Structure de dictionnaire

SPE - Informatique

Anthony Lick
Lycée Janson de Sailly

Introduction

Les dictionnaires

Listes Python

Lorsqu'on veut stocker plusieurs choses en Python, on peut utiliser des **listes** :

- la **liste vide** est notée `[]` ;
- la **taille** d'une liste `L` est obtenue via `len(L)` ;
- si `L` est une liste de taille n, et si $0 \leq i \leq n-1$, le i-ème élément de `L` est obtenu via `L[i]` ;
- on peut rajouter un nouvel élément `x` dans `L` via `L.append(x)`.

Les dictionnaires

Listes Python

Lorsqu'on veut stocker plusieurs choses en Python, on peut utiliser des **listes** :

- la **liste vide** est notée `[]` ;
- la **taille** d'une liste `L` est obtenue via `len(L)` ;
- si `L` est une liste de taille n, et si $0 \leq i \leq n-1$, le i-ème élément de `L` est obtenu via `L[i]` ;
- on peut rajouter un nouvel élément `x` dans `L` via `L.append(x)`.

Limitations des listes

L'inconvénient des listes Python est qu'on ne peut pas choisir n'importe quel i pour indicer un nouvel élément de `L` : i est forcément un **entier**, et l'ensemble des indices d'une liste de taille n doit **exactement** être $[\![0, n-1]\!]$.

Les dictionnaires

Une autre structure permet de stocker des éléments tout en ayant plus de souplesse sur les “indices” : les **dictionnaires**.

Dans une structure de dictionnaire, chaque valeur stockée est associée à une **clé** (l’équivalent des indices pour les listes).

L’intérêt des dictionnaires est que les clés peuvent ne pas être des **entiers**, ni être à valeurs **contigües**.

Les dictionnaires

Si l'on dispose d'une structure de dictionnaire, on peut effectuer les opérations suivantes (avec une bonne complexité) :

- **créer** un **dictionnaire vide** ;
- **tester** si un dictionnaire **est vide** ;
- **tester** si une clé k est **présente** dans le dictionnaire ;
- **trouver** l'élément e **associé** à une clé k dans le dictionnaire ;
- **ajouter** une association **clé-valeur** (k, e) si k n'est pas encore présente dans le dictionnaire ;
- **supprimer** l'entrée (k, e) si k est présente dans le dictionnaire ;
- **modifier** la valeur e associée à une clé k présente dans le dictionnaire.

Les dictionnaires en Python

Les dictionnaires en Python

Python

En Python, le **dictionnaire vide** est noté : `{}`.

Si la variable `dico` contient un dictionnaire :

- on peut obtenir sa **taille** via : `len(dico)` ;
- on peut **tester** s'il **contient** une valeur pour la clé k via : `if k in dico` ;
- si `dico` contient la clé k, on peut obtenir la valeur qui lui est associée via : `dico[k]` ;
- si la clé k est présente dans `dico`, on peut modifier sa valeur associée via : `dico[k] = e` ;
- si la clé k **n'**est **pas** présente dans dico, on peut **ajouter** l'**association clé-valeur** (k, e) via : `dico[k] = e` ;
- on peut **supprimer** une clé k via : `del dico[k]`.

Les dictionnaires en Python

Boucle `for`

On peut parcourir toutes les clés d'un dictionnaire à l'aide d'une boucle **`for`**. Contrairement aux listes, aucune garantie ne peut être donnée sur l'ordre dans lequel les clés vont être énumérées.

```
1 def affiche_cles(dico):
2     for k in dico:
3         print(k)
```

```
1 def affiche_valeurs(dico):
2     for k in dico:
3         print(dico[k])
```

Exemple

Les fonctions ci-dessus affichent toutes les clés (resp. valeurs) stockées dans le dictionnaire pris en entrée.

Les dictionnaires en Python

```
1  dico = {k1:e1, k2:e2, ..., kn:en}
```

Initialisation

On peut initialiser un dictionnaire non vide, contenant les associations clés-valeurs $(k_1, e_1), (k_2, e_2), \ldots, (k_n, e_n)$ avec la syntaxe ci-dessus.

