TP4 : Arbres binaires de recherche

On utilisera le type suivant :

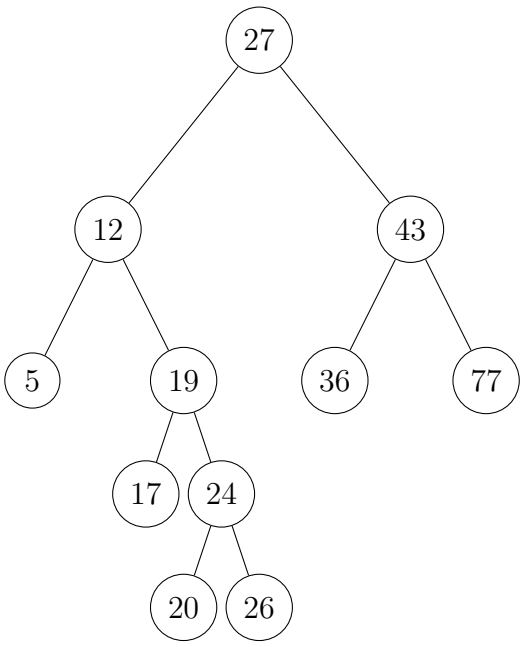
```
type arbre_bin =
  | Vide
  | Noeud of int*(arbre_bin)*(arbre_bin)
;;
```

Un arbre binaire non vide étiqueté est appelé arbre binaire de recherche si les étiquettes appartiennent à un ensemble totalement ordonné et s'il vérifie l'une des deux conditions équivalentes suivantes :

- La liste des étiquettes en ordre infixe est croissante au sens large.
- Pour tout nœud x d'étiquette e, les étiquettes des éléments de la branche gauche de x sont inférieures ou égales à e et les étiquettes des éléments de la branche droite de x sont supérieures ou égales à e.

La structure d'arbre binaire de recherche correspond à une réalisation concrète et persistante de la structure de donnée abstraite de dictionnaire. Il suffit pour cela d'étiqueter les nœuds avec les couples (clé,élément) et de faire en sorte que la structure d'arbre binaire de recherche soit adaptée à l'ordre sur les clés.

1. Définir l'arbre binaire de recherche suivant (noté `ex_1`) :



2. Pour comparer deux étiquettes, on utilise le type :

   ```
   type comparaison = Inf | Egal | Sup ;;
   ```

   Dans cet exemple, la fonction de comparaison sera la fonction de comparaison usuelle des entiers :

   ```
   let compare_int x y =
     if x<y then Inf else if x=y then Egal else Sup ;;
   ```

Selon nos besoins, nous pourrions cependant imaginer une fonction `compare` beaucoup plus loufoque mais qui renverrait également un élément de type `comparaison`.

(a) Écrire la fonction `recherche` qui, étant donné un arbre binaire de recherche `a`, la fonction de comparaison `compare` associée à l'arbre, et un élément x donné, vérifie si x appartient à l'arbre `a`.

```
recherche : arbre_bin -> (int -> int -> comparaison) -> int
    -> bool
```

(b) Justifier que la complexité de la fonction `recherche` est en $O(h)$ où h est la hauteur de l'arbre donné en entrée.

3. Écrire une fonction `min_gauche_max_droit` qui retourne l'étiquette du nœud le plus à gauche, ainsi que l'étiquette du nœud le plus à droite.

```
min_gauche_max_droit : arbre_bin -> int * int
```

La fonction renverra un message d'erreur si l'arbre est vide.

4. Écrire une fonction récursive `verifie_encadre` qui, étant donnés un arbre, une fonction de comparaison donnée, ainsi que deux entiers x et y tels que $x \leqslant y$, renvoie `true` si et seulement si l'arbre est bien un arbre binaire de recherche et que toutes ses étiquettes sont comprises entre x et y.

