Dictionnaires et tables de hachage

Qu'est-ce qu'un dictionnaire?

- Un dictionnaire est une structure de données gardant en mémoire des entrées de la forme (clé, élément)
- Le but d'un dictionnaire est d'être capable d'accéder rapidement à une entrée, en donnant la valeur de sa clé.
- Ici, contrairement aux files avec priorités, on n'a pas besoin d'une relation d'ordre pour comparer les clés.
  - Un dictionnaire ayant une relation d'ordre pour comparer ses clés est appelé un dictionnaire ordonné (à venir)
- Par contre, on doit pouvoir décider si deux clés sont égales.
  - On utilisera pour cela un testeur d'égalité (similaire au comparateur pour les files avec priorités)

Exemples:

1) Comptes de banque:

- Clé: # du compte
- Élément: un dossier contenant l'information sur le propriétaire du compte, une listes de transactions (retraits, dépôts)...

2) Dictionnaire itéractif:

- Clé: mot cherché
- Élément: la définition de ce mot

TAD: Testeur d'égalité

- Un testeur d'égalité est un type abstrait de données qui teste l'égalité de deux clés

- Quand un dictionnaire doit savoir si deux clés sont égales, il utilise un testeur d'égalité

- Opérations:

  - égal(a,b): Retourne vrai si a et b sont considérés égales et faux, sinon.

    Retourne une erreur si a et b ne peuvent être comparés

Exemple:

- Supposons que pour les besoins d'un algorithme en géométrie, un mathématicien impose que deux coordonnées du plan cartésien soient considérées égales, si elles ont la même coordonnée en x.

Testeur d'égalité entre points selon les spécifications du mathématicien:

```
/** Deux points sont considérés égaux s'ils ont la
    même coordonnée en x. */
public class ÉgalitéX implements TesteurÉgal {
   int xa, ya, xb, yb;
   public int égal(Object a, Object b) throws
     ClassCastException {
      xa = ((Point2D) a).getX();
      ya = ((Point2D) a).getY();
      xb = ((Point2D) b).getX();
      yb = ((Point2D) b).getY();
      if (xa = xb)
            return true;
      else
            return false;
   }
}
```

Point objects:

```
/** Class representing a point in the
    plane with integer coordinates */
public class Point2D          {
   protected int xc, yc; // coordinates
   public Point2D(int x, int y) {
      xc = x;
      yc = y;
  }
   public int getX() {
            return xc;
   }
   public int getY() {
            return yc;
   }
}
```

"Maps" (Fonctions)

- Une "map" (ou fonction) est un dictionnaire dans lequel l'insertion de plusieurs éléments ayant la même clé est interdite.

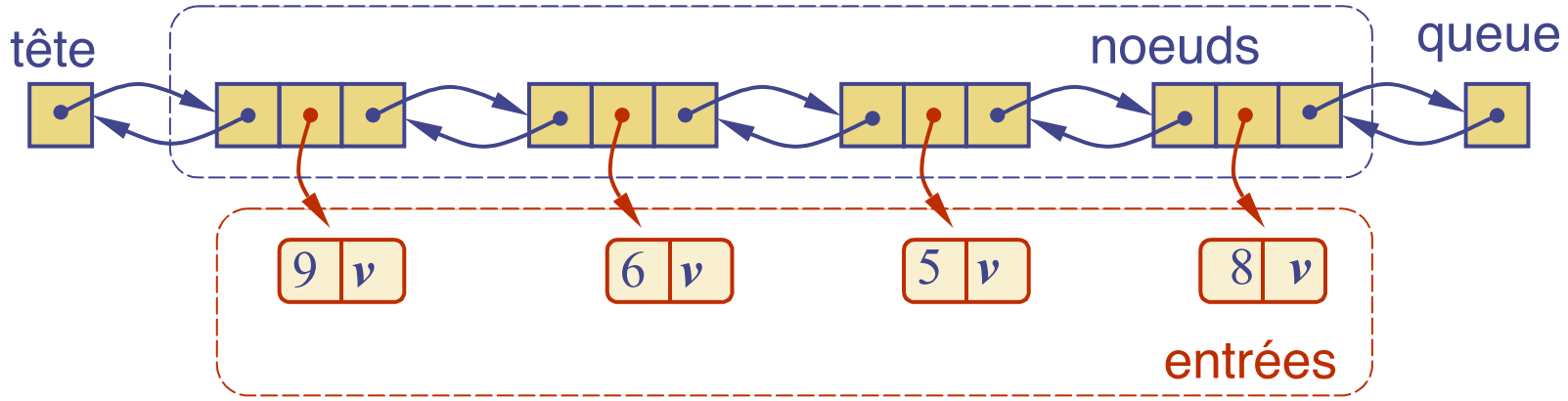
- Les opérations principales sur les "maps" sont celles de recherche, d'insertions et de suppressions d'éléments= (clé,élément)

