

# Tema 3. Diccionaris

Estructures de Dades i Algorismes

FIB

Antoni Lozano

Q1 2019–2020

Versió de 16 d'octubre de 2019

# Tema 3. Diccionaris

## 1 TAD Diccionari

## 2 Taules de dispersió

- Taules d'accés directe
- Taules de dispersió
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)
- Alçària d'un AVL

# Tema 3. Diccionaris

## 1 TAD Diccionari

### 2 Taules de dispersió

- Taules d'accés directe
- Taules de dispersió
- Col·lisions
- Encadenament
- Funcions de dispersió

### 3 Arbres

- Definicions i terminologia

### 4 Arbres binaris de cerca

- Implementació (ABC)

### 5 Arbres AVL

- Implementació (AVL)
- Alçària d'un AVL

## Diccionari

Anomenem **diccionari** (també taula de símbols, *associative array* o *map*) a un conjunt d'elements que tenen un identificador únic i que té les operacions:

- **assignar**: incloure un element nou
- **esborrar**: eliminar un element
- **consultar**: comprovar si un element hi és

L'identificador únic dels elements s'anomena **clau**:

element = (clau, informació)

Es fan servir diccionaris per implementar **taules de símbols** en compiladors i **taules de memòria** en sistemes operatius.

## Diccionari

Anomenem **diccionari** (també taula de símbols, *associative array* o *map*) a un conjunt d'elements que tenen un identificador únic i que té les operacions:

- **assignar**: incloure un element nou
- **esborrar**: eliminar un element
- **consultar**: comprovar si un element hi és

L'identificador únic dels elements s'anomena **clau**:

element = (clau, informació)

Es fan servir diccionaris per implementar **taules de símbols** en compiladors i **taules de memòria** en sistemes operatius.

## Diccionari

Anomenem **diccionari** (també taula de símbols, *associative array* o *map*) a un conjunt d'elements que tenen un identificador únic i que té les operacions:

- **assignar**: incloure un element nou
- **esborrar**: eliminar un element
- **consultar**: comprovar si un element hi és

L'identificador únic dels elements s'anomena **clau**:

element = (clau, informació)

Es fan servir diccionaris per implementar **taules de símbols** en compiladors i **taules de memòria** en sistemes operatius.

## Exemple 1

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

- clau → número de soci
- informació → nom, adreça, telèfon, correu electrònic

## Exemple 2

Un **banc** vol organitzar la informació del seu conjunt de comptes corrents.

- clau → número de compte
- informació → propietari/s, tipus de compte, saldo, moviments...

## Exemple 1

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

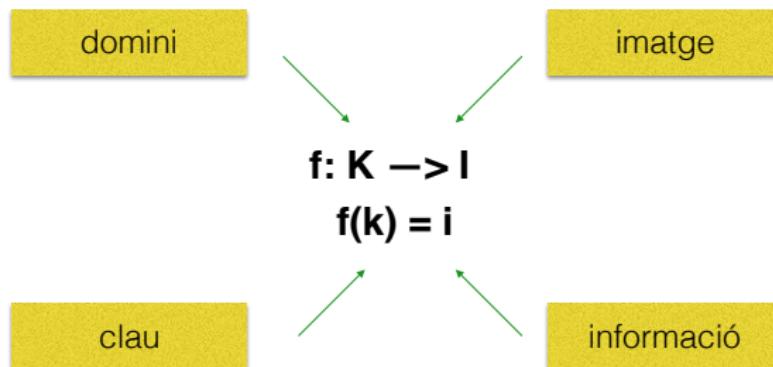
- clau → número de soci
- informació → nom, adreça, telèfon, correu electrònic

## Exemple 2

Un **banc** vol organitzar la informació del seu conjunt de comptes corrents.

- clau → número de compte
- informació → propietari/s, tipus de compte, saldo, moviments...

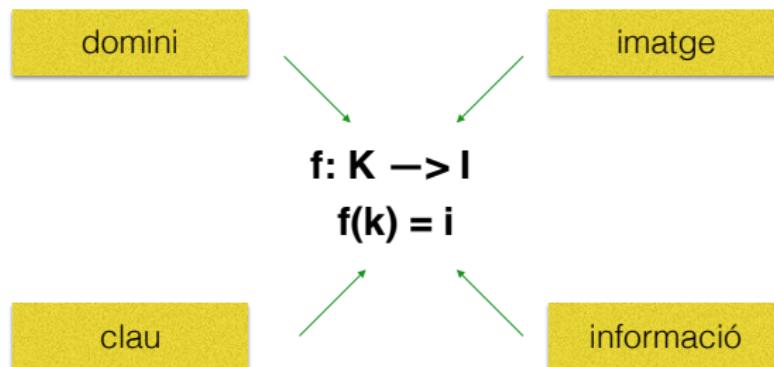
Es pot pensar en els diccionaris com generalitzacions dels **vectors** o com **funcions**:



Suposarem que:

- les funcions són **bjectives**
- les claus formen un **conjunt totalment ordenat**

Es pot pensar en els diccionaris com generalitzacions dels **vectors** o com **funcions**:



Suposarem que:

- les funcions són **bijjectives**
- les claus formen un **conjunt totalmente ordenat**

## Conjunt totalment ordenat

Un conjunt  $K$  està **totalment ordenat** per una operació  $\leq$  si per a tot  $a, b, c \in K$ :

- si  $a \leq b$  i  $b \leq a$ , llavors  $a = b$  (**antisimetria**)
- si  $a \leq b$  i  $b \leq c$ , llavors  $a \leq c$  (**transitivitat**)
- $a \leq b$  o  $b \leq a$  (**totalitat o comparabilitat**)

## Qüestió

Comproveu que el conjunt dels naturals  $\mathbb{N}$  està totalment ordenat per la relació de més petit o igual  $\leq$  però no per la de divisibilitat  $|$ .

## Conjunt totalment ordenat

Un conjunt  $K$  està **totalment ordenat** per una operació  $\leq$  si per a tot  $a, b, c \in K$ :

- si  $a \leq b$  i  $b \leq a$ , llavors  $a = b$  (**antisimetria**)
- si  $a \leq b$  i  $b \leq c$ , llavors  $a \leq c$  (**transitivitat**)
- $a \leq b$  o  $b \leq a$  (**totalitat o comparabilitat**)

## Qüestió

Comproveu que el conjunt dels naturals  $\mathbb{N}$  està totalment ordenat per la relació de més petit o igual  $\leq$  però no per la de divisibilitat  $|$ .

Com a conjunt de **claus** considerarem el conjunt  $\mathbb{N}$  ordenat per la relació  $\leq$ .

Una subfamília important dels diccionaris és la dels anomenats **diccionaris ordenats** en què és possible fer un recorregut ordenat dels elements per ordre creixent de les seves claus.

En C++ s'aconsegueix mitjançant iteradors.

Exemple: escriure equivalències en anglès de paraules en català

L'iterador `it` apunta a un parell `<clau, valor>`, on la clau és una paraula en català i el valor, la traducció a l'anglès.

```
map<string, string> CatEng;  
...  
for (auto it : CatEng)  
    cout << it -> first << " = " << it -> second << endl;
```

Com a conjunt de **claus** considerarem el conjunt  $\mathbb{N}$  ordenat per la relació  $\leq$ .

Una subfamília important dels diccionaris és la dels anomenats **diccionaris ordenats** en què és possible fer un recorregut ordenat dels elements per ordre creixent de les seves claus.

En C++ s'aconsegueix mitjançant iteradors.

Exemple: escriure equivalències en anglès de paraules en català

L'iterador `it` apunta a un parell `<clau, valor>`, on la clau és una paraula en català i el valor, la traducció a l'anglès.

```
map<string, string> CatEng;  
...  
for (auto it : CatEng)  
    cout << it -> first << " = " << it -> second << endl;
```

## Operacions

- **assignar**: afegir un element (clau, informació) al diccionari;  
si existia un element amb la mateixa clau, se sobreescrivia 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
- **consultar**: donada una clau, retorna una referència a la informació  
associada a la clau
- **talla**: retorna la talla del diccionari

## Variants de consultar

Considerarem variants de l'operació

- **consultar**: donada una clau, retorna una referència a la informació associada a la clau

com ara

- **present**: donada una clau, retorna un booleà que indica si hi ha un element amb aquella clau
- **cerca**: donada una clau, retorna una referència a l'element amb aquella clau

En general, però, afegint operacions més complexes s'obtenen noves estructures de dades. Per exemple, afegint l'operació d'**extreure el mínim** dona lloc a les **cues amb prioritat**.

## Variants de consultar

Considerarem variants de l'operació

- **consultar**: donada una clau, retorna una referència a la informació associada a la clau

com ara

- **present**: donada una clau, retorna un booleà que indica si hi ha un element amb aquella clau
- **cerca**: donada una clau, retorna una referència a l'element amb aquella clau

En general, però, afegint operacions més complexes s'obtenen noves estructures de dades. Per exemple, afegint l'operació d'**extreure el mínim** dona lloc a les **cues amb prioritat**.

## 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  $k, k_1, k_2 \in K, i, j \in I, k_1 \neq k_2$

esborrar ( $k$ , crear) = crear

esborrar ( $k$ , assignar ( $k, i, D$ )) = esborrar ( $k, D$ )

esborrar ( $k_1$ , assignar ( $k_2, i, D$ )) = assignar ( $k_2, i$ , esborrar ( $k_1, D$ ))

assignar ( $k, i$ , assignar ( $k, j, D$ )) = assignar ( $k, i, D$ )

assignar ( $k_1, i$ , assignar ( $k_2, j, D$ )) = assignar ( $k_2, j$ , assignar ( $k_1, i, D$ ))

consultar ( $k$ , crear) =  $\perp$

consultar ( $k$ , esborrar ( $k, D$ )) =  $\perp$

consultar ( $k$ , assignar ( $k, i, D$ )) =  $i$

consultar ( $k_1$ , assignar ( $k_2, i, D$ )) = consultar ( $k_1, D$ )

## Nombre d'elements consultats en les operacions

cas pitjor/mitjà	assignar	esborrar	consultar
vector no ordenat	$n, n/2$	$n, n/2$	$n, n/2$
vector ordenat	$n, n/2$	$n, n/2$	$\log n, \log n$
llista no ordenada	$n, n$	$n, n/2$	$n, n/2$
llista ordenada	$n, n/2$	$n, n/2$	$n, n/2$
taula de dispersió	$n, 1$	$n, 1$	$n, 1$
AVL	$\log n, \log n$	$\log n, \log n$	$\log n, \log n$

En vectors, cal desplaçar els elements.

En estructures no ordenades, cal comprovar repetits.

