Projet B-Tree

Présenté par :

DERMEL Madjid

Département Informatique
Université de Lille

Année universitaire 2023-2024

Table des matières

1 Définition globale

1.1 Cahier de charges

Ce projet vise à développer et à mettre en œuvre une structure d'arbre B, ainsi que ses méthodes associées, pour le stockage efficace de clés uniques, en se concentrant sur des nombres pour simplifier.

1.2 Définition d'un B-Tree

Un arbre-B est une structure de données en arbre auto-équilibrante qui conserve des données triées et permet des recherches, un accès séquentiel, des insertions et des suppressions en temps logarithmique. Parfait pour les gros volumes de données, comme ceux des bases de données ou des systèmes de fichiers, il fonctionne très bien même quand il doit gérer beaucoup d'informations en même temps.

1.3 Caractéristique d'un B-Tree

Un arbre-B est une structure de données conçue pour le stockage et la manipulation efficaces de grandes quantités d'informations. Ses caractéristiques principales incluent son auto-équilibrage, qui assure une distribution uniforme des données à travers l'arbre, permettant ainsi des opérations de recherche, d'insertion et de suppression en temps logarithmique, ce qui signifie que même avec un grand nombre de données, ces opérations restent rapides. Contrairement aux arbres de recherche binaires traditionnels, l'arbre-B est optimisé pour les systèmes qui manipulent de grands blocs de données, tels que les bases de données et les systèmes de fichiers, car il minimise les accès disques nécessaires pour effectuer des opérations. Cela est dû à sa capacité à stocker plusieurs clés dans un seul nœud, permettant ainsi à un plus grand nombre d'informations d'être traitées en moins de lectures et d'écritures. En somme, l'arbre-B est essentiel pour gérer efficacement les données dans des environnements où la performance et la rapidité d'accès sont cruciales.

1.4 Propriété d'un B-Tree

Les arbres-B ont des caractéristiques spécifiques qui garantissent leur efficacité : toutes leurs feuilles sont à la même hauteur, ce qui assure un équilibre parfait. Chaque nœud contient entre k/2 et k clés, assurant ainsi un taux de remplissage minimum de 50% et moyen de 75%, optimisant l'utilisation de l'espace. Pour les nœuds internes (pas des feuilles), les clés sont soigneusement organisées pour maintenir l'ordre des données : chaque clé dans un nœud intermédiaire sert de "pointeur" divisant les clés de ses sous-arbres, garantissant ainsi que toutes les clés à gauche d'un pointeur sont plus petites et toutes à droite sont plus grandes. Cette structure ordonnée permet des recherches rapides, des insertions et des suppressions.

2 Conception

2.1 Architecture du logiciel

Notre projet implémente un arbre-B, structuré autour de nœuds interconnectés, avec des fichiers dédiés à la définition des nœuds, la logique de l'arbre et les tests d'application :

- **Node.py** : Définit les éléments de base de l'arbre, c'est-à-dire les nœuds. Chaque nœud peut contenir plusieurs clés et a des liens vers ses nœuds enfants, permettant une structure hiérarchique. Ce fichier est essentiel pour construire l'arbre et déterminer comment les données sont organisées et stockées à chaque niveau.
- **Btree.py** : Implémente la structure et les opérations spécifiques de l'arbre-B, y compris comment les nœuds sont équilibrés, la manière dont les données sont insérées, recherchées, et supprimées, tout en respectant les propriétés uniques de l'arbre-B, comme le maintien d'une hauteur équilibrée et des taux de remplissage des nœuds.
- **Main.py** : Sert de zone de test pour l'arbre-B, où les fonctionnalités de l'arbre sont mises en œuvre, comme l'insertion de données, la recherche et la visualisation de la structure de l'arbre. Cela permet de voir comment l'arbre réagit à différentes opérations et de vérifier que les propriétés de l'arbre-B sont maintenues après diverses manipulations.