Les dictionnaires en Python

Exemple

```python
ages = {"Alice":18, "Bob":20, "Nadine":12}

ages["Charlie"] = 42 # ajout d'une nouvelle entrée
ages["Bob"] = 21 # modification d'une entrée

def moyenne(dico):
    s = 0
    for k in dico:
        s += dico[k]
    return s / len(dico)
```

Console

```
>>> ages
{'Alice': 18, 'Bob': 21, 'Nadine': 12, 'Charlie': 42}
>>> ages["Nadine"]
12
>>> moyenne(ages)
23.25
>>> ages["nadine"] # Attention aux majuscules !
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-32-1505eaa61367> in <module>
----> 1 ages["nadine"]

KeyError: 'nadine'
```

Les dictionnaires en Python

```
1  dico2 = dico.copy()
```

Copie

On peut effectuer une **copie** d'un dictionnaire via l'instruction ci-dessus.

Attention

Si vous n'effectuez pas de copie, les deux dictionnaires seront **liés** dans la mémoire, comme pour les listes !

Les dictionnaires en Python

Exemple

```
1  dico = {"Alice":18, "Bob":20, "Nadine":12}
2
3  dico2 = dico # les deux dictionnaires sont liés dans la mémoire
4
5  dico["Charlie"] = 42 # va aussi rajouter l'entrée dans dico2
6  dico2["Bob"] = 0     # va aussi modifier l'entrée de dico
```

Console

```
>>> dico
{'Alice': 18, 'Bob': 0, 'Nadine': 12, 'Charlie': 42}
>>> dico2
{'Alice': 18, 'Bob': 0, 'Nadine': 12, 'Charlie': 42}
```

Exemple

Sans la copie, chaque modification de **dico2** modifie **dico** et inversement.

Les dictionnaires en Python

Exemple

```
1  dico = {"Alice":18, "Bob":20, "Nadine":12}
2
3  dico2 = dico.copy() # les deux dictionnaires ne sont pas liés dans la mémoire
4
5  dico["Charlie"] = 42 # ne modifie pas dico2
6  dico2["Bob"] = 0     # ne modifie pas dico
```

Console

```
>>> dico
{'Alice': 18, 'Bob': 20, 'Nadine': 12, 'Charlie': 42}
>>> dico2
{'Alice': 18, 'Bob': 0, 'Nadine': 12}
```

Exemple

Avec la copie, chaque modification d'un des dictionnaires ne modifie pas l'autre.

Comment implémenter une structure de dictionnaire ?

Comment implémenter une structure de dictionnaire ?

Implémentation efficace

Dans cette partie, on se demande comment implémenter efficacement une structure de dictionnaire.

Comment implémenter une structure de dictionnaire ?

Implémentation efficace

Dans cette partie, on se demande comment implémenter efficacement une structure de dictionnaire.

Arbres binaires de recherche

Il existe une implémentation utilisant des structures d'arbres, mais les arbres ne sont pas au programme de l'ITC.

Listes

On va voir comment implémenter des dictionnaires à l'aide de listes.

Implémentation de dictionnaires à l'aide de listes

Implémentation naïve

On pourrait simplement stocker les associations clés-valeurs dans une liste des couples (k, e).

Problème : la plupart des opérations se feraient en temps linéaire sur le nombre de clés, ce qui n'est pas satisfaisant...

Tables de hachage

Formalisation du problème

Soit K un ensemble de clés, et E un ensemble de valeurs.

On souhaite trouver un moyen efficace de stocker des associations clés-valeurs, i.e. des couples de $K \times E$ tel que chercher une clé $k \in K$ parmi les associations soit plus efficace que l'approche naïve.

Fonction de hachage

Une **fonction de hachage** est une fonction $h : K \to \mathbb{N}$, qui va permettre de répartir les clés dans les cases d'une liste.

Tables de hachage

Alvéole

Soit h une fonction de hachage de $K \to \mathbb{N}$, soit $n \in \mathbb{N}^*$.

Soit $\mathcal{R}$ la relation binaire sur $K \times E$ définie par :

$$(k, e)\ \mathcal{R}\ (k', e') \iff h(k) \equiv h(k') \mod n$$

On vérifie facilement que $\mathcal{R}$ est une **relation d'équivalence**.

Une **alvéole** est une liste L de couples $(k, e) \in K \times E$ appartenant tous à la même classe d'équivalence, i.e. :

$$\exists i \in [\![0, n-1]\!],\ \forall (k, e) \in L,\ h(k) \equiv i \mod n$$

Lorsque la valeur i est connue, on dit que L est une **alvéole pour la valeur** i.

