Aix*Marseille université
Initiative d'excellence
Année 2019-20

Diplôme Inter-Universitaire
Enseigner l'informatique au lycée

Bloc 4 : TP 8

Structures de données : listes, piles, files

Dans ce TP, on va dans un premier temps implémenter les structures de données pile et file à partir des listes natives Python. On utilisera ensuite ces structures pour résoudre quelques problèmes algorithmiques. Dans un deuxième temps, au lieu de se reposer sur la structure des listes en Python, on codera nous-même les listes chaînées. On constatera qu'avec des implémentations de même interface d'une même spécification, les deux implémentations des listes résolvent de façon équivalente les problèmes algorithmiques considérés. On finira par quelques extensions des problèmes d'ordonnancement vus précédemment.

1 Reprise brève de la POO : listes

On vous fournit une classe `Liste` avec :
- un constructeur `__init__` initialisant l'attribut `contenu` à `[]` (liste vide Python). Cet attribut `contenu` contiendra la liste modélisée par l'objet de type `Liste`.
- une méthode `est_vide(self)` qui renvoie `True` si la liste est vide et `False` sinon
- une méthode `longueur(self)` qui renvoie le nombre d'éléments de la liste
- une méthode `ajout_debut(self,element)` ajoutant `element` en position 0 de la liste et décalant les autres entrées
- une méthode `ajout_fin(self,element)` ajoutant `element` en dernière position de la liste
- une méthode `enleve_debut(self)` qui enlève l'élément en position 0 et le retourne. Si la liste est vide, on renvoie `None` et on affiche une erreur.
- une méthode `enleve_fin(self)` qui enlève l'élément en dernière position et le retourne. Si la liste est vide, on renvoie `None` et on affiche une erreur.
- une méthode `__repr__(self)` renvoyant une chaîne représentant la liste

Ces méthodes constituent **l'interface** de la classe `Liste`, et les explications constituent la **spécification** de ces méthodes. On verra ultérieurement dans le TP que l'implémentation de ces méthodes n'a pas d'importance tant que l'interface ne change pas et que la spécification est respectée. Ici, l'implémentation de la classe `Liste` se base sur les listes natives en Python, mais on verra plus tard qu'on peut implémenter une classe de même interface et de même spécification en se basant sur des *listes chaînées*.

2 Piles

2.1 La classe Pile

Exercice : En se basant sur la classe `Liste`, créer une classe `Pile` avec :
- un constructeur `__init__` qui appelle le `__init__` de `Liste` pour créer un objet de type `Liste` qu'on stocke dans un attribut `contenu`
- une méthode `est_vide(self)` qui renvoie `True` si la pile stockée dans `self.contenu` est vide et `False` sinon. Pour mémoire, cette pile est de type Liste, on peut donc utiliser les méthodes de Liste (remarque valable aussi pour la suite).
- une méthode `profondeur(self)` qui renvoie le nombre d'éléments de la pile
- une méthode `push(self,element)` ajoutant `element` en haut de la pile
- une méthode `pop(self)` qui :
  - retourne None et affiche un message d'erreur si la pile est vide
  - si la pile n'est pas vide, enlève l'élément au sommet de la pile et le renvoie
- une méthode `__repr__(self)` renvoyant une chaîne représentant la pile

2.2 Renversement de pile

Exercice : Ajouter à la classe `Pile` une méthode `retourner()` qui renvoie le renversement de la pile considérée. Mettons que l'objet `p` de la classe `Pile` représente la pile contenant, en partant du haut de la pile, les lettres a, b et c. Alors `p.renverser()` renverra une nouvelle pile contenant, en partant du haut de la pile, les lettres c, b et a.

2.3 Pile et palindromes

Un palindrome est un mot qui se lit de la même façon de gauche à droite et de droite à gauche. Par exemple, `abababa` est un palindrome, `kayak` et `coloc` en sont deux autres issus de Wikipedia. Par opposition, `abababab` n'est pas un palindrome.

Les piles sont des structures très utiles pour détecter les palindromes : on peut lire le mot jusqu'à sa moitié, et empiler les lettres qu'on lit, puis arrivé à la moitié on lit les lettres tout en dépilant et en regardant si le résultat du dépilage correspond à la lettre lue. Si ce n'est pas le cas, le mot en entrée n'est pas un palindrome. S'il y a toujours égalité, c'est un palindrome.

Il faut faire attention à distinguer les mots de longueur paire et impaire. Si le mot est pair, de longueur `2n`, on lit `n` lettres en empilant, puis `n` lettres en dépilant. Si le mot est impair, de longueur `2n+1`, on lit `n` lettres en empilant, on lit la lettre du milieu sans rien faire, puis on lit `n` lettres en dépilant.

