

Algorithmics and Programming II: *Introduction*



Jordi Cortadella and Jordi Petit
Department of Computer Science

Algorithmics and Programming II

- Lecturers:
 - Jordi Cortadella (jordi.cortadella@upc.edu)
 - Jordi Petit (jpetit@cs.upc.edu)
- Sessions:
 - Theory & Problems (Jordi C.)
 - Lab (Jordi P.)
- Languages:
 - English (Theory)
 - Catalan (Problems & Lab)

Material

- Slides, exercises (still under construction):

<https://www.cs.upc.edu/~jordicf/Teaching/AP2>

- Jutge (for lab sessions):

<https://jutge.org>

- Lliçons (by J. Petit and S. Roura):

<https://lliçons.jutge.org>

Evaluation

- Evaluation items:
 - Project (Proj), Parcial Lab (PLab), Final Theory (FTh), Final (FLab).
- Grading:
 - $N_1 = 0.2 \text{ Proj} + 0.2 \text{ PLab} + 0.3 \text{ FTh} + 0.3 \text{ FLab}$
 - $N_2 = 0.2 \text{ Proj} + 0.4 \text{ FTh} + 0.4 \text{ Flab}$
 - $N = \max(N_1, N_2)$

Projects

- Class for Polynomials:
 - Design a class to operate with polynomials (evaluation, addition, multiplication, division, gcd, and Fast Fourier Transform (FFT)).
 - Language: C++.
- *u*oogle (micro-Google):
 - Design a web search application.
 - Optional: implement a simple page rank algorithm.
 - Language: python.

Peer and self assessment

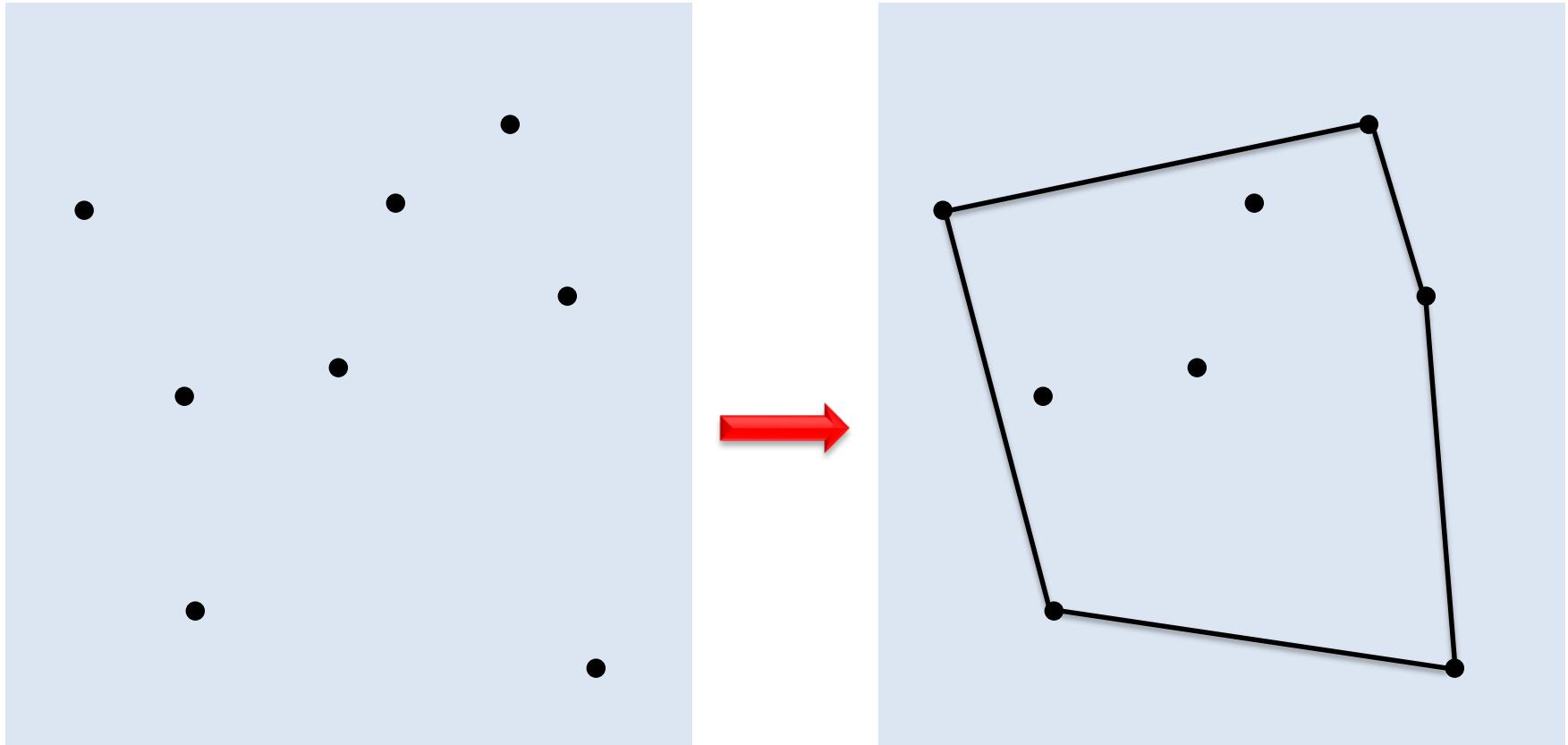
- One of the projects (polynomials) will be evaluated by the students themselves.
- Each project will be evaluated by three students. The grade will be calculated as the average grade given by the students.
- The evaluation will be completely blind.
- Biased evaluations will be detected and penalized.
- Each student will have the right to request the evaluation by the professor (who can upgrade or downgrade the evaluation given by the students).

Objective

Confronting large and difficult problems. How?

- Skills for abstraction and algorithmic reasoning.
- Design and use of complex data structures.
- Techniques for complexity analysis.
- Methodologies for modular programming.
- High-quality code.

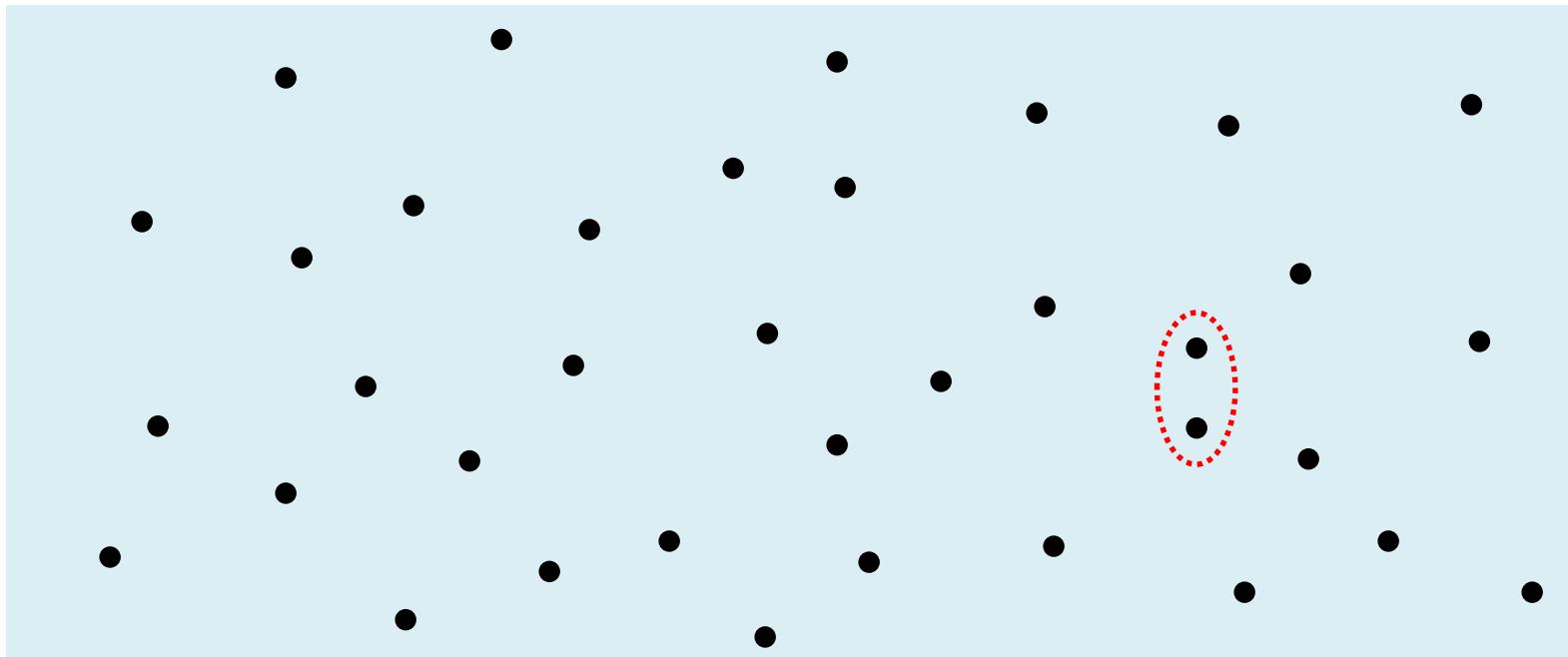
Problems on polygons



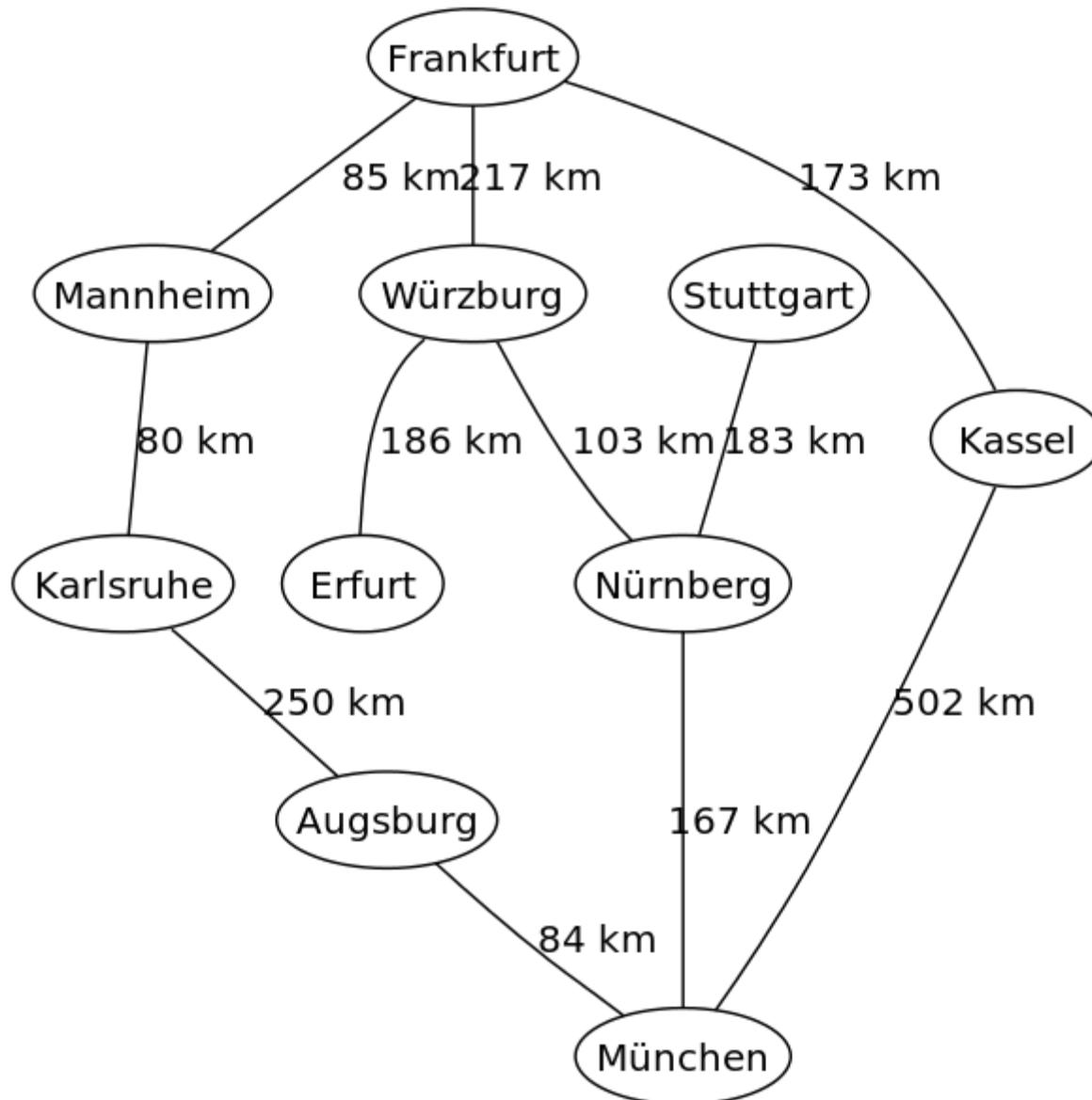
Compute the convex hull of n given points in the plane.

The Closest-Points problem

- **Input:** A list of n points in the plane
 $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$
- **Output:** The pair of closest points
- **Simple approach:** check all pairs $\rightarrow O(n^2)$
- We want an $O(n \log n)$ solution !



Navigation: find the shortest path





The secret: *training, training, training ...*



... up to the finish line

Abstract Data Types (and Object-Oriented Programming)



Jordi Cortadella and Jordi Petit
Department of Computer Science



Wild horses

[Live Science](#) > [Health](#)

Mind's Limit Found: 4 Things at Once

By Clara Moskowitz | April 27, 2008 08:00pm ET

f 468

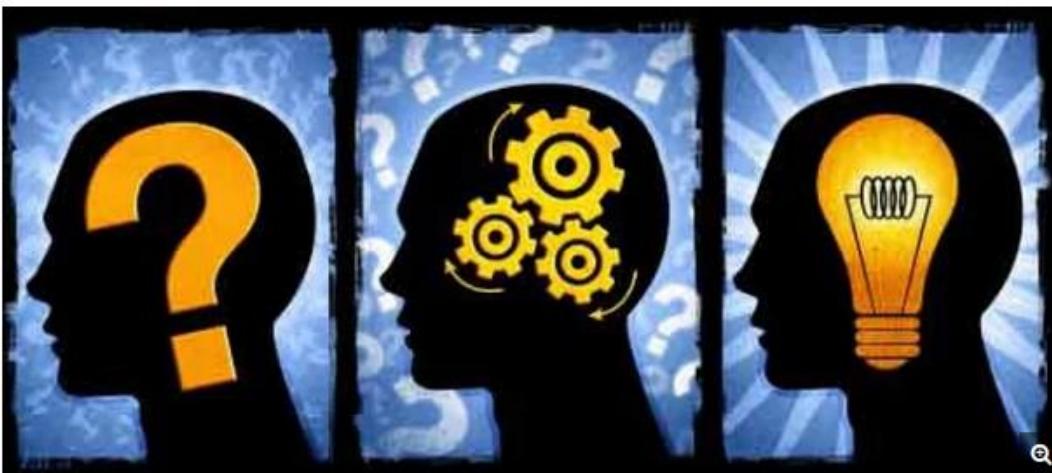
t 17

F

S

u

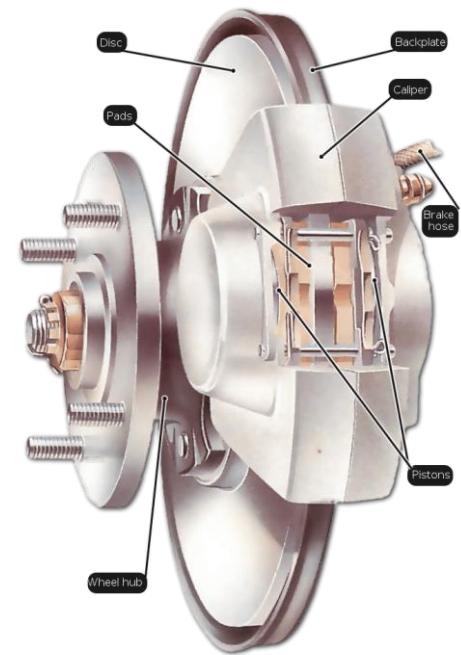
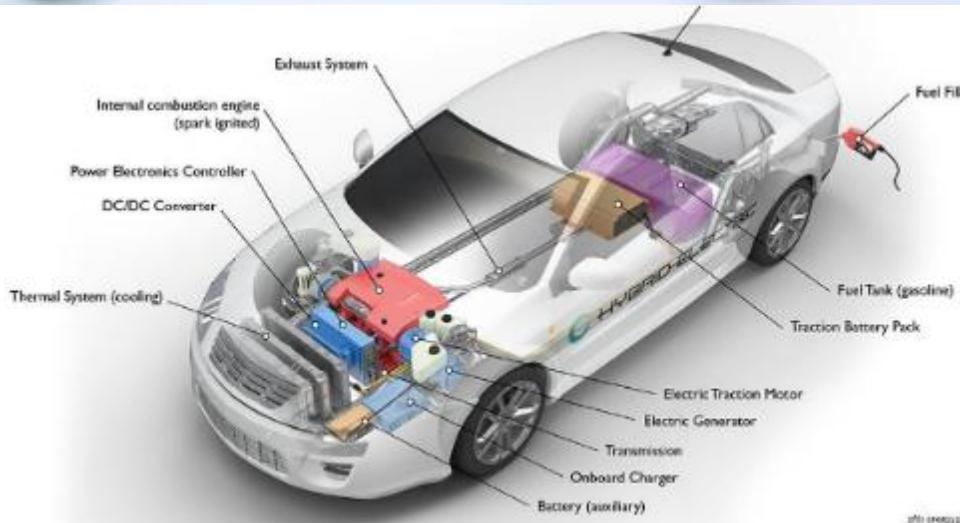
MORE ▾



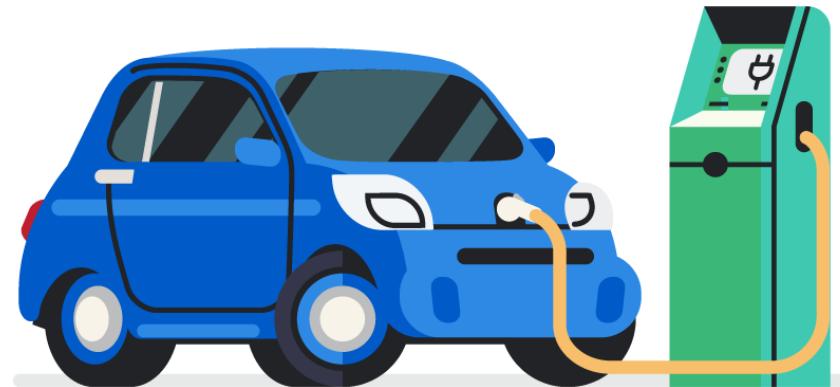
I forget how I wanted to begin this story. That's probably because my mind, just like everyone else's, can only remember a few things at a time. Researchers have often debated the maximum amount of items we can store in our conscious mind, in what's called our working memory, and a new study puts the limit at three or four.

Working memory is a more active version of short-term memory, which refers to the temporary storage of information. Working memory relates to the information we can pay attention to and manipulate.

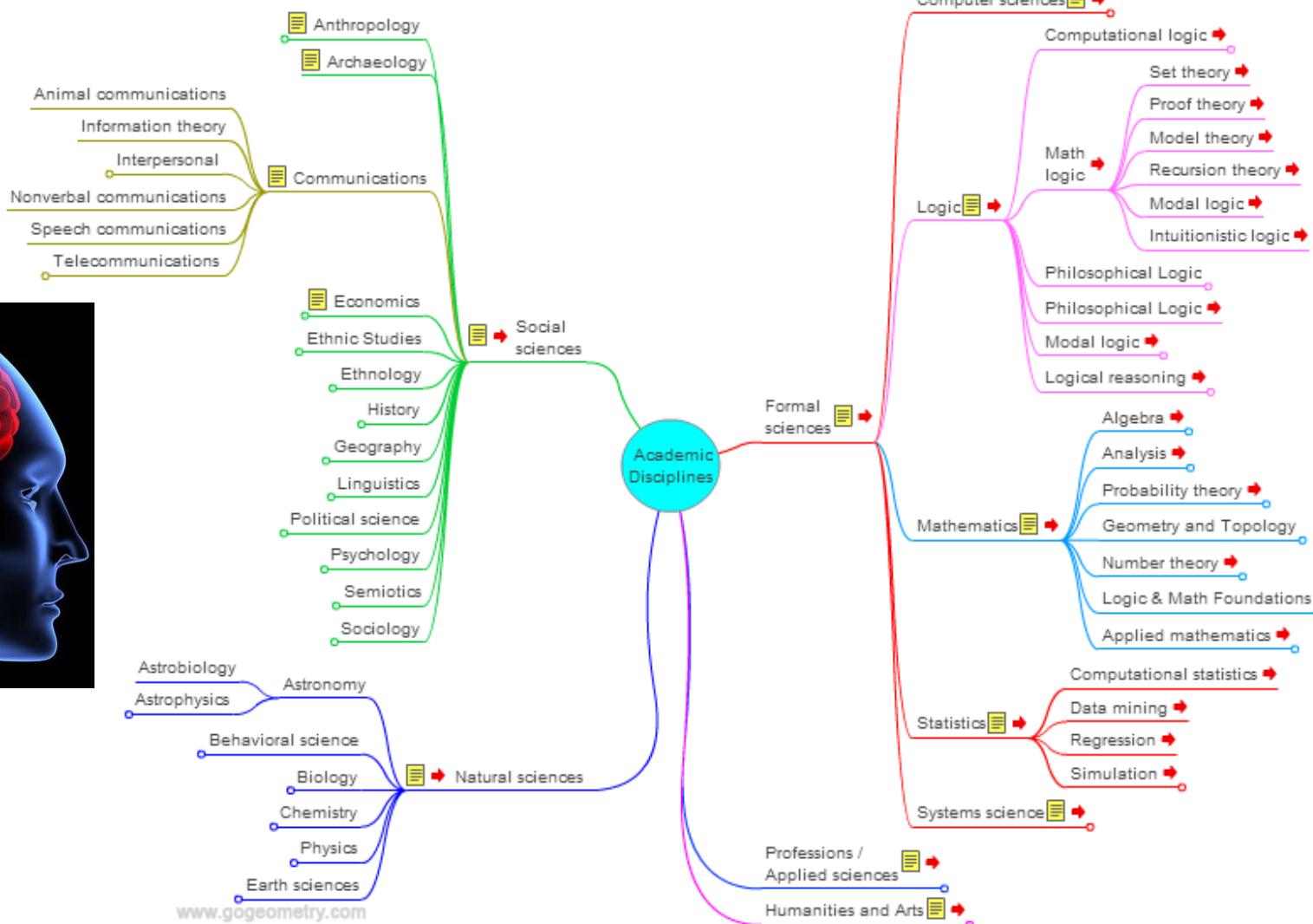
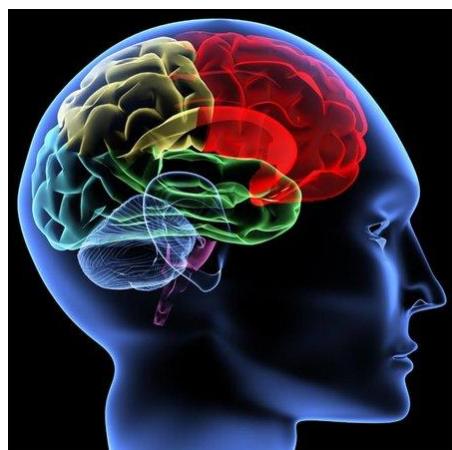
Hiding details: abstractions



Different types of abstractions

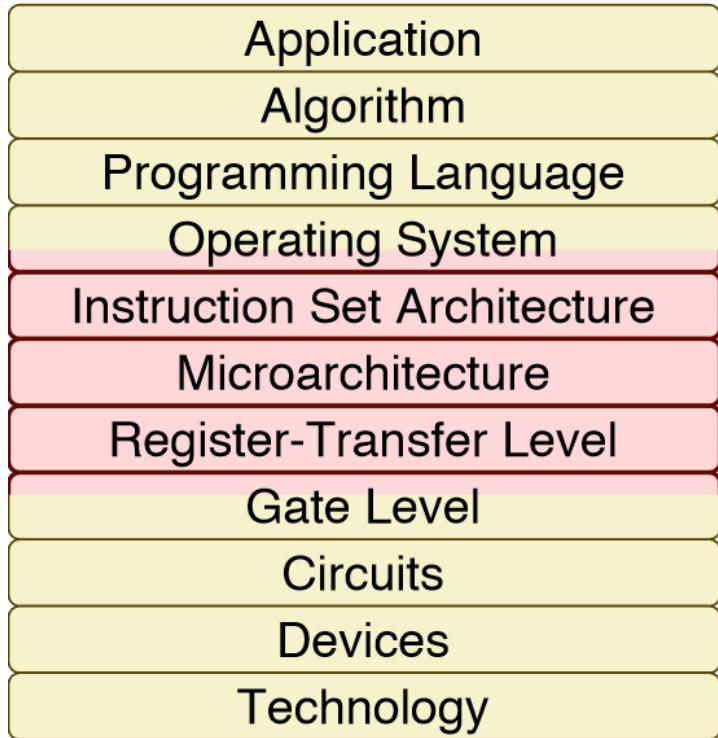


Concept maps are hierarchical: why?



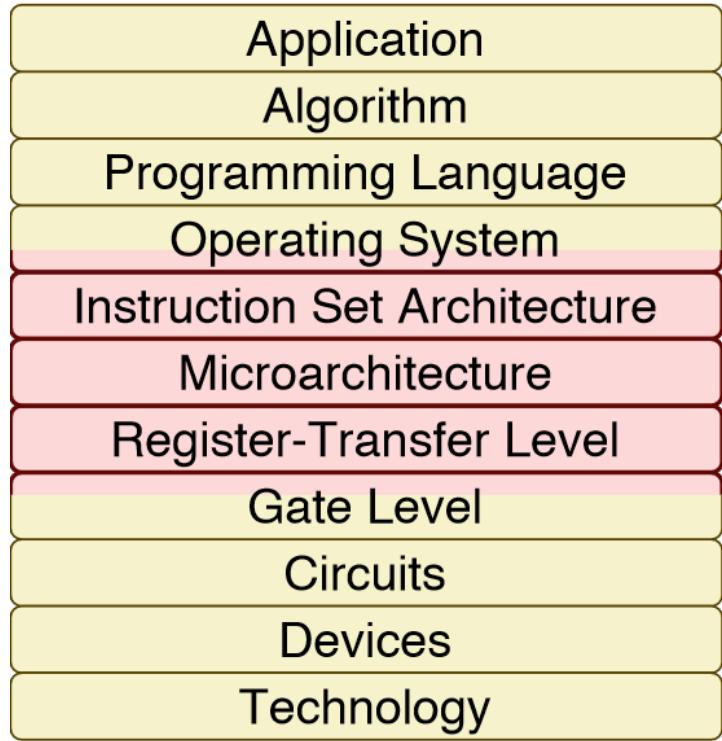
Each level has few items

The computer systems stack

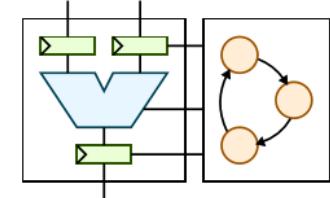


**Image Credit: Christopher Batten,
Cornell University**

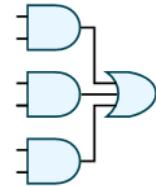
The computer systems stack



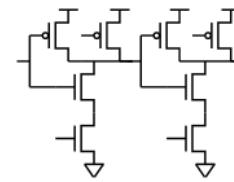
How data flows through system



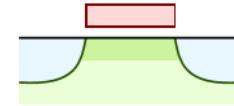
Boolean logic gates and functions



Combining devices to do useful work



Transistors and wires



Silicon process technology

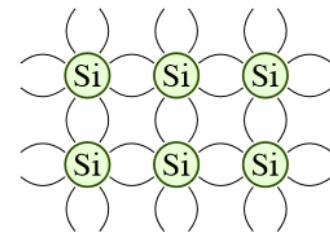
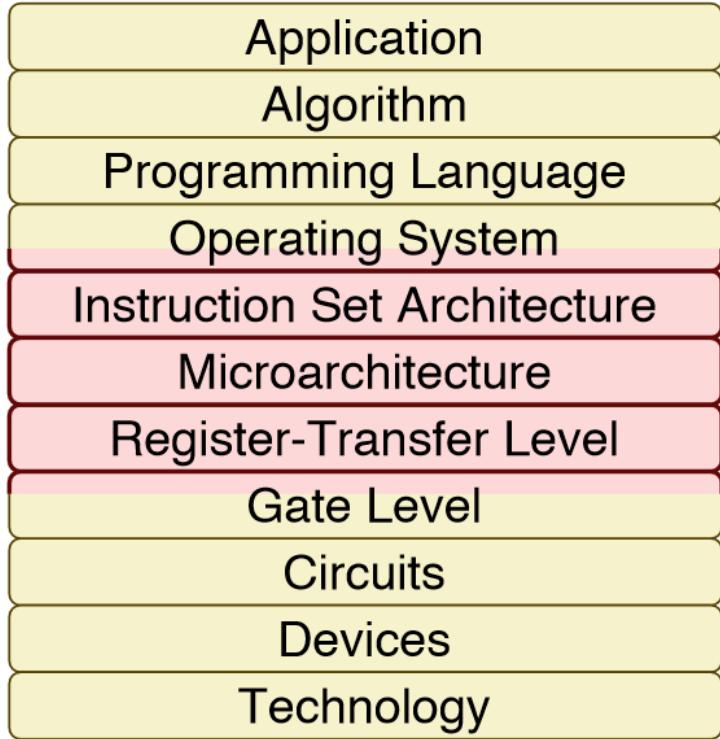


Image Credit: Christopher Batten,
Cornell University

The computer systems stack



Mac OS X, Windows, Linux
Handles low-level hardware management



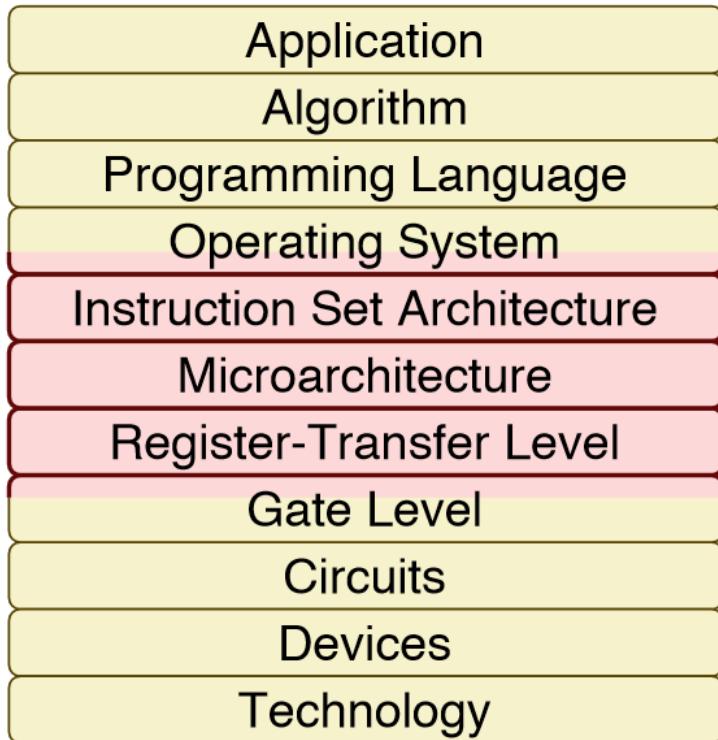
MIPS32 Instruction Set

Instructions that machine executes

```
blez $a2, done
move $a7, $zero
li $t4, 99
move $a4, $a1
move $v1, $zero
li $a3, 99
lw $a5, 0($a4)
addiu $a4, $a4, 4
slt $a6, $a5, $a3
movn $v0, $v1, $a6
addiu $v1, $v1, 1
movn $a3, $a5, $a6
```

Image Credit: Christopher Batten,
Cornell University

The computer systems stack



Sort an array of numbers

2,6,3,8,4,5 -> 2,3,4,5,6,8

Insertion sort algorithm

1. Find minimum number in input array
2. Move minimum number into output array
3. Repeat steps 1 and 2 until finished

C implementation of insertion sort

```
void isort( int b[], int a[], int n ) {  
    for ( int idx, k = 0; k < n; k++ ) {  
        int min = 100;  
        for ( int i = 0; i < n; i++ ) {  
            if ( a[i] < min ) {  
                min = a[i];  
                idx = i;  
            }  
        }  
        b[k] = min;  
        a[idx] = 100;  
    }  
}
```

Image Credit: Christopher Batten,
Cornell University

Our challenge

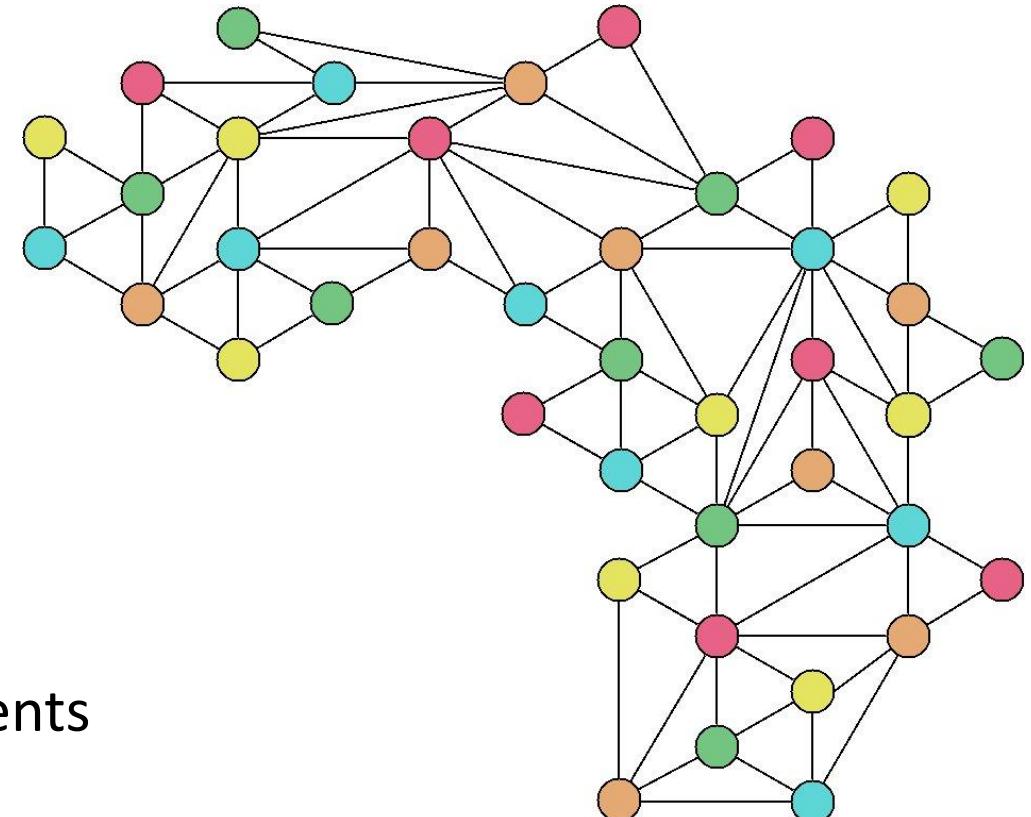
- We need to design large systems.
- We need to reason about complex algorithms.
- Our working memory can only manipulate 4 things at once.
- We need to interact with computers using programming languages.
- Solution: abstraction
 - Abstract reasoning.
 - Programming languages that support abstraction.
- We already use a certain level of abstraction: functions. But it is not sufficient. We need much more.

Data types

- Programming languages have a set of primitive data types (e.g., int, bool, double, char, ...).
- Each data type has a set of associated operations:
 - We can add two integers.
 - We can concatenate two strings.
 - We can divide two doubles.
 - But we cannot divide two strings!
- Programmers can add new operations to the primitive data types:
 - gcd(a,b), match(string1, string2), ...
- The programming languages provide primitives to group data items and create structured collections of data:
 - C++: array, struct.
 - python: list, tuple, dictionary.

Abstract Data Types (ADTs)

A set of objects and a set of operations to manipulate them



Operations:

- Number of vertices
- Number of edges
- Shortest path
- Connected components

Data type: Graph

Abstract Data Types (ADTs)

A set of objects and a set of operations to manipulate them:

$$P(x) = x^3 - 4x^2 + 5$$

Data type: Polynomial

Operations:

- $P + Q$
- $P \times Q$
- P/Q
- $\text{gcd}(P, Q)$
- $P(x)$
- $\text{degree}(P)$

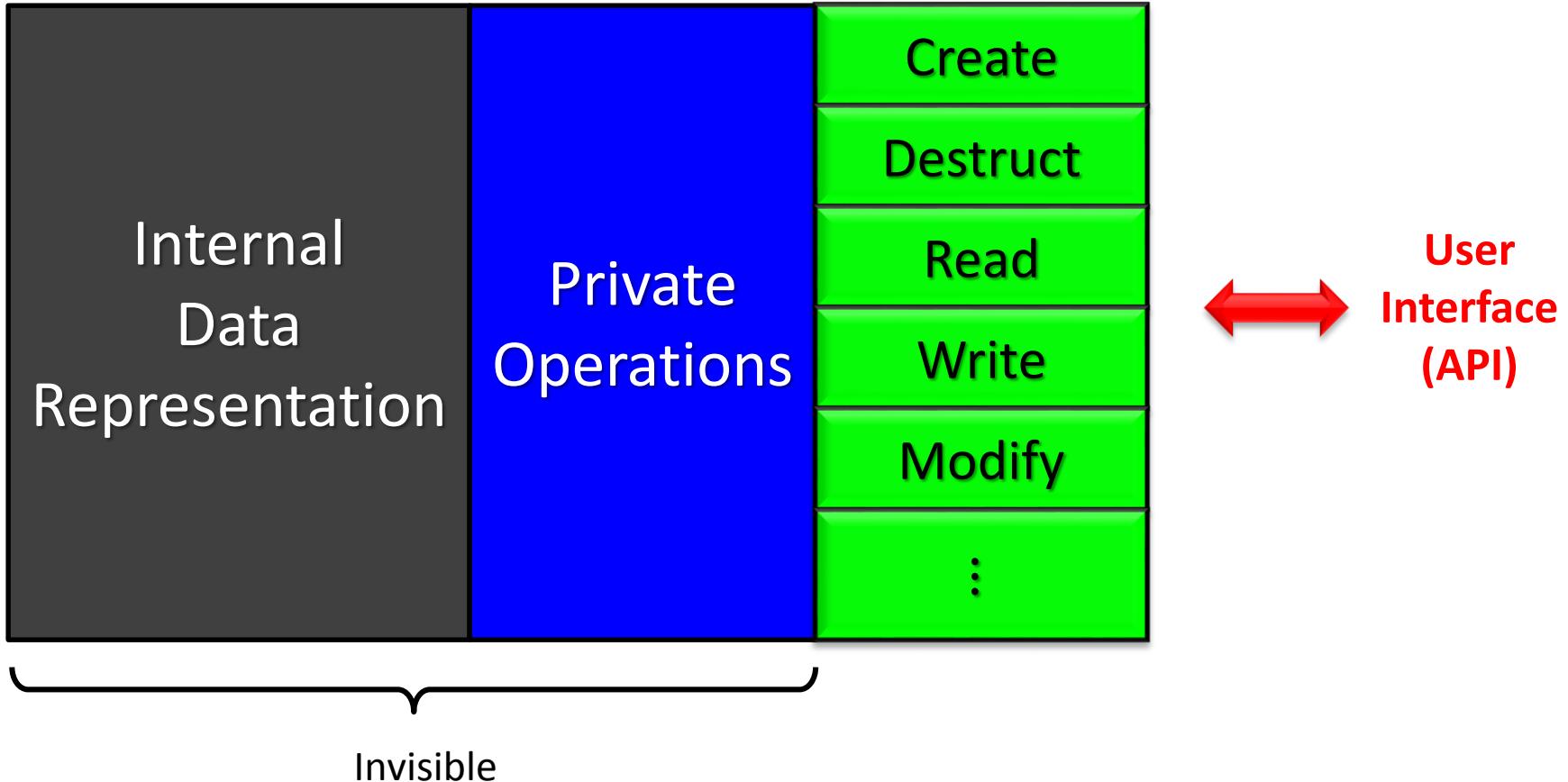
Abstract Data Types (ADTs)

- Separate the notions of specification and implementation:
 - Specification: “what does an operation do?”
 - Implementation: “how is it done?”
- Benefits:
 - Simplicity: code is easier to understand
 - Encapsulation: details are hidden
 - Modularity: an ADT can be changed without modifying the programs that use it
 - Reuse: it can be used by other programs

Abstract Data Types (ADTs)

- An ADT has two parts:
 - **Public** or external: abstract view of the data and operations (methods) that the user can use.
 - **Private** or internal: the actual implementation of the data structures and operations.
- Operations:
 - Creation/Destruction
 - Access
 - Modification

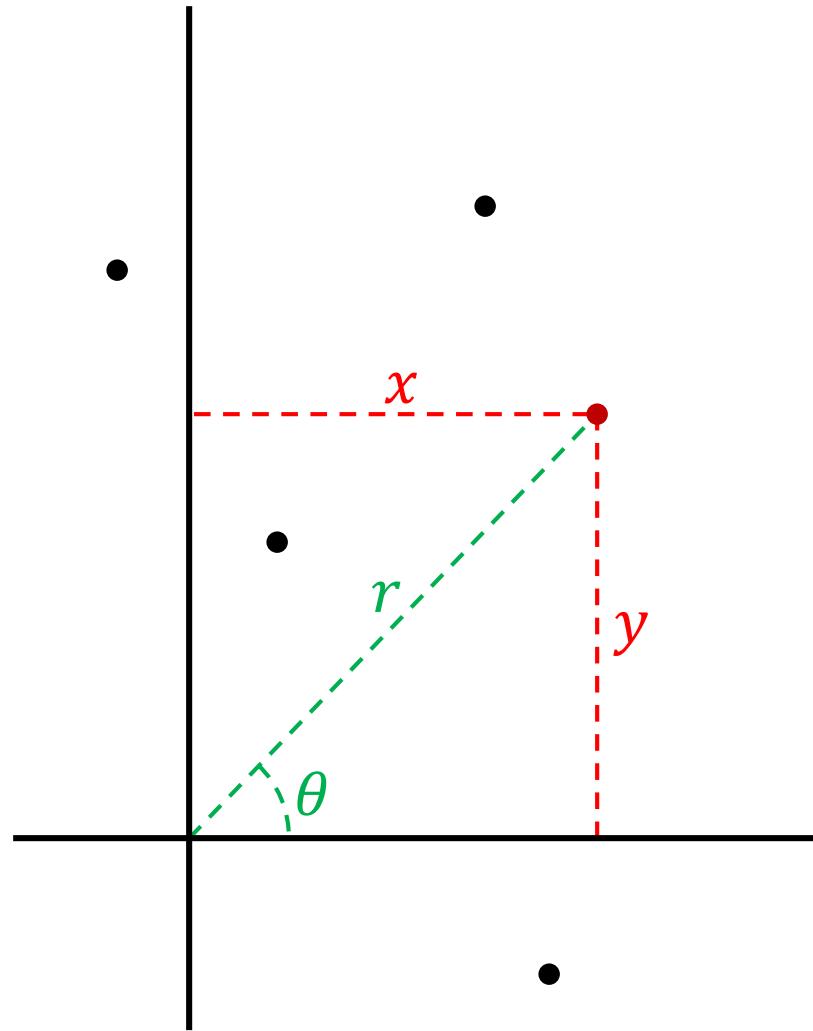
Abstract Data Types (ADTs)



API: Application Programming Interface

Example: a Point

- A point can be represented by two coordinates (x,y) .
- Several operations can be envisioned:
 - Get the x and y coordinates.
 - Calculate distance between two points.
 - Calculate polar coordinates.
 - Move the point by $(\Delta x, \Delta y)$.



Example: a Point

```
// Things that we can do with points

Point p1(5.0, -3.2); // Create a point (a variable)
Point p2(2.8, 0);    // Create another point

// We now calculate the distance between p1 and p2
double dist12 = p1.distance(p2);

// Distance to the origin
double r = p1.radius();

// Create another point by adding coordinates
Point p3 = p1 + p2;

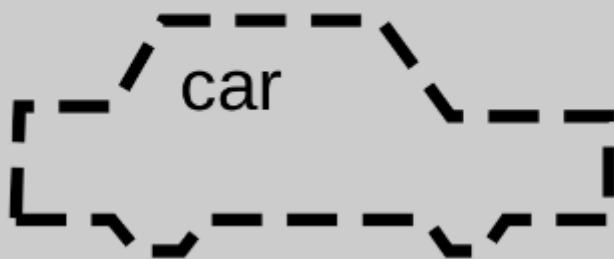
// We get the coordinates of the new point
double x = p3.getX(); // x = 7.8
double y = p3.getY(); // y = -3.2
```

ADTs and Object-Oriented Programming

- OOP is a programming paradigm: a program is a set of objects that interact with each other.
- An object has:
 - fields (or attributes) that contain data
 - functions (or methods) that contain code
- Objects (variables) are instances of classes (types).
A class is a template for all objects of a certain type.
- In OOP, a class is the natural way of implementing an ADT.

Classes and Objects

class



objects



Let us design the new type for Point

```
// The declaration of the class Point
class Point {

public:
    // Constructor
    Point(double x_coord, double y_coord);

    // Gets the x coordinate
    double getX() const;

    // Gets the y coordinate
    double getY() const;

    // Returns the distance to point p
    double distance(const Point& p) const;

    // Returns the radius (distance to the origin)
    double radius() const;

    // Returns the angle of the polar coordinate
    double angle() const;

    // Creates a new point by adding the coordinates of two points
    Point operator + (const Point& p) const;

private:
    double x, y; // Coordinates of the point
};

}
```

Implementation of the class Point

// The constructor: different implementations

```
Point::Point(double x_coord, double y_coord) {  
    x = x_coord; y = y_coord;  
}
```

// or also

```
Point::Point(double x_coord, double y_coord) :  
    x(x_coord), y(y_coord) {}
```

// or also

```
Point::Point(double x, double y) : x(x), y(y) {}
```

All of them are equivalent, but only one of them should be chosen.
We can have different constructors with different *signatures*.

Implementation of the class Point

```
double Point::getX() const {
    return x;
}

double Point::getY() const {
    return y;
}

double Point::distance(const Point& p) const {
    double dx = getX() - p.getX(); // Better getX() than x
    double dy = getY() - p.getY();
    return sqrt(dx*dx + dy*dy);
}

double Point::radius() const {
    return sqrt(getX()*getX() + getY()*getY());
}
```

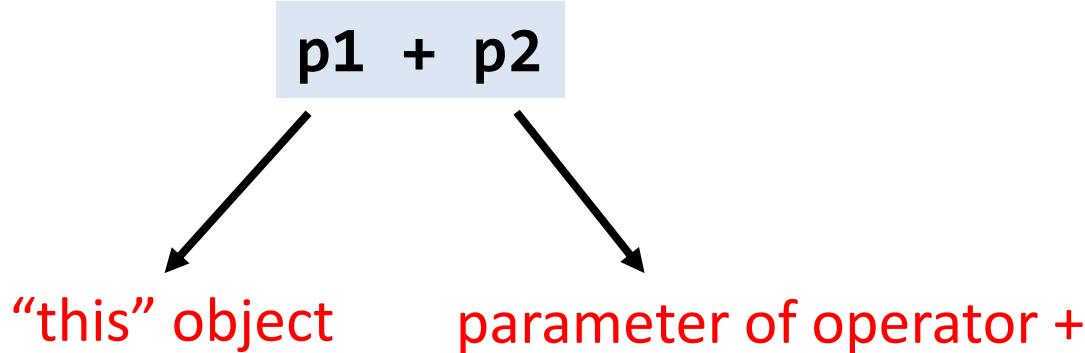
Note: compilers are smart. Small functions are expanded inline.

Implementation of the class Point

```
double Point::angle() const {  
    if (getX() == 0 and getY() == 0) return 0;  
    return atan(getY()/getX());  
}
```

operator
overloading

```
Point Point::operator + (const Point& p) const {  
    return Point(getX() + p.getX(), getY() + p.getY());  
}
```



Public or private?

- What should be public?
 - Only the methods that need to interact with the external world. Hide as much as possible. Make a method public only if necessary.
- What should be private?
 - All the attributes.
 - The internal methods of the class.
- Can we have public attributes?
 - Theoretically yes (C++ and python allow it).
 - Recommendation: never define a public attribute. Why? See the next slides.

Class Point: a new implementation

- Let us assume that we need to represent the point with polar coordinates for efficiency reasons (e.g., we need to use them very often).
- We can modify the private section of the class without modifying the specification of the public methods.
- The private and public methods may need to be rewritten, but not the programs using the public interface.

Class Point: a new implementation

```
// The declaration of the class Point
class Point {

public:
    // Constructor
    Point(double x, double y);

    // Gets the x coordinate
    double getX() const;

    // Gets the y coordinate
    double getY() const;

    // Returns the distance to point p
    double distance(const Point& p) const;

    // Returns the radius (distance to the origin)
    double radius() const;

    // Returns the angle of the polar coordinate
    double angle() const;

    // Creates a new point by adding the coordinates of two points
    Point operator + (const Point& p) const;

private:
    double _radius, _angle; // Polar coordinates
};

}
```

Class Point: a new implementation

```
Point::Point(double x, double y) :  
    _radius(sqrt(x*x + y*y)),  
    _angle(x == 0 and y == 0 ? 0 : atan(y/x))  
{}  
  
double Point::getX() const {  
    return _radius*cos(_angle);  
}  
  
double Point::getY() const {  
    return _radius*sin(_angle);  
}  
  
double Point::distance(const Point& p) const {  
    double dx = getX() - p.getX();  
    double dy = getY() - p.getY();  
    return sqrt(dx*dx + dy*dy);  
}  
  
double Point::radius() const {  
    return _radius;  
}
```

Class Point: a new implementation

```
double Point::angle() const {
    return _angle;
}

// Notice that no changes are required for the + operator
// with regard to the initial implementation of the class
Point Point::operator + (const Point& p) const {
    return Point(getX() + p.getX(), getY() + p.getY());
}
```

Discussion:

- How about having `x` and `y` (or `_radius` and `_angle`) as public attributes?
- Programs using `p.x` and `p.y` would not be valid for the new implementation.
- Programs using `p.getX()` and `p.getY()` would still be valid.

Recommendation (reminder):

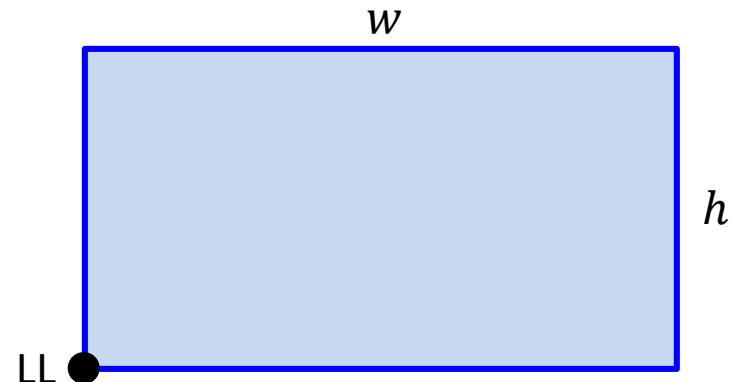
- All attributes should be ***private***.

A new class: Rectangle

- We will only consider orthogonal rectangles (axis-aligned).
- An orthogonal rectangle can be represented in different ways:



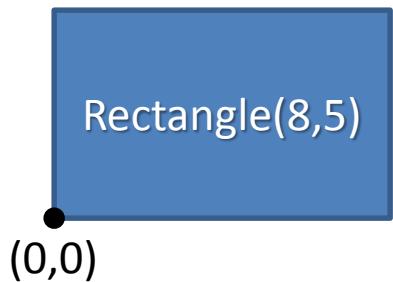
Two points (extremes of diagonal)



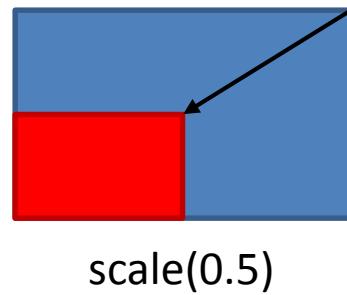
One point, width and height

Rectangle: abstract view

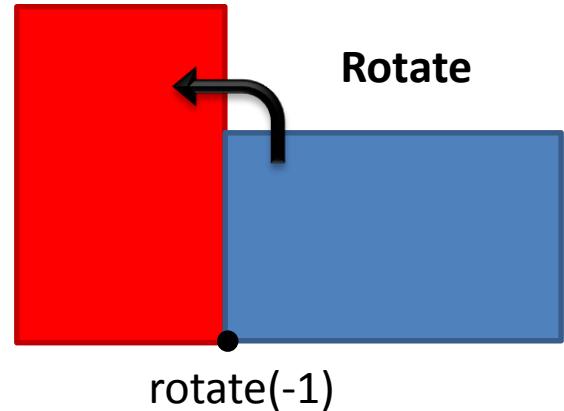
Create



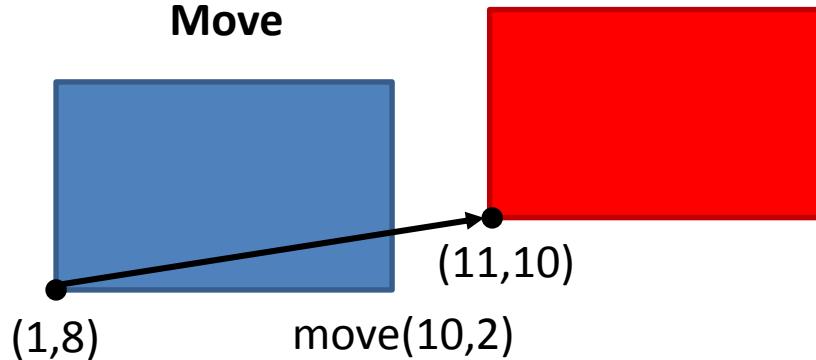
Scale



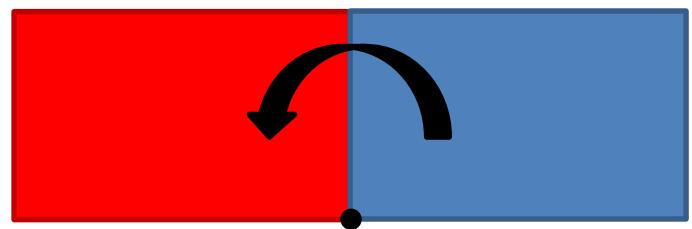
Rotate



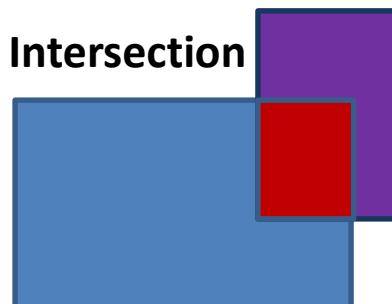
Move



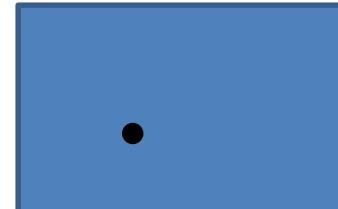
Flip (horizontally/vertically)



Intersection



Point inside?



Rectangle: ADT (incomplete)

```
class Rectangle {  
public:  
    // Constructor (LL at the origin)  
    Rectangle(double width, double height);  
  
    // Returns the area of the rectangle  
    double area() const;  
  
    // Scales the rectangle with a factor s > 0  
    void scale(double s);  
  
    // Returns the intersection with another rectangle  
    Rectangle operator * (const Rectangle& R) const;  
  
    ...  
};
```

Rectangle: using the ADT

```
Rectangle R1(4,5); // Creates a rectangle 4x5
```

```
Rectangle R2(8,4); // Creates a rectangle 8x4
```

```
R1.move(2,3); // Moves the rectangle
```

```
R1.scale(1.2); // Scales the rectangle
```

```
double Area1 = R1.Area(); // Calculates the area
```

```
Rectangle R3 = R1 * R2;
```

```
if (R3.empty()) ...
```

Rectangle: ADT

```
class Rectangle {  
public:  
  
private:  
    Point ll;          // Lower-left corner of the rectangle  
    double w, h;       // width/height of the rect.  
  
    Other private data and methods (if necessary)  
};
```

Rectangle: a rich set of constructors

// LL at the origin

```
Rectangle::Rectangle(double w, double h) :  
    ll(Point(0,0)), w(w), h(h) {}
```

// LL specified at the constructor

```
Rectangle::Rectangle(const Point& p, double w, double h) :  
    ll(p), w(w), h(h) {}
```

// LL and UR specified at the constructor

```
Rectangle::Rectangle(const Point& ll, const Point& ur) :  
    ll(ll), w(ur.getX() - ll.getX()), h(ur.getY() - ll.getY())  
{}
```

// Empty rectangle (using another constructor)

```
Rectangle::Rectangle() : Rectangle(0, 0) {}
```

Rectangle: other public methods

```
Point Rectangle::getLL() const {
    return ll;
}

Point Rectangle::getUR() const {
    return ll + Point(w, h);
}

double Rectangle::getWidth() const {
    return w;
}

double Rectangle::getHeight() const {
    return h;
}
```

```
double Rectangle::area() const {
    return w*h;
}

// Notice: not a const method
void Rectangle::scale(double s) {
    w *= s;
    h *= s;
}

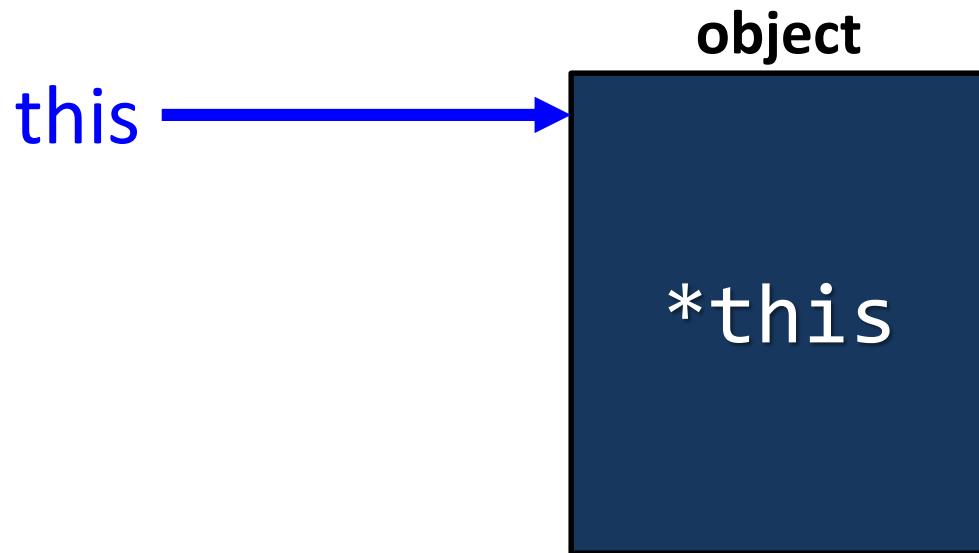
bool Rectangle::empty() const {
    return w <= 0 or h <= 0;
}
```

Rectangle: overloaded operators

```
Rectangle& Rectangle::operator *= (const Rectangle& R) {  
    // Calculate the ll and ur coordinates  
    Point Rll = R.getLL();  
    ll = Point(max(ll.getX(), Rll.getX()),  
               max(ll.getY(), Rll.getY()));  
  
    Point ur = getUR();  
    Point Rur = R.getUR();  
    double urx = min(ur.getX(), Rur.getX());  
    double ury = min(ur.getY(), Rur.getY());  
  
    // Calculate width and height (might be negative → empty)  
    w = urx - ll.getX();  
    h = ury - ll.getY();  
  
    return *this;  
}  
  
// Use *= to implement *  
Rectangle Rectangle::operator * (const Rectangle& R) const {  
    Rectangle result = *this; // Make a copy of itself  
    result *= R;  
    return result;  
}
```

What is `*this`?

- `this` is a pointer (memory reference) to the object (pointers will be explained later)
- `*this` is the object itself



Exercises: implement

```
// Rotate the rectangle 90 degrees clockwise or  
// counterclockwise, depending on the value of the parameter  
void rotate(bool clockwise);  
  
// Flip horizontally or vertically, depending on the value  
// of the parameter.  
void flip(bool horizontally);  
  
// Check whether point p is inside the rectangle  
bool isPointInside(const Point& p) const;
```

Re-implement the class Rectangle using an internal representation with two Points: lower-left (LL) and upper-right (UR) corners.

Let us work with rectangles

```
Rectangle R1(Point(2,3), Point(6,8));
double areaR1 = R1.area(); // areaR1 = 20

Rectangle R2(Point(3,5), 2, 4); // LL=(3,5) UR=(5,9)

// Check whether the point (4,7) is inside the
// intersection of R1 and R2.
bool in = (R1*R2).isPointInside(Point(4,7));
// The object R1*R2 is “destroyed” after the assignment.

R2.rotate(false); // R2 is rotated counterclockwise
R2 *= R1; // Intersection with R1
```

Exercise: draw a picture of R1 and R2 after the execution of the previous code.

Yet another class: Rational

What we would like to do:

```
Rational R1(3);          // R1 = 3
Rational R2(5, 4);      // R2 = 5/4
Rational R3(8, -10);    // R3 = -4/5
```

```
R3 += R1 + Rational(-1, 5); // R3 = 2
```

```
if (R3 >= Rational(2)) {
    R3 = -R1*R2;           // R3 = -15/4
}
```

```
cout << R3.to_str() << endl;
```

The class Rational

```
class Rational {  
private:  
    int num, den; // Invariant: den > 0 and gcd(num,den)=1  
  
    // Simplifies the fraction (used after each operation)  
    void simplify();  
  
public:  
    // Constructor (if some parameter is missing, the default value is taken)  
    Rational(int num = 0, int den = 1);  
  
    // Returns the numerator of the fraction  
    int getNum() const {  
        return num;  
    }  
  
    // Returns the denominator of the fraction  
    int getDen() const {  
        return den;  
    }  
  
    // Returns true if the number is integer and false otherwise.  
    bool isInteger() const {  
        return den == 1;  
    }  
    ...  
};
```

The class Rational

```
class Rational {  
  
public:  
  
...  
// Arithmetic operators  
Rational& operator += (const Rational& r);  
Rational operator + (const Rational& r) const;  
// Similarly for -, * and /  
  
// Unary operator  
Rational operator - () const {  
    return Rational(-getNum(), getDen());  
}  
  
// Equality comparison  
bool operator == (const Rational& r);  
  
// Returns a string representing the rational  
string to_str() const;  
};
```

Rational: constructor and simplify

```
Rational::Rational(int num, int den) : num(num), den(den) {
    simplify();
}

void Rational::simplify() {
    assert(den != 0); // We will discuss assertions later
    if (den < 0) {
        num = -num;
        den = -den;
    }

    // Divide by the gcd of num and den
    int d = gcd(abs(num), den);
    num /= d;
    den /= d;
}
```

Rational: arithmetic and relational operators

```
Rational& Rational::operator += (const Rational& r) {
    num = getNum()*r.getDen() + getDen()*r.getNum();
    den = getDen()*r.getDen();
    simplify();
    return *this;
}

Rational Rational::operator + (const Rational& r) {
    Rational result = *this; // A copy of this object
    result += r;
    return result;
}

bool Rational::operator == (const Rational& r) {
    return getNum() == r.getNum() and getDen() == r.getDen();
}

bool Rational::operator != (const Rational& r) {
    return not operator ==(r);
}

string Rational::to_str() const {
    string s(to_string(getNum()));
    if (not isInteger()) s += "/" + to_string(getDen());
    return s;
}
```

Algorithm Analysis



Jordi Cortadella and Jordi Petit
Department of Computer Science

What do we expect from an algorithm?

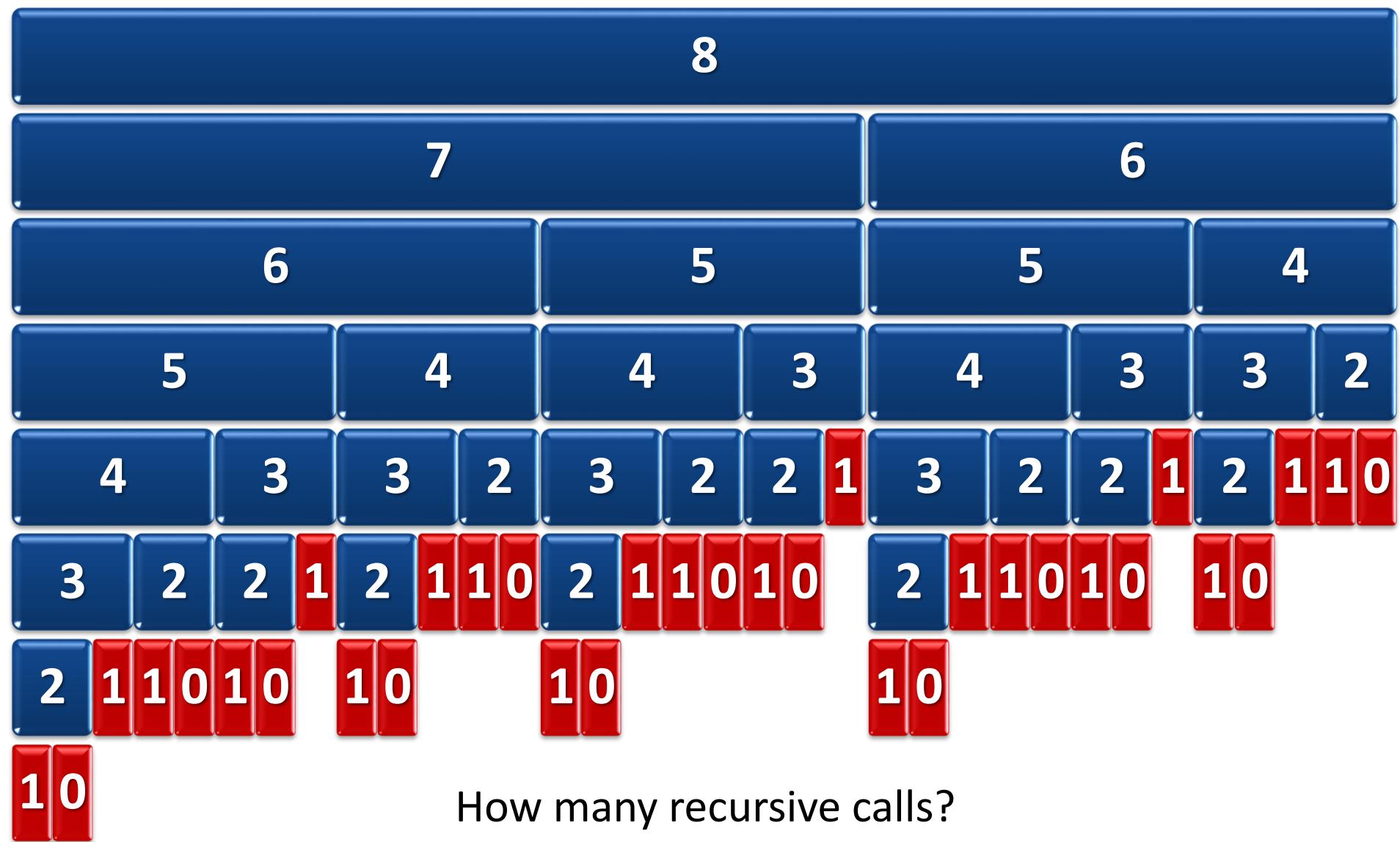
- Correct
- Easy to understand
- Easy to implement
- Efficient:
 - Every algorithm requires a set of resources
 - Memory
 - CPU time
 - Energy

Fibonacci: recursive version

```
// Pre: n ≥ 0
// Returns the Fibonacci number of order n.

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

Fibonacci



How many recursive calls?

Fibonacci: runtime

$$T(0) = 1$$

$$T(1) = 1$$

$$T(n) = T(n - 1) + T(n - 2)$$

Let us assume that $T(n) = a^n$ for some constant a . Then,

$$a^n = a^{n-1} + a^{n-2} \quad \Rightarrow \quad a^2 = a + 1$$

$$a = \frac{1 + \sqrt{5}}{2} = \varphi \approx 1.618 \quad (\text{golden ratio})$$

Therefore, $T(n) \approx 1.6^n$.

If $T(0) = 1$ ns, then $T(100) \approx 2.6 \cdot 10^{20}$ ns > 8000 yrs.

With the age of Universe ($14 \cdot 10^9$ yrs),
we could compute up to `fib(128)`.

Fibonacci numbers: iterative version

```
// Pre: n ≥ 0
// Returns the Fibonacci number of order n.
int fib(int n) { // iterative solution
    int f_i = 0;
    int f_i1 = 1;
    // Inv: f_i is the Fibonacci number of order i.
    //       f_i1 is the Fibonacci number of order i+1.
    for (int i = 0; i < n; ++i) {
        int f = f_i + f_i1;
        f_i = f_i1;
        f_i1 = f;
    }
    return f_i;
}
```

Runtime: n iterations

Fibonacci numbers

Algebraic solution: find matrix A such that

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} \cdot \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$



$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$



$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = A^n \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Fibonacci numbers

$$A^1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

$$A^4 = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix}$$

$$A^8 = \begin{bmatrix} 34 & 21 \\ 21 & 13 \end{bmatrix}$$

$$A^{16} = \begin{bmatrix} 1597 & 987 \\ 987 & 610 \end{bmatrix} \dots A^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Runtime $\approx \log_2 n$ 2x2 matrix multiplications

Algorithm analysis

Given an algorithm that reads inputs from a domain D , we want to define a cost function C :

$$C : D \rightarrow \mathbb{R}^+$$

$$x \mapsto C(x)$$

where $C(x)$ represents the cost of using some resource (CPU time, memory, energy, ...).

Analyzing $C(x)$ for every possible x is impractical.

Algorithm analysis: simplifications

- Analysis based on the size of the input: $|x| = n$
- Only the best/average/worst cases are analyzed:

$$C_{\text{worst}}(n) = \max\{C(x) : x \in D, |x| = n\}$$

$$C_{\text{best}}(n) = \min\{C(x) : x \in D, |x| = n\}$$

$$C_{\text{avg}}(n) = \sum_{x \in D, |x|=n} p(x) \cdot C(x)$$

$p(x)$: probability of selecting input x
among all the inputs of size n .

Algorithm analysis

- Properties:

$$\forall n \geq 0 : C_{\text{best}}(n) \leq C_{\text{avg}}(n) \leq C_{\text{worst}}(n)$$

$$\forall x \in D : C_{\text{best}}(|x|) \leq C(x) \leq C_{\text{worst}}(|x|)$$

- We want a notation that characterizes the cost of algorithms independently from the technology (CPU speed, programming language, efficiency of the compiler, etc.).
- Runtime is usually the most important resource to analyze.

Asymptotic notation

Let us consider all functions $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$

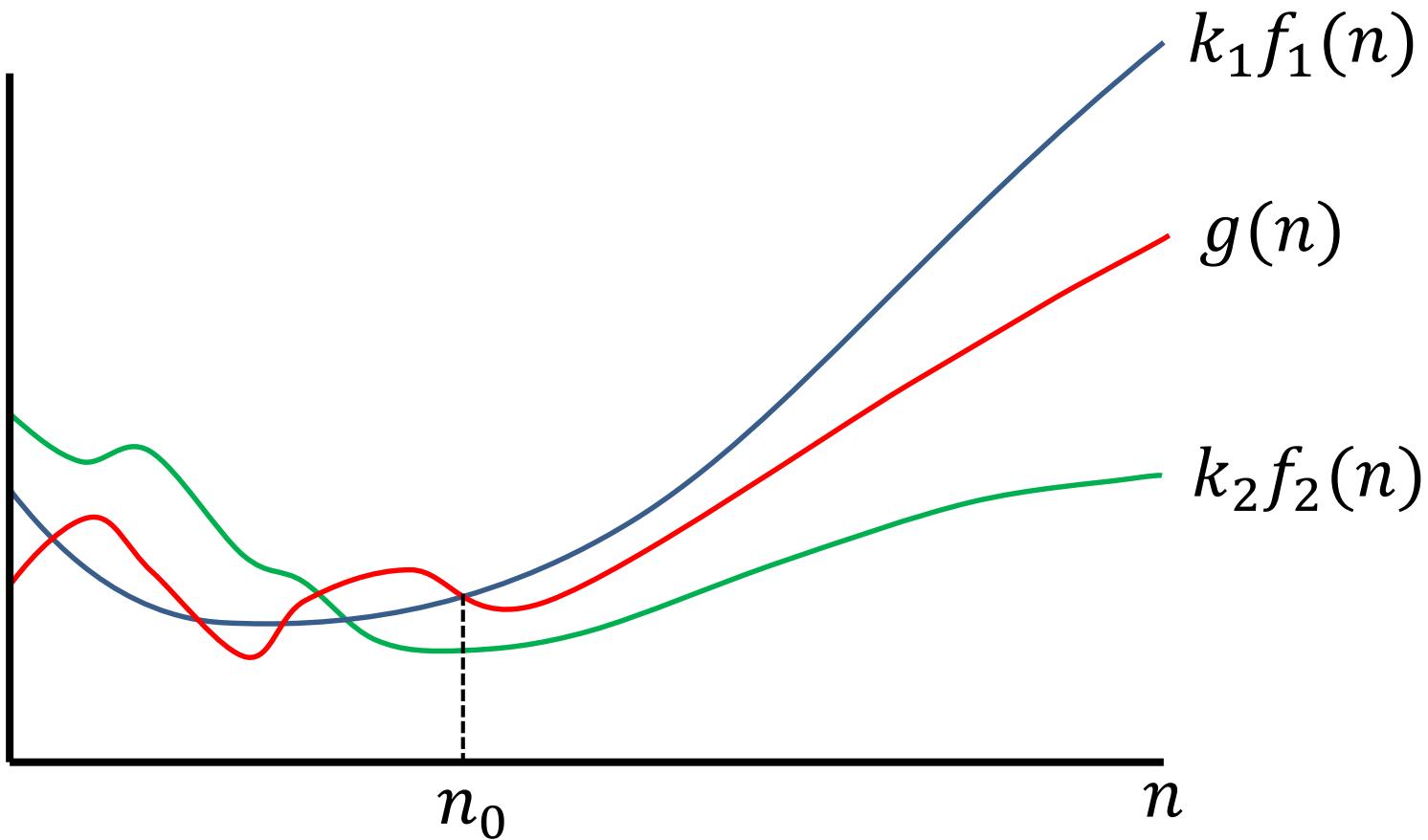
Definitions:

$$O(f(n)) = \{g(n) : \exists k > 0, \exists n_0, \forall n \geq n_0 : g(n) \leq k \cdot f(n)\}$$

$$\Omega(f(n)) = \{g(n) : \exists k > 0, \exists n_0, \forall n \geq n_0 : g(n) \geq k \cdot f(n)\}$$

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

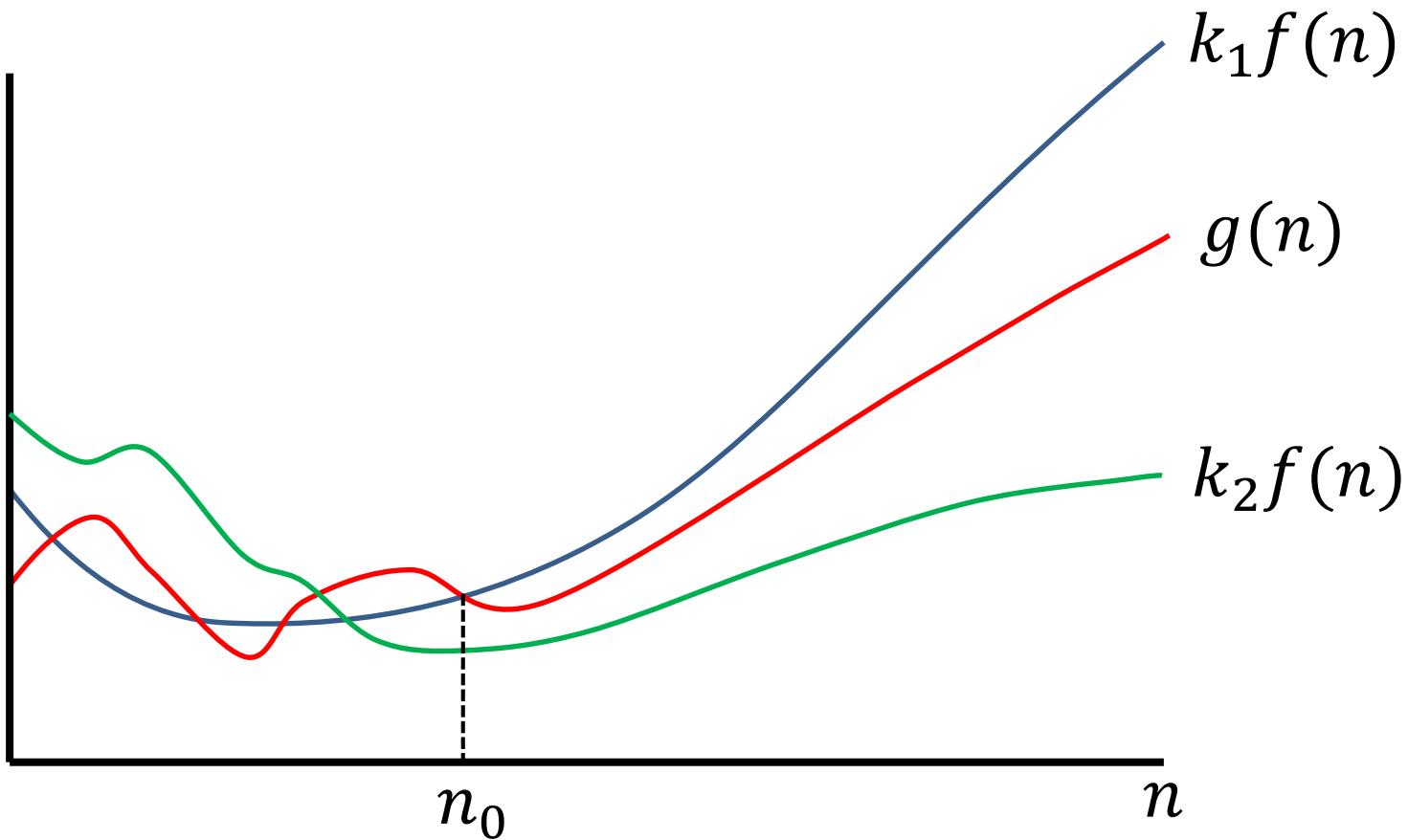
Asymptotic notation



$$g(n) \in O(f_1(n))$$

$$g(n) \in \Omega(f_2(n))$$

Asymptotic notation



$$g(n) \in \Theta(f(n))$$

Examples for Big-O and Big-Ω

$$13n^3 - 4n + 8 \in O(n^3)$$

$$2n - 5 \in O(n)$$

$$n^2 \notin O(n)$$

$$2^n \in O(n!)$$

$$3^n \notin O(2^n)$$

$$3 \log_2 n \in O(\log n)$$

$$3n \log_2 n \in O(n^2)$$

$$O(n^2) \subseteq O(n^3)$$

$$13n^3 - 4n + 8 \in \Omega(n^3)$$

$$n^2 \in \Omega(n)$$

$$n^2 \notin \Omega(n^3)$$

$$n! \in \Omega(2^n)$$

$$3^n \in \Omega(2^n)$$

$$3 \log_2 n \in \Omega(\log n)$$

$$n \log_2 n \in \Omega(n)$$

$$O(n^3) \subseteq \Omega(n^2)$$

Complexity ranking

Function	Common name
$n!$	factorial
2^n	exponential
n^d , $d > 3$	polynomial
n^3	cubic
n^2	quadratic
$n\sqrt{n}$	
$n \log n$	quasi-linear
n	linear
\sqrt{n}	root - n
$\log n$	logarithmic
1	constant

The limit rule

Let us assume that L exists (may be ∞) such that:

$$L = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

$$\begin{cases} \text{if } L = 0 & \text{then } f \in O(g) \\ \text{if } 0 < L < \infty & \text{then } f \in \Theta(g) \\ \text{if } L = \infty & \text{then } f \in \Omega(g) \end{cases}$$

Note: If both limits are ∞ or 0 , use L'Hôpital rule:

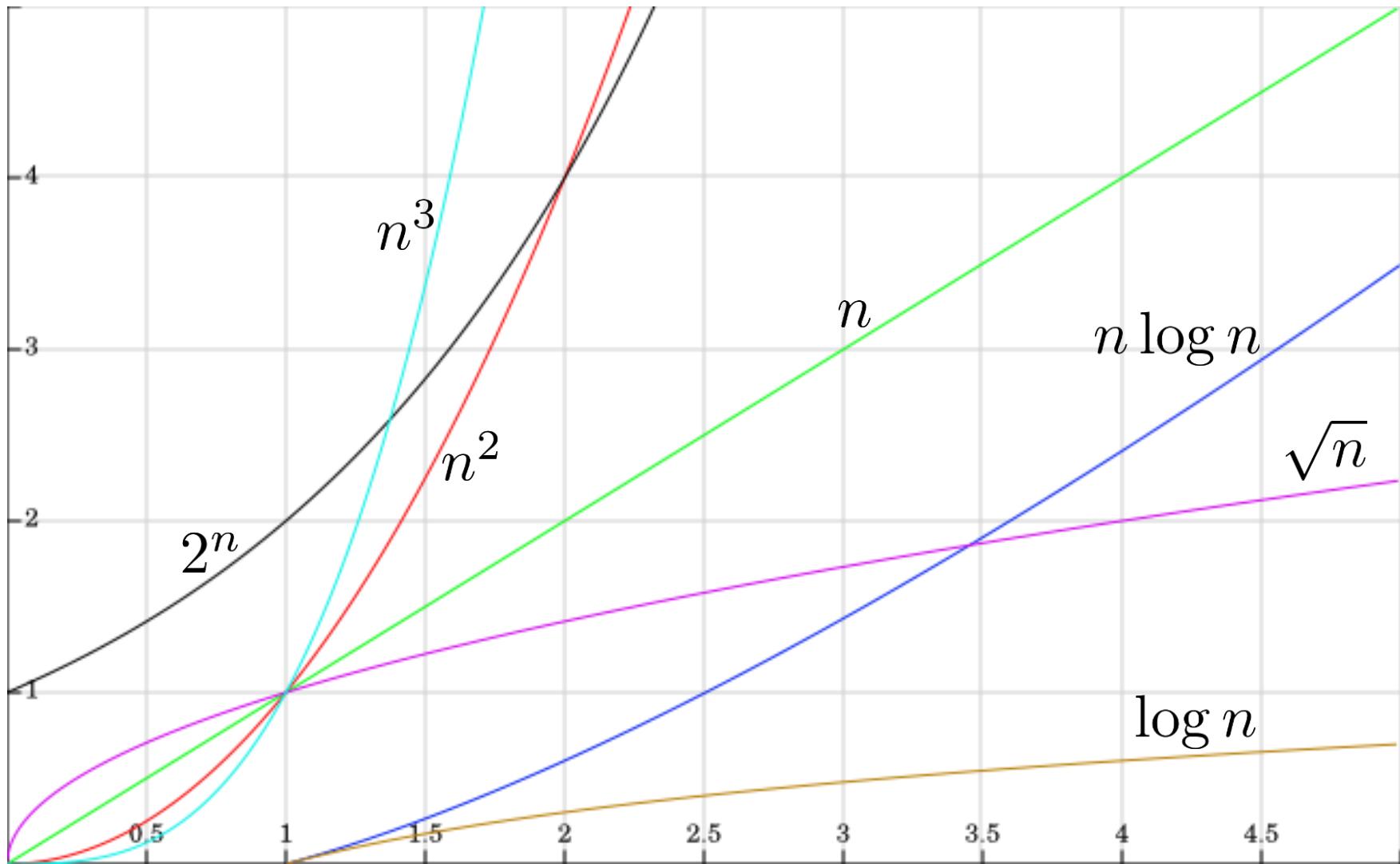
$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}$$

Properties

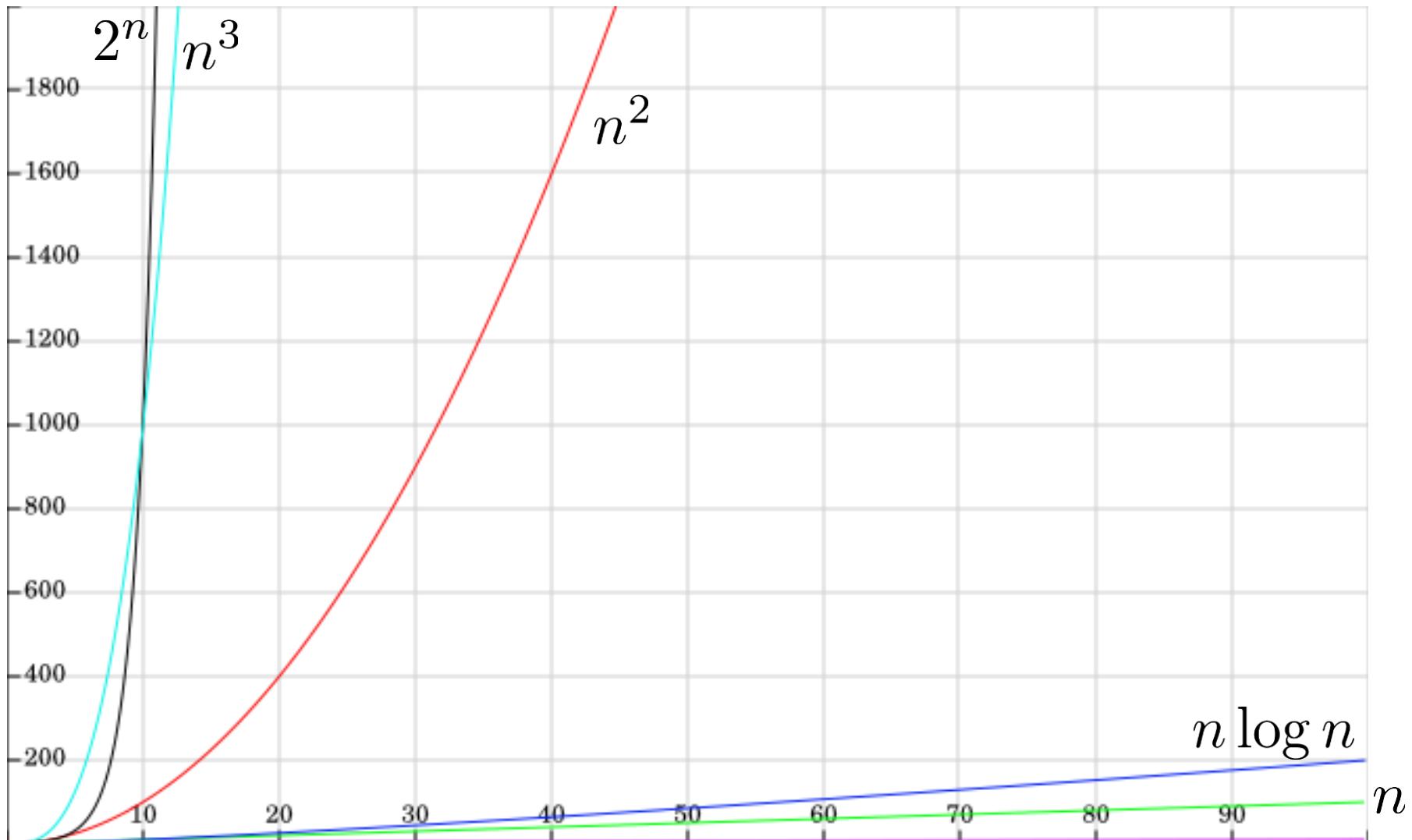
All rules (except the last one) also apply for Ω and Θ :

- $f \in O(f)$
- $\forall c > 0, O(f) = O(c \cdot f)$
- $f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$
- $f_1 \in O(g_1) \wedge f_2 \in O(g_2)$
 $\Rightarrow f_1 + f_2 \in O(g_1 + g_2) = O(\max \{g_1, g_2\})$
- $f \in O(g) \Rightarrow f + g \in O(g)$
- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 \cdot f_2 \in O(g_1 \cdot g_2)$
- $f \in O(g) \Leftrightarrow g \in \Omega(f)$

Asymptotic complexity (small values)



Asymptotic complexity (larger values)



Execution time: example

Let us consider that every operation can be executed in 1 ns (10^{-9} s).

Function	Time		
	$n = 1,000$	$n = 10,000$	$n = 100,000$
$\log_2 n$	10 ns	13.3 ns	16.6 ns
\sqrt{n}	31.6 ns	100 ns	316 ns
n	1 μ s	10 μ s	100 μ s
$n \log_2 n$	10 μ s	133 μ s	1.7 ms
n^2	1 ms	100 ms	10 s
n^3	1 s	16.7 min	11.6 days
n^4	16.7 min	116 days	3171 yr
2^n	$3.4 \cdot 10^{284}$ yr	$6.3 \cdot 10^{2993}$ yr	$3.2 \cdot 10^{30086}$ yr

How about “big data”?

Source: Jon Kleinberg and Éva Tardos, Algorithm Design, Addison Wesley 2006.

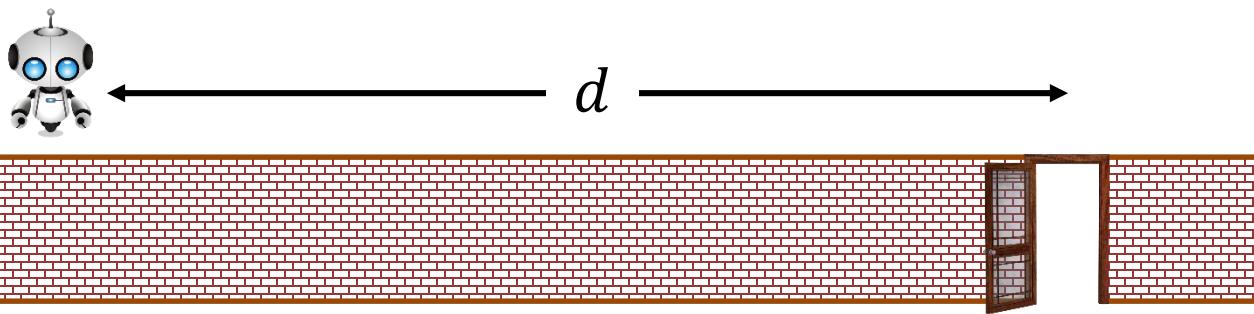
Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long



This is often the practical limit for big data

The robot and the door in an infinite wall

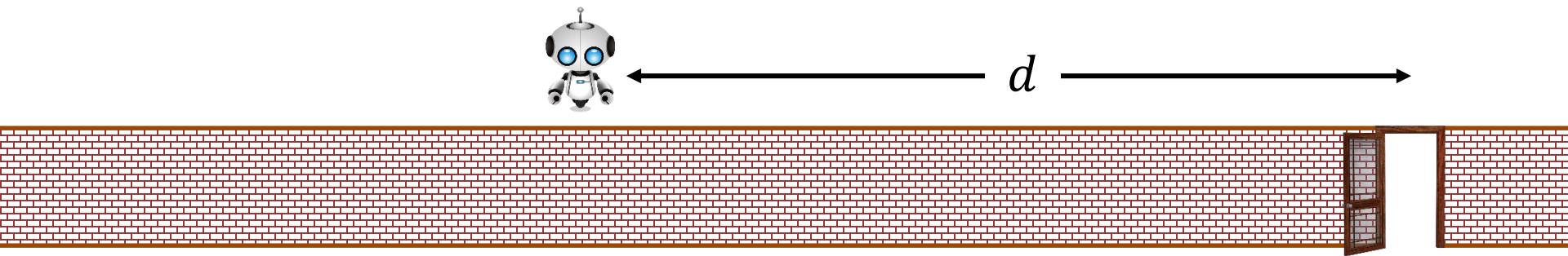


A robot stands in front of a wall that is infinitely long to the right and left side. The wall has a door somewhere and the robot has to find it to reach the other side. Unfortunately, the robot can only see the part of the wall in front of it.

The robot does not know neither how far away the door is nor what direction to take to find it. It can only execute moves to the left or right by a certain number of steps.

Let us assume that the door is at a distance d . How to find the door in a minimum number of steps?

The robot and the door in an infinite wall



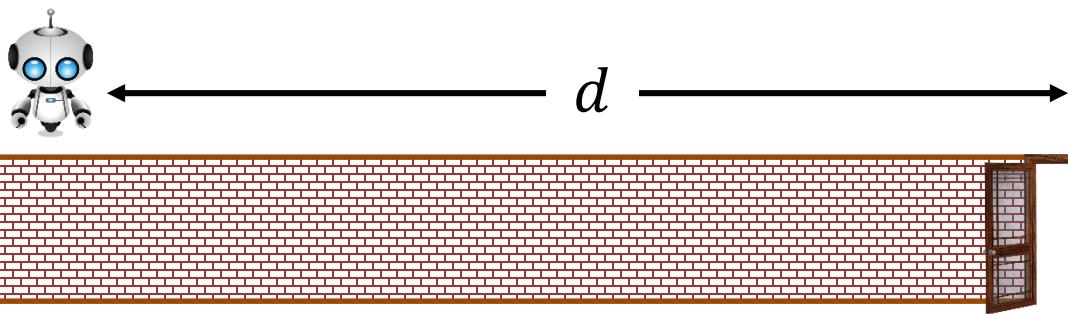
Algorithm 1:

- Pick one direction and move until the door is found.

Complexity:

- If the direction is correct $\rightarrow O(d)$.
- If incorrect \rightarrow the algorithm does not terminate.

The robot and the door in an infinite wall



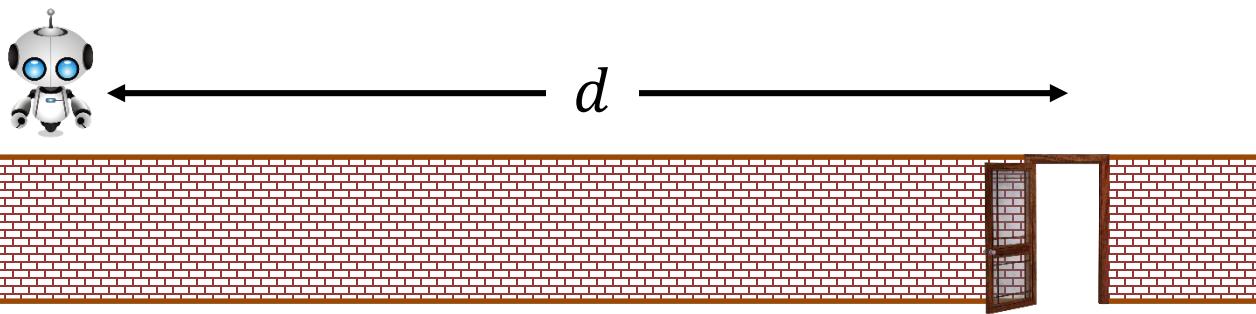
Algorithm 2:

- 1 step to the left,
- 2 steps to the right,
- 3 steps to the left, ...
- ... increasing by one step in the opposite direction.

Complexity:

$$T(d) = \sum_{i=1}^{2d} i = 2d \frac{1 + 2d}{2} = 2d^2 + d = O(d^2)$$

The robot and the door in an infinite wall



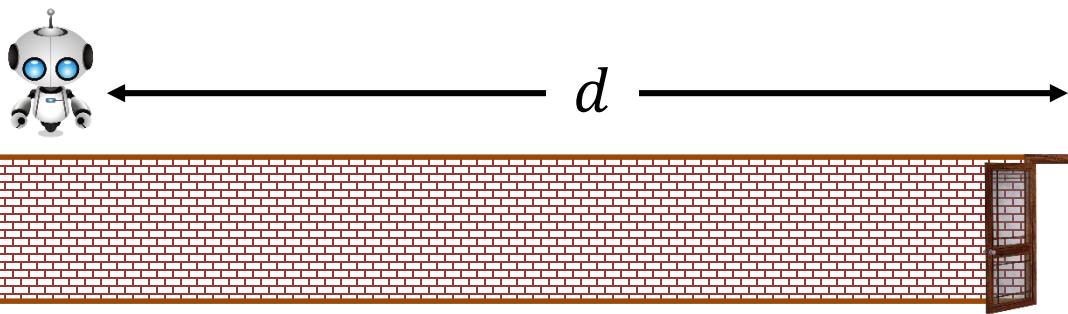
Algorithm 3:

- 1 step to the left and return to origin,
- 2 steps to the right and return to origin,
- 3 steps to the left and return to origin,...
- ... increasing by one step in the opposite direction.

Complexity:

$$T(d) = d + 2 \sum_{i=1}^d i = d + 2 \frac{d(d+1)}{2} = d^2 + 2d = O(d^2)$$

The robot and the door in an infinite wall



Algorithm 4:

- 1 step to the left and return to origin,
- 2 steps to the right and return to origin,
- 4 steps to the left and return to origin,...
- ... doubling the number of steps in the opposite direction.

Complexity (assume that $d = 2^n$):

$$T(d) = d + 2 \sum_{i=0}^n 2^i = d + 2(2^{n+1} - 1) = 5d - 2 = O(d)$$

Runtime analysis rules

- Variable declarations cost no time.
- *Elementary operations* are those that can be executed with a *small number of basic computer steps* (an assignment, a multiplication, a comparison between two numbers, etc.).
- Vector sorting or matrix multiplication are not elementary operations.
- We consider that the cost of elementary operations is $O(1)$.

Runtime analysis rules

- Consecutive statements:
 - If S_1 is $O(f)$ and S_2 is $O(g)$,
then $S_1;S_2$ is $O(\max\{f, g\})$
- Conditional statements:
 - If S_1 is $O(f)$, S_2 is $O(g)$ and B is $O(h)$,
then $\text{if } (B) S_1; \text{else } S_2;$ is $O(\max\{f + h, g + h\})$,
or also $O(\max\{f, g, h\})$.

Runtime analysis rules

- For/While loops:
 - Running time is at most the running time of the statements inside the loop times the number of iterations
- Nested loops:
 - Analyze inside out: running time of the statements inside the loops multiplied by the product of the sizes of the loops

Nested loops: examples

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        DoSomething(); // O(1)      => O(n2)
```

```
for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j)
        DoSomething(); // O(1)      => O(n2)
```

```
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
        for (int k = 0; k < p; ++k)
            DoSomething(); // O(1)      => O(n · m · p)
```

Linear time: $O(n)$

Running time proportional to input size

```
// Compute the maximum of a vector
// with n numbers

int m = a[0];
for (int i = 1; i < a.size(); ++i) {
    if (a[i] > m) m = a[i];
}
```

Linear time: $O(n)$

Other examples:

- Reversing a vector
- Merging two sorted vectors
- Finding the largest null segment of a sorted vector:
a linear-time algorithm exists
(a null segment is a compact sub-vector in which
the sum of all the elements is zero)

Logarithmic time: $O(\log n)$

- Logarithmic time is usually related to divide-and-conquer algorithms
- Examples:
 - Binary search
 - Calculating x^n
 - Calculating the n -th Fibonacci number

Example: recursive x^y

```
// Pre: x ≠ 0, y ≥ 0
// Returns xy

int power(int x, int y) {
    if (y == 0) return 1;
    if (y%2 == 0) return power(x*x, y/2);
    return x*power(x*x, y/2);
}
```

// Assumption: each */% takes O(1)

$$T(x^y) \leq 4 + T((x^2)^{y/2}) \leq 4 + 4 + T((x^4)^{y/4}) \leq \dots$$

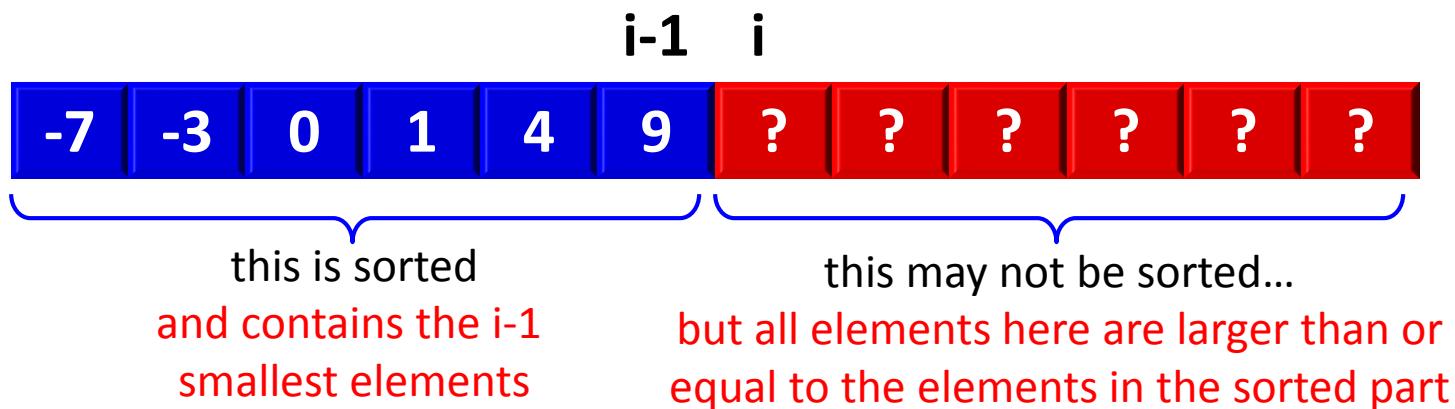
$$T(x^y) \leq \underbrace{4 + 4 + \dots + 4}_{\log_2 y \text{ times}} \implies O(\log y)$$

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

- ***Sorting:*** Merge sort and heap sort can be executed in $O(n \log n)$.
- ***Largest empty interval:*** Given n time-stamps x_1, \dots, x_n on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?
 - $O(n \log n)$ solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Selection Sort

- Selection sort uses this invariant:



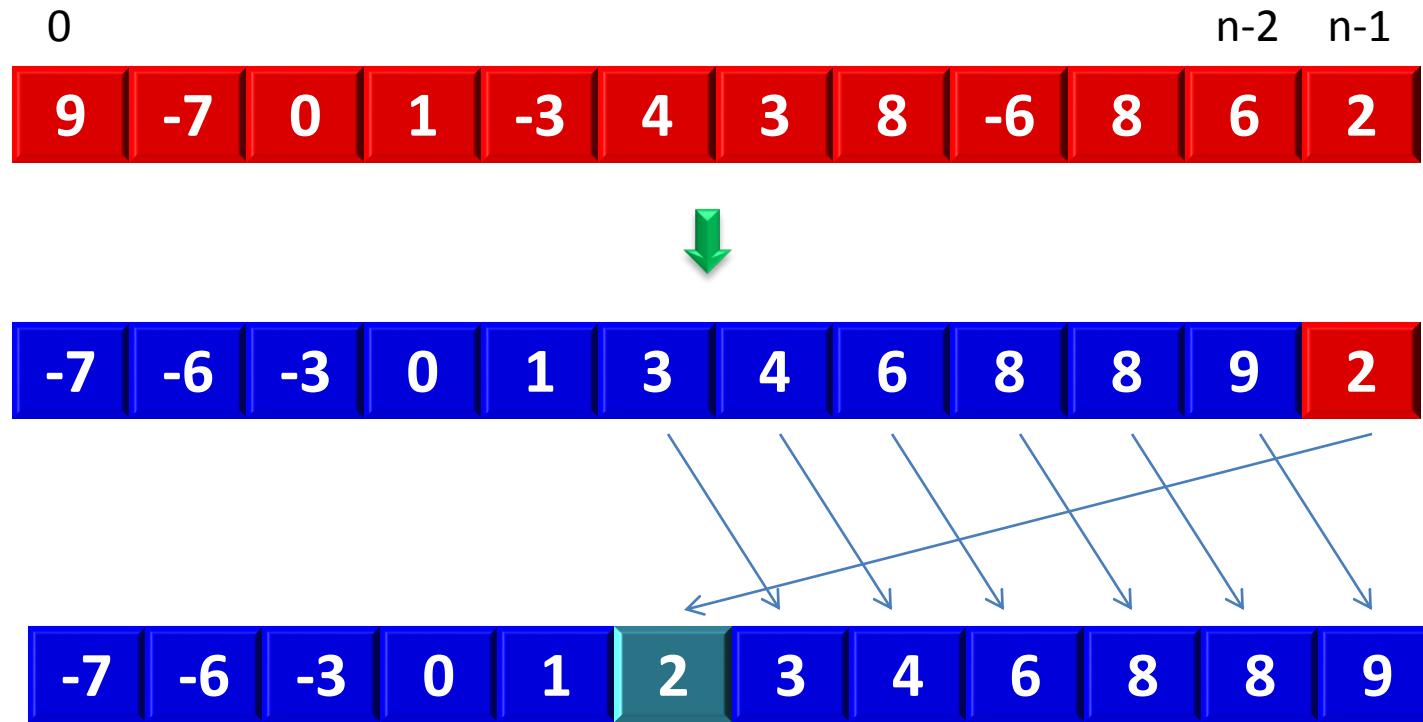
Selection Sort

```
void selection_sort(vector<elem>& v) {
    int last = v.size() - 1;                                // v.size() = n
    for (int i = 0; i < last; ++i) {                         // 0..n-2
        int k = i;
        for (int j = i + 1; j <= last; ++j) { // i+1..n-1
            if (v[j] < v[k]) k = j;
        }
        swap(v[k], v[i]);
    }
}
```

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} O(1) = O(1) \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = O(1) \sum_{i=0}^{n-2} (n - i - 1) \\ &= O(1) \left(\frac{n}{2} (n - 1) \right) = O(1) \cdot O(n^2) = O(n^2) \end{aligned}$$

Insertion Sort

- Let us use inductive reasoning:
 - If we know how to sort arrays of size $n-1$,
 - do we know how to sort arrays of size n ?



