

# Lecture Notes on Data Structures and Algorithms

Conrado Martínez  
U. Politècnica Catalunya

April 1, 2016



# Syllabus

- Part 1: Analysis of Algorithms
- Part 2: Divide and Conquer
- Part 3: Dictionaries
- Part 4: Priority Queues
- Part 5: Graphs
- Part 6: Exhaustive Search and Generation
- Part 7: Notions of Intractability

# Part I

## Analysis of Algorithms

- 1 Introduction
- 2 Asymptotic Notation
- 3 Analysis of Iterative Algorithms
- 4 Analysis of Recursive Algorithms

# Complexity of Algorithms

- Complexity of an algorithm = computational resources it consumes: execution time, memory space
- Analysis of algorithms → Investigate the properties of the complexity of algorithms
  - Compare alternative algorithmic solutions
  - Predict the resources that an algorithm or data structure will use
  - Improve existing algorithms and data structures and guide the design of novel algorithms and DS

# Complexity of Algorithms

In general terms, given an algorithm  $A$  with input set  $\mathcal{A}$ , its **complexity** or **cost** (in time, in memory space, in I/Os, etc.) is a function  $T$  from  $\mathcal{A}$  to  $\mathbb{N}$  (or  $\mathbb{Q}$  or  $\mathbb{R}$ , depending on what we want to study):

$$\begin{aligned}T : \mathcal{A} &\rightarrow \mathbb{N} \\ \alpha &\rightarrow T(\alpha)\end{aligned}$$

Characterizing such a function is too complex and the huge amount of information it yields cannot be handled, and is impractical.

## Worst-, Best-, Average-case Complexity

Let  $\mathcal{A}_n$  denote the set of inputs of size  $n$  and  $T_n : \mathcal{A}_n \rightarrow \mathbb{N}$  the restriction of  $T$  to  $\mathcal{A}_n$ .

- *Best-case cost:*

$$T_{\text{best}}(n) = \min\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Worst-case cost:*

$$T_{\text{worst}}(n) = \max\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Average-case cost:*

$$\begin{aligned} T_{\text{avg}}(n) &= \sum_{\alpha \in \mathcal{A}_n} \Pr(\alpha) T_n(\alpha) \\ &= \sum_{k \geq 0} k \Pr(T_n = k). \end{aligned}$$

# Worst-, Best-, Average-case Complexity

- 1 For all  $n \geq 0$  and for all  $\alpha \in \mathcal{A}_n$

$$T_{\text{best}}(n) \leq T_n(\alpha) \leq T_{\text{worst}}(n).$$

- 2 For all  $n \geq 0$

$$T_{\text{best}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{worst}}(n).$$

## Worst-, Best-, Average-case Complexity

In general we will only study the worst-case complexity:

- ① Provides a guarantee on the complexity of the algorithm,  
the cost will **never** exceed the worst-case cost
- ② It is easier to compute than the average-case cost

# Part I

## Analysis of Algorithms

- 1 Introduction
- 2 Asymptotic Notation
- 3 Analysis of Iterative Algorithms
- 4 Analysis of Recursive Algorithms

# Rates of Growth

A fundamental feature of the cost of an algorithm (a function, in general) is its **rate of growth**

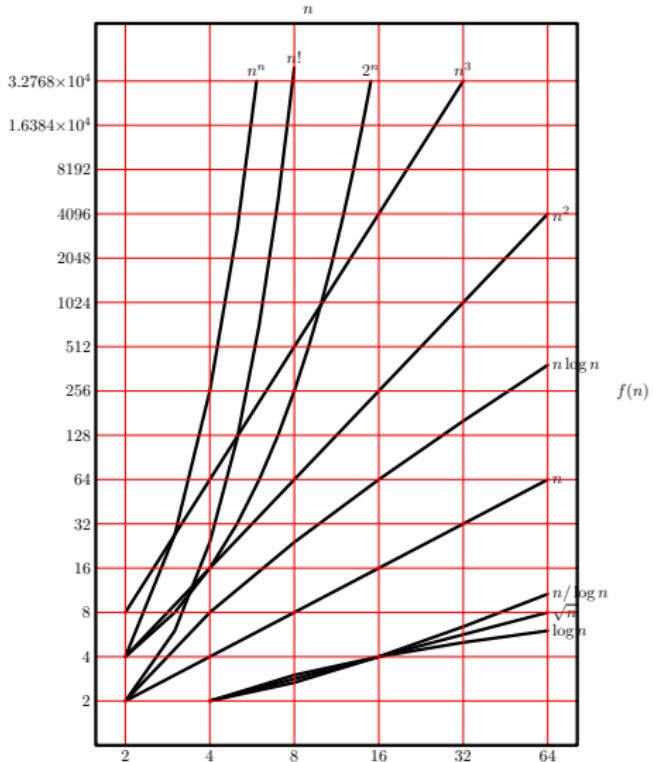
## Example

- ➊ Linear:  $f(n) = a \cdot n + b \Rightarrow f(2n) \approx 2 \cdot f(n)$
- ➋ Quadratic:  $q(n) = a \cdot n^2 + b \cdot n + c \Rightarrow q(2n) \approx 4 \cdot q(n)$

We say that linear and quadratic functions have different rates of growth. We can also say that they are of different **orders of magnitude**.

## Rates of Growth

$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	262144
5	32	160	1024	32768	$6.87 \cdot 10^{10}$
6	64	384	4096	262144	$4.72 \cdot 10^{21}$
...					
$\ell$	$N$	$L$	$C$	$Q$	$E$
$\ell + 1$	$2N$	$2(L + N)$	$4C$	$8Q$	$E^2$



Source: G. Valiente

# Asymptotic Notation: Big-Oh

Constant factors and lower order terms are irrelevant as far as the rate of growth of a function is concerned: for instance,  $30n^2 + \sqrt{n}$  has the same rate of growth as  $2n^2 + 10n \Rightarrow$  **asymptotic notation**

## Definition

Given a function  $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$  the class  $\mathcal{O}(f)$  (big-Oh of  $f$ ) is

$$\mathcal{O}(f) = \{g : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \exists c \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

In words, a function  $g$  is in  $\mathcal{O}(f)$  if there exists a constant  $c$  such that  $g < c \cdot f$  for all  $n$  from some value  $n_0$  onwards.

## Asymptotic Notation: Big-Oh

Although  $\mathcal{O}(f)$  is a set of functions, people often write  $g = \mathcal{O}(f)$  instead of  $g \in \mathcal{O}(f)$ . However, note that  $\mathcal{O}(f) = g$  is nonsensical.

Basic properties of the  $\mathcal{O}$  notation:

- 1 If  $\lim_{n \rightarrow \infty} g(n)/f(n) < +\infty$  then  $g = \mathcal{O}(f)$
- 2 It is reflexive: for all  $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ ,  $f = \mathcal{O}(f)$
- 3 It is transitive: if  $f = \mathcal{O}(g)$  and  $g = \mathcal{O}(h)$  then  $f = \mathcal{O}(h)$
- 4 For all positive constants  $c > 0$ ,  $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

## Asymptotic Notation: Big-Oh

Since constant factors are irrelevant for the asymptotic notation we will systematically omit them: for instance, we will talk about  $\mathcal{O}(n)$ , not about  $\mathcal{O}(4 \cdot n)$  (it is the same class); we will not express the base of logarithms unless they appear in an exponent, hence we will write  $\mathcal{O}(\log n)$ ; we can change from one base to another multiplying by appropriate factor:

$$\log_c x = \frac{\log_b x}{\log_b c}$$

## Asymptotic Notation: Omega and Theta

Other asymptotic notations include  $\Omega$  (omega) and  $\Theta$  (zeta).  $\Omega$  defines the set of functions with rate of growth is bounded from below by the rate of growth of the given function:

$$\Omega(f) = \{g: \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \exists c > 0 \forall n \geq n_0: g(n) \geq c \cdot f(n)\}$$

$\Omega$  is reflexive and transitive; if  $\lim_{n \rightarrow \infty} g(n)/f(n) > 0$  then  $g = \Omega(f)$ . On the other hand,  $\Omega$  and  $\mathcal{O}$  are related as follows: if  $f = \mathcal{O}(g)$  then  $g = \Omega(f)$ , and vice-versa.

## Asymptotic Notation: Omega and Theta

We will often say that  $\mathcal{O}(f)$  is the class of function that grow no faster than  $f$ . Analogously,  $\Omega(f)$  is the class of functions that grow at least as fast as  $f$ .

Finally,

$$\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$$

is the class of functions with the same rate of growth as  $f$ .

$\Theta$  is reflexive and transitive, as the other notations, but it is also symmetric:  $f = \Theta(g)$  if and only if  $g = \Theta(f)$ . If  $\lim_{n \rightarrow \infty} g(n)/f(n) = c$  for some  $c$ ,  $0 < c < \infty$  then  $g = \Theta(f)$ .

## Asymptotic Notation

Additional properties of the asymptotic notations (set inclusions are strict):

- ① For any two constants  $\alpha$  and  $\beta$ , with  $\alpha < \beta$ , if  $f$  is an increasing function then  $\mathcal{O}(f^\alpha) \subset \mathcal{O}(f^\beta)$ .
- ② For any two constants  $a$  and  $b > 0$ , if  $f$  is an increasing function then  $\mathcal{O}((\log f)^a) \subset \mathcal{O}(f^b)$ .
- ③ For any constant  $c > 1$ , if  $f$  is an increasing function  $\mathcal{O}(f) \subset \mathcal{O}(c^f)$ .

## Asymptotic Notation

Conventional operations like sums, subtractions, division, etc. can be extended to classes of functions (as defined by asymptotic notations) as follows:

$$A \otimes B = \{h \mid \exists f \in A \wedge \exists g \in B : h = f \otimes g\},$$

where  $A$  and  $B$  are two sets of functions. Expressions of the form  $f \otimes A$ , where  $f$  a function, denote  $\{f\} \otimes A$ .

With these conventions we can now write expressions such as  $n + \mathcal{O}(\log n)$ ,  $n^{\mathcal{O}(1)}$ , or  $\Theta(1) + \mathcal{O}(1/n)$ .

## Asymptotic Notation: Rule of the sums and products

Rule of sums:

$$\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\}).$$

Rule of products:

$$\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g).$$

Similar rules hold for  $\mathcal{O}$  and  $\Omega$ .

# Part I

## Analysis of Algorithms

- 1 Introduction
- 2 Asymptotic Notation
- 3 Analysis of Iterative Algorithms
- 4 Analysis of Recursive Algorithms

# Analysis of Iterative Algorithms

- ① The cost of an elementary operation (e.g., comparing two integers) is  $\Theta(1)$ .
- ② If the cost of the fragment  $S_1$  is  $f$  and that of  $S_2$  is  $g$  then the cost of  $S_1; S_2$  is  $f + g$  (sequential composition).
- ③ If the cost of  $S_1$  is  $f$ , that of  $S_2$  is  $g$  and the cost of evaluating the Boolean expression  $B$  is  $h$  then the worst-case cost of

```
if  $B$  then  $S_1$ 
else  $S_2$ 
end if
```

is  $\mathcal{O}(\max\{f + h, g + h\})$ .

# Analysis of Iterative Algorithms

- ④ If the cost of  $S$  in the  $i$ -th iteration is  $f_i$ , the cost of evaluating  $B$  is  $h_i$  and the number of iterations is  $g$ , then the cost of  $T$  of

**while**  $B$  **do**

$S$

**end while**

is

$$T(n) = \sum_{i=1}^{i=g(n)} f_i(n) + h_i(n).$$

If  $f = \max\{f_i + h_i\}$  then  $T = \mathcal{O}(f \cdot g)$ .

# Analysis of Iterative Algorithms

```
// example of use:  
//   vector<int> my_vector = read_data();  
//   cout << "min = " << minimum(v.begin(), v.end()) << endl;  
  
template <class Elem, class Iter>  
Elem minimum(Iter beg, Iter end) {  
    if (beg == end) throw NullSequenceError;  
    Elem min = *beg; ++beg;  
    for (Iter curr = beg; curr != end; ++curr)  
        if (*curr < min) min = *curr;  
    return min;  
}
```

## Example (Finding the minimum)

If a comparison between two `Elem`'s or an assignment of an `Elem` to a variable (e.g., `min = *curr`) are elementary operations then

- 1 In the worst-case, the body of the `for` loop takes time  $\Theta(1)$ ; the increment of iterators is also  $\Theta(1)$
- 2 Comparing two iterators is  $\Theta(1)$  since we need only to check that they “point” to the same object
- 3 If the length of the sequence is  $n$  ( $n = \text{end}-\text{beg}$ ) then the loop is executed  $n - 1$  times. Applying the rule of products we have then

$$F(n) = (n - 1) \cdot \Theta(1) = \Theta(n) \cdot \Theta(1) = \Theta(n)$$

# Analysis of Iterative Algorithms

```
typedef vector<double> Row;
typedef vector<Row> Matrix;

// we can use the newly defined operator like this: Matrix C = A * B;
// Pre: A[0].size() == B.size()
Matrix operator*(const Matrix& A, const Matrix& B) {
    if (A[0].size() != B.size()) throw IncompatibleMatrixProduct;
    int m = A.size();
    int n = A[0].size();
    int p = B[0].size();
    Matrix C(m, Row(p, 0.0)); // C = m x p matrix initialize to 0.0
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < p; ++j)
            for (int k = 0; k < n; ++k)
                C[i][j] += A[i][k] * B[k][j];
    return C;
}
```

## Example (Matrix multiplication)

The algorithm above computes the matrix product of  $A = (A_{ij})_{m \times n}$  and  $B = (B_{ij})_{n \times p}$  using its definition:

$$C_{ij} = \sum_{k=0}^n A_{ik} \cdot B_{kj}$$

# Analysis of Iterative Algorithms

## Example (Matrix multiplication (cont'd))

- ➊ The body of the innermost **for** loop (on  $k$ ) has cost  $\Theta(1)$ . Thus the body of the second **for** loop (on  $j$ ) is, applying the rule of products,  $\Theta(n)$ .
- ➋ Similarly the body of the outermost loop (on  $i$ ) has cost  $\Theta(p \cdot n)$ .
- ➌ Thus the cost of the three nested loops is  $\Theta(m \cdot p \cdot n)$ .
- ➍ The other parts of the algorithm have cost  $\Theta(m \cdot p)$ . By the rule of sums, the overall cost of the algorithm is  $\Theta(m \cdot n \cdot p)$ .
- ➎ For square matrices, setting  $N = m = n = p$ , the cost of the algorithm is  $\Theta(N^3)$ .

# Analysis of Iterative Algorithms

```
template <class T, class Comp = std::less<T>>
void insertion_sort(vector<T>& A, Comp smaller) {
    int n = A.size();
    for (int i = 1; i < n; ++i) {
        // put A[i] into its place in A[0..i-1]
        T x = A[i]; int j = i - 1;
        while (j >= 0 and smaller(x, A[j])) {
            A[j+1] = A[j];
            --j;
        };
        A[j] = x;
    }
}
```

## Example (Insertion sort)

Insertion sort is one of the so-called *elementary sort algorithms*. It is very easy to understand and to program. Its running time for any instance is both  $\Omega(n)$  and  $\mathcal{O}(n^2)$ . In particular, the best-case is  $\Theta(n)$  and the worst-case is  $\Theta(n^2)$ .

# Analysis of Iterative Algorithms

```
template <class T, class Comp = std::less<T>>
void insertion_sort(vector<T>& A, Comp smaller) {
    int n = A.size();
    for (int i = 1; i < n; ++i) {
        // put A[i] into its place in A[0..i-1]
        T x = A[i]; int j = i - 1;
        while (j >= 0 and smaller(x, A[j])) {
            A[j+1] = A[j];
            --j;
        };
        A[j] = x;
    }
}
```

## Example (Insertion sort (cont'd))

- ① The `while` can make any number of iterations from 0 (when the vector is already sorted) to  $i$  (when the vector is in reverse order). Its cost is  $\Theta(i)$  assuming that the cost of the comparison `smaller` is  $\Theta(1)$ , and the assignment between elements of class `T` takes also constant time.
- ② Thus the cost of the `for` loop in the worst-case is

$$\sum_{i=1}^{n-1} \Theta(i) = \Theta\left(\sum_{i=1}^{n-1} i\right) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2)$$

# Analysis of Iterative Algorithms

```
template <class T, class Comp = std::less<T>>
void insertion_sort(vector<T>& A, Comp smaller) {
    int n = A.size();
    for (int i = 1; i < n; ++i) {
        // put A[i] into its place in A[0..i-1]
        T x = A[i]; int j = i - 1;
        while (j >= 0 and smaller(x, A[j])) {
            A[j+1] = A[j];
            --j;
        };
        A[j] = x;
    }
}
```

## Example (Insertion sort (cont'd))

- ③ A quick upper bound follows by observing that the cost of the `while` loop is  $\mathcal{O}(i) = \mathcal{O}(n)$ , hence the cost of the algorithm is  $\mathcal{O}(n^2)$ .
- ④ The cost of the `for` loop is  $\Theta(n)$  in the best case, since the cost of the  $i$ -th iteration in the best case is  $\Theta(1)$ .
- ⑤ The average cost of the algorithm is also  $\Theta(n^2)$ , assuming each of the  $n!$  possible initial orderings of the vector is equally likely. The inner `while` loop will perform, on average,  $\approx i/2$  iterations when inserting  $A[i]$ .

# Part I

## Analysis of Algorithms

- 1 Introduction
- 2 Asymptotic Notation
- 3 Analysis of Iterative Algorithms
- 4 Analysis of Recursive Algorithms

## Analysis of Recursive Algorithms

The cost  $T(n)$  (worst-, best-, average-case) of a recursive algorithm satisfies a **recurrence**: an equation where  $T$  appears in both sides, with  $T(n)$  depending on  $T(k)$  for one or more values  $k < n$ . Recurrences appear often in one of the two following forms:

$$\begin{aligned}T(n) &= a \cdot T(n - c) + f(n), \\T(n) &= a \cdot T(n/b) + f(n).\end{aligned}$$

First correspond to algorithms where the non-recursive part has cost  $f(n)$  and they make  $a$  recursive calls on inputs of size  $n - c$ , for some constant  $c$ .

Second corresponds to algorithm with non-recursive cost  $f(n)$  making  $a$  recursive calls on inputs of size (approx.)  $n/b$ , where  $b > 1$ .

# Analysis of Recursive Algorithms

## Theorem

Let  $T(n)$  satisfy the recurrence

$$T(n) = \begin{cases} g(n) & \text{if } 0 \leq n < n_0 \\ a \cdot T(n - c) + f(n) & \text{if } n \geq n_0, \end{cases}$$

where  $n_0$  is a constant,  $c \geq 1$ ,  $g(n)$  is an arbitrary function, and  $f(n) = \Theta(n^k)$  for some constant  $k \geq 0$ .

Then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < 1 \\ \Theta(n^{k+1}) & \text{if } a = 1 \\ \Theta(a^{n/c}) & \text{if } a > 1. \end{cases}$$

# Analysis of Recursive Algorithms

## Theorem

Let  $T(n)$  satisfy the recurrence

$$T(n) = \begin{cases} g(n) & \text{if } 0 \leq n < n_0 \\ a \cdot T(n/b) + f(n) & \text{if } n \geq n_0, \end{cases}$$

where  $a \geq 1$ ,  $b > 1$  and  $n_0$  constants,  $g(n)$  is an arbitrary function and  $f(n) = \Theta(n^k)$  for some constant  $k \geq 0$ .

Let  $\alpha = \log_b a$ . Then

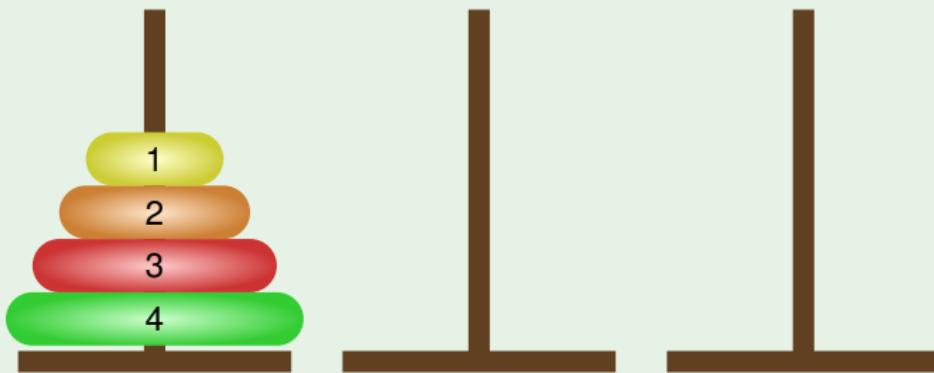
$$T(n) = \begin{cases} \Theta(n^k) & \text{if } \alpha < k \\ \Theta(n^k \log n) & \text{if } \alpha = k \\ \Theta(n^\alpha) & \text{if } \alpha > k. \end{cases}$$

The conditions  $\alpha < k$ ,  $\alpha = k$  and  $\alpha > k$  are equivalent to  $a < b^k$ ,  $a = b^k$  and  $a > b^k$ , respectively.

# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

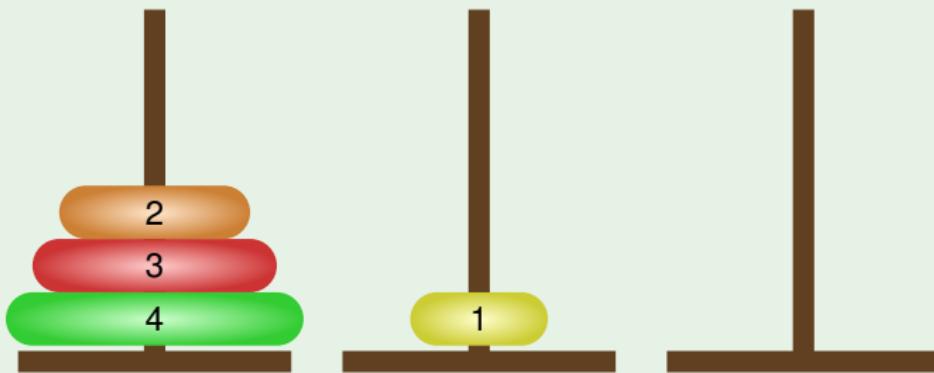
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

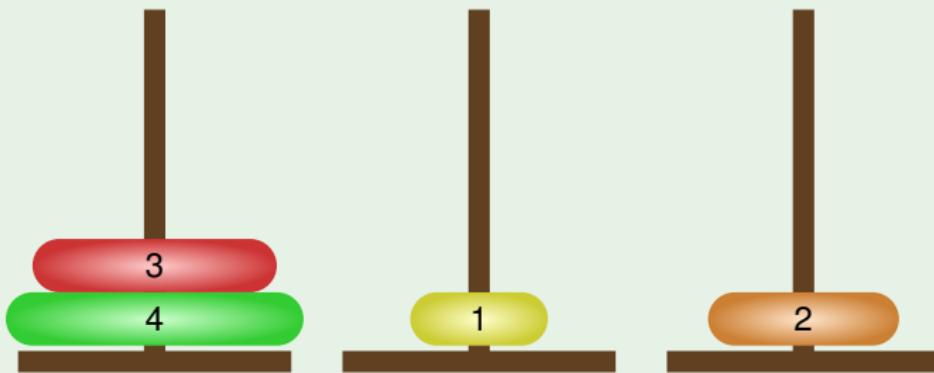
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

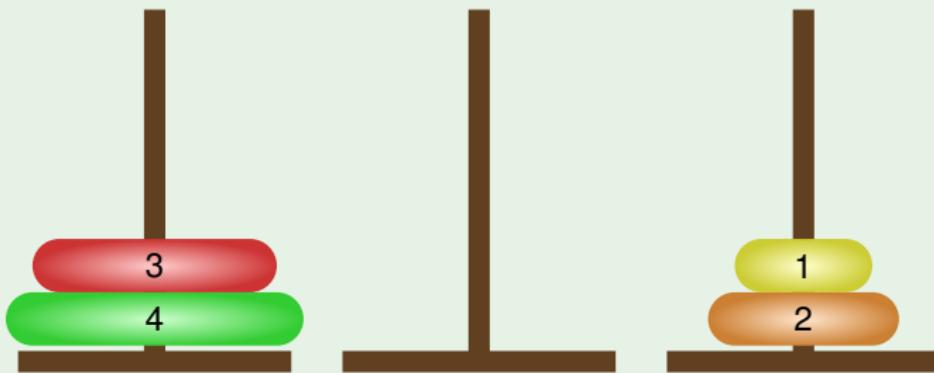
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

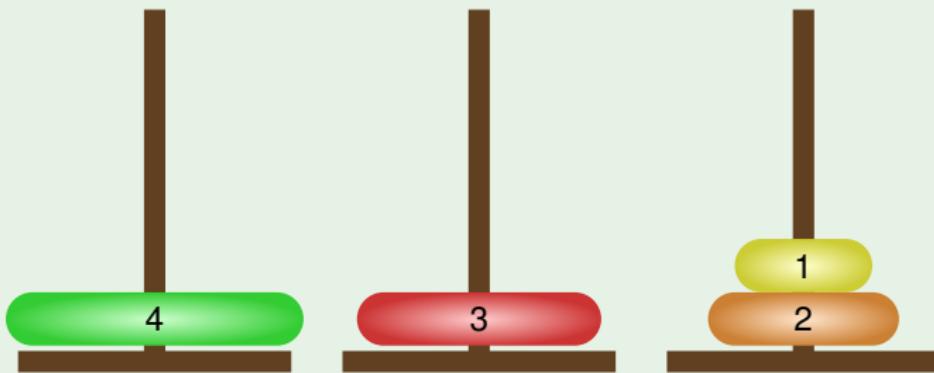
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

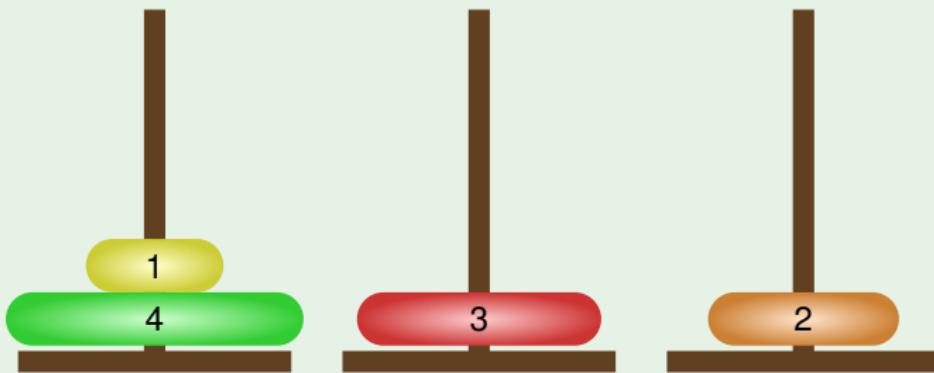
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

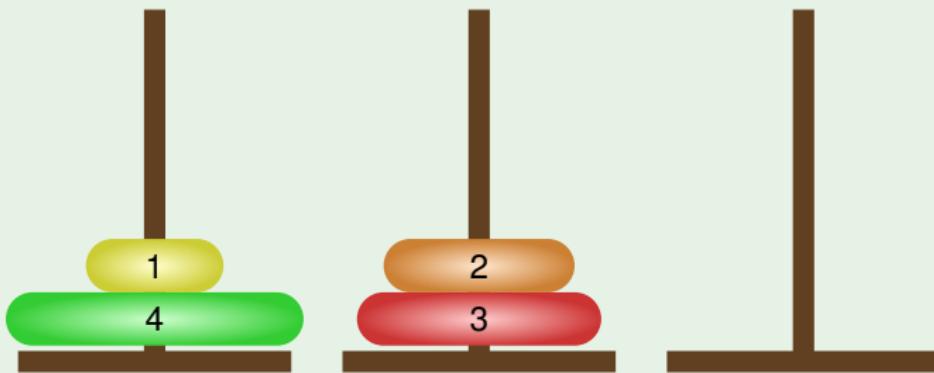
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

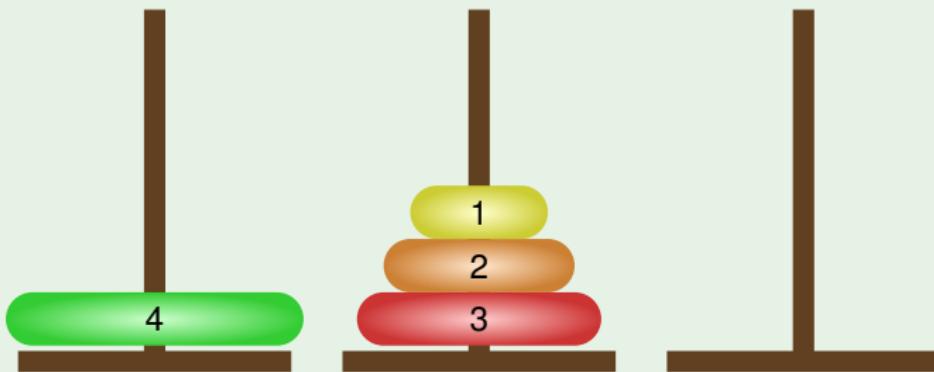
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

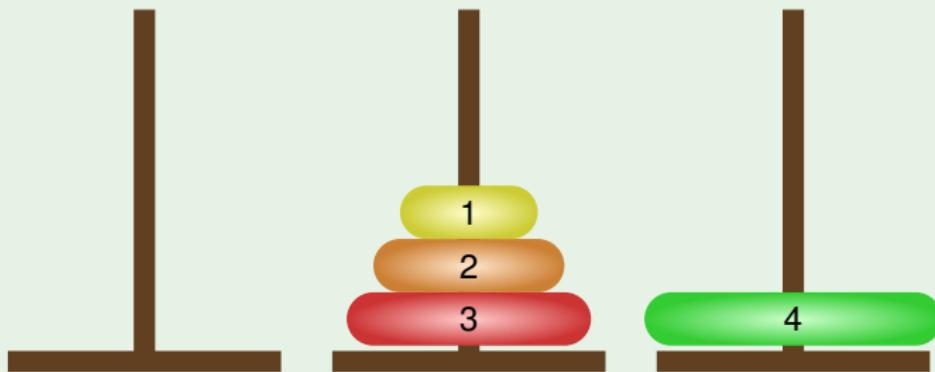
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

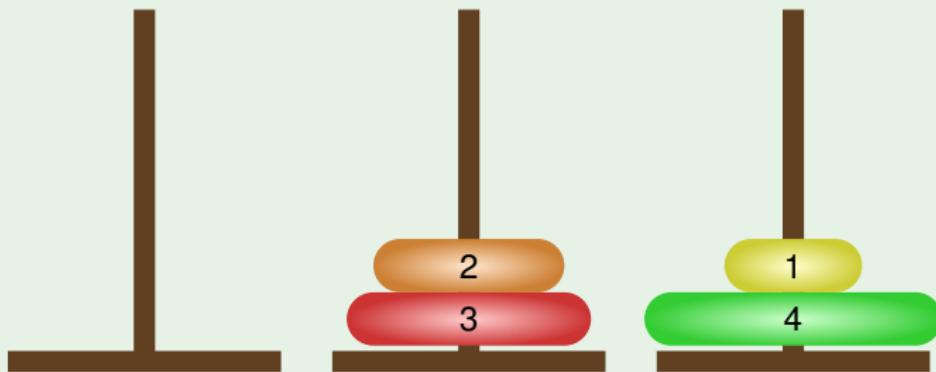
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

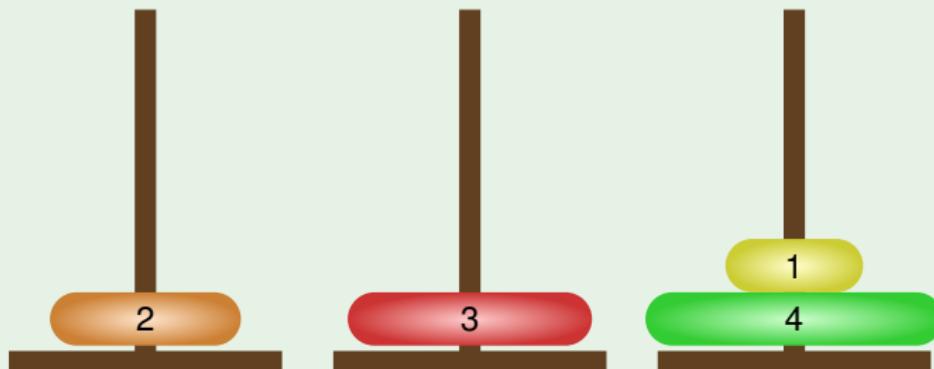
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

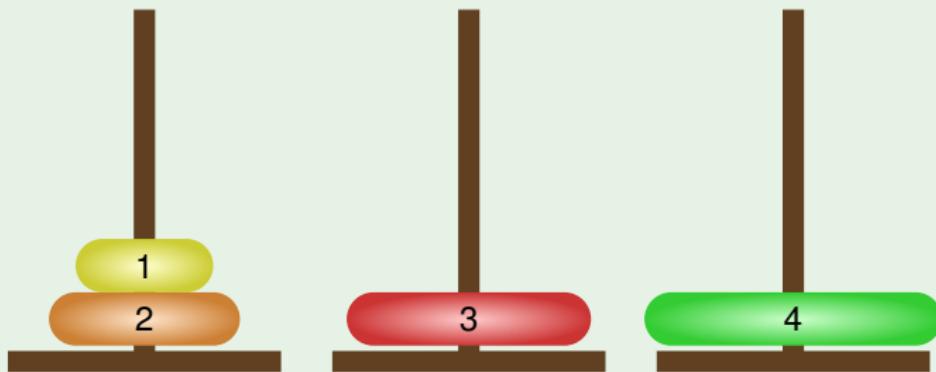
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

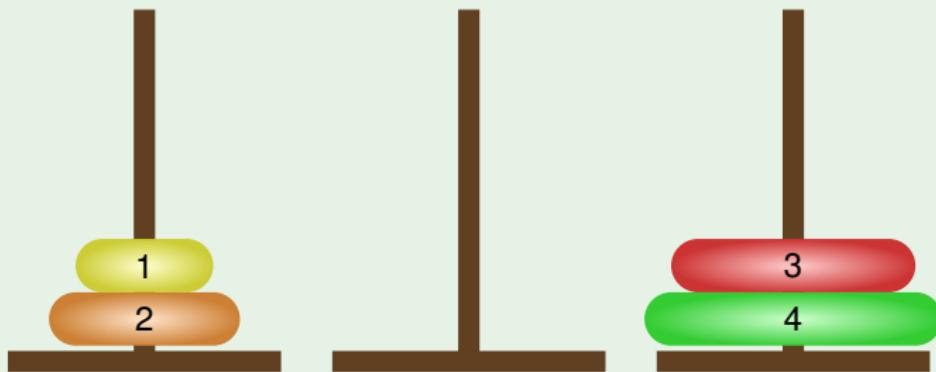
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

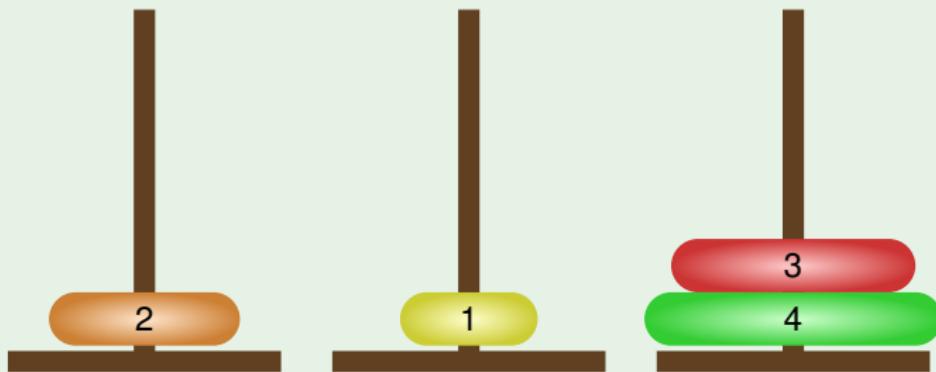
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

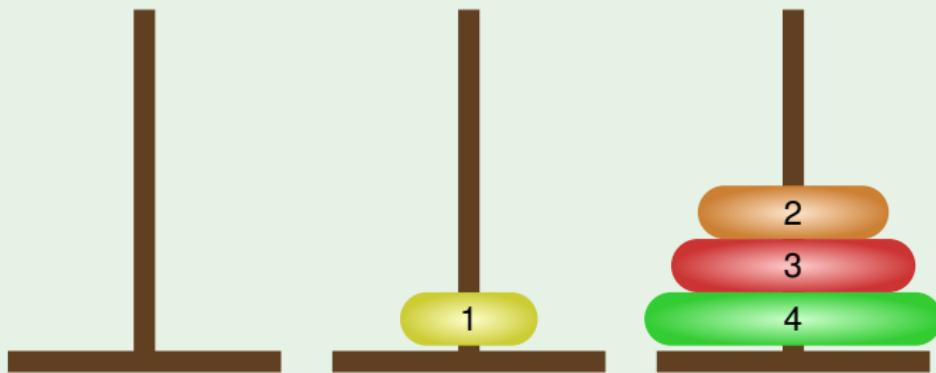
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

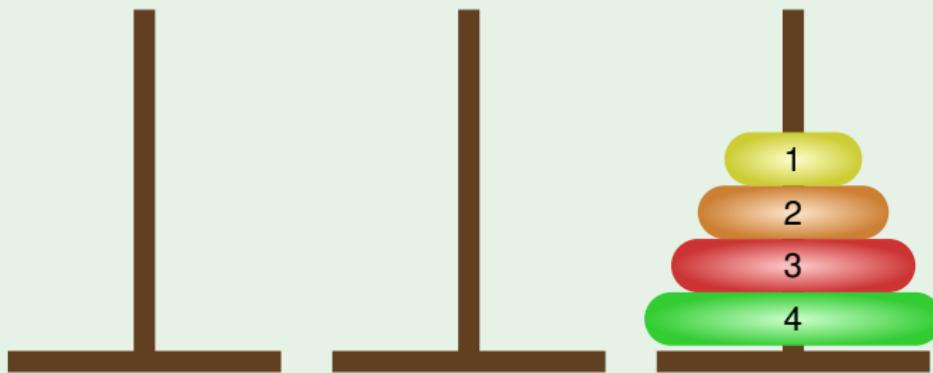
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

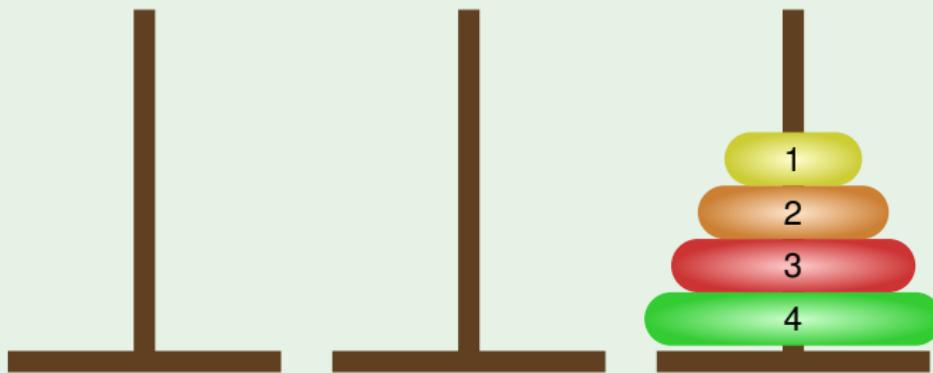
The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

## Example (Towers of Hanoi)

The Towers of Hanoi is a puzzle in which we have  $n$  disks of decreasing diameters with a hole in their center and three poles A, B and C. The  $n$  disks initially sit in pole A and they must be transferred, one by one, to pole C, using pole B for intermediate movements. The rule is that no disk can be put on top of a disk with a larger diameter.



