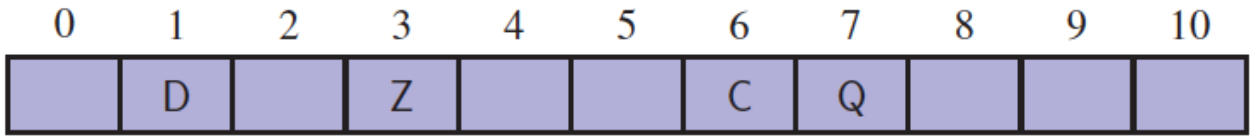
Map dans un tableau
Fonctions et tables de hachage
Codes de hachage
Fonction de compression
Gestion des collisions par chaînage et par sondage
Hachage double
Performances du hachage
Implémentation du *dict* dans Python

Map dans un tableau

- *Map* prend en charge l'abstraction de l'utilisation de clés en tant qu'indices (comme dans un tableau) avec la syntaxe M[k]
- Comme échauffement mental, considérez un cadre restreint dans lequel une *Map* de *n* éléments utilise des clés entières comprises entre 0 et N - 1, pour un certain N ≥ n, et une *Map* contenant 4 éléments (clé, valeur) :

d = { 1: 'D', 3: 'Z', 6: 'C', 7: 'Q' }



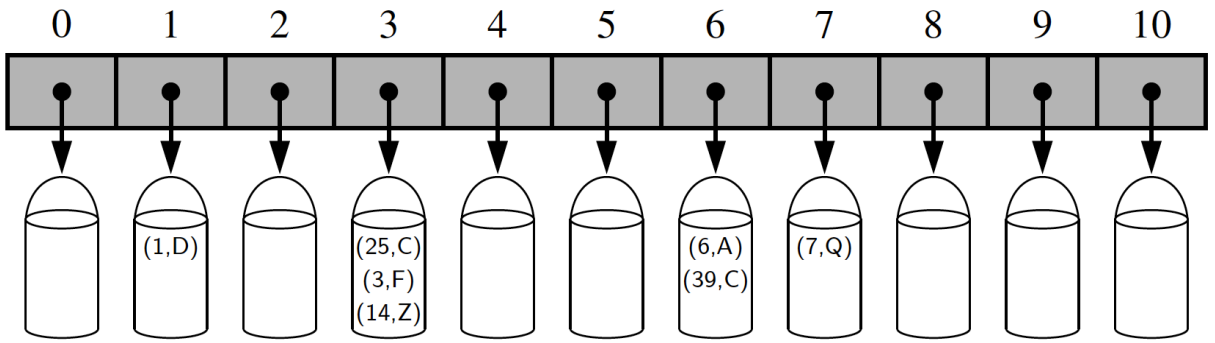
Généralisation

Il y a deux défis à l'extension de ce cadre au contexte plus général d'une *Map* :

Premièrement, nous ne souhaitons peut-être pas consacrer un tableau de longueur N lorsque N >> n.

Deuxièmement, nous n'exigeons généralement pas que les clés d'une *Map* soient des entiers.

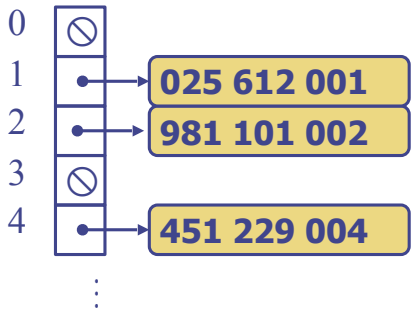
- Le concept novateur est l'utilisation d'une fonction de *hachage* pour mapper les clés à des indices de la table ;
- Idéalement, les clés sont bien réparties dans l'intervalle de 0 à N -1 ;
- Cependant, en pratique on peut avoir plus d'une clés distinctes qui "mappent" le même indice ;
- Conséquemment, conceptualisons la table comme un tableau de contenants pouvant gérer une collection d'éléments.



Que faire si les clés ne sont pas des entiers ?

Utilisez une *fonction de hachage* pour mapper les clés générales aux indices correspondants dans le tableau

Par exemple, les 3 derniers chiffres d'un numéro d'assurance sociale



Fonctions et tables de hachage

- Une *fonction de hachage*, h, assigne les clés d'un type donné à des entiers dans un intervalle fixe [0, N - 1]
- Exemple: h(x) = x mod N est une fonction de hachage pour les clés entières
- L'entier h(x) est appelé la valeur de hachage de la clé x

- Une *table de hachage* pour un type de clé donné consiste en :
  - Une *fonction de hachage* h
  - Un *tableau* (ou table) de taille N
- Lors de l'implémentation de *Map* avec une *table de hachage*, le but est de stocker l'item (k, o) à l'index i = h(k)

Exemple avec les numéros d'assurance sociale

- Nous concevons une table de hachage pour une *Map* stockant des entrées (NAS, Citoyen), où NAS est le numéro d'assurance sociale associé à un citoyen, soit, au Canada, un nombre entier positif à 9 chiffres
- Notre table de hachage utilise un tableau de taille N = 1000 et la fonction de hachage

