Graphes et représentations

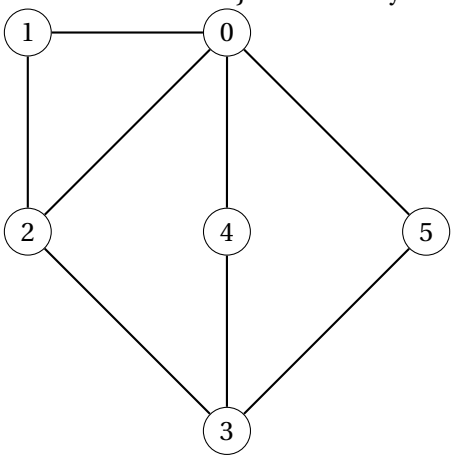
INFORMATIQUE COMMUNE - TP n° 2.3 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ représenter un graphe en machine par une liste d'adjacence ou une matrice d'adjacence
- ☞ transformer une représentation en une autre
- ☞ calculer les degrés des sommets d'un graphe
- ☞ transposer un graphe

A Représentation et transformation d'un graphe

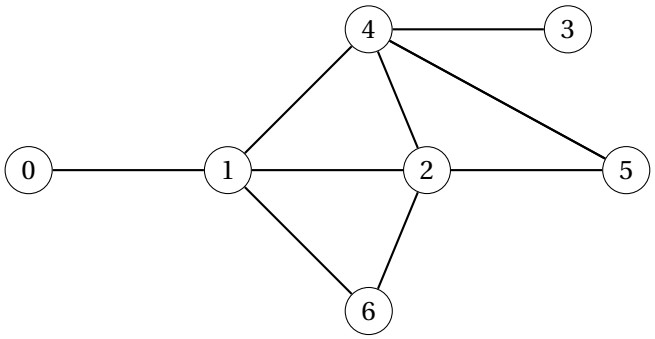
A1. Créer une liste d'adjacence en Python qui représente le graphe suivant :



Solution :

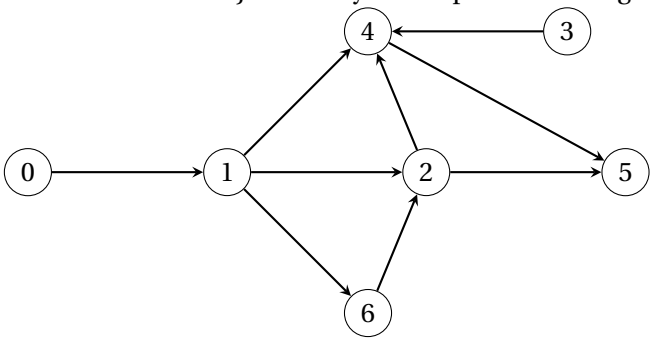
```
ga1 = [[1, 2, 4, 5], [0, 2], [0, 1, 3], [2, 4, 5], [0, 3], [0, 3]]
```

A2. Dessiner le graphe correspondant à la liste d'adjacence :
`[[1],[0,2,4,6],[1,4,5,6],[4],[1,2,3,5],[2,4],[1,2]]`



Solution :

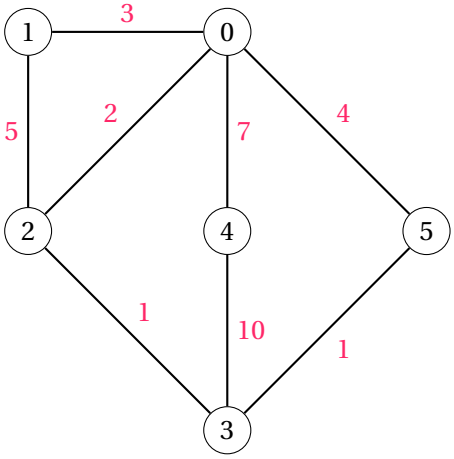
A3. Créer une liste d'adjacence Python représentant le graphe orienté suivant :



Solution :

```
go = [[1], [2, 4, 6], [4, 5], [4], [5], [], [2]]
```

A4. Créer une liste d'adjacence Python représentant le graphe pondéré suivant :



Solution :

```
gp = [[(1,3), (2,2), (4,7), (5,4)], [(0,3), (2,5)], [(0,2), (1,5), (3,1)],
    [(2,1), (4,10), (5,1)], [(0,7), (3,10)], [(0,4), (3,1)]]
```

A5. Modéliser les graphes des questions précédentes en utilisant le concept de matrice d'adjacence.

