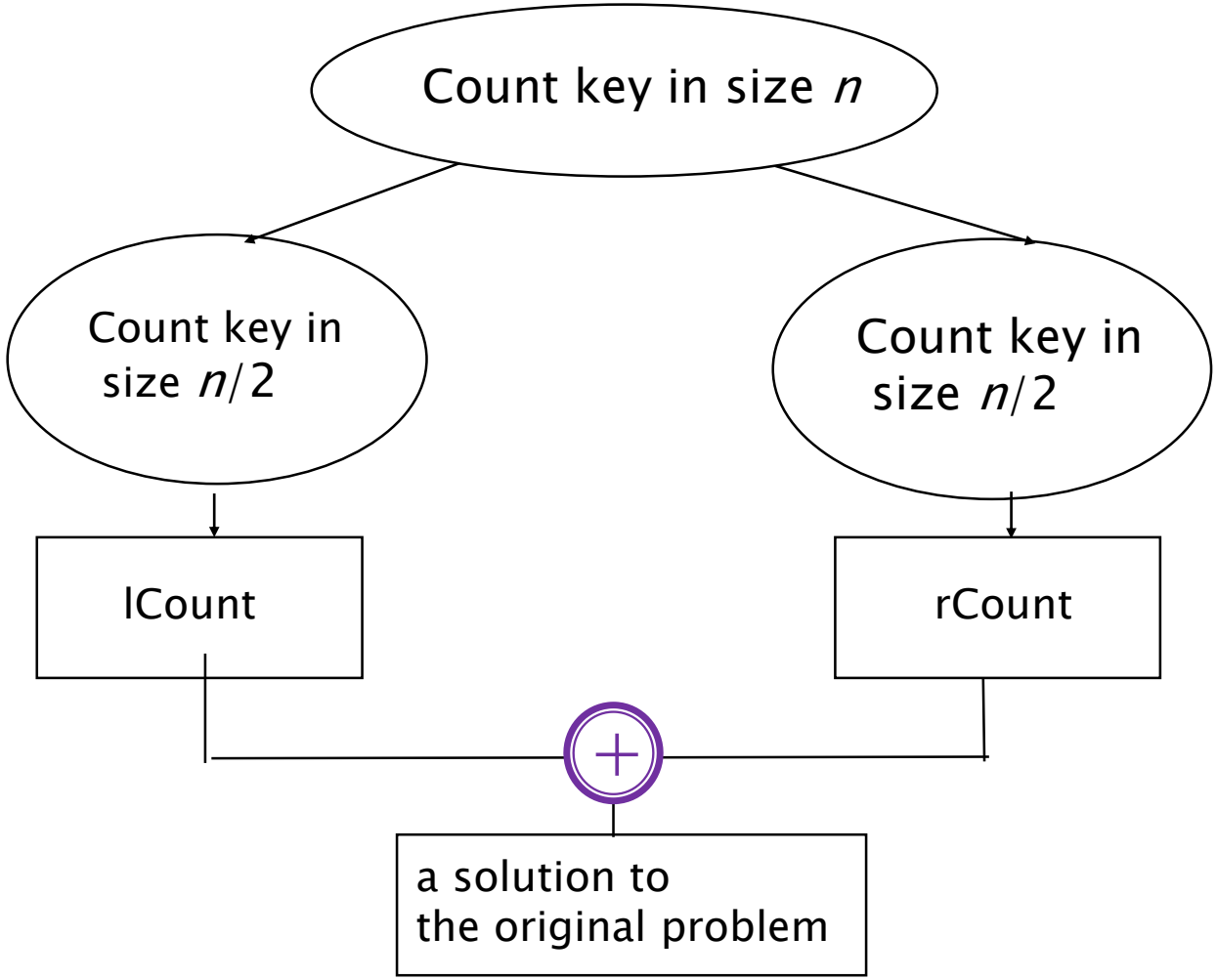
Count a specific key in an array

Problem:
Count the number of times a specific key occurs in an array.

For example:
If input array is A=[2,7,6,6,2,4,6,9,2] and key=6…
… should return the value 3.

Design an algorithm using divide and conquer technique

Count a specific key in an array



Count a specific key in an array

Algorithm CountKey(A[], L, R, Key)

```
//Input: A[] is an array A[0..n-1] from indices l and r (L ≤ R)
//Output: A count of the number of time Key exists in A[L..R]

1.   if L = R
2.      if (A[L] = Key) return 1
3.      else return 0
4.   else
5.      lCount = CountKey(A[],       L,      ⌊(L+R)/2⌋, Key)
6.      rCount = CountKey(A[], ⌊(L+R)/2⌋+1,     R,      Key)
7.      return lCount + rCount
```

Count a specific key in an array

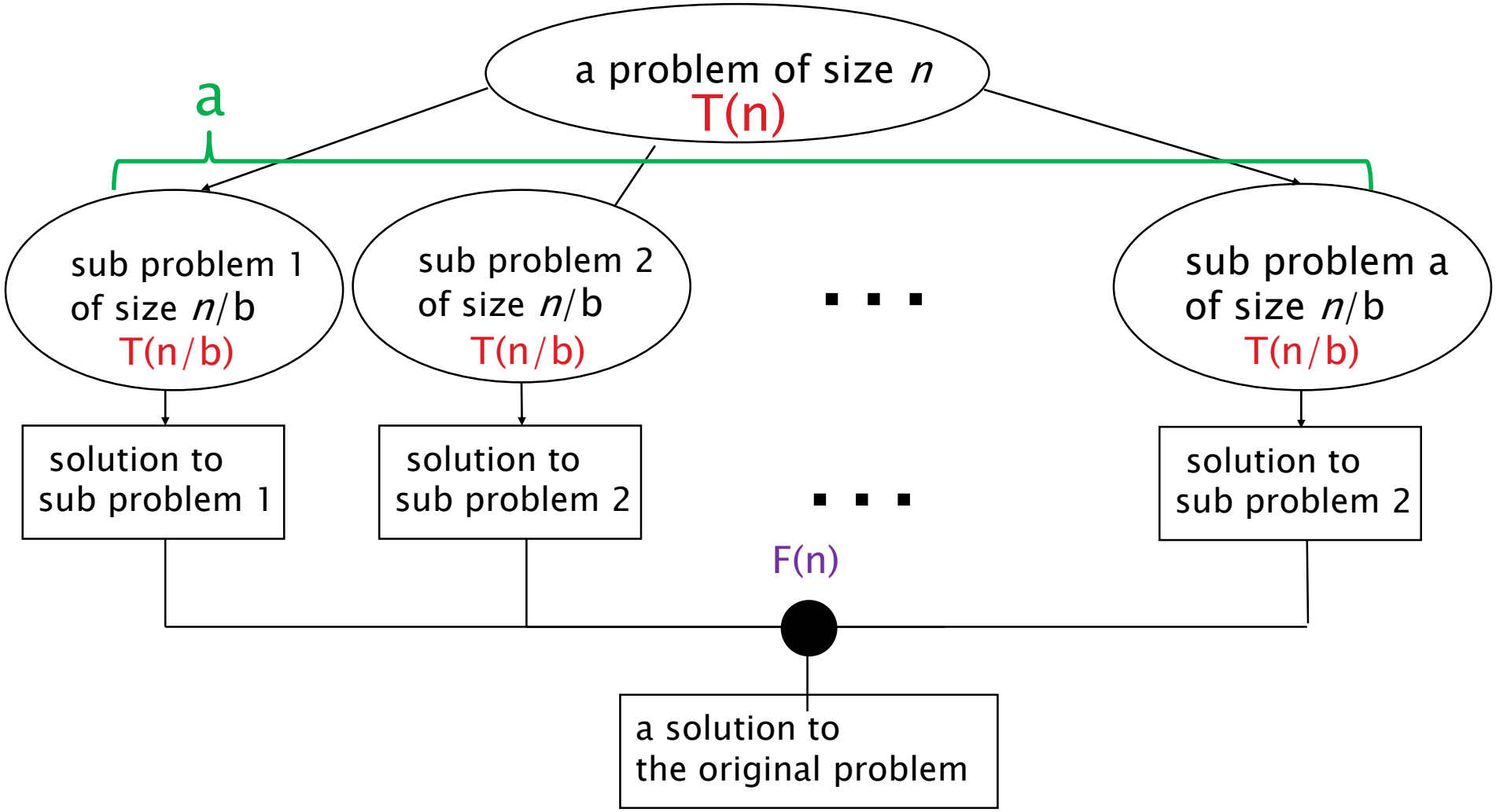
- *CountKey* looks familiar…
- What's the difference between *Binary Search and CountKey?*

- We have to search both sides
  - In the counter, both sides must be searched
  - In Binary Search, one half gets ignored

This week:

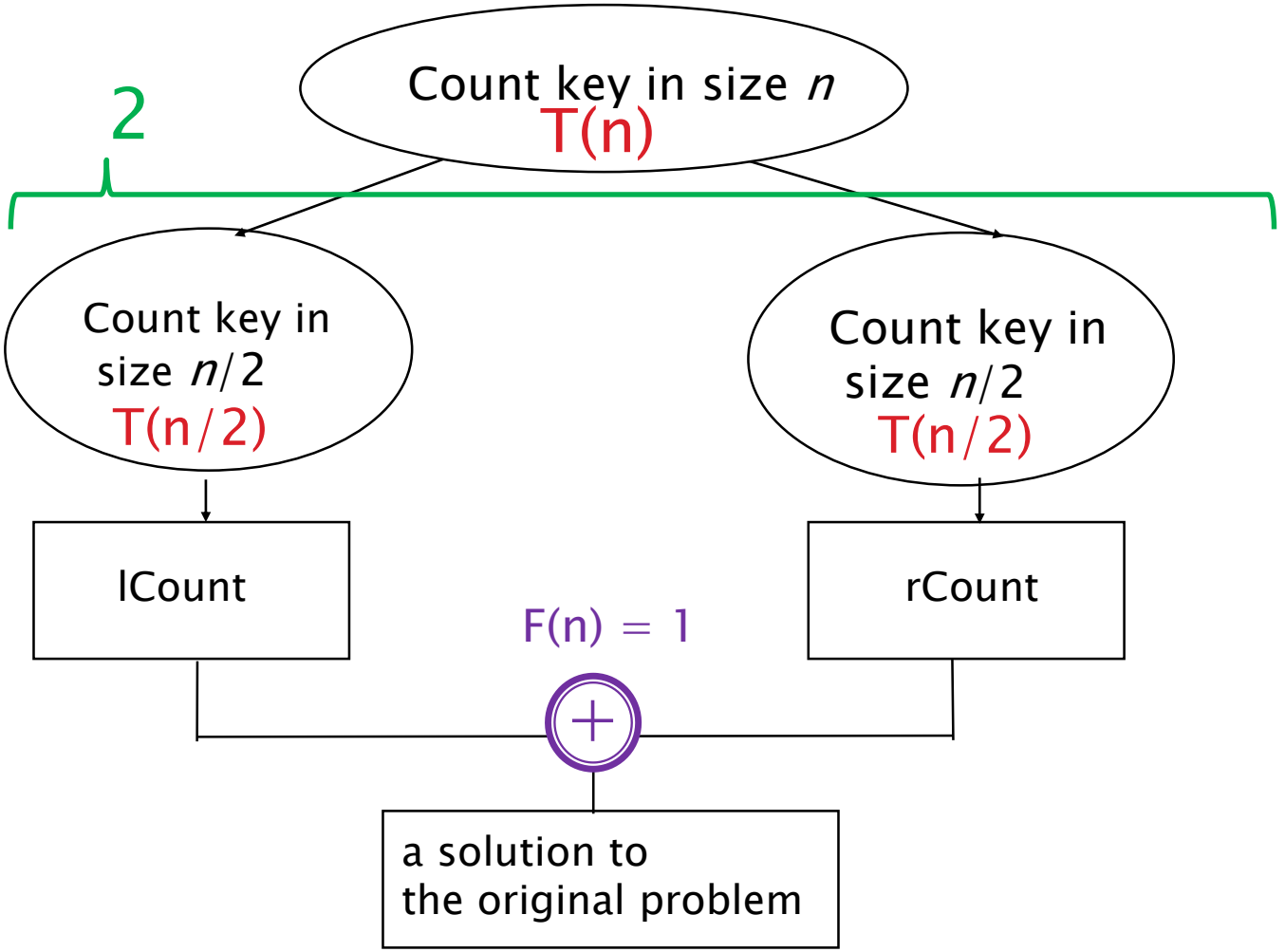
- Divide and Conquer technique
- Count a specific key in an array
- Master theorem
- Merge sort
- Binary tree
  - Computing the height
  - Compute the number of leaves

Analysis of a divide and conquer algorithm



$$T(n) = a\,T(n/b) + F(n)$$

Analysis of Count a specific key in an array



$$T(n) = 2\,T(n/2) + 1$$

Master theorem

$T(n) = a\,T(n/b) + F(n)$

1) If $n^{\log_b a} < F(n)$, $T(n) \in O(F(n))$
2) If $n^{\log_b a} > F(n)$, $T(n) \in O(n^{\log_b a})$
3) If $n^{\log_b a} = F(n)$, $T(n) \in O(n^{\log_b a} \log n)$

Example 1: $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

$a = 4$, $b = 2$ → $n^{\log_b a}$ → $n^{\log_2 4}$ → n^2

$F(n) = n^3$

→ $T(n) \in O(n^3)$

Master theorem

Example 2: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

$a = 4$, $b = 2$ → $n^{\log_b a}$ → $n^{\log_2 4}$ → n^2

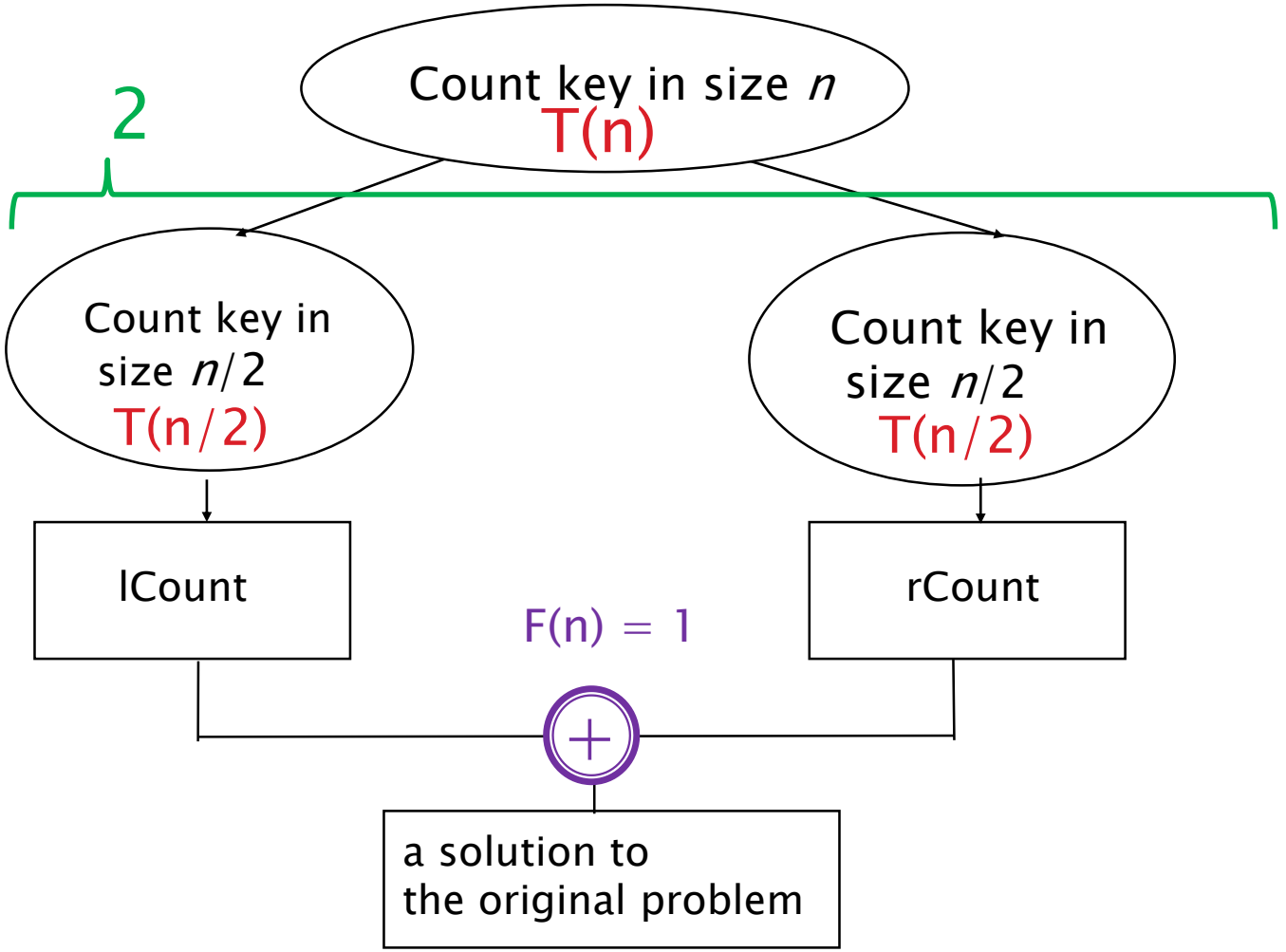
$F(n) = n$ → $T(n) \in O(n^2)$

Example 3: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$a = 4$, $b = 2$ → $n^{\log_b a}$ → $n^{\log_2 4}$ → n^2

$F(n) = n^2$ → $T(n) \in O(n^2 \log n)$

Analysis of count a specific key in an array

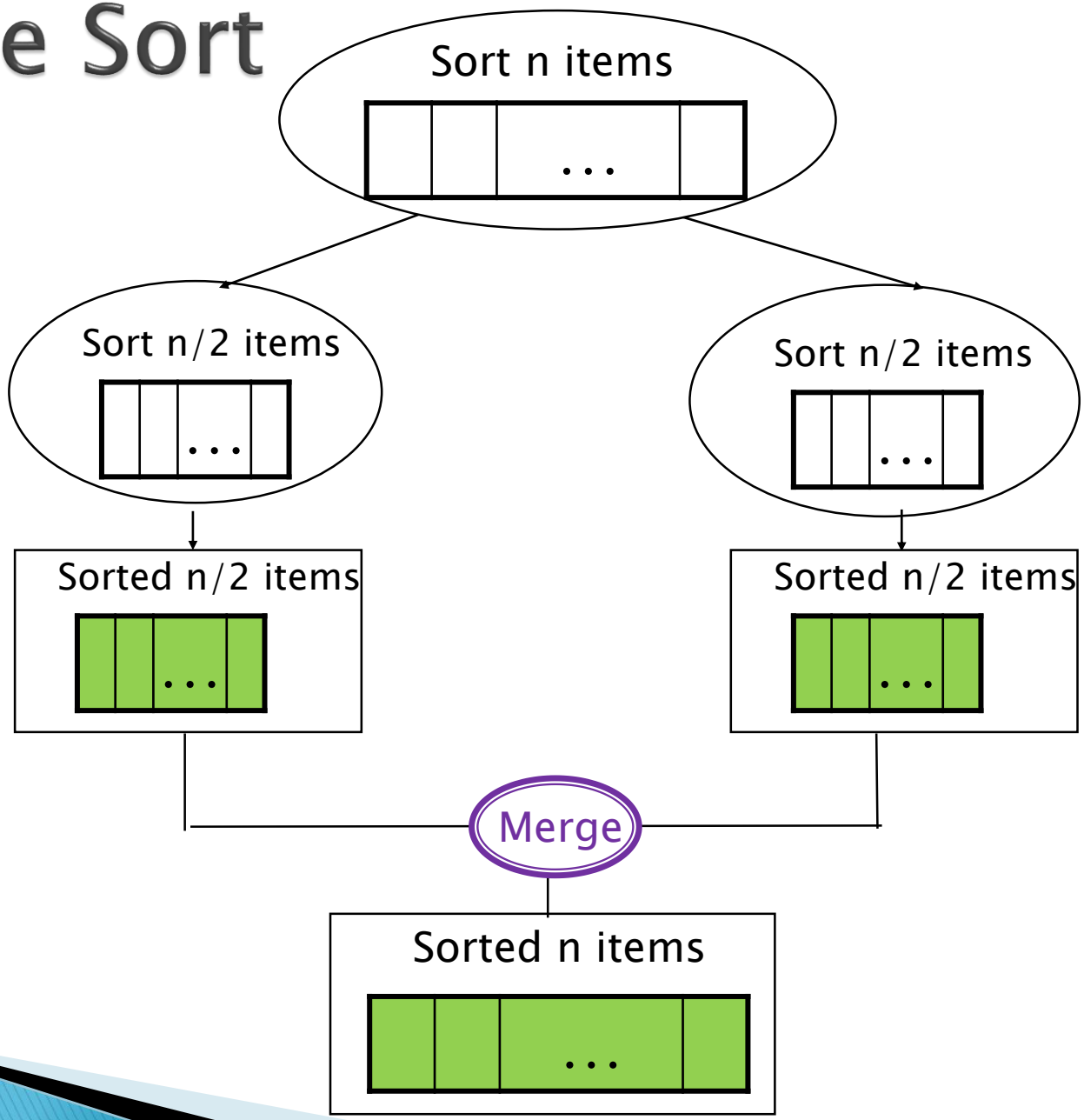


$$T(n) = 2\,T(n/2) + 1 \Rightarrow T(n) \in O(n)$$

This week:

- Divide and Conquer technique
- Count a specific key in an array
- Master theorem
- Merge sort
- Binary tree
  - Computing the height
  - Compute the number of leaves

Merge Sort


Sort n items
Sort n/2 items
Sort n/2 items
Sorted n/2 items
Sorted n/2 items
Merge
Sorted n items

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

```
//Sorts array A[0..n − 1] by recursive mergesort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
if n > 1
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B[0..⌊n/2⌋ − 1])
    Mergesort(C[0..⌈n/2⌉ − 1])
    Merge(B, C, A)
```

Merge Sort

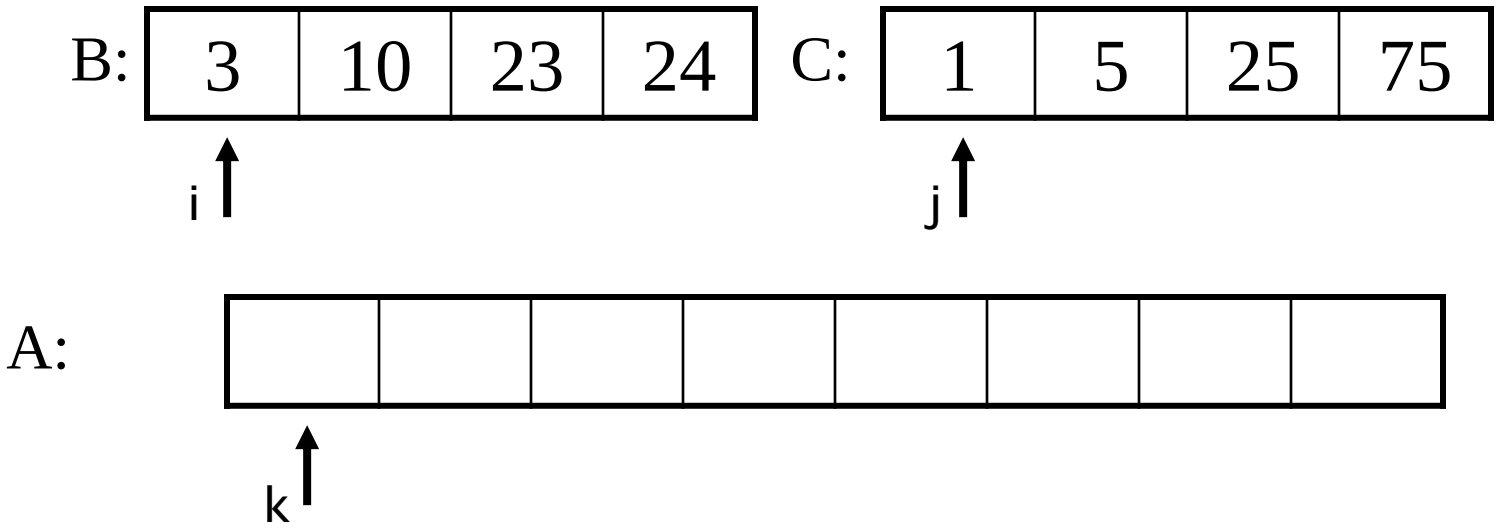
- Important part in merge sort is to merge two sorted array into one

- Example:

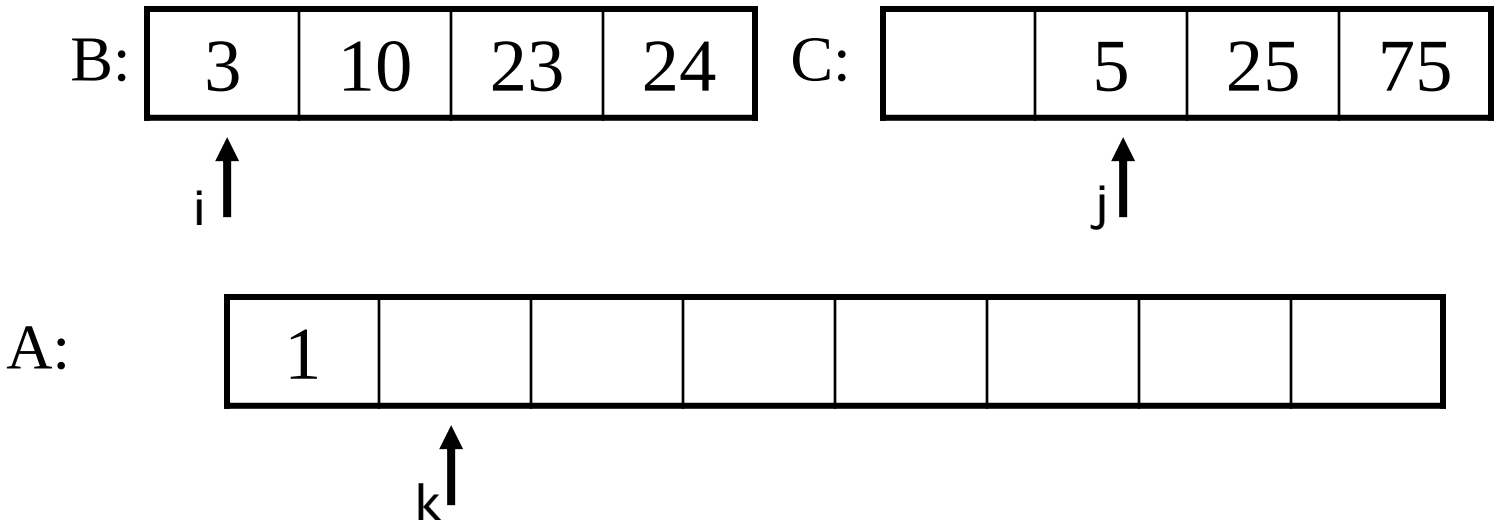
B = { 3 8 9 } C = { 1 5 7 }

merge(B, C) = { 1 3 5 7 8 9 }

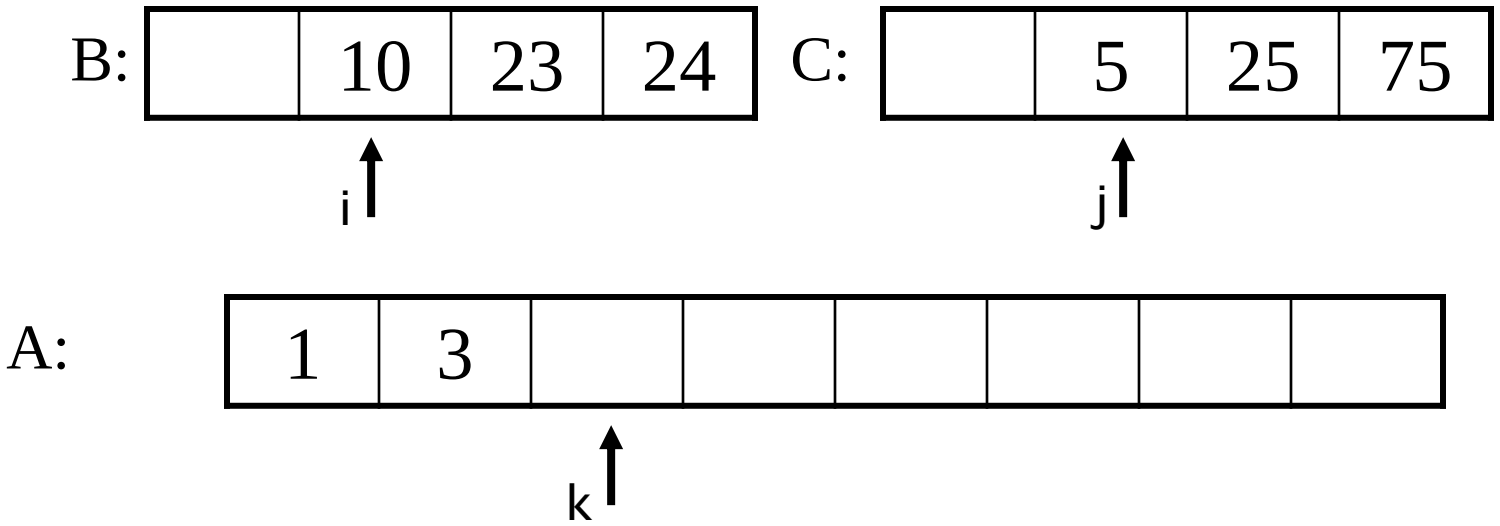
Merging (cont.)