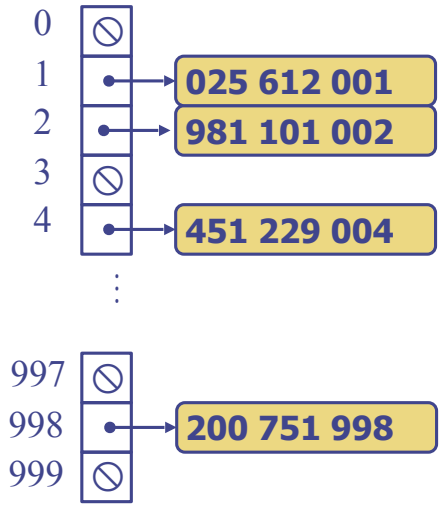
  h(x) = les 3 derniers chiffres de x

- Avec cet exemple, on peut voir que deux NAS différents peuvent pointer vers la même entrée dans la table, provoquant une *collision* !

  h(941 117 004) = 4 !!!

➡ Une table de 1000 entrées pour des millions de citoyens, ça marche pas !

Nous verrons plus loin comment gérer le problème des collisions !



Fonctions de hachage

Une fonction de hachage est généralement spécifiée comme la composition de deux fonctions :

Code de hachage :

h_1 : pour passer des clés à des nombres entiers

Fonction de compression :

h_2: pour passer des entiers (de h_1) à un indice dans la table, [0, N - 1]

Le code de hachage est appliqué en premier et la fonction de compression est appliquée sur le résultat, c'est-à-dire,

$h(x) = h_2(h_1(x))$

Le but de la fonction de hachage est de "disperser" les clés de manière apparemment aléatoire

Options pour des codes de hachage

- Adresse mémoire :
  - Nous réinterprétons l'adresse mémoire de l'objet clé sous la forme d'un entier
- Représentation binaire :
  - Nous réinterprétons les bits de la clé comme un nombre entier
- Une somme de composantes :
  - Nous partitionnons les bits de la clé en composantes de longueur fixe (par exemple de 16 ou 32 bits) et nous additionnons les composantes
- Une somme polynomiale des composantes :
  - Nous partitionnons les bits de la clé en une séquence de composantes de longueur fixe (par exemple de 8, 16 ou 32 bits)

    $x_0\ x_1\ \ldots\ x_{n-1}$
  - Nous évaluons le polynôme

    $p(a) = x_0\, a^{n-1} + x_1\, a^{n-2} + \ldots x_{n-2}\, a + x_{n-1}$

    à une valeur fixe a, en ignorant les débordements

    ➡ Cette option est particulièrement adaptée aux chaînes de caractères. Par exemple, en choisissant $a = 33$ on aura au plus 6 collisions sur un ensemble de 50 000 mots (anglais)

Fonctions de compression

- Division :
  - $h_2(x) = x \bmod N$
  - La taille, N, de la table de hachage est habituellement choisie en fonction du nombre d'éléments à y insérer (sera précisé plus loin)
- Multiplier, Ajouter et Diviser (MAD) :
  - $h_2(x) = (ax + b) \bmod p \bmod N$
  - a et b sont des entiers non négatifs tels que $a \bmod p \neq 0$, sinon chaque entier serait mappé à la valeur b !
  - on prend p nombre premier

Classe `Compression`

```
from random import randrange

#class Compression pour tables de hachage
class Compression:

    #constructeur de compresseur selon la taille et un nombre premier p
    #avec 2 options pour la fonction de compression: division et MAD
    def __init__( self, size = 11, p = 109345121 ): # puissance de 2 à tester 134217728
        self._size = size
        self._p = p
        #pour MAD on détermine une échelle et un décalage
        #on trouve un entier multiplicateur entre 1 et p-2
        #qui n'est pas un multipe de p
        trouve = False
        while not trouve:
            self._scale = 1 + randrange( self._p - 1 )
            if not ( self._scale % self._p ) == 0:
                trouve = True
        #on trouve un entier décalage entre 0 et p-1
        self._shift = randrange( self._p )

    #option division, on prend le modulo avec la taille
    def divide( self, hcode ):
        return hcode % self._size

    #option MAD, on prend (hcode * scale + shift) % p % taille
    def mad( self, hcode ):
        return ( hcode * self._scale + self._shift ) % self._p % self._size
```

Unit testing de la Classe `Compression`

(pour essayer les 2 modes et des valeurs de `size` et `p` avec une distribution uniforme)

```
#unit testing
if __name__ == '__main__':

    print( "Compression unit testing..." )

    h = 16384 # nombre premier à tester 16411
    #on crée un compresseur pour h entrées
    compressor = Compression( h )
    #on rempli la table à 80%
    n = int( h * 0.8 )
    #on simule k fois
    k = 500
    #initialisation d'un compteur de collisions
    collisions = 0
    #pour k fois
    for j in range( 0, k ):
        #initialisation d'un ensemble vide
        #pour imiter les entrées d'un table de hachage
        s = set()
        #pour 80% de la taille de la table
        for i in range( 0, n ):
            #on génère un hashcode au hasard entre 0 et 10*h - 1
            hc = randrange( h * 10 )
            #on compresse le hashcode dans la table
            hcc = compressor.mad( hc )
            #on vérifie s'il y a collision ou non
            if hcc in s:
                #si oui, on la compte
                collisions += 1
            else:
                #si non, on ajoute le hashcode compressé dans l'ensemble
                s.add( hcc )

    print( "Number of collisions on average for", n, "entries in a table of size", h, "is", collisions/k )

    print( "end unit testing..." )
```

