# Dictionnaires et tables de hachage

## I - Dictionnaires Python : définition, création et exploitation

### 1. Définition des entrées / sorties

Un dictionnaire est une structure de données native dans Python (c'est-à-dire présente par défaut). À la différence des structures de données séquentielles (listes, chaînes de caractère ou tuples), qui sont indexées par des nombres, les objets contenus dans les dictionnaires sont indexés par des **clés**. À chaque objet stocké dans le dictionnaire appelé **valeur** est associé une **clé** permettant l'accès à cette **valeur**. Chaque clé **doit être unique** afin de pouvoir retrouver la **valeur** qui lui correspond.

En Python, il y a deux façons de **déclarer un dictionnaire** :

- soit en écrivant directement entre accolades les paires (clé :valeur) sous la forme

  `dico = { cle1 : valeur1, cle2 : valeur2, ..., clen : valeurn }`

  Remarquez que l'on sépare les éléments d'une paire (clé :valeur) par le caractère : et les paires par des virgules. Donnons un premier exemple pour illustrer cette méthode.

```
1 mon_aquarium={"corydoras" : "poisson de fond", "néon" : "autre poisson"}
2 # Ici on a un dictionnaire formé de deux paires clé:valeur où les clés
3 # sont les noms des poissons et les valeurs leurs habitats
```

- soit on peut également partir d'un dictionnaire déjà existant, qu'il soit vide (ce que l'on peut écrire `{}` ou encore `dict()` ) ou non, et y ajouter de nouvelles paires (clé :valeur). Cette fois, la syntaxe pour ajouter une paire est la suivante :

```
1 nouvel_aquarium = {} # C'est un aquarium vide
2 nouvel_aquarium["Discus"] ="eaux chaudes"
3 # On ajoute le couple clé:valeur -> "Discus":"eaux chaudes"
4 mon_aquarium["Discus"] ="eaux chaudes"
5 # On peut également l'ajouter à l'aquarium précédent.
6 print("on a ajouté un poisson dans l'aquarium qui contient désormais ")
7 print(mon_aquarium)
```

Ce code affichera alors :

```
on a ajouté un poisson dans l'aquarium qui contient désormais
{'corydoras': "poisson de fond", 'néon': 'autre poisson', 'Discus': "eaux chaudes"}
```

## 2. Exemples d'application

Sachant que les **clés sont uniques**, que se passe-t-il lorsqu'on exécute maintenant les lignes de code ci-dessous ?

```
1 print('mon aquarium avant de modifier la valeur associée à une clé déjà présente')
2 print(mon_aquarium)
3 mon_aquarium["corydoras"] = "poisson d'eau douce"
4 print('mon aquarium après avoir modifié la valeur associée à une clé déjà présente')
5 print(mon_aquarium)
```

*Réponse* : ce code ne génère pas d'erreur, il va donc affecter la valeur "poisson d'eau douce" à la clé "corydoras". Cette action va tout simplement remplacer l'ancienne valeur associée à cette clé. Le code affiche alors :

```
mon aquarium avant de modifier la valeur associée à une clé déjà présente
{'corydoras': "poisson de fond", 'néon': 'autre poisson', 'Discus': "eaux chaudes"}
mon aquarium après avoir modifié la valeur associée à une clé déjà présente
{'corydoras': "poisson d'eau douce", 'néon': 'autre poisson', 'Discus': "eaux chaudes"}
```

Chaque **clé** est utilisée par l'ordinateur pour référencer la **valeur** qui lui correspond. En effet, les différentes valeurs sont rangées dans une structure séquentielle à un emplacement qui est calculé à partir de la clé (on précisera comment dans la partie *Tables de hachage*). Pour ne pas que l'emplacement d'une valeur change par intermittence, Python exige que les clés soient constituées d'objets **immuables** (que l'on ne peut pas modifier).[1]. Observez les lignes suivantes :

```
dico1 = dict()
dico1[1] = 'une première clé'
dico1[2] = [1,2,3]
dico1['3-ième clé']= 'pas de problème'
```

Ce code fournit alors le dictionnaire, normalement sans surprise,

```
dico1 = {1: 'une première clé', 2: [1, 2, 3], '3-ième clé': 'pas de problème'}
```

En revanche, observons le code suivant.

```
dico2 = dict()
cleliste = [1,2]
dico2[cleliste] = 'et celle-ci ?'
```