Insertion Sort

```
void insertion_sort(vector<elem>& v) {
    for (int i = 1; i < v.size(); ++i) { // n-1 times
        elem x = v[i];
        int j = i;
        while (j > 0 and v[j - 1] > x) { // 0..i times
            v[j] = v[j - 1];
            --j;
        }
        v[j] = x;
    }
}
```

$$\begin{aligned} T(n) &= \Omega(n) \\ T(n) &= O(n^2) \end{aligned}$$

$$T_{\text{worst}}(n) = \sum_{i=1}^{n-1} i \cdot O(1) = O(n^2)$$

⇒ sorted in reverse order

$$T_{\text{best}}(n) = \sum_{i=1}^{n-1} O(1) = O(n)$$

⇒ already sorted

The Maximum Subsequence Sum Problem

- Given (possibly negative) integers A_1, A_2, \dots, A_n , find the maximum value of $\sum_{k=i}^j A_k$. (the max subsequence sum is 0 if all integers are negative).
- Example:
 - Input: -2, 11, -4, 13, -5, -2
 - Answer: 20 (subsequence 11, -4, 13)

(extracted from M.A. Weiss, Data Structures and Algorithms in C++, Pearson, 2014, 4th edition)

The Maximum Subsequence Sum Problem

```
int maxSubSum(const vector<int>& a) {  
    int maxSum = 0;  
    // try all possible subsequences  
    for (int i = 0; i < a.size(); ++i)  
        for (int j = i; j < a.size(); ++j) {  
            int thisSum = 0;  
            for (int k = i; k <= j; ++k) thisSum += a[k];  
            if (thisSum > maxSum) maxSum = thisSum;  
        }  
    return maxSum;  
}
```

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^j 1$$

The Maximum Subsequence Sum Problem

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \sum_{k=i}^j 1 \\ &= \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1) \\ &= \sum_{i=0}^{n-1} \frac{(n - i + 1)(n - i)}{2} = \dots \\ &= \frac{n^3 + 3n^2 + 2n}{6} = O(n^3) \end{aligned}$$

The Maximum Subsequence Sum Problem

```
int maxSubSum(const vector<int>& a) {  
    int maxSum = 0;  
    // try all possible subsequences  
    for (int i = 0; i < a.size(); ++i) {  
        int thisSum = 0;  
        for (int j = i; j < a.size(); ++j) {  
            thisSum += a[j]; // reuse computation  
            if (thisSum > maxSum) maxSum = thisSum;  
        }  
    }  
    return maxSum;  
}
```

$$T(n) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} 1 = O(n^2)$$

Max Subsequence Sum: Divide&Conquer

First half	Second half						
4	-3	5	-2	-1	2	6	-2

The max sum can be in one of three places:

- 1st half
- 2nd half
- Spanning both halves and crossing the middle

In the 3rd case, two max subsequences must be found starting from the center of the vector (one to the left and the other to the right)

Max Subsequence Sum: Divide&Conquer

```
int maxSumRec(const vector<int>& a,
              int left, int right) {
    // base cases
    if (left == right)
        if (a[left] > 0) return a[left];
        else return 0;

    // Recursive cases: left and right halves
    int center = (left + right)/2;
    int maxLeft = maxSumRec(a, left, center);
    int maxRight = maxSumRec(a, center + 1, right);
```

:

Max Subsequence Sum: Divide&Conquer

```
        :  
int maxRCenter = 0, rightSum = 0;  
for (int i = center; i >= left; --i) {  
    rightSum += a[i];  
    if (rightSum > maxRCenter) maxRCenter = rightSum;  
}  
  
int maxLCenter = 0, leftSum = 0;  
for (int i = center + 1; i <= right; ++i) {  
    leftSum += a[i];  
    if (leftSum > maxLCenter) maxLCenter = leftSum;  
}  
  
int maxCenter = maxRCenter + maxLCenter;  
return max3(maxLeft, maxRight, maxCenter);  
}
```

Max Subsequence Sum: Divide&Conquer

$$T(1) = 1$$

$$T(n) = 2T(n/2) + O(n)$$

We will see how to solve this equation formally in the next lesson (Master Theorem). Informally:

$$\begin{aligned} T(n) &= 2T(n/2) + n = 2(2(T(n/4) + n/2)) + n \\ &= 4T(n/4) + n + n = 8T(n/8) + n + n + n = \dots \\ &= 2^k T(n/2^k) + \underbrace{n + n + \dots + n}_k \end{aligned}$$

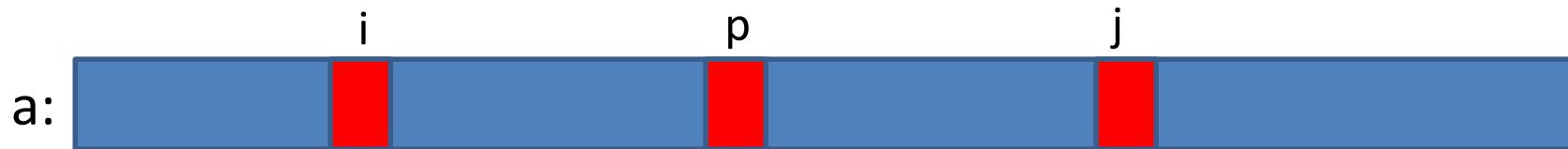
when $n = 2^k$ we have that $k = \log_2 n$

$$T(n) = 2^k T(1) + kn = n + n \log_2 n = O(n \log n)$$

But, can we still do it faster?

The Maximum Subsequence Sum Problem

- Observations:
 - If $a[i]$ is negative, it cannot be the start of the optimal subsequence.
 - Any negative subsequence cannot be the prefix of the optimal subsequence.
- Let us consider the inner loop of the $O(n^2)$ algorithm and assume that $a[i..j-1]$ is positive and $a[i..j]$ is negative:



- If p is an index between $i+1$ and j , then any subsequence from $a[p]$ is not larger than any subsequence from $a[i]$ and including $a[p-1]$.
- If $a[j]$ makes the current subsequence negative, we can advance i to $j+1$.

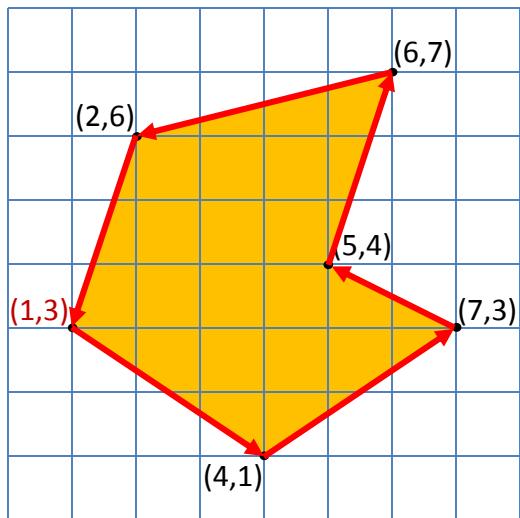
The Maximum Subsequence Sum Problem

```
int maxSubSum(const vector<int>& a) {  
    int maxSum = 0, thisSum = 0;  
    for (int i = 0; i < a.size(); ++i) {  
        int thisSum += a[i];  
        if (thisSum > maxSum) maxSum = thisSum;  
        else if (thisSum < 0) thisSum = 0;  
    }  
    return maxSum;  
}
```

$$T(n) = O(n)$$

a:	4	-3	5	-4	-3	-1	5	-2	6	-3	2
thisSum:	4	1	6	2	0	0	5	3	9	6	8
maxSum:	4	4	6	6	6	6	6	6	9	9	9

Representation of polygons



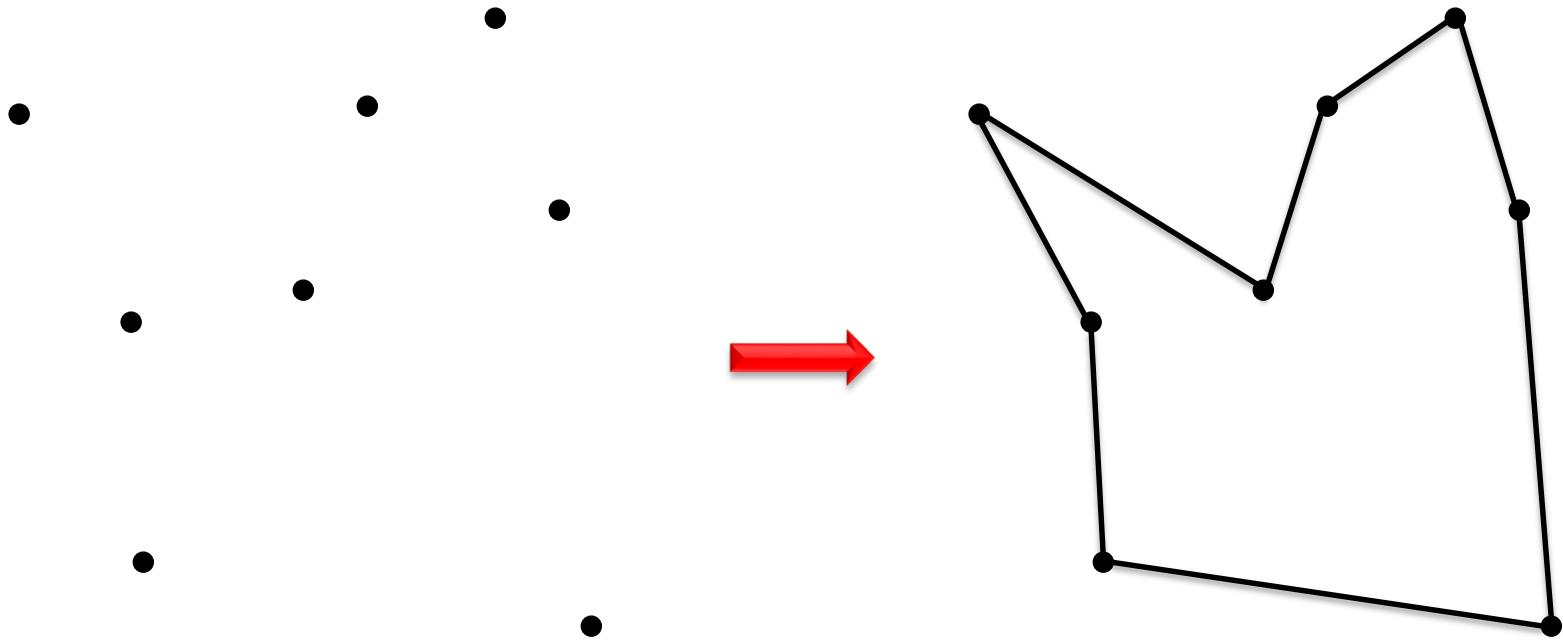
- A polygon can be represented by a sequence of vertices.
- Two consecutive vertices represent an edge of the polygon.
- The last edge is represented by the first and last vertices of the sequence.

Vertices: (1,3) (4,1) (7,3) (5,4) (6,7) (2,6)

Edges: (1,3)-(4,1)-(7,3)-(5,4)-(6,7)-(2,6)-(1,3)

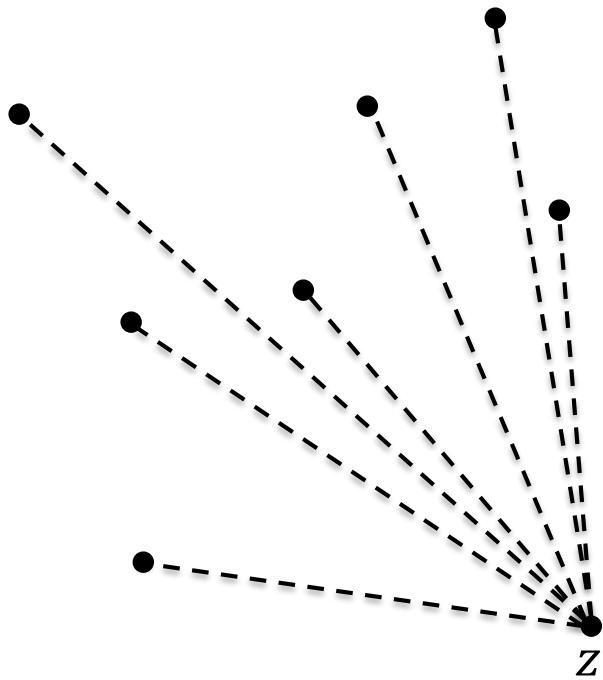
```
// A polygon (an ordered set of vertices)  
using Polygon = vector<Point>;
```

Create a polygon from a set of points



Given a set of n points in the plane, connect them in a simple closed path.

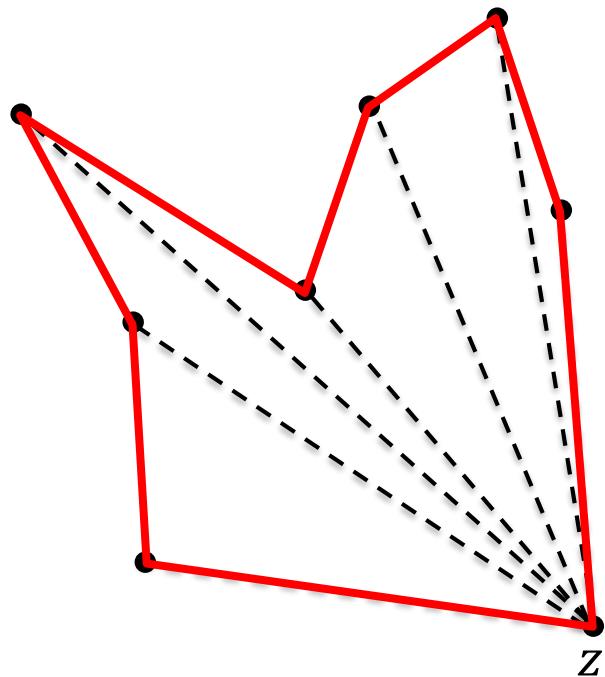
Simple polygon



- **Input:** p_1, p_2, \dots, p_n (points in the plane).
- **Output:** P (a polygon whose vertices are p_1, p_2, \dots, p_n in some order).
- Select a point z with the largest x coordinate (and smallest y in case of a tie in the x coordinate). Assume $z = p_1$.
- For each $p_i \in \{p_2, \dots, p_n\}$, calculate the angle α_i between the lines $z - p_i$ and the x axis.
- Sort the points $\{p_2, \dots, p_n\}$ according to their angles. In case of a tie, use distance to z .

Simple polygon

Implementation details:

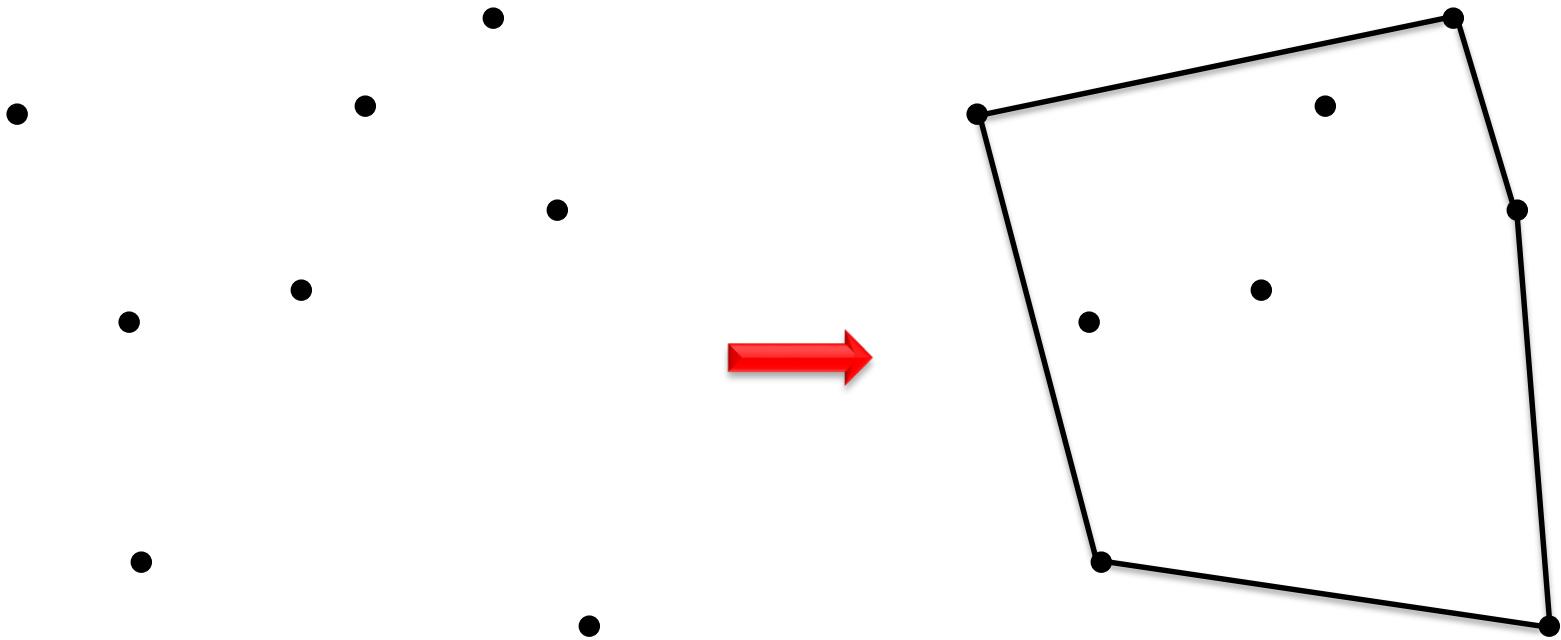


- There is no need to calculate angles (requires arctan). It is enough to calculate slopes ($\Delta y / \Delta x$).
- There is not need to calculate distances. It is enough to calculate the square of distances (no sqrt required).

Complexity: $O(n \log n)$.

The runtime is dominated by the sorting algorithm.

Convex hull

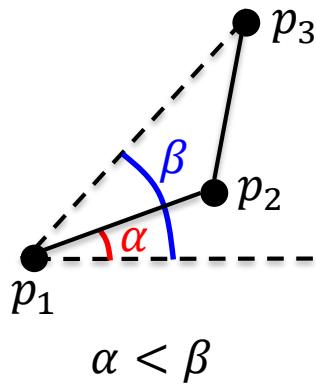


Compute the convex hull of n given points in the plane.

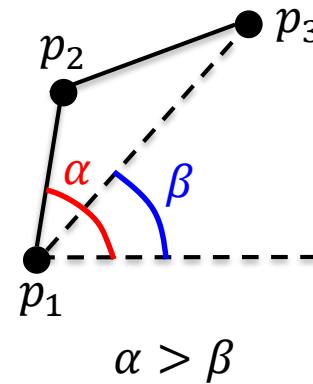
Clockwise and counter-clockwise

How to calculate whether three consecutive vertices are in a **clockwise** or **counter-clockwise** turn.

counter-clockwise
(p_3 at the left of $\overrightarrow{p_1 p_2}$)

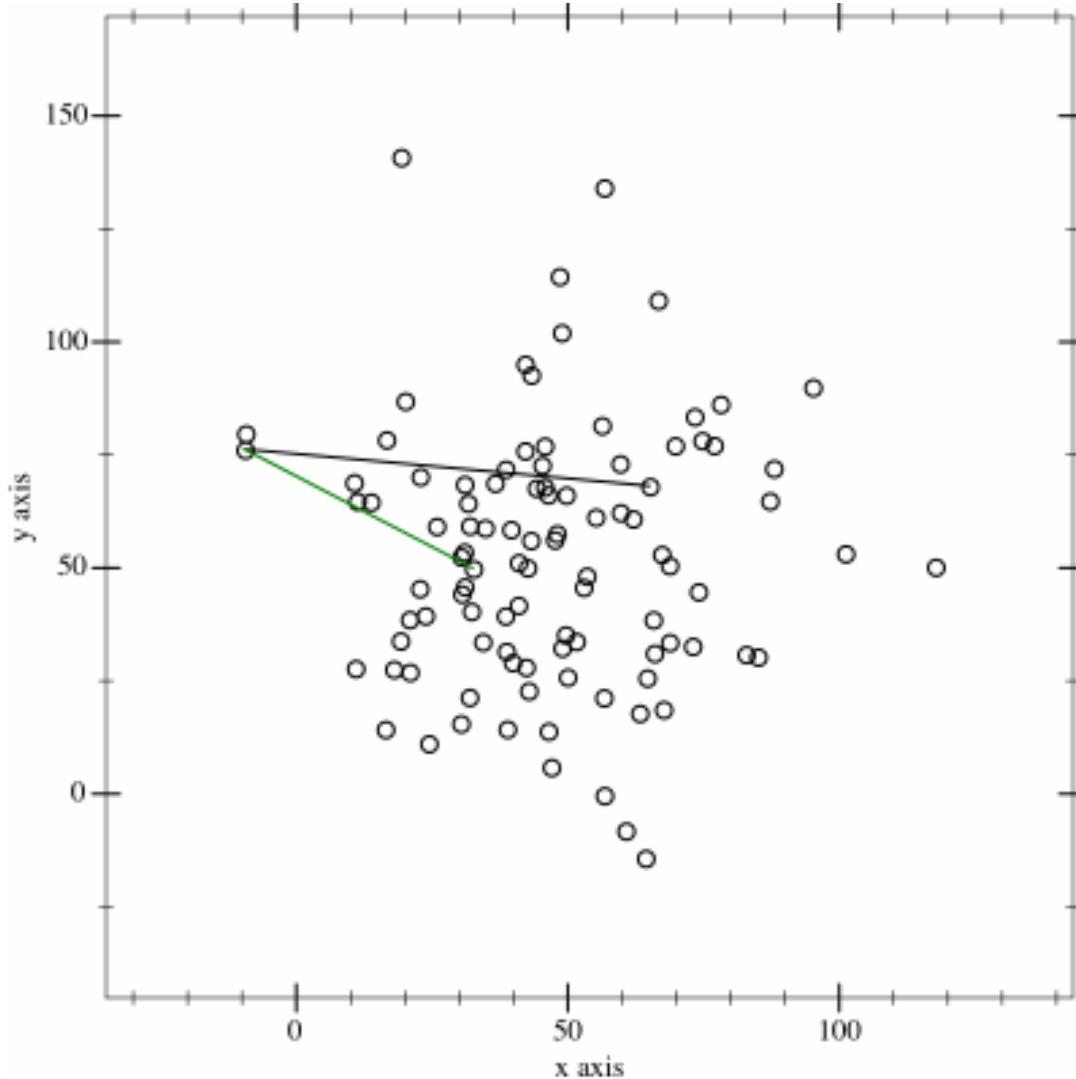


clockwise
(p_3 at the right of $\overrightarrow{p_1 p_2}$)



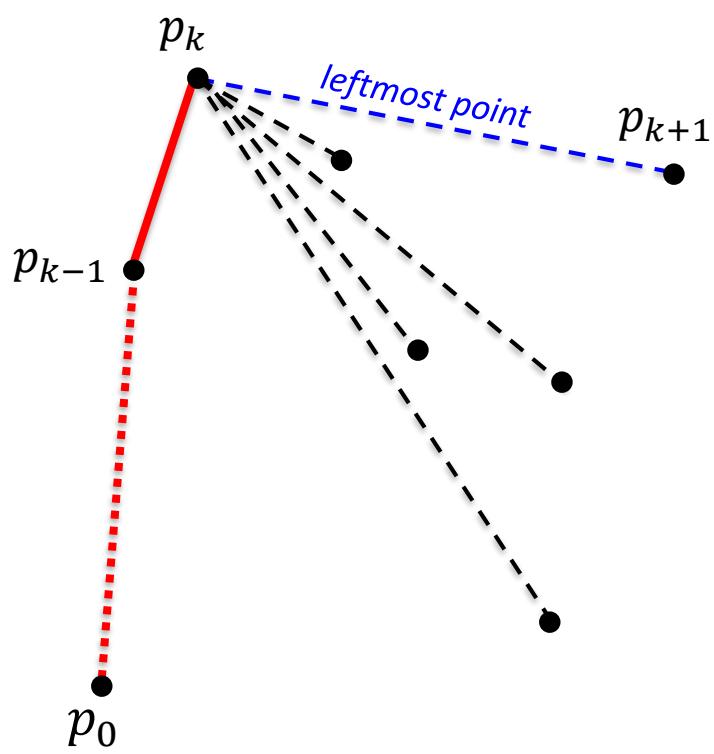
```
// Returns true if  $p_3$  is at the left of  $\overrightarrow{p_1 p_2}$ 
bool leftof(p1, p2, p3) {
    return (p2.x - p1.x) · (p3.y - p1.y) > (p2.y - p1.y) · (p3.x - p1.x);
}
```

Convex hull: gift wrapping algorithm



https://en.wikipedia.org/wiki/Gift_wrapping_algorithm

Convex hull: gift wrapping algorithm



- **Input:** p_1, p_2, \dots, p_n (points in the plane).
- **Output:** P (the convex hull of p_1, p_2, \dots, p_n).
- **Initial points:**
 p_0 with the smallest x coordinate.
- **Iteration:** Assume that a partial path with k points has been built (p_k is the last point). Pick some arbitrary $p_{k+1} \neq p_k$. Visit the remaining points. If some point q is at the left of $\overrightarrow{p_k p_{k+1}}$ redefine $p_{k+1} = q$.
- Stop when P is complete (back to point p_0).

Complexity: At each iteration, we calculate n angles. $T(n) = O(hn)$, where h is the number of points in the convex hull. In the worst case, $T(n) = O(n^2)$.

Convex hull: gift wrapping algorithm

```
vector<Point> convexHull(vector<Point> points) {
    int n = points.size();
    vector<Point> hull;

    // Pick the leftmost point
    int left = 0;
    for (int i = 1; i < n; i++)
        if (points[i].x < points[left].x) left = i;

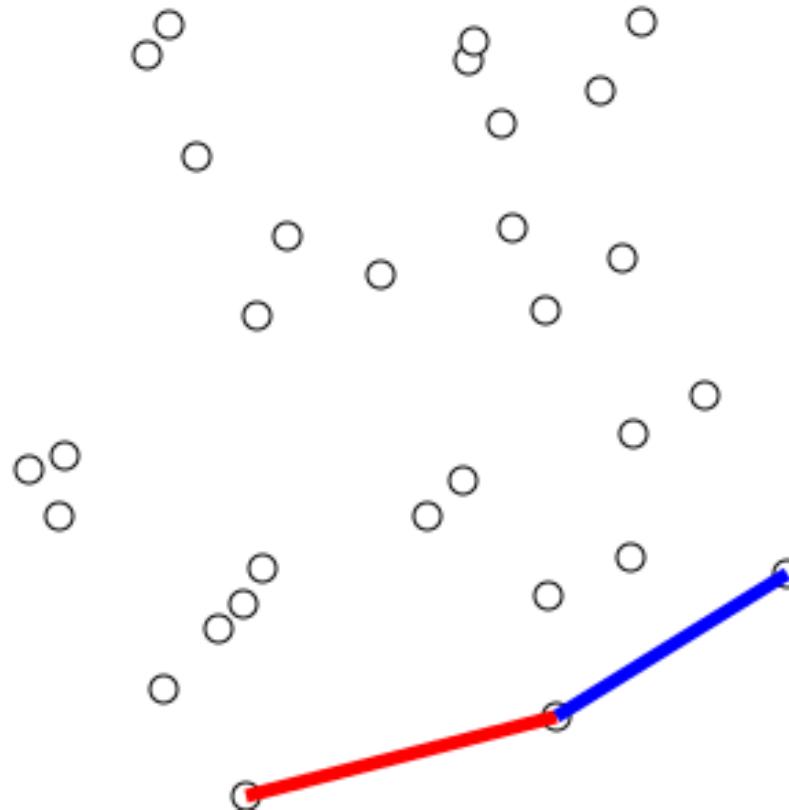
    int p = left, q;
    do {
        hull.push_back(points[p]); // Add point to the convex hull

        q = (p+1)%n; // Pick a point different from p
        for (int i = 0; i < n; i++)
            if (leftof(points[p], points[q], points[i])) q = i;

        p = q; // Leftmost point for the convex hull
    } while (p != left); // While not closing polygon

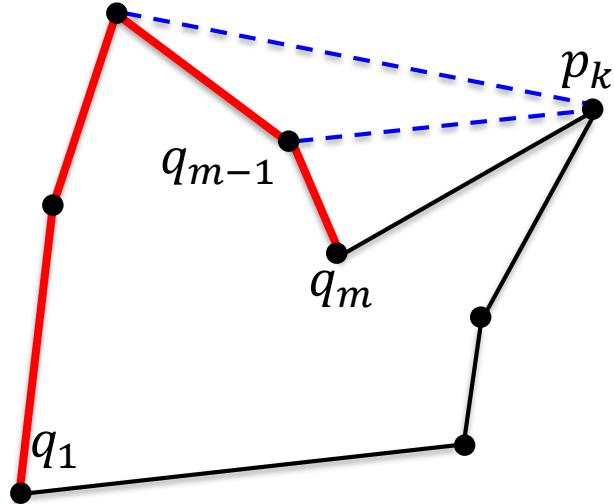
    return hull;
}
```

Convex hull: Graham Scan



https://en.wikipedia.org/wiki/Graham_scan

Convex hull: Graham scan



Input: p_1, p_2, \dots, p_n (points in the plane).

Output: q_1, q_2, \dots, q_m (the convex hull).

Initially:

Create a simple polygon P (complexity $O(n \log n)$).
Assume the order of the points is p_1, p_2, \dots, p_n .

```
// Q = (q1, q2, ...) is a vector where the points
// of the convex hull will be stored.
q1 = p1; q2 = p2; q3 = p3; m = 3;
for k = 4 to n:
    while leftof(qm-1, qm, pk): m = m - 1;
    m = m + 1;
    qm = pk;
```

Observation: each point p_k can be included in Q and deleted at most once.

The main loop of Graham scan has linear cost.

Complexity: dominated by the creation of the simple polygon $\rightarrow O(n \log n)$.

Divide & Conquer



Jordi Cortadella and Jordi Petit
Department of Computer Science

Divide-and-conquer algorithms

- Strategy:
 - Divide the problem into smaller subproblems of the same type of problem
 - Solve the subproblems recursively
 - Combine the answers to solve the original problem
- The work is done in three places:
 - In partitioning the problem into subproblems
 - In solving the basic cases at the tail of the recursion
 - In merging the answers of the subproblems to obtain the solution of the original problem

Conventional product of polynomials

Example:

$$P(x) = 2x^3 + x^2 - 4$$

$$Q(x) = x^2 - 2x + 3$$

$$(P \cdot Q)(x) = 2x^5 + (-4 + 1)x^4 + (6 - 2)x^3 + 8x - 12$$

$$(P \cdot Q)(x) = 2x^5 - 3x^4 + 4x^3 + 8x - 12$$

Conventional product of polynomials

```
function PolynomialProduct( $P$ ,  $Q$ )
    //  $P$  and  $Q$  are vectors of coefficients.
    // Returns  $R = P \times Q$ .
    //  $\text{degree}(P) = \text{size}(P)-1$ ,  $\text{degree}(Q) = \text{size}(Q)-1$ .
    //  $\text{degree}(R) = \text{degree}(P)+\text{degree}(Q)$ .

     $R$  = vector with  $\text{size}(P)+\text{size}(Q)-1$  zeros;

    for each  $P_i$ 
        for each  $Q_j$ 
             $R_{i+j} = R_{i+j} + P_i \cdot Q_j$ 

    return  $R$ 
```

Complexity analysis:

- Multiplication of polynomials of degree n : $O(n^2)$
- Addition of polynomials of degree n : $O(n)$

Product of polynomials: Divide&Conquer

Assume that we have two polynomials with n coefficients (degree $n - 1$)

	$n - 1$	$n/2$	0
$P:$	P_L		P_R
$Q:$	Q_L		Q_R

$$\begin{aligned} P(x) \cdot Q(x) = & P_L(x) \cdot Q_L(x) \cdot x^n + \\ & (P_R(x) \cdot Q_L(x) + P_L(x) \cdot Q_R(x)) \cdot x^{n/2} + \\ & P_R(x) \cdot Q_R(x) \end{aligned}$$

$$T(n) = 4 \cdot T(n/2) + O(n) = O(n^2) \quad \leftarrow \text{Shown later}$$

Product of complex numbers

- The product of two complex numbers requires four multiplications:

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

- Carl Friedrich Gauss (1777-1855) noticed that it can be done with just three: ac, bd and $(a + b)(c + d)$

$$bc + ad = (a + b)(c + d) - ac - bd$$

- A similar observation applies for polynomial multiplication.

Product of polynomials with Gauss's trick

$$R_1 = P_L Q_L$$

$$R_2 = P_R Q_R$$

$$R_3 = (P_L + P_R)(Q_L + Q_R)$$

$$PQ = \underbrace{P_L Q_L}_{R_1} x^n + \underbrace{(P_R Q_L + P_L Q_R)}_{R_3 - R_1 - R_2} x^{n/2} + \underbrace{P_R Q_R}_{R_2}$$

$$T(n) = 3T(n/2) + O(n)$$

Polynomial multiplication: recursive step

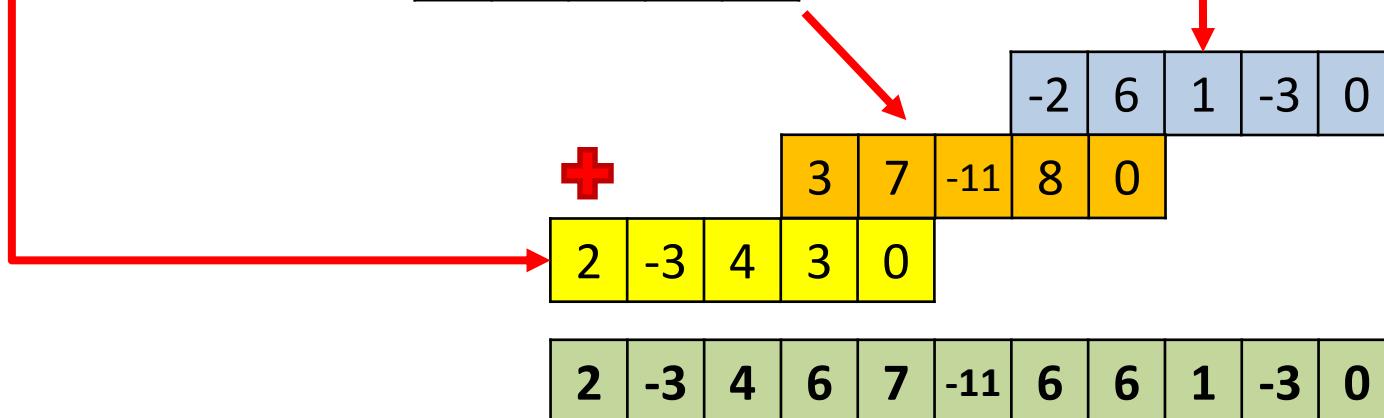
P	1	-2	3	2	0	-1
Q	2	1	0	-1	3	0

P_L	1	-2	3
$\times Q_L$	2	1	0
2	-3	4	3

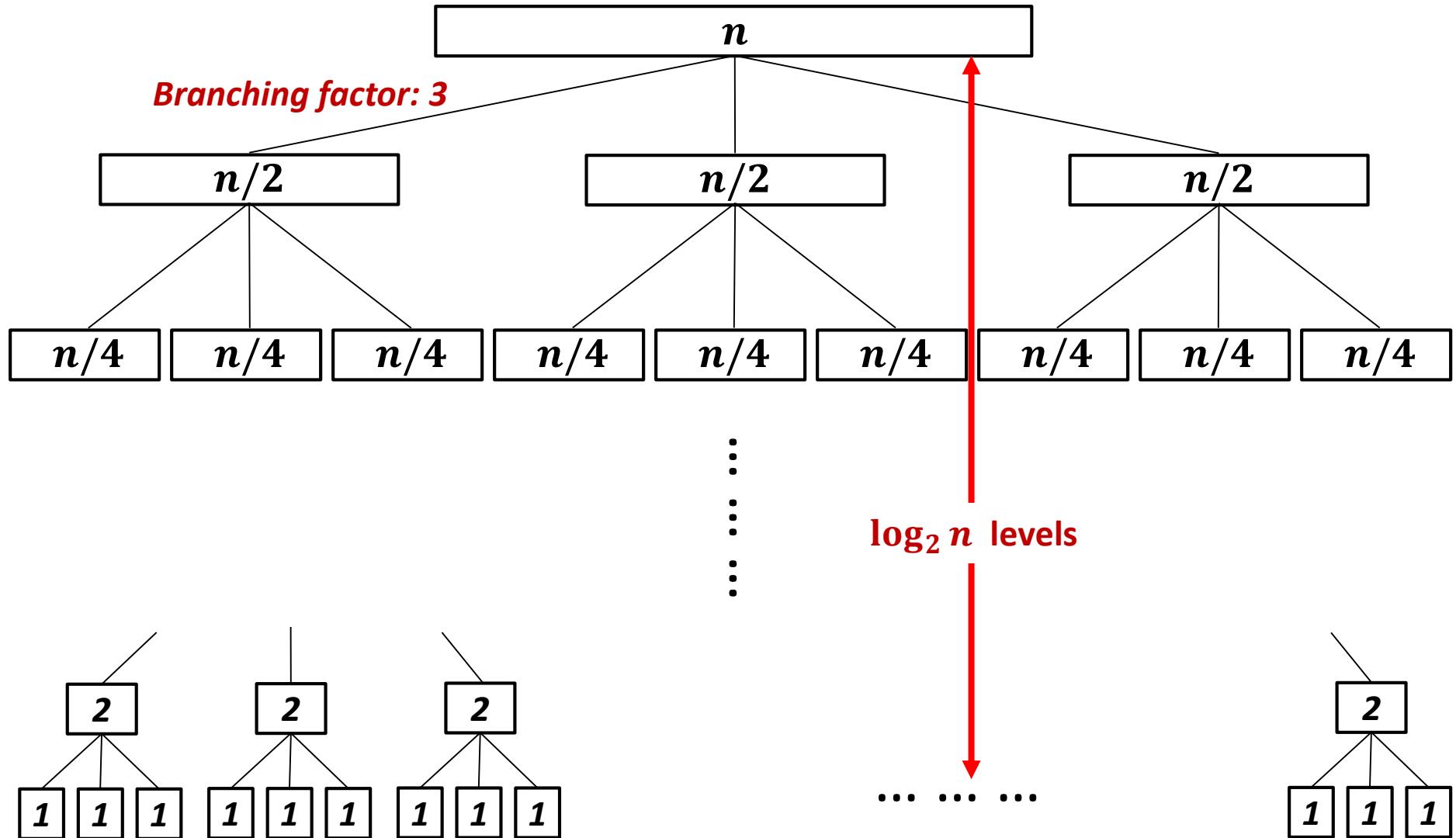
$P_L + P_R$	3	-2	2
$\times Q_L + Q_R$	1	4	0
	3	10	-6

	P_R	2	0	-1
\times	Q_R	-1	3	0
-2	6	1	-3	0

■	2	-3	4	3	0
■	-2	6	1	-3	0
$P_L Q_R + P_R Q_L$	3	7	-11	8	0



Pattern of recursive calls



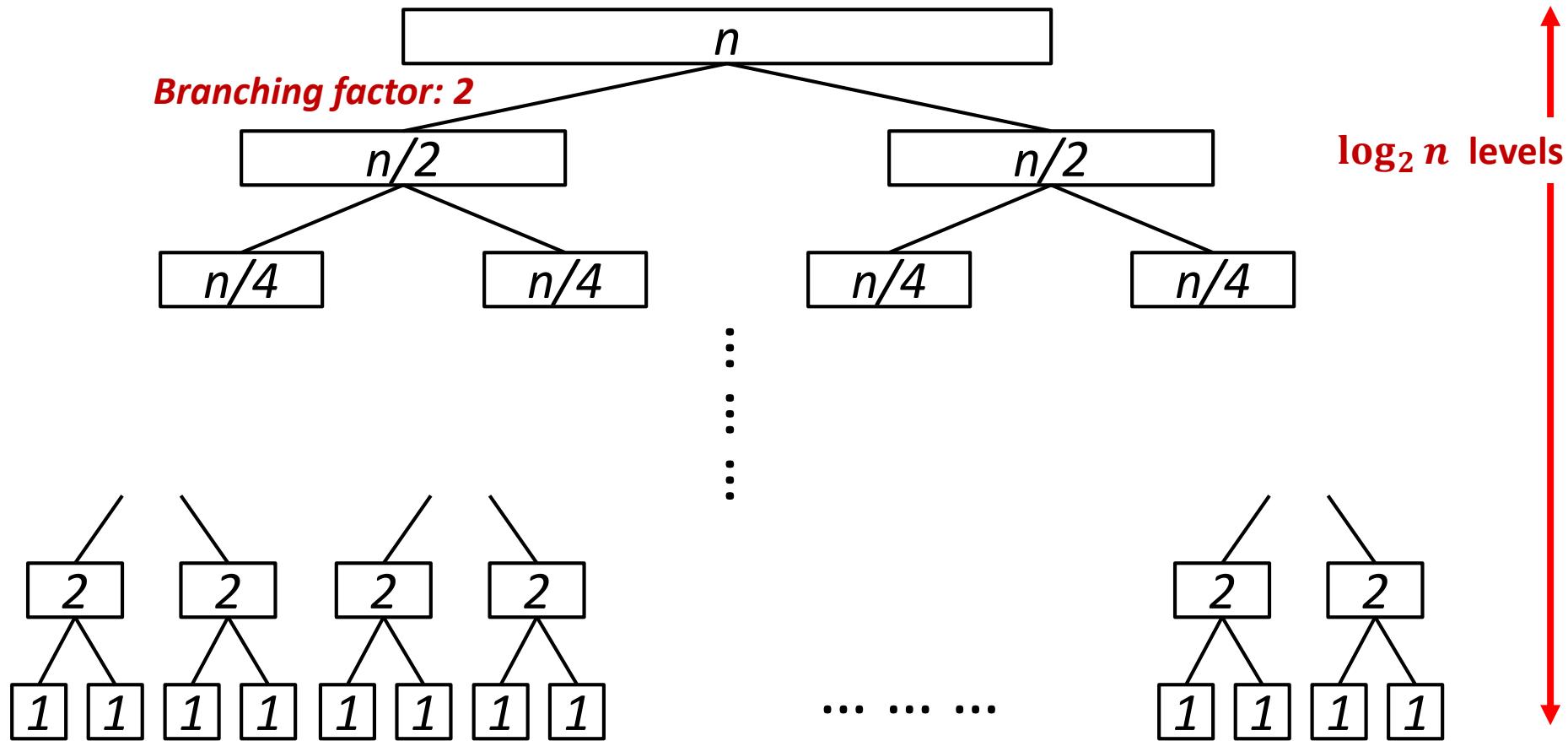
Complexity analysis

- The time spent at level k is

$$3^k \cdot O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \cdot O(n)$$

- For $k = 0$, runtime is $O(n)$.
- For $k = \log_2 n$, runtime is $O(3^{\log_2 n})$, which is equal to $O(n^{\log_2 3})$.
- The runtime per level increases geometrically by a factor of $3/2$ per level. The sum of any increasing geometric series is, within a constant factor, simply the last term of the series.
- Therefore, the complexity is $O(n^{1.59})$.

A popular recursion tree



Example: efficient sorting algorithms.

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

Algorithms may differ on the amount of work done at each level: $O(n^c)$

Examples

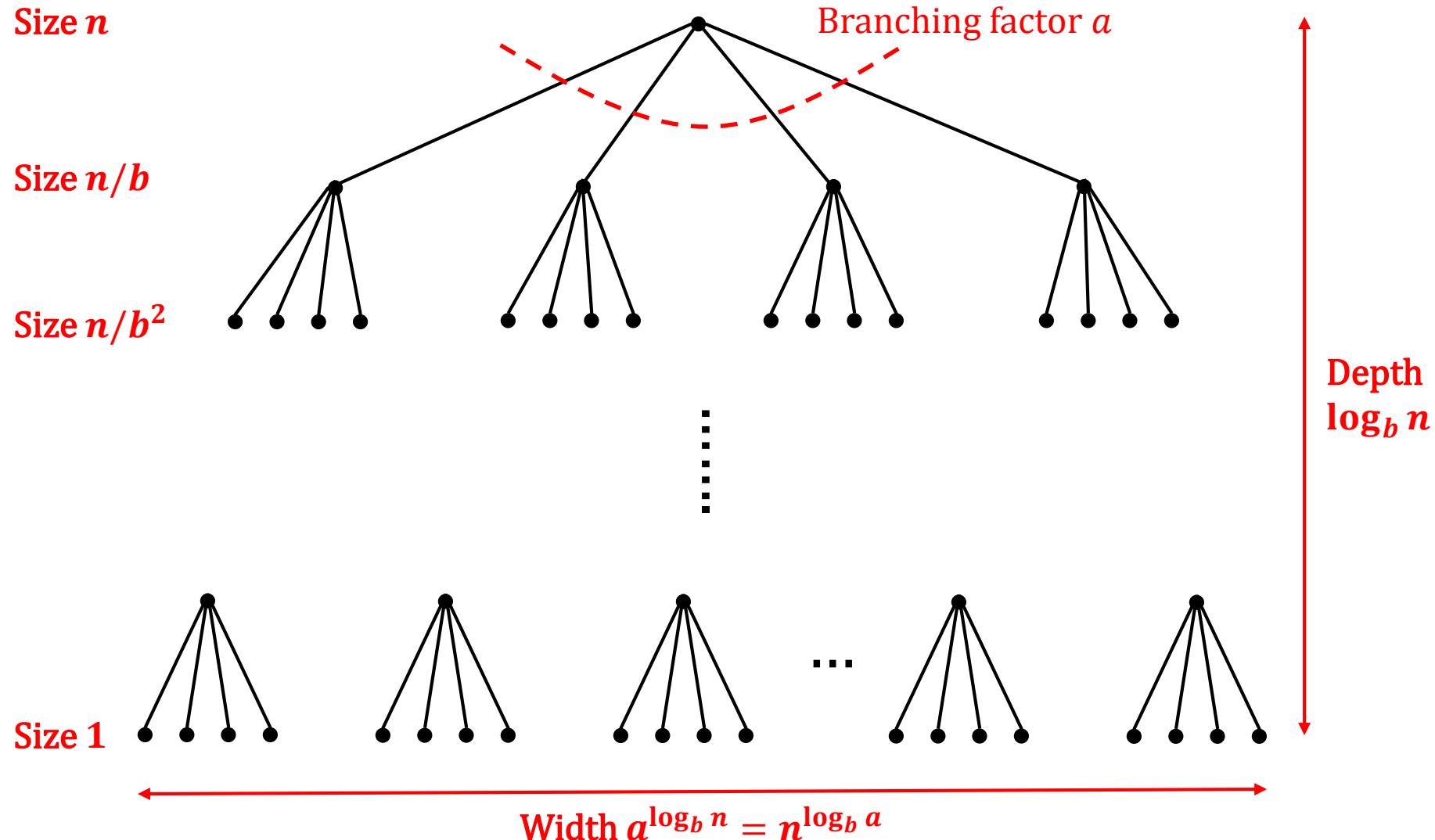
Algorithm	Branch	c	Runtime equation
Power (x^y)	1	0	$T(y) = T(y/2) + O(1)$
Binary search	1	0	$T(n) = T(n/2) + O(1)$
Merge sort	2	1	$T(n) = 2 \cdot T(n/2) + O(n)$
Polynomial product	4	1	$T(n) = 4 \cdot T(n/2) + O(n)$
Polynomial product (Gauss)	3	1	$T(n) = 3 \cdot T(n/2) + O(n)$

Master theorem

- Typical pattern for Divide&Conquer algorithms:
 - Split the problem into a subproblems of size n/b
 - Solve each subproblem recursively
 - Combine the answers in $O(n^c)$ time
- Running time: $T(n) = a \cdot T(n/b) + O(n^c)$
- Master theorem:

$$T(n) = \begin{cases} O(n^c) & \text{if } c > \log_b a & (a < b^c) \\ O(n^c \log n) & \text{if } c = \log_b a & (a = b^c) \\ O(n^{\log_b a}) & \text{if } c < \log_b a & (a > b^c) \end{cases}$$

Master theorem: recursion tree



Master theorem: proof

- For simplicity, assume n is a power of b .
- The base case is reached after $\log_b n$ levels.
- The k th level of the tree has a^k subproblems of size n/b^k .
- The total work done at level k is:

$$a^k \times O\left(\frac{n}{b^k}\right)^c = O(n^c) \times \left(\frac{a}{b^c}\right)^k$$

- As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio a/b^c . We need to find the sum of such a series.

$$T(n) = O(n^c) \cdot \left(1 + \frac{a}{b^c} + \frac{a^2}{b^{2c}} + \frac{a^3}{b^{3c}} + \cdots + \frac{a^{\log_b n}}{b^{(\log_b n)c}} \right)$$

Master theorem: proof

- Case $a/b^c < 1$. Decreasing series. The sum is dominated by the first term ($k = 0$): $O(n^c)$.
- Case $a/b^c < 1$. Increasing series. The sum is dominated by the last term ($k = \log_b n$):

$$\begin{aligned} n^c \left(\frac{a}{b^c}\right)^{\log_b n} &= n^c \left(\frac{a^{\log_b n}}{(b^{\log_b n})^c}\right) = a^{\log_b n} = \\ &= a^{(\log_a n)(\log_b a)} = n^{\log_b a} \end{aligned}$$

- Case $a/b^c = 1$. We have $O(\log n)$ terms all equal to $O(n^c)$.

Master theorem: examples

Running time: $T(n) = a \cdot T(n/b) + O(n^c)$

$$T(n) = \begin{cases} O(n^c) & \text{if } a < b^c \\ O(n^c \log n) & \text{if } a = b^c \\ O(n^{\log_b a}) & \text{if } a > b^c \end{cases}$$

Algorithm	a	c	Runtime equation	Complexity
Power (x^y)	1	0	$T(y) = T(y/2) + O(1)$	$O(\log y)$
Binary search	1	0	$T(n) = T(n/2) + O(1)$	$O(\log n)$
Merge sort	2	1	$T(n) = 2 \cdot T(n/2) + O(n)$	$O(n \log n)$
Polynomial product	4	1	$T(n) = 4 \cdot T(n/2) + O(n)$	$O(n^2)$
Polynomial product (Gauss)	3	1	$T(n) = 3 \cdot T(n/2) + O(n)$	$O(n^{\log_2 3})$

$b = 2$ for all the examples

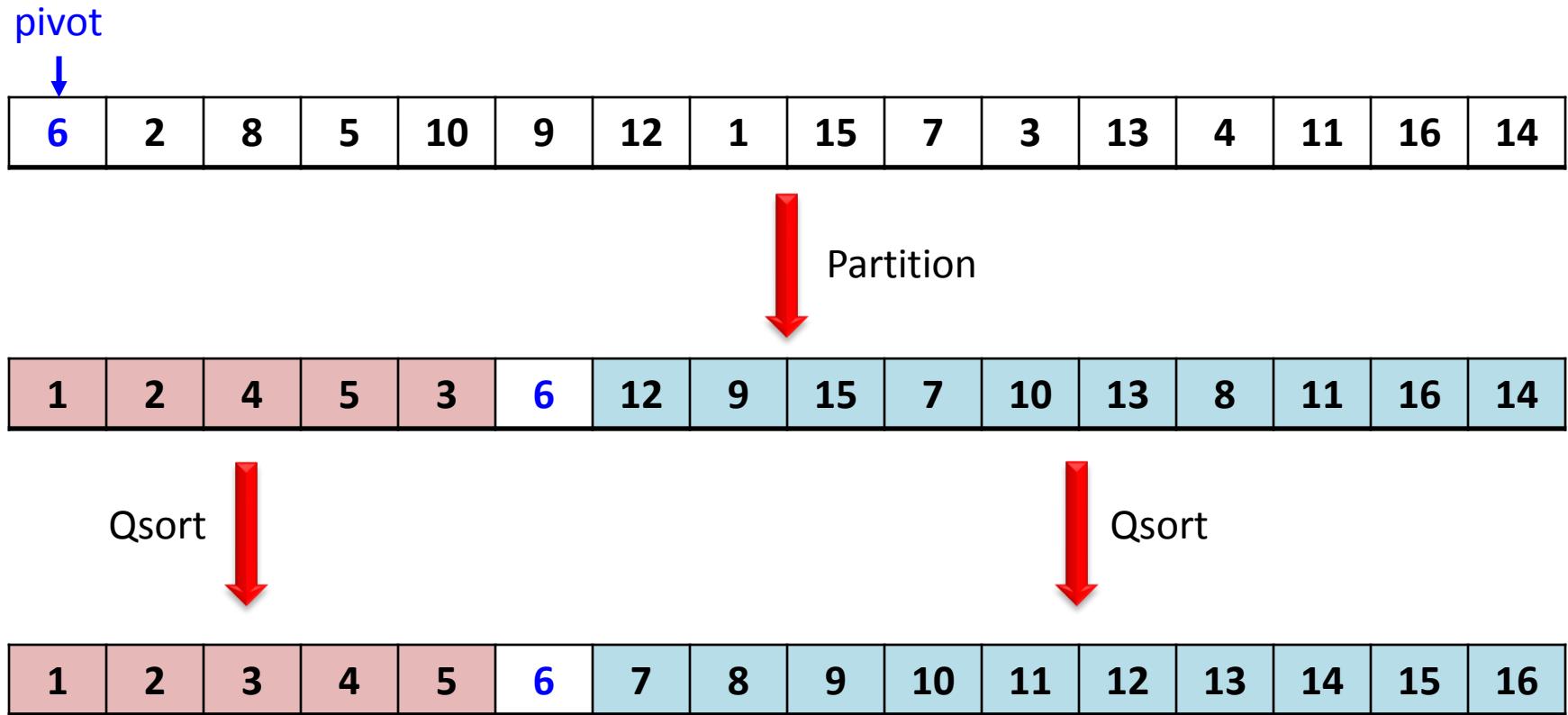
Quick sort (Tony Hoare, 1959)

- Suppose that we know a number x such that one-half of the elements of a vector are greater than or equal to x and one-half of the elements are smaller than x .
 - Partition the vector into two equal parts ($n - 1$ comparisons)
 - Sort each part recursively
- Problem: we do not know x .
- The algorithm also works no matter which x we pick for the partition. We call this number the **pivot**.
- **Observation:** the partition may be unbalanced.

Quick sort with Hungarian, folk dance



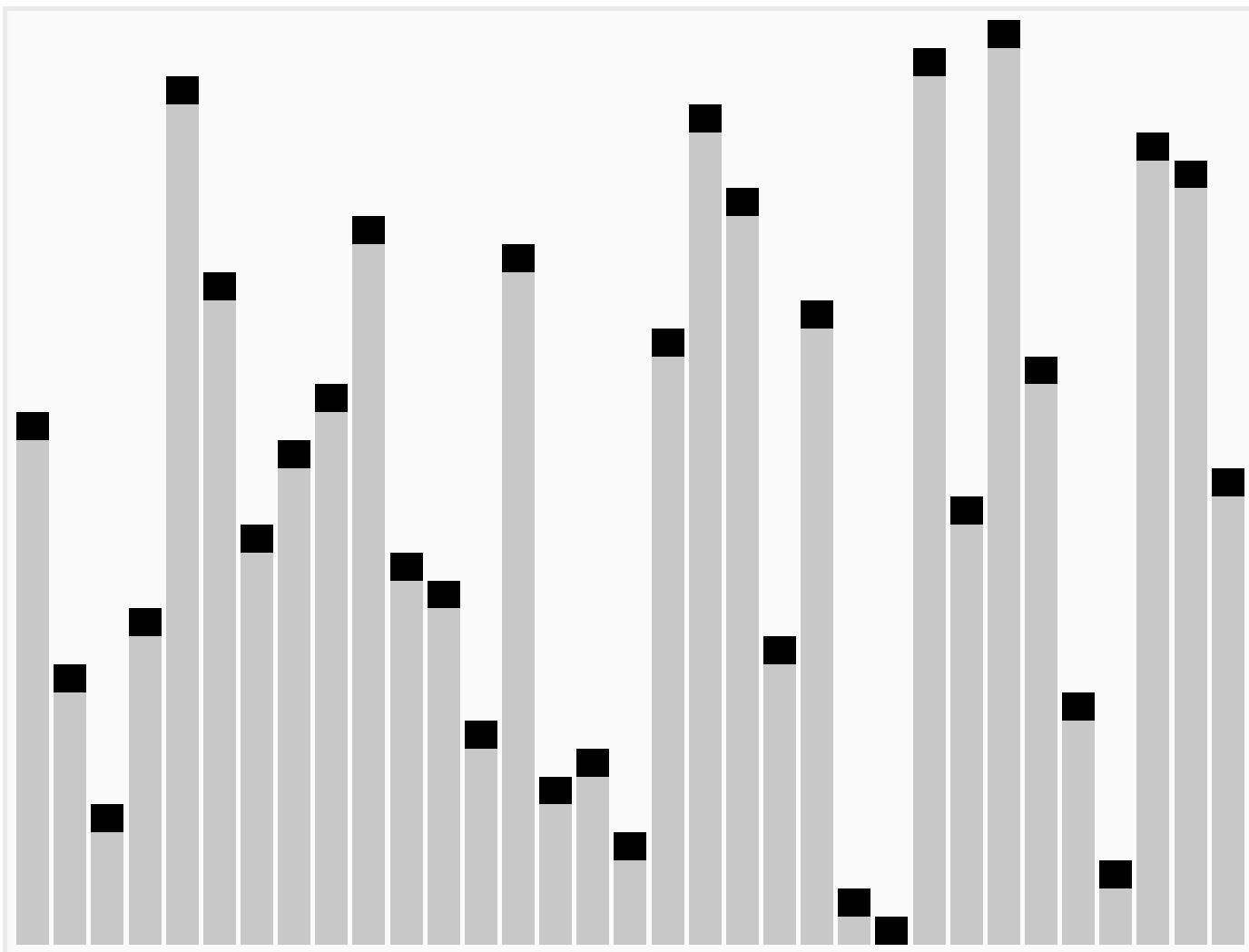
Quick sort: example



The key step of quick sort is the partitioning algorithm.

Question: how to find a good pivot?

Quick sort



<https://en.wikipedia.org/wiki/Quicksort>

Quick sort: partition

```
function Partition(A, left, right)
    // A[left..right]: segment to be sorted
    // Returns the middle of the partition with
    //   A[middle] = pivot
    //   A[left..middle-1] ≤ pivot
    //   A[middle+1..right] > pivot

    x = A[left]; // the pivot
    i = left; j = right;

    while i < j do
        while A[i] ≤ x and i ≤ right do i = i+1;
        while A[j] > x and j ≥ left do j = j-1;
        if i < j then swap(A[i], A[j]);

    swap(A[left], A[j]);
    return j;
```

Quick sort partition: example

pivot



6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
---	---	---	---	----	---	----	---	----	---	---	----	---	----	----	----

6	2	4	5	10	9	12	1	15	7	3	13	8	11	16	14
---	---	---	---	----	---	----	---	----	---	---	----	---	----	----	----

6	2	4	5	3	9	12	1	15	7	10	13	8	11	16	14
---	---	---	---	---	---	----	---	----	---	----	----	---	----	----	----

6	2	4	5	3	1	12	9	15	7	10	13	8	11	16	14
---	---	---	---	---	---	----	---	----	---	----	----	---	----	----	----

1	2	4	5	3	6	12	9	15	7	10	13	8	11	16	14
---	---	---	---	---	---	----	---	----	---	----	----	---	----	----	----

1	2	4	5	3	6	12	9	15	7	10	13	8	11	16	14
---	---	---	---	---	---	----	---	----	---	----	----	---	----	----	----

middle

Quick sort: algorithm

```
function Qsort(A, left, right)  
// A[left..right]: segment to be sorted  
  
if left < right then  
    mid = Partition(A, left, right);  
    Qsort(A, left, mid-1);  
    Qsort(A, mid+1, right);
```

Quick sort: Hoare's partition

```
function HoarePartition(A, left, right)

// A[left..right]: segment to be sorted.
// Output: The left part has elements  $\leq$  than the pivot.
// The right part has elements  $\geq$  than the pivot.
// Returns the index of the last element of the left part.

x = A[left]; // the pivot
i = left-1; j = right+1;

while true do
    do i = i+1; while A[i] < x;
    do j = j-1; while A[j] > x;

    if i  $\geq$  j then return j;

    swap(A[i], A[j]);
```

Admire a unique piece of art by Hoare:
The first swap creates two sentinels.
After that, the algorithm flies ...

Quick sort partition: example

pivot



6	2	8	5	10	9	12	1	15	7	3	13	4	11	16	14
---	---	---	---	----	---	----	---	----	---	---	----	---	----	----	----

First swap: 4 is a sentinel for R; 6 is a sentinel for L \rightarrow no need to check for boundaries

4	2	8	5	10	9	12	1	15	7	3	13	6	11	16	14
i												j			

4	2	3	5	10	9	12	1	15	7	8	13	6	11	16	14
---	---	---	---	----	---	----	---	----	---	---	----	---	----	----	----

4	2	3	5	1	9	12	10	15	7	8	13	6	11	16	14
---	---	---	---	---	---	----	----	----	---	---	----	---	----	----	----

4	2	3	5	1	9	12	10	15	7	8	13	6	11	16	14
---	---	---	---	---	---	----	----	----	---	---	----	---	----	----	----



j (middle)

Quick sort with Hoare's partition

```
function Qsort(A, left, right)  
  
// A[left..right]: segment to be sorted  
  
if left < right then  
    mid = HoarePartition(A, left, right);  
    Qsort(A, left, mid);  
    Qsort(A, mid+1, right);
```

Quick sort: hybrid approach

```
function Qsort(A, left, right)
    // A[left..right]: segment to be sorted.
    // K is a break-even size in which insertion sort is
    // more efficient than quick sort.
    if right - left ≥ K then
        mid = HoarePartition(A, left, right);
        Qsort(A, left, mid);
        Qsort(A, mid+1, right);
```

```
function Sort(A):
    Qsort(A, 0, A.size()-1);
    InsertionSort(A);
```

Quick sort: complexity analysis

- The partition algorithm is $O(n)$.
- Assume that the partition is balanced:

$$T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)$$

- Worst case runtime: the pivot is always the smallest element in the vector $\rightarrow O(n^2)$
- Selecting a good pivot is essential. There are different strategies, e.g.,
 - Take the median of the first, last and middle elements
 - Take the pivot at random

Quick sort: complexity analysis

- Let us assume that x_i is the i th smallest element in the vector.
- Let us assume that each element has the same probability of being selected as pivot.
- The runtime if x_i is selected as pivot is:

$$T(n) = n - 1 + T(i - 1) + T(n - i)$$

Quick sort: complexity analysis

$$T(n) = n - 1 + \frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i))$$

$$T(n) = n - 1 + \frac{1}{n} \sum_{i=1}^n T(i-1) + \frac{1}{n} \sum_{i=1}^n T(n-i)$$

$$T(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \leq 2(n+1)(H(n+1) - 1.5)$$

$H(n) = 1 + 1/2 + 1/3 + \dots + 1/n$ is the Harmonic series, that has a simple approximation: $H(n) = \ln n + \gamma + O(1/n)$.

$\gamma = 0.577 \dots$ is Euler's constant.

$$T(n) \leq 2(n+1)(\ln n + \gamma - 1.5) + O(1) = O(n \log n)$$

Quick sort: complexity analysis summary

- Runtime of quicksort:

$$T(n) = O(n^2)$$

$$T(n) = \Omega(n \log n)$$

$$T_{\text{avg}}(n) = O(n \log n)$$

- Be careful: some malicious patterns may increase the probability of the worst case runtime, e.g., when the vector is sorted or almost sorted.
- Possible solution: use random pivots.

The selection problem

- Given a collection of N elements, find the k th smallest element.
- Options:
 - Sort a vector and select the k th location: $O(N \log N)$
 - Read k elements into a vector and sort them. The remaining elements are processed one by one and placed in the correct location (similar to insertion sort). Only k elements are maintained in the vector.
Complexity: $O(kN)$. Why?

The selection problem using a heap

- Algorithm:
 - Build a heap from the collection of elements: $O(N)$
 - Remove k elements: $O(k \log N)$
 - Note: heaps will be seen later in the course
- Complexity:
 - In general: $O(N + k \log N)$
 - For small values of k , i.e., $k = O(N/\log N)$, the complexity is $O(N)$.
 - For large values of k , the complexity is $O(k \log N)$.

Quick sort with Hoare's partition

```
function Qsort(A, left, right)  
  
// A[left..right]: segment to be sorted  
  
if left < right then  
    mid = HoarePartition(A, left, right);  
    Qsort(A, left, mid);  
    Qsort(A, mid+1, right);
```

Quick select with Hoare's partition

```
// Returns the element at location k assuming
// A[left..right] would be sorted in ascending order.
// Pre: left ≤ k ≤ right.
// Post: The elements of A have changed their locations.

function Qselect(A, left, right, k)
    if left == right then return A[left];

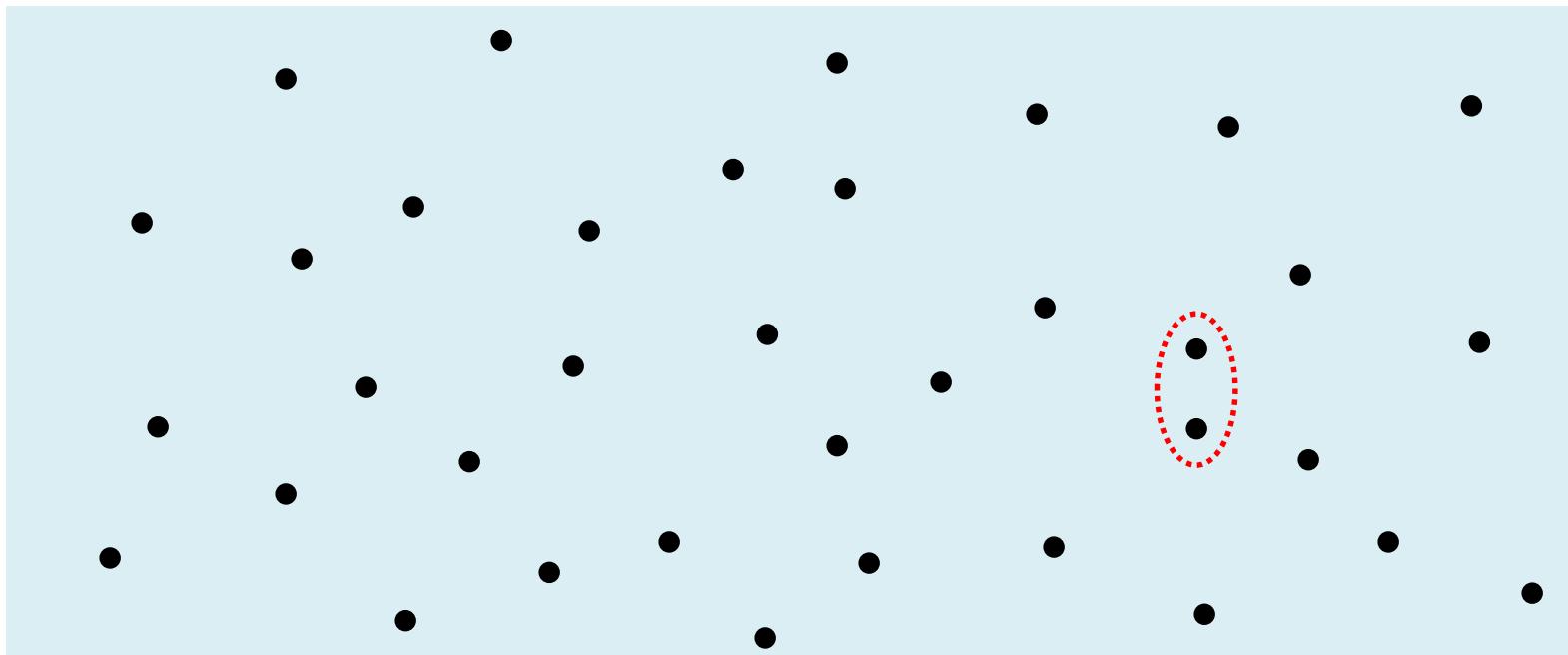
    mid = HoarePartition(A, left, right);
    // We only need to sort one half of A
    if k ≤ mid then Qselect(A, left, mid, k);
    else Qselect(A, mid+1, right, k);
```

Quick Select: complexity

- Assume that the partition is balanced:
 - Quick sort: $T(n) = 2T(n/2) + O(n) = O(n \log n)$
 - Quick select: $T(n) = T(n/2) + O(n) = O(n)$
- The average linear time complexity can be achieved by choosing good pivots (similar strategy and complexity computation to qsort).

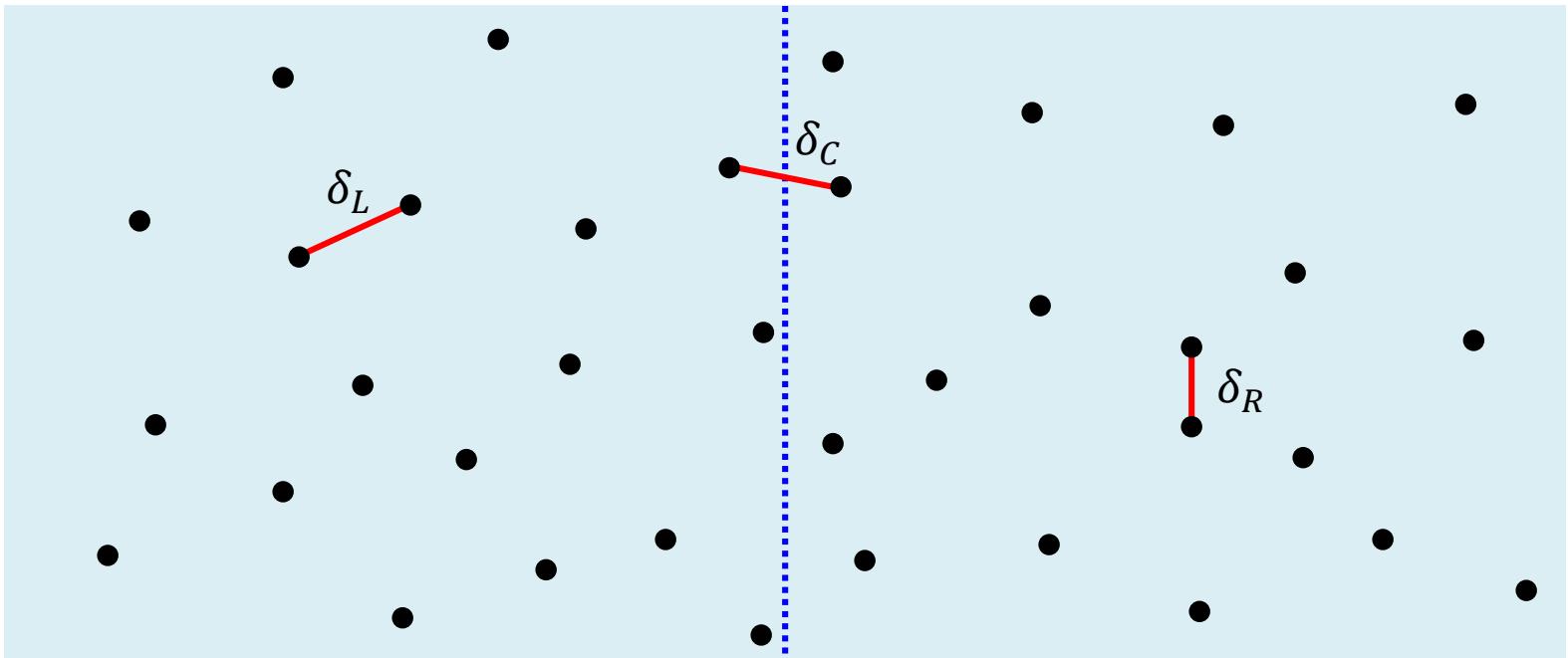
The Closest-Points problem

- **Input:** A list of n points in the plane
 $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$
- **Output:** The pair of closest points
- **Simple approach:** check all pairs $\rightarrow O(n^2)$
- We want an $O(n \log n)$ solution !



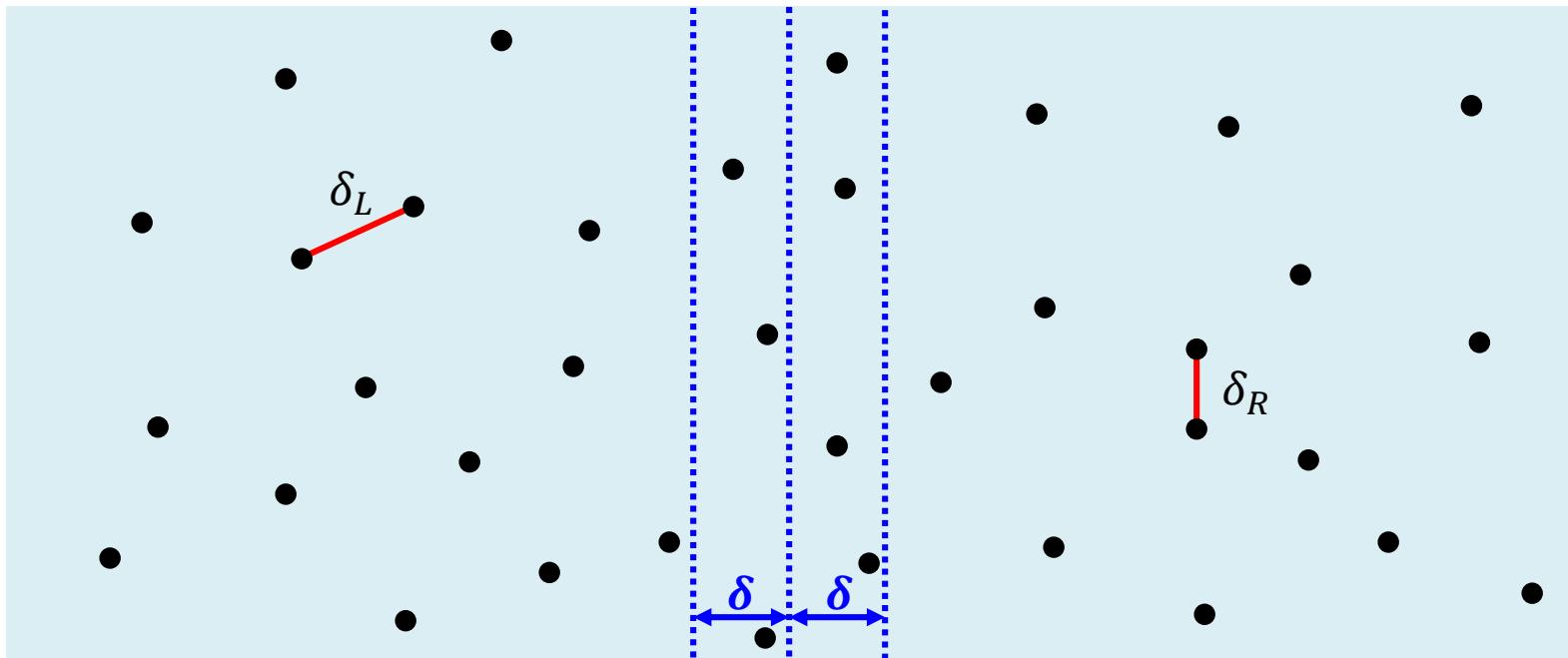
The Closest-Points problem

- We can assume that the points are sorted by the x -coordinate. Sorting the points is free from the complexity standpoint ($O(n \log n)$).
- Split the list into two halves. The closest points can be both at the left, both at the right or one at the left and the other at the right (center).
- The left and right pairs are easy to find (recursively). How about the pairs in the center?



The Closest-Points problem

- Let $\delta = \min(\delta_L, \delta_R)$. We only need to compute δ_C if it improves δ .
- We can define a strip around de center with distance δ at the left and right. If δ_C improves δ , then the points must be within the strip.
- In the worst case, all points can still reside in the strip.
- But how many points do we really have to consider?



The Closest-Points problem

Let us take all points in the strip and sort them by the y -coordinate. We only need to consider pairs of points with distance smaller than δ .

Once we find a pair (p_i, p_j) with y -coordinates that differ by more than δ , we can move to the next p_i .

```
for (i=0; i < NumPointsInStrip; ++i)
    for (j=i+1; j < NumPointsInStrip; ++j)

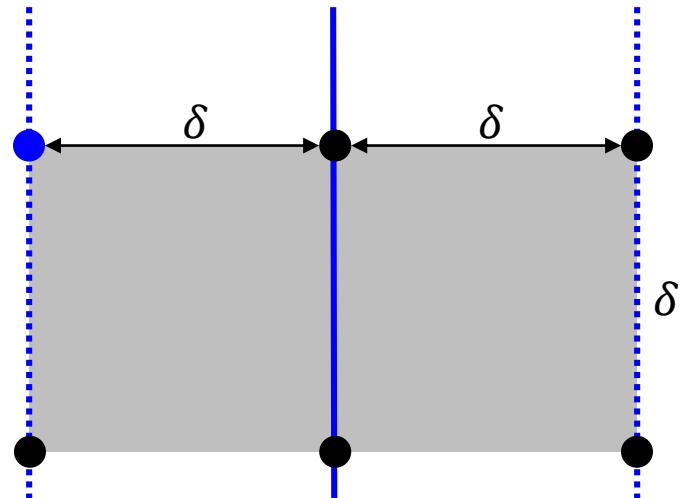
        if ( $p_i$  and  $p_j$ 's  $y$ -coordinate differ by
            more than  $\delta$ ) break; // Go to next  $p_i$ 

        if ( $dist(p_i, p_j) < \delta$ )  $\delta = dist(p_i, p_j);$ 
```

But, how many pairs (p_i, p_j) do we need to consider?

The Closest-Points problem

- For every point p_i at one side of the strip, we only need to consider points from p_{i+1} .
- The relevant points only reside in the $2\delta \times \delta$ rectangle below point p_i . There can only be 8 points at most in this rectangle (4 at the left and 4 at the right). Some points may have the same coordinates.



The Closest-Points problem: algorithm

- Sort the points according to their x -coordinates.
- Divide the set into two equal-sized parts.
- Compute the min distance at each part (recursively).
Let δ be the minimal of the two minimal distances.
- Eliminate points that are farther than δ from the separation line.
- Sort the remaining points according to their y -coordinates.
- Scan the remaining points in the y order and compute the distances of each point to its 7 neighbors.

The Closest-Points problem: complexity

- Initial sort using x -coordinates: $O(n \log n)$.
It comes for free.
- Divide and conquer:
 - Solve for each part recursively: $2T(n/2)$
 - Eliminate points farther than δ : $O(n)$
 - Sort remaining points using y -coordinates: $O(n \log n)$
 - Scan the remaining points in y order: $O(n)$

$$T(n) = 2T(n/2) + O(n) + O(n \log n) = O(n \log^2 n)$$

- Can we do it in $O(n \log n)$? Yes, we need to sort by y in a smart way.

The Closest-Points problem: complexity

- Let Y a vector with the points sorted by the y -coordinates. This can be done initially for free.
- Each time we partition the set of points by the x -coordinate, we also partition Y into two sorted vectors (using an “*unmerging*” procedure with linear complexity)

```
 $Y_L = Y_R = \emptyset$  // Initial lists of points
foreach  $p_i \in Y$  in ascending order of  $y$  do
    if  $p_i$  is at the left part then  $Y_L.push\_back(p_i)$ 
    else  $Y_R.push\_back(p_i)$ 
```

- Now, sorting the points by the y -coordinate at each iteration can be done in linear time, and the problem can be solved in $O(n \log n)$

APPENDIX

Full-history recurrence relation

$$T(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} T(i)$$

A recurrence that depends on all the previous values of the function.

$$nT(n) = n(n - 1) + 2 \sum_{i=0}^{n-1} T(i), \quad (n + 1)T(n + 1) = (n + 1)n + 2 \sum_{i=0}^n T(i)$$

$$(n + 1)T(n + 1) - nT(n) = (n + 1)n - n(n - 1) + 2T(n) = 2n + 2T(n)$$

$$T(n + 1) = \frac{n + 2}{n + 1}T(n) + \frac{2n}{n + 1} \leq \frac{n + 2}{n + 1}T(n) + 2$$

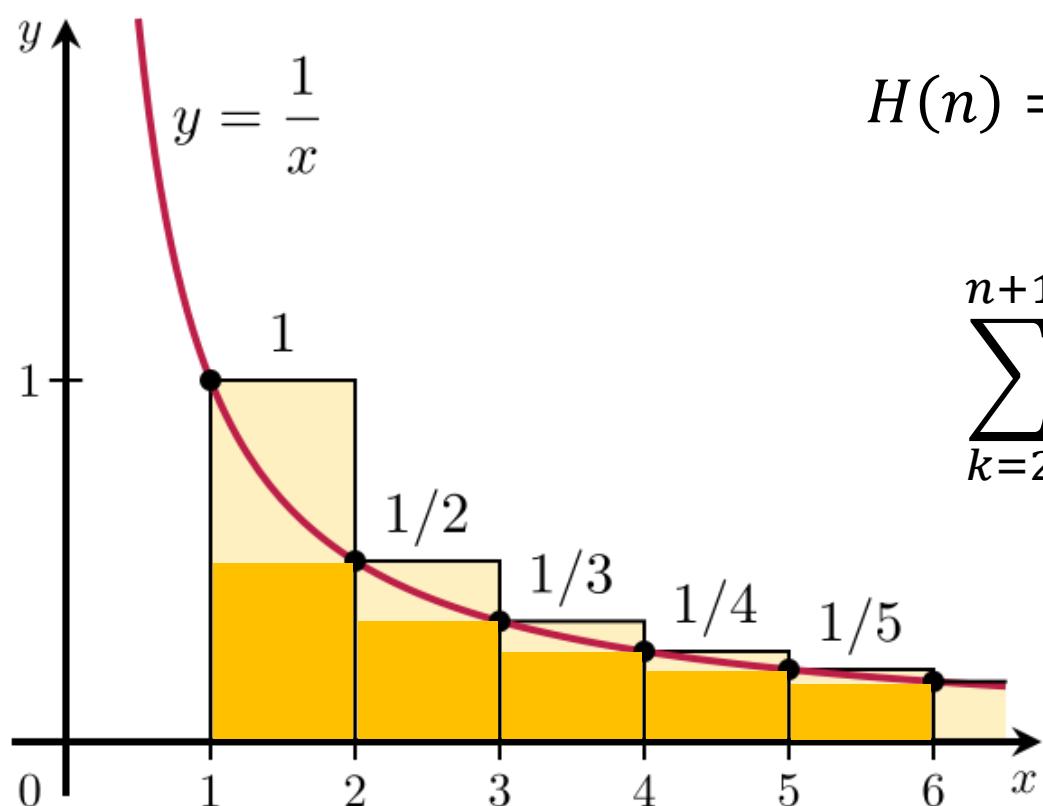
$$T(n) \leq 2 + \frac{n + 1}{n} \left(2 + \frac{n}{n - 1} \left(2 + \frac{n - 1}{n - 2} \left(\dots \frac{4}{3} \right) \right) \right)$$

$$T(n) \leq 2 \left(1 + \frac{n + 1}{n} + \frac{n + 1}{n} \frac{n}{n - 1} + \frac{n + 1}{n} \frac{n}{n - 1} \frac{n - 1}{n - 2} + \dots + \frac{n + 1}{n} \frac{n}{n - 1} \frac{n - 1}{n - 2} \dots \frac{4}{3} \right)$$

$$T(n) \leq 2 \left(1 + \frac{n + 1}{n} + \frac{n + 1}{n - 1} + \frac{n + 1}{n - 2} + \dots + \frac{n + 1}{3} \right) = 2(n + 1) \left(\frac{1}{n + 1} + \frac{1}{n} + \frac{1}{n - 1} + \dots + \frac{1}{3} \right)$$

$$T(n) \leq 2(n + 1)(H(n + 1) - 1.5) = \Theta(n \log n)$$

Harmonic series



$$H(n) = \sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

$$\sum_{k=2}^{n+1} \frac{1}{k} \leq \int_1^{n+1} \frac{dx}{x} \leq \sum_{k=1}^n \frac{1}{k}$$



$$\ln(n + 1)$$

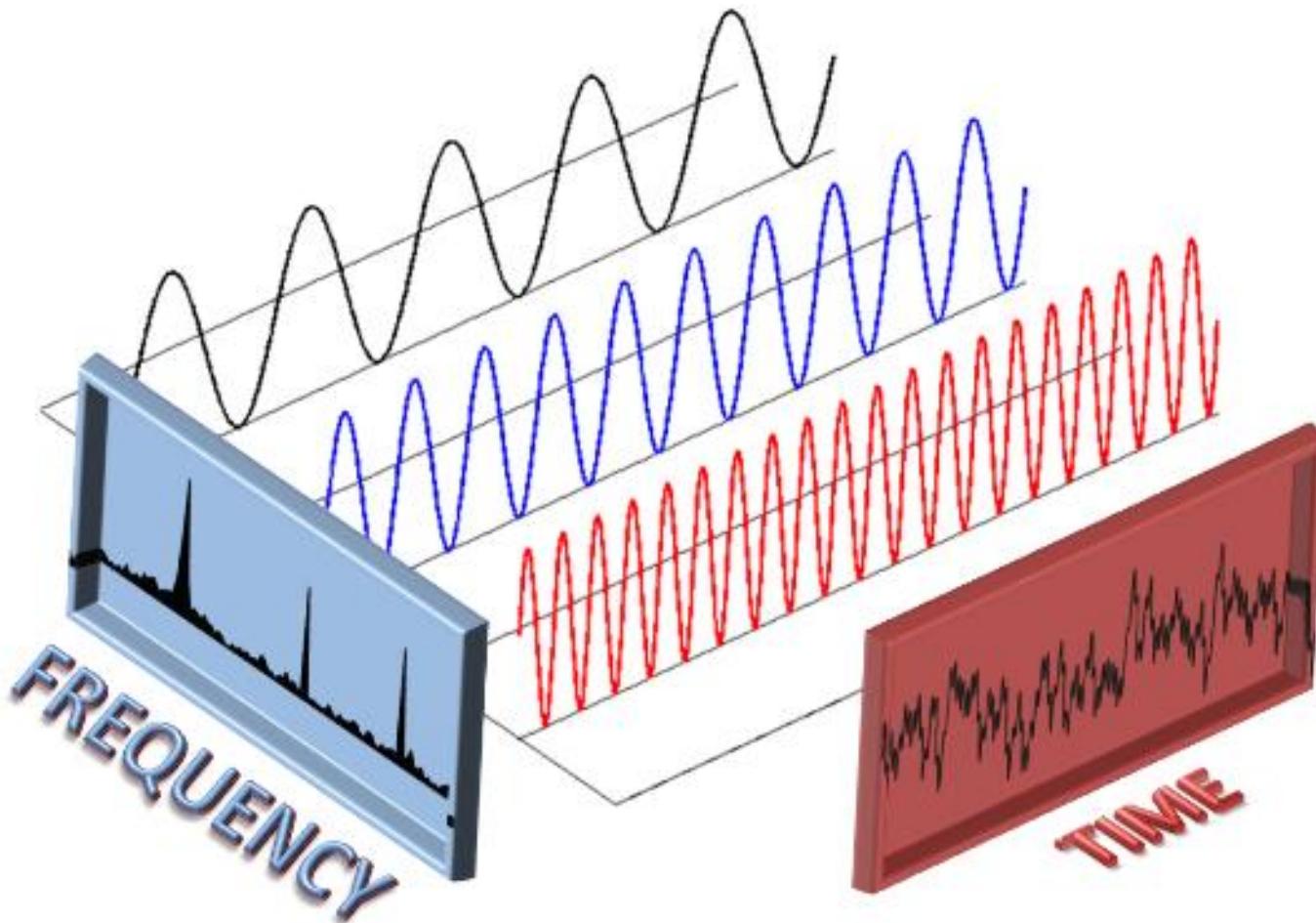
$$H(n) = \Theta(\log n)$$

Fast Fourier Transform



Jordi Cortadella and Jordi Petit
Department of Computer Science

Why Fourier Transform?