# Tema 3. Diccionaris

## 1 TAD Diccionari

### 2 Taules de dispersió

- Taules d'accés directe
- Taules de dispersió
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

## 4 Arbres binaris de cerca

- Implementació (ABC)

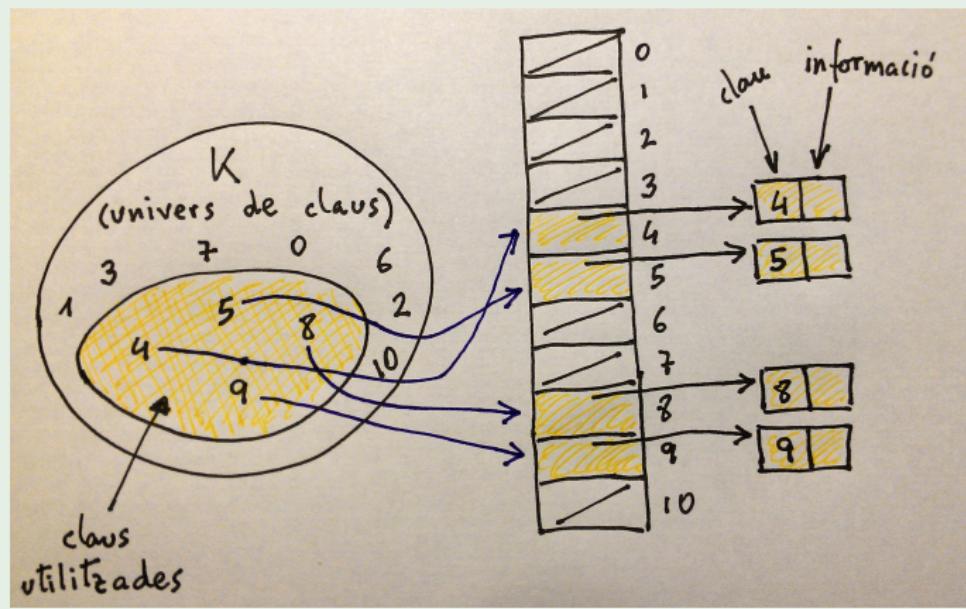
## 5 Arbres AVL

- Implementació (AVL)
- Alçària d'un AVL

# 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 correspon a una clau
- Les operacions seran  $\Theta(1)$  en cas pitjor
- Es pot guardar la informació directament en les posicions de la taula corresponents a la seva clau, però cal indicar si l'espai és buit

## Exercici

Volem implementar un diccionari fent servir adreçament directe en un vector *enorme*. Cal tenir en compte que

- inicialment, les entrades poden contenir deixalles
- no és pràctic inicialitzar el vector a causa de la seva talla

Descriu un mètode per aconseguir implementar les operacions **consultar**, **assignar** i **esborrar** en temps  $\Theta(1)$ .

# Taules d'accés directe

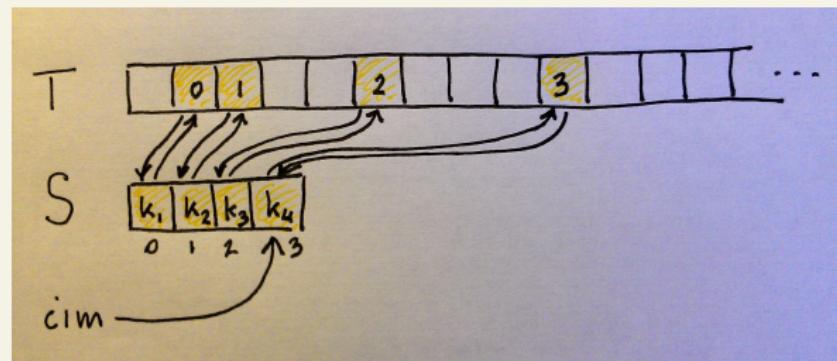
Solució: idea

Definim un vector enorme  $T$  accessible per clau i un vector  $S$  amb tantes entrades com claus utilitzades ( $cim + 1$ ) tal que per a una clau  $k$ :

- $T[k]$  conté l'índex  $j$  d'una entrada vàlida de  $S$
- $S[j]$  conté  $k$

És a dir,  $S[T[k]] = k$  i  $T[S[j]] = j$  (en diem **cicle de validació**).

Definim també un vector  $S'$  que conté els objectes (la informació). Quan la clau  $k$  defineix un cicle de validació,  $S'[T[k]]$  conté l'objecte.



## Solució: operacions

- **inicialitzar**

```
cim = -1;
```

- **consultar.** Donada una clau  $k$ ,

```
if (S[T[k]] == k)
    return S'[T[k]];
else
    return NULL;
```

## Solució: operacions

- **assignar.** Donat un objecte  $x$  amb clau  $k$ , si la clau no hi és,

```
++cim;  
S[cim] = k;  
S'[cim] = x;  
T[k] = cim;
```

- **esborrar.** Donada una clau  $k$  (suposant que la clau hi és), hem d'assegurar que no queda un "forat" a  $S$ .

```
S[T[k]] = S[cim];  
S'[T[k]] = S'[cim];  
T[S[T[k]]] = T[k];  
T[k] = 0;  
--cim;
```

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.

Quan

- el nombre de claus utilitzades és petit en relació al 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.

## Qüestió

Per què hauríem de fer servir els dos últims dígits i no els dos primers?

# Taules de dispersió

La biblioteca expedeix carnets des del 1825 amb números consecutius.

## Qüestió

Per què hauríem de fer servir els dos últims dígits i no els dos primers?

## Pista

La biblioteca podria rebre la visita d'un grup d'amics amb números de carnet semblants, com ara

- 001523
- 001525
- 001531
- 001570

Però no és tan probable que en algun moment molts usuaris tinguin els dos últims dígits idèntics.

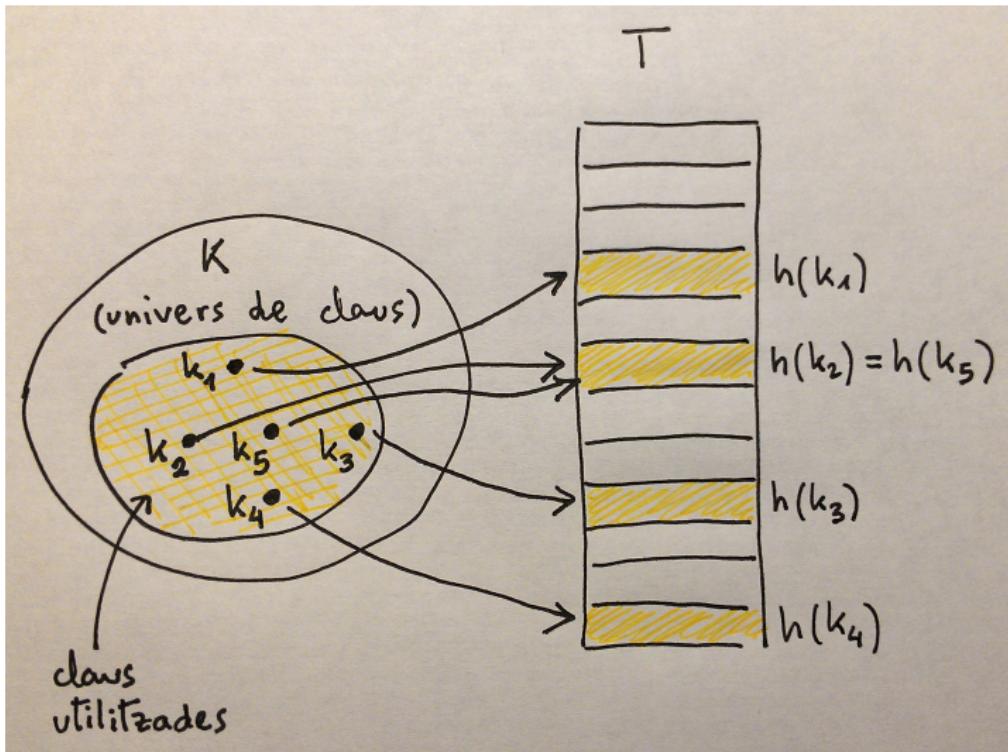
Suposem que volem emmagatzemar un màxim de  $n$  claus. Declarem una taula  $T$  de  $m \leq n$  posicions.

- Amb **adreçament directe**,  $m = n$  i l'element de clau  $k$  va a l'espai  $T[k]$
- Amb **taules de dispersió**,  $m < n$  i l'element va a l'espai  $T[h(k)]$ , on

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

és la **funció de dispersió** (*hash function*) i  $h(k)$ , el **valor de dispersió** de  $k$

# Taules de dispersió



La situació en què dues claus diferents tenen el mateix valor de dispersió i, per tant, coincideixen en el mateix espai (com  $k_2$  i  $k_5$ ) se'n diu **col·lisió**.

Les col·lisions són **molt probables**.

Suposem que les nostres claus són els dies de l'any. Quanta gent ha d'haver-hi en una habitació perquè la probabilitat que almenys dues tinguin l'aniversari el mateix dia sigui del

- 1 100%?
- 2 50%?
- 3 99.9%?

Les col·lisions són **molt probables**.

Suposem que les nostres claus són els dies de l'any. Quanta gent ha d'haver-hi en una habitació perquè la probabilitat que almenys dues tinguin l'aniversari el mateix dia sigui del

- 1 100%? **367**
- 2 50%?
- 3 99.9%?

Les col·lisions són **molt probables**.

Suposem que les nostres claus són els dies de l'any. Quanta gent ha d'haver-hi en una habitació perquè la probabilitat que almenys dues tinguin l'aniversari el mateix dia sigui del

- 1 100%? **367**
- 2 50%? **23**
- 3 99.9%?

Les col·lisions són **molt probables**.