En déduire la fonction `verif_arb_rech` qui retourne vrai si l'arbre est un arbre de recherche, et faux sinon.

```
verif_arb_rech : arbre_bin -> (int -> int -> comparaison) -> bool
```

5. Une deuxième méthode pour vérifier si un arbre est un arbre de recherche consiste à donner la liste des étiquettes dans l'ordre infixe et à vérifier que la liste est triée.

Écrire une fonction `verif_tri` qui vérifie si une liste donnée est triée par une relation d'ordre donné.

```
verif_tri : 'a list -> ('a -> 'a -> comparaison) -> bool
```

En retrouvant la fonction infixe, en déduire la fonction `verif_arb_rech2`.

```
infixe :  arbre_bin -> int list
verif_arb_rech2 : arbre_bin -> (int -> int -> comparaison) -> bool
```

6. On désire insérer un nouvel élément dans un arbre binaire de recherche sans changer sa caractéristique d'arbre de recherche.

(a) Insertion aux feuilles :

Écrire une fonction `insere_feuille` qui place un nouvel élément sur une branche vide de l'arbre binaire de recherche ou renvoie cet arbre si l'élément était déjà présent. L'arbre ainsi crée doit conserver sa caractéristique d'arbre binaire de recherche.

```
insere_feuille : arbre_bin -> (int -> int -> comparaison) -> int
    -> arbre_bin
```

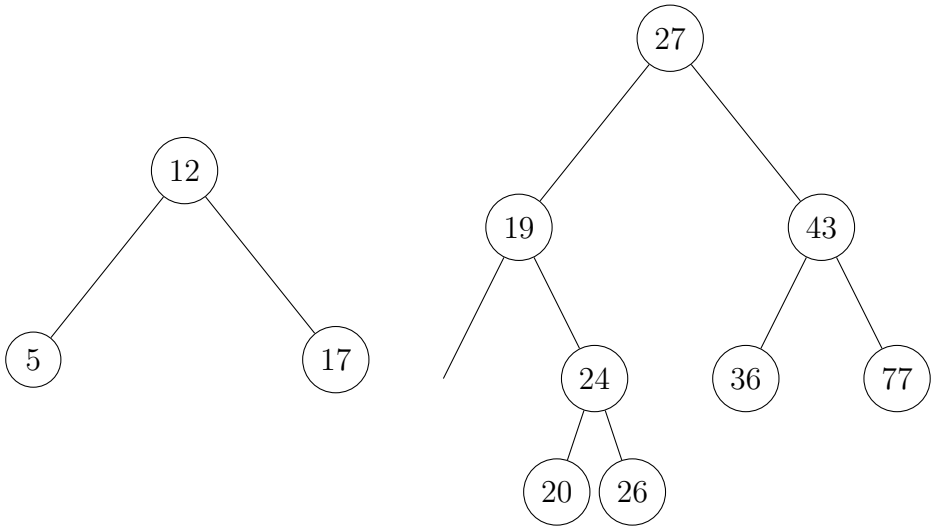
(b) Insertion à la racine :

Écrire une fonction `insere_racine` qui place un nouvel élément à la racine d'un arbre binaire de recherche tout en conservant sa caractéristique d'arbre binaire de recherche, ou renvoie cet arbre si l'élément était déjà présent.

L'idée est de découper l'arbre en deux, et de le recoller avec la nouvelle racine.

```
decoupe : arbre_bin -> (int -> int -> comparaison) -> int
    -> arbre_bin * arbre_bin
insere_racine : arbre_bin -> (int -> int -> comparaison) -> int
    -> arbre_bin
```

La fonction `decoupe`, appliquée à l'arbre utilisé en exemple pour une valeur 18, donne les deux arbres binaires de recherche suivants :



7. (a) Écrire une fonction `construction_feuille` qui utilise la fonction précédente `insere_feuille` pour construire un arbre de recherche composé des éléments d'une liste donnée.

