

Piqûre de rappel (qui ne devrait pas être là)

- Attention: == teste l'égalité des contenus des boîtes.
- Pour tester l'égalité des contenus pointés, utiliser la méthode equals.
- Conséquence: tester l'égalité de chaînes de caractères:

```
String S="Bonjour";  
String T="Bonjour";  
System.out.println((S==T) + ", " + S.equals(T));
```

Seul le deuxième résultat est garanti d'être **true**.

Piûre de rappel: constructeurs

```
class Personne
{ String nom;
  int promo;
  Personne(String s, int p)
  { nom = s;
    promo = p;  }
  Personne(String s)
  { nom = s;
    promo = 2007;
  }
}
```

```
Personne p = new Personne("Robert");
Personne q = new Personne("Augustin",2006);
```

- On créer un objet avec un **new**
- Un constructeur peut avoir un nombre quelconque d'arguments, et pourrait éventuellement faire autre chose qu'initialiser ses champs.

Conversions

```
class Operation {  
  
    char code; // '+' ou '-' ou '*' ou '/' ou 'e' (empilement)  
    double a, b;  
  
    ...  
    public String toString() {  
        switch(code) {  
            case '+':  
            case '-':  
            case '*':  
            case '/':  
                return code + "";  
            default:  
                return a + "";  
        }  
    }  
}
```

- Le `a + ""` force la conversion du réel `a` en une chaîne de caractères.

Une remarque sur le codage des listes vides

- Dans le codage des listes des cours précédents

```
class Liste {  
    int contenu;  
    Liste suivant;  
    Liste (int x, Liste a) {  
        contenu = x;  
        suivant = a;  
    }  
}
```

une liste vide est codée par **null**.

- On crée donc une liste vide par

```
Liste L= null;
```

- Avec la librairie standard et `LinkedList<X>`, on crée une liste vide par

```
LinkedList<Integer> L=new LinkedList<Integer>();
```

Cours 4: Associations. Hachage.

Olivier Bournez

bournez@lix.polytechnique.fr
LIX, Ecole Polytechnique

INF421-a

Bases de la programmation et de l'algorithmique

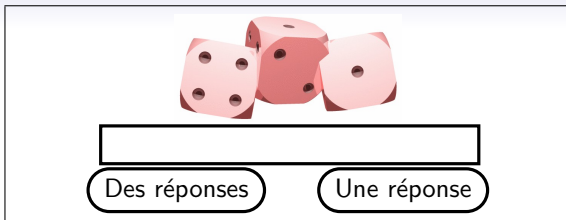
Aujourd'hui

La structure dictionnaire

Le hachage

Associations & Bibliothèques standards JAVA

Un moteur de recherche



La problématique d'un moteur de recherche:

- prendre une clé k dans un univers U .
 - ▶ Exemple: une chaîne de caractères parmi les chaînes possibles.
- retourner une donnée $D(k)$ associée à cette clé.
 - ▶ Exemple:
 - A droite: l'adresse d'une page [www](#).
 - A gauche: une liste d'adresses de pages [www](#).
 - ▶ Autre exemple: un numéro de téléphone.

Le type abstrait `Dictionnaire`

On cherche une structure de donnée, `Dictionnaire`, qui permette

- D'insérer un couple (clé, donnée): `Insérer(k,D)`.
- Rechercher la donnée associée à une clé: `Rechercher(k)`.
- Supprimer la donnée associée à une clé: `Supprimer(k)`.

On suppose

- qu'une clé k identifie une unique donnée

- que k vit dans un univers U .
- que D vit dans un univers V .

Le type abstrait Dictionnaire

On cherche une structure de donnée, Dictionnaire, qui permette

- D'insérer un couple (clé, donnée): `Insérer(k,D)`.
- Rechercher la donnée associée à une clé: `Rechercher(k)`.
- Supprimer la donnée associée à une clé: `Supprimer(k)`.

On suppose

- qu'une clé k identifie une unique donnée
 - ▶ Exemple: numéro de sécurité social \rightarrow personne
 - ▶ Exemple: numéro ISBN \rightarrow livre
- que k vit dans un univers U .
- que D vit dans un univers V .

Le type abstrait Dictionnaire

On cherche une structure de donnée, Dictionnaire, qui permette

- D'insérer un couple (clé, donnée): $\text{Insérer}(k,D)$.
- Rechercher la donnée associée à une clé: $\text{Rechercher}(k)$.
- Supprimer la donnée associée à une clé: $\text{Supprimer}(k)$.

On suppose

- qu'une clé k identifie une unique donnée
 - ▶ Exemple: numéro de sécurité social \rightarrow personne
 - ▶ Exemple: numéro ISBN \rightarrow livre
- que k vit dans un univers U .
- que D vit dans un univers V .
 - ▶ Exemple: $U = \mathbb{N}$, $V =$ les livres.
 - ▶ Exemple: $U = \mathbb{N}$, $V =$ les chaînes de caractères.
 - ▶ Exemple: $U =$ les chaînes de caractères, $V =$ un numéro de téléphone.

Solutions

- Solutions:

- ▶ Tableau, ou liste.
- ▶ Tableau trié, lorsque U peut être ordonné.
- ▶ Table de hachage.

Solutions

- Solutions:

- ▶ Tableau, ou liste.
- ▶ Tableau trié, lorsque U peut être ordonné.
- ▶ Table de hachage.

On note n le nombre d'éléments dans le dictionnaire.

On suppose que chacune des permutations est équiprobable pour les calculs en moyenne.

Solutions

■ Solutions:

- ▶ Tableau, ou liste.
 - Rechercher(k) en temps $\Theta(n)$ au pire et en moyenne.
- ▶ Tableau trié, lorsque U peut être ordonné.
- ▶ Table de hachage.

On note n le nombre d'éléments dans le dictionnaire.

On suppose que chacune des permutations est équiprobable pour les calculs en moyenne.

Solutions

■ Solutions:

- ▶ Tableau, ou liste.
 - Rechercher(k) en temps $\Theta(n)$ au pire et en moyenne.
- ▶ Tableau trié, lorsque U peut être ordonné.
 - Rechercher(k) en temps $\Theta(\log n)$ au pire et en moyenne.
- ▶ Table de hachage.

On note n le nombre d'éléments dans le dictionnaire.