Cette fois-ci, le code génère une erreur `TypeError: unhashable type: 'list'` indiquant que le type **list** n'est pas valide pour construire une paire *clé : valeur*. **On ne peut donc pas utiliser n'importe quel objet pour définir une clé, il faut qu'il soit immuable.**

## 3. Accès aux éléments d'un dictionnaire (écriture de nouveaux couples clé : valeur et accès à une valeur depuis sa clé)

Comme on l'a vu dans l'exemple précédent, si on ajoute une clé qui existe déjà, on va écraser la valeur qui lui était associée. Pour ajouter un couple clé : valeur (ou écraser un tel couple si la clé existe déjà), on utilise la syntaxe

```
1 dictionnaire[cle] = valeur_a_ajouter
```

Si l'on souhaite récupérer une valeur déjà stockée à partir de la connaissance de sa clé, il faut renverser l'expression :

```
1 valeur_presente = dictionnaire[clé]
```

1. Rappelons, comme on l'a vu dans les chapitres précédents, que les *nombres*, les *chaînes de caractères*, les *tuples*, sont des objets **immuables**. En revanche, les *listes* sont des objets **mutables**.

Ainsi, on stocke dans la variable **valeur_présente** la valeur associée à **clé**. Si cette clé n'existe pas dans le dictionnaire, alors l'instruction renverra une erreur. Une bonne manière de traiter le cas où on n'est pas sûr de la présence d'une clé est l'utilisation de la méthode `dictionnaire.get(cle)` qui renvoie la valeur associée à `cle` si c'est une clé du dictionnaire, et **None** sinon. On supposera que cette méthode, tout comme l'accès à une valeur à partir de sa clé ou l'insertion d'un couple (clé : valeur), s'exécute en temps constant ($O(1)$) indépendamment du nombre de clés déjà stockées dans `dictionnaire`. Cette hypothèse est discutée dans la partie *tables de hachage*. On utilise cette méthode notamment pour savoir quand on alterne entre insertions de couples clé :valeur et modification de valeurs associées à des clés existantes :

```
1
2 if dictionnaire.get(cle) == None:
3         dictionnaire[cle] = 1 # insertion du couple clé:valeur
4 else:
5         dictionnaire[cle] += 1 # modification de la valeur précédente
```

En réalité, on peut remplacer l'instruction `dictionnaire.get(cle) == None` par `cle in dictionnaire` qui fait exactement la même chose en Python.

### 4. Syntaxes alternatives pour déclarer un dictionnaire (à savoir comprendre mais pas nécessairement utiliser)

Le constructeur d'objet `dict()`, vu précédemment pour créer un dictionnaire vide, peut être utilisé avec des arguments pour créer directement un dictionnaire non vide. Il attend en entrée soit

- Une liste de paires (clé,valeur) :

```
1 liste_cles = [("voici",4),("dragon",7),("trone",9)]
2 dictionnaire = dict(liste_cles)
3 print(dictionnaire)
```

  ce qui construira le dictionnaire `{'voici': 4, 'dragon': 7, 'trone': 9}`;
- soit des noms de clés passées directement comme des arguments par défaut du constructeur (dans ce cas **les clés** construites seront forcément des chaînes de caractères)

```
1 dictionnaire2 = dict(voici = 4,dragon = 7, trone = 9)
```

  On obtient alors le même dictionnaire que précédemment.

Il est également possible de supprimer une clé d'un dictionnaire (même si en pratique cela ne nous servira jamais) grâce à la commande `del` suivie de l'élément du dictionnaire à supprimer :

```
1 del dictionnaire2['dragon']
```

fournit le dictionnaire `dictionnaire2 = dict(voici = 4, trone = 9)`.

**À Retenir**

- Il faut savoir **définir un dictionnaire** (avec dict() ou {}) et savoir y **ajouter des couples clé :valeur** ;
- Pour **exploiter un dictionnaire**, on souhaite, à partir de la donnée d'une **clé**, pouvoir récupérer la valeur **associée**. La syntaxe pour extraire une valeur est la même que celle pour accéder à un élément d'une liste, sauf qu'ici au lieu d'être indexée par un indice, les valeurs sont indexées par leurs clés ;
- Enfin, **dictionnaire**[**clé**] contient la **valeur** associée à **clé** si le couple **clé :valeur** est dans **dictionnaire**. Si la **clé** n'est pas présente, cette instruction renverra une erreur.

