Examen - 2e session (2 heures)

Les documents (cours, TD, TP) sont autorisés.

Exercice 1. *Fonctions récursives*

Écrivez les fonctions suivantes sur les listes ou les arbres de manière récursive en n'utilisant que les primitives `liste()`, `tete(l)`, `queue(l)`, `vide(l)` et `cons(x,l)` pour les listes et `arbre()`, `racine(a)`, `gauche(a)`, `droit(a)`, `vide(a)` et `cons(x,a1,a2)` pour les arbres :

- **Barème : 1 pt.** `plus_courte(l1,l2)` qui teste si la liste `l1` a moins d'éléments que la liste `l2` ;

  R.

```
def plus_courte(l1, l2):
  if vide(l1):
    return True
  elif vide(l2):
    return False
  else:
    return plus_courte(queue(l1), queue(l2))
```

- **Barème : 1 pt.** `multiple(l,x)` qui renvoie la liste des éléments de `l` qui sont des multiples de l'entier `x` (on peut tester si y est un multiple de x par `y%x==0`) ;

  R.

```
def multiple(l, x):
  if vide(l):
    return l
  elif tete(l) % x == 0:
    return cons(tete(l), multiple(queue(l), x))
  else:
    return multiple(queue(l), x)
```

- **Barème : 1 pt.** `double(l)` qui transforme une liste d'entiers `l` en une liste contenant les doubles des éléments de `l` (par exemple l'image de la liste 1, 3, -2 est 2, 6, -4) ;

  R.

```
def double(l):
  if vide(l):
    return l
  else:
    return cons(2*tete(l), double(queue(l)))
```

- **Barème : 1 pt.** `apparait(x,l,n)` qui teste si un entier `x` apparaît au moins `n` fois dans une liste `l` ;

  R.

```
def apparait(x, l, n):
  if n <= 0:
    return True
  elif vide(l):
    return False
  elif tete(l) == x:
    return apparait(x, queue(l), n-1)
  else:
    return apparait(x, queue(l), n)
```

– **Barème : 1,5 pt.** `min(l1,l2)` qui prend en argument deux listes que l'on suppose de même taille et qui renvoie une liste dont le i-ème élément est le plus petit parmi les i-ème éléments de `l1` et de `l2` (par exemple si `l1` contient 22, 3, 4, 8 et `l2` contient 18, 12, 9, 6, la fonction doit renvoyer la liste 18, 3, 4, 6) ;

R.

```
def min(l1, l2):
  if vide(l1):
    return l1
  elif tete(l1) < tete(l2):
    return cons(tete(l1), min(queue(l1), queue(l2)))
  else:
    return cons(tete(l2), min(queue(l1), queue(l2)))
```

– **Barème : 1,5 pt.** `maximum(a)` qui renvoie le plus grand élément contenu dans un arbre `a` (attention, l'arbre `a` n'est pas nécessairement un arbre binaire de recherche) ;

R.

```
def maximum(a):
  if vide(a):
    return -float("infinity")
  else:
    return max(racine(a), maximum(gauche(a)), maximum(droit(a)))
```

Autre version (plus lourde, mais sans l'astuce du $-\infty$)

```
def maximum(a):
  if vide(gauche(a)) and vide(droit(a)):
    return racine(a)
  elif vide(gauche(a)):
    return max(racine(a), maximum(droit(a)))
  elif vide(droit(a)):
    return max(racine(a), maximum(gauche(a)))
  else:
    return max(racine(a), maximum(gauche(a)), maximum(droit(a)))
```

– **Barème : 1 pt.** `parcours(a,l)` qui prend en argument un arbre et une liste contenant des 0 et des 1. En partant de la racine de l'arbre, on lit les éléments de la liste, si l'élément est un 0 on descend sur le fils gauche de la racine, si c'est un 1 on descend sur le fils droit. On continue à descendre selon les mêmes règles jusqu'à ce que la liste soit vide. La fonction doit alors renvoyer la valeur du nœud de l'arbre sur lequel on se trouve (par exemple si la liste `l` contient 1, 0, 1 il faut renvoyer la valeur du fils droit du fils gauche du fils droit de la racine de l'arbre).