Suposem que les nostres claus són els dies de l'any. Quanta gent ha d'haver-hi en una habitació perquè la probabilitat que almenys dues tinguin l'aniversari el mateix dia sigui del

- 1 100%? **367**
- 2 50%? **23**
- 3 99.9%? **70**

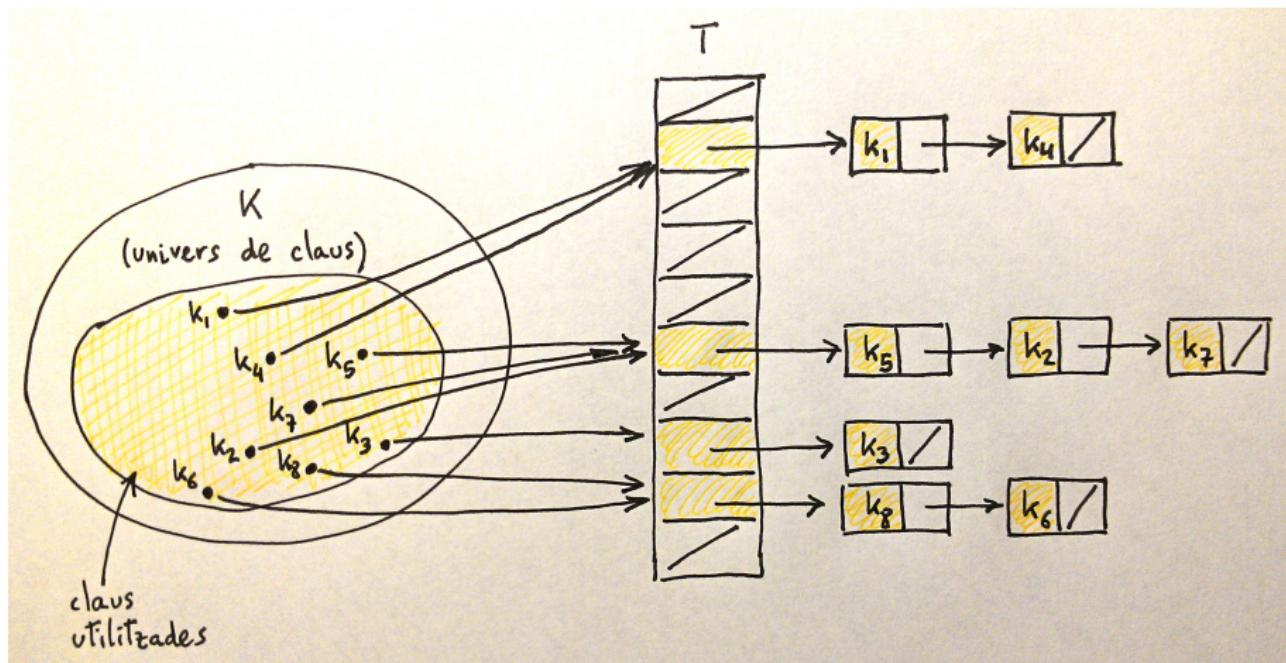
L'ideal seria evitar les col·lisions triant una bona funció de dispersió.

Algunes consideracions sobre la funció de dispersió:

- Ha de ser determinista però ha de semblar aleatòria
- 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**.

# Col·lisions

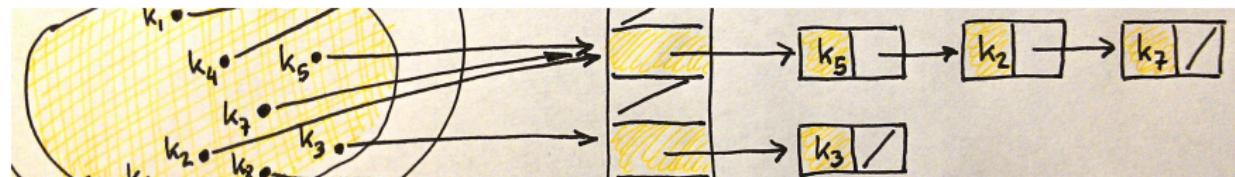


Resolució de col·lisions per **encadenament**. Cada entrada  $T[j]$  conté una **llista encadenada** amb les claus amb valor de dispersió  $j$ .

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** (o **encadenament separat**), 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 el **cas pitjor**: quan tots els elements tenen el mateix valor de dispersió i formen una sola llista
- $\Theta(1)$  en el **cas mitjà**

# Encadenament

Classe Dictionary: implementació (*Algorithms in C++, EDA*)

```
template <typename Key, typename Info>
class Dictionary {

private:

typedef pair<Key, Info> Pair;
typedef list<Pair> List;
typedef typename List::iterator iter;

vector<List> t; // Taula de dispersio
int n;           // Nombre de claus
int M;           // Nombre de posicions
```

## Classe Dictionary: implementació

```
public:  
  
Dictionary (int M = 1009)  
: t(M), n(0), M(M) {}  
  
void assign (const Key& key, const Info& info) {  
    int h = hash(key) % M;  
    iter p = find(key, t[h]);  
    if (p != t[h].end())  
        p->second = info;  
    else {  
        t[h].push_back(Pair(key, info));  
        ++n;  
    }  
}
```

# Encadenament

## Classe Dictionary: implementació

```
void erase (const Key& key) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    if (p != t[h].end()) {
        t[h].erase(p);
        --n;
    }
}
```

```
Info& query (const Key& key) {
    int h = hash(key) % M;
    iter p = find(key, t[h]);
    if (p != t[h].end())
        return p->second;
    else
        throw "Key does not exist";
}
```

## Classe Dictionary: implementació

```
bool contains (const Key& key) {  
    int h = hash(key) % M;  
    iter p = find(key, t[h]);  
    return p != t[h].end();  
}  
  
int size () {  
    return n;  
}
```

El cas pitjor de les operacions assign, erase, query i contains és  $\Theta(n)$ .  
El cost mitjà és  $\Theta(1 + n/M)$ .

## Classe Dictionary: implementació

```
bool contains (const Key& key) {  
    int h = hash(key) % M;  
    iter p = find(key, t[h]);  
    return p != t[h].end();  
}  
  
int size () {  
    return n;  
}
```

El cas pitjor de les operacions assign, erase, query i contains és  $\Theta(n)$ .  
El cost mitjà és  $\Theta(1 + n/M)$ .

## Classe Dictionary: implementació

Els costos previs depenen del cost de fer una cerca, que és  $\Theta(n)$  en cas pitjor i  $\Theta(1 + n/M)$  en mitjana.

**private:**

```
static iter find (const Key& key, list<Pair>& L) {
    iter p = L.begin();
    while (p != L.end() and p->first != key)
        ++p;
    return p;
}
```

## Exercici

Si  $K$  és el conjunt de totes les claus i  $m$  és el nombre de posicions de la taula de dispersió, quantes claus podem assegurar que col·lidiran a un mateix valor de dispersió 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, quantes claus podem assegurar que col·lidiran a un mateix valor de dispersió si

- $|K| > m?$  **2**
- $|K| > 2m?$
- $|K| > 3m?$
- $|K| > nm?$

## Solució

Si  $K$  és el conjunt de totes les claus i  $m$  és el nombre de posicions de la taula de dispersió, quantes claus podem assegurar que col·lidiran a un mateix valor de dispersió si

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

## Solució

Si  $K$  és el conjunt de totes les claus i  $m$  és el nombre de posicions de la taula de dispersió, quantes claus podem assegurar que col·lidiran a un mateix valor de dispersió si

- $|K| > m?$  2
- $|K| > 2m?$  3
- $|K| > 3m?$  4
- $|K| > nm?$

## Solució

Si  $K$  és el conjunt de totes les claus i  $m$  és el nombre de posicions de la taula de dispersió, quantes claus podem assegurar que col·lidiran a un mateix valor de dispersió si

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

## Proposició

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

## Corol·lari

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

# Encadenament: anàlisi del cas mitjà

## Definició

Si  $T$  és una taula amb  $m$  posicions que conté  $n$  elements, el **factor de càrrega** per a  $T$  és  $\alpha = n/m$  (nombre mitjà d'elements per posició).

## Proposició

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

## Demostració

Suposem que cerquem una clau  $k$  a  $T$ .

El cost esperat és  $\Theta(1 + E[X])$ , on  $X$  és la mida de  $T[h(k)]$ .

Definim la variable booleana  $X_e = [h(e) = h(k)]$ . Llavors,

$$E[X] = E\left[\sum_e X_e\right] = \sum_e E[X_e] = \sum_e \text{prob}(X_e = 1) = n \cdot (1/m).$$

Per tant,  $\Theta(1 + E[X]) = \Theta(1 + n/m) = \Theta(1 + \alpha)$ .

# Encadenament: anàlisi del cas mitjà

## Definició

Si  $T$  és una taula amb  $m$  posicions que conté  $n$  elements, el **factor de càrrega** per a  $T$  és  $\alpha = n/m$  (nombre mitjà d'elements per posició).

## Proposició

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

## Demostració

Suposem que cerquem una clau  $k$  a  $T$ .

El cost esperat és  $\Theta(1 + E[X])$ , on  $X$  és la mida de  $T[h(k)]$ .

Definim la variable booleana  $X_e = [h(e) = h(k)]$ . Llavors,

$$E[X] = E\left[\sum_e X_e\right] = \sum_e E[X_e] = \sum_e \text{prob}(X_e = 1) = n \cdot (1/m).$$

Per tant,  $\Theta(1 + E[X]) = \Theta(1 + n/m) = \Theta(1 + \alpha)$ .

# Encadenament: anàlisi del cas mitjà

## Definició

Si  $T$  és una taula amb  $m$  posicions que conté  $n$  elements, el **factor de càrrega** per a  $T$  és  $\alpha = n/m$  (nombre mitjà d'elements per posició).

## Proposició

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

## Demostració

Suposem que cerquem una clau  $k$  a  $T$ .

El cost esperat és  $\Theta(1 + E[X])$ , on  $X$  és la mida de  $T[h(k)]$ .

Definim la variable booleana  $X_e = [h(e) = h(k)]$ . Llavors,

$$E[X] = E\left[\sum_e X_e\right] = \sum_e E[X_e] = \sum_e \text{prob}(X_e = 1) = n \cdot (1/m).$$

Per tant,  $\Theta(1 + E[X]) = \Theta(1 + n/m) = \Theta(1 + \alpha)$ .

# Encadenament: anàlisi del cas mitjà

## Definició

Si  $T$  és una taula amb  $m$  posicions que conté  $n$  elements, el **factor de càrrega** per a  $T$  és  $\alpha = n/m$  (nombre mitjà d'elements per posició).

## Proposició

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

## Demostració

Suposem que cerquem una clau  $k$  a  $T$ .

El cost esperat és  $\Theta(1 + E[X])$ , on  $X$  és la mida de  $T[h(k)]$ .

Definim la variable booleana  $X_e = [h(e) = h(k)]$ . Llavors,

$$E[X] = E\left[\sum_e X_e\right] = \sum_e E[X_e] = \sum_e \text{prob}(X_e = 1) = n \cdot (1/m).$$

Per tant,  $\Theta(1 + E[X]) = \Theta(1 + n/m) = \Theta(1 + \alpha)$ .

# Encadenament: anàlisi del cas mitjà

## Definició

Si  $T$  és una taula amb  $m$  posicions que conté  $n$  elements, el **factor de càrrega** per a  $T$  és  $\alpha = n/m$  (nombre mitjà d'elements per posició).

## Proposició

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

## Demostració

Suposem que cerquem una clau  $k$  a  $T$ .

El cost esperat és  $\Theta(1 + E[X])$ , on  $X$  és la mida de  $T[h(k)]$ .

Definim la variable booleana  $X_e = [h(e) = h(k)]$ . Llavors,

$$E[X] = E\left[\sum_e X_e\right] = \sum_e E[X_e] = \sum_e \text{prob}(X_e = 1) = n \cdot (1/m).$$

Per tant,  $\Theta(1 + E[X]) = \Theta(1 + n/m) = \Theta(1 + \alpha)$ .

