TP8 : Arbres, algorithmes sur les arbres, induction et linéarisation

Second semestre 2023/2024

1 Arbres binaires

Exercice 1. Propriétés sur les arbres

Montrer les propositions suivantes :

1. Dans un arbre binaire, il y a au moins une feuille.
2. Dans un arbre binaire de hauteur h, il y a au plus $2^h - 1$ nœuds internes.
3. Dans un arbre binaire de hauteur h, il y a au moins $\max(h-1, 0)$ nœuds internes.
4. Dans un arbre binaire localement complet à f feuille, il y a exactement $f - 1$ nœuds internes.
5. Dans un arbre à n nœuds, il y a exactement n sous-arbre.
6. Dans un arbre binaire de hauteur h, il y a au plus $2^{h+1} - 1$ nœuds.
7. Montrer qu'un arbre de taille n a au moins une hauteur de $\lceil \log_2(n+1) \rceil - 1$.

Exercice 2. Fonction basiques sur les arbres

On se donne le type suivant :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
```

Pour chacune des spécifications suivantes, proposer une fonction OCaml.

1. Une fonction de signature `'a arbre -> int` qui calcule la taille d'un arbre.
2. Une fonction de signature `'a arbre -> int` qui calcule la hauteur d'un arbre.
3. Une fonction de signature `int arbre -> int` qui calcule la somme des étiquettes d'un arbre.
4. Une fonction de signature `float arbre -> float` qui calcule le produit des étiquettes d'un arbre.
5. Une fonction de signature `'a arbre -> 'a -> bool` qui détermine si une étiquette est dans un arbre.
6. Une fonction de signature `'a arbre -> int` qui calcule la profondeur la plus petite parmi les feuilles d'un arbre.
7. Une fonction de signature `'a arbre -> int` qui compte le nombre de feuilles d'un arbre.

Exercice 3. Construction d'arbres

On se donne le type suivant :

```
type arbre = Vide | Noeud of arbre * arbre
```

Pour chacune des spécifications suivantes, proposer une fonction OCaml.

1. Une fonction jumeau de signature arbre -> arbre qui construit l'arbre dont les deux enfants sont l'arbre en entrée.
2. Une fonction complet de signature int -> arbre qui construit l'arbre complet de hauteur h.
3. Une fonction liane_gauche de signature int -> arbre qui construit l'arbre de profondeur h son entrée et dont chaque nœud interne a exactement un fils gauche.
4. Une fonction de signature int -> arbre qui construit l'arbre dont toutes les feuilles sont de profondeur h qui respecte les propriétés suivantes : la racine a deux enfants ; un fils gauche n'a qu'un fils gauche ou pas de fils du tout ; un fils droit a un fils gauche et un fils droit, ou pas de fils du tout.

Exercice 4. Égalité sur les arbres binaires

On se donne le type suivant :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
```

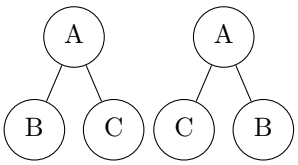
Implicitement, nous avions définie l'égalité entre deux arbres comme étant équivalente aux propriétés suivantes :

— Leurs racines ont la même étiquette ;
— Le premier a un enfant gauche si et seulement si le second a un enfant gauche, et si c'est le cas, ces deux enfants sont égaux ;
— Le premier a un enfant droit si et seulement si le second a un enfant droit, et si c'est le cas, ces deux enfants sont égaux.

1. Proposer une fonction de signature 'a arbre -> 'a arbre -> bool qui détermine si deux arbres sont égaux.
2. Quelle est la complexité de cette fonction ?

Désormais, nous aimerions affaiblir cette question d'égalité, on veut définir la relation suivante, deux arbres sont *semblables* si et seulement leurs racines ont la même étiquettes, le même nombre d'enfant, et que pour chacun de leurs enfants, il existe un enfant de l'autre racine qui soit semblable.

Ainsi, même s'ils ne sont pas égaux, les arbres suivants sont semblables :



3. Proposer une fonction de signature 'a arbre -> 'a arbre -> bool qui détermine si deux arbres sont semblables.
4. Quelle est la complexité de cette fonction ?
5. Montrer que la relation « être semblable à » définie une relation d'équivalence.

Exercice 5. Arbres en C : égalité structurelle, égalité mémoire

On se donne le type d'arbres suivant en C :

```c
typedef struct _arbre {
    int valeur;
    struct _arbre * fg;
    struct _arbre * fd;
} arbre_t;
```

Les champs `fg` et `fd` correspondent respectivement au pointeur vers le fils gauche et le fils droit, potentiellement égaux à `NULL` dans le cas où il n'y aurait pas de fils gauche ou de fils droit.

On précise qu'on distingue l'égalité structurelle (le fait de représenter le même objet mathématique) de l'égalité mémoire (le fait de correspondre au même objet dans la mémoire). En C, cette distinction se fait grâce aux pointeurs : soit on compare la valeur de l'objet pointée par le pointeur (pour l'égalité structurelle), soit on compare l'adresse dans les pointeurs (pour l'égalité mémoire).

1. Proposer une fonction de prototype `arbre_t * feuille(int etiquette)` qui crée un arbre sur le tas qui est composé d'une feuille d'étiquette `etiquette` et qui renvoie un pointeur vers cette étiquette.
2. Proposer une fonction de prototype `int taille (arbre_t * a_ptr)` qui calcule la taille d'un arbre passé en argument au travers d'un pointeur.
3. Proposer une fonction de prototype `int profondeur (arbre_t * a_ptr, arbre_t * b_ptr)` qui renvoie la profondeur de l'arbre `b_ptr` dans l'arbre `a_ptr`.

   On pourra supposer que l'arbre pointé par `b_ptr` est un sous-arbre de l'arbre pointé par `a_ptr`, en on vérifiera qu'il s'agit bien du même arbre en mémoire en vérifiant l'égalité des pointeurs.
4. Quelle est la complexité temporelle dans le meilleur et pire des cas en fonction de la taille n de l'arbre a, le premier arbre en entrée ?
5. On dit que deux arbres sont structurellement égaux si les arbres ont la même forme et qu'ils ont les mêmes étiquettes sans nécessité d'égalité physique.

