Entraînement : Arbres structure récursive

Pour tous les exercices, la grille d'évaluation est la suivante.

Application d'algorithmes.

A (20)	Valeur des fonctions et interprétation correcte
C (11)	Valeurs des fonctions correctes mais mauvaise interprétation en général.
D (8)	Erreurs pour les valeurs dans une des deux fonctions.
E (1)	Les deux sont fausses.

Conception d'algorithmes.

A (20)	L'algorithme répond correctement au problème posé, il est écrit de façon claire et la complexité est optimale.
B (16)	L'algorithme contient quelques erreurs mais reste globalement juste et la complexité est optimale.
C (11)	L'algorithme fonctionne uniquement dans certains cas ou bien la complexité n'est pas optimale.
D (8)	L'algorithme ne fonctionne pas.
E (1)	Algorithme quasi inexistant ou ne répondant pas du tout au problème posé.

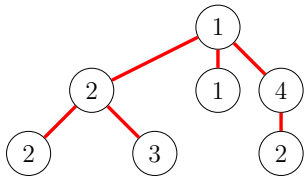
Les questions ♣ sont "bonus" : on ne mettra que A, B ou C, il faut les réussir pour avoir 20 mais elles n'affectent pas la validation du partiel.

Exercice 1.

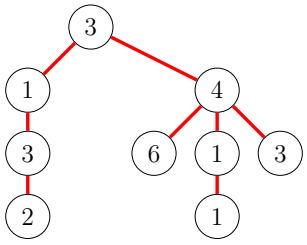
On définit la structure suivante pour représenter les arbres :

```
Structure Arbre :
    valeur, un entier
    nbFils, un entier
    Fils, un tableau de taille nbFils contenant des Arbres
```

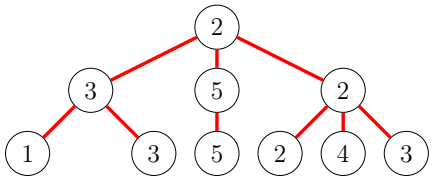
Et les trois exemples suivants,



Exemple 1



Exemple 2



Exemple 3

(1) Donner les valeurs de `mystere1` et `mystere2` sur les 3 exemples d'arbres donnés.

```
Fonction mystere1(Arbre arbre):
    s = 1
    Pour chaque fils f de arbre:
        s = s + mystere1(f)
    retourner s

Fonction mystere2(Arbre arbre):
    m = 0
    Pour chaque fils f de arbre:
```

```
        h = mystere2(e)
        Si h > m:
            m = h
    retourner m + 1
Fin fonction
```

(2) Que calculent `mystere1` et `mystere2` ?

(3) Donner une fonction qui calcule le nombre de *feuilles* de l'arbre (nombre de nœuds qui n'ont pas de fils). La fonction doit retourner 4 sur l'exemple 1, 4 sur l'exemple 2 et 6 sur l'exemple 3.

(4) ♣ On dit qu'un arbre est équilibré si toutes les feuilles sont à la même profondeur. Dans les arbres donnés en exemple, seul l'exemple 3 est équilibré (toutes ses feuilles sont à la profondeur 3, pour les exemples 1 et 2, il y a des feuilles aux profondeurs 2 et 3). Proposer un algorithme qui teste si un arbre donné est équilibré. (Note : il y a plusieurs solutions possibles, plus l'algorithme est optimal en terme de complexité, mieux c'est).

Solution

(1) mystère 1 : Exemple 1 = 7, Exemple 2 = 9, Exemple 3 = 10. mystère 2 : Exemple 1 = 3, Exemple 2 = 4, Exemple 3 = 3.

(2) mystère 1 : nombre de nœuds. mystère 2 : hauteur de l'arbre.