On suppose que chacune des permutations est équiprobable pour les calculs en moyenne.

Solutions

■ Solutions:

- ▶ Tableau, ou liste.
 - Rechercher(k) en temps $\Theta(n)$ au pire et en moyenne.
- ▶ Tableau trié, lorsque U peut être ordonné.
 - Rechercher(k) en temps $\Theta(\log n)$ au pire et en moyenne.
- ▶ Table de hachage.
 - Rechercher(k) en temps $O(1)$ en moyenne ($O(n)$ au pire).

On note n le nombre d'éléments dans le dictionnaire.

On suppose que chacune des permutations est équiprobable pour les calculs en moyenne.

Solutions

- Dans le contexte d'aujourd'hui:
 - ▶ on veut que `Rechercher(k)` réponde très rapidement, en **moyenne**.
- Solutions:
 - ▶ Tableau, ou liste.
 - `Rechercher(k)` en temps $\Theta(n)$ au pire et en moyenne.
 - ▶ Tableau trié, lorsque U peut être ordonné.
 - `Rechercher(k)` en temps $\Theta(\log n)$ au pire et en moyenne.
 - ▶ Table de hachage.
 - `Rechercher(k)` en temps $O(1)$ en moyenne ($O(n)$ au pire).

On note n le nombre d'éléments dans le dictionnaire.

On suppose que chacune des permutations est équiprobable pour les calculs en moyenne.

Solutions

- Dans le contexte d'aujourd'hui:
 - ▶ on veut que Rechercher(k) réponde très rapidement, en **moyenne**.
- Solutions:
 - ▶ Tableau, ou liste.
 - Rechercher(k) en temps $\Theta(n)$ au pire et en moyenne.
 - ▶ Tableau trié, lorsque U peut être ordonné.
 - Rechercher(k) en temps $\Theta(\log n)$ au pire et en moyenne.
 - ▶ Table de hachage.
 - Rechercher(k) en temps $O(1)$ en moyenne ($O(n)$ au pire).

Pour ce contexte: Tableau < Tableau triés < Hachage

On note n le nombre d'éléments dans le dictionnaire.

On suppose que chacune des permutations est équiprobable pour les calculs en moyenne.

Solution "Tableau": du JAVA

- On utilise un tableau contenant les paires (k, D) .

```
class Asso {  
    String k;  
    Personne D;  
    Asso(String kk, Personne DD)  
        {k=kk; D=DD;}}
```

```
class Tableau {  
    int n = 0; //nb clés  
    private Asso[] tab = new Asso[2];  
  
    int index(String k) {  
        for (int i = 0; i < n; i++)  
            if (tab[i].k.equals(k)) return i;  
        return -1; //clé inconnue  
    }  
  
    void doubler() {  
        Asso[] tab2 =  
            new Asso[2*tab.length];
```

```
        for (int i=0; i<tab.length; i++)  
            tab2[i] = tab[i];  
        tab = tab2;}}
```

```
    Personne Rechercher(String k) {  
        int i= index(k);  
        if (i  $\geq$  0) return tab[i].D;  
        return null; } //clé inconnue
```

```
    void Insérer(String k, Personne D)  
    {if (n == tab.length) doubler();  
     tab[n++] = new Asso(k, D);}
```

```
    void Supprimer(String k) {  
        int i=index(k);  
        if (i  $\geq$  0) {  
            for (int j=i+1; j < n; j++)  
                tab[j-1] = tab[j];  
            n--; }  
    }
```

Solution “Tableau”: complexité

■ Au pire cas:

- ▶ Insérer(k, D): $O(n)$.
- ▶ Rechercher(k): $O(n)$.
- ▶ Supprimer(k): $O(n)$.

■ En moyenne (exercice).

- ▶ Rechercher(k): $\Theta(n)$.

■ Remarque:

- ▶ Si l'on rentre n couples (k, D) au clavier, puis l'on fait **une** recherche:
 - Cela fait $O(n) + O(n) = O(n)$ opérations.
 - On ne peut pas faire mieux que $O(n)$.
 - Preuve: il faut au moins lire l'entrée.

Solution "Tableau trié": du JAVA

- On utilise un tableau contenant les paires (k, D) , triées par clé k .

```
...
int index(String k, int min, int max){
    if(min ≥ max) // vide
        return -1;
    int mid = (min + max) / 2;
    if (tab[mid].k.compareTo(k)==0) return mid;
    else if (tab[mid].k.compareTo(k) <0) return index(k, min, mid);
    else return index(k,mid + 1, max);}

void Insérer(String k, Personne D) {
    if (n == tab.length) doubler();
    // Trouver la bonne place
    for (i=0;i<n && tab[i].k.compareTo(k)<0;i++) {}
    // Décaler à droite
    for (int j=n; j >i; j--) tab[j] = tab[j-1];
    n++;
    tab[i]= new Assoc(k,D);}
...

```

Solution “Tableau trié”: complexité

- Au pire cas:
 - ▶ Insérer(k, D): $O(n)$.
 - ▶ Rechercher(k): $O(\log n)$.
 - ▶ Supprimer(k): $O(n)$.
- En moyenne (exercice).
 - ▶ Rechercher(k): $\Theta(\log n)$.
- Remarque:
 - ▶ Avec un algorithme qui fonctionne par comparaison entre la clé et les éléments, sans autre information sur les clés, il n'est pas possible de faire mieux que $O(\log n)$ au pire cas.
 - ▶ Trier peut être vu comme un prétraitement (de coût $O(n \log n)$) qui transforme la complexité d'une recherche de $O(n)$ en $O(\log n)$.

Solution “Tableau trié”: complexité

- Au pire cas:
 - ▶ Insérer(k, D): $O(n)$.
 - ▶ Rechercher(k): $O(\log n)$.
 - ▶ Supprimer(k): $O(n)$.
- En moyenne (exercice).
 - ▶ Rechercher(k): $\Theta(\log n)$.
- Remarque:
 - ▶ Avec un algorithme qui fonctionne par comparaison entre la clé et les éléments, sans autre information sur les clés, il n'est pas possible de faire mieux que $O(\log n)$ au pire cas.
 - Principe de la preuve (laissée en exercice): considérer l'arbre de décision associé à l'algorithme comme dans le cours précédent.
 - ▶ Trier peut être vu comme un prétraitement (de coût $O(n \log n)$) qui transforme la complexité d'une recherche de $O(n)$ en $O(\log n)$.