## Proposició

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

## Corol·lari

El cost en cas mitjà de fer una **assignació**, un **esborrat** o una **consulta** (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, en mitjana, cada llista de sinònims tindrà talla  $\approx 1$  i les operacions, cost  $\Theta(1)$ .

Però com triem la funció de dispersió?

Les claus seran sempre nombres naturals. Si no ho són, s'interpreten com a naturals.

## Exemple

Donada una cadena de caràcters, l'interpretem 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

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

## 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 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

## Exemple 1

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.

Per tant, triem una taula de mida

$$m = 701.$$

Triem 701 perquè és un primer  $\approx 2000/3$  i no és proper a una potència de 2.

La funció de dispersió serà

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

# Funcions de dispersió

## Exemple 2: Factor de càrrega més petit

Volem una taula de dispersió amb encadenament per emmagatzemar  $n = 2000$  cadenes de caràcters. Ens està bé examinar uns 2 elements en una cerca fallida.

Per tant, triem una taula de mida

$$m = 1009.$$

Triem 1009 perquè és un primer  $\approx 2000/2$ .

La funció de dispersió serà

$$h(k) = k \bmod 1009.$$

# Funcions de dispersió

## Exemple 3: Mantenir el factor de càrrega petit

Ara volem emmagatzemar 18000 cadenes de caràcters addicionals. Ens està bé examinar uns 5 elements en una cerca fallida.

Per tant, triem una taula de mida

$$m = 4001.$$

Triem 4001 perquè és un primer  $\approx 20000/5$ .

Fem una redispersió de les cadenes antigues i inserim les noves amb la funció de dispersió

$$h(k) = k \bmod 4001.$$

## El mètode de la multiplicació

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

- ① multipliquem  $k$  per una constant  $A$  t.q.  $0 < A < 1$  i n'extraiem la part fraccional
- ② llavors, multipliquem el valor per  $m$  i prenem la part baixa

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

- **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  $A \approx (\sqrt{5} - 1)/2 = 0,6180339887\dots$

## 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
- no hi ha apuntadors a posicions de la mateixa taula
- la funció de dispersió té dos paràmetres: la clau i la “prova” de posició

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

Per a una clau  $k$ , la seqüència de proves és

$$(h(k, 0), h(k, 1), \dots, h(k, m - 1))$$

# Tema 3. Diccionaris

## 1 TAD Diccionari

## 2 Taules de dispersió

- Taules d'accés directe
- Taules de dispersió
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)
- Alçària d'un AVL

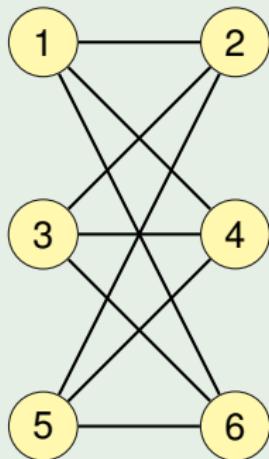
# Definicions i terminologia

## Definició

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

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

Exemple: graf connex i cíclic



- **Connex**: hi ha un camí entre dos vèrtexs qualssevol
- **Cíclic**: hi ha cicles com  $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1$

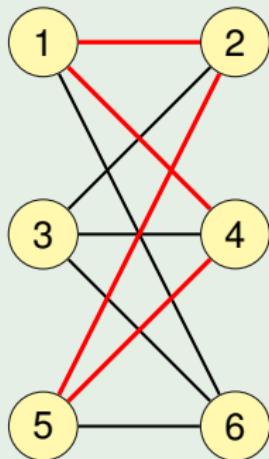
# Definicions i terminologia

## Definició

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

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

Exemple: graf connex i cíclic



- **Connex**: hi ha un camí entre dos vèrtexs qualssevol
- **Cíclic**: hi ha cicles com  $1 \rightarrow 2 \rightarrow 5 \rightarrow 4 \rightarrow 1$

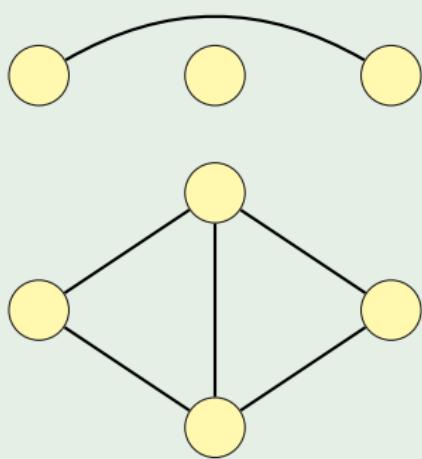
# Definicions i terminologia

## Definició

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

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

Exemple: graf inconnex i cíclic

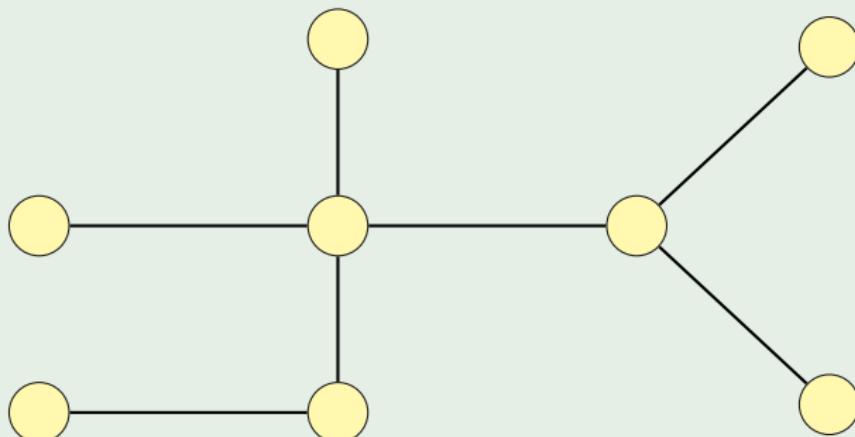


# Definicions i terminologia

## Definició

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

## Exemple: arbre



# Definicions i terminologia

## Teorema

Sigui  $G = (V, E)$  un graf i siguin  $n = |V|$  i  $m = |E|$ .

Les afirmacions següents són equivalents:

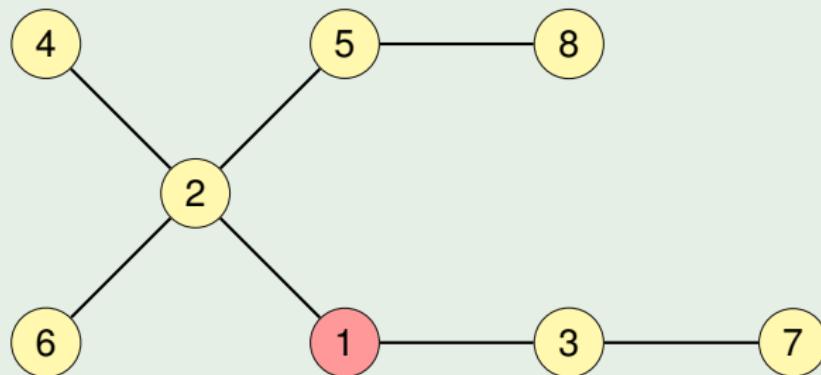
- ①  $G$  és un **arbre**
- ② Tot parell de vèrtexs de  $G$  estan units per un **camí únic**
- ③  $G$  és connex i  $m = n - 1$
- ④  $G$  és connex però, si s'hi elimina una aresta, s'obté un graf inconnex
- ⑤  $G$  és acíclic i  $m = n - 1$
- ⑥  $G$  és acíclic però, si s'hi afegeix una aresta, s'obté un graf cíclic

# Definicions i terminologia

## Definició

Un **arbre arrelat** és un arbre amb un vèrtex distingit.

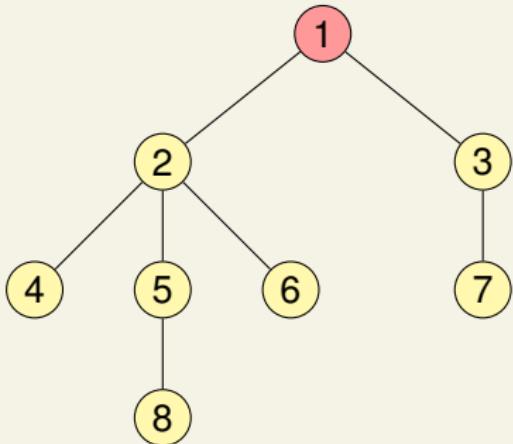
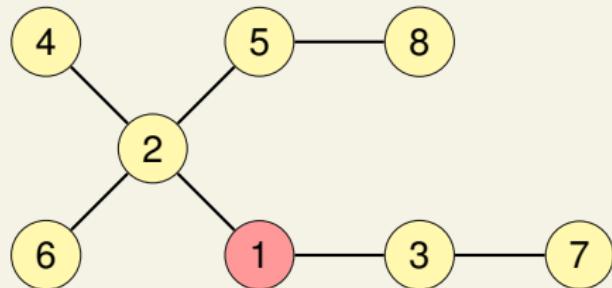
## Exemple: arbre arrelat



# Definicions i terminologia

## Representació

Representarem els arbres arrelats amb el vèrtex distingit a dalt.



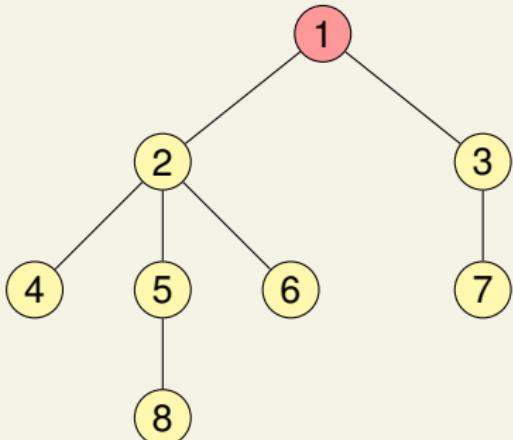
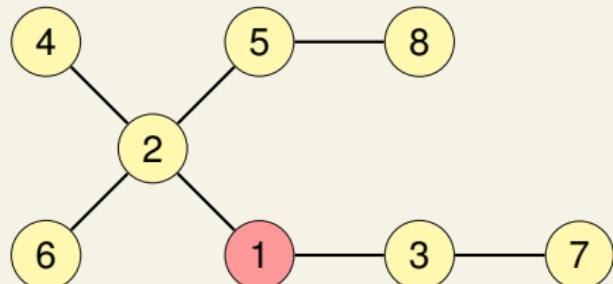
## Terminologia

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

# Definicions i terminologia

## Representació

Representarem els arbres arrelats amb el vèrtex distingit a dalt.