Tables de hachage

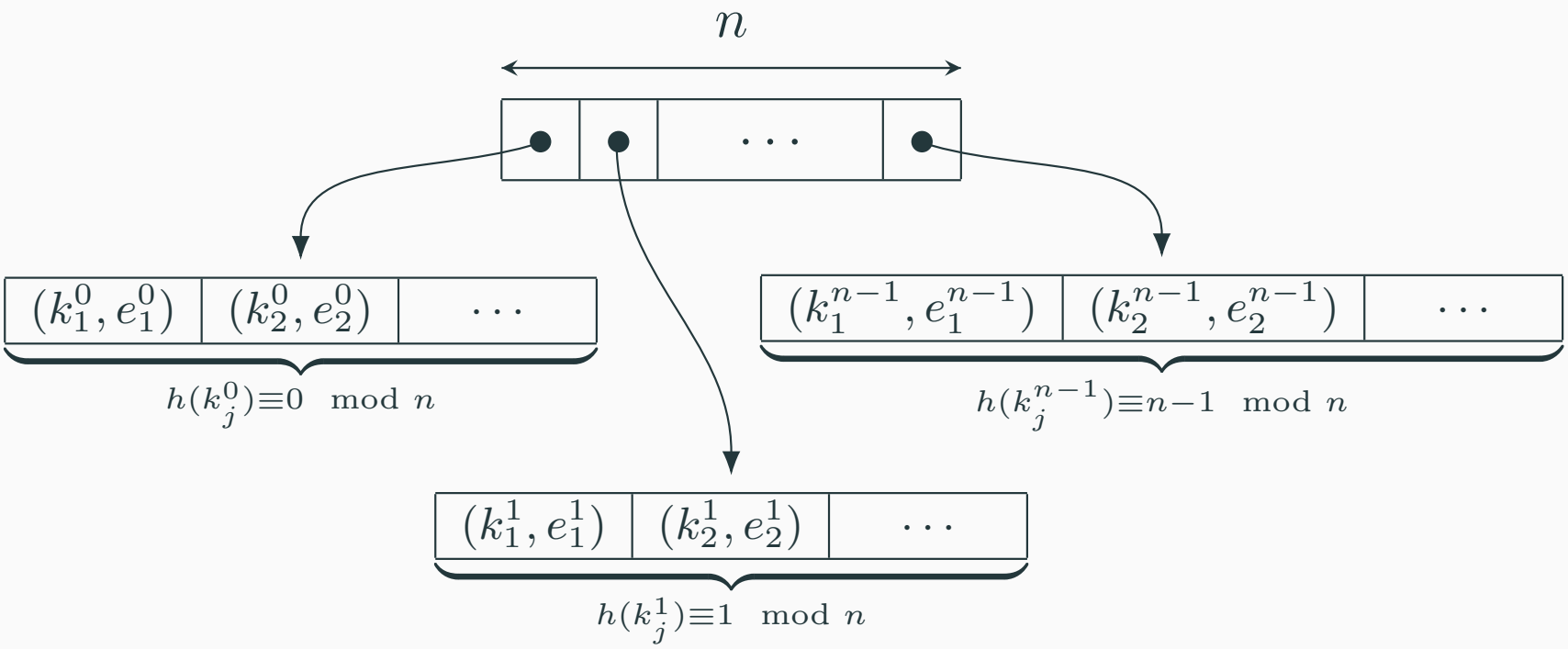
Table de hachage

Soit K un ensemble de clés, et E un ensemble de valeurs.
Soit h une fonction de hachage de $K \to \mathbb{N}$, soit $n \in \mathbb{N}^*$.

Une **table de hachage** pour la fonction de hachage h et la valeur n est une **liste** de taille n telle que :

$$\forall i \in [\![0, n-1]\!],\ L[i] \text{ est une } \textbf{alvéole} \text{ pour la valeur } i.$$