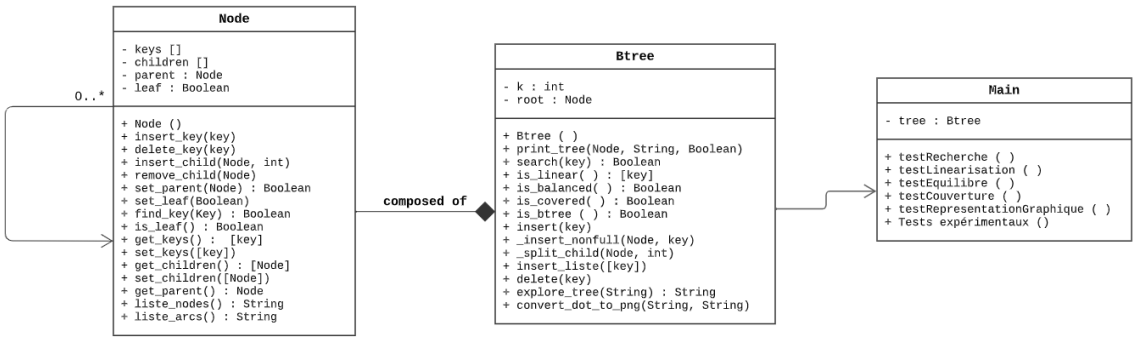

FIGURE 1 – UML de B-Tree

2.2 Algorithmes

Les algorithmes implémentés incluent la recherche, l'insertion, et des vérifications de propriétés spécifiques des arbres B, avec une attention particulière à leur complexité.

2.2.1 search(self, key)

Complexité en temps : $O(\log_k(n))$
Explication :
À chaque étape, la méthode élimine environ la moitié des nœuds restants en choisissant le bon sous-arbre. La hauteur d'un B-Tree est $O(\log_k n)$, où n est le nombre total de clés et k est le facteur. À chaque niveau de l'arbre, un nombre constant d'opérations est effectué (comparaisons et choix du sous-arbre suivant).

Par conséquent, la complexité totale est proportionnelle à la hauteur de l'arbre.

2.2.2 is_linear(self)

Complexité en temps : $O(log(n))$
Explication :
Cette méthode parcourt chaque nœud de l'arbre pour construire une liste linéaire des clés. Chaque clé dans l'arbre est visitée exactement une fois. Puisque le nombre total de clés dans l'arbre est n, la complexité en temps est linéaire par rapport au nombre de clés.

2.2.3 is_balanced(self)

Complexité en temps : $O(log(n))$
Explication :
La méthode vérifie si chaque sous-arbre de l'arbre B est équilibré. Pour ce faire, elle parcourt tous les nœuds de l'arbre. Bien que la hauteur de l'arbre soit $O(\log_k n)$, chaque nœud (et donc chaque clé) doit être visité pour vérifier si le sous-arbre est équilibré.
La complexité est donc linéaire par rapport au nombre total de clés dans l'arbre.

2.2.4 is_covred(self)

Complexité en temps : $O(log(n))$
Explication :
Cette méthode vérifie si chaque nœud respecte les propriétés des arbres B en termes de nombre de clés et d'enfants.
La vérification nécessite un parcours de tous les nœuds pour examiner leurs propriétés. Comme chaque nœud est visité une fois, la complexité est linéaire par rapport au nombre total de nœuds dans l'arbre, soit $O(n)$.

2.2.5 is_btree(self)

Complexité en temps : $O(log(n))$
Explication :
Cette méthode vérifie si l'arbre actuel satisfait à toutes les conditions d'un arbre B valide. Pour valider un arbre B, il est nécessaire de parcourir tous les nœuds et de vérifier ces propriétés.

2.2.6 insert(self)

Complexité en temps : $O(\log_k(n))$
Explication :
Si le nœud a de l'espace, l'insertion est immédiate ; sinon, une scission est nécessaire, mais reste limitée en nombre grâce à la hauteur logarithmique de l'arbre.
À chaque étape, un nombre fixe d'opérations assure l'insertion et la gestion des scissions.

