Arbres rouges-noirs

1 Introduction

1.1 Récupération des fichiers

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répertoire nommé `redblacks` contenant plusieurs fichiers dont une source C :

```
curl cdn.sci-phy.org/mp2i/tp15-redblacks.tgz | tar xvz
```

Le fichier à compléter est le fichier `redblacks.c`. On y trouvera quelques fonctions utiles qui seront présentées au fur et à mesure de nos besoins. On y trouve également un fichier `makefile` aidant à la compilation.

1.2 Buts de la séance

Le but est ici de se familiariser avec la manipulation d'arbres binaires en langage C, et plus particulièrement d'arbres binaires de recherche, dans des opérations de construction, de recherche et d'ajout. On s'intéressera également à leur equilibrage, au travers de la méthode des arbres rouges-noirs, afin de garantir une certaine efficacité aux différentes opérations.

Les structures de données en langage C faisant intervenir des pointeurs, comme c'est le cas ici, ne sont jamais simple à manipuler. Deux objectifs annexes seront poursuivis aujourd'hui :

- avoir une réflexion sur la manière d'écrire les programmes, en particulier concernant l'écriture de fonctions auxiliaires limitant les risques d'erreur de manipulation;
- effectuer un travail sur les tests unitaires pour s'assurer que le programme effectue bien le travail demandé.

Une stratégie de développement courante, appelée *tests-based development*, consiste à écrire tout d'abord un ensemble exhaustifs de tests, puis de développer ensuite le code jusqu'à ce que tous les tests soient réussis. En général, cela nécessite d'écrire une quantité de tests considérables, couvrant tous les cas possible, de manière à valider la moindre ligne de code que l'on va écrire. Il ne nous sera pas possible de procéder tout à fait de la sorte en une séance de deux heures, mais nous allons essayer de nous approcher de cet objectif autant que possible.

Notons que, dans ce TP, on ne fera pas précéder d'un préfixe `ptr_` les noms de variables qui sont des pointeurs, pour ne pas alourdir le contenu. Il est recommandé de se poser régulièrement la question du type et de la nature des objets manipulés.

2 Opérations élémentaires

2.1 Structure des arbres binaires

On définit une structure C permettant de construire des arbres binaires de la sorte :

```c
struct node {
    bool is_red;
    int value;
    struct node* parent;
    struct node* lchild;
    struct node* rchild;
};

typedef struct node Node;
```

La structure définit ce qu'il faut pour décrire un nœud de l'arbre, l'arbre lui-même étant simplement désigné par un pointeur vers sa racine.

On y remarque plusieurs choses : tout d'abord, il s'agit d'une structure destinée à représenter un arbre binaire-unaire, où les nœuds sont étiquetés par des entiers (champ `.value`). Nous nous en servirons pour représenter des arbres binaires de recherche. Un champ `.is_red` nous servira à indiquer la couleur des nœuds, dans un second temps, afin de procéder à des opérations d'équilibrage.

Plus inhabituel, s'il y a un pointeur vers chacun des deux fils (on utilisera `NULL` pour indiquer une absence de fils d'un côté et/ou de l'autre), il se trouve également un pointeur vers le père, ce qui diffère de ce que l'on a eu l'occasion de manipuler jusqu'à présent. Ce pointeur est utile car il permet de remonter dans l'arbre, ce qui peut être commode lorsque les fonctions que l'on écrit ne peuvent pas être de la forme récursive la plus courante. Notons que ce pointeur contiendra `NULL` pour la racine de l'arbre (et seulement pour la racine!)

Davantage de pointeurs signifie davantage de risques de commettre des erreurs. Pour limiter les problèmes, *on ne manipulera aucun pointeur directement*, c'est-à-dire qu'on ne modifiera jamais les champs `.lchild`, `.rchild` et `.parent`, et on s'efforcera même d'éviter de les lire autant que possible.

On dispose d'une fonction `new_node(bool is_red, int value)` prenant en argument un booléen indiquant si le nœud est rouge (`true`) ou noir (`false`) et l'étiquette qu'il porte, et retourne un nœud alloué, étiqueté et colorié, dont les trois pointeurs (enfants et parent) sont initialisés à `NULL`, soit un nœud qui n'est rattaché à aucun autre nœud (en d'autres termes, un arbre réduit à une seule feuille).

On fournit de même deux fonctions `attach_left(Node* parent, Node* child)` et `attach_right(Node* parent, Node* child)` qui permettent d'accrocher un nœud à un autre. Ce sont les deux uniques fonctions que l'on utilisera pour altérer les pointeurs contenus dans la structure ! Si l'on souhaite détacher un fils, on utilisera `NULL` comme paramètre pour `child` (le fils n'est pas supprimé, il faudra sans doute garder un pointeur vers celui-ci pour éviter une fuite de mémoire). Le père, en revanche, doit toujours pointer sur un nœud existant.

1. Étudier *soigneusement* ces deux fonctions pour comprendre ce qu'elles font. On remarquera en particulier qu'elles décrochent soigneusement les liens existants avant de les remplacer par d'autres liens, en prenant garde à ne pas déréférencer un pointeur `NULL`.

