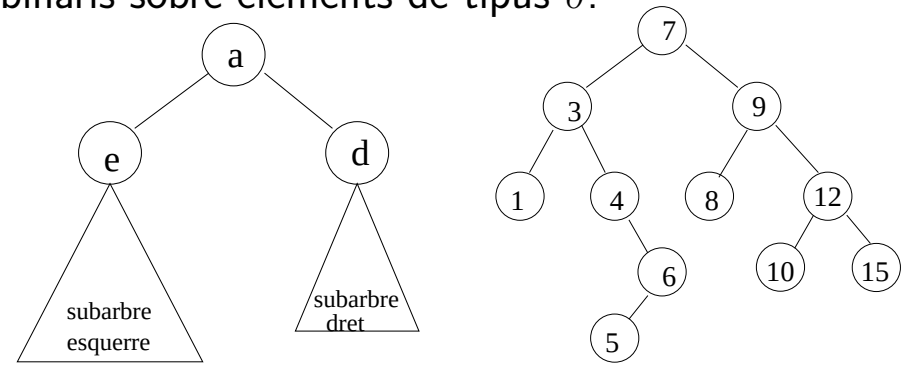
Arbres Binaris de Cerca (ABC)

$\mathcal{A}_\delta$= arbres binaris sobre elements de tipus δ.



- A cada element de l'arbre se li associa una $clau$.
- Sobre les possibles claus existeix una relació d'ordre total.
- Un arbre està format per un node $arrel$ que té un fill esquerre i un fill dret que són tots dos arbres.

Notació

$a \in \mathcal{A}_\delta$	arbre binari de cerca.
dret(a)	subarbre dret de a.
esq(a)	subarbre esquerre de a.
$\emptyset_\mathcal{A}$	arbre buit.
buit(a)	torna CERT si $a = \emptyset_\mathcal{A}$.
arrel(a) $\in \delta$	element en l'arrel de a.
clau(x)	clau associada a l'element x.
clau(a)	sinònim de clau(arrel(a)).

Arbre Binari de Cerca (ABC). Definició

- $\emptyset_{\mathcal{A}}$ és un ABC.
- $a \neq \emptyset_{\mathcal{A}}$ és un ABC si i solament si:

  esq(a) i dre(a) són ABC.

  si esq$(a) \neq \emptyset_{\mathcal{A}} \Longrightarrow$ clau(esq(a)) $<$ clau(a)

  si dre$(a) \neq \emptyset_{\mathcal{A}} \Longrightarrow$ clau(a) $<$ clau(dre(a))

ABC. Operacions bàsiques

En els ABC la operació més important és la cerca. Està molt relacionada amb la cerca binària o dicotòmica.

Donat un arbre a i un element x, trobar el node i de a tal que clau(i)=clau(x).

Algorisme:

- si $a = \emptyset_{\mathcal{A}} \Longrightarrow$ no està.
- sino si clau(a)=clau$(x) \Longrightarrow$ l'hem trobat.
- sino si clau(x) <clau$(a) \Longrightarrow$ continuar amb esq(a)
- sino si clau(a) <clau$(x) \Longrightarrow$ continuar amb dre(a)

ABC. Algorisme Trobar

Trobar (a : $\mathcal{A}_\delta$, x : δ)

// en els algorismes $x = y$ i clau(x)=clau(y) són equivalents //

Mètode:

si $a \neq \emptyset_{\mathcal{A}}$ aleshores

si $x = arrel(a)$ aleshores *trobar*$\leftarrow arrel(a)$

si no si $x <$*arrel(a)* aleshores *trobar(esq(a),x)*

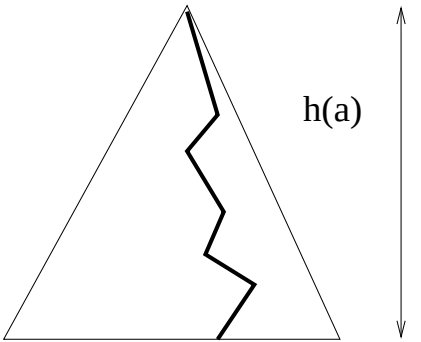
si no *trobar(dre(a),x)*

fsi

si no ELEMENT NO TROBAT

ftrobar

ABC. Algorisme Trobar



El cost de l'algorisme és $\mathcal{O}(1)$ i $\mathcal{O}(h(a))$ en els casos millor i pitjor, respectivament.

La profunditat o altura d'un arbre binari de n elements compleix que

$$\lfloor \lg n \rfloor \leq h(a) \leq n - 1$$

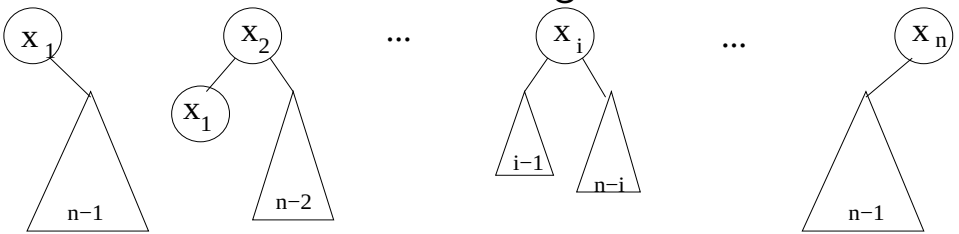
Trobar. Anàlisi per casos

Millor: $\mathcal{O}(1)$ Pitjor: $\mathcal{O}(n)$

Cas mitjà: Siga un ABC amb n elements

$$x_1 < x_2 < \cdots < x_n$$

Ens trobarem en un d'entre els n casos següents



T(n)= suma de les longituds de les branques d'un ABC de n elements.

Trobar. Anàlisi per casos

Si es troba x_i en l'arrel es compleix que

$$T_i(n) = \underbrace{T(i-1) + (i-1)}_{i-1 \text{ branques}} + \underbrace{T(n-i) + (n-i)}_{n-i \text{ branques}}$$

En sumar per als n casos...

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

ABC. Operació afegir

- Segueix exactament la mateixa idea que trobar.
- Si l'element ja s'hi troba dóna error.
- Si no, s'ha arribat a una fulla i el nou element s'afegeix com a fill dret o esquerre.

L'anàlisi per casos és idèntic al de l'operació trobar.

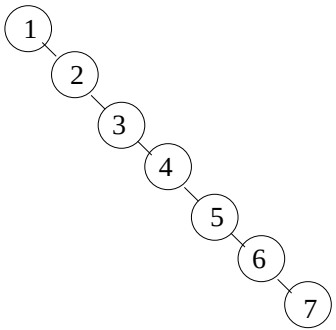
Afegir. Anàlisi amortitzada

ABC amb afegir i trobar com a úniques operacions.

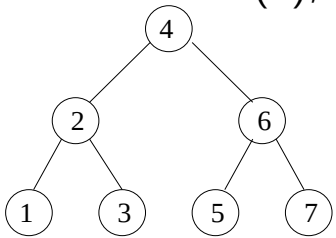
Podem establir un cost amortitzat menor que lineal? (vàlid per a qualsevol seqüencia d'operacions)

Considerem les dues seqüències d'insercions a partir de l'arbre buit:

I(1), I(2), ...I(7)

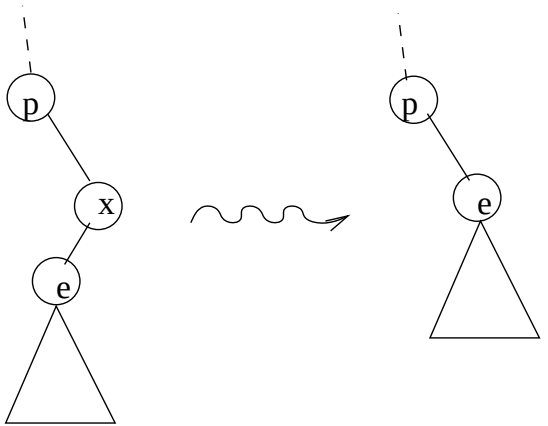


I(4), I(2), I(6), I(1), ...



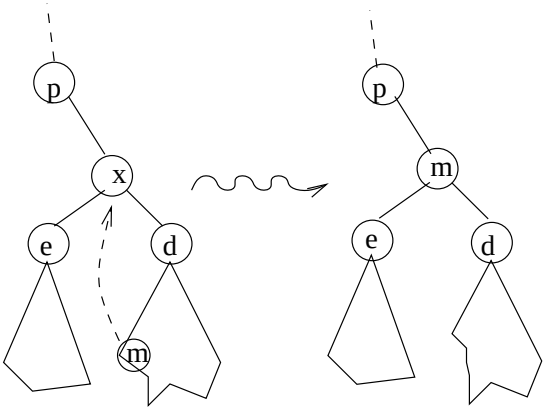
ABC. Operació eliminar

- Una primera opció senzilla és implementar una eliminació **pereosa**
- Eliminació d'un node fulla: trivial
- Eliminació d'un node que no té fill esquerre (o dret): fàcil!



ABC. Operació eliminar

- Si el node a eliminar té dos fills cal trobar l'element mínim del subarbre dret per tal que siga el nou pare dels dos subarbres.



