Thème 1 - Structure de données

BAC Listes, Piles et Files

1. D'après 2022, Métropole, J1, Ex. 1 - Vérification syntaxique de parenthèses ou de balises

D'après 2022, Métropole, J1, Ex. 1

1.1. Partie A : Expression correctement parenthésée

On veut déterminer si une expression arithmétique est correctement parenthésée. À chaque parenthèse fermante ")" correspond une parenthèse précédemment ouverte "(".

Exemples

- L'expression arithmétique `"(2 + 3) × (18/(4 + 2))"` est correctement parenthésée.
- L'expression arithmétique `"(2 + 3) × (18/(4 + 2"` est non correctement parenthésée.

Pour simplifier les expressions arithmétiques, on enregistre, dans une structure de données, uniquement les parenthèses dans leur ordre d'apparition. On appelle expression simplifiée cette structure.

Expression arithmétique	Structure de données
`"(2 + 3) × (18/(4 + 2))"`	`()(())`

1. Indiquer si la phrase « les éléments sont maintenant retirés (pour être lus) de cette structure de données dans le même ordre qu'ils y ont été ajoutés lors de l'enregistrement » décrit le comportement d'une file ou d'une pile. Justifier.

Réponse

Le premier à être retiré était le premier à être ajouté, donc cela correspond à une **file**.

Pour vérifier le parenthésage, on peut utiliser une variable `controleur` qui :

- est un nombre entier égal à 0 en début d'analyse de l'expression simplifiée ;
- augmente de 1 si l'on rencontre une parenthèse ouvrante `"("` ;
- diminue de 1 si l'on rencontre une parenthèse fermante `")"`.

Exemple

On considère l'expression simplifiée A : `"()(())"`

Lors de l'analyse de l'expression A, `controleur` (initialement égal à 0) prend successivement pour valeur 1, 0, 1, 2, 1, 0.

Le parenthésage est correct.

2. Écrire, pour chacune des 2 expressions simplifiées B et C suivantes, les valeurs successives prises par la variable `controleur` lors de leur analyse.

- Expression simplifiée B : " ((()()"
- Expression simplifiée C : "(()))("

Réponse

- Expression simplifiée B : **1, 2, 3, 2, 3, 2**
- Expression simplifiée C : **1, 2, 1, 0, -1, 0**

3. L'expression simplifiée B précédente est mal parenthésée (parenthèses fermantes manquantes) car le `controleur` est différent de zéro en fin d'analyse.
L'expression simplifiée C précédente est également mal parenthésée (parenthèse fermante sans parenthèse ouvrante) car le `controleur` prend une valeur négative pendant l'analyse.

Recopier et compléter uniquement les lignes 13 et 16 du code ci-dessous pour que la fonction `parenthesage_correct` réponde à sa description.

Script Python

```
1  def parenthesage_correct(expression):
2      """ fonction renvoyant True si l'expression arithmétique
3      simplifiée (str) est correctement parenthésée, False sinon.
4      Condition: expression ne contient que
5        des parenthèses ouvrantes et fermantes
6      """
7      controleur = 0
8      for parenthese in expression:  # pour chaque parenthèse
9          if parenthese == '(':
10             controleur = controleur + 1
11         else:  # parenthese == ')'
```

```
12              controleur = controleur - 1
13              if controleur ... :  # test 1 (à recopier et compléter)
14                  # parenthèse fermante sans parenthèse ouvrante
15                  return False
16      return controleur ...  # test 2 (à recopier et compléter)
17      # test 2 est un booléen renvoyé
18      #   True : le parenthésage est correct
19      #   False : parenthèse(s) fermante(s) manquante(s)
```

Réponse

- ligne 13: `(controleur < 0)`
- ligne 16: `(controleur == 0)`

Les parenthèses sont inutiles.

1.2. Partie B : Texte correctement balisé

On peut faire l'analogie entre le texte simplifié des fichiers HTML (uniquement constitué de balises ouvrantes `<nom>` et fermantes `</nom>`) et les expressions parenthésées : par exemple, l'expression HTML simplifiée : `"<p><strong><em></em></strong></p>"` est correctement balisée.

On ne tiendra pas compte dans cette partie des balises ne comportant pas de fermeture comme `<br>` ou `<img ...>`.

Afin de vérifier qu'une expression HTML simplifiée est correctement balisée, on peut utiliser une pile (initialement vide) selon l'algorithme suivant :

- On parcourt successivement chaque balise de l'expression :
  - lorsque l'on rencontre une balise ouvrante, on l'empile ;
  - lorsque l'on rencontre une balise fermante :
    - si la pile est vide, alors l'analyse s'arrête : le balisage est incorrect,
    - sinon, on dépile et on vérifie que les deux balises (la balise fermante rencontrée et la balise ouvrante dépilée) correspondent (c'est-à-dire ont le même nom) si ce n'est pas le cas, l'analyse s'arrête (balisage incorrect).

