Arbres binaires (Binary Trees)

1 Mesures

Exercice 1.1 (Taille (Size))

1. Donner les axiomes définissant l'opération *taille* sur le type abstrait `arbre binaire`.
2. Écrire une fonction qui calcule la taille d'un arbre binaire.

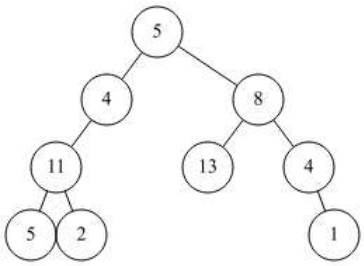

FIGURE 1 –

Exercice 1.2 (Hauteur (Height))

1. Donner les axiomes définissant l'opération *hauteur* sur le type abstrait `arbre binaire`.
2. Écrire une fonction qui calcule la hauteur d'un arbre binaire.

Exercice 1.3 (Longueurs de cheminement et Profondeurs moyennes)

1. Soit les fonctions suivantes :

```
fonction fun_rec(arbrebinaire B, entier h,  ref entier n) : entier
debut
    si B = arbre-vide alors
        retourne 0
    sinon
         n ← n + 1
         retourne h + fun_rec(g(B), h+1, n) + fun_rec(d(B), h+1, n)
    fin si
fin
```

```
fonction fun(arbrebinaire B) : reel
variables
   entier  nb
debut
    si B = arbre-vide alors
         /* Exception */
    sinon
        nb ← 0
        retourne (fun_rec (B, 0, nb) / nb)
    fin si
fin
```

   (a) Quels seront les résultats de la fonction `fun` sur les arbres des figures 1 et 2 ?
   (b) Quelle mesure calcule cette fonction ?
2. Écrire une fonction qui calcule la longueur de cheminement externe d'un arbre binaire.
3. Que faut-il modifier à cette fonction pour qu'elle calcule la profondeur moyenne externe ?

Exercice 1.4 (Maximum Path Sum – *C2# 2017*)

Dans un arbre binaire, nous définissons la *valeur d'une branche* comme étant la somme des valeurs de nœuds sur cette branche.

Écrire la fonction `maxpath(`B`)` qui détermine la plus grande valeur des branches de l'arbre binaire B (dont les clés sont des entiers).

Par exemple, dans l'arbre de la figure 1, la somme retournée sera $26 = 5 + 8 + 13$

2 Parcours

Exercice 2.1 (Parcours profondeur (Depth-first Traversal))

1. En considérant un parcours en profondeur main gauche de l'arbre de la figure 2 donner la liste des nœuds pour chacun des trois ordres induits.

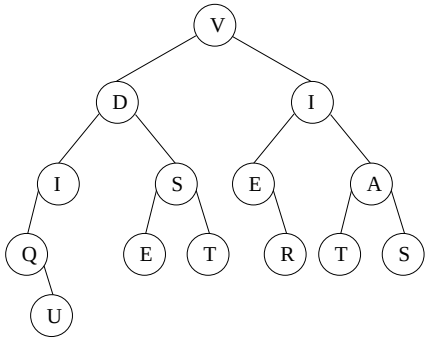

FIGURE 2 – Arbre binaire pour parcours

2. Donner l'arbre 2 sous la forme $<r, G, D>$ avec _ pour représenter l'arbre vide.
3. Écrire une fonction qui affiche un arbre binaire sous la forme $<r, G, D>$, avec _ pour représenter l'arbre vide.

Exercice 2.2 (Parcours largeur (Breadth-first Traversal))

1. Dérouler l'algorithme du parcours largeur sur l'arbre de la figure 2.
2. Écrire une fonction qui calcule la largeur d'un arbre binaire.

