# Algorithmique et Structure discrète : TP3

Ayoub BARICH, Camille VEDANI

**Résumé**

On veut implémenter une table de hachage, contenant les multi-ensembles constitués des lettres des mots. Nous avons eu une approche orientée objet pour notre code. Vous trouverez le code complet en cliquant sur ce lien :

`https://github.com/AyoubBarich/TP3_Barich_Vedani/tree/master`

## 1 Implémentation :

### 1.1 Structures de données

On a choisi de créer notre propre structure de données pour implémenter notre table de hachage. Pour cela, nous nous sommes basés sur les listes chainées de java, mais en ajoutant une méthode set. En effet, lors de l'insertion de nouvelles listes, on regarde si la liste actuelle a une liste à coté (hasNext). Si oui, on applique encore une fois et récursivement la méthode set sur la liste suivante. On continue cela jusqu'à ce qu'il n'y est plus d'élément après et on insère une nouvelle liste contenant le nouvel élément choisi. Cet élément est une liste dont le premier indice contient le multi-ensemble du mot et les indices suivants contiennent les mots associés à ce multi-ensemble. Par exemple, pour 'r','t','a', on aura : [['r','t','a'],"art","rat"]

```
class linkedList():
    """An implementation of a linked list where each element is a list where the first index containts
    the set charecters of our word and the other indexes contains the string  of the words  """
    def __init__(self,element,next):
        """Intlize our class"""
        self.element=element
        self.next=next
        self.size=0


    def hasNext(self):
        """
        returns: true if self has a linkedlist to the left
        param : self
        """
        return (self.next is not None)

    def set(self,element):
        """
        traverses the Linkedlist until the last Linkedlist attaches a linked list wich contains element
        returns: None
        param : self,element:list
        """

        newlist = linkedList(element,None)
        current=self
        if current.hasNext():
            current=current.next
            current.set(element)
        current.next = newlist
        self.size+=1
```

**Figure 1** Capture d'écran du code "Listes chainées"

### 1.2 Taille de la table de hachage

Pour définir la taille de la table, on s'est basé sur la méthode de la division. M doit donc répondre à ses critères suivants : ne pas être une puissance de 2, ni être proche d'une

puissance de 2. Nous avons donc choisi 107 pour le mini-dictionnaire et 141959 pour le grand dictionnaire, en se basant sur ces critères et sur l'estimation grossière du nombre de mots par alvéole (division de n/M, où M est la taille du tableau T et n est le nombre de clés de K).

## 1.3 Initialisation de la table

Pour initialiser la table de hachage, vue notre approche orientée objet, on a simplement fait un constructeur de notre classe. Notre table de hachage est constituée de clés, qu'on obtient par application directe de la fonction de hachage sur notre multi-ensemble. Cela nous permet de créer une table de hachage avec des alvéoles qui sont définies par des listes chaînées.

```
def getHash(self,key:list):
    """
    returns: the hash value

    param : key=type:list of charecters
    """
    K=0
    key.sort()
    for charcter in key:
        K+=ord(charcter)
    K=math.floor(K*PHI)
    return K%self.sizeMax
```

**Figure 2** Fonction de hachage de la table

```
class HashTable:
    """
    HashTable is a class that implements a hash table with its own hach function "getHash".
    HasTable : hash(key) --> Value = [multiSet,words asscociated with the multiSet]
    we initialize the hashtable as empty and we can insert ,find , remove element from our hashtable.

    """


    def __init__(self) :
        """initiates our hash table"""
        self.sizeMax=SIZEMAX
        self.size=0
        self.keys=dict(zip(range(SIZEMAX), repeat(linkedList([],None))))
```

**Figure 3** Initialisation de la table de hachage

## 1.4 Remplissage de la table

Pour remplir notre table, il faut dans un premier temps déterminer notre fonction de hachage. Pour cela, nous avons utilisé la méthode de la multiplication. En effet, on a choisi : $A = \phi - 1 = 0,6180339887...$ La fonction de hachage vaut donc :

$$h(k) = \lfloor m(kAmod1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

Une fois la fonction définie, il suffit juste de d'appliquer la fonction sur notre ensemble de clés. Cela va déterminer dans quelles alvéoles elles se trouveront. En cas de collisions, c'est-à-dire plusieurs plusieurs clés dans une même alvéoles (car obtention de mêmes valeurs par la fonction de hachage), les alvéoles contiendront nos listes chaînées définies précédemment.

```
def set(self,key,value):
    """
    returns: True if the key value pair are correctly set in the hash table

    param : key=type:list of charecters and Value
    """
    self.size+=1
    hash = self.getHash(key)
    linkedlist = self.keys[hash]

    if linkedlist.isEmpty():
        self.keys.__setitem__(hash,linkedList([key,value],None))
        return True
    while linkedlist is not None:
        if key in linkedlist.element:
            if value not in linkedlist.element:
                linkedlist.element.append(value)
                return True
        linkedlist=linkedlist.next

    self.keys[hash].set([key,value])
    return True
```