# Analysis of Recursive Algorithms

```
typedef char pole;

// Initial call: hanoi(n,'A', 'B', 'C');

void hanoi(int n, pole org, pole aux, pole dst) {
    if (n == 1)
        cout << "Move from " << org << " to " << dst << endl;
    else {
        hanoi(n - 1, org, dst, aux);
        // move the largest disk
        cout << "Move from " << org << " to " << dst << endl;
        hanoi(n - 1, aux, org, dst);
    }
}
```

## Example (Towers of Hanoi (cont'd))

The cost  $f(n)$  of the non-recursive part is  $\Theta(1)$ , and for  $n \leq n_0 = 1$  the cost is also  $\Theta(1)$ . The recurrence that describes the cost  $H(n)$  of `hanoi` is

$$H(n) = 2H(n - 1) + \Theta(1), \quad \text{if } n > 1$$

and  $H(1) = \Theta(1)$ . Applying the theorem for “subtractive” recurrences with  $a = 2$  and  $c = 1$  we get  $H(n) = \Theta(2^n)$ . Indeed, it can be easily shown that exactly  $M_n = 2^n - 1$  single moves are necessary (and sufficient) to move the  $n$  disks from A to C.

# Analysis of Recursive Algorithms

## Example (Powers)

Given three positive integers  $x$ ,  $y$  and  $m > 1$ , compute  $x^y \bmod m$ .

- For any  $y_1, y_2$  such that  $y_1 + y_2 = y$ ,

$$x^y \bmod m = ((x^{y_1} \bmod m) \cdot (x^{y_2} \bmod m)) \bmod m,$$

that is, we can take  $\bmod m$  in intermediate steps to avoid dealing with very large numbers

- If we compute  $x^y$ , either iteratively or recursively, using the identity  $x^y = x \cdot x^{y-1}$  for  $y > 0$ , we end up with an algorithm making  $\Theta(y)$  products  $\Rightarrow$  exponential in the size of the input (we need  $\approx \log_2(x) + \log_2(y) + \log_2 m$  bits)

# Analysis of Recursive Algorithms

## Example (Powers)

Given three positive integers  $x$ ,  $y$  and  $m > 1$ , compute  $x^y \bmod m$ .

- For any  $y_1, y_2$  such that  $y_1 + y_2 = y$ ,

$$x^y \bmod m = ((x^{y_1} \bmod m) \cdot (x^{y_2} \bmod m)) \bmod m,$$

that is, we can take  $\bmod m$  in intermediate steps to avoid dealing with very large numbers

- If we compute  $x^y$ , either iteratively or recursively, using the identity  $x^y = x \cdot x^{y-1}$  for  $y > 0$ , we end up with an algorithm making  $\Theta(y)$  products  $\Rightarrow$  exponential in the size of the input (we need  $\approx \log_2(x) + \log_2(y) + \log_2 m$  bits)

# Analysis of Recursive Algorithms

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

## Example (Powers (cont'd))

The cost  $P(y)$  (measured as the number of arithmetical operations) of `power` satisfies the following recurrence<sup>a</sup>

$$P(y) = P(y/2) + \Theta(1),$$

and  $P(0) = 0$ ; we can solve the recurrence using the theorem for “divisive” recurrences with  $k = 0$ ,  $a = 1$  and  $b = 2$ ; since  $\alpha = \log_2 1 = 0 = k$  the solution is  $P(y) = \Theta(\log y) \Rightarrow$  linear number of products in the size of the input

---

<sup>a</sup>Ceilings and floors can be safely ignored; the actual recurrence is  $P(y) = P(\lceil y/2 \rceil) + \Theta(1)$ .

# Part II

## Divide and Conquer

5 Foundations

6 Mathematical Algorithms

7 Searching & Sorting  
• The Continuous Master Theorem

8 Selection

# Introduction

The basic principle of **divide and conquer** (in Catalan, *dividir per conquerir*, in Spanish *divide y vencerás*) is very simple:

- 1 If the given input is simple enough, find the corresponding solution using some direct straightforward method.
- 2 Otherwise *divide* the given input  $x$  into subinstances  $x_1, \dots, x_k$  of the problem, and solve it, independently and recursively, for each of the subinstances.
- 3 The solutions thus obtained  $y_1, \dots, y_k$  are *combined* to get the solution corresponding to the original input  $x$ .

# Divide and Conquer: A Template

- The template for divide and conquer algorithms in pseudocode looks like:

```
procedure DIVIDE_AND_CONQUER( $x$ )
    if  $x$  is simple then
        return DIRECT_SOLUTION( $x$ )
    else
         $\langle x_1, x_2, \dots, x_k \rangle := \text{DIVIDE}(x)$ 
        for  $i := 1$  to  $k$  do
             $y_i := \text{DIVIDE\_AND\_CONQUER}(x_i)$ 
        end for
        return COMBINE( $y_1, y_2, \dots, y_k$ )
    end if
end procedure
```

- Sometimes, when  $k = 1$ , the algorithm is called a *reduction algorithm*.

## Divide and Conquer: Properties

Divide and conquer algorithms share some common traits:

- No subinstance of the problem is solved more than once
- The size of the subinstances is, at least on average, a fraction of the size of the original input; that is, if  $x$  is of size  $n$  then the average size of any  $x_i$  is  $n/c_i$  for some  $c_i > 1$ . Very often the condition holds not only on average, but surely.

# Cost of Divide and Conquer Algorithms

For many Divide and Conquer algorithms the cost satisfies a recurrence of the form

$$T(n) = \begin{cases} f(n) + a \cdot T(n/b) & \text{if } n > n_0, \\ g(n) & \text{if } n \leq n_0. \end{cases}$$

This is the case when a direct solution is returned whenever the size of the input is  $n \leq n_0$  for some  $n_0$ , and otherwise the original input is divided into subinstances of size (roughly)  $n/b$  each, and  $a$  recursive calls are made. The cost of computing the direct solution is  $g(n)$ , and the cost of the *divide* and *combine* steps is  $f(n)$ . In most cases,  $a = b$ , but there are also many examples of divide and conquer algorithms where this is not true.

# Cost of Divide and Conquer Algorithms

Recall:

## Theorem

If  $f(n) = \Theta(n^k)$  and

$$T(n) = \begin{cases} f(n) + a \cdot T(n/b) & \text{if } n > n_0, \\ g(n) & \text{if } n \leq n_0, \end{cases}$$

with constants  $a \geq 1$  and  $b > 1$ , then

$$T(n) = \begin{cases} \Theta(f(n)) & \text{if } \alpha < k, \\ \Theta(f(n) \log n) & \text{if } \alpha = k \\ \Theta(n^\alpha) & \text{if } \alpha > k, \end{cases}$$

where  $\alpha = \log_b a$ .

# Cost of Divide and Conquer Algorithms

Proof (sketch): We will assume that  $n = n_0 \cdot b^j$ . By repeatedly applying the recurrence

$$\begin{aligned} T(n) &= f(n) + a \cdot T(n/b) \\ &= f(n) + a \cdot f(n/b) + a^2 \cdot T(n/b^2) \\ &= \dots \\ &= f(n) + a \cdot f(n/b) + \dots + a^{j-1} \cdot f(n/b^{j-1}) \\ &\quad + a^j \cdot T(n/b^j) \\ &= \sum_{0 \leq i < j} a^i \cdot f(n/b^i) + a^j \cdot T(n_0) \\ &= \Theta\left(\sum_{0 \leq i < j} a^i \left(\frac{n}{b^i}\right)^k\right) + a^j \cdot g(n_0) \\ &= \Theta\left(n^k \cdot \sum_{0 \leq i < j} \left(\frac{a}{b^k}\right)^i\right) + a^j \cdot g(n_0). \end{aligned}$$

# Cost of Divide and Conquer Algorithms

Since  $g(n_0)$  is a constant, the second term is  $\Theta(n^\alpha)$ :

$$\begin{aligned}a^j &= a^{\log_b(n/n_0)} = a^{\log_b n} \cdot a^{-\log_b n_0} \\&= \Theta(b^{\log_b a \cdot \log_b n}) \\&= \Theta(n^{\log_b a}) = \Theta(n^\alpha).\end{aligned}$$

We have thus three different cases to consider, according to  $a/b^k$  being less, equal or greater than 1. Equivalently, since  $\alpha = \log_b a$ , the three cases correspond to  $\alpha$  being greater, equal or less than  $k$ . If  $k > \alpha$  ( $\equiv a/b^k < 1$ ) then the sum of the first term is a geometric series with ratio less than 1 and its value is upper bounded by a constant; therefore the first term is  $\Theta(n^k)$ . Since  $k > \alpha$ , the first term is dominant and  $T(n) = \Theta(n^k) = \Theta(f(n))$ .

# Cost of Divide and Conquer Algorithms

If  $\alpha = k$  then  $a/b^k = 1$  and the sum adds to  $j$ ; as  $j = \log_b(n/n_0) = \Theta(\log n)$ , we conclude that  $T(n) = \Theta(n^k \cdot \log n) = \Theta(f(n) \cdot \log n)$ . Last, but not least, if  $k < \alpha$  then  $a/b^k > 1$  and the sum is

$$\begin{aligned} \sum_{0 \leq i < j} \left( \frac{a}{b^k} \right)^i &= \frac{(a/b^k)^j - 1}{a/b^k - 1} = \Theta \left( \left( \frac{a}{b^k} \right)^j \right) \\ &= \Theta \left( \frac{n^\alpha}{n^k} \right) = \Theta(n^{\alpha-k}). \end{aligned}$$

In this case the second term dominates and  $T(n) = \Theta(n^\alpha)$ .

# Part II

## Divide and Conquer

- 5 Foundations
- 6 Mathematical Algorithms
  - Karatsuba's Algorithm
  - Strassen's Algorithm
- 7 Searching & Sorting
  - The Continuous Master Theorem
- 8 Selection

# Part II

## Divide and Conquer

### 5 Foundations

### 6 Mathematical Algorithms

- Karatsuba's Algorithm
- Strassen's Algorithm

### 7 Searching & Sorting

- Binary Search
- Mergesort
- QuickSort

### 8 Selection

- Quickselect
- Rivest-Floyd's Worst-Case Linear Time Selection

## Karatsuba's Algorithm

The traditional algorithm for integer multiplication has cost  $\Theta(n^2)$  to multiply two integers of  $n$  bits each, since it is equivalent to performing  $n$  additions of  $\Theta(n)$ -bit numbers and each such addition has cost  $\Theta(n)$ .

$$\begin{aligned} 231 \times 659 &= 9 \times 231 + 5 \times 231 \times 10 \\ &\quad + 6 \times 231 \times 100 \\ &= 2079 + 11550 + 138600 \\ &= 152229 \end{aligned}$$

# Karatsuba's Algorithm

The Russian multiplication algorithm has quadratic cost too.

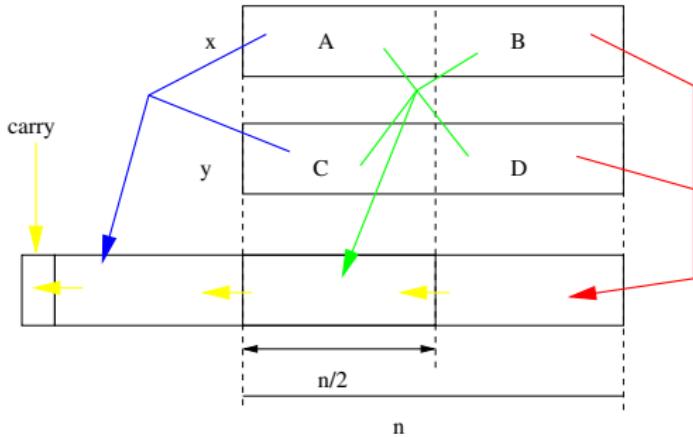
```
 $z := 0$ 
▷  $x = X \wedge y = Y$ 
while  $x \neq 0$  do
    ▷ Inv:  $z + x \cdot y = X \cdot Y$ 
    if  $x$  is even then
         $x := x \text{ div } 2; y := 2 \cdot y$ 
    else
         $x := x - 1; z := z + y$ 
    end if
end while
▷  $z = X \cdot Y$ 
```

# Karatsuba's Algorithm

```
// Pre:  $x = X \wedge y = Y$ 
z = 0;
while (x != 0)
// Inv:  $z + x \cdot y = X \cdot Y$ 
    if (x % 2 == 0) { x /= 2; y *= 2; }
    else { x--; z += y; }
// Post:  $z = X \cdot Y$ 
```

# Karatsuba's Algorithm

Assume both  $x$  and  $y$  are  $n$ -bit integers for  $n = 2^k$  (if  $n$  is not a power of 2 or they are not both of  $n$  bits, they can be padded with 0s to achieve this).  
An idea that does not work:



$$\begin{aligned}x \cdot y &= (A \cdot 2^{n/2} + B) \cdot (C \cdot 2^{n/2} + D) \\&= A \cdot C \cdot 2^n + (A \cdot D + B \cdot C) \cdot 2^{n/2} + B \cdot D.\end{aligned}$$

## Karatsuba's Algorithm

Once we decompose  $x$  as  $x = \langle A, B \rangle$  and  $y = \langle C, D \rangle$ , we compute the 4 products  $A \cdot B$ ,  $A \cdot D$ ,  $B \cdot C$  and  $B \cdot D$ , and then combine these intermediate results by means of sums and *shifts*. The cost of the sums and the shifts is  $\Theta(n)$ . Hence, the cost of this divide-and-conquer multiplication algorithm satisfies the recurrence

$$M(n) = \Theta(n) + 4 \cdot M(n/2), \quad n > 1$$

and the solution is  $M(n) = \Theta(n^2)$ , since  $k = 1$ ,  $a = 4$  and  $b = 2$  ( $k = 1 < \alpha = \log_2 4 = 2$ ).

## Karatsuba's Algorithm

Karatsuba's algorithm (1962) computes the product of  $x$  and  $y$  dividing the two numbers as before but only 3 products need to be recursively computed:

$$U = A \cdot C$$

$$V = B \cdot D$$

$$W = (A + B) \cdot (C + D)$$

The final result is obtained combining the intermediate results above as follows:

$$x \cdot y = U \cdot 2^n + (W - (U + V)) \cdot 2^{n/2} + V.$$

## Karatsuba's Algorithm

The algorithm requires 6 sums (one is a subtraction) instead of the 3 sums needed in the previous algorithm. But the non-recursive cost of Karatsuba's algorithm is still linear, and since the number of recursive calls is smaller, the overall cost is asymptotically faster:

$$M(n) = \Theta(n) + 3 \cdot M(n/2)$$

$$M(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.5849625\dots})$$

The hidden constant in  $\Theta(n^{1.5849625\dots})$  is large and the advantage of Karatsuba's algorithm w.r.t. to the basic quadratic algorithm is not noticeable unless  $n$  is large enough (typically we see no improvement until  $n$  is about 200 to 250 bits)

# Karatsuba's Algorithm

**procedure** **MULT**( $x, y, i, j, i', j'$ )

**Require:**  $1 \leq i \leq j \leq n, 1 \leq i' \leq j' \leq n, j' - i' = j - i$

**Ensure:** Returns the product of  $x[i..j]$  and  $y[i'..j']$

$n := j - i + 1$

**if**  $n < M$  **then**

Use a basic algorithm (hardware?) to compute the product

**else**

$m := (i + j) \text{ div } 2; m' := (i' + j') \text{ div } 2$

$U := \text{MULT}(x, y, m + 1, j, m' + 1, j')$

$V := \text{MULT}(x, y, i, m, i', m')$

$U.\text{LEFT\_SHIFT}(n)$

$W_1 := x[i..m] + y[i'..m']; W_2 := x[m + 1..j] + y[m' + 1..j']$

Pad  $W_1$  or  $W_2$  if necessary

$W := \text{MULT}(W_1, W_2, \dots)$

$W := W - (U + V)$

$W.\text{LEFT\_SHIFT}(n \text{ div } 2)$

**return**  $U + W + V$

**end if**

**end procedure**

# Part II

## Divide and Conquer

### 5 Foundations

### 6 Mathematical Algorithms

- Karatsuba's Algorithm
- Strassen's Algorithm

### 7 Searching & Sorting

- Binary Search
- Mergesort
- QuickSort

### 8 Selection

- Quickselect
- Rivest-Floyd's Worst-Case Linear Time Selection

# Strassen's Algorithm

The multiplication of two square matrices  $n \times n$  has cost  $\Theta(n^3)$  using the straightforward definition of matrix product:

```
for i := 1 to n do
    for j := 1 to n do
        C[i, j] := 0
        for k := 1 to n do
            C[i, j] := C[i, j] + A[i, k] * B[k, j]
        end for
    end for
end for
```

## Strassen's Algorithm

To apply the divide-and-conquer scheme here, we divide the original matrices into four blocks each (a block is a submatrix  $\frac{n}{2} \times \frac{n}{2}$ )

$$\left( \begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right) \cdot \left( \begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right) = \left( \begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right)$$

The rule for matrix multiplication carries over to blocks: thus

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \text{ etc.}$$

## Strassen's Algorithm

Each block in  $C$  requires two block multiplications and a sum (the cost of the sum is  $\Theta(n^2)$ ). The cost of a naïve divide-and-conquer algorithm is still cubic, since the recurrence is

$$M(n) = \Theta(n^2) + 8 \cdot M(n/2),$$

and the solution is  $M(n) = \Theta(n^3)$ .

To achieve a more efficient solution we need to reduce the number of recursive calls, like in Karatsuba's algorithm.

## Strassen's Algorithm

Strassen (1969) proposed such a way<sup>1</sup>. One of the things that make this derivation long and complicated is the fact that matrix product is not commutative.

We have to obtain the following 7 blocks (matrices  $n/2 \times n/2$ ), using 7 products (recursively) and 14 matrix additions/subtractions

$$M_1 = (A_{21} + A_{22} - A_{11}) \cdot (B_{11} + B_{22} - B_{12})$$

$$M_2 = A_{11} \cdot B_{11}$$

$$M_3 = A_{12} \cdot B_{21}$$

$$M_4 = (A_{11} - A_{21}) \cdot (B_{22} - B_{12})$$

$$M_5 = (A_{21} + A_{22}) \cdot (B_{12} - B_{11})$$

$$M_6 = (A_{12} - A_{21} + A_{11} - A_{22}) \cdot B_{22}$$

$$M_7 = A_{22} \cdot (B_{11} + B_{22} - B_{12} - B_{21})$$

---

<sup>1</sup>Brassard & Bratley's *Fundamentals of Algorithmics* give a detailed description of how to find the set of formulas behind Strassen's algorithm.

## Strassen's Algorithm

With 10 matrix additions/subtractions more, we get the blocks of  $C$ :

$$C_{11} = M_2 + M_3$$

$$C_{12} = M_1 + M_2 + M_5 + M_6$$

$$C_{21} = M_1 + M_2 + M_4 - M_7$$

$$C_{22} = M_1 + M_2 + M_4 + M_5$$

Since the matrix additions/subtractions have cost  $\Theta(n^2)$ , the cost of Strassen's algorithm is given by the recurrence:

$$M(n) = \Theta(n^2) + 7 \cdot M(n/2),$$

where the solution is  $\Theta(n^{\log_2 7}) = \Theta(n^{2.807\dots})$ .

## Strassen's Algorithm

Strassen's algorithm had an enormous impact among theoreticians as it was the first matrix multiplication algorithm with complexity asymptotically smaller than  $n^3$ ; that also had implications for the development of more efficient algorithms to compute inverse matrices and determinants, solving linear systems, etc.

Strassen's algorithm was one of the first examples in which **novel** algorithmic techniques helped to break a seemingly unbreakable barrier. Other examples include Karatsuba's algorithm and the Fast Fourier Transform (FFT), which also belong to the family of divide-and-conquer algorithms.

## Strassen's Algorithm

In the years following Strassen's breakthrough, more efficient matrix multiplication algorithms have been developed. The current record is held by the theoretical superior but impractical algorithm of Coppersmith and Winograd (1986) with cost  $\Theta(n^{2.376\dots})$ .

Strassen's algorithm is not faster in practice than the ordinary matrix multiplication algorithm, unless  $n$  is rather large ( $n \gg 500$ ), since the hidden constants and lower order terms in the cost are quite big and their impact in the cost cannot be disregarded except when  $n$  is large enough.

# Part II

## Divide and Conquer

5 Foundations

6 Mathematical Algorithms

7 Searching & Sorting

- Binary Search
- Mergesort
- QuickSort
- The Continuous Master Theorem

8 Selection

# Part II

## Divide and Conquer

### 5 Foundations

### 6 Mathematical Algorithms

- Karatsuba's Algorithm
- Strassen's Algorithm

### 7 Searching & Sorting

- Binary Search
- Mergesort
- QuickSort

### 8 Selection

- Quickselect
- Rivest-Floyd's Worst-Case Linear Time Selection

# Binary Search

Given an array  $A[1..n]$  with  $n$  elements, in increasing order, and some element  $x$ , find the value  $i$ ,  $0 \leq i \leq n$ , such that  $A[i] \leq x < A[i + 1]$  (with the convention that  $A[0] = -\infty$  and  $A[n + 1] = +\infty$ ).

If the array were empty the answer is simple, since  $x$  is not in the array. But it is clear now that we must work with a generalization of the original problem: given a segment or subarray  $A[\ell + 1..u - 1]$ ,  $0 \leq \ell \leq u \leq n + 1$ , in increasing order, and such that  $A[\ell] \leq x < A[u]$ , find the value  $i$ ,  $\ell \leq i \leq u - 1$ , such that  $A[i] \leq x < A[i + 1]$  (this is a typical example of *parameter embedding*). For the generalization, if  $\ell + 1 = u$ , the segment is empty and we can return  $i = u - 1 = \ell$ .

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

# Binary Search

## Example

We are looking for  $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159
----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----

Search ends at position  $k = 8$ :  $A[k] = 83 \leq x < A[k + 1] = 90$

# Binary Search

▷ Initial call: `BINSEARCH( $A, x, 0, n + 1$ )`

**procedure** `BINSEARCH( $A, x, \ell, u$ )`

**Require:**  $0 \leq \ell < u \leq n + 1, A[\ell] \leq x < A[u]$

**Ensure:** Return  $i$  such that  $A[i] \leq x < A[i + 1]$  and  $\ell \leq i < u$

**if**  $\ell + 1 = u$  **then**

**return**  $\ell$

**else**

$m := (\ell + u) \text{ div } 2$

**if**  $x < A[m]$  **then**

**return** `BINSEARCH( $A, x, \ell, m$ )`

**else**

**return** `BINSEARCH( $A, x, m, u$ )`

**end if**

**end if**

**end procedure**

# Binary Search

```
// Initial call: int p = bsearch(A, x, -1, A.size());  
  
template <class T>  
int bsearch(vector<T>& A, const T& x,  
            int l, int u) {  
    if (l == u + 1)  
        return u;  
    int m = (l + u) / 2;  
    if (x < A[m])  
        return bsearch(A, x, l, m);  
    else  
        return bsearch(A, x, m, u);  
}
```

## Binary Search

Let  $n$  denote the size of the array  $A$  where we perform the search. Neglecting the small difference between the actual size of the subarray in the recursive call and  $n/2$ , the cost of a binary search satisfies the recurrence

$$B(n) = \Theta(1) + B(n/2), \quad n > 0,$$

and  $B(0) = b_0$ , since each call makes one recursive call, either in the “left” or the “right”, half of the current segment. Using Theorem 1,  $k = 0$  and  $\alpha = \log_2 1 = 0$  and thus  $B(n) = \Theta(\log n)$ .

# Part II

## Divide and Conquer

### 5 Foundations

### 6 Mathematical Algorithms

- Karatsuba's Algorithm
- Strassen's Algorithm

### 7 Searching & Sorting

- Binary Search
- Mergesort
- QuickSort

### 8 Selection

- Quickselect
- Rivest-Floyd's Worst-Case Linear Time Selection

## Mergesort

Mergesort is one of the oldest efficient sorting methods ever proposed. Different variants of this method are particularly useful for external memory sorts. Mergesort is also one of the best sorting methods for linked lists. The basic idea is simple: divide the input sequence in two halves, sort each half recursively and obtain the final result *merging* the two sorted sequences of the previous step.

## Mergesort

We will assume here that the input is a simple linked list  $L$  that stores the sequence of elements  $x_1, \dots, x_n$ . Each element is stored in a node of the linked list with two fields: `info` stores the element and `next` is a pointer to the next node in the list (the special value `next = null` indicates that the node is the last in the list). The list  $L$  is, in fact, a pointer to its first node. By  $p \rightarrow f$  we mean the field  $f$  in the object pointed to by pointer  $p$ , like in C or C++.

# Mergesort

**procedure** SPLIT( $L, L', n$ )

**Require:**  $L = [\ell_1, \dots, \ell_m], m \geq n$

**Ensure:**  $L = [\ell_1, \dots, \ell_n], L' = [\ell_{n+1}, \dots, \ell_m]$

$p := L$

**while**  $n > 1$  **do**

$p := p \rightarrow next$

$n := n - 1$

**end while**

$L' := p \rightarrow next; p \rightarrow next := \text{null}$

**end procedure**

# Mergesort

```
procedure MERGESORT( $L$ ,  $n$ )
    if  $n > 1$  then
         $m := n \text{ div } 2$ 
        SPLIT( $L$ ,  $L'$ ,  $m$ )
        MERGESORT( $L$ ,  $m$ )
        MERGESORT( $L'$ ,  $n - m$ )
        ▷ merge the two lists  $L$  and  $L'$ 
         $L := \text{MERGE}(L, L')$ 
    end if
end procedure
```

# Mergesort

```
struct node_list {
    Elem info;
    node_list* next;
};

typedef node_list* List;

void split(List& L, List& L2, int n) {
    node_list* p = L;
    while (n > 1) {
        p = p -> next;
        --n;
    }
    L2 = p -> next; p -> next = NULL;
}

List merge(List& L1, List& L2);

void mergesort(List& L, int n) {
    if (n > 1) {
        int m = n / 2;
        List L2;
        split(L, L2, m);
        mergesort(L, m);
        mergesort(L2, n-m);
        L = merge(L, L2);
    }
}
```

## Mergesort

To develop the `Merge` procedure we reason as follows: if  $L$  or  $L'$  is empty the result of the merging is the other list, that is, the one which is (usually) not empty. When both lists are non-empty, we compare their respective first elements. The smaller of the two must be the first element in the merged list, and the remaining elements are the result of merging (recursively) the tail of the list containing the smallest element with the other list.

# Mergesort

```
procedure MERGE( $L, L'$ )
    if  $L = \text{null}$  then return  $L'$ 
    end if
    if  $L' = \text{null}$  then return  $L$ 
    end if
    if  $L \rightarrow \text{info} \leq L' \rightarrow \text{info}$  then
         $L \rightarrow \text{next} := \text{MERGE}(L \rightarrow \text{next}, L')$ 
        return  $L$ 
    else
         $L' \rightarrow \text{next} := \text{MERGE}(L, L' \rightarrow \text{next})$ 
        return  $L'$ 
    end if
end procedure
```

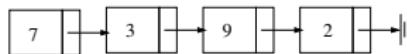
# Mergesort

```
List merge(List L, List L2) {  
    if (L == NULL) return L2;  
    if (L2 == NULL) return L;  
    if (L -> info <= L2 -> info) {  
        L -> next = merge(L -> next, L2);  
        return L;  
    } else { // L -> info > L2 -> info  
        L2 -> next = merge(L, L2 -> next);  
        return L2;  
    }  
}
```

# Mergesort

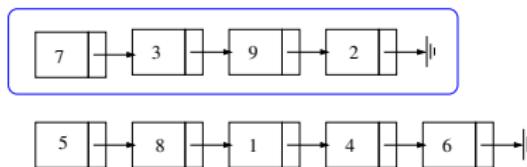


# Mergesort

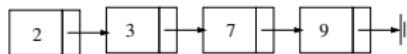


# Mergesort

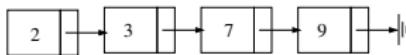
mergesort



# Mergesort



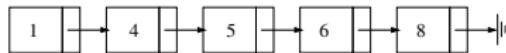
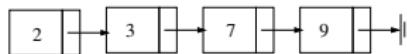
# Mergesort



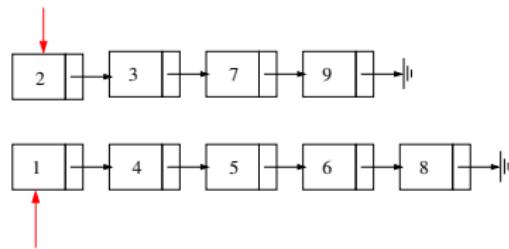
mergesort



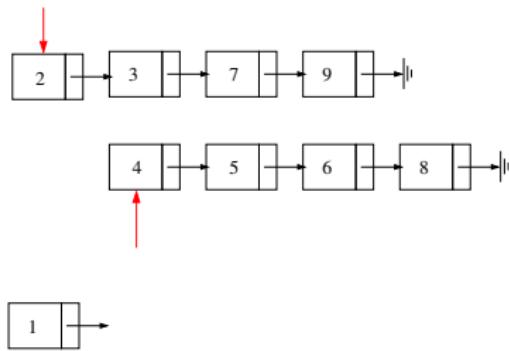
# Mergesort



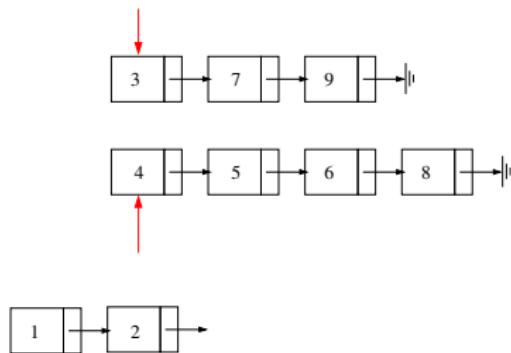
# Mergesort



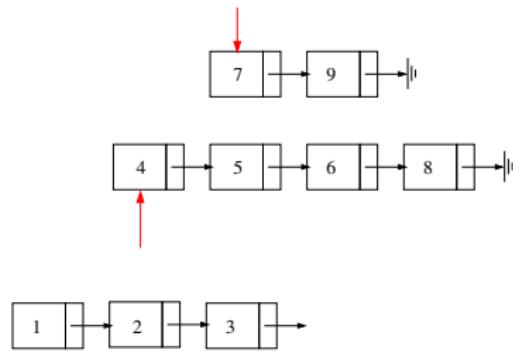
# Mergesort



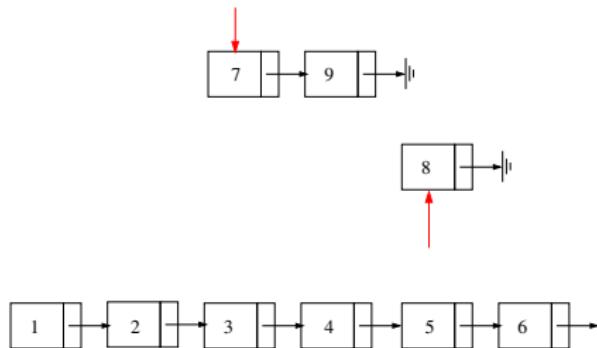
# Mergesort



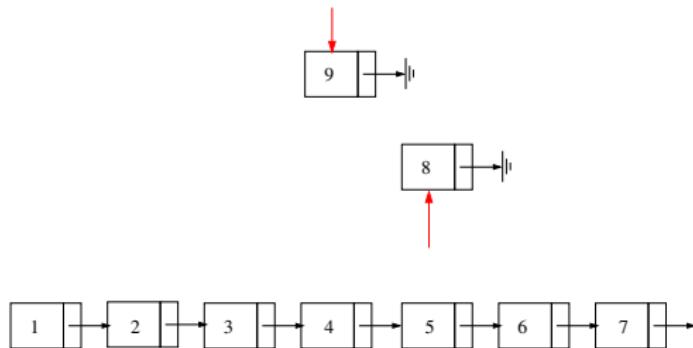
# Mergesort



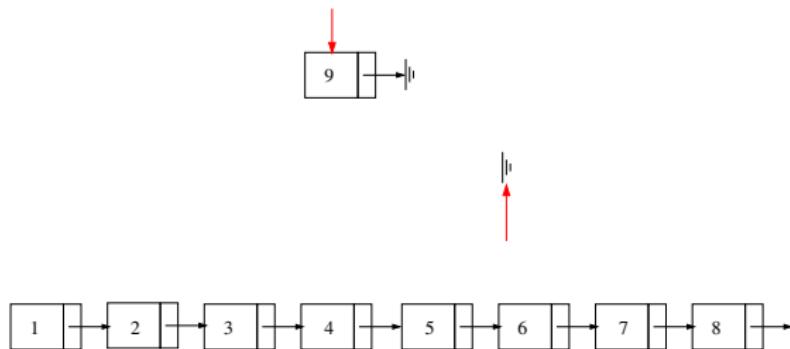
# Mergesort



# Mergesort



# Mergesort



# Mergesort



## Mergesort

Each element of  $L$  and  $L'$  is “visited” exactly once, hence the cost of MERGE is proportional to the sum of the lengths of the lists  $L$  and  $L'$ ; that is, the cost is  $\Theta(n)$ . Since MERGE has *tail recursion* it is quite easy to obtain an iterative version which is slightly more efficient.

