



# Tema 4 Estructures no lineals: Arbres

**Maria Salamó Llorente**  
**Estructura de Dades**

Enginyeria Informàtica  
Facultat de Matemàtiques i Informàtica,  
Universitat de Barcelona



# Contingut

- 4.1 Introducció als arbres
- 4.2 Arbres binaris
- 4.3 Recorreguts en arbres binaris
- 4.4. Arbres de cerca binària
- 4.5. Arbres AVL



## 4.1 Introducció als arbres



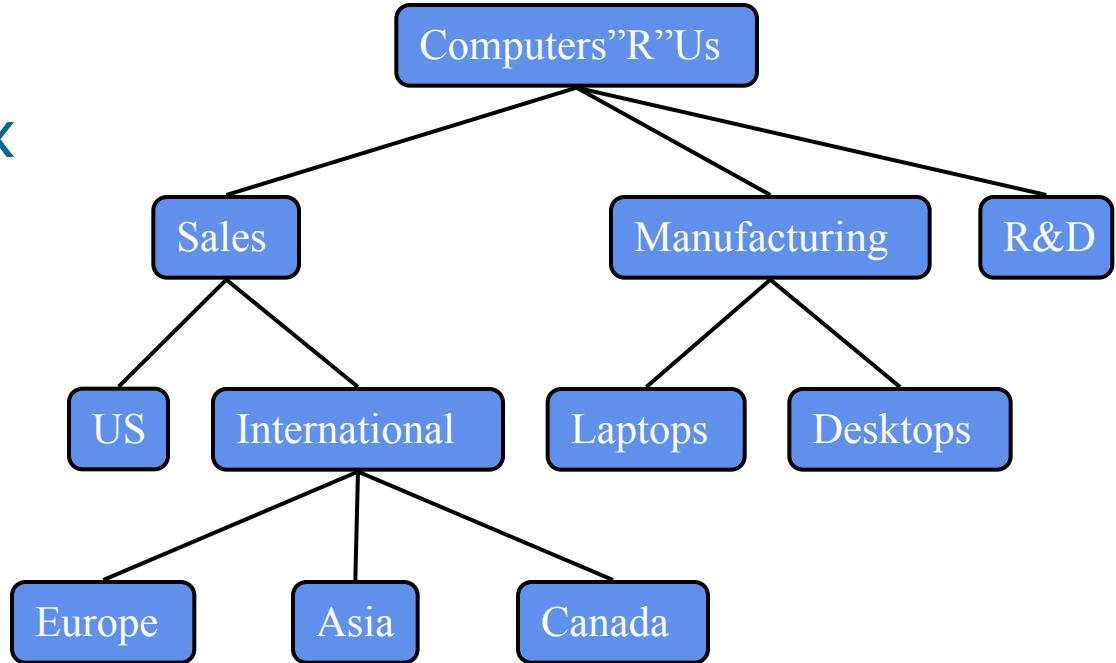


# Introducció

- Les llistes encadenades són estructures de dades lineals
  - seqüèncials, un element darrera de l'altre
- El arbres
  - Junt amb els grafs són estructures de dades no lineals
  - Solventen els inconvenients de les llistes
  - Ofereixen diferents recorreguts
- Perquè són útils?
  - Són útils per **cercar i recuperar informació**

# Arbres

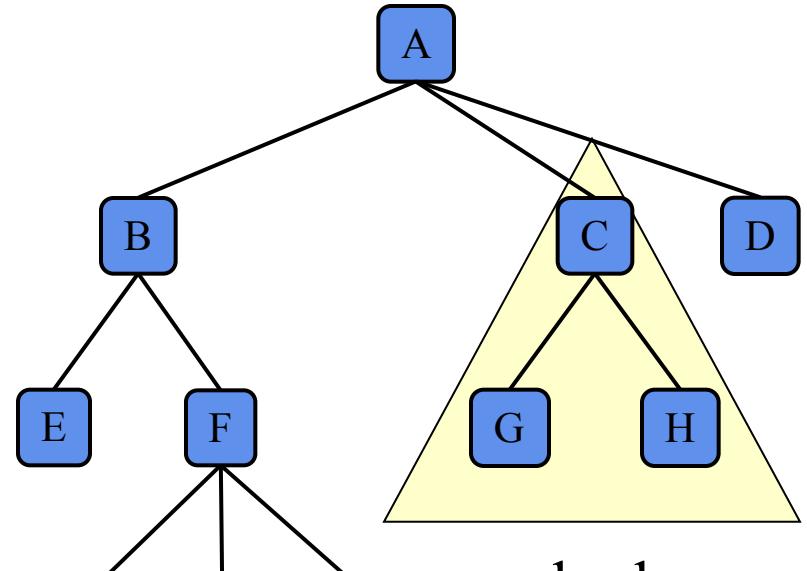
- Model abstracte d'una **estructura jeràrquica**
- Un arbre consisteix en nodes que tenen una relació pare-fill
- Aplicacions:
  - Organització de mapes
  - Sistemes de fitxers
  - Entorns de programació



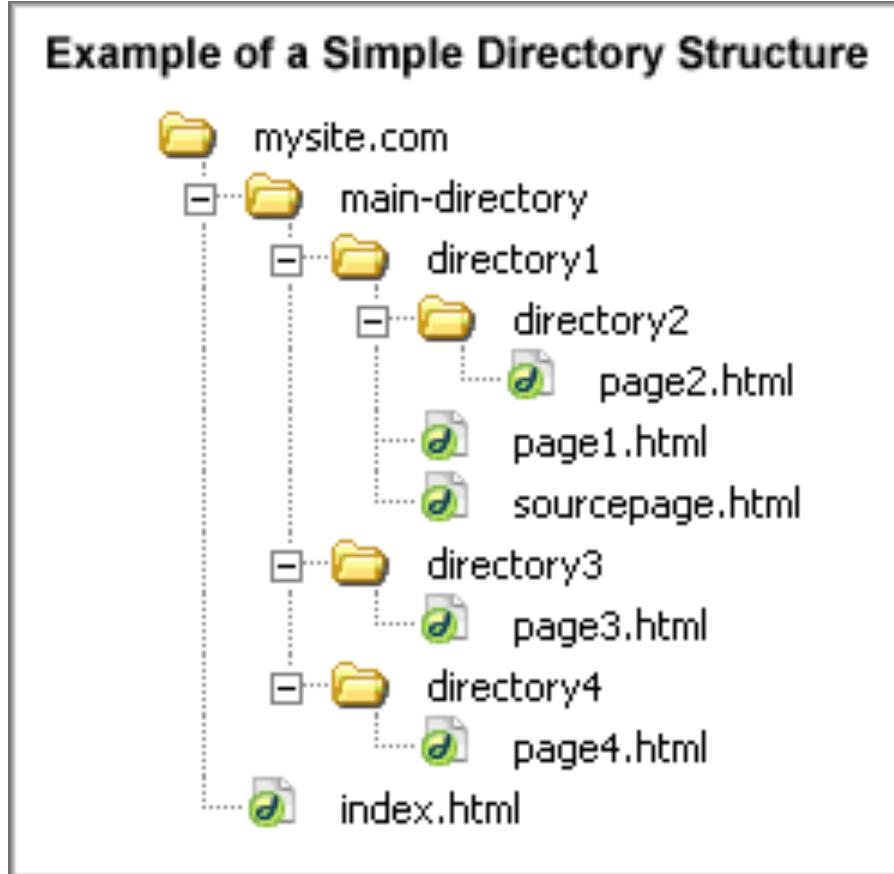
# Terminologia

- **Arrel:** node sense pare (A)
- **Node intern:** node amb com a mínim un fill (A, B, C, F)
- **Node extern (fulla):** node sense fills (E, I, J, K, G, H, D)
- **Ancestres d'un node:** pare, avi, besavi, etc.
- **Profunditat d'un node:** nombre d'ancestres
- **Alçada d'un arbre:** màxima profunditat de qualsevol node (3)
- **Descendent d'un node:** fill, net, besné, etc.

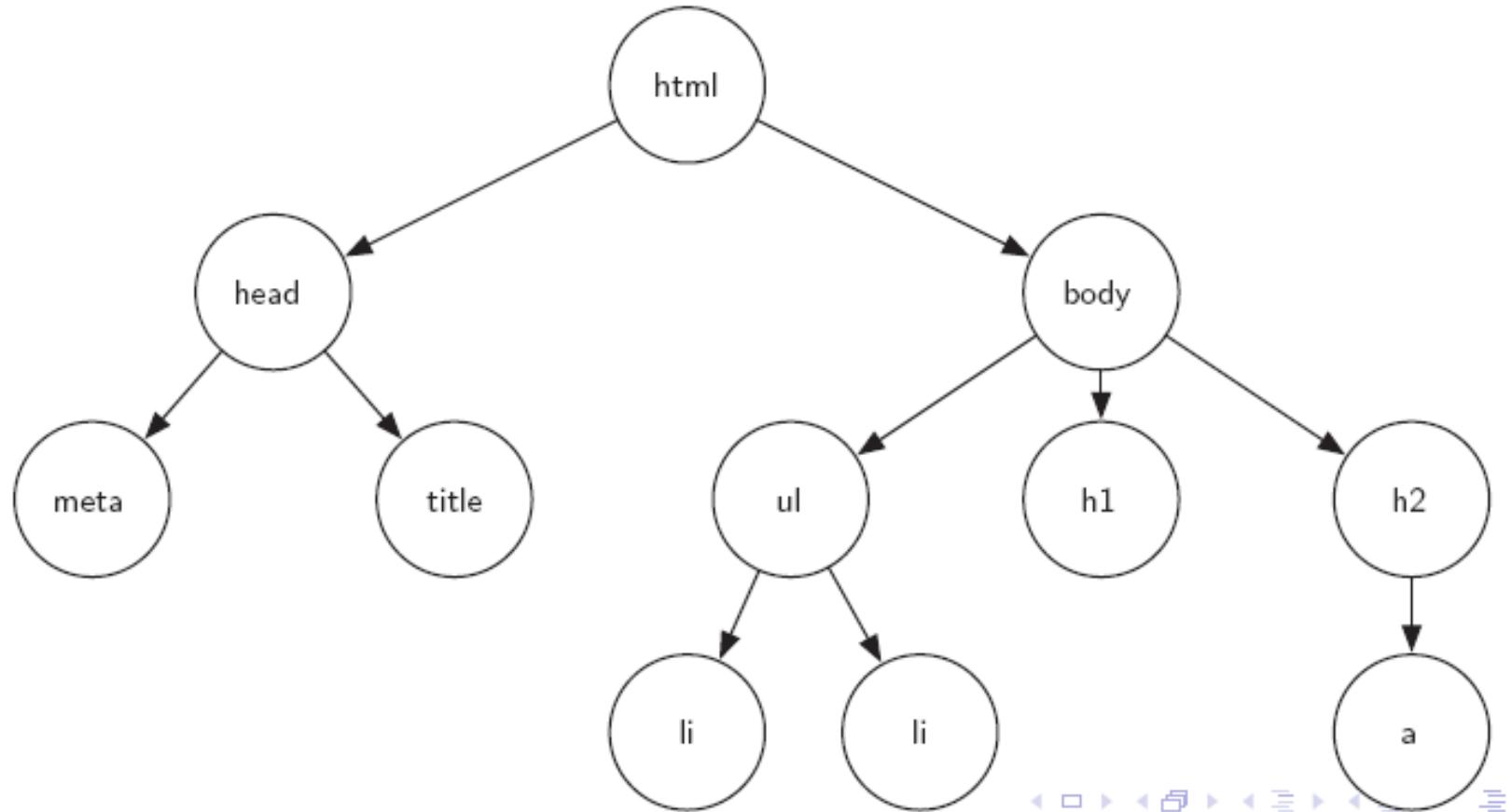
- **Subarbre:** arbre format per un node i els seus descendents



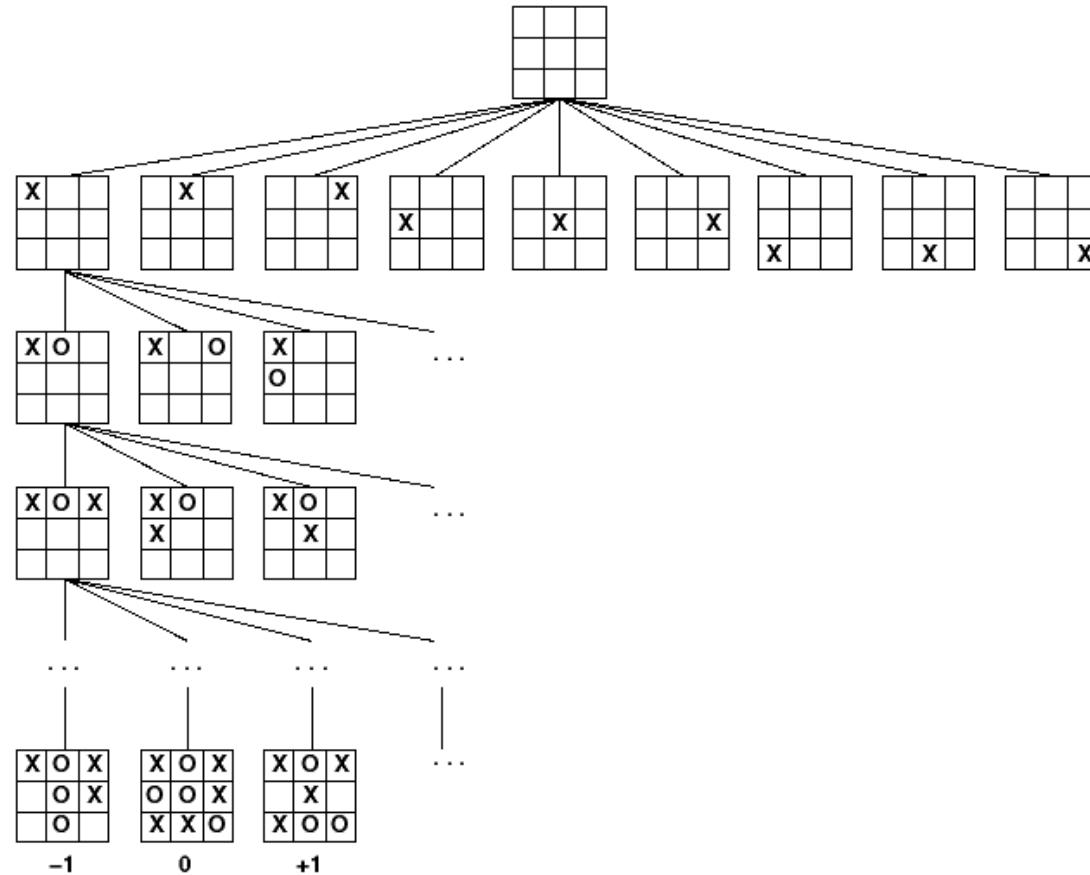
# Exemple estructura de directoris i fitxers



# Exemple d'arbre d'etiquetes d'una pàgina web

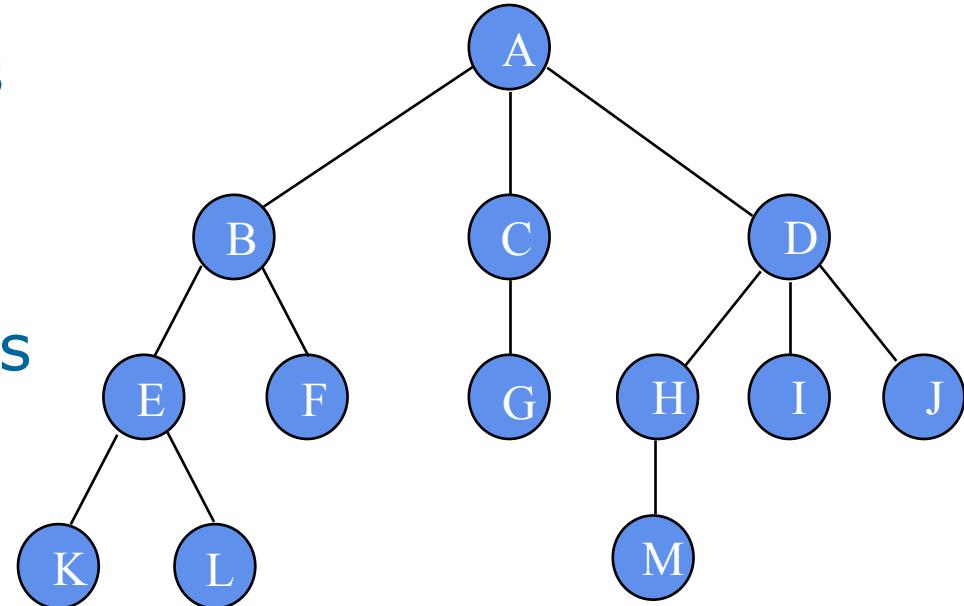


# Exemple joc tres en ratlla



# Propietats d'un arbre

- **Primera propietat:** els arbres són jeràrquics
- **Segona propietat:** Tots els fills d'un node són independents
- **Tercera propietat:** El camí fins a qualsevol node extern (fulla) és únic





# TAD Arbre (Tree)

- **Mètodes genèrics:**

- `integer size()`
- `boolean empty()`
- `list<position> positions()`

- **Mètodes d'accés:**

- `position root()`
- `position p.parent()`
- `list<position> p.children()`

- **Mètodes de consulta:**

- `boolean p.isRoot()`
- `boolean p.isExternal()`

- Mètodes modificadors:

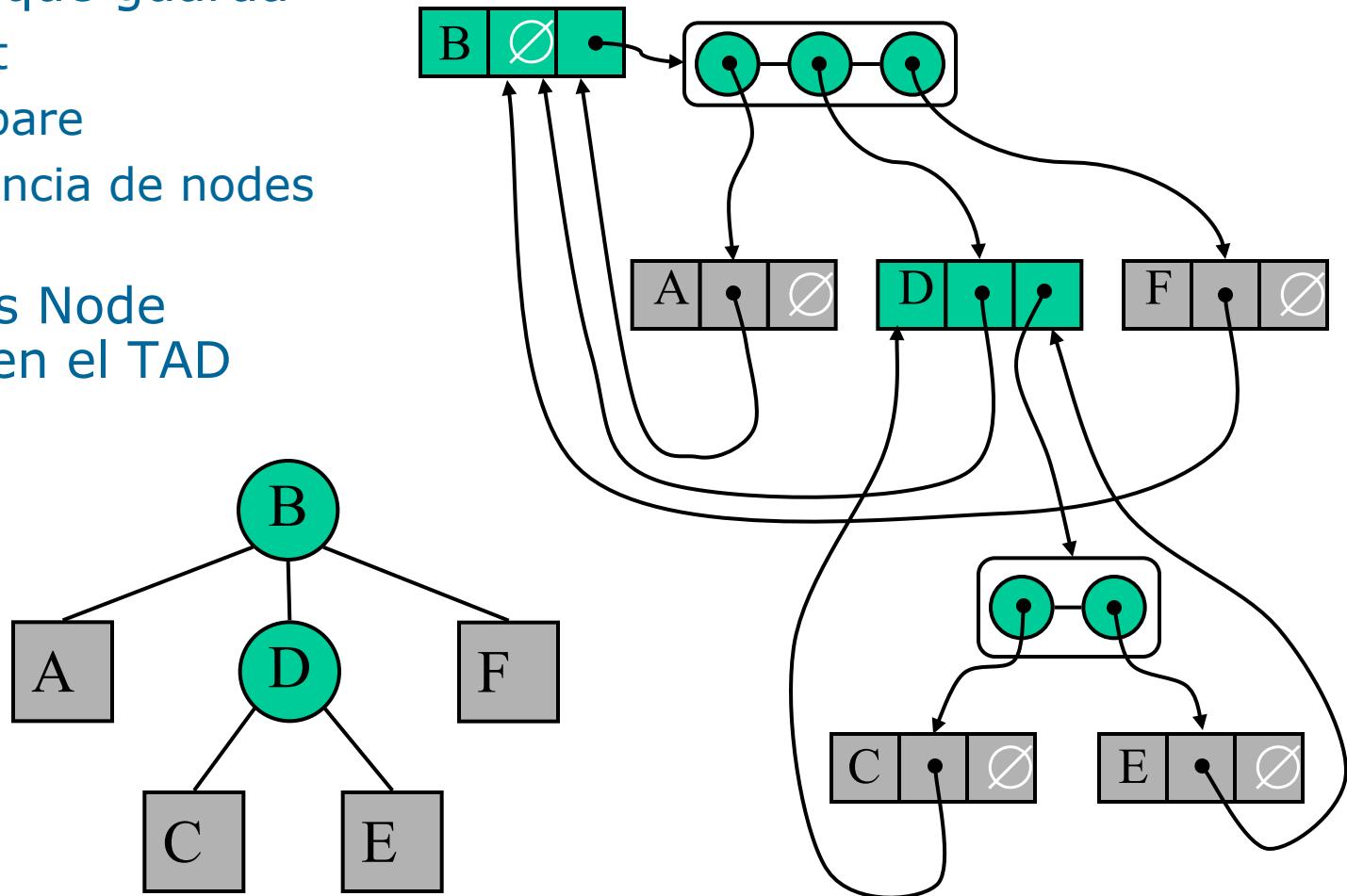
- Es poden definir diferents mètodes de modificació segons la implementació escollida de l'arbre

## Excepcions:

- Posició invàlida, Arbre buit
- Sobrepassar la frontera de l'arbre

# Arbres en estructura encadenada

- Un node es representa per un objecte que guarda
  - l'Element
  - El node pare
  - La seqüència de nodes fills
- Els objectes Node implementen el TAD Position





# Interfície en C++ (no està completa)

```
template <class E>
class Position<E>{
public:
    E& operator*();
    Position parent() const;
    PositionList children() const;
    bool isRoot() const;
    bool isExternal() const;
};
```

- Les posicions d'un arbre són els seus nodes
- operator\* s'usa per retornar l'element que guarda el node



# Interfície en C++ (no està completa)

```
template <class E>
class Tree<E>{
public:
    int size() const;
    bool empty() const;
    Position root() const;
    PositionList positions() const;
};
```

- **PositionList** segueix l'estàndar TAD Llista
  - Es podria implementar amb std::list<Position>

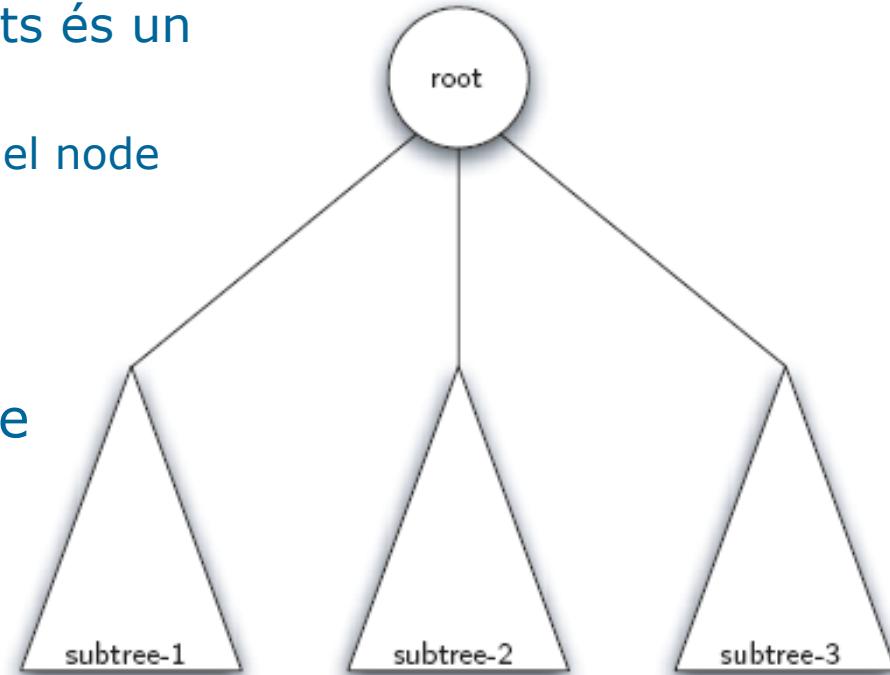
# Definició recursiva d'arbre

- **Definició recursiva:** l'arbre és un conjunt finit de nodes que compleix:
  - Existeix un node arrel
  - La resta de nodes estan en  $n$  ( $n \geq 0$ ) particions de conjunts disjunts  $T_1, T_2, \dots, T_n$  on cadascun d'aquests conjunts és un arbre
    - $T_1, T_2, \dots, T_n$  són els subarbres del node arrel

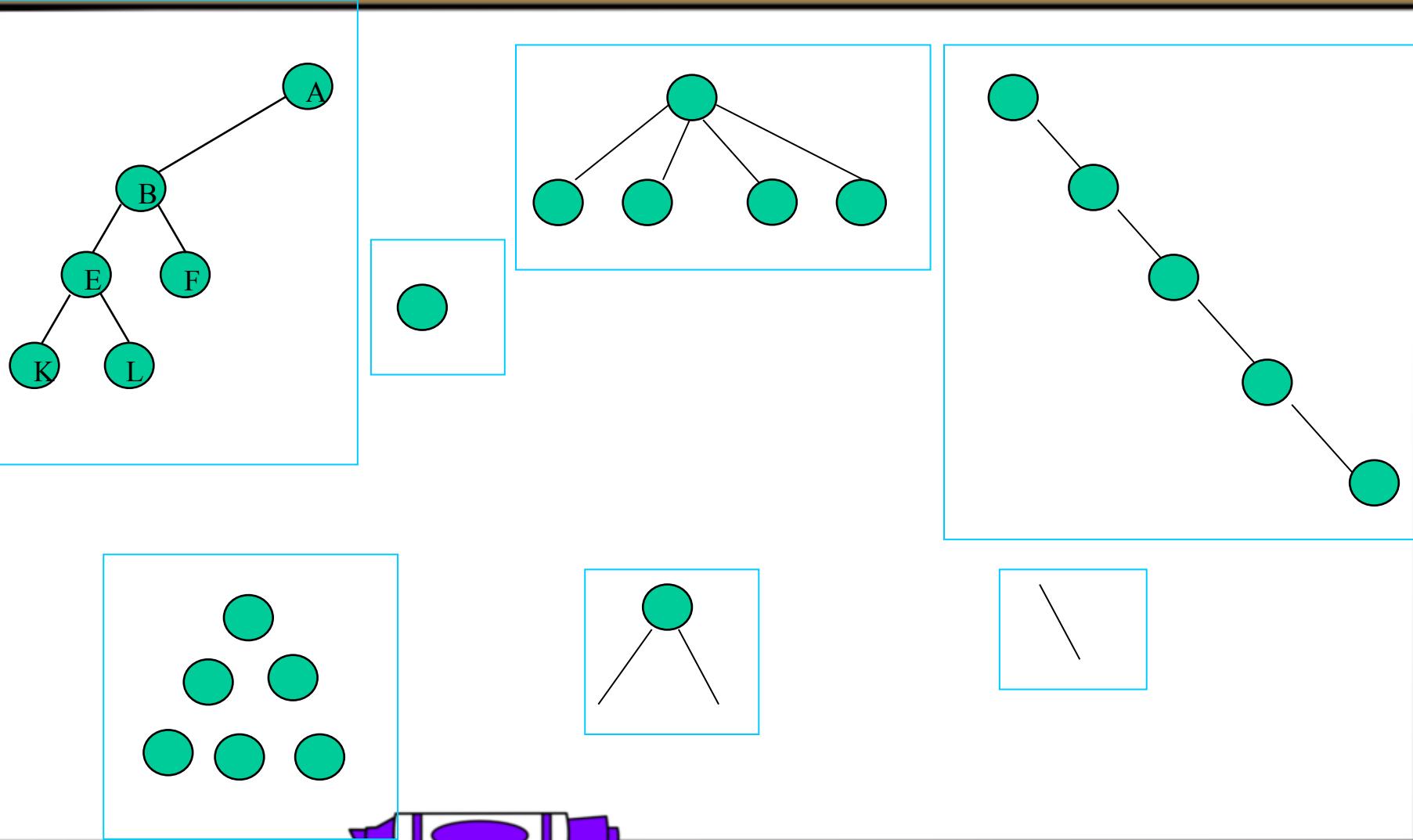
- **Funcions com:**

- Profunditat, Alçada d'un arbre
- Nivell d'un node,
- Grau d'un arbre

**Es calculen de forma recursiva**

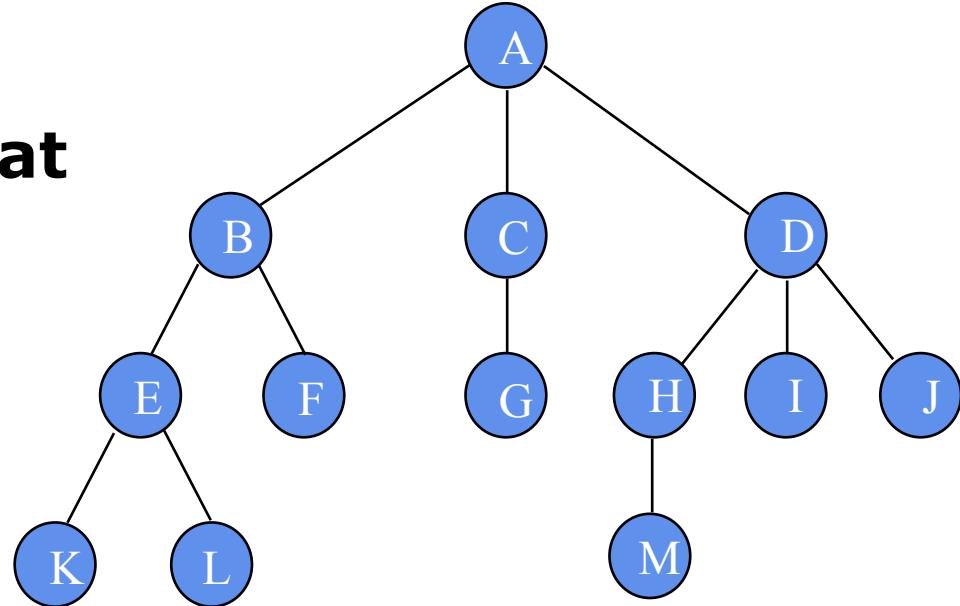


# Són arbres?



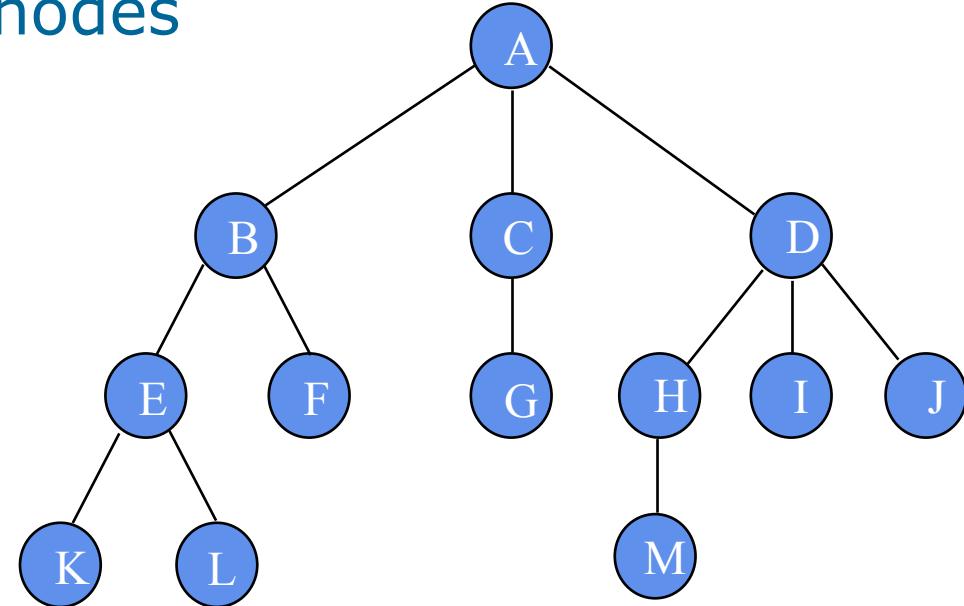
# Nivell d'un node

- El **nivell** d'un node es defineix com:
  - El nivell del **node arrel** és **0**
  - Si un node està en el nivel L, els seus fills estan en el nivell L+1
- La **alçada o profunditat** d'un arbre és el màxim nivell



