Chapitre :
Structures de données

I. Structures linéaires et dictionnaires

1. Tableaux

La structure de tableau permet de stocker des séquences d'éléments de manière contiguë et ordonnée en mémoire. C'est le type `list` en Python.

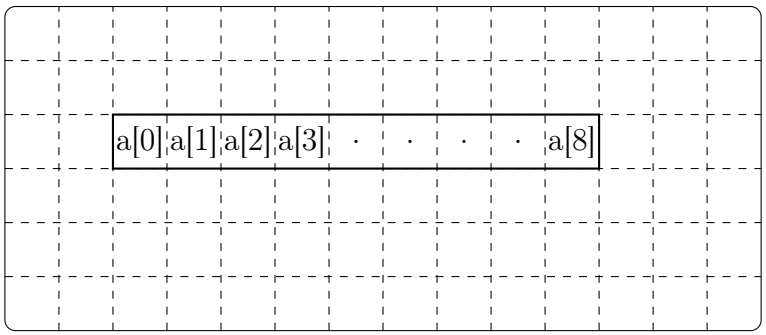
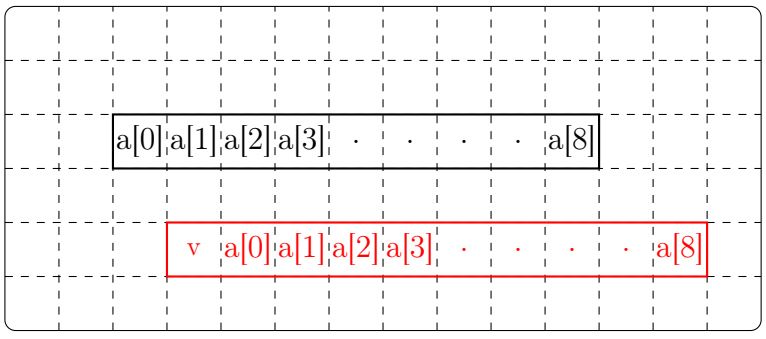


Tableau `a` de 9 éléments stocké en mémoire

Accéder à un élément d'un tableau se fait alors à coût constant, il suffit de connaître l'adresse du premier élément et la taille d'un élément. Par contre, le tableau se prête mal à l'ajout d'un élément. Si l'on reprend notre tableau précédent et qu'on aimerait insérer la valeur `v` à la première position, la case adjacente à `a[0]` étant très probablement occupée par une autre donnée, il faut réserver de la place ailleurs en mémoire, recopier tout le tableau avec la valeur `v` en première position.



Au total, on a réalisé un nombres d'opérations proportionnel à la taille du tableau.
Nous allons étudier dans la partie suivante les **listes chaînées**, qui apportent une meilleure solution au problème de l'insertion au début d'une séquence mais qui nous servira également de brique de base à plusieurs autres structures.

2. Listes chaînées

Une **liste chaînée** représente une liste finie de valeurs où les éléments sont chaînés entre eux. Chaque élément est stocké dans un bloc alloué quelque part en mémoire appelé maillon ou **cellule**, et est accompagné d'une deuxième information : l'adresse mémoire de la cellule suivante.

Liste chaînée contenant les éléments 1, 2 et 3.

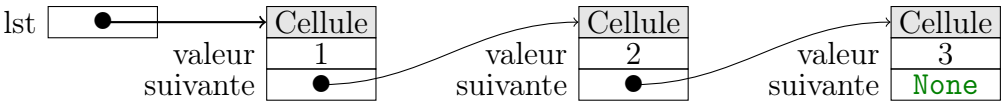
Une façon de représenter une liste chaînée en Python consiste à utiliser une classe `Cellule`.

```
class Cellule:
  '''Une cellule d'une liste chaînée'''
  def __init__(self, v, s):
    self.valeur = v
    self.suivante = s
```

Pour la dernière cellule, on donne la valeur `None` à l'attribut `suivante`.
Pour construire une liste il suffit d'appliquer le constructeur de la classe `Cellule` autant de fois qu'il y a d'éléments dans la liste. Ainsi, l'instruction :

```
lst = Cellule(1, Cellule(2, Cellule(3, None)))
```

construit la liste 1, 2, 3 donnée en exemple et la stocke dans une variable `lst`.
Plus précisément, on a créé trois objets de la classe `Cellule`, que l'on peut visualiser comme suit :

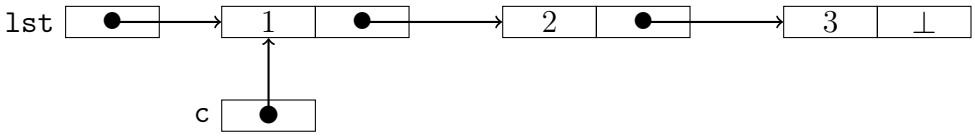


a. Premières opérations sur les listes chaînées

Longueur d'une liste

Pour calculer la longueur d'une liste, on parcours la liste de la première à la dernère cellule en suivant les liens qui relient ces cellules entre elles.
On commence par se donner deux variables : une variable `c` contenant la cellule courante du parcours et une variable `l` contenant la longueur du parcours déjà réalisé.

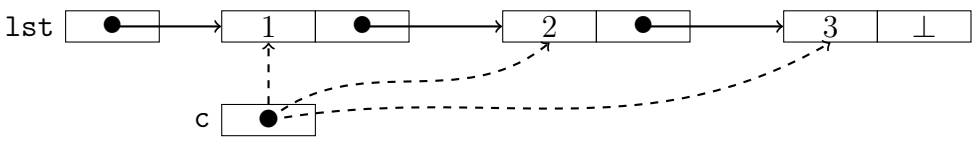
```
def longueur(lst):
  '''renvoie la longueur de la liste lst'''
  c = lst
  l = 0
```



La variable `c` va parcourir à l'aide d'une boucle `while` la liste, en incrémentant `l` à chaque itération.

```
  while c is not None:
    l += 1
    c = c.suivante
```

On s'arrête lorsque `c` est `None`, ce qui correspond bien à la fin de la liste.



Il suffit alors de retourner `l`.

```
def longueur(lst):
  '''renvoie la longueur de la liste lst'''
  c = lst
  l = 0
  while c is not None:
    l += 1
    c = c.suivante
  return l
```

Une autre solution consiste à parcourir la liste de manière récursive. En effet, si la liste est vide sa longueur est 0.

```
  if lst is None:
    return 0
```

Sinon, il faut renvoyer 1 (pour la case en cours de parcours) plus la longueur du reste de la liste.

```
  else:
    return 1 + longueur(lst.suivante)
```

```
def longueur(lst):
  '''renvoie la longueur de la liste lst'''
  if lst is None:
    return 0
  else:
    return 1 + longueur(lst.suivante)
```

La complexité du calcul de la longueur est directement proportionnelle à la longueur elle même. Ainsi pour une liste `lst` de mille cellules, `longueur(lst)` va effectuer mille tests, mille additions et deux milles affectations dans sa version itérative.

N-ième élément d'une liste

Comme pour les tableaux, nous allons prendre comme convention que le premier élément a pour indice 0. On cherche à écrire une fonction de la forme suivante :

```
def nieme_element(n, lst):
  '''renvoie le n-ième élément de la liste lst
  les éléments sont numérotés à partir de 0'''
```

Comme pour la fonction `longueur`, nous avons le choix d'écrire la fonction `nieme_element` comme une fonction récursive ou itérative. Nous faisons ici le choix d'une fonction récursive ; Un exercice proposera d'écrire cette fonction avec une boucle.
La condition d'arrêt s'obtient en considérant le cas où `n = 0`, le premier élément de la liste est alors renvoyé.

```
  if n == 0:
    return lst.valeur
```

Sinon, il faut continuer la recherche dans le reste de la liste. On passe à l'élément suivant en diminuant `n` de 1.

```
  else:
    return nieme_element(n-1,lst.suivante)
```

Reste encore à gérer le cas où la liste est vide, dans ce cas l'indice donné est invalide et on lève une exception `IndexError`.

```
  if lst is None:
    raise IndexError("indice invalide")
```

```
def nieme_element(n, lst):
  '''renvoie le n-ième élément de la liste lst
  les éléments sont numérotés à partir de 0'''
  if lst is None:
    raise IndexError("indice invalide")
  if n == 0:
    return lst.valeur
  else:
    return nieme_element(n-1,lst.suivante)
```

b. Encapsulation dans une classe Liste

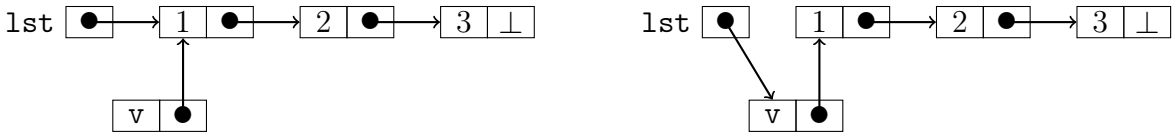
Pour terminer ce chapitre, nous allons définir une classe `Liste`.
Cette classe possède un unique attribut `tete` qui correspond à la tête de la liste.

```
class Liste:
  '''une liste chaînée'''
  def __init__(self):
    self.tete = None
```

