# Cours 6 : L’adressage dispersé

Principe
Variations
Exemples

# Recherche d'un élément dans une collection

- Pour l'instant
  - Meilleure solution = recherche dichotomique dans une collection triée
  - Recherche en $\Theta(\log(n))$
  - Ajout en $\Theta(\log(n))$

- Peut-on faire mieux ?

# Principe de l'adressage dispersé

- On associé à chaque élément une clé
  - En général une valeur numérique entière
- On va ranger l'élément en fonction de sa clé
  - Clé = indice de la case du tableau dans laquelle on doit ranger l'élément
- L'indice de l'élément dans le tableau est *fonction de l'élément*

# Calcul de la clé (ou hash code)

- Via une fonction de hachage
- Exemples :
  - Rang de la première lettre d'un mot ou d'une chaîne
    - **a** $\rightarrow$ 0, **b** $\rightarrow$ 1, etc...
  - Somme des codes ASCII des lettres d'une chaîne
    - Le tout modulo *M* où *M* est la taille du tableau dans lequel on stocke les données

# Une bonne fonction de hachage ?

- Permet d'atteindre toutes les cases du tableau (évidemment)
- Réalise une bonne dispersion
  - Distribution uniforme sur les clés
  - Exemples précédent :
    - 2ème fonction meilleure que la première
    - Beaucoup de chaînes commencent par la même lettre
    - De façon inverse, certaines lettres ne seront guère ou jamais utilisées

# Objectif de l'adressage dispersé

- Opérations de :
  - Insertion
  - Recherche
  - Suppression

- En temps constant : complexité $\Theta(1)$

# Méthode

- Pour insertion, recherche ou suppression
  - On calcule la clé de l'élément à traiter
  - On va a l'indice correspondant à la clé
  - On réalise le test ou l'action

- Si chaque clé se réfère à un *élément unique*, on a effectivement un traitement en temps constant

# Collisions

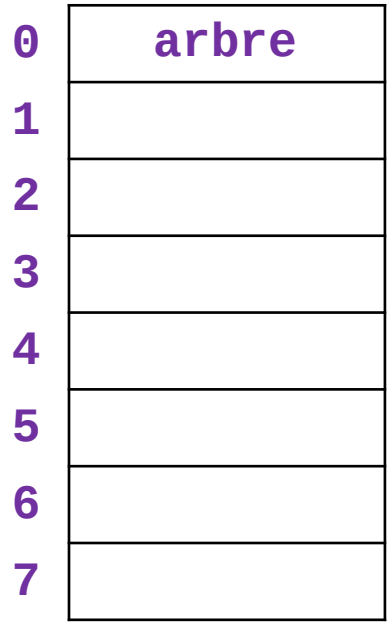
- La situation précédente n'est possible que si la taille de stockage est au moins égale au nombre d'éléments à stocker
- Si ce n'est pas le cas, la fonction de hachage ne peut pas être injective
  - Deux éléments au moins auront la même clé
  - On appelle ce cas de figure une *collision*

# Exemple

- On a des mots à insérer dans un tableau de taille $M = 8$
- La fonction de hachage $h$ est donnée par le rang de la première lettre du mot
  - $h($**`a`**$) = 0$, $h($**`b`**$) = 1$, etc... (modulo la taille du tableau)
  - Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**

# Exemple (2)

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**




- On insère **`arbre`**
- *h*(**`arbre`**) = 0

# Exemple (3)

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**

| | |
|---|---|
| 0 | `arbre` |
| 1 | |
| 2 | `chat` |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

- On insère **`chat`**
- *h*(**`chat`**) = 2

# Exemple (4)

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**


- On insère **`livre`**
- *h*(**`livre`**) = 11 mod 8 = 3

# Exemple (5)

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**

| | |
|---|---|
| 0 | `arbre` |
| 1 | `rouge` |
| 2 | `chat` |
| 3 | `livre` |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

- On insère **`rouge`**
- *h*(**`rouge`**) = 17 mod 8 = 1

# Exemple (6)

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**

| | |
|---|---|
| 0 | `arbre` |
| 1 | `rouge` |
| 2 | `chat` |
| 3 | `livre` |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

- On insère **`café`**
- *h*(**`café`**) = 2

# Gestion des collisions

Tableau des collisions
Chaînage externe
Adressage ouvert

# Le tableau des collisions (hachage coalescent)

- Solution vue en 1ère année
- On crée un tableau annexe
- Lorsque l'on a collision, on ajoute l'élément en fin de tableau des collisions (s'il reste de la place, évidemment)

# Exemple précédent

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | arbre | | | | | | |
| 1 | rouge | | | | | | |
| 2 | chat | café | safran | | | | |
| 3 | livre | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |

- On insère **arbre**, **chat**, **livre**, **rouge**, **café**
- Et **safran**

# Recherche avec un tableau des collisions

- On calcule la clé de l'élément
- Si la case d'indice égal à la clé contient l'élément → **`réussite`**
- Si la case est vide → **`échec`**
- Sinon
  - On cherche (séquentiellement) dans le tableau des collisions

# Evaluation

- Avantages :
  - On garde des structures simples
- Inconvénients :
  - La recherche n’est plus immédiate
  - Il faut parfois parcourir un autre tableau

# Variantes

- Lorsque la zone de collision est pleine, on peut imaginer d'utiliser des cases vides du tableau principal
  - Il faut alors savoir quelles sont ces cases « spéciales »
  - Risque de collisions secondaires (un élément voit sa place prise par un élément issu des collisions)

