# Programmation Procédurale – Hash tables

Polytech'Nice Sophia Antipolis

Erick Gallesio

*2015 – 2016*

# Introduction

Ce cours présente une structure de données que nous allons implémenter en TD: les **tables de hachage** (hash tables)

- Une table de hachage est une structure de données qui permet une association clé-élément.
- Autres structures de données (ATD) permettant ce type d'association.
  - ADT liste: recherche en O(n)
  - ADT arbre: recherche en O(log(n))
  - Le temps d'accès dépend donc du nombre d'éléments dans la structure de donnée
  - si n augmente, le temps de recherche augmente
- Si la clé est un entier positif ou nul, on peut faire mieux:
  - les tableaux on un temps d'accès en O(1)
- *But*: Essayer de faire la même chose pour des clés quelconques.

# Principe

- La table est représentée par un tableau de longueur `n`
- On accède à chaque élément de la table via sa clé.
- L'accès à une élément se fait
  - en transformant la clé en une valeur entière
  - en ramenant cette valeur entière dans l'intervalle `[0..n[`
- Ainsi,
  - on accède directement à la valeur associée à la clé
  - recherche en O(1) (comme pour les tableaux)
  - Le temps d'accès ne dépend donc pas de n

# Exemple

Cet exemple est tiré de la page *Wikipedia* sur les tables de hachage.

