INTRODUCTION À L'ALGORITHMIQUE

-

LES ARBRES BINAIRES DE RECHERCHE

Chargée de cours: Lélia Blin

Transparents:http://www-npa.lip6.fr/~blin/Enseignements.html

Email: lelia.blin@lip6.fr

Arbres binaires de Recherche

Les arbres binaires de recherche sont communement appelé **ABR**

Les ABR sont des structures de données

qui peuvent supporter des opérations courantes

sur des ensembles dynamiques.

ex: rechercher, minimum, maximum, prédécesseur….

Cette structure de données est maintenue:

sous forme d'arbre binaire avec racine.

Un ABR a pour structure logique un arbre binaire.

COMPLEXITE

Les opérations basiques sur un arbre binaire de recherche dépensent:

un temps proportionnel à la hauteur de l'arbre.

Pour un arbre binaire complet à n nœuds, ces opérations s'exécutent en $\Theta(\log_2 n)$ dans le cas le plus défavorable.

En revanche, quand l'arbre se réduit à une chaîne linéaire de n nœuds, les mêmes opérations requièrent un temps $\Theta(n)$ dans le cas le plus défavorable.

La hauteur attendue d'un arbre binaire de recherche construit aléatoirement est $O(\log_2 n)$, ce qui fait que les opérations de base d'ensemble dynamique sur un tel arbre requièrent un temps $\Theta(\log_2 n)$ en moyenne.

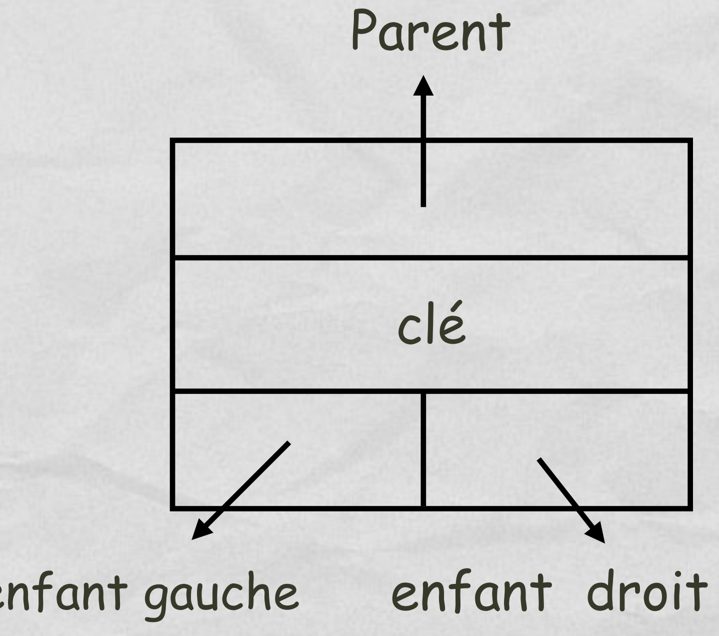
ABR

Comme son nom l'indique, un arbre binaire de recherche est organisé comme un arbre binaire.

On peut le représenter par une structure de données chaînée dans laquelle chaque nœud est un objet.

Chaque noeuds contient les champs suivants:

- Un champ **clé** et des données satellites,
- **gauche**: pointant sur son enfant de gauche
- **droite**: pointant sur son enfant de droite
- **p** qui pointe sur son parent.



PROPRIÉTÉ D'ABR

Les clés d'un ABR sont toujours stockées de manière à satisfaire à la ***propriété d'arbre binaire de recherche*** :

Soit *x* un nœud d'un arbre binaire de recherche.

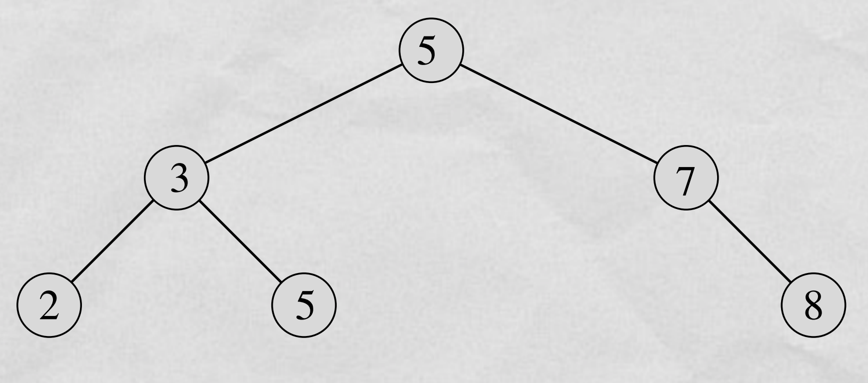
Si *y* est un nœud du sous-arbre de **gauche** de *x*, alors

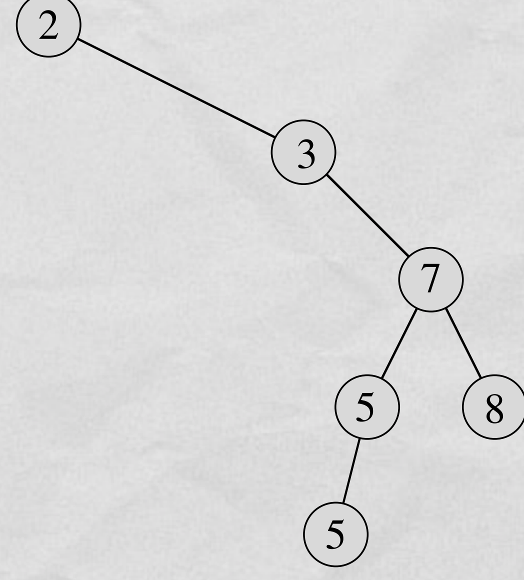
clé[y] ≤ **clé[x]**.

Si *y* est un nœud du sous-arbre de **droite** de *x*, alors

clé[x] ≤**clé[y]**.

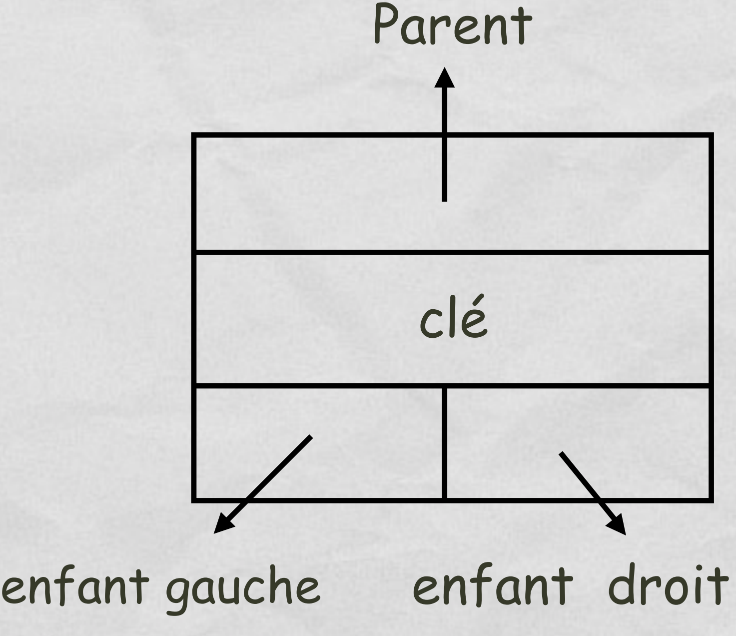
EXEMPLES



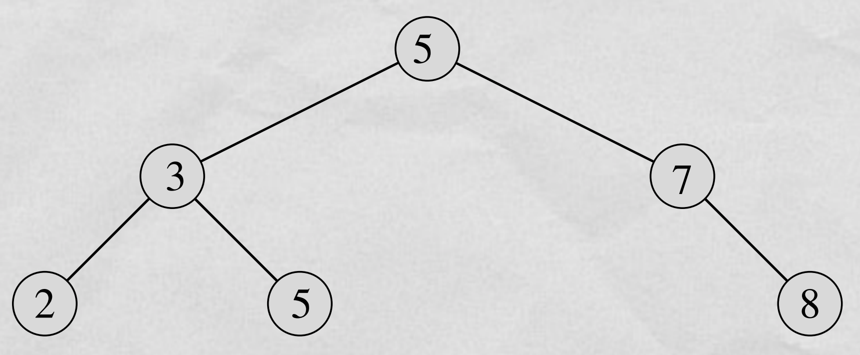


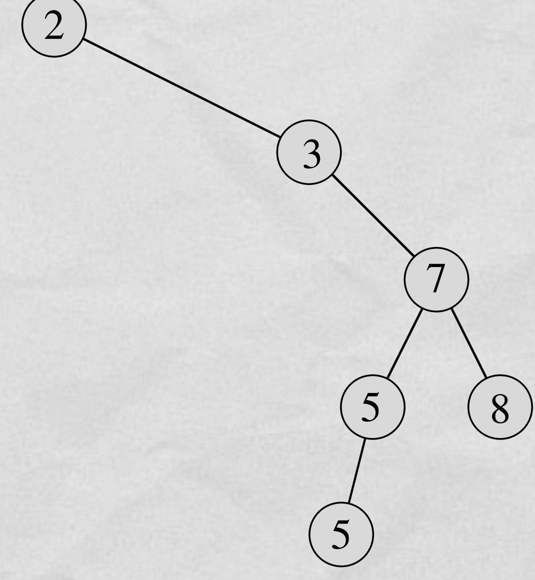
NOEUDS

```
Enregistrement Nœud {
     cle : Entier;
     gauche : ↑Nœud;
     droit : ↑Nœud;
     parent : ↑Nœud;
          }
```



COMMENT AFFICHER LES CLÉS DANS L'ORDRE CROISSANT?





AFFICHAGE DANS L'ORDRE

La propriété d'ABR permet d'afficher

toutes les clés de l'arbre dans l'ordre croissant

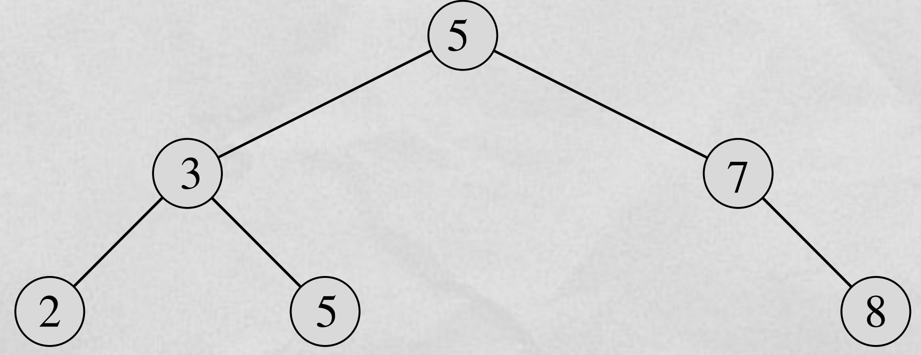
à l'aide d'un algorithme récursif simple de ***parcours infixe***

```
PARCOURS-INFIXE(x)
1   si x ≠ NIL
2       alors PARCOURS-INFIXE(gauche[x])
3             afficher clé[x]
4             PARCOURS-INFIXE(droite[x])
```

AFFICHAGE DANS L'ORDRE EXEMPLE

PARCOURS-INFIXE(x)

1 **si** $x \neq$ NIL
2 **alors** PARCOURS-INFIXE(*gauche*[x])
3 afficher *clé*[x]
4 PARCOURS-INFIXE(*droite*[x])

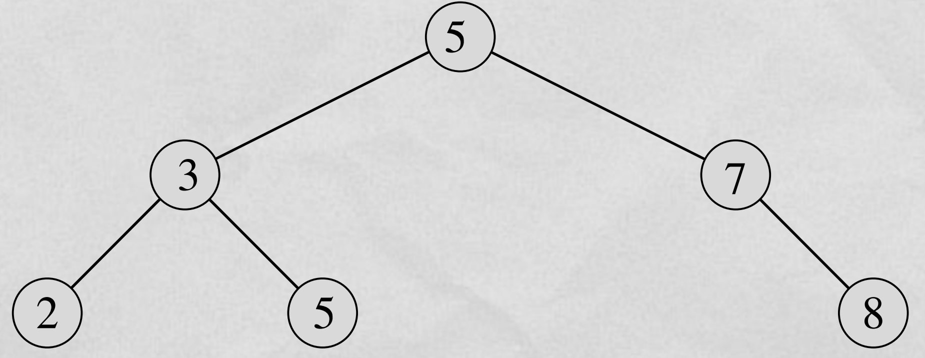


PARCOURS-INFIXE(*5*)
PARCOURS-INFIXE(*3*)
afficher *5*
PARCOURS-INFIXE(*7*)

AFFICHAGE DANS L'ORDRE EXEMPLE

PARCOURS-INFIXE(x)

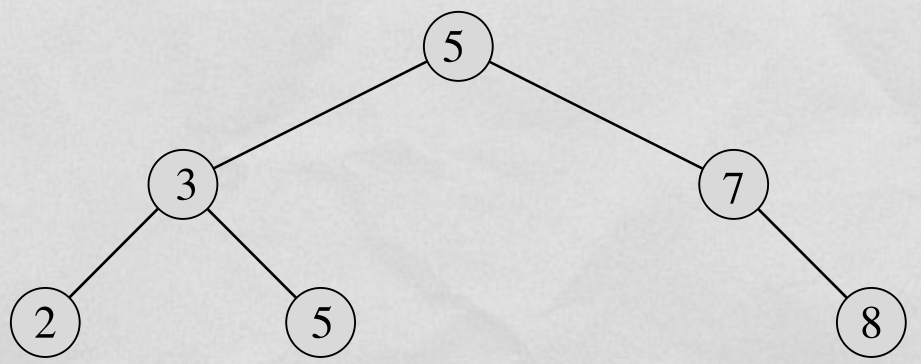
1 **si** $x \neq$ NIL
2 **alors** PARCOURS-INFIXE(*gauche*[x])
3 afficher *clé*[x]
4 PARCOURS-INFIXE(*droite*[x])



