L3. Programmation orientée objet. Cours 11

Marie-Pierre Béal (Cours de Cyril Nicaud)

Compléments : Algorithmique et structures de données en Java. Hachage.

Algorithmique et structures de données en Java.

Hachage

Requêtes sur une structure de données non mutables

- Une *structure de données* est une façon d'organiser la mémoire pour y représenter un ensemble de données
- Une structure de donnée est *non mutable* quand elle n'est pas modifiée après son initialisation
- Une *requête typique* sur ce genre de structure consiste à parcourir ses éléments pour : chercher un élément, ...
- Si on considère que ses éléments sont de plus *ordonnés*, on peut également demander accès au i-ème élément dans l'ordre : c'est le cas pour les tableaux et les listes.

Requêtes sur une structure de données non mutables

On va se focaliser pour le moment sur les deux requêtes suivantes :

- `list.contains(x)` : Est-ce que l'objet `x` est dans `list` ?
- `list.get(i)` : Quel est l'objet en position `i` dans `list` ?

où `list` est une liste tableau non mutable.

	`contains`	`get(i)`
liste tableau	$\Theta(n)$	$\Theta(1)$

Et si la liste tableau est mutable ?

Nouveau cahier des charges

On veut une structure ordonnée `list` avec les requêtes suivantes :

- `list.contains(x)` : Est-ce que l'objet `x` est dans `list` ?
- `list.get(i)` : Quel est l'objet en position `i` dans `list` ?
- `list.add(x)` : Ajoute `x` à la fin de `list`
- `list.removeLast()` : Supprime le dernier élément de `list`

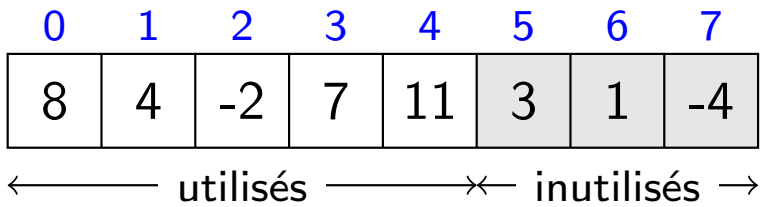
où `list` est une liste tableau mutable, une `ArrayList`

C'est une structure de données *dynamique* : elle évolue au cours du temps avec les ajouts/suppressions.

Structures de données mutables

Regardons pour les `ArrayList`

- Si on un tableau est assez grand (qui n'a pas à être étendu)




```
capacity=8
n=5
```

- Les complexités sont encore en $\Theta(1)$, sauf `contains(x)` en $\Theta(n)$

Structures de données mutables

Regardons pour les `ArrayList`

- Si on a plus assez de place, on *double* la capacité.
- Il faut *ré-allouer* un tableau et recopier les éléments.
- Problème : la complexité devient $\Theta(n)$ quand on déclenche de doublement de la capacité.

Complexité amortie

Définition version Θ. Un algorithme est de *complexité amortie* $\Theta(t_n)$ quand n appels à cet algorithme se font en temps total $\Theta(n \times t_n)$

- On ne regarde pas un seul appel à l'algorithme
- On calcule la **complexité cumulée** de n appels
- On divise par n le résultat pour avoir la **complexité amortie**

$\rightarrow$ *Cela permet de quantifier qu'il ne peut pas y avoir beaucoup de pires cas consécutivement*

Définition version O. Un algorithme est de *complexité amortie* $O(t_n)$ quand n appels à cet algorithme se font en temps total $O(n \times t_n)$

Complexité amortie et tableaux dynamiques

Définition version Θ. Un algorithme est de *complexité amortie* $\Theta(t_n)$ quand n appels à cet algorithme se font en temps total $\Theta(n \times t_n)$

Théorème. L'ajout à la fin d'un tableau dynamique, à partir d'un tableau vide, a une complexité amortie en $\Theta(1)$.

	`contains(x)`	`get(i)`	`add`
tableaux dynamiques	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$ amortie

C'est une structure de données très utilisée :

- ArrayList en java
- listes/tableaux de python

Complément : preuve du théorème

Théorème. L'ajout à la fin d'un tableau dynamique, à partir d'un tableau vide, a une complexité amortie en $\Theta(1)$.