**Figure 4** Remplissage de la table de hachage par un seule couple (clé,valeur).

```
def Dictionary(dictionary,showExecutionTime=False):
    """
    returns: a hash table with all of the words in the given dictionary
    param: dictionary:list
    """
    start = time.time()
    ht = HashTable()
    for word in dictionary:
        key = [*word]
        ht.set(key,word)
    if showExecutionTime:
        print("Execution time : ",(time.time())-start)
    return ht
```

**Figure 5** Remplissage de la table de hachage par un dictionnaire.

## 1.5 Recherche de deux mots dont la 2-somme est R

Pour implémenter la méthode de recherche des deux mots dont la 2-somme est R, on va dans un premier temps initialiser une liste vide. Puis à l'aide de la fonction getAllCombinationsOfaMultiset, on détermine toutes les combinaisons possibles de notre premier ensemble saisi en argument. On fait de même avec l'ensemble complémentaire de notre ensemble saisi. Ensuite on cherche le ou les complémentaires associés à chaque combinaison. On sauvegarde les valeurs dans un tableau de couples de type (mot,complémentaire) et on passe par chaque couple pour prendre les mots associés. Cela donne alors les deux mots de la 2-somme recherchés.

```
def getAllCombinationsOfaMultiSet(self,multiset):
    """
    returns:a list of all possible commbination of a charecter set
    param: multiset:list
    """
    res=[]
    for i in range(1,len(multiset)):
        for subSet in combinations(multiset, i):
            if subSet not in res:
                res.append(subSet)
    return res
```

**Figure 6** Méthode pour trouver toutes les combinaisons de l'ensemble

```
def getComplementaire(self,mutiSet,set ):
    res = []
    setlist = list(set)
    mutiSetList=list(mutiSet)
    for char in mutiSet:
        if char in setlist:
            setlist.remove(char)
        else:
            res.append(char)
    if self.exists(res) :
        return res
    return []
```

**Figure 7** Méthode pour trouver le complémentaire de l'ensemble

```python
def twoSum (self,multiset,showExecutionTime=False):
    """
    returns:the couple of words(U,V) where the sum the set of charecters that make up U and V
    and add up to the multiSet
    param: multiset:list
    """
    start =time.time()
    res=[]
    allCombinationsOfAMultiSet=self.getAllCombinationsOfaMultiSet(multiset)
    allCombinationOfmultiSetandComplementary=[]
    max=len(allCombinationsOfAMultiSet)
    progrss= 0
    print("Step 1 Complete!")
    for combination in allCombinationsOfAMultiSet:
        progrss+=1
        if math.floor(progrss)==0:
            print((progrss/max)*100)
        complementary = self.getComplementaire(multiset,combination)
        if complementary != []:
            print((list(combination),list(complementary)))

            allCombinationOfmultiSetandComplementary.append((list(combination),list(complementary)))
    print("Second step complete!")
    progrss=0
    for couple in allCombinationOfmultiSetandComplementary:
        combination = self.get(list(couple[0])).getElementFromMuliset(list(couple[0]))
        complementary=self.get(list(couple[1])).getElementFromMuliset(list(couple[1]))
        progrss+=1
        if math.floor(progrss)%10==0:
            print((progrss/max)*100)
        if not(combination is None) :
            if not(complementary is None):
                res.append((combination,complementary))

    if showExecutionTime:
        print("Execution Time : ",time.time()-start)
    return res
```

**Figure 8** Méthode 2-Somme

# 2 Expérimentations :

## 2.1 Pour le mini-dictionnaire

```
Question1
Execution time :  0.004376649856567383
Question2
Execution Time :  0.47355055809020996
[(['annuler'], ['dioxine']), (['dioxine'], ['annuler'])]
Question3
[]
Execution time :  1.1646759510040283
ayoub@ayoub-Legion-5-15ACH6H:~/Univ/Algo/TP3_Barich_Vedani$
```

**Figure 9** Capture d'écran du terminal lors des tests sur le mini-dictionnaire

## 2.2 Pour le dictionnaire

Pour le dictionnaire, nous n'avons que le premier test de concluant. Le temps d'exécution est de 56 secondes environ. Pour les autres tests, la table de hachage est bien définie. Néanmoins, ils retournent tous des valeurs arbitraires (mots qui ne peuvent-être obtenus avec les ensembles de tests). On ne sait pas pourquoi. Vous pouvez trouver la solution de ce bug sur gitHub, car par contrainte de temps nous n'avons pu régler le problème maintenant.