Tableau de hachage

TA table de symboles

table de symboles / dictionnaire / *symbol table* / *dictionary* / *mapping*

Type abstrait d'une **table de symboles** (*symbol table*) ou dictionnaire

Objets : ensembles d'éléments avec clés
typiquement : clés comparables (abstraction : nombres naturels)

- ⋆ search(k) : recherche d'un élément à clé k ← peut être fructueuse ou infructueuse

Si dictionnaire modifiable («dynamique») :
⋆ insert(x) : insertion de l'élément x (clé+info)
⋆ delete(k) : supprimer élément avec clé k
Avec clés uniques : search(k) retourne l'élément x avec clé k si un tel existe, ou null sinon ; insert(x) *remplace* l'élément avec la même clé, si un tel est déjà présent dans l'ensemble.

Java

`java.util.Map` et `java.util.Set`

```
public interface Map<K,V>
{
    public V get(Object key);
    public boolean containsKey(Object key);
    public V put(K key, V value); // à option
    public V remove(Object key); // à option
    ...
```

```
public interface Set<E>
{
    public boolean contains(Object key);
    public boolean add(E element); // à option
    public boolean remove(Object key); // à option
    ...
```

`AbstractSet`, `AbstractCollection` et `AnstractMap` demandent seulement l'implémentation d'un `iterator` dans les sous-classes

Inverted index

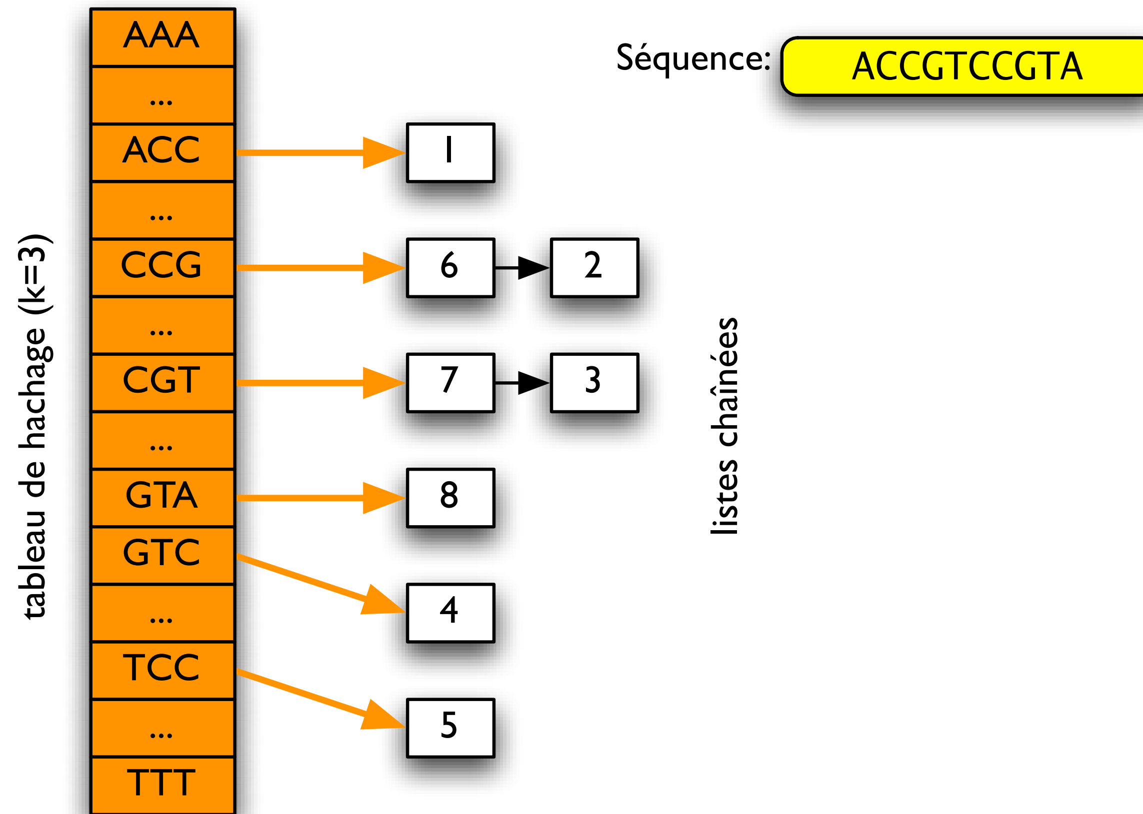
si clés entières $\mathcal{U} = \{0, 1, 2, \ldots, M-1\}$

utiliser un tableau $\mathsf{T}[0..M-1]$, et mettre l'élément avec clé k dans la k-ème cellule

