

# Tema 3. Diccionaris

Estructures de Dades i Algorismes

FIB

Transparències d' [Antoni Lozano](#)  
(amb edicions menors d'altres professors)

Q1 2021 – 22

# Tema 3. Diccionaris

## 1 Introducció

- Tipus abstractes de dades
- Diccionaris

## 2 Taules de dispersió

- Introducció
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)

# Tema 3. Diccionaris

## 1 Introducció

- Tipus abstractes de dades
- Diccionaris

## 2 Taules de dispersió

- Introducció
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)

# Tipus abstractes de dades

Les estructures de dades es formalitzen amb el concepte de

## Tipus Abstracte de Dades (TAD)

TAD = VALORS + OPERACIONS

### Exemple

El tipus **enter** consta de:

- **VALORS**: conjunt d'enters en un interval [min,max]
- **OPERACIONS**: +, -, \*, /, %

# Tipus abstractes de dades

Les estructures de dades es formalitzen amb el concepte de

Tipus Abstracte de Dades (TAD)

TAD = VALORS + OPERACIONS

## Exemple

El tipus **enter** consta de:

- **VALORS**: conjunt d'enters en un interval [min,max]
- **OPERACIONS**: +, -, \*, /, %

# Tipus abstractes de dades

Per saber com s'ha de comportar un TAD,  
cal especificar també les **propietats** de les operacions:

- **valors**
- **operacions**
- **propietats** de les operacions

## Exemple

El tipus **enter** s'especifica amb:

- **valors**: conjunt d'enters en un interval [min,max]
- **operacions**: +, -, \*, /, %
- **propietats**:  $a + 0 = a$ ,  $a * 0 = 0$ ,  $a * 1 = a$ ,  $a * b = b * a$ , etc.

# Tipus abstractes de dades

L'especificació d'un TAD permet descriure el funcionament d'una estructura de dades independentment de la seva implementació.

**TAD:** saber el **què** sense saber el **com**

# Tipus abstractes de dades

## Exemple: TAD *booleans*

- **VALORS:** Conjunt dels booleans  $\text{Bool} = \{\text{cert}, \text{fals}\}$
- **OPERACIONS:** cert, fals, i, o, no

cert:  $\rightarrow \text{Bool}$

fals:  $\rightarrow \text{Bool}$

i:  $\text{Bool} \times \text{Bool} \rightarrow \text{Bool}$

o:  $\text{Bool} \times \text{Bool} \rightarrow \text{Bool}$

no:  $\text{Bool} \rightarrow \text{Bool}$

- **PROPIETATS:**

$$i(\text{cert}, x) = x$$

$$i(\text{fals}, x) = \text{fals}$$

$$o(\text{cert}, x) = \text{cert}$$

$$o(\text{fals}, x) = x$$

$$\text{no}(\text{cert}) = \text{fals}$$

$$\text{no}(\text{fals}) = \text{cert}$$

## Exemple: TAD *naturals*

- **VALORS:** Conjunt  $\mathbb{N}$  dels naturals
- **OPERACIONS:** zero, suc, suma, iguals?

zero:  $\rightarrow \mathbb{N}$

suma:  $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

suc:  $\mathbb{N} \rightarrow \mathbb{N}$

iguals?:  $\mathbb{N} \times \mathbb{N} \rightarrow \text{Bool}$

- **PROPIETATS:**

$$\text{suma}(\text{zero}, x) = x$$

$$\text{suma}(\text{suc}(x), y) = \text{suc}(\text{suma}(x, y))$$

$$\text{iguals?}(\text{zero}, \text{zero}) = \text{cert}$$

$$\text{iguals?}(\text{suc}(x), \text{zero}) = \text{fals}$$

$$\text{iguals?}(\text{zero}, \text{suc}(x)) = \text{fals}$$

$$\text{iguals?}(\text{suc}(x), \text{suc}(x)) = \text{cert}$$

Grosso modo, anomenem **diccionari** a un conjunt amb les operacions:

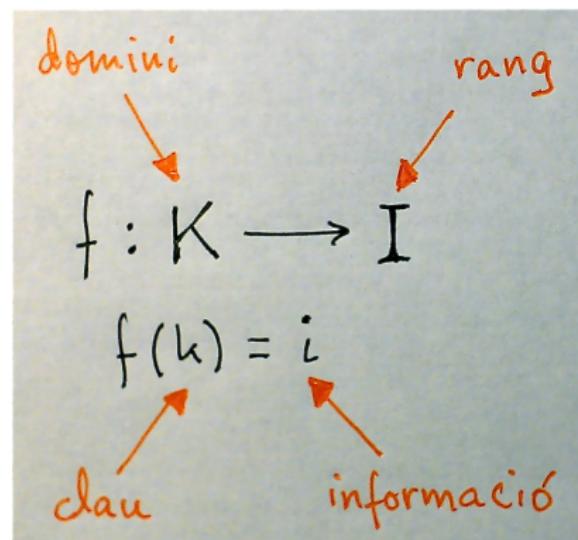
- **assignar**: incloure un element nou
- **esborrar**: eliminar un element
- **consultar**: comprovar si un element amb una certa clau hi és

Els elements disposaran d'un camp anomenat **clau** que els identifica:

element = (clau, informació)

Les claus **no es poden repetir**.

Penseu en els diccionaris com en **funcions** o en generalitzacions dels **vectors**:



## Exemple 1

Si tenim un conjunt de **comptes corrents**,

- la clau pot ser el número de compte:
- la informació és el propietari, el tipus de compte, el saldo, els moviments...

## Exemple 2

Un **club d'esports** vol organitzar la informació referida als socis.

En aquest cas, tria:

- el número de soci com a clau
- el nom, l'adreça, el telèfon i el correu electrònic com a informació

## Exemple 1

Si tenim un conjunt de **comptes corrents**,

- la clau pot ser el número de compte:
- la informació és el propietari, el tipus de compte, el saldo, els moviments...

## Exemple 2

Un **club d'esports** vol organitzar la informació referida als socis.

En aquest cas, tria:

- el número de soci com a clau
- el nom, l'adreça, el telèfon i el correu electrònic com a informació

Considerarem les operacions següents:

- **assignar**: afegir un element (clau, informació) al diccionari.  
Si existia un element amb la mateixa clau, se sobreescrueix la informació.
- **esborrar**: donada una clau, s'esborra l'element que té aquella clau.  
Si no hi ha cap element amb la clau, no es fa res.
- **present**: donada una clau, es retorna un booleà que indica si el diccionari conté un element amb aquella clau.
- **cerca**: donada una clau, retorna una referència a l'element amb aquella clau.
- **consulta**: donada una clau, retorna una referència a la informació associada a la clau.
- **talla**: retorna la talla del diccionari.

## Exemple: TAD *diccionari*

- **VALORS:**  $\mathcal{C} = \mathcal{P}(K \times I)$ , on  $K$  és un conjunt de claus i  $I$  d'informació

- **OPERACIONS:**

crear:  $\rightarrow \mathcal{C}$

assignar:  $K \times I \times \mathcal{C} \rightarrow \mathcal{C}$

esborrar:  $K \times \mathcal{C} \rightarrow \mathcal{C}$

consultar:  $K \times \mathcal{C} \rightarrow \text{Bool}$

- **PROPIETATS:** suposem  $k, k_1, k_2 \in K, i, j \in I, k_1 \neq k_2$

$$\text{esborrar}(k, \text{crear}) = \text{crear}$$

$$\text{esborrar}(k, \text{assignar}(k, i, D)) = \text{esborrar}(k, D)$$

$$\text{esborrar}(k_1, \text{assignar}(k_2, i, D)) = \text{assignar}(k_2, i, \text{esborrar}(k_1, D))$$

$$\text{assignar}(k, i, \text{assignar}(k, j, D)) = \text{assignar}(k, i, D)$$

$$\text{assignar}(k_1, i, \text{assignar}(k_2, j, D)) = \text{assignar}(k_2, j, \text{assignar}(k_1, i, D))$$

$$\text{consultar}(k, \text{crear}) = \perp$$

$$\text{consultar}(k, \text{esborrar}(k, D)) = \perp$$

$$\text{consultar}(k, \text{assignar}(k, i, D)) = i$$

$$\text{consultar}(k_1, \text{assignar}(k_2, i, D)) = \text{consultar}(k_1, D)$$

## Nombre d'elements consultats/copiats en les operacions de diccionaris

cas pitjor/mitjà	assignar	esborrar	consultar
vector no ordenat	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$
llista no ordenada	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$
vector ordenat	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$	$\Theta(\log n)$ , $\Theta(\log n)$
llista ordenada	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$	$\Theta(n)$ , $\Theta(n)$
taula de dispersió	$\Theta(n)$ , $\Theta(1)$	$\Theta(n)$ , $\Theta(1)$	$\Theta(n)$ , $\Theta(1)$

En vectors, cal desplaçar els elements.

Cal comprovar repetits.

A continuació, veurem diferents estructures de dades que implementen diccionaris.

# Tema 3. Diccionaris

## 1 Introducció

- Tipus abstractes de dades
- Diccionaris

## 2 Taules de dispersió

- Introducció
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)

Les **taules de dispersió** (*hash tables*) són estructures de dades eficients per implementar els diccionaris.

- Són generalitzacions dels vectors.
- El temps en cas pitjor pot ser de  $\Theta(n)$ , però amb hipòtesis raonables el temps esperat serà  $\Theta(1)$  en totes les operacions.

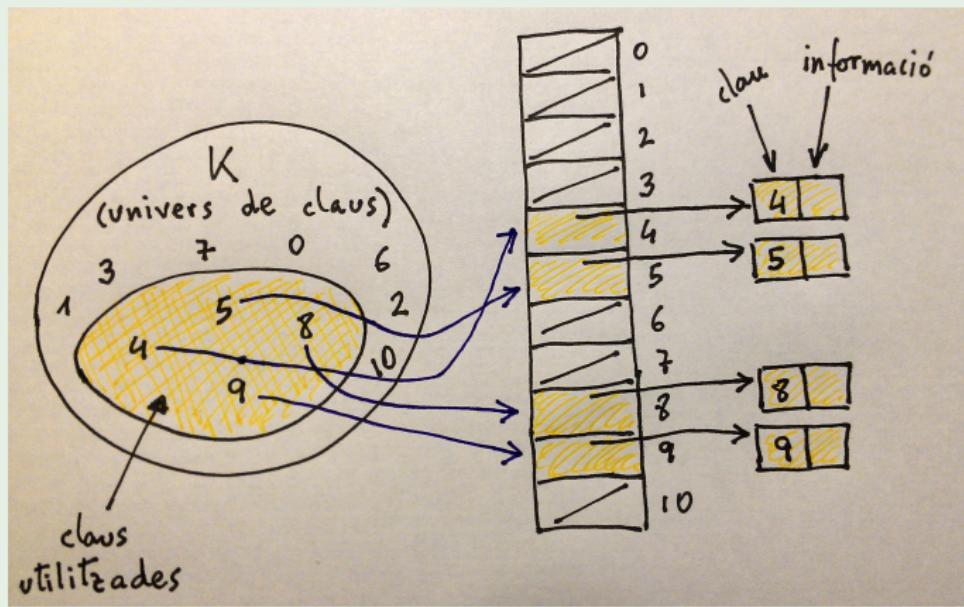
S'han fet servir almenys des dels anys 1950s:

W. W. Peterson. Addressing for random access storage. *IBM Journal of Research and Development*, 1(2), Abril 1957.

# Taules d'accés directe

Suposem que un club d'esports no tindrà mai més de 300 socis. Per implementar un diccionari amb el número de soci com a clau, n'hi ha prou a definir un vector  $V[0..299]$ .

Exemple d'adreçament directe (simplificat)



En l'adreçament directe:

- Cada posició del vector correspon a una clau.
- Es guanya eficiència en temps a costa d'usar més espai del necessari.
- Les operacions seran  $\Theta(1)$  en cas pitjor.
- Es pot guardar la informació directament en les posicions del vector corresponents a la seva clau, però cal indicar si l'espai és buit.

