Arbres binaires (Binary Trees)

1 Implementation

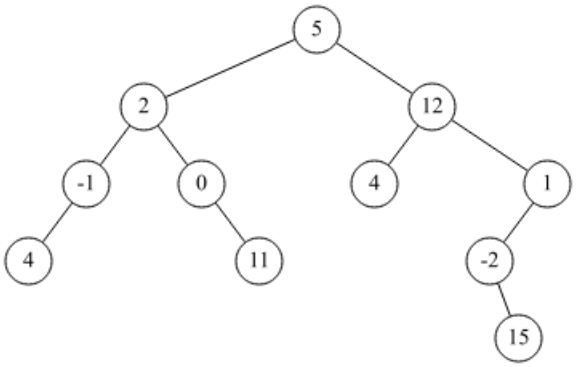

FIGURE 1 –

2 To recurse is divine

Exercice 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.

Exercice 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 3 (Parcours profondeur (Depth-First Search))

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.
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.

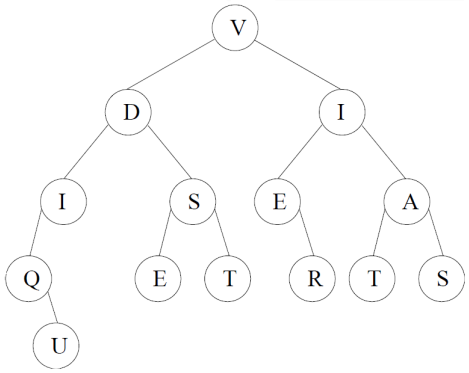

FIGURE 2 – Arbre binaire pour parcours

Exercice 4 (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 ?
   (c) Écrire cette fonction en Python.

2. Que faut-il modifier à cette fonction pour qu'elle calcule la profondeur moyenne externe ?

3 Parcours largeur

Exercice 5 (Parcours largeur (Breadth-first Search) (BFS))

1. Dérouler l'algorithme du parcours largeur sur l'arbre de la figure 2.
2. Écrire une fonction qui affiche les clés d'un arbre binaire en ordre hiérarchique.

Exercice 6 (Largeur moyenne pondérée – *Contrôle 2 - 2021-03*)

Écrire la fonction `get_average(B)` qui prend en paramètre un arbre binaire `B` et qui construit et renvoie une liste `L` telle que :

— `len(L)` = nombre de niveaux de l'arbre `B`

— `L[i]` = somme des clés des nœuds du niveau `i` divisée par le nombre de noeuds du niveau `i`

Exemple d'application sur l'arbre de la figure 3 :

```
>>> get_average(B)
[0.0, 3.0, 4.75, 6.666666666666667]
```

4 Occurrences et numérotation hiérarchique

Numérotation hiérarchique (Hierarchical Numbering)

Exercice 7 (Implémentation hiérarchique)

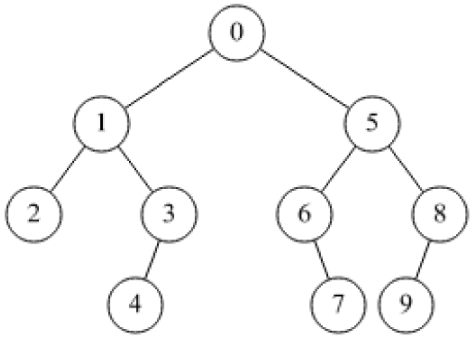

FIGURE 3 – Arbre binaire B

Comment représenter / implémenter un arbre binaire avec un simple vecteur ?

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

Exercice 8 (Object → List)

Écrire une fonction qui construit le vecteur (une liste en Python) contenant la représentation hiérarchique d'un arbre binaire (implémentation "objet" : `BinTree`). La valeur particulière `None` sera utilisée pour indiquer un arbre vide.

5 Tests

Exercice 9 (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) **Bonus :** Écrire une fonction qui teste si un arbre est parfait.

6 Construction

Exercice 10 (List → Object)

Écrire une fonction qui construit un arbre binaire (implémentation "objet" : `BinTree`) à partir du vecteur (une liste en Python) contenant sa représentation hiérarchique. La valeur particulière `None` est utilisée pour indiquer un arbre vide.

7 Bonus

Exercice 11 (Sérialisation : pour stocker les arbres dans des fichiers)

Afin d'enregistrer les arbres binaires dans des fichiers textes, il faut trouver une représentation unique "linéaire". On s'inspire de la représentation en type abstrait.

— L'arbre vide sera représenté par `"()"`

— L'arbre non vide $< r, G, D >$ sera représenté par la chaîne de caractères `"(rGD)"` où `G` est la représentation linéaire de l'arbre G et `D` celle de l'arbre D.

Exemple simple :

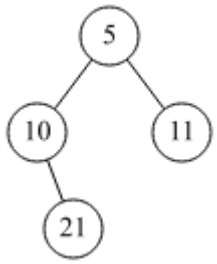

FIGURE 4 – B_2 : `"(5(10()(21()()))(11()()))"`

1. Écrire la fonction `to_linear(`B`)` qui retourne la représentation linéaire de l'arbre B (une chaîne de caractères).

Exercice 12 (BONUS : Load binary trees)

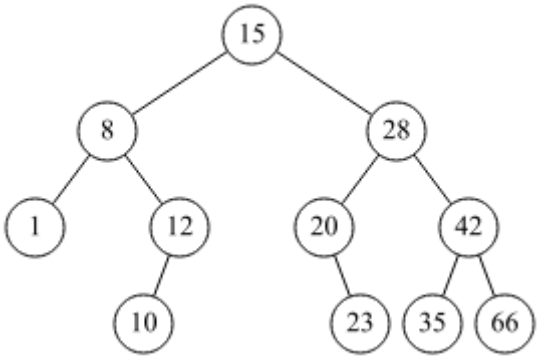

FIGURE 5 – `files/bst.bintree`

L'arbre ci-dessus a été chargé depuis un fichier texte contenant sa représentation linéaire :

```
1 (15(8(1()())(12(10()())()))(28(20()(23()()))(42(35()())(66()()))))
```

Écrire la fonction `load` qui à partir d'un tel fichier construit l'arbre.

Bonus : Ajouter une vérification que le fichier est correct.