# Les B-arbres

Julie Parreaux

2018-2019

*Référence du développement :* Cormen [1, p.447]

*Leçons où on présente le développement :* 901 (Structures de données) ; 921 (Algorithmes de recherche) ; 932 (Base de données).

## 1 Introduction

On se place dans le cadre d'une recherche de clé dans une base de données stockée sur un disque dur externe. On cherche à minimiser le nombre d'action sur un disque externe (c'est ce qui prend le plus de temps, facteur d'un million entre un accès au disque et d'une opération sur processeur). La structure de données que l'on privilégie est un arbre de recherche. On cherche alors un arbre avec une ramification importante ce qui nous permet de minimiser le nombre d'appel au disque (hauteur plutôt faible ce qui minimise les quantités recherchées). En pratique on stocke une page (unité de transfert avec le disque dur) sur un noeud. Dans ce cas, on maximise l'impact de l'accès au disque.

Lors de ce développement, on se donne deux opérations atomiques : ECRITURE-DISQUE et LECTURE-DISQUE. On veut montrer l'optimalité de la structure de donnée face à l'utilisation de ces deux opérations.

Mettre un exemple dans le cas où on a un arbre de degré minimal $t = 2$ (Figure 1).

**Remarques sur le développement**

Ce développement présente une structure de données : il faut donc appuyer sur son implémentation "réelle".

1. Présentation des enjeux de cette structure de données.
2. Présentation de la recherche (algorithme et complexité).
3. Présentation de l'insertion (idées naïve et plus poussée).

## 2 Recherche dans un B-arbre

La recherche dans un B-arbre est analogue à la recherche dans un ABR en rajoutant des intervalles dans les nœuds. On commence alors à chercher dans quel intervalle dans lequel on se trouve, puis on descend.

L'algorithme prend en paramètre l'élément recherché $k$ et le noeud courant $x$. On retourne alors le noeud $x$ et la place de $k$ dans le noeud $x$, si $k$ est dans l'arbre. Sinon, on retourne NIL.

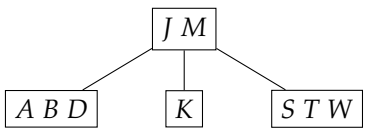



FIGURE 1 – Exemple d'un B-arbre de degré minimal 2.

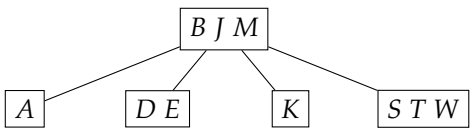



FIGURE 2 – Exemple de l'insertion de $E$ dans le B-arbre précédent.

**Algorithm 1** Recherche dans un B-arbre

```
1:  function RECHERCHE-BARBRE(x, k)                ▷ x a subit une lecture dans le disque LIRE-DISQUE
2:      i ← 1
3:      while i ≤ x.n et k > x.cle_i do            ▷ Recherche de l'intervalle dans lequel se trouve k
4:          i ← i + 1
5:          if i ≤ x.n et k = x.key_i then         ▷ On a trouvé k
6:              Retourne (x, i)
7:          else                                   ▷ On n'a pas trouvé k
8:              if x.feuille then                  ▷ On a fini l'arbre et k n'y est pas
9:                  Retourne NIL
10:             else   ▷ On peut continuer à descendre dans l'arbre : récursivité sur le fils de x
11:                 LIRE-DISQUE(x.c_i)
12:                 Retourne RECHERCHE-BARBRE(x.c_i, k)
13:             end if
14:         end if
15:     end while
16: end function
```

*Hypothèses pour le calcul de la complexité* :

— La racine du B-arbre se trouve toujours en mémoire principal : on n'a pas de LIRE-DISQUE ; cependant, il faudra effectuer un ÉCRITURE-DISQUE lors de la modification de la racine.

— Tout noeud passé en paramètre devra subir un LIRE-DISQUE.

**Théorème.** *Soit $T$ un B-arbre à $n$ élément de degré minimal $t \geq 2$. Alors la hauteur $h$ vérifie $h \leq \log_2 \frac{n+1}{2}$.*

*Démonstration.* — La racine a au moins une clé et les autres nœuds $t-1$.