Quan

- les claus no són nombres naturals, o
- el nombre de claus usades és molt petit comparat amb el nombre possible de claus, o
- el nombre possible de claus és enorme,

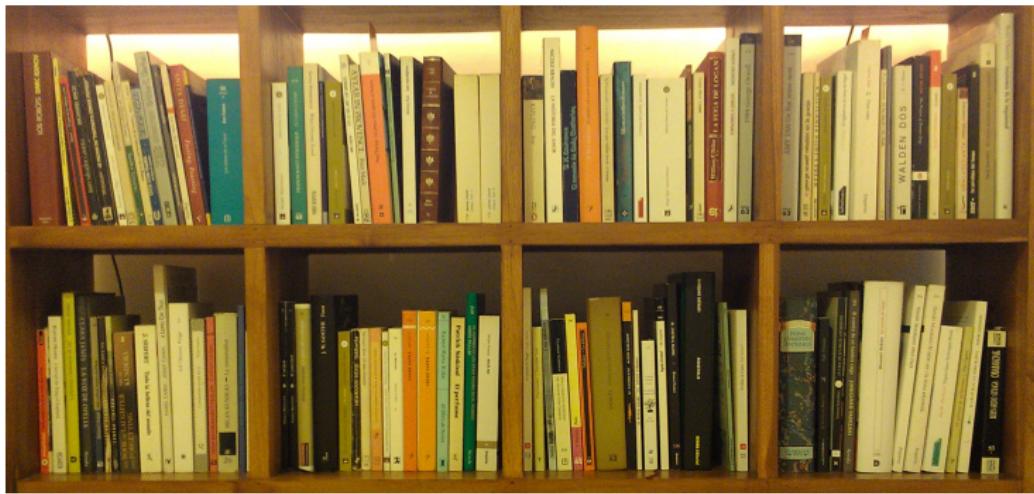
cal abandonar les taules d'accés directe. Llavors,

En lloc de fer servir la clau com a índex per accedir a la taula,  
**l'índex es calcula a partir de la clau.**

# Taules de dispersió

Si es vol consultar un llibre a la biblioteca central de la Universitat de Karlsruhe, cal demanar el llibre amb antelació. Llavors, el personal el busca i el classifica en una habitació amb 100 prestatges d'acord amb la regla següent.

El llibre estarà col·locat en un prestatge numerat amb els dos últims díigits del carnet de la biblioteca de qui l'ha demanat.



# Taules de dispersió

La biblioteca expedeix carnets des del 1825 amb números consecutius.

## Pregunta

Per què els dos últims díigits i no els dos primers?

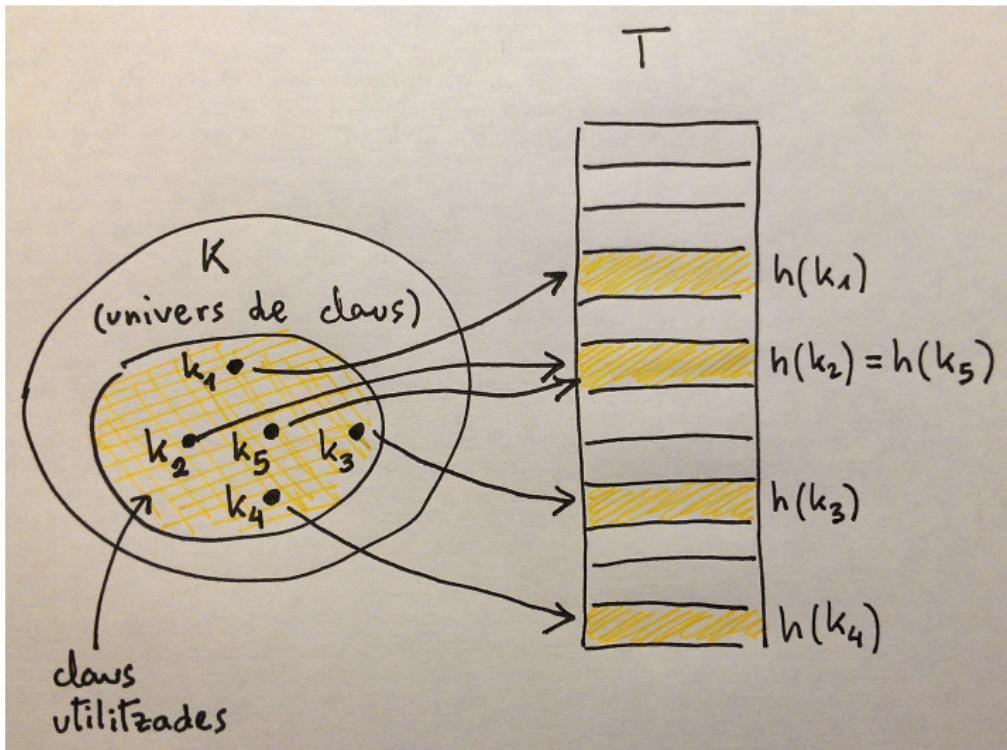
# Taules de dispersió

Suposem que volem emmagatzemar un màxim de  $n$  claus. Declarem una taula  $T$  de  $m \leq n$  posicions.

- Si  $m = n$  i les claus són  $\{0, 1, \dots, m - 1\}$ , usem adreçament directe: l'element de clau  $k$  va a l'espai  $T[k]$
- Si  $m < n$ , usem taules de dispersió:  
l'element de clau  $k$  va a l'espai  $T[h(k)]$ , on  $h$  és la **funció de dispersió**

$$h : K \rightarrow \{0, 1, \dots, m - 1\}$$

# Taules de dispersió



La situació en què dues claus coincideixen en el mateix espai (com  $k_2$  i  $k_5$ ) se'n diu **col·lisió**.

## Exercici

Si  $K$  és el conjunt de totes les claus i  $m$  és el nombre de posicions de la taula de dispersió, volem calcular quin és el nombre màxim de claus que col·lisionen en un mateix valor de dispersió.

Doneu una fita inferior d'aquest valor si

- $|K| > m$ :
- $|K| > 2m$ :
- $|K| > 3m$ :
- $|K| > nm$ :

## Solució

Si  $K$  és el conjunt de totes les claus i  $m$  és el nombre de posicions de la taula de dispersió, volem calcular quin és el nombre màxim de claus que col·lisionen en un mateix valor de dispersió.

Doneu una fita inferior d'aquest valor si

- $|K| > m$ : 2
- $|K| > 2m$ : 3
- $|K| > 3m$ : 4
- $|K| > nm$ : n + 1

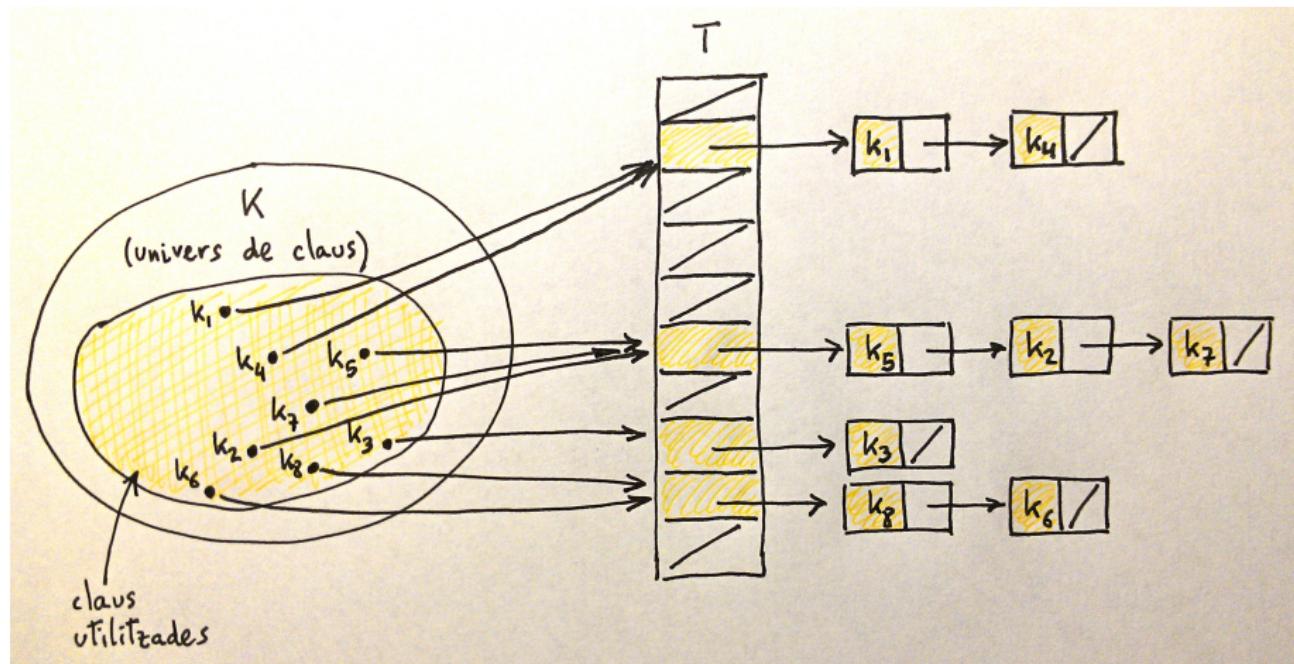
L'ideal seria evitar les col·lisions triant una bona funció de dispersió  $h$ .

Algunes consideracions sobre  $h$ :

- Ha de ser determinista (perquè sempre es comporti igual) però ha de semblar aleatòria (per repartir bé les claus).
- Com que  $|K| > m$ , les col·lisions no es poden evitar.
- Cal triar un mètode per resoldre les col·lisions.

El mètode més simple per resoldre col·lisions és l'**encadenament** (o **encadenament separat**).

# Col·lisions

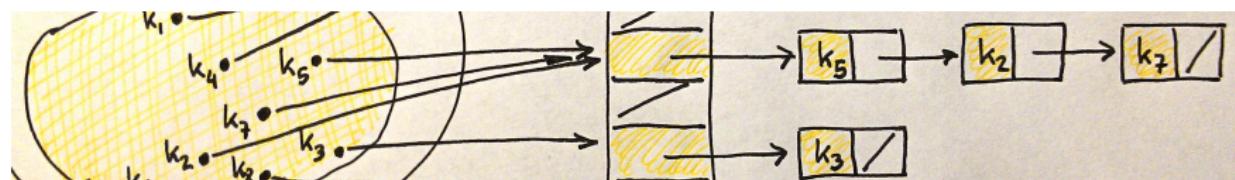


Resolució de col·lisions per encadenament. Cada entrada de la taula  $T[j]$  conté una llista encadenada amb les claus amb valor de dispersió  $j$ . Per exemple,  $h(k_1) = h(k_4)$  i, per tant,  $T[h(k_1)]$  apunta a  $k_1$  seguit de  $k_4$ .

# Encadenament

En l'**encadenament**, els elements que tenen el mateix valor de dispersió es posen en una llista encadenada:

$T[i]$  apunta al cap de la llista dels elements amb valor de dispersió  $h(i)$ .



El cost de fer **una consulta és  $\Theta(n)$  en cas pitjor**: és el cas en què tots els elements tenen el mateix valor de dispersió i formen una sola llista.

En cas mitjà, el cost de fer una consulta és **constant**, assumint que:

- a cada crida amb una nova clau, la funció de dispersió  $h$  retorna un natural entre 0 i  $m - 1$  a l'atzar, independentment de les crides anteriors
- calcular  $h(i)$  triga temps  $\Theta(1)$

# Encadenament

## Proposició

El cost en cas pitjor de fer una **cerca** en l'esquema d'encadenament és  $\Theta(n)$ .

## Corol·lari

El cost en cas pitjor de fer una **assignació** o un **esborrat** (amb cerca prèvia) en l'esquema d'encadenament és  $\Theta(n)$ .

# Encadenament

Suposem que la taula  $T$  té  $m$  posicions i guarda  $n$  elements.

## Definició

El **factor de càrrega**  $\alpha = n/m$  per a  $T$  és el nombre mitjà d'elements per posició.

