BABU Bastien
L2 MIDL
22302768

Compte Rendu TP Algo 3
Séquence 6 : Arbres binaires de recherche équilibrés - Arbres Rouge-Noir.

Ce TP porte sur l'implémentation des arbres binaires de recherches équilibrées, en particulier les arbres rouge-noir. Il s'agit d'une structure de données permettant l'optimisation des opérations du type dictionnaire (recherche, insertion, suppression). Ces arbres se distinguent par leurs propriétés garantissant un équilibrage « automatique », tout en assurant une complexité en **O(log(n))**. Les propriétés des arbres rouge noir sont les suivantes :

I. Les feuilles (NIL), sont noires
II. Les fils d'un nœud rouge sont noirs
III. Le nombre de nœuds noirs le long d'une branche issue de la racine est indépendant de la branche. Toutes les branches contiennent donc le même nombre de nœuds noirs.

L'objectif de ce TP est de comprendre et de mettre en place le fonctionnement interne des arbres rouge-noir, tout en étudiant ses performances à travers des tests pratiques. Nous explorerons également la robustesse des algorithmes implémentées, ainsi que les capacités des algorithmes mis en place à maintenir un équilibrage efficace tout en respectant les invariants et propriétés des arbres rouge-noir.

Dans un premier temps, nous ferons la description des choix d'implémentations dans chaque exercice, puis nous effectuerons la description du comportement des programmes sur les jeux de tests fournis, enfin nous détaillerons une analyse personnelle du travail notamment avec en y mentionnant les problèmes rencontrés et les solutions mises en place.

1) Description des choix d'implémentation

- **Exercice 1 : Coloration de l'arbre**

Dans cet exercice, nous avons étendu la structure de données de l'arbre pour gérer la couleur (rouge ou noir) des nœuds. Le but était d'adapter la représentation des nœuds pour les visualiser correctement en respectant les spécificités des arbres rouge-noir.

Pour ce faire, nous avons introduit un type énuméré *typedef enum {red, black} NodeColor;* pour représenter la couleur des nœuds. Ce type a été déclaré dans le fichier *bstree.c* afin d'éviter d'exposer des détails d'implémentation dans l'interface publique (*bstree.h*), ce qui permet donc de simplifier cette interface. Par ailleurs, afin d'intégrer la possibilité d'ajouter une couleur a un nœud, on a ajouté le champ *NodeColor color;* à la structure de l'arbre *struct _bstree*.

Dans la fonction de construction d'un nœud *bstree_cons(...)* a été modifiée pour initialiser systématiquement la couleur des nouveaux nœuds à red. Cela respecte les règles des arbres rouge-noir, où tout nouveau nœud inséré commence en rouge.

Enfin, la fonction *void bstree_node_to_dot(const BinarySearchTree* t, void* stream);*, a été implémentée dans *bstree.c*, encore une fois afin de laisser exposer dans le fichier publique (*bstree.h*) uniquement ce qui est important pour l'utilisateur. Cette fonction permet de générer un fichier *.dot* pour représenter l'arbre à l'aide de l'outil *graphviz* tout en respectant la couleur de chaque nœud de cet arbre. Il a juste

été le cas d'un ajout d'une condition : *t->color == red ? "style=filled, fillcolor=red" : "style=filled, fillcolor=grey");* , afin de colorer les nœuds par rapport à la fonction : *void node_to_dot(const BinarySearchTree *t, void *stream)* , implémenter dans le *main.c*.

- **Exercice 2 : Rotation de l'arbre**

Dans cet exercice, nous avons implémenter deux fonctions de rotations *leftrotate* et *rightrotate* qui permet de faire une rotation à gauche (respectivement à droite) autour d'un nœud *x* (respectivement *y*) donné en paramètre. Les opérations de rotations permettent de rééquilibrer la structure de l'arbre en modifiant localement la disposition des nœuds. Dans le cadre des arbres rouge-noir, ces opérations sont essentielles car il s'agit d'une structure équilibrée.

