PROJET ARBRE-B

Par

Glenn YOUBI DEGNON

Table des matières

I. Présentation des « Arbre B »

I.1 Historique de l’Arbre B

Les arbres B ont été inventés en 1970 par Rudolf Bayer et Edward M.McCreight dans les laboratoires de recherche de Boeing. Le but était de pouvoir gérer les pages d'index de fichiers de données, en tenant compte du fait que le volume des index pouvait être si grand que seule une fraction des pages pouvait être chargée en mémoire vive.

En informatique, un **arbre B** (appelé aussi **B-arbre** par analogie au terme anglais « *B-tree* ») est une structure de données en arbre équilibré. Les arbres B sont principalement mis en œuvre dans les mécanismes de gestion de bases de données et de systèmes de fichiers. Ils stockent les données sous une forme triée et permettent une exécution des opérations d'insertion et de suppression en temps toujours logarithmique.
Le principe est de permettre aux nœuds parents de posséder plus de deux nœuds enfants : c'est une généralisation de l’arbre binaire de recherche. Ce principe minimise la taille de l'arbre et réduit le nombre d'opérations d'équilibrage. De plus un B-arbre grandit à partir de la racine, contrairement à un arbre binaire de recherche qui croît à partir des feuilles.

I.2 Signification de la lettre B

Le créateur de l'arbre B, Rudolf Bayer, n'a pas expliqué la définition de « B ». L'explication la plus courante est que le B correspond à l'expression anglaise "balanced" (en français: « équilibré"). Cependant, il pourrait aussi découler de « Bayer », du nom du créateur, ou de « Boeing », du nom de la firme pour laquelle le créateur travaillait (Boeing Scientific Research Labs).

I.3 Intérêt des arbres B

Les B-arbres apportent de solides avantages en termes de rapidité et d'efficacité par rapport à d'autres mises en œuvre lorsque la plupart des noeuds sont dans le stockage secondaire, comme un disque dur. En maximisant le nombre de nœuds enfants pour chaque nœud, la hauteur de l'arbre est réduite, l'opération d'équilibrage est nécessaire moins souvent et donc l'augmentation de l'efficacité. En général, ce nombre est réglé de telle sorte que chaque noeud occupe tout un groupe de secteurs: ainsi, étant donné que les opérations de bas niveau pour accéder au disque de cluster, il réduit au minimum le nombre d'accès à elle. Ils offrent d'excellentes performances par rapport aux opérations de recherche d’insertion et de suppression , car elles peuvent être faites avec la complexité logarithmique et par l'utilisation des procédures très simples. Sur eux, il est également possible d'effectuer un traitement séquentiel d'archivage primaire sans qu'il soit nécessaire de le soumettre à une réorganisation.

I.4 Différence entre l'arbre B et l'arbre B+

La principal différence est qu'un arbre B+ peut contenir des copies des clés déjà présentes dans l'arbre tandis qu'un arbre B contient des valeurs uniques. En effet, dans l' arbre B+ des copies des clés sont stockées dans les nœuds internes ; les clés et les enregistrements sont stockés dans des feuilles ; de plus, un nœud feuille peut inclure un pointeur vers le nœud feuille suivant pour accélérer l'accès séquentiel.II. Présentation du Projet