Résultats de unit testing

(n ~ 2^{14} entrées ; k = 5000 ; distribution uniforme)

Méthode de compression	p	n	Collisions (%)	Collisions (%) (biais pour clés paires)
MAD	109345121	2^{14}	**25.0**	**25.1**
MAD	2^{27}	2^{14}	**25.0**	**67.8**
MAD	$2^{14} + 27$	2^{14}	**24.9**	**24.9**
MAD	$2^{14} + 27$	$2^{14} + 27$	**24.9**	**24.9**
MAD	109345121	$2^{14} + 27$	**25.0**	**25.1**
division	N/A	2^{14}	**24.9**	**28.6**
division	N/A	$2^{14} + 27$	**24.9**	**24.9**

Pourquoi utiliser des nombres premiers ?

Choisir une bonne fonction de hachage est de la plus haute importance. Une fonction de hachage uniforme est une fonction qui distribue de manière égale les clés sur toute la table de hachage. Si la fonction de hachage est mal choisie, les clés peuvent avoir tendance à s'agglomérer dans une ou des zones de la table, créant de plus en plus de collisions. Un modèle de dispersion non uniforme et un taux de collision élevé entraînent une dégradation des performances de la structure de données.

La fonction de hachage ne doit pas être biaisée en fonction du modèle spécifique de distribution des clés et de manière à pouvoir assurer une distribution égale des clés sur l'ensemble de la table de hachage.

Généralement, le calcul de k mod m est une opération relativement coûteuse. Cependant, si $m = 2^p$, l'opération consiste simplement à regarder les p bits de poids faibles, réduisant les valeurs possibles de la fonction de compression sur une fraction des bits de k.

```
from Map import Map
from random import randrange

#classe HashMap utilisant MAD
class HashMap( Map ):

    def __init__( self, cap = 11, p = 109345121 ):
        self._T = cap * [None]                  #table de hachage de cap entrées
        self._n = 0                             #nombre d'éléments dans la table
        self._prime = p                         #nombre premier pour la compression MAD
        #pour MAD on détermine une échelle et un décalage
        self._scale = 1 + randrange( p - 1 ) #scale entre 1 et p-2
        #on trouve un entier multiplicateur entre 1 et p-2
        #qui n'est pas un multipe de p
        trouve = False
        while not trouve:
            self._scale = 1 + randrange( p - 1 )
            if not ( self._scale % p ) == 0:
                trouve = True
        self._shift = randrange( p )            #shift entre 0 et p-1
        self._mask = cap

    #fonction de hachage avec compression MAD
    def _hash_function( self, k ):
        return( hash( k ) * self._scale + self._shift ) % self._prime % self._mask

    #taille (nombre d'éléments) dans la table
    def __len__( self ):
        return self._n

    #on utilise des bucket qui seront implémentés
    #dans les sous-classes de HashMap

    #accession à l'élément de clé k en O(1) espéré
    def __getitem__( self, k ):
        #on calcule l'index du bucket
        j = self._hash_function( k )
        #on accède à l'élément de clé k dans le bucket
        return self._bucket_getitem( j, k )
```

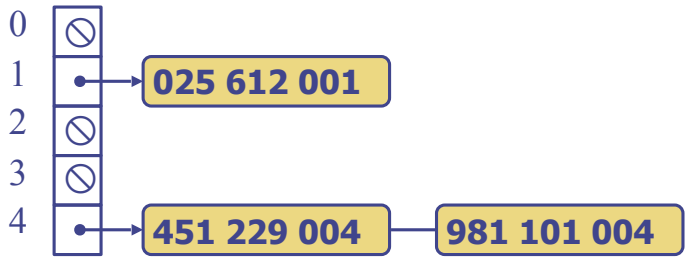
```
#insetion d'un élément (k, v) en O(1) espéré
def __setitem__( self, k, v ):
    #on calcule l'index du bucket
    j = self._hash_function( k )
    #on insère l'élément de clé k dans le bucket
    self._bucket_setitem( j, k, v )
    #si le nombre d'éléments dépasse 50% de la taille de la table
    #on la double
    if self._n > len( self._T ) // 2:
        self._resize( 2 * len( self._T ) - 1 )

#suppression de l'élément de clé k en O(1) espéré
def __delitem__( self, k ):
    #on calcule l'index du bucket
    j = self._hash_function( k )
    #on retire l'élément du bucker
    succes = self._bucket_delitem( j, k )
    #on décrément la taille si l'élément
    #a bel et bien été supprimé (s'il existait)
    if succes:
        self._n -= 1

#redimensionnement de la table à capacité c
def _resize( self, c ):
    #on insère tous les éléments de la table dans une liste
    old = list( self.items() )
    #on crée une nouvelle table avec la nouvelle capacité
    self._T = c * [None]
    #on réinitialise la taille de la table à 0
    self._n = 0
    #on redéfinit le mask à la nouvelle capacité
    self._mask = c
    #on insère les éléments de l'ancienne table un à un
    #cette opération est dans O(n) en temps
    for (k,v) in old:
        self[k] = v
```

Gestion des collisions

Les collisions se produisent lorsque différentes clés sont “mappées" au même indice dans la table

Le chaînage séparé permet à chaque entrée de la table de pointer vers une liste des éléments qui y sont "mappés".