- Exemples d'applications:
  - Carnet d'adresses
  - Base de données des relevés de notes des étudiants

- Opérations principales:
  - trouver(k): Si la "map" M a une entrée de clé k, retourne la valeur
    get(k): associée à cette entrée, sinon retoune NULL.
  - insérer(k,e): insérer l'entrée (k,e) dans M. Si la clé k n'était pas
    put(k,e): déjà dans M, retourner NULL, sinon retourner l'ancienne élément associé à k.
  - enlever(k): Si la "map" M a une entrée de clé k, l'enlever de M et
    remove(k): retourner l'élément associé, sinon retourner NULL
- Opérations auxiliaires:
  - taille(): Retourne le nombre d'entrées dans M
    size():
  - estVide(): Retourne VRAI si M est vide et FAUX, sinon
    isEmpty():

Implémentation sous forme d'une liste doublement chaînée

- On peut implémenter une "map" comme une liste non ordonnée, doublement chaînée
- On garde en mémoire la liste les entrées, dans un ordre arbitraire.



Performance de l'implémentation

- La complexité en temps des opérations trouver(k), insérer(k,e) et enlever (k) est O(n) étant donné que, dans le pire des cas, on traverse toute la liste pour trouver un entrée de clé k

- Cette implémentation est efficace seulement si on travaille avec des dictionnaires "maps" de petites tailles.

- Dans un dictionnaire permettant l'insertion de plusieurs entrées ayant la même valeur de clé, l'opération insérer(k,e) s'exécute en temps constant, étant donné qu'on peut insérer une nouvelle entrée au début ou à la fin de la liste.

  - Dans ce cas, l'implémentation sous forme de liste peut être efficace sur des dictionnaires de grandes tailles, lorsque les opérations trouver(k) et enlever (k) n'ont pas à être utilisées trop souvent. Ex: registres historiques

Rappel: Qu'est-ce qu'un dictionnaire?

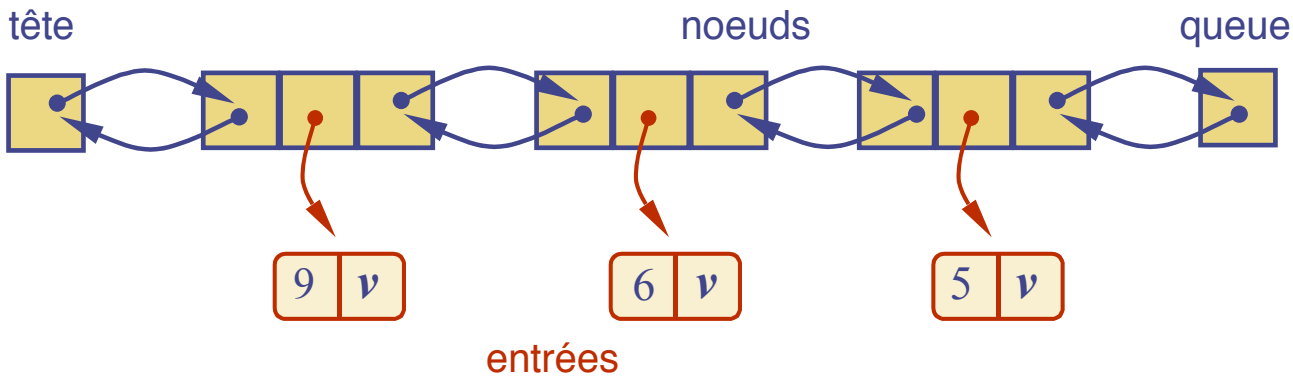
- Un dictionnaire est une structure de données gardant en mémoire des entrées de la forme (clé, élément)
- Le but d'un dictionnaire est d'être capable d'accéder rapidement à une entrée, en donnant la valeur de sa clé.
- Ici, contrairement aux listes avec priorités, on n'a pas besoin d'une relation d'ordre pour comparer les clés.
- Par contre, on doit pouvoir décider si deux clés sont égales.
  - On utilisera pour cela un testeur d'égalité (similaire au comparateur pour les files avec priorités)
- On permet l'insertion de plusieurs entrées ayant la même clé

