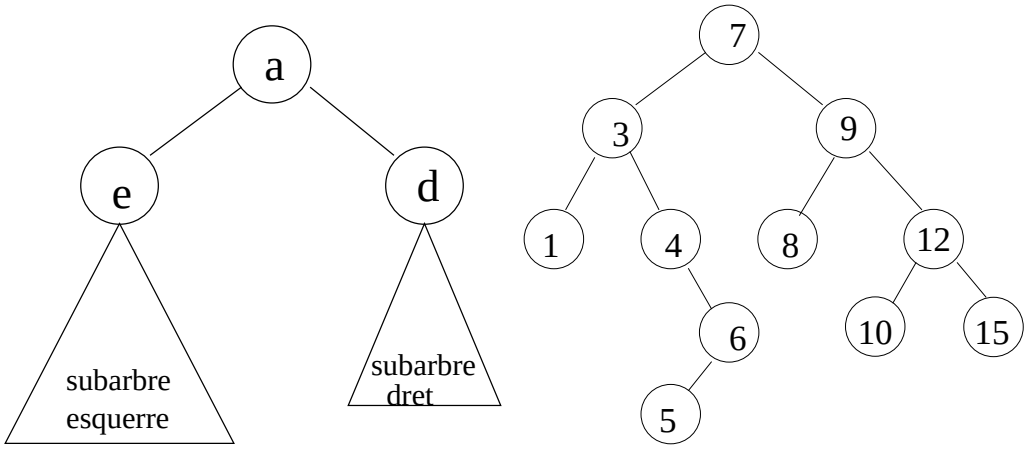
Arbres Binaris de Cerca (ABC)

$\mathcal{A}_\delta$= 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)).

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).

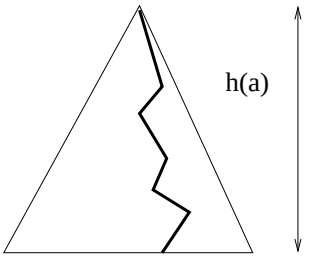
Algorisme:

- 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 //
M

```
  si a ≠ ∅_A aleshores
        si x = arrel(a) aleshores trobar← 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
```



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

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

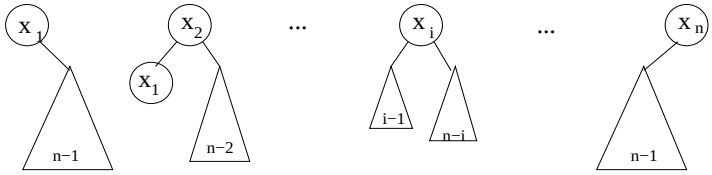
Trobar. Anàlisi per casos

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.

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 la operció trobar.

Afegir. Anàlisi amortitzada

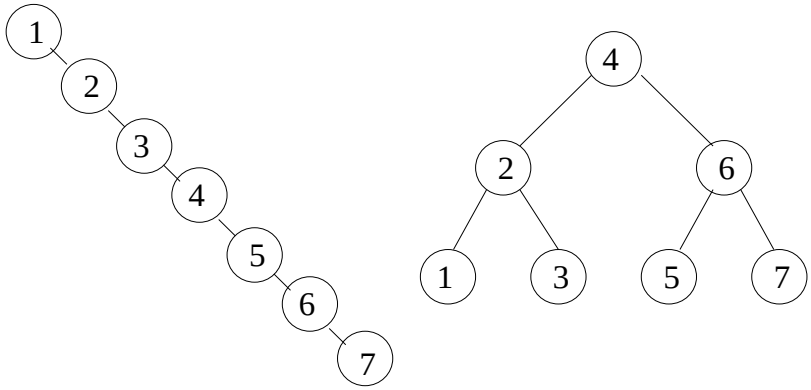
ABC amb afegir i trobar com a úniques operacions.

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

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

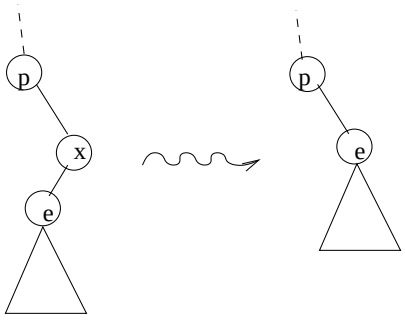
I(1), I(2), ...I(7)

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):



- Si el node a eliminar té dos fills cal trobar l'element mínim del subarbre dret (que serà una fulla) 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
        }