II.1 Le GIT

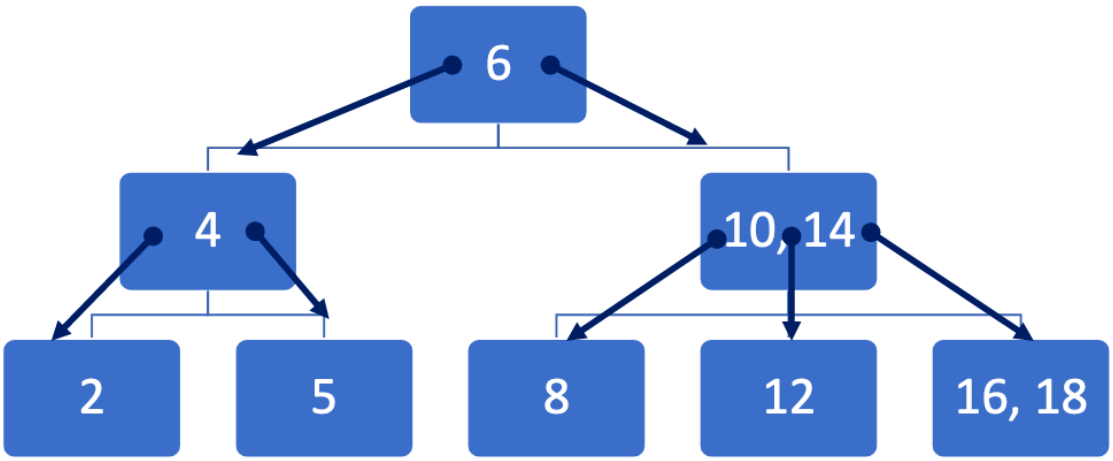
Le lien du git est : https://gitlab-etu.fil.univ-lille1.fr/youbidegnon/projet_s6_youbidegnonglenn
Sur le git je vous ai ajouté en tant que « developer ».

Le git contient les fichiers suivants :

À la racine, un README.md, un .gitignore, un Makefile et un dossier src. Le dossier src contient tout les scripts de code écrit.

II.2 Les Algorithmes

Les algorithmes de recherche et d'insertion s'exécutent correctement mais l'algorithme de suppression n'est pas opérationelle. Cela vous sera expliqué sous les rubriques de chaque algorithme.
Les exemples, que l'on effectuera dans la description des algorithmes, seront effectués sur l'arbre ci-dessous.



arbre. L = 2 et U = 3.

II.2.1 Algorithme de Recherche

```
def btree_search(self, value, node=None):
"""
Rechercher la valeur "value" dans l'arbre, en vérifiant à chaque étape si
la "value" est dans le noeud passer en paramètre de la fonction à la fonction rechercher

Entrées : value: Valeur à recherchez dans l'arbre .  node: noeud de départ de la recherche.
Sortie :  Booléen, Vrai si la valeur est dans l'arbre, Faux sinon.

>>> arbre.btree_search(31)
False
>>> arbre.btree_search(18)
True
"""
        si le node est None:
            node = self.root
        si node.search_in_node(value):
            Retourner Vrai
        si node est une feuille:
            Retourner Faux
        sinon si node.keys[0] > value:
            return self.btree_search(value, node.sons[0])
        sinon si value > node.keys[len(node.keys)- 1]:
            return self.btree_search(value, node.sons[len(node.sons) - 1])
        sinon:
            pour i compris entre 0 et len(node.keys) - 1:
                si node.keys[i] < value < node.keys[i + 1]:
                    Retourner self.btree_search(value, node.sons[i + 1])
```

Afin de rechercher une valeur dans l'arbre, on profite du fait que les valeurs présentes dans l'ordre soient rangées dans un ordre précis. La recherche s'effectue à partir de la racine.

Tout d'abord, on vérifie si la valeur recherchée est présente dans la racine. Si c'est le cas La fonction retourne vrai, si ce n'est pas le cas on vérifie que la racine est une feuille ou non. Si la racine est une feuille, la recherche s'arrête et la fonction retourne Faux.