Polynomials: coefficient representation

- A polynomial is represented as a vector of coefficients $(a_0, a_1, \dots, a_{n-1})$:

$$A(x) = 2x^4 + x^2 - 4x + 3$$

$$A = (3, -4, 1, 0, 2)$$

- Addition: $O(n)$

$$A(x) + B(x) = (a_0 + b_0) + (a_1 + b_1)x + \dots + (a_{n-1} + b_{n-1})x^{n-1}$$

- Evaluation: $O(n)$ using Horner's method

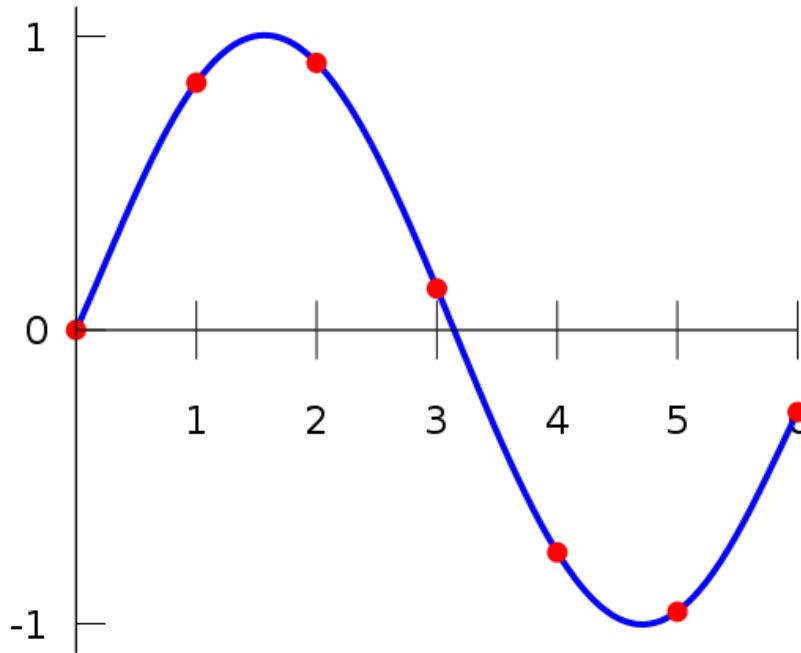
$$A(x) = a_0 + (x(a_1 + x(a_2 + \dots + x(a_{n-2} + x(a_{n-1})) \dots)))$$

- Multiplication: $O(n^2)$ using brute force

$$A(x) \cdot B(x) = \sum_{i=0}^{2n-2} c_i x^i, \quad \text{where } c_i = \sum_{j=0}^i a_j b_{i-j}$$

Polynomials: point-value representation

- **Fundamental Theorem (Gauss):** A degree n polynomial with complex coefficients has exactly n complex roots.
- **Corollary:** A degree $n - 1$ polynomial $A(x)$ is uniquely identified by its evaluation at n distinct values of x .



Polynomials: point-value representation

- A polynomial is represented as a set of pairs (x_i, y_i) :

$$A(x) = \{(x_0, y_0), \dots, (x_{n-1}, y_{n-1})\}$$

$$B(x) = \{(x_0, z_0), \dots, (x_{n-1}, z_{n-1})\}$$

- Addition: $O(n)$

$$A(x) + B(x) = \{(x_0, y_0 + z_0), \dots, (x_{n-1}, y_{n-1} + z_{n-1})\}$$

- Multiplication: $O(n)$, but with $2n - 1$ points

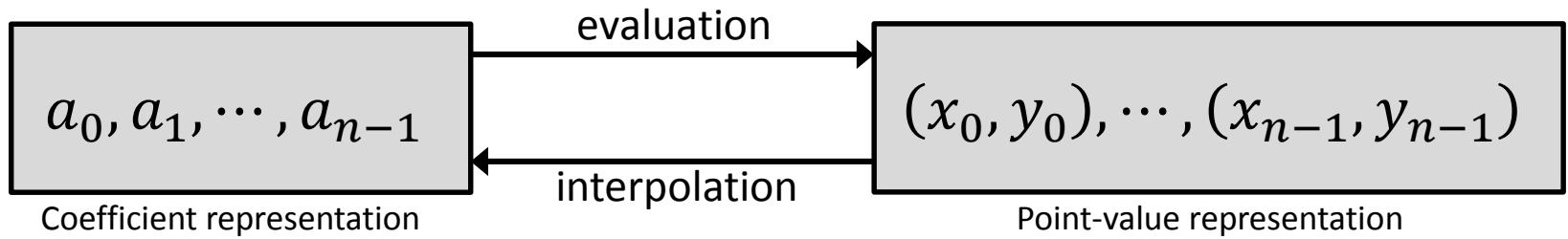
$$A(x) \cdot B(x) = \{(x_0, y_0 \cdot z_0), \dots, (x_{n-1}, y_{n-1} \cdot z_{n-1})\}$$

- Evaluation: $O(n^2)$ using Lagrange's formula

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

Conversion between both representations

representation	addition	multiplication	evaluation
coefficient	$O(n)$	$O(n^2)$	$O(n)$
point-value	$O(n)$	$O(n)$	$O(n^2)$



Could we have an efficient algorithm to move from coefficient to point-value representation and vice versa?

From coefficients to point-values

Given a polynomial $a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$,
evaluate it at n different points x_0, \dots, x_{n-1} :

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Runtime: $O(n^2)$ matrix-vector multiplication (apply Horner n times).

Evaluation by divide-and-conquer

- Credits: based on the intuitive explanation by Dasgupta, Papadimitriou and Vazirani, Algorithms, McGraw-Hill, 2008.
- We want to evaluate $A(x)$ at n different points. Let us choose them to be positive-negative pairs: $\pm x_0, \pm x_1, \dots, \pm x_{n/2-1}$
- The computations for $A(x_i)$ and $A(-x_i)$ overlap a lot.
- Split the polynomial into odd and even powers

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

- The terms in parenthesis are polynomials in x^2 :

$$A(x) = A_e(x^2) + xA_o(x^2)$$

Evaluation by divide-and-conquer

- The calculations needed for $A(x_i)$ can be reused for computing $A(-x_i)$.

$$A(x_i) = A_e(x_i^2) + x_i A_o(x_i^2)$$

$$A(-x_i) = A_e(x_i^2) - x_i A_o(x_i^2)$$

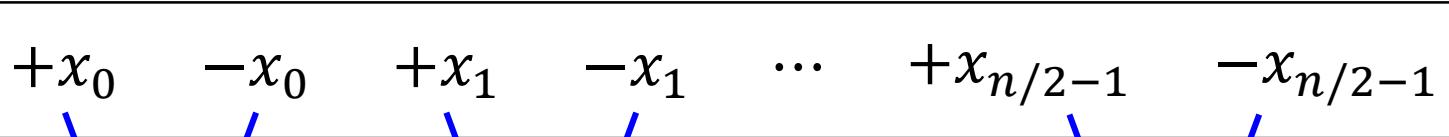
- Evaluating $A(x)$ at n paired points

$$\pm x_0, \pm x_1, \dots, \pm x_{n/2-1}$$

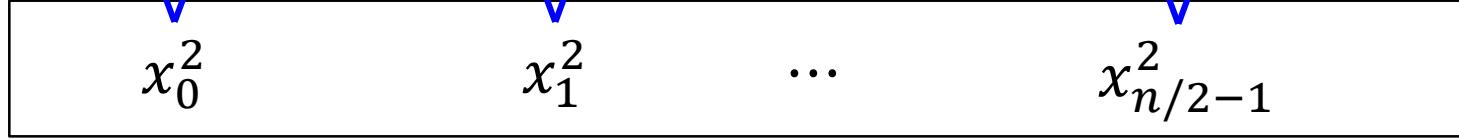
reduces to evaluating $A_e(x)$ and $A_o(x)$ at just $n/2$ points: $x_0^2, \dots, x_{n/2-1}^2$

Evaluation by divide-and-conquer

Evaluate: $A(x)$
degree $\leq n - 1$



Evaluate:
 $A_e(x)$ and $A_o(x)$
degree $\leq n/2 - 1$



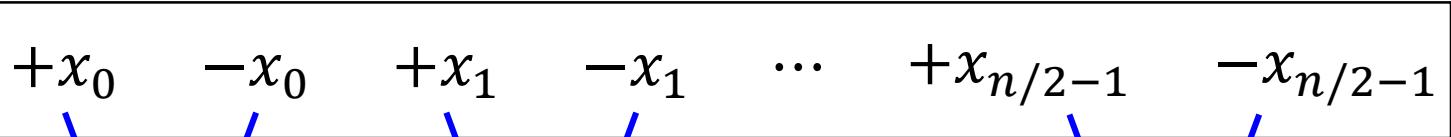
If we could recurse, we would get a running time:

$$T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)$$

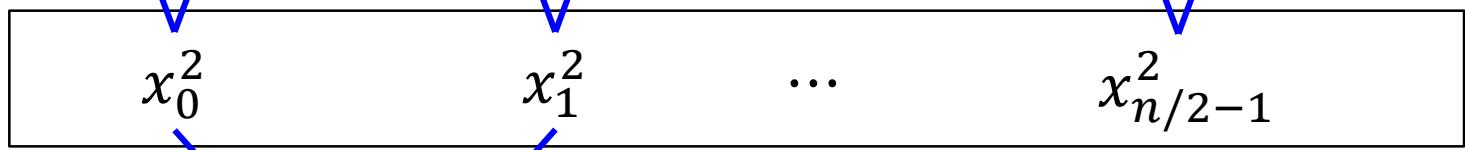
But can we recurse?

Evaluation by divide-and-conquer

Evaluate: $A(x)$
degree $\leq n - 1$



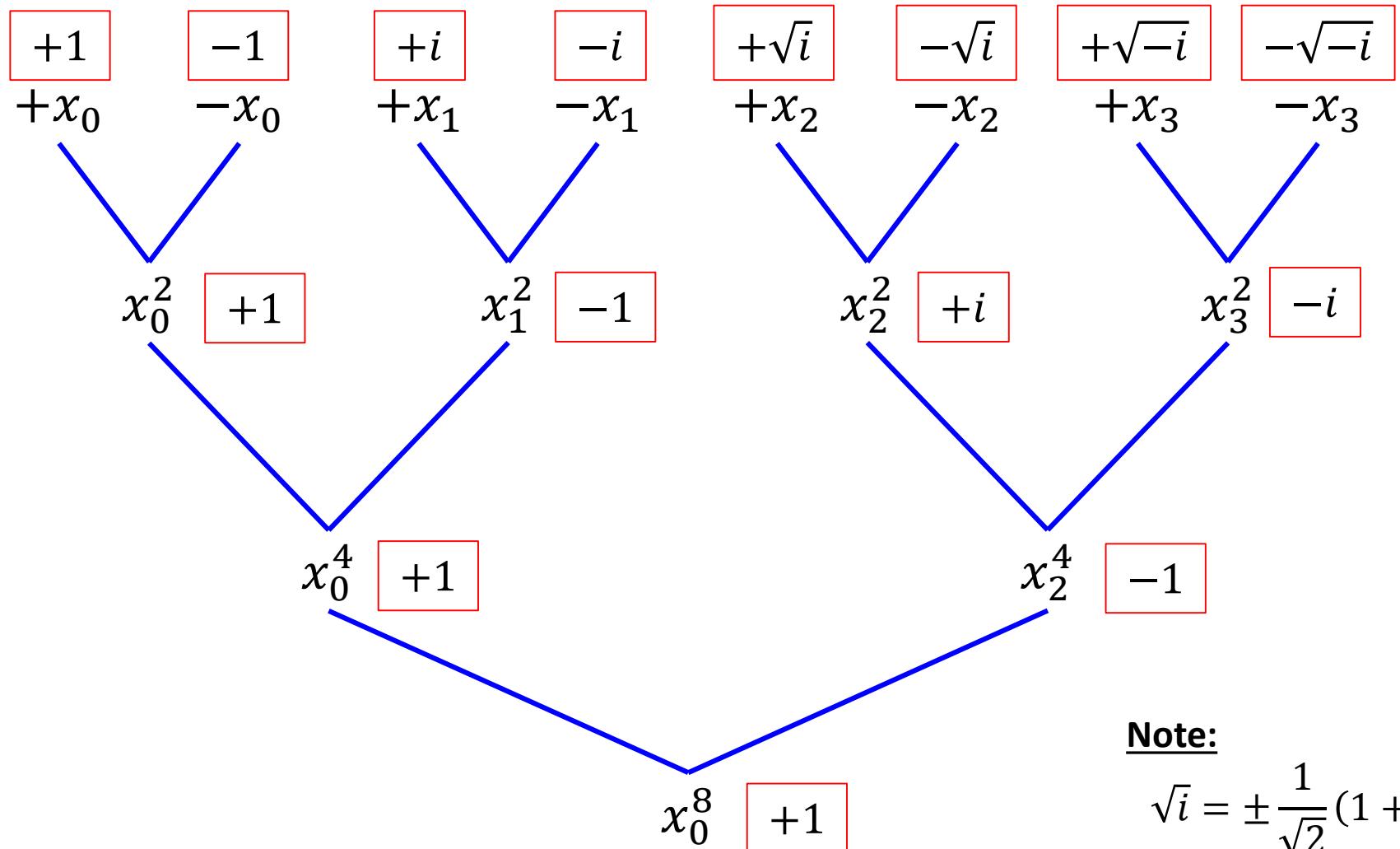
Evaluate:
 $A_e(x)$ and $A_o(x)$
degree $\leq n/2 - 1$



The problem: ?

We need x_0^2 and x_1^2 to be a plus-minus pair.
But a square cannot be negative !

Evaluation by divide-and-conquer



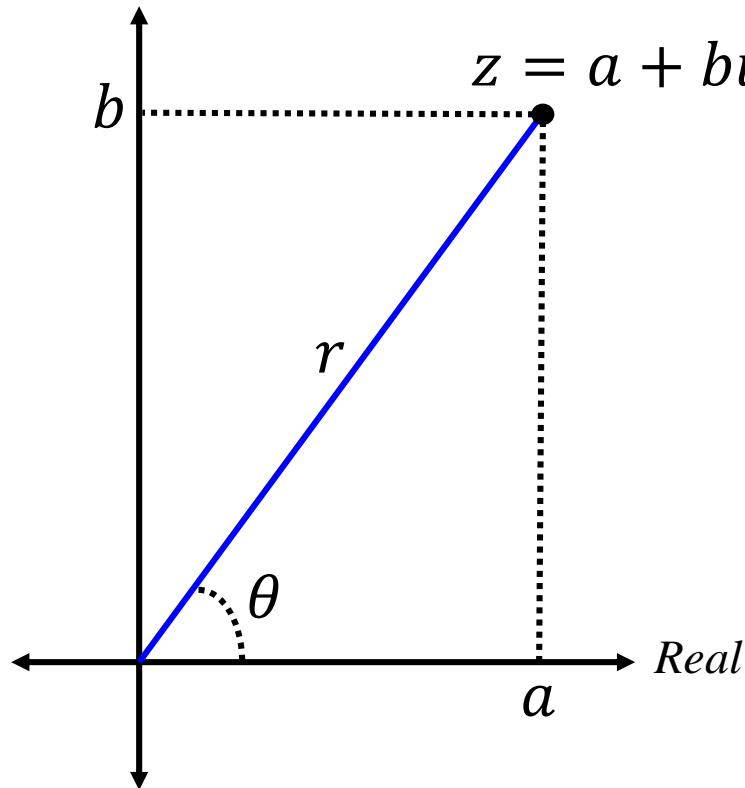
Note:

$$\sqrt{i} = \pm \frac{1}{\sqrt{2}}(1 + i)$$

$$\sqrt{-i} = \pm \frac{1}{\sqrt{2}}(1 - i)$$

Complex numbers: review

Imaginary



Some examples:

$$z = r(\cos \theta + i \sin \theta) = re^{i\theta}$$

Polar coordinates: (r, θ)

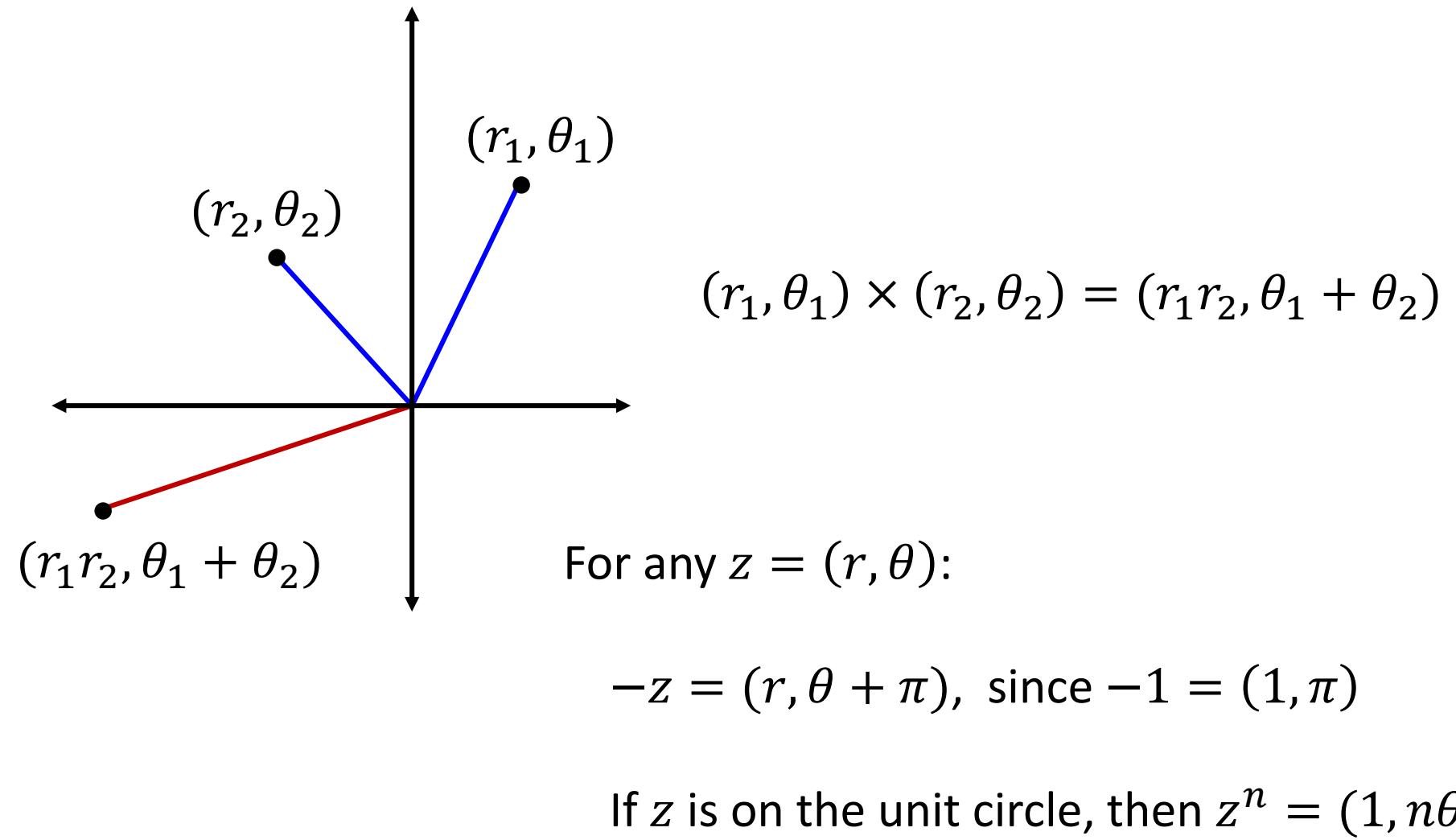
$$\text{Length: } r = \sqrt{a^2 + b^2}$$

$$\text{Angle } \theta \in [0, 2\pi): \cos \theta = \frac{a}{r}, \sin \theta = \frac{b}{r}$$

θ can always be reduced modulo 2π

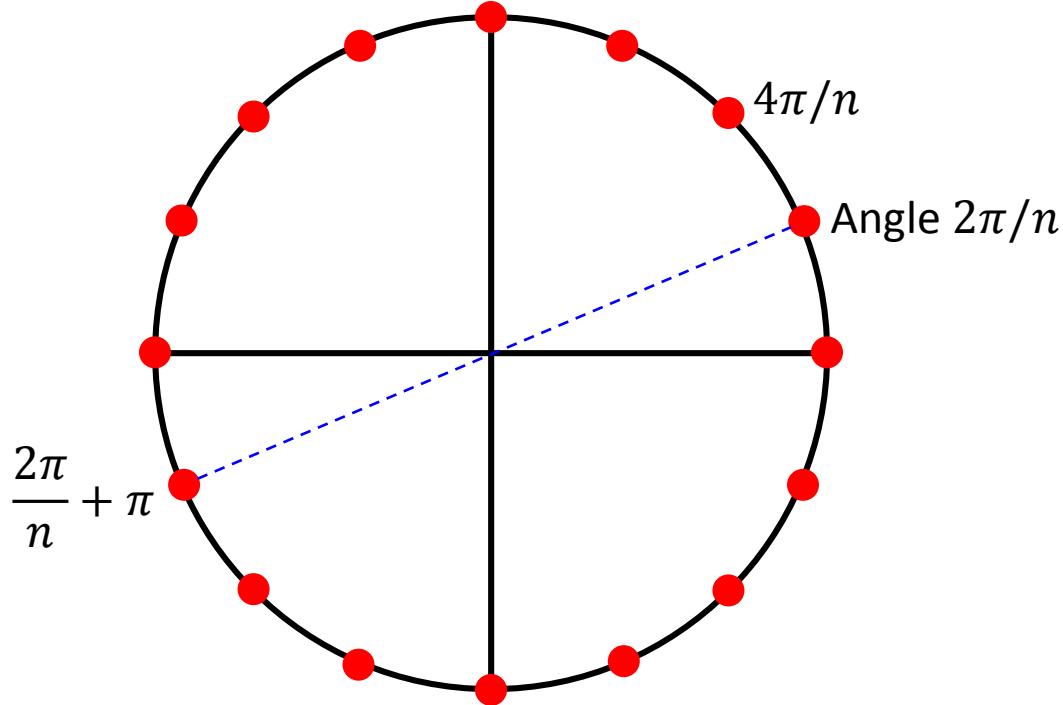
Number	-1	i	$5 + 5i$
Polar coords	$(1, \pi)$	$(1, \pi/2)$	$(5\sqrt{2}, \pi/4)$

Complex numbers: multiplication



Complex numbers: the n th roots of unity

Solutions to the equation $z^n = 1$
 $(n = 16)$

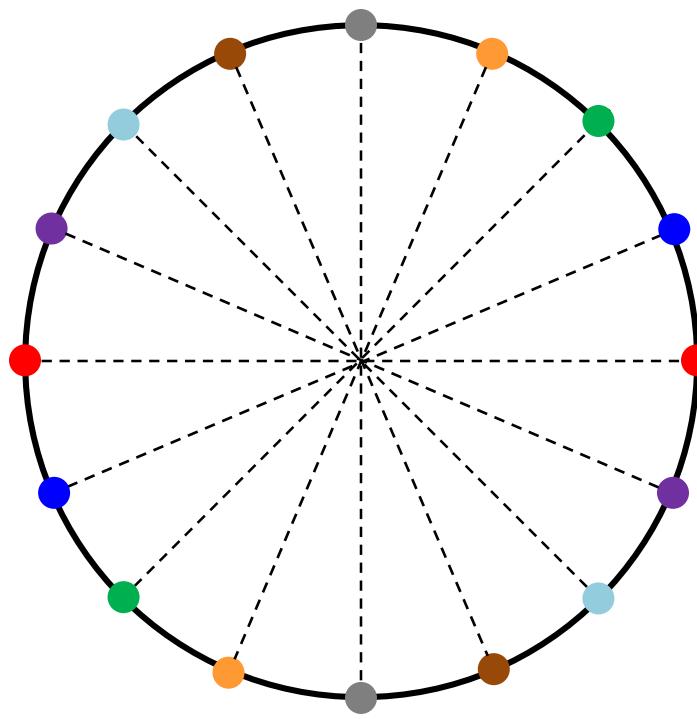


Solutions are $z = (1, \theta)$,
for θ a multiple of $2\pi/n$

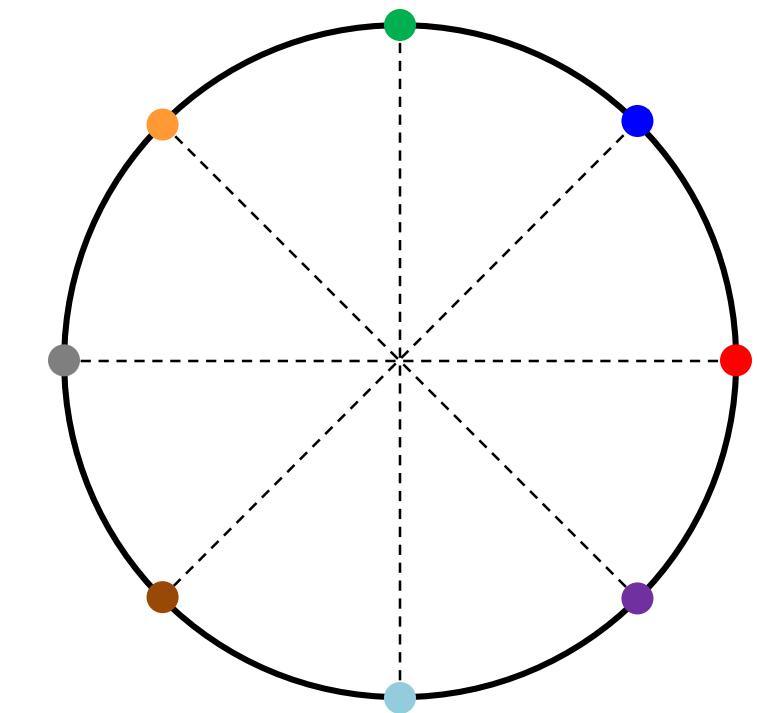
All roots are plus-minus paired:

$$-(1, \theta) = (1, \theta + \pi)$$

Divide-and-conquer step

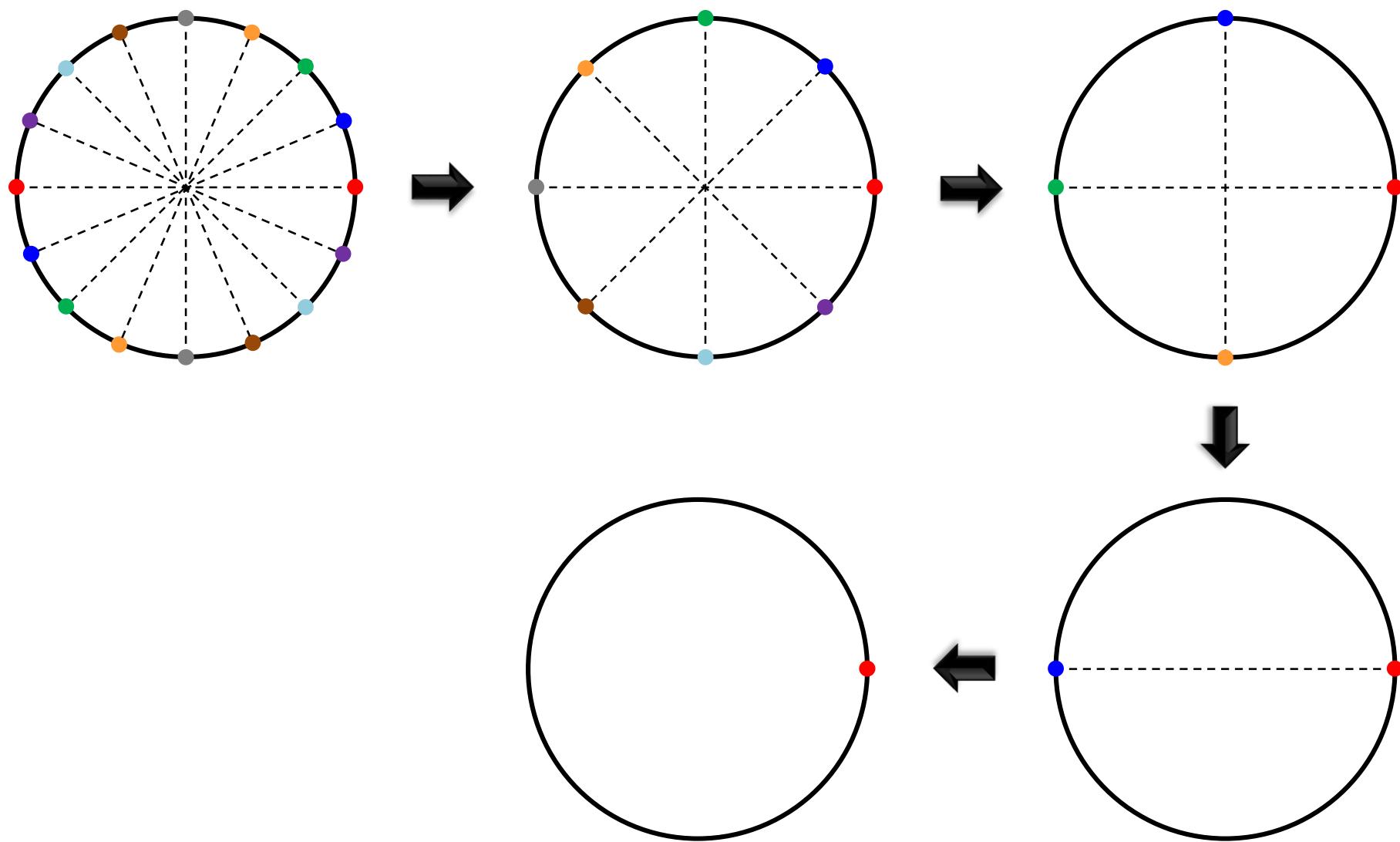


Evaluate $A(x)$ at n th roots of unity

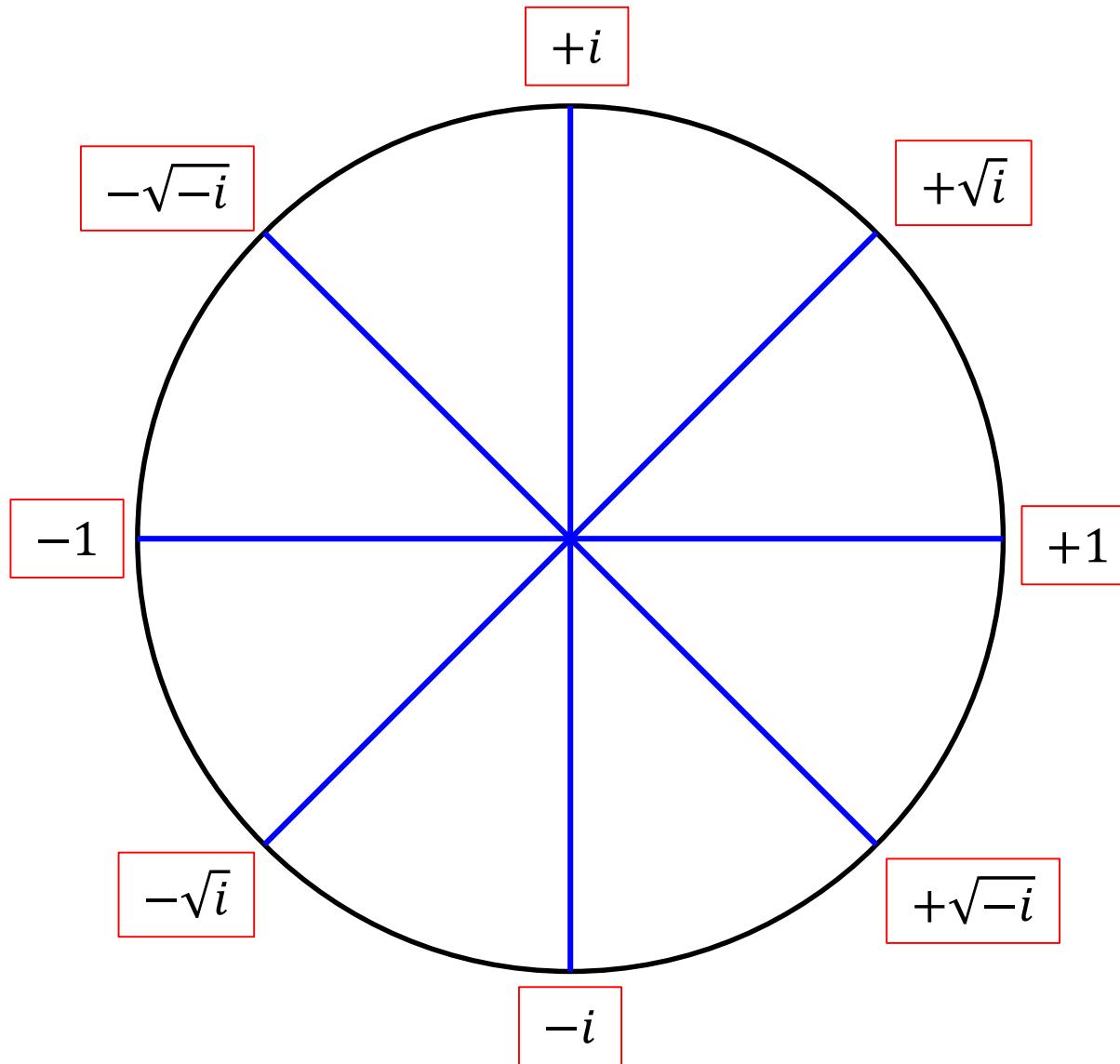


Evaluate $A_e(x)$ and $A_o(x)$ at $(n/2)$ nd roots of unity

Divide-and-conquer steps



Roots of unity for $n = 8$



Fast Fourier Transform

function FFT(A, ω)

Inputs: $A = (a_0, a_1, \dots, a_{n-1})$, for n a power of 2
 ω : A primitive n th root of unity

Output: $(A(1), A(\omega), A(\omega^2), \dots, A(\omega^{n-1}))$

if $\omega=1$: **return** A

$$(A_e(\omega^0), A_e(\omega^2), \dots, A_e(\omega^{n-2})) = \text{FFT}(A_e, \omega^2)$$

$$(A_o(\omega^0), A_o(\omega^2), \dots, A_o(\omega^{n-2})) = \text{FFT}(A_o, \omega^2)$$

for $k = 0$ **to** $n - 1$:

$$A(\omega^k) = A_e(\omega^{2k}) + \omega^k A_o(\omega^{2k})$$

return $(A(1), A(\omega), A(\omega^2), \dots, A(\omega^{n-1}))$

Fast Fourier Transform: example with $n = 4$

```
function FFT(( $a_0, a_1, a_2, a_3$ ),  $\omega$ )
```

$$(A_e(\omega^0), A_e(\omega^2)) = \text{FFT}((a_0, a_2), \omega^2)$$

$$(A_o(\omega^0), A_o(\omega^2)) = \text{FFT}((a_1, a_3), \omega^2)$$

$$\begin{aligned}\omega^0 &= 1 \\ \omega^1 &= i \\ \omega^2 &= -1 \\ \omega^3 &= -i\end{aligned}$$

$$A(\omega^0) = A_e(\omega^0) + \omega^0 A_o(\omega^0)$$

$$A(\omega^1) = A_e(\omega^2) + \omega^1 A_o(\omega^2)$$

$$A(\omega^2) = A_e(\omega^4) + \omega^2 A_o(\omega^4) = A_e(\omega^0) - \omega^0 A_o(\omega^0)$$

$$A(\omega^3) = A_e(\omega^6) + \omega^3 A_o(\omega^6) = A_e(\omega^2) - \omega^1 A_o(\omega^2)$$

```
return (A(1), A( $\omega$ ), A( $\omega^2$ ), A( $\omega^3$ ))
```

Fast Fourier Transform

```
function FFT( $a, \omega$ )
```

Inputs: $a = (a_0, a_1, \dots, a_{n-1})$, for n a power of 2
 ω : A primitive n th root of unity

Output: $(a(1), a(\omega), a(\omega^2), \dots, a(\omega^{n-1}))$

```
if  $\omega=1$ : return  $a$ 
```

```
 $(s_0, s_1, \dots, s_{n/2-1}) = \text{FFT}((a_0, a_2, \dots, a_{n-2}), \omega^2)$ 
```

```
 $(s'_0, s'_1, \dots, s'_{n/2-1}) = \text{FFT}((a_1, a_3, \dots, a_{n-1}), \omega^2)$ 
```

```
for  $k = 0$  to  $n/2 - 1$ :
```

$$r_k = s_k + \omega^k s'_k$$

$$r_{k+n/2} = s_k - \omega^k s'_k$$

```
return  $(r_0, r_1, \dots, r_{n-1})$ 
```

FFT: asymptotic complexity

- The runtime of the FFT can be expressed as:

$$T(n) = 2 \cdot T(n/2) + O(n)$$

- Using the master theorem we conclude:

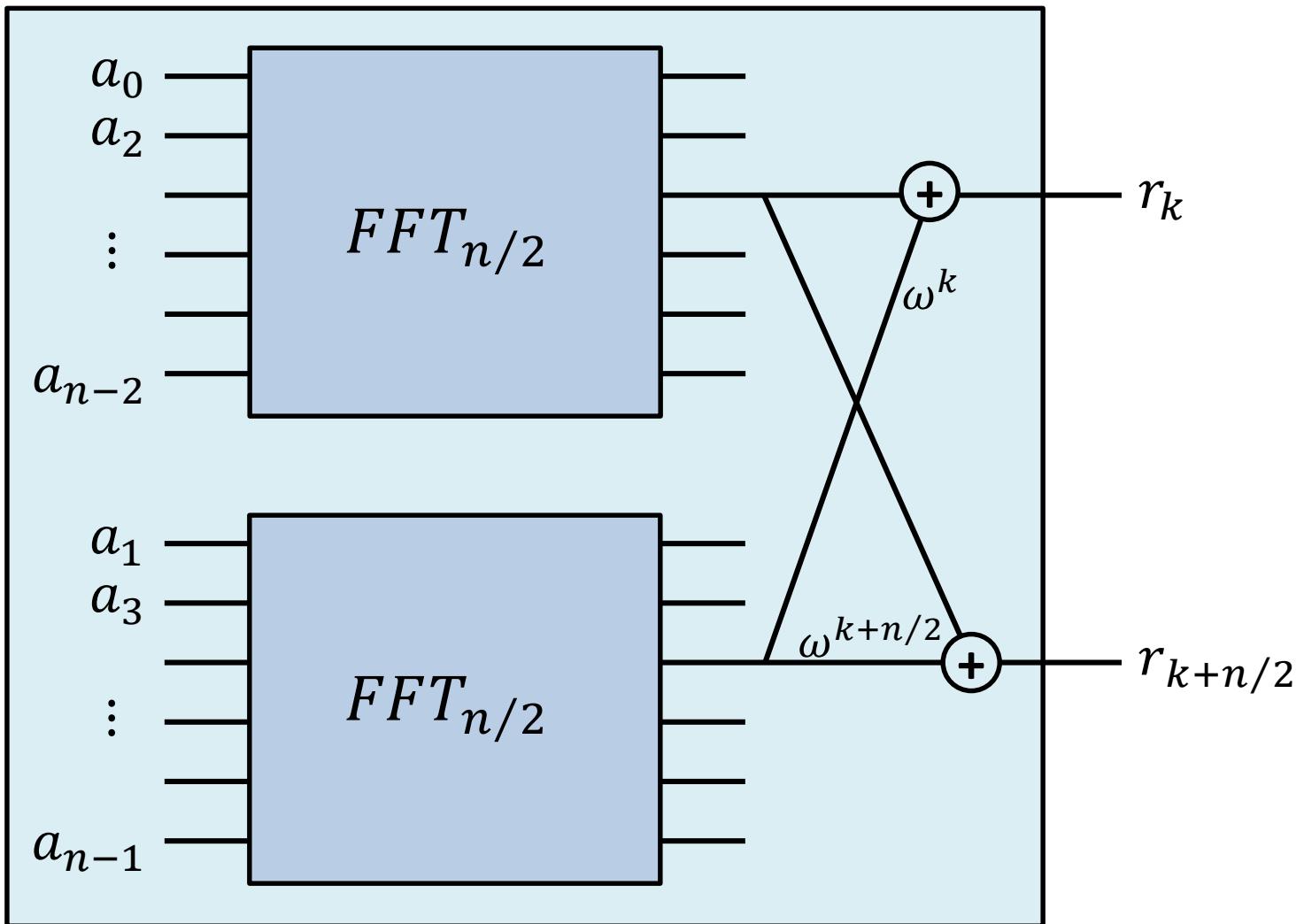
$$\text{Runtime FFT}(n) = O(n \log n)$$

- Gilbert Strang (MIT, 1994):

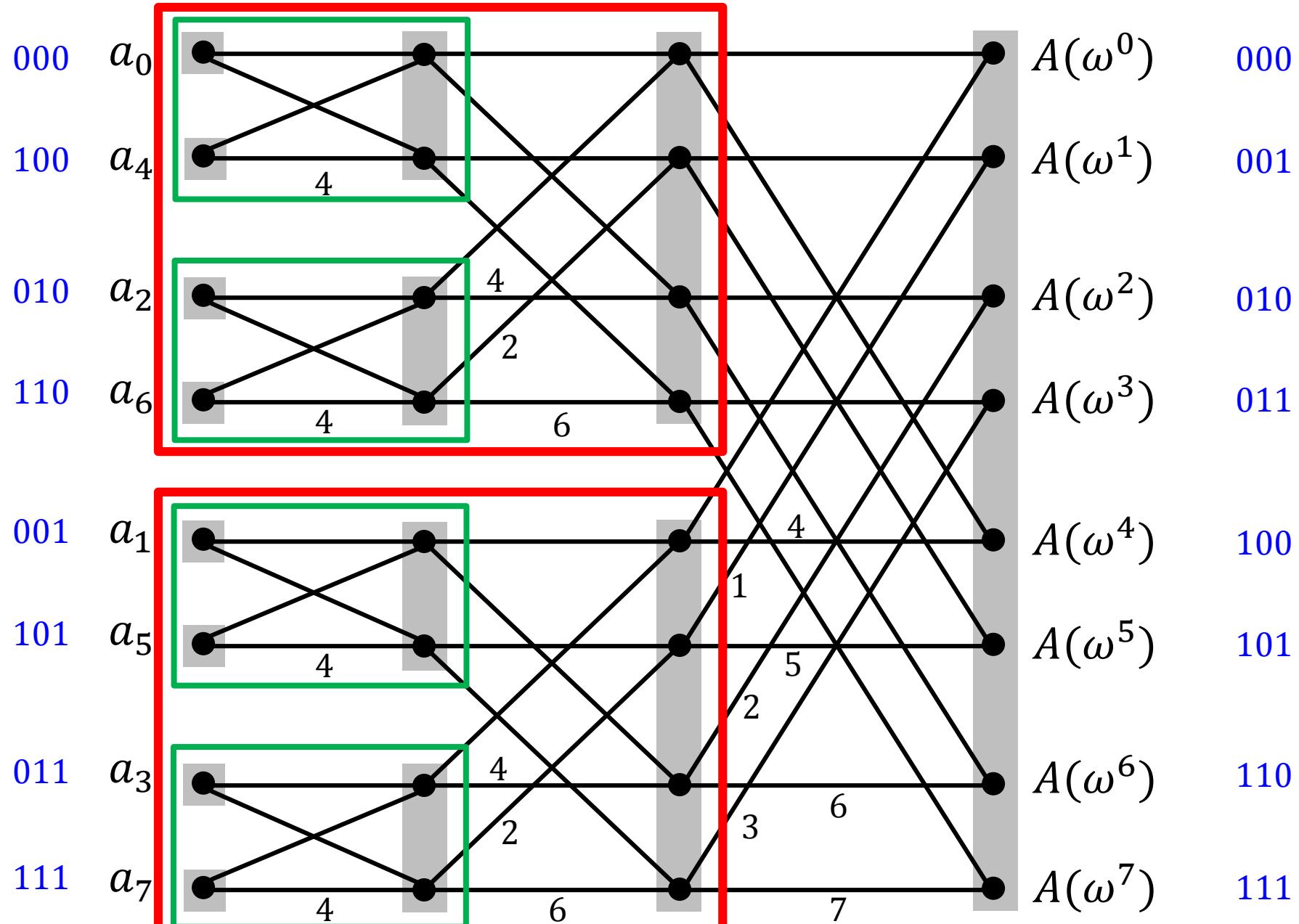
“the most important numerical algorithm
of our lifetime”.

- **Reference:** Cooley, James W., and John W. Tukey, 1965,
“An algorithm for the machine calculation of complex
Fourier series,” Math. Comput. 19: 297-301.

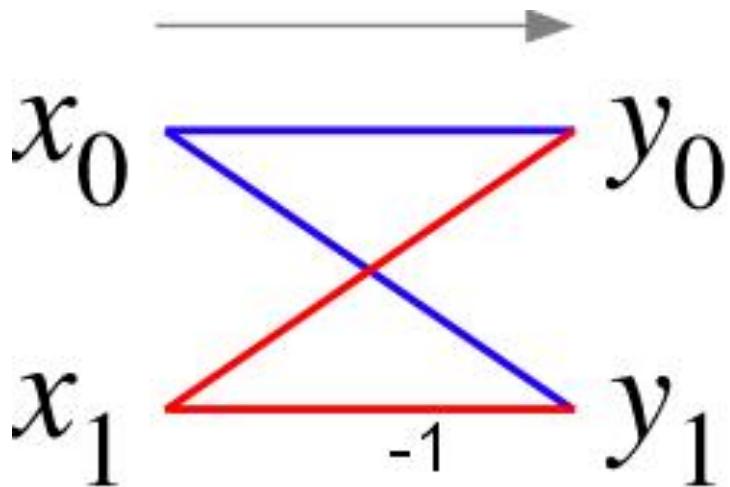
Unfolding the FFT



Unfolding the FFT (butterfly diagram)



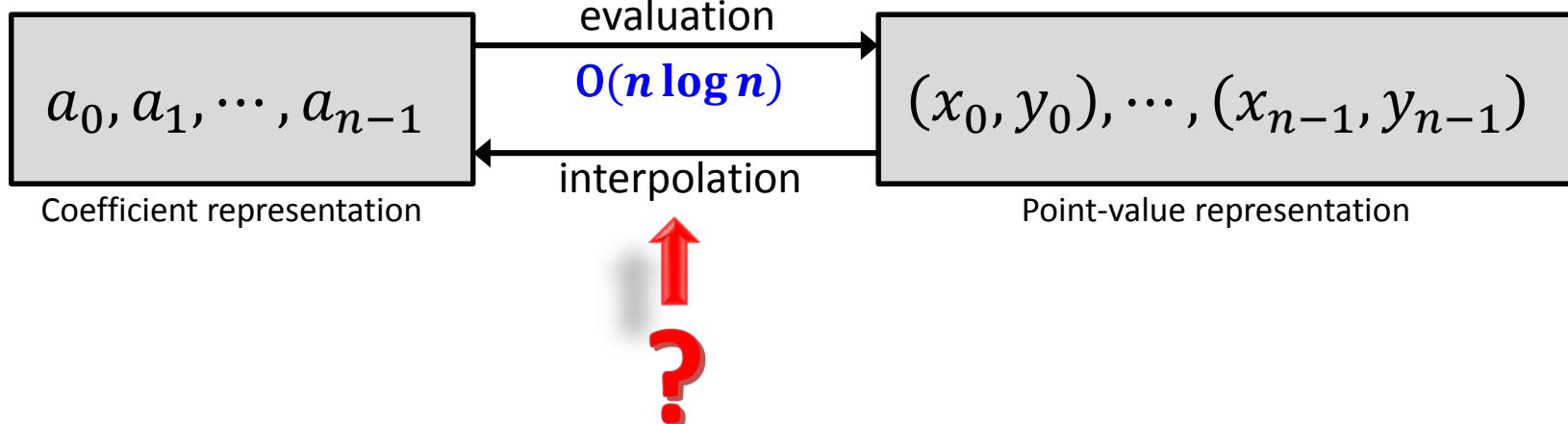
Why is it called a butterfly diagram?



Conversion between both representations

representation	addition	multiplication	evaluation
coefficient	$O(n)$	$O(n^2)$	$O(n)$
point-value	$O(n)$	$O(n)$	$O(n^2)$

$$\langle \text{values} \rangle = \text{FFT}(\langle \text{coefficients} \rangle, \omega)$$



From point-values to coefficients

The Fast Fourier Transform computes:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

where $\omega = e^{2\pi i/n} = (1, 2\pi/n)$.

Let us call $F_n(\omega)$ the Fourier matrix. Thus,

$$y = F_n(\omega) \cdot a$$

How about if we know y and we want to obtain a ?

From point-values to coefficients

$$\begin{aligned} y &= F_n(\omega) \cdot a \\ &\Downarrow \\ [F_n(\omega)]^{-1} \cdot y &= a \end{aligned}$$

$F_n(\omega)$ is a unitary matrix and has the following property:

$$[F_n(\omega)]^{-1} = \frac{1}{n} \cdot F_n(\omega^{-1})$$

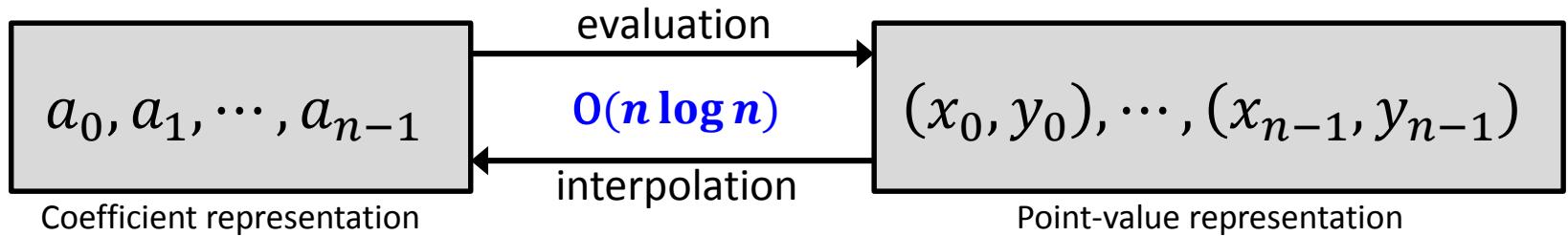
and also

If ω is a primitive n th root of unit,
then $1/\omega$ is also a primitive n th root of unit.

Conversion between both representations

representation	addition	multiplication	evaluation
coefficient	$O(n)$	$O(n^2)$	$O(n)$
point-value	$O(n)$	$O(n)$	$O(n^2)$

$$\langle \text{values} \rangle = \text{FFT}(\langle \text{coefficients} \rangle, \omega)$$



$$\langle \text{coefficients} \rangle = \frac{1}{n} \text{FFT}(\langle \text{values} \rangle, \omega^{-1})$$

Polynomial multiplication

Input: Coefficients of two polynomials $A(x)$ and $B(x)$, of degree d_A and d_B , respectively. Let $d = d_A + d_B$.

Output: The product $C = A \cdot B$.

1. Selection:

- Pick $\omega = (1, 2\pi/n)$, such that $n \geq d + 1$ and n is a power of two.

2. Evaluation (FFT):

- Compute $A(1), A(\omega), A(\omega^2), \dots, A(\omega^{n-1})$.
- Compute $B(1), B(\omega), B(\omega^2), \dots, B(\omega^{n-1})$.

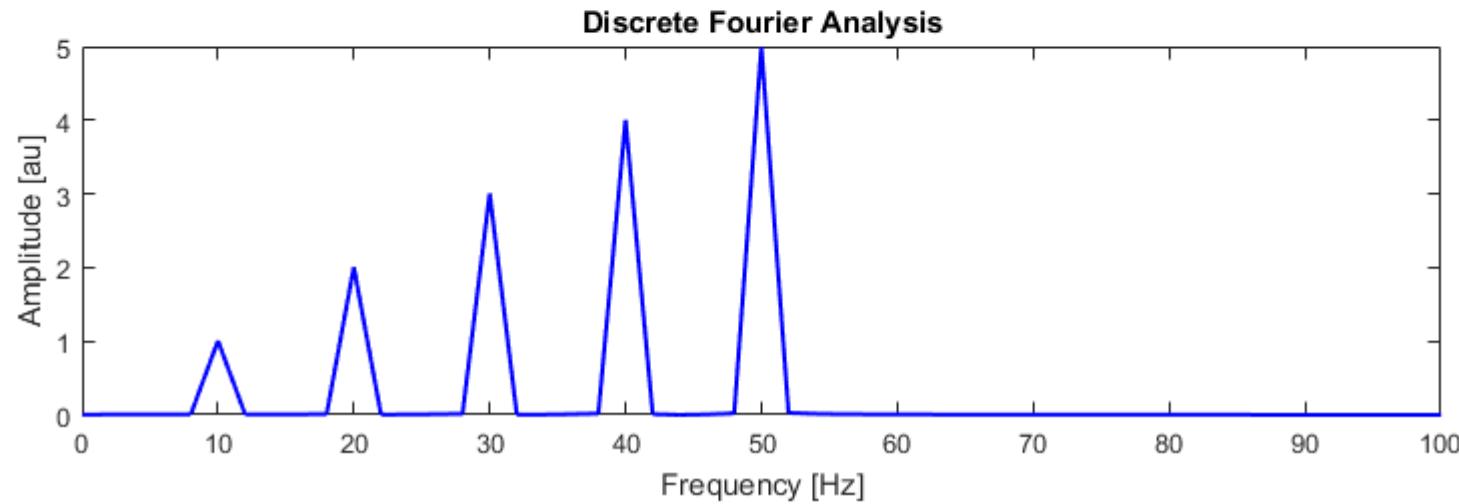
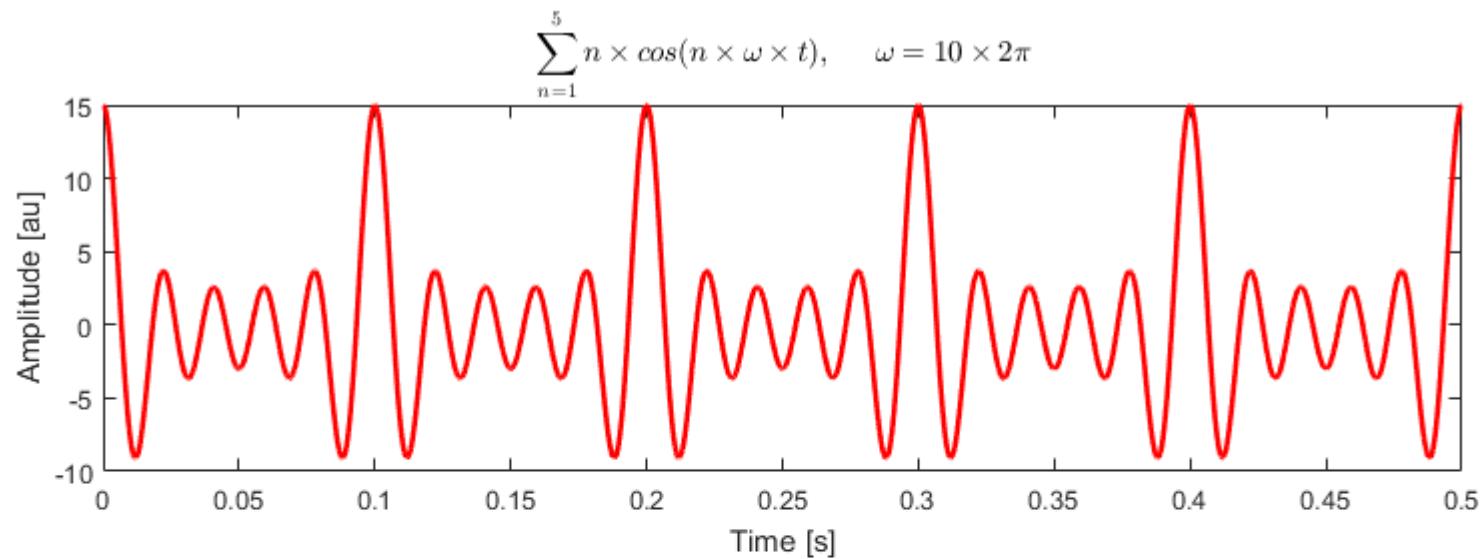
3. Multiplication:

- Compute $C(\omega^k) = A(\omega^k) \cdot B(\omega^k)$, for all $k = 0, \dots, n - 1$.

4. Interpolation (inverse FFT):

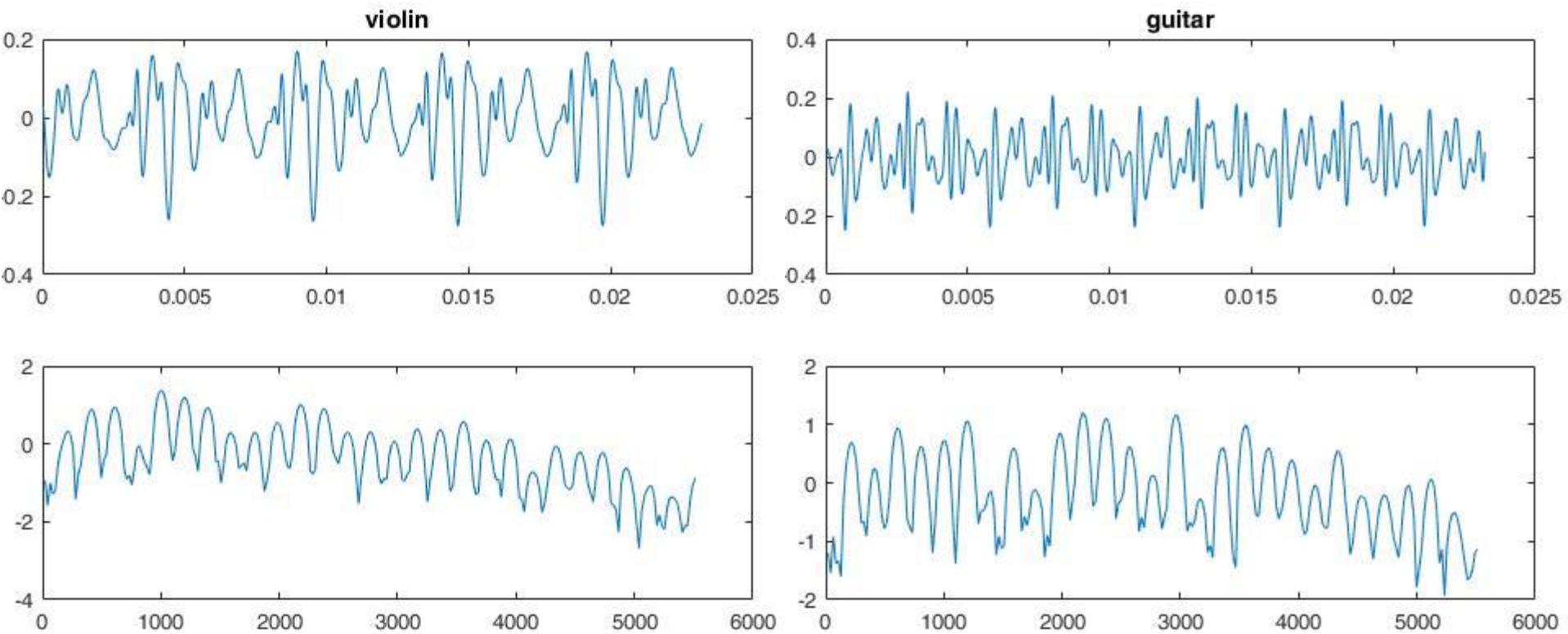
- Recover $C(x) = c_0 + c_1x + c_2x^2 + \dots + c_dx^d$.

FFT application in Signal Processing



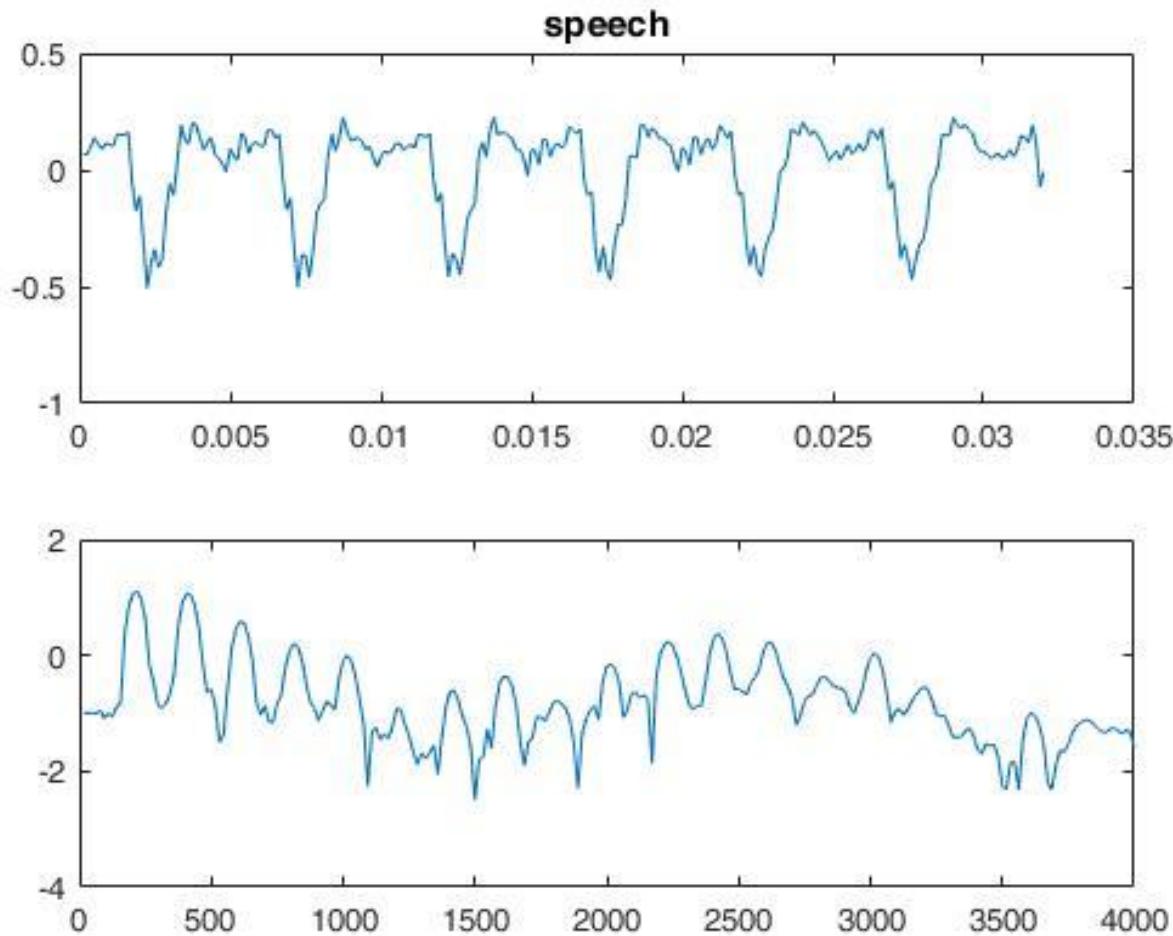
Converting a signal: time domain \leftrightarrow frequency domain

Distinguishing instruments



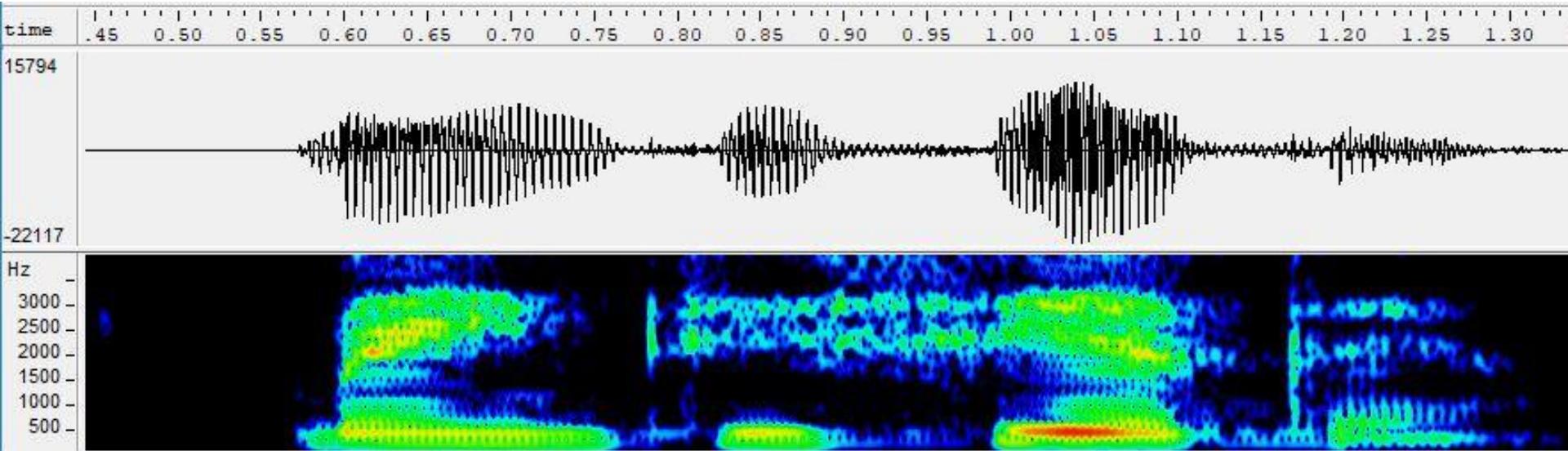
Same note (frequency).
Different timbre (spectral envelope).

Speech



Tone: distance between sidelobes (vocal cords).
Sound: spectral envelope.

Spectrogram

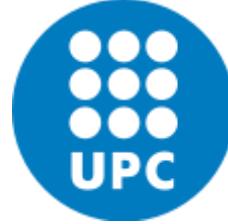


Pronouncing “veintisiete”

Exercises

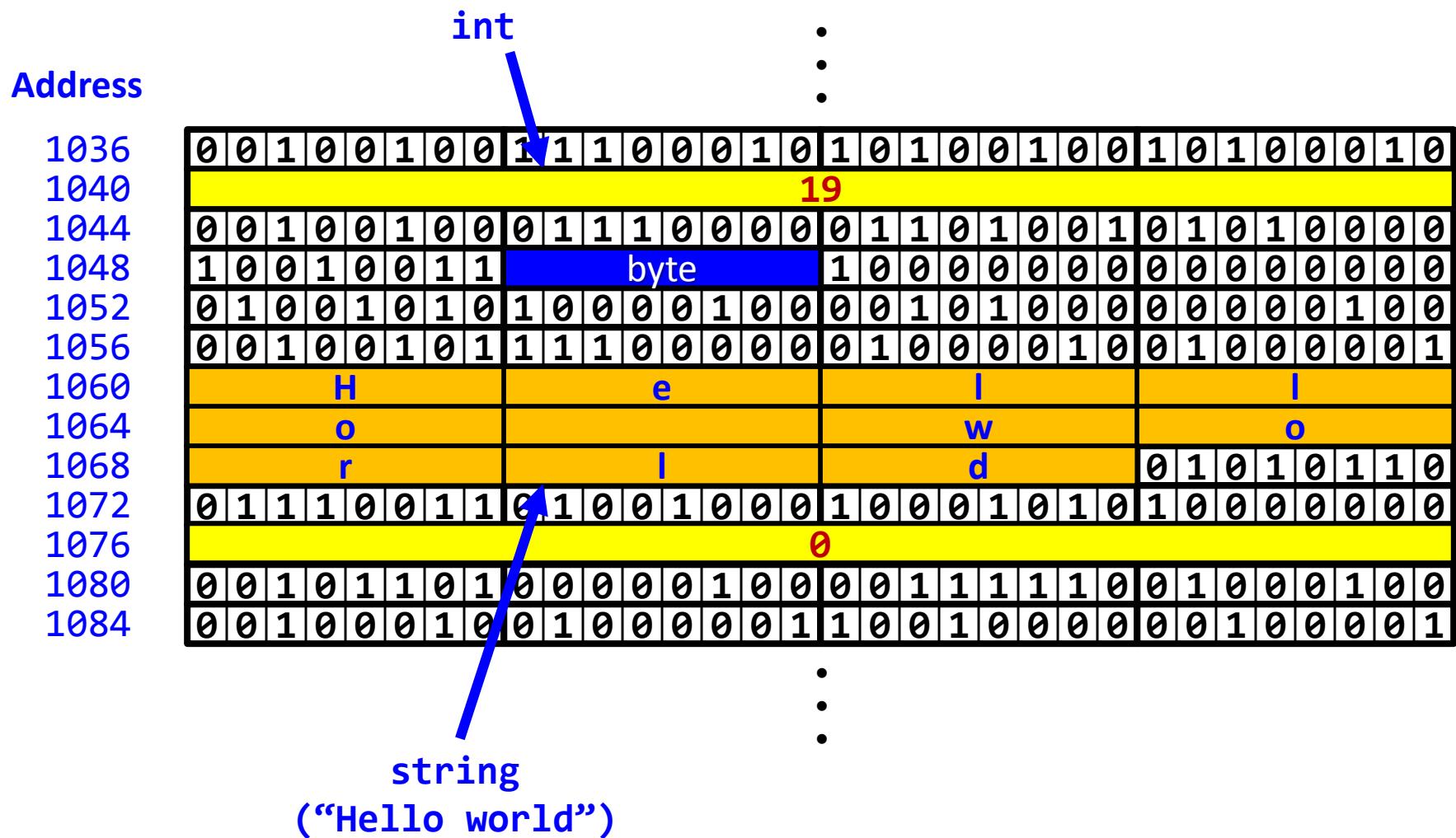
1. Consider the polynomials $1 + x - 2x^2 + x^3$ and $-1 + x^2$:
 - Choose an appropriate power of two to execute the FFT for the polynomial multiplication. Find the value of ω .
 - Give the result of the FFT for $x^2 - 1$ (no need to execute the FFT).
2. Consider the polynomials $-1 + 2x + x^2$ and $1 + 2x$:
 - Choose an appropriate power of two to execute the FFT. Find the value of ω .
 - Calculate their point-value representation using the FFT (execute the FFT algorithm manually).
 - Calculate the product of the point-value representations.
 - Execute the inverse FFT to obtain the coefficients of the product.

Memory management



Jordi Cortadella and Jordi Petit
Department of Computer Science

The memory



Pointers

- Given a type T , T^* is another type called “pointer to T”. It holds the memory address of an object of type T.
- Examples:

```
int* pi;  
myClass* pc;
```

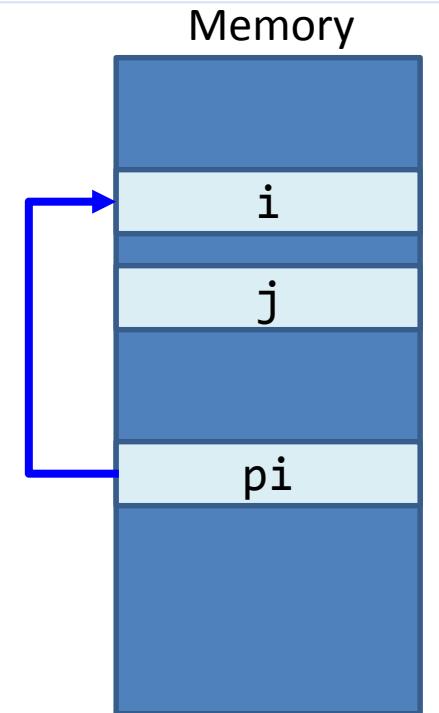
- Pointers are often used to manage the memory space of dynamic data structures.
- In some cases, it might be convenient to obtain the address of a variable during runtime.

Pointers

- Address of a variable (reference operator &):

```
int i;  
int* pi = &i;
```

// &i means: “the address of i”



- Access to the variable (dereference operator *)

```
int j = *pi;
```

// j gets the value of the variable pointed by pi

- Null pointer (points to nowhere; useful for initialization)

```
int* pi = nullptr;
```

Dynamic Object Creation/Destruction

- The *new* operator returns a pointer to a newly created object:

```
myClass* c = new myClass( );  
myClass* c = new myClass{ }; // C++11  
myClass* c = new myClass;
```

- The *delete* operator destroys an object through a pointer (deallocates the memory space associated to the object):

```
delete c; // c must be a pointer
```

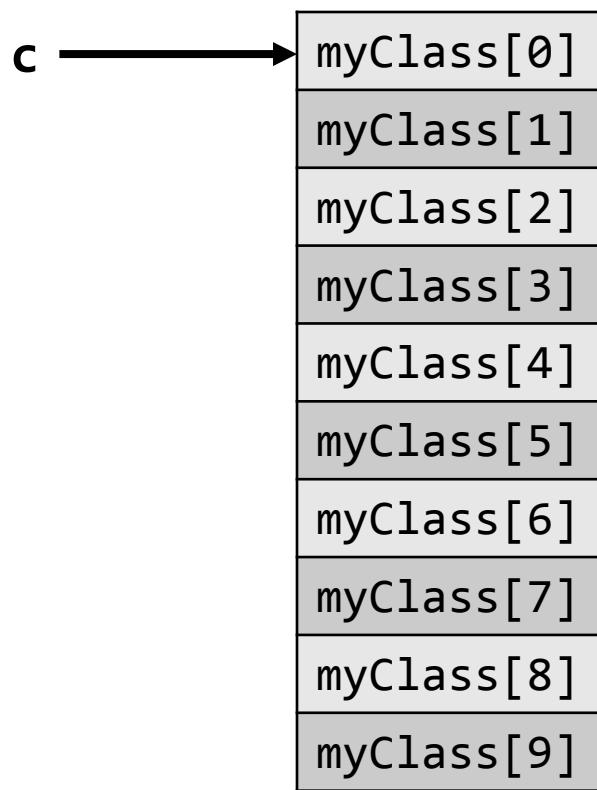
Access to members through a pointer

The members of a class can be accessed through a pointer to the class via the `->` operator:

```
vector<int>* vp = new vector<int> (100);  
...  
vp->push_back(5);  
...  
int n = vp->size();
```

Allocating/deallocating arrays

new and *delete* can also create/destroy arrays of objects



```
// c is a pointer to an  
// array of objects  
myClass* c = new myClass[10];  
  
// deallocating the array  
delete [] c;
```

References

- A reference defines a new name for an existing value (a synonym).
- References are not pointers, although they are usually implemented as pointers (memory references).
- Typical uses:
 - Avoiding copies (e.g., parameter passing)
 - Aliasing long names
 - Range *for* loops

References: examples

```
auto & r = x[getIndex(a, b)].val;  
...  
r.defineValue(n); // avoids a long name for r  
...  
  
// avoids a copy of a large data structure  
bigVector & V = myMatrix.getRow(i);  
...  
  
// An alternative for the following loop:  
// for (int i =0; i < a.size(); ++i) ++a[i];  
  
for (auto x: arr) ++x; // does not work (why?)  
  
for (auto & x: arr) ++x; // it works!
```

Pointers vs. references

- A *pointer* holds the memory address of a variable. A *reference* is an alias (another name) for an existing variable.
- In practice, references are implemented as pointers.
- A pointer can be re-assigned any number of times, whereas a reference can only be assigned at initialization.
- Pointers can be NULL. References always refer to an object.
- You can have pointers to pointers. You cannot have references to references.
- You can use pointer arithmetic (e.g., `&object+3`). Reference arithmetic does not exist.

The Vector class

(an approximation to the STL vector class)

The Vector class

- The natural replacement of C-style arrays.
- Main advantages:
 - It can dynamically grow and shrink.
 - It is a *template* class (can handle any type T)
 - No need to take care of the allocated memory.
 - Data is allocated in a contiguous memory area.
- We will implement a **Vector** class, a simplified version of STL's vector.
- Based on Weiss' book (4th edition), see Chapter 3.4.

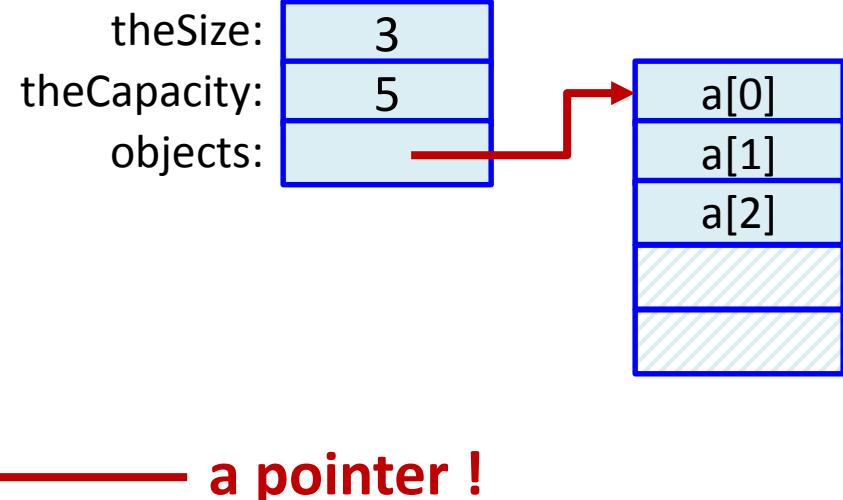
The Vector class

```
template <typename Object>
class Vector {
public:
    ...
private:
    ...
};
```

- What is a class template?
 - A generic abstract class that can handle various datatypes.
 - The template parameters determine the genericity of the class.
- Example of declarations:
 - `Vector<int> V;`
 - `Vector<polygon> Vp;`
 - `Vector<Vector<double>> M;`

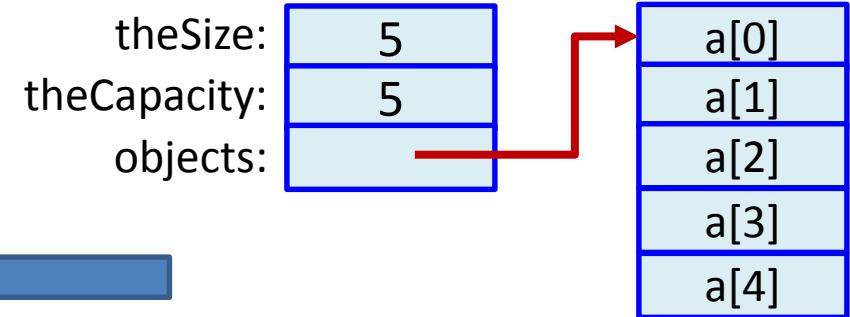
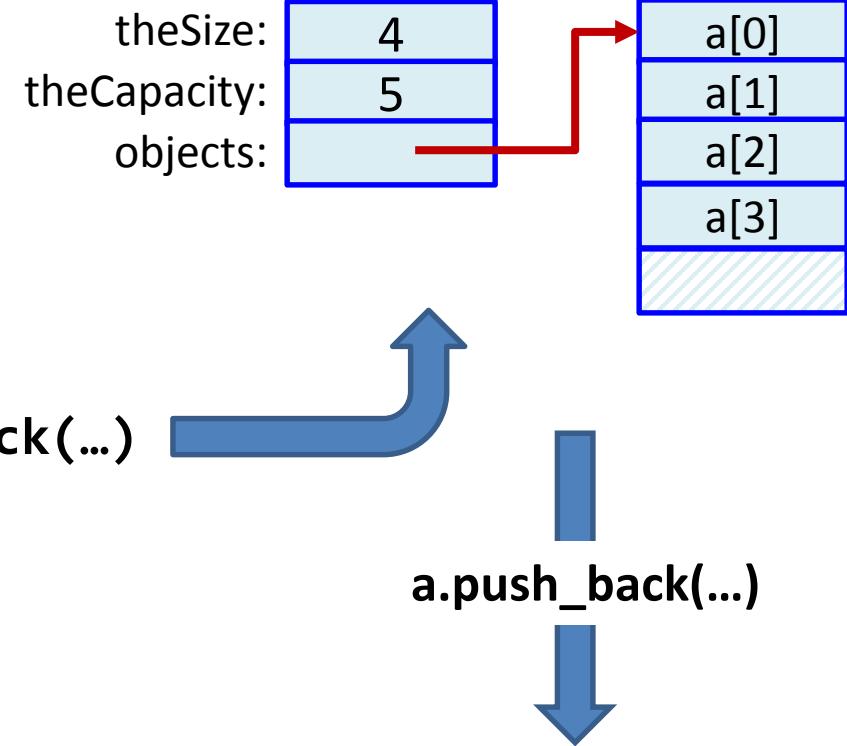
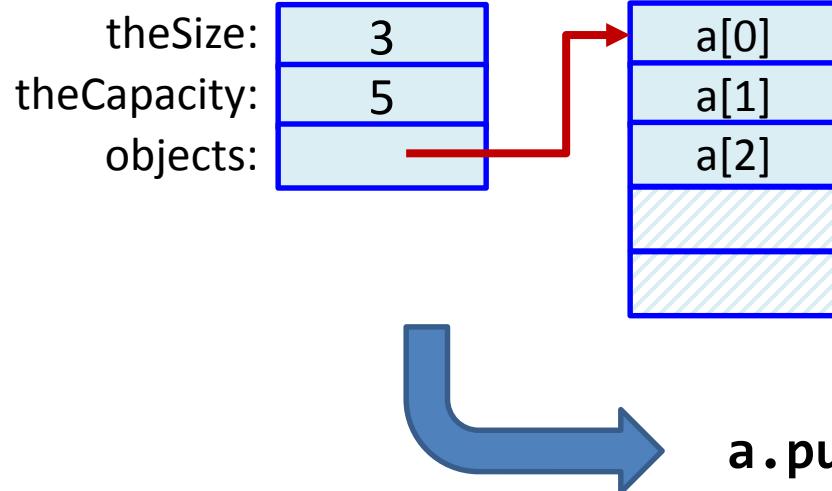
The Vector class

```
template <typename Object>
class Vector {
public:
...
private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```



- A Vector may allocate more memory than needed (size vs. capacity).
- The memory must be reallocated when there is no enough capacity in the storage area.
- The pointer stores the base memory address (location of `objects[0]`). Pointers can be used to allocate/free memory blocks via new/delete operators.

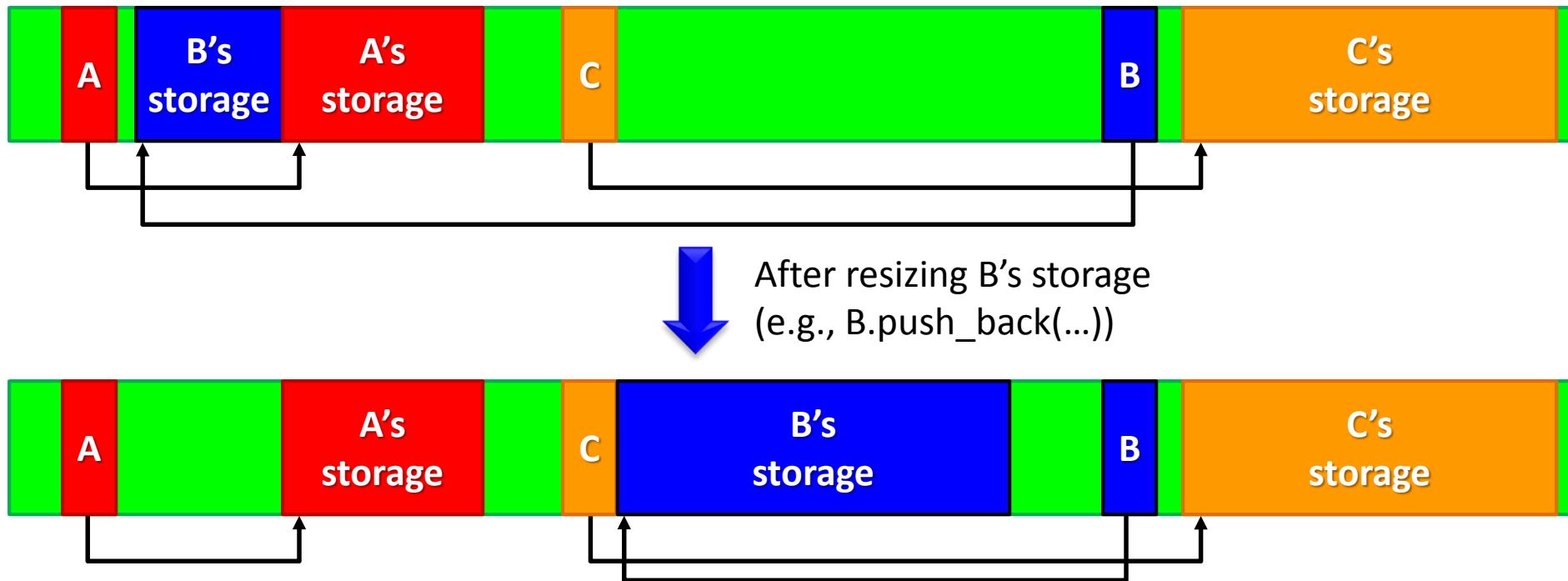
The Vector class



?

← `a.push_back(...)` →

Memory management



- Data structures usually have a descriptor (fixed size) and a storage area (variable size). Memory blocks cannot always be resized. They need to be reallocated and initialized with the old block. After that, the old block can be freed.
- Programmers do not have to take care of memory allocation, but it is convenient to know the basics of memory management.
- Beware of memory leaks, fragmentation, ...

The Vector class

```
public:  
    // Returns the size of the vector  
    int size() const  
    { return theSize; }  
  
    // Is the vector empty?  
    bool empty() const  
    { return size() == 0; }  
  
    // Adds an element to the back of the vector  
    void push_back(const Object & x) {  
        if (theSize == theCapacity) reserve(2*theCapacity + 1);  
        objects[theSize++] = x;  
    }  
  
    // Removes the last element of the array  
    void pop_back()  
    { theSize--; }
```



see later

The Vector class

```
public:  
    // Returns a const ref to the last element  
    const Object& back() const  
    { return objects[theSize - 1]; }  
  
    // Returns a ref to the i-th element  
    Object& operator[](int i)  
    { return objects[i]; }  
  
    // Returns a const ref to the i-th element  
    const Object& operator[](int i) const  
    { return objects[i]; }  
  
    // Modifies the size of the vector (destroying  
    // elements in case of shrinking)  
    void resize(int newSize) {  
        if (newSize > theCapacity) reserve(newSize*2);  
        theSize = newSize;  
    }
```

The Vector class

```
// Reserves space (to increase capacity)
void reserve(int newCapacity) {
    if (newCapacity < theSize) return;

    // Allocate the new memory block
    Object *newArray = new Object[newCapacity];

    // Copy the old memory block
    for (int k = 0; k < theSize; ++k)
        newArray[k] = objects[k];

    theCapacity = newCapacity;
    // Swap pointers and free old memory block
    std::swap(objects, newArray);
    delete [] newArray;
}
```

Constructors, copies, assignments

```
MyClass a, b; // Constructor is used
```

```
MyClass c = a; // Copy constructor is used
```

```
b = c; // Assignment operator is used
```

```
// Copy constructor is used when passing  
// the argument to a function (or returning  
// the value from a function)
```

```
void foo(Myclass x);
```

```
...
```

```
foo(c); // Copy constructor is used
```

The Vector class

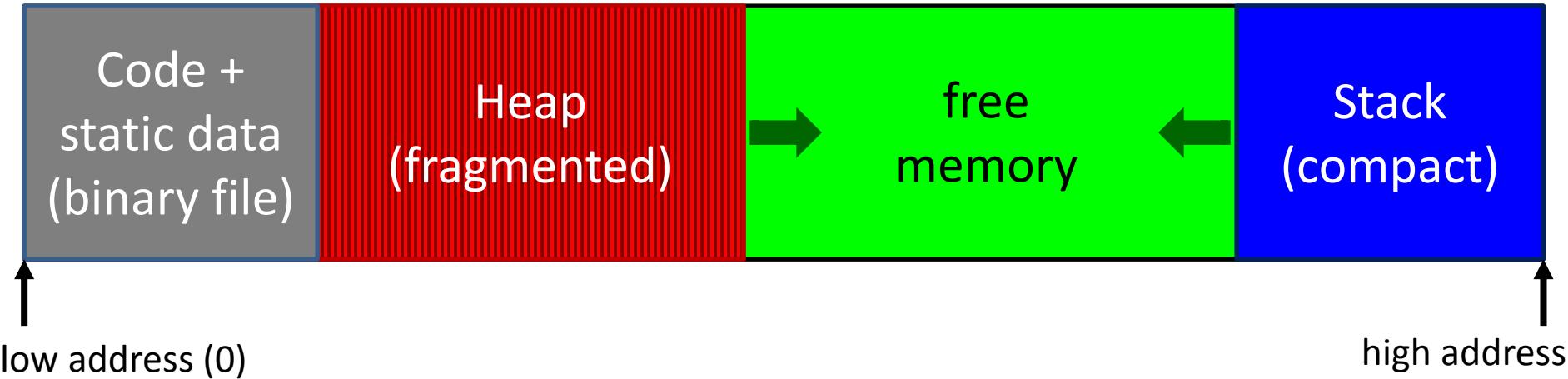
```
// Default constructor with initial size
Vector(int initSize = 0) : theSize{initSize},
    theCapacity{initSize + SPARE_CAPACITY}
{ objects = new Object[theCapacity]; }

// Copy constructor
Vector(const Vector& rhs) : theSize{rhs.theSize},
    theCapacity{rhs.theCapacity}, objects{nullptr} {
    objects = new Object[theCapacity];
    for (int k = 0; k < theSize; ++k) objects[k] = rhs.object[k];
}

// Assignment operator
Vector& operator=(const Vector& rhs) {
    if (this != &rhs) {           // Avoid unnecessary copy if identical
        theSize = rhs.theSize;
        theCapacity = rhs.theCapacity;
        delete [] objects;
        objects = new Object[theCapacity];
        for (int k = 0; k < theSize; ++k) objects[k] = rhs.object[k];
    }
    return *this;
}

// Destructor
~Vector() { delete [] objects; }
```

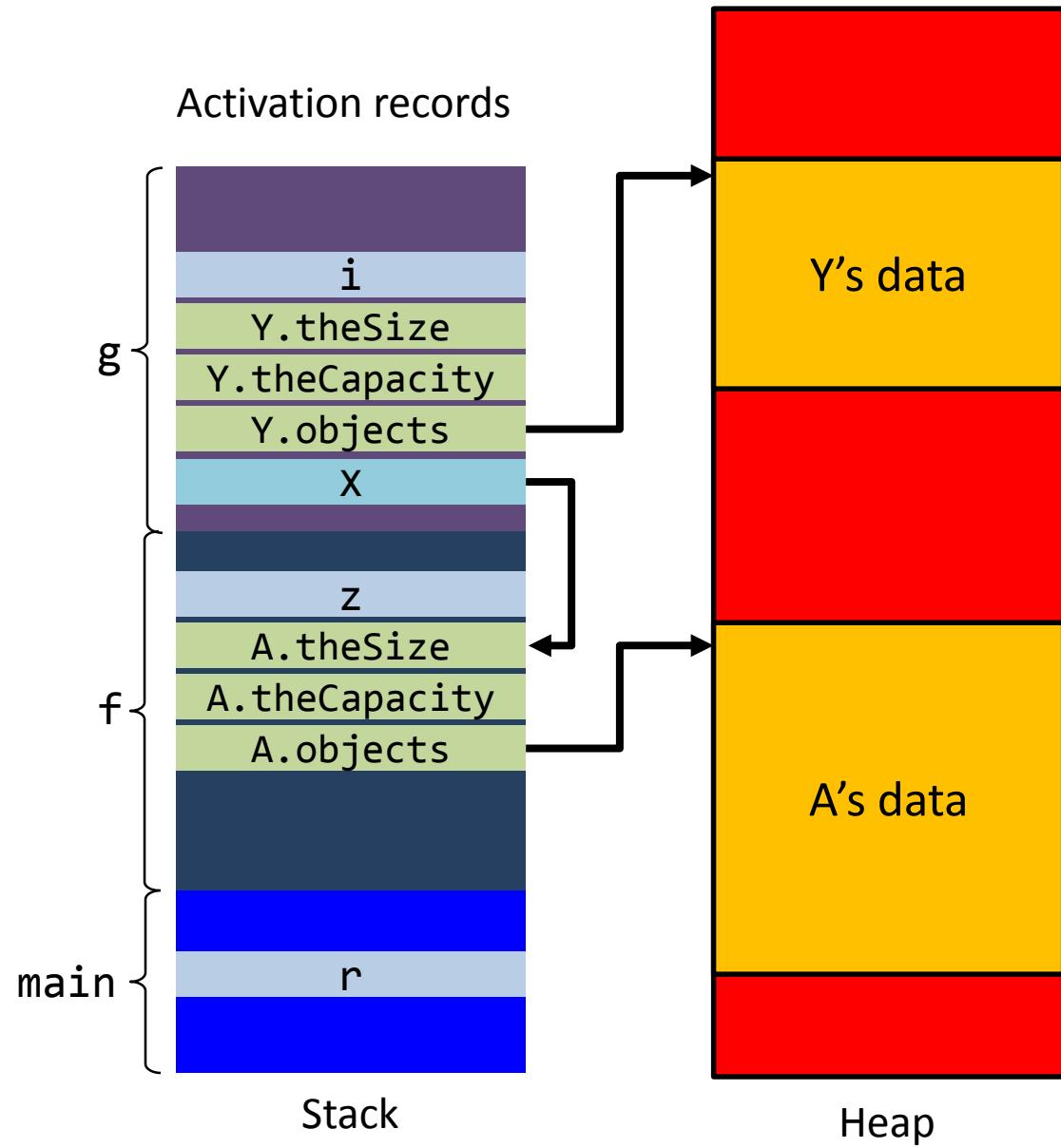
Memory layout of a program



Region	Type of data	Lifetime
Static	Global data	Lifetime of the program
Stack	Local variables of a function	Lifetime of the function
Heap	Dynamic data	Since created (<i>new</i>) until destroyed (<i>delete</i>)

Memory layout of a program

```
int g(vector<int>& X) {  
    int i;  
    vector<double> Y;  
    ...  
}  
  
void f() {  
    int z;  
    vector<int> A;  
    ...  
    z = g(A);  
    ...  
}  
  
int main() {  
    double r;  
    ...  
    f();  
    ...  
}
```



Memory management models

- **Programmer-controlled management:**
 - The programmer decides when to allocate (new) and deallocate (delete) blocks of memory.
 - Example: C++.
 - Pros: efficiency, memory management can be optimized.
 - Cons: error-prone (dangling references and memory leaks)
- **Automatic management:**
 - The program periodically launches a garbage collector that frees all non-referenced blocks of memory.
 - Examples: Java, python, R.
 - Pros: the programmer does not need to worry about memory management.
 - Cons: cannot optimize memory management, less control over runtime.

Dangling references and memory leaks

```
myClass* A = new myClass;  
myClass* B = new myClass;  
// We have allocated space for two objects  
  
A = B;  
// A and B point at the same object!  
// Possible memory leak (unreferenced memory space)  
  
delete A;  
// Now B is a dangling reference  
// (points at free space)  
  
delete B; // Error
```

Pointers and memory leaks

```
for (i = 0; i < 1000000; ++i) {
    myClass * A = new Myclass; // 1Kbyte
    ...
    do_something(A, i);
    ...
    // forgets to delete A
}
// This loop produces a 1-Gbyte memory leak!
```

Recommendations:

- Do not use pointers, unless you are very desperate.
- If you have to use pointers, hide their usage inside a class and define consistent constructors/destructors.
- Use *valgrind* (valgrind.org) to detect memory leaks.
- Remember: no pointers → no memory leaks.

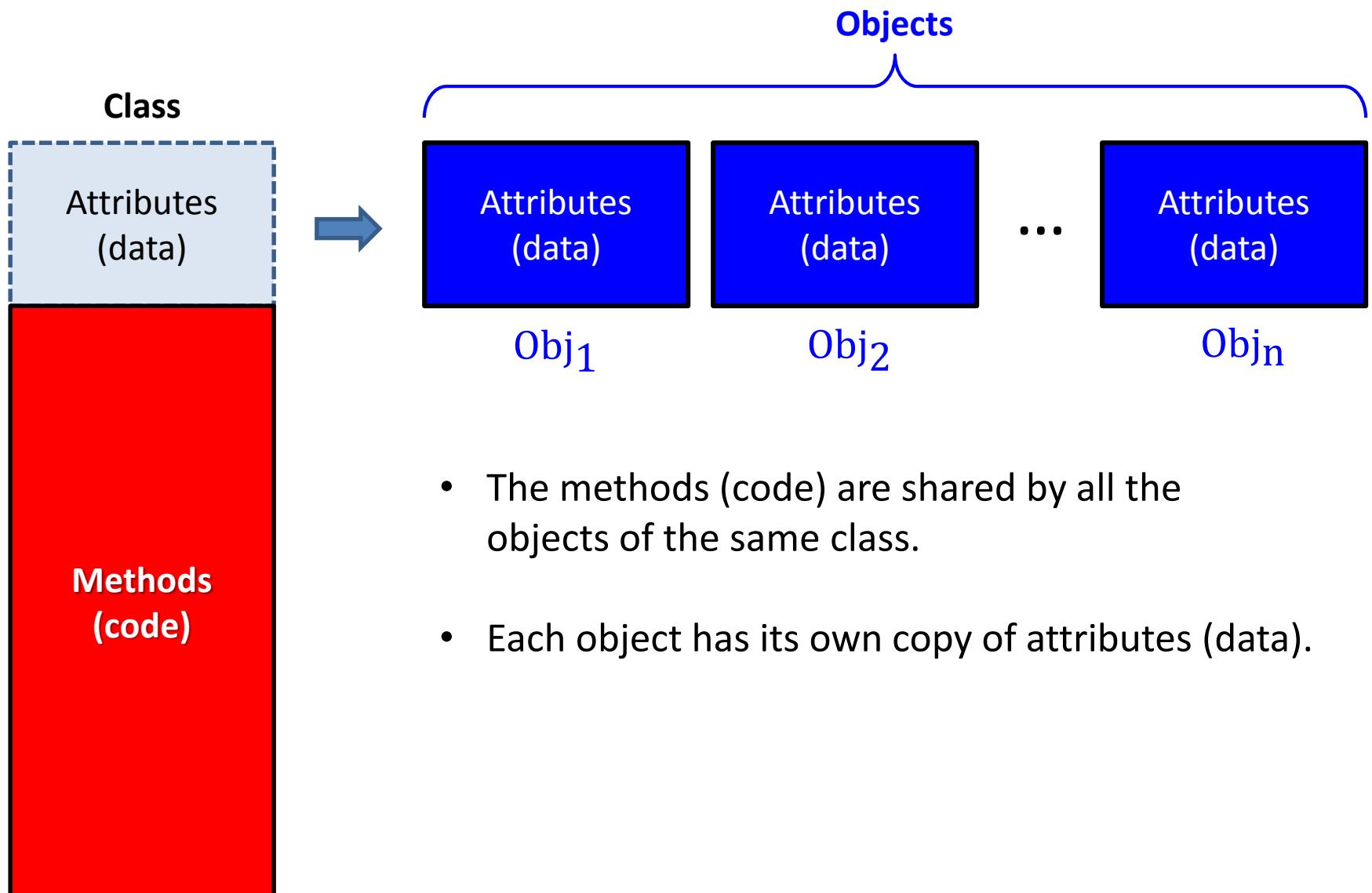
Pointers and references to dynamic data

