Partiel (2 heures)

Remarques :
- Tous les documents (cours, TD, TP en particulier) sont autorisés.
- Le sujet est peut-être trop long pour être traité en 2 heures. Ce sera pris en compte dans la notation (qui sera vraisemblablement faite sur un total de plus de 20 points).
- L'exercice 1 sera noté sur environ 8 points.
- Les trois parties de l'exercice 2 sont assez indépendantes. N'hésitez pas à passer à la suivante si vous êtes bloqués ou ne comprenez vraiment pas une partie.
- Vous pouvez venir poser des questions si vous ne comprenez pas certaines parties du sujet (dans la mesure du raisonnable).

Exercice 1. *Fonctions récursives*

Écrivez les fonctions suivantes de manière récursive en utilisant les primitives `liste()`, `tete(l)`, `queue(l)`, `vide(l)` et `cons(a,l)` vues en cours et en TD (on suppose que toutes les listes contiennent des entiers) :

1. `avant_dernier(l)` qui renvoie l'avant-dernier élément de la liste `l` ;

2. `ajoute(x,l)` qui ajoute `x` à chacun des éléments de `l` ;

3. `multiple(l)` qui calcule le plus petit commun multiple des éléments de `l`. On pourra supposer que l'on a déjà une fonction `ppcm(a,b)` qui calcule le plus petit commun multiple de deux entiers ;

4. `paire(l)` qui teste si la liste `l` ne contient que des entiers pairs ;

5. `repete(n,i,l)` qui répète `n` fois l'élément de `l` en position `i`. Par exemple si $\texttt{n} = 3$, $\texttt{i} = 2$ et $\texttt{l} = 1, 2, 3, 4$, la fonction doit renvoyer la liste $1, 2, 3, 3, 3, 4$ (la numérotation des positions commence à 0) ;

6. `applique(f, l1, l2)` qui applique un certain nombre de fois la fonction `f` aux éléments de `l1`. On suppose que la liste `l2` est de même longueur que `l1` et c'est elle qui donne pour chacun des éléments de `l1` le nombre de fois qu'il faut appliquer `f`. Par exemple, si $\texttt{l1} = 1, 2, 3$ et $\texttt{l2} = 2, 0, 1$, la fonction doit renvoyer la liste $f(f(1)), 2, f(3)$;

7. `alterne(l)` qui teste si une liste est *alternée*, c'est-à-dire si son premier élément est plus petit que le second, que le second est plus grand que le troisième, le troisième est plus petit que le quatrième, le quatrième plus grand que le cinquième et ainsi de suite.

Exercice 2. *Ensembles finis*

Dans cet exercice nous allons nous intéresser aux différentes structures de données permettant de représenter des ensembles finis d'entiers. Afin de construire les ensembles que l'on utilisera, il faut être capable d'ajouter un élément à un ensemble existant. Par ailleurs, on veut pouvoir déterminer efficacement si un élément donné appartient à un ensemble ou non.

1. Si l'on suppose que l'on représente un ensemble par une liste en ajoutant simplement les éléments un par un à une extrêmité de la liste (donc dans l'ordre dans lequel ils arrivent), quel est l'ordre de grandeur du nombre d'opérations à effectuer pour déterminer si un élément donné appartient ou non à la liste, en fonction du nombre d'éléments n de la liste ?

I. Tableaux ordonnés

Afin d'améliorer la complexité de la recherche, on va utiliser des listes ordonnées. On suppose donc qu'à tout moment un ensemble est représenté par une liste contenant les éléments de l'ensemble dans l'ordre croissant (le plus petit en premier, le plus grand à la fin). Lorsque l'on ajoute un nouvel élément, on ne l'ajoute pas à une extrêmité mais à la position qui lui correspond pour que la liste soit toujours ordonnée.
On considère dans un premier temps la représentation des listes par des tableaux vue en cours. Une liste est donc définie par un champ `tableau` contenant un grand tableau dans lequel les éléments de la liste occupent les premières cases, et un champ `longueur` qui indique le nombre d'éléments de la liste.