☞opérations en $O(1)$ mais mémoire de taille $\Theta(M)$
☞utile que si M n'est pas trop grand.

clés multiples ? on peut accommoder des éléments avec clés multiples (p.e., personnes avec date d'anniversaire — mois, jour — comme clé) : mettre des listes dans les cellules.

Tableau de k-mers



Ma, Tromp et Li. *Bioinformatics* 18 :440.

Inverted index

séquence de valeurs $y_0, y_1, \ldots, y_{n-1} \in \{0, 1, \ldots, M-1\}$
on veut une opération $\mathsf{search}(y)$ qui retourne toute position i où $y_i = y$.

structure de données : dictionnaire $i \rightarrow \mathsf{liste}$?
clés entières $0..M-1$, indices entiers $0..n-1$

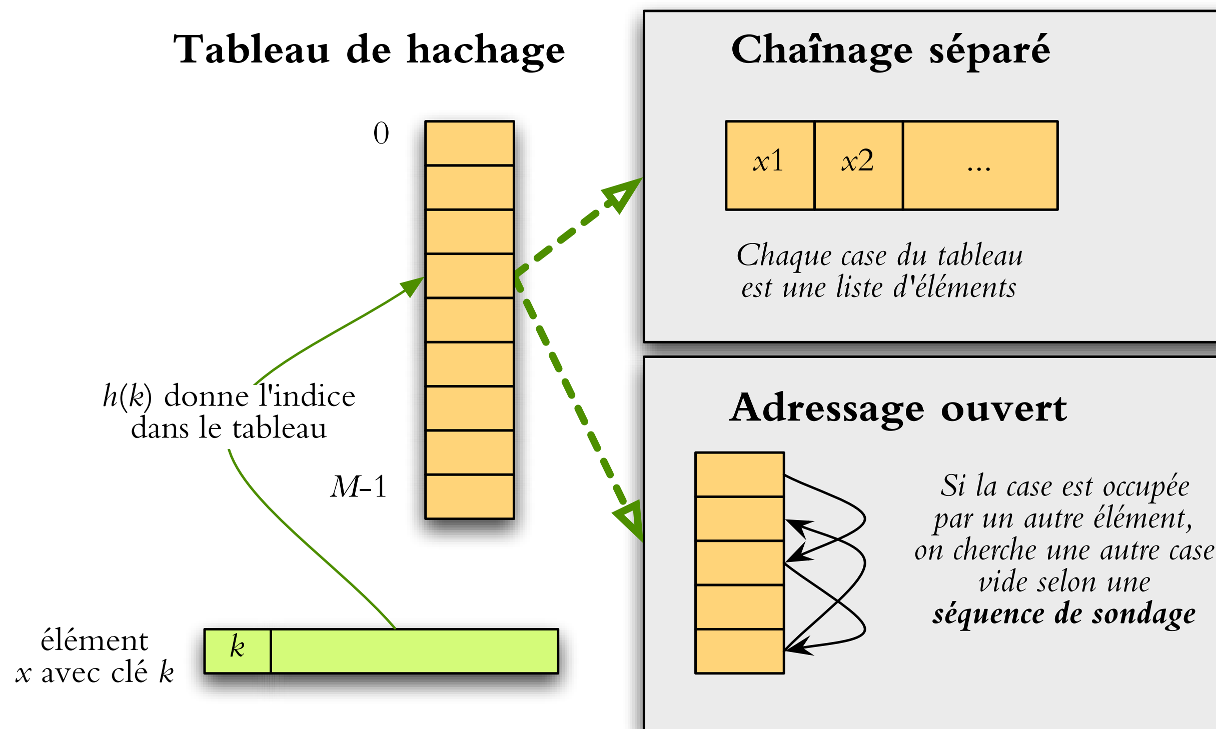
```
1  initialiser head[0..M − 1] = (−1, −1, ..., −1)
2  initialiser next[0..n − 1]
3  for i ← 0, 1, 2 ..., n − 1 do
4       next[i] ← head[y_i] ; head[y_i] ← i          // insertion à la tête

   SEARCH(y)
S1  L ← ∅
S2  i ← head[y]
S3  while i ≠ −1 do
S4       L.add(i) ; i ← next[i]                      // positions en ordre décroissant
S5  return L
```

