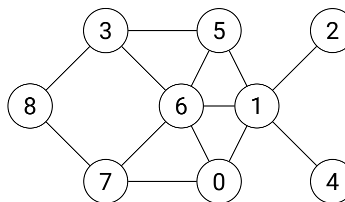
STRUCTURES DE DONNÉES

- Histoire de l'informatique
- Structures de données
- Bases de données
- Architectures matérielles et systèmes d'exploitation
- Langages et programmation
- Algorithmique

1. Introduction

1.1. L'exemple du parcours de graphe

Considérons le graphe de la figure suivante.



Ce graphe peut être décrit par une liste de listes d'adjacences :

```
G = [
  [1, 6, 7],        # voisins de 0
  [0, 2, 4, 5, 6],  # voisins de 1
  [1],              # voisins de 2
  [5, 6, 8],        # voisins de 3
  [1],              # voisins de 4
  [1, 3, 6],        # voisins de 5
  [0, 1, 3, 5, 7],  # voisins de 6
  [0, 6, 8],        # voisins de 7
  [3, 7]            # voisins de 8
]
```

Un des premiers algorithmes qu'on doit savoir utiliser sur un graphe est celui de son parcours. Parcourir un graphe, c'est visiter ses différents sommets, afin de pouvoir opérer une action tour à tour sur eux.
Dans la suite, nous supposons le graphe non orienté et connexe (pour deux sommets quelconques il existe toujours un chemin dans le graphe qui les relie). Nous supposons les sommets numérotés dans un intervalle d'entiers $[\![0, n-1]\!]$ et commençons le parcours avec le sommet 0.

Première approche

On peut écrire par exemple :

```
def visiter(k):
    # action à effectuer dans le parcours lors de la visite du sommet k
  print(k, end=' ')


def parcours(graphe):
  n = len(graphe)
  depuis = []
  depuis.append(0)
  dejavu = [False for k in range(n)]
  while not(len(depuis)==0):
    k = depuis.pop()
    if dejavu[k]:
      continue
    dejavu[k] = True
    visiter(k)
    for s in graphe[k]:
      depuis.append(s)
```

L'appel `parcours(G)` affiche la ligne `0 7 8 3 6 5 1 4 2`.
On dit qu'on a effectué un parcours en profondeur d'abord : on avance autant que possible, et on revient sur ses traces pour les sommets qu'on a laissés de côté préalablement.
Expliquons un peu ce qui se passe ici.
La liste `dejavu` permet de marquer les sommets déjà visités. Il s'agit d'une liste de booléens, initialisée avec `False`. Quand on visite un sommet k, on met à jour à `True` l'élément `dejavu[k]` et on visite effectivement le sommet : c'est le rôle des lignes 15 et 16.
En ligne 12, on récupère le numéro `k` d'un sommet à visiter (qu'on retire de la liste `depuis`) dont on abandonne l'examen (en ligne 14) s'il a déjà été vu.
Observons de plus près les valeurs successives de la liste `depuis`.

```
[]                                  # après la ligne 8
[0]                                 # après la  ligne 9
[]                                  # après la ligne 12, k vaut 0
[1, 6, 7]                           # après la boucle lignes 17-18
[1, 6]                              # après la ligne 12, k vaut 7
[1, 6, 0, 6, 8]                     # après la boucle lignes 17-18
[1, 6, 0, 6]                        # après la ligne 12, k vaut 8
[1, 6, 0, 6, 3, 7]                  # après la boucle lignes 17-18
[1, 6, 0, 6, 3]                     # après la ligne 12, k vaut 7, on passe en ligne 14
[1, 6, 0, 6]                        # après la ligne 12, k vaut 3
[1, 6, 0, 6, 5, 6, 8]               # après la boucle lignes 17-18
[1, 6, 0, 6, 5, 6]                  # après la ligne 12, k vaut 8, on passe en ligne 14
[1, 6, 0, 6, 5]                     # après la ligne 12, k vaut 6
[1, 6, 0, 6, 5, 0, 1, 3, 5, 7]      # après la boucle lignes 17-18
[1, 6, 0, 6, 5, 0, 1, 3, 5]         # après la ligne 12, k vaut 7, on passe en ligne 14
[1, 6, 0, 6, 5, 0, 1, 3]            # après la ligne 12, k vaut 5
[1, 6, 0, 6, 5, 0, 1, 3, 1, 3, 6] # après la boucle lignes 17-18
...
...
```

On observe que la liste `depuis` évolue par la droite : on extrait le dernier élément (c'est le rôle de la méthode `pop()`), on ajoute des éléments à la fin (c'est le rôle de la méthode `append`).