Preuve. On compte le nombre d'écritures dans un tableau, qui correspond bien aux lignes les plus effectuées. On a deux cas selon la taille t du tableau :

- si $t = 2^k$ est une puissance de 2, alors redimensionner le tableau et recopier les valeurs coûte t écritures, plus 1 pour la nouvelle valeur
- sinon, cela coûte juste 1 écriture

Si on compte le nombre E_n d'écritures pour n insertions, on a donc

$$E_n = n + \sum_{k=0}^{m} 2^k, \text{ où } m = \lfloor \log_2(n) \rfloor$$

Comme $\sum_{k=0}^{m} 2^k = 2^{m+1} - 1 \leq 2 \times 2^m \leq 2n$, on a $E_n \leq 3n$
On a aussi $E_n \geq n$, et donc la complexité de n insertions est en $\Theta(n)$.

On a bien montré que la complexité amortie de l'insertion est en $\Theta(1)$

Sets & Maps

Cahier des charges pour représenter des *ensembles* :

- `new XXX<>()` (on prendra `new HashSet<>()`) : initialise un ensemble vide
- `set.add(x)` : ajoute `x` à l'ensemble `set` s'il n'y est pas déjà (pas de doublon dans un ensemble)
- `set.contains(x)` : teste si `x` est dans `set`
- `set.remove(x)` : enlève `x` de l'ensemble `set`.

La structure n'est pas indexée : pas de i-ème élément

Cahier des charges pour les maps

Le cahier des charges pour représenter des maps :

- `new XXX<>()` (on prendra `new HashMap<>()`) : initialise une map sans clé
- `map.put(k, v)` : attribue la valeur `v` à la clé `k` dans la map. S'il y avait déjà une valeur attribuée à la clé k, elle est changée
- `map.get(k)` : renvoie la valeur associée à la clé `k` dans `map`
- `map.contains(k)` : teste si la clé `k` est dans `map`
- `map.remove(k)` : enlève la clé `k` (et la valeur associée) de `map`

La structure n'est pas ordonnée : pas de i-ème élément

Tableaux dynamiques/ HashMap

Si on cherche tous les mots différents d'un texte on utilisant des `ArrayList`, le temps de calcul va être élevé.

Le temps mis sur un roman de 524 kilo-octets est environ 2 secondes.

Pire encore si on veut compter le nombre de fois qu'apparaît chaque mot dans un texte avec des `ArrayList` de couple `(k, v)`.

Pour le roman *La Peste*, cela prend près de 32 secondes !

Avec une `HashMap`, au lieu de 32 secondes, cela prend maintenant 0.02 secondes !

Hachage

Principe pour les ensembles

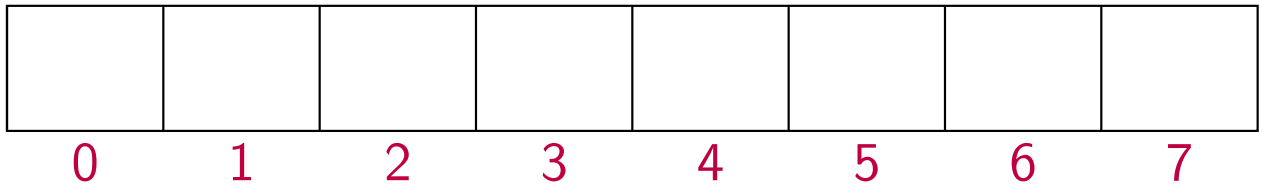
L'idée est d'utiliser

- un **tableau** T, de longueur m, appelé la *table*
- une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la *fonction de hachage*

On souhaite implanter le cahier des charges de la façon suivante :

- pour **ajouter** la clé x, on la met dans la case $T[h(x)]$
- pour **chercher** si x est dans T, on regarde dans la case $T[h(x)]$
- pour **supprimer** la clé x, on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \mod m$ (quand on a une table de taille m)



Principe pour les ensembles

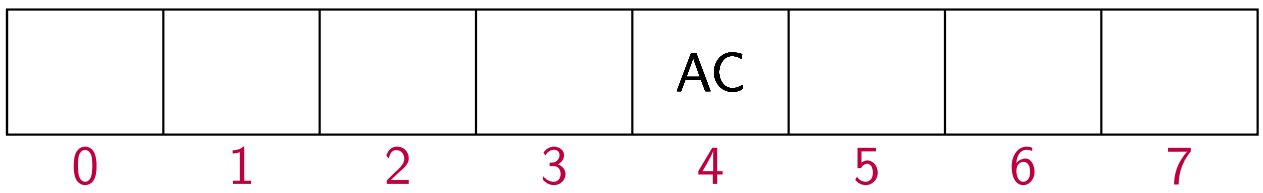
L'idée est d'utiliser

- un **tableau** T, de longueur m, appelé la *table*
- une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la *fonction de hachage*