Exemple de chromosome 1 humain : $n = 250 \cdot 10^6$, 12-mers $M = 4^{12}$

Tableau de hachage

On peut accommoder des clés non-entières ou un espace de clés $\mathcal{U}$ trop grand : utiliser une **fonction de hachage** $h\colon \mathcal{U} \mapsto \{0, 1, \ldots, M-1\}$ pour définir l'emplacement de clé $k \in \mathcal{U}$ dans un tableau $H[0..M-1]$.



Fonctions de hachage

on veut éviter les collisions : disperser les éléments autant que possible

- ⋆ [uniformité] $h(k) = i$ avec probabilité $1/M$ pour tout $i = 0, 1, \ldots, M - 1$.
- ⋆ [indépendance] pour un ensemble de clés $k_1, k_2, \ldots, k_m$, l'ensemble de valeurs de hachage $\big(h(k_1), \ldots, h(k_m)\big)$ a une distribution uniforme.

en vie réelle : on ne sait pas la distribution de k

Méthode de la division

Méthode de la division : $h(k) = k \bmod M$

Choix de M : on veut éviter la réduction de l'espace de valeurs de hachage à cause des données non-aléatoires

– ne devrait pas être $M = 2^k$
  (car les derniers k bits de la clé determinent la valeur de hachage)
– puissances de 10 à éviter
  (car hachage réduit pour nombres décimaux)
– ne devrait pas être un multiple de 3
  (car permutations d'une clé binaire donnent des valeurs de hachage qui diffèrent par multiples de 3)

Règle : choisir un prime qui est loin des pouvoirs de 2 et de 10

Méthode de la multiplication

Méthode de la multiplication : $h(k) = \lfloor M\{\gamma k\}\rfloor$
(partie fractionnaire : $\{x\} = x - \lfloor x\rfloor$).

La dispersion des valeurs de hachage dépend principalement de γ.
Un bon choix est $\gamma = \frac{\sqrt{5}-1}{2}$.

Poser $\gamma = \frac{A}{2^w}$
où w est la longueur d'un mot-machine ($w = 32$ pour Java `int`)

Choix de M : $M = 2^p$ (p bits de l'ordre supérieur de la moitié inférieure de Ak)

```
int h = (A*k) >>> (w-p);
```

Clés composées — hachage universel

On a une clé de r caractères : $k = \langle k_1, k_2, \ldots, k_r \rangle$ (p.e., `String`, ou objet avec r variables d'instance).

Fonction de hachage

$$h^{(r)}(k) = \left(\sum_{i=1}^{r} h_i(k_i) \right) \bmod M$$

avec des fonctions de hachage $h_i(x)$ choisies «au hasard».

En pratique, on utilise une règle approximative simple comme $h_i(x) = a^{r-i} \cdot x$ (où a est un nombre premier) :

initialiser $h \leftarrow 0$; **for** $i \leftarrow 1, 2, \ldots, r$ **do** $h \leftarrow (a \cdot h + k_i) \bmod M$.

String hashcode

P.e., **hashcode**() de `String` (Java) utilise $a = 31$ et $M = 2^{32}$ (32-bit entiers).

```
class String {
    ...
    final char[] value; final int offset; final int count;
    ...
    public int hashCode()
    {
        ...
        int hashCode = 0;
        int limit = count + offset;
        for (int i = offset; i < limit; i++)
            hashCode = hashCode * 31 + value[i];
        return hashCode;
    }
```

Collisions

Fonction de hachage : $h\colon U \mapsto \{0, 1, \ldots, M-1\}$

collision : $h(k) = h(k')$ mais $k \neq k'$

Birthday paradox (clés aléatoires) : collisions avec probabilité non-negligeable à moins que $n = o(\sqrt{M})$.

