CSC-3TC33 : TRAVAUX PRATIQUES — SÉANCE n° 1

Objectif : — Deux grandes manières d'implémenter le *type abstrait liste* ont été vues en cours : avec des tableaux ou avec des maillons chaînés. L'implémentation avec des tableaux privilégie l'accès direct à n'importe quel élément ; l'implémentation avec des maillons simplement chaînés permet d'ajouter facilement un élément en tête tout en conservant intact la queue de la liste au détriment de l'accès direct.

Cette séance a pour but d'implémenter en Python une troisième structure de donnée qui doit permette à la fois d'ajouter facilement un élément en tête et qui autorise, pour tout indice i, la consultation rapide du i-ième élément de la liste. Cette structure de donnée s'appelle les *listes binaires gauches* ou *skew binary lists* ; elle est beaucoup utilisée en programmation fonctionnelle, en raison de son caractère persistent.

Consigne : — Pour chaque question, il est demandé d'écrire des tests pour justifier que la fonction fonctionne bien comme attendu. Chaque test sera introduit par l'instruction `assert` et sera conservé dans le fichier source afin d'être ré-exécuté à chaque modification de votre code.

Ce TP n'est pas noté ; aucun compte rendu n'est attendu. La participation active au TP est attendue ; les absences seront sanctionnées par un malus sur la note finale de l'UE.

Conseil de lecture : — Dans certains cas, il sera demandé d'implémenter des méthodes dont le nom est entouré par `__` : il s'agit de noms réservés à la prise en charge de comportements internes de Python. Nous renvoyons à la section 3.3.7 intitulée « Émulation de types conteneurs » (voir `https://docs.python.org/fr/3/reference/datamodel.html#emulating-container-types`) de la documentation de référence de Python.

1.1. Les arbres complets

Nous avons défini en cours une classe Python pour représenter les maillons d'arbres binaires étiquetés.

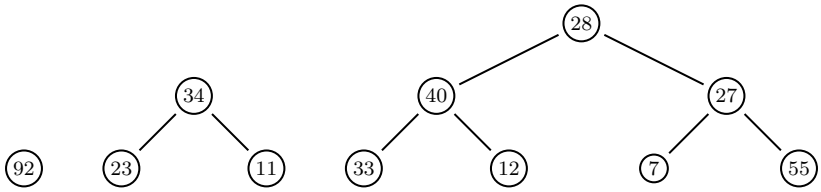
Question 0. — Créer un répertoire `csc3tc33` et un sous-répertoire de travail `tp1` au sein de votre répertoire `Document` (depuis un terminal, vous pouvez utiliser `mkdir` pour créer un répertoire, `cd` pour naviguer dans votre arborescence). Initialiser un fichier `completetrees.py` contenant

```python
# completetrees.py

class Node():
    # Classe à compléter d'après les questions de la partie 1
    def __init__(self, data, left = None, right = None):
        self.data = data
        if not(isinstance(left, Node) or left is None):
            raise ValueError("Left argument is not a node.")
        self.left = left
        if not(isinstance(right, Node) or right is None):
            raise ValueError("Right argument is not a node.")
        self.right = right

if __name__ == "__main__":
    # Tests à écrire au fur et à mesure
    assert(True)
    print("Tous les tests se sont bien passés")
```

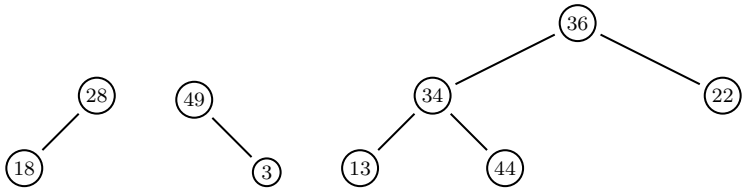
(Vous pouvez télécharger le fichier `squelette_partie1.py` pour vous aider). Exécuter l'instruction `python3 completetrees.py` et vérifier que le fichier s'exécute.

Question 1. — Modifier la classe `Node` pour ajouter un attribut `height` calculé à la construction. Par ailleurs, le constructeur doit désormais lever une exception `raise ValueError("Subtrees should have the same height.")` dans le cas où les arguments d'entrée `left` et `right` n'ont pas la même hauteur.

Vérifier que l'on parvient à créer les arbres arbres a_1, a_2 et a_3



et que leur hauteur est bien calculée ; mais que la création des arbres suivants provoque une erreur.



Question 2. — Afin de tester le constructeur écrit la question précédente, enrichir la classe `Node` d'une méthode `to_list(self)` qui renvoie la liste des étiquettes dans l'ordre préfixe (sous forme de liste Python de type `list`.).

Vérifier, à minima, que votre code fonctionne sur les tests suivants :

Entrée	Sortie
a_1	`[92]`
a_2	`[34, 23, 11]`
a_3	`[28, 40, 33, 12, 27, 7, 55]`

Dans le reste du sujet, il est interdit d'utiliser `to_list(self)`, sauf à des fins de test.