ABC. Node Bàsic

```
package DataStructures;

class BinaryNode
{
       // Constructors
    BinaryNode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    BinaryNode( Comparable theElement, BinaryNode lt, BinaryNode rt )
    {
        element  = theElement;
        left     = lt;
        right    = rt;
    }

        // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    BinaryNode left;         // Left child
    BinaryNode right;        // Right child
}
```

La classe ABC

```
package DataStructures;
public class BinarySearchTree
{
    //METODES PUBLICS
    public BinarySearchTree( )              { root = null; }
    public void insert( Comparable x )      { root = insert( x, root ); }
    public void remove( Comparable x )      { root = remove( x, root );  }
    public Comparable findMin( )            { return elementAt( findMin( root ) ); }
    public Comparable findMax( )            { return elementAt( findMax( root ) ); }
    public Comparable find( Comparable x ) { return elementAt( find( x, root ) ); }
    public void makeEmpty( )                { root = null; }
    public boolean isEmpty( )               { return root == null; }
    //  METODES PRIVATS
    //  ...
    private BinaryNode root;
}
```

ABC. Mètodes privats

```
private BinaryNode find( Comparable x, BinaryNode t )
{
    if( t == null )
        return null;
    if( x.compareTo( t.element ) < 0 )
        return find( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        return find( x, t.right );
    else
        return t;    // Match
}
```

ABC. Mètodes privats

```
        private BinaryNode insert( Comparable x, BinaryNode t )
        {
/* 1*/      if( t == null )
/* 2*/          t = new BinaryNode( x, null, null );
/* 3*/      else if( x.compareTo( t.element ) < 0 )
/* 4*/          t.left = insert( x, t.left );
/* 5*/      else if( x.compareTo( t.element ) > 0 )
/* 6*/          t.right = insert( x, t.right );
/* 7*/      else
/* 8*/          ;  // Duplicate; do nothing
/* 9*/      return t;
        }
```

ABC. Mètodes privats (cont.)

```
private BinaryNode remove( Comparable x, BinaryNode t )
{
    if( t == null )
        return t;   // Item not found; do nothing
    if( x.compareTo( t.element ) < 0 )
        t.left = remove( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = remove( x, t.right );
    else if( t.left != null && t.right != null ) // Two children
    {
        t.element = findMin( t.right ).element;
        t.right = remove( t.element, t.right );
    }
    else
        t = ( t.left != null ) ? t.left : t.right;
    return t;
}

private BinaryNode findMin( BinaryNode t )
{
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

ABC. Interfície genèrica

```
public interface SearchTree
{
    void       insert( Comparable x ) throws DuplicateItem;
    void       remove( Comparable x ) throws ItemNotFound;
    void       removeMin( )           throws ItemNotFound;
    Comparable findMin( )             throws ItemNotFound;
    Comparable findMax( )             throws ItemNotFound;
    Comparable find( Comparable x )   throws ItemNotFound;
    void       makeEmpty( );
    boolean    isEmpty( );
    void       printTree( );
}
```

Arbres Binaris de Cerca Equilibrats (ABCE)

- A un ABC se li pot afegir una condició que assegure que la profunditat és en tot moment **logarítmica**.
- Una proposta senzilla és exigir que els dos subarbres associats a tot node tinguen la mateixa profunditat.
- ... o si açò no és possible que siguen el més paregudes possible.

Arbres AVL = ABC en els quals per a tot node x és compleix que

$$|h(esq(x)) - h(dre(x))| \leq 1$$

Altura màxima d'un AVL

Siga $N(h)$ el nombre mínim de nodes en un AVL d'altura h.

Necessàriament el AVL més menut d'altura h estarà constituït per dos subarbres d'altures $h-1$ i $h-2$ que, hauran de ser els menors AVL de cada una d'aquestes altures.

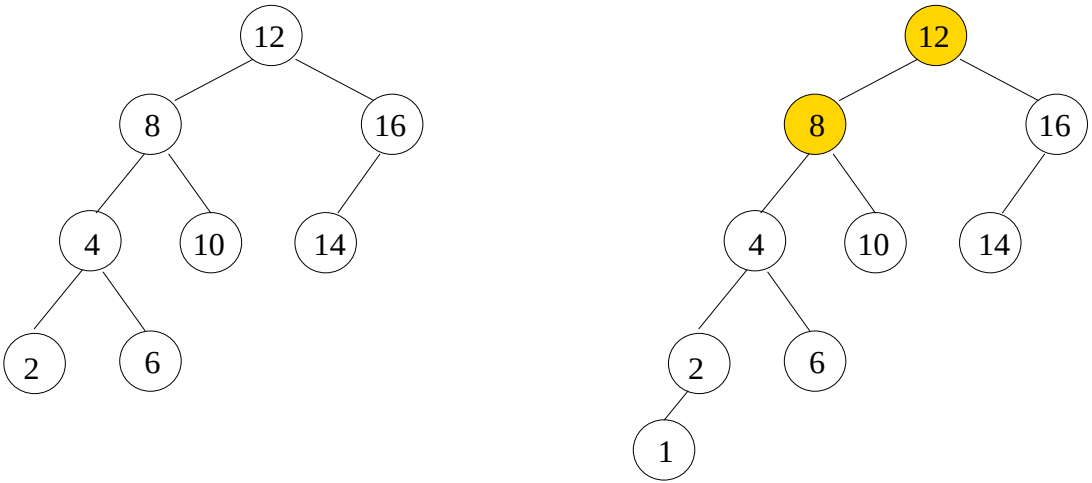
Aleshores es compleix que $$N(h) = N(h-1) + N(h-2) + 1$$

i $N(0) = 1$, $N(1) = 2$.

d'on s'obté que $N(h) \in \theta(\phi^h)$ i, per tant,

La profunditat màxima d'un AVL és de l'ordre de $\lg n$

AVL: Exemple

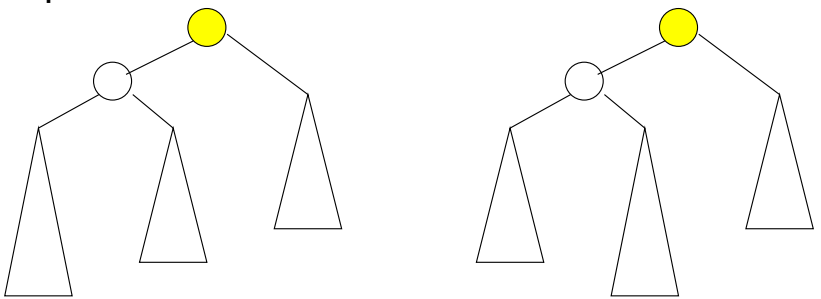


Exemple de ABC que (no) compleix la condició de AVL.

AVL: rotacions

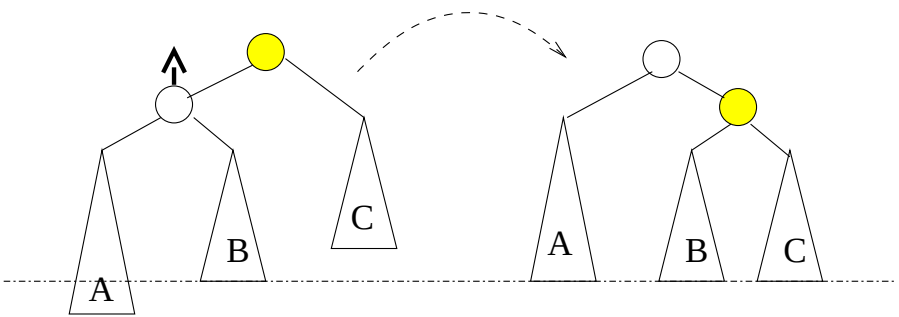
Siga un ABC que només incompleix la condició de AVL en l'arrel:

Podem distingir entre dos casos (i els seus simètrics) en funció de la forma del subarbre més profund.

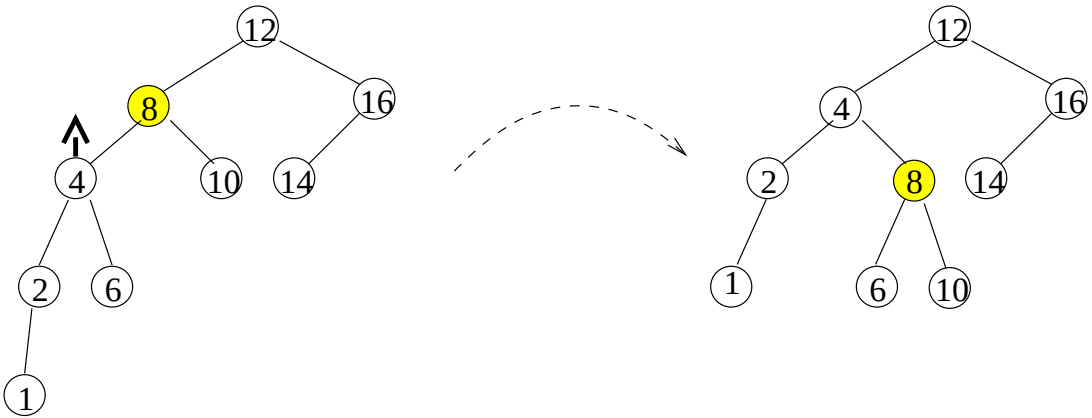


AVL:rotacions simples

En el cas més senzill la propietat AVL es pot recuperar intercanviant l'arrel amb l'arrel del subarbre més profund (i els subarbres associats).

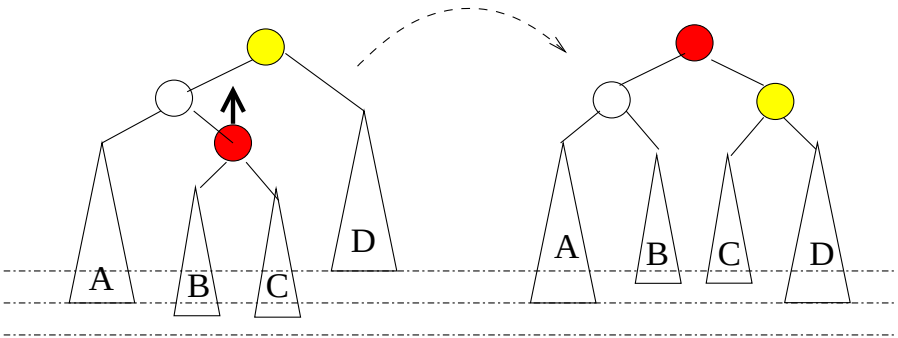


AVL:rotacions simples

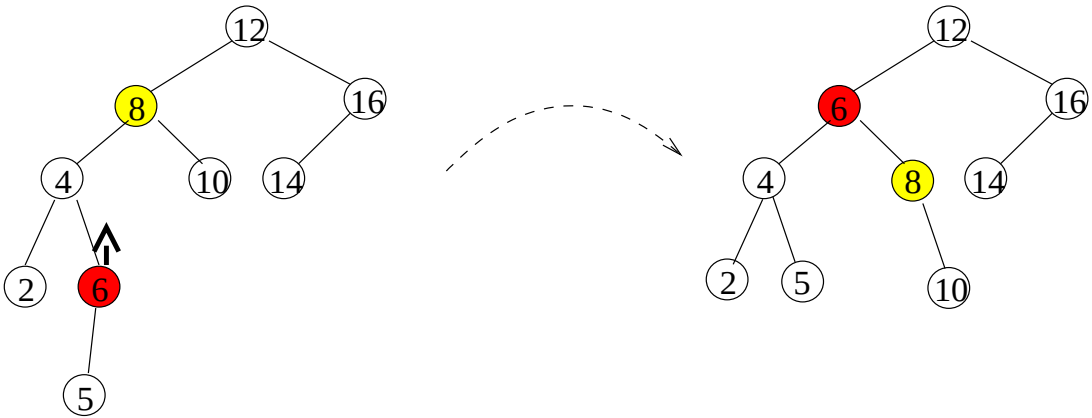


AVL:rotacions dobles

En el cas més complicat cal aplicar el mateix raonament a un nivell més baix.



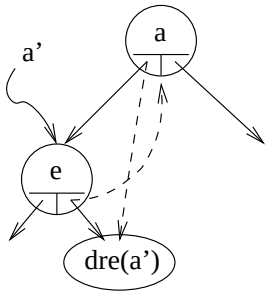
AVL:rotacions dobles



AVL: Afegir

Algorisme afegir ($a : \mathcal{A}_\delta$**,** $x : \delta$**)**
//Insereix x en un arbre AVL. Cal emmagatzemar la profunditat en cada node
  si $a = \emptyset_{\mathcal{A}}$ aleshores
    $a \leftarrow$ *crear*(x)
  si no si $x <$*arrel*(a) aleshores
    afegir$(esq(a), x)$
    reorganitzar-esq(a)
  si no si $x >$*arrel*(a) aleshores
    afegir$(dre(a), x)$
    reorganitzar-dre(a)
  si no *ERROR: x repetit*
fafegir

AVL: Afegir



Algorisme rotar-simple-esq ($a : \mathcal{A}_\delta$)

$a' \leftarrow esq(a)$
$esq(a) \leftarrow dre(a')$
$dre(a') \leftarrow a$
$h(a) = 1 + \max(h(esq(a), h(dre(a)))$
$h(a') = 1 + \max(h(esq(a'), h(dre(a')))$
$a \leftarrow a'$

Algorisme rotar-doble-esq ($a : \mathcal{A}_\delta$)

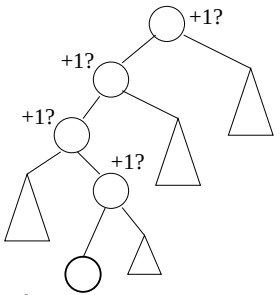
rotar-simple-dre$(esq(a))$
rotar-simple-esq(a)

Algorisme reorganitzar-esq ($a : \mathcal{A}_\delta$, $x : \delta$)

`si` $|h(esq(a)) - h(dre(a))| > 1$ `aleshores`
  `si` $x < esq(a)$
  `aleshores` *rotar-simple-esq*(a)
  `si no` *rotar-doble-esq*(a)
  `fsi`
$h(a) = 1 + \max(h(esq(a), h(dre(a)))$

Cal reorganitzar al llarg de tota la branca?

En afegir una nova fulla alguns predecessors poden augmentar la seua altura en una unitat (la propietat AVL es pot perdre).



Les rotacions (simples i dobles) impliquen necessàriament que el subarbre considerat **redueix** la seua altura en una unitat.

$\Rightarrow$ Una vegada aplicada la primera rotació, tots els nodes per dalt en la mateixa branca compliran la propietat AVL.

AVL: implementació

