Structures de données linéaires

Idée : "Programs = Data structures + algorithms" (Wirth)

Pour améliorer l'efficacité de certains algorithmes nous devons maîtriser certaines structures de données

Dans un premier temps nous allons étudier les structures linéaires (listes, files et piles)

On commence par rappeler la structure de tableau

1 Tableau

Définition 1. *1. Un tableau T est un ensemble de valeurs* ***de même type*** *et qui sont indiciées ou numérotées de 0, la première valeur jusqu'à la dernière $n-1$ où n est la taille du tableau* ***fixé à la construction du tableau***

2. On peut avoir accès à toute valeur d'indice i pour i compris entre 0 et $n-1$ du tableau ***en un temps constant*** *par l'écriture T[i]*

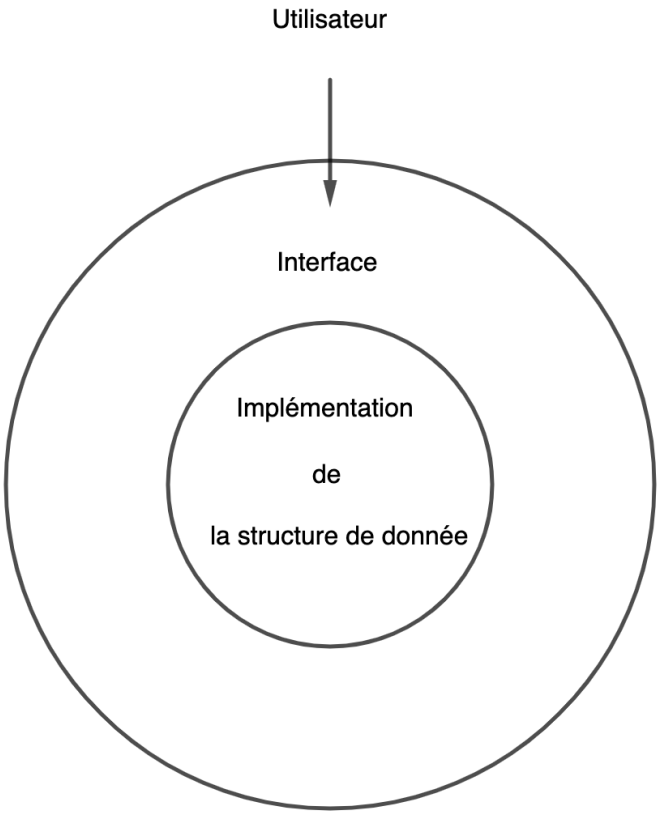
3. On ne peut pas insérer d'élément dans un tableau au delà des emplacements numérotés de 0 à $n-1$

A cause de la dernière remarque on voit qu'une liste en Python n'est pas un tableau, puisqu'avec la méthode `append()` on peut ajouter un élément à une liste donnée

On trouve la structure de tableau dans des langages comme C, Java et C++

Nous allons voir maintenant les structures **dynamiques**, listes, piles et files

2 Interface et implémentation



Les structures **dynamiques**, listes, piles et files ont un comportement **abstrait** identique quel que soit le langage de programmation

Nous allons nous focaliser sur un ensemble de fonctions abstraites ou **primitives** qui définissent chacun de ces types **sans entrer dans le détail de leur définition** on parle alors **d'interface** ou encore **API** pour application programming interface

De la même manière un conducteur d'automobiles maîtrise l'interface d'une voiture définie par les primitives, `démarrer()`, `passer_vitesses()`, etc... **sans connaitre dans les détails le fonctionnement du moteur, de la boîte de vitesses etc...**

L'interface est une sorte de contrat qui nous assure du bon fonctionnement des objets sous certaines conditions

Nous verrons ensuite comment telle primitive peut être définie ou **implémentée** en Python par exemple

3 Liste chainée

On a vu dans le cours de programmation orientée objet la notion de liste chaînée que l'on a implémenté en Python

Une liste chaînée est un ensemble d'éléments de même type

On accède à cet ensemble par le premier élément et de chaque élément on peut accéder au suivant

Les opérations minimales sur une liste chaînée (on parle de primitives ou d'interface) sont ici :

Définition 2.
1. `creer-liste-vide()` *qui renvoie une liste vide*
2. `liste-vide(L)` *qui renvoie vraie si L est une liste vide faux sinon*
3. `tete(L)` *qui renvoie le premier élément de la liste*
4. `successeur(L,x)` *qui renvoie l'élément qui suit x dans la liste L. Si il n'y a pas d'élément renvoie* `None`
5. `ajouter(L,x)` *qui ajoute un élément x dans la liste L (en début de liste)*
6. `supprimer(L,x)` *qui supprime un élément présent x dans la liste L de la liste L.*

