Dictionnaires

Lycée Thiers

Crédits

- Tableaux associatifs (Wikipedia)

Crédits

- Tableaux associatifs (Wikipedia)
- openclassrooms

Crédits

- Tableaux associatifs (Wikipedia)
- openclassrooms
- Informatique Tronc Commun (Serge Bays - Ellipse)

Tableaux associatifs

Définition

Un *tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) est un type de données associant à un ensemble de clefs un ensemble correspondant de valeurs.
Chaque clef est associée à une valeur : un tableau associatif correspond donc à une application en mathématiques.

- Grossièrement, le tableau associatif est une généralisation du tableau : alors que le tableau traditionnel associe des entiers consécutifs à des valeurs d'un certain type, le tableau associatif associe des valeurs d'un type arbitraire à des valeurs d'un autre type.

Tableaux associatifs

Définition

Un *tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) est un type de données associant à un ensemble de clefs un ensemble correspondant de valeurs.
Chaque clef est associée à une valeur : un tableau associatif correspond donc à une application en mathématiques.

- Grossièrement, le tableau associatif est une généralisation du tableau : alors que le tableau traditionnel associe des entiers consécutifs à des valeurs d'un certain type, le tableau associatif associe des valeurs d'un type arbitraire à des valeurs d'un autre type.
- Exemples de la vie courante :

Tableaux associatifs

Définition

Un *tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) est un type de données associant à un ensemble de clefs un ensemble correspondant de valeurs.
Chaque clef est associée à une valeur : un tableau associatif correspond donc à une application en mathématiques.

- Grossièrement, le tableau associatif est une généralisation du tableau : alors que le tableau traditionnel associe des entiers consécutifs à des valeurs d'un certain type, le tableau associatif associe des valeurs d'un type arbitraire à des valeurs d'un autre type.
- Exemples de la vie courante :
  - dictionnaire franco-anglais. Les clés sont les mots en français et les valeurs les traductions en anglais.

Tableaux associatifs

Définition

Un *tableau associatif* (aussi appelé *dictionnaire* ou *table d'association*) est un type de données associant à un ensemble de clefs un ensemble correspondant de valeurs.
Chaque clef est associée à une valeur : un tableau associatif correspond donc à une application en mathématiques.

- Grossièrement, le tableau associatif est une généralisation du tableau : alors que le tableau traditionnel associe des entiers consécutifs à des valeurs d'un certain type, le tableau associatif associe des valeurs d'un type arbitraire à des valeurs d'un autre type.
- Exemples de la vie courante :
  - dictionnaire franco-anglais. Les clés sont les mots en français et les valeurs les traductions en anglais.
  - Annuaire : les clefs sont les noms des usagers, les valeurs sont les numéros de téléphone.

Opérations courantes

- Opérations dont la complexité temporelle attendue est en $O(1)$:

Le plus souvent ces opérations ont un coup moyen en $O(1)$ et une complexité au pire (rarement atteinte) en $O(n)$ (où n est le nombre de clés)

Opérations courantes

- Opérations dont la complexité temporelle attendue est en $O(1)$:
  - ajout : association d'une nouvelle valeur à une nouvelle clef ;

Le plus souvent ces opérations ont un coup moyen en $O(1)$ et une complexité au pire (rarement atteinte) en $O(n)$ (où n est le nombre de clés)

Opérations courantes

- Opérations dont la complexité temporelle attendue est en $O(1)$:
  - ajout : association d'une nouvelle valeur à une nouvelle clef ;
  - modification : association d'une nouvelle valeur à une ancienne clef ;

Le plus souvent ces opérations ont un coup moyen en $O(1)$ et une complexité au pire (rarement atteinte) en $O(n)$ (où n est le nombre de clés)

Opérations courantes

- Opérations dont la complexité temporelle attendue est en $O(1)$:
  - ajout : association d'une nouvelle valeur à une nouvelle clef ;
  - modification : association d'une nouvelle valeur à une ancienne clef ;
  - suppression : suppression d'une clef et de la valeur correspondante ;

Le plus souvent ces opérations ont un coup moyen en $O(1)$ et une complexité au pire (rarement atteinte) en $O(n)$ (où n est le nombre de clés)

Opérations courantes

- Opérations dont la complexité temporelle attendue est en $O(1)$:
  - ajout : association d'une nouvelle valeur à une nouvelle clef ;
  - modification : association d'une nouvelle valeur à une ancienne clef ;
  - suppression : suppression d'une clef et de la valeur correspondante ;
  - recherche : détermination de la valeur associée à une clef, si elle existe ;

  Le plus souvent ces opérations ont un coup moyen en $O(1)$ et une complexité au pire (rarement atteinte) en $O(n)$ (où n est le nombre de clés)

Opérations courantes

- Opérations dont la complexité temporelle attendue est en $O(1)$:
  - ajout : association d'une nouvelle valeur à une nouvelle clef ;
  - modification : association d'une nouvelle valeur à une ancienne clef ;
  - suppression : suppression d'une clef et de la valeur correspondante ;
  - recherche : détermination de la valeur associée à une clef, si elle existe ;

  Le plus souvent ces opérations ont un coup moyen en $O(1)$ et une complexité au pire (rarement atteinte) en $O(n)$ (où n est le nombre de clés)
- Dans les sujets de concours d'ITC, sauf mention du contraire, on peut supposer abusivement que les opérations d'ajout-insertion-recherche-modification sont en temps constant au pire cas.

Opérations courantes

- Opérations dont la complexité temporelle attendue est en $O(1)$:
  - ajout : association d'une nouvelle valeur à une nouvelle clef ;
  - modification : association d'une nouvelle valeur à une ancienne clef ;
  - suppression : suppression d'une clef et de la valeur correspondante ;
  - recherche : détermination de la valeur associée à une clef, si elle existe ;

  Le plus souvent ces opérations ont un coup moyen en $O(1)$ et une complexité au pire (rarement atteinte) en $O(n)$ (où n est le nombre de clés)
- Dans les sujets de concours d'ITC, sauf mention du contraire, on peut supposer abusivement que les opérations d'ajout-insertion-recherche-modification sont en temps constant au pire cas.
- Souvent, on s'accorde la possibilité de boucler sur les clés du dictionnaire.

Ordre des clés

- Dans un tableau (considéré comme un dictionnaire dont les clés sont des entiers), il y a un ordre naturel pour les clefs : celui des entiers.

Ordre des clés

- Dans un tableau (considéré comme un dictionnaire dont les clés sont des entiers), il y a un ordre naturel pour les clefs : celui des entiers.
- Mais la notion théorique de dictionnaire n'impose en rien que l'ensemble des clés possibles soit ordonné.

Ordre des clés

- Dans un tableau (considéré comme un dictionnaire dont les clés sont des entiers), il y a un ordre naturel pour les clefs : celui des entiers.
- Mais la notion théorique de dictionnaire n'impose en rien que l'ensemble des clés possibles soit ordonné.
- Toutefois, les langages implémentant les dictionnaires autorisent souvent le parcours des clés.

Ordre des clés

- Dans un tableau (considéré comme un dictionnaire dont les clés sont des entiers), il y a un ordre naturel pour les clefs : celui des entiers.
- Mais la notion théorique de dictionnaire n'impose en rien que l'ensemble des clés possibles soit ordonné.
- Toutefois, les langages implémentant les dictionnaires autorisent souvent le parcours des clés.
  - En `Python`, un dictionnaire est un *itérable*, c'est à dire qu'on peut parcourir son contenu avec une boucle `for`.