Ces deux fonctions, sont définies dans le fichier *bstree.c* pour rester privées et éviter tout usage non contrôlé. Par conséquent, cela assure que seules les opérations internes utilisent ces fonctions.

En ce qui concerne la complexité, ces deux fonctions effectuent uniquement des affectations (mise à jour de pointeurs en l'occurrence). De ce fait, le complexité temporelle est en un temps constant **O(1)** et la complexité spatiale est aussi en **O(1)** car aucune allocation dynamique n'est réalisée.

- **Exercice 3 : Correction de l'arbre après insertion**

L'objectif de cet exercice était de garantir que les propriétés/invariants d'un arbre rouge-noir (équilibre, coloration) soient maintenus après l'insertion d'un nœud.

Premièrement, nous avons implémenté deux fonctions auxiliaires : *grandparent* qui permet de renvoyer le grand-père d'un nœud (utile pour naviguer dans l'arbre) et *uncle* qui permet de renvoyer l'oncle d'un nœud (utile pour l'analyse des cas de correction après insertion). De plus, il a été modifié, au préalable, la fonction *void bstree_add (ptrBinarySearchTree *t, int v)* , afin qu'elle puisse appeler les opérateurs de correction après insertion.

Pour l'implémentation de ces opérateurs de correction après insertion, nous avons commencé par la fonction principale *fixredblack_insert(BinarySearchTree* x),* qui vérifie les propriétés après l'ajout d'un nœud rouge *x*. Afin d'effectuer cette vérification elle appelle des sous-fonctions comme *BinarySearchTree* fixredblack_insert_case1(BinarySearchTree*x),BinarySearchTree*fixredblack_insert_case2(BinarySearchTree*x),BinarySearchTree*fixredblack_insert_case2_left(BinarySearchTree*x),BinarySearchTree*fixredblack_insert_case2_right(BinarySearchTree* x)*, où chacune de ces fonctions gèrent un cas précis. Globalement, voici comment cela se passe, une fois le nœud ajouté, des violations des propriétés de l'arbre rouge-noir peuvent apparaître, notamment si le nœud ajouté et son parent sont tous deux rouges. Ces cas de violation sont corrigés via des recolorations et/ou des rotations, selon une analyse locale autour du nœud problématique. Les recolorations sont appliquées lorsque l'oncle du nœud ajouté est rouge, ce qui permet de rétablir les hauteurs noires sans modifier la structure de l'arbre. Les rotations, quant à elles, elles interviennent lorsque l'oncle est noir. Ces rotations, ajustent la structure pour équilibrer l'arbre tout en garantissant le respect des propriétés/invariants des arbres rouge-noir.

Enfin, en ce qui concerne la complexité, les recolorations nécessitent allant jusqu'à la racine dans le pire des cas, il s'agit donc d'une complexité en un temps logarithmique **O(log(n))**. Les rotations, comme vu précédemment, est uniquement constituer d'affectations, il s'agit donc d'une complexité en temps constant **O(1)**. En somme, l'insertion et la correction d'un nœud dans un arbre rouge-noir s'exécutent en O(log(n)), ce qui en fait une structure extrêmement efficace pour un accès rapide et un équilibrage dynamique.

- **Exercice 4 : Recherche dans un arbre rouge-noir**

  La recherche d'une valeur dans un arbre Rouge-Noir est indépendante de l'algorithme d'équilibrage et correspond exactement au même algorithme que pour la recherche dans un arbre binaire de recherche.

- **Exercice 5 : Correction de l'arbre après suppression**

  L'objectif de cet exercice était de garantir que les propriétés/invariants d'un arbre rouge-noir (équilibre, coloration) soient maintenus après la suppression d'un nœud.

  Tout d'abord, nous avons modifié la fonction de suppression pour qu'elle appelle les opérateurs de correction après la suppression d'un nœud, garantissant ainsi que l'arbre respecte toujours les propriétés d'un arbre rouge-noir. Pour cela, nous avons implémenté la fonction principale *fixredblack_remove(BinarySearchTree* p, BinarySearchTree* x)*, qui prend en charge les différents cas de restauration des propriétés après suppression.