```
private AvlNode insert( Comparable x, AvlNode t )
{
    if( t == null )
        t = new AvlNode( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
    {
        t.left = insert( x, t.left );
        if( height( t.left ) - height( t.right ) == 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
    }
    else if( x.compareTo( t.element ) > 0 )
    {
        t.right = insert( x, t.right );
        if( height( t.right ) - height( t.left ) == 2 )
            if( x.compareTo( t.right.element ) > 0 )
                t = rotateWithRightChild( t );
            else
                t = doubleWithRightChild( t );
    }
    else
        ;  // Duplicate; do nothing
    t.height = max( height( t.left ), height( t.right ) ) + 1;
    return t;
}
```

AVL: implementació

```
private static AvlNode rotateWithLeftChild( AvlNode k2 )
{
    AvlNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
    k1.height = max( height( k1.left ), k2.height ) + 1;
    return k1;
}

private static AvlNode doubleWithLeftChild( AvlNode k3 )
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}
```

AVL: eliminar

- Si l'element a borrar només té un fill, el subarbre considerat se sustitueix pel seu (únic) fill que necessàriament serà AVL.
- Si l'element a borrar té dos fills, s'aplica l'operació borrarmax (o borrarmin). Com a conseqüència, un dels fills pot disminuir d'altura i caldra reorganitzar **l'altre**.

En els dos casos, caldrà reorganitzar tots els predecessors del node en qüestió.

L'operació borrarmax (o min) per la seua banda, haurà d'assegurar que el resultat és AVL (caldrà reorganitzar internament).

AVL: eliminar

Algorisme eliminar ($a : \mathcal{A}_\delta$**,** $x : \delta$**)**

