Arbres Binaires et Généraux

Arbres binaires (Binary Trees)

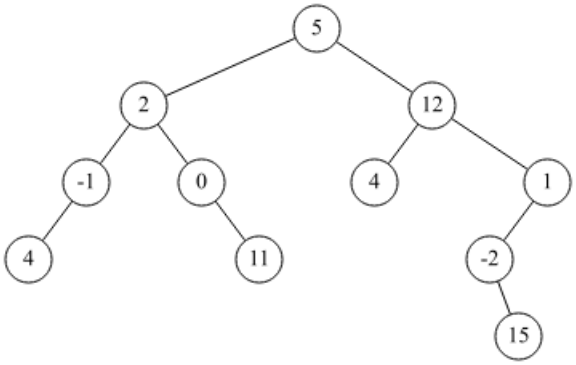

Figure 1 –

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.

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.

2 Parcours

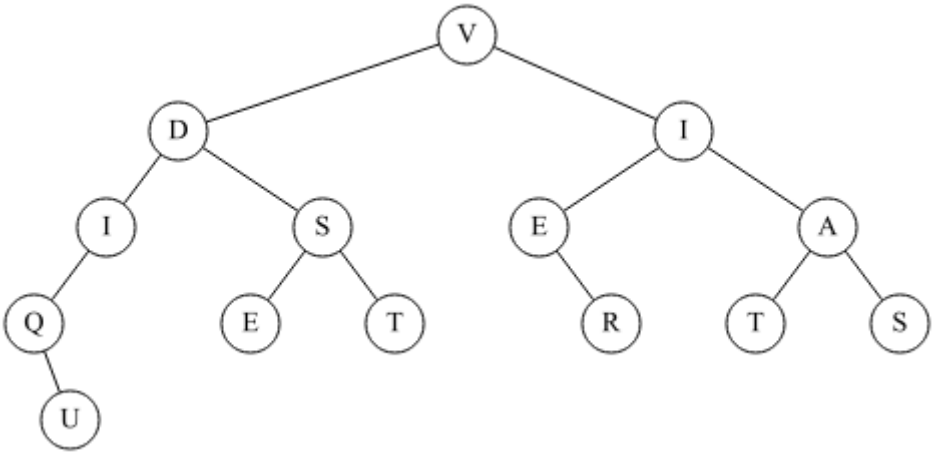

Figure 2 – Arbre binaire B pour parcours

Exercice 2.1 (DFS : 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.

Exercice 2.2 (BFS : Parcours largeur (Breadth-first Search))

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.
Que faut-il modifier pour afficher un niveau par ligne ?

3 Applications

Exercice 3.1 (Sérialisation)

Afin d'enregistrer les arbres binaires dans des fichiers textes, il faut trouver une représentation unique "linéaire". La représentation est basée sur celle en types abstraits simplifiée comme suit :
— 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.

1. Donner les représentations linéaires des arbres suivants :

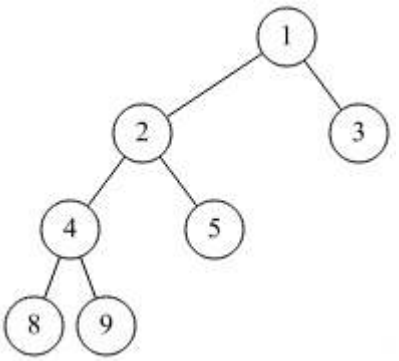

Figure 3 – B_2

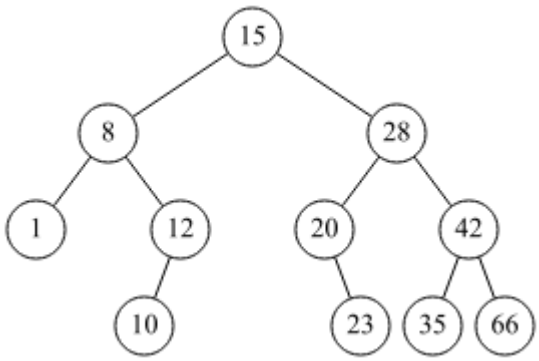

Figure 4 – B_3

2. Dessiner les arbres dont les représentations linéaires sont les suivantes :
   (a) `"(5(4(11(0()())(2()()))())(8(13()())()))"`
   (b) `"(15(8(5()(7()()))(12(10()())()))(25(20()())(42()(66()()))))"`