# Com es mesura el grau d'un arbre

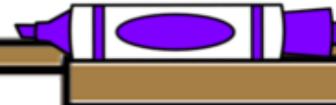
- El nombre de fills d'un node és el grau del node
  - Les **fulles o nodes externs** tenen grau zero
- **Grau d'un arbre:** és el màxim grau dels seus nodes





# Profunditat

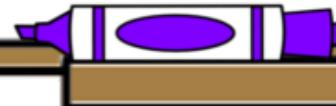
```
int depth(const Tree& T, const Position & p)
{
    ...
}
```





# Alçada 1

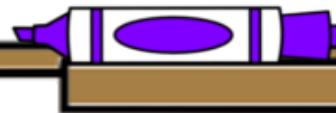
```
int height1(const Tree& T)
{
}
}
```





# Alçada 2

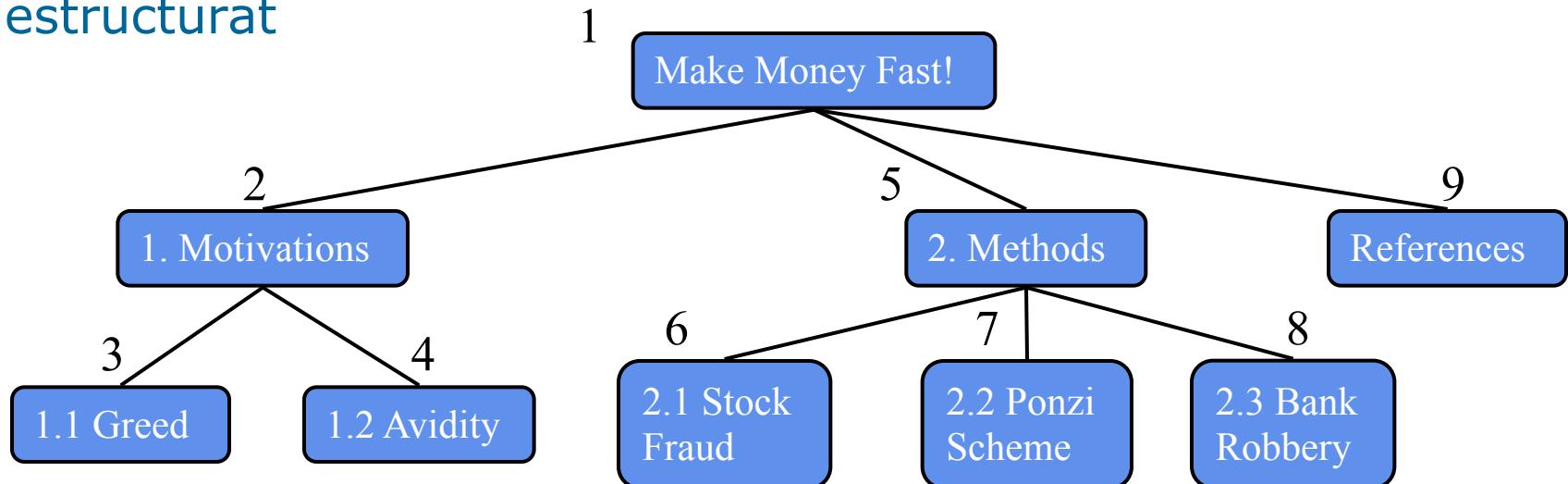
```
int height2(const Tree& T, const Position& p)
{
}
}
```



# Recorregut en preordre

- Un recorregut visita els nodes d'un arbre d'una manera sistemàtica
- En un recorregut en preordre, un node es visita abans que els seus descendents
- Aplicació: imprimir un document estructurat

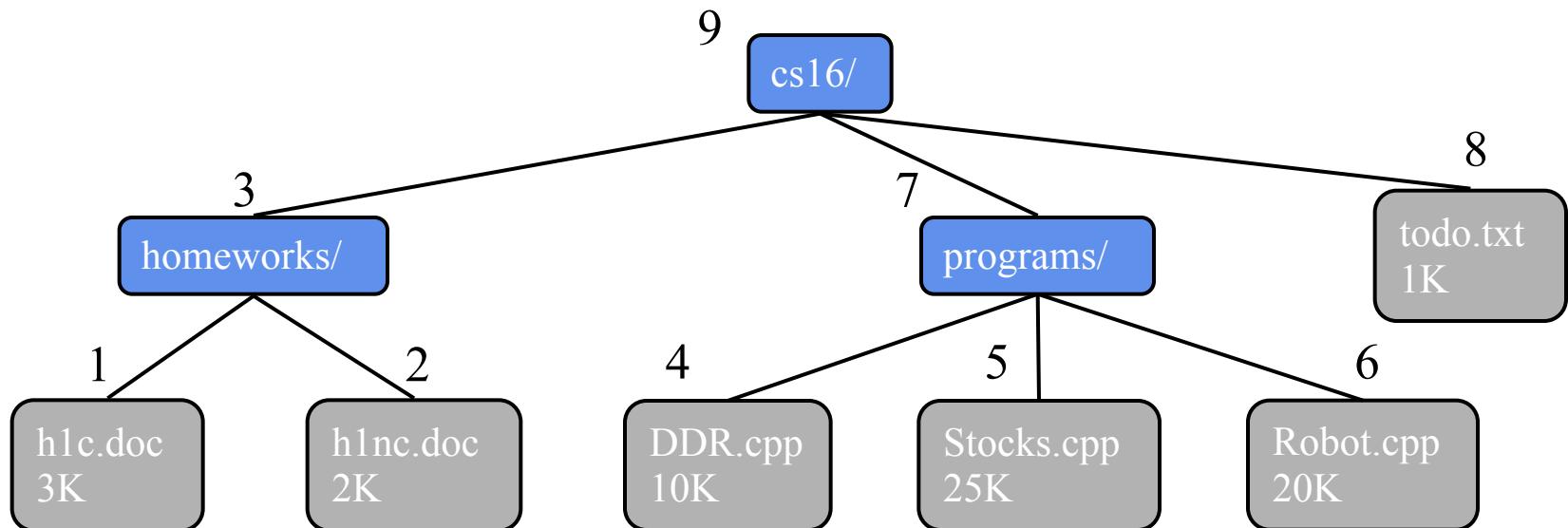
```
Algorithm preOrder(v)
    visit(v)
    for each child w of v
        preOrder (w)
```



# Recorregut en postordre

- Un node es visita després dels seus nodes descendents
- Aplicació: calcular l'espai usat pels fitxers en un directori i subdirectoris

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





# Exercicis

**Exercici 1.** Implementeu el recorregut preordre en C++

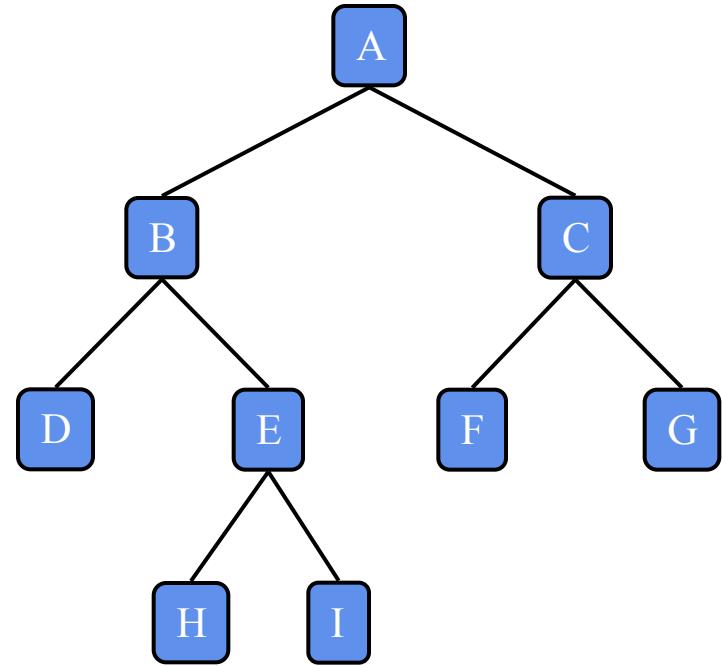
**Exercici 2.** Implementeu el recorregut en postordre en C++



## 4.2 Arbres binaris

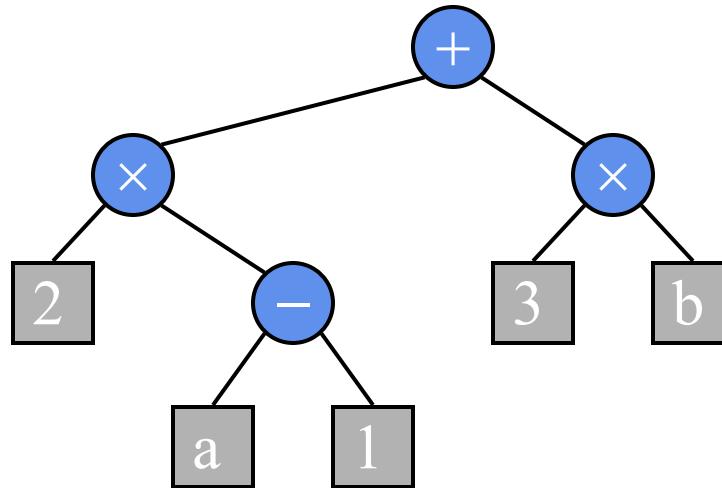
# Arbres binaris

- Un arbre binari és un arbre amb les següents **propietats**:
  - Cada node intern té com a màxim dos fills
  - Els fills d'un node són un parell ordenat
- Els fills d'un node intern s'anomenen **fill esquerra i fill dret**
- **Exemples:**
  - Expressions aritmètiques
  - Processos de decisió
  - Cerques



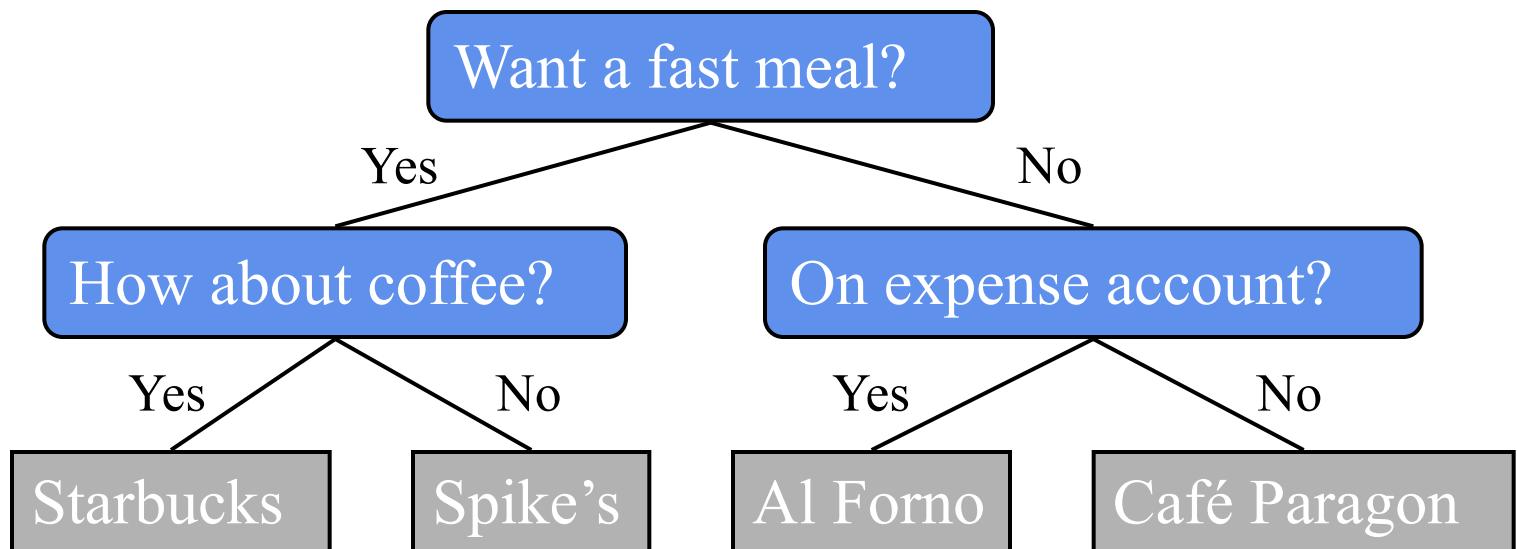
# Arbre d'expressions aritmètiques

- Es pot associar un arbre binari a una expressió aritmètica
  - nodes interns: operadors
  - nodes externs: operands
- Exemple:  $(2 \times (a - 1) + (3 \times b))$



# Arbre de decisió

- Es pot associar un arbre binari a un procés de decisió
  - nodes interns: preguntes amb resposta si/no
  - nodes externs: decisions
- Exemple: decisió per on anar a sopar



# Propietats dels arbres binaris

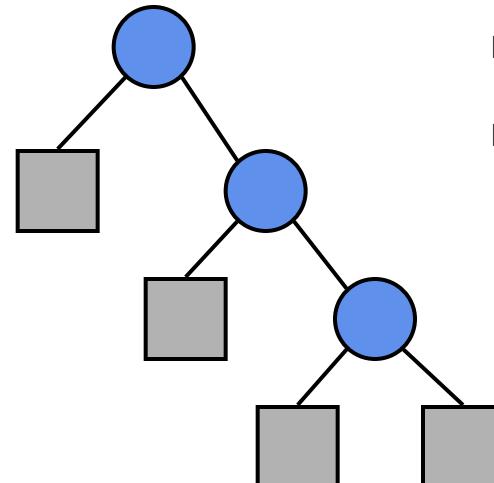
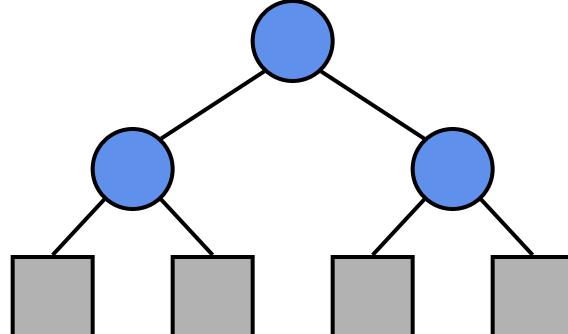
## • Notació

$n$  nombre de nodes

$e$  nombre de nodes externs

$i$  nombre de nodes interns

$h$  alçada



## ◆ Propietats:

- $e = i + 1$
- $n = 2e - 1$
- $h \leq i$
- $h \leq (n - 1)/2$
- $e \leq 2^h$
- $h \geq \log_2 e$
- $h \geq \log_2 (n + 1) - 1$



# TAD BinaryTree

- El TAD arbre binari (**TAD BinaryTree**) extén el **TAD Tree**
  - hereta tots els mètodes del TAD Tree
- Mètodes addicionals:
  - position p.left()
  - position p.right()
- Arbre binari complet: Cada node té 0 o 2 fills



# Interfície en C++ (no està completa)

```
template <class E>
class Position<E>{
public:
    E& operator*(); // getElement
    Position left() const;
    Position right() const;
    Position parent() const;
    bool isRoot() const;
    bool isExternal() const;
};
```

- Les posicions d'un arbre són els seus nodes
- operator\* s'usa per retornar l'element que guarda el node



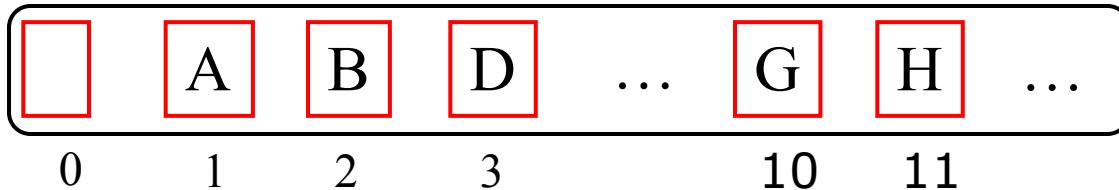
# Interfície en C++ (no està completa)

```
template <class E>
class BinaryTree<E>{
public:
    int size() const;
    bool empty() const;
    Position root() const;
    PositionList positions() const;
};
```

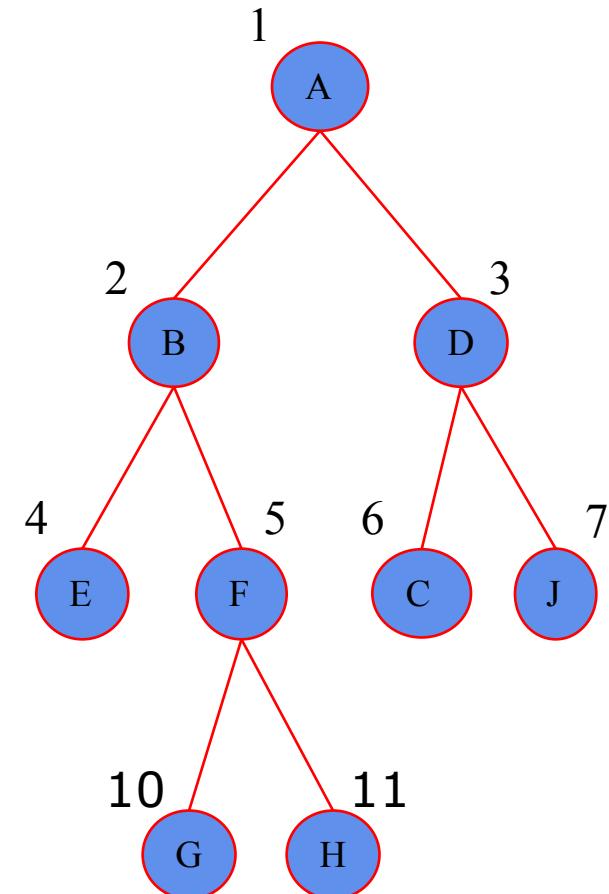
- **PositionList** segueix l'estàndar TAD Llista
  - Es podria implementar amb std::list<Position>

# Implementació basada en vectors

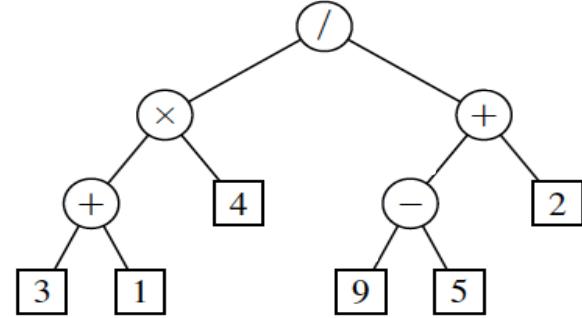
- Els nodes es guarden en un array A



- El node v es guarda a  $A[\text{rank}(v)]$ 
  - $A[1] = \text{root}$
  - si el node és fill esquerra del seu pare
    - $\text{si } A[i] = \text{pare} \rightarrow A[2*i] = \text{fill}$
  - si el node és fill dret del seu pare
    - $\text{si } A[i] = \text{pare} \rightarrow A[2*i + 1] = \text{fill}$



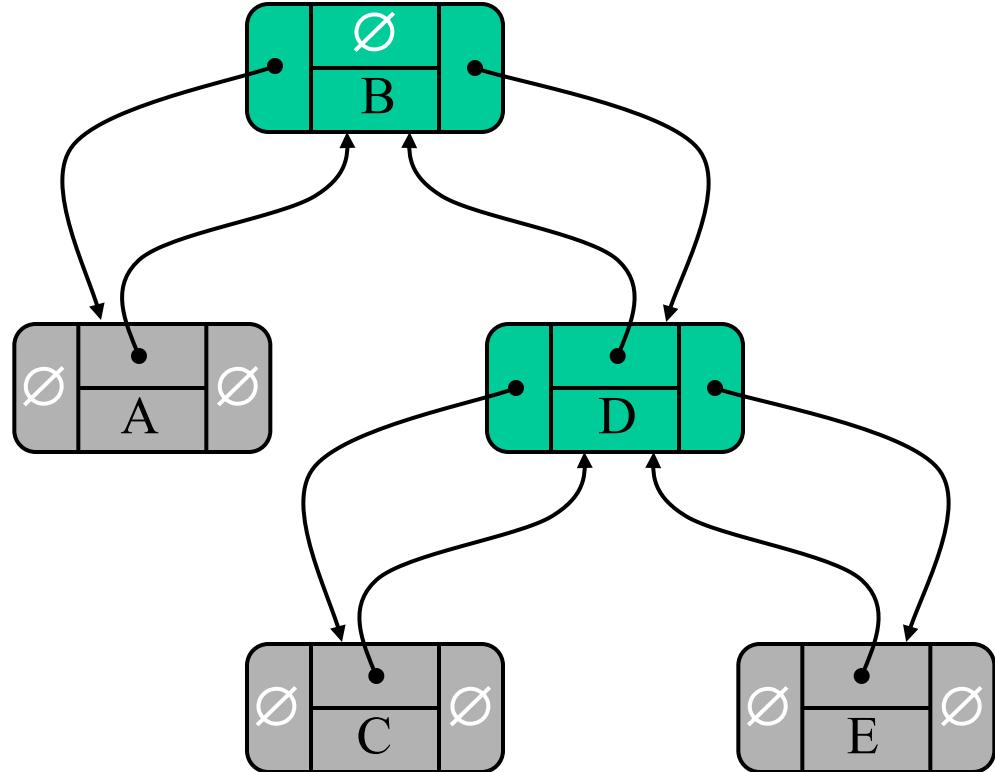
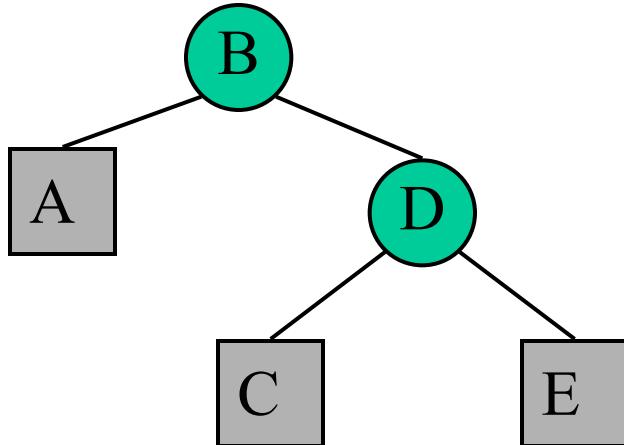
# Exemple



0	1	2	3	4	5	6	7	8	9	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	1	1	2	3	4

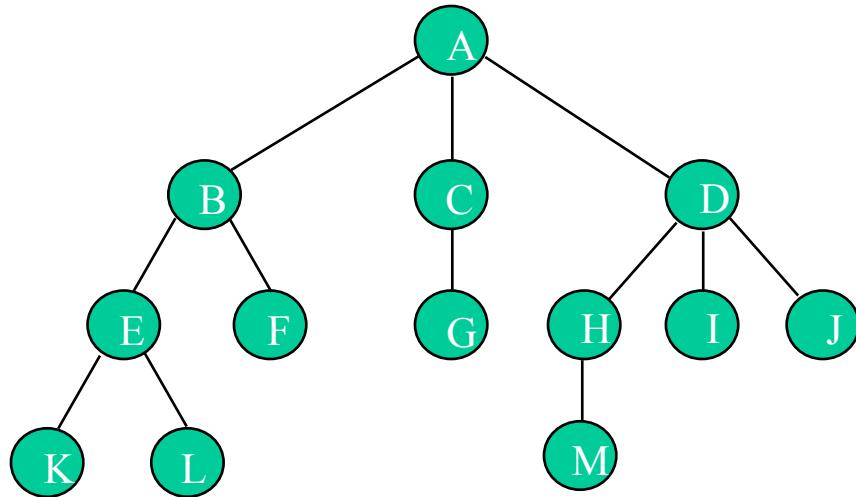
# Arbres binaris en estructura encadenada (LinkedBinaryTree)

- Un node es representa per un objecte que guarda
  - L'Element
  - El node pare
  - El node esquerra
  - El node dret
- Els objectes Node implementen el TAD Position