Note that if the two inspected elements in  $L$  and  $L'$  are identical, we put first the element coming from  $L$  into the result, which ensures that MERGESORT is a *stable* sorting method.

The cost of MERGESORT is described by the recurrence:

$$\begin{aligned}M(n) &= \Theta(n) + M\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + M\left(\left\lceil\frac{n}{2}\right\rceil\right) \\&= \Theta(n) + 2 \cdot M\left(\frac{n}{2}\right)\end{aligned}$$

hence the solution is  $M(n) = \Theta(n \log n)$ , by applying the second case of Theorem 1.

# Part II

## Divide and Conquer

### 5 Foundations

### 6 Mathematical Algorithms

- Karatsuba's Algorithm
- Strassen's Algorithm

### 7 Searching & Sorting

- Binary Search
- Mergesort
- QuickSort
  - The Continuous Master Theorem

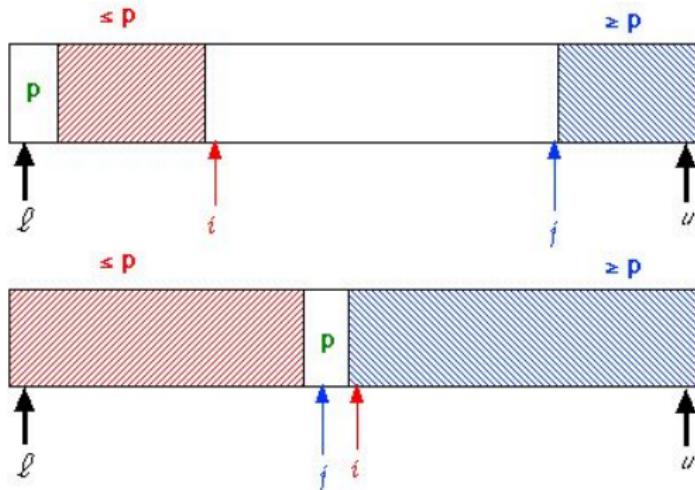
### 8 Selection

- Quickselect
- Rivest-Floyd's Worst-Case Linear Time Selection

# QuickSort

QUICKSORT (Hoare, 1962) is a sorting algorithm using the divide-and-conquer principle too, but contrary to the previous examples, it does not guarantee that the size of each subinstance will be a fraction of the size of the original given instance.

The basis of *Quicksort* is the procedure PARTITION: given an element  $p$ , called the pivot, the (sub)array is rearranged as shown in the picture below.



# QuickSort

PARTITION puts the pivot in its final place. Hence it suffices to recursively sort the subarrays to its left and to its right. While *divide* is simple and *combine* does most of the work in MERGESORT, in QUICKSORT it happens just the contrary: *divide* consists in partitioning the array, and we have nothing to do to *combine* the solutions obtained from the recursive calls.

The QUICKSORT algorithm for a subarray  $A[\ell..u]$  is

```
procedure QUICKSORT( $A, \ell, u$ )
Ensure: Sorts subarray  $A[\ell..u]$ 
  if  $u - \ell + 1 \leq M$  then
    use a simple sorting method, e.g., insertion sort
  else
    PARTITION( $A, \ell, u, k$ )
     $\triangleright A[\ell..k - 1] \leq A[k] \leq A[k + 1..u]$ 
    QUICKSORT( $(A, \ell, k - 1)$ )
    QUICKSORT( $(A, k + 1, u)$ )
  end if
end procedure
```

# QuickSort

Instead of using a simple sorting method each time we reach a subarray with  $M$  or less elements, we can postpone the sorting of small subarrays until the end:

```
QUICKSORT( $A$ , 1,  $A.\text{SIZE}()$ )
INSERTSORT( $A$ , 1,  $A.\text{SIZE}()$ )
```

Since the array is almost sorted upon completion of the call to `QUICKSORT`, the last step using `INSERTSORT` needs only linear time ( $\Theta(n)$ ), where  $n = A.\text{SIZE}()$ .

The typical optimal choice for the cut-off value  $M$  is around 20 to 25.

# QuickSort

```
template <class T>
void partition(vector<T>& v, int l, int u, int& k);

template <class T>
void quicksort(vector<T>& v, int l, int u) {
    if (u - l + 1 > M) {
        int k;
        partition(v, l, u, k);
        quicksort(v, l, k-1);
        quicksort(v, k+1, u);
    }
}

template <class T>
void quicksort(vector<T>& v) {
    quicksort(v, 0, v.size()-1);
    insertion_sort(v, 0, v.size()-1);
}
```

## QuickSort

There are many ways to do the partition; not them all are equally good. Some issues, like repeated elements, have to be dealt with carefully. Bentley & McIlroy (1993) discuss a very efficient partition procedure, which works seamlessly even in the presence of repeated elements. Here, we will examine a basic algorithm, which is reasonably efficient.

We will keep two indices  $i$  and  $j$  such that  $A[\ell + 1..i - 1]$  contains elements less than or equal to the pivot  $p$ , and  $A[j + 1..u]$  contains elements greater than or equal to the pivot  $p$ . The two indices scan the subarray locations,  $i$  from left to right,  $j$  from right to left, until  $A[i] > p$  and  $A[j] < p$ , or until they cross ( $i = j + 1$ ).

# QuickSort

```
procedure PARTITION( $A, \ell, u, k$ )
Require:  $\ell \leq u$ 
Ensure:  $A[\ell..k - 1] \leq A[k] \leq A[k + 1..u]$ 
 $i := \ell + 1; j := u; p := A[\ell]$ 
while  $i < j + 1$  do
    while  $i < j + 1 \wedge A[i] \leq p$  do
         $i := i + 1$ 
    end while
    while  $i < j + 1 \wedge A[j] \geq p$  do
         $j := j - 1$ 
    end while
    if  $i < j + 1$  then
         $A[i] := A[j]$ 
         $i := i + 1; j := j - 1$ 
    end if
end while
 $A[\ell] := A[j]; k := j$ 
end procedure
```

# QuickSort

```
template <class T>
void partition(vector<T>& v, int l, int u, int& k) {
    int i = l;
    int j = u + 1;
    T pv = v[i]; // simple choice for pivot
    do {
        while (v[i] < pv and i < u) ++i;
        while (pv < v[j] and l < j) --j;
        if (i <= j) {
            swap(v[i], v[j]);
            ++i; --j;
        }
    } while (i <= j);
    swap(v[l], v[j]);
    k = j;
}
```

# The Cost of QuickSort

The worst-case cost of QUICKSORT is  $\Theta(n^2)$ , hence not very attractive. But it only occurs if all or most recursive calls one of the subarrays contains very few elements and the other contains almost all. That would happen if we systematically choose the first element of the current subarray as the pivot and the array is already sorted!

The cost of the partition is  $\Theta(n)$  and we would have then

$$\begin{aligned} Q(n) &= \Theta(n) + Q(n - 1) + Q(0) \\ &= \Theta(n) + Q(n - 1) = \Theta(n) + \Theta(n - 1) + Q(n - 2) \\ &= \dots = \sum_{i=0}^n \Theta(i) = \Theta\left(\sum_{0 \leq i \leq n} i\right) \\ &= \Theta(n^2). \end{aligned}$$

## The Cost of QuickSort

However, on average, there will be a fraction of the elements that are less than the pivot (and will be to its left) and a fraction of elements that are greater than the pivot (and will be to its right). It is for this reason that **QUICKSORT** belongs to the family of divide-and-conquer algorithms, and indeed it has a good average-case complexity.

# The Cost of QuickSort

To analyze the performance of `QUICKSORT` it only matters the relative order of the elements to be sorted, hence we can safely assume that the input is a permutation of the elements 1 to  $n$ . Furthermore, we can concentrate in the number of comparisons between the elements since the total cost will be proportional to that number of comparisons.

# The Cost of QuickSort

Let us assume that all  $n!$  possible input permutations are equally likely and let  $q_n$  be the expected number of comparisons to sort the  $n$  elements. Then

$$\begin{aligned} q_n &= \sum_{1 \leq j \leq n} \mathbb{E}[\# \text{ compar.} \mid \text{pivot is the } j\text{-th}] \times \Pr\{\text{pivot is the } j\text{-th}\} \\ &= \sum_{1 \leq j \leq n} (n - 1 + q_{j-1} + q_{n-j}) \times \frac{1}{n} \\ &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq j \leq n} (q_{j-1} + q_{n-j}) \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq j < n} q_j \end{aligned}$$

To solve this recurrence we can use the theorems that we have studied before; we will use a generalization known as the *continuous master theorem* (CMT).

# The Continuous Master Theorem

CMT considers divide-and-conquer recurrences of the following type:

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j, \quad n \geq n_0$$

for some positive integer  $n_0$ , a function  $t_n$ , called the *toll function*, and a sequence of *weights*  $\omega_{n,j} \geq 0$ . The weights must satisfy two conditions:

- 1  $W_n = \sum_{0 \leq j < n} \omega_{n,j} \geq 1$  (at least one recursive call).
- 2  $Z_n = \sum_{0 \leq j < n} \frac{j}{n} \cdot \frac{\omega_{n,j}}{W_n} < 1$  (the size of the subinstances is a fraction of the size of the original instance).

The next step is to find a *shape function*  $\omega(z)$ , a continuous function approximating the discrete weights  $\omega_{n,j}$ .

# The Continuous Master Theorem

## Definition

Given the sequence of weights  $\omega_{n,j}$ ,  $\omega(z)$  is a shape function for that set of weights if

- ①  $\int_0^1 \omega(z) dz \geq 1$
- ② there exists a constant  $\rho > 0$  such that

$$\sum_{0 \leq j < n} \left| \omega_{n,j} - \int_{j/n}^{(j+1)/n} \omega(z) dz \right| = \mathcal{O}(n^{-\rho})$$

A simple trick that works very often, to obtain a convenient shape function is to substitute  $j$  by  $z \cdot n$  in  $\omega_{n,j}$ , multiply by  $n$  and take the limit for  $n \rightarrow \infty$ .

$$\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n}$$

# The Continuous Master Theorem

The extension of discrete functions to functions in the real domain is immediate, e.g.,  $j^2 \rightarrow z^2$ . For binomial numbers one might use the approximation

$$\binom{z \cdot n}{k} \sim \frac{(z \cdot n)^k}{k!}.$$

The continuation of factorials to the real numbers is given by Euler's Gamma function  $\Gamma(z)$  and that of harmonic numbers by  $\Psi$  function:  $\Psi(z) = \frac{d \ln \Gamma(z)}{dz}$ . For instance, in quicksort's recurrence all Wright numbers are equal:  $\omega_{n,j} = \frac{2}{n}$ . Hence a simple valid shape function is  $\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n} = 2$ .

# The Continuous Master Theorem

## Theorem (Roura, 1997)

Let  $F_n$  satisfy the recurrence

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j,$$

with  $t_n = \Theta(n^a (\log n)^b)$ , for some constants  $a \geq 0$  and  $b > -1$ , and let  $\omega(z)$  be a shape function for the weights  $\omega_{n,j}$ . Let  $\mathcal{H} = 1 - \int_0^1 \omega(z) z^a dz$  and  $\mathcal{H}' = -(b+1) \int_0^1 \omega(z) z^a \ln z dz$ . Then

$$F_n = \begin{cases} \frac{t_n}{\mathcal{H}} + o(t_n) & \text{if } \mathcal{H} > 0, \\ \frac{t_n}{\mathcal{H}'} \ln n + o(t_n \log n) & \text{if } \mathcal{H} = 0 \text{ and } \mathcal{H}' \neq 0, \\ \Theta(n^\alpha) & \text{if } \mathcal{H} < 0, \end{cases}$$

where  $x = \alpha$  is the unique non-negative solution of the equation

$$1 - \int_0^1 \omega(z) z^x dz = 0.$$

# Solving QuickSort's Recurrence

We apply CMT to quicksort's recurrence with the set of weights  $\omega_{n,j} = 2/n$  and toll function  $t_n = n - 1$ . As we have already seen, we can take  $\omega(z) = 2$ , and the CMT applies with  $a = 1$  and  $b = 0$ . All necessary conditions to apply CMT are met. Then we compute

$$\mathcal{H} = 1 - \int_0^1 2z \, dz = 1 - z^2 \Big|_{z=0}^{z=1} = 0,$$

hence we will have to apply CMT's second case and compute

$$\mathcal{H}' = - \int_0^1 2z \ln z \, dz = \frac{z^2}{2} - z^2 \ln z \Big|_{z=0}^{z=1} = \frac{1}{2}.$$

Finally,

$$\begin{aligned} q_n &= \frac{n \ln n}{1/2} + o(n \log n) = 2n \ln n + o(n \log n) \\ &= 1.386 \dots n \log_2 n + o(n \log n). \end{aligned}$$

# Part II

## Divide and Conquer

5 Foundations

6 Mathematical Algorithms

7 Searching & Sorting

- The Continuous Master Theorem

8 Selection

- Quickselect
- Rivest-Floyd's Worst-Case Linear Time Selection

# Part II

## Divide and Conquer

### 5 Foundations

### 6 Mathematical Algorithms

- Karatsuba's Algorithm
- Strassen's Algorithm

### 7 Searching & Sorting

- Binary Search
- Mergesort
- QuickSort

### 8 Selection

- Quickselect
- Rivest-Floyd's Worst-Case Linear Time Selection

# QuickSelect

The **selection** problem is to find the  $j$ -th smallest element in a given set of  $n$  elements. More specifically, given an array  $A[1..n]$  of size  $n > 0$  and a **rank**  $j$ ,  $1 \leq j \leq n$ , the selection problem is to find the  $j$ -th element of  $A$  if it were in ascending order.

For  $j = 1$  we want to find the minimum, for  $j = n$  we want to find the maximum, for  $j = \lceil n/2 \rceil$  we are looking for the median, etc.

## QuickSelect

The problem can be trivially but inefficiently (because it implies doing much more work than needed!) solved with cost  $\Theta(n \log n)$  sorting the array. Another solution keeps an unsorted table of the  $j$  smallest elements seen so far while scanning the array from left to right; it has cost  $\Theta(j \cdot n)$ , and using clever data structures the cost can be improved to  $\Theta(n \log j)$ . This is not a real improvement with respect the first trivial solution if  $j = \Theta(n)$ . QUICKSELECT (Hoare, 1962), also known as FIND and as *one-sided* QUICKSORT, is a variant of QUICKSORT adapted to select the  $j$ -th smallest element out of  $n$ .

## QuickSelect

Assume we partition the subarray  $A[\ell..u]$ , that contains the elements of ranks  $\ell$  to  $u$ ,  $\ell \leq j \leq u$ , with respect some pivot  $p$ . Once the partition finishes, suppose that the pivot ends at position  $k$ .

Then  $A[\ell..k - 1]$  contains the elements of ranks  $\ell$  to  $(k - 1)$  in  $A$  and  $A[k + 1..u]$  contains the elements of ranks  $(k + 1)$  to  $u$ . If  $j = k$  we are done since we have found the sought element. If  $j < k$  then we need to recursively continue in the left subarray  $A[\ell..k - 1]$ , whereas if  $j > k$  then the sought element must be located in the right subarray  $A[k + 1..u]$ .

# QuickSelect

## Example

We are looking the fourth element ( $j = 4$ ) out of  $n = 15$  elements

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

# QuickSelect

## Example

We are looking the fourth element ( $j = 4$ ) out of  $n = 15$  elements

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

# QuickSelect

## Example

We are looking the fourth element ( $j = 4$ ) out of  $n = 15$  elements

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

pivot ends at position  $k = 9 > j$

# QuickSelect

## Example

We are looking the fourth element ( $j = 4$ ) out of  $n = 15$  elements

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

# QuickSelect

## Example

We are looking the fourth element ( $j = 4$ ) out of  $n = 15$  elements

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

pivot ends at position  $k = 6 > j$

# QuickSelect

## Example

We are looking the fourth element ( $j = 4$ ) out of  $n = 15$  elements

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

# QuickSelect

## Example

We are looking the fourth element ( $j = 4$ ) out of  $n = 15$  elements

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

pivot ends at position  $k = 4 = j \Rightarrow \text{DONE!}$

# QuickSelect

**procedure** QUICKSELECT( $A, \ell, j, u$ )

**Ensure:** Returns the  $(j + 1 - \ell)$ -th smallest element in  $A[\ell..u]$ ,

$\ell \leq j \leq u$

**if**  $\ell = u$  **then**

**return**  $A[\ell]$

**end if**

PARTITION( $A, \ell, u, k$ )

**if**  $j = k$  **then**

**return**  $A[k]$

**end if**

**if**  $j < k$  **then**

**return** QUICKSELECT( $A, \ell, j, k - 1$ )

**else**

**return** QUICKSELECT( $A, k + 1, j, u$ )

**end if**

**end procedure**

# QuickSelect

Our implementation of QUICKSELECT uses **tail recursion**, it is hence straightforward to derive an efficient iterative solution that needs no auxiliary memory space.

In the worst-case, the cost of QUICKSELECT is  $\Theta(n^2)$ . However, its average cost is  $\Theta(n)$ , with the proportionality constant depending on the ratio  $j/n$ .

Knuth (1971) proved that  $C_n^{(j)}$ , the expected number of comparisons to find the smallest  $j$ -th element among  $n$  is:

$$\begin{aligned} C_n^{(j)} = & 2 \left( (n+1)H_n - (n+3-j)H_{n+1-j} \right. \\ & \left. - (j+2)H_j + n+3 \right) \end{aligned}$$

The maximum average cost corresponds to finding the median ( $j = \lfloor n/2 \rfloor$ ); then we have

$$C_n^{(\lfloor n/2 \rfloor)} = 2(\ln 2 + 1)n + o(n).$$

## QuickSelect

Let us now consider the analysis of the expected cost  $C_n$  when  $j$  takes any value between 1 and  $n$  with identical probability. Then

$$C_n = n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \mathbb{E}[\text{remaining number of comp.} \mid \text{pivot is the } k\text{-th element}] ,$$

as the pivot will be the  $k$ -th smallest element with probability  $1/n$  for all  $k$ ,  $1 \leq k \leq n$ .

## QuickSelect

The probability that  $j = k$  is  $1/n$ , then no more comparisons are needed since we would be done. The probability that  $j < k$  is  $(k - 1)/n$ , then we will have to make  $C_{k-1}$  comparisons. Similarly, with probability  $(n - k)/n$  we have  $j > k$  and we will then make  $C_{n-k}$  comparisons. Thus

$$\begin{aligned} C_n &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \frac{k-1}{n} C_{k-1} + \frac{n-k}{n} C_{n-k} \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq k < n} \frac{k}{n} C_k. \end{aligned}$$

Applying the CMT with the shape function

$$\lim_{n \rightarrow \infty} n \cdot \frac{2}{n} \frac{z \cdot n}{n} = 2z$$

we obtain  $\mathcal{H} = 1 - \int_0^1 2z^2 dz = 1/3 > 0$  and  $C_n = 3n + o(n)$ .

# Part II

## Divide and Conquer

### 5 Foundations

### 6 Mathematical Algorithms

- Karatsuba's Algorithm
- Strassen's Algorithm

### 7 Searching & Sorting

- Binary Search
- Mergesort
- QuickSort

### 8 Selection

- Quickselect
- Rivest-Floyd's Worst-Case Linear Time Selection

## Rivest-Floyd's Worst-Case Linear Time Selection

We can design a selection algorithm with linear worst-case cost; we need only to guarantee that the pivot that we choose in each recursive step will divide the array into two subarrays such that their respective sizes are a fraction of the original size  $n$ . And we must pick such a pivot with cost  $\mathcal{O}(n)$ . Then, in the worst-case, the cost of the algorithm will be

$$C(n) = \mathcal{O}(n) + C(p \cdot n),$$

for some  $p < 1$ . Since  $\log_{1/p} 1 = 0 < 1$  we conclude that  $C(n) = \mathcal{O}(n)$ . On the other hand, it is obvious that  $C(n) = \Omega(n)$ , hence  $C(n) = \Theta(n)$ . This is the idea behind the selection algorithm proposed by Rivest and Floyd (1970).

## Rivest-Floyd's Worst-Case Linear Time Selection

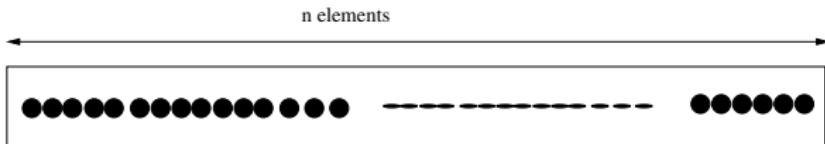
The only difference between Hoare's QUICKSELECT algorithm and Rivest and Floyd's algorithm is the way to choose the pivot of each recursive step. Rivest and Floyd's algorithm picks a “good” pivot using the selection algorithm recursively!

## Rivest-Floyd's Worst-Case Linear Time Selection

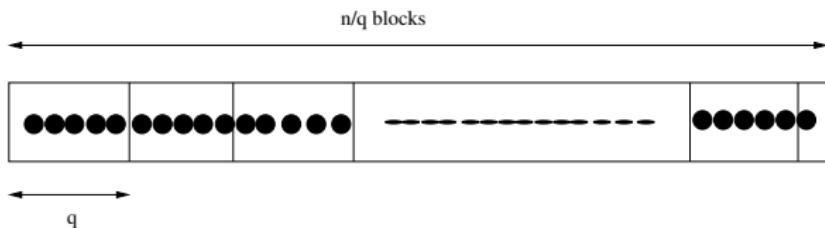
In particular, the algorithm will pick the so-called **pseudo-median** as the pivot.

- 1 The array  $A[\ell..u]$  of size  $n = u - \ell + 1$  is subdivided into blocks of  $q$  elements (except possibly the last block which might contain  $< q$  elements) for some odd constant  $q$ . For each block we obtain the median of the  $q$  elements.
- 2 The selection algorithm is recursively applied to find the median of the  $\lceil n/q \rceil$  medians that were obtained in the previous step.
- 3 The median of the medians (*pseudo-median*) is used as the pivot to partition the original subarray, and a recursive call is made in the appropriate subarray, either left or right to the pivot.

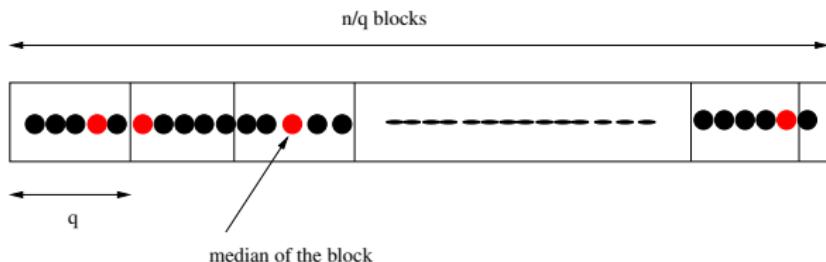
# Rivest-Floyd's Worst-Case Linear Time Selection



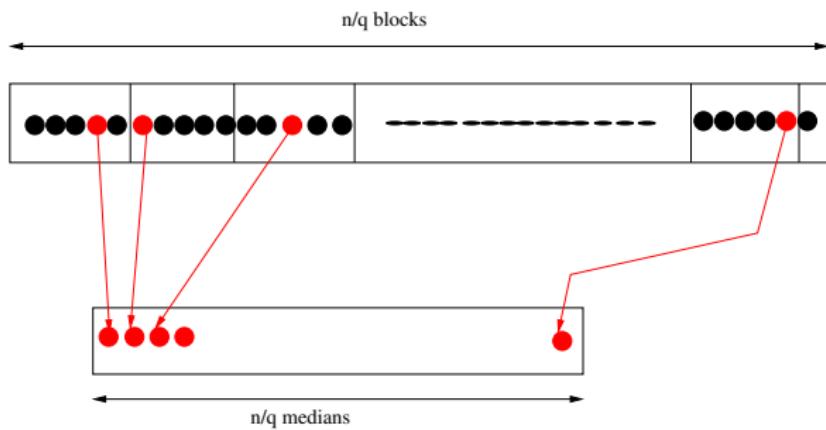
# Rivest-Floyd's Worst-Case Linear Time Selection



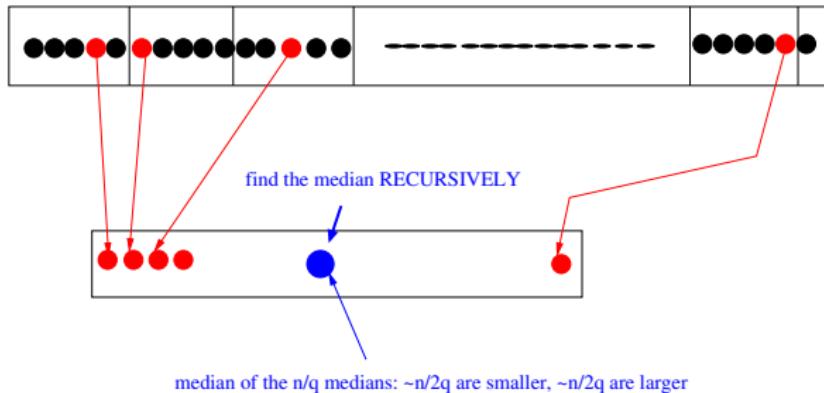
# Rivest-Floyd's Worst-Case Linear Time Selection



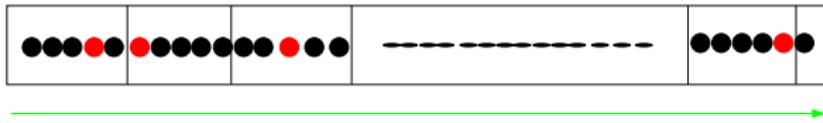
# Rivest-Floyd's Worst-Case Linear Time Selection



# Rivest-Floyd's Worst-Case Linear Time Selection



# Rivest-Floyd's Worst-Case Linear Time Selection



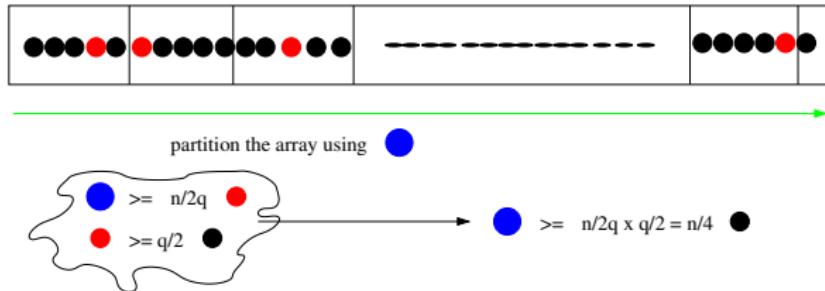
partition the array using



●  $\geq n/2q$  ●

●  $\geq q/2$  ●

# Rivest-Floyd's Worst-Case Linear Time Selection



## Rivest-Floyd's Worst-Case Linear Time Selection

The cost of the first step (computing the medians of the blocks) is  $\Theta(n)$  since the cost of finding the median of each block is  $\Theta(1)$  and there are  $\approx n/q$  blocks. The recursive call to find the median of the medians has cost  $C(n/q)$  since the input consists of the  $\lceil n/q \rceil$  medians of the blocks.

Since the pivot that we pick is the median of  $\lceil n/q \rceil$  medians (this is why we call it the pseudo-median of the  $n$  elements), we have the guarantee that at least  $n/2q \cdot q/2 = n/4$  elements are smaller than the pivot, and similarly, at least  $n/4$  elements are larger. Thus, in the worst-case, after the partition with respect to the pivot (which costs  $\Theta(n)$ ) we will have to proceed recursively into a subarray with at most  $3n/4$  elements.

## Rivest-Floyd's Worst-Case Linear Time Selection

Putting everything together, the worst-case cost of the algorithm satisfies

$$C(n) = \mathcal{O}(n) + C(n/q) + C(3n/4),$$

where the  $\mathcal{O}(n)$  term collects the cost of finding the medians of the blocks and the cost of partitioning the original array around the pivot. The solution to the recurrence above (one can use the *Discrete Master Theorem*) is

$C(n) = \mathcal{O}(n)$  if

$$\frac{3}{4} + \frac{1}{q} < 1.$$

For instance  $q = 3$  does not work, but  $q = 5$ , which was the value originally proposed by Rivest and Floyd, does.

# Part III

## Dictionaries

- 9 Binary Search Trees
- 10 Height-balanced Binary Search Trees: AVLs
- 11 Hash Tables

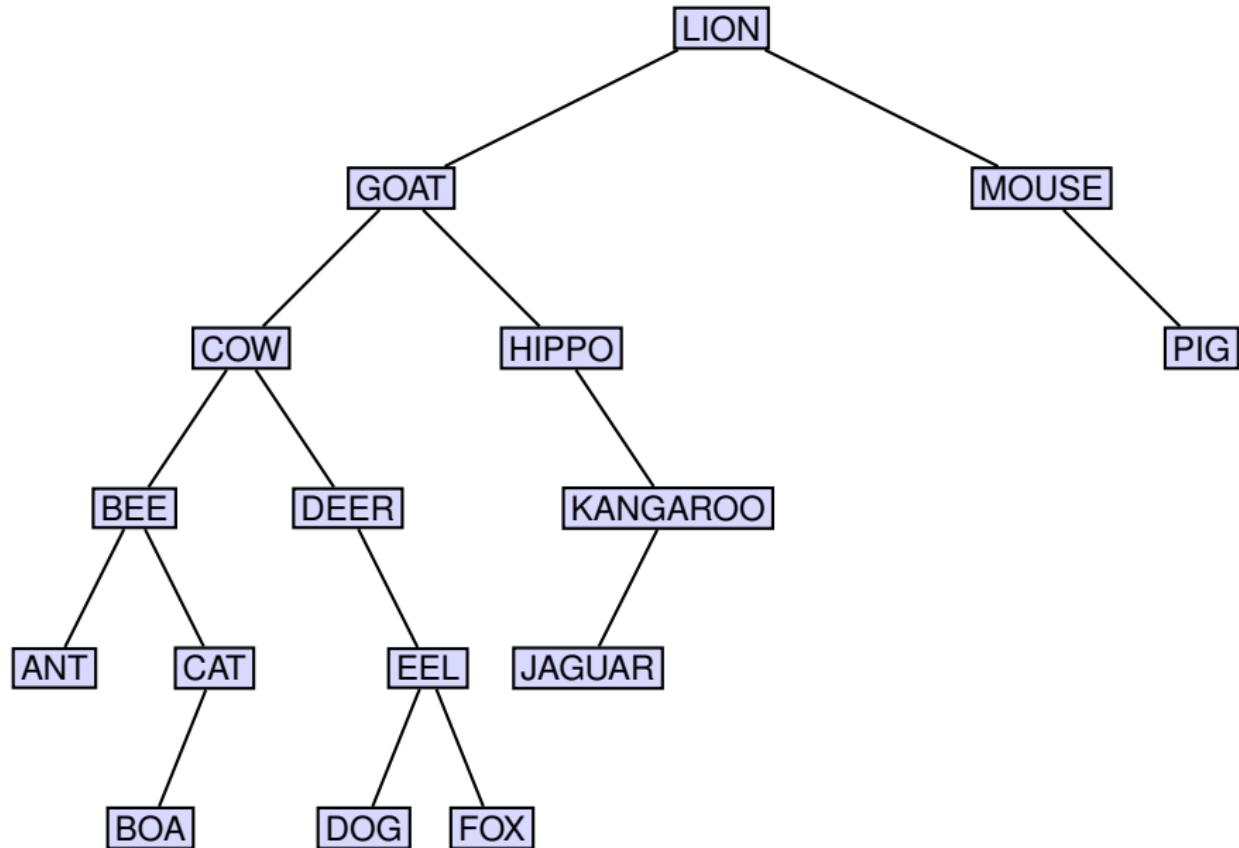
# Definition

## Definition

A **binary search tree** (BST, for short)  $T$  is a binary tree that is either empty, or it contains at least one element  $x$  at its root, and

- ① The left and right subtrees of  $T$ ,  $L$  and  $R$ , respectively, are binary search trees.
- ② For all elements  $y$  in  $L$ ,  $\text{KEY}(y) < \text{KEY}(x)$ , and for all elements  $z$  in  $R$ ,  $\text{KEY}(z) > \text{KEY}(x)$ .