Le chaînage séparé est simple, mais nécessite de la mémoire supplémentaire



```python
from HashMap import HashMap
from UnsortedListMap import UnsortedListMap
import random
import time

#classe ChaineHashMap pour du hachage avec
#chaînage séparé, càd les buckets sont des
#Map externes à la table
class ChainHashMap( HashMap ):

    #implémentation des fonctions pour les buckets

    #accéder à un élément de clé k
    def _bucket_getitem( self, j, k ):
        #j est l'adresse du bucket dans la table
        bucket = self._T[j]
        #si le bucket est vide (n'existe pas)
        #on retourne False
        if bucket is None:
            return False
        #sinon, on retourne l'élément dont la clé est k
        return bucket[k]

    #insertion d'un élément (k, v)
    def _bucket_setitem( self, j, k, v ):
        #si le bucket n'existait pas, on le crée
        #ici une Map non triée
        if self._T[j] is None:
            self._T[j] = UnsortedListMap()
        #on va chercher la taille précédente du bucket
        oldsize = len( self._T[j] )
        #on effectue l'insertion
        self._T[j][k] = v
        #si un élément avec la clé k existait,
        #on a seulement changé sa valeur
        #il ne faut donc pas incrémenter la taille de la table
        if len( self._T[j] ) > oldsize:
            #sinon, on a ajouté l'élément, donc on incérmente la taille
            self._n += 1

    #suppression de l'élément de clé k
    def _bucket_delitem( self, j, k ):
        #j est l'adresse du bucket dans la table
        bucket = self._T[j]
        #si le bucket n'existe pas, l'élément non plus
        if bucket is None:
            return False
        #sinon, on le retire du bucket
        del bucket[k]

    #itérateur des clés de la table de hachage
    def __iter__( self ):
        for bucket in self._T:
            if bucket is not None:
                for key in bucket:
                    yield key
```

Structure de données	Insertion (sec.)	Recherche (sec.)	Suppression (sec.)
Sorted List	O(n)	O(log n)	O(n)
1M	76.5	13.0	38.7
Skip List	O(log n)	O(log n)	O(log n)
1M	30.5	20.8	19.7
ChainHashMap	esp. O(1)	esp. O(1)	esp. O(1)
1M	11.3	3.1	3.4

Sondage linéaire

Adressage ouvert : l'objet en collision est placé dans une entrée différente de la table

Sondage linéaire : gère la collision d'un élément en le plaçant dans la prochaine entrée du tableau (circulairement) disponible.

Les objets qui causent les collisions se regroupent, ce qui augmente le sondage lors de collisions futures

Exemple:

h(x) = x mod 13

Insérez les clés : 18, 41, 22, 44, 59, 32, 31, 73, dans cet ordre

					18							
		41			18							
		41			18				22			
		41			18	44			22			
		41			18	44	59		22			
		41			18	44	59	32	22			
		41			18	44	59	32	22	31		
		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

18 mod 13 = 5
41 mod 13 = 2
22 mod 13 = 9
44 mod 13 = 5 <collision> on prend indice 6
59 mod 13 = 7
32 mod 13 = 6 <collision> on prend indice 8
31 mod 13 = 5 <collision> on prend indice 10
73 mod 13 = 8 <collision> on prend indice 11

Chercher avec sondage linéaire

Considérez une table de hachage qui utilise un sondage linéaire

Pour chercher k, on cherche une entrée avec la clé k

Nous commençons à l'entrée d'indice j = h(k)

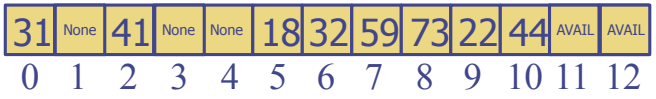
Nous sondons les emplacements consécutifs jusqu'à ce que l'un des événements suivants se produise :

