INT202
Complexity of Algorithms
Introduction

XJTLU/SAT/INT
SEM2 AY2021-2022

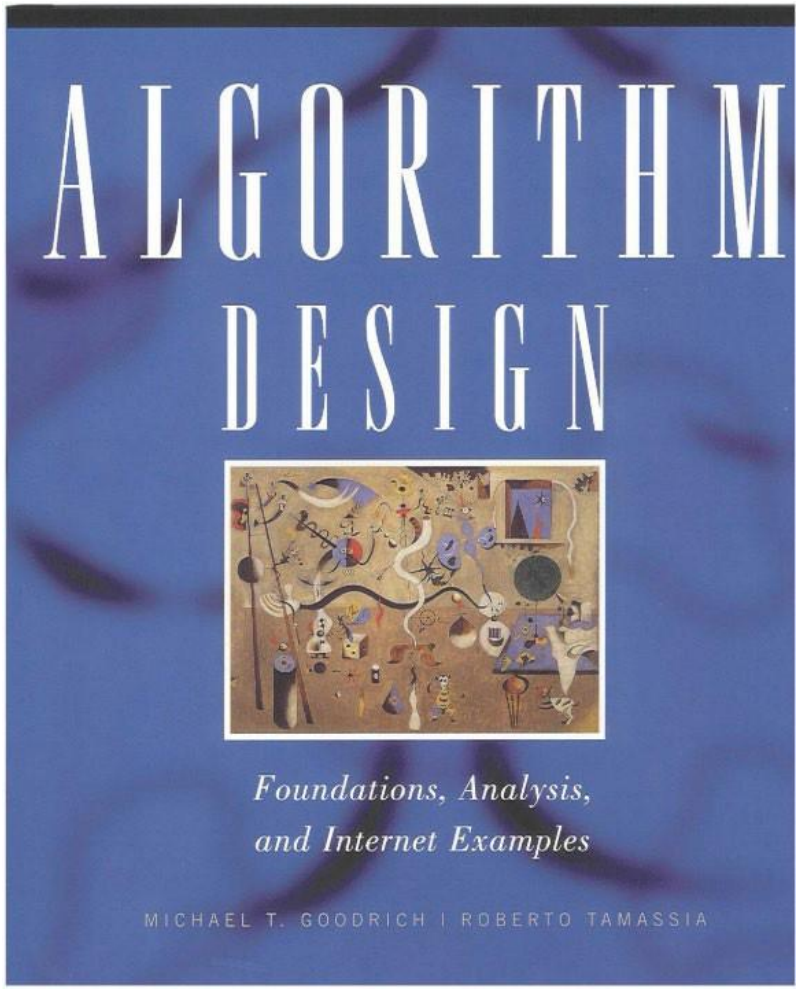
INT202: Complexity of Algorithms

- Module leader: Dr. Rui Yang
  - Email: R.Yang@xjtlu.edu.cn
- Lectures/Tutorials:
  - Monday (17:00-19:00)
  - Thursday (14:00-16:00)
- Assessment:
  - 1 In-Class Test in Week 8 (10%)
  - 1 In-Class Test in Week 14 (10%)
  - 1 Final Exam (80%)

Course Texts

- **Module Text Book:** *Algorithm Design*, Author: M.T. Goodrich and R. Tamassia (sometimes referred to as [GT] in these notes).
- **Secondary (reference):** *Introduction to Algorithms*,

  Author: T.H. Cormen, C. E. Leiserson and R.L. Rivest.



Module Outcomes

- By the end of this module a student should

  1. have an appreciation of the diversity of computational fields to which algorithmics has made significant contributions;

  2. have fluency with the structure and utilization of abstract data types such as stacks, queues, lists, and trees in conjunction with classical algorithmic problems such as searching, sorting, graph algorithms, etc;

  3. have a comfortable knowledge of graphs, their role in representing various relationships, and be familiar with several of the algorithms for solving common problems such as finding minimum spanning trees, maximum flows, shortest paths, and Euler tours;

Module Outcomes (cont.)

- By the end of this module a student should

  4. understand the basic concepts from number theory that are used in “public-key” cryptography, and be able to explain the concepts that underlie the RSA encryption/decryption scheme;

  5. be familiar with formal theories providing evidence that many important computational problems are inherently intractable, e.g., NP-completeness.

These Lectures

- Basic Concepts on Algorithm Analysis
- Introduce the complexity analysis of algorithms

Algorithms and Data Structures

- *A data structure* is a systematic way of organizing and accessing data.
- An *algorithm* is a sequence of steps for performing a task in a finite amount of time.

- Navigate to a place
- Transfer Audio/Video format
- Control solar panels
- Generate an image from 2D/3D model
- Root-finding algorithm
- Compression algorithm
- Optimization algorithm
- Rendering algorithm

Algorithms and Data Structures

- What makes a good algorithm?
- How do you measure efficiency?

Our fundamental interest is to design “good” *algorithms* which utilize efficient *data structures*. In this context “good” means *fast* in a way to be clarified during this course.

Algorithm Analysis

- The analysis of algorithm is the theoretical study of computer program performance and recourse usage.
- **Primary interest**: Running time (*time-complexity*) of the algorithm and the operations on data structures.
- **Secondary interest**: Space (or "memory") usage (*space-complexity*).

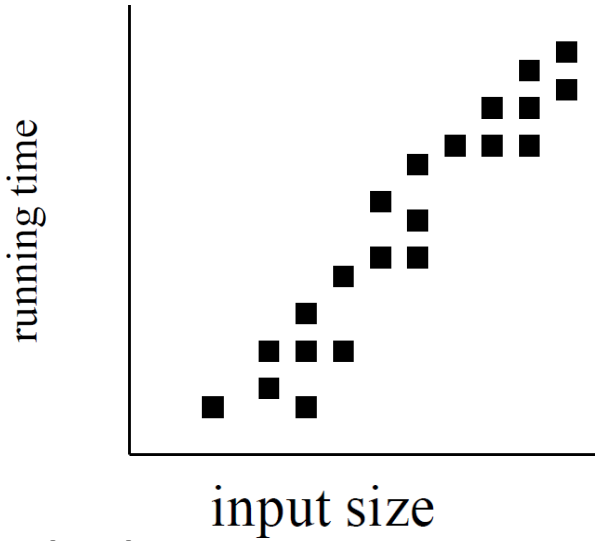
- This course focuses on the *mathematical* design and analysis of algorithms and their associated data structures, utilizing relevant mathematical tools to do so.

Experimental Analysis of Algorithms

- We are interested in *dependency* of the running time or memory requirement on *size* of the input.
- The *size* could mean the number of vertices and edges if we are operating on a graph, the length of a message we're encoding/decoding, and/or the actual length of numbers we're processing in terms of the bits needed to store them in memory.
- To analyze algorithms we sometimes perform *experiments* to empirically observe for example the running time.
- Doing so requires a good choice of sample inputs and an appropriate number of tests (so that we can have *statistical certainty* about our analysis).

Experimental Analysis (cont.)

Running-time depends on both the *size* and *instance* of input and the algorithm used, as well as the software and hardware environment on which it is run.

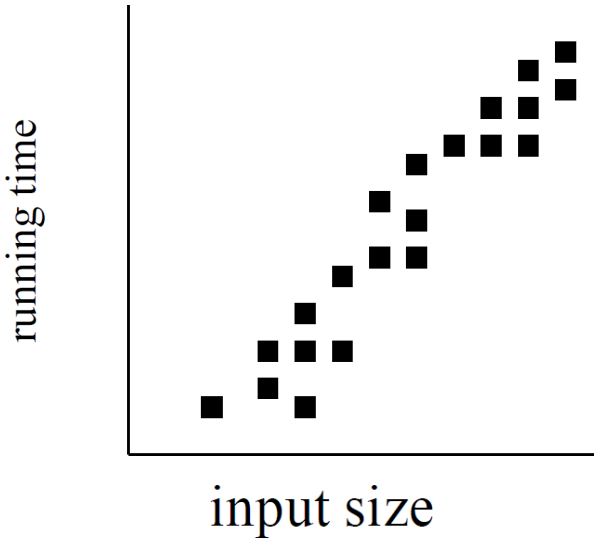


input size

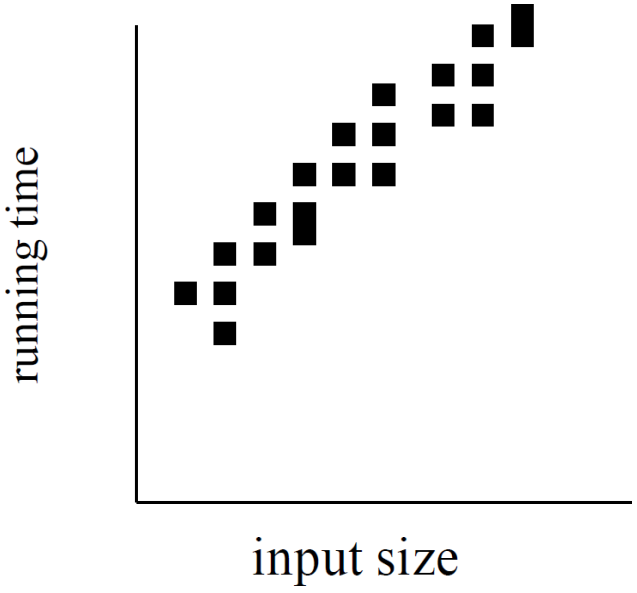
In general, the running time of an algorithm or data structure method increases with the input size

Experimental Analysis (cont.)

Running-time depends on both the *size* and *instance* of input and the algorithm used, as well as the software and hardware environment on which it is run.



a fast computer



a slow computer

Limitations of Experimental Analysis

1. Experiments are performed on a limited set of test inputs.
2. Requires all tests to be performed using same hardware and software.
3. Requires implementation and execution of algorithm.

Theoretical Analysis

Benefits over experimental analysis:

1. Can take all possible inputs into account.
2. Can compare efficiency of two (or more) algorithms, independent of hardware/software environment.
3. Involves studying high-level descriptions of algorithms (*pseudo-code*).

Theoretical Analysis (cont.)

- When we analyze an algorithm in this fashion, we aim to associate a function $f(n)$ to each algorithm, where $f(n)$ characterizes the running-time in terms of some measure of the *input-size*, n.

Typical functions include: n, $\log n$, n^2, $n \log n$, 2^n,...

'Algorithm A runs in time proportional to n'

Theoretical Analysis (requirements)

For formal theoretical analysis, we need:

1. A *language for* describing algorithms.
2. *Computational model* in which algorithms are executed.
3. *Metric* for measuring performance.
4. A way of characterizing performance.

Pseudo-code

- Pseudo-code is a high-level description language for algorithms.

Pseudo-code provides more structured description of algorithms.

Allows high-level analysis of algorithms to determine their running time (and memory requirements).

Task: Give a Pseudo-code for the function *Minimum-Element*, which finds the minimum element of a set of numbers.

Pseudo-code example

Algorithm Minimum-Element(A)

input: An array *A sorting* $n \geq 1$ *integers*

output: minimum element *min*

min ← A[0]

for i ← 1 to n-1 do

    if min > A[i] then

        min ←A[i]

return min

- Instead of the sign arrow ← an equal sign can also be used
- Instead of if-then-end if, the curly brackets {} can also be used

Pseudo-code (cont.)

- Pseudo-code is a mixture of natural language and high-level programming language (e.g., Java, C, etc.).

Describes a generic implementation of data structures or algorithms.

- Pseudo-code includes: expressions, declarations, variable initialization and assignments, conditionals, loops, arrays, method calls, etc.

In here, we won't formally define a strict method for giving pseudo-code, but when using it we aim to describe an algorithm in a manner that would allow a competent programmer to translate the pseudo-code into program code without misinterpretation of the algorithm.

Pseudo-code (cont.)

We define a set of high-level primitive operations that are largely independent from the programming language used.

Primitive operations include the following:

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method.

❖To analyze the running time of an algorithm we *count* the number of operations executed during the course of the algorithm.

Counting Primitive Operations

Algorithm arrayMax(A,n):
input: array *A*
output: maximum element *currentMax*

```
1    currentMax ← A[0]
2    for j ← 1 to n-1 do
3            if currentMax<A[j]
4                then currentMax ← A[j]
5    return currentMax
```

How many primitive operations?

Random Access Machine

This approach of simple counting primitive options gives rise to a computational model called the Random Access Machine.

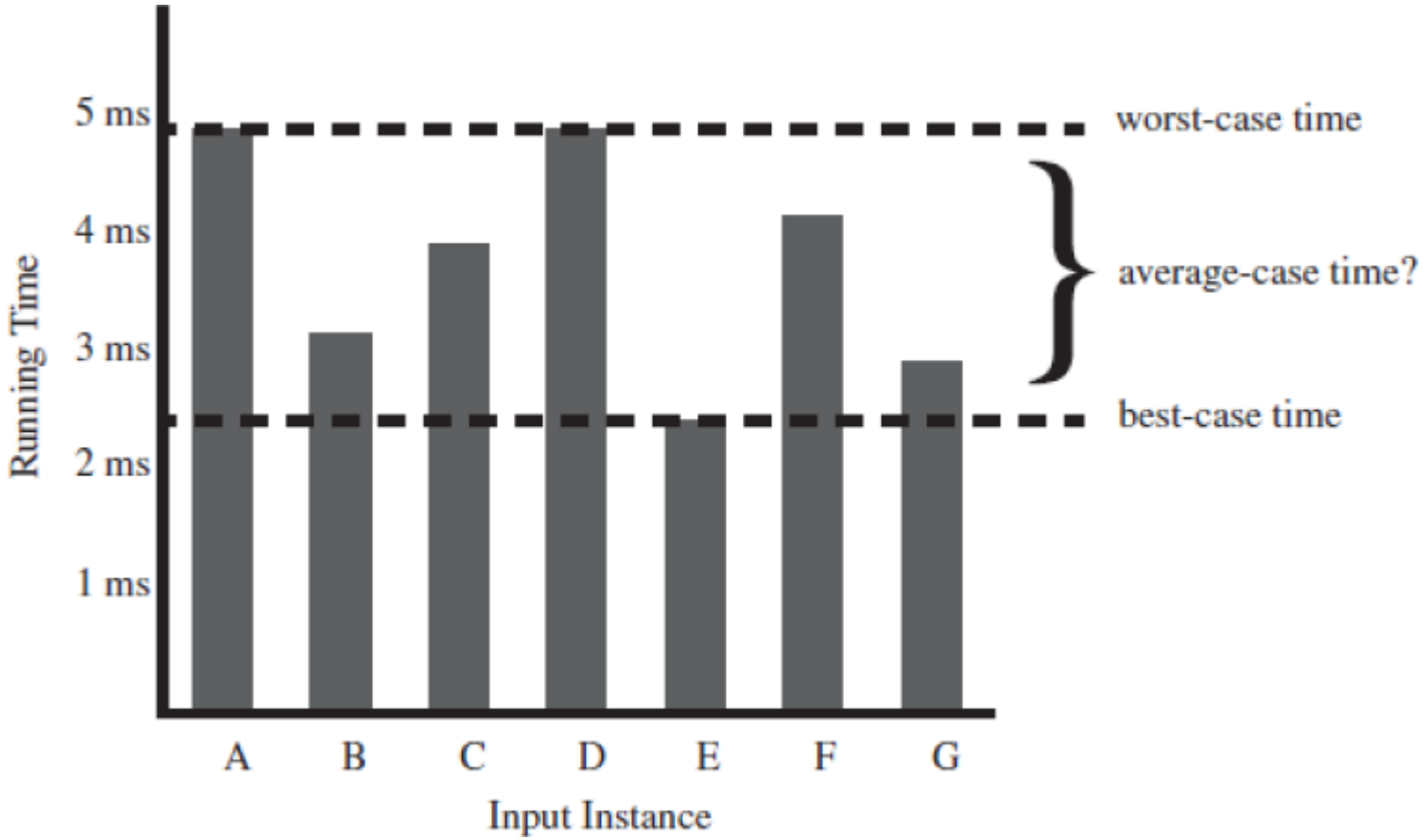
- CPU connected to a bank of *memory cells*.
- Each memory cell can store a number, character, or address.

Assumption: primitive operations (like a single addition, multiplication, or comparison) require *constant time* to perform.

(Not necessarily true, e.g. multiplication takes about four times as long to perform on a computer than addition.)

Average- vs. Worst-Case Complexity

An algorithm may run faster on some inputs compared to others.



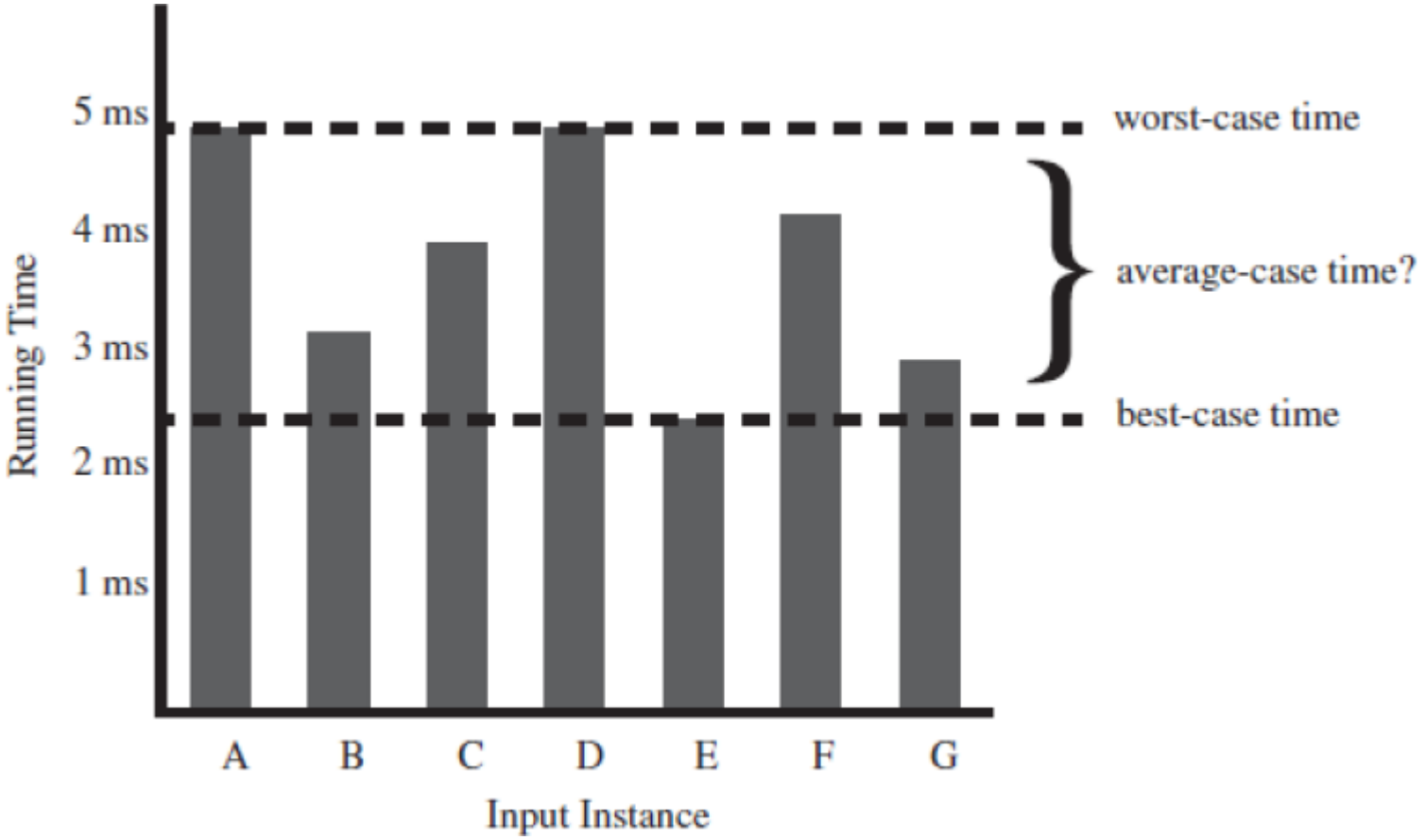
Average- vs. Worst-Case Complexity

An algorithm may run faster on some inputs compared to others.

- *Average-case* complexity refers to running time as an *average taken over all inputs* of the same size.
- *Worst-case* complexity refers to running time as the *maximum taken over all inputs* of the same size.

Usually, we're most interested in worst-case complexity.

Average- vs. Worst-Case Complexity (cont.)



An average-case analysis also typically requires that we calculate expected running times based on a given input distribution.

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

1 *currentMax* ← *A*[0]

2 **for** *j* ← 1 **to** *n-1* ***do***

3 **if** *currentMax*<*A*[*j*]

4 **then** *currentMax* ← *A*[*j*]

5 **return** *currentMax*

How many primitive operations?

1. Line 1*:*
2. Line 2*:*
3. Line 5*:*

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

1 *currentMax* ← *A*[0]

2 **for** *j* ← 1 **to** *n-1* ***do***

3 **if** *currentMax*<*A*[*j*]

4 **then** *currentMax* ← *A*[*j*]

5 **return** *currentMax*

How many primitive operations?

1. Line 1*:* Array indexing + Assignment *2*
2. Line 2*:* Initializing j 1

   Verifying j<n *n*
3. Line 5*:* Returning *1*

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

```
1    currentMax ← A[0]
2    for j ← 1 to n-1 do
3            if currentMax<A[j]
4                then currentMax ← A[j]
5    return currentMax
```

How many primitive operations?

4. Line 3:
5. Line 4*:*

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

```
1    currentMax ← A[0]
2    for j ← 1 to n-1 do
3            if currentMax<A[j]
4                then currentMax ← A[j]
5    return currentMax
```

How many primitive operations?

4. Line 3: Array indexing +Comparing 2(n-1)
5. Line 4: Array indexing +Assignment 2(n-1)

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

```
1    currentMax ← A[0]
2    for j ← 1 to n-1 do
3            if currentMax<A[j]
4                then currentMax ← A[j]
5    return currentMax
```

Worst case

How many primitive operations?

4. Line 3
5. Line 4

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

```
1    currentMax ← A[0]
2    for j ← 1 to n-1 do
3            if currentMax<A[j]
4                then currentMax ← A[j]
5    return currentMax
```

Best case ?

How many primitive operations?

4. Line 3
5. Line 4

Counting Primitive Operations

Algorithm arrayMax(A,n):

input: array *A*

output: maximum element *currentMax*

			Count
1	*currentMax* ← *A*[0]	Array indexing + Assignment	2
2	**for** *j* ← 1 **to** *n-1* ***do***	Initializing j Verifying j<n	1+n
3	**if** *currentMax*<*A*[*j*]	Array indexing + Comparing	2(n-1)
4	**then** *currentMax* ← *A*[*j*]	Array indexing + Assignment	2(n-1)
		Incrementing the counter	2(n-1)
5	**return** *currentMax*	Returning	1

How many primitive operations?

Best case: 2+1+n+4(n-1)+1=5n

Worst case: 2+1+n+6(n-1)+1 =7n-2

Recursive Algorithms

- Recursion involves a procedure calling itself to solve *sub-problems* of a smaller size. These smaller subproblems can then be combined in some way to get a solution to a larger problem.
- Recursive procedures require a *base case* that can be solved directly without using recursion.

Recurrence Relations

- *Recurrence relations* sometimes allow us to define the running-time of an algorithm in the form of an equation.

Suppose that $T(n)$ denotes the running time of algorithm on input of size n. Then we might be able to characterize $T(n)$ in terms of, say, $T(n-1)$. For example, we might be able to show that

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{for } n \geq 2 \end{cases}$$

Ideally, given such a relationship we would then want to express this recurrence relation in a *closed form*.

In the example, we can show that

$$T(n) = 7(n-1) + 3 = 7n - 4.$$

3 10 17 24 …

Recurrence Relations (cont.)

- Recurrence relations may appear in many forms. Some examples include:

1. $C(n) = 3 \cdot C(n-1) + 2 \cdot C(n-2) + C(n-3)$ where

   $C(1) = 1, C(2) = 3, C(3) = 5$

2. The Fibonacci numbers

Recursion example: Fibonacci Numbers

The Fibonacci numbers are defined as the sequence

$f_1 = f_2 = 1$, and $f_n = f_n - 1 + f_n - 2$, for $n \geq 3$. They can be found using the following pseudo-code that computes them recursively.

Problem: Write a piece of pseudocode to compute Fibonacci numbers, n=50.

The terms of the Fibonacci sequence are: 1,1,2,3,5,8,13,21,34.

Problem

Algorithm: fibonacci numbers
Input: upper limit n
Output: The n-th term of Fibonacci

```
int fib (int n){
if n <=2
    return 1;
else
    return fib(n-1)+fib(n-2);
}
```

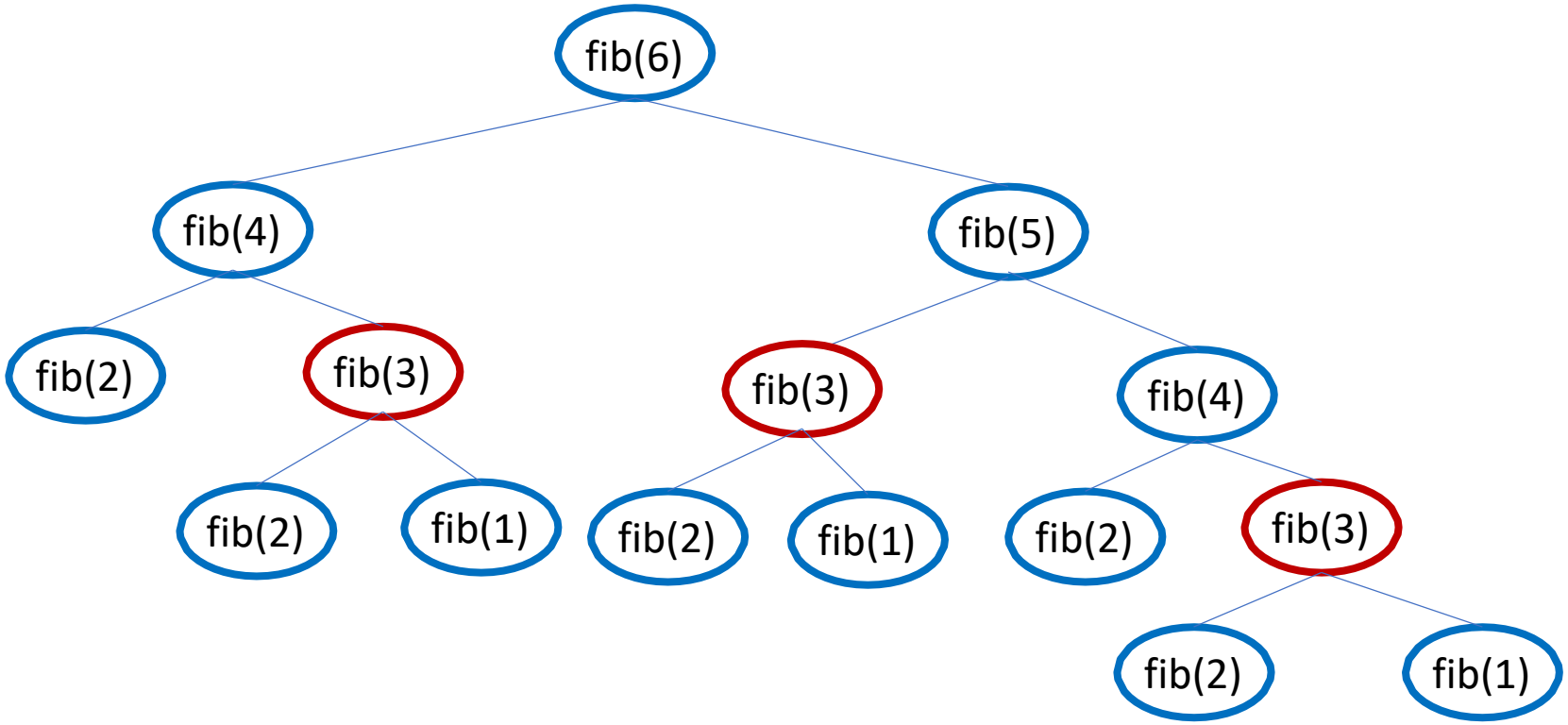
Recursive Algorithms: A Word of Caution…

While recursive algorithms are often “simpler” to write than a non-recursive version, there are often reasons to avoid them.

- In many situations, the smaller subproblems might be solved *repeatedly* during execution of the recursive algorithm.

Recursive Algorithms: A Word of Caution…

To compute Fibonacci(*n*), we must compute Fibonacci(*n* − 1) and Fibonacci(*n* − 2). **Both** of these function calls must then compute Fibonacci(*n* − 3) and Fibonacci(*n* − 4), etc. This repetition of work can massively increase the overall running time of the algorithm.



Recursive Algorithms: A Word of Caution…

To compute Fibonacci(*n*), we must compute Fibonacci(*n* − 1) and Fibonacci(*n* − 2). **Both** of these function calls must then compute Fibonacci(*n* − 3) and Fibonacci(*n* − 4), etc. This repetition of work can massively increase the overall running time of the algorithm.

- Because of the above phenomena, a recursive algorithm could also be impossible to perform on a computer (for large input values) because the repeated function calls might exhaust the memory of the machine.

Recursive Algorithms

- Recursion involves a procedure calling itself to solve *sub-problems* of a smaller size. These smaller subproblems can then be combined in some way to get a solution to a larger problem.
- Recursive procedures require a *base case* that can be solved directly without using recursion.

Exercise

- Write pseudo-code for a non-recursive method to compute *Fibonacci*(*n*), assuming that *n* is a positive integer.

Hint: Avoid the repeated computations mentioned above, by starting from the beginning of the sequence and “working up” to get the term you want.

Exercise

```
Algorithm: fibonacci numbers
Input: upper limit Nmax
Int f(int Nmax)
{
f1 ← 1;
f2 ←1;
  for n←3:(Nmax-2){
       fn← f2 + f1;
      f1 ← f2;
      f2 ← fn;
   return fn;
}}
```

INT202
Complexity of Algorithms
Introduction

XJTLU/SAT/INT
SEM2 AY2021-2022

Asymptotic notation

- *Asymptotic notation* allows characterization of the *main factors* affecting running time.

Used in a *simplified analysis* that estimates the number of primitive operations executed *up to a constant factor*.

Such notation lets us compare the running times of two algorithms.

Importance of asymptotics

Maximum size allowed for an input instance for various running times to be solved in 1 second, 1 minute and 1 hour, assuming a 1MHz machine:

Running Time	Maximum problem size (n)		
	1 second	1 minute	1 hour
$400n$	2,500	150,000	9,000,000
$20n \log n$	4,096	166,666	7,826,087
$2n^2$	707	5,477	42,426
n^4	31	88	244
2^n	19	25	31

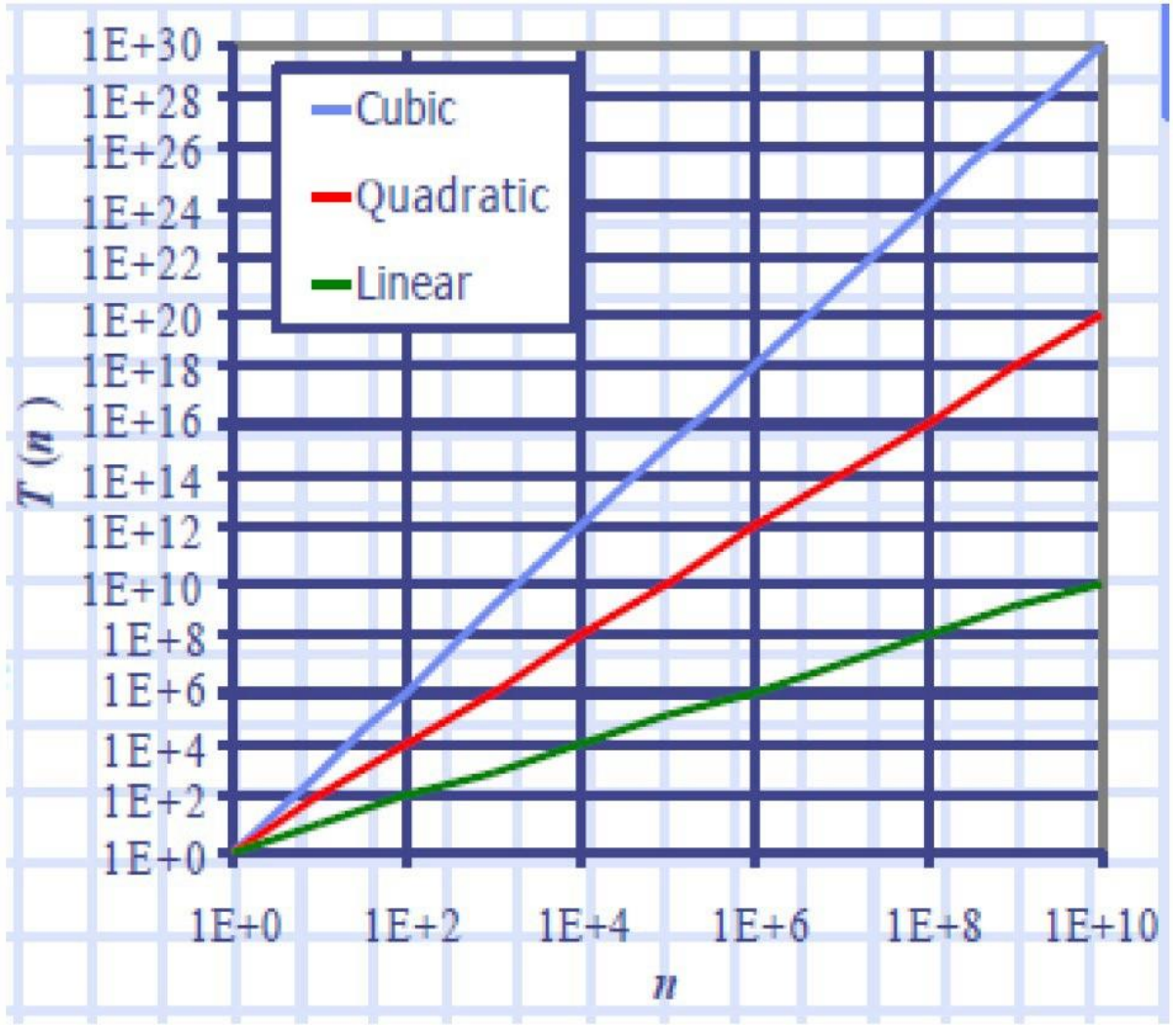
An algorithm with an asymptotically slow running time is beaten in the long run by an algorithm with an asymptotically faster running time.

Growth rate

Growth rates of functions:

- Linear ≈n
- Quadratic ≈ n^2
- Cubic ≈ n^3

In a log-log chart, the slope of the line corresponds to the growth rate of the function



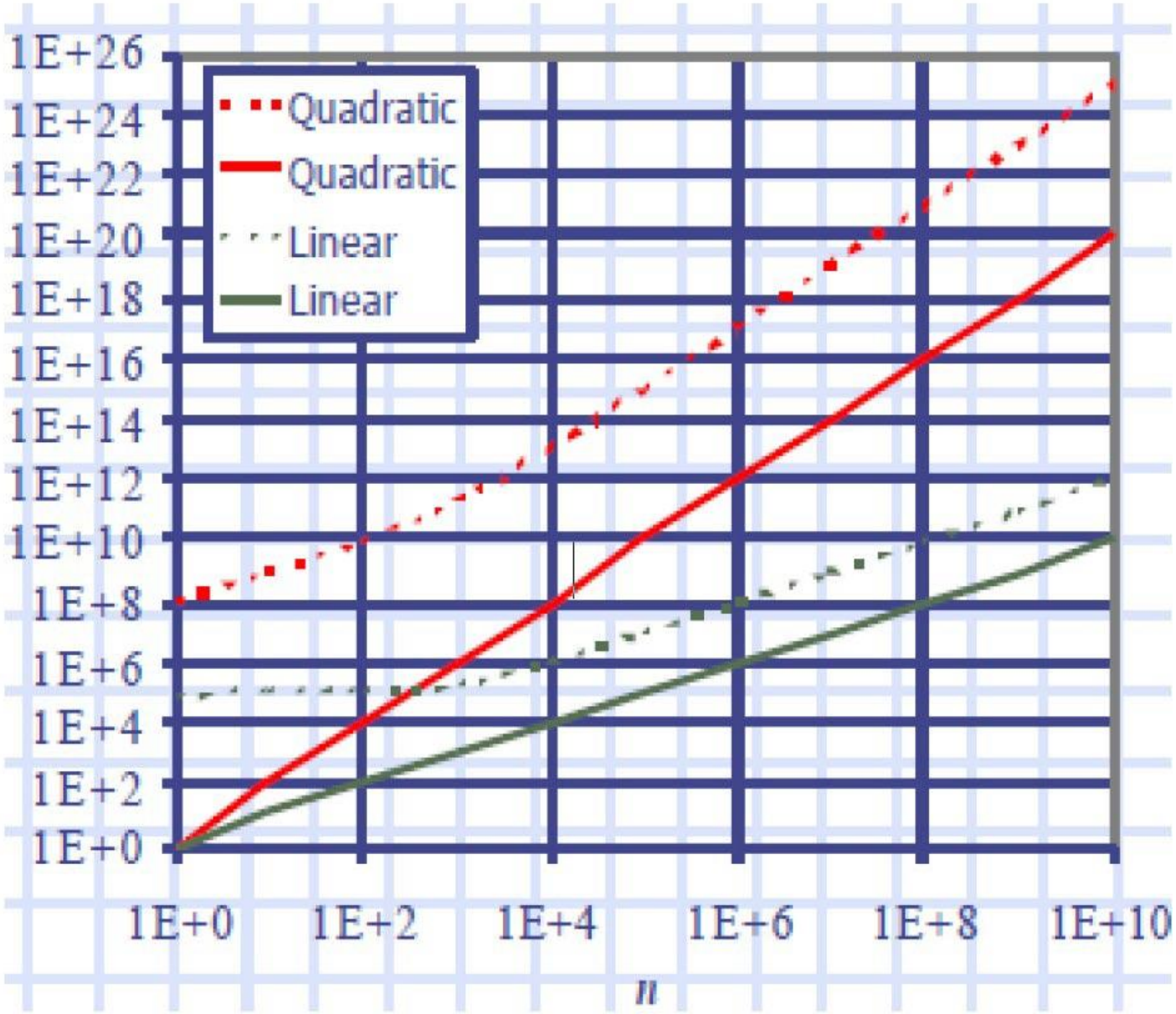
Growth rate

The growth rate is not affected by

- constant factors or
- lower-order terms

Examples

- $10^2 n + 10^5$ is a linear function
- $10^5 n^2 + 10^8 n$ is a quadratic function



“Big-Oh” Notation

“Big-Oh” notation is probably the most commonly used form of asymptotic notation.

- Given two positive functions $f(n)$ and $g(n)$ (defined on the nonnegative integers), we say f (n) is O(g(n)), written f (n) ∈O(g(n)), if there are constants c and n_0 such that:

$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

“Big-Oh” Notation

- Given two positive functions $f(n)$ and $g(n)$ (defined on the nonnegative integers), we say f (n) is O(g(n)), written f (n) ∈O(g(n)), if there are constants c and n_0 such that:

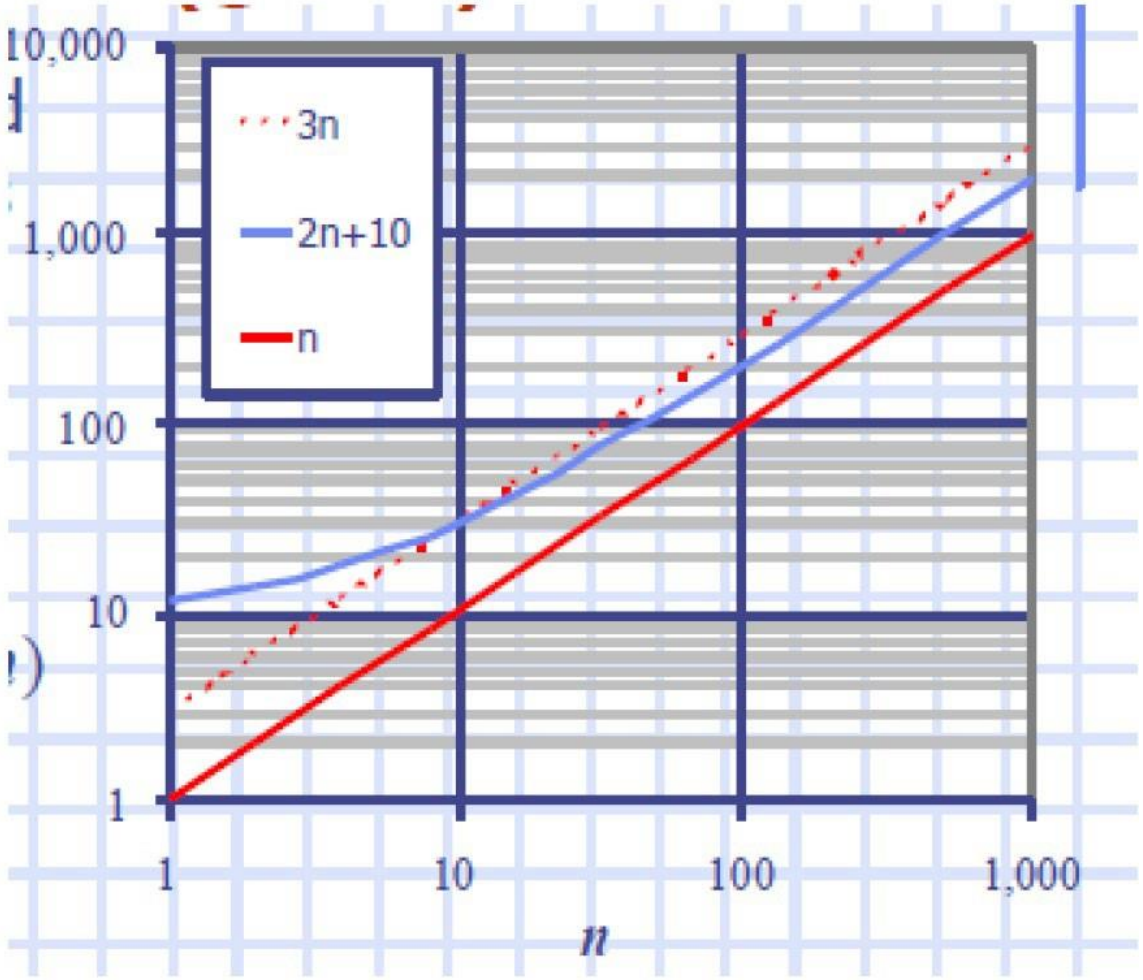
$$f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

Example: 2***n*** + 10 is ***O***(***n***)

“Big-Oh” Notation

Example: $2n + 10$ is $O(n)$

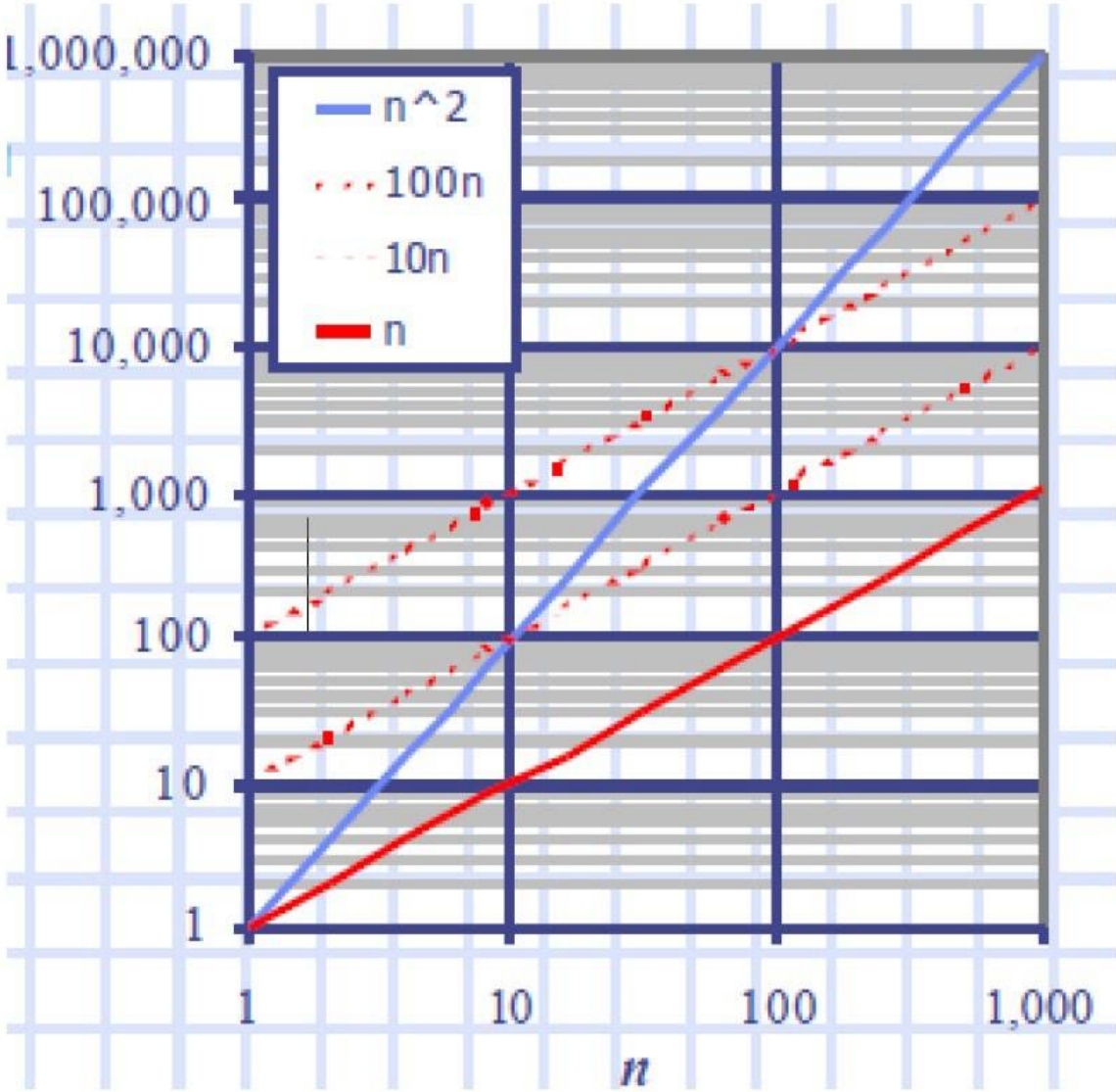
- $2n + 10 \leq cn$
- $(c - 2)\, n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$



“Big-Oh” Notation

Example: the function n^2 is not $O(n)$

- $n^2 \leq cn$
- $n \leq c$
- The above inequality cannot be satisfied since c must be a constant



Growth rates (running time)

Functions ordered by growth rate:

$\log_2 n$	$N^{1/2}$	n	$n \log_2 n$	n^2	n^3	2^n
1	1.4	2	2	4	8	4
2	2	4	8	16	64	16
3	2.8	8	24	64	512	256
4	4	16	64	256	4096	65536
5	5.7	32	160	1024	32768	4294967296
6	8	64	384	4096	262144	1.84^{19}
7	11	128	896	16384	2097152	2.40×10^{38}
8	16	256	2048	65536	16777216	1.15^{77}
9	23	512	4608	262144	134217728	1.34×10^{154}
10	32	1024	10240	1048576	1073741824	1.79^{308}

More “Big-Oh” Examples

- 7n-2
  7n-2 is **O(n)**
  need **c** > 0 and $\boldsymbol{n_0} \geq 1$ such that $7\boldsymbol{n}\text{-}2 \leq \boldsymbol{cn}$ for $\boldsymbol{n} \geq \boldsymbol{n_0}$
  this is true for **c** = 7 and $\boldsymbol{n_0} = 1$

- $3\boldsymbol{n}^3 + 20\boldsymbol{n}^2 + 5$
  $3\boldsymbol{n}^3 + 20\boldsymbol{n}^2 + 5$ is $\boldsymbol{O(n^3)}$
  need **c** > 0 and $\boldsymbol{n_0} \geq 1$ such that $3\boldsymbol{n}^3 + 20\boldsymbol{n}^2 + 5 \leq \boldsymbol{cn}^3$ for $\boldsymbol{n} \geq \boldsymbol{n_0}$
  this is true for **c** = 4 and $\boldsymbol{n_0} = 21$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function.
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	g(n) is O(f(n))
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Common functions

Here is a list of classes of functions that are commonly encountered when analyzing algorithm

- *Constant* $O(1)$
- *Logarithmic* $O(\log n)$
- *Linear* $O(n)$
- *Log-linear* $O(n\log n)$
- *Quadratic* $O(n^2)$
- Cubic $O(n^3)$
- *Polynomial* $O(n^k)$
- *Exponential* $O(a^n), a > 1$
- Factorial O(n!)

