# B-arbres

*(inspiré de la version de Julie Parreaux)*

**Leçons : 901 ; 921 ; 932**

**Références : CORMEN–LEISERSON–RIVEST–STEIN, *Introduction à l'algorithmique* (p.425) [?] et LESESVRE–MONTAGNON–LE BARBENCHON–PIERRON, *131 développements pour l'oral* (p. 658) [?]**

**Prérequis :**

- les opérations élémentaires [1] pour les arbres binaires de recherche.

On donne la définition d'un B-arbre :

**Définition 1.** 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 :
$$k_0 \leq x.cle_1 \leq \cdots \leq x.cle_n \leq k_n$$
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.

**Remarques :**
Ne pas réécrire la définition au tableau, il faut qu'elle figure dans le plan.

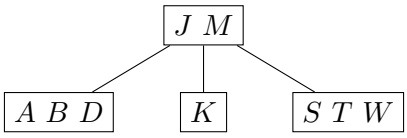
**Introduction :**

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.

1. recherche, insertion, suppression

Exemple avec $t = 2$ :




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

## 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 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.

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.

**Algorithme 1** Recherche B-arbre

```
Procédure RECHERCHE-BARBRE(x, k)
    [x a subi une lecture dans le disque LIRE-DISQUE]
    i ← 1 ;
    Tant que (i ≤ x.n et k > x.cle_i) faire
        [Recherche de l'intervalle dans lequel se trouve k]
        i ← i + 1 ;
    Fait
    Si (i ≤ x.n et k = x.cle_i) alors
        [On a trouvé k]
        Retourne (x, i)
    Sinon
        [On n'a pas trouvé k]
        Si (x.feuille) alors
            [On a fini l'arbre et k n'y est pas]
            Retourne NIL
        Sinon
            [On peut continuer à descendre dans l'arbre : récursivité sur le fils de x]
            LIRE-DISQUE(x.c_i)
            Retourne RECHERCHE-BARBRE(x.c_i, k)
        Fin Si
    Fin Si
Fin
```

**Théorème 1.** Soit $T$ un B-arbre à $n$ éléments de degré minimal $t \geq 2$. Alors la hauteur $h$ vérifie

$$h \leq \log_t \frac{n+1}{2}$$

*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 $\log_t$.

□

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 $O(th) = O(t \log n)$.

## 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 obligé 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.

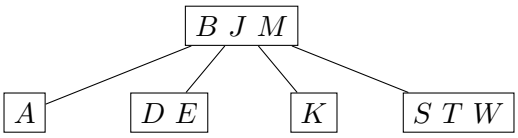
On peut avoir le **problème** suivant : on peut violer la capacité du nœud.

Pour palier ce problème, on pourrait séparer le nœud en deux et on fait remonter la clé médiane dans le noeud du dessus 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.

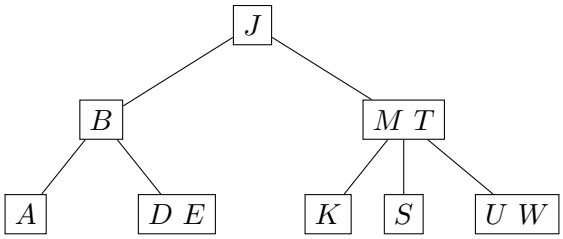
On peut faire **mieux** : si le nœud est plein lors de la descente de l'arbre (recherche de l'emplacement), on sépare le nœud (on garanti qu'il n'y a aucun nœud plein sur notre parcours de l'arbre). On alors un unique passage.

On peut avoir un **nouveau problème** si la racine est pleine, alors on partagera la racine ce qui aura pour effet d'augmenter la hauteur de l'arbre de 1.

Insertion de $E$ dans l'arbre précédent :




Partage de la racine qui augmente la hauteur de l'arbre dû à l'ajout de $U$ :




Complexité en accès au disque : $O(h) = O(\log_t n)$ où $h$ est la hauteur de l'arbre.

**Remarques :**

La suppression est encore plus compliquée à mettre en place que l'insertion, lire les pages du Cormen [?].

**Astuces de l'agrégatif :**

Ce développement est assez long surtout si on veut donner l'algorithme d'insertion, j'ai fait le choix d'essayer d'expliquer le plus clairement possible, comment on insère, quelles sont les difficultés liées à l'insertion pour préserver la structure, etc. Je ne pense pas que ce soit faisable de traiter la suppression en plus.

**Questions possibles :**

- Donner les algorithmes de recherche, d'insertion et de suppression pour un arbre binaire de recherche.