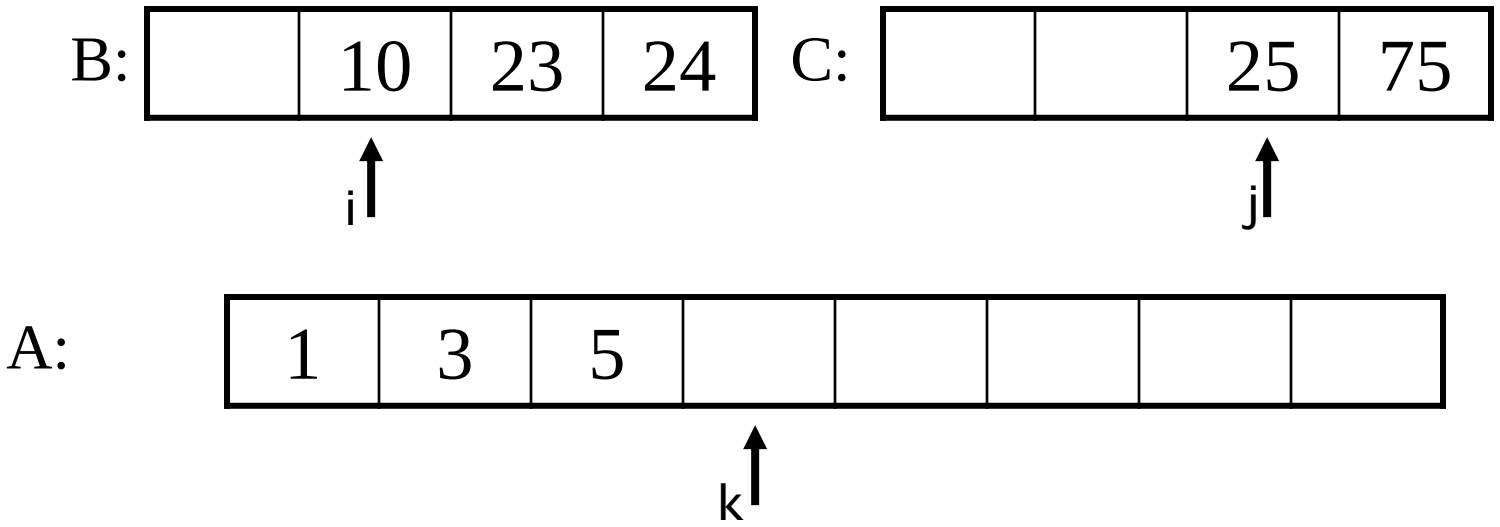
Merging (cont.)



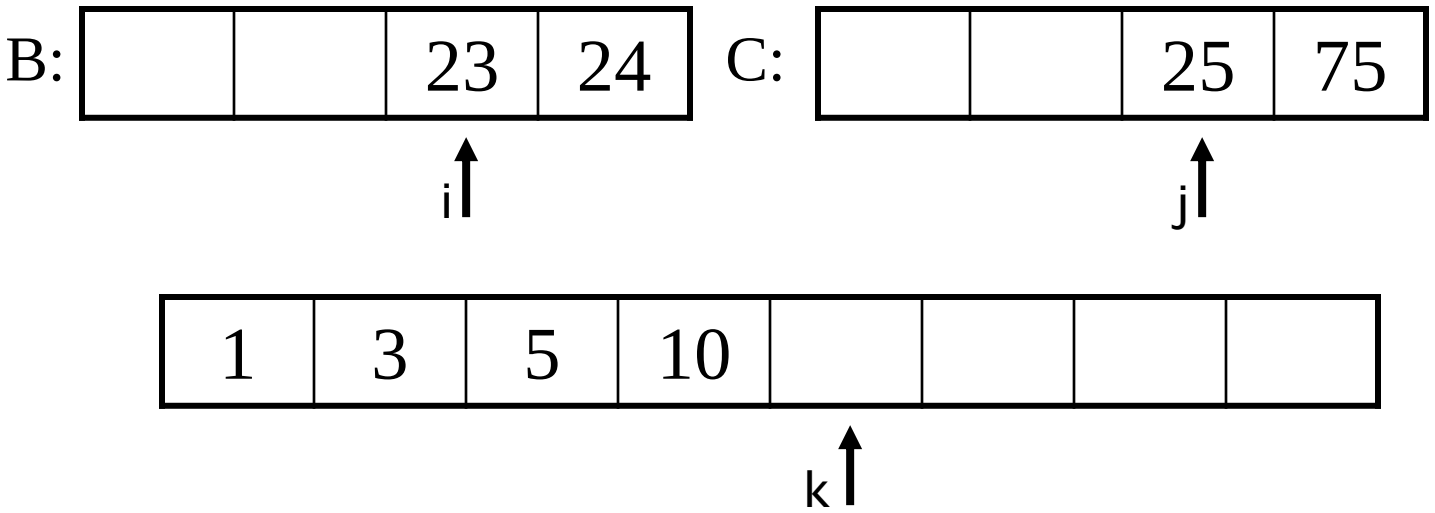
Merging (cont.)



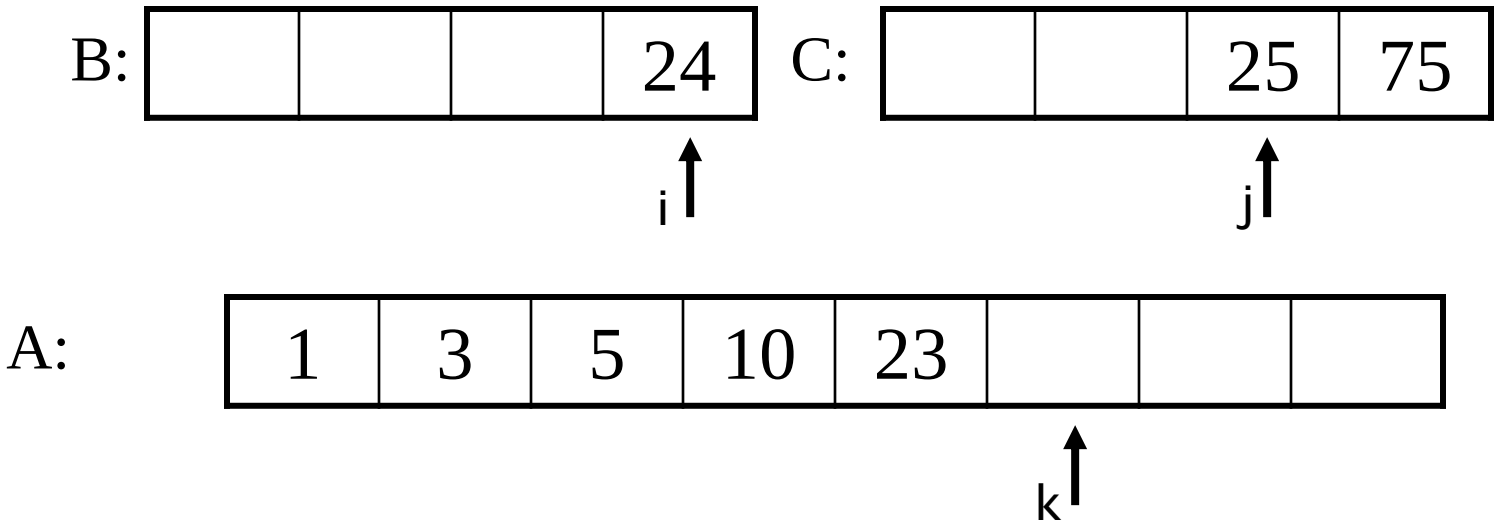
Merging (cont.)



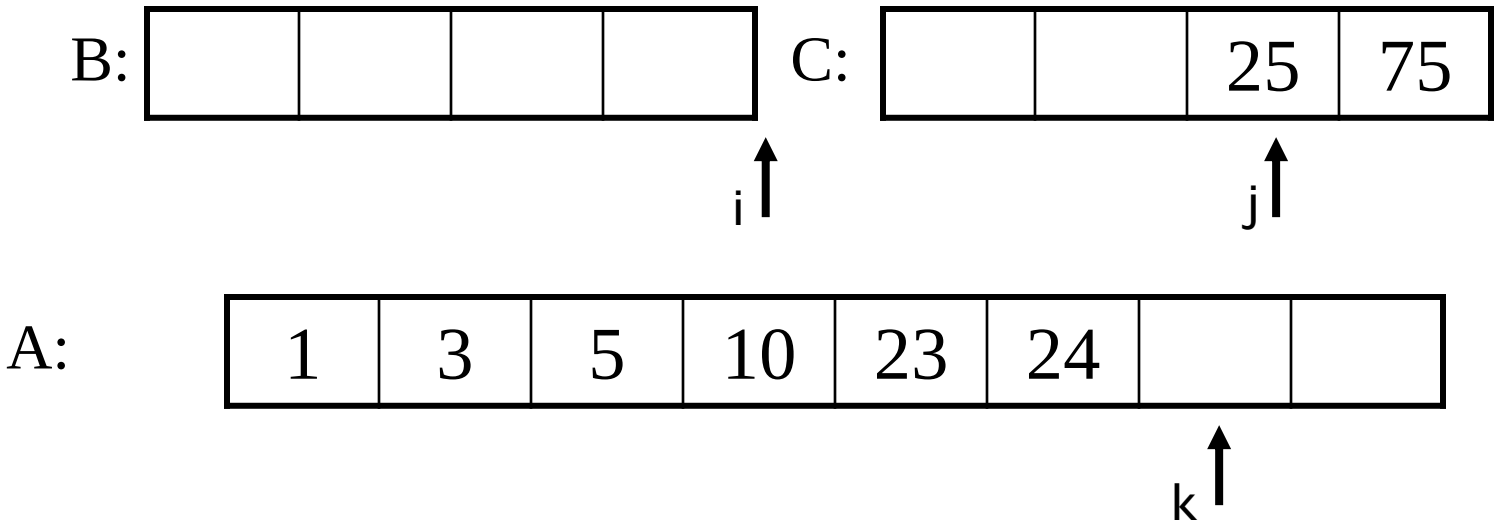
Merging (cont.)



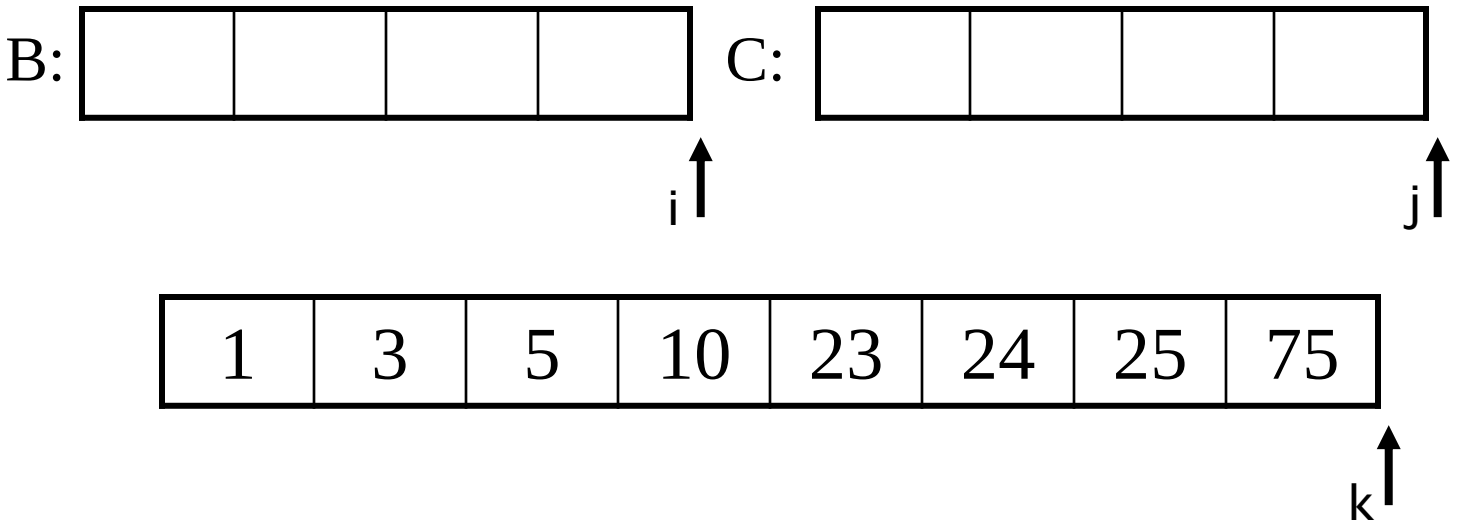
Merging (cont.)