Big-Oh Rules

$2n$ is $O(n^2)$?

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
- Drop lower-order terms
- Drop constant factors

- Use the smallest possible class of functions
- Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

- Use the simplest expression of the class
- Say "$3n+5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Further examples of Big-Oh

1. 13 n^3 + 7*n* log *n* + 3 is *O*(_ n^3 _).

   Proof: 13 n^3 + 7*n* log *n* + 3 $\leq$ 16 n^3, for n ≥1

2. 3 log *n* + log log *n* is *O*(_log n__).

   Proof: 3 log *n* + log log *n* $\leq$ 4 log n, for n ≥ 2

3. 2^{70} is *O*(_ 1_).

   Proof: $2^{70} \leq 2^{70}$ *1, for n ≥1.

Asymptotic Algorithm Analysis

The asymptotic analysis of an algorithm determines the running time in big-Oh notation

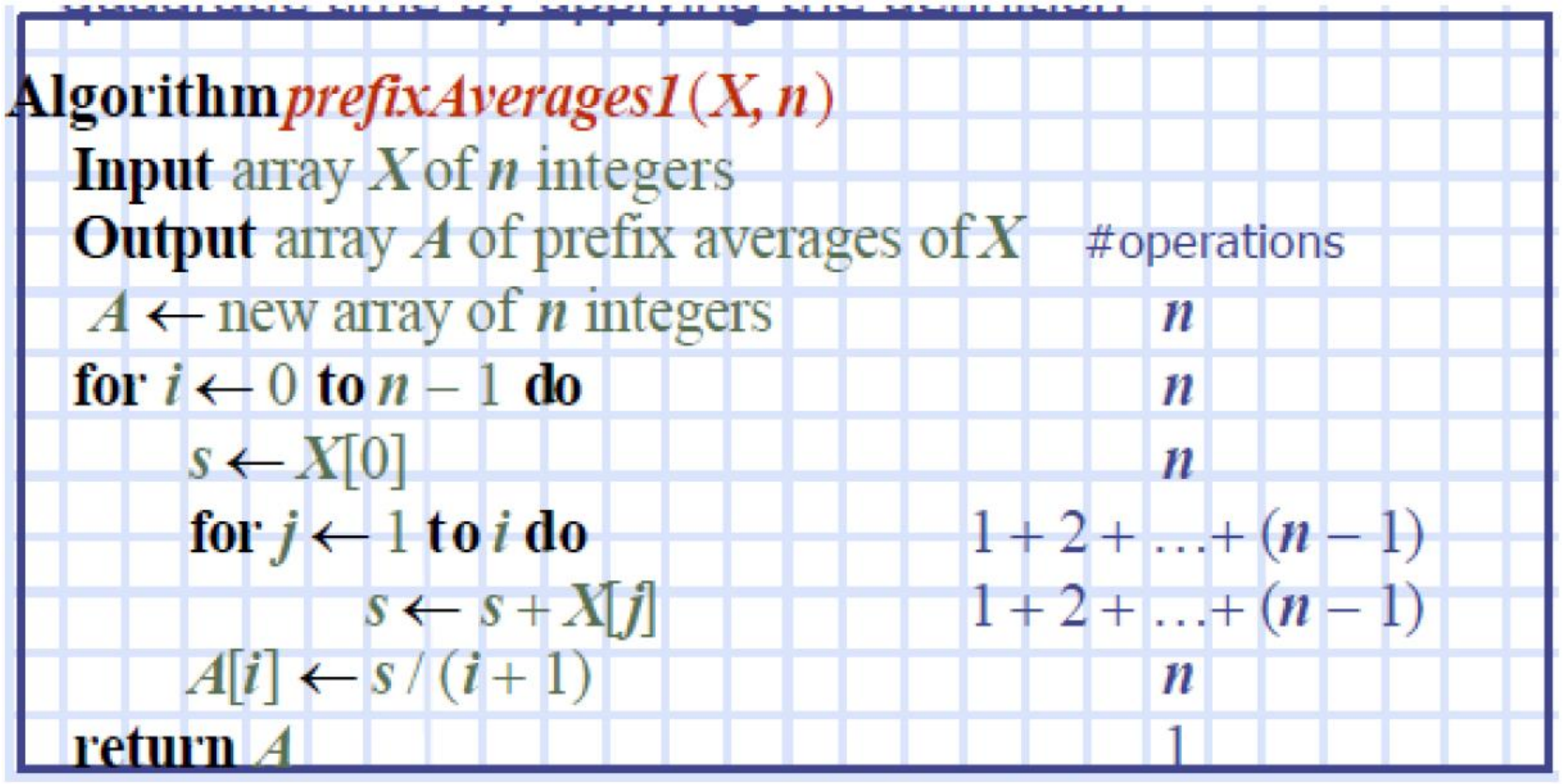
To perform the asymptotic analysis

- We find the worst-case number of primitive operations executed as a function of the input size
- We express this function with big-Oh notation

Example:

- We determine that the algorithm “Maximum-Element(A) ” executes at most $7n - 2$ primitive operations
- We say that algorithm “runs in $O(n)$ time”

Asymptotic Algorithm Analysis: Example



```
Algorithm prefixAverages1(X, n)
    Input array X of n integers
    Output array A of prefix averages of X      #operations
    A ← new array of n integers                 n
    for i ← 0 to n − 1 do                       n
        s ← X[0]                                n
        for j ← 1 to i do                       1 + 2 + …+ (n − 1)
            s ← s + X[j]                        1 + 2 + …+ (n − 1)
        A[i] ← s / (i + 1)                      n
    return A                                    1
```

Algorithm prefixAverage1 runs in O(?)

Asymptotic Algorithm Analysis: Example

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Asymptotic Algorithm Analysis: Example

```
Algorithm prefixAverages2(X, n)                    #operations
    Input array X of n integers
    Output array A of prefix averages of X
    A ← new array of n integers                        n
    s ← 0                                              1
    for i ← 0 to n − 1 do                              n
        s ← s + X[i]                                   n
        A[i] ← s / (i + 1)                             n
    return A                                           1
```

Algorithm prefixAverage2 O(n)

Asymptotic Algorithm Analysis: Exercises

1 Give a **big-Oh** characterization, in terms of n, of the running time of the method Loop1.

2 Perform a similar analysis for method Loop2.

3 Perform a similar analysis for method Loop3.

4 Perform a similar analysis for method Loop4.

Algorithm Loop1(n):

```
1   s ← 0
2   for i ← 1 to n do
3       s ← s + i
```

Algorithm Loop2(n):

```
1   p ← 1
2   for i ← 1 to 2n do
3       p ← p · i
```

Algorithm Loop3(n):

```
1   p ← 1
2   for i ← 1 to n^2 do
3       p ← p · i
```

Algorithm Loop4(n):

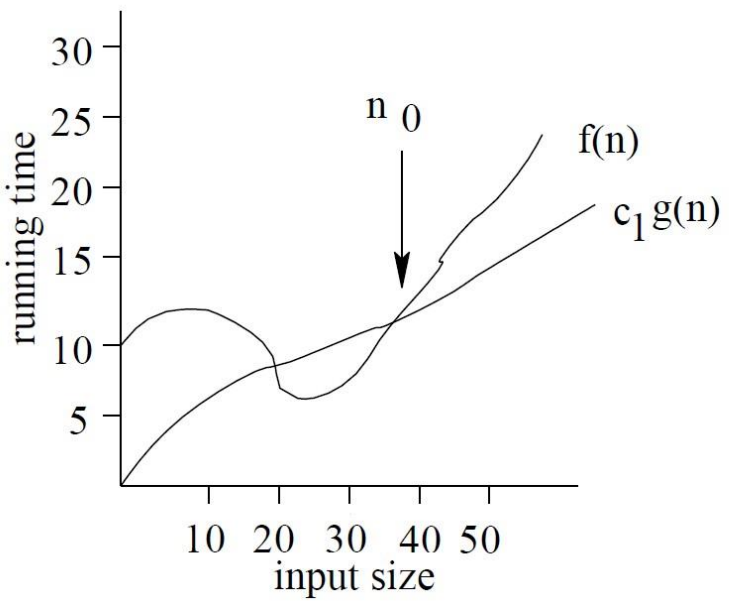
```
1   s ← 0
2   for i ← 1 to 2n do
3       for j ← 1 to i do
4           s ← s + i
```

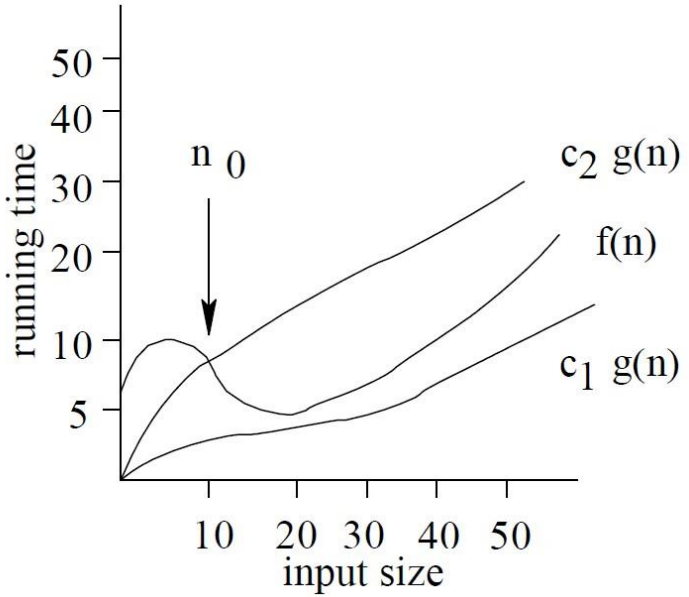
$\Omega(n)$ and $\Theta(n)$ notation

- We say that $f(n)$ is $\Omega(g(n))$ (*big-Omega*) if there are real constants c and n_0 such that:

$$f(n) \geq cg(n) \text{ for all } n \geq n_0.$$

- We say that $f(n)$ is $\Theta(g(n))$ (*Theta*) if $f(n)$ is $\Omega(g(n))$ and $f(n)$ is also $O(g(n))$.





Intuition for Asymptotic Notation

- Big-Oh

  f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)
- Big-Omega

  f(n) is Ω(g(n)) if f(n) is asymptotically greater than or equal to g(n)
- Big-Theta

  f(n) is Θ(g(n)) if f(n) is asymptotically equal to g(n)

Examples

1. $3\log n + \log\log n$ is $\Omega(\_ \log n\_\_\_)$.

Proof: $3\log n + \log\log n \geq 3 \log n$, for $n \geq 2$.

2. $3\log n + \log\log n$ is $\Theta(\_\log n \_\_\_)$.

Space Complexity

- Space complexity is a measure of the amount of working storage an algorithm needs. That means how much memory, in the worst case, is needed at any point in the algorithm.
- As with time complexity, we‘re mostly concerned with how the space needs grow, in big-Oh terms, as the size N of the input problem grows.

Space Complexity

```
int sum(int x, int y, int z) {
    int r = x + y + z;
    return r;
}
```

requires 3 units of space for the parameters and 1 for the local variable, and this never changes, so this is O(1).

Space Complexity

```
int sum(int a[], int n) {
    int r = 0;
    for (int i = 0; i < n; ++i) {
        r += a[i];
      }
    return r;
 }
```

requires N units for a, plus space for n, r and i, so it's O(N).

INT202
Complexity of Algorithms
Data Structures

XJTLU/SAT/INT

SEM2 AY2021-2022

Review

- The actual run time is difficult to assess.
  - Input, CPU frequency, RAM, data transmission speed, programs preempting resources
- The size of the problem is often the most important factor in determining algorithm running time.
  - Define the time complexity of the algorithm
  - Growth rate, asymptotic notations

Data Structures

- Algorithmic computations require data (information) upon which to operate.

- The speed of algorithmic computations is (partially) dependent on efficient use of this data.

- *Data structures* are specific methods for storing and accessing for information.

- We study different kinds of data structures as one kind may be more appropriate than another depending upon the *type* of data stored, and *how* we need to use it for a particular algorithm.

D

Data Structures: Stacks

A stack is a *Last-In, First-Out* (LIFO) data structure.

- Items can be inserted into a stack at any time (assuming no stack-overflow).

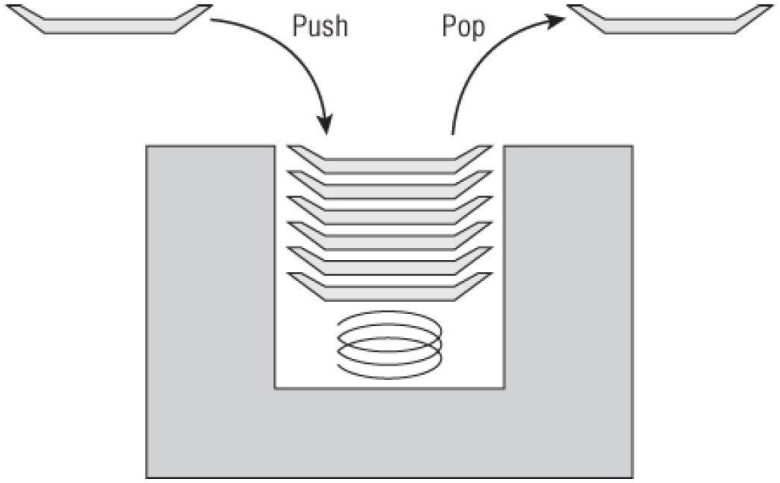
- We only have direct access to the last element that was inserted.

For example, Web browsers store recently visited web addresses on a stack.

Stack Abstract Data Type

A *stack* is an Abstract Data Type (ADT) and supports the following methods:

- *push(Obj)* : Insert object *Obj* onto the top of the stack.
- *pop()* : Remove (and return) the object from the top of the stack. An error occurs if the stack is empty.



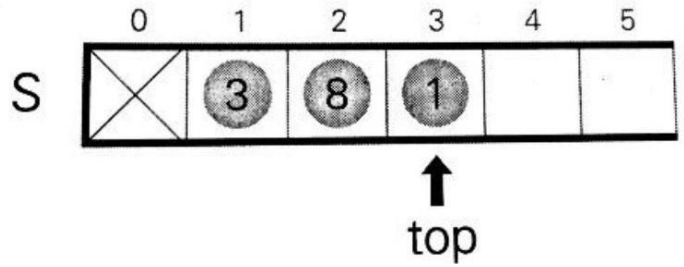
In addition to *push* and *pop* there are also typically methods like:

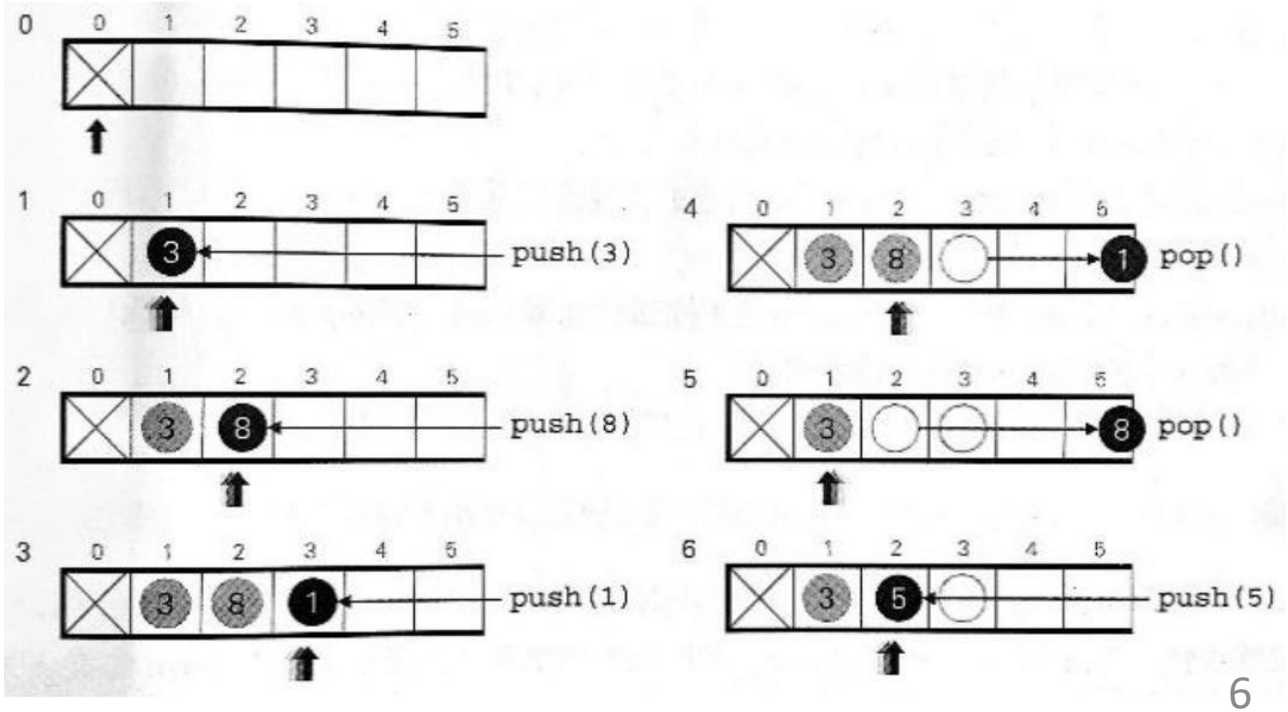
- *initialize() : initialize a stack*
- *isEmpty()* : returns a **true** if stack is empty, **false** otherwise.
- *isFull()* : returns a true if stack is full, false otherwise.

Stack: Push and Pop methods

```
PUSH(Obj)
1   ▷ Check to see if stack is full
2   if size() == N
3       then indicate stack-full error occurred
4   else t ← t + 1
5            S[t] ← Obj
```

```
POP()
1   ▷ Check to see if stack is empty
2   if isEmpty()
3       then indicate "stack empty" error occurred
4   else Obj ← S[t]
5            S[t] ← null
6            t ← t − 1
7   return Obj
```





Stacks: Applications

1. Important in *run-time environments* of modern procedural languages (e.g., C, C++, Java).

2. Evaluating arithmetic expressions can be performed using a stack if they are given using *postfix* notation (Reverse Polish notation (RPN), named for its developer Jan Łukasiewicz).

Stacks: Applications

Problem : Reversing an Array

The following pseudocode shows this algorithm:

```
ReverseArray(Data: values[])
      // Push the values from the array onto the stack.
      Stack: stack = New Stack
      For i = 0 To <length of values> - 1
          stack.Push(values[i])
      Next i
      // Pop the items off the stack into the array.
      For i = 0 To <length of values> - 1
          values[i] = stack.Pop()
      Next i
End ReverseArray
```

Stacks: Applications

Problem : Reverse Polish notation

In a standard arithmetic expression we write operands interspersed with the arithmetic operations, e.g. x + y where x and y are operands and the + addition operator is applied to add y to x . This is also referred to as infix notation.

With infix notation it is important to understand the precedence of operations. For example, which operation is performed first in the expression $4 + 3 * 9$?

Stacks: Applications

Problem : Reverse Polish notation

In postfix notation the operands precede the arithmetic operations, e.g. x y +, x y z + $*$ or x y + z $*$. When you encounter an operator, it applies to the previous two operands in the list, in that order.

e.g.: You'd want to represent an expression like $-x$, i.e. the unary negation symbol, as something like x -1 $*$ using postfix notation

Stacks: Applications

Problem : Reverse Polish notation

x y +, x y z + ∗, x y + z ∗

In the first example above we simply take x and y and add them together.

In the second we take y and add z to it, then finish by multiplying y + z by x .

The third expression computes (x + y) ∗ z.

Stacks: Applications

Problem : Reverse Polish notation

► The postfix expression

$$x\ y\ w\ z\ /\ -\ \ *$$

gets translated into the expression

$$x * (y - w/z)$$

whereas

$$x\ y\ w\ /z\ -\ *$$

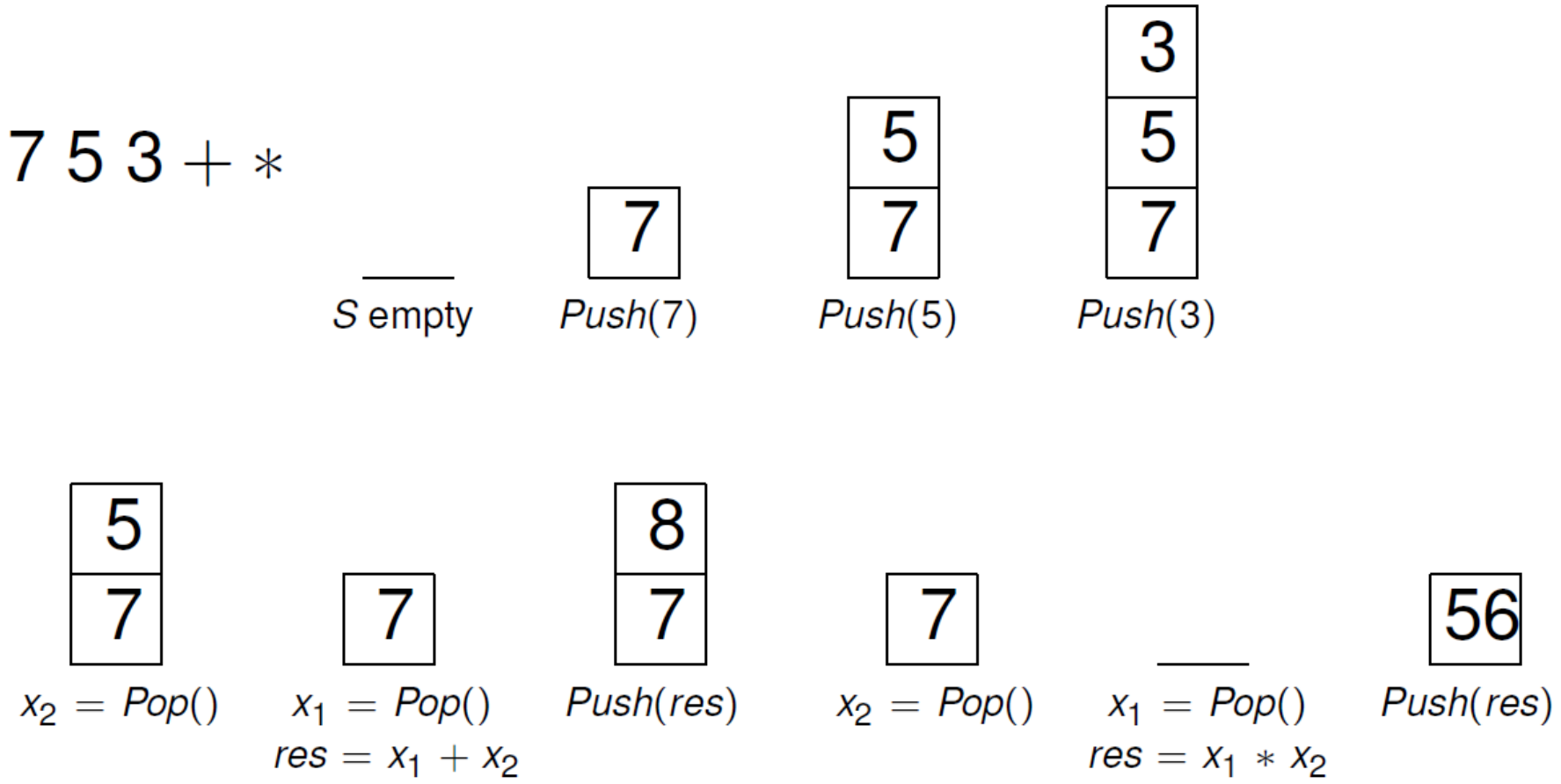
means

$$x * (y/w - z).$$

► Given an expression in postfix notation we can use a stack to obtain the result.

Stacks: Applications

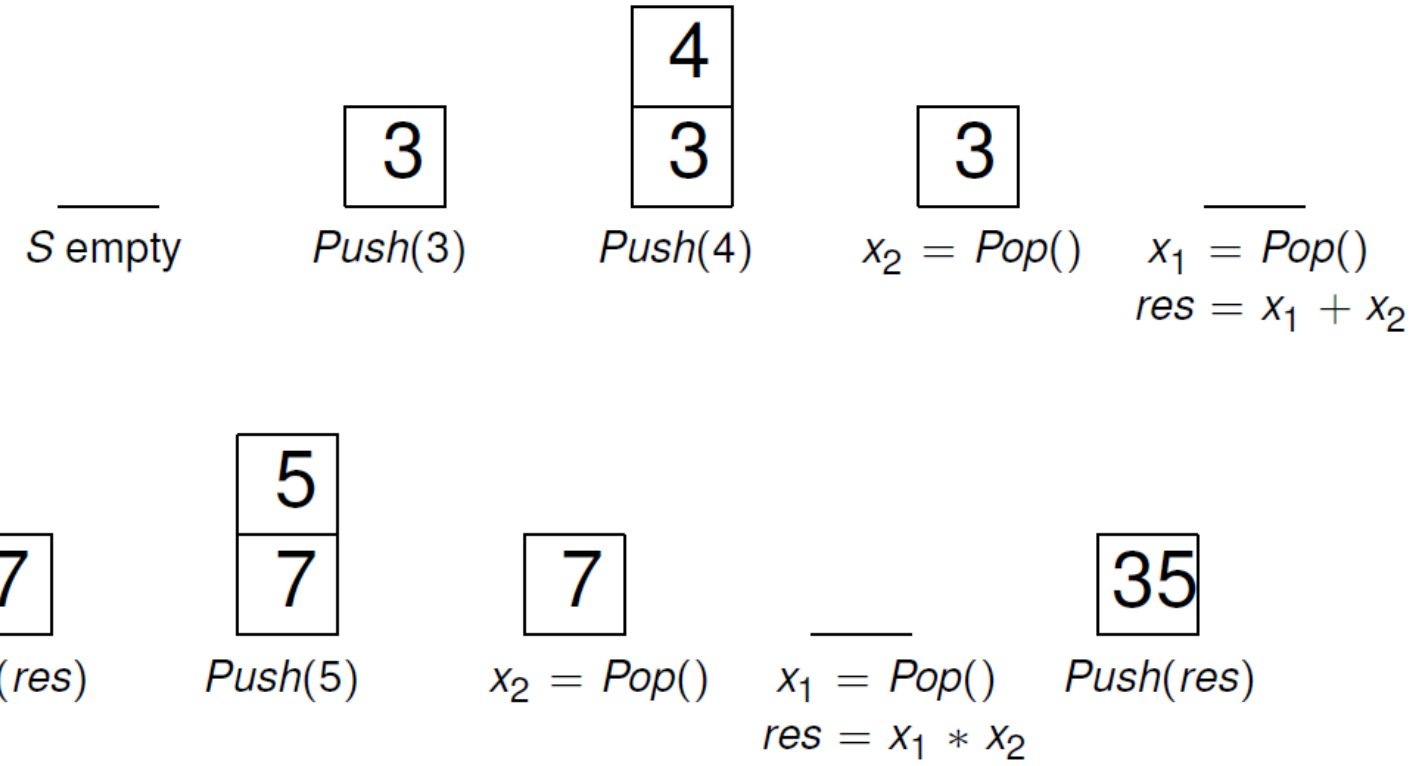
Problem : Reverse Polish notation



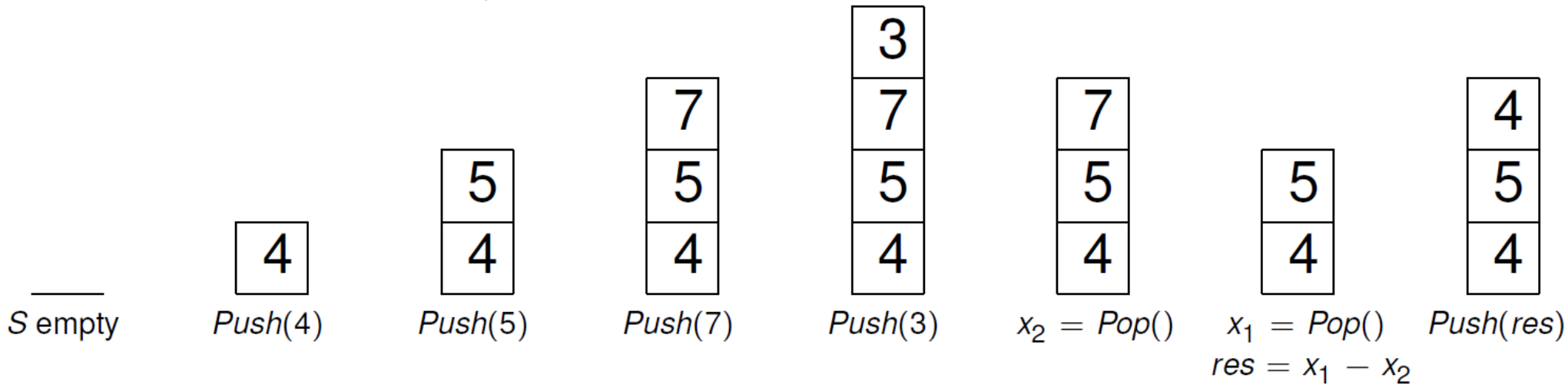
Stacks: Applications

Problem : Reverse Polish notation

$3\ 4\ +\ 5\ *$



$4\ 5\ 7\ 3\ -\ *\ +\ 3\ /$


S empty
Push(4)
Push(5)
Push(7)
Push(3)
x2 = Pop()
x1 = Pop()
res = x1 − x2
Push(res)
x2 = Pop()
x1 = Pop()
res = x1 ∗ x2
Push(res)
x2 = Pop()
x1 = Pop()
res = x1 + x2
Push(res)
Push(3)
x2 = Pop()
x1 = Pop()
res = x1 / x2
Push(res)

Stacks: Applications

Problem : Reverse Polish notation

► Using postfix notation, there is no operator precedence, the order (from left to right) of the operators in the postfix notation is the order in which they are executed.

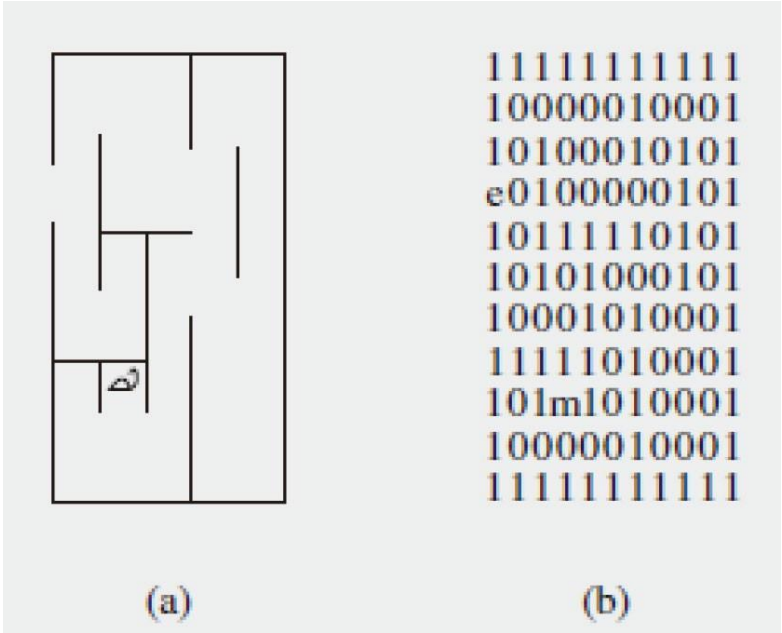
► Postfix notation really mimics the way in which we go about computing an expression given in the “usual” (infix) notation.

► Postfix notation removes the need for brackets (provided we don’t confuse the subtraction operator and the “-" symbol that appears in a negative number, so correct formatting and processing of numbers is important if read by a machine).

Stacks: Applications

Problem : Exiting a Maze.

Consider the problem of a trapped mouse that tries to find its way to an exit in a maze.



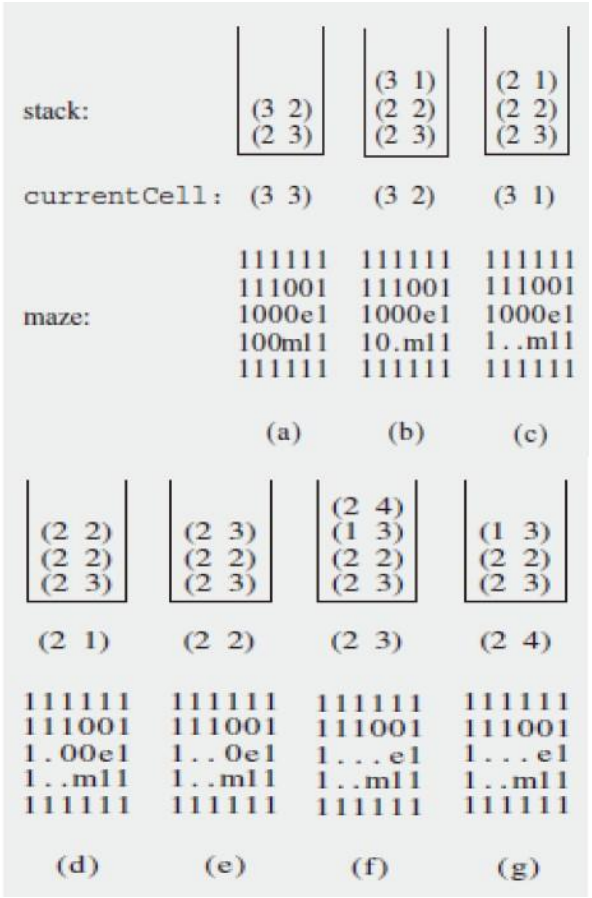
Write a program to solve the above problem by using stacks.

Stacks: Applications

Problem : Exiting a Maze.

Consider the problem of a trapped mouse that tries to find its way to an exit in a maze.

```
 012345

0111111

1111001

21000e1

3100m11

4111111
```



Write a program to solve the above problem by using stacks.

```
exitMaze()
initialize stack, exitCell, entryCell, currentCell = entryCell;
while currentCell is not exitCell
        mark currentCell as visited;
        push onto the stack the unvisited neighbors
of currentCell;
         If stack is empty
               failure;
          else pop off a cell from the stack and make it
currentCell;
success;
```

Data Structures: Queues

- A queue is a *First-In, First-Out* (FIFO) data structure, like queues you're used to in the real world.
- Objects can be inserted (at the rear) into a queue at *any* time, but only the element at the front of the queue can be removed.
- An example of a queue is a list of *jobs* sent to a printer for printing waiting lines
- We say that elements *enter* the queue at the *rear* and are *removed* from the front.

Queue ADT

A queue ADT supports the following fundamental methods:

- *enqueue(Obj)*: inserts object *Obje*t the *rear* of the queue.
- *dequeue()*: removes and returns the object from the *front* of the queue. An error occurs if the queue is *empty*.

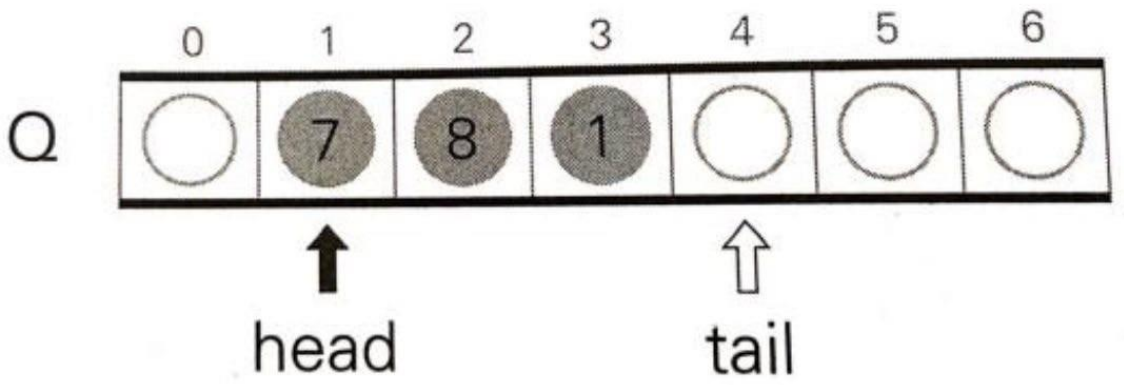
In addition to *enqueue()* and *dequeue()* there are also:

- *size(): Return the number of objects in the queue.*
- *isEmpty()*: Returns true if queue is empty, and false otherwise.
- isFull(): Returns true if queue is full, and false otherwise.
- front(): Return, but do not remove, the object at the front of the queue. An error is returned if the queue is empty.

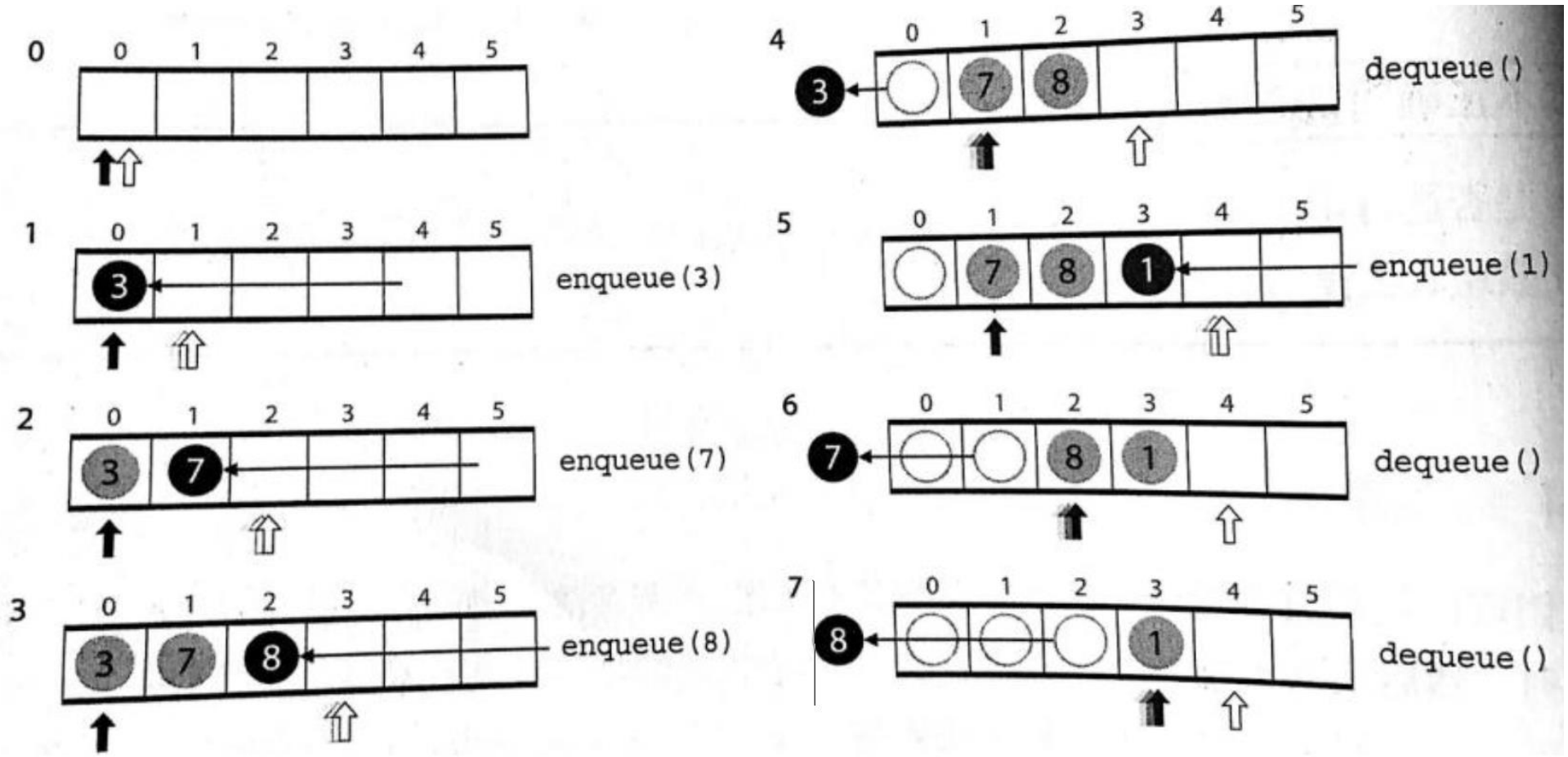
Queue ADT

A queue ADT supports the following fundamental methods:

- *enqueue(Obj)*: inserts object *Obje*t the *rear* of the queue.
- *dequeue()*: removes and returns the object from the *front* of the queue. An error occurs if the queue is *empty*.



