Arbres 2

Exercice 1 :

1. Écrire une fonction `meme_structure` prenant en argument deux arbres binaires et renvoyant `true` si les deux arbres ont la même structure et `false` sinon. On utilisera un type d'arbre adapté au problème.
2. Écrire une fonction `sous_arbre` prenant en argument deux arbres binaires et renvoyant `true` si le premier est un sous-arbre du second et `false` sinon.

Exercice 2 : Autour des arbres binaire de recherche
On considère des arbres binaire du type

```
type 'a ab = Vide | Noeud of 'a * 'a ab * 'a ab;;
```

1. Écrire une fonction `est_un_ABR` prenant en argument un arbre binaire et renvoyant `true` si l'arbre est un arbre binaire de recherche et `false` sinon.
   On fera en sorte de ne parcourir l'arbre qu'une seule fois.
2. Écrire une fonction `cherche` prenant en argument un élément et un arbre binaire de recherche et renvoyant `true` si l'élément est présent dans l'arbre et `false` sinon.
3. Écrire une fonction `minimum` prenant en argument un arbre binaire de recherche et renvoyant son minimum.
4. Écrire une fonction `keme` prenant en argument un entier k et un arbre binaire de recherche et renvoyant son k-ème plus petit élément.
5. Écrire une fonction `ajout` prenant en argument un élément et un arbre binaire de recherche et ajoutant l'élément en conservant une structure d'arbre binaire de recherche.
6. En déduire une fonction `tri` permettant de trier une liste.
7. Écrire une fonction `ajout_racine` prenant en argument un élément x et un arbre binaire de recherche T et renvoyant un arbre binaire de recherche ayant x comme racine et pour clé, x et celles de T.
8. (a) Écrire une fonction `extraire_max` prenant en argument un arbre binaire de recherche et renvoyant son maximum ainsi qu'un arbre binaire de recherche dont les clés sont celles du précédent hormis le maximum.
   (b) Écrire une fonction `supp_racine` prenant en argument un arbre binaire de recherche et renvoyant un arbre binaire de recherche dont les clés sont celles du précédent hormis la racine.
   (c) Écrire une fonction `supp` prenant en argument élément x et un arbre binaire de recherche et renvoyant un arbre binaire de recherche dont les clés sont celles du précédent hormis x.

Exercice 3 : Introduction aux files
Une façon d'effectuer le parcours en largeur d'un arbre consiste à créer une liste d'arbres. Cette liste contient initialement l'arbre dont on veut effectuer le parcours en largeur et elle évolue de la façon suivante jusqu'à ce qu'elle soit vide :

— on affiche la racine de l'arbre en tête de la liste ;
— son fils gauche puis son fils droit sont insérés à la fin de la liste d'arbres.

Le problème est que rajouter un élément en fin de liste est une opération linéaire en la taille de la liste. On va donc passer par une structure de file où l'on peut accéder directement au premier et au dernier élément. On se propose d'implémenter une file comme un couple de listes, la tête de la première correspondant au début de la file et la tête de la seconde correspondant au bout de la file.

1. Définir un type qui représente une file.
2. Écrire une fonction `creer_file_vide` qui créée une file vide, une fonction `est_vide` et une fonction `enfiler` qui prend en paramètre un élément x et une file et qui renvoie la file à laquelle on a rajouté x au bout. Quelle est la complexité de cette fonction ?
3. Écrire une fonction `defiler` qui renvoie la tête de la file et la file étêtée.
   Quelle est la complexité de cette fonction ?
4. Écrire une fonction `parcours_largeur` qui prend en paramètre un arbre binaire et qui affiche ses nœuds parcourus en largeur.