        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

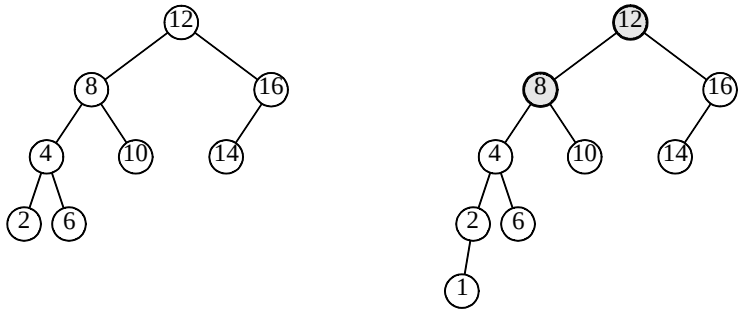
$$N(h) = N(h-1) + N(h-2) + 1$$

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

 187



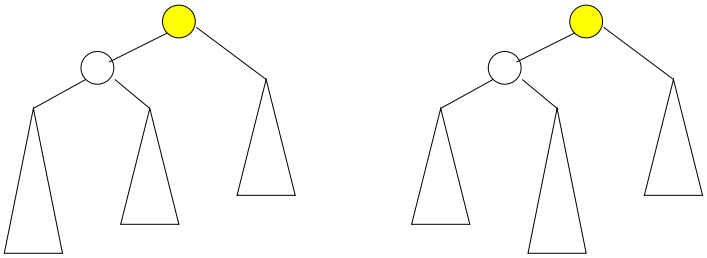
Two binary search trees: the left tree is an AVL tree, but the right tree is not (unbalanced nodes are darkened)

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:

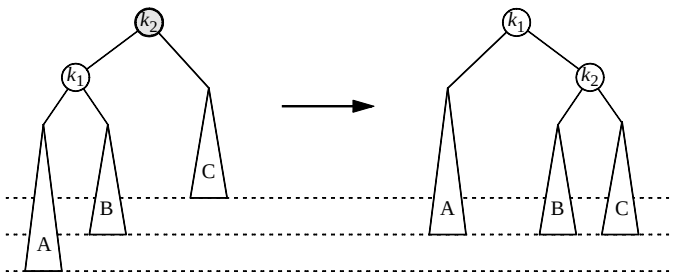
Podem distingir entre dos casos (i els seus simètrics) en funció de la forma del subarbre més profund.



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).