Ensuite, on poursuit la recherche dans les fils du noeud courant(dans notre cas la racine). On distingue trois cas à partir de ce stade.
Le premier cas, la valeur recherchée est inférieur à la première clé du noeud courant dans ce cas on rappelle la fonction *btree_search* avec la même valeur recherchée mais avec pour noeud le fils à la position 0 dans la liste sons du noeud courant: *self.btree_search(value, node.sons[0])* .
Le deuxième cas, la valeur recherchée est supérieure à la dernière clé du noeud courant dans ce cas on rappelle la fonction *btree_search* avec la même valeur recherchée mais avec pour noeud le fils à la position *len(node.sons) - 1* dans la liste sons du noeud courant *self.btree_search(value, node.sons[len(node.sons) - 1])*. C'est-à-dire le dernier fils Du noeud.
Le troisième cas, la valeur recherchée n'est pas dans le noeud courant mais elle est comprise entre l'intervalle *[node.keys[0], node.keys[len(node.keys) - 1]*. Dans cas on parcourt les clés du noeud et si la valeur recherchée est comprise entre [node.keys[i],node.keys[i + 1]], on rappelle la fonction *btree_search* sur le fils à la position ***i+1*** dans la liste des en fils : *self.btree_search(value, node.sons[i + 1]) .*

La recherche d'une valeur dans un noeud s'effectue avec la recherche dichotomique avec qui s'effectue en O(log n) avec n le nombre de clés dans le noeud courant. La boucle « *pour i compris entre 0 et len(node.keys) - 1: »* parcours n-1 clés dans le pire cas donc une complexité en O(n-1).
De cela, on peut estimer que notre algorithme a une complexité O((n-1)+(log n)).

Recherchons la valeur 8 dans l'arbre d'exemple.
On vérifie si 8 est dans la racine |6|. 8 n'est pas dans la racine et 8 est supérieur à 6 donc on appelle récursive ment la fonction *btree_search* sur les fils à position len(node.sons) - 1.
Donc : *btree_search(8,node.sons[len(node.sons) - 1]) = > btree_search(8, |10,14|) .*

On vérifie si 8 est dans le noeud |10, 14|. 8 n'est pas dans le noeud et 8 est inférieur à 10 donc on appelle récursive ment la fonction *btree_search* sur les fils à position 0.
Donc : *btree_search(8,node.sons[0]) = > btree_search(8, |8|).*

On vérifie si 8 est dans le noeud |8|. 8 est pas dans le noeud la fonction retourne True et elle s'arrête.

II.2.2 Algorithme d' Insertion

```
def insert(self, value, node=None):
    """
    Insèrer la valeur "value" passer en paramètre dans l'arbre.
    :param value: Int, valeur à ajouter dans l'arbre.

    Entrées : value: Valeur à ajouter dans l'arbre  node: Noeud de départ pour l'insertion,
optionnel.
    Sortie :  Booléen, Vrai si la valeur a pu être insérer, Faux sinon.
    """
    si la valeur est dans l'arbre:
       affiche la valeur est déjà dans l'arbre
       Retourne False
    node_insert = self.get_node_to_insert(value, self.root)
    Insère la valeur dans node_insert
    Si le. Noeud n'est pas valide:
       Équilibrer le node_insert
       retire le node_insert de la liste leaf_nodes
       Retourne True
    sinon:
       Retourne True
```

Afin d'insérer une valeur dans l'arbre, on vérifie tout d'abord que la valeur n'est pas présente dans l'arbre. Si la valeur se trouve déjà dans l'arbre la fonction retourne False.

Si la valeur n'est pas présente dans l'arbre, nous récupérons d'abord le noeud dans lequel la valeur doit être insérer, avec la fonction *get_node_to_insert(value, self.root).* On insère la valeur « value » dans le noeud et on vérifie si le noeud est valide. Un noeud est considéré comme valide, après l'insertion d'une valeur, si son nombre de clés est inférieur au nombre de clés maximales acceptées par un noeud de l'arbre.

Dans le cas où le noeud est invalide, la suite de l'insertion est gérée par la fonction *balance* qui a pour rôle de rééquilibrer l'arbre.