Ordre des clés

- Dans un tableau (considéré comme un dictionnaire dont les clés sont des entiers), il y a un ordre naturel pour les clefs : celui des entiers.
- Mais la notion théorique de dictionnaire n'impose en rien que l'ensemble des clés possibles soit ordonné.
- Toutefois, les langages implémentant les dictionnaires autorisent souvent le parcours des clés.
  - En `Python`, un dictionnaire est un *itérable*, c'est à dire qu'on peut parcourir son contenu avec une boucle `for` .
  - Actuellement en `Python` (mais ça peut changer), l'ordre du parcours des clés est celui de l'insertion.

Ordre des clés

- Dans un tableau (considéré comme un dictionnaire dont les clés sont des entiers), il y a un ordre naturel pour les clefs : celui des entiers.
- Mais la notion théorique de dictionnaire n'impose en rien que l'ensemble des clés possibles soit ordonné.
- Toutefois, les langages implémentant les dictionnaires autorisent souvent le parcours des clés.
  - En `Python`, un dictionnaire est un *itérable*, c'est à dire qu'on peut parcourir son contenu avec une boucle `for`.
  - Actuellement en `Python` (mais ça peut changer), l'ordre du parcours des clés est celui de l'insertion.
- Il faut bien comprendre que même si l'ensemble des clés est ordonné, le parcours des clés ne respecte pas en général ce caractère.

Structures courantes

- L'implémentation la plus simple est la *liste d'association* (liste de couples clé, valeur)

```
1 l = [("toto",45),(53,89), ("XX",203.5)]
```

  Insertion en $O(1)$, recherche en $O(n)$.

Structures courantes

- L'implémentation la plus simple est la *liste d'association* (liste de couples clé, valeur)

```
1 l = [("toto",45),(53,89), ("XX",203.5)]
```

  Insertion en $O(1)$, recherche en $O(n)$.

- Les dictionnaires sont le plus souvent utilisés lorsque l'opération de recherche est la plus fréquente. Pour cette raison, la conception privilégie fréquemment cette opération, au détriment de l'efficacité de l'ajout et de l'occupation mémoire par rapport à d'autres structures de données.

Structures courantes

- L'implémentation la plus simple est la *liste d'association* (liste de couples clé, valeur)

```
1 l = [("toto",45),(53,89), ("XX",203.5)]
```

  Insertion en $O(1)$, recherche en $O(n)$.
- Les dictionnaires sont le plus souvent utilisés lorsque l'opération de recherche est la plus fréquente. Pour cette raison, la conception privilégie fréquemment cette opération, au détriment de l'efficacité de l'ajout et de l'occupation mémoire par rapport à d'autres structures de données.
- Deux structures sont particulièrement efficaces pour représenter les tableaux associatifs : la *table de hachage* et l'*arbre équilibré* (cette dernière notion n'est pas explicitement au programme d'ITC).

Implémentation

Comparaison des complexités temporelles

Opérations	Tables de hachage	Arbre équilibré
Insertion	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.
recherche	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.

- En général, les tables de hachage ont une représentation plus compacte en mémoire.

Implémentation

Comparaison des complexités temporelles

Opérations	Tables de hachage	Arbre équilibré
Insertion	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.
recherche	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.

- En général, les tables de hachage ont une représentation plus compacte en mémoire.
- Les tables de hachage imposent la création (souvent difficile) d'une *fonction de hachage*, les arbres équilibrés ont juste besoin d'un ordre total sur les clefs.

Implémentation

Comparaison des complexités temporelles

Opérations	Tables de hachage	Arbre équilibré
Insertion	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.
recherche	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.

- En général, les tables de hachage ont une représentation plus compacte en mémoire.
- Les tables de hachage imposent la création (souvent difficile) d'une *fonction de hachage*, les arbres équilibrés ont juste besoin d'un ordre total sur les clefs.
- L'ensemble des clés n'a pas besoin d'un ordre total pour les tables de hachage.

Implémentation

Comparaison des complexités temporelles

Opérations	Tables de hachage	Arbre équilibré
Insertion	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.
recherche	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.

- En général, les tables de hachage ont une représentation plus compacte en mémoire.
- Les tables de hachage imposent la création (souvent difficile) d'une *fonction de hachage*, les arbres équilibrés ont juste besoin d'un ordre total sur les clefs.
- L'ensemble des clés n'a pas besoin d'un ordre total pour les tables de hachage.
- Les arbres équilibrés s'implémentent bien avec des structures de données persistantes.
  Les tables de hachage utilisent un tableau (donc mutable).

Implémentation

Comparaison des complexités temporelles

Opérations	Tables de hachage	Arbre équilibré
Insertion	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.
recherche	$O(1)$ moyenne, $O(n)$ pire	$O(\log n)$ moyenne et pire.

- En général, les tables de hachage ont une représentation plus compacte en mémoire.
- Les tables de hachage imposent la création (souvent difficile) d'une *fonction de hachage*, les arbres équilibrés ont juste besoin d'un ordre total sur les clefs.
- L'ensemble des clés n'a pas besoin d'un ordre total pour les tables de hachage.
- Les arbres équilibrés s'implémentent bien avec des structures de données persistantes.
  Les tables de hachage utilisent un tableau (donc mutable).
- En `Python`, les dictionnaires sont implémentés par des tables de hachage.

Un bémol sur la complexité temporelle

- En coût amorti, les dictionnaires implantés par tables de hachages (comme étudiés dans le cours) ont une insertion et une recherche en $O(1)$ en *coût amorti* dans un monde merveilleux sans collision.

Un bémol sur la complexité temporelle

- En coût amorti, les dictionnaires implantés par tables de hachages (comme étudiés dans le cours) ont une insertion et une recherche en $O(1)$ en *coût amorti* dans un monde merveilleux sans collision.
- Cette affirmation doit être relativisée :

Un bémol sur la complexité temporelle

- En coût amorti, les dictionnaires implantés par tables de hachages (comme étudiés dans le cours) ont une insertion et une recherche en $O(1)$ en *coût amorti* dans un monde merveilleux sans collision.
- Cette affirmation doit être relativisée :
  - D'abord le hachage lui-même de la clé peut être coûteux (en général de l'ODG de la taille de la clé). Sauf si (comme c'est souvent le cas), on travaille avec des clés de tailles bornées.

Un bémol sur la complexité temporelle

- En coût amorti, les dictionnaires implantés par tables de hachages (comme étudiés dans le cours) ont une insertion et une recherche en $O(1)$ en *coût amorti* dans un monde merveilleux sans collision.
- Cette affirmation doit être relativisée :
  - D'abord le hachage lui-même de la clé peut être coûteux (en général de l'ODG de la taille de la clé). Sauf si (comme c'est souvent le cas), on travaille avec des clés de tailles bornées.
  - Dans le cas d'une gestion des collisions par listes chaînées ou par sondage linéaire, si le hachage ne suit pas une loi de distribution uniforme, la complexité des insertions et recherches peut être alourdie.

Fonctionnement d'un tableau

- Considérons une liste L Python dont les éléments sont notés **E0,E1,E2,....**

Fonctionnement d'un tableau

- Considérons une liste *L* Python dont les éléments sont notés **E0,E1,E2,....**
- Les adresses des éléments de **L** (des *mots binaires* de 8 ou 4 octets) sont enregistrées dans un tableau d'adresse. Ces mots/adresses occupent des places contiguës en mémoire.