Exercice : implémenter à l'aide d'une pile un programme qui prend en entrée un mot et renvoie `True` si c'est un palindrome et `False` sinon.

3 Files

3.1 La classe File

Exercice : En se basant sur la classe `Liste`, créer une classe `File` avec :

- un constructeur `__init__` qui appelle le `__init__` de `Liste` pour créer un objet de type `Liste` qu'on stocke dans un attribut `contenu`
- une méthode `est_vide(self)` qui renvoie `True` si la file stockée dans `self.contenu` est vide et `False` sinon. Pour mémoire, cette file est de type Liste, on peut donc utiliser les méthodes de Liste (remarque valable aussi pour la suite).
- une méthode `longueur(self)` qui renvoie le nombre d'éléments de la file
- une méthode `enfile(self,element)` ajoutant `element` à la fin de la file
- une méthode `defile(self)` qui :
  - retourne None et affiche un message d'erreur si la file est vide
  - si la file n'est pas vide, enlève l'élément au début de la file et le renvoie
- une méthode `__repr__(self)` renvoyant une chaîne représentant la file

3.2 Une file avec deux piles

On peut coder une file à l'aide de deux piles `P1` et `P2` ; `P1` est dite pile de fin et `P2` pile de début. On procède comme suit :

- pour simuler un ajout à la file, on empile sur la pile de fin `P1`
- pour simuler un retrait de la file :
  - si la pile de début `P2` n'est pas vide, on dépile depuis `P2` et on retourne le résultat,
  - si `P2` est vide et que `P1` n'est pas vide, on renverse `P1` sur `P2`, puis on dépile depuis `P2` et on retourne le résultat,
  - si `P1` et `P2` sont vides, la file est vide et il y a une erreur. On peut par exemple retourner `None` et afficher un message d'erreur.

Exercice : implémenter une classe `File` à l'aide de la classe `Pile` en suivant cette méthodologie.

3.3 File et ordonnancement

L'ordonnancement consiste, pour le système d'exploitation, à optimiser l'utilisation du processeur en lui affectant tour à tour différentes tâches à exécuter. On appelle *processus* un programme en cours d'exécution. Il peut y en avoir des centaines à la fois sur une machine, alors qu'il n'y a que quelques processeurs (souvent 4). L'ordonnanceur va répartir le temps de calcul entre les programmes, afin que tous puissent avancer dans leur exécution de manière satisfaisante, et que les programmes qui n'ont pas besoin de temps processeur à un certain moment (par exemple parce qu'ils attendent une réponse de l'utilisateur avant de continuer) ne gaspillent pas de temps de calcul.

La plupart des ordonnanceurs modernes utilisent des files pour garder en mémoire de façon optimale les programmes à exécuter. En effet, tout comme la pile était une structure naturelle pour gérer les palindromes à l'exercice précédent, la file est parfaitement adaptée à l'ordonnancement : les programmes qui demandent du temps de calcul sont insérés en bout de file, et ceux qui seront défilés pour obtenir effectivement du temps processeur sont ceux qui attendent depuis le plus longtemps.

Exercice :

— créer une classe `Activite` pour modéliser des activités ayant trois attributs : `nom`, `duree` et `priorite`. On programmera les méthodes `__init__` et `__repr__` appropriées.

— créer une classe `Ordonnanceur` sur le patron suivant :

```
class Ordonnanceur:
    def __init__(self):
        self.file = File()


    def ajout_activite(self,activite):
        # à remplir

    def step(self):
        # à remplir

    def run(self):
        # à remplir
```

— remplir la méthode `ajout_activite` qui ajoute une activité passée en paramètre à la file de processus de l'ordonnanceur

— remplir la méthode `step` qui effectue un "tour" d'ordonnancement comme suit : si la file est vide, on le dit et on ne fait rien. S'il y a au moins une activité dans la file, on défile et on exécute l'activité en affichant son nom et sa durée et en attendant le temps correspondant à la durée de l'activité.

— remplir la méthode `run` qui itère `step` jusqu'à obtenir une file de processus vide

— créer une liste de 10 activités de durée et de priorité aléatoires (durée entre 1 et 10 et priorité entre 0 et 2, la priorité n'étant de toute façon utilisée qu'en fin de TP)

— à l'aide d'une boucle, mettre toutes les activités dans la file de l'ordonnanceur puis exécuter l'ordonnanceur

— *Bonus* : ajouter à `step` un tirage probabiliste qui ajoute une nouvelle activité aléatoire avec probabilité 3/10, puis exécuter à nouveau l'ordonnanceur sur un exemple.