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

AVL:rotacions simples



Single rotation to fix case 1

F. J. Ferri, **Dept. d'Informàtica** *Universitat de València*

AVL:rotacions dobles

AVL: implementació

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

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

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

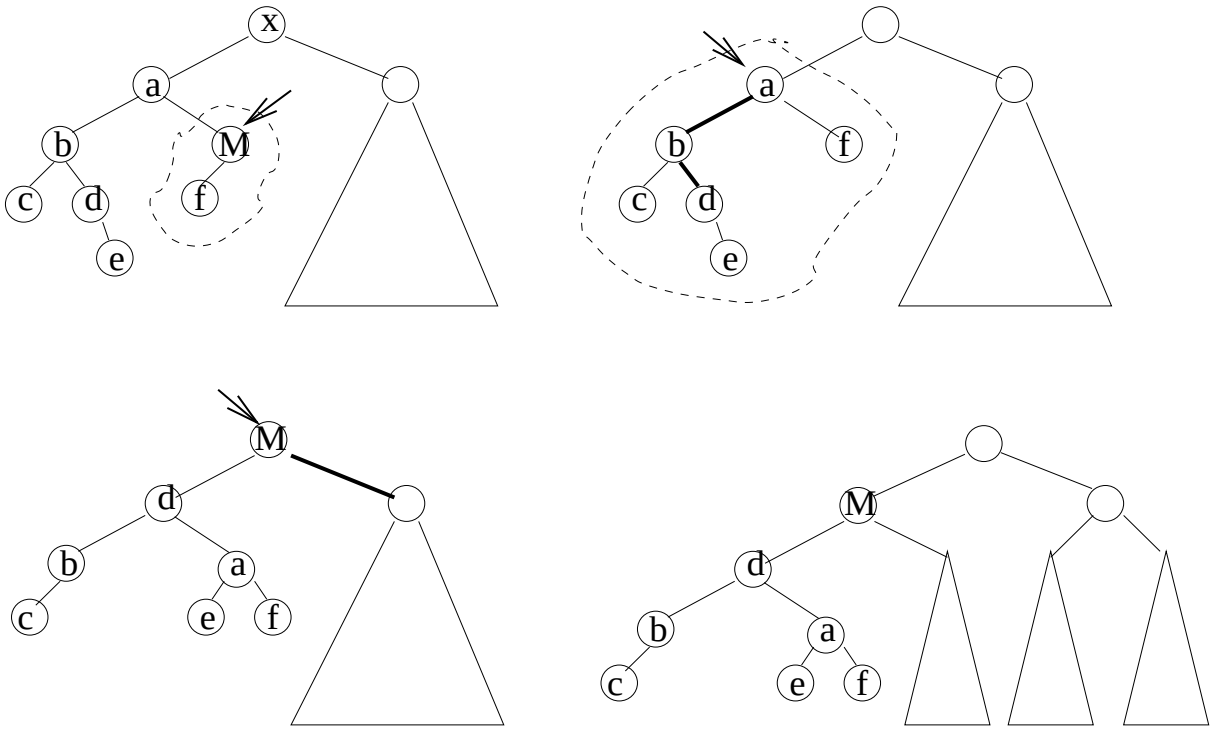
```
private static AvlNode rotateWithLeftChild( AvlNode k2 )
{
    AvlNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
    k1.height = max( height( k1.left ), k2.height ) + 1;
    return k1;
}

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

AVL: eliminar

Algorisme eliminar ($a : \mathcal{A}_\delta$**,** $x : \delta$**)**
//Elimina x en un arbre AVL
  si $a = \emptyset_{\mathcal{A}}$ aleshores
    ERROR: no trobat
  si no si $x <$*arrel(a)* aleshores
    eliminar$(esq(a), x)$
    reorganitzar-esq(a)
  si no si $x >$*arrel(a)* aleshores
    eliminar$(dre(a), x)$
    reorganitzar-dre(a)
  si no *//trobat x*
    $m \leftarrow$ *borramax(esq*(a)*)*
    arrel$(a) \leftarrow m$
    reorganitzar-dre(a)
    fsi
feliminar

Algorisme borramax ($a : \mathcal{A}_\delta$**,** $x : \delta$**) :** δ
  si *dre*$(a) = \emptyset_{\mathcal{A}}$ aleshores
    borramax$\leftarrow$ *arrel*(a)
    $a \leftarrow$ *esq*(a)
    $h(a) \leftarrow h(a) - 1$
  si no
    borramax$\leftarrow$ *borramax(dre*(a)*)*
    reorganitzar-esq(a)
