Sujet n°6

Algorithmique et Structure de Données

Les arbres

`Temps de réalisation: 3h00mn`

Les objectifs du TP sont :
- définir et manipuler des structures d'arbres
- exploiter des arbres binaires de recherche
- mobiliser ces structures pour la résolution de problèmes

Échelle de progression :

● « débutant·e » ● « confirmé·e » ● « avancé·e » ● « expert·e »

Ex.	01	02	03	04	05	06
	●	●	●	●	●	●

Pour acquérir un niveau de compétence donné, il faut achever **tous** les exercices de ce niveau et des niveaux inférieurs. L'objectif est d'atteindre le niveau *confirmé·e* (●) à l'issue du TP.

Table des matières

Exercice 1: Les arbres binaires

QUESTION 1. Proposer une structure de données `BinaryTree` pour représenter par « liaison explicite » un type abstrait *arbre binaire* dont les clés sont des nombres entiers. Les nœuds de l'arbre sont de type `Node`.

Remarque: *L'usage d'un nœud fictif que l'on nomme* terminal*, vers lequel pointe chaque feuille de l'arbre, présente des avantages pour les algorithmes de parcours d'arbre. Il remplace alors la valeur* `None`*. Attention toutefois, ce nœud dit terminal est* commun à tous les arbres *de la classe* `BinaryTree` *et se déclare soit de manière globale, soit dans la classe* `BinaryTree` *à l'aide de l'annotation de type* `ClassVar` *du module* `typing` [1].

1. Dans le langage de la programmation par objets, il s'agit d'une variable de classe.

Astuce: *Pour faciliter l'appréhension de la notion de nœud terminal, il est recommandé de la représenter sur l'arbre binaire de la Figure 1.*

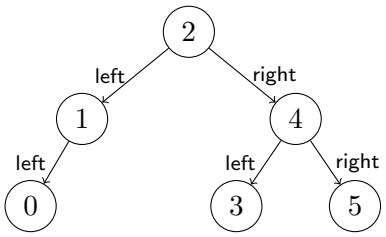

FIGURE 1 – Exemple d'arbre binaire construit à partir de la séquence $(2, 1, 4, 0, \perp, 3, 5)$.

QUESTION 2. Écrire les premières fonctions :

```
def isEmpty() -> bool
def root() -> Node
def terminal() -> Node
```

Un arbre vide est un arbre sans racine. La fonction `terminal` retourne le nœud terminal de l'arbre ou à défaut, `None`.

QUESTION 3. Proposer un constructeur d'arbres binaires de signature :

```
def __init__(nodes: list[int | None] | None = None) -> BinaryTree
```

Si `nodes` vaut `None` ou `[]`, alors le constructeur fournit un arbre binaire vide.

Le paramètre `nodes` est un tableau de valeurs entières qui encode l'arbre dans son intégralité. Les nœuds sont énumérés par niveau, de gauche à droite, de telle sorte que pour tout nœud d'indice i, ses fils gauche et droit se trouvent resp. aux indices $2i+1$ et $2i+2$ dans le tableau. Par exemple, l'arbre binaire de la Figure 1 est encodé par le tableau :

```
>>> t = [2, 1, 4, 0, None, 3, 5]
```

On observe que les fils gauche et droit de $t[1]$=1 par exemple, sont respectivement $t[3]$=0 et $t[4]$=`None`, suivant le motif des indices $2 \times 1 + 1$ et $2 \times 1 + 2$. De la même manière, les fils de $t[2]$=4 sont $t[5]$=3 et $t[6]$=5. Notez que les fils de $t[3]$=0 sont théoriquement $t[7]$ et $t[8]$ qui n'existent pas : 0 est donc une feuille. De façon similaire, tous les fils de `None` sont soit hors tableau, soit `None` eux-mêmes.

Remarque: *L'encodage sous la forme du tableau* ***`nodes`*** *est en fait une autre implémentation possible pour les arbres binaires (sic !).*

Astuce: *La conversion du tableau en une structure hiérarchique de* `Node` *s'opère de façon simple par récursivité sur les nœuds. Il s'agit, pour tout indice i du tableau, d'observer que le sous-arbre gauche est construit récursivement à l'indice $2i+1$, et le sous-arbre droit de la même manière à l'indice $2i+2$. Ensuite, ne reste plus qu'à créer le nœud courant à l'aide de la valeur à l'indice i et des deux sous-arbres.*

QUESTION 4. Écrire les fonctions suivantes, par récursivité sur les nœuds :

```
def height() -> int   # calcule la hauteur
def size() -> int     # calcule le nombre de noeuds
```

La hauteur d'un arbre est la taille du plus long chemin de la racine vers une feuille. La hauteur d'un arbre vide est, par convention égale à -1.

