TP2 Caml : Arbres blancs et noirs

Lors du dernier TP nous avons vu que les opérations de base sur les arbres binaires de recherche (insérer, supprimer, rechercher) se faisait dans un temp $\mathcal{O}(h)$ avec h la hauteur de l'arbre. Vous avez vu en cours que si n est le nombre de noeuds de l'arbre

$$\log_2(n+1) \leq h \leq n$$

Pour que les opérations évoquées précédemment soit efficaces (par rapport à une simple structure de liste), la hauteur h doit rester petite. Nous allons nous intéresser ici à des arbres binaires de recherche particuliers, les *arbres blancs et noirs*. Ces arbres possèdent la propriété suivante

$$\log_2(n+1) \leq h \leq 2\log_2(n+1)$$

On peut ainsi effectuer des mises à jours et des recherches en $\mathcal{O}(\log_2(n))$.

1 Définition et structure de donnée

Un arbre blanc et noir et un arbre binaire de recherche dont chaque nœud ou feuille possède une couleur. Il doit vérifier les contraintes suivantes :

1. un nœud est soit blanc, soit noir,
2. une feuille est noire,
3. la racine est noire,
4. le père d'un nœud blanc est noir,
5. tous les chemins partant d'un nœud donné et se terminant à une feuille contiennent le même nombre de nœuds noirs.

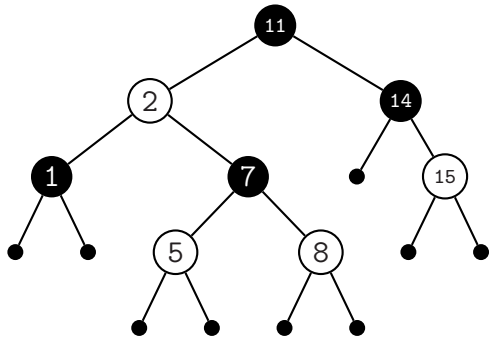
Pour représenter ces arbres en Caml on utilise le type suivant :

```
type Couleur = Blanc | Noir ;;
type ArbreRN =
   Vide
 | Noeud of enreg_Noeud
and enreg_Noeud =
  { mutable val     : int;
    mutable couleur : Couleur;
    mutable gauche  : ArbreRN;
    mutable droit   : ArbreRN;
    mutable pere    : ArbreRN; } ;;
```

Pour pouvoir parcourir l'arbre en montant et en descendant, on a rajouté un champ `pere` qui pointe vers le père du nœud courant. Si un nœud n'a pas de père , on met son champ `pere` à `vide` (c'est le cas de la racine d'un arbre).
Afin de tester les fonctions écrites dans la suite de ce TP, on considérera cet exemple :

```
let rec a11 = Noeud {val=11; couleur=Noir; gauche=a2; droit=a14; pere=Vide}
and a2 = Noeud {val=2; couleur=Blanc; gauche=a1; droit=a7; pere=a11}
and a1 = Noeud {val=1; couleur=Noir; gauche=Vide; droit=Vide; pere=a2}
and a7 = Noeud {val=7; couleur=Noir; gauche=a5; droit=a8; pere=a2}
and a5 = Noeud {val=5; couleur=Blanc; gauche=Vide; droit=Vide; pere=a7}
and a8 = Noeud {val=8; couleur=Blanc; gauche=Vide; droit=Vide; pere=a7}
and a14 = Noeud {val=14; couleur=Noir; gauche=Vide; droit=a15; pere=a11}
and a15 = Noeud {val=15; couleur=Blanc; gauche=Vide; droit=Vide; pere=a14} ;;
```

Il correspond à cet arbre :



2 Lecture d'information

Toutes les fonctions de cette partie devront renvoyer un message d'erreur si l'arbre passé en argument n'a pas la structure requise.
1. Ecrire une fonction `pere : arbreRN -> arbreRN` qui renvoie le père du nœud racine d'un arbre. Définir de la même manière les fonctions `fils_gauche`, `fils_droit`, `valeur` et `couleur`.