- Un élément avec la clé k est trouvé, ou
- Une entrée vide est trouvée

```
#classe ProbeHashMap pour hachage par probing linéaire
class ProbeHashMap( HashMap ):

    #il faut distinguer entre une entrée vide et une entrée disponible
    _AVAIL = object()

    #une entrée est disponible si elle assignée None ou _AVAIL
    def _is_available( self, j ):
        return self._T[j] is None or self._T[j] is ProbeHashMap._AVAIL

    #chercher l'entrée de la clé k à partir de l'index j
    #assume qu'il existe au moins une entrée None dans la table
    #assuré par l'extension de la table lorsqu'on dépasse 50% d'occupation
    def _find_slot( self, j, k ):
        #on assume aucune entrée n'est disponible
        firstAvail = None
        #on boucle jusqu'à ce que l'une des 2 conditions suivantes
        #est remplie: l'entrée est None, indiquant l'échec
        #             l'entrée contient la clé, indiquand le succès
        while True:
            #si l'entrée est None ou _AVAIL
            if self._is_available( j ):
                #on note l'index de cette (première) entrée (libre)
                if firstAvail is None:
                    firstAvail = j
                #si l'entrée est None, on a terminé de parcourir
                #les entrées de ce sondage sans avoir trouvé la clé k
                #on retourne False et le premier index libre de ce sondage
                if self._T[j] is None:
                    return ( False, firstAvail )
            #sinon, si l'entrée contient la clé, on retourne True et son index
            elif k == self._T[j]._key:
                return ( True, j )
            #on avance circulairement dans la table
            j = (j + 1) % len( self._T )

    #accéder à l'élément de clé k
    def _bucket_getitem( self, j, k ):
        #on le cherche à partir de son index initial j
        #en effectuant un sondage avec _find_slot
        found, s = self._find_slot( j, k )
        #si la recherche échoue, on retourne False
        if not found:
            return False
        #sinon, on retourne la valeur de l'élément de clé k
        return self._T[s]._value
```

Comment fonctionne _find_slot ?

```
#chercher l'entrée de la clé k à partir de l'index j
#assume qu'il existe au moins une entrée None dans la table
#assuré par l'extension de la table lorsqu'on dépasse 50% d'occupation
def _find_slot( self, j, k ):
    #on assume aucune entrée n'est disponible
    firstAvail = None
    #on boucle jusqu'à ce que l'une des 2 conditions suivantes
    #est remplie: l'entrée est None, indiquant l'échec
    #             l'entrée contient la clé, indiquand le succès
    while True:
        #si l'entrée est None ou _AVAIL
        if self._is_available( j ):
            #on note l'index de cette (première) entrée (libre)
            if firstAvail is None:
                firstAvail = j
            #si l'entrée est None, on a terminé de parcourir
            #les entrées de ce sondage sans avoir trouvé la clé k
            #on retourne False et le premier index libre de ce sondage
            if self._T[j] is None:
                return ( False, firstAvail )
        #sinon, si l'entrée contient la clé, on retourne True et son index
        elif k == self._T[j]._key:
            return ( True, j )
        #on avance circulairement dans la table
        j = (j + 1) % len( self._T )
```



_find_slot(4, 82)
  firstAvail = 4
  return(False, 4)

_find_slot(5, 44)
  j = 6
  j = 7
  j = 8
  j = 9
  j = 10
  return(True, 10)

_find_slot(10, 49)
  j = 11
  j = 12
  j = 0
  j = 1
  firstAvail = 11
  return(False, 11)

_find_slot(5, 31)
  j = 6
  j = 7
  j = 8
  j = 9
  j = 10
  j = 11
  j = 12
  j = 0
  return(True, 0)

Mettre à jour avec sondage linéaire

C'est pour gérer les insertions et les suppressions que nous avons introduit un objet spécial, *_AVAIL*, qui remplace les éléments supprimés

Pour retirer l'élément de clé k, on cherche une entrée avec la clé k :

Si une telle entrée est trouvée, nous la remplaçons par l'objet spécial *_AVAIL* et nous retournons la valeur de l'élément

Sinon, nous retournons *False*

Pour insérer un élément de clé k, on cherche un index d'insertion :

Nous commençons à la cellule i = h(k)

Nous sondons les entrées consécutivement et jusqu'à ce que l'un des événements suivants se produise :

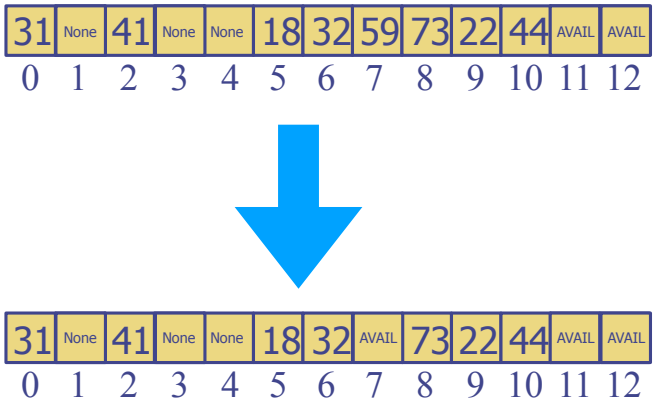
- Une cellule est trouvée vide ou stockant l'objet *_AVAIL*, dans quel cas nous insérons l'élément dans cette cellule, ou
- Une cellule contenant un élément de clé k est trouvée, dans quel cas nous mettons à jour la valeur de cet élément