PARCOURS-INFIXE(*5*)
PARCOURS-INFIXE(*3*)
PARCOURS-INFIXE(*2*)
afficher *3*
PARCOURS-INFIXE(*5*)
afficher *5*
PARCOURS-INFIXE(*7*)
PARCOURS-INFIXE(*nil*)
afficher *7*
PARCOURS-INFIXE(*8*)

AFFICHAGE DANS L'ORDRE EXEMPLE

PARCOURS-INFIXE(x)
1 **si** $x \neq$ NIL
2 **alors** PARCOURS-INFIXE($gauche[x]$)
3 afficher $clé[x]$
4 PARCOURS-INFIXE($droite[x]$)

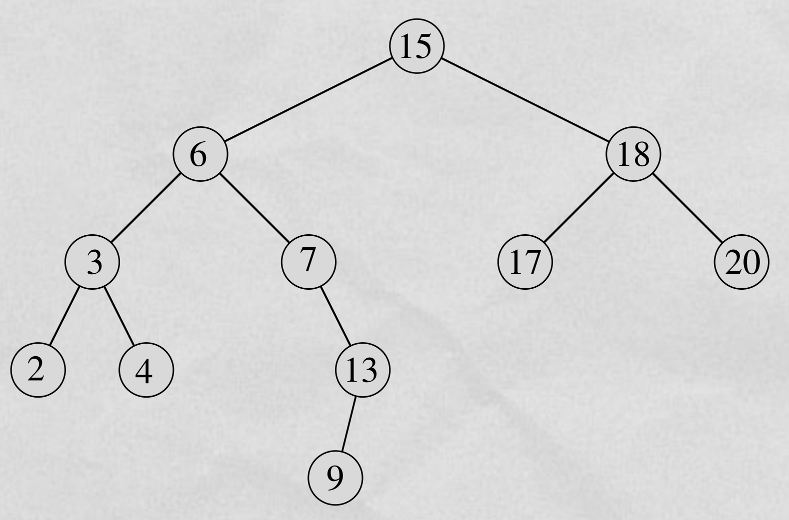


PARCOURS-INFIXE(*5*)
PARCOURS-INFIXE(*3*)
PARCOURS-INFIXE(*2*)
PARCOURS-INFIXE(*nil*)
afficher ***2***
PARCOURS-INFIXE(*nil*)
afficher ***3***
PARCOURS-INFIXE(*5*)
PARCOURS-INFIXE(*nil*)
afficher ***5***
PARCOURS-INFIXE(*nil*)
afficher *5*
PARCOURS-INFIXE(*7*)
PARCOURS-INFIXE(*nil*)
afficher *7*
PARCOURS-INFIXE(*8*)
PARCOURS-INFIXE(*nil*)
afficher ***8***
PARCOURS-INFIXE(*nil*)

REQUETES DANS UN ABR

RECHERCHE (RECURSIF)

ARBRE-RECHERCHER(x, k)
1 **si** x = NIL ou k = *clé*[x]
2 **alors retourner** x
3 **si** k < *clé*[x]
4 **alors retourner** ARBRE-RECHERCHER(*gauche*[x], k)
5 **sinon retourner** ARBRE-RECHERCHER(*droite*[x], k)

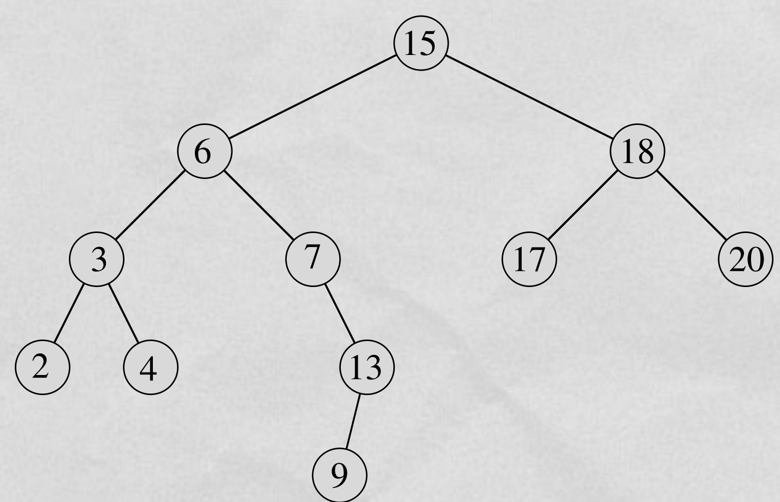


- ARBRE-RECHERCHER(r, *13*)
  - si *13* < *clé*[r]=15
    - alors retourner ARBRE-RECHERCHER(*gauche*[r], *13*)
- ARBRE-RECHERCHER(*6*, *13*)
  - 13>6:
    - ARBRE-RECHERCHER(*droite*[*6*], *13*)
- ARBRE-RECHERCHER(*7*, *13*)
  - 13>7:
    - ARBRE-RECHERCHER(*droite*[*7*], *13*)
- ARBRE-RECHERCHER(*13*, *13*)
  - retourner 13

RECHERCHE (ITERATIF)

ARBRE-RECHERCHER-ITÉRATIF(x, k)

1 **tant que** $x \neq$ NIL et $k \neq$ *clé*[x]
2 **faire si** $k <$ *clé*[x]
3 **alors** $x \leftarrow$ *gauche*[x]
4 **sinon** $x \leftarrow$ *droite*[x]
5 **retourner** x



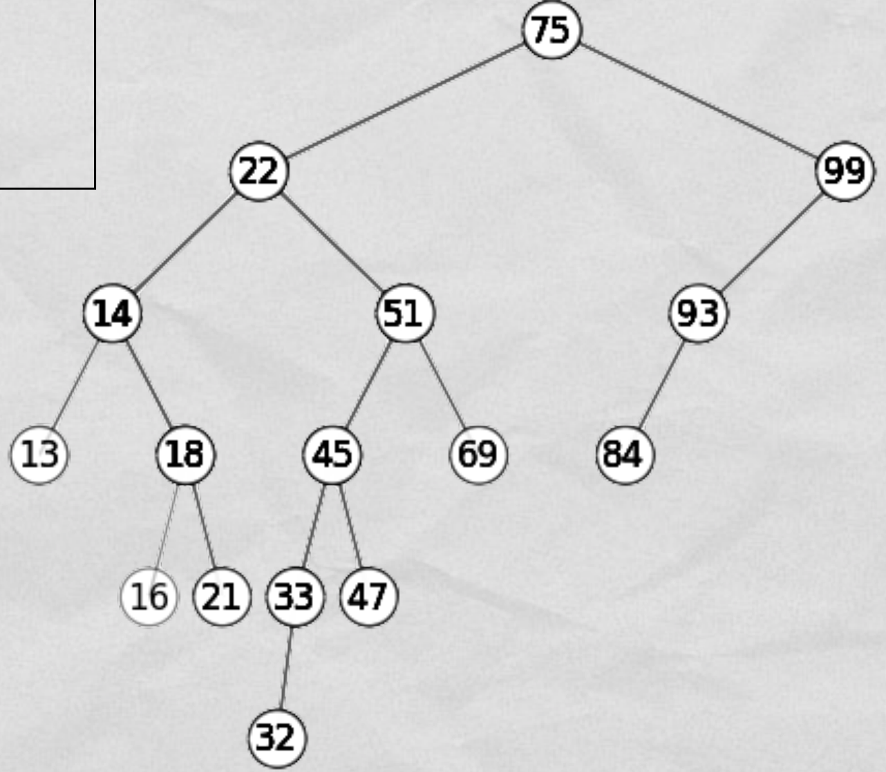
ARBRE-RECHERCHER-ITÉRATIF(r, *13*)

- *clé*[x]=15
  faire si *13* **<** *15*
  alors $x \leftarrow$ *gauche*[*15*]
- *clé*[x]=6
  faire si *13* **<** *6*
  sinon $x \leftarrow$ *droite*[*6*]
- *clé*[x]=7
  faire si *13* **<** *7*
  sinon $x \leftarrow$ *droite*[*7*]
- *clé*[x]=13
  retourner 13

RECHERCHE DE LA CLÉ MINIMUM

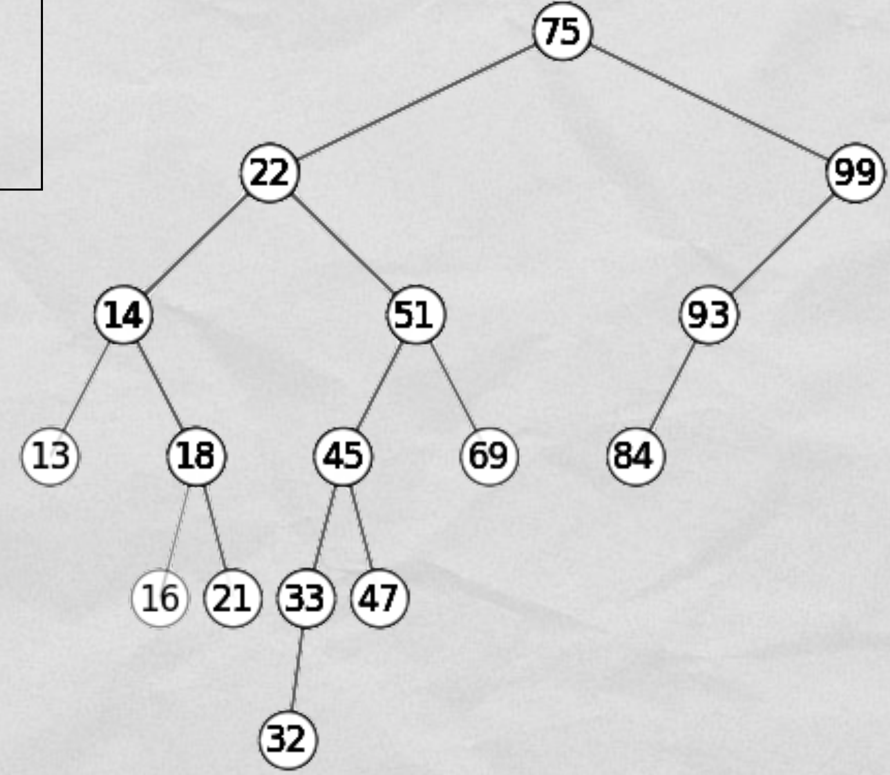
ARBRE-MINIMUM(x)

1 **tant que** $gauche[x] \neq$ NIL
2 **faire** $x \leftarrow gauche[x]$
3 **retourner** x



RECHERCHE DE LA CLÉ MAXIMUM

```
ARBRE-MAXIMUM(x)
1   tant que droite[x] ≠ NIL
2       faire x ← droite[x]
3   retourner x
```



SUCESSEURS ET PREDECESSEURS

Le **successeur** d'un nœud *x* est le nœud possédant

la plus petite clé **supérieure** à *clé*[*x*].

Le **predecesseur** d'un nœud *x* est le nœud possédant

la plus grande clé **inférieure** à *clé*[*x*].

Le successeur de 51:

69

Le predecesseur de 51:

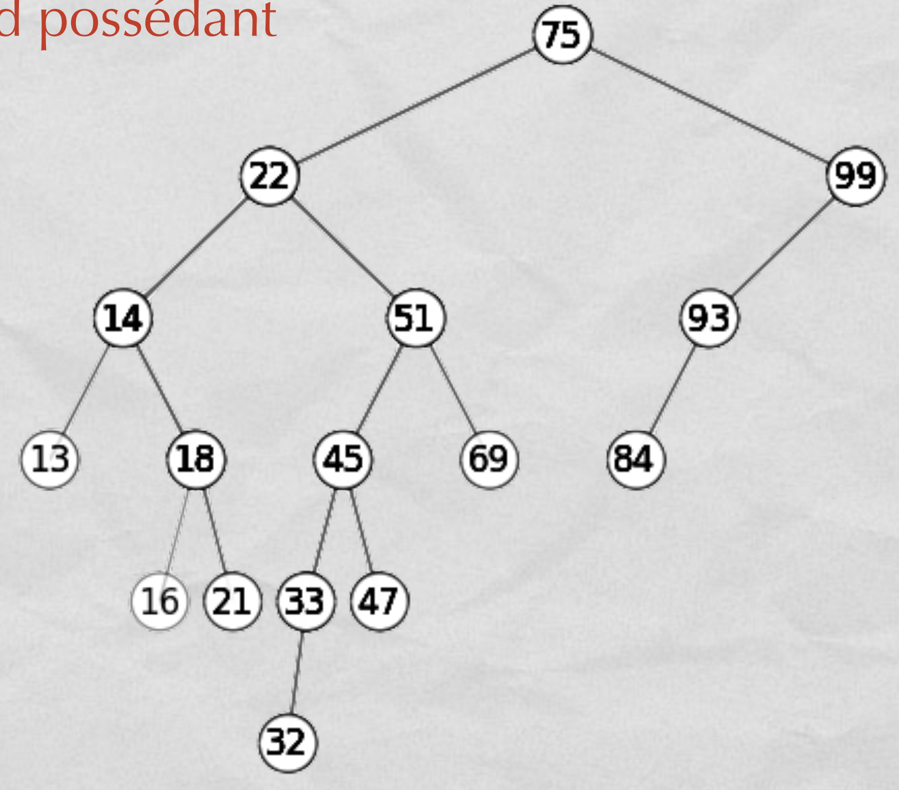
47

Le successeur de 84:

93

Le predecesseur de 84:

75



SUCESSEURS

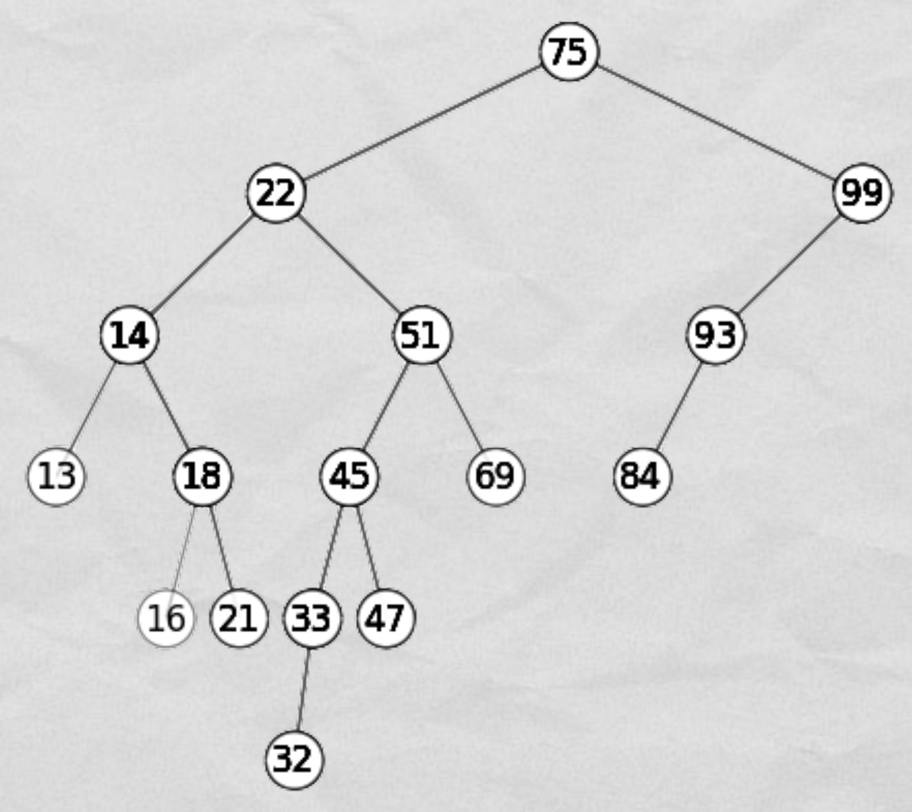
Soit x un nœud

on cherche y tel que

- clé(y)>clé(x)

et tel que pour tout nœud z

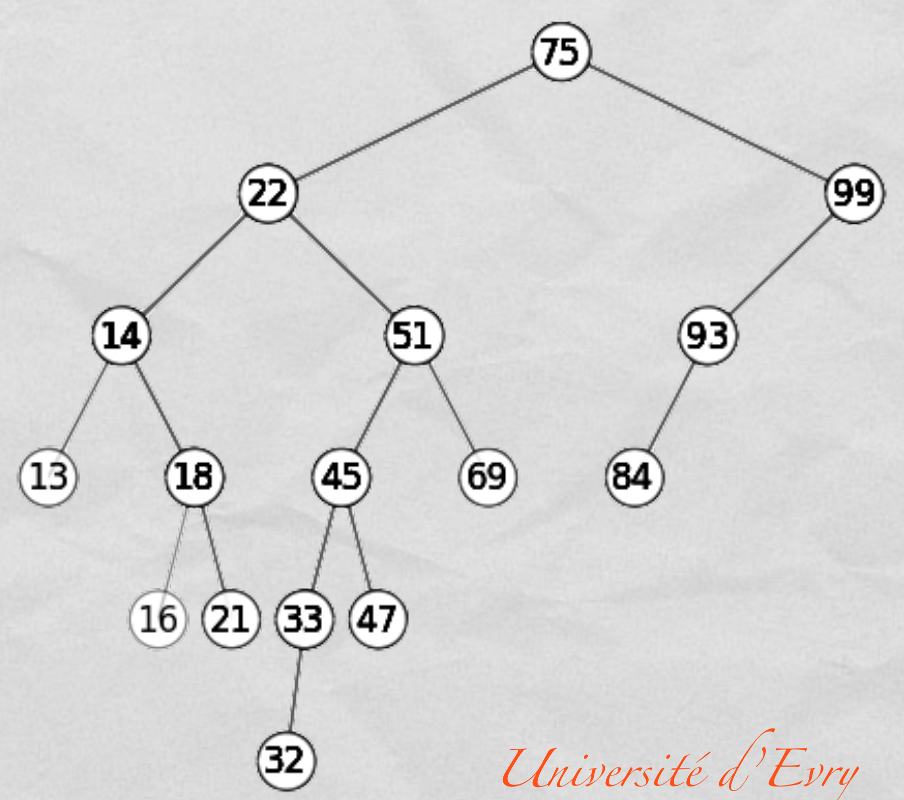
- z≠x et z≠y
- on n'ait pas clé(y)>clé(z)>clé(x)



SUCESSEURS

Le successeur d'un nœud x est le nœud possédant la plus petite clé dans le sous-arbre droit si x en possède un

sinon c'est le nœud qui est le 1er ancêtre de x dont le fils gauche est aussi un ancêtre de x (ou x lui-même)

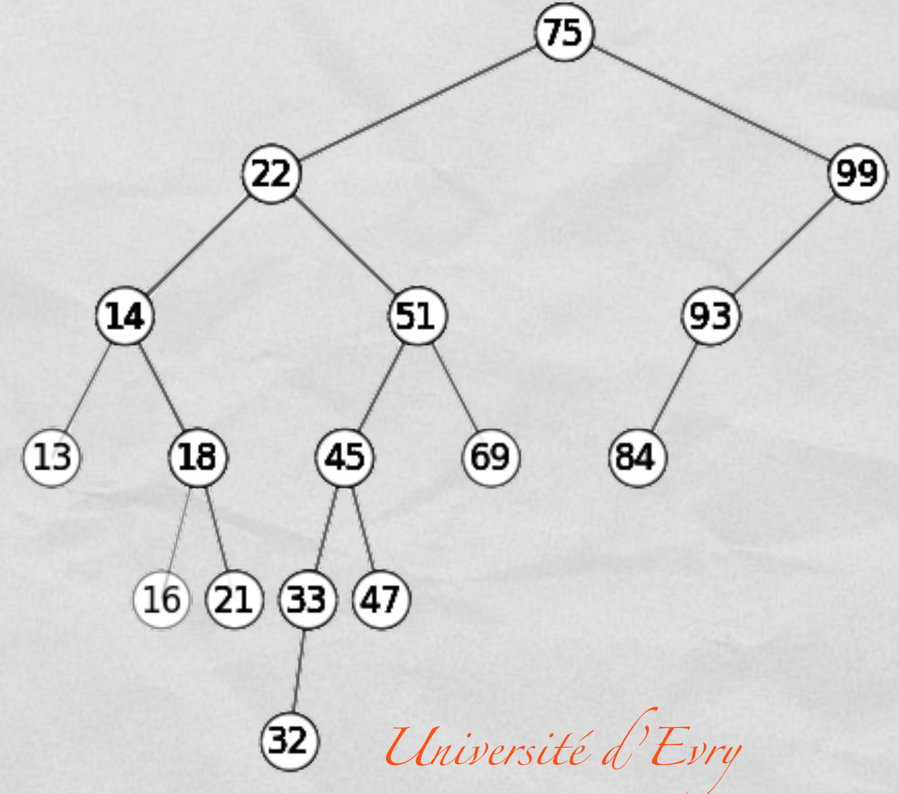


SUCESSEURS

Le successeur d'un nœud x est le nœud possédant la plus petite clé dans le sous-arbre droit si x en possède un

sinon c'est le nœud qui est le 1er ancêtre de x dont le fils gauche est aussi un ancêtre de x (ou x lui-même)

```
ARBRE-SUCCESSEUR(x)
1  si droite[x] ≠ NIL
2      alors retourner ARBRE-MINIMUM(droite[x])
3  y ← p[x]
4  tant que y ≠ NIL et x = droite[y]
5       faire x ← y
6             y ← p[y]
7  retourner y
```



Lélia Blin

Université d'Evry

SUCESSEURS

ARBRE-SUCCESSEUR(x)

1 **si** $droite[x] \neq$ NIL
2 **alors retourner** ARBRE-MINIMUM($droite[x]$)
3 $y \leftarrow p[x]$
4 **tant que** $y \neq$ NIL et $x = droite[y]$
5 **faire** $x \leftarrow y$
6 $y \leftarrow p[y]$
7 **retourner** y

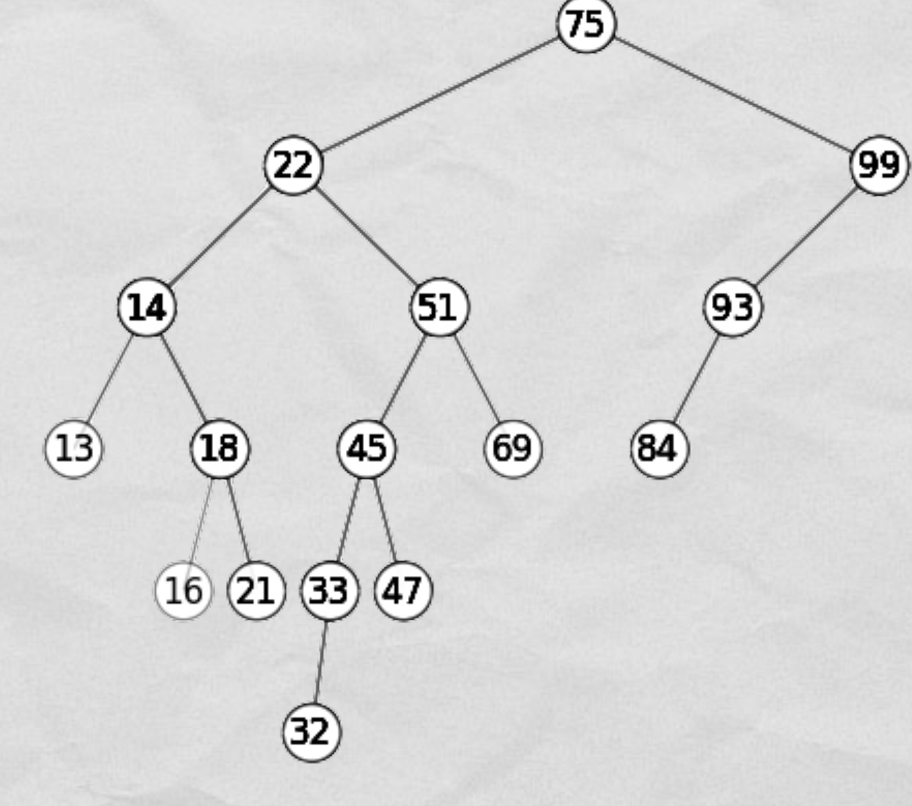
ARBRE-SUCCESSEUR(*69*)

69 n'a pas de sous arbre droit

$y \leftarrow p[x]=51$

- **tantque** $y \neq$NIL et $x=droite[y]$
  - $x \leftarrow 51$
  - **y ← 22**
- **tantque** $y \neq$NIL et $x=droite[y]$
  - $x \leftarrow 22$
  - **y ← 75**
- **tantque** $y \neq$NIL et $x=droite[y]$
  - **non**

retourner *75*



INSERTION ET SUPPRESSION DANS UN ABR

INSERTION ET SUPPRESSION

Les opérations d'insertion et de suppression

modifient l'ensemble dynamique représenté par un ABR.

La structure de données doit être modifiée pour refléter ce changement

tout en conservant la propriété d'ABR.

Comme nous le verrons,

modifier l'arbre pour y insérer un nouvel élément est relativement simple,

mais la suppression est un peu plus délicate à gérer.