Queue ADT

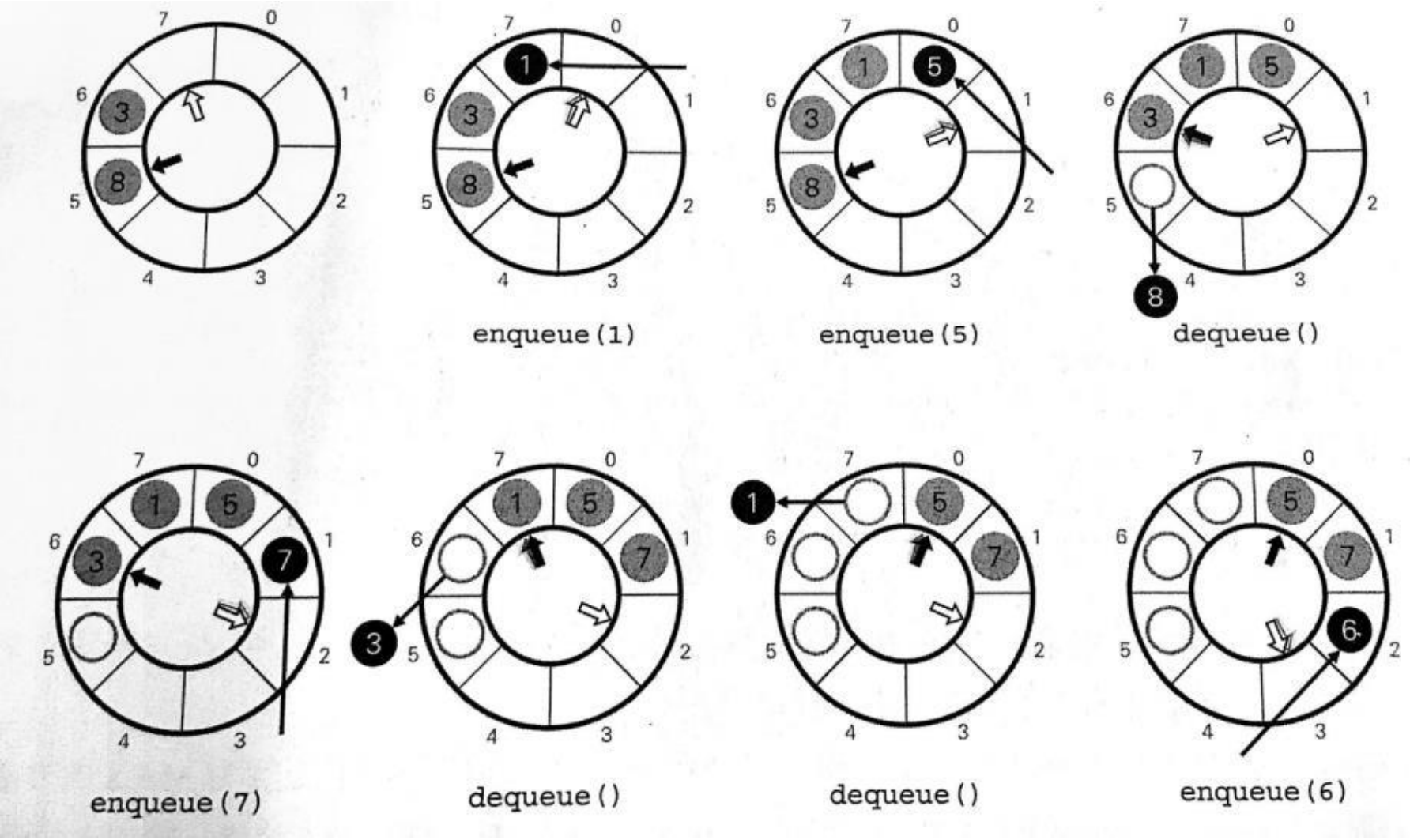


Moving to the end of the array

->out of memory

Queue ADT

A circular queue holding the values 3 and 8



Queue and Multiprogramming

► Multiprogramming achieves a limited form of parallelism.
► Allows multiple tasks or threads to be run at the same time.
► For example, a computer (or program) might use multiple threads, one of which is responsible for catching mouse clicks, whilst another may be responsible for drawing animations on the screen.
► When designing programs (or operating systems) that use multiple threads, we must not allow a single thread to monopolize the CPU.
► One solution is to use a queue to allocate CPU time to threads in a round robin protocol. Here we can use a queue that holds the collection of threads. The thread at the beginning of the queue is removed, allocated some portion of the CPU time, and then replaced at the rear of the queue.

Data Structures: List ADT

- A *list* is a collection of items, with each item being stored in a *node* which contains a *data* field and a *pointer* to the next element in a list.
- Data can be inserted *anywhere* in the list by inserting a new node into the list and reassigning pointers.
- A list ADT supports: *referring*, *update* (both *insert* and *delete*) as well as *searching* methods.
- We can implement the list ADT as either a *singly-*, or *doubly-*linked list.

List ADT: Referring methods

A list ADT supports the following *referring* methods:

- *first()*: Return position of first element; error occurs if list S is empty.
- *last()*: Return the position of the last element; error occurs if list S is empty.
- *isFirst(p)*: Return **true** if element *p* is first item in list, **false** otherwise.
- *isLast(p)*: Return **true** is element *p* is last element in list, **false** otherwise.
- *before(p)*: Return the position of the element in S preceding the one at position *p*; error if *p* is first element.
- *after(p)*: Return the position of the element in S following the one at position *p*; error if *p* is last element.

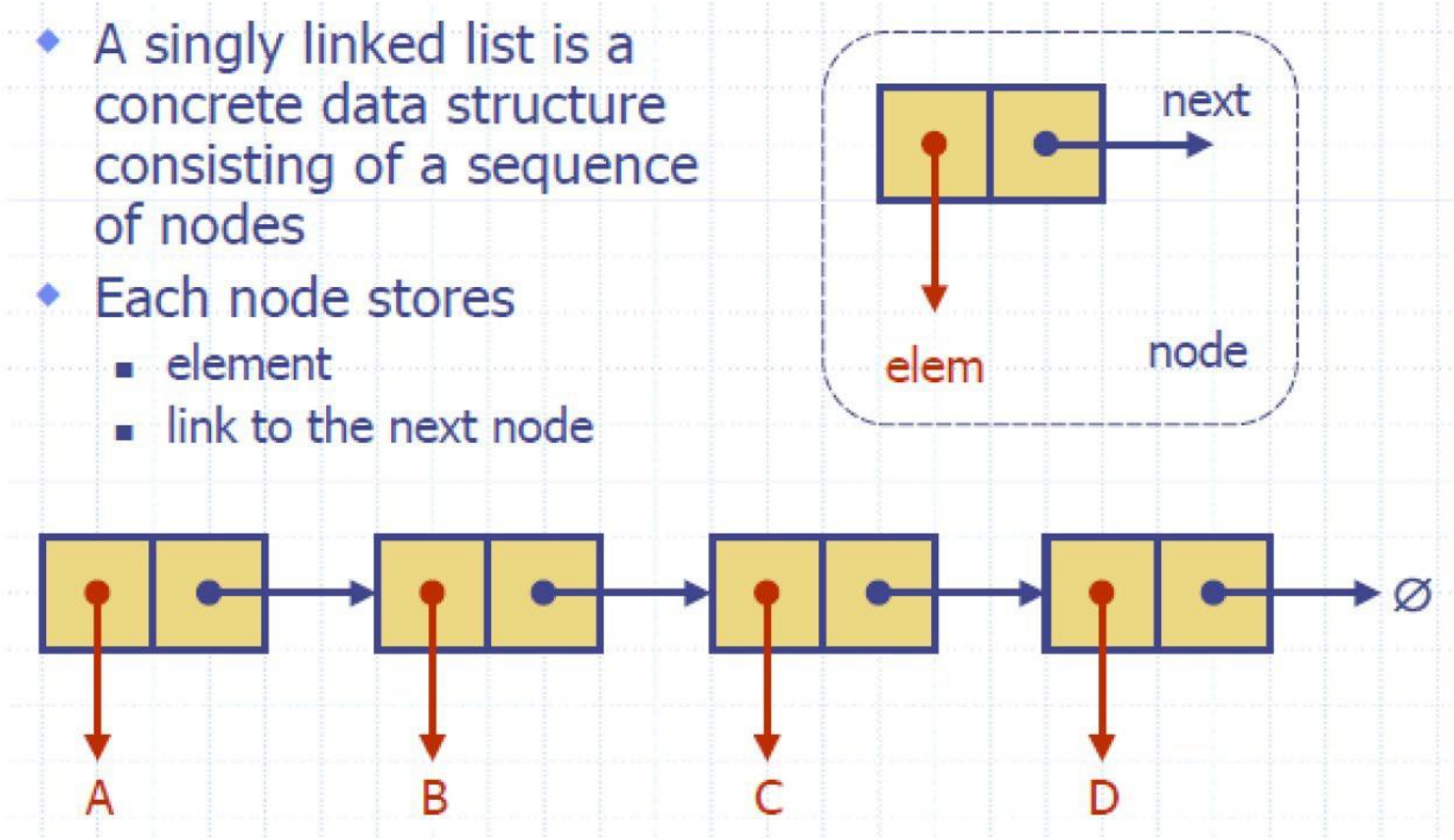
List ADT: Update methods

A list ADT supports the following *update* methods:

- *replaceElement(p,e)*: *p* - position, *e* - element.
- *swapElements(p,q)*: *p,q* - positions.
- *insertFirst(e)*: *e* - element.
- *insertLast(e)*: *e* - element.
- *insertBefore(p,e)*: *p* - position, *e* - element.
- *insertAfter(p,e)*: *p* - position, *e* - element.
- *remove(p)*: *p* - position.

Linked List

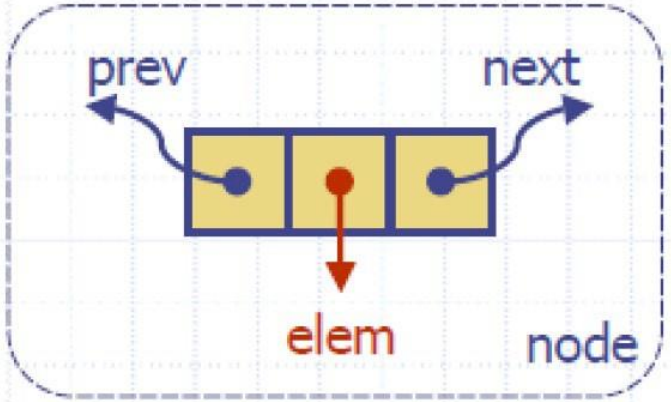
▶ A *node* in a *singly*-linked list stores a *next* link pointing to next element in list (**null** if element is last element).

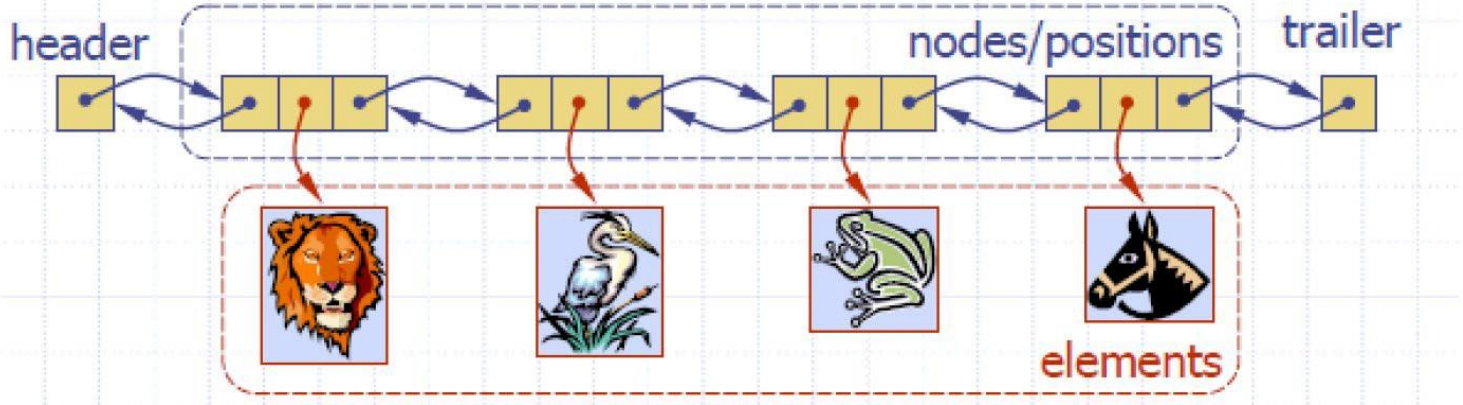


Linked List

A *node* in a *doubly*-linked list stores two links: a *next* link, pointing to the next element in list, and a *prev* link, pointing to the previous element in the list.

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes



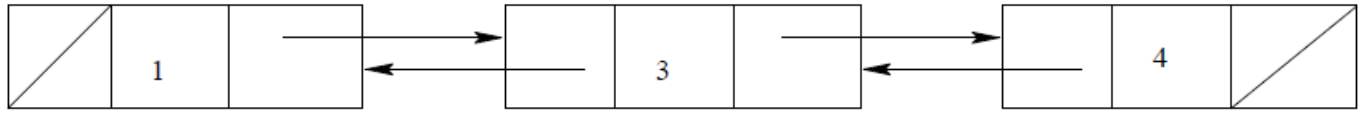


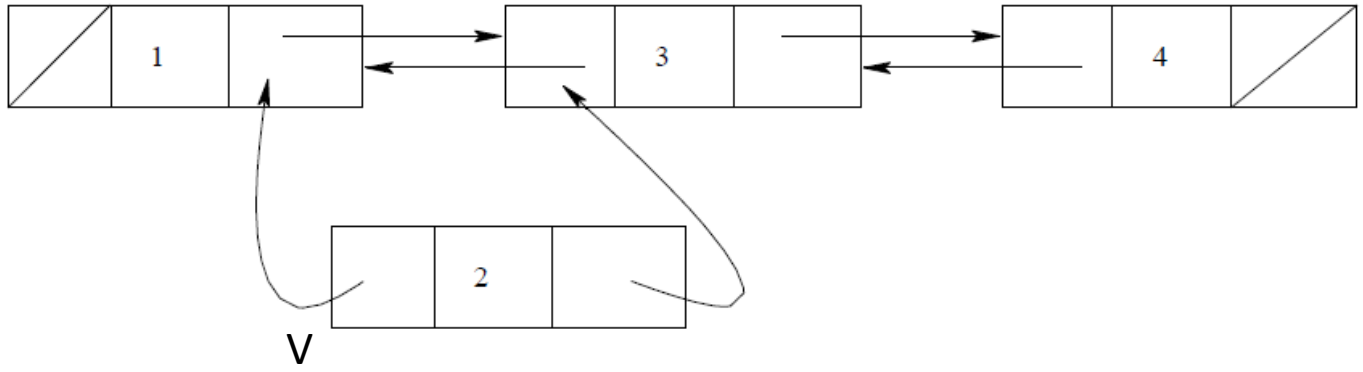
From now on, we will concentrate on doubly-linked lists.

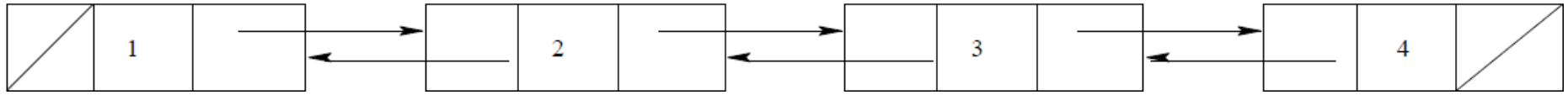
List update: Element insertion

How to insert an element?

Example: *insertAfter(1,e)*





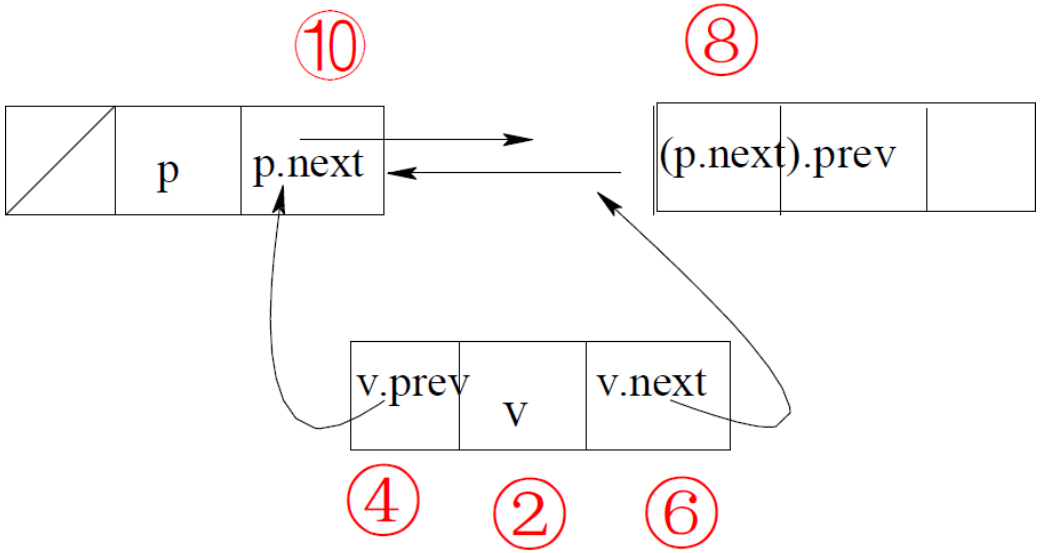


List update: Element insertion

Pseudo-code for *insertAfter(p,e)* :

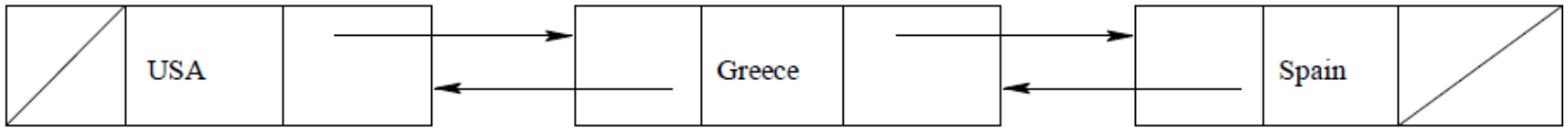
INSERTAFTER(*p,e*)

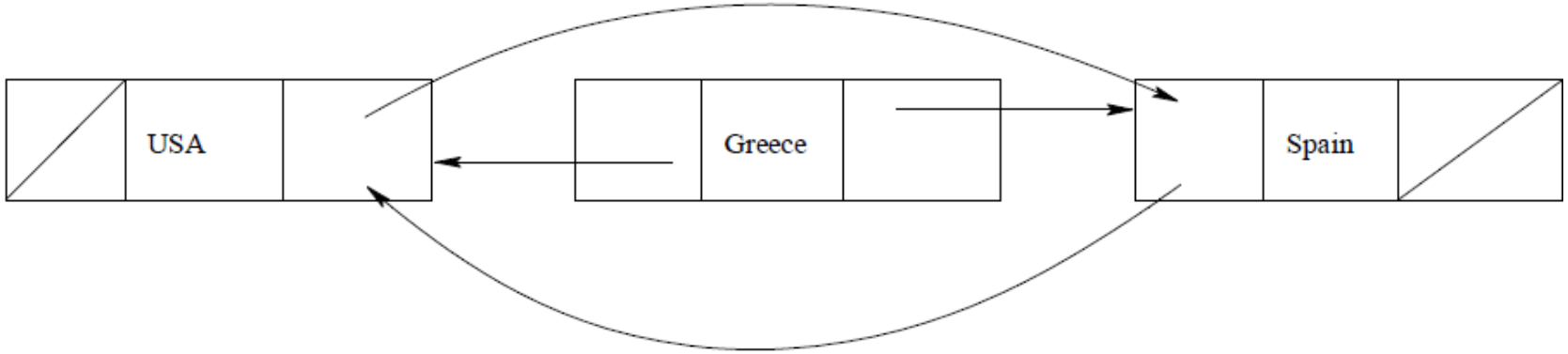
```
    //Create a new node v
2   v.element ← e
    //Link v to its predecessor
4   v.prev ← p
    //Link v to its successor
6   v.next ← p.next
    //Link p's old successor to v
8   (p.next).prev ← v
    //Link p to its new successor v
10  p.next ← v
11  return v
```



List update: Element removal

Example: *remove(2)*



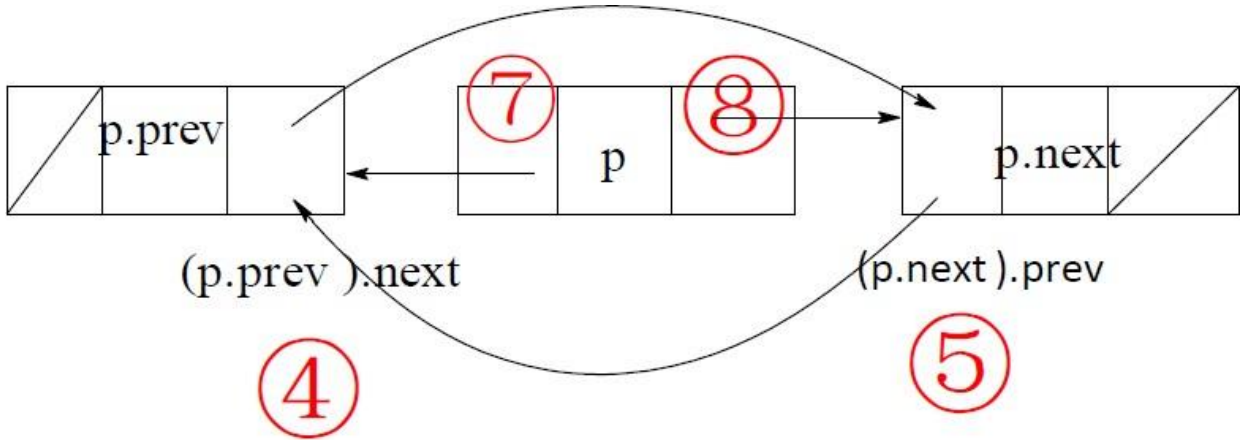


List update: Element removal

The pseudo-code for *remove(p)*:

REMOVE(*p*)

```
    //Assign a temporary variable to hold return value
2   t ← p.element
    //Unlink p from list
4   (p.prev).next ← p.next
5   (p.next).prev ← p.prev
    //invalidate p
7   p.prev ← null
8   p.next ← null
9   return t
```



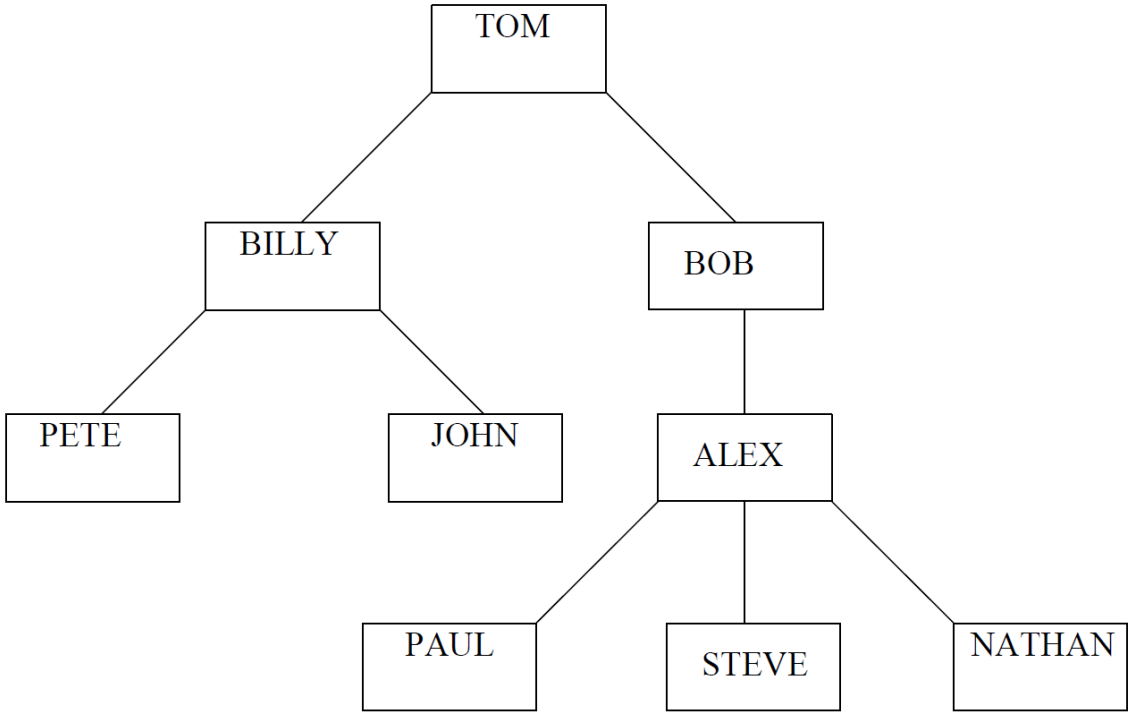
INT202
Complexity of Algorithms
Data Structures

XJTLU/SAT/INT

SEM2 AY2021-2022

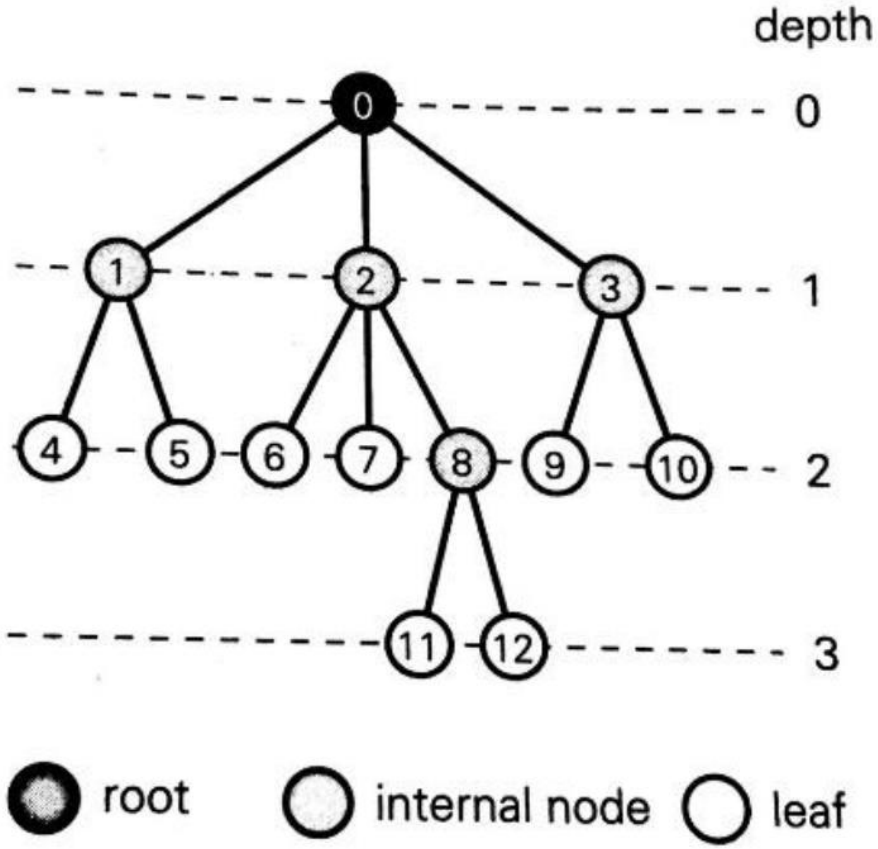
Data Structures: Rooted Trees

- A rooted tree, *T*, is a set of nodes which store elements in a parent-child relationship.
- *T* has a special node, *r* , called the *root* of *T*.
- Each node of *T* (excluding the root node *r*) has a *parent* node.



Rooted trees: terminology

- If node *u* is the *parent* of node *v*, then *v* is a *child* of *u*.
- Two nodes that are *children* of the same *parent* are called *siblings*.
- A node is a *leaf* (external) if it has no *children* and *internal otherwise*
- A tree is *ordered* if there is a *linear* ordering defined for the *children* of each internal node (i.e. an internal node has a distinguished first child, second child, etc).



Binary Trees

- A *binary tree* is a rooted ordered tree in which every node has *at most two children*.
- A binary tree is *proper* if each internal node has *exactly two children*.
- Each *child* in a binary tree is labeled as either a *left child* or a *right child*.

Tree ADT Methods

Tree ADT access methods:

- *root()*: return the root of the tree.
- *parent(v)*: return parent of *v*.
- *children(v)*: return links to *v*'s children.

Tree ADT query methods:

- *isInternal(v)*: test whether *v* is internal node.
- *isExternal(v)*: test whether *v* is external node.
- *isRoot(v)*: test whether *v* is the root.

Tree ADT Methods (cont.)

Tree ADT generic methods:

- *size()*: return the number of nodes in the tree.
- *elements()*: return a list of all elements.
- *positions()*: return a list of addresses of all elements.
- *swapElements(u,v)*: swap elements stored at positions *u* and *v*.
- *replaceElements(v,e)*: replace element at address *v* with element *e*.

Depth of a node in a tree

▶ The *depth* of a node, *v*, is number of ancestors of *v*, excluding *v* itself. This is easily computed by a recursive function.

DEPTH(*T,v*)

1 **if** *T.isRoot*(*v*)

2 **then return** 0

3 **else return** 1 + DEPTH(*T, T.parent*(*v*))

Depth of a node in a tree

▶ The *depth* of a node, *v*, is number of ancestors of *v*, excluding *v* itself. This is easily computed by a recursive function.

DEPTH(*T,v*)

1 **if** *T.isRoot*(*v*)

2 **then return** 0

3 **else return** 1 + DEPTH(*T, T.parent*(*v*))

Height of a tree

▶ The *height* of a tree is equal to the maximum depth of an external node in it. The following pseudo-code computes the height of the subtree rooted at v.

```
HEIGHT(T,v)
1 if ISEXTERNAL(v )
2    then return 0
3    else
4         h = 0
5         for each w ∈ T.CHILDREN(v)
6              do
7                   h = MAX(h, HEIGHT(T,w))
8         return 1 + h
```

Tree Traversal

▶ In a *traversal*, the goal is for the algorithm to visit all the nodes in the tree in some order and perform an operation on them.

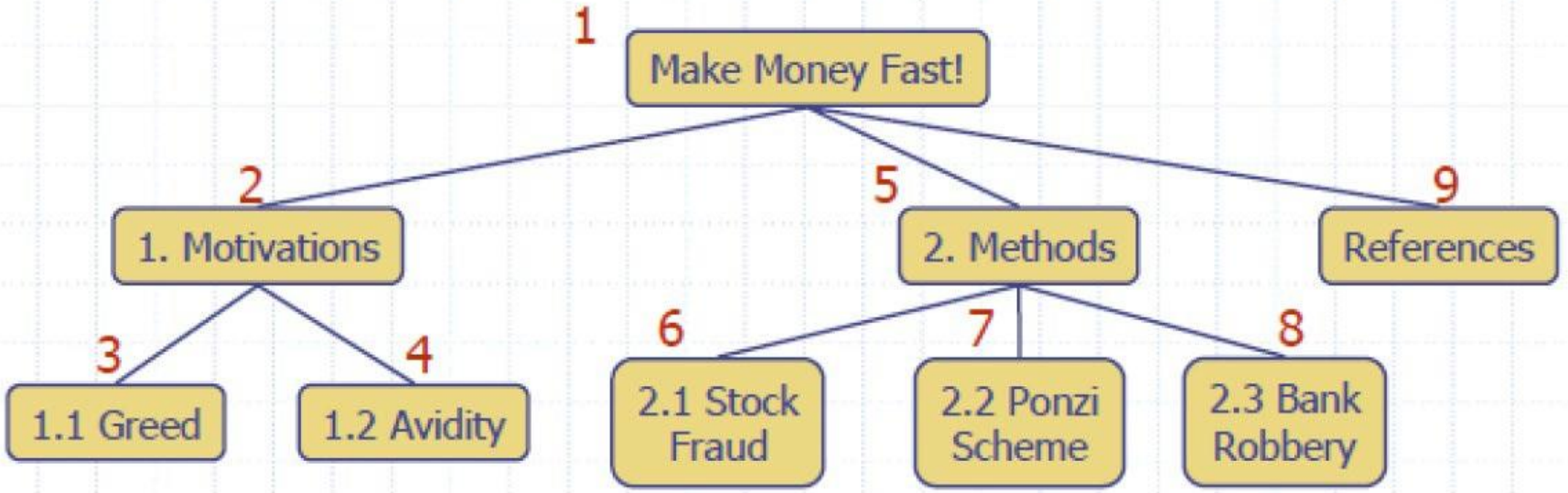
▶ Traversal and Searching

▶ Binary trees have three kinds of traversals: preorder, postorder, and inorder.

Preorder traversal in trees

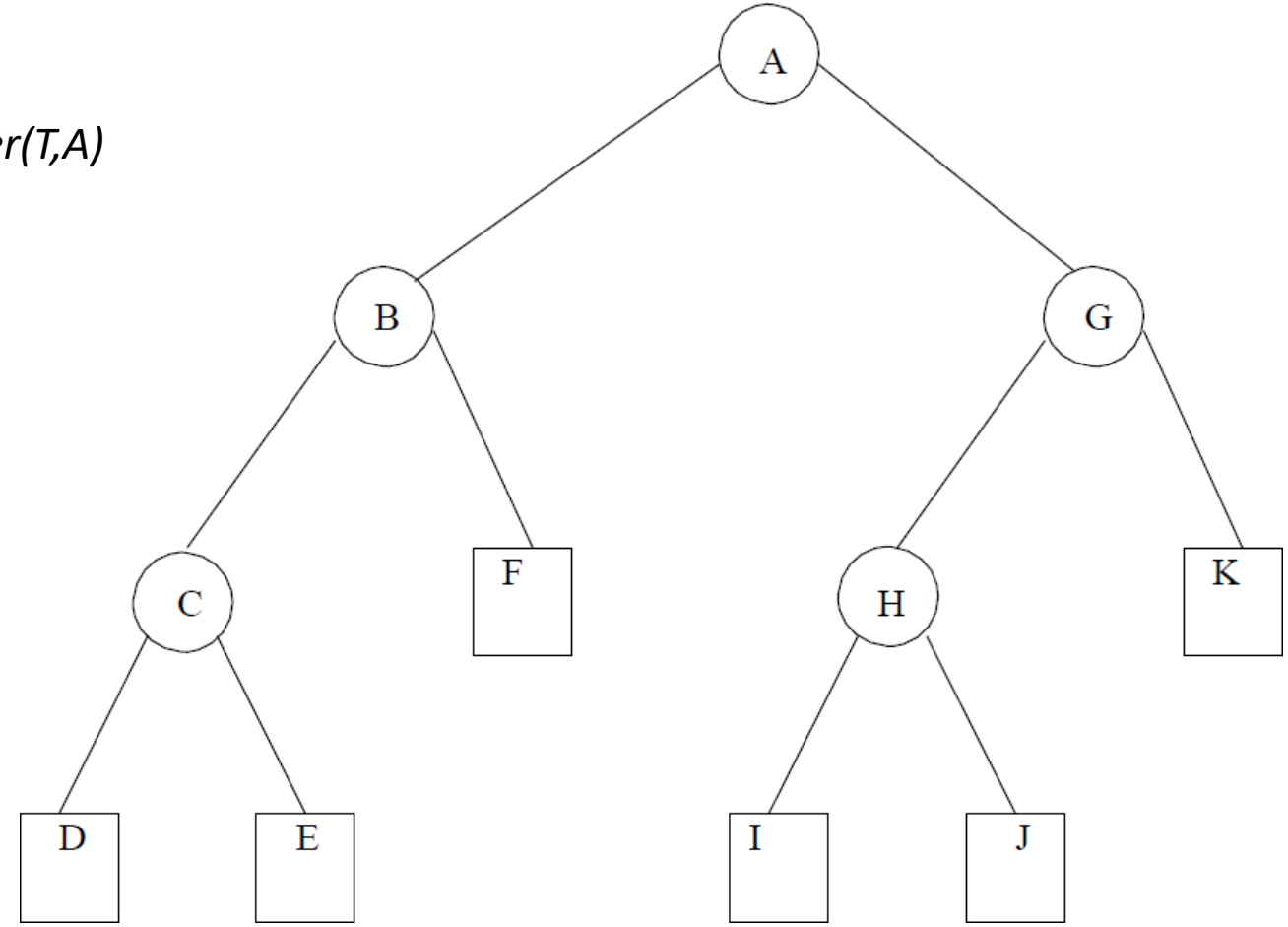
- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)

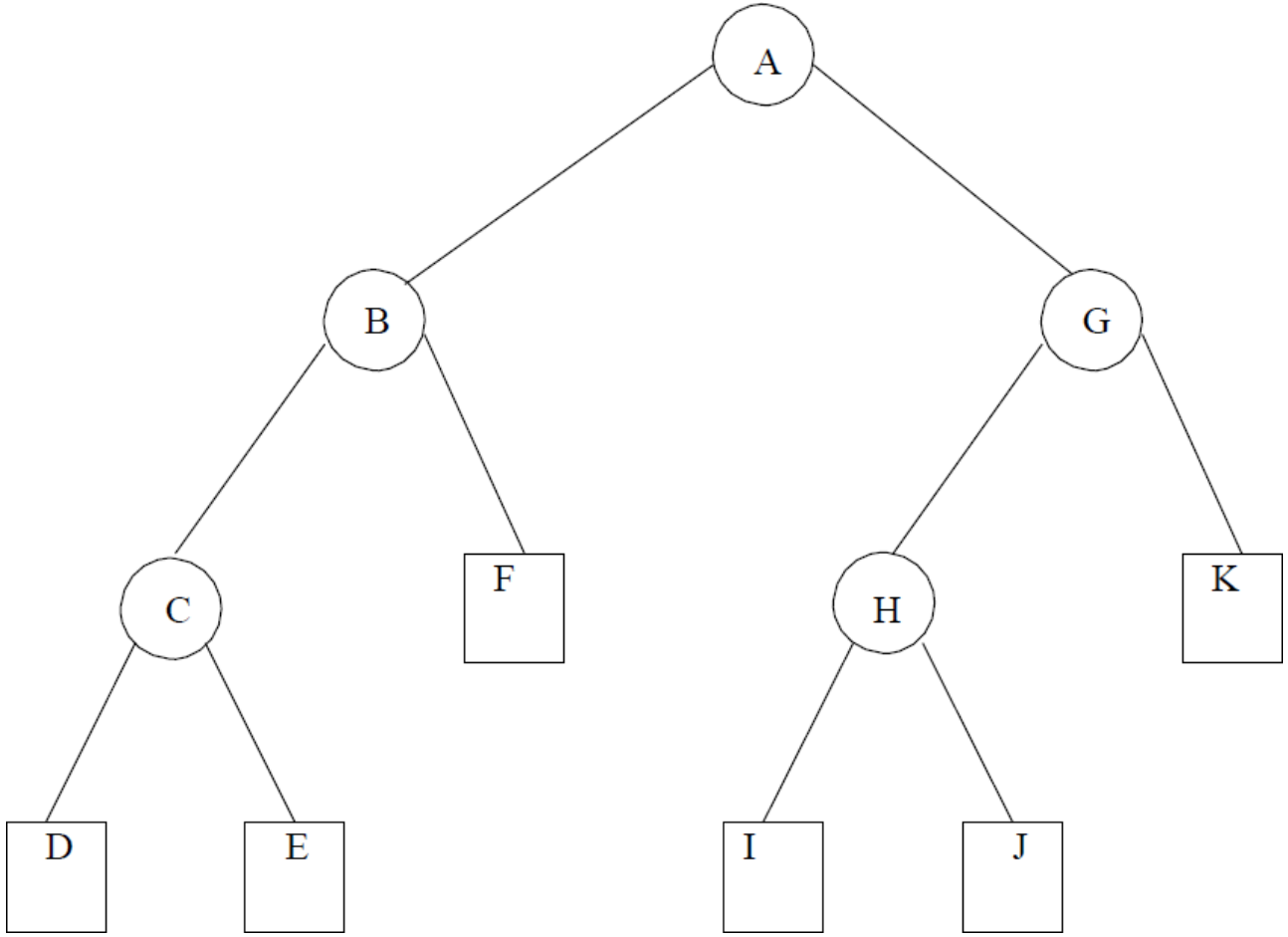


Preorder traversal in trees (cont.)

A call to *preorder(T,A)*



Preorder traversal in trees (cont.)

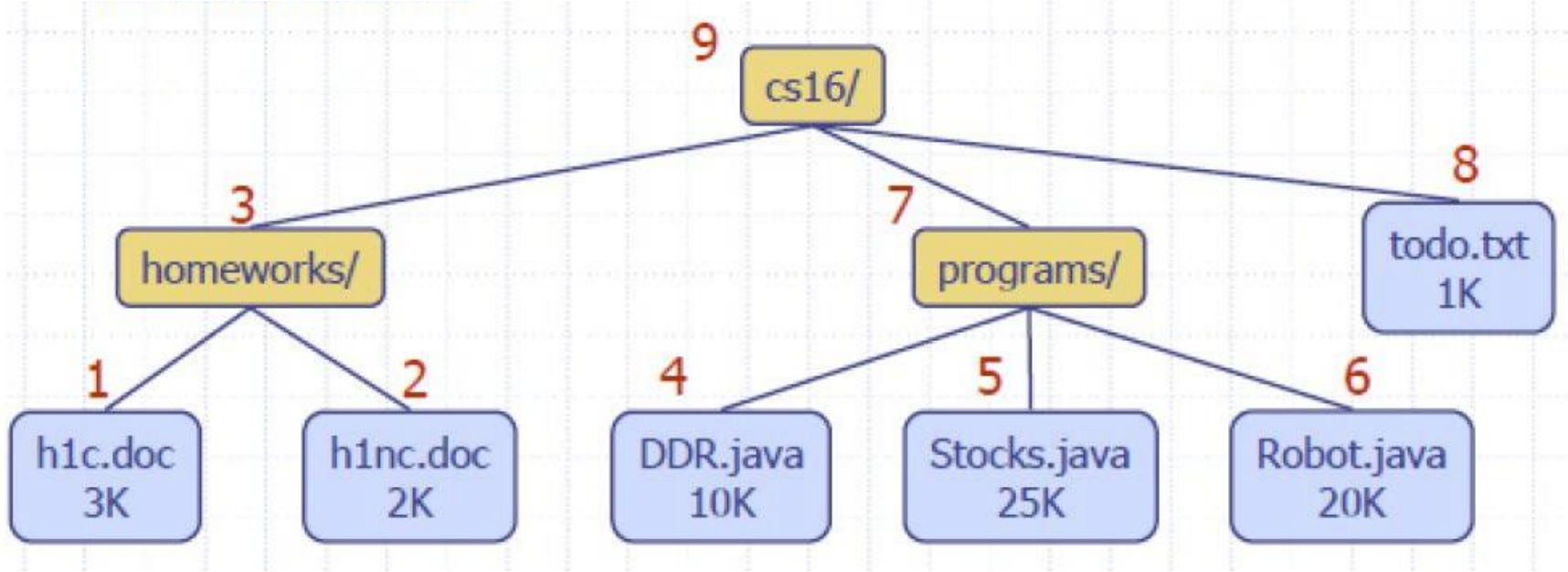


- A call to *preorder(T,A)* would produce: A,B,C,D,E,F,G,H,I,J,K.

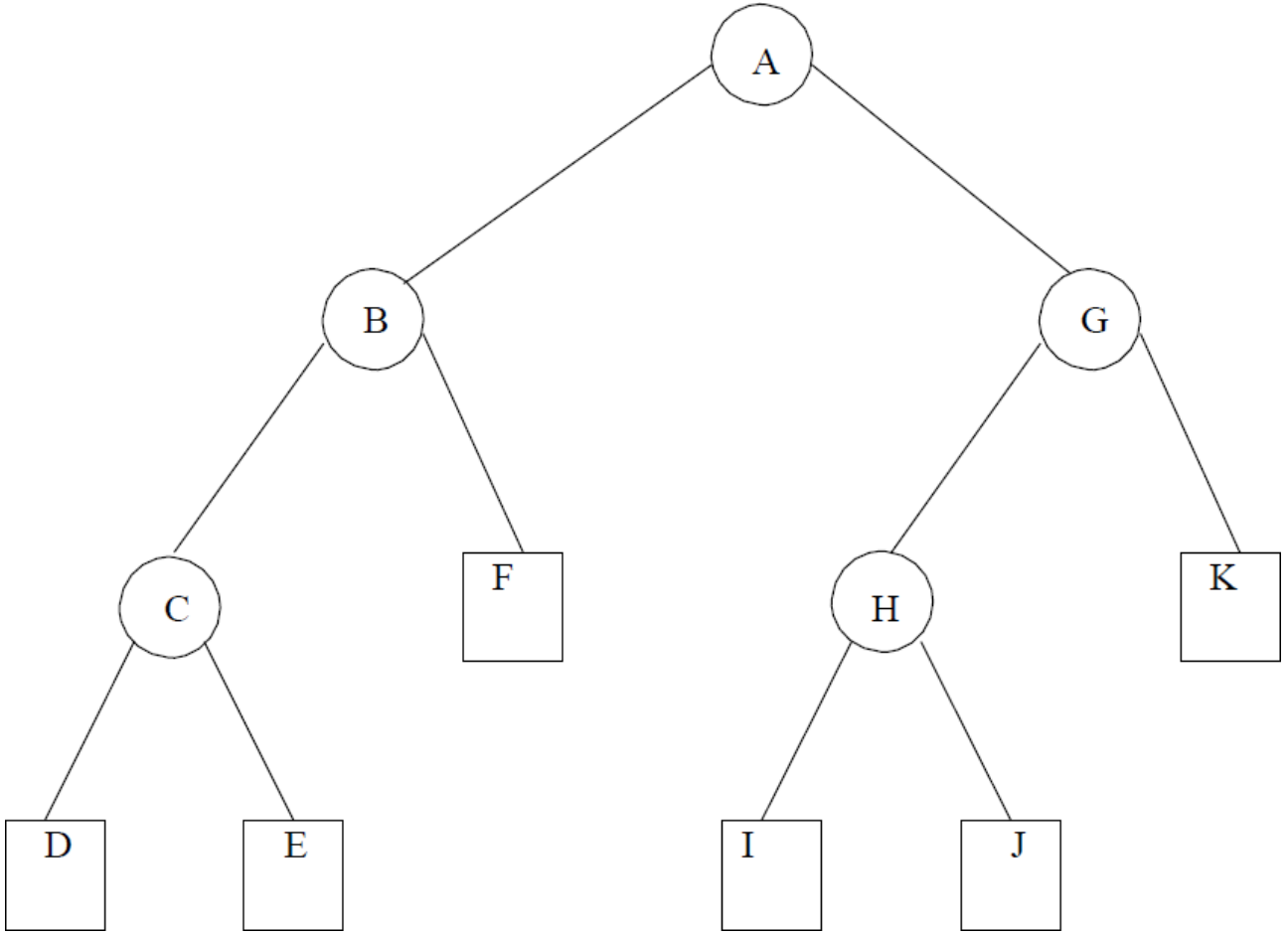
Postorder traversal of trees

- In a postorder traversal, a node is visited after its descendants.
- Application: compute space used by files in a directory and is sub-directions.