3 Tests

Exercice 3.1 (Tests – *C2 - 2017*)

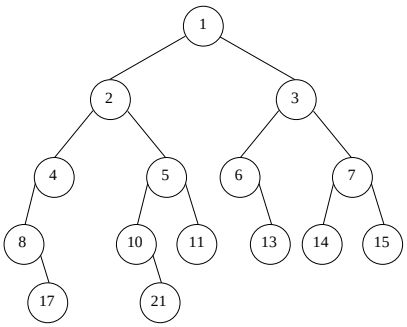

FIGURE 3 – Arbre binaire B

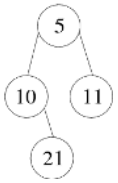

FIGURE 4 – Un sous-arbre de B

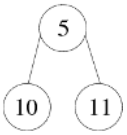

FIGURE 5 – Pas un sous-arbre de B

1. Écrire la fonction `equal` qui vérifie si deux arbres binaires sont identiques : ils contiennent les mêmes valeurs aux mêmes places.
2. Écrire la fonction `isSubTree(`S`, `B`)` qui vérifie si l'arbre binaire S est un sous-arbre de l'arbre binaire B. Il n'y a pas deux clés identiques dans B.

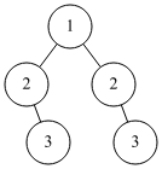

FIGURE 6 – Arbre non symétrique

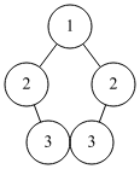

FIGURE 7 – Arbre symétrique

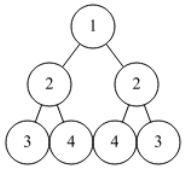

FIGURE 8 – Arbre symétrique

Exercice 3.2 (Symmetric – *C2# 2017*)

Un arbre binaire est dit *symétrique* s'il est identique à son miroir (arbre renversé selon un axe vertical).

Écrire la fonction `symmetric` qui vérifie si un arbre binaire est symétrique.

Exercice 3.3 (Dégénéré, parfait ou complet)

1. **Arbre dégénéré (*degenerate*) :**
   (a) Rappeler ce qu'est un arbre dégénéré.
   (b) Comment tester si un arbre est dégénéré si on connaît sa taille et sa hauteur ?
   (c) Écrire une fonction qui détermine si un arbre binaire est dégénéré (sans utiliser `hauteur` et `taille`).
2. **Arbre complet (*perfect*) :**
   (a) Donner plusieurs définitions d'un arbre complet.
   (b) Comment tester si un arbre est complet en utilisant sa taille et sa hauteur ?
   (c) Écrire une fonction qui teste si un arbre est complet (vous pouvez utiliser une seule fois la fonction qui calcule la `hauteur`).
   (d) Écrire à nouveau la fonction de test sans utiliser `hauteur`.
3. **Arbre parfait (*complete*) :**
   (a) Rappeler ce qu'est un arbre parfait.
   (b) Comment modifier les fonctions qui testent si un arbre est complet pour qu'elles testent si l'arbre est parfait ?

4 Construction

Exercice 4.1 (Tree serialization)

Voici la liste des valeurs en ordre préfixe de rencontre lors du parcours profondeur de l'arbre figure 2 :

```
L_pref = ['V', 'D', 'I', 'Q', 'U', 'S', 'E', 'T', 'I', 'E', 'R', 'A', 'T', 'S']
```

Il n'est pas possible de reconstruire l'arbre à partir de cette seule liste : il y a plusieurs arbres qui donnent cet ordre préfixe.

Par contre, c'est possible à partir de cette liste :

```
L_serial = ['V', 'D', 'I', 'Q', None, 'U', None, None, None, 'S', 'E', None, None,
            'T', None, None, 'I', 'E', None, 'R', None, None, 'A', 'T', None, None,
            'S', None, None]
```

On parle de *sérialisation*. (Preorder traversal serialization of a binary tree.)

1. Donner la forme *sérialisée* de l'arbre de la figure 3.
2. Écrire une fonction qui retourne un arbre binaire sous sa forme *sérialisée* : une liste comme celle présentée ci-dessus.
3. Écrire une fonction qui reconstruit l'arbre à partir de sa forme *sérialisée*.