2. Écrivez une fonction `ajouter(x,l)` qui ajoute l'élément `x` à la liste triée `l` de telle sorte que la liste reste triée (cette fonction ressemble à la fonction écrite en cours pour insérer un élément en position i dans une liste représentée par un tableau, mais il faut d'abord trouver la position à laquelle insérer le nouvel élément).

3. Quelle est la complexité (ordre de grandeur du nombre d'opérations effectuées) de la fonction `ajouter(x,l)` en fonction de la longueur n de la liste ?

4. Écrivez de manière naïve la fonction `recherche(x,l)` qui renvoie `True` si `x` appartient à la liste, et `False` sinon. Quelle est sa complexité ?

Il existe une méthode plus efficace pour chercher un élement dans un tableau trié, appelé *méthode par dichotomie* :
– On recherche initialement un élément qui peut se trouver entre les positions `min=0` et `max=l.longueur`
– On regarde la valeur de l'élément du milieu en position `(min+max)/2`. Si cet élément est plus grand que x, il suffit de regarder les éléments en positions inférieures, on peut donc prendre `max=(min+max)/2`, sinon on recherche x dans les positions au dessus du milieu en prenant `min=(min+max)/2+1`
– on recommence la recherche en divisant l'intervalle de recherche en deux jusqu'à ce que les valeurs `min` et `max` soient égales.
– lorsqu'il ne reste plus qu'une position possible, on regarde si c'est x ou non et on peut répondre.
La technique précédente permet de faire une recherche dans une liste en ne regardant que $O(\log(n))$ valeurs dans la liste (à chaque fois qu'on regarde une nouvelle valeur, on divise l'intervalle de recherche par 2, or on ne peut diviser n que $\log(n)$ fois par 2 avant d'obtenir une valeur inférieure à 1).

5. Écrivez la fonction `recherche(x,l)` en utilisant la méthode de dichotomie.

II. Listes chaînées ordonnées

Une autre façon de représenter les listes est d'utiliser des listes chaînées (une liste est donnée par son premier nœud, chaque nœud contient un élément de la liste et un pointeur vers le nœud suivant).

6. En utilisant la représentation des listes par des listes chaînées vue en cours, écrivez la fonction `ajouter(x,l)` qui ajoute un élément `x` à une liste triée `l` de telle sorte que la liste reste triée. Quelle est sa complexité ?

7. Écrivez une fonction `recherche(x,l)` qui teste si un élément `x` appartient ou non à la liste `l` en utilisant le fait que la liste est ordonnée. Combien d'éléments de la liste doivent-être observés par la fonction `recherche` en moyenne ?

III. Arbres binaires de recherche

Le fait d'utiliser des listes chaînées triées ne facilite pas véritablement la recherche car on est toujours obligé de parcourir tous les premiers éléments de la liste pour avancer (on ne peut pas sauter directement au i-ème élément comme dans le cas des tableaux). Pour effectuer efficacement la recherche, il est nécessaire d'enrichir la structure de listes chaînées. On peut pour cela utiliser des arbres au lieu des listes.
Un arbre binaire est semblable à une liste chaînée, puisqu'il est également constitué de nœuds reliés les uns aux autres. Chaque nœud contient une valeur, mais à la différence des listes chaînées, un nœud a

maintenant deux successeurs, qu'on appelle ses fils (un fils *droit* et un fils *gauche*). Un arbre binaire peut être représenté graphiquement comme illustré sur la figure 1 (le premier nœud de l'arbre, appelé *racine* est le nœud contenant la valeur 6 sur l'exemple).