Deuxième approche

On aurait pu procéder autrement, en écrivant le programme suivant :

```
1  def parcours(graphe):
2    n = len(graphe)
3    depuis = []
4    depuis.append(0)
5    dejavu = [False for k in range(n)]
6    while not(len(depuis)==0):
7      k = depuis.pop(0)
8      if dejavu[k]:
9        continue
10     dejavu[k] = True
11     visiter(k)
12     for s in graphe[k]:
13       depuis.append(s)
```

Il ne diffère du premier que par la ligne 7 où on a écrit `depuis.pop(0)` au lieu de `depuis.pop()`.
C'est-à-dire qu'à chaque étape on sélectionne le sommet dont le numéro est au début de la liste `depuis`, autrement dit le plus anciennement introduit dans la liste, au contraire de la démarche précédente.
Cette fois, l'appel `parcours(G)` affiche `0 1 6 7 2 4 5 3 8`.
Il s'agit d'un parcours en largeur d'abord : on constate bien qu'après avoir visité le sommet 0, on attendra d'en avoir visité tous ses voisins, 1, 6 et 7, avant d'aller plus loin.

Discussion

On a quand même bien l'impression d'avoir écrit deux fois le même programme :

```
def parcours(graphe):
  # graphe est une liste de listes d'adjacence : graphe[k] est la liste des voisins du
  # sommet k
  n = len(graphe)
  depuis = CollectionVide()
  depuis.ajoute(0)
  dejavu = [False for k in range(n)]
  while not(estVide(depuis)):
    k = depuis.extrait()
    if dejavu[k]:
      continue
    dejavu[k] = True
    visiter(k)
    for s in graphe[k]:
      depuis.ajoute(s)
```

Ici on choisit d'utiliser trois méthodes, `CollectionVide`, `extrait` et `ajoute`,
L'appel `depuis = CollectionVide()` renvoie un objet `depuis` qui est une collection d'éléments, initialement vide.
L'appel `depuis.extrait()` choisit un élément de la collection qui en est retiré : l'objet `depuis` est modifié par cet appel.
L'appel `depuis.ajoute(k)` ajoute à la collection un nouvel élément : l'objet `depuis` est là aussi modifié par cet appel.
On a ainsi opéré sur une structure de données, qui gère ces trois méthodes. On dit qu'il s'agit d'une structure mutable (ou impérative) puisque les appels de méthodes peuvent modifier l'objet sur lequel elles s'appliquent.

Notons qu'on n'a pas défini du tout la sémantique concernant le choix de l'objet par la méthode `extrait()`.

C'est justement la possibilité d'utiliser deux sémantiques différentes qui permet au même programme (écrit avec `CollectionVide` ↪ , `extrait` et `ajoute`) de gérer deux types de parcours différents :

- le premier programme utilise la sémantique LIFO (*last in first out*), c'est-à-dire celle d'une pile. On extrait le dernier élément ajouté à la collection, ce qui est implémenté dans la liste `depuis` dont toute l'évolution a lieu en fin de liste ;
- le deuxième programme utilise la sémantique FIFO (*first in first out*), c'est-à-dire celel d'une file. On extrait l'élément le plus anciennement ajouté à la collection, ce qui est implémenté dans la liste `depuis` où l'ajout se fait en queue et l'extraction en tête de liste.

1.2. Une structure de données : la pile

Une pile est une structure de données définie par :

- son interface, c'est-à-dire la liste des méthodes qu'elle définit, à savoir : la création d'une pile vide, `estVide()` qui renvoie `True` si et seulement si la pile est vide, `ajoute(x)` qui permet d'ajouter un élément `x` à la pile et `extrait()` qui extrait un élément et renvoie sa valeur ;
- une sémantique, qui, pour la pile, est la sémantique LIFO que nous avons déjà décrite.

Le plus simple, en Python, est de la programmer à l'aide d'une classe [1] de la façon suivante :

```
class PileVide(Exception):
  """
    extraction à partir d'une pile vide
  """
  pass

class Pile:
  def __init__(self):
    self.memoire = []
    self.taille = 0

  def estVide(self):
    return self.taille == 0

  def ajoute(self,x):
    self.memoire.append(x)
    self.taille += 1
    return None

  def extrait(self):
    if self.taille == 0:
      raise PileVide
    else:
      x = self.memoire.pop()
      self.taille -= 1
      return x
```