Solution :

```
import numpy as np

ma1 = np.array([[0,1,1,0,1,1],[1,0,1,0,0,0],[1,1,0,1,0,0],
                [0,0,1,0,1,1],[1,0,0,1,0,0],[1,0,0,1,0,0]])
print(ma1)
# [[0 1 1 0 1 1]
#  [1 0 1 0 0 0]
#  [1 1 0 1 0 0]
#  [0 0 1 0 1 1]
#  [1 0 0 1 0 0]
#  [1 0 0 1 0 0]]
ma2 = np.array
    ([[0,1,0,0,0,0,0],[1,0,1,0,1,0,1],[0,1,0,0,1,1,1],[0,0,0,0,1,0,0],
                [0,1,1,1,0,1,0],[0,0,1,0,1,0,0],[0,1,1,0,0,0,0]])
print(ma2)
# [[0 1 0 0 0 0 0]
#  [1 0 1 0 1 0 1]
#  [0 1 0 0 1 1 1]
#  [0 0 0 0 1 0 0]
#  [0 1 1 1 0 1 0]
#  [0 0 1 0 1 0 0]
#  [0 1 1 0 0 0 0]]

mo = np.array
    ([[0,1,0,0,0,0,0],[0,0,1,0,1,0,1],[0,0,0,0,1,1,0],[0,0,0,0,1,0,0],
                [0,0,0,0,0,1,0],[0,0,0,0,0,0,0],[0,0,1,0,0,0,0]])
print(mo)
# [[0 1 0 0 0 0 0]
#  [0 0 1 0 1 0 1]
#  [0 0 0 0 1 1 0]
#  [0 0 0 0 1 0 0]
#  [0 0 0 0 0 1 0]
#  [0 0 0 0 0 0 0]
#  [0 0 1 0 0 0 0]]
```

A6. Écrire une fonction de prototype `ladj_to_madj(g)` qui prend comme paramètre un graphe (non pondéré) sous la forme d'une liste d'adjacence et renvoie le même graphe sous la forme d'une matrice d'adjacence. Le type retourné est une liste de listes Python.

Solution :

```
def ladj_to_madj(g):
    n = len(g)
    m = [[0 for _ in range(n)] for _ in range(n)]
    # on peut créer m avec une double boucle !
    for i in range(n):
        for v in g[i]:
            m[i][v] = 1
    return m
```

A7. Écrire une fonction de prototype `ladj_to_madj_n(g)` qui prend comme paramètre un graphe (non pondéré) sous la forme d'une liste d'adjacence et renvoie le même graphe sous la forme d'une matrice d'adjacence. Le type retourné est un tableau Numpy.

Solution :

```
def ladj_to_madj_n(g):
    n = len(g)
    m = np.zeros((n,n))
    for i in range(n):
        for v in g[i]:
            m[i,v] = 1
    return m
```

A8. Écrire une fonction de prototype `madj_to_ladj(g)-> list[list[int]]` qui prend comme paramètre un graphe (non pondéré) sous la forme d'une matrice d'adjacence et renvoie le même graphe sous la forme d'une liste d'adjacence. La matrice d'entrée pourra indifféremment être donnée sous la forme d'un tableau Numpy ou d'une liste de liste. La matrice de sortie sera une liste de listes.

Solution :

```
# version qui marche pour un tab numpy et une liste Python
def madj_to_ladj(g):
    n = len(g)
    L = [[] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if g[i][j] != 0:
                L[i].append(j)
    return L
```

A9. Écrire une fonction de prototype `transpose_m(g)` qui prend en paramètre un graphe orienté sous la forme d'une matrice d'adjacence et qui renvoie le graphe transposé correspondant, c'est-à-dire le graphe dont les arcs sont dirigés dans le sens opposé. Quelle est la complexité de cette fonction ?

Solution : La complexité de cette fonction est en $O(n^2)$, si n est l'ordre du graphe.