Comment fonctionne _AVAIL ?

```
#chercher l'entrée de la clé k à partir de l'index j
#assume qu'il existe au moins une entrée None dans la table
#assuré par l'extension de la table lorsqu'on dépasse 50% d'occupation
def _find_slot( self, j, k ):
    #on assume aucune entrée n'est disponible
    firstAvail = None
    #on boucle jusqu'à ce que l'une des 2 conditions suivantes
    #est remplie: l'entrée est None, indiquant l'échec
    #             l'entrée contient la clé, indiquand le succès
    while True:
        #si l'entrée est None ou _AVAIL
        if self._is_available( j ):
            #on note l'index de cette (première) entrée (libre)
            if firstAvail is None:
                firstAvail = j
            #si l'entrée est None, on a terminé de parcourir
            #les entrées de ce sondage sans avoir trouvé la clé k
            #on retourne False et le premier index libre de ce sondage
            if self._T[j] is None:
                return ( False, firstAvail )
        #sinon, si l'entrée contient la clé, on retourne True et son index
        elif k == self._T[j]._key:
            return ( True, j )
        #on avance circulairement dans la table
        j = (j + 1) % len( self._T )
```

Suppression de 59
_find_slot(7, 59)
return(True, 7)

insérer *_AVAIL* à l'index 7



```
#accéder à l'élément de clé k
def _bucket_getitem( self, j, k ):
    #on le cherche à partir de son index initial j
    #en effectuant un sondage avec _find_slot
    found, s = self._find_slot( j, k )
    #si la recherche échoue, on retourne False
    if not found:
        return False
    #sinon, on retourne la valeur de l'élément de clé k
    return self._T[s]._value

#assignation de l'élément (k, v)
def _bucket_setitem( self, j, k, v ):
    #on cherche une entrée disponible à partir de son index initial j
    #en effectuant un sondage avec _find_slot
    found, s = self._find_slot( j, k )
    #si la clé k n'est pas utilisée, on ajoute l'élément dans la table
    #au premier index disponible du sondage, il est retourné dans s
    #et on incrémente la taille de la table
    if not found:
        self._T[s] = self._Item( k, v )
        self._n += 1
    #sinon, on l'a trouvé, alors on met à jour sa valeur
    else:
        self._T[s]._value = v

#suppression de l'élément de clé k
def _bucket_delitem( self, j, k ):
    #on le cherche à partir de son index initial j
    #en effectuant un sondage avec _find_slot
    found, s = self._find_slot( j, k )
    #si la recherche échoue, on soulève une erreur
    if not found:
        return False
    #sinon, on rend l'entrée disponible en lui assignant _AVAIL
    #et on retourne la valeur de l'élément
    value = self._T[s]._value
    self._T[s] = ProbeHashMap._AVAIL
    return value
```

Data structure	Insertion (sec.)	Search (sec.)	Delete (sec.)
Sorted List	O(n)	O(log n)	O(n)
1M	50.1	12.2	36.7
Skip List	O(log n)	O(log n)	O(log n)
1M	26.2	19.0	18.6
ChainHashMap	exp. O(1)	exp. O(1)	exp. O(1)
1M	11.3	3.1	3.4
ProbeHashMap	exp. O(1)	exp. O(1)	exp. O(1)
1M	8.0	3.5	3.7

Hachage double

Le hachage double utilise une fonction de hachage secondaire, d(k), et gère les collisions en plaçant un objet dans la première entrée disponible de la série :

$(i + j\ d(k))$ mod N pour $j = 0, 1, ..., N - 1$

i étant le résultat de la fonction de hachage primaire.

La taille de la table *N* doit être un nombre premier pour permettre le sondage de toutes les cellules

Un choix standard pour la fonction secondaire est :

$d(k) = q - k \bmod q$, où $q < N$ et q est un nombre premier

d(*k*) prend ses valeurs dans { 1, 2, ..., q }

Exemple de hachage double

Considérez une table de hachage stockant des clés entières et gérant les collisions avec un hachage double :

$N = 13$

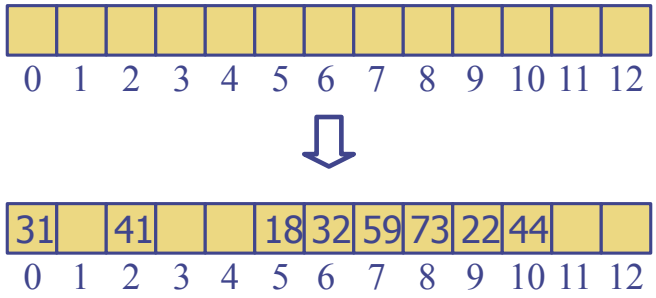
$h(k) = k \bmod 13$

$d(k) = 7 - k \bmod 7$

Insérer les clés 18, 41, 22, 44, 59, 32, 31, 73, dans cet ordre

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

x mod y = x - ⌊x/y⌋ * y



h(44) = 5
d(44) = 5
h(k) + j d(k) = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65]
Sondage = [5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, 8, 0]

h(31) = 5
d(31) = 4
h(k) + j d(k) = [5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53]
Sondage = [5, 9, 0, 4, 8, 12, 3, 7, 11, 2, 6, 10, 1]

Performances du hachage

Dans le pire des cas, les recherches, les insertions et les suppressions dans une table de hachage prennent en pire cas O(n)-temps

Le pire cas se produit lorsque toutes les clés insérées dans la *Map* entrent en collision

Le facteur de charge, $\alpha = n / N$, affecte la performance d'une table de hachage. Il est préférable de garder α plus petit que 1.

