# Arbre B

## Introduction

Les arbres B ont été crées en 1970 par Rudolf Bayer et Edward M.McCreight. La lettre pourrait avoir différente signification tel que « balanced », « Bayer » ou bien « Boieng ». L'intérêt d'un arbre B est le fait qu'il possède plusieurs enfants, limitant ainsi les calculs pour le rééquilibrage de l'arbre, comme on pourrait l'avoir dans un arbre de recherche. La différence entre un arbre b et b+ réside dans le stockage des données, dans un arbre b+ les données se trouvent uniquements dans les nœuds feuilles. Ce qui change les temps de calculs ainsi que la lecture de l'arbre.

## Les algorithmes

Les exemples des différents algorithmes et fonctions se trouvent au sein de la documentation python pour cela il suffit d'éxécuter le fichier NodeB.py avec pyCharm/Thonny ou la commande dans le dossier src/ :

```
python3 NodeB.py
```

Pour les tests ils se trouvent dans Arbreb.py.

### 1) L'algorithme de recherche

Pour l'algorithme de recherche, on utilise une fonction nommée whichIndex qui permet de situer dans notre arbre où pourrait se trouver la valeur. Dans notre cas, on compare un à un les valeurs et quand on trouve une valeur supérieur on s'arrête en retournant l'indice. Celui-ci n'est pas optimal, en utilisant la dichotomie serait bien meilleur en temps, on passerait d'une complexité de O(n) à O(log(n)). Lorsque la valeur ne se trouve pas dans notre nœud et qu'il ne s'agit pas d'une feuille on fait un appel récursif vers l'enfant se trouvant à l'indice déterminée par whichIndex. Dans le cas où nous sommes dans une feuille on renvoie False.

## 2) L’algorithme d’insertion

Pour l’insertion d’une valeur dans un nœud on appele la fonction insertInNode.
Au départ, on parcourt l’arbre afin de situer correctement notre valeur. Pour cela, on utilise à nouveau la fonction whichIndex, qui à chaque fois donnera l’indice de l’enfant qui est adapté à la valeur. Puis on procède par récursivité la fonction sur l’enfant en question.

Une fois arrivé à un nœud feuille, on utilise la fonction insert avec la valeur en paramètre. La fonction va d’abord utiliser whichIndex afin de situer notre valeur, puis l’insérée dans notre nœud. Par la suite, on vérifie si le nombre de valeur maximum est atteint quand la condition est rempli on appele la méthode balance.

Au départ balance appel une méthode nommée splitNode.
Celle-ci va récupérer la valeur médiane du nœud, créer un nouveau nœud qui aura les même propriétés que le nœud actuel (si il est feuille ou non) et l’indice de la médiane.
Tant que la taille du tableau des valeurs est supérieur à l’indice de la médiane on rajoute des valeurs à l’indice de la médiane dans le nouveau nœud. Ensuite on vérifie si notre nœud est parent, dans le cas où il est tant que le nombre d’enfant est inférieur à l’indice de la médiane +1, on ajoute dans le nouveau nœud les enfants à l’indice de la médiane. On oublie pas de redéfinir dans les nœuds enfants le nouveau parent. A la fin, on retourne un dictionnaire contenant deux clés : la première qui est la médiane et la seconde le nouveau nœud.

De retour dans la méthode balance, on vérifie si notre nœud est la racine c’est à dire qu’il n’a pas de parent, dans ce cas on utilise la méthode insertRoot avec comme paramètre le dictionnaire.
Elle consiste à créer une nouvelle racine, l’ancienne racine est le nœud se trouvant dans le dictionnaire on comme nouveau parent la nouvelle racine et elle prend la valeur médiane en dans son values.
Dans le cas où on ne se trouve pas dans la racine on appele la fonction insertP avec aussi le dictionnaire en paramètre.
Au départ on situe dans le nœud parent où se trouve la médiane afin d’insérer le nouveau nœud en prenant en compte si l’indice vaut la taille de la liste des valeurs du nœud puis on utilise la méthode insert avec pour paramètre la valeur médiane. Ce qui amène le processus de récursivité est fait en sorte que l’arbre s’équilibre tout seul.

Une fois l’équilibrage fini et que l’on retourne au sein de la classe Arbreb on vérifie si notre racine actuelle possède maintenant un parent si c’est le cas ce parent est la nouvelle racine.

## 3**) L’algorithme de suppression**

Pour l’algorithme de suppresion on utilise la méthode deleteValueInTree qui prend en paramètre la valeur. Elle vérifie d’abord si la valeur est présente dans l’arbre si ce n’est pas le cas elle renvoie un message précisant que la valeur n’est pas présente. Sinon elle éxécute la fonction deleteValue.

La fonction deleteValue a pour but de se situer à l’endroit où se trouve la valeur. On utilise encore whichIndex, on appele récursivement la fonction si la valeur n’est pas dans le nœud sinon si on se trouve dans une feuille on fait appel à delete.

Delete supprime la valeur en paramètre à l’indice i donné par whichIndex.

Dans le cas où on se trouve dans un nœud on utilise deleteAsNode. La fonction retire d’abord la valeur du nœud puis prend l’indice avec whichIndex selon l’indice si il est différent de 0 ou non on utilise soit swapFromLeft/swapFromRight avec en paramètre la valeur que l’on souhaite supprimer.
La première va retourner la valeur maximale provenant du fils à gauche puis inverser la valeur. On parcourt l’arbre en passant par le nœud enfant le plus à droite tant qu’il n’est pas une feuille.
Le seconde va retourner la valeur minimale provenant du fils à droite puis inverser la valeur.
Le parcours est le même mais on prend le fils le plus à gauche.
Une fois l’inversement de valeur faite, on insére la valeur que l’on à récuperer.
Si notre nœud n’est pas une feuille est que l’enfant en est une alors on utilise delete sinon deleteValue. Enfin on rééquilibre le nœud avec balanceOnDel.

La fonction de rééquilibrage suite à une suppresion balanceOnDel détermine dans quel cas de rééquilibrage nous sommes qui sont les suivants :

- le premier est que le voisin de gauche est dans la capacité de donné une valeur dans ce cas là on utilise la méthode balancingFromLeft.

- le second est que le voisin de droite est dans la capacité de donné une valeur dans ce cas là on utilise la méthode balancingFromRight.

- Sinon on utilise la méthode merge.

La fonction balancingFromLeft consiste à prendre la valeur maximale du voisin de gauche qui va remplacer la valeur à l'indice actuelle qui elle s'ajoutera dans le nœud qui en a besoin.

La fonction balancingFromRight consiste à prendre la valeur minimale du voisin de droite qui va remplacer la valeur à l'indice actuelle qui elle s'ajoutera dans le nœud qui en a besoin.

La fonction merge s'occupe selon l'indice si il est différent de 0 ou non, va récuperer le nœud enfant à l'indice (+1 si il faut) donné en paramètre qui sera donné à la liste des valeurs de l'enfant se trouvant à gauche. Puis il parcourt les enfants et valeurs du nœud retirer afin de les ajouters dans l'autre nœud.

De cette manière chaque lorsque l'équilibrage d'un des nœuds est fini la fonction va remonter au nœud précédent qui va se rééquilibreé à son tour.