Plancke Aurélien, Plé Lucas

Arbre-B - Projet Semestre 6

Sommaire

1. Introduction

a. Historique de l’arbre-B

Rudolf Bayer et Edward M.McCreight ont publié un article décrivant le fonctionnement des arbres-B en 1970. À cette époque, ils travaillaient tous les deux pour Boeing. Ils souhaitaient gérer les pages d'index de fichiers de données. Il y avait déjà à l'époque énormément d'index et il était impossible de les charger tous complètement en mémoire sur un ordinateur de l'époque. Il était nécessaire de trouver une solution permettant d’accéder plus simplement à ces données.

b. Signification du B

Les deux inventeurs des arbres-B ne se sont jamais prononcés sur la signification du "B". La signification de ce dernier est donc sujet à interprétation. Certains pensent que cela fait référence à "Balanced", "Bayer" ou même "Boeing".

c. Intérêt des arbres-B

Le but des arbres-b est de permettre le stockage des données en les gardant triées en permanence et que l'on puisse réaliser des opérations de recherches, d'insertions et de suppressions toujours en temps logarithmique. Ils sont utilisés dans des systèmes de base de données puisqu'ils permettent de garantir une exécution rapide et une persistance des données sous forme triée.

d. Différence entre arbre-B et arbre-B+

Les arbres-B+ sont différents des arbres-B. Contrairement à ces derniers, on ne peut pas stocker de référence vers les nœuds fils dans un arbre-B+. Toutes les données sont stockées dans les feuilles sous forme de pointeurs. Les feuilles sont également liées entre elles. On peut ainsi parcourir l'arbre de gauche à droite au lieu de le parcourir en profondeur. Une feuille d'un arbre-B+ va donc contenir des valeurs triées par ordre croissant ainsi qu'un pointeur vers la feuille suivante qui comporte les prochaines valeurs. Cette liste chaînée se termine en général par un élément vide pour signifier la fin de l'arbre.

Les avantages d'un arbre-B+ sont qu'il est possible de stocker plus de données dans une plage de mémoire et qu'il est plus simple d'accéder aux données, mais aussi qu’on a seulement besoin d'un pointeur sur la première feuille pour parcourir l'arbre au lieu de réaliser un parcours complet en profondeur pour un arbre-B classique.

Cependant, avec un arbre-B simple, on peut placer les données souvent accédées proches de la racine : il sera donc plus rapide d'y accéder que dans un arbre-B+.

2. Documentation

Le projet est enregistré dans le dépôt git suivant : https://gitlab-etu.fil.univ-lille1.fr/plancke/projet_arbre_s6 . Des instructions se trouvent dans un fichier “README” pour récupérer le projet et exécuter les batteries de tests.

a. Classes

Une des contraintes de ce projet était d’utiliser une technologie objet. Nous avons décidé d’utiliser Python comme langage, car nous connaissons déjà bien Java et que nous n’avons pas eu l’occasion de beaucoup le pratiquer. Pour réaliser ce projet, nous avons mis en place une architecture simple. Comme nous le voyons sur l’UML ci-dessous, nous n’avons besoin que de deux classes.