### 5. Exercices d'application sur la manipulation des dictionnaires

**1** Créer un dictionnaire `pokemon` dont les clés sont les mots carapuce, salameche et bulbizarre et dont les valeurs sont le nombre de lettre dans la chaîne de caractères correspondant à la clé.

**2** Écrire l'instruction permettant d'ajouter pikachu à ce dictionnaire.

**3** Créer un second dictionnaire `pokemon_2` dont les clés sont les mots kaiminus, héricendre et germignon et les valeurs le nombre de lettres dans la chaîne de caractères correspondant à la clé. Créer enfin un dictionnaire `starter` constitué des couples (clé :valeur) : `("1ère génération":pokemon)` et `("2ème génération":pokemon_2)`. Ainsi chacune des valeurs de `starter` et l'un des dictionnaires précédemment définis.

**4** Que renvoie l'instruction `starter["2ème génération"][kaiminus]` ? Et `starter["1ère génération"][kaiminus]`

## II - Extraction des clés ou des valeurs d'un dictionnaire

### 1. Syntaxe

Lorsqu'on ne sait pas à l'avance quelles sont les clés / valeurs présentes dans un dictionnaire, ou tout simplement si l'on souhaite parcourir tous ses éléments, on peut utiliser les méthodes `.keys()`, `.values()` ou `.items()`. Reprenons l'exemple précédent, le code :

```
1 print(dictionnaire)
2 print(dictionnaire.keys())
3 print(dictionnaire.values())
4 print(dictionnaire.items())
```

affiche les résulats :

```
{'voici': 4, 'dragon': 7, 'trone': 9}
dict_keys(['voici', 'dragon', 'trone'])
dict_values([4, 7, 9])
dict_items([('voici', 4), ('dragon', 7), ('trone', 9)])
```

On remarque que les méthodes `.keys()`, `.values()` et `.items()` permettent bien d'extraire les séquences des clés, des valeurs ou des couples (clé,valeur). Mais ceux-ci sont stockés dans des structures un peu étranges `dict_keys`, `dict_values` et `dict_items`. Ces structures sont des **itérables**, il est donc **possible de les parcourir** avec une structure de la forme

```
1 for cle in dictionnaire.keys():
```

En revanche, **ce ne sont pas des listes** d'éléments, ainsi `dictionnaire.keys()[0]` ne renverra pas la première clé mais une erreur. On peut toutefois **transformer ces structures en listes** en utilisant la commande `list(...)` :

```
1 print(list(dictionnaire.keys()))
2 print(list(dictionnaire.values()))
3 print(list(dictionnaire.items()))
```

La console affiche alors :

```
['voici', 'dragon', 'trone']
[4, 7, 9]
[('voici', 4), ('dragon', 7), ('trone', 9)]
```

## 2. Exercices sur les dictionnaires

**Questions d'application directe**

**1** Écrire une fonction `noms_en_n(pokemons,n_lettres)` qui renvoie la liste des noms des pokemons dont le nom est constitué de `n_lettres` lettres. L'argument `pokemons` est un dictionnaire dont la structure est semblable aux variables `pokemon` et `pokemon_2` construites précédemment.

**2** Écrire une fonction `starters_en_n(starters,n_lettres)` qui effectue la même tâche que la fonction précédente mais en se basant sur la variable `starter` définie précédemment (cette fois la fonction renvoie donc la liste de tous les starters, des générations 1 et 2 dont le nom est constitué de `n_lettres` lettres).

**Utilité (ou non) des dictionnaires : Un problème de comptage**

On travaille sur une séquence ADN de longueur $N$ constituée des nucléotides A,T,C,G. Une telle séquence peut être représentée en Python par une chaîne de caractère de la forme `sequence =" AAAGGTCACCGTCGATC...ATGAT"` On appelle $n$-gramme une sous-chaîne de caractères de taille $n$ extraite de la chaîne de caractère originale. Les 1-grammes de `sequence` sont "A","T","C" et "G". Les 2-grammes sont "AA","AT",...,,"TG","CG".

**1** Écrire une fonction `statsLettres(sequence)` qui reçoit en argument une chaîne de caractères `sequence` et renvoie un dictionnaire dont les clés sont les lettres du texte et les valeurs sont le nombre d'occurrences de chaque lettre. On pourra vérifier le bon fonctionnement de votre fonction grâce au test :

```
1 statsLettres('ACCTGAAGTC') == {'A':3,'C':3,'T':2,'G':2}
```

**Rappel** : pour tester si une clé `cle` est présente dans un dictionnaire `dico`, on peut utiliser la méthode `dico.key(cle)` qui est une opération de complexité temporelle $O(1)$.