2.2.7 Interface graphique

On a implémenté une interface graphique, générée par les méthodes "explore_tree" et "convert_dot_to_png" facilitent la visualisation d'un arbre B en créant d'abord une représentation textuelle de sa structure dans un fichier .dot et en convertissant ensuite ce

fichier en une image .png à l'aide de Graphviz. La première méthode construit une description complète de l'arbre, incluant les nœuds et leurs relations, et l'enregistre dans un fichier spécifié par l'utilisateur. La seconde méthode utilise cette description pour générer une représentation graphique de l'arbre, rendant la structure et les connexions entre les nœuds facilement visibles et compréhensibles. Ensemble, elles offrent une interface graphique intuitive pour examiner et comprendre les arbres B, rendant les opérations de débogage et d'analyse plus accessibles.

2.3 Tests

Dans le cadre du développement de notre projet arbre-B, l'accent a été mis sur l'importance cruciale des tests pour assurer la fiabilité et l'efficacité du logiciel. Nous avons adopté une approche rigoureuse en intégrant des tests à chaque étape du processus de développement. Initialement, nous avons commencé par des tests unitaires, conçus avant même l'implémentation des méthodes. Cette approche prédictive nous a permis de définir les comportements attendus pour chaque fonctionnalité, garantissant ainsi que le code développé répond précisément aux besoins spécifiés. Pour ce faire, nous avons utilisé l'outil doctest de Python, qui facilite la rédaction et l'exécution des tests directement dans la documentation du code. Parallèlement, notre stratégie de test comprenait des tests d'intégration, menés à une échelle plus large, telle que l'insertion de 10 000 éléments dans l'arbre. Ces tests avaient pour but de vérifier la cohérence et la stabilité de l'ensemble du système sous des conditions d'utilisation intensives. Après chaque opération majeure, nous procédions à une vérification rigoureuse pour s'assurer que l'arbre restait conforme aux propriétés fondamentales d'un arbre-B. Cette vérification était réalisée grâce à la fonction is_btree() qui vérifie chacune des propriétés d'un B-Tree, assurant ainsi que notre structure de données conservait ses caractéristiques essentielles même sous forte sollicitation. Grâce à cette démarche méthodique, notre projet arbre-B combine robustesse et performance, répondant aux standards les plus exigeants en matière de développement logiciel.

```python
def test_experimentations_1(self):
    print("-----------------------------------Test 6-----------------------------------")
    tree = Btree(Node(),2)
    tree.insert_liste([2,4,5,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,7,9,11,13])
    list = tree.is_linear()
    assert(len(list)==23)
    assert(list==sorted(list))
    assert(tree.is_btree())
    assert(tree.is_balanced())
    assert(tree.is_covered())
    print("test_experimentations_1 passé avec succès")

def test_experimentations_2(self):
    print("-----------------------------------Test 7-----------------------------------")
    tree = Btree(Node(),3)
    for i in range(10000):
        tree.insert(i)
    list = tree.is_linear()
    assert(len(list)==10000)
    assert(list==sorted(list))
    assert(tree.is_btree())
    assert(tree.is_balanced())
    assert(tree.is_covered())
    print("test_experimentations_2 passé avec succès")
```

FIGURE 2 – tests expérimentaux

3 Problèmes rencontrés

Au cours du développement, nous avons rencontré plusieurs défis, notamment :

- L'implémentation de l'insertion et la suppression, nécessitant une gestion précise des cas d'équilibrage de l'arbre.
- La gestion des cas limite lors de la suppression, en particulier dans les arbres très déséquilibrés.
- Optimisation des performances pour les opérations de recherche.

4 Points à améliorer

Pour améliorer le projet à l'avenir, plusieurs axes sont envisagés :

- Prendre en charge de types de données supplémentaires au-delà des nombres, comme les chaînes de caractères ou les objets complexes.
- Implémenter une interface plus flexible pour les méthodes de comparaison.
- Optimiser les algorithmes pour réduire la complexité des opérations d'insertion et de suppression, surtout dans les scénarios de charge élevée.
- Renforcer la robustesse des tests, notamment en ajoutant des tests de performance et de charge, pour garantir la fiabilité du système dans des conditions d'utilisation variées.