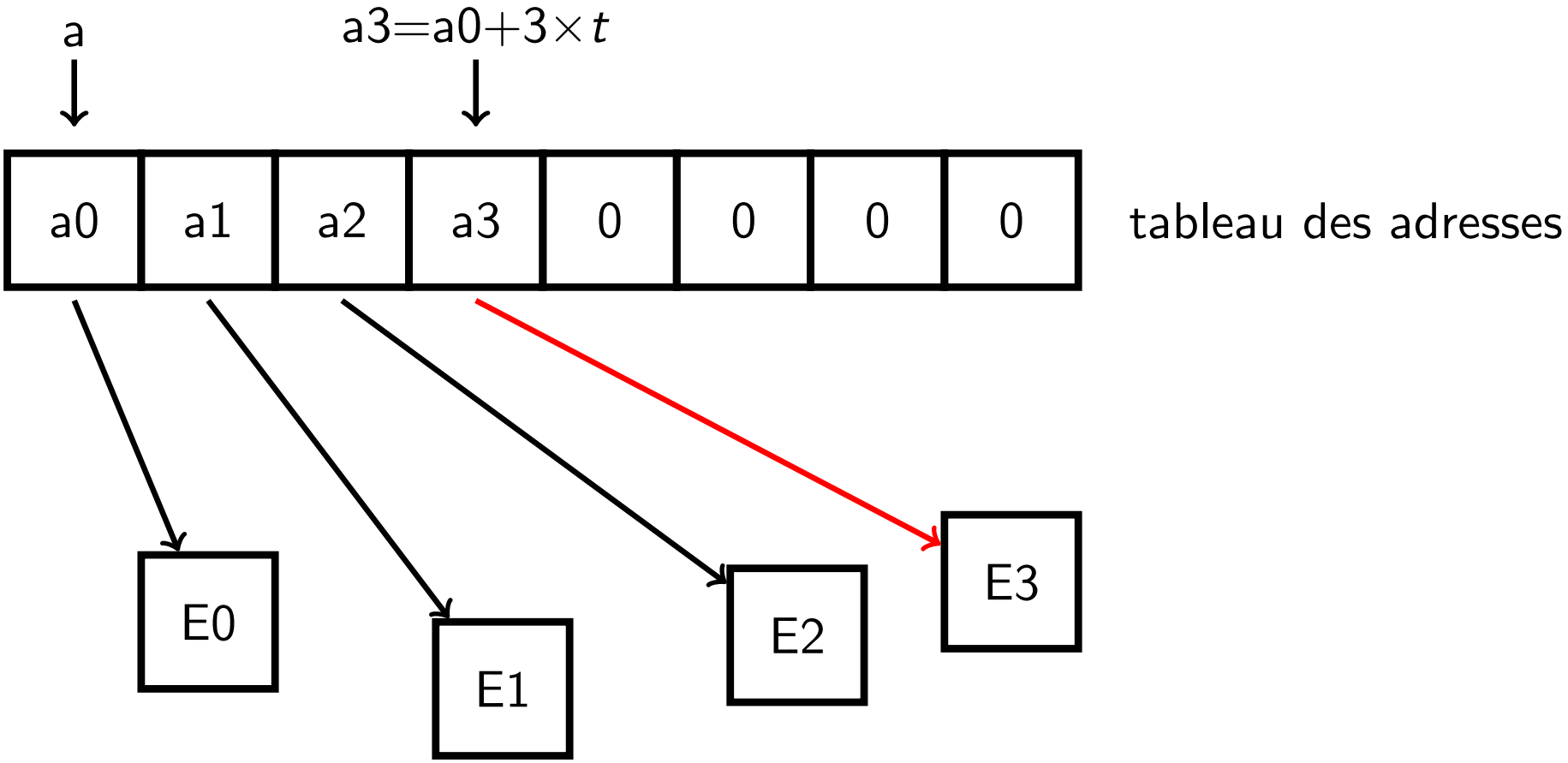
Fonctionnement d'un tableau

- Considérons une liste *L* Python dont les éléments sont notés **E0,E1,E2,....**
- Les adresses des éléments de **L** (des *mots binaires* de 8 ou 4 octets) sont enregistrées dans un tableau d'adresse. Ces mots/adresses occupent des places contiguës en mémoire.
- Python connaît l'adresse du début du tableau des adresses de la liste **L**. Il retrouve n'importe quel élément par une opération simple. Par exemple il trouve l'adresse de **E3** en calculant

$$\textbf{adresse de E0} + 3 \times \textbf{taille d'une adresse}$$

Dans la suite on note **a0** l'adresse de **E0** et **t** la taille d'une adresse.

Trouver le 3ème élément d'une liste **L**



Lorsqu'on entre **L[3]**, Python accède à l'adresse où est stockée **E3** en ajoutant 3 fois la taille d'une adresse à l'adresse **a0**. Python retourne alors le contenu trouvé à l'adresse **a3**.

Objectif et fonctionnement

- Une *fonction de hachage* répartit les paires clé–valeur dans un tableau d'*alvéoles*.

Objectif et fonctionnement

- Une *fonction de hachage* répartit les paires clé–valeur dans un tableau d'*alvéoles*.
- Elle transforme une clé en une valeur de hachage, ce qui donne la position dans un tableau.

Objectif et fonctionnement

- Une *fonction de hachage* répartit les paires clé–valeur dans un tableau d'*alvéoles*.
- Elle transforme une clé en une valeur de hachage, ce qui donne la position dans un tableau.
- Son calcul se fait parfois en deux temps :

Objectif et fonctionnement

- Une *fonction de hachage* répartit les paires clé–valeur dans un tableau d'*alvéoles*.
- Elle transforme une clé en une valeur de hachage, ce qui donne la position dans un tableau.
- Son calcul se fait parfois en deux temps :
  - Une fonction de hachage particulière à l'application est utilisée pour produire un nombre entier à partir de la clé ;

Objectif et fonctionnement

- Une *fonction de hachage* répartit les paires clé–valeur dans un tableau d'*alvéoles*.
- Elle transforme une clé en une valeur de hachage, ce qui donne la position dans un tableau.
- Son calcul se fait parfois en deux temps :
  - Une fonction de hachage particulière à l'application est utilisée pour produire un nombre entier à partir de la clé ;
  - Ce nombre entier est converti en une position possible de la table, en général en calculant le reste modulo la taille de la table.

Objectif et fonctionnement

- Une *fonction de hachage* répartit les paires clé–valeur dans un tableau d'*alvéoles*.
- Elle transforme une clé en une valeur de hachage, ce qui donne la position dans un tableau.
- Son calcul se fait parfois en deux temps :
  - Une fonction de hachage particulière à l'application est utilisée pour produire un nombre entier à partir de la clé ;
  - Ce nombre entier est converti en une position possible de la table, en général en calculant le reste modulo la taille de la table.
- Si la clé n'est pas un entier naturel, on fait en sorte de la considérer comme telle. Par exemple, pour les chaînes de caractères :

Objectif et fonctionnement

- Une *fonction de hachage* répartit les paires clé–valeur dans un tableau d'*alvéoles*.
- Elle transforme une clé en une valeur de hachage, ce qui donne la position dans un tableau.
- Son calcul se fait parfois en deux temps :
  - Une fonction de hachage particulière à l'application est utilisée pour produire un nombre entier à partir de la clé ;
  - Ce nombre entier est converti en une position possible de la table, en général en calculant le reste modulo la taille de la table.
- Si la clé n'est pas un entier naturel, on fait en sorte de la considérer comme telle. Par exemple, pour les chaînes de caractères :
  - on considère chaque caractère comme un nombre (par exemple avec le code ASCII) ;

Objectif et fonctionnement

- Une *fonction de hachage* répartit les paires clé–valeur dans un tableau d'*alvéoles*.
- Elle transforme une clé en une valeur de hachage, ce qui donne la position dans un tableau.
- Son calcul se fait parfois en deux temps :
  - Une fonction de hachage particulière à l'application est utilisée pour produire un nombre entier à partir de la clé ;
  - Ce nombre entier est converti en une position possible de la table, en général en calculant le reste modulo la taille de la table.
- Si la clé n'est pas un entier naturel, on fait en sorte de la considérer comme telle. Par exemple, pour les chaînes de caractères :
  - on considère chaque caractère comme un nombre (par exemple avec le code ASCII) ;
  - puis on le combine tous les nombres obtenus par une fonction rapide (souvent le XOR -OU EXCLUSIF-).
    Les divisions sont à éviter en raison de leur relative lenteur sur certaines machines.