Exercice 2: Fonctions avancées sur les arbres binaires

Question 1. Écrire un test d'égalité de 2 arbres binaires :

```
def isEqual(bt2: BinaryTree) -> bool
```

Le test retourne *vrai* si les arbres $\texttt{bt}_1$ et $\texttt{bt}_2$ ont la même structure et les mêmes valeurs dans chaque nœud.

Remarque: *Le test d'égalité ci-dessus réalise une comparaison des 2 objets « en profondeur », par opposition aux tests d'égalité superficielle* `bt1 is bt2` *et par défaut* `bt1 == bt2`.

Question 2. Écrire la fonction de test suivante :

```
def isHeap() -> bool
```

qui vérifie que pour tout nœud de l'arbre, la valeur est supérieure ou égale à la valeur de chacun de ses fils non vides. On a alors à faire à une structure de tas (*heap* en anglais). La fonction renvoie *vrai* si cette propriété est satisfaite et *faux* sinon.

Question 3. Écrire la fonction

```
def lca(a: int, b: int) -> int
```

qui calcule le plus petit ancêtre commun [2] (*Lowest Common Ancestor* ou LCA en anglais) des clés a et b présentes dans l'arbre `bt`. L'hypothèse supplémentaire est que chaque nœud porte une valeur entière unique. Cela garantit que a et b désignent précisément deux nœuds de l'arbre.

Astuce: *Une manière élégante de mener ce calcul de LCA passe par la recherche de a et de b dans chacun des sous-arbres d'un nœud x donné, en partant de la racine ; x est alors le LCA de a et b ssi les deux clés ne se trouvent pas dans le même sous-arbre !*

Remarque: *Une manière plus évidente mais moins efficace consiste à retenir les chemins de la racine à a et à b, puis éliminer les préfixes communs.*

Question 4. Écrire la fonction de représentation sous forme de chaîne de caractères d'un arbre binaire :

```
def str() -> str
```

de telle sorte qu'elle réalise un « joli affichage » (*pretty print*) d'un arbre binaire, proche de l'exemple suivant :

```
>>> print(tree.str())
    __(2)____
  _(1)     _(4)__
(0)      (3)   (5)
```

2. Célèbre problème qui fut énoncé la première fois par Aho, Hopcroft et Ullman en 1973.

Astuce: *La fonction doit être capable de parcourir l'arbre en largeur d'abord (*breadth-first search*), autrement dit niveau par niveau. Une fois que ce parcours est en place, il reste à transmettre certains paramètres de positionnement d'un appel récursif à l'autre pour formater chaque ligne—représentant un niveau—de l'arbre en fonction de la situation « d'en dessous ».*

Exercice 3: Les expressions arithmétiques

On considère des arbres qui représentent des expressions arithmétiques. On appelle cette structure un *arbre de syntaxe abstraite* (ASA ou *Abstract Syntax Tree*—AST—en anglais). Les valeurs des nœuds intermédiaires sont des opérateurs parmi $\{+, -, \times, \div\}$, et celles des feuilles sont des nombres. Par exemple, l'expression arithmétique préfixée

`(+ (- 4 2) (* 3 5))` ou sans parenthèses : `+ - 4 2 * 3 5`

sera représentée par l'arbre binaire de la Figure 2.

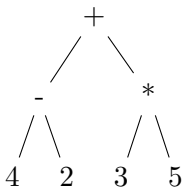

FIGURE 2 – ASA de l'expression arithmétique $(4 - 2) + (3 \times 5)$.

Nous allons représenter un ASA d'expressions arithmétiques à l'aide du type `BinaryTree`. Ne disposant que d'arbres binaires de nombres entiers, il est possible d'encoder les valeurs des nœuds intermédiaires, par exemple selon la correspondance arbitraire $\{+/0, -/1, \times/2, \div/3\}$.

Dans la suite de l'exercice, nous utilisons l'alias de type `ASTree` pour désigner ce type d'arbres. Déclarer un alias de type est trivial :

```
ASTree = BinaryTree
```

QUESTION 1. Écrire la fonction

```
def eval_ast(ast: ASTree) -> float
```

qui, étant donné un ASA, calcule le résultat numérique de l'expression correspondante. On supposera que l'expression est bien formée, et par conséquent, que l'ASA est strict [3].