2. En déduire la fonction `est_fils_gauche : arbreRN -> bool` qui test si le nœud racine d'un arbre est le fils gauche de son père. Définir de la même manière les fonctions `est_fils_droit`

3. En déduire les fonctions `frere` et `oncle : arbreRN -> arbreRN`.

3 Tests

4. Pour pouvoir afficher ces arbres, écrire une fonction `transforme` qui convertit un arbre de type `arbreRN` en arbre de type `ArbreClassique` où ce dernier type est défini par :

```
type ArbreClassique =
   V
 | N of int*Couleur*ArbreClassique*ArbreClassique ;;
```

5. Ecrire une fonction `test_pere : arbreRN -> bool` qui vérifie que tous les champs `pere` d'un arbre sont valides. On utilisera une fonction auxiliaire `test_pere_aux : arbreRN -> arbreRN -> bool` telle que `test_pere_aux p a` test l'arbre `a` en imposant que `p` soit le père de la racine de `a`.

6. Ecrire une fonction `test_prop4 : arbreRN -> bool` qui teste si un arbre de type `arbreRN` vérifie la contrainte 4 des arbres blancs et noirs.

7. Ecrire une fonction `nb_noirs : arbreRN -> int` qui renvoie le nombre de nœuds noirs que contient n'importe quel chemin reliant la racine d'un arbre à une de ses feuilles. Si au cours du calcul, la contraintes 5 des arbres blancs et noirs n'est pas vérifié la fonction renvoie une erreur de ce type :

```
exception StopProp5 ;;
```

8. En déduire une fonction `test_prop5 : arbreRN -> bool` qui teste si un arbre de type `arbreRN` vérifie la contrainte 5 des arbres blancs et noirs (en utilisant `try...`).

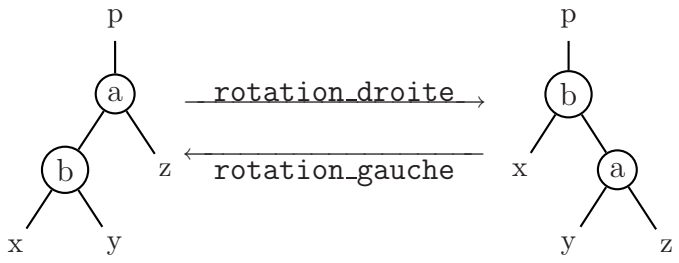
9. En déduire une fonction `testRN : arbreRN -> bool` qui teste si un arbre de type `arbreRN` est un arbre blanc et noir.

4 Modifications

10. Ecrire une fonction `inverse_couleur : arbreRN -> unit` qui inverse la couleur du nœud racine d'un arbre.

11. Ecrire une fonction `adopte_gauche` de type `arbreRN -> arbreRN -> unit` telle que `adopte_gauche f p` donne à la racine de `p` l'arbre `f` comme fils gauche et modifie en conséquence `f`. Définir de manière symétrique la fonction `adopte_droit`.

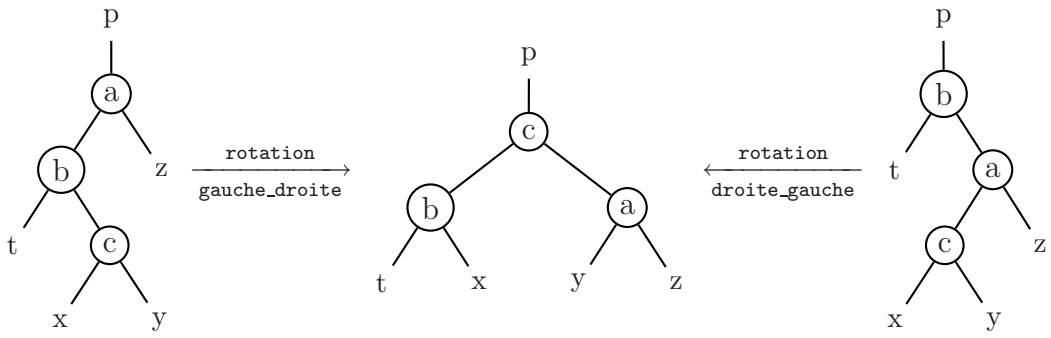
12. En déduire les fonctions `rotation_droite` et `rotation_gauche : arbreRN -> unit` qui réalisent les modifications suivantes sur un arbre :



Vérifier que ces deux transformations conservent les propriétés d'un arbre binaire de recherche.