## An Example



# Binary Search Trees: Basic Operations

```
template <typename Key, typename Value>
class Dictionary {
public:
    ...
    void lookup(const Key& k,
                bool& exists, Value& v) const;
    void insert(const Key& k,
                const Value& v);
    void remove(const Key& k);
    ...
private:
    struct node_bst {
        Key _k;
        Value _v;
        node_bst* _left;
        node_bst* _right;
        // constructor for the class/struct node_bst
        node_bst(const Key& k, const Value& v,
                 node_bst* left = nullptr, node_bst* right = nullptr);
    };
    node_bst* root;

    static node_bst* bst_lookup(node_bst* p,
                               const Key& k);
    static node_bst* bst_insert(node_bst* p,
                               const Key& k, const Value& v);
    static node_bst* bst_remove(node_bst* p,
                               const Key& k);
    static node_bst* join(node_bst* t1,
                         node_bst* t2);
    static node_bst* relocate_max(node_bst* p);
    ...
}
```

# Traversals

## Lemma

An *inorder traversal* of a binary search tree visits all the elements it contains in ascending order of their keys.

Traversals are often implemented via **iterators**; then the class offers methods like `begin` and `end` (among others) for the traversals.

# Traversals

A simpler (but less flexible) strategy is to have a method that “dumps” the dictionary contents into a sorted list.

```
template <typename Key, typename Value>
class Dictionary {
public:
    ...
void dump(list<pair<Key,Value>>& L) const {
    L.clear();
    bst_dump(root, L);
}
...
private:
    ...
static void bst_dump(node_bst* p, list<pair<Key,Value>>& L) {
    if (p != nullptr) {
        bst_dump(p -> _left, L);
        L.push_back(make_pair(p -> _k, p -> _v));
        bst_dump(p -> _right, L);
    }
}
...
```

# Search

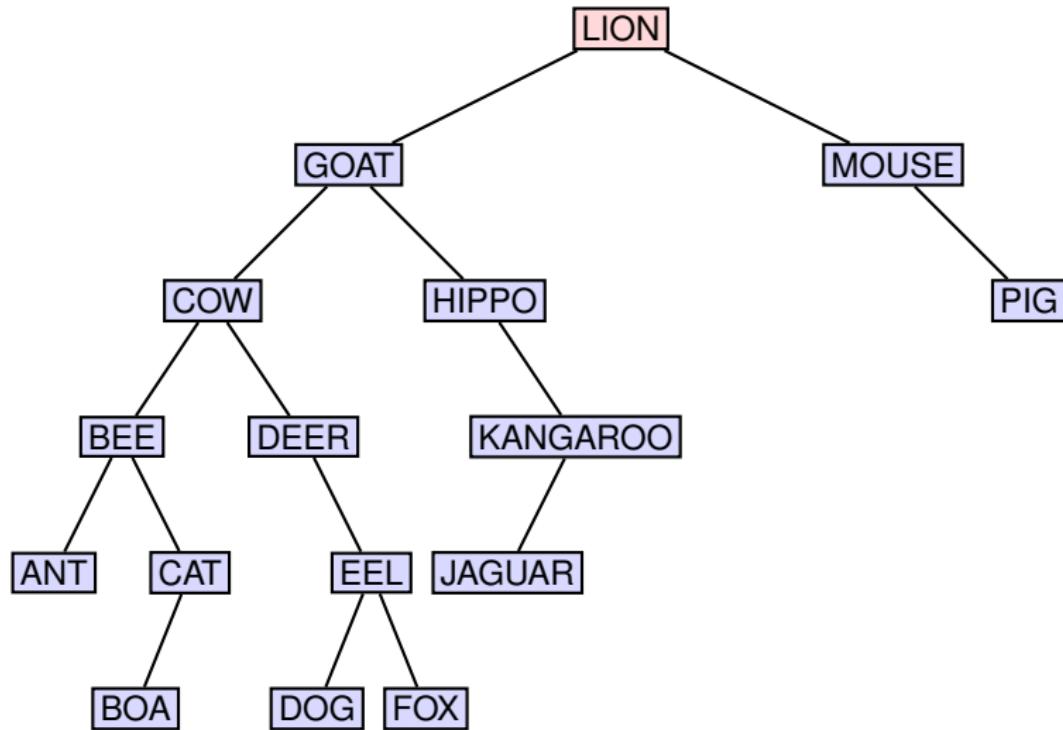
Let us consider now the search algorithm in a BST. Because of their recursive definition of BST, we will start with a recursive algorithm. Let  $T$  be a BST representing the dictionary and let  $k$  be the key we are looking for. If  $T = \square$  then  $k$  is not in the dictionary and we need only to signal this event in some convenient way (e.g. we return `false`). If  $T$  is not empty we will need to check the relation between  $k$  and the key of the element  $x$  stored at the root of  $T$ .

# Search

If  $k = \text{KEY}(x)$  the search is successful and we can stop it there, returning  $x$  or the information associated to  $x$  which we care about. If  $k < \text{KEY}(x)$ , following the definition of BSTs, if there exists an element in  $T$  with key  $k$  it must be stored in the left subtree of  $T$ ; hence, we must make a recursive call on the left subtree of  $T$  to continue the search. Analogously, if  $k > \text{KEY}(x)$  then the search must recursively continue in the right subtree of  $T$ .

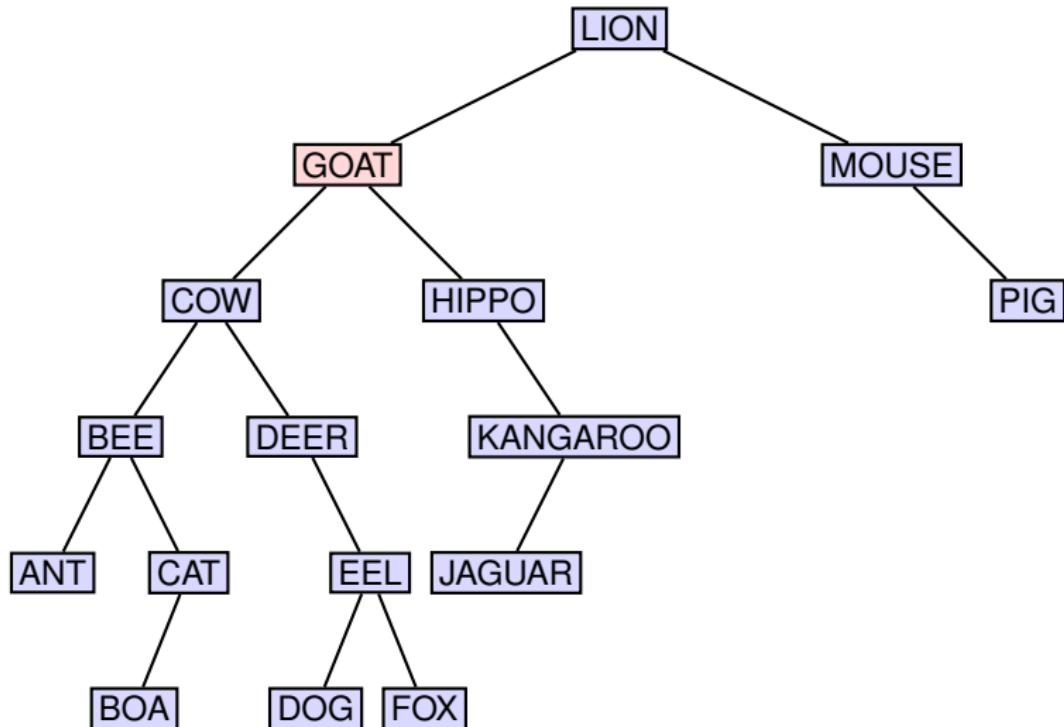
# Search: An Example

Searching for DOG



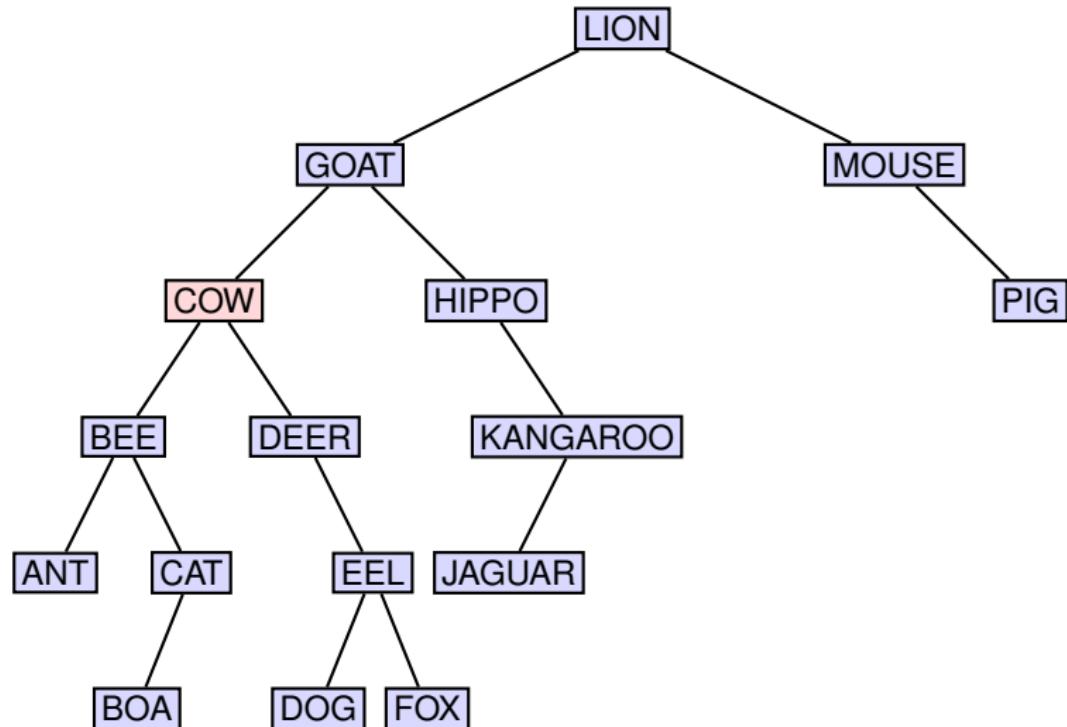
# Search: An Example

Searching for DOG



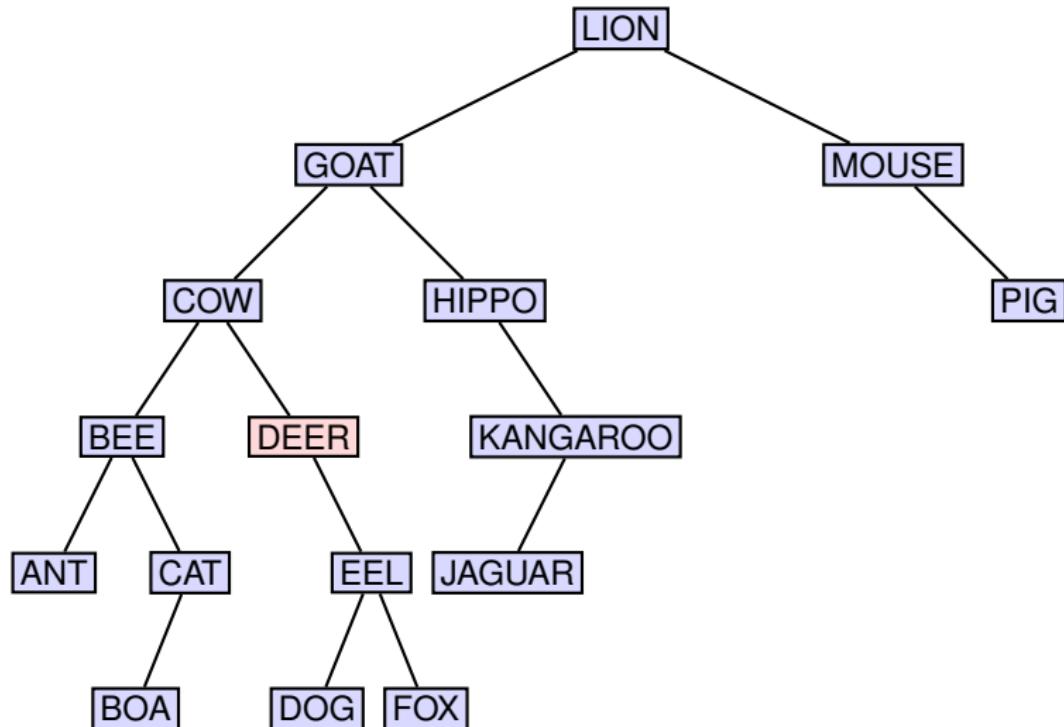
# Search: An Example

Searching for DOG



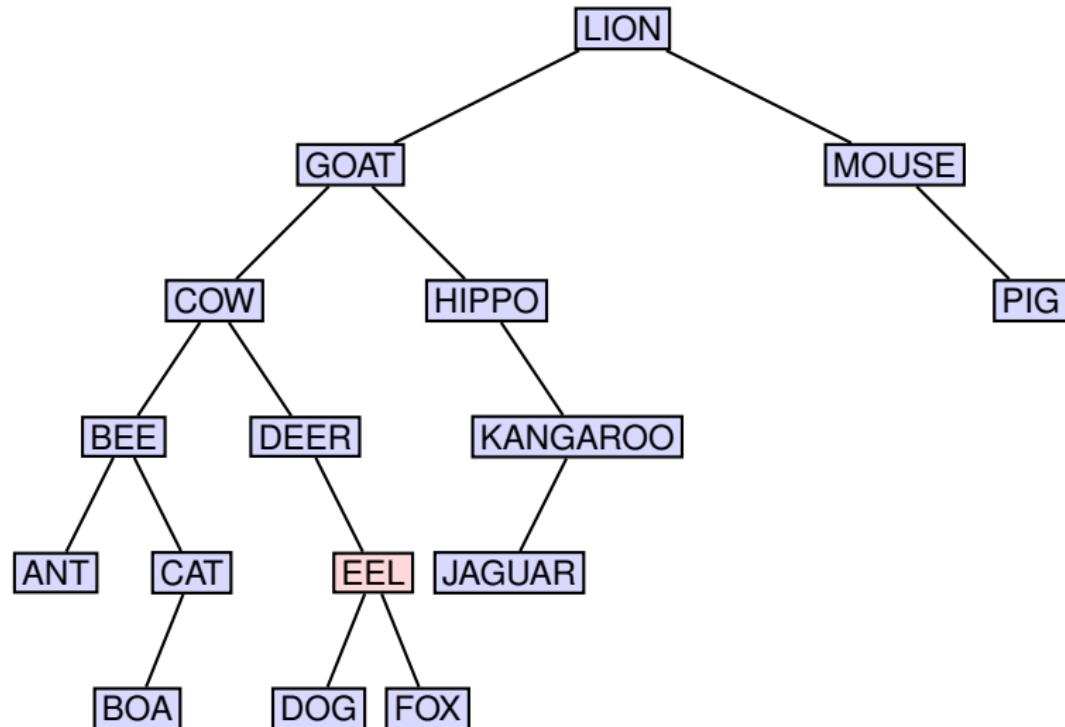
# Search: An Example

Searching for DOG



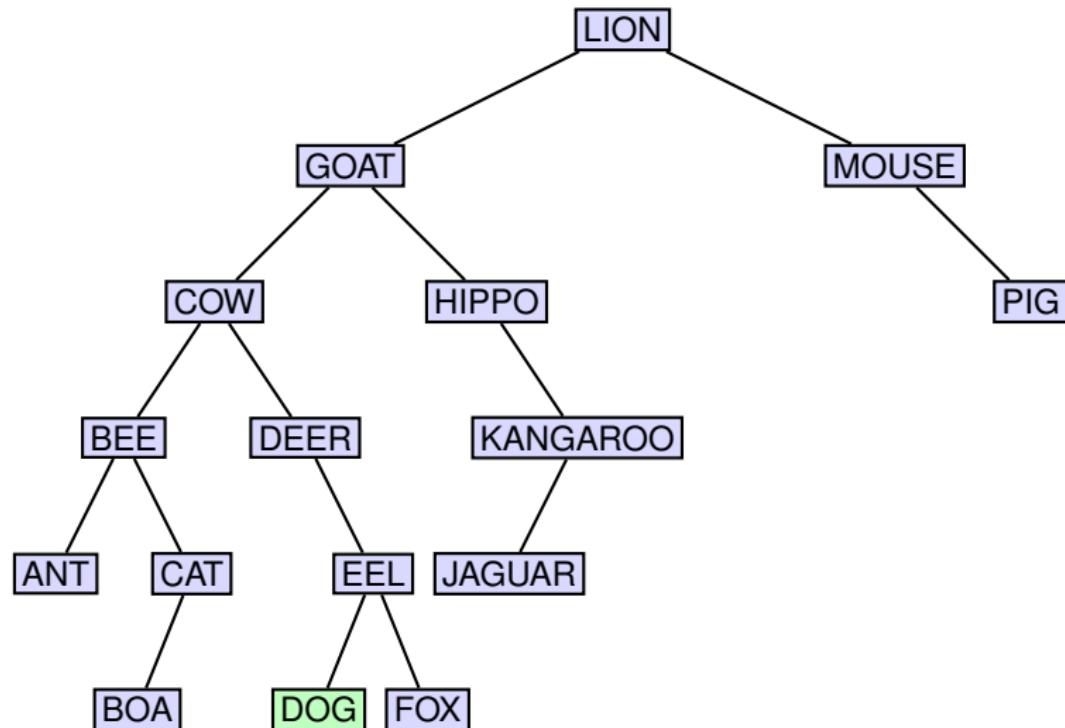
# Search: An Example

Searching for DOG



# Search: An Example

Searching for DOG



# Search

The public method LOOKUP uses the private class method BST\_LOOKUP. This one receives a pointer  $p$  to the root of the BST where we have to perform the search for the key  $k$ . It returns a pointer, either a null pointer if the search is unsuccessful or a pointer to the node that contains the element with the sought key.

```
template <typename Key, typename Value>
void Dictionary<Key,Value>::lookup(const Key& k,
    bool& exists, Value& v) const {
    node_bst* p = bst_lookup(root, k);
    if (p == nullptr)
        exists = false;
    else {
        exists = true;
        v = p -> _v;
    }
}
```

# Search

The recursive implementation of BST\_LOOKUP is almost immediate from the definition of BST. If the tree is empty or its root (the node pointed to by  $p$ ) contains the given  $k$  we are done and return  $p$ ; otherwise, we compare  $k$  with the key stored at the root node and continue either on the left or on the right, as necessary.

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::bst_lookup(node_bst* p,
                                const Key& k) {

    if (p == nullptr or k == p->_k)
        return p;

    // p != nullptr and k != p->_k
    if (k < p->_k)
        return bst_lookup(p->_left, k);
    else // p->_k < k
        return bst_lookup(p->_right, k);
}
```

# Search

The tail recursion in the previous recursive implementation of BST\_LOOKUP can be easily removed to get an iterative implementation.

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::bst_lookup(node_bst* p,
    const Key& k) {

    while (p != nullptr and k != p->_k) {
        if (k < p->_k)
            p = p->_left;
        else // p->_k < k
            p = p->_right;
    }
    return p;
}
```

## Insertions

The insertion algorithm is also very easy and we can design it by using the same reasoning as for the search algorithm: if the new key to be inserted is smaller than the key at the root insert in the left subtree; otherwise, insert into the right subtree.

The public method `INSERT` relies on the private class method `BST_INSERT`. It gets the  $\langle \text{key}, \text{value} \rangle$  to insert and a pointer to the root of the BST (the pointer is null when the tree is empty); the method returns a pointer to the root of the resulting tree. If the given key already appears in the tree then no new element is inserted, but the value associated to the key is changed to the new given value.

# Insertions

First, a trivial implementation for the constructor of the class `node_bst`:

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst(const Key& k,
    Value& v, node_bst* left, node_bst* right) :
    _k(k), _v(v), _left(left), _right(right) {
}
```

# Insertions

```
template <typename Key, typename Value>
void Dictionary::insert(const Key& k,
    const Value& v) {

    root = bst_insert(root, k, v);

}

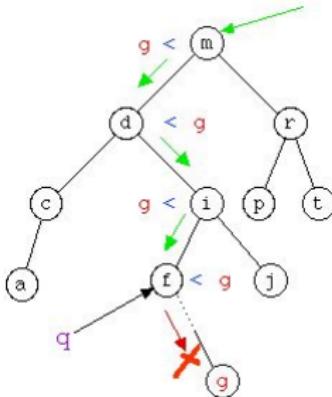
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::bst_insert(node_bst* p,
    const Key& k, const Value& v) {

    if (p == nullptr)
        return new node_bst(k, v);

    // p != nullptr, continue the insertion in the
    // appropriate subtree or update the associated
    // value if p -> _k == k
    if (k < p -> _k)
        p -> _left = bst_insert(p -> _left, k, v);
    else if (p -> _k < k)
        p -> _right = bst_insert(p -> _right, k, v);
    else // p -> _k == k
        p -> _v = v;
    return p;
}
```

# Insertions

The iterative implementation of the insertion is more complicated; besides locating the leaf (empty subtree) where the new elements must be inserted, we will need to keep a pointer  $f$  to the father of the current node. When the loop finds the empty subtree where the new element should be inserted,  $f$  points to the node that will become the father of the new node.



# Insertions

```
template <typename Key, typename Value>
Dictionary<Key,Value>::::node_bst*
Dictionary<Key,Value>::::bst_insert(node_bst* p,
    const Key& k, const Value& v) {

    // if the BST is empty
    if (p == nullptr)
        return new node_bst(k, v);

    // otherwise
    node_bst* f = nullptr;
    node_bst* root = p;

    // lookup the leaf where we have to insert (or
    // end in a node with key k to perform an update
    // of the associated value)
    while (p != nullptr and k != p -> _k) {
        f = p;
        if (k < p -> _k)
            p = p -> _left;
        else // p -> _k < k
            p = p -> _right;
    }

    // insert or update
    if (p == nullptr) { // insert a new node
        if (k < f -> _k)
            f -> _left = new node_bst(k, v);
        else // k > f -> _k
            f -> _right = new node_bst(k, v);
    }
    else // update value, k == p -> _k
        p -> _v = v;

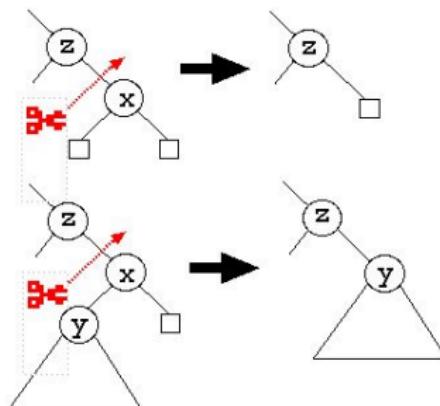
    return root;
}
```

# Deletions

Deleting an element with given key  $k$  involves two phases:

- ① Locating the node with key  $k$  (if any) in the tree
- ② Removing that node

The first phase is a lookup. For the second phase, if the node  $x$  to be removed has only one non-empty subtree, it is not very difficult to remove the node: lift the non-empty subtree so that the father of  $x$  points to the subtree's root instead of pointing to  $x$ .



## Deletions

The problematic situation is when the node  $x$  to be removed has two non-empty subtrees.

We will deal with all possible cases, even the complicated one, solving the following problem: given two BSTs  $T_1$  and  $T_2$  such that all keys in  $T_1$  are smaller than all keys in  $T_2$ , **join**  $T_1$  and  $T_2$  into a single BST  $T$  with all the keys; we will write

$$T = \text{JOIN}(T_1, T_2).$$

Quite obviously:

$$\text{JOIN}(T, \square) = T,$$

$$\text{JOIN}(\square, T) = T.$$

In particular,  $\text{JOIN}(\square, \square) = \square$ .

# Deletions

```
template <typename Key, typename Value>
void Dictionary<Key,Value>::remove(
    const Key& k) {

    root = bst_remove(root, k);
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::bst_remove(node_bst* p,
    const Key& k) {

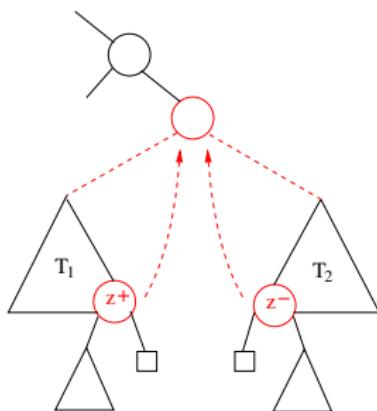
    // key k is not in the tree if it is empty
    if (p == nullptr) return p;

    if (k < p->_key)
        p->_left = bst_remove(p->_left, k);
    else if (p->_key < k)
        p->_right = bst_remove(p->_right, k);
    else { // k == p->_key
        node_bst* to_kill = p;
        p = join(p->_left, p->_right);
        delete to_kill;
    }
    return p;
}
```

## Deletions

Let  $z^+$  be the largest key in  $T_1$ . Since it is larger than the other keys in  $T_1$  but smaller than any key in  $T_2$ , we can join  $T_1$  and  $T_2$  putting  $z^+$  as the root of  $T$ ,  $T_2$  as its right subtree, and the result of removing  $z^+$  from  $T_1$  —call it  $T'_1$ — as the left subtree.

The good news is that since  $z^+$  has the largest key in  $T_1$ , the corresponding node cannot have a non-empty right subtree: therefore it is trivial to remove  $z^+$  from  $T_1$ .



# Deletions

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::join(node_bst* t1,
    node_bst* t2) {

    // trivial if one or two of the trees are empty
    if (t1 == nullptr) return t2;
    if (t2 == nullptr) return t1;

    // t1 != nullptr and t2 != nullptr
    node_bst* z = relocate_max(t1);
    z -> _right = t2;
    return z;

    // alternative: z = relocate_min(t2);
    //                 z -> _left = t1;
    //                 return z;
}
```

# Deletions

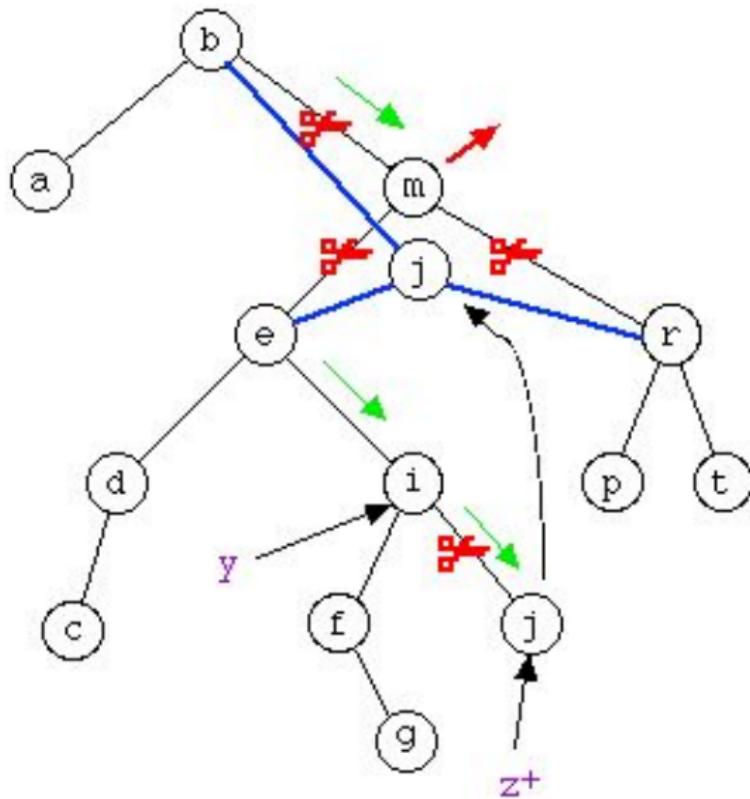
The private class method RELOCATE\_MAX receives a pointer  $p$  to the root of some BST  $T$  and it returns a pointer to the root of a new BST  $T'$ . That root contains the element with largest key in  $T$ , the right subtree of  $T'$  is empty and the left subtree of  $T'$  contains all the keys in  $T$  except the largest one which has been uplifted to the root.

There is only one tricky situation to deal with: when the largest key of  $T$  is already in the root of  $T'$ ; if such situation is detected the method has nothing more to do.

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_bst*
Dictionary<Key,Value>::relocate_max(
    node_bst* p) {

    node_bst* f = nullptr;
    node_bst* orig_p = p;
    while (p -> _right != nullptr) {
        f = p;
        p = p -> _right;
    }
    if (f != nullptr) {
        f -> _right = p -> _left;
        p -> _left = orig_p;
    }
    return p;
}
```

## Deletions



## Deletions

The joining of two trees can also be done by “relocating” the element with smallest key in  $T_2$  to become the common root.

Experimental work has shown that it is better to alternate between both versions of join (for instance, making a random choice or using one version or the other on an alternating basis). Using systematically either of the two variants leads to significantly biased and unbalanced search trees in the long run, even if the insertions are “random”.

# The Cost of Searches and Updates in BSTs

A BST of size  $n$  can be equivalent to a linked lists, with one node with no children and  $n - 1$  nodes with only one non-empty subtree. Such a tree can be built by inserting  $n$  elements in ascending or descending order of their keys into an initially empty BST.

Thus in the height of a BST can be as high as  $n$ ; in the opposite extreme a perfectly balanced BST of size  $n$  has height at least  $\lceil \log_2(n + 1) \rceil$ .

For a BST of height  $h$  the worst-case cost of a search, an insertion or a deletion has cost  $\Theta(h)$ . Hence, the worst-case for search and update operations in a BST of size  $n$  is  $\Theta(n)$ .

# The Cost of Searches and Updates in BSTs

However, on average, the height of a random BST is  $\Theta(\log n)$ , and thus the average cost of the basic operations is  $\Theta(\log n)$  too.

By a **random BST** of size  $n$ , we mean a tree built by  $n$  insertions, with all possible  $n!$  orderings of the  $n$  insertions being equally likely.

For successful searches we will assume that the sought element is any of the  $n$  elements in the tree; for unsuccessful searches (and insertions) we will assume that the search ends at any of the  $n + 1$  empty subtrees with identical probability.

# The Cost of Searches and Updates in BSTs

Moreover, we will consider only the number of comparisons between the given key and the keys of the nodes that we examine. The cost of the operation will be proportional to this number of comparisons. Let  $C(n)$  be the expected number of comparisons made by a successful search on a random BST of size  $n$ , and  $C(n, k)$  the expected number of comparisons conditioned to the root storing the element with the  $k$ -th smallest key. Then

$$C(n) = \sum_{1 \leq k \leq n} C(n; k) \times \mathbb{P}[\text{root is the } k\text{-th}] .$$

# The Cost of Searches and Updates in BSTs

$$\begin{aligned}C(n) &= \frac{1}{n} \sum_{1 \leq k \leq n} C(n; k) \\&= 1 + \frac{1}{n} \sum_{1 \leq k \leq n} \left( \frac{1}{n} \cdot 0 + \frac{k-1}{n} \cdot C(k-1) + \frac{n-k}{n} \cdot C(n-k) \right) \\&= 1 + \frac{1}{n^2} \sum_{0 \leq k < n} (k \cdot C(k) + (n-1-k) \cdot C(n-1-k)) \\&= 1 + \frac{2}{n^2} \sum_{0 \leq k < n} k \cdot C(k).\end{aligned}$$

An alternative way to analyze the cost of successful searches is to consider the so-called **Internal Path Length** (IPL). Given a BST, its IPL is the sum of the depths from the root to every other node in the tree. The *External Path Length* (EPL) is defined similarly and it is used in the analysis of unsuccessful searches/insertions.

# The Cost of Searches and Updates in BSTs

If  $I(n)$  denotes the expected value of the IPL of a random BST of size  $n$  then

$$C(n) = 1 + \frac{I(n)}{n}$$

The expected value of the IPL satisfies the following recurrence:

$$I(n) = n - 1 + \frac{2}{n} \sum_{0 \leq k < n} I(k), \quad I(0) = 0. \quad (1)$$

Indeed, every node except the root of the tree contributes to the total IPL one unit plus its contribution to the IPL of the subtree of the root to which the node belongs.

Note also that the IPL of a tree does not only give us the cost of successful searches in that tree; it is also the cost of building that tree starting from an empty tree.

# The Cost of Searches and Updates in BSTs

Recurrence (1) is identical to the recurrence for the cost of quicksort!! That's no coincidence. The recursion tree associated to a quicksort execution is a BST of size  $n$ : each node holds the pivot used at the corresponding recursive call. Now, every element in the tree has been compared (not being the pivot) against every pivot selected in the sequence of recursive calls leading from the initial call to the recursive call where the element is finally chosen as the pivot. That is, every element is compared to all its proper ancestors in the BSTs, or in other terms, the number of comparisons it gets involved (not as a pivot) is its depth in the BST. Hence  $I(n) = Q(n)$ , where  $Q(n)$  is the expected number of comparisons made by quicksort to sort an array of size  $n$ .

# The Cost of Searches and Updates in BSTs

In what follows we provide a solution of the recurrence (although we had already solved it using CMT). Our first step is to compute  $(n + 1)I(n + 1) - nI(n)$ :

$$\begin{aligned}(n + 1)I(n + 1) - nI(n) &= (n + 1)n - n(n - 1) + 2I(n) \\ &= 2n + 2I(n);\end{aligned}$$

$$(n + 1)I(n + 1) = 2n + (n + 2)I(n)$$

# The Cost of Searches and Updates in BSTs

$$\begin{aligned}I(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1} I(n) = \frac{2n}{n+1} + \frac{2(n-1)(n+2)}{n(n+1)} + \frac{n+2}{n} I(n-1) \\&= \frac{2n}{n+1} + \frac{2(n-1)(n+2)}{n(n+1)} + \frac{2(n-2)(n+2)}{n(n-1)} + \frac{n+2}{n-1} I(n-2) \\&= \frac{2n}{n+1} + 2(n+2) \sum_{i=1}^{i=k} \frac{n-i}{(n-i+1)(n-i+2)} + \frac{n+2}{n-k+1} I(n-k) \\&= \frac{2n}{n+1} + 2(n+2) \sum_{i=1}^{i=n} \frac{i}{(i+1)(i+2)} \\&= \mathcal{O}(1) + 2(n+2) \sum_{1 \leq i \leq n} \left( \frac{2}{i+2} - \frac{1}{i+1} \right) \\&= \mathcal{O}(1) + 2(n+2) \left( \frac{2}{n+2} + \frac{1}{n+1} + H_n - 2 \right) \\&= 2nH_n - 4n + 4H_n + \mathcal{O}(1),\end{aligned}$$

# The Cost of Searches and Updates in BSTs

where  $H_n = \sum_{1 \leq i \leq n} 1/i$ .

Since  $H_n = \ln n + \mathcal{O}(1)$  we have

$$\begin{aligned} I(n) &= Q(n) = 2n \ln n + \mathcal{O}(n) \\ &= 1.386 \dots n \log_2 n + \mathcal{O}(n) \end{aligned}$$

For a successful search in a BST of size  $n$ , the average number of comparisons is then

$$C(n) = 1 + \frac{I(n)}{n} = 2 \ln n + \mathcal{O}(1).$$

## Other Operations

Most other operations in BSTs are also very simple to implement and have good expected cost. To achieve logarithmic expected cost in searches and deletions by rank, or in counting operations it is necessary to modify the standard implementation of BSTs (their representation and the basic operations) so that each node in the BST holds updated information about the size of the subtree rooted at that node.

We conclude with a particular example here: given two keys  $k_1$  and  $k_2$  with  $k_1 < k_2$  we implement a new method `in_range` which returns a list of the elements in the BST (actually a list of pairs  $\langle key, value \rangle$ ) which key  $k$  is within the range  $[k_1 \dots k_2]$ , that is,  $k_1 \leq k \leq k_2$ .

## Other Operations: `in_range`

If the BST is empty, the result is an empty list. If the BST is not empty, let  $k$  be the key at the root. If  $k < k_1$  then no element in the left subtree will be returned and no recursive call on the left subtree should be made. Likewise, if  $k_2 < k$ , no element in the right subtree is within the range and we have thus to avoid the recursive call in the right subtree.

If  $k_1 \leq k \leq k_2$  then the element at the root is within the range and must be added to the result; moreover, elements from both the left and the right subtrees might be within the range too and thus both subtrees must be recursively explored. To make sure that the returned list is in ascending order of keys, we must first make the recursive call in the left subtree, then add the root element to the list, finally make the recursive call into the right subtree.

It can be shown that the average cost is  $\Theta(\log n + F)$ , where  $F$  is the length of the result (it can go from  $\Theta(1)$  when  $k_1$  and  $k_2$  aren't too far apart, to  $\Theta(n)$  if every key is within the range).

## Other Operations: in\_range

