Dictionnaires

I Table de hachage

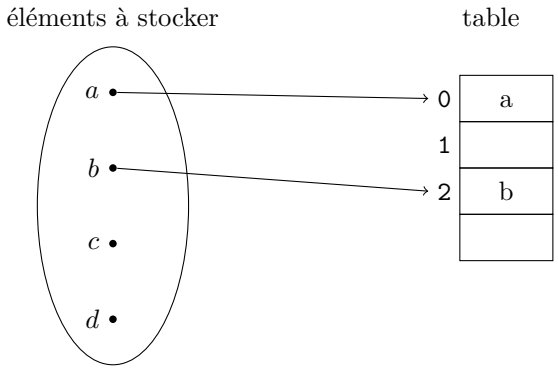
Il est important de disposer de structures de données permettant de stocker des informations de façon efficace, c'est-à-dire de façon à ce que les opérations de bases aient une complexité la plus faible possible.
Une liste est une structure de données pratique à utiliser mais qui possède certains défauts. Certaines opérations ont une complexité constante (ie indépendante de la taille de la liste) : créer une liste, accéder ou modifier un élément en utilisant son indice, accéder à la taille de la liste, ajouter ou supprimer le dernier élément de la liste, ... Par contre la recherche de l'appartenance d'un objet à une liste (tester avec `in` par exemple) demande de la parcourir intégralement, surtout si cet objet n'appartient pas à la liste, donc cette opération possède une complexité linéaire (ie $O(n)$ si n est la longueur de la liste).
Le principe d'une *table de hachage* est de stocker les informations dans un « tableau », suffisamment grand, dans lequel la position de n'importe quel élément peut être recalculée rapidement ; ainsi, lorsque on cherche si un élément appartient au tableau, il suffit de tester la case dans laquelle il est susceptible de se trouver, s'il n'y est pas, il n'est pas dans le tableau.

1. Fonction de hachage

On suppose donc posséder un tableau de taille N, suffisamment grand pour y stocker les informations que l'on souhaite mémoriser. À chaque objet que l'on veut mémoriser, il faut associer un entier qui correspondra à son indice dans le tableau ; une *fonction de hachage* est une fonction `h` qui associe à chaque objet cet indice. Il est important qu'une telle fonction soit simple à calculer pour des raison d'efficacité. En Python, il existe une fonction `hash` qui associe à un objet un entier (compris entre -2^{63} et $2^{63}-1$ pour un système 64 bits).

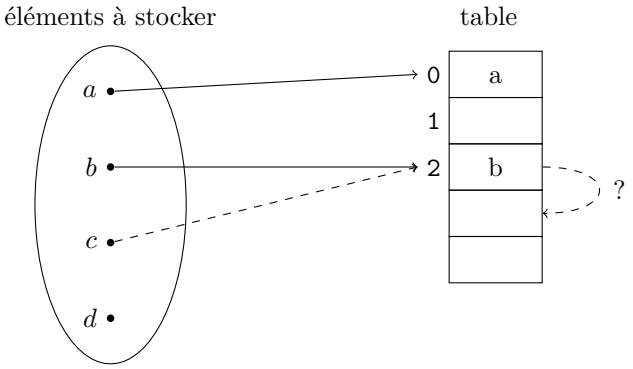
```
>>> hash(2023)
2023

>>> hash('PSI1')
-4814400283070324320

>>> hash((-1,3))
-3713082714465710744

# les listes (mutables) ne sont pas hachables
>>> hash([1,2])
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Utiliser directement cette fonction pour calculer l'indice n'est donc pas possible car la taille du tableau à manipuler serait trop grande. Si la taille du tableau est N, on peut associer à un objet l'indice `hash(obj)%N` par exemple, de façon à rester dans les indices existants.



2. Collisions

Le nombre d'objets que l'on peut être amené à considérer étant infini (il existe déjà une infinité de chaînes de caractères possibles par exemple), une telle fonction ne peut pas être injective : avec $N = 4$ par exemple, `hash(1)%4` et `hash(5)%4` renvoient `1`. On pourrait penser augmenter la valeur de N mais cela ne règlerait de toute façon pas le problème (la fonction `hash` elle même n'est pas injective). Si on souhaite stocker deux éléments auxquels la fonction de hachage associe le même entier, il devraient se situer au même endroit, on a affaire à une *collision*.



Il existe plusieurs façons de gérer ces collisions :

- on peut choisir de mettre les différents objets qui entrent en collision dans une liste ; cette solution (*adressage ouvert*) augmente la place nécessaire pour mémoriser le tableau en mémoire.
- on peut choisir le mettre l'objet qui entre en collision dans une autre case libre du tableau ; la taille du tableau reste alors la même (*adressage fermé*). La recherche de ce nouvel emplacement (*sondage*) peut se faire de différentes façons :
  - on peut sonder les cases les unes après les autres : si la case de collision est h, on teste $(h+1)\%N$, $(h+2)\%N$,...Cette méthode a pour effet de remplir des plages de cases consécutives du tableau (qui vont rendre les sondages suivants plus longs)
  - on peut sonder les cases dans l'ordre h, $(h+3)\%N$, $(h+6)\%N$, ..., $(h+3i)\%N$,... ou h, $h+1^2$, $h+2^2$,..., $h+i^2$,...
  - faire un sondage plus aléatoire dans les autres cases du tableau
  - ...

Reste un problème à gérer : la taille du tableau. Pour qu'il soit toujours possible de trouver une place par sondage, il faut s'assurer que le tableau soit assez grand. De même, plus le taux de remplissage du tableau (quotient du nombre de cases occupées par la taille du tableau) est grand, plus la probabilité de rencontrer une collision est grande. Pour contourner ces problèmes, lorsque le taux de remplissage du tableau devient trop grand, on augment automatiquement la taille du tableau ; en Python, la taille prévue pour un dictionnaire double lorsque le taux de remplissage dépasse 2/3.

3. Une implémentation simpliste

Une implémentation d'une table de hachage à 8 éléments avec des fonctions simplifiées :
Création d'une table : on initialise une liste de longueur 8, remplie de `None` (en supposant que les éléments que l'on aura à stocker ne valent jamais `None`)