**2** Écrire une fonction `statsBigrammes(sequence)` qui renvoie le dictionnaire dont les clés sont les bigrammes (mots de 2 lettres) de la chaîne de caractères `sequence` et les valeurs associées sont le nombre d'occurrences de ces bigrammes. On pourra vérifier la validité de notre fonction avec le test :

```
1 statsBigrammes('ACCTAC') == {'AC':2,'CC':1,'CT':1,'TA':1}
```

**3** (*Question de dénombrement*, vous pouvez la sauter éventuellement) Quel est le nombre de $n$-grammes différents que l'on peut constituer à partir des lettres prises dans un alphabet de $k$ symboles (pour `sequence`, $k = 4$).

On souhaite compter le nombre d'apparition de chaque $n$-gramme. En pratique pour des valeurs raisonnablement grandes de $n$, les $n$-grammes qui apparaissent dans sequence ne sont issus que d'une fraction réduite de tous les $n$-grammes possibles.

**4** (*Question de dénombrement*) Pour $n$ fixé, pour quelles valeurs de $N$ est on certain que tous les $n$-grammes n'apparaîtront pas dans la séquence de nucléotides.

Dans la suite, on supposera que $n$ est suffisamment grand pour qu'il ne soit pas efficace de stocker tous les $n$-grammes possibles.

**5** *Stratégie numéro 1 : décompte par tableau de correspondances* - Écrire une fonction `compte_n_gram(sequence,n)` qui renvoie deux listes `L_n_grams` et `compte_n_grams` telles que `compte_n_grams[i]` est le nombre d'apparitions du $n$-gramme `L_n_grams[i]` dans `sequence`.

*Aide : on pourra s'aider du canevas ci-dessous*

```
1 def compte_n_gram(sequence,n):
2         N = len(sequence)
3         L_n_grams,compte_n_grams = [],[]
4         for i_debut in range(N-n+1):
5                 n_gram = sequence[....:....]
6                 indice_n_gram = indice(n_gram,L_n_grams)
7                 if indice_n_gram == None:
8                         # Ajout du n_gram à ceux découverts :
```

```
9                        ................
10                       ................
11               else:
12                       # augmentation du décompte du n_gram de 1
13                       ................
14       return L_n_grams,compte_n_grams
15
16 def indice(n_gram,L_n_grams):
17       '''renvoie l'indice d'apparition de n_gram dans L_n_grams s'il est
18        présent et None sinon'''
19       i = 0
20       while i<len(L_n_grams):
21               if .......................:
22                       return i
23               ...............
24       return None
```

**6** Évaluer la complexité de la fonction précédente en fonction de $n$ et $N$.

**7** *Stratégie numéro 2 : décompte par dictionnaire* - Écrire une fonction `compte_n_gram_dico(sequence,n)` qui renvoie le dictionnaire dont les clés sont les $n$-grammes de `sequence` et les valeurs sont les nombres d'apparitions correspondantes. *Indication : C'est juste une généralisation de statsBigrammes*

**8** On admettra que les accès aux éléments du dictionnaire, les insertions de paires (clé,valeur) et le test de présence d'une clé dans le dictionnaire se font en $O(1)$. Quelle est la complexité de `compte_n_gram_dico(sequence,n)` . Conclure.

## Représentation de graphes par des dictionnaires

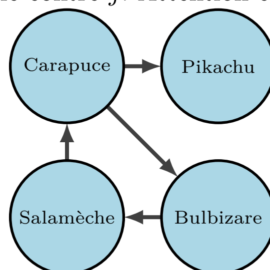
On a vu en première année que l'on peut représenter les graphes soit

- Par une matrice d'adjacence, c'est à dire une matrice $M$ de taille $n \times n$ où $n$ est le nombre de nœuds du graphe telle que $M_{i,j} = 1 \iff$ il existe un arc reliant les nœuds d'indice $i$ et $j$ et 0 sinon.
- Ou par une liste d'adjacence c'est à dire une liste de longueur $n$ telle que l'élément à la position $i$ est la liste des indices des nœuds $j$ pour lesquels il existe un arc de $i$ vers $j$.