Présentation simplifiée

- On dispose donc d'une fonction *h* dite de *hachage* qui va de l'ensemble des clés vers l'ensemble des indexes d'un tableau de couples (clés,valeurs) (ces derniers sont appelés *alvéoles*).

Présentation simplifiée

- On dispose donc d'une fonction h dite de *hachage* qui va de l'ensemble des clés vers l'ensemble des indexes d'un tableau de couples (clés,valeurs) (ces derniers sont appelés *alvéoles*).
- Lorsqu'on ajoute une nouvelle association (John Smith,+1555-8976), la valeur h(John Smith) est calculée. C'est l'indice d'une case du tableau d'association.

Présentation simplifiée

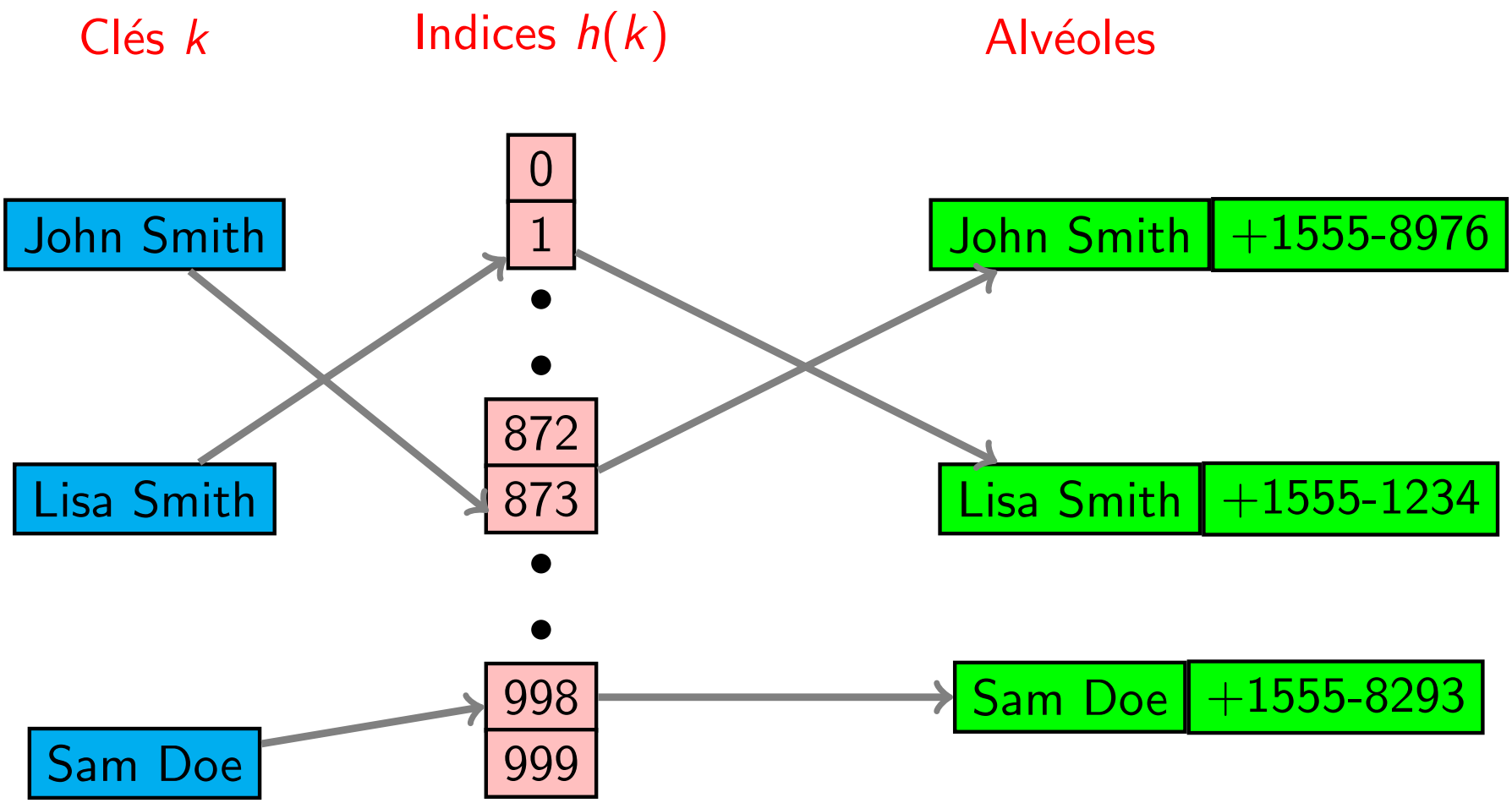
- On dispose donc d'une fonction h dite de *hachage* qui va de l'ensemble des clés vers l'ensemble des indexes d'un tableau de couples (clés,valeurs) (ces derniers sont appelés *alvéoles*).
- Lorsqu'on ajoute une nouvelle association (John Smith,+1555-8976), la valeur h(John Smith) est calculée. C'est l'indice d'une case du tableau d'association.
- A la case h(John Smith), on met le tuple (John Smith,+1555-8976).

Présentation simplifiée

- On dispose donc d'une fonction h dite de *hachage* qui va de l'ensemble des clés vers l'ensemble des indexes d'un tableau de couples (clés,valeurs) (ces derniers sont appelés *alvéoles*).
- Lorsqu'on ajoute une nouvelle association (John Smith,+1555-8976), la valeur h(John Smith) est calculée. C'est l'indice d'une case du tableau d'association.
- A la case h(John Smith), on met le tuple (John Smith,+1555-8976).
- Si la fonction de hachage est injective (ce qui n'arrive presque jamais), deux clés différentes ont deux valeurs de hachage différentes et on n'a aucun problème de collision.

Exemple d'un annuaire

Cas sans collision



Gestion des collisions

Par chaînage

- On gère un tableau de liste chaînées.

Gestion des collisions

Par chaînage

- On gère un tableau de liste chaînées.
- Les alvéoles ne sont plus des tuples (clé,valeur) mais des listes chaînées de tuples (clés,valeurs). Ces listes sont vides par défaut.