```
si a = ∅_A aleshores                ERROR: no trobat
si no si x <arrel(a) aleshores
        eliminar(esq(a), x)
        reorganitzar2-dre(a)
si no si x >arrel(a) aleshores
        eliminar(dre(a), x)
        reorganitzar2-esq(a)
si no si {només un fill} aleshores arrel(a) ← fill
si no   //trobat x
        m ← borramax(esq(a))
        arrel(a) ← m
        reorganitzar2-dre(a)
        fsi
```

AVL: eliminar

Algorisme borramax ($a : \mathcal{A}_\delta$**,** $x : \delta$**) :** δ

<u>si</u> $dre(a) = \emptyset_{\mathcal{A}}$ <u>aleshores</u>

$borramax \leftarrow arrel(a)$

$a \leftarrow esq(a)$

$h(a) \leftarrow h(a) - 1$

<u>si no</u>

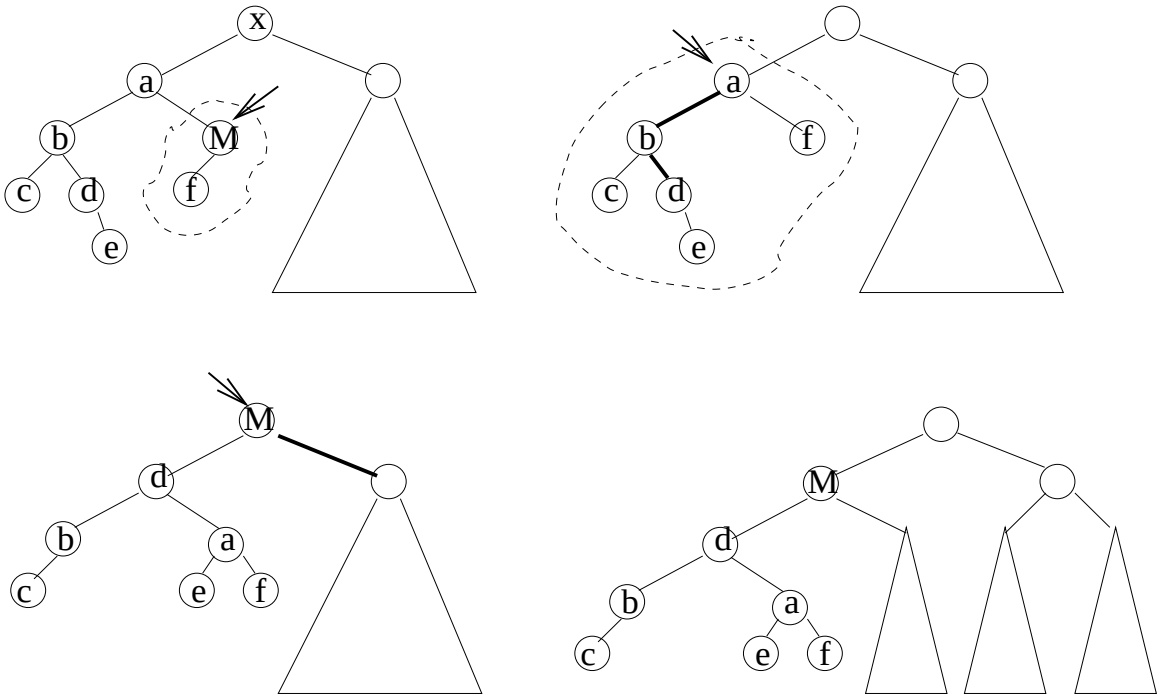
$borramax \leftarrow borramax(dre(a))$

$reorganitzar2\text{-}esq(a)$

<u>fsi</u>

fborramax

AVL: eliminar



Arbres Roig-Negre (ARN)

Són ABCE en els quals la condició d'equilibri s'introdueix de forma indirecta.

En lloc de calcular i emmagatzemar l'altura (informació global) es considera el **color** del node (informació local).

Els ARN són menys equilibrats que els AVL.

Permeten inserció i eliminació de nodes mijantçant un únic recorregut descendent.

Es poden implementar sense recursió de manera simple i eficient.

ARN: definició

Un ARN és un ABC en el qual tot node té assignat un color (roig o negre) i que, a més a més compleix:

P1) color(arrel(a))=color(a)=negre
(l'arrel és sempre negra).

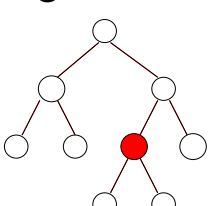
P2) per a tot subarbre x es compleix
color(x)=roig $\Rightarrow$ color(esq(x))=color(dre(x))=negre
(nodes rojos han de tindre fills negres) def: color($\emptyset_{\mathcal{A}}$)=negre

P3) Tot camí des de qualsevol node, x fins a un subarbre buit, ha de contindre el mateix nombre de nodes negres.

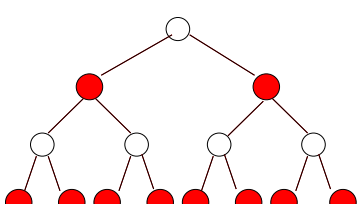
ARN: definició

En relació a la propietat P3 es defineix la **altura negra** d'un node, $h_N(x)$, com el nombre de nodes negres que hi ha en tot camí des de x fins a qualsevol subarbre buit.

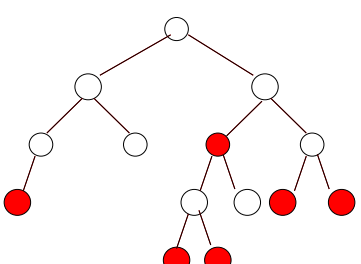
La propietat P2 garanteix que el nombre de nodes rojos no puga ser massa gran.



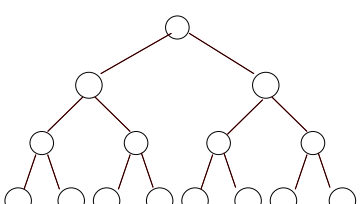
$h_N(a) = 3$



$h_N(a) = 2$



$h_N(a) = 3$



$h_N(a) = 4$

ARN: propietats

1) $|x| \geq 2^{h_N(x)} - 1$ (*) (Per inducció en el nombre de nodes)

- Base: $|\emptyset_{\mathcal{A}}| = 0$, $h_N(\emptyset_{\mathcal{A}}) = 0$.
- Pas d'inducció:

$$\text{P3} \Rightarrow \underbrace{h_N(x) - 1 \leq h_N(fill(x))}_{(**)} \leq h_N(x)$$

