# TD8
# Tables de hachage

Nous avons vu précédemment la structure de dictionnaire qui associe des *valeurs* à des *clefs*.

```
dico = {'cle1' : valeur1, 'cle2': valeur2, ..., 'clen': valeurn}
```

Que se passe-t-il lorsque la taille du dictionnaire $n$ devient grande ? Lorsque l'on veut accéder à une valeur `dico[clef]`, il faut a priori tester toutes les clefs, l'accès se fait alors en $O(n)$. Pour une liste, l'opération est seulement en $O(1)$ car l'*indice* est suffisamment explicite en lui-même pour trouver l'adresse mémoire. La complexité est la même si l'on veut savoir si un item est la clef d'un dictionnaire (opération de recherche).

Une solution efficace pour diminuer la complexité est donnée par une structure de données un peu spéciale appelée **table de hachage**. Une table de hachage est un tableau associant des valeurs à un *indice*, similairement à une liste sauf que plusieurs valeurs peuvent être associées à un même indice.

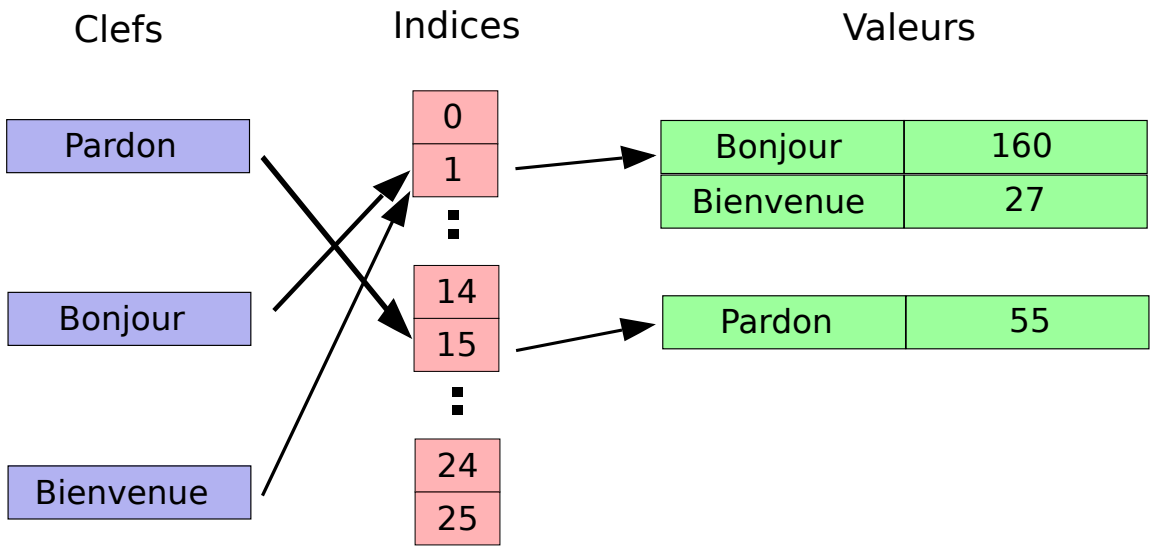


Figure 1: Exemple d'une table de hachage

Une table de hachage va de paire avec une fonction qui transforme des valeurs de *clefs* en *indices*. Cette fonction est appelée **fonction de hachage**. Étant donnée une fonction de hachage $f$, on peut transformer un dictionnaire en table de hachage $T$ de la façon suivante. Au départ $T$ est initialisée comme une liste de listes vides comprenant `nb_indices` indices. Puis pour chaque couple clef-valeur `(c, v)` du dictionnaire :

- On calcule l'indice correspond à `c` grâce à la fonction de hachage : $i = f(c)$
- On ajoute le couple `(c,v)` à la liste $T[i]$

Comment retrouve-t-on la valeur associée à une clef avec la table de hachage ? D'abord, on retrouve l'indice associé $i = f(c)$, toujours avec la fonction de hachage. Ensuite il reste à tester les éléments de $T[i]$ jusqu'à ce que l'on retrouve la clef et la valeur associée.