```
vector<myClass> a;  
...  
// do some push_back's to a  
...  
const myClass& x = a[0]; // ref to a[0]  
// x contains a memory address pointing at a[0]  
  
a.push_back(something);  
// the address of a[0] might have changed!  
// x might be pointing at garbage data
```

Recommendation: don't trust on pointers/references to dynamic data.

Possible solution: use indices/keys to access data in containers.

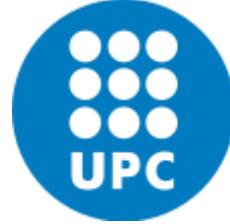
How much memory space does an object take?



Summary

- Memory is a valuable resource in computing devices. It must be used efficiently.
- Languages are designed to make memory management transparent to the user, but a lot of inefficiencies may arise unintentionally (e.g., copy by value).
- Pointers imply the use of the heap and all the problems associated to memory management (memory leaks, fragmentation).
- Recommendation: do not use pointers unless you have no other choice. Not using pointers will save a lot of debugging time.
- In case of using pointers, try to hide the pointer manipulation and memory management (new/delete) inside the class in such a way that the user of the class does not need to “see” the pointers.

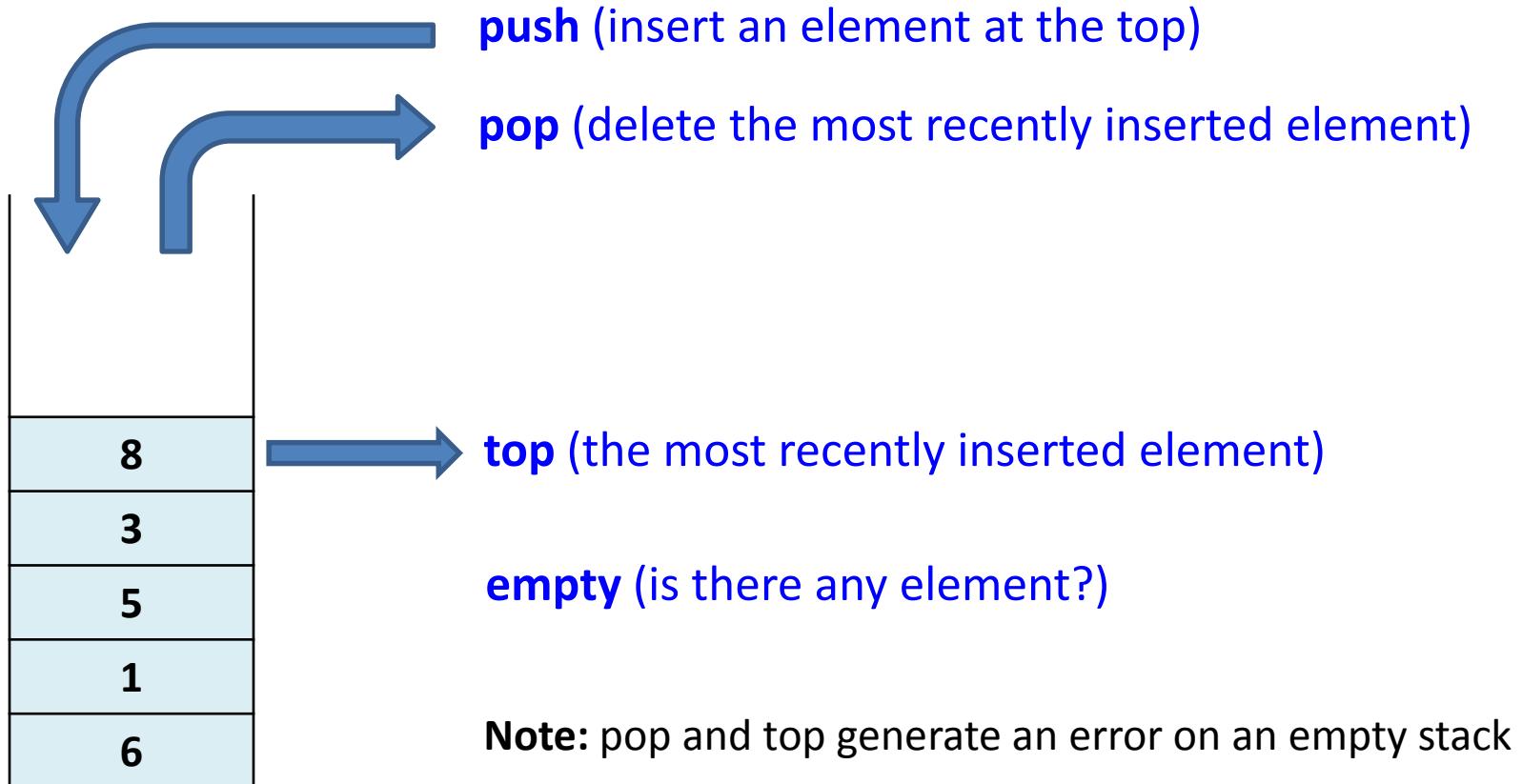
Containers: Stack



Jordi Cortadella and Jordi Petit
Department of Computer Science

The Stack ADT

- A stack is a list of objects in which insertions and deletions can only be performed at the top of the list.
- Also known as LIFO (Last In, First Out)



The Stack ADT

```
template <typename T>
class Stack {
public:
    // Default constructor
    Stack() {}

    ...

private:
    vector<T> data;
};
```

- The definition can handle generic stacks of any type T.
- The default constructor does not need to do anything: a zero-sized vector is constructed by default.

The Stack ADT

```
template <typename T>
class Stack {
public:
    bool empty() const {
        return data.size() == 0;
    }

    const T& top() const { // Returns a const reference
        assert (not empty());
        return data.back();
    }

    T& top() { // Returns a reference
        assert (not empty());
        return data.back();
    }

    void pop() {
        assert (not empty());
        data.pop_back();
    }

    void push(const T& x) {
        data.push_back(x);
    }
};
```

Balancing symbols

- **Balancing symbols:** check for syntax errors when expressions have opening/closing symbols, e.g., () [] {}

Correct: [(){}()[]{}]()]

Incorrect: [(){}(}...

- **Algorithm** (linear): read all chars until end of file. For each char, do the following:
 - If the char is opening, push it onto the stack.
 - If the char is closing and stack is empty → error, otherwise pop a symbol from the stack and check they match. If not → error.
 - At the end of the file, check the stack is empty.
- **Exercise:** implement and try the above examples.

Evaluation of postfix expressions

- This is an infix expression. What's his value? 42 or 968?

8 * 3 + 10 + 2 * 4

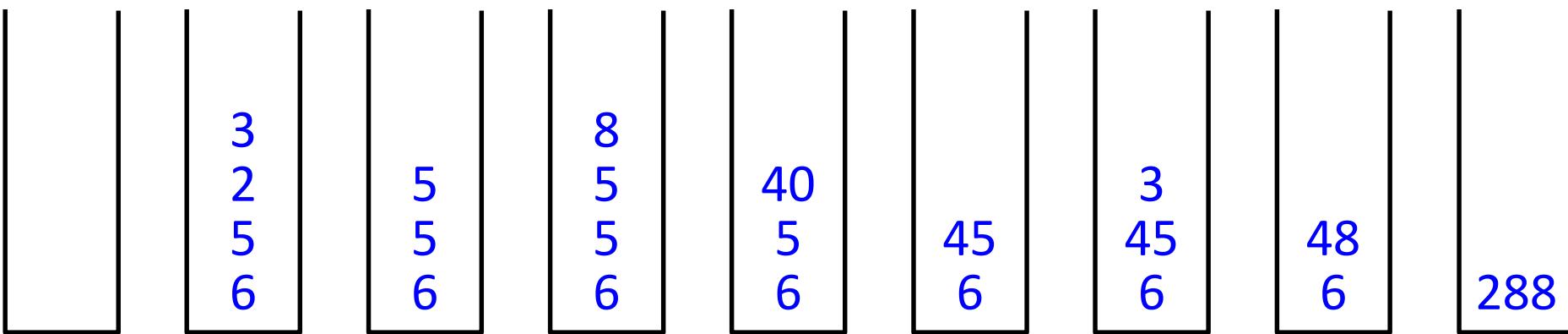
- It depends on the operator precedence. For scientific calculators, * has precedence over +.
- Postfix (reverse Polish notation) has no ambiguity:

8 3 * 10 + 2 4 * +

- Postfix expressions can be evaluated using a stack:
 - each time an operand is read, it is pushed on the stack
 - each time an operator is read, the two top values are popped and operated. The result is push onto the stack

Evaluation of postfix expressions: example

6 5 2 3 + 8 * + 3 + *



push(6)
push(5)
push(2)
push(3)

+

push(8)

*

+

push(3)

+

*

From infix to postfix

a + b * c + (d * e + f) * g



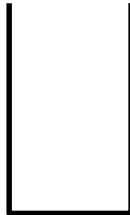
a b c * + d e * f + g * +

Algorithm:

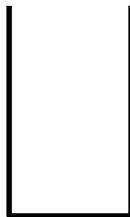
- When an operand is read, write it to the output.
- If we read a right parenthesis, pop the stack writing symbols until we encounter the left parenthesis.
- For any other symbol (+, *, (,), pop entries and write them until we find an entry with lower priority. After popping, push the symbol onto the stack. Exception: (can only be removed when finding a).
- When the end of the input is reached, all symbols in the stack are popped and written onto the output.

From infix to postfix

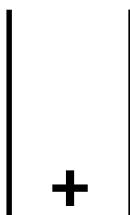
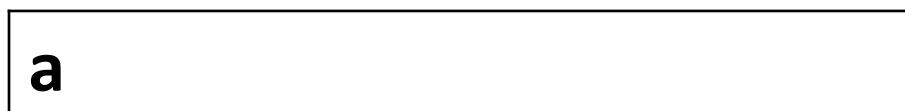
a + b * c + (d * e + f) * g



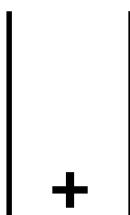
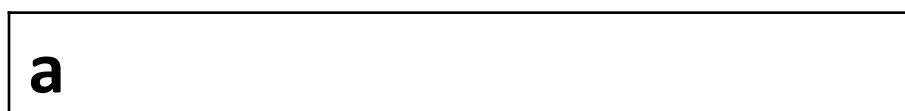
Output



a



a

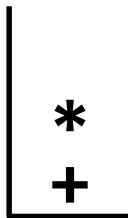


a b

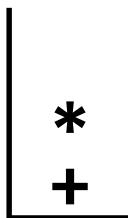


From infix to postfix

a + b * c + (d * e + f) * g



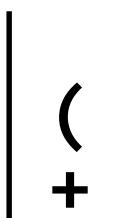
a b



a b c



a b c * +



a b c * +

From infix to postfix

a + b * c + (d * e + f) * g

(
+
)

a b c * + d

*
(
+
)

a b c * + d

*
(
+
)

a b c * + d e

+
(
+
)

a b c * + d e *

From infix to postfix

a + b * c + (d * e + f) * g

+
(
+

a b c * + d e * f

)
+

a b c * + d e * f +

*
+

a b c * + d e * f +

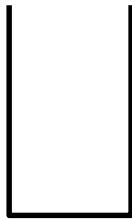
*

+

a b c * + d e * f + g

From infix to postfix

a + b * c + (d * e + f) * g



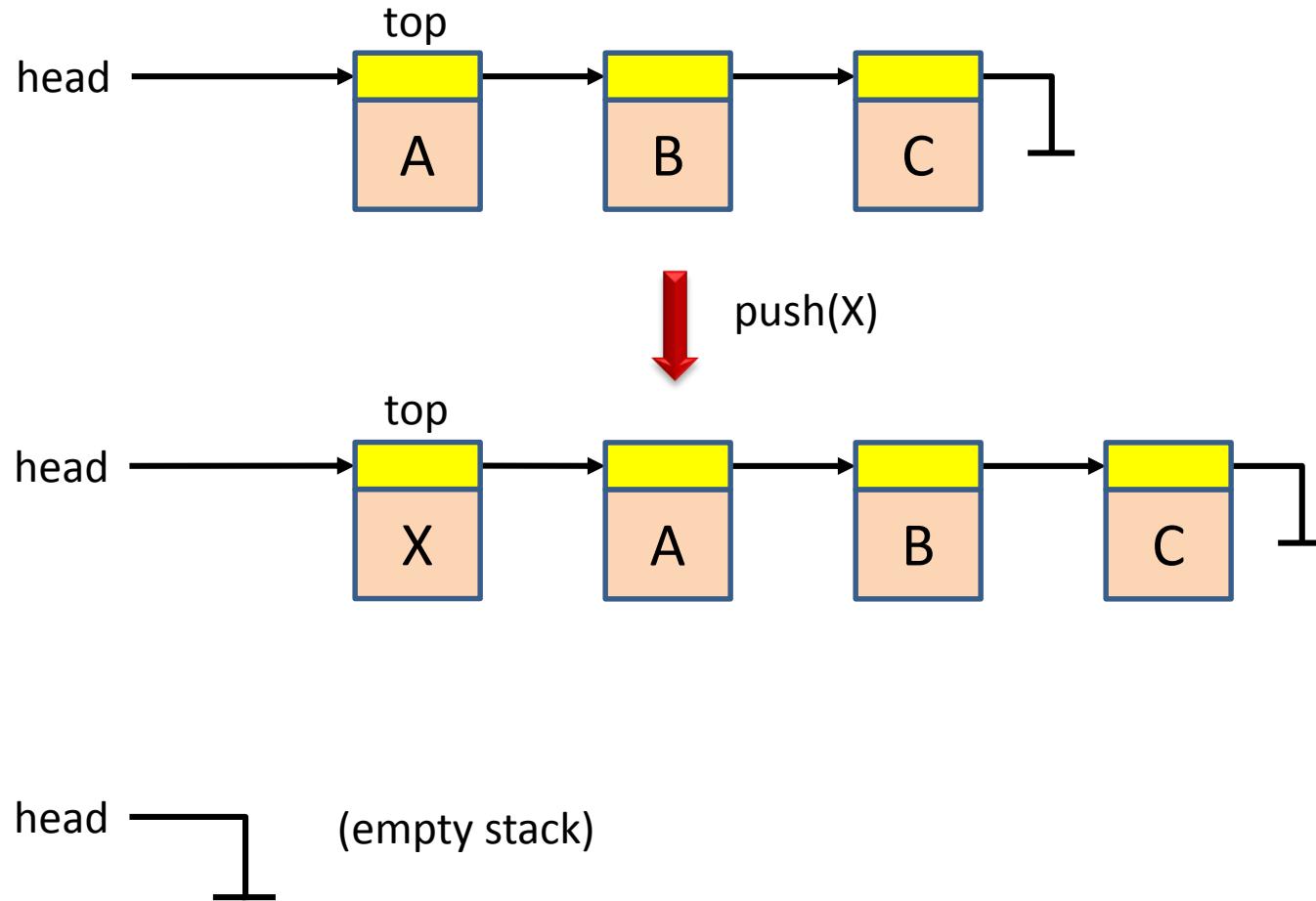
a b c * + d e * f + g * +

Complexity: $O(n)$

Suggested exercise:

- Add subtraction (same priority as addition) and division (same priority as multiplication).

The Stack ADT with pointers



The Stack ADT with pointers

```
template <typename Type>
class Stack {
    private:
        // The internal cell to represent a node
        struct Node {
            Type elem;      // Data
            Node* next;    // pointer to the next element
        };
        Node* head;    // Pointer to the top of the stack
        int n;         // Number of elements
};
```

The Stack ADT with pointers

```
public:  
    int size() const {  
        return n;  
    }  
  
    bool empty() const {  
        return size() == 0;  
        // or also head == nullptr  
    }  
  
    void push(const Type& x) {  
        head = new Node{x, head};  
        ++n;  
    }  
  
    void pop() {  
        assert(not empty());  
        Node* old = head;  
        head = head->next;  
        delete old;  
        --n;  
    }
```

```
// Returns a copy  
Type top() const {  
    assert(not empty());  
    return head->elem;  
}  
  
// Returns a reference  
Type& top() {  
    assert(not empty());  
    return head->elem;  
}  
  
// Returns a const reference  
const Type& top() const {  
    assert(not empty());  
    return head->elem;  
}
```

The Stack ADT with pointers: usage

```
Stack<Person> S; // Empty stack
Person p("MyName", myAge);

S.push(p);

Person q1 = S.top(); // Makes a copy
Person& q2 = S.top(); // Ref to the top
const Person& q3 = S.top(); // Const ref to the top

q1.name = "newName"; // Only local copy modified
q2.name = "newName"; // Top of the stack modified
q3.name = "newName"; // Illegal (q3 is const)
q2 = q1; // Illegal: references cannot be modified
```

The Stack ADT with pointers

```
private:

    // Recursive copy of the elements.
    // Exercise: design an iterative version.
    Node* copy(Node* p) {
        if (p) return new Node{p->elem, copy(p->next)};
        else return nullptr;
    }

    // Recursive deallocation of the elements
    void free(Node* p) {
        if (p) {
            free(p->next);
            delete p;
        }
    }
}
```

The Stack ADT with pointers

```
public:  
    // Default constructor (empty stack)  
    Stack() : head(nullptr), n(0) {}  
  
    // Copy constructor  
    Stack(const Stack& s) : head(copy(s.head)), n(s.n) {}  
  
    // Assignment constructor  
    Stack& operator= (const Stack& s) {  
        if (&s != this) { // Make a new copy only if different  
            free(head);  
            head = copy(s.head);  
            n = s.n;  
        }  
        return *this; // Must return a ref to the object  
    }  
  
    // Destructor  
    ~Stack() {  
        free(head);  
    }
```

Assertions vs. error handling

- Assertions:
 - Runtime checks of properties (e.g., invariants, pre-/post-conditions).
 - Useful to detect internal errors.
 - They should always hold in a bug-free program.
 - They should give meaningful error messages to the programmers.
- Error handling:
 - Detect improper use of the program (e.g., incorrect input data).
 - The program should not halt unexpectedly.
 - They should give meaningful error messages to the users.

Assertions vs. error handling

```
assert (x >= 0);
double y = sqrt(x);

assert (i >= 0 and i < A.size());
int v = A[i];

assert (p != nullptr);
int n = p->number;

string id;
cin >> id;
if (isdigit(name[0])) {
    cerr << "Invalid identifier" << endl;
    return false;
}
```

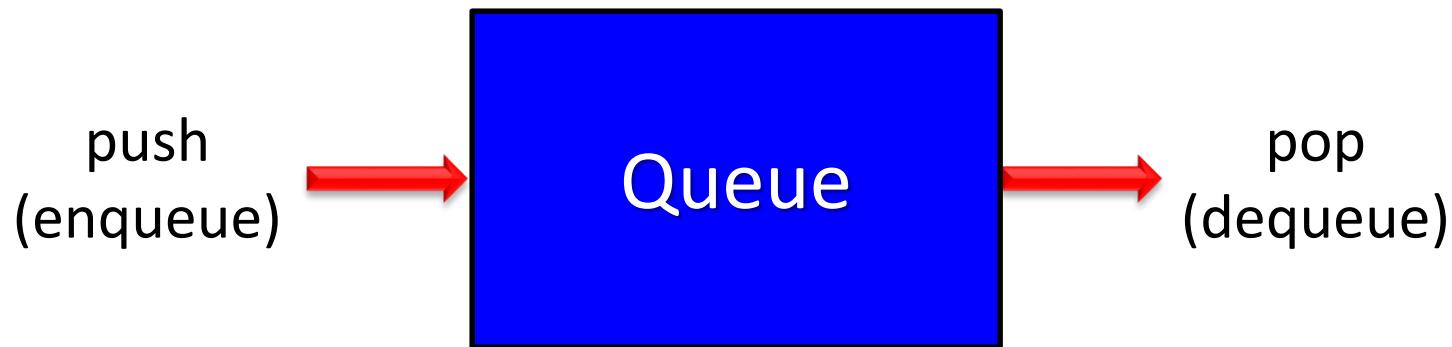
Containers: Queue and List



Jordi Cortadella and Jordi Petit
Department of Computer Science

Queue

- A container in which insertion is done at one end (the tail) and deletion is done at the other end (the head).
- Also called FIFO (First-In, First-Out)



Queue usage

```
Queue<int> Q; // Constructor

Q.push(5);      // Inserting few elements
Q.push(8);
Q.push(6);

int n = Q.size(); // n = 3

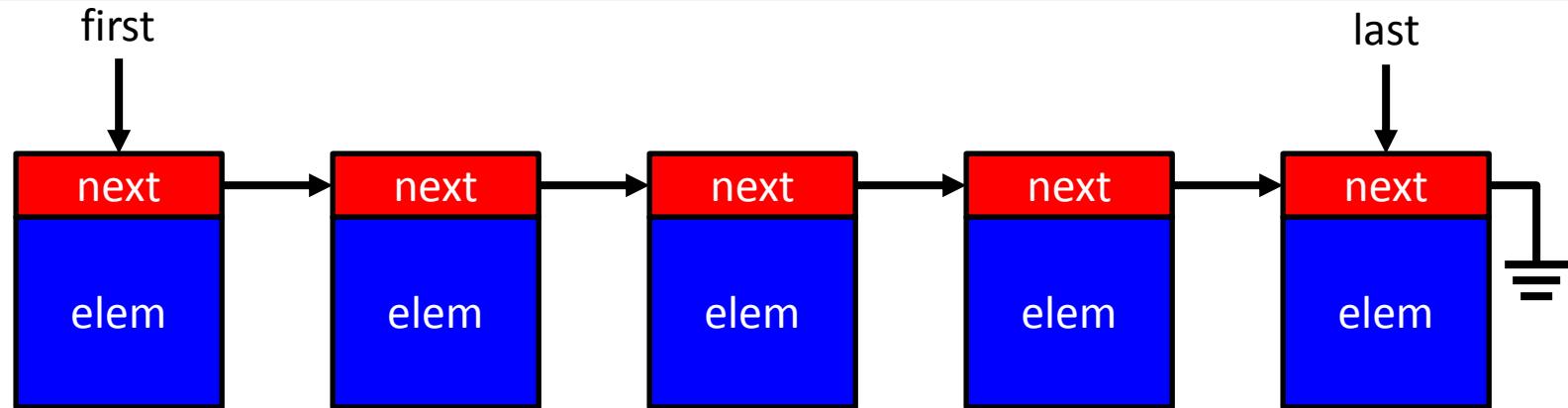
while (not Q.empty()) {
    int elem = Q.front();           // Get the first element
    cout << elem << endl;
    Q.pop();                        // Delete the element
}
```

The class Queue

```
template<typename T>
class Queue {
public:

    Queue();      // Constructor
    ~Queue();     // Destructor
    Queue(const T& Q);           // Copy constructor
    Queue& operator= (const Queue& Q); // Assignment operator
    void push(const& T x);       // Enqueues an element
    void pop();                // Dequeues the first element
    T front() const;           // Returns the first element
    int size() const;           // Number of elements in the queue
    bool empty() const;         // Is the queue empty?
};
```

Implementation with linked lists



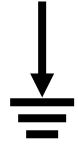
```
template<typename T>
class Queue {

private:
    struct Node {
        T elem;
        Node* next;
    };

    Node *first;           // Pointer to the first element
    Node *last;            // Pointer to the last element
    int n;                // Number of elements
};
```

Implementation with linked lists

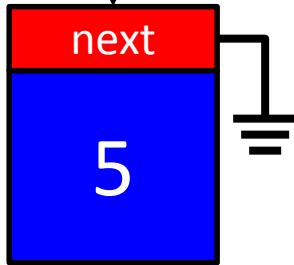
first, last



Q.push(5)



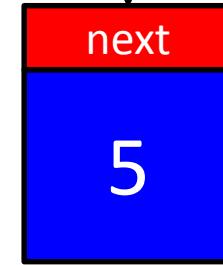
first, last



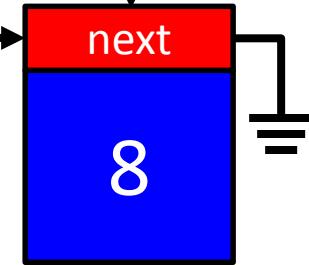
Q.push(8)



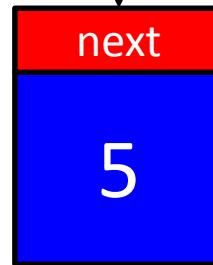
first



last



first



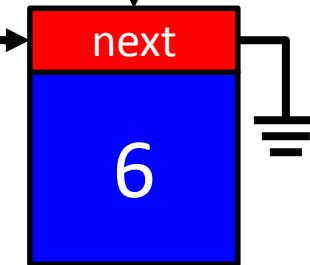
Q.push(6)



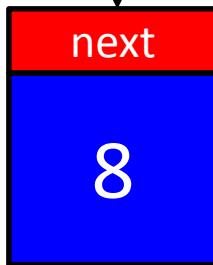
next



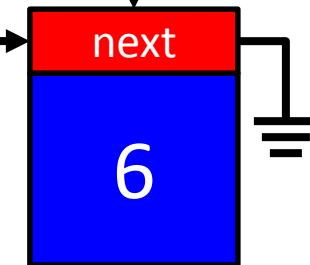
last



first



last



Q.pop()



Queue: some methods

```
/** Returns the number of elements. */
int size() const {
    return n;
}

/** Checks whether the queue is
empty. */
bool empty() const {
    return size() == 0;
}

/** Inserts a new element at the end
of the queue. */
void push(const T& x) {
    Node* p = new Node {x, nullptr};
    if (n++ == 0) first = last = p;
    else last = last->next = p;
}
```

```
/** Removes the first element.
Pre: the queue is not empty. */
void pop() {
    assert(not empty());
    Node* old = first;
    first = first->next;
    delete old;
    if (--n == 0) last = nullptr;
}

/** Returns the first element.
Pre: the queue is not empty. */
T front() const {
    assert(not empty());
    return first->elem;
}
```

Queue: constructors and destructor

```
/** Default constructor: an empty queue. */
Queue() : first(nullptr), last(nullptr), n(0) { }

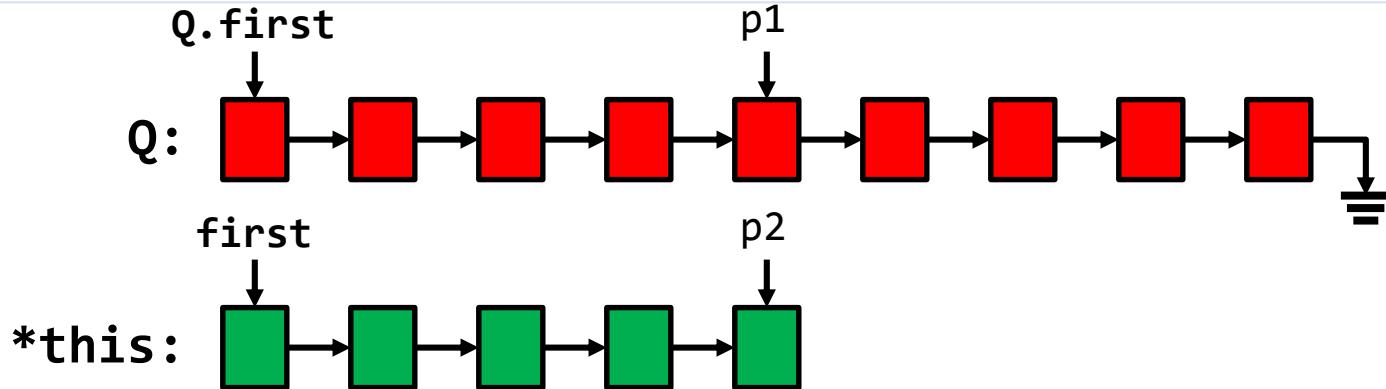
/** Copy constructor. */
Queue(const Queue& Q) {
    copy(Q);
}

/** Assignment operator. */
Queue& operator= (const Queue& Q) {
    if (&Q != this) {
        free();
        copy(Q);
    }
    return *this;
}

/** Destructor. */
~Queue() {
    free();
}
```

```
private:
    /** Frees the linked list of
     * nodes in the queue. */
    void free() {
        Node* p = first;
        while (p) {
            Node* old = p;
            p = p->next;
            delete old;
        }
    }
```

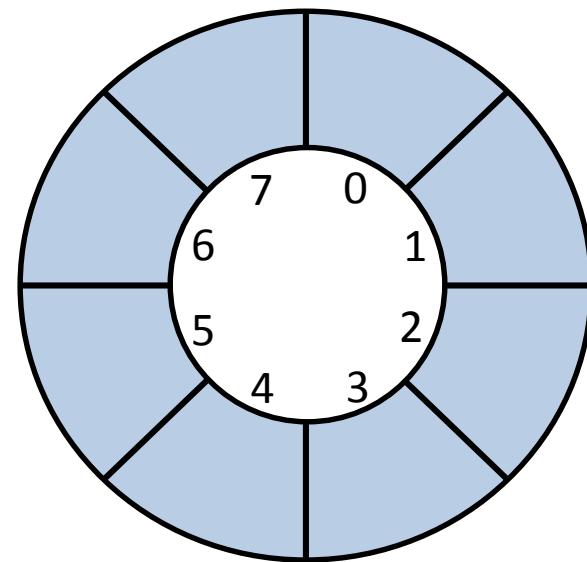
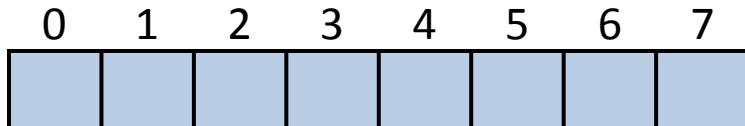
Queue: copy (private)



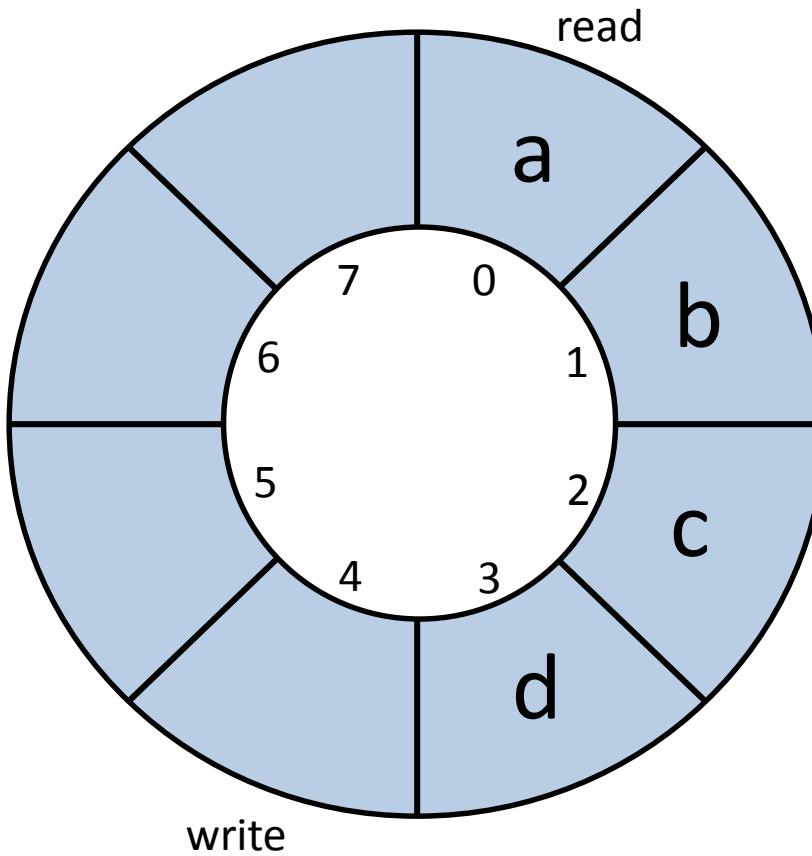
```
/** Copies a queue. */
void copy(const Queue& Q) {
    n = Q.n;
    if (n == 0) {
        first = last = nullptr;
    } else {
        Node* p1 = Q.first;
        Node* p2 = first = new Node {p1->elem};
        while (p1->next) {
            p1 = p1->next;
            p2 = p2->next = new Node {p1->elem};
        }
        p2->next = nullptr;
        last = p2;
    }
}
```

Implementation with circular buffer

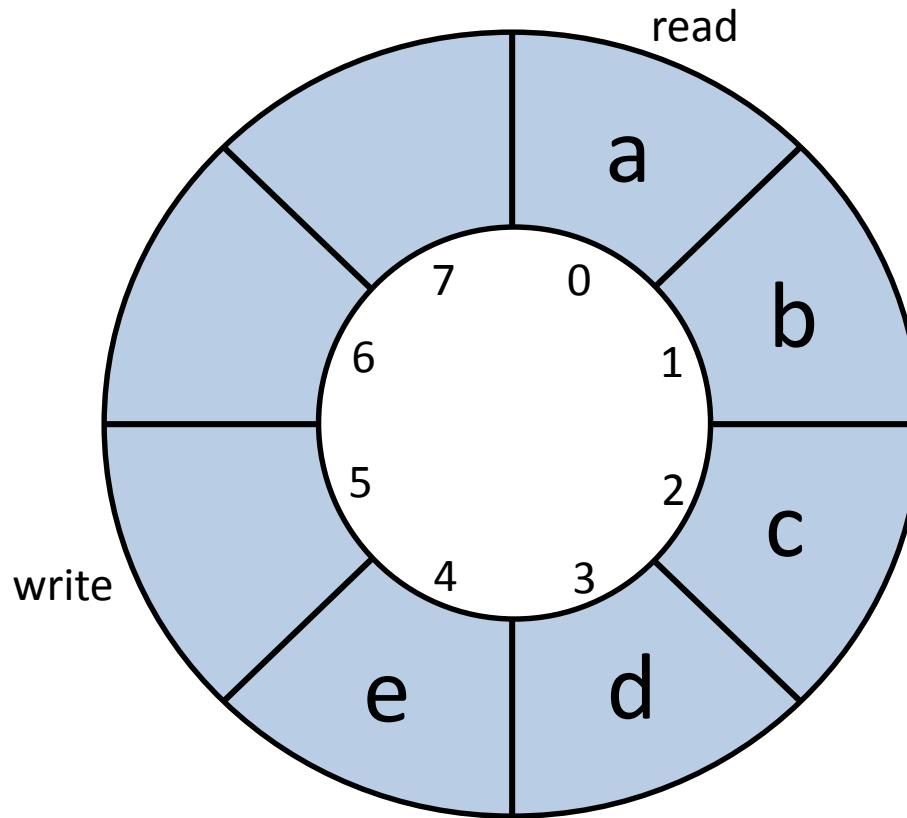
- A queue can also be implemented with an array (vector) of elements.
- It is a more efficient representation if the maximum number of elements in the queue is known in advance.



Implementation with circular buffer

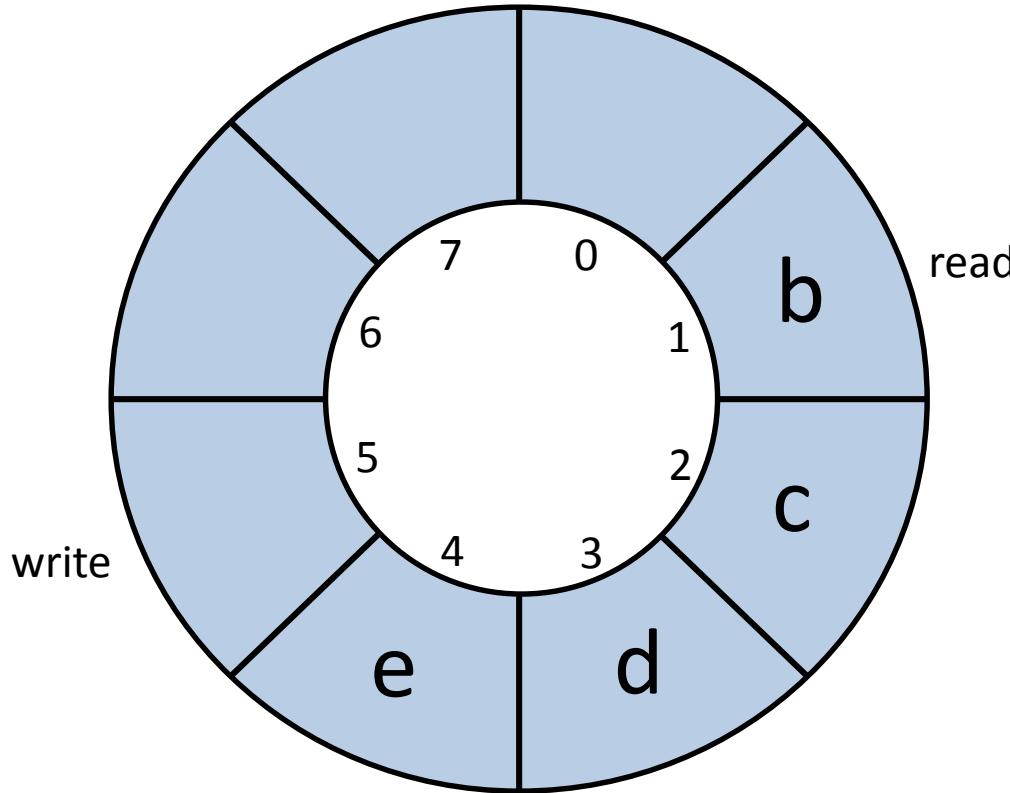


Implementation with circular buffer



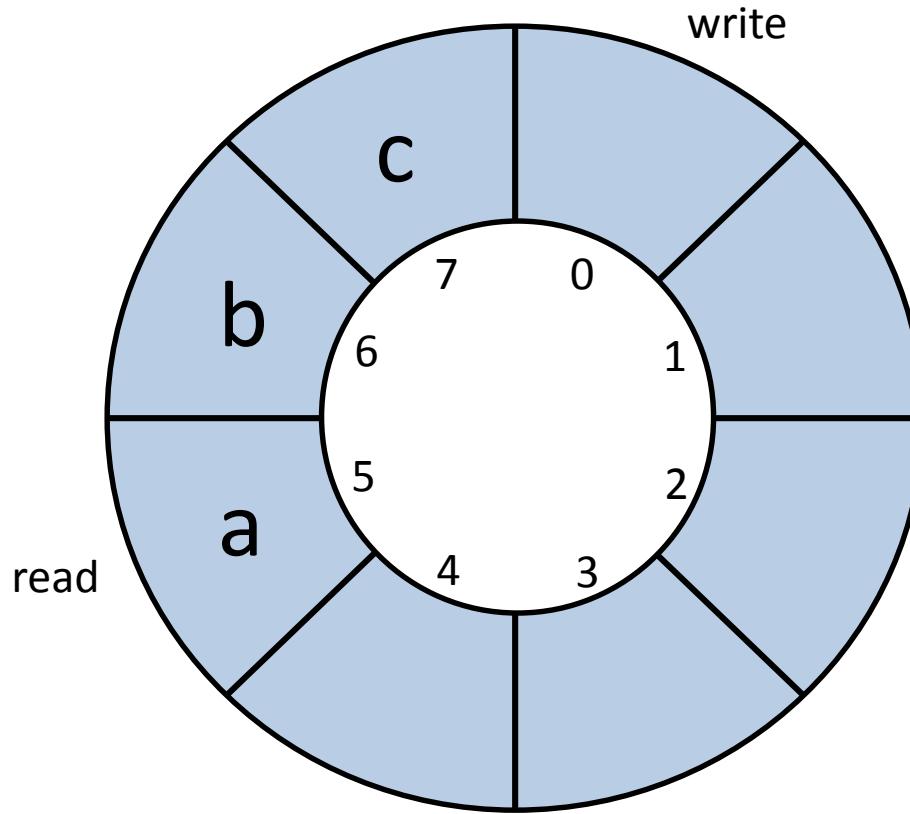
after **Q.push(e)**

Implementation with circular buffer

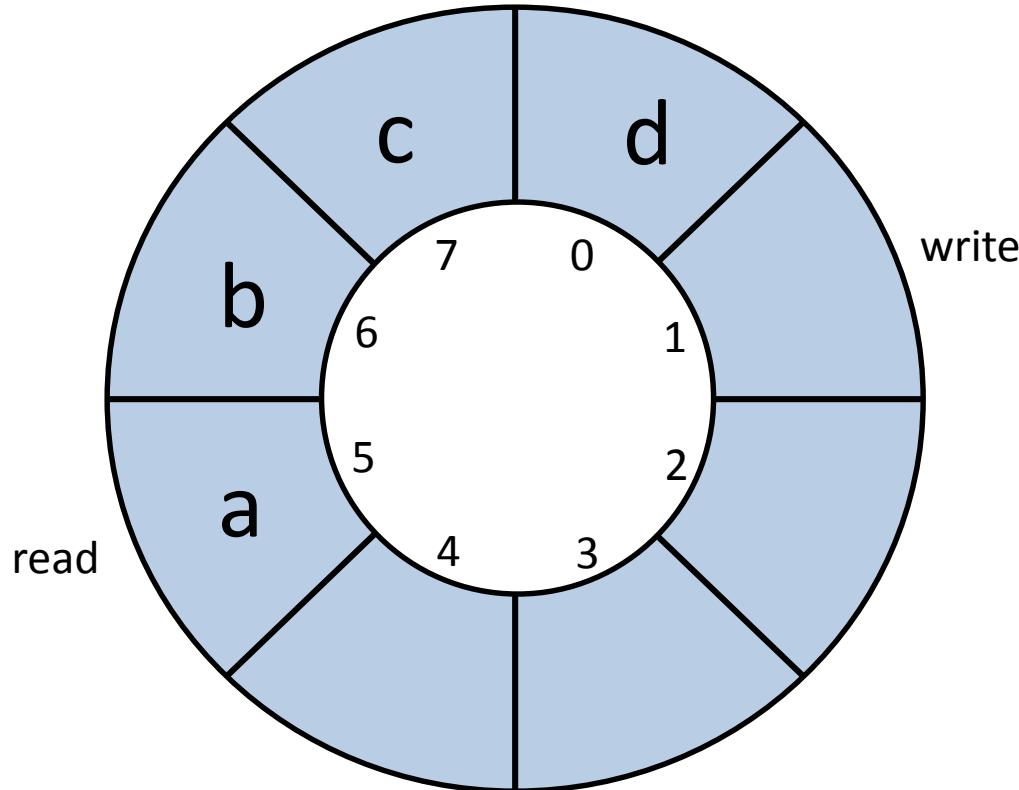


after **Q.pop()**

Implementation with circular buffer



Implementation with circular buffer



after **Q.push(d)**

The class Queue

```
template<typename T>
class Queue {
public:

    Queue(int capacity = 1000); // Constructor (with capacity)

    ~Queue(); // Destructor

    Queue(const T& Q); // Copy constructor

    Queue& operator= (const Queue& Q); // Assignment operator

    void push(const& T x); // Enqueues an element

    void pop(); // Dequeues the element at the head

    T front() const; // Returns the first element

    int size() const; // Number of elements in the queue

    int capacity() const; // Returns the capacity of the queue

    bool empty() const; // Is the queue empty?

    bool full() const; // Is the queue full?

};
```

The class Queue (incomplete)

```
template<typename T>
class Queue {

private:
    vector<T> buffer;           // The buffer to store elements
    int read, write;             // The read/write indices
    int n;                      // The number of elements

public:
    /** Constructor with capacity of the queue. */
    Queue(int capacity=1000) :
        buffer(capacity), read(0), write(0), n(0) {}

    /** Returns the size of the queue. */
    int size() const {
        return n;
    }

    /** Returns the capacity of the queue. */
    int capacity() const {
        return buffer.size();
    }

    ...
}
```

The class Queue (incomplete)

```
/** Checks whether the queue is full. */
bool full() const {
    return size() == capacity();
}

/** Enqueues a new element.
    Pre: the queue is not full. */
void push(const T& x) {
    ++n;
    assert(not full());
    buffer[write] = x;
    inc(write);
}

/** Dequeues the first element.
    Pre: the queue is not empty. */
void pop() {
    assert(not empty());
    inc(read);
    --n;
}
```

```
/** Returns the first element.
    Pre: the queue is not empty. */
T front() const {
    assert(not empty());
    return buffer[read];
}

private:

/** Increases index circularly. */
void inc(int& i) {
    if (++i == capacity()) i = 0;
}
```

Queue: Complexity

- All operations in queues can run in constant time, except for:
 - Copy: linear in the size of the list.
 - Delete: linear in the size of the list.
- Queues do not allow to access/insert/delete elements in the middle of the queue.

Exercise

- Extend the vector-based implementation of the queue to remove the constraint on maximum capacity.
- How:
 - Increase capacity of the vector.
 - Reorganize the elements in the queue.

List

- List: a container with sequential access.
- It allows to insert/erase elements in the middle of the list in constant time.
- A list can be considered as a sequence of elements with one or several cursors (iterators) pointing at internal elements.
- For simplicity, we will only consider lists with one iterator.
- Check the STL list: it can be visited by any number of iterators.

List: graphical representation

first

5

8

3

0

4



last

8

1

4

9

3

L.insert(7)

5

8

3

0

4

7



9

1

4

8

L.move_right()

5

8

3

0

4

7

3



9

1

4

L.erase()

5

8

3

0

4

7

3

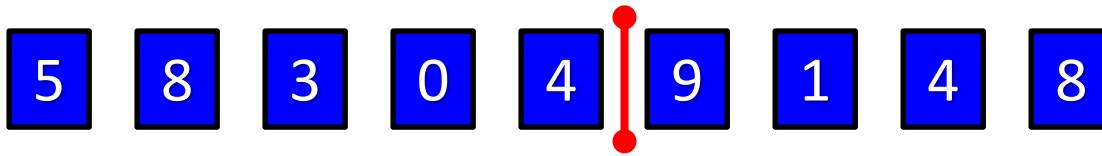


1

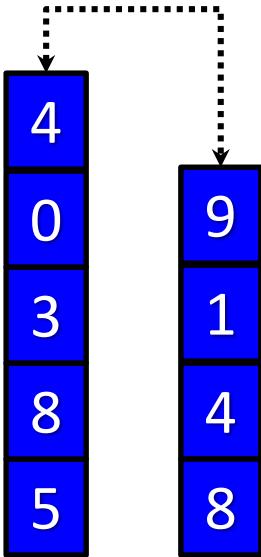
4

8

List implementations



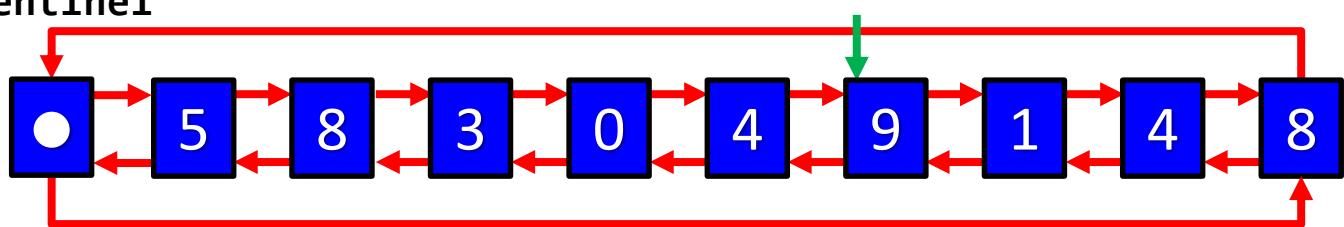
Two stacks



Doubly linked nodes

Sentinel

cursor



The class List: private representation

```
template <typename T>
class List {

    /** Doubly linked node of the list. */
    struct Node {
        Node* prev;    /** Pointer to the previous node. */
        T elem;        /** The element of the list. */
        Node* next;   /** Pointer to the next element. */
    };

    Node* sentinel; /** Sentinel of the list. */
    Node* cursor;   /** Node after the cursor. */
    int n;          /** Number of elements (without sentinel). */
};
```

The class List: public methods

```
public:  
    /** Constructor of an empty list. */  
    List() : sentinel(new Node), cursor(sentinel), n(0) {  
        sentinel->next = sentinel->prev = sentinel;  
    }  
  
    /** Destructor. */  
    ~List() {  
        free();  
    }  
  
    /** Copy constructor. */  
    List(const List& L) {  
        copy(L);  
    }  
  
    /** Assignment operator. */  
    List& operator= (const List& L) {  
        if (&L != this) {  
            free();  
            copy(L);  
        }  
        return *this;  
    }  
  
    /** Returns the number of  
     * elements in the list. */  
    int size() const {  
        return n;  
    }  
  
    /** Checks whether the list  
     * is empty. */  
    bool empty() const {  
        return size() == 0;  
    }
```

The class List: public methods

```
public:  
    /** Checks whether the cursor is at the beginning of the list. */  
    bool is_at_front() const {  
        return cursor == sentinel->next;  
    }  
  
    /** Checks whether the cursor is at the end of the list. */  
    bool is_at_end() const {  
        return cursor == sentinel;  
    }  
  
    /** Moves the cursor one position backward.  
        Pre: the cursor is not at the beginning of the list. */  
    void move_backward() {  
        assert(not is_at_front());  
        cursor = cursor->prev;  
    }  
  
    /** Moves the cursor one position forward.  
        Pre: the cursor is not at the end of the list. */  
    void move_forward() {  
        assert(not is_at_end());  
        cursor = cursor->next;  
    }
```

The class List: public methods

public:

```
/** Moves the cursor to the
   beginning of the list. */
```

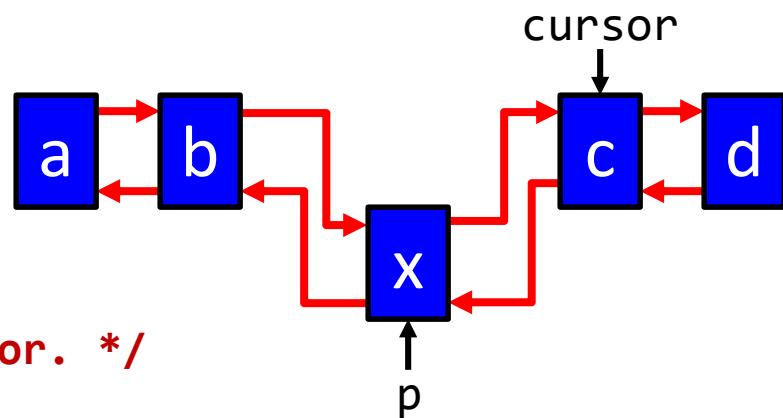
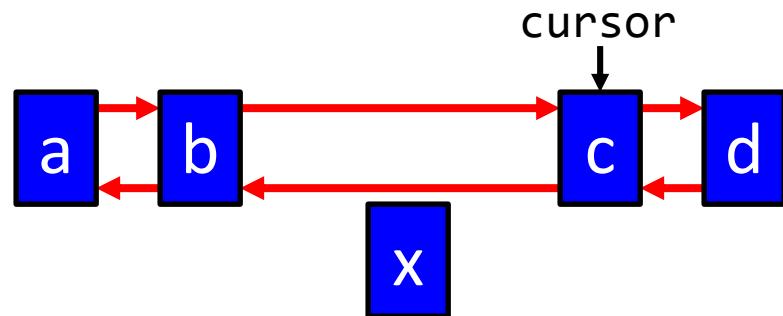
```
void move_to_front() {
    cursor = sentinel->next;
}
```

```
/** Moves the cursor to the
   end of the list. */
```

```
void move_to_end() {
    cursor = sentinel;
}
```

```
/** Inserts an element x before the cursor. */
```

```
void insert(const T& x) {
    Node* p = new Node {cursor->prev, x, cursor};
    cursor->prev = cursor->prev->next = p;
    ++n;
}
```



The class List: public methods

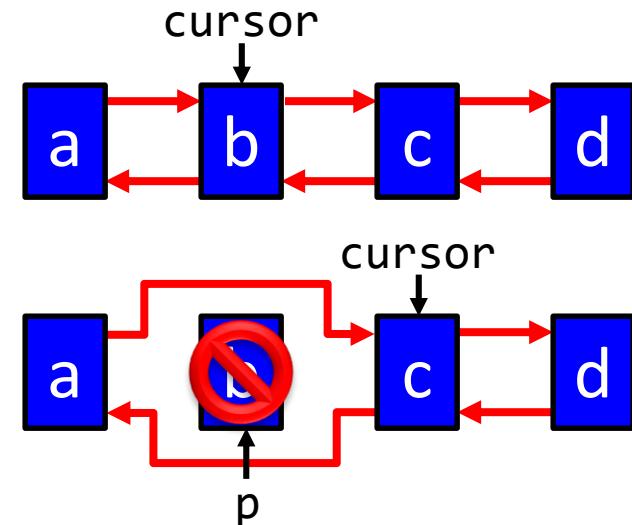
public:

```
/** Erases the element after the cursor.  
 Pre: cursor is not at the end. */
```

```
void erase() {  
    assert(not is_at_end());  
    Node* p = cursor;  
    p->next->prev = p->prev;  
    cursor = p->prev->next = p->next;  
    delete p;  
    --n;  
}
```

```
/** Returns the element after the cursor.  
 Pre: the cursor is not at the end. */
```

```
T front() const {  
    assert(not is_at_end());  
    return cursor->elem;  
}
```



Exercises: implement the private methods **copy()** and **free()**.

Exercises for lists

- Design the method `reverse()` that reverses the contents of the list:
 - No auxiliary lists should be used.
 - No copies of the elements should be performed.
- Solve the Josephus problem, for n people and executing every k -th person, using a circular list:

https://en.wikipedia.org/wiki/Josephus_problem

Exercises for lists

- Design the method `merge(const List& L)` that merges the list with another list L, assuming that both lists are sorted. Assume that a pair of elements can be compared with the operator `<`.
- Design the method `sort()` that sorts the list according to the `<` operator. Consider merge sort and quick sort as possible algorithms.
- Extend the previous methods with the compare function as a parameter of each method.

Higher-order functions

- A higher-order function is a function that can receive other functions as parameters or return a function as a result.
- Most languages support higher-order functions (C++, python, R, Haskell, Java, JavaScript, ...).
- They have different applications:
 - **sort** in STL is a higher-order function (the compare function is a parameter).
 - functions to visit the elements of containers (lists, trees, etc.) can be passed as parameters.
 - Mathematics: functions for composition and integration receive a function as parameter.
 - etc...

Higher-order functions: example

```
template <typename T>
class List {
...
/** Transforms every element of the list using f.
    It returns a reference to the list. */
List<T>& transform(void f(T&));

/** Returns a list with the elements for which f is true */
List<T> filter(bool f(const T&)) const;

/** Applies f sequentially to the list and returns a
    single value. For the list  $[x_1, x_2, x_3, \dots, x_n]$  it returns
     $f(\dots f(f(x_1, x_2), x_3) \dots, x_n)$ . If the list has one element,
    it returns  $x_1$ . The list is assumed to be non-empty */
T reduce(T f(const T&, const T&)) const;
}
```

Higher-order functions: example

```
/** Checks whether a number is prime */
bool isPrime(int n) {...}

/** Adds two numbers */
int add(int x, int y) {
    return x + y;
}

/** Returns the square of a number */
int square(int x) {
    return x*x;
}
```

/** The following code computes:

$$\sum_{x \in L, x \text{ is prime}} x^2$$

Note: it assumes that there is at least one prime in the list.
*/

```
int n = L.filter(isPrime).transform(square).reduce(add);
```

Higher-order functions: example

```
List<T>& transform(void f(T&)) {
    Node* p = sentinel->next;
    while (p != sentinel) { // Visit all elements and apply f to each one
        f(p->elem);
        p = p->next;
    }
    return *this;
}

List<T> filter(bool f(const T&)) const {
    List<T> L;
    Node* p = sentinel->next;
    while (p != sentinel) { // Pick elements only if f is asserted
        if (f(p->elem)) L.insert(p->elem);
        p = p->next;
    }
    return L;
}

T reduce(T f(const T&, const T&)) const {
    assert(L.size() > 0);
    T x = sentinel->next->elem; // First element (and result)
    Node* p = sentinel->next->next;
    while (p != sentinel) {
        x = f(x, p->elem); // Composition with next element
        p = p->next;
    }
    return x;
}
```

Containers: Priority Queues

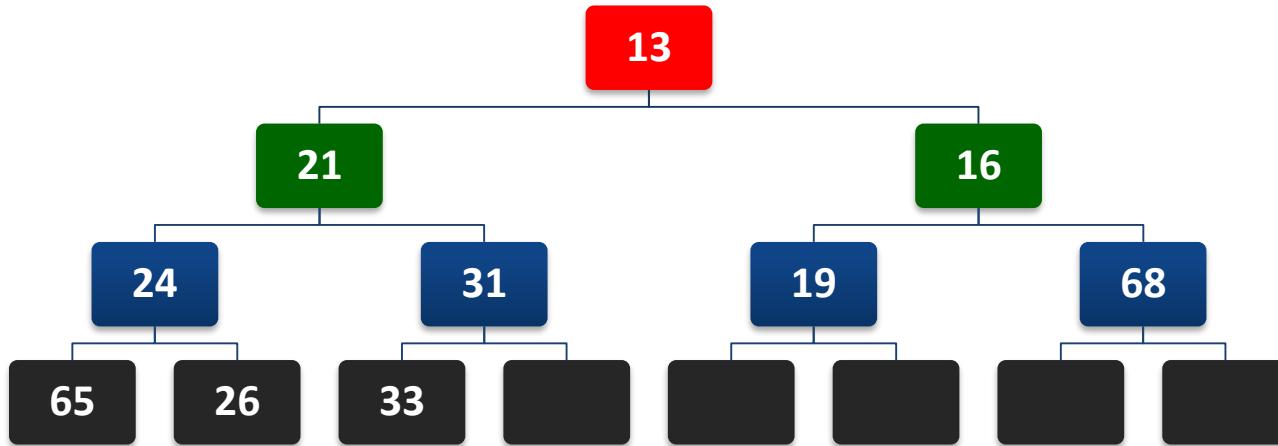


Jordi Cortadella and Jordi Petit
Department of Computer Science

A priority queue

- A priority queue is a queue in which each element has a priority.
- Elements with higher priority are served before elements with lower priority.
- It can be implemented as a vector or a linked list. For a queue with n elements:
 - Insertion is $O(n)$.
 - Extraction is $O(1)$.
- A more efficient implementation can be proposed in which insertion and extraction are $O(\log n)$: ***binary heap***.

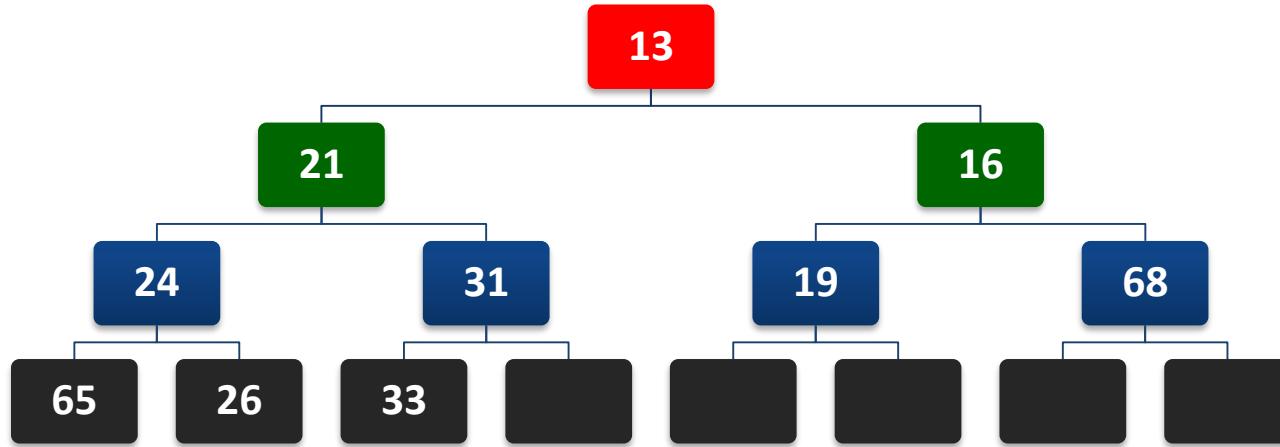
Binary Heap



- Complete binary tree (except at the bottom level).
- Height h : between 2^h and $2^{h+1} - 1$ nodes.
- For N nodes, the height is $O(\log N)$.
- It can be represented in a vector.



Binary Heap



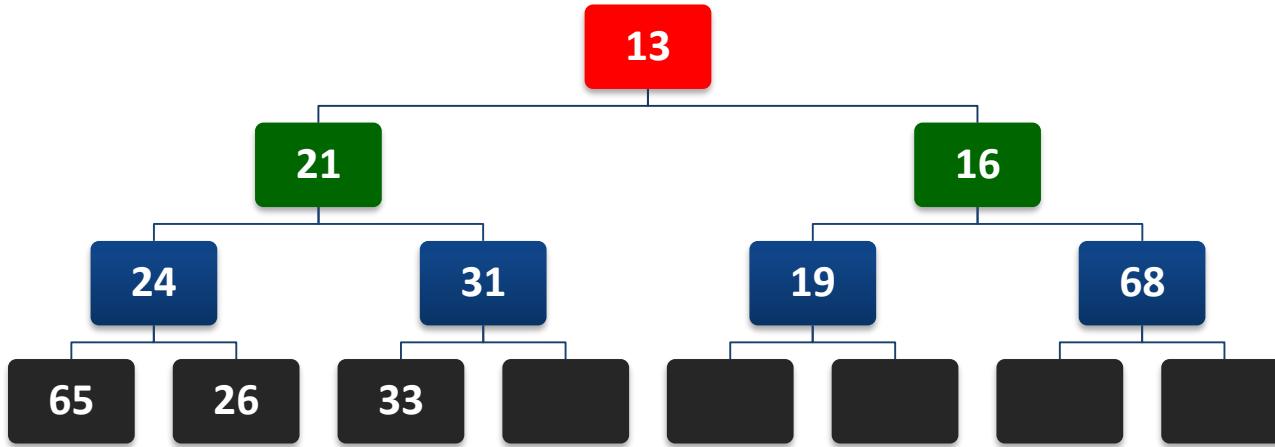
Locations in the vector:



Heap-Order Property: the key of the parent of X is smaller than (or equal to) the key in X.



Binary Heap



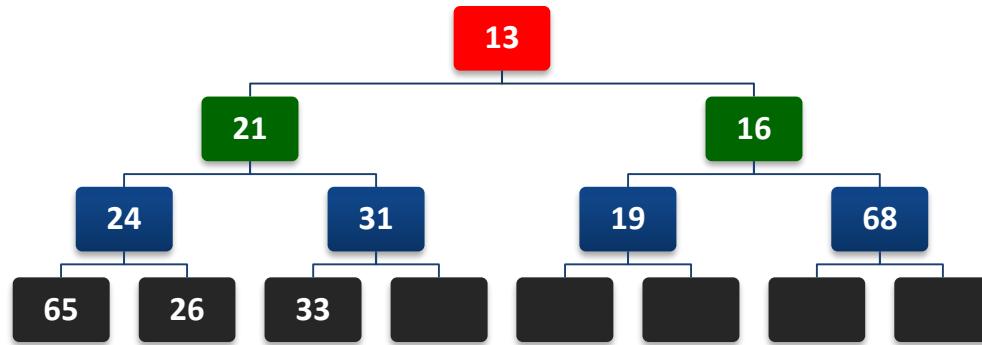
Two main operations on a binary heap:

- Insert a new element
- Remove the min element

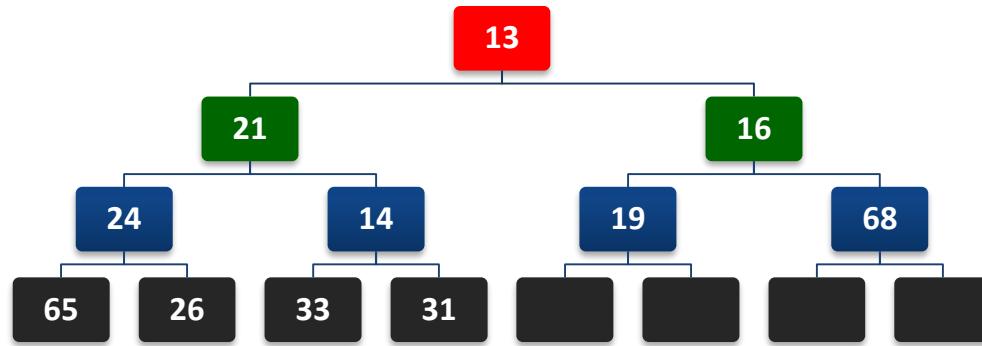
Both operations must preserve the properties of the binary heap:

- Completeness
- Heap-Order property

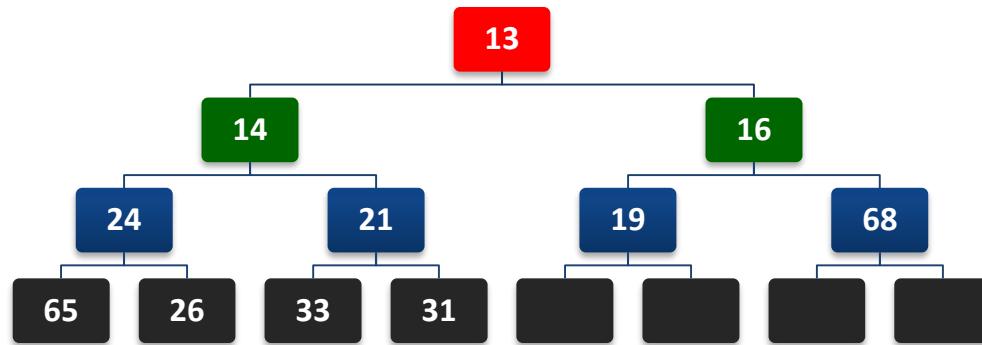
Binary Heap: insert 14



Insert in the last location

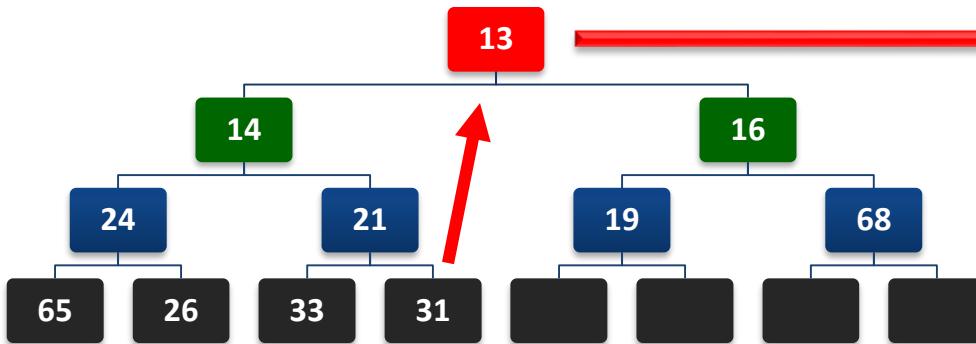


... and bubble up ...

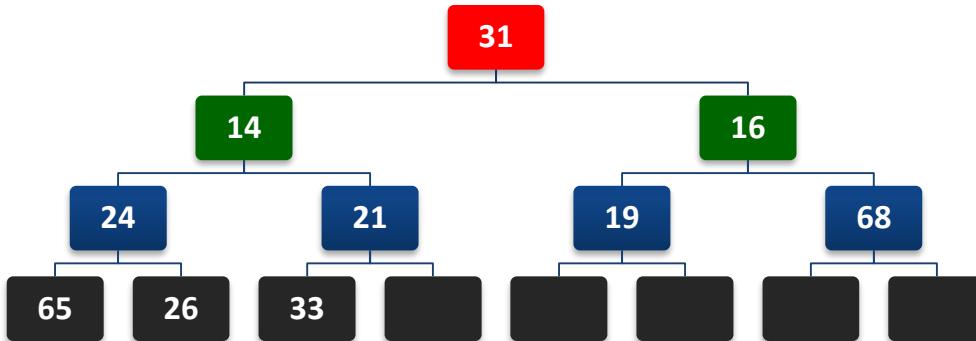


done !

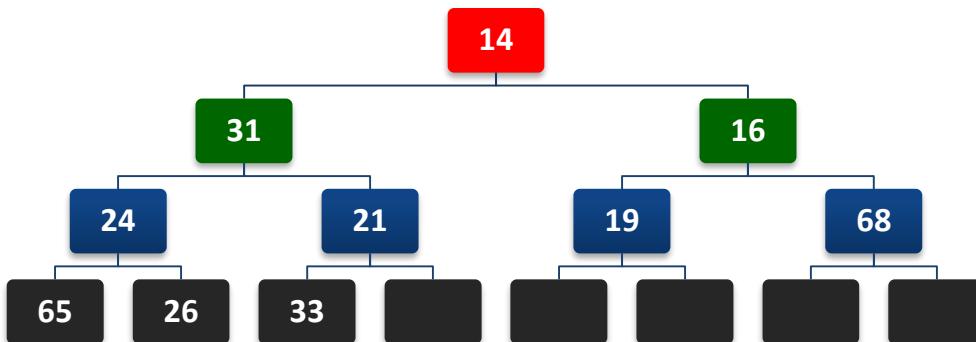
Binary Heap: remove min



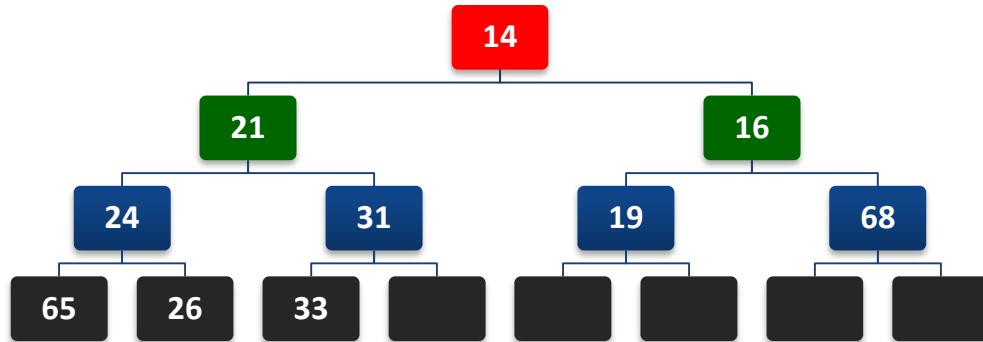
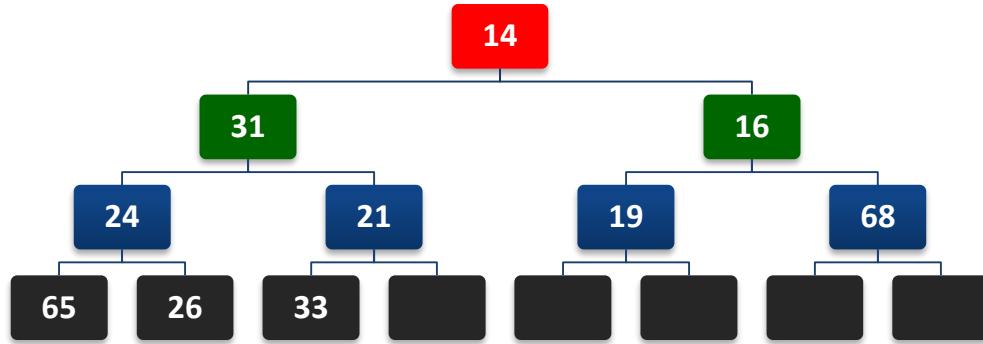
Extract the min element
and move the last one
to the root of the heap



... and bubble down ...



Binary Heap: remove min



done !

Binary Heap: implementation

```
// Elem must be a comparable type
template <typename Elem>
class PriorityQueue {
private:
    vector<Elem> v; // Table for the heap (location 0 not used)

public:

    // Constructor (one fake element in the vector)
    PriorityQueue() {
        v.push_back(Elem());
    }

    /** Number of elements in the queue */
    int size() {
        return v.size() - 1; // The 0 location does not count
    }

    /** Checks whether the queue is empty */
    bool empty() {
        return size() == 0;
    }
}
```

Binary Heap: implementation

```
public:  
    /** Returns the min element of the queue */  
    Elem minimum() {  
        assert(not empty());  
        return v[1];  
    }  
  
    /** Inserts an element into the queue */  
    void insert(const Elem& x) {  
        v.push_back(x); // Put element at the bottom  
        bubble_up(size()); // ... and bubble up  
    }  
  
    /** Extracts and returns the min element from the queue */  
    Elem remove_min () {  
        assert(not empty());  
        Elem x = v[1]; // Store the element at the root  
        v[1] = v.back(); // Move the last element to the root  
        v.pop_back();  
        bubble_down(1); // ... and bubble down  
        return x;  
    }
```

Binary Heap: implementation

private:

```
/** Bubbles up the element at location i */
void bubble_up(int i) {
    if (i != 1 and v[i/2] > v[i]) {
        swap(v[i], v[i/2]);
        bubble_up(i/2);
    }
}

/** Bubbles down the element at location i */
void bubble_down(int i) {
    int n = size();
    int c = 2*i;
    if (c <= n) {
        if (c+1 <= n and v[c+1] < v[c]) c++;
        if (v[i] > v[c]) {
            swap(v[i], v[c]);
            bubble_down(c);
        }
    }
}
```

Binary Heap: complexity

- Bubble up/down operations do at most h swaps, where h is the height of the tree and

$$h = \lfloor \log_2 N \rfloor$$

- Therefore:
 - Getting the min element is $O(1)$
 - Inserting a new element is $O(\log N)$
 - Removing the min element is $O(\log N)$

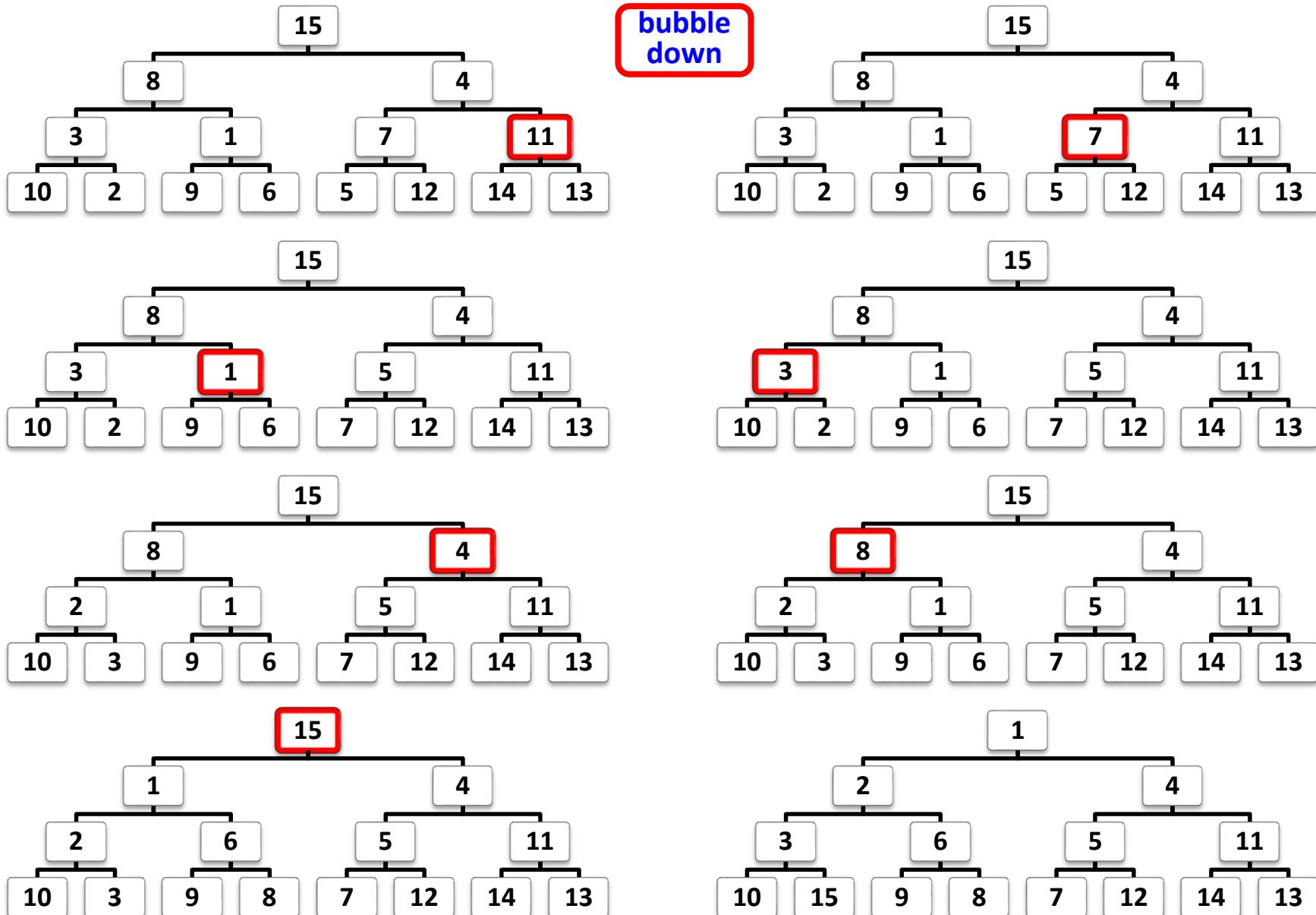
Binary Heap: other operations

- Let us assume that we have a method to know the location of every key in the heap.
- Increase/decrease key:
 - Modify the value of one element in the middle of the heap.
 - If decreased → bubble up.
 - If increased → bubble down.
- Remove one element:
 - Set value to $-\infty$, bubble up and remove min element.

Building a heap from a set of elements

- Heaps are sometimes constructed from an initial collection of N elements. How much does it cost to create the heap?
 - Obvious method: do N insert operations.
 - Complexity: $O(N \log N)$
- Can it be done more efficiently?

Building a heap from a set of elements



Building a heap: implementation

```
// Constructor from a collection of items
PriorityQueue(const vector<Elem>& items) {
    v.push_back(Elem());
    for (auto& e: items) v.push_back(e);
    for (int i = size()/2; i > 0; --i) bubble_down(i);
}
```

Sum of the heights of all nodes:

- 1 node with height h
- 2 nodes with height $h - 1$
- 4 nodes with height $h - 2$
- 2^i nodes with height $h - i$

$$S = \sum_{i=0}^h 2^i(h - i)$$

$$S = h + 2(h - 1) + 4(h - 2) + 8(h - 3) + 16(h - 4) + \dots + 2^{h-1}(1)$$

$$2S = 2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + \dots + 2^h(1)$$

Subtract the two equations:

$$S = -h + 2 + 4 + 8 + \dots + 2^{h-1} + 2^h = (2^{h+1} - 1) - (h + 1) = O(N)$$

A heap can be built from a collection of items in linear time.

Heap sort

```
template <typename T>
void heapSort(vector<T>& v) {
    PriorityQueue<T> heap(v);
    for (T& e: v) e = heap.remove_min();
}
```

- Complexity: $O(n \log n)$
 - Building the heap: $O(n)$
 - Each removal is $O(\log n)$, executed n times.

Exercises

- Given the binary heap implemented in the following vector, draw the tree represented by the vector and the corresponding vector after inserting the element 3 and removing the minimum:

7	11	9	23	41	27	12	29
---	----	---	----	----	----	----	----

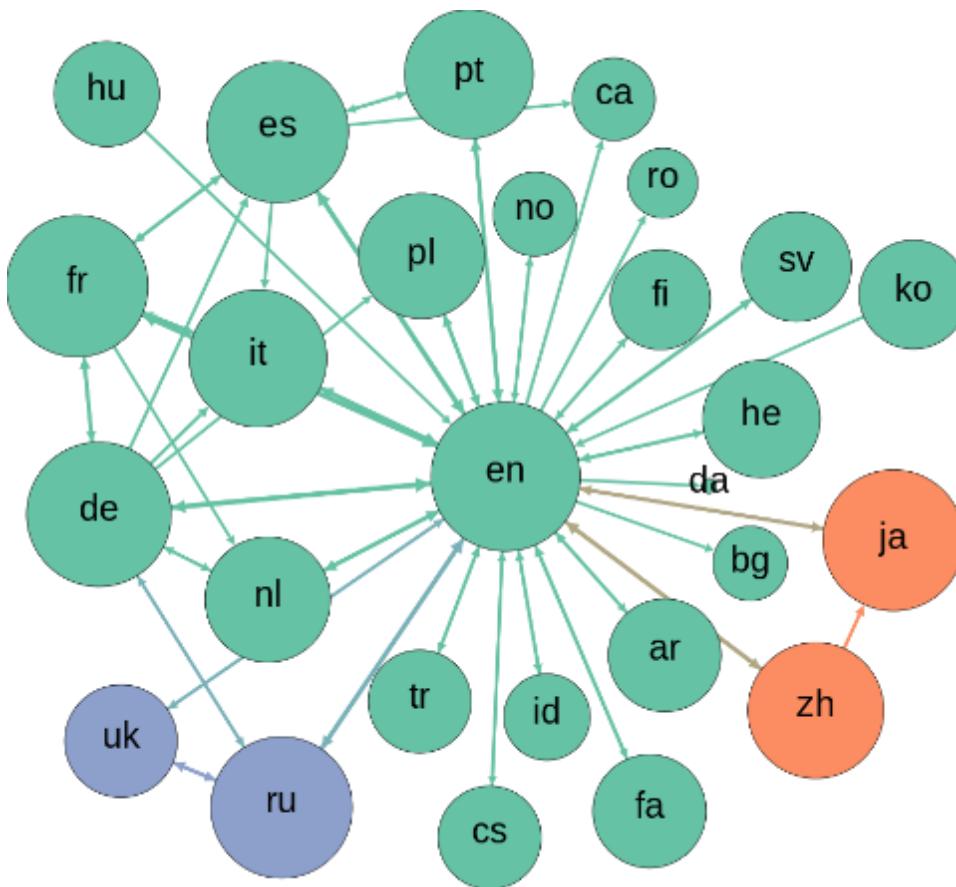
- The k -th element of n sorted vectors.** Let us consider n vectors sorted in ascending order. Design an algorithm with cost $\Theta(k \log n + n)$ that finds the k -th global smallest element.

Graphs: Connectivity



Jordi Cortadella and Jordi Petit
Department of Computer Science

A graph



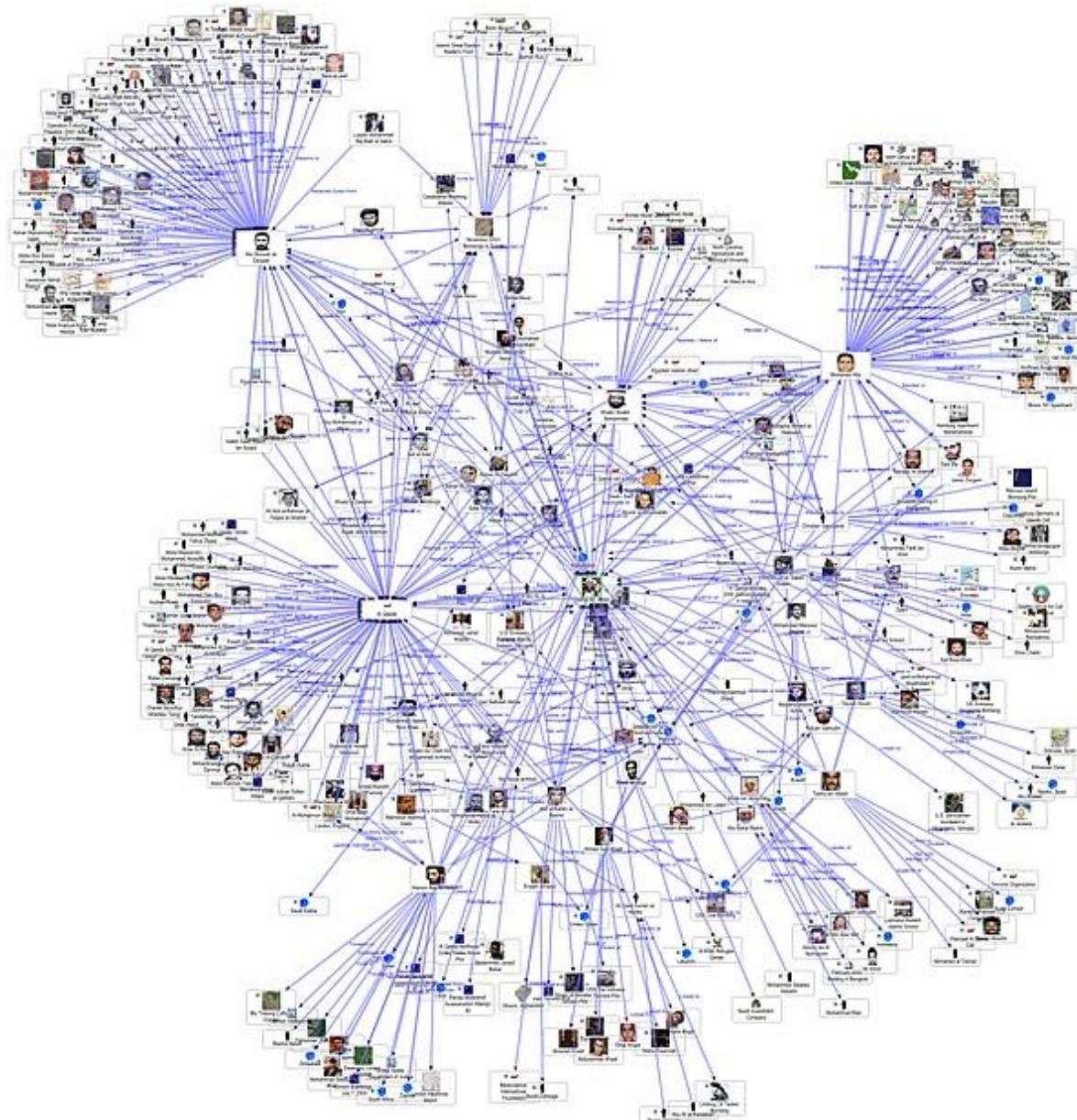
Source: Wikipedia

The network graph formed by Wikipedia editors (edges) contributing to different Wikipedia language versions (vertices) during one month in summer 2013

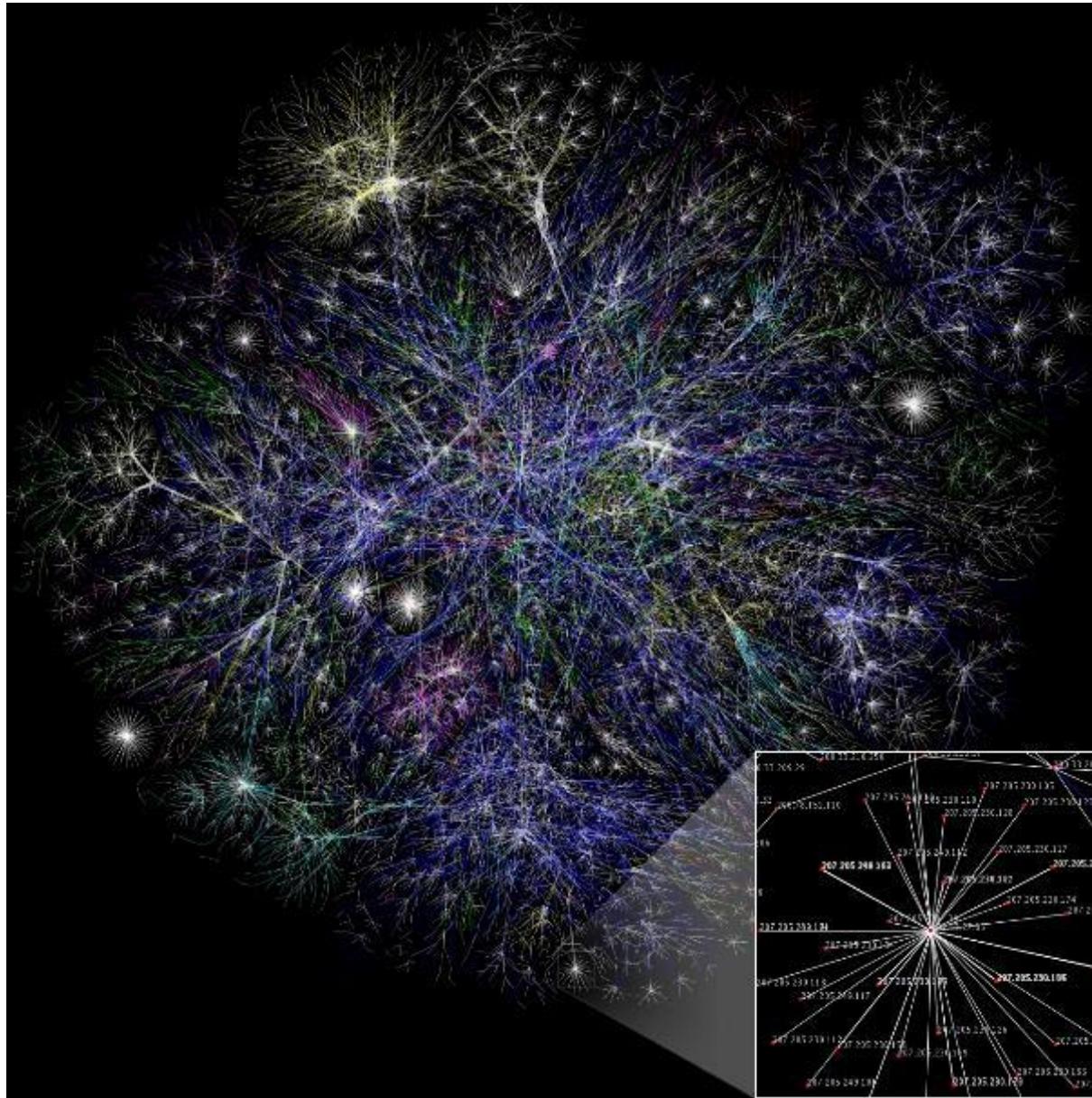
Transportation systems



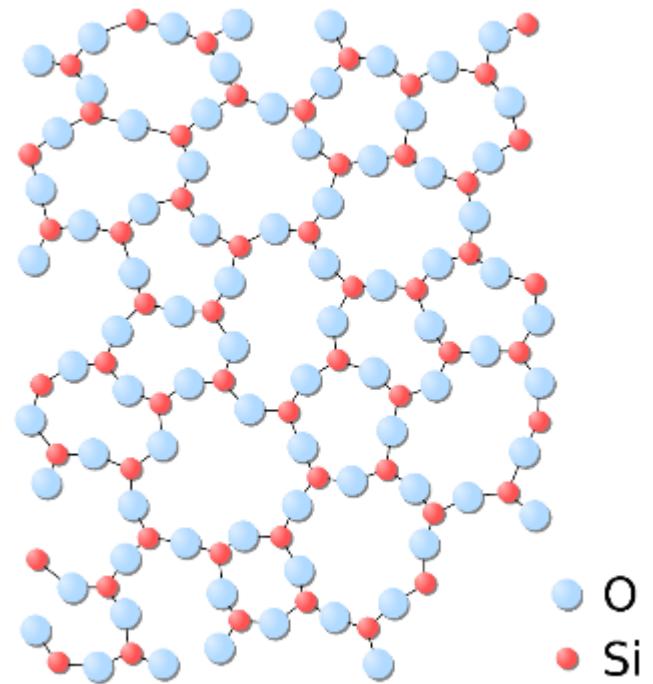
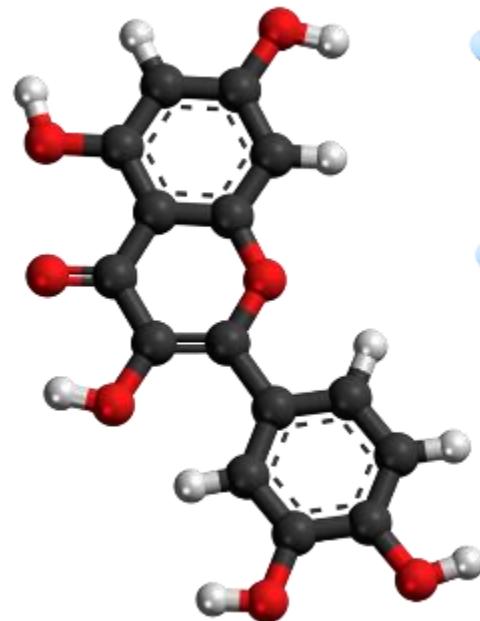
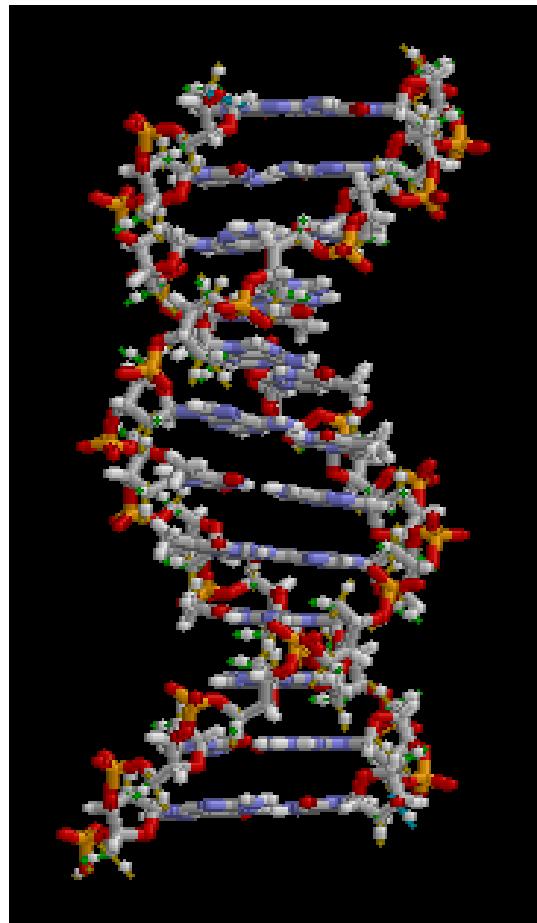
Social networks



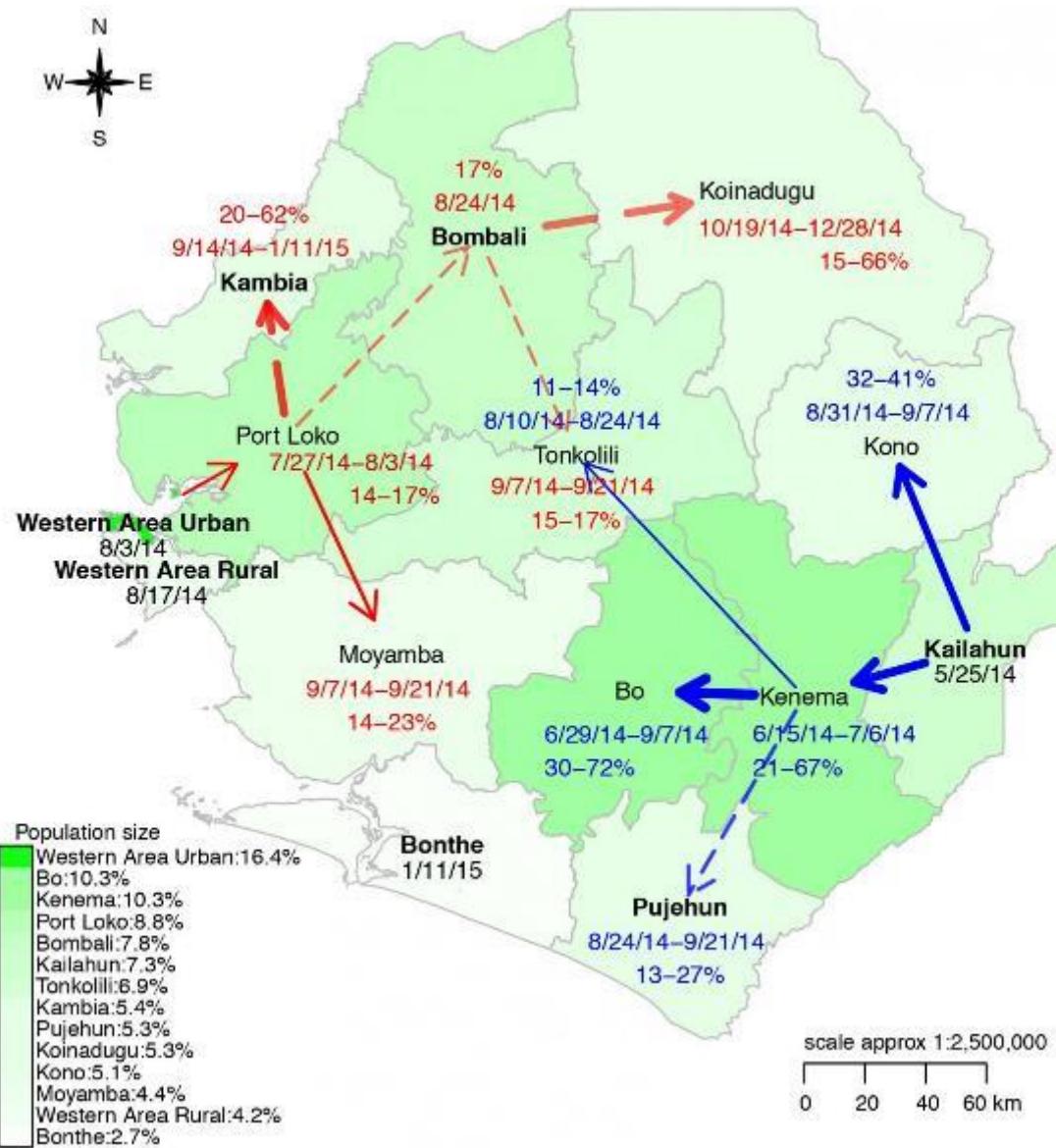
World Wide Web



Biology

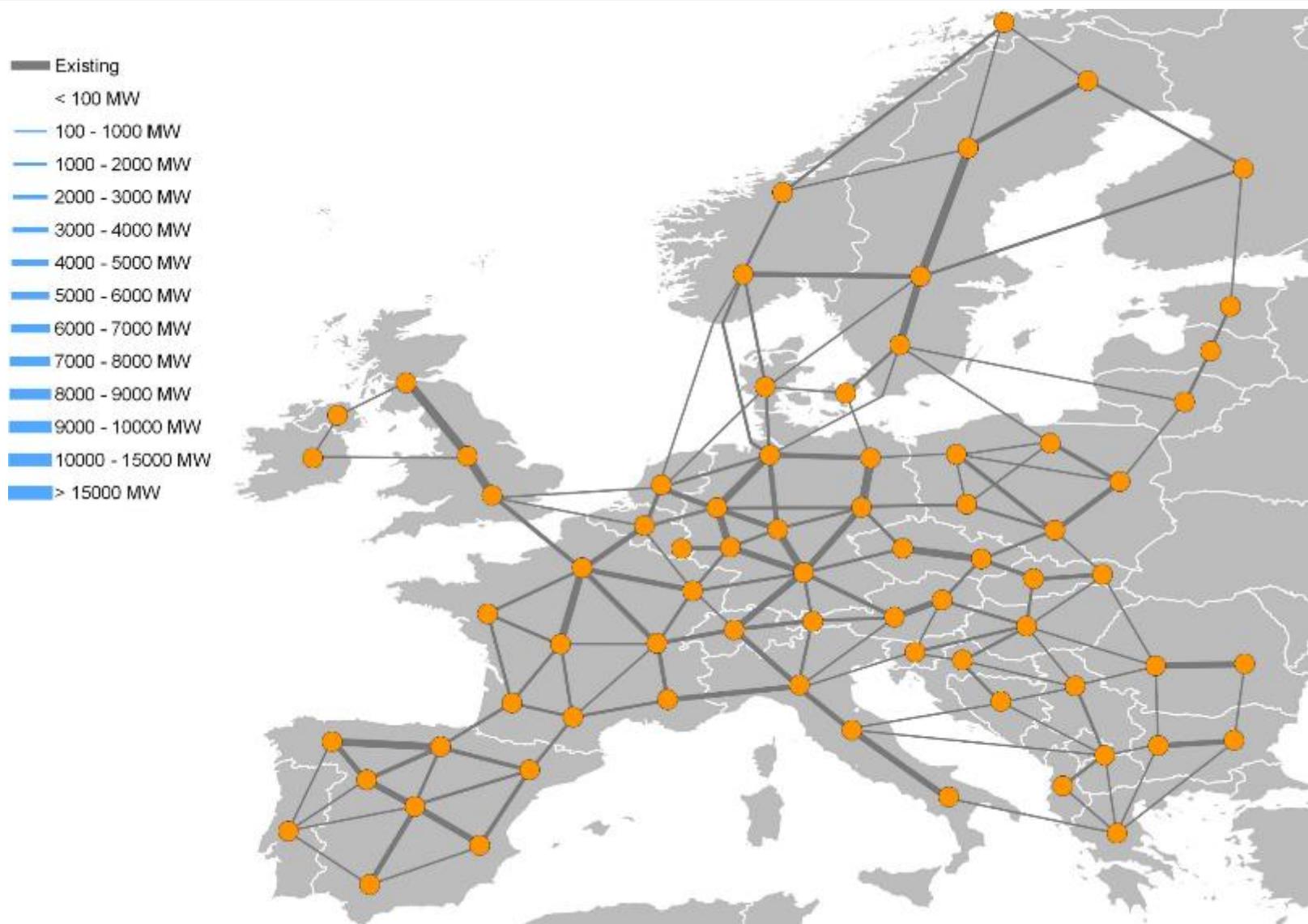


Disease transmission network



<https://medicalxpress.com/news/2015-11-reveals-deadly-route-ebola-outbreak.html>

Transmission of renewable energy



Topology of regional transmission grid model of continental Europe in 2020
<https://blogs.dnvgi.com/energy/integration-of-renewable-energy-in-europe>

What would we like to solve on graphs?

- Finding paths: which is the shortest route from home to my workplace?
- Flow problems: what is the maximum amount of people that can be transported in Barcelona at rush hours?
- Constraints: how can we schedule the use of the operating room in a hospital to minimize the length of the waiting list?
- Clustering: can we identify groups of friends by analyzing their activity in twitter?

Credits

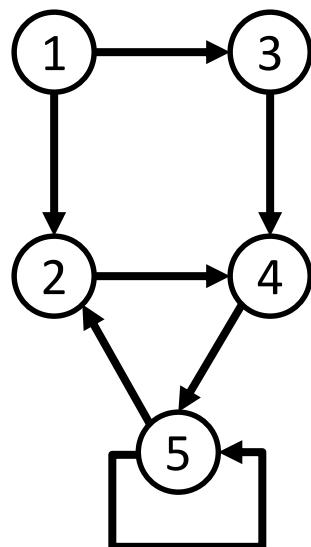
A significant part of the material used in this chapter has been inspired by the book:

Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani,
Algorithms, McGraw-Hill, 2008. [DPV2008]

(several examples, figures and exercises are taken from the book)

Graph definition

A graph is specified by a set of vertices (or nodes) V and a set of edges E .



$$V = \{1,2,3,4,5\}$$

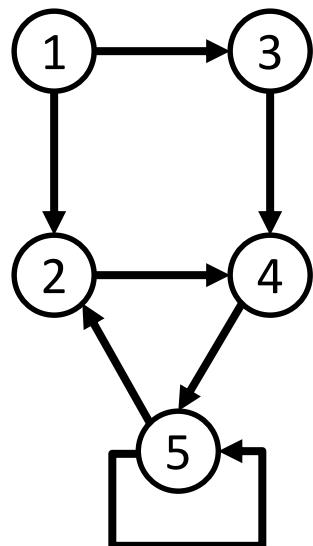
$$E = \{(1,2), (1,3), (2,4), (3,4), (4,5), (5,2), (5,5)\}$$

Graphs can be directed or undirected.
Undirected graphs have a symmetric relation.

Graph representation: adjacency matrix

A graph with $n = |V|$ vertices, v_1, \dots, v_n , can be represented by an $n \times n$ matrix with:

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$



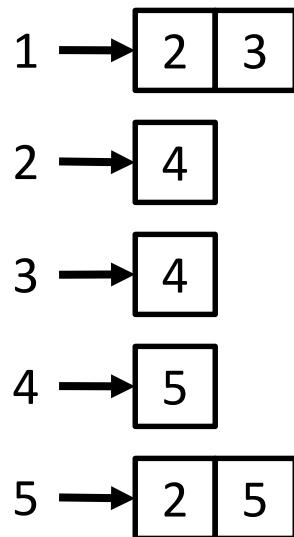
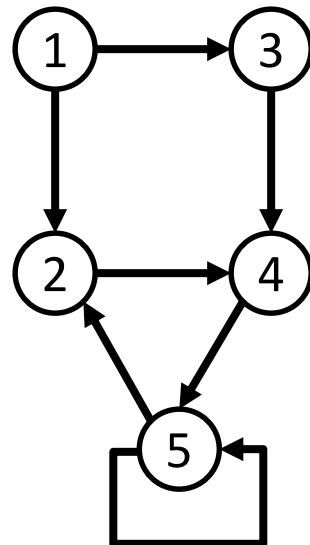
$$a = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Space: $O(n^2)$

For undirected graphs, the matrix is symmetric.

Graph representation: adjacency list

A graph can be represented by $|V|$ lists, one per vertex. The list for vertex u holds the vertices connected to the outgoing edges from u .



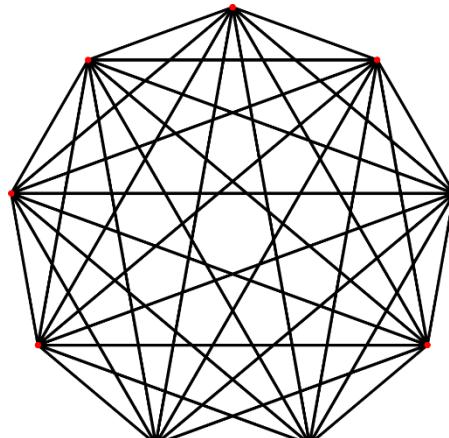
The lists can be implemented in different ways (vectors, linked lists, ...)

Space: $O(|E|)$

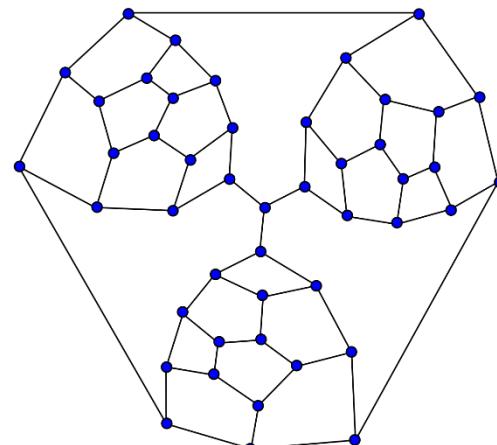
Undirected graphs: use bi-directional edges

Dense and sparse graphs

- A graph with $|V|$ vertices could potentially have up to $|V|^2$ edges (all possible edges are possible).
- We say that a graph is **dense** when $|E|$ is close to $|V|^2$. We say that a graph is **sparse** when $|E|$ is close to $|V|$.
- How big can a graph be?

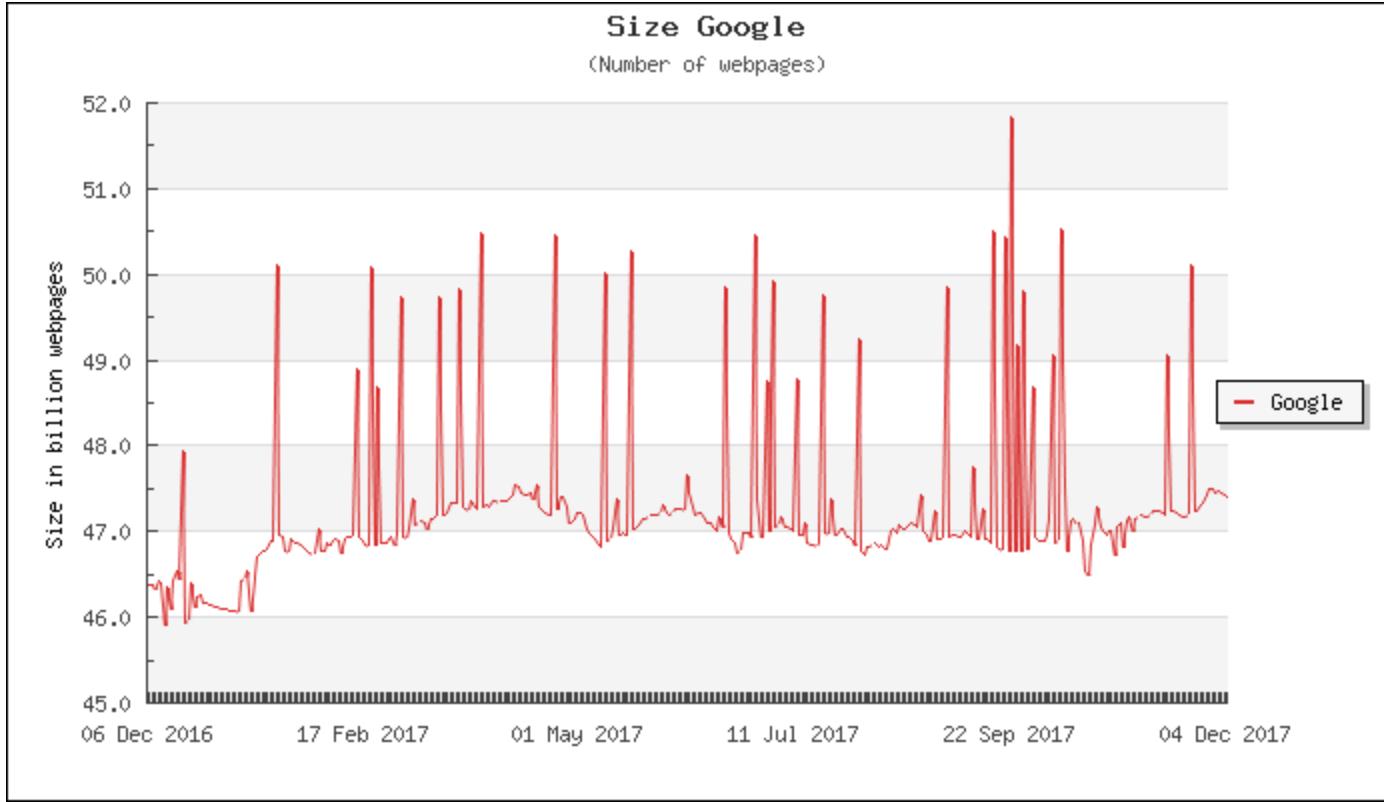


Dense graph



Sparse graph

Size of the World Wide Web



www.worldwidewebsize.com

- December 2017: 50 billion web pages (50×10^9).
- Size of adjacency matrix: 25×10^{20} elements.
(not enough computer memory in the world to store it).
- Good news: The web is very sparse. Each web page has about half a dozen hyperlinks to other web pages.

Adjacency matrix vs. adjacency list

- Space:
 - Adjacency matrix is $O(|V|^2)$
 - Adjacency list is $O(|E|)$
- Checking the presence of a particular edge (u, v) :
 - Adjacency matrix: constant time
 - Adjacency list: traverse u 's adjacency list
- Which one to use?
 - For dense graphs → adjacency matrix
 - For sparse graphs → adjacency list
- For many algorithms, traversing the adjacency list is not a problem, since they require to iterate through all neighbors of each vertex. For sparse graphs, the adjacency lists are usually short (can be traversed in constant time)

Graph usage: example