QUESTION 2. Ajouter la fonction « constructeur »

```
def exp_to_ast(tokens: list[str]) -> ASTree
```

qui considère l'expression arithmétique sous forme de liste de termes (`tokens`) et qui construit l'ASA correspondant. Si la syntaxe est incorrecte, l'ASA est vide.

QUESTION 3. Enfin, combiner les briques précédentes pour définir la fonction

```
def eval(expr: str) -> float | None
```

qui prend en paramètre une chaîne de caractères `expr` représentant une expression arithmétique préfixée—sans parenthèses—et qui renvoie le résultat de l'évaluation de l'expression correspondante si elle est bien formée, `None` sinon. La fonction doit être capable de traiter des expressions en nombres entiers.

3. Chaque nœud possède soit 0 fils soit 2 fils.

Astuce: *Pour l'examen de la chaîne de caractères, il est recommandé d'utiliser la fonction* `str.split`.

Exercice 4: Arbres binaires de recherche

Dans cet exercice, nous nous intéressons aux *arbres binaires de recherche* (ABR ou *Binary Search Tree*—BST—en anglais), une structure de données permettant de représenter une collection d'éléments ordonnés par leurs clés—des valeurs quelconques munies d'une relation d'ordre. La collection peut être un ensemble, un tableau associatif, parfois même un multi-ensemble.

Pour mémoire, un ABR, aussi appelé *arbre binaire ordonné*, est un arbre binaire tel que pour tout nœud x :

— toutes les clés du sous-arbre gauche de x sont inférieures—ou égales—à la clé de x ;

— toutes les clés du sous-arbre droit de x sont supérieures à la clé de x.

Étant donné que les clés sont *a priori* triées, les ABR offrent l'avantage d'une procédure efficace de recherche de clé dans un ensemble, à condition que l'arbre soit bien équilibré. De même, les ABR permettent de trouver aisément le minimum et le maximum d'un ensemble, ainsi que les majorants et minorants d'une clé.

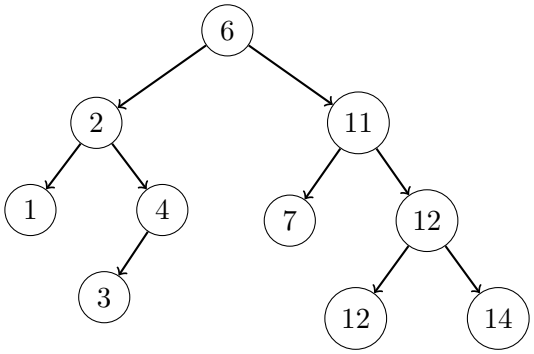

FIGURE 3 – Exemple d'ABR qui représente un multi-ensemble de nombres entiers.

Pour l'exercice, nous définissons un alias de type :

```
BSTree = BinaryTree     # ABR
```

QUESTION 1. Écrire la fonction :

```
def isBst() -> bool
```

qui teste si un arbre binaire est un ABR.

Astuce: *À chaque étape d'un parcours infixe, il faut tester la clé du nœud vis-à-vis de bornes inférieure et supérieure, établies pendant le parcours.*

QUESTION 2. Écrire successivement les fonctions :

```
def lookup_rec(key: int) -> Node
def lookup_it(key: int) -> Node
```

qui recherchent la clé `key` dans un ABR, par une procédure récursive (`_rec`) puis itérative (`_it`). Quelle est la complexité de ces algorithmes ?

QUESTION 3. Écrire la fonction :

```
def insert(key: int) -> BSTree
```

qui ajoute la clé `key` à l'ensemble (sans doublons) représenté par l'ABR `bst`. Quelle est la complexité de cet algorithme ?

Question 4. Proposer une fonction de suppression de clé dans un ABR :

```
def remove(key: int) -> BSTree
```

Exercice 5: Conversion d'un ABR vers une liste doublement chaînée

Selon Nick Parlante[4], il s'agit de « *one of the neatest recursive pointer problems ever devised* », autrement dit, un problème compliqué de manipulation de références !

L'idée est pourtant simple : un ABR représente un ensemble totalement ordonné d'éléments, donc il devrait être possible de le convertir en une liste qui respecte cet ordre. Pour ce faire, nous disposons du type `LinkedList` de listes circulaires doublement chaînées. Or, il est aisé d'observer l'analogie entre un maillon de la liste qui maintient deux références vers son prédécesseur et son successeur, et un nœud de l'arbre binaire de recherche qui maintient également deux références vers son fils gauche et son fils droit !