— $T$ possède 2 nœuds à la profondeur 1, $2t$ à la profondeur 2,... Par récurrence, on montre que $T$ possède $2t^{h-1}$ nœuds à la profondeur $h$.

— On en déduit $n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)(\frac{t^h-1}{t-1}) = 2t^h - 1$.

— $t^h \leq \frac{(n+1)}{2}$ et on conclut en passant au $\log_t$.

□

Si on garde l'indice $t$ dans le log, il faut savoir le justifier : ce $t$ est très grand et donc $\log_t$ est relativement faible. Dans la vrai vie, $t$ vaut le nombre de tuple que l'on peut mettre sur une page (unité de transfert entre le disque dur et l'ordinateur).

*Complexité en accès au disque* : $O(h)$ où $h$ est la hauteur de l'arbre. Donc en $O(log_t n)$ lecture-écriture sur le disque (on ne fait que des lectures). Comme $n < 2t$, la boucle **while** s'effectue en $O(t)$, le temps processeur total est $O(th) = O(t \log n)$.

# 3 Insertion dans un B-arbre

L'insertion dans un B-arbre est plus délicate que dans une ABR. Comme dans un ABR, on commence par chercher la feuille sur laquelle on doit insérer cette nouvelle clé. Cependant, on ne peut pas juste créer une nouvelle feuille (on obtient un arbre illicite). On est alors obliger d'insérer l'élément dans un noeud déjà existant.

L'opération de l'ajout consiste en la recherche de la place de la clé dans les feuilles de l'arbre puis on insère dans la feuille de l'arbre correspondant.

**Problème** on peut violer la capacité du nœud.

**Solution naïve** on joute le nœud et on fait remonter la clé correspondante tant qu'on n'a pas trouvé de nœud capable de garder la clé. Cependant, il nous faut deux passages dans l'arbre : un pour trouver l'emplacement de la nouvelle clé et un pour remonter le superflu. Ce n'est pas optimal.

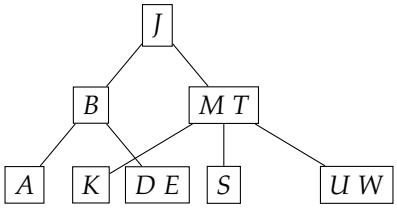



FIGURE 3 – Exemple de l'insertion de $U$ dans le B-arbre précédent.

**Idée** si le nœud est plein lors de la descente de l'arbre (recherche de l'emplacement), on sépare le nœud (on garanti qu'il n'y a aucun nœud plein sur notre parcours de l'arbre). On alors un unique passage.

Montrer un exemple : Figures 2 et 3
L'arbre grandi vers le haut à l'aide de la fonction PARTAGE quand la racine est pleine.
*Complexité en accès au disque* : $O(h) = O(\log_t n)$ où $h$ est la hauteur de l'arbre

# 4 Compléments sur les $B$-arbres

**Motivations** : Minimiser le nombre d'action sur un disque externe (c'est ce qui prend le plus de temps, en ms).

Un B-arbre est un arbre de la recherche avec une ramification importante et une hauteur plutôt faible. En pratique un noeud de notre arbre est une page de notre disque externe.

*Hypothèses* : on se donne deux opérations atomiques : ECRITURE-DISQUE et LECTURE-DISQUE (on ne peut pas les découper en sous fonctions, elles travaillent uniquement sur une page du disque dur). On veut montrer l'optimalité de la structure de donnée face à l'utilisation de ces deux opérations.

**Définition d'un B-arbre et exemple** On donne maintenant la définition d'un B-arbre. La figure 4 nous donne un exemple d'un tel arbre.