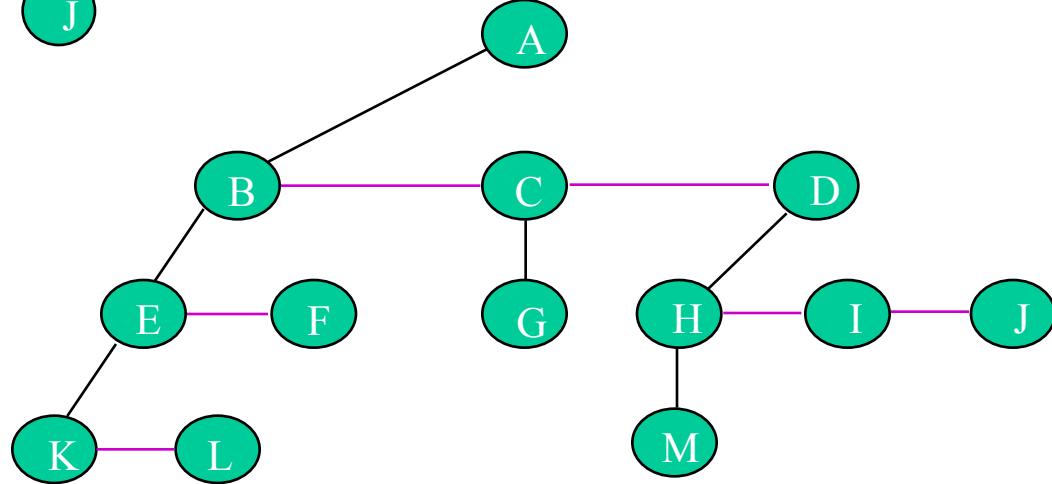


# Com representar arbres amb arbres binaris

**Propietat:** Un arbre de qualsevol grau es pot representar com un arbre binari

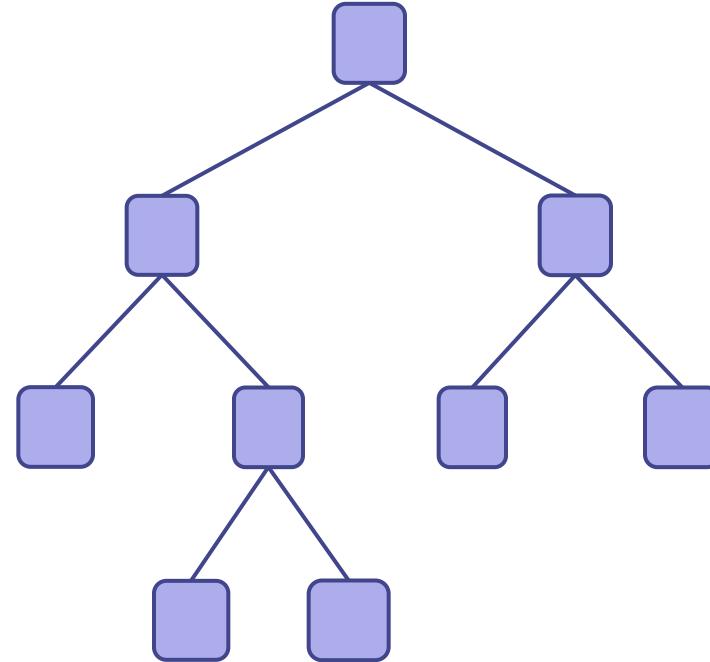


**Regla:** Els fills esquerres continuen sent fills esquerres i els seus germans passen a ser fills drets

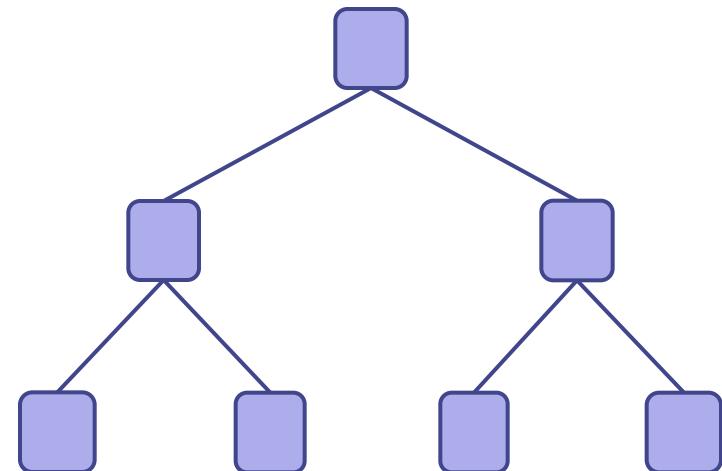


# Arbre binari perfecte

- Un arbre binari és **perfecte** si cada nivell està completament ple



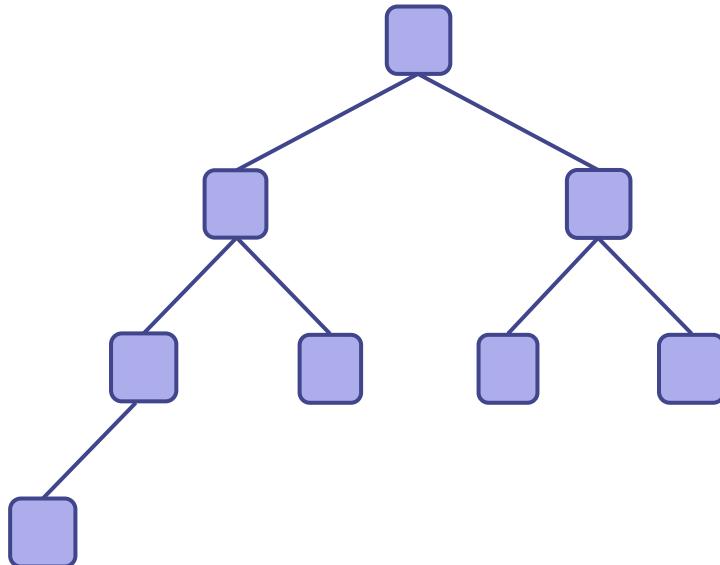
No perfecte



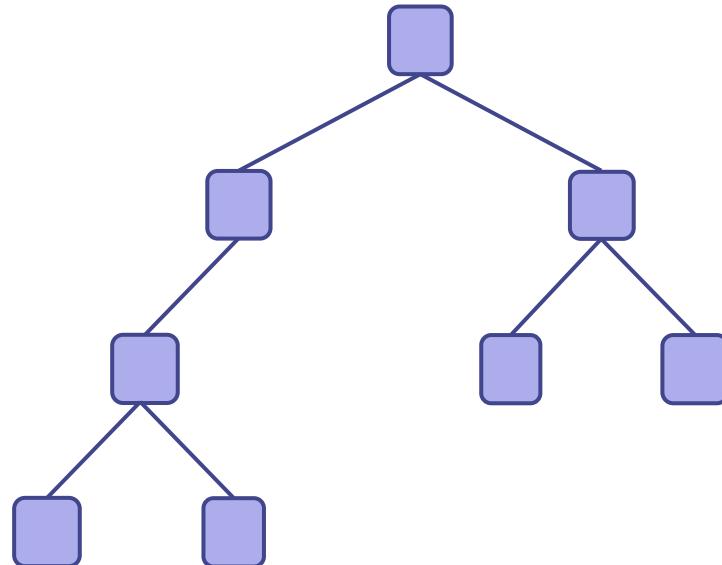
**Perfecte!**

# Complet

- Un arbre binari és quasi-complet (o complet) si:
  - Cada nivell està completament ple, excloent el nivell més baix
  - Tots estan el màxim a l'esquerra possible



**Quasi-complet!**



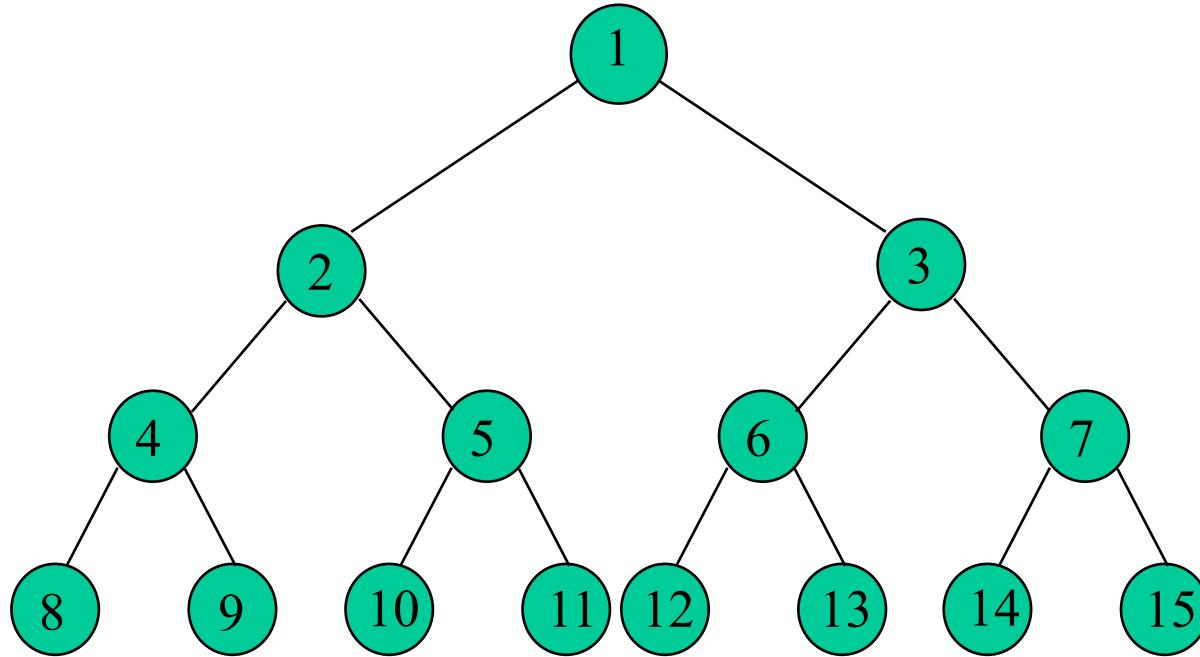
**No complet ni quasi-complet**



## 4.3 Recorreguts en arbres binaris

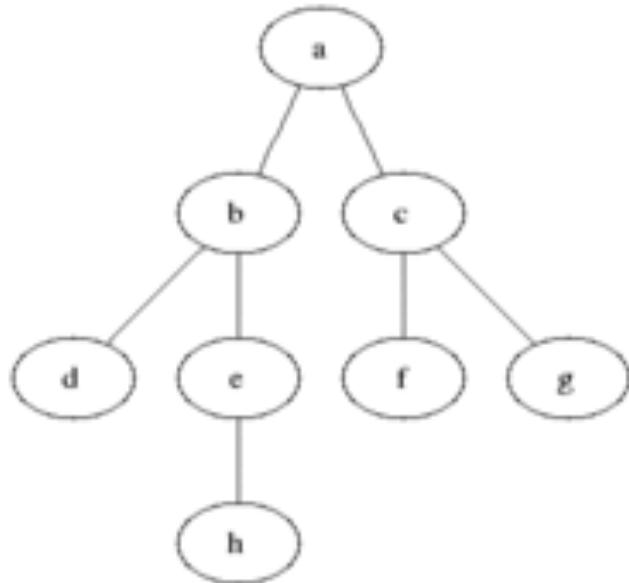
# Com recórrer un arbre binari?

- Quines estratègies es poden seguir?
  - Top to Bottom? Left to Right?

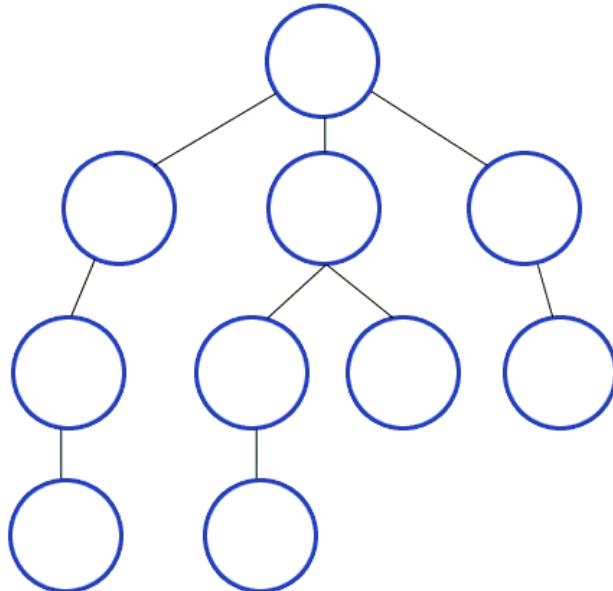


# Recorregut Amplada vs. Profunditat

BFT



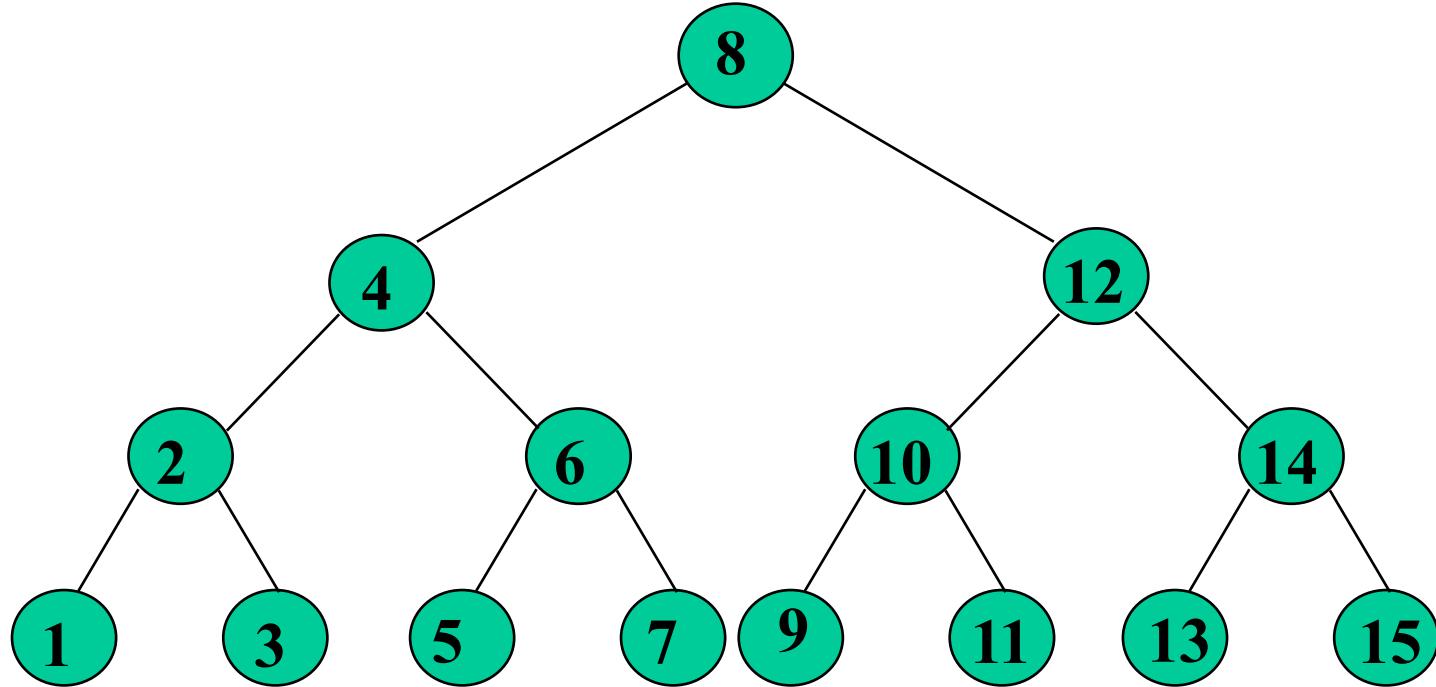
DFT



BFT gif: <http://brochure.omniscryptum.com/info-lap/0537275001445433253>

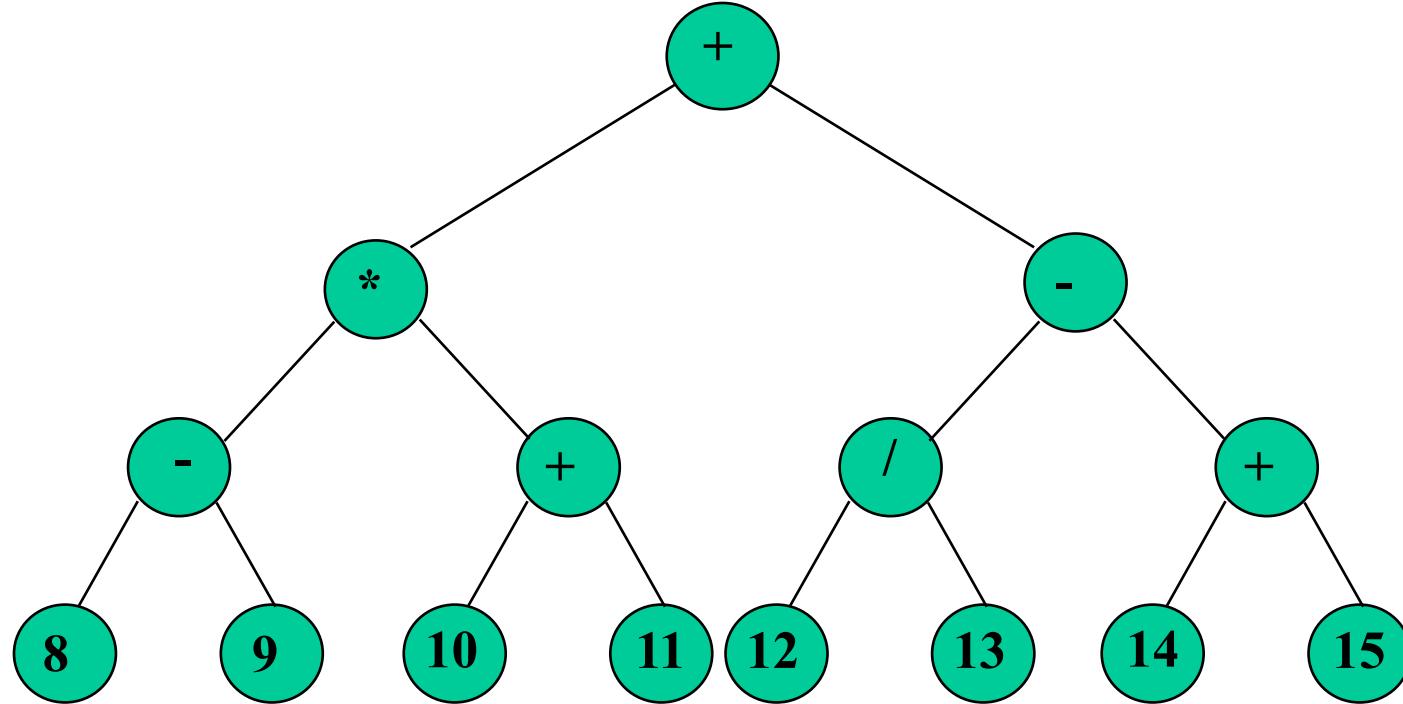
DFT gif: <https://upload.wikimedia.org/wikipedia/commons/7/7f/Depth-First-Search.gif>

# Com recórrer un arbre binari?



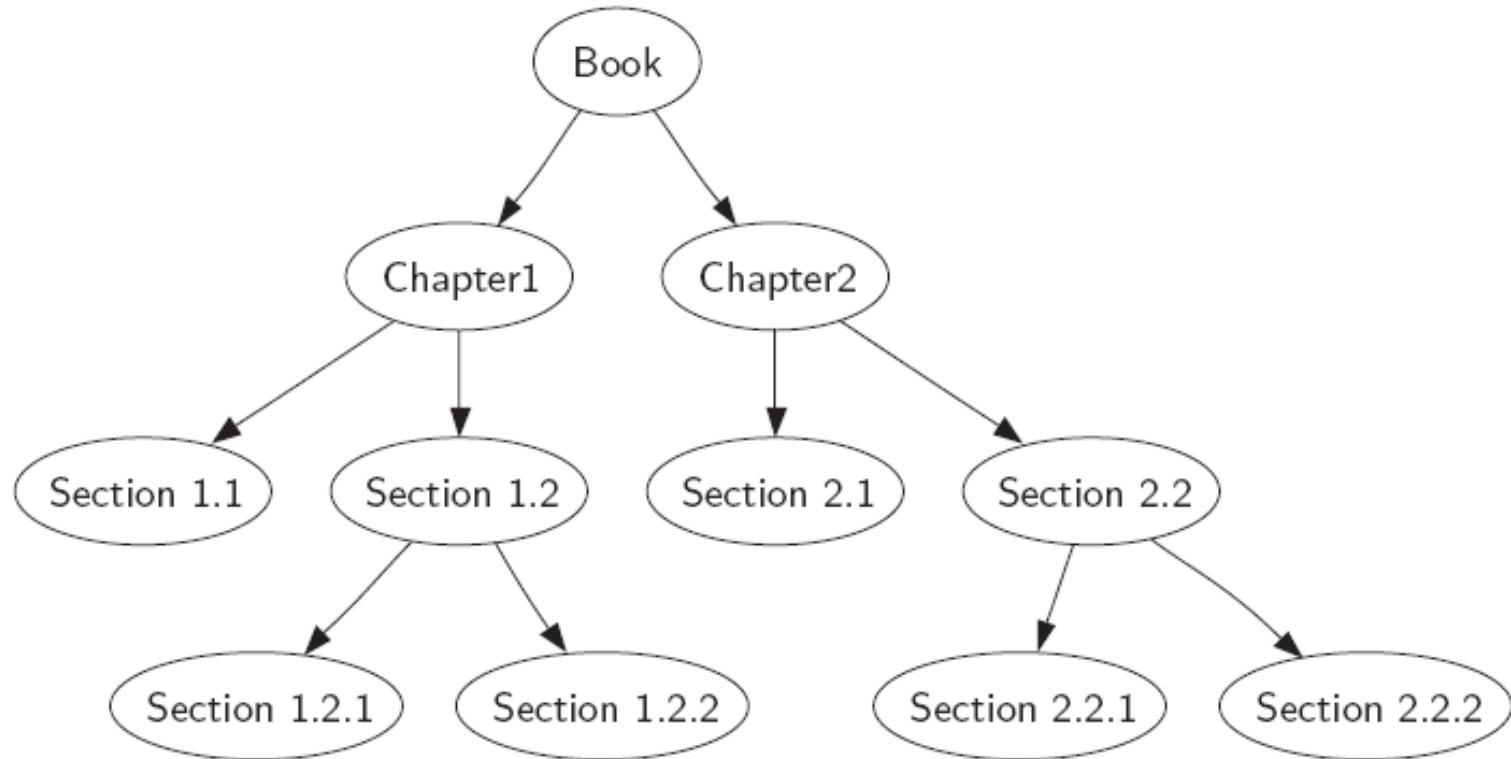
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15

# Com recórrer un arbre binari?



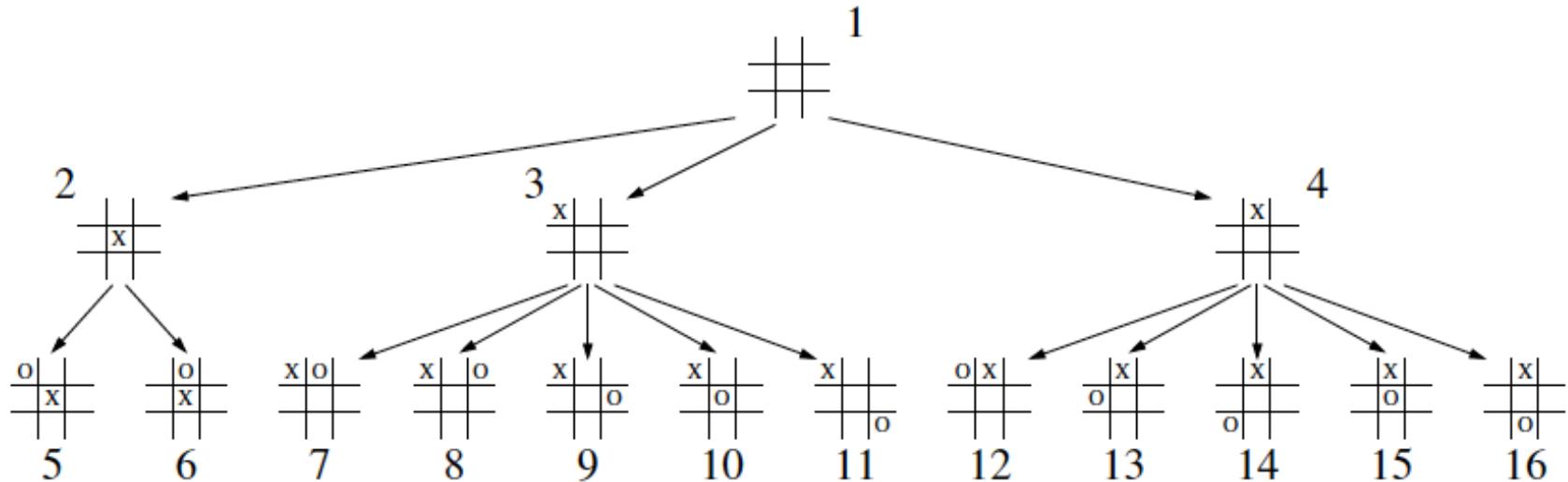
$$((8 - 9) * (10 + 11)) + ((12/13) - (14+15))$$

# Com recórrer un arbre binari?



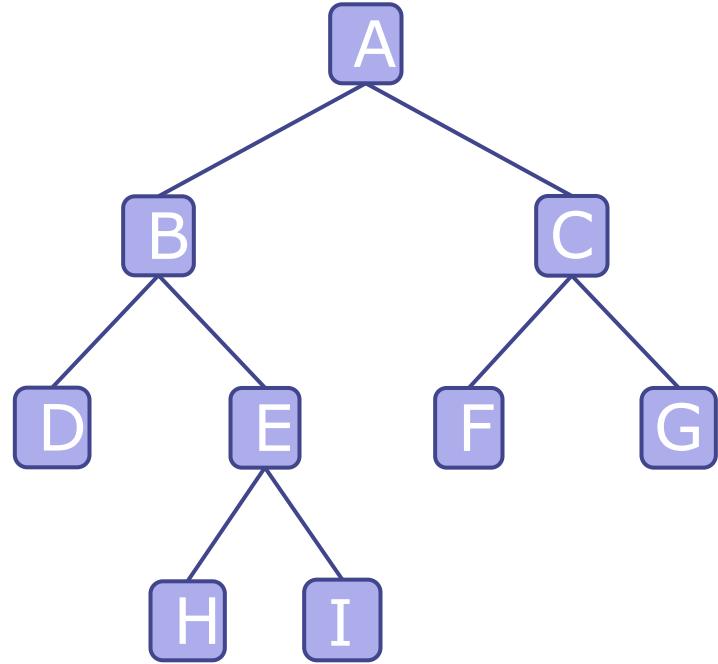
# Recorregut en amplada

- Recorregut **breadth-first**
  - El recorregut en amplada és molt comú en el desenvolupament de jocs. Un arbre de joc representa els possibles moviments en el joc que pot fer un jugador o un ordinador, essent l'arrel de l'arbre el moviment inicial.



# Recorregut en amplada: Breadth-first

- Es comença des del node arrel, visitem tots els seus fills, després visitem tots els seus néts, després els besnéts, després ....



**Recorregut  
Breadth-first:**  
A,B,C,D,E,F,G,H,I



# Recorregut en amplada: Breadth-first

**Estratègia general:** Utilitzar una **cua** (Queue) per fer el seguiment de l'ordre dels nodes

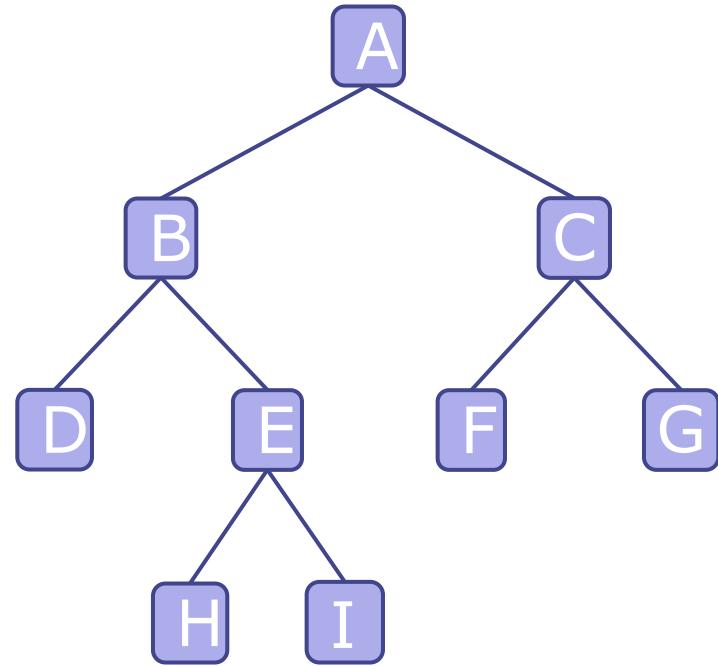
**Pseudo-codi:**

```
Algorithm bft(root):
    //Input: root node of tree
    //Output: None
    Q = new Queue()
    enqueue root
    while Q is not empty:
        node = Q.dequeue()
        visit(node)
        enqueue node's left & right children
```

La cua garanteix que tots els nodes d'un mateix nivell seran visitats abans que qualsevol dels seus fills

# Recorregut en profunditat

- Es comença des del node arrel, s'explora cada branca el més profundament possible abans de realitzar backtracking
- Això pot generar ordenacions molt diferents dependent de quina branca es visita primer
  - Normalment es decideix visitar d'esquerra a dreta



**Recorregut en profunditat:**

A,C,G,F,B,E,I,H,D

o

A,B,D,E,H,I,C,F,G



# Recorregut en profunditat

- **Estratègia General:** Utilitzar una **pila** per fer un seguiment de l'ordre dels nodes.

