Recherche et insertion dans un B-arbre

Antoine Dequay

21 septembre 2022

Notes

- Prof : Loïc Hélouët.
- Leçon : 901, 921, 932.
- Référence :
  - Le Barbenchon.

Théorème 1 La recherche dans un B-arbre se fait en $O(\log(n))$.

Preuve. Ici, n désigne le nombre de clés stockées dans un B-arbre. Un B-arbre représentant une base de données externe en général, on calculera la complexité des algorithmes en fonctions des appels aux fonctions $LIRE$ et $ECRIRE$. En outre, on suppose qu'on a accès, pour x un noeud et T un arbre, à :

— $x.feuille$, test booléen,

— $x.n$, correspondant au nombre de clés contenues dans x,

— $x.cle$, le tableau trié des clés de x,

— $x.fils$, le tableau trié des fils de x,

— $T.racine$, donnant la racine de l'arbre. On n'appelle pas $LIRE$ pour cette fonction, mais en cas de modification, on doit utiliser $ECRIRE$.

L'algorithme utilisé suit le même schéma que l'algorithme de recherche sur un arbre de recherche classique :

```
RECHERCHE(x = T.racine, c):   # x est un noeud de T, initialisé à la racine, et
    c la clé à chercher.
    i = 1  # On cherche l'intervalle dans lequel c peut apparaître
    while i <= x.n and c > x.cle[i]:
        i += 1
    if i <= x.n and c = x.cle[i-1]:
        return (x,i)
    if x.feuille:
        return NIL
    LIRE(x.fils[i-1])
    return RECHERCHE(x.fils[i-1], c)
```

On note t l'ordre d'un B-arbre. Alors, par définition, on a $1 \leq T.racine.n$ et $t-1 \leq x.n$ pour tout autre noeud x. Ainsi, T possède au moins $2t^{k-1}$ noeuds (ou feuilles pour $k=h$) à la profondeur k, pour $k \in [\![0,h]\!]$ où h désigne la hauteur de l'arbre. On a donc :

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

d'où

$$h \leq \log_t\left(\frac{n+1}{2}\right).$$

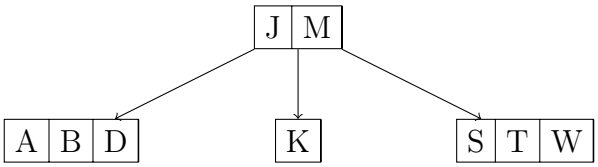
Au vu de la structure de l'algorithme, celui-ci a une complexité temporelle en $O(h)$, d'où un $O(\ln(n))$.

□

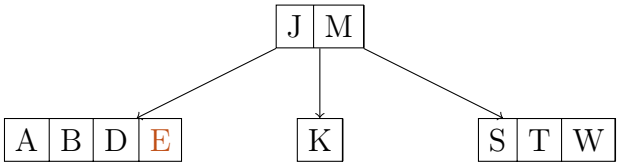
Décrivons à présent l'algorithme d'insertion [1].

1. pour l'écriture formelle des algorithmes, se rapprocher de la référence.

On se donne le B-arbre d'ordre 2 suivant :

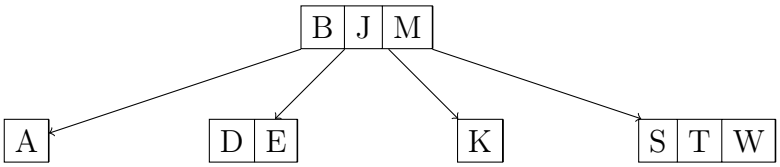


Commençons par essayer de lui ajouter "E". Pour cela, on commence par chercher l'endroit où le placer dans une feuille de l'arbre, via l'algorithme de recherche.

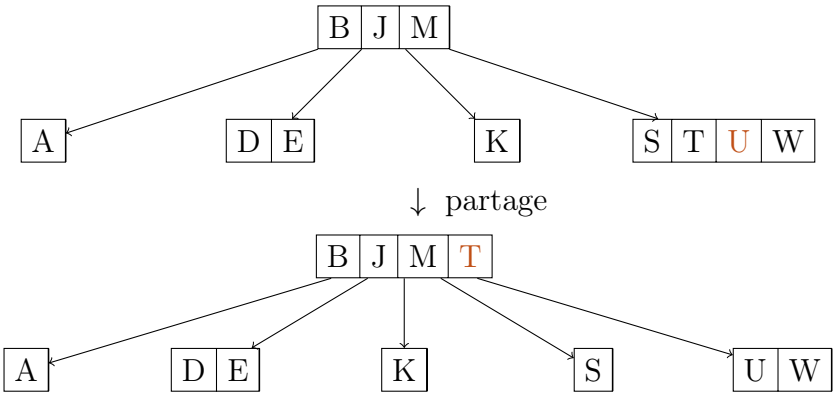


Si la feuille en question contient des "cases vides", l'algorithme peut s'arrêter. Sinon, on utilise une fonction partage.

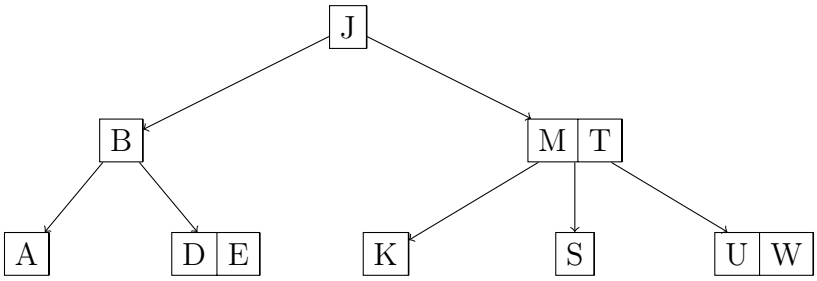
A partir d'un noeud possédant une clé de plus que l'ordre de l'arbre, celui ci va "couper" en 3 morceaux : un premier noeud de taille $t-1$, puis une case unique et enfin un noeud de taille t, et va réarranger l'arbre comme suit :



Cette fonction "partage" permet de préserver la structure de l'arbre : les feuilles restent toutes à même hauteur. Si le noeud sur lequel on a fait remonter la case isolée est lui-même plein, on réapplique la fonction à celui-ci, et on itère, jusqu'à arriver à la racine de l'arbre.



Si le problème se pose à la racine, la case isolée lors du découpage devient la nouvelle racine de l'arbre. La hauteur est donc augmentée de 1, mais la structure est préservée.



Remarque On peut prouver que c'est algorithme se fait également en $O(\ln(n))$: La recherche est en $O(\ln(n))$, le partage en $O(1)$ et on doit en faire au maximum un par niveau de l'arbre, d'où un $O(h) = O(\ln(n))$.