On a défini une exception `PileVide` pour rendre la programmation plus sûre.
Notons qu'on aurait pu choisir une autre implémentation (c'est-à-dire une autre façon de programmer les méthodes de la classe) que celle que nous avons reprise de notre premier exemple d'introduction. L'important est de comprendre que, pour l'utilisateur qui n'a pas accès au texte de définition de la classe, mais seulement à son interface, c'est totalement indifférent. Il lui suffit de savoir qu'il

1. La programmation orientée objet est hors programme ; on ne présente les classes que sur quelques exemples, en guidant les élèves sur la syntaxe, et on utilise le vocabulaire classe, objet (ou instance de classe), méthode sans en donner de définition formelle.

créera une nouvelle pile vide p avec l'appel `p = Pile()`, qu'il pourra ajouter l'entier 3 dans la pile p en appelant `p.ajoute(3)` et qu'il pourra extraire l'élément ajouté le plus récemment avec l'instruction `x = p.extrait()`.
L'usage veut aussi qu'on précise, pour chaque implémentation, le coût d'exécution de chacune des méthodes proposées par l'interface : ici, tous les coûts sont unitaires.

1.3. Une autre structure de données : la file

La file a la même interface que la pile, mais une sémantique différente, FIFO, où la méthode `extrait()` va choisir l'élément le plus ancien résidant encore dans la file.
On pourrait définir une classe comme on vient de le faire pour la pile, en remplaçant simplement dans la définition de la méthode `extrait()` la ligne `x = self.memoire.pop()` par `x = self.memoire.pop(0)`, ainsi qu'on l'a expliqué dans l'introduction.
Mais pour bien montrer qu'il peut exister plusieurs implémentations pour une même structure de données, nous en proposons une autre.

```
class FileVide(Exception):
  """
    extraction à partir d'une file vide
  """
  pass

class File:
  def __init__(self):
    self.pA = Pile()
    self.pB = Pile()
    self.taille = 0

  def estVide(self):
    return self.taille == 0

  def ajoute(self,x):
    self.pA.ajoute(x)
    self.taille += 1
    return None

  def extrait(self):
    if self.taille == 0:
      raise FileVide
    elif self.pB.estVide():
      while not self.pA.estVide():
        self.pB.ajoute(self.pA.extrait())
    x = self.pB.extrait() # ce n'est pas une erreur d'indentation !
    self.taille -= 1
    return x
```

Les méthodes de création, de test de file vide, ou d'ajout, sont ici de coût unitaire.
La méthode `extrait` a une complexité plus difficile à maîtriser. On peut démontrer que le coût amorti [2] d'un appel à cette méthode est unitaire.

1.4. Utiliser un module Python

Il existe encore une autre solution : utiliser un module prédéfini de Python, comme `deque`.

2. La notion de coût amorti est hors programme. *Grosso modo*, il s'agit de la moyenne des coûts de plusieurs opérations successives.

Ce module s'importe avec la commande `from collections import deque` après laquelle on dispose des commandes suivantes :

- ▷ `d = deque([])` crée une collection vide ;
- ▷ `d.count()` renvoie le nombre d'éléments présents dans `d` ;
- ▷ `d.append(x)` et `d.appendleft(y)` permettent d'ajouter à d ou bien `x` à droite ou bien `y` à gauche ;
- ▷ `x = d.pop()` et `y = d.popleft()` permettent d'extraire le dernier élément `x` ou le premier `y`.

Ainsi, en utilisant le couple `append` et `pop` [3] on obtient le comportement d'une pile ; en utilisant le couple `append` et `popleft` [4] celui d'une file.
Il est important de remarquer que le module `deque` propose de nombreuses autres fonctions ou méthodes, que nous n'avons pas décrites ici. Mais seules celles qui sont utiles à la définition des piles et des files nous intéressent ici : c'est très souvent le cas avec des modules généralistes qui offrent beaucoup plus que ce qui est utile à la structure de données qu'on est en train d'étudier.
Quand on écrit soi-même des classes, il est recommandé de se limiter à implémenter ce qu'exige la définition de la structure de données.

2. Récapitulation

Une structure de données permet de gérer un ensemble de données à partir d'un jeu réduit de *méthodes* qui sont les seuls moyens d'accéder à tel ou tel élément, de modifier l'ensemble des données, d'en créer un nouveau, etc.
La description de ce jeu réduit de *méthodes* ainsi que de leur *sémantique* s'appelle *l'interface* de la structure de données.
Pour une même interface, on peut proposer diverses *implémentations* de la structure de données, et il est recommandé de pouvoir donner le coût d'exécution de chaque méthode dans le cadre d'une implémentation particulière.
L'utilisateur d'une structure de données n'a besoin de connaître que son interface, et nullement les détails de son implémentation.