Recherche rapide?

- Ces complexités dépendent de n , la taille de la base.
 - ▶ pour un moteur de pages www, $n = 20$ à 25 Milliards de pages¹.
 - ▶ pour un annuaire, en France $n = 40$ millions pour lignes fixes, $n = 55,7$ millions pour lignes mobiles².
- Objectif de ce qui suit: une recherche en $O(1)$ en moyenne.

¹Estimation mai 2008. Il n'y a plus de communication sur ce chiffre de la part des principaux moteurs depuis 2005.

²Chiffres 1ier trimestre 2008.

Aujourd'hui

La structure dictionnaire

Le hachage

Associations & Bibliothèques standards JAVA

Le cas simple: un petit univers de clés, et $U \subset \mathbb{N}$

- Lorsque les clés appartiennent à $\{0, 1, 2, \dots, m-1\}$, avec $n \leq m$, on sait aller très vite, en temps $O(1)$:
 - ▶ Il suffit d'utiliser un tableau *tab* de taille *m*.
 - ▶ L'élément de clé *k* correspond à *tab*[*k*].
- Exemple: si *V* correspond aux chaînes de caractères.

```
String tab[] = new String[m];  
  
void Insérer(int k, String D) {tab[k] = D;}  
void Supprimer(int k) {tab[k] = null;}  
String Rechercher(int k) {return tab[k];}
```

- Insérer(*k*,*D*), Rechercher(*k*), Supprimer(*k*) se font en temps $O(1)$.
 - ▶ (raison physique: un tableau est stocké de façon contiguë en mémoire, et une mémoire sait accéder au contenu d'une adresse en temps constant).

Exemple

- U = les numéros de départements.
- V = noms de départements.

On peut prendre un tableau de taille 975 (puisque'ils vont jusqu'à 974, la réunion).

Après insertion de la table des départements, en mémoire:

- `tab[1]` vaut "Ain"
- `tab[91]` vaut "Essonne"
- `tab[974]` vaut "La réunion"

- `tab[374]` n'est pas défini.

Le cas général: principe du hachage

- Si U n'est pas constitué d'entiers, ou est de grande taille, on se ramène à ce cas.
- On va utiliser une fonction $h : U \rightarrow \{0, 1, \dots, m - 1\}$, pour h et m bien choisis.
 - ▶ On utilise un tableau tab de taille m .
 - ▶ L'élément de clé k correspondra à $tab[h(k)]$.
- Les cases du tableau tab sont appelées des *alvéoles*.
- Bien choisis:
 - On veut que
 - ▶ $h(k)$ se calcule vite (temps $O(1)$).

Le cas général: principe du hachage

- Si U n'est pas constitué d'entiers, ou est de grande taille, on se ramène à ce cas.
- On va utiliser une fonction $h : U \rightarrow \{0, 1, \dots, m - 1\}$, pour h et m bien choisis.
 - ▶ On utilise un tableau tab de taille m .
 - ▶ L'élément de clé k correspondra à $tab[h(k)]$.
- Les cases du tableau tab sont appelées des *alvéoles*.
- Bien choisis:

On veut que

- ▶ $h(k)$ se calcule vite (temps $O(1)$).
- ▶ $m \ll Taille(Univers\ U)$: on voudrait un tableau pas trop grand.

Le cas général: principe du hachage

- Si U n'est pas constitué d'entiers, ou est de grande taille, on se ramène à ce cas.
- On va utiliser une fonction $h : U \rightarrow \{0, 1, \dots, m - 1\}$, pour h et m bien choisis.
 - ▶ On utilise un tableau tab de taille m .
 - ▶ L'élément de clé k correspondra à $tab[h(k)]$.
- Les cases du tableau tab sont appelées des *alvéoles*.
- Bien choisis:

On veut que

- ▶ $h(k)$ se calcule vite (temps $O(1)$).
- ▶ $m \ll Taille(Univers\ U)$: on voudrait un tableau pas trop grand.
- ▶ Si $m \geq n$, on aimerait h injective: on voudrait éviter les collisions, c'est-à-dire les clés $k, k', k \neq k'$ avec $h(k) = h(k')$.

Le cas général: principe du hachage

- Si U n'est pas constitué d'entiers, ou est de grande taille, on se ramène à ce cas.
- On va utiliser une fonction $h : U \rightarrow \{0, 1, \dots, m - 1\}$, pour h et m bien choisis.
 - ▶ On utilise un tableau tab de taille m .
 - ▶ L'élément de clé k correspondra à $tab[h(k)]$.
- Les cases du tableau tab sont appelées des *alvéoles*.
- Bien choisis:

On veut que

- ▶ $h(k)$ se calcule vite (temps $O(1)$).
- ▶ $m \ll Taille(Univers\ U)$: on voudrait un tableau pas trop grand.
- ▶ Si $m \geq n$, on aimerait h injective: on voudrait éviter les collisions, c'est-à-dire les clés $k, k', k \neq k'$ avec $h(k) = h(k')$.
- ▶ Si $m < n$, il faut nécessairement gérer les collisions.

Exemple

- U = noms de départements.
- V = les numéros de départements.

On peut prendre $m = 50$ et la (très mauvaise) fonction de hachage h qui renvoie la somme des codes ASCII modulo m .

$$h(\text{"Ain"}) = 65 + 105 + 110 \pmod{50} = 280 \pmod{50} = 30$$

$$h(\text{"Essonne"}) = 31$$

$$h(\text{"La réunion"}) = 5.$$

Après insertion de la table des départements, en mémoire:

- $\text{tab}[30]$ contient "Ain"
- $\text{tab}[31]$ contient "Essonne"
- $\text{tab}[5]$ contient "La réunion"

Exemple

- U = noms de départements.
- V = les numéros de départements.

