Structures de données : listes, piles, files

Charles Grellois

Structures de données

On va dans les cours à venir parler de *structures de données*. Ces structures permettent d'organiser les données, et choisir une structure adaptée à un problème va permettre de le résoudre plus facilement et plus efficacement.

On va commencer dans ce cours par les listes, piles et files :

- les *listes* permettent de... lister les objets, par exemple les participants à ce cours (qui seraient modélisés comment ?)
- les *piles* permettent d'entasser des objets. La machine les utilise notamment avec la pile d'exécution. On verra qu'elles sont utiles pour détecter les palindromes, par exemple.
- les *files* permettent de mettre des objets à la queue-leu-leu. On les utilisera pour simuler un ordonnanceur simple.

On va voir que plusieurs implémentations différentes sont possibles.

Listes, piles, files

Listes en Python

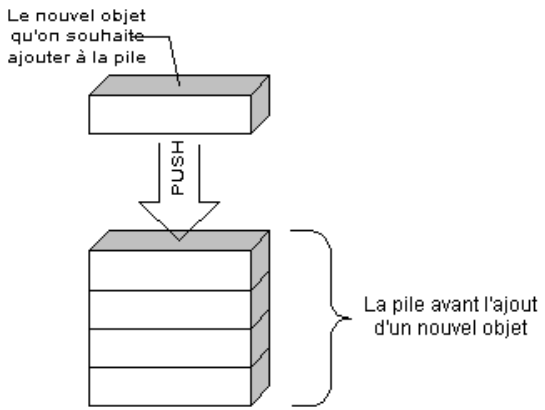
Une **liste** est un agrégat d'éléments qui sont repérés par leur *position* (ou *indice*).

Quelles opérations de base connaissez-vous sur les listes ? Cf le manuel de Python ici et là.

Dans une liste codée ainsi, on accède à n'importe quel élément grâce à son indice.

La pile

Une pile est comme une pile d'assiettes : on pose les assiettes sur le dessus, et on les récupère à partir du haut de la pile.



La pile

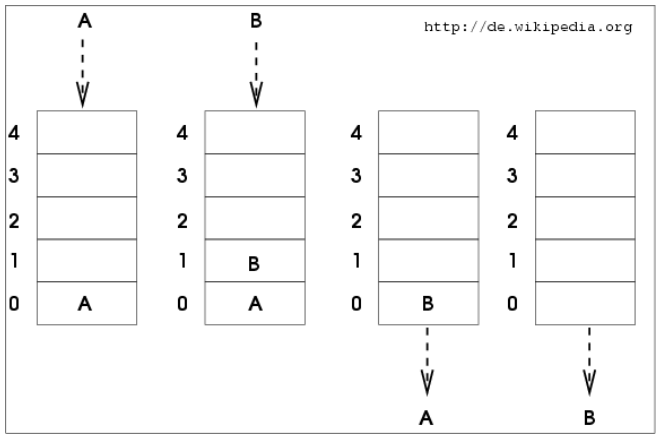
Opérations de base :

- `push(element)` qui insère un objet `element` en haut de la pile
- `pop()` qui échoue si la pile est vide et sinon dépile l'élément le plus haut de la pile et le renvoie
- `est_vide()` qui teste si la pile est vide
- `profondeur()` qui renvoie le nombre d'éléments dans la pile

A partir de ceci, on définit une **interface** et une **spécification**, et toute implémentation de cette interface respectant la spécification sera utilisable pour gérer une pile.

La file

Une file est comme une file d'attente : on entre à la fin, on en sort quand on est en première position.



La file

Opérations de base :

- `enfiler(element)` ajoute un objet à la fin de la file
- `defiler()` échoue si la file est vide et sinon défile le premier élément et le renvoie
- `est_vide()` qui teste si la file est vide
- `longueur()` qui renvoie le nombre d'éléments dans la file

On va implémenter piles et files en TP, ainsi que des algorithmes pour lesquels ces structures de données sont de "bons choix" : elles aideront à résoudre naturellement les problèmes.

Un aparté sur la programmation fonctionnelle

Manipulation des fonctions

Dans un langage fonctionnel, on peut manipuler des fonctions comme on manipulerait n'importe quelle autre valeur. On dit que les fonctions sont traitées comme des citoyens de première classe (*first-class citizen* en anglais).

On peut ainsi prendre des fonctions en argument et en renvoyer comme résultat d'une fonction.

A ce titre, Python est un langage fonctionnel.

Manipulation des fonctions

```
import math

def comp(f,g):
    return lambda x : f(g(x))

h = comp(math.sin,math.cos)
print(h(0.5))
```

On a pris en argument des fonctions, et retourné une fonction à l'aide du constructeur `lambda`.

Le pattern-matching

Dans des langages fonctionnels tels que OCaml, on dispose du *pattern-matching*, qui n'existe en Python que pour les tuples.