## Terminologia

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

# Definicions i terminologia

## Terminologia pròpia dels arbres

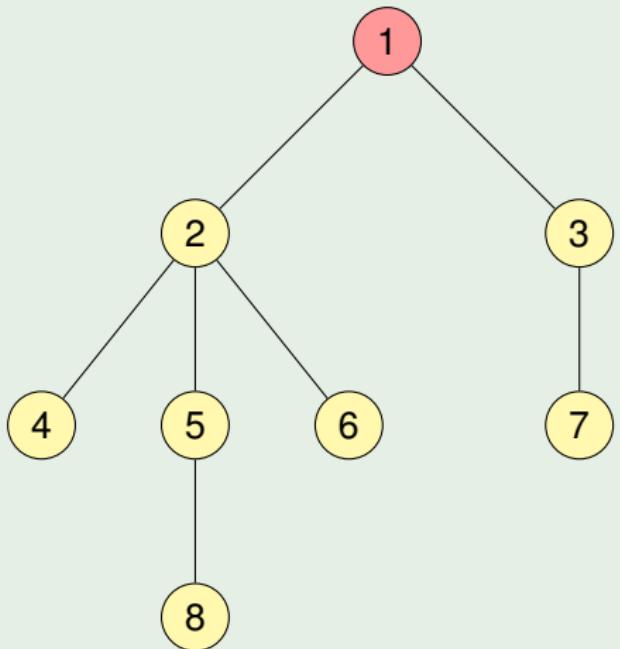
Els vèrtexs s'anomenen **nodes**. El node distingit s'anomena **arrel**.

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  $y$  és un predecessor de  $x$  i existeix l'aresta  $\{y, x\}$  a  $T$ , llavors  $y$  és el **pare** de  $x$  i  $x$  és el **fill** de  $y$
- el pare del pare de  $x$  s'anomena **avi** de  $x$ ; si  $y$  és l'avi de  $x$ , llavors  $x$  és el **nét** 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'**alçària** de  $T$  és la màxima distància (nombre d'arestes d'un camí) entre l'arrel i una fulla

# Definicions i terminologia

## Exemple

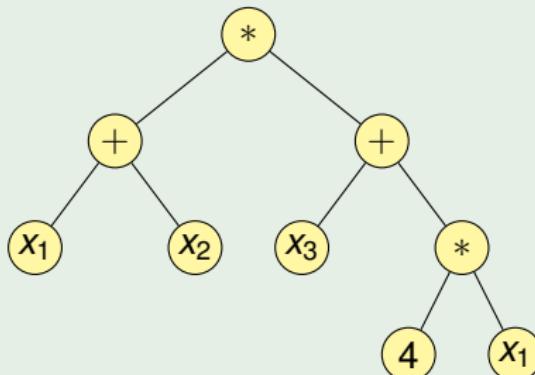


- l'arrel és 1
- 3 és un node intern
- 7 és una fulla
- 2 és pare de 5
- 8 és nét de 2
- els predecessors de 8 són 5, 2 i 1
- els successors de 2 són 4, 5, 6 i 8
- 4, 5, 6 són germans
- l'arbre té alçària 3

# Definicions i terminologia

## Exemple

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. Si les operacions no fossin commutatives, seria important diferenciar entre el fill esquerre i el fill dret.

# Definicions i terminologia

## Definició

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

# Definicions i terminologia

## Definició

Un **arbre binari** és un arbre arrelat 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 formada sobre un conjunt finit de nodes que:

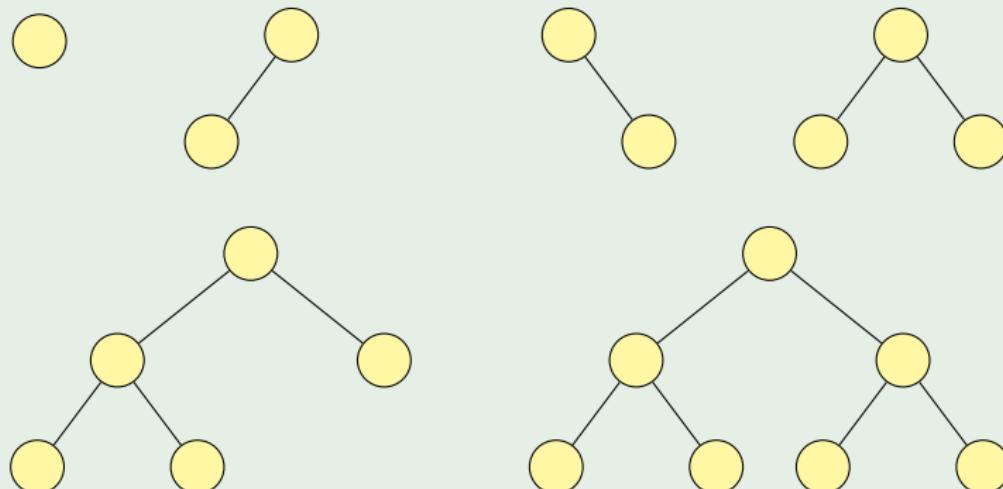
- no conté cap node o
- està formada per tres conjunts de nodes: l'**arrel**, un arbre binari anomenat **subarbre esquerre** i un arbre binari anomenat **subarbre dret**

# Definicions i terminologia

## Definició

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

## Exemple



# Tema 3. Diccionaris

## 1 TAD Diccionari

## 2 Taules de dispersió

- Taules d'accés directe
- Taules de dispersió
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)
- Alçària d'un AVL

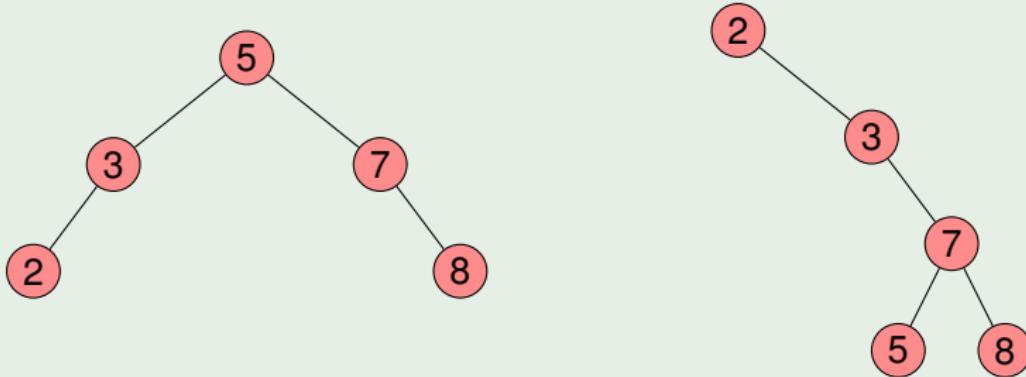
# Introducció

## 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 que la clau de cada node 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



Operacions dels diccionaris en els ABC:

- Les **operacions bàsiques dels diccionaris** (**consultar, assignar, esborrar**) es poden fer en temps proporcional a l'alçària
- L'**alçària esperada** d'un ABC de  $n$  nodes és de  $\Theta(\log n)$
- Per tant, les operacions bàsiques tenen un **cost mitjà** de  $\Theta(\log n)$
- L'**alçària màxima** d'un ABC de  $n$  nodes és de  $\Theta(n)$
- Per tant, els costos de les operacions bàsiques són  $\Theta(n)$  en cas pitjor

La propietat dels ABC permet implementar altres operacions com

- 1 fer la **llista ordenada** dels elements en temps  $\Theta(n)$
- 2 trobar el **mínim** o el **màxim** en temps mitjà  $\Theta(\log n)$
- 3 trobar l'**anterior** o el **següent** d'un element donat en temps mitjà  $\Theta(\log n)$

## Llista ordenada dels elements d'un ABC

- 1 Per fer la llista ordenada dels elements d'un ABC, n'hi ha prou a fer un recorregut inordre de l'arbre:

RECORREGUT-INORDRE( $x$ )

**if**  $x \neq \text{NUL}$  **llavors**

RECORREGUT-INORDRE(esquerre( $x$ ))

**escriure** clau( $x$ )

RECORREGUT-INORDRE(dret( $x$ ))

## Llista ordenada dels elements d'un ABC

- 1 Per fer la llista ordenada dels elements d'un ABC, n'hi ha prou a fer un recorregut inordre de l'arbre:

RECORREGUT-INORDRE( $x$ )

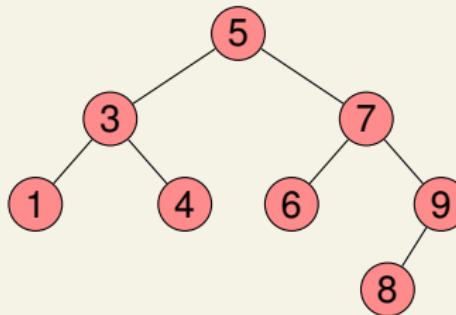
**if**  $x \neq \text{NUL}$  **llavors**

RECORREGUT-INORDRE(esquerre( $x$ ))

**escriure** clau( $x$ )

RECORREGUT-INORDRE(dret( $x$ ))

El recorregut inordre de l'ABC següent és: 1, 3, 4, 5, 6, 7, 8, 9.



## Llista ordenada dels elements d'un ABC

- 1 Per fer la llista ordenada dels elements d'un ABC, n'hi ha prou a fer un recorregut inordre de l'arbre:

RECORREGUT-INORDRE( $x$ )

**if**  $x \neq \text{NUL}$  **llavors**

RECORREGUT-INORDRE(esquerre( $x$ ))

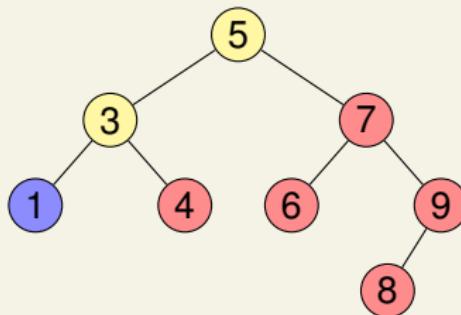
**escriure** clau( $x$ )

RECORREGUT-INORDRE(dret( $x$ ))

Com que cada node es visita un cop, el cost és  $\Theta(n)$ .

