

# Les arbres-B

Nicolas Delestre, Géraldine Del Mondo

Fondé sur le cours de Michel Mainguenaud



arbre-B - v2.2

1 / 36

Définition

## Contexte

L'arbre-B (où *B-Tree* en anglais et *B* pour *balanced*) est un type de donnée utilisé dans les domaines des :

- systèmes de gestion de fichiers : ReiserFS (version modifiée des arbres-B) ou Btrfs (*B-Tree file system*)
- bases de données : gestion des index

L'arbre-B reprend le concept d'ABR équilibré mais en stockant dans un nœud  $k$  valeurs (nommées clés dans le contexte des arbres-B) et en référençant  $k + 1$  sous arbres :

- « minimise la taille de l'arbre et réduit le nombre d'opérations d'équilibrage » (Wikipédia)
- utile pour un stockage sur une unité de masse



arbre-B - v2.2

3 / 36

## Plan...

- 1 Définition
- 2 Recherche dans un Arbre-B
- 3 Insertion dans un Arbre-B
- 4 Suppression dans un Arbre-B



arbre-B - v2.2

2 / 36

Définition

## Arbre-B

### 🔦 Définition ARBRE-B

Un arbre-B d'ordre  $m$  est un arbre tel que :

- 1 Chaque nœud contient  $k$  clés triées avec :  $m \leq k \leq 2m$  (nœud non racine) et  $1 \leq k \leq 2m$  (nœud racine).
- 2 Chaque chemin de la racine à une feuille est de même longueur
- 3 Un nœud est :
  - Soit terminal (feuille)
  - Soit possède  $(k + 1)$  fils tels que les clés du  $i$ ème fils ont des valeurs comprises entre les valeurs du  $(i - 1)$ ème et  $i$ ème clés du père



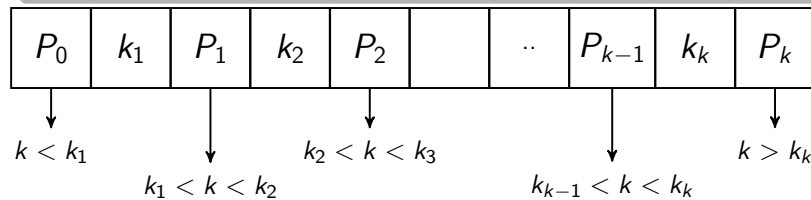
arbre-B - v2.2

4 / 36

## Structure d'un nœud

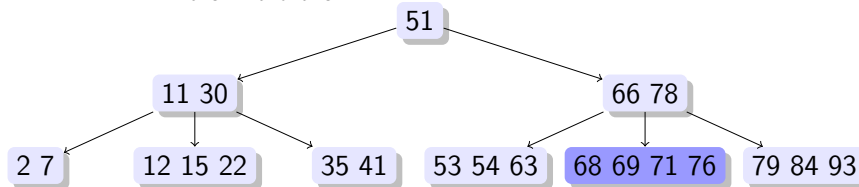
### 💡 Définition Nœud

- $k$  clés triées
- $k + 1$  références (pointeur mémoire, indice d'un cluster, etc. avec une valeur particulière, noté ici NIL, pour dénoter l'absence de référencement) tels que :
  - Tous sont différents de NIL si le nœud n'est pas une feuille
  - Tous à NIL si le nœud est une feuille



## Exemple

EXEMPLE : Arbre-B d'ordre 2



- Chaque nœud, sauf la racine contient  $k$  clés avec  $2 \leq k \leq 4$
- La racine contient  $k$  clé(s) avec  $1 \leq k \leq 4$

## Capacité

### • Nombre de clés

Arbre-B d'ordre  $m$  et de hauteur  $h$  :