Dans les deux cas, l'encodage du graphe repose sur une indexation des nœuds par des entiers. On se propose d'étendre la notion de liste d'adjacence à celle de dictionnaire d'adjacence : un dictionnaire d'adjacence est un dictionnaire tel que

- Les clés correspondent aux identifiants des nœuds.
- Chaque clé $i$ est associée à une valeur qui est la liste (éventuellement vide) des nœuds $j$ tels qu'il existe un arc de $i$ vers $j$.

**1** Représenter sous la forme d'un dictionnaire d'adjacence le graphe ci-contre tel qu'il existe un arc entre les pokemons $i$ et $j$ si le pokemon $i$ est faible contre $j$. Attention ce dictionnaire est censé avoir 4 clés.




**2** On considère le dictionnaire d'adjacence `dico` obtenu à l'issue de la définition ci-dessous. Le graphe correspondant est il orienté ? Acyclique ?

```
1 dico = {'A':'B','C':'B'}
2 dico['B'] = 'D'
3 dico['A'] = 'D'
4 dico['D'] = 'C'
5 print(dico)
```

# III - Tables de hachage

On a vu que les dictionnaires permettent d'effectuer les opérations élémentaires suivantes :

- création d'un dictionnaire (éventuellement vide),
- ajout d'une correspondance entre une clé $c$ et une valeur $v$,
- suppression d'une correspondance associée à une clé ( en réalité on s'en sert très peu et on ne détaillera pas l'analyse de cette opération mais en Python, on peut le faire avec l'instruction `del dico[cle]`).
- test de la présence d'une clé et récupération de la valeur associée

Ce qui est moins clair, c'est quelle est la complexité associée à ces différentes opérations. Dans cette section on s'intéresse au fonctionnement et aux propriétés des tables de hachages sur lesquelles repose l'implémentation Python des dictionnaires.

## 1. Motivation

Les dictionnaires sont des structures permettant d'accéder à des données en utilisant une clé. On peut construire une structure naïve de dictionnaire reposant sur deux listes : une de clés et une autre de valeurs pour implémenter un dictionnaire. Pour implémenter avec une telle structure le dictionnaire présenté en exemple dans la partie II, on aurait la structure `dictionnaire` ci-dessous. On a également implémenté une fonction `get_valeur` permettant d'accéder à la valeur de `dictionnaire` pour une `cle` passée en argument :

```
1  cles = ['voici','dragon','trone']
2  valeurs = [4,7,9]
3  dictionnaire = (cles,valeurs) # le dictionnaire est une paire de listes
4  def get_valeur(cle,dictionnaire):
5          ''' Renvoie la valeur associée à cle dans dictionnaire si cette cle
6          est présente. Renvoie None sinon '''
7          cles,valeurs = dictionnaire
8          for i in range(len(cles)):
9                  if cles[i] == cle:
10                         return valeurs[i]
11         return None # cle n'est pas présente dans dictionnaire
```

**1** Quelle est la complexité dans le meilleur cas et dans le pire cas de la fonction `get_valeur` (en fonction du nombre de clés du dictionnaire $n$).

**2** Proposer une fonction `insere_couple(paire_cle_valeur,dictionnaire)` qui insère le couple `(cle,valeur)` contenu dans la variable `paire_cle_valeur` à l'intérieur de `dictionnaire` si la clé `cle` n'y est pas déjà et remplace le couple existant sinon. La fonction renverra le dictionnaire ainsi modifié.

**3** Montrer que la complexité dans le pire cas de la fonction `insere_couple` est $O(n)$.

On remarque que la complexité est la même que celle de la recherche d'un élément dans une liste de longueur $n$. Or on a vu en première année que dans le cas des listes triées, on peut considérablement réduire cette complexité quitte à travailler avec une liste triée. La stratégie est ici différente, mais les tables de hachage vont permettre, quitte à introduire des restrictions sur les clés utilisables, d'améliorer considérablement la complexité des opérations élémentaires.

