Les piles

Exercice 1 :

Dans la chambre des enfants, un nouveau jeu vient d'être inventé :
la plus haute montagne de livres.

Chacun son tour, un enfant pose un nouveau livre sur la montagne.

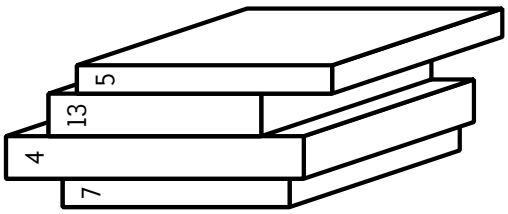
On modélise une montagne par une liste où le dernier élément correspond au dernier livre ajouté.
Lorsque le jeu n'a pas commencé, la montagne est vide et la liste aussi.

Les enfants ne sont capables que d'effectuer quatre opérations :

- ❏ Créer une montagne vide ;
- ❏ Vérifier si la montagne est vide ;
- ❏ Ajouter un livre à la montagne ;
- ❏ Retirer un livre à la montagne ;

Par soucis de simplification, on dira qu'un livre est représenté par un entier.

Par exemple, la liste `[7, 4, 13, 5]` représente la montagne :



En python, les quatre actions possibles pour un enfant sont représentées par les fonctions suivantes :

Action	Fonction
Créer une montagne vide	`creer_montagne() -> list`
Vérifier si la montagne est vide	`est_vide(L : list) -> bool`
Ajouter un livre à la montagne	`ajouter_livre(L : list, n : int) -> None`
Retirer un livre à la montagne	`retirer_livre(L : list) -> int`

1. Quelles ont été les instructions pour créer la montagne `[7, 4, 13, 5]` ?

2. Bastien souhaite récupérer le livre numéro 4 et reconstruire la montagne sans celui-ci.
Quelles instructions doit-il écrire pour cela ?

D.W.

3. Écrire une implantation des quatre actions possibles pour un enfant.

4. a. Écrire une fonction `retourner(L : list) -> list` qui renvoie la version retournée de `L`.
Attention : en n'utilisant que les fonctions disponibles.

b. On considère le script :

```
L = [7, 4, 13, 5]
R = retourner(L)
```

Que vaut la liste `R` et que vaut la liste `L` ?

c. Compléter le code de la fonction `hauteur(L : list) -> int` qui renvoie la hauteur de la montagne.

```
def hauteur(L : list) -> int:
    """ Hauteur d'une montagne """
    res = 0
    R = retourner(L) # Attention L est vide maintenant

    while not est_vide(...):
        ajouter_livre(..., ...)
        res = ...

    return res
```

d. On propose le code suivant :

```
def foo(L : list) -> int:
    """ Fonction secrète. """
    if est_vide(L):
        res = 0
    else:
        sommet = retirer_livre(L)
        res = 1 + foo(L)
        ajouter_livre(L, sommet)
    return res
```

Donner les appels récursifs lors de l'instruction `foo(L)` avec `L = [7, 4, 13, 5]`.

5. En calquant les fonctions précédentes :

a. Écrire une fonction itérative `recuperer(L : list, n : int) -> bool` qui retire le livre n de la montagne et renvoie `True` s'il a été trouvé et `False` sinon.

Par exemple, avec `L = [7, 4, 13, 5]`, l'instruction `recuperer(L, 4)` renverrait `True` et la montagne `L` serait égale à `[7, 13, 5]`.

b. Proposer une version récursive de la fonction précédente.

Cours

En informatique, une **PILE** est une structure de données abstraite qui respecte le principe suivant :

Premier arrivé = Dernier sorti

On résume ce principe par l'acronyme **FILO** pour **First In, Last Out**.
Cette structure de données peut être modélisée avec différents objets en Python :

- ❏ Une liste ;
- ❏ Un dictionnaire à deux entrées : `"sommet"` et `"reste"` ;
- ❏ Une classe (Voir le chapitre sur la programmation objet) ;

L'**interface** de cette structure est constituée des primitives suivantes :

nom	argument(s)	description
creer_pile	vide	Crée une pile vide
est_vide	Pile p	Vérifie si la pile p est vide
empiler	Pile p, valeur v	Empile la valeur v dans la pile p
depiler	Pile p	Retire la valeur en tête de pile (si possible) et la renvoie

Exercice 2 :

On considère le programme suivant :

```
pile = creer_pile()
empiler(pile, 13)
empiler(pile, 7)
empiler(pile, 18)
a = depiler(pile)
empiler(pile, 2*a)
empiler(pile, depiler(pile))
```

1. Donner l'état de la pile après chacune des instructions de ce programme.

Exercice 3 :

Dans cet exercice, on propose l'implantation suivante de la primitive `creer_pile` :

```
def creer_pile() -> dict:
    """ Crée une pile version dictionnaire """
    res = {"sommet" : None, "reste" : []}
    return res
```

Par exemple, la pile $\begin{array}{|c|}\hline 8 \\ \hline 3 \\ \hline 5 \\ \hline 1 \\ \hline \end{array}$ est représentée par le dictionnaire `{"sommet" : 8, "reste" : [1, 5, 3]}`

1. Écrire l'implantation de `empiler(P : dict, n : int) -> None` qui empile l'entier n dans la pile P.
2. Écrire l'implantation de `depiler(P : dict) -> int` qui dépile le sommet de P.