## Proposició

El cost en cas mitjà de fer una **cerca** en l'esquema d'encadenament és  $\Theta(1 + \alpha)$ .

## Corol·lari

El cost en cas mitjà de fer una **assignació** o un **esborrat** (amb cerca prèvia) en l'esquema d'encadenament és  $\Theta(1 + \alpha)$ .

# Encadenament

Suposem que la taula  $T$  té  $m$  posicions i guarda  $n$  elements.

## Definició

El **factor de càrrega**  $\alpha = n/m$  per a  $T$  és el nombre mitjà d'elements per posició.

## Proposició

El cost en cas mitjà de fer una **cerca** en l'esquema d'encadenament és  $\Theta(1 + \alpha)$ .

## Corol·lari

El cost en cas mitjà de fer una **assignació** o un **esborrat** (amb cerca prèvia) en l'esquema d'encadenament és  $\Theta(1 + \alpha)$ .

## Exemple

Volem emmagatzemar la informació de matrícula dels alumnes:

- Si un nom té  $\leq 20$  caràcters, l'espai de possibles claus és de  $\approx 27^{20}$ .
- El nombre màxim d'alumnes nous és de  $n = 300$ .

Definim un vector de  $m = 300$  posicions, de manera que cada llista de “sinònims” tindrà una talla  $\approx 1$  i, en mitjana, les operacions tindran cost  $\Theta(1)$ .

Però com triem la funció de dispersió?

# Funcions de dispersió

Assumim que les claus són nombres naturals.  
Si no ho són, s'interpreten com a tals.

## Exemple

Donada una cadena de caràcters, l'interpretarem com un natural.  
Donada la cadena CLRS:

- valors en ASCII: C= 67, L= 76, R= 82, S= 83
- hi ha 128 caràcters en ASCII
- per tant, CLRS es transforma en el natural

$$\begin{aligned} 67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 \\ = 141.764.947 \end{aligned}$$

# Funcions de dispersió

## El mètode de la divisió

Dispersem una clau  $k$  en una de les  $m$  posicions a través de la funció

$$h(k) = k \bmod m$$

- **Exemple:** si la taula de dispersió té mida  $m = 12$  i  $k = 100$ , llavors  $h(k) = 4$ .
- **Avantatge:** és força ràpid.
- **Desavantatge:** cal evitar certs valors de  $m$  com  $2^i$
- **Bones tries per a  $m$ :** primers no gaire propers a potències de 2 (per a que les claus quedin ben distribuïdes)

# Funcions de dispersió

## Exemple

Suposem que volem una taula de dispersió amb les col·lisions resoltes per encadenament, que emmagatzemi unes  $n = 2000$  cadenes de caràcters.

No ens fa res examinar uns 3 elements en una cerca fallida, de manera que triem una taula de mida

$$m = 701.$$

Triem 701 perquè és un primer proper a  $2000/3$  però no és una potència de 2. La funció de dispersió serà

$$h(k) = k \bmod 701.$$

## El mètode de la multiplicació

Per dispersar una clau  $k$  en una de les  $m$  posicions:

- ① multipliquem  $k$  per una constant  $A \in \mathbb{R}$  t.q.  $0 < A < 1$  i n'extraiem la part fraccional
- ② llavors, multipliquem el valor per  $m$  i prenem la part entera per defecte:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor.$$

- **Avantatge:** el valor de  $m$  no és crític.
- **Desavantage:** és més lent que el mètode de divisió.
- **Bones tries per a  $m$ :** potències de 2 (fan la implementació fàcil).
- **Bona tria per a  $A$ :** Knuth suggereix  $A \approx (\sqrt{5} - 1)/2 = 0,6180339887\dots$

# Funcions de dispersió

## Exemple

En una taula de dispersió amb  $m = 1024$ ,  $k = 123$  i  $A = 0,61803399$ , tenim

$$\begin{aligned} h(k) &= \lfloor 1024 \cdot (123 \cdot A - \lfloor 123 \cdot A \rfloor) \rfloor \\ &= \lfloor 1024 \cdot (76,0181808 - 76) \rfloor \\ &= \lfloor 1024 \cdot 0,0181808 \rfloor = 18. \end{aligned}$$

Una alternativa a l'encadenament és l'**adreçament obert**:

- tots els elements s'emmagatzemen en la mateixa taula.
- quan cerquem un element,  
examinem les posicions de manera sistemàtica fins a trobar-lo
- la funció de dispersió té dos paràmetres: la clau i la “prova” de posició

$$h : K \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$

La seqüència de proves és  $(h(k, 0), h(k, 1), \dots, h(k, m-1))$ .

# Tema 3. Diccionaris

## 1 Introducció

- Tipus abstractes de dades
- Diccionaris

## 2 Taules de dispersió

- Introducció
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)

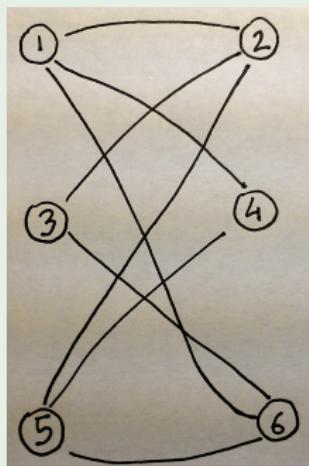
# Definicions i terminologia

## Definició

Un **graf** és un parell  $(V, E)$  on:

- $V$  és un conjunt finit (**vèrtexs** o **nodes**)
- $E$  és un conjunt de parells no ordenats de vèrtexs (**arestes**)

## Exemple de graf (1)



Graf d'amistat en una xarxa social.

- **Connex:**  
es pot arribar de qualsevol vèrtex a qualsevol altre (a través d'arestes).
- **Cíclic:**  
hi ha cicles, com  $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1$ .

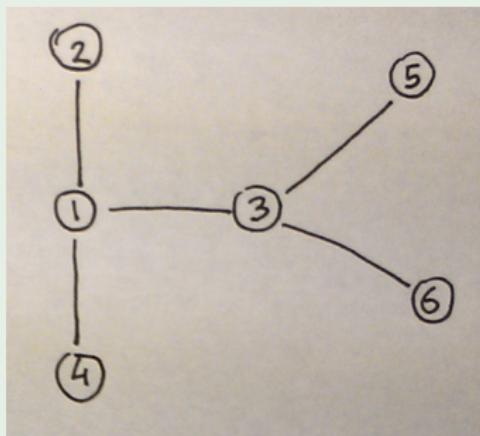
# Definicions i terminologia

## Definició

Un **graf** és un parell  $(V, E)$  on:

- $V$  és un conjunt finit (**vèrtexs** o **nodes**)
- $E$  és un conjunt de parells no ordenats de vèrtexs (**arestes**)

## Exemple de graf (2)



Graf connex i acíclic.

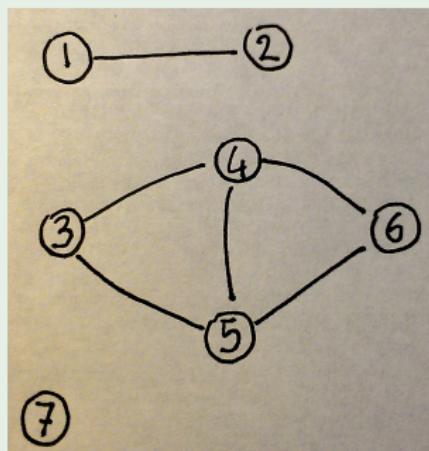
# Definicions i terminologia

## Definició

Un **graf** és un parell  $(V, E)$  on:

- $V$  és un conjunt finit (**vèrtexs** o **nodes**)
- $E$  és un conjunt de parells no ordenats de vèrtexs (**arestes**)

## Exemple de graf (3)



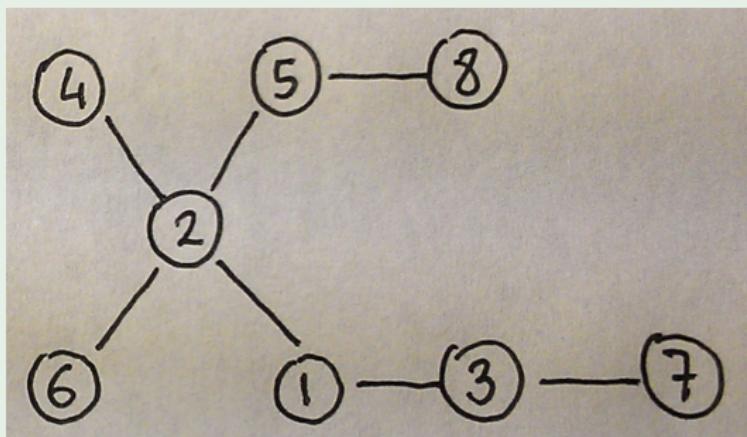
Graf inconnex i cíclic.

# Definicions i terminologia

## Definició

Un **arbre** és un graf connex i acíclic.

## Exemple d'arbres (1)

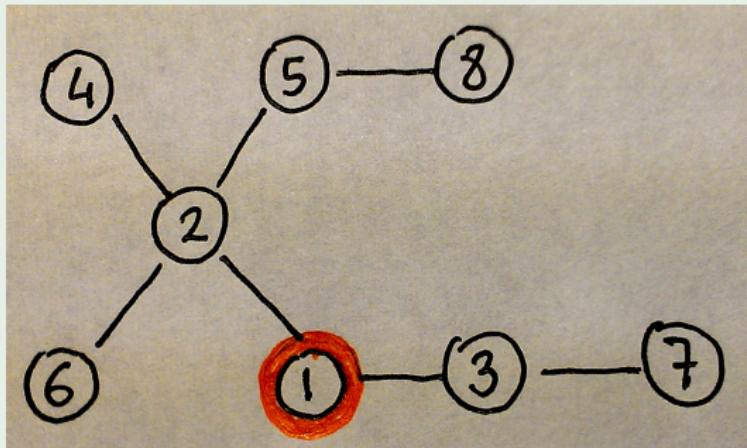


# Definicions i terminologia

## Definició

Un **arbre arrelat** és un graf connex i acíclic amb un vèrtex distingit (l'**arrel**).

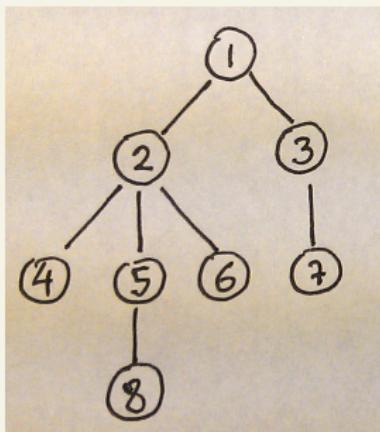
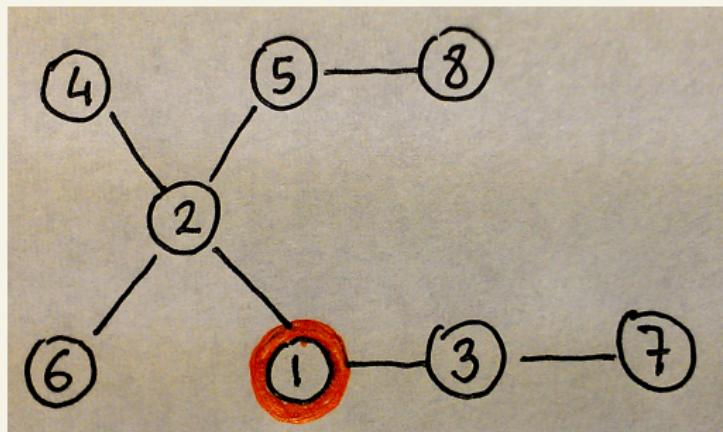
## Exemple d'arbres (2)



# Definicions i terminologia

## Representació

Representarem els arbres arrelats amb el vèrtex distingit a dalt.