Merging (cont.)



Merging (cont.)



Merging (cont.)

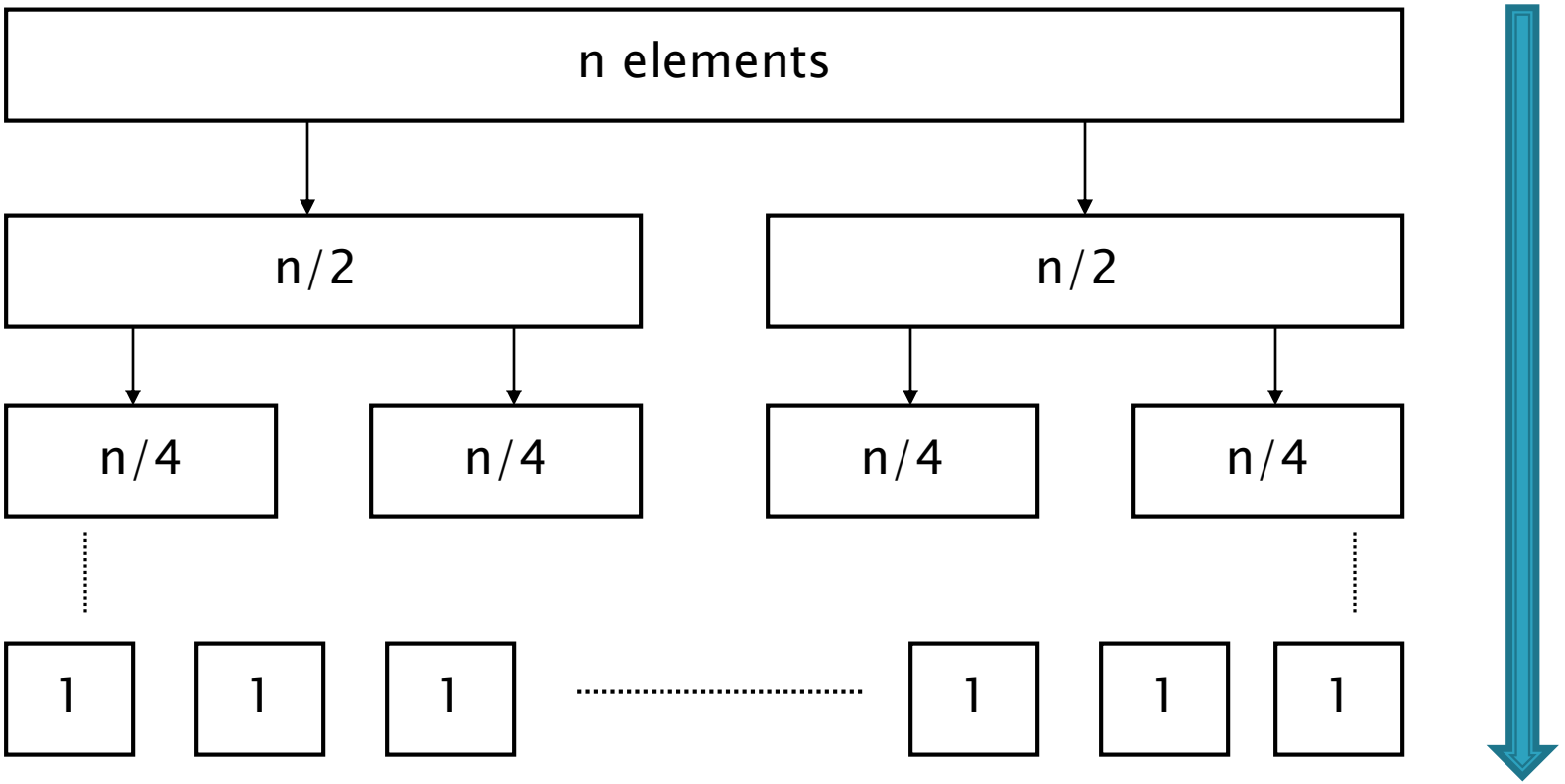
- Must put the next-smallest element into the merged list at each point
- Each next-smallest could come from either list

Pseudocode of Merge

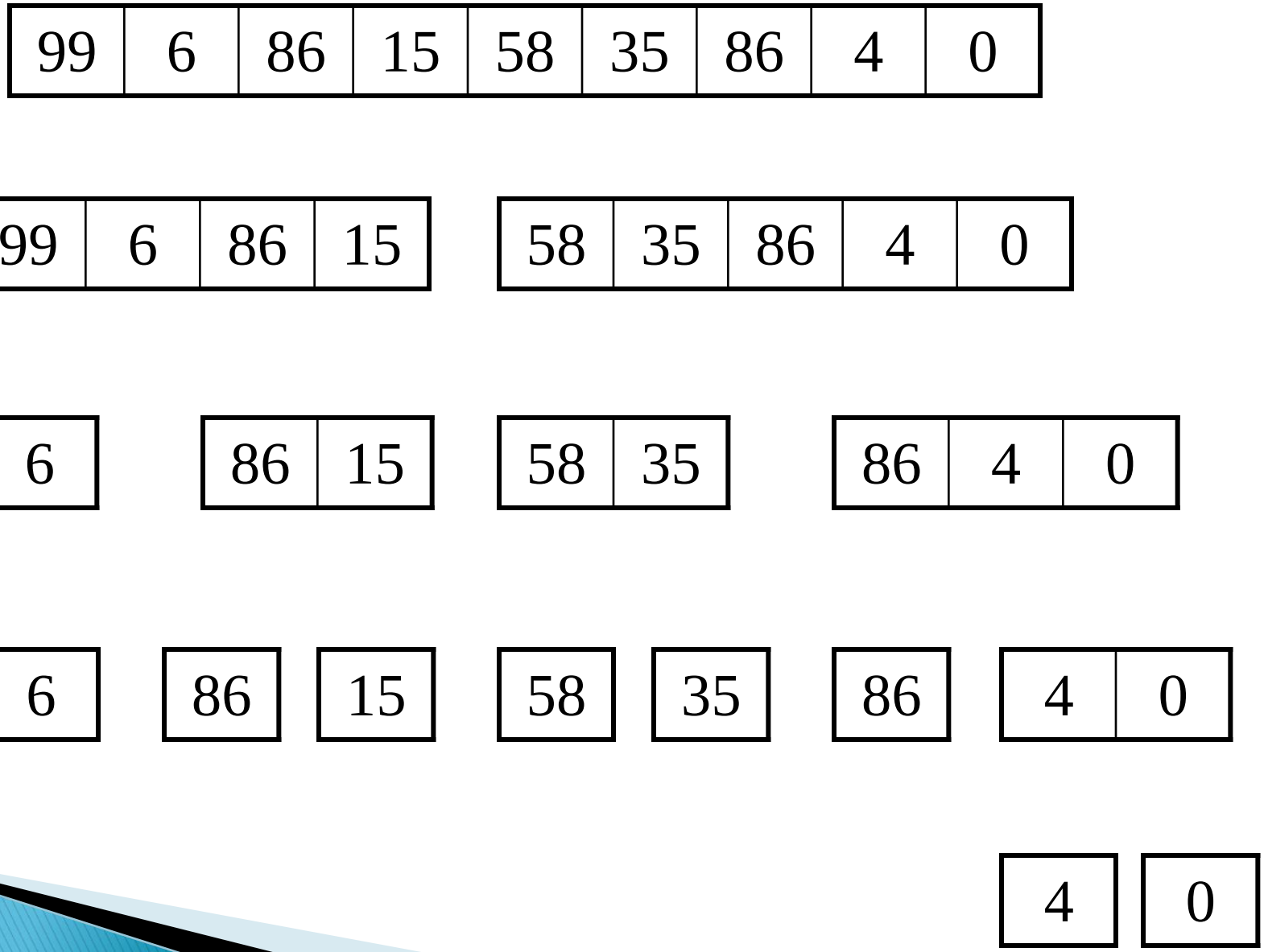
ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i+1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