## 2. Tables d'associations et tables de hachage

La structure de dictionnaire développée dans la section motivation est appelée **table d'association**. Dans le cas présenté, les clés étaient rangées sous la forme d'une liste non triée. Il existe également des implémentations

de tables d'association reposant sur l'usage de listes triées (il existe alors des techniques pour définir une relation d'ordre sur les clés) ou sur des structures de graphes particulières (arbres ABR nécessitant à nouveau une relation d'ordre sur les clés encodées). Dans ces types d'implémentations, on peut réduire la complexité des opérations d'accès à un élément et d'ajout / retrait d'un couple (clé,valeur) à $O(\log(n))$. L'étude de telles structures ne relève pas du programme d'informatique de tronc commun et ne seront traitées qu'en TD. En revanche on va s'intéresser à l'implémentation la plus classique des dictionnaires reposant sur un principe légèrement différent : les tables de hachage. Ces structures utilisent **une fonction de hachage** : On note $E$ un ensemble auxquelles appartiennent les clés d'un dictionnaire et $n$ un entier positif.

Une fonction de hachage $h$ est une fonction $h : E \mapsto \{0, 1, \ldots, n-1\}$

Ainsi pour un élément $x$ de $E$, on appelle $h(x)$ le *haché* de $x$. Ces fonctions permettent d'associer à des éléments de l'ensemble $E$ (qui peuvent être des objets très différents : des nombres, des chaînes de caractères, etc) un entier. Cet entier permet alors de définir une adresse (au sein d'un tableau de longueur $n$ par exemple) où sera stockée la valeur associée à la clé hachée. On résume la stratégie (naïve) d'utilisation de la fonction de hachage sur la Figure 1.

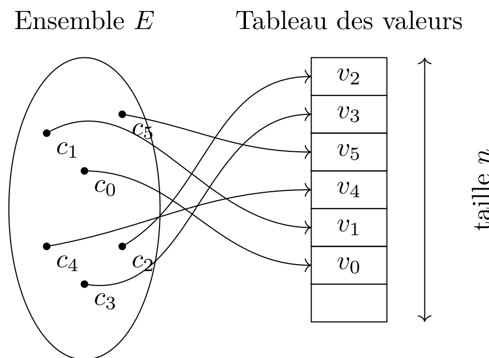



Figure 1 – Schéma symbolique de la représentation d'un dictionnaire. Chaque clé de $E$ est représentée par un élément $c_k$ et chaque valeur associée par un élément $v_k$. La fonction de hachage fait correspondre à chaque valeur un indice ce qui permet de les ranger dans un tableau.

Une *bonne* fonction de hachage est une fonction qui à chaque élément de E associe un entier différent. Malheureusement pour des ensembles $E$ de cardinal infini (et c'est le cas : tous les nombres entiers et flottants, toutes les chaînes de caractères, etc) il n'existe pas de telles fonctions car l'ensemble des hachés est lui de cardinal $n$. Il en résulte que pour une fonction de hachage donnée, on pourra trouver une paire $x, y \in E$ telle que $x \neq y$ et $h(x) = h(y)$. On parle alors de **collision**. Le risque dans ce cas est de ne plus pouvoir attribuer des valeurs différentes à des clés différentes comme on le voit sur la représentation Figure 2.

Il existe de nombreuses stratégies pour résoudre le problème des collisions et on en présente une dans la partie Exercices (*gestion des collisions par chaînage*).

**Remarque** : La fonction de hachage fait correspondre un unique entier à une entrée. C'est pourquoi on ne peut pas utiliser d'objets mutables en tant que clés : si l'objet pouvait être modifié, la fonction de hachage pourrait faire varier l'indice correspondant dans le tableau des valeurs ce qui forcerait à faire des réactualisations du dictionnaire à chaque modification de la clé.

### 3. Complexité des opérations élémentaires sur les dictionnaires implémentés par tables de hachage

En vue d'établir la complexité des opérations élémentaires sur les dictionnaires implémentés avec des tables de hachages, on supposera que les collisions ont un impact négligeable. Cette hypohtèse est vérifiée si

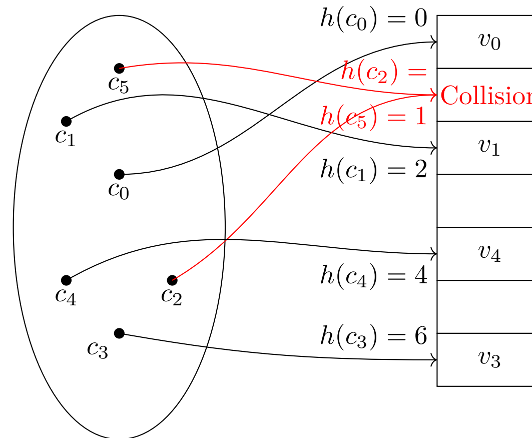



FIGURE 2 – Schéma illustrant une collision.