Ajoutons une méthode qui permet d'ajouter un élément en tête de liste.

```
  def ajoute(self,v):
    '''ajoute la valeur v en tête de la liste'''
```

On aimerait ajouter la valeur `v` en tête de liste. Pour cela on crée une nouvelle cellule qui a pour valeur `v` et qui a pour cellule suivante la cellule pointé par `self.tete`, la tête de liste. Cette nouvelle cellule devient ensuite notre nouvelle tête :



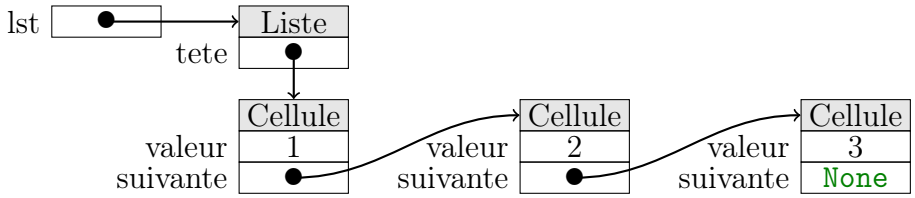

```
    self.tete = Cellule(v, self.tete)
```

L'ajout d'un élément en tête d'une liste chaînée se fait à coût constant.

```
  def ajoute(self, v):
    self.tete = Cellule(v, self.tete)
```

Si par exemple on exécute les quatre instructions de gauche, on obtient la situation représentée à droite :

```
lst = Liste()
lst.ajoute(3)
lst.ajoute(2)
lst.ajoute(1)
```



Les fonctions `longueur` et `nieme_element` vues à la section précédente vont nous permettre de définir les méthodes `__len__` et `__getitem__` :

```
  def __len__(self):
    return longueur(self.tete)
  def __getitem__(self, i):
    return nieme_element(i, self.tete)
```

On pourra alors écrire comme pour un tableau `lst[i]` et `len(lst)`.
Beaucoup d'autres méthodes sont encore possibles, comme celles proposées dans les exercices.
Voici une première classe `List` :

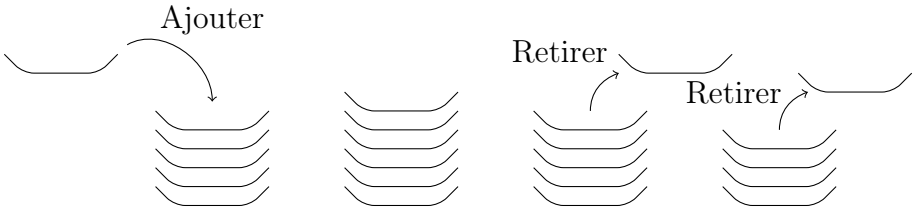
```
class Liste:
  '''une liste chaînée'''
  def __init__(self):
    self.tete = None

  def ajoute(self, v):
    self.tete = Cellule(v, self.tete)

  def __len__(self):
    return longueur(self.tete)

  def __getitem__(self, i):
    return nieme_element(i, self.tete)

  def est_vide(self):
    return self.tete is None
```

On pourra ajouter, à titre d'exercice, une méthode `__setitem__(self, i)`.

3. Piles

a. Interface

Une **pile** est une structure de données qui permet de stocker des éléments de même type. On l'associe souvent à l'image d'une pile d'assiettes ; chaque nouvelle assiette est ajoutée au-dessus des précédentes, et l'assiette retirée est systématiquement celle du sommet. On qualifie ce comportement de « dernier entré, premier sorti » ou encore LIFO (Last In, First Out).

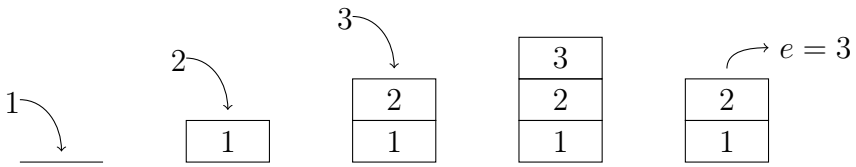


L'opération d'ajout d'un élément au sommet d'une pile est traditionnellement appelée **empiler** (ou **push** en anglais). L'opération de retrait de l'élément au sommet de la pile est appelé **dépiler** (ou **pop** en anglais).
Les deux autres opérations élémentaires sur les piles sont `creer_pile` qui crée une pile vide et `est_vide` qui teste si la pile est vide.
L'interface d'une pile est alors la suivante :

fonction	description
`creer_pile()`	crée et renvoie une pile vide
`est_vide(p)`	renvoie `True` si la pile est vide et `False` sinon
`empiler(p, e)`	ajoute `e` au sommet de `p`
`depiler(p)`	retire et renvoie l'élément au sommet de `p` si `p` n'est pas vide, et lève une exception sinon

Si par exemple on exécute les instructions de gauche, on obtient la situation représentée à droite :

```
pile = creer_pile()
empiler(pile, 1)
empiler(pile, 2)
empiler(pile, 3)
e = depiler(pile)
```



Utilisation d'une pile : bouton retour en arrière

Considérons un navigateur web pour lequel on s'intéresse à deux opérations : aller à une nouvelle page et revenir à la page précédente.
En plus de l'adresse courante, qui peut être stockée dans une variable à part, il nous faut donc conserver l'ensemble des pages précédentes auxquelles il est possible de revenir. Puisque le retour en arrière se fait vers la dernière page qui a été quittée, le comportement est de la forme « dernier entré, premier sorti », les adresses précédentes peuvent donc être stockées dans une pile.

```
adresse_courante = ""
adresses_precedentes = creer_pile()
```

Lorsqu'on navigue vers une nouvelle page, l'adresse courante est ajoutée au sommet de la pile, l'adresse cible devient alors la nouvelle adresse courante.

```
def aller_a(adresse_cible):
  empiler(adresses_precedentes, adresse_courante)
  adresse_courante = adresse_cible
```

Pour revenir en arrière, il suffit de revenir à l'adresse au sommet de la pile.

```
def retour():
  if not est_vide(adresses_precedentes):
    adresse_courante = depiler(adresses_precedentes)
```

b. Réalisation à l'aide d'un tableau Python

Les tableaux Python réalisent directement une structure de pile avec leurs opérations `append` et `pop`. On peut ainsi construire une classe `Pile` définie avec un unique attribut `contenu` associé à l'ensemble des éléments de la pile, stockés sous la forme d'un tableau Python.
La pile vide est alors définie par un `contenu` correspondant au tableau vide.

```
class Pile:
  '''structure de pile'''
  def __init__(self):
    self.contenu = []
```

On peut ainsi tester si la pile est vide en regardant la taille de son `contenu`.

```
  def est_vide(self):
    return len(self.contenu) == 0
```

Les opérations `append` et `pop` réalisent respectivement les opérations `empiler` et `depiler`.

```
  def empiler(self, e):
    self.contenu.append(e)

  def depiler(self):
    if self.est_vide():
      raise IndexError('dépiler sur une pile vide')
    else:
      return self.contenu.pop()
```

On résume l'ensemble de la classe dans le programme ci-dessous.

```
class Pile:
  '''structure de pile'''
  def __init__(self):
    self.contenu = []

  def est_vide(self):
    return len(self.contenu) == 0

  def empiler(self, e):
    self.contenu.append(e)

  def depiler(self):
    if self.est_vide():
      raise IndexError('dépiler sur une pile vide')
    else:
      return self.contenu.pop()
```

c. Réalisation à l'aide d'une liste chaînée

La méthode précédente est raisonnable dans le cadre d'un programme Python, les opérations `append` et `pop` s'exécutant en moyenne en temps constant. Cette solution ne s'exporte cependant pas à n'importe quel autre langage.

La structure de liste chaînée donne une manière élémentaire de réaliser une pile. Empiler un élément revient à ajouter un élément en tête et dépiler un élément revient à supprimer l'élément de tête. L'attribut `contenu` est alors initialisé à la liste vide, c'est à dire `None`.

```
class Pile:
  '''structure de pile'''
  def __init__(self):
    self.contenu = None
```

La pile est vide si son contenu est vide.

```
  def est_vide(self):
    return self.contenu is None
```

Empiler un élément, c'est ajouter un élément en tête de la liste chaînée `contenu`.

```
  def empiler(self, e):
    self.contenu = Cellule(e,self.contenu)
```

Dépiler revient à supprimer l'élément en tête de la liste chaînée `contenu` si celle-ci n'est pas vide.

```
  def depiler(self):
    if self.est_vide():
      raise IndexError('dépiler sur une pile vide')
    else:
      v = self.contenu.valeur
      self.contenu = self.contenu.suivante
      return v
```

On résume l'ensemble de la classe dans le programme ci-dessous.

```
class Cellule:
  '''une cellule d'une liste chaînée'''
  def __init__(self, v, s):
    self.valeur = v
    self.suivante = s

class Pile:
  '''structure de pile'''
  def __init__(self):
    self.contenu = None

  def est_vide(self):
    return self.contenu is None

  def empiler(self, e):
    self.contenu = Cellule(e,self.contenu)

  def depiler(self):
    if self.est_vide():
      raise IndexError('dépiler sur une pile vide')
    else:
      v = self.contenu.valeur
      self.contenu = self.contenu.suivante
      return v
```

On peut alors définir les fonctions de l'interface définie plus haut :

```
def creer_pile():
  return Pile()

def est_vide(p):
  return p.est_vide()

def empiler(p, e):
  p.empiler(e)

def depiler(p):
  return p.depiler()
```

4. Files

a. Interface

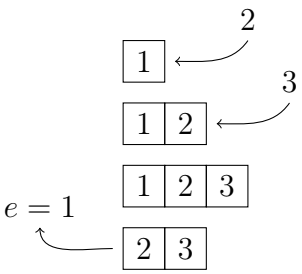
Une **file** est une structure de donnée qui permet de stocker des éléments de même type. On l'associe souvent à l'image d'une file d'attente ; chaque nouvelle personne arrive en fin de file, et la personne suivante à être servi est celle en tête de file. On qualifie ce comportement de « premier entré, premier sorti » ou encore FIFO (Firt In, First Out).