# Definicions i terminologia

## Terminologia d'arbres

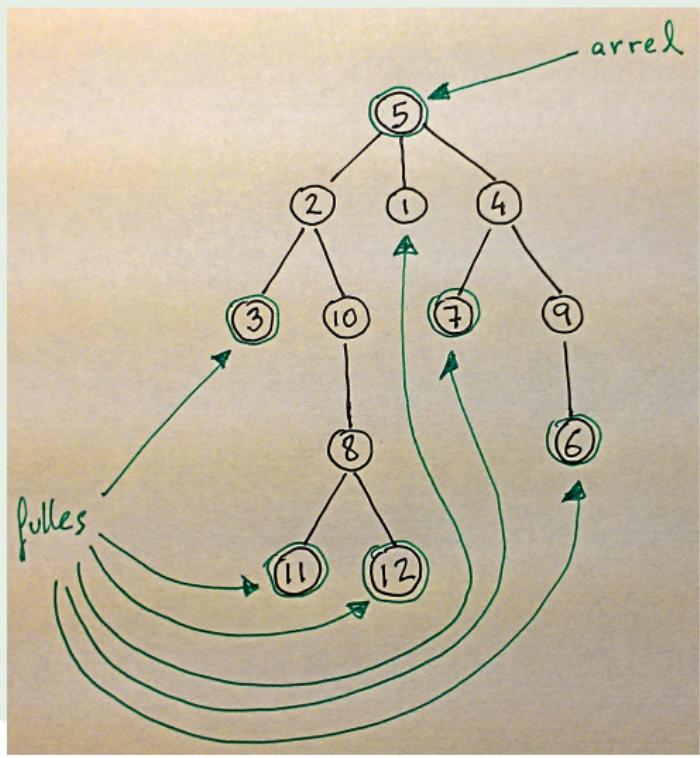
El node distingit s'anomena **arrel**.

Sigui  $x$  un node d'un arbre  $T$  amb arrel  $r$ :

- tot node  $y$  en el camí de  $r$  a  $x$  és un **predecessor** de  $x$
- si  $y$  és un predecessor de  $x$ , llavors  $x$  és un **descendent** de  $y$
- si hi ha l'aresta  $\{y, x\}$  a  $T$ , llavors  $y$  és el **pare** de  $x$  i  $x$  és **fill** de  $y$
- dos nodes amb el mateix pare s'anomenen **germans**
- un node sense fills es diu que és una **fulla**
- si un node no és una fulla, es diu que és un **node intern**
- l'**alçada** (o **profunditat**) de  $T$  és la màxima distància (nombre d'arestes d'un camí) entre l'arrel i una fulla

# Definicions i terminologia

## Exemple d'arbres (3)



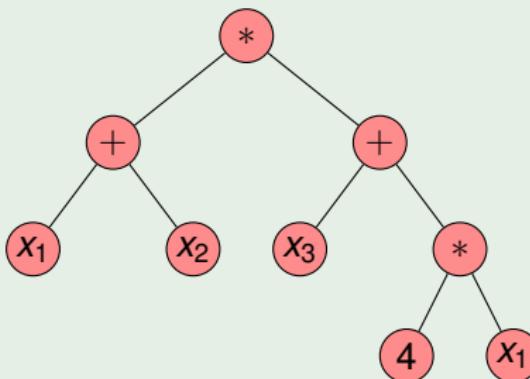
- 4 és **pare** de 9
- els **predecessors** de 10 són 5, 2 i 10
- els **successors** de 10 són 10, 8, 11 i 12
- 3 i 10 són **germans**
- 9 és un **node intern**
- l'arbre té **alçada** 4 (distància entre 5 i 11, o entre 5 i 12)

# Definicions i terminologia

En aquest tema, per **arbre** entendrem **arbre arrelat**.

## Exemple d'arbres (4)

Arbre que representa l'expressió  $(x_1 + x_2) * (x_3 + 4 * x_1)$ :



En l'exemple anterior, cada node té un màxim de dos fills.

# Definicions i terminologia

## Definició

Un **arbre binari** és un arbre on cada node té un màxim de dos fills, que es diferencien com a **fill esquerre** i **fill dret**.

També es poden definir els arbres binaris de manera recursiva.

## Definició

Un **arbre binari** és una estructura amb un conjunt finit de nodes tal que:

- no conté cap node o
- està formada per tres conjunts de nodes:
  - l'**arrel**,
  - un arbre binari anomenat **subarbre esquerre**, i
  - un arbre binari anomenat **subarbre dret**.

# Definicions i terminologia

## Definició

Un **arbre binari** és un arbre on cada node té un màxim de dos fills, que es diferencien com a **fill esquerre** i **fill dret**.

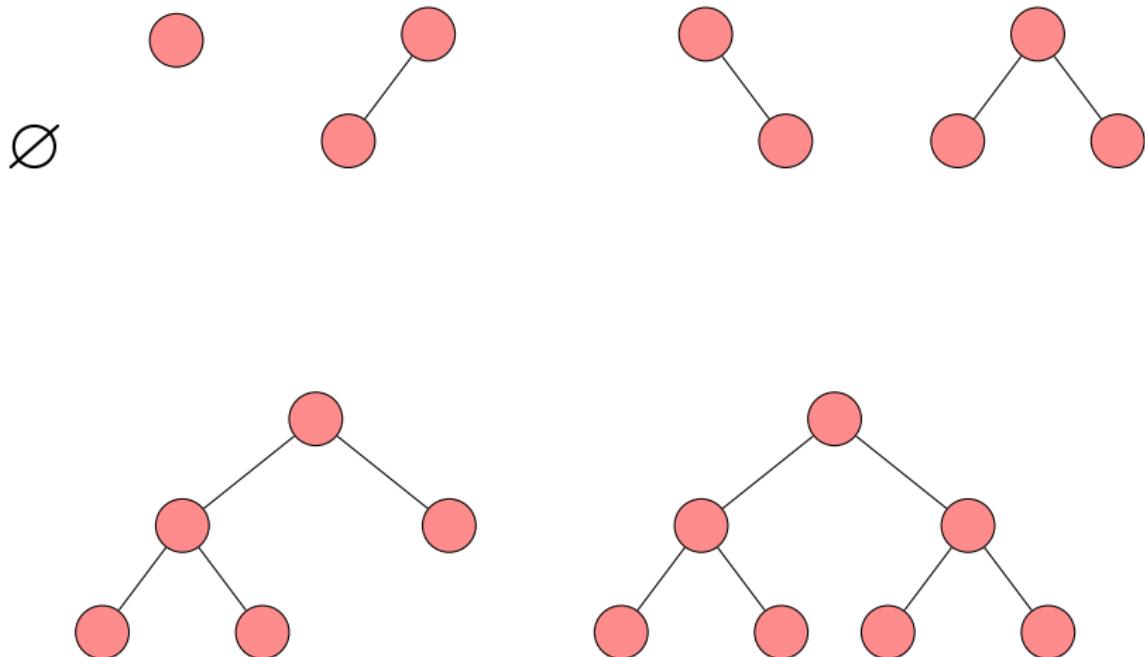
També es poden definir els arbres binaris de manera recursiva.

## Definició

Un **arbre binari** és una estructura amb un conjunt finit de nodes tal que:

- no conté cap node o
- està formada per tres conjunts de nodes:
  - l'**arrel**,
  - un arbre binari anomenat **subarbre esquerre**, i
  - un arbre binari anomenat **subarbre dret**.

# Arbres binaris



# Tema 3. Diccionaris

## 1 Introducció

- Tipus abstractes de dades
- Diccionaris

## 2 Taules de dispersió

- Introducció
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)

# Introducció

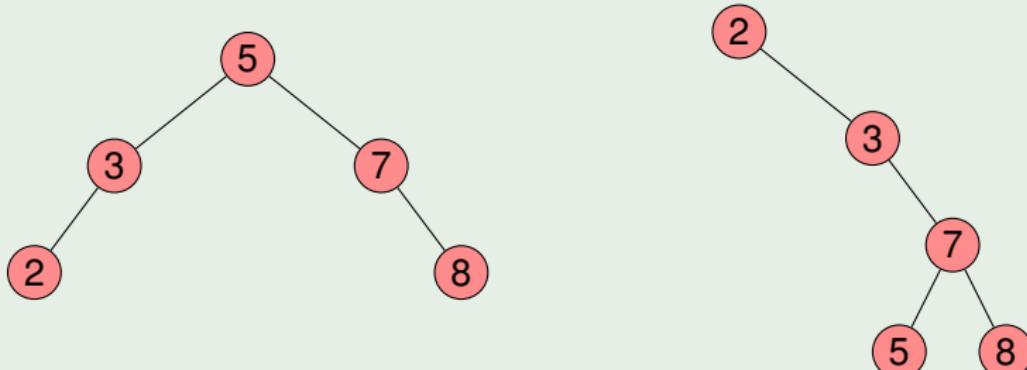
A partir d'ara suposarem que les claus formen un **conjunt totalment ordenat**.

## Definició

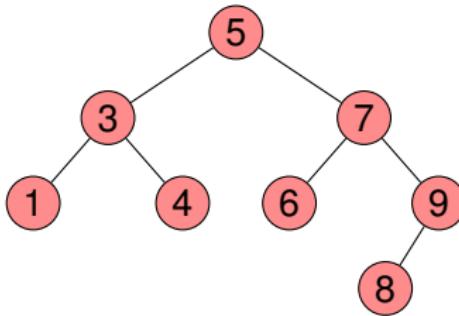
Un **arbre binari de cerca** (ABC, *Binary Search Tree* o BST en anglès) és un arbre binari que té una clau associada a cada node i que compleix la **propietat ABC**: la clau d'un *node qualsevol* és

- més gran que la de tots els nodes del seu subarbre esquerre i
- més petita que la de tots els nodes del seu subarbre dret.

## Exemple



Exemple de **consulta** en un ABC:

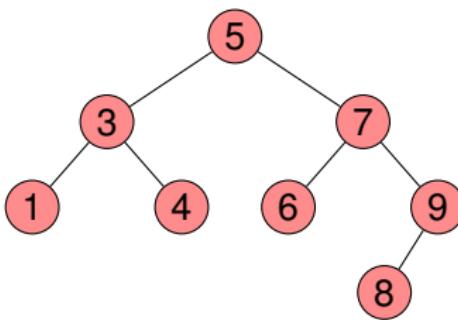


Volem saber si el diccionari conté la clau 6:

- Miro l'arrel. Com que  $5 < 6$ , cal mirar al subarbre de la dreta.
- Miro l'arrel. Com que  $7 > 6$ , cal mirar al subarbre de l'esquerra.
- Miro l'arrel. Com que  $6 = 6$ , ja hem acabat.
- Hem trobat la clau!

El nombre de comparacions de claus és  $\leq$  l'alçada de l'arbre.

Exemple d'**inserció** en un ABC:

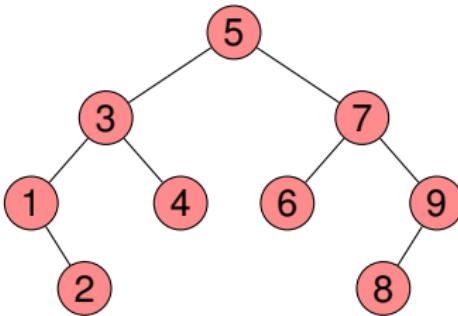


Volem inserir al diccionari la clau 2:

- Miro l'arrel. Com que  $5 > 2$ , cal inserir al subarbre de l'esquerra.
- Miro l'arrel. Com que  $3 > 2$ , cal inserir al subarbre de l'esquerra.
- Miro l'arrel. Com que  $1 < 2$ , cal inserir al subarbre de la dreta.
- Cal inserir a l'arbre buit: només cal afegir un node amb 2

El nombre de comparacions de claus és  $\leq$  l'alçada de l'arbre.