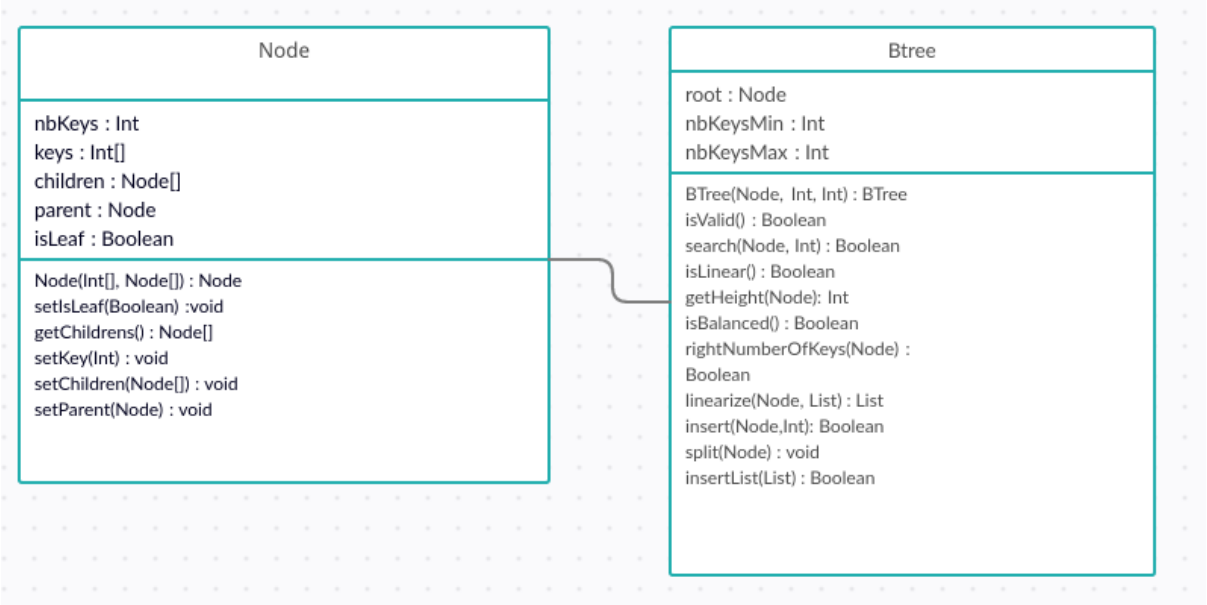

Figure 1. UML du projet

Tout d’abord nous avons une classe Node (Nœud en Français) qui s'occupe de gérer toutes les propriétés des nœuds, nous retrouvons dans cet objet, son nœud parent, son statut (feuille ou non), ses nœuds enfants et bien sûr, ses clefs.

En ce qui concerne la classe BTree (Arbre-B), celle-ci est décomposée en un nœud qui est sa racine (codé en un objet de la classe Node), son nombre de clefs minimal et maximal. Ensuite, cet objet comporte des méthodes permettant la vérification de la validité de l’arbre. En effet il y a plusieurs méthodes telles que isLinear, isBalanced, rightNumberOfKeys, qui sont rassemblées dans la méthode isValid et qui servent à vérifier que l’arbre est correct.

Pour qu’un arbre-B soit valide, il faut que toutes les feuilles soient à la même hauteur et que les nombres de clefs minimal et maximal par nœud soient bien respectés. Bien sûr, il y a également le paramètre de la linéarité : les valeurs lorsqu’on aplatit l’arbre doivent être triés dans l’ordre croissant.

Finalement, nous avons les méthodes qui concernent les interactions avec l'arbre, nous retrouvons "search", qui permet de vérifier si une clef est présente dans l'arbre et nous avons "insert", qui insère une clef au bon endroit dans l'arbre. Ces fonctions sont aidées d'autres méthodes internes, comme par exemple "split", qui va s'occuper de vérifier si nous ne dépassons pas le nombre de clefs maximal pour un nœud et, si besoin, va remanier l'arbre pour que celui-ci reste valide après une insertion.

b. État actuel du projet

En l'état, notre implémentation de l'arbre-B permet de créer l'arbre, d'insérer des valeurs et de les rechercher. Nous n'avons malheureusement pas terminé la suppression d'une donnée. Les deux batteries de tests demandées sont présentes et nous avons également développé en utilisant la méthode TDD. Nous avons donc, en plus des batteries de tests, des tests unitaires que nous avons définis en même temps que notre implémentation et que l'on peut également lancer. La documentation des méthodes et des classes a été rédigée en utilisant le format docstring de Python.

c. Algorithmes

Nous allons donner dans cette partie le pseudo-code des algorithmes du projet.
On suppose que l'on dispose d'une fonction "Eclater" pour l'insertion permettant d'effectuer l'éclatement d'un nœud lorsque c'est nécessaire ainsi que d'une fonction "MettreÀJour" pour la suppression permettant de reconstruire un nœud après la suppression d'une clef si les conditions de remplissage du nœud ne sont plus réunies.