Les deux opérations élémentaires dont on a besoin pour réaliser cette structure est ajouter un élément en fin de file et retirer l'élément en tête de file. Comme pour les piles, il faut également pouvoir créer une file vide et tester si une file est vide. On obtient alors l'interface suivante :

fonction	description
`creer_file()`	crée et renvoie une file vide
`est_vide(f)`	renvoie `True` si la file est vide et `False` sinon
`ajouter(f, e)`	ajoute `e` à l'arrière de `f`
`retirer(p)`	retire et renvoie l'élément à l'avant de `f` si `f` n'est pas vide, et lève une exception sinon

Si par exemple on exécute les instructions de gauche, on obtient la situation représentée à droite :

```
f = creer_file()
ajouter(f, 1)
ajouter(f, 2)
ajouter(f, 3)
e = retirer(f)
```



b. Réalisation à l'aide d'une liste chaînée

La structure de liste chaînée donne également une manière de réaliser une file. La différence avec la réalisation d'une pile est de pouvoir accéder à la dernière cellule de la liste. Il est alors intéressant de conserver dans notre structure un attribut permettant d'accéder directement à cette dernière cellule. On peut construire une classe `File` avec deux attributs, l'un appelé `tete` et l'autre appelé `queue`, et désignant respectivement la première cellule et la dernière cellule de la liste chaînée.

```
class File:
  '''structure de file'''
  def __init__(self):
    self.tete = None
    self.queue = None
```

Pour tester si la file est vide, il suffit de tester si un de ces deux attributs est `None`.

```
  def est_vide(self):
    return self.tete is None
```

Pour ajouter un élément en queue de file, il faut créer une nouvelle cellule qui n'a pas de suivant.

```
  def ajouter(self, e):
    c = Cellule(e, None)
```

Cette cellule devient alors la suivante de la queue actuelle et devient la nouvelle queue.

```
    self.queue.suivante = c
    self.queue = c
```

On a cependant le besoin de traiter le cas particulier où la file est vide et dans ce cas la cellule devient l'unique cellule de la liste et donc celle de tête.

```
  def ajouter(self, e):
    c = Cellule(e, None)
    if self.est_vide():
      self.tete = c
    else:
      self.queue.suivante = c
    self.queue = c
```

Finalement pour retirer un élément, on procède de la même manière que pour une pile, il suffit de penser à redéfinir l'attribut `queue` à `None` lorsque l'opération vide la file.
On obtient alors le programme suivant :

```
class File:
  '''structure de file'''
  def __init__(self):
    self.tete = None
    self.queue = None

  def est_vide(self):
    return self.tete is None

  def ajouter(self, e):
    c = Cellule(e, None)
    if self.est_vide():
      self.tete = c
    else:
```

```
      self.queue.suivante = c
    self.queue = c

  def retirer(self):
    if self.est_vide():
      raise IndexError('retirer sur une file vide')
    else:
      v = self.tete.valeur
      self.tete = self.tete.suivante
      if self.tete is None:
        self.queue = None
      return v
```

On peut alors définir les fonctions de l'interface donnée plus haut :

```
def creer_file():
  return File()

def est_vide(f):
  return f.est_vide()

def ajouter(f, e):
  return f.ajouter(e)

def retirer(f):
  return f.retirer()
```

5. Dictionnaires

Cette section est un rappel de première.
Un dictionnaire Python est une structure de données permettant de stocker des informations sous la forme de couple « clef : valeur ». On accède alors à une valeur du dictionnaire non plus à l'aide d'un indice mais via sa clef.
Voici un premier exemple :

```
>>> voiture = { "marque": "Ford", "modèle": "Mustang", "année": 1964}
>>> voiture["modèle"]
Mustang
```

Dans cet exemple le dictionnaire s'appelle `voiture` et les clefs sont `"marque"`, `"modèle"` et `"année"`. La valeur associée à `"marque"` est `"Ford"`, celle associée à `"modèle"` est `"Mustang"` et celle associée à `année` est `1964`.

a. Créer un dictionnaire

En Python, un dictionnaire vide peut être créé à l'aide des accolades `{}`.

```
>>> mon_dict_vide = {}
>>> mon_dict_vide
{}
```

Remarque On peut également écrire `mon_dict_vide = dict()`.

On peut aussi créer un dictionnaire contenant une séquence de couple « clef : valeur » en plaçant ces éléments entre accolades `{}`, séparés par une virgule.

```
>>> voiture = { "marque": "Ford", "modèle": "Mustang", "année": 1964}
```

Les dictionnaires ne peuvent pas avoir deux éléments avec la même clef :

```
>>> voiture = {"marque": "Ford", "modèle": "Mustang", "année": 1964,
↪  "année": 2021}
>>> voiture
{"marque": "Ford", "modèle": "Mustang", "année": 2021}
```

b. Clef et valeur

Dans l'exemple `voiture` précédent les clefs sont `marque`, `modèle` et `année` et les valeurs sont `Ford`, `Mustang` et `1964`.
Pour obtenir la valeur associé à une clef, il suffit d'utiliser la notation avec crochet :

```
>>> voiture["modèle"]
Ford
```

Remarque On peut également utiliser la méthode `get`, `voiture.get("modèle")`.

Pour vérifier la présence d'une clef dans un dictionnaire, on utilise la syntaxe suivante :

```
>>> "année" in voiture
True
```

Remarque On peut aussi utiliser la méthode `has_key`, `voiture.has_key("année")`.

c. Modifier un dictionnaire

Pour ajouter une entrée dans un dictionnaire, il faut indiquer une clef et une valeur :

```
>>> voiture["couleur"] = "rouge"
>>> voiture
{ "marque": "Ford", "modèle": "Mustang", "année": 1964, "couleur":
↪  "rouge"}
```

Pour supprimer une entrée, on utilise la fonction `del` :

```
>>> del voiture["couleur"]
```

d. Parcourir un dictionnaire

La méthode `keys()` nous permet de parcourir l'ensemble des clefs d'un dictionnaire :

```
>>> for k in voiture.keys():
...     print(k)
marque
modèle
année
```

Remarque On peut simplement écrire `for k in voiture`.

La méthode `values()` nous permet de parcourir l'ensemble des valeurs d'un dictionnaire :

```
>>> for v in voiture.values():
...     print(v)
Ford
Mustang
1964
```

Pour parcourir à la fois les clefs et les valeurs, on utilise la méthode `items()` :

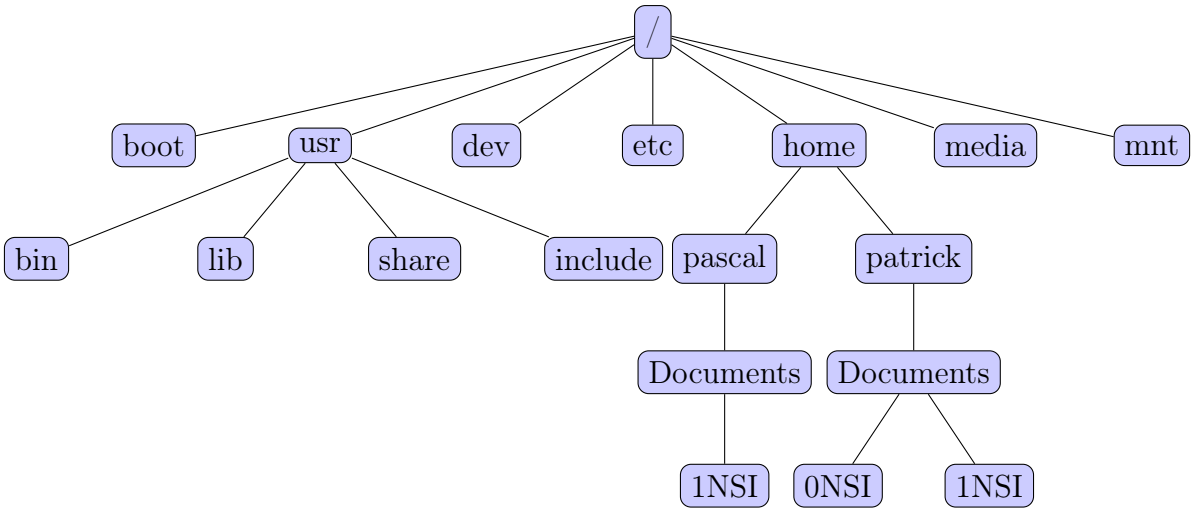
```
>>> for k, v in voiture.items():
...     print(k, v)
marque Ford
modèle Mustang
année 1964
```

II. Arbres

1. Structures arborescentes

a. Introduction

Dans un chapitre précédent, nous avons exploré une première structure chaîné, la liste chaînée. Les structures arborescentes sont une autre famille de structures chaînés très présentes en informatique dont vous avez déjà rencontré un exemple l'année dernière : l'arborescence des fichiers d'un ordinateur.