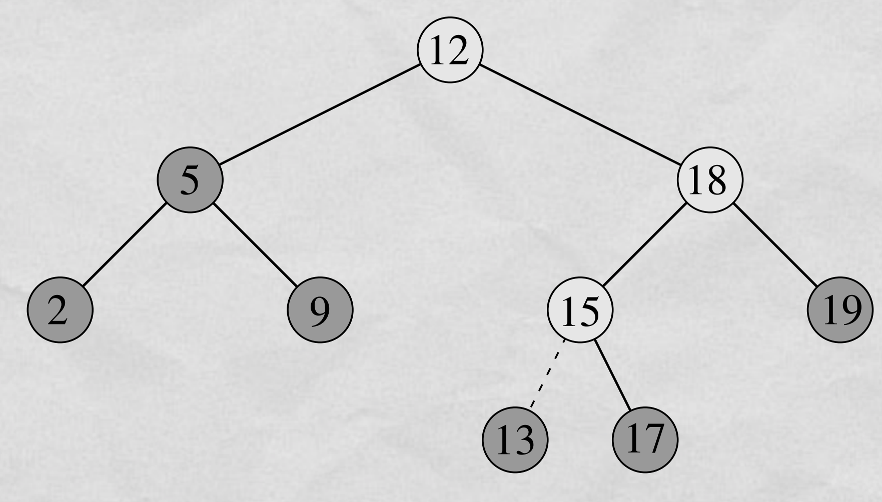
INSERTION

2 techniques :

- Insertion à la racine l'ABR
  - Modification de la structure de l'ABR
- Insertion aux feuilles de l'ABR
  - Modification de la hauteur de l'ABR

INSERTION AUX FEUILLES

```
ARBRE-INSÉRER(T, z)
 1   y ← NIL
 2   x ← racine[T]
 3   tant que x ≠ NIL
 4        faire y ← x
 5                 si clé[z] < clé[x]
 6                    alors x ← gauche[x]
 7                    sinon x ← droite[x]
 8   p[z] ← y
 9   si y = NIL
10       alors racine[T] ← z
11       sinon si clé[z] < clé[y]
12                    alors gauche[y] ← z
13                    sinon droite[y] ← z
```



INSERTIONS SUCESSIVES

Faites une insertion sucessives des éléments de liste suivante:

75-22-14-51-99-18-45-33-93-32-47-84-21-69-13-16

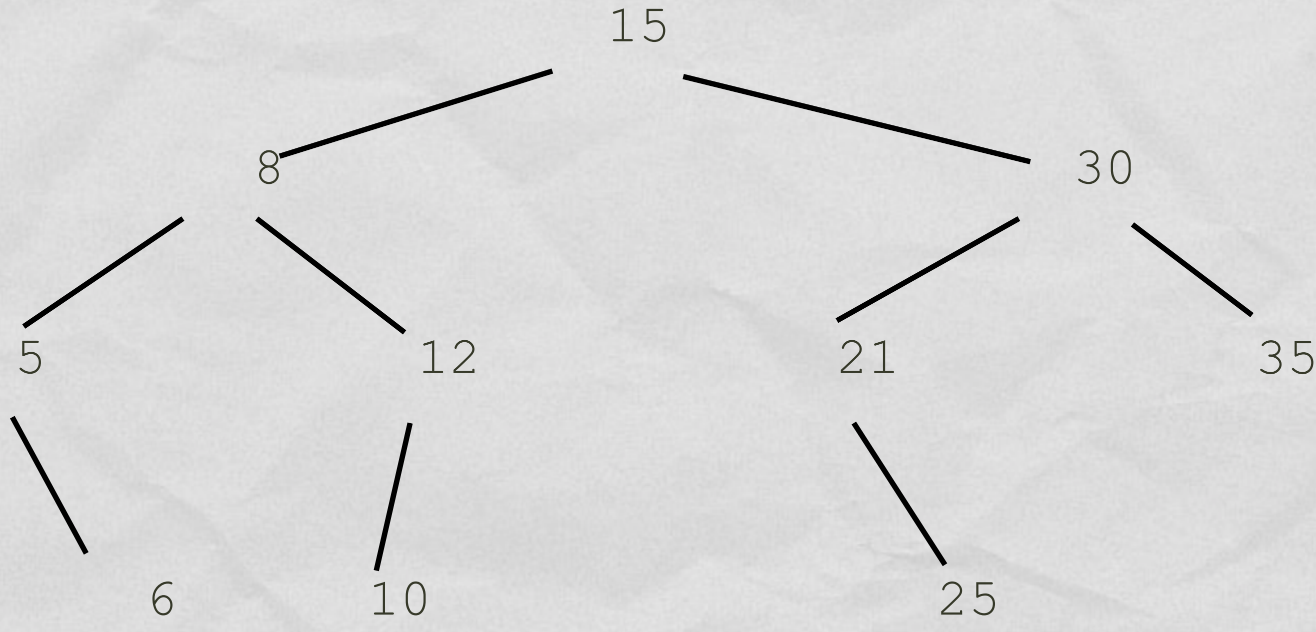
INSERTIONS SUCESSIVES AUX FEUILLES

Insertion in binary search tree

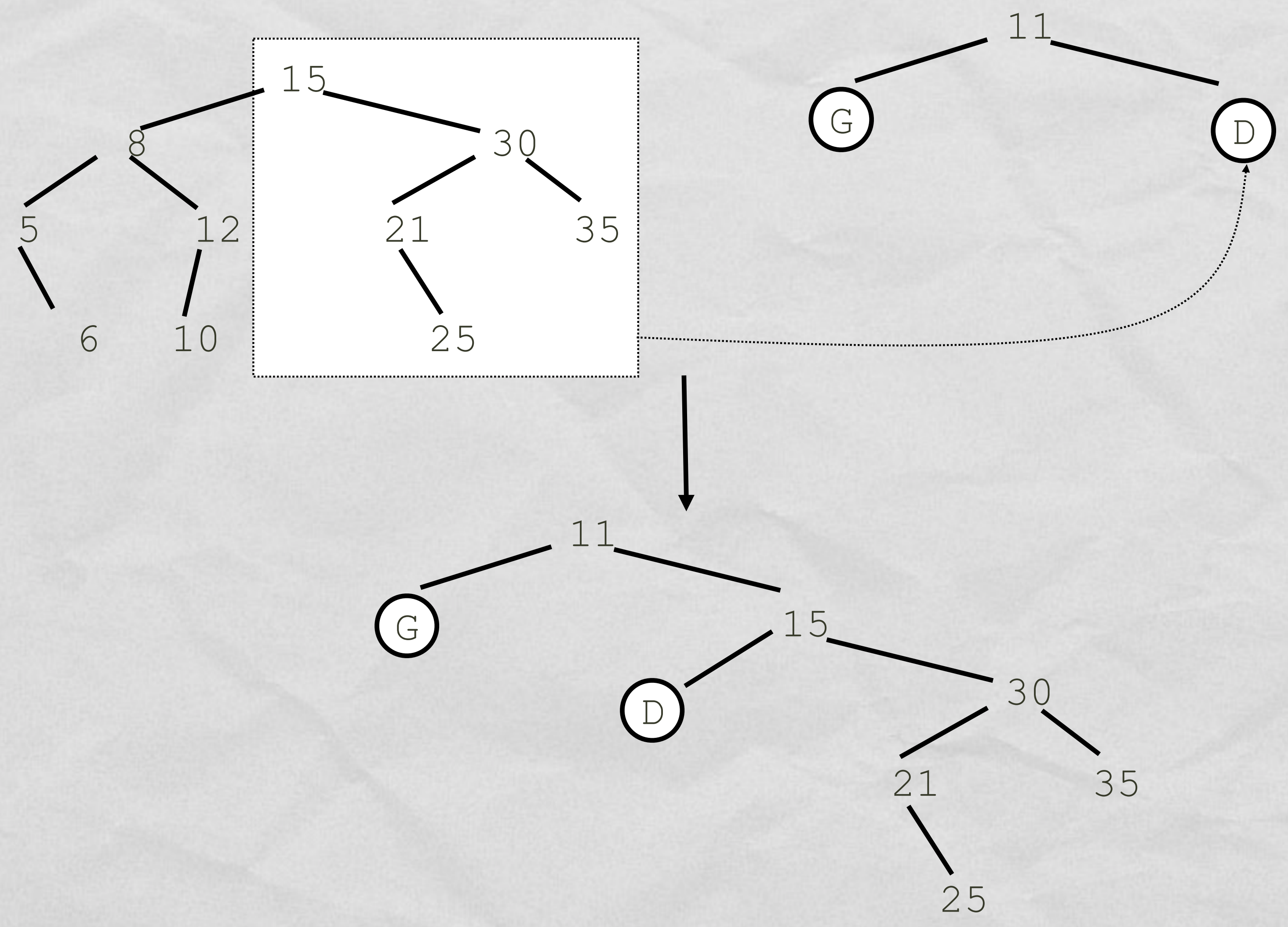
(75)-22- 14- 51- 99- 18- 45- 33- 93- 32- 47- 84- 21- 69- 13- 16

INSERTIONS À LA RACINE

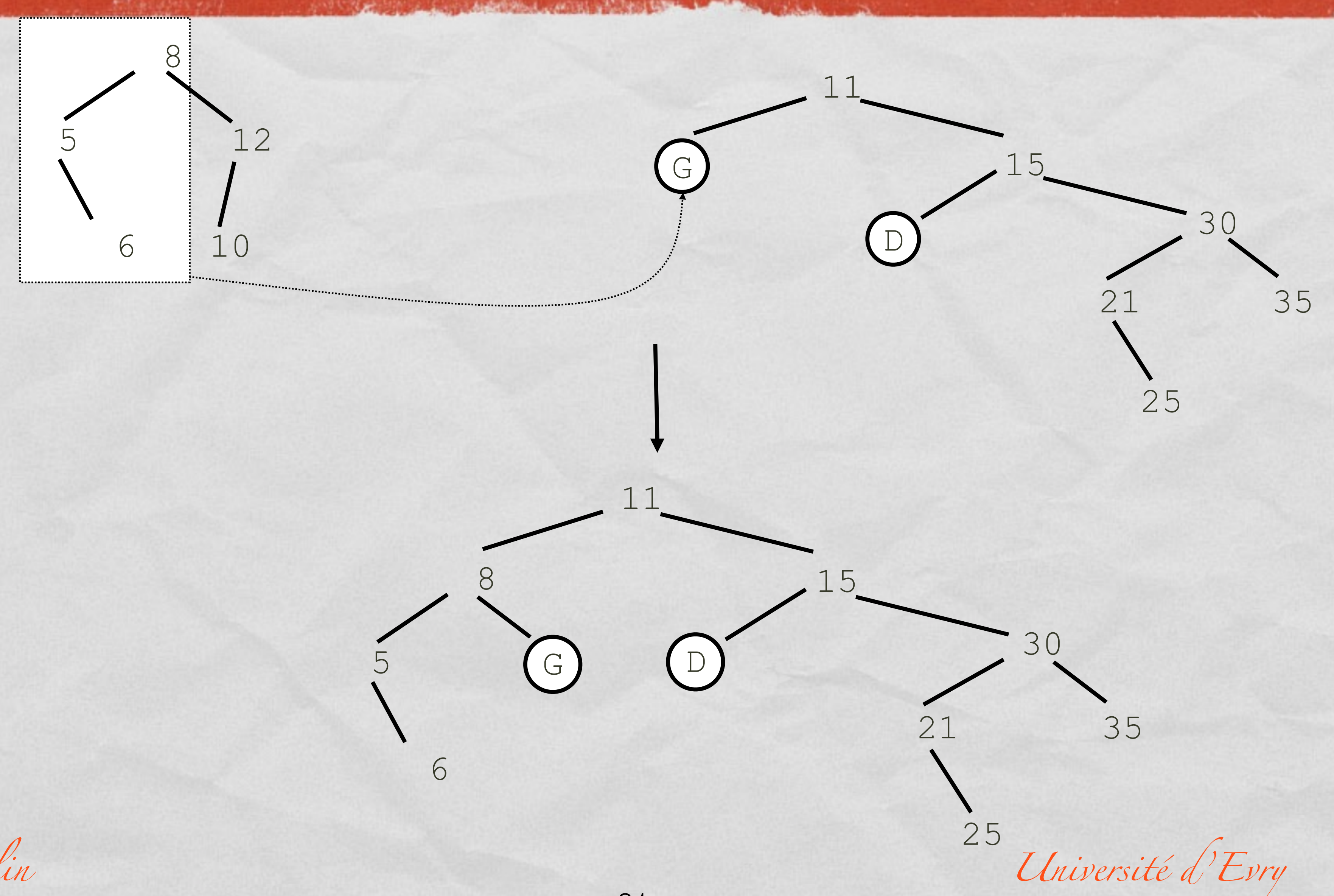
- Ajout de 11 à l'arbre suivant :