Les modules Python proposent en général des interfaces très riches, ce qui généralise leur utilisation au détriment de la clarté : ce sont souvent plusieurs structures de données auxquelles il est donné accès par le biais d'un même module.

3. Les listes (chaînées)

3.1. Rappel historique

Citons cet extrait de Wikipedia.

> Le langage Lisp fut inventé par John McCarthy en 1958 alors qu'il était au Massachusetts Institute of Technology (MIT). Il publia un article intitulé *« Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I »* (soit *« Fonctions Récursives d'expressions symboliques et leur évaluation par une Machine, partie I »*) dans la revue CACM en 1960 ; la partie II ne fut jamais publiée.
> Le premier interpréteur fonctionnait sur un ordinateur IBM 704 et deux instructions de cette machine devinrent les deux opérations primitives de Lisp pour décomposer les listes :
>
> - ▷ *car* (contents of address register) : le premier élément de la liste ;
> - ▷ *cdr* (contents of decrement register) : le reste de la liste.
> - ▷ L'opération qui consiste à fabriquer une liste à partir d'un premier élément et d'une liste est notée *cons*.

Lisp, qui est toujours vivant aujourd'hui, s'appuie donc fortement sur les listes [5], structure de données que nous décrivons maintenant.

3. on pourrait aussi utiliser `appendleft` et `popleft`
4. on pourrait aussi utiliser `appendleft` et `pop`
5. on dit aussi listes (simplement) chaînées

3.2. Description informelle des listes

Remarque Python abuse du terme *liste* qu'il utilise pour ce qui sont des tableaux dynamiques munis de méthodes d'accès typiques des listes. Nous nous intéressons ici à ce que les informaticiens appellent vraiment des listes.

Une liste est une structure de données dont l'interface se réduit à peu de choses :

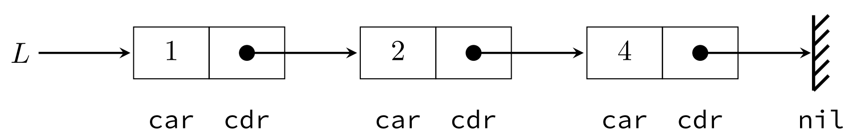
- ▷ une constante, la liste vide, souvent notée `nil` ;
- ▷ un test qui indique si la liste à laquelle il s'applique est vide ;
- ▷ deux accesseurs, historiquement appelés `car` et `cdr`, qui, appliqués à une liste, renvoie son premier élément et la liste (éventuellement vide) obtenue à partir de la liste initiale en supprimant son premier élément ;
- ▷ un constructeur, historiquement appelé `cons`, qui à un élément et une liste associe une nouvelle liste, dont l'élément fourni est le premier et la liste des suivants est celle qu'on a donnée.

Autrement dit, pour une liste L non vide, on a : L ≡ `cons(car(L),cdr(L))`.
C'est cette équivalence qui, s'il on veut, définit la sémantique de cette structure de données.
`car(L)` s'appelle la *tête* de la liste L, `cdr(L)` s'appelle sa *queue*.

Notons qu'il s'agit donc d'une structure linéaire, qu'on peut parcourir en partant du début, mais sans aucun accès direct à un élément autre que le premier. Ainsi, pour déterminer la longueur d'une telle liste, on est obligé de parcourir ses éléments tour à tour, jusqu'à tomber sur une liste donc la queue est la liste vide.

3.3. Entrons dans les détails

Voici la représentation schématique d'une liste comportant, dans cet ordre, les trois éléments 1, 2 et 4.



La liste est constituée de trois cellules, représentées ici par des rectangles.
Chaque cellule possède une première partie, nommée `car`, qui contient l'entier correspondant, et d'une deuxième partie, nommée `cdr`, qui contient un lien vers la cellule suivante.
On a donc affaire à une chaîne de cellules, ce qui justifie la dénomination courante de *liste chaînée*.
Le champ `cdr` de la dernière cellule renvoie vers une liste vide, conventionnellement représentée par un hachurage.
On aura noté que le premier champ d'une cellule contient ici un entier (car on considère une liste d'entiers) alors que le deuxième est une liste, c'est-à-dire un lien vers une autre cellule.
Rien n'interdit en réalité de mixer les types des champs `car` des différentes cellules, c'est-à-dire les types des éléments de notre liste chaînée.