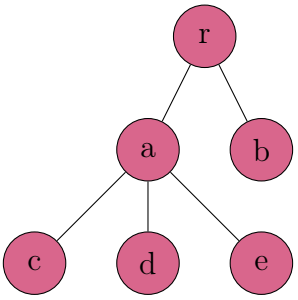


L'organisation sous forme d'arbre des fichiers d'un ordinateur permet notamment d'accéder en un petit nombre d'étape à n'importe quel fichier choisi parmi des dizaines de milliers, *pour peu qu'on aille dans la bonne direction.*

b. Définitions et vocabulaire

Les éléments d'un arbre sont appelés des **nœuds**. Le nœud au sommet de l'arbre est appelé **racine**. Un nœud peut avoir un ou plusieurs **fils** qui sont également des nœuds. Un nœud sans fils est une **feuille** (ou nœud externe), les autres nœuds sont des **nœuds internes**.
La **taille** d'un arbre est le nombre de nœuds qui le composent. La **hauteur** d'un arbre est le plus grand nombre de nœuds rencontrés en descendant du sommet jusqu'une feuille.

Exemple
On considère l'arbre ci-contre.



- Les nœuds sont `r`, `a`, `b`, `c`, `d` et `e`.
- La racine est le nœud `r`.
- Le nœud `r` a pour fils les nœuds `a` et `b`. Le nœud `a` a pour fils les nœuds `c`, `d` et `e`. Le nœud `b` n'a pas de fils.
- Les feuilles sont les nœuds `b`, `c`, `d` et `e`.
- La taille de l'arbre est 6.
- La hauteur de l'arbre est 3.

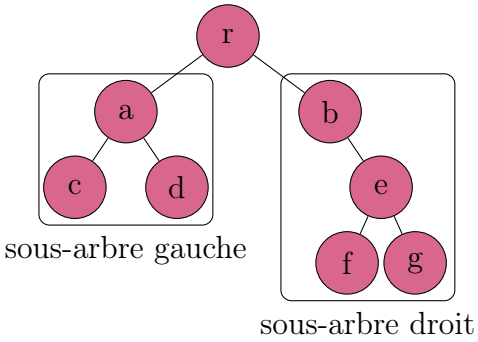
Remarque La hauteur d'un arbre vide est 0.

2. Arbres binaires

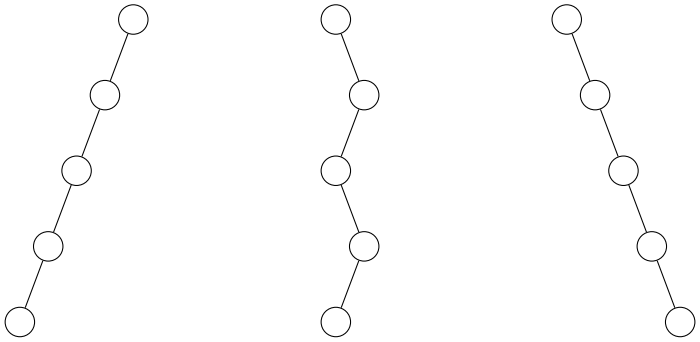
a. Définitions et propriétés

Un **arbre binaire** est un arbre dont chaque nœud possède au plus 2 fils généralement ordonnés, le fils gauche et le fils droit, chacun respectivement racine du **sous-arbre gauche** et du **sous-arbre droit**.
Soit N la taille d'un arbre binaire et h sa hauteur, alors :

$$h \leq N \leq 2^h - 1$$

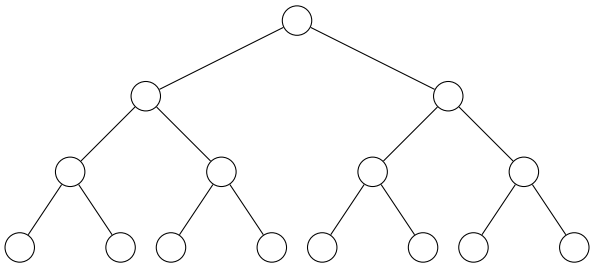


A gauche, l'égalité est atteinte avec des arbres formés d'un seul nœud à chaque niveau :



Dans cette situation, la hauteur est égale à la taille et l'arbre obtenu n'est pas différent d'une liste chaînée.

Pour l'autre côté, l'égalité est atteinte pour un arbre binaire **parfait** où toutes les feuilles ont exactement la même profondeur :



Le nombre de nœuds est $2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{h-1} = 2^h - 1$.

b. Représentation en Python

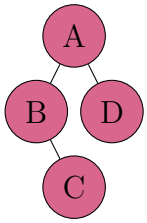
Il y a de nombreuses façon de représenter un arbre binaire en Python. Comme pour les listes chaînées, une façon habituelle consiste à représenter chaque nœud par objet d'une classe qui ici s'appellera `Noeud`.

```
class Noeud:
  '''un nœud d'un arbre binaire'''
  def __init__(self, v, g, d):
    self.valeur = v
    self.gauche = g
    self.droit = d
```

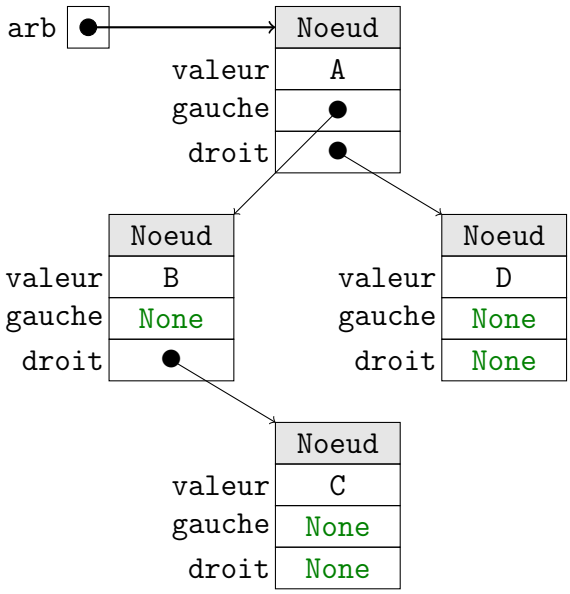
La classe contient trois attributs : un attribut `valeur` pour la valeur contenue par le nœud, un attribut `gauche` pour pour le sous arbre gauche et un attribut `droit` pour le sous arbre droit.

Pour construire un arbre, il suffit alors d'appliquer le constructeur de la classe `Noeud` autant de fois que nécessaire.

```
arb = Noeud('A',
            Noeud('B', None, Noeud('C', None, None)),
            Noeud('D', None, None))
```



On a ainsi créé quatre objets de la classe `Noeud` qui sont illustrés dans la figure ci-dessous.



c. Algorithmes

Taille et hauteur d'un arbre binaire

La définition d'un arbre binaire étant récursive, il est naturel d'utiliser un algorithme récursif pour calculer sa taille.