R.

```
def parcours(a, l):
  if vide(l):
    return racine(a)
  elif tete(l) == 0:
    return parcours(gauche(a), queue(l))
  else:
    return parcours(droit(a), queue(l))
```

Exercice 2. *Boule de gomme*

On définit la fonction `mystere` suivante, qui prend en argument une liste :

```
def mystere(l):
  def aux(x, l):
    if vide(l):
```

```
      return l
    if tete(l) < x:
      return cons(x, aux(tete(l), queue(l)))
    else:
      return cons(tete(l), aux(x, queue(l)))
  return aux(tete(l), queue(l))
```

1. Barème : 0,5 pt. Effectuez à la main le déroulement de la fonction `mystere` quand on lui donne comme argument la liste 2, 3, 1, 4 (vous pouvez vous contenter de donner le résultat renvoyé).

R. Sur l'entrée 2, 3, 1, 4, la fonction renvoie la liste 3, 2, 4.

2. Barème : 1 pt. De manière générale, que fait la fonction `mystere` ?

R. La fonction `mystere` commence par prendre le premier élément de la liste (`tete(l)`) comme valeur de `x` puis avance dans la liste en comparant chacun des éléments à la valeur courante de `x`. Si l'élément de la liste est inférieur à `x`, on remet `x` dans la liste renvoyée et on prend le nouveau plus petit élément à la place.
Ainsi, la fonction renvoie une liste contenant les mêmes éléments que la liste donnée en entrée privée de son plus petit élément. L'ordre des éléments restants est légèrement modifié car un élément qui est temporairement pris comme plus petit est replacé à l'endroit où se trouve le premier élément qui lui est inférieur (dans l'exemple de la question précédente, le 2 était pris comme valeur de `x` jusqu'à ce qu'on rencontre le 1 (qui est le véritable minimum) et donc le 2 se retrouve à l'endroit où était le 1 initialement).

Exercice 3. *Listes à trous*

Dans cet exercice, nous allons nous intéresser à une variante des listes dans lesquelles on marque certains éléments pour qu'ils soient ignorés quand on parcourt la liste. L'utilisation principale d'une telle structure est la gestion très simple de la suppression d'éléments (on se contente de marquer l'élément comme supprimé, mais il reste dans la structure de données).

I. Listes chaînées

Pour représenter des listes à trous par des listes chaînées, on modifie légèrement la classe `Noeud` en lui ajoutant un champ `vide` qui contient un booléen indiquant si le nœud doit ou non être ignoré (par défaut un nouveau nœud ne doit pas être ignoré) :

```
class Noeud:
  pass
def noeud(x):
  n = Noeud()
  n.valeur = x
  n.vide = False
  n.suivant = None
  return n
class Liste:
  pass
def liste():
  l = Liste()
  l.premier = None
  return l
```

1. Barème : 1 pt. Écrivez la fonction `supprime(x, l)` qui prend en argument un entier et une liste et supprime de la liste `l` toutes les occurrences de `x`.
Indication : Il faut parcourir la liste comme dans le cas usuel mais lorsqu'on trouve un nœud à supprimer, on se contente de changer la valeur du champ `vide` en `False` puis on passe au nœud suivant. Il n'y a donc pas de liens à changer dans la chaîne.

R.

```
def supprime(x, l):
  n = l.premier
  while n != None:
    if n.valeur == x:
      n.vide = True
    n = n.suivant
```

2. Barème : 1 pt. Écrivez la fonction `longueur(l)` qui renvoie la longueur d'une liste à trous.
Indication : Cette fonction est très proche de la fonction écrite pour les listes chaînées « usuelles » à ceci près qu'il ne faut pas compter les nœuds pour lesquels le champ `vide` vaut `True`.