  fsi
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 la 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).

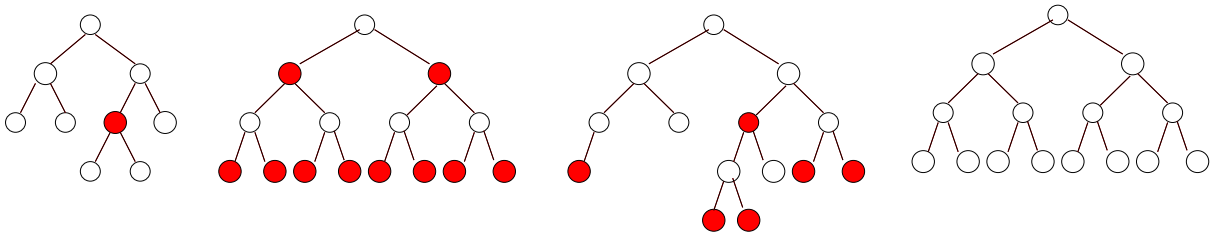
P2) per a tot subarbre x es compleix
color(x)=roig $\Rightarrow$
color(esq(x))=color(dre(x))=negre
(nodes rojos han de tindre fills negres)

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:

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

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

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

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

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).

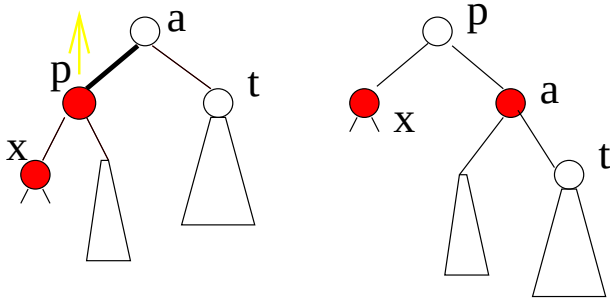
Afegir una fulla roja pot produir que son pare incomplesca P2 (si és roig).

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

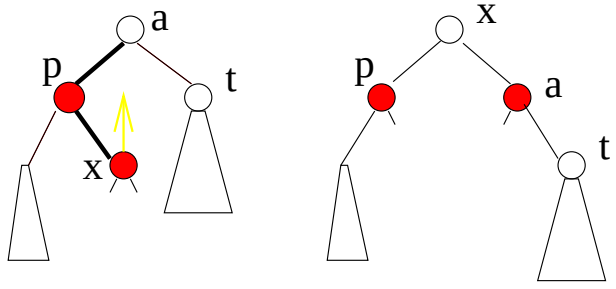
ARN: afegir ascendent

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

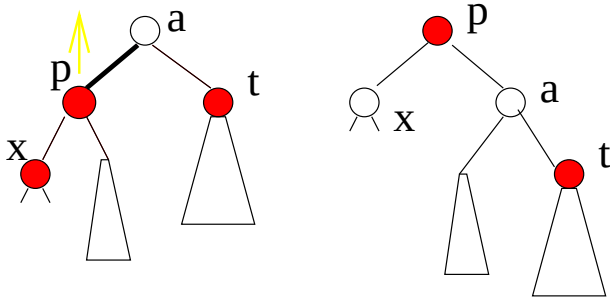
cas 1: tio negre, x extern $\Rightarrow$ rotació simple



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



cas 3: tio roig, x indiferent (*) $\Rightarrow$ rotació simple



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

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 el seu pare siga negre

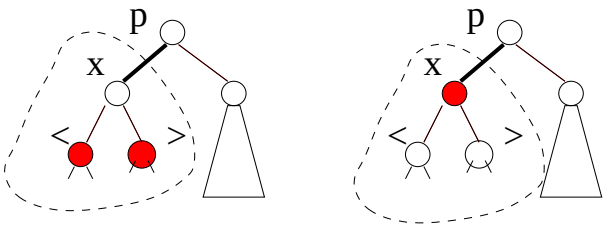
Si durant el descens trobem un node negre amb dos fills rojos ho canviarem per a que els dos fills siguen negres i descendrem per un dels dos.