**Définition.** Un B-arbre est un arbre $T$ possédant les propriétés suivantes :

1. Chaque noeud $x$ contient les attributs ci-après :
   (a) $x.n$ le nombre de clés conservées dans le noeud (le noeud est alors d'arité $n+1$) ;
   (b) les $x.n$ clés $x.cle_1, \dots x.cle_n$ stockées dans un ordre croissant (structure de donnée associée : liste) ;
   (c) un booléen, $x.feuille$ marquant si le noeud est une feuille ;
   (d) les $n+1$ pointeurs des fils, $\{x.c_1, \dots x.c_{n+1}\}$ ;
2. Les clés et les valeurs dans les arbres fils vérifient la propriété suivante : $k_0 \leq x.cle_1 \leq \dots \leq x.cle_n \leq k_n$ ;
3. toutes les feuilles ont la même profondeur, qui est la hauteur $h$ de l'arbre ;
4. Le nombre de clé d'un noeud est contenu entre $t-1 \leq n \leq 2t-1$. (Pour la racine on n'a que la borne supérieure, la borne inférieure est 1 : on demande qu'il y ait au moins une clé dans la racine) où $t$ est le degré minimal du B-arbre.

*Remarque.* Dans la vrai vie, $t$ est choisi en fonction du nombre de tuples que peut contenir une page du disque.

**Théorème.** *Si $n \geq 1$, alors pour tout B-arbre $T$ à $n$ clés de hauteur $h$ et de degré minimal $t \geq 2$, $h \leq \log_2 \frac{n+1}{2}$.*

*Démonstration.* On montre par récurrence sur la hauteur de l'arbre que

$$n \geq 1 + (t-1) \sum_{i=1}^{h} 2t^{i-1} = 2t^h - 1$$

D'où $t^h \leq \frac{(n+1)}{2}$ et on conclut en passant au $\log_t$. □

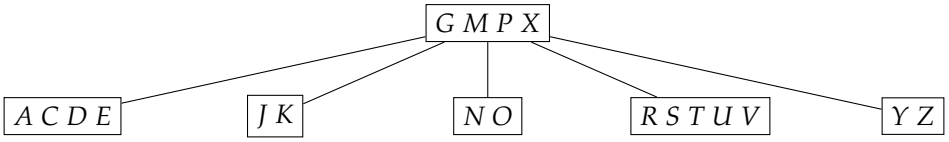



FIGURE 4 – Exemple d'un B-arbre de degré minimal $t = 3$.

*Hypothèses* :

— La racine du B-arbre se trouve toujours en mémoire principal : on n'a pas de LIRE-DISQUE ; cependant, il faudra effectuer un ÉCRITURE-DISQUE lors de la modification de la racine.

— Tout noeud passé en paramètre devra subir un LIRE-DISQUE.

**Recherche dans un B-arbre** La recherche dans un B-arbre est analogue à la recherche dans un ABR mais dans ce cas on prend une décision sur les $n$ clés de l'arbre, on cherche donc à placer la clé cherchée dans un des intervalles définis par les $n$ clés.

L'algorithme prend en paramètre l'élément recherché $k$ et le noeud courant $x$. On retourne alors le noeud $x$ et la place de $k$ dans le noeud $x$, si $k$ est dans l'arbre. Sinon, on retourne NIL.

**Algorithm 2** Recherche dans un B-arbre

```
1:  function RECHERCHE-BARBRE(x, k)          ▷ x a subit une lecture dans le disque LIRE-DISQUE
2:      i ← 1
3:      while i ≤ x.n et k > x.cle_i do       ▷ Recherche de l'intervalle dans lequel se trouve k
4:          i ← i + 1
5:          if i ≤ x.n et k = x.key_i then    ▷ On a trouvé k
6:              Retourne (x, i)
7:          else                              ▷ On n'a pas trouvé k
8:              if x.feuille then             ▷ On a fini l'arbre et k n'y est pas
9:                  Retourne NIL
10:             else        ▷ On peut continuer à descendre dans l'arbre : récursivité sur le fils de x
11:                 LIRE-DISQUE(x.c_i)
12:                 Retourne RECHERCHE-BARBRE(x.c_i, k)
13:             end if
14:         end if
15:     end while
16: end function
```

*Complexité* : $O(log_t n) = O(h)$ lecture-écriture sur le disque (on ne fait que des lectures) où $h$ est la hauteur et $n$ le nombre de nœuds. Comme $n < 2t$, la boucle **while** s'effectue en $O(t)$, le temps processeur total est $O(th) = O(t \log n)$.