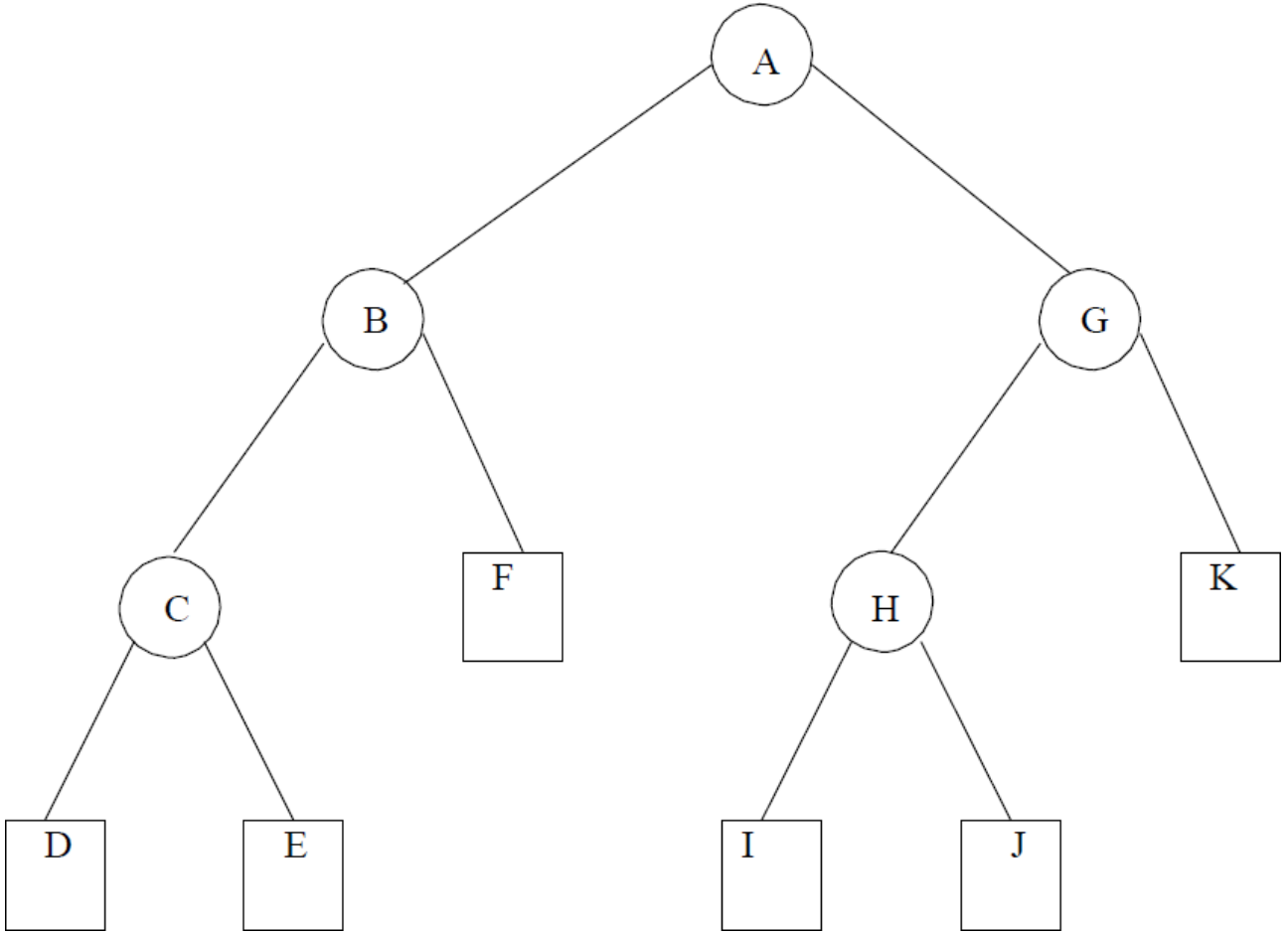
```
Algorithm postOrder(v)
    for each child w of v
        postOrder (w)
    visit(v)
```



Postorder traversal in trees (cont.)



Postorder traversal in trees (cont.)

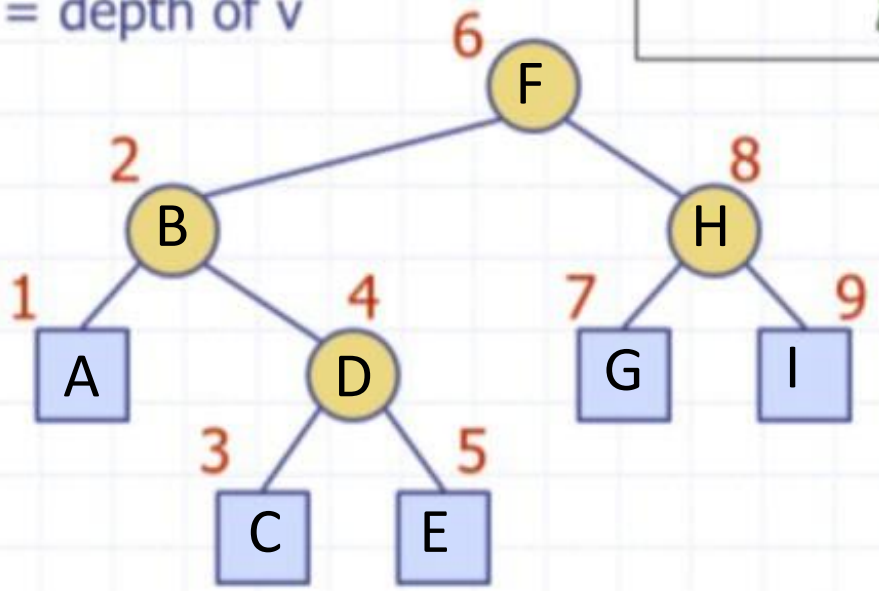


- A call to postorder(T,A) would produce: D,E,C,F,B,I,J,H,K,G,A.

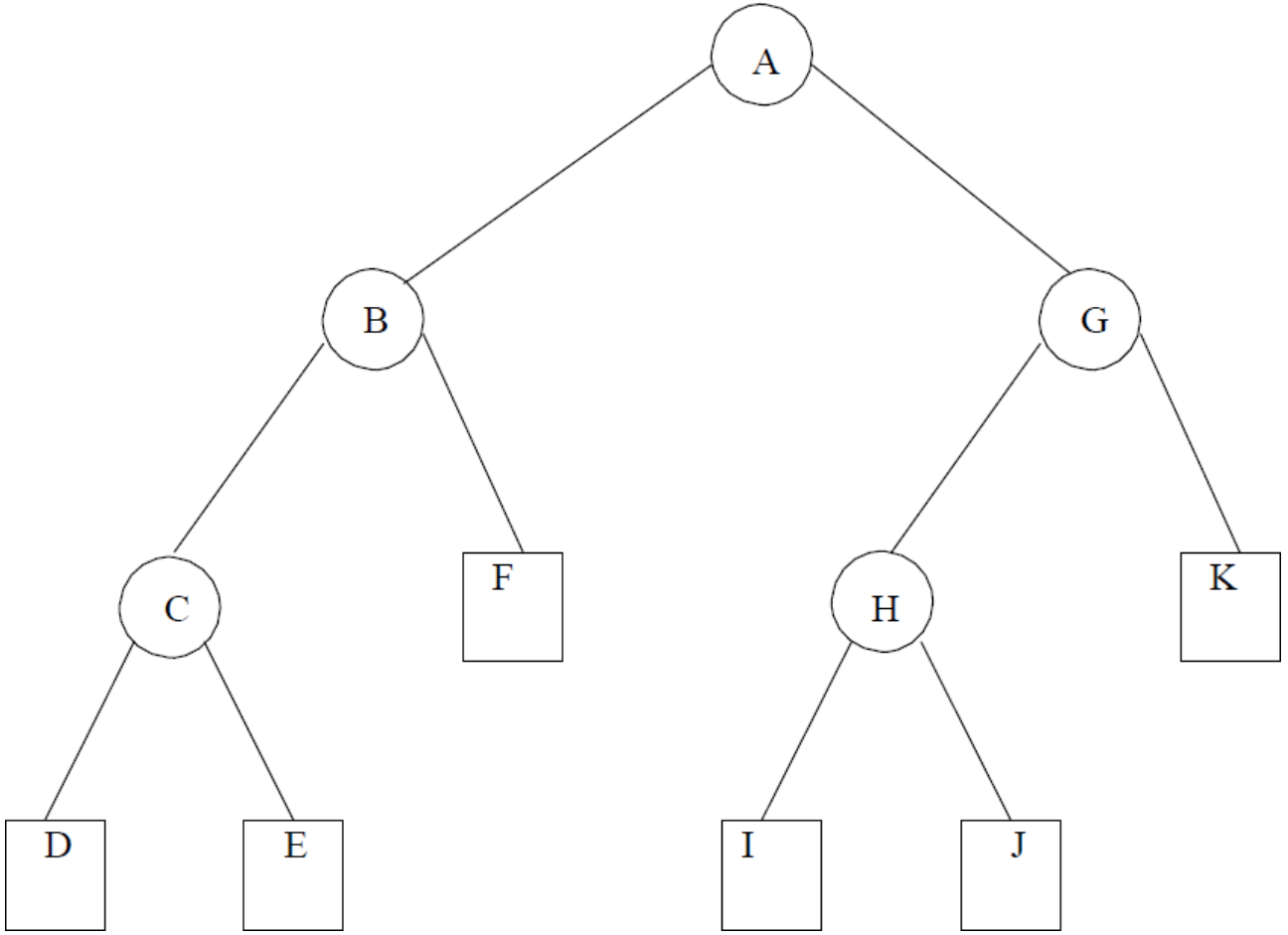
Inorder traversal in trees

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder(v)*
  if *hasLeft (v)*
    inOrder (left (v))
  visit(v)
  if *hasRight (v)*
    inOrder (right (v))



Inorder traversal in trees (cont.)

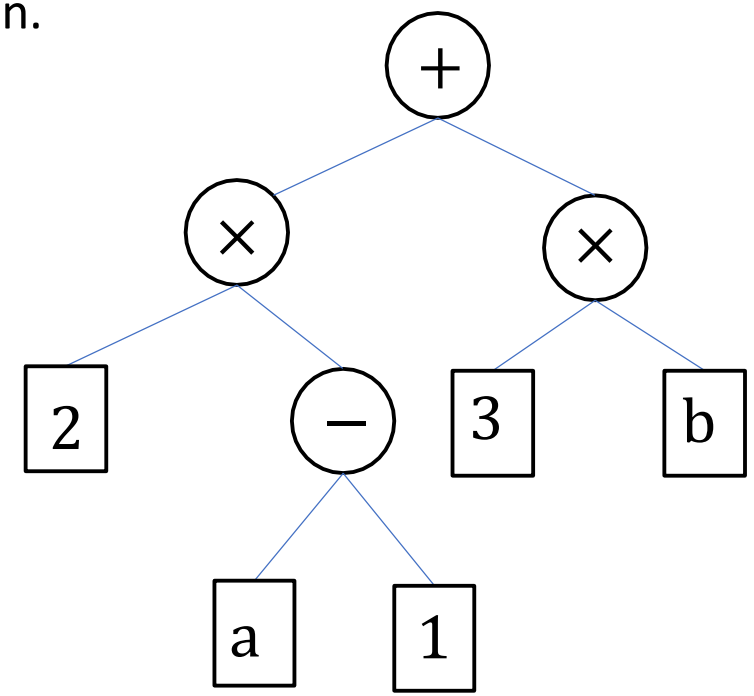


- A call to *inorder*(T,A) would produce: D,C,E,B,F,A,I,H,J,G,K.

Example: Parsing arithmetic expressions

Binary tree associated with an arithmetic expression

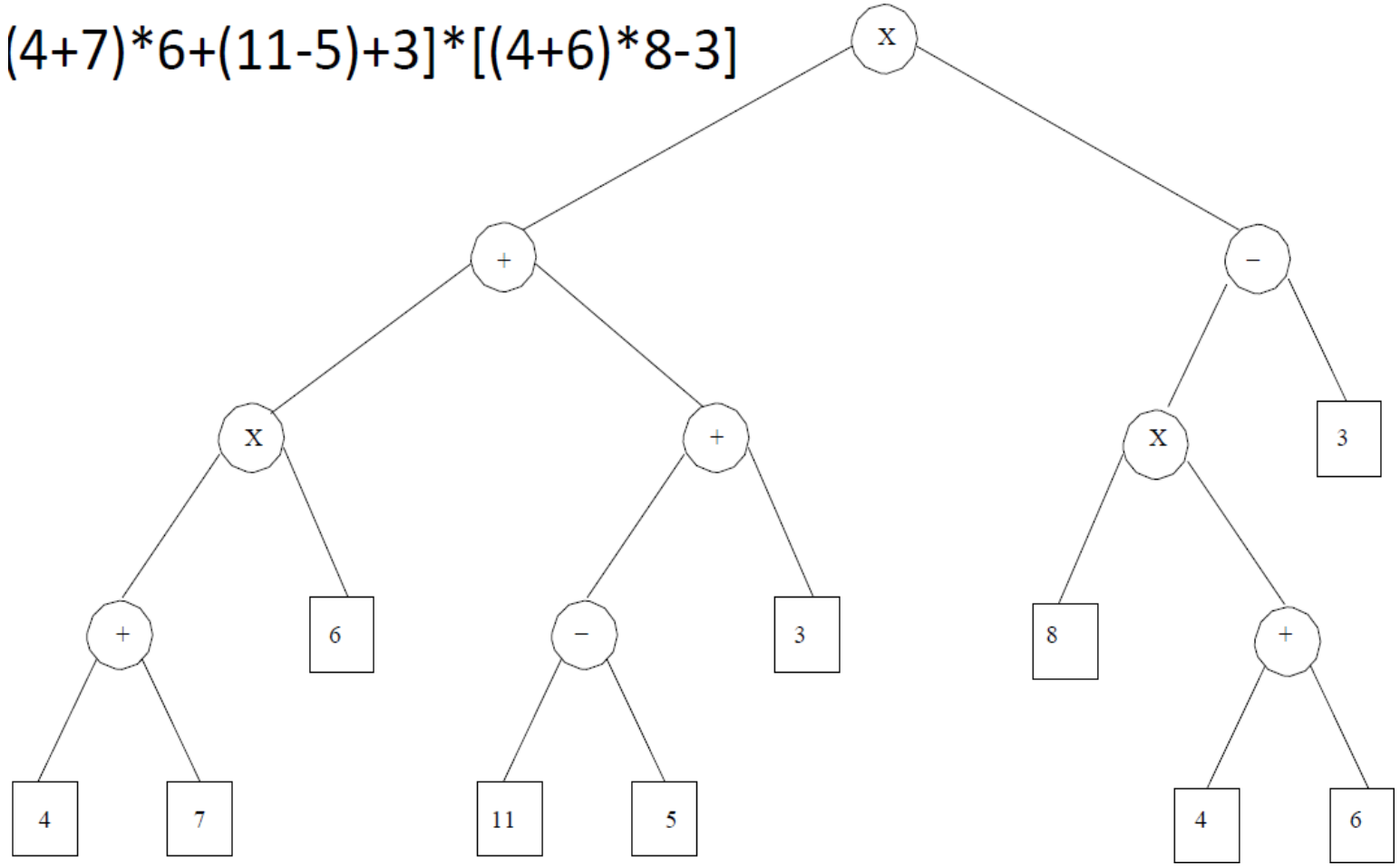
- Each *external node* is a *variable* or a *constant*.
- Each *internal node* defines an *arithmetic operation* on its two children.



Traversing the tree in inorder gives the valid *postfix* expression that represents this arithmetic calculation: (2*(a-1)+(3*b))

Example: Parsing arithmetic expressions

Y=[(4+7)*6+(11-5)+3]*[(4+6)*8-3]

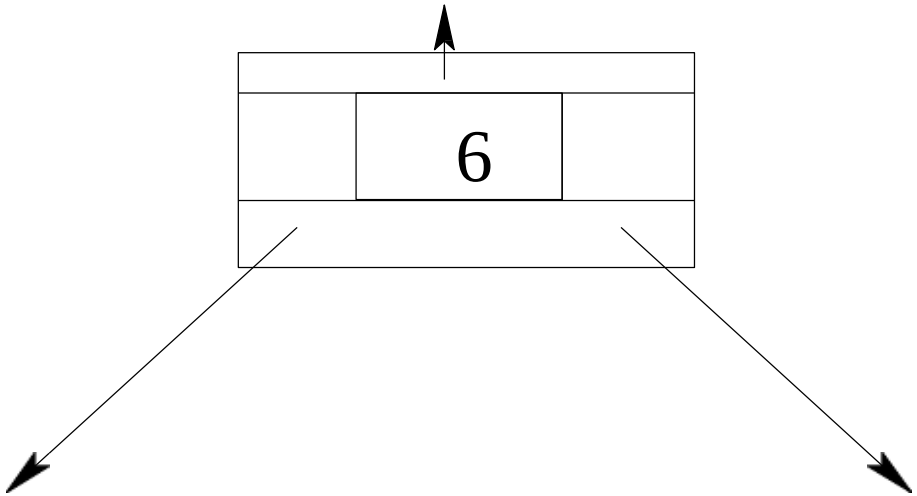


INT202
Complexity of Algorithms
Search Algorithms

XJTLU/SAT/INT
SEM2 AY2021-2022

Data structures for trees

- Linked structure: each node v of T is represented by an object with references to the element stored at v and positions of its parents and children.



Data structures for trees

- For rooted trees where each node has at most t children, and is of bounded depth, you can store the tree in an array A.

- Consider, for example, a binary tree. The root is stored in A[0]. The (possibly) two children of the root are stored in A[1] and A[2]. The two children of A[1] are stored in A[3] and A[4], and the two children of A[2] are in A[5] and A[6], and so forth.

Data structures for trees

- In general, the two children of node A[i] are in $A[2 * i + 1]$ and
- $A[2 * i + 2]$.
- The parent of node A[i] (except the root) is the node in position $A[\lfloor (i - 1)/2 \rfloor]$.
- This can be generalized to the case when each node has at most t children

Ordered Data

- We often wish to store data that is ordered in some fashion (typically in numeric order, or alphabetical order, but there may be other ways of ordering it).
- Here we study a few methods for storing such ordered data, and maintaining the order as we add more data or remove it.
- In later lectures we will discuss efficient methods for *sorting* data into such an ordered collection. In these lecture notes we talk about methods for maintaining ordered data *as we store it* (which is different from sorting the data *after* we have already received the aggregated data as input).

Ordered Dictionary

In an ordered dictionary, we wish to perform the usual dictionary operation:

- ▷ *findElement(k)*: position *k*
- ▷ *insertElement(k,e)*: position *k* , element *e*
- ▷ *removeElement(k)*: position *k*

An *ordered* dictionary maintains an order relation for its elements, where the items are ordered by their keys.

Sorted Tables

If a dictionary *D*, is *ordered*, we can store its items in a vector, *S*, by *non-decreasing* order of keys. (This generally assumes that we're not going to add more items into the dictionary.)

Storing the dictionary in a vector in this fashion allows faster searching than if *S* is stored using a linked list.

A lookup table is a dictionary implemented by means of a sorted sequence

- We store the items of the dictionary in an array-based sequence, sorted by key
- We use an external comparator for the keys

Binary Search

▷ Accessing an element of S (in an array-based representation of size n) by its rank.

▷ The item at rank i has a key no smaller than keys of the items of ranks $1,\ldots,i-1$ and no larger than keys of the items of ranks $i+1, i+2,\ldots,n$.

The *search* is done on a decreasing range of the elements in S.

▷ Looking for k in S, the current range of S we consider is defined as a pair of ranks: *low* and *high*.

▷ Initially, $low = 1$ and $high = n$.

- ▷ $key(i)$ denotes the key at rank i.
- ▷ $elem(i)$ denotes the element at rank i.

Binary Search (cont.)

In order to *decrease* the size of the range we compare k to the key of the median, *mid*, of the range, i.e.,

$$mid = \lfloor (low + high)/2 \rfloor$$

Three cases are possible:

- $k = key(mid)$, the search is completed successfully.
- $k < key(mid)$, search continued with $high = mid - 1$.
- $k > key(mid)$, search continued with $low = mid + 1$.

Binary Search - Algorithm

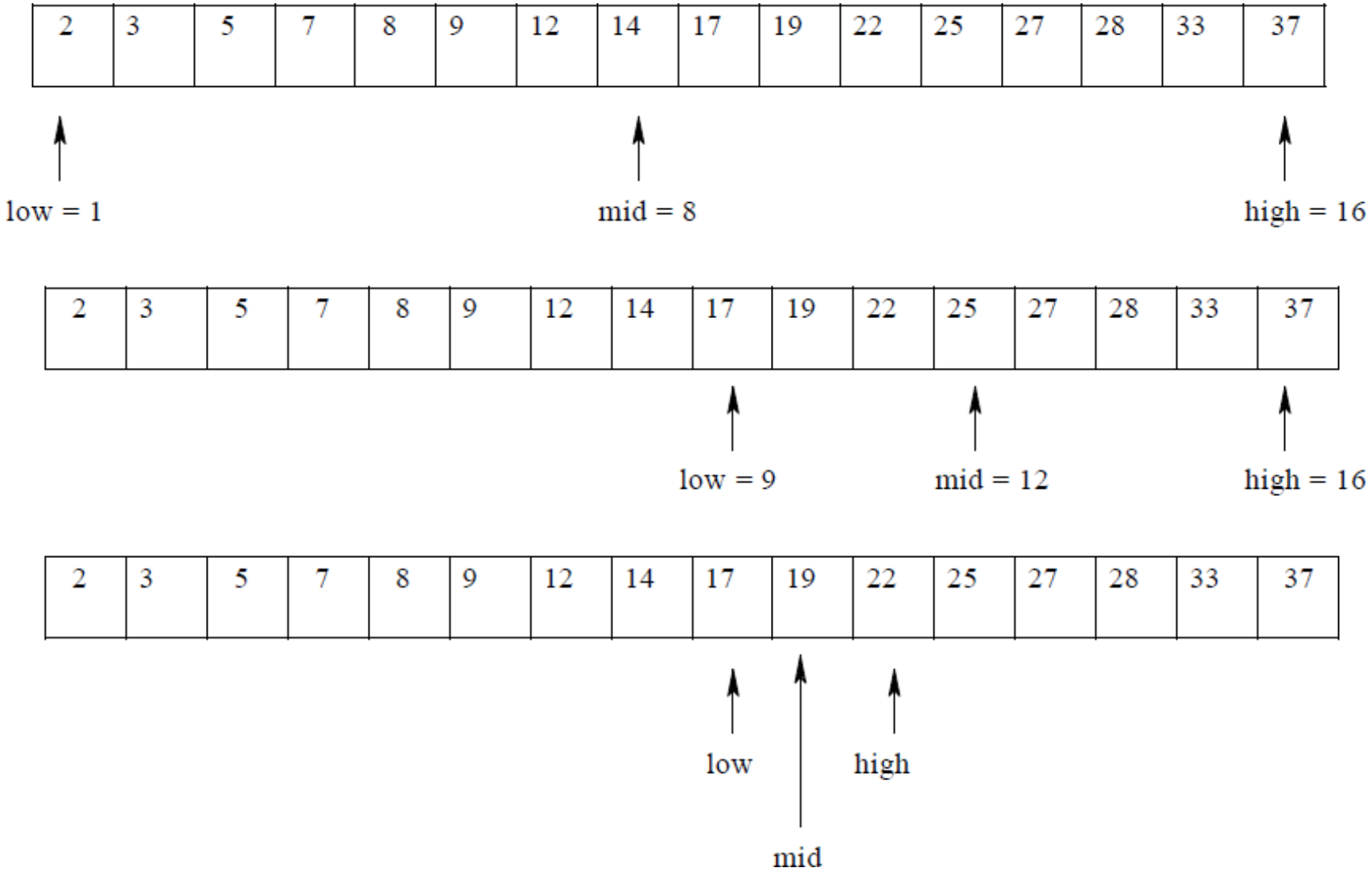
Here's a recursive search algorithm.

BINARY**S**EARCH(*S, k,low,high*)

```
   //Input is an ordered array of elements.
   //Output: Element with key k if it exists, otherwise an error.
1  if low > high
2     then return NO_SUCH_KEY                          // Not present
3  else
4     mid ←mid = ⌊(low + high)/2⌋
5      if k = key(mid)
6         then return elem(mid)                        //Found it
7       elseif k < key(mid)
8          then return BINARYSEARCH(S, k,low,mid-1)    // try bottom 'half'
9       else return BINARYSEARCH(S,k,mid + 1,high)     //try top 'half'
```

Binary Search - Example

- **B**INARY**S**EARCH(*S*,19,1,*size*(*S*))



Complexity of Binary Search

Considering the running time of binary search, we observe that a constant number of operations are executed at each recursive call.

Hence, the running time is proportional to the number of recursive calls performed.

The number of candidate items still to be searched in the array A is given by the value high− low +1.

Moreover, the number of remaining candidates is reduced by at least one half with each recursive call

$$(\text{mid} - 1) - \text{low} + 1 = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor - \text{low} \leq \frac{\text{high} - \text{low} + 1}{2}$$

or

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor \leq \frac{\text{high} - \text{low} + 1}{2}.$$

Complexity of Binary Search

Initially, the number of candidate is n; after the first call to BinarySearch, it is at most n/2; after the second call, it is at most n/4; and so on.

That is, if we let a function, T(n), represent the running time of this method, then we can characterize the running time of the recursive binary search algorithm as follows

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ T(n/2) + b & \text{otherwise} \end{cases}$$

where *b* is a constant that denotes the time for updating *low* and *high* and other overhead.

In general, this recurrence equation shows that the number of candidate items remaining after each recursive call is at most $n/2^i$.

The maximum number of recursive calls performed is the smallest integer such that $n/2^m < 1$. In other words, m > log n.

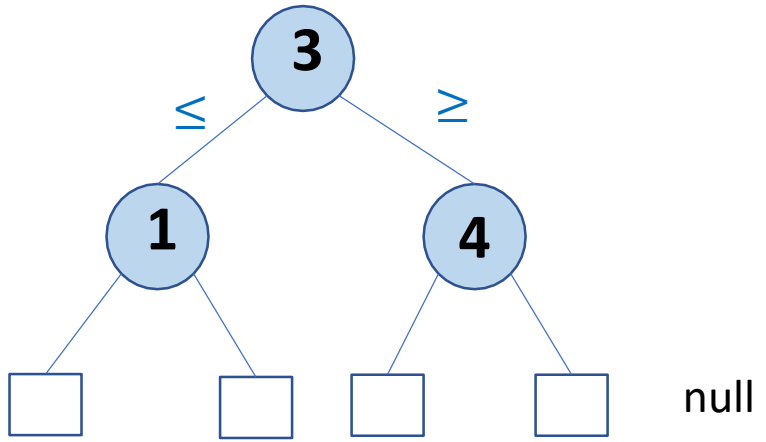
Thus, we have m = $\lfloor \log n \rfloor + 1$, which implies that BinarySearch(A, k, 1, n) runs in O(log n) time.

Binary Search Tree

A Binary Search Tree (*BST*) applies the motivation of binary search to a *tree-based* data structure.

In a *BST* each internal node stores an element, *e* (or, more generally, a key *k* which defines the ordering, and some element *e*).

A *BST* has the property that, given a node *v* storing the element *e*, all elements in the *left subtree* at *v* are *less than or equal to e*, and those in the *right subtree* at *v* are *greater than or equal to e*.



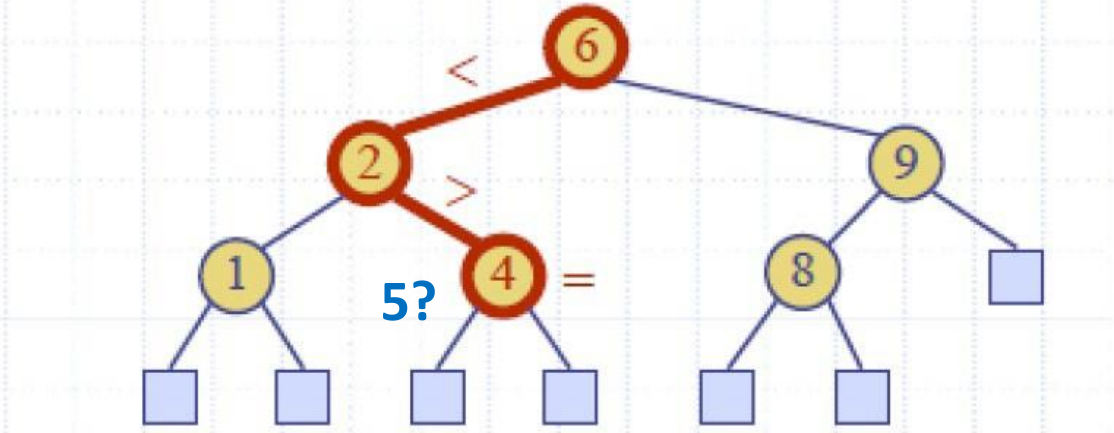
Inorder traversal of a BST: 1,3,4 (non-decreasing order)

Binary Search Tree

- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return NO—SUCH KEY

Example: findElement(4)

```
Algorithm findElement(k, v)
    if T.isExternal (v)
        return NO_SUCH_KEY
    if k < key(v)
        return findElement(k, T.leftChild(v))
    else if k = key(v)
        return element(v)
    else { k > key(v) }
        return findElement(k, T.rightChild(v))
```



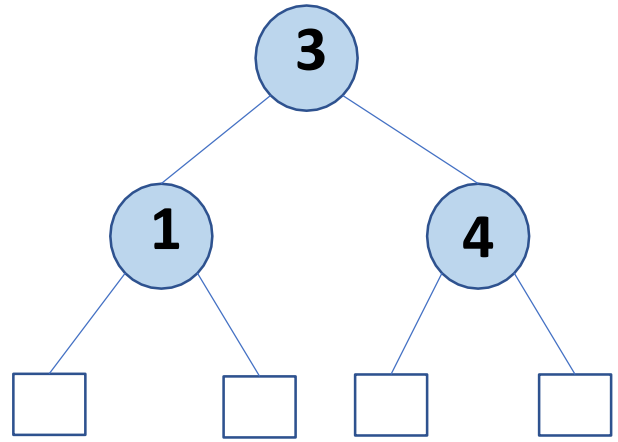
Binary Search Tree

Time complexity

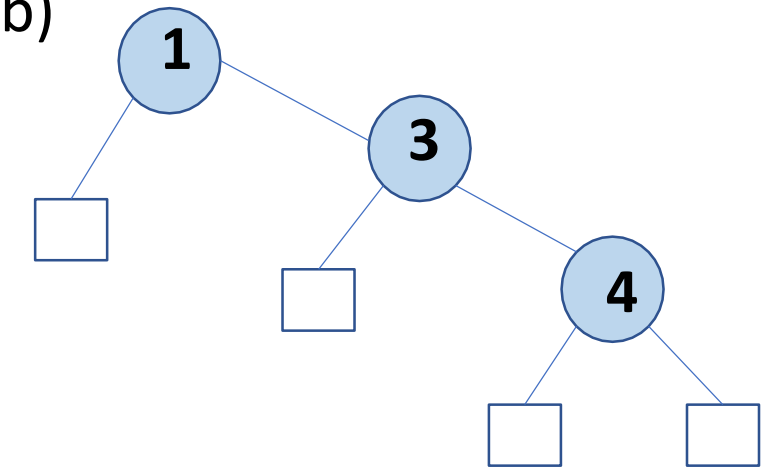
How long dose it takes in a n element tree?

```
Algorithm findElement(k, v)
    if T.isExternal (v)
        return NO_SUCH_KEY
    if k < key(v)
        return findElement(k, T.leftChild(v))
    else if k = key(v)
        return element(v)
    else { k > key(v) }
        return findElement(k, T.rightChild(v))
```

(a)

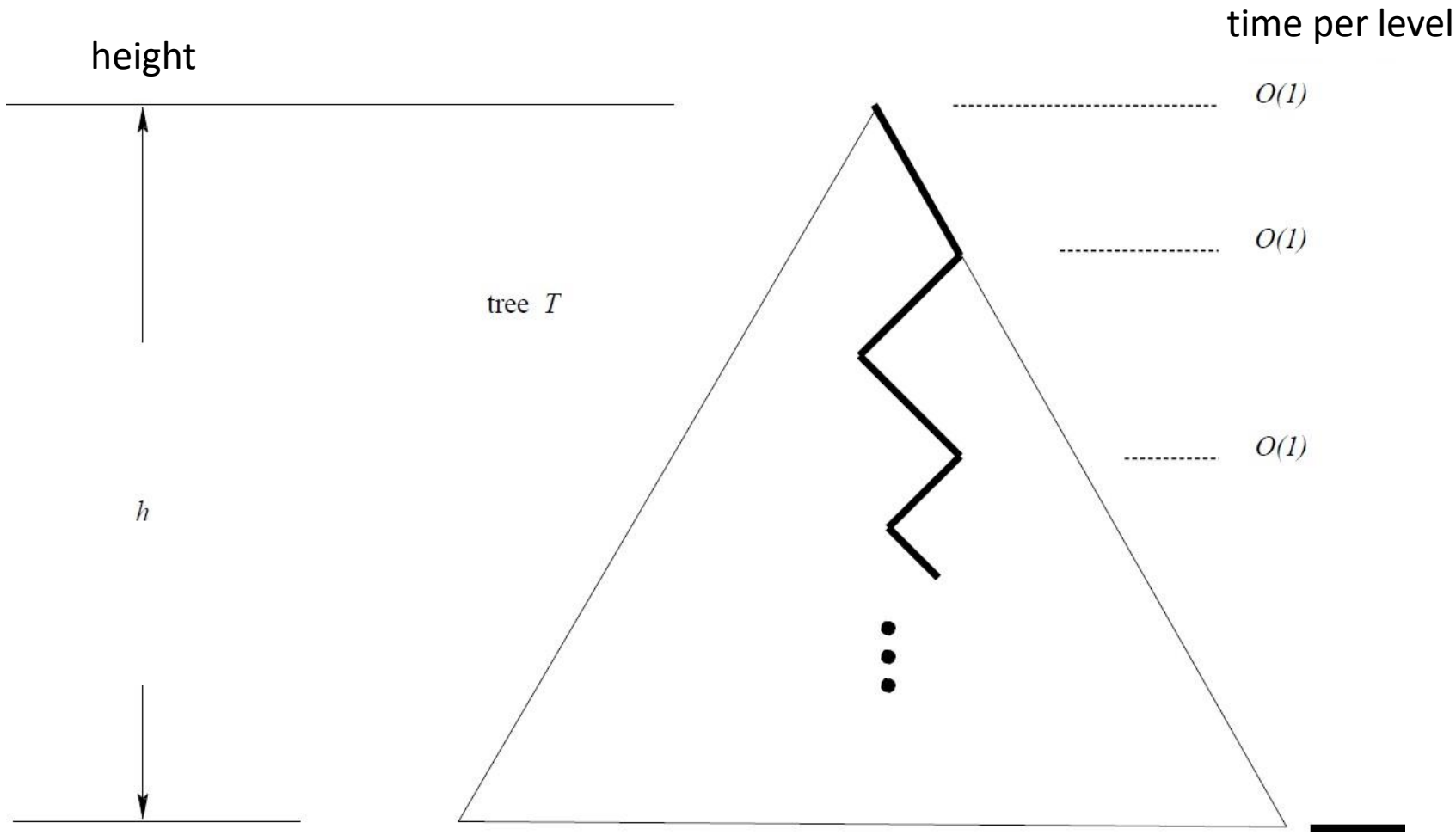


(b)



findElement(4)

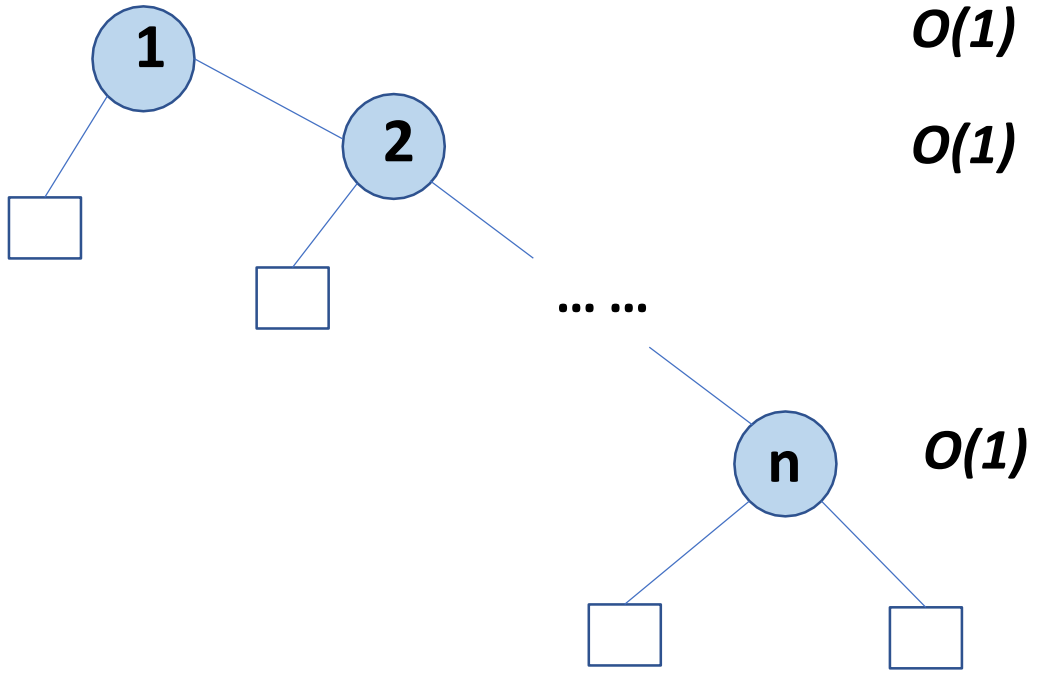
Complexity of searching in a BST



Total time O(h)

As small as log n or as large as n

findElement (n)



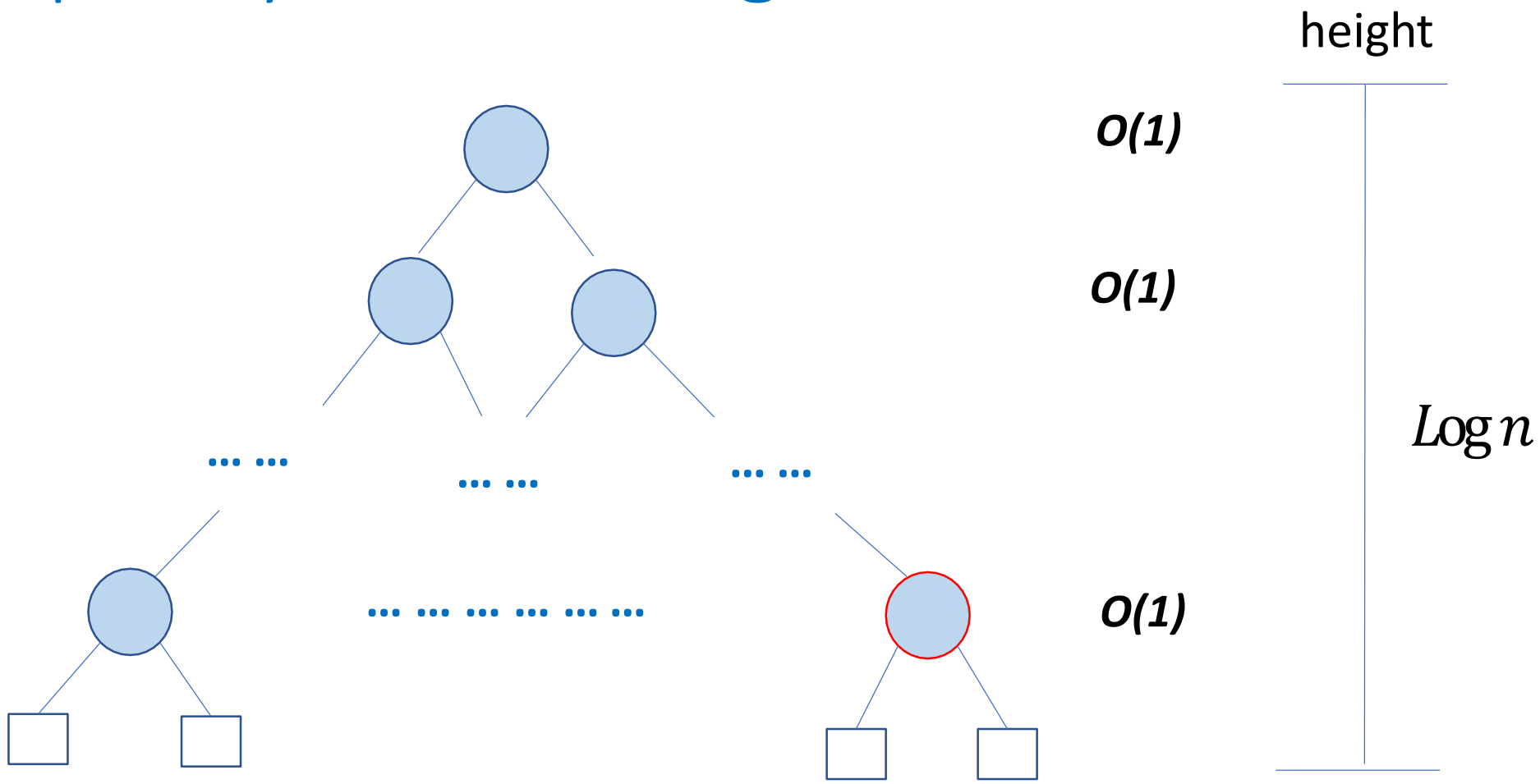
O(1)

O(1)

O(1)

Worse-case

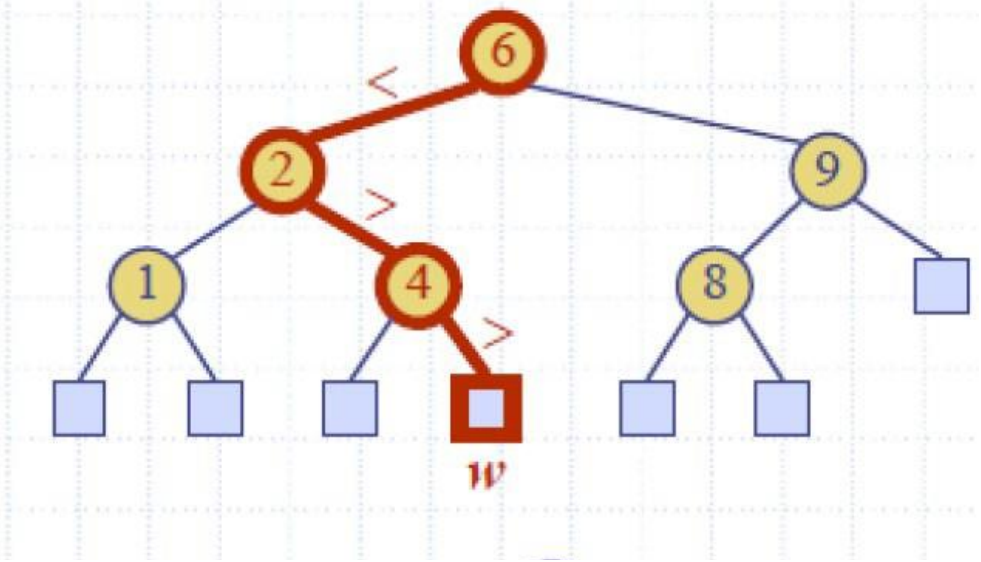
Complexity of searching in a BST



Insertion in a BST

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node

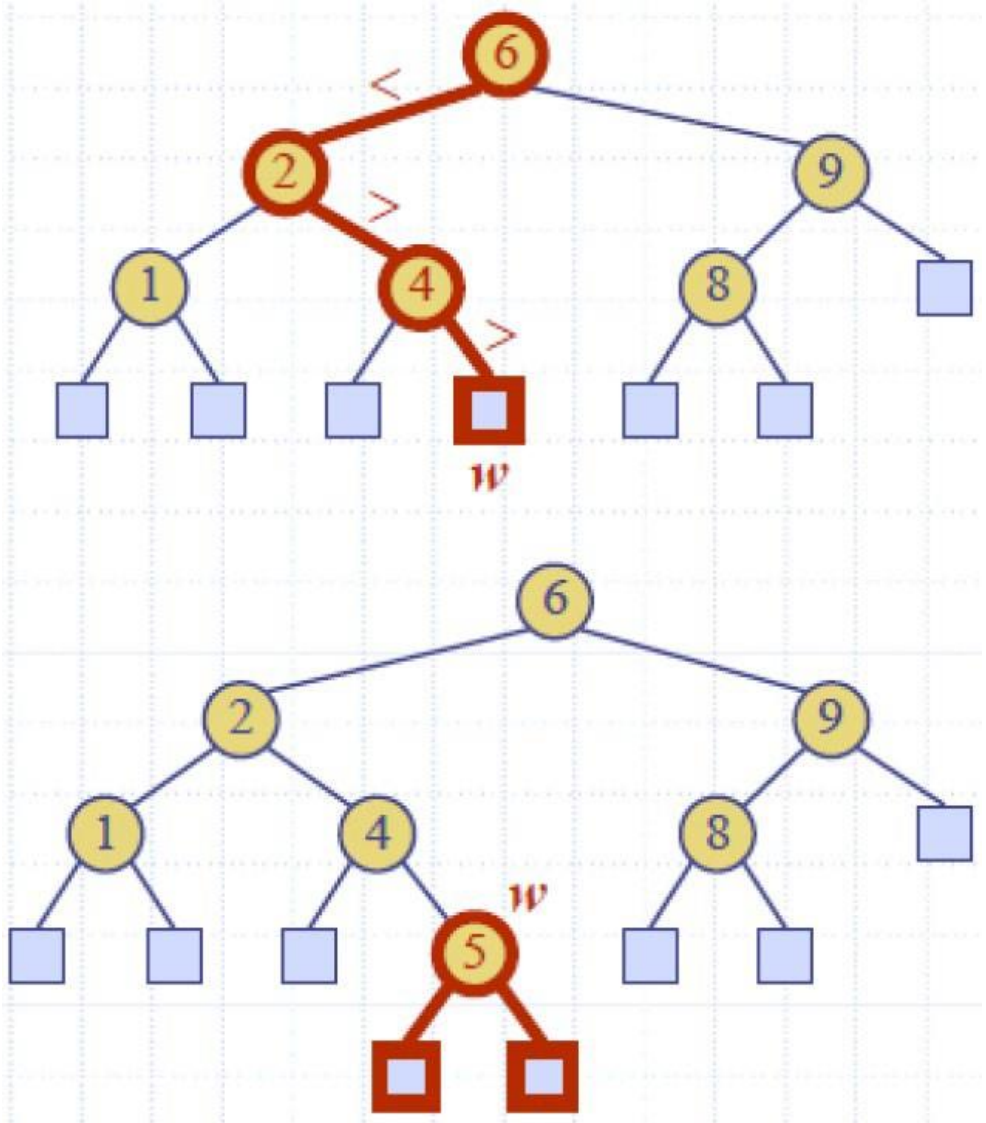
Example: insert 5



Insertion in a BST

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node

Example: insert 5

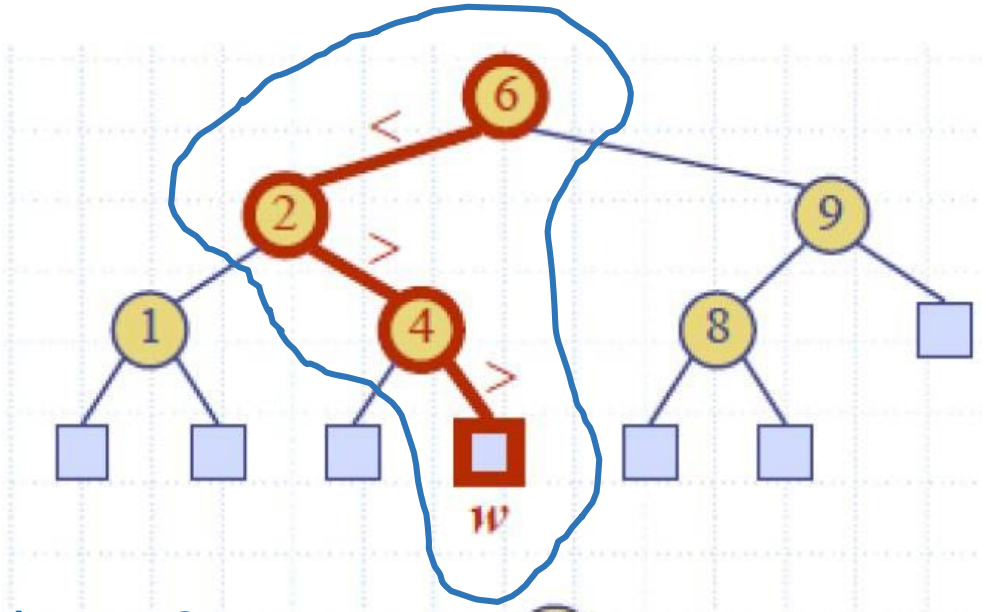


expandExternal(w)

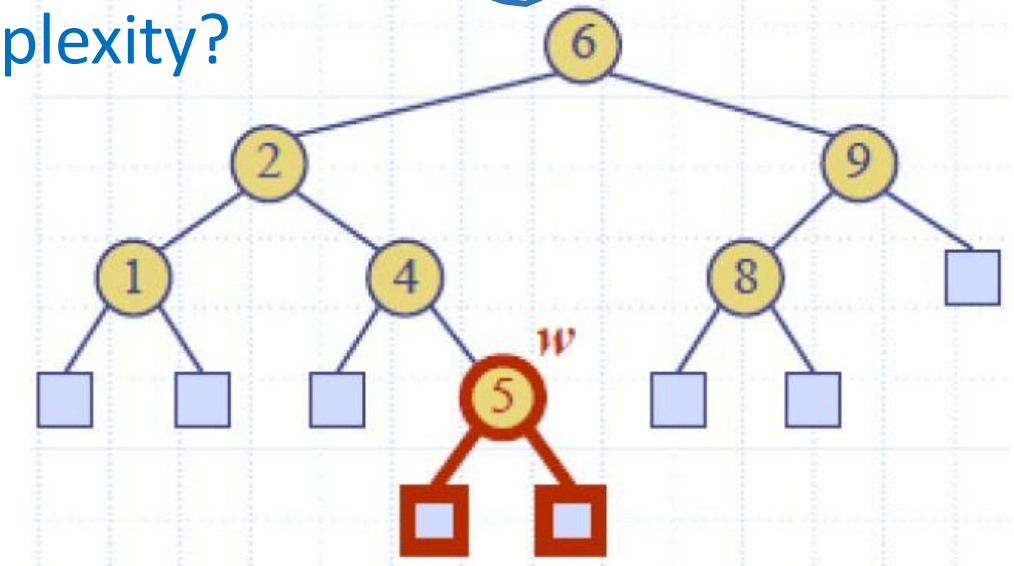
Insertion in a BST

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node

Example: insert 5



Complexity?