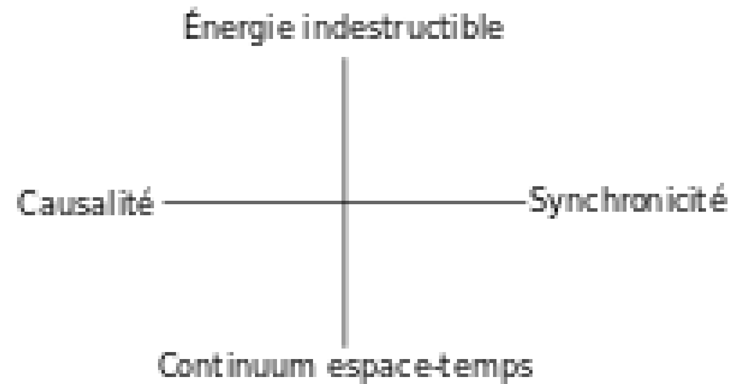
Probabilité d'aucune collision

$$p = 1\Bigl(1-\frac{1}{M}\Bigr)\Bigl(1-\frac{2}{M}\Bigr)\cdots\Bigl(1-\frac{n-1}{M}\Bigr)$$
$$< \prod_{i=0}^{n-1} e^{-i/M} = \exp\Bigl(-\frac{(n-1)n}{2M}\Bigr)$$

si $n > \sqrt{M 2 \ln 2} \approx 1.18\sqrt{M}$, alors au moins une collision avec proba $\geq 1/2$.

Détour : coïncidences remarquables

syncronicité . . . phénomène psychologique/ésotérique (Jung / Pauli)



http://understandinguncertainty.org/coincidences

Understanding Uncertainty

Home Blog Articles Videos Animations Guest Articles Links About Us

Home

Cambridge Coincidences Collection

Coïncidences

Six Flags Visit

On opening day (around April 2017) of Six Flags in Gurnee, IL I was in line with about a thousand people and there was a lady with her son in front of my son, nephew, and me. We talked for awhile until we entered the park and then went Are own ways. Forward 6 months later (October 2017) I take my son, nephew, and myself back to Six Flags in Gurnee, IL for Freight Fest that was planned a couple months ago. Well guess who I see at the park??? Exactly The lady and her son that I talked to in line for an hour on opening day.

Rate this.

Read more

lost two relatives on the same day 200 miles apart

On 28.11.96, my maternal grandmother passed away in the morning. At 7pm that evening, my father passed away, 200 miles away. I then met a man in 2010 who'd lost his mother in the morning and his father in the evening through unrelated events.

Rate this.

Read more

Some coincidences take a long time

When I was around 15 or 16, I'd listen to Radio Luxembourg in the late evenings. I didn't care much for the music, but the news reports by an IRN reporter from El Salavador were gripping. A few years later, I met the love of my life, who, incidentally, lived in the house next door to the one I'd been born in and grew up in, although I'd moved away by then. One day, one of us raised how impactful listiening to the IRN El Salavador news reports by one particular news reporter had been, and the other said they too had listened to them and remembered them.

Rate this.

Read more

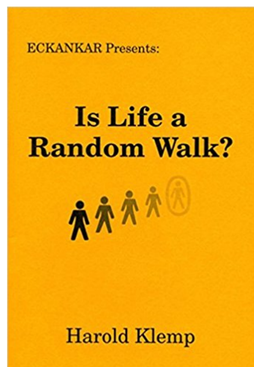
Coïncidences remarquables

probabilité + effet psychologique

How would you rank your coincidence story?

- ◯ N/A
- ◯ * Quite surprising (eg would only expect to happen to me once a year)
- ◯ ** Very surprising (eg a once-in-a-lifetime event)
- ◯ *** Extraordinary (eg once-in-a-hundred-lifetimes)
- ◯ **** Eerily bizarre (amazed this would ever happen to anybody)

Le hasard dans la vie réelle



[réponse de l'auteur] *Soul's whole purpose for being in this world is to find divine love.*

Each of us is connected to God through Divine Spirit (the ECK), which can be heard as Sound and seen as Light. Eckankar offers a spiritual toolkit to help you experience the Light and Sound of God.