Exercice 4 :

Dans cet exercice, on propose l'implantation suivante de la primitive `creer_pile` :

```
def creer_pile() -> dict:
""" Crée une pile version dictionnaire """
   res = {"sommet" : None, "reste" : {}}
   return res
```

où `"reste"` est associé à un dictionnaire représentant la pile restante sous le sommet.

Par exemple, la pile $\begin{array}{|c|}\hline 8 \\ \hline 3 \\ \hline 5 \\ \hline 1 \\ \hline \end{array}$ est représentée par le dictionnaire

```
{"sommet" : 8,
 "reste"  : {"sommet" : 3,
             "reste"  : { "sommet" : 5,
                          "reste"  : { "sommet" : 1,
                                       "reste"  : { "sommet" : None,
                                                    "reste" : {}}}}}}
```

1. Écrire le dictionnaire associé à la pile $\begin{array}{|c|}\hline 2 \\ \hline 10 \\ \hline 3 \\ \hline 7 \\ \hline \end{array}$

2. Écrire l'implantation de `empiler(P : dict, n : int) -> None` qui empile l'entier n dans la pile P.

3. Écrire l'implantation de `depiler(P : dict) -> int` qui dépile le sommet de P.

Cours

Il importe peu de la façon dont la pile a été modélisée.
Seules vont changer les implantations des fonctions primitives associées à la structure.

Exercice 5 :

On s'intéresse au bon parenthèsage d'une expression arithmétique.
Par exemple l'expression $((3 \times 2) - 1 + 4)$ est bien parenthésée alors que $(3 + 2/(4 - 1) - 3) + 2)$ non.

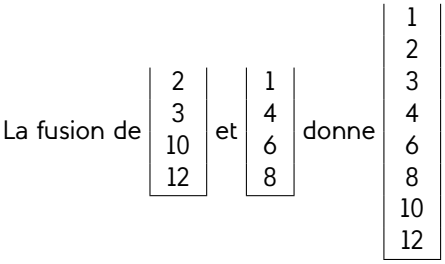
Pour vérifier qu'une expression arithmétique (modélisée par une chaine de caractères) est bien parenthésée, on la parcourt caractère par caractère en respectant la règle :

- ❏ si le caractère est une parenthèse ouvrante, on l'empile dans une pile ;
- ❏ si le caractère est une parenthèse fermante, on dépile si possible.

1. Écrire `bien_parenthesee` qui vérifie qu'une expression passée en paramètre est bien parenthésée.

Exercice 6 :

On s'intéresse à la fusion de deux piles triées dans l'ordre croissante (de la tête vers le fond).
Par exemple :



1. Proposer une fonction `fusion` qui fusionne deux listes passées en paramètres.

Exercice 7 :

La notation polonaise inverse (NPI) permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses. Par exemple l'expression `((1 + 2) * 4) + 3` peut s'écrire `1 2 + 4 * 3 +`

Pour évaluer une expression en NPI, on utilise une pile. Dans l'exemple ci-dessus, on a :

	entrée	opération	pile
Étape 1	1	Empiler l'opérande	1
Étape 2	2	Empiler l'opérande	2 1
Étape 3	+	Addition	3
Étape 4	4	Empiler l'opérande	4 3
Étape 5	*	Multiplication	12
Étape 6	3	Empiler l'opérande	3 12
Étape 7	+	Addition	15

1. On considère l'expression écrite en NPI : `7 2 - 4 + 3 * 9 / 1 -`.

a. Évaluer cette expression.

b. Écrire cette expression en notation classique.

2. On considère l'expression `2 3 + 7 * 4 - 2 1 16 2 / + * +`.
Donner l'évolution de l'état de la pile lors de l'évaluation de cette expression.

3. a. Proposer une implémentation d'une fonction `addition` qui à une pile contenant au moins deux éléments remplace les deux éléments de tête par leur somme.

b. On suppose que `multiplication`, `soustraction` et `division` ont été implémentées.
Proposer une fonction `evaluation` qui évalue une expression écrite en NPI.

D.W.