→ Nombre de clé(s) minimal =  $2 * (m + 1)^h - 1$

→ Nombre de clés maximal =  $(2 * m + 1)^{h+1} - 1$

EXEMPLE :  $m = 100$ ,  $h = 2$  : Nombre maximal de clés  $\approx 8\,000\,000$

### • Stockage sur disque

→ Un nœud = Un bloc ou *cluster* (= ensemble de secteurs)

## Analyse

### TAD

**Nom :** ArbreB  
**Paramètre :** Cle (Possède un ordre total)  
**Opérations :**  
 arbreB: **NaturelNonNul**  $\rightarrow$  ArbreB  
 ordre: ArbreB  $\rightarrow$  **NaturelNonNul**  
 estPresent: ArbreB  $\times$  Cle  $\rightarrow$  **Booleen**  
 inserer: ArbreB  $\times$  Cle  $\rightarrow$  ArbreB  
 supprimer: ArbreB  $\times$  Cle  $\rightarrow$  ArbreB  
**Préconditions :**  
 inserer(a,c): non estPresent(a,c)  
**Axiomes :**  
 - ordre(arbreB(n))=n  
 - estPresent(inserer(a,c),c)  
 - non estPresent(supprimer(a,c),c)

### Conception préliminaire

- fonction arbreB (ordre : NaturelNonNul) : ArbreB
- fonction ordre (a : ArbreB) : NaturelNonNul
- fonction estPresent (a : ArbreB) : Booleen
- procédure insérer (E/S a : ArbreB, E c : Cle)
  - | précondition(s) non estPresent(a,c)
- procédure supprimer (E/S a : ArbreB, E c : Cle)

### Exemple de conception détaillée en mémoire

Type ArbreB = ^Noeud

Type Noeud = Structure

nbCles : NaturelNonNul

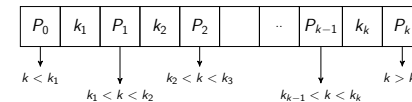
cles : Tableau[1..MAX] de Cle

sousArbres : Tableau[0..MAX] de ArbreB

finstructure

### Pour s'abstraire de l'implantation

- fonction feuille (cles : Liste<Cle>, ordre : NaturelNonNul) : ArbreB
  - | précondition(s)  $1 \leq \text{longueur}(\text{cles}) \leq 2 * \text{ordre}$
- fonction noeud (cles : Liste<Cle>, sousArbres : Liste<ArbreB>, ordre : NaturelNonNul) : ArbreB
  - | précondition(s)  $1 \leq \text{longueur}(\text{cles}) \leq 2 * \text{ordre}$   
 $\text{logueur}(\text{sousArbres}) = \text{longueur}(\text{cles}) + 1$   
 $\forall a \in \text{sousArbres}, \text{ordre} \leq \text{nbCles}(a) \leq 2 * \text{ordre}$
- fonction estUneFeuille (a : ArbreB) : Booleen
- fonction ordre (a : ArbreB) : NaturelNonNul
- fonction nbCles (a : ArbreB) : NaturelNonNul
- fonction cles (a : ArbreB) : Liste<Cle>
- fonction sousArbres (a : ArbreB) : Liste<ArbreB>
- fonction cle (a : ArbreB, pos : NaturelNonNul) : Cle
  - | précondition(s)  $\text{pos} \leq \text{nbCles}(a)$
- fonction sousArbre (a : ArbreB, pos : Naturel) : ArbreB
  - | précondition(s)  $\text{pos} \leq \text{nbCles}(a)$
- procédure changerCle (E/S a : ArbreB, E pos : NaturelNonNul, c : Cle)
  - | précondition(s)  $\text{pos} \leq \text{nbCles}(a)$
- procédure changerSousArbre (E/S a : ArbreB, E pos : Naturel, ssa : ArbreB)
  - | précondition(s)  $\text{pos} \leq \text{nbCles}(a)$



### Principe

À partir de la racine, pour chaque nœud examiné :