Dans ce TP on se propose d'implémenter une version simplifiée de table de hachage dont les clefs sont des chaînes de caractères et les indices les entiers de 0 à 25.

*Note :* Le dictionnaire de Python implémente en fait automatiquement une table de hachage. L'accès à une valeur d'un dictionnaire est en $O(1)$, car le nombre d'indices disponibles est grand devant le nombre de clefs. Ainsi en pratique on ne s'inquiète jamais de l'efficacité d'une structure de dictionnaire.

## Exercice 1 : Nombre d'occurrences de mots

1. Écrire une fonction `creerListeMots(s)` qui sépare une chaîne de caractères en mots et renvoie cette chaîne en liste de mots (on considère qu'il n'y a pas de ponctuation ni caractères spéciaux dans `s`).

2. Écrire une fonction `creerDictionnaire(s)` qui renvoie un dictionnaire mettant en correspondance chaque mot apparaissant dans la chaîne avec le nombre de ses occurrences dans la chaîne.
   Par exemple, si la fonction prend en argument la chaîne `"etre ou ne pas etre"`, le dictionnaire renvoyé par la fonction aura pour valeur : `{'etre':2,'ou':1,'ne':1,'pas':1}`.

## Exercice 2 : Création de la table

1. Définir une fonction de hachage `f_hash(s)` sur des chaînes de caractère renvoyant l'entier correspondant à la première lettre apparaissant dans `s`. ex: `f_hash("arbre")` renvoie 0. Vous pouvez vous aider de la fonction `ord(char)`.

2. Écrire la fonction `ajoute(T, f, c, v)` qui ajoute le couple *clef-valeur* `(c,v)` à la table `T` à l'aide de la fonction de hachage `f`.

3. Écrire la fonction `creerTable(d, f, nb_indices)` qui crée une table de hachage à partir d'un dictionnaire `d` et de la fonction `f`.

## Exercice 3 : Opérations

1. Écrire la fonction `recherche(T, f, c)` qui, étant données une table `T` et la fonction de hachage `f` associée, indique si `c` est une *clef* de la table.

2. Écrire la fonction `valeur(T, f, c)` qui retourne la valeur `v` associée à `c`.

## Exercice 4 : Fonctions de hachage

L'efficacité d'une table de hachage dépend de la fonction de hachage. Si un indice revient souvent, la fonction est peu efficace.

1. Selon la fonction de hachage et si la table comprend 26 indices et $n$ valeurs, combien d'opérations de comparaison faut-il effectuer dans le pire des cas ? dans le meilleur des cas ? (quelle que soit la clef en paramètre)

2. Créer un dictionnaire des occurences puis la table correspondante pour le texte `micromegas.txt` donné dans le dossier du TD sur github.

   *Rappel pour la lecture d'un fichier* :

   - changer le répertoire courant avec `os.chdir`
   - pour lire un fichier :

```
with open('micromegas.txt', 'r') as fl:
    texte = fl.read()
```

3. Pour la table créée précédemment, trouver le nombre d'éléments par indice. La table est-elle équilibrée ? Pourquoi ? Si le temps le permet, vous pouvez tracer un histogramme avec l'aide de la fonction `plt.bar` de `matplotlib.pyplot`.

4. Comparer avec les fonctions de hachage suivante (`hash` est une primitive de Python) :

```
def f_hash2(s):
    return hash(s)%26
```

```
def f_hash3(s):
    return hash(s)%1024
```

La fonction primitive de Python est une fonction complexe qui assure une sortie *presque* aléatoire en fonction de la chaîne d'entrée (càd toute régularité dans les données est "cassée" par la fonction de hachage). Une bonne fonction de hachage doit être rapide et avoir cette propriété de désordre.
Attention pour `f_hash3`, `nb_indices` est égal à 1024.