  Les principales étapes de la correction après suppression incluent plusieurs sous-fonctions qui gèrent des cas spécifiques. Nous avons notamment implémenté les fonctions *fixredblack_remove_case1* et *fixredblack_remove_case2*, chacune traitant des situations différentes en fonction de la couleur du frère du nœud à corriger (principe de « double-noir »). Ces fonctions appliquent des rotations et des recolorations afin de rétablir les propriétés de l'arbre. En ce qui concerne le cas 1, il intervient si le frère du nœud à corriger est noir. En fonction de la couleur de ses fils, des rotations et des recolorations son effectuées pour rétablir l'équilibre de l'arbre. En ce qui concerne le cas 2, il intervient si le frère du nœud est rouge. Ce cas nécessite d'abord une rotation afin d'équilibrer l'arbre, puis il pour la correction il fait intervenir le cas 1. À noter que les recolorations et les rotations sont essentielles pour rétablir les invariants de l'arbre rouge-noir. Les recolorations ajustent simplement les couleurs des nœuds pour rétablir les hauteurs noires, tandis que les rotations modifient la structure de l'arbre pour équilibrer les sous-arbres tout en maintenant les propriétés de l'arbre rouge-noir.

  En termes de complexité, les recolorations nécessitent une remontée jusqu'à la racine dans le pire des cas, ce qui donne une complexité logarithmique, soit **O(log(n))**. Les rotations, quant à elles, sont des opérations à temps constant **O(1)**, car elles ne nécessitent que des réaffectations de pointeurs et des modifications de couleurs. En somme, la suppression et la correction d'un nœud dans un arbre rouge-noir s'exécutent en **O(log(n))**, ce qui en fait une structure extrêmement efficace pour un accès rapide et un équilibrage dynamique.

2) Description du comportement du programme sur les jeux de tests

Le programme a été testé avec succès sur les différentes fonctionnalités de l'arbre rouge-noir à l'aide des fonctions de test fournies. Chaque test a permis de valider le bon fonctionnement des algorithmes en s'assurant que les résultats obtenus correspondent aux attentes.

- **Test de l'insertion dans l'arbre rouge-noir :**

La fonction *void bstree_add(ptrBinarySearchTree* t, int v)* permet d'insérer les nouvelles valeurs dans l'arbre rouge-noir et de conservers les invariants de celui-ci en faisant appelle à la fonction *fixredblack_insert(BinarySearchTree* x),* entièrement conçue pour cette conservation.

Les tests ont montré que les éléments sont correctement insérés tout en maintenant l'équilibre de l'arbre et en respectant les règles de coloration des nœuds. Nous avons pu vérifier cela grâce au fichier *DOT*, afin de visualiser l'arbre directement.

- **Test de la recherche dans un arbre rouge-noir :**

La fonction *const BinarySearchTree* bstree_search(const BinarySearchTree* t, int v*), permet d'effectuer des recherches dans un arbre rouge-noir pour vérifier que les valeurs sont correctement retrouvées.

Les tests ont confirmé que la fonction de recherche trouve les valeurs présentes dans l'arbre en respectant une complexité **O(log(n))**, ce qui est attendu pour un arbre rouge-noir.

- **Test de la suppression dans un arbre rouge-noir :**

La fonction *void bstree_remove(ptrBinarySearchTree* t, int v)*, permet de supprimer des valeurs dans l'arbre rouge-noir tout en préservant ses invariants. Cette fonction fait appel à la fonction *fixredblack_remove(BinarySearchTree* x)*, conçue spécifiquement pour rétablir les propriétés de l'arbre après suppression.

Les tests ont montré que les valeurs sont correctement supprimées, même dans les cas limites, tels que la suppression d'une feuille, d'un nœud avec un seul enfant, ou d'un nœud avec deux enfants. À mentionner que j'ai été contraint de modifier le fichier de test testfilesimple.txt afin d'obtenir le même résultat que sur le sujet fournis. En effet, dans le sujet fournis on devait supprimer la valeur 7 et non la valeur 8 qui été préalablement entrer.