Gestion des collisions

Par chaînage

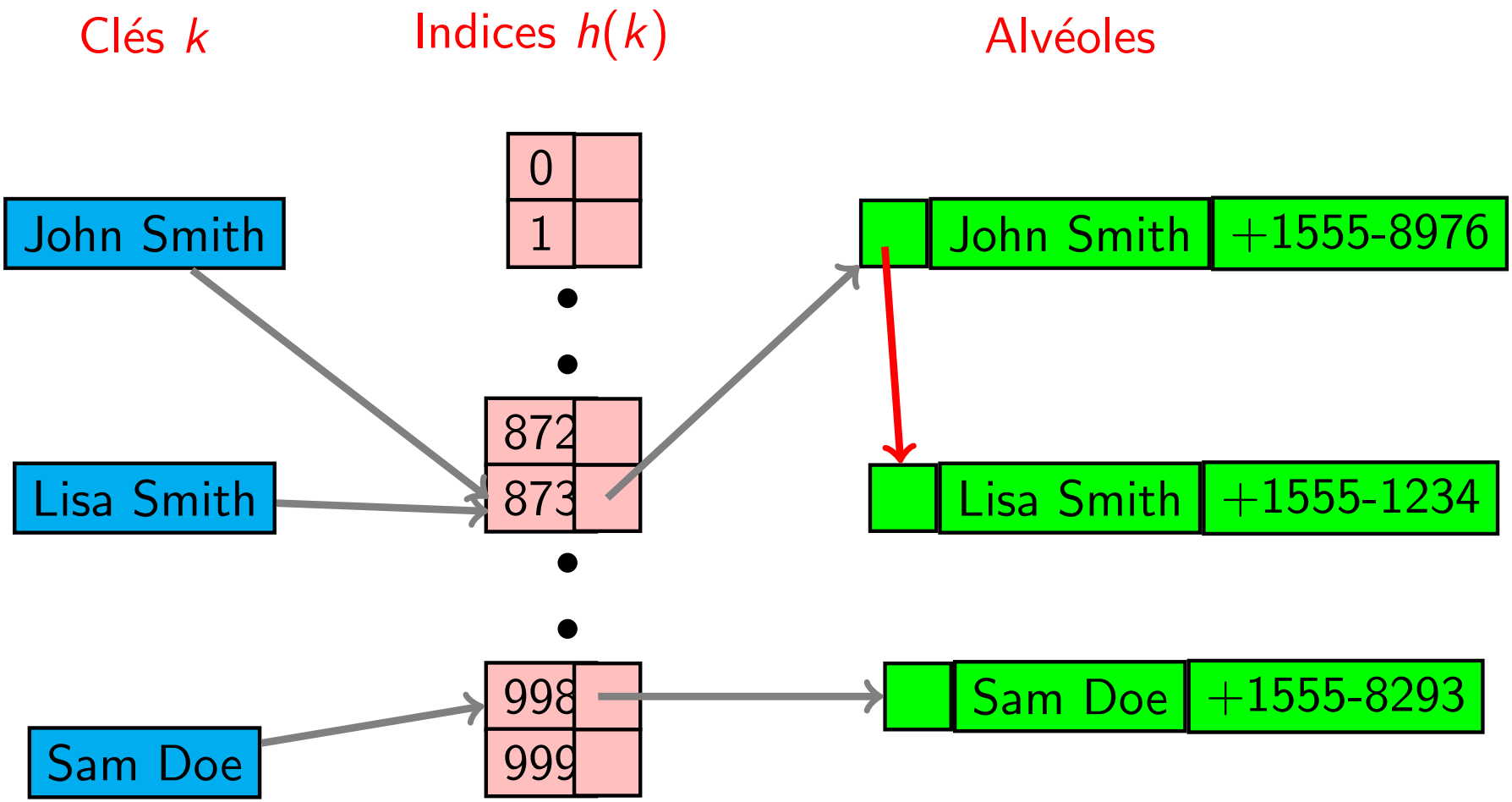
- On gère un tableau de liste chaînées.
- Les alvéoles ne sont plus des tuples (clé,valeur) mais des listes chaînées de tuples (clés,valeurs). Ces listes sont vides par défaut.
- Lors de l'ajout d'une nouvelle association comme (B,230), on cherche l'avéole à l'indice $h(B)$ du tableau des alvéoles. c'est une liste chaînée. On ajoute au bout de cette liste chaînée l'association (B,230).

Gestion des collisions

Par chaînage

- On gère un tableau de liste chaînées.
- Les alvéoles ne sont plus des tuples (clé,valeur) mais des listes chaînées de tuples (clés,valeurs). Ces listes sont vides par défaut.
- Lors de l'ajout d'une nouvelle association comme (B,230), on cherche l'avéole à l'indice $h(B)$ du tableau des alvéoles. c'est une liste chaînée. On ajoute au bout de cette liste chaînée l'association (B,230).
- Dans une recherche, le pire cas se produit quand la fonction de hachage associe un même indice à de nombreuses clés. Pour rechercher une clé particulière, il faut alors parcourir toute la liste chaînée de l'alvéole.

Gestion des collisions par chaînage



Gestion des collisions par adressage ouvert

- Les enregistrements (clés,valeurs) sont stockés dans un tableau.

Gestion des collisions par adressage ouvert

- Les enregistrements (clés,valeurs) sont stockés dans un tableau.
- Lorsqu'on insère l'association (k, v), on vérifie si la case $h(k)$ du tableau est libre.

Gestion des collisions par adressage ouvert

- Les enregistrements (clés,valeurs) sont stockés dans un tableau.
- Lorsqu'on insère l'association (k, v), on vérifie si la case $h(k)$ du tableau est libre.
  - Si c'est le cas, tout va bien, on place l'association case $h(k)$.

Gestion des collisions par adressage ouvert

- Les enregistrements (clés,valeurs) sont stockés dans un tableau.
- Lorsqu'on insère l'association (k, v), on vérifie si la case $h(k)$ du tableau est libre.
  - Si c'est le cas, tout va bien, on place l'association case $h(k)$.
  - Sinon, on part de la case $h(k)$ et on explore le tableau selon une certaine *fonction de sondage*.

Gestion des collisions par adressage ouvert

- Les enregistrements (clés,valeurs) sont stockés dans un tableau.
- Lorsqu'on insère l'association (k, v), on vérifie si la case $h(k)$ du tableau est libre.
  - Si c'est le cas, tout va bien, on place l'association case $h(k)$.
  - Sinon, on part de la case $h(k)$ et on explore le tableau selon une certaine *fonction de sondage*.
    - La plus simple de ces fonctions consiste à exlorer d'abord la case $h(k) + 1$ puis la case $h(k) + 2$ etc. On l'appelle *sondage linéaire* (l'intervalle entre deux cases explorées est constant).
      Bien sûr, il faut travailler modulo le nombre de cases dans le tableau.

Gestion des collisions par adressage ouvert

- Les enregistrements (clés,valeurs) sont stockés dans un tableau.
- Lorsqu'on insère l'association (k, v), on vérifie si la case $h(k)$ du tableau est libre.
  - Si c'est le cas, tout va bien, on place l'association case $h(k)$.
  - Sinon, on part de la case $h(k)$ et on explore le tableau selon une certaine *fonction de sondage*.
    - La plus simple de ces fonctions consiste à exlorer d'abord la case $h(k) + 1$ puis la case $h(k) + 2$ etc. On l'appelle *sondage linéaire* (l'intervalle entre deux cases explorées est constant).
      Bien sûr, il faut travailler modulo le nombre de cases dans le tableau.
    - Autre méthode, le *sondage quadratique* : l'écart entre deux sondages augmente linéairement. Encore une fois, travailler modulo.

Gestion des collisions par adressage ouvert

Cas d'un sondage linéaire (Insertion)

- On insère d'abord John Smith et son numéro. La valeur de hachage est 152. Pas de problème.

Gestion des collisions par adressage ouvert

Cas d'un sondage linéaire (Insertion)

- On insère d'abord John Smith et son numéro. La valeur de hachage est 152. Pas de problème.
- On veut insérer ensuite Sandra Dee et son numéro. Malheureusement, la valeur de hachage est aussi 152 : il y a collision. On ajoute donc les coordonnées de Sandra à la première case libre après 152, soit 153.