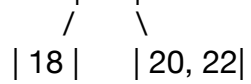
Pour équilibrer l'arbre, on « split » le noeud invalide. Pour ce faire on divise le noeud en 3 noeuds distincts dont l'un est le père des deux autres noeuds. Pour effectuer la division on récupère la clé situé à la médiane dans la liste des clés du noeud à « spliter ». Les valeurs qui précèdent la valeur médiane sont ajoutés comme clé au nouveau noeud fils gauche. Les valeurs qui succèdent la valeur médiane sont ajoutés comme clé au nouveau noeud fils droits. Ensuite on gère la parenté en précisant aux fils gauche et droit que leur parent est le 3ème noeud créé et en précisant au troisième noeud qu'il a pour fils le fils gauche et le fils droit.

Lorsque la fonction *split_node* termine son exécution, la fonction *balance* reprend son exécution. Elle effectue les dernières vérifications nécessaires.
Premièrement, elle vérifie si le noeud « spliter » avait un parent. Si c'est le cas, On ajoute la valeur du ***nouveau noeud parent,*** obtenu avec la méthode « split ». dans le noeud parent du noeud « spliter ».
On retire le noeud « spliter » de la liste des enfants du parent. On rajoute les fils de nouveaux fils gauche et fils droit à la liste des enfants du noeud parent. On précise aux noeud fils gauche et droit que leur parent est maintenant le noeud parent du noeud « spliter ».

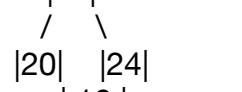
Et dernièrement, on vérifie si le noeud parent est valide, si il n'est pas on lui applique la fonction ***balance.***

Exemple d'exécution:

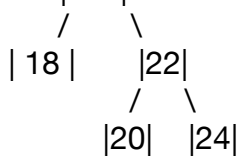
Soit l'arbre B, arbre2 avec L = 2 et U = 3, suivant avec pour racine : | 16 |.



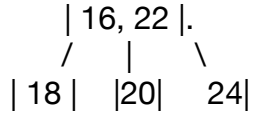
Ajoutons 24:
Arbre2.insert(18) => |20, 22, 24| donc on exécute ***balance*** car le noeud n'est plus valide.
balance (|20, 22, 24| ***)*** => première étape on salit le noeud.
split_node(|20, 22, 24| ***)*** **=>**



Et l'arbre devient :



À partir de la ***balance***, il insère 22 dans le noeud 16 et ajoute 20 et 24 comm fils du noeud | 16, 22 |.
Et l'arbre devient =>



Et l'insertion s'arrête.

Calcul de la complexité:
La fonction de recherche d'une valeur dans l'arbre a une complexité de O((n-1)*(log n)), avec n le nombres de clés dans le noeud courant. La fonction ***split_node*** a une complexité en O(m*n) car elle parcoure toutes les clés d'un noeud et tous les fils d'un noeud. Avec m le nombre de fils du noeud courant et m > n pour tout noeud de l'arbre, sauf pour la racine au tout début de l'insertion. De cela, on peut déduire que notre algorithme d'insertion a une complexité de O(m*n).

II.2.3 Algorithme de Suppression

L' algorithme de suppression avait été pensée afin de gérer 3 cas du suppressions distincts :
1. Suppression d'une valeur présente dans la racine.
2. Suppression d'une valeur présente dans un noeud interne.
3. Suppression d'une valeur présente dans une feuille.

Cas 1: Suppression d'une valeur présente dans la racine

1. La suppression de la clé ne viole pas la propriété du nombre minimum de clés qu'un nœud doit détenir.

2. La suppression de la clé viole la propriété du nombre minimum de clés qu'un nœud doit détenir. Dans ce cas, nous empruntons une clé au nœud voisin immédiat, dans l'ordre de gauche à droite.
Tout d'abord, nous visitons le nœud voisin immédiat de gauche. Si le nœud frère gauche possède plus qu'un nombre minimum de clés, alors on emprunte une clé à ce nœud. Sinon, vérifiez si vous pouvez emprunter une clé au nœud voisin immédiat de droite.

Cas 2: Suppression d'une valeur présente dans un noeud interne.

1. le nœud interne, qui est supprimé, est remplacé par un prédécesseur dans l 'ordre si l'enfant gauche a plus que le nombre minimum de clés.