- La clé  $C$  est présente (recherche qui peut être dichotomique) → **succès**
- $C < k_1 \rightarrow$  recherche dans le sous-arbre le plus à gauche (via le pointeur  $P_0$ )
- $C > k_k \rightarrow$  recherche dans le sous-arbre le plus à droite (via le pointeur  $P_k$ )
- $k_i < C < k_{i+1}$  (recherche qui peut être dichotomique) → recherche dans le sous-arbre (via le pointeur  $P_i$ )
- Si l'arbre est vide (pointeur vaut NIL) → **échec**

```

fonction estPresent (a : ArbreB, c : Cle) : Booleen
debut
  si estVide(a) alors
    retourner FAUX
  sinon
    present, filsPossible ← estPresentDansRacine(a,c)
    si present alors
      retourner VRAI
    sinon
      si estUneFeuille(a) alors
        retourner FAUX
      sinon
        retourner estPresent(filsPossible, c)
      finsi
    finsi
  finsi
fin

```

```

fonction estPresentDansRacine (a : ArbreB, c : Valeur) : Booleen, ArbreB
  Déclaration g,d,m : NaturelNonNul
debut
  g ← 1
  d ← nbCle(a)
  tant que g ≠ d faire
    m ← (g+d) div 2
    si cle(a,m) ≥ c alors
      d ← m
    sinon
      g ← m+1
  finsi
  fintantque
  si cle(a,g) = c alors
    retourner VRAI, arbreB(ordre(a))
  sinon
    retourner FAUX, sousArbre(a,g-1)
  finsi
fin

```

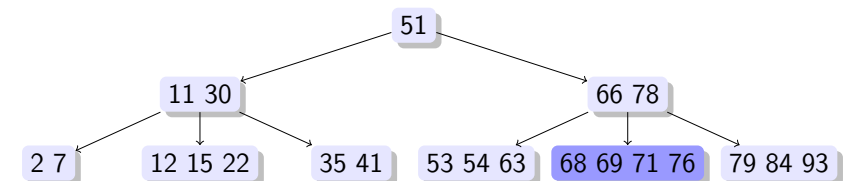
**Remarque**

$g$  et  $d$  référencent la position de l'élément supérieur ou égal à la valeur recherchée

**Principe**

- ① L'insertion se fait récursivement au niveau des feuilles
- ② Si un nœud a alors plus  $2m + 1$  clés, il y a éclatement du nœud et remontée (grâce à la récursivité) de la clé médiane au niveau du père
- ③ Il y a augmentation de la hauteur de l'arbre lorsque la racine se retrouve avec  $2m + 1$  clés  
 ⇨ l'augmentation de la hauteur de l'arbre se fait donc au niveau de la racine !

EXEMPLE : Insertion de **75** ?



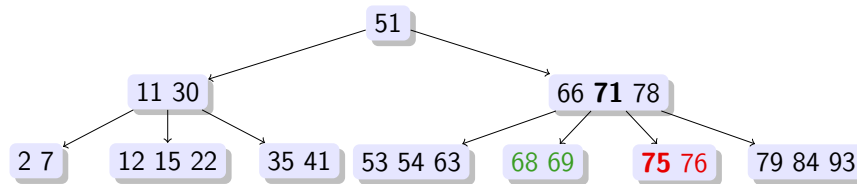
Rappel : ici nombre de clés par nœud  $\leq 4$

**Méthode**

- ① Eclatement du nœud en deux :
  - Les (deux) plus **petites** clés restent dans le nœud
  - Les (deux) plus **grandes** clés sont insérées dans un nouveau nœud
- ② Remontée de la clé médiane dans le nœud père (e.g. ici **71**)

## Exemple : cas d'un nœud plein 2 / 2

EXEMPLE : Insertion de **75** ?