**Insertion dans un B-arbre** L'insertion dans un B-arbre est plus délicate que dans une ABR. Comme dans un ABR, on commence par chercher la feuille sur laquelle on doit insérer cette nouvelle clé. Cependant, on ne peut pas juste créer une nouvelle feuille (on obtient un arbre illicite). On est alors obliger d'insérer l'élément dans un noeud déjà existant. Comme on ne peut pas insérer la clé dans un noeud plein, on introduit la fonction PARTAGE qui sépare le noeud plein en deux sous noeud distincts autours de la valeur médiane qui remonte dans le noeud père (à qui il faut peut-être appliquer PARTAGE). Cette fonction PARTAGE, nous permet, contrairement aux AVL ou arbre rouge-noir, de faire grandir l'arbre vers le haut à l'aide de la fonction PARTAGE. Nous conservons alors un équilibre "parfait".

*Principe de l'algorithme* : On descend alors dans l'arbre pour chercher l'emplacement de la nouvelle clé, de manière analogue à la fonction RECHERCHE-BARBRE. On souhaite alors insérer la clé dans la feuille que l'on trouve. Dans le cas où cette feuille est pleine, nous devons appliqué la fonction PARTAGE qui conduit à insérer une clé dans le noeud père. Dans le pire des cas, on est alors amené à remonter entièrement l'arbre avec cette fonction PARTAGE. En implémentant intelligemment la fonction INSERTION-BARBRE, nous pouvons faire qu'une descente dans l'arbre. Lors de la descente de l'arbre

nous allons appliquer PARTAGE à tous les nœuds pleins que nous parcourons. On effectue la fonction PARTAGE avant toutes opérations d'insertion dans l'arbre (la parité du nombre de clé nous permet de séparer le noeud plus facilement).

La fonction PARTAGE qui effectue un copié-collé des bon pointeurs, prend en paramètre un noeud $x$ (en mémoire vive) et un paramètre $i$ qui indique le noeud que l'on doit séparer ($x.c_i$). Elle modifie alors $x$ en lui ajoutant la valeur médiane de $x.c_i$, $x.c_i$ en lui retirant la moitié de ces valeurs et créer un nouveau noeud $z$ fils de $x$ contenant les valeurs restantes.

**Algorithm 3** Partage un noeud plein dans un B-arbre

```
1:  function PARTAGE(x, i)                          ▷ x a une lecture dans le disque LIRE-DISQUE
2:      z ← ALLOUER-NOEUD() ▷ z va récupérer les valeurs les plus grandes et devenir un enfant de x
3:      y ← x.c_i                                   ▷ Récupère les plus petites valeurs
4:      z.feuille ← y.feuille ; z.n ← t − 1
5:      for j = 1 à t − 1 do                        ▷ Fabrication du noeud z : partage de ces clés
6:          z.cle_j ← y.cle_{j+t}
7:      end for
8:      if non y.feuille then                       ▷ Fabrication du noeud z : partage de ces fils
9:          for j = 1 à t do
10:             z.c_j ← y.c_{j+1}
11:         end for
12:     end if
13:     y.n ← t − 1
14:     for j = x.n + 1 décrois jusqu'à i + 1 do    ▷ Modification de x pour la nouvelle valeur : fils
15:         x.c_{j+1} ← x.c_j
16:     end for
17:     x.c_{i+1} ← z
18:     for j = x.n décrois jusqu'à i do            ▷ Modification de x pour la nouvelle valeur : clés
19:         x.cle_{j+1} ← x.cle_j
20:     end for
21:     x.cle_i ← y.cle_t
22:     x.n ← x.n + 1
23:     ÉCRIRE-DISQUE(x) ; ÉCRIRE-DISQUE(y) ; ÉCRIRE-DISQUE(z)   ▷ Partage du résultat
24: end function
```

*Complexité* : $O(1)$ en lecture-écriture sur le disque (on ne fait que trois écritures et deux lecture et une allocation qui utilise ALLOUER-NOEUD : alloue une nouvelle page sur le disque en $O(1)$.). Par les deux boucles **for** en $\Theta(t)$, le temps processeur total est $\Theta(t)$.

