Compte rendu de la séquence 6 - Arbres Rouges et Noir

Au long de ces séances de TP, nous avons implémenté un arbre binaire de recherche rouge et noir, fonctionnant comme un arbre binaire de recherche classique, mais avec des propriétés supplémentaires, nous permettant d'avoir, pour toute opération de recherche, d'ajout ou de suppression, une complexité en O(log(n)) dans le pire cas.
Ce rapport a pour but d'expliquer uniquement la "partie rouge et noir" de cette implémentation, c'est-à-dire celle qui concerne spécifiquement le TP6.

Implémentations

Coloration des noeuds

Nous avons ajouté un champ à notre structure d'arbre binaire : la couleur d'un nœud, représentée de la manière suivante :

```
typedef enum {
 red,
 black
} NodeColor;
```

Il s'agit d'une énumération qui possède deux valeurs : red pour rouge et black pour noir. L'énumération reste privée pour l'utilisateur, ce dernier n'en ayant pas besoin pour utiliser l'arbre binaire de recherche.

Notre structure d'arbre binaire de recherche ressemble donc à ceci désormais :

```
struct _bstree {
 NodeColor color;
 BinarySearchTree* parent;
 BinarySearchTree* left;
 BinarySearchTree* right;
 int key;
};
```

Plusieurs fonctions ont été implémentées afin de gérer la couleur d'un nœud, notamment la fonction `color` :

```

```

```
NodeColor color(const BinarySearchTree *x){
 return x && x->color == red
  ? red
  : black;
}
```

Cette fonction renvoie simplement la couleur d'un noeud x, si ce dernier est vide, la couleur renvoyer est donc `black`.

Opérateurs de rotations

Afin de pouvoir mener certaines opérations à bien sur notre arbre rouge et noir, nous avons ajouté deux opérateurs privés pour les rotations à gauche et à droite : `leftrotate` et `rightrotate`.
Penchons nous sur `leftrotate` :

```
void leftrotate(BinarySearchTree *x){
 assert(!bstree_empty(x)&&!bstree_empty(x->right));

 ptrBinarySearchTree x_parent_or_x = x->parent ? x->parent : x;
 ptrBinarySearchTree y = x->right;

 //échange de x et du fils gauche : y
 bstree_swap_nodes(&x_parent_or_x,x,y);

 //échange des deux enfants de y.
 y->right = y->left;
 y->left = x;

 //échange des enfants de x
 ptrBinarySearchTree tmp_left_x = x->left;
 x->left = x->right;
 x->right= tmp_left_x;

 //échange du fils gauche de x avec l'enfant droit de y
 ptrBinarySearchTree tmp_right_y = y->right;
 y->right= x->left;
 x->left = tmp_right_y;

 if(y->right) y->right->parent = y;
 if(x->left)  x->left->parent = x;

}
```

`rightrotate` fonctionne de manière similaire, mais de façon symétrique entre la droite et la gauche.

Cette fonction est maintenue privée, car l'utilisateur n'en a pas besoin, et son utilisation pourrait déséquilibrer l'arbre, réduisant ainsi son efficacité.

Corection de l'insertion d'une valeur dans l'arbre

Afin de garantir les propriétés de l'arbre binaire de recherche rouge et noir, nous devons corriger l'arbre à chaque insertion. Pour cela, nous avons implémenté la fonction `fixredblack_insert`, qui fonctionne de la manière suivante :

```
BinarySearchTree* fixredblack_insert(BinarySearchTree* x) {
 if (x->parent == NULL) {
  // Case 0
  x->color = black;
  return x;
 }

 ptrBinarySearchTree p = x->parent;

 if (color(p) == black)
  return x;

 // Case 1 or Cases 2
 return fixredblack_insert_case1(x);
}
```

Cette fonction a pour but de vérifier si une correction peut être apportée directement sans grandes modifications, ou si des ajustements supplémentaires sont nécessaires. Dans ce cas, on vérifie un cas 1 puis un cas 2.
Ces cas ne sont pas détaillés ici, car leur logique est très bien expliquée dans le sujet et il suffit de suivre cette logique à la lettre.

Corection de la supression d'une valeur dans l'arbre

De la même manière que précédement nous "corrigeons" l'arbre après en avoir supprimer une valeur, avec la fonction `fixredblack_remove` :

```
ptrBinarySearchTree fixredblack_remove(ptrBinarySearchTree p,
ptrBinarySearchTree x){
 if(!p){
  if(x) x->color = black;
  return x;
 }
 if(color(sibling(p,x)) == black)
  return fixredblack_remove_case1(p,x);
 else
  return fixredblack_remove_case2(p,x);
}
```

Cette fonction fonctionne comme `fixredblack_insert`, elle décide simplement du cas à traiter et le traite.