Rappel : ici nombre de clés par nœud  $\leq 4$

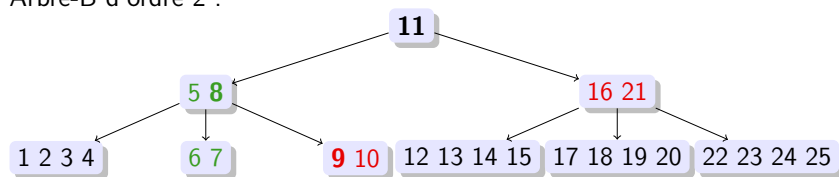
### Méthode

- 1 Eclatement du nœud en deux :
  - Les (deux) plus **petites** clés restent dans le nœud
  - Les (deux) plus **grandes** clés sont insérées dans un nouveau nœud
- 2 Remontée de la clé médiane dans le nœud père (e.g. ici **71**)

Insertion Arbre-B

## Exemple : cas de l'augmentation de la hauteur 2 / 2

Arbre-B d'ordre 2 :



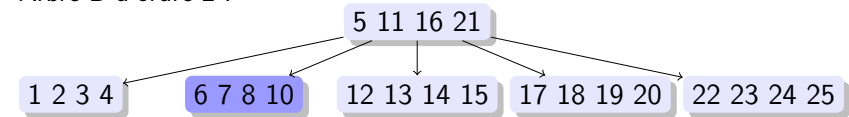
- Insertion clé **9** → Eclatement + remontée de la clé **8** au nœud père
- Remontée de la clé **8** au nœud père → Eclatement + création nouvelle racine (e.g. ici **11**)

↪ **Augmentation d'une unité de la hauteur**

Insertion Arbre-B

## Exemple : cas de l'augmentation de la hauteur 1 / 2

Arbre-B d'ordre 2 :



- Insertion clé **9** → Eclatement + remontée de la clé **8** au nœud père
- Remontée de la clé **8** au nœud père → Eclatement + création nouvelle racine (e.g. ici **11**)

Insertion Arbre-B

## Algorithme 1 / 2

### Prérequis

On suppose posséder les fonctions/procédures suivantes :

- **procédure** eclater (**E/S** a : ArbreB)  
 [précondition(s)] nbCles(a) = 2 \* ordre(a) + 1
- **fonction** positionDInsertion (a : ArbreB, c : Cle) : **NaturelNonNul**
- **procédure** insererCleDansFeuille (**E/S** a : ArbreB, E c : Cle, pos : **NaturelNonNul**)  
 [précondition(s)] estUneFeuille(a)
- **procédure** insererRacineDeTailleUnDansNoeud (**E/S** a : ArbreB, E racine : ArbreB, pos : **NaturelNonNul**)  
 [précondition(s)] nbCles(racine) = 1

## Algorithme 2 / 2

```

procédure insérer (E/S a : ArbreB, E c : Cle, ordre : NaturelNonNul)
  précondition(s) 2*ordre < MAX
  Déclaration ...
debut
  si estVide(a) alors
    l ← liste()
    insérer(l, c)
    a ← feuille(l)
  sinon
    pos ← positionDInsertion(a, c)
    si estUneFeuille(a) alors
      insérerCleDansFeuille(a, c, pos)
      si nbCles(a) > 2*ordre(a) alors
        éclater(a)
      finsi
    sinon
      ssArbre ← sousArbre(a, pos-1)
      insérer(ssArbre, c)
      si nbCles(ssArbre) = 1 alors
        insérerRacineDeTailleUnDansNoeud(a, ssArbre, pos)
        si nbCles(a) > 2*ordre(a) alors
          éclater(a)
        finsi
      finsi
    finsi
  fin

```



arbre-B - v2.2

21 / 36

Suppression dans un arbre-B d'ordre  $m$ 
**Principe**

- ① La suppression se fait toujours au niveau des feuilles  
 ⇨ Si la clé à supprimer n'est pas dans une feuille, alors la remplacer par la plus grande valeur des plus petites (ou plus petite valeur des plus grandes) et supprimer cette dernière
- ② Si la suppression de la clé d'une feuille (récursivement d'un nœud) amène à avoir moins de  $m$  clés :
  - ① Combinaison avec un nœud voisin (avant ou après)
  - ② Descente de la clé associant ces deux nœuds (éclatement du nœud si nécessaire et donc remonté d'une nouvelle clé)
 ⇨ la récursivité de ce principe peut amener à diminuer la hauteur de l'arbre (par le haut)