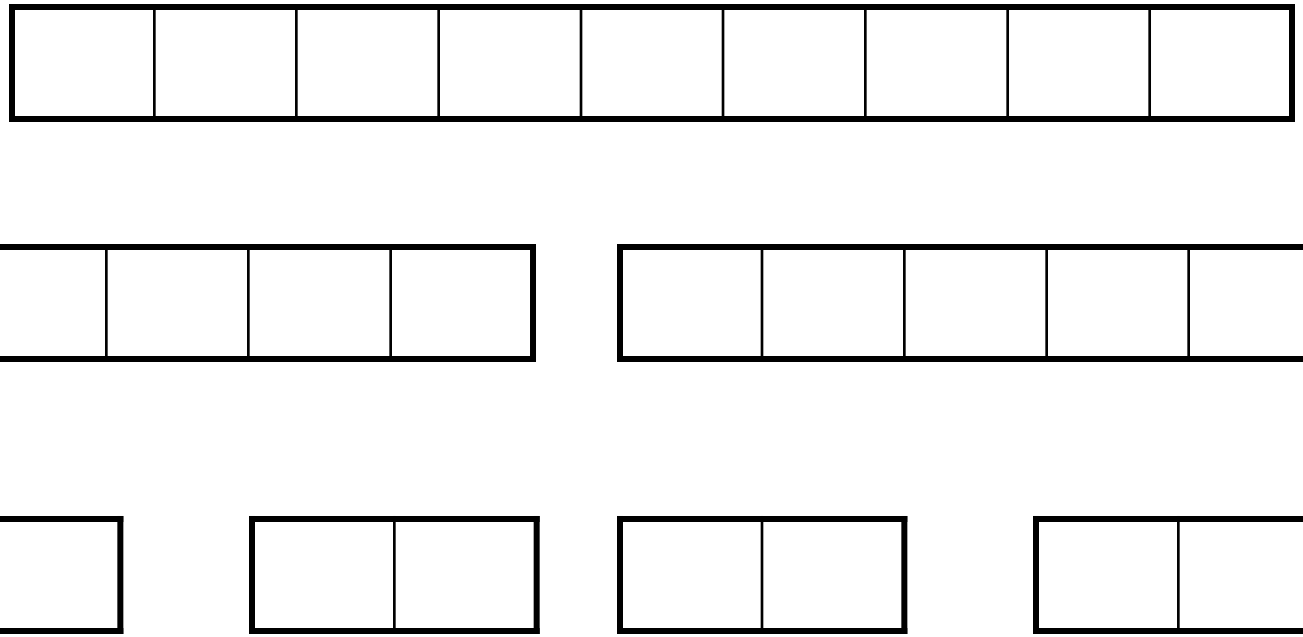
Merge Sort



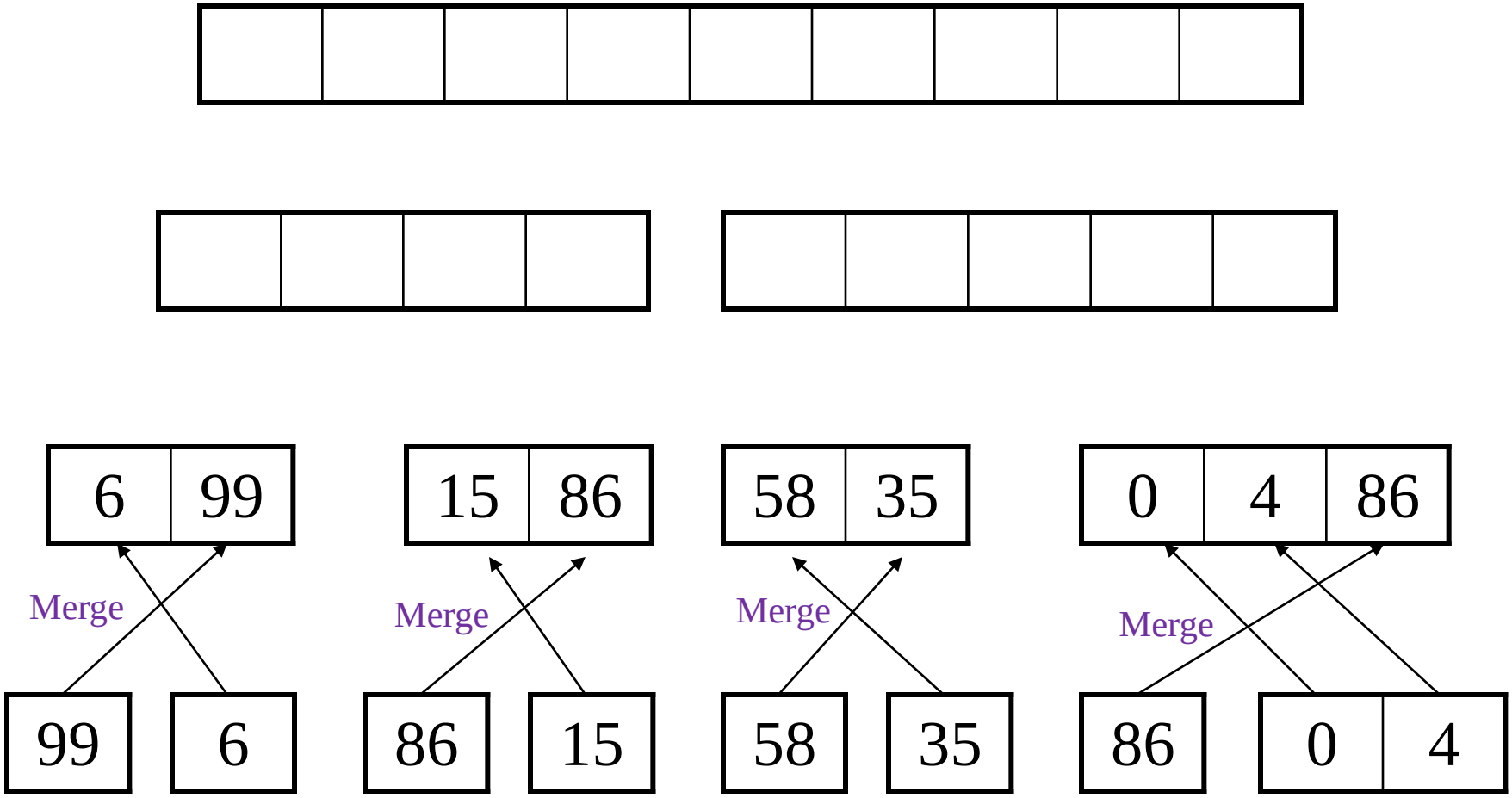
Merge Sort Example



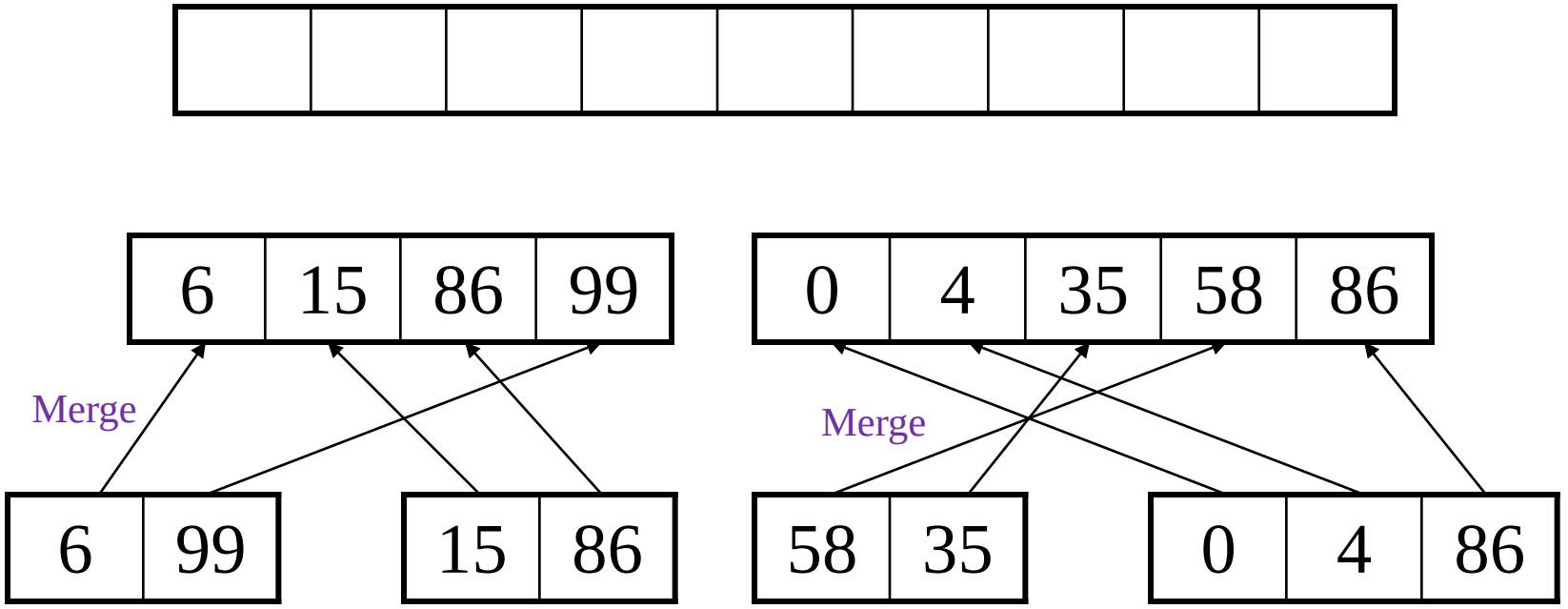
Merge Sort Example



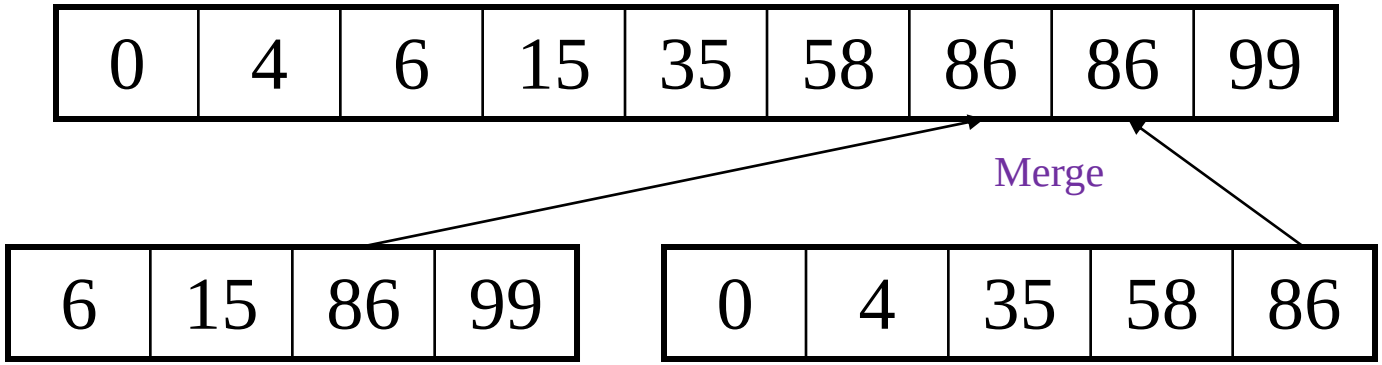
Merge Sort Example



Merge Sort Example



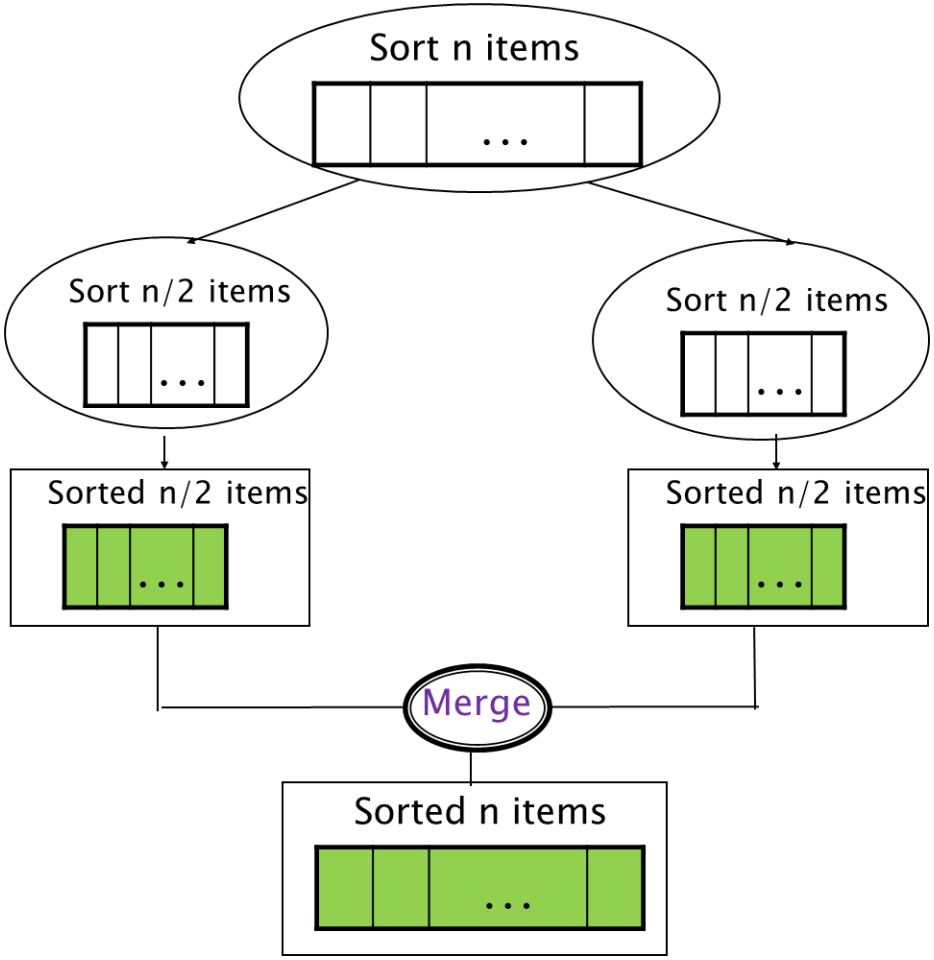
Merge Sort Example



Merge Sort Example

0	4	6	15	35	58	86	86	99

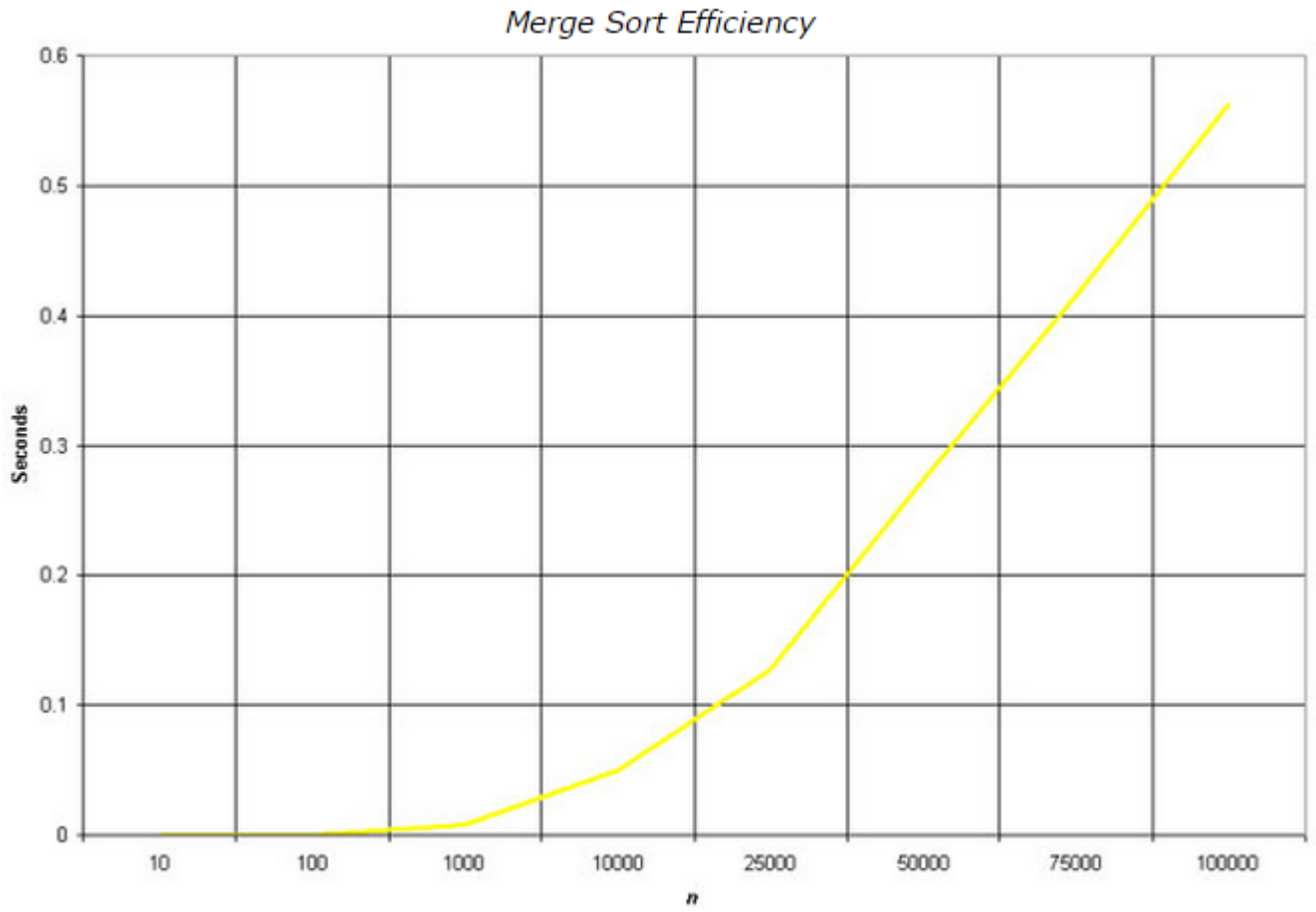
Running Time



$$T(n) = 2\,T(n/2) + n$$

$$T(n) \in O(n \log n)$$

Merge sort


Merge Sort Efficiency
Seconds
0.6
0.5
0.4
0.3