Exemple d'**inserció** en un ABC:



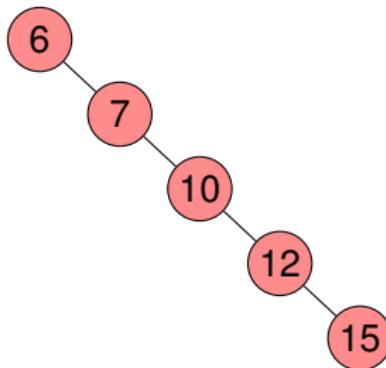
Volem inserir al diccionari la clau 2:

- Miro l'arrel. Com que  $5 > 2$ , cal inserir al subarbre de l'esquerra.
- Miro l'arrel. Com que  $3 > 2$ , cal inserir al subarbre de l'esquerra.
- Miro l'arrel. Com que  $1 < 2$ , cal inserir al subarbre de la dreta.
- Cal inserir a l'arbre buit: només cal afegir un node amb 2

El nombre de comparacions de claus és  $\leq$  l'alçada de l'arbre.

- En general, els ABC permeten fer les **operacions bàsiques de diccionaris (consultar, assignar, esborrar)** en temps proporcional a l'alçada de l'arbre.
- L' **alçada esperada d'un ABC** de  $n$  nodes és de  $\Theta(\log n)$ .
- Per tant, les operacions bàsiques dels diccionaris tenen un **cost mitjà de  $\Theta(\log n)$**  fets amb ABC.

- Pel cas pitjor, tenim que  $h$  pot coincidir amb el nombre de nodes:



- Per tant, el cost en el cas pitjor de les operacions bàsiques en ABC és  $\Theta(n)$ , on  $n$  és el nombre de nodes.

## Exercici

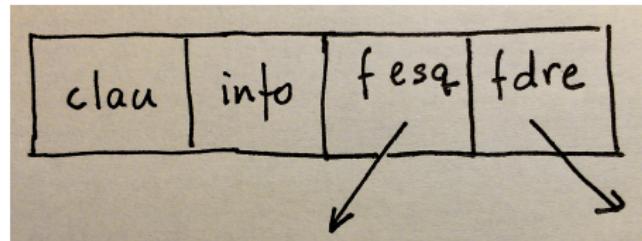
Demostreu que un ABC amb  $n$  nodes es pot reestructurar de manera que tingui alçada  $\Theta(\log n)$ .

# Implementació (ABC)

Implementació: *Algorismes en C++ (EDA)*, J. Petit, S. Roura, A. Atserias.

Representació:

- Un **node** és:



- Un **diccionari** implementat amb un ABC té dos camps:
  - el nombre de nodes de l'ABC
  - un punter a l'arrel de l'ABC

Els **costos** que es donen:

- es refereixen al cas pitjor
- depenen de  $n$ , que és el nombre d'elements (nodes).

# Implementació (ABC)

## Definició de Diccionari i Node

Un Node emmagatzema

una clau, la seva informació associada i dos punters a dos altres Node.

```
template <typename Clau, typename Info>
class Diccionari {
private:
    struct Node {
        Clau clau;
        Info info;
        Node* fesq; // Punter al fill esquerre
        Node* fdre; // Punter al fill dret
        Node (const Clau& c, const Info& i, Node* fe, Node* fd)
            : clau(c), info(i), fesq(fe), fdre(fd) { }
    };
    int n;           // Nombre d'elements en l'ABC
    Node* arrel;   // Punter a l'arrel de l'ABC
```

# Implementació (ABC): funcions públiques

Constructores de creació i còpia / Destructora

Crear Diccionari:  $\Theta(1)$ .

```
Diccionari () {  
    n = 0;  
    arrel = null;  
}
```

Fer una còpia:  $\Theta(n)$ .

```
Diccionari (const Diccionari& d) {  
    n = d.n;  
    arrel = copia(d.arrel);  
}
```

Destructora:  $\Theta(n)$ .

```
~Diccionari () {  
    alliberar(arrel);  
}
```

# Implementació (ABC): funcions públiques

## Constructora d'assignació

Redefinició de l'assignació:  $\Theta(n + d.n)$ .

```
Diccionari& operator= (const Diccionari& d) {  
    if (&d != this) {  
        alliberar(arrel);  
        n = d.n;  
        arrel = copia(d.arrel);  
    }  
    return *this;  
}
```

# Implementació (ABC): funcions públiques

## Assignar i esborrar

Assignar a clau el valor info:  $\Theta(n)$ .

```
void assignar (const Clau& clau, const Info& info) {  
    assignar(arrel, clau, info);  
}
```

Esborrar clau i la informació associada (si no hi és, no canvia):  $\Theta(n)$ .

```
void esborrar (const Clau& clau) {  
    esborrar_3(arrel, clau);  
}
```

# Implementació (ABC): funcions públiques

## Consultes

Donada una clau, retornar la referència a la informació associada:  $\Theta(n)$ .

```
Info& consulta (const Clau& clau) {
    if (Node* p = cerca(arrel, clau)) {
        return p->info;
    } else {
        throw ErrorPrec("La clau no era present");
    }
}
```

Indicar si la clau hi és o no present:  $\Theta(n)$ .

```
bool present (const Clau& clau) {
    return cerca(arrel, clau) != null;
}
```

Retornar la talla del diccionari:  $\Theta(1)$ .

```
int talla () {
    return n;
}
```

# Implementació (ABC): funcions privades

Suposarem que l'arbre a tractar (apuntat per  $p$ ) té

- $s$  nodes i
- alçada  $h$ .

Els costos en el cas pitjor vindran donats per  $\Theta(s)$  o  $\Theta(h)$ .

# Implementació (ABC): funcions privades

## Esborrar

Eliminar l'arbre apuntat per  $p$ :  $\Theta(s)$ .

```
static void alliberar (Node* p) {  
    if (p) {  
        alliberar(p->esq);  
        alliberar(p->fdre);  
        delete p;  
    } }
```

## Copiar

Retornar un punter a una còpia de l'arbre apuntat per  $p$ :  $\Theta(s)$ .

```
static Node* copia (Node* p) {  
    return p ? new Node(p->clau, p->info,  
                        copia(p->fesq), copia(p->fdre))  
             : null;  
}
```

# Implementació (ABC): funcions privades

## Cerca

Retornar un punter al node de l'arbre apuntat per `p` que conté `clau` (o `null` si no hi és):  $\Theta(h)$ .

```
static Node* cerca (Node* p, const Clau& clau) {  
    if (p) {  
        if (clau < p->clau) {  
            return cerca(p->fesq, clau);  
        } else if (clau > p->clau) {  
            return cerca(p->fdre, clau);  
        } }  
    return p;  
}
```

La cerca es fa baixant pels subarbres adequats fent ús de la propietat ABC

# Implementació (ABC): funcions privades

## Assignar

Assignar `info` a `clau` si la clau és al subarbre apuntat per `p`; si no hi és, afegir un nou node amb `clau` i `info`:  $\Theta(h)$ .

```
void assignar
    (Node*& p, const Clau& clau, const Info& info) {
if (p) {
    if (clau < p->clau) {
        assignar(p->fesq, clau, info);
    } else if (clau > p->clau) {
        assignar(p->fdre, clau, info);
    } else {
        p->info = info;
    }
} else {
    p = new Node(clau, info, 0, 0);
    ++n;
}
}
```

# Implementació (ABC): funcions privades

## Mínim

Retornar un punter al node que conté el valor mínim en el subarbre apuntat per p (suposant que p no és nul):  $\Theta(h)$ .

```
static Node* minim (Node* p) {  
    return p->fesq ? minim(p->fesq) : p;  
}
```

## Màxim

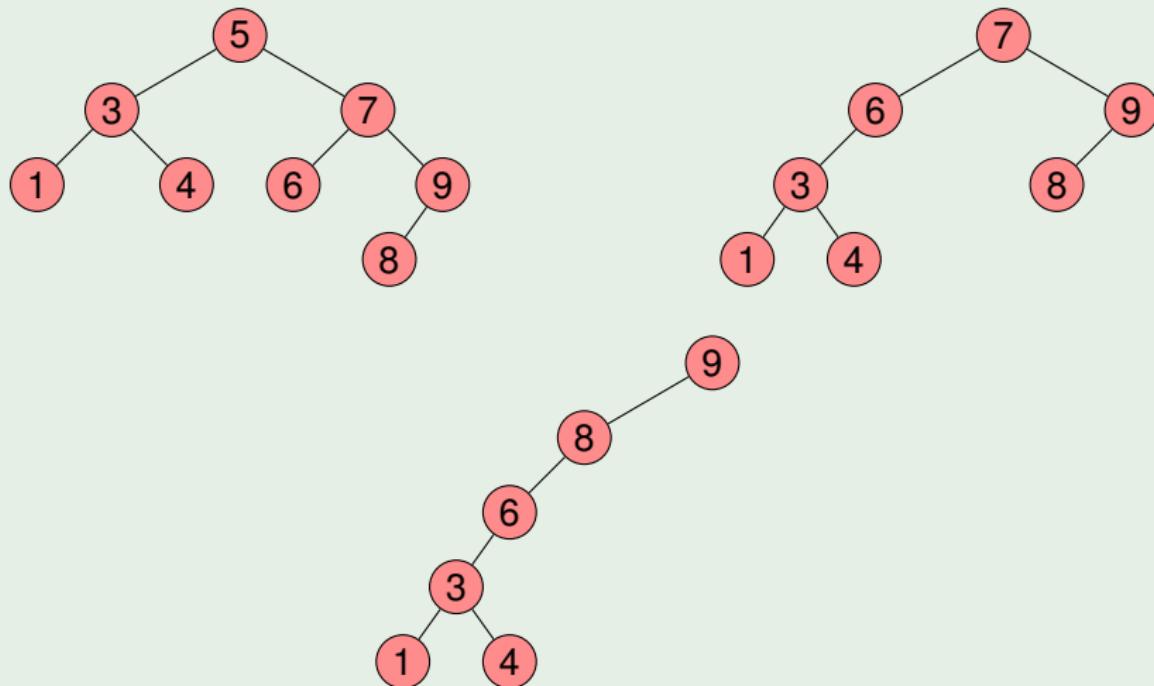
Retornar un punter al node que conté el valor màxim en el subarbre apuntat per p (suposant que p no és nul):  $\Theta(h)$ .

```
static Node* maxim (Node* p) {  
    while (p->fdre) p = p->fdre;  
    return p;  
}
```

# Implementació (ABC): funcions privades

## Exemple

Esborrat del 5 i el 7. Els arbres es degraden aviat.



# Implementació (ABC): funcions privades

## Esborrar (1)

Esborrar el node que conté `clau` en el subarbre apuntat per `p`:  $\Theta(h)$ .

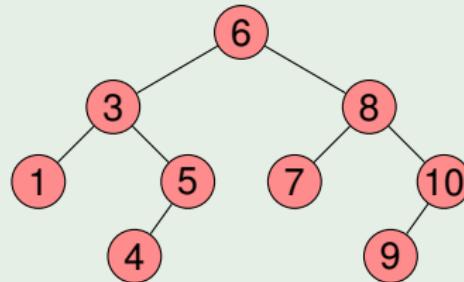
Es penja el subarbre esquerre com a nou fill esquerre del mínim del fill dret.

```
void esborrar_1 (Node*& p, const Clau& clau) {
    if (p) {
        if (clau<p->clau) {
            esborrar_1(p->fesq, clau);
        } else if (clau>p->clau) {
            esborrar_1(p->fdre, clau);
        } else {
            Node* q = p;
            if (!p->fesq) p = p->fdre;
            else if (!p->fdre) p = p->fesq;
            else {
                Node* m = minim(p->fdre);
                m->fesq = p->fesq;
                p = p->fdre;
            }
            delete q; --n;
        }
    }
}
```

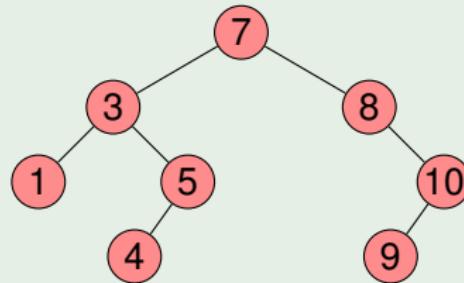
# Implementació (ABC): funcions privades

## Example

Donat l'arbre



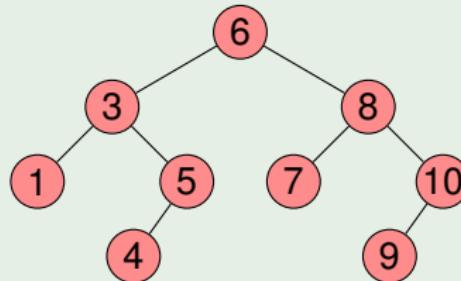
n'esborren l'arrel



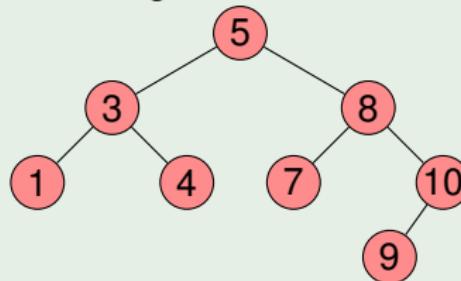
# Implementació (ABC): funcions privades

## Example

Dona't l'arbre



per esborrar l'arrel, tenim una segona alternativa



## Esborrar (2)

Esborrar el node que conté `clau` en el subarbre apuntat per `p`:  $\Theta(h)$ .