   Proposer une définition inductive de l'égalité structurelle.
6. Proposer une fonction de prototype `bool egalite_structurelle (arbre_t * a_ptr, arbre_t * b_ptr)` qui vérifie si les deux arbres sont égaux struturellement.
7. Quelle est la complexité temporelle dans le meilleur des cas de cette fonction ?
8. Montrer que la complexité temporelle dans le pire des cas est en $O(\min(n, m))$ où n et m sont les tailles respectives de a et b, les deux arbres en entrée.
9. Proposer une fonction de prototype `bool est_sous_arbre(arbre_t * a_ptr, arbre_t * b_ptr)` qui vérifie si un arbre en entrée a a pour sous-arbre un arbre égaux structurellement à b.
10. Montrer que cette fonction a une complexité en $O(n \min(n, m))$ où n et m sont les tailles respectives de a et b, les deux arbres en entrée.

Exercice 6. Caractère mutable des arbres en C

On se donne le type d'arbre suivant en C :

```c
typedef struct _arbre {
    int valeur;
    struct _arbre * fg;
    struct _arbre * fd;
} arbre_t;
```

1. Proposer une fonction de prototype `arbre_t * jumeau(arbre_t * t, int racine)` qui à partir d'un arbre construit l'arbre (et l'alloue sur le tas) dont la racine est étiquettée par `racine` et dont les deux enfants sont `t`.
2. Proposer une fonction de prototype `void doubler_gauche(arbre_t * t)` qui multiplie par deux toutes les étiquettes dans le fils droit d'un arbre.
3. En utilisant ces deux fonctions, montrer la mutabilité des arbres en C. *On pourra se contenter d'exemples simples à afficher*

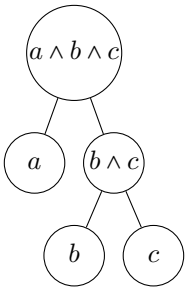
Exercice 7. Arbre XOR

On se donne le type d'arbre suivant :

```
typedef struct _arbre {
    int valeur;
    struct _arbre * fg;
    struct _arbre * fd;
} arbre_t;
```

Un arbre XOR est un arbre dont l'étiquette d'un nœud interne est le ou exclusif bit-à-bit de l'étiquette de ses enfants. (Si le nœud n'a qu'un enfant, son étiquette est égale à son enfant).

Par exemple, l'arbre suivant est un arbre XOR pour toutes les valeurs de a, b, et c :



1. Donner une fonction de prototype `bool est_xor(arbre_t * a)` qui renvoie si un arbre a est un arbre XOR.
2. Donner une fonction de prototype `void rendre_xor(arbre_t * a)` qui modifie les étiquettes des nœuds internes d'un arbre a de sorte à ce que a devienne un arbre XOR.
3. ($\star$) On se donne cette fois-ci la structure suivante :

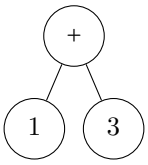
```
typedef struct _arbre {
    int valeur;
    bool est_fixe;
    struct _arbre * fg;
    struct _arbre * fd;
} arbre_t;
```

Proposer une fonction de prototype `bool rendre_xor_avec_contrainte(arbre_t * a)` qui modifie si possible les étiquettes des nœuds d'un arbre a de sorce à ce que a devienne un arbre XOR, mais qui ne peut pas modifier les étiquettes dont les nœuds ont la valeur `est_fixe` égale à `true`.

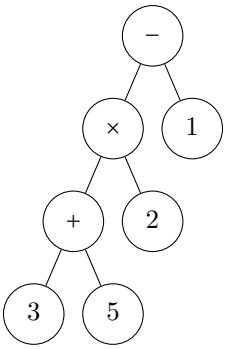
La fonction renverra si la transformation a été réussie ou non. On précisera la complexité du code obtenu.

Exercice 8. Arbre de syntaxe abstraite d'un calcul

On veut représenter des formules par des arbres. L'idée est la suivante, on représente une formule arithmétique par l'arbre des opérations que l'on doit réaliser. Par exemple, l'addition $1+3$ est représentée par l'arbre suivant :



On peut combiner plusieurs opérateurs, par exemple, pour la formule suivante $(3+5)\times 2-1$, on a l'arbre suivant :



On se donne le type suivant pour un arbre de syntaxe d'un calcul :

```
type asa = Valeur of int | Addition of asa * asa | Soustraction of asa * asa |
    Multiplication of asa * asa
```

1. Proposer une fonction de signature `asa -> int` qui évalue la valeur d'un arbre.
2. Proposer une fonction de signature `asa -> string` qui renvoie la formule associée à un arbre de syntaxe abstraite.

   On pourra afficher des parenthèses surnuméraires.
3. Proposer une fonction de signature `string -> asa` qui construie l'arbre de syntaxe à partir d'une formule en argument.

   On pourra supposer que toutes les opérations sont parenthésées.
4. La même chose, sans supposer que les opérations sont parenthésées partout, et en utilisant les règles de priorité usuelles.

2 Quelques variantes d'arbres

Exercice 9. Une autre construction des arbres binaires localement complets

On dit qu'un arbre binaire est localement complet quand tous ses nœuds internes ont deux enfants.
On se donne le type suivant :

```
type 'a complet = Feuille of 'a | Noeud of 'a * 'a complet * 'a complet
```

1. (a) Proposer une fonction de signature `'a complet -> int` qui calcule la taille d'un arbre complet localement.
   (b) Proposer une fonction de signature `'a complet -> int` qui calcule la hauteur d'un arbre complet localement.