$$|x| = |esq(x)| + |dre(x)| + 1 \overset{(H.I.)}{\geq}$$

$$(2^{h_N(esq(x))} - 1) + (2^{h_N(dre(x))} - 1) + 1 \overset{(**)}{\geq}$$

$$2 \cdot 2^{h_N(x)-1} - 1 = 2^{h_N(x)} - 1$$

ARN: propietats

2) $\frac{h(a)}{2} \leq h_N(a) \leq h(a) + 1$ (per P2)

3) Els ARN tenen profunditat logarítmica

$$n = |a| \geq 2^{h_N(x)} - 1 \geq 2^{\frac{h(a)}{2}} - 1$$

$$\boxed{h(a) \leq 2\lg(n+1)}$$

ARN: afegir

Es fa servir la inserció dels ABC.

Com que el nou node ha de ser una nova fulla ha de ser **roig** (per P3).

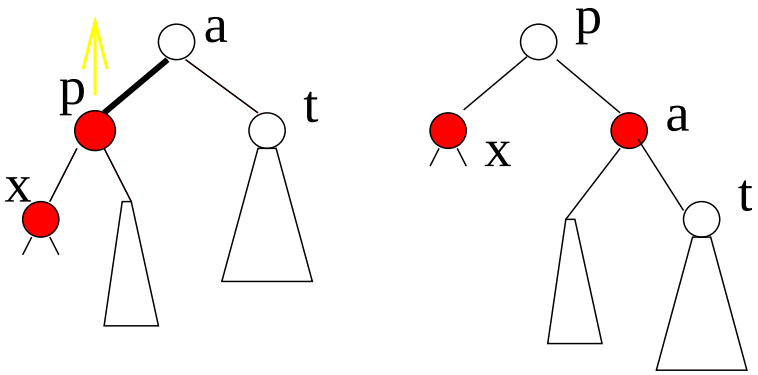
Afegir una fulla roja pot produir que son pare incomplesca P2 (si és roig).

Es pot fer que l'arbre torne a ser ARN després d'afegir una fulla roja a un node roig ...

ARN: afegir ascendent

S'acaba d'afegir una fulla roja, x, a un node roig:

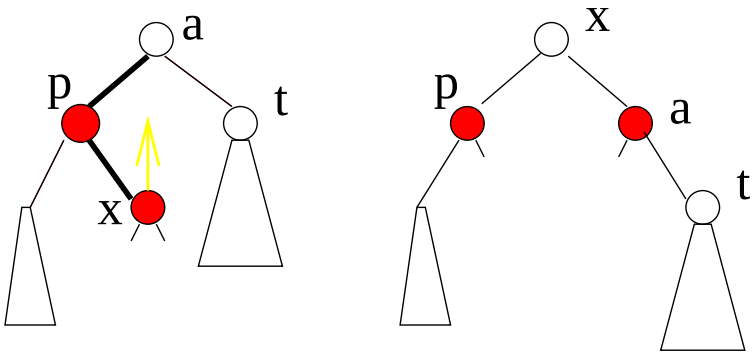
cas 1: tio negre, x extern
$\Rightarrow$ rotació simple



Es compleix que $h_N(a)$ és la mateixa abans i després!

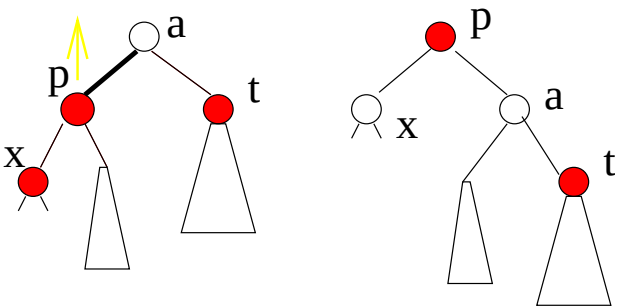
ARN: afegir ascendent

cas 2: tio negre, x intern
$\Rightarrow$ rotació doble

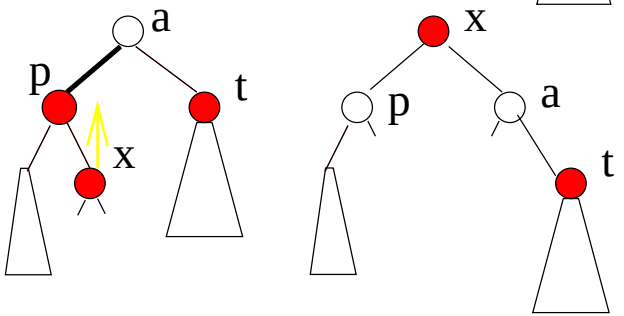


ARN: afegir ascendent

cas 3: tio roig,
x extern $\Rightarrow$ rotació simple

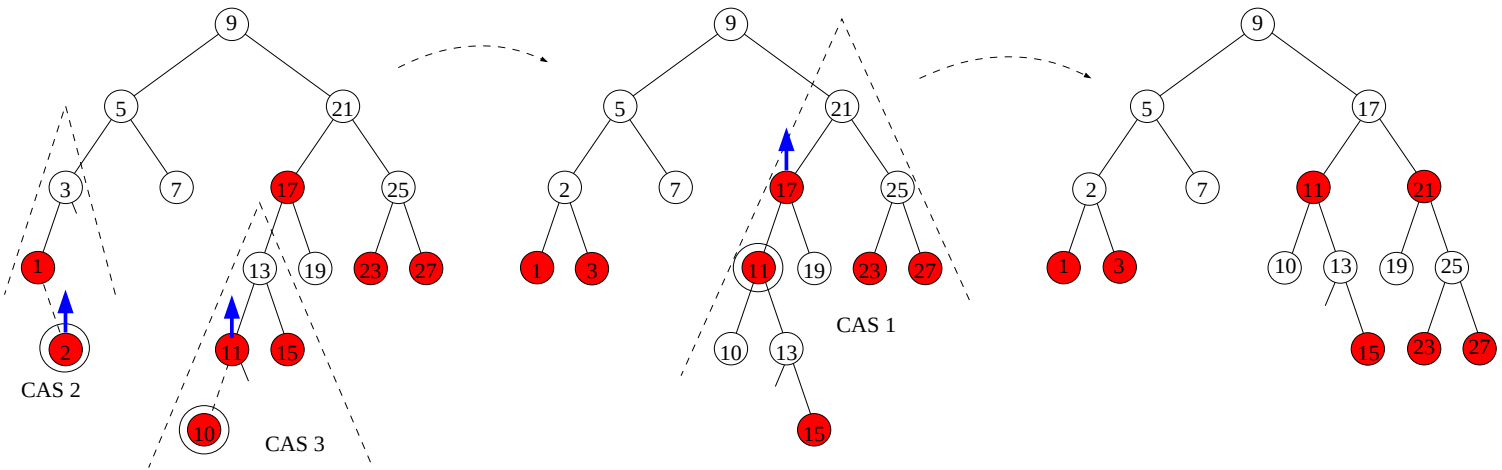


x intern $\Rightarrow$ rotació doble



però ara hem fet l'avi **roig** $\Rightarrow$ recursió

Exemple: Afegir 2 i 10

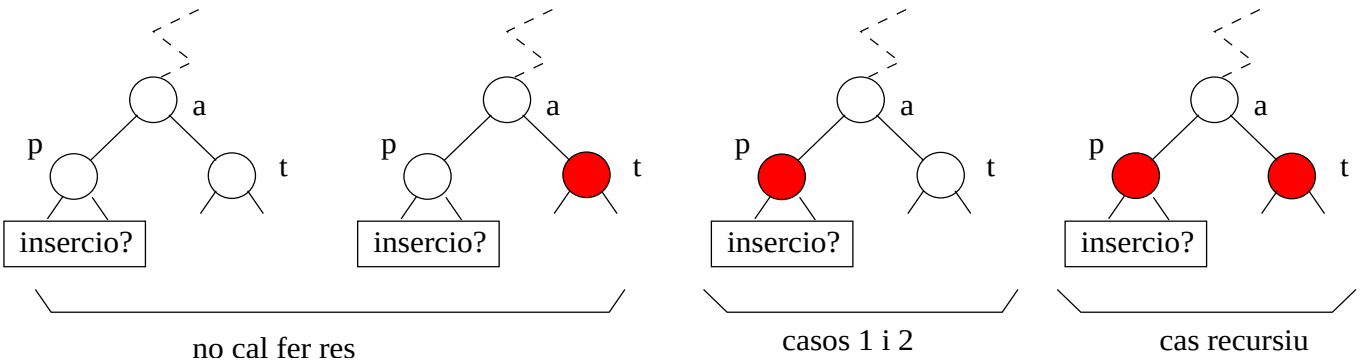


Afegir: cas pitjor

ARN: afegir descendent

El ARN es reorganitza al mateix temps que es recorre la la branca.

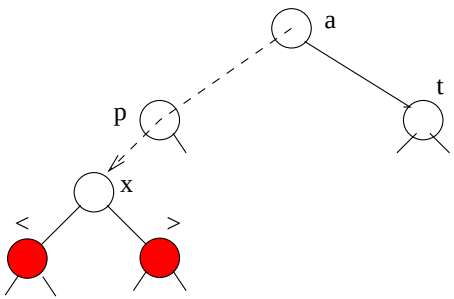
Objectiu: conseguir que quan s'inseresca la nova fulla (roja) , es puga reorganitzar l'arbre en 1 rotació com a molt



ARN: afegir descendent

Si durant el descens trobem un node negre amb dos fills rojos ho canviarem per a que els dos fills siguen negres i descendrem per un dels dos.

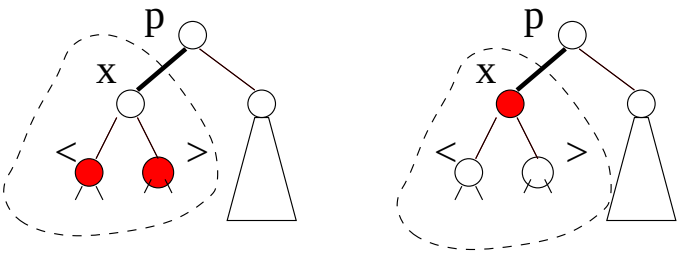
Notació:



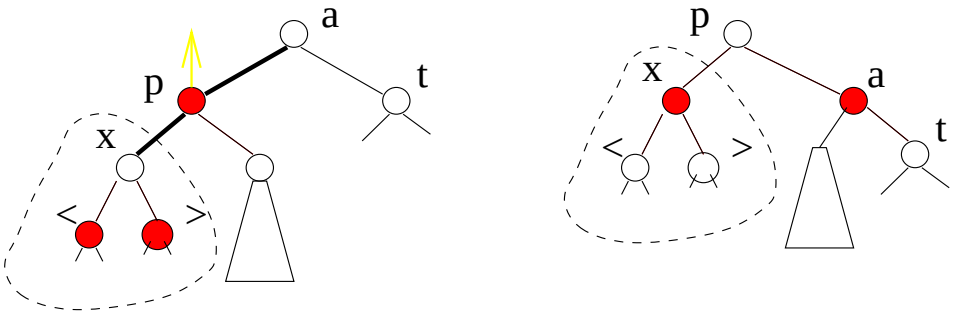
x	:	node on s'acaba d'arribar.
$<, >$	:	fills de x per on continuarà la cerca.
p, a, t	:	pare, avi i tio, respectivament.

ARN: afegir descendent

cas 1: pare negre $\Rightarrow$ canvi de colors

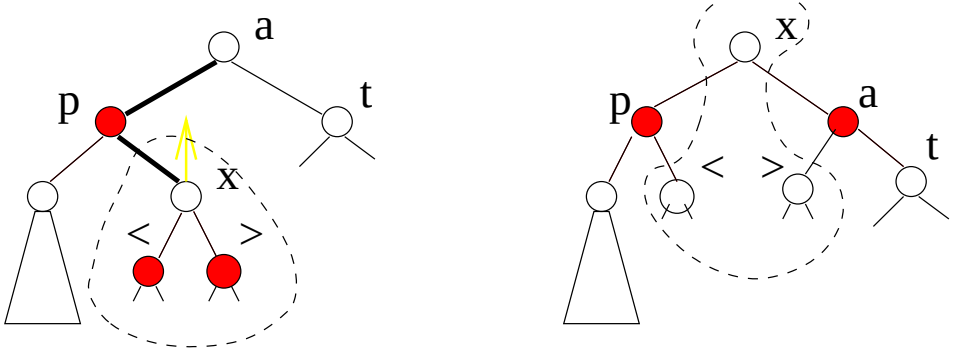


cas 2: pare roig, tio negre, x extern $\Rightarrow$ rotació simple



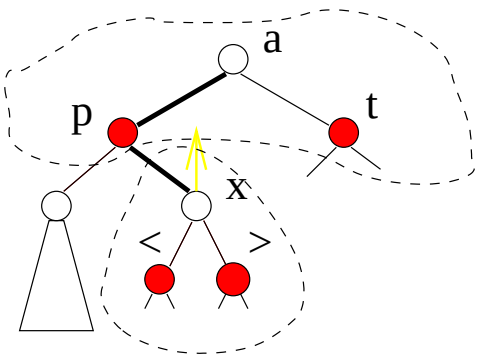
ARN: afegir descendent

cas 3: pare roig, tio negre,
x intern $\Rightarrow$ rotació doble