On peut prendre $m = 50$ et la (très mauvaise) fonction de hachage h qui renvoie la somme des codes ASCII modulo m .

$$h(\text{"Ain"}) = 65 + 105 + 110 \pmod{50} = 280 \pmod{50} = 30$$

$$h(\text{"Essonne"}) = 31$$

$$h(\text{"La réunion"}) = 5.$$

$$h(\text{"Nord"}) = 3$$

$$h(\text{"Drome"}) = 3$$

Après insertion de la table des départements, en mémoire:

- $\text{tab}[30]$ contient "Ain"
- $\text{tab}[31]$ contient "Essonne"
- $\text{tab}[5]$ contient "La réunion"
- $\text{tab}[3]$ contient "Nord" et "Drome" (une collision).

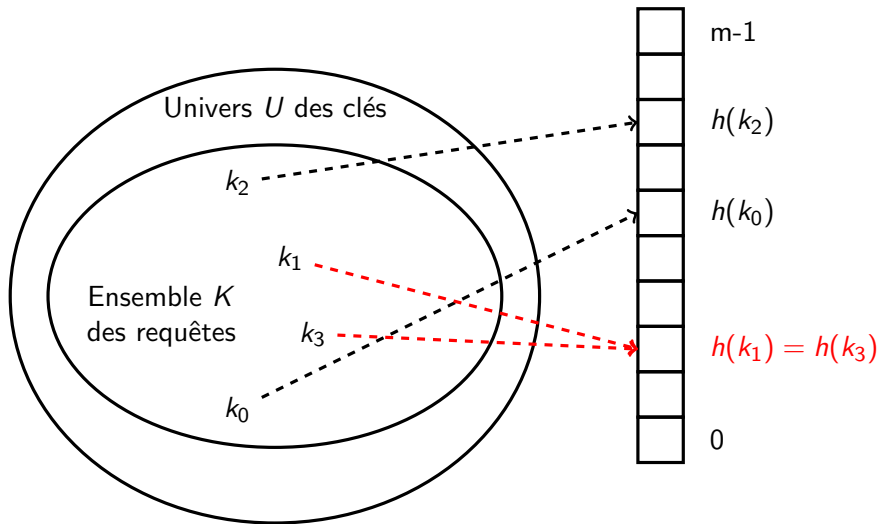
Digression: ASCII

- American Standard Code for Information Interchange
- Les caractères sont codés sur 7 bits:
 - ▶ 128 caractères possibles de 0 à 127.
- Majuscules: *A*: 65, *B*: 66, ..., *Z*: 90
- Minuscules: *a*: 97, *b*: 98, ..., *z*: 122
- Autres: : 32, ...

```
char c='B';  
System.out.println((int) c);
```

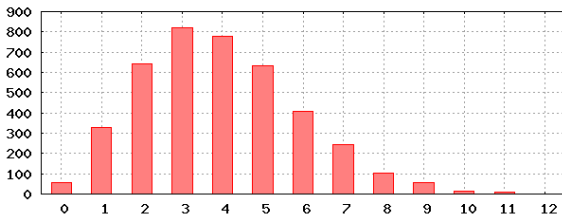
Affiche: *66*

Hachage: collisions



Une fonction de hachage idéale

- Une fonction de hachage pas très bonne:



- On veut que h minimise les collisions en moyenne:
 - ▶ Hypothèse de hachage uniforme: par définition,

$$\sum_{\{k|h(k)=j\}} P(k) = \frac{1}{m}.$$

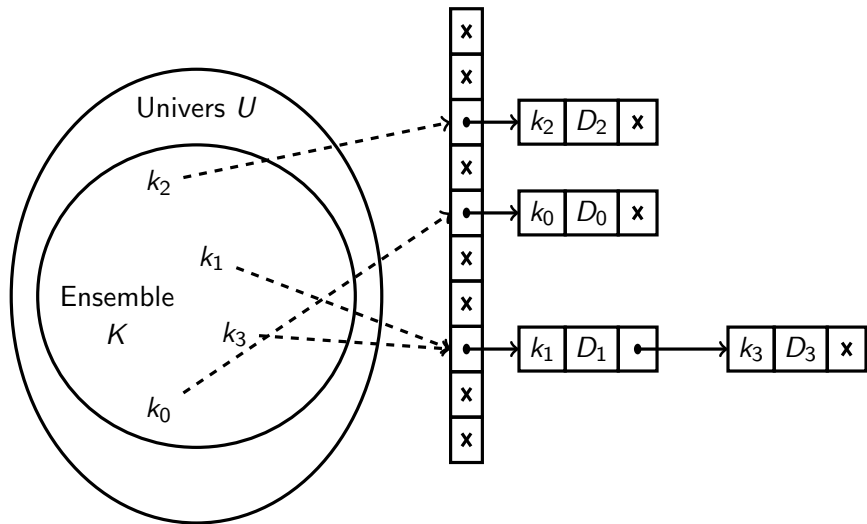
- ▶ Sous cette hypothèse, le nombre moyen de collisions par alvéole vaut $\alpha = n/m$.
- ▶ Paramètre naturel: $\alpha = n/m$, le facteur de charge.

Résoudre les collisions

Plusieurs grandes techniques pour résoudre les collisions

- *Le chaînage*: les données correspondant à une même valeur de hachage sont stockées dans une liste.
- *L'adressage ouvert*: on place les données dans la première place libre dans le tableau.

Hachage par chaînage: principe



Hachage par chaînage: complexité

- Insérer(k, D) se réalise en $O(1)$.
- Rechercher(k):
 - ▶ Au pire cas: $O(n)$.
 - ▶ En moyenne:
 - $O(1 + \alpha)$, où $\alpha = n/m$ est le facteur de charge, avec une hypothèse de hachage uniforme.
- Supprimer(k): même chose, car il suffit de trouver l'élément, et de le supprimer (suppression en $O(1)$).

Hachage par chaînage: complexité

- Insérer(k, D) se réalise en $O(1)$.
- Rechercher(k):
 - ▶ Au pire cas: $O(n)$.
 - ▶ En moyenne:
 - $O(1 + \alpha)$, où $\alpha = n/m$ est le facteur de charge, avec une hypothèse de hachage uniforme.
 - Soit $O(1)$, si l'on maintient α constant, c'est-à-dire m proportionnel à n .
- Supprimer(k): même chose, car il suffit de trouver l'élément, et de le supprimer (suppression en $O(1)$).