Exemple

État de la pile lors du déroulement de cet algorithme pour l'expression simplifiée `"<p><em></p></em>"` qui n'est pas correctement balisée.

État de la pile lors du déroulement de l'algorithme

Départ

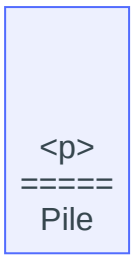
Parcours de l'expression

```
"<p><em></p></em>"
↑
```

Étape 1

Parcours de l'expression

```
"<p><em></p></em>"
  ↑                 Balise <p>  ouvrante, on empile
```



Étape 2

Parcours de l'expression

```
"<p><em></p></em>"
     ↑              Balise <em> ouvrante, on empile
```

<p>
=====
Pile

Étape 3

Parcours de l'expression

```
"<p><em></p></em>"
          ↑          Balise </p> fermante, on dépile
```

<p>
=====
Pile

`<em>` et `</p>` ne correspondent pas !

Le balisage est incorrect.

4. Cette question traite de l'état de la pile lors du déroulement de l'algorithme.

4.a. Représenter la pile à chaque étape du déroulement de cet algorithme pour l'expression `"<p><em></em></p>"` (balisage correct).

Réponse

Départ

Parcours de l'expression

```
"<p><em></em></p>"
↑
```

Étape 1

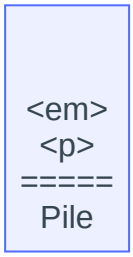
Parcours de l'expression

```
"<p><em></em></p>"
  ↑                 Balise  <p>  ouvrante, on empile
```

Étape 2

Parcours de l'expression

```
"<p><em></em></p>"
      ↑             Balise  <em> ouvrante, on empile
```



Étape 3

Parcours de l'expression

```
"<p><em></em></p>"
          ↑          Balise </em> fermante, on dépile
```

`<em>` et `</em>` se correspondent.

Étape 4

Parcours de l'expression

```
"<p><em></em></p>"
               ↑     Balise </p>  fermante, on dépile
```

`<p>` et `</p>` se correspondent.

Fin

Parcours de l'expression

```
"<p><em></em></p>"
                ↑
```

La pile est vide. Le balisage est correct.

4.b. Indiquer quelle condition simple (sur le contenu de la pile) permet alors de dire que le balisage est correct lorsque toute l'expression HTML simplifiée a été entièrement parcourue, sans que l'analyse ne s'arrête.

Réponse

Il suffirait de vérifier que la pile est **vide**.

5. Une expression HTML correctement balisée contient 12 balises.

Indiquer le nombre d'éléments que pourrait contenir au maximum la pile lors de son analyse.

Réponse

6 éléments au maximum seront empilés, dans le cas où 12 balises HTML sont imbriquées. 6 ouvrantes qui seront empilées, puis les 6 fermantes.

2. D'après 2022, Métropole, J2, Ex. 2 - Jeu de la poussette

D'après 2022, Métropole, J2, Ex. 2

---

title: Jeu de la poussette author: Nicolas Revéret

La poussette est un jeu de cartes en solitaire. Cet exercice propose une version simplifiée de ce jeu basée sur des nombres.

On considère une pile constituée de nombres entiers tirés aléatoirement. Le jeu consiste à réduire la pile suivant la règle suivante : quand la pile contient du haut vers le bas un triplet dont les termes du haut et du bas sont de même parité, on supprime l'élément central.

Par exemple :

- Si la pile contient du haut vers le bas le triplet 1 ; 0 ; 3, on supprime le 0, car 1 et 3 sont tous les deux impairs.
- Si la pile contient du haut vers le bas le triplet 1 ; 0 ; 8, la pile reste inchangée, car 1 et 8 n'ont pas la même parité.

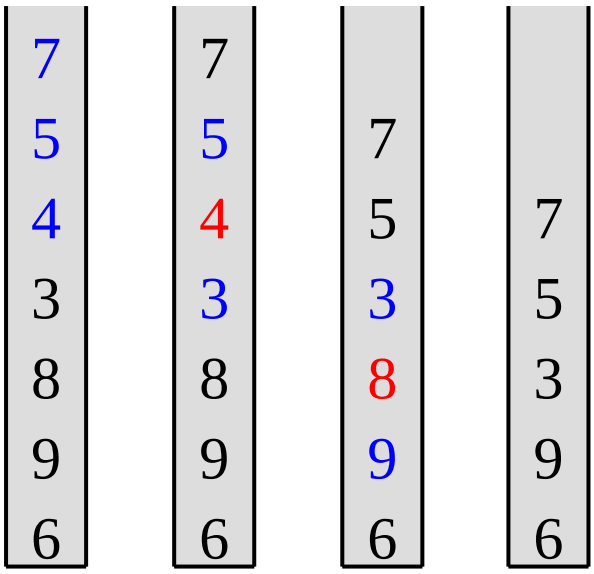
On parcourt la pile ainsi de haut en bas et on procède aux réductions.