- la taille $n$ du tableau de valeurs est grande devant le nombre valeurs réellement stockées
- et si la fonction de hachage choisie répartit uniformément les indices correspondant aux clés dans $\{0, \ldots, n-1\}$.

Sous ces hypothèses, on peut évaluer les complexités des opérations d'accès à la valeur correspondant à une clé et d'insertion d'un nouveau couple `(cle,valeur)`. On notera `L_valeurs` le tableau contenant les différentes valeurs et $h$ la fonction de hachage faisant correspondre à chaque clé un indice dans `L_valeurs`.

- L'accès à la valeur correspondant à une clé `cle` nécessite de (1) calculer `h(cle)` (complexité temporelle en $O(1)$), puis (2) récupérer la valeur `L_valeurs[h(cle)]` (également en $O(1)$). Cette opération a donc une complexité temporelle en $O(1)$.
- L'insertion d'un nouveau couple `(cle,valeur)` nécessite de (1) calculer `h(cle)` (complexité temporelle en $O(1)$), puis placer `valeur` dans `L_valeurs[h(cle)]` (également en $O(1)$). Cette opération a donc une complexité temporelle est donc en $O(1)$.

Les tables de hachages permettent donc d'avoir des complexités sur les opérations élémentaires indépendantes du nombre de couples `(cle,valeur)` déjà présents dans le dictionnaire.

## 4. Exercices

**Manipulation des fonctions de hachages et collisions :**

**1** Soit la fonction $f$ définie sur $\mathbb{Z}$ par $f : x \mapsto (x+2)\ mod\ 10$ où $a\ mod\ b$ désigne le reste de la division euclidienne de $a$ par $b$. Montrer que $f$ est une fonction de hachage valide en précisant sur quel ensemble ainsi que le nombre $n$ de hachés différents produits.

**2** Déterminer l'ensemble des entiers naturels qui sont en collision avec $x = 1$.

On construit une fonction de hachage plus complexe $g$ définie sur $\mathbb{N} \cup \mathcal{A}$ où $\mathcal{A}$ est l'ensemble des chaînes de caractères (représentées par des objets de type `str`). On note $pos(lettre)$ la position de la première lettre de $mot$ dans l'alphabet ($pos('abeille') = 0$, $pos('bazar') = 1$, etc). $g$ est définie par la relation :

$$\forall x \in \mathbb{N} \cup \mathcal{A}\ g(x) = \begin{cases} h(x) \text{ si } x \in \mathbb{N} \\ pos(x) + 10 \text{ sinon} \end{cases}$$

**3** Que vaut $g('hachage')$ ?

**4** Spécifier l'ensemble des chaînes de caractères qui peuvent rentrer en collision avec $x = 20$. Et avec $x = 4$ ?

**5** En inspectant les résultats produits par la fonction `hash` de Python, préciser si la stratégie de gestion des types différents proposée ici est réellement utilisée en pratique. Mettre en avant un cas de collision pour cette fonction de hashage.

## Implémentation d'un dictionnaire par une paire de listes ordonnées selon les clés

Dans ce problème, on souhaite stocker des paires (clés,valeurs) en stockant les clés dans une liste (`L_cles`) et les valeurs dans une autre (`L_valeurs`). Un dictionnaire est donc une paire `(L_cles,L_valeurs)` telle que les éléments `L_cles[i]` et `L_valeurs[i]` forment une paire (clé,valeur). Contrairement à l'implémentation proposée dans la section Motivation, on dispose ici d'une fonction $f$ qui permet de représenter chaque clé par un nombre entier différent ($f$ n'est pas une fonction de hachage au sens où celle-ci fournit toujours des images différentes pour deux clés différentes, il n'y a donc pas de risque de collision). La liste `L_cles` est supposée initialement triée selon les $f(cle)$ croissants.