R.

```
def longueur(l):
  n = l.premier
  total = 0
  while n != None:
    if n.vide == False:
      total += 1
  return total
```

3. Barème : 0,5 pt. De quoi dépend la complexité de la fonction `longueur` ? Quels sont les inconvénients des listes à trous ? Décrivez une situation où il serait très long de calculer la longueur d'une liste n'ayant pourtant que très peu d'éléments.

R. La complexité de la fonction longueur dépend du nombre de nœuds présents dans la structure de données, y compris de ceux qui doivent être ignorés (puisqu'on doit tout de même passer par ces nœuds). Ainsi, si l'on a créé une liste ayant beaucoup de nœuds et que la plupart de ces nœuds ont été supprimés (en les marquant comme vides mais en les gardant dans la structure de données), il sera très long de calculer la longueur de la liste, bien que cette longueur soit très petite.

Pour limiter le problème de la question précédente, qui apparaît lorsque l'on supprime beaucoup de nœuds dans une liste à trous, on va essayer de réutiliser au maximum les nœuds vides lorsque l'on insère un nouvel élément dans la liste.

On considère maintenant des listes à trous triées, c'est à dire que les valeurs se trouvant dans les nœuds non-vides sont rangées de la plus petite (en tête de liste) à la plus grande (en queue de liste). Les valeurs se trouvant sur les nœuds vides n'ont pas d'importance.

Lorsque l'on insère une nouvelle valeur dans la liste, on parcourt la liste comme dans le cas classique jusqu'à trouver une valeur supérieure à la valeur à insérer. Cependant, une fois cet emplacement trouvé, au lieu de créer un nouveau nœud, on commence par regarder si le nœud se trouvant juste avant la valeur devant laquelle on veut insérer le nouvel élément est vide. Si ce nœud est vide, on modifie sa valeur et on change son champ `vide` pour qu'il redevienne un nœud actif. Si par contre il n'y a pas de nœud vide à l'emplacement voulu, on créé un nouveau nœud. La figure 1 illustre l'insertion d'un nouvel élément dans une liste à trous.

4. Barème : 1,5 pt. Écrivez une fonction `insere(x, l)` qui insère la valeur `x` dans la liste à trous triée `l`.
Remarque : Il faut que votre fonction réutilise les nœuds vides disponibles, comme décrit précédemment.

R.

```
def insere(x, l):
  if (l.premier == None) or (l.premier.vide == False and l.premier.valeur <= x):
  # cas particulier où le nouveau noeud est en tête de liste
  nx = noeud(x)
  nx.suivant = l.premier
```

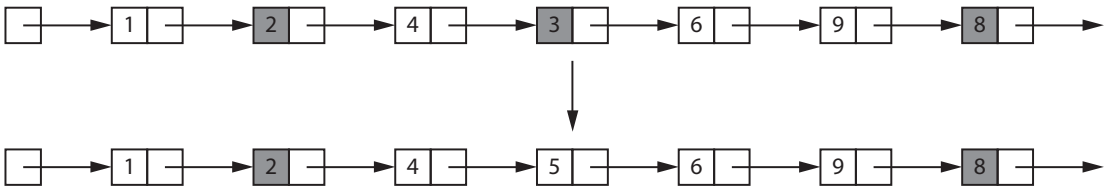

FIGURE 1 – On veut insérer la valeur 5 dans la liste (les nœuds en gris sont les nœuds vides). On parcourt la liste jusqu'à trouver la première valeur supérieure à la valeur à insérer (ici c'est le 6) puis on regarde si le nœud précédent cette valeur est vide. Comme il l'est, on modifie sa valeur et on l'active.

```
  l.premier = nx
  else:
    n = l.premier
    while n.suivant != None and (n.suivant.vide or n.suivant.valeur < x):
    # on avance jusqu'au bout de la liste ou jusqu'à trouver un noeud actif
    # dont la valeur est supérieure à x
      n = n.suivant
    if n.vide: # si le noeud est inactif, on change sa valeur et on l'active
      n.valeur = x
      n.vide = False
    else: # sinon on ajoute un nouveau noeud comme dans le cas classique
      nx = noeud(x)
      nx.suivant = n.suivant
      n.suivant = nx
```