Hypothèse testable ! (le hasard, et non pas Eckankar) :
événements rares et indépendants $A_1, A_2, \ldots$;
nombres d'événements $N = \sum_i A_i$ (variable aléatoire)

- ⋆ moyen $\mu = \sum_i \mathbb{P}A_i$
- ⋆ $\mathbb{P}\{N = 0\} \sim e^{-\mu}$
- ⋆ distribution $N \sim \mathsf{Poisson}(\mu)$

Distribution Poisson avec paramètre λ :

$$p_0 = e^{-\lambda} \qquad p_1 = \lambda e^{-\lambda} \qquad p_2 = \frac{\lambda^2}{2} e^{-\lambda}$$

$$p_k = \frac{\lambda^k}{k!} e^{-\lambda} \qquad \{k = 0, 1, 2, \ldots\}$$

Approximation Poisson

jumeaux astrologiques : $\binom{n}{2}$ événements, $\epsilon = 1/365$ probabilité

$$p_0 \approx \exp\Bigl(-\frac{n(n-1)}{2}\epsilon\Bigr),$$

triples astrologiques : $\binom{n}{3}$ événements, $\epsilon = 1/365^2$ probabilité

avions (exemple de David Aldous) : 3 catastrophes en 8 jours

⋆ Malaysian Airlines 17 juillet

⋆ TransAsia 23 juillet

⋆ Air Algerie 24 juillet

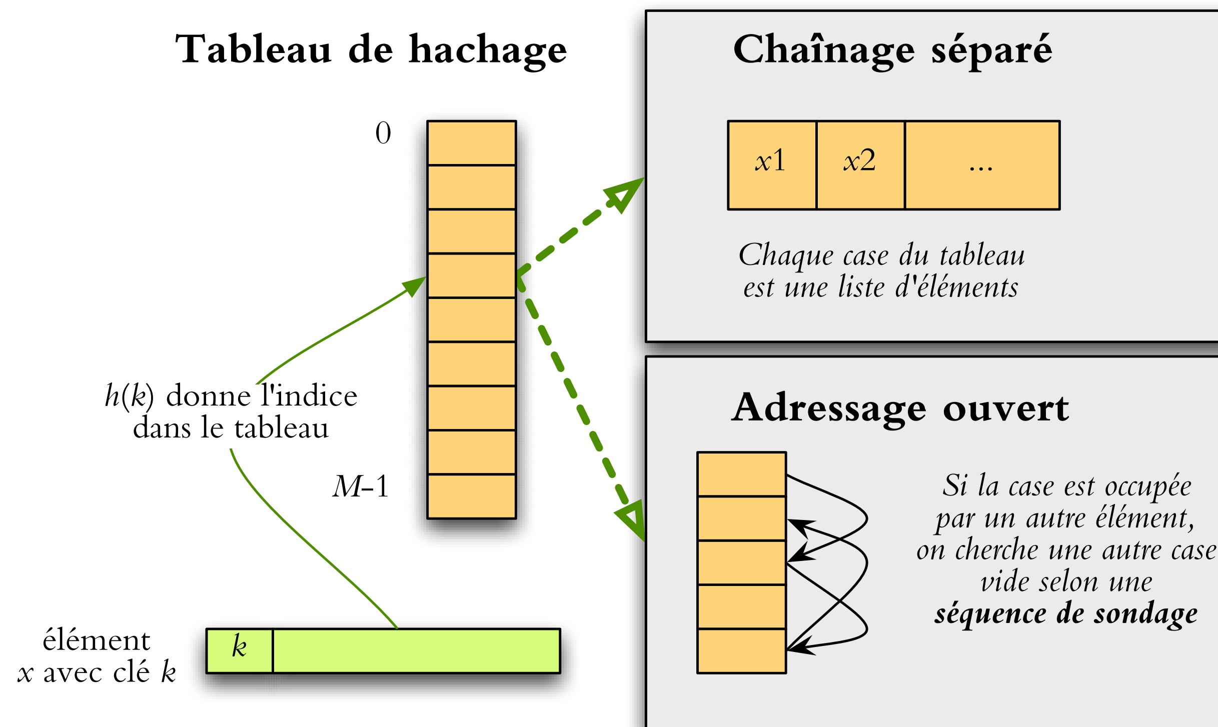
historiquement : 1 catastrophe par 40 jours

$\mathbb{P}\{N = 3\} \approx e^{-8/40}\frac{(8/40)^3}{3!} \approx 1.1 \cdot 10^{-3}$