Nous avons pu confirmer la bonne exécution des suppressions grâce à l'utilisation de fichiers *DOT*, permettant de visualiser directement l'état de l'arbre après chaque opération.

3) Analyse personnelle du travail effectué, problèmes rencontrés et solutions mises en place

La réalisation de ce TP sur les arbres rouge-noir a été à la fois exigeante et enrichissante, avec des niveaux de difficulté variables selon les exercices. Les exercices 1 et 2 ont été relativement simples, car ils reposaient sur des concepts vus en cours. En revanche, les exercices 3 (insertion) et 5 (suppression) ont été beaucoup plus complexes,

nécessitant une compréhension approfondie des mécanismes des arbres rouge-noir et une rigueur particulière dans la mise en œuvre des fonctions.

L'exercice 3 (insertion) a été l'une des parties les plus difficiles. La gestion des cas de correction après l'insertion, combinée à la nécessité de modifier la fonction *bstree_add(…)*, a généré plusieurs erreurs initiales. En effet, j'avais oublié d'appeler les fonctions de correction dans *bstree_add(…)* comme ce qui était demandé au début, ce qui faisait que les propriétés de l'arbre n'étaient pas rétablies. De plus, de mauvaises conditions dans la fonction de correction *fixredblack_insert(…)* ont entraîné de nombreuses erreurs de segmentation.

Pour résoudre ces problèmes, j'ai dû adopter une méthode rigoureuse de débogage. J'ai inséré des impressions (*print*) à différents endroits critiques du code afin de suivre étape par étape l'évolution des valeurs et des pointeurs. Il est vrai que l'utilisation de Valgring aurait été beaucoup plus efficace, mais je ne peux pas le télécharger car ce logiciel ne fournis pas d'extensions pour Mac récent. Néanmoins, cette autre méthode m'a permis d'identifier les zones problématiques et de corriger progressivement les erreurs. Une fois la fonction *bstree_add(…)* correctement modifiée et les conditions ajustées, les tests ont commencé à passer, validant ainsi le bon fonctionnement de l'insertion.

L'exercice 5 (suppression) a été encore plus complexe que l'exercice 3. La suppression dans un arbre rouge-noir est délicate en raison des multiples ajustements nécessaires pour rétablir les invariants après le retrait d'un nœud. L'erreur la plus persistante provenait du fait que, lors du remplacement (*swap*) d'un nœud avec son successeur, j'oubliais de corriger les couleurs des nœuds échangés. Cela faussait les résultats des conditions pour les fonctions de correction *case1_left, case1_right, case2_left, et case2_right*.

Ce problème a été particulièrement difficile à localiser, car il n'entraînait pas toujours un échec immédiat mais provoquait des incohérences dans l'arbre après plusieurs suppressions. Une fois encore, le débogage avec des impressions a été essentiel pour repérer et corriger cette erreur. J'ai également dû revoir en détail le fonctionnement des cas de correction et m'assurer que toutes les couleurs et rotations étaient appliquées correctement après chaque suppression. J'ai aussi souvent comparé les arbres attendus générer avec *graphviz* et ceux que j'obtenais.

Malgré les difficultés rencontrées, ce TP a été extrêmement formateur. Il m'a permis de renforcer ma compréhension des arbres rouge-noir et d'améliorer mes compétences en gestion des pointeurs et en débogage. La rigueur nécessaire pour préserver les invariants de cette structure de données m'a également appris à anticiper les erreurs et à structurer mon code de manière plus robuste.

4) Références concernant les documentations et aides utilisées

Pour la réalisation de ce projet, j'ai consulté plusieurs sources afin de m'assurer que mon implémentation était correcte et efficace :

- Camarades : Ils ont été une source d'aide, m'apportant des explications et des conseils précieux pour résoudre les problèmes complexes que je rencontrais.
- ChatGPT : Discussions et échanges d'idées pour clarifier certains concepts et erreurs rencontrés.
- PDF du cours : Matériel fourni en cours, essentiel pour la compréhension des concepts abordés dans le TP et la réalisation de celui-ci.