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

Exercice 3.2 (Longueurs de cheminement et Profondeurs moyennes)

1. Donner les définitions de la longueur de cheminement et de la profondeur moyenne d'un arbre binaire.
2. Écrire une fonction qui calcule la profondeur moyenne d'un arbre binaire.

4 Occurrences et numérotation hiérarchique

Numérotation hiérarchique (Hierarchical Numbering)

Exercice 4.1 (Recherche du numéro hiérarchique)

Écrire la fonction `search_hier(B, x)` qui prend en paramètre un arbre binaire `B` et une valeur `x` et qui retourne le numéro en ordre hiérarchique de `x` s'il est présent dans B, `None` sinon. Les clés de l'arbre binaire B sont supposées distinctes.

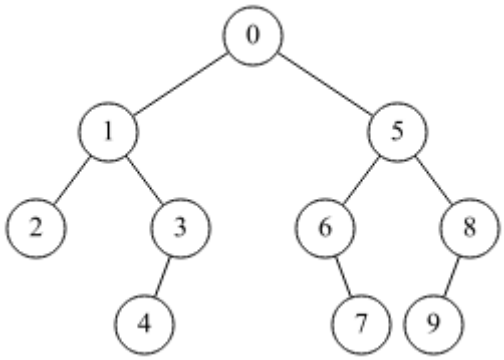

Figure 5 – Arbre binaire B

Exemples d'applications sur l'arbre de la figure 2 :

```
>>> print(search_hier(B, 10))
None
>>> search_hier(B, 3)
5
>>> search_hier(B, 0)
1
>>> search_hier(B, 7)
13
```

Exercice 4.2 (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.

Dans un premier temps, on pourra supposer que la taille de l'arbre est donnée.

Occurrences

Exercice 4.3 (Liste d'occurrences)

1. Donner la représentation sous forme de liste d'occurrences de l'arbre de la figure 8.

2. Écrire la fonction qui retourne la représentation par occurrences d'un arbre binaire (une liste de chaînes).

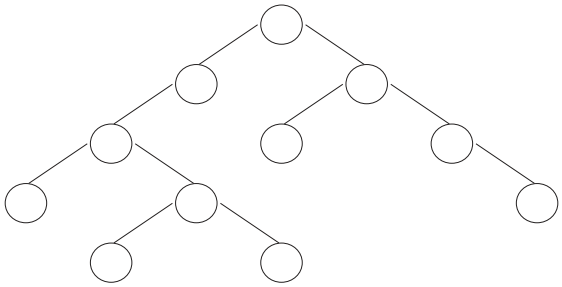

Figure 6 – Arbre binaire pour occurrences

Exercice 4.4 (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	f	H	m	n	u
code	0	1100	111	1101	101	100

   Décoder 1111001100110011010101

2. L'encodage est représenté par un arbre dont les **feuilles** sont les lettres à encoder (le champ `key` 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.

   (a) Écrire la fonction `code_list(`T`)` qui construit la liste des couples `(lettre, code)` à partir de l'arbre T.

   Exemple d'application avec `HT` l'arbre de la question 3 :

```
>>> code_list(HT)
[('a', '0'), ('u', '100'), ('n', '101'), ('H', '111'), ('f', '1100'),
   ('m', '1101')]
```

   (b) Écrire la fonction `search_code(`T`, `c`)` qui retourne le code de la lettre c si elle existe dans l'arbre T, la valeur `None` sinon.