On se donne le type suivant dans la suite :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
```

2. (a) Proposer une fonction de signature `'a arbre -> bool` qui détermine si un arbre est complet localement.
   (b) Proposer une fonction de signature `'a complet -> 'a arbre` qui convertit un arbre complet localement vers sa représentation en tant qu'arbre binaire générale.
   (c) Proposer une fonction de signature `'a arbre -> 'a complet option` qui convertit si possible un arbre en sa représentation en tant qu'arbre complet localement, et qui renvoie `None` sinon.
3. (a) Montrer qu'un arbre complet est nécessairement complet localement.
   (b) Montrer qu'un arbre est complet si et seulement si il est complet localement et que toutes ses feuilles ont la même profondeur.

(c) Proposer une fonction de signature 'a complet -> bool qui détermine si un arbre complet localement est complet.
(d) Proposer une fonction de signature 'a arbre -> bool qui détermine si un arbre est complet.

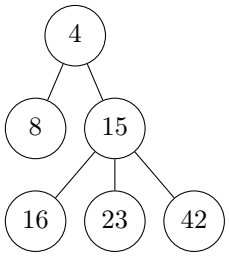
Exercice 10. Arbres d'arité arbitraire en OCaml

On propose le type d'arbres d'arité arbitraire suivant :

```
type 'a arbre =Noeud of  'a * ('a arbre list)
```

Un arbre est donc représenté par un couple composé de son étiquette et d'une liste de ses enfants.

1. Comment représenter l'arbre suivant dans ce type :



2. Que représente l'arbre suivant :

```
('c',
    [('e',[]);
    ('f', [
            ('g', [('a',[])]);
            ('s', []);
            ('i', [])
        ]
    )]
)
```

3. Proposer une fonction de signature 'a arbre -> 'a qui calcule la hauteur d'un arbre.
4. Proposer une fonction de signature int arbre -> int qui calcule la somme des étiquettes d'un arbre.
5. Proposer une fonction de signature 'a arbre -> int qui calcule l'arité maximum des nœuds de l'arbre.
6. On se donne le type suivant :

```
type 'a binaire = Vide | Noeud of 'a * 'a binaire * 'a binaire
```

(a) Proposer une fonction de signature 'a binaire -> 'a arbre qui convertit un arbre binaire vers sa représentation en tant qu'arbre d'arité quelconque.
(b) Proposer une fonction de signature 'a arbre ->'a binaire option qui convertit si possible un arbre vers sa représentation binaire, et None sinon.

Exercice 11. Arbres d'arité arbitraire en C avec un tableau d'enfants

Pour représenter des arbres d'arités arbitraire en C, on représente les enfants d'un nœud comme étant un tableau des pointeurs vers ses enfants.

```
struct _arbre{
    int val;
    int nb_enfants;
    arbre ** enfants;
};
typedef struct _arbre arbre_t;
```

On impose dans l'arbre que le tableau de pointeurs vers des arbres ne contient pas d'adresse NULL.

1. Proposer une fonction de signature `int taille(arbre_t * a_ptr)` qui renvoie la taille d'un arbre dont un pointeur a été passé en argument.
2. Proposer une fonction de signature `arbre_t * complet(int arite, int hauteur)` qui renvoie un arbre complet de hauteur `hauteur` dont chaque nœud interne est d'arité `arite`.
3. Proposer une fonction de signature `arbre_t * descendant_kieme(arbre_t * a_ptr, int k)` qui traverse l'arbre depuis sa racine en prenant à chaque fois l'enfant d'indice k jusqu'à arriver à un enfant qui a strictement moins de $k+1$ enfants.

   La fonction renverra un pointeur vers le nœud dans lequel cette fonction termine.
4. Proposer une fonction de signature `void effacer_aine(arbre_t * a)` qui modifie un arbre pour retirer l'ainé de chaque nœud interne.

Exercice 12. Arbres d'arité arbitraire en C avec une liste chaînée des enfants

On se donne les types définis de la manière suivante :

```c
struct _arbre;
typedef struct _enfants {
    struct _arbre * cible;
    struct _enfants * suivant;
} enfants_t;
typedef struct _arbre {
    int val ;
    struct _enfants * enfants;
} arbre_t;
```

Ainsi, un arbre est représenté par une structure qui contient sa valeur, et un pointeur vers la liste chaînée de ses enfants.

Mêmes questions qu'à l'exercice précédent.

Exercice 13. Arborescence de fichiers

On souhaite représenter une arboresce de fichiers à l'aide d'un arbre d'arité quelconque étiquetté par des chaînes de caractéres en C.

```c
typedef struct _arbre {
    nom char *;
    int nb_enfants;
    struct _arbre * parent;
    struct _arbre ** enfants;
} arbre_t;
```

Ici, on a choisi de ne représenter que les dossiers. Par ailleurs, on a rajouté un pointeur vers le parent. Par convention, on suppose que ce champs vaut `NULL` pour la racine.

Par ailleurs, on se donne une notion d'environnement qui se souvient de l'arbre total, ainsi que d'un pointeur vers le nœud actuel.

