Partiel (2 heures)

Quelques remarques préliminaires :
- Les notes de cours, TD et TP sont autorisées.
- Le sujet est probablement trop long (j'ai du mal à estimer le temps qu'il faut pour répondre aux questions), et ce sera donc pris en compte dans la notation. Essayez de bien faire les questions que vous traitez, quitte à ne pas tout faire.
- La plupart des questions demandent d'écrire une fonction. Il est évidemment préférable d'écrire le code en *Python*, mais si vous n'êtes pas sûrs de la syntaxe n'hésitez pas à mettre du *pseudo-code* (c'est-à-dire à inventer plus ou moins la syntaxe que vous ne connaissez pas). La conception des fonctions est plus importante que les détails de syntaxe (mais c'est quand même mieux si la syntaxe est bonne...).
- L'exercice 2 est plus long et plus intéressant, mais n'utilise pas vraiment de fonctions récursives (si vous arrivez à écrire les choses récursivement c'est aussi bien, mais la plupart du temps ce n'est pas très adapté). L'exercice 1 est donc là pour vous faire écrire des fonctions récursives. L'exercice 1 vaudra beaucoup de points (par rapport au nombre de questions du moins) donc essayez de bien le faire.

Exercice 1. *Fonctions récursives*

Parce qu'il faut bien vérifier que vous savez écrire des fonctions récursives sur les listes, écrivez les fonctions suivantes de manière récursive en utilisant les primitives `liste()`, `cons(x,l)`, `tete(l)`, `queue(l)` et `vide(l)` vues en cours, TD et TP (on suppose que les listes contiennent des entiers) :

1. `somme(l)` qui renvoie la somme des éléments de `l`.

2. `ordre(l)` qui renvoie `True` si la liste `l` est ordonnée (les éléments sont rangés du plus petit au plus grand) et `False` sinon.

3. `apparait(x,n,l)` qui renvoie `True` si `x` apparaît au moins `n` fois dans `l`.

4. `double(x,l)` qui double toutes les occurrences de `x` dans `l`, par exemple : `double(2, [1,2,3,2,2,4])` renvoie `[1,2,2,3,2,2,2,4]`.

Exercice 2. *Listes pointées*

Le but de cet exercice est de voir différentes implémentations d'une structure de *listes pointées*. D'un point de vue abstrait (indépendant de l'implémentation), les listes que l'on considère sont des suites finies ordonnées d'éléments ayant un premier et un dernier élément. En plus de cela, on distingue un élément particulier sur chaque suite qui représente l'élément *courant* sur lequel on peut agir.
Visuellement, on pourrait représenter une liste pointée à 6 éléments dont le troisième élément est l'élément courant ainsi (la flèche pointe sur l'élément courant) :

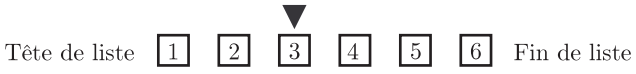

FIGURE 1 – Une liste pointée à 6 éléments

On veut pouvoir effectuer les opérations suivantes :
- `lire(l)` renvoie l'élément courant de la liste (celui sur lequel se trouve le curseur) ;
- `deplacer_gauche(l)` et `deplacer_droite(l)` déplacent le curseur d'un élément vers la gauche (vers la tête de liste) ou vers la droite (vers la fin de la liste) ;

– `ajouter_gauche(x,l)` et `ajouter_droite(x,l)` ajoutent un nouvel élément `x` dans la liste à gauche ou à droite de l'élément courant ;
– `debut(l)` et `fin(l)` renvoient `True` si le curseur est sur la tête de liste ou la fin de la liste, et `False` sinon.

Pour des raisons pratiques on ne considérera pas la liste vide (sur laquelle on ne peut pas pointer d'élément).

I. Listes chaînées simples

On décide dans un premier temps de représenter les listes à l'aide de listes chaînées simples (celles qui ont été vues en cours). On propose d'utiliser les deux classes suivantes :

```
class Noeud():
  def __init__(self, x):
    self.valeur = x
    self.suivant = None
class Liste():
  def __init__(self, x):
    n = Noeud(x)
    self.premier = n
    self.curseur = n
```

Tout comme pour les listes qui ont été vues en cours, le champ `suivant` du dernier nœud de la liste vaut `None`. Pour tous les autres nœuds, le champ `suivant` est le nœud suivant de la liste.

Remarque : Les listes ainsi définies sont légèrement différentes de celles vues en cours puisque l'on ne peut plus créer de liste vide, donc la création d'une nouvelle liste (à un élément) se fait par la commande `Liste(x)` où `x` est la valeur de l'élément de la liste.

1. Écrivez la fonction `lire(l)` qui renvoie la valeur de l'élément sur lequel se trouve le curseur de `l` (appelé `l.curseur`).

2. Écrivez la fonction `deplacer_droite(l)` qui déplace le curseur de la liste `l` d'un pas vers la droite.

3. Écrivez les fonctions `debut(l)` et `fin(l)` (ces deux fonctions sont très simples mais ne fonctionnent pas de la même manière).

Indication : On peut tester si deux nœuds `n1` et `n2` sont égaux en écrivant `n1 == n2`.

4. Écrivez la fonction `ajouter_droite(x,l)` qui ajoute un élement à la liste chaînée à droite du nœud indiqué par le curseur. Expliquez le fonctionnement de la fonction à l'aide d'un schéma (comme ceux faits en cours).