On définit la fonction INSERTION-INCOMPLET-BARBRE (une sous fonction de INSERER-BARBRE) qui insère $k$ dans un noeud d'un B-arbre de racine $x$, supposé non plein. L'utilisation de cette fonction dans l'insertion globale garantie l'hypothèse.

*Complexité* : $O(log_t n) = O(h)$ lecture-écriture sur le disque (on ne fait que $O(1)$ de lecture-écriture entre deux appels récursif) où $h$ est la hauteur et $n$ le nombre de nœuds. Comme $n < 2t$, les boucles **while** s'effectue en $O(t)$, le temps processeur total est $O(th) = O(t \log n)$.

On définit la fonction INSERTION-BARBRE qui insère $k$ dans B-arbre $T$. On utilise PARTAGE et INSERTION-INCOMPLET-BARBRE pour assurer que la récursivité ne descendent jamais sur un noeud plein. Cela nous assure qu'on a toujours une unique descente dans l'arbre.

*Complexité* : $O(log_t n) = O(h)$ lecture-écriture sur le disque (on ne fait qu'une seule descente de l'arbre) où $h$ est la hauteur et $n$ le nombre de nœuds. Le temps processeur total est $O(th) = O(t \log n)$.

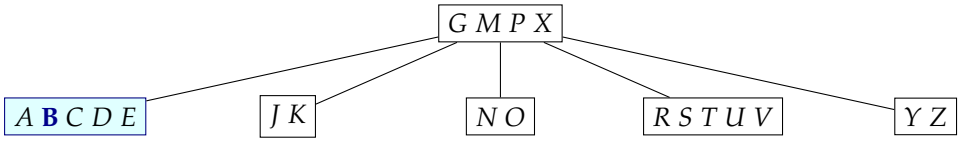
**Suppression dans un B-arbre** La suppression dans un B-arbre est plus délicate que l'insertion car nous sommes amenés à potentiellement modifier tous les nœuds internes (et plus seulement les feuilles). De manière analogue à l'insertion, la suppression peut construire un arbre avec un noeud illégal (ne effet, il peut devenir trop petit). Un algorithme naïf pourrait avoir tendance à rebrousser chemin quand un noeud devient trop petit (et à faire deux plusieurs aller-retour dans l'arbre). Nous allons présenter un algorithme qui se fait en une seule passe et ne remonte pas dans l'arbre (sauf lorsque nous

**Algorithm 4** Insertion dans un noeud non-plein d'un B-arbre

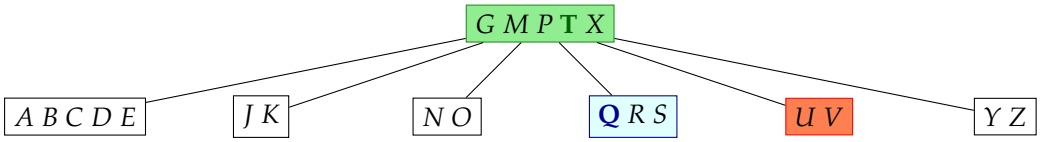
```
1:  function INSERTION-INCOMPLET-BARBRE(x, k)        ▷ x a une lecture dans le disque LIRE-DISQUE
2:      if x.feuille then                            ▷ Il faut qu'on insère k dans x
3:          while i ≥ n et k < x.cle_i do
4:              x.cle_{i+1} ← x.cle_i ; i ← i − 1
5:          end while
6:          x.cle_{i+1} ← k ; x.n ← x.n + 1
7:          ÉCRIRE-DISQUE(x)
8:      else                                         ▷ k doit être insérer dans un fils de x ; récursivité
9:          while i ≥ n et k < x.cle_i do
10:             i ← i − 1
11:         end while
12:         i ← i + 1
13:         ÉCRIRE-DISQUE(x.c_i)
14:         if x.c_i.n = 2t − 1 then
15:             PARTAGE(x, i)
16:             if k > x.cle_i then
17:                 i ← i + 1
18:             end if
19:         end if
20:         INSERTION-INCOMPLET-BARBRE(x.c_i, k)
21:     end if
22: end function
```