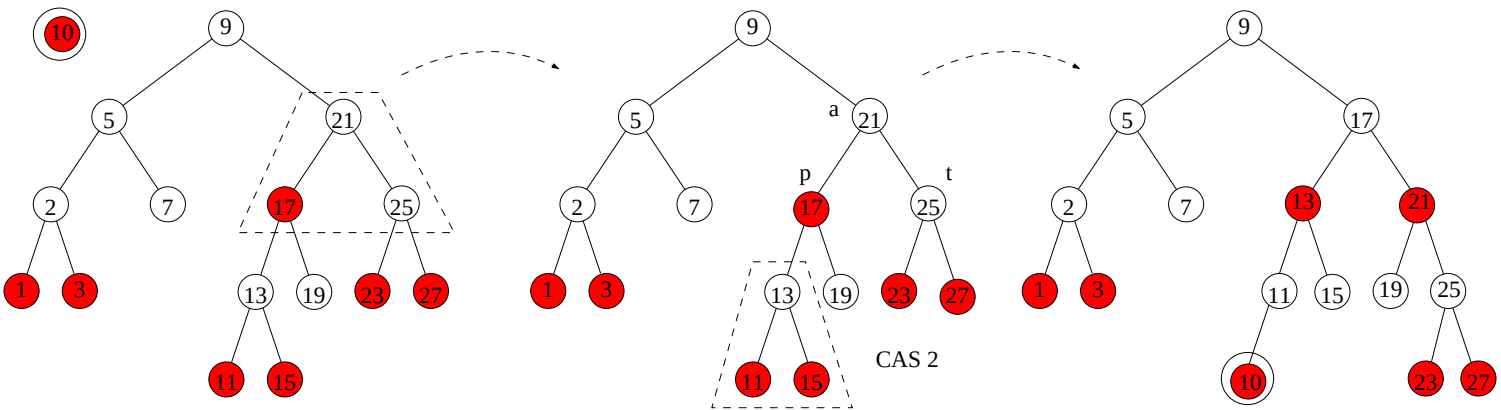


ARN: afegir descendent

cas 4: pare roig, tio roig $\Rightarrow$ impossible!

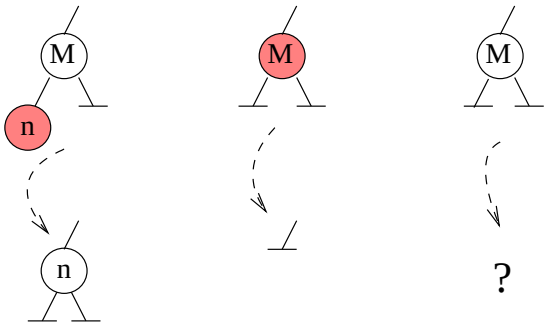


ARN descendent: afegir 10



ARN: eliminar

La eliminació en ABC sempre es produeix (realment) en nodes que tenen algun subarbre buit.



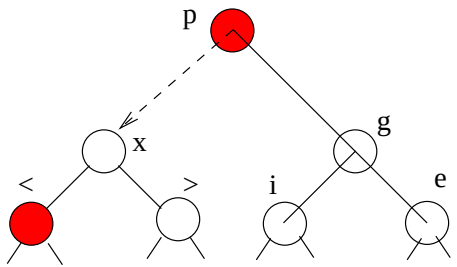
En els dos primers casos, l'arbre resultant de l'eliminació és un ARN. Si és negre sense fills, l'eliminació requereix modificar altres parts de l'arbre.

ARN: eliminar

Estratègia descendent: cal conseguir que el node a eliminar siga roig.

Mentre es cerca l'element es fa roig el node actual

Notació:



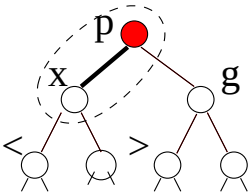
x	:	node on s'acaba d'arribar.
$<, >$	:	fills de x.
p, g, i, e	:	pare, germà, i nebots intern i extern, respectivament.

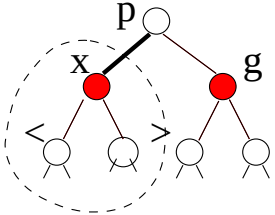
ARN: eliminar

El problema el tenim quan arribem a un node negre. Suposarem que en aquesta situació el pare hem conseguit fer-lo roig en l'operació anterior

suposem p roig!!

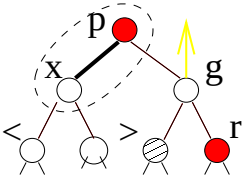
cas 1: fills de x i fills de g negres $\Rightarrow$ canvi de colors

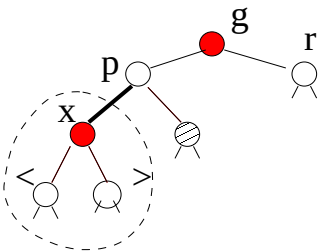




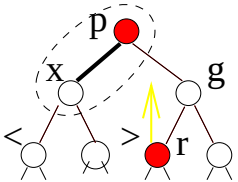
ARN: eliminar

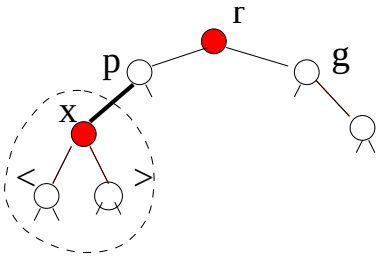
cas 2: fills de x negres, nebot extern roig, intern indiferent $\Rightarrow$ rotació simple





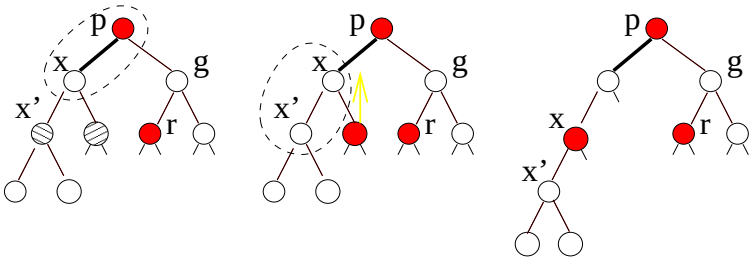
cas 3: fills de x negres, nebot intern roig, extern indiferent (negre, o cas anterior!) $\Rightarrow$ rotació doble





ARN: eliminar

cas 4: algun fill de x roig $\Rightarrow$ descendir

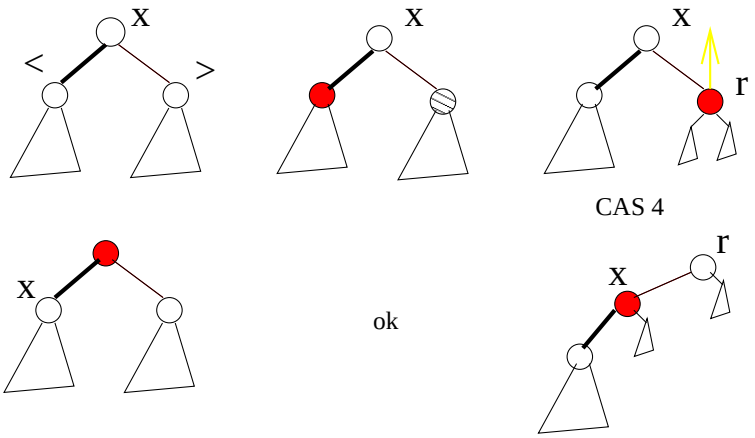


Si el següent node visitat, x', és roig, perfecte!

Si x' és negre (el seu germa roig), amb una rotació simple sobre el germà es recupera la situació inicial però un nivell més avall. (r i el seu germà poden ser qualsevol cosa i no canvien!)

> Si s'apliquen aquestes modificacions mentre es recorre la branca, el node que s'eliminarà serà sempre roig

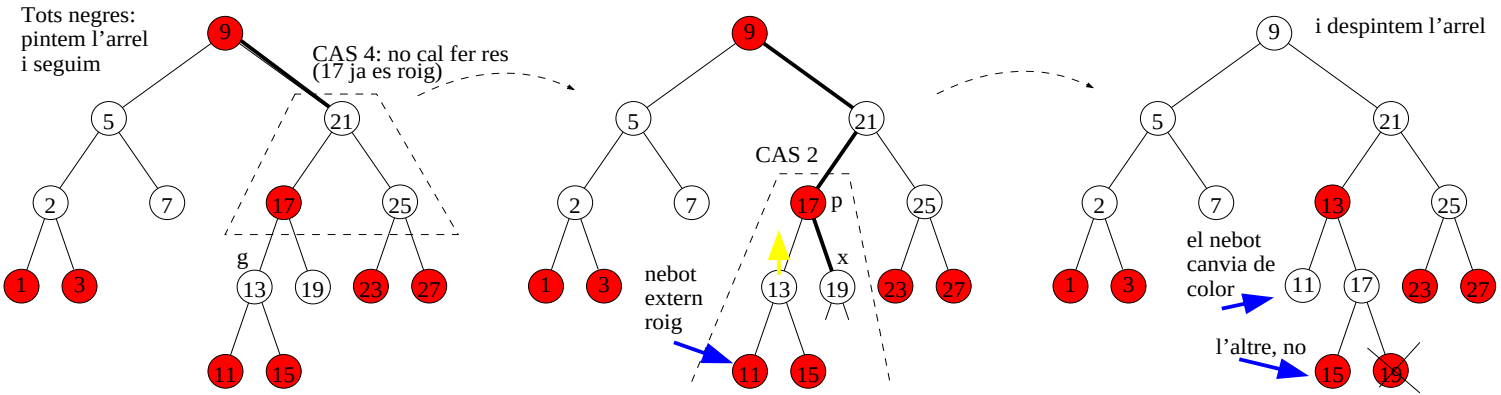
ARN: eliminar



Situació inicial: a) tots dos negres, b) següent node roig, c) següent node negre.

a) es fa l'arrel roja i es baixa un nivell
b) no cal fer res
c) es com el CAS 4 anterior

ARN: eliminar 19



Arbres Binaris de Cerca Desplegats (ABCD)

Arbres Desplegats (o Eixamplats) en anglés Splay Trees

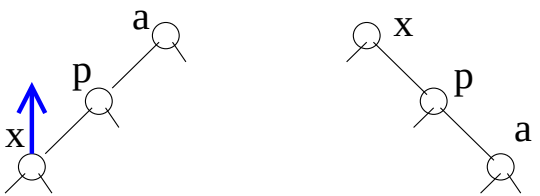