# Chaînage externe

- Principe : on « regroupe » les éléments ayant des clés identiques au sein d'une même structure
  - Nom anglais : *buckets*
  - Dans une écrasante majorité des cas, on utilise une liste chaînée
- La table de hachage est alors un *tableau de listes chaînées*

# Opérations

- Soit une fonction de hachage *h*
- Ajout de *e* :
  - On ajoute dans la liste tab[*h*(*e*)]
- Recherche de *e*
  - On recherche dans la liste tab[*h*(*e*)]
- Suppression de *e*
  - On le supprime de la liste tab[h(*e*)]

# Evaluation

- Avantages :
  - Un seul tableau (de listes)

- Inconvénients :
  - Utilisation de listes chaînées
  - La recherche n’est plus immédiate

# Complexité de la recherche

- *N* = nombre d'éléments à stocker
- *M* = taille du tableau
  - Ou plutôt taille de l'ensemble image de la fonction de hachage
  - Nombre de clés différentes
- *k*=*N*/*M* : nombre moyen d'éléments par clé
  - Pour une fonction de hachage parfaite, toutes les clés correspondent à *k* éléments

# Complexité de la recherche

- Complexité de la recherche d'un élément dans le tableau :
  - On calcule la clé
  - On parcours une liste de *k* éléments en moyenne

$$\Theta(1+k)$$

# Hachage à adressage ouvert

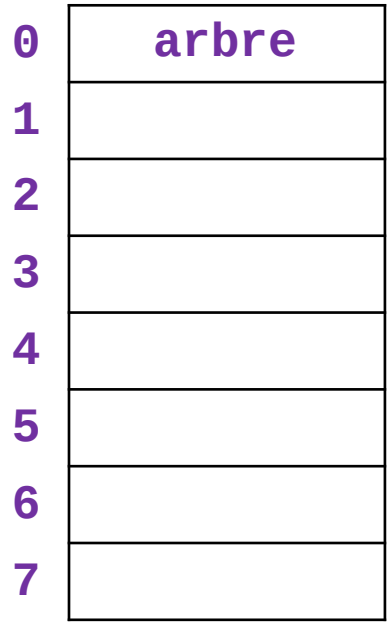
- Principe : on ne gère pas les collisions
- Idée (pour recherche et ajout) :
  - Si la case est occupée, on en tente une autre
  - Si celle-ci est occupée, on en tente encore une autre
  - etc...

# Hachage linéaire

- Si la case est occupée, on prend la suivante
  - Si celle-ci est occupée on prend la suivante et ainsi de suite
  - Si on arrive à la fin du tableau on revient au début
  - Pour l'ajout, si le tableau n'est pas plein on trouve forcément une place
  - Pour la recherche, si on rencontre une case vide ou qu'on revient sur la case de départ on sait qu'on a échoué

# Exemple précédent

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**




- On insère **`arbre`**
- *h*(**`arbre`**) = 0

# Exemple (2)

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**

| | |
|---|---|
| 0 | `arbre` |
| 1 | |
| 2 | `chat` |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

- On insère **`chat`**
- *h*(**`chat`**) = 2

# Exemple (3)

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**


- On insère **`livre`**
- *h*(**`livre`**) = 11 mod 8 = 3

# Exemple (4)

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**

| | |
|---|---|
| 0 | `arbre` |
| 1 | `rouge` |
| 2 | `chat` |
| 3 | `livre` |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

- On insère **`rouge`**
- *h*(**`rouge`**) = 17 mod 8 = 1

# Exemple (5)

- Les mots sont **`arbre`**, **`chat`**, **`livre`**, **`rouge`** et **`café`**

| | |
|---|---|
| 0 | arbre |
| 1 | rouge |
| 2 | chat |
| 3 | livre |
| 4 | café |
| 5 | |
| 6 | |
| 7 | |

- On insère **`café`**
- *h*(**`café`**) = 2 → prise
- *h*(**`café`**)+1 = 3 → prise
- *h*(**`café`**)+2 = 4 → libre

# Evaluation

- Avantages :
  - On garde une structure unique simple
- Inconvénients :
  - Risque de collisions secondaires
  - Il faut « marquer » les cases inoccupées

# Complexité de la recherche

- $\alpha$ = taux de remplissage du tableau

- Nombre d'accès moyen :
  - Pour une recherche infructueuse : $\frac{1}{1-\alpha}$
  - Pour une recherche fructueuse : $\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)+\frac{1}{\alpha}$

# Complexité de la recherche

- Pour une table a moitié pleine (quelle que soit la taille)
  - En moyenne 2 accès si absent
  - En moyenne 3,387 accès si présent

- On a bien du $\Theta(1)$

# Autres modes de dispersion

- On peut vouloir une évolution des indices qui « disperse » davantage les cases :
  - Hachage quadratique :
    - On teste les cases $(h(e)+i^2) \bmod M$
      - On cherche si $h(e)$ est libre, sinon $h(e)+1$, sinon $h(e)+4$, sinon $h(e)+9$, …
  - Utilisation d'une fonction de hachage secondaire $k$ :
    - On teste les cases $(h(e)+i*k(e)) \bmod M$

# Complexité de la recherche

- Varie selon les cas
  - Mais on reste globalement en $\Theta(1)$

# A retenir

- L’indice auquel on range un objet est calculé à partir de sa valeur
- Il n’y a pas de fonction de hachage universelle
  - Cas par cas
  - Bonne fonction = bonne dispersion
- Lorsqu’il y a collision (2 éléments ont même clé)
  - On utilise un tableau annexe
  - On fait des listes
  - On change de case