```
template <typename Key, typename Value>
void Dictionary<Key,Value>::in_range(
    const Key& k1, const Key& k2,
    list<pair<Key,Value> >& result) const {

    bst_in_range(root, k1, k2, result);
}

template <typename Key, typename Value>
void Dictionary<Key,Value>::bst_in_range(node_bst* p,
    const Key& k1, const Key& k2,
    list<pair<Key,Value> >& result) const {

    if (p == nullptr) return;
    if (k1 <= p->_k)
        bst_in_range(p->_left, k1, k2, result);

    if (k1 <= p->_k and p->_k <= k2)
        result.push_back(make_pair(p->_k, p->_v));

    if (p->_k <= k2)
        bst_in_range(p->_right, k1, k2, result);
}
```

# Part III

## Dictionaries

9 Binary Search Trees

10 Height-balanced Binary Search Trees: AVLs

11 Hash Tables

## Balanced Trees

The main drawback of standard BSTs is that they can be strongly unbalanced.

In the worst-case, the height of BST of size  $n$  is  $\Theta(n)$ , hence the worst-case cost of search, insertion and deletion is  $\Theta(n)$  too.

Several alternatives have been proposed to overcome this problem: the oldest and possibly one of the simplest and most elegant solution is **height balancing**, originally proposed by the Russian computer scientists Adelson-Velskii and Landis. These search trees are named **AVLs** after their inventors.

# AVLs

## Definition

An AVL  $T$  is a binary search tree that is either empty or

- 1 Its left and right subtrees,  $L$  and  $R$ , respectively are AVLs
- 2 The height of  $L$  and  $R$  differs by one at most:

$$|\text{height}(L) - \text{height}(R)| \leq 1$$

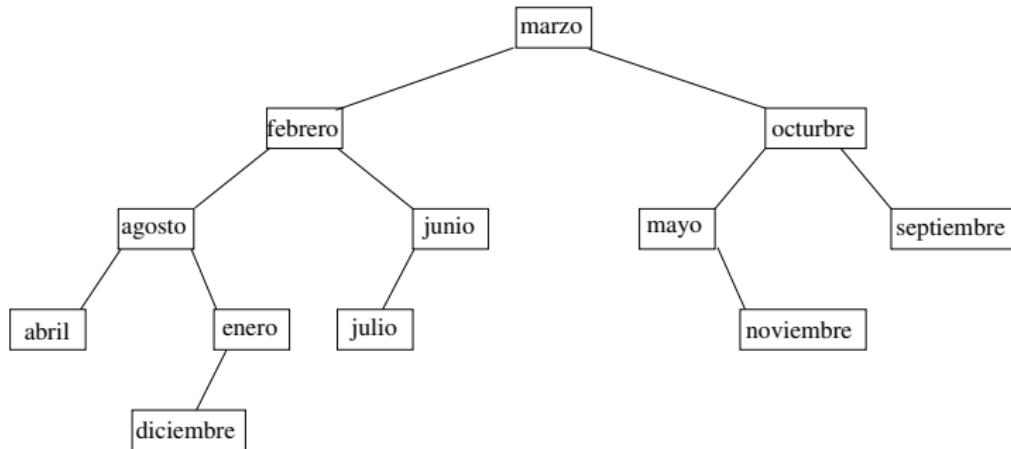
## AVLs

The previous recursive definition guarantees that the balance invariant holds for all nodes (roots of the corresponding subtrees) of an AVL. If we denote  $\text{bal}(x)$  the difference between the height of the left and right subtrees of an arbitrary node  $x$ , then we must have  $\text{bal}(x) \in \{-1, 0, +1\}$ .

Since AVLs are BSTs, the search algorithm in an AVL is exactly the same as the algorithm for standard BSTs; moreover, an inorder traversal of an AVL visits its nodes in ascending order of the keys.

# AVLs

{enero, febrero, marzo, abril, mayo, junio, julio, agosto, septiembre, octubre  
noviembre, diciembre}



# The Cost of AVLs

## Lemma

*The height  $h(T)$  of an AVL  $T$  of size  $n$  is  $\Theta(\log n)$ .*

*Proof.* Since any AVL is a binary tree we have  $h(T) \geq \lceil \log_2(n + 1) \rceil$ , that is,  $h(T) \in \Omega(\log n)$ . We need just to show that  $h(T) \in \mathcal{O}(\log n)$ .

Let  $N_h$  the least number of nodes we need to build an AVL of size  $h$ . Clearly,  $N_0 = 0$  and  $N_1 = 1$ . The most unbalanced possible AVL with height  $h > 1$  can be built by putting a root then a left subtree which is a most unbalanced AVL of height  $h - 1$  (using only  $N_{h-1}$  nodes), and a right subtree which is a most unbalanced AVL of height  $h - 2$  (using  $N_{h-2}$ ). Hence

$$N_h = 1 + N_{h-1} + N_{h-2}$$

# The Cost of AVLs

A few values of the sequence  $\{N_h\}_{h \geq 0}$ :

$$0, 1, 2, 4, 7, 12, 20, 33, 54, 88, \dots$$

Its similarity to the Fibonacci sequence draws our attention. It is not difficult to prove (for instance, by induction) that  $N_h = F_{h+1} - 1$  for all  $h \geq 0$ . Indeed,  $N_0 = F_1 - 1 = 1 - 1 = 0$  and

$$N_h = 1 + N_{h-1} + N_{h-2} = 1 + (F_h - 1) + (F_{h-1} - 1) = F_{h+1} - 1$$

The  $n$ -th Fibonacci number is

$$F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor,$$

where  $\phi = (1 + \sqrt{5})/2 \approx 1.61803\dots$  denotes the *golden ratio*.

# The Cost of AVLs

Consider now an AVL of size  $n$  and height  $h$ . Then we must have  $n \geq N_h$  by definition, and

$$n \geq F_{h+1} - 1 \geq \frac{\phi^{h+1}}{\sqrt{5}} - \frac{3}{2}$$

Hence

$$(n + \frac{3}{2}) \frac{\sqrt{5}}{\phi} \geq \phi^h$$

Taking logarithms base  $\phi$ , and after simplification

$$h \leq \log_\phi n + \mathcal{O}(1) = 1.44 \log_2 n + \mathcal{O}(1)$$

## Updating AVLs

The previous lemma guarantees that the cost of any search in an AVL of size  $n$  is  $\mathcal{O}(\log n)$ , even in the worst case.

The problem we face now is how to maintain the balance invariant of AVLs after insertions and deletions.

Both insertions and deletions act as their counterparts for standard BSTs, but all the nodes in the path from the root to the ins/del point must be checked for balance; if the operation invalidates the balance condition at some node  $x$ , we must do something to fix it and reestablish the invariant.

The first observation is that we will need to store at each node information about its height or its balance ( $\text{bal}(\cdot)$ ) to avoid costly computations.

# Implementation of AVLs

```
template <typename Key, typename Value>
class Dictionary {
public:
    ...
    void lookup(const Key& k,
                bool& exists, Value& v) const;
    void insert(const Key& k,
                const Value& v);
    void remove(const Key& k);
    ...
private:
    struct node_avl {
        Key _k;
        Value _v;
        int _height;
        node_avl* _left;
        node_avl* _right;
        // constructor for class node_avl
        node_avl(const Key& k, const Value& v,
                 int height = 1,
                 node_avl* left = nullptr,
                 node_avl* right = nullptr);
    };
    node_avl* root;
    ...
    static int height(node_avl* p);
    static void update_height(node_avl* p);
};
```

# Implementation of AVLs

```
int max(int x, int y) {
    return x > y ? x : y;
}

template <typename Key, typename Value>
static int Dictionary<Key,Value>::height(
    node_avl* p) {

    if (p == nullptr)
        return 0;
    else
        return p -> _height;
}

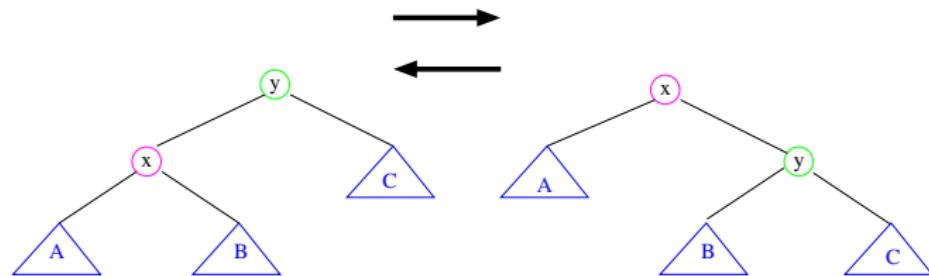
template <typename Key, typename Value>
static void Dictionary<Key,Value>::update_height(
    node_avl* p) {

    p -> _height = 1 + max(height(p -> _left), height(p -> _right));
}
```

# Rotations

To reestablish the balance invariant in a node where it didn't hold we will use **rotations**.

Figure below shows the simplest rotations. Note that if the tree on the left is a BST ( $A < x < B < y < C$ ) then the tree on the right is a BST too.



## Rotations

Assume there is an AVL subtree with root  $y$  and that we are inserting a key  $k$  such that  $k < x < y$ ; once the insertion is made, assume that  $y$  gets unbalanced.

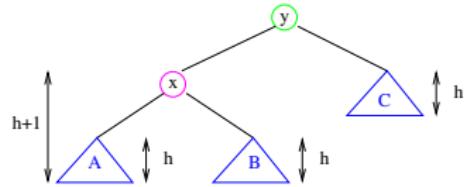
If the height of  $C$  is  $h$  then the subtree rooted at  $x$  must have height  $h$  or height  $h + 1$ . If the height of  $x$  was  $h$  then the insertion of the new key  $k$  cannot make  $y$  unbalanced, hence we must assume that the height of  $x$  is  $h + 1$ .

## Rotations

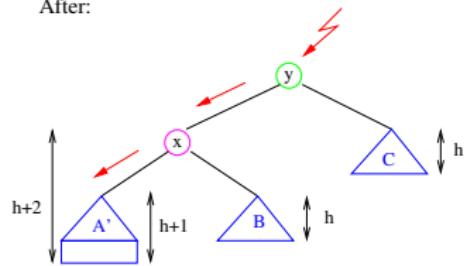
Analogous reasonings lead us to conclude that the height of  $A$  must be  $h$ , hence the height of  $y$  before the insertion is  $h + 2$ . Assume now that the height of  $A$  increases as a consequence of the insertion, that is, the resulting subtree  $A'$  has height  $h + 1$ . If we assume that  $x$  is still balanced after the insertion (but  $y$  does not) we must conclude that  $B$  has height  $h$  too. After the insertion the subtree rooted at  $x$  increases its height to  $h + 2$  and thus  $\text{bal}(y) = +2$ !

# Rotations

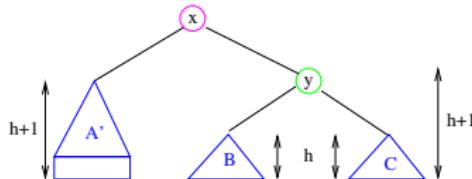
Before:



After:



LL



## Rotations

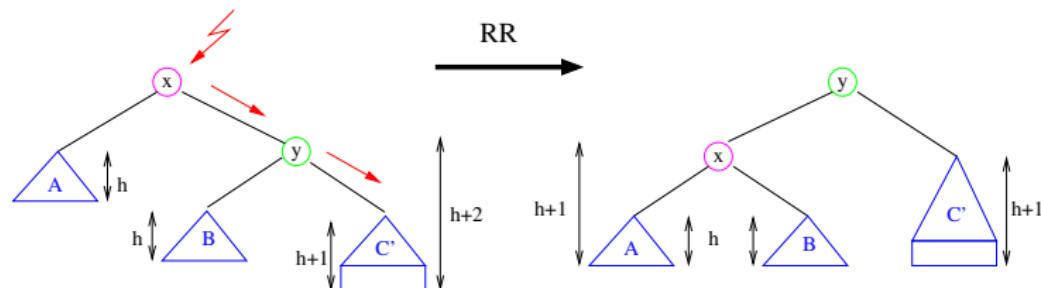
Applying the rotation to node  $y$  we preserve the BST property but we also reestablish the AVL condition on  $y$ . By hypothesis,  $A'$ ,  $B$  and  $C$  are AVLs. After the rotation the height of  $y$  is  $h + 1$  and its balance is 0, and the height of  $x$  changes to  $h + 2$  and its balance is 0 too.

We have not just solved the unbalance of  $y$ : the subtree where we continued with the insertion is an AVL and its height is the same as the height of the subtree prior to the insertion, which means that no ancestor of  $y$  in the AVL will be unbalanced, once the unbalance has been fixed.

The rotation that we have used in this example is often called **simple rotation LL** (left-left).

# Rotations

An analogous analysis reveals that when the new inserted key  $k$  satisfies  $x < y < k$  we maybe unbalance the node  $x$ ; this is unbalance is corrected applying a **simple rotation RR** (right-right) on node  $x$ . Rotation RR is the symmetric of a rotation LL.



## Rotations

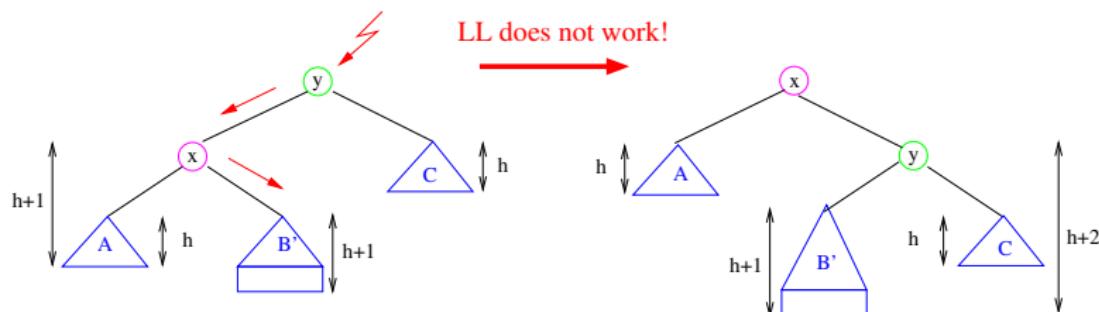
We analyze now the case where we have an AVL subtree with root  $y$  and we are making the insertion of a key  $k$  with  $x < k < y$ . Suppose that once  $k$  has been inserted, node  $y$  gets unbalanced.

Reasoning as before we conclude that if  $\text{height}(C) = h$  then the height of the subtree rooted at  $x$  must be  $h + 1$  and  $\text{height}(B) = h$ .

Let  $B'$  be the result of inserting the new key into  $B$ . Suppose that the height of  $B$  increases after the insertion, i.e.,  $\text{height}(B') = h + 1$ . If  $x$  is still balanced, but  $y$  does not, we conclude that  $\text{height}(x) = h$ . After the insertion, the height of the subtree rooted at  $x$  changes to  $h + 2$  and hence  $\text{bal}(y) = +2$ !

# Rotations

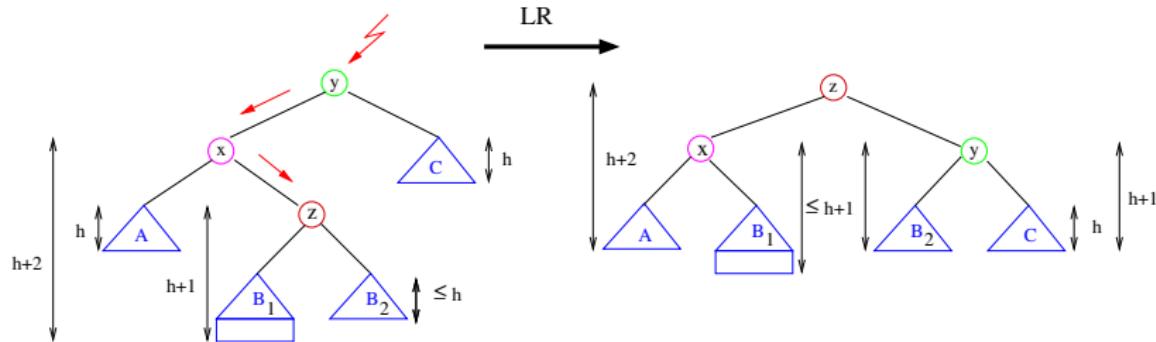
If we apply a simple rotation LL on node  $y$ , the BST condition is preserved but the AVL condition is not satisfied. Indeed, by hypothesis,  $A$ ,  $B'$  and  $C$  are AVLs. After a rotation LL height( $y$ ) =  $h + 2$  and  $\text{bal}(y) = +1$ ; but the height( $x$ ) =  $h + 3$  and  $\text{bal}(x) = -2$ !



We cannot solve the problem with simple rotations. We must figure out something else.

# Rotations

Suppose the root of  $B'$  ( $B' = \text{the result of insertion into } B$ ) is  $z$  and that the subtrees of  $z$  are the two AVLs  $B_1$  and  $B_2$ . Since  $\text{height}(B') = h + 1$  and  $B'$  is an AVL, at least one subtree  $B_i$  has height  $= h$  and the other  $B_j$  has height  $= h$  or  $= h - 1$ . We will apply a rotation bringing  $z$  to the root, with  $x$  the root of the left subtree and subtrees  $A$  and  $B_1$ , and  $y$  the root of the right subtree with children  $B_2$  and  $C$ .



Note that inorder traversal before and after the rotation remains the same; the new type of rotation preserves the BST property

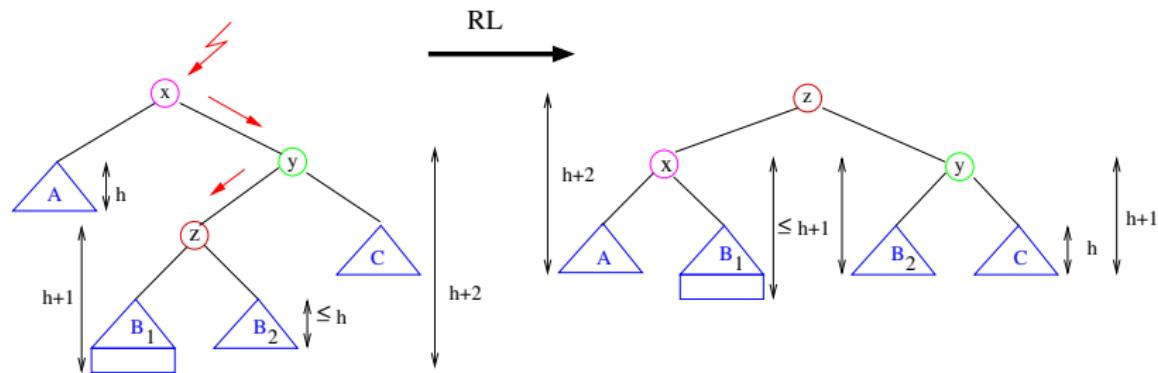
## Rotations

The rotation on node  $y$  preserves the BST property, but it also reestablishes the balance at node  $y$ . By hypothesis,  $A$ ,  $B$  and  $C$  are AVLs. After rotation the height of the subtree rooted at  $x$  is  $h + 1$  and its balance is 0 or +1, the height of  $y$  is  $h + 1$  and its balance 0 or -1, and  $\text{height}(z)$  becomes  $h + 2$  and the balance continues to be 0.

Similar to rotations LL and RR, the application of the **double rotation LR** (left-right) on node  $y$  reestablishes its balance, we now that no further violations of the AVL condition will happen in any ancestor of  $y$ .

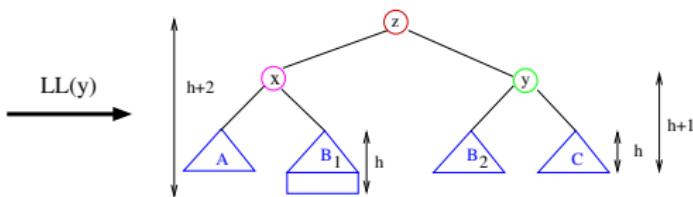
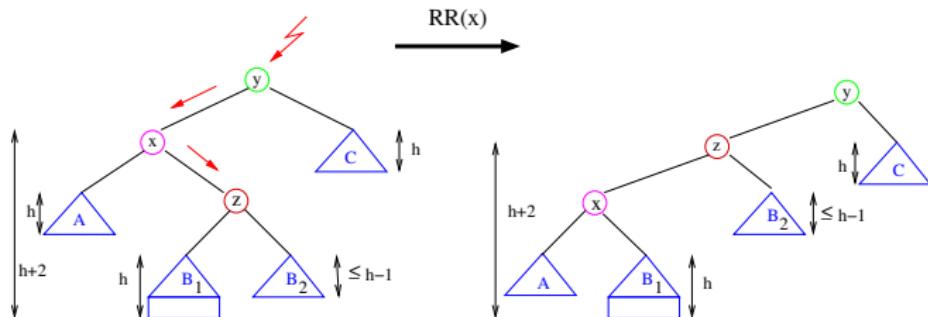
# Rotations

When a new key goes first to the right, then to the left of a node, the unbalance at the node (if it happens) is corrected with a double rotation RL, which is symmetric to a rotation LR.



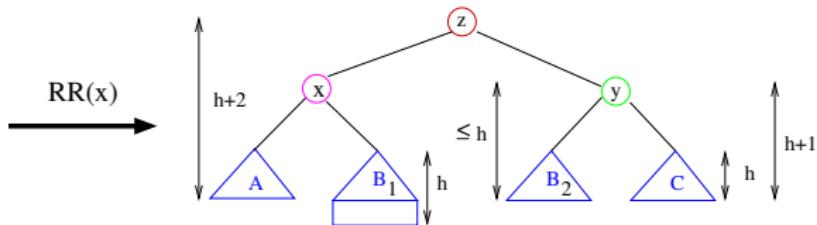
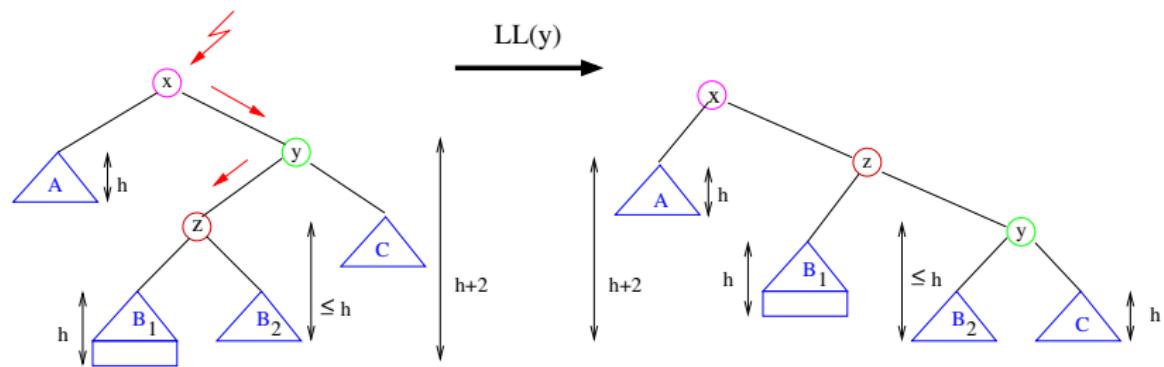
# Rotations

Double rotations LR and RL are called **double** because we can decompose each as a sequence of two simple rotations. For instance, a double rotation LR on node  $y$  is equivalent to applying a rotation RR on  $x$  followed by a rotation LL on  $y$  (note that its left subtree is, after the simple rotation,  $z$ , not  $x$ ).



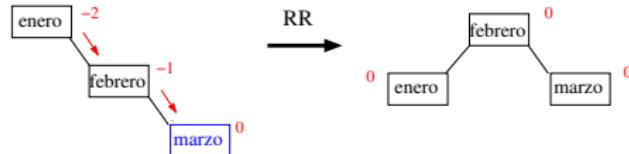
# Rotations

A rotation RL is the sequential composition of a rotation LL, followed by a rotation RR.

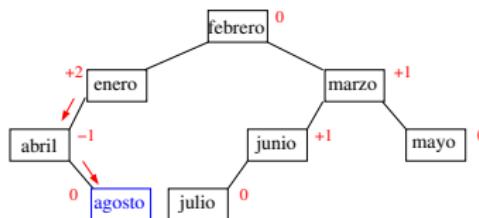


# Insertions in AVLs

Example of an AVL storing the Spanish names of the months:

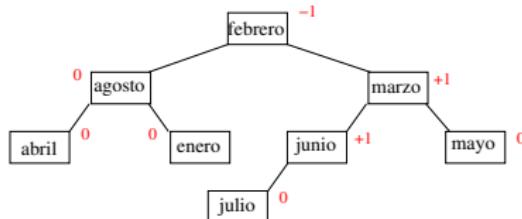


+{enero, febrero, marzo}



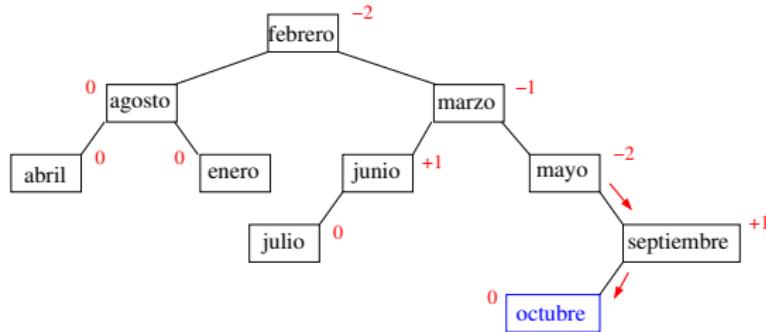
+ {abril, mayo, junio, julio, agosto}

↓ LR



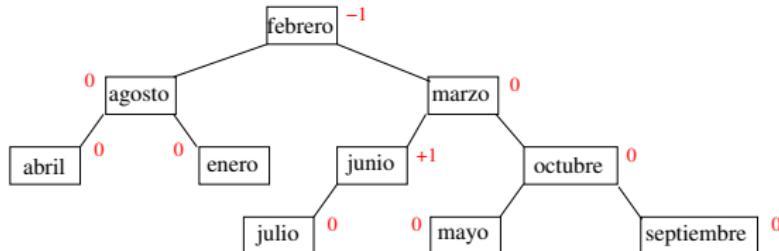
# Insertions in AVLs

Example of an AVL storing the Spanish names of the months:



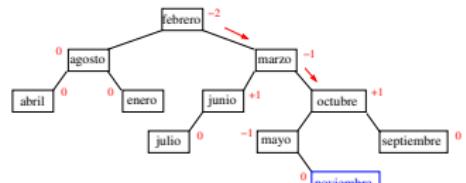
+ {septiembre, octubre}

↓  
RL

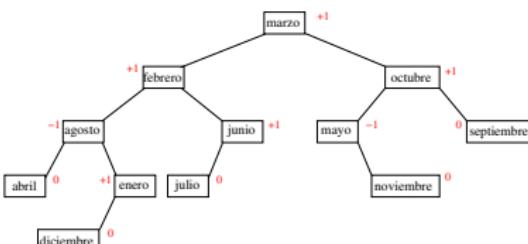
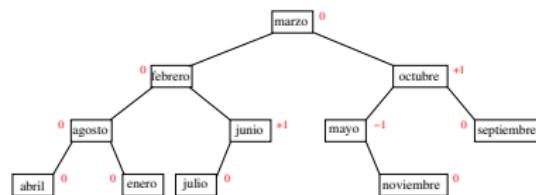


# Insertions in AVLs

Example of an AVL storing the Spanish names of the months:



+ {noviembre}



+ {diciembre}

# Insertions in AVLs

Facts:

- ➊ A rotation (simple or double) does only involve update a few pointers and the update of the `_height` fields, and its cost is  $\Theta(1)$ , independent of the tree's size.
- ➋ During an insertion in an AVL we will need to perform a rotation at most to reestablish the AVL condition. The rotation is applied (if needed) to a node in the path from the root to the insertion point.
- ➌ The worst-case cost of an insertion in an AVL is  $\Theta(\log n)$ .

# Insertions in AVLs

```
template <typename Key, typename Value>
class Dictionary {
public:
    ...
private:
    struct node_avl {
        ...
    };
    node_avl* root;
    ...
    static node_avl* avl_insert(node_avl* p,
        const Key& k, const Value& v);
    static node_avl* rotate_LL(node_avl* p);
    static node_avl* rotate_LR(node_avl* p);
    static node_avl* rotate_RL(node_avl* p);
    static node_avl* rotate_RR(node_avl* p);
    ...
};
```

# Insertions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::rotate_LL(node_avl* p) {
    node_avl* q = p -> _left;
    p -> _left = q -> _right;
    q -> _right = p;
    update_height(p);
    update_height(q);
    return q;
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::rotate_RR(node_avl* p) {
    node_avl* q = p -> _right;
    p -> _right = q -> _left;
    q -> _left = p;
    update_height(p);
    update_height(q);
    return q;
}
```

# Insertions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::rotate_LR(node_avl* p) {
    p -> _left = rotate_RR(p -> _left);
    return rotate_LL(p);
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::rotate_RL(node_avl* p) {
    p -> _right = rotate_LL(p -> _right);
    return rotate_RR(p);
}
```

# Insertions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::::node_avl*
Dictionary<Key,Value>::::avl_insert(node_avl* p,
    const Key& k, const Value& v) {

    if (p == nullptr)
        return new node_avl(k, v);

    if (k < p->_k) {
        p->_left = avl_insert(p->_left, k, v);
        // chec balance at p and rotate if needed
        if (height(p->_left) - height(p->_right) == 2) {
            // p->_left cannot be empty
            if (k < p->_left->_k) // LL
                p = rotate_LL(p);
            else // LR
                p = rotate_LR(p);
        }
    }
    else if (p->_k < k) { // symmetric case
        p->_right = avl_insert(p->_right, k, v);
        if (height(p->_right) - height(p->_left) == 2) {
            if (p->_right->_k < k) // RR
                p = rotate_RR(p);
            else // RL
                p = rotate_RL(p);
        }
    }
    else // p->_k == k
        p->_v = v;

    update_height(p);

    return p;
}
```

## Insertions in AVLs

We can write an iterative version of the AVL insertion algorithm, but it is quite complicated. Once the new item is inserted to replace a leaf (empty subtree) we must unroll the followed path to check if there is some node there which got unbalanced and apply the corresponding rotation.

A recursive implementation does the unrolling of the search path “for free”, after returning from a recursive call we need just to check if the current node is still balanced or not and act accordingly. For an iterative implementation we would need either: 1) explicit pointers to the parent; or 2) store the path from the root in a stack, and pop elements (=pointers to nodes) from the stack to unroll the path; or 3) more exotic alternatives such as *pointer reversal*.

## Deletions in AVLs

The deletion algorithm for AVLs draws upon the same ideas, although the analysis to determine what rotation must be applied is a bit more complex. The worst-case of deletions in AVLs is  $\Theta(\log n)$ . Unbalance in a node is corrected by a simple or a double rotation, but it might be necessary to apply several rotations to fix the balance. However, not more than a rotation is necessary in any given node in the path from the root to the deletion point, since there are only  $\mathcal{O}(\log n)$ , the worst-case is  $\Theta(\log n)$ . The algorithm is slightly more complicated than that of insertions; the iterative version is still more complicated.