Notació:

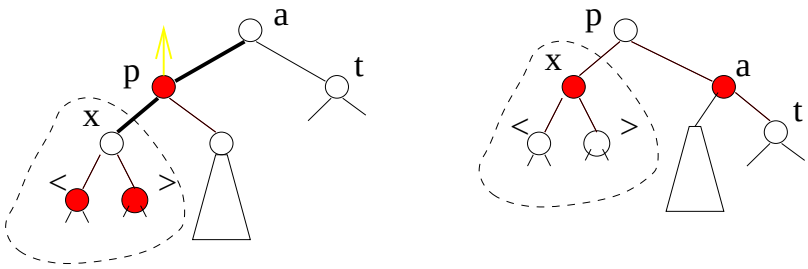
x	:	node on s'acaba d'arribar.
$<, >$	:	fills de x per on continuarà la cerca.
p, a, t	:	pare, avi i tio, respectivament.

ARN: afegir descendent

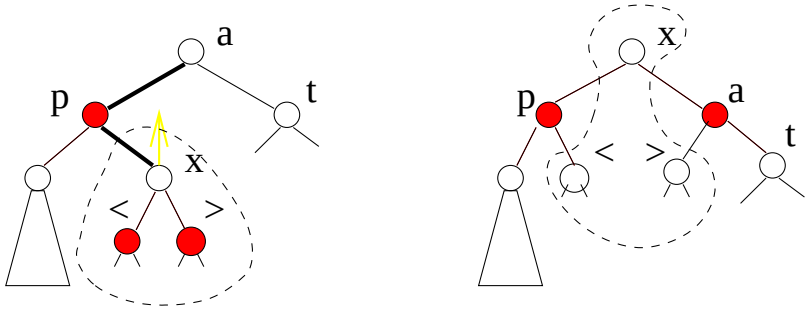
cas 1: pare negre
$\Rightarrow$ canvi de colors



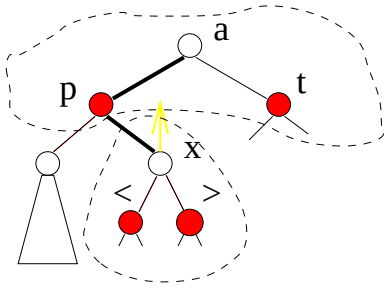
cas 2: pare roig, tio negre, x extern
$\Rightarrow$ rotació simple



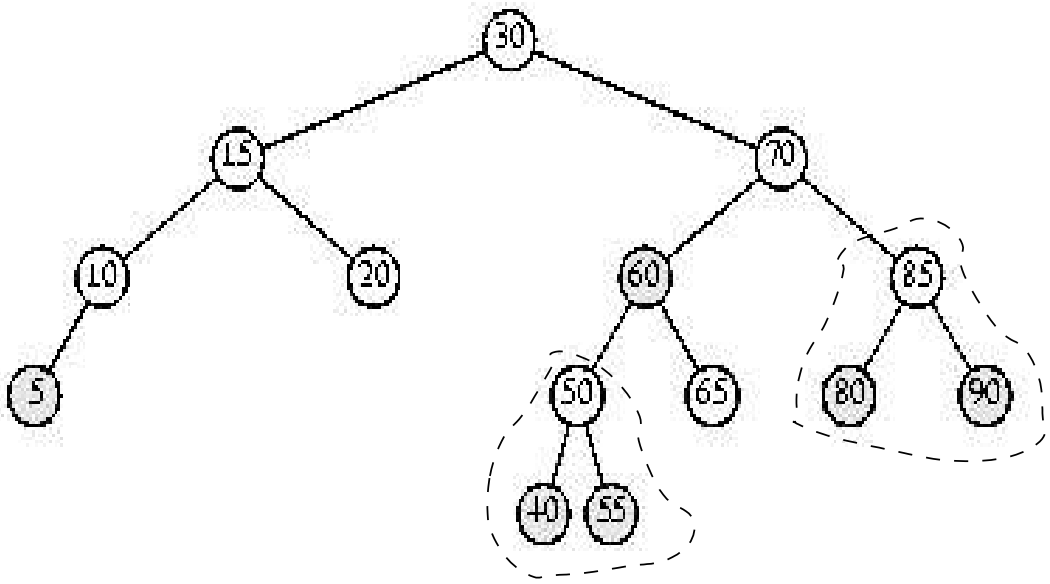
cas 3: pare roig, tio negre, x intern
$\Rightarrow$ rotació doble



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



ARN: afegir (exemple)



ARN: eliminar

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

Si el node que s'elimina és roig el resultat és un ARN

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.

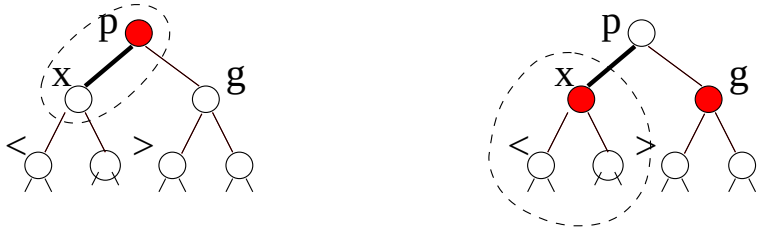
p, g : pare i germà, respectivament.

El problema el tenim quan arribem a un node negre. Suposarem que en aquesta situació el pare hem conseguit fer-lo roig (*)

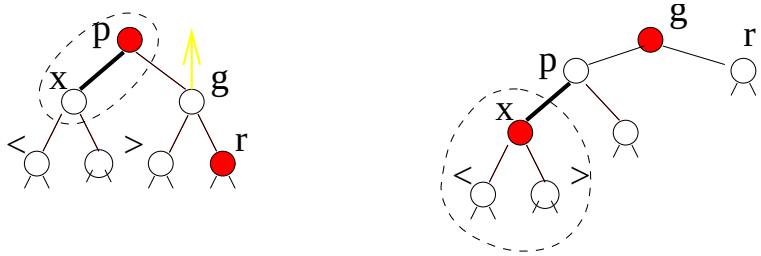
ARN: eliminar

suposem p roig!!

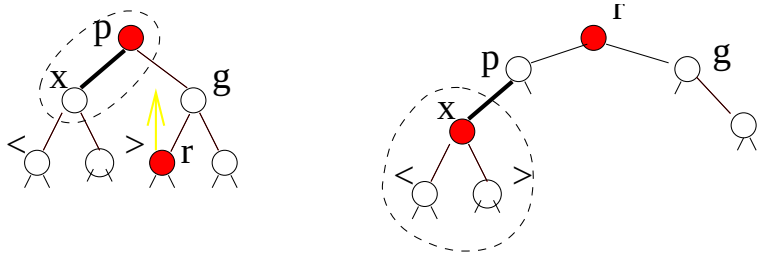
cas 1: fills de x i fills de g negres $\Rightarrow$ canvi de colors



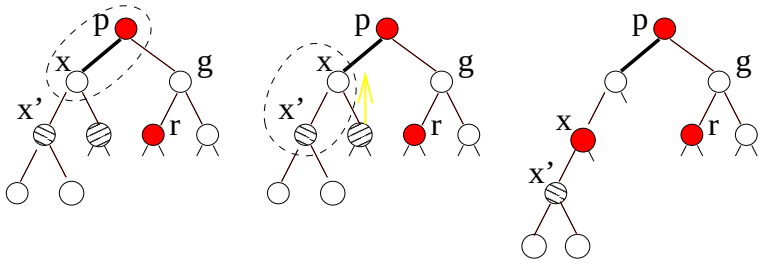
cas 2: fills de x negres, nebot extern roig $\Rightarrow$ rotació simple



cas 3: fills de x negres, nebot intern roig $\Rightarrow$ rotació doble



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