En supposant que les valeurs de hachage sont distribuées de manière aléatoire, on peut montrer que la longueur attendue d'un sondage linéaire pour une insertion avec adressage ouvert est de :

$1 / (1 - \alpha)$; par exemple pour *$n / N = 0.5$, on aurait une longueur de 2*

Le temps d'exécution espéré ("*expected*") de toutes les opérations de l'ADT *Map* dans une table de hachage est O(1)-temps

En pratique, le hachage est très rapide à condition que le facteur de charge soit petit.

Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$

Implémentation du *dict* dans Python

- Rappelons que *dict* dand Python n'implémente pas l'ADT Map !
- *dict* de Python est essentiel, car ils est utilisé par plusieurs autres éléments du langage (e.g. les classes et les sous-classes utilisent *dict* pour stocker leurs attributs).
- Une autre utilisation de *dict* est pour la transmission des arguments à une fonction.
- Tout programme Python en cours d'exécution a de nombreux *dict* actifs en même temps. Il est important que les *dict* puissent être créés et détruits rapidement et qu'ils n'utilisent pas trop de mémoire.

L'expérience de l'implémentation de *dict* en Python enseigne que l'analyse comparative empirique est essentielle et représente la seule vraie façon de découvrir ce qui vaut vraiment la peine d'optimiser.

L'objet *PyDictEntry*

- Les *dict* Python sont implémentés avec des tables de hachage avec adressage ouvert pour résoudre les collisions
- La table de hachage Python est un bloc de mémoire contigu permettant la recherche par indice en temps dans O(1) (comme pour un tableau)
- Chaque entrée dans la table correspond à une combinaison des valeurs : <**hash**, **clé**, **valeur**>, implémenté dans un *struct* C
- Un nouveau *dict* est initialisé avec 8 entrées.

Inside the Dictionary

Dictionaries are represented by a C structure, `PyDictObject`, defined in *Include/dictobject.h*. Here's a schematic of the structure representing a small dictionary mapping `"aa"`, `"bb"`, `"cc"`, ..., `"mm"` to the integers 1 to 13:

```
int ma_fill         13
int ma_used         13
int ma_mask         31

PyDictEntry ma_table[]:
[0]: aa, 1              hash(aa) == -1549758592, -1549758592 & 31 = 0
[1]: ii, 9              hash(ii) == -1500461680, -1500461680 & 31 = 16
[2]: null, null
[3]: null, null
[4]: null, null
[5]: jj, 10             hash(jj) == 653184214, 653184214 & 31 = 22
[6]: bb, 2              hash(bb) == 603887302, 603887302 & 31 = 6
[7]: null, null
[8]: cc, 3              hash(cc) == -1537434360, -1537434360 & 31 = 8
[9]: null, null
[10]: dd, 4             hash(dd) == 616211530, 616211530 & 31 = 10
[11]: null, null
[12]: null, null
[13]: null, null
[14]: null, null
[15]: null, null
[16]: gg, 7             hash(gg) == -1512785904, -1512785904 & 31 = 16
[17]: ee, 5             hash(ee) == -1525110136, -1525110136 & 31 = 8
[18]: hh, 8             hash(hh) == 640859986, 640859986 & 31 = 18
[19]: null, null
[20]: null, null
[21]: kk, 11            hash(kk) == -1488137240, -1488137240 & 31 = 8
[22]: ff, 6             hash(ff) == 628535766, 628535766 & 31 = 22
[23]: null, null
[24]: null, null
[25]: null, null
[26]: null, null
[27]: null, null
[28]: null, null
[29]: ll, 12            hash(ll) == 665508394, 665508394 & 31 = 10
[30]: mm, 13            hash(mm) == -1475813016, -1475813016 & 31 = 8
[31]: null, null
```

hash(aa) = -1549758592, et -1549758592 mod 31 = 0, de sorte que l'indice de la clé aa est 0

Les champs de *PyDictEntry*

Le préfixe ***ma***_ dans les noms des champs provient du mot "***ma***pping". Les champs de la structure sont :

ma_used : Nombre d'entrées occupées (dans le cas de l'exemple 13). Incrémenté et décrémenté selon qu'on insère ou retire des éléments de la table

ma_fill : Nombre d'entrées occupées par des clés ou valeurs factices (également 13). N'est pas décrémenté lorsqu'on retire un élément mais est incrémenté lorsqu'on en ajoute un

ma_mask : La taille de la table de hachage contient ***ma_mask*** + 1 entrées (dans l'exemple 32). Le nombre d'entrées dans la table est toujours une puissance de 2

ma_table : Pointeur vers une structure *PyDictEntry*, qui lui contient des pointeurs vers la **clé**, l'**élément** et une **copie en cache du code de hachage** de la clé

Le code de hachage est mis en cache pour des raisons de rapidité. Lors de la recherche d'une clé, les valeurs de hachage peuvent être rapidement comparées avant d'effectuer une comparaison plus lente et complète des clés. Le rehachage d'un *dict* nécessite également les codes de hachage de chaque clé, et ainsi on évite de les recalculer.