(3)

```
Feuilles(Arbre a):
    Si a.nbFils = 0:
        Retourner 1
    s <- 0
    Pour chaque fils f de a:
        s <- s + Feuilles(f)
    Retourner s
```

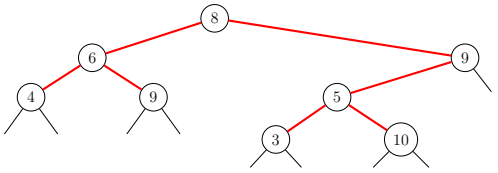
(4) Une solution possible (pas la seule), on effectue un parcours en largeur par niveaux

```
Equilibre(Arbre a):
   N1 <- [a] (tableau contenant a)
   Tant que len(N1) > 0:
       N2 <- [] (tableau vide)
       TousFeuilles <- Vrai
       UneFeuille <- Faux
       Pour chaque n dans N1:
          Si n.nbFils = 0:
              UneFeuille <- Vrai
          Sinon :
              TousFeuilles <- Faux
              Pour chaque b dans n.Fils:
                  N2.append(b)
       Si TousFeuilles :
           Retourner Vrai
       Sinon :
           Si UneFeuille :
               RetournerFaux
```

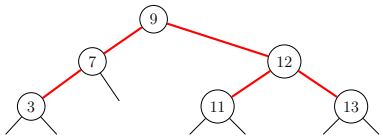
Exercice 2 (Arbres Binaires)**.**
On définit la structure suivante pour représenter les arbres binaires :

```
Structure ArbreBinaire :
    valeur, un entier
    filsGauche, un ArbreBinaire
    filsDroit, un ArbreBinaire
```

Si un fils est vide, on considérera qu'il est égal à `None`, sur les exemples, on représente les arbres vides par des arrêtes sans noeud au bout.



Exemple 1



Exemple 2

(1) Donner un algorithme calculant la hauteur d'un arbre binaire. Sur les exemples, vous devez trouver 4 sur l'Exemple 1 et 3 sur l'Exemple 2.

(2) Donner les valeurs de la fonction `Strahler` sur les 2 exemples d'arbres donnés.

```
Strahler
Input :
    - A, un arbre binaire ou bien None
Output :
    - Un entier
Procédé :
        Si A = None:
            retourner 0
        s1 <- Strahler(A.filsGauche)
        s2 <- Strahler(A.filsDroit)
        Si s1 = s2:
            retourner s1+1
        Sinon:
            retourner max(s1,s2)
```

(3) Donner le résultat de la fonction suivante sur les deux exemples d'arbres donnés. Comment s'appelle ce parcours ?

```
Parcours
Input :
    - A, un arbre binaire ou bien None
Output :
    - Une liste d'entiers
Procédé :
        Si A = None:
            retourner [] # Liste vide
        L <- [A.valeur] # liste contenant une seule valeur
        L1 <- Parcours(A.filsGauche)
        L2 <- Parcours(A.filsDroit)
        retourner L1 + L + L2 # concaténation de listes
```

(4) On rappelle qu'un arbre binaire de recherche est un arbre tel que pour chaque noeud, les noeuds de son sous arbre gauche ont des valeurs **plus petites ou égales** et les noeuds de son sous arbre droit ont des valeurs **strictement supérieures**. L'exemple 1 n'est pas un arbre binaire de recherche et l'exemple 2 en est un.

(a) Écrivez une fonction qui prend en paramètre un arbre binaire de recherche et un entier v et qui retourne Vrai si v est dans l'arbre et Faux sinon.

(b) De quoi dépend la complexité de votre algorithme ?

(c) Quelle est la particularité d'un parcours infixe dans un arbre binaire de recherche ?

(d) ♣ Utilisez cette propriété pour écrire une fonction qui teste si un arbre donné est un arbre binaire de recherche. (Vous pouvez utiliser les fonctions précédentes). On ne traitera que le cas de valeurs distinctes (pas d'égalité).

Solution

(1)
```
Hauteur
Input : un arbre binaire A
Procédé :
Si A = None :
    Renvoyer 0
Renvoyer 1 + max(Hauteur(A.filsG), Hauteur(A.filsD))
```

(2) Exemple 1 : 3 – Exemple 2 : 2

(3) Exemple 1 : [4,6,9,8,3,5,10,9] – Exemple 2 : [3,7,9,11,12,13]

(4) (a)
```
Recherche
Input :
    - un arbre binaire de recherche A
    - un entier v
Procédé :
Si A = None :
    Renvoyer Faux
Si v = A.valeur :
    Renvoyer Vrai
Si v < A.valeur :
    Renvoyer Recherche(A.filsG ,v)
Sinon :
    Renvoyer Recherche(A.filsD , v)
```

(b) La complexité dépend de la hauteur de l'arbre

(c) Un parcours infixe sur un arbre binaire de recherche renvoie les valeurs dans l'ordre croissant.