A partir de ces primitives on peut écrire en pseudo-code d'autres fonctions par exemple une fonction `longueur(L)` qui renvoie un entier égal au nombre d'éléments dans la liste L

Algorithme 1 : Longueur d'une liste

```
début
    /* Définition de la fonction longueur(L)  */
1   longueur (L)
    Données : Une liste L
    Résultat : un entier le nombre d'éléments de L
2   début
3       nbElts ← 0
4       x = tete(L)
5       tant que x n'est pas None faire
6           nbElts ← nbElts + 1
7           x = successeur(L,x)
8       fin
9       retourner nbElts
10  fin
fin
```

Voici par exemple une partie de l'API de Python sur les listes Python en notation objet. On se rend compte que la classe `list` de Python est plus complexe qu'une simple liste chaînée.

`list.` **`append`**(*x*)
Ajoute un élément à la fin de la liste. Équivalent à `a[len(a):] = [x]`.

`list.` **`extend`**(*iterable*)
Étend la liste en y ajoutant tous les éléments de l'itérable. Équivalent à `a[len(a):] = iterable`.

`list.` **`insert`**(*i*, *x*)
Insère un élément à la position indiquée. Le premier argument est la position de l'élément avant lequel l'insertion doit s'effectuer, donc `a.insert(0, x)` insère l'élément en tête de la liste et `a.insert(len(a), x)` est équivalent à `a.append(x)`.

`list.` **`remove`**(*x*)
Supprime de la liste le premier élément dont la valeur est égale à *x*. Une exception `ValueError` est levée s'il n'existe aucun élément avec cette valeur.

On se rend compte aussi que **tout est fait dans l'interface pour simplifier l'utilisation de la structure de donnée par l'usager** ainsi dans la méthode `list.remove(x)`, `x` est

une valeur et non un pointeur.

Dans le chapitre sur la programmation objet nous avons commencé à implémenter une classe `Liste` dans laquelle la méthode `supprimer` est moins facile d'utilisation.

Dans l'exercice 1 on vous propose de réécrire cette méthode de telle sorte que sa spécification se rapproche de celle de la méthode `remove` de Python.

4 Pile

Une image d'une pile est une pile d'assiettes dont la taille varie avec le temps.

La **dernière** assiette empilée au sommet de la pile est la **première** accessible (LAST IN FIRST OUT ou LIFO)

Les opérations possibles sur une pile (on parle de primitives ou d'interface) sont

Définition 3.
1. *`creer-pile-vide()` qui renvoie une pile vide*
2. *`pile-vide(P)` qui renvoie vraie si P est la pile vide faux sinon*
3. *`empiler(P,x)` qui renvoie rien mais qui met x au sommet de la pile P*
4. *`depiler(P)` qui renvoie le sommet de la pile P qui est donc supprimé de la pile P*

A partir de ces primitives on peut écrire en pseudo-code d'autres fonctions par exemple `hauteur(P)` qui renvoie un entier égal au nombre d'éléments dans la pile P

Algorithme 2 : Hauteur d'une pile

```
début
    /* Définition de la fonction hauteur(P)    */
    Données : Une pile P
    Résultat : un entier le nombre d'éléments de P
1   hauteur (P)
2   début
3       nbElts ← 0
4       Q ← creer_pile_vide()
5       tant que non (est-vide(P)) faire
6           empiler(Q,depiler(P))
7           nbElts ← nbElts+1
8       fin
9       //on remet P dans son état initial
10      tant que non (est-vide(Q)) faire
11          empiler(P,depiler(Q))
12      fin
13      retourner nbElts
14  fin
fin
```

5 File

Une image d'une file est une file d'attente.