Hachage

1. chaînage séparé : stocker une liste pour chaque valeur de hachage

2. adressage ouvert : séquence de sondage



Chaînage séparé

implémentation par défaut en Java : solution générique

encapsulation : nœuds avec clé, valeur, prochain nœud sur la même liste

(java.util.HashMap : openjdk 6b14)

```
public class HashMap<K,V> ...
{
   static class Entry<K,V>
   {
      final K key;
      V value;
      Entry<K,V> next;
      final int hash; // hashcode stocké ici

      Entry(int h, K k, V v, Entry<K,V> n)
      {
         value = v; next = n; key = k; hash = h;
      }
      ...
```

HashMap

```
public class HashMap<K,V> extends AbstractMap<K,V>
                          implements Map<K,V>, ...
{
   Entry[] table;
   int size;
   final float loadFactor;
   ...
   public HashMap(int initialCapacity, float loadFactor)
   {
      // Find a power of 2 >= initialCapacity
      int capacity = 1;
      while (capacity < initialCapacity) capacity <<= 1;

      this.loadFactor = loadFactor;
      this.table = new Entry[capacity];
      this.size = 0;
      ...
   }
```

Chaînage séparé – recherche

fonction de hachage → indice dans le tableau → parcours de la liste qui y débute

```
public V get(Object key)
{
   int hash = hash(key.hashCode());
   int i = indexFor(hash, table.length);
   for (Entry<K,V> e = table[i]; e != null; e = e.next)
   {
      Object k;
      if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
         return e.value; // fructueuse :)
   }
   return null; // infructueuse :(
}
static int hash(int h) { // mixer les bits mieux
   h ^= (h >>> 20) ^ (h >>> 12);
   return h ^ (h >>> 7) ^ (h >>> 4);
}
static int indexFor(int h, int length) {
   return h & (length-1); // length est toujours 2^p ici
}
```

Chaînage séparé – insertion

fonction de hachage → indice dans le tableau → insertion/remplacement

```
public V put(K key, V value)
{
   int hash = hash(key.hashCode());
   int i = indexFor(hash, table.length);
   for (Entry<K,V> e = table[i]; e != null; e = e.next)
   {
      Object k;
      if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
      { // clé déjà là!
         V oldValue = e.value; e.value = value;
         return oldValue;
      }
   }
   addEntry(hash, key, value, i);
   return null;
}
```

Chaînage séparé – insertion

nouvelle clé : insertion à la tête

```
void addEntry(int hash, K key, V value, int bucketIndex)
{
   Entry<K,V> e = table[bucketIndex];
   table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
   if (size++ >= threshold) resize(2 * table.length);
}
```

Chaînage séparé : efficacité

Tableau de hachage : taille M, pour stocker n éléments

facteur de remplissage (*load factor*) $\alpha = n/M$ — on permet $\alpha > 1$

Liste chaînée — performance : longueur d'une liste en moyenne est α

nombre de nœuds visités pour recherche fructueuse $(1 + \alpha)/2$ en moyenne

nombre de nœuds visités pour recherche infructueuse $(1 + \alpha)$

insertion : $O(1)$ so pas de recherche nécessaire (on est sûr que c'est une nouvelle clé — insertion à la tête de la liste sans parcours)

suppression : comme recherche fructueuse

Adressage ouvert

Sondage d'une séquence de positions : dépend de la clé à insérer

On essaie les cases $h_0(k)$, $h_1(k)$, ...

Avec une fonction f : $h_i(k) = h(k) + f(i,k) \bmod M$

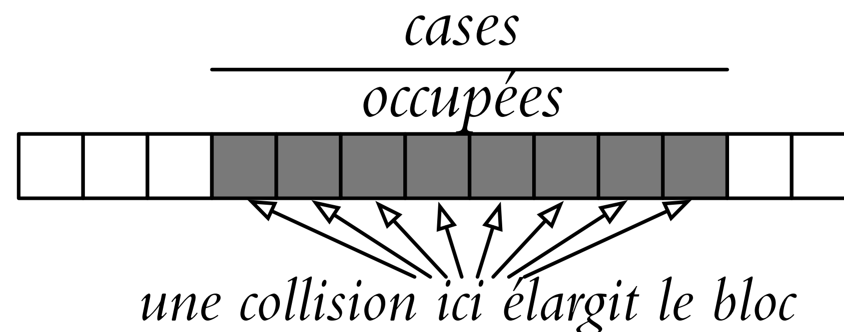
f : stratégie de résolution de collision