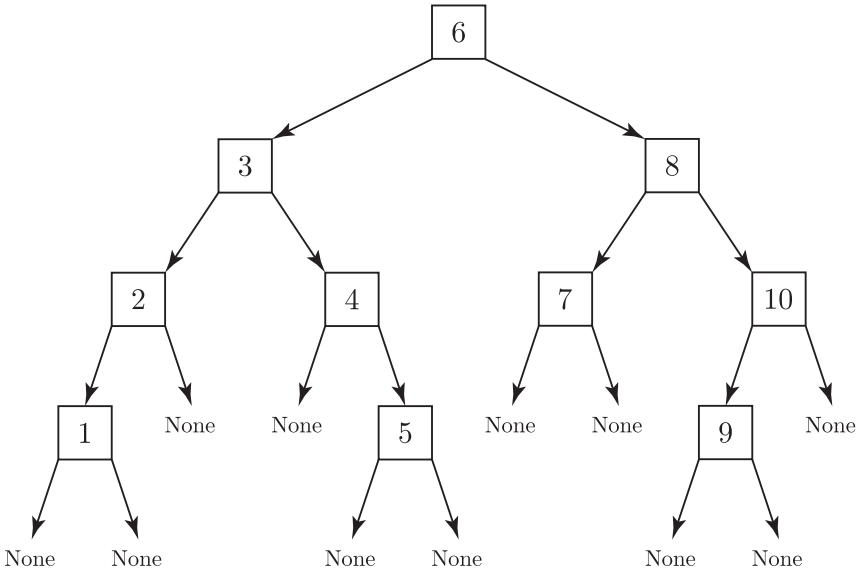

FIGURE 1 – Un arbre binaire (qui se trouve être également un arbre binaire de recherche)

Tout comme les listes chaînées, les fils d'un nœud peuvent être vides (ce qui permet d'avoir des arbres finis), on met alors la valeur `None` dans le champ du fils.
On peut utiliser les classes et constructeurs suivants :

```
class Arbre:
    pass
class Noeud:
    pass
def arbre():
    a = Arbre()
    a.racine = None
    return a
def noeud(x):
    n = Noeud()
    n.valeur = x
    n.gauche = None
    n.droit = None
    return n
```

8. Dessinez l'arbre `a` obtenu par la suite d'instructions suivante :

```
>> a = arbre()
>> n1 = noeud(1)
>> n2 = noeud(2)
>> n3 = noeud(3)
>> n4 = noeud(4)
>> a.racine = n3
>> n3.gauche = n2
>> n3.droit = n4
>> n4.gauche = n1
```

On appelle *arbre binaire de recherche* un arbre binaire tel que pour tout nœud, l'ensemble des valeurs qui sont à sa gauche (la valeur de son fils gauche et tous les descendants de ce fils gauche) sont inférieures à sa valeur, tandis que l'ensemble des valeurs qui sont à sa droite sont supérieures à sa valeur. Par exemple, l'arbre binaire représenté sur la figure 1 est un arbre binaire de recherche. Les arbres binaires de recherche sont des arbres binaires triés.

9. L'arbre produit par les instructions de la question précédente est-il un arbre binaire de recherche ? Justifiez votre réponse.

10. Que fait la fonction suivante qui prend en argument un entier `x` et un arbre binaire de recherche `a` ?

```
def mystere(x, a):
    def myst_rec(x, n):
        if x < n.valeur:
            if n.gauche == None:
                nx = noeud(x)
                n.gauche = nx
            else:
                myst_rec(x, n.gauche)
        else:
            if n.droit == None:
                nx = noeud(x)
                n.droit = nx
            else:
                myst_rec(x, n.droit)
    myst_rec(x, a.racine)
```

11. Écrivez la fonction `recherche(x,a)` qui teste si un entier `x` appartient ou non à un arbre binaire de recherche `a` (pensez à exploiter le fait que l'arbre est trié, c'est plus simple que de chercher dans un arbre binaire quelconque !).

La *profondeur* d'un arbre est la longueur du plus long chemin descendant dans l'arbre (par exemple, la profondeur de l'arbre de la figure 1 est 3 car un des plus longs chemins est $6 \to 3 \to 2 \to 1$ qui est de longueur 3). Si l'arbre est bien « équilibré », sa profondeur est de l'ordre de $\log(n)$ où n est le nombre d'éléments dans l'arbre.

12. Quelle est la complexité de la fonction `recherche(x,a)` de la question précédente en fonction de la profondeur de l'arbre ?