2. Le noeud interne, qui est supprimé, est remplacé par un successeur dans l'ordre si l'enfant de droite a plus que le nombre minimum de clés.

3. Si l'un des enfants a exactement un nombre minimum de clés, alors, on fusionne les enfants de gauche et de droite.

```
def delete(self,value):
    """
    Supprime une valeur de l'arbre rééquilibre selon les necessistés
    afin de respecter la structure d'un arbre B.

    :param value: Int, valeur à supprimer de l'arbre.
    :return: Bool, True si la valeur a pu être retirer, False sinon.
    """
    Si l'arbre ne contient la valeur value:
       print("CETTE VALEUR N'EST PAS PRESENTE DANS L'ARBRE.")
       Retourne False
    node_delete = self.get_node_to_delete(self,value,arbre.root)
    type_of_deletion = self.get_type_of_deletion(node_delete)
    Si type_of_deletion est égale 0:
       self.delete_in_root(value)
     Snon si i type_of_deletion est égale 0:
        self.delete_in_leaf(value)
     sinon:
        self.delete_in_internal_node(value)
```

II.2.4 Algorithme d'Affichage

Notre arbre sera affiché par niveau. De O à n, avec n la hauteur de l'arbre et 0 le noeud de la racine. Notre arbre d'exemple serait affiché comme suit:

```
Level  0    |6|
Level  1    |4|   fils de -> [6]
Level  2    |2|   fils de -> [4]
Level  2    |5|   fils de -> [4]
Level  1    |10,14|   fils de -> [6]
Level  2    |8|   fils de -> [10, 14]
```

```
Level  2    |12|   fils de -> [10, 14]
Level  2    |16,18|   fils de -> [10, 14]
```

```
def display_tree(self, node, level=0):
    """
    Affiche l'arbre par noeud et par niveau. Chaque noeud est affiché en précisant à
    sa suite son parent si il en a. Sous la forme (exemple):
    "Level 0  |Noeud|  fils de -> [Noeud parent]".
    :param node: Noeud à partir du quel l'affichage sera réalisée.
    :param level: Niveau du noeud dans l'arbre.
    :return: Affiche l'arbre sur la sortie standard.
    """
    Affiche "Level ", level, end="    |"
    pour i entre 0 et len(node.keys) :
       si i est égale len(node.keys) - 1:
          si le parent de node n'est pas None:
             Affiche node.keys[i], end="|   fils de -> " + str(node.parent.keys) + ""
          sinon:
             Affiche node.keys[i], end="|"
       sinon:
          print(node.keys[i], end=",")
    Affiche un saut de ligne
    level += 1
    Si len(node.sons) est supérieur à 0:
       pour i dans la liste node.sons:
          self.display_tree(i, level)
```

II.2 Comment exécuter le code?

À la racine du projet, il y'a un makefile pour faciliter l'exécution du programme. Toutes les commandes suivantes sont à exécutées dans un terminal et à la racine du projet. C'est-à-dire le dossier projet_s6_youbidegnonglenn.

Tests unitaires:

Pour exécuter les tests unitaires concernant les noeud entrez la commande : make node_test .

Pour exécuter les tests unitaires concernant l'arbre B entrez la commande : make btree_test .

Tests de monsieur Djeraba:

Pour exécuter le premier jeu de tests entrez la commande : make test1 .
Pour exécuter le second jeu de tests entrez la commande : make test2 .

La suppression ne marchant pas, les tests n'effectuent pas de suppresion.

Pour utiliser les fonctionnalités disponobles dans l’arbre un script est mis à votre disposition.

Pour le lancer entrez la commande : make user .
Et suivez les instructions.

III. Configuration systèmes requises

Vous devriez avoir sur **python3** votre machine.
Afin d’explorer le code, tout éditeur de texte ferai l’affaire, mais nous vous recommandons PyCharm ou Visual Studio Code.