```
N = 8

def creer():
    return [None]*N
```

Pour ajouter un élément à la table (en supposant qu'elle ne sera jamais pleine) :

```
def ajouter(elt,dic) :
    h = hash(elt)%N
    i = h
    while dic[i] != None : # sondage de toutes les cases qui suivent
        i = (i+1)%N
    dic[i] = elt
```

Pour chercher si un élément est dans la table (toujours en supposant que la table n'est pas pleine) :

```
def chercher(elt,dic) :
    h = hash(elt)%N
    trouve = False
    i = h
    while not trouve and dic[i] != None : # le sondage est case après case
       donc si on arrive sur None, c'est fini
        trouve = (dic[i]==elt)
        i = (i+1)%N
    return trouve
```

Si on souhaite rajouter une fonction pour supprimer un élément, les choses se compliquent : si on se contente de remettre `None`, la fonction de recherche ne marchera plus. Si dans une table de longueur 8, on place les éléments 1, 9 et 17, la table sera `[None,1,9,17,None,None,None,None]` ; puis si on supprime 9, `[None,1,None,17,None,None,None,None]`. Si maintenant on cherche si 17 est dans la table, on teste la cas `17%8=1` et comme la suivante contient `None`, la boucle s'interrompt sans trouver 17. Une solution est de mettre dans la table une autre valeur (`'vide'`), qui ne devra pas être parmi celles que l'on cherchera à stocker, à la place des éléments supprimés ; cela nécessite aussi de modifier les sondages en cas de collision puisque les cases vides peuvent contenir soit `None`, soit `'vide'`. Une solution permettant de supprimer des éléments et de gérer le redimensionnement du tableau est la suivante : on commence par rajouter une case qui permettra de stocker le nombre d'éléments présents dans la tableau afin de déterminer le taux de remplissage

```
N = 8

def creer() :
    return [None]*N+[0]

def ajouter(elt,dic) :
    N = len(dic)-1
    if dic[-1]/N > 0.5 :
        redimensionner(dic)
        N = len(dic)-1
    h = hash(elt)%N
    i = h
    while dic[i] not in [None,'vide'] :
        i = (i+1)%N
    dic[i] = elt
    dic[-1] += 1
    print(dic)

def chercher(elt,dic) :
    N = len(dic)-1
    h = hash(elt)%N
    trouve = False
    i = h
    while not trouve and dic[i] != None :
        trouve = (dic[i]==elt)
        i = (i+1)%N
    return trouve
```

