Correction du TP4 : Arbres binaires de recherche

1. Voici les commandes pour définir l'arbre de recherche `ex_1` :

```
let ex_1= Noeud(27,
                Noeud(12,
                      Noeud(5,Vide,Vide),
                      Noeud(19,
                            Noeud(17,Vide,Vide),
                            Noeud(24,
                                  Noeud(20,Vide,Vide),
                                  Noeud(26,Vide,Vide)))),

                Noeud(43,
                      Noeud(36,Vide,Vide),
                      Noeud(77,Vide,Vide)))
;;
```

2. (a) Voici la fonction `recherche` demandée :

```
let rec recherche arbre compare x  = match arbre with
  | Vide              -> false
  | Noeud (e, ag, ad) -> match compare x e with
    | Egal -> true
    | Inf  -> recherche ag compare x
    | _    -> recherche ad compare x
;;
```

   (b) Choisissons comme taille des données la hauteur h de l'arbre `a` et comme opérations fondamentales les comparaisons. Calculons la complexité au pire.
   Notons $C(h)$ le nombre de comparaisons effectuée par l'appel de `recherche a compare x` dans le pire des cas.
   $C(-1) = 0$; sinon, si $h \geqslant 0$, la fonction fait appel à la fonction compare qui effectue au pire 2 comparaisons puis un appel récursif à elle-même sur une donnée de taille inférieure à $h-1$ (sous-arbre de gauche ou de droite).
   Donc $C(h) = 2 + C(h-1)$. On reconnaît une suite arithmétique de raison 2 et de premier terme $C(0) = 2$ donc $C(h) = 2(h+1) = 2h+2 = O(h)$.

3. Voici la fonction `min_gauche_max_droit` :

```
let rec min_gauche_max_droit arbre = match arbre with
  | Vide                -> failwith "arbre vide"
  | Noeud(e,Vide,Vide) -> (e,e)
  | Noeud(e,ag,Vide)    ->
    let (min,max) = (min_gauche_max_droit ag) in (min,e)
  | Noeud(e,Vide,ad)    ->
    let (min,max) = (min_gauche_max_droit ad) in (e,max)
  | Noeud(_,ag,ad)      ->
    (fst (min_gauche_max_droit ag), snd (min_gauche_max_droit ad))
;;
```

4. Voici les fonctions `verifie_encadre` et `verif_arb_rech` :

```
let rec verifie_encadre a compare x y = match a with
  | Vide -> true
  | Noeud(e,ag,ad) -> not (compare e x = Inf)
                      && not (compare e y = Sup)
                      && verifie_encadre ag compare x e
                      && verifie_encadre ad compare e y
;;

let verif_arb_rech arbre compare =
  let min, max = min_gauche_max_droit arbre in
    verifie_encadre arbre compare min max
;;
```

5. Voici les fonctions `verif_tri`, `infixe` et `verif_arb_rech2` :

```
let rec verif_tri l compare = match l with
  | [x]   ->true
  | t1::q -> let t2::q2 = q in
    not (compare t1 t2 = Sup) && (verif_tri q compare)
;;

let rec infixe arbre = match arbre with
  | Vide            -> []
  | Noeud(e,ag,ad)  -> (infixe ag)@[e]@(infixe ad)
;;

let verif_arb_rech2 arbre compare = verif_tri (infixe arbre) compare;;
```

6. (a) Voici la fonction `insere_feuille` :

```
let rec insere_feuille arbre compare x = match arbre with
  | Vide           -> Noeud(x,Vide,Vide)
  | Noeud(e,ag,ad) -> match compare x e with
    | Egal -> arbre
    | Inf  -> Noeud(e,insere_feuille ag compare x,ad)
    | Sup  -> Noeud(e,ag,insere_feuille ad compare x)
;;
```

(b) Voici les fonctions `decoupe` et `insere_racine` :

```
let rec decoupe arbre compare x = match arbre with
  | Vide -> Vide, Vide
  | Noeud(e,ag,ad) -> match compare x e with
    | Egal -> ag,ad
    | Inf  -> let ag2,ad2 = decoupe ag compare x in
        ag2, Noeud(e,ad2,ad)
    | Sup  -> let ag2,ad2 = decoupe ad compare x in
        Noeud(e,ag,ag2), ad2
;;
```

```
let insere_racine arbre compare x =
  let ag,ad = decoupe arbre compare x in Noeud(x,ag,ad)
;;
```

7. (a) Voici la fonction `construction_feuille` :

```
let rec construction_feuille l compare= match l with
  | [] -> Vide
  | t::q ->
    insere_feuille (construction_feuille q compare) compare t
;;
```

   (b) Voici la fonction `construction_racine`

```
let rec construction_racine l compare= match l with
  | [] -> Vide
  | t::q ->
    insere_racine (construction_racine q compare) compare t
;;
```