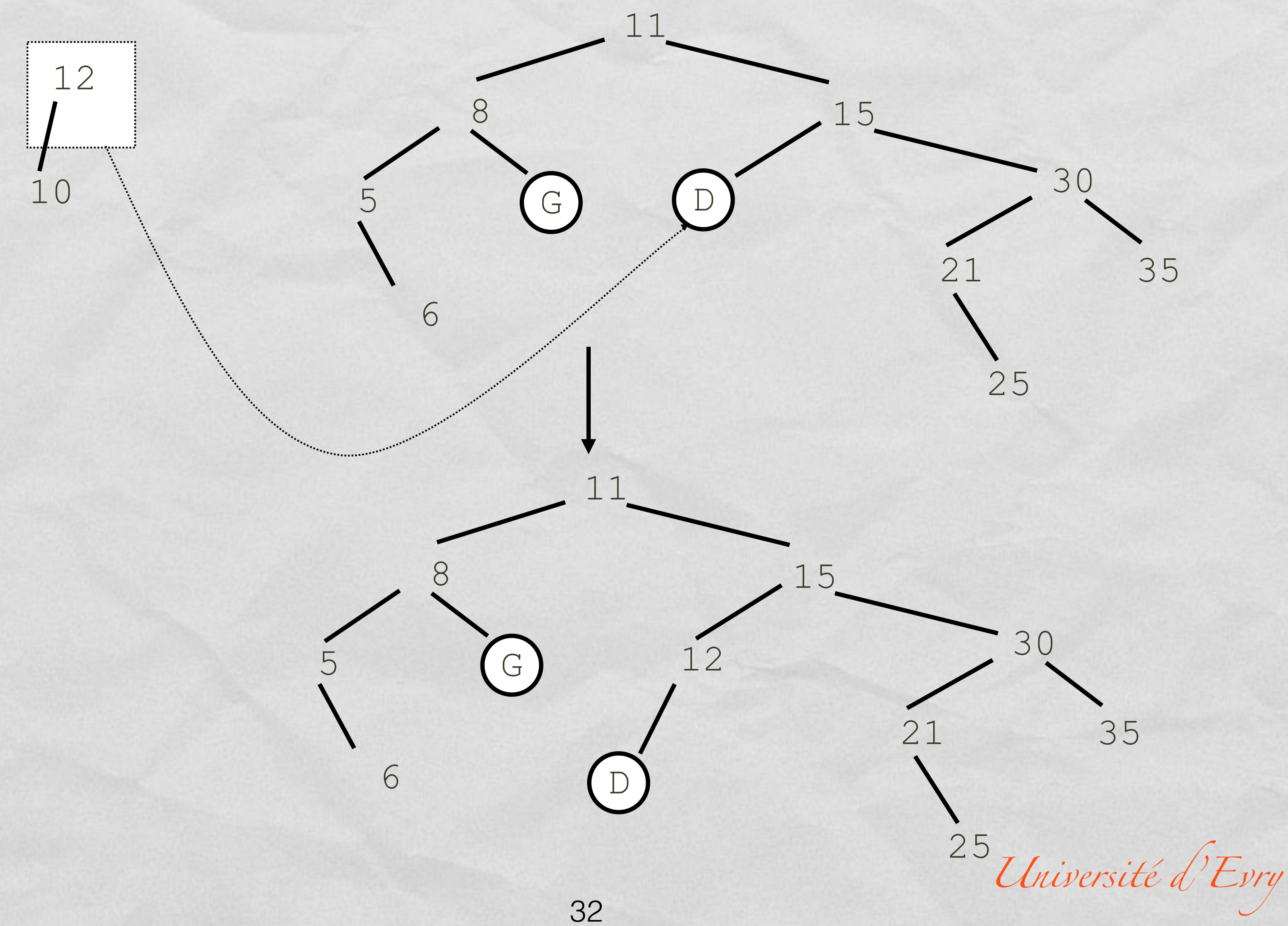
INSERTIONS À LA RACINE



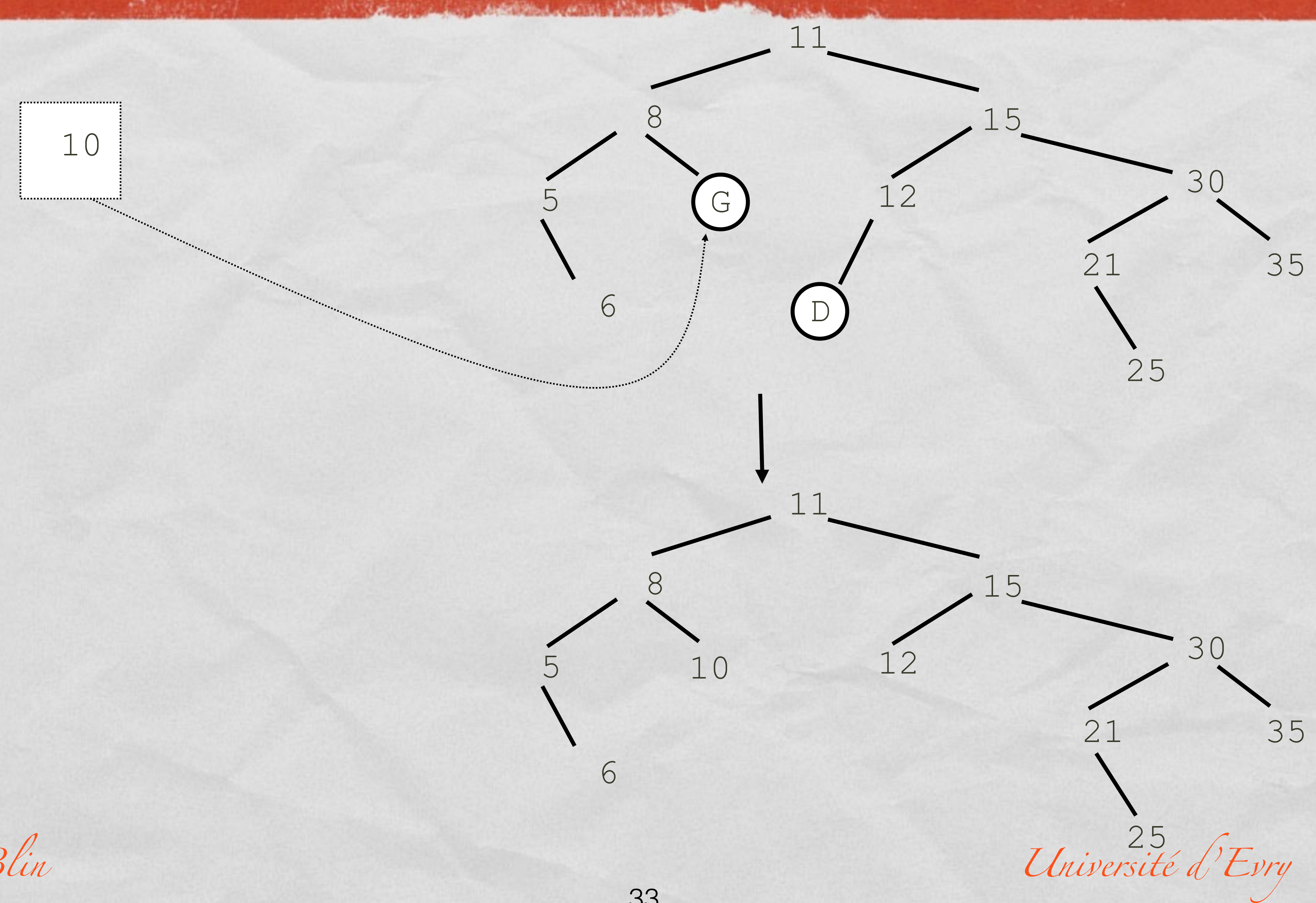
INSERTIONS À LA RACINE



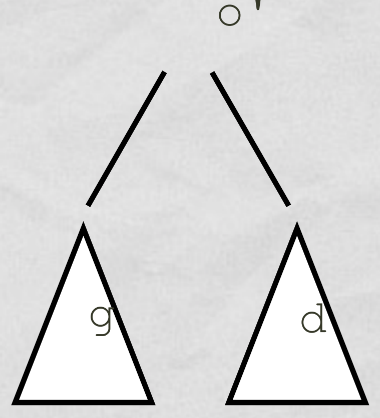
INSERTIONS À LA RACINE



INSERTIONS À LA RACINE



INSERTIONS À LA RACINE GÉNÉRALISATION



Soit un arbre a = <o′, g, d>

Ajouter le nœud o à a, c′est construire l′arbre <o, a1, a2> tel que :

- a1 contienne tous les nœuds dont la clé est inférieure à celle de o
- a2 contienne tous les nœuds dont la clé est supérieure à celle de o

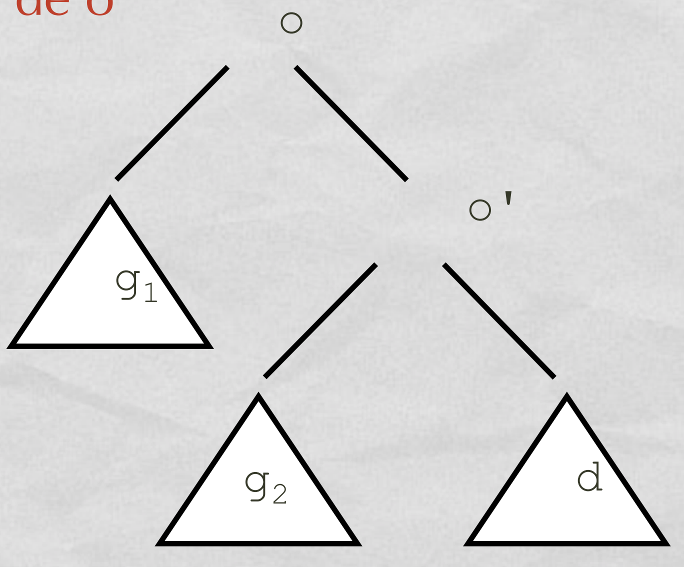
INSERTIONS À LA RACINE GÉNÉRALISATION

si la_clé(o) < la_clé(o′)

a1 = g1 et a2 = <o′, g2, d>

g1 = nœuds de g dont la clé est inférieure à la clé de o

g2 = nœuds de g dont la clé est supérieure à la clé de o



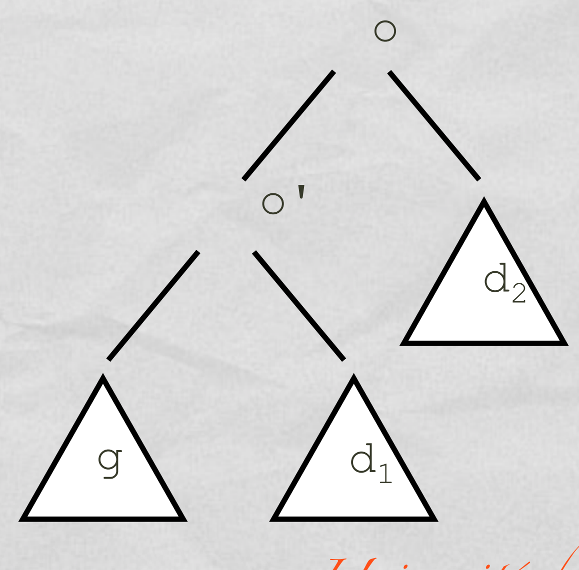
INSERTIONS À LA RACINE GÉNÉRALISATION

si la_clé(o) > la_clé(o')

a1 = <o', g, d1> et a2 = d2

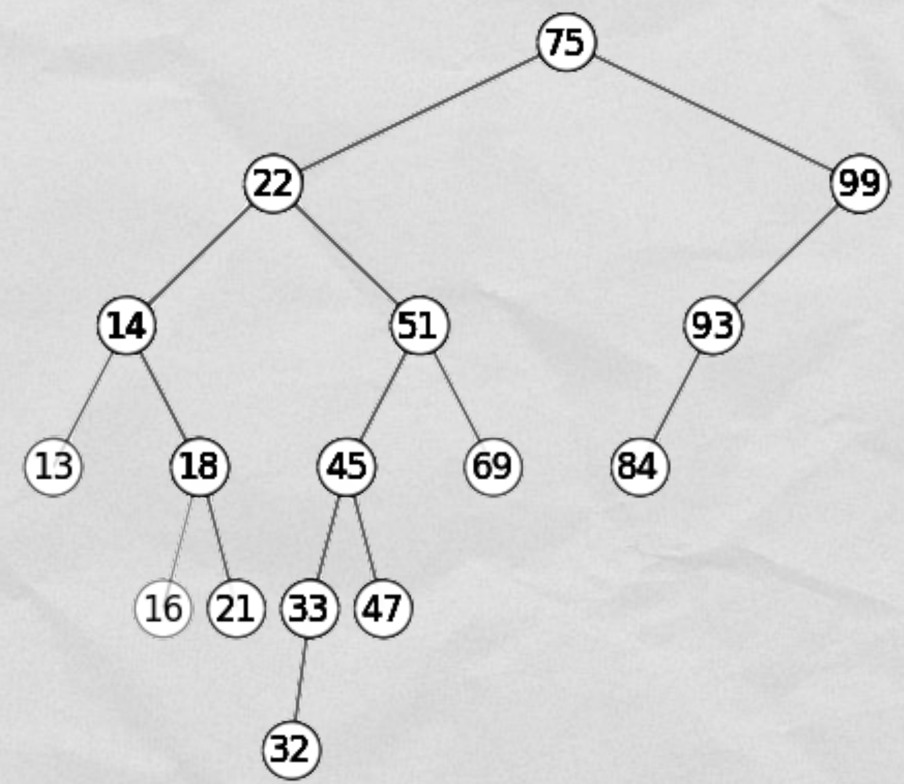
d1 = nœuds de d dont la clé est inférieure à la clé de o

d2 = nœuds de d dont la clé est supérieure à la clé de o



INSERTIONS A LA RACINE

Faites l'insertion du noeud de valeur 35 à la racine de cet arbre:



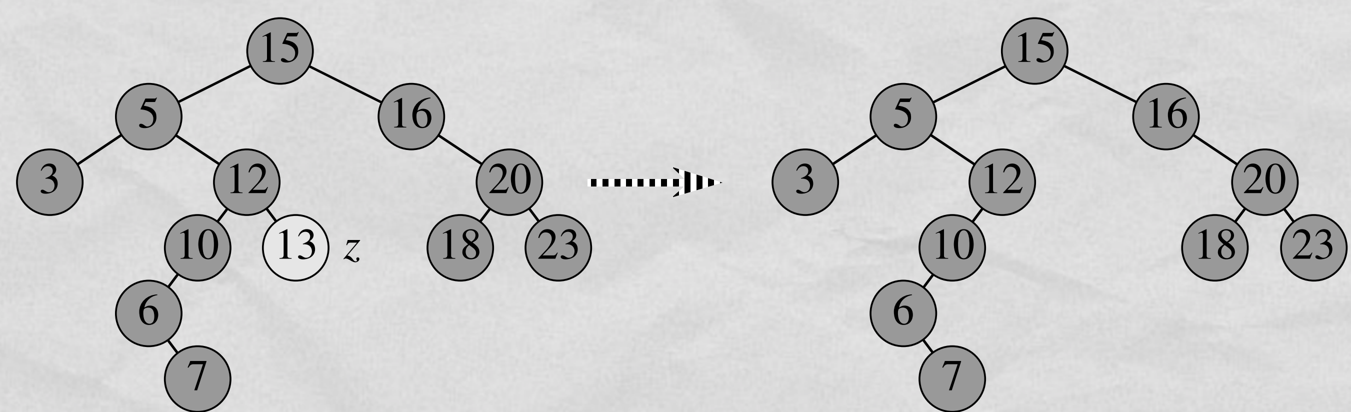
SUPPRESSION

Premièrement on recherche le nœud à supprimer.