```
// Declaration of a graph that stores  
// a string (name) for each vertex
```

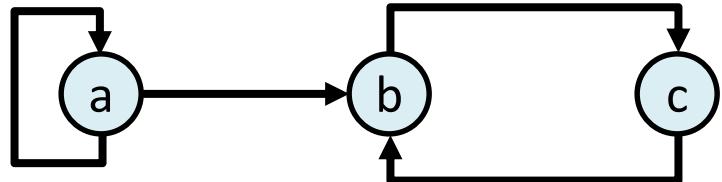
```
Graph<string> G;
```

```
// Create the vertices
```

```
int a = G.addVertex("a");  
int b = G.addVertex("b");  
int c = G.addVertex("c");
```

```
// Create the edges
```

```
G.addEdge(a,a);  
G.addEdge(a,b);  
G.addEdge(b,c);  
G.addEdge(c,b);
```



```
// Print all edges of the graph
```

```
for (int src = 0; src < G.numVertices(); ++src) { // all vertices  
    for (auto dst: G.succ(src)) { // all successors of src  
        cout << G.info(src) << " -> " << G.info(dst) << endl;  
    }  
}
```

	info	succ	pred
0	"a"	{0,1}	{0}
1	"b"	{2}	{0,2}
2	"c"	{1}	{1}

Graph implementation

```
template<typename vertexType>
class Graph {
private:
    struct Vertex {
        vertexType info; // Information of the vertex
        vector<int> succ; // List of successors
        vector<int> pred; // List of predecessors
    };
    vector<Vertex> vertices; // List of vertices

public:
    /** Constructor */
    Graph() {}

    /** Adds a vertex with information associated to the vertex.
     * Returns the index of the vertex */
    int addVertex(const vertexType& info) {
        vertices.push_back(Vertex{info});
        return vertices.size() - 1;
    }
}
```

Graph implementation

```
/** Adds an edge src → dst */
void addEdge(int src, int dst) {
    vertices[src].succ.push_back(dst);
    vertices[dst].pred.push_back(src);
}

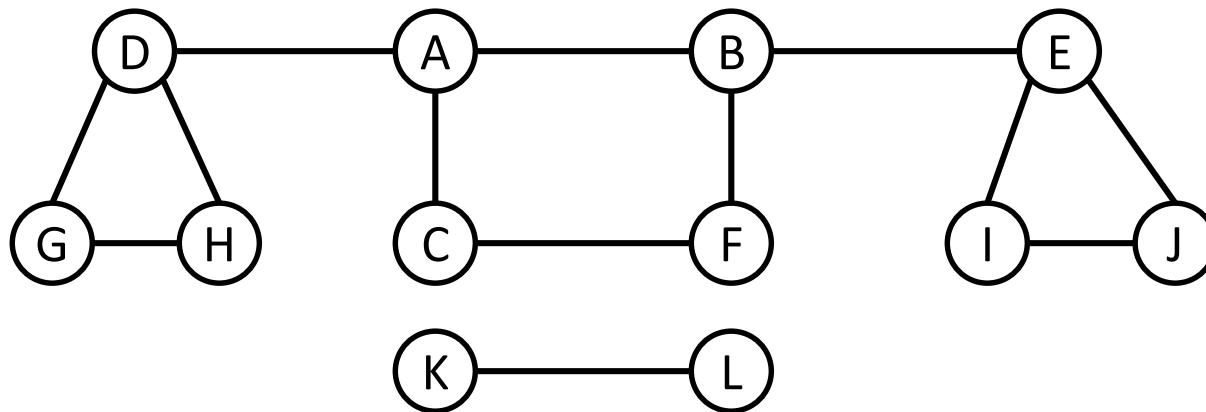
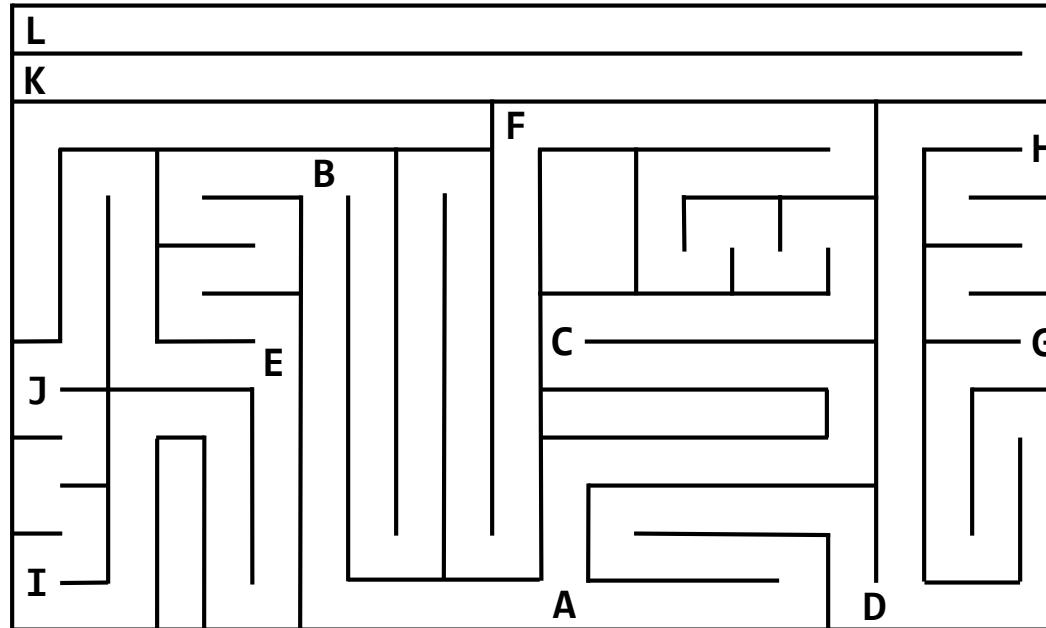
/** Returns the number of vertices of the graph */
int numVertices() {
    return vertices.size();
}

/** Returns the information associated to vertex v */
const vertexType& info(int v) const {
    return vertices[v].info;
}

/** Returns the list of successors of vertex v */
const vector<int>& succ(int v) const {
    return vertices[v].succ;
}

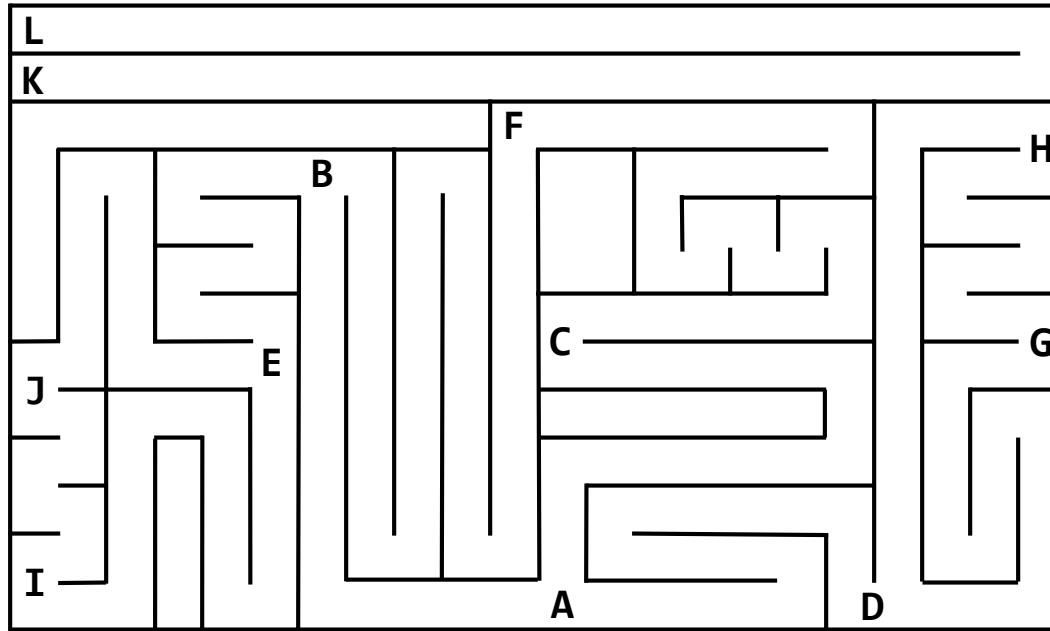
/** Returns the list of predecessors of vertex v */
const vector<int>& pred(int v) const {
    return vertices[v].pred;
}
};
```

Reachability: exploring a maze



Which vertices of the graph are reachable from a given vertex?

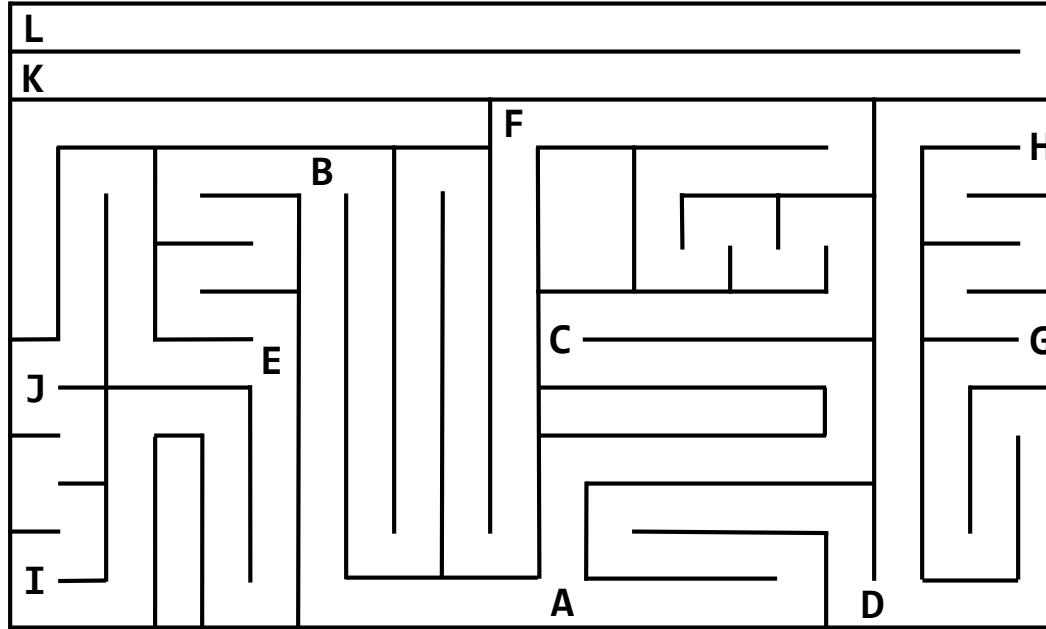
Reachability: exploring a maze



To explore a labyrinth we need a ball of string and a piece of chalk:

- The chalk prevents looping, by marking the visited junctions.
- The string allows you to go back to the starting place and visit routes that were not previously explored.

Reachability: exploring a maze



How to simulate the string and the chalk with an algorithm?

- Chalk: a boolean variable for each vertex (visited).
- String: a stack
 - push vertex to unwind at each junction
 - pop to rewind and return to the previous junction

Note: the stack can be simulated with recursion.

Finding the nodes reachable from another node

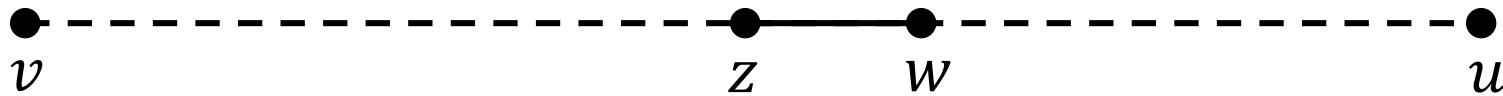
```
function explore( $G$ ,  $v$ ):  
    // Input:  $G = (V, E)$  is a graph  
    // Output: visited( $u$ ) is true for all the  
    //          nodes reachable from  $v$   
  
    visited( $v$ ) = true  
    previsit( $v$ )  
    for each edge  $(v, u) \in E$ :  
        if not visited( $u$ ): explore( $G, u$ )  
    postvisit( $v$ )
```

Note: pre/postvisit functions are not required now.

Finding the nodes reachable from another node

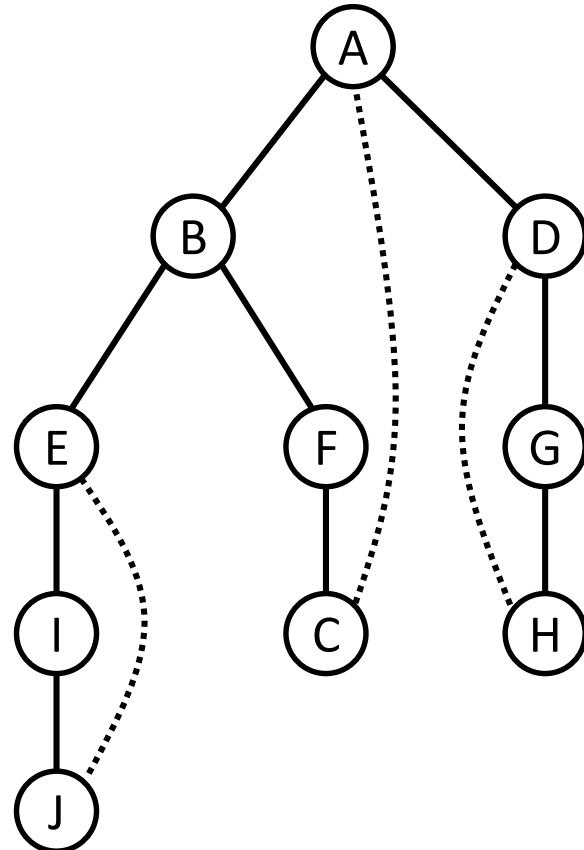
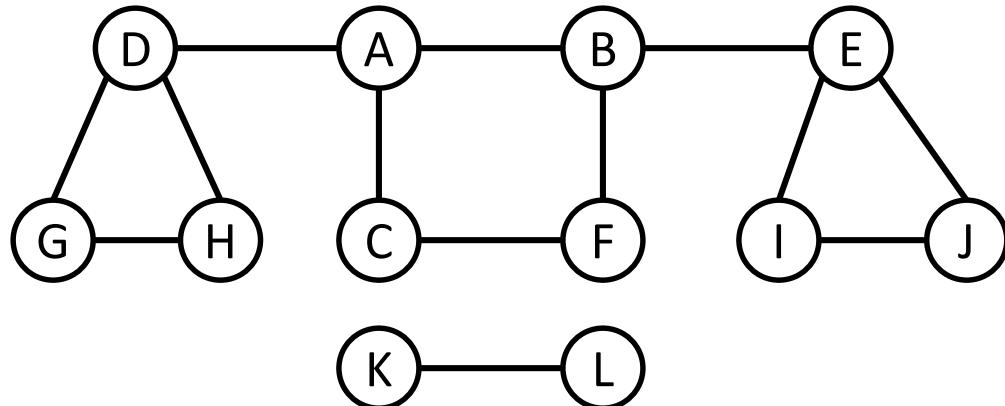
```
function explore( $G, v$ ):  
    visited( $v$ ) = true  
    for each edge  $(v, u) \in E$ :  
        if not visited( $u$ ): explore( $G, u$ )
```

- All visited nodes are reachable because the algorithm only moves to neighbors and cannot jump to an unreachable region.
- Does it miss any reachable vertex? No. Proof by contradiction.
 - Assume that a vertex u is missed.
 - Take any path from v to u and identify the last vertex that was visited on that path (z). Let w be the following node on the same path. Contradiction: w should have also been visited.



Finding the nodes reachable from another node

```
function explore(G, v):
    visited(v) = true
    for each edge (v,u) ∈ E:
        if not visited(u): explore(G,u)
```



Dotted edges are ignored (*back edges*): they lead to previously visited vertices.
The solid edges (*tree edges*) form a tree.

Depth-first search

```
function DFS( $G$ ):  
    for all  $v \in V$ :  
        visited( $v$ ) = false  
    for all  $v \in V$ :  
        if not visited( $v$ ): explore( $G, v$ )
```

DFS traverses the entire graph.

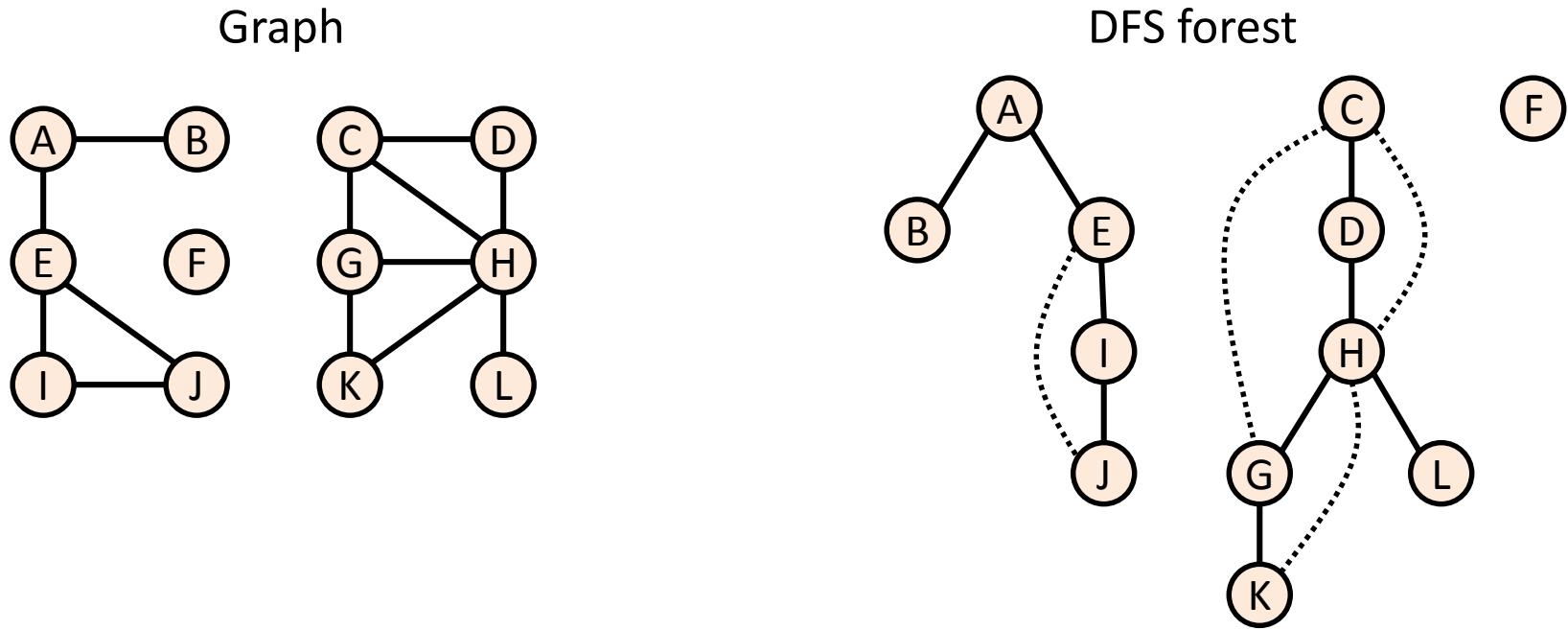
Complexity:

- Each vertex is visited only once (thanks to the chalk marks)
- For each vertex:
 - A fixed amount of work (pre/postvisit)
 - All adjacent edges are scanned

Running time is $O(|V| + |E|)$.

Difficult to improve: reading a graph already takes $O(|V| + |E|)$.

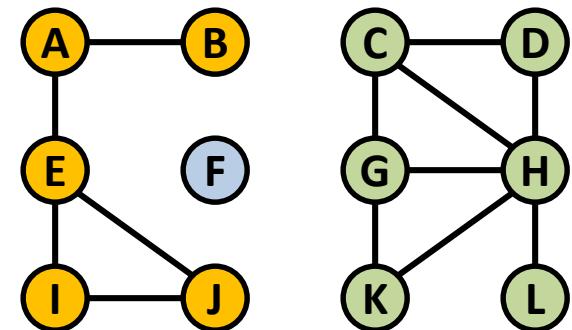
DFS example



- The outer loop of DFS calls *explore* three times (for A, C and F)
- Three trees are generated. They constitute a *forest*.

Connectivity

- An undirected graph is connected if there is a path between any pair of vertices.
- A disconnected graph has disjoint *connected components*.
- Example: this graph has 3 connected components:



$$\{A, B, E, I, J\} \quad \{C, D, G, H, K, L\} \quad \{F\}.$$

Connected Components

```
function explore(G, v, cc):
// Input: G = (V,E) is a graph, cc is a CC number
// Output: ccnum[u] = cc for each vertex u in the same CC as v
    ccnum[v] = cc
    for each edge (v,u) ∈ E:
        if ccnum[u] ≠ cc: explore(G, u, cc)

function ConnComp(G):
// Input: G = (V,E) is a graph
// Output: Every vertex v has a CC number in ccnum[v]
    for all v ∈ V: ccnum[v] = 0; // Clean cc numbers
    cc = 1; // Identifier of the first CC
    for all v ∈ V:
        if ccnum[v] = 0: // A new CC starts
            explore(G, v, cc); cc = cc + 1;
```

- Performs a DFS traversal assigning a CC number to each vertex.
- The outer loop of **ConnComp** determines the number of CC's.
- The variable $\text{ccnum}[v]$ also plays the role of $\text{visited}[v]$.

Revisiting the explore function

```
function explore(G, v):
    visited(v) = true
    previsit(v)
    for each edge (v,u) ∈ E:
        if not visited(u):
            explore(G,u)
    postvisit(v)
```

Let us consider a global variable **clock** that can determine the occurrence times of previsit and postvisit.

```
function previsit(v):
    pre[v] = clock
    clock = clock + 1

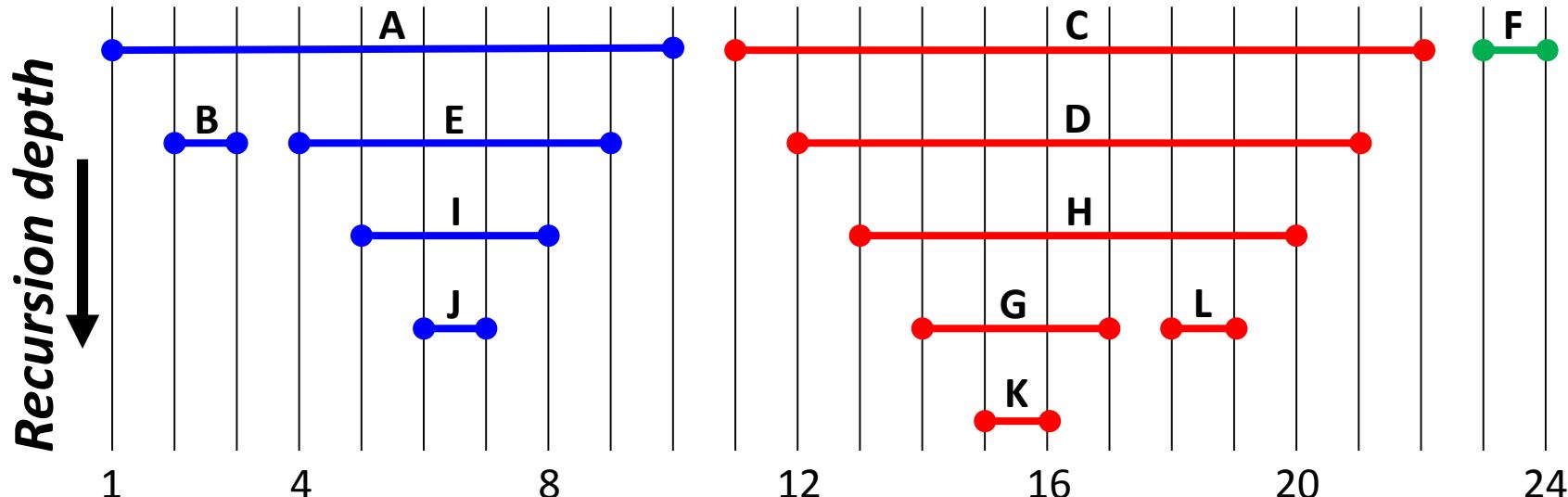
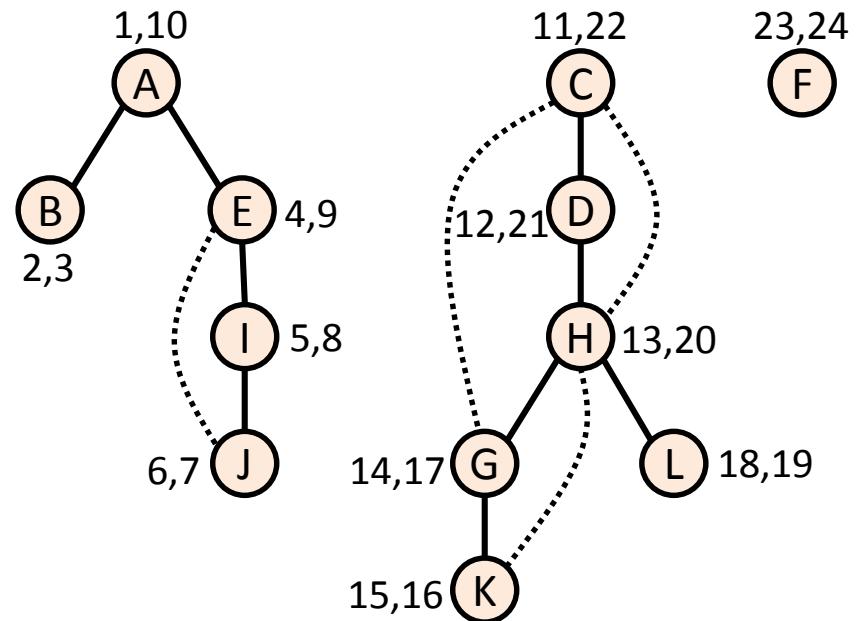
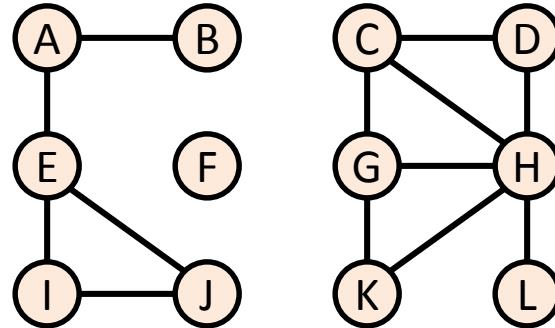
function postvisit(v):
    post[v] = clock
    clock = clock + 1
```

Every node v will have an interval $(\text{pre}[v], \text{post}[v])$ that will indicate the time the node was first visited (pre) and the time of departure from the exploration (post).

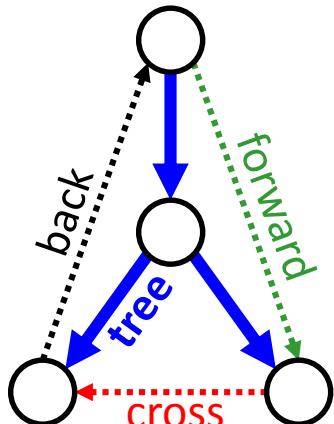
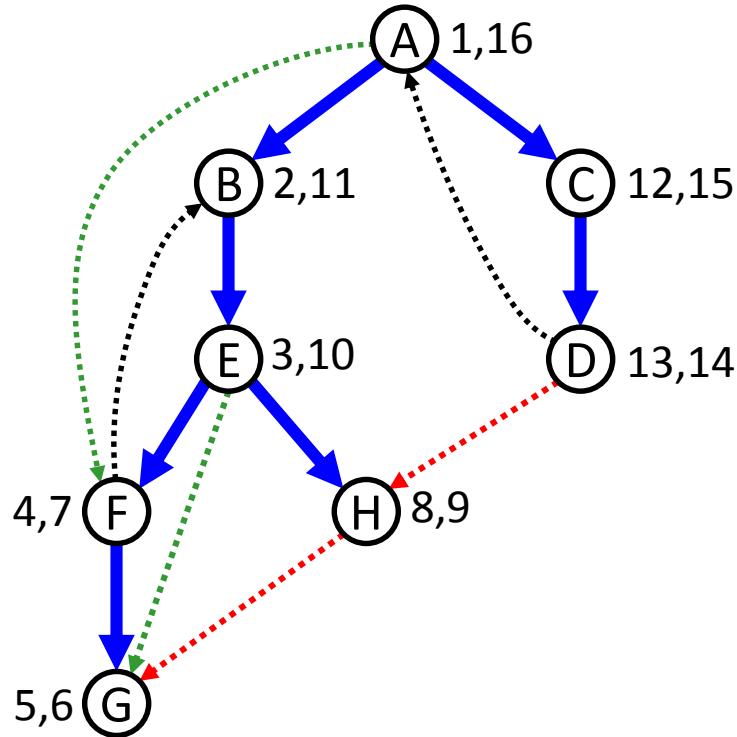
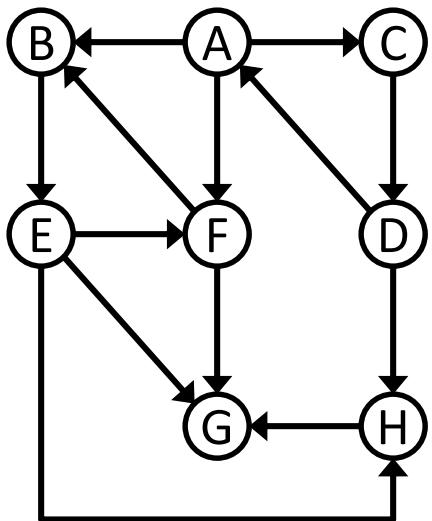
Property: Given two nodes u and v , the intervals $(\text{pre}[u], \text{post}[u])$ and $(\text{pre}[v], \text{post}[v])$ are either disjoint or one is contained within the other.

The pre/post interval of u is the lifetime of $\text{explore}(u)$ in the stack (LIFO).

Example of pre/postvisit orderings



DFS in directed graphs: types of edges



- **Tree edges:** those in the DFS forest.
- **Forward edges:** lead to a nonchild descendant in the DFS tree.
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor.

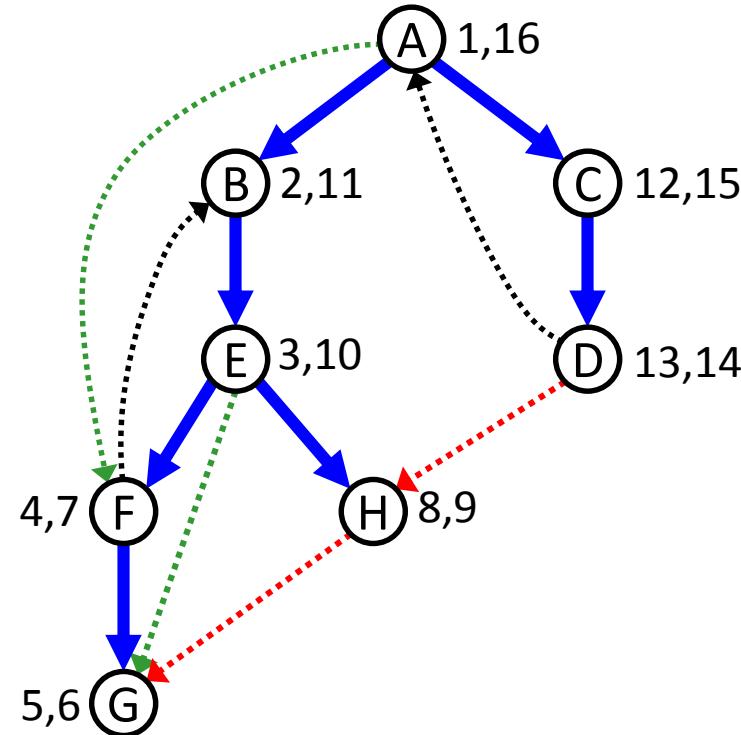
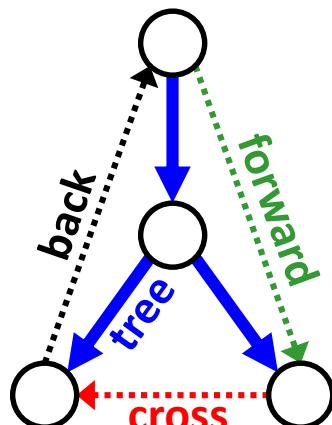
DFS in directed graphs: types of edges

pre/post ordering for (u, v)

$\left(\begin{array}{c} u \\ v \end{array} \right) \quad \left(\begin{array}{c} v \\ v \end{array} \right) \quad \left(\begin{array}{c} u \\ u \end{array} \right)$ tree/forward

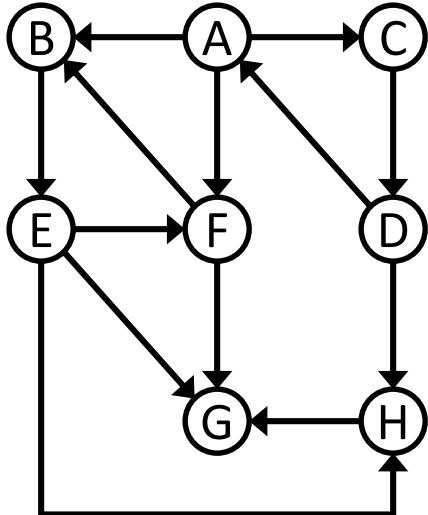
$\left(\begin{array}{c} v \\ u \end{array} \right) \quad \left(\begin{array}{c} u \\ u \end{array} \right) \quad \left(\begin{array}{c} v \\ v \end{array} \right)$ back

$\left(\begin{array}{c} v \\ v \end{array} \right) \quad \left(\begin{array}{c} u \\ u \end{array} \right) \quad \left(\begin{array}{c} u \\ u \end{array} \right)$ cross



- **Tree edges:** those in the DFS forest.
- **Forward edges:** lead to a nonchild descendant in the DFS tree.
- **Back edges:** lead to an ancestor in the DFS tree.
- **Cross edges:** lead to neither descendant nor ancestor.

Cycles in graphs



A **cycle** is a circular path:

$$v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_0.$$

Examples:

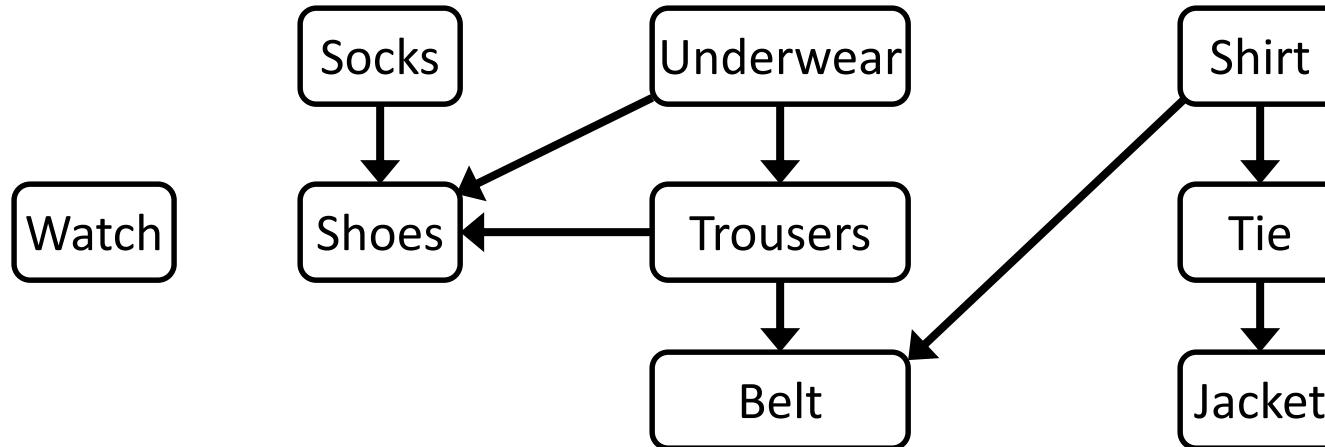
$$\begin{aligned} A &\rightarrow F \rightarrow B \rightarrow A \\ C &\rightarrow D \rightarrow A \rightarrow C \end{aligned}$$

Property: A directed graph has a cycle iff its DFS reveals a back edge.

Proof:

- ⇐ If (u, v) is a back edge, there is a cycle with (u, v) and the path from v to u in the search tree.
- ⇒ Let us consider a cycle $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k \rightarrow v_0$. Let us assume that v_i is the first discovered vertex (lowest pre number). All the other v_j on the cycle are reachable from v_i and will be its descendants in the DFS tree. The edge $v_{i-1} \rightarrow v_i$ leads from a vertex to its ancestor and is thus a back edge.

Getting dressed: DAG representation

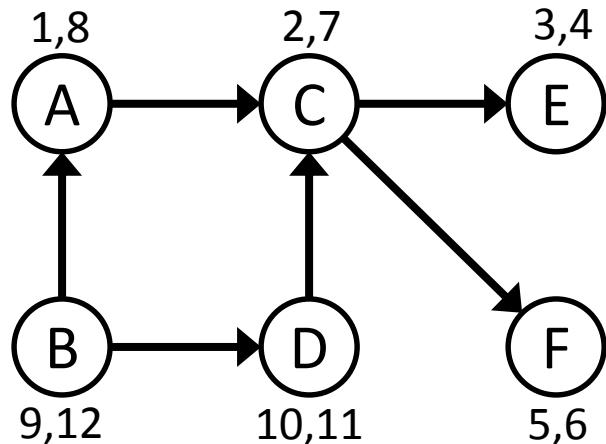


A list of tasks that must be executed in a certain order (cannot be executed if it has cycles).

Legal task *linearizations* (or *topological sorts*):



Directed Acyclic Graphs (DAGs)



A **DAG** is a directed graph without cycles.

DAGs are often used to represent causalities or temporal dependencies, e.g., task A must be completed before task C.

- Cyclic graphs cannot be linearized.
- All DAGs can be linearized. How?
 - Decreasing order of the post numbers.
 - The only edges (u, v) with $\text{post}[u] < \text{post}[v]$ are back edges (do not exist in DAGs).
- **Property:** In a DAG, every edge leads to a vertex with a lower post number.
- **Property:** Every DAG has at least one source and at least one sink.
(source: highest post number, sink: lowest post number).

Topological sort

```
function explore(G, v):
    visited(v) = true
    previsit(v)
    for each edge (v,u) ∈ E:
        if not visited(u):
            explore(G,u)
    postvisit(v)
```

```
Initially: TSort = ∅

function postvisit(v):
    TSort.push_front(v)

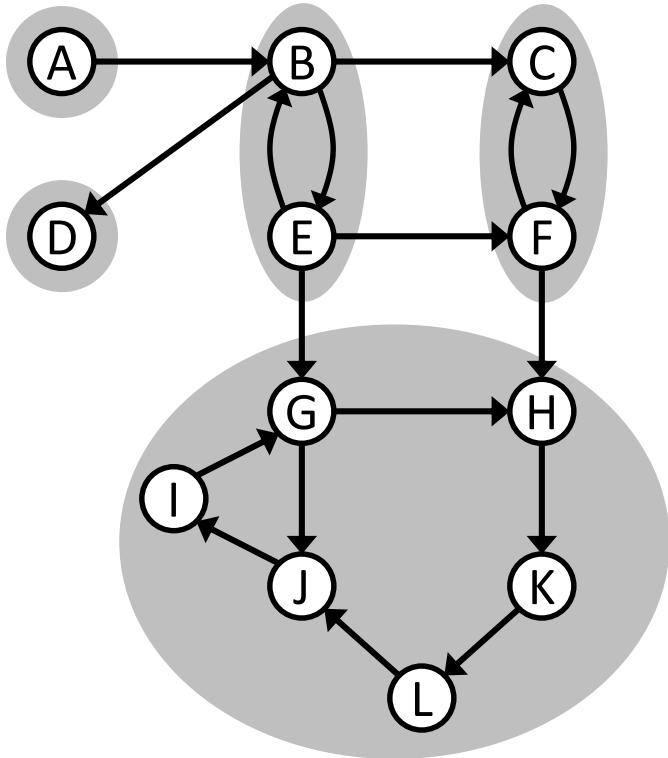
// After DFS, TSort contains
// a topological sort
```

Another algorithm:

- Find a source vertex, write it, and delete it (mark) from the graph.
- Repeat until the graph is empty.

It can be executed in linear time. How?

Strongly Connected Components



This graph is connected (undirected view), but there is no path between any pair of nodes.

For example, there is no path $K \rightarrow \dots \rightarrow C$ or $E \rightarrow \dots \rightarrow A$.

The graph is not *strongly connected*.

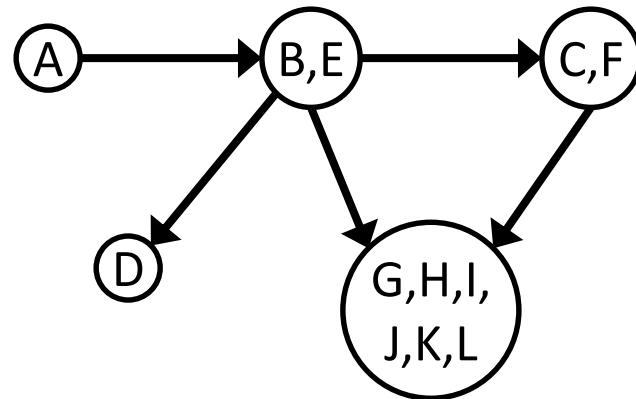
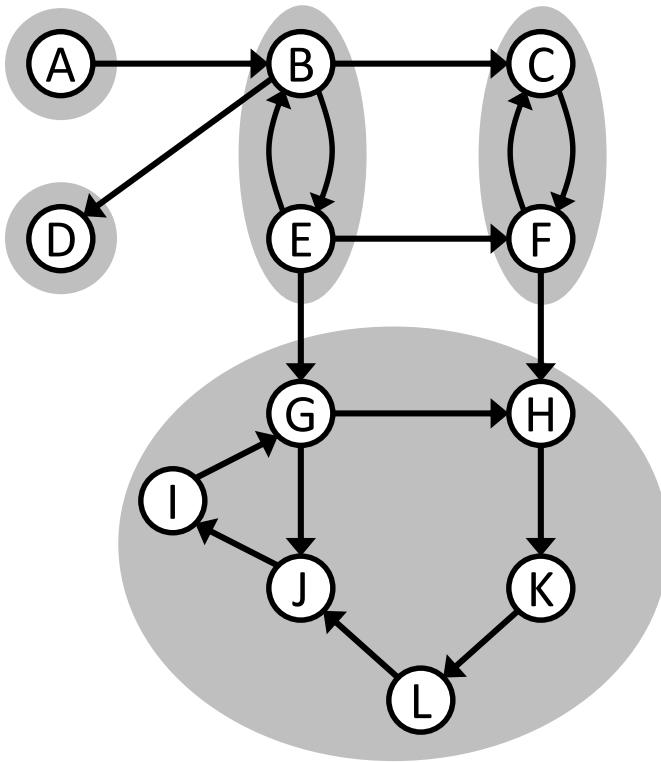
Two nodes u and v of a directed graph are connected if there is a path from u to v and a path from v to u .

The *connected* relation is an equivalence relation and partitions V into disjoint sets of *strongly connected components*.

Strongly Connected Components

- $\{A\}$
- $\{B, E\}$
- $\{C, F\}$
- $\{D\}$
- $\{G, H, I, J, K, L\}$

Strongly Connected Components



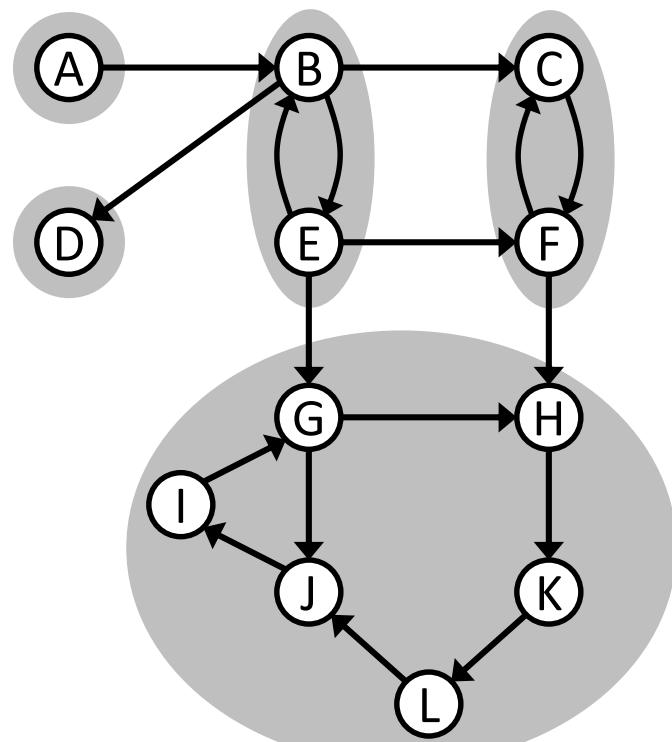
Property: every directed graph is a DAG of its strongly connected components.

A directed graph can be seen as a 2-level structure. At the top we have a DAG of SCCs. At the bottom we have the details of the SCCs.

Every directed graph can be represented by a ***meta-graph***, where each meta-node represents a strongly connected component.

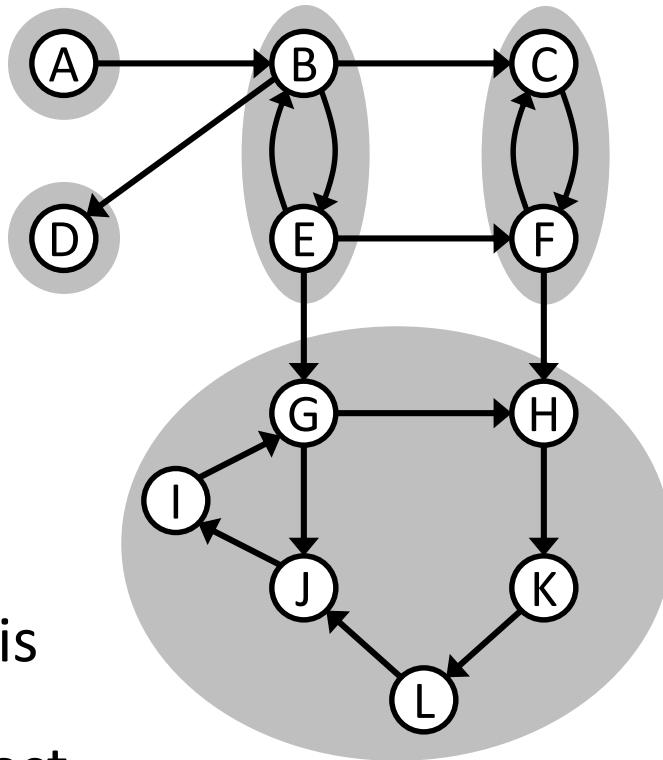
Properties of DFS and SCCs

- **Property:** If the *explore* function starts at u , it will terminate when all vertices reachable from u have been visited.
 - If we start from a vertex in a sink SCC, it will retrieve exactly that component.
 - If we start from a non-sink SCC, it will retrieve the vertices of several components.
- **Examples:**
 - If we start at K it will retrieve the component $\{G, H, I, J, K, L\}$.
 - If we start at B it will retrieve all vertices except A .

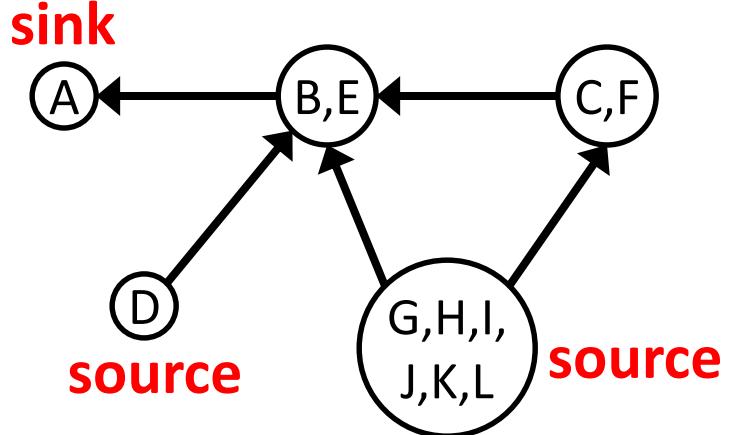
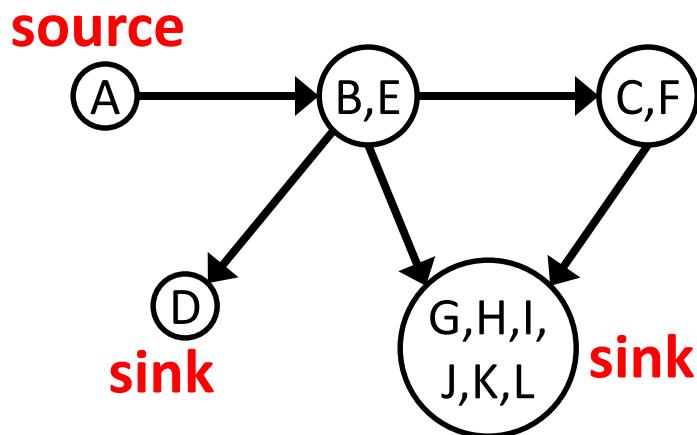
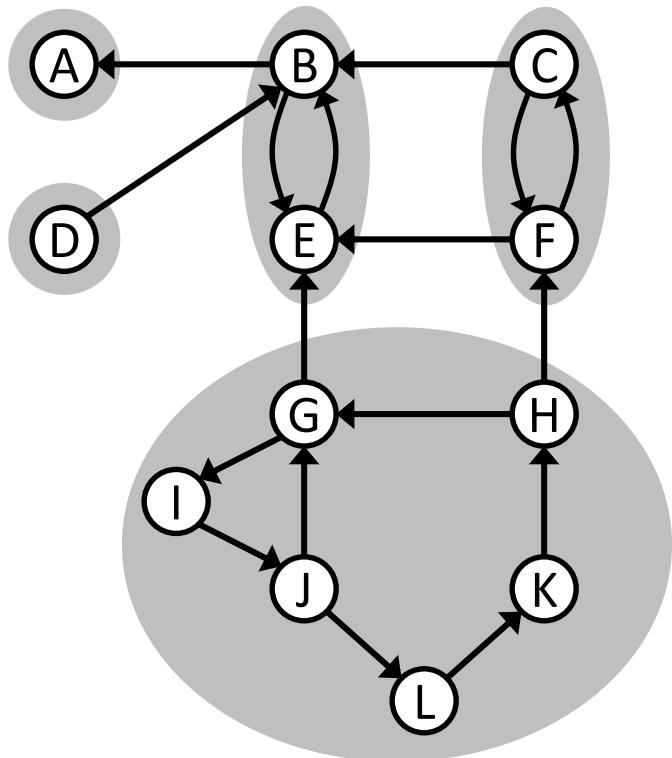
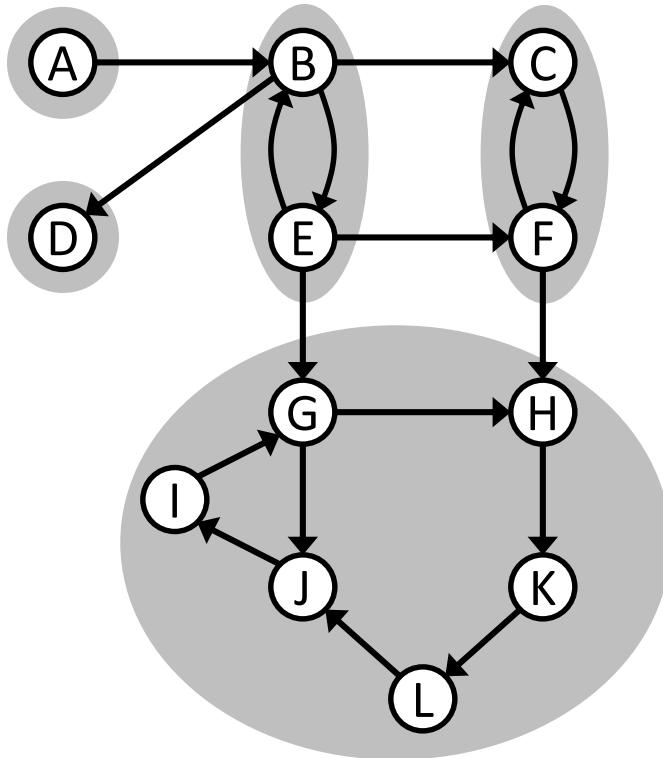


Properties of DFS and SCCs

- **Intuition for the algorithm:**
 - Find a vertex located in a sink SCC
 - Extract the SCC
- **To be solved:**
 - How to find a vertex in a sink SCC?
 - What to do after extracting the SCC?
- **Property:** If C and C' are SCCs and there is an edge $C \rightarrow C'$, then the highest post number in C is bigger than the highest post number in C' .
- **Property:** The vertex with the highest DFS post number lies in a source SCC.



Reverse graph (G^R)



SCC algorithm

```
function SCC( $G$ ):  
    // Input:  $G(V, E)$  a directed graph  
    // Output: each vertex  $v$  has an SCC number in  $\text{ccnum}[v]$   
     $G^R = \text{reverse}(G)$   
    DFS( $G^R$ ) // calculates post numbers  
    sort  $V$  // decreasing order of post number  
    ConnComp( $G$ )
```

Runtime complexity:

- DFS and ConnComp run in linear time $O(|V| + |E|)$.
- Can we reverse G in linear time?
- Can we sort V by post number in linear time?

Reversing G in linear time

```
function SCC( $G$ ):  
    // Input:  $G(V, E)$  a directed graph  
    // Output: each vertex  $v$  has an SCC number in  $\text{ccnum}[v]$   
     $G^R = \text{reverse}(G)$   
    DFS( $G^R$ ) // calculates post numbers  
    sort  $V$  // decreasing order of post number  
    ConnComp( $G$ )
```

```
function reverse( $G$ )  
    // Input:  $G(V, E)$  graph represented by an adjacency list  
    //         edges[ $v$ ] for each vertex  $v$ .  
    // Output:  $G(V, E^R)$  the reversed graph of  $G$ , with the  
    //         adjacency list edgesR[ $v$ ].  
  
    for each  $u \in V$ :  
        for each  $v \in \text{edges}[u]$ :  
            edgesR[ $v$ ].insert( $u$ )  
    return ( $V$ , edgesR)
```

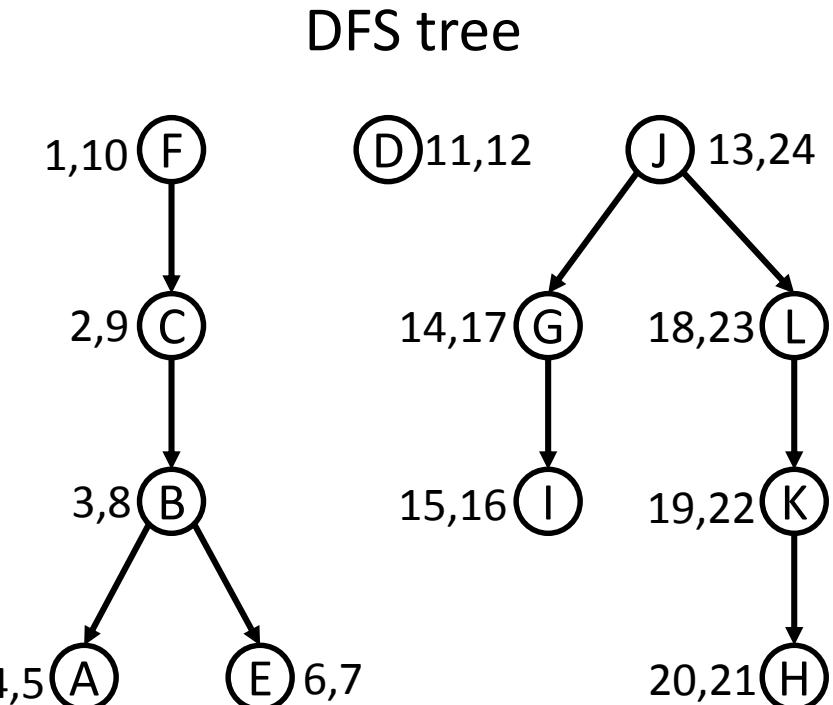
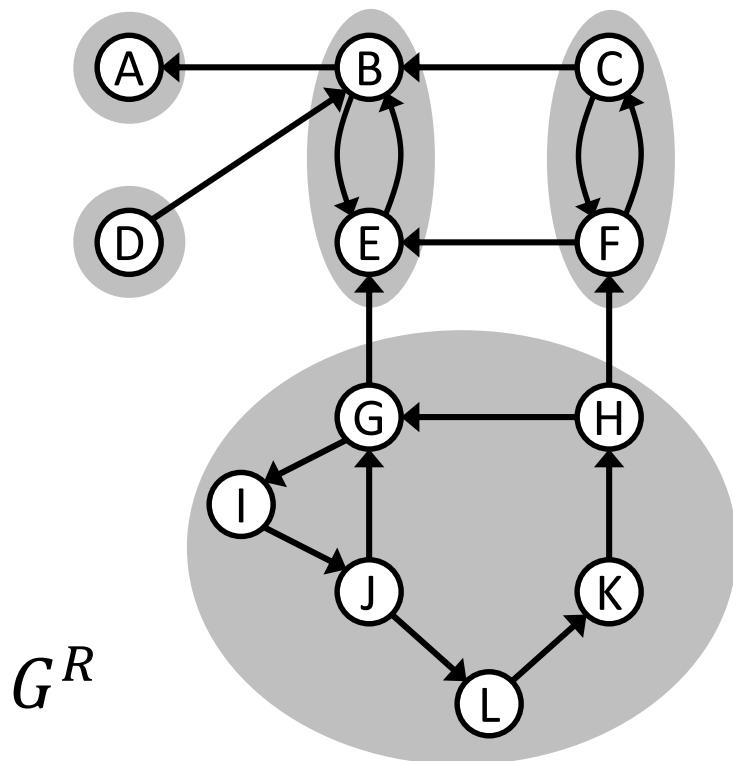
Sorting V in linear time

```
function SCC( $G$ ):  
    // Input:  $G(V, E)$  a directed graph  
    // Output: each vertex  $v$  has an SCC number in  $\text{ccnum}[v]$   
     $G^R = \text{reverse}(G)$   
    DFS( $G^R$ ) // calculates post numbers  
    sort  $V$  // decreasing order of post number  
    ConnComp( $G$ )
```

Use the explore function for topological sort:

- Each time a vertex is post-visited, it is inserted at the top of the list.
- The list is ordered by decreasing order of post number.
- It is executed in linear time

Sorting V in linear time



Assume the initial order:

$F, A, B, C, D, E, J, G, H, I, K, L$

Vertex:	J	L	K	H	G	I	D	F	C	B	E	A
Post:	24	23	22	21	17	16	12	10	9	8	7	5

Crawling the Web

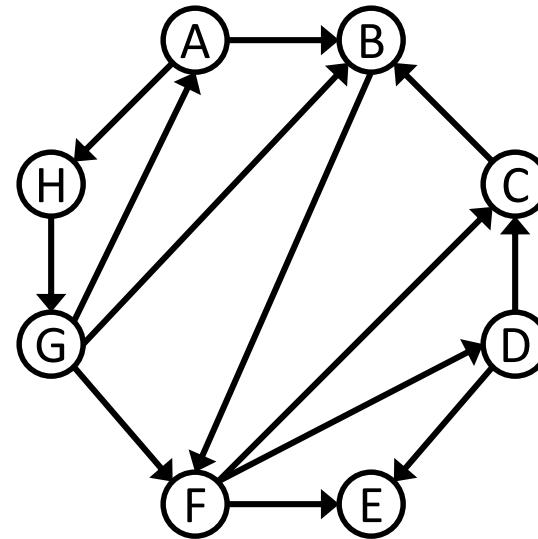
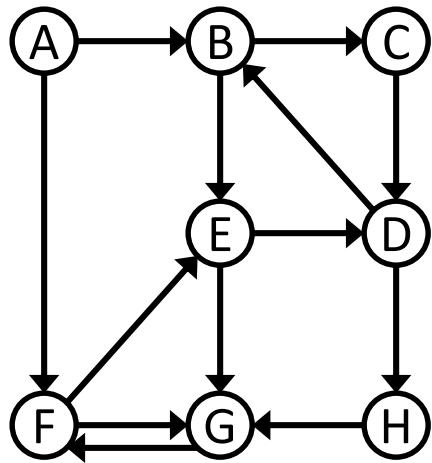
- Crawling the Web is done using depth-first search strategies.
- The graph is unknown and no recursion is used. A stack is used instead containing the nodes that have already been visited.
- The stack is not exactly a LIFO. Only the most “interesting” nodes are kept (e.g., page rank).
- Crawling is done in parallel (many computers at the same time) but using a central stack.
- How do we know that a page has already been visited?
Hashing.

Summary

- Big data is often organized in big graphs (objects and relations between objects)
- Big graphs are usually sparse. Adjacency lists is the most common data structure to represent graphs.
- Connectivity can be analyzed in linear time using depth-first search.

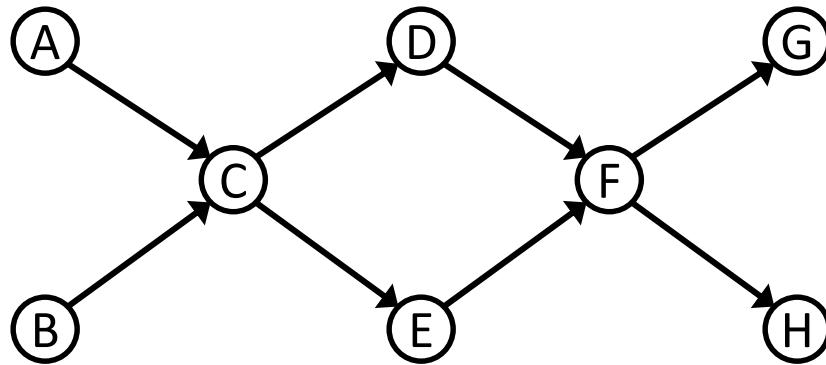
EXERCISES

Exercise (3.2 from [DPV2008])



Perform DFS on the two graphs. Whenever there is a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, forward edge, back edge or cross edge, and give the pre and post number of each vertex.

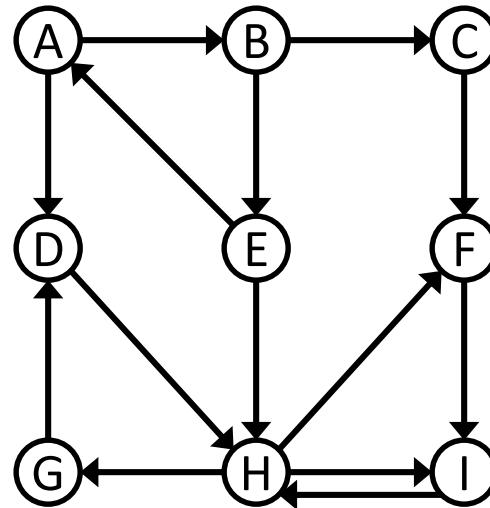
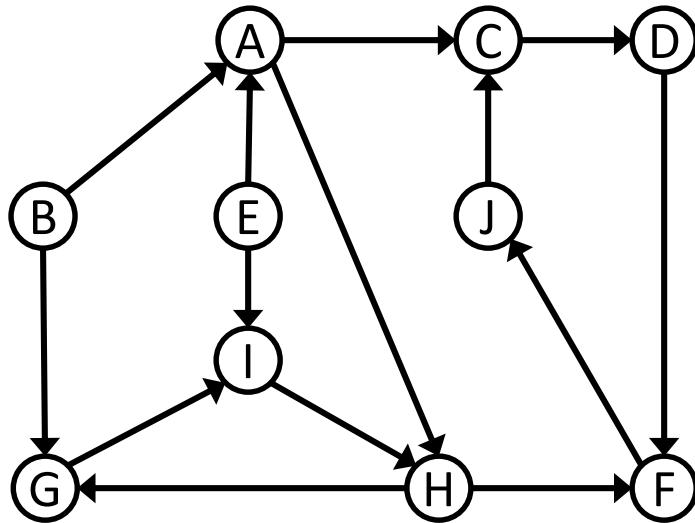
Exercise (3.3 from [DPV2008])



Run the DFS-based topological ordering algorithm on the graph. Whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

1. Indicate the pre and post numbers of the nodes.
2. What are the sources and sinks of the graph?
3. What topological order is found by the algorithm?
4. How many topological orderings does this graph have?

Exercise (3.4 from [DPV2008])



Run the SCC algorithm on the two graphs. When doing DFS of G^R : whenever there is a choice of vertices to explore, always pick the one that is alphabetically first. For each graph, answer the following questions:

1. In what order are the SCCs found?
2. Which are source SCCs and which are sink SCCs?
3. Draw the meta-graph (each meta-node is an SCC of G).
4. What is the minimum number of edges you must add to the graph to make it strongly connected?

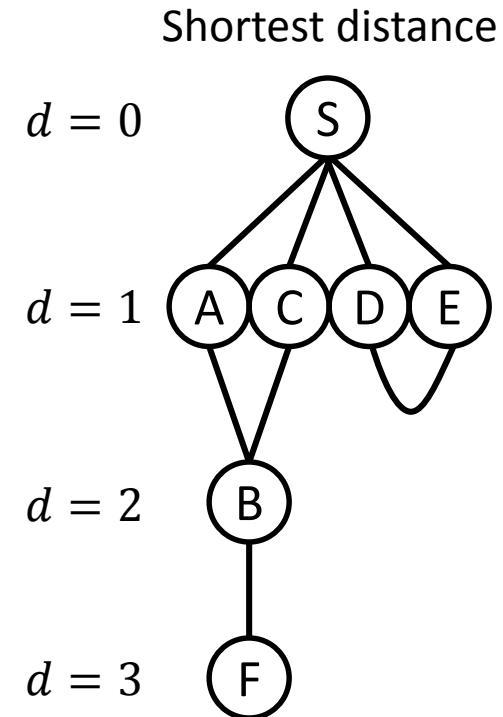
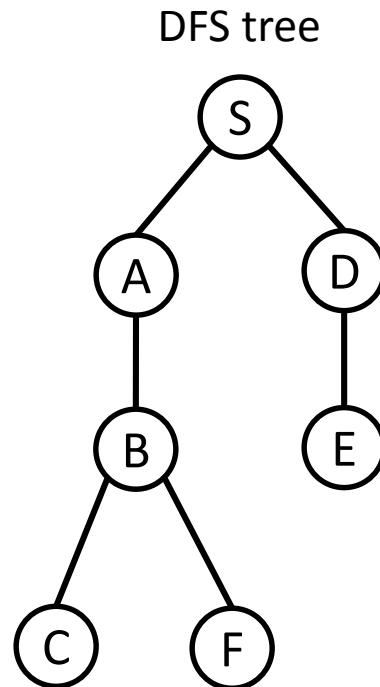
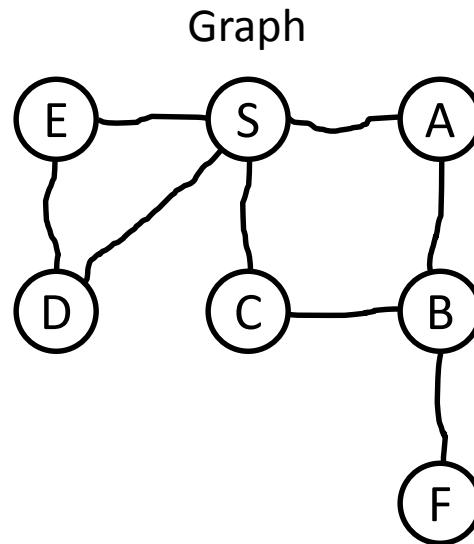
Graphs: Paths, trees and flows



Jordi Cortadella and Jordi Petit
Department of Computer Science

Distance in a graph

Depth-first search finds vertices reachable from another given vertex. The paths are not the shortest ones.



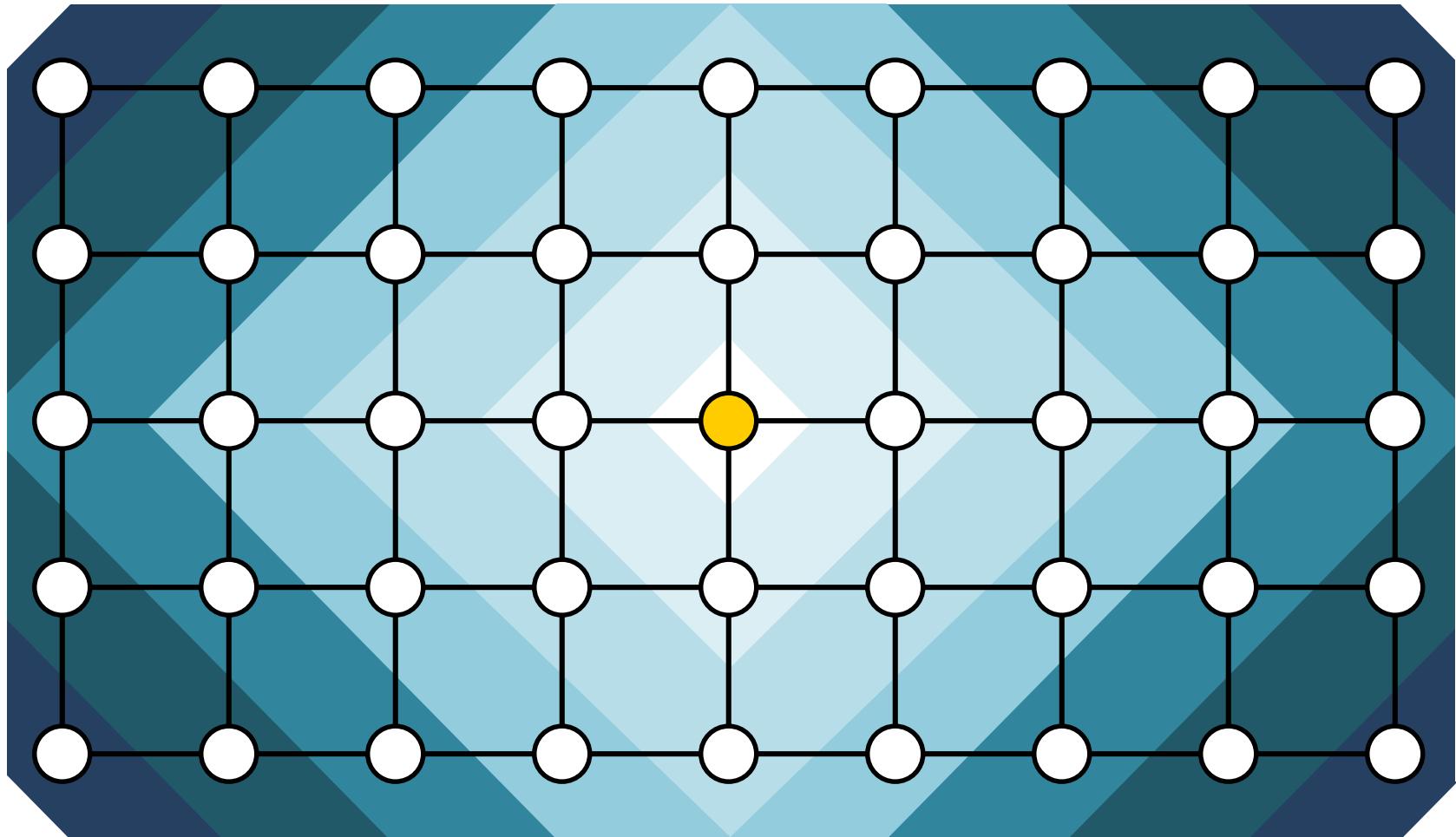
Distance between two nodes: length of the shortest path between them

Breadth-first search

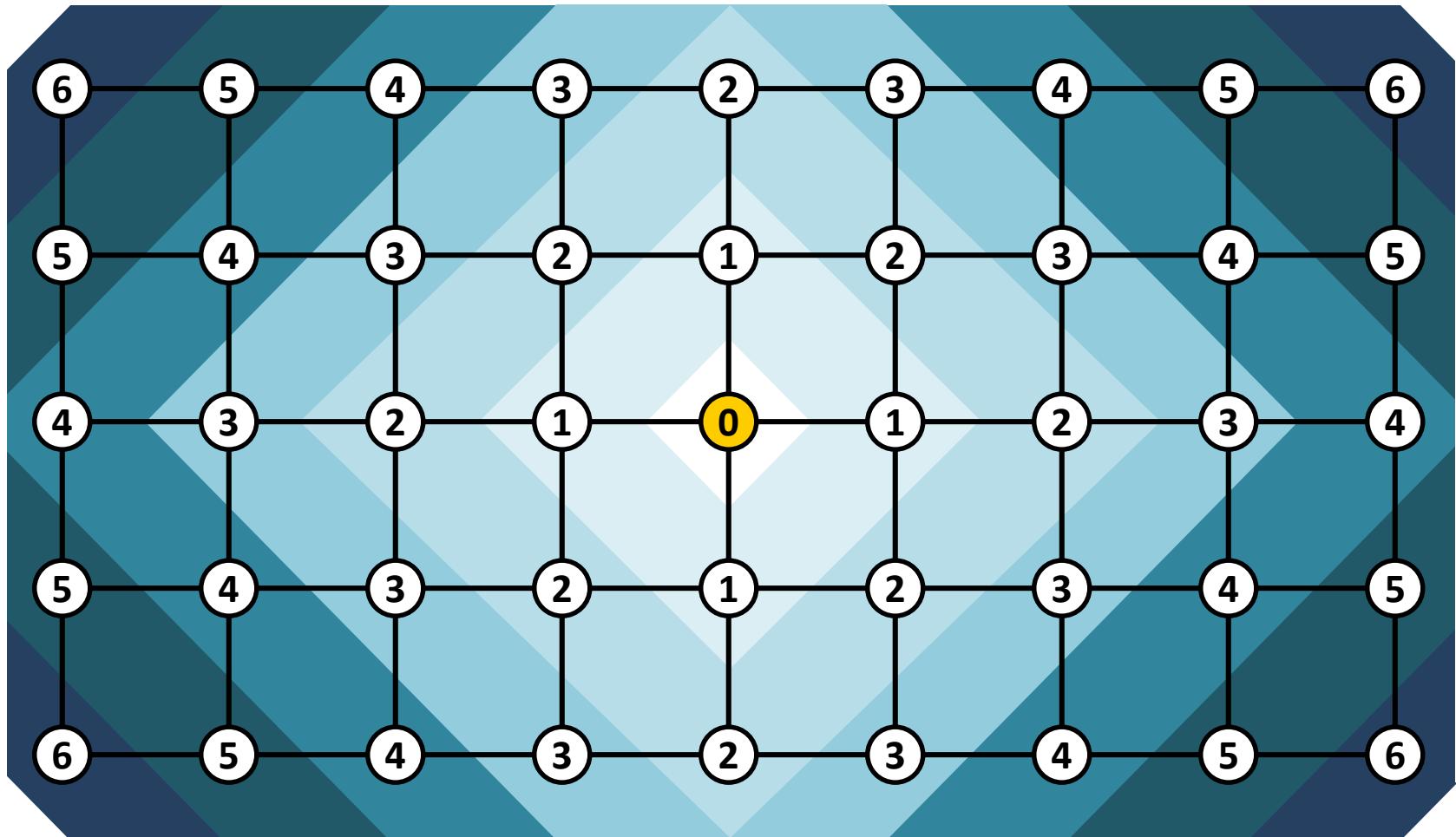


Similar to a wave propagation

Breadth-first search



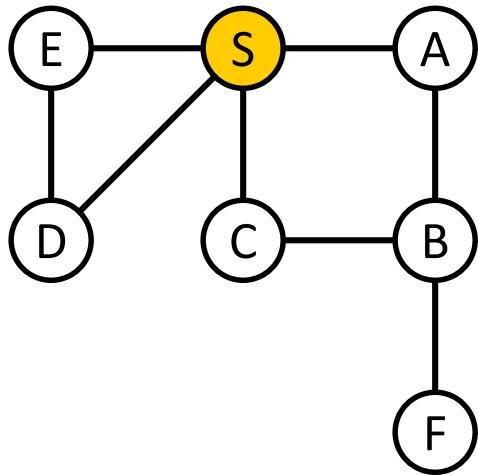
Breadth-first search



BFS algorithm

- BFS visits vertices layer by layer: $0, 1, 2, \dots, d$.
- Once the vertices at layer d have been visited, start visiting vertices at layer $d + 1$.
- Algorithm with two active layers:
 - Vertices at layer d (currently being visited).
 - Vertices at layer $d + 1$ (to be visited next).
- Central data structure: a queue.

BFS algorithm



	S_0	<table border="1"> <tr><td>S</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td></tr> <tr><td>0</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td></tr> </table>	S	A	B	C	D	E	F	0	∞	∞	∞	∞	∞	∞
S	A	B	C	D	E	F										
0	∞	∞	∞	∞	∞	∞										
S_0	$A_1 \ C_1 \ D_1 \ E_1$	<table border="1"> <tr><td>A_1</td><td>C_1</td><td>D_1</td><td>E_1</td><td>B_2</td></tr> <tr><td>0</td><td>1</td><td>∞</td><td>1</td><td>1</td></tr> </table>	A_1	C_1	D_1	E_1	B_2	0	1	∞	1	1				
A_1	C_1	D_1	E_1	B_2												
0	1	∞	1	1												
A_1	$C_1 \ D_1 \ E_1 \ B_2$	<table border="1"> <tr><td>C_1</td><td>D_1</td><td>E_1</td><td>B_2</td><td>∞</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>1</td><td>1</td></tr> </table>	C_1	D_1	E_1	B_2	∞	0	1	2	1	1				
C_1	D_1	E_1	B_2	∞												
0	1	2	1	1												
C_1	$D_1 \ E_1 \ B_2$	<table border="1"> <tr><td>D_1</td><td>E_1</td><td>B_2</td><td>∞</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>1</td></tr> </table>	D_1	E_1	B_2	∞	0	1	2	1						
D_1	E_1	B_2	∞													
0	1	2	1													
D_1	$E_1 \ B_2$	<table border="1"> <tr><td>E_1</td><td>B_2</td><td>∞</td></tr> <tr><td>0</td><td>1</td><td>2</td></tr> </table>	E_1	B_2	∞	0	1	2								
E_1	B_2	∞														
0	1	2														
E_1	B_2	<table border="1"> <tr><td>B_2</td><td>∞</td></tr> <tr><td>0</td><td>1</td></tr> </table>	B_2	∞	0	1										
B_2	∞															
0	1															
B_2	F_3	<table border="1"> <tr><td>F_3</td><td>∞</td></tr> <tr><td>0</td><td>1</td></tr> </table>	F_3	∞	0	1										
F_3	∞															
0	1															
F_3		<table border="1"> <tr><td>∞</td><td>∞</td></tr> <tr><td>0</td><td>1</td></tr> </table>	∞	∞	0	1										
∞	∞															
0	1															

BFS algorithm

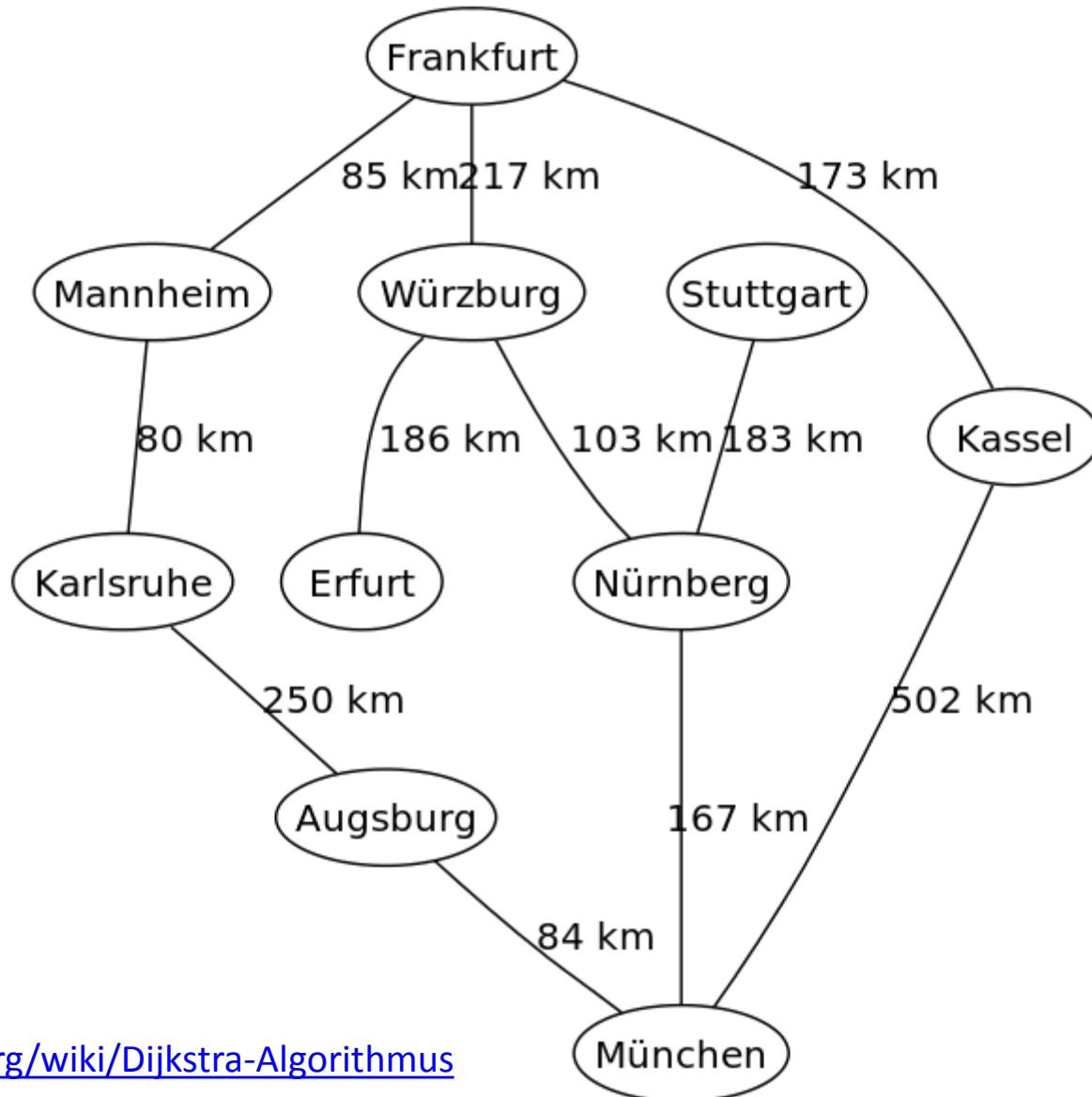
```
function BFS( $G$ ,  $s$ )
    // Input: Graph  $G(V,E)$ , source vertex  $s$ .
    // Output: For each vertex  $u$ ,  $\text{dist}[u]$  is
    //          the distance from  $s$  to  $u$ .