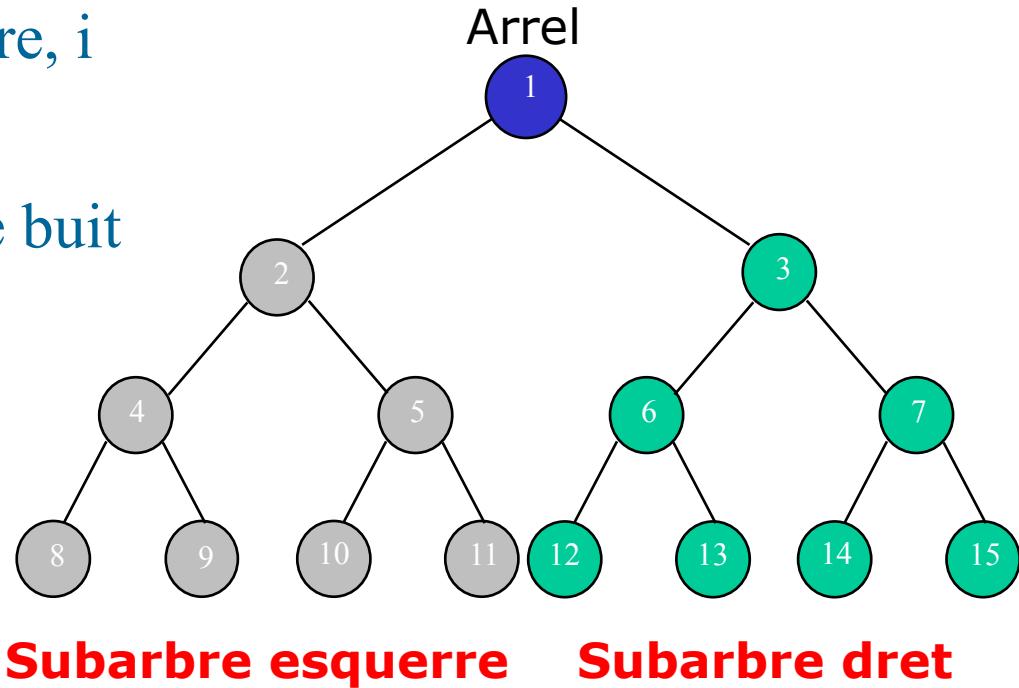
Pseudo-codi:

```
Algorithm dft(root):
    //Input: root node of tree
    //Output: none
    S = new Stack()
    push root onto S
    while S is not empty:
        node = S.pop()
        visit(node)
        push node's right and left children
```

- La pila garanteix que s'explora cada branca fins arribar a la fulla abans d'explorar una altra

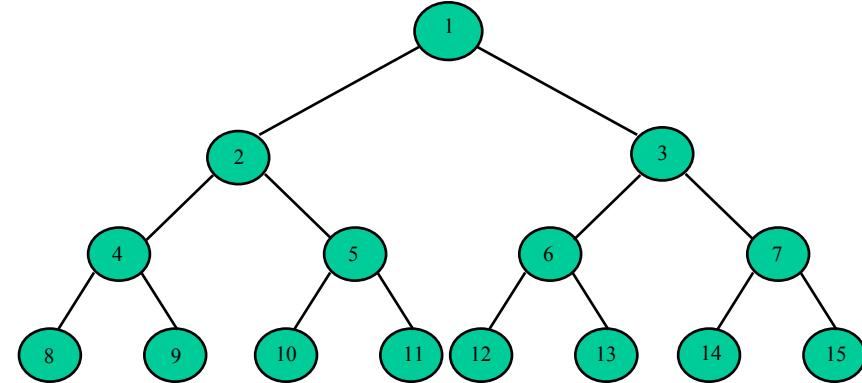
# Recursivitat en arbres binaris

- Els mètodes de **recorregut en profunditat** són especialment òptims usant la **recursivitat**
- **Plantejament:**
  - **Cas general:** s'aplica l'operació recursiva sobre:
    - el subarbre esquerre, i
    - el subarbre dret
  - **Cas particular:** arbre buit



# Com es pot recórrer un arbre binari en profunditat?

- Si denominem amb:
  - L: moviment a l'esquerra
  - V: “visitar” el node
  - R: moviment a la dreta
- Tenim sis combinacions possibles de recorregut:
  - **LVR, LRV, VLR, VRL, RVL, RLV**
- Si optem per realitzar primer el moviment a l'esquerra, tenim les tres primeres possibilitats
  - LVR l'anomenarem **inordre**,
  - VLR l'anomenarem **preordre**, i
  - LRV l'anomenarem **postordre**



# Recorreguts vistos anteriorment

**Algorithm *binaryPreorder*(T, p)**

*visit(p)*

**if** p is an internal node **then**

*binaryPreorder* (T, p.left())

*binaryPreorder* (T, p.right())

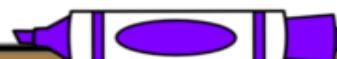
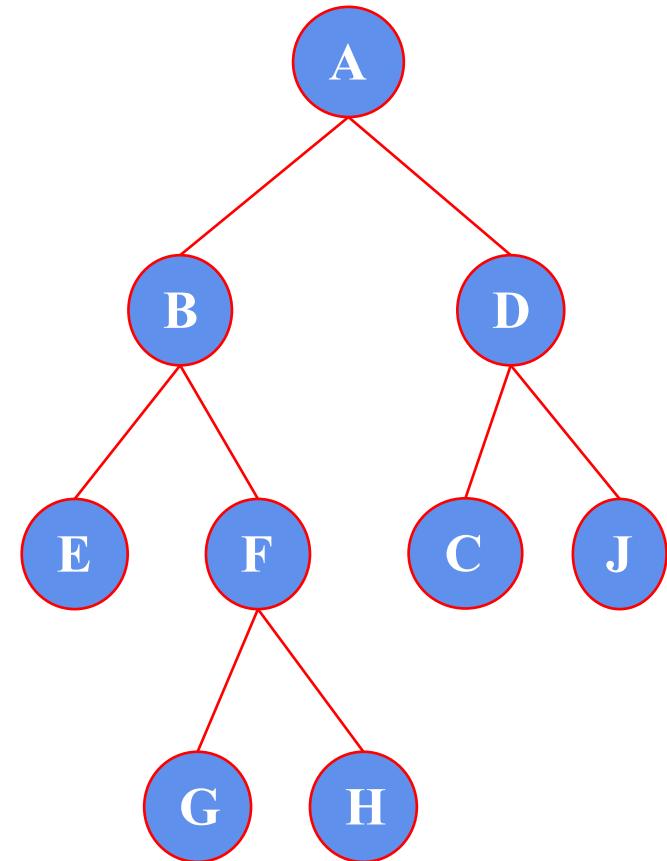
**Algorithm *binaryPostorder*(v)**

**if** p is an internal node **then**

*binaryPostorder* (T, p.left())

*binaryPostorder* (T, p.right())

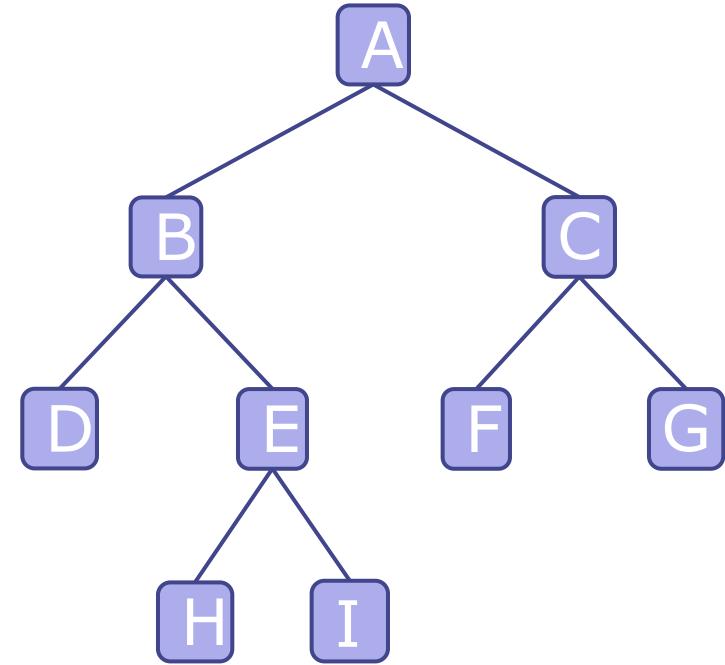
*visit(p)*



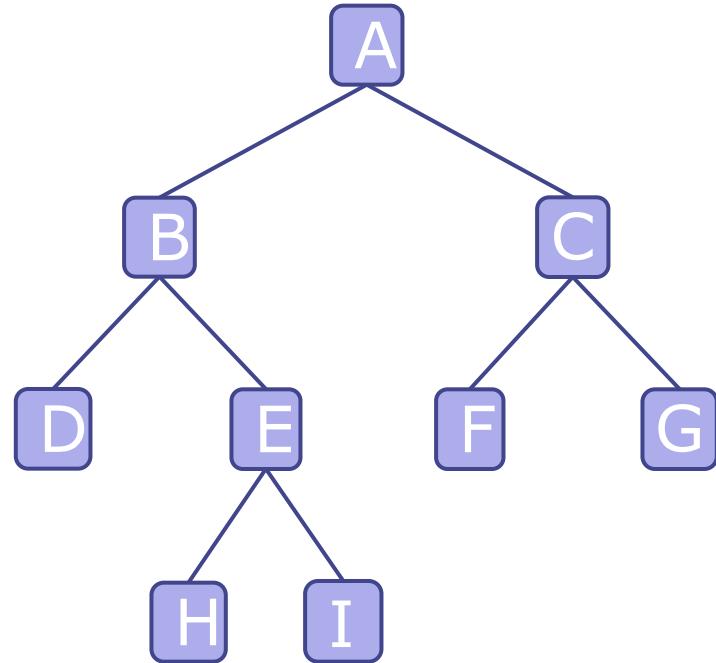
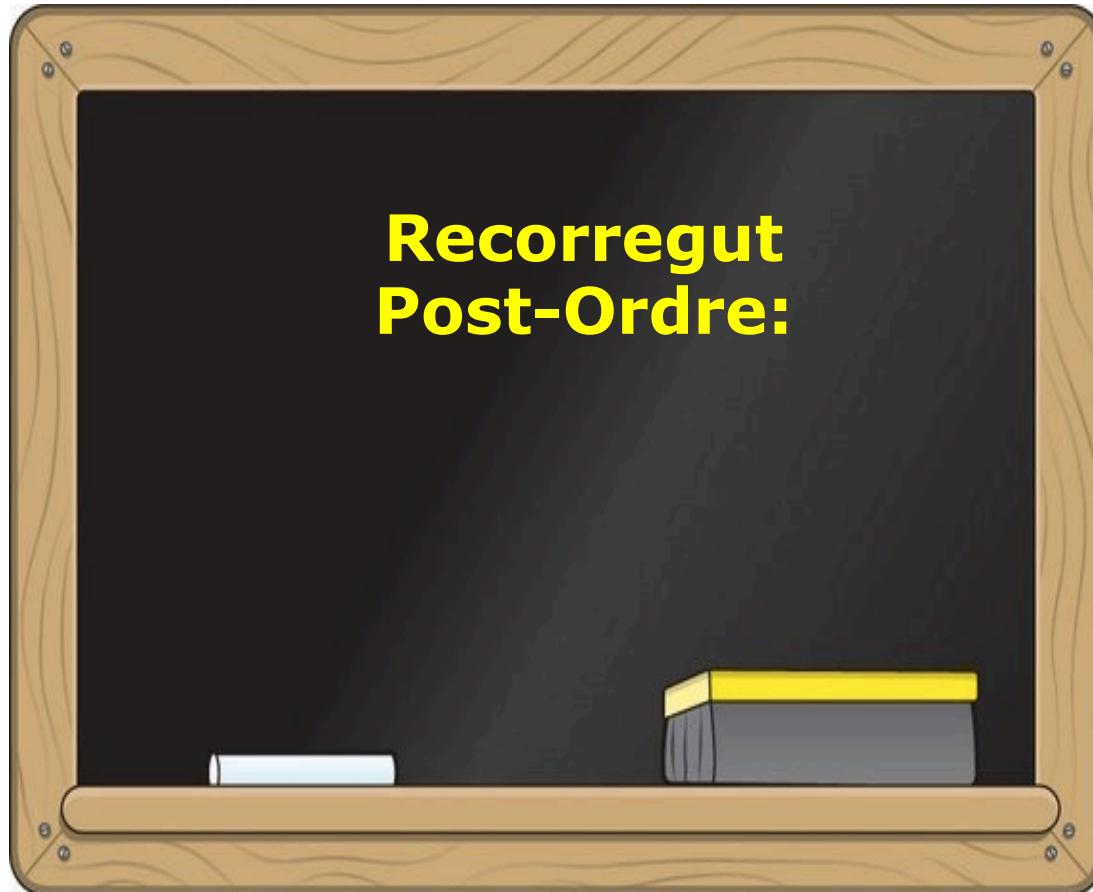
# Recorregut en Pre-Ordre



**Recorregut  
Pre-Ordre:**



# Recorregut en Post-ordre

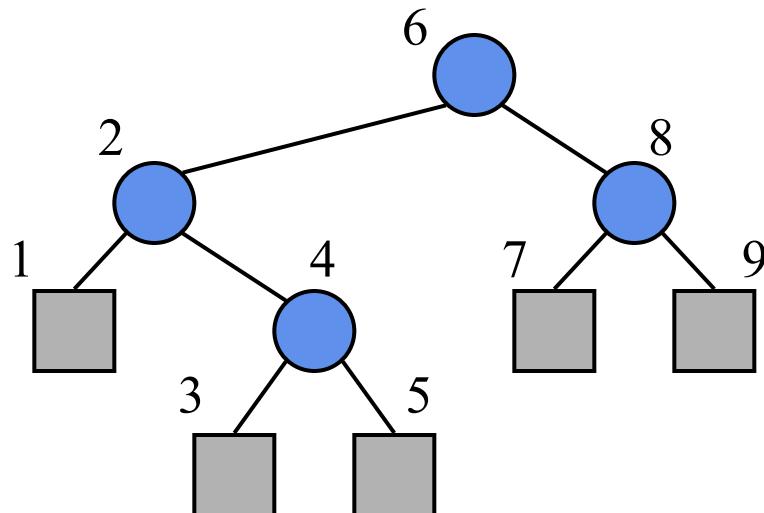


# Recorregut en inordre

- En un recorregut en inordre un node es visita quan ja s'ha visitat el seu subarbre esquerra i abans del seu subarbre dret
- Aplicació: pintar un arbre binari

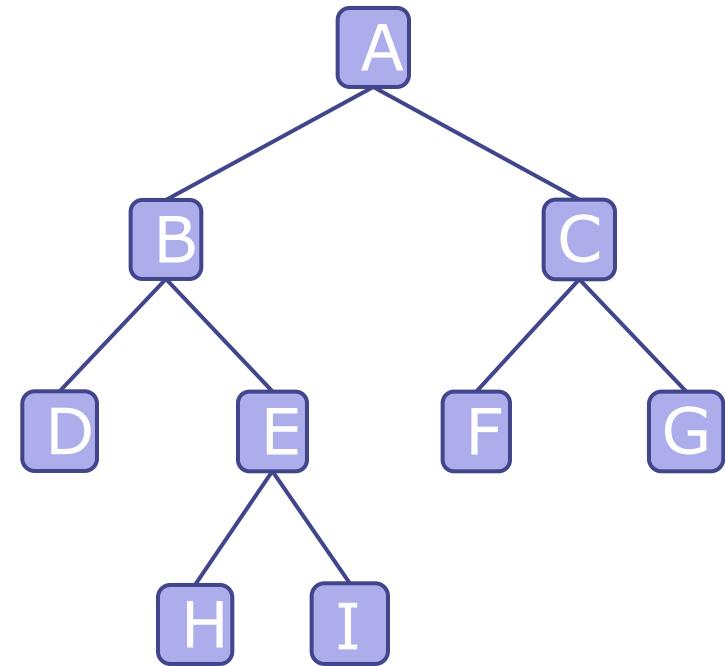
**Algorithm *inOrder* (T, p)**

```
if p is an internal node then
    inOrder(T, p.left())
    visit(p)
    if p is an internal node then
        inOrder(T, p.right())
```



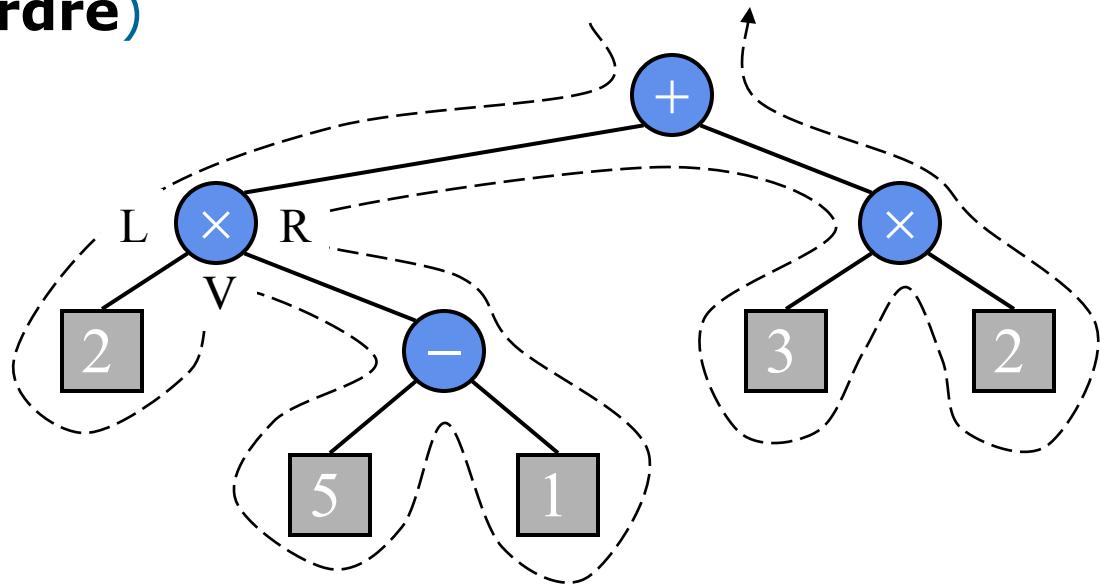
# Recorregut en In-ordre

**Recorregut  
In-Ordre:**



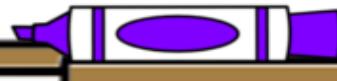
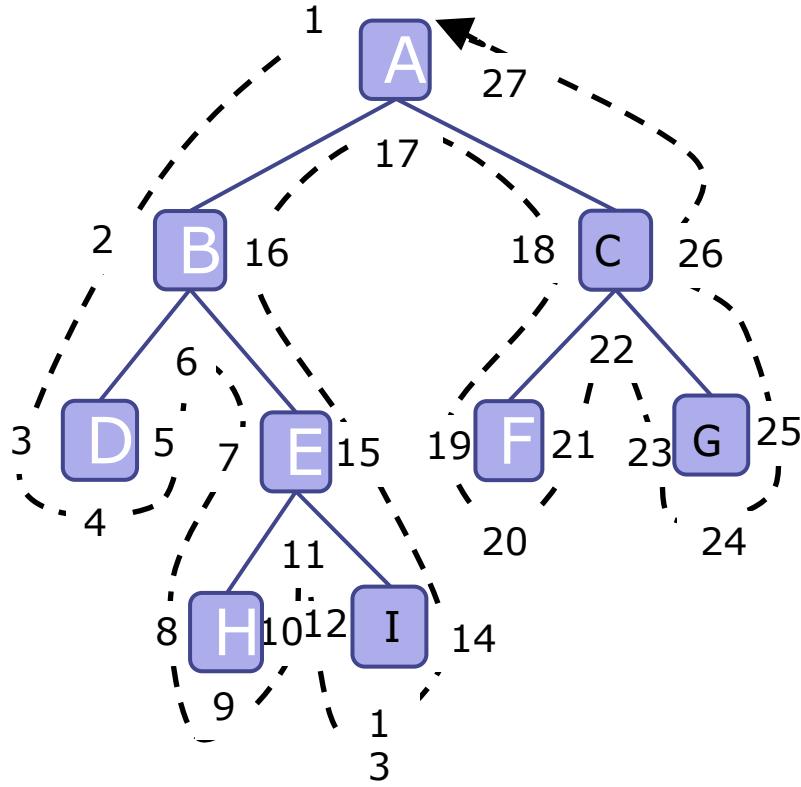
# Recorregut d'Euler

- Recorregut genèric d'un arbre binari
- Inclou els casos especials de recorreguts preordre, inordre i postordre
- Cada node es visita tres cops:
  - per l'esquerra (**preordre**)
  - per sota (**inordre**)
  - per la dreta (**postordre**)



# Recorregut d'Euler: Pseudo-codi

```
Algorithm eulerTour(node):
    // Input: root node of tree
    // Output: None
```





# Quan usar cada tipus de recorregut?

- Com sabem quin és el millor mètode per recórrer un arbre?
- Algunes vegades no importa, però normalment hi ha un mètode que ens permet solucionar el problema de forma molt més senzilla i eficient



# Problema 1

- Indicar en cada node quants descendents hi ha
- Millor recorregut: **post-ordre**
  - És fàcil calcular el nombre de descendents d'un node si ja sabem quants descendents tenen els seus fills
  - El post-ordre visita els nodes fills abans que el node pare, això és exactament el que volem

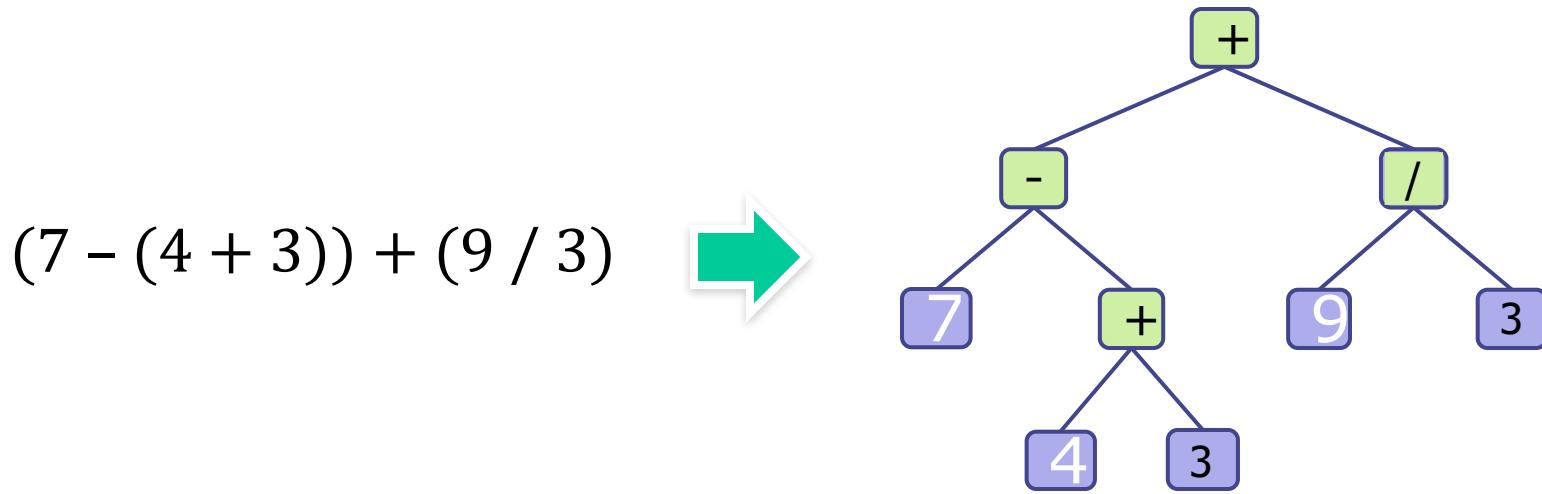


# Problema 2

- Donada l'arrel d'un arbre, determinar si un arbre és perfecte.
- Millor recorregut: **breadth-first**
  - La millor solució es troba explorant l'arbre nivell a nivell
  - Podem fer un seguiment del nivell actual i comptar el nombre de nodes que té
  - Cada nivell tindrà el doble de nodes que el nivell anterior

# Problema 3

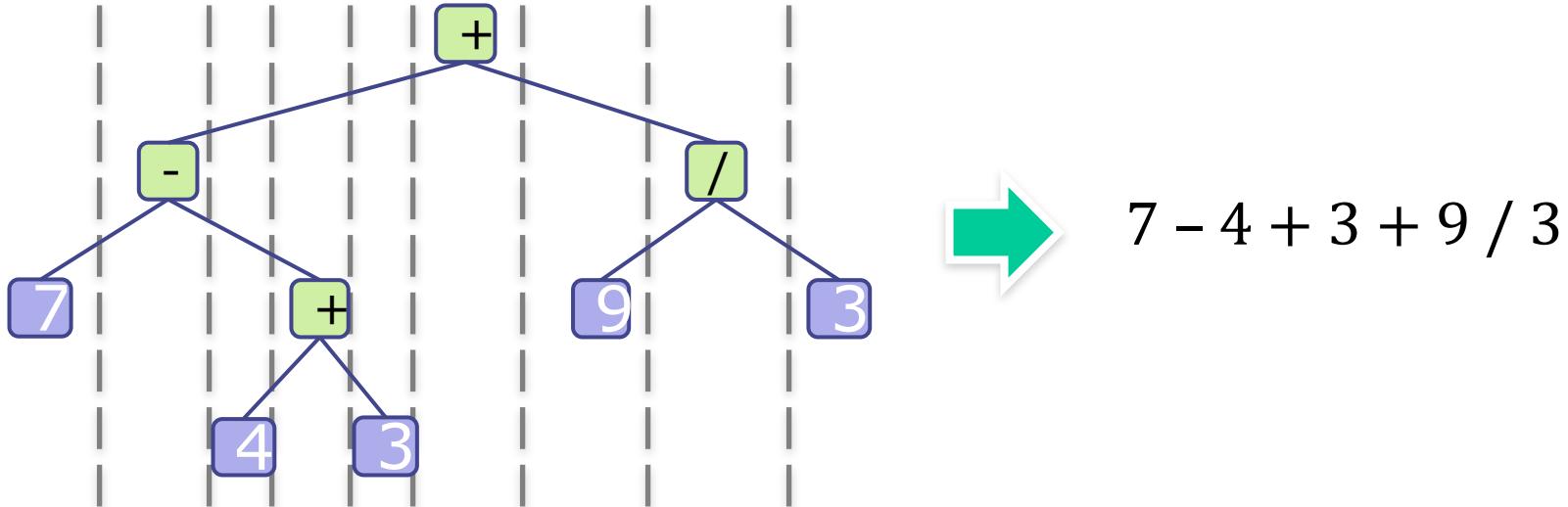
- Donat l'arbre que representa una expressió aritmètica, avaluar el seu resultat:



- Millor recorregut: **post-ordre**
  - Per avaluar l'expressió aritmètica, primer cal avaluar les sub-expressions de cada fill

# Problema 4

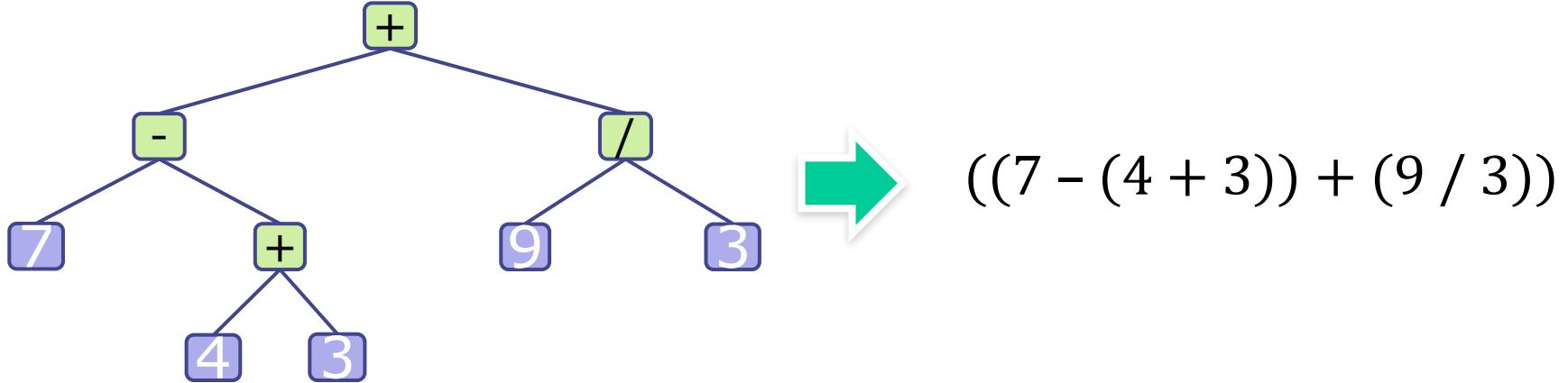
- Donada una expressió aritmètica, imprimir el seu resultat sense parèntesis



- Millor recorregut: **in-ordre**
  - El recorregut inordre ens dona els nodes d'esquerra a dreta

# Problema 5

- Donat un arbre que representa una expressió aritmètica, imprimir l'expressió amb parèntesis



- Millor recorregut: **Recorregut d'Euler**
  - Si el node és un node intern (un operador):
    - Per al pre-ordre imprimir "("
    - Per al in-ordre imprimir l'operador
    - Per al post-ordre imprimir ")"
  - Si el node és una fulla (un número)
    - No fer res per al pre-ordre ni post-ordre
    - Per al in-ordre, imprimir el número



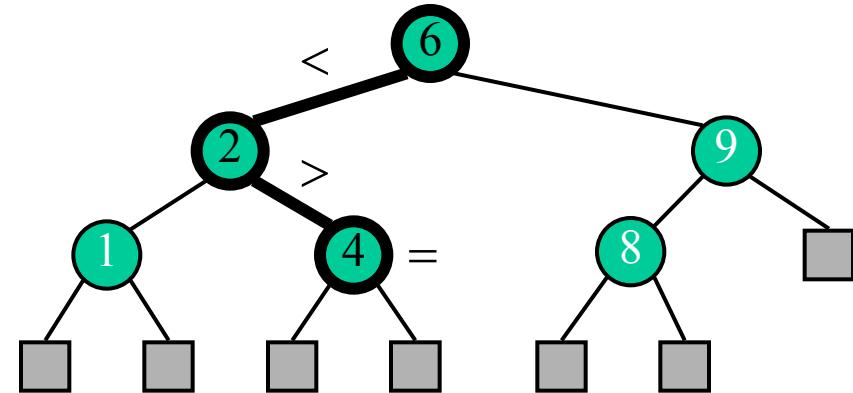
# Exercicis

**Exercici 1.** Comptar el número de nodes d'un arbre

**Exercici 2.** Comparar dos arbres, retorna CERT si són iguals, altrament retorna FALS

**Exercici 3.** Copiar recursivament un arbre

## 4.4 Arbres binaris de cerca (arbres de cerca binària)



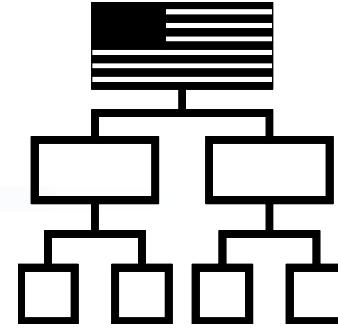
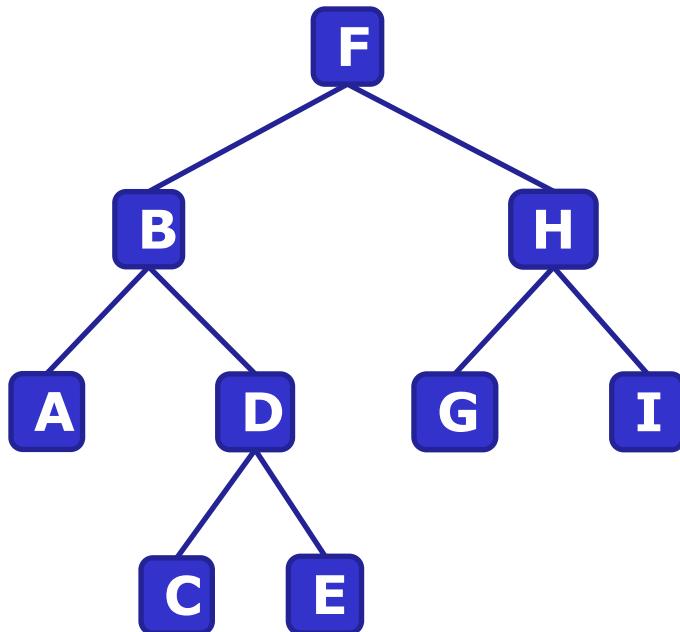
# Arbre binari de cerca

- Un **arbre binari de cerca** (BST) és un arbre binari amb unes propietats especials.