Puis 3 cas sont a examiner:

si le noeud est une feuille:

suppression simple

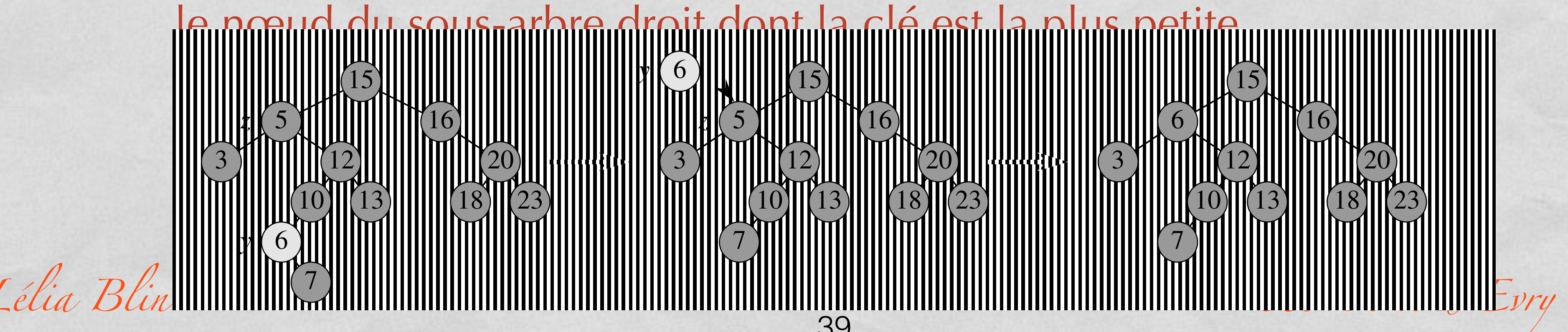


SUPPRESSION

Premièrement on recherche le nœud à supprimer.

Puis 3 cas sont a examiner:

- si le noeud est une feuille:
    - suppression simple
- si nœud interne avec 2 enfants on remplace
    - le nœud du sous-arbre gauche dont la clé est la plus grande
    - le nœud du sous-arbre droit dont la clé est la plus petite

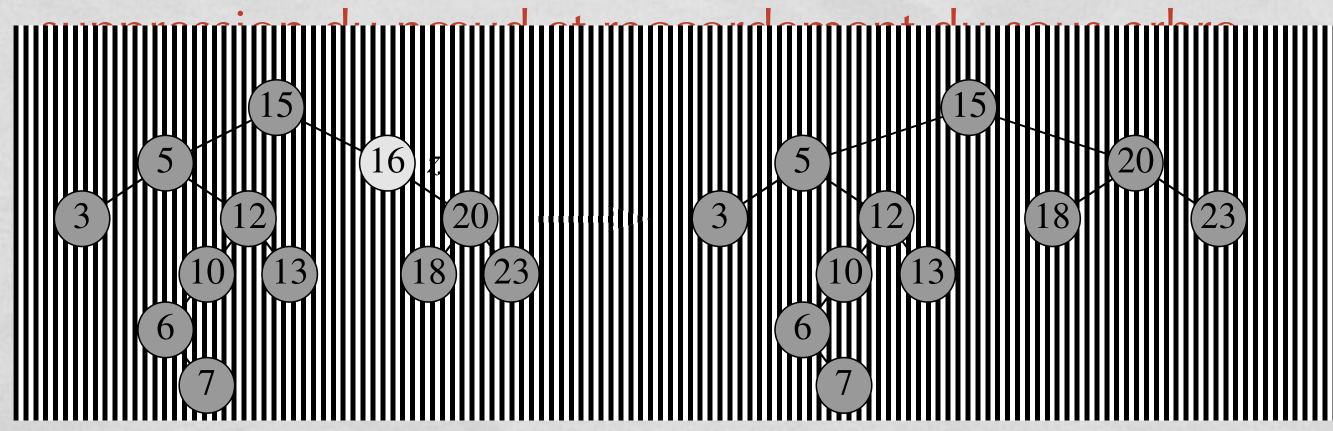


SUPPRESSION

Premièrement on recherche le nœud à supprimer.

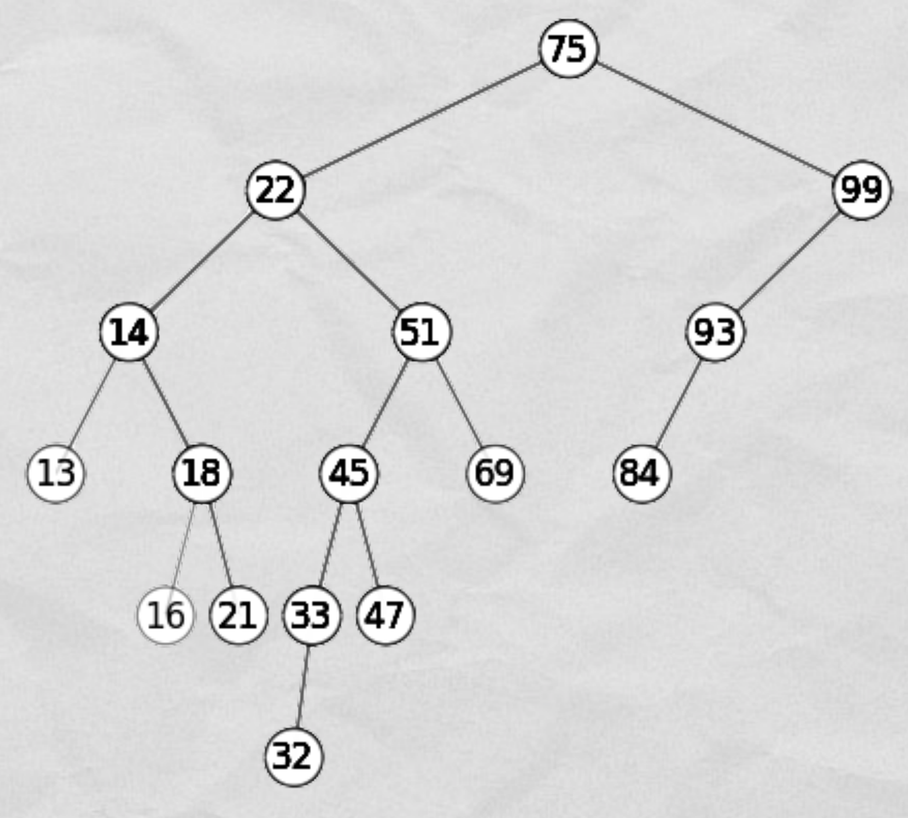
Puis 3 cas sont a examiner:

- si le noeud est une feuille:
- si nœud interne avec 2 enfants on remplace
- si le noeud est un nœud interne au sens large:
  - suppression du nœud et raccordement du sous-arbre



SUPPRESSION

```
ARBRE-SUPPRIMER(T, z)
 1  si gauche[z] = NIL ou droite[z] = NIL
 2      alors  y ← z
 3      sinon  y ← ARBRE-SUCCESSEUR(z)
 4  si gauche[y] ≠ NIL
 5      alors  x ← gauche[y]
 6      sinon  x ← droite[y]
 7  si x ≠ NIL
 8      alors  p[x] ← p[y]
 9  si p[y] = NIL
10      alors  racine[T] ← x
11      sinon si y = gauche[p[y]]
12                alors  gauche[p[y]] ← x
13                sinon  droite[p[y]] ← x
14  si y ≠ z
15      alors  clé[z] ← clé[y]
16             copier données satellites de y dans z
17  retourner y
```



COMPLEXITE

- Les différentes opérations ont une complexité au pire de h(a)
  - $\lfloor \log_2 n \rfloor \leq h(a) \leq n-1$
  - Pour les arbres complets, complexité en $O(\log_2 n)$
  - Pour les arbres dégénérés, complexité en $O(n)$
- La complexité dépend de la forme de l'arbre, qui dépend des opérations d'ajout et de suppression
  - ajout d'éléments par clés croissantes → arbre dégénéré
  - en moyenne, la profondeur est de $2 \log_2 n$
  - but = équilibrer les arbres en hauteur

ARBRES H-ÉQUILIBRÉS

Arbres H-équilibrés

Définitions

déséquilibre(a) = h(g(a)) – h(d(a))

un arbre a est **H-équilibré** si pour tous ses sous-arbres b, on a :

déséquilibre(b) ∈ {-1, 0, 1}

un **arbre AVL** est un arbre de recherche qui est H-équilibré

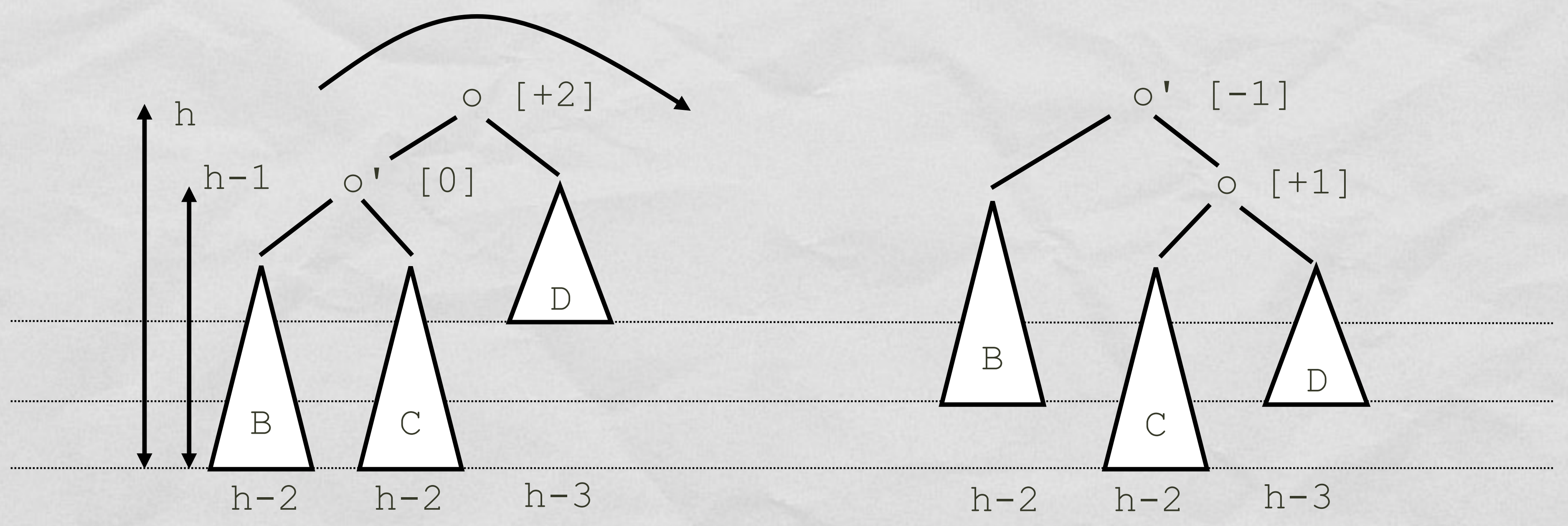
les propriétés et les opérations définies sur les arbres de recherche peuvent s'appliquer aux arbres AVL

Son nom est dû au initiales des auteurs: Adelson-Velskii et E. M. Landis (1962)

OPERATION DE ROTATION

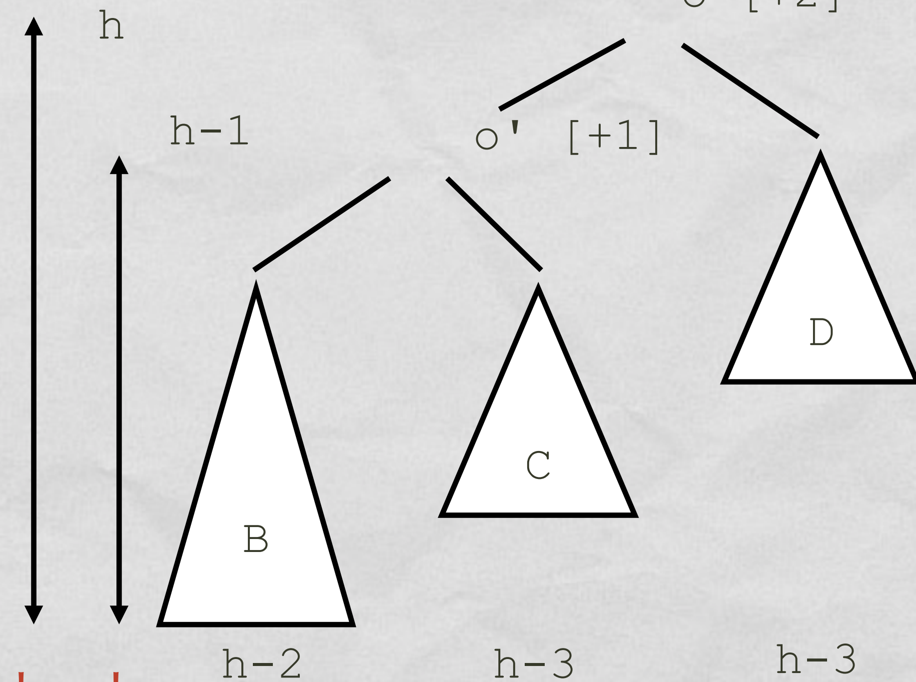
- Le problème est d'essayer de rééquilibrer un arbre déséquilibré afin de le ramener à un arbre H-équilibré.
- Cas d'un déséquilibre +2
  - on suppose que les sous-arbres droit et gauche sont H-équilibrés
  - hauteur du sous-arbre gauche supérieure de 2 à la hauteur du sous-arbre droit
  - opération à pratiquer dépend du déséquilibre du sous-arbre gauche qui peut être +1, 0, -1