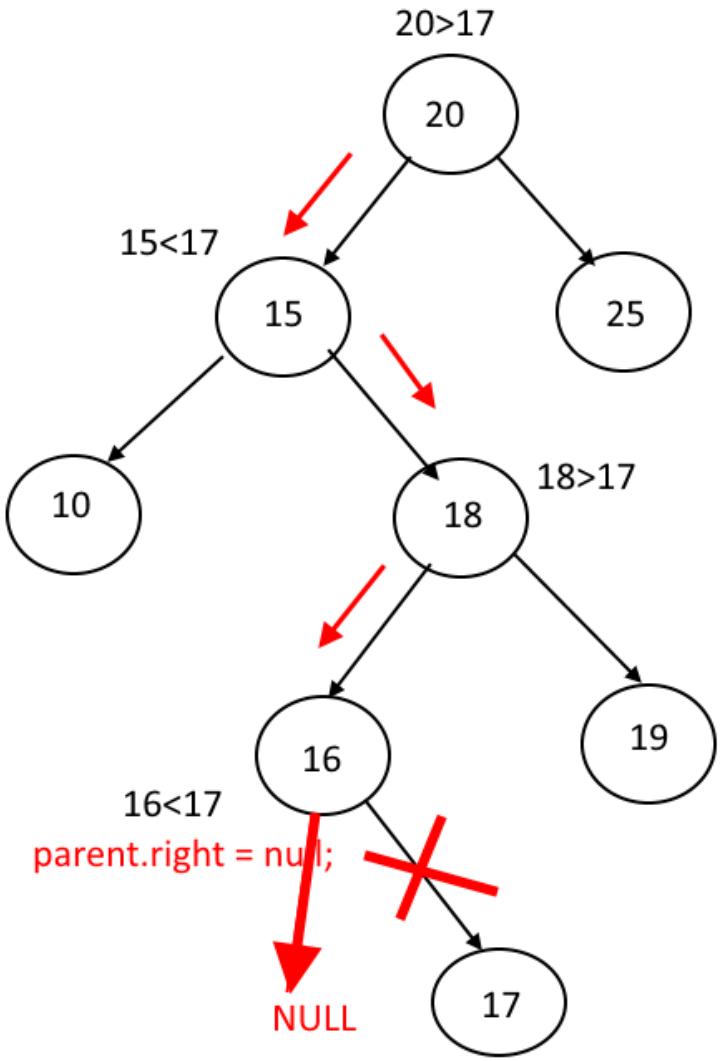
Deletion in a BST

- The following algorithm for deletion has the advantage of minimizing restructuring and unbalancing of the tree. The method returns the tree that results from the removal.

1. Find the node to be removed. We'll call it *remNode*.
2. If it's a leaf, remove it immediately by returning null.
3. If it has only one child, remove it by returning the other child node.
4. Otherwise, remove the *inorder successor* of *remNode*, copy its key into *remNode*, and return *remNode*.

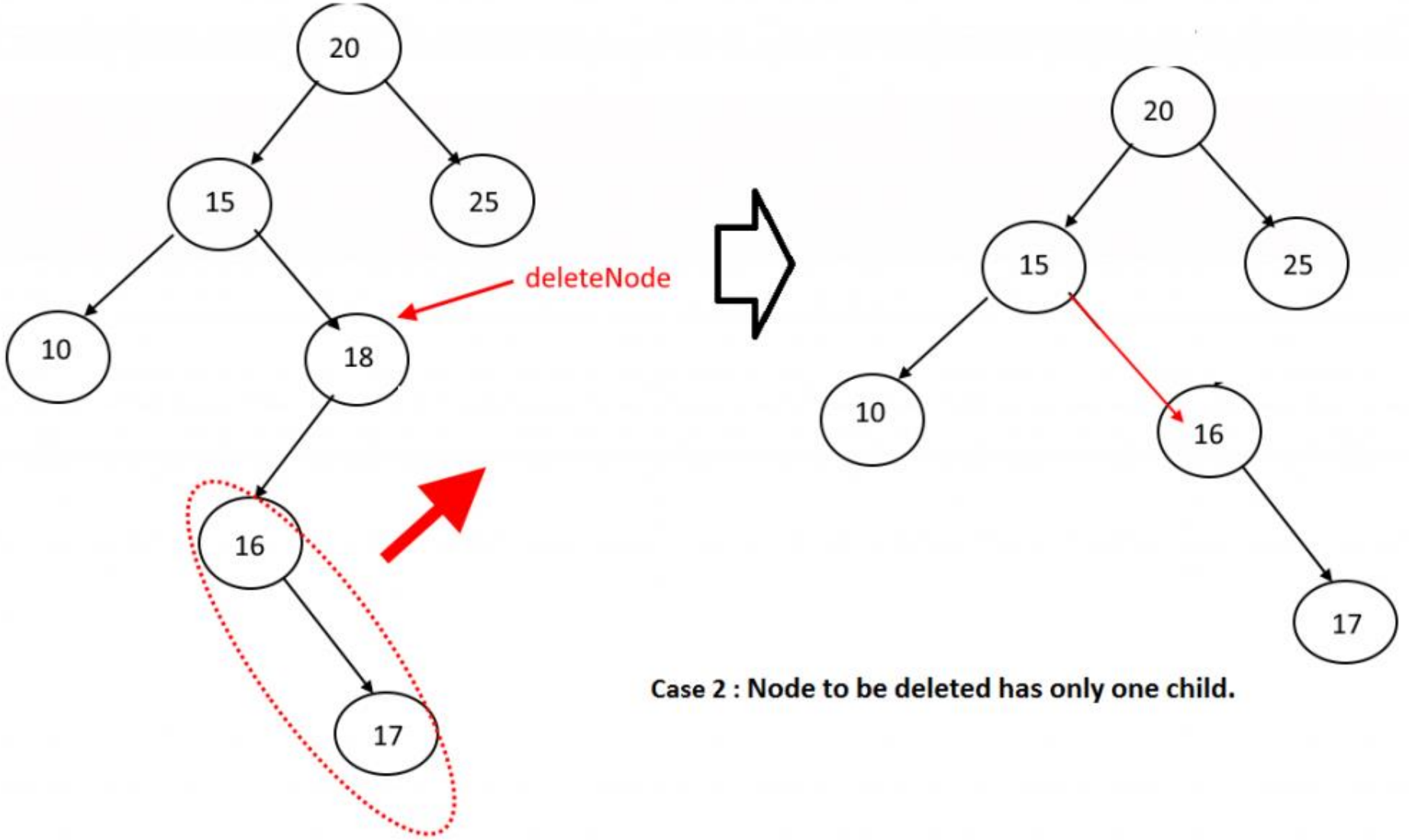
   (inorder successor: the node that would appear after the remNode if you were to do an inorder traversal of the tree)

Deletion in a BST



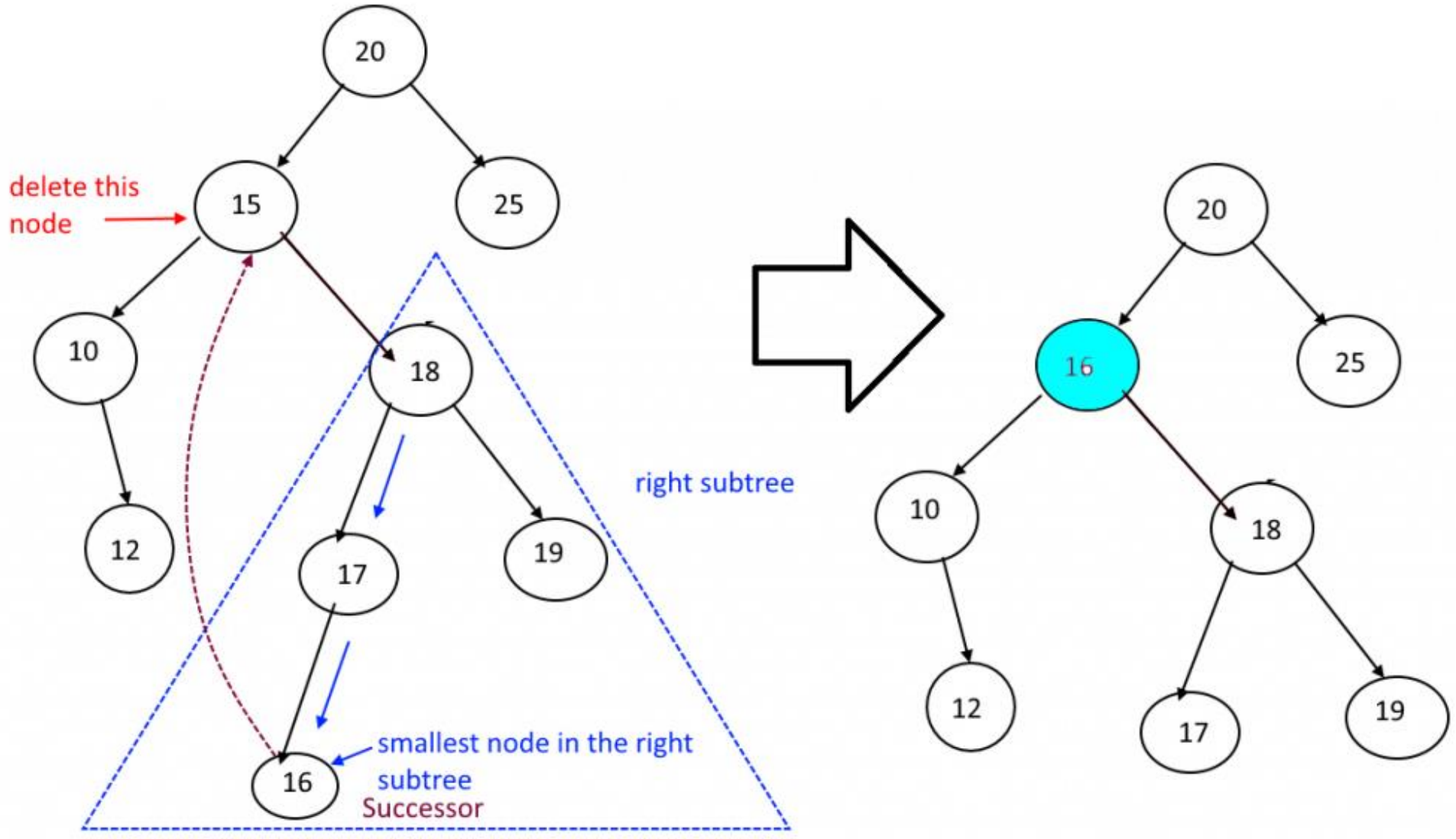
Case 1 : Node to be deleted is a leaf node (No Children).

Deletion in a BST



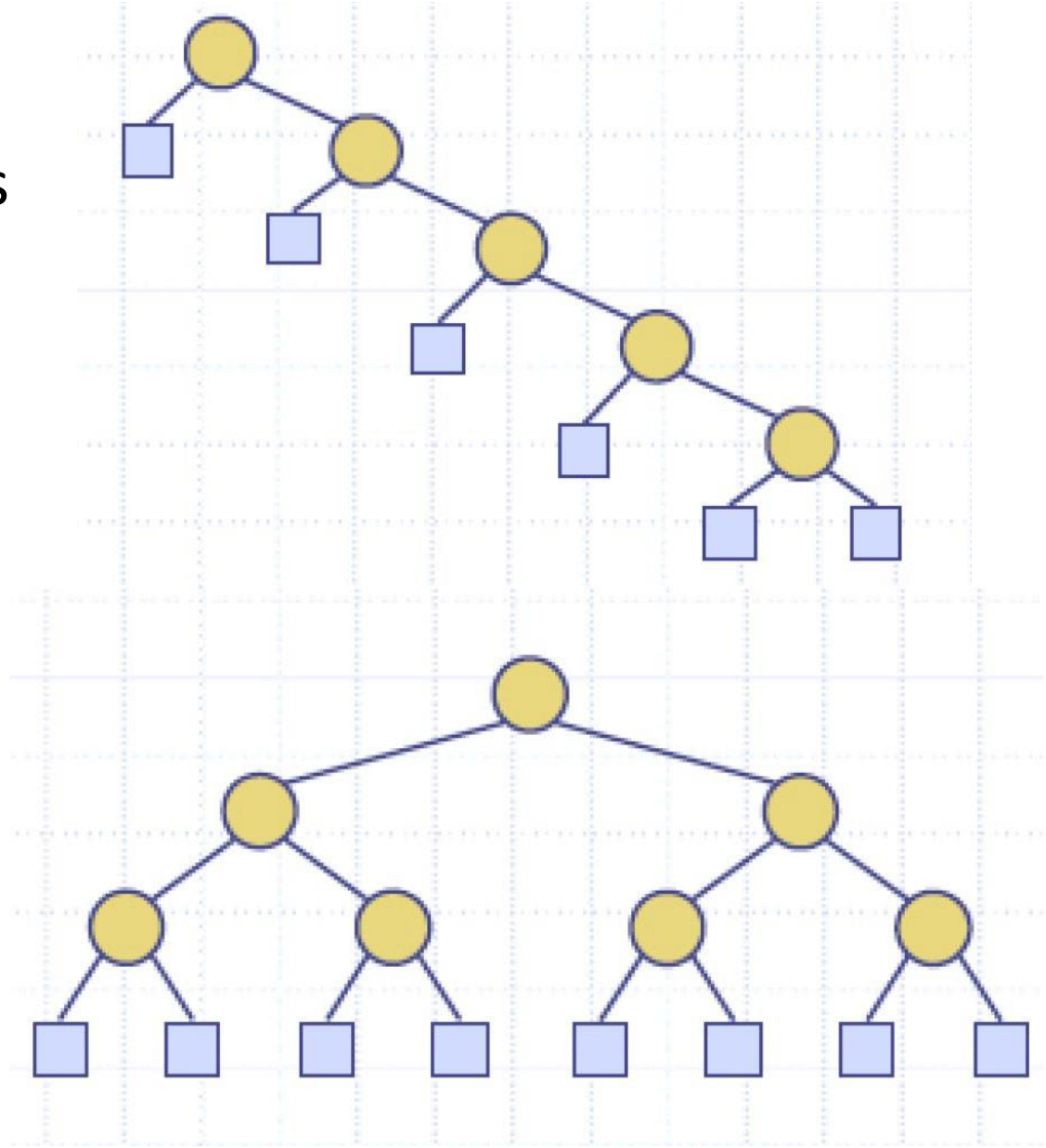
Case 2 : Node to be deleted has only one child.

Deletion in a BST



BST Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is O(n)
  - method findElement take O(h) time
- The height h is O(n) in the worst case and O(log n) in the best case



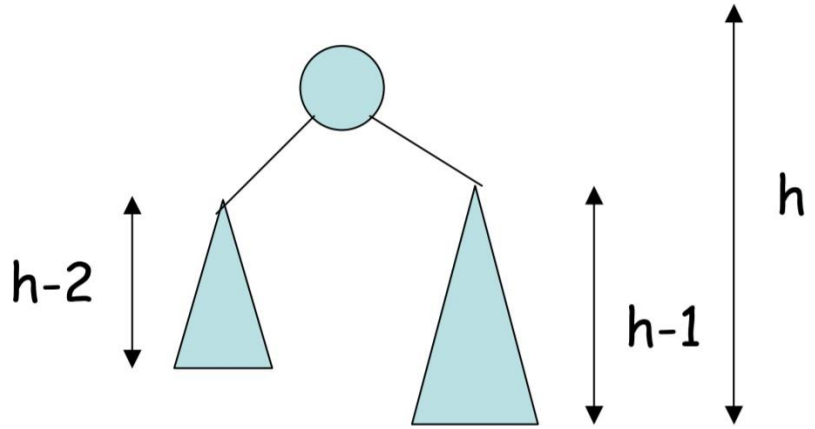
Inefficiency of general BSTs

▷ Unfortunately, *h* may be as large as *n*, e.g. for a *degenerate tree, where each parent node has only one associated child node (similar as linked list)*.

▷ The main advantage of binary searching (i.e. the $O(\log n)$ search time) may be lost if the BST is not *balanced*. In the worst case, the search time may be $O(n)$.

AVL trees (by Adel'son-Vel'skii and Landis)

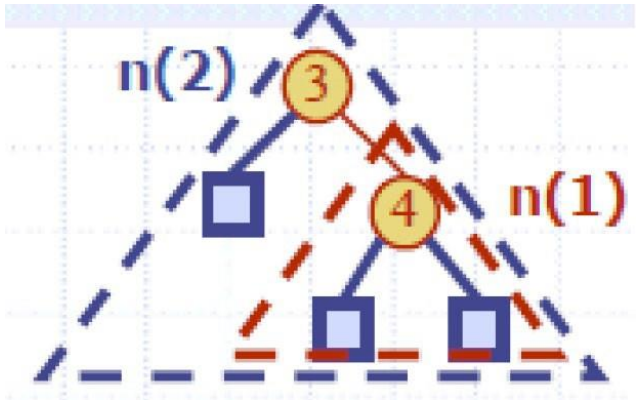
▷ *Height-Balance Property*: for any node n, the heights of n's left and right subtrees can differ by at most 1.



Height of an AVL Tree

- **Theorem**: The height of an AVL tree storing n keys is O(log n).

- **Theorem**: The height of an AVL tree storing n keys is O(log n).
- Proof: n(h): the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2.



- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
  n(h) > 2n(h-2), n(h-2) > 2n(h-4), n(h-4) > 2n(n-6) => n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
  Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

h-2i = 1 or 2, depending on if h is even or odd: i = $\lceil h/2 \rceil - 1$.

log n(h) > log $2^{h/2-1}$, log n(h)>h/2-1

AVL trees (by Adel'son-Vel'skii and Landis)

An *AVL tree* is a tree that has the Height-Balance Property.

- **Theorem**: The height of an *AVL tree* storing n keys is $O(\log n)$.

Consequence 1: A *search* in an AVL tree can be performed in time $O(\log n)$.

Consequence 2: Insertions and removals in AVL need more careful implementations (using *rotations* to maintain the height-balance property).

Insertion & deletion in AVL trees

An insertion in an AVL tree begins as an insertion in a general BST, i.e., attaching a new external node (leaf) to the tree.

- This action may result in a tree that *violates* the *height-balance property* because the heights of some nodes increase by 1.

A deletion in an AVL tree begins as a removal in a general BST.

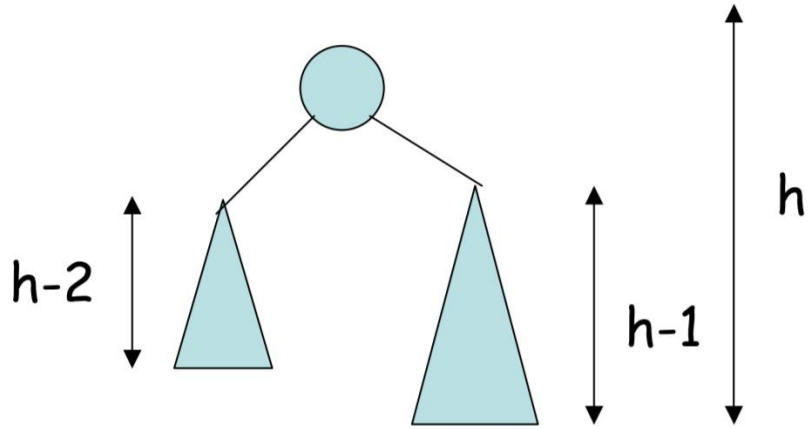
- Action may violate the height-balance property.

INT202
Complexity of Algorithms
Search Algorithms

XJTLU/SAT/INT
SEM2 AY2021-2022

AVL trees (by Adel'son-Vel'skii and Landis)

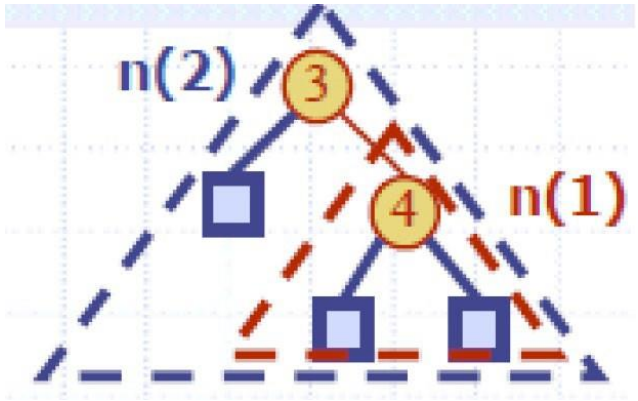
▷ *Height-Balance Property*: for any node n, the heights of n's left and right subtrees can differ by at most 1.



Height of an AVL Tree

- **Theorem**: The height of an AVL tree storing n keys is O(log n).

- **Theorem**: The height of an AVL tree storing n keys is O(log n).
- Proof: n(h): the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2.



- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
  n(h) > 2n(h-2), n(h-2) > 2n(h-4), n(h-4) > 2n(n-6) => n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$
  Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

h-2i = 1 or 2, depending on if h is even or odd: i = $\lceil h/2 \rceil - 1$.

log n(h) > log $2^{h/2-1}$, log n(h)>h/2-1

AVL trees (by Adel'son-Vel'skii and Landis)

An *AVL tree* is a tree that has the Height-Balance Property.

- **Theorem**: The height of an *AVL tree* storing n keys is $O(\log n)$.

Consequence 1: A *search* in an AVL tree can be performed in time $O(\log n)$.

Consequence 2: Insertions and removals in AVL need more careful implementations (using *rotations* to maintain the height-balance property).

Insertion in AVL trees

An *insertion* in an *AVL tree* begins as an insertion in a *general BST*, i.e., attaching a new *external* node (leaf) to the tree.

- This action may result in a tree that *violates* the *height-balance property* because the heights of some nodes increase by 1.

Insertion in AVL trees

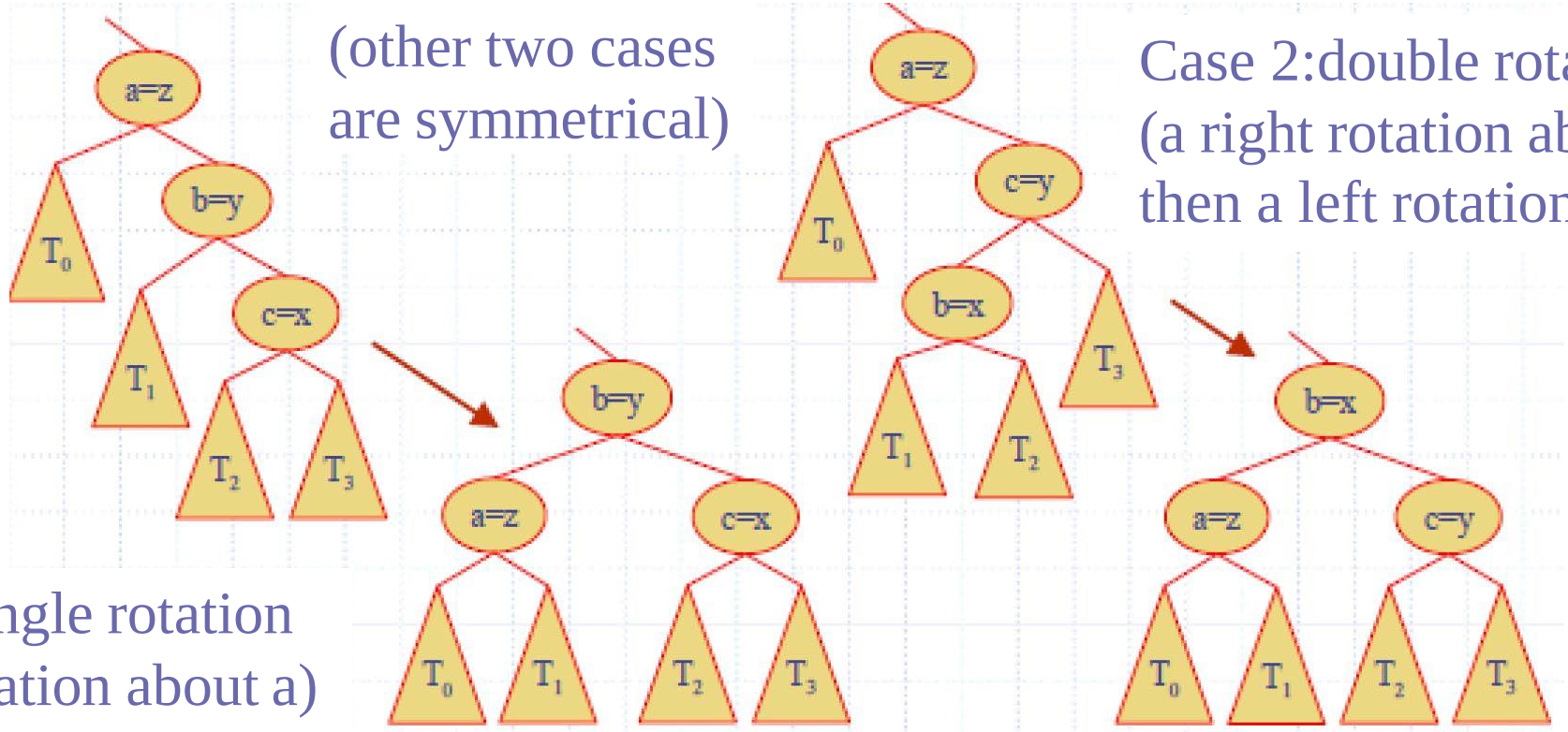
If an insertion causes T to become unbalanced, we travel up the tree from the newly created node until we find the first node x such that its grandparent z is unbalanced node.

• Since z became unbalanced by an insertion in the subtree rooted at its child y, to rebalance the subtree rooted at z, we must perform a restructuring

- we rename x, y, and z to a, b and c based on the order of the nodes in an in-order traversal.
- z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other subtrees formerly children of x, y, and z.

Restructuring

- let (a,b,c) be an inorder listing of x, y, z
- z is replaced by b, whose children are now a and c whose children, in turn, consist of the four other subtrees formerly children of x, y, and z.

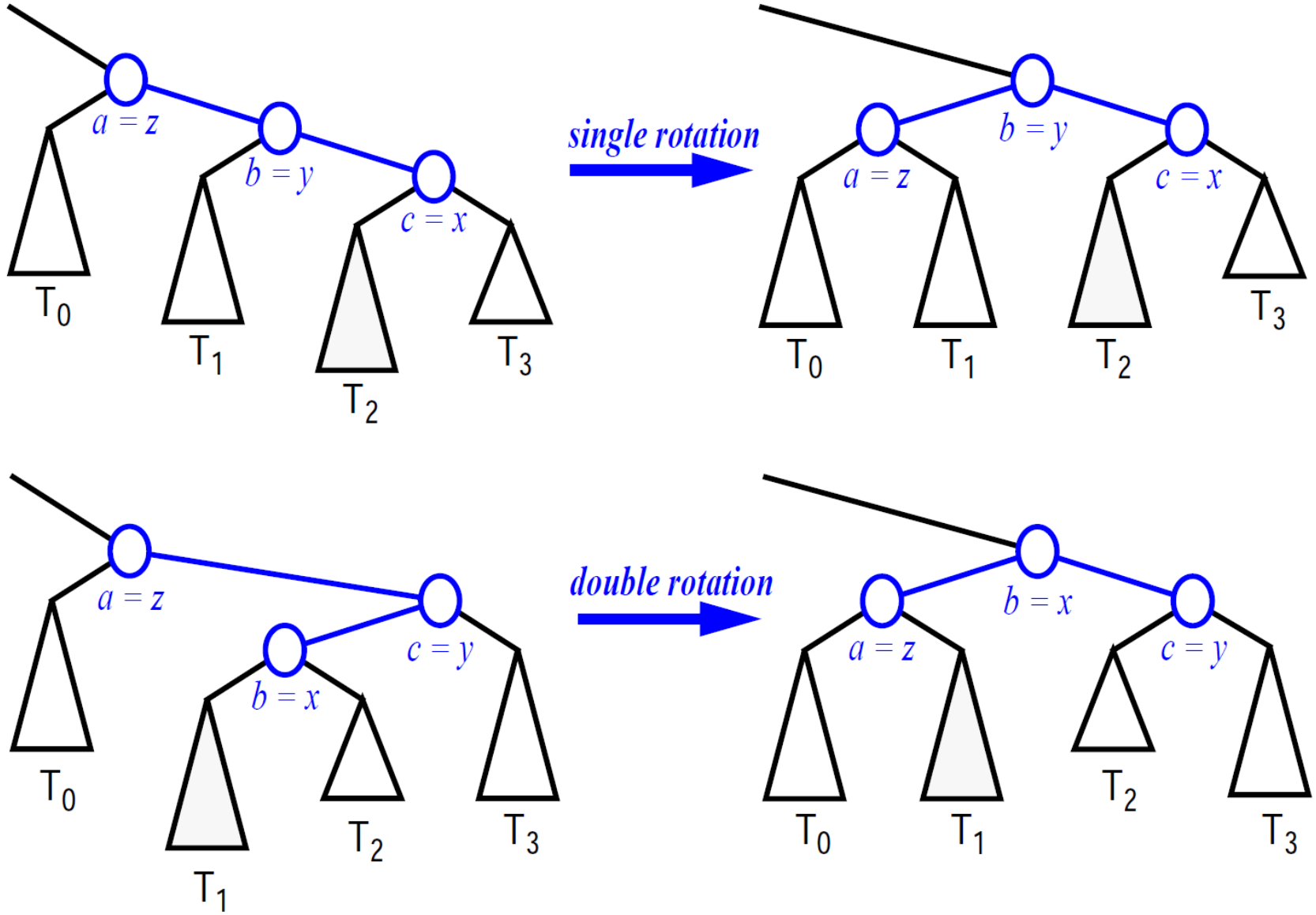


Restructuring

- Algorithm *restructure*(x):
- Input: A node x of a binary search tree T that has both a parent y and a grandparent z
- Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving nodes x, y, and z
- 1: Let (a,b,c) be a left-to-right (inorder) listing of the nodes x, y, and z, and let (T0, T1, T2, T3) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.
- 2: Replace the subtree rooted at z with a new subtree rooted at b.
- 3: Let a be the left child of b and let T0 and T1 be the left and right subtrees of a, respectively.
- 4: Let c be the right child of b and let T2 and T3 be the left and right subtrees of c, respectively.

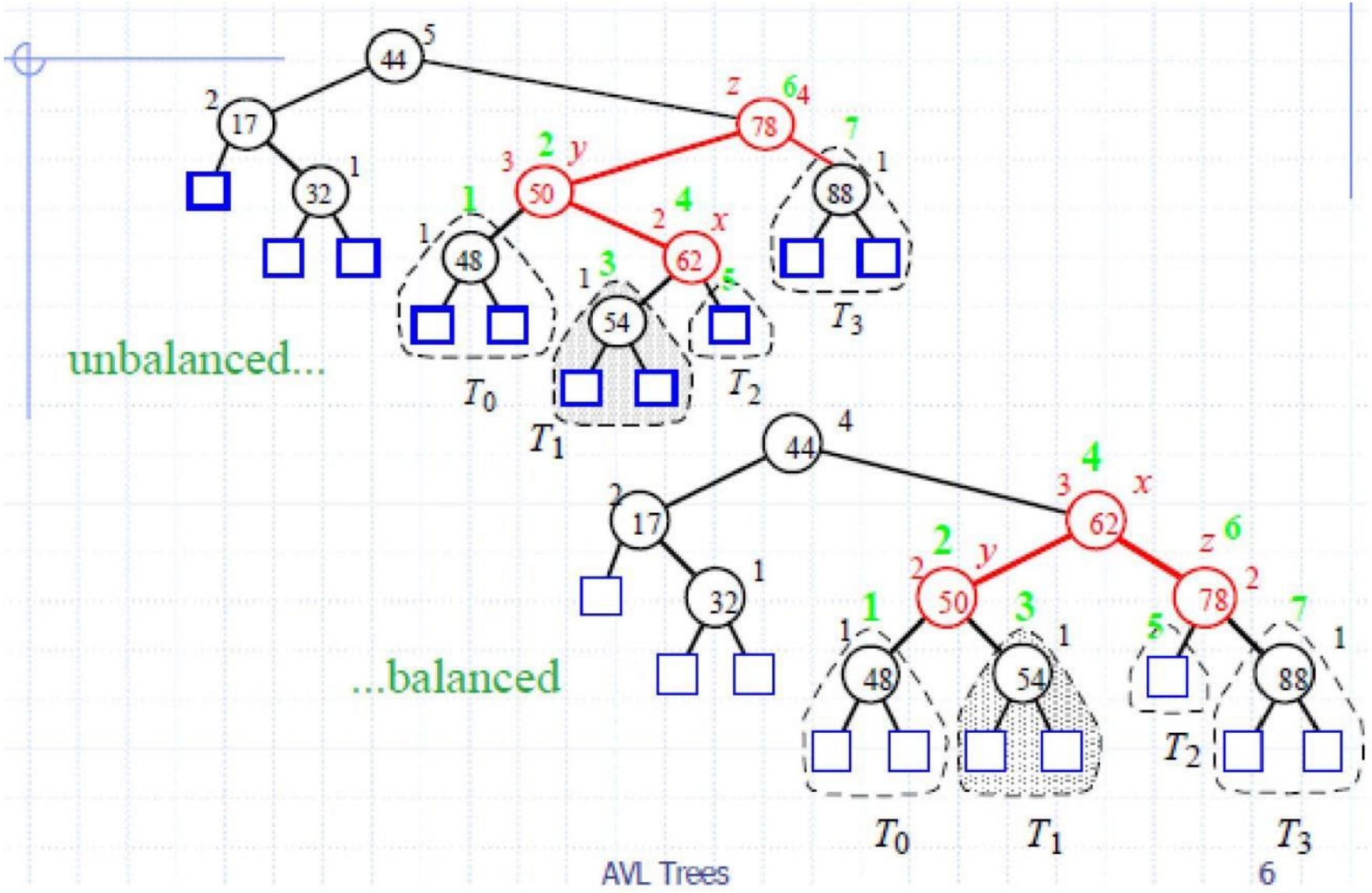
Insertion Example

- 2: Replace the subtree rooted at z with a new subtree rooted at b.
- 3: Let a be the left child of b and let T0 and T1 be the left and right subtrees of a, respectively.
- 4: Let c be the right child of b and let T2 and T3 be the left and right subtrees of c, respectively.



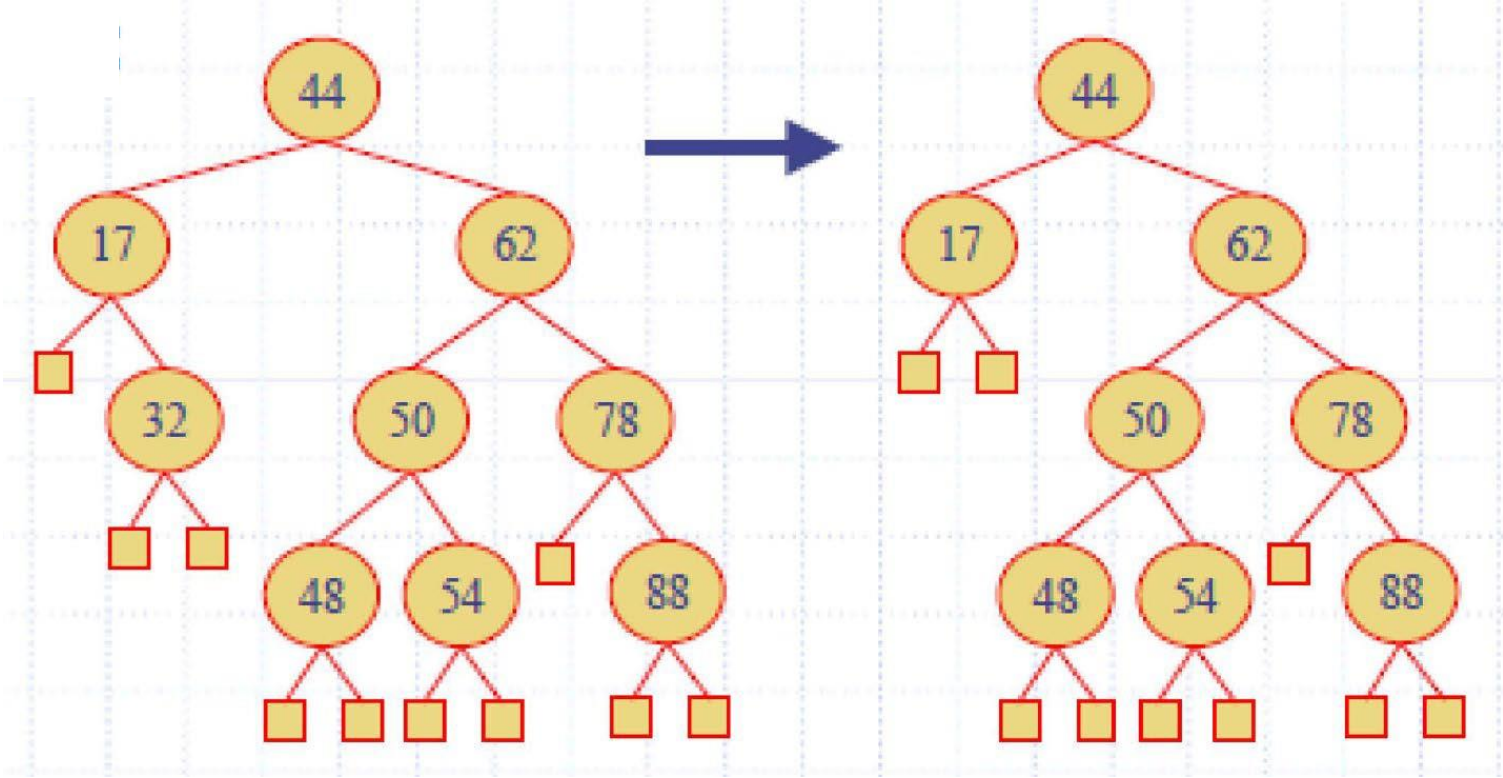
Insertion Example

- 2: Replace the subtree rooted at z with a new subtree rooted at b.
- 3: Let a be the left child of b and let T0 and T1 be the left and right subtrees of a, respectively.
- 4: Let c be the right child of b and let T2 and T3 be the left and right subtrees of c, respectively.