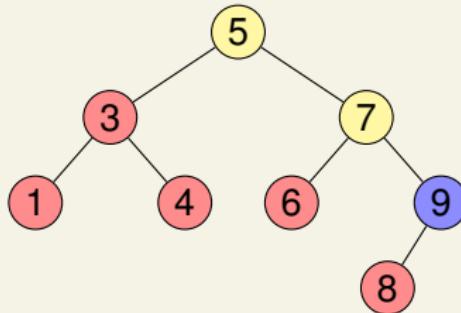
## Trobar el mínim i el màxim d'un ABC

### 2 Mínim



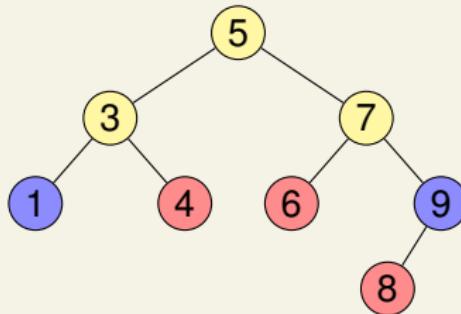
Trobar el mínim i el màxim d'un ABC

## 2 Màxim



## Trobar el mínim i el màxim d'un ABC

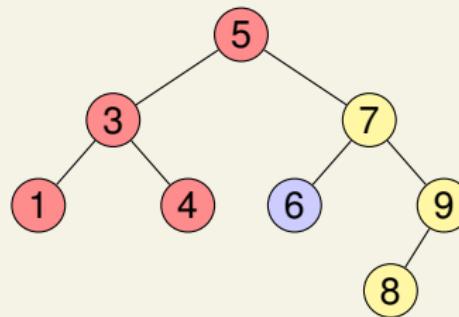
### 2 Mínim o màxim



El cost mitjà correspon a l'alçària esperada de l'arbre:  $\Theta(\log n)$ .

## Trobar l'anterior o el següent d'un element en un ABC

- ③ Següent de 5: el mínim del seu subarbre dret



Llavors, el cost mitjà és  $\Theta(\log n)$ .

## Exercici

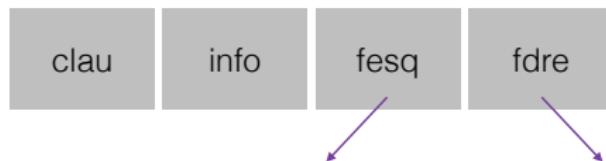
Demostreu que un ABC amb  $n$  nodes es pot reestructurar de manera que tingui alçària  $\lfloor \log n \rfloor$ . Per què una alçària més petita és insuficient?

# Implementació (ABC)

Implementació: *Algoritmes 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 apuntador a l'arrel de l'ABC

Els **costos** que es donen:

- es refereixen al cas pitjor
- depenen de  $n$ , que és el camp del mateix nom del diccionari  
**(nombre d'elements)**

# Implementació (ABC)

## Definició de Diccionari i Node

Un **Node** emmagatzema una clau, la seva informació associada i apuntadors a dos altres nodes.

```
template <typename Clau, typename Info>
class Diccionari {
private:
struct Node {
    Clau clau;
    Info info;
    Node* fesq; // Apuntador al fill esquerre
    Node* fdre; // Apuntador 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 de l'ABC
Node* arrel;   // Apuntador 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);  
}
```

## Constructora d'assignació

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

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

## 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& consultar (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; }
```

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

- $s$  nodes
- alçària  $h$

Els costos en 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 apuntador 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 apuntador 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 descendint pels subarbres adequats fent ús de la propietat dels 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 (recursiu)

Retornar un apuntador 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 (iteratiu)

Retornar un apuntador 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

## Esborrar (1)

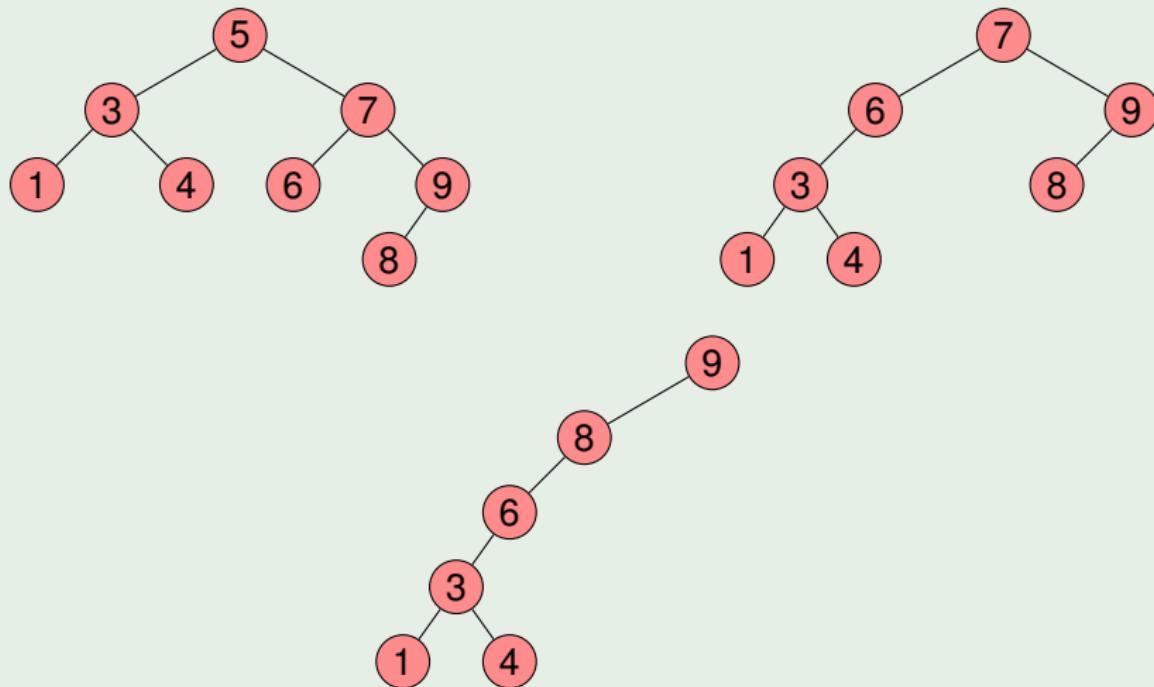
Esborrar el node que conté `clau` en el subarbre apuntat per `p`:  $\Theta(h)$ .  
Es penja el subarbre esquerre del mínim del subarbre dret.

```
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

## Exemple

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



# Implementació (ABC): funcions privades

## Esborrar (2)

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

Es copia el mínim del subarbre dret al node esborrat i després 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 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);
        }
    }
}
```

# Implementació (ABC): funcions privades

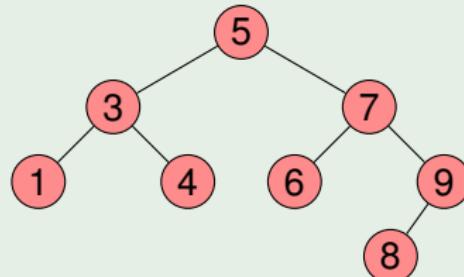
## Esborrar (2)

Inconvenient: es copien claus i informació, més costós que copiar apuntadors.

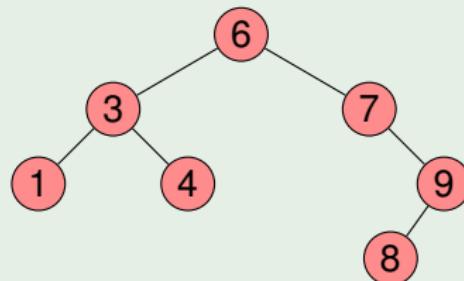
```
void esborrar_2 (Node*& p, const Clau& clau) {
    if (p) {
        if (clau < p->clau) {
            esborrar_2(p->fesq, clau);
        } else if (clau > p->clau) {
            esborrar_2(p->fdre, clau);
        } else if (!p->fesq) {
            Node* q = p; p = p->fdre; delete q; --n;
       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

## Exemple

Donat l'arbre



n'esborrem l'arrel



# Implementació (ABC): funcions privades

## Esborrar (3)

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

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

```
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;  
    } }
```

## Exercici

Canvieu la implementació dels ABCs perquè:

- ① el temps requerit per a les operacions  $\text{minim}$  i  $\text{maxim}$  sigui  $O(1)$  en lloc de  $O(h)$  sense que això afecti al temps de la resta d'operacions
- ② es puguin implementar dues operacions noves  $\text{succ}$  i  $\text{pred}$  (que retornen, respectivament, el successor i el predecessor d'una clau donada) en temps  $O(1)$  sense que això afecti al temps de la resta d'operacions

# Tema 3. Diccionaris

## 1 TAD Diccionari

## 2 Taules de dispersió

- Taules d'accés directe
- Taules de dispersió
- Col·lisions
- Encadenament
- Funcions de dispersió

## 3 Arbres

- Definicions i terminologia

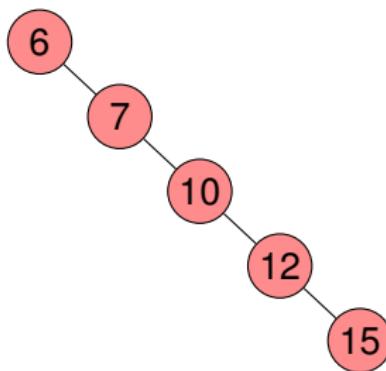
## 4 Arbres binaris de cerca

- Implementació (ABC)

## 5 Arbres AVL

- Implementació (AVL)
- Alçària d'un AVL

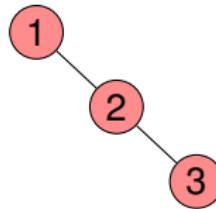
En els **arbres binaris de cerca**, hem vist que el cost de les operacions és  $\Theta(h)$ , on  $h$  és l'alçària. Però  $h$  pot coincidir amb el nombre de nodes:



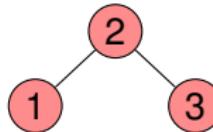
Per tant, el cost en cas pitjor és de  $\Theta(n)$ , on  $n$  és el nombre de nodes.

Per millorar el cost lineal es poden fer dues coses:

- demostrar que si en un ABC s'insereixen els elements de forma aleatòria, l'alçària és  $\approx \log n$
- fer les insercions i els esborrats de manera que l'alçària es mantingui en  $\approx \log n$ . En comptes de tenir



tindríem



## Com mantenir els subarbres equilibrats?

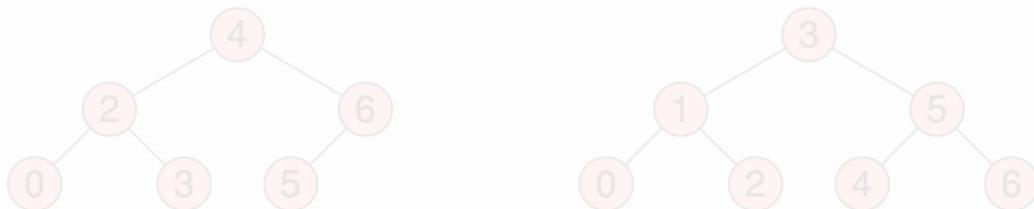
- 1 Forçant que l'ABC sigui **complet**.