Es copia el mínim del fill dret al node esborrat i després també s'esborra.

```
void esborrar_2 (Node*& p, const Clau& clau) {  
    if (p) {  
        if (clau < p->clau) {  
            esborrar_2(p->fesq, clau);  
        } else if (clau > p->clau) {  
            esborrar_2(p->fdre, clau);  
        } else {  
            Node* mínim = p->fdre;  
            while (mínim->fesq != NULL)  
                mínim = mínim->fesq;  
            p->clau = mínim->clau;  
            esborrar_2(mínim, clau);  
        }  
    }  
}
```

# Implementació (ABC): funcions privades

## Esborrar (2), cont.

```
} else if (!p->fesq) {  
    Node* q = p; p = p->fdre;  
    delete q; --n;  
} else if (!p->fdre) {  
    Node* q = p; p = p->fesq;  
    delete q; --n;  
} else {  
    Node* m = minim(p->fdre);  
    p->clau = m->clau; p->info = m->info;  
    esborrar_2(p->fdre, m->clau);  
} } }
```

Inconvenient: es copien claus i informacions, més costós que copiar punters.

# Implementació (ABC): funcions privades

## Esborrar (3)

Esborrar el node que conté `clau` en el subarbre apuntat per `p`:  $\Theta(h)$ .  
S'esborra el mínim del fill dret, que serà la nova arrel.

```
void esborrar_3 (Node*& p, const Clau& clau) {
    if (p) {
        if (clau<p->clau) {
            esborrar_3(p->fesq, clau);
        } else if (clau>p->clau) {
            esborrar_3(p->fdre, clau);
        } else {
            Node* q = p;
            if (!p->fesq) p = p->fdre;
            else if (!p->fdre) p = p->fesq;
            else {Node* m = esborrar_minim(p->fdre);
                   m->fesq = p->fesq; m->fdre = p->fdre;
                   p = m;}
            delete q; --n; } } }
```

# Implementació (ABC): funcions privades

## Esborrar mínim

Esborrar i retornar el node que conté l'element mínim de p:  $\Theta(h)$ .

```
Node* esborrar_minim (Node*& p) {  
    if (p->fesq) {  
        return esborrar_minim(p->fesq);  
    } else {  
        Node* q = p;  
        p = p->fdre;  
        return q;  
    } }
```

# Tema 3. Diccionaris

## 1 Introducció

- Tipus abstractes de dades
- Diccionaris

## 2 Taules de dispersió

- Introducció
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

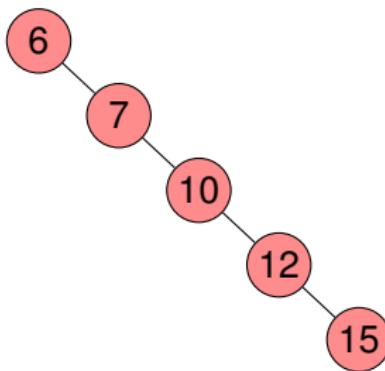
## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)

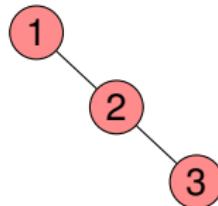
En els **arbres binaris de cerca**, hem vist que el cost de les operacions bàsiques és  $\Theta(h)$ , on  $h$  és l'alçada. Però  $h$  pot coincidir amb el nombre de nodes.



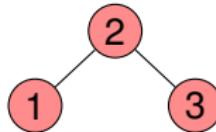
Per tant, el cost en el cas pitjor és  $\Theta(n)$ , on  $n$  és el nombre de nodes.

Per millorar el cost lineal, cal que l'alçada es mantingui en  $\approx \log n$ .

En lloc de permetre arbres “desequilibrats” com



caldria tenir arbres “equilibrats” com



# Introducció

Com mantenir els arbres equilibrats?

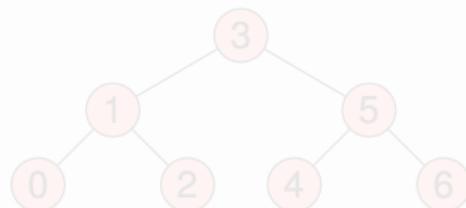
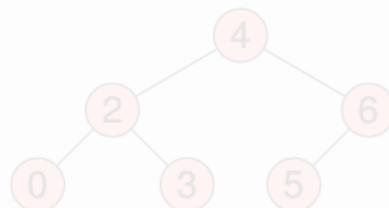
- 1 Fent que l'ABC sigui complet.

Un arbre complet és aquell que té tots els nivells plens, tret potser de l'últim, on els nodes estan el més a l'esquerra possible.

Però llavors afegir un element pot ser massa costós.

## Exemple

Per afegir l'element 1 en el primer arbre, cal canviar-ho gairebé tot!



# Introducció

Com mantenir els arbres equilibrats?

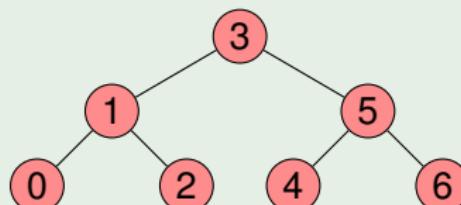
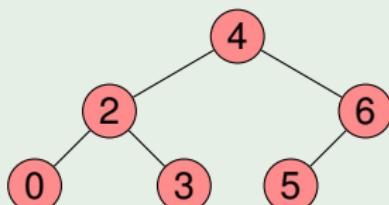
- 1 Fent que l'ABC sigui complet.

Un arbre complet és aquell que té tots els nivells plens, tret potser de l'últim, on els nodes estan el més a l'esquerra possible.

Però llavors afegir un element pot ser massa costós.

## Exemple

Per afegir l'element 1 en el primer arbre, cal canviar-ho gairebé tot!

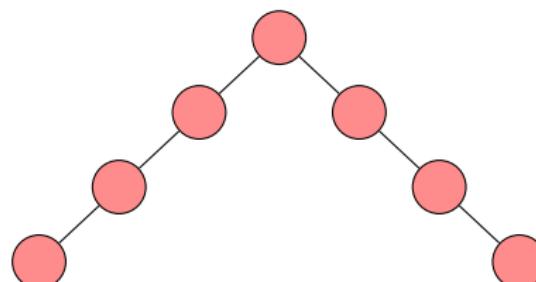


Com mantenir els arbres equilibrats?

2 Demanant que els dos subarbres de l'arrel tinguin la mateixa alçada.

Però és una condició insuficient:

el cost de les operacions bàsiques en l'arbre següent és de  $\approx n/2$ .



Com mantenir els arbres equilibrats?

- 3 Permetent un **petit desequilibri** en les alçades dels subarbres esquerre i dret: una diferència màxima d'1.

Però cal fer-ho per a tots els nodes!

## Definició

Un arbre binari està **equilibrat** si

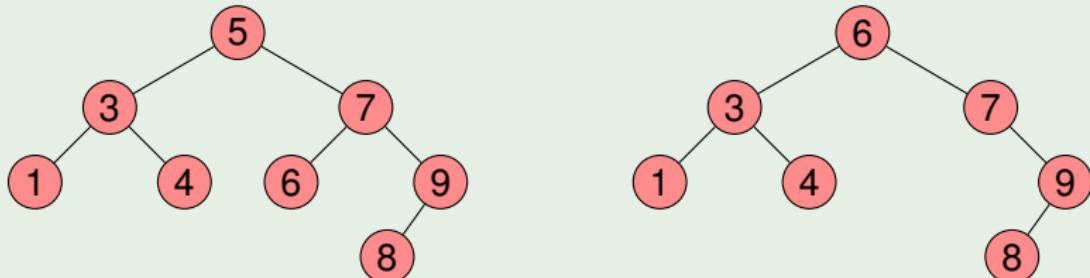
- és buit o
- la diferència d'alçades dels seus subarbres és com a màxim d'1, i els seus subarbres també estan equilibrats.

# Introducció

Els ABC amb la condició d'equilibri es diuen **arbres AVL**  
(de **Adelson-Velskii i Landis**)

## Exemple

L'arbre de l'esquerra està equilibrat, però si esborrem l'arrel com en els ABC, l'arbre resultant ja no està equilibrat.



## Teorema

Un arbre AVL amb  $n$  nodes té alçada  $\Theta(\log n)$ .

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $\Omega(\log n)$ .

Un arbre binari d'alçada  $h$  té com a molt  $1 + 2 + \dots + 2^h = 2^{h+1} - 1$  nodes.

Per tant, si un arbre AVL d'alçada  $h$  té  $n$  nodes llavors

$$n \leq 2^{h+1} - 1$$

i per tant

$$\log_2(n + 1) - 1 \leq h$$

Així doncs  $h = \Omega(\log n)$

# Introducció

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $O(\log n)$ .

Sigui  $T(h)$  el nombre mínim de nodes d'un arbre AVL amb alçada  $h$ .

Construirem recursivament un arbre AVL  $T$  amb el mínim de nodes entre els AVL d'alçada  $h$ .

Si  $T$  ha de tenir alçada  $h$ , o el fill esquerre  $E$  o el fill dret  $D$  té alçada  $h - 1$ .

Suposem que és l'esquerre.

Prenem com a  $E$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 1$ .

Farem que  $D$  tingui la menor alçada possible.

Com que  $T$  ha de ser AVL,  $D$  ha de tenir alçada  $h - 2$ .

Prenem com a  $D$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 2$ .

Tenim doncs la recurrència  $T(h) = T(h - 1) + T(h - 2) + 1$ .

Per tant  $T(h) \geq 2T(h - 2)$ .

Com que  $T(0) = 1$  i  $T(1) = 2$ , per inducció  $T(h) \geq \sqrt{2}^h$ .

Per tant, un arbre AVL amb  $n$  nodes tindrà una alçada  $h$  tal que  $n \geq \sqrt{2}^h$ , i per tant  $\log_{\sqrt{2}} n \geq h$ . O sigui que  $h = O(\log n)$ .

# Introducció

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $O(\log n)$ .

Sigui  $T(h)$  el nombre mínim de nodes d'un arbre AVL amb alçada  $h$ .

Construirem recursivament un arbre AVL  $T$  amb el mínim de nodes entre els AVL d'alçada  $h$ .

Si  $T$  ha de tenir alçada  $h$ , o el fill esquerre  $E$  o el fill dret  $D$  té alçada  $h - 1$ .

Suposem que és l'esquerre.

Prenem com a  $E$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 1$ .

Farem que  $D$  tingui la menor alçada possible.

Com que  $T$  ha de ser AVL,  $D$  ha de tenir alçada  $h - 2$ .

Prenem com a  $D$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 2$ .

Tenim doncs la recurrència  $T(h) = T(h - 1) + T(h - 2) + 1$ .

Per tant  $T(h) \geq 2T(h - 2)$ .

Com que  $T(0) = 1$  i  $T(1) = 2$ , per inducció  $T(h) \geq \sqrt{2}^h$ .