Per cada node:

- Tots els descendents del seu subarbre esquerra tenen un valor menor
- Tots els descendents del seu subarbre dret tenen un valor major

- El **recorregut en inodre** d'un arbre de cerca binària visitarà els valors en ordre creixent

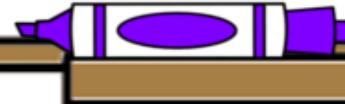




# Arbre BST

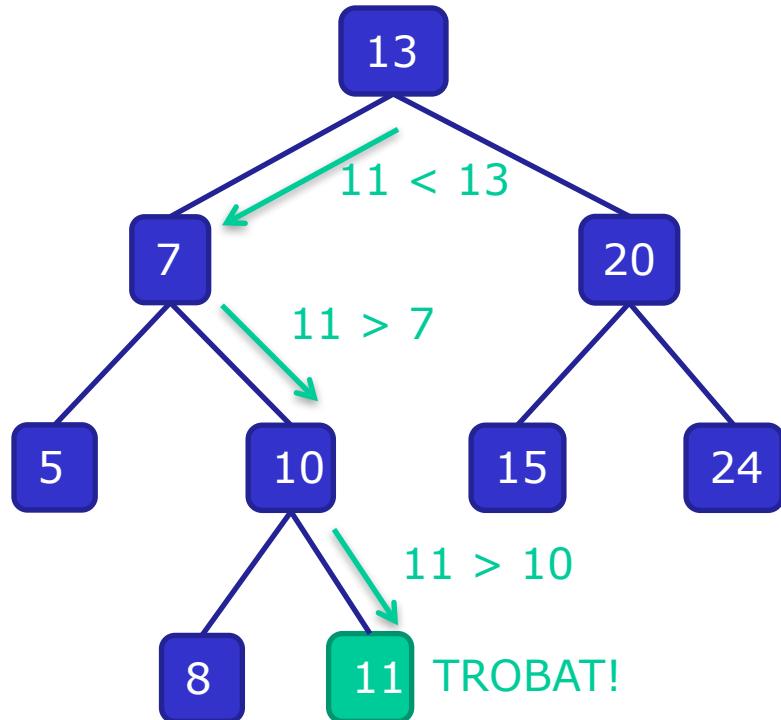
Volem crear un ABB on cada node es correspongui amb una paraula de la següent frase:

“In this example we see how BST should be created”



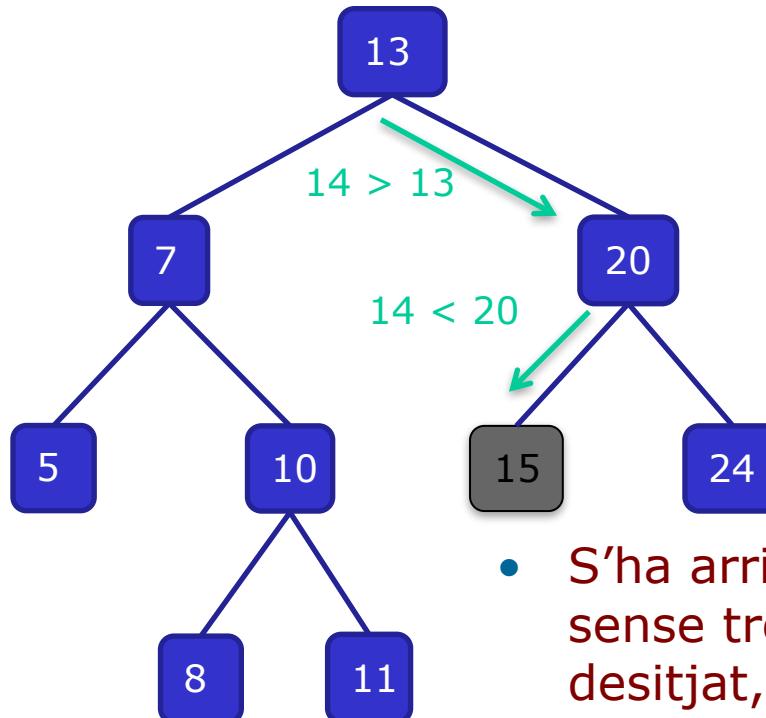
# Cerca en un BST

- Per cercar un valor  $k$ , s'ha de traçar un camí cap a baix començant des de l'arrel
- El següent node a visitar depèn de la comparativa de  $k$  amb el valor del node actual
- **Exemple:** volem trobar el valor 11 en l'arbre



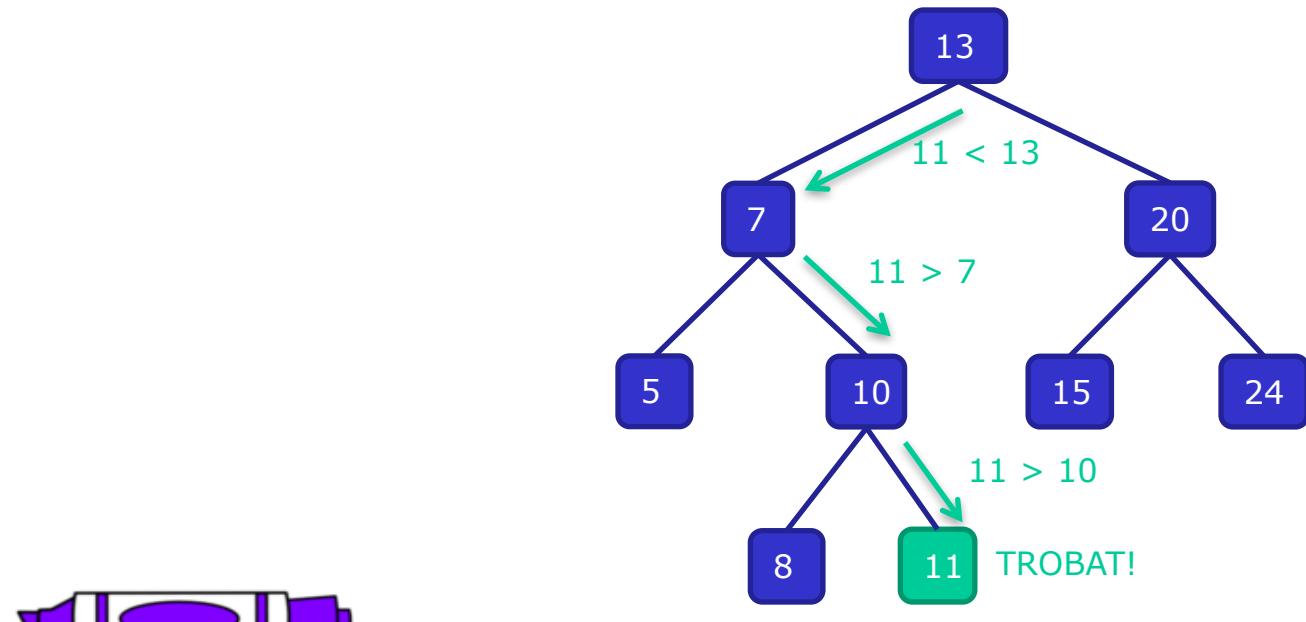
# Cerca en un BST

- Que ocorre si l'element no es troba en l'arbre?
- Imagina que s'està cercant el número 14



- S'ha arribat a una fulla sense trobar el número desitjat, per tant el número no es troba en l'arbre

# Cerca en BST: Pseudocodi

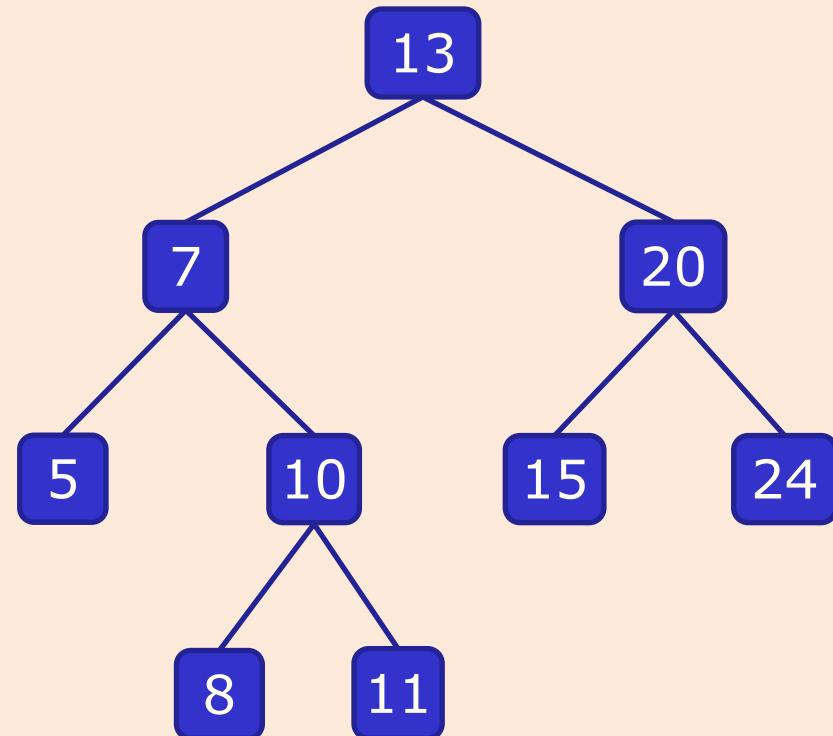


# Afegir en un BST

- Per afegir en un BST s'utilitza la mateixa estratègia que en la cerca
- Un nou ítem sempre s'afegeix com una nova fulla

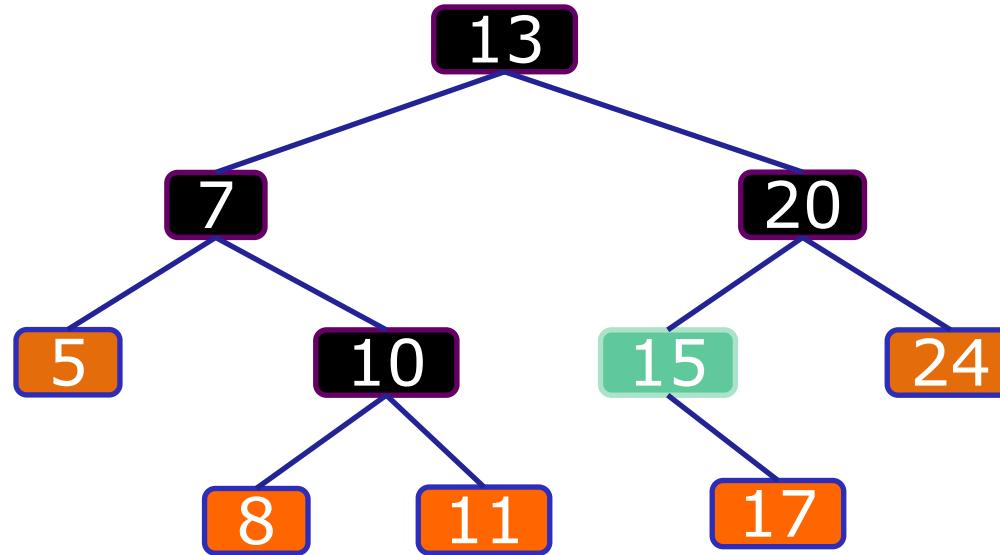
Abre inici:

-> Afegir element 17



# Eliminar en un BST

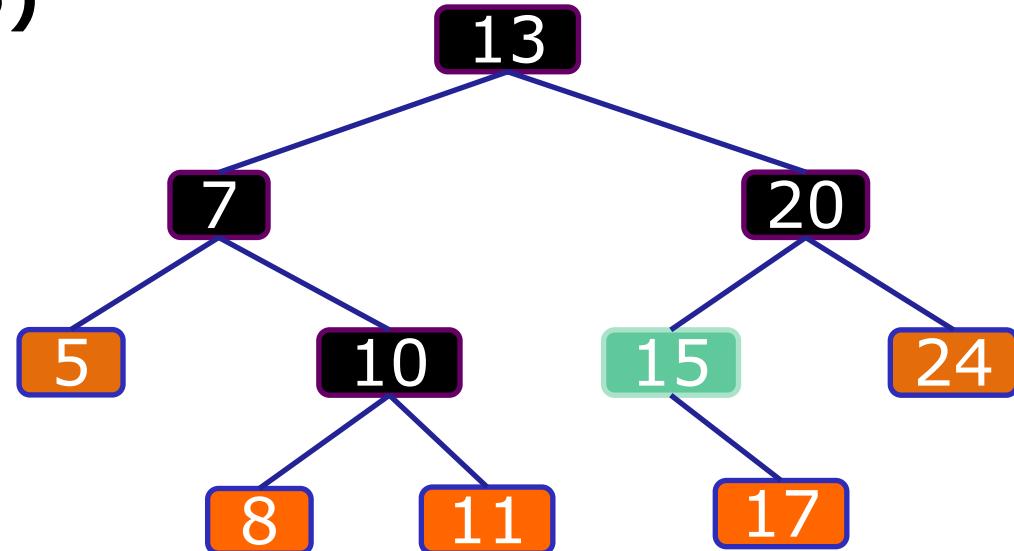
- Eliminar elements en un ABB és un algorisme més complexe
- S'han de considerar 3 casos:
  - CAS 1. Eliminar una **fulla**. S'elimina el node corresponent
  - CAS 2. Eliminar un **node intern amb un fill**
  - CAS 3. Eliminar un **node intern amb dos fills**



# Eliminar en un BST: Cas 2

## Cas 2: Eliminar un node intern amb un fill

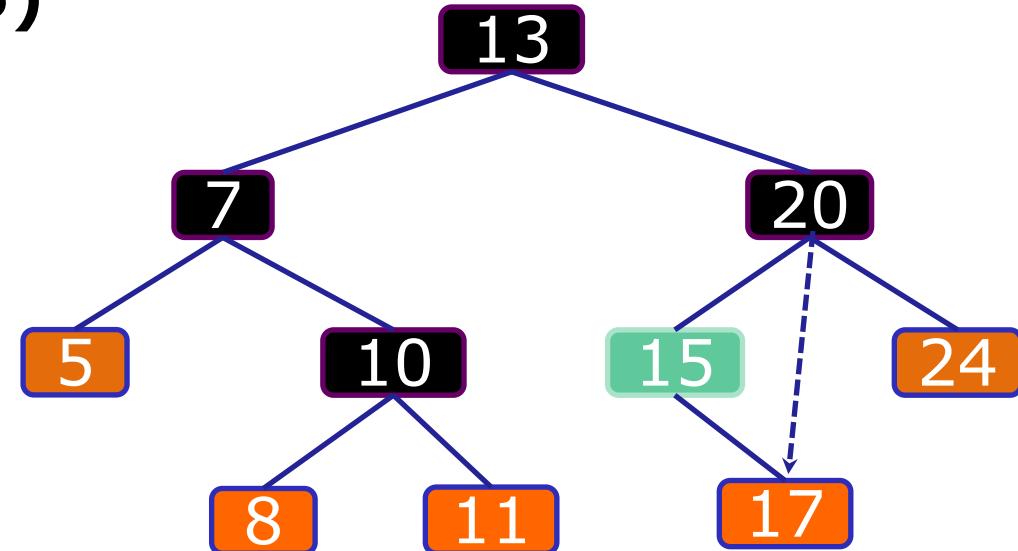
- Estratègia General:
  - S'elimina l'element corresponent connectant el node del pare amb el node del fill del node a eliminar
- Exemple: **remove(15)**



# Eliminar en un BST: Cas 2

## Cas 2: Eliminar un node intern amb un fill

- Estratègia General:
  - S'elimina l'element corresponent connectant el node del pare amb el node del fill del node a eliminar
- Exemple: **remove(15)**
  1. S'assigna com a fill esquerra del node (20) pare la referència del fill del node 15



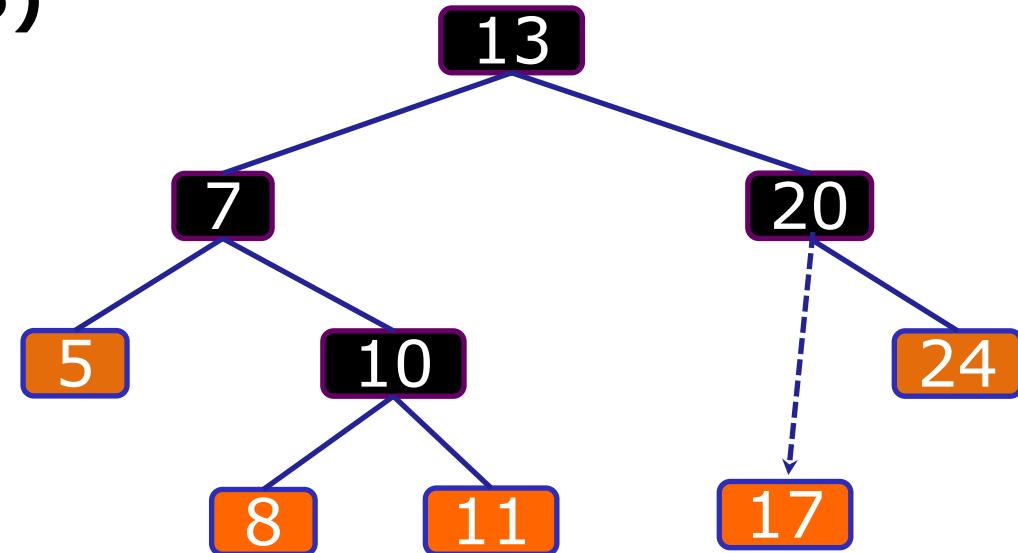
# Eliminar en un BST: Cas 2

## Cas 2: Eliminar un node intern amb un fill

- Estratègia General:
  - S'elimina l'element corresponent connectant el node del pare amb el node del fill del node a eliminar

- Exemple: **remove(15)**

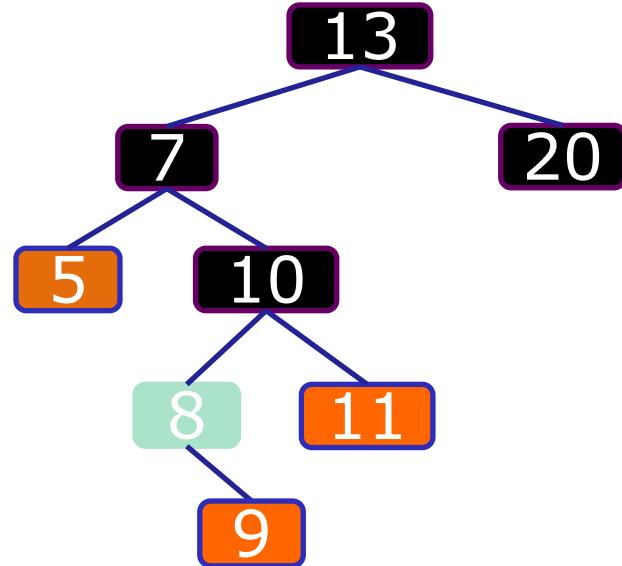
1. S'assigna com a fill esquerra del node (20) pare la referència del fill del node 15
2. S'elimina el node 15



# Eliminar en un BST: Cas 3

## Cas 3: Eliminar un node intern amb dos fills

- Estratègia General:
  - Canviar les dades del node a eliminar per les dades del successor del node
    - Es canvia pel node amb valor menor de tot el subarbre dret
  - Eliminar el node successor



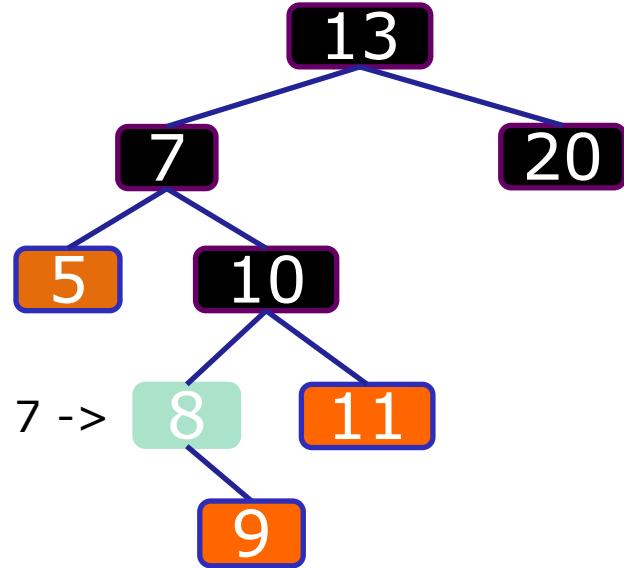
# Eliminar en un BST: Cas 3

## Cas 3: Eliminar un node intern amb dos fills

- Exemple: remove(7)
  - Primer busquem el successor del node
    - El successor del node 7 és el node 8

```
void successor (Position node):  
    // Input: node - the node for  
    // which to find the successor  
    Position curr = node.right  
    while (curr.left != null):  
        curr = curr.left  
    return curr
```

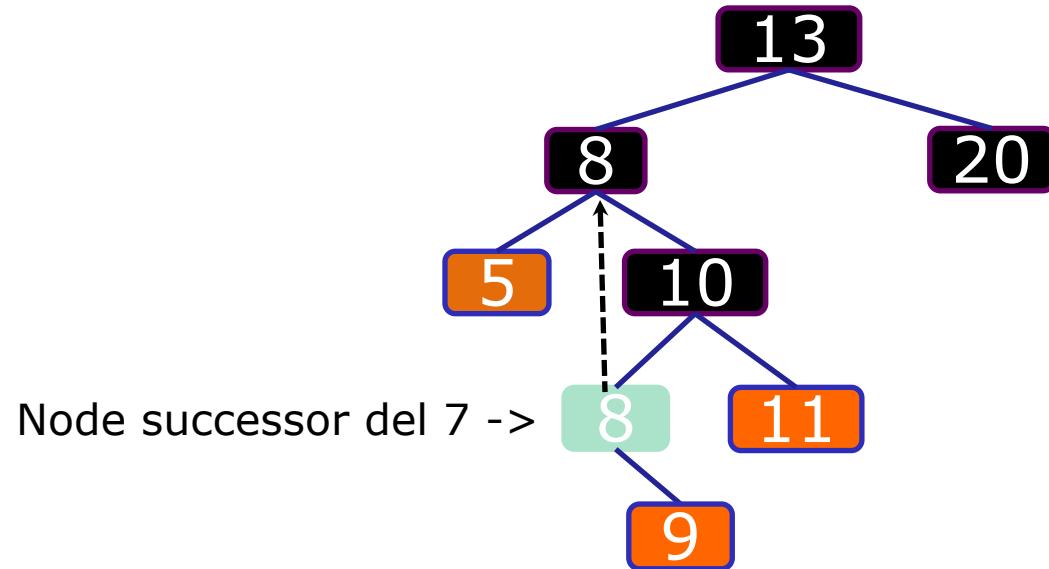
Node successor del 7 ->



# Eliminar en un BST: Cas 3

## Cas 3: Eliminar un node intern amb dos fills

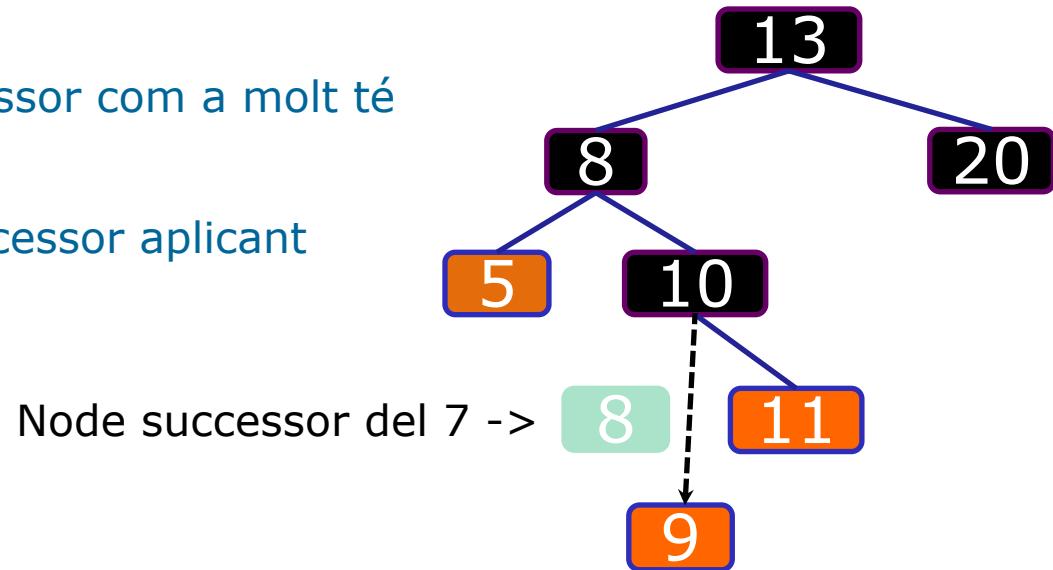
- Exemple: remove(7)
  - Segon, es reemplaça les dades del node a eliminar per les dades del successor