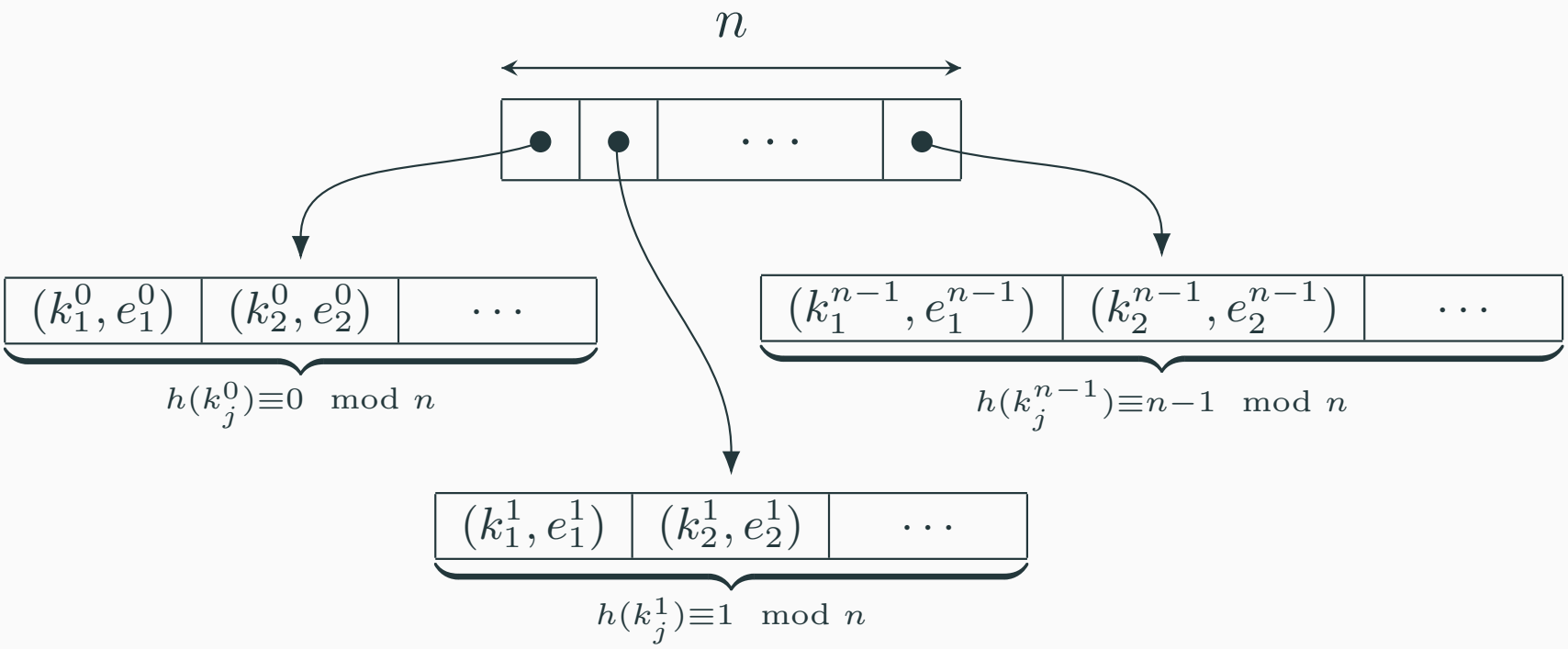
Tables de hachage



Principe des tables de hachage

Autrement dit, pour stocker une **association clé-valeur** (k, e) dans la **table de hachage**, on va calculer $h(k) \mod n$, et la stocker dans l'**alvéole** correspondante.

Tables de hachage



Principe des tables de hachage

Ainsi, pour **chercher** une clé k dans la table de hachage, il n'y a pas besoin de parcourir toutes les clés de la table, mais seulement celles de la "**bonne**" **alvéole**.

Tables de hachage

Complexité

En suivant ce principe, chaque opération sur les tables de hachages se fera en temps linéaire en la taille de la plus grande alvéole.

En pratique, on cherche à avoir une fonction de hachage permettant de bien répartir les clés dans chacune des alvéoles en moyenne.

Pour cela, il faut éviter d'avoir une fonction de hachage permettant de facilement produire des **collisions**.

Tables de hachage

Collision

Soit h une fonction de hachage et $n \in \mathbb{N}^*$.

On dit que deux clés $(k, k') \in K^2$ sont en **collision** si

$$h(k) \equiv h(k') \mod n.$$

Sécurité

Si un utilisateur malveillant trouve un moyen de générer des collisions automatiquement pour une certaine fonction de hachage, il pourrait s'en servir pour mener une attaque contre des services utilisant cette fonction de hachage.

Tables de hachage dynamiques

Si jamais certaines alvéoles deviennent trop remplies, on peut choisir de désormais travailler avec un $n' > n$.

Il faut alors répartir toutes les associations clés-valeurs dans une nouvelle table de hachage possédant n' alvéoles.

Si on choisit un n' suffisamment grand par rapport à n (par exemple $n' = 2n + 1$), ce sera rentable sur le long terme (on parle de **complexité amortie**).