Arrivé en bas de la pile, **on recommence la réduction** en repartant du sommet de la pile jusqu'à ce que la pile ne soit plus réductible.

Une partie est « gagnante » lorsque la pile finale est réduite à deux éléments exactement.

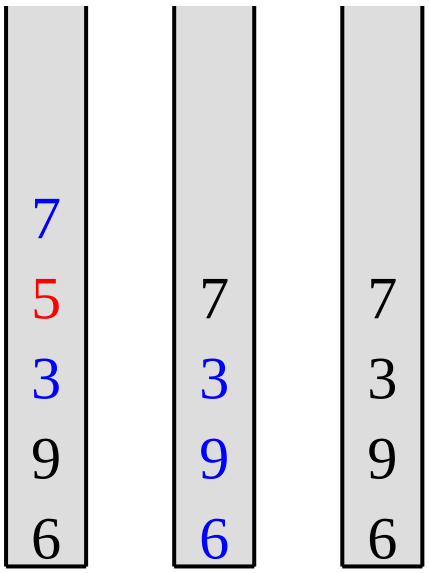
Voici un exemple détaillé de déroulement d'une partie.

Premier parcours de la pile



- La première comparaison (7, 5 et 4) laisse la pile inchangée.
- On retire le 4 lors de la deuxième itération.
- On retire le 8 lors de la troisième.
- Il ne reste plus que deux valeurs en bas de la pile (9 et 6) : on a fini le premier parcours.

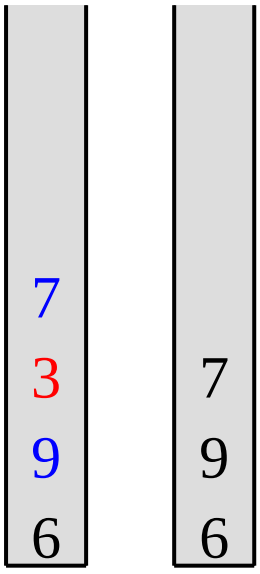
Deuxième parcours



- On recommence à partir du haut de la pile : on retire le 5.
- Le triplet suivant (3, 9 et 6) n'entraîne pas de suppression.

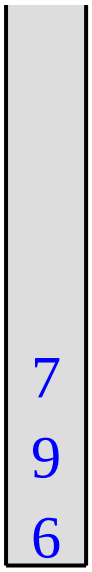
- Il ne reste plus que deux valeurs à étudier (9 et 6) : on a terminé le deuxième parcours.

Troisième parcours



- On recommence en haut de la pile avec 7, 2 et 9 : on retire le 2.
- Il ne reste que le 9 et le 6 : on a terminé le troisième parcours.

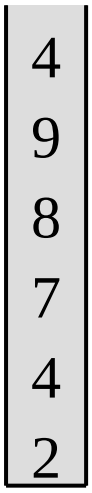
Quatrième parcours



- On recommence en haut de la pile avec 7, 9 et 6. La pile est inchangée.

- La pile n'a pas été modifiée lors de ce parcours : la partie est terminée et cette pile n'est pas gagnante.

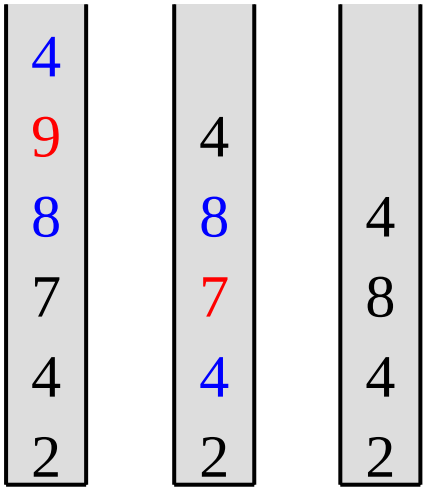
1.a. Donner les différentes étapes de réduction de la pile suivante :



Réponse

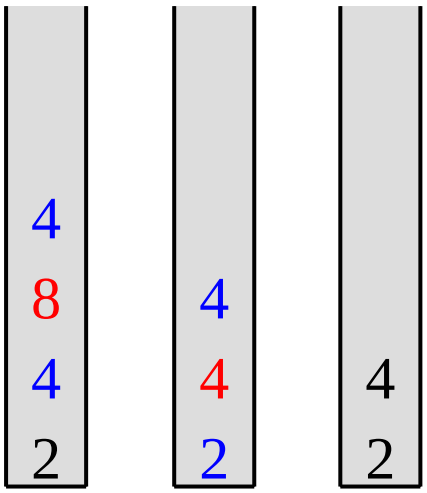
Il s'agit d'une pile gagnante :

Premier parcours



- Lors de la première comparaison, on retire le 9.
- Lors de la seconde comparaison, on retire le 7.
- Il ne reste plus que deux valeurs en bas de la pile (4 et 2) : on a fini le premier parcours.