La **première** personne arrivée est la **première** personne à quitter la file d'attente (FIRST IN FIRST OUT ou FIFO)

La dernière personne arrivée prend sa place à la fin de la file

Les opérations possibles sur une file sont

Définition 4. *1.* `creer-file-vide()` *qui renvoie une file vide*

2. `file-vide(F)` *qui renvoie vraie si F est vide faux sinon*

3. `enfiler(F,x)` *qui renvoie rien mais qui met x à la fin de file*
4. `defiler(F)` *qui renvoie la tête de la File F et qui est supprimée de la file F*

A partir de ces primitives on peut écrire en pseudo-code d'autres fonctions par exemple une fonction `longueur(F)` qui renvoie un entier égal au nombre d'éléments dans la file F

Algorithme 3 : Longueur d'une file

```
Données : Une file F
Résultat : un entier le nombre d'éléments de F
1  début
       /* Définition de la fonction longueur(F)  */
2      longueur (F)
3      début
4          nbElts ← 0
5          G ← creer_file_vide()
6          tant que non (est-vide(F)) faire
7              enfiler(G,defiler(F))
8              nbElts ← nbElts+1
9          fin
10         tant que non (est-vide(G)) faire
11             defiler(G)
12         fin
13         retourner nbElts
14     fin
15 fin
```

6 Implémentations d'une pile

Implémenter signifie mettre en oeuvre, autrement dit il s'agit de mettre en oeuvre les primitives d'une pile

Le langage Python avec le type `list` propose une implémentation toute faite d'une pile

5.1.1. Utilisation des listes comme des piles

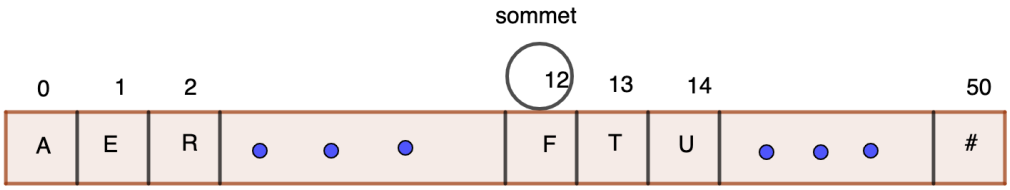
Les méthodes des listes rendent très facile leur utilisation comme des piles, où le dernier élément ajouté est le premier récupéré (« dernier entré, premier sorti » ou LIFO pour *last-in, first-out* en anglais). Pour ajouter un élément sur la pile, utilisez la méthode append(). Pour récupérer l'objet au sommet de la pile, utilisez la méthode pop() sans indicateur de position. Par exemple :

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Regardons d'autres implémentations possibles

6.1 Implémentation d'une pile avec un tableau

Pour les langages comme Java



1. On réserve un tableau d'assez grande taille (50 sur l'exemple ci-dessus)
2. Au fur et à mesure qu'on empile et dépile on mémorise le sommet de la pile par une variable `sommet` de type entier

   Les éventuelles valeurs qui sont placées après la valeur de `sommet` ne sont plus significatives

Voici l'implémentation de la primitive `pilevide(P)`

Algorithme 4 : pilevide

```
début
    /* Définition de la fonction pile-vide(T) */
1   pilevide (T,sommet-pile)
2   début
        Données : Un tableau T, un entier sommet-pile
        Résultat : un booléen Vrai si la pile est vide,
                   Faux sinon
3       si sommetpile = -1 alors
4           retourner Vrai
5       sinon
6           retourner Faux
7       fin
8   fin
fin
```

Voici l'implémentation de la primitive `empiler(P,x)`

Algorithme 5 : empiler(T,x,sommetPile)

```
début
    /* Définition de la fonction empiler(T)   */
1   empiler (T,x,sommetPile)
    Données : Un tableau T, une valeur x,un entier
              sommetPile
    Résultat : Rien
2   début
3       sommetPile ← sommetPile + 1
4       T[sommetPile] ← x
5   fin
fin
```

6.2 Implémentation d'une pile avec un deux files

On doit simuler une pile P à l'aide de deux files F1 et F2

Comment ?

La file F1 ne contient que le sommet de la pile, si la pile F1 est vide cela signifie que la pile est vide

La file F2 contient le reste de la pile

Cependant pour chaque opération empiler et dépiler il faut maintenir l'équilibre défini ci-dessus

Voici l'implémentation de la primitive `pile-vide(P)`

Algorithme 6 : PileVide

```
début
    /* Définition de la fonction pileVide(F1,F2)
       */
1   pileVide (P)
2   début
        Données : Une file F1, une file F2
        Résultat : un booléen Vrai si la pile est vide,
                   Faux sinon
3       retourner file-vide(F1)
4   fin
fin
```

Voici l'implémentation de la primitive `empiler(P,x)`

Algorithme 7 : empiler(F1,F2,x)

```
début
    /* Définition de la fonction empiler()    */
1   empiler (F1,F2,x)
2   début
        Données : Deux files F1 et F2, une valeur x
        Résultat : Rien
3       si non(file-vide(F1)) alors
4           enfiler(F2,defiler(F1))
5       fin
6       enfiler(F1,x)
7   fin
fin
```