Per tant, un arbre AVL amb  $n$  nodes tindrà una alçada  $h$  tal que  $n \geq \sqrt{2}^h$ , i per tant  $\log_{\sqrt{2}} n \geq h$ . O sigui que  $h = O(\log n)$ .

# Introducció

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $O(\log n)$ .

Sigui  $T(h)$  el nombre mínim de nodes d'un arbre AVL amb alçada  $h$ .

Construirem recursivament un arbre AVL  $T$  amb el mínim de nodes entre els AVL d'alçada  $h$ .

Si  $T$  ha de tenir alçada  $h$ , o el fill esquerre  $E$  o el fill dret  $D$  té alçada  $h - 1$ .

Suposem que és l'esquerre.

Prenem com a  $E$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 1$ .

Farem que  $D$  tingui la menor alçada possible.

Com que  $T$  ha de ser AVL,  $D$  ha de tenir alçada  $h - 2$ .

Prenem com a  $D$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 2$ .

Tenim doncs la recurrència  $T(h) = T(h - 1) + T(h - 2) + 1$ .

Per tant  $T(h) \geq 2T(h - 2)$ .

Com que  $T(0) = 1$  i  $T(1) = 2$ , per inducció  $T(h) \geq \sqrt{2}^h$ .

Per tant, un arbre AVL amb  $n$  nodes tindrà una alçada  $h$  tal que  $n \geq \sqrt{2}^h$ , i per tant  $\log_{\sqrt{2}} n \geq h$ . O sigui que  $h = O(\log n)$ .

# Introducció

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $O(\log n)$ .

Sigui  $T(h)$  el nombre mínim de nodes d'un arbre AVL amb alçada  $h$ .

Construirem recursivament un arbre AVL  $T$  amb el mínim de nodes entre els AVL d'alçada  $h$ .

Si  $T$  ha de tenir alçada  $h$ , o el fill esquerre  $E$  o el fill dret  $D$  té alçada  $h - 1$ .

Suposem que és l'esquerre.

Prenem com a  $E$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 1$ .

Farem que  $D$  tingui la menor alçada possible.

Com que  $T$  ha de ser AVL,  $D$  ha de tenir alçada  $h - 2$ .

Prenem com a  $D$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 2$ .

Tenim doncs la recurrència  $T(h) = T(h - 1) + T(h - 2) + 1$ .

Per tant  $T(h) \geq 2T(h - 2)$ .

Com que  $T(0) = 1$  i  $T(1) = 2$ , per inducció  $T(h) \geq \sqrt{2}^h$ .

Per tant, un arbre AVL amb  $n$  nodes tindrà una alçada  $h$  tal que  $n \geq \sqrt{2}^h$ , i per tant  $\log_{\sqrt{2}} n \geq h$ . O sigui que  $h = O(\log n)$ .

# Introducció

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $O(\log n)$ .

Sigui  $T(h)$  el nombre mínim de nodes d'un arbre AVL amb alçada  $h$ .

Construirem recursivament un arbre AVL  $T$  amb el mínim de nodes entre els AVL d'alçada  $h$ .

Si  $T$  ha de tenir alçada  $h$ , o el fill esquerre  $E$  o el fill dret  $D$  té alçada  $h - 1$ .

Suposem que és l'esquerre.

Prenem com a  $E$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 1$ .

Farem que  $D$  tingui la menor alçada possible.

Com que  $T$  ha de ser AVL,  $D$  ha de tenir alçada  $h - 2$ .

Prenem com a  $D$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 2$ .

Tenim doncs la recurrència  $T(h) = T(h - 1) + T(h - 2) + 1$ .

Per tant  $T(h) \geq 2T(h - 2)$ .

Com que  $T(0) = 1$  i  $T(1) = 2$ , per inducció  $T(h) \geq \sqrt{2}^h$ .

Per tant, un arbre AVL amb  $n$  nodes tindrà una alçada  $h$  tal que  $n \geq \sqrt{2}^h$ , i per tant  $\log_{\sqrt{2}} n \geq h$ . O sigui que  $h = O(\log n)$ .

# Introducció

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $O(\log n)$ .

Sigui  $T(h)$  el nombre mínim de nodes d'un arbre AVL amb alçada  $h$ .

Construirem recursivament un arbre AVL  $T$  amb el mínim de nodes entre els AVL d'alçada  $h$ .

Si  $T$  ha de tenir alçada  $h$ , o el fill esquerre  $E$  o el fill dret  $D$  té alçada  $h - 1$ .

Suposem que és l'esquerre.

Prenem com a  $E$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 1$ .

Farem que  $D$  tingui la menor alçada possible.

Com que  $T$  ha de ser AVL,  $D$  ha de tenir alçada  $h - 2$ .

Prenem com a  $D$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 2$ .

Tenim doncs la recurrència  $T(h) = T(h - 1) + T(h - 2) + 1$ .

Per tant  $T(h) \geq 2T(h - 2)$ .

Com que  $T(0) = 1$  i  $T(1) = 2$ , per inducció  $T(h) \geq \sqrt{2}^h$ .

Per tant, un arbre AVL amb  $n$  nodes tindrà una alçada  $h$  tal que  $n \geq \sqrt{2}^h$ , i per tant  $\log_{\sqrt{2}} n \geq h$ . O sigui que  $h = O(\log n)$ .

# Introducció

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $O(\log n)$ .

Sigui  $T(h)$  el nombre mínim de nodes d'un arbre AVL amb alçada  $h$ .

Construirem recursivament un arbre AVL  $T$  amb el mínim de nodes entre els AVL d'alçada  $h$ .

Si  $T$  ha de tenir alçada  $h$ , o el fill esquerre  $E$  o el fill dret  $D$  té alçada  $h - 1$ .

Suposem que és l'esquerre.

Prenem com a  $E$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 1$ .

Farem que  $D$  tingui la menor alçada possible.

Com que  $T$  ha de ser AVL,  $D$  ha de tenir alçada  $h - 2$ .

Prenem com a  $D$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 2$ .

Tenim doncs la recurrència  $T(h) = T(h - 1) + T(h - 2) + 1$ .

Per tant  $T(h) \geq 2T(h - 2)$ .

Com que  $T(0) = 1$  i  $T(1) = 2$ , per inducció  $T(h) \geq \sqrt{2}^h$ .

Per tant, un arbre AVL amb  $n$  nodes tindrà una alçada  $h$  tal que  $n \geq \sqrt{2}^h$ , i per tant  $\log_{\sqrt{2}} n \geq h$ . O sigui que  $h = O(\log n)$ .

# Introducció

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $O(\log n)$ .

Sigui  $T(h)$  el nombre mínim de nodes d'un arbre AVL amb alçada  $h$ .

Construirem recursivament un arbre AVL  $T$  amb el mínim de nodes entre els AVL d'alçada  $h$ .

Si  $T$  ha de tenir alçada  $h$ , o el fill esquerre  $E$  o el fill dret  $D$  té alçada  $h - 1$ .

Suposem que és l'esquerre.

Prenem com a  $E$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 1$ .

Farem que  $D$  tingui la menor alçada possible.

Com que  $T$  ha de ser AVL,  $D$  ha de tenir alçada  $h - 2$ .

Prenem com a  $D$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 2$ .

Tenim doncs la recurrència  $T(h) = T(h - 1) + T(h - 2) + 1$ .

Per tant  $T(h) \geq 2T(h - 2)$ .

Com que  $T(0) = 1$  i  $T(1) = 2$ , per inducció  $T(h) \geq \sqrt{2}^h$ .

Per tant, un arbre AVL amb  $n$  nodes tindrà una alçada  $h$  tal que  $n \geq \sqrt{2}^h$ , i per tant  $\log_{\sqrt{2}} n \geq h$ . O sigui que  $h = O(\log n)$ .

# Introducció

Anem a veure que un arbre AVL amb  $n$  nodes té alçada  $O(\log n)$ .

Sigui  $T(h)$  el nombre mínim de nodes d'un arbre AVL amb alçada  $h$ .

Construirem recursivament un arbre AVL  $T$  amb el mínim de nodes entre els AVL d'alçada  $h$ .

Si  $T$  ha de tenir alçada  $h$ , o el fill esquerre  $E$  o el fill dret  $D$  té alçada  $h - 1$ .

Suposem que és l'esquerre.

Prenem com a  $E$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 1$ .

Farem que  $D$  tingui la menor alçada possible.

Com que  $T$  ha de ser AVL,  $D$  ha de tenir alçada  $h - 2$ .

Prenem com a  $D$  l'AVL amb el mínim de nodes entre els AVL d'alçada  $h - 2$ .

Tenim doncs la recurrència  $T(h) = T(h - 1) + T(h - 2) + 1$ .

Per tant  $T(h) \geq 2T(h - 2)$ .

Com que  $T(0) = 1$  i  $T(1) = 2$ , per inducció  $T(h) \geq \sqrt{2}^h$ .

Per tant, un arbre AVL amb  $n$  nodes tindrà una alçada  $h$  tal que  $n \geq \sqrt{2}^h$ , i per tant  $\log_{\sqrt{2}} n \geq h$ . O sigui que  $h = O(\log n)$ .

# Implementació (AVL)

## Definició de Diccionari i Node

Com la dels ABC però afegint-hi l'alçada.

```
template <typename Clau, typename Info>
class Diccionari {
private:
    struct Node {
        Clau clau;
        Info info;
        Node* fesq; // Punter al fill esquerre
        Node* fdre; // Punter al fill dret
        int alc;    // Alçada de l'arbre
        Node (const Clau& c, const Info& i,
              Node* fe, Node* fd, int a)
            : clau(c), info(i), fesq(fe), fdre(fd), alc(a) {} };
    int n;          // Nombre d'elements en l'AVL
    Node* arrel; // Punter a l'arrel de l'AVL
```

Les funcions públiques i les privades **alliberar**, **copia** i **cerca** són iguals que en els ABC.

Pels AVL calen les dues funcions següents referents a l'alçada, de cost  $\Theta(1)$ .

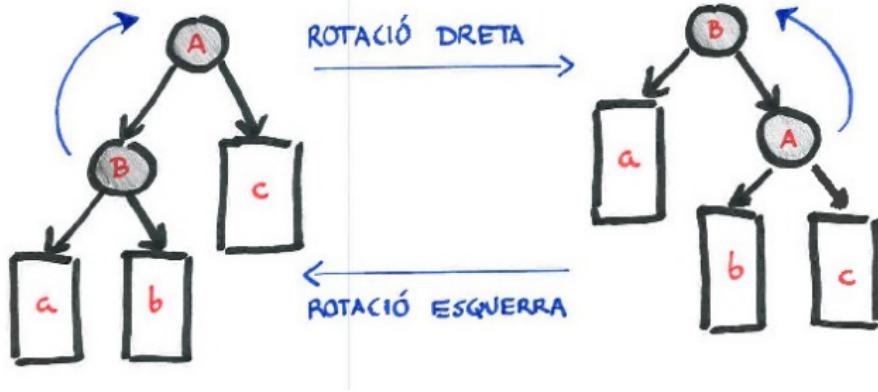
## Retornar i actualitzar l'alçada

```
static int alcada (Node* p) {  
    return p ? p->alc : -1;  
}
```

```
static void actualitzar_alcada (Node* p) {  
    p->alc = 1 + max(alcada(p->fesq), alcada(p->fdre));  
}
```

# Implementació (AVL)

Per tal de mantenir equilibrat un arbre després d'una assignació o esborrat, considerem dues operacions sobre arbres anomenades **girs** o **rotacions**:



Tenen cost  $\Theta(1)$ , ja que com veurem a continuació només requereixen assignacions de punters i enters.