Second parcours

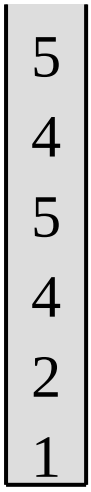


- Lors de la première comparaison, on retire le 8.

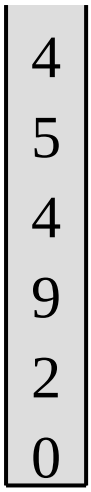
- Lors de la seconde comparaison, on retire le 4.
- Il ne reste plus que deux valeurs en bas de la pile (4 et 2) : la pile est gagnante.

1.b. Parmi les piles proposées ci-dessous, donner celle qui est gagnante.

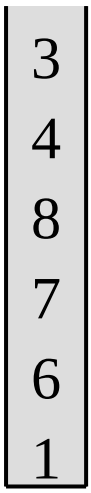
Pile A



Pile B



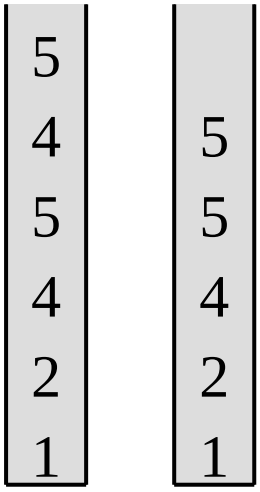
Pile C


3
4
8
7
6
1

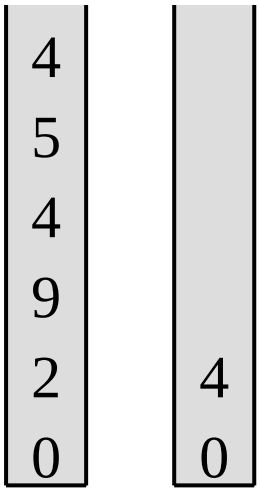
Réponse

Seule la pile B est gagnante. On fournit ci dessous les piles en début et fin de partie :

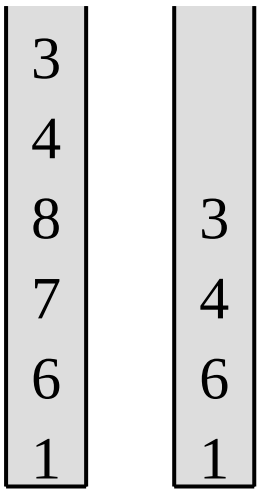
Pile A



Pile B



Pile C



L'interface d'une pile est proposée ci-dessous :

- `creer_pile_vide()` renvoie une pile vide,
- `est_vide(p)` renvoie `True` si `p` est vide, `False` sinon,
- `empiler(p, element)` ajoute `element` au sommet de `p`,
- `depiler(p)` retire l'élément au sommet de `p` et le renvoie,
- `sommet(p)` renvoie l'élément au sommet de `p` sans le retirer de `p`,
- `taille(p):` renvoie le nombre d'éléments de `p`.

Dans la suite de l'exercice on utilisera uniquement ces fonctions.

2. La fonction `reduire_triplet_au_sommet` permet de supprimer l'élément central des trois premiers éléments en partant du haut de la pile, si l'élément du bas et du haut sont de même parité. Les éléments dépilés et non supprimés sont replacés dans le bon ordre dans la pile.

Recopier et compléter sur la copie le code de la fonction `reduire_triplet_au_sommet` prenant une pile `p` en paramètre et la modifiant en place. Cette fonction renvoie le booléen `est_reduit` indiquant si le triplet du sommet a été réduit ou non.

Script Python

```
def reduire_triplet_au_sommet(p):
    haut = depiler(p)
    milieu = depiler(p)
    bas = sommet(p)
    est_reduit = ...
    if haut % 2 != ...:
        empiler(p, ...)
        ...
    empiler(p, ...)
    return ...
```

Réponse

Script Python

```
def reduire_triplet_au_sommet(p):
    haut = depiler(p)
    milieu = depiler(p)
    bas = sommet(p)
    est_reduit = True
    if haut % 2 != bas % 2:
        empiler(p, milieu)
        est_reduit = False
    empiler(p, haut)
    return est_reduit
```

3. On se propose maintenant d'écrire une fonction `parcourir_pile_en_reduisant` qui parcourt la pile du haut vers le bas en procédant aux réductions pour chaque triplet rencontré quand cela est possible.

La pile est toujours modifiée en place.

La fonction `parcourir_pile_en_reduisant` renvoie un booléen indiquant si la pile a été réduite à au moins une reprise lors du parcours.

3.a. Donner la taille minimale que doit avoir une pile pour être réductible.

Résultat

Si une pile a une taille de 2 ou moins, elle n'est pas réductible.

