Cette *dix-hutième* colle vous fera écrire des fonctions sur des arbres non forcément binaires en OCaml, et deux petites fonctions sur des tableaux 1D ou 2D en C.

On travaillera depuis la machine virtuelle ClefAgreg2019, et on compilera les fichiers écrits avant d'exécuter les binaires produits.

Ex.1 Arbres non forcément binaires - OCaml (45 minutes)

On considère dans cet exercice une structure d'arbre, non nécessairement binaire. Un tel arbre peut être soit réduit à l'arbre vide (représenté par le constructeur `Vide`), soit constitué d'une racine étiquetée (de type quelconque `'a`) et d'une liste ordonnée de sous-arbres (représenté par le constructeur `Noeud(etiquette, fils)` où `fils` est une `'a arbre list`).

1. Proposer un type récursif en OCaml permettant de représenter un tel arbre avec des étiquettes entières. Comment l'adapter pour avoir des étiquettes de type quelconque polymorphe `'a` ?

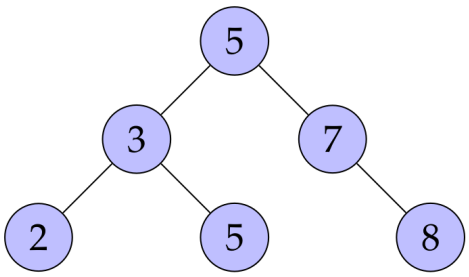

Figure 1 – Un exemple d'arbre A_1 qui se trouve être un arbre binaire

2. Implémenter en OCaml l'arbre donné dans la figure ci-dessus. Ici contrairement aux arbres binaires, il n'y a pas de notion de sous-arbres gauche et droite, donc l'étiquette 8 n'est pas vraiment située à droite de 7 c'est juste son unique fils.
3. Implémenter une fonction récursive `hauteur : 'a arbre -> int` qui renvoie la hauteur d'un arbre. On rappelle que par convention, l'arbre vide est de hauteur -1. Vérifier sur l'arbre exemple et l'arbre vide. Quelle est sa complexité en fonction de la hauteur h ou du nombre n de nœuds de l'arbre ?

On pourra introduire au préalable une fonction `max_liste : 'a list -> 'a` qui calcule la valeur maximum d'une liste non vide. On pourra penser à utiliser la fonction `List.map : ('a -> 'b) -> 'a list -> 'b list` de la bibliothèque standard.

4. Implémenter une fonction récursive `nombre_noeuds : 'a arbre -> int` qui renvoie le nombre de nœuds d'un arbre. On rappelle que par convention, l'arbre vide n'a aucun nœud. Vérifier sur l'arbre exemple et l'arbre vide. Quelle est sa complexité ?

On pourra au préalable introduire une fonction `somme_liste`, calculée récursivement ou avec un `List.fold_left` bien choisi (déjà vu).

5. Sans se restreindre à des arbres binaires de recherche, implémenter une fonction récursive `noeud_max : 'a arbre -> 'a` qui renvoie l'étiquette maximale d'un arbre étiqueté par des

étiquettes de type `'a`. On déclenchera une exception `failwith "Arbre vide"` sur l'arbre vide. On rappelle que la fonction `max : 'a -> 'a -> 'a` est polymorphe et fonctionne avec n'importe quel type. Vérifier sur l'arbre exemple et l'arbre vide. On pourra réutiliser la fonction `max_liste`.

Quelle est sa complexité ?

6. Implémenter une fonction `enraciner_sous_arbre_gauche : 'a arbre -> 'a arbre -> 'a arbre` telle qu'appelée sur `a1 a2` elle ajoute l'arbre `a2` comme le sous-arbre le plus *à gauche* des sous-arbres de l'arbre `a1`. On peut supposer que `a1` n'est pas vide.

Quelle est sa complexité en fonction de l'arité de la racine de `a1` (rappel : l'arité d'un nœud signifie son nombre de sous-arbres) ?

7. De même, implémenter une fonction `enraciner_sous_arbre_droit : 'a arbre -> 'a arbre -> 'a arbre` telle qu'appelée sur `a1 a2` elle ajoute l'arbre `a2` comme le sous-arbre le plus *à droite* des sous-arbres de l'arbre `a1`. On peut encore supposer que `a1` n'est pas vide.

Quelle est sa complexité en fonction de l'arité de la racine de `a1` ?

Ex.2 Deux petites fonctions en C (10 minutes)

On écrira un fichier C `colle18.c`, important `stdio.h` (pour `printf`) et `assert.h` (pour `assert`).

On rappelle qu'on compile ce fichier avec `COMPILATEUR = gcc` ou `clang` avec la ligne de commande suivante, puis on exécute le binaire produit avec la deuxième ligne (sans les dollars, qui représentent le prompt de la ligne de commande du terminal) :

```
$ COMPILATEUR -O0 -Wall -Wextra -Werror -fsanitize=address
  -fsanitize=undefined -pedantic -std=c11 -o colle18.exe colle18.c
$ ./colle18.exe
```

1. Écrire une fonction de prototype `void extrema(int t[], int taille, int* min, int* max)`, dont les préconditions sont :

— la longueur de `t` vaut `taille`, et elle est strictement positive ;

— `min` et `max` sont des pointeurs valides.

La fonction affectera le minimum de `t` à l'objet pointé par `min`, et le maximum à celui pointé par `max`. On s'imposera de ne faire qu'une seule passe de lecture du tableau.

Écrivez dans votre fonction `main` au moins deux exemples, que vous testerez avec des `assert`, et afficher leurs résultats.

2. Écrire une fonction de prototype `double somme_diagonale(int n, int m, double mat[n][m])`, dont les préconditions sont :

— `mat` est un tableau bidimensionnel de dimensions $n \times m$, avec $n, m \geqslant 1$ des entiers strictement positifs.

La fonction calculera la somme des valeurs situées sur la diagonale de la matrice (non nécessairement carrée, attention).

Quelle est la complexité temporelle de votre fonction exprimée en fonction de ses dimensions n et m ?

Écrivez dans votre fonction `main` au moins deux exemples, que vous testerez avec des `assert`, et afficher leurs résultats.