# Deletions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::::node_avl*
Dictionary<Key,Value>::::avl_remove(node_avl* p,
    const Key& k) {
    if (p == nullptr) return p;
    if (k < p->_k) {
        p->_left = avl_remove(p->_left, k);
        // check balance and rotate if needed
        if (height(p->_right) - height(p->_left) == 2) {
            // p->_right cannot be empty
            if (height(p->_right->_left)
                - height(p->_right->_right) == 1)
                p = rotate_RL(p);
            else
                p = rotate_RR(p);
        }
    }
    else if (p->_k < k) { // symmetric case
        p->_right = avl_remove(p->_right, k);
        if (height(p->_left) - height(p->_right) == 2) {
            if (height(p->_left->_right)
                - height(p->_right->_left) == 1)
                p = rotate_LR(p);
            else
                p = rotate_LL(p);
        }
    }
    else { // p->_k == k
        node_avl* to_kill = p;
        p = join(p->_left, p->_right);
        delete to_kill;
    }
    update_height(p);
    return p;
}
```

# Deletions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::join(node_avl* t1,
    node_avl* t2) {

    // trivial, if one of the trees is empty
    if (t1 == nullptr) return t2;
    if (t2 == nullptr) return t1;

    // t1 != nullptr and t2 != nullptr
    node_avl* z;
    remove_min(t2, z);
    z -> _left = t1;
    z -> _right = t2;
    update_height(z);
    return z;
}
```

# Deletions in AVLs

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_avl*
Dictionary<Key,Value>::remove_min(
    node_avl*& p, node_avl*& z) {

    if (p -> _left != nullptr) {
        remove_min(p -> _left, z);
        // check balance and rotate if needed
        if (height(p -> _right) - height(p -> _left) == 2) {
            // p -> _right cannot be empty
            if (height(p -> _right -> _left)
                - height(p -> _right -> _right) == 1)
                p = rotate_RL(p);
            else
                p = rotate_RR(p);
        }
    } else {
        z = p;
        p = p -> _right;
    }
    update_height(p);
}
```

# Part III

## Dictionaries

- 9 Binary Search Trees
- 10 Height-balanced Binary Search Trees: AVLs
- 11 Hash Tables

# Hash Tables

A **hash table** (cat: *taula de dispersió*, esp: *tabla de dispersión*) allows us to store a set of elements (or pairs  $\langle key, value \rangle$ ) using a **hash function**

$h : K \implies I$ , where  $I$  is the set of indices or addresses into the table, e.g.,  $I = [0..M - 1]$ .

Ideally, the hash function  $h$  would map every element (their keys) to a distinct address of the table, but this is hardly possible in a general situation, and we should expect to find **collisions** (different keys mapping to the same address) as soon as the number of elements stored in the table is  $n = \Omega(\sqrt{M})$ .

# Hash Tables

If the hash function evenly “spreads” the keys, the hash table will be useful as there will be a small number of keys mapping to any given address of the table.

Given two distinct keys  $x$  and  $y$ , we say that they are **synonyms**, also that they **collide** if  $h(x) = h(y)$ .

A fundamental problem in the implementation of a dictionary using a hash table is to design a **collision resolution strategy**.

# Hash Tables

```
template <typename T> class Hash {
public:
    int operator()(const T& x) const;
};

template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
public:
    ...
private:
    struct node {
        Key _k;
        Value _v;
        ...
    };
    nat _M; // capacity of the table
    nat _n; // number of elements (size) of the table
    double _alpha_max; // max. load factor
    HashFunct<Key> h;

    // open addressing
    node* _Thash; // an array with pairs <key,value>

    // separate chaining
    // node** _Thash; // an array of pointers to linked lists of synonyms

    int hash(const Key& k) {
        return h(k) % _M;
    }
};
```

# Hash Functions

A good hash function  $h$  must enjoy the following properties

- ➊ It is easy to compute
- ➋ It must evenly spread the set of keys  $K$ : for all  $i$ ,  $0 \leq i < M$

$$\frac{\#\{k \in K \mid h(k) = i\}}{\#\{k \in K\}} \approx \frac{1}{M}$$

# Hash Functions

Defining a good hash function is not an easy task and it requires a solid background in several branches of Math. This task is very closely related to the definition of good (pseudo)random number generators.

As a general rule, the keys are first converted to positive integers (reading the binary representation of the key as a number), some mathematical transformation is applied, taking the result modulo  $M$  (the size of the table). For various theoretical reason, it is a good idea that  $M$  is a prime number.

## Hash Functions

In our implementation, the class `Hash<T>` overloads operator `()` so that for an object `h` of the class `Hash<T>`, `h(x)` is the result of “applying” `h` to the object `x` of class `T`. The operation returns a positive integer.

The private method `hash` in class `Dictionary` computes

$$h(x) \% \_M$$

to obtain a valid position into the table, an index between 0 and `\_M - 1`.

# Hash Functions

```
// specialization of the template for T = string
template <> class Hash<string> {
public:
    int operator()(const string& x) const {
        int s = 0;
        for (int i = 0; i < x.length(); ++i)
            s = s * 37 + x[i];
        return s;
    };

// specialization of the template for T = int
template <> class Hash<int> {
    static long const MULT = 31415926;
public:
    int operator()(const int& x) const {
        long y = ((x * x * MULT) << 20) >> 4;
        return y;
    };
};
```

Other sophisticated hash functions use weighted sums or non-linear transformations (e.g., they square the number represented by the  $k$  central bits of the key).

# Collision Resolution

Collision resolution strategies can be grouped into two main families. By historical reasons (not very logically) they are called

- **Open hashing:** separate chaining, coalesced hashing, ...
- **Open addressing:** linear probing, double hashing, quadratic hashing, Idots

# Collision Resolution

Collision resolution strategies can be grouped into two main families. By historical reasons (not very logically) they are called

- Open hashing: **separate chaining**, coalesced hashing, ...
- Open addressing: **linear probing**, double hashing, quadratic hashing, Idots

# Separate Chaining

In separate chaining, each slot in the hash table has a pointer to a linked list of synonyms.

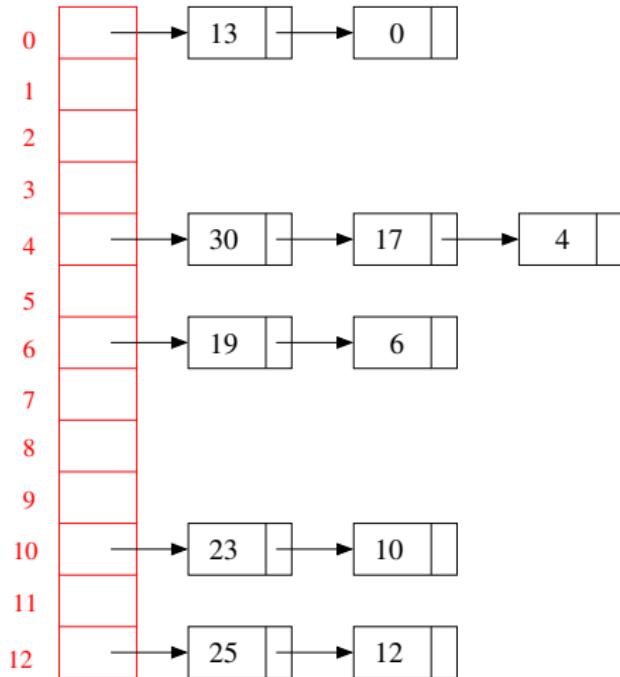
```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
    ...
private:
    struct node {
        Key _k;
        Value _v;
        node* _next;
        // constructor for class node
        node(const Key& k, const Value& v,
              node* next = nullptr);
    };
    node** _Thash; // array of pointers to linked lists of synonyms
    int _M;         // capacity of the table
    int _n;         // number of elements
    double _alpha_max; // max. load factor

    node* lookup_sep_chain(const Key& k, int i) const ;
    void insert_sep_chain(const Key& k,
                          const Value& v);
    void remove_sep_chain(const Key& k) ;
};
```

# Separate Chaining

$$M = 13 \quad X = \{ 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30 \}$$

$$h(x) = x \bmod M$$



## Separate Chaining

For insertions, we access the appropriate linked list using the hash function, and scan the list to find out whether the key was already present or not. If present, we modify the associated value; if not, a new node with the pair  $\langle key, value \rangle$  is added to the list.

Since the lists contain very few elements each, the simplest and more efficient solution is to add elements to the front. There is no need for double links, sentinels, etc. Sorting the lists or using some other sophisticated data structure instead of linked lists does not report real practical benefits.

## Separate Chaining

Searching is also simple: access the appropriate linked list using the hash function and sequentially scan it to locate the key or to report unsuccessful search.

# Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert(const Key& k,
                                              const Value& v) {
    insert_sep_chain(k, v);
    if (_n / _M > _alpha_max)
        // the current load factor is too large, raise here an exception or
        // resize the table and rehash
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert_sep_chain(
    const Key& k, const Value& v) {
    int i = hash(k);
    node* p = lookup_sep_chain(k, i);
    // insert as first item in the list
    // if not present
    if (p == nullptr) {
        _Thash[i] = new node(k, v, _Thash[i]);
        ++_n;
    }
    else
        p -> _v = v;
}
```

# Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::lookup(const Key& k,
                                              bool& exists, Value& v) const {
    node* p = lookup_sep_chain(k, hash(k));
    if (p == nullptr)
        exists = false;
    else {
        exists = true;
        v = p -> _v;
    }
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
Dictionary<Key,Value,HashFunct>::node*
Dictionary<Key,Value,HashFunct>::lookup_sep_chain(const Key& k,
                                                int i) const {
    node* p = _Thash[i];
    // sequential search in the i-th list of synonyms
    while (p != nullptr and p -> _k != k)
        p = p -> _next;

    return p;
}
```

# Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value, HashFunct>::remove(const
    Key& k)  {

    remove_sep_chain(k);
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value, HashFunct>::remove_sep_chain
(const Key& k)  {

    int i = hash(k);
    node* p = _Thash[i];
    node* prev = nullptr; // predecessor of p

    // sequential search in the i-th list of synonyms
    while (p != nullptr and p -> _k != k) {
        prev = p;
        p = p -> _next;
    }

    // if p != nullptr, remove from the
    // i-th list; check whether p is the first
    // element in the list or not
    if (p != nullptr) {
        --_n;
        if (prev != nullptr)
            prev -> _next = p -> _next;
        else
            _Thash[i] = p -> _next;
        delete p;
    }
}
```

## The Cost of Separate Chaining

Let  $n$  be the number of elements stored in the hash table. On average, each linked list contains  $\alpha = n/M$  elements and the cost of lookups (either successful or unsuccessful), of insertions and of deletions will be proportional to  $\alpha$ . If  $\alpha$  is a small constant value then the cost of all basic operations is, on average,  $\Theta(1)$ . However, it can be shown that the expected length of the largest synonym list is  $\Theta(\log n)$ .

The value  $\alpha$  is called **load factor**, and the performance of the hash table will be dependent on it.

## Open Hashing

Other variants of open hashing (e.g., coalesced hashing) store the synonyms in a particular area of the hash table, often called the cellar. The positions corresponding to the cellar are not addressable, that is, the hash function never maps a key into that area.

# Open Addressing

In **open addressing**, synonyms are stored in the hash table. Searches and insertions probe a sequence of positions until the given key or an empty slot is found. The sequence of probes starts in position  $i_0 = h(k)$  and continues with  $i_1, i_2, \dots$ . The different open addressing strategies use different rules to define the sequence of probes. The simplest one is **linear probing**:

$$i_1 = i_0 + 1, i_2 = i_1 + 1, \dots,$$

taking modulo  $M$  in all cases.

# Linear Probing

```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>

class Dictionary {
    ...
private:
    struct node {
        Key _k;
        Value _v;
        bool _free;
        // constructor for class node
        node(const Key& k, const Value& v, bool libre = true);
    };
    node* _hash; // array of nodes
    int _M; // capacity of the table
    int _n; // number of elements
    double _alpha_max; // max. load factor (must be < 1)

    int lookup_linear_probing(const Key& k) const ;
    void insert_linear_probing(const Key& k,
                               const Value& v);
    void remove_linear_probing(const Key& k) ;
};
```

# Linear Probing

M = 13

X = { 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30 }

$h(x) = x \bmod M$  (incremento 1)

0	0
1	
2	
3	
4	4
5	
6	6
7	
8	
9	
10	10
11	
12	12



0	0	occupied
1	13	occupied
2		free
3		free
4	4	occupied
5	17	occupied
6	6	occupied
7	19	occupied
8		free
9		free
10	10	occupied
11	23	occupied
12	12	occupied

0	0	occupied
1	13	occupied
2	25	occupied
3		free
4	4	occupied
5	17	occupied
6	6	occupied
7	19	occupied
8	30	occupied
9		free
10	10	occupied
11	23	occupied
12	12	occupied

+ { 0, 4, 6, 10, 12 }

+ { 13, 17, 19, 23 }

+ { 25, 30 }

# Linear Probing

```
// This method is only used if there is at least a
// free slot in the table: _n < _M
template <typename Key, typename Value,
          template <typename> class HashFunct>
void
Dictionary<Key,Value,HashFunct>::insert_linear_probing(
    const Key& k, const Value& v) {

    int i = hash(k);
    while (not _Thash[i]._free and _Thash[i]._k != k)
        i = (i + 1) % _M;
    _Thash[i]._k = k; _Thash[i]._v = v;
    if (_Thash[i]._free) ++_n;
    _Thash[i]._free = false;
}
```

# Linear Probing

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup(
    const Key& k,
    bool& exists, Value& v) const {
    int i = lookup_linear_probing(k);

    if (not _Thash[i]._free and _Thash[i]._k == k) {
        exists = true;
        v = _Thash[i]._v;
    }
    else
        exists = false;
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup_linear_probing(
    const Key& k) const {

    int i = hash(k);
    int visited = 0; // this is only necessary if
                     // _n == _M, otherwise there is at least
                     // a free position

    while (not _Thash[i]._free and _Thash[i]._k != k
           and visited < _M) {
        ++visited;
        i = (i + 1) % _M;
    }
    return i;
}
```

## Deletions in Open Addressing

There is no general solution for true deletions in open addressing tables. It is not enough to mark the position of the element to be removed as “free”, since later searches might report as not present some element which is stored in the table.

The general technique that can be used is **lazy deletions**. Each slot can be free, occupied or **deleted**. Deleted slots can be used to store there a new element, but they are not free and searches must pass them over and continue.

# Deletions in Linear Probing

For linear probing, we can do true deletions. The deletion algorithm must continue probing the positions after the removed element, and moving to the emptied slot any element whose hash address is equal (or smaller in the cyclic order) to the address of the emptied slot. Moving an element creates a new emptied slot, and the procedure is repeated until an empty slot is found. In our implementation we will use the function `displ(j, i)` which gives us the distance between  $j$  e  $i$  in the cyclic order: if  $j > i$  we must turn around position  $M - 1$  and go back to position 0.

```
int displ(j, i, M) {
    if (i >= j)
        return i - j;
    else
        return M + (i - j);
}
```

# Linear Probing

```
// we assume _n < _M

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::remove_linear_probing(
    const Key& k) const {
    int i = lookup_linear_probing(k);

    if (not _Thash[i]._free) {
        // _Thash[i] is the element to remove
        int free = i; i = (i + 1) % _M; int d = 1;
        while (not _Thash[i]._free) {
            int i_home = hash(_Thash[i]._k);
            if (displ(i_home, i, _M) >= d) {
                _Thash[free] = _Thash[i]; free = i; d = 0;
            }
            i = (i + 1) % _M; ++d;
        }
        _Thash[free]._free = true; --_n;
    }
}
```

# The Cost of Linear Probing

Besides simplicity, linear probing offers several advantages since synonyms tend to group into consecutive positions; this is particularly useful when the hash table is stored in external memory, in combination with some **bucketing** technique: each slot of the table stores up to  $B$  elements.

On the other hand linear probing suffers the problem of **clustering** and its performance rapidly degrades as the loading factor  $\alpha = n/M$  approaches 1.

## Linear Probing

For  $\alpha < 1$  the cost of successful searches (and of modifications of the value associated to a key) is proportional to

$$\frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right),$$

whereas the cost of unsuccessful searches, insertions and deletions is proportional to

$$\frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right).$$

# The Cost of Linear Probing

The phenomenon called **clustering** appears in open addressing strategies when an element does not occupy its “preferred” location due to the presence of some element which is not a synonym (so it is not the “preferred” location of that element either).

As more and more new elements are inserted and the load factor  $\alpha \rightarrow 1$ , groups of synonyms merge into big **clusters**; when we lookup for some key we have not only to examine its synonyms but perhaps some large number of unrelated elements.

# Rehashing

Many programming languages allow the creation of arrays giving their size in execution time. Then we can use **resizing** to avoid high load factors. When a new insertion makes the load factor larger than some fixed threshold, a new array that roughly doubles the size of the current array is created (claiming the storage from dynamic memory). The contents of the old array are inserted (“rehash”) into the new array, and the memory used for the old array is freed (released back to the dynamic memory).

# Rehashing

**Rehashing** is a costly operation, with time proportional to  $n$ . But we need to do it only from time to time, the frequency exponentially decaying with the size of the table. Rehashing allows us to grow a dictionary without imposing a maximum number of elements, while keeping a reasonable load factor. The technique can be used in reverse, to avoid very low load factors (which mean a substantial waste of memory).

C++ vectors have the `resize` method, that does exactly this (allocate new array, copy contents, release old array). It can be used to implement hash tables, but we will have to take care of the rehashing anyway.

# Rehashing

It can be proved that, although a single insertion/deletion might have expected cost  $\Theta(n)$  (because of the rehashing), any sequence of  $n$  operations will have expected total cost  $\mathcal{O}(n)$ . We can say then that the  $n$  operations have expected **amortized cost**  $\mathcal{O}(1)$  each.

# Part IV

## Priority Queues

- 12 Priority Queues
- 13 Heapsort

# Introduction

A **priority queue** (cat: *cua de prioritat*; esp: *cola de prioridad*) stores a collection of elements, each one endowed with a value called its **priority**.

Priority queues support the insertion of new elements and the query and removal of an element of minimum (or maximum) priority.

# Introduction

```
template <typename Elemt, typename Prio>
class PriorityQueue {
public:
    ...
    // Adds an element x with priority p to the priority queue.
    void insert(cons Elemt& x, const Prio& p);

    // Returns an element of minimum priority; throws an
    // exception if the queue is empty.
    Elemt min() const;

    // Returns the priority of an element of minimum priority; throws an
    // exception if the queue is empty.
    Prio min_prio() const;

    // Removes an element of minimum priority; throws an
    // exception if the queue is empty.
    void remove_min();

    // Returns true iff the priority queue is empty
    bool empty() const;
};
```

# Introduction

```
// We have two arrays Weight and Symb with the atomic
// weights and the symbols of n chemical elements, e.g.,
// Symb[i] = "C" y Weight[i] = 12.2, for some i.
// We use a priority queue to sort the information in alphabetic
// ascending order

PriorityQueue<double, string> P;
for (int i = 0; i < n; ++i)
    P.insert(Weigth[i], Symb[i]);
int i = 0;
while(not P.empty()) {
    Weight[i] = P.min();
    Symb[i] = P.min_prio();
    ++i;
    P.remove_min();
}
```

## Introduction

We can use the  $k$ th smallest element in an unsorted array. Insert the first  $k$  elements in a max-priority queue. For each remaining element, compare the current element with the maximum element in the PQ; if it is larger or equal then continue, if it is smaller, remove the maximum and insert the current element in the PQ.

The priority queue keeps, after the first  $k$  insertions, the  $k$  smallest elements seen so far in the sequence. Its maximum element is the  $k$ th smallest element.

# Introduction

- Several techniques that we have seen for the implementation of dictionaries can also be used for priority queues (not hash tables).
- For instance, AVLs can be used to implement a PQ with cost  $\mathcal{O}(\log n)$  for insertions and deletions

# Heaps

## Definition

A **heap** is a binary tree such that

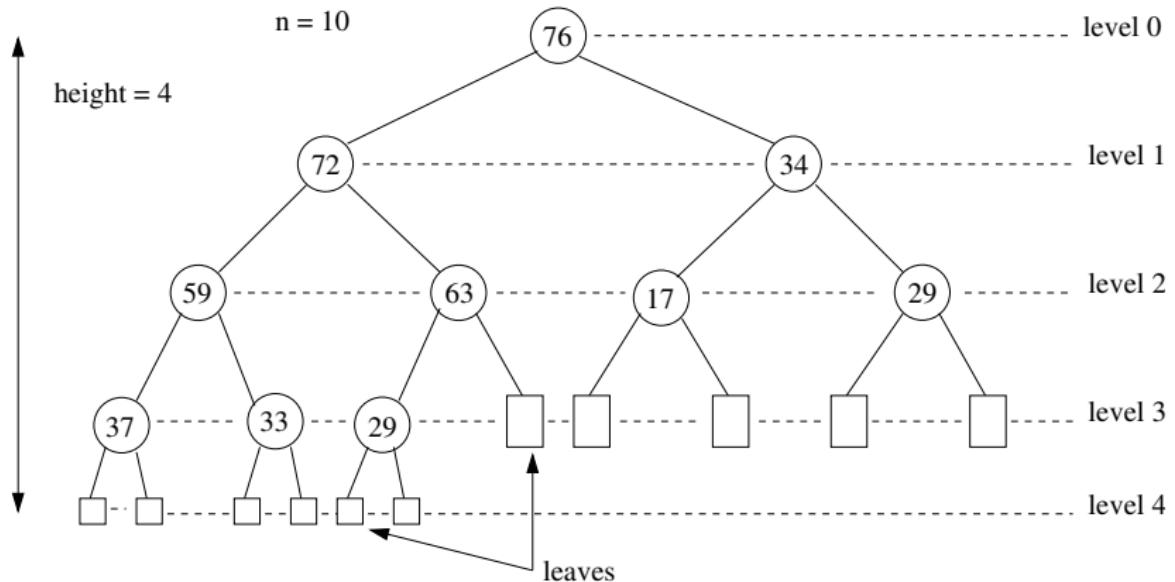
- 1 All empty subtrees are located in the last two levels of the tree.
- 2 If a node has an empty left subtree then its right subtree is also empty.
- 3 The priority of any element is larger or equal than the priority of any element in its descendants.

# Heaps

Because of properties 1–2 in the definition, a heap is a **quasi-complete** binary tree. Property #3 is called **heap order** (for **max-heaps**).

If the priority of an element is smaller or equal than that of its descendants then we talk about **min-heaps**.

# Heaps



# Heaps

## Proposition

- 1 *The root of a max-heap stores an element of maximum priority.*
- 2 *A heap of size  $n$  has height*

$$h = \lceil \log_2(n + 1) \rceil.$$

If heaps are used to implement a PQ the query for a max/min element and its priority is trivial: we need only to examine the root of the heap.

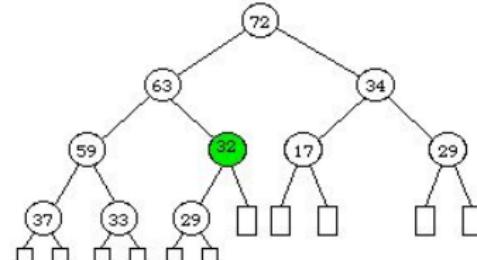
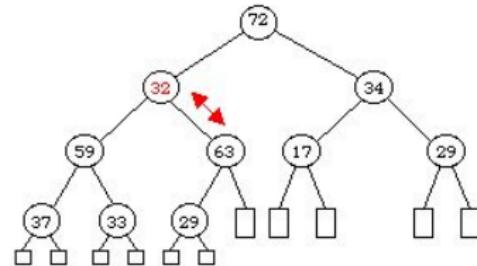
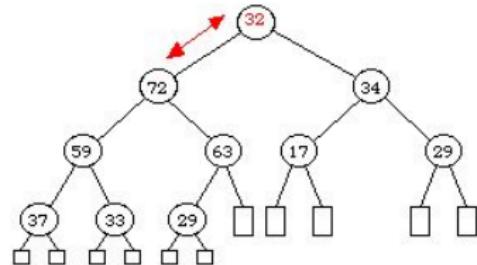
## Heaps: Removing the maximum

- 1 Replace the root of the heap with the last element (the rightmost element in the last level)
- 2 Reestablish the invariant (heap order) **sinking** the root:  
The function `sink` exchanges a given node with its largest priority child, if its priority is smaller than the priority of its child, and repeats the same until the heap order is reestablished.

## Heaps: Removing the maximum

- 1 Replace the root of the heap with the last element (the rightmost element in the last level)
- 2 Reestablish the invariant (heap order) **sinking** the root:  
The function `sink` exchanges a given node with its largest priority child, if its priority is smaller than the priority of its child, and repeats the same until the heap order is reestablished.

## Heaps: Removing the maximum



# Heaps: Adding a new element

- 1 Add the new element as rightmost node of the last level of the heap (or as the first element of a new deeper level)
- 2 Reestablish the heap order *sifting up* (a.k.a. *floating*) the new added element:  
The function `siftup` compares the given node to its father, and they are exchanged if its priority is larger than that of its father; the process is repeated until the heap order is reestablished.

## Heaps: Adding a new element

- 1 Add the new element as rightmost node of the last level of the heap (or as the first element of a new deeper level)
- 2 Reestablish the heap order **sifting up** (a.k.a. *floating*) the new added element:  
The function `siftup` compares the given node to its father, and they are exchanged if its priority is larger than that of its father; the process is repeated until the heap order is reestablished.

## The Cost of Heaps

Since the height of a heap is  $\Theta(\log n)$ , the cost of removing the maximum and the cost of insertions is  $\mathcal{O}(\log n)$ .

We can implement heaps with dynamically allocated nodes, and three pointers per node (left, right, father) ... But it is much easier and efficient to implement heaps with vectors!

Since the heap is a quasi-complete binary tree this allows us to avoid wasting memory: the  $n$  elements are stored in the first  $n$  components of the vector, which implicitly represent the tree.

# Implementing Heaps

To make the rules easier we will use a vector  $A$  of size  $n + 1$  and discard  $A[0]$ . Resizing can be used to allow unlimited growth.

- ➊  $A[1]$  contains the root
- ➋ If  $2i \leq n$  then  $A[2i]$  contains the left child of  $A[i]$  and if  $2i + 1 \leq n$  then  $A[2i + 1]$  contains the right subtree of  $A[i]$
- ➌ If  $i \geq 2$  then  $A[i/2]$  contains the father of  $A[i]$

# Implementing Heaps

To make the rules easier we will use a vector  $A$  of size  $n + 1$  and discard  $A[0]$ . Resizing can be used to allow unlimited growth.

- 1  $A[1]$  contains the root
- 2 If  $2i \leq n$  then  $A[2i]$  contains the left child of  $A[i]$  and if  $2i + 1 \leq n$  then  $A[2i + 1]$  contains the right subtree of  $A[i]$
- 3 If  $i \geq 2$  then  $A[i/2]$  contains the father of  $A[i]$

# Implementing Heaps

To make the rules easier we will use a vector  $A$  of size  $n + 1$  and discard  $A[0]$ . Resizing can be used to allow unlimited growth.

- 1  $A[1]$  contains the root
- 2 If  $2i \leq n$  then  $A[2i]$  contains the left child of  $A[i]$  and if  $2i + 1 \leq n$  then  $A[2i + 1]$  contains the right subtree of  $A[i]$
- 3 If  $i \geq 2$  then  $A[i/2]$  contains the father of  $A[i]$

# Implementing Heaps

```
template <typename Ele, typename Prio>
class PriorityQueue {
public:
    ...
private:
    // Component of index 0 is not used
    vector<pair<Ele, Prio> > h;
    int nelems;

    void siftup(int j) throw();
    void sink(int j) throw();
};
```

# Implementing Heaps

```
template <typename Elemt, typename Prio>
bool PriorityQueue<Elemt,Prio>::empty() const {
    return nelems == 0;
}

template <typename Elemt, typename Prio>
Elemt PriorityQueue<Elemt,Prio>::min() const {
    if (nelems == 0) throw EmptyPriorityQueue;
    return h[1].first;
}

template <typename Elemt, typename Prio>
Prio PriorityQueue<Elemt,Prio>::min_prio() const {
    if (nelems == 0) throw EmptyPriorityQueue;
    return h[1].second;
}
```

# Implementing Heaps

```
template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::insert(cons Elemt& x,
                                         cons Prio& p) {
    ++nelems;
    h.push_back(make_pair(x, p));
    siftup(nelems);
}

template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::remove_min() const {
    if (nelems == 0) throw EmptyPriorityQueue;
    swap(h[1], h[nelems]);
    --nelems;
    h.pop_back();
    sink(1);
}
```

# Implementing Heaps

```
// Cost: O(log(n/j))
template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::sink(int j) {

    // if j has no left child we are at the last level
    if (2 * j > nelems) return;

    int minchild = 2 * j;
    if (minchild < nelems and
        h[minchild].second > h[minchild + 1].second)
        ++minchild;

    // minchild is the index of the child with minimum priority
    if (h[j].second > h[minchild].second) {
        swap(h[j], h[minchild]);
        sink(minchild);
    }
}
```

# Implementing Heaps

```
// Cost: O(log j)
template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::siftup(int j) {

    // if j is the root we are done
    if (j == 1) return;

    int father = j / 2;
    if (h[j].second < h[father].second) {
        swap(h[j], h[father]);
        siftup(father);
    }
}
```

# Part IV

## Priority Queues

12

Priority Queues

13

Heapsort

# Heapsort

Heapsort (Williams, 1964) sorts an array of  $n$  elements building a heap with the  $n$  elements and extracting them, one by one, from the heap (cif. our example of the atomic weights and chemical symbols).

The originally given array is used to build the heap; heapsort works **in-place** and only some constant auxiliary memory space is needed.

Since insertions and deletions in heaps have cost  $\mathcal{O}(\log n)$  the cost of the algorithm is  $\mathcal{O}(n \log n)$ .

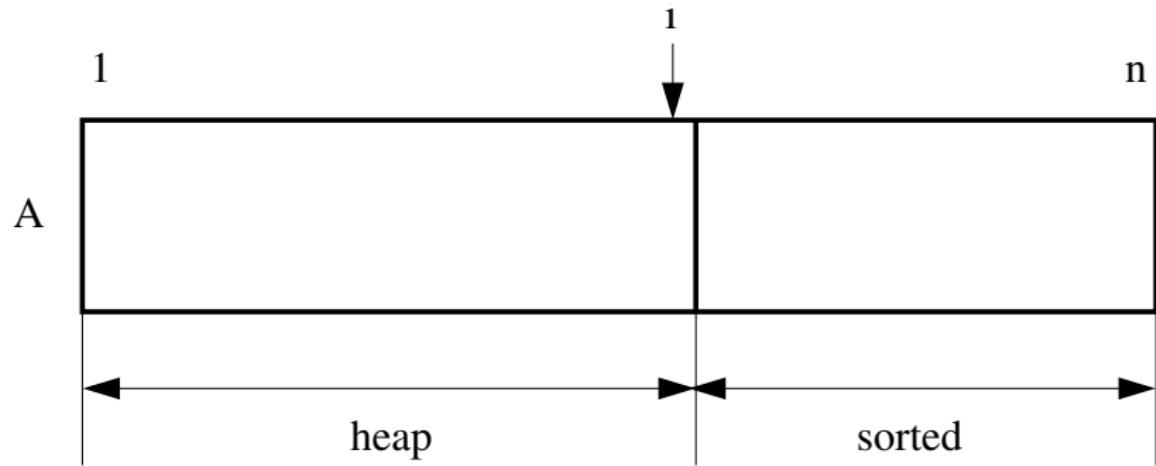
# Heapsort

```
template <typename Elem>
void sink(Elem v[], int sz, int pos);

// Sort v[1..n] in ascending order
// (v[0] is unused)
template <typename Elem>
void heapsort(Elem v[], int n) {
    make_heap(v, n);
    for (int i = n; i > 0; --i) {
        // extract largest element from the heap
        swap(v[1], v[i]);

        // sink the root to reestablish max-heap order
        // in the subarray v[1..i-1]
        sink(v, i-1, 1);
    }
}
```

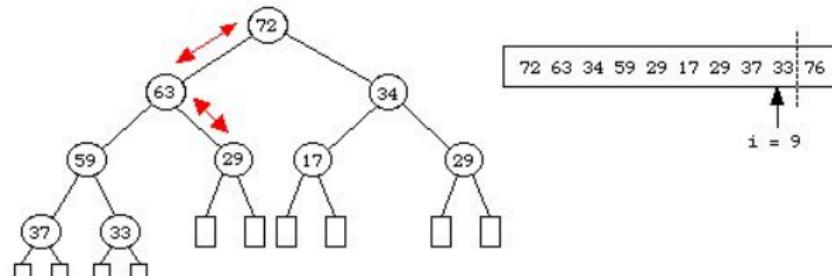
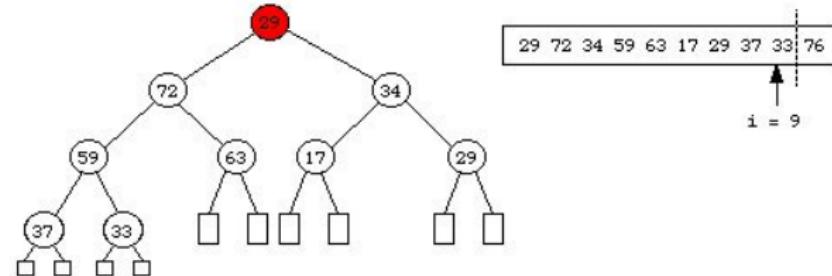
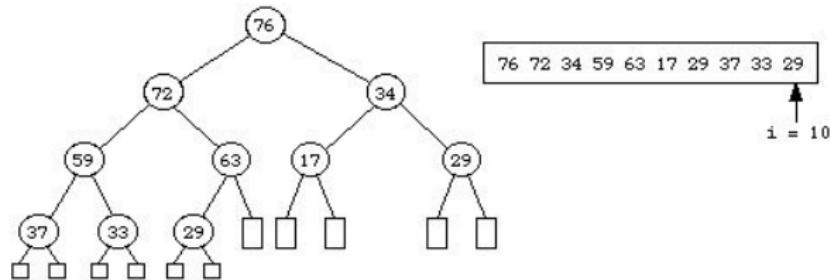
# Heapsort



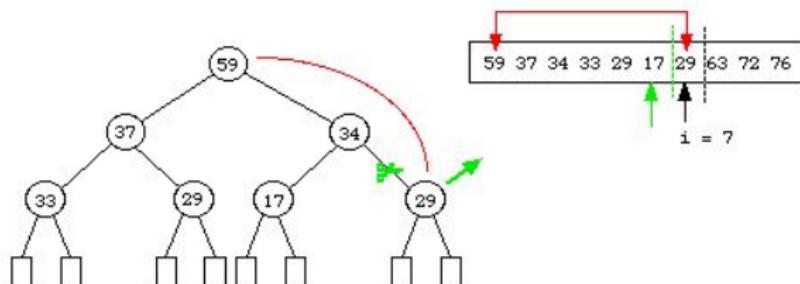
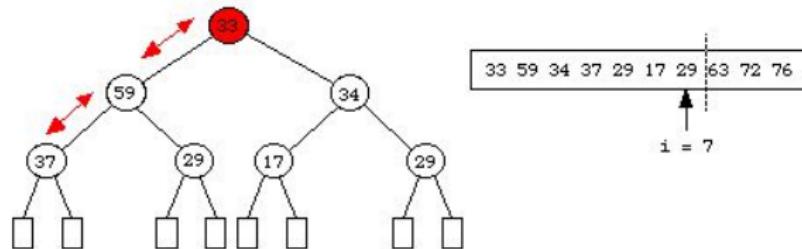
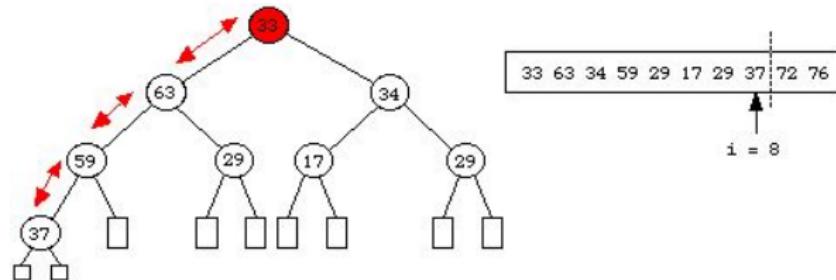
$$A[1] \leq A[i+1] \leq A[i+2] \leq \dots \leq A[n]$$

$$A[1] = \max_{1 \leq k \leq i} A[k]$$

# Heapsort



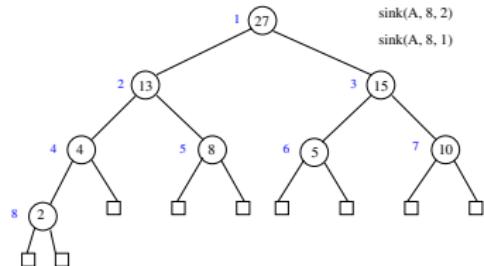
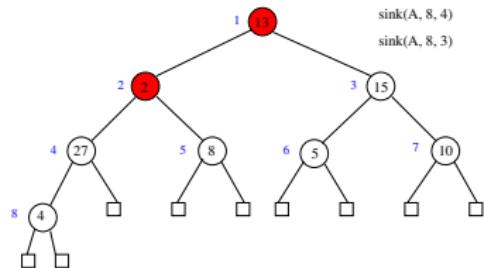
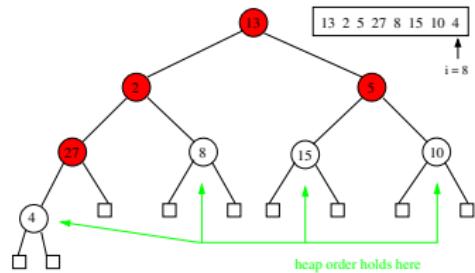
# Heapsort



# Heapify