Question 3. — Enrichir la classe `Node` d'une méthode `__contains__(self, item)` qui renvoie le booléen Vrai si `item` est l'une des étiquettes de l'arbre `self`.

Vérifier que, si `t` est de type `Node`, on peut désormais appeler directement `x in t`.

Vérifier que l'élément 0 n'appartient pas aux arbres a_1, a_2 et a_3. Pour chaque élément x de a_1, a_2 et a_3, vérifier qu'il appartient à l'arbre.

Question 4. — Enrichir la classe `Node` d'une méthode `__len__(self)` qui renvoie le cardinal de `self`, c'est-à-dire le nombre de sommets.

Vérifier que, si `t` est de type `Node`, on peut désormais appeler directement `len(t)`.

Vérifier que le cardinal de a_1 vaut 1, le cardinal de a_2 vaut 3, le cardinal de a_3 vaut 7.

Question 5. — Enrichir la classe `Node` d'une méthode `__getitem__(self, key)` qui renvoie l'élément n° `key` quand on parcourt `self` dans l'ordre préfixe. Si `key` n'appartient pas à l'intervalle $[\![0, \texttt{len(self)}[\![$, une exception est levée à l'aide de l'instruction `raise IndexError("complete tree index out of range")`.

Vérifier que, si `t` est de type `Node` et `k` est un entier, on peut désormais utiliser directement la syntaxe `t[k]` pour des lectures.

Remarque 1.1. — Grâce à l'existence de `__getitem__(self, key)`, Python se débrouille même pour créer des itérateurs. Vérifier que l'on peut désormais utiliser la syntaxe de boucle « `for v in t:` ». Dans la partie 3, nous tenterons de mieux faire que des accès par indices pour itérer.

Remarque 1.2. — Pour modifier un terme de l'arbre avec la syntaxe la syntaxe `t[k] = x`, il nous faudrait écrire une méthode `__setitem__(self, key)`. Nous ne le faisons pas ici car nous souhaitons que notre structure soit persistante.

1.2. Les listes binaires gauches

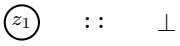
Définition 1.3. — Une *liste binaire gauche* est une suite simplement chaînée d'arbres étiquetés complets $\mathfrak{a}_1 :: \mathfrak{a}_2 :: \cdots :: \mathfrak{a}_t$ de hauteurs telles que

$$(\maltese) \qquad h(\mathfrak{a}_1) \leq h(\mathfrak{a}_2) < h(\mathfrak{a}_3) < h(\mathfrak{a}_4) < \cdots < h(\mathfrak{a}_t).$$

(La première inégalité est large, les suivantes sont toutes strictes). Les éléments d'une liste binaire gauche sont rangés : il y a d'abord les éléments de $\mathfrak{a}_1$ dans l'ordre préfixe,

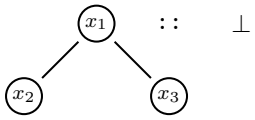
puis les éléments de $\mathfrak{a}_2$ dans l'ordre préfixe, etc, jusqu'aux éléments de $\mathfrak{a}_t$ dans l'ordre préfixe.

Exemple 1.4. — La liste binaire gauche de cardinal 1 a pour représentation

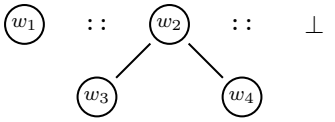


Exemple 1.5. — La liste binaire gauche de cardinal 2 a pour représentation

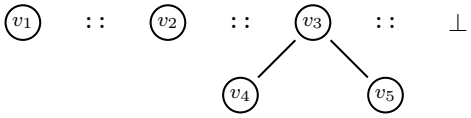
Exemple 1.6. — La liste binaire gauche de cardinal 3 a pour représentation



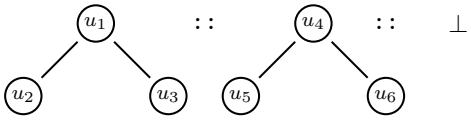
Exemple 1.7. — La liste binaire gauche de cardinal 4 a pour représentation



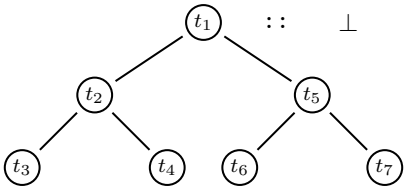
Exemple 1.8. — La liste binaire gauche de cardinal 5 a pour représentation



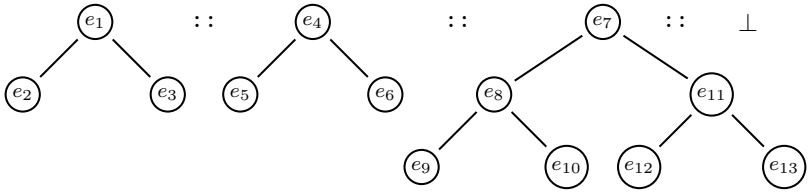
Exemple 1.9. — La liste binaire gauche de cardinal 6 a pour représentation



Exemple 1.10. — La liste binaire gauche de cardinal 7 a pour représentation



Exemple 1.11. — La liste binaire gauche de cardinal 13 a pour représentation



Question 0 bis. — Initialiser un fichier `skewbinarylists.py` contenant

```
18 # skewbinarylists.py
19
20 from completetrees import Node
21
22 class SkewBinaryList():
23     # Classe à compléter
24     def __init__(self, hd = None, tl = None):
25         self.head = hd
26         self.next = tl
27
28 if __name__ == "__main__":
29     # Ecrire les tests ici au fur et à mesure
30     print("Exécution terminée")
```

(Vous pouvez télécharger le fichier `squelette_partie2.py` pour vous aider). Dans un terminal, exécuter l'instruction `python3 skewbinarylists.py`.