Si une pile est réductible, alors sa taille est supérieure ou égale à 3.

3.b. Recopier et compléter sur la copie :

Script Python

```
def parcourir_pile_en_reduisant(p):
    q = creer_pile_vide()
    reduction_pendant_parcours = False
    while taille(p) >= 3:
        if ...:
            reduction_pendant_parcours = ...
        e = depiler(p)
        empiler(q, e)
    while not est_vide(q):
        ...
        ...
    return ...
```

Résultat

Script Python

```
def parcourir_pile_en_reduisant(p):
    q = creer_pile_vide()
    reduction_pendant_parcours = False
    while taille(p) >= 3:
        if reduire_triplet_au_sommet(p):
            reduction_pendant_parcours = True
        e = depiler(p)
        empiler(q, e)
    while not est_vide(q):
        e = depiler(q)
        empiler(p, e)
    return reduction_pendant_parcours
```

4. Partant d'une pile d'entiers `p`, on propose ici d'implémenter une fonction récursive `jouer` jouant une partie complète sur la pile `p`.

On effectue donc autant de parcours que nécessaire.

Une fois la pile parcourue de haut en bas, on effectue un nouveau parcours à condition que le parcours précédent ait modifié la pile. Si à l'inverse, la pile n'a pas été modifiée, on ne fait rien, car la partie est terminée.

Script Python

```
def jouer(p):
    if parcourir_pile_en_reduisant(...):
        ...(...)
```

Résultat

Script Python

```
def jouer(p):
    if parcourir_pile_en_reduisant(p):
        jouer(p)
```

3. D'après 2022, Polynésie, J1, Ex. 4 - Tri d'une pile

D'après 2022, Polynésie, J1, Ex. 4

---

title: Tri d'une pile author: Franck Chambon

La classe `Pile` utilisée dans cet exercice est implémentée en utilisant des listes Python et propose quatre éléments d'interface :

- Un constructeur qui permet de créer une pile vide, représentée par `[]` ;
- La méthode `est_vide()` qui renvoie `True` si l'objet est une pile ne contenant aucun élément, et `False` sinon ;
- La méthode `empiler` qui prend un objet quelconque en paramètre et ajoute cet objet au sommet de la pile. Dans la représentation de la pile dans la console, cet objet apparait à droite des autres éléments de la pile ;
- La méthode `depiler` qui renvoie l'objet présent au sommet de la pile et le retire de la pile.

Exemples :

Console Python

```
>>> ma_pile = Pile()
>>> ma_pile.empiler(2)
>>> ma_pile
[2]
>>> ma_pile.empiler(3)
>>> ma_pile.empiler(50)
>>> ma_pile
[2, 3, 50]
>>> ma_pile.depiler()
50
>>> ma_pile
[2, 3]
```

La méthode `est_triee` ci-dessous renvoie `True` si, en dépilant tous les éléments, ils sont traités dans l'ordre croissant, et `False` sinon.

Script Python

```
1  def est_triee(self):
2      if not self.est_vide():
3          e1 = self.depiler()
4          while not self.est_vide():
5              e2 = self.depiler()
6              if e1 ... e2 :
7                  return False
8              e1 = ...
9      return True
```

1. Recopier sur la copie les lignes 6 et 8 en complétant les points de suspension.

Réponse

Script Python

```
def est_triee(self):
    if not self.est_vide():
        e1 = self.depiler()
        while not self.est_vide():
            e2 = self.depiler()
            if e1 > e2 :
                return False
            e1 = e2
    return True
```

On crée dans la console la pile `A` représentée par `[1, 2, 3, 4]`.

2.a. Donner la valeur renvoyée par l'appel `A.est_triee()`.

Réponse

La valeur est d'abord dépilée, puis . L'ordre n'est pas croissant, ainsi `A.est_triee()` renvoie `False`.

2.b. Donner le contenu de la pile `A` après l'exécution de cette instruction.

Réponse

`A` sera représenté par `[1, 2]`.

On souhaite maintenant écrire le code d'une méthode `depile_max` d'une pile non vide ne contenant que des nombres entiers et renvoyant le plus grand élément de cette pile en le retirant de la pile.

Après l'exécution de `p.depile_max()`, le nombre d'éléments de la pile `p` diminue donc de 1.

Script Python

```
def depile_max(self):
    assert not self.est_vide(), "Pile vide"
    q = Pile()
    maxi = self.depiler()
    while not self.est_vide():
        elt = self.depiler()
        if maxi < elt:
            q.empiler(maxi)
            maxi = ...
        else :
            ...
    while not q.est_vide():
        self.empiler(q.depiler())
    return maxi
```

3. Recopier sur la copie les lignes 9 et 11 en complétant les points de suspension.

Réponse

Script Python

```
1  def depile_max(self):
2      assert not self.est_vide(), "Pile vide"
3      q = Pile()
4      maxi = self.depiler()
5      while not self.est_vide():
6          elt = self.depiler()
7          if maxi < elt:
8              q.empiler(maxi)
9              maxi = elt
10         else :
11             q.empiler(elt)
12     while not q.est_vide():
13         self.empiler(q.depiler())
14     return maxi
```

On crée la pile `B` représentée par `[9, -7, 8, 12, 4]` et on effectue l'appel `B.depile_max()`.