```c
typedef struct _env{
    arbre_t * racine;
    arbre_t * actuel;
} env_t;
```

1. Proposer une fonction de prototype `void ls(env_t e)` qui affiche les nombs des dossiers présents dans le répertoire courant.
2. Proposer une fonction de prototype `void mkdir(env_t *e_ptr, const char nom[])` qui crée un dossier de nom `nom` dans le répertoire courant.
3. Proposer une fonction de prototype `void cd_enfant(env_t *e_ptr, const char cible[])` qui modifie l'environnement pour se rendre dans l'enfant qui est supposé se trouver dans le répertoire courant.
4. Proposer une fonction de prototype `char * chemin_absolu(env_t *e_ptr)` qui affiche le chemin absolu depuis la racine vers le répertoire courant.

5. Proposer une fonction de prototype `void rm_enfant(env_t *e_ptr, const char cible[])` qui modifie l'environnement de sorte à pouvoir retirer un dossier supposé présent dans le répertoire courant.

   On rappelle que sans options, il n'est possible que d'effacer un dossier vide avec la commande rm.

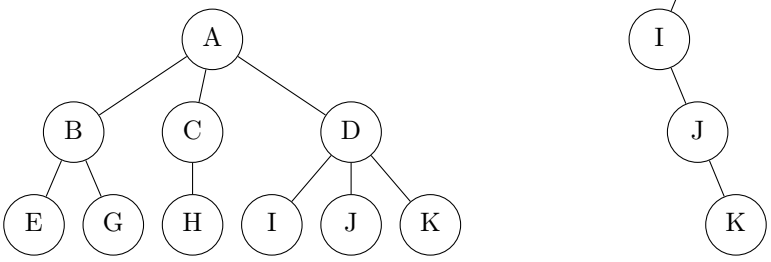
6. Proposer une fonction de prototype `void rm_enfant_recursif(env_t *e_ptr, const char cible[])` qui efface un enfant du répertoire courant, ainsi que tous les descendants de cet enfant.
7. Proposer une fonction de prototype `void cd(env_t *e_ptr, const char cible[])` qui modifie le répertoire courant à partir d'une destination précisée par `cible`.

   On prendra en compte la possibilité d'avoir un chemin qui soit absolu ou relatif, et qui comprenne plusieurs éléments.
8. Proposer une fonction de prototype `void rm_recursif(env_t *e_ptr, const char cible [])` qui efface un élément dans l'arbre, en prenant en compte la possiblité d'avoir un chemin absolu ou relatif, avec potentiellement un chemin en plusieurs étapes.

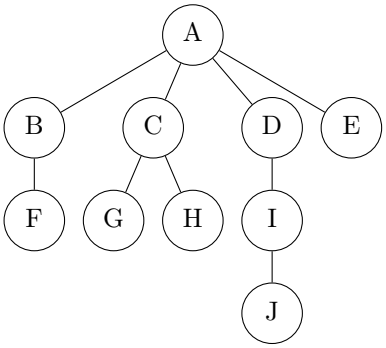
Exercice 14. Fils gauche frère droit

À partir d'un arbre d'arité quelconque on peut construire un arbre binaire de la manière suivante :
- Les nœuds sont les mêmes, avec les mêmes étiquettes ;
- l'enfant gauche d'un nœud reste l'éventuel premier enfant qu'il avait dans l'arbre d'origine ;
- l'enfant droit d'un nœud devient l'éventuel frère droit qu'il avait dans l'arbre d'origine.

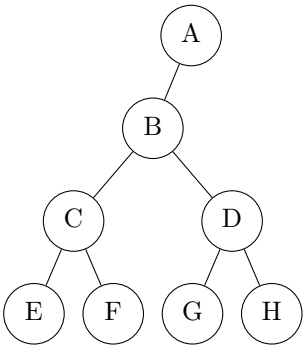
Dans l'exemple suivant, l'arbre de gauche est ainsi transformé en l'arbre de droite :



1. Pour l'exemple suivant, représenter l'arbre binaire suite à la transformation :



2. Trouver un arbre d'arité quelconque qui donne l'arbre suivant après transformation :



3. Donner un exemple d'arbre qui ne peut pas être obtenu par cette construction.

On se donne les types d'arbres suivants :

```
type binaire = Vide | Noeud of int * binaire * binaire
type arbre = Noeud of int * arbre list
```

Le premier type correspond à la forme des arbres binaires dont les étiquettes sont des entiers, le second correspond aux arbres d'arité quelconque étiquettée par des entiers.

4. Montrer que cette construction est une injection, c'est-à-dire que, pour un arbre binaire donné, il existe au plus un arbre qui après cette construction donne cet arbre binaire.
5. Proposer une fonction `vers_binaire` de signature `arbre -> binaire` qui construit l'arbre « fils gauche frère droit ».
6. Quels sont les arbres binaires qui ont un antécédent par cette construction ?
7. Proposer une fonction `vers_quelconque` de signature `binaire -> arbre option` qui construit si possible l'arbre quelconque qui a donné l'arbre binaire en entrée.

Exercice 15. Forêts d'arbres binaires

On se donne le type suivant pour une forêt :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
type 'a foret = 'a arbre list
```

Proposer des fonctions pour calculer la taille et la hauteur d'une forêt.

3 Parcours d'arbres binaires

Exercice 16. Parcours d'arbres binaires

On se donne le type suivant en OCaml :

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

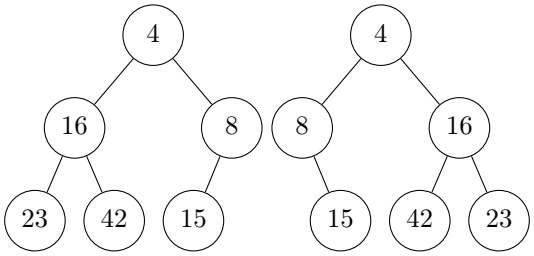
1. Proposer une fonction de signature `arbre -> int list` qui renvoie les étiquettes de l'arbre d'entrée dans l'ordre préfixe.
2. Proposer une fonction de signature `arbre -> int list` qui renvoie les étiquettes de l'arbre d'entrée dans l'ordre postfixe.
3. Proposer une fonction de signature `arbre -> int list` qui renvoie les étiquettes de l'arbre d'entrée dans l'ordre infixe.

4. Proposer une fonction de signature `arbre -> int list` qui renvoie les étiquettes de l'arbre d'entrée dans l'ordre par niveau.

Exercice 17. Relation entre les ordres

On dit qu'un arbre est le miroir d'un autre lorsqu'il a été retourné par rapport à un axe parrallèle à la profondeur de l'arbre.

Par exemple les deux arbres suivants sont le miroir l'un de l'autre.



Intuitivement, on remarque qu'il existe exactement un arbre miroir pour chaque arbre, et que le miroir du miroir est l'arbre lui-même.

On se donne le type OCaml suivant :

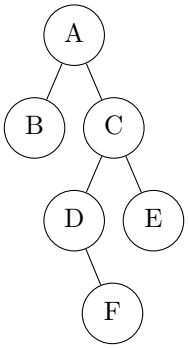
```
type arbre = Vide | Noeud of int * arbre * arbre
```

1. Proposer une définition inductive du miroir d'un arbre binaire.
2. En déduire une fonction de signature `arbre ->arbre` qui construit le miroir d'un arbre.
3. Montrer que l'ordre préfixe d'un arbre est l'inverse de l'ordre postfixe de son arbre miroir.