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} c = O(n^2)$$

```
def transpose_m(g):
    n = len(g)
    tg = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        for j in range(n):
            tg[j][i] = g[i][j]
    return tg
```

A10. Écrire une fonction de prototype `transpose(g)` qui prend en paramètre un graphe orienté sous la forme d'une liste d'adjacence et qui renvoie le graphe transposé correspondant, c'est-à-dire le graphe dont les arcs sont dirigés dans le sens opposé. Quelle est la complexité de cette fonction ?

Solution :

```
def transpose(g):
    n = len(g)
    tg = [[] for _ in range(n)]
    for i in range(n):
        voisins = g[i]
        for v in voisins:
            tg[v].append(i)
    return tg
```

La complexité de cette fonction est en $O(n+m)$, si n est l'ordre du graphe et m le nombre d'arc du graphe. La boucle extérieure effectue autant d'itération que de sommets dans le graphe (n). La boucle intérieure effectue autant d'itérations que d'arêtes incidente au sommet i. De plus, le reste des opérations est de complexité constante. Ceci s'écrit :

$$C(n) = \sum_{i=0}^{n-1}\left(c + \sum_{v\in g[i]} c\right) = \sum_{i=0}^{n-1} c + \sum_{i=0}^{n-1}\sum_{v\in g[i]} c = nc + \sum_{a\in A} c = nc + mc = O(n+m)$$

où A est l'ensemble des arcs du graphe $G = (S, A)$ de cardinal $|A| = m$. Dans la deuxième double somme, (i, v) est un arc du graphe G. Cela signifie qu'on parcourt tous les sommets.

A11. Écrire une fonction de prototype `degrees(g)` qui renvoie la liste des degrés des sommets d'un graphe non orienté. Le paramètre est donné sous la forme d'une liste d'adjacence. Par exemple pour le graphe de la première question, la fonction renvoie : `[4, 2, 3, 3, 2, 2]`.

Solution :

```
def degrees(g):
    return [len(L) for L in g]
```

A12. Écrire une fonction de prototype `in_degrees(g)` qui renvoie la liste des degrés entrants des sommets d'un graphe orienté. Le paramètre est donné sous la forme d'une liste d'adjacence. Par exemple, pour le graphe orienté de la question 3, la fonction renvoie `[0, 1, 2, 0, 3, 2, 1]`.

Solution :

```
def in_degrees(g):
    return [len(L) for L in transpose(g)]
```

B Création d'un graphe à partir de données dans un fichier

On dispose d'un fichier qui contient des distances entre des villes de l'ouest de la France (le télécharger sur le site!). On souhaite construire un graphe dont les sommets sont les villes, les arêtes les liaisons routières entre les villes et le poids des arêtes la distance en km entre les villes.

B1. Importer les données présentes dans le fichier dans une liste dont les éléments sont des tuples ("ville de départ", "ville d'arrivée", distance en km).

Solution : Au besoin, si ce code est difficile à comprendre, ne pas hésiter à faire des `print` à chaque étape pour visualiser le contenu des variables.

```
def import_csv(filename):
    # ouverture du fichier en lecture seule
    file = open(filename, 'r')
    # lecture de toutes les lignes ->  une liste de chaînes de caractères.
    all_lines = file.readlines()
    # fermeture du fichier (on en a plus besoin)
    file.close()
    head = all_lines[0].split(',')
    data = []
    for i in range(1, len(all_lines)): # pour chaque ligne
        # on récupère la liste des champs de la ligne courante
        words = all_lines[i].split(',')
        #print(words)
        # on ajoute les données en enlevant les espaces
        data.append((words[0].strip(), words[1].strip(), int(words[2])))
    return data

def import_csv_bis(filename):
    # ouverture du fichier en lecture seule
    file = open(filename, "r")
    # récupération des entêtes
    head = file.readline().split(',')
    data = []
    for line in file:  # énumère les lignes suivantes du fichier
        # on récupère la liste des champs de la ligne courante
        words = line.split(",")
        # on ajoute les données en enlevant les espaces
        data.append((words[0].strip(), words[1].strip(), int(words[2])))
    # fermeture du fichier (on en a plus besoin)
    file.close()
    return data
```