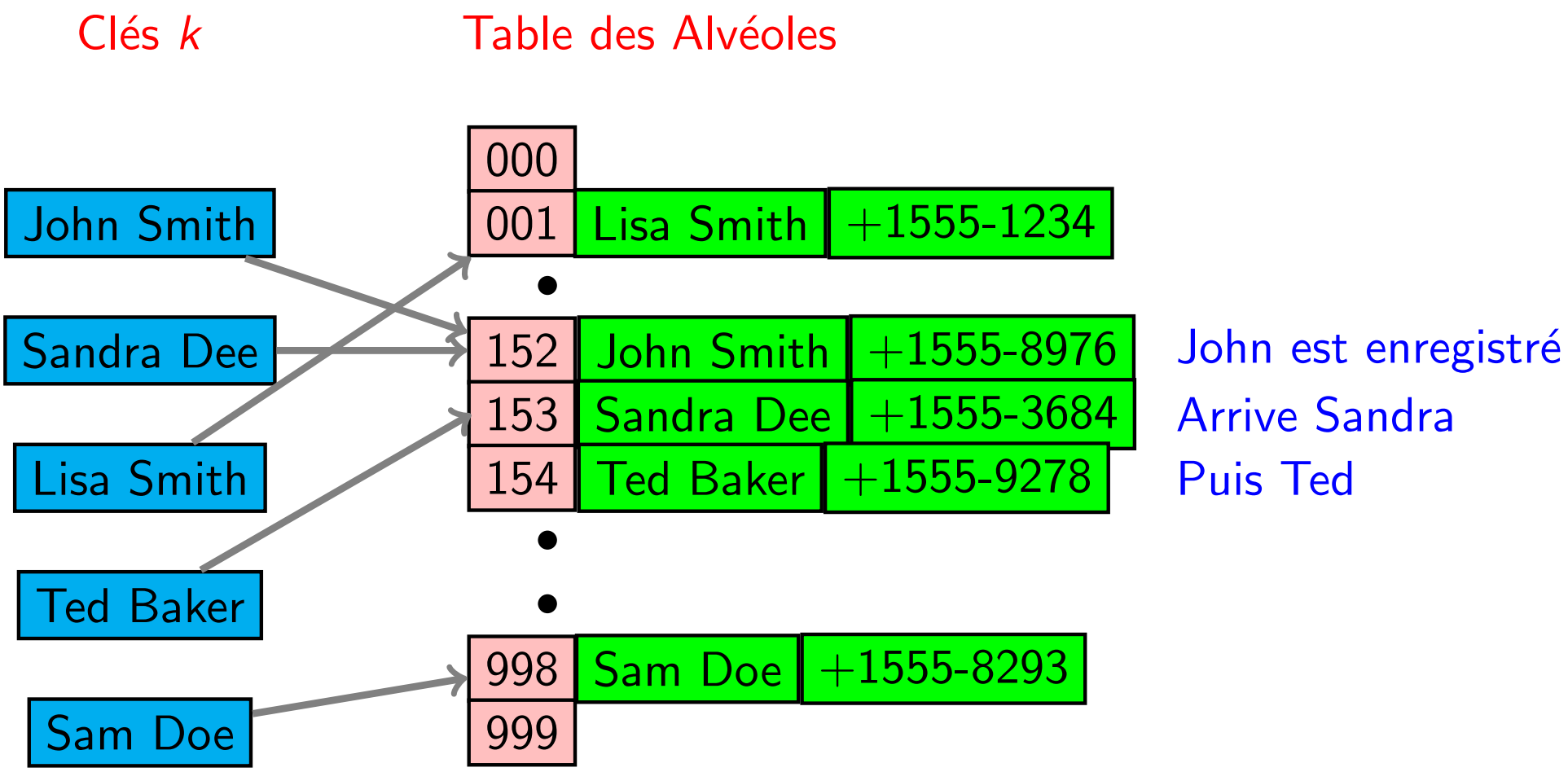
Gestion des collisions par adressage ouvert

Cas d'un sondage linéaire (Insertion)

- On insère d'abord John Smith et son numéro. La valeur de hachage est 152. Pas de problème.
- On veut insérer ensuite Sandra Dee et son numéro. Malheureusement, la valeur de hachage est aussi 152 : il y a collision. On ajoute donc les coordonnées de Sandra à la première case libre après 152, soit 153.
- Arrive Ted Baker. La valeur de hachage associée est 153. Cette valeur est bien unique mais la case correspondante est hélas occupée par Sandra (laquelle a été déplacée par rapport à sa valeur de hachage). On cherche donc pour Ted, la prochaine case vide, soit 154.

Gestion des collisions par adressage ouvert

Cas d'un sondage linéaire (Wikipedia)



Gestion des collisions par adressage ouvert

Cas d'un sondage linéaire (Recherche)

- On cherche les coordonnées de Ted. La valeur de hachage est 153.

Gestion des collisions par adressage ouvert

Cas d'un sondage linéaire (Recherche)

- On cherche les coordonnées de Ted. La valeur de hachage est 153.
- En inspectant l'alvéole de la case 153, on se rend compte que la clé stockée est Sandra Dee et non Ted Baker.

Gestion des collisions par adressage ouvert

Cas d'un sondage linéaire (Recherche)

- On cherche les coordonnées de Ted. La valeur de hachage est 153.
- En inspectant l'alvéole de la case 153, on se rend compte que la clé stockée est Sandra Dee et non Ted Baker.
- On examine donc la prochaine case selon le sondage linéaire (ici, case courante + 1). Cette fois-ci, la clé stockée est bien Ted Baker. On renvoie son numéro.

Sondage quadratique et double sondage

D'après Wikipedia

- Sondage quadratique : l'intervalle entre les alvéoles augmente linéairement (les indices des alvéoles augmentent donc quadratiquement), ce qui peut s'exprimer par la formule :

$$h_i(x) = \left(h(x) + (-1)^{i+1} \cdot \left\lceil \frac{i}{2} \right\rceil^2 \right) \bmod k$$

où k est le nombre d'alvéoles et i le numéro de la tentative (les $i-1$ premières tentatives ont échoué à trouver une case vide) ;

Sondage quadratique et double sondage

D'après Wikipedia

- Sondage quadratique : l'intervalle entre les alvéoles augmente linéairement (les indices des alvéoles augmentent donc quadratiquement), ce qui peut s'exprimer par la formule :

$$h_i(x) = \left(h(x) + (-1)^{i+1} \cdot \left\lceil \frac{i}{2} \right\rceil^2 \right) \bmod k$$

  où k est le nombre d'alvéoles et i le numéro de la tentative (les $i-1$ premières tentatives ont échoué à trouver une case vide) ;
- le double hachage : l'indice de l'alvéole est donné par une deuxième fonction de hachage, ou *hachage secondaire*. En cas de collision en case $h(k)$ on calcule la nouvelle position en $h_2(h(k))$.

Comparaison des performances

- Le sondage linéaire possède la meilleure performance en termes de cache, mais est sensible à l'effet de grumelage décrit plus haut.

Comparaison des performances

- Le sondage linéaire possède la meilleure performance en termes de cache, mais est sensible à l'effet de grumelage décrit plus haut.
- Le double hachage ne permet pas d'utiliser le cache efficacement, mais permet de réduire presque complètement ce grumelage, au prix d'une complexité plus élevée.

Comparaison des performances

- Le sondage linéaire possède la meilleure performance en termes de cache, mais est sensible à l'effet de grumelage décrit plus haut.
- Le double hachage ne permet pas d'utiliser le cache efficacement, mais permet de réduire presque complètement ce grumelage, au prix d'une complexité plus élevée.
- Le sondage quadratique se situe entre le linéaire et le double hachage au niveau des performances.

Choix d'une bonne fonction de hachage

- Pour de bonnes performances il faut trouver un compromis entre

Choix d'une bonne fonction de hachage

- Pour de bonnes performances il faut trouver un compromis entre
  - la rapidité du calcul du hachage

Choix d'une bonne fonction de hachage

- Pour de bonnes performances il faut trouver un compromis entre
  - la rapidité du calcul du hachage
  - La taille à réserver pour l'espace de hachage,

Choix d'une bonne fonction de hachage

- Pour de bonnes performances il faut trouver un compromis entre
  - la rapidité du calcul du hachage
  - La taille à réserver pour l'espace de hachage,
  - La réduction du risque de collisions

Choix d'une bonne fonction de hachage

- Pour de bonnes performances il faut trouver un compromis entre
  - la rapidité du calcul du hachage
  - La taille à réserver pour l'espace de hachage,
  - La réduction du risque de collisions
- Prendre un nombre premier comme taille de table de hachage évite les problèmes de diviseurs communs et donc de collisions.
  Prendre une puissance de 2 comme taille permet un calcul modulo rapide.