   (c) On a avec les listes données dans l'énoncé :

```
let liste_ex_1 = [1;2;3;4;5;6;7;8;9];;
let liste_ex_2 = [1;3;5;7;9;8;6;4;2];;

construction_feuille liste_ex_1 compare_int;;

- : arbre_bin =
Noeud (9,
 Noeud (8,
  Noeud (7,
   Noeud (6,
    Noeud (5,
     Noeud (4, Noeud (3, Noeud (2, Noeud (1, Vide, Vide), Vide), Vide), Vide), Vide)
     Vide),
    Vide),
   Vide),
  Vide),
 Vide)

construction_racine liste_ex_1 compare_int;;

- : arbre_bin =
Noeud (1, Vide,
 Noeud (9,
  Noeud (8,
   Noeud (7,
    Noeud (6,
     Noeud (5, Noeud (4, Noeud (3, Noeud (2, Vide, Vide), Vide), Vide), Vide), Vide)
     Vide),
```

```
    Vide),
   Vide),
  Vide))

construction_feuille liste_ex_2 compare_int;;

- : arbre_bin =
Noeud (2, Noeud (1, Vide, Vide),
 Noeud (4, Noeud (3, Vide, Vide),
  Noeud (6, Noeud (5, Vide, Vide),
   Noeud (8, Noeud (7, Vide, Vide), Noeud (9, Vide, Vide)

construction_racine liste_ex_2 compare_int;;

- : arbre_bin =
Noeud (1, Vide,
 Noeud (2, Vide,
  Noeud (4, Noeud (3, Vide, Vide),
   Noeud (6, Noeud (5, Vide, Vide),
    Noeud (8, Noeud (7, Vide, Vide), Noeud (9, Vide, Vide))))))
```

Les arbres obtenus sont très déséquilibrés.

8. Voici les fonctions `rotation_gauche` et `rotation_droite` :

```
let rotation_gauche arb = match arb with
  | Noeud(x,a,Noeud(y,b,c)) -> Noeud(y,Noeud(x,a,b),c)
  | _                       -> arb
;;

let rotation_droite arb = match arb with
  | Noeud(y,Noeud(x,a,b),c) -> Noeud(x,a,Noeud(y,b,c))
  | _                       -> arb
;;
```

9. (a) Cette fonction fusionne deux arbres binaires de recherche.

   (b) Pour s'assurer de la terminaison, on vérifie la décroissance stricte selon l'ordre lexicographique de la suite des $(|a1|, |a2|)$:
   - Si $e1 < e2$, on lance un premier appel récursif avec comme arguments $a'1 = Noeud(e1, ag1, Vide)$ qui est $a1$ privé de son sous-arbre droit et $ag2$. Comme $|a'1| \leqslant |a1|$ et $|ag2| < |a2|$, on a bien $(|a'1|, |ag2|) <_{\text{lex}} (|a1|, |a2|)$. Le deuxième appel récursif est fait avec $ad1$ et un arbre c, mais comme $|ad1| < |a1|$, on a à nouveau $(|ad1|, |c|) <_{\text{lex}} (|a1|, |a2|)$.
   - On raisonne de même si $e1 \geqslant e2$.

   Ainsi, $(\mathbb{N}^2, \leqslant_{\text{lex}})$ étant bien fondé, l'algorithme se termine en un temps fini.

   (c) Pour prouver la correction, nous allons raisonner par principe d'induction forte sur $(\mathbb{N}^2, \leqslant_{\text{lex}})$. Prouvons le prédicat $\mathcal{P}_{(n_1,n_2)}$: "L'algorithme est correct pour tous les arbres `a1` et `a2` de tailles respectives n_1 et n_2".

Ini. Si l'un des deux arbres est vide, il est clair que la fusion des deux est l'autre. L'algorithme est donc correct pour les cas de base c'est-à-dire les éléments de tailles $(n, 0)$ et $(0, n)$.

Héré. Soient $(a1, a2)$ un couple d'arbres non vides. Supposons que l'algorithme est correct pour tous les couples d'arbres dont le couple de tailles est strictement inférieur à $(|a1|, |a2|)$.

Si $e1 < e2$, alors on sait que $e1$, $ag1$ et $ag2$ ont des étiquettes inférieures à $e2$ et $ad2$ a des étiquettes supérieures à $e2$, mais on ne sait *a priori* pas situer $ad1$ par rapport à $e2$.

Dans ce cas, on fusionne $Noeud(e1, ag1, Vide)$ et $ag2$, on crée l'arbre c ayant cette fusion comme sous-arbre gauche, $e2$ comme racine et $ad2$ comme sous-arbre droit (qui est bien un arbre binaire de recherche) , et on fusionne $ad1$ avec cet arbre c.

La première fusion se fait sur les arbres $a'1 = Noeud(e1, ag1, Vide)$ qui est $a1$ privé de son sous-arbre droit et $ag2$. Avec $(|a'1|, |ag2|) <_{\text{lex}} (|a1|, |a2|)$ et la seconde fusion se fait sur les arbres $ad1$ et c, mais comme $|ad1| < |a1|$, on a à nouveau $(|ad1|, |c|) <_{\text{lex}} (|a1|, |a2|)$.

Si on suppose la fusion correcte pour les arbres dont les arguments sont strictement inférieurs, alors le résultat obtenu est bien un arbre binaire de recherche.

On raisonne de même si $e1 \geqslant e2$.

Ccl. Par principe d'induction forte, l'algorithme renvoie bien un nouvel arbre binaire de recherche obtenu en fusionnant les deux arbres donnés en entrée.