5. Barème : 1,5 pt. Écrivez une fonction `nettoie(l)` qui supprime de la liste tous les nœuds vides. **Indication :** Il faut parcourir la liste et à chaque fois qu'on rencontre un nœud vide, on relie le nœud précédent au prochain nœud non vide. Attention, il est possible que le premier nœud de la liste soit vide (il faut alors modifier le champ `premier` de `l` pour qu'il pointe vers le premier nœud non vide. La figure 2 illustre le résultat de la fonction `nettoie` sur la liste de la figure 1.

R.

```
def nettoie(l):
  n = l.premier
  while n != None and n.vide:
    # on cherche le premier noeud non-vide pour le mettre en tete de liste
    n = n.suivant
  l.premier = n
  precedent = n # on se souvient du dernier noeud actif
  while n != None:
    if n.vide:
      # quand on trouve un noeud ignoré, on relie le précédent à son successeur
      precedent.suivant = n.suivant
    else:
      # si le noeud est actif, il devient le nouveau précédent
      precedent = n
    n = n.suivant
```

II. Tableaux

Il n'est en général pas très judicieux d'utiliser des nœuds vides pour supprimer des éléments dans une liste chaînée car il n'est pas coûteux de supprimer réellement un nœud de la liste (en reliant le nœud

FIGURE 2 – Résultat de la fonction `nettoie` sur la seconde liste de la figure 1.

précédent au suivant). Cependant, dans le cas de listes représentées par des tableaux, la suppression d'un élément nécessite de décaler toutes les valeurs suivantes du tableau, ce qui est très coûteux.

Nous allons donc maintenant considérer des listes représentées par des tableaux et voir comment utiliser les trous pour rendre la suppression plus efficace.

Plutôt que d'ajouter un champ dans les cases du tableau (ce qui est un peu compliqué à faire), on utilise une valeur spéciale qui sert à marquer que la case du tableau n'est plus utilisée. Si les listes ne contiennent que des entiers, on peut par exemple utiliser la valeur `None` pour indiquer une case vide.

On utilise donc les mêmes classes que dans le cas usuel :

```
class Liste:
  pass
def liste():
  l = Liste()
  l.tab = [None]*100
  l.taille = 0
  return l
```

Les différences avec le cas usuel sont les suivantes :

- Le tableau `l.tab` peut maintenant contenir la valeur `None` dans des cases plus proches du début que certaines valeurs de la liste ;
- Le champ `t.taille` indique simplement la dernière case du tableau dans laquelle une véritable valeur de la liste a été écrite, mais ne correspond pas à la longueur réelle de la liste. En effet, s'il y a des trous dans le tableau, la valeur de `l.taille` (qui est simplement le dernier indice du tableau à considérer quand on cherche des valeurs de la liste) sera supérieure à la longueur réelle de la liste (nombre de valeurs différentes de `None` dans le tableau.

6. Barème : 1 pt. Écrivez la fonction `supprime(x, l)` qui supprime toutes les occurrences de l'entier `x` dans la liste `l` représentée par un tableau à trous.

R.

```
def supprime(x, l):
  for i in range(l.taille):
    if l.tab[i] == x:
      l.tab[i] = None
```

7. Barème : 1 pt. Écrivez la fonction `longueur(l)` qui renvoie la longueur de la liste `l`.
Indication : Il faut parcourir le tableau du premier indice jusqu'à l'indice `l.taille` en comptant le nombre de valeurs distinctes de `None`.