# Eliminar en un BST: Cas 3

## Cas 3: Eliminar un node intern amb dos fills

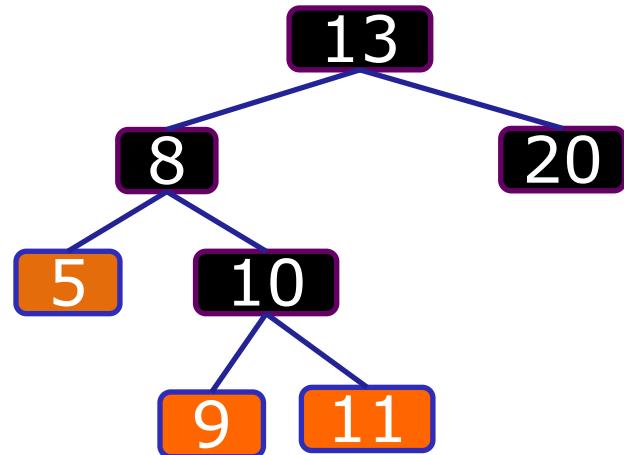
- Exemple: remove(7)
  - Per últim, s'ha d'eliminar el node successor
- IMPORTANT!! Tenim la seguretat que el node successor no té fill esquerra
  - Per tant el node successor com a molt té un fill dret
  - Podem eliminar el successor aplicant el cas 1 o el cas 2



# Eliminar en un BST: Cas 3

## Cas 3: Eliminar un node intern amb dos fills

- Exemple: remove(7)
  - Per últim, s'ha d'eliminar el node successor
- IMPORTANT!! Tenim la seguretat que el node successor no té fill esquerra
  - Per tant el node successor com a molt té un fill dret
  - Podem eliminar el successor aplicant el cas 1 o el cas 2





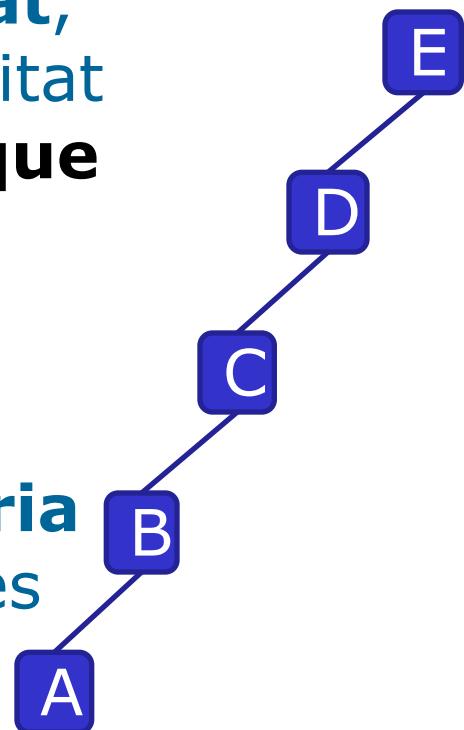
# Eliminar en un BST

**Exercici:** Escriviu un algorisme en pseudocodi per eliminar un node d'un arbre BST

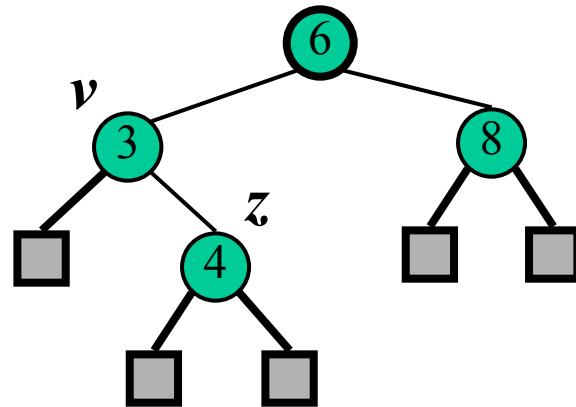
**void remove (Position node)**

# Rendiment en un BST

- Depèn de l'alçada de l'arbre. En el **pitjor cas**, tindrem un arbre amb **l'alçada igual que el nombre de nodes**
- Si l'arbre està **perfectament balancejat**, l'alçada és  $\log_2 n$ , el que fa que la coplexitat de les funcions sigui  $O(\log_2 n)$ . **MENYS que lineal!!**
- En el cas d'un **arbre totalment desbalancejat**, l'arbre de cerca binària **és una Ordered Linked List**, i les seves funcions tenen una complexitat  $O(n)$

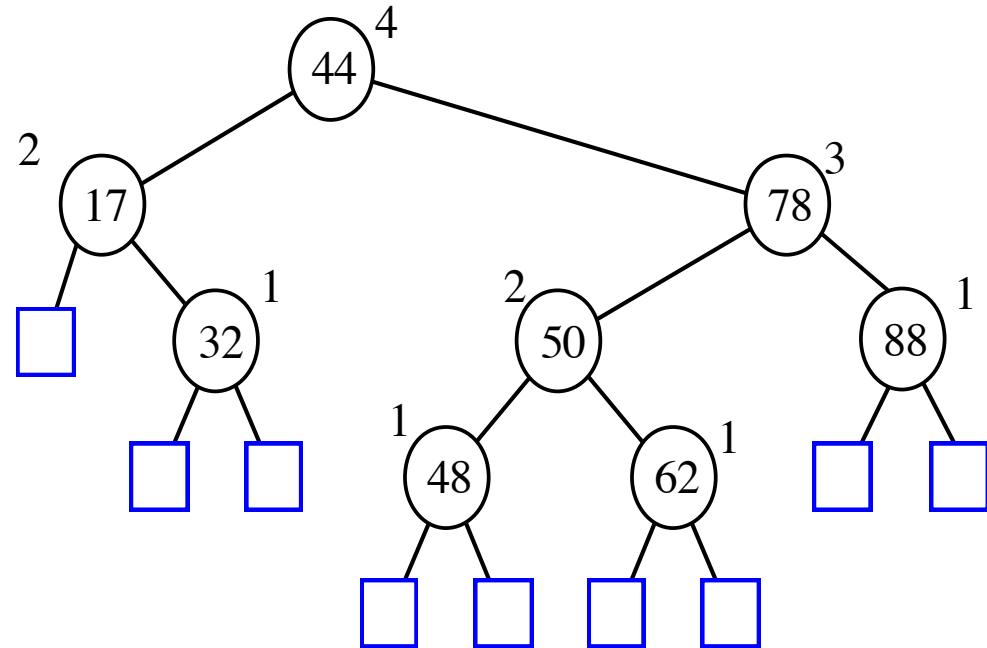


## 4.5 Arbres AVL (Adelson - Velskii i Landis)



# Arbres AVL

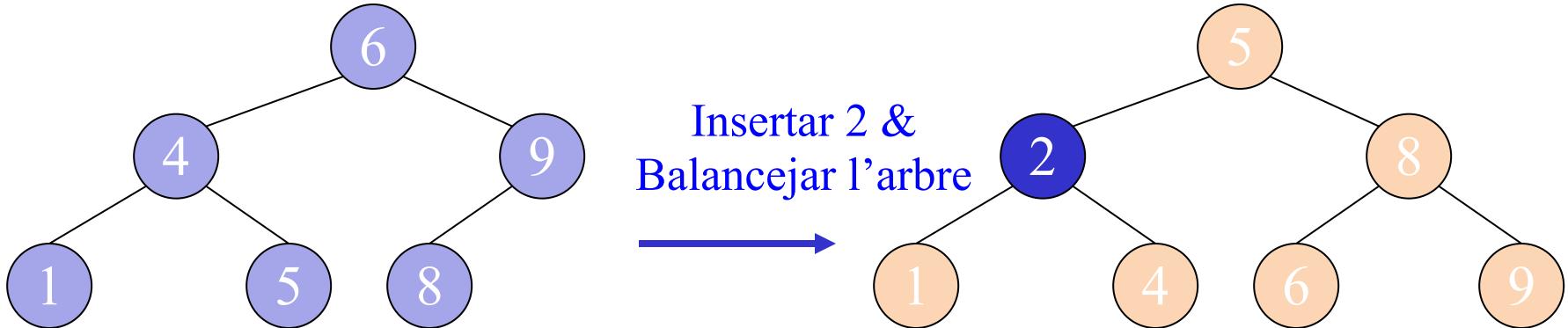
- Els arbres AVL són arbres equilibrats (perfectament balancejats)
- **Un arbre AVL** és un **arbre de cerca binària** tal que cada node intern  $v$  de  $T$ , compleix que, les alçades dels fills de  $v$  poden diferir com a molt de 1



Exemple d'un arbre AVL on les alçades estan a cada node

# Arbres perfectament balancejats

- Es vol un arbre complet després de cada inserció
- Aquesta operació té una alta complexitat
  - Per exemple, insertar 2 en l'arbre esquerra i després reconstuir-lo a un arbre complet





# Arbre balancejat o AVL

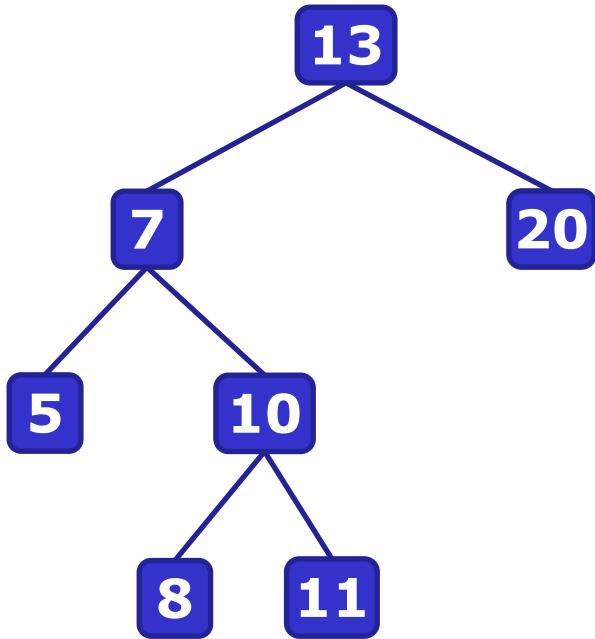
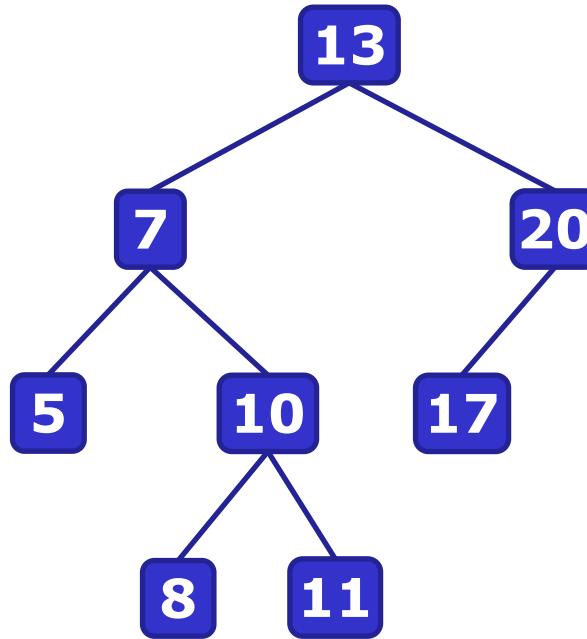
- Els arbres AVL també s'anomenen ABB d'alçada balancejada

## Factor de Balaç d'un node:

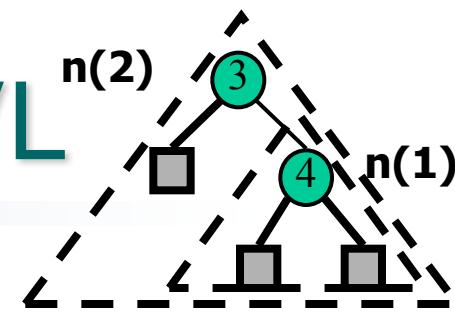
Alçada(subarbre esquerra) – Alçada(subarbre dret)

- Un arbre AVL té el factor de balanç calculat en cada node
  - L'alçada del subarbre dret i esquerra només pot diferir en 1
  - Es guarda l'alçada en cada node

# Són arbres AVL?

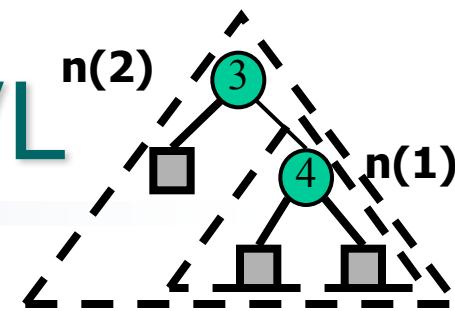


# Alçada d'un arbre AVL



- **Proposició:** L'alçada d'un arbre AVL que guarda  $n$  nodes és  $O(\log n)$
- **Justificació:** La forma més fàcil de demostrar aquest problema és trobar  $n(h)$ : el nombre mínim de nodes interns d'un arbre AVL d'alçada  $h$
- Facilment es pot veure que  $n(1) = 1$  i  $n(2) = 2$
- Per  $n > 2$ , un arbre AVL d'alçada  $h$  conté un node arrel, un subarbre AVL d'alçada  $n-1$  i un altre d'alçada  $n-2$
- És a dir,  $n(h) = 1 + n(h-1) + n(h-2)$

# Alçada d'un arbre AVL

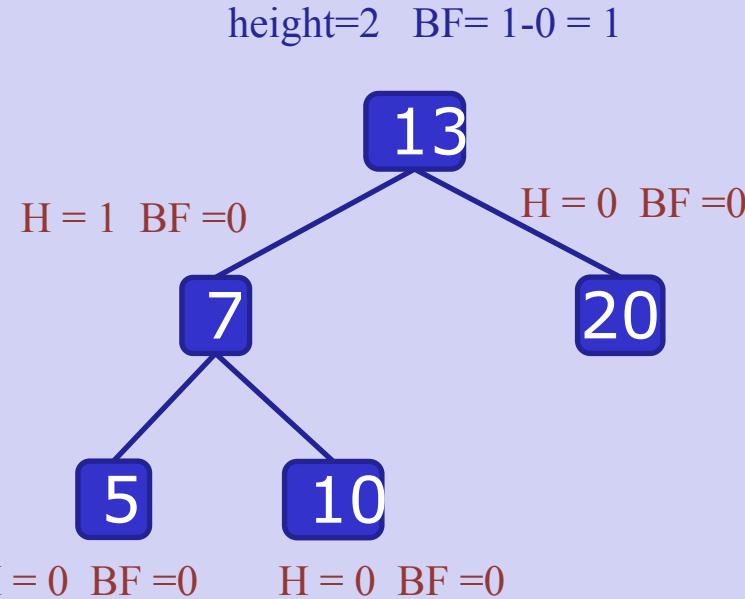


- (Continuació ...) És a dir,  $n(h) = 1 + n(h-1) + n(h-2)$
- Sabent que  $n(h-1) > n(h-2)$ , es pot veure que  $n(h) > 2n(h-2)$   
 $n(h) > 2n(h-2),$   
 $n(h) > 4n(h-4),$   
 $n(h) > 8n(h-6), \dots$  (per inducció),  
 $n(h) > 2^i n(h-2i)$
- En general:  $n(h) > 2^{h/2-1}$
- Extreient logaritmes per aïllar h:  $h < 2\log n(h) + 2$
- Per tant, l'alçada d'un arbre AVL és  $O(\log n)$

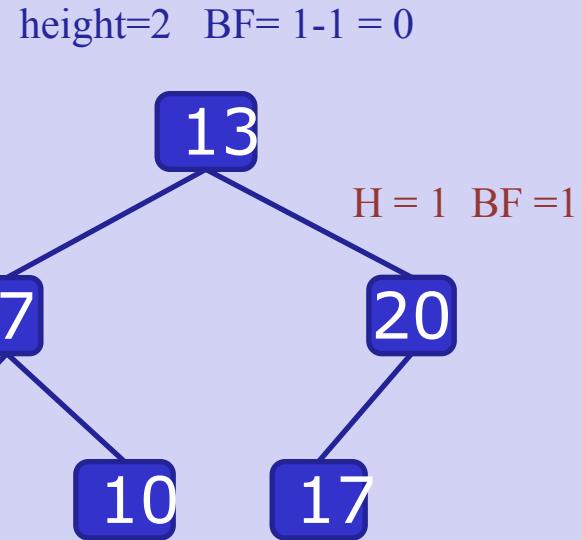
# Alçada (Height) d'un node

height of node =  $h$   
balance factor =  $h_{\text{left}} - h_{\text{right}}$   
empty height = -1

ARBRE A (AVL)



ARBRE B (AVL)

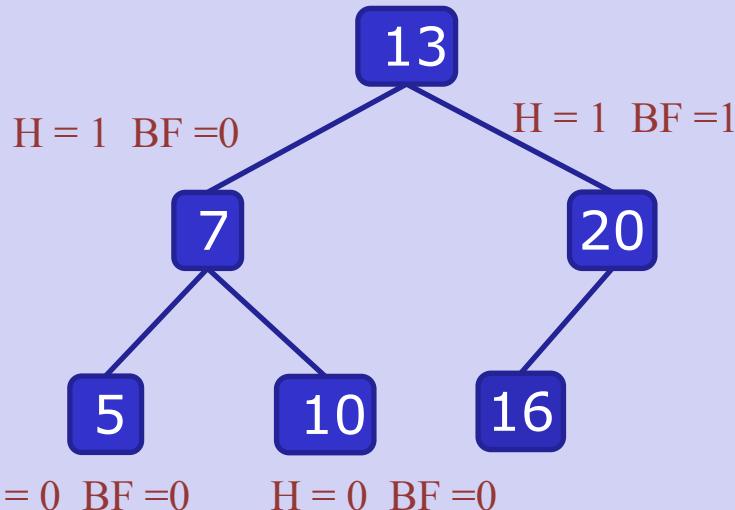


# Alçada (Height) d'un node

height of node =  $h$   
balance factor =  $h_{\text{left}} - h_{\text{right}}$   
empty height = -1

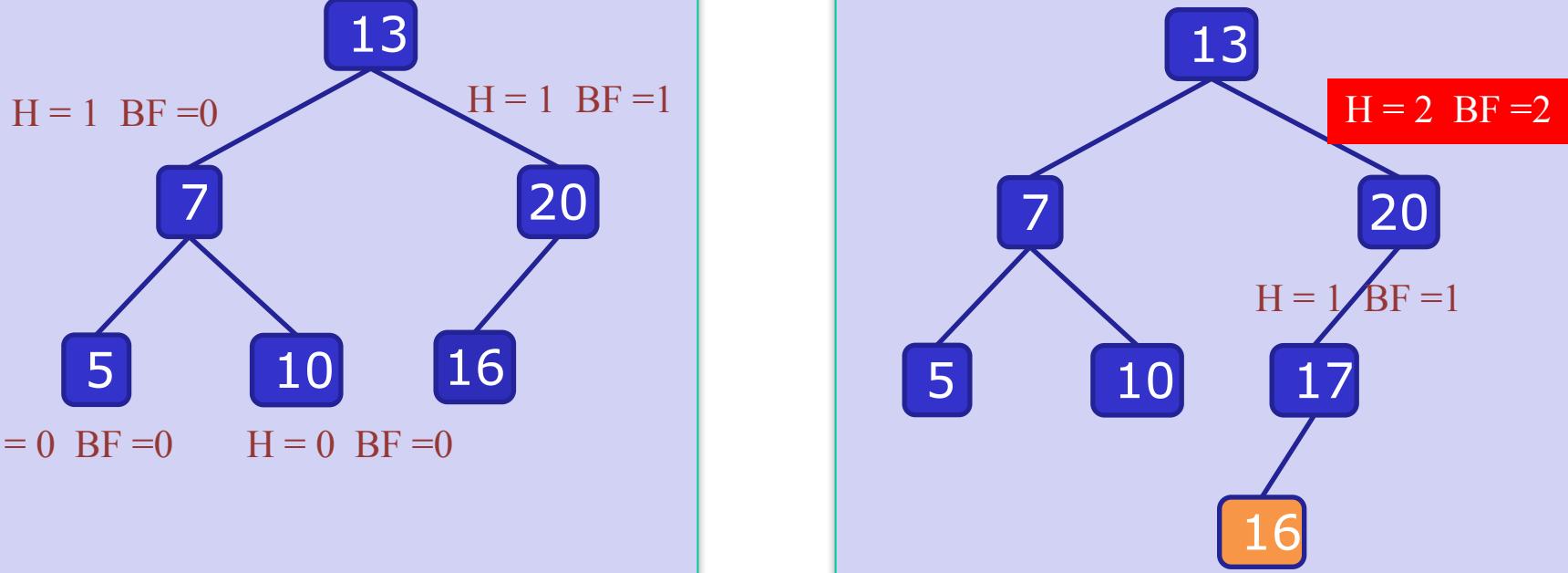
ARBRE A (AVL)

height=2 BF=  $1-1 = 0$



ARBRE B (AVL)

Height=3 BF=  $1-2 = -1$



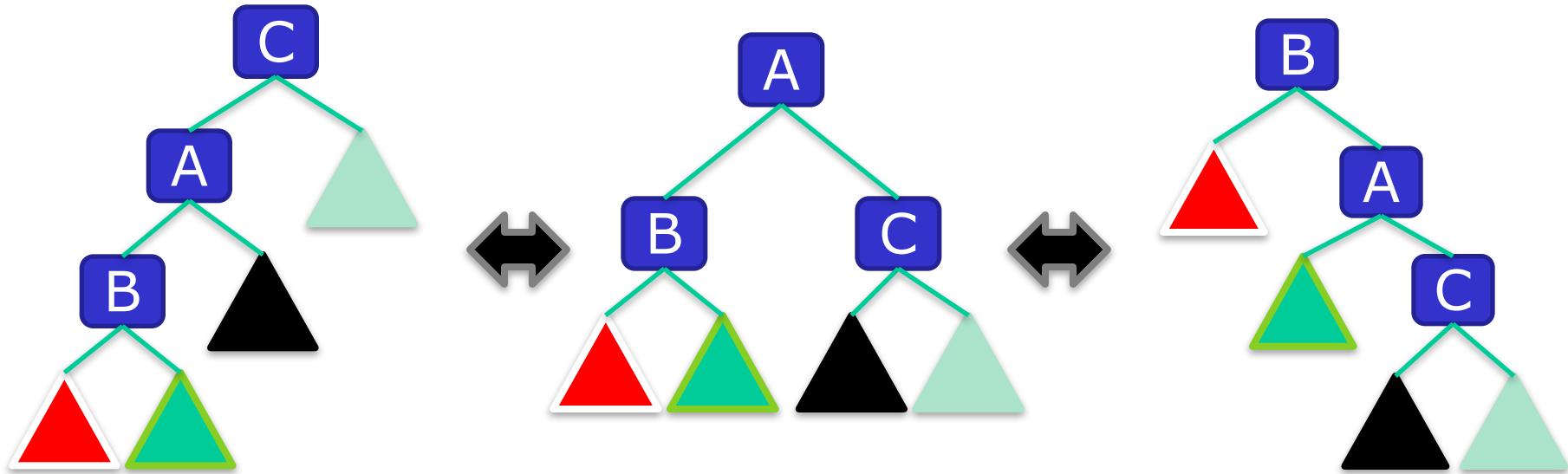


# Insertar i rotar arbres AVL

- La inserció d'un element pot causar que el factor de balanç es converteixi en 2 o -2 per algun node
  - Només els nodes des del punt d'inserció fins l'element arrel tenen la possibilitat de canviar la seva alçada
  - Per tant, la inserció **va cap a dalt** fins a l'arrel passant per cada node actualitant la seva alçada
  - Si el nou factor de balanç és 2 o -2, s'ha d'ajustar l'arbre mitjançant **rotacions** del node

# Balancejat d'un ABB: AVL

- Si l'arbre binari es converteix en un arbre no balancejat, podem revertir això mitjançant una sèrie de rotacions

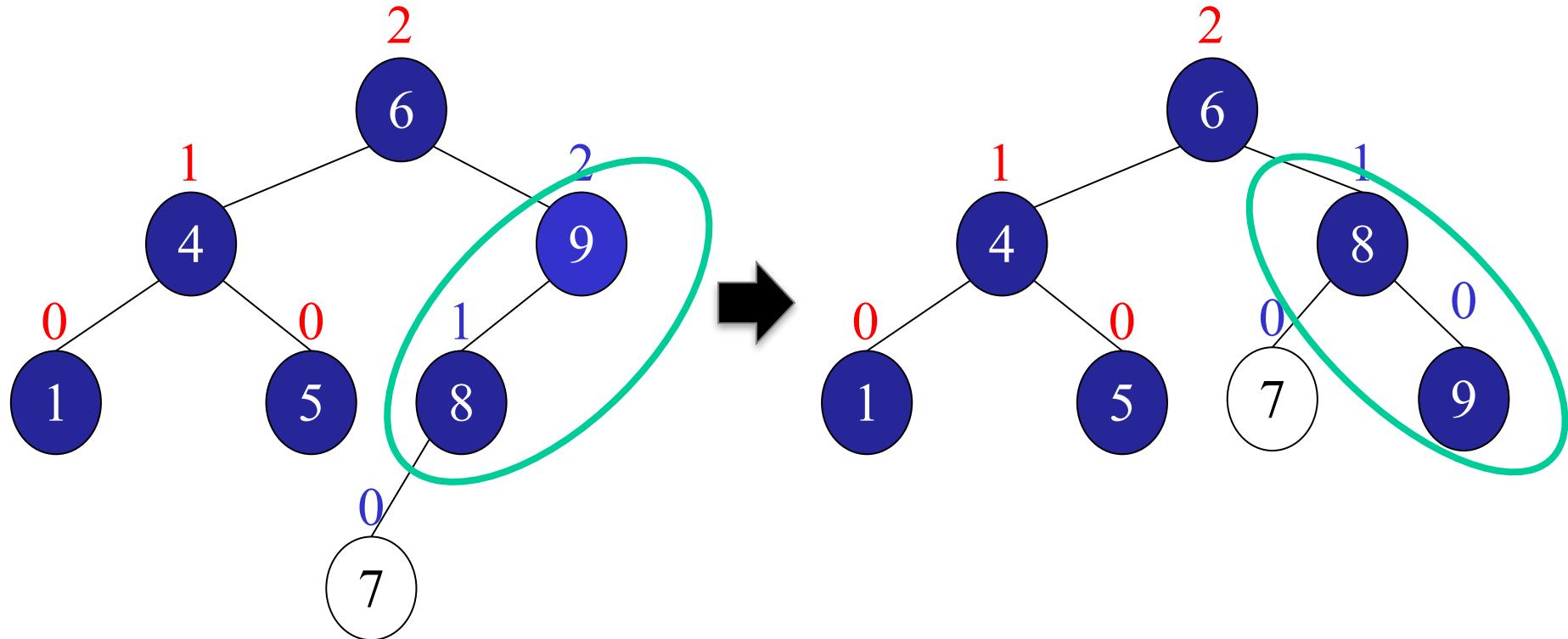


Podem veure que amb un recorregut inordre obtenim el mateix resultat



El que significa que **l'ordre dels ABB es manté amb les rotacions**

# Rotacions en un arbre AVL





# Inserció en arbres AVL

Donat un node  $\alpha$  que necessita re-balanceig:

Hi ha 4 casos:

**Casos externs** (requereix una rotació simple) :

1. Inserció en l'arbre esquerra del fill esquerra de  $\alpha$ .
2. Inserció en l'arbre dret del fill dret de  $\alpha$ .