Removal in an AVL Trees

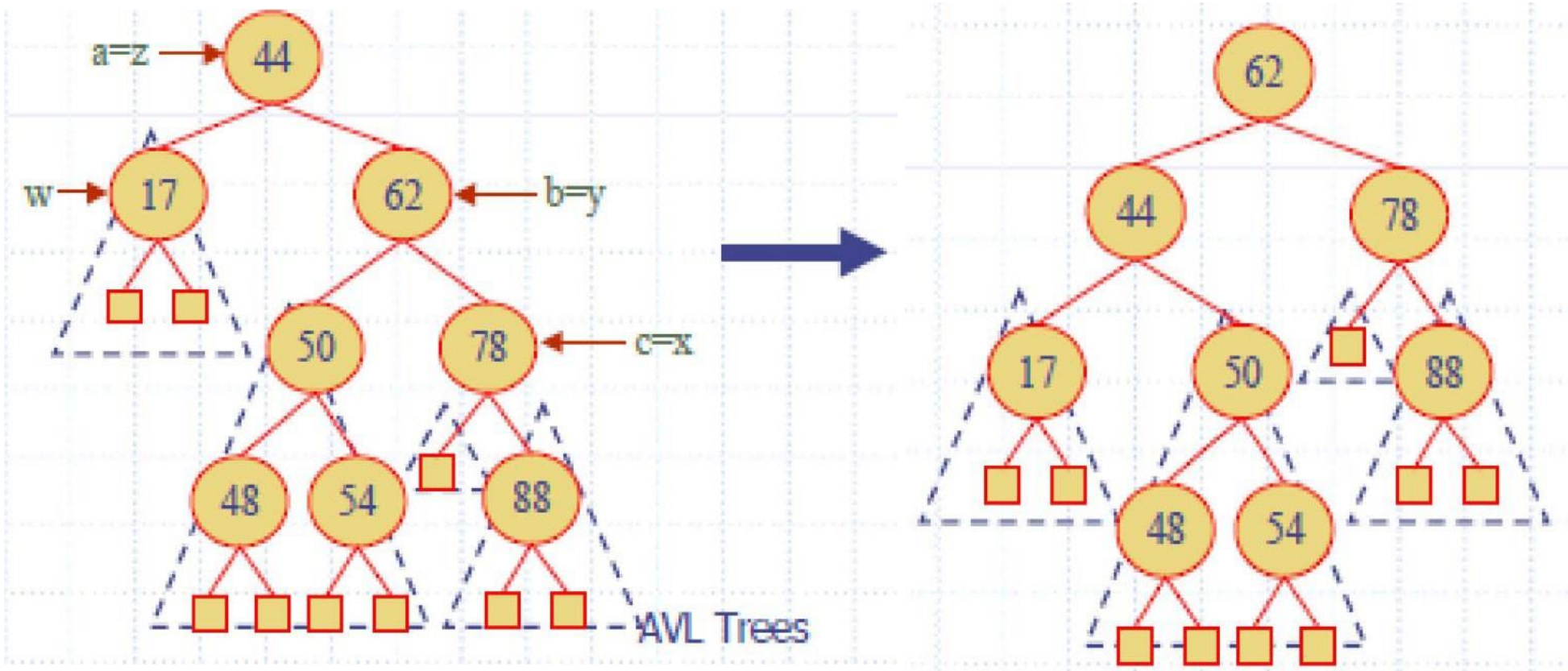
- A removal in an AVL tree begins as a removal in a general BST.
- Action may violate the height-balance property.



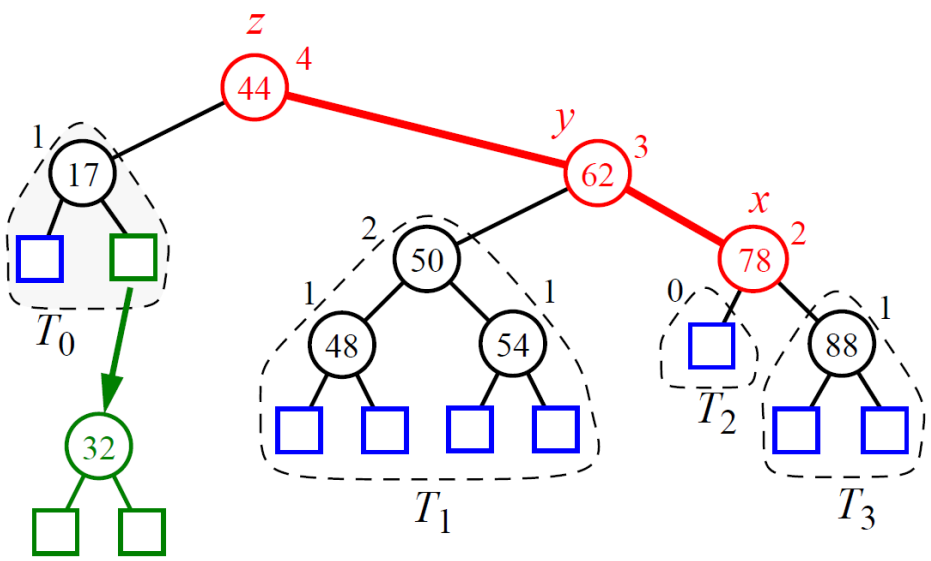
Before deletion of 32 After deletion

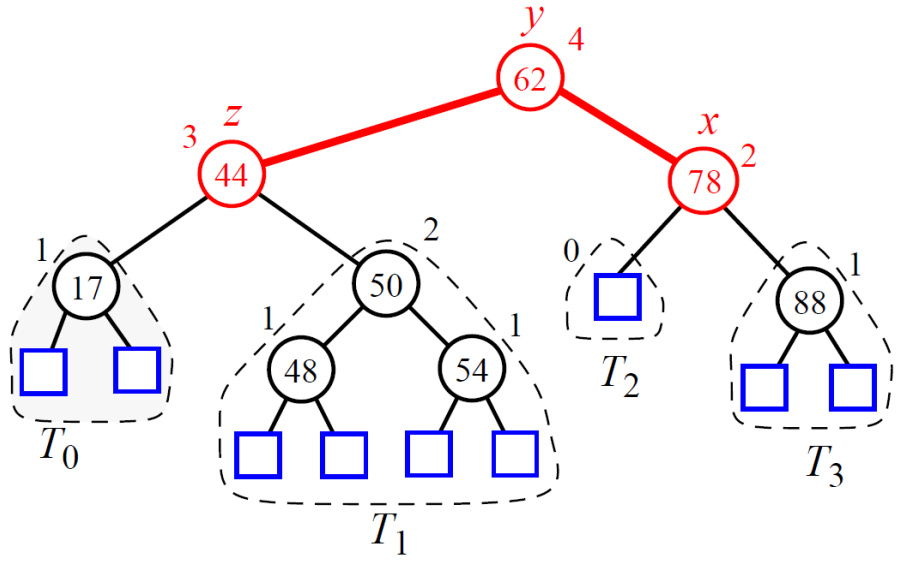
Removal in an AVL Trees

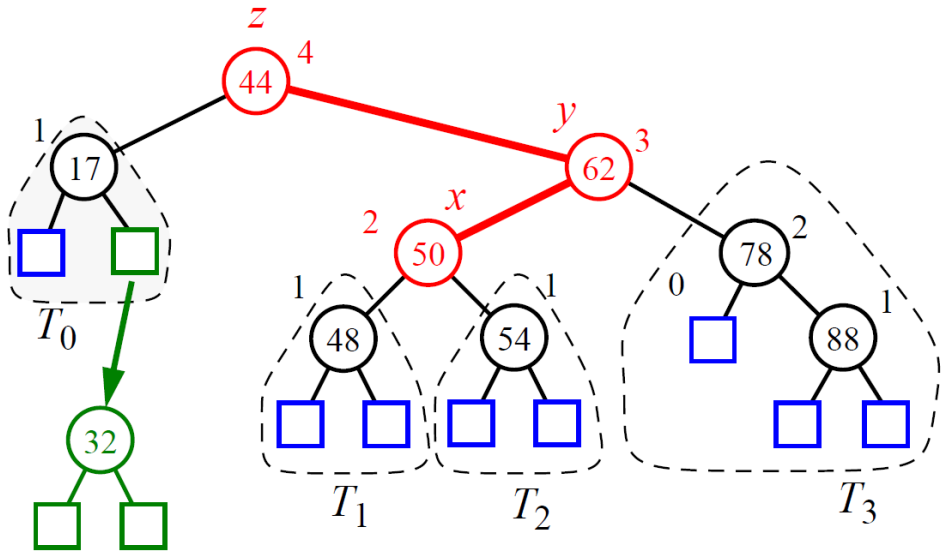
- We perform restructure(x) to restore balance.

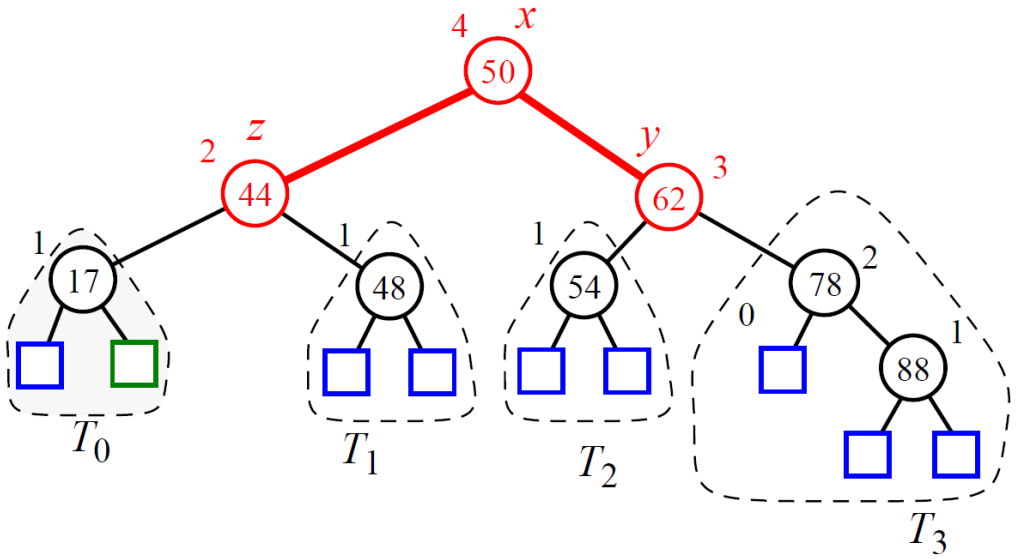


Removal in an AVL Trees



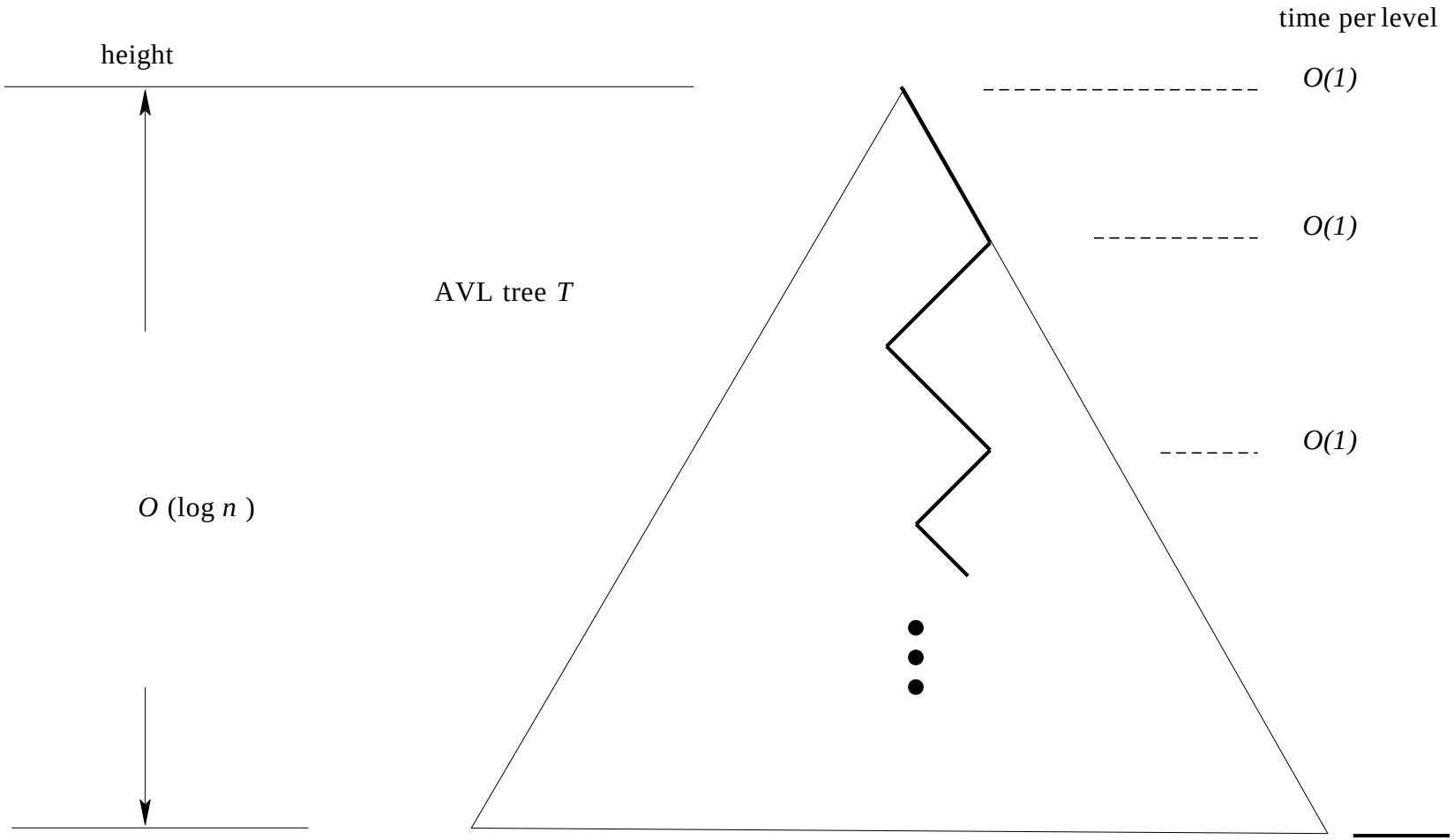






AVL performance

- All operations (*search, insertion*, and *removal*) on an AVL tree with n elements can be performed in $O(\log n)$ time.



AVL performance

- A single restructure is O(1)

- Search is O(log n)
  - height of tree is O(log n), no restructures needed

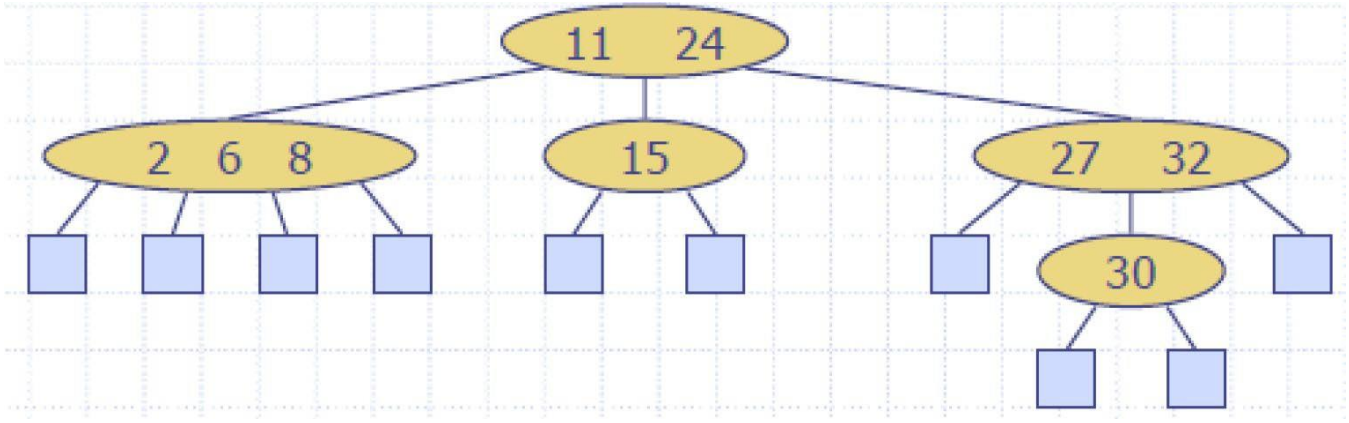
- Insertion is O(log n)
  - Initial search is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

- Removal is O(log n)
  - Initial search is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

Multi-Way Search Tree

❖ A multi-way search tree is an ordered tree such that

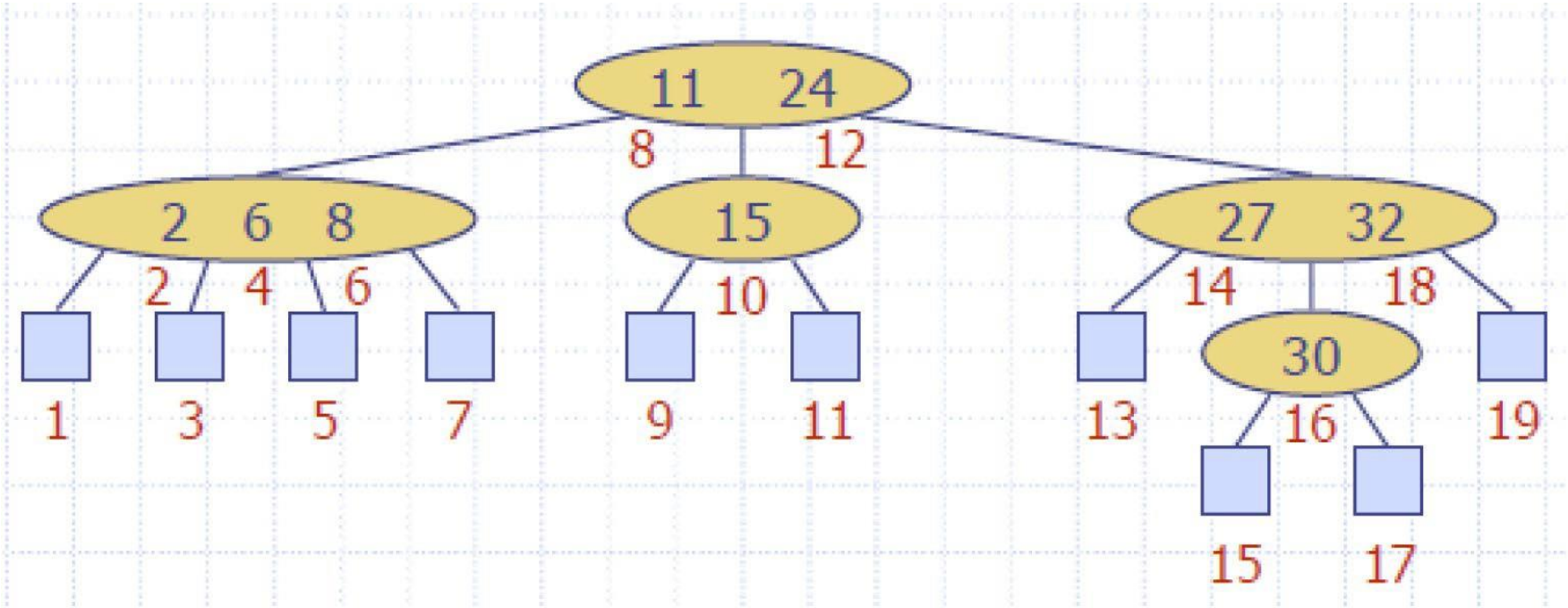
- Each internal node has at least two children and stores d-1 key-element items (k_i, o_i) where d is the number of children
  - For a node with children $v_1, v_2, \ldots v_d$,storing keys $k_1, k_2, \ldots k_{d-1}$
    - keys in the subtree of v_1 are less than k_1
    - keys in the subtree of v_i are between $k_{i-1}$and k_i (i = 2, . .. , d - 1)
    - keys in the subtree of v_d are greater than k_{d-1}
  - The leaves store no items and serve as placeholders



Multi-Way Inorder Traversal

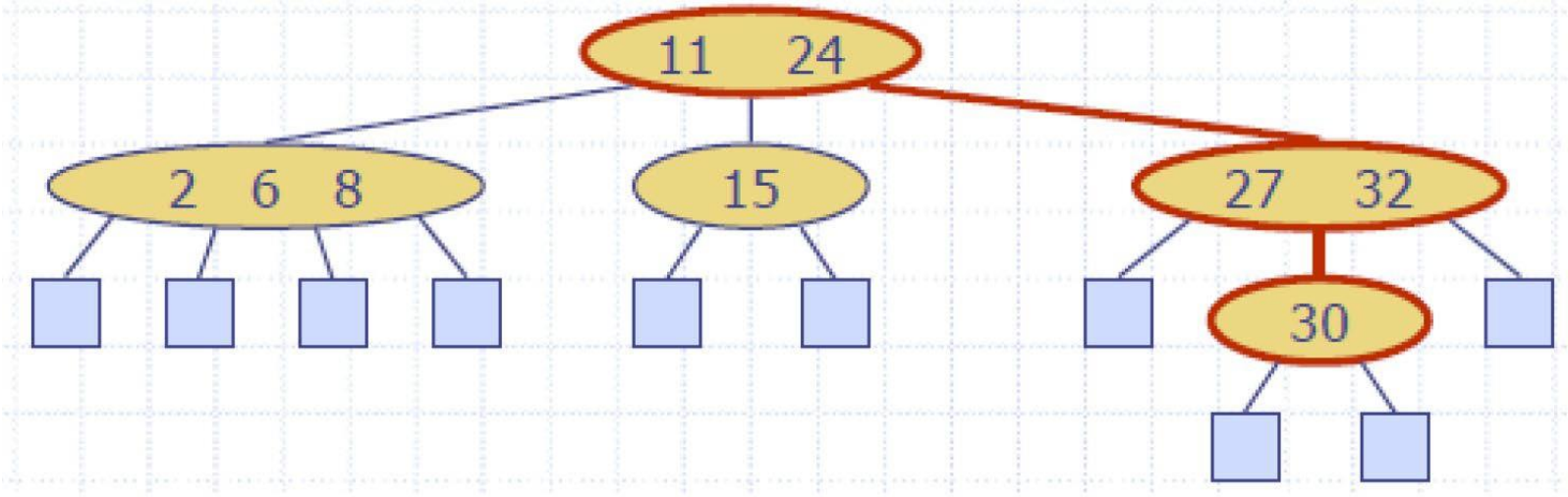
◆We can extend the notion of inorder traversal from binary trees to multi-way search trees

◆Namely, we visit item (k_i, o_i) of node v between the recursive traversals of the subtrees of v_i rooted at children v_i and v_{i+1}



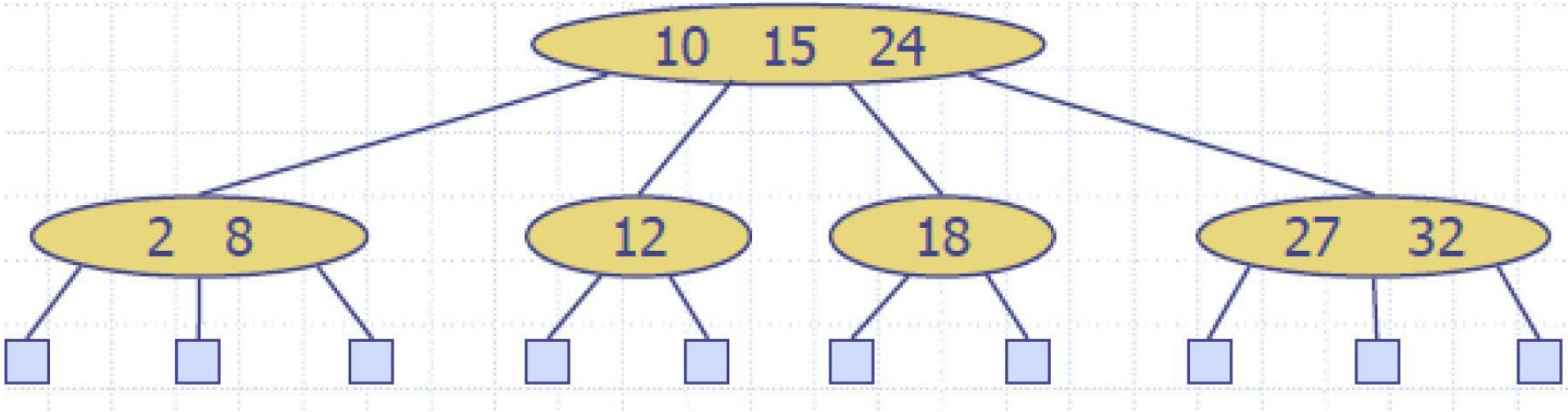
Multi-Way Searching

- Similar to search in a binary search tree
- A each internal node with children $v_1, v_2, \ldots v_d$ and keys $k_1, k_2, \ldots k_{d-1}$
  - $k = k_i$ (i = 1, …, d-1) : the search terminates successfully
  - $k < k_1$: we continue the search in child v_1
  - $k_{i-1} < k < k_i$ (i = 2, …, d-1): we continue the search in child v_i
  - $k > k_{d-1}$: we continue the search in child v_d
- Example: search for 30



(2, 4) trees

- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node



(2, 4) trees

❖ Theorem: A (2,4) tree storing n items has height *O(log n)*

Proof:

■ Let n be the height of a (2,4) tree with n items

■ Since there are at least 2^i items at depth i = 0, …, h-1 and no items at depth, h, we have

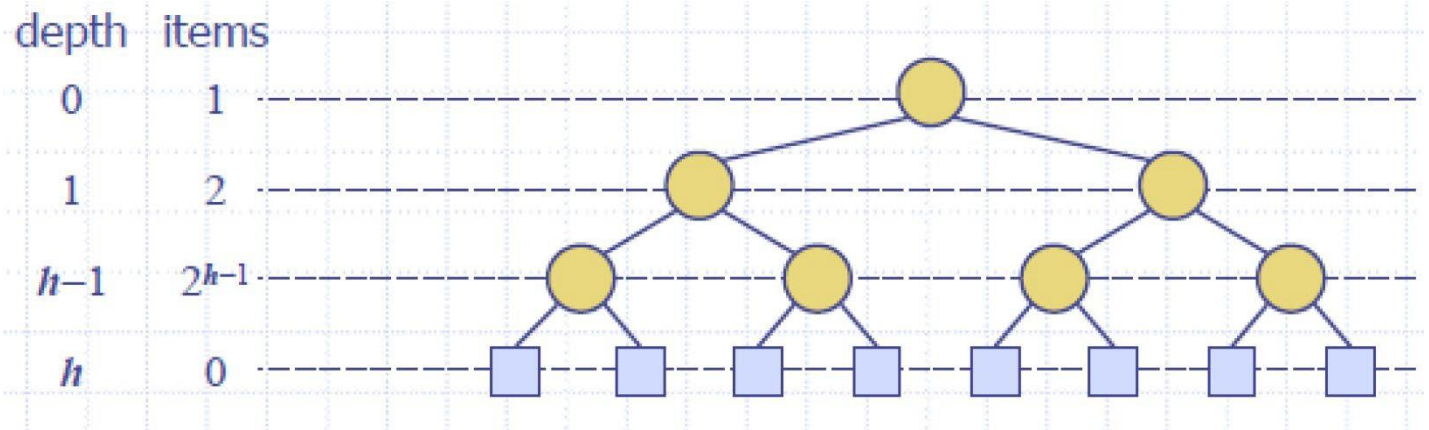
$$n \geq 1 + 2 + 4 + \cdots + 2^{h-1} = 2^h - 1$$

■ Thus, $h \leq \log(n+1)$

Sum of a Geometric Series

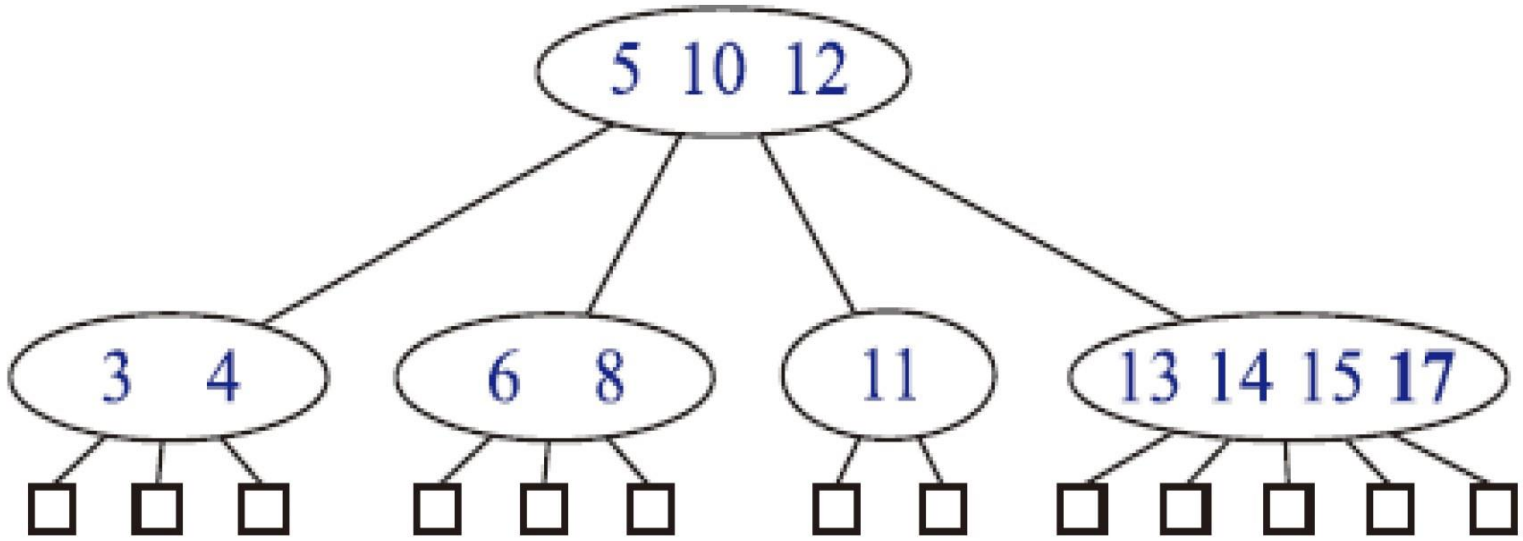
$$\sum_{k=0}^{n-1}(ar^k) = a\left(\frac{1-r^n}{1-r}\right)$$

❖Searching in a (2,4) tree with n items takes *O(log n)* time depth items



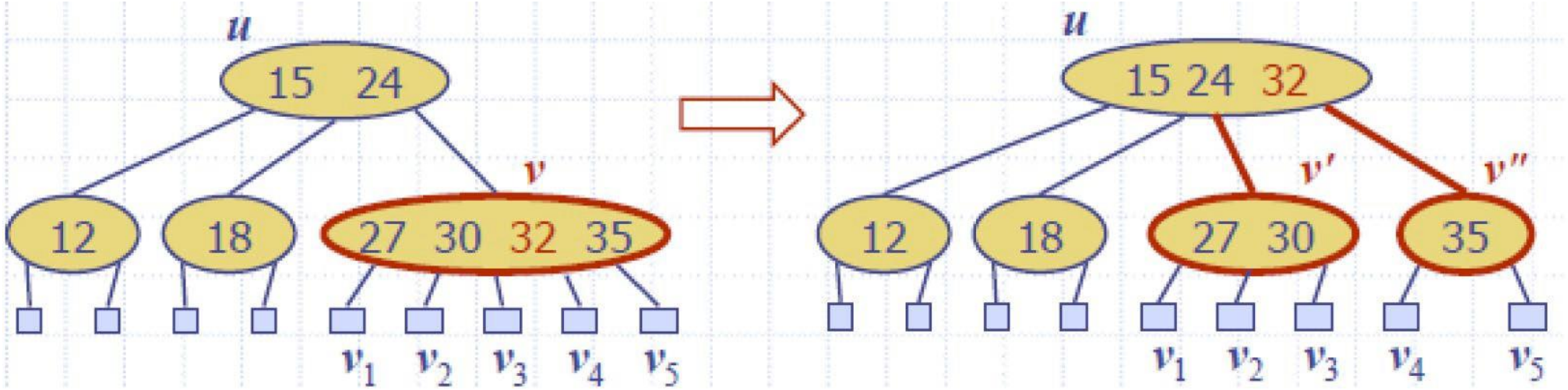
(2, 4) tree - Insertion

- We insert a new item *(k, o)* at the parent v of the leaf reached by searching for k
  - We preserve the depth property but
  - We may cause an overflow (i.e., node v may become a 5-node)
- Example: inserting key 17 causes an overflow

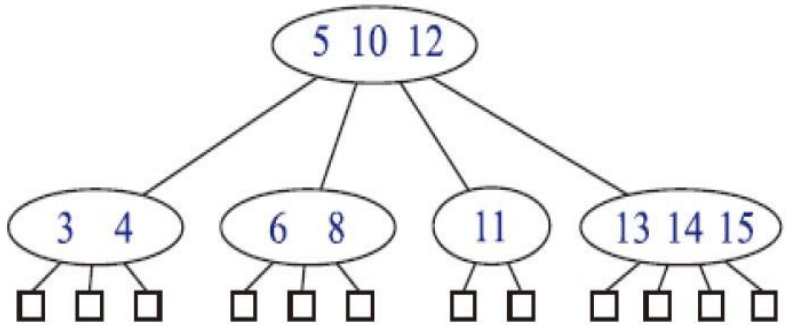


(2, 4) trees - Split operation

- We handle an overflow at a 5-node v with a split operation:
  - let $v_1, v_2, \ldots v_5$ be the children of v and $k_1 \ldots k_4$ be the keys of v
  - node v replaced nodes v' and v''
    - v' is a 3-node with keys $k_1 k_2$ and children $v_1 v_2 v_3$
    - v'' is a 2-node with key k_4 and children $v_4 v_5$
  - key k_3 is inserted into the parent u of v (a new root may be created)
- The overflow may propagate to the parent node u

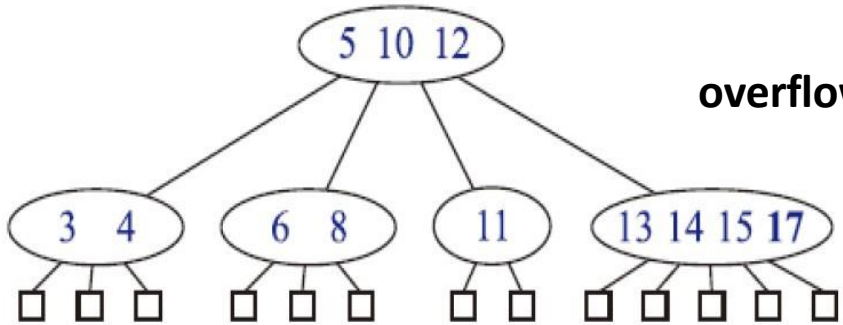


(2, 4) tree - Insertion



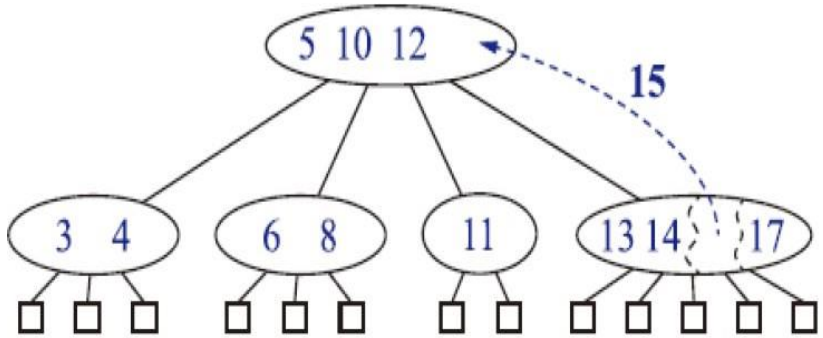
(a)

overflow



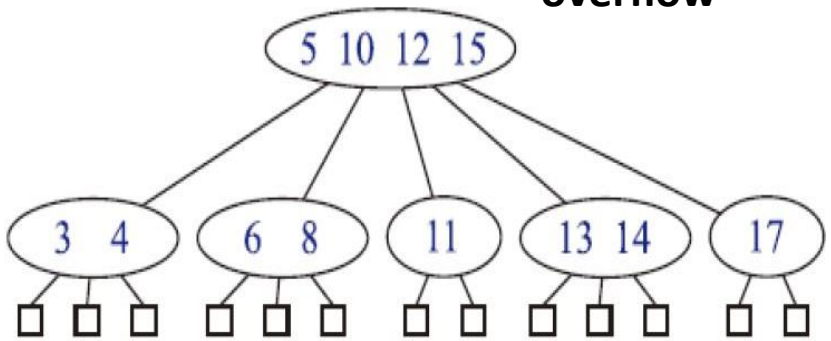
(b)

node split



(c)

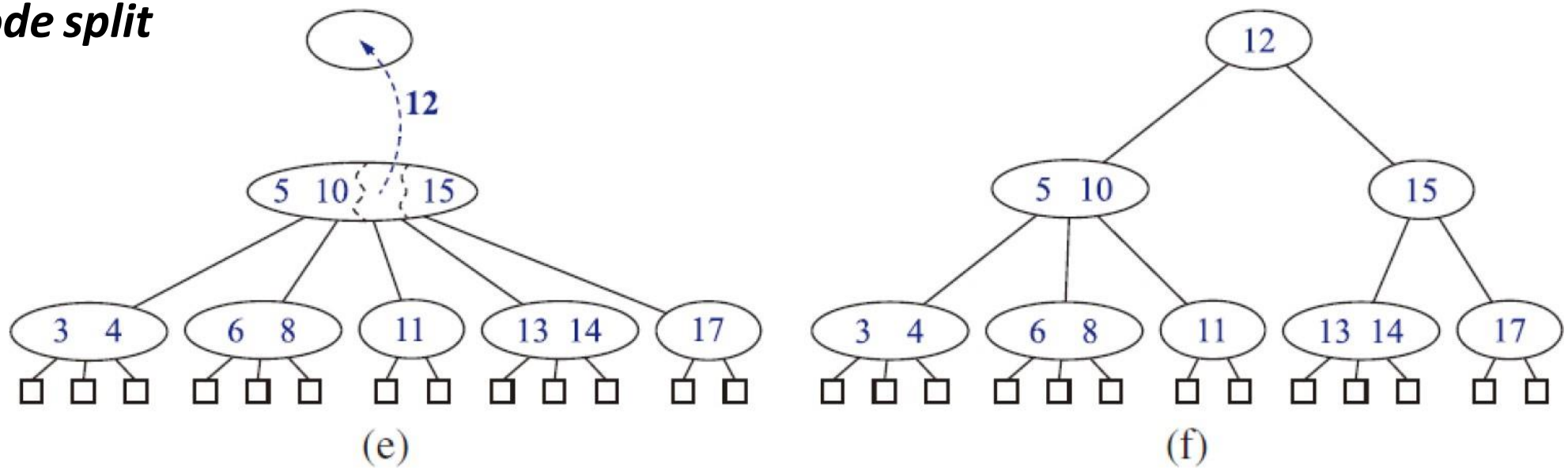
overflow



(d)

(2, 4) tree - Insertion

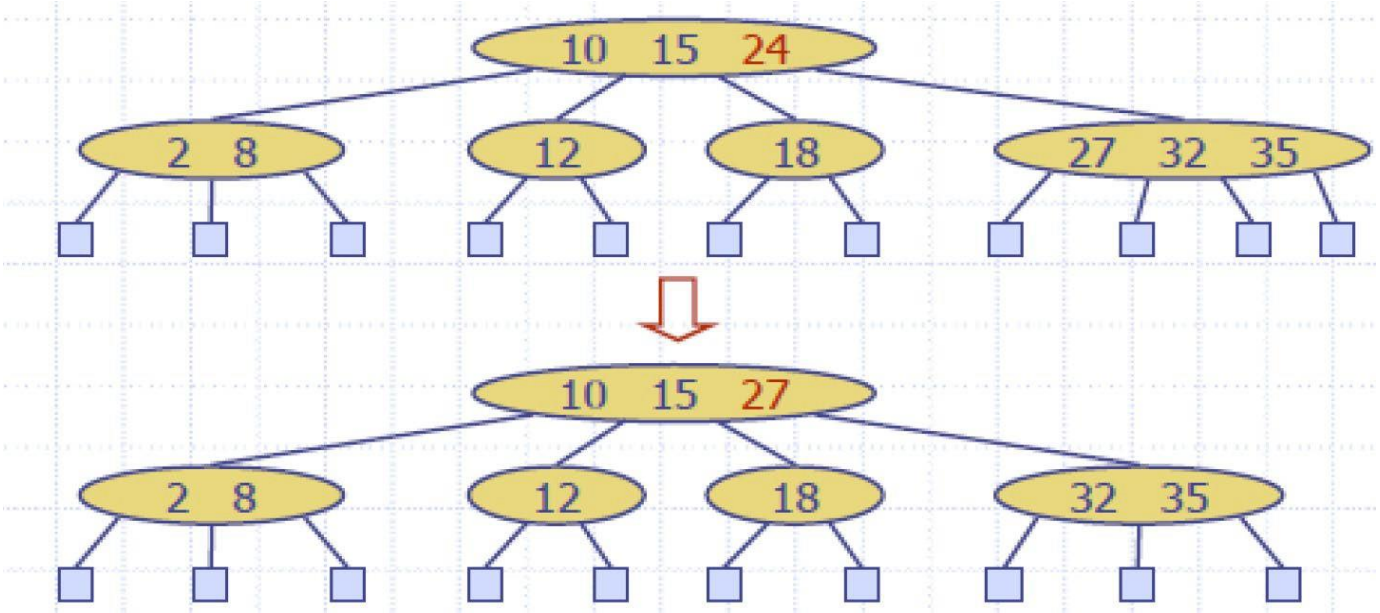
node split



A split operation either eliminates the overflow or propagates it into the parent of the current node.
Indeed, this propagation can continue all the way up to the root of the search tree.
But if it does propagate all the way to the root, it will finally be resolved at that point.

(2, 4) trees - Deletion

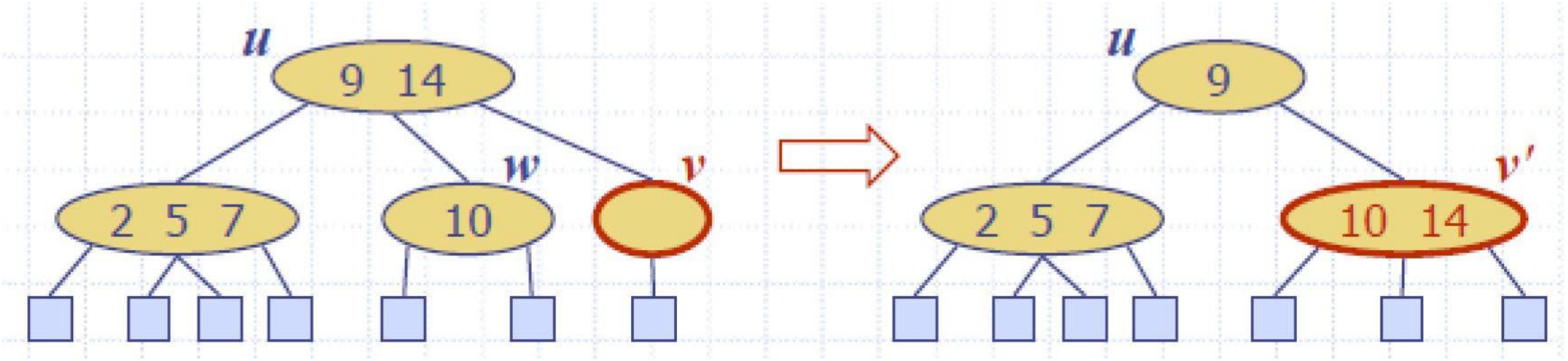
- We reduce deletion of an item to the case where the item is at the node with leaf children
- Otherwise, we replace the item with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter item
- Example: to delete key 24, we replace it with 27 (inorder successor)



(2, 4) trees - Deletion

Underflow and Fusion

- Deleting an item from a node v, may cause an **underflow**, where node v becomes a 1-node with one child and no keys
- To handle an underflow at node v with parent u, we consider two cases
- Case 1: the adjacent siblings of v are 2-nodes
  - Fusion operation: we merge v with an adjacent sibling w and move an item from u to the merged node v'
  - After a fusion, the underflow may propagate to the parent u

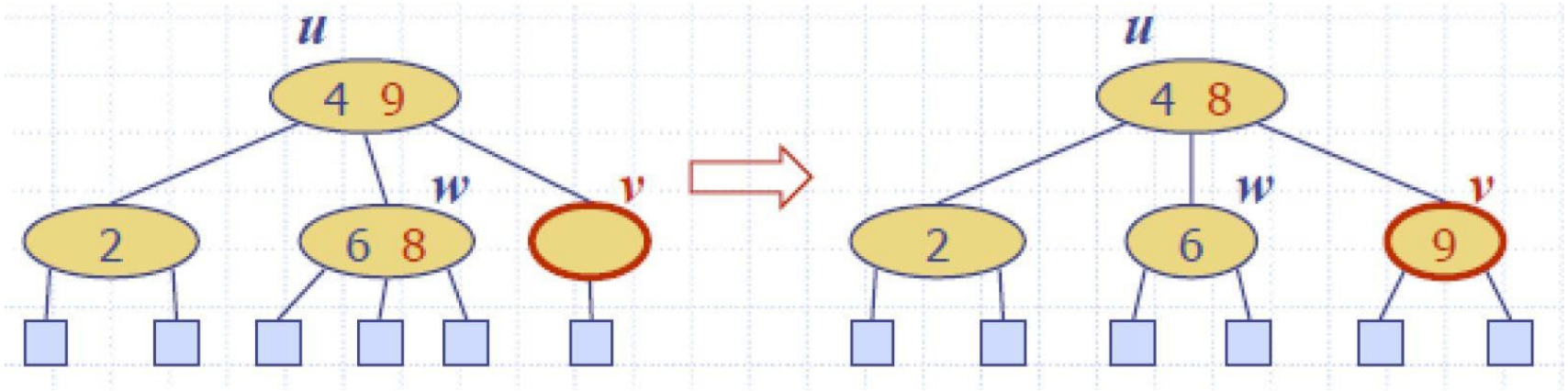


(2, 4) trees - Deletion

Underflow and Transfer

❖ Case 2: an adjacent sibling w of v is a 3-node or a 4-node

- Transfer operation
  1. we move a child of w to v
  2. we move an item from u to v
  3. we move an item from w to u
- After a transfer, no underflow occurs



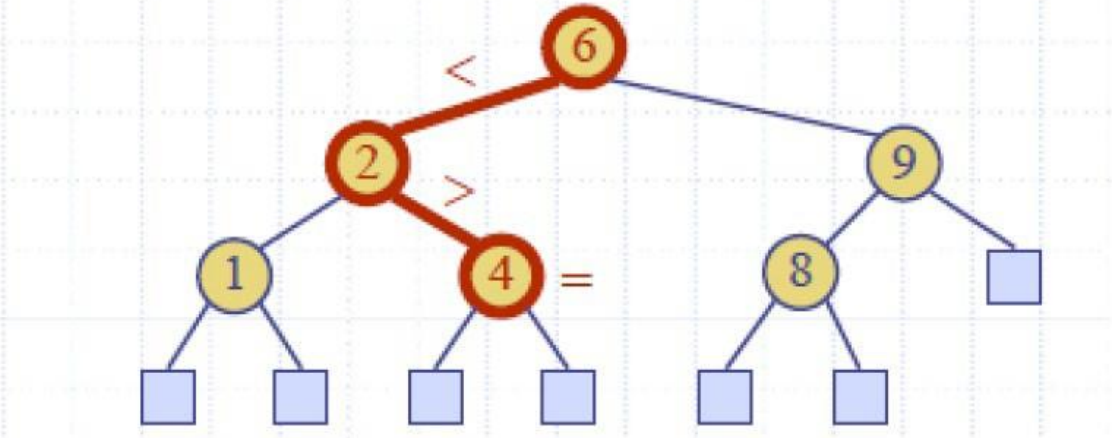
INT202
Complexity of Algorithms
Sorting Algorithms

XJTLU/SAT/INT
SEM2 AY2021-2022

Review

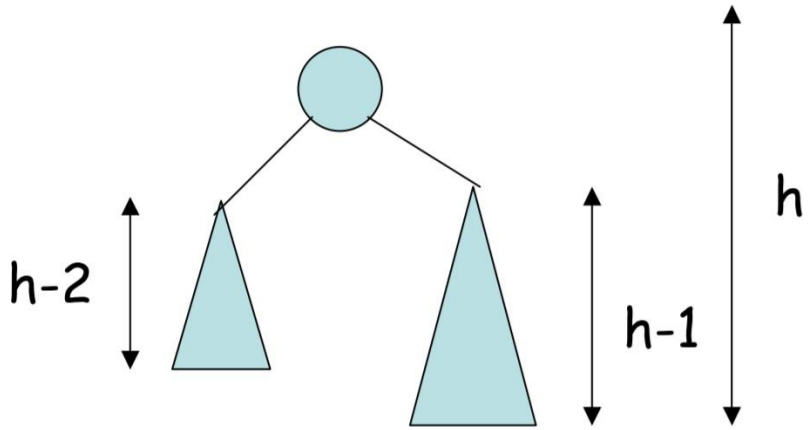
- **Binary Search Tree (BST)**
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return NO—SUCH KEY

```
Algorithm findElement(k, v)
    if T.isExternal (v)
        return NO_SUCH_KEY
    if k < key(v)
        return findElement(k, T.leftChild(v))
    else if k = key(v)
        return element(v)
    else { k > key(v) }
        return findElement(k, T.rightChild(v))
```

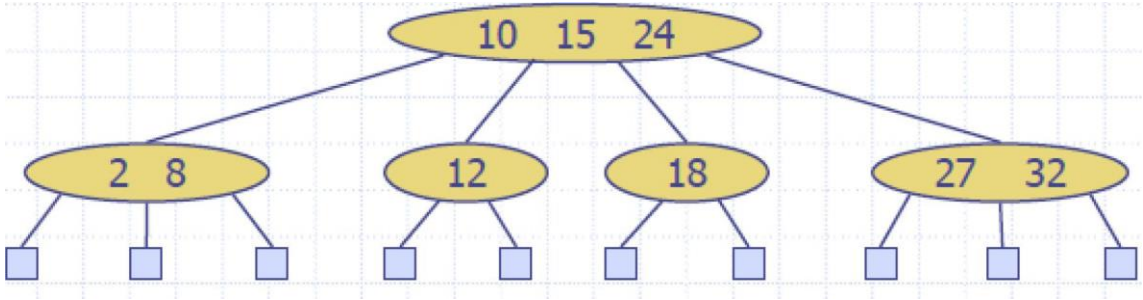


Review

- **Binary Search Tree (BST)**
- **AVL Tree**
- *Height-Balance Property*: for every *internal* node, *v*, of *T*, the heights of the children of *v* can differ by at most 1.