On souhaite implanter le cahier des charges de la façon suivante :

- pour **ajouter** la clé x, on la met dans la case $T[h(x)]$
- pour **chercher** si x est dans T, on regarde dans la case $T[h(x)]$
- pour **supprimer** la clé x, on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \mod m$ (quand on a une table de taille m)



Eléments ajoutés : "AC" (4),

Principe pour les ensembles

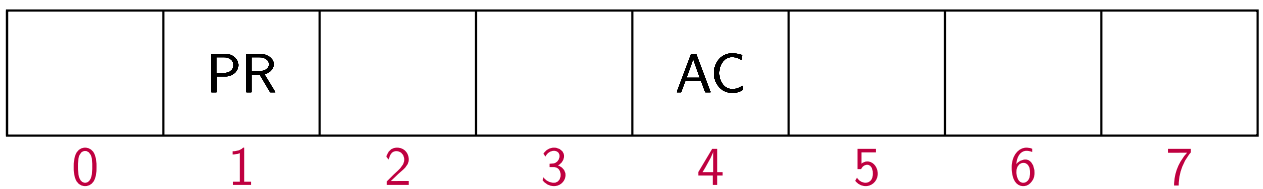
L'idée est d'utiliser

- un **tableau** T, de longueur m, appelé la *table*
- une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la *fonction de hachage*

On souhaite implanter le cahier des charges de la façon suivante :

- pour **ajouter** la clé x, on la met dans la case $T[h(x)]$
- pour **chercher** si x est dans T, on regarde dans la case $T[h(x)]$
- pour **supprimer** la clé x, on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \mod m$ (quand on a une table de taille m)



Eléments ajoutés : "AC" (4), "PR" (1),

Principe pour les ensembles

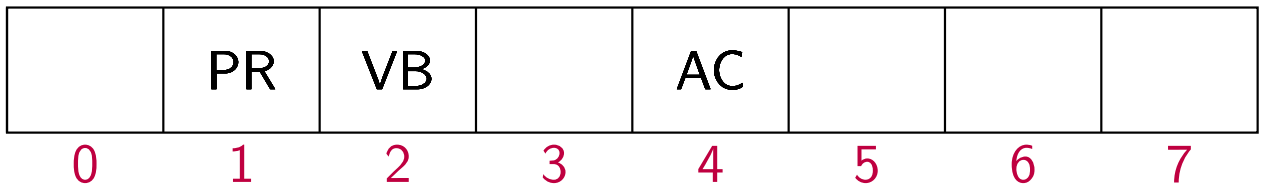
L'idée est d'utiliser

- un **tableau** T, de longueur m, appelé la *table*
- une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la *fonction de hachage*

On souhaite implanter le cahier des charges de la façon suivante :

- pour **ajouter** la clé x, on la met dans la case $T[h(x)]$
- pour **chercher** si x est dans T, on regarde dans la case $T[h(x)]$
- pour **supprimer** la clé x, on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \mod m$ (quand on a une table de taille m)



Eléments ajoutés : "AC" (4), "PR" (1), "VB" (10),

Principe pour les ensembles

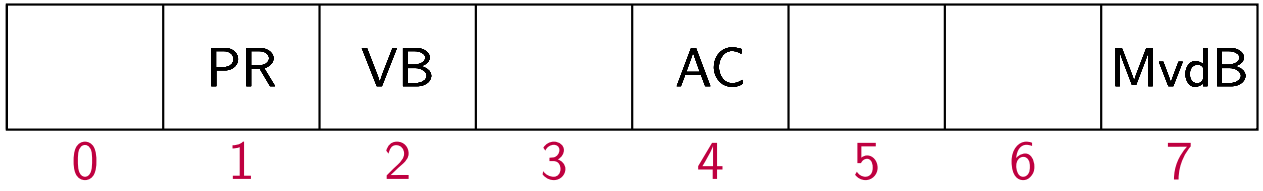
L'idée est d'utiliser

- un **tableau** T, de longueur m, appelé la *table*
- une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la *fonction de hachage*

On souhaite implanter le cahier des charges de la façon suivante :