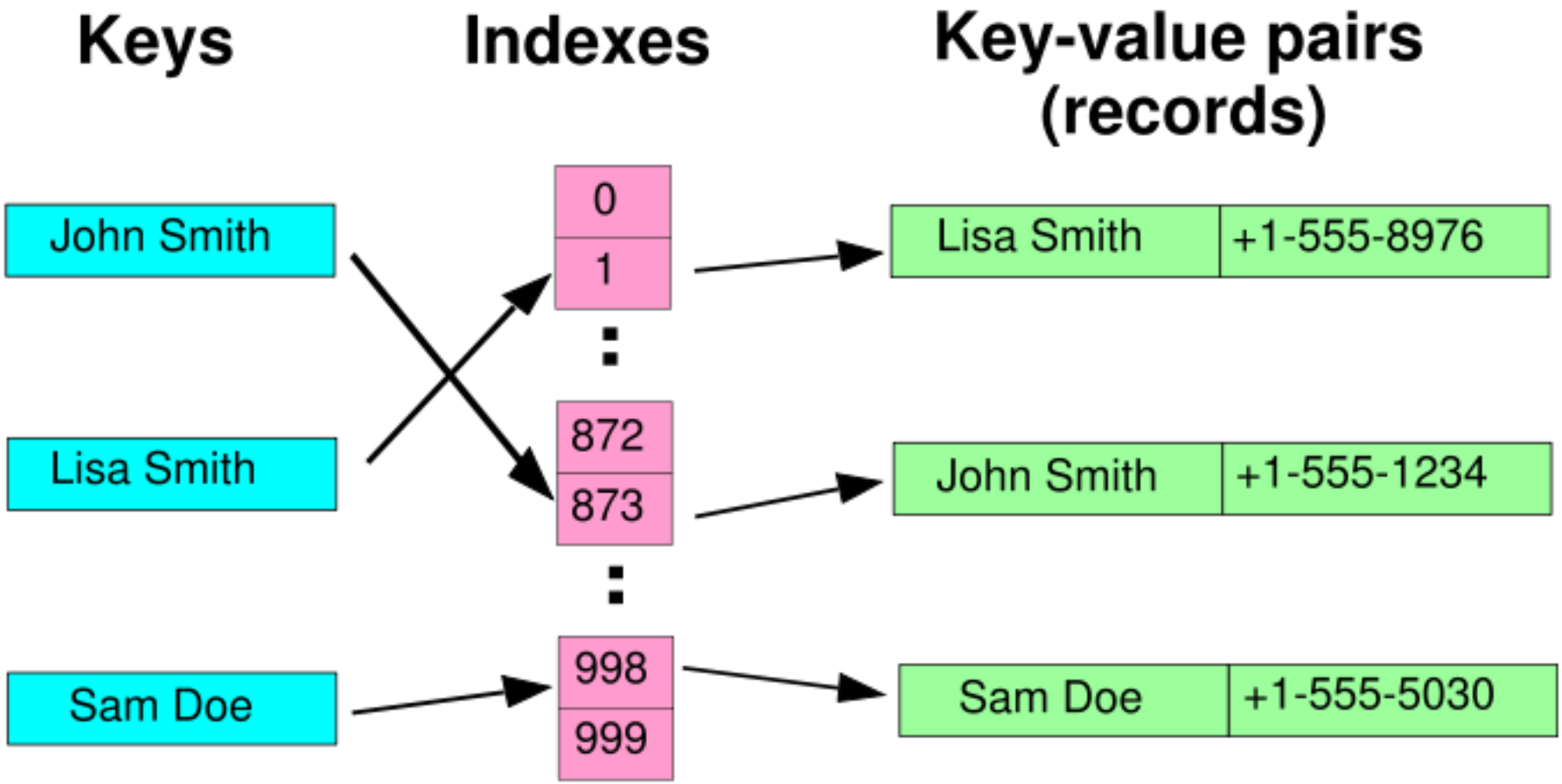


Figure : Un annuaire représenté comme une table de hachage.

# Collisions

- La création d'une valeur entière comprise dans `[0..n[` peut engendrer un problème de **collision**.
- On a une collision lorsque **deux clés différentes**, voire davantage, pourront se retrouver associées à la **même valeur de hash**,
- Lorsqu'il y a collision, on a donc plusieurs clés qui donnent accès à la même position dans le tableau. Il faut:
  - trouver une fonction qui minimise le risque de collisions
  - implémenter un mécanisme de résolution des collisions.

# Fonction de hash (1/3)

C'est la fonction qui permet de passer de la clé à une valeur entière

Cette fonction est un compromis entre:

- rapidité de la fonction hachage
- taille à réserver pour l'espace de hachage
- réduction du risque des collisions

La fonction de hash dépend du type de la clé.

# Fonction de hash (2/3)

Si la clé est une chaîne, on pourrait avoir

```
unsigned long hash(const char *key) {
  unsigned int hash = 0;

  for (int i = 0; key[i]; i++) {
    hash += (unsigned int) key[i];
  }
  return hash;
}
```

- Cette fonction n'est pas *terrible*
- "bcd", "dcb", et toutes leurs permutations produisent las même valeur de hash
- "bcd", "add" et "ABMY" produisent aussi la même valeur

# Fonction de hash (3/3)

Trouver une bonne fonction de hash (surtout générale) est en général assez difficile.

La fonction suivant a un bon comportement en moyenne:

```
unsigned long hash(const char *str) {
  // djb2: this function was first reported by Dan
  // Bernstein many years ago in comp.lang.c
  unsigned long hash = 5381;
  for (   ; *str; str++)
    hash = ((hash << 5) + hash) + *str; // hash*33 + *str
  return hash;
}
```

On obtient:

$bcd \longrightarrow 193487086$

$dcb \longrightarrow 193489262$

$ABMY \longrightarrow 6383851310$

# Résolution des collisions

Lorsque deux clés ont la même valeur de hachage:

- ces clés ne peuvent être stockées à la même position.
- on doit alors employer une méthode de résolution des collisions.

Même si la fonction de hachage a une distribution parfaitement uniforme,

- il y a 95 % de chances d'avoir une collision dans une table de taille 1 million avant même qu'elle ne contienne 2500 éléments.
- Les collisions ne posent cependant de réel problème que si elles sont nombreuses **au même endroit**.
- Même une collision unique sur chaque clé utilisée n'a pas d'effet très perceptible.

Plusieurs méthodes de traitement des collisions existent.

- méthode par chaînage
- adressage ouvert

# Chaînage (1/2)

- Chaque case de la table est en fait une liste chaînée des clés qui ont la même valeur de hash.
- Une fois la case trouvée, la recherche est alors linéaire en la taille de la liste chaînée.
- Dans le pire des cas on a un temps de recherche O(n)