Exercice 4.2 (La taille en plus – *P2# - 2018*)

Pour cet exercice, on ajoute une nouvelle implémentation des arbres binaires dans laquelle chaque nœud contient la taille (`size`!) de l'arbre dont il est racine.

```
class BinTreeSize:
    def __init__(self, key, left, right, size):
        self.key = key
        self.left = left
        self.right = right
        self.size = size
```

Écrire la fonction `copyWithSize(`B`)` qui prend en paramètre un arbre binaire B "classique" (`BinTree` sans la taille) et qui retourne un autre arbre binaire, équivalent au premier (contenant les mêmes valeurs aux mêmes places) mais avec la taille renseignée en chaque nœud (`BinTreeSize`).

5 Occurrences et Numérotation hiérarchique (Hierarchical Numbering)

Exercice 5.1 (Affichage par occurrences)

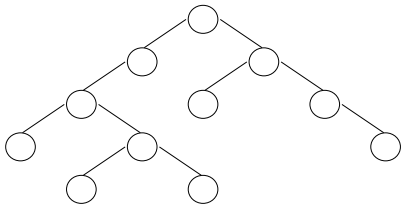

FIGURE 9 – Arbre binaire pour occurrences

1. Donner la représentation sous forme de liste d'occurrences de l'arbre de la figure 9.
2. Écrire la fonction permettant d'afficher la représentation par occurrences d'un arbre binaire.

Exercice 5.2 (Arbre binaire et code préfixe)

Une méthode pour compresser des fichiers textes consiste à encoder les caractères par des mots binaires. On considère ici un encodage de taille variable : chaque caractère peut être représenté par un nombre différent de bits. Il est évidemment conseillé d'utiliser des codes courts pour les caractères fréquents, et de réserver les codes longs pour les caractères rares.
Il faut d'autre part qu'aucun code ne soit préfixe d'un autre. Par exemple si on choisit d'encoder 'a' par 11 et 'b' par 111, on ne saura plus si 11111 désigne 'ab' ou 'ba'.

1. Considérons le code suivant :

lettre	a	b	c	d	e	f
code	0	101	100	111	1101	1100

   Décoder 11011100110001001101.

2. L'encodage est représenté par un arbre dont les **feuilles** sont les lettres à encoder (le champ `cle` contient la lettre) : le code d'une lettre se lit en suivant la branche de cette lettre à partir de la racine (fils gauche = 0, fils droit = 1).
   À quoi correspond le code d'une lettre ?
3. Dessiner l'arbre correspondant au code de la question 1.
4. L'arbre représentant l'encodage est un arbre localement complet dont toutes les feuilles sont utilisées. Écrire une fonction qui retourne le code d'une lettre donnée si elle existe dans l'arbre. Par exemple, avec le code de la question 1, si la lettre donnée est 'e', la fonction retournera "1101".

Exercice 5.3 (Implémentation hiérarchique)

Rappel :

On peut utiliser un simple tableau (une liste en Python) pour représenter un arbre binaire. Il suffit de stocker chaque valeur à la position correspondant au numéro en ordre hiérarchique du noeud la contenant.

1. Donner le tableau représentant l'arbre de la figure 10.
2. Que faut-il modifier aux parcours (profondeur et largeur) si l'arbre en paramètre est donné sous la forme d'une liste (implémentation "hiérarchique") ?

Exercice 5.4 (Object ↔ List)

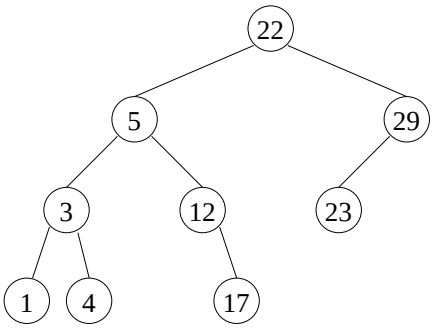

FIGURE 10 – Arbre pour implémentation hiérarchique

1. Écrire une fonction qui construit la liste contenant la représentation hiérarchique d'un arbre binaire (implémentation "classique"). La valeur particulière `None` sera utilisée pour indiquer un arbre vide.

   L'arbre ici est considéré comme quelconque. Que peut-on modifier si l'arbre est parfait ?

2. Écrire la fonction qui construit la représentation "objet" de l'arbre donné sous forme hiérarchique.