J'ai choisi d'utiliser dans ce corrigé la commande `function` plutôt que la commande `fun` pour programmer les fonctions du TP. Contrairement à la commande `fun`, la commande `function` n'admet qu'un paramètre, mais demande moins d'efforts de parenthésage.

1 Opérations de base sur les arbres binaires

On remarque que les quatre fonctions suivantes ont toutes la même structure.

```
let rec hauteur = function
   Feuille _ -> 0
 | Noeud (_,g,d) -> 1 + max (hauteur g) (hauteur d) ;;

let rec nombre_feuille = function
   Feuille _ -> 1
 | Noeud (_,g,d) -> nombre_feuille g + nombre_feuille d ;;

let rec nombre_noeud = function
   Feuille _ -> 0
 | Noeud (_,g,d) -> 1 + nombre_noeud g + nombre_noeud d ;;

let rec miroir = function
   Feuille f -> Feuille f
 | Noeud (n,g,d) -> Noeud (n,miroir d,miroir g) ;;
```

2 Arbres binaires de recherche

```
let rec insere comp x = function
   Vide -> Noeud (x,Vide,Vide)
 | Noeud (n,g,d) -> if x=n then Noeud (n,g,d)
                    else if comp x n then Noeud (n,insere comp x g,d)
                         else Noeud (n,g,insere comp x d) ;;

let rec retire_plus_grand = function
   Vide -> failwith "arbre vide"
 | Noeud (n,g,Vide) -> (n,g)
 | Noeud (n,g,d) -> let (m,a) = retire_plus_grand d in (m,Noeud (n,g,a)) ;;
```

D'un point de vue complexité, il serait catastrophique d'écrire pour cette dernière ligne :

```
 ...
 | Noeud (n,g,d) -> (fst (retire_plus_grand d),
                     Noeud (n,g,snd (retire_plus_grand d))) ;;
```

En effet, on fait alors deux appels récursifs au lieu d'un et la complexité passe d'un $\mathcal{O}(h)$, à un $\mathcal{O}(h^2)$ (où h est la hauteur de l'arbre).

```
Let retire_racine = function
   Vide -> failwith "arbre vide"
 | Noeud (n,Vide,d) -> d
 | Noeud (n,g,d) -> let (m,a) = retire_plus_grand g in Noeud (m,a,d) ;;

let rec retire comp x = function
  Vide -> Vide
| Noeud (n,g,d) -> if x=n then retire_racine (Noeud (n,g,d))
                   else if comp x n then Noeud (n,retire comp x g,d)
                        else Noeud (n,g,retire comp x d) ;;

let rec separe comp x = function
  Vide -> (Vide,Vide)
| Noeud (n,g,d) -> if x=n then (g,d)
                   else if comp x n then
                        let (a,b)=separe comp x g in (a,Noeud (n,b,d))
                   else let (a,b)=separe comp x d in (Noeud (n,g,a),b) ;;

let insere_racine comp x arbre =
  let (g,d)=separe comp x arbre in Noeud (x,g,d) ;;

let rec test comp a b = function
  Vide -> true
| Noeud (n,g,d) -> comp a n & comp n b &
                   test comp a n g & test comp n b d;;

let test_int = test (prefix <) min_int max_int;;
```

On utilise ici les valeurs `min_int` et `max_int` prédéfinies en CAML pour donner l'intervalle de définition des entiers. La fonction `prefix` permet quant à elle d'obtenir la version préfixe de n'importe quel opérateur infixe de CAML.

3 Représentation avec des pointeurs

Les deux fonctions demandées sont en fait des procédures : elles ne renvoient pas de résultat (plus précisement elles renvoient `()` de type `unit`). Elles agissent sur l'arbre qu'on leurs donne en paramètre.

```
let rec miroir = function
  Vide_p -> ()                              (* rien à faire *)
| Noeud n ->  begin
                miroir n.gauche;        (* on modifie le fils gauche *)
                miroir n.droit;         (* on modifie le fils droit *)
                let temp=n.gauche in
                n.gauche <- n.droit;    (* on permute les deux fils *)
                n.droit <- temp
              end ;;
```

Cette fonction d'insertion se limite à un arbre non vide car on ne peut pas modifier directement l'arbre qui est passé en paramètre : on ne peut modifier que ses champs mutables. Il faut donc prendre garde à ne pas l'appeler récursivement sur un fils vide du nœud courant. Si le fils convoité est vide on remplace la valeur `Vide_p` du champs mutable correspondant par le nœud `Noeud { val=x; gauche=Vide_p; droit=Vide_p }`. Cette modification n'aurait pas été possible si les champs `gauche` et `droit` n'était pas mutables.

```
let rec insere comp x = function
   Vide_p -> failwith "arbre_vide"
 | Noeud n ->
     if n.val=x then ()
     else if comp x n.val then
        match n.gauche with
          Vide_p -> n.gauche <- Noeud { val=x; gauche=Vide_p; droit=Vide_p }
        | Noeud ng -> insere comp x n.gauche
     else match n.droit with
           Vide_p -> n.droit <- Noeud { val=x; gauche=Vide_p; droit=Vide_p }
        | Noeud nd -> insere comp x n.droit ;;
```