- pour **ajouter** la clé x, on la met dans la case $T[h(x)]$
- pour **chercher** si x est dans T, on regarde dans la case $T[h(x)]$
- pour **supprimer** la clé x, on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \mod m$ (quand on a une table de taille m)



Eléments ajoutés : "AC" (4), "PR" (1), "VB" (10), "MvdB" (15),

Principe pour les ensembles

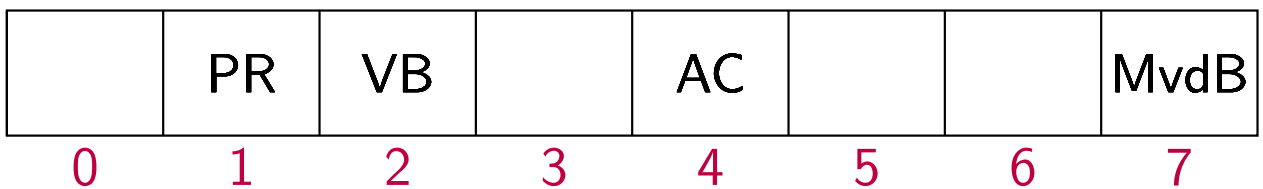
L'idée est d'utiliser

- un **tableau** T, de longueur m, appelé la *table*
- une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la *fonction de hachage*

On souhaite implanter le cahier des charges de la façon suivante :

- pour **ajouter** la clé x, on la met dans la case $T[h(x)]$
- pour **chercher** si x est dans T, on regarde dans la case $T[h(x)]$
- pour **supprimer** la clé x, on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \mod m$ (quand on a une table de taille m)



"PR" $\in T$: on calcule $h(PR) = 1$ et on le trouve en $T[1]$

Principe pour les ensembles

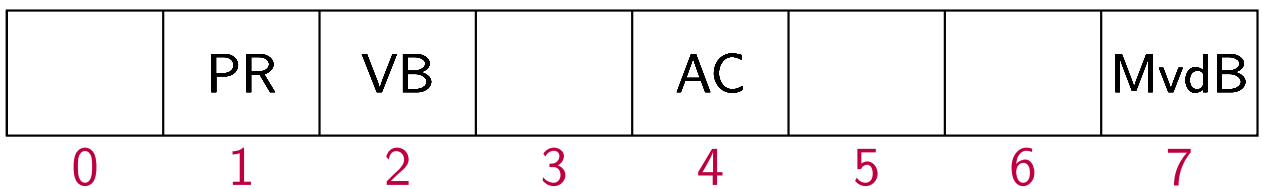
L'idée est d'utiliser

- un **tableau** T, de longueur m, appelé la *table*
- une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la *fonction de hachage*

On souhaite implanter le cahier des charges de la façon suivante :

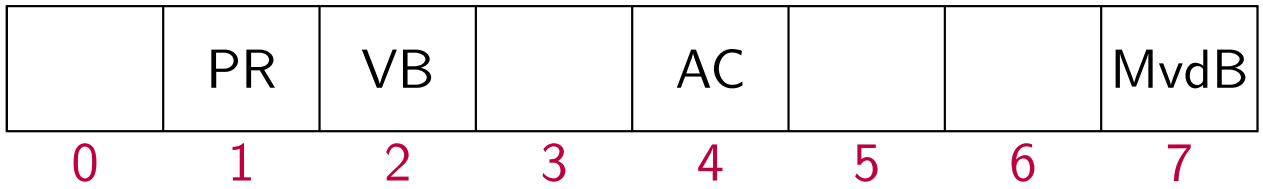
- pour **ajouter** la clé x, on la met dans la case $T[h(x)]$
- pour **chercher** si x est dans T, on regarde dans la case $T[h(x)]$
- pour **supprimer** la clé x, on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \mod m$ (quand on a une table de taille m)



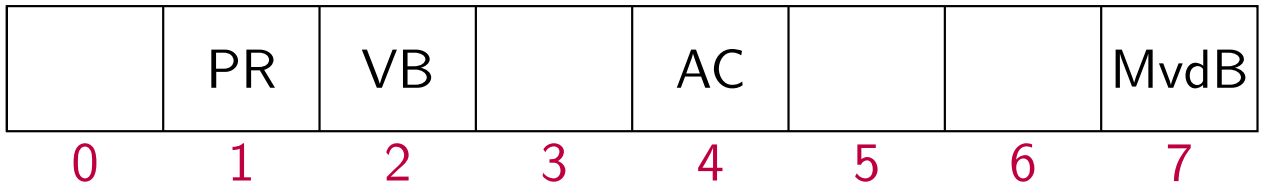
"GR" $\notin T$: on calcule $h(GR) = 11$ et il n'est pas en $T[3]$

Trois points à régler



- Que faire si on veut ajouter dans une case déjà occupée ? Par exemple la clé "CN" avec $h(CN) = 4$?
- Comment choisir la fonction de hachage h ?
- Comment choisir la taille m de la table T ?