Tables de hachage

Implémentation

Afin d'implémenter proprement toutes les idées discutées précédemment, nous allons utiliser une notion qui n'est pas au programme d'ITC mais qui est utilisée par tous les programmeurs Python dans la vraie vie : la **programmation orientée objet** (**POO**).

Introduction à la programmation orientée objet

Programmation orientée objet

POO

La **programmation orientée objet (POO)** permet d'implémenter des classes abstraites.

Exemple

Les listes Python sont des objets, de la classe "`list`".

```
1  >>> L=[0,1,2]
2  >>> type(L)
3  <class 'list'>
```

Programmation orientée objet

Objets et méthodes

Une **classe** regroupe les fonctions et les attributs définissant un **objet**. Les fonctions associées à une classe sont appelées des **méthodes**.

Exemple

La classe `list` de Python possède les méthodes `pop` et `append`.

On va maintenant voir comment créer nous-mêmes des classes et définir des méthodes sur les objets de la classe.

Exemple

```
1   class personne:
2       def __init__(self,n,p):
3           self.nom=n
4           self.prenom=p
5           self.mail=None
6
7       def fixe_mail(self,s):
8           self.mail=s
9
10      def nom_personne(self):
11          return self.nom
12
13      def prenom_personne(self):
14          return self.prenom
```

Attributs

- On définit ici une classe **personne** ayant 3 attributs : **nom**, **prenom** et **mail**.
- Si **a** est un objet de cette classe, on accède à ses attributs à l'aide de **a.nom**, **a.prenom** et **a.mail**.
- La méthode `__init__` est spéciale : elle est appelée lors de la création d'un objet de la classe.

Exemple

```
1   class personne:
2       def __init__(self,n,p):
3           self.nom=n
4           self.prenom=p
5           self.mail=None
6
7       def fixe_mail(self,s):
8           self.mail=s
9
10      def nom_personne(self):
11          return self.nom
12
13      def prenom_personne(self):
14          return self.prenom
```

Méthodes

- Les méthodes **nom_personne** et **prenom_personne** permettent d'accéder à deux de ces attributs.
- La méthode **fixe_mail** permet de modifier l'attribut **mail**.
- Toutes les méthodes ont en paramètre une variable `self` qui fait référence à l'objet sur lequel on travaille.

Exemple

```
1  class personne:
2      def __init__(self,n,p):
3          self.nom=n
4          self.prenom=p
5          self.mail=None
6
7      def fixe_mail(self,s):
8          self.mail=s
9
10     def nom_personne(self):
11         return self.nom
12
13     def prenom_personne(self):
14         return self.prenom
```

```
1  >>> a=personne("Dupont","Martin")
2  >>> a.prenom
3  'Martin'
4  >>> a.fixe_mail("truc@bidule.machin")
5  >>> a.mail
6  'truc@bidule.machin'
7  >>> a.nom_personne()
8  'Dupont'
```

Méthodes magiques de Python

Méthodes magiques

La méthode `__init__` est ce qu'on appelle une **méthode magique** de Python : elle est appelée automatiquement lorsqu'on crée un objet.

Il existe d'autres méthodes magiques (qui sont toutes encadrées par des "__") :

- `obj.__len__()` est appelée lorsqu'on fait `len(obj)` ;
- `obj.__contains__(k)` est appelée lorsqu'on fait `k in obj` ;
- `obj.__getitem__(k)` est appelée lorsqu'on fait `obj[k]` ;
- `obj.__setitem__(k,e)` est appelée lorsqu'on fait `obj[k] = e` ;
- `obj.__delitem__(k)` est appelée lorsqu'on fait `del obj[k]` ;
- ...

Implémentation d'une table de hachage

```python
class hashtbl:
    def __init__(self,h,n):
        self.h = h
        self.n = n
        self.L = [[] for _ in range(n)]
        self.size = 0

    def __getitem__(self,k):
        i = self.h(k) % self.n
        for (k2,e) in self.L[i]:
            if k == k2:
                return e
        # Message d'erreur
        raise Exception("clé non présente")

    def __setitem__(self,k,e):
        i = self.h(k) % self.n
        alveole = self.L[i]
        for j in range(len(alveole)):
            (k2,e2) = alveole[j]
            if k == k2:
                alveole[j] = (k,e)
                return
        # si k n'est pas dans l'alveole
        alveole.append((k,e))
        self.size += 1

#...
```

```python
#...
    def __delitem__(self,k):
        i = self.h(k) % self.n
        alveole = self.L[i]
        for j in range(len(alveole)):
            (k2,e2) = alveole[j]
            if k == k2:
                alveole.pop(j)
                self.size -= 1
                return

    def __contains__(self,k):
        i = self.h(k) % self.n
        for (k2,_) in self.L[i]:
            if k == k2:
                return True
        return False

    def __len__(self):
        return self.size

    def cles(self):
        # renvoie la liste des clés
        L_cles = []
        for alveole in self.L:
            for (k,_) in alveole:
                L_cles.append(k)
        return L_cles
```