DÉSÉQUILIBRE 0 SUR LE FILS GAUCHE



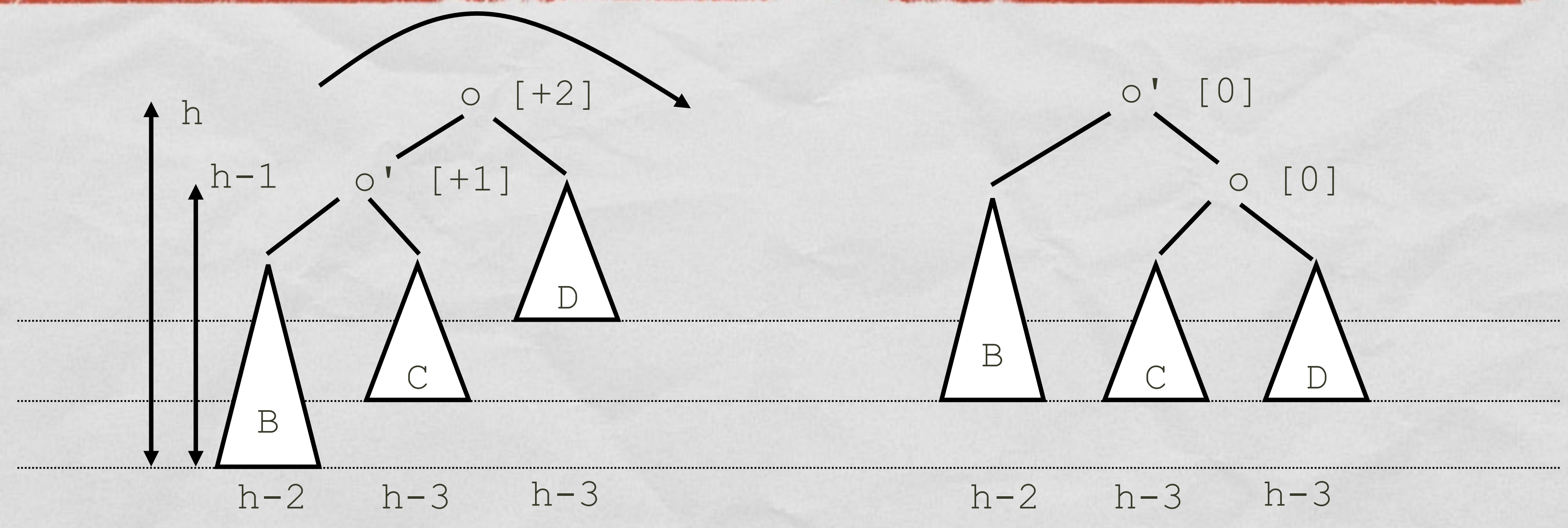
- Si l'arbre A est un ABR, le résultat est un ABR
- le déséquilibre de l'arbre résultant est de -1
- arbre H-équilibré dont la hauteur est identique

DÉSÉQUILIBRE +1 SUR LE FILS GAUCHE



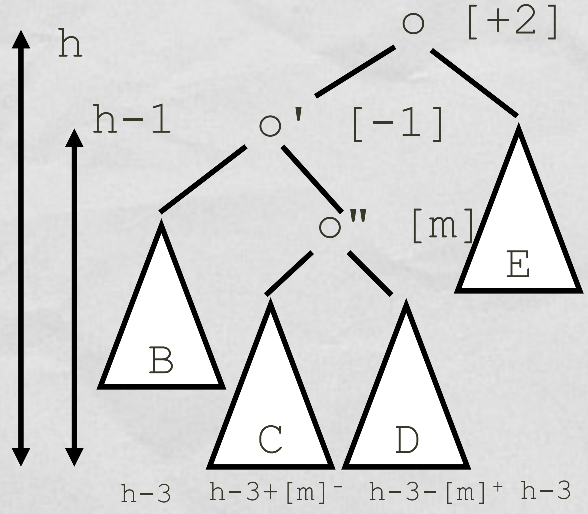
- rotation à droite :
- o′ devient la racine de l′arbre
- Le fils droit de o′ devient le fils gauche de o
- o devient le fils droit de o′

DÉSÉQUILIBRE +1 SUR LE FILS GAUCHE



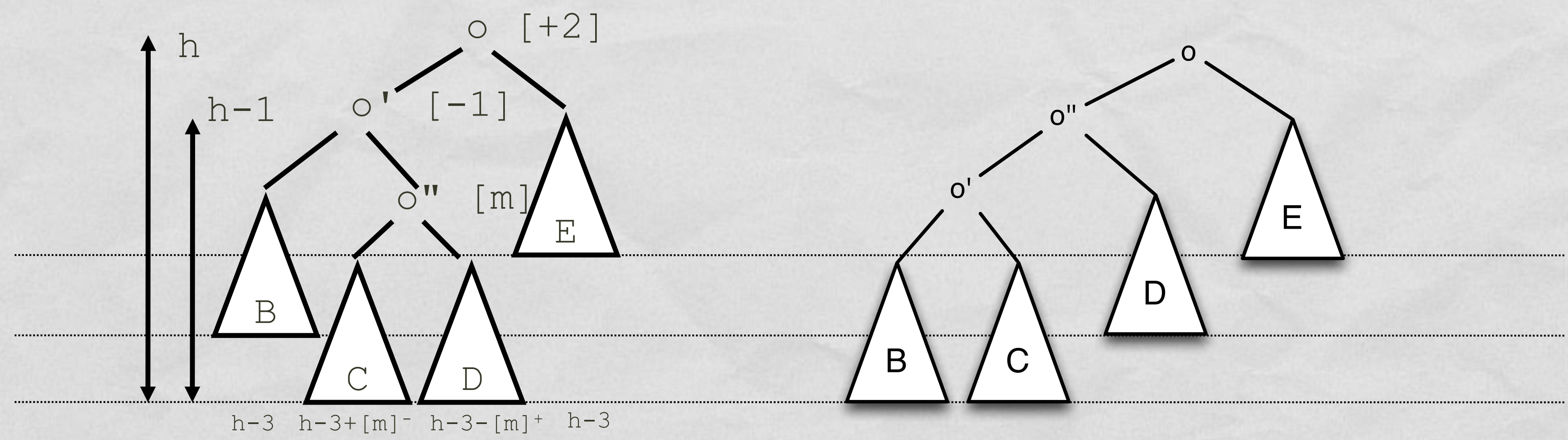
- Si l'arbre A est un ABR, le résultat est un ABR
- le déséquilibre de l'arbre résultant est nul
- arbre H-équilibré dont la hauteur a été diminué de 1

DÉSÉQUILIBRE -1 SUR LE FILS GAUCHE



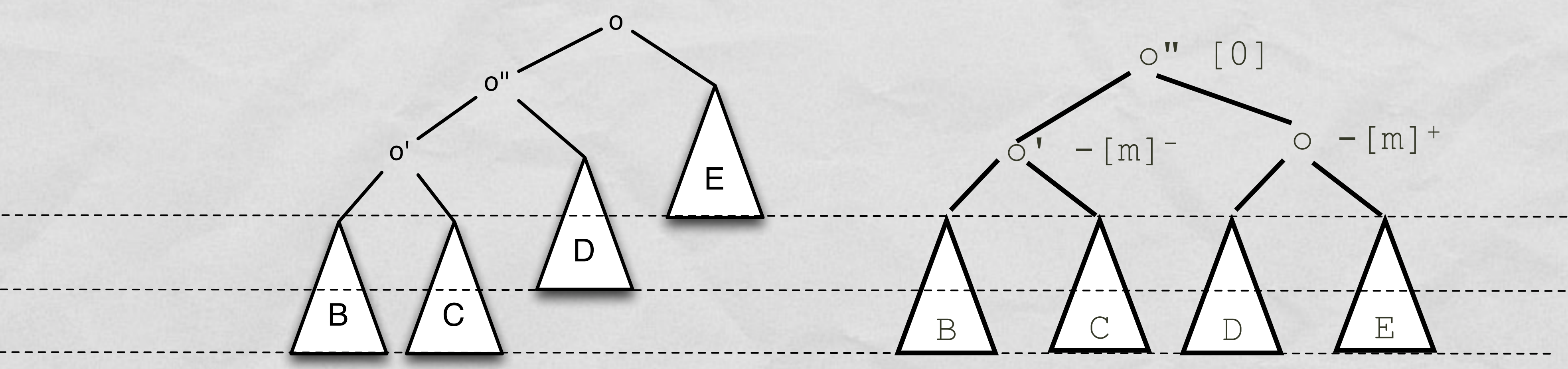
- Rotation gauche-droite :
- Rotation à gauche sur le fils gauche
- Rotation à droite sur la racine

DÉSÉQUILIBRE -1 SUR LE FILS GAUCHE



- Rotation à gauche sur le fils gauche

DÉSÉQUILIBRE -1 SUR LE FILS GAUCHE



- Rotation à droite sur la racine
- Si l'arbre A est un arbre binaire de recherche, le résultat est un arbre binaire de recherche
  - le déséquilibre de l'arbre résultant est de 0
  - L'arbre H-équilibré dont la hauteur est diminuée d'un

OPERATION DE RE-EQUILIBRAGE

arbre origine	opération	résultat	hauteur
+2, +1	rotation droite	0, 0	diminution
+2, 0	rotation droite	−1, +1	identique
+2, −1, −1	rotation gauche droite	0, +1, 0	diminution
+2, −1, 0	rotation gauche droite	0, 0, 0	diminution
+2, −1, +1	rotation gauche droite	0, 0, −1	diminution
−2, −1	rotation gauche	0, 0	diminution
−2, 0	rotation gauche	+1, −1	identique
−2, +1, −1	rotation droite gauche	0, +1, 0	diminution
−2, +1, 0	rotation droite gauche	0, 0, 0	diminution
−2, +1, +1	rotation droite gauche	0, 0, −1	diminution

OPERATION D'AJOUT

- Principe :
  - ajout du nœud par l'opération ajouter-f
  - rééquilibrage de l'arbre en partant de la feuille et en remontant vers la racine
- Complexité :
  - complexité au pire en $O(\log_2 n)$ en nb de comparaisons
  - au plus une rotation
  - expérimentalement : en moyenne une rotation pour 2 ajouts

OPERATION DE SUPPRESSION

- Principe :
  - suppression du nœud par l'opération sur les arbres de recherche
  - rééquilibrage de l'arbre en partant du nœud supprimé et en remontant vers la racine
- Complexité :
  - complexité au pire en $O(\log_2 n)$ en nb de comparaisons et en nb de rotations
  - il peut y avoir plus d'une rotation
  - expérimentalement : en moyenne une rotation pour 5 suppressions

PROBLÈME

- Si on souhaites gerer de grands ensembles d'éléments
  - Il est impossible de conserver toutes les infos en mémoire centrale
  - On est obligé d'utiliser un ou plusieurs disques dur
  - Chaque disque dur est découpé en secteurs
  - Les secteurs sont regroupés en blocs
  - On doit transférer en une seule opération des groupes de secteurs consécutifs

CONSTATS

- La taille d'un élément est généralement inférieure à un secteur de disque
- Il faut donc stockager plusieurs éléments dans un secteur ou bloc disque
- Le temps d'accès disque est important par rapport
  - aux opérations de traitement en mémoire centrale
- Il faut donc limiter au maximum
  - le nombre d'accès disque nécessaires pour effectuer une recherche
  - Et essayer de faire en sorte que les nœuds voisins
    - soient stockés dans un même bloc disque

ARBRES GÉNÉRAUX DE RECHERCHE

ARBRE GÉNÉRAUX DE RECHERCHE

- Généralisation des arbres binaires de recherche
- pour chaque nœud:
  - **plusieurs** valeurs de clés permettant de trier les nœuds fils
  - nombre de clés nécessaires = nb de fils – 1
- soient e_1, e_2, ..., e_n, ces clés
  - elles doivent être triées pour pouvoir faire effectivement une partition sur les éléments des descendants

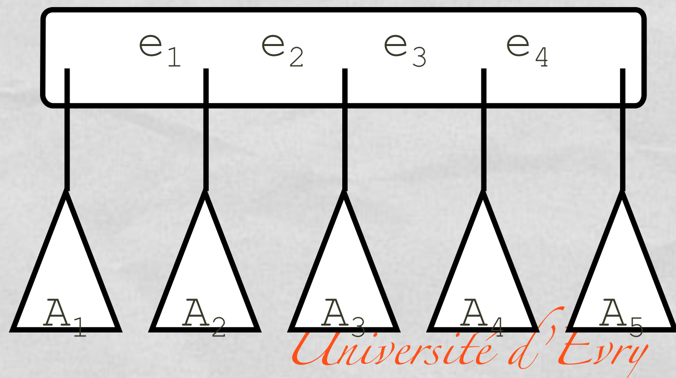
$A_1 \leq e_1 \leq A_2 \leq e2 \leq A_3 \leq e_3 \leq A_4 \leq e_4 \leq A_5$

e_1 e_2 e_3 e_4

A_1 A_2 A_3 A_4 A_5

OPERATIONS DE RECHERCHE

- Identique à celle des ABR sauf que pour chaque nœud
  - Soit la clé c cherchée est dans la liste des valeurs du nœud → fin
  - Soit il faut parcourir l'ensemble des clés
    - pour savoir dans quel sous-arbre se trouve la clé recherchée
      - si $e_2 < c < e_3$, alors si l'élément est dans l'arbre, il sera dans A_3