B2. Afin de construire un graphe, on souhaite utiliser une correspondance arbitraire entre le nom des villes et un numéro. Un dictionnaire Python est une structure de données qui associe une clef à une valeur et permet de réaliser cette correspondance. Écrire une fonction de prototype `create_mapping(data)` dont le paramètre est la liste `data` des données importées et qui renvoie le dictionnaire suivant :

```
mapping = {"Paris": 0, "Limoges": 1, "Toulouse": 2, "Tours": 3, "Poitiers": 4, "
    Bordeaux": 5, "Bayonne": 6, "Pau": 7, "Nantes": 8, "Vannes": 9, "Lorient":
    10, "Quimper": 11, "Brest": 12, "Rennes": 13, "Le Mans": 14}
```

On affecte un numéro différente en incrémentant au fur et à mesure que les villes apparaissent.

Solution :

```
def create_mapping(data):
    mapping = {} # un dictionnaire vide
    numero = 0 # le numéro de la première ville
    for v1, v2, _ in data:
        if v1 not in mapping:
            mapping[v1] = numero
            numero += 1
        if v2 not in mapping:
            mapping[v2] = numero
            numero += 1
    return mapping
```

B3. En utilisant le dictionnaire `mapping`, écrire une fonction de protoype `create_graph(data, mapping)` qui renvoie le graphe correspondant aux données importées sous la forme d'une liste d'adjacence.

Solution :

```
def create_graph(data, mapping):
    n = len(mapping)
    g = [[] for i in range(n)]
    for t1, t2, d in data:
        g[mapping[t1]].append((mapping[t2], d))
    return g
```

B4. En utilisant le dictionnaire `mapping`, écrire une fonction de protoype `create_matrix_graph(data, mapping)` renvoie le graphe correspondant aux données importées sous la forme d'une matrice d'adjacence de type tableau Numpy.

Solution :

```
def create_matrix_graph(data, mapping):
    n = len(mapping)
    m = np.zeros((n,n))
    for t1, t2, d in data:
       m[mapping[t1], mapping[t2]] = d
    return m
```

B5. En utilisant la fonction de Numpy `count_nonzero` (cf. documentation en ligne) et la représentation matricielle du graphe, construire la liste des degrés de chaque sommet du graphe. Quel sens pouvez-vous donner à cette information ?

Solution : Il s'agit du nombre de destinations directes possibles (voisines) à partir d'une ville. Plus le nombre est grand, moins une ville est isolée.

```python
def degrees(m):
    return np.count_nonzero(m, axis=1)
```

B6. En utilisant les fonction de Numpy (notamment `nonzero`, cf. documentation en ligne) et la représentation matricielle du graphe, calculer les distances minimales, maximales, moyennes ainsi que l'écart-type des distances entre les villes.

Solution :

```python
def stats(m):
    return np.min(m[np.nonzero(m)]), np.max(m), np.mean(m[np.nonzero(m)]),
        np.std(m[np.nonzero(m)])
```

B7. À l'aide des fonctions programmées à la section précédente, transformer la matrice d'adjacence du graphe en liste d'adjacence.

Solution :

```python
g = madj_to_ladj(m)
```

Algorithme 1 Parcours en largeur d'un graphe

```
1 : Fonction PARCOURS_EN_LARGEUR(G, s)                ▷ s est un sommet de G
2 :     F ← une file d'attente vide                   ▷ F comme file
3 :     D ← ∅                                         ▷ D ensemble des sommets découverts
4 :     P ← une liste vide                            ▷ P comme parcours
5 :     ENFILER(F,s)
6 :     AJOUTER(D,s)
7 :     tant que F n'est pas vide répéter
8 :         v ← DÉFILER(F)
9 :         AJOUTER(P, v)                             ▷ ou bien traiter le sommet en place
10 :        pour chaque voisin x de v dans G répéter
11 :            si x ∉ D alors                        ▷ x n'a pas encore été découvert
12 :                AJOUTER(D, x)
13 :                ENFILER(F, x)
14 :    renvoyer P                                    ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

B8. Écrire une fonction de prototype `parcours_largeur(g,s)` qui parcourt en largeur un graphe `g` sous la forme d'une liste d'adjacence à partir du sommet `s`. Cette fonction renvoie une liste de sommets représentant le parcours en largeur du graphe à partir de `s`. On utilisera une file d'attente réalisée à partir d'une liste Python en utilisant `append` et `pop(0)`.

Solution :