```
// Establish (max) heap order in the
// array v[1..n] of Elem's; Elem == priorities
// this is a.k.a. as heapify
template <typename Elemt>
void make_heap(Elemt v[], int n) {
    for (int i = n/2; i > 0; --i)
        sink(v, n, i);
}
```

# Heapify



# The Cost of Heapsort

Let  $H(n)$  be the worst-case cost of `heapsort` and  $B(n)$  the cost `make_heap`. Since the worst-case cost of `sink(v, i - 1, 1)` is  $\mathcal{O}(\log i)$  we have

$$\begin{aligned} H(n) &= B(n) + \sum_{i=1}^{i=n} \mathcal{O}(\log i) \\ &= B(n) + \mathcal{O}\left(\sum_{1 \leq i \leq n} \log_2 i\right) \\ &= B(n) + \mathcal{O}(\log(n!)) = B(n) + \mathcal{O}(n \log n) \end{aligned}$$

A rough analysis of  $B(n)$  shows that  $B(n) = \mathcal{O}(n \log n)$  since it makes  $\Theta(n)$  calls to `sink`, each one with cost  $\mathcal{O}(\log n)$ . Hence,  $H(n) = \mathcal{O}(n \log n)$ ; actually,  $H(n) = \Theta(n \log n)$  in any case if all elements are different.

# The Cost of Heapify

A refined analysis of  $B(n)$ :

$$\begin{aligned} B(n) &= \sum_{1 \leq i \leq \lfloor n/2 \rfloor} \mathcal{O}(\log(n/i)) \\ &= \mathcal{O}\left(\log \frac{n^{n/2}}{(n/2)!}\right) \\ &= \mathcal{O}\left(\log(2e)^{n/2}\right) = \mathcal{O}(n) \end{aligned}$$

Since  $B(n) = \Omega(n)$ , we conclude  $B(n) = \Theta(n)$ .

## The Cost of Heapify

Alternative proof: Let  $h = \lceil \log_2(n + 1) \rceil$  the height of the heap. Level  $h - 1 - k$  contains at most

$$2^{h-1-k} < \frac{n+1}{2^k}$$

elements; in the worst-case each one will sink down to level  $h - 1$  with cost  $\mathcal{O}(k)$

# The Cost of Heapify

$$\begin{aligned}B(n) &= \sum_{0 \leq k \leq h-1} \mathcal{O}(k) \frac{n+1}{2^k} \\&= \mathcal{O}\left(n \sum_{0 \leq k \leq h-1} \frac{k}{2^k}\right) \\&= \mathcal{O}\left(n \sum_{k \geq 0} \frac{k}{2^k}\right) = \mathcal{O}(n),\end{aligned}$$

since

$$\sum_{k \geq 0} \frac{k}{2^k} = 2.$$

In general, if  $0 < |r| < 1$ ,

$$\sum_{k \geq 0} k \cdot r^k = \frac{r}{(1-r)^2}.$$

# The Cost of Heapify

Despite  $H(n) = \Theta(n \log n)$ , the refined analysis of  $B(n)$  is important: using a *min-heap* we can get the smallest  $k$  elements in an array in ascending order with cost:

$$\begin{aligned} S(n, k) &= B(n) + k \cdot \mathcal{O}(\log n) \\ &= \mathcal{O}(n + k \log n). \end{aligned}$$

If  $k = \mathcal{O}(n / \log n)$  then  $S(n, k) = \mathcal{O}(n)$ .

# Part V

## Graphs

14

Graphs

15

Depth-First Search (DFS)

16

Breadth-First Search (BFS)

17

Shortest Paths: Dijkstra's Algorithm

18

Minimum Spanning Trees: Prim's Algorithm

# Basic Graph Theory

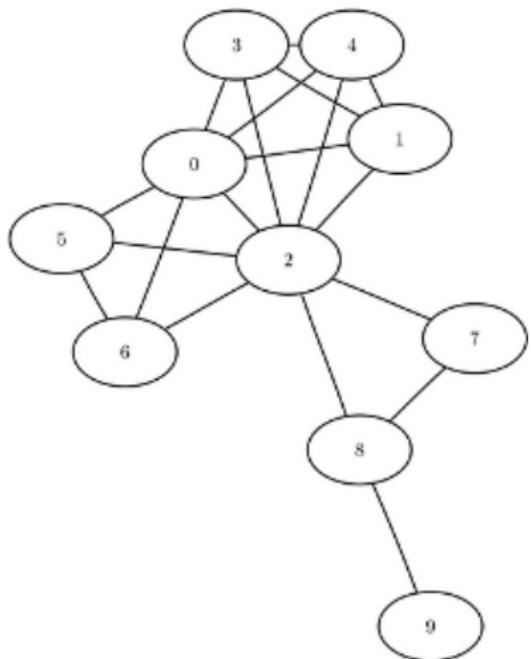
## Definition

An (undirected) graph is a pair  $G = \langle V, E \rangle$  where  $V$  is a finite set of vertices (a.k.a. nodes) and  $E$  is a set of edges; each edge  $e \in E$  is an unordered pair  $\{u, v\}$  with  $u \neq v$  and  $u, v \in V$ .

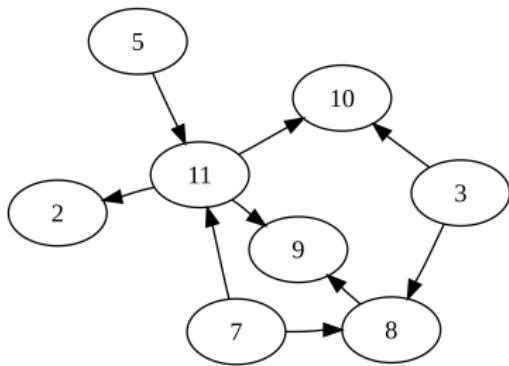
## Definition

A directed graph or digraph is a pair  $G = \langle V, E \rangle$  where  $V$  is the finite set of vertices and  $E$  is a set of arcs (but sometimes we will call them edges); each arc  $e \in E$  is a pair  $(u, v)$  with  $u \neq v$  and  $u, v \in V$ .

# Basic Graph Theory

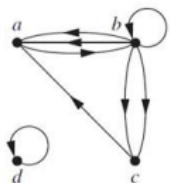


Undirected graph



Directed graph

# Basic Graph Theory



When we have a multiset of arcs or edges instead of a set, and arcs or edges with both extremes identical (**loops**) are allowed, we talk about **multigraphs** (directed or undirected).

For an arc  $e = (u, v)$ , vertex  $u$  is called the **source** and a vertex  $v$  the **target**. We say that  $v$  is a **successor** of  $u$ ; conversely,  $u$  is a predecessor of  $v$ . For an edge  $e = \{u, v\}$ , the vertices are called its **extremes** and we say  $u$  and  $v$  are **adjacent**. We also say that the edge  $e$  is **incident** to  $u$  and  $v$ .

# Basic Graph Theory

The number of edges incident to a vertex  $u$  (equivalently, the number of vertices  $v$  adjacent to  $u$ ) is called the **degree** of  $u$  ( $\deg(u)$ ).

For digraphs, we define the **in-degree** and **out-degree** of a vertex  $u$ , the number of predecessors and of successors, respectively, of the vertex  $u$ , and we note these as  $\text{in-deg}(u)$  and  $\text{out-deg}(u)$ .

## Lemma (Handshaking Lemma)

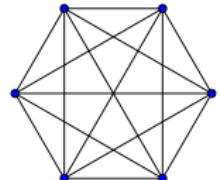
For any digraph  $G = \langle V, E \rangle$

$$\sum_{v \in V} \text{in-deg}(v) = \sum_{v \in V} \text{out-deg}(v) = |E|$$

For any graph  $G = \langle V, E \rangle$

$$\sum_{v \in V} \deg(v) = 2 \cdot |E|$$

# Basic Graph Theory



For any graph  $G$  the number of edges  $|E| = m$  is between 0 and

$$\binom{n}{2} = \frac{n(n - 1)}{2}$$

where  $n = |V|$ . A graph with  $m = n(n - 1)/2$  is called **complete** and often denoted  $K_n$ . For digraphs, the number of arcs  $m$  must be between 0 and  $n(n - 1)$ .

Given a family of graphs, the family is **dense** if for any member  $m = \Theta(n^2)$ ; otherwise it is called **sparse**. For instance, the family of complete graphs is dense, and the family of cyclic graphs and the family of  $d$ -regular graphs (with  $d$  a constant) is sparse.

# Basic Graph Theory

## Definition

A **path**  $P = v_0, v_1, v_2, \dots, v_n$  of length  $n$  in a graph  $G = \langle V, E \rangle$  is a sequence of  $n + 1$  vertices from  $G$  such that for all  $i$ ,  $0 \leq i < n$

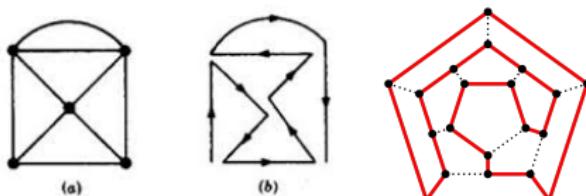
$$\{v_i, v_{i+1}\} \in E$$

Vertex  $v_0$  is called the **origin** of  $P$  and  $v_n$  the **target**.

(Directed) paths in digraphs are defined in the same way: for all  $i$ ,  $(v_i, v_{i+1})$  is an arc in the digraph.

A path is called **simple** if no vertex appears more than once in the sequence, except possibly  $v_0$  and  $v_n$ . If  $v_0 = v_n$  the path is called a **cycle**.

# Basic Graph Theory



A simple path that “visits” all vertices of the graph is a **Hamiltonian path** (or cycle, if closed). A graph is **Hamiltonian** if and only if contains at least a Hamiltonian path.

A path that contains all edges/arcs of the graph is an **Eulerian path**. A graph is **Eulerian** if and only if contains an Eulerian path.

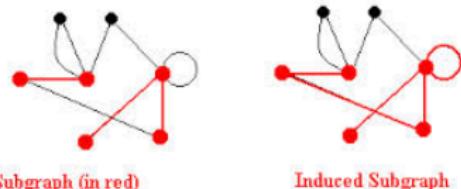
# Basic Graph Theory

## Definition

A graph  $G = \langle V, E \rangle$  is **connected** if and only if there exists a path in  $G$  from  $u$  to  $v$  for all pairs of vertices  $u, v \in V$ .

The definition for digraphs is the same, but we will say that the digraph is **strongly connected**.

# Basic Graph Theory



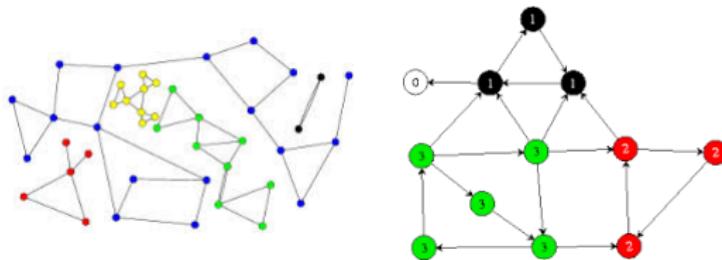
## Definition

Given a graph  $G = \langle V, E \rangle$ , the graph  $H = \langle V', E' \rangle$  is a **subgraph** of  $G$  if and only if  $V' \subseteq V$ ,  $E' \subseteq E$ , and for all edges  $e' = \{u', v'\} \in E'$  both  $u'$  and  $v'$  belong to  $V'$ .

If  $E'$  contains all edges in  $E$  which are incident to two vertices in  $V'$ , the subgraph  $H$  is called the subgraph **induced** by  $V'$ . If  $V' = V$  then  $H$  is called a **spanning subgraph** of  $G$ .

Subgraphs of digraphs are defined in a completely analogous way.

# Basic Graph Theory

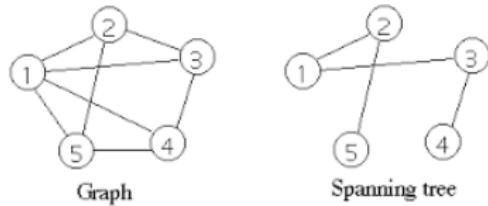


## Definition

A **connected component**  $C$  of a graph  $G$  is a maximal connected induced subgraph of  $G$ . By *maximal* we mean that adding any vertex  $v$  to  $V(C)$  the resulting induced subgraph is not connected.

For digraphs, the analogous concept is that of **strongly connected components**.

# Basic Graph Theory



## Definition

A connected acyclic (that is, with no cycles) graph is a **(free) tree**. If  $G$  is a connected graph, a spanning subgraph  $T = \langle V, E' \rangle$  which is a tree is a **spanning tree**. An acyclic graph is a **forest** and each of its connected components is a tree. If a spanning induced subgraph is acyclic, then it is a **spanning forest** and each of its connected components is a spanning tree for the corresponding connected component of  $G$ .

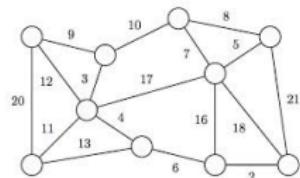
# Basic Graph Theory

Trees (in the graph-theoretic sense) have no root and there is no order among the adjacent vertices to a given vertex in the tree.

## Lemma

*If  $G = \langle V, E \rangle$  is a tree then  $|E| = |V| - 1$ . If  $G$  is a forest with  $c$  connected components (trees) then  $|E| = |V| - c$ .*

# Basic Graph Theory



We will often consider graphs or digraphs in which each edge (arc) bears a **label**. A labelling in a graph  $G$  is a function  $\omega : E \rightarrow \mathcal{L}$  from the set of edges  $E$  to the (potentially infinite) set of labels  $\mathcal{L}$ .

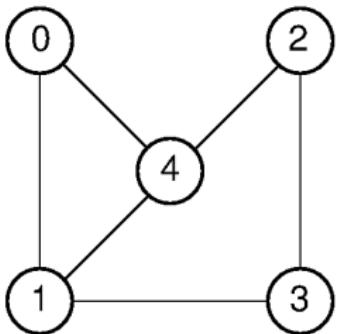
The most common case is that in which labels are numeric (integers, rational, real numbers); labelled graphs are then called **weighted graphs** (and the label  $\omega(e)$  of an edge  $e$  is called its **weight**).

# Implementation of Graphs

The two most frequent ways to implement a graph are:

- 1 Using **adjacency matrices**: entry  $A[i][j]$  of the adjacency matrix  $A$  is a Boolean indicating whether  $(i, j)$  is an edge/arc in  $E(G)$  or not. For weighted (di)graph  $A[i][j]$  stores the label assigned to the edge/arc  $(i, j)$ .
- 2 Using **adjacency lists**: we use an array or vector  $T$  such that for a vertex  $u$ ,  $T[u]$  points to a list of the edges incident to  $u$  or a list of the vertices  $v \in V$  which are adjacent to  $u$ . For digraphs, we will have the list of the successors of a vertex  $u$ , for all vertices  $u$ , and, possibly, the list of predecessors of a vertex  $u$ , for all vertices  $u$ .

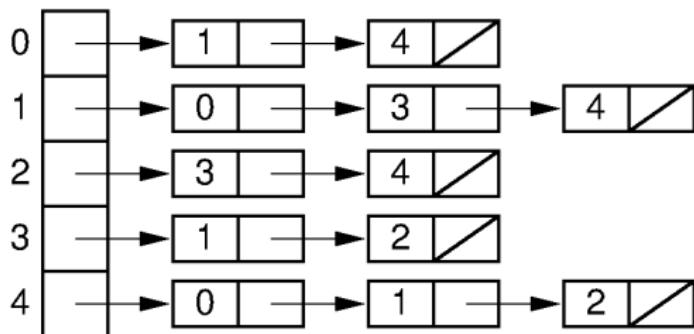
# Implementation of Graphs



(a)

0	1	2	3	4
1			1	1
2	1			1
3		1	1	
4	1	1	1	

(b)



(c)

# Implementation of Graphs

Adjacency matrices is usually too costly in space; if  $|V(G)| = n$  it needs space  $\Theta(n^2)$  to represent the graph, independent of the number of edges/arcs in the graph. It is fine to reperesent dense graphs or when we need to efficiently answer whether an edge  $(u, v) \in E$  or not.

As a general rule, the preferred implementation will be adjancency lists. They require space  $\Theta(n + m)$  where  $n = |V|$  and  $m = |E|$ ; the space is thus linear in the size of the graph.

# Implementation of Graphs

```
typedef int vertex;
typedef pair<vertex, vertex> edge;

class Graph {
    // undirected graphs with V = {0, ..., n-1}
public:
    // create an empty graph (no vertices and no edges)
    Graph();
    // create an empty graph with n vertices (but no edges)
    Graph(int n);
    // adds a new vertex to the graph; the new vertex will have identifier
    // 'n' where n is the number of vertices in the graph, before adding the
    // the new vertex
    void add_vertex();
    // adds edge (u,v) to the graph, either giving an edge e=(u,v) or its
    // extremes u and v
    void add_edge(vertex u, vertex v);
    void add_edge(edge e);
    // return the number of vertices and edges, respectively
    int nr_vertices() const;
    int nr_edges() const;
    // return the list of edges incident to vertex u
    list<edge> adjacent(vertex u) const;
    ...
private:
    int n, m;
    vector<list<edge>> T;
    // alternatives: vector<list<vertex>> T;
    //                  vector<vector<vertex>> T;
    ...
}
```

# Graph Traversals

## Definition

Given a connected graph  $G = \langle V, E \rangle$ , a **traversal** of  $G$  is a sequence of all the vertices in  $V(G)$  where each vertex  $v$  appears exactly once, and it holds that either  $v$  is the first vertex in the sequence or there exists an edge in  $E(G)$  joining  $v$  with a vertex  $w$  appearing before  $v$  in the sequence.

A traversal of a graph is a sequence of traversals of the corresponding connected components. No vertex appears in the sequence if the traversal of the connected components of preceding vertices in the sequence hasn't been completed or the preceding vertex is in the same connected component.

# Graph Traversals

For digraphs, a traversal starting in a vertex  $v$  visits every **accesible** vertex  $w$  from  $v$  exactly once; visiting a vertex  $w \neq v$  implies that there exists some other vertex  $u$  preceding  $w$  in the traversal such that  $(u, w)$  is an arc in  $E$ .

A traversal of a (di)graph visits all its vertices, following the topology of the graph since except the initial vertices, all the others are visited by following an edge or arc in the graph.

# Graph Traversals

Traversals (or a combination of them) allow us to efficiently solve many problems on graphs. For instance:

- Decide whether a graph is connected or not.
- Find the connected components of an undirected graph.
- Find the strongly connected components of a digraph.
- Find whether a graph contains cycles or not.
- Decide if a graph is bipartite or not (equivalently, if a graph is 2-coloreable or not)
- Decide whether a graph is biconnected or not (a graph is biconnected if and only if the removal of any vertex and its incident edges does not disconnect the graph).
- Find the shortest path (with least number of edges/arcs) between any two given vertices.
- Etc.

# Part V

## Graphs

14 Graphs

15 Depth-First Search (DFS)

16 Breadth-First Search (BFS)

17 Shortest Paths: Dijkstra's Algorithm

18 Minimum Spanning Trees: Prim's Algorithm

# Depth-First Search

In a **Depth-First Search** (DFS) (cat: *recorregut en profunditat*, esp: *recorrido en profundidad*) of a graph  $G$  we visit a vertex  $v$  and from there we traverse, recursively, each non-visited vertex  $w$  which is adjacent/a successor of  $v$ .

When we visit a vertex  $u$  we say the vertex is **open**; it remains open until the recursive traversal of all its adjacent/successors has been finished, then  $u$  gets **closed**.

▶ DFS animation

## Depth-First Search

The **direct** or **DFS number** of a vertex is the ordinal number in which the vertex is open by the DFS; if the DFS starts at vertex  $v$ , then the DFS number of  $v$  is 1.

The **inverse number** of a vertex is the ordinal number in which the vertex is closed by the DFS. The first vertex in the DFS for which all neighbors/successors have been visited has inverse number 1.

# Depth-First Search

▷ *visited*, *ndfs*, *ninv*, *num\_dfs*, *num\_inv* are global variables, for simplicity

```
procedure DFS( $G$ )
    for  $v \in V(G)$  do
        visited[ $v$ ] := false
        ndfs[ $v$ ] := 0; ninv[ $v$ ] := 0
    end for
    num_dfs := 0; num_inv := 0
    for  $v \in V(G)$  do
        if  $\neg \text{visited}[v]$  then
            DFS-REC( $G, v, v$ )
        end if
    end for
end procedure
```

# Depth-First Search

```
procedure DFS-REC( $G, v, father$ )
    PRE-VISIT( $v$ )
     $visited[v] := \text{true}$ 
     $num\_dfs := num\_dfs + 1; ndfs[v] := num\_dfs$ 
    for  $w \in G.\text{ADJACENT}(v)$  do
        if  $\neg visited[w]$  then
            PRE-VISIT-EDGE( $v, w$ )
            DFS-REC( $G, w, v$ )
            POST-VISIT-EDGE( $v, w$ )
        else
            ▷ if  $w \neq father$  there is a cycle (that contains edge  $(v, w)$ )
            end if
        end for
        POST-VISIT( $v$ )
         $num\_inv := num\_inv + 1; ninv[v] := num\_inv$ 
    end procedure
```

# Depth-First Search

```
typedef list<edge>::iterator edge_iter;
...

// Some useful macros
#define forall(v,G) for(vertex (v) = 0; (v) < (G).nr_vertices(); ++(v))

#define forall_adj(e, u, G) \
    for(edge_iter (e) = (G).adjacent((u)).begin(); \
        (e) != (G).adjacent((u))].end() ; \
        ++(e))
// this can be written as
// for (edge e : G.adjacent(u)) in C++11

#define target(eit) ((eit) -> second)
#define source(eit) ((eit) -> first)
```

# Depth-First Search

```
void DFS(const Graph& G) {
    vector<bool> visited(G.nr_vertices(), false);
    vector<int> ndfs(G.nr_vertices(), 0);
    vector<int> ninv(G.nr_vertices(), 0);
    int num_dfs = 0;
    int num_inv = 0;

    forall(v, G)
        if (not visited[v])
            DFS(G, v, v, visited, num_dfs, num_inv, ndfs, ninv);
}
```

# Depth-First Search

```
void DFS(const Graph& G, vertex v, vertex padre,
         vector<bool>& visited,
         int& num_dfs, int& num_inv,
         vector<int>& ndfs,
         vector<int>& ninv) {

    PRE-VISIT(v);
    visited[v] = true;
    ++num_dfs; ndfs[v] = num_dfs;
    forall_adj(e, v, G) {
        vertex w = target(e);
        if (not visited[w]) {
            PRE-VISIT-EDGE(v,w);
            DFS(G, w, v, visited, num_dfs, num_inv, ndfs, ninv);
            POST-VISIT-EDGE(v,w);
        } else {
            // we've found a cycle!
        }
    }
    POST-VISIT(v);
    ++num_inv; ninv[v] = num_inv;
}
```

# Depth-First Search

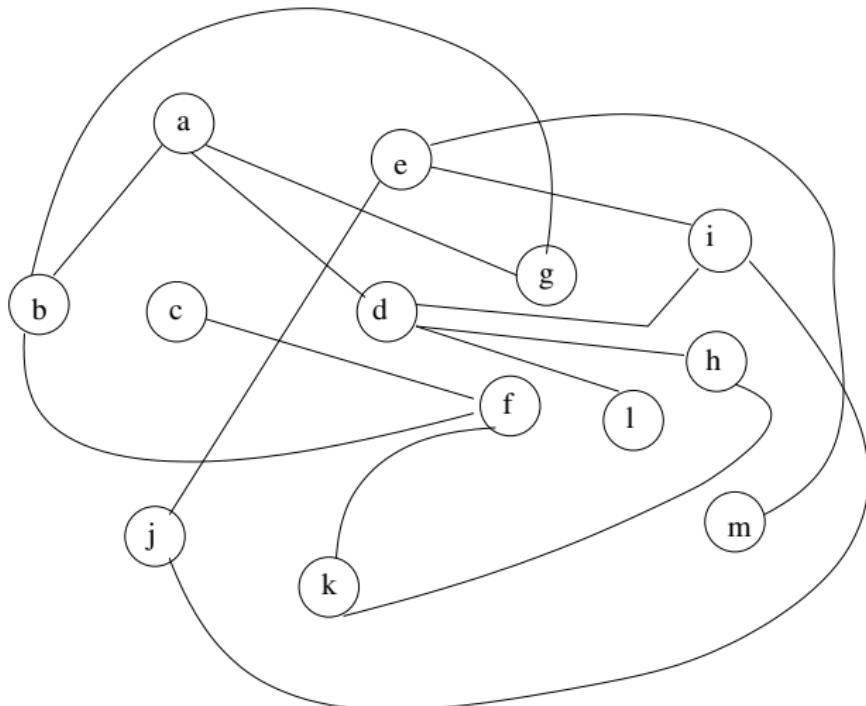
A call to  $\text{DFS}(v, \dots)$  visits the connected component of  $v$  and induces a spanning tree of that component; we call such spanning tree the **DFS tree**  $T_{\text{DFS}}$ .

All edges in the connected component can be thus classified in two types:

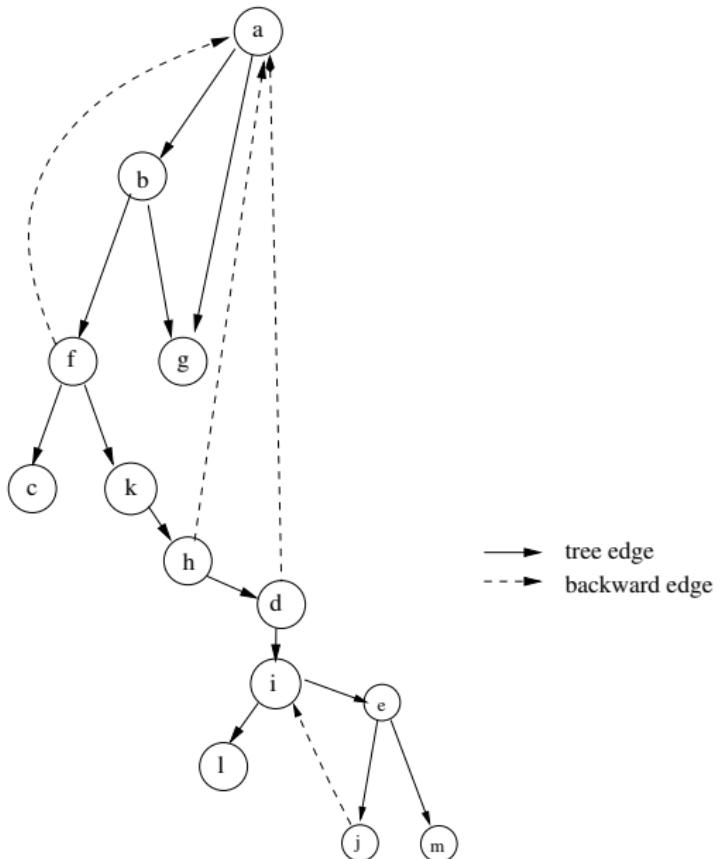
- Tree edges
- Backward edges. Backward edges join the currently visited vertex  $u$  with a previously visited vertex which is not the one from where the recursive traversal of  $u$  was launched. Backward edges close a cycle.

A full DFS of a graph induces a spanning forest.

# Depth-First Search



# Depth-First Search



## Depth-First Search

Consider the following basic problem:

For each vertex  $v$ , we want  $CC[v]$  to be the number of its connected component (the number is unimportant; what matters is that  $CC[u] = CC[v]$  if and only if  $u$  and  $v$  are in the same CC). We also want  $Tree\ Edges[i]$  to be the set of tree edges in CC number  $i$  and  $Back\ Edges[i]$  to be the set of backward edges. The total number of CCs will be  $ncc$ .

# Depth-First Search

```
procedure DFS( $G$ )
    for  $v \in V(G)$  do
         $visited[v] := \text{false}$ 
         $CC[v] := 0$ 
    end for
     $ncc := 0$ 
    for  $v \in V(G)$  do
        if  $\neg visited[v]$  then
             $ncc := ncc + 1$ 
             $TreeEdges[ncc] := \emptyset$ 
             $BackEdges[ncc] := \emptyset$ 
            DFS-REC( $G, v, v$ )
        end if
    end for
end procedure
```

# Depth-First Search

```
procedure DFS-REC( $G, v, father$ )
     $visited[v] := \text{true}$ 
     $CC[v] := ncc$ 
    for  $w \in G.\text{ADJACENT}(v)$  do
        if  $\neg visited[w]$  then
             $TreeEdges[ncc] := TreeEdges[ncc] \cup \{(v, w)\}$ 
            DFS-REC( $G, w, v$ )
        else if  $w \neq father$  then
             $BackEdges[ncc] := BackEdges[ncc] \cup \{(v, w)\}$ 
        end if
    end for
end procedure
```

# Applications of DFS



Detecting cycles or find the CCs (as before) are trivial examples of applications of the DFS.

Another DFS-based simple algorithm allows us to decide if a given graph is **bipartite** or not.

A graph is bipartite if there exists a partition  $\langle A, B \rangle$  of the set of vertices  $V$  (i.e.,  $A \cup B = V$ ;  $A \cap B = \emptyset$ ) such that every edge joins a vertex in  $A$  with a vertex in  $B$ . Equivalently,

- a graph is bipartite if and only if it is 2-coloreable
- a graph is bipartite if and only if it contains no cycle of odd length

Why?

# Applications of DFS: Bipartite Graphs

```
procedure ISBIPARTITE( $G$ )
  for  $v \in V(G)$  do
     $color[v] := -1$ 
  end for
   $c := 0$ 
   $is\_bip := \text{true}$ 
  for  $v \in V(G)$  while  $is\_bip$  do
    if  $color[v] = -1$  then
       $is\_bip := \text{ISBIPARTITE-REC}(G, v, c)$ 
    end if
  end for
  return  $is\_bip$ 
end procedure
```

# Applications of DFS: Bipartite Graphs

```
procedure ISBIPARTITE-REC( $G, v, c$ )
    is_bip := true
    color[ $v$ ] :=  $c$ 
    for  $w \in G.\text{ADJACENT}(v)$  while is_bip do
        if color[ $w$ ] =  $-1$  then
            is_bip := ISBIPARTITE-REC( $G, w, 1 - c$ )
        else
            is_bip := ( $c \neq \text{color}[w]$ )
        end if
    end for
    return is_bip
end procedure
```

## The Cost of DFS

Let us assume that the cost of visiting a vertex (PRE- and POST-VISIT) or visiting an edge (either a tree or a backward edge) is  $\Theta(1)$ .

Then the cost of a DFS is

$$\begin{aligned} \sum_{v \in V} (\Theta(1) + \Theta(\deg(v))) &= \Theta \left( \sum_{v \in V} (1 + \deg(v)) \right) = \\ &\Theta \left( \sum_{v \in V} 1 + \sum_{v \in V} \deg(v) \right) = \Theta(n + m) \end{aligned}$$

## DFS in Digraphs

DFS in digraphs are a bit more complicated to understand than DFS in undirected graphs. When we launch a DFS in a digraph from a vertex  $v$  we do not visit just its strongly connected component (SCC), but all accessible (not visited) vertices from  $v$ ; that includes all vertices in  $v$ 's SCC but some others.

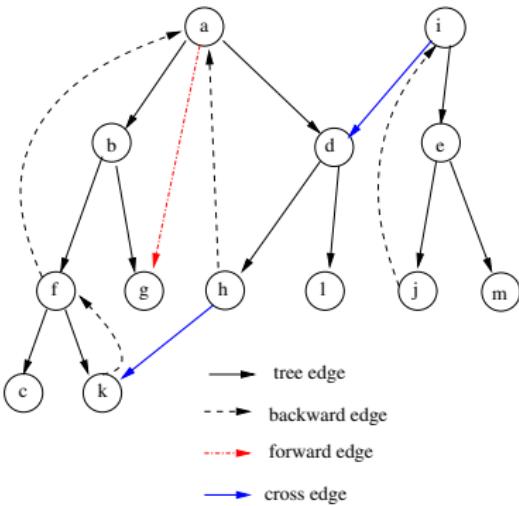
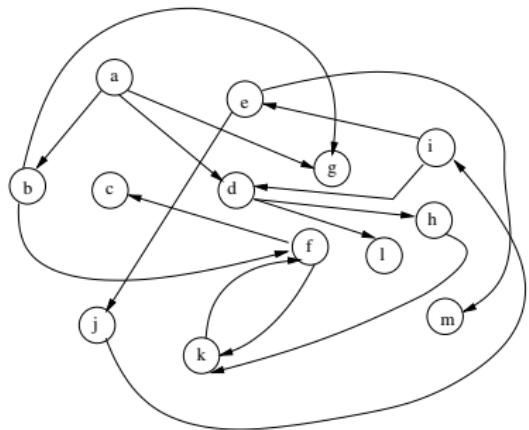
Each call to  $\text{DFS}(v, \dots)$  induces a directed tree, with  $v$  as a root. DFS numbers and inverse numbers are defined as in DFS for undirected graphs.

## DFS in Digraphs

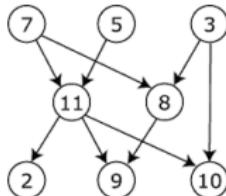
DFS in a digraph also induces a classification of the arcs, but we have now four different types:

- 1 Tree edges: from currently visited vertex  $v$  to a non-visited vertex  $w$
- 2 Backward edges: from currently visited vertex  $v$  to an descendant  $w$  in the DFS tree  $T_{\text{DFS}}$ ;  $\text{ndfs}[w] < \text{ndfs}[v]$  and  $\text{ndfs}[w]$  is open
- 3 **Forward edges**: from currently visited vertex  $v$  to a previously visited descendant  $w$  in  $T_{\text{DFS}}$ ;  $\text{ndfs}[v] < \text{ndfs}[w]$
- 4 **Cross edges**: from currently visited vertex  $v$  to a previously visited vertex  $w$  in the same DFS tree or a different traversal tree;  
 $\text{ndfs}[w] < \text{ndfs}[v]$ , but  $w$  is already closed ( $ninv[w] \neq 0$ )

# DFS in Digraphs



# Directed Acyclic Graphs



A **directed acyclic graph** (DAG) is a digraph that contains no cycles (as its name indicates).

DAGs have many applications since they model well requirements and precedence. For instance, in a large complex software system, each node is a subsystem and each arc  $(A, B)$  indicates that subsystem  $A$  uses subsystem  $B$ .

Vertices with no predecessors are called **roots**; vertices with no successors are called **leaves** of the DAG.

# Topological Sort

A special type of traversal that makes sense in a DAG is a **topological sort**. A topological sort of a DAG  $G$  is a sequence of all its vertices such that for any vertex  $w$ , if  $v$  precedes  $w$  in the sequence then  $(w, v) \notin E$ .  
In other words, no vertex is visited until all its predecessors have been visited.

# Topological Sort

Suppose that we have  $\text{pred}[v]$ , the number of predecessors of  $v$  in  $G$  that have not yet been visited. Then if any vertex  $v$  has  $\text{pred}[v] = 0$  we can visit it and decrement by one  $\text{pred}[w]$ , for all its successors  $w$ . The algorithm has thus two steps:

- 1 Compute the number of predecessors  $\text{pred}[v]$  for all vertices  $v \in G$
- 2 Repeatedly, until all vertices have been visited, visit a vertex  $v$  and update  $\text{pred}[w]$  for all its successors.

# Topological Sort

The two steps are basic variants of the DFS, which take advantage of the fact that there are no cycles in the graph (by definition).