Méthodes :
– sondage linéaire $f(i,k) = i$
– double hachage $f(i,k) = ih'(k)$ avec fonction de hachage auxiliaire h'
Ne permet pas $\alpha \geq 1$

Sondage linéaire

(*Linear probing*) : $h(k), h(k)+1, h(k)+2, \ldots$

Problème : grappe forte (*primary clustering*) — blocs de cases occupées



Nombre moyen de sondages lors de recherche fructueuse : $\approx \frac{1}{2}\Bigl(1 + \frac{1}{1-\alpha}\Bigr)$

sondages lors de recherche infructueuse ou insertion : $\approx \frac{1}{2}\Bigl(1 + \frac{1}{(1-\alpha)^2}\Bigr)$

$\Rightarrow$ utiliser jusqu'à $\alpha < 0.75$ (ou un autre seuil sympa < 1)

Sondage linéaire / Java

```
public class LinearProbing extends AbstractSet<Object> {
    private static final int HASH_MULTIPLY =
            1 | (int)((3.0-Math.sqrt(5))*(1<<31)); // <0 mais OK
    private Object[] table;
    private int size;
    private int capacity_bits;

    public LinearProbing(int initial_capacity)
    {
        int b = 1; int cap = 1<<b;
        while (cap<initial_capacity){cap = cap*2;b++;}
        this.table = new Object[cap];
        this.capacity_bits = b;
    }

    private int getTableIndex(int x)
    {
        int idx = (HASH_MULTIPLY*x) >>> (32-capacity_bits);
        return idx;
    }
```

`AbstractSet` : il reste à implémenter les méthodes add et iterator

Sondage linéaire / search

```
// class LinearProbing
    /**
     * Search for an element.
     *
     * @param key query
     * @return index in the table where found,
     *         or where it should be placed on insertion
     */
    private int search(Object key)
    {
        int h = key.hashCode();
        int i = getTableIndex(h);
        while (table[i]!= null && !table[i].equals(key))
            i=(i+1) % table.length;
        return i;
    }
```

Sondage linéaire / insert

```
// class LinearProbing
    @Override
    public boolean add(Object emt)
    {
        if (emt==null) throw new UnsupportedOperationException();
        int i = search(emt);
        if (table[i]==null)
        {
            table[i]=emt; ++size;
            if (loadFactor()>max_load_factor) rehash();
            return true;
        } else
            return false;
    }
```

`add` retourne **false** si insertion n'est pas possible (clé répétée)

Sondage linéaire / rehash

si la facteur de remplissage est atteint, il fait réallouer un autre tableau + copier les éléments selon une nouvelle fonction de hachage

```
// class LinearProbing
    private void rehash()
    {
        int newcapbits = this.capacity_bits+1;
        Object[] old_table = this.table;
        this.table = new Object[1<<newcapbits];
        this.capacity_bits = newcapbits;
        for (int i=0; i<old_table.length; i++)
        {
            if (old_table[i] != null)
            {
                Object E = old_table[i];
                table[ search(E)] = E;
            }
        }
    }
```

Double hachage

Généralisation de sondage linéaire : $h(k), h(k)+c, h(k)+2c, h(k)+3c, \ldots$
c dépend de la clé k : $c = h'(k)$

Exemples (éviter $c = 0$!) : $h'(x) = 1 + x \bmod M'$ avec $M' < M$
$h'(x) = M' - (x \bmod M')$

Très proche d'une résolution idéale

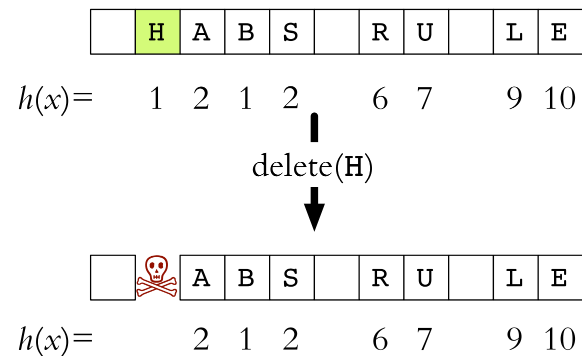
Nombre moyen de sondages lors d'insertion ou recherche infructueuse :
$\approx 1 + \alpha + \alpha^2 + \cdots \approx \frac{1}{1-\alpha}$

sondages lors de recherche fructueuse :
$\approx \frac{1}{n} \sum_{i=0}^{n-1} \frac{1}{1-i/M} \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