    for all  $u \in V$ :  $\text{dist}[u] = \infty$ 

     $\text{dist}[s] = 0$ 
     $Q = \{s\}$  // Queue containing just  $s$ 
    while not  $Q.\text{empty}()$ :
         $u = Q.\text{pop\_front}()$ 
        for all  $(u,v) \in E$ :
            if  $\text{dist}[v] = \infty$ :
                 $\text{dist}[v] = \text{dist}[u] + 1$ 
                 $Q.\text{push\_back}(v)$ 
```

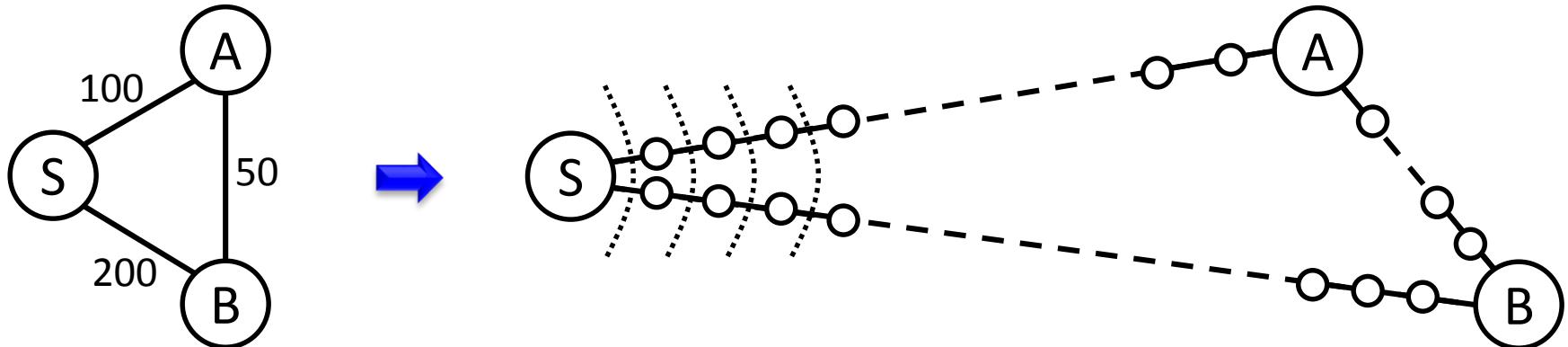
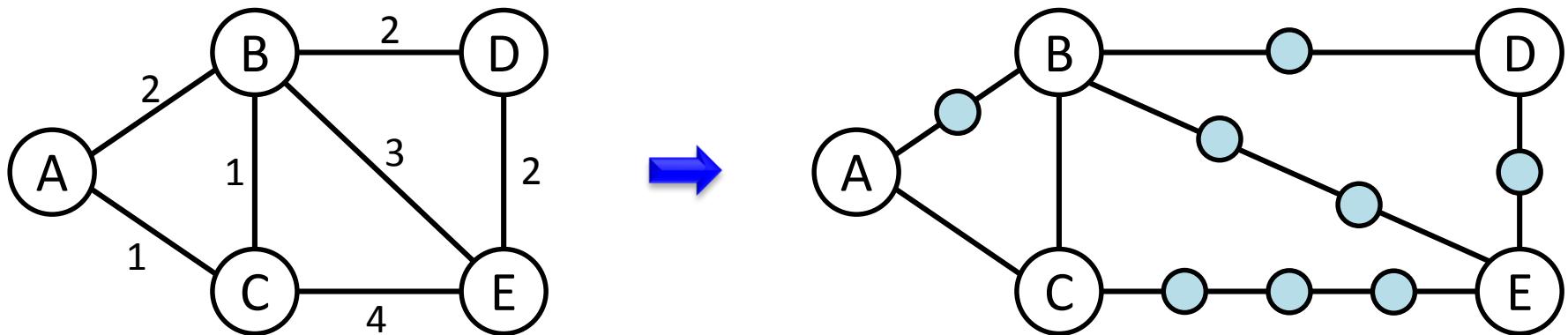
Runtime $O(|V| + |E|)$: Each vertex is visited once, each edge is visited once (for directed graphs) or twice (for undirected graphs).

Distances on edges



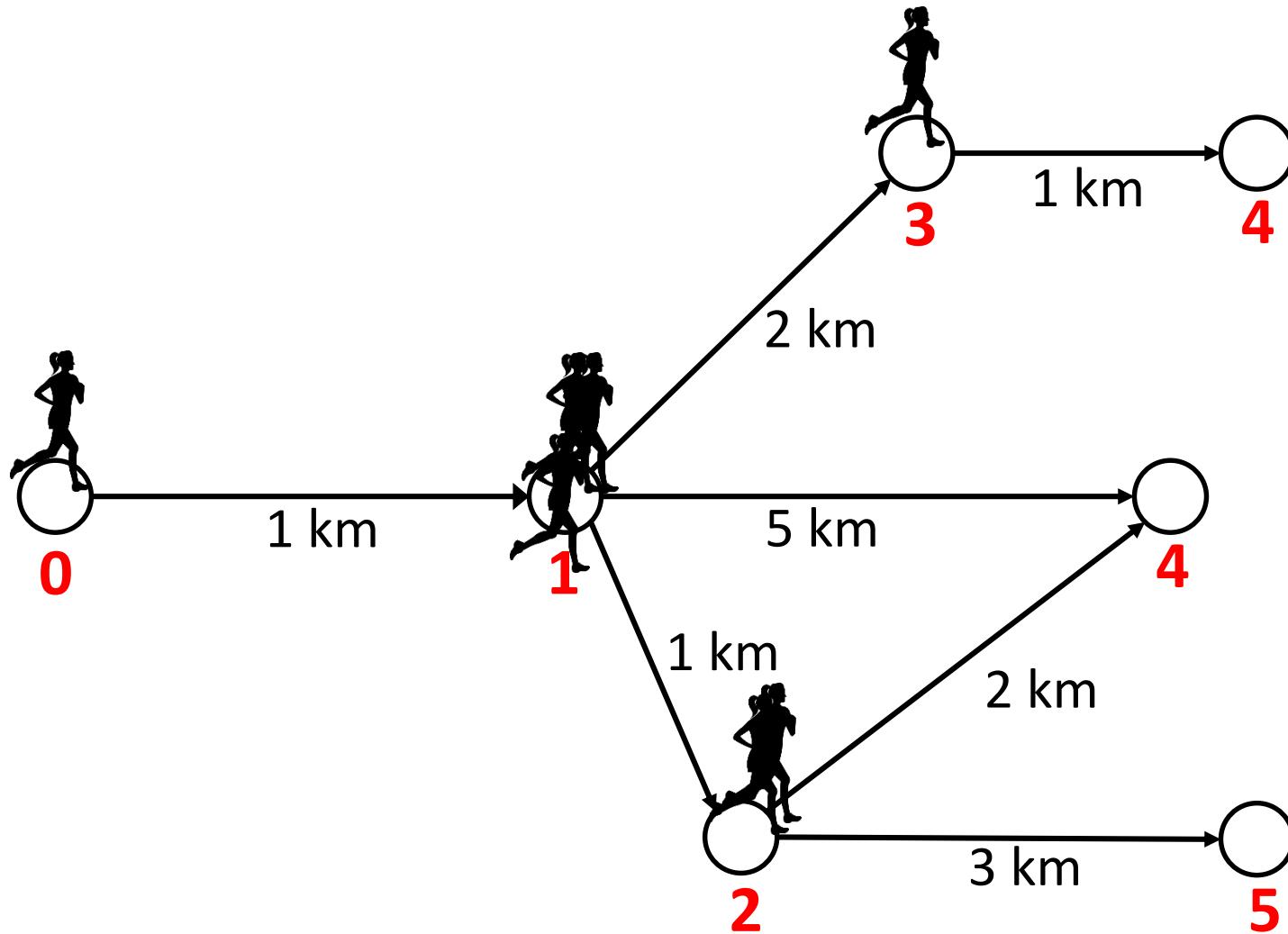
<https://de.wikipedia.org/wiki/Dijkstra-Algorithmus>

Reusing BFS



Inefficient: many cycles without any interesting progress. How about real numbers?

Tracking runners



Annotate the time of the first arrival at each node.

The runners algorithm

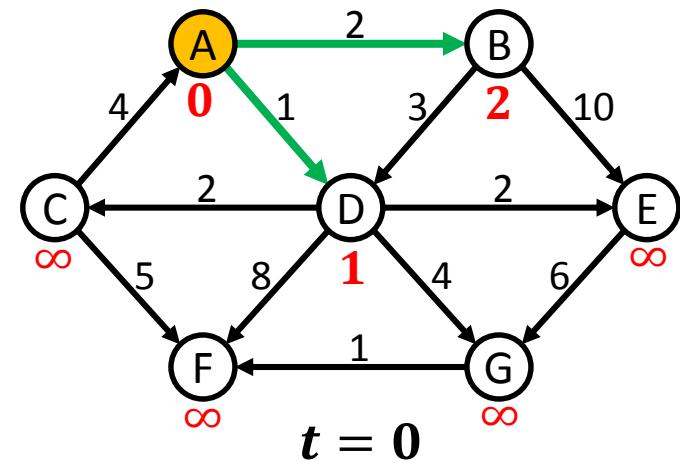
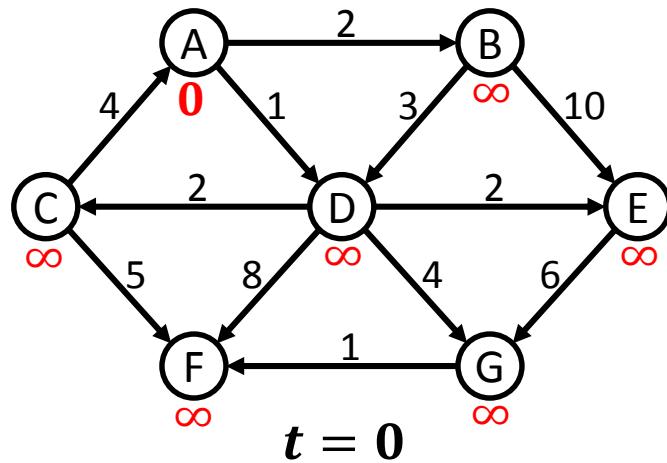
At the beginning, we have one runner for each edge $u \rightarrow v$ waiting at u for the arrival of the first runner. When the first runner arrives at u , all runners waiting at u start running along their edge $u \rightarrow v$. All runners run at a constant speed: $1 \text{ } d_{\text{unit}} / t_{\text{unit}}$

We just need to annotate the first arrival time at every vertex.

Algorithm:

- Set the expected arrival time at s to 0.
No expected arrival time for the rest of vertices (∞).
- Repeat until there are no more runners:
 - Select the earliest expected arrival time at a vertex not visited yet (vertex u , time T). Annotate final distance $s \rightarrow u$ to T .
 - For each outgoing $u \rightarrow v$:
If v has not been visited yet and the expected arrival time for the runner $u \rightarrow v$ is smaller than the expected arrival time at v by any other active runner, start running along $u \rightarrow v$ and stop the other runners going to v .
Otherwise, do not start running.

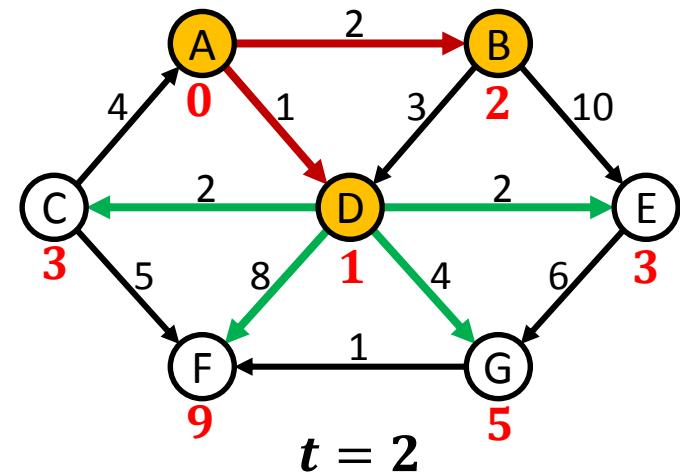
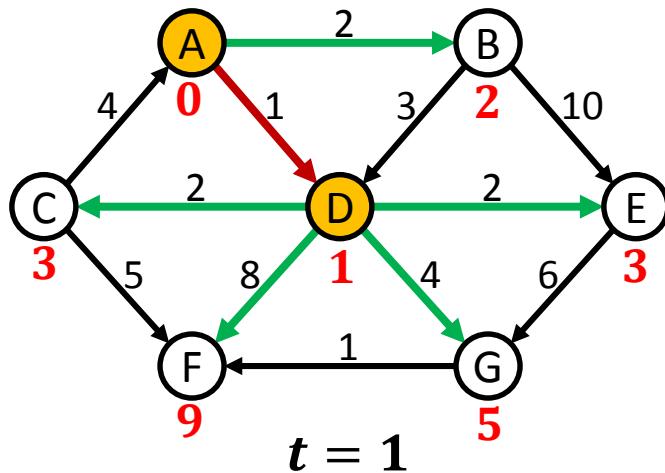
Example



Done	Queue
	A:0
	B: ∞
	E: ∞
	D: ∞
	C: ∞
	F: ∞
	G: ∞

Done	Queue
A:0	D:1
	B:2
	E: ∞
	C: ∞
	F: ∞
	G: ∞

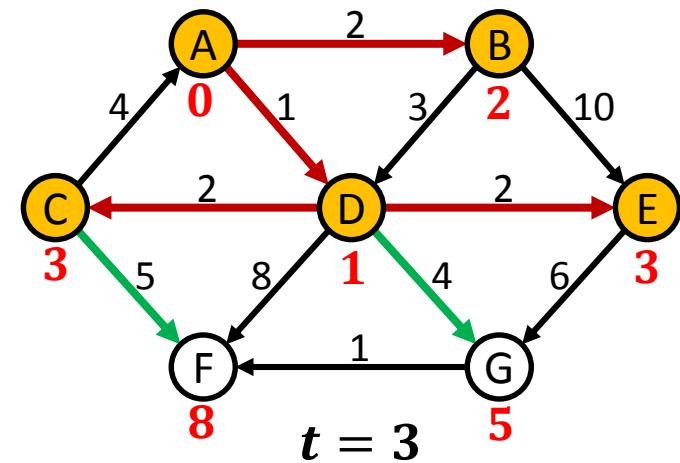
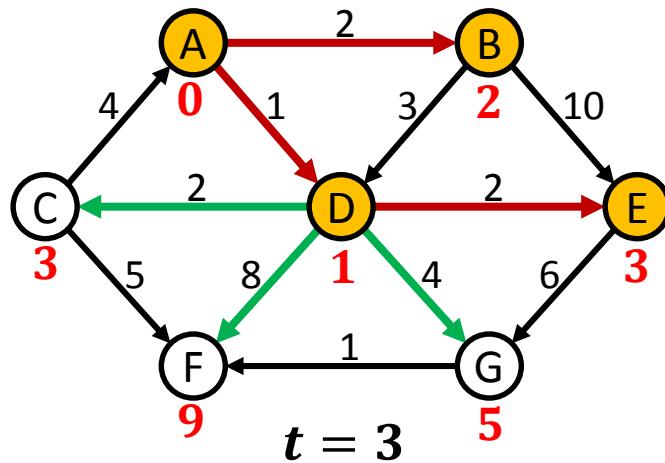
Example



Done	Queue
A:0	B:2
D:1	E:3
	C:3
	G:5
	F:9

Done	Queue
A:0	E:3
D:1	C:3
B:2	G:5
	F:9

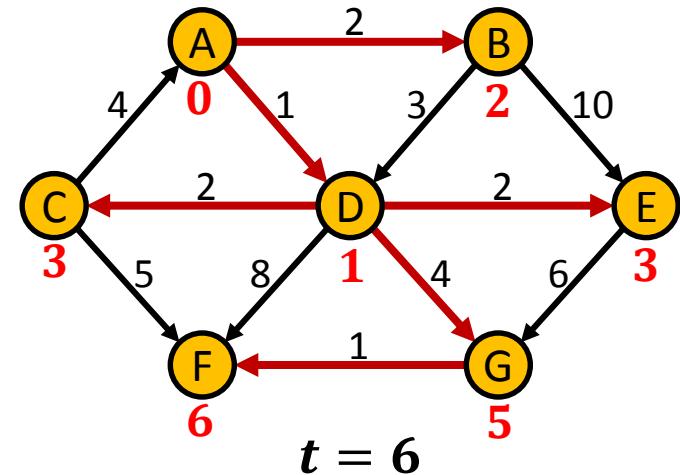
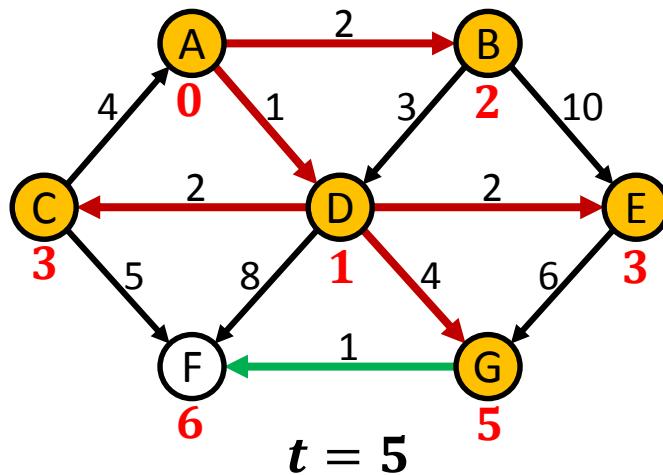
Example



Done	Queue
A:0	C:3
D:1	G:5
B:2	F:9
E:3	

Done	Queue
A:0	G:5
D:1	F:8
B:2	
E:3	
C:3	

Example

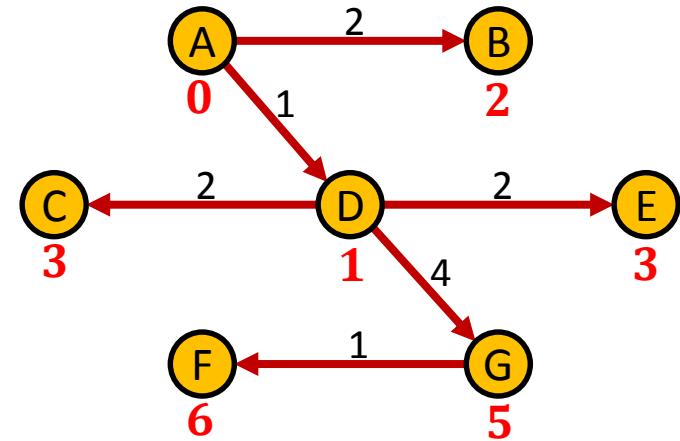


Done	Queue
A:0	F:6
D:1	
B:2	
E:3	
C:3	
G:5	

Done	Queue
A:0	
D:1	
B:2	
E:3	
C:3	
G:5	
F:6	

Example

Shortest-path tree



We need to:

- keep a list of active runners and their expected arrival times.
- select the earliest runner arriving at a vertex.
- update the active runners and their arrival times.

Dijkstra's algorithm for shortest paths

```
function ShortestPaths( $G, l, s$ )
  // Input: Graph  $G(V, E)$ , source vertex  $s$ ,
  //         positive edge lengths  $\{l_e : e \in E\}$ 
  // Output:  $\text{dist}[u]$  has the distance from  $s$ ,
  //           $\text{prev}[u]$  has the predecessor in the tree

  for all  $u \in V$ :
     $\text{dist}[u] = \infty$ 
     $\text{prev}[u] = \text{nil}$ 

   $\text{dist}[s] = 0$ 
   $Q = \text{makequeue}(V)$  // using  $\text{dist}$  as keys

  while not  $Q.\text{empty}()$  :
     $u = Q.\text{deletemin}()$ 
    for all  $(u, v) \in E$ :
      if  $\text{dist}[v] > \text{dist}[u] + l(u, v)$ :
         $\text{dist}[v] = \text{dist}[u] + l(u, v)$ 
         $\text{prev}[v] = u$ 
         $Q.\text{decreasekey}(v)$ 
```

Dijkstra's algorithm: complexity

```
Q = makequeue(V)
while not Q.empty():
    u = Q.deletemin() ← |V| times
    for all (u,v) ∈ E:
        if dist[v] > dist[u] + l(u,v):
            dist[v] = dist[u] + l(u,v)
            prev[v] = u
    Q.decreasekey(v) ← |E| times
```

- The skeleton of Dijkstra's algorithm is based on BFS, which is $O(|V| + |E|)$
- We need to account for the cost of:
 - **makequeue**: insert $|V|$ vertices to a list.
 - **deletemin**: find the vertex with min dist in the list ($|V|$ times)
 - **decreasekey**: update dist for a vertex ($|E|$ times)
- Let us consider two implementations for the list: **vector** and **binary heap**

Dijkstra's algorithm: complexity

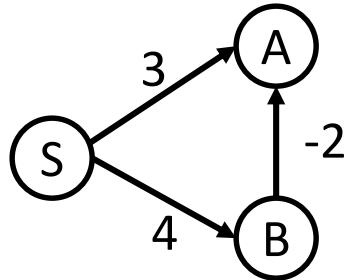
Implementation	deletemin	insert/ decreasekey	Dijkstra's complexity
Vector	$O(V)$	$O(1)$	$O(V ^2)$
Binary heap	$O(\log V)$	$O(\log V)$	$O((V + E) \log V)$

Binary heap:

- The elements are stored in a complete (balanced) binary tree.
- **Insertion:** place element at the bottom and let it *bubble up* swapping the location with the parent (at most $\log_2 |V|$ levels).
- **Deletemin:** Remove element from the root, take the last node in the tree, place it at the root and let it *sift down* (at most $\log_2 |V|$ levels).
- **Decreasekey:** decrease the key in the tree and let it bubble up (same as insertion). A data structure might be required to known the location of each vertex in the heap (table of pointers).

Graphs with negative edges

- Dijkstra's algorithm does not work:



Dijkstra would say that the shortest path $S \rightarrow A$ has length=3.

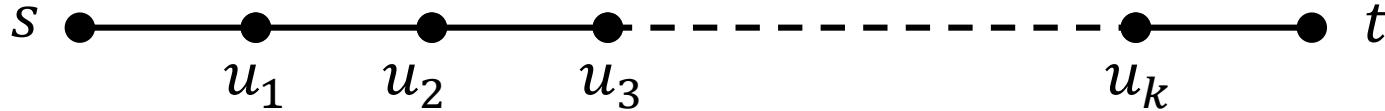
- Dijkstra is based on a safe update each time an edge (u, v) is treated:

$$\text{dist}(v) = \min\{\text{dist}(v), \text{dist}(u) + l(u, v)\}$$

- Problem: non-promising updates are not done.
- Solution: let us not abort updates that early.

Graphs with negative edges

- The shortest path from s to t can have at most $|V| - 1$ edges:



- If the sequence of updates includes

$$(s, u_1), (u_1, u_2), (u_2, u_3), \dots, (u_k, t),$$

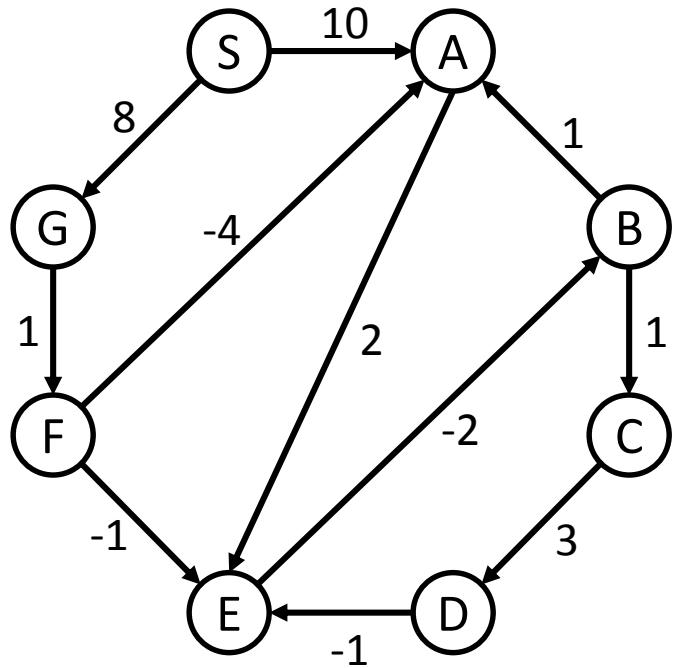
in that order, the shortest distance from s to t will be computed correctly (updates are always safe). Note that the sequence of updates does not need to be consecutive.

- Solution: update all edges $|V| - 1$ times !
- Complexity: $O(|V| \cdot |E|)$.

Bellman-Ford algorithm

```
function ShortestPaths( $G, l, s$ )  
  
    // Input: Graph  $G(V,E)$ , source vertex  $s$ ,  
    //         edge lengths  $\{l_e : e \in E\}$ , no negative cycles.  
    // Output:  $\text{dist}[u]$  has the distance from  $s$ ,  
    //           $\text{prev}[u]$  has the predecessor in the tree  
  
    for all  $u \in V$ :  
         $\text{dist}[u] = \infty$   
         $\text{prev}[u] = \text{nil}$   
  
     $\text{dist}[s] = 0$   
    repeat  $|V| - 1$  times:  
        for all  $(u,v) \in E$ :  
            if  $\text{dist}[v] > \text{dist}[u] + l(u,v)$ :  
                 $\text{dist}[v] = \text{dist}[u] + l(u,v)$   
                 $\text{prev}[v] = u$ 
```

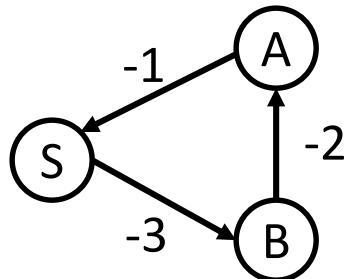
Bellman-Ford: example



Node	Iteration							
	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Negative cycles

- What is the shortest distance between S and A?



Bellman-Ford does not work as it assumes that the shortest path will not have more than $|V| - 1$ edges.

- A negative cycle produces $-\infty$ distances by endlessly applying rounds to the cycle.
- How to detect negative cycles?
 - Apply Bellman-Ford (update edges $|V| - 1$ times)
 - Perform an extra round and check whether some distance decreases.

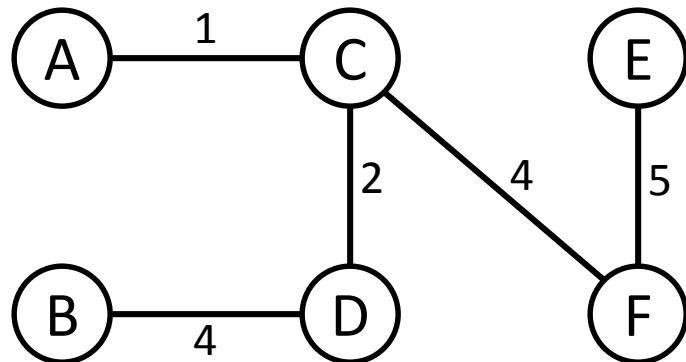
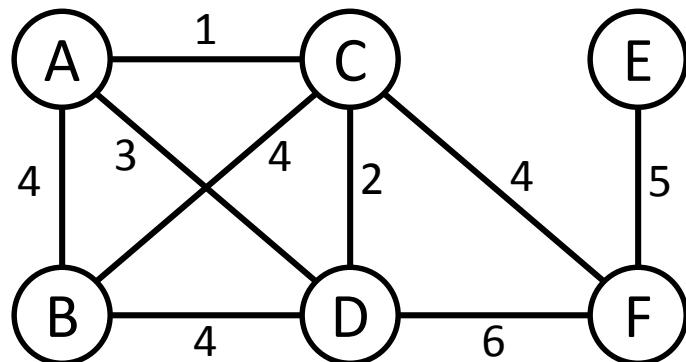
Shortest paths in DAGs

- DAG's property:
In any path of a DAG, the vertices appear in increasing topological order.
- Any sequence of updates that preserves the topological order will compute distances correctly.
- Only one round visiting the edges in topological order is sufficient: $O(|V| + |E|)$.
- How to calculate the longest paths?
 - Negate the edge lengths.
 - Compute the shortest paths.

DAG shortest paths algorithm

```
function DagShortestPaths( $G, l, s$ )  
  
    // Input: DAG  $G(V, E)$ , source vertex  $s$ ,  
    //        edge lengths  $\{l_e : e \in E\}$ .  
    // Output:  $\text{dist}[u]$  has the distance from  $s$ ,  
    //           $\text{prev}[u]$  has the predecessor in the tree  
  
    for all  $u \in V$ :  
         $\text{dist}[u] = \infty$   
         $\text{prev}[u] = \text{nil}$   
  
     $\text{dist}[s] = 0$   
    Linearize  $G$   
    for all  $u \in V$  in linearized order:  
        for all  $(u, v) \in E$ :  
            if  $\text{dist}[v] > \text{dist}[u] + l(u, v)$ :  
                 $\text{dist}[v] = \text{dist}[u] + l(u, v)$   
                 $\text{prev}[v] = u$ 
```

Minimum Spanning Trees



- Nodes are computers
- Edges are links
- Weights are maintenance cost
- Goal: pick a subset of edges such that
 - the nodes are connected
 - the maintenance cost is minimum

The solution is not unique.
Find another one !

Property:
An optimal solution cannot contain a cycle.

Properties of trees

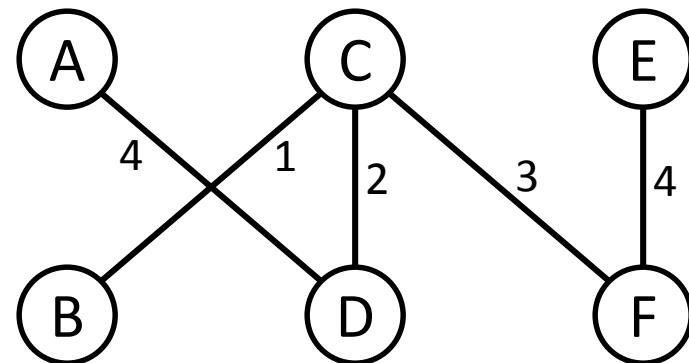
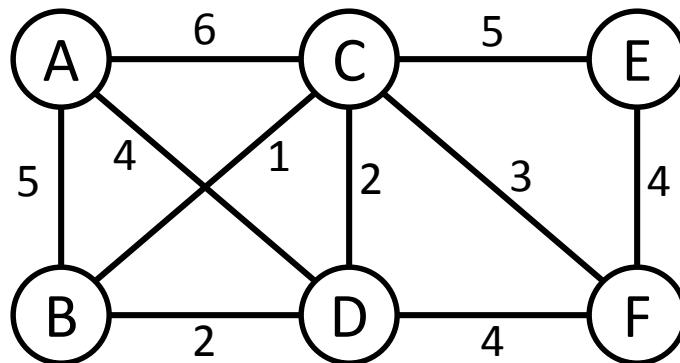
- A tree is an undirected graph that is connected and acyclic.
- Property: A tree on n nodes has $n - 1$ edges.
 - Start from an empty graph. Add one edge at a time making sure that it connects two disconnected components.
- Property: Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.
 - It is sufficient to prove that G is acyclic. If not, we can always remove edges from cycles until the graph becomes acyclic.
- Property: Any undirected graph is a tree iff there is a unique path between any pair of nodes.
 - If there would be two paths between two nodes, the union of the paths would contain a cycle.

Minimum Spanning Tree

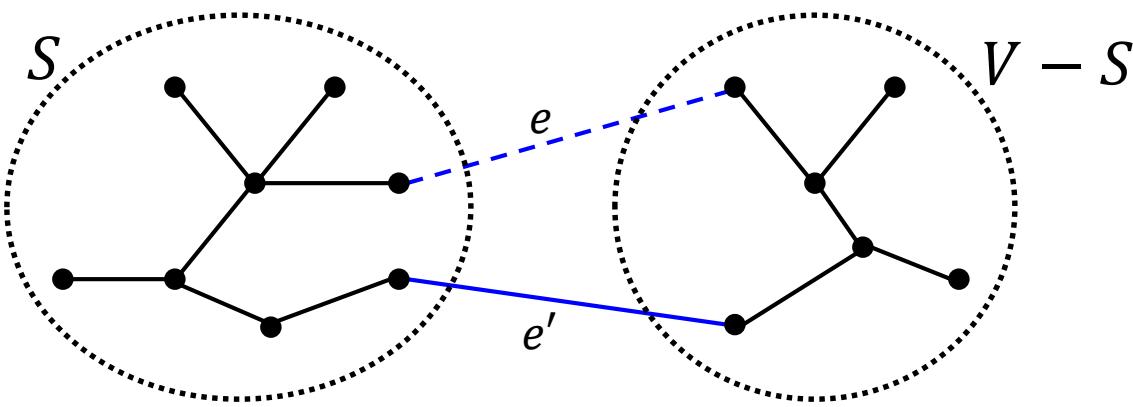
- Given an undirected graph $G = (V, E)$ with edge weights w_e , find a tree $T = (V, E')$, with $E' \subseteq E$, that minimizes

$$\text{weight}(T) = \sum_{e \in E'} w_e .$$

- Kruskal's algorithm: repeatedly add the next lightest edge that does not produce a cycle.



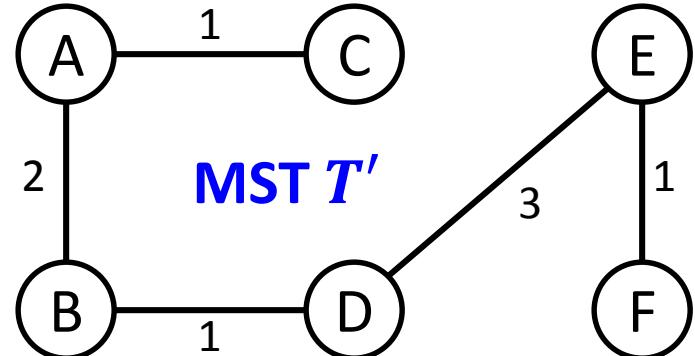
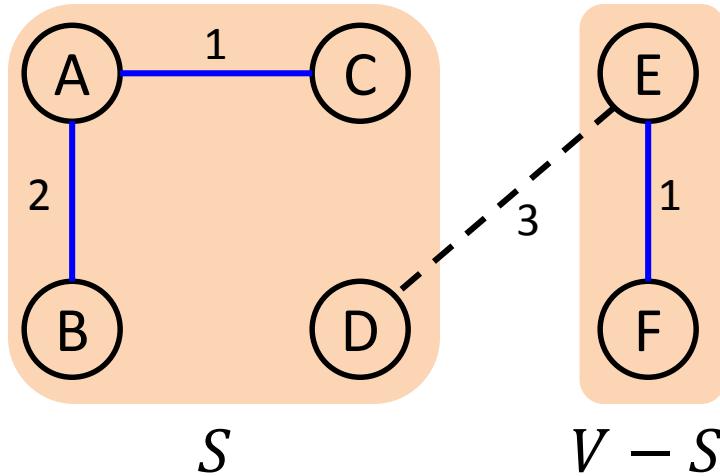
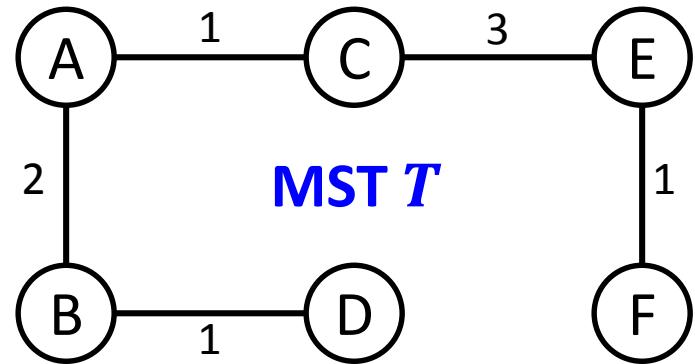
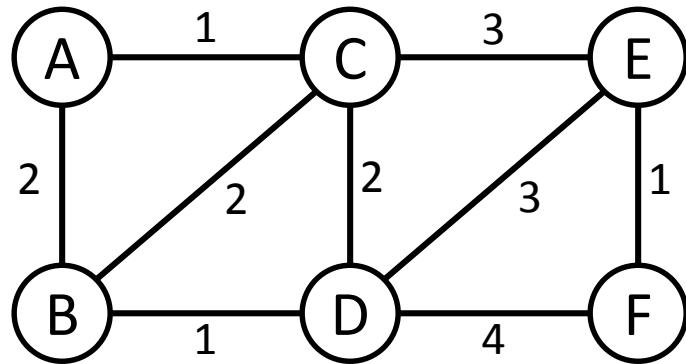
The cut property



Suppose edges X are part of an MST of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V - S$, and let e be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.

Proof (sketch): Let T be an MST and assume e is not in T . If we add e to T , a cycle will be created with another edge e' across the cut $(S, V - S)$. We can now remove e' and obtain another tree T' with $\text{weight}(T') \leq \text{weight}(T)$. Since T is an MST, then the weights must be equal.

The cut property: example



Minimum Spanning Tree

Any scheme like this works (because of the properties of trees):

$X = \{ \}$

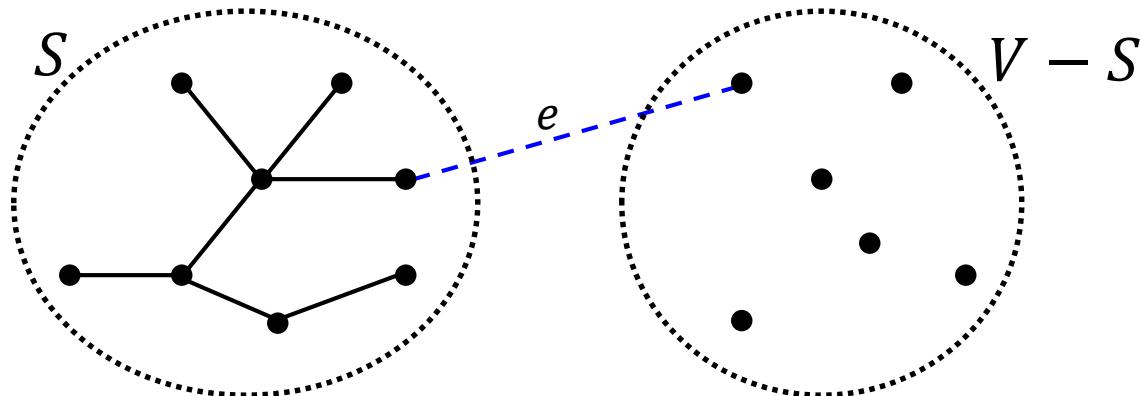
repeat $|V| - 1$ times:

pick a set $S \subset V$ for which X has no edges between S and $V - S$

let $e \in E$ be the minimum-weight edge between S and $V - S$

$X = X \cup \{e\}$

Prim's algorithm: X always forms a subtree and S is the set of X 's nodes:

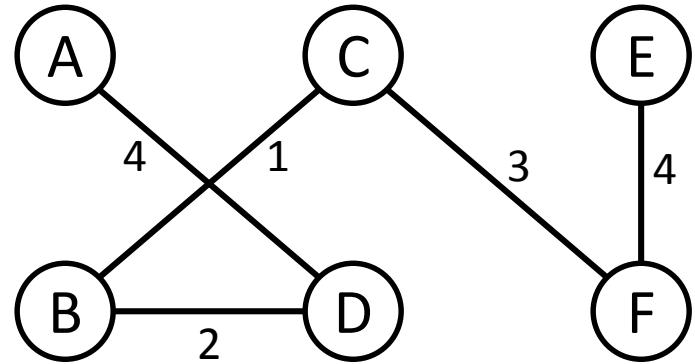
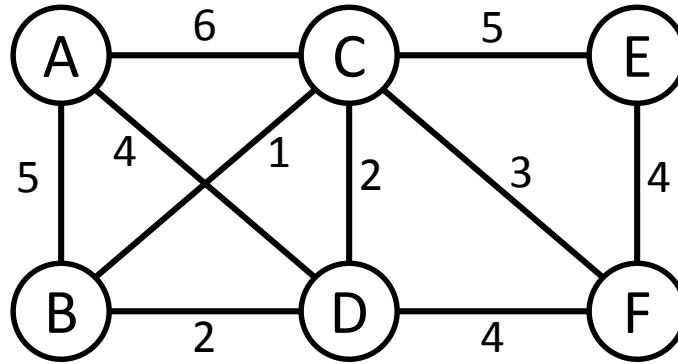


Prim's algorithm

```
function Prim( $G, w$ )
// Input: A connected undirected Graph  $G(V, E)$ 
//         with edge weights  $w_e$ .
// Output: An MST defined by the vector prev.
for all  $u \in V$ :
    cost( $u$ ) =  $\infty$ 
    prev( $u$ ) = nil
pick any initial node  $u_0$ 
cost( $u_0$ ) = 0

//  $H$  priority queue using cost as key (set  $V - S$ )
 $H$  = makequeue( $V$ )
while  $H$  is not empty:
     $v$  = deletemin( $H$ )
    for each  $(v, z) \in E, z \in H$ :
        if cost( $z$ ) >  $w(v, z)$ :
            cost( $z$ ) =  $w(v, z)$ 
            prev( $z$ ) =  $v$ 
            decreasekey( $H, z$ )
```

Prim's algorithm



Set S	A	B	C	D	E	F
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$5/A$	$6/A$	4/A	∞/nil	∞/nil
A, D		$2/D$	$2/D$		∞/nil	$4/D$
A, D, B			$1/B$		∞/nil	$4/D$
A, D, B, C					$5/C$	3/C
A, D, B, C, F					4/F	

Complexity: the same as Dijkstra's algorithm, $O((|V| + |E|) \log |V|)$

Kruskal's algorithm

Informal algorithm:

- Sort edges by weight.
- Visit edges in ascending order of weight and add them as long as they do not create a cycle.

How do we know whether a new edge will create a cycle?

```
function Kruskal(G, w)

// Input: A connected undirected Graph  $G(V, E)$ 
//         with edge weights  $w_e$ .

// Output: An MST defined by the edges in  $X$ .
```

$X = \{\}$

sort the edges in E by weight

for all $(u, v) \in E$, in ascending order of weight:

 if (X has no path connecting u and v):

$X = X \cup \{(u, v)\}$

Disjoint sets

- A data structure to store a collection of disjoint sets.
- Operations:
 - $\text{makeset}(x)$: creates a singleton set containing just x .
 - $\text{find}(x)$: returns the identifier of the set containing x .
 - $\text{union}(x, y)$: merges the sets containing x and y .
- Kruskal's algorithm uses disjoint sets and calls
 - makeset : $|V|$ times
 - find : $2 \cdot |E|$ times
 - union : $|V| - 1$ times

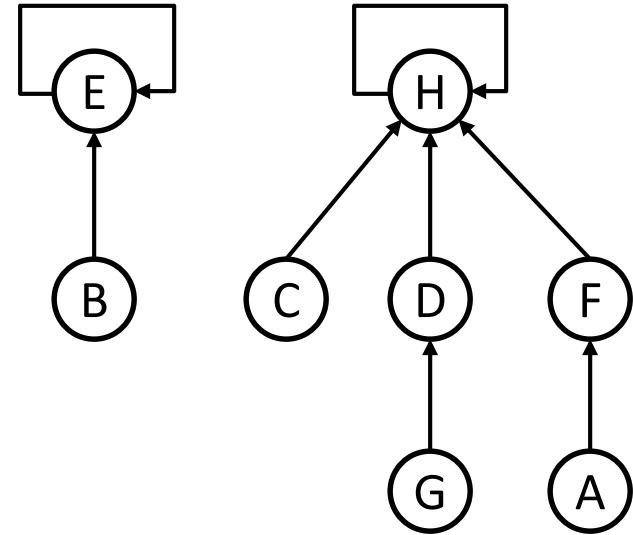
Kruskal's algorithm

```
function Kruskal( $G, w$ )
    // Input: A connected undirected Graph  $G(V, E)$ 
    //         with edge weights  $w_e$ .
    // Output: An MST defined by the edges in  $X$ .

    for all  $u \in V$ : makeset( $u$ )
     $X = \{ \}$ 
    sort the edges in  $E$  by weight
    for all  $(u, v) \in E$ , in ascending order of weight:
        if (find( $u$ ) ≠ find( $v$ )):
             $X = X \cup \{(u, v)\}$ 
            union( $u, v$ )
```

Disjoint sets

- The nodes are organized as a set of trees. Each tree represents a set.
- Each node has two attributes:
 - parent (π): ancestor in the tree
 - rank: height of the subtree
- The root element is the representative for the set: its parent pointer is itself (self-loop).
- The efficiency of the operations depends on the height of the trees.

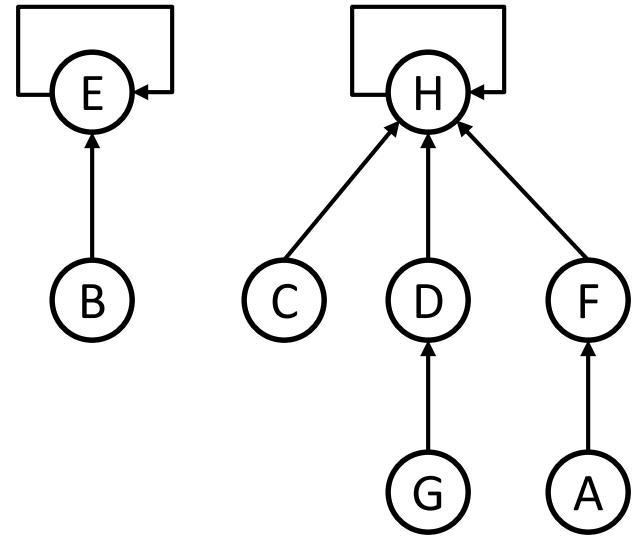


```
function makeset(x):  
     $\pi(x) = x$   
    rank(x) = 0  
  
function find(x):  
    while  $x \neq \pi(x)$ :  $x = \pi(x)$   
    return  $x$ 
```

Disjoint sets

```
function union(x, y):
    rx = find(x)
    ry = find(y)
    if rx = ry: return

    if rank(rx) > rank(ry):
        π(ry) = rx
    else:
        π(rx) = ry
        if rank(rx) = rank(ry):
            rank(ry) = rank(ry) + 1
```



```
function makeset(x):
    π(x) = x
    rank(x) = 0

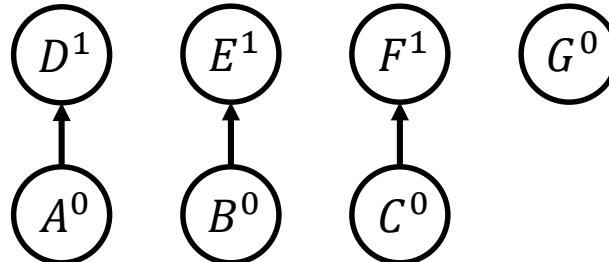
function find(x):
    while x ≠ π(x): x = π(x)
    return x
```

Disjoint sets

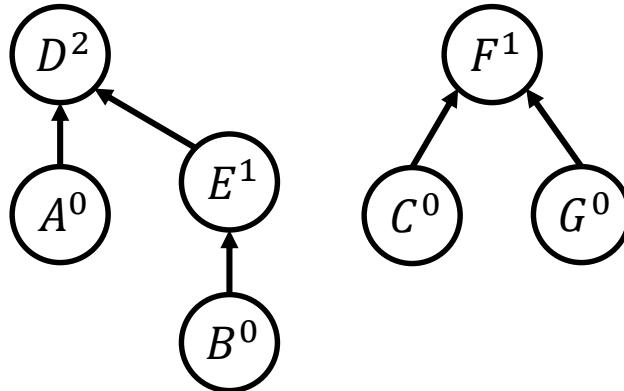
After $\text{makeset}(A), \dots, \text{makeset}(G)$:



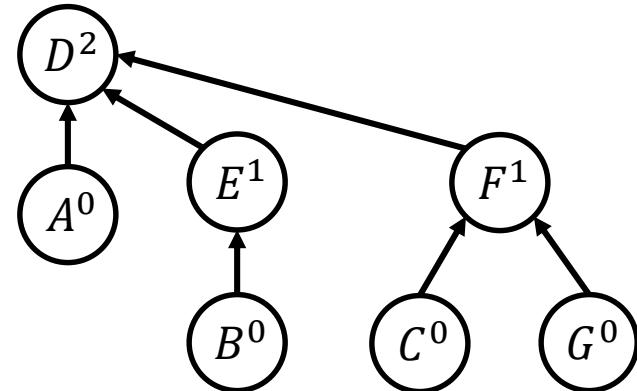
After $\text{union}(A,D), \text{union}(B,E), \text{union}(C,F)$:



After $\text{union}(C,G), \text{union}(E,A)$:



After $\text{union}(B,G)$:

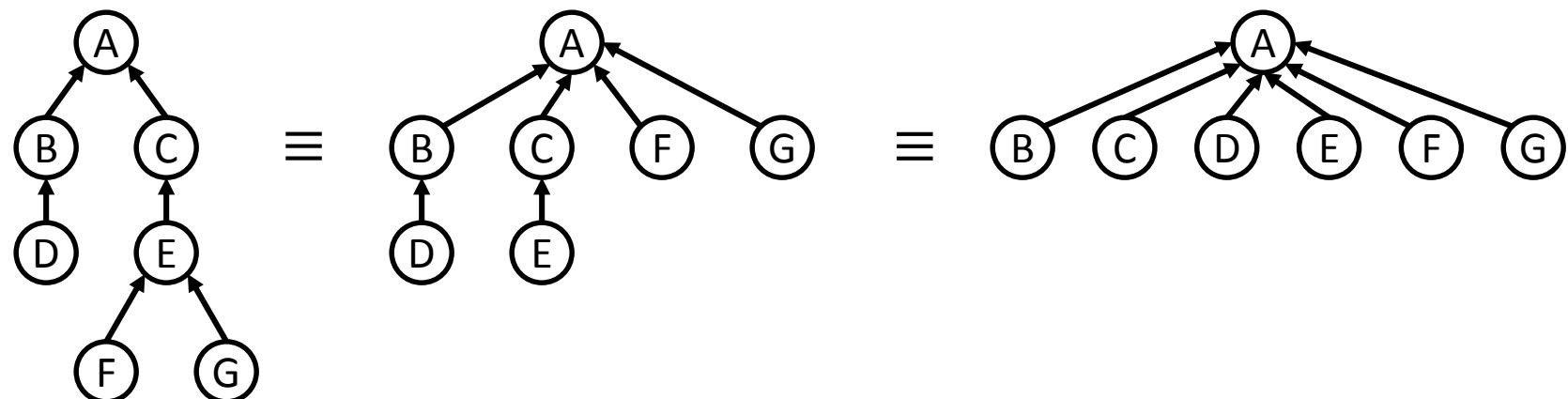


Property: Any root node of rank k has at least 2^k nodes in its tree.

Property: If there are n elements overall, there can be at most $n/2^k$ nodes of rank k . Therefore, all trees have height $\leq \log n$.

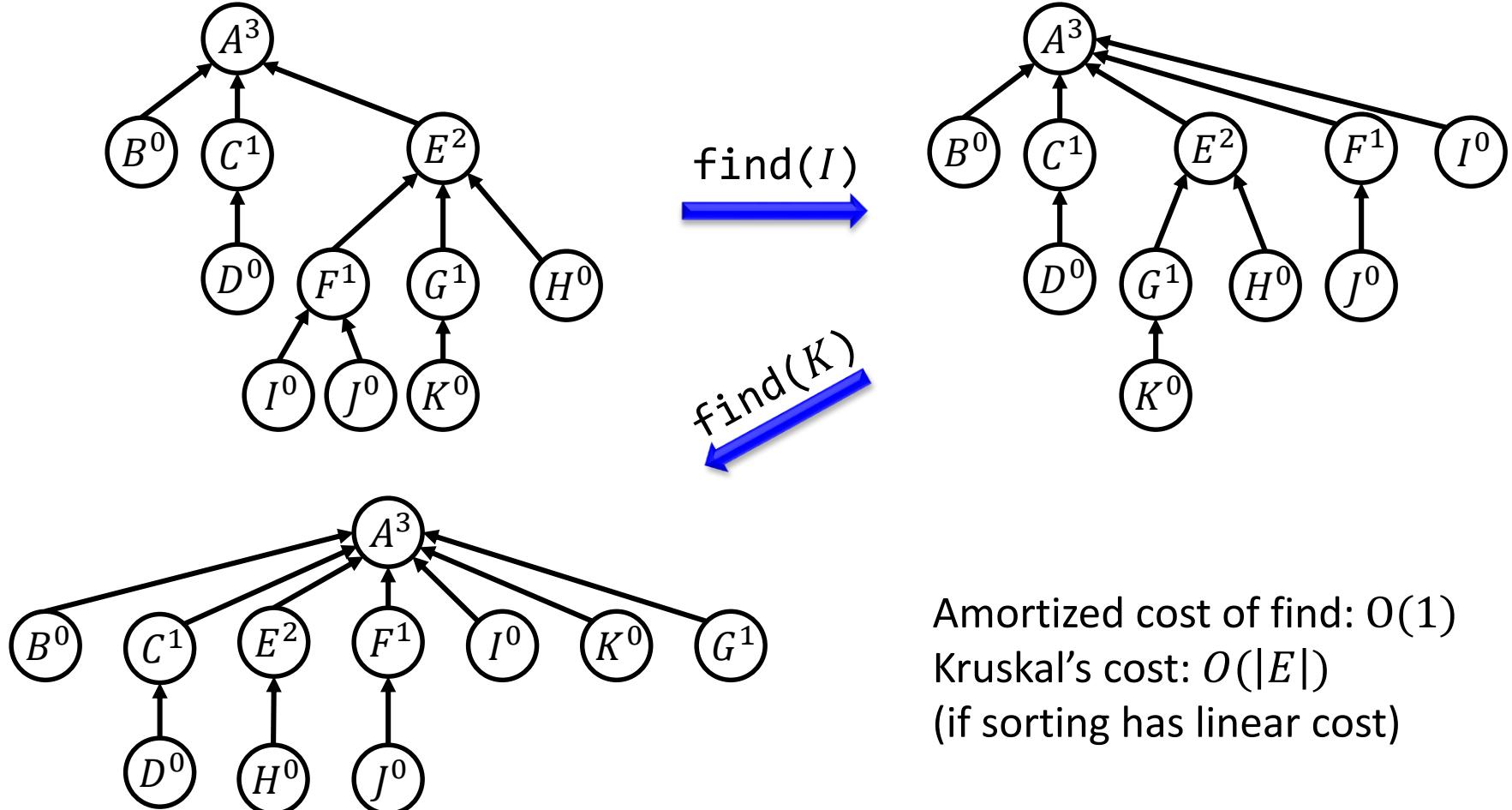
Disjoint sets: path compression

- Complexity of Kruskal's algorithm: $O(|E| \log |V|)$.
 - Sorting edges: $O(|E| \log |E|) \approx O(|E| \log |V|)$.
 - Find + union ($2 \cdot |E|$ times): $O(|E| \log |V|)$.
- How about if the edges are already sorted or sorting can be done in linear time (weights are small)?
- Path compression:

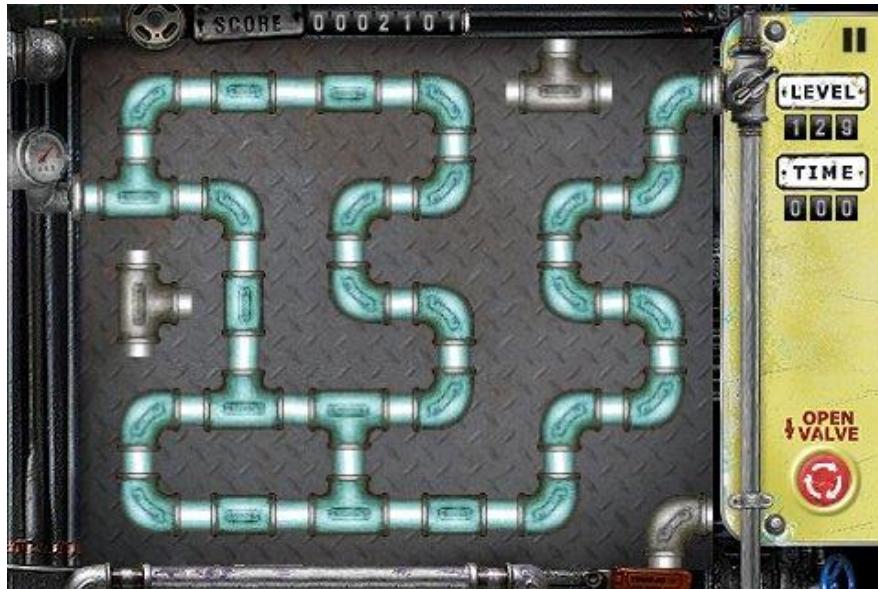


Disjoint sets: path compression

```
function find( $x$ ):  
    if  $x \neq \pi(x)$ :  $\pi(x) = \text{find}(\pi(x))$   
    return  $\pi(x)$ 
```

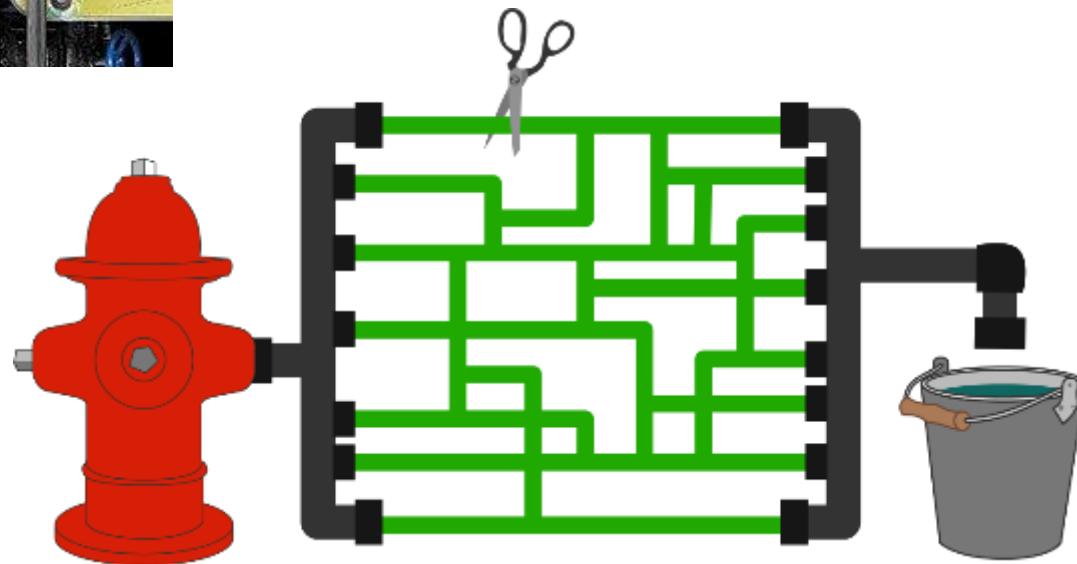


Max-flow/min-cut problems



OpenValve, by JAE HYUN LEE

What is the fewest number of green tubes that need to be cut so that no water will be able to flow from the hydrant to the bucket?



Max-flow/Min-cut algorithm. Brilliant.org.

<https://brilliant.org/wiki/max-flow-min-cut-algorithm/>

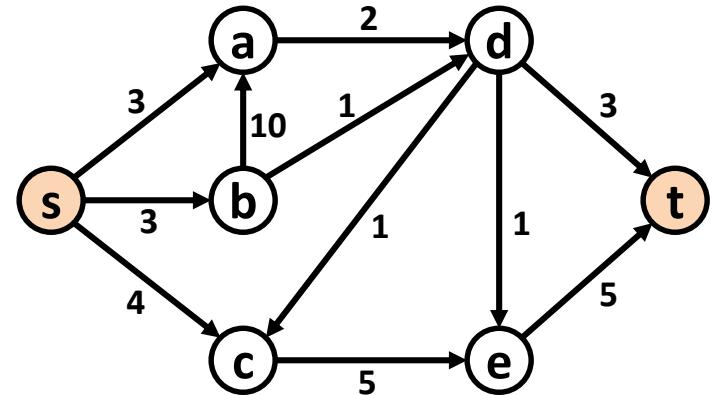
Max-flow/min-cut problems: applications

- Networks that carry data, water, oil, electricity, cars, etc.
 - How to maximize usage?
 - How to minimize cost?
 - How to maximize reliability?
- Multiple application domains:
 - Computer networks
 - Image processing
 - Computational biology
 - Airline scheduling
 - Data mining
 - Distributed computing
 - ...

Max-flow problem

Model:

- a directed graph $G = (V, E)$.
- Two special nodes $s, t \in V$.
- Capacities $c_e > 0$ on the edges.



Goal: assign a flow f_e to each edge e of the network satisfying:

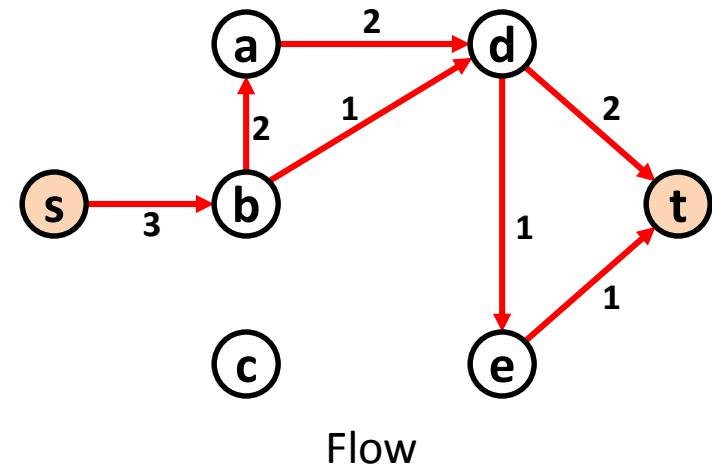
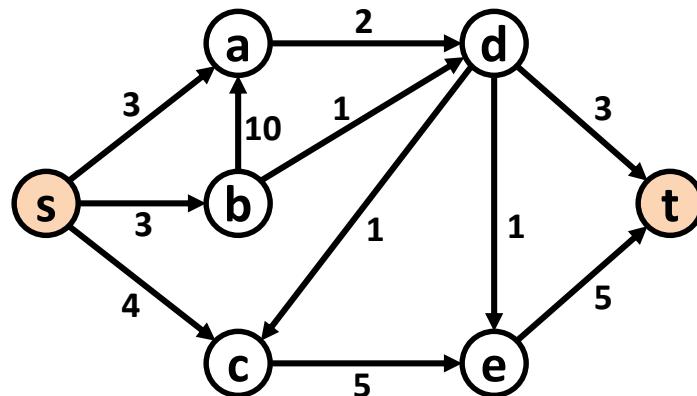
- $0 \leq f_e \leq c_e$ for all $e \in E$ (edge capacity not exceeded)
- For all nodes u (except s and t), the flow entering the node is equal to the flow exiting the node:

$$\sum_{(w,u) \in E} f_{wu} = \sum_{(u,z) \in E} f_{uz}.$$

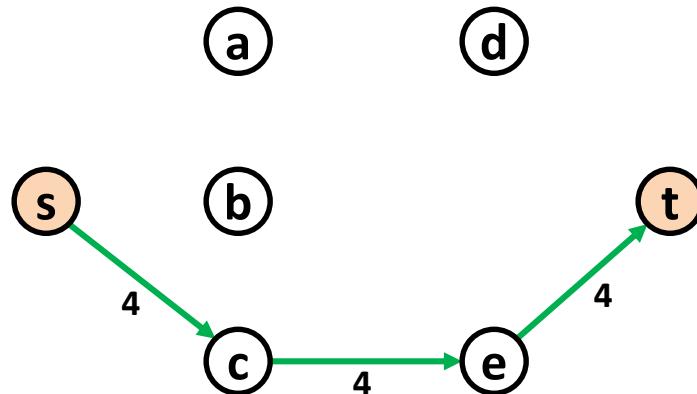
Size of a flow: total quantity sent from s to t (equal to the quantity leaving s):

$$\text{size}(f) = \sum_{(s,u) \in E} f_{su}$$

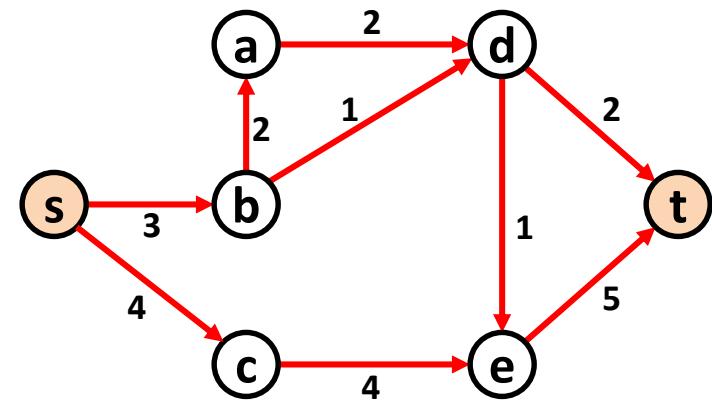
Augmenting paths



Given a flow, an ***augmenting path*** represents a feasible additional flow from s to t .

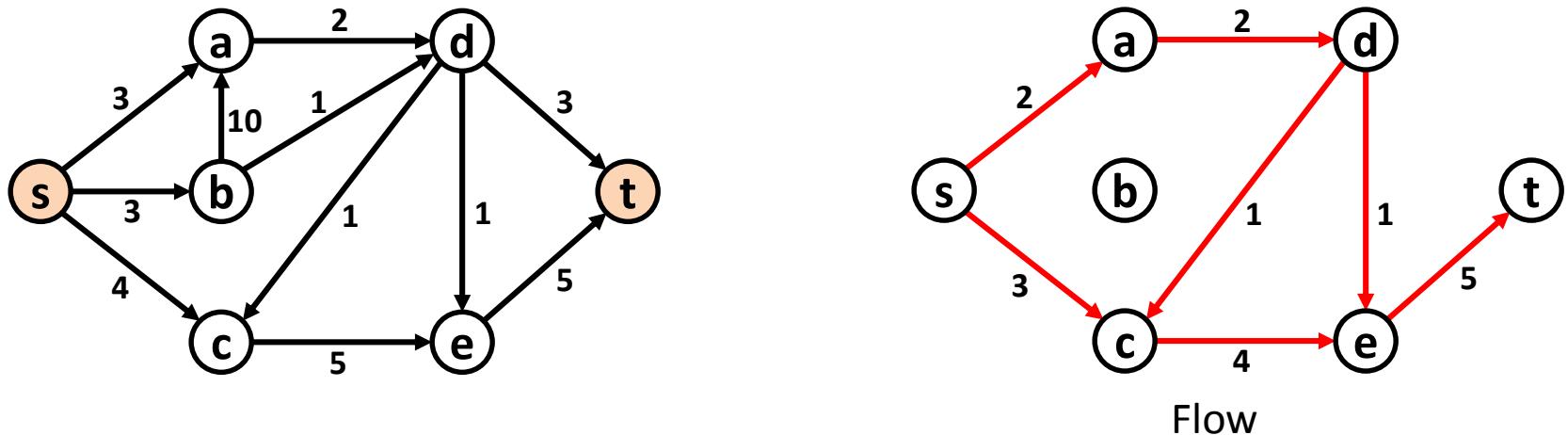


Augmenting path

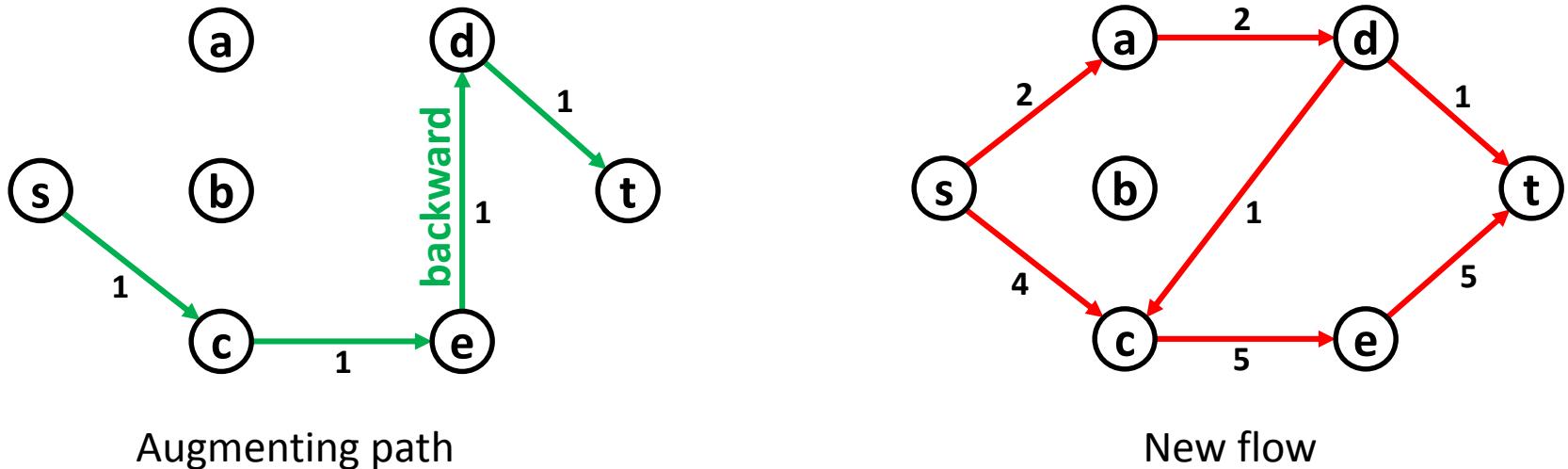


New flow

Augmenting paths



Augmenting paths can have *forward* and *backward* edges.

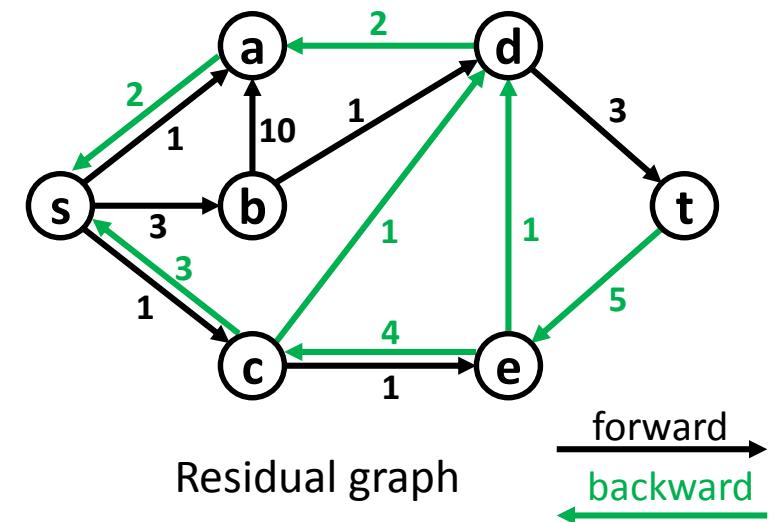
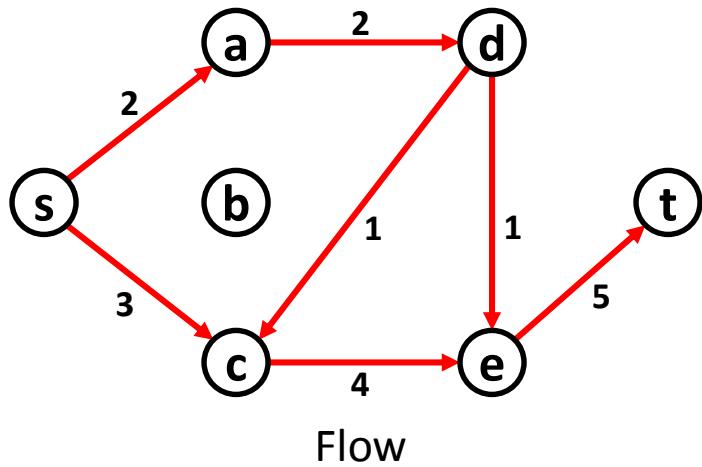
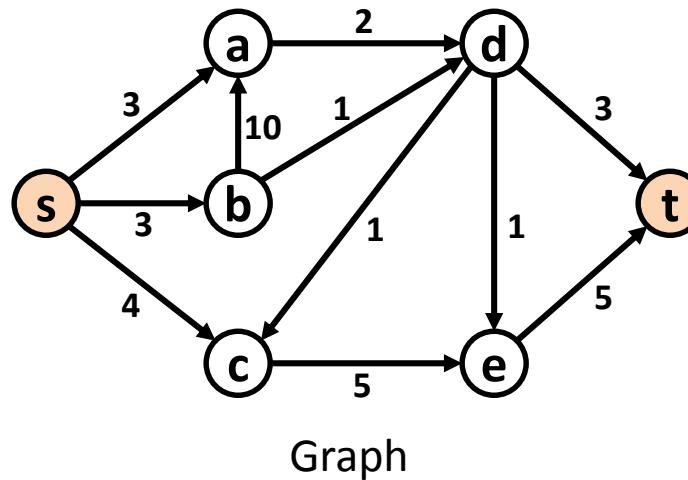


Augmenting paths

Given a flow f , an augmenting path is a directed path from s to t , which consists of edges from E , but not necessarily in the same direction. Each of these edges e satisfies exactly one of the following two conditions:

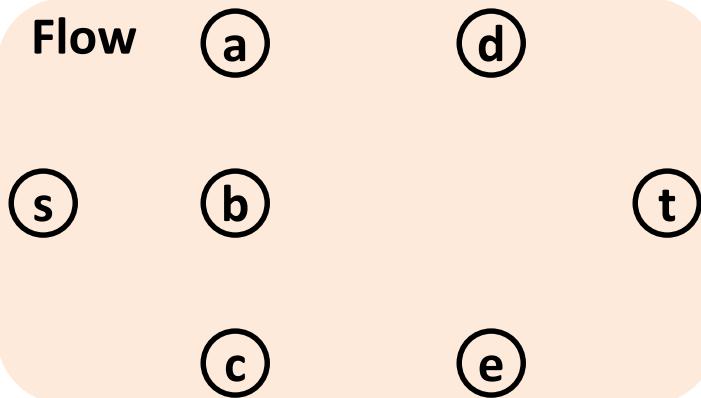
- e is in the same direction as in E (forward) and $f_e < c_e$. The difference $c_e - f_e$ is called the *slack* of the edge.
- e is in the opposite direction (backward) and $f_e > 0$. It represents the fact that some flow can be borrowed from the current flow.

Residual graph

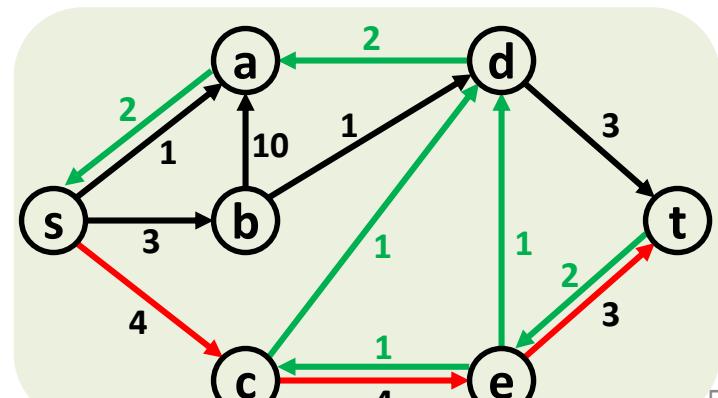
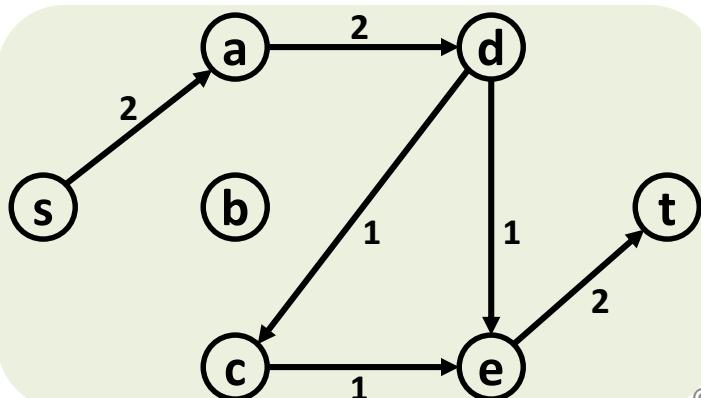
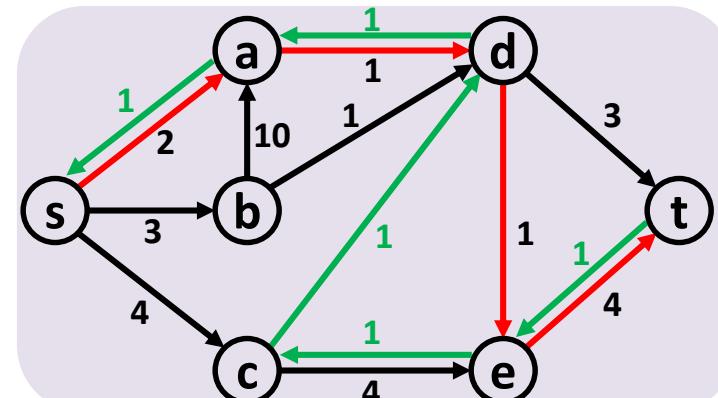
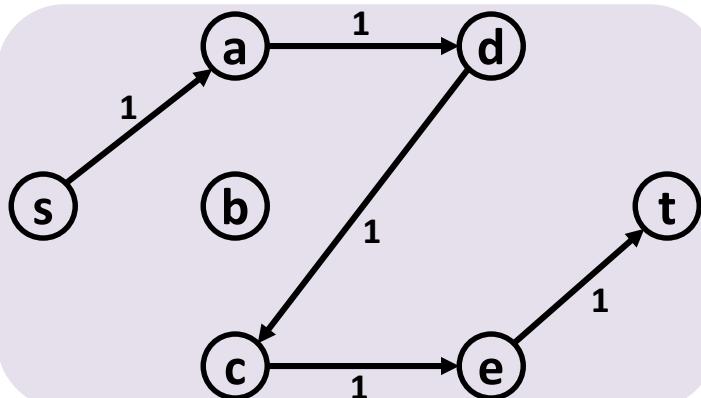
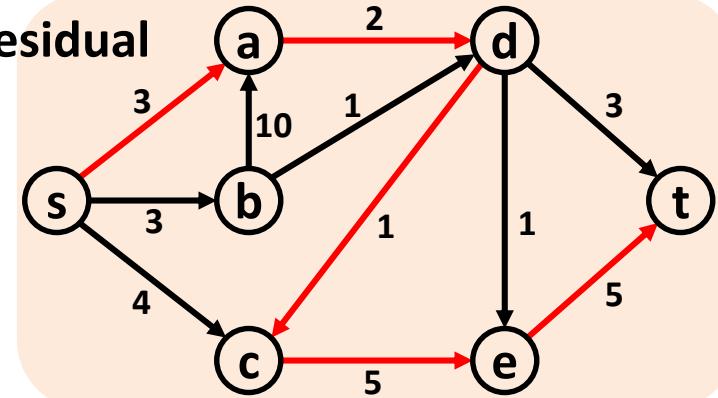


Ford-Fulkerson algorithm: example

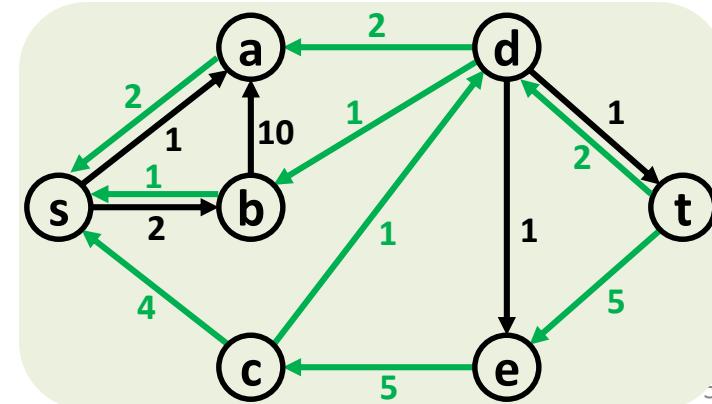
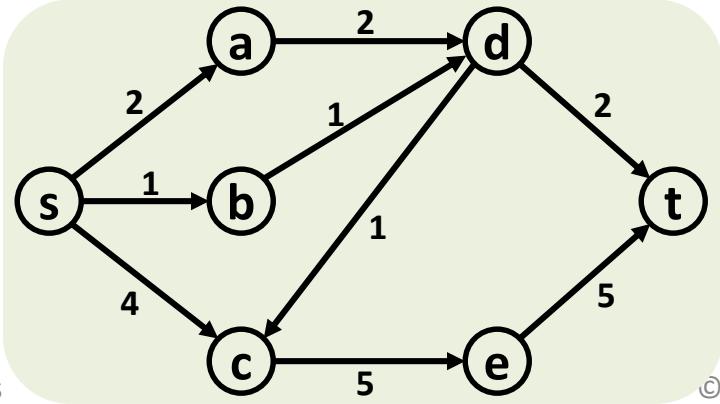
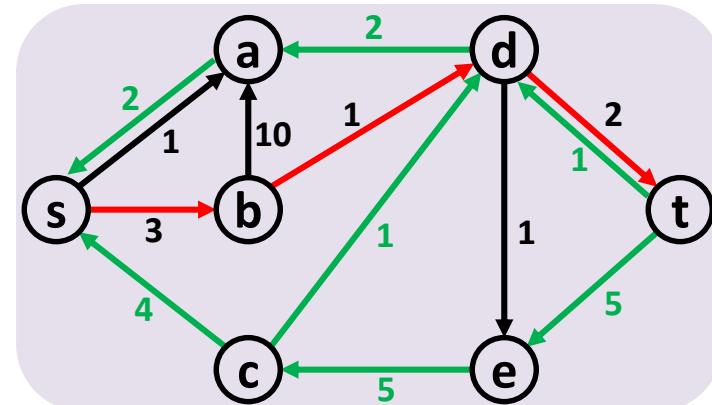
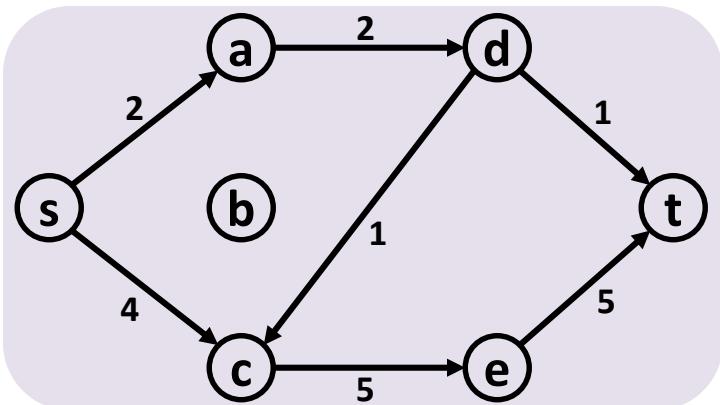
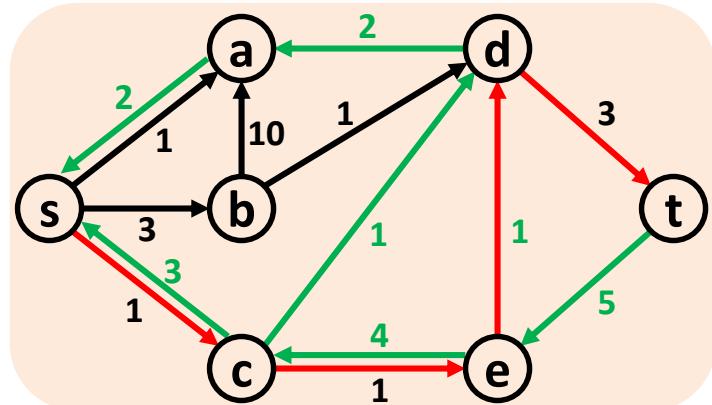
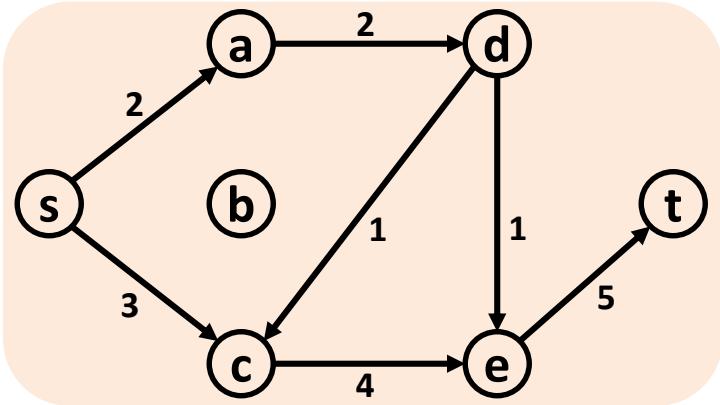
Flow



Residual



Ford-Fulkerson algorithm: example



Ford-Fulkerson algorithm

```
function Ford-Fulkerson( $G, s, t$ )
    // Input: A directed Graph  $G(V, E)$  with edge capacities  $c_e$ .
    //          $s$  and  $t$  and the source and target of the flow.
    // Output: A flow  $f$  that maximizes the size of the flow.
    //         For each  $(u, v) \in E$ ,  $f(v, u)$  represents its flow.