Choix d'une bonne fonction de hachage

- Pour de bonnes performances il faut trouver un compromis entre
  - la rapidité du calcul du hachage
  - La taille à réserver pour l'espace de hachage,
  - La réduction du risque de collisions
- Prendre un nombre premier comme taille de table de hachage évite les problèmes de diviseurs communs et donc de collisions.
  Prendre une puissance de 2 comme taille permet un calcul modulo rapide.
- Il y a *grumelage* quand les valeurs de hachage se retrouvent côte à côte dans la table (cf exemple avec Ted Baker). Pour l'éviter :

Choix d'une bonne fonction de hachage

- Pour de bonnes performances il faut trouver un compromis entre
  - la rapidité du calcul du hachage
  - La taille à réserver pour l'espace de hachage,
  - La réduction du risque de collisions
- Prendre un nombre premier comme taille de table de hachage évite les problèmes de diviseurs communs et donc de collisions.
  Prendre une puissance de 2 comme taille permet un calcul modulo rapide.
- Il y a *grumelage* quand les valeurs de hachage se retrouvent côte à côte dans la table (cf exemple avec Ted Baker). Pour l'éviter :
  - Privilégier les fonctions de hachage avec une distribution uniforme des valeurs de hachage.

Choix d'une bonne fonction de hachage

- Pour de bonnes performances il faut trouver un compromis entre
  - la rapidité du calcul du hachage
  - La taille à réserver pour l'espace de hachage,
  - La réduction du risque de collisions
- Prendre un nombre premier comme taille de table de hachage évite les problèmes de diviseurs communs et donc de collisions.
  Prendre une puissance de 2 comme taille permet un calcul modulo rapide.
- Il y a *grumelage* quand les valeurs de hachage se retrouvent côte à côte dans la table (cf exemple avec Ted Baker). Pour l'éviter :
  - Privilégier les fonctions de hachage avec une distribution uniforme des valeurs de hachage.
  - Le rapport $\mathrm{fc} = \frac{n}{k}$ où n est le nombre de tuples clés-valeurs et k la capacité du tableau est appelé *facteur de compression*.
    Si $\mathrm{fc} > \frac{1}{3}$, le risque de collision augmente.

Choix d'une bonne fonction de hachage

- Pour de bonnes performances il faut trouver un compromis entre
  - la rapidité du calcul du hachage
  - La taille à réserver pour l'espace de hachage,
  - La réduction du risque de collisions
- Prendre un nombre premier comme taille de table de hachage évite les problèmes de diviseurs communs et donc de collisions.
  Prendre une puissance de 2 comme taille permet un calcul modulo rapide.
- Il y a *grumelage* quand les valeurs de hachage se retrouvent côte à côte dans la table (cf exemple avec Ted Baker). Pour l'éviter :
  - Privilégier les fonctions de hachage avec une distribution uniforme des valeurs de hachage.
  - Le rapport $\mathrm{fc} = \frac{n}{k}$ où n est le nombre de tuples clés-valeurs et k la capacité du tableau est appelé *facteur de compression*.
    Si $\mathrm{fc} > \frac{1}{3}$, le risque de collision augmente.
  - En `Python`, redimensionnement du tableau dès que le facteur de compression dépasse 2/3. La capacité est alors multipliée par 2.

La fonction `hash`

- La fonction `hash` code les clés (Les clés doivent être immutables -tuples, chaînes de caractères...-).

La fonction `hash`

- La fonction `hash` code les clés (Les clés doivent être immutables -tuples, chaînes de caractères...-).
- Puis, `Python` calcule `hash(clé)%n` où **n** est la taille du tableau sous-jacent.

La fonction `hash`

- La fonction `hash` code les clés (Les clés doivent être immutables -tuples, chaînes de caractères...-).
- Puis, `Python` calcule `hash(clé)%n` où **n** est la taille du tableau sous-jacent.
- Deux clés considérées comme égales par `==` ont la même valeur de hash. Ainsi

Listing 3 – Python Console

```
>>> hash(1)==hash(1.0)
True
```

Taille du tableau sous-jacent

- En `Python`, on redimensionne le dictionnaire si le facteur de compression dépasse 2/3

Taille du tableau sous-jacent

- En `Python`, on redimensionne le dictionnaire si le facteur de compression dépasse 2/3
- Les tailles sont des puissances de 2

Taille du tableau sous-jacent

- En `Python`, on redimensionne le dictionnaire si le facteur de compression dépasse $2/3$
- Les tailles sont des puissances de 2
  - De 1 à 5 éléments la capacité est $2^3 = 8$ car $\frac{3}{2} \times 5 = 7.5 < 2^3$; en revanche $\frac{3}{2} \times 6 = 9$ (redimensionner).

Taille du tableau sous-jacent

- En `Python`, on redimensionne le dictionnaire si le facteur de compression dépasse $2/3$
- Les tailles sont des puissances de 2
  - De 1 à 5 éléments la capacité est $2^3 = 8$ car $\frac{3}{2} \times 5 = 7.5 < 2^3$; en revanche $\frac{3}{2} \times 6 = 9$ (redimensionner).
  - De 6 jusqu'à 10 éléments la capacité est 16 puisque $\frac{3}{2} \times 10 = 15 < 2^4$; mais $\frac{3}{2} \times 11 = 16.5$

Taille du tableau sous-jacent

- En `Python`, on redimensionne le dictionnaire si le facteur de compression dépasse $2/3$
- Les tailles sont des puissances de 2
  - De 1 à 5 éléments la capacité est $2^3 = 8$ car $\frac{3}{2} \times 5 = 7.5 < 2^3$; en revanche $\frac{3}{2} \times 6 = 9$ (redimensionner).
  - De 6 jusqu'à 10 éléments la capacité est 16 puisque $\frac{3}{2} \times 10 = 15 < 2^4$; mais $\frac{3}{2} \times 11 = 16.5$
  - Ona $\frac{2}{3} \times 32 = 21.3$: donc à partir de 22 la capacité passe à 64

Calcul du modulo

- La capacité du dictionnaire est de la forme $n = 2^p$

Calcul du modulo

- La capacité du dictionnaire est de la forme $n = 2^p$
- $2^p - 1$ s'écrit en binaire comme un mot avec p lettres 1

Calcul du modulo

- La capacité du dictionnaire est de la forme $n = 2^p$
- $2^p - 1$ s'écrit en binaire comme un mot avec p lettres 1
- `h%n` et `h&(n-1)` ont la même valeur (`&` désigne le ET bit à bit).

Calcul du modulo

- La capacité du dictionnaire est de la forme $n = 2^p$
- $2^p - 1$ s'écrit en binaire comme un mot avec p lettres 1
- `h%n` et `h&(n-1)` ont la même valeur (`&` désigne le ET bit à bit).
- Par exemple, si on crée un dictionnaire avec un élément, la capacité est 8. On calcule la position pour insérer un nouvel élément de clé `c` par `hash(c)%8` ou encore `h(c)&7` .

PropriétéS de `hash`

- Si $i \in \mathbb{N}$, $i \neq -1$ et $-2^{61}+1 < i < 2^{61}+1$, alors `hash(i)` vaut i.

  `hash(2**61-1)` et `hash(-2**61+1)` valent 0, `hash(-1)` vaut -2

PropriétéS de `hash`

- Si $i \in \mathbb{N}$, $i \neq -1$ et $-2^{61}+1 < i < 2^{61}+1$, alors `hash(i)` vaut i. `hash(2**61-1)` et `hash(-2**61+1)` valent 0, `hash(-1)` vaut -2
- Si x est un flottant égal à p/q alors `hash(x)` vaut `int(p*M/q)%M` avec $M = 2^{61} - 1$ (on l'appelle le *modulus*).