4.a. Donner le contenu des piles `B` et `q` à la fin de chaque itération de la boucle `while` de la ligne 5.

Réponse

Initialisation

- `B` contient `[9, -7, 8, 12]` ;
- `q` est vide ;
- `maxi` est égal à `4`.

Juste avant le premier tour de boucle

Fin du tour 1

- `B` contient `[9, -7, 8]` ;
- `q` contient `[4]` ;
- `maxi` est égal à `12`.

Fin du tour 2

- `B` contient `[9, -7]` ;
- `q` contient `[4, 8]` ;
- `maxi` est égal à `12`.

Fin du tour 3

- `B` contient `[9]` ;
- `q` contient `[4, 8, -7]` ;
- `maxi` est égal à `12`.

Fin du tour 4

- `B` est vide ;
- `q` contient `[4, 8, -7, 9]` ;
- `maxi` est égal à `12`.

4.b. Donner le contenu des piles `B` et `q` avant l'exécution de la ligne 14.

Réponse

La dernière boucle renverse la pile `q` dans la pile `B`, ainsi, à la ligne 14 :

- `q` est vide ;
- `B` contient `[9, -7, 8, 4]`.

4.c. Donner un exemple de pile qui montre que l'ordre des éléments restants n'est pas préservé après l'exécution de `depile_max`.

Réponse

Avec une pile `B` qui contient `[3, 1, 2]`

Initialisation

- `B` contient `[3, 1]` ;
- `q` est vide ;
- `maxi` est égal à `2`.

Juste avant le premier tour de boucle

Fin du tour 1

- `B` contient `[3]` ;
- `q` contient `[1]` ;
- `maxi` est égal à `2`.

Fin du tour 2

- `B` est vide ;
- `q` contient `[1, 2]` ;
- `maxi` est égal à `3`.

La dernière boucle renverse la pile `q` dans la pile `B`, ainsi, à la ligne 14 :

- `q` est vide ;
- `B` contient `[2, 1]`.

Sans `3` dans la pile `B` initiale, on a dans l'ordre `[1, 2]` ce qui est différent de `[2, 1]` obtenu ici avec `depile_max`.

On a ainsi un exemple où l'ordre des éléments restants n'est pas préservé après l'exécution de `depile_max`.

On donne le code de la fonction `traiter` :

Script Python

```
1  def traiter(self):
2      q = Pile()
3      while not self.est_vide():
4          q.empiler(self.depile_max())
5      while not q.est_vide():
6          self.empiler(q.depiler())
```

5.a. Donner les contenus successifs des piles `B` et `q`

- avant la ligne 3,
- avant la ligne 5,
- à la fin de l'exécution de la fonction `traiter` lorsque la fonction `traiter` est appelée avec la pile `B` contenant `[1, 6, 4, 3, 7, 2]`.

Réponse

Avec `B = [1, 6, 4, 3, 7, 2]`, un appel `B.traiter()` conduit successivement à :

- Avant la ligne 3,
  - `B` contient `[1, 6, 4, 3, 7, 2]` ;
  - `q` est vide.
- Avant la ligne 5,
  - `B` est vide ;
  - `q` contient `[7, 6, 4, 3, 2, 1]`
- À la fin,
  - `B` contient `[1, 2, 3, 4, 6, 7]`
  - `q` est vide.

5.b. Expliquer le traitement effectué par cette méthode.

Réponse

Ce traitement est un tri de la pile. On construit d'abord `q` comme la pile des éléments de `self` dans l'ordre décroissant. On renverse ensuite la pile, qui se retrouve comme si on avait empilé les éléments de `self` dans l'ordre croissant.

Attention, il s'agit de l'ordre inverse de celui proposé par la fonction `est_triee` vu à la question **1**. ici, si on dépile les éléments, ils sont désormais dans l'ordre décroissant.

D'après 2022, Centres étrangers, J1, Ex. 3

---

title: Dictionnaire modélisant le contenu d'un répertoire author: Franck Chambon

---

Afin d'organiser les répertoires et les fichiers sur un disque dur, une structure arborescente est utilisée. Les fichiers sont dans des répertoires qui sont eux-mêmes dans d'autres répertoires, etc.