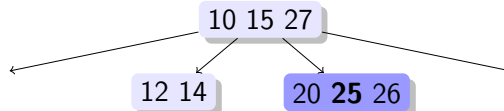


arbre-B - v2.2

22 / 36

## Ex. cas 1 : très simple

Arbre-B d'ordre 2 :

Rappel : ici nombre de clés par nœud non racine  $> 1$ ⇨ **EXEMPLE** : Suppression de la clé **25** ?**Méthode**

- ① Suppression de la valeur (décalage des valeurs dans le tableau)

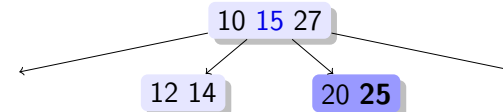


arbre-B - v2.2

23 / 36

## Ex. cas 2 : simple 1 / 2

Arbre-B d'ordre 2 :

Rappel : ici nombre de clés par nœud non racine  $\geq 2$ ⇨ **EXEMPLE** : Suppression de la clé **25** ?**Méthode**

- ① Combinaison avec un nœud voisin
- ② Descente de la clé (ici 15)
- ③ Suppression du nœud

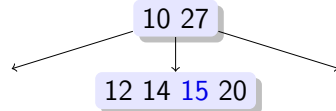


arbre-B - v2.2

24 / 36

## Ex. cas 2 : simple 2 / 2

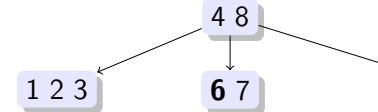
Arbre-B d'ordre 2 :

Rappel : ici nombre de clés par nœud non racine  $\geq 2$ ⇨ **EXEMPLE** : Suppression de la clé 25 ?**Méthode**

- ① Combinaison avec un nœud voisin ([12 14] et 20)
- ② Descente de la clé médiane (ici 15)
- ③ Suppression du nœud

## Ex. cas 3 : avec éclatement 1 / 2

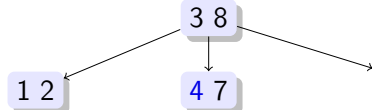
Arbre-B d'ordre 2 :

Rappel : ici nombre de clés par nœud non racine  $\geq 2$  et  $\leq 4$ ⇨ **EXEMPLE** : Suppression de la clé 6 ?

- Suppression clé 6 → Combinaison [1 2 3] et 7 + descente de la clé 4 au nœud fils
- Descente de la clé 4 au nœud fils → Redistribution + remontée de clé médiane (e.g. ici 3)

## Ex. cas 3 : avec éclatement 2 / 2

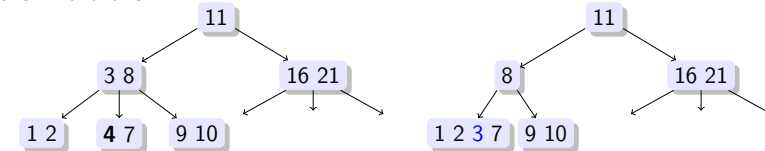
Arbre-B d'ordre 2 :

Rappel : ici nombre de clés par nœud non racine  $\geq 2$  et  $\leq 4$ ⇨ **EXEMPLE** : Suppression de la clé 6 ?

