Cette *dix-septième* colle vous fera écrire arbres et des parcours d'arbres en OCaml, et deux fonctions sur la recherche d'un mot dans une chaîne de caractères en C.

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

Ex.1 Arbres binaires - OCaml (15 minutes)

Un arbre binaire peut être soit réduit à l'arbre vide (représenté par le constructeur `Vide`), soit constitué d'une racine étiquetée (de type quelconque `'a`) et de deux sous-arbres gauche et droit (représenté par le constructeur `Noeud(gauche, etiquette, droit)`).

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

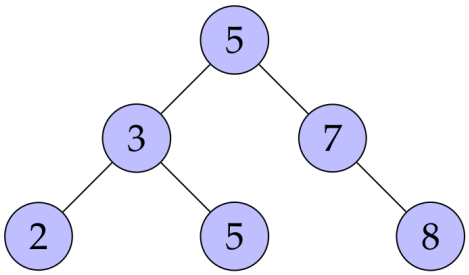

Figure 1 – Un exemple d'arbre binaire A_1

2. Implémenter en OCaml l'arbre donné dans la figure ci-dessus. Est-ce un arbre binaire de recherche au sens strict ?
3. Implémenter une fonction récursive `hauteur : 'a arbre -> int` qui renvoie la hauteur d'un arbre binaire. 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 de l'arbre ?
4. Implémenter une fonction récursive `nombre_noeud : 'a arbre -> int` qui renvoie le nombre de nœud d'un arbre binaire. 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 en fonction du nombre n de nœuds ?
5. Sans se restreindre à des arbres binaires de recherche, implémenter une fonction récursive `noeud_min : 'a arbre -> 'a` qui renvoie l'étiquette minimale d'un arbre binaire étiqueté par des étiquettes de type `'a`. On déclenchera une exception `failwith "Arbre vide"` sur l'arbre vide. On rappelle que la fonction `min : 'a -> 'a -> 'a` est polymorphe et fonctionne avec n'importe quel type. Vérifier sur l'arbre exemple et l'arbre vide. On pourra commencer par implémenter une fonction `min_trois : 'a -> 'a -> 'a -> 'a` telle que `min_trois x y z` soit le minimum entre `x`, `y` et `z`. Quelle est sa complexité en fonction du nombre n de nœuds et/ou de la hauteur h de l'arbre ?

Ex.2 Parcours d'arbres - OCaml (10 minutes)

1. Implémenter le parcours en profondeur préfixe en une fonction récursive de signature `parcours_profondeur_prefixe : 'a arbre -> ('a -> unit) -> unit` qui prend en argument un arbre binaire A étiqueté par des étiquettes de type `'a`, une fonction de traitement `'a -> unit` et applique cette fonction de traitement dans l'ordre du parcours en profondeur préfixe.
2. Quelle est sa complexité en fonction du nombre n de nœuds et/ou de la hauteur h de l'arbre ?
3. Faire de même pour le parcours en profondeur infixe, et post-fixe.
4. Tester sur l'arbre exemple et une fonction de traitement qui fait l'affichage de l'étiquette sur une ligne à part, comme `Printf.printf "%i\n"` de type `int -> unit` par exemple. Observer les différences des trois parcours.
5. (bonus à faire à la fin) En utilisant le module `Queue` qui donne accès à une structure de données de file, implémenter aussi le parcours en largeur en une fonction de signature `parcours_largeur : 'a arbre -> ('a -> unit) -> unit` qui prend en argument un arbre binaire A étiqueté par des étiquettes de type `'a`, une fonction de traitement `'a -> unit` et applique cette fonction de traitement dans l'ordre du parcours en largeur. Tester sur l'arbre exemple.

Ex.3 Recherche d'un mot dans une chaîne de caractères en C

On écrira un fichier C `colle17.c`, important `stdio.h` (pour `printf` et les fonctions sur les fichiers), `stdbool.h` (pour `true` et `false`) et `string.h` (pour `strlen`).

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 -Wvla -Werror -fsanitize=address
  -fsanitize=undefined -pedantic -std=c11 -o colle17.exe colle17.c
```

1. Écrire une fonction `bool occurrence (char* mot, char* texte, int i)` qui vérifie si un mot `mot` apparaît en position i d'un texte `texte`, c'est-à-dire si `mot = texte`$[i : i+|$`mot`$|[$.
2. Donner sa complexité en fonction de m la longueur de la (petite) chaîne `mot` et de n la longueur de la (grande) chaîne `texte`. Pourquoi pourrait-on penser que c'est indépendant de n ?
3. En utilisant la fonction `occurrence`, implémenter la fonction `bool recherche (char* mot, char* texte)` qui renvoie `true` s'il existe une occurrence de la sous-chaîne `mot` dans la chaîne `texte`, et `false` sinon.
4. Justifier la terminaison et la correction de cette fonction. Donner sa complexité en fonction de n et m.