# Chaînage (2/2)

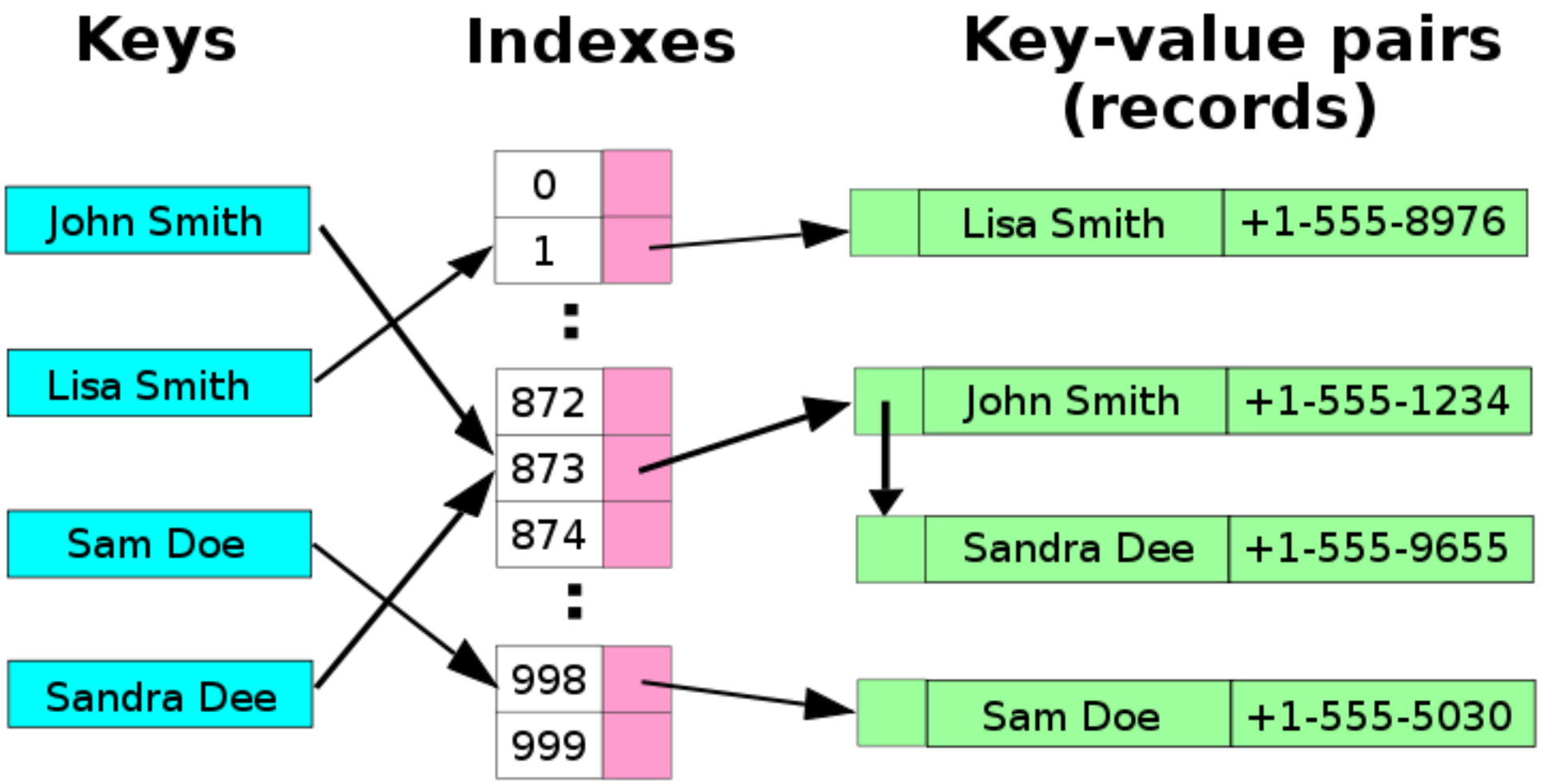


Figure : Résolution des collisions par chaînage.

Ici “John Smith” et “Sandra Dee” ont la même valeur de hash (873)

# Adressage ouvert (1/2)

Dans le cas ou il y a une collision:

- on cherche une autre case dans la table qui n'est pas utilisée
- on peut par exemple
  - prendre la case suivante du tableu
  - avancer avec un intervalle qui change à chaque itération (e.g. $h_i(x) = \left(h(x) + (-1)^{i+1} \cdot \left\lceil \frac{i}{2} \right\rceil^2\right) \bmod n$)
  - utiliser une fonction de hachage secondaire