```
procedure TOPOLOGICALSORT( $G$ )
  for  $v \in G$  do  $pred[v] := 0$ 
  end for
  for  $v \in G$  do
    for  $w \in G.\text{SUCCESSORS}(v)$  do  $pred[w] := pred[w] + 1$ 
    end for
  end for
   $Q := \emptyset$ 
  for  $v \in G$  do
    if  $pred[v] = 0$  then
       $Q.\text{PUSH\_BACK}(v)$ 
    end if
  end for
  ...
end procedure
```

## Topological Sort (cont'd)

```
procedure TOPOLOGICALSORT( $G$ )
    ...
    while  $\neg Q.\text{EMPTY}()$  do
         $v := Q.\text{POP\_FRONT}()$ 
        VISIT( $v$ )
        for  $w \in G.\text{SUCCESSORS}(v)$  do
             $pred[w] := pred[w] - 1$ 
            if  $pred[w] = 0$  then
                 $Q.\text{PUSH\_BACK}(w)$ 
            end if
        end for
    end while
end procedure
```

# Part V

## Graphs

14 Graphs

15 Depth-First Search (DFS)

16 Breadth-First Search (BFS)

17 Shortest Paths: Dijkstra's Algorithm

18 Minimum Spanning Trees: Prim's Algorithm

## Breadth-First Search

In a **Breadth-First Search** (BFS) (cat: *recorregut en amplada*, esp: *recorrido en anchura*) of a graph  $G$  starting from some vertex  $s$  we visit all vertices in the connected component of  $s$  in increasing distance from  $s$ .

When a vertex is visited, all its adjacent non-visited vertices are put into a list of vertices yet to be visited.

## Breadth-First Search

In BFS we keep a queue of vertices that have been not yet visited. All the vertices in the queue are at distance  $d$  or  $d + 1$  from the starting vertex  $s$ , where  $d$  is the distance to  $s$  of the last visited vertex. Moreover, all vertices at distance  $d$  in the queue precede the vertices at distance  $d + 1$ , and all vertices of  $G$  at distance  $d$  from  $s$  have already been visited or they are in the queue.

The BFS thus visits the connected component of  $s$  by traversing the levels or layers defined by the distance to  $s$ .

# Breadth-First Search

```
procedure BFS( $G, s, D$ )
```

▷ Assumes  $G$  is connected; all vertices will be visited

▷ After execution,  $D[v] = \text{distance from } s \text{ to } v$

```
for  $v \in G$  do
```

$D[v] := -1$  ▷  $D[v] = -1$  indicates that  $v$  hasn't been visited

```
end for
```

```
 $Q := \emptyset$ ;  $Q.\text{PUSH}(s)$ ;  $D[s] := 0$ 
```

```
while  $\neg Q.\text{EMPTY}()$  do
```

```
     $v := Q.\text{POP}()$ 
```

```
    VISIT( $v$ )
```

```
    for  $w \in G.\text{ADJACENT}(v)$  do
```

```
        if  $D[w] = -1$  then
```

```
             $D[w] := D[v] + 1$ 
```

```
             $Q.\text{PUSH}(w)$ 
```

```
        end if
```

```
    end for
```

```
end while
```

```
end procedure
```

## The Cost of BFS

Assume that the cost of each visit of a vertex is  $\Theta(1)$ , and that the graph is implemented with adjacency lists. Each vertex  $v$  will be inserted into  $Q$  once and removed and visited once; for each vertex we go through its adjacent vertices and push them into the queue, if not visited. Therefore the cost of BFS is clearly

$$\sum_{v \in V} \Theta(1) + \Theta(\deg(v)) = \Theta(|V| + \sum_{v \in V} \deg(v)) = \Theta(|V| + |E|)$$

that is, linear in the size of the graph.

## Applications of BFS: Diameter and center

Given a connected graph  $G$  its **diameter** is the maximal distance between a pair of vertices. The **center** of the graph is the vertex such that the maximal distance to any other vertex is minimal.

A surprising finding in the study of many graphs/networks arising in many areas, specially in social sciences, is that despite the graphs are quite large their diameter is not. That result has been often known as the **six degrees of separation** phenomenon.

▶ Erdős number

▶ The Oracle of Bacon

## Applications of BFS: Diameter and center

To compute the diameter we need to perform a BFS starting from each vertex  $v$  in  $G$ . That will give us cost  $\Theta(|V| \cdot (|V| + |E|)) = \Theta(n(n + m)) = \Theta(n \cdot m)$ , since the graph is connected (then  $m \geq n - 1$ ).

A call  $\text{BFS}(G, s, D)$  allows us to compute  $\text{maxD}[s] = \max\{D[v] \mid v \in G\}$ ; this can actually be done within the internal loop when we update  $D[w] := D[v] + 1$ . The diameter  $D$  is

$$D = \max\{\text{maxD}[s] \mid s \in G\}$$

while the center  $c$  is the vertex such that  $\text{maxD}[c]$  is minimum,

$$c = \arg \min\{\text{maxD}[s] \mid s \in G\}$$

# Part V

## Graphs

- 14 Graphs
- 15 Depth-First Search (DFS)
- 16 Breadth-First Search (BFS)
- 17 Shortest Paths: Dijkstra's Algorithm
- 18 Minimum Spanning Trees: Prim's Algorithm

# Dijkstra's Algorithm

Given a weighted digraph  $G = \langle V, E \rangle$ , Dijkstra's algorithm (1959) finds all shortest paths from a given vertex (the **source**) to all other vertices in the digraph.

If weights can be negative then the digraph can contain negative weight cycles, and then shortest paths are not well defined; hence, Dijkstra's algorithm can fail if there are arcs in the digraph with negative weight. We will assume from now on that all weights  $\omega : E \rightarrow \mathbb{R}^+$  are positive.

Let  $\mathcal{P}(u, v)$  denote the set of all (simple) paths between two vertices  $u$  and  $v$  of  $G$ . Given a path  $\pi = [u, \dots, v] \in \mathcal{P}(u, v)$  its weight is the sum of the weights of the arcs in  $\pi$ :

$$\omega(\pi) = \omega(u, v_1) + \omega(v_1, v_2) + \cdots + \omega(v_{n-1}, v).$$

## Dijkstra's Algorithm

Let  $\Delta(u, v) = \min\{\omega(\pi) \mid \pi \in \mathcal{P}(u, v)\}$  and  $\pi^*(u, v)$  a path in  $\mathcal{P}(u, v)$  with minimal weight. If  $\mathcal{P}(u, v) = \emptyset$  then we define  $\Delta(u, v) = +\infty$ , by convention.

Our first version of Dijkstra's algorithm computes  $\Delta(u, v) = \omega(\pi^*(u, v))$  for all  $v \in V(G)$ , for some given vertex  $u$ . Later we will see how to find the shortest paths  $\pi^*(u, v)$ , not just their weights.

# Dijkstra's Algorithm

- ▷  $G = \langle V, E \rangle$  is a weighted digraph, all weights are positive
- ▷  $s \in V$

$\text{DIJKSTRA}(G, s, D)$

- ▷ For all  $v \in V$ ,  $D[v] = \Delta(s, v)$

- ▷  $G = \langle V, E \rangle$  is a weighted digraph, all weights are positive
- ▷  $s \in V$

$\text{DIJKSTRA}(G, s, D, path)$

- ▷ For all  $v \in V$ ,  $D[v] = \Delta(s, v)$
- ▷ For all  $v \in V$ ,  $D[v] < +\infty \implies path[v] = \pi^*(s, v)$

## Dijkstra's Algorithm

Dijkstra's algorithm works iteratively. On each iteration the set of vertices is partitioned in two parts: the vertices that we have **visited** and those that haven't been visited yet, also called **candidates**. The invariant of the loop establishes that

- 1 For all visited vertices  $u$ ,  $D[u] = \Delta(s, u)$
- 2 For all candidate vertices  $u$ ,  $D[u]$  is the weight of the shortest path from  $s$  to  $u$  which passes exclusively through visited vertices, i.e., only the last vertex in the path,  $u$ , is a candidate.

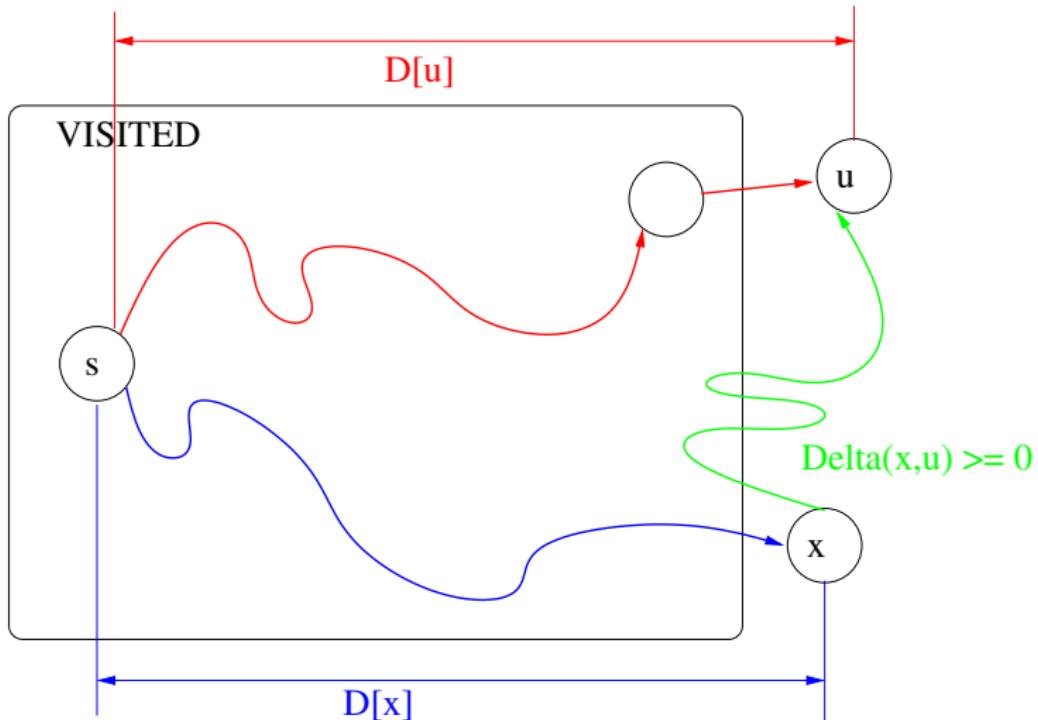
Each iteration of the main loop selects a candidate vertex, updates  $D$  and marks  $u$  as **visited**. When all vertices have been visited, we have the desired answer.

## Dijkstra's Algorithm

Which candidate vertex do we need to select on each iteration of Dijkstra's algorithm? Intuitively, the candidate to choose is the one with minimal  $D$ . Let  $u$  be that vertex. According to the invariant,  $D[u]$  is minimal among all paths passing only through visited vertices. But it must also be the minimal weight for all paths.

If there were a shortest path going through a candidate vertex  $x$ , the weight of that path would be  $D[x] + \Delta(x, u) < D[u]$ . Since  $\Delta(x, u) \geq 0$  we would have  $D[x] < D[u]$ , and that's a contradiction since by assumption  $D[u]$  is minimal among all candidate vertices.

# Dijkstra's Algorithm



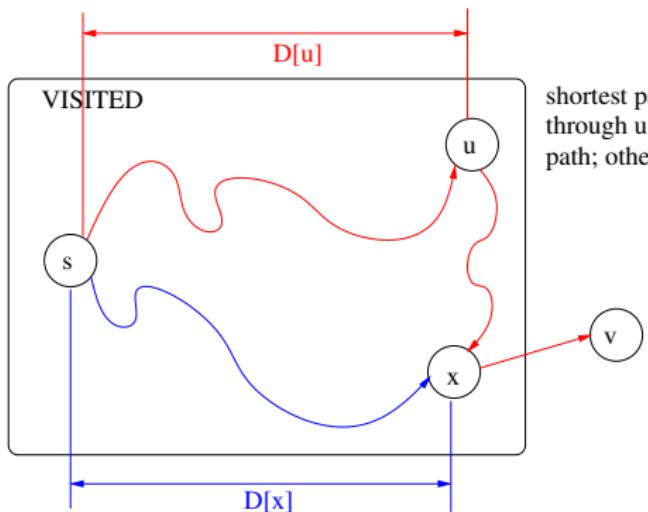
## Dijkstra's Algorithm

As we have just seen  $D[v] = \Delta(s, v)$  for all visited vertices  $v$ , and for the vertex  $u$  to be visited. We need only to consider how to maintain the second part of the invariant.

Consider a candidate vertex  $v$ . Since  $u$  becomes a visited vertex in the current iteration, we may have a new shortest path from  $s$  to  $v$  going only through visited vertices that now include  $u$ .

A simple reasoning shows that if such shortest path includes  $u$  then  $u$  must be the immediate predecessor of  $v$  in the path (assuming the contrary leads to a contradiction).

# Dijkstra's Algorithm



shortest path from  $s$  to  $v$  passes  
through  $u \Rightarrow u$  is the last visited vertex in the  
path; otherwise  $D[x] > D[u] \Rightarrow$  contradiction!

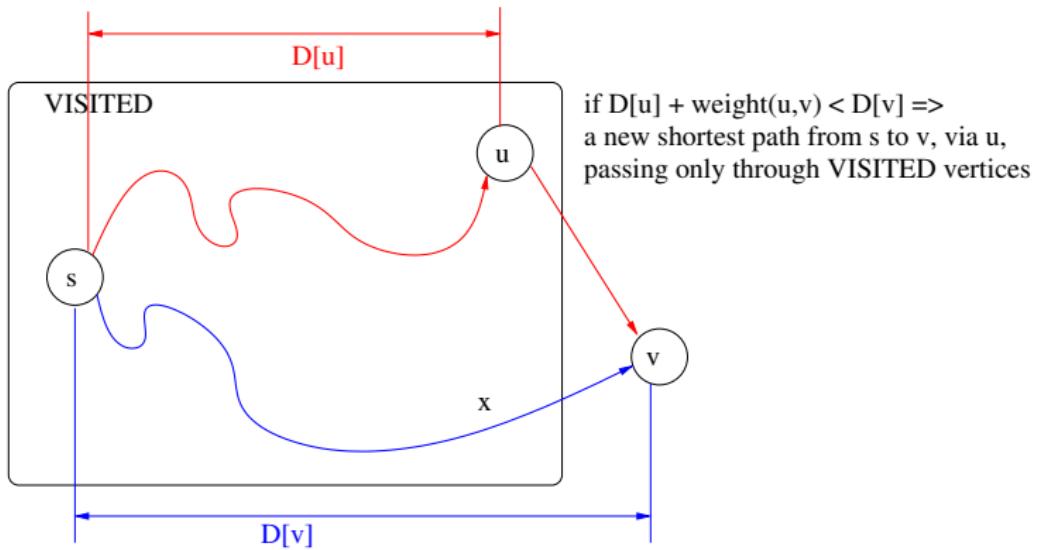
## Dijkstra's Algorithm

It follows that  $D[v]$  can change if and only if  $v$  is a successor of  $u$ ; in particular, if

$$D[v] > D[u] + \omega(u, v).$$

then  $D[v]$  must be updated  $D[v] := D[u] + \omega(u, v)$ . The condition above is never true if  $v$  has already been visited.

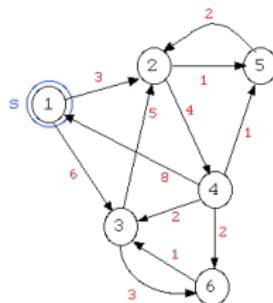
# Dijkstra's Algorithm



# Dijkstra's Algorithm

```
procedure DIJKSTRA( $G, s, D$ )
     $\triangleright$  At the end  $D[u] = \Delta(s, u)$  for all  $u \in V(G)$ 
     $\triangleright$   $cand$  is the subset of candidates in  $V(G)$ 
    for  $v \in V(G)$  do
         $D[v] := +\infty$ 
    end for
     $D[s] := 0$ 
     $cand := V(G)$ 
    while  $cand \neq \emptyset$  do
         $u :=$  the vertex in  $cand$  with minimum  $D$ 
         $cand := cand - \{u\}$ 
        for  $v \in G.\text{SUCCESSORS}(u)$  do
             $d := D[u] + G.\text{WEIGHT}(u, v)$ 
            if  $d < D[v]$  then
                 $D[v] := d$ 
            end if
        end for
    end while
end procedure
```

# Dijkstra's Algorithm



D						CANDIDATES
1	2	3	4	5	6	
0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\{1, 2, 3, 4, 5, 6\}$
0	3	6	$\infty$	$\infty$	$\infty$	$\{2, 3, 4, 5, 6\}$
0	3	6	7	4	$\infty$	$\{3, 4, 5, 6\}$
0	3	6	7	4	$\infty$	$\{3, 4, 6\}$
0	3	6	7	4	8	$\{4, 6\}$
0	3	6	7	4	8	$\{6\}$
						$\emptyset$

## Dijkstra's Algorithm

To compute the shortest paths (not just their weights) we need the following key observation: if the shortest path from  $s$  to  $u$  goes through  $x$  then the subsequence going from  $s$  to  $x$  must be a shortest path from  $s$  to  $x$ . We shall hence compute a (implicit) shortest paths tree: for all vertices

$$path[v] = \begin{cases} s & \text{if } v = s, \\ u & \text{if } (u, v) \text{ is the last arc in } \pi^*(s, v), \\ \perp & \text{if } D[v] = \Delta(s, v) = +\infty, \end{cases}$$

( $path[v] = \perp$  indicates that no path exists from  $s$  to  $v$ ).

# Dijkstra's Algorithm

The only changes to our code:

- 1 Initialize  $\text{path}[s] := s$ ; Initialize  $\text{path}[v] = \perp$  for all other vertices  $v$
- 2 Update  $\text{path}$  every time  $D$  is updated

```
...
for  $v \in G.\text{SUCESSORS}(u)$  do
     $d := D[u] + G.\text{WEIGHT}(u, v)$ 
    if  $d < D[v]$  then
         $D[v] := d$ 
         $\text{path}[v] := u$ 
    end if
end for
...
```

To recover the full path from  $s$  to  $u$ , if  $\text{path}[u] \neq \perp$  then it is enough to roll back:

$$\pi^*(s, u) = [\text{path}[u], \text{path}[\text{path}[u]], \dots, \text{path}[\dots [\text{path}[u]] \dots], s]^{\text{reverse}}.$$

## The Cost of Dijkstra's Algorithm

Let  $n = |V(G)|$  and  $m = |E(G)|$ , as usual. Assume that the digraph is implemented with an adjacency matrix. Then the cost of the algorithm is  $\Omega(n^2)$  since there are  $n$  iterations, and we incur a cost  $\Omega(n)$  to go through the successors  $v$  of the chosen candidate (because we are scanning the corresponding row in the matrix). Finding the minimum  $D$  at each iteration is  $\mathcal{O}(n)$  and therefore the cost of the algorithm is actually  $\Theta(n^2)$ .

Let's now assume that the graph is implemented with adjacency lists. Then cost of the internal loop that goes through the successors of  $u$  is

$$\Theta(\deg(u) \cdot \text{cost of updating } D).$$

## The Cost of Dijkstra's Algorithm

If the set *cand* and the table *D* are implemented as simple arrays (*cand*[*i*] = true if and only if *i* is a candidate) then the cost of the algorithm is

$$\Theta(n^2 + m) = \Theta(n^2)$$

since we have cost  $\Theta(i + \deg(u))$  during the iteration in which we have still *i* candidates and *u* is selected to be visited.

## Improving the Cost of Dijkstra's Algorithm

For arbitrary sets of vertices  $V(G)$  (not just  $V = \{0, \dots, n - 1\}$ ), we can achieve the same cost implementing *cand* (and *D*) with hash tables. However the problem is that selecting a candidate to visit is still inefficient.  
To improve the efficiency we need to convert *cand/D* to a priority queue; the elements are the candidate vertices and the priority of candidate  $v$  is its  $D[v]$ .

# Improving the Cost of Dijkstra's Algorithm

- ▷  $cand$ : a min-priority queue
- ▷ elements are vertices in  $G$ ,
- ▷ priority of  $u$  is  $D[u]$
- ...
- ▷  $D[v] = +\infty$  for all  $v \neq s$ ;  $D[s] = 0$

**for**  $v \in V(G)$  **do**  
     $cand.\text{INSERT}(v, D[v])$

**end for**

**while**  $\neg cand.\text{EMPTY}()$  **do**  
     $u := cand.\text{MIN}(); cand.\text{REMOVE\_MIN}()$   
    **for**  $v \in G.\text{SUCESSORS}(u)$  **do**  
        ...

**end for**

**end while**

# Improving the Cost of Dijkstra's Algorithm

But since we can update priorities, our data structure must support another method to decrease the priority of a given element.

```
...
for  $v \in G.\text{SUCESSORS}(u)$  do
     $d := D[u] + G.\text{WEIGHT}(u, v)$ 
    if  $d < D[v]$  then
         $D[v] := d$ 
         $cand.\text{DECREASE\_PRIO}(v, d)$ 
    end if
end for
...
```

# Improving the Cost of Dijkstra's Algorithm

If all operations on the priority queue have cost  $\mathcal{O}(\log n)$  then the cost of Dijkstra's algorithm is

$$\begin{aligned}D(n) &= \sum_{v \in V(g)} \mathcal{O}(\log n) \cdot (1 + \text{out-deg}(v)) \\&= \Theta(n \log n) + \mathcal{O}(\log n) \sum_{v \in V(g)} \text{out-deg}(v) \\&= \Theta(n \log n) + \mathcal{O}(m \log n) = \mathcal{O}((n + m) \log n)\end{aligned}$$

The cost of the initializations is  $\mathcal{O}(n \log n)$ .

Then the total cost is

$$\mathcal{O}((n + m) \log n)$$

The worst-case is actually  $\Theta((m + n) \log n)$ , but the cost in many situations is often significantly smaller since we need not to decrease the priority of all successors of the selected candidate, or the cost of decreasing the priority is not  $\Theta(\log n)$ .

# Improving the Cost of Dijkstra's Algorithm

What we need is a data structure that combines the features of priority queue and of dictionary (preferably a hash, if  $V \neq \{0, \dots, n - 1\}$ ), since to find the priority of an element or to decrease its priority we must (quickly) locate the element within the heap.

```
template <class Elem, class Prio>
class DijkstraPrioQueue<ELEM, Prio> {
public:
    DijkstraPrioQueue(int n = 0);
    void insert(const Elem& e, const Prio& prio);
    Elem min() const;
    void remove_min();
    Prio prio(const Elem& e) const; // returns the priority of element v
    bool contains(const Elem& e) const;
    void decrease_prio(const Elem& e, const Prio& newprio);
    bool empty() const;
    ...
};
```

# Improving the Cost of Dijkstra's Algorithm

```
typedef vector<list<pair<int, double>>> graph;

...
void Dijkstra(const graph& G, int s, ... ) {
    DijkstraPrioQueue<int, double> cand(G.size());
    for (int v = 0; v < G.size(); ++v)
        cand.insert(v, INFINITY);
    cand.decrease_prio(s, 0);

    while(not cand.empty()) {
        int u = cand.min(); double du = cand.prio(u);
        cand.remove_min();
        for (auto e : G[u]) {
            // e is a pair <v, weight(u,v)>
            int v = e.first;
            double d = du + e.second;
            if (d < cand.prio(v))
                cand.decrease_prio(v, d);
        }
    }
}
```

# Part V

## Graphs

- 14 Graphs
- 15 Depth-First Search (DFS)
- 16 Breadth-First Search (BFS)
- 17 Shortest Paths: Dijkstra's Algorithm
- 18 Minimum Spanning Trees: Prim's Algorithm

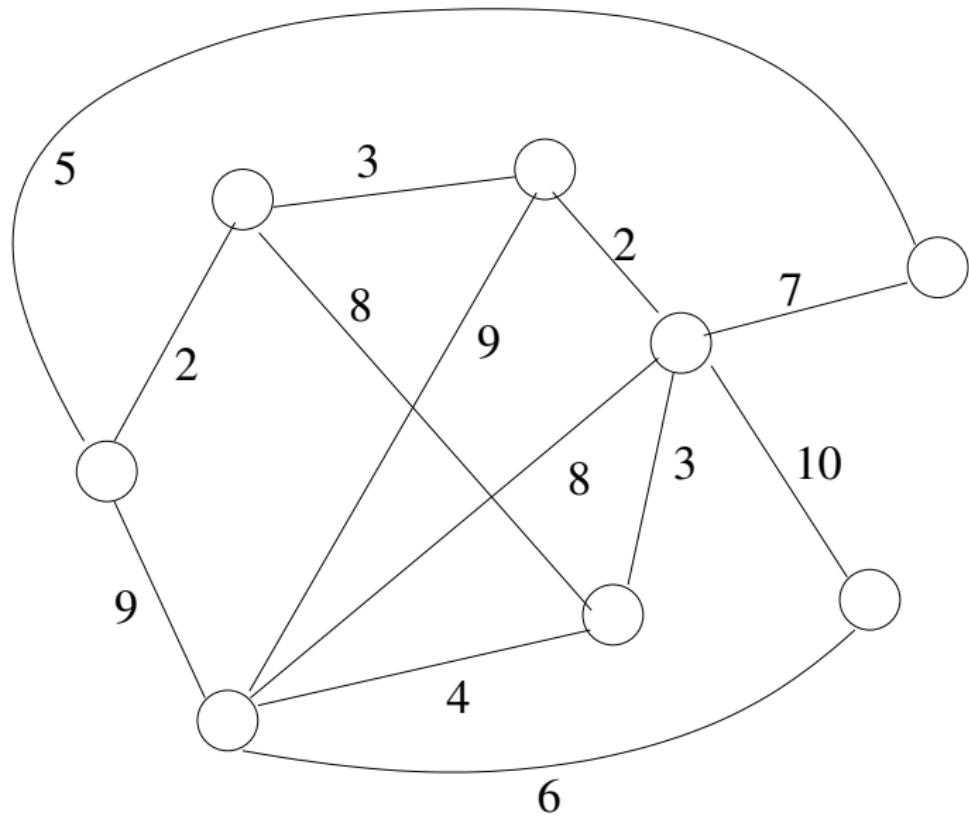
# Minimum Spanning Trees (MSTs)

Given a weighted undirected connected graph  $G = \langle V, E \rangle$ , with weights  $\omega : E \rightarrow \mathbb{R}$ , a **minimum spanning tree** (MST) of  $G$  is a spanning subgraph  $T = \langle V, A \rangle$  of  $G$  ( $V(T) = V(G)$ ), that is a tree (connected and acyclic) and its total weight

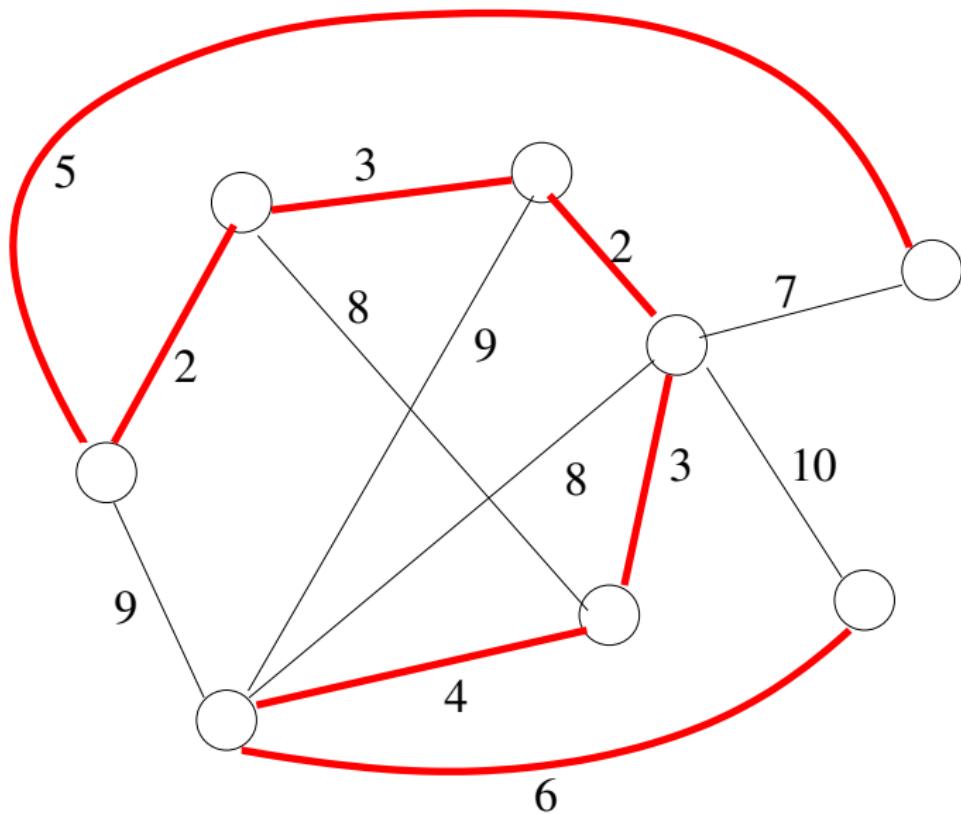
$$\omega(T) = \sum_{e \in A} \omega(e)$$

is minimum among all possible spanning trees of  $G$ .

# Minimum Spanning Trees (MSTs)



# Minimum Spanning Trees (MSTs)



# Minimum Spanning Trees (MSTs)

There are many different algorithms to compute MSTs. All them follow the **greedy** scheme.

```
A := ∅; Candidates := E
while |A| ≠ |V(G)| – 1 do
    Select an edge  $e \in Candidates$  that
        does not close a cycle in  $T = \langle V, A \rangle$ 
    A := A ∪ {e}
    Candidates := Candidates – {e}
end while
```

# Minimum Spanning Trees (MSTs)

## Definition

A subset of edges  $A \subset E(G)$  is **promising** if and only if

- 1  $A$  contains no cycle
- 2  $A$  is a subset of the edges of some MST of  $G$

# Minimum Spanning Trees (MSTs)

## Definition

A **cut** in a graph  $G$  is a partition of the set of vertices  $V$  in two subsets  $C$  and  $C'$ , with

$$C \cup C' = V(G) \quad \text{and} \quad C \cap C' = \emptyset$$

## Definition

An edge  $e$  **respects** a cut  $\langle C, C' \rangle$  if both ends of  $e$  belong to the same part ( $C$  or  $C'$ ); otherwise, we say that  $e$  **crosses** the cut.

# Minimum Spanning Trees (MSTs)

## Theorem

*Let  $A$  be a promising set of edges that respects some cut  $\langle C, C' \rangle$  of  $G$ . Let  $e$  be an edge of minimum weight among those crossing the cut  $\langle C, C' \rangle$ . Then*

$$A \cup \{e\}$$

*is promising.*

## Minimum Spanning Trees (MSTs)

Previous theorem gives us a “recipe” to design MST algorithms: start with an empty set of edges  $A$ ; for each iteration, define what is the cut of  $G$ , then select the edge with minimum weight among those crossing the cut and add the edge to  $A$ .

Since  $A$  respects the cut and  $e$  crosses it, the addition of  $e$  to  $A$  cannot create a cycle.

Any algorithm following the scheme above is automatically guaranteed correct.

## Minimum Spanning Trees (MSTs)

Proof of Theorem 39:

Let  $A'$  be the set of edges of some MST  $T'$  such that  $A \subset A'$  (this is well defined, as  $A$  is assumed to be *promising*). As  $A$  respects the cut there must exist at least one edge crossing the cut belonging to  $A'$ , otherwise  $T'$  would not be connected. Let  $e'$  be one such edge. If  $e'$  is of minimum weight then  $A \cup \{e'\}$  is promising and the theorem is proved in this case. Assume now that  $e'$  is not of minimum weight.

# Minimum Spanning Trees (MSTs)

Proof of Theorem 39 (cont'd):

The total weight (or **cost**) of  $T'$  includes:

- ① the sum of the weights of the edges in  $A$
- ② the weight  $\omega(e')$
- ③ the sum of the weight of some other edges

In the theorem we add the edge  $e$  of minimum weight crossing the cut to  $A$  (and we are assuming now that  $e \notin A'$ ). Then it will create a cycle, since  $T'$  is a tree, and there will be some other  $e' \in A'$  crossing the cut, hence part of the cycle too.

# Minimum Spanning Trees (MSTs)

Proof of Theorem 39 (cont'd):

If we remove  $e'$

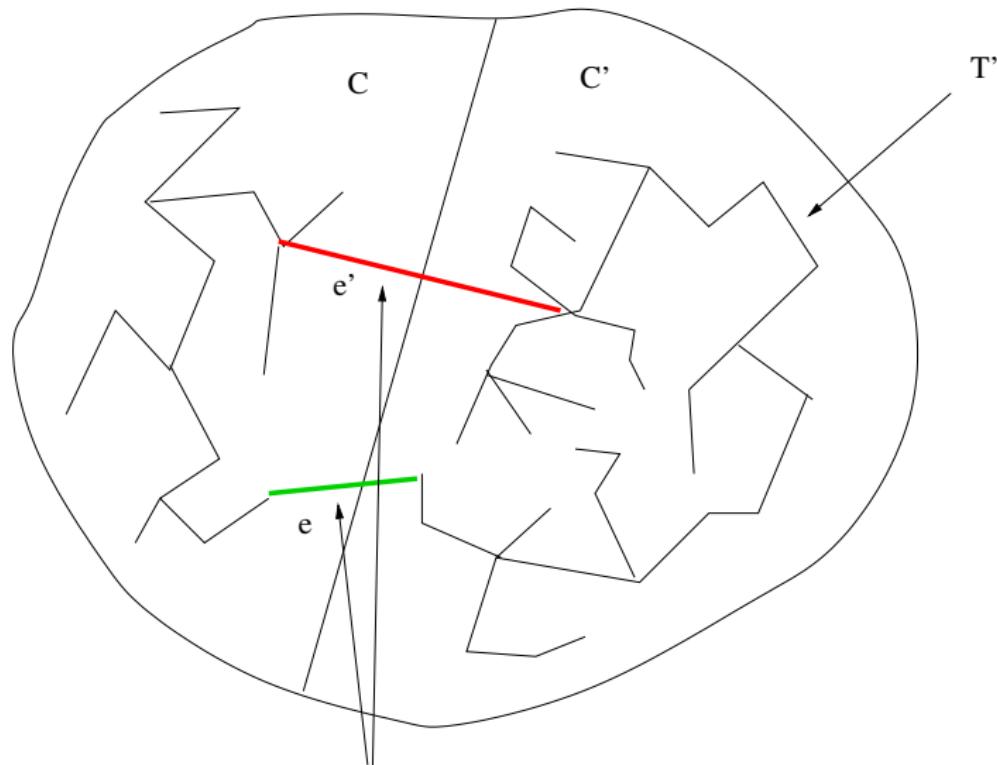
$$T = T' \cup \{e\} - \{e'\}$$

we get a new spanning tree  $T$  and its total weight is

$$\omega(T) = \omega(T') + \omega(e) - \omega(e') \leq \omega(T')$$

which is a contradiction unless  $\omega(e) = \omega(e')$  and  $\omega(T) = \omega(T')$ . That completes the proof, since  $A \cup \{e\}$  is a subset of  $T$  which a MST of  $G$ .

# Minimum Spanning Trees (MSTs)



podemos sustituir  $e'$  por  $e$  para obtener un nuevo MST

## Part VI

# Exhaustive Search and Generation

19

General Concepts

## Part VII

# Notions of Intractability

- 20 The Class P and the Class NP
- 21 Reductions and NP-Completeness

## Part VII

# Notions of Intractability

20

The Class P and the Class NP

21

Reductions and NP-Completeness