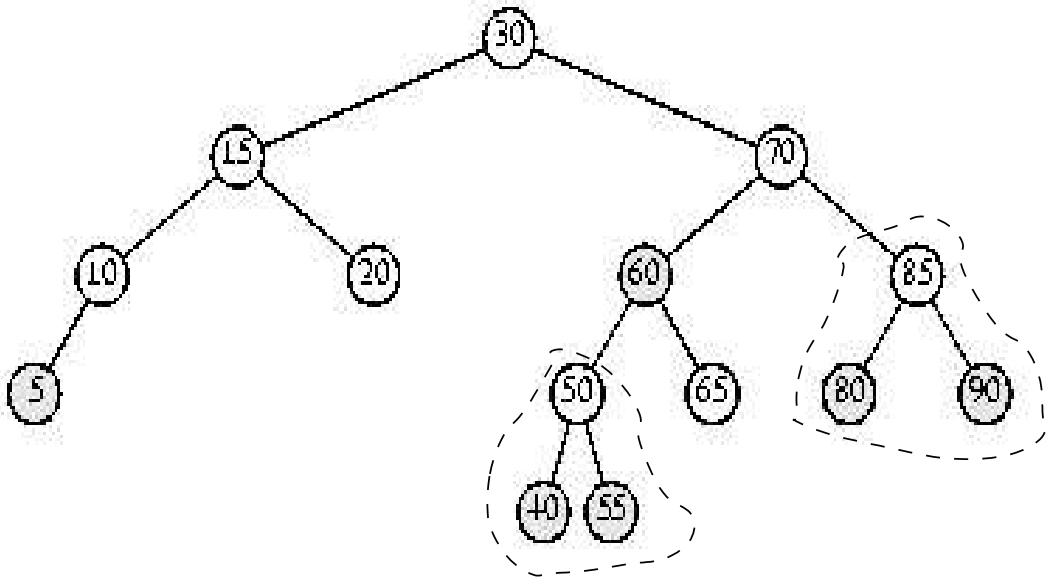


O trobem roig, o recuperem la situació inical amb 1 rotació.

ARN: eliminar (exemple)

En començar fem l'arrel roja. Si es dona el cas 3 anterior, es torna a fer negra en acabar.

Aniran aplicant-se els casos anteriors fins que s'arribe al node a eliminar que serà en eixe moment roig.



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.

Si aquesta remodelació es fa de manera que l'arbre resultant està (un poc) més equilibrat, a la llarga les operacions seran més eficients.

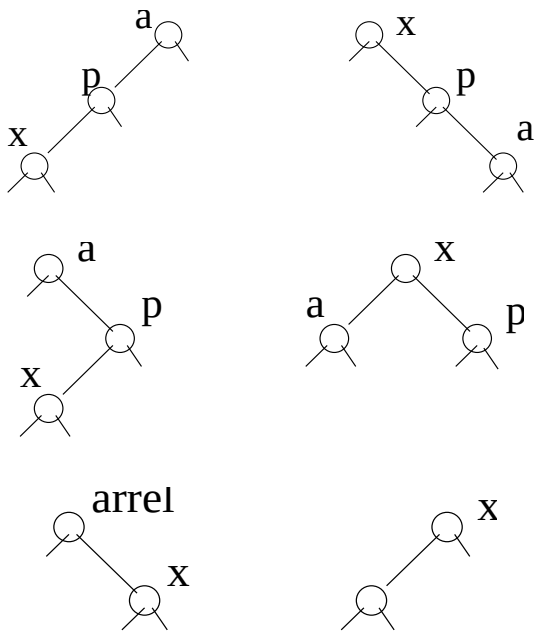
Aquesta operació de remodelació bàsica rep el nom de **eixample**, **desplegament** o **splay** i es pot implementar mijantçant una seqüència de rotacions molt similars a les dels AVL i els ARN.

ABCD: l'operació splay

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

$$\mathsf{splay}(x, a) = \texttt{mentre } x \neq \mathsf{arrel}(a) \texttt{ fer } \mathsf{rotar}(x)$$

L'operació rotar(x) fa pujar x un o dos nivells en l'arbre i presenta 3 casos:

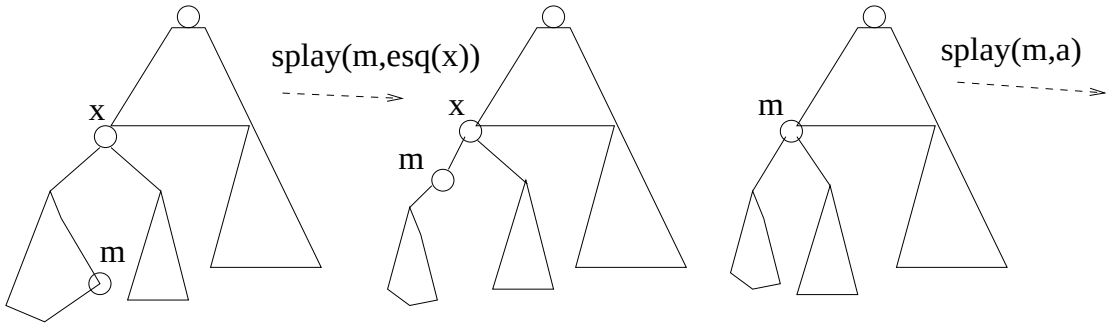


ABCD: operacions

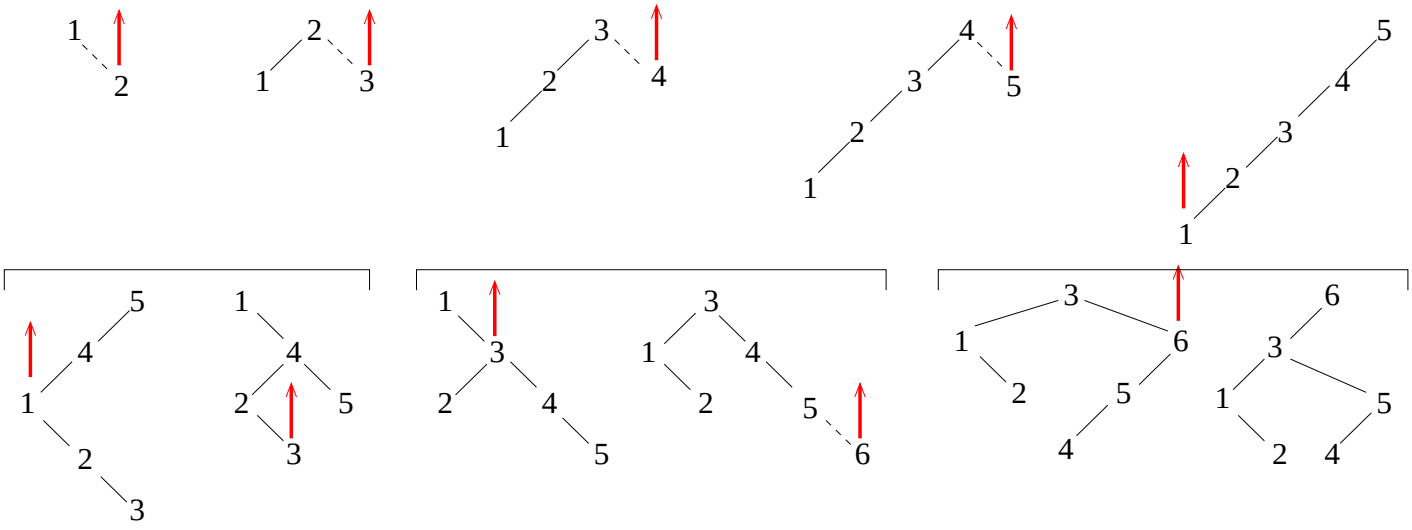
trobar Com en els ABC. Després es fa splay sobre el node trobat o sobre el més pròxim (major o menor).

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

eliminar Trobar x, després l'element anterior (o posterior). Es fa splay sobre aquest element en el subarbre esquerre (o dret) de x. S'elimina físicament x. Es torna a fer splay sobre el mateix element