# Adressage ouvert (2/2)

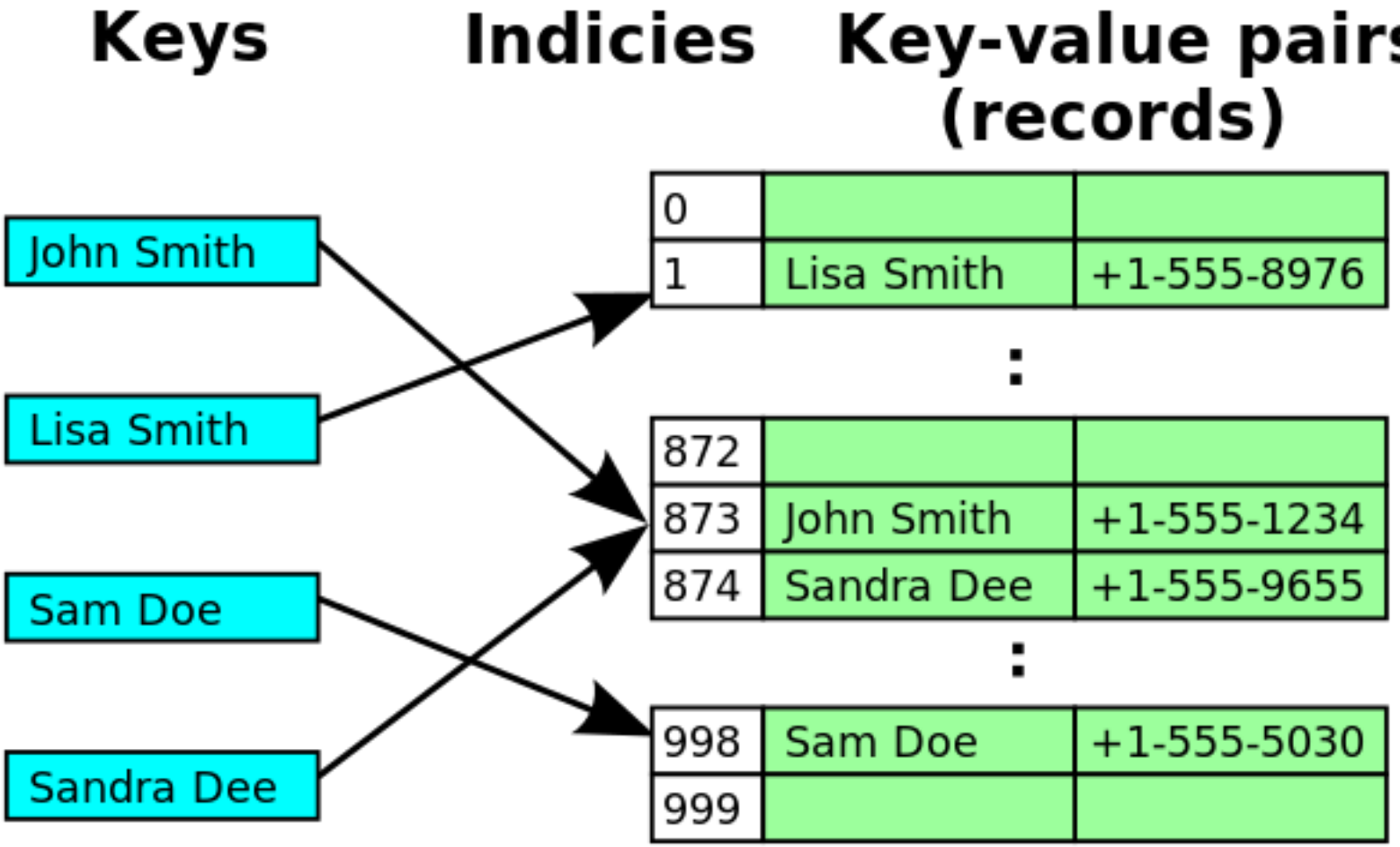


Figure : Résolution des collisions par chaînage.

Là encore, “John Smith” et “Sandra Dee” ont la même valeur de hash (873)

# Comparaison des méthodes de résolution des collisions

- L'avantage du chaînage est que
  - la suppression d'une clé est facile,
  - l'agrandissement de la table peut être retardé plus longtemps que dans l'adressage ouvert
  - les performances se dégradant moins vite.
- L'inconvénient du chaînage:
  - stockage extérieur de la table implique le passage par de l'allocation dynamique
  - temps de mise en œuvre (bookeeping) plus élevé

# Modification de la taille de la table

- Lorsque la table commence à se remplir, les collisions deviennent fréquentes.
- On peut augmenter la taille de la table lorsque son taux de remplissage atteint un certain seuil.
- L'augmentation de la taille de la table correspond en fait à
  - la création d'une nouvelle table vide
  - le remplissage de la nouvelle table avec les valeurs de l'ancienne table
  - la désallocation de l'ancienne table
- L'augmentation de la taille de la table est une opération
  - longue
  - et coûteuse en mémoire

# Une implémentation générale

En TDs, on implémentera un ADT **hash-table**:

- les clés sont des chaînes
- les valeurs associées à une clé sont de n'importe quel type.

*Extension:*

- Allocation d'une "petite" table pour commencer
- la taille de la table augmente en fonction des besoins