- Suppression clé 6 → Combinaison [1 2 3] et 7 + descente de la clé 4 au nœud fils
- Descente de la clé 4 au nœud fils → Redistribution + remontée de clé médiane (e.g. ici 3)

Ex. cas 4 : avec un nombre de clés inférieurs à  $m$  1 / 2

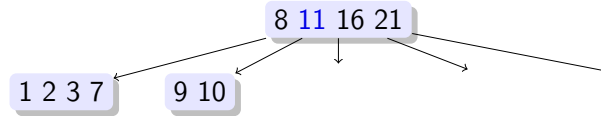
Arbre-B d'ordre 2 :

⇨ **EXEMPLE** : Suppression de la clé 4 ?

- Combinaison ([1 2] et 7) + descente de la clé 3

Ex. cas 4 : avec un nombre de clés inférieures à  $m$  2 / 2

Arbre-B d'ordre 2 :

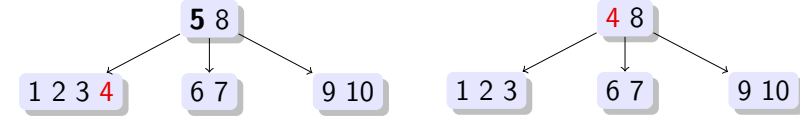
↪ **EXEMPLE** : Suppression de la clé 4 ?

- Combinaison + descente de la clé 3
- Combinaison (8 et [16 21]) + descente de la clé 11

↪ **Diminution d'une unité de la hauteur**

## Ex. cas 5 : suppression non feuille

Arbre-B d'ordre 2 :

↪ **EXEMPLE** : Suppression de la clé 5 ?**Méthode**

- 1 Recherche d'une clé adjacente **A** à la clé à supprimer → on choisit la plus **grande** du sous arbre gauche
- 2 Remplacement de la clé à supprimer par **A**
- 3 Suppression de la clé **A** du sous arbre gauche

## Algorithme 1 / 5

**Prérequis**

On suppose posséder :

- Les fonctions/procédures suivantes :
  - fonction plusGrandeCle (a : ArbreB) : Cle  
[précondition(s) non estVide(a)]
  - fonction positionCleDansNoeudRacine (a : ArbreB, c : Cle) : Entier  
[précondition(s) non estVide(a)]  
-1 si non présent
  - fonction positionSsArbrePouvantContenirCle (a : Arbre, c : Valeur) : Naturel  
[précondition(s) non estVide(a) et positionCleDansNoeudRacine(a,c)=-1]
  - fonction freresEtClesPeres (a : ArbreB, posSsArbre : Naturel) : ArbreB, ArbreB, Cle, Cle  
[précondition(s) non estVide(a) et non estVide(sousArbre(a, posSsArbre))]
  - procédure supprimerCleDansFeuille (E/S a : ArbreB, E c : Cle)  
[précondition(s) estUneFeuille(a)]
  - procédure fusionner (E/S a : ArbreB, E frere : ArbreB, frereGauche : Booleen, clePere : Cle)
- Type Fusion = {AUCUNE, GAUCHE, DROITE}



```

procédure supprimer (E/S a : ArbreB, E c : Cle)
  Déclaration typeFusion : Fusion, uneCleGauche, uneCleDroite : Cle
  debut
    supprimerR(a, c, arbreB(ordre(a)), arbreB(ordre(a)), uneCleGauche, uneCleDroite, typeFusion)
  fin
  
```

## Algorithme 2 / 5

**Cas simples**

```

procédure supprimerR (E/S a : ArbreB, E c : Cle, frereG, frereD : ArbreB, clePereGauche, clePereDroite : Cle, S
typeFusion : Fusion)
  Déclaration ...
  debut
    typeFusion ← AUCUNE
    si non estVide(a) alors
      pos ← positionCleDansNoeudRacine(a,c)
      si estUneFeuille(a) alors
        si pos ≠ -1 alors
          si estVide(frereG) et estVide(frereD) et nbCles(a)=1 alors
            liberer(a) // n'est possible que pour la racine
          sinon
            Algo #1 : supprimer c dans une feuille
          fin
        fin
      sinon
        si pos = -1 alors
          posSsArbre ← positionSsArbrePouvantContenirCle(a,c)
          sinon
            // cas # 5 des exemples
            cleRemplacement ← plusGrandeCle(sousArbre(a,pos-1))
            changerCle(a,pos,cleRemplacement)
            c ← cleRemplacement
            posSsArbre ← pos-1
          fin
          Algo #2 : supprimer c dans un sous-arbre de rang posSsArbre
        fin
      fin
    fin
  
