Durée : 2h.
Aucun document ni calculatrice autorisé.
Les 3 exercices sont indépendants.

Important : Lorsqu'une fonction est demandée, celle-ci doit être écrite en OCaml.

Quand l'énoncé demande de coder une fonction, il n'est pas nécessaire de justifier que celle-ci est correcte. On suppose que la fonction n'est utilisée que sur des entrées satisfaisant les préconditions de la spécification. Il est acceptable (et même normal, pour certaines questions) qu'une fonction telle qu'écrite ait un type plus large que celui demandé (par exemple, une fonction de type `'a -> 'a` quand on demande le type `int -> int`) tant qu'elle respecte la spécification.

Une grande importance sera apporté à la clarté et la lisibilité des programmes. Il est recommandé de choisir des noms de variables intelligibles. Il n'est en revanche pas nécessaire de réécrire la spécification d'une fonction si celle-ci est donnée dans l'énoncé.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractères diérentes désigne la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`).

On rappelle qu'en OCaml, les listes sont immuables. Aucune réponse utilisant des listes comme si elles étaient mutables ne recevra de points, même partiels.

Il en va de même des types structurés définis par union, si ceux-ci n'utilisent pas par ailleurs de types mutables.

Exercice 1 :

On définit le type suivant pour des arbres binaires d'entiers :

```
type arbre_bin = Vide | Noeud of arbre_bin * int * arbre_bin
```

On considère maintenant les arbres binaires de recherches (avec uniquement des clés entières, on ne se préoccuppe pas de leur associer des valeurs).

1. Rappeler la définition d'un arbre binaire de recherche (ABR).
2. Écrire une fonction `mini : arbre_bin -> int` qui étant donné un arbre **non vide** et dont on suppose qu'il **vérifie la définition d'un ABR**, calcule l'étiquette minimale de cet arbre.
   Si l'arbre passé en argument est vide, on déclenchera une erreur. Si l'arbre n'est pas un ABR, le comportement n'est pas défini.
   Votre fonction doit avoir une complexité linéaire en la hauteur de l'arbre **(on ne demande pas de le prouver)**.
3. On se donne la fonction `remove : arbre_bin -> int -> arbre_bin` qui étant donné a un ABR et x un entier, calcule un ABR contenant les mêmes entiers que a, sauf x.
   (a) Prouver que si x n'est pas une clé de a, ou bien que x est une clé de a et que l'unique nœud de a d'étiquette x a au plus un enfant, alors la complexité de `remove a x` est au plus linéaire en la hauteur de a.
   (b) Prouver que la complexité de `remove` est linéaire en la hauteur de l'arbre dans le pire cas

```
let rec remove a x =
  match a with
  | Vide -> Vide
  | Noeud (gauche, y, droit) ->
      if y < x then Noeud (gauche, y, remove droit k)
      else if x < y then Noeud (remove gauche k, y, droit)
      else (* y = x *) if gauche = Vide then droit
      else (* y = x *) if droit = Vide then gauche
      else (* y = x et les deux sous-arbres sont non vides *)
        let z = mini droit in
        Noeud (gauche, z, remove droit z)
```

Exercice 2 : **Extrait de E3A 2017**

Dans cet exercice, on considère des arbres binaires stricts non vides (ABSNV) dont les étiquettes des feuilles sont des entiers, et dont les nœuds internes ne sont pas étiquetés.

On définit ainsi le type :

```
type absnv = Feuille of int | Noeud of absnv * absnv
```

On définit ici un *peigne* comme un ABSNV dont tous les nœuds internes à l'exception éventuelle de la racine ont au moins une feuille pour fils. **Attention il ne s'agit pas de la définition usuelle, qui correspond ici aux peignes stricts.**

On dit qu'un peigne est *strict* si sa racine a également au moins une feuille pour fils, ou s'il est réduit à une feuille.

On dit qu'un peigne est *rangé* si le fils droit d'un nœud interne est toujours une feuille. Un arbre réduit à une feuille est un peigne rangé. Un peigne rangé est donc en particulier un peigne strict.

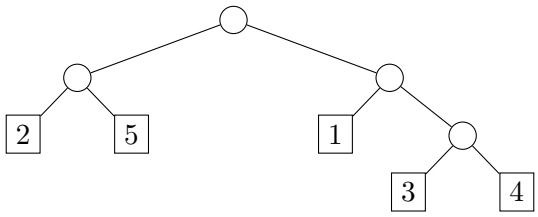

FIGURE 1 – Un peigne à cinq feuilles

1. Représenter un peigne rangé à 5 feuilles.
2. On rappelle que la hauteur d'un ABSNV réduit à une feuille est par convention 0. Quelle est la hauteur d'un peigne rangé à n feuilles ? On justifiera la réponse.
3. Écrire une fonction `est_peigne_range : absnv -> bool` qui indique si l'arbre donné en argument est un peigne rangé.
4. Écrire une fonction `est_peigne_strict : absnv -> bool` qui indique si l'arbre donné en argument est un peigne strict.

   En déduire une fonction `est_peigne : absnv -> bool` qui indique si l'arbre donné en argument est un peigne.

On souhaite ranger un peigne donné. Supposons que le fils droit N de sa racine ne soit pas une feuille. Notons A_1 le sous-arbre gauche de la racine, F l'une des feuilles du noeud N et A_2 l'autre sous-arbre du noeud N . On va utiliser l'opération de *rotation gauche* (illustrée sur la figure 2) qui construit un nouveau peigne où

- le fils droit de la racine est le sous-arbre A_2 ;
- le fils gauche de la racine est un noeud de sous-arbre gauche A_1 et de sous-arbre droit la feuille F .