```
def supprimer(elt,dic) :
    N = len(dic)-1
    h = hash(elt)%N
    trouve = False
    i = h
    while not trouve and dic[i] != None :
        trouve = (dic[i]==elt)
        i = (i+1)%N
    dic[(i-1)%N] = 'vide'
    dic[-1] -= 1
    print(dic)

def redimensionner(dic) :
    N = len(dic)-1
    nb = dic.pop()
    d = []
    for i in range(N) :
        k = dic[i]
        if k not in [None,'vide'] :
            d.append(k)
        dic[i] = None
        dic.append(None)
    dic.append(0)
    for k in d :
        ajouter(k,dic)
```

II Les dictionnaires en Python

Les dictionnaires sont, en Python, des objets importants : Python crée des dictionnaires en arrière plan chaque fois que l'on définit une variable, que l'on exécute une fonction,...

1. Création d'un dictionnaire

Un dictionnaire peut être vu comme un ensemble (non ordonné) de couples du type :

<clé> :<valeur>

on accède à une valeur par l'intermédiaire de sa clé : la clé remplace plus ou moins l'indice dans l'utilisation d'une liste. Il existe plusieurs façons de créer un dictionnaire

```
>>> d = {} # dictionnaire vide

>>> d = {'un': 1, 'deux': 2, 'trois': 3} # par la liste de ses éléments

>>> d = {str(k)*k:k for k in range(4)} # par compréhension
>>> d
{'': 0, '1': 1, '22': 2, '333': 3} # même une chaîne vide peut être une clé
```

Les clés d'un dictionnaires peuvent être n'importe quel objet hachable : une chaîne de caractère, un nombre, un tuple mais pas une liste ou un objet qui contiendrait une liste (ou tout autre objet mutable).

```
>>> d = {[1]:'un'}
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Les valeurs peuvent être n'importe quoi : listes, chaînes de caractères, nombres, listes de tuples, ou même un autre dictionnaire,...

2. Manipulation d'un dictionnaire

On accède à une valeur en utilisant sa clé :

```
>>> d = {'un': 1, 'deux': 2, 'trois': 3}
>>> d['deux']
2

>>> d['cinq'] # ce n'est pas une clé
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError: 'cinq'
```

Un dictionnaire est mutable (donc non hachable et donc non utilisable en tant que clé dans un autre dictionnaire) : on peut modifier la valeur associée à une clé

```
>>> d['trois'] = 4
>>> d
{'un': 1, 'deux': 2, 'trois': 4}
```

La même syntaxe permet de rajouter une clé si elle n'existe pas :

```
>>> d['quatre'] = 4
>>> d
{'un': 1, 'deux': 2, 'trois': 4, 'quatre': 4}
```

La fonction `len` renvoie le nombre de couples (clé :valeur) du dictionnaire (donc peut permettre de tester si un dictionnaire est vide).

```
>>> len(d)
4
>>> len(d) == 0 # le test d == {} fonctionne aussi
False
```

On peut récupérer les clés, les valeurs ou les couples (clé,valeurs) avec les méthodes `.keys()`, `.values()` et `.items()` :

```
>>> d.keys()
dict_keys(['un', 'deux', 'trois', 'quatre'])

>>> list(d.keys()) # on peut ensuite en faire une liste
['un', 'deux', 'trois', 'quatre']

>>> d.values()
dict_values([1, 2, 4, 4])

>>> d.items()
dict_items([('un', 1), ('deux', 2), ('trois', 4), ('quatre', 4)])
```

On peut tester l'appartenance d'une clé à un dictionnaire avec `in` :

```
>>> 'un' in d
True

>>> 1 in d
False

>>> 1 in d.values()
True
```

Un dictionnaire est itérable : l'itération se fait sur les clés

```
>>> for c in d :
        print(d[c])
1
2
4
4

>>> for c in d :
        d[c*2] = 1 # rajouterait les clés 'unun', 'deuxdeux' ...
```

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
# on ne peut pas modifier la taille du dictionnaire sur lequel se fait l'ité
    ration
```

On peut supprimer un élément avec la commande `del` :

```
>>> del(d['un'])
>>> d
{'deux': 2, 'trois': 4, 'quatre': 4}
```

Un dictionnaire étant mutable, il pose les mêmes problèmes que les listes pour la copie :

```
>>> d1 = {'un':[1],'deux':[2]}
>>> d2 = d1
>>> d2
{'un': [1], 'deux': [2]}
>>> d1['un'] = 1
>>> d2
{'un': 1, 'deux': [2]} # modifié aussi !


>>> d3 = d1.copy()
>>> d1['un'] = 'one'
>>> d3
{'un': 1, 'deux': [2]} # mieux !
>>> d1['deux'][0] = 3
>>> d3
{'un': 1, 'deux': [3]} # mais pas en profondeur
```