```



## Algorithme 3 / 5

**Algo #1 : supprimer  $c$  dans une feuille**

```

supprimerCleDansFeuille(a,c)
si nbCles(a) < ordre(a) et (non estVide(frereG) ou non estVide(frereD)) alors
    si non estVide(frereG) alors
        fusionner(a, frereG, VRAI, clePereGauche)
        typeFusion ← GAUCHE
    sinon
        fusionner(a, frereD, FAUX, clePereDroite)
        typeFusion ← DROITE
    finsi
    si nbCles(a) > 2*ordre(a) alors
        eclater(a)
    finsi
finsi

```

## Algorithme 4 / 5

**Algo #2 : supprimer  $c$  dans le sous-arbre**

```

frereG, frereD, clePereG, clePereD ← freresEtClesPeres(a, posSsArbre)
temp ← sousArbre(a, posSsArbre)
supprimerR(temp, c, frereG, frereD, cle(a, posSsArbre), typeFusion)
si typeFusion ≠ AUCUN alors
    si nbCles(temp) = 1 alors
        // cas #3 des exemples : il y a eu éclatement après combinaison
        changerCle(a, posSsArbre, cle(temp, 1))
        changerSousArbre(a, posSsArbre-1, sousArbre(temp, 0))
        changerSousArbre(a, posSsArbre, sousArbre(temp, 1))
        liberer(temp)
    sinon
        si typeFusion = GAUCHE alors
            supprimerCleA(a, posSsArbre)
            supprimerSousArbreA(a, posSsArbre)
        sinon
            supprimerCleA(a, 1)
            supprimerSousArbreA(a, 1)
        finsi
        si nbCles(a) < ordre(a) alors
            // Cas #4
        finsi
    finsi
finsi

```

## Algorithme 5 / 5

**Fin Algo #2 : supprimer  $c$  dans le sous-arbre, cas #4**

```

si non estVide(frereG) ou non estVide(frereD) alors
    si non estVide(frereG) alors
        fusionner(a, frereG, VRAI, clePereGauche)
        typeFusion ← GAUCHE
    sinon
        fusionner(a, frereD, FAUX, clePereDroite)
        typeFusion ← DROITE
    finsi
    si nbCles(a) > 2*ordre(a) alors
        eclater(a)
    finsi
finsi

```

## Conclusion

## Comparaison AVL vs Arbre-B

AVL	Arbre-B
<ul style="list-style-type: none"> <li><b>Structure</b> : arbres binaires de recherche équilibrés, différence de hauteur entre les sous-arbres gauche et droit limitée à 1</li> <li><b>Équilibre</b> : maintenu en réajustant la hauteur des sous-arbres lors des opérations d'insertion et de suppression, à l'aide de simple et double rotation</li> <li><b>Recherche</b> : temps de recherche logarithmiques</li> <li><b>Utilisation</b> : mémoire</li> </ul>	<ul style="list-style-type: none"> <li><b>Structure</b> : arbres équilibrés généraux pouvant avoir plusieurs clés par nœud et plusieurs enfants</li> <li><b>Équilibre</b> : maintenu en redistribuant les clés entre les nœuds lors des opérations d'insertion et de suppression, conçus pour minimiser le nombre de mouvements nécessaires</li> <li><b>Recherche</b> : temps de recherche logarithmiques</li> <li><b>Utilisation</b> : systèmes de gestion de bases de données (SGBD) et les systèmes de fichiers (BTRFS, ZFS, HFS+)</li> </ul>