    for all  $(u, v) \in E$ :
         $f(u, v) = c(u, v)$  // Forward edges
         $f(v, u) = 0$  // Backward edges

    while there exists a path  $p = s \rightsquigarrow t$  in the residual graph:
         $f(p) = \min\{f(u, v) : (u, v) \in p\}$ 
        for all  $(u, v) \in p$ :
             $f(u, v) = f(u, v) - f(p)$ 
             $f(v, u) = f(v, u) + f(p)$ 
```

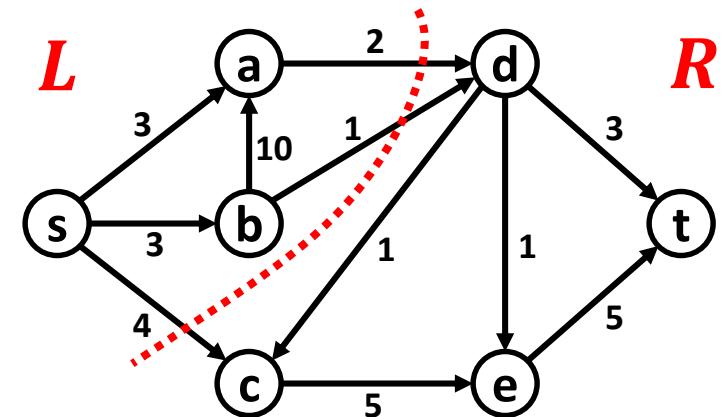
Ford-Fulkerson algorithm: complexity

- Finding a path in the residual graph requires $O(|E|)$ time (using BFS or DFS).
- How many iterations (augmenting paths) are required?
 - The worst case is really bad: $O(C \cdot |E|)$, with C being the largest capacity of an edge (if only integral values are used).
 - By selecting the path with fewest edges (using BFS) the maximum number of iterations is $O(|V| \cdot |E|)$.
 - By carefully selecting *fat* augmenting paths (using some variant of Dijkstra's algorithm), the number of iterations can be reduced.
- Ford-Fulkerson algorithm is $O(|V| \cdot |E|^2)$ if BFS is used to select the path with fewest edges.

Max-flow problem

Cut: An (s, t) -cut partitions the nodes into two disjoint groups, L and R , such that $s \in L$ and $t \in R$.

For any flow f and any (s, t) -cut (L, R) :
 $\text{size}(f) \leq \text{capacity}(L, R)$.



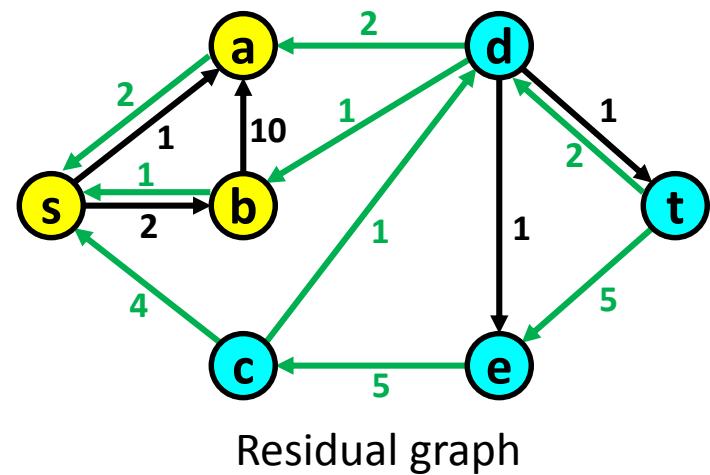
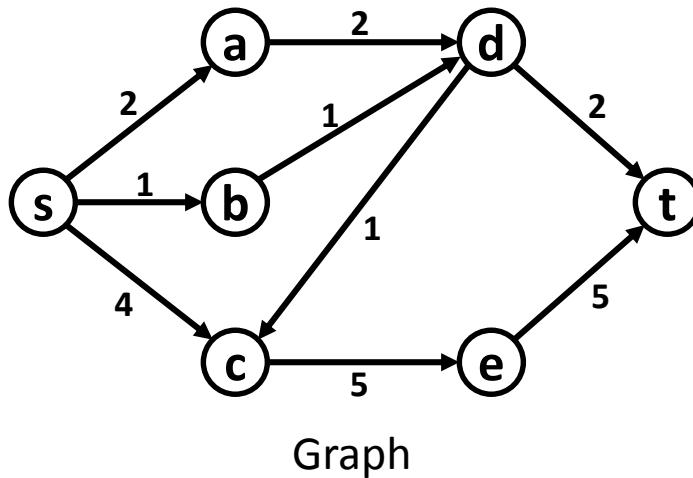
The max-flow min-cut theorem:

The size of the maximum flow equals the capacity of the smallest (s, t) -cut.

The augmenting-path theorem:

A flow is maximum iff it admits no augmenting path.

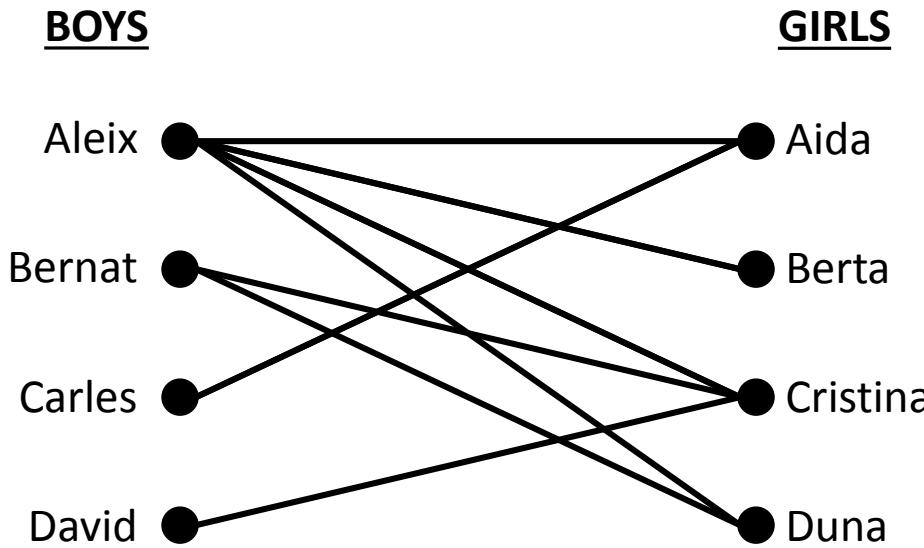
Min-cut algorithm



Finding a cut with minimum capacity:

1. Solve the max-flow problem with Ford-Fulkerson.
2. Compute L as the set of nodes reachable from s in the residual graph.
3. Define $R = V - L$.
4. The cut (L, R) is a min-cut.

Bipartite matching



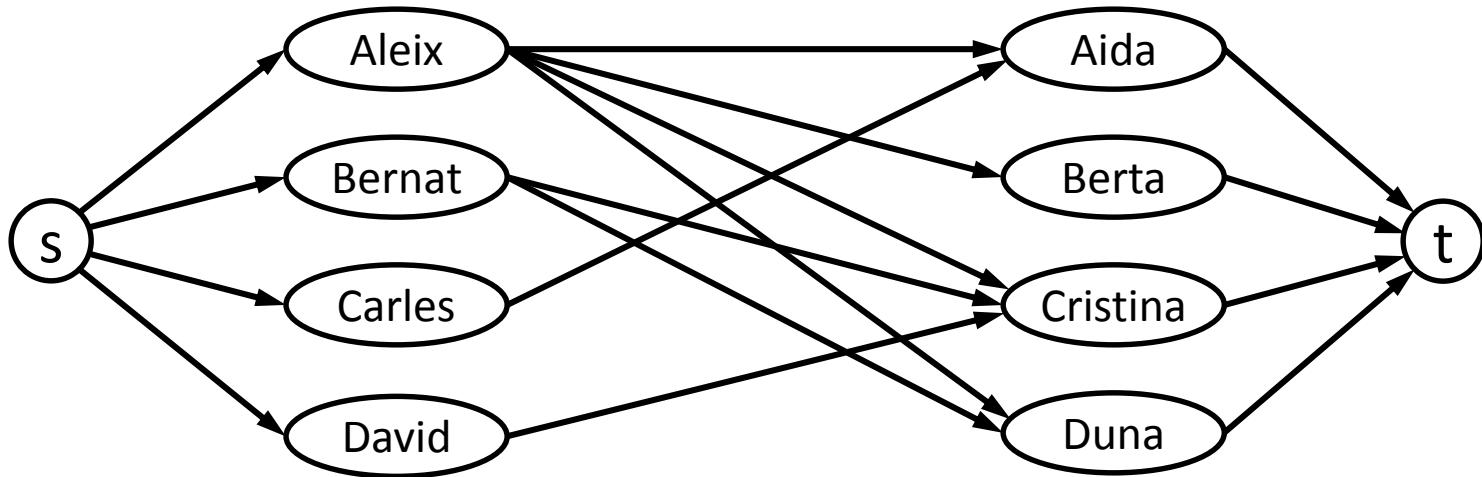
There is an edge between a boy and a girl if they like each other.

Can we pick couples so that everyone has exactly one partner that he/she likes?

Bad matching: if we pick (Aleix, Aida) and (Bernat, Cristina), then we cannot find couples for Berta, Duna, Carles and David.

A ***perfect matching*** would be: (Aleix, Berta), (Bernat, Duna), (Carles, Aida) and (David, Cristina).

Bipartite matching



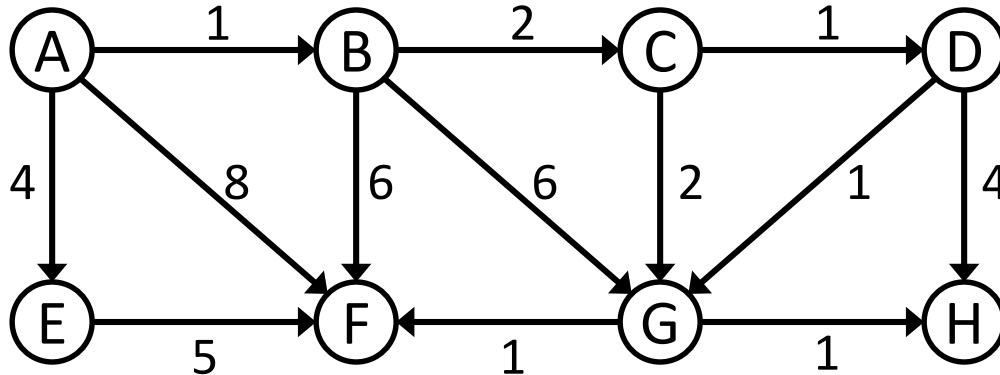
Reduced to a max-flow problem with $c_e = 1$.

Question: can we always guarantee an integer-valued flow?

Property: if all edge capacities are integer, then the optimal flow found by Ford-Fulkerson's algorithm is integral. It is easy to see that the flow of the augmenting path found at each iteration is integral.

EXERCISES

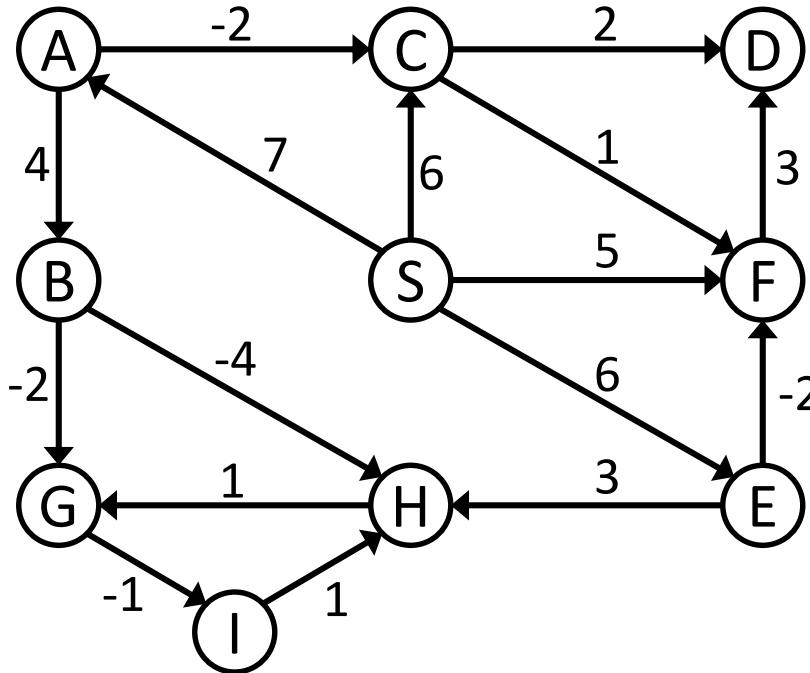
Dijkstra (4.1 in [DPV2008])



Run Dijkstra's algorithm starting at node A:

- Draw a table showing the intermediate distance values of all the nodes at each iteration
- Show the final shortest-path tree

Bellman-Ford (4.2 in [DPV2008])



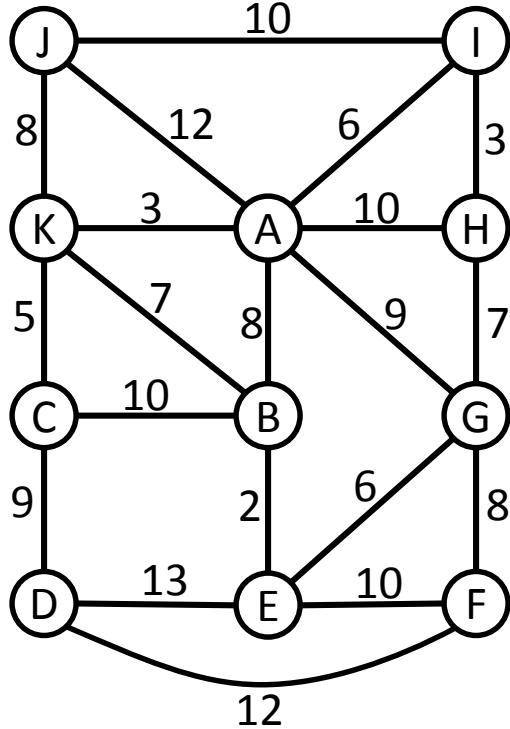
Run Bellman-Ford algorithm starting at node S:

- Draw a table showing the intermediate distance values of all the nodes at each iteration
- Show the final shortest-path tree

New road (4.20 in [DPV2008])

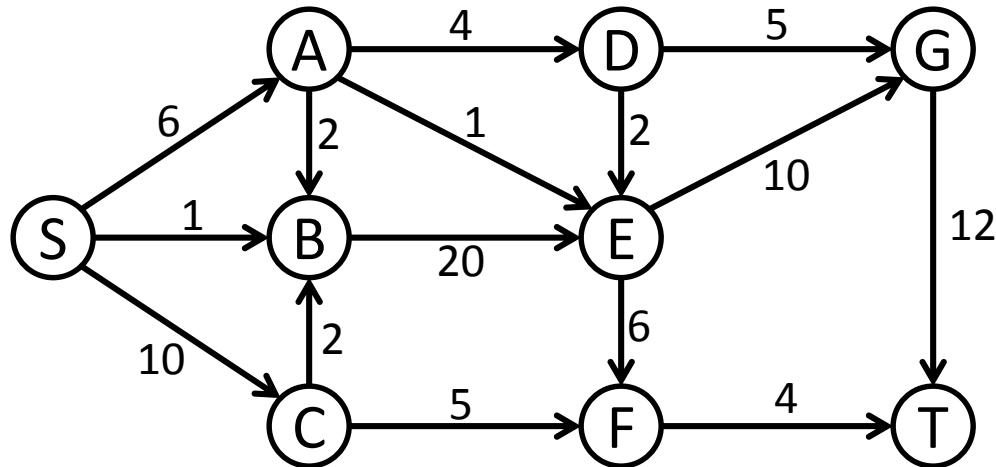
There is a network of roads $G = (V, E)$ connecting a set of cities V . Each road in E has an associated length l_e . There is a proposal to add one new road to this network, and there is a list E' of pairs of cities between which the new road can be built. Each such potential road $e' \in E'$ has an associated length. As a designer for the public works department you are asked to determine the road $e' \in E'$ whose addition to the existing network G would result in the maximum decrease in the driving distance between two fixed cities s and t in the network. Give an efficient algorithm for solving this problem.

Minimum Spanning Trees



- Calculate the shortest path tree from node A using Dijkstra's algorithm.
- Calculate the MST using Prim's algorithm. Indicate the sequence of edges added to the tree and the evolution of the distance table.
- Calculate the MST using Kruskal's algorithms. Indicate the sequence of edges added to the tree and the evolution of the disjoint sets. In case of a tie between two edges, try to select the one that is not in Prim's tree.

Flow networks



[from DPV2008]

- Find the maximum flow from S to T. Give a sequence of augmenting paths that lead to the maximum flow.
- Draw the residual graph after finding the maximum flow.
- Find a minimum cut between S and T.

Blood transfusion

Enthusiastic celebration of a sunny day at a prominent northeastern university has resulted in the arrival at the university's medical clinic of 169 students in need of emergency treatment. Each of the 169 students requires a transfusion of one unit of whole blood. The clinic has supplies of 170 units of whole blood. The number of units of blood available in each of the four major blood groups and the distribution of patients among the groups is summarized below.

Blood type	A	B	O	AB
Supply	46	34	45	45
Demand	39	38	42	50

Type A patients can only receive type A or O; type B patients can receive only type B or O; type O patients can receive only type O; and type AB patients can receive any of the four types.

Give a maxflow formulation that determines a distribution that satisfies the demands of a maximum number of patients.

Can we have enough blood units for all the students?

Source: Sedgewick and Wayne, Algorithms, 4th edition, 2011.

Edge-disjoint paths

Given a digraph $G = (V, E)$ and vertices $s, t \in V$, describe an algorithm that finds the maximum number of edge-disjoint paths from s to t .

Note: two paths are edge-disjoint if they do not share any edge.

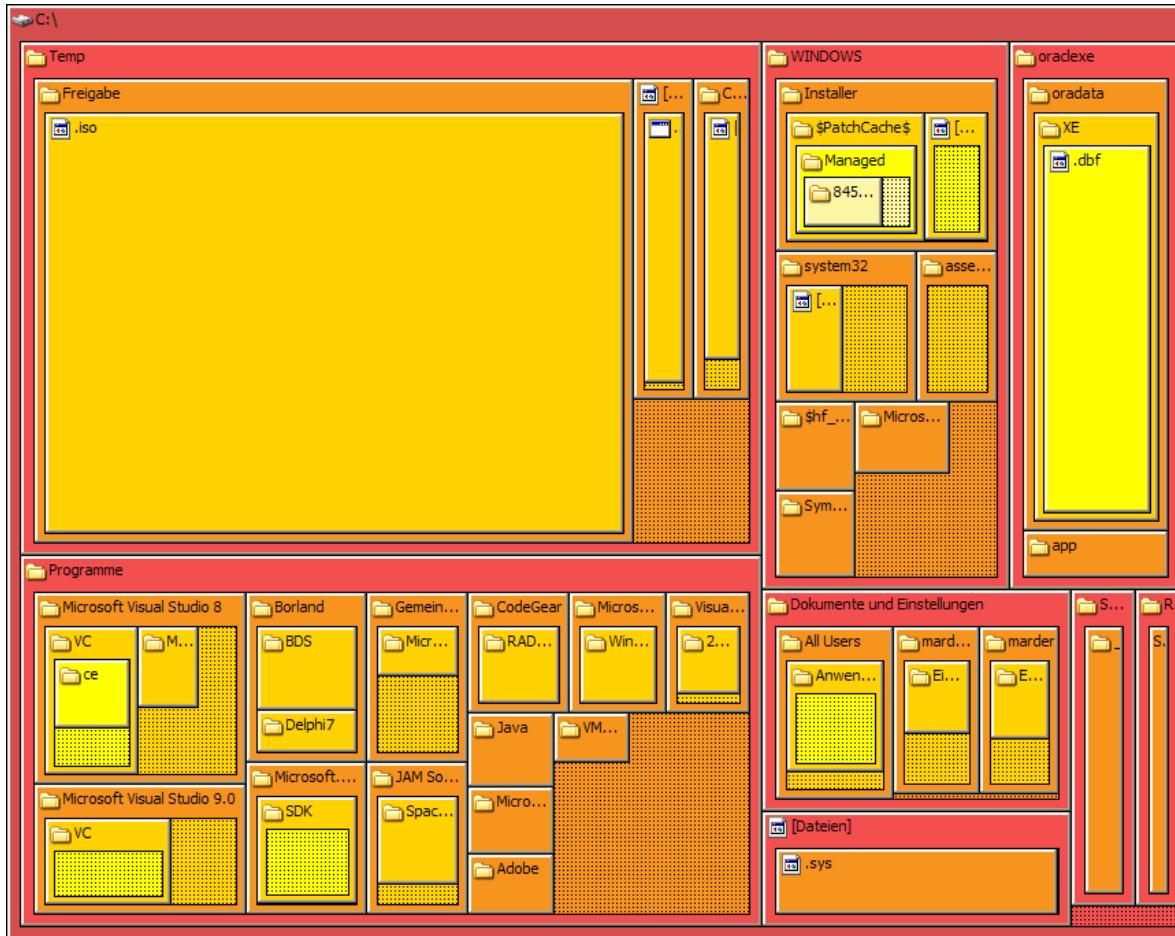
Trees



Jordi Cortadella and Jordi Petit
Department of Computer Science

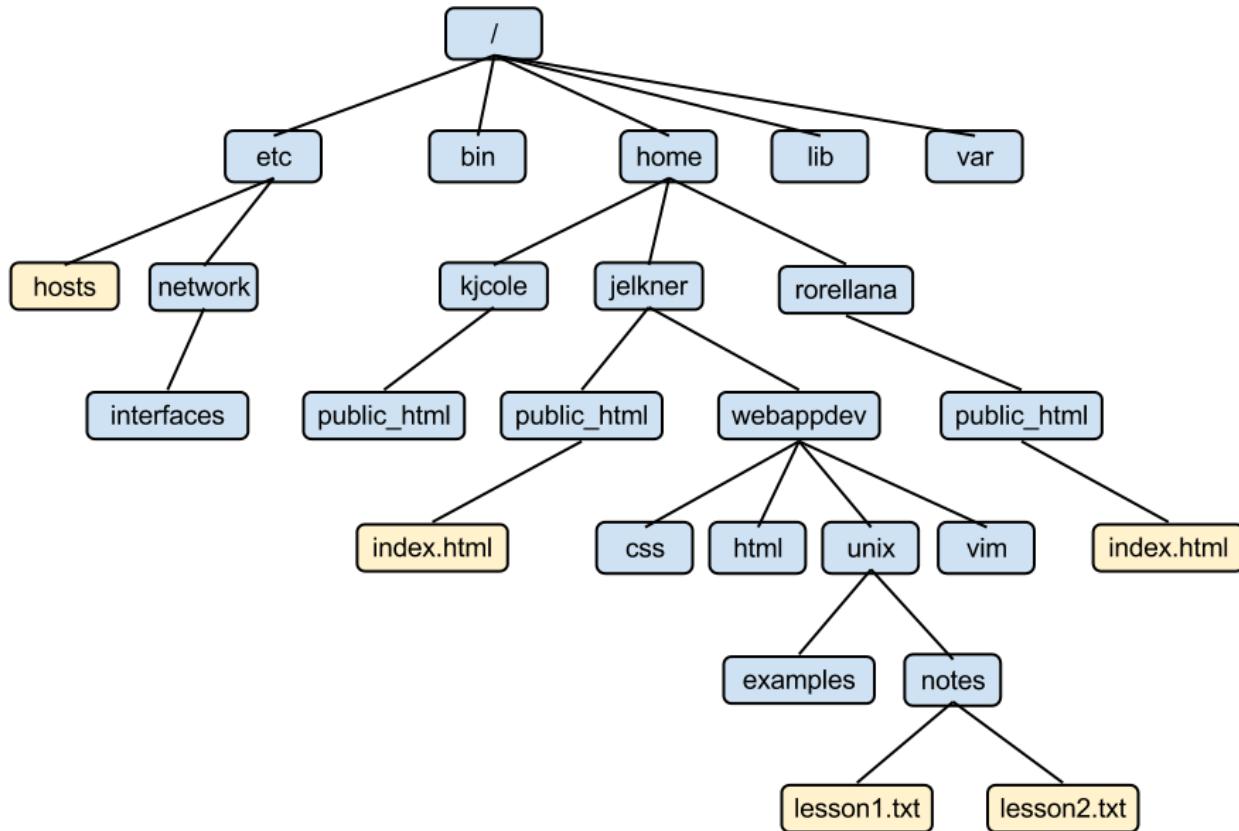
Trees

Data are often organized hierarchically



source: https://en.wikipedia.org/wiki/Tree_structure

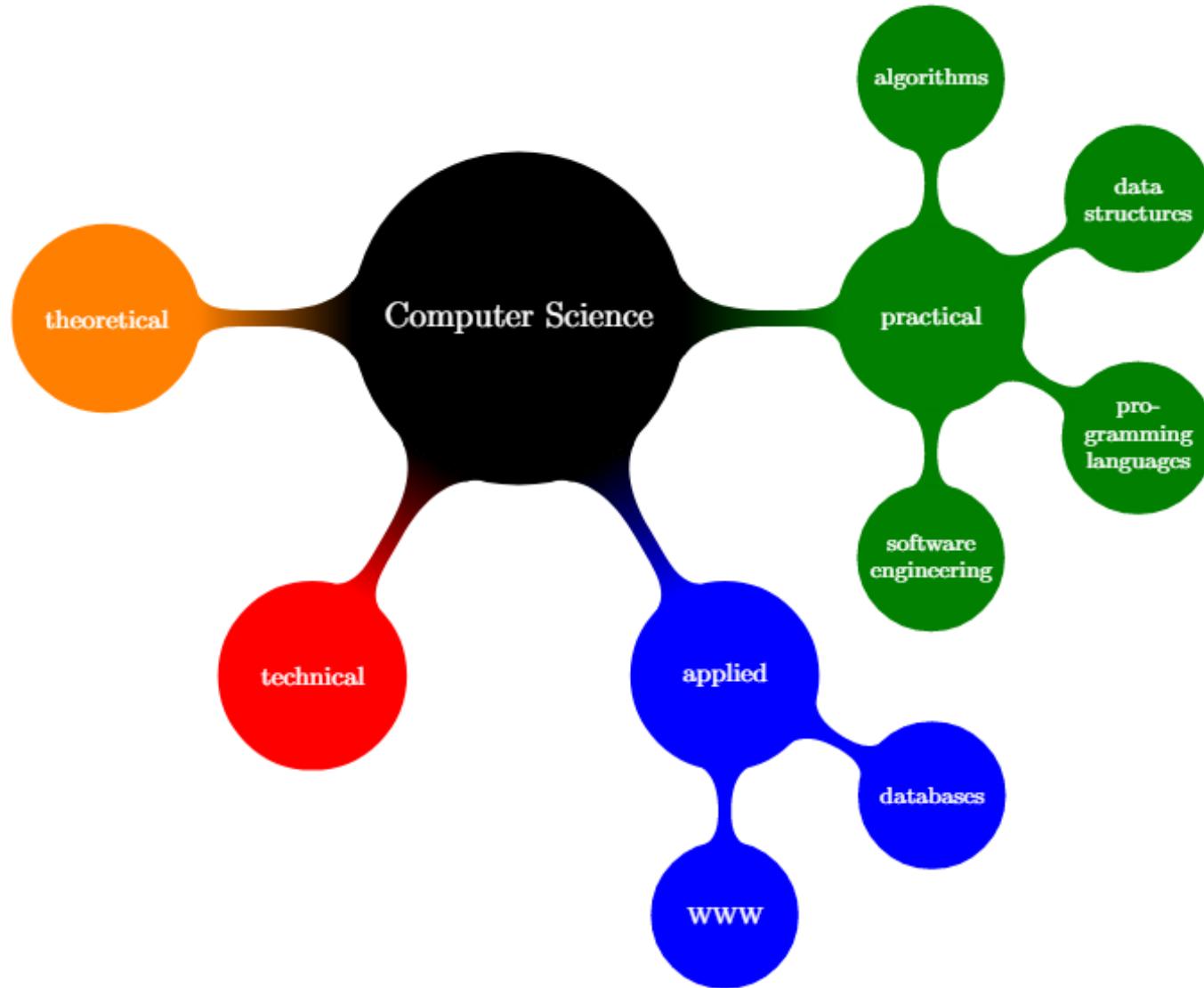
Filesystems



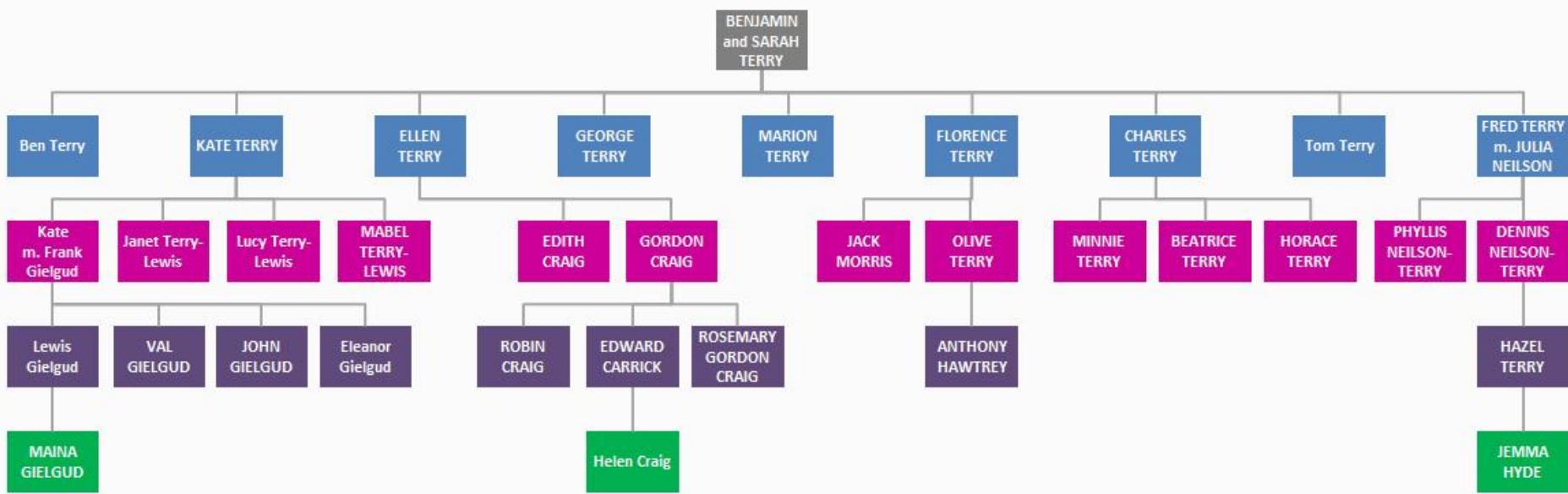
```
jelkner@rms:~$ tree webappdev/
webappdev/
├── css
│   └── examples
│       └── notes
├── html
│   └── examples
│       └── notes
└── unix
    ├── examples
    │   └── notes
    │       ├── lesson1.txt
    │       └── lesson2.txt
    └── vim
        ├── examples
        └── notes

12 directories, 2 files
jelkner@rms:~$
```

Mind maps

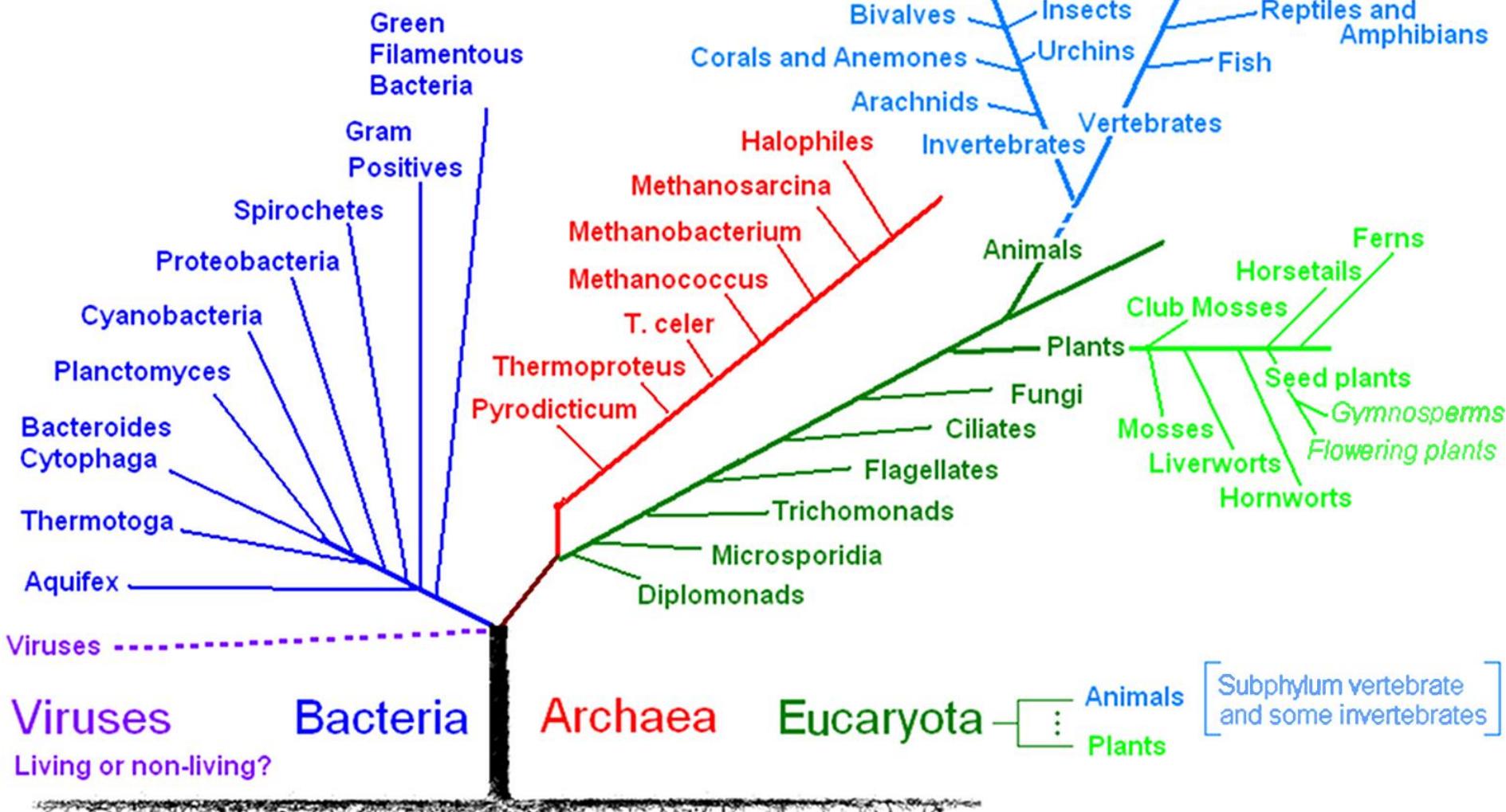


Genealogical trees

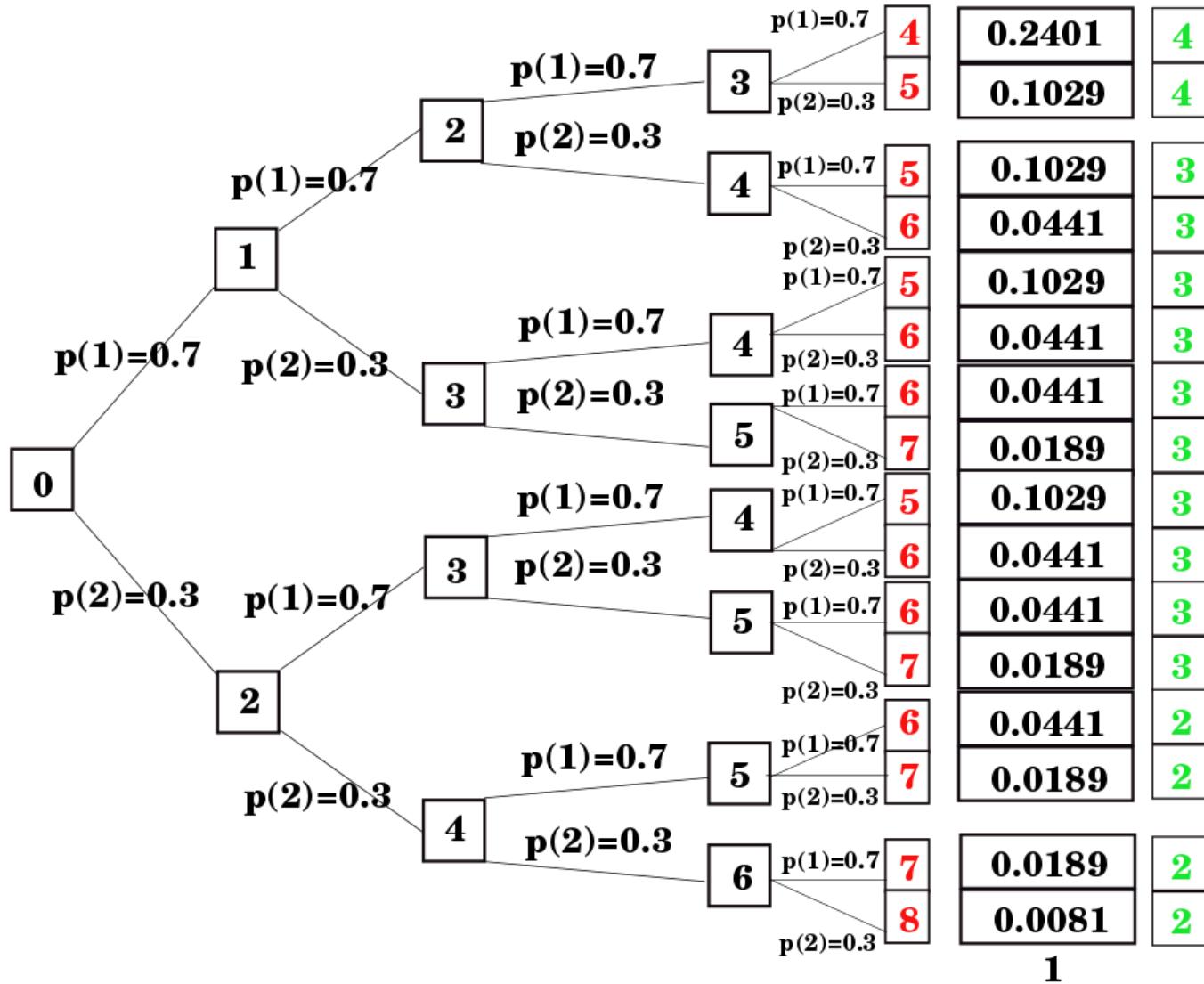


Tree of Life

<http://www.greennature.ca/>



Probability trees



Parse trees

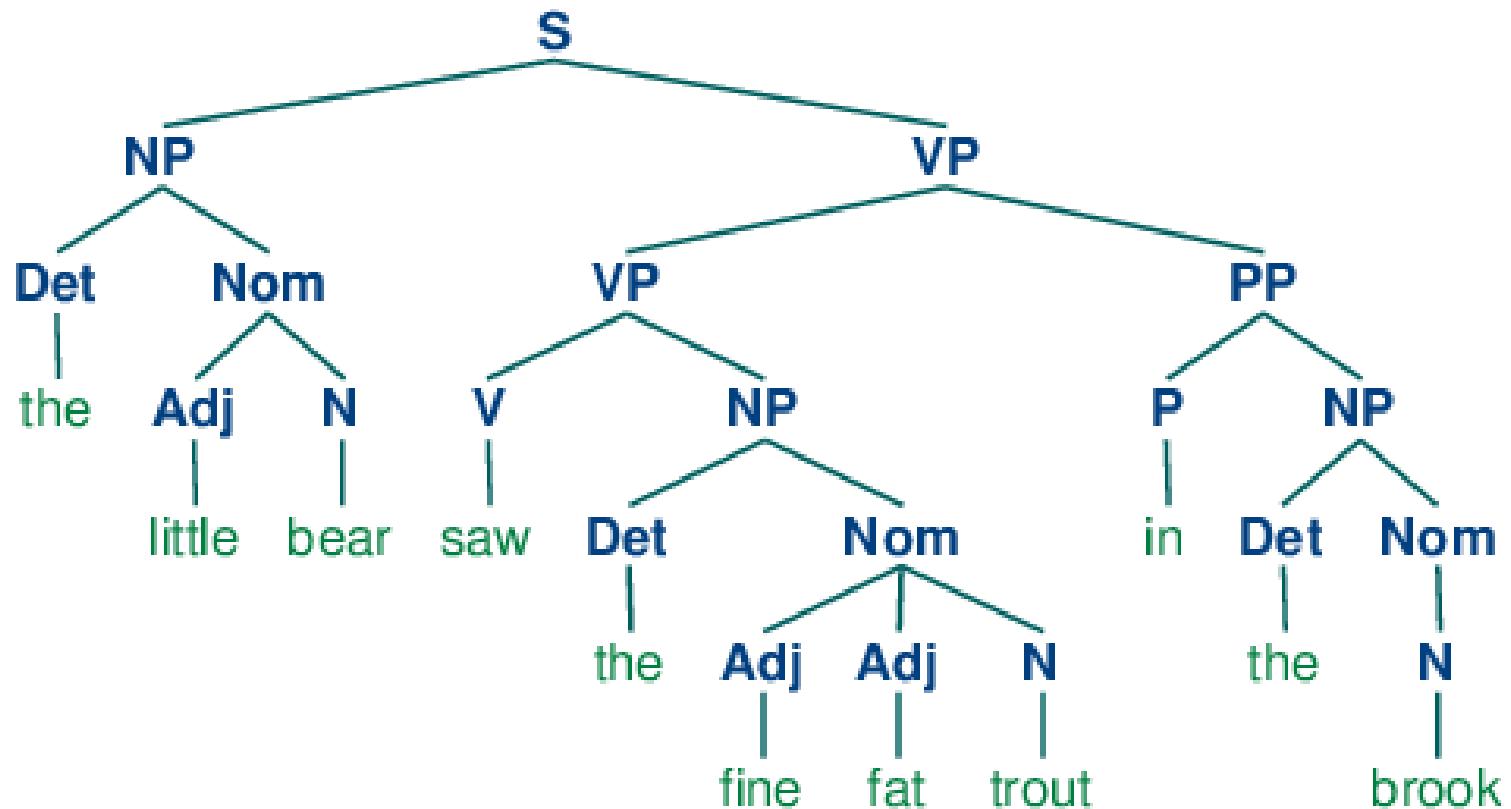
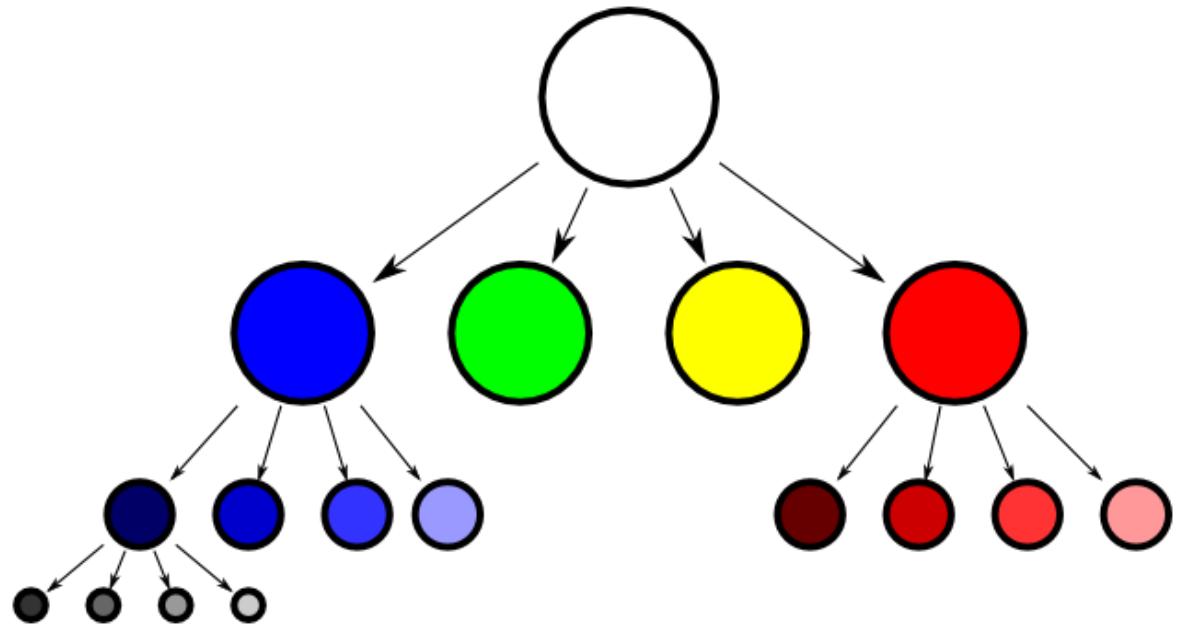
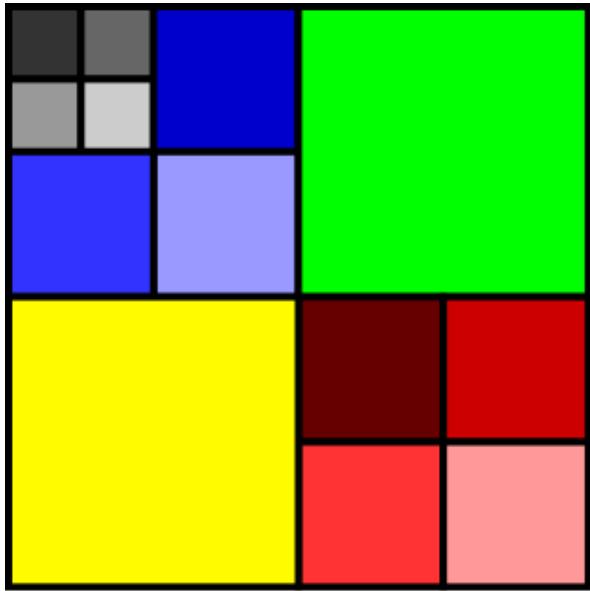
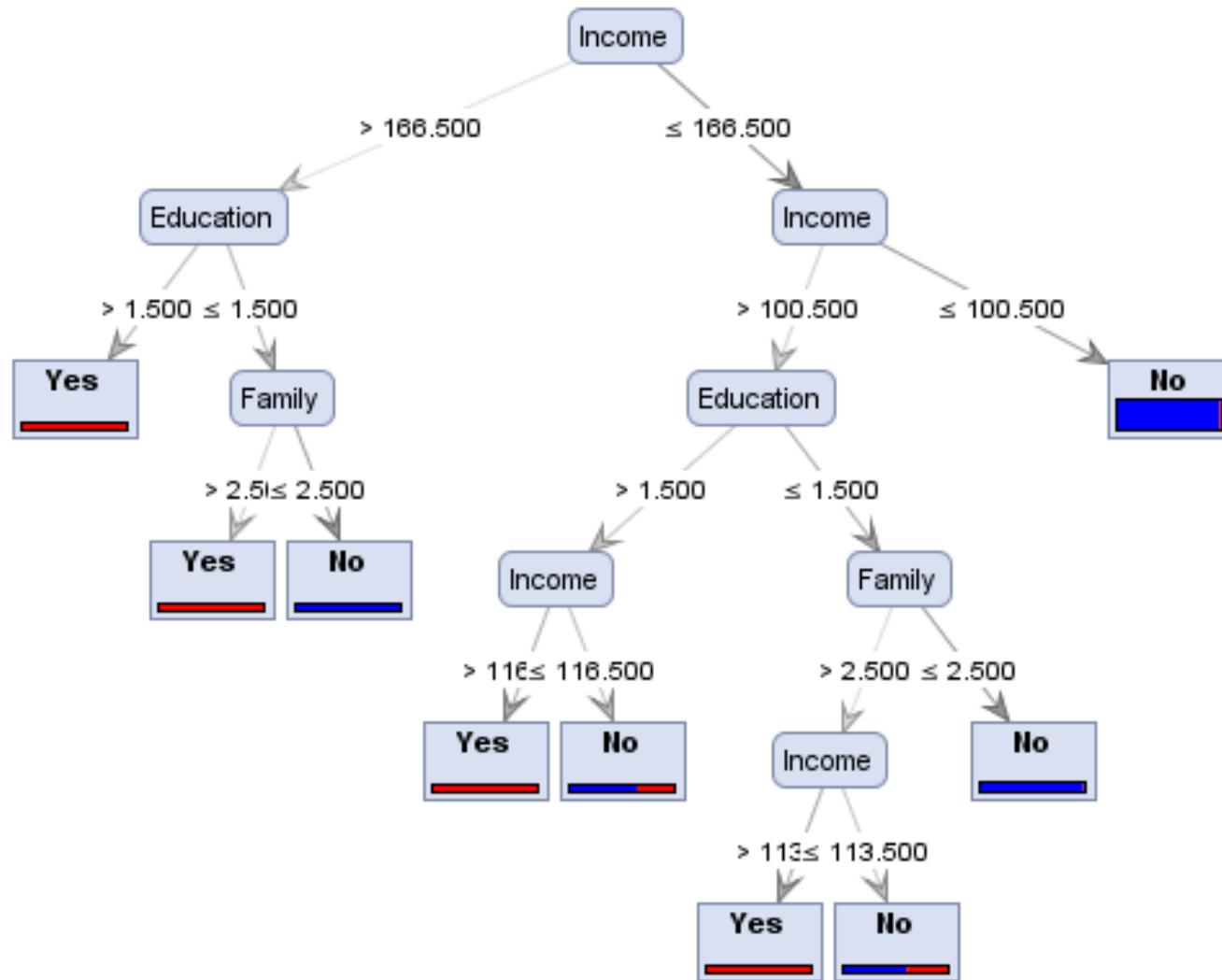


Image representation (quad-trees)



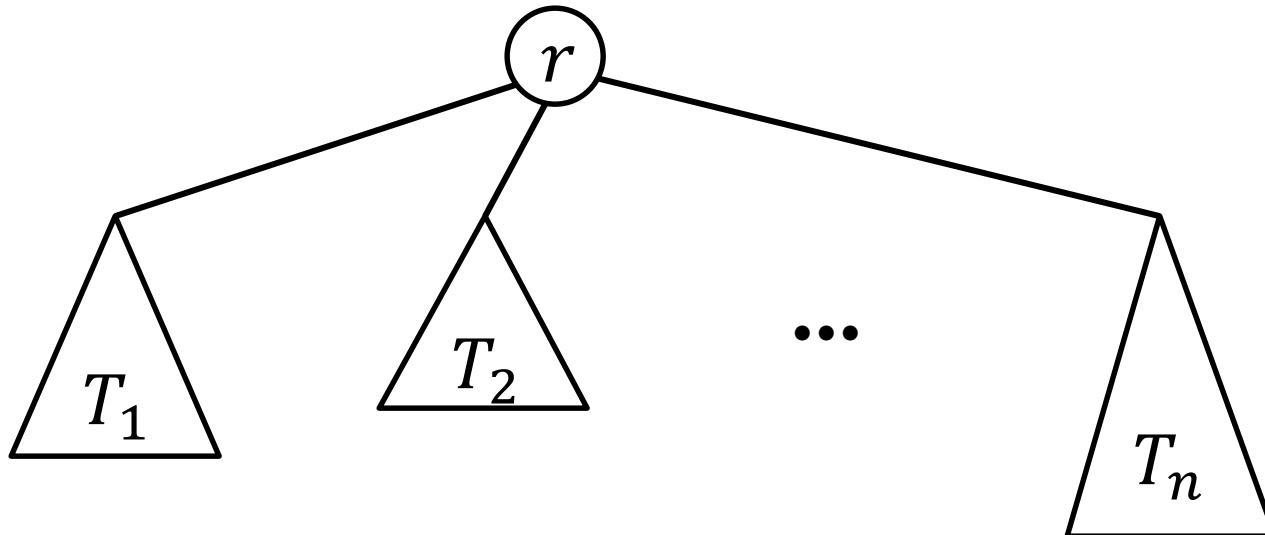
Decision trees



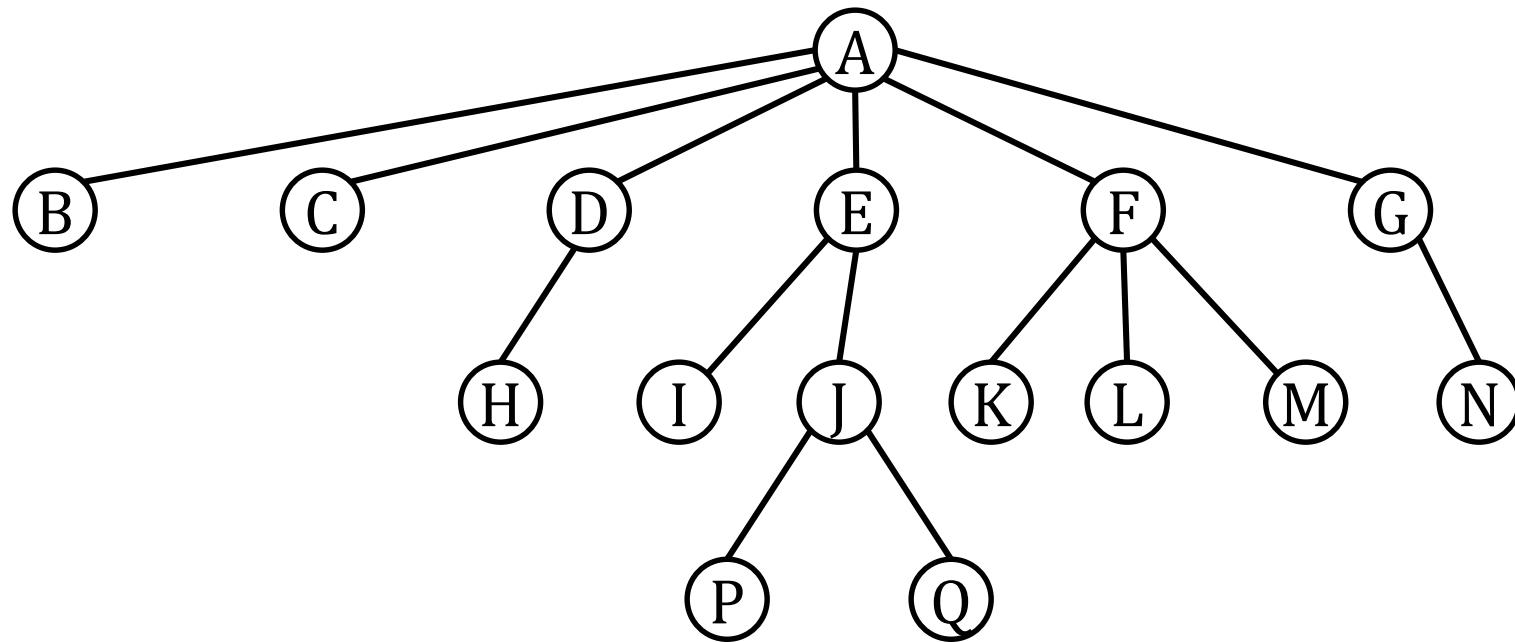
source: <http://www.simafore.com/blog/bid/94454/A-simple-explanation-of-how-entropy-fuels-a-decision-tree-model>

Tree: definition

- Graph theory: a tree is an undirected graph in which any two vertices are connected by exactly one path.
- Recursive definition (CS). A non-empty tree T consists of:
 - a root node r
 - a list of trees T_1, T_2, \dots, T_n that hierarchically depend on r .

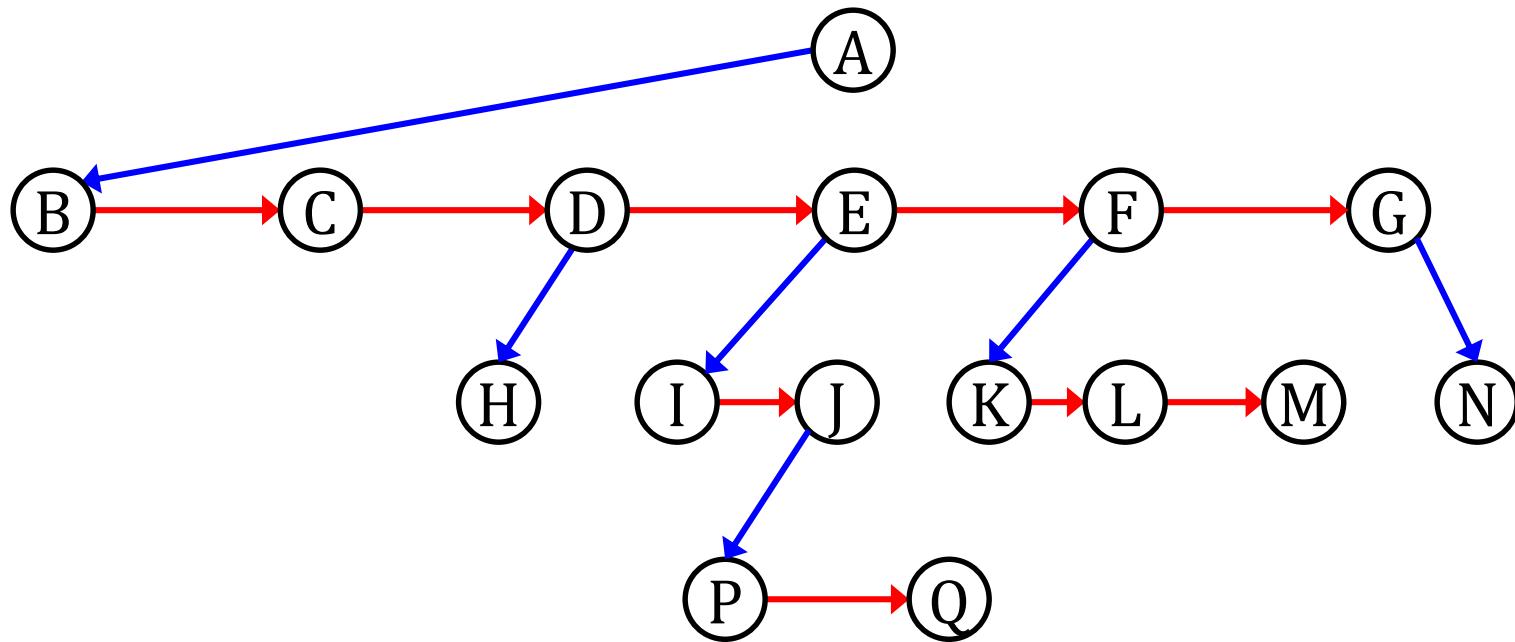


Tree: nomenclature



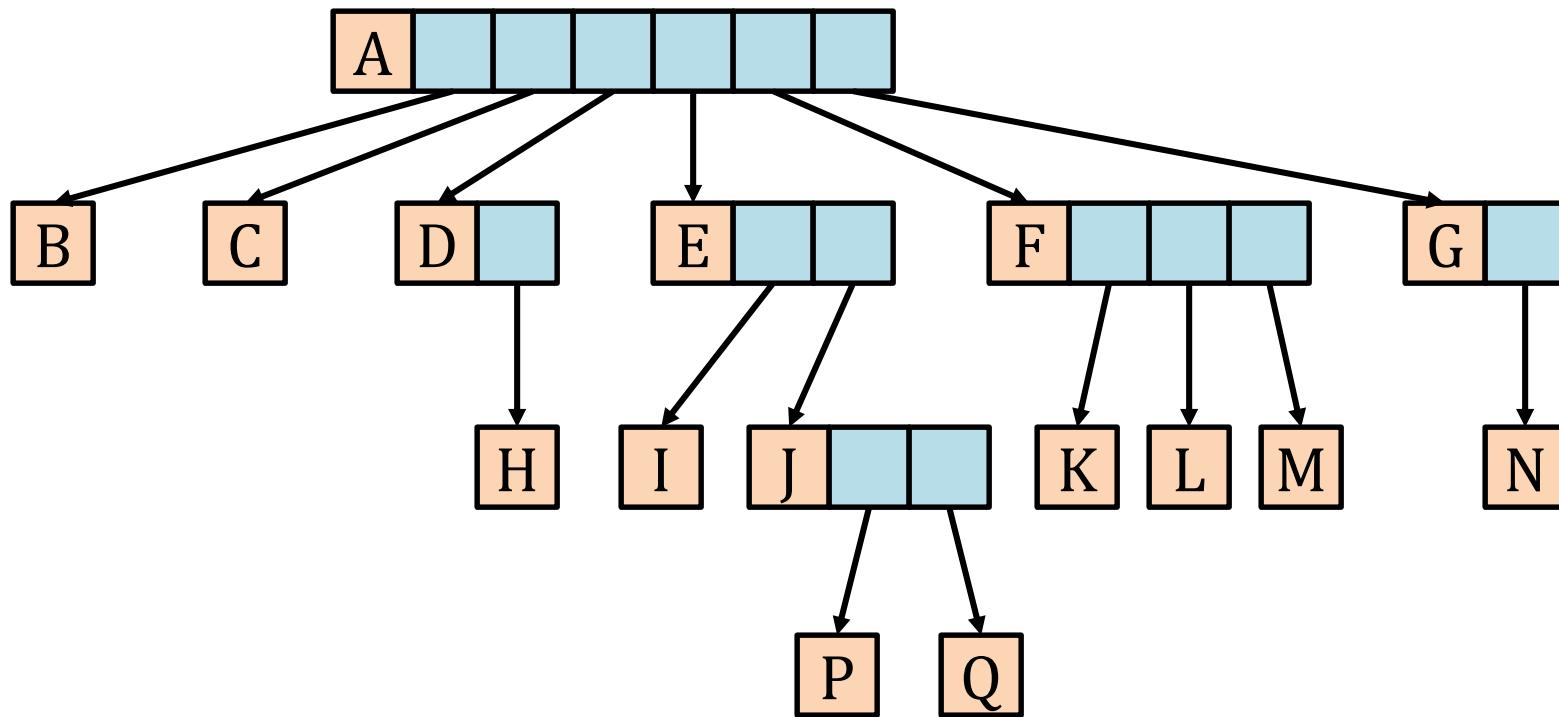
- A is the **root** node.
- Nodes with no children are **leaves** (e.g., B and P).
- Nodes with the same parent are **siblings** (e.g., K, L and M).
- The **depth** of a node is the length of the path from the root to the node. Examples: $\text{depth}(A)=0$, $\text{depth}(L)=2$, $\text{depth}(Q)=3$.

Tree: representation with linked lists



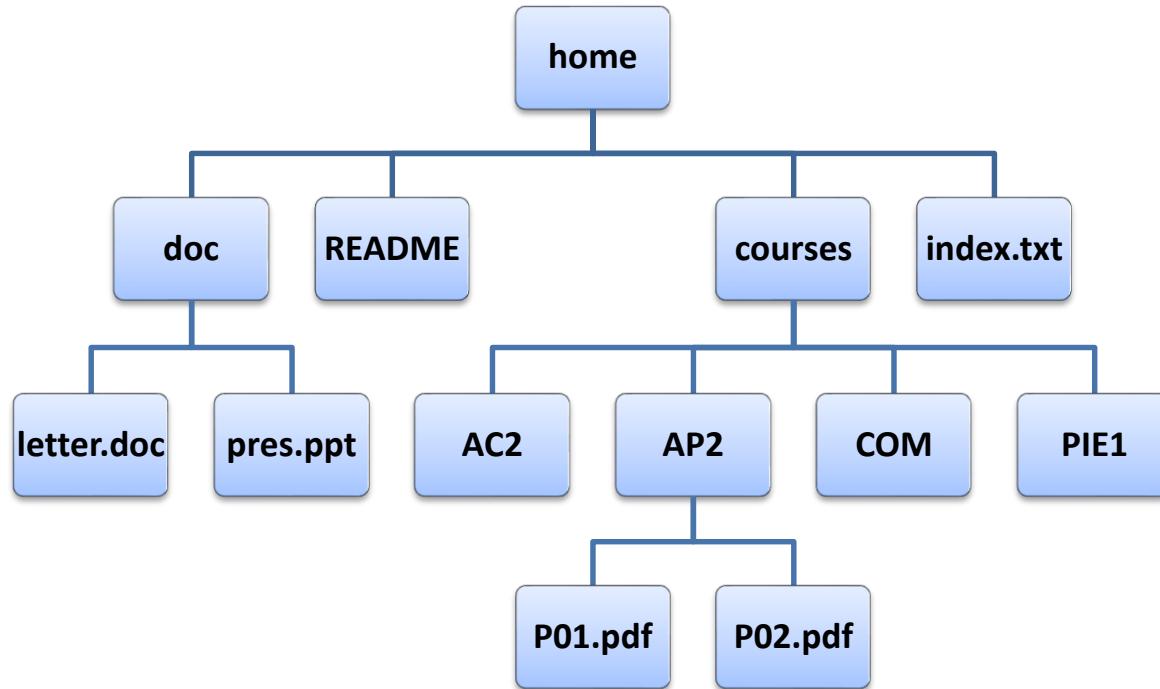
```
struct TreeNode {  
    Type element;  
    list<TreeNode> children; // Linked list of children  
};
```

Tree: representation with vectors



```
struct TreeNode {  
    Type element;  
    vector<TreeNode> children; // Vector of children  
};
```

Print a tree



```
home  
doc  
letter.doc  
pres.ppt  
README  
courses  
AC2  
AP2  
P01.pdf  
P02.pdf  
COM  
PIE1  
index.txt
```

```
struct Tree {  
    string name;  
    vector<Tree> children;  
};
```

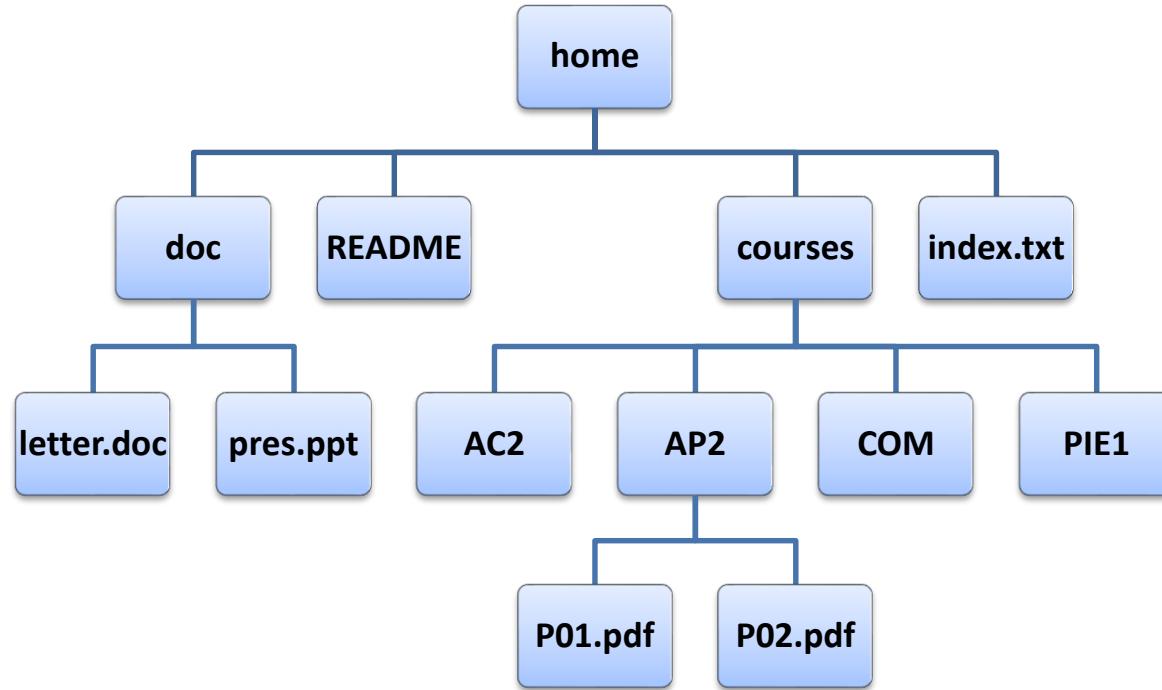
```
print(const Tree& T, int depth=0);
```

Print a tree

```
/** Prints a tree indented according to depth.  
 * Pre: The tree is not empty. */  
print(const Tree& T, int depth) {  
  
    // Print the root indented by 2*depth  
    cout << string(2*depth, ' ') << T.name << endl;  
  
    // Print the children with depth + 1  
    for (const Tree& child: T.children)  
        print(child, depth + 1);  
}
```

This function executes a *preorder* traversal of the tree:
each node is processed *before* the children.

Print a tree (postorder traversal)



letter.doc
pres.ppt
doc
README
AC2
P01.pdf
P02.pdf
AP2
COM
PIE1
courses
index.txt
home

Postorder traversal: each node is processed after the children.

Print a tree (postorder traversal)

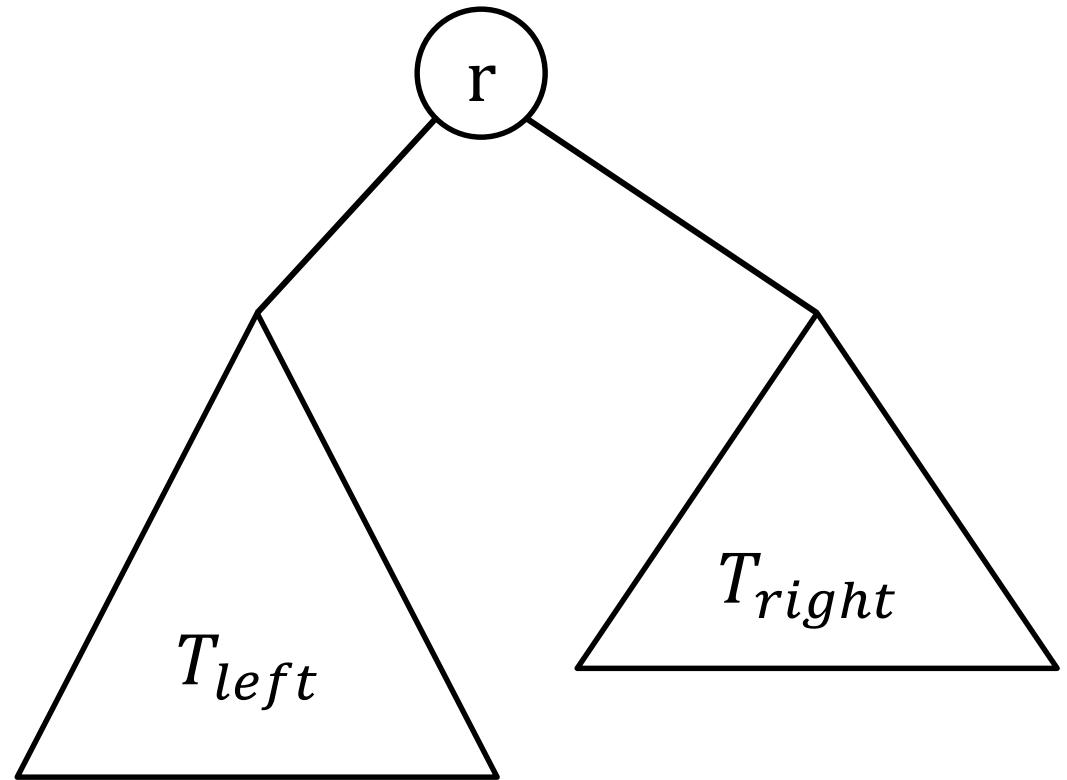
```
/** Prints a tree (in postorder) indented according to depth.  
 * Pre: The tree is not empty. */  
printPostOrder(const Tree& T, int depth) {  
  
    // Print the children with depth + 1  
    for (const Tree& child: T.children)  
        printPostOrder(child, depth + 1);  
  
    // Print the root indented by 2*depth  
    cout << string(2*depth, ' ') << T.name << endl;  
}
```

This function executes a **postorder** traversal of the tree:
each node is processed *after* the children.

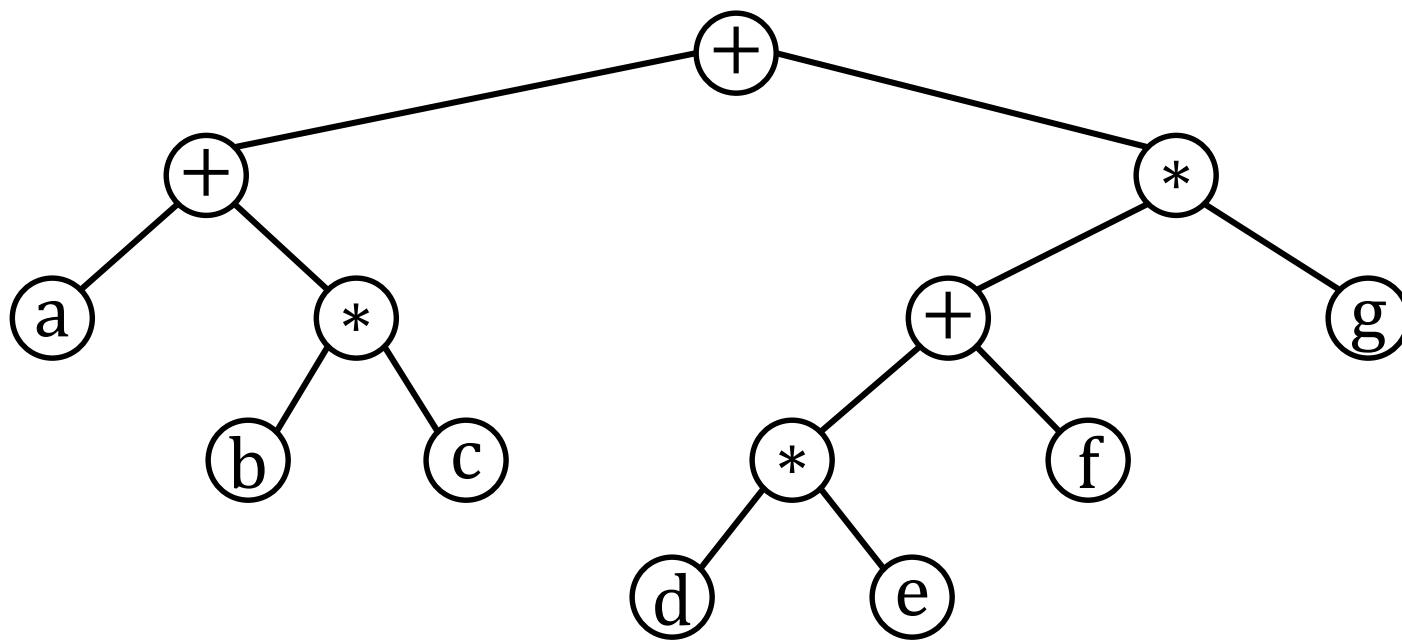
Binary trees

Nodes with at most two children.

```
struct BinTree {  
    Type element;  
    BinTree* left;  
    BinTree* right;  
};
```



Example: expression trees



Expression tree for: $a + b*c + (d*e + f) * g$

Postfix representation: a b c * + d e * f + g * +

How can the postfix representation be obtained?

Example: expression trees

```
struct ExprTree {  
    char op; // operand or operator  
    ExprTree* left;  
    ExprTree* right;  
};
```

Expressions are represented by strings in postfix notation in which the characters 'a'...'z' represent operands and the characters '+' and '*' represent operators.

```
/** Builds an expression tree from a correct  
 * expression represented in postfix notation. */  
ExprTree* buildExpr(const string& expr);  
  
/** Generates a string with the expression in  
 * infix notation. */  
string infixExpr(const ExprTree* T);  
  
/** Evaluates an expression taking V as the value of the  
 * variables (e.g., V['a'] contains the value of a). */  
int evalExpr(const ExprTree* T, const map<char,int>& V);
```

Example: expression trees

```
ExprTree* buildExpr(const string& expr) {
    stack<ExprTree*> S;
    // Visit the chars of the string sequentially
    for (char c: expr) {
        if (c >= 'a' and c <= 'z') {
            // We have an operand in {'a'...'z'}. Create a leaf node.
            S.push(new ExprTree{c, nullptr, nullptr});
        } else {
            // c is an operator ('+' or '*')
            ExprTree* right = S.top();
            S.pop();
            ExprTree* left = S.top();
            S.pop();
            S.push(new ExprTree{c, left, right});
        }
    }
    // The stack has only one element and is freed after return
    return S.top();
}
```

Example: expression trees

```
/** Returns a string with an infix representation of T. */
string infixExpr(const ExprTree* T) {

    // Let us first check the base case (an operand)
    if (T->left == nullptr) return string(1, T->op);

    // We have an operator. Return ( T->left T->op T->right )
    return "(" +
        infixExpr(T->left) +
        T->op +
        infixExpr(T->right) +
    ")";

}
```

Inorder traversal: node is visited *between* the left and right children.

Example: expression trees

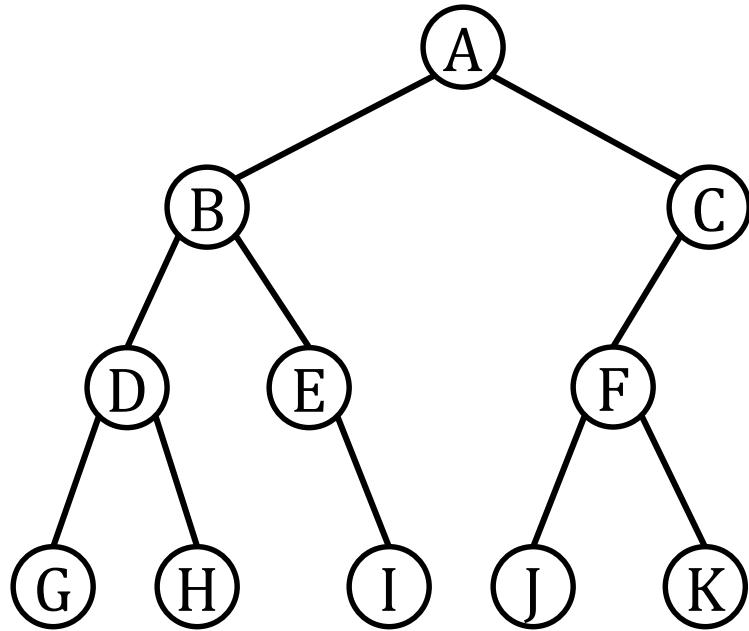
```
/** Evaluates an expression taking V as the value of the
 * variables (e.g., V['a'] contains the value of a). */
int evalExpr(const ExprTree* T, const map<char,int>& V) {
    if (T->left == nullptr) return V[T->op];
    int l = evalExpr(T->left, V);
    int r = evalExpr(T->right, V);
    return T->op == '+' ? l+r : l*r;
}

/** Example of usage of ExprTree. */
int main() {
    ExprTree* T = buildExpr("abc*+de*f+g*+");
    cout << infixExpr(T) << endl;
    cout << "Eval = "
        << evalExpr(T, {{'a',3}, {'b',1}, {'c',0}, {'d',5},
                        {'e',2}, {'f',1}, {'g',6}})
        << endl;
    freeExpr(T); // Not implemented yet
}
```

Exercises

- Design the function `freeExpr`.
- Modify `infixExpr` for a nicer printing:
 - Minimize number of parenthesis.
 - Add spaces around + (but not around *).
- Extend the functions to support other operands, including the unary – (e.g., $-a/b$).

Tree traversals



Traversal: algorithm to visit the nodes of a tree in some specific order.

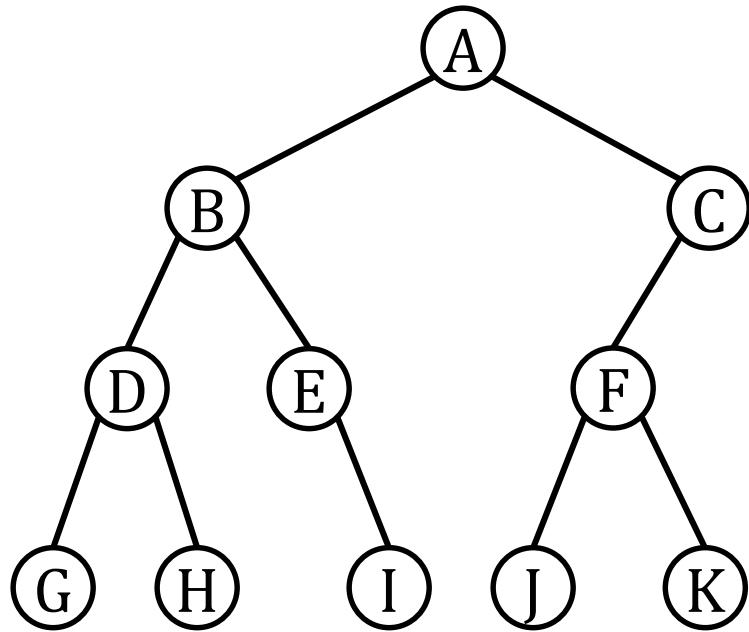
The actions performed when visiting each node can be a parameter of the traversal algorithm.

```
using visitor = void (int &);

// This function matches the type visitor
void print(int& i) {
    cout << i << endl;
}

void traversal(Tree* T, visitor v);
```

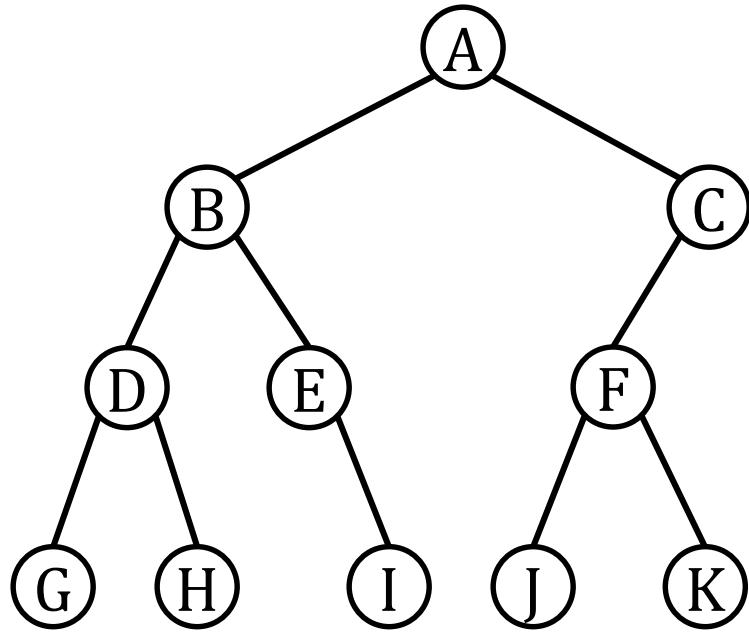
Tree traversals



Preorder: A B D G H E I C F J K

```
void preorder(Tree* T, visitor v) {  
    if (T != nullptr) {  
        v(T->elem);  
        preorder(T->left, v);  
        preorder(T->right, v);  
    }  
}
```

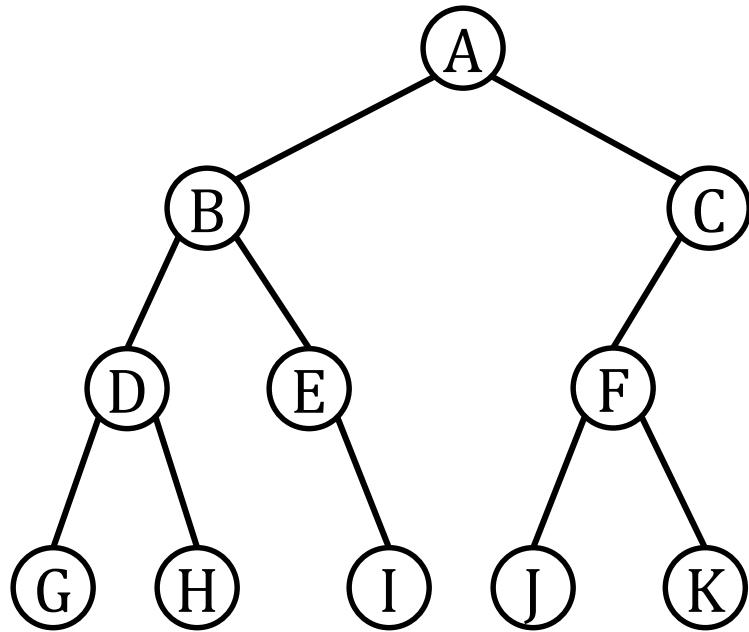
Tree traversals



Preorder: A B D G H E I C F J K
Postorder: G H D I E B J K F C A

```
void postorder(Tree* T, visitor v) {
    if (T != nullptr) {
        postorder(T->left, v);
        postorder(T->right, v);
        v(T->elem);
    }
}
```

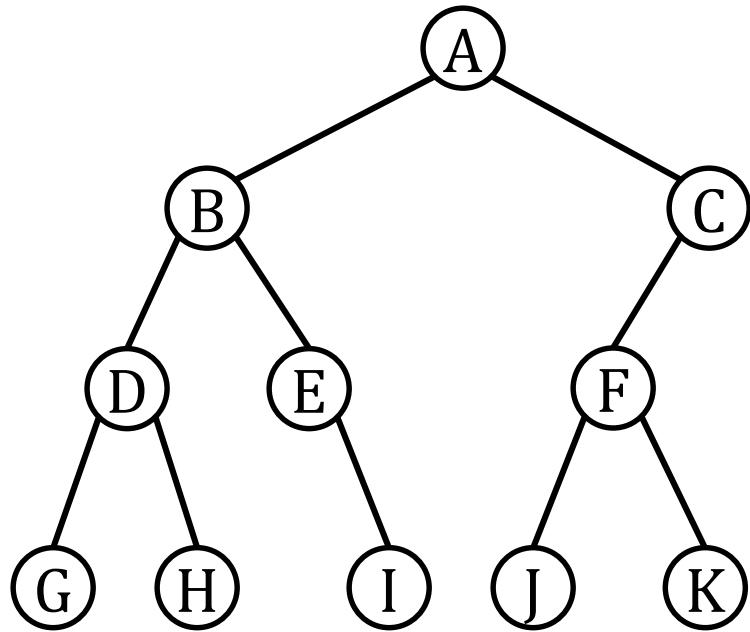
Tree traversals



Preorder: A B D G H E I C F J K
Postorder: G H D I E B J K F C A
Inorder: G D H B E I A J F K C

```
void inorder(Tree* T, visitor v) {
    if (T != nullptr) {
        inorder(T->left, v);
        v(T->elem);
        inorder(T->right, v);
    }
}
```

Tree traversals



Preorder: A B D G H E I C F J K
Postorder: G H D I E B J K F C A
Inorder: G D H B E I A J F K C
By levels: A B C D E F G H I J K

```
void byLevels(Tree* T, visitor v) {
    queue<Tree*> Q; Q.push(T);
    while (not Q.empty()) {
        T = Q.front(); Q.pop();
        if (T != nullptr) {
            v(T->elem);
            Q.push(T->left); Q.push(T->right);
        }
    }
}
```

EXERCISES

Traversals: Full Binary Trees

- A Full Binary Tree is a binary tree where each node has 0 or 2 children.
- Draw the full binary trees corresponding to the following tree traversals:
 - Preorder: 2 7 3 6 1 4 5; Postorder: 3 6 7 4 5 1 2
 - Preorder: 3 1 7 4 9 5 2 6 8; Postorder: 1 9 5 4 6 8 2 7 3
- Given the pre- and post-order traversals of a binary tree (not necessarily full), can we uniquely determine the tree?
 - If yes, prove it.
 - If not, show a counterexample.

Traversals: Binary Trees

- Draw the binary trees corresponding the following traversals:
 - Preorder: 3 6 1 8 5 2 4 7 9; Inorder: 1 6 3 5 2 8 7 4 9
 - Level-order: 4 8 3 1 2 7 5 6 9; Inorder: 1 8 5 2 4 6 7 9 3
 - Postorder: 4 3 2 5 9 6 8 7 1; Inorder: 4 3 9 2 5 1 7 8 6
- Describe an algorithm that builds a binary tree from the preorder and inorder traversals.

Drawing binary trees

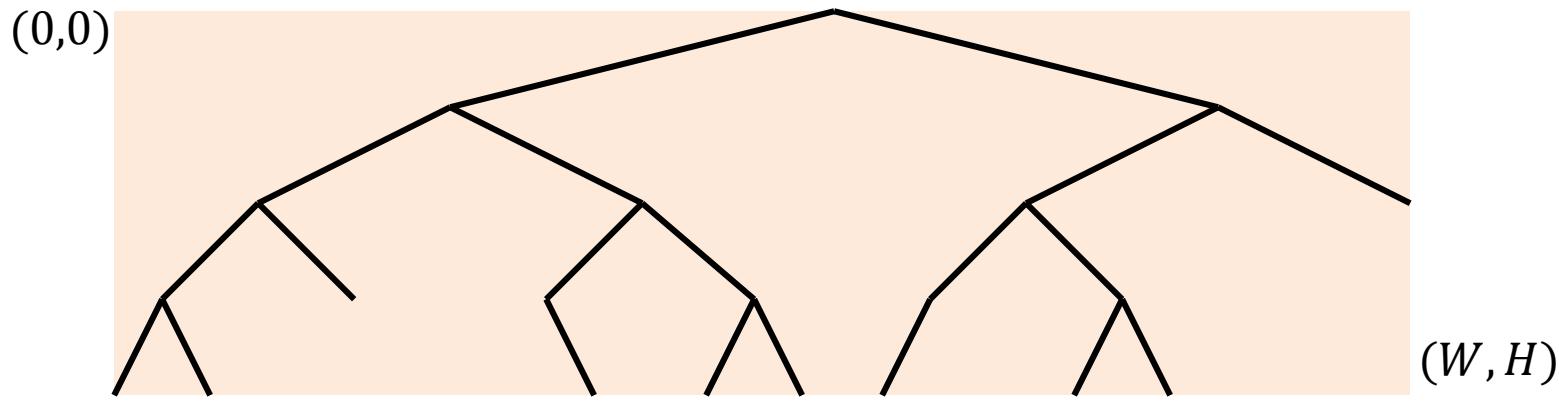
We want to draw the skeleton of a binary tree as it is shown in the figure. For that, we need to assign (x, y) coordinates to each tree node. The layout must fit in a pre-defined bounding box of size $W \times H$, with the origin located in the top-left corner.

Design the function

```
void draw(Tree* T, double W, double H)
```

to assign values to the attributes `x` and `y` of all nodes of the tree in such a way that the lines that connect the nodes do not cross.

Suggestion: calculate the coordinates in two steps. First assign (x, y) coordinates using some arbitrary unit. Next, shift/scale the coordinates to exactly fit in the bounding box.



Containers: Set and Dictionary



Jordi Cortadella and Jordi Petit
Department of Computer Science

Sets and Dictionaries

- A set: a collection of items. The typical operations are:
 - Add/remove one element
 - Does it contain an element?
 - Size?, Is it empty?
 - Visit all items
- A dictionary (map): a collection of key-value pairs. The typical operations are:
 - Put a new key-value pair
 - Remove a key-value pair with a specific key
 - Get the value associated to a key
 - Does it contain a key?
 - Visit all key-value pairs

Sets and Dictionaries

- A dictionary can be treated as a set of keys, each key having an associated value.
- We will focus on the implementation of sets and later extend the implementation to dictionaries.

Phone List

key	value
Alex	x154
Dana	x642
Kim	x911
Les	x120
Sandy	x124

Domain Name Resolution

aclweb.org	128.231.23.4
amazon.com	12.118.92.43
google.com	28.31.23.124
python.org	18.21.3.144
sourceforge.net	51.98.23.53

set

dictionary (map)

Word Frequency Table

computational	25
language	196
linguistics	17
natural	56
processing	57

Source: Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper

Possible implementations of a set

Unsorted list or vector

Insertion	$O(n)$, if checking for duplicate keys, $O(1)$ otherwise.
Deletion	$O(n)$ since it has to find the item along the list.
Lookup	$O(n)$ since the list must be scanned.
Good for	Small sets.

Sorted vector

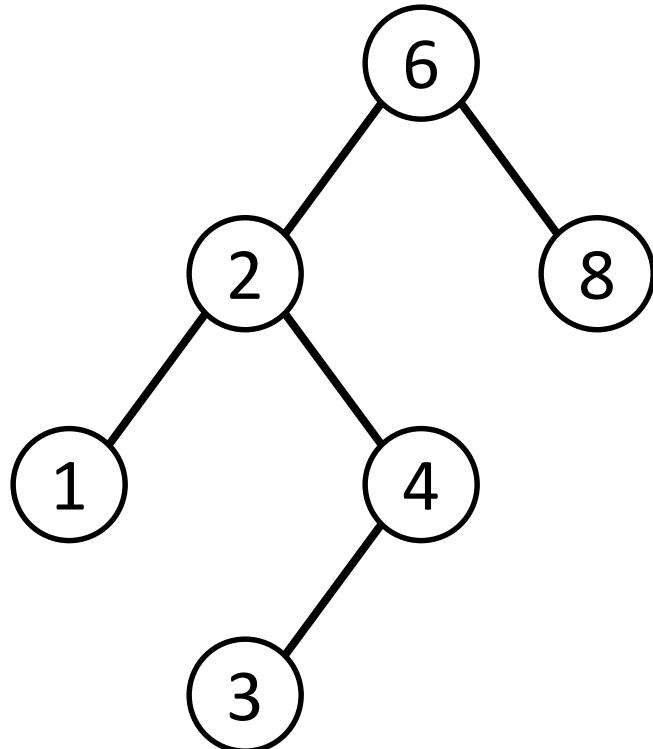
Insertion	$O(n)$ in the worst case (similar to insertion sort)
Deletion	$O(n)$ since it has to sift the elements after deletion.
Lookup	$O(\log n)$ with binary search.
Good for	Read-only collections (only lookups) or very few updates.

Note: n is the number of items in the set.

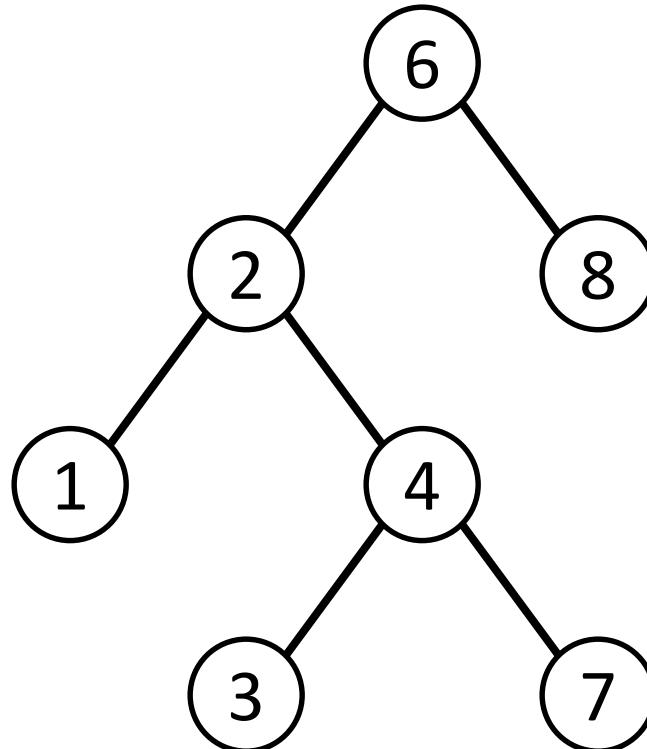
Binary Search Trees

BST property: for every node in the tree with value V:

- All values in the left subtree are smaller than V.
- All values in the right subtree are larger than V.



This is a binary search tree



This is **not** a binary search tree

BST: public methods

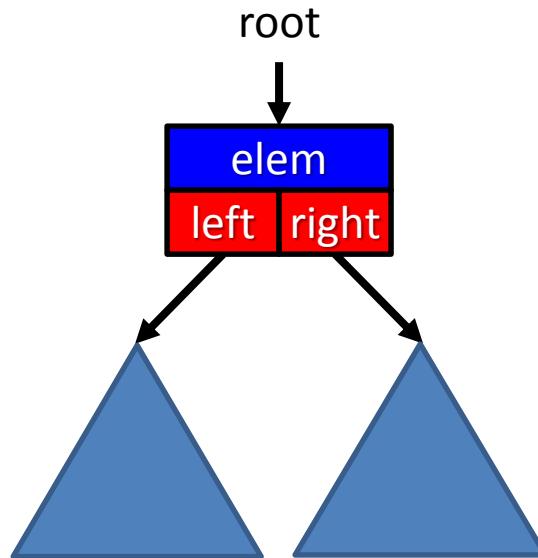
```
template<typename T>
class Set {
public:
    // Constructors, assignment and destructor
    Set();
    Set(const Set& S);
    Set& operator=(const Set& S);
    ~Set();

    // Finding elements
    const T& findMin() const;
    const T& findMax() const;
    bool contains(const T& x) const;
    int size() const;
    bool isEmpty() const;

    // Insert/remove methods
    void insert(const T& x);
    void remove(const T& x);
```

BST: internal implementation

```
private:  
    struct Node {  
        T elem;           // The element stored in the node  
        Node* left;       // Pointer to the left subtree  
        Node* right;      // Pointer to the right subtree  
    };  
  
    Node* root;         // Pointer to the root of the tree  
    int n;              // Number of elements
```



BST: private methods

private:

```
// Public methods require a private pointer-based  
// version to traverse the tree.
```

```
// Finding elements
```

```
Node* findMin(Node* t) const;
```

```
Node* findMax(Node* t) const;
```

```
bool contains(const T& x, Node* t) const;
```

```
// Insert/remove methods
```

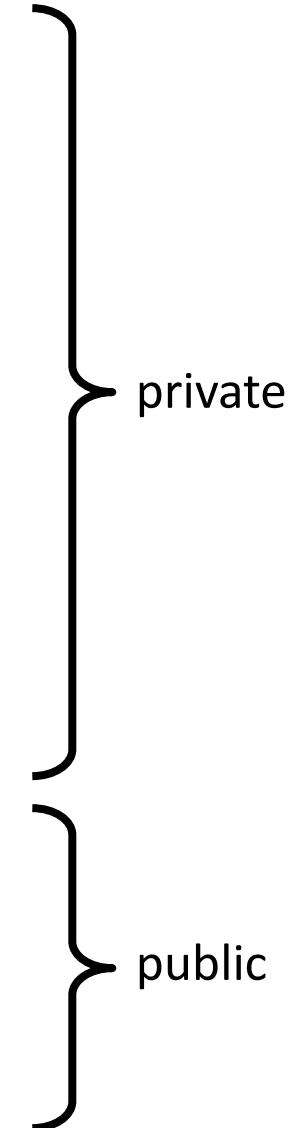
```
void insert(const T& x, Node*& t);
```

```
void remove(const T& x, Node*& t);
```

```
void makeEmpty(Node*& t);
```

findMin (recursive) and findMax (iterative)

```
/** Find the smallest item in the (non-empty) subtree t.  
 * Returns a ptr to the node with the smallest item. */  
Node* findMin(Node* t) const {  
    if (t->left == nullptr) return t;  
    return findMin(t->left);  
}  
  
/** Find the largest item in the (non-empty) subtree t.  
 * Returns a ptr to the node with the largest item. */  
Node* findMax(Node* t) const {  
    while (t->right != nullptr) t = t->right;  
    return t;  
}  
  
/** Find the smallest item in the (non-empty) Set. */  
const T& findMin() const {  
    assert(not isEmpty());  
    return findMin(root)->elem;  
}  
  
// findMax has a similar implementation
```



private

public

contains and isEmpty

```
/** Find an item in the subtree represented by t.  
 * Returns true if found, and false otherwise. */  
bool contains(const T& x, Node* t) const {  
    if (t == nullptr) return false;  
    if (x < t->elem) return contains(x, t->left);  
    if (x > t->elem) return contains(x, t->right);  
    return true;  
}
```

private

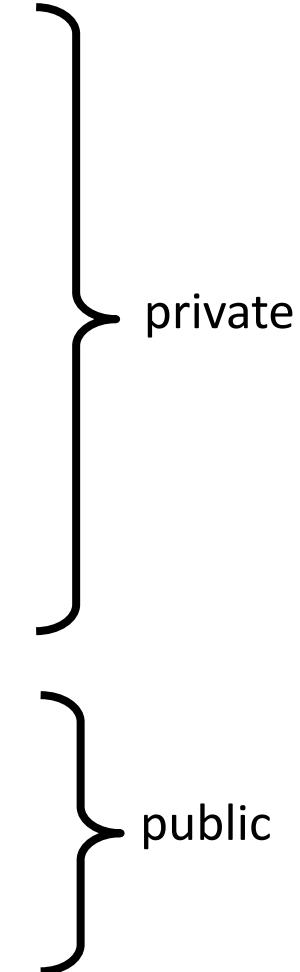
```
/** Find an item in the set.  
 * Returns true if found, and false otherwise. */  
bool contains(const T& x) const {  
    return contains(x, root);  
}
```

public

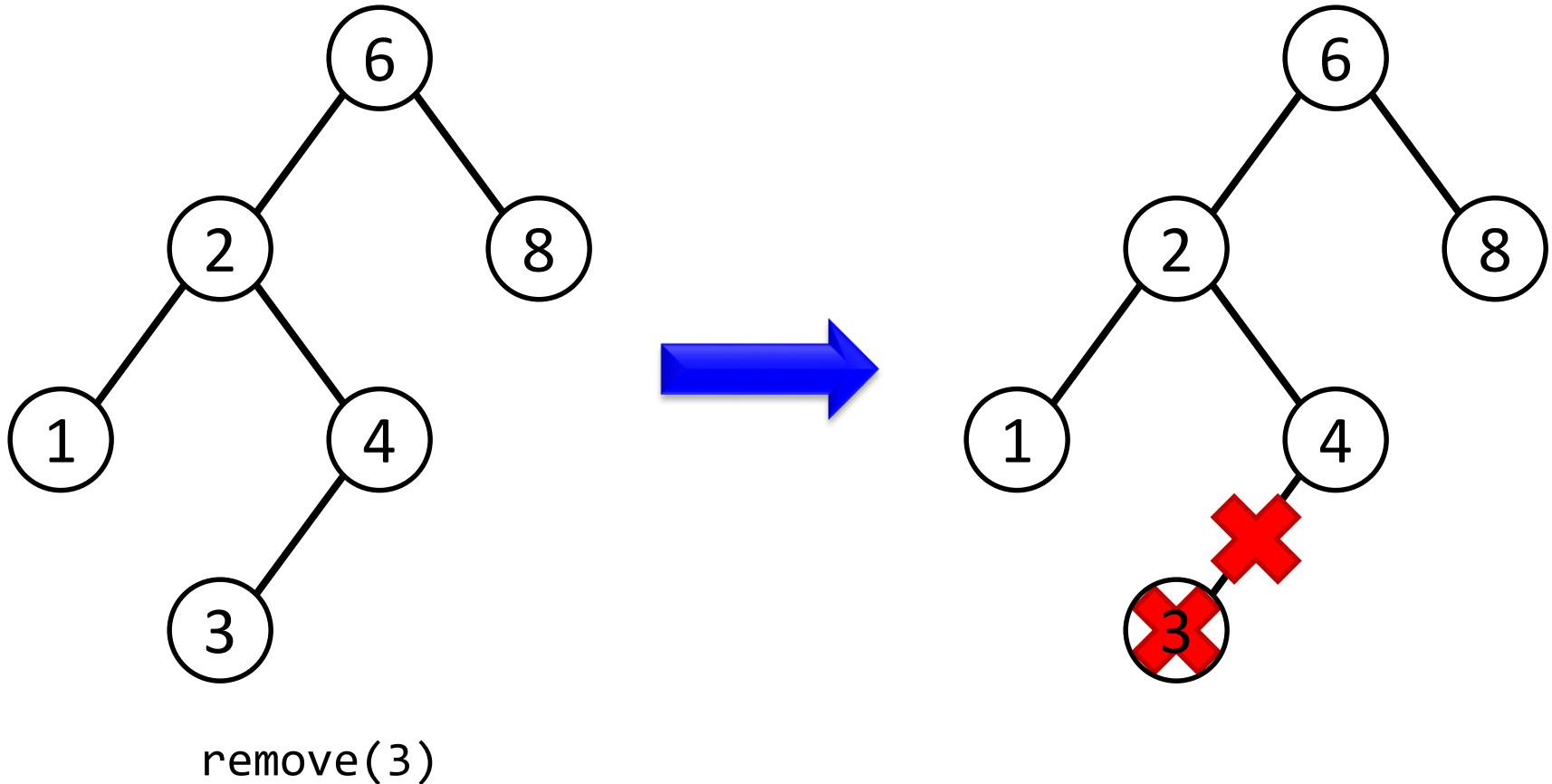
```
/** Checks whether the tree is empty. */  
bool isEmpty() const {  
    return size() == 0; // or also root == nullptr  
}
```

insert

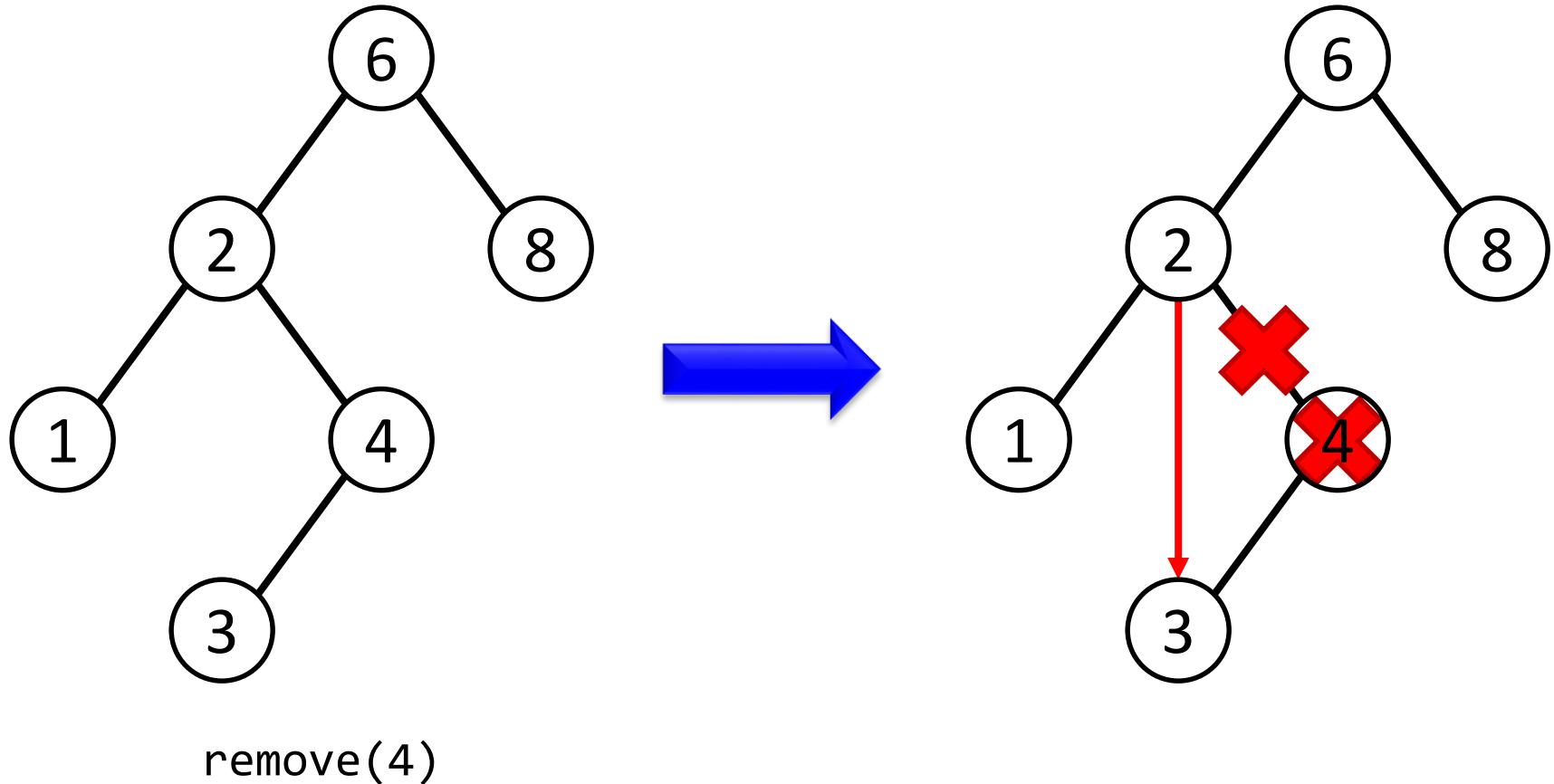
```
/** Inserts item x into the subtree t.  
 * It may modify the value of t. */  
void insert(const T& x, Node*& t) {  
    if (t == nullptr) {  
        t = new Node {x, nullptr, nullptr};  
        ++n;  
    }  
    else if (x < t->elem) insert(x, t->left);  
    else if (x > t->elem) insert(x, t->right);  
    // else: duplicated item, do nothing  
}  
  
/** Inserts item x into the set. */  
void insert(const T& x) {  
    insert(x, root);  
}
```



remove: simple case (no children)



remove: simple case (one child)

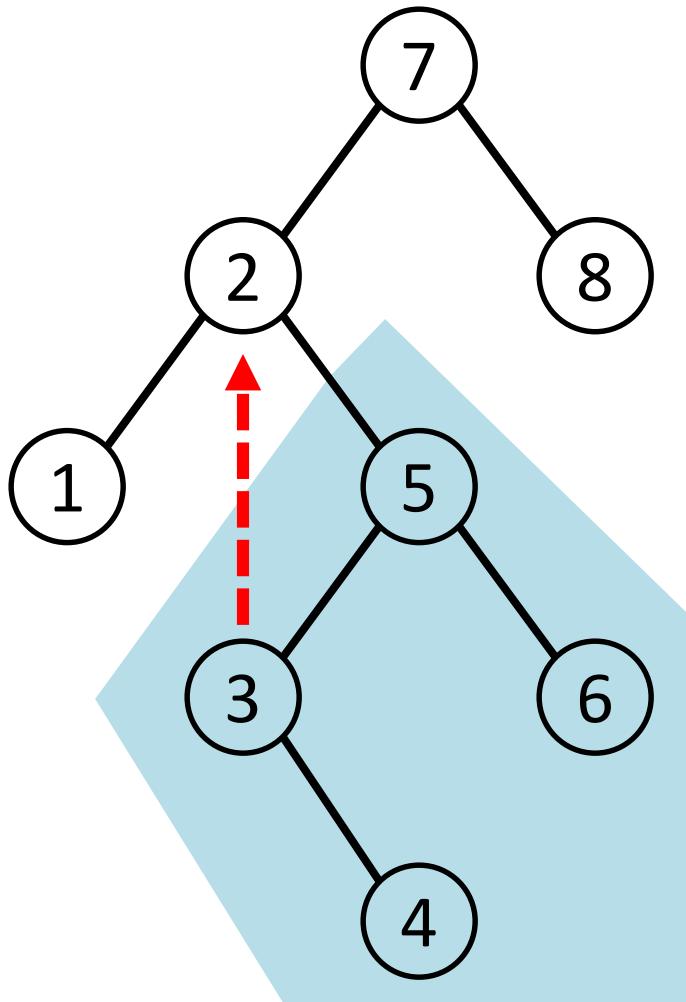


remove: simple cases

