

```

# Code écrit par Morgane Farez L1 CMI Informatique USMB
# Hash Table version 2
#liste : adressage lineaire ouvert

collision = [0]
tab_nb_elements = [0]

#taille des listes de collisions
tab_liste_collisions = [0]
### DEBUT ###

### FONCTION HACHAGE ###
def hashFunction(cle):
    """
    Associe une case a un element donne.
    Parameters
    -----
    cle : entier, chaine de caracteres
    table : table de hachage
    Returns
    -----
    pos : entier
        indice de l'element dans la table de hashage.

    """
    #fonction hachage 2
    if type(cle) == int:
        pos = cle
    else :
        pos = 0
        for i in range(len(cle)):
            pos = pos*17 + ord(cle[i])*97 #ord() permet de recuperer le numero ascii du caractere
    return pos%(2**64)

### TABLE DE HACHAGE ###
class HashTable:
    def __init__(self, size):
        """
        creer un tableau de tableaux de la taille demandee a l'initialisation
        Parameters
        -----
        size : entier
        Returns
        -----
        None.
        """
        self.nb_moins_un = 0

```

```

self.table = []
#compter le nombre de case remplie dans la table = compter le nombre d'elements inseres
self.nb_elements = 0
self.taille = size #nombre de cases vides et remplies
self.nb_collisions = 0
for i in range(self.taille):
    self.table.append(None)

def __str__(self):
    """
    permet un affichage comme les dictionnaires python
    Returns
    -----
    affichage : chaine de caracteres
    """
    affichage = '{'
    for i in range(len(self.table)):
        if self.table[i] != None and self.table[i] != -1:
            affichage = affichage + \
                f'{self.table[i][0]} : {self.table[i][1]}, \n '
    affichage += '}'
    return affichage

def ajouter_valeur(self, cle, valeur):
    """
    Ajoute l'element a la position indique de la table si la place est libre
    sinon parcourt la table de 1 en 1 jusqu'a trouver une case vide
    Parameters
    -----
    cle : entier, chaine de caracteres
    valeur : entier, flottant, chaine de caractere
    Returns
    -----
    None.
    """
    position = (hashFunction(cle)) % len(self.table)
    indice = 0
    for i in range(len(self.table)):
        # adressage ouvert LINEAIRE
        position = (position + i) % len(self.table)
        if self.table[position] != None and self.table[position] != -1:
            if self.table[position][0] == cle:
                # met a jour la valeur de la cle ajoute si celle-ci est deja dans la table
                self.table[position] = (cle, valeur)
                break
        else: #si la case est vide (contient une des 2 valeurs par default)
            self.nb_elements += 1
            self.table[position] = (cle, valeur)

```

```

        indice = i
        break
if indice > 0:
    self.nb_collisions+=1

tab_liste_collisions.append(indice)

if collision[len(collision)-1] != self.nb_collisions :
    tab_nb_elements.append(self.nb_elements)
    collision.append(self.nb_collisions)

self.agrandir_table() # agrandit la table si necessaire

def __setitem__(self, cle, valeur):
    """
    permet d'ajouter un couple cle/valeur a la table avec de la facon
    suivante:
        t['r']='t'
    """
    self.ajouter_valeur(cle, valeur)

def recherche_valeur(self, cle):
    """
    Renvoie la valeur correspondant a la cle
    Parameters
    -----
    cle : entier ou chaine de caracteres
    Returns
    -----
    valeur : pas de type particulier
    """
    position = (hashFunction(cle)) % len(self.table)
    valeur = -1
    trouve = False
    indice = 0
    while not trouve and self.table[position] != None and indice < len(self.table):
        if self.table[position] != -1 and self.table[position][0] == cle:
            trouve = True
            valeur = self.table[position][1]
            indice += 1
            position = (position + indice) % len(self.table)
    return valeur

def __getitem__(self, cle):
    """
    permet de faire une recherche d'une valeur dans une table T de la facon
    suivante:
        T[cle]
    """

```

Parameters

cle : entier ou chaîne de caractères

Returns

renvoie la valeur associée à la cle

"""

return self.recherche_valeur(cle)

```
def supprimer_tuple(self, cle):
```

```
    """
```

```
    Supprime le couple cle-valeur de la cle indiquée
```

```
    Parameters
```

```
    -----
```

```
    cle : entier ou chaîne de caractères
```

```
    Returns
```

```
    -----
```

```
    None.
```

```
    """
```

```
    #sans -1
```

```
    position = (hashFunction(cle)) % len(self.table)
```

```
    for i in range(self.taille):
```

```
        position = (position + i) % len(self.table)
```

```
        if (self.table[position] != -1
```

```
            and self.table[position] != None
```

```
            and self.table[position][0] == cle):
```

```
            self.table[position] = None
```

```
            self.nb_elements -= 1
```

```
            self.nb_moins_un += 1
```

```
            break
```

```
    for j in range(self.taille): #repositionne les elements
```

```
        if self.table[j] != None:
```

```
            cle = self.table[j][0]
```

```
            pos = (hashFunction(cle)) % len(self.table)
```

```
            if j != pos:
```

```
                valeur = self.table[j][1]
```

```
                self.table[j] = None
```

```
                self.ajouter_valeur(cle, valeur)
```

```
### version 2 ###
```

```
def supprimer_tuple2(self, cle):
```

```
    #avec -1
```

```
    position = (hashFunction(cle)) % len(self.table)
```

```
    for i in range(self.taille):
```

```
        position = (position + i) % len(self.table)
```

```
        if (self.table[position] != -1
```

```
            and self.table[position] != None
```

```
            and self.table[position][0] == cle):
```

```

        self.table[position] = -1
        self.nb_elements -= 1
        self.nb_moins_un += 1
        break
self.val_par_defaut(cle)

```

```

def __delitem__(self, cle):
    """
    permet de supprimer un element d'une table t de la facon suivante :
    del t['em']
    """
    return self.supprimer_tuple(cle)

```

```

def agrandir_table(self):
    """
    test si la table est remplie a plus de 2/3 de sa taille et double sa
    taille si c'est le cas
    Returns
    -----
    None.
    """
    if self.nb_elements >= ((0.5)*self.taille):
        ancienneTable = self.table
        self.__init__(self.taille*4)
        for i in range(len(ancienneTable)):
            # recalcule la position de chaque elements presents dans ancienneTable
            if ancienneTable[i] != None and ancienneTable[i] != -1:
                self.ajouter_valeur(ancienneTable[i][0], ancienneTable[i][1])

```

```

#version 2
def val_par_defaut(self, cle):
    """
    Remplace les -1 par la valeur par defaut lorsqu'il y en a trop.
    Returns
    -----
    None.
    """
    if self.nb_moins_un >= (1/3*self.taille):
        for i in range(self.taille):
            if self.table[i]==-1:
                self.table[i]=None
            elif self.table[i]!=None:
                cle = self.table[i][0]
                pos = (hashFunction(cle)) % len(self.table)
                if i!=pos:
                    valeur = self.table[i][1]
                    self.table[i]=None

```

```
self.ajouter_valeur(cle, valeur)
```