2. Dans `main`, écrire un programme utilisant ces trois fonctions pour construire un arbre de hauteur 2, et de taille 4, de la forme que vous souhaiterez, et dont les nœuds auront la couleur et l'étiquette qu vous choisirez.

On dispose d'une fonction `disp_tree` prenant en argument un pointeur vers la racine d'un arbre et affiche l'arbre en question dans la sortie standard.

3. Vérifier, en affichant l'arbre, qu'il est bien celui que l'on a souhaité définir.

2.2 Manipulation des arbres

Dans un premier temps, nous allons écrire quelques fonctions générales sur les arbres binaires.

4. Compléter la fonction `size` afin qu'elle retourne la taille de l'arbre binaire passé en paramètre.

5. Compléter la fonction `get_values` afin qu'elle effectue un parcours en profondeur *infixe* de l'arbre binaire passé en premier paramètre, place dans le tableau fourni en second paramètre les éléments obtenus (dans le cas d'un arbre binaire de recherche, ces éléments devraient donc être rangés par ordre croissant), **et retourne le nombre de valeurs ainsi écrites**. On supposera le tableau suffisamment grand.

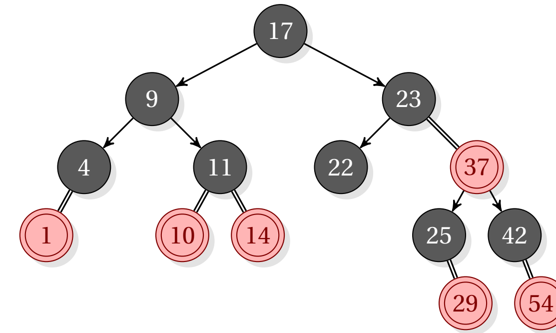
Une fois ces deux fonctions convenablement écrites, on peut utiliser la fonction `disp_values` qui prend en argument un arbre et affiche les valeurs obtenues par le parcours en profondeur infixe.

Pour tester les différentes fonctions que l'on va écrire, on fournit une fonction `build_tree` permettant de construire plus facilement des arbres. Pour ce faire, on a besoin d'un tableau contenant une description de l'arbre à construire, sous la forme d'un parcours en profondeur postfixe de l'arbre, `NULL` compris. Le tableau `ex_tree_data` en est un exemple. `{'X',0}` fait référence à `NULL`, `{'B',42}` à un nœud noir portant la valeur 42, `{'R',37}` à un nœud rouge portant la valeur 37.

À partir d'un tel tableau et de sa taille, on peut construire un arbre, comme le fait l'instruction, au début de la fonction `main` :

```
Node* ex_tree = build_tree(ex_tree_data,
                           sizeof(ex_tree_data)/sizeof(struct rbdata));
```

Il s'agit de l'arbre ci-dessous :



6. Utiliser la fonction `size` sur cet arbre exemple pour vérifier son bon fonctionnement. On pourra également utiliser `size` sur l'arbre de taille 4 déclaré précédemment, et sur un arbre vide (`NULL`).

7. Utiliser la fonction `disp_values` et vérifier que toutes les valeurs de l'arbre sont bien affichées dans le bon ordre.

8. En s'inspirant de `disp_values`, proposer une fonction `is_bst` prenant en argument un arbre et retournant un booléen indiquant si les valeurs respectent les règles d'un arbre binaire de recherche.

2.3 Recherche d'un nœud

Les arbres que l'on manipule sont des arbres binaires de recherche. Il nous faut donc une fonction capable de retrouver, dans un arbre donné, un nœud portant une étiquette donnée s'il en existe un. La fonction devra retourner un **pointeur** vers un tel nœud s'il existe, et `NULL` dans le cas contraire.

9. Proposer une implémentation de la fonction `find` réalisant cette opération.

10. En déduire une fonction `mem` qui vérifie la présence d'un élément dans l'arbre fourni en paramètre, et retourne un booléen correspondant.

11. Tester la fonction `mem` sur l'arbre d'exemple avec les valeurs 42, 22, 1, 54 (tous les quatre présents) et -1, 15, 68 (tous trois absents).

12. Expliquer les choix effectués pour les tests. Il y a une raison distincte pour chacun des sept tests.

2.4 Liens de parenté

Pour faciliter l'écriture de certaines fonctions, on se propose d'écrire des fonctions auxiliaires permettant d'obtenir un pointeur vers un nœud à partir d'un pointeur vers un autre nœud et d'un lien de parenté. Un exemple est la fonction `parent` qui prend un pointeur vers un nœud et retourne un pointeur vers son parent.

Si, pour quelque raison que ce soit (le pointeur fourni est `NULL`, le parent n'existe pas, etc.), ces fonctions doivent systématiquement retourner `NULL` sans causer d'erreur.

13. Proposer des implémentations pour les fonctions `lchild` (fils gauche), `rchild` (fils droit), `brother` (frère), `uncle` (oncle) et `greatparent` (grand-parent).