0.2
0.1
0
10
100
1000
10000
25000
50000
75000
100000
n

Merge sort properties

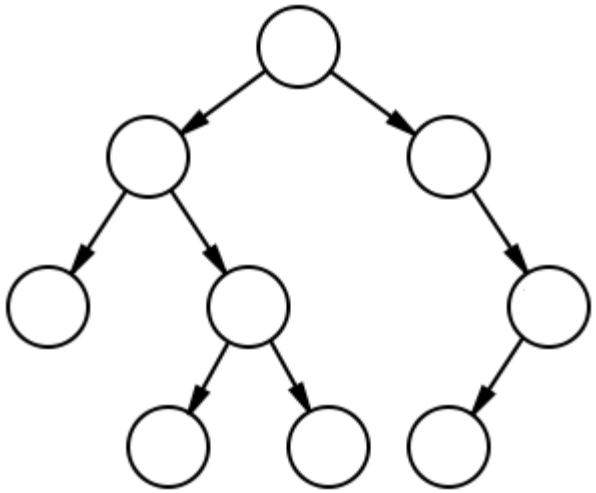
- Is not “in-place”
  - requires extra storage
  - an extra array with n elements
- Is “stable”
  - equal elements are kept in order

This week:

- Divide and Conquer technique
- Count a specific key in an array
- Master theorem
- Merge sort
- Binary tree
  - Computing the height
  - Compute the number of leaves

Tree structure

- Some real world example:
  - XML/Markup parsers
  - Computer games
  - Social networking