Hachage par chaînage: gestion des débordements

- Si l'on veut maintenir un facteur de charge raisonnable, il faut parfois redimensionner la table.
- Comment redimensionner:
 - ▶ on double la taille de la table.
 - ▶ on modifie la fonction de hachage.
 - ▶ on réaffecte le contenu de l'ancienne table dans la nouvelle avec cette nouvelle fonction de hachage.

Hachage par adressage ouvert

- Plutôt que de maintenir des listes externes au tableau, on utilise directement les alvéoles libres.
- Principe de $\text{Insérer}(k, D)$
 - ▶ Si l'alvéole $h(k)$ est vide, on l'utilise.
 - ▶ Sinon, pour $r = 1, 2, \dots$
 - Si l'alvéole $h(k) + r \bmod m$ est vide on l'utilise.
 - ▶ Si l'on suppose un facteur de charge $\alpha < 1$, on finira par tomber sur une telle alvéole.

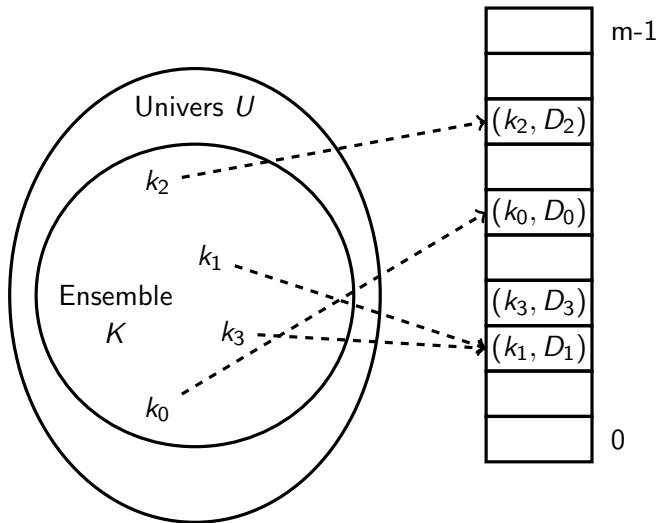
Hachage par adressage ouvert

- Plutôt que de maintenir des listes externes au tableau, on utilise directement les alvéoles libres.
- Principe de Insérer(k, D)
 - ▶ Si l'alvéole $h(k)$ est vide, on l'utilise.
 - ▶ Sinon, pour $r = 1, 2, \dots$
 - Si l'alvéole $h(k) + r \bmod m$ est vide on l'utilise.
 - ▶ Si l'on suppose un facteur de charge $\alpha < 1$, on finira par tomber sur une telle alvéole.
- Principe de Rechercher(k)
 - ▶ Si la clé de l'alvéole $h(k)$ correspond à k , retourner son contenu.
 - ▶ Sinon, pour $r = 1, 2, \dots$
 - Si l'alvéole $h(k) + r \bmod m$ correspond à k , on retourne son contenu.
 - ▶ On arrête la boucle dès que l'on tombe sur une alvéole vide (ce qui est garanti si $\alpha < 1$).

Hachage par adressage ouvert

- Plutôt que de maintenir des listes externes au tableau, on utilise directement les alvéoles libres.
- Principe de Insérer(k, D)
 - ▶ Si l'alvéole $h(k)$ est vide, on l'utilise.
 - ▶ Sinon, pour $r = 1, 2, \dots$
 - Si l'alvéole $h(k) + r \bmod m$ est vide on l'utilise.
 - ▶ Si l'on suppose un facteur de charge $\alpha < 1$, on finira par tomber sur une telle alvéole.
- Principe de Rechercher(k)
 - ▶ Si la clé de l'alvéole $h(k)$ correspond à k , retourner son contenu.
 - ▶ Sinon, pour $r = 1, 2, \dots$
 - Si l'alvéole $h(k) + r \bmod m$ correspond à k , on retourne son contenu.
 - ▶ On arrête la boucle dès que l'on tombe sur une alvéole vide (ce qui est garanti si $\alpha < 1$).
- Supprimer(k): plus compliqué. Il faut tout "réhacher".

Hachage par adressage ouvert: principe



Amélioration: double hachage

- La méthode précédente a tendance à saturer les alvéoles dont l'indice est proche d'un indice correspondant à une collision.
- Une solution: le double hachage:
 - ▶ On remplace dans la méthode précédente $h(k) + r \pmod m$, par $h(k) + h'(r) \pmod m$, où h' est une autre fonction de hachage.
- Théorie + Expérimentation: Nombre moyen de comparaisons avec des hypothèses raisonnables (distribution uniforme, fonctions de hachage uniformes et indépendantes).

| Facteur de charge | 50 % | 80 % | 90 % | 99 % |
|-------------------|------|------|-------|--------|
| Succès | 1.39 | 2.01 | 2.56 | 4.65 |
| Echec | 2.00 | 5.00 | 10.00 | 100.00 |

- $\alpha = 0.8$ est un bon compromis (performances équivalentes à $\alpha = 0.5$ dans le cas linéaire).

Comment obtenir une fonction de hachage idéale?

- Toutes les contraintes sur la fonction de hachage idéale ne sont pas satisfiables en pratique,

Comment obtenir une fonction de hachage idéale?

- Toutes les contraintes sur la fonction de hachage idéale ne sont pas satisfiables en pratique,
 - ▶ En particulier, obtenir un hachage uniforme parfait suppose de connaître P , la distribution des clés des requêtes qui seront utilisées,
 - ▶ ce qui est un problème (au mieux) statistique souvent difficile à appréhender.

Comment obtenir une fonction de hachage idéale?

- Toutes les contraintes sur la fonction de hachage idéale ne sont pas satisfiables en pratique,
 - ▶ En particulier, obtenir un hachage uniforme parfait suppose de connaître P , la distribution des clés des requêtes qui seront utilisées,
 - ▶ ce qui est un problème (au mieux) statistique souvent difficile à appréhender.
- mais on peut chercher à s'en approcher, ou supposer qu'on s'en approche.