4 Listes chaînées

4.1 La liste chaînée

On veut implémenter une classe `ListeChainee` de même interface et de même spécification que `Liste`, mais dont l'implémentation soit directement basée sur celle de la liste chaînée.

Chaque cellule de la liste est modélisée par un objet de la classe `Noeud` qui suit :

```
class Noeud:
    def __init__(self,suivant,contenu):
        self.contenu = contenu
        self.suivant = suivant

    def __repr__(self):
        # Suppose que le contenu est transformable en chaîne de caractères
        return str(self.contenu)
```

Chaque noeud contient donc une valeur, son attribut `contenu`, et un pointeur vers le noeud suivant, stocké dans l'attribut `suivant`. En fin de liste, cet attribut vaut `None` : il n'y a pas de noeud suivant donc le pointeur est nul.

On a commencé l'implémentation de la classe comme suit :

```
class ListeChainee:
    def __init__(self):
        self.debut = None

    def ajout_debut(self,valeur):
        self.debut = Noeud(self.debut,valeur)

    def ajout_fin(self,valeur):
       # à remplir ainsi que les autres méthodes de l'interface
```

Exercice : terminer l'implémentation de la liste chaînée, en respectant la même interface et la même spécification que pour la classe `Liste` du début du TP.

4.2 Piles et chaînage

On a défini précédemment la classe `Pile` à l'aide de la classe `Liste`. Comme expliqué en cours, comme les classes `Liste` et `ListeChainee` ont même interface et même spécification, elles sont interchangeables. On va le vérifier.

Exercice :

— reprendre la classe `Pile` et remplacer l'initialisation dans `__init__`, qui utilisait `Liste`, par une initialisation d'un objet de type `ListeChainee`

— constater sur quelques exemples qu'on a toujours une pile et qu'il n'y a rien eu d'autre à remplacer. Pourquoi a-t-il suffi de changer une ligne et rien de plus ?

— *Bonus* : reprendre l'exercice sur les palindromes avec cette nouvelle implémentation de la pile. Que faut-il modifier ? Constater que tout fonctionne comme auparavant : l'algorithme détecte toujours bien les palindromes.

5 Améliorer l'ordonnancement avec des files

5.1 Ordonnancement préemptif, la stratégie "round-robin"

L'ordonnancement à l'aide d'une file vu précédemment est dit *coopératif* : l'ordonnanceur laisse à chaque processus tout le temps dont il a besoin. Dans une telle stratégie, c'est le processus qui doit rendre la main de lui-même, parce qu'il a terminé ou parce qu'il se met en attente (on n'a pas modélisé cette éventualité). Dans les systèmes d'exploitation récents, l'ordonnancement est *préemptif* : au bout d'un certain temps, si le

processus auquel est actuellement alloué le temps processeur n'a pas terminé, on le met en attente et on alloue le processeur à un autre processus.

Exercice : reprendre votre ordonnanceur pour qu'il prenne à l'initialisation un paramètre `tmax` dénotant le temps maximal laissé à un processus. Modifier ensuite la méthode `step` pour qu'à chaque étape elle considère une activité et :

— soit le temps d'exécution est inférieur à `tmax` et on exécute l'activité que l'on a défilée
— soit ce temps d'exécution est strictement supérieur à `tmax`, alors on exécute l'activité défilée pendant `tmax`, puis on la renfile avec un temps d'exécution diminué de `tmax`

Si vous n'avez pas implémenté le *Bonus* de l'exercice sur l'ordonnancement précédent, il est temps de le faire.

5.2 Ordonnancement multifile : les priorités

Dans les systèmes d'exploitation modernes, il y a une notion de priorité : certains processus plus critiques doivent être exécutés avant les autres. On suppose ici que les activités sont des entiers entre 0 et 2 (0 est la plus haute priorité, 2 la plus faible). Une stratégie approchant ce qui se fait dans les systèmes modernes est la suivante :

— on crée une file pour chaque priorité 0, 1 et 2
— à l'ajout dans l'ordonnanceur, on ajoute l'activité à la bonne file
— la fonction "step" de l'ordonnanceur se comporte comme un round-robin, mais qui commence sur la file de priorité 0. Si elle est vide, il cherche une tâche dans la file de priorité 1. Si elle est vide, on regarde la file de priorité 2.

Exercice : Implémenter un tel ordonnanceur et le faire tourner sur un exemple. Il sera intéressant, dans l'esprit des *Bonus* précédents, d'avoir des activités qui soient ajoutées avec une certaine probabilité et dont la priorité sera aléatoire. Observez notamment ce qui se passe si une file est vide, par exemple celle de priorité 0, et qu'une nouvelle activité de priorité 0 est ajoutée.