   Exemples d'applications avec `HT` l'arbre de la question 3 :

```
>>> search_code(HT, 'a')
'0'
>>> search_code(HT, 'm')
'1101'
>>> print(search_code(HT, 'w'))
None
```

5 Tests

Exercice 5.1 (Dégénéré, parfait ou complet)

Pour chaque type d'arbre, donner le lien entre sa taille et sa hauteur.

1. **Arbre dégénéré (*degenerate*) :**
   (a) Rappeler ce qu'est un arbre dégénéré.
   (b) Écrire une fonction qui détermine si un arbre binaire est dégénéré.

2. **Arbre complet (*perfect*) :**
   (a) Donner plusieurs définitions d'un arbre complet.
   (b) Écrire une fonction qui teste si un arbre est complet connaissant sa hauteur.
   (c) 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 Constructions

Exercice 6.1 (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.

Exercice 6.2 (Transpose)

Le but de l'exercice est de transposer un arbre binaire :

— Transposer un arbre vide donne un arbre vide
— Transposer l'arbre non vide $A =< r, G, D >$ (r contenu du nœud racine de A) revient à créer un arbre $A_T =< r_T, G_T, D_T >$ tel que
  — $r_T = f(r)$ avec $f : x \mapsto 2x$
  — G_T est l'arbre D transposé
  — D_T est l'arbre G transposé

Écrire la fonction `transpose(`B`)` qui transpose l'arbre B selon la définition ci-dessus. La fonction retourne l'arbre résultat.

Exemple :

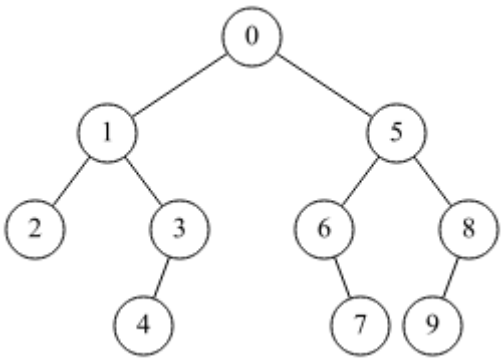

Figure 7 – B

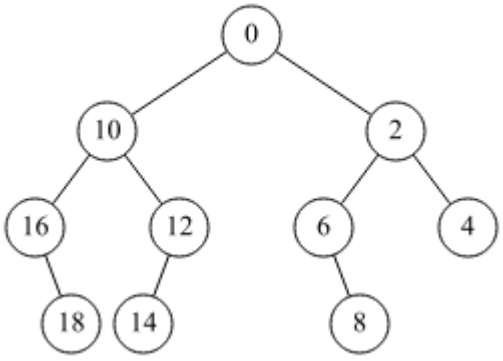

Figure 8 – `transpose(`B`)`

Exercice 6.3 (La taille en plus)

On ajoute une nouvelle implémentation des arbres binaires dans laquelle chaque nœud contient la taille (le champs `size`) de l'arbre dont il est racine : `BinTreeSize`.

```
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`).

7 Midterms and Bonus

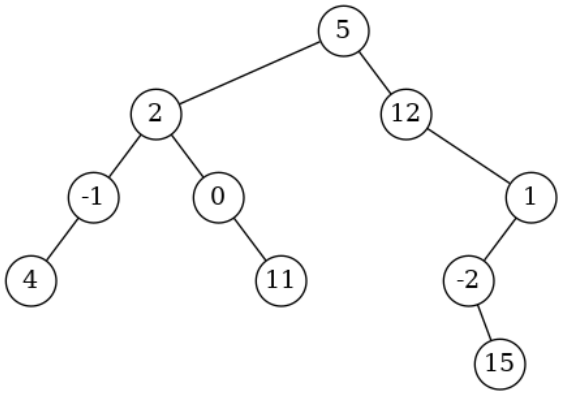

Figure 9 – Arbre binaire `B5`

Exercice 7.1 (Parenté)

Le degré de parenté entre deux noeuds d'un arbre binaire est le nombre de liens les séparant.

Écrire la fonction `get_kinship(B, x, y)` qui renvoie le degré de parenté entre les noeuds de l'arbre binaire `B` contenants les valeurs `x` et `y` s'ils sont sur la même branche, -1 sinon. Les clés de l'arbre binaire `B` sont supposées distinctes.

Exemples d'applications avec `B5` l'arbre de la figure 12 :