On peut créer des types élaborés :

```
type expr =
      | Plus of expr * expr          (* means a + b *)
      | Minus of expr * expr         (* means a - b *)
      | Times of expr * expr         (* means a * b *)
      | Divide of expr * expr        (* means a / b *)
      | Value of string              (* "x", "y", "n", etc. *)
;;
```

Le pattern-matching

On peut ensuite déconstruire les expressions formées à l'aide des types :

```
let factorize e =
    match e with
    | Plus (Times (e1, e2), Times (e3, e4)) when e1 = e3 ->
       Times (e1, Plus (e2, e4))
    | Plus (Times (e1, e2), Times (e3, e4)) when e2 = e4 ->
       Times (Plus (e1, e3), e4)
    | e -> e;;
```

Le pattern-matching, autre exemple

```
let rec multiply_out e = match e with
    | Times (e1, Plus (e2, e3)) ->
       Plus (Times (multiply_out e1, multiply_out e2),
             Times (multiply_out e1, multiply_out e3))
    | Times (Plus (e1, e2), e3) ->
       Plus (Times (multiply_out e1, multiply_out e3),
             Times (multiply_out e2, multiply_out e3))
    | Plus (left, right) ->
       Plus (multiply_out left, multiply_out right)
    | Minus (left, right) ->
       Minus (multiply_out left, multiply_out right)
    | Times (left, right) ->
       Times (multiply_out left, multiply_out right)
    | Divide (left, right) ->
       Divide (multiply_out left, multiply_out right)
    | Value v -> Value v;;
```

Pattern-matching en Python : seulement sur les tuples

Exemple tiré de PythonTutor :

```
def listSum(numbers):
    if not numbers:
        return 0
    else:
        (f, rest) = numbers
        return f + listSum(rest)

myList = (1, (2, (3, None)))
total = listSum(myList)
```

Pattern-matching sur les couples : `(f,rest) = ...`

Manipuler les pointeurs : listes chaînées

Listes chaînées

Une liste chaînée désigne en informatique une **structure de données** représentant une collection *ordonnée* et de *taille arbitraire* d'éléments de même type (*), dont la représentation en mémoire de l'ordinateur est une succession de cellules faites d'un **contenu** et d'un **pointeur vers une autre cellule**.

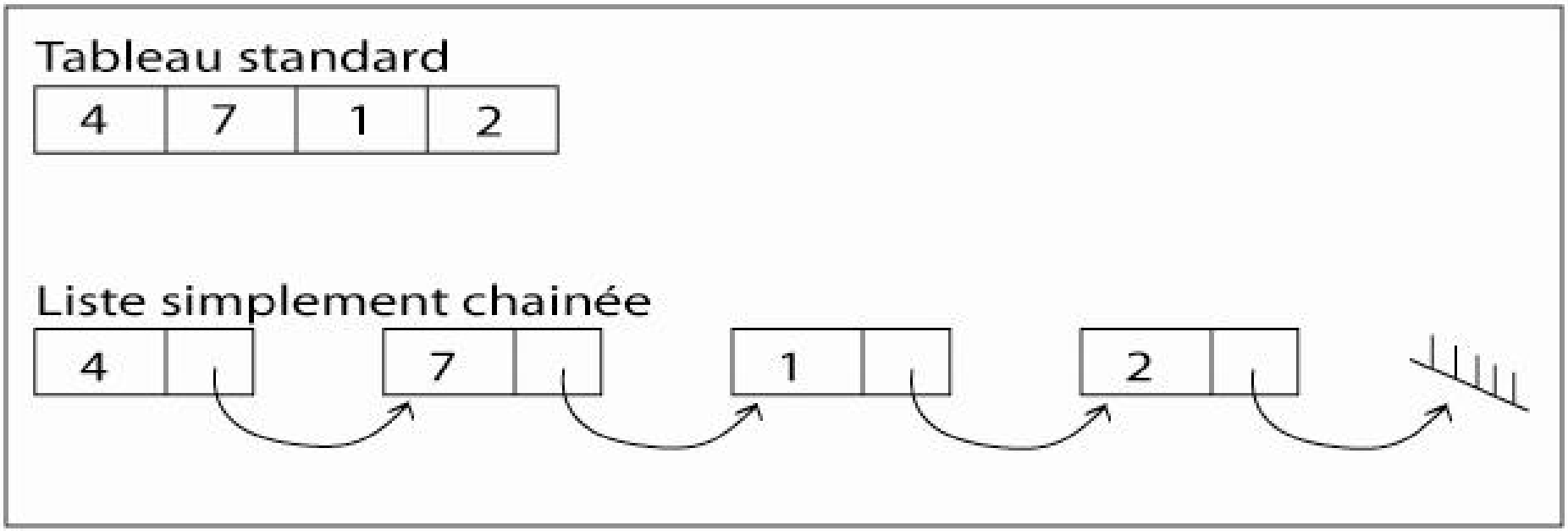
(*) certains langages dont Python autorisent cependant des listes dont les éléments ne sont pas du même type.

Le code de PythonTutor manipule une liste chaînée, analysons-le !

Remarquez bien la structure de la liste chaînée : chaque case mémoire contient une valeur et un pointeur vers la cellule suivante.

Il n'y a plus de notion d'indice mais une notion de **successeur**.

Listes chaînées



A implémenter en TP !

Interface et spécification

Deux classes de même interface et de même spécification sont **interchangeables**, puisqu'elles ont les mêmes méthodes et que ces méthodes font les mêmes choses, même si ça peut être codé différemment.

On a codé la pile et la file à l'aide des listes Python : on va voir en TP qu'on peut les remplacer par des listes chaînées implémentées par nos soins sans rien changer à l'exécution de programmes utilisant ces classes.