COMPLEXITE

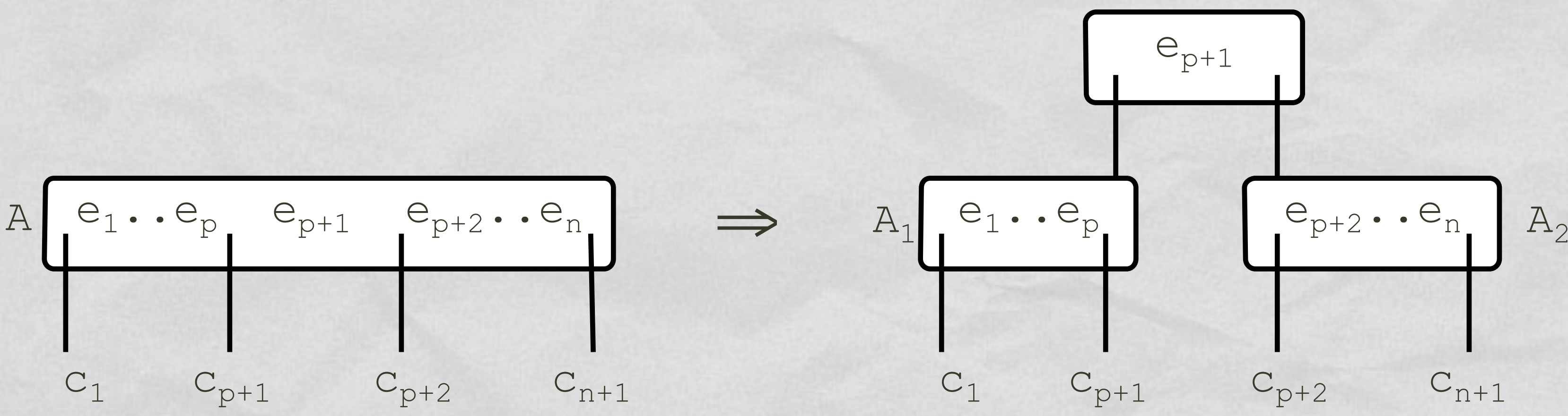
- Opération fondamentale
  - recherche du rang dans un nœud
  - lecture d'un nœud
- Complexité en O(h(A))
- Limitations
  - le nombre d'éléments d'un nœud doit être borné
  - la lecture d'un nœud doit se faire en une seule fois

OPERATIONS DE RE-EQUILIBRAGE

- But = conserver le degré des nœuds borné
- **éclatement** d'un arbre A:
  - dont le **degré est trop important** en
  - deux sous-arbre A_1 et A_2
  - séparés par un élément e issu de la liste de la racine de A
- **regroupement** d'un arbre A:
  - avec l'un de ses frères gauche et droit quand son degré est trop faible
- **répartition** entre deux sous-arbres voisins lorsque leurs degrés sont trop différents

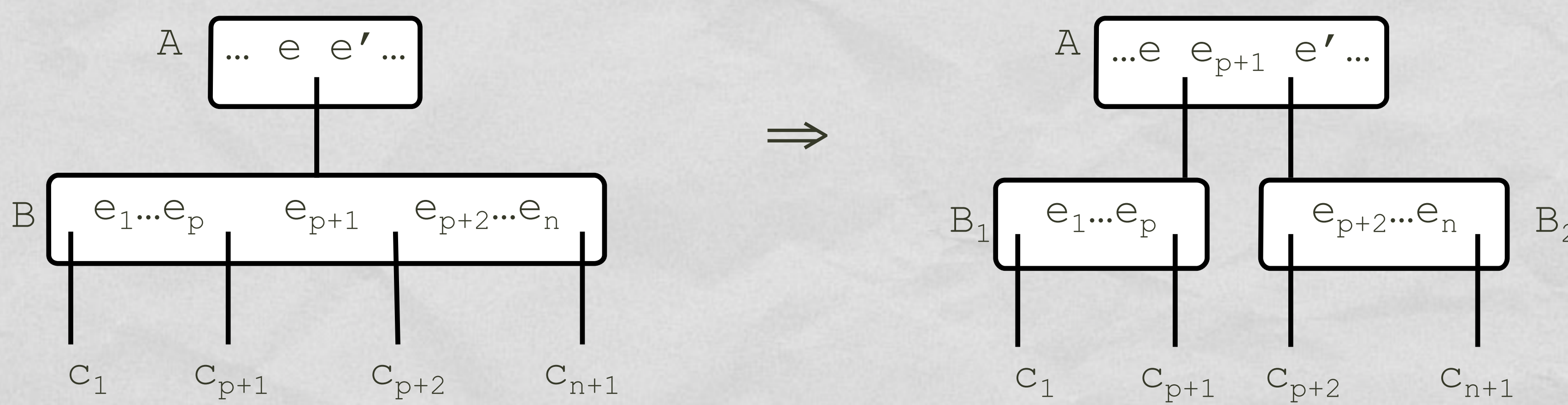
ECLATEMENT À LA RACINE

- éclatement de A en A_1 et A_2 séparés par e, issu de A
- augmente la hauteur de l'arbre de un



ECLATEMENT D'UN NOEUD

- éclatement de B en B_1 et B_2 séparés par e, issu de B
- construction de A' avec B remplacé par B_1 et B_2, et e inséré à la racine A
- le degré de A augmente de 1



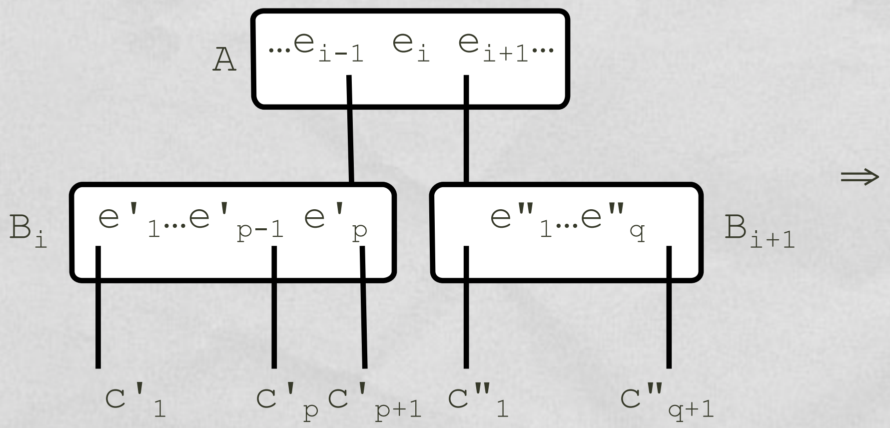
REGROUPEMENT

- Inverse de l'opération d'éclatement
- regroupement de deux sous-arbres avec la valeur qui les sépare

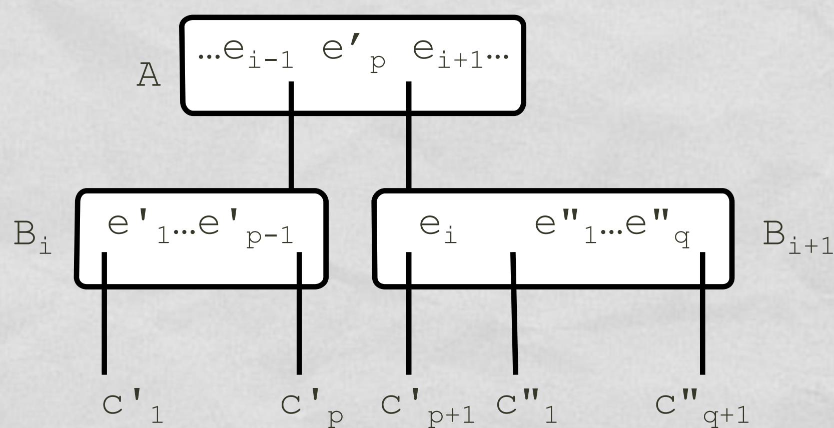
RÉPARTITION

soient deux sous-arbres Bi et Bi+1 séparés par ei

- transfert depuis l'arbre dont le degré est le plus élevé
- transfert de gauche à droite
  - faire descendre ei dans Bi+1
  - le remplacer par le dernier élément de Bi
  - transférer le dernier sous-arbre de Bi dans Bi+1



⇒



ARBRES BALANCÉS

Arbres généraux de recherche permettent:

de stocker un ensemble d'éléments

et de les retrouver par leur clé

Propriété :

si le degré des nœuds est borné,

la complexité de l'opération de recherche est au pire proportionnelle à la hauteur de l'arbre

Il est donc important de minimiser la hauteur de l'arbre

DÉFINITION

Un arbre balancé, ou B-arbre, d'ordre m est:

- un arbre général de recherche tel que:
  - toutes les feuilles soient au même niveau,
  - tout nœud contienne au plus 2m éléments
  - et tout nœud sauf la racine contienne au moins m éléments

PROPRIÉTÉS

Minorant de n

racine à au moins 1 élément

tout nœud a au moins m éléments

n ≥ 1 + 2m(1 + (m+1) + (m+1)2 + … + (m+1)h-1) = 2(m+1)h – 1

Majorant de n

tout nœud a au plus M éléments

n ≤ M(1 + (M+1) + (M+1)2 + … + (M+1)h) = (M+1)h+1 – 1

$$\log_{M+1}(n+1) - 1 \leq h \leq \log_{m+1}\left(\frac{n+1}{2}\right)$$

la hauteur d'un arbre de ce type contenant n éléments est telle que

PROPRIÉTÉS

On peut vérifier que les opérations d'éclatement, de regroupement et de répartition permettent de maintenir les propriétés sur le degré des nœuds

En pratique, m est compris entre 20 et 500, suivant la taille d'un élément et la taille d'un bloc mémoire

pour m=100

- 8 millions pour une hauteur de 2
- 1,6 milliards d'éléments pour une hauteur 3

OPÉRATIONS D'AJOUT

- on recherche l'élément dans l'arbre
- on rajoute l'élément dans la bonne feuille s'il n'existait pas
- si la feuille dans laquelle on insère le nouvel élément contient moins de 2m-1 éléments, on obtient un B-arbre
- sinon, c'est un arbre de recherche qu'il faut corriger pour retrouver un B-arbre

ECLATEMENT À LA REMONTÉE

Si la feuille dans laquelle on insère le nouvel élément contient déjà 2m éléments

- on la fait éclater en deux feuilles de degré m
- si la feuille n'est pas la racine, cela fait remonter un élément chez le père
- si le père contient déjà 2m éléments, il faut l'éclater à son tour, et ainsi de suite jusqu'à un ascendant contenant moins de 2m éléments ou jusqu'à la racine, elle-même éventuellement éclatée si elle contient déjà m élément, ce qui augmente la taille de l'arbre de 1

Inconvénient

- nécessite de conserver le chemin depuis la racine jusqu'à la feuille

ECLATEMENT À LA DESCENTE

- si on doit faire l'adjonction dans un sous-arbre qui contient 2m éléments
  - on le fait éclater
  - on continue l'adjonction dans celui des deux sous-arbres qui convient
- inconvénients
  - les sous-arbres peuvent devenir de degré m-1
  - on fait des éclatements qui sont peut-être inutiles

SUPPRESSION

- on recherche l'élément dans l'arbre
- si il se trouve dans une feuille, il est extrait de la liste de la feuille
- si il se trouve dans un nœud qui n'est pas une feuille, il doit être remplacé par un élément qui continue de faire la séparation entre les deux sous-arbres qui l'encadrent
  - soit i le rang de l'élément à supprimer, i et i+1 les rangs des sous-arbres, on peut choisir soit :
    - le + grand élément du sous-arbre i, le dernier élément de sa feuille la + à droite
    - le + petit élément du sous-arbre i+1, le premier élément de sa feuille la + à gauche
- Après suppression, il faut éventuellement corriger l'arbre pour en faire un B-arbre, en partant du sous-arbre i ou i+1 dans lequel a été choisi l'élément remplaçant l'élément supprimé, et en remontant vers le sommet par regroupement ou répartition

COMPLEXITÉ

- Le degré des nœuds est borné
  - le traitement d'un nœud a une durée constante
- Les opérations de recherche, d'ajout et de suppression ont une complexité :
  - exprimée en terme d'opérations de traitement d'un nœud
  - proportionnelle à la hauteur des arbres
  - donc $\leq \log_{m+1}((n+1)/2)$

COMPLEXITÉ

Le traitement d'un nœud nécessite

- un accès disque (lecture ou écriture)
- le traitement du nœud en mémoire centrale (recherche, ajout, suppression dans une liste de taille au plus 2m)

Remarques

- même si le traitement du nœud est en O(m), il restera inférieur au temps d'accès disque (10000 fois plus lents que les accès à la mémoire centrale)
- but de la diminution de la hauteur de l'arbre = diminuer le nb d'accès disques

COMPLEXITÉ

- Lors d'un ajout, il peut y avoir éclatement de nœuds en cascade jusqu'à la racine
  - 3 accès disque à chaque éclatement
  - nb d'éclatements borné par la hauteur de l'arbre
  - en moyenne 1 éclatement pour 1,38 ajouts
- Facteur important = taux de remplissage
  - rapport entre le nb d'éléments dans le B-arbre et le nb d'éléments qu'il pourrait contenir au maximum
    - compris entre 0,5 et 1
    - en moyenne de 0,7

BIBLIOGRAPHIE

Certaine partie de ce cours sont tiré du livre Introduction à l'Algorithmique