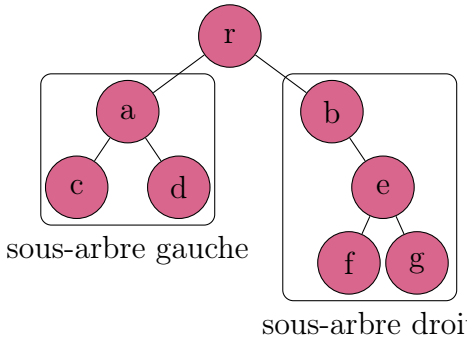
```
def taille(arb):
  '''calcul la taille d'un arbre binaire'''
```

Le cas de base (condition d'arrêt) est celui d'un arbre vide dont la taille est 0.

```
  if arb is None:
    return 0
```

La taille d'un arbre non vide, est la somme des tailles des deux sous-arbres gauche et droit plus 1 pour la racine.

```
  else:
    return 1 + taille(arb.gauche) + taille(arb.droite)
```



`taille`(arbre) = 1+`taille`(sous-arbre gauche)+`taille`(sous-arbre droit)

Ce qui donne la fonction `taille` suivante :

```
def taille(arb):
  '''calcul la taille d'un arbre binaire'''
  if arb is None:
    return 0
  else:
    return 1 + taille(arb.gauche) + taille(arb.droite)
```

De la même manière, si arbre est vide, sa hauteur est 0, et dans le cas contraire sa hauteur est 1 plus le maximum des hauteurs des sous-arbres droit et gauche. Ce qui donne la fonction `hauteur` suivante :

```
def hauteur(arb):
  '''calcul la hauteur d'un arbre binaire'''
  if arb is None:
    return 0
  else:
    return 1 + max(hauteur(arb.gauche),hauteur(arb.droite))
```

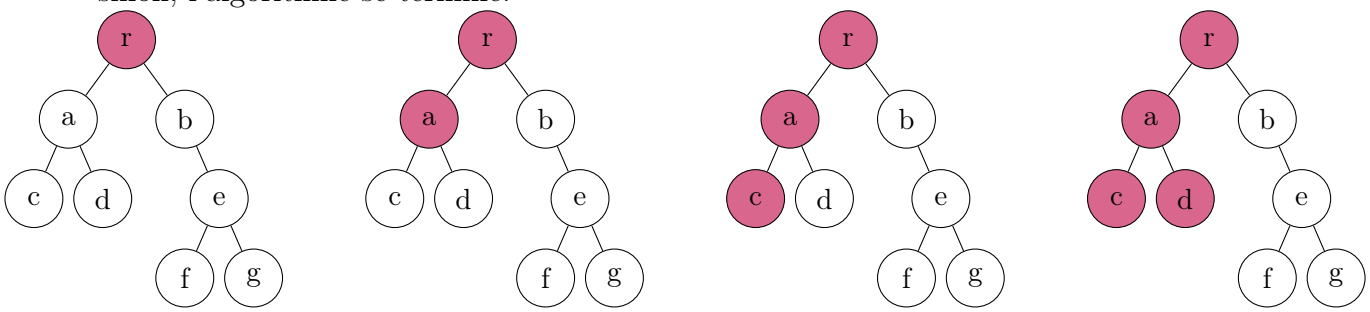
Parcours d'un arbre binaire

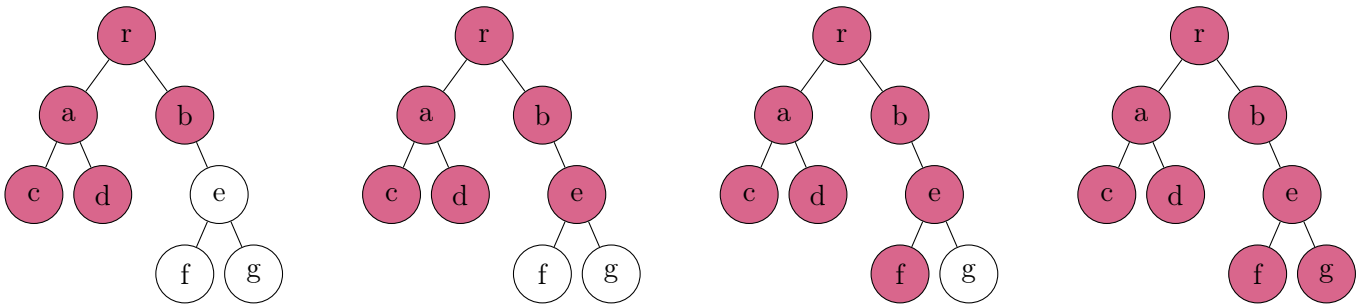
Les fonctions `hauteur` et `taille` parcourent tous les nœuds de l'arbre ; l'ordre dans lequel l'arbre est parcouru n'a pas d'importance. Si l'on veut afficher les valeurs des nœuds, l'ordre dans lequel on rencontre les nœuds devient important.

Parcours en profondeur d'abord

Dans le cas où il est **parcouru en profondeur d'abord**, on explore complètement une branche avant d'explorer la branche voisine à l'aide de l'algorithme suivant :

- si l'arbre est non vide, on parcours de manière récursive son sous-arbre gauche puis son sous-arbre droit ;
- sinon, l'algorithme se termine.





On distingue trois types de parcours selon le moment où est traitée (affichée) la racine d'un sous-arbre :

- Dans un **parcours préfixe**, la racine est traité avant ses deux sous-arbres ;
- Dans un **parcours infixe**, a racine est traité entre ses deux sous-arbres ;
- Dans un **parcours postfixe** (on peut lire aussi parfois **suffixe**), la racine est traité après ses deux sous-arbres.

Voici leur définition en Python :

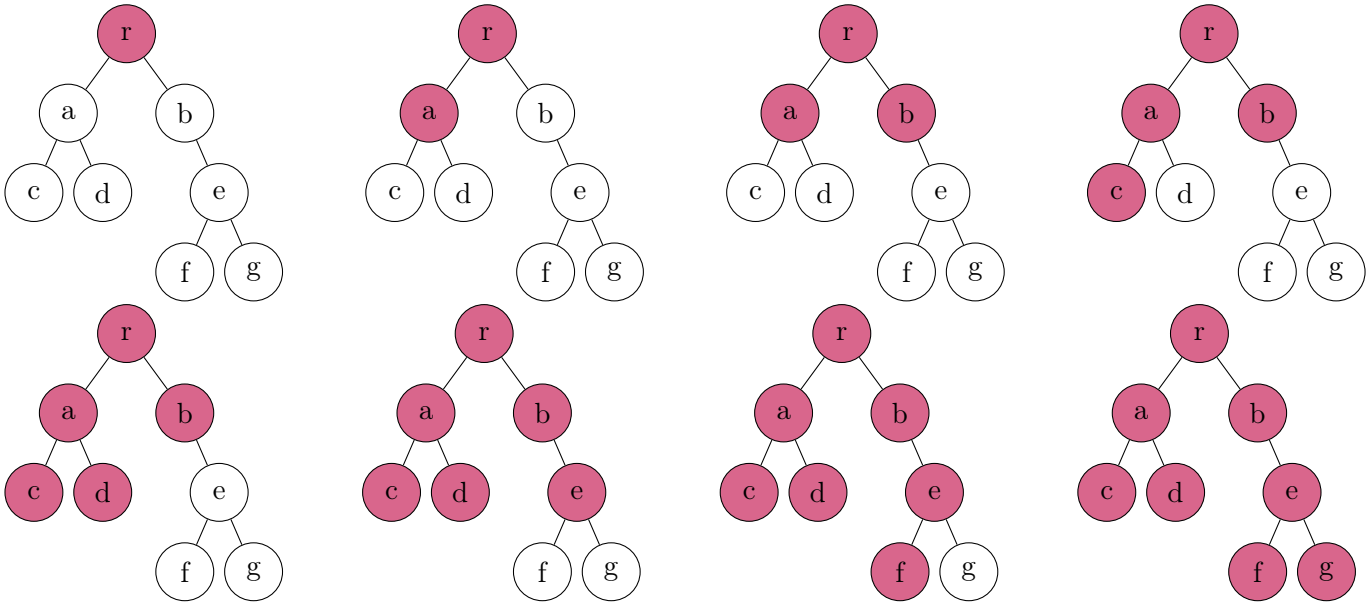
```
def parcours_prefixe(arb):
  '''affiche les nœuds de arb dans un parcours préfixe'''
  if arb is not None:
    print(arb.valeur, end=" ")
    parcours_prefixe(arb.gauche)
    parcours_prefixe(arb.droit)

def parcours_infixe(arb):
  '''affiche les nœuds de arb dans un parcours infixe'''
  if arb is not None:
    parcours_infixe(arb.gauche)
    print(arb.valeur, end=" ")
    parcours_infixe(arb.droit)

def parcours_postfixe(arb):
  '''affiche les nœuds de arb dans un parcours postfixe'''
  if arb is not None:
    parcours_postfixe(arb.gauche)
    parcours_postfixe(arb.droit)
    print(arb.valeur, end=" ")
```

Sur l'exemple d'illustration, on obtient les résultats suivants :

```
>>> parcours_prefixe(arb)
R A C D B E F G
>>> parcours_infixe(arb)
C A D R B F E G
>>> parcours_postfixe(arb)
C D A F G E B R
```

Parcours en largeur d'abord

Dans le cas d'un **parcours en largeur d'abord**, on parcourt l'arbre étage par étage. On commence par la racine, puis les deux racines de chacun des sous-arbres, puis les quatre racines de chacun des quatre sous-arbres et ainsi de suite.