ABCD: operacions (exemples)



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

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

ABCD: anàlisi

Cost en el pitjor cas: lineal (com els ABC)
Cost amortitzat: ?

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

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

Totes les operacions tenen un cost (real) proporcional al cost de l'operació splay. Aquesta operació consisteix en una seqüència de rotacions que només modifiquen el rang de dos o tres nodes.

ABCD: anàlisi

Es pot analitzar l'increment de potencial que produeix cada tipus de rotació sobre x:

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

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

$$\Delta\phi_R = \begin{cases} \Delta r(x) + \Delta r(p) + \Delta r(a) & \text{rotació doble} \\ \Delta r(x) + \Delta r(p) & \text{rotació simple} \end{cases}$$

Acte de fe:

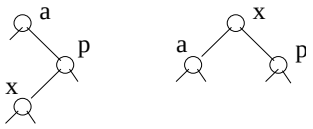
$$\Delta\phi_R \leq \begin{cases} 3\Delta r(x) - 2 & \text{rotació doble} \\ 3\Delta r(x) & \text{rotació simple} \end{cases}$$

Demostració de l'acte de fe

Rotació simple:

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

Rotació doble:



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

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

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

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

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

Splay: cost amortitzat

- Cost amortitzat de splay(x) (k rotacions):

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

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

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

(*) si la última rotació és simple

Per tant,

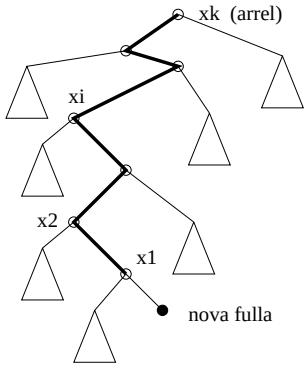
$$\hat{c_s} = c_s + \Delta\phi_s \leq \left\{ \begin{array}{ccc} 2k & + & 3r(a) - 2k \\ 2k \underbrace{-1}_{(*)} & + & 3r(a) - 2k \underbrace{+2}_{(*)} \end{array} \right\} \leq 1 + 3\lg n$$

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.



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

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?

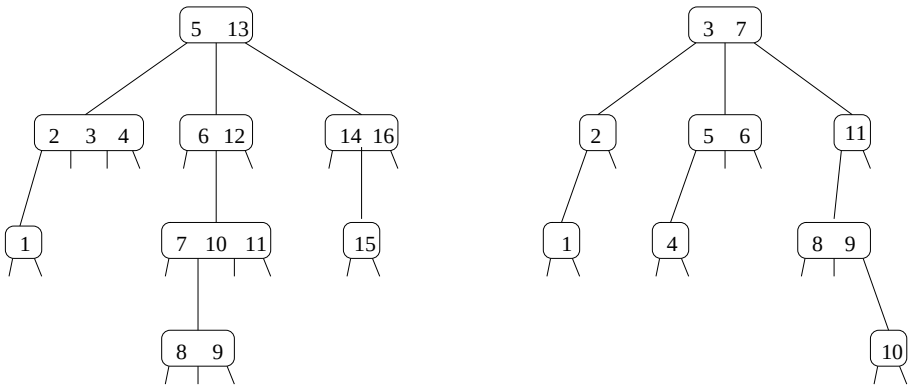
Arbres B: definició

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 (sub-arbres), $\{s_i\}_{j=1}^{j=m'}$, 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 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

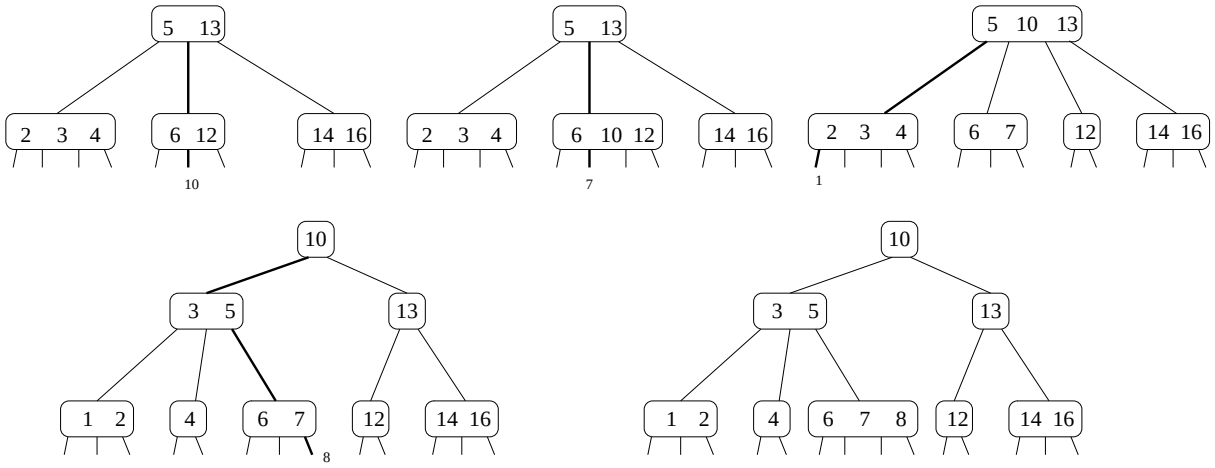
- Arbres B d'ordre 2 = ABC
- Arbres B d'ordre 3 (Arbres 2-3)
- Arbres B d'ordre 4 (Arbres 2-3-4)
- ...

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)

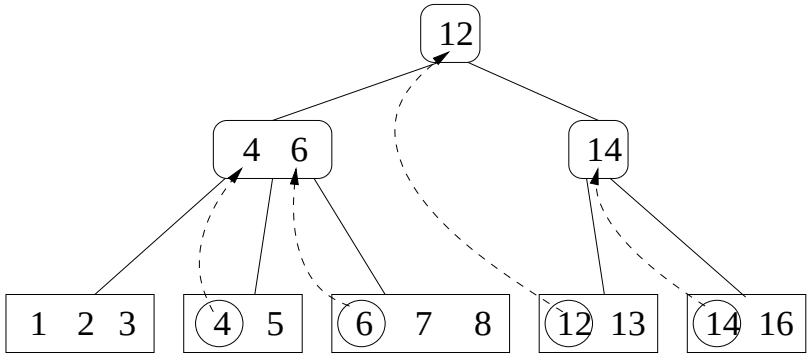
Arbres B: eliminar

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.

Arbres B: implementació

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



Aplicacions amb memòria secundària