L'art du hachage

Quelques bons candidats (si distribution uniforme)

- pour $U \subset \mathbb{N}$:
 - ▶ $h(k) = k \bmod m$, avec m un nombre premier.
 - ▶ $h(k) = k(k + 3) \bmod m$, avec m un nombre premier.
 - ▶ $h(k) = \lfloor ((k * \theta) \bmod 1) * m \rfloor$, où $0 < \theta < 1$ est un nombre réel, m un entier quelconque.
 - $\theta = \frac{\sqrt{5}-1}{2}$ ou $\theta = 1 - \frac{\sqrt{5}-1}{2}$ donnent de bons résultats en théorie (évitent les accumulations aux extrémités du tableau).
- pour des chaînes de caractères:

La solution de JAVA par défaut dans la classe String:

```
public int hashCode() {  
    int h = 0 ;  
    for (int k = 0 ; k < this.length ; k++)  
        h = 31 * h + this.charAt(k) ;  
    return h ;  
}
```

- ▶ `this.charAt(k)` correspond au codage ASCII du caractère numéro k de la chaîne.

Aujourd'hui

La structure dictionnaire

Le hachage

Associations & Bibliothèques standards JAVA

Les tables de hachage de la bibliothèque

- Les tables de hachage sont tellement utilisées qu'elles sont proposées dans les classes JAVA standards.
- Elles correspondent à la classe générique à deux paramètres `HashMap<K,V>` du package `java.util`.

```
import java.util.*;  
...  
HashMap<String,Integer> t = new HashMap<String,Integer> ();
```

Les tables de hachage de la bibliothèque

- Les tables de hachage sont tellement utilisées qu'elles sont proposées dans les classes JAVA standards.
- Elles correspondent à la classe générique à deux paramètres `HashMap<K,V>` du package `java.util`.
- Attention: `K` et `V` doivent être des classes.

```
import java.util.*;  
...  
HashMap<String,Integer> t = new HashMap<String,Integer> ();
```

Les tables de hachage de la bibliothèque

- Les tables de hachage sont tellement utilisées qu'elles sont proposées dans les classes JAVA standards.
- Elles correspondent à la classe générique à deux paramètres `HashMap<K,V>` du package `java.util`.
- Attention: `K` et `V` doivent être des classes.
- Et donc
 - ▶ `Integer` plutôt que `int`, `Char` plutôt que `char`,...,etc
 - ▶ et aussi `int.Value()`, `Integer.valueOf()`, etc.

```
import java.util.*;  
...  
HashMap<String,Integer> t = new HashMap<String,Integer> ();
```

Les associations en pratique

- `HashMap<K, V>` réalise des associations entre instances de la classe `K` (des clés), et instances de la classe `V` (des données).
- On peut voir cela comme un tableau indexé par des instances de la classe `K`.
- La méthode `V put(K k, V D)`:
 - ▶ insère l'association entre la clé `k` et la donnée `D`.
 - ▶ retourne la précédente donnée associée à `k`, ou `null` s'il n'y en avait pas.
- La méthode `V get(K k)` retourne la donnée associée à la clé `k` si elle existe, `null` sinon.
- La méthode `remove(K k)` supprime l'association dont la clé est `k`.

Exemple

```
import java.util.*;

class hach {
    public static void main(String[] args) {
        HashMap<String,Integer> t= new HashMap<String,Integer>();

        t.put("Rhone",69);
        t.put("Yvelines",78);
        t.put("Essonne",91);
        t.put("Meurthe et Moselle",54);
        t.put("Doubs",25);

        System.out.println(t.get(args[0]));    }}

```

java hach Essonne

Affiche: 91

Une application: la génération automatique de textes (non sérieux)

- On va supposer que l'enchaînement des mots dans un texte suit un modèle de Markov:
 - ▶ deux mots w_1 , w_2 qui se suivent, déterminent le mot w_3 qui les suit.
 - ▶ ce mot w_3 sera choisi au hasard, uniformément, parmi les mots w_3 qui suivent w_1 et w_2 dans un texte de référence.
- Partant du texte de référence donné par le chapitre sur le hachage dans le poly, on obtient par exemple:

Le fait pertinent est de remarquer que le compilateur vérifie. Un objet L peut prendre m égal à une expérience consistant à appliquer le programme Freq qui compte le nombre de leurs occurrences. Supposons les premier et troisième points résolus. Le troisième par un tri et le premier par une information associée à k. Selon cette technique, une fois entrée dans la table, alors la nouvelle classe H déclare implémenter l'interface Assoc (mot-clé implements). Cette déclaration entraîne deux conséquences importantes.

Construire la table 1/3

- Il nous faut construire une table qui associe à chaque couple de mots (w_1, w_2) la liste des mots w_3 qui les suivent dans le texte de référence.
- On suppose donnée une classe `WordReader`
 - ▶ avec un constructeur `WordReader(String nom)` qui prend comme argument un nom de fichier `nom`.
 - ▶ et une méthode `String next()` qui renvoie le prochain mot dans ce fichier (ou `null` s'il n'y en a plus).

Construire la table 2/3

- Il nous faut une classe pour les paires de mots.

```
class Paire {  
    String w1, w2 ;  
  
    Paire () {  
        this.w1 = "" ; this.w2 = "" ;  
    }  
  
    Paire (String w1, String w2) {  
        this.w1 = w1 ; this.w2 = w2 ;  
    }  
}
```

Construire la table 3/4

- Il faut une table qui associe à une paire de mots une liste de mots.

```
HashMap<Paire,LinkedList<String>>  
tab = new HashMap<Paire,LinkedList<String>> () ;
```

- ▶ On utilise ici la classe générique `LinkedList<E>` de la bibliothèque, pour gérer les listes. La méthode `addFirst(E e)` de cette classe ajoute en tête de liste.

Construire la table 4/4

- Il suffit alors de parcourir les mots, et de les insérer:

```
WordReader wr = new WordReader(arg[0]) ;
Paire k = new Paire () ; String D ;
while ((D = wr.next()) ≠ null) {
    // A ce moment, k.w1, k.w2, D sont trois mots consécutifs
    // On récupère la liste correspondante dans l
    LinkedList<String> l = tab.get(k) ;
    // Si elle n'existe pas encore, on la crée
    if (l == null) l = new LinkedList<String> () ;
    // On ajoute le mot dans la liste l
    l.addFirst(D);
    // On remet la liste dans la table
    tab.put(k,l) ;
    // Et on passe au mot suivant
    k = new Paire(k.w2,D);}}
```

- La table est maintenant construite en mémoire.