En s'appuyant sur cette représentation, il est relativement aisé d'écrire le programme définissant la classe `Liste`. Il est important toutefois à bien séparer les deux classes `Liste` et `Cellule`, car le champ `cdr` d'une cellule n'est pas une cellule, mais bien une liste, c'est-à-dire, sur le schéma, un lien vers une cellule.
On définit tour à tour la méthode de test à vide (`estVide`), les méthodes d'accès à la tête et à la queue (`car` et `cdr`), le constructeur `cons`, et la constante `nil`.

```
class Cellule:
    def __init__(self, tete, queue):
        self.car = tete
        self.cdr = queue


class Liste:
    def __init__(self, c):
        self.cellule = c

    def estVide(self):
        return self.cellule is None

    def car(self):
        assert not(self.cellule is None), 'Liste vide'
        return self.cellule.car

    def cdr(self):
        assert not(self.cellule is None), 'Liste vide'
        return self.cellule.cdr


def cons(tete, queue):
    return Liste(Cellule(tete, queue))


nil = Liste(None)
```

Remarque : plutôt que de définir une exception particulière, on a recouru à la possibilité d'un deuxième argument lors de l'appel d'`assert`.

En s'appuyant sur cette interface, on peut définir quelques fonctions d'usage courant.

```
def longueurListe(L):
    n = 0
    while not(L.estVide()):
        n += 1
        L = L.cdr()
    return n


def listeElements(L):
    t = []
    while not(L.estVide()):
        t.append(L.car())
        L = L.cdr()
    return t
```

La fonction `listeElements` renvoie la liste (au sens de Python) des éléments d'une liste chaînée. C'est pratique pour vérifier que tout se passe bien.

Voici un exemple d'utilisation dans la console :

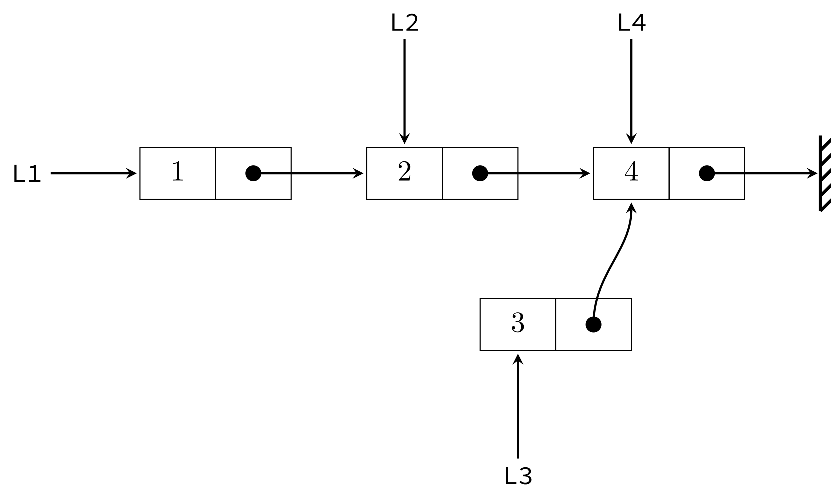
```
>>> L1 = cons(1, cons(2, cons(4, nil)))
>>> longueurListe(L1)
3
>>> listeElements(L1)
[1, 2, 4]
>>> L1.car()
1
>>> L1.cdr().cdr().car()
4
```

3.4. Une structure mutable

Une caractéristique intéressante de notre implémentation est qu'on obtient des listes mutables [6], ce qui permet par exemple d'insérer un élément en tête de liste ou même au milieu.
Par exemple, insérons l'élément 3 entre le 2 et le 4.
On commence par nommer quelques listes intermédiaires :

```
>>> L2 = L1.cdr()
>>> L4 = L1.cdr().cdr()
>>> L3 = cons(3, L4)
```

Nous sommes arrivés dans une situation qu'on peut représenter ainsi :



Il suffit maintenant de modifier la cellule pointée par `L2`.
Pour cela, on exécute l'affectation `L2.cellule = Cellule(L2.car(), L3)` et le tour est joué !
On vérifie :

```
>>> longueurListe(L1)
4
>>> listeElements(L1)
[1, 2, 3, 4]
```

On a bien modifié en place la liste `L1`.

6. On dit parfois qu'il s'agit d'un structure impérative. Le contraire est une structure immuable, ou encore persistante, ou encore fonctionnelle.