Exercice 18. Affichage d'arbres binaires

On cherche une méthode pour afficher un arbre binaire de manière plus lisible.

On propose un objectif à partir d'un exemple.



On cherche à ce que l'arbre précédent soit représenté de la manière suivante :

```
A
| - B
| - C
    | - D
        | - F
    | - E
```

On se donne par ailleurs le type suivant :

```
type arbre = Vide | Noeud of string * arbre * arbre
```

1. Dans quel ordre de parcours les nœuds sont affichés ligne par ligne ?
2. À quoi correspond le nombre d'indentations présentes devant le nom du nœud ?
3. En déduire une fonction de signature `arbre -> unit` qui affiche un arbre en entrée.

Exercice 19. Construction de l'arbre à partir du parcours

On se donne deux ensembles E et F d'intersection nulle, et on considère A l'ensemble des arbres localement complets dont les étiquettes des nœuds internes sont dans E et les étiquettes des feuilles sont dans F.

1. Montrer qu'il existe au plus un unique arbre $a \in A$ qui puisse avoir un ordre postfixe donné.
2. Montrer qu'il existe au plus un unique arbre $a \in A$ qui puisse avoir un ordre prefixe donné.
3. Montrer qu'avec un ordre infixe donné, il peut y avoir plusieurs arbres dans A qui ont cet ordre infixe.
4. On se donne le type suivant en OCaml :

```
type arbre = Feuille of int | Noeud of char * arbre * arbre
```

Les arbres sont donc localement complets, les nœuds internes sont étiquettés par des caractères et les feuilles sont étiquettées par des entiers. On se donne par ailleurs le type suivant :

```
type etiquette = I of int | C of char
```

(a) Proposer une fonction de signature `etiquette list -> arbre` qui construit l'arbre à partir de ses étiquettes dans l'ordre postfixe.
(b) Proposer une fonction de signature `etiquette list -> arbre` qui construit l'arbre à partir de ses étiquettes dans l'ordre préfixe.
(c) Proposer une fonction de signature `etiquette list -> arbre` qui construit un arbre à partir de ses étiquettes dans l'ordre infixe.

Exercice 20. Parcours par niveau d'un arbre

On se donne le type d'arbre suivant :

```
type arbre = Vide | Feuille int * arbre * arbre
```

Proposer une fonction de signature `arbre -> int -> bool` qui vérifie s'il existe un élément dans l'arbre en entrée avec l'étiquette en entrée en utilisant un parcours par niveau.

On pourra utiliser le module `Queue`.

4 Autres structures

4.1 Arbres binaires de Recherche

Exercice 21. Implémentation d'un arbre binaire de recherche en OCaml

On se donne le type d'arbre binaire de recherche suivant :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre * 'a arbre
```

On rappelle que la comparaison en OCaml est polymorphique.

1. Proposer une fonction de signature 'a arbre ->bool qui vérifie si un arbre correspond à un arbre binaire de recherche.
2. Proposer une fonction de signature 'a arbre -> 'a ->bool qui vérifie si un élément est dans un arbre binaire de recherche.
3. Proposer une fonction de signature 'a arbre -> 'a -> 'a arbre qui ajoute dans un arbre binaire de recherche.
4. Proposer une fonction de signature 'a arbre -> 'a -> 'a arbre qui ajoute un élément dans un arbre binaire de recherche s'il n'est pas déjà présent.
5. Proposer une fonction de signature 'a arbre -> 'a -> 'a arbre qui retire un élément d'étiquette donnée dans un arbre binaire de recherche.
6. Proposer une fonction de signature 'a arbre -> 'a -> 'a qui renvoie l'étiquette du prédecesseur du nœud d'étiquette précisée en argument.
7. Proposer une fonction de signature 'a arbre -> 'a -> 'a qui renvoie l'étiquette du successeur du nœud d'étiquette précisée en argument.

Exercice 22. Implémentation d'un arbre binaire de recherche en C

On se donne le type suivant :

```
struct arbre {
    int val;
    struct arbre * gauche;
    struct arbre * droit;
};
```

1. Proposer une fonction de prototype `bool verifier (struct arbre * a)` qui vérifie si un arbre correspond à un arbre binaire de recherche.
2. Proposer une fonction de prototype `struct arbre * recherche(struct arbre * a, int x)` qui vérifie si un élément est dans un arbre binaire de recherche et renvoie un pointeur vers le nœud concerné.
3. Proposer une fonction de prototype `void ajouter(struct arbre * a, int val)` qui ajoute dans un arbre binaire de recherche.
4. Proposer une fonction de prototype `void ajouter_unique(struct arbre * a, int val)` qui ajoute dans un arbre binaire de recherche un élément s'il n'est pas déjà présent.
5. Proposer une fonction de prototype `void retirer (struct arbre * a, int val)` qui retire un élément d'étiquette donnée dans un arbre binaire de recherche.
6. Proposer une fonction de prototype `struct arbre * predecesseur(struct arbre * a, int x)` qui renvoie un pointeur vers le prédecesseur du nœud d'étiquette précisée en argument.
7. Proposer une fonction de prototype `struct arbre * successeur(struct arbre * a, int x)` qui renvoie un pointeur vers le successeur du nœud d'étiquette précisée en argument.

Exercice 23. Pire des cas dans un arbre binaire de recherche

1. Pour quels arbre la complexité de recherche dans un arbre binaire de recherche correspond au pire des cas, c'est-à-dire à une complexité linéaire ?
2. Avec quels ajouts successifs on peut obtenir ce type d'arbre ?
3. Comment esquiver ce genre de problème dans la pratique ?

Exercice 24. Trouver le sucesseur et le prédecesseur en C

On se donne le type d'arbre suivant en C :