   ```
   construction_feuille : int list -> (int -> int -> comparaison)
       -> arbre_bin
   ```

   (b) Écrire une fonction `construction_racine` qui utilise la fonction précédente `insere_racine` pour construire un arbre de recherche composé des éléments d'une liste donnée.

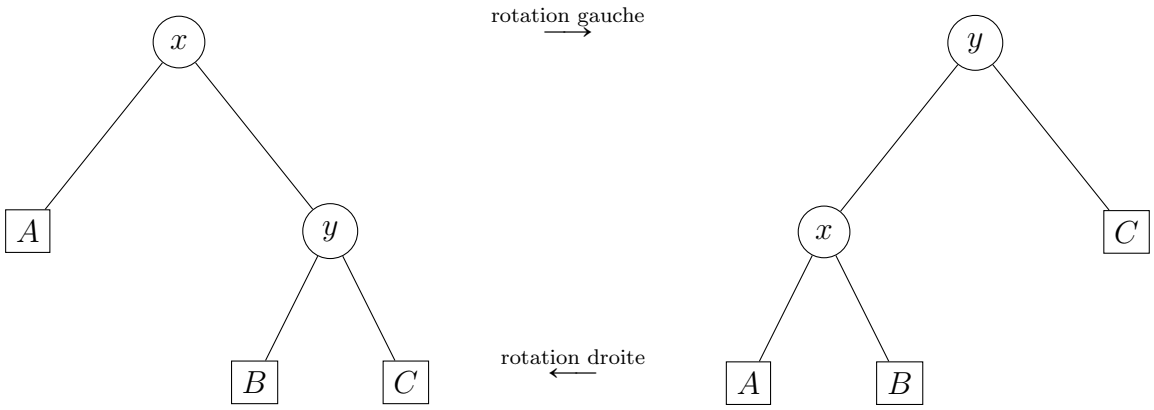
   ```
   construction_racine : int list -> (int -> int -> comparaison)
       -> arbre_bin
   ```

   (c) Comparer les arbres obtenus avec les listes :

   ```
   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];;
   ```

8. La complexité des fonctions précédentes dépend de h, la hauteur de l'arbre. On a donc tout intérêt à maintenir nos arbres binaires de recherche à une hauteur minimale lors des étapes d'insertion.

   Il peut ainsi être intéressant de rééquilibrer au fur et à mesure l'arbre avec les opérations de rotation expliquées par le schéma suivant :



Par exemple, si on insère un élément dans le sous-arbre C, il est raisonnable d'effectuer une rotation gauche.
Les opérations de rotation préservent la structure d'arbre binaire de recherche et sont de complexité $O(1)$.

Écrire une fonction `rotation_gauche` (respectivement `rotation_droite`) qui effectue l'opération rotation gauche (respectivement droite) d'un arbre binaire s'il a une forme adaptée à cette rotation et renvoie l'arbre non modifié sinon.

9. (a) Que fait la fonction `mystere` suivante ?

```
let rec mystere a1 a2 compare = match a1,a2 with
  | _,Vide -> a1
  | Vide,_ -> a2
  | Noeud(e1,ag1,ad1), Noeud(e2,ag2,ad2) when compare e1 e2 = Inf
      -> mystere ad1 (Noeud(e2,mystere (Noeud(e1,ag1,Vide))
            ag2 compare, ad2)) compare
  | Noeud(e1,ag1,ad1), Noeud(e2,ag2,ad2)
      -> mystere ag1 (Noeud(e2,ag2,mystere (Noeud(e1,Vide,ad1))
            ad2 compare)) compare
;;
```

   (b) Prouver la terminaison de la fonction `mystere`.
   En notant $|a|$ la taille (nombre de nœuds) de l'arbre a, on prouvera la stricte décroissance de $(|a1|, |a2|)$ selon l'ordre lexicographique sur $\mathbb{N}^2$.

   (c) Prouver la correction de la fonction `mystere`.