TAD: Dictionnaires

- Opérations principales:
  - trouver(k): Si le dictionnaire a une entrée de clé k, retourne la valeur associée à cette entrée, sinon retoune NULL.
  - findAll(k): Retourne un itérateur de toutes les entrées de clé k, ou NULL si aucune entrée de clé k
  - insérer(k,e): insérer l'entrée (k,e) dans le dictionnaire
    put(k,e):
  - enlever(k): Si le dictionnaire a une entrée de clé k, l'enlever et
    remove(k): retourner sa valeur associée, sinon retourner NULL
- Opérations auxiliaires:
  - taille(): size():
  - estVide(): isEmpty():

Implémentation sous forme d'une liste doublement chaînée

- On peut implémenter un dictionnaire avec une liste non ordonnée, doublement chaînée
- On garde en mémoire dans la liste les entrées, dans un ordre arbitraire.



Algorithme enlever(*k*):
B =*L*.positions() {*B* est un itérateur des noeuds de *L*}
tant que *B*.àSuivant() **faire**
p = *B*.suivant()
si clé(p) = *k* **alors**
t = valeur(p)
L.enlève(*p*)
n = *n* – 1 {décrémenter le nombre d'entrées}
retourner *t* {retourner la valeur enlevée}
Retourner NULL {il n'y a pas d'entrée de clé k}

$O(n)$

Algorithme trouver(*k*):
B = *L*.positions() {*B* est un itérateur des noeuds de *L*}
tant que *B*.àSuivant() **faire**
p = *B.suivant*()
si clé(p) = *k* **alors**
retourner valeur(p)
retourner NULL {il n'y a pas d'entrée de clé k}

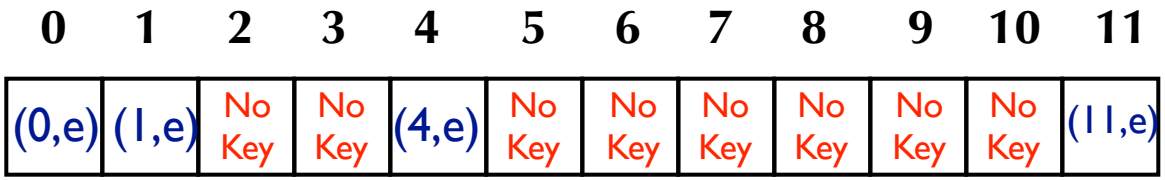
$O(n)$

Algorithme insérer(*k,v*):
L.insèreFin((*k,v*))
n = *n* + 1 {incrémenter le nombre d'entrées}

$O(1)$

Implémentation sous forme d'un tableau

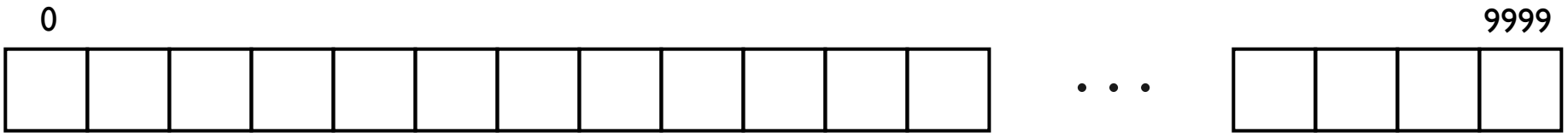
- Lorsque les clés de nos entrées sont des entiers, on peut facilement implémenter notre dictionnaire à l'aide d'un tableau de longueur N
- La cellule d'indice i du tableau est vue comme un contenant pour les entrées de clé i.
- Si aucune entrée du dictionnaire à une clé k, la cellule k contient un objet spécial No_Key
- Si on essaie d'insérer plus d'un élément de clé k, il y a collision



Problèmes de cette implémentation

- Les clés doivent être des entiers positifs entre 0 et N-1 (où N est la longueur de la table)

- La complexité en espace est $O(N)$, ce qui est souvent beaucoup plus grand que n, le nombre d'entrées du dictionnaire.