# Implementació (AVL)

## Assignar

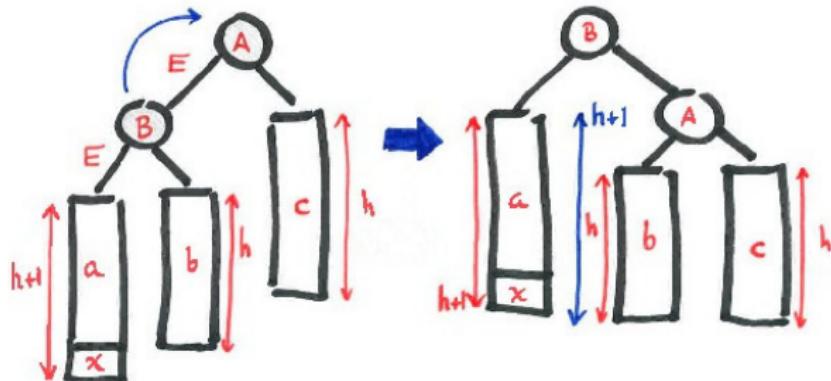
```
void assignar(Node*& p, const Clau& clau, const Info& info) {
    if (p) {
        if (clau < p->clau) {
            assignar(p->fesq, clau, info);
            if (alcada(p->fesq) - alcada(p->fdre) == 2) {
                if (clau < p->fesq->clau) LL(p);
                else LR(p); }
            actualitzar_alcada(p);
        } else if (clau > p->clau) {
            assignar(p->fdre, clau, info);
            if (alcada(p->fdre) - alcada(p->fesq) == 2) {
                if (clau > p->fdre->clau) RR(p);
                else RL(p); }
            actualitzar_alcada(p);
        } else p->info = info;
    } else {
        p = new Node(clau, info, nullptr, nullptr, 0);
        ++n; } }
```

```
void assignar(Node*& p, const Clau& clau, const Info& info) {
    if (p) {
        if (clau < p->clau) {
            assignar(p->fesq, clau, info);
            if (alcada(p->fesq) - alcada(p->fdre) == 2) {
                if (clau < p->fesq->clau) LL(p);
                else LR(p);
            }
            actualitzar_alcada(p);
        } else if (clau > p->clau) {
            ...
        }
    }
}
```

- Si  $\text{alcada}(\text{p}->\text{fesq}) - \text{alcada}(\text{p}->\text{fdre}) == 2$  llavors l'arbre no compleix la condició d'equilibri: el subarbre esquerre és massa alt
- Si  $\text{clau} < \text{p}->\text{fesq}->\text{clau}$  llavors la inserció s'ha fet al subarbre esquerre del subarbre esquerre: cas esquerra-esquerra (LL, en anglès)

# Implementació (AVL)

Fem una rotació dreta així:



## Cas LL

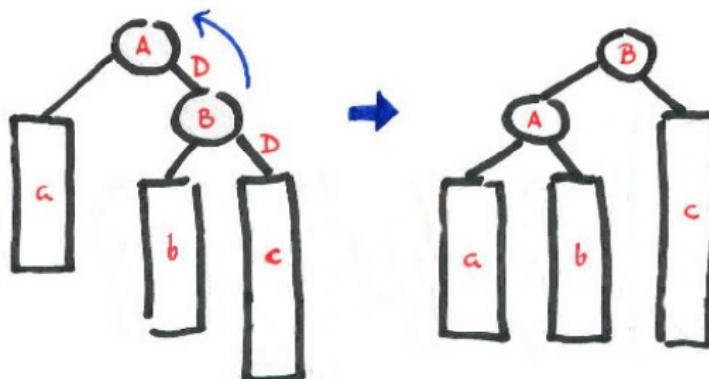
```
static void LL (Node*& p) {  
    Node* q = p;  
    p = p->fesq;  
    q->fesq = p->fdre;  
    p->fdre = q;  
    actualitzar_alcada(q);  
    actualitzar_alcada(p); }
```

```
...
} else if (clau > p->clau) {
    assignar(p->fdre, clau, info);
    if (alcada(p->fdre) - alcada(p->fesq) == 2) {
        if (clau > p->fdre->clau) RR(p);
        else RL(p);
    }
    actualitzar_alcada(p);
} else ...
}
```

- Si  $\text{alcada}(\text{p}-\text{>} \text{fdre}) - \text{alcada}(\text{p}-\text{>} \text{fesq}) == 2$  llavors l'arbre no compleix la condició d'equilibri: el subarbre dret és massa alt
- Si  $\text{clau} > \text{p}-\text{>} \text{fdre}-\text{>} \text{clau}$  llavors la inserció s'ha fet al subarbre dret del subarbre dret: cas dreta-dreta (RR, en anglès)

# Implementació (AVL)

Fem una rotació esquerra així:



## Cas RR

```
static void RR (Node*& p) {  
    Node* q = p;  
    p = p->fdre;  
    q->fdre = p->fesq;  
    p->fesq = q;  
    actualitzar_alcada(q);  
    actualitzar_alcada(p); }
```

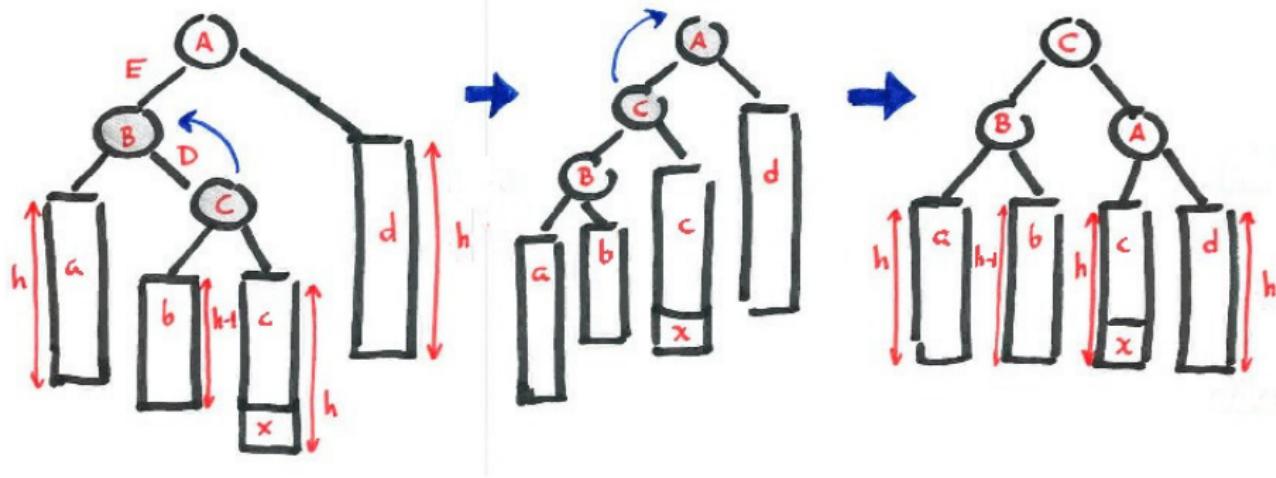
# Implementació (AVL)

```
void assignar(Node*& p, const Clau& clau, const Info& info) {
    if (p) {
        if (clau < p->clau) {
            assignar(p->fesq, clau, info);
            if (alcada(p->fesq) - alcada(p->fdre) == 2) {
                if (clau < p->fesq->clau) LL(p);
                else LR(p);
            }
            actualitzar_alcada(p);
        } else if (clau > p->clau) {
            ...
        }
    }
}
```

- Si  $\text{alcada}(\text{p}->\text{fesq}) - \text{alcada}(\text{p}->\text{fdre}) == 2$  llavors l'arbre no compleix la condició d'equilibri: el subarbre esquerre és massa alt
- Si  $\text{clau} > \text{p}->\text{fesq}->\text{clau}$  llavors la inserció s'ha fet al subarbre dret del subarbre esquerre: cas esquerra-dreta (LR, en anglès)

# Implementació (AVL)

Fem una rotació esquerra seguida d'una rotació dreta així:



## Cas LR

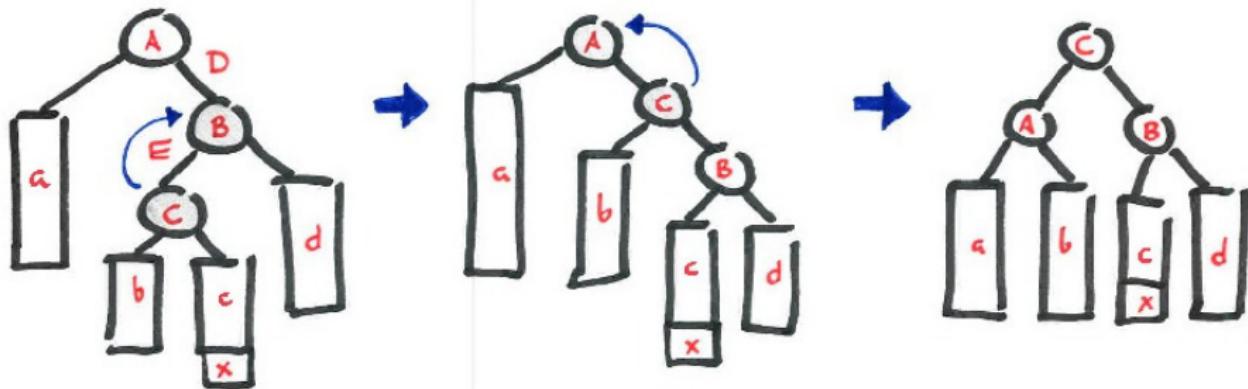
```
static void LR (Node*& p) {  
    RR (p->fesq);  
    LL (p);  
}
```

```
...
} else if (clau > p->clau) {
    assignar(p->fdre, clau, info);
    if (alcada(p->fdre) - alcada(p->fesq) == 2) {
        if (clau > p->fdre->clau) RR(p);
        else RL(p);
    }
    actualitzar_alcada(p);
} else ...
}
```

- Si  $\text{alcada}(p\rightarrow\text{fdre}) - \text{alcada}(p\rightarrow\text{fesq}) == 2$  llavors l'arbre no compleix la condició d'equilibri: el subarbre dret és massa alt
- Si  $\text{clau} < p\rightarrow\text{fdre}\rightarrow\text{clau}$  llavors la inserció s'ha fet al subarbre esquerre del subarbre dret: cas dreta-esquerra (RL, en anglès)

# Implementació (AVL)

Fem una rotació dreta seguida d'una rotació esquerra així:



## Cas RL

```
static void RL (Node*& p) {  
    LL (p->fdre);  
    RR (p);  
}
```

# Implementació (AVL)

## Esborrar

```
void esborrar(Node*& p, const Clau& clau) {
    if (p) {
        if (clau < p->clau) {
            esborrar(p->fesq, clau);
            equilibrar_esq(p);
        } else if (clau > p->clau) {
            esborrar(p->fdre, clau);
            equilibrar_dre(p);
        } else {
            Node* old = p;
            if (p->alcada == 0) p = 0;
            else if (!p->fesq) p = p->fdre;
            else if (!p->fdre) p = p->fesq;
            else {
                Node* q = esborrar_minim(p->fdre);
                q->fesq = p->fesq; q->fdre = p->fdre; p = q;
                equilibrar_dre(p);
            }
            delete old; --n; } } }
```

## Esborrar, cont.

```
Node* esborrar_minim (Node*& p) {  
    if (p->fesq) {  
        Node* q = esborrar_minim(p->fesq);  
        equilibrar_esq(p);  
        return q;  
    } else {  
        Node* q = p;  
        p = p->fdre;  
        return q;  
    } }
```

# Implementació (AVL)

## Esborrar, cont.

```
void equilibrar_esq (Node*& p) {  
    if (alcada(p->fdre)-alcada(p->fesq)==2) {  
        if (alcada(p->fdre->fesq) - alcada(p->fdre->fdre) == 1) {  
            RL(p);  
        } else {  
            RR(p);  
        }  
    } else actualitzar_alcada(p); }  
  
void equilibrar_dre (Node*& p) {  
    if (alcada(p->fesq)-alcada(p->fdre)==2) {  
        if (alcada(p->fesq->fdre) - alcada(p->fesq->fesq) == 1) {  
            LR(p);  
        } else {  
            LL(p);  
        }  
    } else actualitzar_alcada(p); }
```

Les operacions **assignar**, **esborrar** fan servir les rotacions i tenen cost, en el cas pitjor,  $\Theta(\log n)$ .

L'operació **consulta** té cost en el cas pitjor  $\Theta(\log n)$ .