Lazy deletion

suppression avec adressage ouvert : pas trivial

☞utiliser suppression paresseuse (*lazy deletion*) : marquer la case «supprimée»
remplacer l'élément supprimé par une sentinelle DELETED

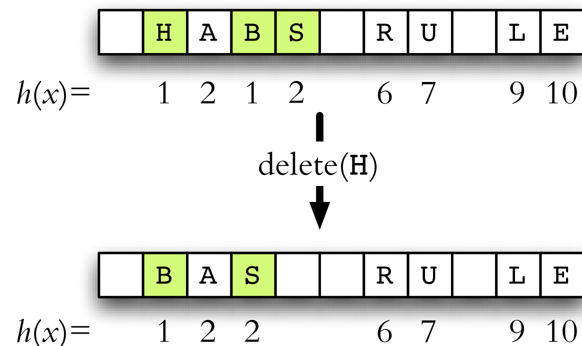


search doit passer par les sentinelles
insert peut recycler la case avec DELETED
⇒ facteur de remplissage inclut les éléments supprimés

Suppression impatiente

☞ utiliser suppression impatiente (*eager deletion*) : remplir le «trou» en réinsérant les éléments dans le même bloc de cases occupées

```
    delete(x) // suppression de la clé x
S1  i ← h(x) ; while H[i] ≠ x && H[i] ≠ null do i ← (i + 1) mod M
S2  if H[i] = x then // sinon x inconnue
S3      H[i] ← null; i ← (i + 1) mod M
S4      while H[i] ≠ null do // éléments à déplacer
S5          y ← H[i]; H[i] ← null; insert(y)
S6          i ← (i + 1) mod M
```



Java hashCode()

hashcode : utilisé dans les tableaux de hachage

```
class Object
{
   public native int hashCode(); // redéfinir à volonté
   ...
```

contrat : si deux objets sont égaux par `equals`, alors ils doivent avoir le même `hashcode`

- ⋆ *Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.*
- ⋆ *If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.*

Java equals()

```
class Object
{
   public boolean equals(Object obj) // redéfinir à volonté
   {
      return this==obj;
   }
```

contrat : rélation d'équivalence entre références non-null

- ⋆ *It is reflexive : for any non-null reference value x, x.equals(x) should return true.*
- ⋆ *It is symmetric : for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.*
- ⋆ *It is transitive : for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.*
- ⋆ *It is consistent : for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.*
- ⋆ *For any non-null reference value x, x.equals(null) should return false.*

Java hashCode+equals

```
public class Integer extends Number implements Comparable<Integer>
{
   private final int value;
   public Integer(int value){ this.value = value;}
   @Override
   public boolean equals(Object o)
   {
      if (o instanceof Integer)
      {
         return value == ((Integer)o).intValue();
      }
      return false;
   }
   @Override
   public int hashCode()
   {
      return Integer.hashCode(value);
   }
   public static int hashCode(int value){ return value;}
```

Java hashCode+equals

recette pour hashcode : mixer les bits pour les valeurs examinées dans equals

```
public class Point2D
{
   @Override
   public boolean equals(Object obj)
   {
      if (obj instanceof Point2D) // si même classe
      {
         Point2D p2d = (Point2D) obj;
         return (getX() == p2d.getX()) && (getY() == p2d.getY());
      }
      return super.equals(obj); // sinon déléguer à la superclasse
   }

   @Override
   public int hashCode()
   {
      long bits = java.lang.Double.doubleToLongBits(getX());
      bits ^= java.lang.Double.doubleToLongBits(getY()) * 31;
      return (((int) bits) ^ ((int) (bits >> 32)));
   }
```

Conclusion

- ⋆ tableau de hachage : structure simple à implémenter (chaînage séparée ou sondage linéaire)
- ⋆ performance excellente si facteur de remplissage reste bornée : $O(1)$ pour toute opération **en moyenne**
- ⋆ performance horrible au pire : $\Theta(n)$ si collision totale