Fonctions et tables de hachage

- Une fonction de hachage h transforme les clés d'un certain type en des entiers entre 0 et N-1
- Exemple:

$$h(x) = x \mod N$$

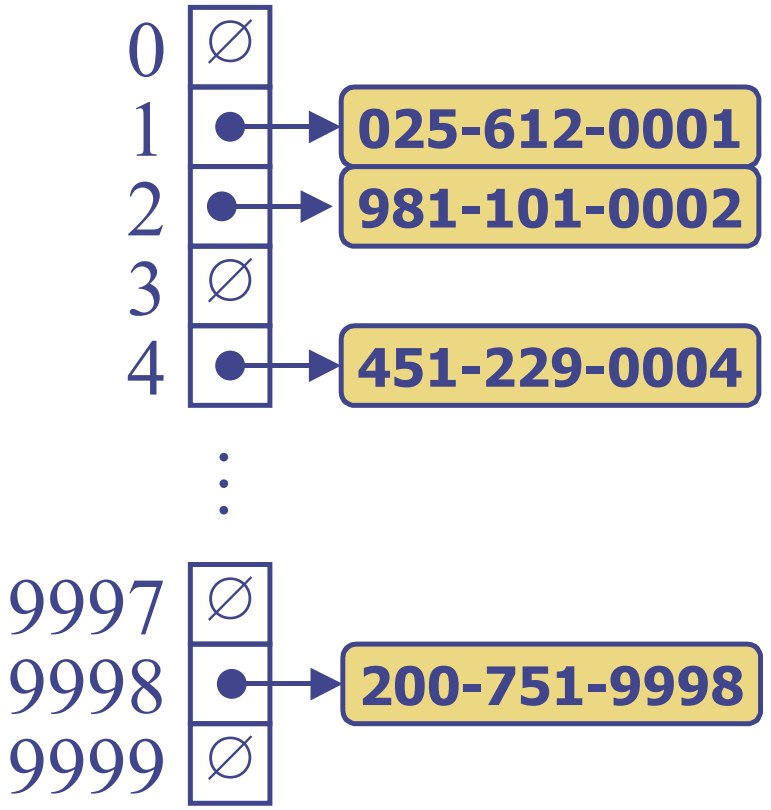
  est une fonction de hachage, lorsque les clés sont des entiers.
- Une fonction de hachage est dite parfaite si elle transforme des clés différentes en entiers différents
- Une table de hachage pour un type de clés donné, consiste en
  - Une fonction de hachage h
  - Un tableau de taille N
- Lorsqu'on implémente un dictionnaire à l'aide d'une table de hachage, le but est d'insérer l'entrée (k, v), dans la cellule d'indice $i = h(k)$

Exemple: extraction

- On veut implémenter un dictionnaire dont les entrées sont de la forme (numéro, NOM), où numéro est un code d'employés de 9 chiffres. (Ex: 020-111-6314

- Notre table de hachage utilise un tableau de taille N = 10 000 et la fonction de hachage

$$h(x) = \text{les 4 derniers chiffres de } x$$



Fonctions de hachage

- Une fonction de hachage est définie habituellement comme la composition de deux fonctions:

  1) Le code de hachage:

  $$h_1 : \text{clé} \rightarrow \text{entiers}$$

  2) La fonction de compression:

  $$h_2 : \text{entiers} \rightarrow [0, N-1]$$

- Le code de hachage est utilisé en premier, puis, la fonction de compression, i.e

  $$h(x) = h_2(h_1(x))$$

- Le but de la fonction de hachage est de "disperser" les clés d'une façon qui semble aléatoire.

Codes de hachage

- Représentation des bits par un entier:
  - On re-interprête la représentation en bits de la clé par un entier
  - Bon pour les clés de types: byte, short, int, char, float, boolean

    Float.floatToIntBits(x)
- Addition de composantes:
  - Pour les clés dont la représentation est de plus de 32 bits, on partitionne les bits de la clé en composantes de 32 bits et on fait la somme de ces composants
  - Bon pour les clés de types: long, double
  - Peut être utilisé pour les clés de type "string" mais pas le meilleur choix.

Codes de hachage (suite)

- Codes de hachage polynomials:
  - On partitionne les bits représentant la clé en une séquence de composantes de même longueur (8,16 ou 32 bits)

$$a_o a_1 \ldots a_{n-1}$$

  - On évalue le polynôme

$$p(z) = a_o + a_1 z + \ldots + a_{n-1} z^{n-1}$$

    pour une valeur donnée fixe de $z \neq 0$

  - Très bonne méthode pour les clés de type "String" (il a été démontré que le choix de $z = 33$, donne au plus 6 collisions sur un ensemble de 50 000 mots en anglais).