```
# lazy implemetation : lists
def parcours_largeur(g, depart):
    file = []
    decouverts = [False for _ in range(len(g))]
    parcours = []
    file.append(depart)
    decouverts[depart] = True
    while len(file) > 0:
        u = file.pop(0)   # O(n)
        parcours.append(u)
        for x in g[u]:
            if not decouverts[x]:   # O(1)
                decouverts[x] = True
                file.append(x)
    return parcours
```

Algorithme 2 Parcours en profondeur d'un graphe (version récursive)

```
1 : Fonction PARCOURS_EN_PROFONDEUR(G, s, D, C)          ▷ s est un sommet de G
2 :     AJOUTER(C, s)                                    ▷ s est ajouté au parcours du graphe
3 :     AJOUTER(D, s)                                    ▷ s est marqué découvert
4 :     pour chaque voisin x de s dans G répéter
5 :         si x ∉ D alors                               ▷ x n'a pas encore été découvert
6 :             PARCOURS_EN_PROFONDEUR(G, x, D, C)
```

B9. Écrire une fonction récursive de prototype `parcours_prof(g, s, decouverts, path)` qui parcourt en profondeur un graphe `g` sous la forme d'une liste d'adjacence à partir du sommet `s`. Cette fonction renvoie une liste de sommets représentant le parcours en profondeur du graphe à partir de `s`.

Solution :

```
def parcours_prof(g, s, decouverts, parcours):  # parcours en profondeur
    parcours.append(s)
    decouverts[s] = True
    for u in g[s]:
        if not decouverts[u]:
            parcours_prof(g, u, decouverts, parcours)  # récursif
```

B10. Écrire une fonction impérative de prototype `parcours_prof_i(g,s)` qui parcourt en profondeur un graphe `g` sous la forme d'une liste d'adjacence à partir du sommet `s`. Cette fonction renvoie une liste de sommets représentant le parcours en profondeur du graphe à partir de `s`. On utilisera une pile réalisée à partir d'une liste Python à l'aide des fonctions `append` et `pop()`.

Solution :

```
def parcours_prof_i(g, s):
```

Algorithme 3 Parcours en profondeur d'un graphe (version itérative)

```
1 : Fonction PARCOURS_EN_PROFONDEUR(G, s)                ▷ s est un sommet de G
2 :     P ← une pile vide                                ▷ P comme pile
3 :     D ← un ensemble vide                             ▷ D comme découverts
4 :     C ← un liste vide                                ▷ C pour le parcours
5 :     EMPILER(P,s)
6 :     tant que P n'est pas vide répéter
7 :         v ← DÉPILER(P)
8 :         AJOUTER(C, v)
9 :         pour chaque voisin x de v dans G répéter
10 :            si x ∉ E alors
11 :                EMPILER(P, x)
12 :                AJOUTER(D, x)
13 :    renvoyer C                                       ▷ Facultatif, on pourrait traiter chaque sommet v en place
```

```
        pile = []  # used as stack
        decouverts = [False for _ in range(len(g))]
        parcours = []
        pile.append(s)
        decouverts[s] = True
        while len(pile) > 0:
            u = pile.pop()
            parcours.append(u)
            for x in g[u]:
                if not decouverts[x]:
                    decouverts[x] = True
                    pile.append(x)
        return parcours
```

C Importation de données de fichiers textes

On dispose d'un fichier de données en mode texte [1] nommé `"data.csv"` dont le format est le suivant : `# id_objet, vitesse (m/s), masse (kg)`, c'est-à-dire chaque ligne est composée de trois champs d'information séparés par une virgule :

```
# id_objet, vitesse (m/s), masse (kg)
KA_2996, 3.6, 455
ML_432,5.7, 654
DY_332, 1.78, 345
...
```

Pour importer les données on procède comme suit :

```
# ouverture du fichier en lecture seule
```

1. Cela signifie que les informations sont codées dans le fichier par des caractères (code ASCII ou UTF-8 par exemple).