```
/** Removes item x from the subtree t. */
void remove(const T& x, Node*& t) {
    if (t == nullptr) return; // Not found
    if (x < t->elem) return remove(x, t->left);
    if (x > t->elem) return remove(x, t->right);

    // We have found the item
    if (t->left == nullptr or t->right == nullptr) {
        Node* old = t;
        t = t->left ? t->left : t->right;
        delete old;
        --n;
    } else {
        ...
        ... // Case with two children
        ...
    }
}
```

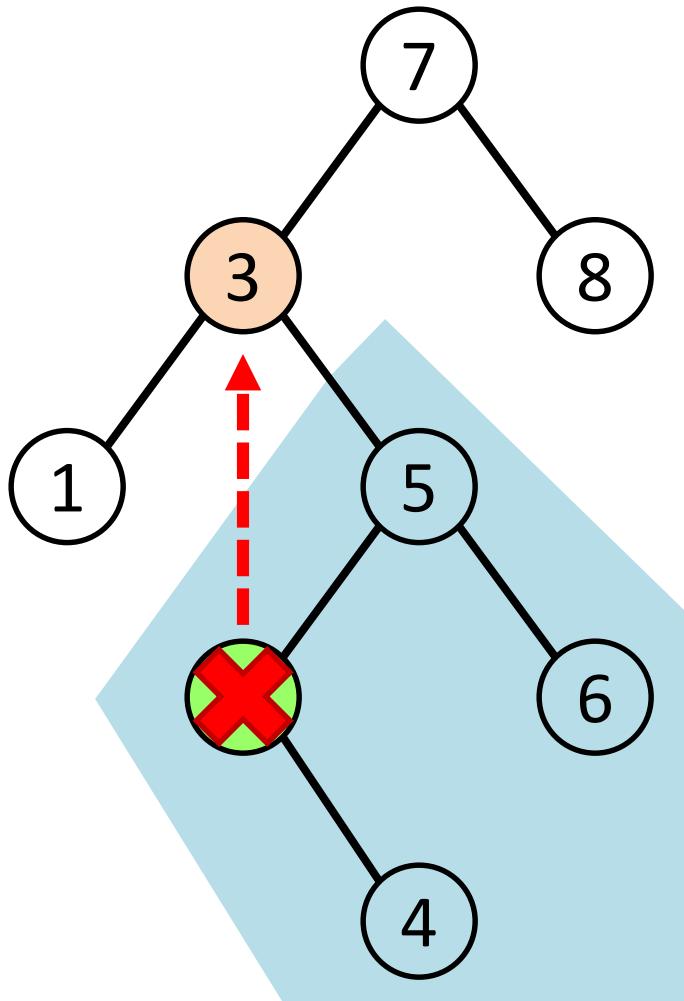
remove: complex case (two children)



remove(2)

1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.

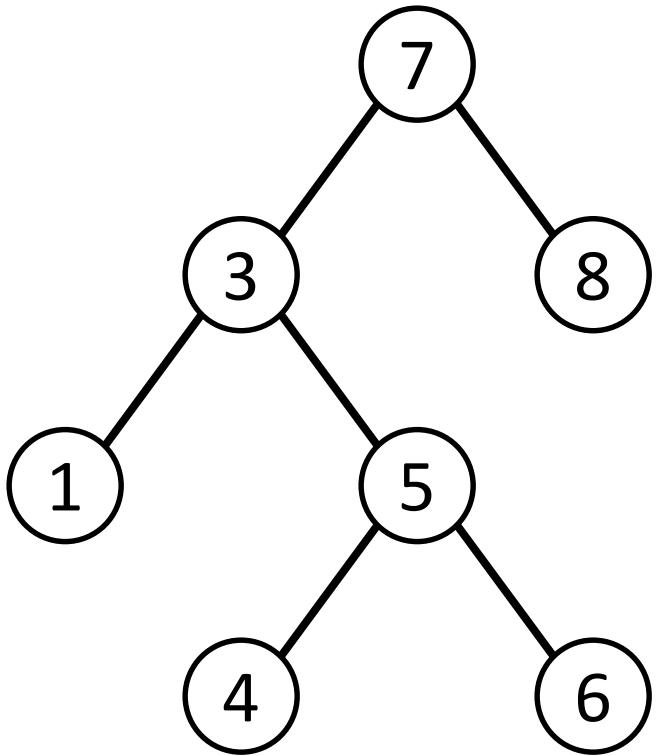
remove: complex case (two children)



remove(2)

1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.
4. Remove the min value in the right subtree (simple case).

remove: complex case (two children)



1. Find the element.
2. Find the min value of the right subtree.
3. Copy the min value onto the element to be removed.
4. Remove the min value in the right subtree (simple case).

remove(2)

remove: all cases

```
/** Removes item x from the subtree t. */
void remove(const T& x, Node*& t) {
    if (t == nullptr) return; // Not found
    if (x < t->elem) return remove(x, t->left);
    if (x > t->elem) return remove(x, t->right);

    // We have found the item
    if (t->left == nullptr or t->right == nullptr) {
        Node* old = t;
        t = t->left ? t->left : t->right;
        delete old;
        --n;
    } else { // Case with two children (simple version with copy)
        // A version manipulating only pointers is also possible
        t->elem = findMin(t->right)->elem; // Copy the min element
        remove(t->elem, t->right);           // Remove the min elem
    }
}

/** Public method for remove. */
void remove(const T& x) {
    remove(x, root);
}
```

Constructors and destructor

```
/** Default constructor (empty set). */
Set() : root(nullptr), n(0) {}

/** Copy constructor. */
Set(const Set& S) {
    root = copy(S.root);
    n = S.n;
}

/** Assignment operator. */
Set& operator=(const Set& S) {
    if (&S != this) {
        makeEmpty(root);
        root = copy(S.root);
        n = S.n;
    }
    return *this;
}

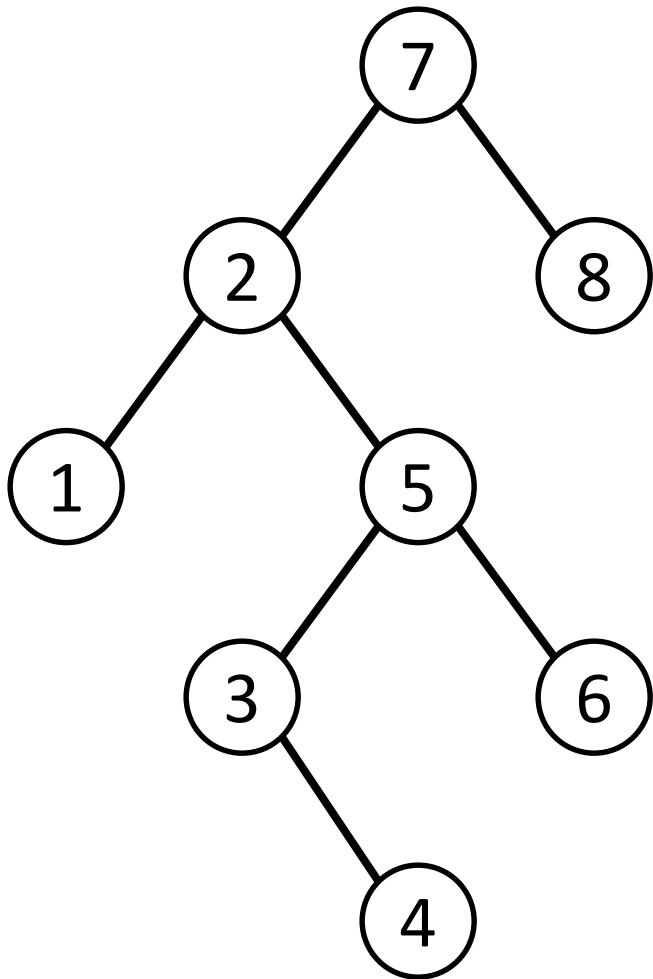
/** Destructor. */
~Set() {
    makeEmpty(root);
}
```

copy and free (private methods)

```
/** Recursive method to clone a subtree. */
Node* copy(Node* t) const {
    if (t == nullptr) return nullptr;
    return new Node{t->elem, copy(t->left), copy(t->right)};
}
```

```
/** Recursive method to clean a subtree. */
void makeEmpty(Node*& t) {
    if (t != nullptr) {
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }
    t = nullptr;
}
```

Visiting the items in ascending order



Question:

How can we visit the items of a BST in ascending order?

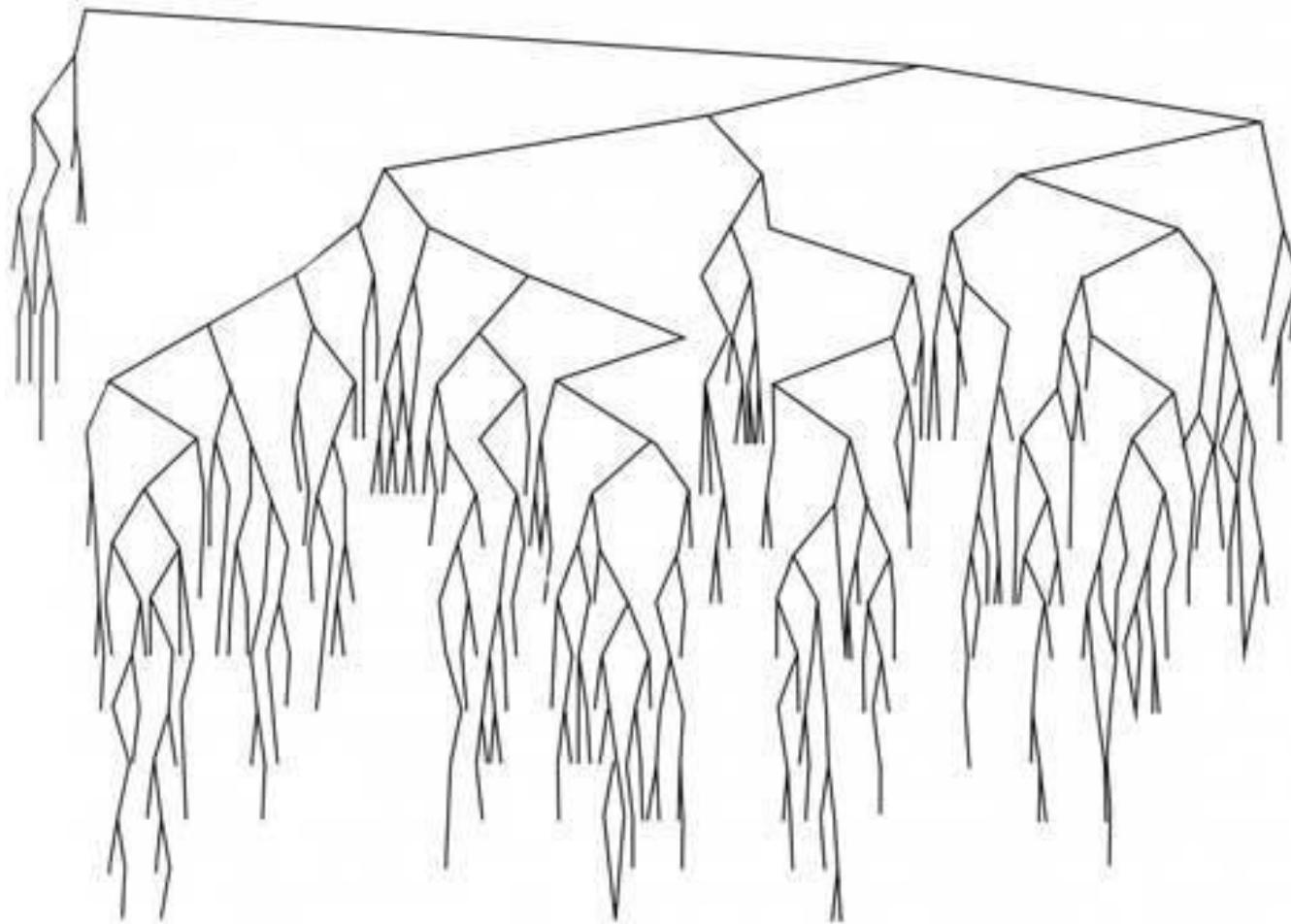
Answer:

Using an in-order traversal

BST: runtime analysis

- Let us assume that the set has n elements. The operations `copy` and `makeEmpty` take $O(n)$.
- We are mostly interested in the runtime of the `insert`/`remove`/`contains` methods.
 - The complexity is $O(d)$, where d is the depth of the node containing the required element.
- But, how large is d ?

Random BST



Source: Fig 4.29 of Weiss textbook

BST: runtime analysis

- Internal path length (IPL): The sum of the depths of all nodes in a tree. Let us calculate the average IPL considering all possible insertion sequences.
- $D(n)$ is the IPL of a tree with n nodes. $D(1) = 0$. The left subtree has i nodes and the right subtree has $n - i - 1$ nodes. Thus,

$$D(n) = D(i) + D(n - i - 1) + (n - 1)$$

- If all subtree sizes are equally likely, then the average value for $D(i)$ and $D(n - i - 1)$ is

$$\frac{1}{n} \sum_{j=0}^{n-1} D(j)$$

BST: runtime analysis

- Therefore,

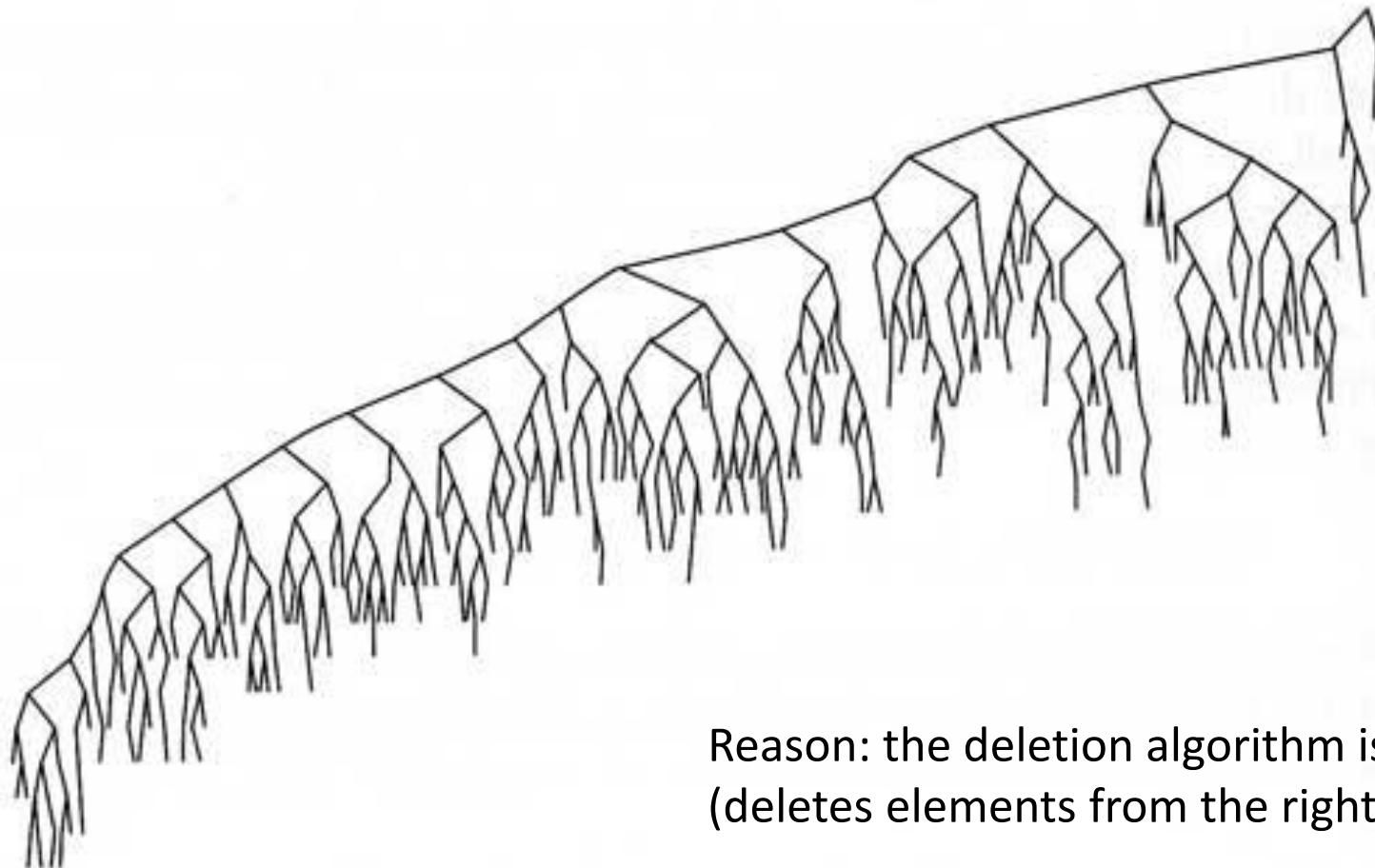
$$D(n) = \frac{2}{n} \left[\sum_{j=0}^{n-1} D(j) \right] + n - 1$$

- The previous recurrence gives:

$$D(n) = O(n \log n)$$

- The average height of nodes after n random insertions is $O(\log n)$.
- However, the $O(\log n)$ average height is not preserved when doing deletions.

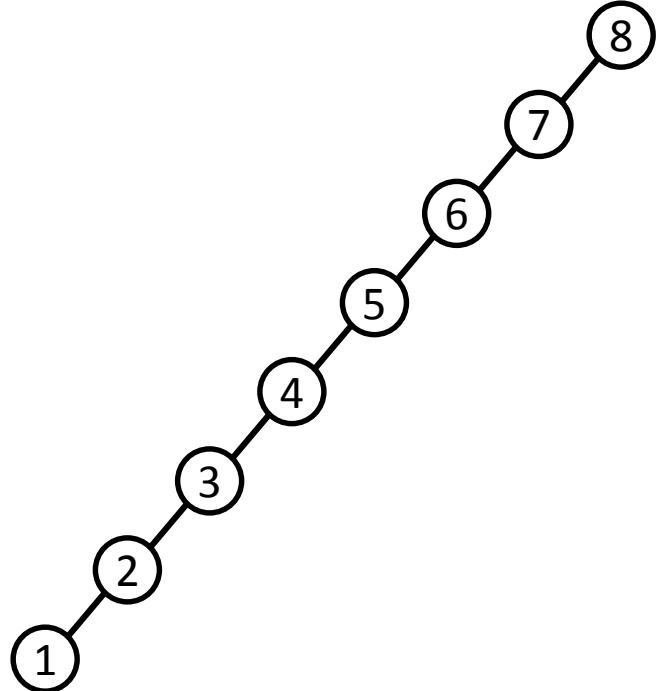
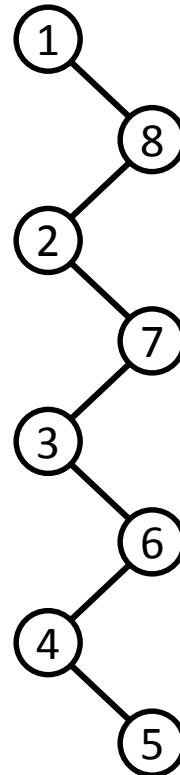
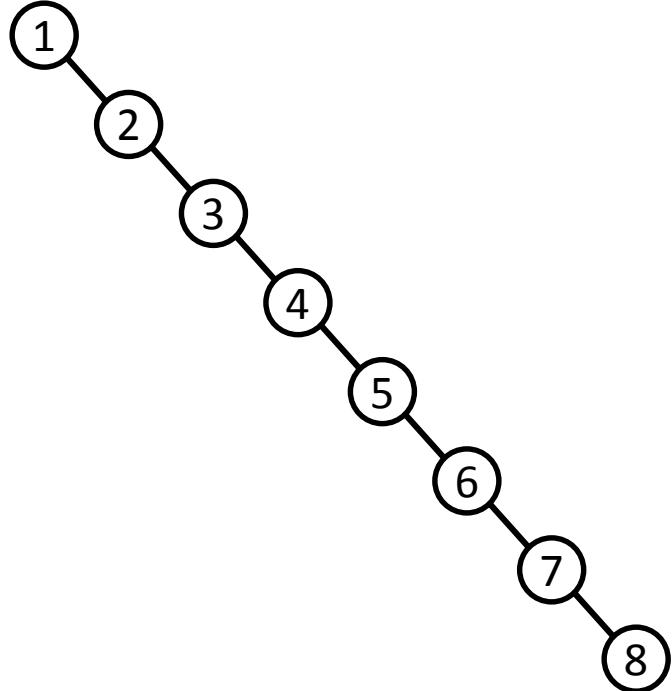
Random BST after n^2 insert/removes



Reason: the deletion algorithm is asymmetric
(deletes elements from the right subtree)

Source: Fig 4.30 of Weiss textbook

Worst-case runtime: $O(n)$



Balanced trees

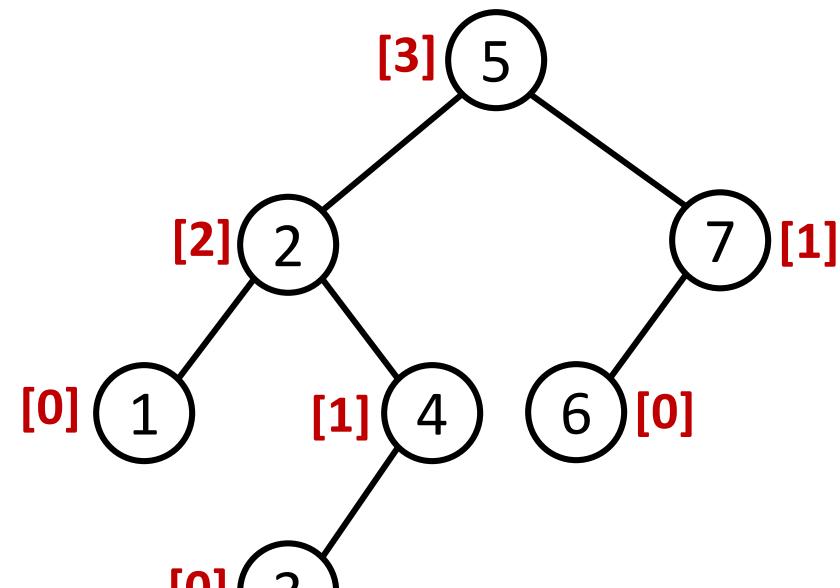
- The worst-case complexity for insert, remove and search operations in a BST is $O(n)$, where n is the number of elements.
- Various representations have been proposed to keep the height of the tree as $O(\log n)$:
 - AVL trees
 - Red-Black trees
 - Splay trees
 - B-trees

AVL trees

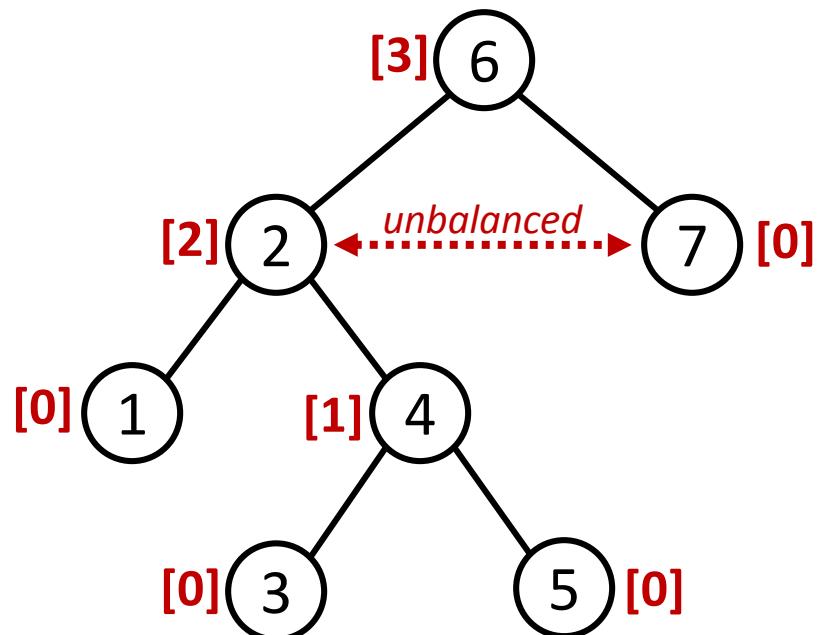
- Named after Adelson-Velsky and Landis (1962).
- Main idea: invest some additional time to balance the tree each time a new element is inserted or deleted.
- Properties:
 - The height of the tree is always $\Theta(\log n)$.
 - The time devoted to balancing is $O(\log n)$.

AVL tree: definition

- An AVL tree is a BST such that, for every node, the difference between the heights of the left and right subtrees is at most 1.



AVL

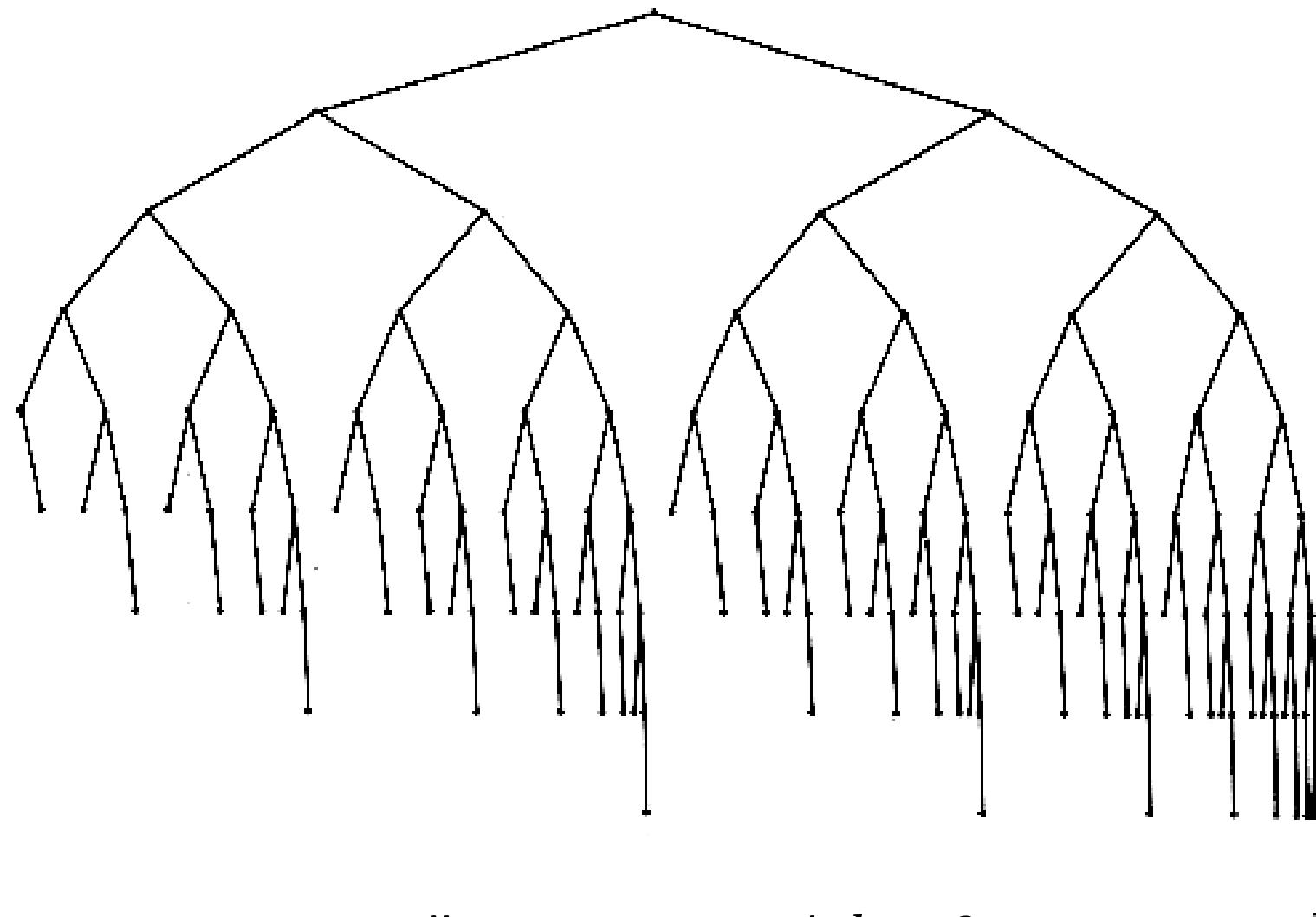


not AVL

AVL tree in action

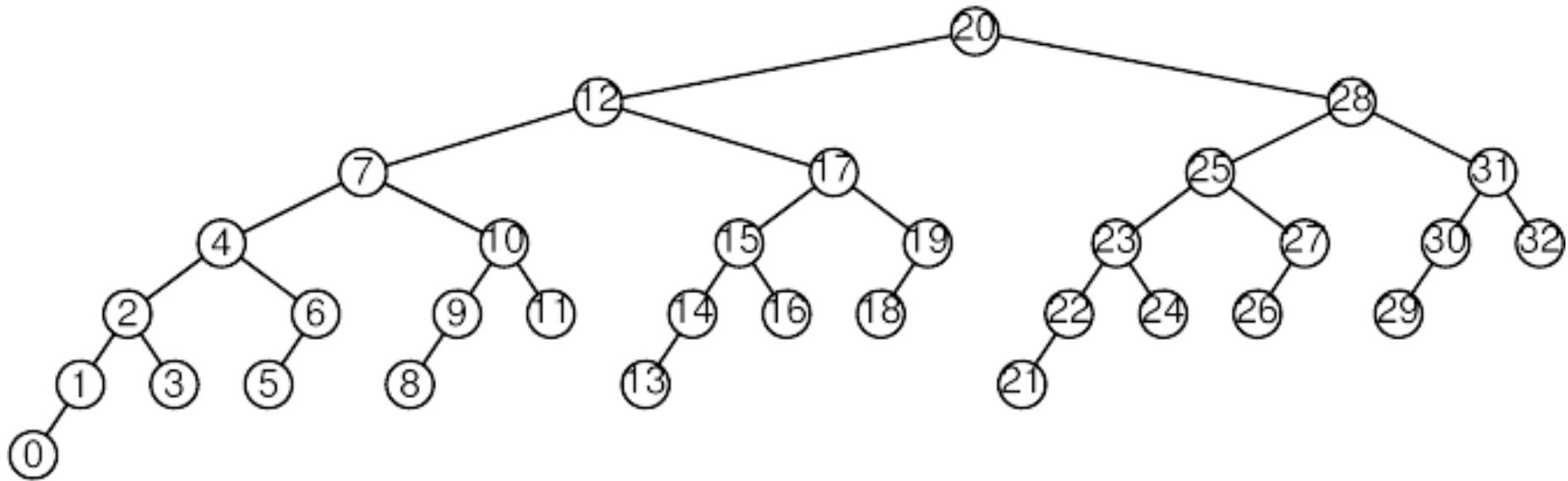
https://en.wikipedia.org/wiki/AVL_tree

AVL trees



Smallest AVL tree with $h = 9$.

AVL trees



Smallest AVL tree with $h = 6$.

The important question: what is the size of an AVL tree with height h ?

Height of an AVL tree

- Theorem: The height of an AVL tree with n nodes is $\Theta(\log n)$.
- Proof in two steps:
 - The height is $\Omega(\log n)$.
 - The height is $O(\log n)$.

The height is $\Omega(\log n)$

- The size n of a tree with height h is:

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

(all levels full of nodes)

- Therefore,

$$\log_2(n + 1) - 1 \leq h$$

and $h = \Omega(\log n)$.

The height is $O(\log n)$

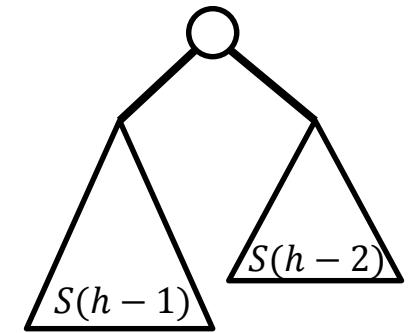
- Let $S(h)$ be the min number of nodes of an AVL tree with height h .
- One of the children (e.g., left) must have height $h - 1$. The other child must have height $h - 2$ (because the AVL has min size).

- Therefore,

$$S(h) = S(h - 1) + S(h - 2) + 1.$$

- Thus,

$$S(h) \geq 2 \cdot S(h - 2).$$



- Given that $S(0) = 1$ and $S(1) = 2$, it can be easily proven, by induction, that:

$$S(h) \geq 2^{h/2}$$

- Since $n \geq S(h)$ and $\log_2 S(h) \geq h/2$, then $h \leq 2 \log_2 n$:

$$h = O(\log n).$$

Height of an AVL tree

- The recurrence

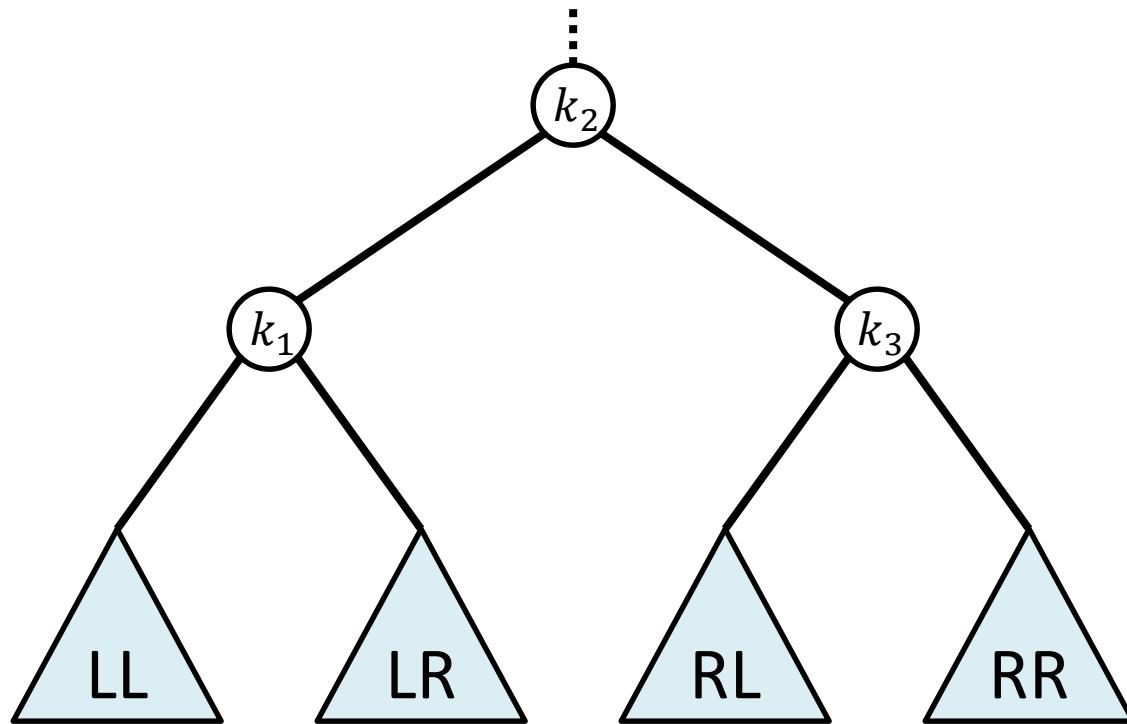
$$S(h) = S(h - 1) + S(h - 2) + 1$$

resembles the one of the Fibonacci numbers.
A tighter bound can be obtained.

- Theorem: the height of an AVL tree with n internal nodes satisfies:

$$h < 1.44 \log_2(n + 2) - 1.328$$

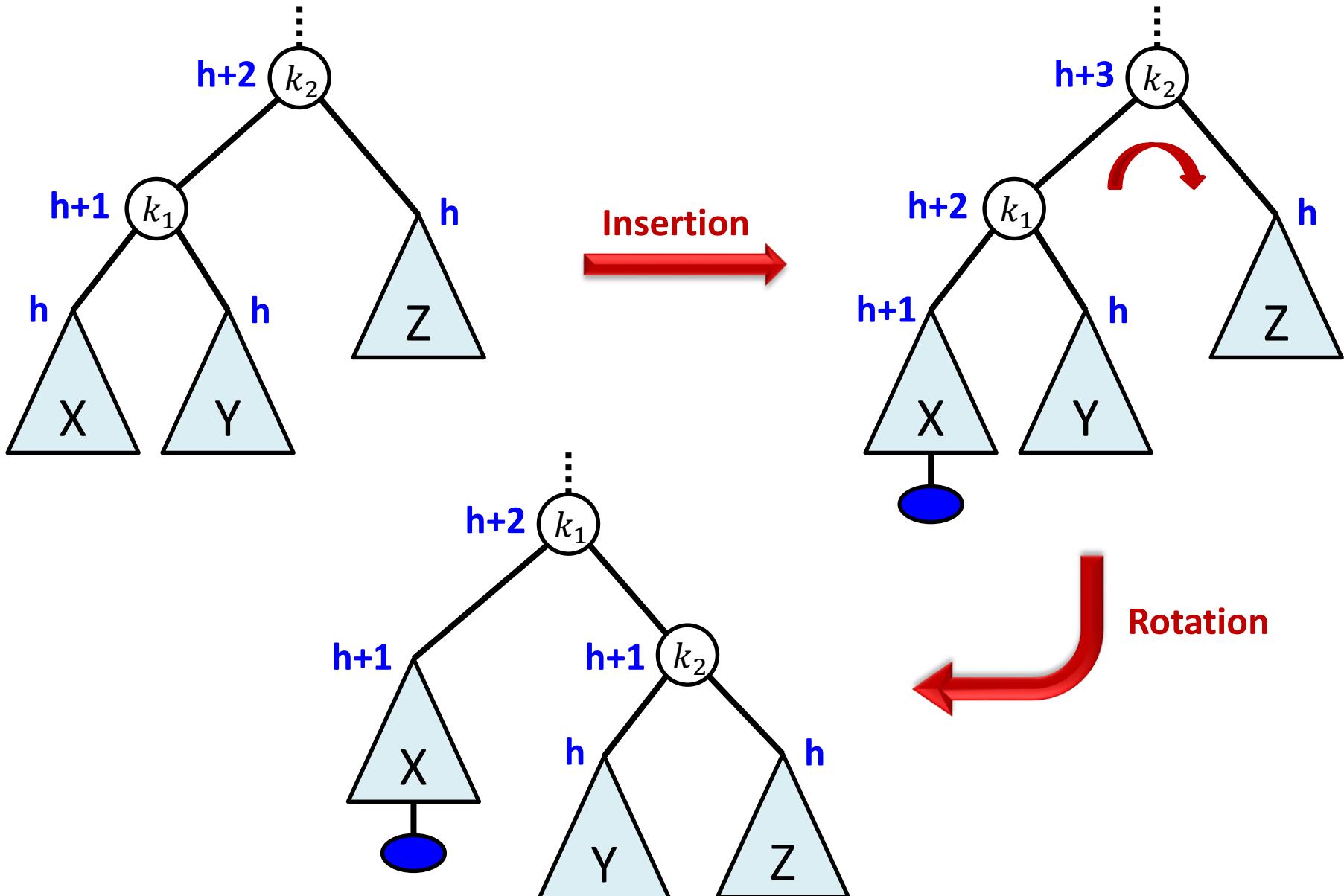
Unbalanced insertion: 4 cases



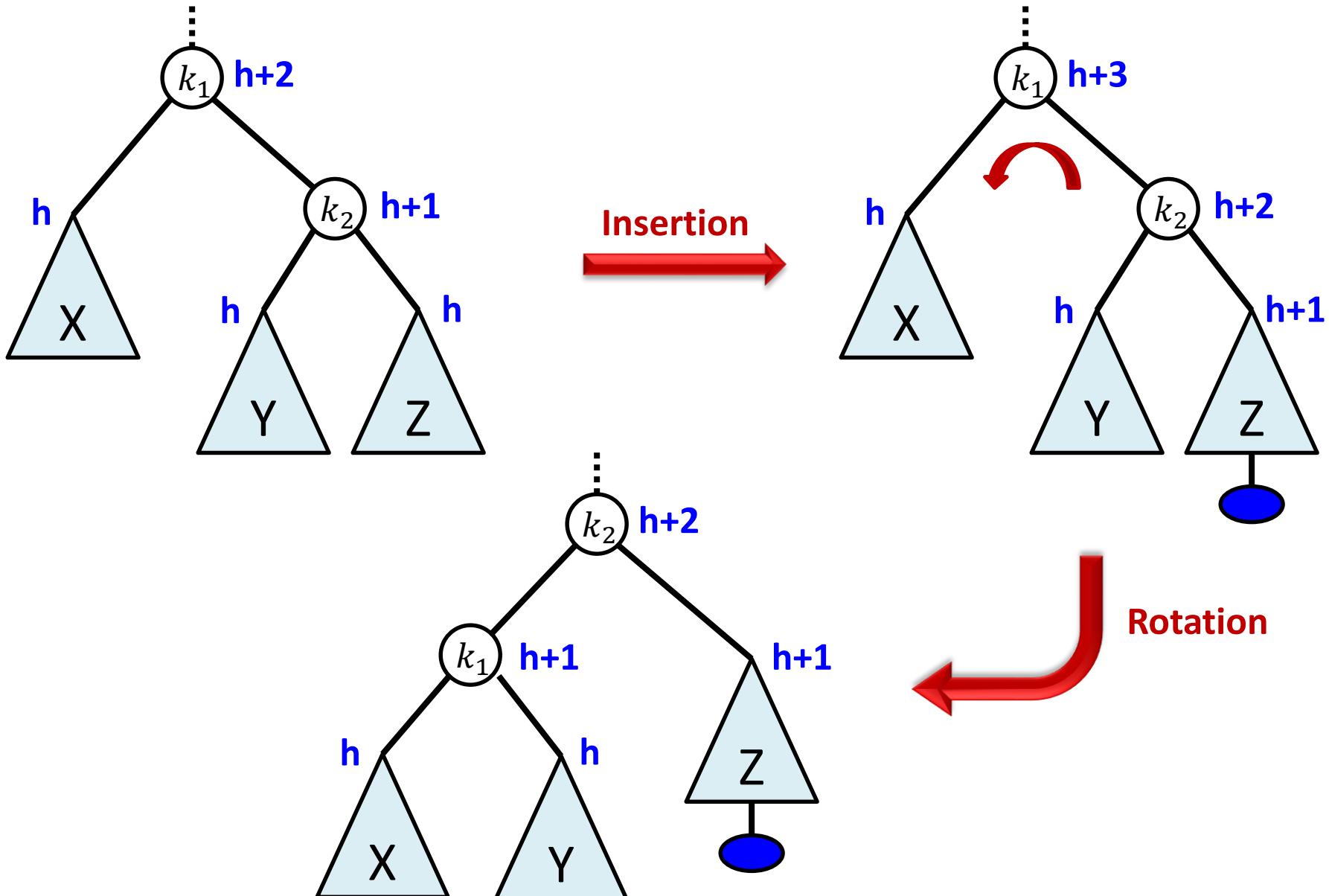
Any newly inserted item may fall into any of the four subtrees (LL, LR, RL or RR).

A new insertion may violate the balancing property. Re-balancing might be required.

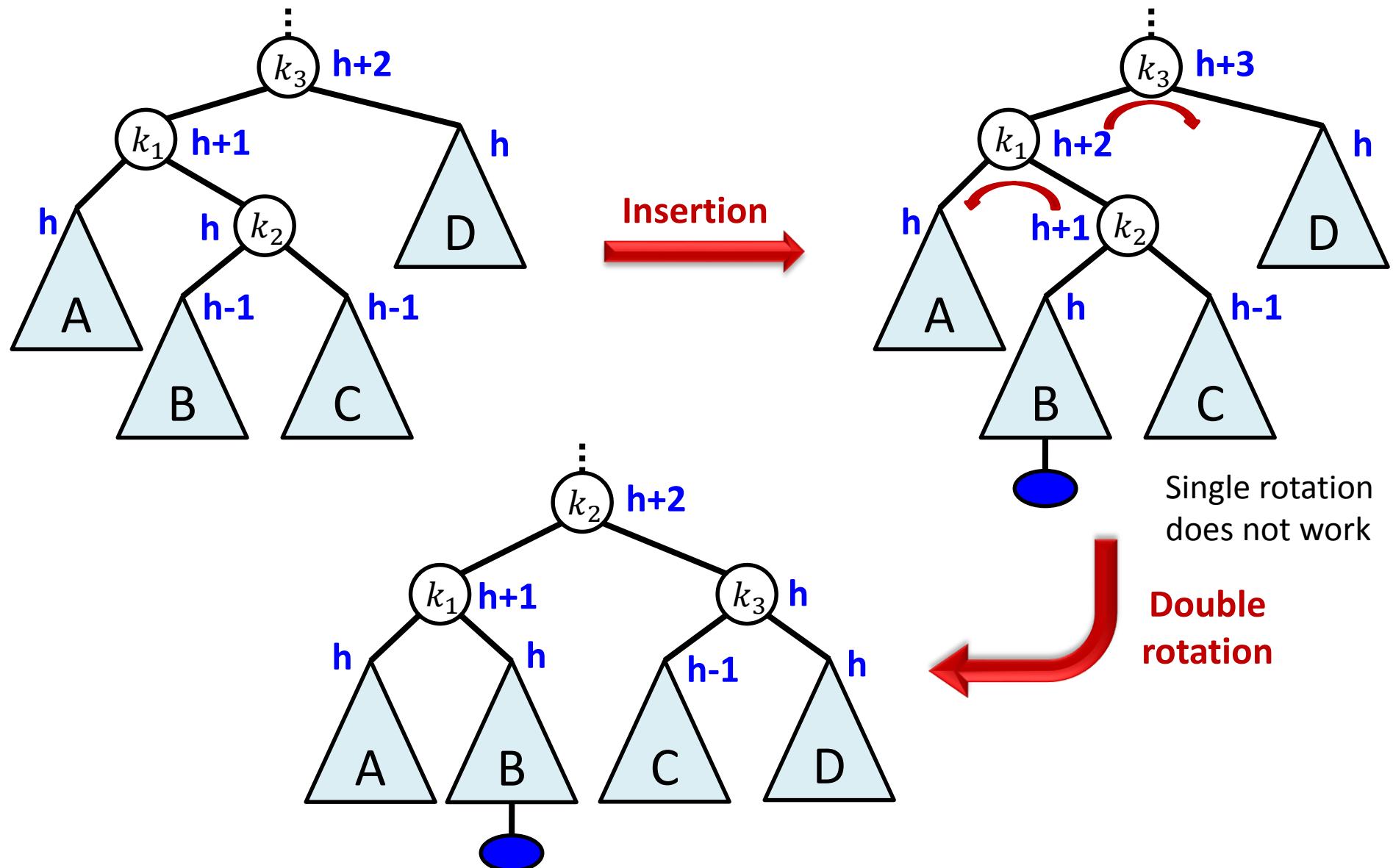
Single rotation: the left-left case



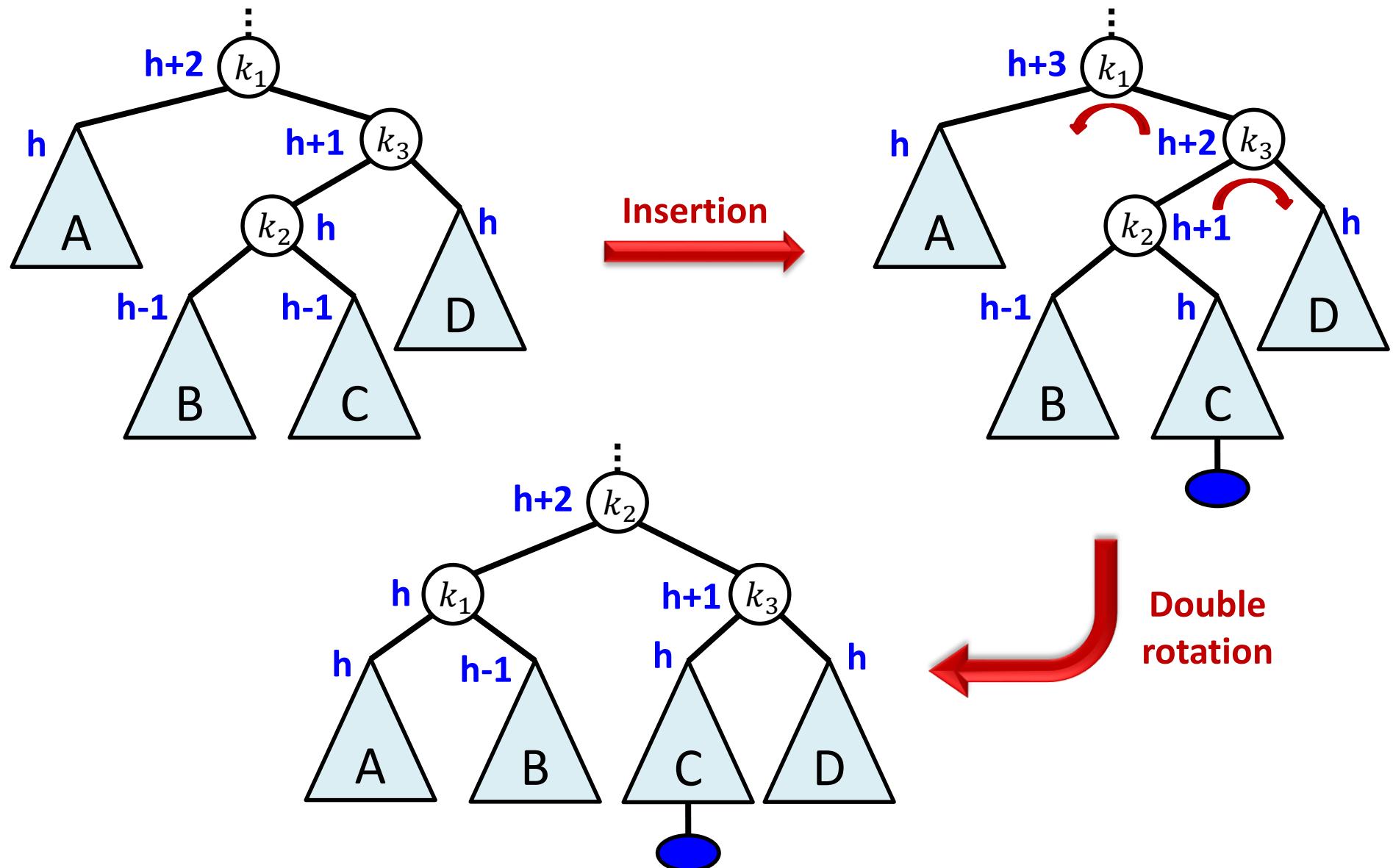
Single rotation: the right-right case



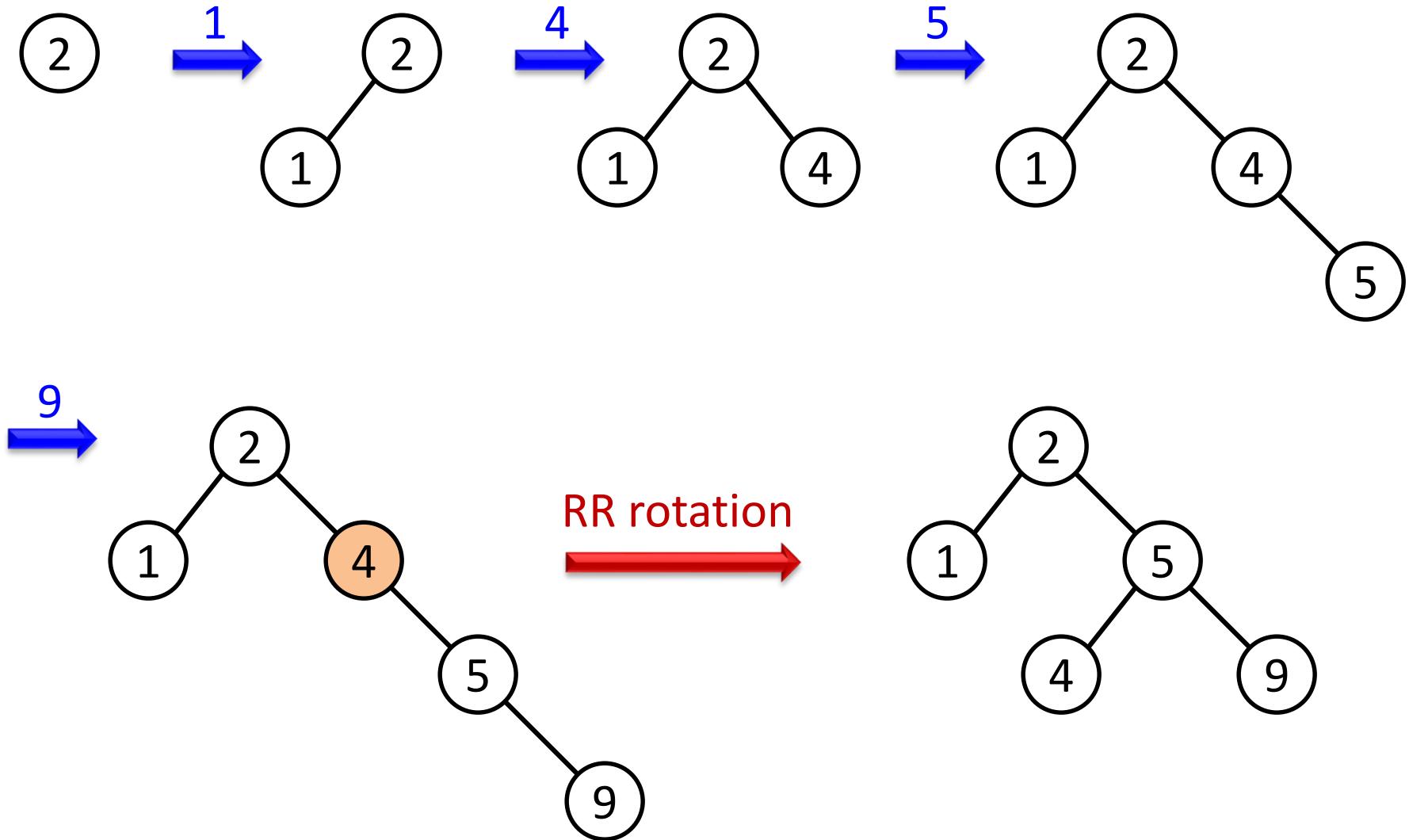
Double rotation: the left-right case



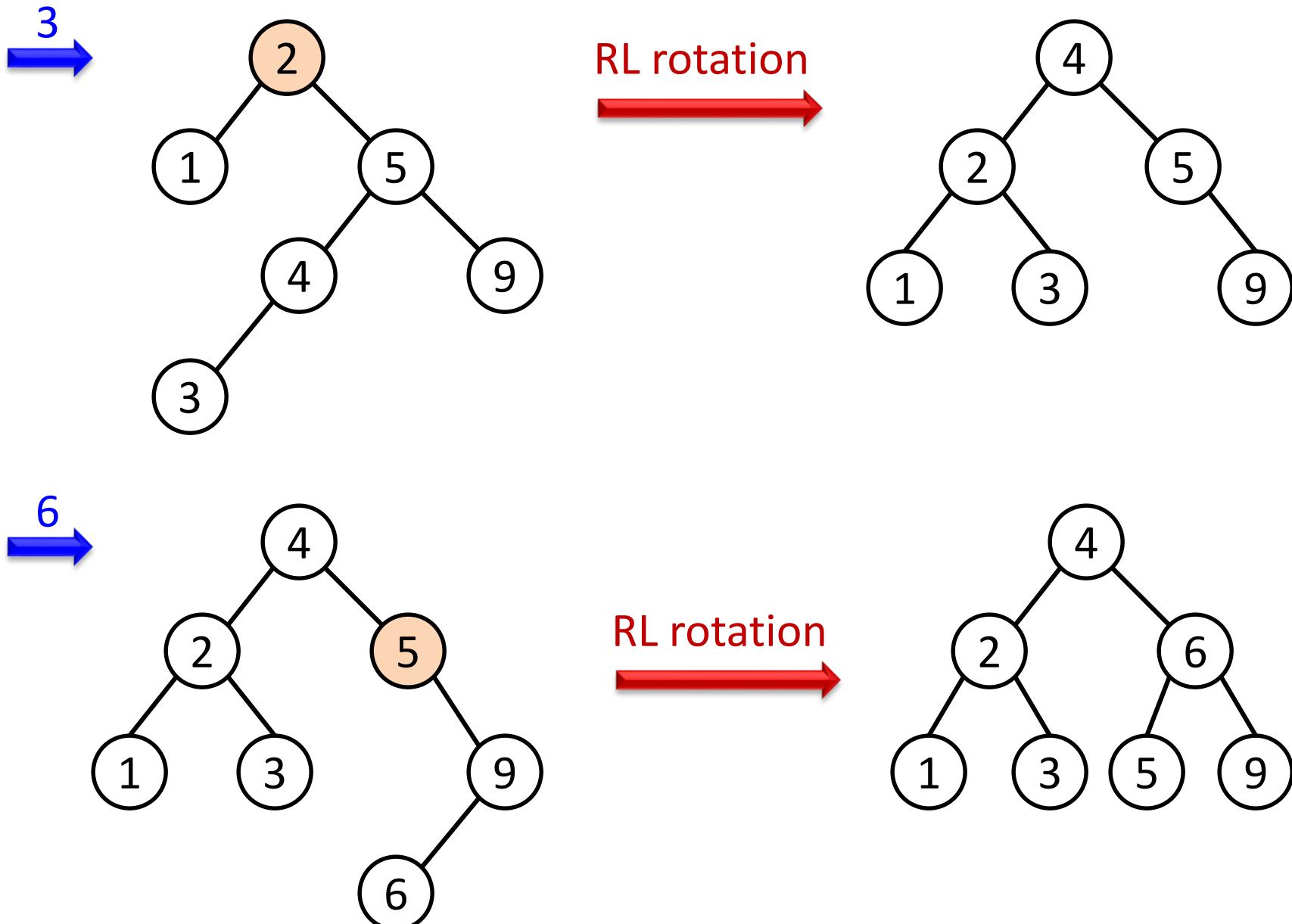
Double rotation: the right-left case



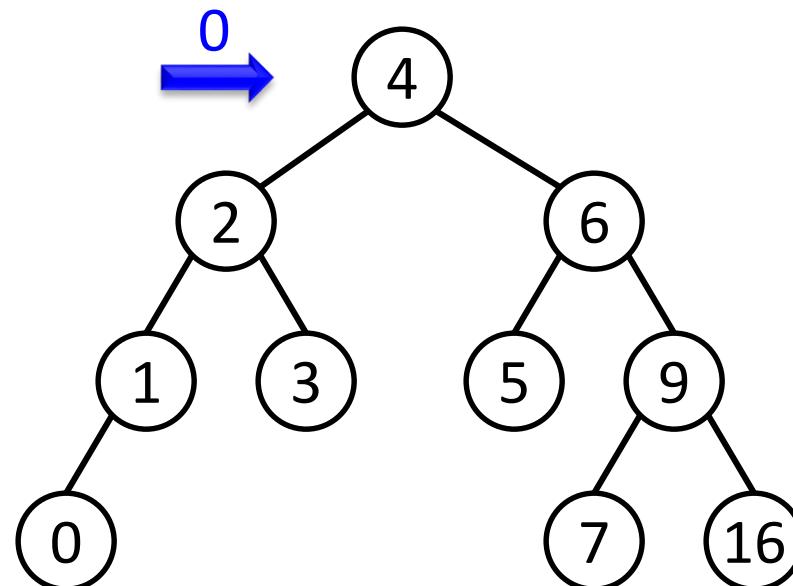
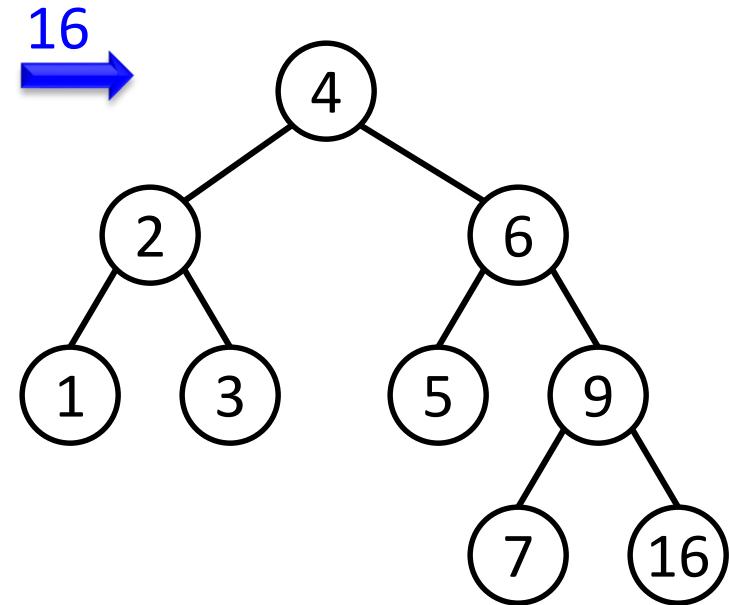
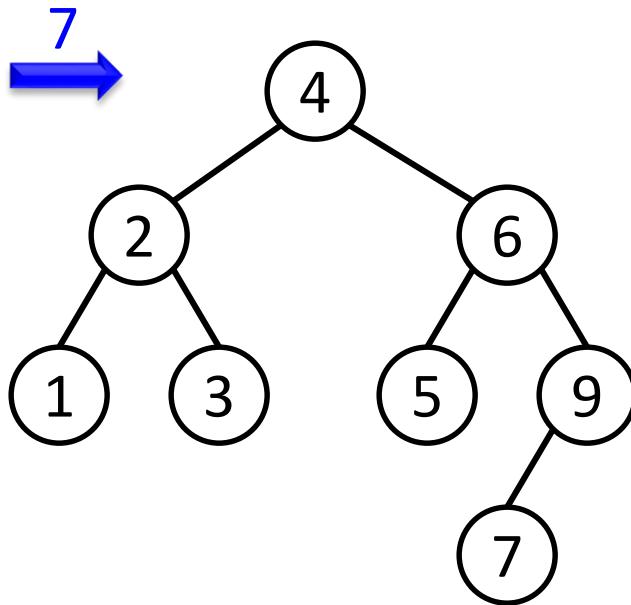
Example: insertions



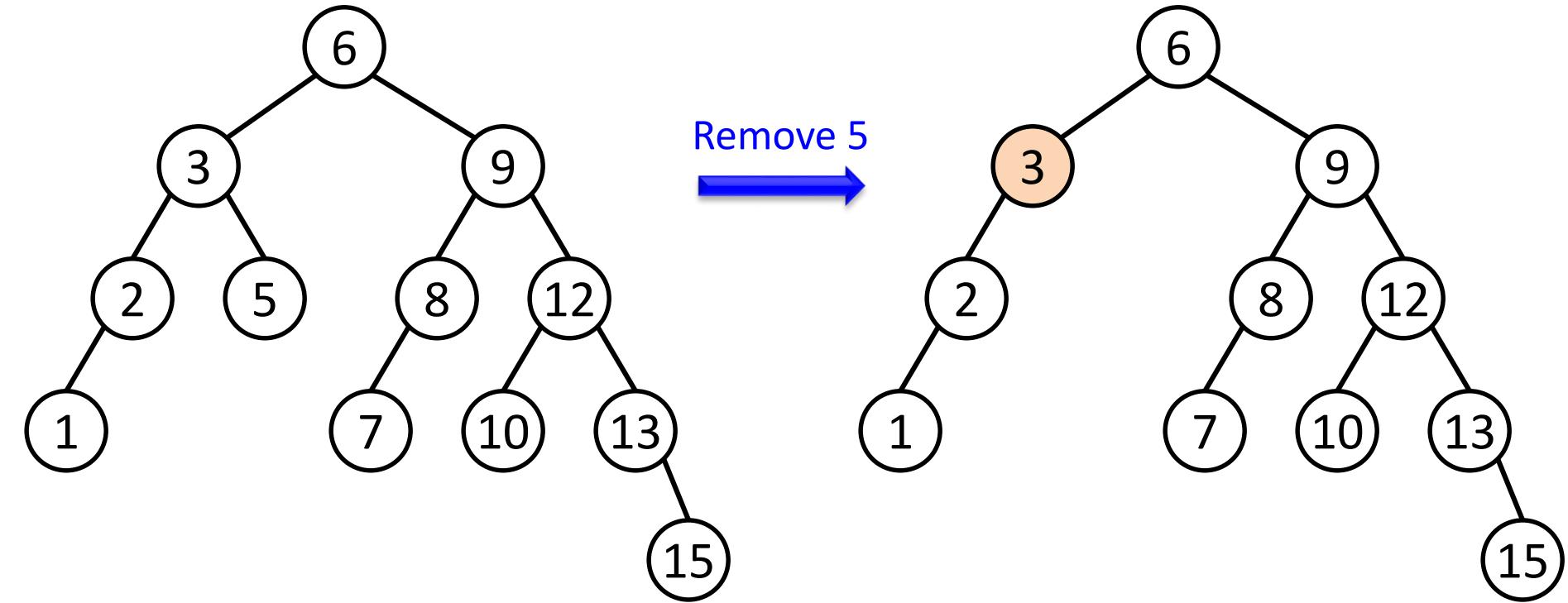
Example: insertions



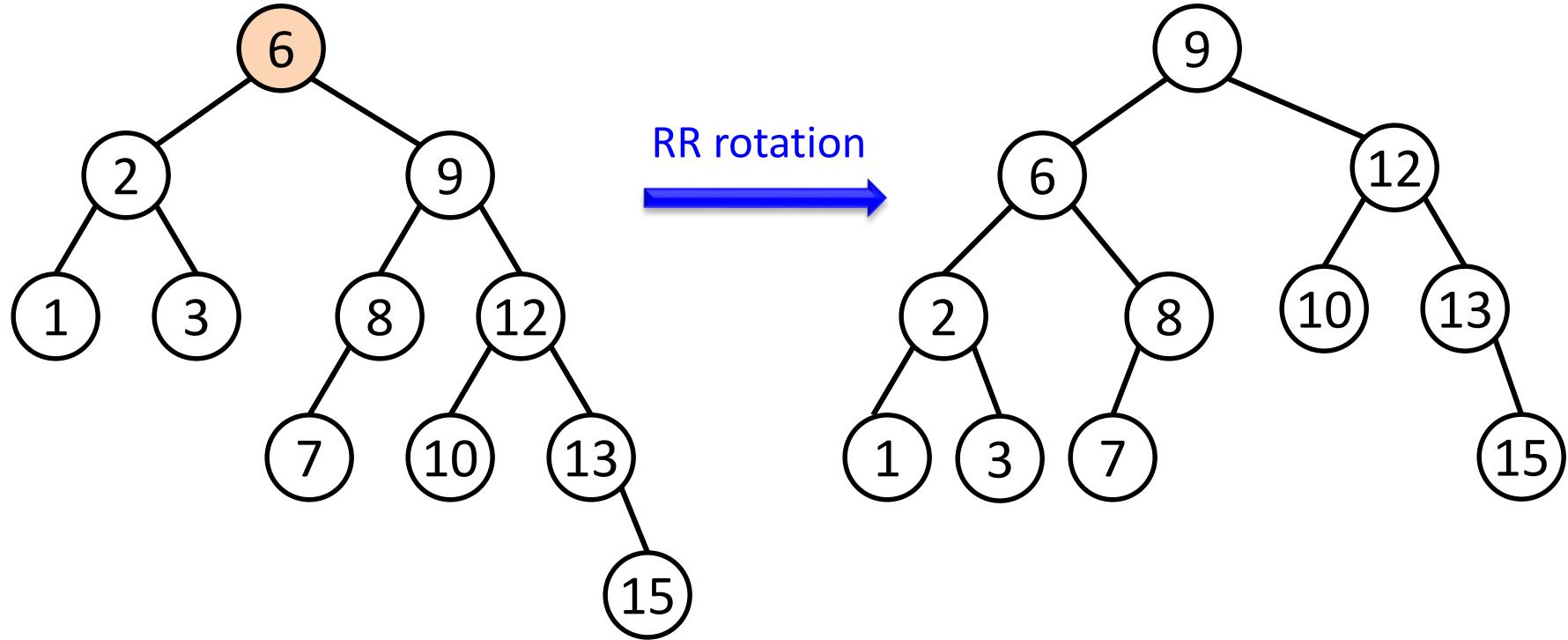
Example: insertions



Example: deletion



Example: deletion



Implementation details

- The height must be stored at each node. Only the unbalancing factor ($\{-1,0,1\}$) is strictly required.
- The insertion/deletion operations are implemented similarly as in BSTs (recursively).
- The re-balancing of the tree is done when the recursive calls return to the ancestors (check heights and rotate if necessary).

Complexity

- Single and double rotations only need the manipulation of few pointers and the height of the nodes ($O(1)$).
- Insertion: the height of the subtree after a rotation is the same as the height before the insertion. Therefore, at most only one rotation must be applied for each insertion.
- Deletion: more complicated. More than one rotation might be required.
- Worst case for deletion: $O(\log n)$ rotations (a chain effect from leaves to root).

EXERCISES

BST

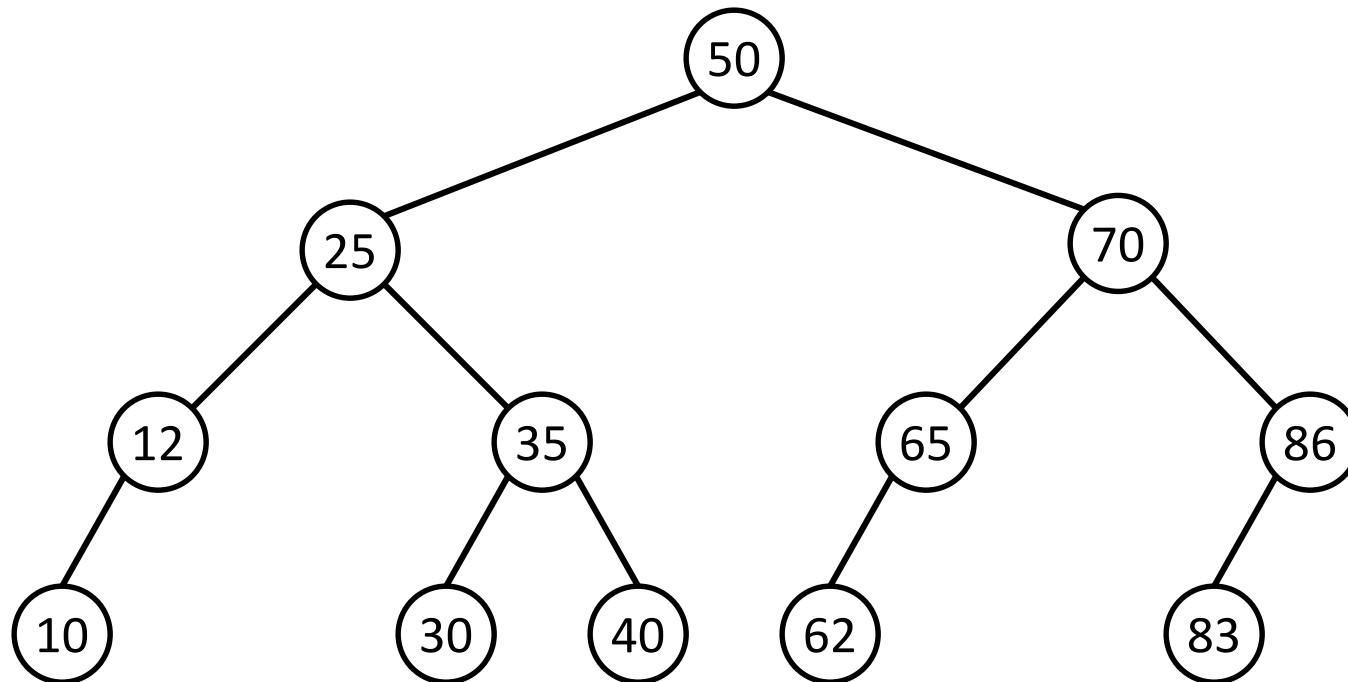
- Starting from an empty BST, depict the BST after inserting the values 32, 15, 47, 67, 78, 39, 63, 21, 12, 27.
- Depict the previous BST after removing the values 63, 21, 15 and 32.

Merging BSTs

- Describe an algorithm to generate a sorted list from a BST. What is its cost?
- Describe an algorithm to create a balanced BST from a sorted list. What is its cost?
- Describe an algorithm to create a balanced BST that contains the union of the elements of two BSTs. What is its cost?

AVL

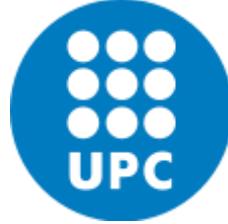
Depict the three AVL trees after sequentially inserting the values 31, 32 and 33 in the following AVL tree:



AVL

- Build an AVL tree by inserting the following values: 15, 21, 23, 11, 13, 8, 32, 33, 27. Show the tree before and after applying each rotation.
- Depict the AVL tree after removing the elements 23 and 21 (in this order). When removing an element, move up the largest element of the left subtree.

Hashing



Jordi Cortadella and Jordi Petit
Department of Computer Science

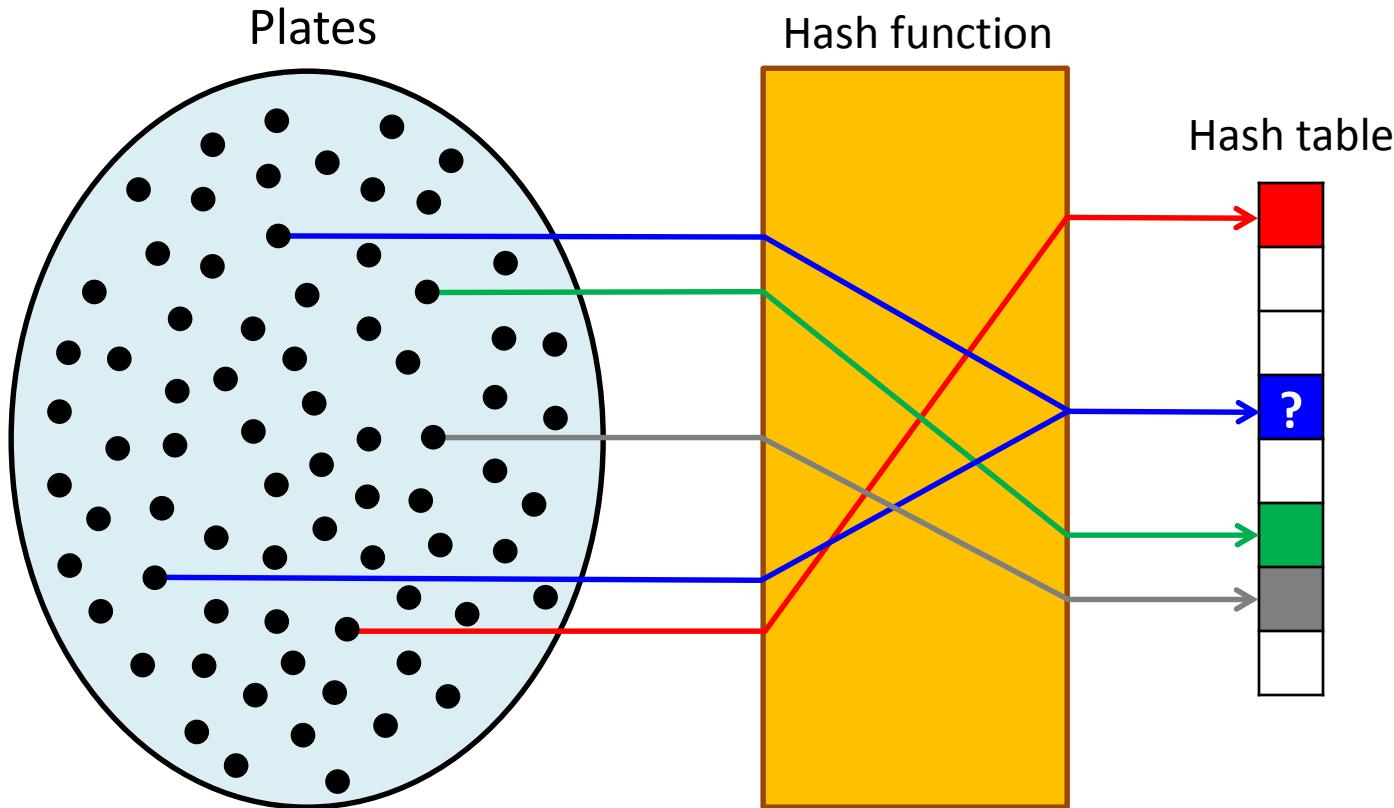
The parking lot

- We want to keep a database of the cars inside a parking lot. The database is automatically updated each time the cameras at the entry and exit points of the parking read the plate of a car.
- Each plate is represented by a free-format short string of alphanumeric characters (each country has a different system).
- The following operations are needed:
 - Add a plate to the database (when a car enters).
 - Remove a plate from the database (when a car exits).
 - Check whether a car is in the parking.
- **Constraint:** we want the previous operations to be very efficient, i.e., executed in ***constant time***.
(This constraint is overly artificial, since the activity in a parking lot is extremely slow compared to the speed of a computer.)

Naïve implementation options

- Lists, vectors or binary search trees are not valid options, since the operations take too long:
 - Unsorted lists: adding takes $O(1)$. Removing/checking takes $O(n)$.
 - Sorted vector: adding/removing takes $O(n)$. Checking takes $O(\log n)$.
 - AVL trees: adding/removing/checking takes $O(\log n)$.
- A (Boolean) vector with one location for each possible plate:
 - The operations could be done in constant time!, but ...
 - The vector would be extremely large (e.g., only the Spanish system can have 80,000,000 different plates).
 - We may not even know the size of the domain (all plates in the world).
 - Most of the vector locations would be “empty” (e.g. assume that the parking has 1,000 places).
- Can we use a data structure with size $O(n)$, where n is the size of the parking?

Hashing



A hash function maps data of arbitrary size to a table of fixed size.
Important questions:

- How to design a good hash function?
- How to handle collisions?

Hash function

- We can calculate the location for item x as

$$h(x) \bmod m$$

where h is the hash function and m is the size of the hash table.

- A good hash function must scatter items *randomly* and *uniformly* (to minimize the impact of collisions).
- A hash function must also be *consistent*, i.e., give the same result each time it is applied to the same item.

Hashing the plates: some attempts

- Add the last three characters (e.g., ASCII codes) of plate:

$$h(x) = x_{n-1} + x_{n-2} + x_{n-3}$$

Bad choice: For the Spanish system, this would concentrate the values between 198 (BBB) and 270 (ZZZ).

- Multiply the last three characters:

$$h(x) = x_{n-1} \cdot x_{n-2} \cdot x_{n-3}$$

The values are distributed between 287,496 and 729,000. However the distribution is not uniform. The last three characters denote the age of the car. The population of new cars is larger than the one of old cars (e.g., about 15% of the cars are less than 1-year old).

Moreover: consecutive plates would fall into the same slot. Some companies (e.g., car renting) have cars with consecutive plates and they could be located in the neighbourhood of the parking lot.

Hashing the plates: some attempts

- Multiply all characters of the plate:

$$h(x) = x_0 \cdot x_1 \cdots x_{n-1}$$

Better choice, but not fully random and uniform. Two plates with permutations of characters would fall into the same slot, e.g., 3812 DXF and 8321 FDX.

- The perfect hash function does not exist, but using prime numbers is a good option since most data have no structure related to prime numbers.
- Where can we use prime numbers?
 - In the size of the hash table
 - In the coefficients of the hash function

Example of hash function for strings

- A usual hash function for a string with size n is as follows:

$$h(x) = \sum_{i=0}^{n-1} x_i \cdot p^i$$

where p is a prime number and x_i is the character at location i . This function can be efficiently implemented using Horner's rule for the evaluation of a polynomial.

- Here is a slightly different implementation (reversed string):

```
/** Hash function for strings */
unsigned int hash(const string& key, int tableSize) {
    unsigned int hval = 0;
    for (char c: key) hval = 37*hval + c;
    return hval%tableSize;
}
```

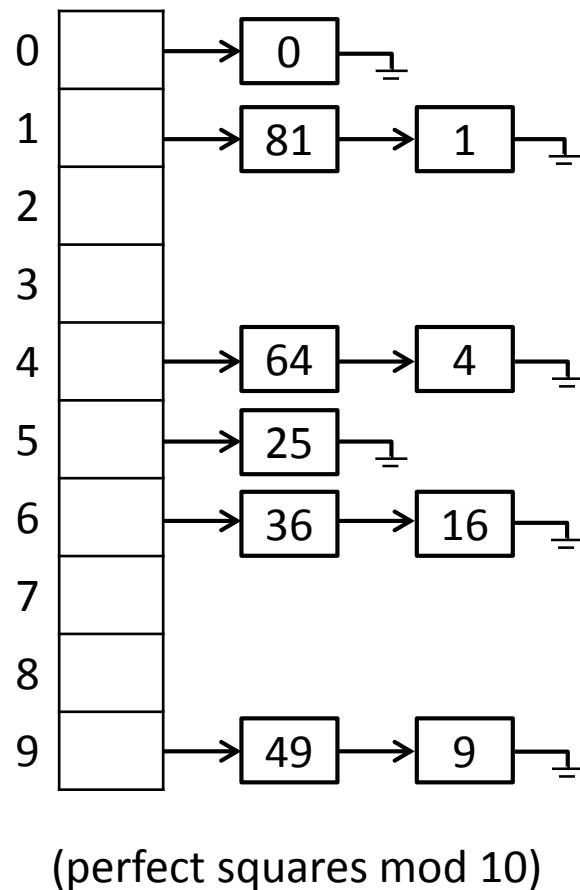
Handling collisions

- A collision is produced when

$$h(x_1) \equiv h(x_2) \bmod m$$

- There are two main strategies to handle collisions:
 - Using lists of items with the same hash value
(separate chaining)
 - Using alternative cells in the same hash table
(linear probing, double hashing, ...)

Separate chaining



Each slot is a list of the items that have the same hash value.

Load factor: $\lambda = \frac{\text{number of items}}{\text{table size}}$

λ is the average length of a list.

A successful search takes $1 + \frac{\lambda}{2}$ links to be traversed, on average. Why?

The search visits 1 node plus half of the other nodes in the list. The number of “other nodes” is about λ .

Table size: make it similar to the number of expected items.

Common strategy: when $\lambda > 1$, do rehashing.

Using the same hash table

- If the slot is occupied, find alternative cells in the same table. To avoid long trips finding empty slots, the load factor should be below $\lambda = 0.5$.
- Deletions must be “lazy” (slots must be invalidated but not deleted, thus avoiding truncated searches).
- Linear probing: if the slot is occupied, use the next empty slot in the table.
- Double hashing: if the slot is occupied using the first hash function h_1 , use a second hash function h_2 . The sequence of slots that is visited is $h_1(x), h_1(x) + h_2(x), h_1(x) + 2h_2(x)$, etc.

Rehashing

- When the table gets too full, the probability of collision increases (and the cost of each operation).
- Rehashing requires building another table with a larger size and rehash all the elements to the new table. Running time: $O(n)$.
- New size: $2n$ (or a prime number close to it). Rehashing occurs very infrequently and the cost is amortized by all the insertions. The average cost remains constant.

A simple implementation

```
template <typename Key, typename Info>
class Dictionary {

private:
    const int None = -1; /* Value for “not found” */
    using Pair = pair<Key, Info>; /* An item */

    /* A list of items with the same hash value */
    using List = vector<Pair>

    vector<List> Table; /* The hash table */
    int n; /* The number of items */
}
```

A simple implementation

public:

```
/** Creates an empty dictionary. Cost: O(M). */
Dictionary(int M = 1009) : Table(M), n(0) { }

/** Assigns info to key. If the key already is in the
 * dictionary, the information is modified.
 * Worst case: O(n). Average case: O(1+n/M). */
void assign (const Key& key, const Info& info) {
    const int h = hash(key) % Table.size();
    const int p = position(key, h);
    if (p != None) Table[h][p].second = info;
    else {
        Table[h].push_back({key, info});
        ++n;
    }
}
```

A simple implementation

/* Erases key and its associated information from the dictionary. If the key does not belong to the dictionary, nothing changes.
Worst case: O(n). Average case: O(1+n/M). */

```
void erase (const Key& key) {  
    const int h = hash(key) % Table.size();  
    const int p = position(key, h);  
    if (p != None) {  
        Table[h][p] = Table[h].back();  
        Table[h].pop_back();  
        --n;  
    }  
}
```

A simple implementation

private:

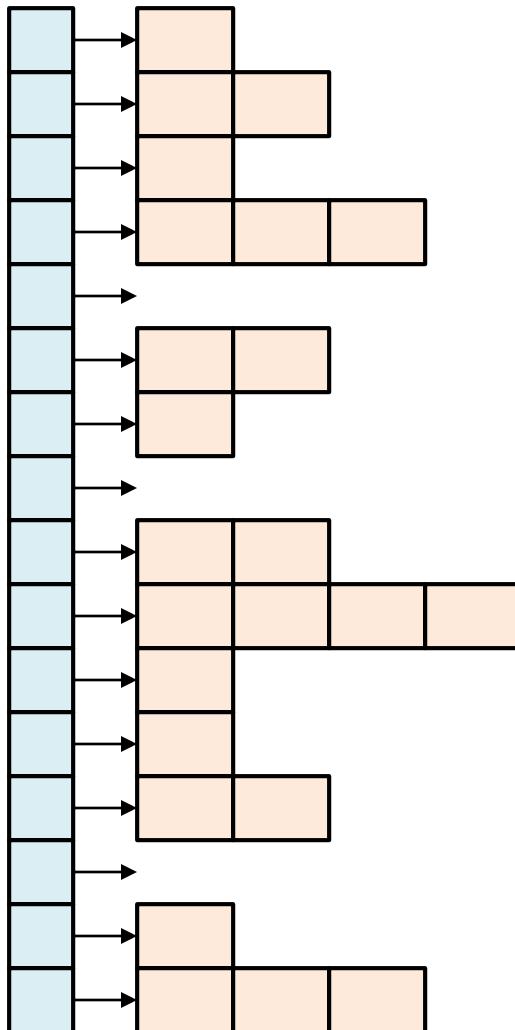
```
/** Returns the position of key in the list of
 * items of slot h, or None if not found. */
int position(const Key& key, int h) {
    const List& L = Table[h];
    for (int i = 0; i < L.size(); ++i) {
        if (L[i].first == key) return i;
    }
    return None;
}
```

Exercises:

- Implement a method that returns the information associated to a key.
- Implement rehashing when the load factor is $\lambda \geq 1$.

Complexity analysis

M slots n items



The hash table occupies $O(M + n)$ space.
Each slot has n/M items, on average.
The runtime to find an item is $O(n/M)$, on average.

Cases	Space: $O(n + M)$	Time: $O(n/M)$
$M \gg n$	$O(M)$	$O(1)$
$n \gg M$	$O(n)$	$O(n)$
$M = O(n)$	$O(n)$	$O(1)$

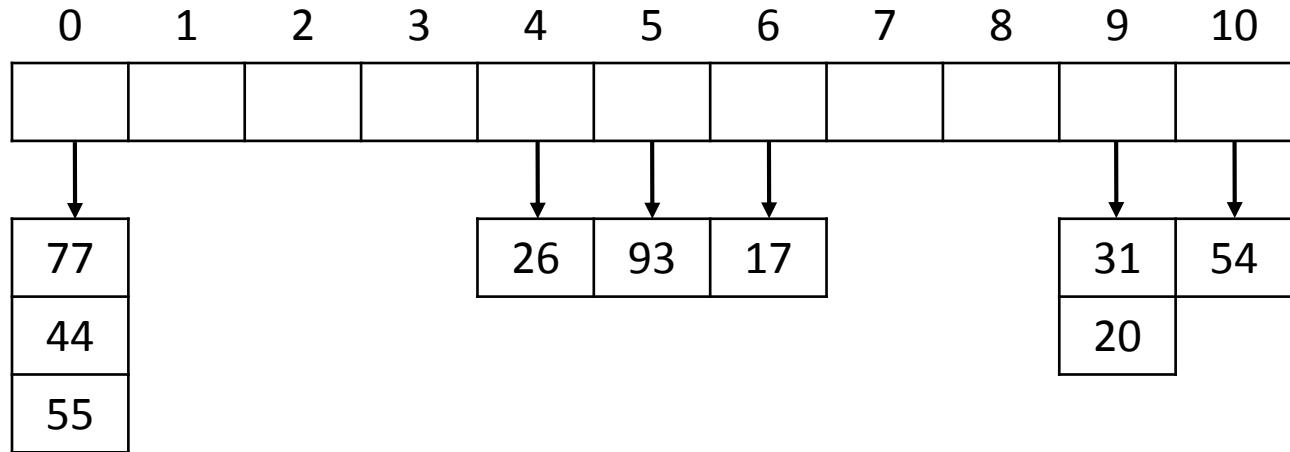
The best strategy is to have $M = O(n)$ that allows to maintain a constant-time access without wasting too much memory.

Rehashing should be applied to maintain $M = O(n)$.

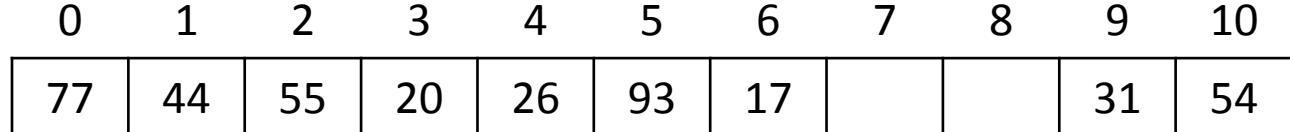
An example

Insertion of the elements 54, 26, 93, 17, 77, 31, 44, 55, 20.
Hash function: $h(x) = x \bmod 11$.

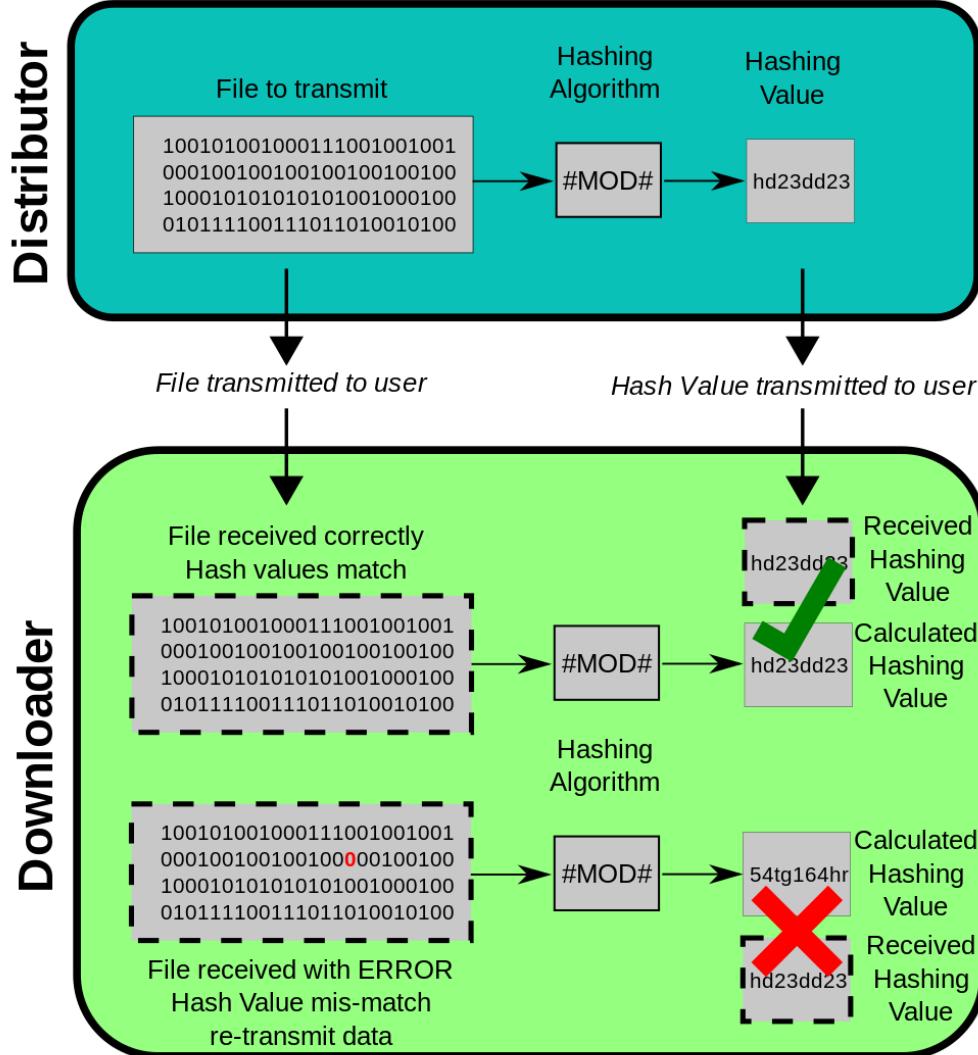
Separate chaining:



Linear probing:



Application: data integrity check

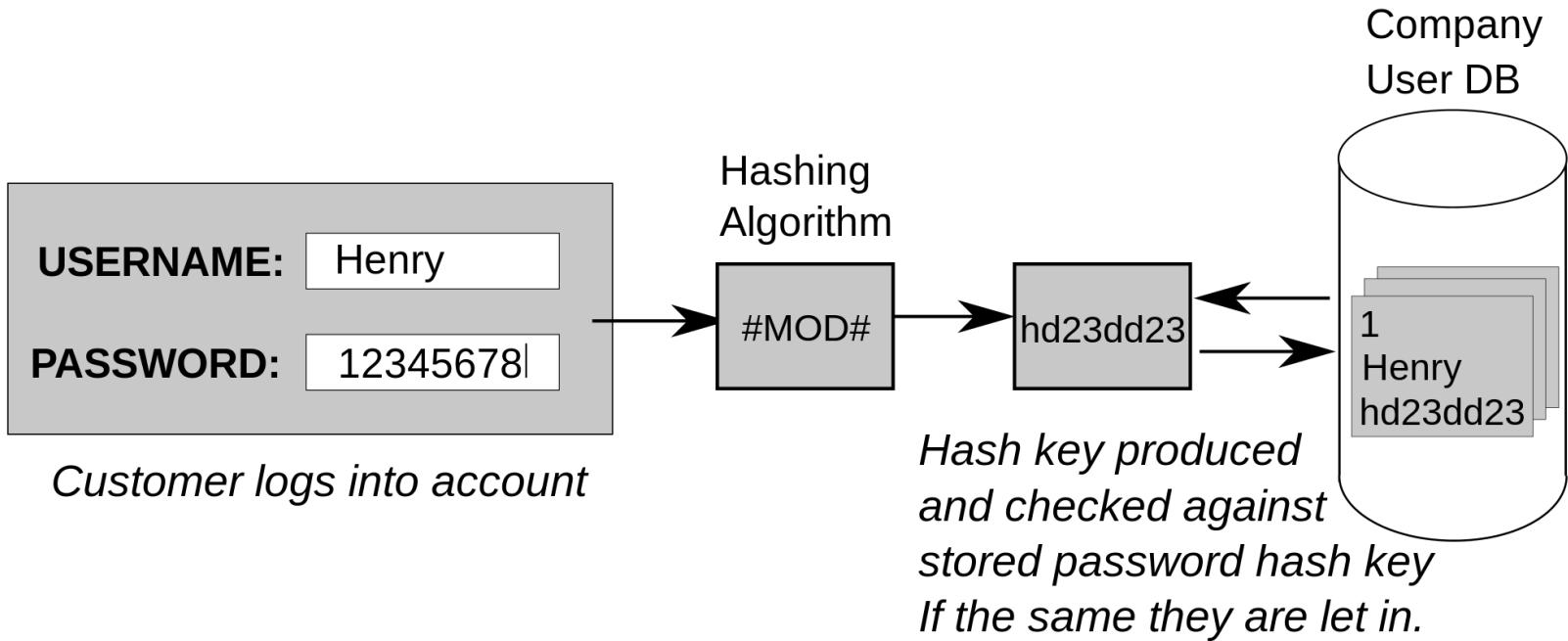


Hash functions are used to guarantee the integrity of data (files, messages, etc) when distributed between different locations.

Different hashing algorithms exist:
MD5, SHA1, SHA255, ...

The probability of collision is extremely low.

Application: password verification



Security is based on the fact that hashing functions are cryptographic (not reversible).

Be careful: there are databases of hash values for “popular” passwords (e.g., 1234, qwert, Messi10, Barcelona92,...).

EXERCISES

Hash function

Given the values {2341, 4234, 2839, 430, 22, 397, 3920}, a hash table of size 7, and hash function $h(x) = x \bmod 7$, show the resulting tables after inserting the values in the given order with each of these collision strategies:

- Separate chaining
- Linear probing

All elements different

Let us assume that we have a list with n elements. Design an algorithm that can check that all elements are different. Analyze the complexity of the algorithm considering different data structures:

- Checking the elements without any additional data structure, i.e., using the same list.
- Using AVLs.
- Using hash tables.