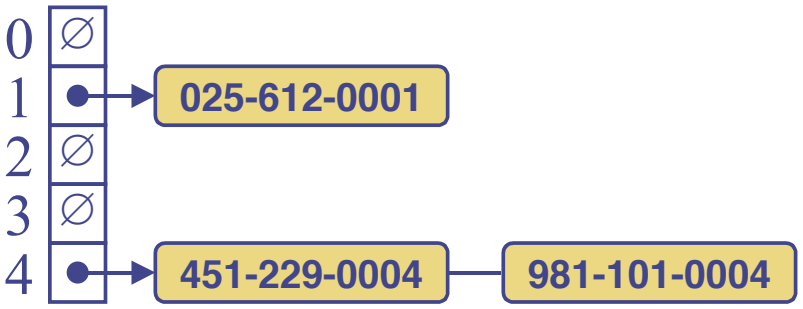
  - Le polynôme $p(z)$ peut être calculé en temps $O(n)$

Fonctions de compression

- Division:
  - $h_2(y) = y \mod N$
  - La taille N de la table de hachage est un nombre premier
  - Raison: théorie des nombres (voir tableau)
- Multiplier, additionner et diviser (MAD):
  - $h_2(y) = (ay + b) \mod N$
  - a et b sont des entiers non négatifs tels que
    $$a \mod N \neq 0$$
  - Sinon, tous les entiers seraient envoyés sur la valeur b

Résolution de collisions (chaînage)

- Les collisions surviennent lorsque différentes entrées sont envoyées dans la même cellule par la fonction de hachage.
- Cette méthode de résolution de collisions est appelée résolution par chaînage, ou hachage ouvert
- Chaque cellule i de la table pointe vers une liste chaînée contenant les éléments de clé k, telle que h(k)=i.



- Simple, mais requière de la mémoire externe à la table

Résolution de collisions: hachage fermé
1. sondage linéaire

- On parle de hachage fermé lorsque les entrées en collisions sont placées dans une cellule différente de la table (toujours au plus une entrée par cellule)

- La méthode de résolution de collisions par sondage linéaire, place l'entrée en collision dans la prochaine cellule disponible (table circulaire), i.e qu'on a les fonctions de hachage suivantes:

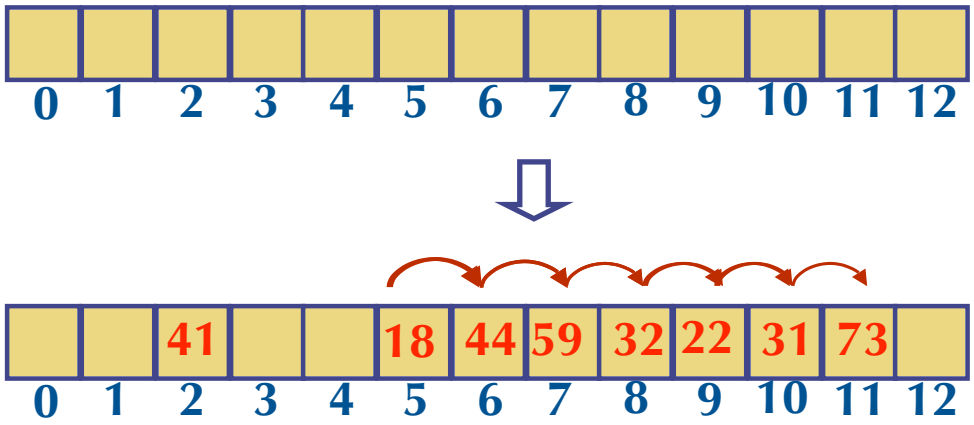
$$h_i(k) = h(k) + i \mod N$$

  qu'on essaie pour $i = 0, \ldots, N-1$, jusqu'à ce qu'on trouve une cellule libre.

- Chaque cellule inspectée lors de cette procédure est appelée une sonde

Résolution de collisions: hachage fermé
1. sondage linéaire (suite)

- Exemple:
  - $h_i(k) = k + i \mod 13$
  - On veut insérer les clés 18, 41, 22, 44, 59, 32, 31 et 73, dans cet ordre



Résolution de collisions: hachage fermé
1. sondage linéaire (suite)

- Algorithme trouver(k):
  - On commence la recherche à la cellule h(k)
  - On sonde les cellules suivantes, jusqu'à ce que l'une des conditions suivantes soit remplie:
    - Une entrée de clé k est trouvée
    - Une cellule vide est trouvée
    - N cellules ont été sondées sans succès

Résolution de collisions: hachage fermé
1. sondage linéaire (suite)

- Pour implémenter les opérations insérer(k,v) et enlever(k), on introduit un objet spécial "DISPONIBLE" qui va remplacer les entrées enlevées.