```c
struct arbre {
    int val;
    struct arbre * parent;
    struct arbre * gauche;
    struct arbre * droit;
};
```

Par convention, la racine a le pointeur NULL pour parent.

1. Proposer une fonction de prototype `struct arbre * successeur(struct arbre * n)` qui renvoie un pointeur vers le successeur d'un nœud dans un arbre.
2. Proposer une fonction de prototype `struct arbre * predecesseur(struct arbre * n)` qui renvoie un pointeur vers le prédécesseur d'un nœud dans un arbre.

Exercice 25. Ordre infixe dans le parcours d'un arbre binaire de recherche

Montrer que les étiquettes dans le parcours infixe d'un arbre binaire de recherche sont triées par ordre croissant.

Exercice 26. Nombre d'arbres binaires de taille n

On note C_n le nombre d'arbres binaires (ordonnés) de taille n sans étiquettes. Par convention, on pose $C_0 = 1$.

1. Combien vaut C_1, C_2 ?
2. Montrer que, pour tout $n \in \mathbb{N}$, $C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}$.
3. Montrer que C_n est le nombre d'arbres localement complets à $n+1$ feuilles.
4. ($\star$) Montrer que $C_n = \binom{2n}{n} - \binom{2n}{n+1}$.

Exercice 27. Profondeur moyenne d'un arbre binaire de recherche généré à partir d'une permutation

On considèrent les arbres aléatoires qui résultent de l'ajout des éléments de $\{1, ..., n\}$ dans un ordre aléatoire équiprobable. On note $h(x)$ la profondeur du nœud d'étiquette x, et on essaye d'en calculer l'espérance.

1. Montrer que $E(h(x)) = \sum_{y \neq x} P(\text{« y est un ancêtre de x »})$.
2. Montrer que y est un ancêtre de x si et seulement si y est le premier élément ajouté à l'arbre parmi les éléments entre y et x inclus.
3. En déduire que $E(h(x)) = O(\log n)$.
4. Illustrer ce résultat par l'expérience.

Exercice 28. Recherche et Ajout dans un arbre de recherche bicolore

On utilise le type suivant en C pour stocker un arbre rouge-noir :

```c
struct _arbreRN{
    struct _arbreRN * parent;
    struct _arbreRN * gauche;
    struct _arbreRN * droit;
    int val;
    bool est_noir;
};
typedef struct _arbreRN arbreRN_t;
```

Les feuilles ne seront pas représentés. Tous les pointeurs null pour les enfants feront figure de pointeur vers des feuilles supposées noires. Par ailleurs, la racine aura pour parent NULL.

1. Proposer une fonction de prototype `bool est_valide (arbreRN_t * arbre)` qui renvoie si un arbre de ce type vérifie toutes les propriétés d'un arbre rouge-noir. Dans la mesure du possible, cette fonction affichera dans la mesure du possible des informations sur la sortie standard de pourquoi l'arbre n'est pas valide.
2. Proposer une fonction de prototype `bool afficher (arbreRN_t * arbre)` qui affiche un arbre rouge noir en faisant figurer la couleur des nœuds d'une manière ou d'une autre.

   On pourra par exemple afficher les nœuds dans l'ordre préfixe en les indentants proportionnellement à leur profondeur en faisant figurer leur couleur.
3. Proposer une fonction de prototype `bool chercher(arbreRN_t * arbre, int n)` qui recherche dans un arbre si un élément est présent.
4. Proposer une fonction de prototype `void ajouter(arbreRN_t * arbre, int n)` qui ajoute un élément dans un arbre.
5. (⋆)Proposer une fonction de prototype `void retirer (arbreRN_t * arbre, int n)` qui retire un élément dans un arbre.

4.2 Table de hachages, ensembles, table d'association

Exercice 29. Table de hachage par chaînage en OCaml

Une table de hachage est une structure de données qui consiste à stocker des éléments dans des cases grâce à une fonction de hachage qui permet de savoir, à partir de l'objet à enregistrer, exactement dans quelle case ajouter ou chercher.

On utilise le type suivant pour représenter une table de hachage.

```
type 'a table = 'a list array
```

Chaque case d'indice k contient une liste des éléments dont les valeurs par la fonction de hachage vaut k dans un ordre qui n'a pas d'importance.

`[16; 8]`	`[]`	`[23]`	`[15; 4; 42]`

On utilisera la fonction de hachage `Hashtbl.hash` qui associe à un objet de n'importe quel type un entier, et on prendra le résultat modulo le nombre d'alvéoles.

1. Proposer une fonction de signature `'a table -> 'a -> unit` qui modifie une table de hachage pour ajouter un élément passé en argument. Quelle est la complexité de l'ajout ?
2. Proposer une fonction de signature `'a table -> 'a -> bool` qui renvoie si un élément est présent dans une table de hachage. Quelle est la complexité de la recherche en fonction de n le nombre d'élément dans la table de hachage, et de N le nombre d'alvéoles dans le meilleur, pire des cas, et cas moyen ?
3. Proposer une fonction de signature `'a table -> 'a -> unit`
4. Comment faudrait-il modifier le type de table de hachage pour pouvoir la redimensionner de manière mutable ? Proposer une fonction de redimensionnement avec ce type-là.

5. Proposer une implémentation d'une fonction ajouter qui modifie la taille de la table de hachage si la table contient trop d'éléments.
6. Montrer que, pour des redimensionnements raisonnables, la complexité amortie de l'ajouts d'éléments se fait en $O(1)$.

Exercice 30. Table de hachage par chaînage en C

Proposer un type pour implémenter les questions de l'exercice précédent en C, et implémenter les fonctions.

Exercice 31. Table de hachage par adressage ouvert en C

L' adressage ouvert pour la résolution de conflit d'une table de hachage qui résout le cas de conflits (cas où plusieurs éléments ont la même valeur par la fonction de hachage) en, dans le cas où la case attendue est déjà occupée de sonder d'autres cases dans la table de manière déterministe jusqu'à trouver une case disponible.

Pour la recherche, il est donc nécessaire pas seulement de regarder à la case attendue, mais de répéter le protocole de sondage.

On se donne une constante pour le nombre de cases dans une table de hachage.

```c
const int taille_table = 256;
```

On se donne un type pour le contenu d'une cellule qui contient un booléen pour savoir si la case est occupée, et l'élément à stocker sous forme d'un entier.