Trois points à régler



- Que faire si on veut ajouter dans une case déjà occupée ? Par exemple la clé "CN" avec $h(CN) = 4$?
- Comment choisir la fonction de hachage h ?
- Comment choisir la taille m de la table T ?

Gestion des collisions

Définition. On dit qu'il y a une *collision* quand deux clés sont envoyées sur la même case de la table. Mathématiquement : $h(x) \equiv h(y) \mod m$ pour deux clés $x \neq y$

Il existe deux façons principales de gérer les collisions

- **Externe :** chaque case contient une liste chaînée des clés.
- **Interne :** si une autre clé est déjà présente, on insère ailleurs dans la table (un peu plus loin)

python utilise du hachage interne, java du hachage externe

Et les maps ?

Question : On n'a vu que les **sets**, peut-on utiliser les **tables de hachages** pour les **maps** ?

OUI ! il suffit de

- Stocker des paires $[k, v]$, où k est la clé et v la valeur
- Manipuler les fonctions de hachage uniquement sur les clés k
- . . .et ça fonctionne pareil

	PR 17	VB -8		AC 7			MvdB 9
0	1	2	3	4	5	6	7

$$h(AC) = 4 \quad h(PR) = 1 \quad h(VB) = 10 \quad h(MvdB) = 15$$

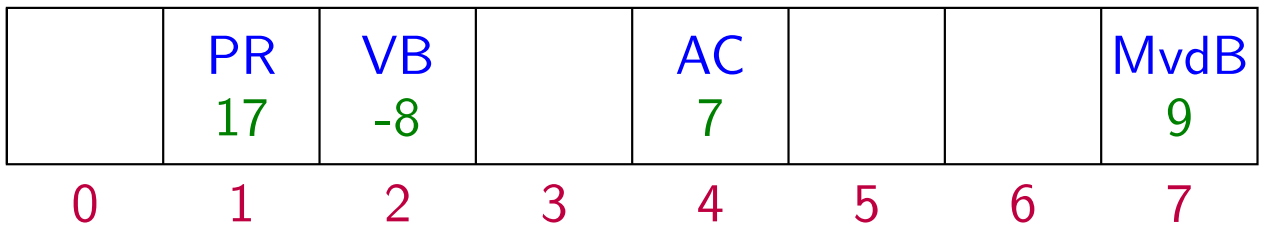
- Pareil pour le **hachage externe**, on stocke les paires $[k, v]$ dans les listes chaînées ou dans les tableaux dynamiques

Et les maps ?

Question : On n'a vu que les **sets**, peut-on utiliser les **tables de hachages** pour les **maps** ?

OUI ! il suffit de

- Stocker des paires $[k, v]$, où k est la clé et v la valeur
- Manipuler les fonctions de hachage uniquement sur les clés k
- . . .et ça fonctionne pareil



$$h(AC) = 4 \quad h(PR) = 1 \quad h(VB) = 10 \quad h(MvdB) = 15$$

- Pareil pour le **hachage externe**, on stocke les paires $[k, v]$ dans les listes chaînées ou dans les tableaux dynamiques