IDEA: cada vegada que s'accedeix a un node, x, es modifica el ABC de forma que x està en l'arrel i l'arbre està (un poc) més equilibrat.

Estructura autoajustable

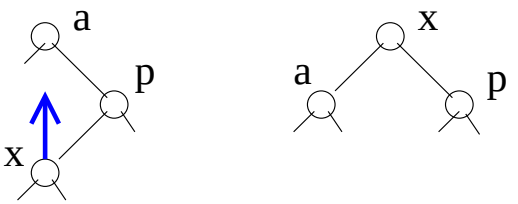
Aquesta operació de remodelació bàsica rep el nom de **eixample**, **desplegament** o **splay** i es pot implementar mijantçant una seqüència de rotacions molt similars a les dels AVL i els ARN.

ABCD: operació `rotar`(x)

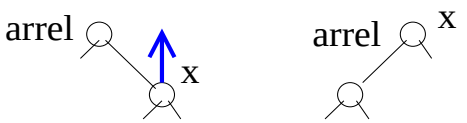
rotació esquerra-esquerra (o dreta-dreta)



rotació dreta-esquerra (o esquerra-dreta)



rotació dreta (o esquerra)



ABCD: operació $\texttt{splay}(x, a)$

Donat un arbre a i un node x (identificarem x amb el seu subarbre associat quan interesse), definim l'operació desplegat de x respecte de a:

$$\texttt{splay}(x) = \texttt{mentre } x \neq \text{arrel}(a)^* \texttt{ fer rotar}(x)$$

L'operació es pot efectuar sobre un arbre o sobre qualsevol subarbre. La comprovació es fa respecte de l'arrel del (sub)arbre corresponent!

ABCD: operacions

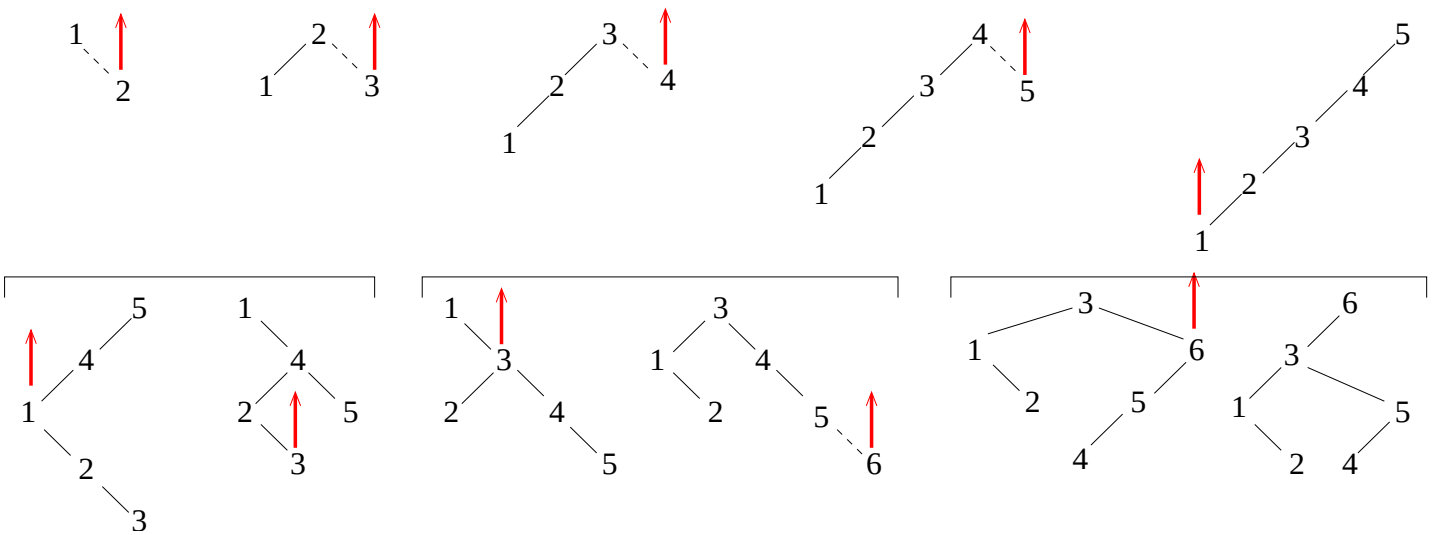
trobar Com en els ABC. Després es fa splay sobre el node trobat o sobre el més pròxim (el pare del subarbre buit a què s'arribe).

afegir Com en els ABC. Després es fa splay sobre el nou node.

Les dues operacions recorren una branca sencera dues vegades. Com a resultat de `splay`, la longitud de la branca afectada es modificarà.

Com que els arbres no estan equilibrats, els dos recorreguts tenen un cost lineal i la profunditat de l'arbre pot seguir sent lineal!

ABCD: Exemple



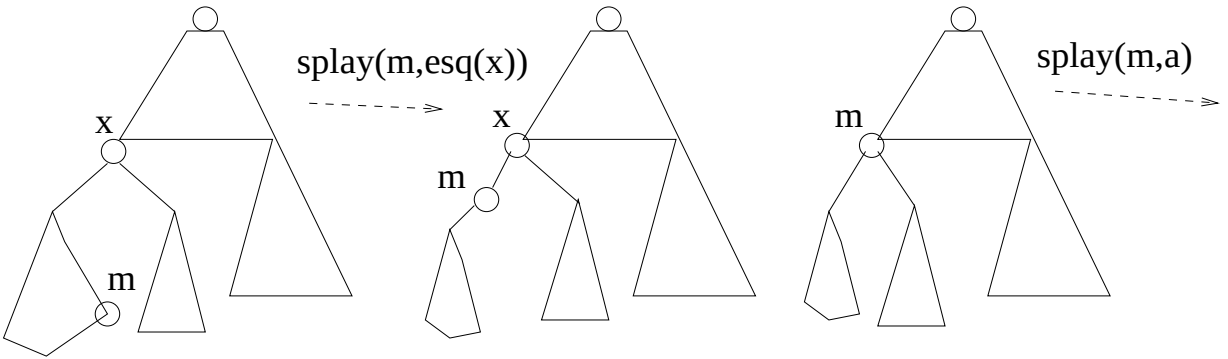
afegir(1), afegir(2), afegir(3), afegir(4), afegir(5)

trobar(1), trobar(3), afegir(6).

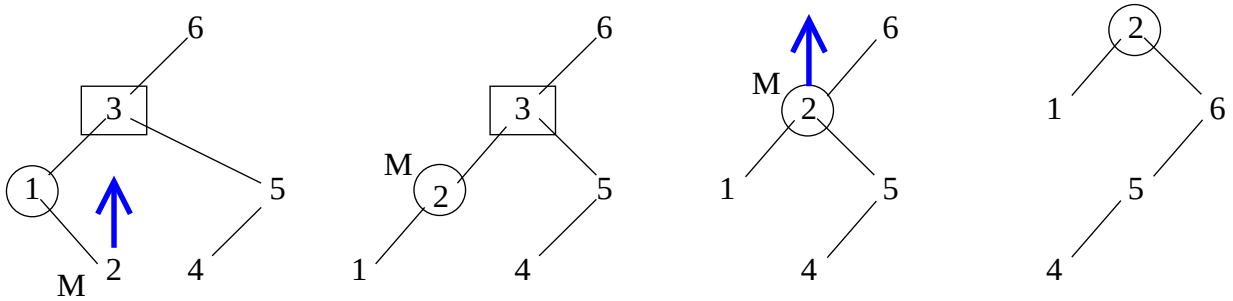
ABCD: Eliminar

Una vegada localitzat x i el màxim de esq(x), M, es fa `splay` de M respecte de esq(x).

Com que M penjarà de x i no tindrà fill dret, es pot eliminar x i es fa `splay` de M respecte de tot l'arbre.



ABCD: eliminar 3



ABCD: anàlisi de `splay`

Totes les operacions tenen un cost (real) proporcional al cost de l'operació splay (lineal en el pitjor cas).

`splay` consisteix en una seqüència de rotacions que modifiquen l'estructura de l'arbre, no el nombre de nodes.

Cost amortitzat: potencial?

ABCD: anàlisi de `splay`

idea feliç: Si assignem un **rang** a cada node (subarbre) en funció de la seua talla, podem definir el potencial (de l'arbre) com el sumatori dels rangs dels seus nodes. Per exemple, si $r(x) = |x|$:

```
-5-4-3-2-1
```

$$\phi(-) = \Sigma_{i=1}^{5}\, r(i) = 1+2+3+4+5 = 15$$

```
-1-4-5
   |
   2-3
```

$$\phi(-) = \Sigma_{i=1}^{5} = r(i) = 5+2+1+4+1 = 13$$

El potencial serà menor com més equilibrat siga l'arbre. Si definim el rang com una funció monòtona creixent del nombre de nodes també es complirà.

ABCD: anàlisi de `splay`

En particular: rang$(x) = r(x) = \lg |x|$ = logaritme del nombre de nodes en el subarbre l'arrel del qual és x. I el potencial

$$\phi(a) = \sum_{x \quad \text{subarbre de } a} r(x)$$

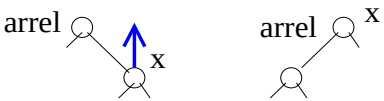
ABCD: anàlisi de `splay`

Es pot analitzar l'increment de potencial que produeix cada tipus de rotació sobre x i expressar-lo en funció d'increments de rang.

x' és el subarbre associat a x després de la rotació

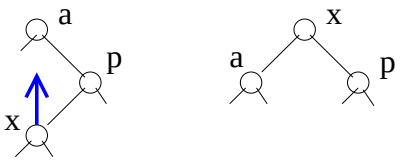
$\Delta r(x) = r(x') - r(x)$

Rotació simple:



$$\Delta\phi_R = r(x') - r(x) + \underbrace{r(p') - r(p)}_{|p'|<|p| \Rightarrow r(p')<r(p)} \leq \Delta r(x) < 3\Delta r(x)$$

ABCD: anàlisi de `splay`



Rotació doble:

$$\Delta\phi_R = \underbrace{r(x')}_{=r(a)} - r(x) + r(p') \underbrace{-r(p)}_{\leq -r(x)} + r(a') - \underbrace{r(a)}_{=r(x')}$$

$$\left(\begin{array}{rcl} |x'| = |a| & \Rightarrow & r(x') = r(a) \\ |p| \geq |x| & \Rightarrow & r(p) \geq r(x) \end{array} \right.$$

$$\Delta\phi_R \leq r(p') + r(a') - 2r(x)$$

ABCD: anàlisi de `splay`

$$\left(\begin{array}{rcl} |p'| + |a'| \leq |x'| & \Rightarrow & r(p') + r(a') \leq 2r(x') - 2 \\ b + c \leq d & \Rightarrow & \lg b + \lg c \leq 2 \lg d - 2, \forall b, c > 0 \end{array} \right.$$

$$\Delta\phi_R \leq 2r(x') - 2r(x) - 2 \leq 3\Delta r(x) - 2$$

Conclusió:

$$\Delta\phi_R \leq \left\{ \begin{array}{cl} 3\Delta r(x) - 2 & \text{rotació doble} \\ 3\Delta r(x) & \text{rotació simple} \end{array} \right.$$

Splay: cost amortitzat

• increment de potencial degut a splay(x) (k rotacions):

Siguen x_i els diferents subarbres dels quals va sent arrel x mentre puja a l'arrel ($x'_i = x_{i+1}$).

$$\Delta\phi_s \leq \sum_{i=1}^{k}[3\Delta r(x_i)) - 2] \underbrace{+2}_{(*)} =$$

$$= \sum_{i=1}^{k}[3(r(x_{i+1}) - r(x_i)) - 2] + 2 = 3r(a) - \underbrace{3r(x_1)}_{\geq 0} - 2k + 2 \leq 3r(a) - 2k + 2$$

(*) si l'última rotació és simple

Splay: cost amortitzat

Comptem un pas per cada rotació (simple)

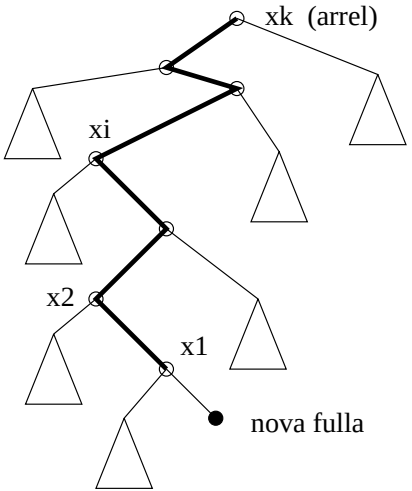
$$\hat{c_s} = c_s + \Delta\phi_s \leq \left\{ \begin{array}{rcl} 2k & + & 3r(a) - 2k \\ 2k \underbrace{-1}_{(*)} & + & 3r(a) - 2k \underbrace{+2}_{(*)} \end{array} \right\} \leq 1 + 3\lg n$$

El cost amortitzat de l'operació `splay` és logarítmic.

Splay (afegir): cost amortitzat

- Cost amortitzat de afegir:

Serà el mateix sempre que afegir (abans de fer splay) no provoque un augment del potencial major que logarítmic.



Afegir recorre una branca de l'arbre fins insertar una nova fulla. Siguen x_i i x'_i l'i-èssim node de la branca abans i després d'afegir una nova fulla.

Splay (afegir): cost amortitzat

$$|x'_i| = |x_i| + 1 \leq |x_{i+1}| \Longrightarrow r(x'_i) \leq r(x_{i+1}) \ (*)$$

L'increment de potencial al llarg de la branca:

$$\Delta\phi = \sum_{i=1}^{k}(r(x'_i) - r(x_i)) \overset{(*)}{\leq} \sum_{i=1}^{k}(r(x_{i+1}) - r(x_i)) = r(a) - \underbrace{r(x_1)}_{>0} \leq \lg n$$

Qualsevol altra operació no incrementa el potencial per tant tindrà un cost amortitzat proporcional al de splay també.

Arbres Binaris de Cerca No Binaris

Què passaria si definirem arbres de cerca ternaris o m-aris en general?

- profunditat?
- operacions?
- manteniment de l'equilibri?

Exercici:

S'obtendria algun tipus d'avantatge? Quin seria el valor òptim de m?

Arbres B: definició

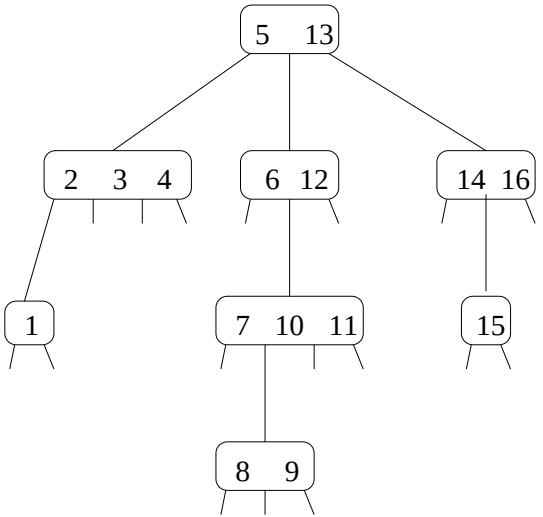
Un m'**-node** en un arbre m-ari és un node amb $m'-1$ claus, $\{k_i\}_{j=1}^{j=m'-1}$, i m' fills (subarbres), $\{s_i\}_{j=1}^{j=m'}$, de manera que es compleix

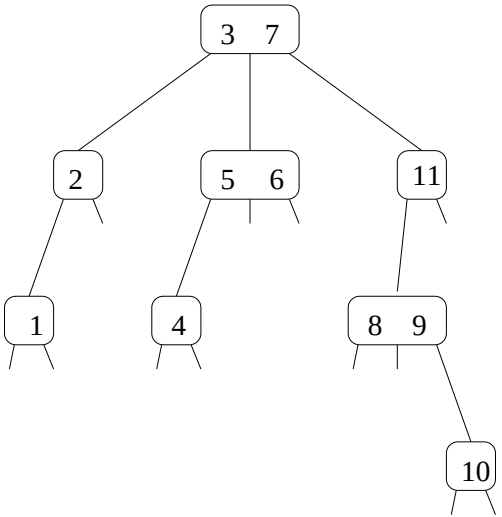
$$k_{i-1} < s_i < k_i, \qquad i = 1, 2, \ldots m'$$

sent $k_0 = -\infty$ i $k_{m'} = +\infty$.

Un arbre B d'ordre m és un arbre m-ari format per m' nodes tals que $\lceil \frac{m}{2} \rceil \leq m' \leq m$.

Arbres B: definició





Arbres B equilibrats

Un arbre B es diu equilibrat quan tots els subarbres buits estan a la mateixa profunditat.

El manteniment d'equilibri en els arbres B se simplifica mijantçant canvis d'ordre dels nodes implicats

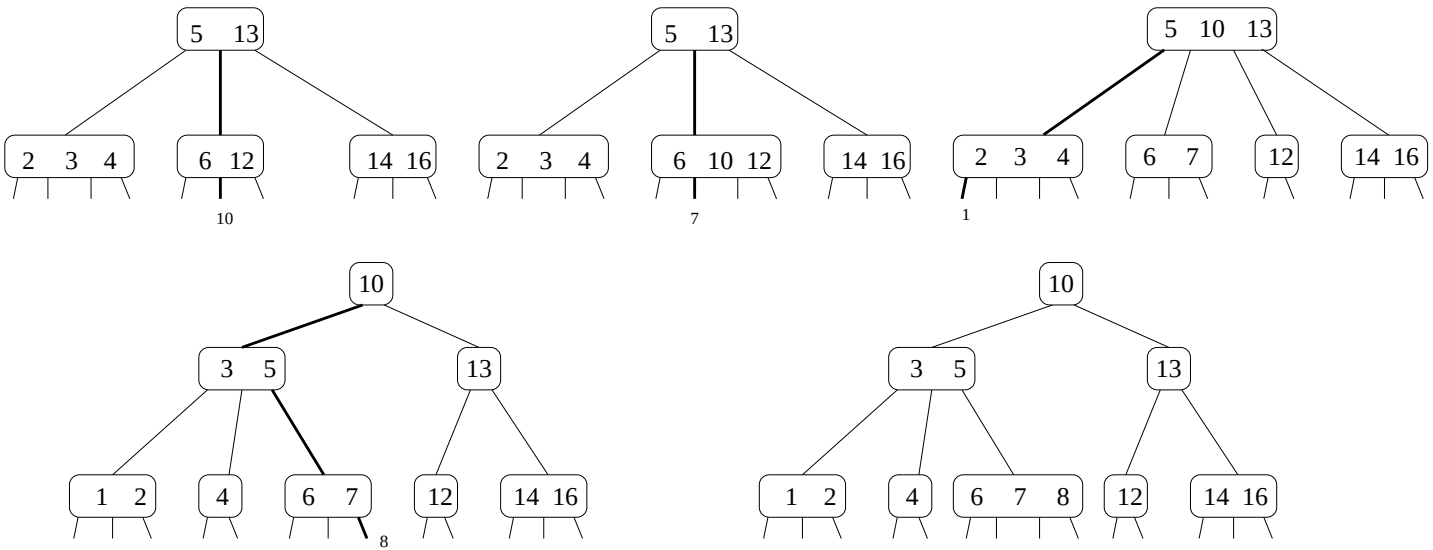
- Arbres B d'ordre 2 = ABC
- Arbres B d'ordre 3 (Arbres 2-3)
- Arbres B d'ordre 4 (Arbres 2-3-4), etc.

Moltes vegades la condició d'equilibri se suposa implícita en la d'arbre B.

Arbres B: afegir

- Es localitza el node d'on hauria de penjar x com a fulla i s'adjunta a les claus del node (hi ha una de més).
- Si és un $(m+1)$-node (m claus), es parteix en un $\lceil \frac{m+1}{2} \rceil$-node i un $\lfloor \frac{m+1}{2} \rfloor$-node ($m-1$ claus en total) i la clau central puja un nivell.
- Si en pujar la clau el pare es converteix en un $(m+1)$-node, es parteix també el pare de la mateixa manera.
- Si es parteix l'arrel, es crea un nou 2-node per a la clau que puja que serà la nova arrel.

Arbres B: afegir (exemple)



afegir(10), afegir(7), afegir(1), afegir(8)

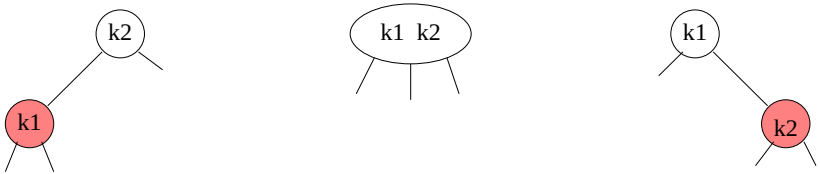
Arbres B: eliminar

La eliminació és més complicada (hi ha més variants)

- Si s'elimina una clau d'un m'-node on m' no és el mínim, una clau dels subarbres implicats ocuparà el lloc de l'eliminat i es trasllada el problema a un subarbre.
- Si m' és el mínim, el lloc cal implicar un node d'un subarbre germà.
- En el cas que el node del subarbre germà també siga mínim, es fa desapareixer l'arrel.

Relació amb els arbres roig-negre

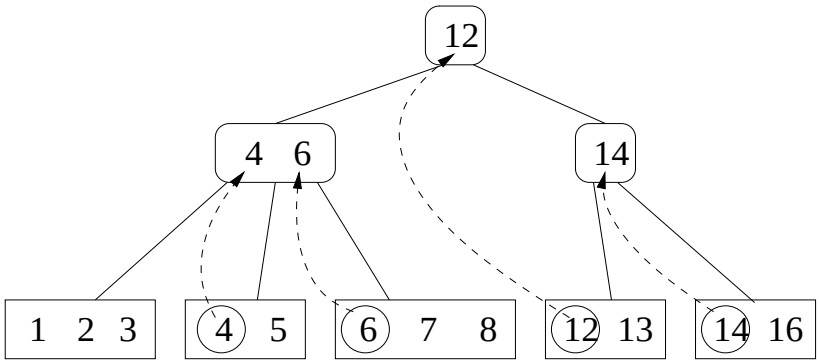
Hi ha una relació directa entre els arbres 2-3 i els arbres roig negre:



Els arbres roig-negre també estan perfectament equilibrats quant a l'altura negra.

Arbres B: implementació

Es pot emmagatzemar tota la informació associada a les claus **només** en les fulles.



Totes les claus es troben en les fulles. Com a molt cada clau apareix dues vegades. Variants B^* i B^+.