13. En déduire les fonctions `rotation_gauche_droite` et `rotation_droite_gauche` sui-

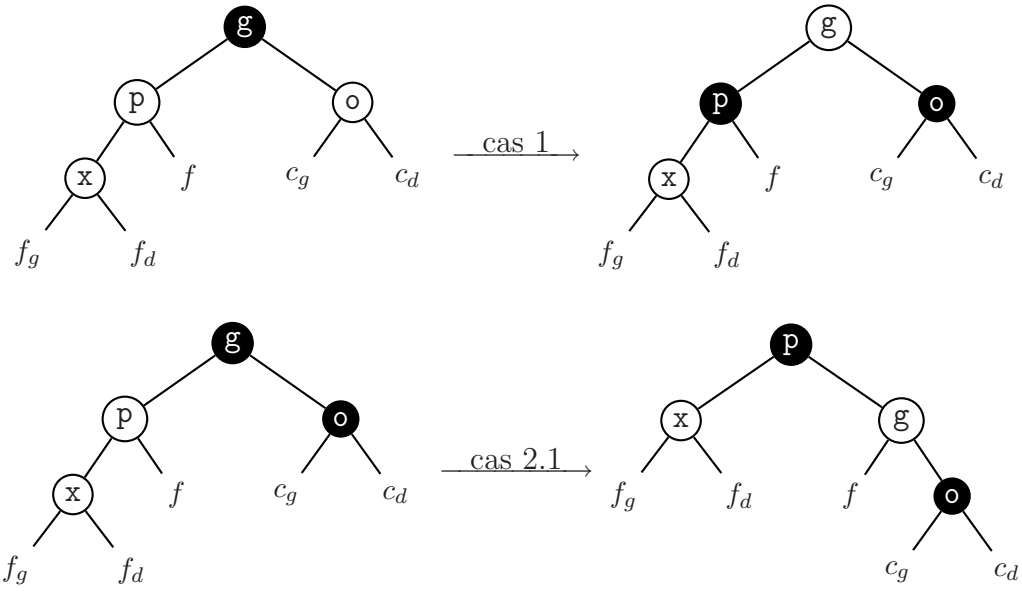
vantes :



14. Ecrire une fonction `insere_blanc: int -> arbreRN -> arbreRN` qui insere une valeur dans un arbre blanc et noir au niveau des feuilles et en lui donnant une couleur blanche. La fonction doit renvoyer le sous-arbre dont le nouveau nœud est la racine. Cette insertion doit seulement maintenir la structure d'arbre binaire de recherche. Si la valeur à insérer est déja présente dans l'arbre la fonction déclenche l'exception suivante :

```
exception FinInsertion ;;
```

15. Ecrire une fonction `mauvais_blanc : arbreRN -> unit` qui prend un sous-arbre `x` en argument dont la racine est blanche et qui appartient à un arbre «quasiment»blanc et noir : cet arbre vérifie toutes les contraintes énoncées sauf éventuellement au niveau de `x` dont le père peut être blanc. Cette fonction doit alors modifier l'arbre pour qu'il respecte toutes les propriétés des arbres blancs et noirs. Utiliser pour cela le squelette fourni dans le fichier `TP2.squ` et les schémas suivants qui donnent la solution pour deux cas particuliers :



16. En déduire la fonction `insere : int -> arbreRN -> unit` qui insere une valeur dans un arbre blanc et noir en conservant ses propriétés.

17. Modifier les fonctions précedentes de façon à connaître la racine de l'arbre modifiée.