**Algorithm 5** Insertion d'un élément dans un B-arbre

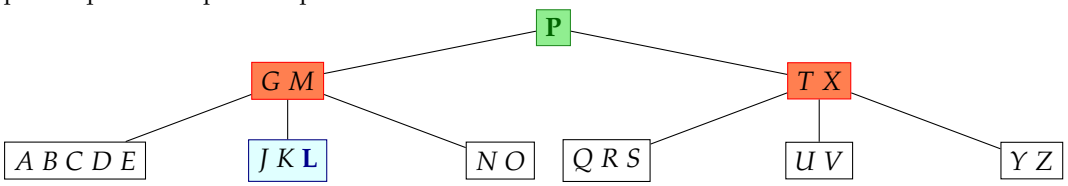
```
1:  function INSERTION-BARBRE(T, k)                  ▷ La racine de l'arbre est en mémoire vive
2:      r ← T.racine
3:      if r.n = 2t − 1 then                         ▷ La racine est pleine
4:          s ← ALLOUER-NOEUD()
5:          T.racine ← s ; s.feuille ← FAUX
6:          x.n ← 0 ; x.c_1 ← r
7:          PARTAGE(s, 1)
8:          INSERTION-INCOMPLET-BARBRE(s, k)
9:      else
10:         INSERTION-INCOMPLET-BARBRE(r, k)
11:     end if
12: end function
```



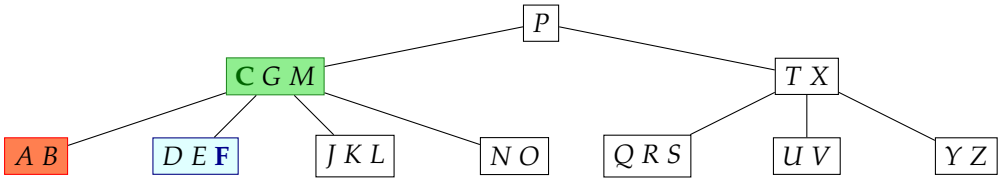

(a) Insertion de la lettre B dans le B-arbre (Figure 4). Le noeud en bleu contient cette nouvelle clé.




(b) Insertion de la lettre Q dans le B-arbre précédent. Le noeud bleu contient cette nouvelle clé, le noeud vert est le noeud qui a hérité de clé suite à l'action de PARTAGE et le noeud rouge est le nouveau noeud crée suite à PARTAGE pour lequel on n'a pas manipulé ces valeurs.




(c) Insertion de la lettre L dans le B-arbre précédent. Le noeud bleu contient cette nouvelle clé. Comme la racine était pleine, nous lui avons appliqué l'opération PARTAGE : le noeud vert est le noeud (la nouvelle racine) qui a hérité de clé suite médiane fait grandir l'arbre vers le haut. Les nœuds rouges sont les deux fils crées suite à PARTAGE pour lequel on n'a pas manipulé leurs valeurs.




(d) Insertion de la lettre L dans le B-arbre précédent. Le noeud bleu contient cette nouvelle clé. Comme la racine était pleine, nous lui avons appliqué l'opération PARTAGE : le noeud vert est le noeud (la nouvelle racine) qui a hérité de clé suite médiane fait grandir l'arbre vers le haut. Les nœuds rouges sont les deux fils crées suite à PARTAGE pour lequel on n'a pas manipulé leurs valeurs.

FIGURE 5 – Insertions dans un B-arbre.

devons rétrécir la hauteur de l'arbre). Dans le cas où un noeud se retrouverait sans clé alors il est supprimé et son unique fils devient la racine de l'arbre (la hauteur de l'arbre diminue de 1). L'arbre rétrécie par le haut (comme il grandit).

