1 En autonomie

Pour chacune des structures qui suivent :

- la shématiser ;
- indiquez comment la construire à partir des constructeurs vus en cours.

1. $[3, 1, 4]$
2. $[[3, 1], [], [4, 1, 5]]$
3. (1, Δ, Δ)
4. (3, (1, Δ, (4, (1, Δ, (5, Δ, Δ)), Δ)), Δ)

2 Opérations courantes sur les listes

2.1 Longueur d'une liste

Réalisez une fonction itérative `length` qui renvoie la longueur d'une liste.

2.2 n-ième élément

Réalisez une fonction nommée `nth` qui renvoie l'élément d'indice n d'une liste ℓ si cet élément existe, et déclenche une exception `IndexError` sinon, l'entier n et la liste ℓ étant les deux paramètres de cette fonction.

2.3 Dernier élément

Réalisez la fonction `last` qui renvoie le dernier élément d'une liste non vide et déclenche une exception si la liste est vide.

2.4 Concaténer deux listes

Concaténer deux listes ℓ_1 et ℓ_2, c'est construire une liste contenant les éléments de ℓ_1 suivis de ceux de ℓ_2.

Réalisez la fonction `concat` qui renvoie la concaténation des deux listes passées en paramètre.

2.5 Renverser une liste

Renverser une liste ℓ, c'est construire une liste qui contient les mêmes éléments que ℓ mais dans l'ordre inverse.

1. Réalisez la fonction `reverse` qui renvoie la liste passée en paramètre renversée, Écrivez une version récursive d'abord, puis récursive terminale.
2. Donnez une version récursive terminale de la fonction `concat`

2.6 Appliquer une fonction à tous les éléments d'une liste

Appliquer une fonction f à tous les éléments d'une liste ℓ, c'est construire une liste contenant les images $f(x)$ de tous les éléments x de ℓ.

Par exemple, appliquer la fonction carrée à la liste $[1, 2, 3]$ donne la liste $[1, 4, 9]$.

1. Réalisez une fonction nommée `map` qui applique une fonction aux éléments d'une liste.

2.7 Aplatir une liste de listes

Aplatir une liste de listes, c'est construire une liste contenant tous les éléments des listes de la liste. Par exemple, la liste aplatie correspondant à la liste $[[3, 1, 4], [], [1, 5]]$ est la liste $[3, 1, 4, 1, 5]$.

1. Réalisez une fonction nommée `flatten` qui renvoie la liste aplatie.

2.8 Conversions listes/listes

Réalisez une fonction qui convertit une liste python en une de nos listes, tout en conservant l'ordre des éléments dans la liste produite. Et réciproquement.

```
>>> l = native2list([3, 1, 4, 1, 5])
>>> l
ApLst(3, ApLst(1, ApLst(4, ApLst(1, ApLst(5, ApLst())))))
```

```
>>> list2native(l)
[3, 1, 4, 1, 5]
```

2.9 *Zip* de deux listes

Zipper [1] deux listes ℓ_1 et ℓ_2 c'est construire une liste de même longueur que la longueur supposée commune de ℓ_1 et ℓ_2 dont les éléments sont des couples dont la première composante est dans ℓ_1 et la seconde dans ℓ_2.

1. Réalisez une fonction nommée `zip` qui prend deux listes de même longueur en paramètre et les *zippe*.

En voici un exemple d'utilisation :

```
>>> l1 = native2list([1, 3, 5, 7])
>>> l2 = native2list(['Timoleon', 'Calbuth', 'Talon', 'Carmen'])
>>> l = zip(l1, l2)
>>> l
ApLst((1, 'Timoleon'), ApLst((3, 'Calbuth'), ApLst((5, 'Talon'), ApLst((7, 'Carmen'), ApLst()))))
```

2. Réalisez la fonction réciproque que vous nommerez `unzip`.

```
>>> l1,l2 = unzip(l)
>>> l1
ApLst(1, ApLst(3, ApLst(5, ApLst(7, ApLst()))))
>>> l2
ApLst('Timoleon', ApLst('Calbuth', ApLst('Talon', ApLst('Carmen', ApLst()))))
```

---

3 Exemples d'arbres binaires

Dans cet exercice, Δ désigne l'arbre vide.

Dessinez chacun des arbres binaires ci-dessous, donnez sa taille et sa hauteur, le nombre de feuilles, le nombre de nœuds à chaque profondeur.

1. $(1, \Delta, \Delta)$
2. $(3, (1, \Delta, (4, (1, \Delta, (5, \Delta, \Delta)), \Delta)), \Delta)$
3. $(3, (1, (1, \Delta, \Delta), \Delta), (4, (5, \Delta, \Delta), (9, \Delta, \Delta)))$
4. $(3, (1, (1, \Delta, \Delta), (5, \Delta, \Delta)), (4, (9, \Delta, \Delta), (2, \Delta, \Delta)))$

4 Lien entre hauteur et taille des arbres binaires

5. Combien de feuilles au minimum comporte un arbre binaire de hauteur h ?
   Au maximum ?
6. Combien de nœuds au minimum comporte un arbre binaire de hauteur h ?
   Au maximum ?

5 Prédicats dans les arbres binaires.

Dans cet exercice, on se donne une classe `ArbreBinaire` disposant :

- d'un constructeur acceptant zéro (arbre vide) ou trois paramètres (arbre non vide) tel que présenté en cours ;
- d'un prédicat de vacuité `est_vide` qui renvoie `True` ssi l'arbre est vide ;
- de deux sélecteurs : `gauche` et `droit` renvoyant respectivement les sous-arbres gauche et droit d'un arbre non vide ;
- d'un sélecteur `etiquette` qui renvoie la valeur contenue dans la racine d'un arbre d'un arbre non vide.