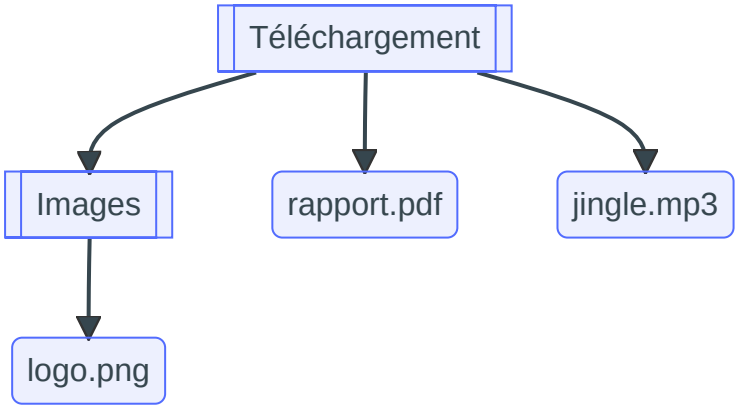
Dans une arborescence, chaque répertoire peut contenir des fichiers et des répertoires, qui sont identifiés par leur nom. Le contenu d'un répertoire est modélisé par la structure de données dictionnaire. Les clés de ce dictionnaire sont des

chaines de caractères donnant le nom des fichiers et des répertoires contenus.

Exemple illustré

Le répertoire appelé `Téléchargements` contient deux fichiers `rapport.pdf` et `jingle.mp3` ainsi qu'un répertoire `Images` contenant simplement le fichier `logo.png`.

Il est représenté ci-dessous.



Ce répertoire `Téléchargements` est modélisé en Python par le dictionnaire suivant :

```
{"Images": {"logo.png": 36}, "rapport.pdf": 450, "jingle.mp3": 4800}
```

Les valeurs numériques sont exprimées en ko (kilo-octets).

`"logo.png": 36` signifie que le fichier `logo.png` occupe un espace mémoire de 36 ko sur le disque dur.

On rappelle, ci-dessous, quelques commandes sur l'utilisation d'un dictionnaire :

- `dico = dict()` crée un dictionnaire vide appelé `dico`,
- `dico[cle] = contenu` met la valeur `contenu` pour la clé `cle` dans le dictionnaire `dico`,
- `dico[cle]` renvoie la valeur associée à la clé `cle` dans le dictionnaire `dico`,
- `cle in dico` renvoie un booléen indiquant si la clé `cle` est présente dans le dictionnaire `dico`.
- `for cle in dico:` permet d'itérer sur les clés d'un dictionnaire.
- `len(dico)` renvoie le nombre de clés d'un dictionnaire.

L'**adresse** d'un fichier ou d'un répertoire correspond au nom de tous les répertoires à parcourir depuis la racine afin d'accéder au fichier ou au répertoire. Cette adresse est modélisée en Python par la liste des noms de répertoire à parcourir pour y accéder.

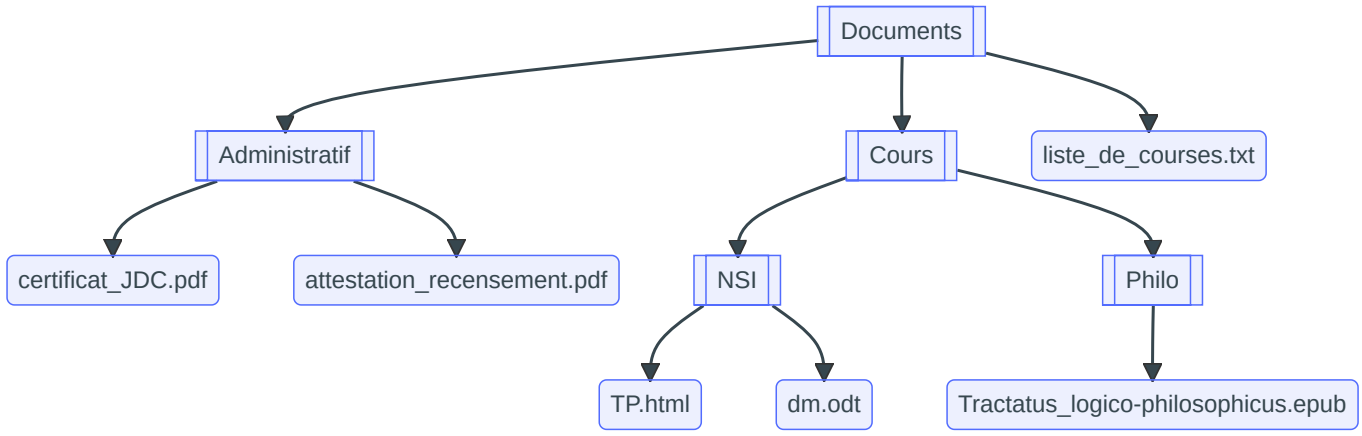
Exemple : L'adresse du répertoire : `/home/pierre/Documents/` est modélisée par la liste `["home", "pierre", "Documents"]`.