Générer le texte 1/2

- Il ne reste plus qu'à générer le texte...

```
Paire p = new Paire ("Ce","chapitre") ;  
// Les premiers mots du chapitre sont ''Ce chapitre aborde''  
System.out.print(p.w1 + " " + p.w2) ;  
for (int i=0 ; i ≤100; i++) {  
    // Récupérer le prochain mot  
    String w = suivant(tab,p) ;  
    if (w == null) return ;  
    System.out.print(w + " ") ;  
    p.w1=p.w2; p.w2=w;  
}
```

- ... c'est-à-dire à écrire la fonction suivant

Générer le texte 2/2

- La fonction suivant.

```
import java.util.*
...
static Random rand = new Random () ;
...

static String suivant(HashMap<Paire,LinkedList<String>> tab,
Paire p) {
    // On récupère la liste des mots qui suivent p.w1 et p.w2
    // dans le texte de référence
    LinkedList<String> l = tab.get(p) ;
    if (l==null) return null;
    // On en choisit un selon une loi uniforme
    int size=l.size();
    return l.get(rand.nextInt(size)) ;
}
```

Attention!

- Le code précédent compile, mais ne marche pas en pratique (rien n'est produit à part le texte "Ce chapitre").
- Pour utiliser un `HashMap<K,V>`, il faut s'assurer que
 - ▶ la méthode `equals` est correctement définie pour les objets de la classe `K`.
 - la redéfinir si besoin avec un code du type:

```
public boolean equals(Object o) {  
    Paire p = (Paire)o ;  
    return w1.equals(p.w1) && w2.equals(p.w2) ;}
```

- ▶ la méthode `hashCode` est correctement définie pour les objets de la classe `K`.
 - la redéfinir si besoin, en définissant:

```
public int hashCode() {...}
```

- ▶ Un contrat avec JAVA:
 - lorsque `k.equals(k')` est vrai, on doit avoir `k.hashCode() == k'.hashCode()` pour `k` et `k'` de type `K`.

Générer le texte 3/2

- La solution: dire explicitement à JAVA comment comparer et hacher les clés.

```
class Paire {
    String w1, w2 ;

    Paire () { this.w1 = "" ; this.w2 = "" ; }

    Paire (String w1, String w2) {this.w1 = w1;
        this.w2 = w2 ;}

    public boolean equals(Object o) {
        Paire p = (Paire)o ;
        return w1.equals(p.w1) && w2.equals(p.w2) ;}

    public int hashCode() {
        return 37 * w1.hashCode() + w2.hashCode() ;}
}
```

Cette fois...

Ce chapitre aborde un problème très fréquemment rencontré en pratique. Dans un deuxième temps, il est plus convenable, et ce n'est pas gratuit, il est plus convenable, et ce n'est pas un cas qui se présente en pratique avec le hachage modulo un nombre premier, et ce n'est pas un cas aussi simple est exceptionnel en pratique. Le fait pertinent est de considérer les chaînes comme des entiers pris dans un `Assoc t = new HashMap<String,Integer> () ;` } La table H basée sur le hachage à adressage ouvert `class O implements Assoc { final static int SIZE = 1024 ;`

Object?

- Tout ce qui n'est pas primitif est un Object.
- Tout objet hérite des méthodes définies par la classe Object.
- Ces méthodes peuvent être redéfinies.
- On l'a déjà fait pour toString

```
class Personne
{ String nom;
  int age;
  public String toString() {
    return ("nom: " + nom + " age: " + age);
  }}
```

- D'autres méthodes utiles:
 - ▶ **public boolean** equals(Object o) qui gère les comparaisons d'égalité.
 - ▶ **public int** hashCode() qui dit comment hacher.

Attention

1.

```
public boolean equals(Paire p) {  
    return w1.equals(p.w1) && w2.equals(p.w2) ;}
```

ne fonctionne pas (car cela définit la méthode `equals` avec la signature `public boolean equals(Paire p)`, ce qui ne surcharge pas/rédéfinit pas la méthode

2.

```
public boolean equals(Object o) {  
    return w1.equals(o.w1) && w2.equals(o.w2) ;}
```

ne compile pas (la classe `Object` ne possède pas de champ `w1` ni `w2`).

3.

```
public boolean equals(Object o) {  
    Paire p = (Paire)o ;  
    return w1.equals(p.w1) && w2.equals(p.w2) ;  
}
```

est la version correcte (on a besoin de dire explicitement que l'argument est une `Paire`).

Pour comprendre: un peu plus sur `equals` 1/2

```
class Paire {  
    String w1, w2 ;  
    Paire (String w1, String w2) {  
        this.w1 = w1 ; this.w2 = w2 ;  
    }  
}
```

```
...  
Paire p= new Paire ("un","deux");  
Paire q= new Paire ("un","deux");  
String s="un", t="un";  
System.out.println(p.equals(q)+  
    ", "+s.equals(t));
```

Affiche: *false,true*

- Pourquoi?

Pour comprendre: un peu plus sur `equals` 1/2

```
class Paire {  
    String w1, w2 ;  
    Paire (String w1, String w2) {  
        this.w1 = w1 ; this.w2 = w2 ;  
    }  
}
```

```
...  
Paire p= new Paire ("un","deux");  
Paire q= new Paire ("un","deux");  
String s="un", t="un";  
System.out.println(p.equals(q)+  
    ", "+s.equals(t));
```

Affiche: *false,true*

■ Pourquoi?

- ▶ Lors de l'appel d'une méthode, JAVA cherche une déclaration de méthode qui corresponde à cette signature.
- ▶ Ici, il n'y a pas de méthode `equals` dans la classe `Paire`.
- ▶ Puisque tout objet, et donc `Paire`, hérite de la classe `Object`, JAVA cherche dans la classe `Object`. Or dans `Object` **class** on peut lire:

```
public boolean equals(Object obj) {return (this==obj);};
```

Pour comprendre: un peu plus sur `equals` 1/2

```
class Paire {  
    String w1, w2 ;  
    Paire (String w1, String w2) {  
        this.w1 = w1 ; this.w2 = w2 ;  
    }  
}
```

```
...  
Paire p= new Paire ("un","deux");  
Paire q= new Paire ("un","deux");  
String s="un", t="un";  
System.out.println(p.equals(q)+  
    ", "+s.equals(t));
```

Affiche: *false,true*

■ Pourquoi?