- Algorithme enlever(k):
  - On cherche une entrée de clé k
  - Si une telle entrée (k,v) est trouvée, on remplace cette entrée par "DISPONIBLE" et on retourne v
  - Sinon, on retourne NULL

- Algorithme insérer(k,v):
  - On envoie une exception si la table est pleine
  - On commencer à la cellule h(k)
  - On sonde les cellules, jusqu'à ce que une des conditions suivantes soit satisfaite:
    - Une cellule i est trouvée qui est soit vide ou garde en mémoire: "DISPONIBLE"
    - N cellules ont été sondées
  - On insère dans la cellule i

Résolution de collisions: hachage fermé

2 et 3) hachage quadratique et hachage aléatoire

- Hachage quadratique:
  - On trouve la prochaine cellule disponible selon les fonctions de hachage:

$$h_i(k) = k + i^2 \mod N$$

- Hachage aléatoire:
  - On trouve la prochaine cellule disponible selon les fonctions de hachage:

$$h_i(k) = k + d_i \mod N$$

où d est un permutation aléatoire de [1, N-1]

Résolution de collisions: hachage fermé

4) Hachage double

- La méthode de résolution de collisions par hachage double, utilise une fonction de hachage secondaire $d(k)$ pour résoudre les collisions, en plaçant l'entrée dans la première cellule disponible dans la série

$$(h(k) + jd(k)) \mod N$$

pour $j = 0, 1, \ldots, N-1$

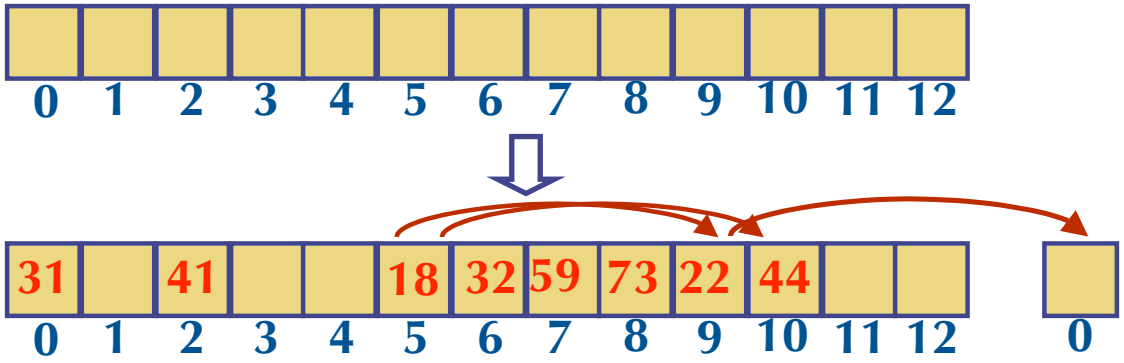
- La fonction de hachage secondaire $d(k)$ ne peut jamais être nulle.
- La taille de la table doit être un nombre premier
- Un choix commun pour d(k) est

$$d(k) = q - k \mod q$$

où q est un nombre premier < N

Exemple de hachage double:

- Considérons une table de hachage gardant en mémoire des entiers et utilisant la méthode suivante de double hachage pour gérer les collisions:
  - $N = 13$
  - $h(k) = k \mod 13$
  - $d(k) = 7 - k \mod 7$
- Insérer les clés 18, 41, 22, 44, 59, 32, 31 et 73 dans cet ordre:



Performance du hachage:

- Dans les pires des cas les opérations d'insertion, de recherche et de suppression se font en O(n)
- Le pire des cas arrive lorsqu'il y a collision à chaque insertion d'une entrée
- Le facteur de chargement de la table $\alpha = n/N$ affecte la performance de la table de hachage
- Si on assume que la fonction de hachage agit en dispersant les clés de façon aléatoire, on peut montrer que le nombre de sondes a inspectées pour une insertion avec une méthode de hachage ouvert est de