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

Però 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 subarbres equilibrats?

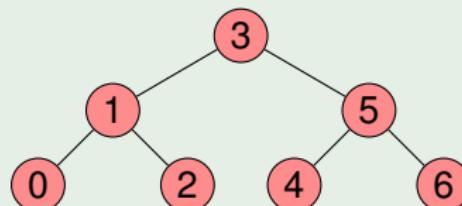
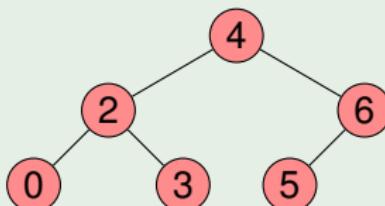
## 1 Forçant que l'ABC sigui **complet**.

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

Però afegir un element pot ser massa costós.

### Exemple

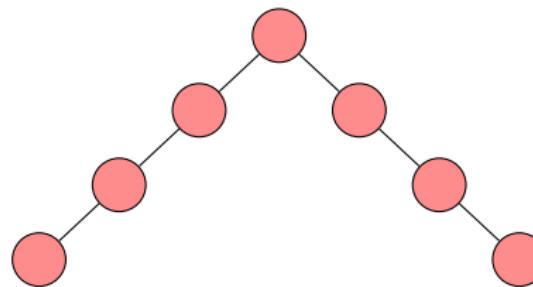
Per afegir l'element 1 en el primer arbre, cal canviar-ho gairebé tot.



## Com mantenir els subarbres equilibrats?

- ② Demanant que els dos subarbres de l'arrel tinguin la **mateixa alçària**.

Però és una condició insuficient: el cost de les operacions bàsiques en l'arbre següent és  $\approx n/2$ :



Com mantenir els subarbres equilibrats?

- ③ Permetent un **petit desequilibri** en les alçàries 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

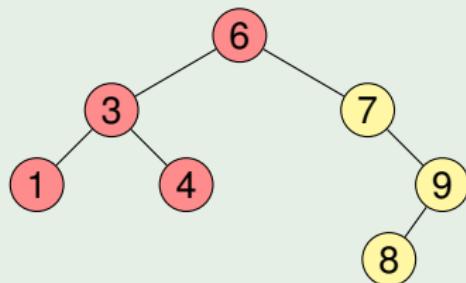
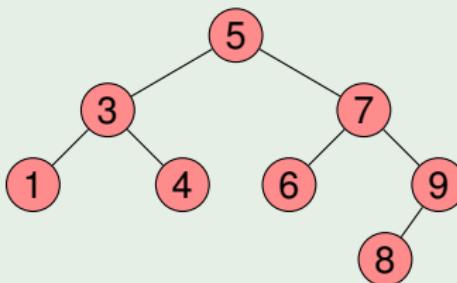
- és buit o
- la diferència d'alçàries dels seus subarbres és com a màxim d'1 i els seus subarbres també estan equilibrats

# Introduction

Els ABC amb la condició d'equilibri s'anomenen **arbres AVL**.

## Exemple

L'arbre de l'esquerra està equilibrat, però si n'esborrem l'arrel com en els ABC, deixarà d'estar equilibrat.



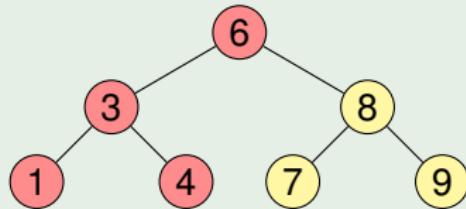
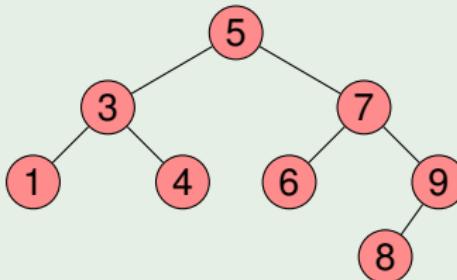
Subarbre desequilibrat en groc

# Introduction

Els ABC amb la condició d'equilibri s'anomenen **arbres AVL**.

## Exemple

L'arbre de l'esquerra està equilibrat, però si n'esborrem l'arrel com en els ABC, deixarà d'estar equilibrat. **Els AVL usen “rotacions” per solucionar-ho.**



Rotació aplicada al subarbre groc

# Introduction

Els ABC amb la condició d'equilibri s'anomenen **arbres AVL**.

AVL és acrònim d'Adelson-Velskii i Landis.



Georgii Adelson-Velskii



Evgenii Landis

# Implementació (AVL)

## Definició de Diccionari i Node

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

```
template <typename Clau, typename Info>
class Diccionari {
private:
    struct Node {
        Clau clau;
        Info info;
        Node* fesq; // Apuntador al fill esquerre
        Node* fdre; // Apuntador al fill dret
        int alc; // Alcària 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; // Apuntador a l'arrel de l'AVL
```

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

En els AVL necessitarem dues funcions senzilles referents a l'alcària, totes dues de cost  $\Theta(1)$ .

## Retornar i actualitzar l'alcària

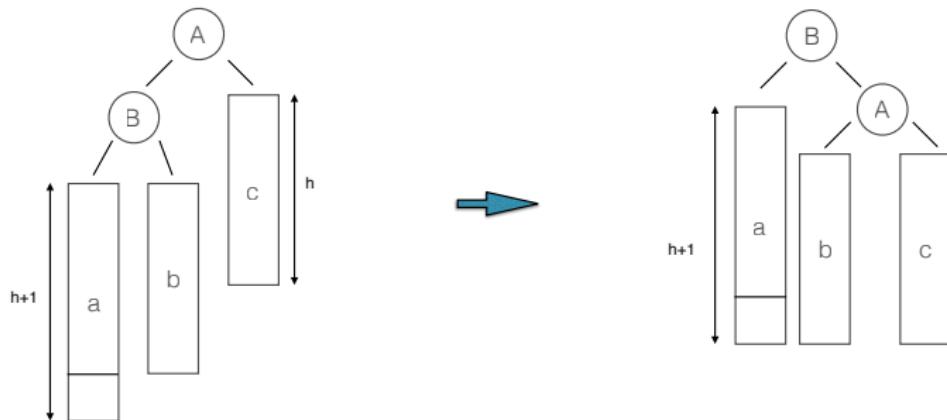
```
static int alcaria (Node* p) {
    return p ? p->alc : -1;
}

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

Per tal de mantenir equilibrat un arbre després d'una assignació o esborrat, considerem quatre **rotacions** de l'arbre: EE, DD, ED i DE.

Totes quatre tenen cost  $\Theta(1)$ .

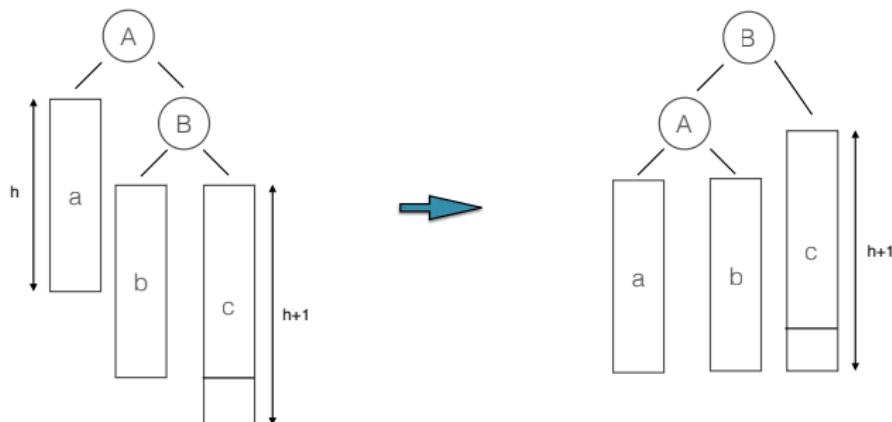
# Implementació (AVL)



## Rotació EE (LL, en anglès)

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

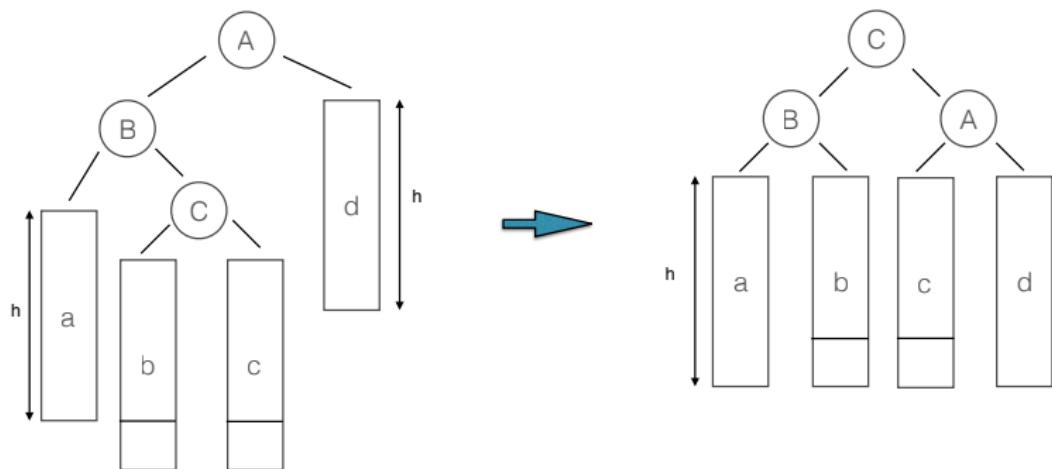
# Implementació (AVL)



## Rotació DD (RR, en anglès)

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

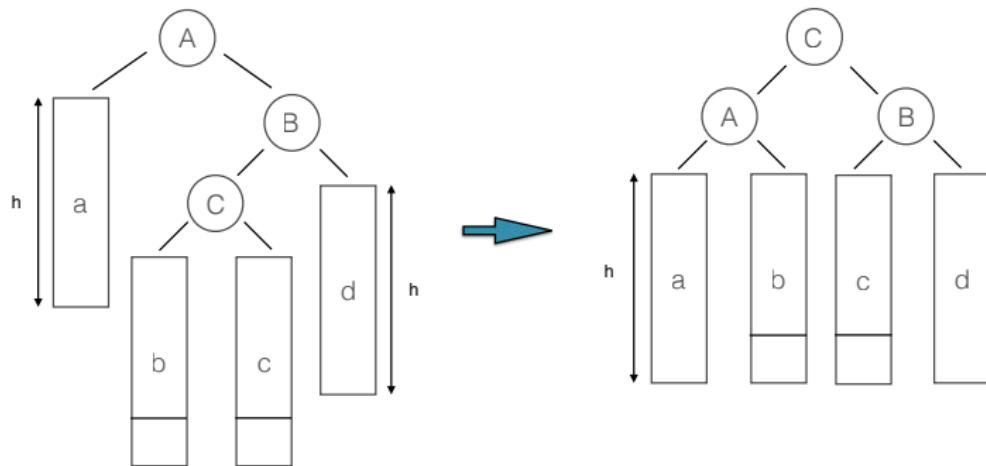
# Implementació (AVL)



## Rotació ED (LR, en anglès)

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

# Implementació (AVL)



## Rotació DE (RL, en anglès)

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

# 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 (alcaria(p->fesq) - alcaria(p->fdre) == 2) {
                if (clau < p->fesq->clau) LL(p);
                else LR(p);
            }
            actualitzar_alcaria(p);
        } else if (clau > p->clau) {
            assignar(p->fdre, clau, info);
            if (alcaria(p->fdre) - alcaria(p->fesq) == 2) {
                if (clau > p->fdre->clau) RR(p);
                else RL(p);
            }
            p->info = info;
        } else {
            p = new Node(clau, info, nullptr, nullptr, 0);
            ++n;
        }
    }
}
```

En **mitjana**, es fa una rotació cada dues assignacions.