- ▶ Lors de l'appel d'une méthode, JAVA cherche une déclaration de méthode qui corresponde à cette signature.
- ▶ Ici, il n'y a pas de méthode `equals` dans la classe `Paire`.
- ▶ Puisque tout objet, et donc `Paire`, hérite de la classe `Object`, JAVA cherche dans la classe `Object`. Or dans `Object.class` on peut lire:

```
public boolean equals(Object obj) {return (this==obj);};
```

- L'implémentation par défaut est donc l'égalité physique.

Pour comprendre: un peu plus sur `equals` 1/2

```
class Paire {  
    String w1, w2 ;  
    Paire (String w1, String w2) {  
        this.w1 = w1 ; this.w2 = w2 ;  
    }  
}
```

```
...  
Paire p= new Paire ("un","deux");  
Paire q= new Paire ("un","deux");  
String s="un", t="un";  
System.out.println(p.equals(q)+  
    ", "+s.equals(t));
```

Affiche: *false,true*

■ Pourquoi?

- ▶ Lors de l'appel d'une méthode, JAVA cherche une déclaration de méthode qui corresponde à cette signature.
- ▶ Ici, il n'y a pas de méthode `equals` dans la classe `Paire`.
- ▶ Puisque tout objet, et donc `Paire`, hérite de la classe `Object`, JAVA cherche dans la classe `Object`. Or dans `Object` **class** on peut lire:

```
public boolean equals(Object obj) {return (this==obj);};
```

- L'implémentation par défaut est donc l'égalité physique.
- Pour les `String`, la méthode `equals` est redéfinie.

Pour comprendre: Un peu plus sur equals 2/2

```
class Paire {  
    String w1, w2 ;  
    Paire (String w1, String w2) {  
        this.w1 = w1 ; this.w2 = w2 ;  
    }  
    public boolean equals(Paire p){  
        return w1.equals(p.w1)  
        && w2.equals(p.w2) ;  
    } }  
}
```

```
...  
Paire p= new Paire ("un","deux");  
Paire q= new Paire ("un","deux");  
String s="un", t="un";  
System.out.println(p.equals(q)+  
    ", "+s.equals(t));
```

Affiche: *true,true*

- Ce problème est donc résolu.

Pour comprendre: Un peu plus sur equals 2/2

```
class Paire {  
    String w1, w2 ;  
    Paire (String w1, String w2) {  
        this.w1 = w1 ; this.w2 = w2 ;  
    }  
    public boolean equals(Paire p){  
        return w1.equals(p.w1)  
            && w2.equals(p.w2) ;  
    } }  
}
```

```
...  
Paire p= new Paire ("un","deux");  
Paire q= new Paire ("un","deux");  
String s="un", t="un";  
System.out.println(p.equals(q)+  
    ", "+s.equals(t));
```

Affiche: *true,true*

- Ce problème est donc résolu.
- Bilan:
 - ▶ Utiliser equals est toujours mieux que d'utiliser ==.
 - ▶ Mais ce n'est pas toujours suffisant.

Pour comprendre: Un peu plus sur `equals` 2/2

```
class Paire {  
    String w1, w2 ;  
    Paire (String w1, String w2) {  
        this.w1 = w1 ; this.w2 = w2 ;  
    }  
    public boolean equals(Paire p){  
        return w1.equals(p.w1)  
        && w2.equals(p.w2) ;  
    } }  
}
```

```
...  
Paire p= new Paire ("un","deux");  
Paire q= new Paire ("un","deux");  
String s="un", t="un";  
System.out.println(p.equals(q)+  
    ", "+s.equals(t));
```

Affiche: *true,true*

- Ce problème est donc résolu.
- Bilan:
 - ▶ Utiliser `equals` est toujours mieux que d'utiliser `==`.
 - ▶ Mais ce n'est pas toujours suffisant.
- Mais alors, pourquoi ne fait on pas comme cela?

Le code de la librairie standard

```
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    int hash = hash(key.hashCode());
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
        e ≠ null;
        e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key
            || key.equals(k)))
            return e.value;}
    return null;}

```

- Il est écrit `key.equals(k)`, où `k` et `key` sont des `Objects`.

Le code de la librairie standard

```
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    int hash = hash(key.hashCode());
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
        e ≠ null;
        e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key
            || key.equals(k)))
            return e.value;}
    return null;}
}
```

- Il est écrit `key.equals(k)`, où `k` et `key` sont des `Objects`.
- Même en ayant défini `public boolean equals(Paire p)` dans la classe `Paire`, JAVA ira chercher la déclaration `public boolean equals(Object obj)` dans la classe `Object`, car `key` n'est pas une `Paire`, mais un `Object`.

Le code de la librairie standard

```
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    int hash = hash(key.hashCode());
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
        e ≠ null;
        e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key
            || key.equals(k)))
            return e.value;}
    return null;}

```

- Il est écrit `key.equals(k)`, où `k` et `key` sont des `Objects`.
- Même en ayant défini `public boolean equals(Paire p)` dans la classe `Paire`, JAVA ira chercher la déclaration `public boolean equals(Object obj)` dans la classe `Object`, car `key` n'est pas une `Paire`, mais un `Object`.
- L'égalité utilisée est donc toujours l'égalité physique.

Que retenir de tout cela?

- Utiliser `equals` est toujours mieux que d'utiliser `==`.
- Pour utiliser un `HashMap<K,V>`, il faut s'assurer que
 - ▶ la méthode `equals` est correctement définie pour les objets de la classe `K`.
 - la redéfinir si besoin avec un code du type:

```
public boolean equals(Object o) {  
    Paire p = (Paire)o ;  
    return w1.equals(p.w1) && w2.equals(p.w2) ;}
```

- ▶ la méthode `hashCode` est correctement définie pour les objets de la classe `K`.
 - la redéfinir si besoin, en définissant:

```
public int hashCode() {...}
```

- ▶ Un contrat avec JAVA:
 - lorsque `k.equals(k')` est vrai, on doit avoir `k.hashCode() == k'.hashCode()` pour `k` et `k'` de type `K`.
- Quelque part, la "vraie" sémantique de `get` implémentée est:
 - ▶ La méthode `V get(K k)` retourne la donnée associée à une clé `k'`, si il existe une clé `k'` avec `k'.equals(k)`, **null** sinon.