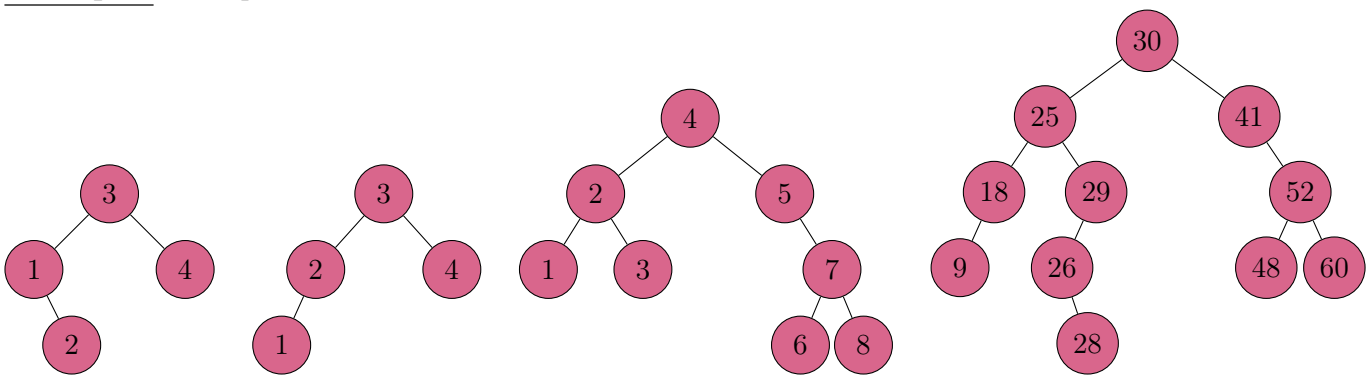


3. Arbres binaires de recherche

a. Définition et exemples

Un **arbre binaire de recherche** est un arbre binaire tel que, pour chaque nœud, toutes les valeurs du sous-arbre gauche sont plus petites que la valeur du nœud et toutes les valeurs du sous-arbre droit sont plus grandes que la valeur du nœud.

Exemples Les quatre arbres suivants sont des arbres binaires de recherche.



b. Représentation en Python

La représentation en Python d'un arbre binaire de recherche est réalisée, comme pour un arbre binaire du chapitre précédent, à l'aide de la classe `Noeud`.

c. Algorithmes

Les fonctions `taille` et `hauteur` restent valable ainsi que les différents parcours. En particulier le parcours infixe permet l'affichage des valeurs d'un arbre binaire de recherche par ordre croissant.
Les fonctions spécifiques des arbres binaires de recherche que nous allons voir vont supposer (en entrée) et garantir (en sortie) leurs propriétés.

Recherche d'un élément

L'intérêt d'un arbre binaire de recherche est l'efficacité de la recherche d'un élément.

```
def appartient(arb, v):
  '''détermine si la valeur v appartient à l'ABR arb'''
```

Une première condition d'arrêt est le cas où l'arbre est vide.

```
  if arb is None:
    return False
```

Si la valeur de la racine est la valeur recherchée `v`, on est également dans un cas de base.

```
  elif arb.valeur == v:
    return True
```

Sinon, soit `v` est plus petit que la valeur de la racine et alors, si `v` appartient à l'arbre, `v` appartient au sous-arbre gauche.

```
  elif v < arb.valeur:
    return appartient(arb.gauche, v)
```

Soit `v` est plus grand que la valeur de la racine et alors, si `v` appartient à l'arbre, `v` appartient au sous-arbre droit.

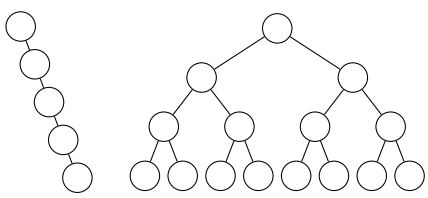
```
  else:
    return appartient(arb.droite, v)
```

On obtient le programme suivant :

```
def appartient(arb, v):
  '''détermine si la valeur v appartient à l'ABR arb'''
  if arb is None:
    return False
  elif arb.valeur == v:
    return True
  elif v < arb.valeur:
    return appartient(arb.gauche, v)
  else:
    return appartient(arb.droite, v)
```

Cet algorithme réalise un nombre d'opérations au plus égal à sa hauteur.
Cette dernière dépend de la forme de l'arbre.

Dans le pire des cas, l'arbre est filiforme et la hauteur est égale au nombre de nœuds. La recherche est alors de complexité linéaire car susceptible de parcourir tous les nœuds de l'arbre comme pour une recherche dans une liste chaînée.
Dans le meilleur des cas, l'arbre est parfait. On élimine alors la moitié des éléments à chaque étape comme dans une recherche dichotomique qui est de complexité logarithmique.



Ajout d'un élément

Pour construire un arbre binaire de recherche, nous allons utiliser une fonction `ajout`.

```
def ajout(arb, v):
  '''ajoute l'élément v à l'ABR arb et retourne un nouvel ABR'''
```

Le cas de base est l'ajout d'un élément dans un arbre vide.

```
  if arb is None:
    return Noeud(v, None, None)
```

Si la valeur à ajouter `v` est inférieure à la valeur de la racine, on ajoute `v` au sous-arbre gauche. La valeur de la racine reste inchangée, ainsi que le sous-arbre droit.

```
   elif v < arb.valeur:
    return Noeud(arb.valeur, ajout(arb.gauche, v), arb.droit)
```

Sinon, `v` est supérieure à la valeur de la racine. On ajoute alors `v` au sous-arbre droit, la valeur de la racine reste inchangée, ainsi que le sous-arbre gauche.

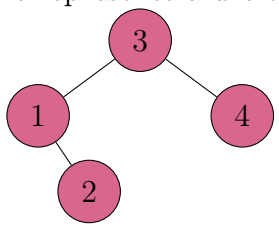
```
  else :
    return Noeud(arb.valeur, arb.gauche, ajout(arb.droit, v))
```

On obtient le programme suivant :

```
def ajout(arb, v):
  '''ajoute l'élément v à l'ABR arb et retourne un nouvel ABR'''
  if arb is None:
    return Noeud(v, None, None)
  elif v < arb.valeur:
    return Noeud(arb.valeur, ajout(arb.gauche, v), arb.droit)
  else :
    return Noeud(arb.valeur, arb.gauche, ajout(arb.droit, v))
```

Si par exemple on exécute les cinq instructions de gauche, on obtient l'arbre représenté à droite :

```
arb = None
arb = ajoute(arb, 3)
arb = ajoute(arb, 1)
arb = ajoute(arb, 2)
arb = ajoute(arb, 4)
```



d. Encapsulation

Comme pour les listes chaînées, on peut encapsuler nos arbres binaires de recherche dans une classe ici appelé `ABR`.

```
class ABR:
  '''un arbre binaire de recherche'''
  def __init__(self):
    self.racine = None

  def ajouter(self, v):
    self.racine = ajout(self.racine, v)

  def contient(self, v):
    return appartient(self.racine, v)
```

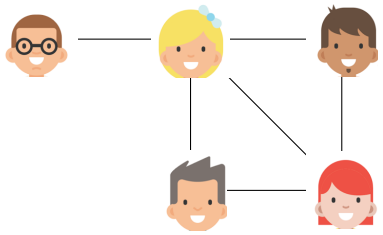
Cette classe comporte un unique attribut `racine` initialisé à `None` et deux méthodes `ajouter` et `contient` qui correspondent aux fonctions `ajout` et `appartient`.
Toute autre fonction comme `taille`, `hauteur`, `parcours_infixe`, etc. peut également être ajoutée à cette classe.

III. Graphes

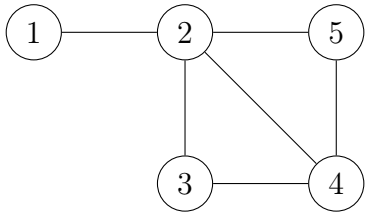
1. Définitions

Les graphes sont une manière de représenter un réseau de connexions entre des éléments.

Exemple On peut considérer un groupe de personnes et s'intéresser à un type de relation entre eux (amitié, familiale, etc.) :



En réduisant l'information au minimum, on obtient la représentation d'un graphe :



Ce graphe a :

- 5 sommets (numérotés 1,2,3,4 et 5)
- 6 arêtes (les liaisons entre les sommets)

Le graphe (G) est le couple formé par l'ensemble de ses sommets (S) et celui de ses arêtes (A) , pour lesquels on peut utiliser les notations suivantes :

- $G = (S,A)$
- $S = \{1,2,3,4,5\}$
- $A = \{\{1,2\},\{2,3\},\{2,4\},\{2,5\},\{3,4\},\{4,5\}\}$

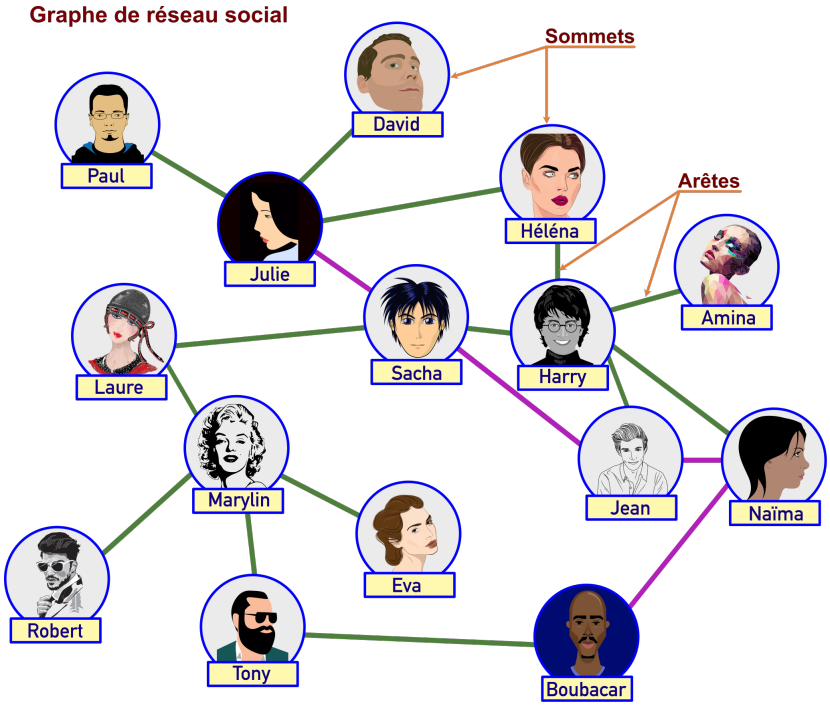
Ainsi :