Voici l'implémentation de la primitive `depiler(P)`

Algorithme 8 : depiler(F1,F2)

```
   début
       /* Définition de la fonction depiler()     */
1      depiler (F1,F2)
2      début
           Données : Deux files F1 et F2, F1 est non vide
           Résultat : Le sommet de la pile
3          x ← defiler(F1)
           /* On défile F2 dans F1                 */
4          nbElts ← 0
5          tant que non (est-vide(F2)) faire
6              enfiler(F1,defiler(F2))
7              nbElts ← nbElts + 1
8          fin
           /* On défile F1 dans F2 en laissant le
              dernier elt de F1 dans F1            */
9          i ← 1
10         tant que i < nbElts faire
11             enfiler(F2,defiler(F1))
12             i ← i + 1
13         fin
14         retourner x
15     fin
   fin
```

7 Implémentations d'une file

7.1 Implémentation d'une file avec un tableau

Voir TP en Python

8 Exercices

Ex 1

Nous avons commencé à implémenter une classe `Liste` dans le cours sur la programmation objet (dans le fichier `struct.py`)

On veut modifier l'implémentation de telle sorte que la méthode `supprimer` de la classe `Liste` ait en paramètre non pas une adresse mémoire sur la cellule à supprimer, mais la valeur que l'on veut supprimer.

Si la valeur n'est pas dans la liste, une exception `ValueError` est levée.

```
class Cellule:
    def __init__(self,v,s):
        self.val = v
        self.succ = s

# fonctions
def rechercher(liste:Cellule,val:int)-> Cellule:
    """
    renvoie un pointeur (adresse mémoire)
    sur le premier élément de valeur val
    dans la liste. S'il n'y a pas d'élément
    ayant pour valeur val on renvoie None
    """

def inserer(liste:Cellule,val:int)-> Cellule:
    """
    inserer en tête de liste
    """

def supprimer(liste:Cellule,adr:Cellule)-> Cellule:
```

```
    """
    élimine de la liste non vide un élément
    repéré par un pointeur (adr)
    """

class Liste:
    def __init__(self):
        self.tete = None

    def rechercher(self,val):
        return rechercher(self.tete,val)

    def inserer(self,val):
        self.tete =  inserer(self.tete,val)

    def supprimer(self,adr):
        self.tete = supprimer(self.tete,adr)
```

1. Modifier l'en-tête de la méthode `supprimer` de la classe `Liste` en remplaçant le paramètre `adr` par `val`.
   De même dans le corps de la méthode remplacer `adr` par `val` dans l'appel de la fonction `supprimer`. Suite à cette modification d'un des paramètres de la fonction `supprimer` il faut donc réécrire la fonction `supprimer`.
2. On met le symbole `_` devant l'ancienne fonction `supprimer` pour signifier que c'est une méthode interne à la classe que l'utilisateur n'est pas sensé utilisé. (Dans des langages objets comme Java on distingue des méthodes privées non utilisables par l'usager et des méthodes publiques utilisables par l'usager)
3. Ecrire une nouvelle fonction `supprimer(liste,val)` qui

utilise `_supprimer(liste,adr)` et la fonction `rechercher`. Une exception `ValueError` est levée si `val` n'est pas dans `liste`

Ex 2

Ecrire en pseudo-code une fonction `rechercher(L,x)` qui renvoie vrai si x est dans la liste L et faux sinon.

Ex 3

En partant de la pile vide P implémentée dans un tableau de longueur 6, faire un dessin montrant l'évolution de la pile P après chacune des opérations `empiler(P,3)`, `empiler(P,2)`, `empiler(P,1)`, `depiler(P)`, `empiler(P,5)` et `depiler(P)`

Ex 4

Définir en Python une classe Pile ayant un seul attribut `sommet` de type `Cellule` (fichier struct.py) et dont les méthodes sont les primitives d'une pile

Ex 5

En partant de la file vide F implémentée dans un tableau de longueur 6, faire un dessin montrant l'évolution de la file F après chacune des opérations `enfiler(F,3)`, `enfiler(F,2)`, `enfiler(F,1)`, `defiler(F)`, `enfiler(F,5)` et `defiler(F)`