4. Un autre exemple : les arbres binaires

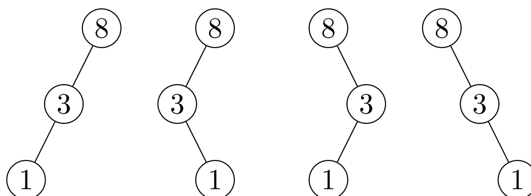
4.1. Généralités

Nous présentons ici une nouvelle structure, qui n'est plus linéaire, mais souvent rangée dans la catégorie des structures hiérarchiques, les arbres binaires. Pour simplifier nous supposerons que les données rangées dans les nœuds de la structure sont des entiers.
Il existe différents types d'arbres, et la nomenclature n'est pas universelle.
Par exemple, on peut trouver la définition suivante : un arbre est un graphe connexe sans cycle. Les arbres ainsi définis ne sont pas enracinés, ils n'ont pas une arité fixe (un nœud peut avoir un nombre variable de fils), et même pour un nœud possédant deux fils, il n'y a pas d'ordre sur les fils, donc ni fils gauche ni fils droit. Nous n'étudions pas ici ce type d'arbres.

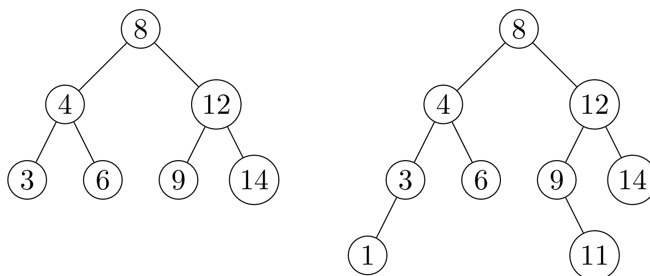
Définition

Un *arbre binaire* est défini de la façon suivante : ou bien il est vide, ou bien il est constitué d'un nœud *racine* qui possède deux fils, un fils gauche et un fils droit, qui sont tous les deux des arbres binaires.
Il s'agit donc d'une définition récursive.

Dans la figure suivante, tous les arbres dessinés sont deux à deux distincts, ce qui confirme qu'il faut distinguer fils droit et fils gauche, et qu'un des fils peut être vide.



Considérons les deux arbres de la figure suivante.



Le premier a pour racine un nœud contenant l'entier 8, la racine de son fils gauche contient l'entier 4, la racine de son fils droit contient l'entier 12.
Remarque : deux nœuds différents peuvent contenir la même valeur.

Le nombre de nœuds d'un arbre binaire s'appelle sa *taille*. Ainsi la taille de l'arbre de gauche est égale à 7, celle de l'arbre de droite est égale à 9.

On dit que la racine est à *profondeur* 0, les racines de ses fils à profondeur 1, etc.
Ainsi, pour l'arbre de droite, il y a un nœud à profondeur 0 (c'est 8), deux nœuds à profondeur 1 (ce sont 4 et 12), quatre nœuds à profondeur 2 (ce sont 3, 6, 9 et 14), et deux nœuds à profondeur 3 (1 et 11).
La profondeur maximale d'un nœud de l'arbre s'appelle la *hauteur* de l'arbre.
Ainsi l'arbre de gauche est de profondeur 2 et l'arbre de droite de profondeur 3.
Remarquons que la hauteur d'un arbre vide n'est pas définie [7].

On peut également définir la hauteur d'un arbre comme la longueur (en nombre d'arètes) du plus long chemin allant d'un nœud à la racine.

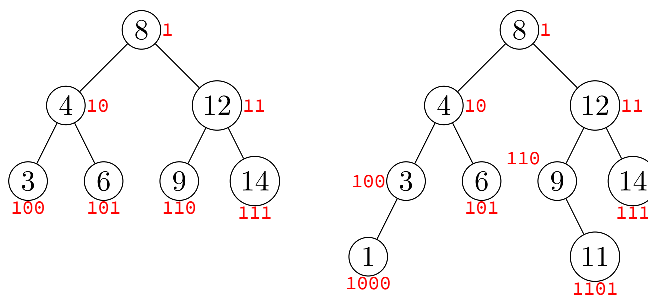
7. si on veut donner une profondeur à un arbre vide, il ne faut pas choisir 0 mais -1

Un étiquetage intéressant

On peut étiqueter les nœuds d'un arbre de la façon suivante :

- la racine porte l'étiquette `0` ;
- si un nœud quelconque porte l'étiquette `x...yz`, la racine de son éventuel fils gauche porte l'étiquette `x...yz0` et la racine de son éventuel fils droit l'étiquette `x...yz1`.

On observe que les étiquettes des nœuds de profondeur p sont constituées de $p+1$ bits, et sont deux à deux distinctes. On en déduit que toutes les étiquettes des nœuds d'un même arbre sont deux à deux distinctes.
Les étiquettes peuvent être vues comme les écritures de numéros en base 2 : on a ainsi numéroté les différents nœuds de l'arbre.
Voici le résultat de cet étiquetage sur les deux arbres que nous avons déjà considérés.



