

CONTRÔLE 2

Listes - Implémentation et algorithmes


On utilisera dans toutes les questions de ce contrôle la classe Maillon ainsi que l'interface des objets de type Liste tels que définis dans le cours. Ainsi, on pourra utiliser sans justification supplémentaire :

- utiliser le constructeur `Maillon(v: int, s: Maillon)` pour créer un nouveau maillon ;
- accéder aux attributs `valeur` et `suivant` des objets `Maillon` ;
- utiliser les fonctions `creer_vide`, `est_vide`, `est_singleton`, `singleton`, `ajoute`, `tete`, `queue` et `affiche` afin de manipuler des objets de type `Liste` ou `Maillon`.

Ces fonctions ne modifient pas les objets qu'elles prennent en argument. On rappelle que la fonction `ajoute` a pour signature `Liste, int -> Liste`.

Code python

```
1 m = Maillon(2, Maillon(3, None))
2 m = ajoute(m, 1)
3 affiche(m)
```

 Résultat

1 - 2 - 3 - x

Exercice 1. 1. On considère le code suivant :

Code python

```
1 m4 = Maillon(4, None)
2 m3 = Maillon(3, m4)
3 m2 = Maillon(2, None)
4 m1 = Maillon(1, m2)
```

- a. Représenter à l'aide d'un schéma simplifié l'état de la mémoire de l'ordinateur à l'issue de l'exécution des instructions précédentes. En déduire l'affichage réalisé par le code :

Code python

```
1 for m in [m1, m2, m3, m4]:
2     affiche(m)
```

- b. Quel est l'affichage réalisé par les instructions `affiche(m1)` et `affiche(m2)` **après** l'exécution de l'instruction :

Code python

```
1 m1 = Maillon(5, Maillon(6, m3))
```

Justifier **en complétant** le schéma de la question précédente en vert.

- c. Quel est l'affichage réalisé par les instructions `affiche(m1)` et `affiche(m2)` **après** l'exécution de l'instruction :

Code python

```
1 m2.suivant = m3
```

Justifier **en complétant** le schéma de la question précédente en rouge.

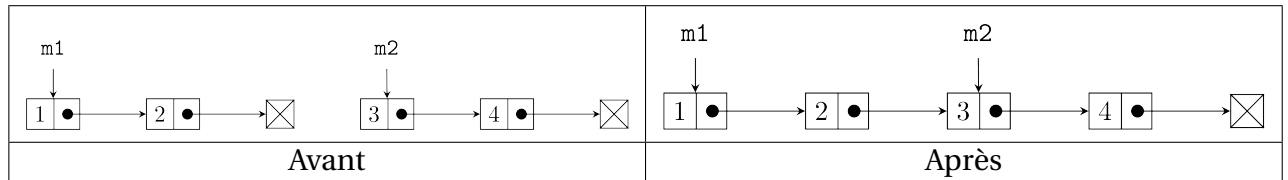
- d. Que se passe-t-il lorsque l'on exécute l'instruction `affiche(m1)` **après** l'exécution de l'instruction :

Code python

```
1 m4.suivant = m2
```

2. On souhaite écrire le code d'une fonction `concatene`, itérative, qui prend en entrée deux arguments `m1` et `m2` de type `Maillon` supposés non vides et qui relie les deux chaînes correspondantes entre elles en affectant `m2` à l'attribut suivant du dernier maillon non vide de la chaîne commençant par le maillon `m1`. La fonction `concatene` modifie en place le maillon en question, elle renvoie `None`.

On décrit dans la figure ci-dessous l'état de la mémoire avant et après l'exécution de l'instruction `concatene(m1, m2)`.



- a. Écrire le code d'une fonction `concatene` correspondant à la description de l'énoncé, sans se soucier d'en écrire la documentation.
- b. On considère le code suivant :

Code python

```

1  a = Maillon(1, Maillon(2, None))
2  b = Maillon(3, Maillon(4, None))

```

Justifier vos réponses aux questions suivantes en réalisant un schéma explicatif par question.

- i. On exécute l'instruction `concatene(a, b)`.
Quel est l'affichage alors réalisé par l'instruction `affiche(a)` ?
- ii. On exécute une deuxième fois l'instruction `concatene(a, b)`.
Quel est l'affichage alors réalisé par l'instruction `affiche(a)` ?
- iii. On exécute une troisième fois l'instruction `concatene(a, b)`.
Que se passe-t-il ?

Exercice 2. 1. Soit la fonction `d` suivante :

Code python

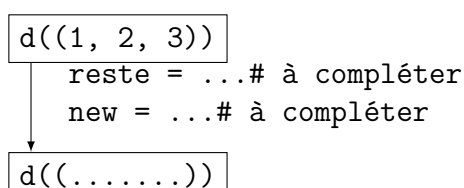
```

1  def d(l):
2      """ Liste -> Liste """
3      if est_vide(l):
4          return l
5      else:
6          reste = d(queue(l))
7          new = ajoute(ajoute(reste, tete(l)), tete(l))
8          return new

```

Déterminer l'affichage réalisé par `affiche(d(1))`, si `l` est la liste `(1, 2, 3)`.