14. Afficher le père, le grand-père, le frère, l'oncle et les deux fils du nœud 37 pour l'arbre fourni en exemple. On pourra procéder de la sorte, par exemple, pour le père :

```
printf("Le père de 37 est ");
print_node( parent( find(ex_tree, 37) ) );
printf("\n");
```

15. Faire de même avec le nœud 9 pour lequel certains des liens de parenté ne conduisent à aucun nœud (`disp_node` affiche « None » dans ce cas).

3 Modifications de l'arbre

3.1 Ajout d'un élément

Dans un premier temps, nous allons ajouter les éléments sans nous soucier d'équilibrer l'arbre ni de respecter les contraintes.

16. Proposer une fonction `add` prenant en argument un pointeur vers un arbre et un entier, et ajoutant cet entier à l'arbre. On n'ajoutera l'élément que s'il n'est pas déjà présent dans l'arbre (on considérera donc un arbre binaire de recherche dépourvu de doublons!) et on effectuera l'ajout au niveau des feuilles, sous la forme d'un nœud rouge. On retournera un pointeur vers la racine de l'arbre modifié.

17. Pourquoi faut-il retourner le pointeur vers le nouvel arbre? Dans quel cas diffère-t-il du pointeur fourni à la fonction?

18. Construire un arbre avec les entiers de 1 à 20, insérés dans cet ordre, et afficher le résultat avec `disp_tree`. Que penser de l'arbre obtenu obtenu?

19. Faire de même en construisant un arbre en insérant successivement les entiers $(17 \times i) \mod 21$ pour i de 1 à 20. Comparer le résultat obtenu en affichant l'arbre. On vérifiera aussi (par exemple avec `disp_values` que l'on a bien obtenu un arbre binaire de recherche étiqueté avec les éléments de 1 à 20.

3.2 Propriétés des arbres rouge-noir

Pour vérifier que les fonctions qui vont imposer les contraintes supplémentaires des arbres rouge-noir fonctionnent bien on va écrire des fonctions vérifiant les propriétés des arbres.

20. Proposer une fonction `has_double_red` prenant en argument un arbre et retournant un booléen indiquant si l'arbre contient un nœud rouge dont le parent est rouge.

21. Vérifier que l'arbre fourni en exemple passe ce critère.

22. Changer un nœud noir bien choisi en nœud rouge et vérifier que l'arbre modifié ne passe plus la vérification.

23. Proposer une fonction `black_height` prenant en argument un arbre et retournant sa hauteur noire, si toutes les branches ont la même hauteur noire, et -1 si leurs hauteurs noires diffèrent.

24. Vérifier la hauteur noire de l'arbre passé en argument.

25. Changer un nœud noir bien choisi en nœud rouge et vérifier que l'arbre modifié donne -1 lors d'un appel à `black_height`.

26. Il manque une condition pour que l'arbre puisse être considéré comme un arbre rouge-noir valide, laquelle? Proposer une fonction `is_rb_tree` retournant un booléen indiquant si l'arbre passé en paramètre est un arbre respectant les règles des arbres rouges-noirs.

3.3 Équilibrage

Maintenant que l'on dispose de tous les outils nécessaires pour vérifier le bon fonctionnement des différentes fonctions, nous allons essayer de corriger le problème possiblement induit par l'ajout d'un nœud supplémentaire rouge dans l'arbre.

27. Proposer une fonction `repair_red` prenant en argument un pointeur vers un nœud rouge possiblement problématique, et corrigeant le problème comme décrit dans le cours.

Pour simplifier l'écriture de cette fonction, outre les fonctions de parenté précédemment décrites, on dispose de fonctions auxiliaires `is_left_child`, `is_right_child`, `is_root`, `is_red_node` et `is_black_node` permettant de tester les propriétés d'un nœud. Dans les trois dernier cas, les fonctions retournent `false` si le nœud n'existe pas. Par exemple, pour tester s'il y a un oncle rouge, on peut écrire :

```
if is_red_node( uncle( node ) ) { ... }
```

28. Ajouter l'appel à la fonction précédente dans la fonction `add`.

29. Reprendre l'exemple de la création d'un arbre en partant d'un arbre vide et en insérant les entiers de 1 à 20 dans un arbre initialement vide. Vérifier que le résultat

obtenu est le bon, en affichant le résultat (avec `disp_tree` et `disp_values`) et en testant les propriétés de l'arbre (avec `has_double_red` et `black_height`).

3.4 Suppression

Les plus rapides peuvent s'essayer à la suppression d'un élément. On pourra d'abord supprimer l'élément sans tenir compte des contraintes des arbres rouge-noir (on pourra préalablement créer une fonction `successor` retournant un successeur dans l'arbre s'il en existe, `NULL` si ce n'est pas le cas), puis dans un second temps envisager la réparation de l'arbre suite à la suppression en créant deux fonctions récursives, `rebalance_left` et `rebalance_right` à appeler sur les nœuds présentant un déséquilibre de hauteur noire entre la gauche et la droite.