Dans le cas de l'arbre de gauche, si on traduit en décimal les étiquettes des nœuds, on constate qu'on les a simplement numérotés de 1 à 7, de haut en bas et de gauche à droite.
Pour l'arbre de droite, on a pioché dans les numéros de 1 à 15, sans utiliser les numéros 9, 10, 11, 12, 14, 15 qui correspondent à des nœuds absents.
Un arbre binaire de hauteur h tel qu'à chaque profondeur p tous les nœuds sont présents (il y en a 2^p) est dit *complet*. C'est le cas de l'arbre de gauche de la figure ci-dessus.
Un arbre binaire qui est complet sauf à la profondeur $p = h$ où tous les nœuds sont à gauche sera étiqueté avec tous les entiers de 1 à n, sans « trou » dans la numérotation : on dira que c'est un arbre bien tassé [8]. L'arbre de droite de la figure n'est pas bien tassé : il manque quatre nœuds (le fils droit de 3, les deux fils de 6 et le fils gauche de 9).

Relation entre taille et profondeur

Pour une taille n fixée, la hauteur maximale d'un arbre binaire est $h = n - 1$, qu'on obtient avec des arbres « filiformes » comme ceux de la première figure que nous avons dessinée.
Les arbres de taille n de hauteur minimale sont les arbres bien tassés. On a vu qu'alors l'étiquetage proposé utilise tous les entiers de 1 à n. Or l'étiquette du dernier nœud, de profondeur $p = h$, est l'écriture binaire de n, et s'écrit — on l'a déjà remarqué — avec $p+1$ bits. Donc pour cet arbre, on a $h = \lfloor \log_2(n) \rfloor$.
En effet, le logarithme en base 2 d'un entier n n'est autre [9] que le nombre de bits nécessaire à son écriture en base 2 diminué d'une unité.
On a donc pour n'importe quel arbre binaire de hauteur h et de taille n les inégalités :

$$\lfloor \log_2(n) \rfloor \leqslant h \leqslant n - 1.$$

8. cette terminologie n'est pas standard
9. C'est la définition du logarithme pour les informaticiens.

4.2. Une classe pour les arbres binaires

On programme une classe `Noeud` (similaire à la classe `Cellule` de la partie sur les listes) et une classe `ArbreBinaire` (similaire à la classe `Liste`).

```
class Noeud:
    def __init__(self, valeur, gauche, droit):
        self.r = valeur
        self.g = gauche
        self.d = droit


class ArbreBinaire:
    def __init__(self, c):
        self.n = c

    def creeVide():
        return ArbreBinaire(None)

    def creeNGD(valeur, gauche, droit):
        return ArbreBinaire(Noeud(valeur, gauche, droit))

    def estVide(self):
        return self.n is None

    def racine(self):
        assert not(self.n is None), 'Arbre vide'
        return self.n.r

    def filsGauche(self):
        assert not(self.n is None), 'Arbre vide'
        return self.n.g

    def filsDroit(self):
        assert not(self.n is None), 'Arbre vide'
        return self.n.d


vide = ArbreBinaire(None)

def creeFeuille(x):
    return ArbreBinaire.creeNGD(x, vide, vide)
```

On a défini une fonction `creeFeuille` pour créer un arbre réduit à un nœud-racine sans fils, pour simplifier l'utilisation de ces classes.

Il est alors facile de définir des fonctions récursives calculant la taille ou la profondeur d'un arbre binaire.

```
def taille(a):
    if a.estVide():
        return 0
    else:
        return 1 + taille(a.filsGauche()) + taille(a.filsDroit())

def profondeur(a):
    if a.estVide():
        return -1
    else:
        return 1 + max(profondeur(a.filsGauche()), profondeur(a.filsDroit()))
```

Voici un exemple d'utilisation.

```
>>> a1 = ArbreBinaire.creeNGD(4, ArbreBinaire.creeNGD(3, creeFeuille(1), vide),
   ↪ creeFeuille(6))
>>> a2 = ArbreBinaire.creeNGD(12, ArbreBinaire.creeNGD(9, vide, creeFeuille(11)),
   ↪ creeFeuille(14))
>>> a = ArbreBinaire.creeNGD(8, a1, a2)
>>> taille(a)
9
>>> profondeur(a)
3
>>> a.filsDroit().filsGauche().racine()
9
>>> a.filsGauche().filsGauche().filsGauche().racine()
1
```

On a ici construit l'arbre de droite de la figure qu'on a nommé `a`, en commençant par construire son sous-arbre gauche `a1` et son sous-arbre droite `a2`.