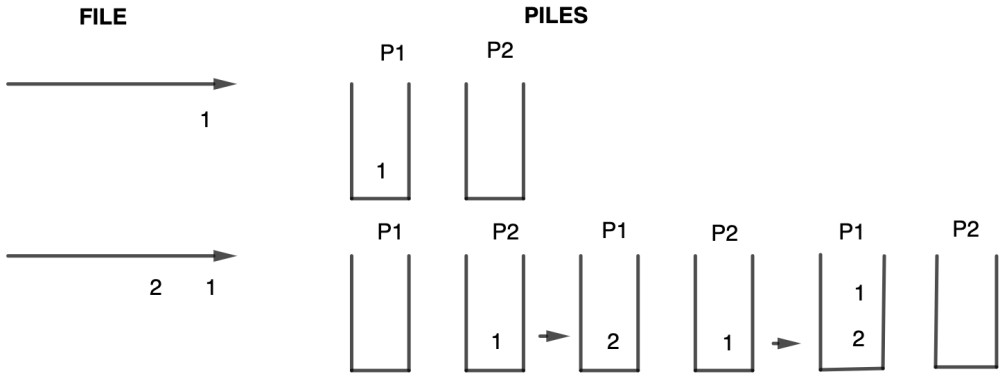
Ex 6

Définir en Python une classe `File` (à partir du type `Cellule_2` pour créer des listes doublement chainées voir le fichier struct.py)

Réfléchir aux attributs afin que les primitives aient une complexité la moins élevée possible.

Ex 7

Implémenter une file avec deux piles



Ex 8

Inverser l'ordre des éléments dans une pile P à l'aide de deux piles Q et R

Ex 9

Dans cet exercice on s'intéresse à des piles contenant des lettres minuscules

1. Deux piles de lettres sont dites égales si elles contiennent les même lettres dans le même ordre d'empilement ainsi si on utilise une liste Python pour implémenter une pile le sommet de la pile est le dernier élément de la liste

   Par exemple P = ["i","n","f","o"] et Q = ["i","n","f","o"] sont égales mais P et R = ["o","f","n","i"] ne sont pas égales

   Ecrire en pseudo-code une fonction `sont_egales(P,Q)` qui renvoie vrai si P et Q sont égales et faux sinon (on ne peut utiliser que les primitives de l'interface)

2. On ajoute une primitive supplémentaire `peek()` qui retourne le sommet de la pile **sans l'enlever de la pile**

   En utilisant cette nouvelle primitive écrire une fonction `est_palindrome(P)` qui renvoie vrai si les lettres empilées dans P forment un palindrome

Par exemple P = ["r","a","d","a","r"] est un palindrome

Ex 10 (évaluation d'une expression arithmétique postfixée)

Pour cet exercice on s'intéresse à l'évaluation d'une expression arithmétique comme 5 + 2 ou (5 + 2) * 3
Evaluer une expression arithmétique comme 5 + 2 c'est lui associer le résultat du calcul ici 7

Pour réaliser cela on va supposer que les expressions sont sous la forme postfixée, c'est à dire que l'opérateur + est écrit après les nombres 5 et 2 Ainsi 5 + 2 est écrit dans une liste sous la forme [5, 2 ,'+'] et (5 + 2) * 3 devient [5,2,'+',3,'*']
Regardons sur un exemple
Il s'agit d'évaluer (5 + 2) * 3 en partant de la liste [5,2,'+',3,'*'] , voici l'algorithme :

1. On parcourt la liste
2. Si l'élément lu est un nombre on l'empile sur une pile P
3. Si l'élément lu est un opérateur parmi '+','-','*' on dépile P deux fois et on calcule le résultat de l'opération associée à l'opérateur et appliquée aux deux éléments dépilés.
4. Ce résultat est empilé sur la pile P
5. A la fin du parcours de la liste, il reste une seule valeur dans la Pile qui est le résultat du calcul

Calculer le résultat de l'opération associé à l'opérateur se fait par l'intermédiaire d'un dictionnaire `op` qui fait l'association entre le symbole de l'opérateur par exemple '+' de type str et la fonction associée appelée somme

Ecrire une fonction `eval(expr:str)->int`

Ex 10

Ecrire une fonction Python qui affiche vrai si une chaîne de caractères contenant uniquement des parenthèses, des accolades et des crochets est équilibrée

Par exemple la fonction retourne vrai pour [()][()()]() et faux pour [(]).

BAC

1. Ex 1 Sujet 0 Métropole 2021 (Pile)
2. Ex 5 Amérique du Nord 2021 (File)