Table de hachage externe en Java : les HashSet

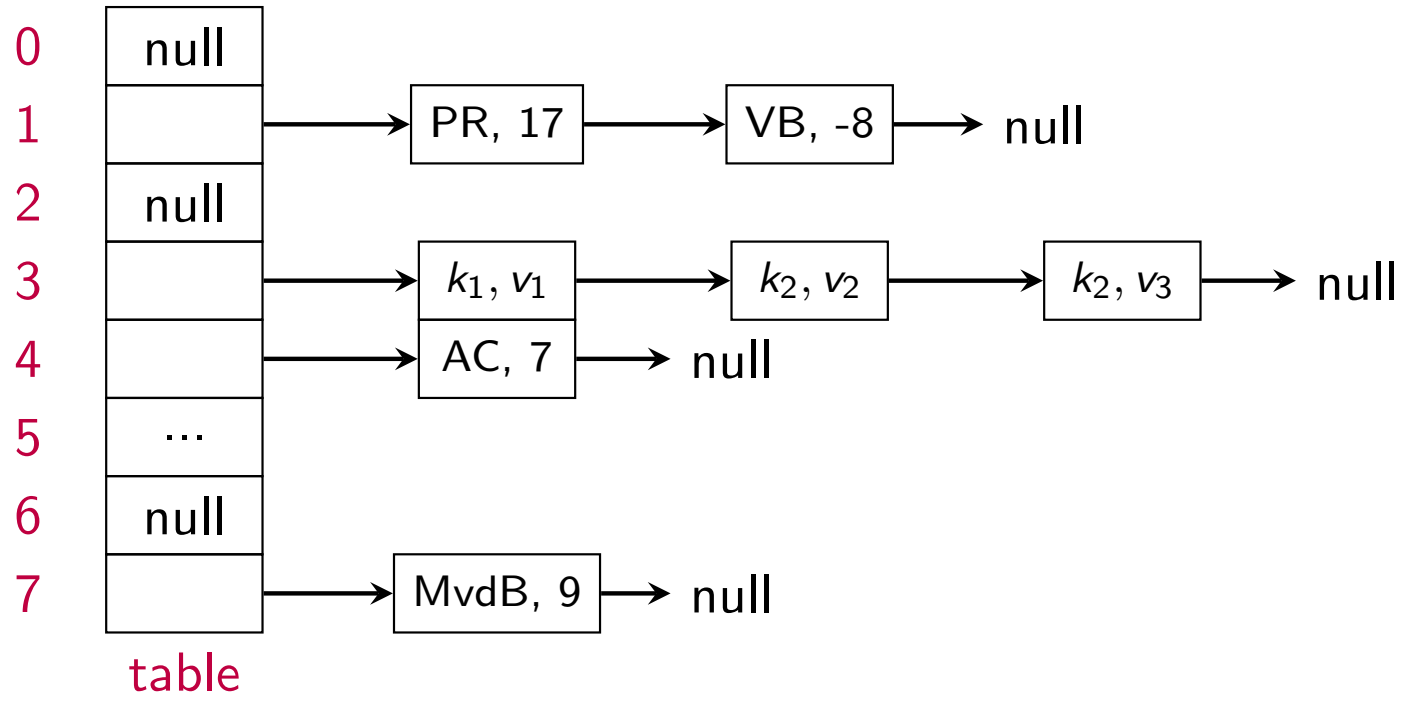
En fait, en Java un HashSet est implémenté avec une HashMap

```
public class HashSet<E> ... {
  HashMap<E,Object> map;
  // Dummy value to associate with an Object in the backing Map
  static final Object PRESENT = new Object();

  public HashSet() {
    map = new HashMap<>();
  }
  public boolean contains(Object o) {
    return map.containsKey(o);
  }
  public boolean add(E e) {
    return map.put(e, PRESENT)==null;
  }
  public boolean remove(Object o) {
    return map.remove(o)==PRESENT;
  }
}
```

Table de hachage externe en Java : les `HashMap`

$(AC, 7)$ $hash(AC) = 4$ $(PR, 17)$ $hash(PR) = 1$ $(VB, -8)$ $hash(VB) = 1$ $(MvdB, 9)$ $hash(MvdB) = 7$



`table[i]` : liste chaînée des couples `(k, v)` tels que `hash(k) = i`.

Table de hachage externe en Java : les `HashMap`

Le opérations principales sur une `HashMap` sont

- `map.get(k)`. On calcule `i = hash(k)`. On parcourt la liste `table[i]` pour chercher la clé `k` dans cette liste. Si on la trouve on renvoie a valeur associée à `k`
- `map.put(k, v)`. On calcule `i = hash(k)`. On parcourt la liste `table[i]` pour chercher la clé `k` dans cette liste. Si on la trouve on remplace sa valeur associée par `v`. Sinon on ajoute le couple `(k, v)` à la fin de la liste.