PropriétéS de `hash`

- Si $i \in \mathbb{N}$, $i \neq -1$ et $-2^{61}+1 < i < 2^{61}+1$, alors `hash(i)` vaut i. `hash(2**61-1)` et `hash(-2**61+1)` valent 0, `hash(-1)` vaut -2
- Si x est un flottant égal à p/q alors `hash(x)` vaut `int(p*M/q)%M` avec $M = 2^{61} - 1$ (on l'appelle le *modulus*).
- Si la clé est une chaîne de caractères, la valeur de hachage

PropriétéS de `hash`

- Si $i \in \mathbb{N}$, $i \neq -1$ et $-2^{61}+1 < i < 2^{61}+1$, alors `hash(i)` vaut i.
  `hash(2**61-1)` et `hash(-2**61+1)` valent 0, `hash(-1)` vaut -2
- Si x est un flottant égal à p/q alors `hash(x)` vaut `int(p*M/q)%M` avec $M = 2^{61} - 1$ (on l'appelle le *modulus*).
- Si la clé est une chaîne de caractères, la valeur de hachage
  - est calculée à partir d'une fonction aléatoirement choisie à chaque ouverture de session,

PropriétéS de `hash`

- Si $i \in \mathbb{N}$, $i \neq -1$ et $-2^{61}+1 < i < 2^{61}+1$, alors `hash(i)` vaut i.
  `hash(2**61-1)` et `hash(-2**61+1)` valent 0, `hash(-1)` vaut -2
- Si x est un flottant égal à p/q alors `hash(x)` vaut `int(p*M/q)%M` avec $M = 2^{61} - 1$ (on l'appelle le *modulus*).
- Si la clé est une chaîne de caractères, la valeur de hachage
  - est calculée à partir d'une fonction aléatoirement choisie à chaque ouverture de session,
  - et a pour valeur un entier sur 64 bits.

La classe ou le type dict

En Python, un dictionnaire est une instance de la classe `dict`.

```
1 mydico=dict()#creation d'un dico vide
2 type(mydico)
```

```
dict
```

```
1 mydico={}#creation d'un dico vide
2 type(mydico)
```

```
dict
```

Ajout de clés et valeurs

- Ajouts successifs depuis un dictionnaire vide :

```
1 mydico={}#cree dico vide
2 mydico["ceciEstUneClef1"]="CeciEstUneValeur1"
3 mydico["blabla"]=132
4 mydico[125.36]=[1,2,3]
5 print(mydico)# clés et valeurs peuvent etre n'importe quoi
```

Ajout de clés et valeurs

- Ajouts successifs depuis un dictionnaire vide :

```
1 mydico={}#cree dico vide
2 mydico["ceciEstUneClef1"]="CeciEstUneValeur1"
3 mydico["blabla"]=132
4 mydico[125.36]=[1,2,3]
5 print(mydico)# clés et valeurs peuvent etre n'importe quoi
```

- Méthode 2

```
1 dico2={'ceciEstUneClef1': 'CeciEstUneValeur1',\
2     'blabla': 132, 125.36: [1, 2, 3]}
```

Accès

- Accès. On indique entre crochet la clé à laquelle on veut accéder :

```
1 mydico["blabla"]
```

```
123
```

Accès

- Accès. On indique entre crochet la clé à laquelle on veut accéder :

```
1 mydico["blabla"]
```

```
123
```

- Si la clé n'existe pas, une exception est soulevée :

Listing 5 – clé inexistante

```
1 mydico["cleFarfelue"]
```

```
---
KeyError                  Traceback (most recent call last)
<ipython-input-4-62aa81b7dbb7> in <module>()
      1 mydico["blabla"]
----> 2 mydico["cleFarfelue"]#exception KeyError
KeyError: 'cleFarfelue'
```

Complétion simultanée, suppression de clé

- On peut créer un dictionnaire complexe d'une seule commande :

```
1 placard = {"chemise":3, "pantalon":6, "tee-shirt":7}
```

- On supprime une clé et la valeur associée, sans retour de valeur :

```
1 del placard["chemise"]# pas de valeur retournee
2 placard
```

```
{'tee-shirt': 7, 'pantalon': 6}
```

- Suppression de clé avec retour de la valeur supprimée correspondante :

```
1 print(placard.pop('tee-shirt')) # une valeur retournee
2 placard
```

```
7
{'pantalon': 6}
```

Modifier

- Clé existante, la valeur correspondante est modifiée :

```
1 placard['pantalon']=19 # passe de 6 a 19 pantalons
```

```
{'pantalon': 19}
```

- Si la clé n'existe pas déjà, elle est créée :

```
1 placard['smoking']=2
2 placard
```

```
{'smoking': 2, 'pantalon': 19}
```

Valeurs hachables

```
1    d = {"toto":1}
2    l = [1,2,3]
3    d[l] = 3
```

```
---------------------------------------------
TypeError   Traceback (most recent call last)
Cell In[14], line 3
      1 d = {"toto":1}
      2 l = [1,2,3]
----> 3 d[l] = 3

TypeError: unhashable type: 'list'
```

Les clefs de dictionnaires doivent être *immuables*.

Parcours

Parcourir les clés :

- Avec la méthode `.keys`

```
1 for cle in placard.keys():
2   print("la cle ",cle, "correspond a la valeur ",placard[cle])
```

```
la cle  smoking correspond a la valeur 2
la cle  pantalon correspond a la valeur 19
```

Parcours

Parcourir les clés :

- Avec la méthode `.keys`

```
1 for cle in placard.keys():
2   print("la cle ",cle, "correspond a la valeur ",placard[cle])
```

```
la cle  smoking correspond a la valeur 2
la cle  pantalon correspond a la valeur 19
```

- En utilisant une syntaxe allégée :

Listing 7 – Effet similaire au précédent

```
1 for cle in placard:
2   print("la cle ",cle, "correspond a la valeur ",placard[cle])
```

Parcours

- Parcourir les valeurs :

```
1 inventaire={"pommes":30,"bières":1,"bananes":17}
2 for v in inventaire.values():
3     print(v, end=",")
```

```
30,1,17
```

- Tester une valeur :

```
1 for k,v in inventaire.items():
2     if v==1:
3         print("Attention, refaire le stock de {}".format(k))
```

```
Attention, refaire le stock de bières
```

Objectif

- On veut passer à une fonction un nombre arbitraire de variables (par exemple) numériques et en faire (par exemple) la somme.
- Exemple d'appel :

```
1 somme(p=1,q=3,r=2)
```

- On souhaite effectuer 1+3+2.
- `p,q,r` sont ce qu'on appelle des *paramètres nommés*. Ils sont passé à la fonction en arguments de la forme `nom_var=valeur`.
- On ne connaît pas à l'avance les nom, les valeur ni le nombre des paramètres nommés.

Récupérer les paramètres nommés

- Récupérer les paramètres nommés : faire précéder un nom quelconque par deux étoiles **.

  Listing 8 – Les paramètres nommés sont en fait stockés dans un dictionnaire

```
1 def recupere(**parametres_nommes):
2     print "paramètres nommés{}".format(parametres_nommes)
3 recupere(p=1,q=2)
```

paramètres nommés : {'q': 2, 'p': 1}

Exercice

Écrire la fonction `somme` qui fait la somme de ses paramètres nommés (ceux-ci étant en nombre arbitraire)

Somme

```
1 def somme(**args):
2     return sum([args[k] for k in args])
```

Listing 9 – Deux façons équivalentes d'appeler la fonction

```
1 somme(p=6, q=3), somme(**{'p':6,'q':3})
```

```
(9,9)
```

Construction par compréhension

Comme les listes, les dictionnaires peuvent se construire par compréhension.

Listing 10 – dictionnaire `plusDeTrois` construit par compréhension

```
1       bac_legumes={'bananes':6,'courgettes':2,'carottes':10}
2       plusDeTrois={c:v for (c,v) in bac_legumes.items() if v>2}
3       plusDeTrois
```

```
{'bananes': 6, 'carottes': 10}
```