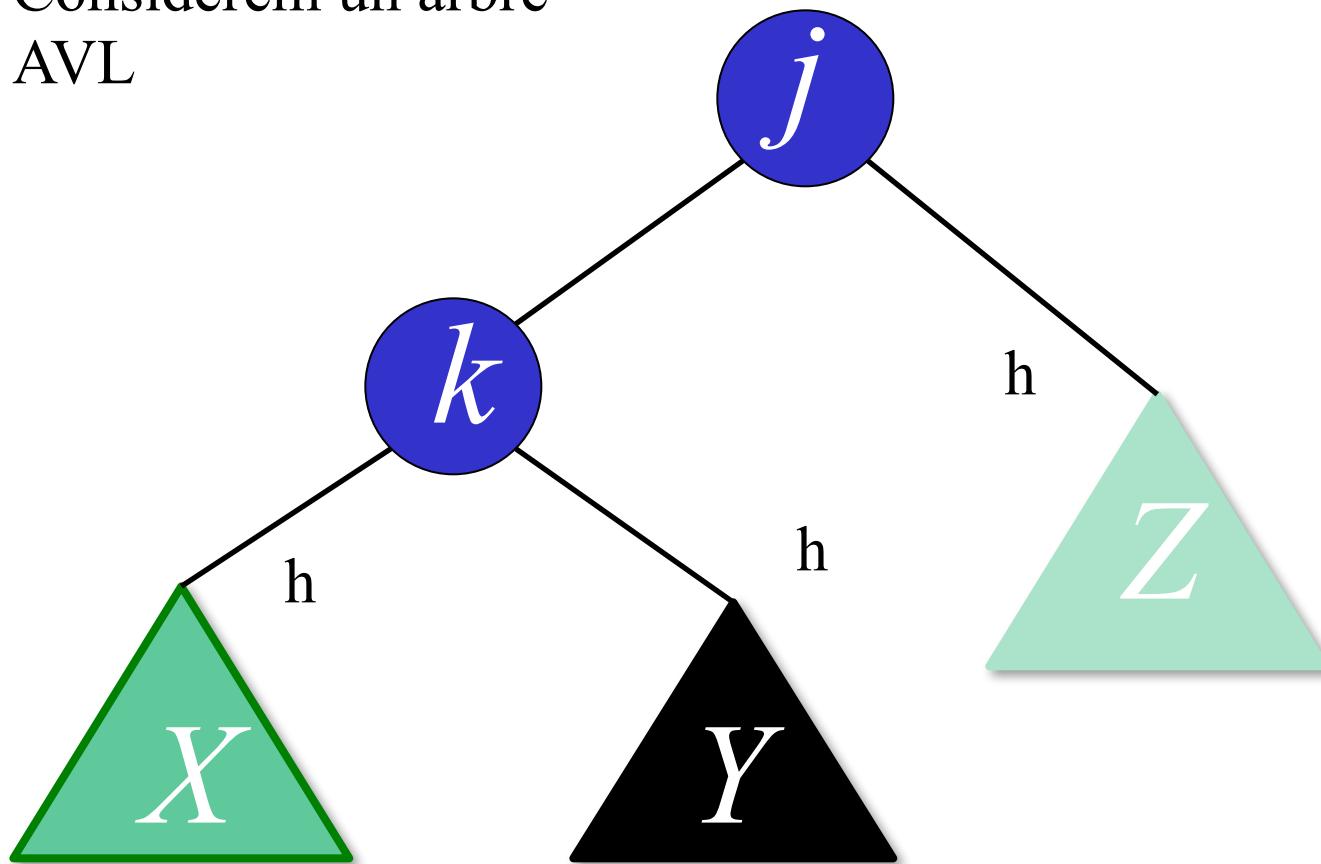
**Casos Interns** (requereix una rotació doble) :

3. Inserció en l'arbre dret del fill esquerra de  $\alpha$ .
4. Inserció en l'arbre esquerra del fill dret de  $\alpha$ .

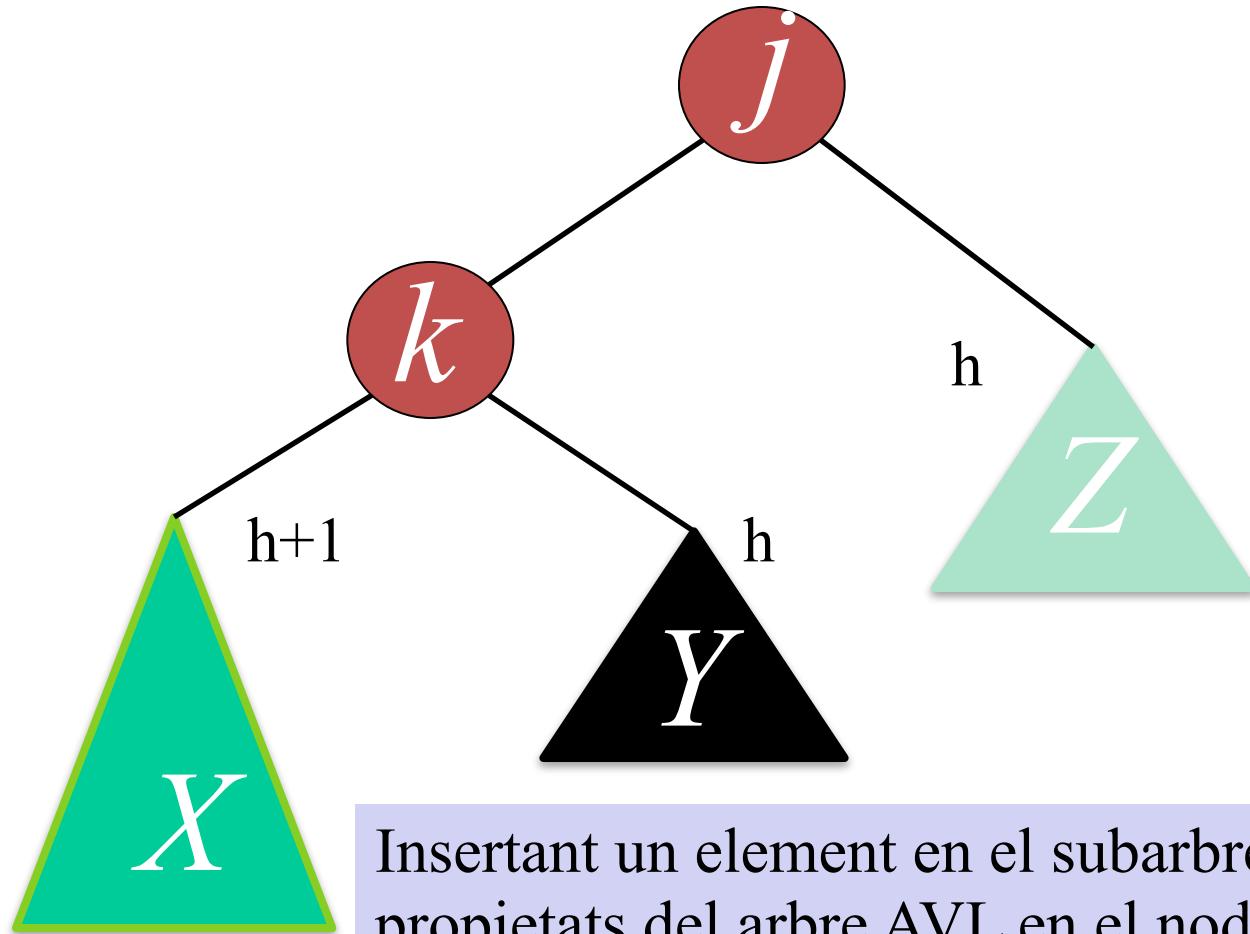
**El re-balanceig es fa mitjançant  
4 algorismes de rotació diferents**

# Inserció en un AVL : Cas extern

Considerem un arbre  
AVL

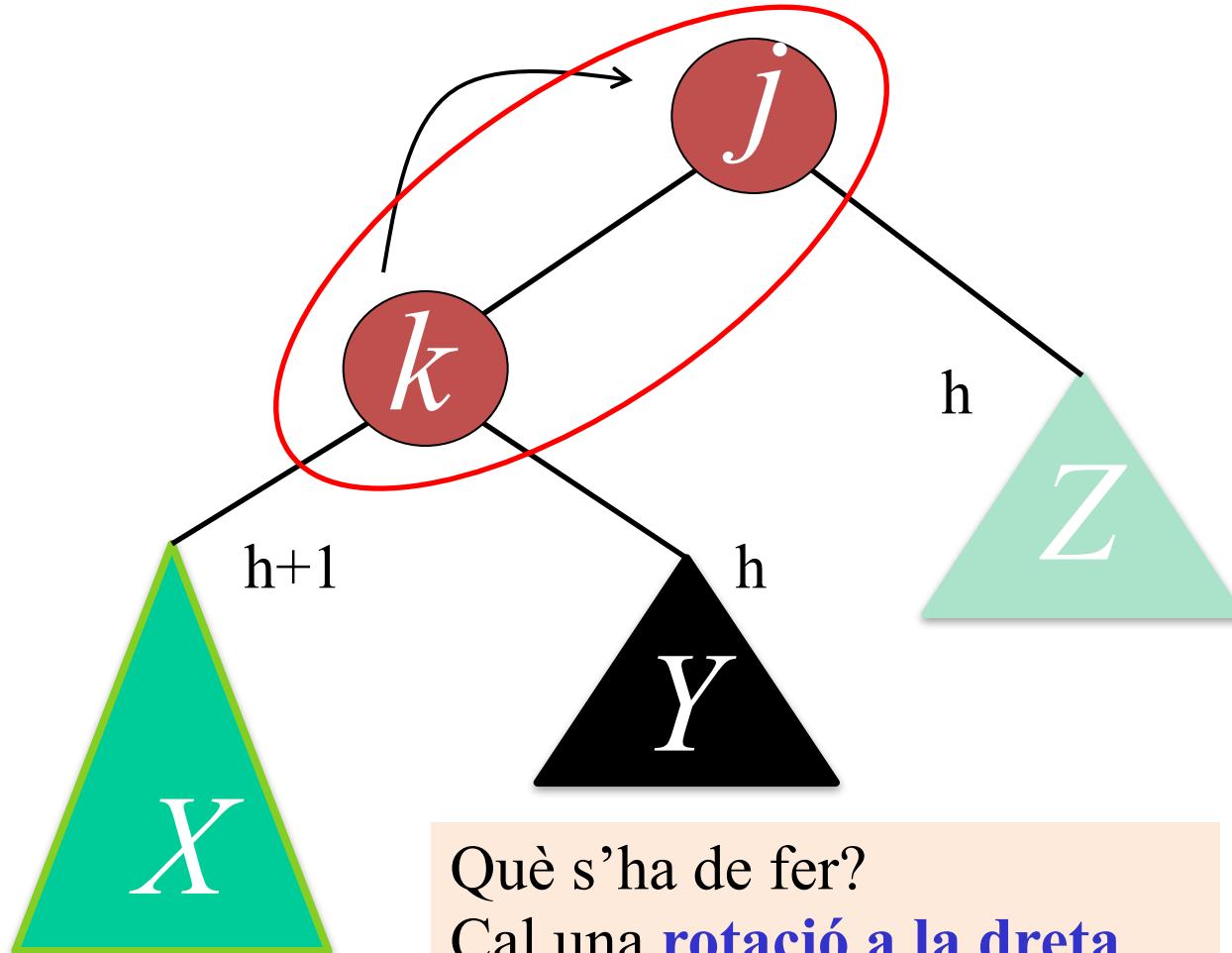


# Inserció en un AVL : Cas extern

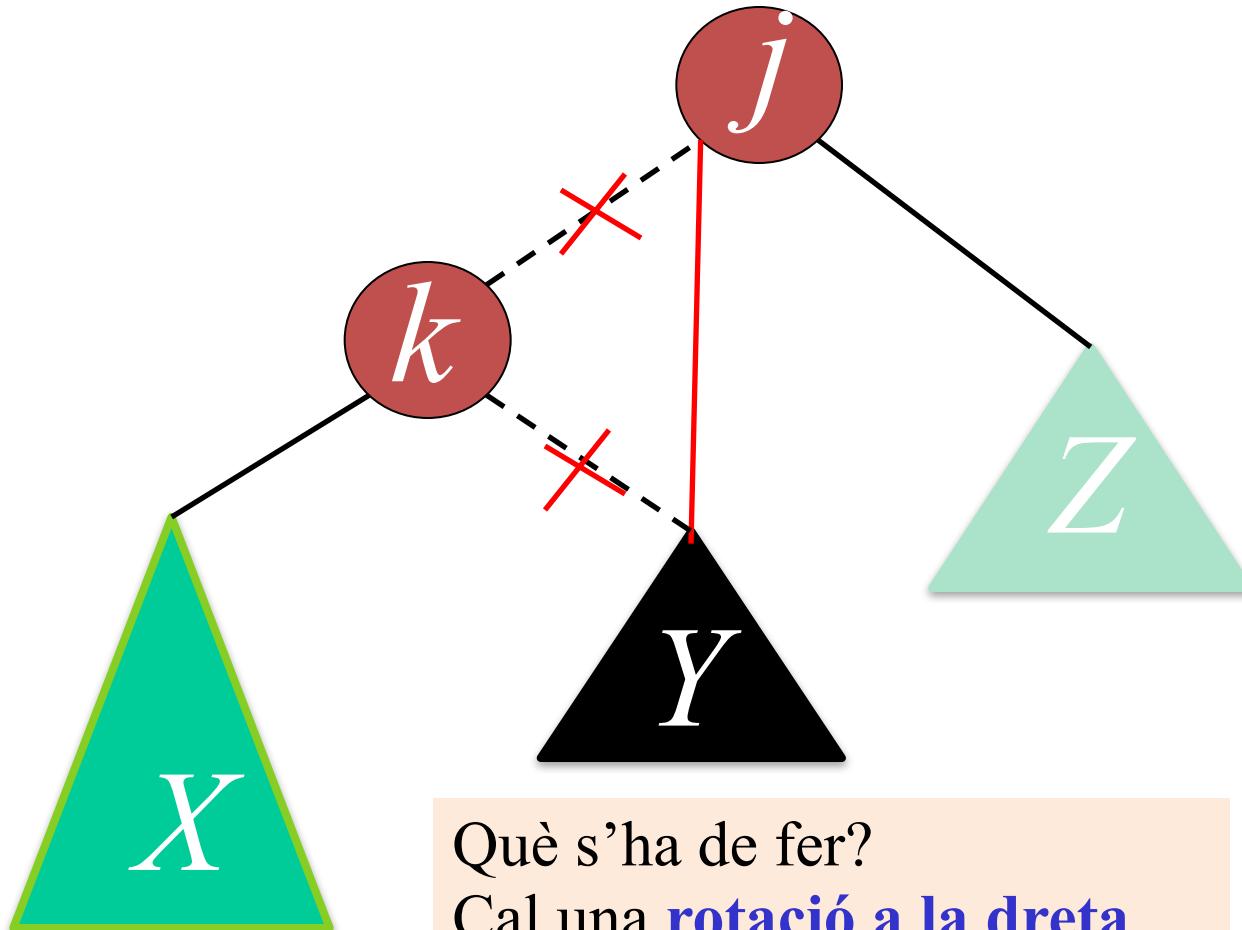


Insertant un element en el subarbre X destruim les propietats del arbre AVL en el node j

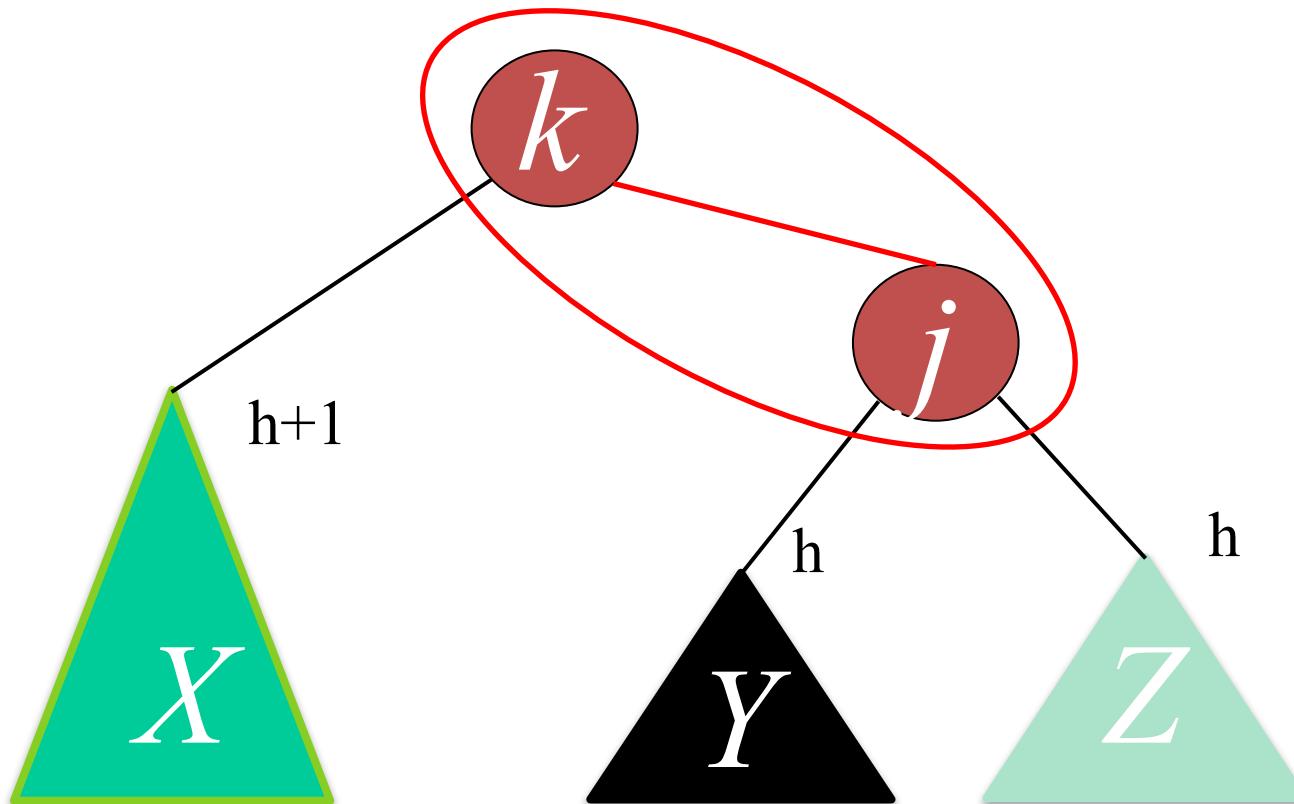
# Inserció en un AVL : Cas extern



# Rotació dreta simple



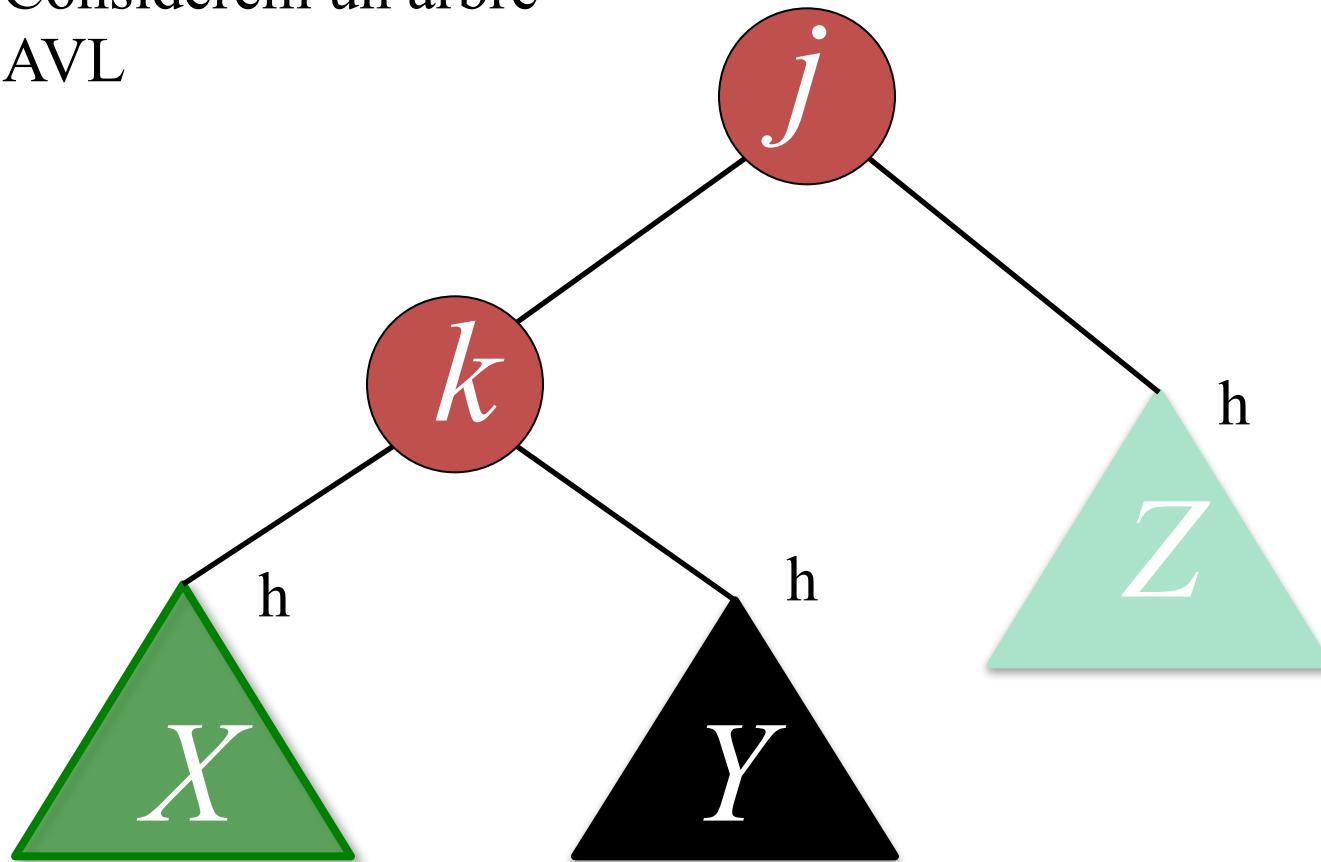
# Cas extern completat!!



“Rotació dreta” realitzada!  
 (“Rotació esquerra” és un cas simètric)  
 Les propietats de l’arbre AVL s’han restaurat

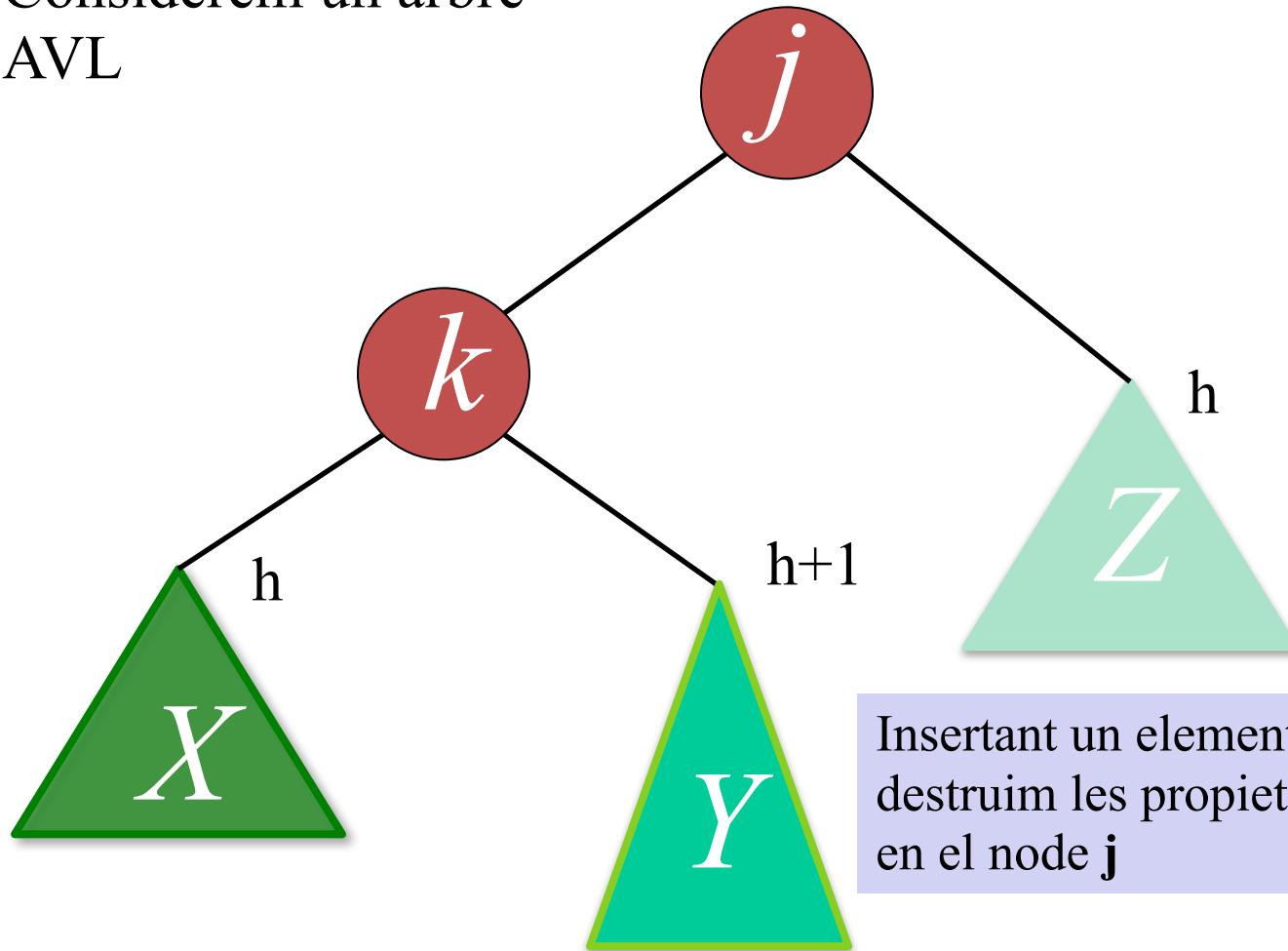
# Inserció en un AVL : Cas intern

Considerem un arbre  
AVL



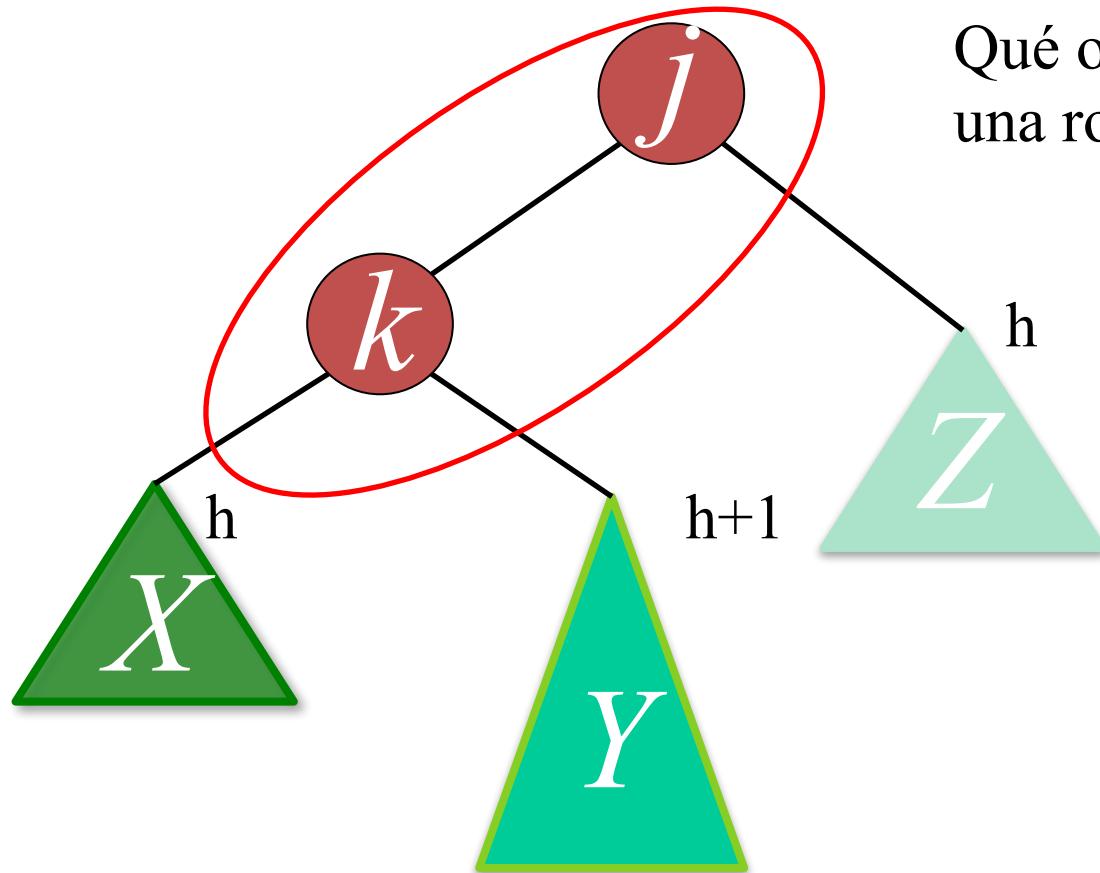
# Inserció en un AVL : Cas intern

Considerem un arbre  
AVL



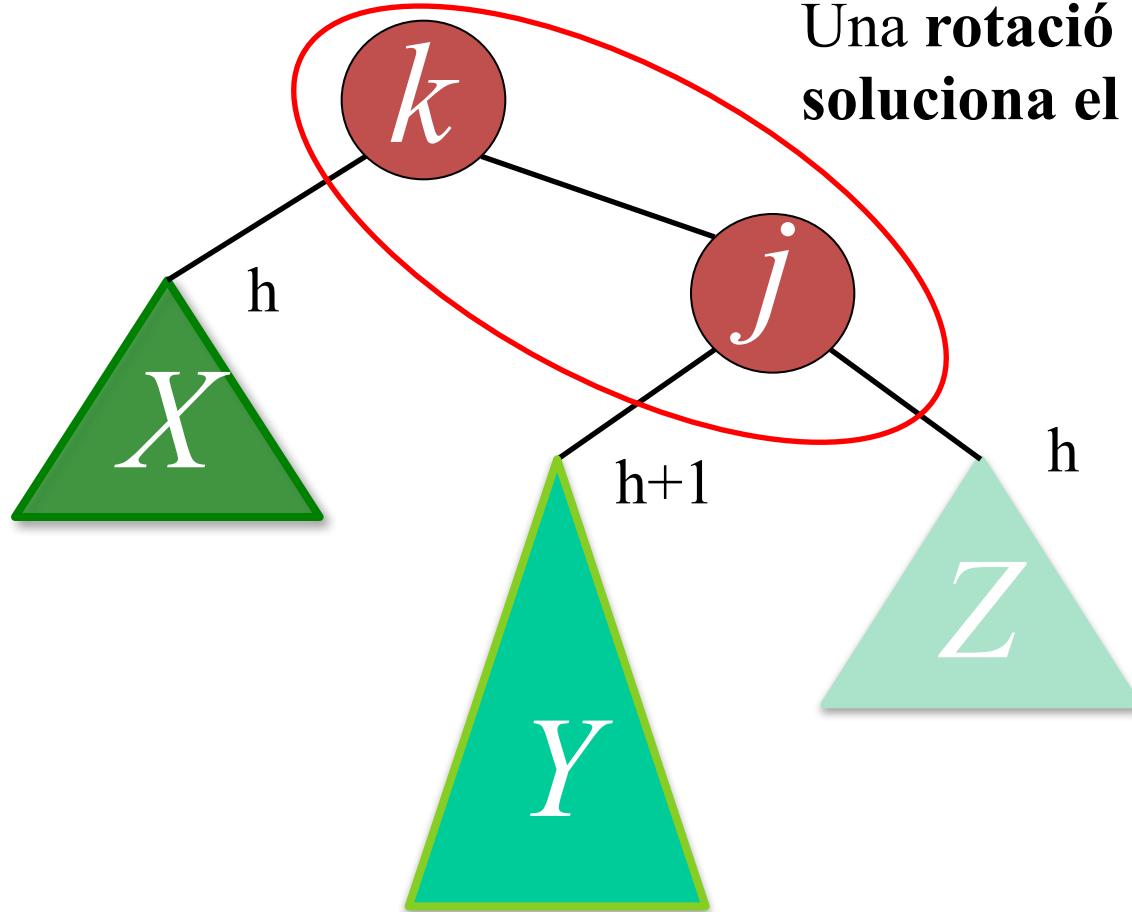
Insertant un element en el subarbre  $Y$   
destruim les propietats de l'arbre AVL  
en el node  $j$