Implémentation d'une table de hachage

Exemple

```
1  def h(k):
2      return k
3
4  H = hashtbl(h,5)
5
6  for k in range(10):
7      H[k] = 2*k
```

Console

```
>>> len(H)
10
>>> H.L
[[(0, 0), (5, 10)],
 [(1, 2), (6, 12)],
 [(2, 4), (7, 14)],
 [(3, 6), (8, 16)],
 [(4, 8), (9, 18)]]
>>> H[12] = 42
>>> H.L
[[(0, 0), (5, 10)],
 [(1, 2), (6, 12)],
 [(2, 4), (7, 14), (12, 42)],
 [(3, 6), (8, 16)],
 [(4, 8), (9, 18)]]
>>> H[5] = 666
>>> H[9], H[5]
(18, 666)
>>> len(H)
11
```

Héritage

```
1  class ClasseFille(ClasseMere):
```

Héritage

Outre le fait de pouvoir mieux organiser son code, le gros point fort de la programmation orientée objet est la notion d'**héritage**.

Une fois une classe `ClasseMere` créée, on peut créer une classe `ClasseFille` et indiquer que cette nouvelle classe doit posséder tous les attributs et toutes les méthodes de la classe `ClasseMere` : on dit que `ClasseFille` **hérite** (des attributs et méthodes) de `ClasseMere`.

Pour cela, il suffit de commencer la déclaration de la classe `ClasseFille` comme ci-dessus.

Implémentation de tables de hachage dynamiques

```
1  class hashtbl_dyna(hashtbl):
2      def augmenter_n(self):
3          old_L = self.L
4          self.n = 2 * self.n + 1
5          self.L = [[] for _ in range(self.n)]
6          self.size = 0 # attention !
7          for alveole in old_L:
8              for (k,e) in alveole:
9                  self.__setitem__(k,e) # incrémente self.size
```

Tables de hachage dynamiques

Pour pouvoir augmenter la valeur de n automatiquement via un appel de méthode, on peut tout simplement créer une nouvelle classe qui va hériter de la classe `hashtbl` définie précédemment, et lui rajouter une nouvelle méthode.

Remarque

Nous n'en avons pas besoin ici, mais si jamais l'une des méthodes héritées ne convient pas pour notre nouvelle classe, il est tout à fait possible de la redéfinir : la nouvelle méthode va alors écraser l'ancienne pour la nouvelle classe.

Implémentation de tables de hachage dynamiques

Exemple

```
1  def h(k):
2      return k
3
4  H = hashtbl_dyna(h,5)
5
6  for k in range(10):
7      H[k] = 2*k
```

Console

```
>>> len(H)
12
>>> H.L
[[(0, 0), (5, 10), (10, 20)],
 [(1, 2), (6, 12), (11, 22)],
 [(2, 4), (7, 14)],
 [(3, 6), (8, 16)],
 [(4, 8), (9, 18)]]
>>> H.augmenter_n()
>>> len(H)
12
>>> H.L
[[(0, 0), (11, 22)],
 [(1, 2)],
 [(2, 4)],
 [(3, 6)],
 [(4, 8)],
 [(5, 10)],
 [(6, 12)],
 [(7, 14)],
 [(8, 16)],
 [(9, 18)],
 [(10, 20)]]
```

Implémentation de fonctions de hachage

Fonctions de hachage

Par soucis de simplicité, dans les exemples précédents, nous avons choisi $h = id_{\mathbb{N}}$.

Si nous voulions travailler avec des clés non entières, il faudrait alors écrire une fonction de hachage fonctionnant avec le type désiré. Nous verrons en TP quelques fonctions de hachage naïves, i.e. qui n'ont aucune garantie de nous protéger contre des collisions.

L'élaboration d'une "bonne" fonction de hachage n'est pas au programme de CPGE. Si vous souhaitez utiliser une meilleure fonction de hachage, Python en possède déjà une : la fonction `hash`.