Le problème peut s'énoncer ainsi : étant donné un ABR, proposer une fonction *récursive*

```
def toList() -> LinkedList
```

de conversion en une liste doublement chaînée qui présente les éléments dans l'ordre. Chaque référence vers le sous-arbre gauche, resp. droit, doit correspondre à une référence vers le maillon précédent, resp. suivant.

Remarque: *Le cœur du problème réside dans l'assemblage des appels récursifs pour réorganiser les références de chaque sous-arbre lors de la phase d'induction, plutôt que dans la compréhension fine de ce qui se produit lors des appels récursifs*[5].

Astuce: *La phase d'induction créé deux sous-listes, une par sous-arbre, et concatène ces sous-listes autour du nœud courant.*

1 Pour aller plus loin...

Exercice 6: Codage de Huffman

Le codage de Huffman est une technique de compression de données sans perte. C'est une technique fréquentielle –statistique– au même titre que le morse, qui met en œuvre un principe simple : plus le symbole est fréquent, plus son code sera court. L'encodage d'un message fonctione en 3 étapes :

1. compter le nombre d'occurrences de chaque caractère ;
2. construire l'arbre de codage ;
3. suivre dans l'arbre le chemin gauche-droite (0-1) de chaque caractère pour produire le code du message.

4. http://cslibrary.stanford.edu/109/TreeListRecursion.html
5. C'est trop compliqué ! Et ici repose la magie de la récursivité...

Le décodage fonctionne de la même manière, à partir du message codé et de l'arbre de codage : pour décoder un caractère, il suffit de suivre dans l'arbre la séquence de bits du message codé, de la racine à une feuille.

La construction de l'arbre d'encodage suit le procédé ci-dessous :

1. initialiser un ensemble d'arbres enracinés par chaque caractère, chacun étant associé à un poids correspondant au nombre d'occurrences du caractère dans le message ;
2. fusionner deux à deux les arbres en prenant systématiquement les arbres de poids minimum. Les deux arbres deviennent les sous-arbres gauche et droite, le plus léger à gauche ; en cas d'égalité, on convient d'utiliser l'ordre lexicographique. Le poids de l'arbre résultat est la somme des poids des deux arbres.

Le dernier arbre est l'arbre de codage. Par exemple, le processus complet en 4 étapes, à partir du message $m =$`aabbbccccddddddddddd`, est détaillé ci-dessous :

1. initialisation : (a), 2 ; (b), 3 ; (c), 4 ; (d), 11
2. après la fusion de 2 et 3 : (c), 4 ; (. a b), 5 ; (d), 11
3. après la fusion de 4 et 5 : (. (c) (. a b)), 9 ; (d), 11
4. après la fusion de 9 et 11 : (. (. (c) (. a b)) (d)), 20

L'arbre de codage résultat de ce procédé est dessiné à la Figure 4.

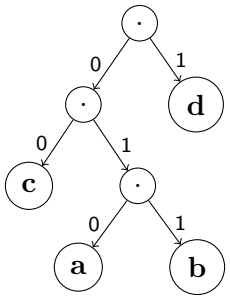

FIGURE 4 – Arbre de codage de Huffman pour $(a, 2); (b, 3); (c, 4); (d, 11)$.

Ainsi, le code de Huffman $H_m(.)$ de chaque caractère du message m est :
— $H_m(a) =$ `010`
— $H_m(b) =$ `011`
— $H_m(c) =$ `00`
— $H_m(d) =$ `1`

Notez qu'aucun code n'est le préfixe d'un autre ! Cette propriété importante du codage de Huffman permet de décoder un message de manière totalement déterministe.

QUESTION 1. Écrire la fonction d'encodage qui produit un message codé à partir d'un message clair et de son arbre de codage. Il pourra être utile de construire la table d'encodage comme structure intermédiaire, qui donne pour tout caractère c, son code $H_m(c)$.

QUESTION 2. Écrire la fonction de décodage qui produit un message clair à partir d'un message codé et de son arbre de codage.

QUESTION 3. Écrire la fonction de construction de l'arbre de codage à partir d'un message clair.

QUESTION 4. Appliquer le codage/décodage de Huffman aux messages suivants :
— "intelligence artificielle"
— "quarante crackers croient que croquer des carottes crues crée des crampes"
Calculer à chaque fois le taux de compression, en considérant que les messages sont encodés en ASCII, soit 7 bits par caractère.

En marge: cf. *le codage entropique vu en théorie de l'information.*