7. Écrire une fonction `est_feuille` d'entête :

```
def est_feuille(a: ArbreBinaire) -> bool:
    """Renvoie True ssi a est une feuille.

    Précondition : aucune.
```

[1] Rien à voir avec la compression des données, mais plutôt avec les fermetures éclairs.

```
    $$$ VIDE = ArbreBinaire()
    $$$ est_feuille(VIDE)
    False
    $$$ est_feuille(ArbreBinaire(1, VIDE, VIDE))
    True
    $$$ est_feuille(ArbreBinaire(1, VIDE, ArbreBinaire(2, VIDE, VIDE)))
    False
    """
```

8. Un arbre *localement complet* est un arbre dont tous les noeuds ont zéros ou deux fils.

Écrire un prédicat `est_localement_complet` d'entête :

```
def est_localement_complet(a: ArbreBinaire) -> bool:
    """Renvoie True ssi a est localement comple.

    Précondition : aucune

    $$$ VIDE = ArbreBinaire()
    $$$ est_localement_complet(VIDE)
    True
    $$$ fg = ArbreBinaire(1, VIDE, VIDE)
    $$$ est_localement_complet(fg)
    True
    $$$ a1 = ArbreBinaire(2, fg, VIDE)
    $$$ est_localement_complet(a1)
    False
    $$$ a2 = ArbreBinaire(3, fg, ArbreBinaire(4, VIDE, VIDE))
    $$$ est_localement_complet(a2)
    True
    """
```

6 Mesures dans les arbres binaires.

Dans cet exercice, on considère des arbres étiquetés par des entiers.

9. Écrire une fonction `nombre_feuilles` qui renvoie le nombre de feuilles d'un arbre binaire.
10. Écrire une fonction `nombre_profondeur` d'entête :

```
def nombre_profondeur(a: ArbreBinaire, p: int) -> int:
```

qui renvoie le nombre de noeuds de profondeur p d'un arbre a.

11. Écrire une fonction `arbre_max` prenant en paramètre un arbre non vide et qui renvoie la valeur maximale des étiquettes.
12. Écrire une fonction `nbre_occurrences` d'entête :

```
def nombre_occurrences(a: ArbreBinaire, et: int) -> int:
```

qui renvoie le nombre d'étiquette de l'arbre a égale à la valeur et

7 Manipulation d'arbres

Le miroir d'un arbre est un arbre obtenu en échangeant, pour chaque noeud, les sous-arbres gauche et droit.

13. Écrire une fonction `miroir` prenant en paramètre un arbre et qui renvoie son miroir.

```
def miroir(a: ArbreBinaire) -> ArbreBinaire:
    """Renvoie le miroir de a.

    Précondition : aucune
    Exemple(s):
    $$$ VIDE = ArbreBinaire()
    $$$ miroir(VIDE)
    VIDE
```

```
    $$$ ab1 = ArbreBinaire('a', ArbreBinaire('b', VIDE, VIDE), VIDE)
    $$$ miroir(ab1)
    ArbreBinaire('a', VIDE, ArbreBinaire('b', VIDE, VIDE))
    $$$ ab2 = ArbreBinaire('c', ArbreBinaire('d', VIDE, VIDE), ab1)
    $$$ miroir(ab2)
    ArbreBinaire('c', miroir(ab1), ArbreBinaire('d', VIDE, VIDE))
    """
```

14. Écrire une fonction `arbre_applique` d'entête

```
def arbre_applique(a: ArbreBinaire, f: Callable) -> ArbreBinaire:
    """Construit un nouvel arbre obtenu en appliquant f aux étiquettes de a.

    Précondtion : Les étiquettes sont du domaine de définition de f.

    Exemple(s):

    $$$ VIDE = ArbreBinaire()
    $$$ arbre_applique(VIDE, len)
    VIDE
    $$$ ab1 = ArbreBinaire('a', ArbreBinaire('ab', VIDE, VIDE), VIDE)
    $$$ arbre_applique(ab1, len)
    ArbreBinaire(1, ArbreBinaire(2, VIDE, VIDE), VIDE)
    """
```

qui construit un arbre de même forme que a obtenu en appliquant la fonction f aux étiquettes de a.

8 Parcours récursifs d'arbres binaires

Parcourir un arbre, c'est appliquer un traitement à chacun de ces noeuds dans un ordre défini par la structure de l'arbre.

Dans cet exercice, le traitement d'un noeud consiste à affiche son étiquette. On dispose d'un arbre représentant une expression arithmétique (le type des étiquettes est `str`).

```
>>> from arbrebinaire import *
>>> VIDE = ArbreBinaire()
>>> a = ArbreBinaire('*',
...         ArbreBinaire('+',
...             ArbreBinaire('1', VIDE, VIDE),
...             ArbreBinaire('x', VIDE, VIDE)),
...         ArbreBinaire('-',
...             ArbreBinaire('y', VIDE, VIDE),
...             ArbreBinaire('7', VIDE, VIDE)))
```

15. Dessinez l'arbre `a`.

voici trois affichages produits en appelant trois fonctions de parcours sur l'arbre `a` :

```
>>> from arbres_parcours import *
>>> parcours_infixe(a)
1 + x * y - 7
>>> parcours_postfixe(a)
1 x + y 7 - *
>>> parcours_prefixe(a)
* + 1 x - y 7
```

16. Écrire les fonctions `parcours_infixe`, `parcours_postfixe` et `parcours_prefixe`.