Exercice 1

Cours : structures de données linéaires

1.1 Structure de données, type abstrait, structure de données linéaire

Une **structure de données** est une manière de stocker, d'accéder à, et de manipuler des données.

La résolution d'un problème par un programme nécessite :

- de définir les données du problème
- d'utiliser des instructions du langage sur ces données

Souvent, la résolution du problème demande d'arranger ces données dans un *type abstrait*, afin d'avoir une résolution plus efficace.

Un type abstrait est défini par son *interface*, qui est indépendante de son *implémentation*.

Les types abstraits sont des types *structurés*. On a déjà vu des types *structurés* natifs : les types séquences de texte (string), les types séquentiels (listes, tuples), et les mappages (dictionnaires).

Un type abstrait est caractérisé par une *interface* de programmation qui permet de manipuler les données de ce type. Sans pour autant avoir connaissance du contenu des fonctions proposées par l'interface.

Un **type abstrait** décrit essentiellement une interface, indépendamment du langage de programmation.

Spécifier un type abstrait, c'est définir son *interface*, sans prejuger de la façon dont ses opérations sont implémentées.

Structure de données **linéaire** : les données sont rangées sur un ligne ou une colonne.

1.2 Pile

1.2.1 Def et interface

Une Pile est une structure de données linéaire. Le dernier élément entré sera aussi le premier à sortir (Last In First Out : LIFO). C'est une structure très utilisée pour le *retour en arrière*, *backtracking*, le parcours en profondeur d'un graphe…

L'interface d'une structure de données décrit de quelle manière on peut la manipuler. Les opérations possibles sont :

- test d'une pile vide (renvoie `True` si la pile est vide)
- ajout d'un élément (empiler = `push`), mis au sommet de la pile
- retire le premier élément de la pile, celui au sommet (dépiler = pop) si la pile est non vide et renvoie cet élément.
- lire le sommet de la pile.

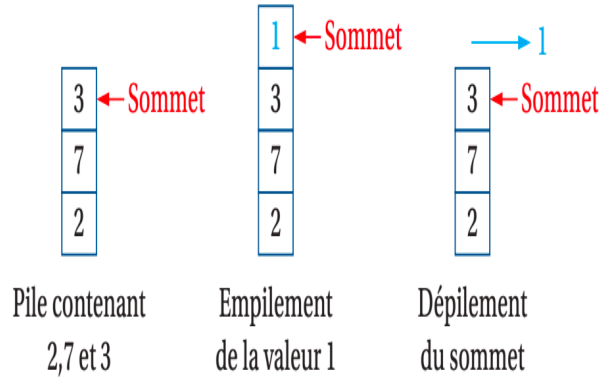

FIGURE 1 – illustration du fonctionnement d'une Pile

1.2.2 Implémenter un pile en Python en langage natif

Le type *List* en Python possède déjà toutes les méthodes d'une pile :

```
pile = []             # creation d'une pile
pile == []            # tester si la pile est vide
pile.append(valeur) # empile valeur dans la pile
pile.pop()          # depiler l'element au sommet
pile[-1]              # lire l'element au sommet de la pile
```

Lorsque l'on fait référence à une structure de données de type *pile* en traduisant un algorithme en Python, il faudra se contenter de ces 5 instructions.

1.2.3 Créer une nouvelle Interface

Les types abstraits, comme les piles, sont définis par leur **interface** (comment on s'en sert) plutôt que par leur **implémentation** (comment ils fonctionnent). Ils permettent d'étudier des algorithmes indépendamment du langage utilisé.

Rappels :

- L'*interface* de la structure de données décrit de quelle manière on peut la manipuler, par exemple en utilisant `append` pour le type list ou `get` pour le type dict.
- L'*implémentation* de la structure de données, contient le code de ces méthodes (comment fait Python). Il n'est pas nécessaire de connaître l'implémentation pour manipuler la structure de données.

On verra ici deux nouvelles manières d'implémenter la *pile*.

1.2.4 Implémentation fonctionnelle

Pour traduire un algorithme utilisant cette structure *pile*, il peut être préférable de définir des instructions portant les noms de ces 5 instructions :

```
def Pile():
  """creation d'une pile vide à l'aide de l'instruction :
  pile = Pile()
  """
  return []

```

```
def est_vide(pile)
  # à compléter

def empile(valeur,pile):
  # à compléter

def depile(pile):
  # à completer

def sommet(pile):
  # à completer
```

1.2.5 Implémentation en Programmation Objet

On peut créer un type (= une **classe**) `Pile` personnelle en Python.

Les méthodes (fonctions) déclarées précédemment sont alors associées à l'objet `Pile`. L'appel d'une méthode se fait à l'aide de l'instruction :

```
pile = Pile()
pile.est_vide()
pile.empile(valeur)
pile.depile()
pile.sommet()
```

Voir le cours sur la programmation orientée objet.

Rappel : Objet = type mutable. Quelle que soit la façon avec laquelle la pile est implémentée, il faudra se souvenir que celle-ci est un objet *mutable*. Cela a des conséquences sur la manière avec laquelle la pile est passée en argument d'une fonction, et comment cette fonction modifie la pile.

1.3 Comment la pile d'appel fonctionne t-elle avec une fonction recursive

illustration

1.4 File

1.5 Definition et interface

Une file (queue ou FIFO pour first in, first out) est une collection d'objets munie des opérations suivantes :

- savoir si la file est vide (is_empty) ;
- ajouter un élément dans la file (enfiler ou enqueue) en temps constant ;
- retirer et renvoyer l'élément le plus ancien de la file (defiler ou dequeue).

C'est une structure de données utilisée en informatique pour par exemple gérer la file d'attente lors de la gestion des impressions.

1.6 Implémentations en python natif

L'implementation d'une File est imparfaite avec une liste en python pour l'opération de *defiler*. On peut utiliser l'instruction `file.pop(0)` :

```
> F = []
> F.append(1)
> F.append(2)
> F.append(3)
> print(F)
[1,2,3]
> F.pop(0)
> print(F)
[2,3]
```

Le problème est que celle-ci ne s'effectue pas en temps constant.

1.7 Implémentation avec l'extension `deque`

La *bibliothèque standard* possède toutefois un module nommé `collections` qui contient quelques structures de données supplémentaires, dont le type `deque`, qui est une implémentation de file :

```
from collection import deque
f = deque()
f.append("Premier")
f.append("Deuxieme")
print(f.popleft())
f.append("Troisieme")
while f:
  print(f.popleft())
```

affiche :

```
Deuxieme
Troisieme
```

Exercice 2

Listes

2.1 Principe

Une liste chaînée est une structure linéaire, constituée de cellules. Une cellule (ou *maillon* contient :

- une valeur (de n'importe que type),
- et un pointeur qui dirige soit vers la cellule suivante, soit vers `None`, auquel cas la cellule est la *dernière* de la séquence.

Il s'agit d'une structure *linéaire*, mais dans laquelle les éléments n'occupent pas *à priori*, des positions contigües dans la mémoire :

(voir cours sd2 : Listes)