- **(2,4) tree**
- A multi-way search
- Node-Size Property: every internal node has at most four children
- Depth Property: all the external nodes have the same depth



Review

findElement, insertItem, removeElement

- **BST**

  All operations in a BST are performed in $O(h)$, where h is the height of the tree.

- **AVL Tree , (2,4) Tree**

  All these operations are performed in O(log n)

Sorting

Sorting problem: Given a collection, *C*, of *n* elements (and a total ordering) arrange the elements of *C* into *non-decreasing* order, e.g.

45	3	67	1	5	16	105	8

1	3	5	8	16	45	67	105

Sorting

Sorting is a fundamental algorithmic problem in computer science.

We will investigate various methods that we can use to sort items.

Many algorithms perform sorting (as a subroutine) during their execution. Hence, efficient sorting methods are crucial to achieving good algorithmic performance.

We may not always require a fully sorted list, so some methods might be more appropriate depending upon the exact task at hand.

Sorting algorithms might be directly adaptable to perform additional tasks and directly provide solutions in this fashion.

Priority Queues

A **Priority Queue** is a container of elements, each having an associated *key*.

Keys determine the *priority* used in picking elements to be removed.

A *priority Queue* (PQ) has these fundamental methods:

- *insertItem(k,e)*: insert element *e* having key *k* into PQ.
- *removeMin()*: remove minimum element.
- *minElement()*: return minimum element.
- *minKey()*: return key of minimum element.

PQ Sorting - Algorithm

How can we use a priority queue to perform sorting on a set *C*?
Do this in two phases:

- *First phase*: *Put* elements of *C* into an initially empty priority queue, *P*, by a series of *n* insertItem operations.
- *Second phase*: *Extract* the elements from *P* in *non-decreasing* order using a series of *n* removeMin operations.

Heap Data Structure

A *heap* is a realization of a Priority Queue that is *efficient* for both *insertions* and *deletions*.

A *heap* allows insertions and deletions to be performed in *logarithmic* time.

In a *heap* the *elements* and their *keys* are stored in an almost complete binary tree. Every level of the binary tree, except possibly the last one, will have the maximum number of children possible.

Complete Binary Tree

- Here are two important types of binary trees. Note that the definitions, while similar, are logically independent.

Definition: a binary tree T is *full* if each node is either a leaf or possesses exactly two child nodes.

Definition: A *complete* binary tree of height h is a binary tree which contains exactly 2^d nodes at depth d, $0 \leq d \leq h$.

Definition: A *nearly complete* binary tree of height h is a binary tree of height h in which:
a) There are 2^d nodes at depth d for d = 1,2,...,h−1,
b) The nodes at depth h are as far left as possible.

*some books use 'complete' to represent 'nearly complete', and use 'perfect' to represent 'complete'

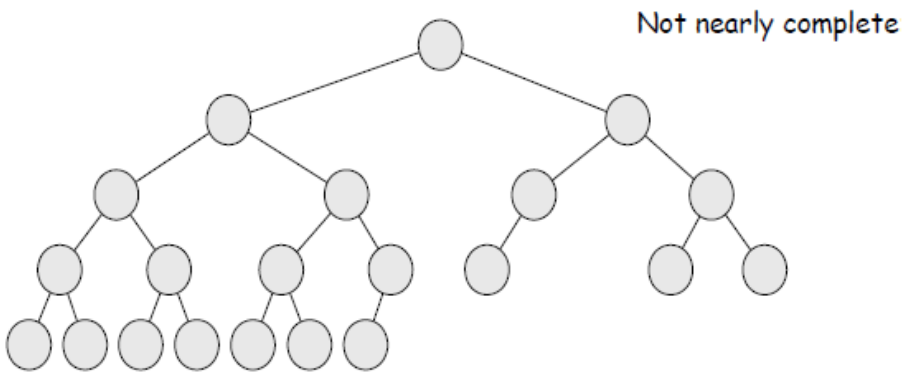
Complete Binary Tree

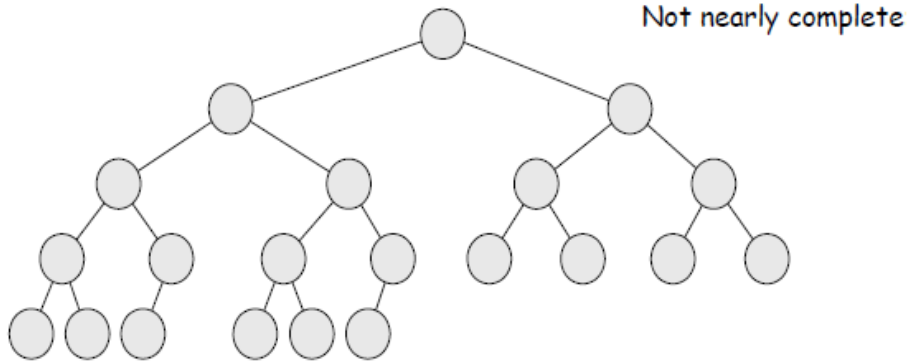
Definition: A *complete* binary tree of height h is a binary tree which contains exactly 2^d nodes at depth d, $0 \le d \le h$.

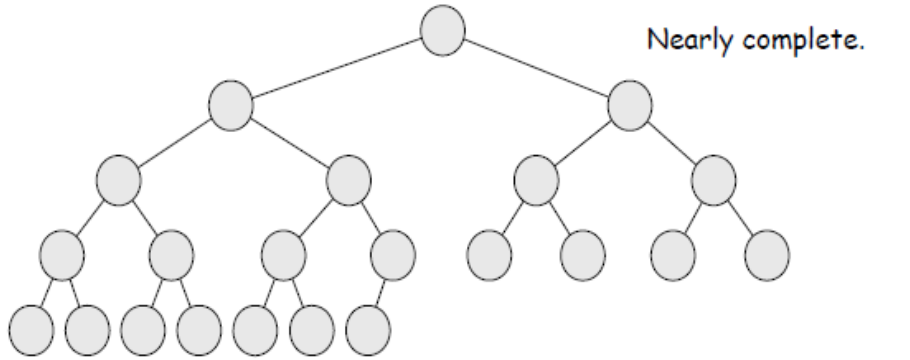
Definition: A *nearly complete* binary tree of height h is a binary tree of height h in which:

a) There are 2^d nodes at depth d for d = 1,2,...,h−1,

b) The nodes at depth h are as far left as possible.





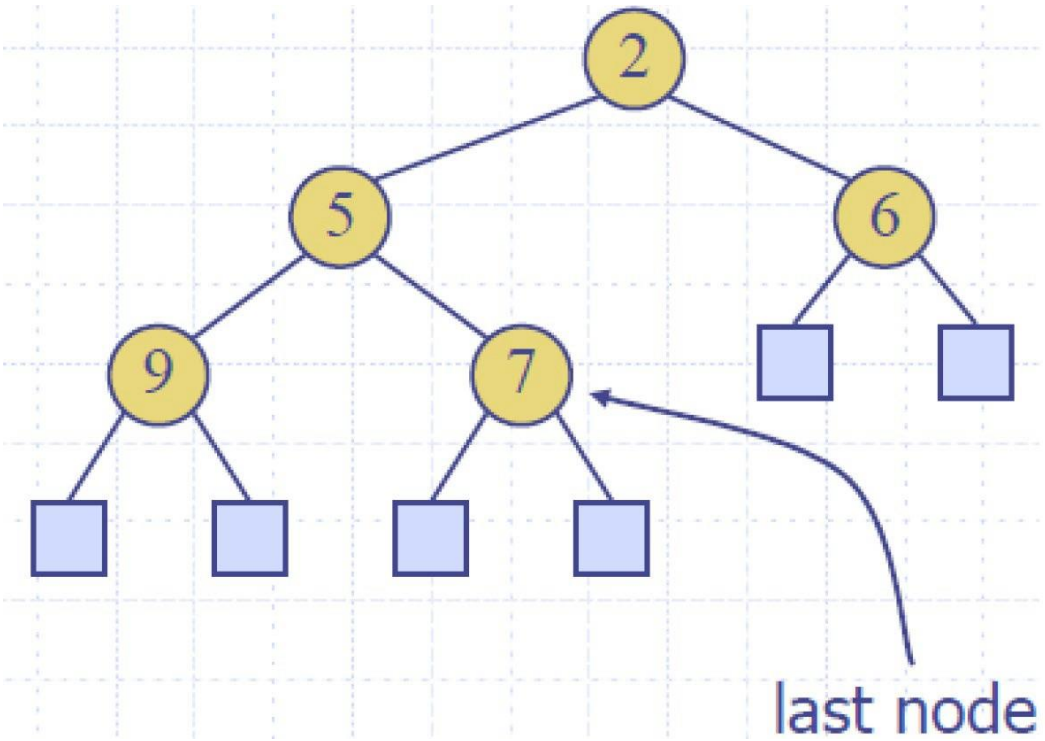


Heap Data Structure

❖A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:

- Heap-Order: for every internal node v other than the root,
  key(v) ≥key(parent(v))

The Min Heap



Heap Data Structure

Binary heap. Array representation of a heap-ordered complete binary tree

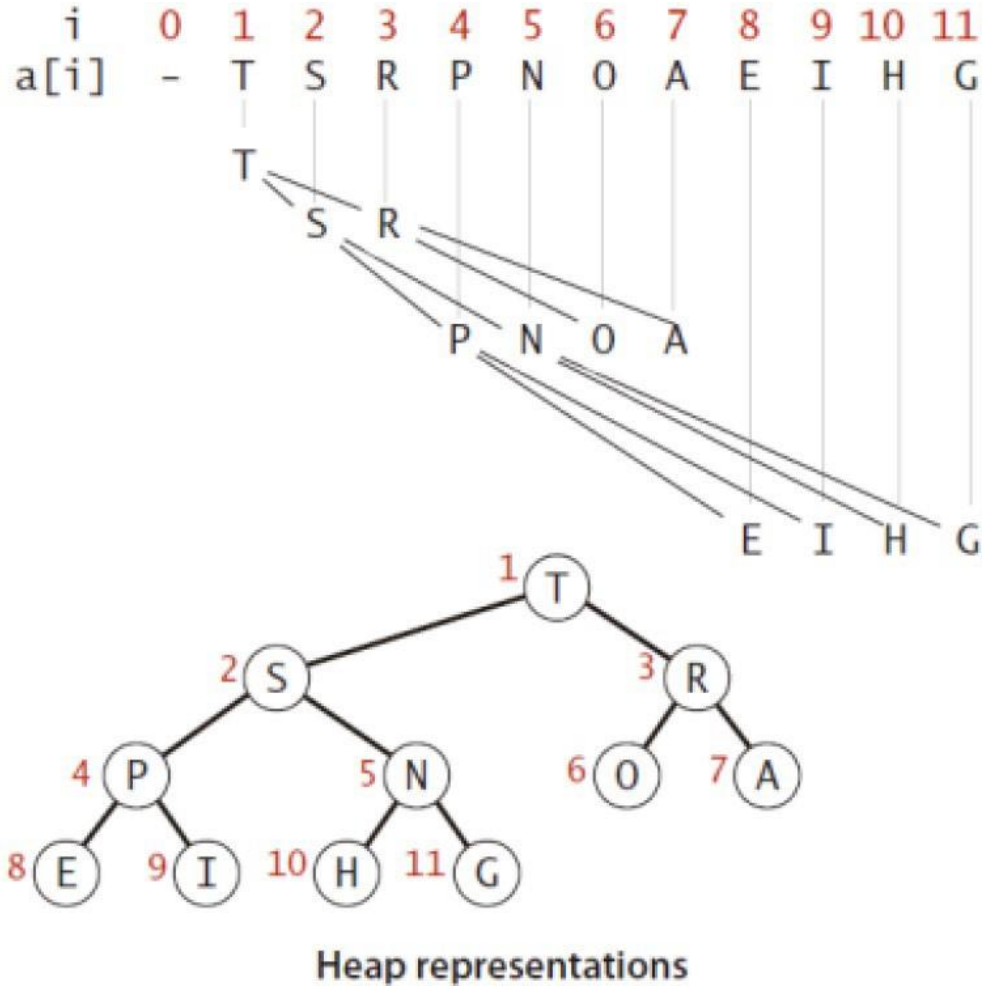
Heap-ordered binary tree.

- Keys in nodes.
- Parent's key no larger than children's keys.

Array representation.

- Indices start at 1 .
- Take nodes in level order.
- No explicit links needed !

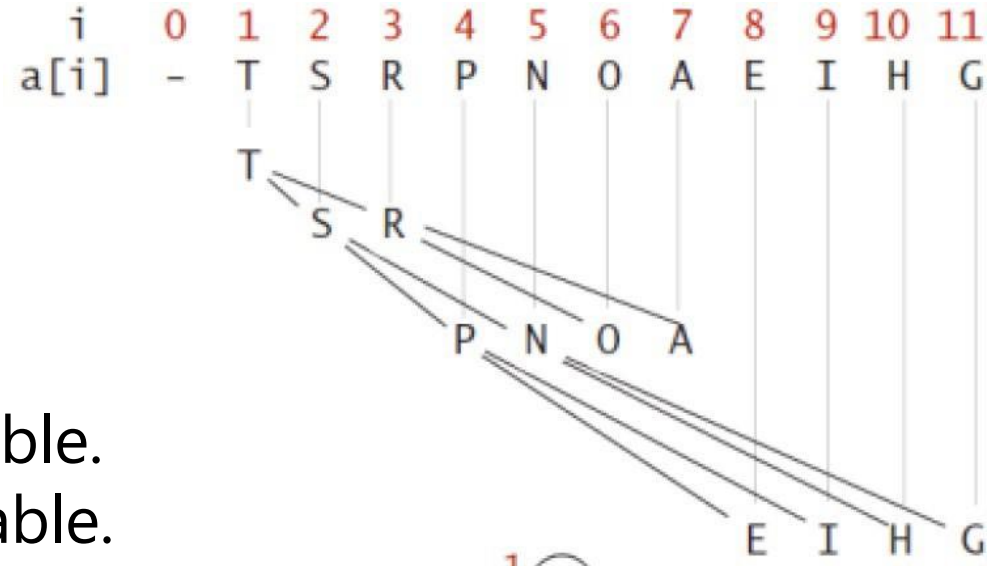
An efficient realization of a heap can be achieved using an array for storing the elements.



Heap representations

Heap Data Structure

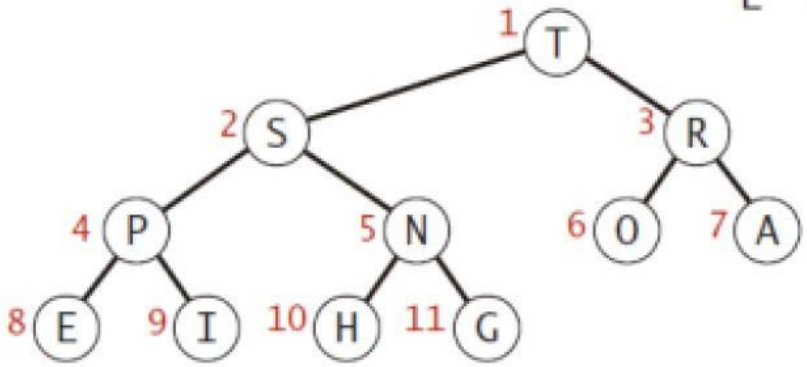
Binary heap. Array representation of a heap-ordered complete binary tree



For any given node at position i:

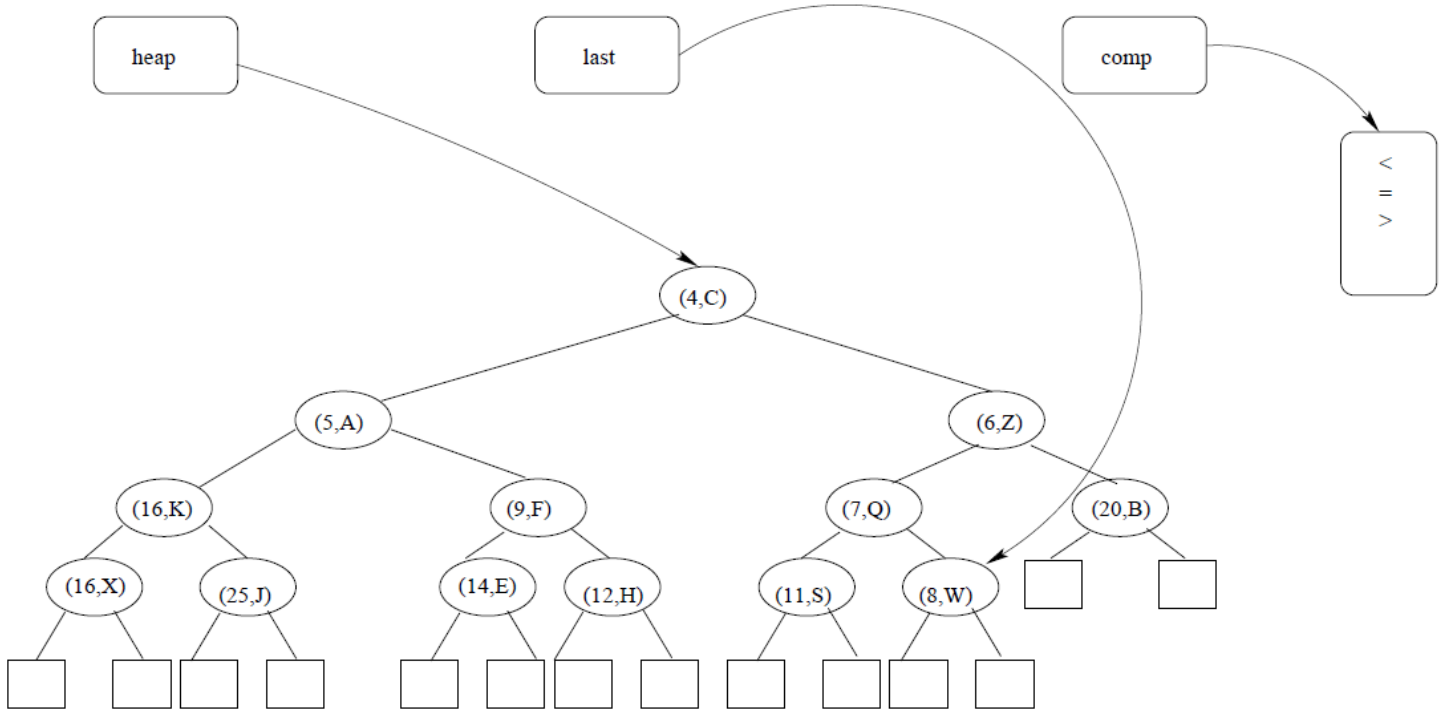
- Its **Left Child** is at **[2*i]** if available.
- Its **Right Child** is at **[2*i+1]** if available.
- Its **Parent Node** is at **[⌊i/2⌋]** if available.

An efficient realization of a heap can be achieved using an array for storing the elements.



Heap representations

PQ/Heap implementation



heap: A (nearly complete) binary tree T containing elements with keys satisfying the heap-order property, stored in an array.

last: A reference to the last used node of T in this array representation.

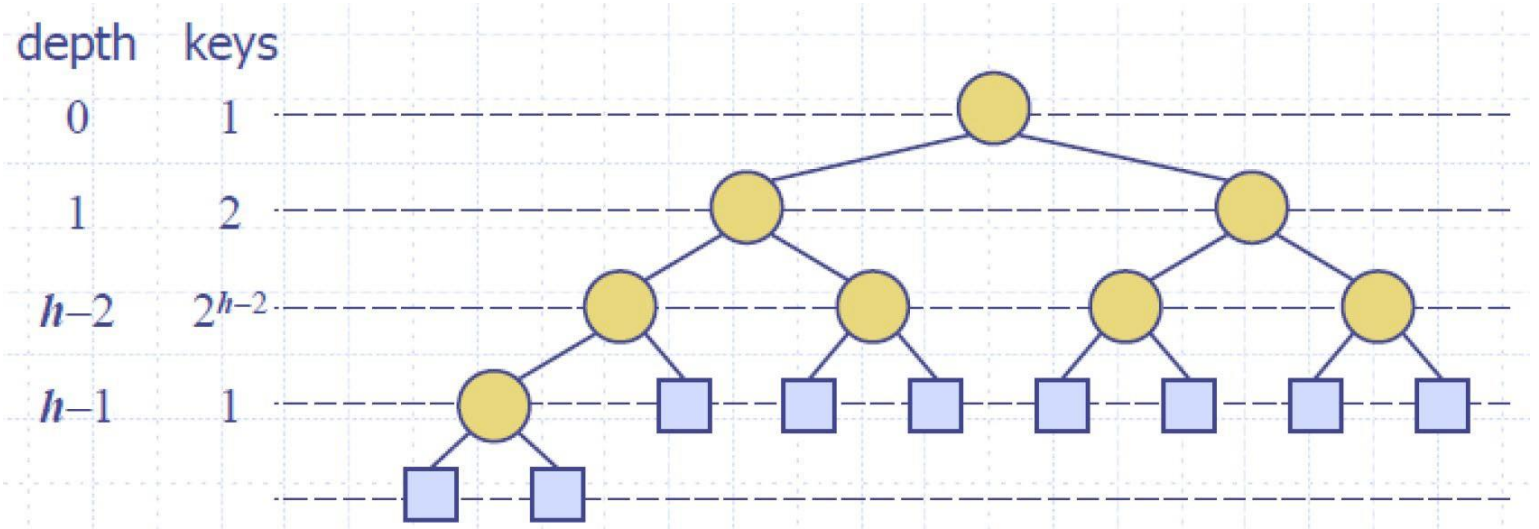
comp: A comparator function that defines the total order relation on keys and which is used to maintain the minimum (or maximum) element at the root of T .

PQ/Heap implementation

❖Theorem: A heap storing n keys has height $\boldsymbol{O}(\log n)$

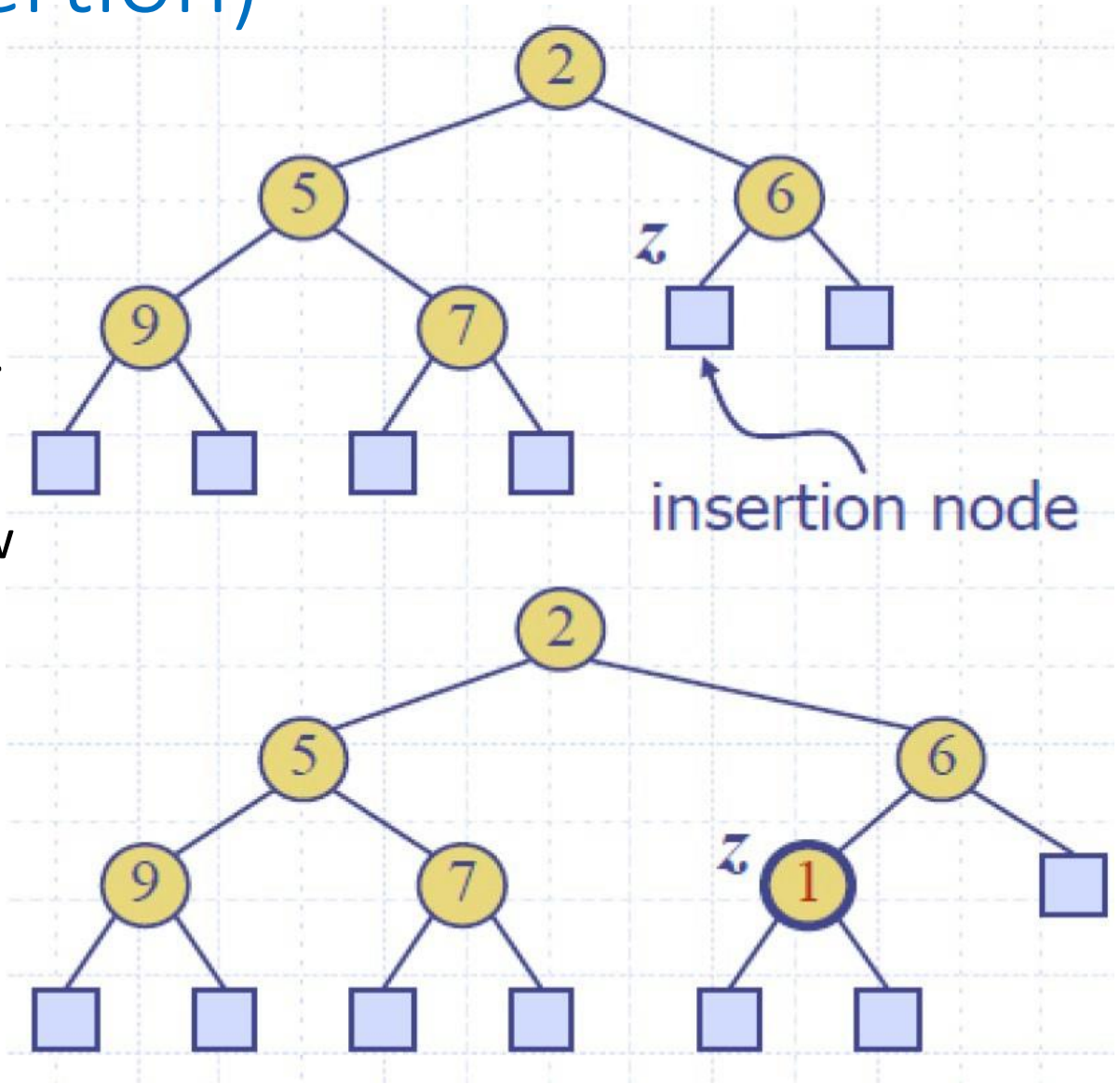
Proof: (we apply the complete binary tree property)

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $\mathrm{n} \geq 1 + 2 + 4 + \cdots + 2^{h-2} + 1$
- Thus, $\mathrm{n} \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



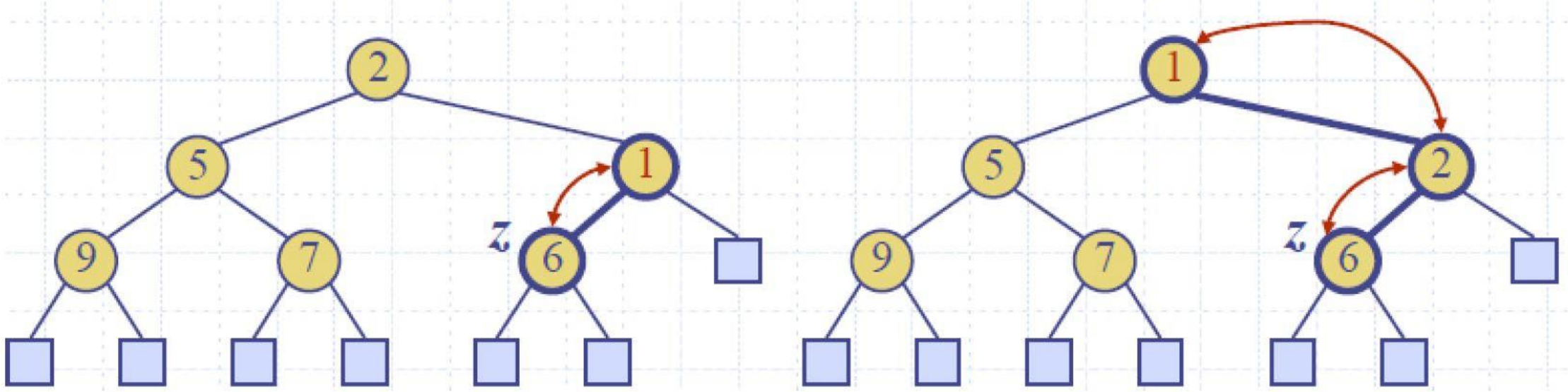
Up-heap bubbling (insertion)

- Method *insertItem* of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z and expand z into an internal node
  - Restore the heap-order property (discussed next)



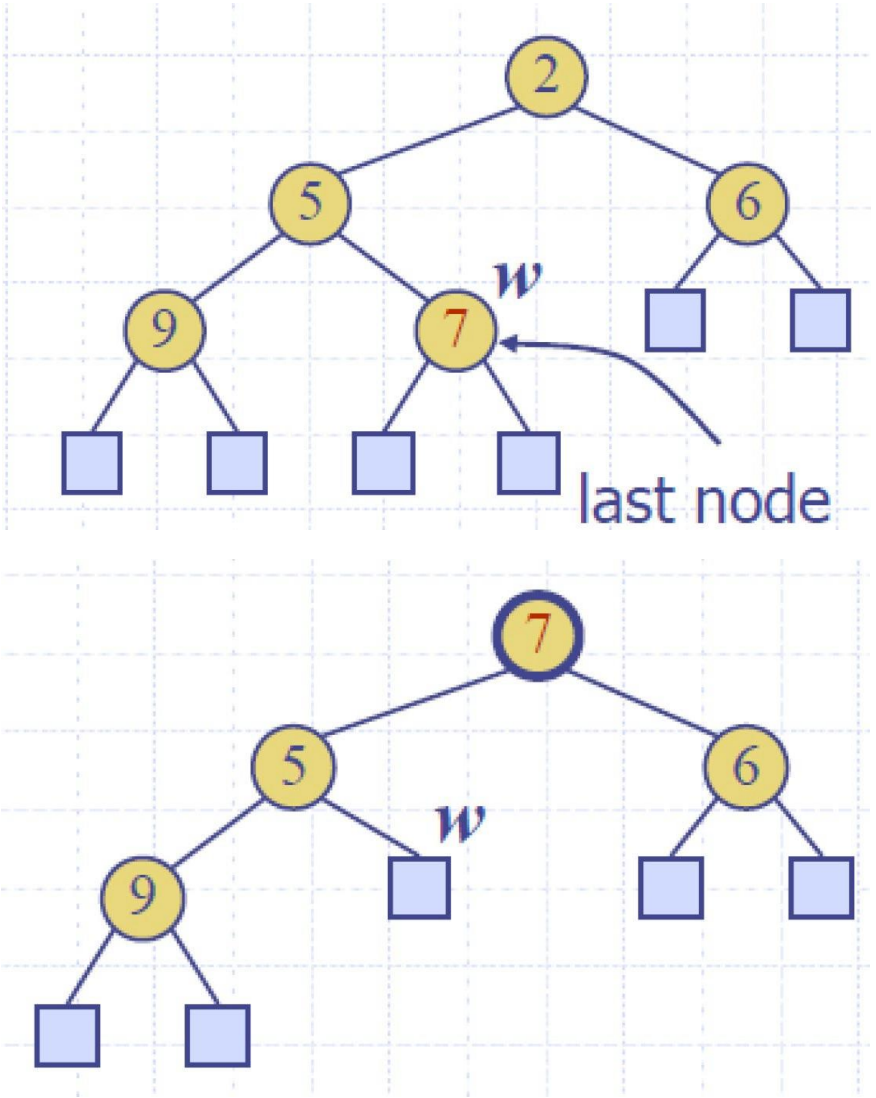
Up-heap bubbling (insertion) (cont.)

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



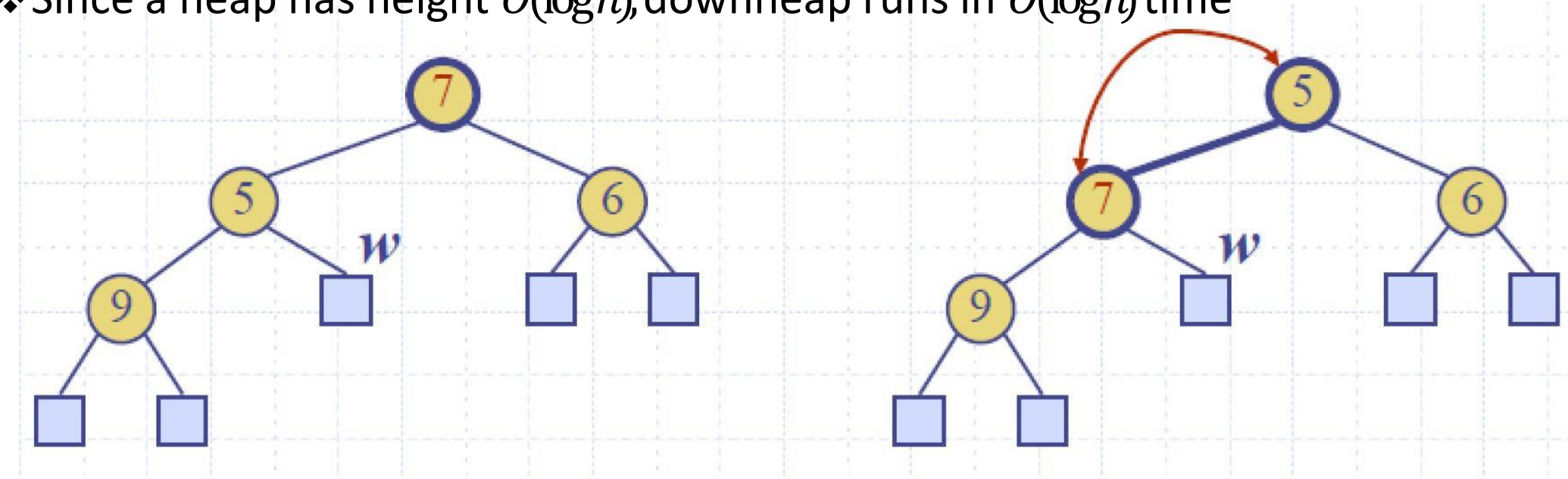
Down-heap bubbling (removal of top element)

- Method *removeMin* of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node *w*
  - Compress *w* and its children into a leaf
  - Restore the heap-order property (discussed next)



Down-heap bubbling (cont.)

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



Heap-Sorting

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods *insertitem* and *removeMin* take $O(\log n)$ time
  - methods *size, isEmpty, minKey*, and *minElement* take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort

Divide-and-Conquer

The divide-and-conquer method is a means that can be used to solve some algorithmic problems. This general method consists of the following steps:

- *Divide*: If the input size is *small* then solve the problem directly; otherwise, divide the input data into two or more *disjoint* subsets.
- *Recur*: *Recursively* solve the sub-problems associated with subsets.
- *Conquer*: Take the solutions to sub-problems and *merge* into a solution to the original problem.

MergeSort

Merge-sort on an input sequence S with n elements consists of three steps:

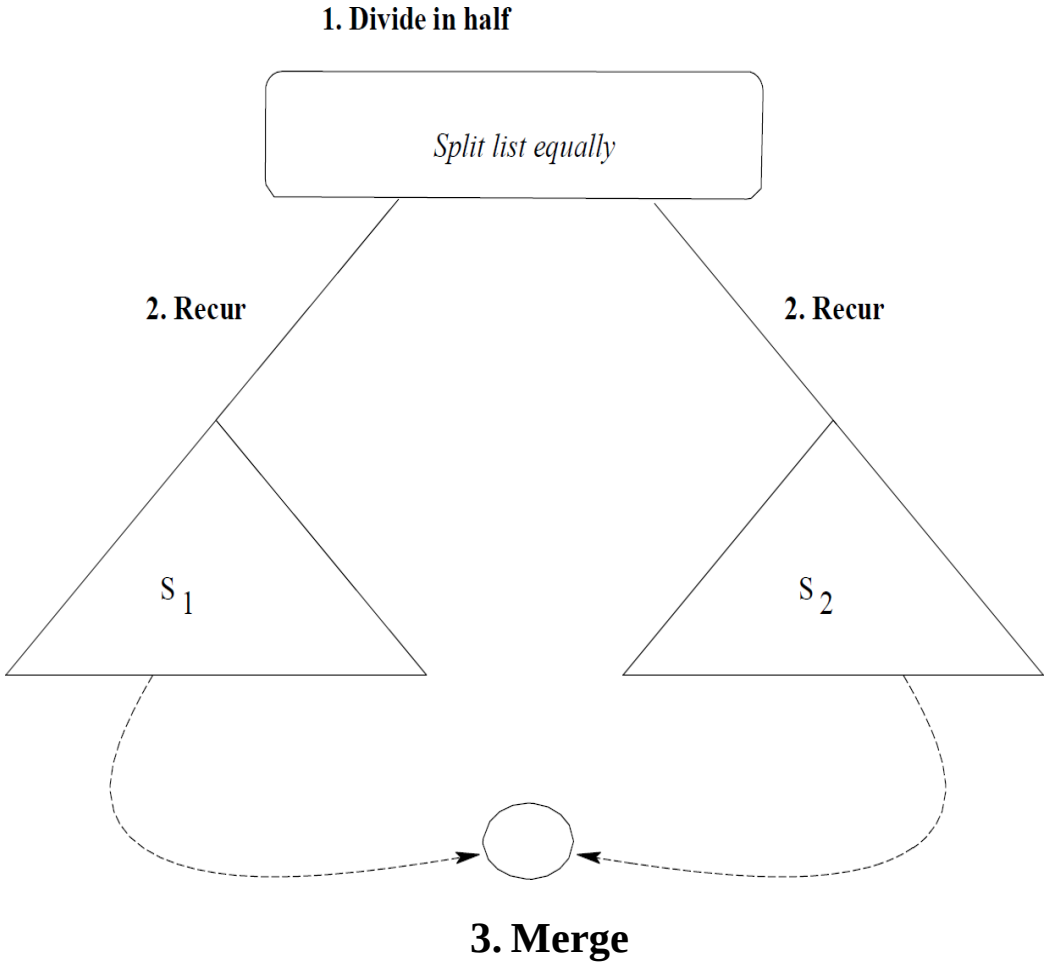
- Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
- Recur: recursively sort S_1 and S_2
- Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort(S, C)*
  Input sequence S with n elements, comparator C
  Output sequence S sorted according to C
  If $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1, C)$
    $mergeSort(S_2, C)$
    $S \leftarrow merge(S_1, S_2)$

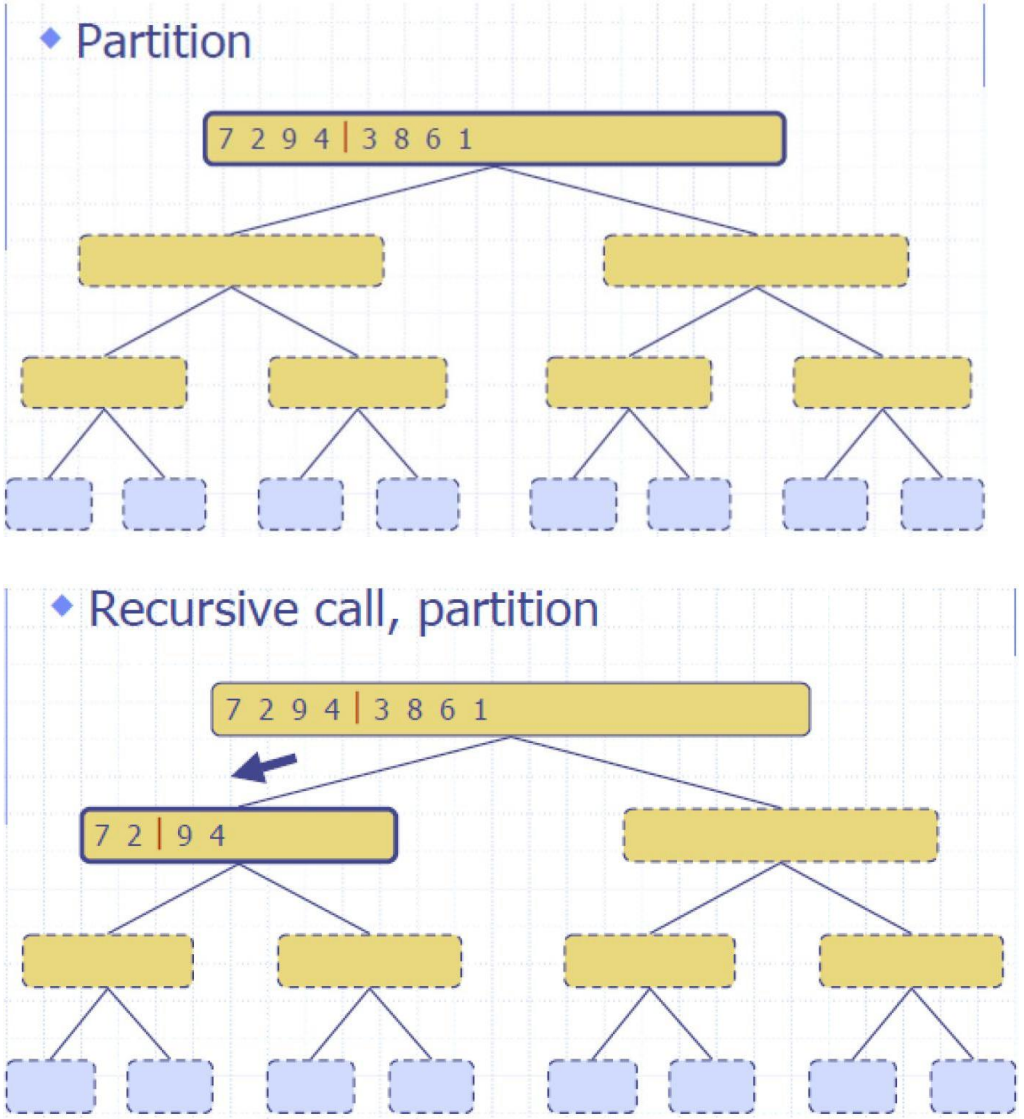
MergeSort - Illustration

Merge-sort on an input sequence S with n elements consists of three steps:

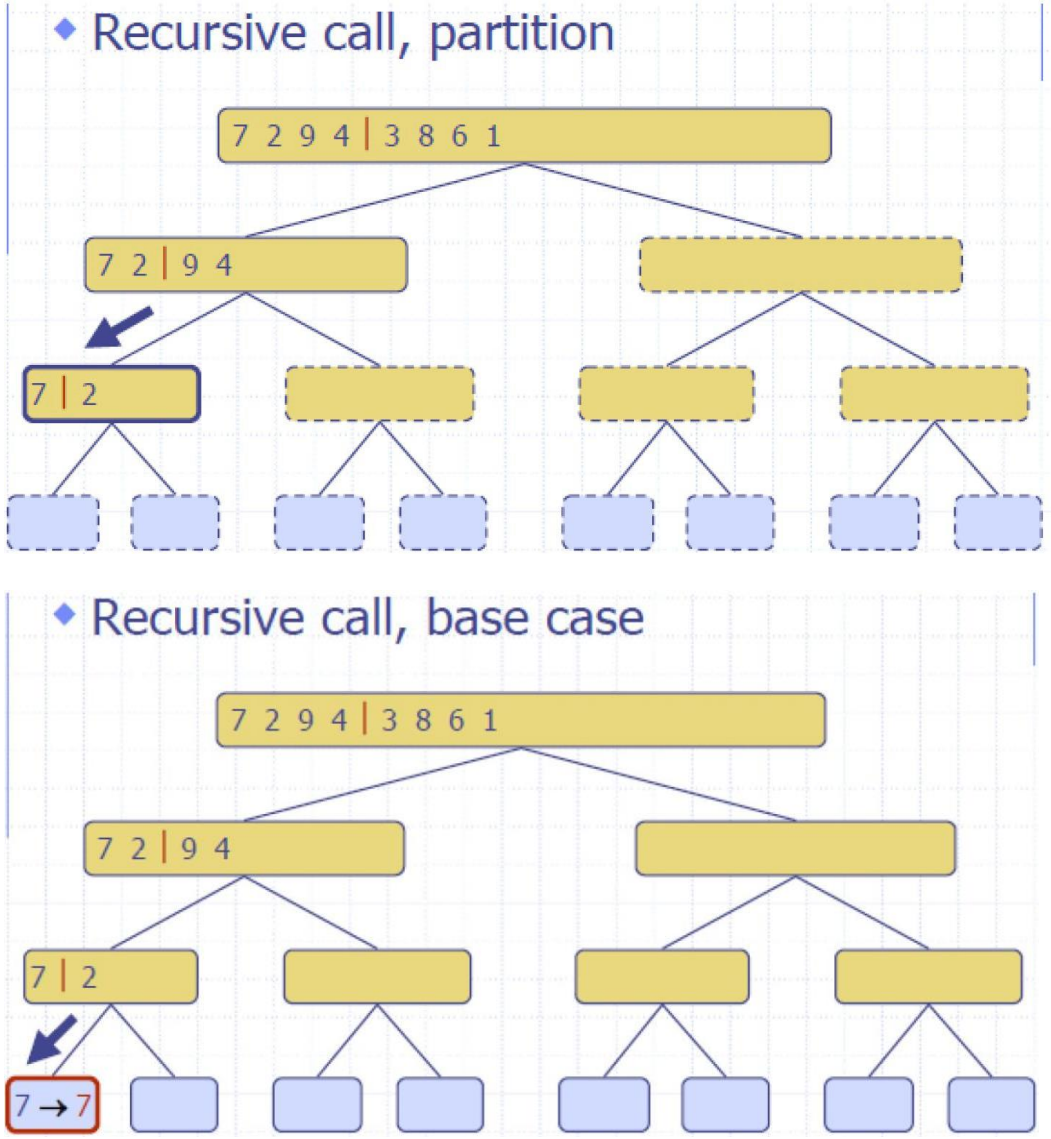
- Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
- Recur: recursively sort S_1 and S_2
- Conquer: merge S_1 and S_2 into a unique sorted sequence



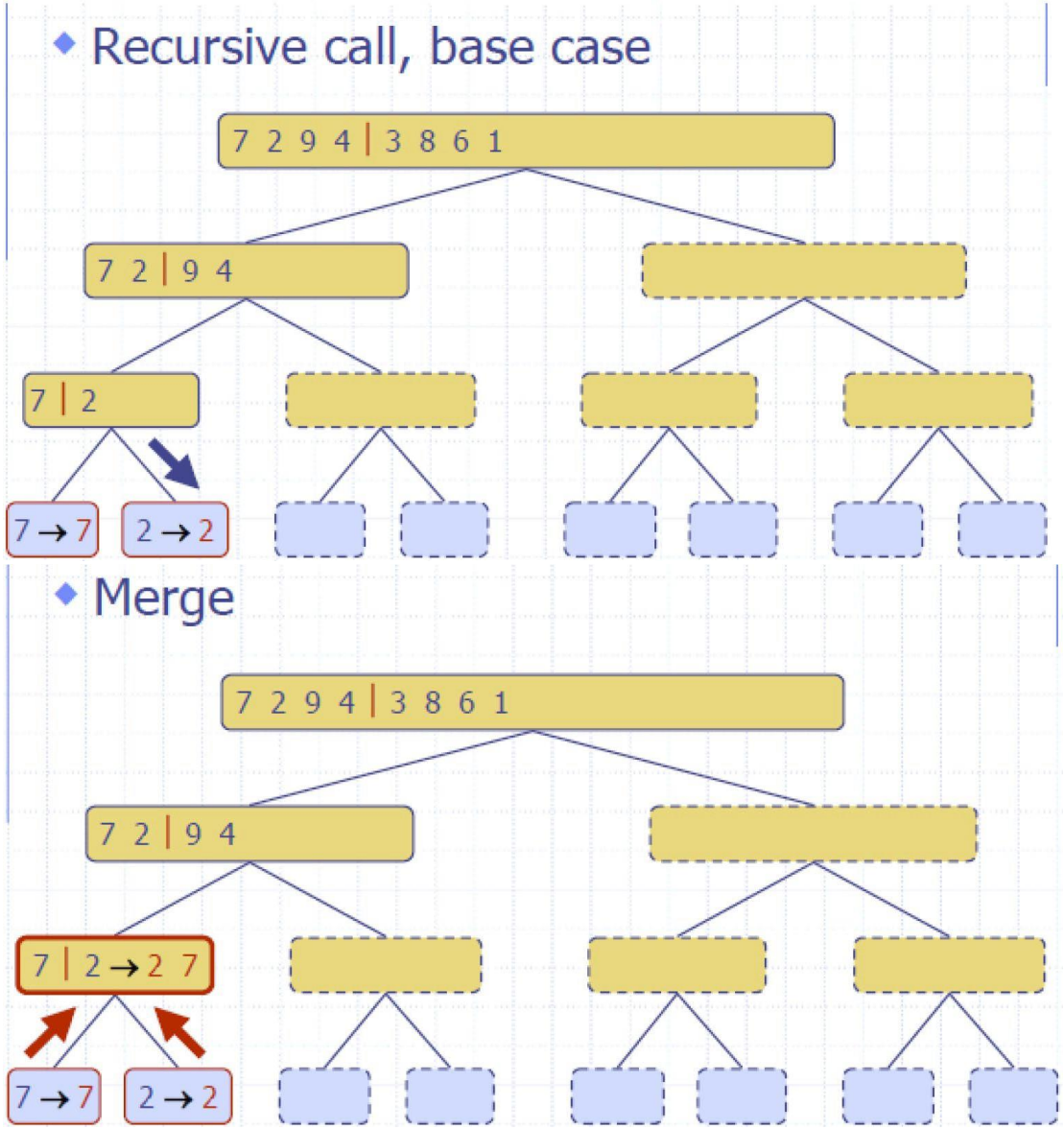
MergeSort - Example



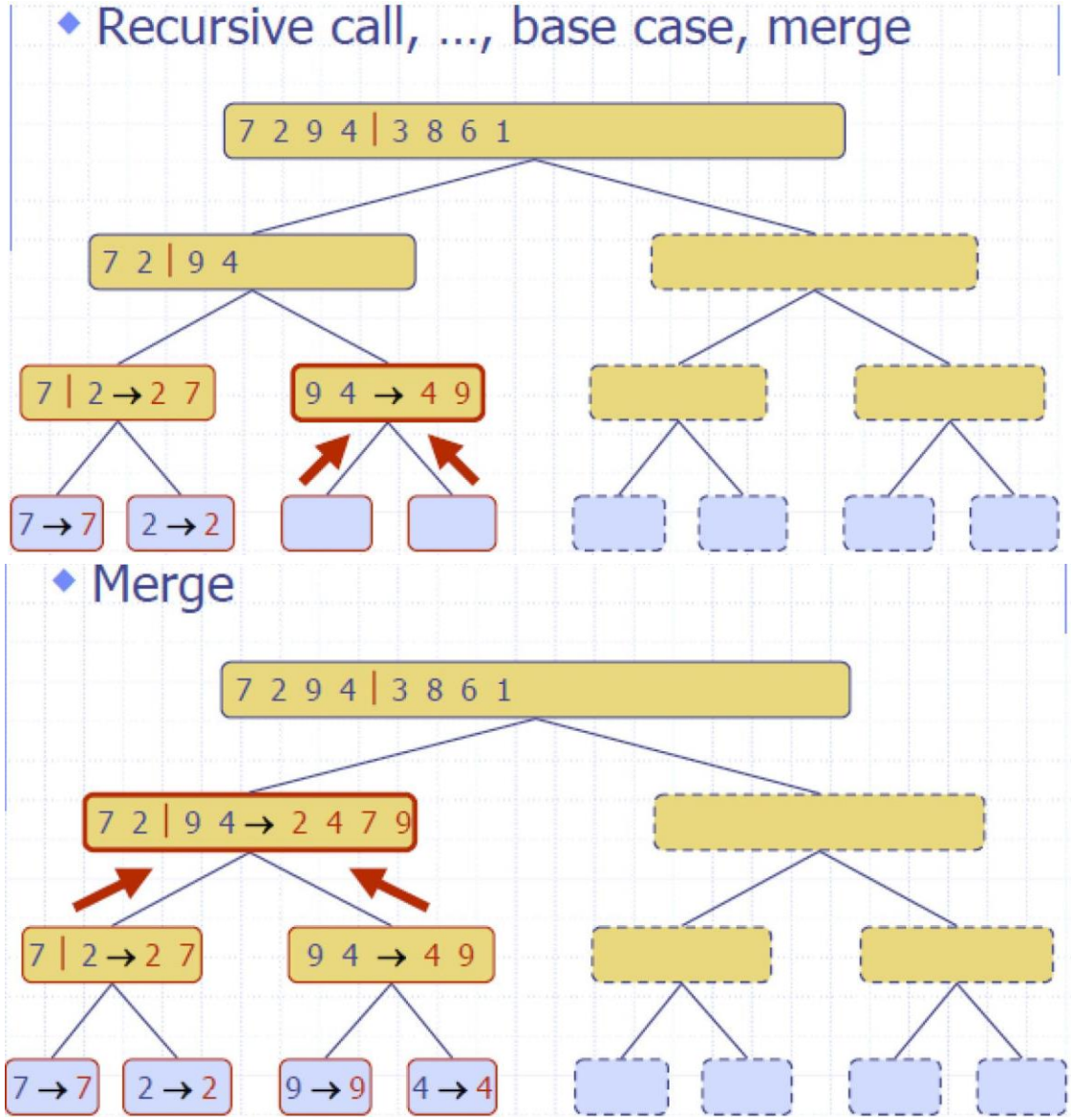
MergeSort - Example



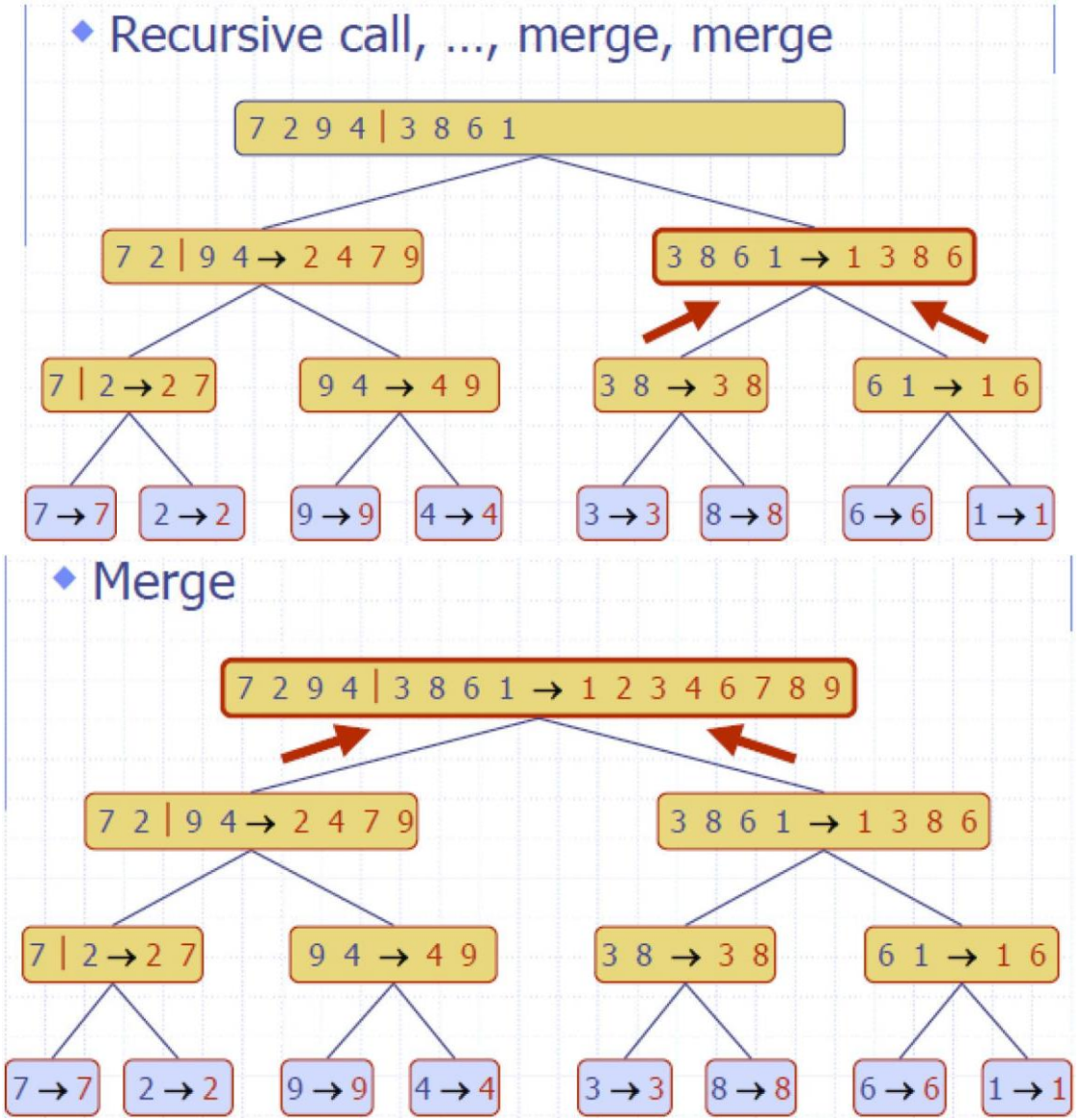
MergeSort - Example



MergeSort - Example



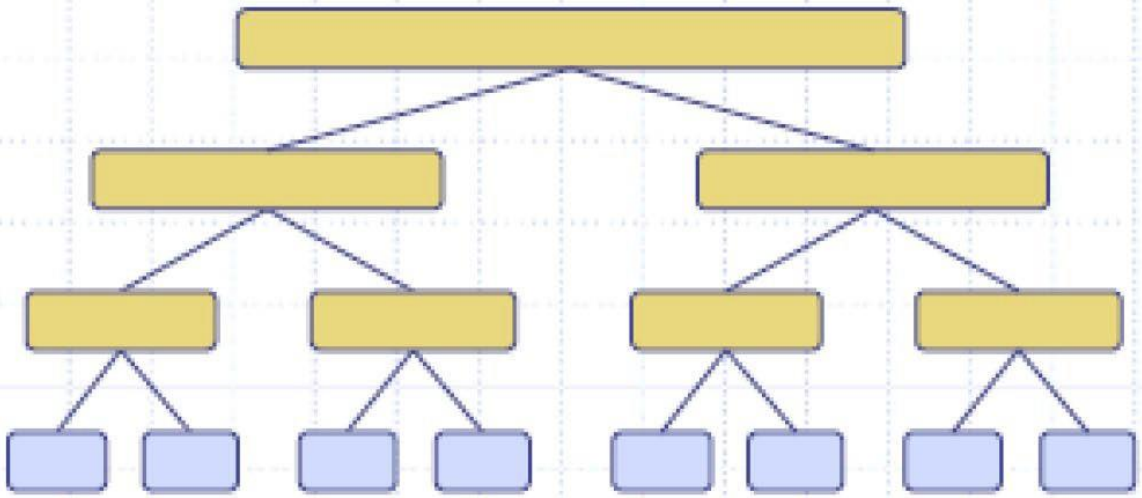
MergeSort - Example



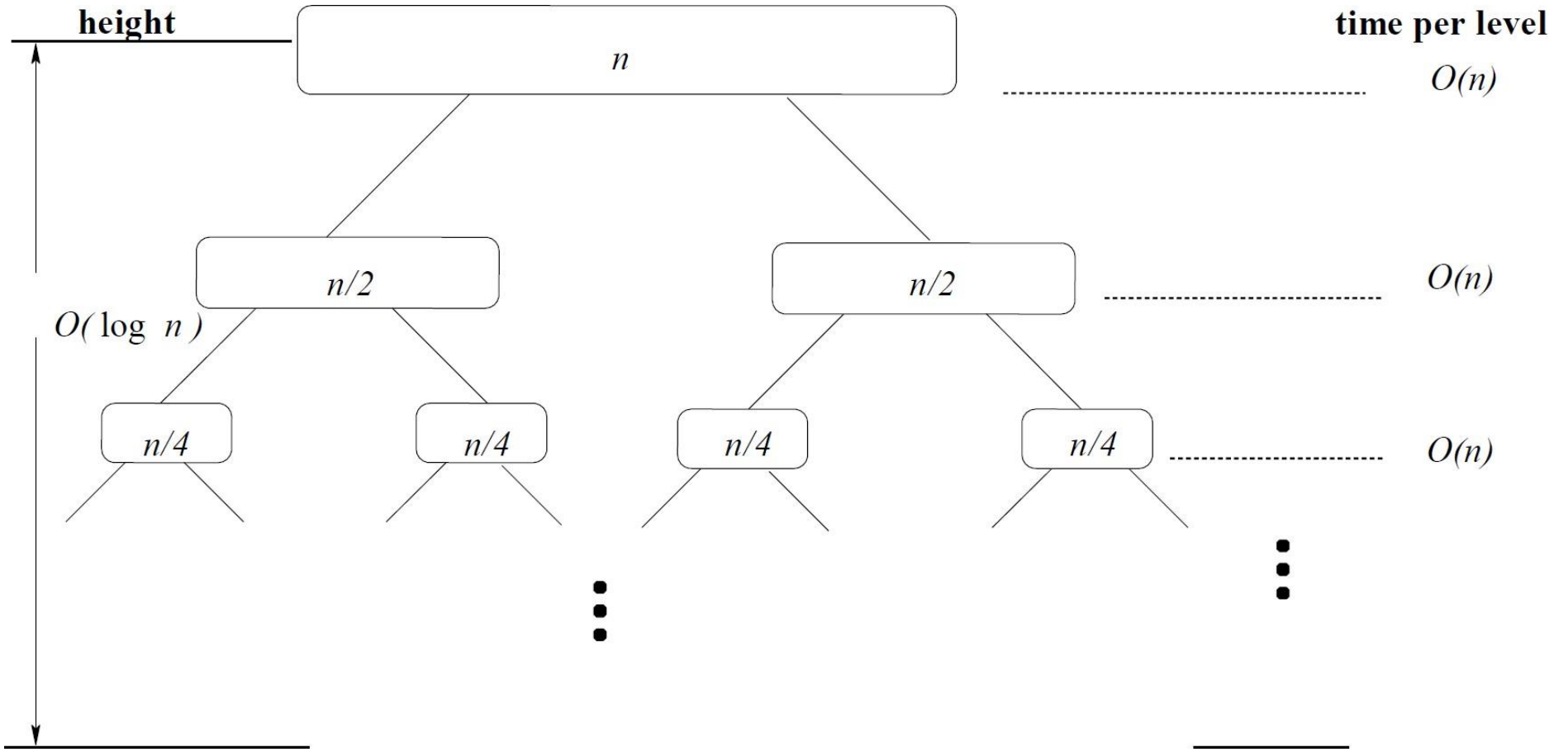
MergeSort - Analysis

- 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 or work done at the nodes of depth i 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	$n/2^i$
…	…	…



MergeSort - Analysis

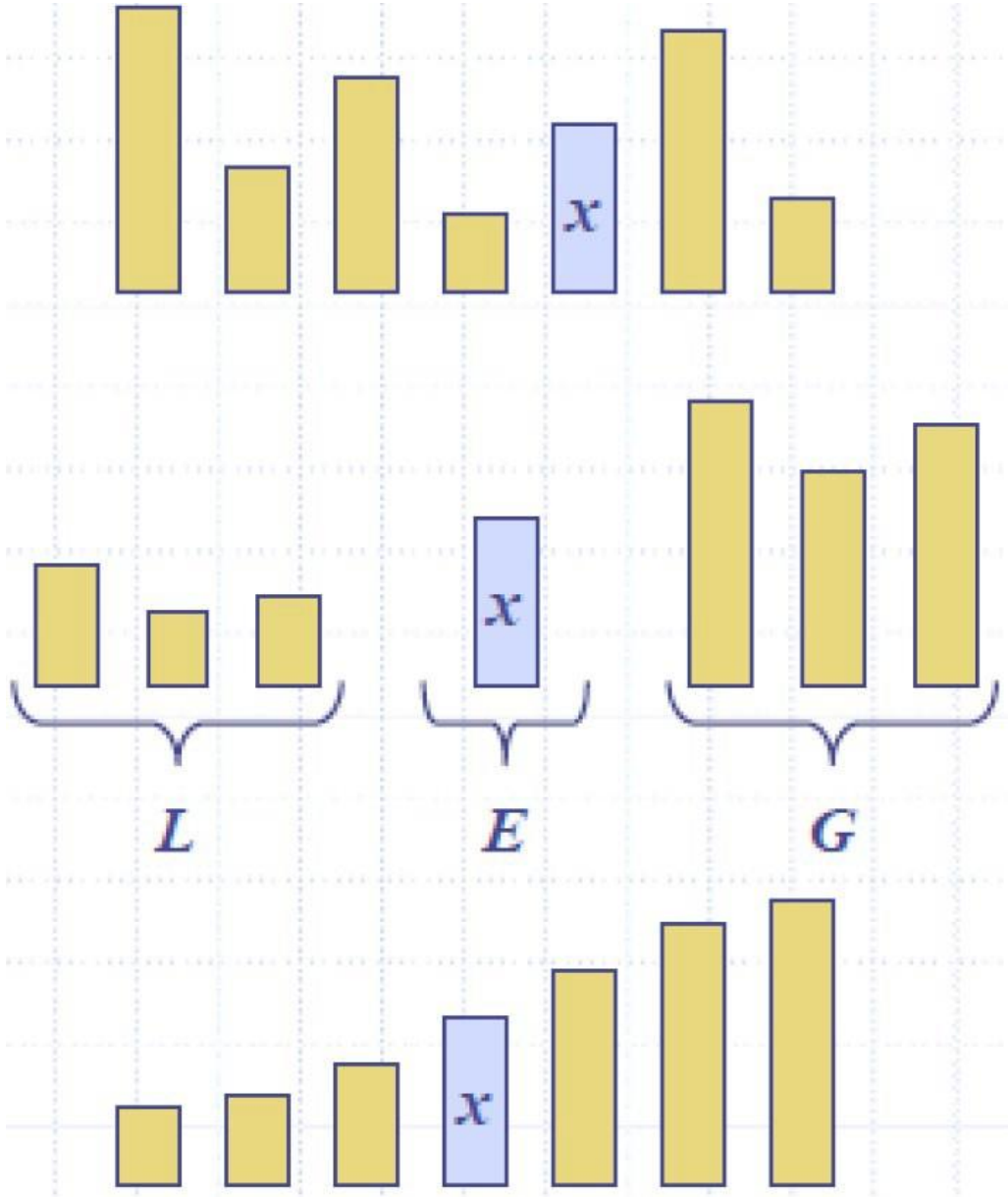


Total time: $O(n \log n)$

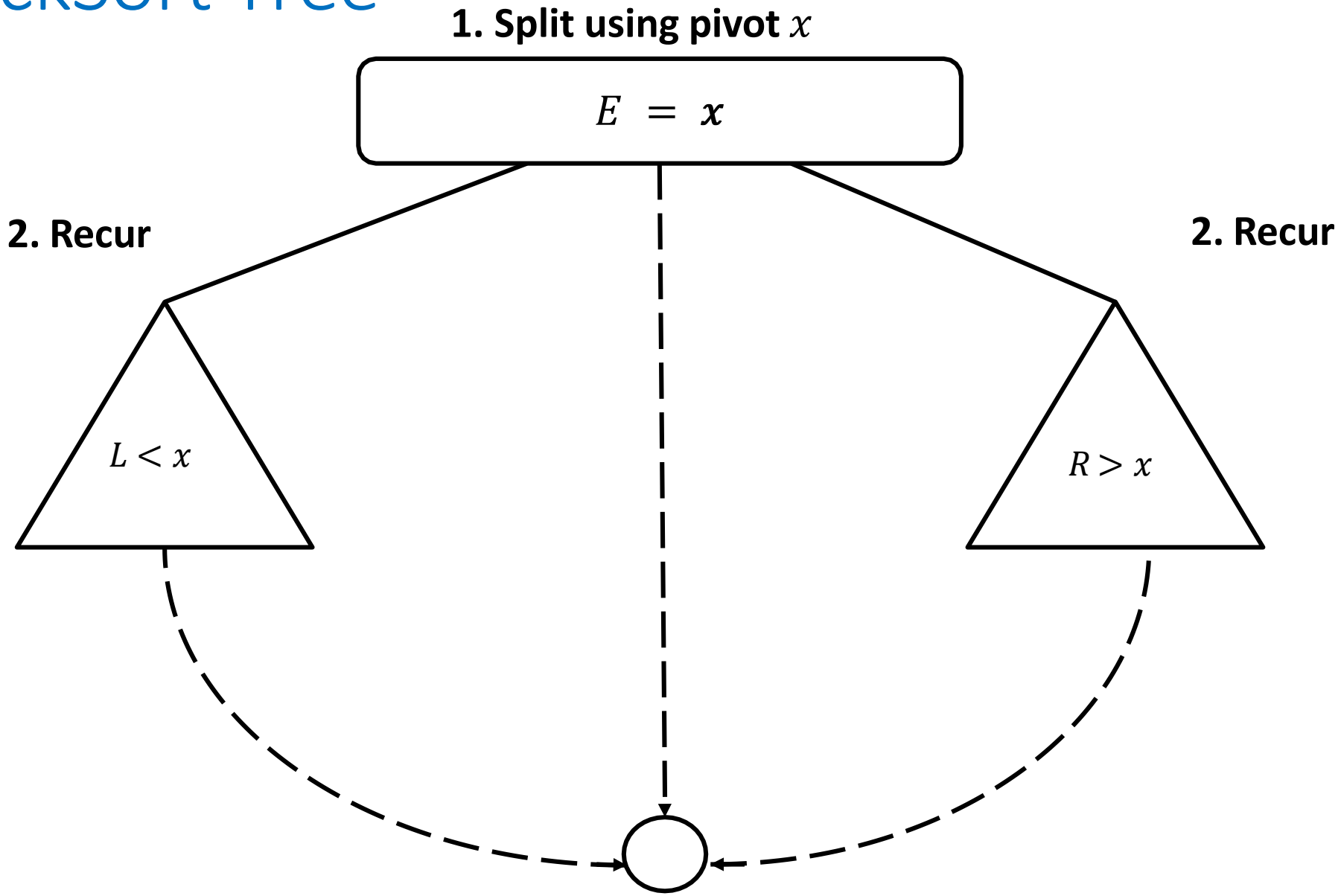
QuickSort

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



QuickSort Tree

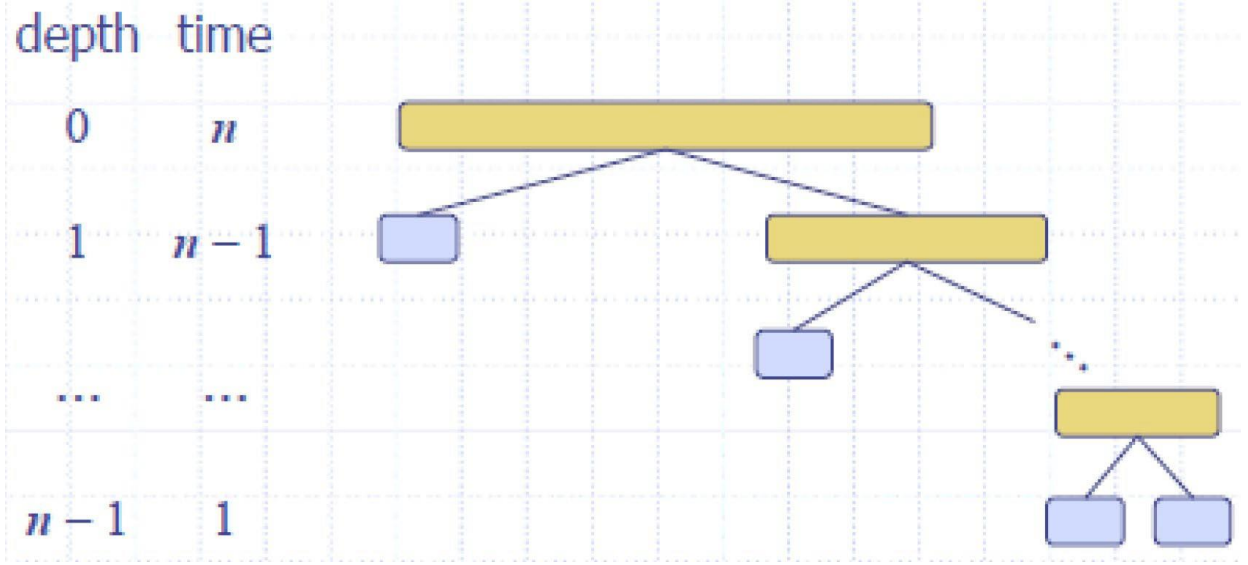


QuickSort -worst-case running time

- 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 $O(n^2)$



INT202
Complexity of Algorithms
Fundamental Techniques

XJTLU/SAT/INT

SEM2 AY2021-2022

Divide-and-Conquer

We have already discussed the divide-and-conquer method when we talked about sorting. To remind you, here is the general outline for using this method:

- *Divide*: If the input size is small then solve directly, otherwise divide the input data into two or more disjoint subsets.
- *Recur*: Recursively solve the sub-problems associated with the subsets.
- *Conquer*: Take the solutions to the sub-problems and merge them into a solution to the original problem.

Divide-and-Conquer

To analyze the running time of a divide-and-conquer algorithm we typically utilize a *recurrence relation*, where $T(n)$ denotes the running time on an input of size n.

We then want to characterize $T(n)$ using an equation that relates $T(n)$ to values of function T for problem sizes smaller than n, e.g.

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{otherwise} \end{cases}$$

Substitution Method

In the iterative substitution, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$
\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2)) + b(n/2)) + bn \\
&= 2^2 T(n/2^2) + 2bn \\
&= 2^3 T(n/2^3) + 3bn \\
&= 2^4 T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^i T(n/2^i) + ibn
\end{aligned}
$$

Note that base, T(n)=b, case occurs when $2^i=n$. That is, i = log n.

So, $T(n) = bn + bn \log n$

Thus, T(n) is O(n log n).

The Master Method

It is a "cook-book" method to solve

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{otherwise.} \end{cases}$$

wherein $d \geq 1, a > 0, c > 0, b > 1$

The Master Method

It is a "cook-book" method to solve

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

The Master Method (cont.)

In the **The Master Theorem**:

1. Case 1: applies where $f(n)$ is polynomially smaller than the special function $n^{\log_b a}$.
2. Case 2: applies where $f(n)$ is asymptotically close to the special function $n^{\log_b a}$.
3. Case 3: applies where $f(n)$ is polynomially larger than the special function $n^{\log_b a}$.

*f(n) is polynomially smaller than g(n) if $f(n)=O(g(n)/n^{\epsilon})$ for some $\epsilon>0$.

*f(n) is polynomially larger than g(n) if $f(n)=\Omega(g(n)n^{\epsilon})$ for some $\epsilon>0$

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

The Master Method (cont.)

$$T(n) = aT(n/b) + f(n)$$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example 5.7: *Consider the recurrence*

$$T(n) = 4T(n/2) + n.$$

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. *Thus, we are in Case 1, for* $f(n)$ *is* $O(n^{2-\varepsilon})$ *for* $\varepsilon = 1$. *This means that* $T(n)$ *is* $\Theta(n^2)$ *by the master method.*

The Master Method (cont.)

$$T(n) = aT(n/b) + f(n)$$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

$$T(n) = T(2n/3) + 1,$$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

The Master Method (cont.)

$$T(n) = aT(n/b) + f(n)$$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Example 5.9: *Consider the recurrence*

$$T(n) = T(n/3) + n,$$

which is the recurrence for a geometrically decreasing summation that starts with n. In this case, $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. Thus, we are in Case 3, for $f(n)$ is $\Omega(n^{0+\varepsilon})$, for $\varepsilon = 1$, and $af(n/b) = n/3 = (1/3)f(n)$. This means that $T(n)$ is $\Theta(n)$ by the master method.

The Master Method (cont.)

$$T(n) = aT(n/b) + f(n)$$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

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

even though it appears to have the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. You might mistakenly think that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^{ϵ} for any positive constant ϵ.

The Master Method (cont.)

Example 5.11: *Finally, consider the recurrence*

$$T(n) = 2T(n^{1/2}) + \log n.$$

Unfortunately, this equation is not in a form that allows us to use the master method. We can put it into such a form, however, by introducing the variable $k = \log n$, *which lets us write*

$$T(n) = T(2^k) = 2T(2^{k/2}) + k.$$

Substituting into this the equation $S(k) = T(2^k)$, *we get that*

$$S(k) = 2S(k/2) + k.$$

Now, this recurrence equation allows us to use master method, which specifies that $S(k)$ *is* $O(k \log k)$. *Substituting back for* $T(n)$ *implies* $T(n)$ *is* $O(\log n \log \log n)$.

Matrix Multiplication: An example

Question Suppose we are given two $n \times n$ matrices X and Y, and we wish to compute their product Z = XY, which is defined so that

$$Z[i,j] = \sum_{k=0}^{n-1} X[i,k] \cdot Y[k,j]$$

which is an equation that immediately gives rise to a simple $O(n^3)$ time algorithm.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Matrix Multiplication: An example

Submatrices Suppose n is a power of two and let us partition X, Y, and Z each into four $(n/2)\times(n/2)$ matrices, so that we can rewrite Z = XY as

$$\begin{pmatrix} I & J \\ K & L \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

Thus,

$$I = AE + BG$$
$$J = AF + BH$$
$$K = CE + DG$$
$$L = CF + DH$$

Matrix Multiplication: An example

Divide-and-conquer algorithm computes $Z = XY$ by computing I, J, K, and L from the subarrays A through G.

$$I = AE + BG$$
$$J = AF + BH$$
$$K = CE + DG$$
$$L = CF + DH$$

By the above equations, we can compute I, J, K, and L from the eight recursively computed matrix products on $(n/2)\times(n/2)$ subarrays, plus four additions that can be done in $O(n^2)$ time.

Thus, the above set of equations give rise to a divide-and-conquer algorithm whose running time T(n) is characterized by the recurrence

$$T(n) = 8T\left(\frac{n}{2}\right) + bn^2$$

for some constant b > 0.

This equation implies: T(n) = $O(n^3)$ by the master theorem.

Matrix Multiplication: An example

Strassen's algorithm organises arithmetic involving the subarrays A through G so that we can compute I, J, K, and L using just seven recursive matrix multiplications (by Volker Strassen 1969).

Define seven submatrix products:

$$
\begin{aligned}
S_1 &= A(F - H) \\
S_2 &= (A + B)H \\
S_3 &= (C + D)E \\
S_4 &= D(G + E) \\
S_5 &= (A + D)(E + H) \\
S_6 &= (D - E)(G + H) \\
S_7 &= (A - C)(E + F)
\end{aligned}
$$

Given these seven submatrix products, we can compute I, J, K, and L as

$$
\begin{aligned}
I &= S_5 + S_6 + S_4 - S_2 = AE + BG. \\
J &= S_1 + S_2 = AF + BH. \\
K &= S_3 + S_4 = CE + DG. \\
L &= S_1 - S_7 - S_3 + S_5 = CF + DH.
\end{aligned}
$$

Matrix Multiplication: An example

Strassen's algorithm organises arithmetic involving the subarrays A through G so that we can compute $I, J, K,$ and L using just seven recursive matrix multiplications.

Given these seven submatrix products, we can compute $I, J, K,$ and L as

$$I = S_5 + S_6 + S_4 - S_2 = AE + BG.$$
$$J = S_1 + S_2 = AF + BH.$$
$$K = S_3 + S_4 = CE + DG.$$
$$L = S_1 - S_7 - S_3 + S_5 = CF + DH.$$

Thus, we can compute $Z = XY$ using seven recursive multiplications of matrices of size $(n/2)\times(n/2)$. Thus, we characterize the running time $T(n)$ as

$$T(n) = 7T\left(\frac{n}{2}\right) + bn^2$$

for some constant $b > 0$.

By the master theorem, we can multiply two n x n matrices in $O(n^{\log 7})$ time using Strassen's algorithm.

Matrix Multiplication: An example

$$Z[i,j] = \sum_{k=0}^{n-1} X[i,k] \cdot Y[k,j]$$

The exponent of matrix multiplication: smallest number ω such that for all ε>0 $O(n^{\omega+\varepsilon})$ operations suffice

- Standard algorithm ω ≤ 3
- Strassen (1969) ω < 2.81
- Pan (1978) ω < 2.79
- Bini et al. (1979) ω < 2.78
- Schönhage (1981) ω < 2.55
- Pan; Romani; Coppersmith + Winograd (1981-1982) ω < 2.50
- Strassen (1987) ω < 2.48
- Coppersmith + Winograd (1987) ω < 2.375
- Stothers (2010) ω < 2.3737
- Williams (2011) ω < 2.3729
- Le Gall (2014) ω < 2.37286

Counting inversions: Another example

This example is inspired by (if not directly related to) some of the "ranking systems" that are becoming more popular on some websites.

Suppose that you've rated a set of films or books (for example). In particular, you've rated *n* films by ranking them from your most favorite (ranked at 1) to least favorite (ranked at *n*).

In order to give a recommendation to you, this website wants to compare your ratings of these films with those of other people (for the same films) to see how similar they are.

How can you do this?

Counting inversions (cont.)

In other words, how can you compare your rankings

1 2 3 4 5 6 7 8 9 10

to another ranking

2 7 10 4 6 1 3 9 8 5?

Or even to another person's rankings

8 9 10 1 3 4 2 5 6 7?

Which one of these is "closest" to your rankings?

Counting inversions (cont.)

One proposed way of measuring the similarity is to count the number of *inversions*.

Suppose that

$$a_1, a_2, a_3, \ldots, a_n$$

denotes a permutation of the integers 1, 2, . . . , n. The pair of integers i , j are said to form an inversion if $i < j$, but $a_i > a_j$.

(We can generalize this idea to any sequence of distinct integers.)

We will count the number of inversions to measure the similarity of someone's rankings to yours.

Counting inversions (cont.)

For example, the permutation

1 2 4 3

contains one inversion (the 4 and the 3), while the permutation

1 4 3 2

has three (the 3, 4 pair, the 2, 3 and the 2, 4 pair).

In other words, to find the number of inversions, we count the pairs $i \neq j$ that are *out of order* in the permutation.

Counting inversions (cont.)

The number of inversions can range from 0, for the permutation

$$1\ 2\ 3\ \ldots\ n,$$

up to $\binom{n}{2} = \frac{n(n-1)}{2}$ for the permutation

$$n\ \ n-1\ \ldots\ 2\ \ 1.$$

$$C_n^m = \frac{A_n^m}{m!} = \frac{n!}{m!\,(n-m)!}$$

Other examples:

2 1 3 4 5 has one inversion,

2 3 4 5 1 has four inversions.

Counting inversions: How do we do it?

So how do we count the number of inversions in a given permutation of n numbers?

The "naive" approach is to check all $\binom{n}{2}$ pairs to see if they form an inversion in the permutation. This gives an algorithm with $O(n^2)$ running time.

Can we do better?

Claim: We can count inversions using a divide-and-conquer algorithm that runs in time O(n log n).

A divide-and-conquer way to count inversions

Idea:

As with similar divide-and-conquer algorithms, we divide the permutation into two (nearly equal) parts. Then we (recursively) count the number of inversions in each part.

This gives us most of the inversions. We then need to get the number of inversions that involve one element of the first list, and one element of the second.

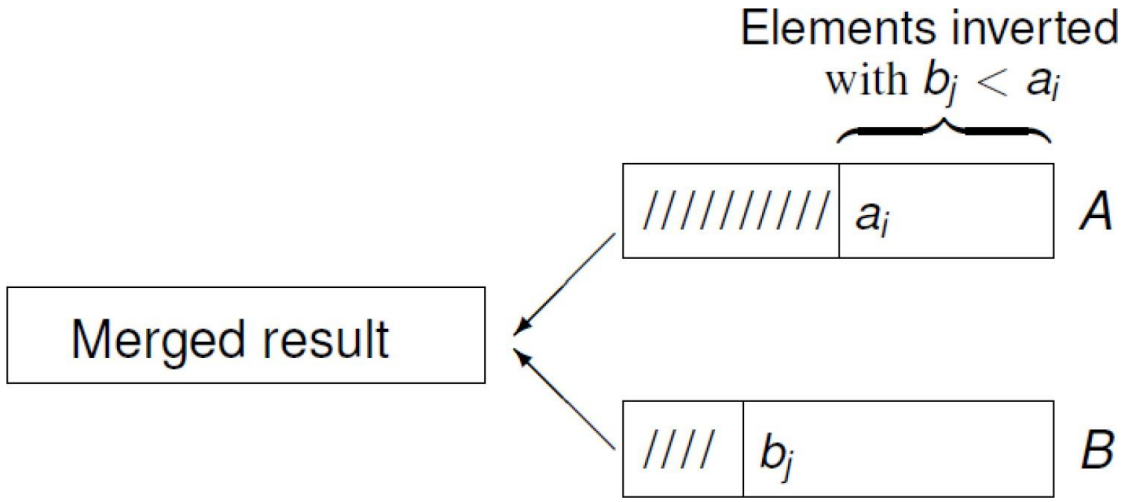
To do that we *sort* each sublist and merge them into a single (sorted) list. As we merge them together into a single list, we can count the inversions from such pairs mentioned above.

In other words, we're performing a modified MergeSort!

Divide-and-conquer for counting inversions

Suppose that we've divided the list into *A* (the first half) and *B* (the second half) and have counted the inversions in each.

After sorting them, the idea for counting the additional inversions is as follows:



As we merge the lists, every time we take an element from the list *B*, it forms an inversion with all of the *remaining* (unused) elements in list *A*.

A recursive algorithm for counting inversions

COUNT**I**NVERSIONS(L)

```
    ▷ Input: A list, L, of distinct integers.
    ▷ Output: The number of inversions in L.
1   if L has one element in it then
2           there are no inversions, so Return (0, L)
3   else
            ▷ Divide the list into two halves
4               A contains the first ⌊n/2⌋ elements
5               B contains the last ⌈n/2⌉ elements
6           (k_A, A) = CountInversions(A)
7           (k_B, B) = CountInversions(B)
8           (k, L) = MergeAndCount(A, B)
9           Return (k_A + k_B + k, L)
```

The MERGEANDCOUNT method

MERGE**A**ND**C**OUNT(A, B)

```
1   Current_A ← 0
2   Current_B ← 0
3   Count ← 0
4   L ← empty list
5   while both lists (A and B) are non-empty
6           Let a_i and b_j denote the elements pointed to
                    by Current_A and Current_B.
7           Append the smaller of a_i and b_j to L.
8           if b_j is the smaller element then
9                   Increase Count by the number of elements
                            remaining in A.
10          Advance the Current pointer of the appropriate list.
11  Once one of A and B is empty, append the remaining
            elements to L.
12  Return (Count, L)
```

Counting inversions - The payoff

As mentioned earlier, this method for counting inversions is basically a modified version of the MergeSort algorithm.

Hence, we can count the number of inversions in a permutation in time $O(n \log n)$.

In terms of the ranking system describe earlier, the number of inversions for a permutation is a measure of how “out of order”

it is as compared to the identity permutation

$$1\ 2\ 3\ \ldots\ n$$

and hence could be used to measure the “similarity” to the identity permutation.