# Inserció en un AVL : Cas intern



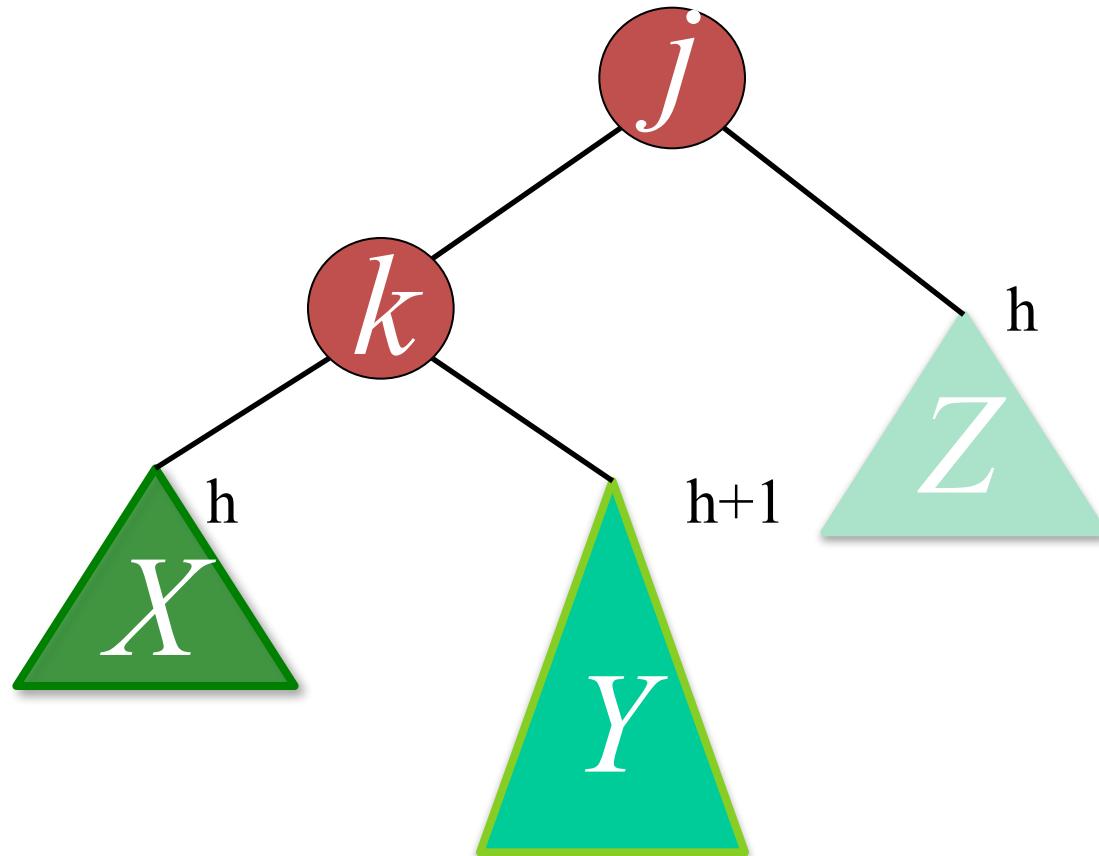
Qué ocorre si fem  
una rotació a la dreita?

# Inserció en un AVL : Cas intern



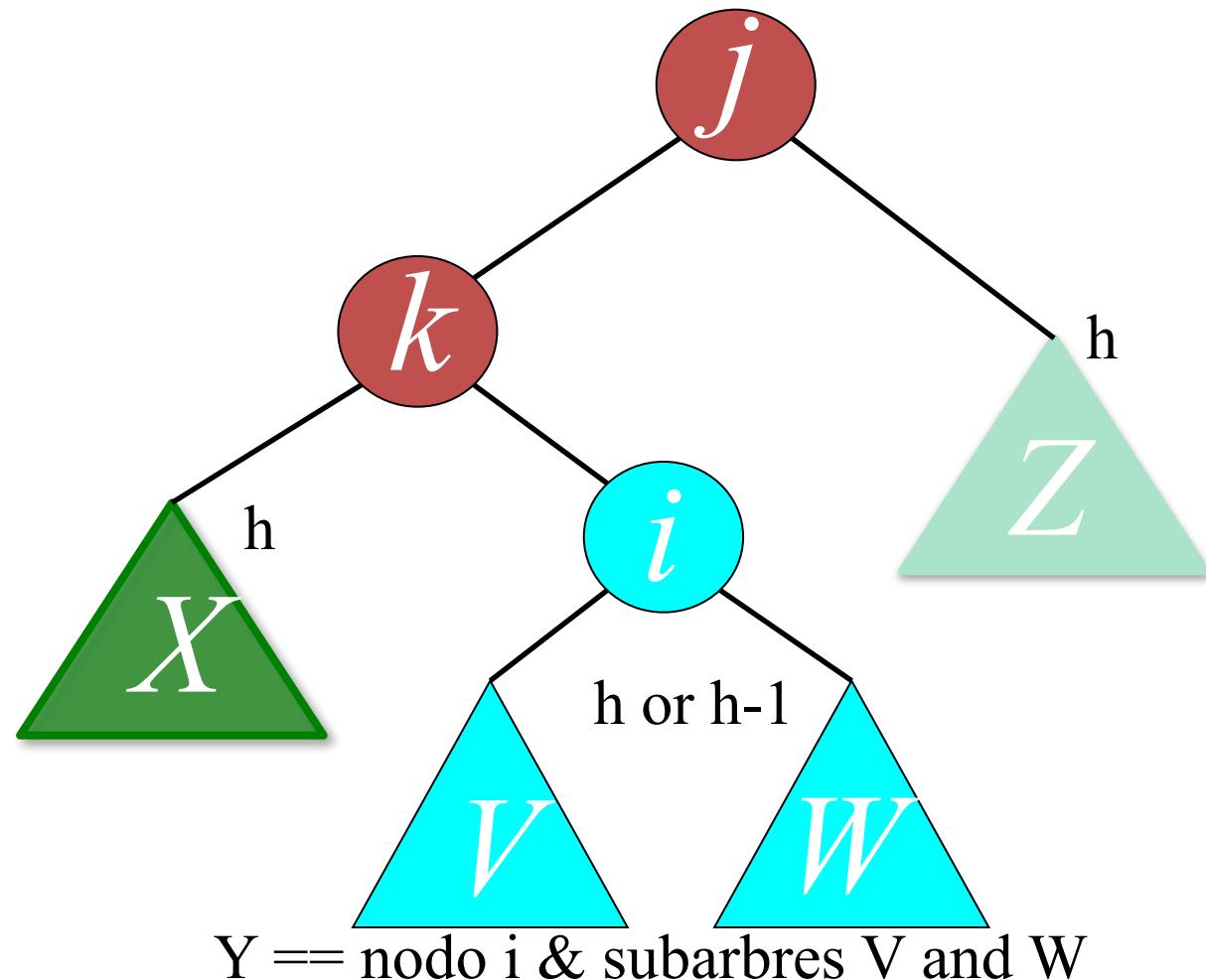
Una rotació a la dreta no soluciona el desbalanceig

# Inserció en un AVL : Cas intern

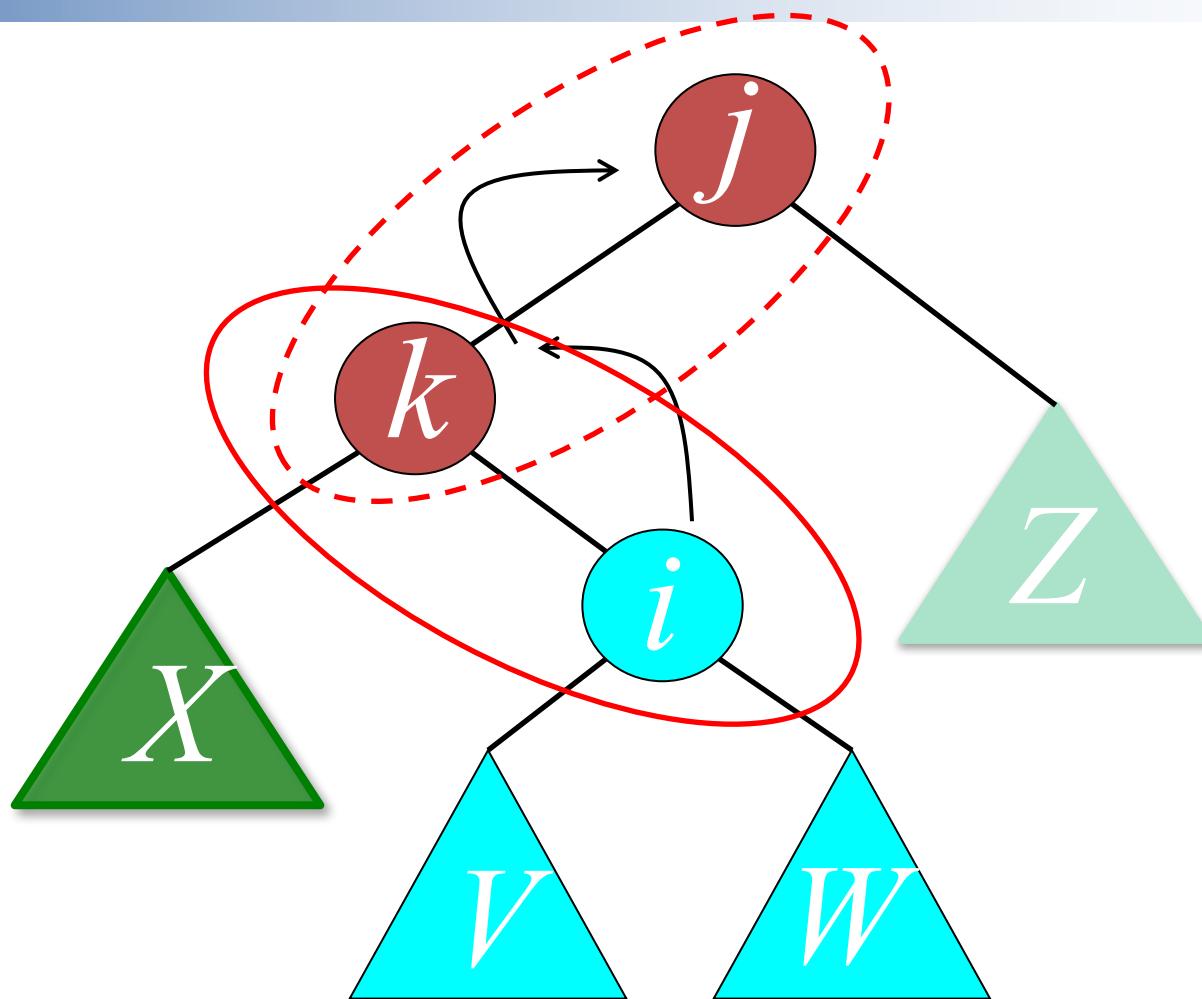


Considerem l'estructura definida pel subarbre Y

# Inserció en un AVL : Cas intern

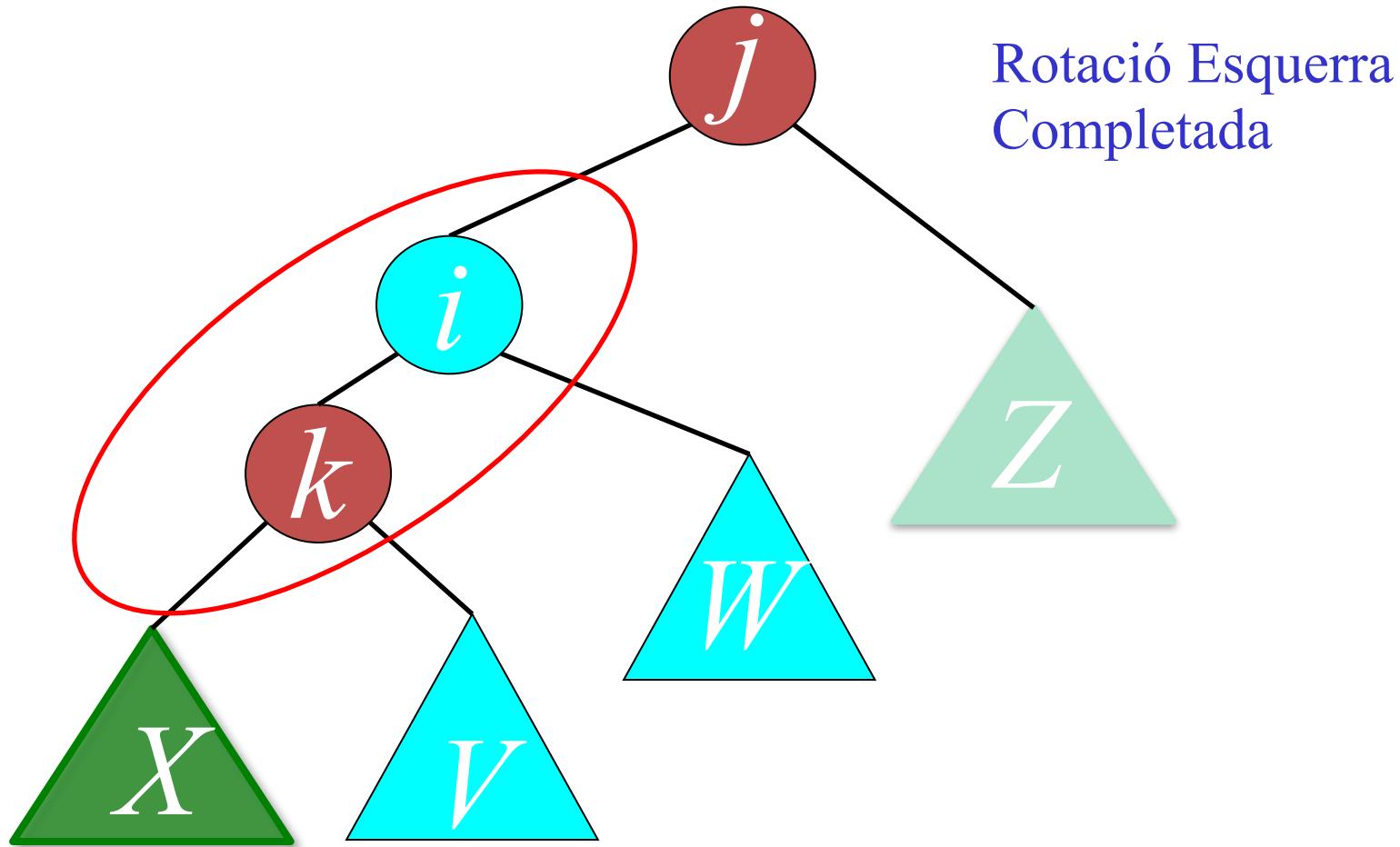


# Inserció en un AVL : Cas intern

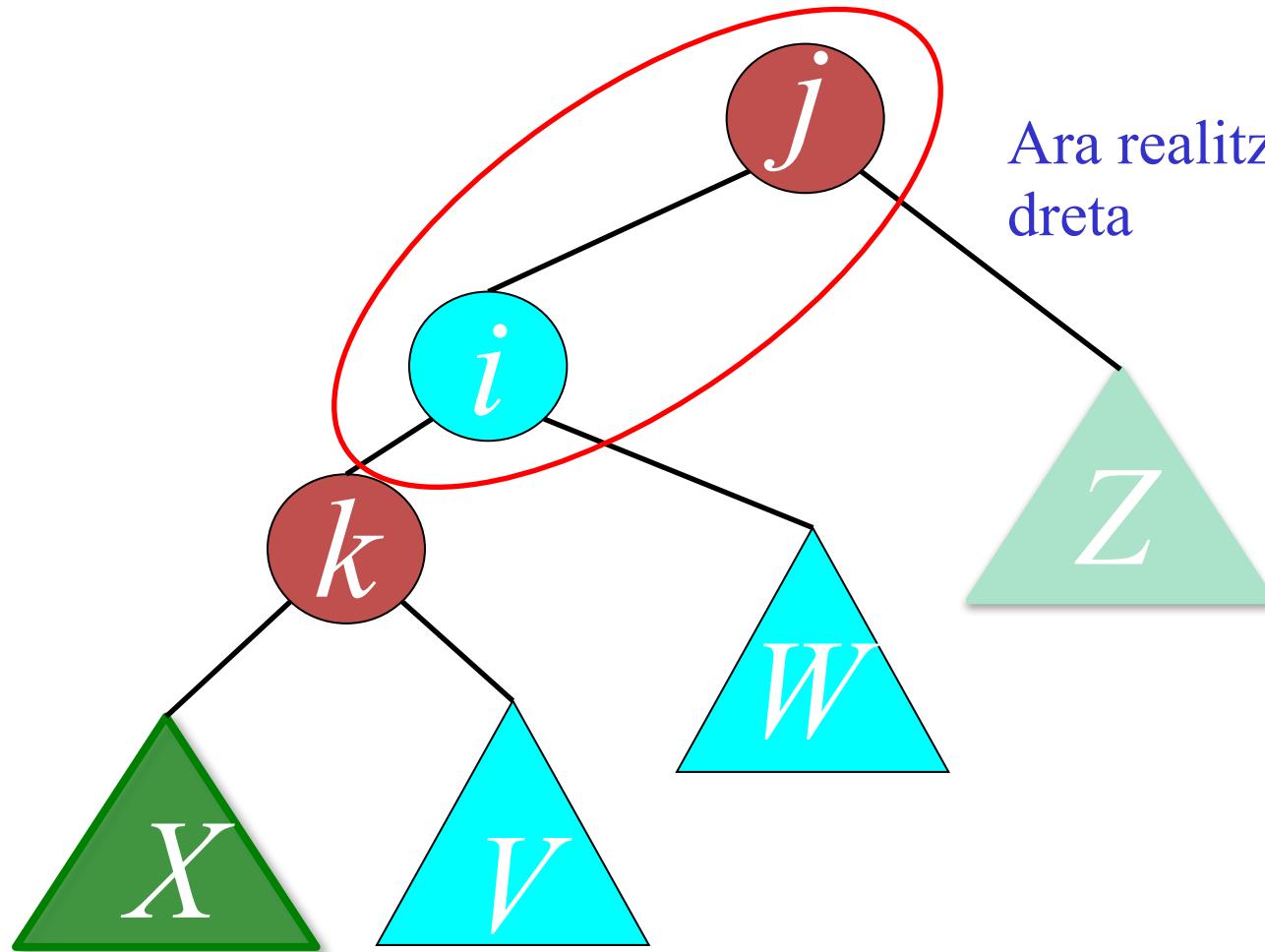


Anem a fer una doble rotació esquerra-dreta...

# Rotació Doble: Primera rotació



# Rotació Doble: Segona rotació

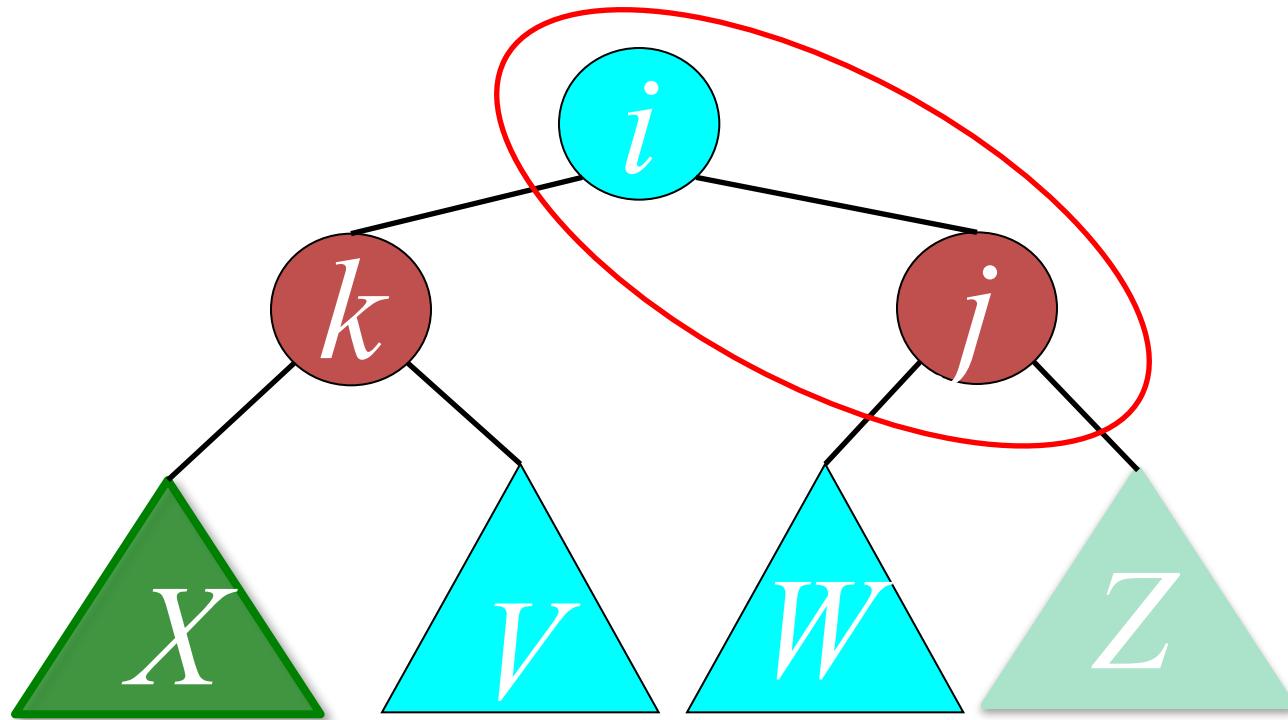


Ara realitzem una rotació a la dreta

# Rotació Doble: Segona rotació

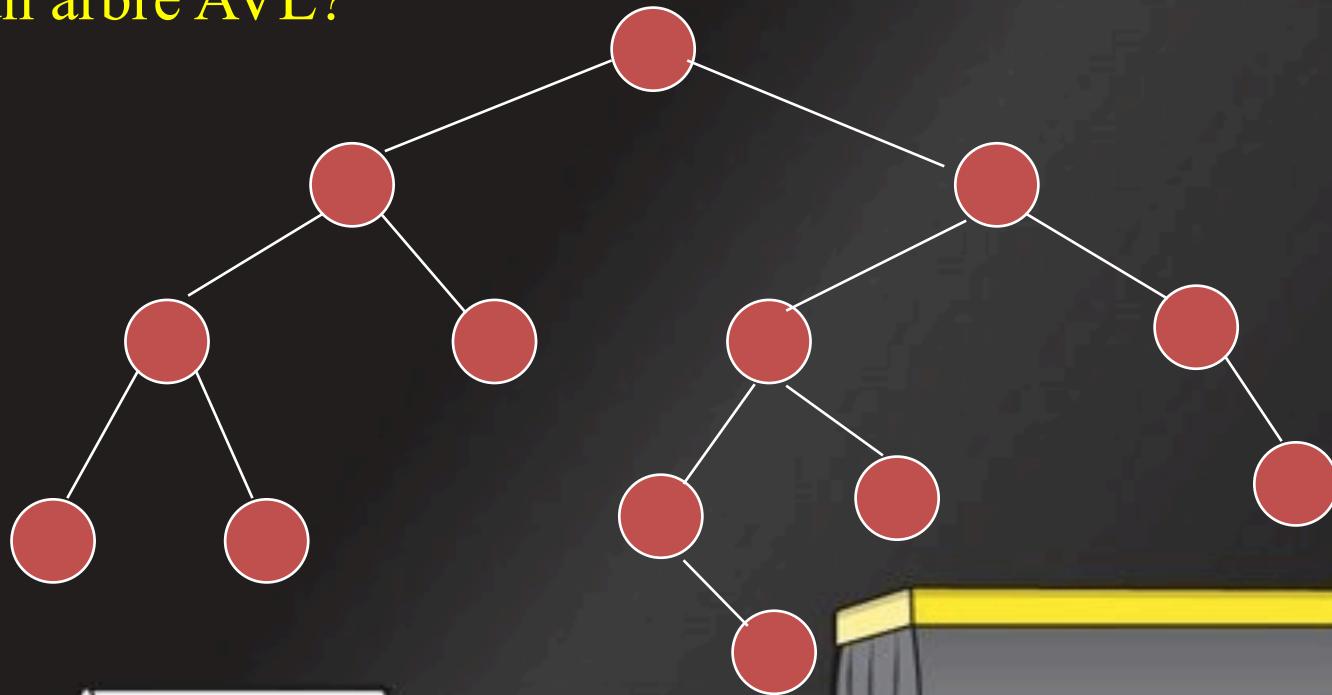
Rotació dreta completada

Arbre balancejat recuperat



# Factor de Balanç per un arbre AVL

Quin és el factor de balanç en cada node?  
És un arbre AVL?





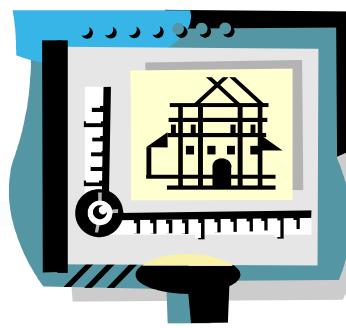
# Construcció d'un arbre AVL

Quin arbre obtenim al crear un AVL amb els següents nodes?

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



# Arbre AVL: Rendiment



- Una reestructuració simple costa,  $O(1)$  en temps
  - Usant un arbre binari amb encadenaments
- Search costa  $O(\log n)$  en temps
  - L'alçada de l'arbre és  $O(\log n)$ , no necessita reestructurar
- Insert costa  $O(\log n)$  en temps
  - La cerca inicial és de  $O(\log n)$
  - Reestructurar cap a dalt de l'arbre, mantenint les alçades té un cost de  $O(\log n)$
- Remove costa  $O(\log n)$  en temps
  - La cerca inicial és de  $O(\log n)$
  - Reestructurar cap a dalt de l'arbre, mantenint les alçades té un cost de  $O(\log n)$