`https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/HashMap.java#L281`

Table de hachage externe en Java : les HashMap

```
public class HashMap<K,V> ...{

  static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
  }

  static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
    ...
  }
  Node<K,V>[] table;
```

Table de hachage externe en Java : les HashMap

```
public V get(Object key) {
  Node<K,V> e;
  return (e = getNode(key)) == null ? null : e.value;
}
final Node<K,V> getNode(Object key) {
  Node<K,V>[] tab; Node<K,V> first, e; int m, hash; K k;
  if ((tab = table) != null && (m = tab.length) > 0 &&
      (first = tab[(m - 1) & (hash = hash(key))]) != null) {
    if (first.hash == hash && // always check first node
       ((k = first.key) == key || (key != null && key.equals(k))))
      return first;
    if ((e = first.next) != null) {
      ...
      do {
         if (e.hash == hash &&
           ((k = e.key) == key || (key != null && key.equals(k))))
           return e;
      } while ((e = e.next) != null);
     }
   }
  return null;
} // version simplifiée
```

Table de hachage externe en Java : les HashMap

```
public V put(K key, V value) {
  return putVal(hash(key), key, value, false, true);
}
V putVal(int hash, K key, V value, ...) {
  Node<K,V>[] tab; Node<K,V> p; int m, i;
  if ((tab = table) == null || (m = tab.length) == 0)
    m = (tab = resize()).length;
  if ((p = tab[i = (m - 1) & hash]) == null)
    tab[i] = newNode(hash, key, value, null);
  else {
    Node<K,V> e; K k;
    if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
      e = p;
    else {
      for (int binCount = 0; ; ++binCount) {
        ...
       }
     }
    ...
  }
  if (++size > threshold) resize();
  return null;
} // version simplifiée
```

Table de hachage externe en Java : les HashMap

```
V putVal(int hash, K key, V value, ...) {
  ...
    else {
      for (int binCount = 0; ; ++binCount) {
        if ((e = p.next) == null) { // la clé n'est pas trouvée on ajoute au bout
          p.next = newNode(hash, key, value, null);
          break;
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k
          break;
        p = e;
       }
     } // si la clé est trouvée on renvoie l'ancienne valeur associée
     if (e != null) {
        V oldValue = e.value;
        ...
        return oldValue;
  }
  ...
} // version simplifiée
```

Table de hachage externe en Java

- On calcule `hash := hash(key) mod m` et une opération effectue le parcours de la liste chaînée `table[hash]`
- Les complexités (sauf la construction de `table`) sont en $\Theta(\ell)$, où ℓ est la taille de la liste chaînée `table[hash]`
- Il est très important que `hash(key)` donne toujours la même valeur (quand on l'utilise dans une `put` ou dans un `get`). Comme `hash(key)` est calculé à partir du `hashCode` de la clé, les clés doivent être non mutables.

Fonction de hachage

Ce qu'on veut pour la fonction de hachage :

- Qu'elle **répartisse bien** les clés : deux clés différentes, même très similaires doivent avoir des valeurs de hachage très différentes
- Qu'elle se **calcule rapidement** : on va avoir besoin de calculer la valeur de hachage souvent

$\Rightarrow$ Idéalement, on voudrait que $h(x) \mod m$ soit un entier au hasard (uniforme) de $\{0, \ldots, m-1\}$, mais on ne peut pas utiliser le hasard, car il faut que $h(x)$ soit constant (qu'on puisse retrouver x après)

On veut donc une *fonction de hachage* qui soit **rapide à calculer** et qui se comporte **comme si on avait tiré au sort** chaque valeur, mais déterministe (sans effectuer de tirage au sort pour avoir toujours le même résultat) c'est compliqué !

Règle d'or : ne faîtes pas vous-même vos fonctions de hachage, utilisez celle intégrées dans le langage ou trouvées dans des livres

Fonction de hachage

Ce qu'on veut pour la fonction de hachage :

- Qu'elle **répartisse bien** les clés : deux clés différentes, même très similaires doivent avoir des valeurs de hachage très différentes
- Qu'elle se **calcule rapidement** : on va avoir besoin de calculer la valeur de hachage souvent

$\Rightarrow$ Idéalement, on voudrait que $h(x) \mod m$ soit un entier au hasard (uniforme) de $\{0, \ldots, m-1\}$, mais on ne peut pas utiliser le hasard, car il faut que $h(x)$ soit constant (qu'on puisse retrouver x après)

On veut donc une *fonction de hachage* qui soit **rapide à calculer** et qui se comporte **comme si on avait tiré au sort** chaque valeur, mais déterministe (sans effectuer de tirage au sort pour avoir toujours le même résultat) c'est compliqué !