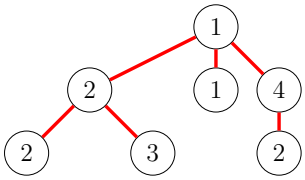
(d)
```
TestABR
Input : Un arbre binaire A
Procédé :
L = Parcours(A) # le parcours infixe décrit plus haut
Pour i allant de 1 à taille(L):
    Si L[i] < L[i-1]:
        Renvoyer Faux
Renvoyer Vrai
```

(Remarque : dans le cas où un arbre contient des valeurs égales, il se peut que le parcours renvoie une liste triée alors que ce n'est pas un ABR).

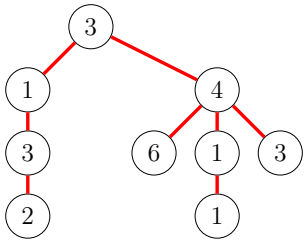
Exercice 3 (Partiel 2018)**.**
On définit la structure suivante pour représenter les arbres :

```
Structure Arbre :
    valeur, un entier
    nbFils, un entier
    Fils, un tableau de taille nbFils contenant des Arbres
```

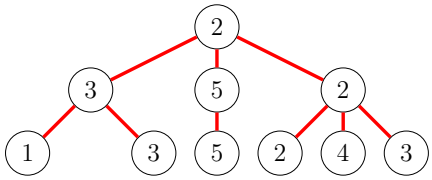
Et les trois exemples suivants,



Exemple 1



Exemple 2



Exemple 3

(1) Donner les valeurs de `mystere1` et `mystere2` sur les 3 exemples d'arbres donnés.

```
Fonction mystere1(Arbre arbre):
    Si arbre.nbFils = 0:
        Retourner 1
    s = 0
    Pour chaque fils f de arbre:
        s = s + mystere1(f)
    retourner s

Fonction mystere2(Arbre arbre):
    Si arbre.nbFils = 0:
        Retourner 0
    s = 1
    Pour chaque fils f de arbre:
        s = s + mystere2(f)
    retourner s
```

(2) Que calculent `mystere1` et `mystere2` ?

(3) On suppose que ces arbres représentent des chemins dans un labyrinthe dont l'entrée est la racine et les feuilles sont les sorties. On veut calculer la *distance minimale* entre la racine et la sortie. Sur les exemples, on obtiendrait la valeur 2 pour l'exemple 1 (chemin $1 \to 1$) , et 3 pour les exemples 2 et 3 (chemins $3 \to 4 \to 6$ et $2 \to 3 \to 1$). Écrire une **fonction récursive** qui calcule cette distance minimale en utilisant un **parcours en profondeur**.

(4) ♣ Imaginez un arbre tel que sa première branche à gauche soit une longue ligne de 10^6 nœuds tandis que sa branche de droite a une profondeur de 10. Dans ce cas, le parcours en profondeur pour calculer la distance minimale entre la racine et une feuille n'est pas optimal. Écrire une fonction qui calcule cette distance minimale à l'aide d'un parcours en largeur.

Solution

(1) mystere1 : 4,4,6 – mystere2 : 3,5,4

(2) mystere1 : nombre de feuilles – mystere2 : nombre de noeuds internes

(3)

```
DistMin
Input : un Arbre a
Procédé :
d <- 0
Pour chaque fils f dans a.Fils:
    df <- DistMin(f)
    si d = 0 ou si df < d:
        d <- df
Retourner d+1
```

(4)

```
DistMin
Input : un Arbre a
Procédé :
d <- 1
L <- [a]
Tant que Vrai :
    Lnext <- []
    Pour tout arbre a de L:
        Si a.nbFils = 0:
            Retourner d
        Pour tout fils f de a.fils:
            Ajouter f à Lnext
    d <- d+1
    L <- Lnext
```