Algorithme de recherche

Le principe de cet algorithme est de parcourir l'arbre en profondeur via un appel récursif jusqu'à trouver la valeur passée en paramètre.

```
Recherche(nœud, valeur) :
Entrées : Un noeud node, une valeur à chercher
Sortie : Vrai si la recherche est concluante, faux sinon
Si valeur dans node.clefs()
        alors retourner Vrai
Si node.estUneFeuille
        alors retourner Faux
Si value < node.clefs[0]
        alors arbre.Recherche(enfantLePlusAGauche, valeur)
Si value > node.clef[longueur(node.clef)]
        alors arbre.Recherche(enfantLePlusADroite, valeur)
Sinon
        faire pour i allant de 0 node.clefs().taille()
                intervalle ← [nœud.clefs[i], nœud.clefs[i+1]]
                si valeur dans intervalle
                        alors arbre.Recherche(enfantEntreCesDeuxClef, valeur)
```

Algorithme d'insertion

Le principe de cet algorithme est de réaliser un parcours en profondeur de l'arbre pour trouver le nœud dans lequel il faut insérer la valeur. Si, lorsqu'on insère la valeur dans le nœud, on dépasse le nombre maximum de clefs dans le nœud autorisé, il faut faire éclater le nœud pour respecter cette règle et garantir que l'arbre sera toujours valide après l'insertion.

```
Insérer(nœud, valeur) ;
Entrées : Un nœud de départ, une valeur à insérer
Sortie : Vrai si la valeur a pu être insérée, Faux sinon
Si Recherche(arbre.racine, valeur) :
        alors retourner Faux
Si nœud.estUneFeuille :
        alors indexOùInserer ← 0
        tant que indexOùInserer < nœud.clefs.taille() et
nœud.clefs[indexOùInserer] < indexOùInserer :
                faire indexOùInserer ← indexOùInserer + 1
        nœud.clefs.insérer(indexOùInserer, valeur)
        Si node.clefs.taille() <= arbre.nombreDeClefsMax :
                alors retourner Vrai
        arbre.eclater(nœud)
        retourner Vrai
sinon si valuer < nœud.clefs[0] :
        alors retourner arbre.Insérer(nœuds.enfants[nœud.enfants.taille()],
valeur)
sinon :
        faire pour i allant de 1 à nœud.clefs.taille() :
                si valeur < nœud.clefs[i] :
                        retourner arbre.Insérer(nœuds.enfants[i], valeur)
retourner Vrai
```

Algorithme de suppression

Le principe de cet algorithme est le même que pour l'insertion. On parcourt l'arbre en profondeur jusqu'à trouver le nœud où se trouve la valeur à supprimer. Si on ne respecte plus la règle du nombre de clefs minimum après cette suppression, il faut mettre à jour l'arbre en remontant certaines clefs dans le nœud en s'assurant de garder un arbre correct.

```
Supprimer(nœud, valeur):
Entrées : Un nœud, une valeur à supprimer
Sortie : Vrai si la suppression à été effectuée, faux sinon
Si Recherche(racine,valeur) == Faux
        alors retourner Faux
nbClefs ← nœud.clefs().taille() - 1
si nœud.estUneFeuille :
        alors nœud.clefs().supprime(valeur)
        si nbClefs > nœud.nombreDeClefsMin
                alors retourner Vrai
        sinon
                faire MettreAJour(nœud.parent, noeud)
        si valeur dans nœud.clefs
                alors nœud.clefs().supprimer(valeur)
                si nbClef > nœud.nombreDeClefsMin
                        alors retourner Vrai
                sinon
                        faire arbre.MettreAJour(node.parent, noeud)
        sinon si valeur < node.keys[0]
                alors retourner arbre.Supprimer(premierEnfant, valeur)
        sinon si valeur > node.keys[longueur(node.keys)]
                alors retourner arbre.Supprimer(dernierEnfant, valeur)
        sinon
                faire pour i allant de 0 à nœud.clefs().taille()
                        si valeurs < nœud.clefs[i]
                                alors retourner arbre.Supprimer(nœud.enfant[i],
valeur)
```

Bibliographie

Voici les ressources autres que le cours accessible sur moodle que nous avons utilisé :

https://en.wikipedia.org/wiki/B-tree
https://www.cs.usfca.edu/~galles/visualization/BTree.html
https://fr.acervolima.com/difference-entre-l-arbre-b-et-l-arbre-b/
https://www.geeksforgeeks.org/introduction-of-b-tree-2/
https://www.geeksforgeeks.org/insert-operation-in-b-tree/
https://www.geeksforgeeks.org/delete-operation-in-b-tree/