5. Donner le résultat d'une rotation sur le peigne de la figure 1.
6. Montrer que le résultat d'une rotation sur un peigne dont le sous-arbre droit n'est pas une feuille est toujours un peigne.
7. Écrire une fonction `rotation : absnv -> absnv` qui, si son argument est un peigne dont le sous-arbre droit n'est pas une feuille, effectue l'opération décrite ci-dessus.

   La fonction renverra l'arbre passé en argument si une rotation n'est pas possible.

   Si l'arbre initial n'est pas un peigne, on ne définit pas le comportement de la fonction (il peut être une erreur, ou un résultat arbitraire).

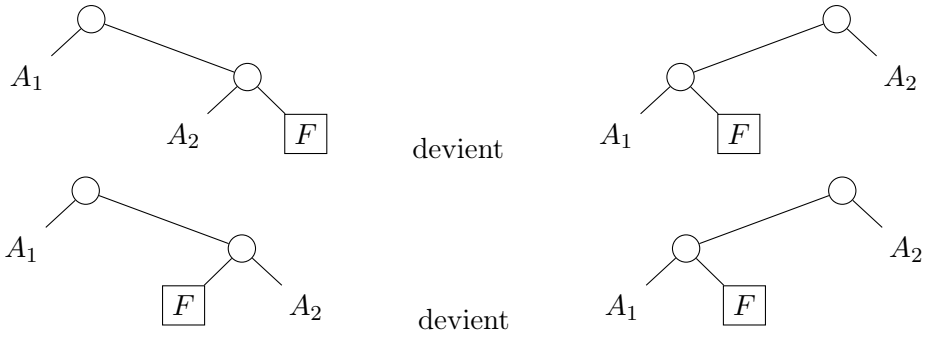

FIGURE 2 – Deux cas de rotation gauche

8. Écrire une fonction `rangement : absnv -> absnv` qui range un peigne donné en argument, c'est à dire qu'il renvoie un peigne rangé ayant les mêmes feuilles que celui donné en argument.
   Si l'arbre initial n'est pas un peigne, on ne définit pas le comportement de la fonction.
9. Prouver la correction de `rangement`.

Exercice 3 : **Extrait de E3A 2018**

Tout entier naturel non nul n s'écrit de manière unique $n = 2^v k$ avec v un entier naturel (qui peut valoir zéro) et k un entier naturel impair. Dans la suite de cet exercice, nous appelons valuation de n l'entier v et résidu de n l'entier k. Par convention, le résidu de 0 est 0.

1. Expliquer succintement comment, à partir de l'écriture binaire de $n \in \mathbb{N}^*$, on peut lire la valuation et le résidu de n.
2. L'entier 192 s'écrit en base deux 11000000. Donner sa valuation et son résidu.
3. Écrire une fonction `residu : int -> int` qui prend en argument un entier positif ou nul et renvoie son résidu.

Pour calculer le pgcd (plus grand commun diviseur) de deux entiers, nous pouvons utiliser l'algorithme des soustractions successives décrit ci-après.

```
Entrees : deux entiers naturels non nuls a et b
Sorties : pgcd(a, b)
Algo :
  tantQue b est non nul :
    Remplacer b par |a - b|
    Remplacer a par le minimum de a et de l'ancienne valeur de b
  finTantQue
  Renvoyer a
```

4. Écrire une fonction récursive `pgcd : int -> int -> int` qui calcule le pgcd de a et b entiers naturels non nuls en utilisant cet algorithme (et pas un autre) adapté au cadre récursif.
5. On s'intéresse à $C(n)$ la complexité dans le pire cas de cet algorithme en fonction de $n = \max(a, b)$.
   (a) Montrer que $C(n) = \mathcal{O}(n)$.
   (b) Montrer que dans le pire cas (que vous expliciterez en fonction de n), la complexité est effectivement linéaire.

La méthode chinoise, mentionnée dans *Les Neuf Chapitres sur l'art mathématique* écrit aux débuts de la dynastie Han, consiste à remplacer la ligne 3 par la ligne suivante :

```
    Remplacer b par le résidu de |a - b|
```

On admet que cette méthode permet de calculer le pgcd de a et b si a ou b est impair. Ce résultat peut notamment être utilisé pour répondre à la question suivante.

6. Que calcule la méthode chinoise lorsque a et b sont pairs ? *(Indication : On peut distinguer le cas $a = b$ du cas $a \neq b$.)*.
7. On s'intéresse à $C'(m)$ la complexité dans le pire cas de ce nouvel algorithme fonction de $m = a + b$, dans le cas où a et b sont tous les deux impairs. Montrer qu'il existe une constante K telle que pour tout $m \geqslant 4, C'(m) \leqslant \max\limits_{\substack{1 \leqslant k < \frac{3m}{4} \\ k \text{ pair}}} C'(k) + K$.

   (Indice : on distinguera le cas où $\min(a, b) \leqslant \dfrac{\max(a, b) - 1}{2}$ *et celui où* $\min(a, b) \geqslant \dfrac{\max(a, b) + 1}{2}$*.)*
8. En déduire que $C'(m) = \mathcal{O}(\log(m))$.
9. Écrire une fonction `pgcd_chinois : int -> int -> int` qui calcule le pgcd de deux entiers naturels non nuls (pairs ou impairs) avec une complexité $\mathcal{O}(log(n))$, où $n = \max(a, b)$.
   Justifier la complexité de `pgcd_chinois` .