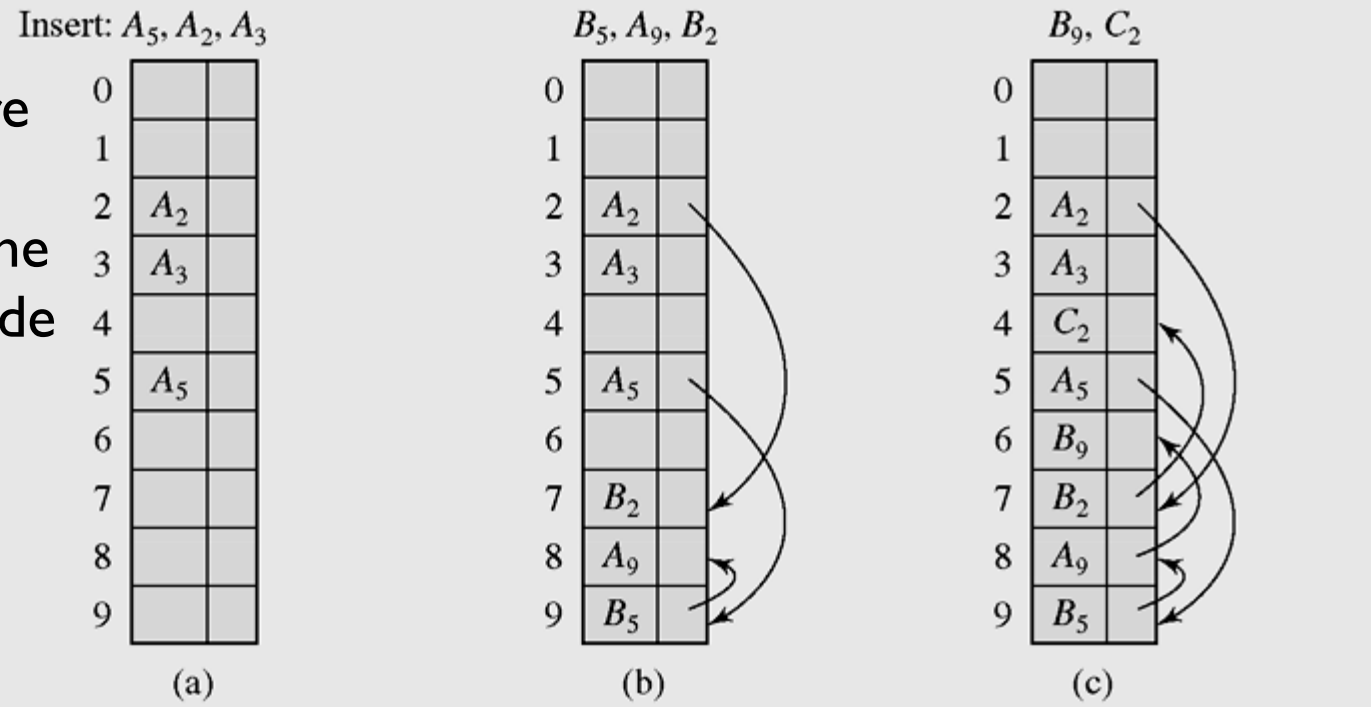
$$1/(1-\alpha)$$

Chaînage fusionné

- Combine la méthode de résolution de collisions par chaînage avec la méthode du sondage linéaire.

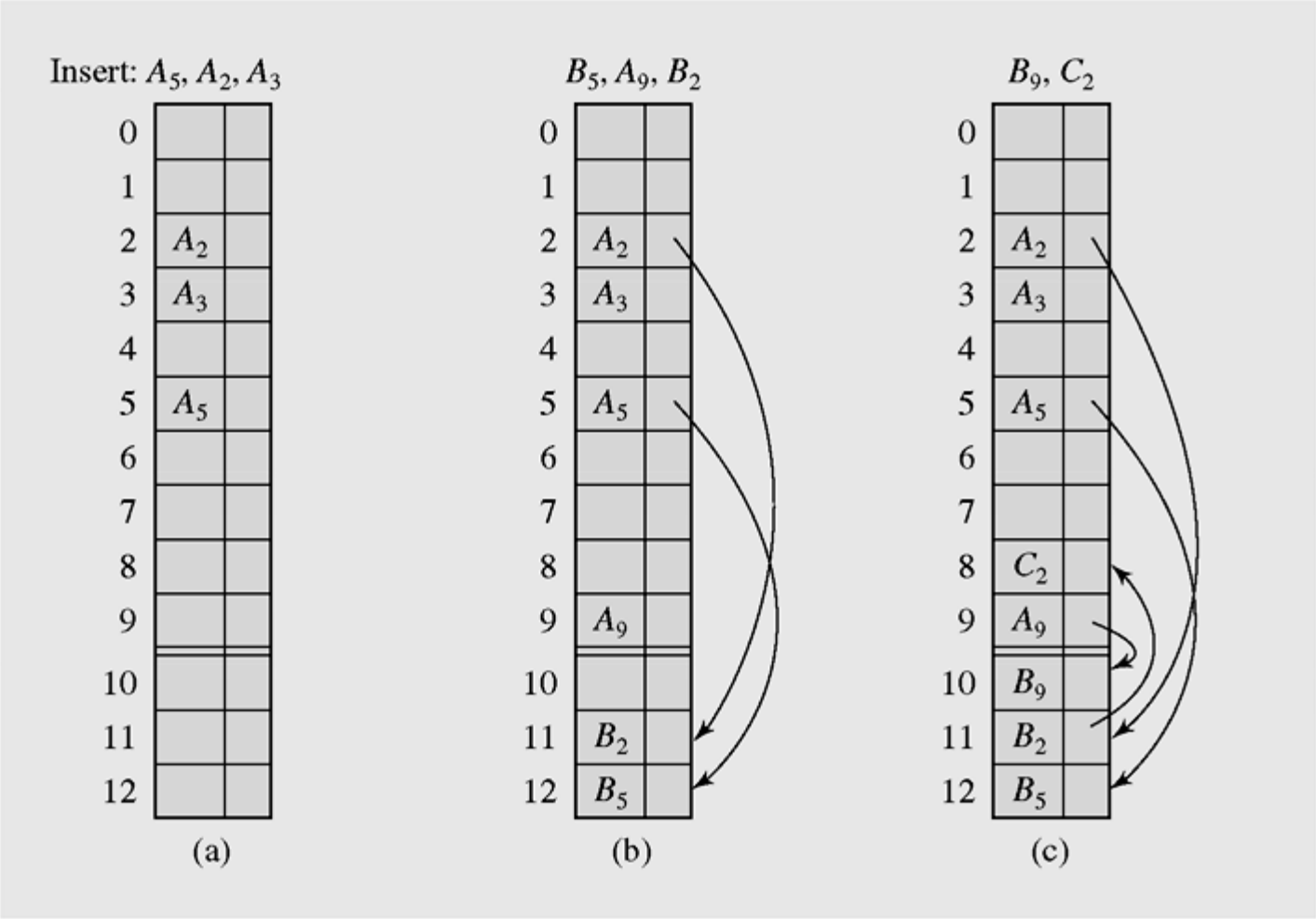
FIGURE 10.6 Coalesced hashing puts a colliding key in the last available position of the table.