```
>>> get_kinship(B, 4, 11)
-1
>>> get_kinship(B, 42, 0)
-1
>>> get_kinship(B, 12, 15)
3
>>> get_kinship(B, -1, 5)
2
```

Exercice 7.2 (Somme)

Écrire la fonction `check_sum(`B`)` qui vérifie si chaque point double de l'arbre binaire `B` contient la somme des clés de ses deux fils. Le **parcours profondeur** doit obligatoirement être utilisé.

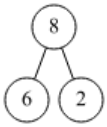

Figure 10 – B1

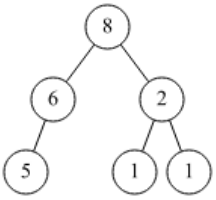

Figure 11 – B2

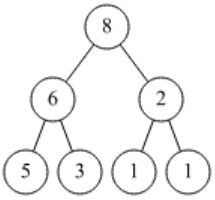

Figure 12 – B3

Exemples d'applications :

```
>>> check_sum(None)
True
>>> check_sum(B1)
True
>>> check_sum(B2)
True
>>> check_sum(B3)
False
```

Exercice 7.3 (n^{ieme} noeud)

Ecrire la fonction `get_node(B, lvl, n)` qui retourne la clé du n^{ieme} nœud (base 0) du niveau `lvl` de l'arbre binaire `B` s'il existe, `None` sinon. Les entiers `lvl` et `n` sont supposés positifs ou nuls.

Exemples d'applications avec `B5` l'arbre de la figure 12 :

```
>>> get_node(None, 0, 0) #None
>>> get_node(B5, 0, 0)
5
>>> get_node(B5, 2, 2)
1
>>> get_node(B5, 2, 5) #None
>>> get_node(B5, 4, 0)
15
```

Exercice 7.4 (BONUS : Load binary trees)

L'arbre de la figure 12 a été chargé depuis un fichier texte contenant sa représentation linéaire :

```
(5(2(-1(4()())())(0()(11()()))))(12()(1(-2()(15()()))())))
```

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

Bonus : Ajouter une vérification que le fichier est correct.

Annexes : types, méthodes et fonctions autorisées

Les arbres binaires

— L'arbre vide est `None`

— L'arbre non vide est (une référence sur) un objet de la class `BinTree` avec 3 attributs : `key`, `left`, `right`.

- `B` : classe `BinTree`
- `B.key` : contenu du nœud racine
- `B.left` : le sous-arbre gauche
- `B.right` : le sous-arbre droit

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

Files

Les méthodes de la classe `Queue`, que l'on suppose importée :
— `Queue()` retourne une nouvelle file ;
— q`.enqueue(`e`)` enfile e dans q ;
— q`.dequeue()` supprime et retourne le premier élément de q ;
— q`.isempty()` teste si q est vide.

Fonctions et méthodes autorisées

Vous pouvez utiliser la méthode `append` et la fonction `len` sur les listes ainsi que la fonction `range`.

```
>>> L = []

>>> for i in range(5):
        L.append(i)

>>> L
[0, 1, 2, 3, 4]

>>> len(L)
5

>>> for i in range(5, 10):
        L.append(i)
>>> L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> for i in range(9, -1, -1):
        print(i, end=" ")
9 8 7 6 5 4 3 2 1 0
```

Les fonctions `min` et `max`, mais uniquement avec deux valeurs entières !

Aucun opérateur n'est autorisé sur les listes (`+`, `*`, `==` ...).

Vos fonctions

Vous pouvez écrire des fonctions 'intermédiaires' / 'supplémentaires', dans ce cas vous devez donner leurs spécifications : on doit savoir ce qu'elles font.

Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.