1. Dessiner l'arbre donné par le dictionnaire suivant, qui correspond au répertoire `Documents`.

Script Python

```
Documents = {
    "Administratif":{
        "certificat_JDC.pdf": 1500,
        "attestation_recensement.pdf": 850
    },
    "Cours": {
        "NSI": {
            "TP.html": 60,
            "dm.odt": 345
        },
        "Philo": {
            "Tractatus_logico-philosophicus.epub": 2600
        }
    },
    "liste_de_courses.txt": 24
}
```

Réponse



2. On donne la fonction `parcourt` suivante qui prend en paramètres un répertoire racine et une liste représentant une adresse, et qui renvoie le contenu du répertoire cible correspondant à l'adresse.

Exemple : Si la variable `docs` contient le dictionnaire de l'exemple de la question 1 alors `parcourt(docs, ["Cours", "Philo"])` renvoie le dictionnaire `{"Tractatus_logico-philosophicus.epub": 2600}`.

2.a. Recopier et compléter la ligne 4

Script Python

```
def parcourt(racine, adr):
    repertoire = racine
```

```
    for nom_repertoire in adr:
        repertoire = ...
    return repertoire
```

Réponse

Script Python

```
def parcourt(racine, adr):
    repertoire = racine
    for nom_repertoire in adr:
        repertoire = repertoire[nom_repertoire]
    return repertoire
```

2.b. Soit la fonction suivante :

Script Python

```
def affiche(racine, adr, nom_fichier):
    repertoire = parcourt(racine, adr)
    print(repertoire[nom_fichier])
```

Qu'affiche l'instruction `affiche(docs, ["Cours", "NSI"], "TP.html")` sachant que la variable `docs` contient le dictionnaire de la question 1 ?

Réponse

- La première instruction fait que `repertoire` correspond au dictionnaire `"NSI"` qui vaut `{"TP.html": 60, "dm.odt": 345}`.
- La seconde affiche la valeur associée à la clé `"TP.html"` de ce dictionnaire, c'est-à-dire le poids en ko de ce fichier.

L'affichage est donc

Console Python

```
>>> affiche(docs, ["Cours", "NSI"], "TP.html")
60
```

3.a. La fonction `ajoute_fichier` suivante, de paramètres `racine`, `adr`, `nom_fichier` et `taille`, ajoute au dictionnaire `racine`, à l'adresse `adr`, la clé `nom_fichier` associé à la valeur `taille`.

Une ligne de la fonction donnée ci-dessous contient une erreur. Laquelle ? Proposer une correction.

Script Python

```
def ajoute_fichier(racine, adr, nom_fichier, taille):
    repertoire = parcourt(racine, adr)
    taille = repertoire[nom_fichier]
```

Réponse

Script Python

```
def ajoute_fichier(racine, adr, nom_fichier, taille):
    repertoire = parcourt(racine, adr)
    repertoire[nom_fichier] = taille
```

3.b. Écrire une fonction `ajoute_repertoire` de paramètres `racine`, `adr` et `nom_repertoire` qui crée un dictionnaire représentant un répertoire vide appelé `nom_repertoire` dans le dictionnaire `racine` à l'adresse `adr`.

Réponse

Script Python

```
def ajoute_repertoire(racine, adr, nom_repertoire):
    repertoire = parcourt(racine, adr)
    repertoire[nom_repertoire] = dict()
```

4.a.

`isinstance` pour vérifier le type d'une variable

`isinstance(variable, A)` renvoie `True` si `variable` est de type `A` et `False` sinon.

`A` peut être le type `int`, `dict` ou tout autre type Python.

Écrire une fonction `est_fichier` de paramètre `racine`, un dictionnaire non vide, qui détermine si `racine` est un répertoire ou un fichier. **On supposera que l'arborescence est bien formée** :

- les répertoires et les fichiers sont des dictionnaires ;
- les répertoires ne contiennent, comme clés, que des répertoires et des fichiers ;
- un répertoire peut être vide ;
- la valeur associée à un fichier associée est toujours un entier.

On pourra compléter le code suivant ou en proposer un autre

Script Python

```
def est_fichier(racine):
    for cle in racine:
        if isinstance(racine[cle], ...):
            return ...
    return ...
```

Réponse

Script Python

```
def est_fichier(racine):
    for cle in racine:
        if isinstance(racine[cle], int):
        return True
    return False
```

4.b Écrire une fonction `taille` de paramètre `racine` qui prend en paramètre un dictionnaire `racine` modélisant un répertoire et qui renvoie le total d'espace mémoire occupé par les fichiers contenus dans ce répertoire.

Réponse

Script Python

```
def taille(racine):
    if est_fichier(racine):
        for cle in racine:
            return racine[cle]
            # il n'y a qu'un tour de boucle
            # racine[cle] sera la taille du fichier visé
    else:
        cumul = 0
        for cle in racine:
            cumul += taille(racine[cle])
            # racine[cle] sera soit un fichier, soit un répertoire
            # la fonction est donc récursive
        return cumul
```