- Si une clé entre en collision avec une autre clé dans la cellule i, on trouve la prochaine cellule vide j et on garde en position i, un pointeur vers j



Chaînage fusionné (suite)

FIGURE 10.7 Coalesced hashing that uses a cellar.



Définitions - Hachage parfait

- Une fonction de hachage est dite parfaite si elle transforme des clés différentes en des entiers différents
- Une fonction de hachage parfaite est dite minimale si elle envoie n éléments dans exactement n cellules
- On peut construire une telle fonction de hachage lorsque l'information à emmagasiner est fixe:
  - Mots réservés pour les assembleurs ou compilateurs
  - Fichiers sur disque non effaçable
  - Dictionnaire
  - etc....

Méthode de Cichelli*

- Méthode utilisée lorsqu'on a un ensemble, pas trop grand, de N mots réservés à garder en mémoire

$$h(mot) = (longueur(mot) + g(1^{ière} lettre(mot)) + g(dernière lettre(mot))) \ mod \ N$$

- L'algorithme de Cichelli construit la fonction g de sorte que la fonction h résultante retourne une valeur unique entre 0 et N-1 pour chaque mot de l'ensemble prédéfini au départ

- La fonction g peut assigner deux fois la même valeur à des lettres différentes

* Cichelli, R.J., *Minimal perfect hash functions made simple*, Communication of the ACM, **23**(1), 17-19, 1980

Exemple de construction de g et h

- L'algorithme de Cichelli comprend trois étapes:

1) Calculer, étant donné la liste de mots réservés, les fréquences d'apparition de chaque lettre en première ou dernière position des mots

8 4 8 12 7 4 8

Exemple: {Calliope, Clio, Erato, Euterpe, Melponeme, Polyhymnia, Thepsichore, Thalia, Urania}

5 4

Fréquences:

C	E	O	M	P	A	T	U
2	6	2	1	1	3	2	1

2) Ordonner les mots de sorte que celui dont la plus grande somme des fréquences de sa première et dernière lettre soit en premier.

Euterpe, Calliope, Erato, Thepsichore, Melponeme, Thalia, Clio, Polyhymnia, Urania

Exemple de construction de g et h (suite)

3) Construire g et h

- On choisit un maximun
  - Habituellement $\lfloor \frac{N}{2} \rfloor$
  - Ici, $\lfloor \frac{9}{2} \rfloor = 4$
- Pour le premier mot de la liste, on essaie g(première lettre) = i et g(dernière lettre)=j pour i, j dans {0, ..., maximum}, jusqu'à ce qu'on trouve des valeurs pour lesquelles h(mot) n'est pas déjà pris
- On sauvegarde ces valeurs de g et de h et on continue avec le prochain mot
- Si pour un mot, aucune valeur de g ne fonctionne, on "backtrack"