Inside the Dictionary

Dictionaries are represented by a C structure, `PyDictObject`, defined in *Include/dictobject.h*. Here's a schematic of the structure representing a small dictionary mapping "aa", "bb", "cc", ..., "mm" to the integers 1 to 13:

```
int ma_fill         13
int ma_used         13
int ma_mask         31

PyDictEntry ma_table[]:
[0]: aa, 1                  hash(aa) == -1549758592, -1549758592 & 31 = 0
[1]: ii, 9                  hash(ii) == -1500461680, -1500461680 & 31 = 16
[2]: null, null
[3]: null, null
[4]: null, null
[5]: jj, 10                 hash(jj) == 653184214, 653184214 & 31 = 22
[6]: bb, 2                  hash(bb) == 603887302, 603887302 & 31 = 6
[7]: null, null
[8]: cc, 3                  hash(cc) == -1537434360, -1537434360 & 31 = 8
[9]: null, null
[10]: dd, 4                 hash(dd) == 616211530, 616211530 & 31 = 10
[11]: null, null
[12]: null, null
[13]: null, null
[14]: null, null
[15]: null, null
[16]: gg, 7                 hash(gg) == -1512785904, -1512785904 & 31 = 16
[17]: ee, 5                 hash(ee) == -1525110136, -1525110136 & 31 = 8
[18]: hh, 8                 hash(hh) == 640859986, 640859986 & 31 = 18
[19]: null, null
[20]: null, null
[21]: kk, 11                hash(kk) == -1488137240, -1488137240 & 31 = 8
[22]: ff, 6                 hash(ff) == 628535766, 628535766 & 31 = 22
[23]: null, null
[24]: null, null
[25]: null, null
[26]: null, null
[27]: null, null
[28]: null, null
[29]: ll, 12                hash(ll) == 665508394, 665508394 & 31 = 10
[30]: mm, 13                hash(mm) == -1475813016, -1475813016 & 31 = 8
[31]: null, null
```

Sondage utilisé en Python

Une séquence de sondage quadratique est utilisée pour trouver une cellule libre.

```
1 j = (5*j) + 1 + perturb;
2 perturb >>= PERTURB_SHIFT;
3 use j % 2**i as the next table index;
```

```
PERTURB_SHIFT = 5

#class Sondage pour mimer le sondage du pyDict
class Sondage:

    def __init__( self, size = 31 ):
        self._size = size

    def sondage( self, hashcode ):
        sondes = []
        j = hashcode % self._size
        perturb = j
        sondes.append( j )
        for k in range( 0, self._size - 1 ):
            j = (5 * j) + 1 + perturb
            #décalage à droite
            perturb >>= PERTURB_SHIFT
            j = j % ( self._size + 1 )
            sondes.append( j )
        print( sondes )
```

Exemple) `sondage( hash( 3 ) )`
avec `hash( 3 )` = 3
on aura la séquence suivante :

```
[3, 19, 0, 1, 6, 31, 28, 13, 2, 11, 24, 25, 30, 23, 20, 5, 26, 3, 16, 17, 22, 15, 12, 29, 18, 27, 8, 9, 14, 7, 4]
```

Le facteur de perturbation *perturb* commence avec le code de hachage complet et ses bits sont ensuite progressivement décalés par 5 bits à la fois. Ce décalage garantit que chaque bit dans le code de hachage affecte le sondage. Le facteur de perturbation descend à zéro, et le modèle dans ce cas devient simplement slot = (5 * slot) + 1, générant chaque entier entre 0 et ***ma_mask***.

La recherche va éventuellement trouver la clé (sur une opération de recherche) ou un emplacement vide (lors d'une opération d'insertion).

La valeur de décalage de 5 bits a été choisie empiriquement. 5 bits minimisent les collisions légèrement mieux que 4 ou 6 bits. Les versions antérieures utilisaient des opérations plus complexes telles que la multiplication ou la division, mais bien que ces versions aient d'excellentes statistiques de collisions, les calculs prennent légèrement plus de temps.

Rehachage

La taille de la table de hachage d'un *dict* doit être ajustée à mesure que les clés sont ajoutées. Le code vise à garder la table aux deux tiers pleine. Si un *dict* contient *n* clés, la table doit avoir au moins *n* / (2/3) entrées libres. Ce ratio est un compromis : remplir la table plus densément entraîne plus de collisions mais utilise moins de mémoire et donc s'intègre mieux dans la cache. Des expériences où le ratio 2/3 a été ajusté en fonction de la taille du *dict* n'ont pas montré de résultats concluants. On vérifie à chaque opération d'insertion si le *dict* doit être rehaché, ce qui ralentit l'opération.

Quelle doit être la nouvelle taille d'un dict après rehachage ?

Pour les dictionnaires de petite ou moyenne tailles (<= 50 000 clés), la nouvelle taille est ***ma_used*** * 4.

Pour les grands dictionnaires (> 50 000 clés), la nouvelle taille est ***ma_used*** * 2, ce qui évite de consommer trop de mémoire pour des entrées vides.

Il n'y a pas de rehachage sur le retrait d'éléments !