```
file = open("data.csv", "r")
# récupération des entêtes : on interprète , comme le séparateur des champs
head = file.readline().split(',')
data = []  # on récupère les données dans une liste
for line in file:  # énumère les lignes suivantes du fichier
    # récupérer la liste des champs de la ligne courante
    words = line.split(",")
    # ajouter les données en enlevant les espaces
    data.append((words[0], float(words[1]), int(words[2])))
    # fermeture du fichier (on en a plus besoin)
file.close()
```

- `open(filename,mode)` ouvre le fichier en mode lecture ("r") et/ou écriture "wr"
- `close()` ferme le fichier afin de signifier au système d'exploitation que la ressource est libre d'accès.
- `readline()` lire une ligne du fichier. La fonction renvoie une chaîne de caractères qui représente la ligne lue. La prochaine fois qu'on appelle cette fonction, c'est la ligne suivante qui sera lue.
- `readlines()` lire toutes les lignes du fichier. La fonction renvoie une liste de chaînes de caractères. Chaque élément de cette liste représente une ligne du fichier, dans l'ordre de lecture.
- `split(",")` décompose une chaîne de caractères d'après le séparateur ",". Cette fonction renvoie une liste de chaînes de caractères, chaque élément de la liste représente un champ d'information de la ligne.
- `strip()` permet de supprimer les caractères blancs (espaces, tabulation) des chaînes de caractères.
- `float(s)` permet de convertir une chaîne de caractères en nombre flottant (transtypage).
- `int(s)` permet de convertir une chaîne de caractères en nombre entier (transtypage).

D Dictionnaires

Un dictionnaire est un **tableau associatif** qui associe une clef `k` à une valeur `v` (cf. figure 1). À l'inverse des listes, cette structure n'est pas ordonnée. Les opérations sur un dictionnaire sont :

1. rechercher la présence d'une clef dans le dictionnaire (en $O(1)$),
2. accéder à la valeur correspondant à une clef,
3. insérer une valeur associée à une clef dans le dictionnaire,
4. supprimer une valeur associée à une clef dans le dictionnaire.

Voici deux exemples :

```
d = {}  # Empty dict
print(d, type(d))  # {} <class 'dict'>
d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
# keys are strings, values integers
print(d)  # {'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1}
d = {13: [1, 3], 219: [2, 1, 9], 42: [4, 2]}
print(d, type(d))  # keys are integers, values lists
# {13: [1, 3], 219: [2, 1, 9], 42: [4, 2]} <class 'dict'>
```

La fonction `len` renvoie le nombre de clefs d'un dictionnaire.

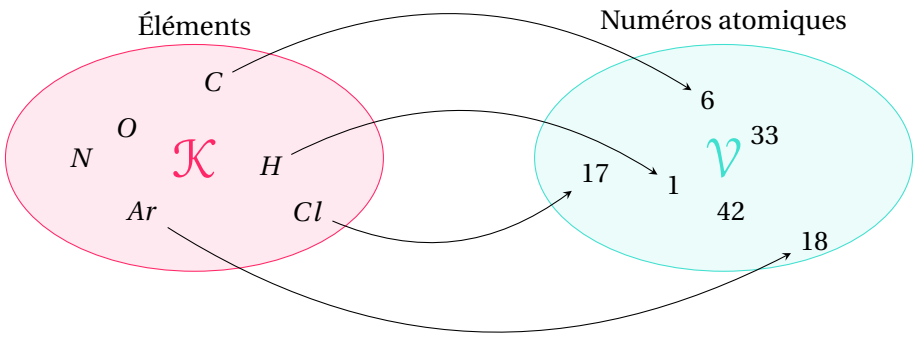


FIGURE 1 – Illustration du concept de dictionnaire, ensembles concrets

```
d = dict([("Ar", 18), ("Na", 11), ("Cl", 17), ("H", 1)])
print(len(d)) # 4
```

Les mots-clefs `in` et `not in` permettent de tester l'appartenance à un dictionnaire et renvoient les booléens correspondants.

```
d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
if  "Ar" in d:
    print("Ar", d["Ar"]) # Ar 18
if "O" not in d:
    print("O is not in d")  # O is not in d
```

L'opérateur `[]` est nécessaire pour ajouter une valeur d'après sa clef.

```
d = {"Ar": 18, "Na": 11, "Cl": 17, "H": 1}
d["O"] = 8
print(d)  # {'Ar': 18, 'Na': 11, 'Cl': 17, 'H': 1, 'O': 8}
```