Règle d'or : ne faîtes pas vous-même vos fonctions de hachage, utilisez celle intégrées dans le langage ou trouvées dans des livres

Complexité des opérations (hachage externe)

On utilise n pour **le nombre de clés** dans la table de hachage, et m pour la **taille de la table**.

Pour le *hachage externe* :

- Dans le **pire cas**, toutes les clés ont la même valeur de hachage, les opérations (hors création de la table) sont en $\Theta(n)$, on travaille avec une seule liste $\leftarrow$ le pire cas n'est pas pertinent ici
- Une fonction de hachage **idéale** est modélisée par des valeurs aléatoires et avec ce modèle on obtient que les opérations sont en $\Theta(\frac{n}{m})$ **en moyenne**

Théorème. Si $\alpha := \frac{n}{m}$ est borné par une constante et que l'on ne redimensionne pas la table, alors les opérations dans une table de **hachage externe** ont une complexité moyenne $\Theta(1)$.

C'est pour ça que c'est très efficace en pratique

Complexité des opérations (hachage externe)

On utilise n pour **le nombre de clés** dans la table de hachage, et m pour la **taille de la table**.

Pour le *hachage externe* :

- Dans le **pire cas**, toutes les clés ont la même valeur de hachage, les opérations (hors création de la table) sont en $\Theta(n)$, on travaille avec une seule liste $\leftarrow$ le pire cas n'est pas pertinent ici
- Une fonction de hachage **idéale** est modélisée par des valeurs aléatoires et avec ce modèle on obtient que les opérations sont en $\Theta(\frac{n}{m})$ **en moyenne**

Théorème. Si $\alpha := \frac{n}{m}$ est borné par une constante et que l'on ne redimensionne pas la table, alors les opérations dans une table de **hachage externe** ont une complexité moyenne $\Theta(1)$.

C'est pour ça que c'est très efficace en pratique

Bilan complexité et choix de m

- On **redimensionne la table** si le taux de remplissage $\alpha = \frac{n}{m}$ dépasse un certain seuil α_0 fixé à l'avance, en doublant la taille par exemple
- Comme la complexité pire cas n'est pas pertinente, on modélise la **fonction de hachage** par une **fonction aléatoire**
- On obtient ainsi des complexités en $\Theta(1)$ amortie en moyenne pour les opérations `add`, `remove` et `contains`
- C'est vrai pour le **hachage externe** et le **hachage interne**

En pratique :

- Si on suit la **règle d'or** et donc qu'on utilise des bonnes fonctions de hachage, le modèle aléatoire décrit bien le comportement
- On a des opérations `add`, `remove` et `contains` très efficaces
- Les **table de hachage** sont des structures de données très efficaces et très utilisées

Complexité des opérations (hachage externe)

Théorème. Les opérations dans une table de **hachage externe** ont une complexité amortie en moyenne $\Theta(1)$.

- En pratique on garde $\alpha \leq \alpha_0$ avec $\alpha_0 \in [0.66, 5]$ pour le **hachage externe**
- Si, en ajoutant des clés, α devient supérieur à α_0, alors on **redimensionne** la table en doublant m. Et on ré-insère toutes les clés (on est obligé à cause du *mod* m).

Important : précision sur la complexité

Theorem

Les opérations add/put, remove et contains sur les HashSet et HashMap ont une complexité amortie en moyenne $\Theta(1)$.

Attention : ce théorème considère que les opérations élémentaires sur les clés se font en temps constant $\Theta(1)$.

En réalité il faut/faudrait prendre en compte le temps de calcul de la valeur de hachage, qui n'est par exemple pas constant pour une chaîne (même s'il est rapide). Le temps de calcul de la comparaison de deux clés n'est pas non plus constant pour des chaînes.

Pour la véritable complexité on peut dire qu'une opération coûte :

- un appel à la fonction de hachage
- $\Theta(1)$ appels, en amortie et en moyenne, à la fonction de comparaison des clés.

En résumé

Vous devez savoir

- Ce qu'est un tableau dynamique, comment il fonctionne algorithmiquement, notamment le redimensionnement
- Ce qu'est la complexité amortie
- La notion d'ensemble (set) et de fonction (map)
- Comment fonctionnent les tables de hachage externe
- Ce qu'on attend d'une bonne fonction de hachage
- Que les complexités sont en $\Theta(1)$ amortie en moyenne si on utilise le redimensionnement pour garder un α en dessous d'un seuil α_0.