4.3. Une structure de données qui s'appuie sur la classe des arbres binaires

Sur cette base on peut définir une structure de données appelée *arbre binaire de recherche*.
Cette structure permet

- ▷ la création d'une structure vide ;
- ▷ l'ajout d'un élément à la structure de données ;
- ▷ la recherche d'un élément, pour savoir s'il est présent ou non dans la structure.

On pourrait ajouter d'autres méthodes, l'une listant les éléments de la structure (assez facile), l'autre supprimant un élément (plus difficile) ; nous laissons leur programmation en exercice.
La sémantique demande une petite précision : on peut ajouter un élément qui existe déjà, il sera alors présent deux fois dans la structure.
Pour l'implémentation, on utilise un arbre binaire, qui vérifie de plus la propriété suivante : les éléments qui figurent dans le fils gauche d'un arbre sont tous inférieurs ou égaux à la racine, ceux qui figurent dans le fils droit sont tous supérieurs ou égaux à la racine. En outre, chaque sous-arbre vérifie également cette propriété.
Un arbre binaire qui vérifie cette propriété est appelé *arbre binaire de recherche*.
Remarquons que les arbres dessinés dans la figure précédente sont déjà tous des arbres binaires de recherche.

La recherche d'un élément

Cette propriété facilite la recherche d'un élément : s'il est égal à la racine de l'arbre, on l'a trouvé ; s'il est inférieur, on le recherche récursivement dans le fils gauche ; s'il est supérieur, on le recherche récursivement dans le fils droit.

Le nombre de comparaisons effectuées dans la recherche est donc borné par $h+1$ où h est la hauteur de l'arbre. Il est donc de l'ordre du logarithme de sa taille n pour peu que l'arbre soit équilibré [10].
On crée une classe ABR qui hérite de la classe `ArbreBinaire`.

```
class ABR(ArbreBinaire):
    def __init__(self,c):
        ArbreBinaire.__init__(self,c)

    def creeNGD(valeur, gauche, droit):
        return ABR(Noeud(valeur, gauche, droit))

    def cherche(self,x):
        if ArbreBinaire.estVide(self):
            return False
        elif x == self.racine():
            return True
        elif x < self.racine():
            return self.filsGauche().cherche(x)
        else:
            return self.filsDroit().cherche(x)

    # def ajoute(self,x):
    # voir plus bas
    #   ...
    #   ...

def creeFeuille(x):
  return ABR.creeNGD(x, ABR(None), ABR(None))
```

On peut tester, en prenant garde d'avoir bien redéfini comme ci-dessus `vide` et `creeFeuile`.

```
>>> a1 = ABR.creeNGD(4, ABR.creeNGD(3, creeFeuille(1), vide), creeFeuille(6))
>>> a2 = ABR.creeNGD(12, ABR.creeNGD(9, vide, creeFeuille(11)), creeFeuille(14))
>>> a = ABR.creeNGD(8, a1, a2)
>>> a.cherche(9)
True
>>> a.cherche(1)
True
>>> a.cherche(2)
False
```

10. il existe de nombreux algorithmes qui garantissent l'équilibrage de l'arbre, et donc l'ordre de grandeur du coût d'une recherche, mais ils dépassent l'ambition de ce document

L’ajout d’un élément

L’ajout d’un élément est une méthode un peu plus compliquée à programmer.
L’idée est de commencer comme par la recherche — sans s’arrêter si l’élément est déjà présent puisqu’on est convenu d’accepter les doublon — et d’effectuer l’ajout quand on arrive à un arbre vide.
Voici la déclaration complète de la classe ABR qu’on obtient ainsi.

```
class ABR(ArbreBinaire):
    def __init__(self,c):
        ArbreBinaire.__init__(self,c)

    def creeNGD(valeur, gauche, droit):
        return ABR(Noeud(valeur, gauche, droit))

    def cherche(self,x):
        if ArbreBinaire.estVide(self):
            return False
        elif x == self.racine():
            return True
        elif x < self.racine():
            return self.filsGauche().cherche(x)
        else:
            return self.filsDroit().cherche(x)

    def ajoute(self,x):
        if ArbreBinaire.estVide(self):
            self.n = Noeud(x, ABR(None), ABR(None))
        elif x <= self.racine():
            self.filsGauche().ajoute(x)
        else:
            self.filsDroit().ajoute(x)
```