```c
typedef struct _alveole{
    bool libre;
    int val;
} alveole_t;
```

Une table de hachage est donc un tableau de taille `taille_table` qui contient des alvéoles :

```c
typedef alveole_t * table_t;
```

Pour fonction de hachage, nous utiliserons dans un premier temps la fonction qui calcule la valeur de l'entrée modulo `taille_table`.

1. Proposer une fonction de prototype `table_t initialiser (void)` qui initialise une table de hachage sur le tas.
2. Proposer une foction de prototype `int nombre_case_libre(table_t t)` qui calcule le nombre de cases libres dans une table de hachage par adressage ouvert.
3. Proposer une fonction de prototype `void ajouter(table_t t, int val)` qui ajoute un élément dans une table de hachage par adressage ouvert.
4. Proposer une fonction de prototype `bool recherche(table_t t, int val)` qui vérifie si un élément est présent dans une table de hachage par adressage ouvert.
5. (a) Quelle est la complexité dans le meilleur et pire des cas des fonctions d'ajout et de recherche en fonction de k le nombre de cellules occupées dans la table ?
   (b) On suppose que la table est de taille infinie et que la probabilité qu'une cellule a une probabilité $p < 1$ d'être occupée de manière indépendante des autres.
   Quelle est la complexité moyenne de l'ajout et de la recherche dans la table ?
   (c) (⋆) On note k le nombre de cellules occupées dans la table, et N le nombre de cases totales. On suppose $k < N$, et on suppose que les k cases sont réparties de manière équiprobables dans
   Quelle est la complexité moyenne de l'ajout et de la recherche dans la table ?
   (d) Que se passe-t-il quand $k = N$?

6. Souvent, quand les éléments que l'on veut stocker dans une table de hachage par adressage ouvert sont grands, on préfère stocker un pointeur vers un l'objet dans la table, et on utilise le pointeur NULL dans le cas où la case est libre.

   On se donne un type `struct element` qui sont les valeurs à stocker.

   Comment adapter les types et fonctions précédentes pour pouvoir prendre en compte cette modification ?
7. Comment traiter le redimensionnement dans une table de hachage par adressage ouvert ?
8. Quels sont les soucis liés au retrait dans une table de hachage ouvert ?

Exercice 32. Table de hachage par adressage ouvert en OCaml

Proposer un type pour implémenter les questions de l'exercice précédent en OCaml, et implémenter les fonctions.

Exercice 33. Ensemble à partir d'une table de hachage

À partir d'une table de hachage du module Hashtbl, proposer une implémentation d'un ensemble en C.

Proposer des fonctions pour l'ajout, le retrait et la recherche d'un élément, ainsi que pour l'union, l'intersection, et la différence ensembliste.

Exercice 34. Dictionaire à partir d'une table de hachage en OCaml

À partir d'une table de hachage, proposer une implémentation d'une table d'association en OCaml. On précisera la table de hachage utilisée, et la fonction de hachage utilisée qui pourra ne prendre en compte qu'une partie des pairs clef-valeur.

Exercice 35. Dictionnaire à partir d'une table de hachage en C

Même chose qu'à l'exercice précédent en C.

Exercice 36. Utilisation d'une table d'association pour la gestion de contexte

Nous aimerions avoir une table d'association pour gérer un contexte, c'est-a-dire une table qui nous permet de savoir pour chaque nom de variable quelle est la valeur dans cette variable.

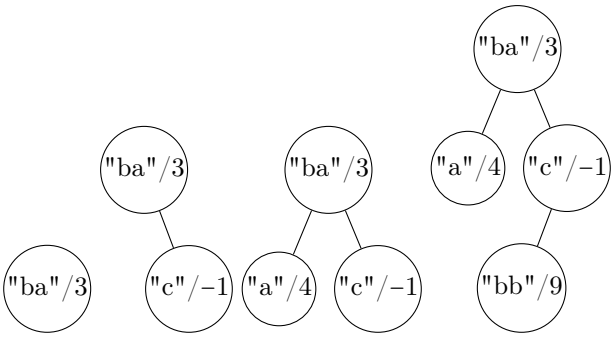
On propose d'implémenter une table d'association à l'aide d'un arbre binaire de recherche du type suivant en OCaml :

```
type table = Vide | Feuille of string * int * table * table
```

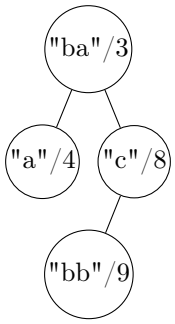
Dans chaque nœud, la première valeur du 4-uplet est une chaîne de caractère qui nous sers de clef, c'est-à-dire la valeur unique par rapport à laquelle on compare pour l'ajout et la second est l'entier que l'on associe à cette valeur.

On utilise l'ordre lexicographique sur les chaînes de caractère, c'est-à-dire l'ordre du dictionnaire usuel.

Par exemple si on pars d'une chaîne de caractère `"ba"` à laquelle on associe la valeur 3, puis qu'on ajoute successivement -1 associé à `"c"`, 4 associé à `"a"`, 9 associé à `"bb"`, on obtient successivement :



Si on essaye d'ajouter une valeur avec une clef qui est déjà présente, on modifie seulement la valeur dnans le nœud concerné, par exemple, si o n essaye d'ajouter 8 associé à c, on obtient :



1. Proposer une fonction de signature recherche `table -> string -> int option` qui renvoie `Some x` où x est la valeur dans la table associée à la chaîne en entrée si possible, et `None` sinon.
2. Proposer une fonction de signature modifier `table -> string -> int -> table` qui envoie la nouvelle valeur de la table après modification pour ajouter une valeur indexé par la chaîne de caractère si la clef n'était pas déjà présente, ou modifier la valeur associée à une clef si celle-ci est déjà présente.

   On prendra soin de limiter le nombre de parcours de l'arbre.

Exercice 37. Arbre de syntaxe en OCaml

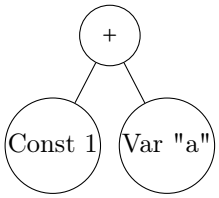
On cherche à représenter des arbres de syntaxes qui nous permettent de traiter un langage abstrait minimaliste.

1. Dans un premier temps, on s'intéresse aux epressions qu'on représente avec le type suivant :