Indication : Il faudra créer un nouveau nœud contenant la bonne valeur, puis l'insérer dans la liste en modifiant les liens nécessaires.

5. Le nombre d'opérations effectuées par les fonctions `deplacer_droite`, `debut`, `fin` et `ajouter_droite` dépend-il de la longueur de la liste ?

6. Écrivez la fonction `deplacer_gauche(l)` qui déplace le curseur d'un élément vers la gauche (cette fonction est plus compliquée que la fonction `deplacer_droite`).

7. Écrivez la fonctions `ajouter_gauche(x,l)` qui ajoute à la liste `l` l'élément `x` avant le nœud pointé par le curseur de `l` (plus difficile que `ajouter_droite`). Quelle opération supplémentaire doit-on effectuer si le curseur se trouve sur la tête de liste quand on appelle `ajouter_gauche` ?

8. Quel est l'ordre de grandeur du nombre d'opérations effectuées par les deux fonctions `deplacer_gauche` et `ajouter_gauche` sur une liste de longueur n dans le pire des cas ?

II. Listes doublement chaînées

Les listes de l'exercice précédent n'étaient pas adaptées pour représenter les listes pointées car les déplacements et les opérations effectués étaient beaucoup plus compliqués dans une direction que dans l'autre.
Pour résoudre ce problème, on peut utiliser des *listes doublement chaînées*, c'est-à-dire des listes pour lesquelles chaque nœud connaît non seulement le nœud suivant mais également le nœud précédent.
On utilisera les deux classes suivantes :

```
class Noeud():
  def __init__(self, x):
    self.valeur = x
    self.suivant = None
    self.precedent = None
class Liste():
  def __init__(self, x):
    n = Noeud(x)
    self.premier = n
    self.dernier = n
    self.curseur = n
```

9. Écrivez les fonctions `deplacer_gauche(l)` et `ajouter_gauche(x,l)` pour cette nouvelle implémentation.

10. Que peut-on dire sur la complexité de ces fonctions (nombre d'opérations effectuées) ?

III. Piles

Les listes doublement chaînées permettent d'effectuer correctement les opérations de base sur les listes pointées. Cependant, nous allons voir qu'une utilisation astucieuse des piles suffit pour effectuer efficacement la plupart de ces opérations.
L'idée est de représenter une liste pointée par deux piles : une pile contient les éléments qui sont à gauche du curseur tandis que la seconde pile contient les éléments qui sont à droite du curseur :

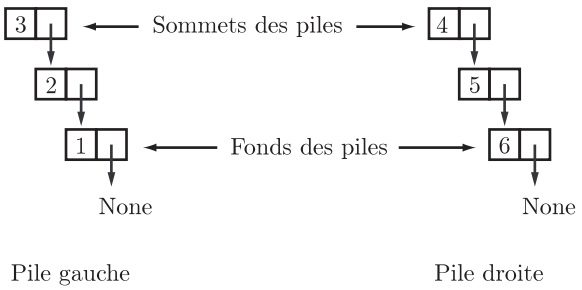

FIGURE 2 – Représentation de la liste pointée de la figure 1 par deux piles.

En particulier l'élément pointé de la liste (celui sur lequel se trouve le curseur) est l'élément au sommet de la première pile. La clé de la construction est de placer, sur les deux piles, les éléments les plus proches du curseur en haut de la pile, ainsi la tête de la liste est au fond de la première pile et la fin de la liste est au fond de la seconde pile.
On utilisera les classes suivantes :

```
class Noeud():
  def __init__(self, x):
    self.valeur = x
    self.suivant = None
```

```
class Liste():
  def __init__(self, x):
    n = Noeud(x)
    self.gauche = n
    self.droite = None
```

Remarque : On représente les piles par des listes simplement chaînées donc pour une liste pointée `l`, `l.gauche` représente le nœud au sommet de la pile de gauche et `l.droite` est le nœud au sommet de la pile de droite.

11. Expliquez l'effet des opérations `deplacer_gauche` et `deplacer_droite` sur les piles à l'aide d'un schéma.

12. Écrivez la fonction `deplacer_gauche(l)`. Sa complexité dépend-elle de la taille de la liste ?

13. Écrivez les fonctions `ajouter_droite(x,l)` et `ajouter_gauche(x,l)` (`ajouter_gauche` est légèrement plus compliquée). Quelle est leur complexité ?

14. Écrivez les fonctions `debut(l)` et `fin(l)`.

IV. Tableaux

Il serait également possible de représenter les listes pointées à l'aide de tableaux. Une liste pointée serait alors représentée par un grand tableau de taille fixe, un entier indiquant la longueur de la liste (comme il a été vu en cours) et un entier indiquant la position de l'élément courant.

15. Écrivez la classe `Liste` représentant des listes pointées à l'aide de tableaux de taille 1000 (il faut écrire la fonction `__init__(self,x)` qui créé une liste à un élément).

16. Parmi les 7 opérations de base sur les listes pointées, lesquelles pourront être réalisées en effectuant un nombre constant d'opérations (indépendant de la taille de la liste) et lesquelles seront plus longues à effectuer si la liste est longue ?

Remarque : On ignorera les problèmes qui apparaissent lorsque le tableau est plein, qui peuvent être résolus par un redimensionnement dynamique dont le coût amorti est constant, comme il a été vu en TD.

Conclusion

17. Quelle est la structure de données qui vous semble être la meilleure pour représenter les listes pointées si l'on va devoir effectuer un grand nombre de fois chacune des différentes opérations de base ? (Justifiez votre réponse)