Vous justifierez votre réponse par un arbre d'appel, en y représentant les listes de manière "naturelle". Ainsi, le premier nœud de l'arbre sera :



2. Soit f la fonction suivante :

Code python

```

1 def f(l1, l2):
2     """ Liste, Liste -> Liste """
3     if est_vide(l1):
4         return l2
5     elif est_vide(l2):
6         return l1
7     else:
8         intermediaire = f(queue(l1), queue(l2))
9         return ajoute(ajoute(intermediaire, tete(l2)), tete(l1))

```

Déterminer l'affichage réalisé par `affiche(f(l1, l2))`, si $l1$ est la liste (1, 2, 3) et $l2$ est la liste (4, 5, 6). Vous justifierez vos réponses par un arbre d'appel.

Exercice 3. L'objectif de cet exercice est d'étudier une implémentation récursive des algorithmes de tri classique : le tri par insertion et le tri par sélection.

Partie A

On commence par détailler le fonctionnement de l'algorithme du tri par insertion. Pour ce faire, il est nécessaire d'écrire une fonction auxiliaire, `insérer_dans_liste_triee`.

1. La fonction `insérer_dans_liste_triee` prend en argument une liste l **supposée triée par ordre croissant** et un entier e quelconque et renvoie la liste composée des éléments de l et de e également triée par ordre croissant. Par exemple, si l est la liste (1, 2, 3, 5) et e est l'entier 4, `insérer_dans_liste_triee(l, e)` renvoie la liste (1, 2, 3, 4, 5).

a. Déterminer sans justifier votre réponse dans chacun des cas ci-dessous la liste renvoyée par l'instruction :

`insérer_dans_liste_triee(l, e)`

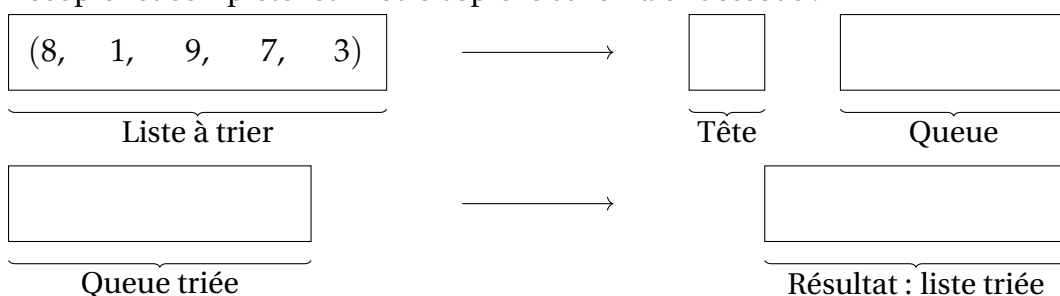
- i. $l = ()$, $e = 3$
- ii. $l = (4, 5, 6)$, $e = 3$
- iii. $l = (11, 14, 18)$, $e = 4$
- iv. $l = (19, 20, 23)$, $e = 22$.
- v. $l = (5, 12, 36, 42)$, $e = 16$.

b. En déduire une fonction récursive `insérer_dans_liste_triee`, qui réponde à la question de l'énoncé. On ne se souciera pas d'écrire la documentation de cette fonction.

2. On rappelle l'algorithme du tri par insertion d'une liste l :

- Si la liste l est vide alors elle est triée.
- Sinon :
 - On trie récursivement la queue de la liste l . On note cette liste l' .
 - On insère l'élément de tête de la liste l "au bon endroit" dans l' de telle sorte que la liste résultant soit également triée.

a. Recopier et compléter sur votre copie le schéma ci-dessous :



b. Compléter directement sur l'énoncé le code ci-dessous.

Code python

```
1  def tri_insertion(l):
2      """ Liste -> Liste
3      Renvoie la liste des éléments de l triés par ordre croissant.
4      ↪      """
5      if est_vide(l):
6          return l
7      else:
8          # 1 - On trie récursivement la queue de la liste
9          trie_intermediaire = .....
10         # Affichage pour analyse
11         affiche(trie_intermediaire)
12         # 2 - On insère dans la queue triée récursivement
13         # l'élément de tête de la liste. La liste résultat
14         # est triée par ordre croissant
15         resultat = .....
16         # Affichage pour analyse
17         print(f"On y insère {tete(l)}")
18         return .....
```

c. Déterminer les affichages réalisés par l'instruction `tri_insertion(l)`, quand `l` est la liste `(8, 9, 1)`.

On pourra suivre l'exécution de l'algorithme à l'aide d'un arbre d'appel.

Partie B

On rappelle que l'on a écrit en TP le code des fonctions suivantes :

- `minimum(l: Liste) -> int` : renvoie le plus petit élément de `l` ;
- `supprime(l: Liste, e: int) -> Liste` : renvoie la liste des éléments de `l` où la première occurrence de `e` a été supprimée (`e` est supposé présent dans la liste `l`).

1. En utilisant uniquement les fonctions de l'interface du type `Liste` ainsi que les fonctions `minimum` et `supprime`, écrire une fonction récursive `tri_selection` qui trie la liste `l` à l'aide de l'algorithme du tri par sélection dont on rappelle le principe ci-dessous :

- Si la liste `l` est vide, alors il n'y a rien à faire.
- Sinon :
 - On calcule la valeur du minimum `m` de la liste `l` ;
 - On calcule la liste `l'` obtenue en supprimant de la liste des éléments de `l` la première occurrence de `m` ;
 - On ajoute l'élément `m` en tête de la liste `l'` (que l'on a préalablement triée).

2. Justifier que votre fonction `tri_selection` est récursive.