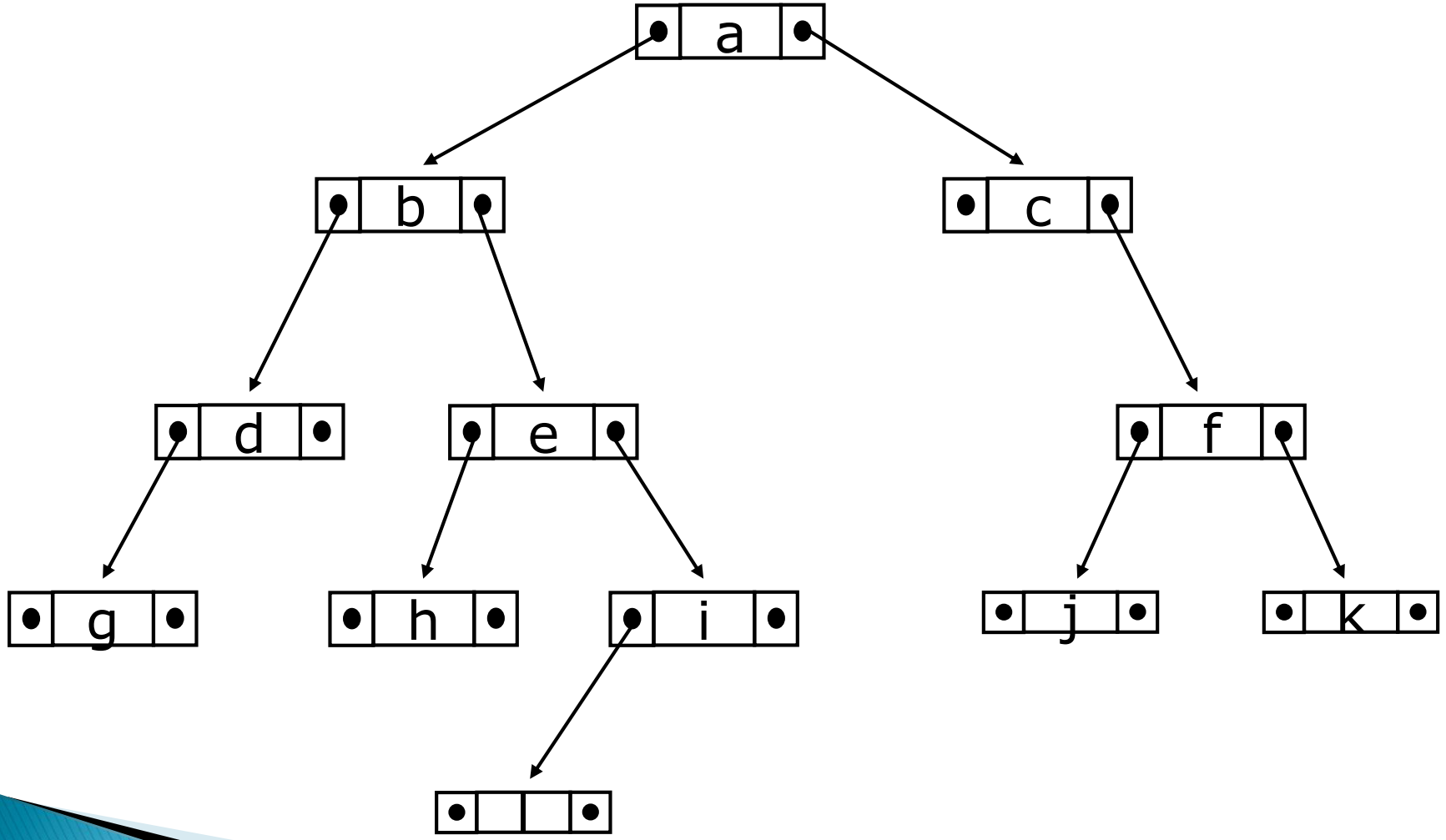


Tree node implementation

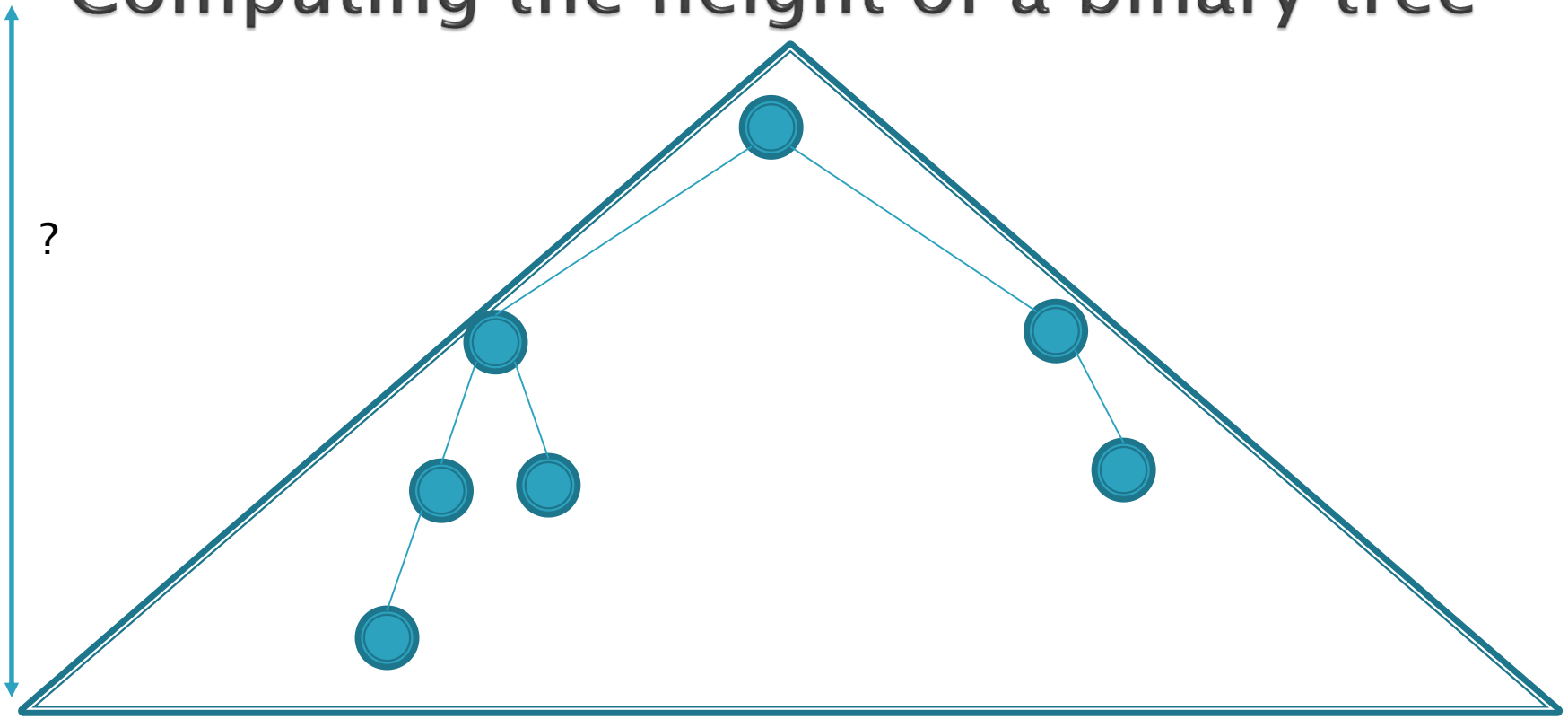
```
public class Node {
        public int data;
        public Node left;
        public Node right;

        public Node(int d) {
                data = d;
        }
```

Node

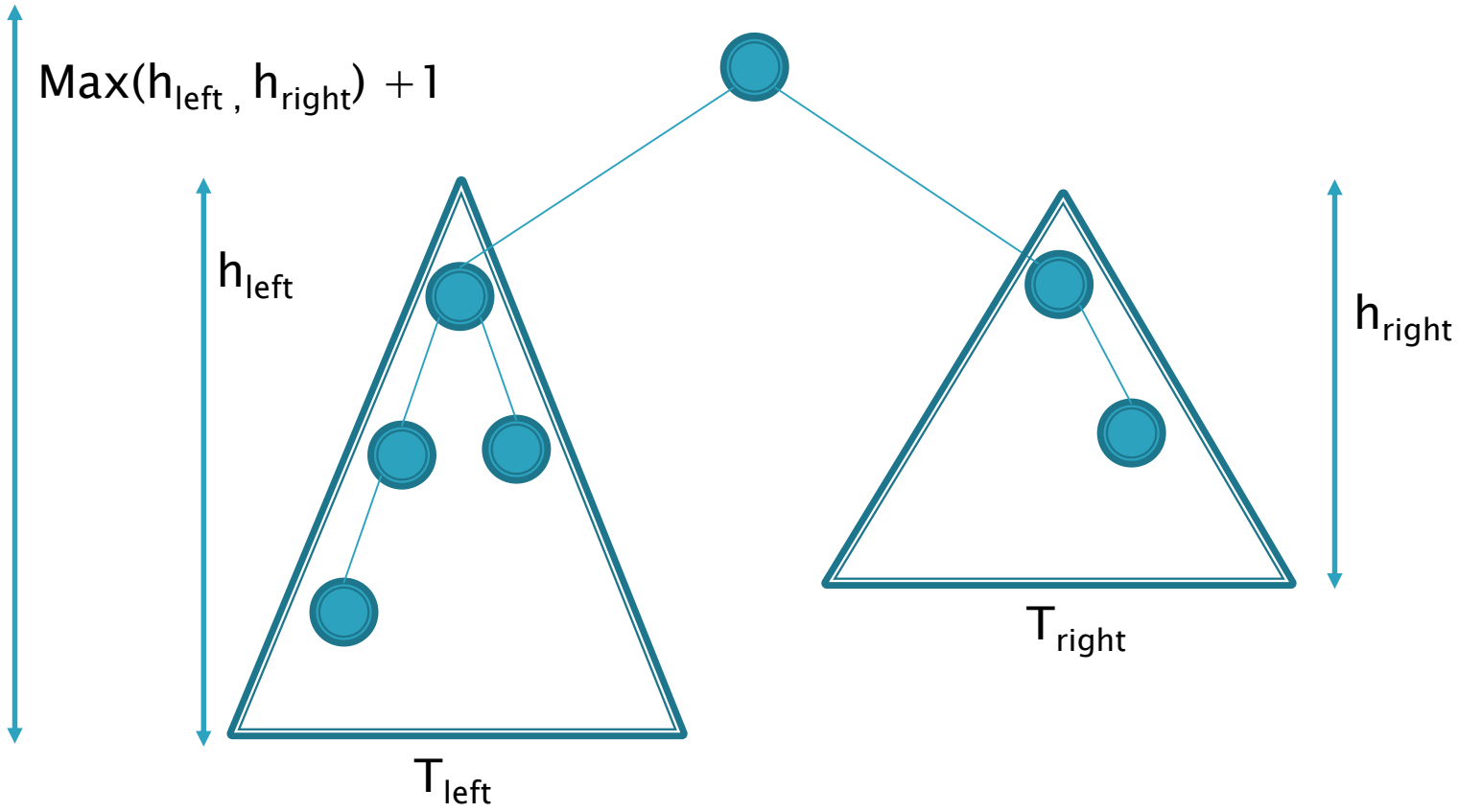
Binary tree implementation



Computing the height of a binary tree



Computing the height of a binary tree



Computing the height of a binary tree

ALGORITHM $Height(T)$

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

This week:

- Divide and Conquer technique
- Count a specific key in an array
- Master theorem
- Merge sort
- Binary tree
  - Computing the height
  - Compute the number of leaves

Compute the number of leaves

Algorithm *LeafCounter*(T)
//Computes recursively the number of leaves in a binary tree
//Input: A binary tree T
//Output: The number of leaves in T
if $T = \varnothing$ **return** 0 //empty tree
else if $T_L = \varnothing$ **and** $T_R = \varnothing$ **return** 1 //one-node tree
else return *LeafCounter*(T_L)+ *LeafCounter*(T_R) //general case

Try it/ homework

1. Chapter 5.1, page 174, questions 1, 6
2. Chapter 5.3, page 185, question 2
3. Implement a function to check if a tree is balanced. A balanced tree is defined to be a tree such that no two leaf nodes differ in distance from the root by more than one.