```
type expression =
    | Somme of expression * expression
    | Difference of expression * expression
    | Produit of expression * expression
    | Const of int
    | Var of string
```

   Ainsi, une expression est la somme, la différence ou le produit de deux expressions, une constant entière, ou bien une variable caractérisée par son nom. On suppose qu'on dispose d'une table d'association qui contient les valeurs pour chaque nom de variable dont le type et les opérations sont décrits dans l'exercice précédent.

   Ainsi, la valeur de l'expression représenté par l'arbre suivant va dépendre de notre table d'association. Si `a` vaut 1, la valeur totale sera 2, tandis que si `a` vaut 4, la valeur totale sera 5 :



Proposer une fonction de signature `expression -> table -> int` qui à partir d'une expression et d'un contexte sous la forme d'une table d'association renvoie la valeur de l'expression dans ce contexte.

2. On décide de représenter une instruction par le type suivant :

```
type instruction =
    | Affectation of string * expression
    | Entree of string
    | Sortie of string
```

Ainsi, une instruction est : soit une affectation qui modifie une variable dans le contexte par une expression ; soit une demande d'entrée qui fait un appel à `read_line` pour demander la saisie de la valeur d'une variable dont l'identifiant est précisé ; soit une sortie qui affiche sur la sortie standard la valeur d'une variable dont le nom est précisé.

Proposer une fonction de signature `instruction -> table -> table` qui à partir d'une instruction et d'une table qui correspond à un contexte, exécute cette instruction, et renvoie le contexte éventuellement modifié.

3. On décrit désormais un programme comme une suite d'instruction.

```
type programme = instruction list
```

Proposer une fonction de signature `programme -> unit` qui exécute un programme en partant d'un contexte vide.

4.3 Tas

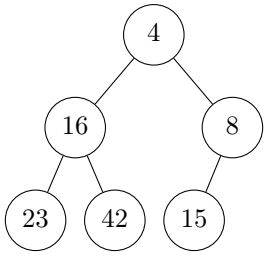
Exercice 38. Tas en OCaml en utilisant un tableau dynamique

On se donne le type suivant en OCaml :

```
type tas = int array ref
```

L'idée est d'implémenter un tas d'entier avec une référence de tableau d'entier au lieu de passer par un tableau dynamique. On utilise la première case du tableau pour se souvenir du nombre d'éléments qui sont dans le tas.

Par exemple, on peut considérer le tas suivant :



On le représente en mémoire par le tableau suivant :

6	4	16	8	23	42	15	#

Si on ajoute un élément (par exemple 2), il faut à la fois à nouveau conserver la structure de tas, mais aussi modifier le premier élément du tableau pour incrémenter le nombre d'élément. On obtient alors :

7	2	16	4	23	42	15	8

Enfin, si on rajoute un élément, on doit changer la taille du tableau comme pour un tableau dynamique classique et le tableau devient alors :

7	2	16	4	23	42	15	8	108	#	#	#	#	#	#	#

1. Quel est l'indice dans le tableau du parent du nœud dont l'étiquette est stockée à l'indice i ? Son fils gauche ? Son fils droit ?
2. Proposer une fonction de signature `tas -> int -> unit` qui ajoute un élément dans un tas.
3. Proposer une fonction de signature `tas -> int` qui retire un élément dans un tas.

Exercice 39. Tas en C

Proposer une implémentation d'un tas en C de manière similaire à l'exercice précédent. On pourra utiliser un tableau équipé de sa taille en mémoire.

Exercice 40. Implémentation du tri par tas en OCaml

Proposer une fonction de signature `'a array -> unit` qui réalise le tri par tas d'un tableau.

Exercice 41. Tris par tas en C

Proposer une fonction de prototype `void tri_par_tas(int * tableau, int taille )` qui réalise le tri par tas d'un tableau en C.

5 Ordre bien fondé, Induction

Exercice 42. Une récurrence est une induction sur $\mathbb{N}$

1. Proposer une construction sans ambiguïté de $\mathbb{N}$ par induction.
2. Montrer qu'une induction sur $\mathbb{N}$ est une récurrence.

Exercice 43. Induction sur les listes chaînées

1. Proposer une construction sans ambiguïté de l'ensemble des listes dont les éléments sont dans $\mathbb{N}$.
2. Proposer une définition par induction de la somme d'une liste.

Exercice 44. Ordre produit

Soit $(E, \leqslant_E)$ et $(F, \leqslant_F)$ deux ensembles ordonnés.

L'ordre produit $\leqslant$ de $\leqslant_E$ et $\leqslant_F$ est l'ordre sur $E \times F$ de sorte à ce que, pour tout $e, e' \in E$ et $f, f' \in F$, on a :

$$(e, f) \leqslant (e', f') \Leftrightarrow e \leqslant_E e' \text{ et } f \leqslant_F f'$$

1. Montrer que l'ordre produit définie bien une relation d'ordre.
2. Montrer que $\leqslant$ est bien fondé si et seulement si $\leqslant_E$ et $\leqslant_F$ sont bien fondés.
3. Proposer une généralisation au produit de n ordres.

Exercice 45. Ordre lexicographique

Soit $(E, \leqslant_E)$ et $(F, \leqslant_F)$ deux ensembles ordonnés.

L'ordre lexicographique $\leqslant$ de $\leqslant_E$ et $\leqslant_F$ est l'ordre sur $E \times F$ de sorte à ce que, pour tout $e, e' \in E$ et $f, f' \in F$, on a :

$$(e, f) \leqslant (e', f') \Leftrightarrow e \leqslant_E e' \text{ ou } (e = e' \text{ et } f \leqslant_F f')$$

1. Montrer que l'ordre lexicographique définie bien une relation d'ordre.
2. Montrer que $\leqslant$ est bien fondé si et seulement si $\leqslant_E$ et $\leqslant_F$ sont bien fondés.
3. Proposer une généralisation à l'ordre lexicographique sur n ordres.