Définition Un **graphe** G est la donnée d'un couple $G = (S,A)$ tel que :

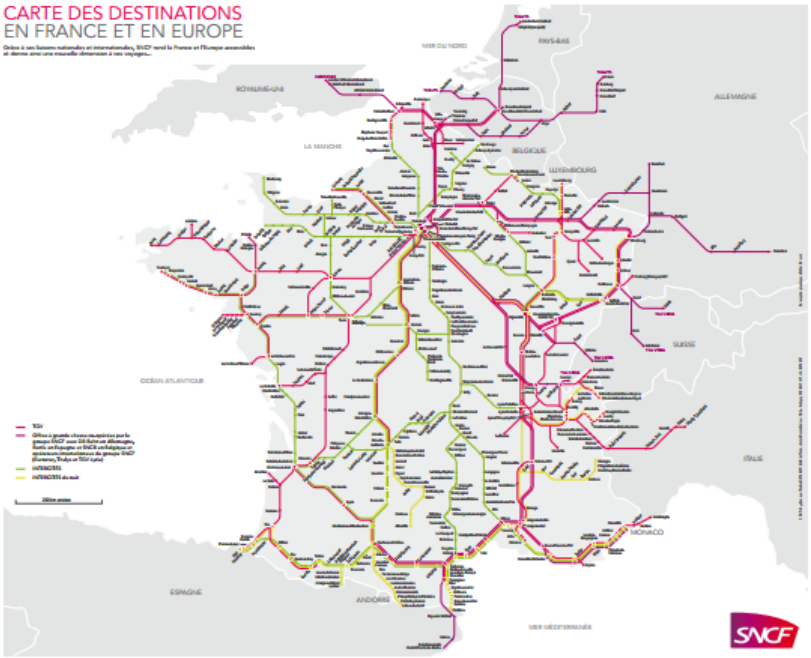
- S est un ensemble fini de **sommets** ;
- A est un ensemble de couples non ordonnés de sommets $\{s_i,s_j\} \in S^2$, les **arêtes**.

Exemples Voici plusieurs situations qui se prêtent à une modélisation par des graphes :

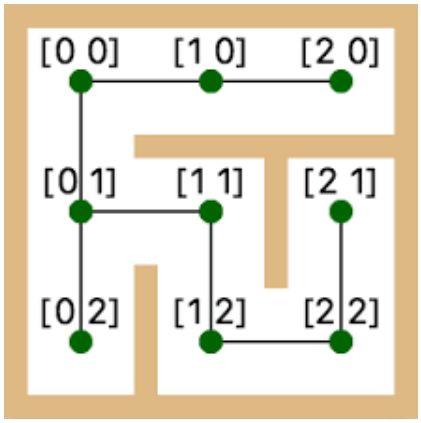
- Les réseaux sociaux : les sommets sont les personnes, deux personnes sont adjacentes dans ce graphe lorsqu'elles sont "amies".



- Les réseaux routier, ferroviaires : chaque gare (ville) est un sommet, les voies entre deux gares sont les arêtes.



- Un labyrinthe constitué de pièces reliées entre elles par des couloirs peut être vu comme un graphe.



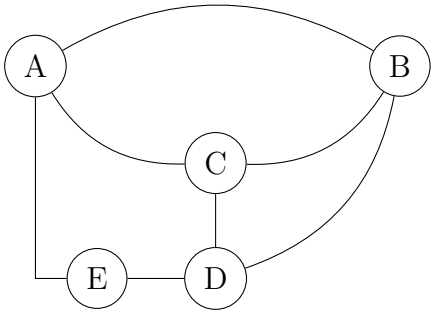
Définitions

- L'**ordre** d'un graphe est le nombre de ses sommets.
- Le **degré** d'un sommet est le nombre d'arêtes lié à ce sommet.
- Deux sommets s_i et s_j sont **adjacents** (ou **voisins**) si le graphe contient l'arête $\{s_i,s_j\}$.
- Un **chemin** d'un sommet s_1 à un sommet s_2 est une séquence finie de sommets reliés deux à deux par des arêtes et menant de s_1 à s_2.
  On peut l'écrire en notant la succession des sommets en les séparant par des tirets ou des flèches.
- Un chemin est dit **simple** s'il n'emprunte pas deux fois la même arête.
  Un chemin simple reliant un sommet à lui-même est appelé un **cycle**.

Exemple

Dans le graphe ci-contre,

- Il y a 7 arêtes ;
- L'ordre est 5 ;
- Le degré du sommet B est 3.
- Les sommets A et E sont adjacents, mais pas A et D ;
- Le chemin $A-C-D-B$ est simple,
  mais pas $B-C-B-A$;
- Le chemin $A-C-B-A$ est un cycle ;



Définition (Graphe orienté)

Dans certaines situations, on souhaite que les arêtes aient un sens de parcours, que l'on montre avec des flèches. On parle alors de **graphe orienté** et les arêtes orientées entre deux sommets sont alors appelées des **arcs**.

La définition rigoureuse d'un graphe orienté est la suivante :

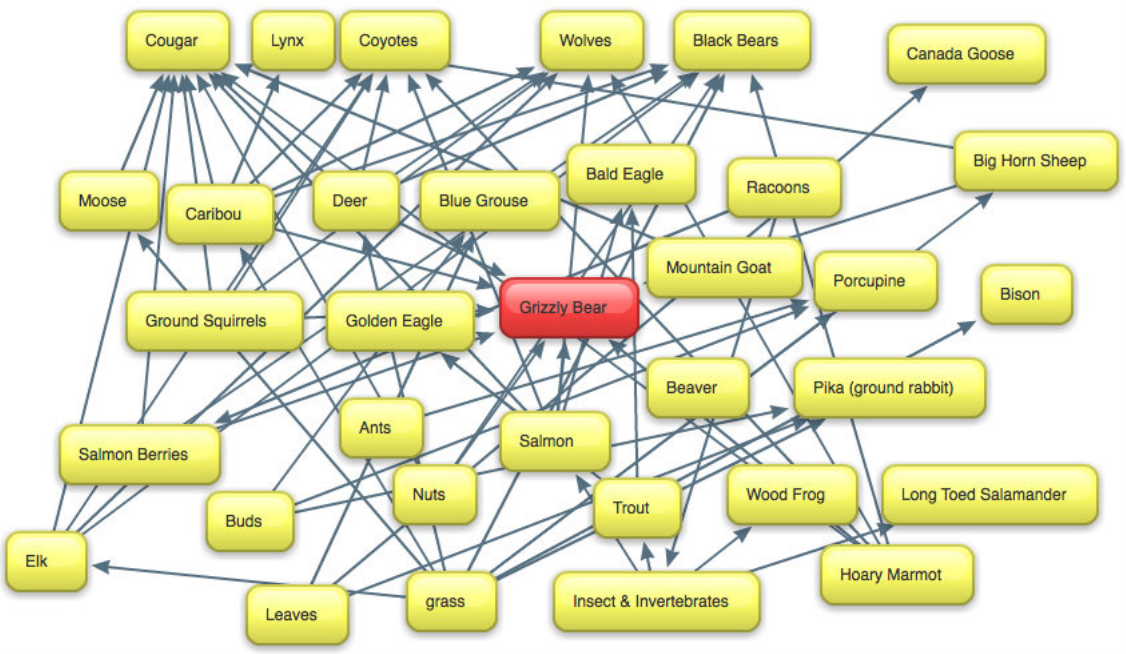
Un **graphe orienté** G est la donnée d'un couple $G=(S,A)$ tel que :

- S est un ensemble fini de sommets ;
- A est un ensemble de couples **ordonnés** de sommets $(s_i,s_j) \in S^2$, les **arcs**.

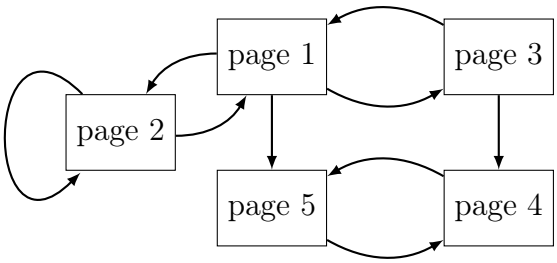
Les autres définitions sur les graphes non orientés sont valables pour les graphes orientés.