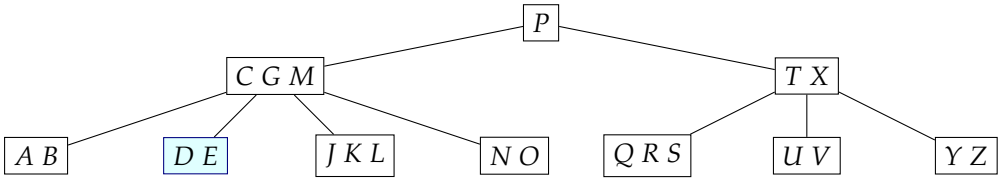
*Principe de la suppression* : On cherche la clé que l'on souhaite supprimer. Comme pour l'insertion, nous nous assurons que nous pouvons descendre dans l'arbre et supprimer la clé en toute légalité.

1. Si la clé $k$ est dans le noeud $x$ qui est une feuille : on supprime $k$.
2. Si la clé $k$ est dans le noeud $x$ qui n'est pas une feuille :
   (a) Si l'enfant $y$ qui précède $k$ a au moins $t$ clés, on cherche le prédécesseur de $k$, $k'$ (max dans la liste des clés de $y$). On supprime récursivement $k'$ dans $y$ et on remplace $k$ par $k'$. Permet de conserver l'arité du noeud $x$.
   (b) Si l'enfant $y$ qui précède $k$ a $t-1$ clés, on examine symétriquement le fils suivant $z$. Si $z$ a au moins $t$ clés, on cherche le successeur de $k$, $k'$ (min dans la liste des clés de $z$). On supprime récursivement $k'$ dans $z$ et on remplace $k$ par $k'$. Permet de conserver l'arité du noeud $x$.
   (c) Si $y$ et $z$ ont $t-1$ clés, on fusionne $k$ dans le contenu de $z$ et on fait tout passer (en copiant) dans $y$ qui contient $2t-1$ clés ($x$ perd son pointeur vers $z$ et $k$). On libère $z$ et on supprime récursivement $k$ dans $y$.
3. Si la clé $k$ n'est pas dans le noeud $x$, on cherche le sous arbre $x.c_i$ qui devrait contenir $k$ (si il est dans l'arbre). Si $x.c_i.n = t-1$ on exécute un des deux sous-cas suivants (selon les besoin) pour garantir que l'on descend dans un noeud à au moins $t$ clés. On applique alors la récursivité sur l'enfant approprié de $x$.
   (a) Si $x.c_i$ n'a que $t-1$ clé mais qu'un de ces frères immédiats, $y$ à au moins $t$ clés, on bascule une des clés de $y$ (min ou max en fonction du côté) dans $x$ et on bascule la clé supplémentaire de $x$ dans $x.c_i$. Puis on reconnecte les pointeur la où il faut.
   (b) Si $x.c_i$ et ses frères immédiats n'ont que $t-1$ clés, nous fusionnons les deux frères en descendant la bonne clé de $x$ (devient la clé médiane du noeud).

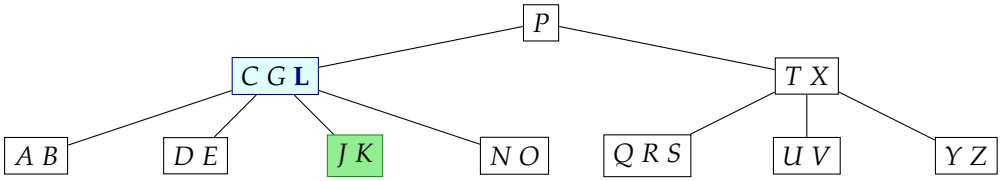
*Complexité* : $O(log_t n) = O(h)$ lecture-écriture sur le disque (on ne fait que $O(1)$ de lecture-écriture entre deux appels récursif) où $h$ est la hauteur et $n$ le nombre de nœuds. Comme $n < 2t$, les boucles **while** s'effectue en $O(t)$, le temps processeur total est $O(th) = O(t \log n)$.