R.

```
def longueur(l):
  total = 0
  for i in range(l.taille):
    if l.tab[i] != None:
      total += 1
  return total
```

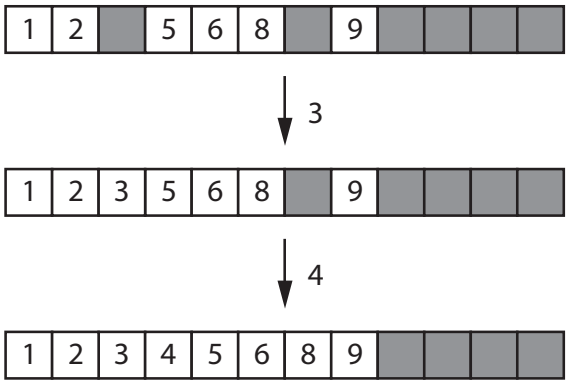

FIGURE 3 – Résultat de l'insertion successive des valeurs 3 et 4 dans une liste triée représentée par un tableau à trous. La première insertion se fait en remplissant un trou, tandis que pour la seconde il faut décaler des valeurs jusqu'au premier trou disponible.

L'insertion dans un tableau trié est semblable au cas des listes chaînées. On parcourt le tableau jusqu'à trouver l'emplacement où mettre la nouvelle valeur. Si cet emplacement est vide, on met la valeur à la place de la valeur `None`. Sinon il faut décaler les valeurs suivantes vers la droite (comme dans le cas normal). Cependant, on peut se contenter de déplacer les valeurs jusqu'à arriver sur une case vide, comme l'illustre la figure 3.

8. Barème : 1,5 pt. Écrivez la fonction `insere(x,l)` qui insère la valeur `x` dans une liste représentée par un tableau trié contenant des trous. Il faut bien sûr remplir les trous lorsque c'est possible.

R.

```
def insere(x, l):
  i = 0
  while i < l.taille and (l.tab[i] == None or l.tab[i] < x):
  # on avance dans le tableau jusqu'à trouver la première valeur supérieure à x
    i += 1
  while i > 0 and l.tab[i-1] == None:
  # on recule tant qu'on trouve des None avant la position où insérer x
    i -= 1
  if l.tab[i] == None:
    # si on se trouve sur un trou, on insère directement la valeur
    l.tab[i] = x # on insère la nouvelle valeur dans un trou
    if i+1 > l.taille:
      # on met à jour la taille de la liste si nécessaire
      l.taille = i+1
  else:
    # si on n'est pas sur un trou, il faut décaler les valeurs pour faire
    # de la place
    j = i+1
    while l.tab[j] != None:
      j += 1
    for k in range(j, i, -1):
      l.tab[k] = l.tab[k-1]
      l.tab[i] = x
    if j+1 > l.taille:
      # on met à jour la taille de la liste si nécessaire
      l.taille = j+1
```

9. Barème : 1,5 pt. Écrivez la fonction `nettoie(l)` qui décale toutes les valeurs du tableau pour ne plus avoir de trous et qui met à jour la valeur de `l.taille` pour qu'elle corresponde au premier indice libre du tableau.

R. On parcourt la liste avec deux indices : un indice i indique la position du trou disponible le plus à gauche (donc le premier emplacement à remplir) tandis qu'un second indice j avance dans la liste pour trouver toutes les valeurs différentes de `None`.
Quand on trouve une vraie valeur à l'indice j, on la place à l'indice i et on incrémente les deux indices. S'il n'y a pas de vraie valeur à l'indice j on incrémente j pour chercher plus loin dans la liste.

```
def nettoie(l):
  i = 0 # indice du premier trou disponible
  while l.tab[i] != None:
    i += 1
  j = i + 1 # indice auquel on cherche les éléments de la liste
  while j < l.taille:
    if l.tab[j] == None:
      j += 1
    else:
      l.tab[i] = l.tab[j]
      l.tab[j] = None
      i += 1
      j += 1
  l.taille = i
```