En el **cas pitjor**:

- **assignar i esborrar** fan  $\Theta(\log n)$  rotacions, on una rotació costa  $\Theta(1)$   
 $\Rightarrow \Theta(\log n)$
- el cost de **consultar**, com en els ABC, és  $\Theta(\log h)$ , on  $h$  és l'alçària de l'arbre  $\Rightarrow \Theta(\log n)$

## Proposició

En els AVL, les operacions **assignar, esborrar i consultar** tenen cost en cas pitjor  $\Theta(\log n)$ .

## Exercici 1

Un node en un arbre binari és un **fill únic** si té pare però no germà. El **factor de soledat** d'un arbre binari  $T$  de  $n$  nodes,  $\mathcal{FS}(T)$ , es defineix com:

$$\mathcal{FS}(T) = \frac{\text{nombre de fills únics a } T}{n}.$$

- ① Demostreu que si  $T$  és un AVL no buit, llavors  $\mathcal{FS}(T) \leq 1/2$
- ② És cert que si  $T$  és un arbre binari i  $\mathcal{FS}(T) \leq 1/2$ , llavors l'alçària de  $T$  és  $O(\log n)$ ?

## Solució

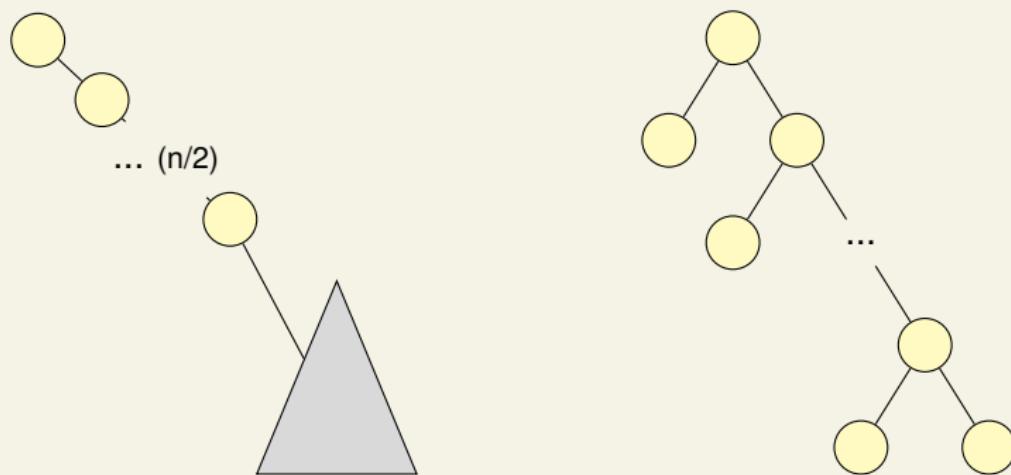
- 1 Demostreu que si  $T$  és un AVL no buit, llavors  $\mathcal{FS}(T) \leq 1/2$

En un AVL els fills únics només poden aparèixer a les fulles (altrament, es produiria un desequilibri de 2 o més). Com que cada fill únic té un pare diferent (que a més no és fill únic), el nombre total de fills únics ha de ser com a màxim  $\lfloor n/2 \rfloor$ . Però llavors  $\mathcal{FS}(T) \leq \frac{\lfloor n/2 \rfloor}{n} \leq \frac{n/2}{n} = \frac{1}{2}$ .

## Solució

- ② És cert que si  $T$  és un arbre binari i  $\mathcal{FS}(T) \leq 1/2$ , llavors l'alçària de  $T$  és  $O(\log n)$ ?

No, si  $\mathcal{FS}(T) \leq 1/2$  llavors hi pot haver en  $T$  fins a  $\lfloor n/2 \rfloor$  fills únics. Això permet una alçària d'almenys  $n/2$ :



## Exercici 2

Inseriu la seqüència següent de claus en un AVL partint de l'arbre buit mitjançant l'operació assignar: 11, 29, 13, 31, 37, 17, 19, 23.

# Alçària d'un AVL

Veurem el fet següent sobre AVLs.

## Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

Per demostrar-lo, definim

$$n_k = \text{mínim nombre de nodes d'un AVL d'alçària } k$$

## Exercici

- ① Calculeu  $n_0$ ,  $n_1$ ,  $n_2$  i  $n_3$
- ② A partir d'aquests nombres, obtingueu una definició inductiva per a  $n_k$

# Alçària d'un AVL

Veurem el fet següent sobre AVLs.

## Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

Per demostrar-lo, definim

$$n_k = \text{mínim nombre de nodes d'un AVL d'alçària } k$$

## Solució

Obtenim  $n_0 = 1$ ,  $n_1 = 2$ ,  $n_2 = 4$  i  $n_3 = 7$ .

En general, podem definir  $n_k$  inductivament com:

- $n_0 = 1$
- $n_1 = 2$
- $n_k = \underbrace{n_{k-1}}_{\substack{\text{cal un subarbre} \\ \text{d'alçària } k-1}} + \underbrace{n_{k-2}}_{\substack{\text{l'alçària } k-2 \text{ dona} \\ \text{mín. # de nodes}}} + 1$

# Alçària d'un AVL

## Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

## Demostració

- 1 Com que  $n_m \geq n_{m-1}$  per a tot  $m > 0$ , tenim

$$n_k = n_{k-1} + n_{k-2} + 1 \geq 2n_{k-2} + 1 \geq 2n_{k-2}$$

- 2 Per substitució repetida, tenim

$$n_k \geq 2n_{k-2} \geq 4n_{k-4} \geq \dots \geq \underbrace{2^{\frac{k+1}{2}}}_{\text{k senar}} \geq \underbrace{2^{\frac{k}{2}}}_{\text{k parell}}$$

- 3 Donat un AVL  $T$  d'alçària  $k$  i  $n$  nodes:

$$n \geq n_k \geq 2^{\frac{k}{2}}$$

- 4 Prenent logaritmes,  $k \leq 2 \log_2 n \in O(\log n)$

# Alçària d'un AVL

## Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

## Demostració

- 1 Com que  $n_m \geq n_{m-1}$  per a tot  $m > 0$ , tenim

$$n_k = n_{k-1} + n_{k-2} + 1 \geq 2n_{k-2} + 1 \geq 2n_{k-2}$$

- 2 Per substitució repetida, tenim

$$n_k \geq 2n_{k-2} \geq 4n_{k-4} \geq \dots \geq \underbrace{2^{\frac{k+1}{2}}}_{\text{k senar}} \geq \underbrace{2^{\frac{k}{2}}}_{\text{k parell}}$$

- 3 Donat un AVL  $T$  d'alçària  $k$  i  $n$  nodes:

$$n \geq n_k \geq 2^{\frac{k}{2}}$$

- 4 Prenent logaritmes,  $k \leq 2\log_2 n \in O(\log n)$

# Alçària d'un AVL

## Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

## Demostració

- 1 Com que  $n_m \geq n_{m-1}$  per a tot  $m > 0$ , tenim

$$n_k = n_{k-1} + n_{k-2} + 1 \geq 2n_{k-2} + 1 \geq 2n_{k-2}$$

- 2 Per substitució repetida, tenim

$$n_k \geq 2n_{k-2} \geq 4n_{k-4} \geq \dots \geq \underbrace{2^{\frac{k+1}{2}}}_{k \text{ senar}} \geq \underbrace{2^{\frac{k}{2}}}_{k \text{ parell}}$$

- 3 Donat un AVL  $T$  d'alçària  $k$  i  $n$  nodes:

$$n \geq n_k \geq 2^{\frac{k}{2}}$$

- 4 Prenent logaritmes,  $k \leq 2 \log_2 n \in O(\log n)$

# Alçària d'un AVL

## Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

## Demostració

- 1 Com que  $n_m \geq n_{m-1}$  per a tot  $m > 0$ , tenim

$$n_k = n_{k-1} + n_{k-2} + 1 \geq 2n_{k-2} + 1 \geq 2n_{k-2}$$

- 2 Per substitució repetida, tenim

$$n_k \geq 2n_{k-2} \geq 4n_{k-4} \geq \dots \geq \underbrace{2^{\frac{k+1}{2}}}_{k \text{ senar}} \geq \underbrace{2^{\frac{k}{2}}}_{k \text{ parell}}$$

- 3 Donat un AVL  $T$  d'alçària  $k$  i  $n$  nodes:

$$n \geq n_k \geq 2^{\frac{k}{2}}$$

- 4 Prenent logaritmes,  $k \leq 2 \log_2 n \in O(\log n)$

# Alçària d'un AVL

## Teorema

L'alçària d'un AVL de  $n$  nodes és  $O(\log n)$ .

## Demostració

- 1 Com que  $n_m \geq n_{m-1}$  per a tot  $m > 0$ , tenim

$$n_k = n_{k-1} + n_{k-2} + 1 \geq 2n_{k-2} + 1 \geq 2n_{k-2}$$

- 2 Per substitució repetida, tenim

$$n_k \geq 2n_{k-2} \geq 4n_{k-4} \geq \dots \geq \underbrace{2^{\frac{k+1}{2}}}_{k \text{ senar}} \geq \underbrace{2^{\frac{k}{2}}}_{k \text{ parell}}$$

- 3 Donat un AVL  $T$  d'alçària  $k$  i  $n$  nodes:

$$n \geq n_k \geq 2^{\frac{k}{2}}$$

- 4 Prenent logaritmes,  $k \leq 2 \log_2 n \in O(\log n)$

La definició dels nombres de Fibonacci és:

- $F_0 = 1$
- $F_1 = 1$
- $F_k = F_{k-1} + F_{k-2}$  for  $k > 1$

## Exercici

Demostreu que  $n_k = F_{k+2} - 1$ .