Un des principes de la *programmation défensive* consiste à ne pas faire confiance aux données entrées par l'utilisateur.

Question 6. — Compléter le code du constructeur de la classe `SkewBinaryList` afin qu'une exception soit levée si l'argument `hd` n'est pas une instance de la classe `Node` ou encore si l'argument `tl` n'est pas soit `None`, soit une instance de la classe `SkewBinaryList`.

Question 7. — On souhaite enrichir la classe `SkewBinaryList` d'une méthode `cons(self, item)` dont la valeur de retour est une liste binaire gauche où l'on a ajouté l'élément `item` en tête de `self` et qui conserve l'ordre sur les éléments suivants. Réfléchir à votre algorithme en faisant des dessins puis implémenter votre solution. Quelle est la complexité de votre fonction ?

Question 8. — Afin de tester le constructeur écrit la question précédente, enrichir la classe `SkewBinaryList` d'une méthode `to_list(self)` qui renvoie les éléments sous forme de liste de type `list`.

Faites des tests ! Refaites des tests ! Tester encore ! Au fait, ne serait-ce pas un bon moment pour faire des tests ?

Question 9. — Enrichir la classe `SkewBinaryList` d'une méthode `__contains__(self, item)` qui renvoie le booléen Vrai si `item` appartient à `self` et Faux sinon.

Question 10. — Enrichir la classe `SkewBinaryList` d'une méthode `__len__(self)` qui renvoie le cardinal de `self`, c'est-à-dire le nombre d'éléments dans la liste.

Question 11. — Enrichir la classe `SkewBinaryList` d'une méthode `__getitem__(self, key)` qui renvoie l'élément n° `key` quand on parcourt `self` dans l'ordre préfixe. Si `key` n'appartient pas à l'intervalle $[\![0, \texttt{len(self)}[\![$, une exception est levée à l'aide de l'instruction `raise IndexError("skew binary list index out of range")`. Quelle est la complexité de votre fonction ?

Question 12. — Enrichir la classe `SkewBinaryList` d'une méthode `tail(self)` dont la valeur de retour est une nouvelle liste binaire gauche où l'on a retiré l'élément en tête de `self` et qui conserve l'ordre sur les éléments suivants. Quelle est la complexité de votre fonction ?

1.3. Pour aller plus loin

À ce stade, Python est capable d'itérer sur une liste binaire gauche grâce à l'accès par indice. Il serait plus efficace d'itérer en se déplaçant dans l'arbre d'un sommet à ses voisins.

Question 13. — Enrichir la classe `SkewBinaryList` d'une méthode `__iter__(self)` et d'une méthode `__next__(self)` qui permet de munir la classe d'un itérateur. À la fin de l'itération, une exception est levée par `__next__` à l'aide de l'instruction `raise StopIteration`.

À ce stade, lors d'un test d'égalité `L1 == L2` entre deux listes binaires gauches L_1 et L_2, Python se contente de comparer l'identifiant de chaque objet pour rendre une décision (en gros, l'identifiant s'agit de son adresse mémoire, à ceci près que la notion d'adresse mémoire n'existe pas dans la sémantique de Python).

Question 14. — Enrichir la classe `SkewBinaryList` d'une méthode `__eq__(self, other)` qui évalue l'égalité entre les contenus des listes `self` et `other`.

À ce stade, il n'est pas possible de créer un ensemble (de type `set`) contenant une liste binaire gauche, ou un dictionnaire (de type `dict`) dont la clé est une liste binaire gauche, du fait de l'inexistence d'une fonction de hachage. Comme, nous nous sommes efforcés à écrire une structure de donnée immuable, il est imaginable de doter nos listes d'une fonction pour les hacher.

Question 15. — Enrichir la classe `SkewBinaryList` d'une méthode `__hash__(self, other)` qui produit un haché du contenu de la liste `self`. Il est rappelé qu'une fonction de hachage doit satisfaire

$$\forall L_1, L_2, \quad \texttt{L1 == L2} \text{ implique } \texttt{hash(L1) == hash(L2)}.$$

À ce stade, vos tests sont éparpillés dans le code source et mélangés avec la partie de code vraiment intéressante.

Question 16. — Reprendre l'ensemble des tests de votre code et les regrouper à travers le *framework* de tests unitaires `pytest`.