# Références

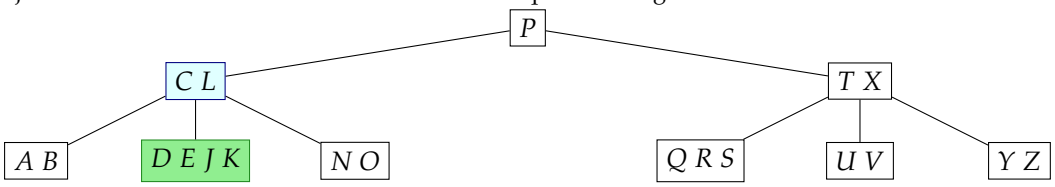
[1] Rivest R. Stein C. Cormen T., Leiserson C. *Algorithmique, 3ème édition*. Dunod, 2010.



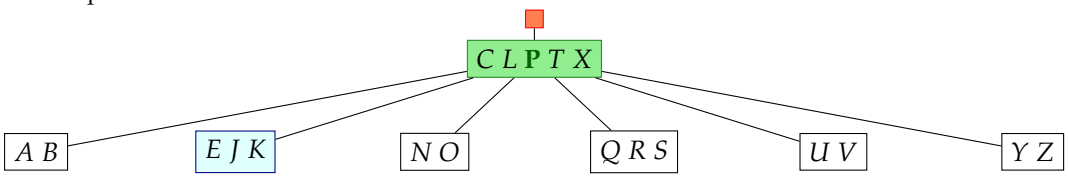

(a) Suppression de la lettre F (**cas 1**) dans le B-arbre (Figure 5). Le noeud en bleu contenait la clé supprimée, c'est le noeud que nous avons manipulé.



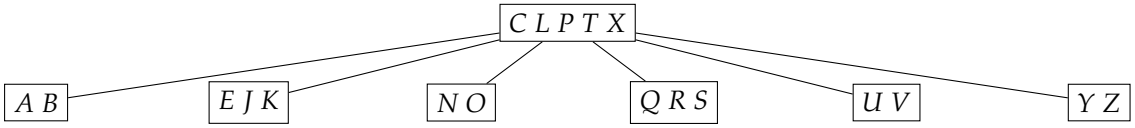

(b) Suppression de la lettre M dans le B-arbre précédent (**cas 2a**). Le noeud bleu contenait la clé supprimé. On lui a ajouter une nouvelle clé issue du noeud vert afin qu'il reste légal.



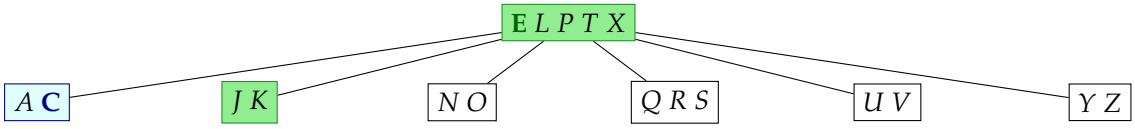

(c) Suppression de la lettre G dans le B-arbre précédent (**cas 2c**). Le noeud bleu contenait la clé supprimé. Le noeud vert est le résultat (légal) de la fusion des deux fils de la clé enlevée : on ne pouvait pas appliquer une rotation comme précédemment.




(d) Suppression de la lettre D dans le B-arbre précédent (**cas 3b**). Comme on ne peut pas descendre dans la récursivité (car le noeud *C L* ne contient que deux clés) nous avons fusionner les deux fils de la racines en rajoutant la clé *P*. Le noeud vert est le résultat de cette fusion. Le noeud bleu contenait la clé supprimé. Le noeud rouge quant à lui est le noeud de la racine (vide).




(e) Suppression de la racine vide. Dans ce cas la hauteur de l'arbre est diminue de 1.




(f) Suppression de la lettre D dans le B-arbre précédent (**cas 3a**). Comme on ne peut pas descendre dans la récursivité (car le noeud *A B* ne contient que deux clés) mais nous ne pouvons pas fusionner les deux fils de la racine car le fils gauche contient trois clés. Les nœuds verts est le résultat d'une rotation des clés. Le noeud bleu contenait la clé supprimé.

FIGURE 6 – Suppressions dans un B-arbre.