**1** Compléter la fonction `get_valeur(cle,dictionnaire,f)` qui renvoie la valeur associée à `cle` si cette clé est dans `dictionnaire` et `None` sinon. Les clés sont triées par $f(cle)$ croissants. Quelle est la complexité de cette fonction (en fonction du nombre $n$ de clés du dictionnaires), on supposera que la fonction $h$ s'exécute en $O(1)$.

```
1  def get_valeur(cle,dictionnaire,f):
2          cles,valeurs = dictionnaire
3          gauche,droite = 0, len(cles) - 1
4          while (droite - gauche) > ........:
5                  milieu = (gauche+droite)//2
6                  if cles[milieu]  == cle:
7                          return ............
8                  if f(cles[milieu]) > .................. :
9                          ........  = milieu
10                 else:
11                         ........ = milieu
12
13         if f(cles[milieu])  == cle:
14                 return ............
15         else:
16                 return ............
```

**2** On souhaite désormais insérer un nouveau couple (`(cle,valeur)` dans le dictionnaire. Pour cela on peut utiliser une fonction similaire à `get_valeur` pour calculer l'indice auquel `cle` et `valeur` doivent être insérés avec une fonction `indice_insertion(element,dictionnaire)` puis écrire une fonction `inserer_dico(element,indice,dictionnaire)` qui renvoie le dictionnaire `dictionnaire` après insertion du couples (clé,valeur) contenu dans `element` à la position `indice`. Implémenter ces fonctions. (*Aide :* `inserer_dico` *il s'agit d'une fonction très proche de celle vue dans le cas du tri par insertion - cahier de vacances / TD 1ère année)*

**3** Quelle est la complexité de la fonction précédente dans le meilleur et dans le pire cas ? Préciser si le fait de travailler avec une liste triée selon les clés apporte une amélioration pour le problème d'insertion d'un couple (`clé,valeur`) dans un dictionnaire.

## Un exemple de table de hachage sans gestion des collisions

On souhaite implémenter une structure élémentaire de table de hachage sans gestion des collisions. On se restreint ici à des clés entières et on utilisera la fonction de hachage définie sur $\mathbb{N}$ par $h(x) = x + 2 \ mod \ 10$. Remarquons que dans le cas étudié, un dictionnaire n'est représenté que par un tableau de longueur $n$ stockant différentes valeurs associées à des clés.

**1** Quelle sera une bonne valeur de $n$ pour notre table de hachage ?

**2** Écrire en Python la fonction $h$ (on prendra soin de vérifier que l'entrée de $h$ est un entier positif au moyen d'une assertion).

Les valeurs associées aux clés sont stockées dans une liste `L_valeurs` de longueur $n$. Le dictionnaire vide sera représenté par une liste de longueur $n$ ne contenant que la valeur `None`.

**3** Écrire une fonction `get_valeur(L_valeurs,cle,h)` renvoyant la valeur associée à `cle` si elle existe et None sinon.

**4** Écrire une fonction `insere_valeur(L_valeurs,(cle,valeur),h)` qui insère `valeur` au bon emplacement de `L_valeurs` pour la clé `cle`.

**5** Écrire une fonction `est_vide(L_valeur)` qui renvoie `True` si le dictionnaire représenté par `L_valeur` est vide et `False` sinon.

## Résolution des collisions par chaînage séparé

Une stratégie de résolution des collisions repose sur le chaînage séparé. Cette méthode consiste à placer dans le tableau des valeurs des listes pouvant contenir plus d'un élément. Dans ce cas, on n'y stocke plus uniquement les valeurs mais des listes contenant les couples `(cle,valeur)` afin de distinguer des couples de clés qui entrent en collision pour la fonction de hachage. On note $h$ la fonction de hachage utilisée. Le code ci-dessous fourni un exemple d'implémentation où les clés sont des chaînes de caractères et les valeurs sont des entiers :

```
1 dictionnaire = [[],[('bonjour',42)],[],[('a',12),('bazar',21)],[]]
```

**1** Combien de valeurs différentes peut produire la fonction de hachage utilisée ?

**2** Que vaut $h('a')$ ?

**3** Écrire une fonction `get_valeur(L_valeurs,cle,h)` renvoyant la valeur associée à `cle` si elle existe et None sinon.

**4** Écrire une fonction `insere_valeur(L_valeurs,(cle,valeur),h)` qui insère le couple `(cle,valeur)` au bon emplacement de `L_valeurs` pour la clé `cle`. Si la clé existe déjà, la valeur correspondante sera remplacée. Si celle ci n'existe pas encore et s'il y a collision, le couple `(cle,valeur)` sera placé à la fin de la liste des couples entrant en collision avec celui-ci.

**5** Écrire une fonction `est_vide(L_valeur)` qui renvoie `True` si le dictionnaire représenté par `L_valeur` est vide et `False` sinon.