Exemples Voici deux cas pouvant être représentés par des graphes orientés :

- Une chaîne alimentaire : chaque sommet est une espèce animale et chaque arc d'origine s_i et d'extrémité s_j correspond à une espèce s_j qui se nourrit d'une espèce s_i.



- Le web : chaque sommet du graphe représente une page web, chaque arc entre deux sommets est la représentation d'un lien hypertexte présent dans la page de départ vers la page d'arrivée.

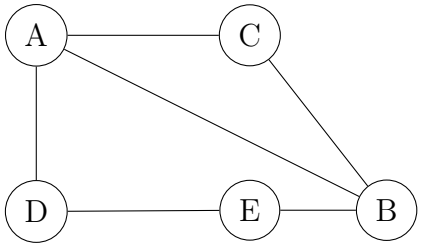


2. Deux représentations possibles

a. Par matrice d'adjacence

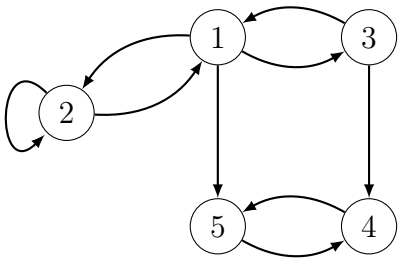
Définition La matrice d'adjacence d'un graphe d'ordre n est la matrice carrée M de dimension $n \times n$ telle que l'élément m_{ij} vaut le nombre d'arêtes reliant le sommet i au sommet j.
En particulier, si i et j ne sont pas adjacents, alors $m_{ij} = 0$.

Exemples On donne ci-dessous un graphe non orienté et sa matrice d'adjacence associée, en considérant les sommets rangés par ordre alphabétique :



$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

La matrice d'adjacence d'un graphe non orienté est symétrique (par rapport à la diagonale principale, celle qui va de en haut à gauche vers en bas à droite).
Pour un graphe orienté, ce n'est généralement plus le cas :



$$M = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

La représentation en Python d'une matrice d'adjacence peut être sous forme de liste de listes, chaque liste étant une ligne.

⚠ Les indices commencent à `0`, comme toujours en Python, ce qui implique que les noms des sommets doivent être (re)numérotés dans l'ordre de `0` à `n-1` si `n` est l'ordre du graphe.

Exemple Pour le graphe non orienté des exemples ci-dessus, on obtient alors :

```
M = [[0,1,1,1,0], [1,0,1,0,1], [1,1,0,0,0], [1,0,0,0,1], [0,1,0,1,0]]
```

On pourra se donner comme exercice de définir la matrice d'adjacence pour le graphe orienté.

On peut s'aider de fonctions d'aide à la création de graphe. Ainsi :

```
def matrice_adj(n):
        '''Crée une matrice d'adjacence d'un graphe à n sommets'''
        return [[0 for j in range(n)] for i in range(n)]

def ajouter_arete(adj,s1,s2):
        '''Ajoute une arête allant du sommet s1 vers
        le sommet s2 du graphe représenté par la
        matrice adj'''
        adj[s1][s2] = 1
        adj[s2][s1] = 1

def ajouter_arc(adj,s1,s2):
        '''Ajoute un arc allant du sommet s1 vers
        le sommet s2 du graphe représenté par la
        matrice adj'''
        adj[s1][s2] = 1
```

On remarque que dans le cas d'un graphe non orienté, avec les arêtes, il faut respecter la symétrie.

Exemple Ainsi, si l'on souhaite créer le graphe non orienté des exemples précédents, on va devoir exécuter les instructions suivantes :

```
M=matrice_adj(5)
ajouter_arete(M,0,1)
ajouter_arete(M,0,2)
ajouter_arete(M,0,3)
ajouter_arete(M,1,2)
ajouter_arete(M,1,4)
ajouter_arete(M,3,4)
```

À titre d'exercice on pourra faire de même pour le graphe orienté.

On peut vouloir ajouter d'autres fonctions, comme :

- `arete(adj,s1,s2)` (pour les graphes non orientés) qui retourne `True` si une arête part du sommet `s1` vers le sommet `s2` et `False` sinon.
- `arc(adj,s1,s2)` (pour les graphes orientés) qui retourne `True` si un arc part du sommet `s1` vers le sommet `s2` et `False` sinon.
- `voisins(adj,s)` qui retourne la liste des voisins (sommets adjacents) du sommet `s` dans le graphe dont la matrice d'adjacence est `adj`.

On peut ensuite chercher à encapsuler ces fonctions dans une classe `Graphe` :

```
class Graphe:
        '''un graphe d'ordre n donné représenté par une matrice
         d'adjacence, où les sommets sont numérotés
         0, 1, ..., n-1'''
        def __init__(self, n):
                self.n = n
                self.adj = [[0 for j in range(n)] for i in range(n)]
```

Tout ceci fera l'objet d'un exercice.

b. Par liste d'adjacence

Plutôt qu'à l'aide de sa matrice d'adjacence, on peut vouloir modéliser un graphe par les listes d'adjacence de chacun de ses sommets.
Autrement dit, pour un graphe d'ordre `n`, on donne un tableau `T` de `n` listes. Pour chaque sommet `i`, `T[i]` est la liste de tous les sommets `j` tels que (`i`,`j`) est un arc (ou une arête) du graphe.
On peut donner ce tableau sous forme d'un dictionnaire, ce qui permet de conserver les noms des sommets.

Exemple Pour le graphe non orienté déjà étudié, on a :

```
T = {'A':['B','C','D'], 'B':['A','C','E'], \
     'C':['A','B'], 'D':['A','E'], 'E':['B','D']}
```

À titre d'exemple on pourra faire de même pour le graphe orienté (dont les noms des sommets sont ici des entiers de 1 à 5).

La représentation du graphe en Python est alors un dictionnaire, au départ vide. On peut définir une fonction qui ajoute un sommet, puis une fonction qui ajoute une arête (ou un arc), etc. :

```
def ajouter_sommet(adj, s):
        '''Ajoute le sommet s à la liste d'adjacence adj'''
        if s not in adj:
                adj[s] = []

def ajouter_arete(adj, s1, s2):
        '''Ajoute l'arc allant du sommet s1 au sommet s2
        à la liste d'adjacence adj'''
        if s1 in adj and s2 in adj:
                adj[s1].append(s2)
                adj[s2].append(s1)
```

Exemple Ainsi, la création du graphe non orienté se fait de la manière suivante :

```
adj = {}
ajouter_sommet(adj,'A')
ajouter_sommet(adj,'B')
ajouter_sommet(adj,'C')
ajouter_sommet(adj,'D')
ajouter_sommet(adj,'E')
ajouter_arete(adj,'A','B')
ajouter_arete(adj,'A','C')
ajouter_arete(adj,'A','D')
ajouter_arete(adj,'B','C')
ajouter_arete(adj,'B','E')
ajouter_arete(adj,'D','E')
```

À titre d'exercice, on pourra faire de même pour le graphe orienté.

De même que pour la représentation précédente, on peut encapsuler ces fonctions dans une classe. On conserve toujours la possibilité d'obtenir l'une ou l'autre des représentation à partir de l'autre, autrement dit la matrice d'adjacence à partir des listes d'adjacences ou les listes d'adjacences à partir de la matrice d'adjacence. Des méthodes dans la classe `Graphe` peuvent être ajoutées qui permettent d'obtenir l'une ou l'autre des représentations.

3. Algorithmes : Parcours dans un graphe

a. Parcours en profondeur

Comme il n'y a pas de racine dans un graphe, on choisit arbitrairement un sommet de départ. Définition récursive :

```
def parcours_profondeur(g, s):
    '''parcours en profondeur depuis le sommet s'''

    vus = []

    def parcours_rec(g, s):
        '''Utilisation d'une fonction récursive'''
        if s not in vus:
            vus.append(s) # préfixe
            for v in g.voisins(s):
                parcours_rec(g, v)

    parcours_rec(g, s)
    return vus
```

Définition itérative, utilisant les piles :

```
def parcours_profondeur_it(g, s):
    '''parcours en profondeur depuis le sommet s'''
    vus = []
    vus.append(s)
    pile = Pile()
    pile.empiler(s)
    while not pile.est_vide():
        s = pile.consulter()
        # Liste des voisins qui n'ont pas encore été visités
        non_vus = [voisin for voisin in g.voisins(s)
                   if voisin not in vus]
        # Si un voisin n'a pas encore été visité,
        # il est empilé et on le visite
        if non_vus:
            voisin = non_vus[0]
            vus.append(voisin)
            pile.empiler(voisin)
        # Si tous les voisins sont visités, le sommet est dépilé
        else:
            pile.depiler()
    return vus
```

b. Parcours en largeur

Ce parcours utilise les files :

```
def parcours_largeur(g, s):
    vus = []
    vus.append(s)
    file = File()
    file.ajouter(s)
    while not file.est_vide() :
        u = file.consulter()
        for v in g.voisins(u):
            if v not in vus:
                vus.append(v)
                file.ajouter(v)
        file.retirer()
    return vus
```